Docker 实战

Docker in Action

[美] Jeff Nickoloff 著
胡震 杨润青 黄帅 译

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

目　　录

第 1 部分　保持一台整洁的机器

第 1 章　欢迎来到 Docker 世界 …… 2

1.1　什么是 Docker …… 3

1.1.1　容器 …… 3

1.1.2　容器不是虚拟化 …… 4

1.1.3　在隔离的容器中运行软件 …… 4

1.1.4　分发容器 …… 6

1.2　Docker 解决了什么问题 …… 6

1.2.1　组织有序 …… 7

1.2.2　提高可移植性 …… 8

1.2.3　保护你的机器 …… 9

1.3　为什么 Docker 如此重要 …… 10

1.4　何时何处使用 Docker …… 11

1.5　案例：“Hello World” …… 11

1.6　小结 …… 13

第 2 章　在容器中运行软件 …… 14

2.1　从 Docker 命令行工具获得帮助 …… 14

2.2　控制容器：建立一个网站的监控器 …… 15

2.2.1　创建和启动一个新的容器 …… 16

2.2.2　运行交互式容器 …… 17

2.2.3 列举、停止、重新启动和查看容器输出……18
2.3 已解决的问题和 PID 命名空间……20
2.4 消除元数据冲突：构建一个网站农场……23
2.4.1 灵活的容器标识……24
2.4.2 容器的状态和依赖……26
2.5 构建与环境无关的系统……28
2.5.1 只读文件系统……29
2.5.2 环境变量的注入……31
2.6 建立持久化的容器……34
2.6.1 自动重启容器……35
2.6.2 使用 init 和 supervisor 进程维持容器的运行状态……36
2.7 清理……38
2.8 小结……39
第 3 章 软件安装的简化……40
3.1 选择所需的软件……41
3.1.1 什么是仓库……41
3.1.2 使用标签……42
3.2 查找和安装软件……43
3.2.1 命令行使用 Docker Hub……43
3.2.2 通过网站访问 Docker Hub……45
3.2.3 使用替代注册服务器……47
3.2.4 镜像文件……47
3.2.5 从 Dockerfile 安装……49
3.3 安装文件和隔离……49
3.3.1 镜像层实战……50
3.3.2 分层关系……51
3.3.3 容器文件系统抽象和隔离……52
3.3.4 分层文件系统及其工具的优点……53
3.3.5 Union 文件系统的不足……53
3.4 小结……54

第 4 章　持久化存储和卷间状态共享……55
4.1　存储卷的简介……56
4.1.1　存储卷提供容器无关的数据管理方式……56
4.1.2　NoSQL 数据库使用存储卷……57
4.2　存储卷的类型……60
4.2.1　绑定挂载卷……60
4.2.2　Docker 管理卷……63
4.3　共享存储卷……65
4.3.1　主机依赖的共享……65
4.3.2　共享和 volumes-from 标志……66
4.4　管理卷的生命周期……68
4.4.1　管理卷的权限……68
4.4.2　存储卷的清理……69
4.5　存储卷的高级容器模式……70
4.5.1　卷容器模式……70
4.5.2　数据打包的存储卷容器……72
4.5.3　多态容器模式……73
4.6　小结……74
第 5 章　网络访问……75
5.1　网络相关的背景知识……76
5.1.1　基础：协议，接口和端口……76
5.1.2　高级：网络，NAT 和端口转发……77
5.2　Docker 的网络……79
5.2.1　本地 Docker 网络的拓扑结构……79
5.2.2　四种网络容器原型……80
5.3　Closed 容器……81
5.4　Bridged 容器……83
5.4.1　访问外部网络……84
5.4.2　自定义命名解析……85
5.4.3　开放对容器的访问……88

5.4.4 跨容器通信 ……91
5.4.5 修改网桥接口的配置 ……92
5.5 Joined 容器 ……93
5.6 Open 容器 ……95
5.7 跨容器依赖 ……96
5.7.1 链接——本地服务发现 ……97
5.7.2 链接别名 ……98
5.7.3 环境变量的改动 ……99
5.7.4 链接的本质和缺点 ……101
5.8 小结 ……102
第 6 章 隔离——限制危险 ……103
6.1 资源分配 ……104
6.1.1 内存限制 ……104
6.1.2 CPU ……105
6.1.3 设备的访问权 ……108
6.2 共享内存 ……108
6.2.1 跨容器的进程间通信 ……109
6.2.2 开放内存容器 ……110
6.3 理解用户 ……111
6.3.1 Linux 用户命令空间 ……111
6.3.2 run-as 用户 ……111
6.3.3 用户和卷 ……114
6.4 能力——操作系统功能的授权 ……116
6.5 运行特权容器 ……117
6.6 使用加强工具创建更健壮的容器 ……118
6.6.1 指定额外的安全选项 ……119
6.6.2 微调 LXC ……120
6.7 因地制宜地构建容器 ……121
6.7.1 应用 ……121
6.7.2 高层的系统服务 ……122

6.7.3 低层的系统服务 ……122
6.8 小结 ……122

第 2 部分　镜像发布：如何打包软件

第 7 章　在镜像中打包软件 ……126
7.1 从容器构建镜像 ……126
7.1.1 打包 Hello World ……127
7.1.2 打包 Git ……128
7.1.3 审查文件系统的改动 ……128
7.1.4 Commit——创建新镜像 ……129
7.1.5 可配置的镜像属性 ……130
7.2 深入 Docker 镜像和层 ……131
7.2.1 深入联合文件系统 ……132
7.2.2 重新认识镜像、层、仓库和标签 ……134
7.2.3 镜像体积和层数限制 ……137
7.3 导出和导入扁平文件系统 ……139
7.4 版本控制的最佳实践 ……141
7.5 小结 ……143
第 8 章　构建自动化和高级镜像设置 ……144
8.1 使用 Dockerfile 打包 Git ……144
8.2 Dockerfile 入门 ……148
8.2.1 元数据指令 ……148
8.2.2 文件系统指令 ……152
8.3 注入下游镜像在构建时发生的操作 ……155
8.4 使用启动脚本和多进程容器 ……158
8.4.1 验证环境相关的先决条件 ……158
8.4.2 初始化进程 ……160
8.5 加固应用镜像 ……161
8.5.1 内容可寻址镜像标识符 ……161
8.5.2 用户权限 ……162

8.5.3 SUID 和 SGID 权限……164
8.6 小结……166
第 9 章 公有和私有软件分发……168
9.1 选择一个分发方法……169
9.1.1 分发选项图谱……169
9.1.2 选择标准……169
9.2 通过托管 Registry 发布……172
9.2.1 通过公有仓库发布：你好！Docker Hub……172
9.2.2 使用自动构建发布公有项目……174
9.2.3 私有托管仓库……176
9.3 私有 Registry 介绍……178
9.3.1 使用 Registry 镜像……180
9.3.2 从 Registry 使用镜像……181
9.4 镜像的手动发布和分发……181
9.5 镜像源代码分发工作流程……186
9.6 小结……189
第 10 章 运行自定义 Registry……190
10.1 运行个人 Registry……191
10.1.1 再度介绍镜像……192
10.1.2 介绍 V2 API……193
10.1.3 定制镜像……195
10.2 集中式 Registry 的增强……196
10.2.1 创建一个反向代理……197
10.2.2 在反向代理上配置 HTTPS（TLS）……199
10.2.3 添加身份认证层……202
10.2.4 客户端兼容性……206
10.2.5 应用于生产环境之前……208
10.3 持久化的 BLOB 存储……210
10.3.1 微软 Azure 托管远程存储……211
10.3.2 AWS S3 托管远程存储……212

10.3.3 RADOS（Ceph）的内部远程存储……214
10.4 扩展访问和延迟的改进……215
10.4.1 与元数据缓存集成……215
10.4.2 使用存储中间件简化 BLOB 传输……217
10.5 通过通知集成……219
10.6 小结……224

第 3 部分　多容器和多主机环境

第 11 章　Docker Compose 声明式环境……228
11.1 Docker Compose：第一天的启动并运行……228
11.1.1 用一个简单的开发环境入门……229
11.1.2 一个复杂的架构：分布式系统和 Elasticsearch 的集成……231
11.2 环境内的迭代……233
11.2.1 构建、启动和重新构建服务……234
11.2.2 服务伸缩和删除……237
11.2.3 迭代和持久化状态……238
11.2.4 网络和连接问题……239
11.3 开始一个新项目：三个示例中的 Compose YAML……240
11.3.1 启动前的构建、环境、元数据和网络……240
11.3.2 已知的组件和绑定挂载卷……241
11.3.3 卷容器和扩展服务……242
11.4 小结……243
第 12 章　Docker Machine 和 Swarm 集群……245
12.1 介绍 Docker Machine……246
12.1.1 构建和管理 Docker Machine……246
12.1.2 配置 Docker 客户端与远程 Daemon 工作……249
12.2 Docker Swarm 介绍……252
12.2.1 借助于 Docker Machine 构建 Swarm 集群……252
12.2.2 Swarm 扩展了 Docker 远程 API……255
12.3 Swarm 调度……258

