

容器技术系列

Docker 进阶与实战

华为 Docker 实践小组 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Docker 进阶与实战 / 华为 Docker 实践小组著. —北京: 机械工业出版社, 2016.2
(容器技术系列)

ISBN 978-7-111-52339-0

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

中国版本图书馆 CIP 数据核字 (2015) 第 303557 号



Docker 进阶与实战

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: 余 洁

责任校对: 董纪丽

印 刷:

版 次: 2016 年 2 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 16.5

书 号: ISBN 978-7-111-52339-0

定 价: 59.00 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

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

我们这个团队的主业是操作系统内核开发。“太阳底下没有新鲜事”，这句话对于操作系统来说，有着深刻的意义。一个爆红的技术，寻根溯源，你会发现它往往已经在操作系统里潜伏很久。这种例子俯拾皆是。

虚拟化技术的源头可以追溯到 20 世纪 70 年代初期 IBM 的 S370，但直到 2003 年的 SOSOP 会议上一篇关于虚拟化的论文《Xen and the Art of Virtualization》引起广泛关注之后，虚拟化才走上发展的快车道。在软件领域，虚拟化技术把 VMware 打造成 400 亿美元量级的行业明星，又在硬件领域搅动了 CPU、网络、存储等各个市场，迫使市场上的行业领袖做出相应的创新。现在，计算虚拟化、网络虚拟化、存储虚拟化这些概念已经深入人心。

而容器技术也不是全新的概念，系统容器最早可以追溯到 20 世纪 80 年代初期的 chroot；打着轻量级虚拟化旗号的商用软件也是在 21 世纪之初由 Virtuozzo 提出的。但当时这个技术只是在系统管理员的小圈子里口耳相传，不温不火地发展着。直到 2013 年，有一家叫作 dotCloud 的小公司开源了一个叫 Docker 的小项目……

若将 Docker 的核心技术层层剥离开来分析，作为操作系统开发人员，我们是无法理解 Docker 为什么会爆发成为行业里的新星的。因为严格来说，Docker 用的所有关键技术都早已存在：

- ❑ Cgroup (Control Group) 是 Google 在 2006 年启动开发的，算起来也有将近 10 年的历史了。
- ❑ 对于 Namespace，从最早的 Mount namespace 算起，不断迭代到今天，已成为包括 UTS（系统标识）、IPC（进程间通信）、PID（进程标识）、Network（网络设备、IP 地址以及路由表）、User（用户标识）等的技术，可谓洋洋大观。
- ❑ Aufs 的历史可以追溯到 1993 年的 Inheriting File System，虽然 Aufs 没有进入 Linux 主线，但也已经在 Debian、Gentoo 这样的主流发行版中得到应用。

这些“大叔辈”的技术，通过 Docker 引擎的组合，焕发出“小鲜肉”的吸引力。而从另一个方面看，那些在技术和理念上更先进的项目，比如 OSv，反而远没有得到这种众星

捧月般的待遇。

为什么会这样？这个疑问促使我们摘下操作系统开发人员的帽子，带上系统运维人员的帽子，带上应用开发者的帽子，换个角度审视自己从前的工作。

在这个角色转换的过程中，我们得到了很多的收获：

首先，我们代表国内的技术人为 Docker 社区做出了一些贡献，此为收获一。

因为换了一个角度，对这个技术兴起背后的原因有了更深刻的理解，此为收获二。

利用工作之余，将技术经验转化为文字，把容器技术传播给更广泛的受众，此为收获三。

如果读者在阅读本书和实践后，不仅知其然，而且知其所以然，并与我们一同把容器技术的发展推向下一个阶段，那可以算是最大的收获了。

是为序！

华为 2012 实验室 操作系统专家 胡欣蔚

2015 年 11 月



为什么要写这本书

在计算机技术日新月异的今天，Docker 也算是其中异常璀璨的一员了。它的生态圈涉及内核、操作系统、虚拟化、云计算、DevOps 等热门领域，受众群体也在不断扩大。

Docker 在国内的发展如火如荼，短短一两年时间里就陆续出现了一批关于 Docker 的创业公司。华为公司作为国内开源领域的领导者，对 Docker 也有很大的投入，我们认为有必要把自己的知识积累和实践经验总结出来分享给广大开发者。除了吸引更多的人投入到 Docker 的生态建设以外，我们也希望通过本书帮助更多的读者更好、更快地掌握 Docker 关键技术。

关于本书

目前市场已经有一些不错的 Docker 入门图书，但多侧重于入门和具体的应用，本书会介绍一些 Docker 关键技术原理和高级使用技巧，适合有一定基础的读者。另外，本书会对 Docker 涉及各个模块、关系和原理进行系统梳理，帮助读者对 Docker 加深认识，更好地应用 Docker 部署生产环境，最大程度安全有效地发挥 Docker 的价值。

本书不仅适合一般的 Docker 用户，也适合 Docker 生态圈中的开发者，希望它可以成为一本 Docker 进阶的图书，帮助读者快速提升。

本书是由华为整个 Docker 团队合作完成的，笔者包括（排名不分先后）：邓广兴、胡科平、胡欣蔚、黄强、雷继棠、李泽帆、凌发科、刘华、孙远、谢可杨、杨书奎、张伟、张文涛、邹钰。

本书的内容

本书的定位是有一定 Docker 基础的读者，所以在基本的概念和使用上，我们不会花过

多的篇幅讲解，而是给出相应有价值的链接，作为读者的延伸阅读。

在内容上，除了对 Docker 进行系统的梳理外，同时还会对 Docker 背后的核心技术（即容器技术）及其历史进行介绍，进一步帮助读者更好地理解 Docker。

章节划分则以功能模块为粒度，对每一个重要的模块进行了深入分析和讲解，同时也为热门领域单独开辟了章节。在每一章的最后都会讲解一些高级用法、使用技巧或实际应用中遇到的问题。虽然各章节的内容相对独立，但也会有一些穿插的介绍和补充，以帮助读者融会贯通，系统深入地理解 Docker 的每一个细节。

另外，本书的笔者都是一线的开发者和 Docker 社区活跃的贡献者，因此书中还专门准备了一个章节来介绍参与 Docker 开发的流程和经验。同时，伴随 Docker 的发展，Docker 生态圈也在不断扩大并吸引了越来越多的人的关注。Docker 集群管理和生态圈的介绍也将作为本书重点章节详细讲解。此外，Docker 测试也是比较有特色的内容，分享了笔者在测试方面的经验。最后，附录中所包含的常用的 Docker 相关信息，可供读者需要时查询。

本书的内容和代码都是基于 Docker 1.8 版本的。在代码示例中，使用“#”开头的命令表示以 root 用户执行，以“\$”开头的命令表示以普通用户执行。

勘误和支持

由于笔者水平有限，编写的时间也很仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。读者可以把书中发现的问题或建议发送到邮箱 docker@huawei.com，我们会尽快回复大家的疑问，并把收集的信息整理修正。

致谢

本书是由整个 Docker 团队协作完成的，由于繁忙的工作书稿撰写几度中止。感谢我们的项目经理裴斐月女士，正是她的整体协调和督促，以及与出版社的大量沟通，才促成了本书的出版。感谢李泽帆，他不仅参与了本书的写作，而且承担了全书的审读工作，给出了大量有价值的建议。还要感谢 Stephen Li、陈佳波、杨开封、胡欣蔚和张殿芳，以及其他华为公司主管对我们写书的大力支持，感谢机械工业出版社的编辑耐心专业的指导和审核。最后，感谢我们每一位家人的支持陪伴，我们的工作因为有了家人的支持和期待才变得更

华为 Docker 实践小组

2015 年 11 月

Contents 目 录

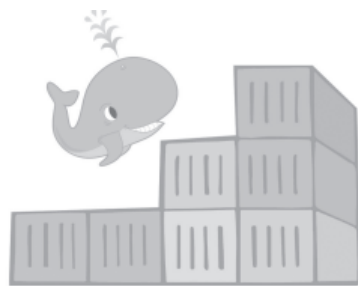
序	1.5 本章小结	10
前 言	第2章 关于容器技术	11
第1章 Docker简介	2.1 容器技术的前世今生	11
1.1 引言	2.1.1 关于容器技术	11
1.1.1 Docker 的历史和发展	2.1.2 容器技术的历史	12
1.1.2 Docker 的架构介绍	2.2 一分钟理解容器	14
1.2 功能和组件	2.2.1 容器的组成	14
1.2.1 Docker 客户端	2.2.2 容器的创建原理	15
1.2.2 Docker daemon	2.3 Cgroup 介绍	16
1.2.3 Docker 容器	2.3.1 Cgroup 是什么	16
1.2.4 Docker 镜像	2.3.2 Cgroup 的接口和使用	17
1.2.5 Registry	2.3.3 Cgroup 子系统介绍	18
1.3 安装和使用	2.4 Namespace 介绍	20
1.3.1 Docker 的安装	2.4.1 Namespace 是什么	20
1.3.2 Docker 的使用	2.4.2 Namespace 的接口和使用	21
1.4 概念澄清	2.4.3 各个 Namespace 介绍	22
1.4.1 Docker 在 LXC 基础上做了 什么工作	2.5 容器造就 Docker	26
1.4.2 Docker 容器和虚拟机之间有 什么不同	2.6 本章小结	27
9	第3章 理解Docker镜像	28
	3.1 Docker image 概念介绍	28

3.2 使用 Docker image	29	4.5 Index 及仓库高级功能	64
3.2.1 列出本机的镜像	29	4.5.1 Index 的作用和组成	64
3.2.2 Build: 创建一个镜像	31	4.5.2 控制单元	65
3.2.3 Ship: 传输一个镜像	32	4.5.3 鉴权模块	66
3.2.4 Run: 以 image 为模板启动 一个容器	32	4.5.4 数据库	67
3.3 Docker image 的组织结构	33	4.5.5 高级功能	68
3.3.1 数据的内容	33	4.5.6 Index 客户端界面	69
3.3.2 数据的组织	35	4.6 本章小结	69
3.4 Docker image 扩展知识	37	第5章 Docker网络	71
3.4.1 联合挂载	37	5.1 Docker 网络现状	71
3.4.2 写时复制	37	5.2 基本网络配置	73
3.4.3 Git 式管理	40	5.2.1 Docker 网络初探	73
3.5 本章小结	40	5.2.2 Docker 网络相关参数	80
第4章 仓库进阶	41	5.3 高级网络配置	85
4.1 什么是仓库	41	5.3.1 容器跨主机多子网方案	85
4.1.1 仓库的组成	41	5.3.2 容器跨主机多子网配置 方法	86
4.1.2 仓库镜像	42	5.4 网络解决方案进阶	90
4.2 再看 Docker Hub	43	5.4.1 Weave	90
4.2.1 Docker Hub 的优点	43	5.4.2 Flannel	91
4.2.2 网页分布	44	5.4.3 SocketPlane	94
4.2.3 账户管理系统	46	5.5 本章小结	98
4.3 仓库服务	49	第6章 容器卷管理	99
4.3.1 Registry 功能和架构	49	6.1 Docker 卷管理基础	99
4.3.2 Registry API	50	6.1.1 增加新数据卷	99
4.3.3 Registry API 传输过程分析	53	6.1.2 将主机目录挂载为数据卷	100
4.3.4 鉴权机制	57	6.1.3 创建数据卷容器	100
4.4 部署私有仓库	61	6.1.4 数据卷的备份、转储和 迁移	101
4.4.1 运行私有服务	61		
4.4.2 构建反向代理	61		

6.1.5 Docker 卷管理的问题	101	8.2.3 容器组网	135
6.2 使用卷插件	102	8.2.4 容器 + 全虚拟化	136
6.2.1 卷插件简介	102	8.2.5 镜像签名	136
6.2.2 卷插件的使用	102	8.2.6 日志审计	136
6.3 卷插件剖析	103	8.2.7 监控	137
6.3.1 卷插件工作原理	104	8.2.8 文件系统级防护	137
6.3.2 卷插件 API 接口	105	8.2.9 capability	137
6.3.3 插件发现机制	105	8.2.10 SELinux	138
6.4 已有的卷插件	106	8.2.11 AppArmor	142
6.5 本章小结	107	8.2.12 Seccomp	144
第7章 Docker API	108	8.2.13 grsecurity	145
7.1 关于 Docker API	108	8.2.14 几个与 Docker 安全相关的 项目	146
7.1.1 REST 简介	108	8.3 安全加固	146
7.1.2 Docker API 初探	109	8.3.1 主机逃逸	147
7.1.3 Docker API 种类	110	8.3.2 安全加固之 capability	150
7.2 RESTful API 应用示例	110	8.3.3 安全加固之 SELinux	151
7.2.1 前期准备	111	8.3.4 安全加固之 AppArmor	152
7.2.2 Docker API 的基本示例	116	8.4 Docker 安全遗留问题	153
7.3 API 的高级应用	123	8.4.1 User Namespace	153
7.3.1 场景概述	123	8.4.2 非 root 运行 Docker daemon	153
7.3.2 场景实现	124	8.4.3 Docker 热升级	153
7.4 本章小结	131	8.4.4 磁盘限额	154
第8章 Docker 安全	132	8.4.5 网络 I/O	154
8.1 深入理解 Docker 的安全	132	8.5 本章小结	154
8.1.1 Docker 的安全性	132	第9章 Libcontainer 简介	155
8.1.2 Docker 容器的安全性	132	9.1 引擎的引擎	155
8.2 安全策略	133	9.1.1 关于容器的引擎	155
8.2.1 Cgroup	133	9.1.2 对引擎的理解	156
8.2.2 ulimit	135		

9.2 Libcontainer 的技术原理	157	11.1.1 Compose 概述	185
9.2.1 为容器创建新的命名空间 ..	158	11.1.2 Compose 配置简介	186
9.2.2 为容器创建新的 Cgroup	159	11.2 Machine	187
9.2.3 创建一个新的容器	160	11.2.1 Machine 概述	187
9.2.4 Libcontainer 的功能	164	11.2.2 Machine 的基本概念及 运行流程	188
9.3 关于 runC	166	11.3 Swarm	188
9.3.1 runC 和 Libcontainer 的 关系	166	11.3.1 Swarm 概述	188
9.3.2 runC 的工作原理	167	11.3.2 Swarm 内部架构	189
9.3.3 runC 的未来	168	11.4 Docker 在 OpenStack 上的 集群实战	190
9.4 本章小结	169	11.5 本章小结	196
第10章 Docker实战	170	第12章 Docker生态圈	197
10.1 Dockerfile 简介	170	12.1 Docker 生态圈介绍	197
10.1.1 一个简单的例子	171	12.2 重点项目介绍	198
10.1.2 Dockerfile 指令	171	12.2.1 编排	198
10.1.3 再谈 Docker 镜像制作	173	12.2.2 容器操作系统	203
10.2 基于 Docker 的 Web 应用和 发布	174	12.2.3 PaaS 平台	206
10.2.1 选择基础镜像	174	12.3 生态圈的未来发展	208
10.2.2 制作 HTTPS 服务器镜像 ..	175	12.3.1 Docker 公司的发展和完善 方向	208
10.2.3 将 Web 源码导入 Tomcat 镜像中	178	12.3.2 OCI 组织	209
10.2.4 部署与验证	179	12.3.3 生态圈格局的分化和 发展	210
10.3 为 Web 站点添加后台服务 ..	180	12.4 本章小章	211
10.3.1 代码组织结构	180	第13章 Docker测试	212
10.3.2 组件镜像制作过程	183	13.1 Docker 自身测试	212
10.3.3 整体部署服务	183	13.1.1 Docker 自身的测试框架 ..	212
10.4 本章小结	184	13.1.2 运行 Docker 测试	213
第11章 Docker集群管理	185	13.1.3 在容器中手动运行测试 ..	
11.1 Compose	185		

用例	215	14.2 编译自己的 Docker	235
13.1.4 运行集成测试中单个或多个测试用例	215	14.2.1 使用 <code>make</code> 工具编译	235
13.1.5 Docker 测试用例集介绍 ...	216	14.2.2 手动启动容器编译	235
13.1.6 Docker 测试需要改进的方面	217	14.2.3 编译动态链接的可执行文件	237
13.1.7 构建和测试文档	217	14.2.4 跑测试用例及小结	237
13.1.8 其他 Docker 测试套	218	14.3 开源的沟通和交流	238
13.2 Docker 技术在测试中的应用	220	14.3.1 Docker 沟通和交流的途径	238
13.2.1 Docker 对测试的革命性影响	221	14.3.2 开源沟通和交流的建议 ...	238
13.2.2 Docker 技术适用范围	222	14.4 Docker 项目的组织架构	239
13.2.3 Jenkins+Docker 自动化环境配置	223	14.4.1 管理模型	239
13.3 本章小结	229	14.4.2 组织架构	240
第14章 参与Docker开发	230	14.5 本章小章	242
14.1 改进 Docker	230	附录A FAQ	243
14.1.1 报告问题	230	附录B 常用Dockerfile	247
14.1.2 提交补丁	231	附录C Docker信息获取渠道	250



第 1 章 *Chapter 1*

Docker 简介

1.1 引言

1.1.1 Docker 的历史和发展

自从 2013 年年初一个叫 dotCloud 的 PaaS 服务供应商将一个内部项目 Docker 开源之后，这个名字在短短几年内就迅速成为一个热词。似乎一夜之间，人人都开始谈论 Docker，以至于这家公司干脆出售了其所持有的 PaaS 平台业务，并且改名为 Docker.Inc，从而专注于 Docker 的开发和推广。

对于 Docker，目前的定义是一个开源的容器引擎，可以方便地对容器（关于容器，将在第 2 章详细介绍）进行管理。其对镜像的打包封装，以及引入的 Docker Registry 对镜像的统一管理，构建了方便快捷的“Build, Ship and Run”流程，它可以统一整个开发、测试和部署的环境和流程，极大地减少运维成本。另外，得益于容器技术带来的轻量级虚拟化，以及 Docker 在分层镜像应用上的创新，Docker 在磁盘占用、性能和效率方面相较于传统的虚拟化都有非常明显的提高，所以理所当然，Docker 开始不断蚕食传统虚拟化的市场。

随着 Docker 技术的迅速普及，Docker 公司持续进行融资，并且其估值也在不断攀升，同时，Docker 公司也在不断地完善 Docker 生态圈，这一切使得 Docker 正慢慢成为轻量级虚拟化的代名词。在可预见的未来，很可能需要不断地刷新对 Docker 的定义。

目前 Docker 已加入 Linux 基金会，遵循 Apache 2.0 协议，其代码托管于 [Github] (<https://github.com/docker/docker>)。

1.1.2 Docker 的架构介绍

要了解 Docker，首先要看看它的架构图，如图 1-1 所示。

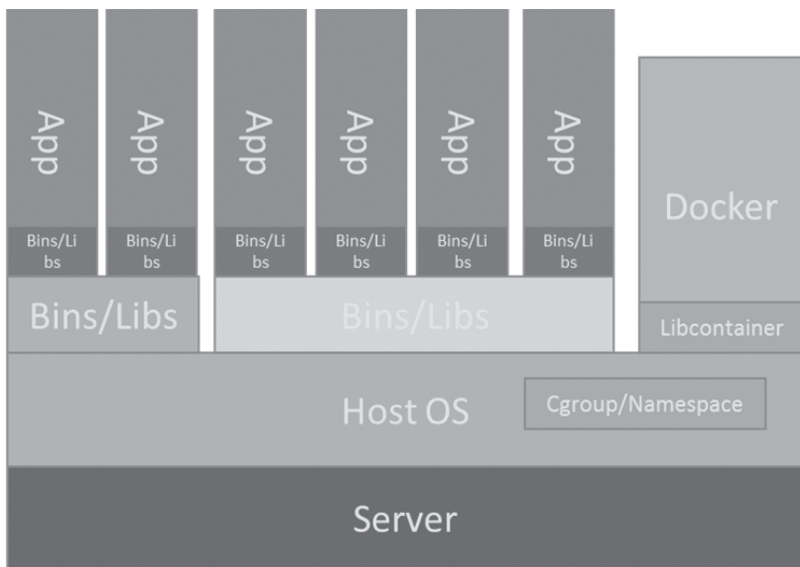



图 1-1 Docker 架构图

从图 1-1 可知，Docker 并没有传统虚拟化中的 Hypervisor 层。因为 Docker 是基于容器技术的轻量级虚拟化，相对于传统的虚拟化技术，省去了 Hypervisor 层的开销，而且其虚拟化技术是基于内核的 Cgroup 和 Namespace 技术，处理逻辑与内核深度融合，所以在很多方面，它的性能与物理机非常接近。

在通信上，Docker 并不会直接与内核交互，它是通过一个更底层的工具 Libcontainer 与内核交互的。Libcontainer 是真正意义上的容器引擎，它通过 clone 系统调用直接创建容器，通过 pivot_root 系统调用进入容器，且通过直接操作 cgroupfs 文件实现对资源的管控，而 Docker 本身则侧重于处理更上层的业务。

 **提示** Libcontainer 的详细介绍可参见第 9 章。

Docker 的另一个优势是对层级镜像的创新应用，即不同的容器可以共享底层的只读镜像，通过写入自己特有的内容后添加新的镜像层，新增的镜像层和下层的镜像一起又可以作为基础镜像被更上层的镜像使用。这种特性可以极大地提高磁盘利用率，所以当你的系统上有 10 个大小为 1GB 的镜像时，它们总共占用的空间大小可能只有 5GB，甚至更少。另外，Docker 对 Union mount 的应用还体现在多个容器使用同一个基础镜像时，可极大地减少内存占用等方面，因为不同的容器访问同一个文件时，只会占用一份内存。当然这需要支持 Union mount 的文件系统作为存储的 Graph Driver，比如 AUFS 和 Overlay。

1.2 功能和组件

Docker 为了实现其所描述的酷炫功能，引入了以下核心概念：

- ❑ Docker 客户端
- ❑ Docker daemon
- ❑ Docker 容器
- ❑ Docker 镜像
- ❑ Registry

下面就分别来简单地介绍一下。

1.2.1 Docker 客户端

Docker 是一个典型的 C/S 架构的应用程序，但在发布上，Docker 将客户端和服务端统一在同一个二进制文件中，不过，这只是对于 Linux 系统而言的，在其他平台如 Mac 上，Docker 只提供了客户端。

Docker 客户端一般通过 Docker command 来发起请求，另外，也可以通过 Docker 提供的一整套 RESTful API 来发起请求，这种方式更多地被应用在应用程序的代码中。

1.2.2 Docker daemon

Docker daemon 也可以被理解成 Docker Server，另外，人们也常常用 Docker Engine 来直接描述它，因为这实际上就是驱动整个 Docker 功能的核心引擎。

简单地说，Docker daemon 实现的功能就是接收客户端发来的请求，并实现请求所要求的功能，同时针对请求返回相应的结果。在功能的实现上，因为涉及了容器、镜像、存储等多方面的内容，daemon 内部的机制会复杂很多，涉及了多个模块的实现和交互。

1.2.3 Docker 容器

在 Docker 的功能和概念中，容器是一个核心内容，相对于传统虚拟化，它作为一项基础技术在性能上给 Docker 带来了极大优势。

在功能上，Docker 通过 Libcontainer 实现对容器生命周期的管理、信息的设置和查询，以及监控和通信等功能。而容器也是对镜像的完美诠释，容器以镜像为基础，同时又为镜像提供了一个标准的和隔离的执行环境。

在概念上，容器则很好地诠释了 Docker 集装箱的理念，集装箱可以存放任何货物，可以通过邮轮将货物运输到世界各地。运输集装箱的邮轮和装载卸载集装箱的码头都不用关心集装箱里的货物，这是一种标准的集装和运输方式。类似的，Docker 的容器就是“软件界的集装箱”，它可以安装任意的软件和库文件，做任意的运行环境配置。开发及运维人员在转移和部署应用的时候，不用关心容器里装了什么软件，也不用了解它们是如何配置的。

而管理容器的 Docker 引擎同样不关心容器里的内容，它只要像码头工人一样让这个容器运行起来就可以了，就像所有其他容器那样。

容器不是一个新的概念，但是 Docker 在对容器进行封装后，与集装箱的概念对应起来了，它之所以被称为“软件界的创新和革命”，是因为它会改变软件的开发、部署形态，降低成本，提高效率。Docker 真正把容器推广到了全世界。

1.2.4 Docker 镜像

与容器相对应，如果说容器提供了一个完整的、隔离的运行环境，那么镜像则是这个运行环境的静态体现，是一个还没有运行起来的“运行环境”。

相对于传统虚拟化中的 ISO 镜像，Docker 镜像要轻量化很多，它只是一个可定制的 rootfs。Docker 镜像的另一个创新是它是层级的并且是可复用的，这在实际应用场景中极为有用，多数基于相同发行版的镜像，在大多数文件的内容上都是一样的，基于此，当然会希望可以复用它们，而 Docker 做到了。在此类应用场景中，利用 Unionfs 的特性，Docker 会极大地减少磁盘和内存的开销。

Docker 镜像通常是通过 Dockerfile 来创建的，Dockerfile 提供了镜像内容的定制，同时也体现了层级关系的建立。另外 Docker 镜像也可以通过使用 `docker commit` 这样的命令来手动将修改后的内容生成镜像，这些都将在后续的章节详细介绍。

1.2.5 Registry

在前面提到的镜像中，曾指出 Docker 通过容器集装箱可以很方便地转运软件，其实，Registry 也在其中扮演了重要的角色。

Registry 是一个存放镜像的仓库，它通常被部署在互联网服务器或者云端。通常，集装箱是需要通过邮轮经行海洋运输到世界各地的，而互联网时代的传输则要方便很多，在镜像的传输过程中，Registry 就是这个传输的重要中转站。假如我们在公司将一个软件的运行环境制作成镜像，并上传到 Registry 中，这时就可以很方便地在家里的笔记本上，或者在客户的生产环境上直接从 Registry 上下载并运行了。当然，对 Registry 的操作也是与 Docker 完美融合的，用户甚至不需要知道 Registry 的存在，只需要通过简单的 Docker 命令就可以实现上面的操作。

Docker 公司提供的官方 Registry 叫 Docker Hub，这上面提供了大多数常用软件和发行版的官方镜像，还有无数个人用户提供的个人镜像。其实，Registry 本身也是一个单独的开源项目，任何人都可以下载后自己部署一个 Registry。因为免费的 Docker Hub 功能相对简单，所以多数企业会选择自己部署 Docker Registry 后二次开发，或者购买功能更强大的企业版 Docker Hub。