12.3.1 Spread 算法 ····· 258
12.3.2 用过滤器调整调度 ····· 260
12.3.3 BinPack 和随机调度算法 ····· 263
12.4 Swarm 服务发现 ····· 265
12.4.1 Swarm 和单主机网络 ····· 266
12.4.2 服务发现生态系统和权宜之计 ····· 268
12.4.3 展望多主机网络 ····· 269
12.5 小结 ····· 270
后记 ····· 271

第 1 部分

保持一台整洁的机器

隔离是众多计算模式、资源管理策略和一般审计实践的一个核心概念，很难在一开始就编制好一切。学习 Linux 容器如何为运行的程序提供隔离，以及如何使用 Docker 来控制隔离，可以实现惊人的创举：重用、资源有效配置和系统简化。

对这部分内容的深入理解是每个读者掌握不断发展的 Docker 和容器生态系统的坚实基础。像 Docker 工具集本身，这部分提供了可构建的工具块，用来解决更大的问题。出于这个原因，我建议大家尽量不要跳过这部分内容的学习。解决你心中的具体问题可能需要一些时间，但我相信这个过程会让你收获更多的启示。

第 1 章　欢迎来到 Docker 世界

本章介绍

- Docker 是什么
- 容器的简介
- Docker 如何解决大多数人只能忍受的软件问题
- 何时、何地、为何你应该使用 Docker
- 举例："Hello World"

如果我们是相似的人，我猜为了完成那些不愉快或者平庸的任务，你也会喜欢只做必需之事。这可能是因为你同样喜欢使用一个简单易用的工具，来解决那些复杂或费时的工作。如果我猜得没错，我想你会对学习 Docker 感兴趣的。

假设你想尝试一款新的 Linux 软件，但是担心运行起来的是恶意代码。为了保护你的机器，在 Docker 中运行该软件是很好的第一步。因为 Docker 可以帮助大多数普遍软件用户利用好强大的安全工具。

如果你是一名系统管理员，将 Docker 作为软件管理工具集的基础，将节省你的时间，让你专注于高价值的事情，因为 Docker 可以最大限度地减少你花在琐碎工作上的时间。

如果你编写软件，通过 Docker 发布，你的用户会更容易安装并运行它。在 Docker 封装的开发环境中编写软件，将节省配置或共享环境的时间，因为从软件的角度来看，每个环境都是一样的。

假设你拥有或管理大型系统或数据中心。使用 Docker 来创建 build，测试和部署管道会变得很简单，因为通过这样一个管道可以应用到其他任何软件。

2013 年 3 月推出的 Docker，可以和操作系统协作来打包、分发和运行软件。你可以把 Docker 作为软件分发供应商，用来节省你的时间，让你专注于高价值的事情。你可以使用 Docker 构建网络应用，如 Web 服务器、数据库和邮件服务器，也可以构建终端应用程序，比如文本编辑器、编译器、网络分析工具和脚本；在某些情况下，它甚至可用来运行 GUI 程序，如网页浏览器和生产力类的软件。

不仅仅是 Linux Docker 是一款 Linux 软件，但可以很好地运行在大多数操作系统上。

Docker 不是一种编程语言，并且也不是构建软件的框架。Docker 是一个工具，可以帮助解决如安装、拆卸、升级、分发、信任和管理软件等常见问题。它是开源的 Linux 软件，这意味着任何人都可以为之做出贡献，Docker 因此已在诸多方面受益匪浅。公司赞助开源项目的开发也很常见。在这种意义上，Docker 公司（Inc.）是主赞助商。你可以在下面网址找到更多关于 Docker 公司的信息：https://docker.com/company/。

1.1 什么是 Docker

Docker 包括一个命令行程序、一个后台守护进程，以及一组远程服务。它解决了常见的软件问题，并简化了安装、运行、发布和删除软件。这一切能够实现是通过使用一项 UNIX 技术，称为容器。

1.1.1 容器

从历史上看，UNIX 风格的操作系统都使用 *jail* 这个术语来形容一个修改过的运行时环境，以防止该程序访问受保护的资源。自 2005 年以来，Sun 的 Solaris 10 和 Solaris 容器发布后，容器已经成为这样一个运行环境的首选术语。而我们的目标，已经从防止对受保护资源的访问，扩展到隔离所有的资源，除非明确允许。

使用容器已经是很长一段时间的最佳做法。但手动创建容器，仍然具有挑战性，而且很容易出错。错误配置的容器却让他人产生安全的错觉。这个问题直到 Docker 的出现终于得到解决。任何使用 Docker 运行的软件其实是在一个容器内运行。Docker 使用现有的容器引

擎，根据最佳实践提供一致的 Docker 构建方案。这给大家带来了触手可及且更强的安全性。

有了 Docker，用户以更低的成本获得容器。随着 Docker 和容器引擎改进，你获得最新和最好用的“jail”功能。不用再紧跟着迅速发展且高技术性的构建强大应用 jail 的世界了，相反你只要让 Docker 帮你处理这些事情就好。这将节省大量的时间和金钱，并带来心灵的平静。

1.1.2 容器不是虚拟化

在没有 Docker 的时代，商家通常使用硬件虚拟化（也称为虚拟机），以提供隔离。虚拟机提供虚拟的硬件，可安装一个操作系统和其他程序。它们需要很长的时间（通常以分钟计）来创建，也需要显著的资源开销，因为它们除了要执行你需要的软件，还得运行整个操作系统的副本。

不同于虚拟机，Docker 容器不使用硬件虚拟化。运行在 Docker 容器中的程序接口和主机的 Linux 内核直接打交道。因为容器中运行的程序和计算机的操作系统之间没有额外的中间层，没有资源被冗余软件的运行或虚拟硬件的模拟而浪费掉。这是一个很重要的区别。Docker 不是一个虚拟化技术。相反，它可以帮助使用已经内置到操作系统中的容器技术。

1.1.3 在隔离的容器中运行软件

如上所述，容器已经存在了几十年。Docker 使用的是 2007 年就已经成为 Linux 一部分的 Linux 命名空间和 cgroups。Docker 并不提供容器技术，但它使得容器更易于使用。要了解系统中的容器长什么样子，让我们先建立一条基准线。如图 1-1 所示画出了在一个简化的计算机系统体系结构上运行的基本容器示例。

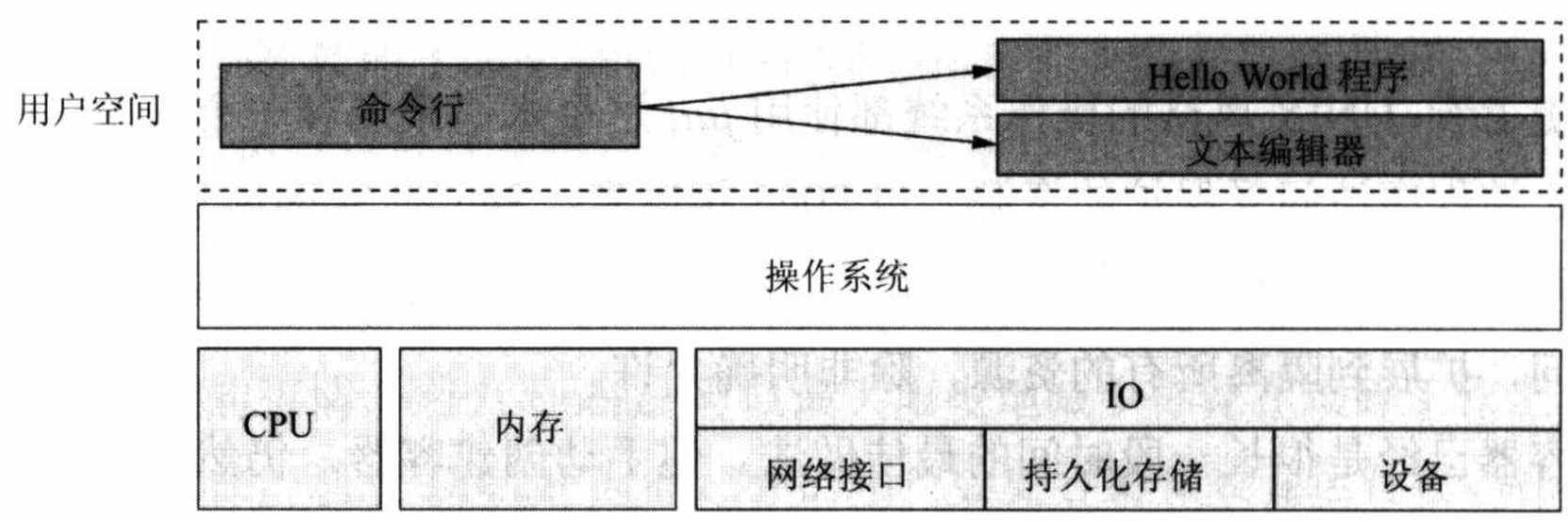


图 1-1 运行由命令行启动的两个应用所需的程序栈

请注意，命令行工具或 CLI 在被称为用户空间的内存中运行，就像是在操作系统上运行的其他程序。在理想情况下，用户空间运行的程序不能修改内核空间的内存。从广义上讲，操作系统是所有用户程序和该计算机上运行的硬件之间的接口。

如图 1-2 所示，运行 Docker 可以认为是在用户空间运行着两个程序。首先是 Docker 守护进程。如果正确安装，这个进程应始终处于运行状态。另一个是 Docker CLI，它是与用户交互的 Docker 程序。如果要启动、停止或安装软件，你可使用 Docker CLI 执行相应的命令。

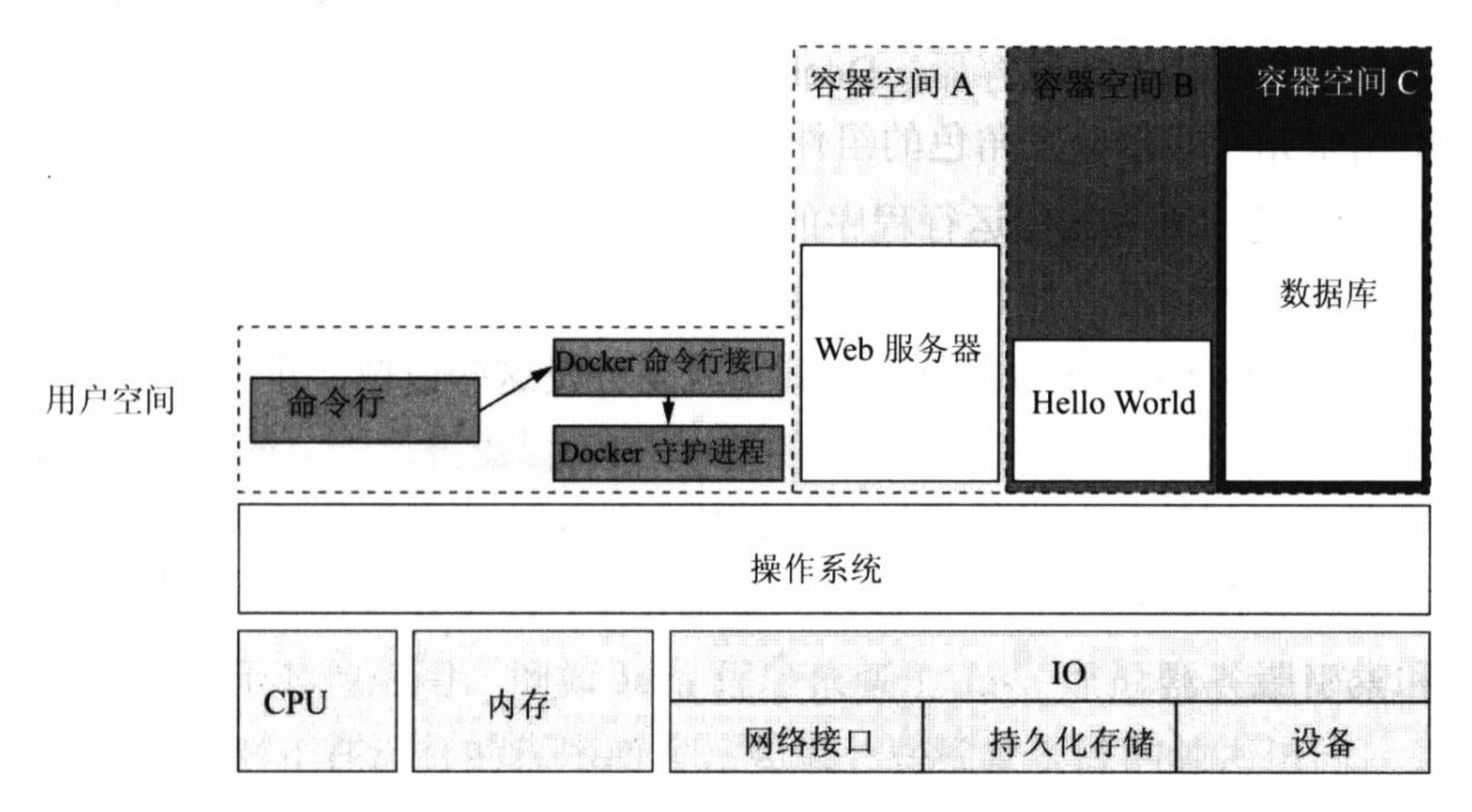


图 1-2 Docker 在 Linux 系统中运行三个容器

图 1-2 也显示了三个运行着的容器。每个都是以 Docker 守护程序的子进程运行，封装在容器中，并授权其在自身用户空间的存储子空间中运行。在容器中运行的程序，只能访问属于自己的该容器审定过的内存空间和资源。

Docker 构建的容器隔离包括 8 个方面。本书的第一部分通过对 Docker 容器功能的探索来覆盖每一个方面。具体如下：

- *PID 命名空间* —— 进程标识符和能力
- *UTS 命名空间* —— 主机名和域名
- *MNT 命名空间* —— 文件系统访问和结构
- *IPC 命名空间* —— 通过共享内存的进程间通信
- *NET 命名空间* —— 网络访问和结构
- *USR 命名空间* —— 用户名和标识
- `chroot()` —— 控制文件系统根目录的位置
- *cgroups* —— 资源保护

Linux 的命名空间和 cgroups 管理着运行时的容器。Docker 采用另一套技术，就像运输集装箱那样来为文件提供容器。

1.1.4 分发容器

你可以把一个 Docker 容器看成是物理运输的集装箱。这就是你存储和运行应用程序及其所有依赖的一个盒子。正如吊车、卡车、火车、船舶可以方便地运输集装箱，所以 Docker 也可以执行、复制和轻松地分发容器。Docker 通过一种打包和分发的软件，完成传统容器的封装。这个用来充当容器分发角色的组件被称为镜像。

Docker 镜像，是一个容器中运行程序的所有文件的捆绑快照。只要你想，你就可以从镜像中创造尽可能多的容器。但是，当你这样做时，从相同的镜像启动的容器不共享文件系统的更改。当你使用 Docker 分发软件，其实就是分发这些镜像，并在接收的机器上创建容器。镜像在 Docker 生态系统中是可交付的基本单位。

Docker 提供了一组简化分发镜像的基础设施组件。这些组件包括*注册服务器*和*索引服务器*。你可以使用由 Docker 公司公开提供的基础设施，以及其他托管公司提供或者自有的注册服务器和索引服务器。

1.2 Docker 解决了什么问题

软件的安装是复杂的。安装之前，你必须考虑使用什么样的操作系统，软件需要的资源，其他已经安装的软件，以及它们的依赖。你需要决定它应该被安装在哪个位置。然后，你需要知道如何安装。安装过程变化如此之大令人惊讶。要考虑的名单又长，又苛刻。软件的安装怎么说也是不一致的，而且过于复杂。

大多数计算机都安装并运行着多个应用程序。大多数应用程序依赖于其他软件。当你想使用的两个或多个应用程序不能在一起工作，会发生什么？这是一个灾难。整个事情会变得很复杂，两个或多个应用程序共享的依赖关系就像：

- 如果一个应用程序需要已升级的依赖，而另一个不需要，会发生什么？
- 当你删除一个应用程序会发生什么？是不是真的删除了？
- 你能删除旧的依赖吗？
- 你还记得现在想的删除软件，还能回退在当时安装时被迫做出的所有改动吗？

简单的事实是：你使用的软件越多，则越难管理。即使你可以花时间和精力找出安装

和运行应用程序的方法，如何能有信心地保证安全呢？开源和闭源代码发布的安全更新持续不断，考虑到所有的问题往往是不可能的。运行的软件越多，它受攻击的风险就越大。

所有这些问题都可以通过仔细核算资源管理来解决，但这也很琐碎，而且并不那么好处理。你的时间应该更好地花在你试图安装、升级或发布的软件上面。构建 Docker 的人都会有此共识，要感谢他们的辛勤工作，让你在任何时候都可用最小的工作量使得这一切变得轻而易举。

可能大多数问题在今天看来还是可接受的。也许是因为你已经习惯了，觉得微不足道。但等你读到如何使用 Docker，使这些问题变得可以解决，很可能就会注意到自己观念开始转变。