1.3 安装和使用

1.3.1 Docker 的安装

Docker 的安装和使用有一些前提条件，主要体现在体系架构和内核的支持上。对于体系架构，除了 Docker 一开始就支持的 x86-64，其他体系架构的支持则一直在不断地完善和推进中，用户在安装前需要到 Docker 官方网站查看最新的支持情况。对于内核，目前官方的建议是 3.10 以上的版本，除了内核版本以外，Docker 对于内核支持的功能，即内核的配置选项也有一定的要求（比如必须开启 Cgroup 和 Namespace 相关选项，以及其他的网络和存储驱动等）。如果你使用的是主流的发行版，那通常它们都已经打开了，如果使用的是定制化的内核，Docker 源码中提供了一个检测脚本（目前的路径是 `./contrib/check-config.sh`）来检测和引导内核的配置。

在满足前提条件后，安装就非常的简单了，对于多数主流的发行版，通常只需要一条简单的命令即可完成安装，比如在 Ubuntu 下，可以使用如下命令安装：

```
$ sudo apt-get install docker.io
```

当然，实际情况可能会相对复杂些，比如，虽然 Ubuntu 中通常自带了 Docker，但用户常常需要使用最新版本的 Docker，以至于不得不对其进行升级。对于安装和升级，以及不同发行版上的操作方法，官方网站上提供了更加详细的说明，本书不做过多的赘述，下面的链接给出了常用发行版的安装方法：

- ❑ [Ubuntu](<http://docs.docker.com/installation/ubuntulinux/>)
- ❑ [Fedora](<http://docs.docker.com/installation/fedora/>)
- ❑ [Debian](<http://docs.docker.com/installation/debian/>)
- ❑ [Centos](<http://docs.docker.com/installation/centos/>)
- ❑ [Gentoo](<http://docs.docker.com/installation/gentoolinux/>)
- ❑ [Arch Linux](<http://docs.docker.com/installation/archlinux/>)
- ❑ [Windows](<http://docs.docker.com/installation/windows/>)
- ❑ [Mac OS X](<http://docs.docker.com/installation/mac/>)

另外，用户也可以直接获取 Docker binary 来运行，<http://docs.docker.com/installation/binaries/> 网址介绍了获取的方法。虽然这样更简单，但还是推荐使用完整安装的方式，因为通过软件包安装的 Docker，除了有可执行文件之外，还包括了 Shell 自动完成脚本、man 手册、服务运行和配置脚本等内容，可以帮助用户更好地配置和使用 Docker。



提示 Docker 还有一些其他更方便的安装方式，这将在后面的章节中详细介绍。

1.3.2 Docker 的使用

对于 Docker 的使用，可以花整本书来介绍其中的各种细节、使用技巧和实战经验等，本节更希望告诉读者学习使用的方法，而对于使用技巧和实战经验会在本书的其他部分贯穿说明。

对于初学者，官方提供的 [tryit] (<https://www.docker.com/tryit/>) 是最好的快速入门途径，建议每一个初次接触 Docker 的用户都可以试一试。对于有一定经验的用户，在使用中遇到问题或者不确定具体的用法时，可以通过以下途径来查看帮助信息。

1) 在控制台直接运行 docker，这样会列出 Docker 支持的所有命令和一些通用的参数，如下：

```
$ docker
Usage: docker [OPTIONS] COMMAND [arg...]
       docker daemon [ --help | ... ]
       docker [ -h | --help | -v | --version ]

A self-sufficient runtime for containers.

Options:
    --config=~/.docker                Location of client config files
    -D, --debug=false                 Enable debug mode
    -H, --host=[]                     Daemon socket(s) to connect to
    -h, --help=false                  Print usage
    -l, --log-level=info               Set the logging level
    --tls=false                       Use TLS; implied by --tlsverify
    --tlscacert=~/.docker/ca.pem      Trust certs signed only by this CA
    --tlscert=~/.docker/cert.pem      Path to TLS certificate file
    --tlskey=~/.docker/key.pem        Path to TLS key file
    --tlsverify=false                 Use TLS and verify the remote
    -v, --version=false               Print version information and quit

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
    cp        Copy files/folders from a container to a HOSTDIR or to STDOUT
    create    Create a new container
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    exec      Run a command in a running container
    export    Export a container's filesystem as a tar archive
    history   Show the history of an image
    images    List images
    import    Import the contents from a tarball to create a filesystem image
    info      Display system-wide information
    inspect   Return low-level information on a container or image
```

kill	Kill a running container
load	Load an image from a tar archive or STDIN
login	Register or log in to a Docker registry
logout	Log out from a Docker registry
logs	Fetch the logs of a container
pause	Pause all processes within a container
port	List port mappings or a specific mapping for the CONTAINER
ps	List containers
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rename	Rename a container
restart	Restart a running container
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save an image(s) to a tar archive
search	Search the Docker Hub for images
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop a running container
tag	Tag an image into a repository
top	Display the running processes of a container
unpause	Unpause all processes within a container
version	Show the Docker version information
wait	Block until a container stops, then print its exit code

Run 'docker COMMAND --help' for more information on a command.

2) 在控制台执行“docker + 命令 + --help”，比如“docker start --help”，这样会列出 docker start 命令支持的所有参数，如下：

```
$ docker start --help
```

```
Usage: docker start [OPTIONS] CONTAINER [CONTAINER...]
```

```
Start one or more stopped containers
```

```
-a, --attach=false      Attach STDOUT/STDERR and forward signals
--help=false           Print usage
-i, --interactive=false Attach container's STDIN
```

3) 使用 man 命令查看帮助文档。对于通过 rpm 包等方式安装的 Docker，一般都会默认安装对应的 man 文档，此时可通过“man + docker + command”的方式查看子命令的帮助文档，比如“man docker start”，通常 man 手册中包含的帮助信息会更丰富一些，通过完整地阅读 man 手册，基本上就可以掌握该命令的常规用法。

DOCKER(1)

JUNE 2014

DOCKER(1)

NAME

8 ◆ Docker 进阶与实战

```
docker-start - Start one or more stopped containers

SYNOPSIS
    docker start [-a|--attach[=false]] [--help] [-i|--interactive[=false]]
CONTAINER [CONTAINER...]

DESCRIPTION
    Start one or more stopped containers.

OPTIONS
    -a, --attach=true|false
        Attach container's STDOUT and STDERR and forward all signals to the
process. The default is false.

    --help
        Print usage statement

    -i, --interactive=true|false
        Attach container's STDIN. The default is false.

See also
    docker-stop(1) to stop a running container.

HISTORY
    April 2014, Originally compiled by William Henry (whentry at redhat dot
com) based on docker.com source material and internal work.
    June 2014, updated by Sven Dowideit (SvenDowideit@home.org.au)
```

1.4 概念澄清

本书的附录 A 是关于 Docker 常见问题的解答，但对于 Docker 基本概念方面的问题，希望读者可以在阅读完本章后就有清晰的认识，所以本节会针对与 Docker 概念息息相关的几个常见问题进行说明。

1.4.1 Docker 在 LXC 基础上做了什么工作

首先我们需要明确 LXC 的概念，但这常常有不同的认识。LXC 目前代表两种含义：

- ❑ LXC 用户态工具 (<https://github.com/lxc/lxc>)。
- ❑ Linux Container，即内核容器技术的简称。

这里通常指第二种，即 Docker 在内核容器技术的基础上做了什么工作。简单地说，Docker 在内核容器技术（Cgroup 和 Namespace）的基础上，提供了一个更高层的控制工具，该工具包含以下特性。

- 跨主机部署。Docker 定义了镜像格式，该格式会将应用程序和其所依赖的文件打包到同一个镜像文件中，从而使其在转移到任何运行 Docker 的机器中时都可以运行，

并能够保证在任何机器中该应用程序执行的环境都是一样的。LXC 实现了“进程沙盒”，这一重要特性是跨主机部署的前提条件，但是只有这一点远远不够，比如，在一个特定的 LXC 配置下执行应用程序，将该应用程序打包并拷贝到另一个 LXC 环境下，程序很有可能无法正常执行，因为该程序的执行依赖于该机器的特定配置，包括网络、存储、发行版等。而 Docker 则将上述相关配置进行抽象并与应用程序一同打包，所以可以保证在不同硬件、不同配置的机器上 Docker 容器中运行的程序 and 其所依赖的环境及配置是一致的。

- 以应用为中心。Docker 为简化应用程序的部署过程做了很多优化，这一目的在 Docker 的 API、用户接口、设计哲学和用户文档中都有体现，其 Dockerfile 机制大大简化和规范了应用的部署方法。
- 自动构建。Docker 提供了一套能够从源码自动构建镜像的工具。该工具可以灵活地使用 make、maven、chef、puppet、salt、debian 包、RPM 和源码包等形式，将应用程序的依赖、构建工具和安装包进行打包处理，而且当前机器的配置不会影响镜像的构建过程。
- 版本管理。Docker 提供了类似于 Git 的版本管理功能，支持追踪镜像版本、检验版本更新、提交新的版本改动和回退版本等功能。镜像的版本信息中包括制作方式和制作者信息，因此可以从生产环境中回溯到上游开发人员。Docker 同样实现了镜像的增量上传下载功能，用户可以通过获取新旧版本之间新增的镜像层来更新镜像版本，而不必下载完整镜像，类似 Git 中的 pull 命令。
- 组件重用。任何容器都可以用作生成另一个组件的基础镜像。这一过程可以手动执行，也可以写入自动化构建的脚本。例如，可以创建一个包含 Python 开发环境的镜像，并将其作为基础镜像部署其他使用 Python 环境进行开发的应用程序。
- 共享。Docker 用户可以访问公共的镜像 Registry，并向其中上传有用的镜像。Registry 中同样包含由 Docker 公司维护的一些官方标准镜像。Docker Registry 本身也是开源的，所以任何人都可以部署自己的 Registry 来存储并共享私有镜像。
- 工具生态链。Docker 定义了一系列 API 来定制容器的创建和部署过程并实现自动化。有很多工具能够与 Docker 集成并扩展 Docker 的能力，包括类 PaaS 部署工具（Dokku、Deis 和 Flynn）、多节点编排工具（Maestro、Salt、Mesos、OpenStack nova）、管理面板（Docker-ui、OpenStack Horizon、Shipyard）、配置管理工具（Chef、Puppet）、持续集成工具（Jenkins、Strider、Travis）等。Docker 正在建立以容器为基础的工具集标准。

1.4.2 Docker 容器和虚拟机之间有什么不同

容器与虚拟机是互补的。虚拟机是用来进行硬件资源划分的完美解决方案，它利用了硬件虚拟化技术，例如 VT-x、AMD-V 或者 privilege level（权限等级）会同时通过一个

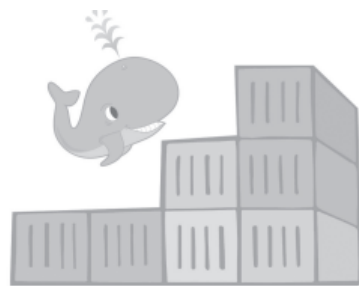
hypervisor 层来实现对资源的彻底隔离；而容器则是操作系统级别的虚拟化，利用的是内核的 Cgroup 和 Namespace 特性，此功能完全通过软件来实现，仅仅是进程本身就可以与其他进程隔离开，不需要任何辅助。

Docker 容器与主机共享操作系统内核，不同的容器之间可以共享部分系统资源，因此容器更加轻量级，消耗的资源也更少。而虚拟机会独占分配给自己的资源，几乎不存在资源共享，各个虚拟机实例之间近乎完全隔离，因此虚拟机更加重量级，也会消耗更多的资源。我们可以很轻松地在一台普通的 Linux 机器上运行 100 个或者更多的 Docker 容器，而且不会占用太多系统资源（如果容器中没有执行运算任务或 I/O 操作）；而在单台机器上不可能创建 100 台虚拟机，因为每一个虚拟机实例都会占用一个完整的操作系统所需要的所有资源。另外，Docker 容器启动很快，通常是秒级甚至是毫秒级启动。而虚拟机的启动虽然会快于物理机器，但是启动时间也是在数秒至数十秒的量级。

因此，可以根据需求的不同选择相应的隔离方式。如果需要资源完全隔离并且不考虑资源消耗，可以选择使用虚拟机；而若是想隔离进程并且需要运行大量进程实例，则应该选择 Docker 容器。

1.5 本章小结

本章对 Docker 的基本概念、组成和使用方法做了介绍，使读者对 Docker 有一个整体的认识，后续的章节会对本章提到的内容展开更详细的讲解，让读者对 Docker 有全面且细致的理解。



第 2 章 *Chapter 2*

关于容器技术

在第 1 章对 Docker 的介绍中，已经知道容器技术是 Docker 的一项基础技术，而在当前对 Docker 的火热讨论中，容器也时常跟 Docker 一起被提及。作为 Docker 的进阶书籍，有必要对容器技术做一些探讨，以深刻理解 Docker 与相关技术之间的关联。

2.1 容器技术的前世今生

2.1.1 关于容器技术

容器技术，又称为容器虚拟化，从字面上看它首先是一种虚拟化技术。在如今的技术浪潮下，虚拟化技术层出不穷，包括硬件虚拟化、半虚拟化、操作系统虚拟化等。本书不会对虚拟化技术展开介绍，只需要知道容器虚拟化是一种操作系统虚拟化，是属于轻量级的虚拟化技术即可。又因为在实现原理上，每一种虚拟化技术之间都有较大的差别，所以即使没有虚拟化的技术背景，也是可以单独来学习容器虚拟化的。

容器技术之所以受欢迎，一个重要的原因是它已经集成到了 Linux 内核中，已经被当作 Linux 内核原生提供的特性。当然在其他平台上也有相应的容器技术，但本书讨论的以及 Docker 涉及的都是指 Linux 平台的容器技术。

对于容器，目前并没有一个严格的定义，但普遍认可的说法是，它首先必须是一个相对独立的运行环境，在这一点上，有点类似虚拟机的概念，但又没有虚拟机那样彻底。另外，在一个容器环境内，应该最小化其对外界的影响，比如不能在容器中把 host 上的资源全部消耗掉，这就是资源控制。

一般来说，容器技术主要包括 Namespace 和 Cgroup 这两个内核特性。

- ❑ Namespace 又称为命名空间（也可翻译为名字空间），它主要做访问隔离。其原理是针对一类资源进行抽象，并将其封装在一起提供给一个容器使用，对于这类资源，因为每个容器都有自己的抽象，而它们彼此之间是不可见的，所以就可以做到访问隔离。
- ❑ Cgroup 是 control group 的简称，又称为控制组，它主要是做资源控制。其原理是将一组进程放在一个控制组里，通过给这个控制组分配指定的可用资源，达到控制这一组进程可用资源的目的。

实际上，Namespace 和 Cgroup 并不是强相关的两种技术，用户可以根据需要单独使用它们，比如单独使用 Cgroup 做资源控制，就是一种比较常见的做法。而如果把它们应用到一起，在一个 Namespace 中的进程恰好又在一个 Cgroup 中，那么这些进程就既有访问隔离，又有资源控制，符合容器的特性，这样就创建了一个容器。

对于 Namespace 和 Cgroup，后面的章节会做详细的介绍。

2.1.2 容器技术的历史

上文提到容器技术属于一种操作系统虚拟化，事实上，其最早的原型可以简化为对目录结构的简单抽象，如图 2-1 所示。

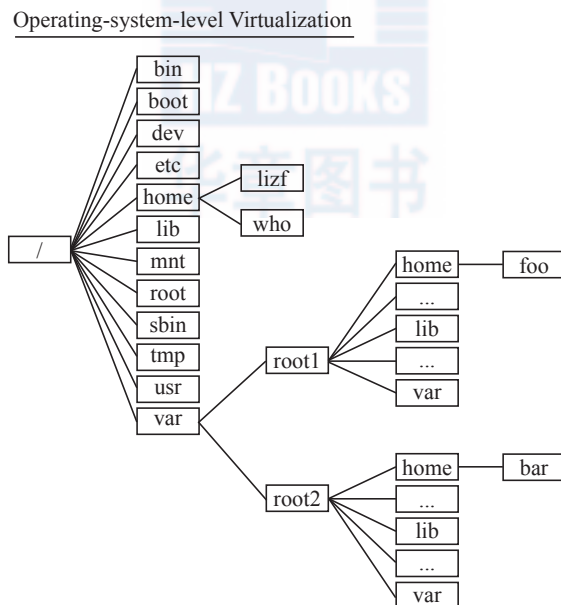



图 2-1 容器技术最早原型

图 2-1 所示为在普通的目录结构中创建一个完整的子目录结构。这种抽象化目录结构的出现最早源于 1982 年，那时通过 chroot 技术把用户的文件系统根目录切换到某个指定的

目录下，实现了简单的文件系统视图上的抽象或虚拟化。但是这种技术只是提供了有限的文件系统隔离，并没有任何其他隔离手段，而且人们后来发现这种技术并不安全，用户可以逃离设定的根目录从而访问 host 上的文件。

针对上面提到的安全性问题，在 2000 年，内核版本 2.3.41 引入了 pivot_root 技术，它可以有效地避免 chroot 带来的安全性问题。今日的容器技术，比如 LXC、Docker 等，也都是使用了 pivot_root 来做根文件系统的切换。然而 pivot_root 也仅仅是在文件系统的隔离上做了一些增强，并没有在其他隔离性上有所提高。

同样在 2000 年左右，市场上出现了一些商用的容器技术，比如 Linux-VServer 和 SWsoft（现在的 Odin）开发的 Virtuozzo，虽然这些技术相对当时的 XEN 和 KVM，有明显的性能提升，但是因为各种原因，并未在当时引起市场太多的关注。

 **注意** 这里只讨论 Linux 系统上的容器技术，同时期还有很多有名的非 Linux 平台的容器技术，比如 FreeBSD 的 jail、Solaris 上的 Zone 等。

到了 2005 年，同样是 Odin 公司，在 Virtuozzo 的基础上发布了 OpenVZ 技术，同时开始推动 OpenVZ 中的核心容器技术进入 Linux 内核主线，而此时 IBM 等公司也在推动类似的技术，最后在社区的合作下，形成了目前大家看到的 Cgroup 和 Namespace，这时，容器技术才开始逐渐进入大众的视野。

对于 Namespace，其各个子系统进入内核的版本号及贡献公司如表 2-1 所示。

表 2-1 Namespace 内核版本支持

子系统	引入版本	贡献公司
Mount Namespace	2.4.19	N/A
UTS Namespace	2.6.19	IBM、Parallels
IPC Namespace	2.6.19	Parallels
PID Namespace	2.6.24	Parallels
Net Namespace	2.6.24	Parallels、IBM、XMission
User Namespace	2.6.23 & 3.8	IBM、XMission

 **说明** User Namespace 在 3.8 版本重新实现。

对于 Cgroup，其各个子系统进入内核的版本号及贡献公司如表 2-2 所示。

表 2-2 Cgroup 内核版本支持

子系统	引入版本	贡献公司
cpuset	2.6.12	SGI
ns (之后被删除)	2.6.24	Google
CPU	2.6.24	IBM
memory	2.6.25	IBM

(续)

子系统	引入版本	贡献公司
device	2.6.26	IBM
freezer	2.6.28	IBM
blkio	2.6.33	Redhat

 **注意** 以上只列举了早期主要的子系统，较新的子系统如 net cls、hugetlb 等并未列出。

整个容器的发展历史可以通过图 2-2 来展示。

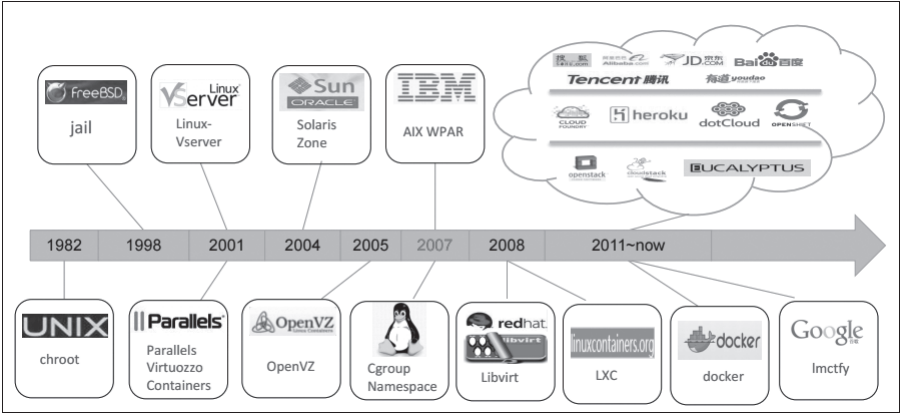


图 2-2 容器发展历史

随着容器技术在内核主线中的不断成熟和完善，2013 年诞生的 Docker 真正让容器技术得到了全世界技术公司和开发人员的关注，相信容器技术的未来一定会比它的前世和今生更加精彩。

2.2 一分钟理解容器

2.2.1 容器的组成

上文已多次提及，容器的核心技术是 Cgroup + Namespace，但光有这两个抽象的技术概念是无法组成一个完整的容器的。在 2.1.2 节也提到过最早的容器概念就包括了对文件目录视图的抽象隔离，而所有的这一切，都需要有工具来驱动，需要有一个工具来提供用户可操作的接口，来创建一个容器。所以笔者认为，对于 Linux 容器的最小组成，可以由以下公式来表示：

容器 = cgroup + namespace + rootfs + 容器引擎（用户态工具）

其中各项的功能分别为：

- ❑ Cgroup: 资源控制。
- ❑ Namespace: 访问隔离。
- ❑ rootfs: 文件系统隔离。
- ❑ 容器引擎: 生命周期控制。

目前市场上所有 Linux 容器项目都包含以上组件。

2.2.2 容器的创建原理

至此对容器的描述还一直停留在文件和概念的层面，本小节将通过简单的代码抽象，清晰地展现容器的创建原理，使读者对容器有更深刻的理解。

代码一：

```
pid = clone(fun, stack, flags, clone_arg);
(flags: CLONE_NEWPID | CLONE_NEWNS |
        CLONE_NEWUSER | CLONE_NEWNET |
        CLONE_NEWIPC | CLONE_NEWUTS |
        ...)
```

代码二：

```
echo $pid > /sys/fs/cgroup/cpu/tasks
echo $pid > /sys/fs/cgroup/cpuset/tasks
echo $pid > /sys/fs/cgroup/blkio/tasks
echo $pid > /sys/fs/cgroup/memory/tasks
echo $pid > /sys/fs/cgroup/devices/tasks
echo $pid > /sys/fs/cgroup/freezer/tasks
```

代码三：

```
fun()
{
    ...
    pivot_root("path_of_rootfs/", path);
    ...
    exec("/bin/bash");
    ...
}
```

- ❑ 对于代码一，通过 clone 系统调用，并传入各个 Namespace 对应的 clone flag，创建了一个新的子进程，该进程拥有自己的 Namespace。根据以上代码可知，该进程拥有自己的 pid、mount、user、net、ipc、uts namespace。
- ❑ 对于代码二，将代码一中产生的进程 pid 写入各个 Cgroup 子系统中，这样该进程就可以受到相应 Cgroup 子系统的控制。
- ❑ 对于代码三，该 fun 函数由上面生成的新进程执行，在 fun 函数中，通过 pivot_root 系统调用，使进程进入一个新的 rootfs，之后通过 exec 系统调用，在新的

Namespace、Cgroup、rootfs 中执行 “/bin/bash” 程序。

通过以上操作，成功地在 “容器” 中运行了一个 bash 程序。对于 Cgroup 和 Namespace 的技术细节，将在以下两节详细描述。

2.3 Cgroup 介绍

2.3.1 Cgroup 是什么

Cgroup 是 control group 的简写，属于 Linux 内核提供的一个特性，用于限制和隔离一组进程对系统资源的使用，也就是做资源 QoS，这些资源主要包括 CPU、内存、block I/O 和网络带宽。Cgroup 从 2.6.24 开始进入内核主线，目前各大发行版都默认打开了 Cgroup 特性。

从实现的角度来看，Cgroup 实现了一个通用的进程分组的框架，而不同资源的具体管理则是由各个 Cgroup 子系统实现的。截止到内核 4.1 版本，Cgroup 中实现的子系统及其作用如下：

- ❑ devices：设备权限控制。
- ❑ cpuset：分配指定的 CPU 和内存节点。
- ❑ cpu：控制 CPU 占用率。
- ❑ cpuacct：统计 CPU 使用情况。
- ❑ memory：限制内存的使用上限。
- ❑ freezer：冻结（暂停）Cgroup 中的进程。
- ❑ net_cls：配合 tc（traffic controller）限制网络带宽。
- ❑ net_prio：设置进程的网络流量优先级。
- ❑ huge_tlb：限制 HugeTLB 的使用。
- ❑ perf_event：允许 Perf 工具基于 Cgroup 分组做性能监测。

在 Cgroup 出现之前，只能对一个进程做一些资源控制，例如通过 sched_setaffinity 系统调用限定一个进程的 CPU 亲和性，或者用 ulimit 限制一个进程的打开文件上限、栈大小等。另外，使用 ulimit 可以对少数资源基于用户做资源控制，例如限制一个用户能创建的进程数。而 Cgroup 可以对进程进行任意的分组，如何分组是用户自定义的，例如安卓的应用分为前台应用和后台应用，前台应用是直接跟用户交互的，需要响应速度快，因此前台应用对资源的申请需要得到更多的保证。为此安卓将前台应用和后台应用划分到不同的 Cgroup 中，并且对放置前台应用的 Cgroup 配置了较高的系统资源限额。



提示

从 1.6 版本开始，Docker 也支持 ulimit，读者可以查阅相关 Docker 文档及 Linux 用户手册。

2.3.2 Cgroup 的接口和使用

Cgroup 的原生接口通过 cgroupfs 提供，类似于 procfs 和 sysfs，是一种虚拟文件系统。以下用一个实例演示如何使用 Cgroup。

(1) 挂载 cgroupfs

命令如下：

```
# mount -t cgroup -o cpuset cpuset /sys/fs/cgroup/cpuset
```

首先必须将 cgroupfs 挂载起来，这个动作一般已经在启动时由 Linux 发行版做好了。可以把 cgroupfs 挂载在任意一个目录上，不过标准的挂载点是 /sys/fs/cgroup。



注意 实际上 sysfs 里面只有 /sys/fs/cgroup 目录，并且 sysfs 是不允许用户创建目录的，这里可以将 tmpfs 挂载上去，然后在 tmpfs 上创建目录。

(2) 查看 cgroupfs

```
# ls /sys/fs/cgroup/cpuset
cgroup.clone_children    cpuset.memory_pressure
cgroup.procs             cpuset.memory_pressure_enabled
cgroup.sane_behavior     cpuset.memory_spread_page
cpuset.cpu_exclusive     cpuset.memory_spread_slab
cpuset.cpus              cpuset.mems
cpuset.effective_cpus    cpuset.sched_load_balance
cpuset.effective_mems    cpuset.sched_relax_domain_level
cpuset.mem_exclusive     notify_on_release
cpuset.mem_hardwall      release_agent
cpuset.memory_migrate    tasks
```