1.2.1 组织有序

如果没有 Docker，一台机器运行完看起来就像一个装满垃圾的抽屉。应用程序有各种各样的依赖。一些应用程序依赖于特定的系统库，常见的像语音、网络、图形等。其他依赖于所使用语言的标准库。

某些应用依赖于其他应用，例如 Java 程序依赖于 Java 虚拟机，或 Web 应用程序可能依于一个数据库。对正在运行的程序而言，要求一些稀缺资源的独占访问，如网络连接或文件，是很常见的。

今天，要是没有 Docker，应用遍布整个文件系统，并最终创造出一个相互影响的凌乱关系。如图 1-3 所示为没有 Docker 时示例应用是如何依赖于例库的。

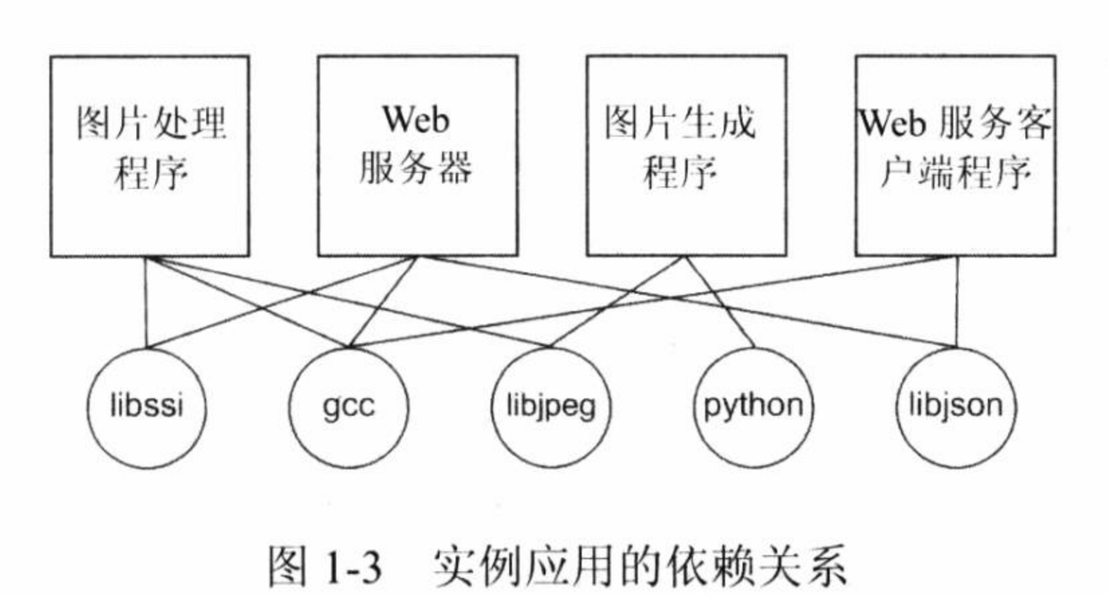


图 1-3 实例应用的依赖关系

Docker 保留了通过容器和镜像进行隔离的所有一切。如图 1-4 所示说明了这些相同的应用程序及其依赖在容器中如何运行。随着链接断裂，以及每个应用程序整齐在列，理解该系统并非难事。

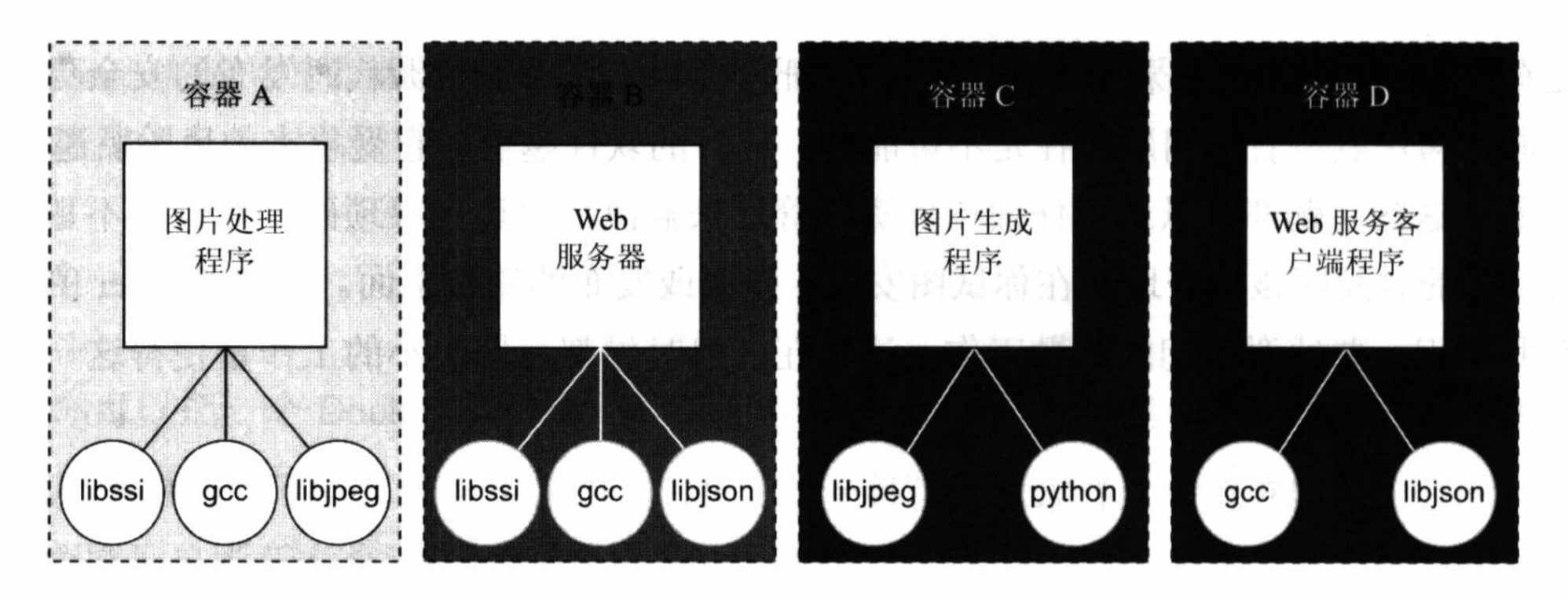


图 1-4 示例的应用及依赖运行在容器中

1.2.2 提高可移植性

另一个软件的问题是：应用程序的依赖关系通常包括特定的操作系统。操作系统之间的可移植性是软件用户的一个主要问题。虽然 Linux 软件和 OS X 之间可能有某种兼容性，但想在 Windows 上使用相同的软件则更加困难。想这样做需要构建整个可移植的软件版本，即使是 Windows 中存在合适的替代依赖。这将是该应用程序维护者的主要开销，而且经常会被忽略。对于用户来说，不幸的是功能强大的软件整体移植太困难，或在他们的系统上根本无法使用。

目前，Docker 可原生运行在 Linux 上，在 OS X 和 Windows 环境中通过单独的虚拟机也可运行。在 Linux 上的这种融合，意味着 Docker 容器运行该软件只需对一组依赖写一次即可。你可能正好想到，“等一下。你刚刚告诉我，Docker 比虚拟机更好。”这也是对的，二者其实是互补的技术。使用虚拟机运行单个程序是浪费的，尤其是当你在同一台计算机上运行多个虚拟机的时候。在 OS X 和 Windows，Docker 使用一个小而单一的虚拟机来运行所有的容器。通过这种方法，运行虚拟机的开销是固定的，而且容器的数量可扩展。

这种新的可移植性在几个方面有助于用户使用。第一，它将软件以前无法使用的地方彻底解锁。第二，它可在任何系统上运行相同的软件。这意味着你的桌面、开发环境、公司的服务器，以及公司的云都可以运行相同的程序。运行一致的环境是非常重要的。这样做有助于最大限度地减少采用新技术的学习曲线。它可以帮助软件开发人员更好地了解将要运行的系统。第三，软件维护人员可以集中精力在单一平台和一套依赖关系中编写他们的软件，这节省了大量的时间，对他们和他们的客户是一个伟大的胜利。

如果没有 Docker 或虚拟机，可移植性一般以单个程序为级别，通过一些常用的工具来实现。例如，Java 程序员可以编写一个程序，在多个操作系统上几乎都能正常工作，因为

程序依赖于一种被称为 Java 虚拟机（JVM）的程序。虽然这在编写软件时是一个适当的做法，其他人、其他公司写了大多数我们使用的软件。例如，如果有，我想用一个流行的 Web 服务器，但它不是用 Java 或其他类似的可移植性语言写的，我怀疑作者并不会花时间来重写。除了这个缺点，语言解释器和软件库都带来依赖的问题。Docker 改善了每个程序的可移植性，无论它用什么语言编写，为什么操作系统而设计，或是在什么样的运行环境下。