可以看到这里有很多控制文件，其中以 cpuset 开头的控制文件都是 cpuset 子系统产生的，其他文件则由 Cgroup 产生。这里面的 tasks 文件记录了这个 Cgroup 的所有进程（包括线程），在系统启动后，默认没有对 Cgroup 做任何配置的情况下，cgroupfs 只有一个根目录，并且系统所有的进程都在这个根目录中，即进程 pid 都在根目录的 tasks 文件中。



注意 实际上现在大多数 Linux 发行版都是采用的 systemd，而 systemd 使用了 Cgroup，所以在这些发行版上，当系统启动后看到的 cgroupfs 是有一些子目录的。

(3) 创建 Cgroup

```
# mkdir /sys/fs/cgroup/cpuset/child
```

通过 mkdir 创建一个新的目录，也就创建了一个新的 Cgroup。

(4) 配置 Cgroup

```
# echo 0 > /sys/fs/cgroup/cpuset/child/cpuset.cpus
# echo 0 > /sys/fs/cgroup/cpsuet/child/cpuset.mems
```

接下来配置这个 Cgroup 的资源配额，通过上面的命令，就可以限制这个 Cgroup 的进程只能在 0 号 CPU 上运行，并且只会从 0 号内存节点分配内存。

(5) 使能 Cgroup

```
# echo $$ > /sys/fs/cgroup/cpuset/child/tasks
```

最后，通过将进程 id 写入 tasks 文件，就可以把进程移动到这个 Cgroup 中。并且，这个进程产生的所有子进程也都会自动放在这个 Cgroup 里。这时，Cgroup 才真正起作用了。



提示 \$\$ 表示当前进程。另外，也可以把 pid 写入 cgroup.procs 中，两者的区别是写入 tasks 只会把指定的进程加到 child 中，写入 cgroup.procs 则会把这个进程所属的整个线程组都加到 child 中。

2.3.3 Cgroup 子系统介绍

对实际资源的分配和管理是由各个 Cgroup 子系统完成的，下面介绍几个主要的子系统。

1. cpuset 子系统

cpuset 可以为一组进程分配指定的 CPU 和内存节点。cpuset 一开始是用在高性能计算 (HPC) 上的，在 NUMA 架构的服务器上，通过将进程绑定到固定的 CPU 和内存节点上，来避免进程在运行时因跨节点内存访问而导致的性能下降。当然，现在 cpuset 也广泛用在了 kvm 和容器等场景上。

cpuset 的主要接口如下。

- ❑ cpuset.cpus: 允许进程使用的 CPU 列表 (例如 0 ~ 4,9)。
- ❑ cpuset.mems: 允许进程使用的内存节点列表 (例如 0 ~ 1)。

2. cpu 子系统

cpu 子系统用于限制进程的 CPU 占用率。实际上它有三个功能，分别通过不同的接口来提供。

- ❑ CPU 比重分配。这个特性使用的接口是 cpu.shares。假设在 cgroupfs 的根目录下创建了两个 Cgroup (C1 和 C2)，并且将 cpu.shares 分别配置为 512 和 1024，那么当 C1 和 C2 争用 CPU 时，C2 将会比 C1 得到多一倍的 CPU 占用率。要注意的是，只有当它们争用 CPU 时 cpu share 才会起作用，如果 C2 是空闲的，那么 C1 可以得到全部的 CPU 资源。
- ❑ CPU 带宽限制。这个特性使用的接口是 cpu.cfs_period_us 和 cpu.cfs_quota_us，这两个接口的单位是微秒。可以将 period 设置为 1 秒，将 quota 设置为 0.5 秒，那么 Cgroup 中的进程在 1 秒内最多只能运行 0.5 秒，然后就会被强制睡眠，直到进入下一个 1 秒才能继续运行。

- ❑ 实时进程的 CPU 带宽限制。以上两个特性都只能限制普通进程，若要限制实时进程，就要使用 `cpu.rt_period_us` 和 `cpu.rt_runtime_us` 这两个接口了。使用方法和上面类似。

3. cpuacct 子系统

cpuacct 子系统用来统计各个 Cgroup 的 CPU 使用情况，有如下接口。

- ❑ `cpuacct.stat`：报告这个 Cgroup 分别在用户态和内核态消耗的 CPU 时间，单位是 `USER_HZ`。`USER_HZ` 在 x86 上一般是 100，即 1 `USER_HZ` 等于 0.01 秒。
- ❑ `cpuacct.usage`：报告这个 Cgroup 消耗的总 CPU 时间，单位是纳秒。
- ❑ `cpuacct.usage_percpu`：报告这个 Cgroup 在各个 CPU 上消耗的 CPU 时间，总和也就是 `cpuacct.usage` 的值。

4. memory 子系统

memory 子系统用来限制 Cgroup 所能使用的内存上限，有如下接口。

- ❑ `memory.limit_in_bytes`：设定内存上限，单位是字节，也可以使用 k/K、m/M 或者 g/G 表示要设置数值的单位，例如

```
# echo 1G > memory.limit_in_bytes
```

默认情况下，如果 Cgroup 使用的内存超过上限，Linux 内核会尝试回收内存，如果仍然无法将内存使用量控制在上限之内，系统将会触发 OOM，选择并“杀死”该 Cgroup 中的某个进程。

- ❑ `memory.memsw.limit_in_bytes`：设定内存加上交换分区的使用总量。通过设置这个值，可以防止进程把交换分区用光。
- ❑ `memory.oom_control`：如果设置为 0，那么在内存使用量超过上限时，系统不会“杀死”进程，而是阻塞进程直到有内存被释放可供使用时；另一方面，系统会向用户态发送事件通知，用户态的监控程序可以根据该事件来做相应的处理，例如提高内存上限等。
- ❑ `memory.stat`：汇报内存使用信息。

5. blkio 子系统

blkio 子系统用来限制 Cgroup 的 block I/O 带宽，有如下接口。

- ❑ `blkio.weight`：设置权重值，范围在 100 到 1000 之间。这跟 `cpu.shares` 类似，是比重分配，而不是绝对带宽的限制，因此只有当不同的 Cgroup 在争用同一个块设备的带宽时才会起作用。
- ❑ `blkio.weight_device`：对具体的设备设置权重值，这个值会覆盖上述的 `blkio.weight`。例如将 Cgroup 对 `/dev/sda` 的权重设为最小值：

```
# echo 8:0 100 > blkio.weight_device
```

- ❑ `blkio.throttle.read_bps_device`：对具体的设备，设置每秒读磁盘的带宽上限。例如对 `/dev/sda` 的读带宽限制在 1MB/秒：

```
# echo "8:0 1048576" > blkio.throttle.read_bps_device
```

- ❑ `blkio.throttle.write_bps_device`：设置每秒写磁盘的带宽上限。同样需要指定设备。
- ❑ `blkio.throttle.read_iops_device`：设置每秒读磁盘的 IOPS 上限。同样需要指定设备。
- ❑ `blkio.throttle.write_iops_device`：设置每秒写磁盘的 IOPS 上限。同样需要指定设备。



注意 `blkio` 子系统有两个重大的缺陷。一是对于比重分配，只支持 CFQ 调度器。另一个缺陷是，不管是比重分配还是上限限制，都只支持 Direct-IO，不支持 Buffered-IO，这使得 `blkio cgroup` 的使用场景很有限，好消息是 Linux 内核社区正在解决这个问题。

6. devices 子系统

`devices` 子系统用来控制 Cgroup 的进程对哪些设备有访问权限，有如下接口。

- ❑ `devices.list`。只读文件，显示目前允许被访问的设备列表。每个条目都有 3 个域，分别为：
 - 类型：可以是 `a`、`c` 或 `b`，分别表示所有设备、字符设备和块设备。
 - 设备号：格式为 `major:minor` 的设备号。
 - 权限：`r`、`w` 和 `m` 的组合，分别表示可读、可写、可创建设备结点 (`mknod`)。

例如 “`a *:* rmw`”，表示所有设备都可以被访问。而 “`c 1:3 r`”，表示对 `1:3` 这个字符设备（即 `/dev/null`）只有读权限。

- ❑ `devices.allow`。只写文件，以上面描述的格式写入该文件，就可以允许相应的设备访问权限。
- ❑ `devices.deny`。只写文件，以上面描述的格式写入该文件，就可以禁止相应的设备访问权限。

2.4 Namespace 介绍

2.4.1 Namespace 是什么

Namespace 是将内核的全局资源做封装，使得每个 Namespace 都有一份独立的资源，因此不同的进程在各自的 Namespace 内对同一种资源的使用不会互相干扰。

这样的解释可能不清楚，举个例子，执行 `sethostname` 这个系统调用时，可以改变系统的主机名，这个主机名就是一个内核的全局资源。内核通过实现 UTS Namespace，可以将不同的进程分隔在不同的 UTS Namespace 中，在某个 Namespace 修改主机名时，另一个 Namespace 的主机名还是保持不变。

目前 Linux 内核总共实现了 6 种 Namespace:

- ❑ IPC: 隔离 System V IPC 和 POSIX 消息队列。
- ❑ Network: 隔离网络资源。
- ❑ Mount: 隔离文件系统挂载点。
- ❑ PID: 隔离进程 ID。
- ❑ UTS: 隔离主机名和域名。
- ❑ User: 隔离用户 ID 和组 ID。

2.4.2 Namespace 的接口和使用

对 Namespace 的操作, 主要是通过 clone、setns 和 unshare 这 3 个系统调用来完成的。

clone 可以用来创建新的 Namespace。它接受一个叫 flags 的参数, 这些 flag 包括 CLONE_NEWNS、CLONE_NEWIPC、CLONE_NEWUTS、CLONE_NEWNET、CLONE_NEWPID 和 CLONE_NEWUSER, 我们通过传入这些 CLONE_NEW* 来创建新的 Namespace。这些 flag 对应的 Namespace 都可以从字面上看出来, 除了 CLONE_NEWNS, 这是用来创建 Mount Namespace 的。指定了这些 flag 后, 由 clone 创建出来的新进程, 就位于全新的 Namespace 里了, 并且很自然地这个新进程以后创建出来的进程, 也都在这个 Namespace 中。



提示 Mount Namespace 是第一个实现的 Namespace, 当初实现时并不是为了实现 Linux 容器, 因此也就没有预料到会有新的 Namespace 出现, 因此用了 CLONE_NEWNS 而不是 CLONE_NEWMNT 之类的名字。

那么, 能不能为已有的进程创建新的 Namespace 呢? 答案是可以, unshare 就是用来达到这个目的的。调用这个系统调用的进程, 会被放进新创建的 Namespace 里, 要创建什么 Namespace 由 flags 参数指定, 可以使用的 flag 也就是上面提到的那些。

以上两个系统调用都是用来创建新的 Namespace 的, 而 setns 则可以将进程放到已有的 Namespace 里, 问题是如何指定已有的 Namespace? 答案在 procfs 里。每个进程在 procfs 下都有一个目录, 在那里就有 Namespace 相关的信息, 如下。

```
# ls -l /proc/$$/ns
total 0
lrwxrwxrwx 1 root root 0 Jun 16 14:39 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 net -> net:[4026531957]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 uts -> uts:[4026531838]
```

这里每个虚拟文件都对应了这个进程所处的 Namespace。因此, 如果另一个进程要进

入这个进程的 Namespace，可以通过 `open` 系统调用打开这里的虚拟文件并得到一个文件描述符，然后把文件描述符传给 `setns`，调用返回成功的话，就进入这个进程的 Namespace 了。



提示 `docker exec` 命令的实现原理就是 `setns`。

以下是一个简单的程序，在 Linux 终端调用这个程序就会进入新的 Namespace，同时也可以打开另一个终端，这个终端是在 host 的 Namespace 里，这样就可以对比两个 Namespace 的区别了。

```
#define _GNU_SOURCE
#include <sys/wait.h>
#include <sys/utsname.h>
#include <sched.h>
#include <stdio.h>

#define STACK_SIZE (1024 * 1024)
static char stack[STACK_SIZE];
static char* const child_args[] = { "/bin/bash", NULL };

static int child(void *arg)
{
    execv("/bin/bash", child_args);
    return 0;
}

int main(int argc, char *argv[])
{
    pid_t pid;

    pid = clone(child, stack+STACK_SIZE, SIGCHLD|CLONE_NEWUTS, NULL);

    waitpid(pid, NULL, 0);
}
```

这个程序创建了 UTS Namespace，可以通过修改 flag，创建其他 Namespace，也可以创建几个 Namespace 的组合。这个程序将会用来为下面的内容做演示。

2.4.3 各个 Namespace 介绍

1. UTS Namespace

UTS Namespace 用于对主机名和域名进行隔离，也就是 `uname` 系统调用使用的结构体 `struct utsname` 里的 `nodename` 和 `domainname` 这两个字段，UTS 这个名字也是由此而来的。

那么，为什么要使用 UTS Namespace 做隔离？这是因为主机名可以用来代替 IP 地址，因此，也就可以使用主机名在网络上访问某台机器了，如果不做隔离，这个机制在容器里就会出问题。

调用之前的程序后，在 Namespace 终端执行以下命令：

```
# hostname container
# hostname
container
```