1.2.3 保护你的机器

到目前为止，我所提到 Docker 的大多数情况，都是从软件工作的角度来看的，以及在容器外部这样做的好处。但是 Docker 也会保护我们在容器内运行的软件。软件程序可能有各种各样的情况，出现错误或带来安全风险：

- 程序可能是由攻击者编写的。
- 好心的开发人员也会编写出有害的错误程序。
- 程序可能会意外地由于输入处理的故障，被攻击者利用。

无论何种方式，运行软件都将给计算机带来安全威胁。由于运行软件是拥有计算机的主旨，应谨慎地应用实际的降险措施。

像物理的牢房，容器里的任何东西只能访问在它内部的东西。此规则有例外，但仅在用户显式创建时。容器限制了一个程序对其他程序带来的影响范围、可访问的数据和系统资源的影响范围。图 1-5 说明了容器内部运行和外部运行软件的区别。

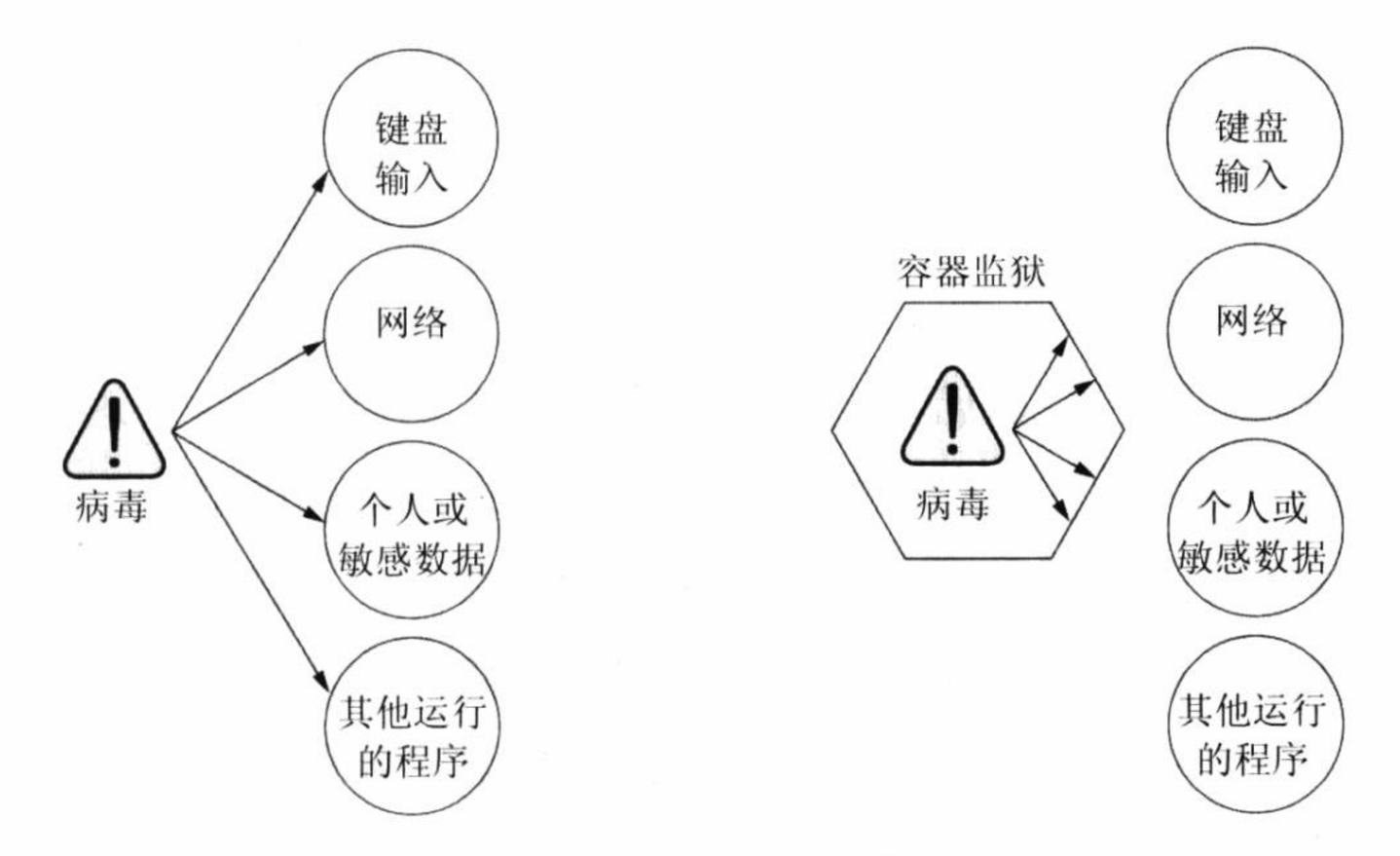


图 1-5 左：恶意程序可直接访问敏感资源

右：容器中的恶意程序

这对你或者你的企业意味着受到安全威胁的范围仅限于所运行的特定应用程序本身。要创建一个强大的应用程序容器是复杂的，也是任何深度防御战略的重要组成。这实在太容易忽略，或很难尽心尽力。

1.3 为什么 Docker 如此重要

第一，Docker 提供了所谓的抽象。抽象允许你以简化的方式处理复杂的工作。所以，在使用 Docker 的前提下，不用再专注于与安装应用程序相关的复杂性和细节，我们需要考虑的是我们想安装什么样的软件。这就像起重机装载航运集装箱到船舶上，用 Docker 安装任何软件的过程都是相同的。该起重机拾取集装箱的形状或装运集装箱的大小可能会不同，但拾取的方式将始终是相同的。而且所有工具对任何分发的容器而言，均可重复使用。

这也适合移除应用时的情况。当你要删除软件时，你只要告诉 Docker 要删除哪个软件。没有散落在外的文件，因为它们都被封装在容器中。你的机器将会变得清洁，就像在安装之前那个样子。

容器抽象和 Docker 所提供的容器管理工具，将改变系统管理和软件开发。Docker 很重要，是因为它使得大家都可以使用容器，用它来节省时间、金钱和精力。

第二，有一个重要的推广使用容器和 Docker 的软件社区。这种推动是如此强大，像亚马逊、微软和谷歌都共同努力，促进其发展，并在自己的云产品中采用它。这些公司已经很难得地走到了一起，支持这个开源项目，而不是开发和发布自己的解决方案。

第三，它为 PC 端所做的像应用商店为移动设备所做的一样。它使软件安装、划分和删除变得十分简单。更妙的是，Docker 有跨平台和开放的特点。试想一下，如果所有主要智能手机享用相同的应用程序商店，这将是一个相当大的市场。有了这项技术，操作系统之间的界限开始变得模糊，在第三方产品的选择中操作系统不再是一个考量因素。

第四，我们终于开始看到通过操作系统实现的更好、更先进的隔离功能。这似乎不起眼，但是不少人正试图让计算机通过操作系统层面的隔离获得更大的安全。很遗憾他们辛勤工作了这么久，才看到大规模的普及。容器以这种或那种形式存在了几十年，Docker 的伟大之处在于，帮我们充分利用这些功能的同时，又没有那些复杂性。

1.4 何时何处使用 Docker

Docker 可以运行在大多数计算机上，无论工作或是在家都可使用。实际上，到底有多普遍呢？

Docker 可以运行在几乎任何地方，但是，这并不意味着你会想这样做。例如，目前 Docker 只能运行 Linux 操作系统上的应用程序。这意味着，如果你想运行 OS X 或 Windows 的本地应用程序，你没有办法通过 Docker 做到。

应用范围限定在 Linux 服务器或桌面上运行的软件，即可以在容器中运行几乎所有的应用程序。这包括服务器应用程序，如 Web 服务器、邮件服务器、数据库、代理服务器等。桌面软件如网络浏览器、文字处理软件、电子邮件客户端或其他工具也很适合。即使是信任的程序也一样危险，就像从互联网上下载后运行的程序会交换用户提供的数据或网络数据。在容器中使用低权限的用户运行，将有助于保护系统免受攻击。

除了增加防守的深度，每天的日常任务使用 Docker 有助于保持计算机的清洁。保持一台整洁的机器能避免共享资源的问题，易于软件安装和移除。同样也会便于计算机安装、移除和分配，简化计算机管理，进而从根本上改变企业维护的方式。

记住一件最重要的事情，要了解容器何时不适用——容器不能改善程序的安全，特别当不得不用最高权限访问计算机的时候。在写本文的时候，这样做是可能的，但很复杂。容器不是安全问题的总解决方案，但它可以用来预防许多类型的攻击。请记住，你不应该使用不受信任来源的软件。如果软件需要管理权限，请弄清楚是不是合理的。也就是说，盲目地将客户提供的容器运行在互相协作的环境中是一个坏主意。