这里已经改变了主机名，现在通过 host 终端来看看 host 的主机名：

```
# hostname
linux-host
```

可以看到，host 的主机名并没有变化，这就是 Namespace 所起的作用。

2. IPC Namespace

IPC 是 Inter-Process Communication 的简写，也就是进程间通信。Linux 提供了很多种进程间通信的机制，IPC Namespace 针对的是 SystemV IPC 和 Posix 消息队列。这些 IPC 机制都会用到标识符，例如用标识符来区别不同的消息队列，然后两个进程通过标识符找到对应的消息队列进行通信等。

IPC Namespace 能做到的事情是，使相同的标识符在两个 Namespace 中代表不同的消息队列，这样也就使得两个 Namespace 中的进程不能通过 IPC 进程通信了。

举个例子，在 namespace 终端创建了一个消息队列：

```
# ipcmk -Q
Message queue id: 65536
# ipcs -q
----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
0x0ec037c7 65536      root       644        0
```

这个消息队列的标识符是 65536，现在在 host 终端看一下：

```
# ipcs -q
----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
```

在这里看不到任何消息队列，IPC 隔离的效果达到了。

3. PID Namespace

PID Namespace 用于隔离进程 PID 号，这样一来，不同的 Namespace 里的进程 PID 号就可以是一样的了。

当创建一个 PID Namespace 时，第一个进程的 PID 号是 1，也就是 init 进程。init 进程有一些特殊之处，例如 init 进程需要负责回收所有孤儿进程的资源。另外，发送给 init 进程的任何信号都会被屏蔽，即使发送的是 SIGKILL 信号，也就是说，在容器内无法“杀死”init 进程。

但是当用 ps 命令查看系统的进程时，会发现竟然可以看到 host 的所有进程：

```
# ps ax
  PID TTY          STAT TIME COMMAND
    1 ?           Ss    0:24 init [5]
    2 ?           S      0:06 [kthreadd]
    3 ?           S      1:37 [ksoftirqd/0]
    5 ?          S<     0:00 [kworker/0:0H]
    7 ?           S      0:16 [kworker/u33:0]
...
7585 pts/0        S+     0:00 sleep 1000
```

这是因为 `ps` 命令是从 `procfs` 读取信息的，而 `procfs` 并没有得到隔离。虽然能看到这些进程，但由于它们其实是在另一个 PID Namespace 中，因此无法向这些进程发送信号：

```
# kill -9 7585
-bash: kill: (7585) - No such process
```

4. Mount Namespace

Mount Namespace 用来隔离文件系统挂载点，每个进程能看到的文件系统都记录在 `/proc/$$/mounts` 里。在创建了一个新的 Mount Namespace 后，进程系统对文件系统挂载 / 卸载的动作就不会影响到其他 Namespace。

之前看到，创建 PID Namespace 后，由于 `procfs` 没有改变，因此通过 `ps` 命令看到的仍然是 host 的进程树，其实可以通过在这个 PID Namespace 里挂载 `procfs` 来解决这个问题，如下：

```
# mount -t proc none /proc
# ps ax
  PID TTY          STAT TIME COMMAND
    1 pts/2        S+     0:00 newns
    3 pts/2        R+     0:00 ps ax
```

但此时由于文件系统挂载点没有隔离，因此 host 看到的 `procfs` 也会是这个新的 `procfs`，这样在 host 上就会出问题：

```
# ps ax
Error, do this: mount -t proc none /proc
```

可如果同时使用 Mount Namespace 和 PID Namespace，新的 Namespace 里的进程和 host 上的进程将会看到各自的 `procfs`，故而也就不存在上面的问题了。

5. Network Namespace

这个 Namespace 会对网络相关的系统资源进行隔离，每个 Network Namespace 都有自己的网络设备、IP 地址、路由表、`/proc/net` 目录、端口号等。网络隔离的必要性是很明显的，举一个例子，在没有隔离的情况下，如果两个不同的容器都想运行同一个 Web 应用，而这个应用又需要使用 80 端口，那就会有冲突了。

新创建的 Network Namespace 会有一个 loopback 设备，除此之外不会有任何其他网络设备，因此用户需要在这里面做自己的网络配置。IP 工具已经支持 Network Namespace，可以通过它为新的 Network Namespace 配置网络功能。首先创建 Network Namespace：

```
# ip netns add new_ns
```

使用“ip netns exec”命令可以对特定的 Namespace 执行网络管理：

```
# ip netns exec new_ns ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

看到确实只有 loopback 这个网络接口，并且它还因处于 DOWN 状态而不可用：

```
# ip netns exec new_ns ping 127.0.0.1
connect: Network is unreachable
```

通过以下命令可以启用 loopback 网络接口：

```
# ip netns exec new_ns ip link set dev lo up
# ip netns exec new_ns ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.053 ms
...
```

最后可以这样删除 Namespace：

```
# ip netns delete new_ns
```

容器的网络配置是一个很大的话题，后面有专门的章节讲解，因此这里暂不展开。

6. User Namespace

User Namespace 用来隔离用户和组 ID，也就是说一个进程在 Namespace 里的用户和组 ID 与它在 host 里的 ID 可以不一样，这样说可能读者还不理解有什么实际的用处。User Namespace 最有用的地方在于，host 的普通用户进程在容器里可以是 0 号用户，也就是 root 用户。这样，进程在容器内可以做各种特权操作，但是它的特权被限定在容器内，离开了这个容器它就只有普通用户的权限了。



注意 容器内的这类 root 用户，实际上还是有很多特权操作不能执行，基本上如果这个特权操作会影响到其他容器或者 host，就不会被允许。

在 host 上，可以看到我们是 lizf 用户。

```
$ id
uid=1000(lizf) gid=100(users) groups=100(users)
```

现在创建新的 User Namespace，看看又是什么情况？

```
$ new-usersns
$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

可以看到，用户名和组名都变了，变成 65534，不再是原来的 1000 和 100。

接下来的问题是，怎么设定 Namespace 和 host 的 UID 的映射关系？方法是在创建新的 Namespace 后，设置这个 Namespace 里进程的 /proc/<PID>/uid_map。在 Namespace 终端看到的是这样的：

```
$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
$ echo $$
17074
$ cat /proc/17074/uid_map
$
```

可以看到 uid_map 是空的，也就是还没有 UID 的映射。这可以在 host 终端上通过 root 用户设置，如下。

```
# echo "0 1000 65536" > /proc/17074/uid_map
```

上面命令表示要将 [1000, 66536] 的 UID 在 Namespace 里映射成 [0, 65536]。再切回到 Namespace 终端看看：

```
$ id
uid=0(root) gid=65534(nogroup) 65534(nogroup)
```

可以看到，我们成功地将 lizf 用户映射成容器里的 root 用户了。对于 gid，也可以做类似的操作。

至此，关于 Namespace 和 Cgroup 的知识就讲解完了，可以看到，Namespace 和 Cgroup 的使用是很灵活的，同时这里面又有不少需要注意的地方，因此直接操作 Namespace 和 Cgroup 并不是很容易。正是因为这些原因，Docker 通过 Libcontainer 来处理这些底层的事情。这样一来，Docker 只需要简单地调用 Libcontainer 的 API，就能将完整的容器搭建起来。而作为 Docker 的用户，就更不用操心这些事情了，而只需要学习 Docker 的使用手册，就能通过一两条简单的 Docker 命令启动容器。

2.5 容器造就 Docker

关于容器是否是 Docker 的核心技术，业界一直存在着争议。有人认为 Docker 的核心技术是对分层镜像的创新使用，有人认为其核心是统一了应用的打包分发和部署方式，为服务器级别的“应用商店”提供了可能，而这将会是颠覆传统行业的举措。事实上，这一系列创新并不是依赖于容器技术的，基于传统的 hypervisor 也可以做到，业界也由此诞生

了一些开源项目，比如 Hyper、Clear Linux 等。另外，Docker 官方对 Docker 核心功能的描述“Build, Ship and Run”中也确实没有体现与容器强相关的内容。

尽管如此，笔者依然认为容器是 Docker 的核心技术之一。

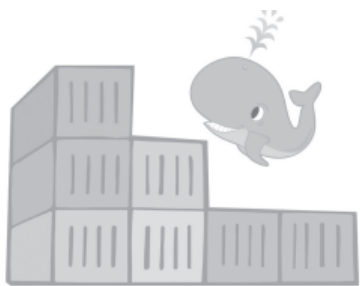
首先从 Docker 的诞生历史上，它主要是为了完善当时不愠不火的容器项目 LXC，使用户可以更方便地使用容器，让容器可以更好地应用到项目开发和部署的各个流程中。从一开始 LXC 就是 Docker 上的唯一容器引擎也可以看出这一点。所以可以说，Docker 是为容器而生的。

另外，更重要的一点，跟 Docker 一起发展和被大家熟知的，还有叫做“微服务”(micro service)的设计哲学，而这会把容器的优势发挥得更加淋漓尽致。容器作为 Linux 平台的轻量级虚拟化，其核心优势是跟内核的无缝融合，其在运行效率上的优势和极小的系统开销，与需要将各个组件单独部署的微服务应用完美融合。而且微服务在隔离性问题上更加可控，这也避免了容器相对传统虚拟化在隔离性上的短板。所以，未来在微服务的设计哲学下，容器必将跟 Docker 一起得到更加广泛的应用和发展。

在理解了容器，理解了容器的核心技术 Cgroup 和 Namespace，理解了容器技术如何巧妙且轻量地实现“容器”本身的资源控制和访问隔离之后，可以看到 Docker 和容器是一种完美的融合和相辅相成的关系，它们不是唯一的搭配，但一定是最完美的组合。与其说是容器造就了 Docker，不如说是它们造就了彼此，容器技术让 Docker 得到更多的应用和推广，Docker 也使得容器技术被更多人熟知。在可预见的未来，它们也一定会彼此促进，共同发展，在全新的解决方案和生态系统中扮演着重要的角色。

2.6 本章小结

本章对容器技术做了详细的剖析，相信读者已经对容器的概念和原理有了清晰的认识，在这样的基础之上，可以更加轻松和深刻地理解 Docker 的一系列技术了，接下来的章节会针对 Docker 中的各个子系统做详细的介绍。



Chapter 3 第 3 章

理解 Docker 镜像

Docker 所宣称的用户可以随心所欲地“Build、Ship and Run”应用的能力，其核心是由 Docker image（Docker 镜像）来支撑的。Docker 通过把应用的运行时环境和应用打包在一起，解决了部署环境依赖的问题；通过引入分层文件系统这种概念，解决了空间利用的问题。它彻底消除了编译、打包与部署、运维之间的鸿沟，与现在互联网企业推崇的 DevOps 理念不谋而合，大大提高了应用开发部署的效率。Docker 公司的理念被越来越多的人理解和认可也就是理所当然的了，而理解 Docker image 则是深入理解 Docker 技术的一个关键点。

本章主要介绍 Docker image 的使用和相关技术细节。其中 3.1 节介绍 Docker image 的基本概念；3.2 节从 image 生命周期的角度介绍其使用方法；3.3 节介绍 Docker image 的组织结构；3.4 节介绍 Docker image 相关的一些扩展知识。

3.1 Docker image 概念介绍

简单地说，Docker image 是用来启动容器的只读模板，是容器启动所需要的 rootfs，类似于虚拟机所使用的镜像。首先需要通过一定的规则和方法表示 Docker image，如图 3-1 所示。

图 3-1 是典型的 Docker 镜像的表示方法，可以看到其被“/”分为了三个部分，其中每部分都可以类比 Github 中的概念。下面按照从左到右的顺序介绍这几个部分以及相关的一些重要概念。

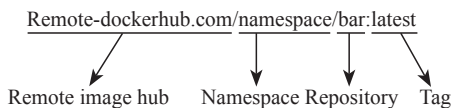


图 3-1 Docker 镜像的典型表示法

- ❑ Remote docker hub：集中存储镜像的 Web 服务器地址。该部分的存在使得可以区分从不同镜像库中拉取的镜像。若 Docker 的镜像表示中缺少该部分，说明使用的是默认镜像库，即 Docker 官方镜像库。
- ❑ Namespace：类似于 Github 中的命名空间，是一个用户或组织中所有镜像的集合。
- ❑ Repository：类似于 Git 仓库，一个仓库可以有多个镜像，不同镜像通过 tag 来区分。
- ❑ Tag：类似 Git 仓库中的 tag，一般用来区分同一类镜像的不同版本。
- ❑ Layer：镜像由一系列层组成，每层都用 64 位的十六进制数表示，非常类似于 Git 仓库中的 commit。
- ❑ Image ID：镜像最上层的 layer ID 就是该镜像的 ID，Repo:tag 提供了易于人类识别的名字，而 ID 便于脚本处理、操作镜像。

镜像库是 Docker 公司最先提出的概念^①，非常类似应用市场的概念。用户可以发布自己的镜像，也可以使用别人的镜像。Docker 开源了镜像存储部分的源代码（Docker Registry 以及 Distribution），但是这些开源组件并不适合独立地发挥功能，需要使用 Nginx 等代理工具添加基本的鉴权功能，才能搭建出私有镜像仓库。本地镜像则是已经下载到本地的镜像，可以使用 `docker images` 等命令进行管理。这些镜像默认存储在 `/var/lib/docker` 路径下，该路径也可以使用 `docker daemon -g` 参数在启动 Daemon 时指定。



提示 Docker 的镜像已经支持更多层级，比如用户的命名空间之前可以包含组织（Remote-dockerhub.com/group/namespace/bar:latest）。但是目前 Docker 官方的镜像库还不具备该能力。