1.5 案例："Hello World"

我喜欢让人们用一个示例开始。按照惯例，我们将使用"Hello World"。在开始之前，先下载并为系统安装 Docker。https://docs.docker.com/installation/上有针对每个可用系统且保持更新的详细说明。OS X 和 Windows 用户需要 Docker 工具箱来安装完整 Docker 套件。当你安装了 Docker 并连接网络，移到命令提示符，然后输入以下命令：

```
docker run dockerinaction/hello_world
```

提示 Docker 以系统 root 用户运行。在某些系统上，你需要使用 sudo 来执行 Docker 命令行工具。如果不这样做会导致权限错误。你可以通过创建一个"Docker"组来避开这个要求，设置该组的 Docker 套接字所有者，并添加用户到该组。详细说

明可查询你所用发行版的在线 Docker 文档，或尝试这两种方法后选择最适合自己的。为了保持一致性，本书将省略 sudo 的前缀。

在此之后，Docker 被激活。它会开始下载各种组件，并最终打印出“Hello World”。再次运行时则只是打印“Hello World”。在这个例子中做了好几个事情，命令本身有几个不同的部分。

第一，可以使用 `docker run` 命令来启动一个新容器。这种单一的命令将触发安装、运行序列（如图 1-6 所示），以及暂停在容器内的程序。

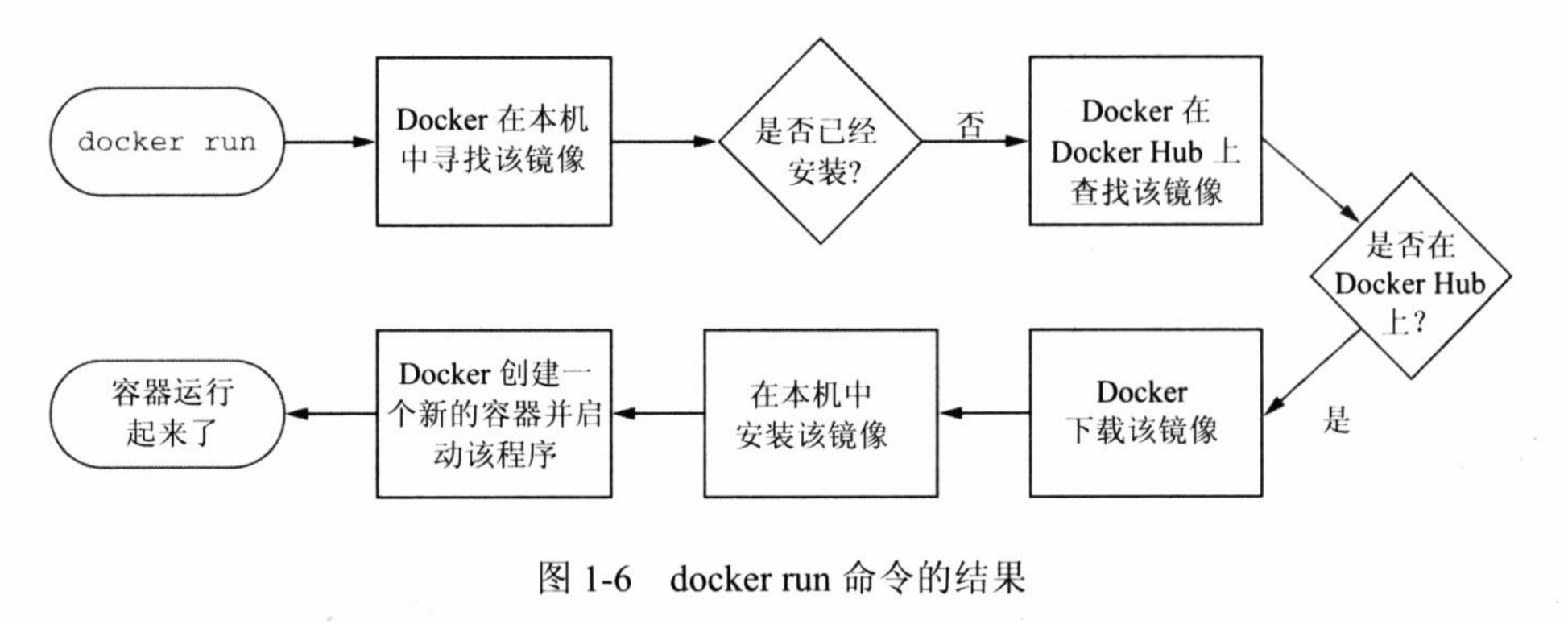


图 1-6 docker run 命令的结果

第二，在容器中运行的程序是 `dockerinaction/hello world`。这就是所谓的仓库（或镜像）的名称。现在，你可以把仓库的名称认为是安装或运行程序的名称。

注：该仓库和其他几个，是专门为了支持在本书中的示例而创建。第 2 部分结束时，你应该会很舒服地来研究这些开源的示例。随时欢迎对它们有任何改进建议。

当你第一次下达命令时，Docker 会先弄清楚 `dockerinaction/hello world` 是否已经安装。如果无法找到（因为这是你使用 Docker 的第一件事），Docker 将问询 Docker Hub。Docker Hub 是 Docker 公司提供的公共注册服务器。Docker Hub 将会答复你的计算器上运行的 Docker，`dockerinaction/hello world` 在哪里可以找到，然后 Docker 开始下载。

一旦安装完毕，Docker 创建一个新的容器，并运行一个命令。在这种情况下，该命令是很简单的：

```
echo "hello world"
```

在命令输出“Hello World”到终端后，它将退出，容器自动停止。要明白容器的运行

状态直接和容器内单次运行的程序状态关联。如果程序在运转，容器运行；如果程序停止，该容器被停止。重新启动容器会再次运行该程序。

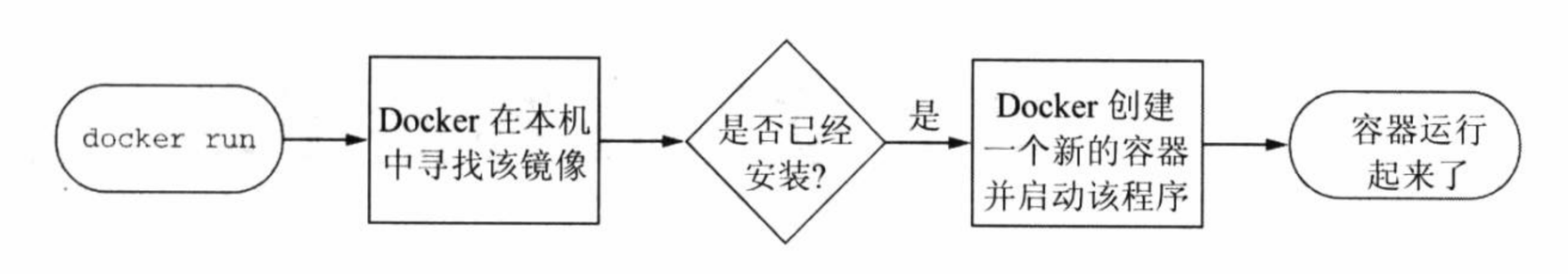


图 1-7 再次运行 docker run，由于镜像已经下载好，Docker 能立即启动新容器

当你第二次执行该命令，Docker 会再次检查是否安装了 dockerinaction/hello world。这一次，它找到并建立了一个新的容器执行。我想强调的一个重要的细节是：当第二次执行 docker run 命令，它会创建来自同一仓库的第二容器（图 1-7 对此有所说明）。这意味着，如果你重复使用 Docker run 命令创造了一群容器，你需要获取已经创建的容器列表，并可能在某个时刻销毁。使用容器工作和创建容器一样简单直接，这两个主题均包含在第 2 章内容中。

恭喜！你现在已是正式的 Docker 用户。花一点时间反思一下整个过程是多么简单。

1.6 小结

本章简要介绍了 Docker，它可以帮助系统管理员、开发人员和其他软件用户解决问题。在本章中，你已经了解到：

- Docker 采用后端的方法来解决常见的软件问题，并简化了安装、运行、发布和删除软件的体验。它由一个命令行程序、一个后台守护进程和一套远程服务组成。它与 Docker 公司提供的社区工具集成在一起。
- 容器的抽象是其后端方法的核心。
- 容器为软件创建了一致的界面和使其能够开发更复杂的工具。
- 容器有助于保持你的机器整洁，因为容器内的软件不能使用容器以外的任何东西，也没有共享的依赖关系。
- 由于 Docker 在 Linux、OS X 和 Windows 上可用并被支持，大多数包装在 Docker 镜像中的软件可以在任何计算机上使用。
- Docker 不提供容器技术，它隐藏了直接和容器软件打交道的复杂性。

第 2 章　在容器中运行软件

本章介绍

- 使用容器执行互动和后台终端程序
- 容器和 PID 命名空间
- 容器配置和输出
- 容器中运行多个程序
- 注入配置到容器
- 持久化容器和容器的生命周期
- 容器的清理

学完本章，你就会明白容器使用的基本知识，以及 Docker 如何帮助解决混乱和冲突的问题。你将通过示例了解 Docker 功能的使用，而这些示例你可能会在日常使用中遇到。

2.1　从 Docker 命令行工具获得帮助

本书下文都将使用 Docker 命令行工具。为了让你学会使用，我想告诉你如何从 Docker 程序本身获得有关命令的信息。这样，你就会明白在你的机器上如何使用合适版本的 Docker。打开一个终端或命令提示符，运行以下命令：

```
docker help
```

运行 docker help 将显示 Docker 命令行工具的基本语法，以及命令的完整列表。尝试一下，片刻后就知道可以做什么了。

docker help 只给你有关哪些命令可用的信息。要获取有关特定命令的详细帮助内容，须增加<COMMAND>命令参数。例如，你可以输入以下命令，来找出如何将一个容器中的文件复制到主机上：

```
docker help cp
```

这将显示 docker cp 命令的使用方式、命令功能的描述，及其参数的详细分析。我想，通过使用下文所介绍的命令，你就能知道如何找到相应的帮助。

2.2 控制容器：建立一个网站的监控器

本书中大多数示例都将使用真实的软件。实际的示例会帮助介绍 Docker 的功能，并说明你将如何在日常活动中使用它们。在第一个示例中，你要安装一个名为 NGINX 的 Web 服务器。Web 服务器使网站文件和程序可通过 Web 浏览器访问。你不是要建立一个网站，而是要安装并使用 Docker 启动 Web 服务器。如果按照此示例中的说明，Web 服务器将只会对你机器上的其他程序开放访问。

假设一个新客户走进你的办公室，以离谱的报价请你为他们打造一个新的网站——并受到相应的监控。这个特别的客户想要经营自己的业务，所以他们会要你提供一个解决方案，可以通过电子邮件，告之他们的团队服务器宕机了。他们还听说，这个流行的 Web 服务器软件叫 NGINX，使用上有特别要求。看了关于 Docker 使用的优劣，你决定将其用于这一项目。如图 2-1 所示为你计划的项目架构。

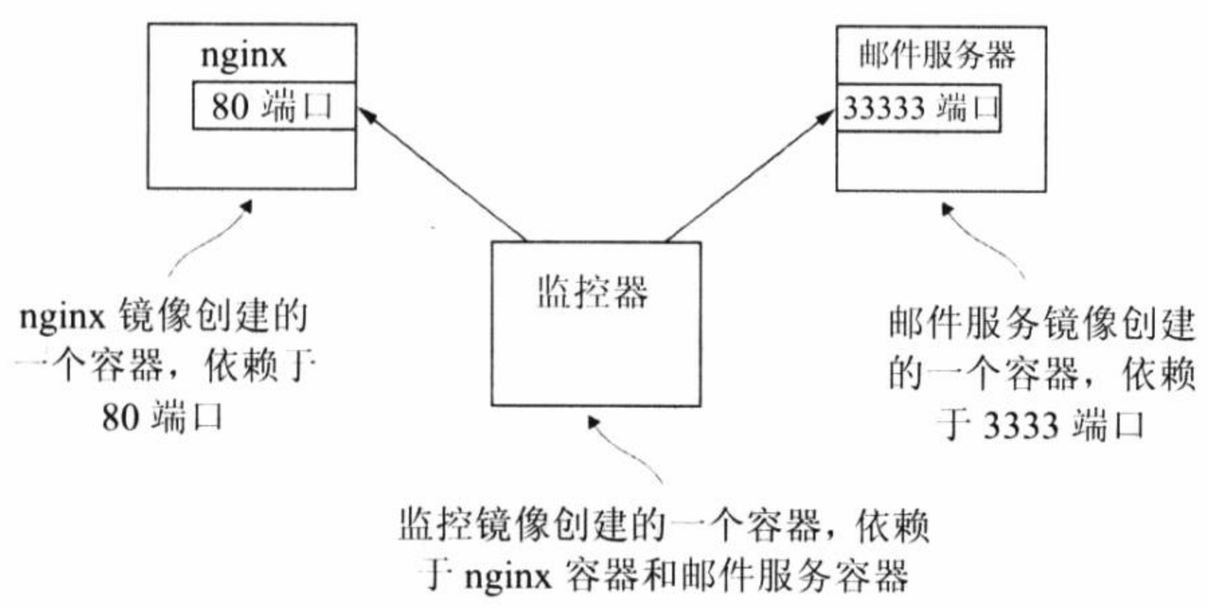


图 2-1 本例中构建的三个容器

本例使用了三个容器：第一个将运行 NGINX；第二个将运行一个邮件程序。这两个容器，都以守护的方式运行。守护意味着它们将在后台运行，而不连接到任何输入或输出流。第三个程序叫作监控器，将在一个交互的容器中运行。对于这个示例，邮件和监控器都是用少量脚本创建的。在本节中，你将学习如何做到以下几点：

- 创建守护式和交互式容器
- 列出系统中的容器
- 查看日志容器
- 暂停并重新启动容器
- 重新连接终端到容器
- 分离已连接的容器

事不宜迟，让我们开始填写客户的订单。

2.2.1 创建和启动一个新的容器

当用 Docker 安装软件时，我们会说自己正在安装镜像。安装镜像的方式不同，镜像来源也有几种。镜像会在第 3 章深入讲解。本例中我们要为 NGINX 从 Docker Hub 下载和安装镜像。请记住，Docker Hub 是 Docker 公司提供的公共注册服务器。这里 Nginx 的镜像，来自 Docker 公司的一个受信仓库。一般情况下，发布该软件的个人或基金会控制该软件的受信仓库。运行以下命令将下载、安装并开始运行 NGINX 的容器：

```
docker run --detach \
    --name web nginx:latest
```

注意守护标记

运行此命令，Docker 将从 Docker Hub（第 3 章会介绍）上的 NGINX 仓库下载、安装 `nginx:latest` 镜像，然后运行该软件。在 Docker 安装好并开始运行 NGINX 后，一行看似随机的字符串将被写入所述终端。看起来像这个样子：

```
7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

这串字符是刚创建运行的 NGINX 容器的唯一标识符。每次你用 `docker run` 创建一个新的容器时，都会得到一个类似的唯一标识符。用户常常捕获该输出，当作其他命令使用的变量。你不需要为这个示例这样做。标识显示出来之后，看不出发生了什么事情。那是因为你使用了`--detach` 选项，并在后台启动该程序。这意味着，程序启动但不会附着到终端。这是有道理的，之所以这样启动 NGINX，是因为我们要运行几个不同的程序。

运行守护式容器非常适合那些在后台静默运行的程序。这类程序被称为守护程序。守

护程序通常通过网络或其他通信工具和其他程序或人进行交互。当你要在后台运行容器的守护程序或其他程序，记得请使用--detach 标志或其缩写形式，如-d。

你的客户需要的另一个守护程序是一个邮件程序。邮件程序等待来自一个呼叫者的连接，然后发送电子邮件。以下命令将安装并运行邮件程序：

```
docker run -d \
    --name mailer \
```

启动守护容器

此命令使用--detach 标志的缩写形式，启动了一个名为 mailer 的新容器。此时，你已经运行两个命令和交付了三分之二客户想要的系统。最后一个组成部分，称为监控器，非常适合交互式容器。

2.2.2 运行交互式容器

与用户交互的程序往往会倾向于体现更强的交互性。基于终端的文本编辑器就是一个很好的示例。Docker 命令行工具是一个很好的交互式终端程序示例。这类程序可能需要用户的输入或终端显示输出。通过 Docker 运行的交互式程序，你需要绑定部分终端到正在运行容器的输入或输出上。

开始使用交互式容器，要运行以下命令：

```
docker run --interactive --tty \
    --link web:web \
    --name web_test \
    busybox:latest /bin/sh
```

创建虚拟终端并绑定标准输入

该命令使用 run 命令的两个标志：--interactive（或-i）和--tty（或-t）。首先，--interactive 选项告诉 Docker 保持标准输入流（stdin，标准输入）对容器开放，即使容器没有终端连接。其次，--tty 选项告诉 Docker 为容器分配一个虚拟终端，这将允许你发信号给容器。这通常就是你想从交互式的命令行工具得到的。当在交互式容器中运行程序，如 shell，通常两种都会用到。

与交互式的标志同样重要的是，当你启动这个容器，得让程序在容器内运行起来。在这种情况下，运行一个叫作 sh 的 shell 程序。这样就可以在容器内运行任何程序。

用交互式容器示例的命令创建一个容器，启动一个 UNIX shell，命令被链接到运行 NGINX 的容器上（链接在第 5 章介绍）。你可以在 shell 上运行一个命令来验证 Web 服务器是否运行正常：