3.2 使用 Docker image

Docker 内嵌了一系列命令制作、管理、上传和下载镜像。可以调用 REST API 给 Docker daemon 发送相关命令，也可以使用 client 端提供的 CLI 命令完成操作。本书的第 7 章会详细阐述 Docker REST API 的细节，本节则主要根据功能对涉及 image 的命令进行说明。下面就从 Docker image 的生命周期角度说明 Docker image 的相关使用方法。

3.2.1 列出本机的镜像

下面的命令可以列出本地存储中镜像，也可以查看这些镜像的基本信息。

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04.2	2103b00b3fdf	5 months ago	188.3 MB
ubuntu	latest	2103b00b3fdf	5 months ago	188.3 MB

^① 见 <https://hub.docker.com>。

ubuntu	trusty	2103b00b3fdf	5 months ago	188.3 MB
ubuntu	trusty-20150228.11	2103b00b3fdf	5 months ago	188.3 MB
ubuntu	14.04	2d24f826cb16	5 months ago	188.3 MB
busybox	buildroot-2014.02	4986bf8c1536	7 months ago	2.43 MB
busybox	latest	4986bf8c1536	7 months ago	2.43 MB

此外，通过 `--help` 参数还可以查询 `docker images` 的详细用法，如下：

```
$ docker images --help

Usage: docker images [OPTIONS] [REPOSITORY]

List images

-a, --all=false      Show all images (default hides intermediate images)
--digests=false     Show digests
-f, --filter=[]      Filter output based on conditions provided
--help=false        Print usage
--no-trunc=false     Don't truncate output
-q, --quiet=false    Only show numeric IDs
```

其中，`--filter` 用于过滤 `docker images` 的结果，过滤器采用 `key=value` 的这种形式。目前支持的过滤器为 `dangling` 和 `label`。`--filter "dangling=true"` 会显示所有“悬挂”镜像。“悬挂”镜像没有对应的名称和 `tag`，并且其最上层不会被任何镜像所依赖。`docker commit` 在一些情况下会产生这种“悬挂”镜像。下面第一条命令产生了一个“悬挂”镜像，第二条命令则根据其特点过滤出该镜像了。图 3-2 中的 `d08407d841f3` 就是这种镜像。

```
$ docker commit 0d6cbf57f660
$ docker images --filter "dangling=true"
REPOSITORY      TAG          IMAGE ID      CREATED        VIRTUAL SIZE
<none>          <none>      d08407d841f3  3 hours ago    2.43 MB
```

在上面的命令中，`--no-trunc` 参数可以列出完整长度的 `Image ID`。若添加参数 `-q` 则会只输出 `Image ID`，该参数在管道命令中很有用处。一般来说悬挂镜像并不总是我们所需要的，并且会浪费磁盘空间。可以使用如下管道命令删除所有的“悬挂”镜像。

```
$ docker images --filter "dangling=true" -q | xargs docker rmi
Deleted: 8a39aa048fe3f2e319651b206073b2a2e437dcf85c15fedb6f437cfe86105145
```

这里的 `--digests` 比较特别，这个参数是伴随着新版本的 Docker Registry V2（即 Distribution）产生的，在本书接下来的第 4 章会详细说明。

按照 Docker 官方路标和最近的动作，Docker 只会保留最核心的 `image` 相关命令和功能，因此那些非核心功能就会被删除。比如 `--tree` 和 `--dot` 已经从 Docker 1.7 中删掉。官方推荐使用 `dockerviz`^① 工具分析 Docker image。执行以下命令，可以图形化地展示 Docker image

① 见 <https://github.com/justone/dockviz>。

的层次关系。

```
# dockviz images -d | dot -Tpng -o images.png
```

执行结果如图 3-2 所示，可以看到，同一个仓库中的镜像并不一定要有特别的关系，比如 ubuntu: 14.04 和 ubuntu:14.04.2 之间就没有共享任何层。

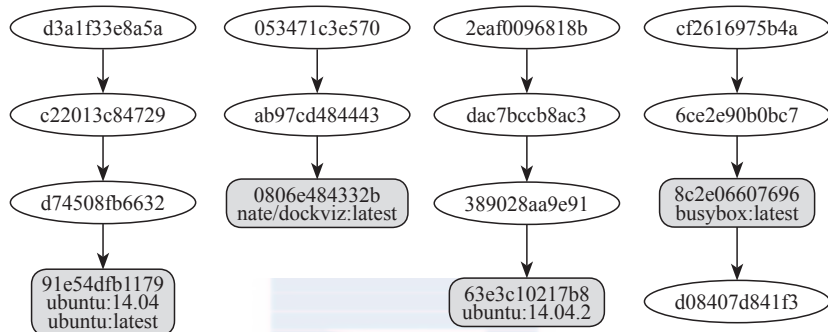


图 3-2 Docker image 层次关系

3.2.2 Build：创建一个镜像

创建镜像是一个很常用的功能，既可以从无到有地创建镜像，也可以以现有的镜像为基础进行增量开发，还可以把容器保存为镜像。下面就详细介绍这些方法。

1. 直接下载镜像

我们可以从镜像仓库下载一个镜像，比如，以下为下载 busybox 镜像的示例。

```
$ docker pull busybox
Using default tag: latest
Pulling repository docker.io/library/busybox
8c2e06607696: Download complete
cf2616975b4a: Download complete
6ce2e90b0bc7: Download complete
Status: Downloaded newer image for busybox:latest
```

具体使用镜像仓库的方法，本书会在后续章节详细描述，这里暂不做说明。

2. 导入镜像

还可以导入一个镜像，对此，Docker 提供了两个可用的命令 `docker import` 和 `docker load`。`docker load` 一般只用于导入由 `docker save` 导出的镜像，导入后的镜像跟原镜像完全一样，包括拥有相同的镜像 ID 和分层等内容。下面的第一行命令可以导出 busybox 为 busybox.tar，第二条命令则是导入该镜像：

```
$ docker save -o busybox.tar busybox
$ docker load -i busybox.tar
```

不同于 `docker load`, `docker import` 不能用于导入标准的 Docker 镜像, 而是用于导入包含根文件系统的归档, 并将之变成 Docker 镜像。

3. 制作新的镜像

前面说过, `docker import` 用于导入包含根文件系统的归档, 并将之变成 Docker 镜像。因此, `docker import` 常用来制作 Docker 基础镜像, 如 Ubuntu 等镜像。与此相对, `docker export` 则是把一个镜像导出为根文件系统的归档。

 **提示** 读者可以使用 Debian 提供的 `Debootstrap` 制作 Debian 或 Ubuntu 的 Base image, 可以在 Docker 官网找到教程[⊖]。

Docker 提供的 `docker commit` 命令可以增量地生成一个镜像, 该命令可把容器保存为一个镜像, 还能注明作者信息和镜像名称, 这与 `git commit` 类似。当镜像名称为空时, 就会形成“悬挂”镜像。当然, 使用这种方式每新增加一层都需要数个步骤 (比如, 启动容器、修改、保存修改等), 所以效率是比较低的, 因此这种方式适合正式制作镜像前的尝试。当最终确定制作的步骤后, 可以使用 `docker build` 命令, 通过 Dockerfile 文件生成镜像。

3.2.3 Ship: 传输一个镜像

镜像传输是连接开发和部署的桥梁。可以使用 Docker 镜像仓库做中转传输, 还可以使用 `docker export/docker save` 生成的 tar 包来实现, 或者使用 Docker 镜像的模板文件 Dockerfile 做间接传输。目前托管在 Github 等项目上的项目, 已经越来越多地包含有 Dockerfile 文件; 同时 Docker 官方镜像仓库使用了 github.com 的 webhook 功能, 若代码被修改就会触发流程自动重新制作镜像。

3.2.4 Run: 以 image 为模板启动一个容器

启动容器时, 可以使用 `docker run` 命令, 该命令在相关章节会详细描述, 本节不做深入说明。

图 3-3 总结了上文提到的 Docker 镜像生命周期管理的相关命令。现阶段 Docker 镜像相关的命令存在一些问题, 包括:

- 命令间逻辑不一致, 比如列出容器使用的是 `docker ps`, 列出镜像使用的是 `docker images`。
- 混用命令导致命令语义不清晰, 比如查看容器和镜像详细信息的命令都是 `docker inspect`。

所以基于这些考虑, Docker 项目的路标中提到会把相关命令归类, 并使用二级命令来管理。因此读者可以着重学习命令的用法和其实现的功能, 不用过分关心命令的形式。

⊖ <https://docs.docker.com/articles/baseimages/>。

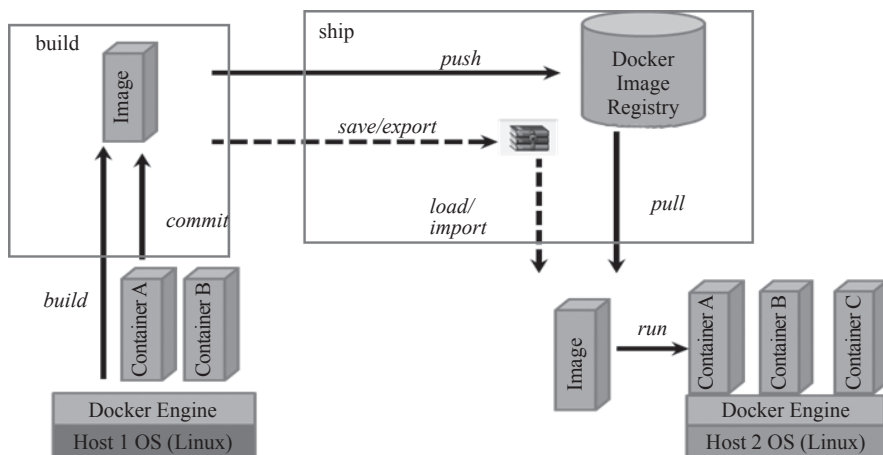


图 3-3 Docker image 生命周期图

3.3 Docker image 的组织结构

上节讲到 Docker image 是用来启动容器的只读模板，提供容器启动所需要的 rootfs，那么 Docker 是怎么组织这些数据的呢？

3.3.1 数据的内容

Docker image 包含着数据及必要的元数据。数据由一层层的 image layer 组成，元数据则是一些 JSON 文件，用来描述数据（image layer）之间的关系以及容器的一些配置信息。下面使用 overlay 存储驱动对 Docker image 的组织结构进行分析，首先需要启动 Docker daemon，命令如下：

```
# docker daemon -D -s overlay -g /var/lib/docker
```

这里从官方镜像库下载 busybox 镜像用作分析。由于前面已经下载过该镜像，所以这里并没有重新下载，而只是做了简单的校验。可以看到 Docker 对镜像进行了完整性校验，这种完整性的凭证是由镜像仓库提供的。相关内容会在后面的章节提到，这里不再展开介绍。

```
$ docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
cf2616975b4a: Already exists
8c2e06607696: Already exists
Digest: sha256:38a203e1986cf79639fb9b2e1d6e773de84002f6ea2d4eb006b52004ee8502d
Status: Image is up to date for busybox:latest
$ docker history busybox    # 为了排版对结果做了一些整理
```

IMAGE	CREATED	CREATED BY	SIZE
8c2e06607696	4 months ago		0 B
6ce2e90b0bc7	4 months ago	/bin/sh -c #(nop) ADD file	2.43 MB
cf2616975b4a	4 months ago	/bin/sh -c #(nop) MAINTAINER	0 B

该镜像包含 cf2616975b4a、6ce2e90b0bc7、8c2e06607696 三个 layer。让我们先到本地存储路径一探究竟吧。

```
# ls -l /var/lib/docker
total 44
drwx----- 2 root root 4096 Jul 24 18:41 containers          # 存放容器运行相关信息
drwx----- 3 root root 4096 Apr 13 14:32 execdriver
drwx----- 6 root root 4096 Jul 24 18:43 graph              # Image 各层的元数据
drwx----- 2 root root 4096 Jul 24 18:41 init
-rw-r--r-- 1 root root 5120 Jul 24 18:41 linkgraph.db
drwxr-xr-x 5 root root 4096 Jul 24 18:43 overlay            # Image 各层数据
-rw----- 1 root root 106 Jul 24 18:43 repositories-overlay # Image 总体信息
drwx----- 2 root root 4096 Jul 24 18:43 tmp
drwx----- 2 root root 4096 Jul 24 19:09 trust            # 验证相关信息
drwx----- 2 root root 4096 Jul 24 18:41 volumes          # 数据卷相关信息
```

1. 总体信息

从 repositories-overlay 文件可以看到该存储目录下的所有 image 以及其对应的 layer ID。为了减少干扰，实验环境之中只包含一个镜像，其 ID 为 8c2e06607696bd4af，如下。

```
# cat repositories-overlay |python -m json.tool
{
  "Repositories": {
    "busybox": {
      "latest": "8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55"
    }
  }
}
```

2. 数据和元数据

graph 目录和 overlay 目录包含本地镜像库中的所有元数据和数据信息。对于不同的存储驱动，数据的存储位置和存储结构是不同的，本章不做深入的讨论。可以通过下面的命令观察数据和元数据中的具体内容。元数据包含 json 和 layersize 两个文件，其中 json 文件包含了必要的层次和配置信息，layersize 文件则包含了该层的大小。

```
# ls -l graph/8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55/
total 8
-rw----- 1 root root 1446 Jul 24 18:43 json
-rw----- 1 root root 1 Jul 24 18:43 layersize
# ls -l overlay/8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55/
total 4
drwxr-xr-x 17 root root 4096 Jul 24 18:43 root
```

可以看到 Docker 镜像存储路径下已经存储了足够的信息，Docker daemon 可以通过这些信息还原出 Docker image：先通过 repositories-overlay 获得 image 对应的 layer ID；再根据 layer 对应的元数据梳理出 image 包含的所有层，以及层与层之间的关系；然后使用联合挂载技术还原出容器启动所需要的 rootfs 和一些基本的配置信息。

3.3.2 数据的组织

从上节看到，通过 repositories-overlay 可以找到某个镜像的最上层 layer ID，进而找到对应的元数据，那么元数据都存了哪些信息呢？可以通过 `docker inspect` 得到该层的元数据。为了简单起见，下面的命令输出中删除了一些与讨论无关的层次信息。



注意 `docker inspect` 并不是直接输出磁盘中的元数据文件，而是对元数据文件进行了整理，使其更易读，比如标记镜像创建时间的条目由 `created` 改成了 `Created`；标记容器配置的条目由 `container_config` 改成了 `ContainerConfig`，但是两者的数据是完全一致的。

```
$ docker inspect busybox:latest
[
{
  "Id": "8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55",
  "Parent": "6ce2e90b0bc7224de3db1f0d646fe8e2c4dd37f1793928287f6074bc451a57ea",
  "Comment": "",
  "Created": "2015-04-17T22:01:13.062208605Z",
  "Container": "811003e0012ef6e6db039bcef852098d45cf9f84e995efb93a176alle9aca6b9",
  "ContainerConfig": {
    "Hostname": "19bbb9ebab4d",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "ExposedPorts": null,
    "PublishService": "",
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": null,
    "Cmd": [
      "/bin/sh",
      "-c",
      "#(nop) CMD [\"/bin/sh\"]"
    ],
  },
  "DockerVersion": "1.6.0",
  "Author": "Jevome Petazzoni <u003cjerome@docker.com>u003e",
  "Config": {
```

```

    "Hostname": "19bbb9ebab4d",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "ExposedPorts": null,
    "PublishService": "",
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": null,
    "Cmd": [
        "/bin/sh"
    ],
    "Architecture": "amd64",
    "Os": "linux",
    "Size": 0,
    "VirtualSize": 2433303,
    "GraphDriver": {
        "Name": "aufs",
        "Data": null
    }
}
]

```

对于上面的输出，有几项需要重点说明一下：

- ❑ **Id**: Image 的 ID。通过上面的讨论，可以看到 image ID 实际上只是最上层的 layer ID，所以 `docker inspect` 也适用于任意一层 layer。
- ❑ **Parent**: 该 layer 的父层，可以递归地获得某个 image 的所有 layer 信息。
- ❑ **Comment**: 非常类似于 Git 的 commit message，可以为该层做一些历史记录，方便其他人理解。
- ❑ **Container**: 这个条目比较有意思，其中包含哲学的味道。比如前面提到容器的启动需要以 image 为模板。但又可以把该容器保存为镜像，所以一般来说 image 的每个 layer 都保存自一个容器，所以该容器可以说是 image layer 的“模板”。
- ❑ **Config**: 包含了该 image 的一些配置信息，其中比较重要的是：“env”容器启动时会作为容器的环境变量；“Cmd”作为容器启动时的默认命令；“Labels”参数可以用于 `docker images` 命令过滤。
- ❑ **Architecture**: 该 image 对应的 CPU 体系结构。现在 Docker 官方支持 amd64，对其他体系架构的支持也在进行中。

通过这些元数据信息，可以得到某个 image 包含的所有 layer，进而组合出容器的 roots，再加上元数据中的配置信息（环境变量、启动参数、体系架构等）作为容器启动时的参数。至此已经具备启动容器必需的所有信息。

3.4 Docker image 扩展知识

Cgroup 和 Namespace 等容器相关技术已经存在很久，在 VPS、PaaS 等领域也有很广泛的应用，但是直到 Docker 的出现才真正把这些技术带入到大众的视野。同样，Docker 的出现才让我们发现原来可以这样管理镜像，可以这样糅合老技术以适应新的需求。Docker 引入联合挂载技术（Union mount）使镜像分层成为可能；而 Git 式的管理方式，使基础镜像的重用成为可能。现在就了解一下相关的技术吧。

3.4.1 联合挂载

联合文件系统这种思想由来已久，这类文件系统会把多个目录（可能对应不同的文件系统）挂载到同一个目录，对外呈现这些目录的联合。1993 年 Werner Almsberger 实现的“*Inheriting File System*”可以看作是一个开端。但是该项目最终废弃了，而后其他开发者又为 Linux 社区贡献了 unionfs（2003 年）、aufs（2006 年）和 Union mounts（2004 年）^①，但都因种种原因未合入社区。直到 OverlayFS^②在 2014 年合入 Linux 主线，才结束了 Linux 主线中无联合文件系统的历史。

这种联合文件系统早期是用在 LiveCD 领域。在一些发行版中我们可以使用 LiveCD 快速地引导一个系统去初始化或检测磁盘等硬件资源。之所以速度很快，是因为我们不需要把 CD 中的信息拷贝到磁盘或内存等可读可写的介质中。而只需把 CD 只读挂载到特定目录，然后在其上附加一层可读可写的文件层，任何导致文件变动的修改都会被添加到新的文件层内。这就是写时复制（copy-on-write）的概念。

3.4.2 写时复制

写时复制是 Docker image 之所以如此强大的一个重要原因。写时复制在操作系统领域有很广泛的应用，fork 就是一个经典的例子。当父进程 fork 子进程时，内核并没有为子进程分配内存（当然基本的进程控制块、堆栈还是需要的），而是让父子进程共享内存。当两者之一修改共享内存时，会触发一次缺页异常导致真正的内存分配。这样做既加速了子进程的创建速度，又减少了内存的消耗（如图 3-4 所示）。

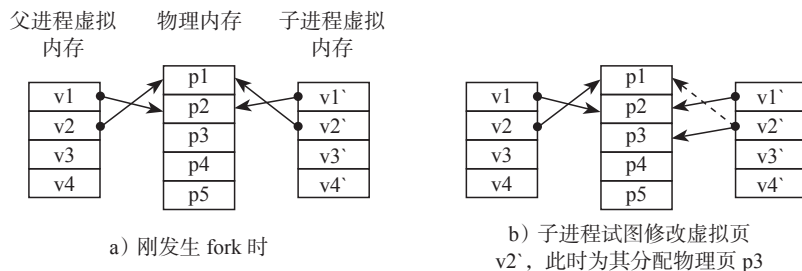


图 3-4 fork 中的写时复制

① <https://lwn.net/Articles/396020/>。

② 现在一般称为 Overlay 文件系统。

Docker image 使用写时复制也是为了达到相同目的：快和节省空间。我们以内核主线中的 OverlayFS 作为例子介绍一下写时复制。

OverlayFS 会把一个“上层”的目录和“下层”的目录组合在一起：“上层”目录和“下层”目录或者组合，或者覆盖，或者一块呈现。当然“下层”目录也可以是联合文件系统的挂载点^①。

首先你需要有支持 OverlayFS 的 Linux 环境（内核 3.18 以上）。Ubuntu 用户可以从 Ubuntu 维护的 kernel 版本^②中下载最新的内核安装包（比如 vivid 版本）。当然也可以手工编译新版的 kernel，但这不是本文的重点，所以暂不细说。下面的测试为了突出变化，删除了无用的文件。

```
$ cat /proc/filesystems | grep overlay
nodev overlay
```

利用上述命令可确定内核支持 OverlayFS。下面以建楼的形式来描述联合文件系统的工作方式，首先需要有混凝土和钢筋等基础原料作为最底层依赖。示例如下：

```
$ mkdir material
$ echo "bad concrete" > material/concrete
$ echo "rebar" > material/rebar
```

但是在建设之前，发现混凝土的质量有问题，所以运来了新的混凝土，同时运来了大理石用作地板砖。示例如下：

```
$ mkdir material2
$ echo "good concrete" > material2/concrete
$ echo "marble" > material2/marble
```

现在已经准备好了建筑所需要的所有材料，下面创建 build 目录作为具体施工的层。另外每个 OverlayFS 挂载点还依赖一些必要的目录，包括 merge（工作目录）、work（OverlayFS 所必须的一个空目录）等，如下：

```
$ mkdir merge work build
$ ls
build material material2 merge work
```

然后挂载 OverlayFS，下面的命令指定了 material 目录为最底层，material2 目录为次底层，build 目录为上层。至此已经完成了建楼所需要的所有依赖。

```
# mount -t overlay overlay -olowerdir= material: material2,upperdir=
build,workdir=work merge
```

① <https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt>。

② <http://kernel.ubuntu.com/~kernel-ppa/mainline/>。

1. 覆盖

现在，在 merge 目录中可以看到混凝土、钢筋和大理石了。并且混凝土是合格的，也就是说 material2 目录中的 concrete 覆盖了 material 目录的对应文件。所以目录所处的层级是很重要的，上层的文件会覆盖同名的下层文件；另外现在的文件系统中会保存两份混凝土数据，所以不合理地修改一个大文件会使 image 的 size 大增。示例如下：

```
$ ls -l */*
-rw-r--r-- 1 root root 19 Aug 31 15:19 material/concrete
-rw-r--r-- 1 root root 12 Aug 31 15:19 material/rebar
-rw-r--r-- 1 root root 20 Aug 31 15:19 material2/concrete
-rw-r--r-- 1 root root 13 Aug 31 16:03 material2/marble
-rw-r--r-- 1 root root 12 Aug 31 15:19 material2/rebar
-rw-r--r-- 1 root root 20 Aug 31 15:19 merge/concrete
-rw-r--r-- 1 root root 13 Aug 31 16:03 merge/marble
-rw-r--r-- 1 root root 12 Aug 31 15:19 merge/rebar
$ cat merge/concrete
good concrete
```

2. 新增

接下来要在 merge 目录下建立我们的建筑框架，此时可以看到 frame 文件出现在了 build 目录中。示例如下：

```
# echo "main structure" >merge/frame
$ ls */* -l
-rw-r--r-- 1 root root 15 Aug 31 17:48 build/frame
-rw-r--r-- 1 root root 19 Aug 31 15:19 material/concrete
-rw-r--r-- 1 root root 12 Aug 31 15:19 material/rebar
-rw-r--r-- 1 root root 20 Aug 31 15:19 material2/concrete
-rw-r--r-- 1 root root 13 Aug 31 16:03 material2/marble
-rw-r--r-- 1 root root 12 Aug 31 15:19 material2/rebar
-rw-r--r-- 1 root root 19 Aug 31 15:19 merge/concrete
-rw-r--r-- 1 root root 15 Aug 31 17:48 merge/frame
-rw-r--r-- 1 root root 13 Aug 31 16:03 merge/marble
```

3. 删除

如果此时客户又提出了新的需求，他们不希望使用大理石地板了，那么我们就得在 merge 目录删掉大理石。可以看到删除底层文件系统中的文件或目录时，会在上层建立一个同名的主设备号都为 0 的字符设备，但并没有直接删掉 marble 文件。所以删除并不一定能减小 image 的大小，并且要注意的是，如果制作 image 时使用到了一些关键的信息（用户名、密码等），则需要同层删除，不然这些信息依然会存在于 image 中。

```
$ rm merge/marble
$ ls -l
c----- 1 root root 0, 0 Aug 31 18:00 build/marble
-rw-r--r-- 1 root root 19 Aug 31 15:19 merge/concrete
-rw-r--r-- 1 root root 15 Aug 31 17:48 merge/frame
```

联合文件系统是实现写时复制的基础。现在社区和操作系统厂家都维护着几种该类文件系统，比如 Ubuntu 系统自带 aufs 的支持，Redhat 和 Suse 则采用的是 devicemapper 方案等。一些文件系统比如 btrfs 也具有写时复制的能力，故也可以作为 Docker 的存储驱动。这些存储驱动的存储结构和性能都有显著的差异[⊖]，所以我们需要根据实际情况选用合理的后端存储驱动。

3.4.3 Git 式管理

Git 是由 Linux 之父 Linus Torvalds 创立的一个开源项目，是一种代码的分布式版本控制工具。因其具有强大的分支能力、便于协作开发等优点而取得了空前的成功，github.com 作为托管代码的仓库也变得越来越流行。两者的合力直接变革了传统的软件托管方案。

Docker 作为新的开源项目，充分借鉴了 Git 的优点（利用分层）来管理镜像，使 image layer 的复用变成了可能，并且类比 Github 提出了 Dockerhub 的概念，一定程度上变革了软件发布流程。

3.5 本章小结

本章主要介绍了 Docker image 的使用方法，另外还介绍了 Docker image 在存储格式和数据上的组织方式，以及一些具体的实现细节。但是这种设计也存在着一些问题需要去克服：

- ❑ image 难以加密。本质上 Docker image 是共享式的，如果我们加密了其中的 layer，那么该层就无法被共享。值得注意的是，Docker 提供了一套基于 notary 的镜像的签名机制，可以一定程度做到镜像的安全分发。
- ❑ image 分层后产生了大量的元数据，不便于存储。现在很多分布式存储对小文件的支持都不是很好。所以搭建私有的镜像仓库时需要选用合理的存储后端。
- ❑ image 制作完成后无法修改。Docker 未提供修改或合并层的命令，因此，如果制作镜像的过程中需要使用到一些隐私信息，则一定要在使用完后立即删除。

[⊖] <http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/>。