```
wget -O - HTTP://web:80 /
```

这个名为 wget 的程序，将一个 HTTP 请求发送到 Web 服务器（早先在一个容器中启动的 NGINX 服务器），然后在终端上显示网页的内容。在内容的其他行，应该有“Welcome to NGINX!”这样的消息。如果你看到这样的信息，那么一切正常，你可以继续输入 exit 来关闭这个互动容器。这将终止 shell 程序，并停止该容器。

也可以先创建一个交互式容器，然后手动启动该容器中的一个进程，最后再分离终端。你可以通过按住【Crtl】（或【Control】）键，然后按【P】键，接着按【Q】键。只有使用了--tty 选项，以上操作才会生效。

为了完成客户的要求，你需要启动监控器。与过去的示例一样，将测试 Web 服务器，如果 Web 服务器停止，就通过邮件发送消息。此命令将使用缩略标志在交互式容器中启动监控器：

```
docker run -it \
    --name agent \
    --link web:insideweb \
    --link mailer:insidemailer \
    dockerinaction/ch2_agent
```

创建虚拟终端并绑定标准输入

运行之后，该容器将每秒测试 Web 容器，并打印如下所示的消息：

```
System up.
```

现在看到了什么？它从终端分离容器。具体地说，当你启动容器，并开始写“System up”并按住【Ctrl】（或【Control】）键，然后按【P】键，再按【Q】键，之后就会返回到主机的 shell。不要停止程序，否则，监视器将停止检查 Web 服务器。

虽然你会经常使用守护式容器来部署网络服务器，但交互式容器在桌面或服务器手动运行时非常有用。此时，你已经完成了客户需要启动的所有三个应用程序。在宣告大功告成之前，别忘了测试系统。

2.2.3 列举、停止、重新启动和查看容器输出

你需要测试当前设置的第一件事，是通过 docker ps 运行命令检查哪些容器正在运行：

```
docker ps
```

运行该命令会显示每个运行的容器中的以下信息：

- 该容器 ID
- 使用的镜像
- 容器中执行的命令

- 容器运行的时长
- 容器暴露的网络端口
- 容器名

此时，你应该有三个正在运行的容器，名字分别是：web、mailer 和 agent。如果有任何缺失，程序可能已被误停止。这不是一个问题，因为 Docker 有一个命令来重新启动容器。接下来的三个命令使用容器名称重启每个容器。选择合适的命令来重新启动运行容器列表中所缺少的容器。

```
docker restart web
docker restart mailer
docker restart agent
```

现在所有三个容器都运行起来了，你需要测试的是，该系统操作是否正常。要做到这一点，最好的办法是检查每个容器的日志。在 Web 容器中输入：

```
docker logs web
```

会显示一个很长很长的日志，其中几行包含以下子串：

```
"GET / HTTP / 1.0" 200
```

这意味着该 Web 服务器正在运行，并且监控器正在测试该站点。每个监控器测试网站时，这些内容将被写入到日志中。`docker log` 命令可以为这些问题带来帮助，不过依赖于此也是很危险的。该程序写入到标准输出或标准错误输出流的所有内容都会被记录在此日志中。问题是，日志从不轮转或截断，所以写入的日志将持久化保存、持续增长，只要该容器还存在。长期持久性会成为长期进程的一个问题。一个更好的方式是，使用存储卷来处理日志数据，这将在第 4 章中讨论。

你可以仅仅通过检查 Web 服务器的日志来判断该监控器是否在监控 Web 服务器。但为了完整起见，你应该检查邮件服务器和监控器的日志输出：

```
docker logs mailer
docker logs agent
```

对于邮件日志，应该是这个样子：

```
CH2 Example Mailer has started.
```

对于监控器日志，其中有几行就像启动容器时所看到的下面这行：

```
System up.
```

提示 `docker logs` 命令有一个标志，`--follow` 或 `-f`，用来显示整个日志，然后将继续监视和更新日志的显示，不放过任何日志中的变化。完成后，按【Ctrl】（或【Command】）+【C】键中断 `logs` 命令。

现在你已经验证了这些容器的运行状况，也确认监控器可以访问 Web 服务器，你应该测试一下，当 Web 服务器停止时监控器是否会发送通知。当发生这种情况时，监控器应触发邮件服务器，事件将在监控器和邮件服务器日志中被记录。`docker stop` 命令会暂停容器中的 PID #1 程序。用这些命令来测试这个系统：

```
docker stop web
docker logs mailer
```

暂停该容器，即停业 Web 服务

等待几秒种，检查邮件服务日志

查一查邮件服务器的日志，会有以下类似的内容：

```
"Sending email: To: admin@work  Message: The service is down!"
```

这说明监控器成功地检测到 Web 容器中的 NGINX 服务器已经停止。恭喜！你的客户会很开心！你用容器和 Docker 建立了第一个真正的系统。

学习基本的 Docker 功能是一回事，但理解为什么如此用，以及如何使用它建立更复杂的系统完全是另一个任务。开始学习的最佳起点是 Linux 所提供的 PID 命名空间。

2.3 已解决的问题和 PID 命名空间

每一个运行的程序或进程，在 Linux 机器都有一个唯一编号，叫作进程标识符（PID）。一个 PID 命名空间是一组识别进程的数字。Linux 提供了工具可以创建多个 PID 命名空间。每个命名空间拥有一套完整的 PID。这意味着，每个 PID 命名空间将包含其自己的 PID1、2、3，依此类推。从进程的一个命名空间角度来看，PID1 可能是指像 `runit` 或 `supervisord` 这样的 init 系统进程。在不同的命名空间中，PID1 可能是指诸如 `bash` 的 `shell` 命令。为每个容器创建一个 PID 命名空间是 Docker 的关键特征。运行以下命令可以看到：

```
docker run -d --name namespaceA \
    busybox:latest /bin/sh -c "sleep 30000"
docker run -d --name namespaceB \
    busybox:latest /bin/sh -c "nc -l -p 0.0.0.0:80"

docker exec namespaceA ps        ❶
docker exec namespaceB ps        ❷
```

命令❶应该产生类似如下的进程列表：

```
PID   USER     COMMAND
  1   root     /bin/sh -c sleep 30000
  5   root     sleep 30000
  6   root     ps
```

命令❷应该产生一个稍微不同的进程列表：

```
PID   USER     COMMAND
  1   root     /bin/sh -c nc -l -p 0.0.0.0:80
  7   root     nc -l -p 0.0.0.0:80
  8   root     ps
```

在这个示例中，使用 `docker exec` 命令在运行的容器中运行额外的进程。在这种情况下，使用的命令被称为 `ps`，显示所有正在运行的进程和它们的 PID。从输出上很明显地看到，每个容器都有一个带有 PID 1 的进程。

若没有 PID 命名空间，在一个容器内运行的进程将和其他容器或主机共享相同的 ID 空间。这样，容器无法确定其他主机有哪些进程在运行。更糟的是，命名空间将许多授权决策转交域来决定。这意味着，一个容器中的进程可能控制其他容器中的进程。没有 PID 命名空间的 Docker 显得苍白无力。Docker 使用的是 Linux 的功能，如命名空间，可以帮你完整解决这一层面的软件问题。

像大多数 Docker 的隔离功能，你可以有选择地创建没有 PID 命名空间的容器。可以通过在命令 `docker create` 或 `docker run` 中设置`--pid` 标志以及将该值设置为 `host` 来自行尝试。也可以运行 BusyBox 的 Linux 容器和使用 `ps` 命令试试：

```
docker run --pid host busybox:latest ps
```

应该列出本机中运行着的所有进程

回想一下前面 Web 服务器监控的示例。假设你没有使用 Docker，只是在你的机器上直接运行 NGINX。现在假设你忘了已经为另一个项目启动了 NGINX。当再次启动 NGINX，第二个进程将不能访问它需要的资源，因为第一个进程已经占有了这些资源。这是一个软件冲突的基本示例。你可以尝试在同一容器中运行 Nginx 的两个副本看看：

```
docker run -d --name webConflict nginx:latest
docker logs webConflict
docker exec webConflict nginx -g 'daemon off;'
```

输出应为空

在同一个容器中启动第二个 nginx 进程

最后一个命令应该输出如下内容：

```
2015/03/29 22:04:35 [emerg] 10#0: bind() to 0.0.0.0:80 failed (98:
Address already in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)
...
```

第二进程无法正常启动，并报告它所需要的地址已在使用中，这就是所谓的端口冲突。这是真实系统中的一个常见问题，发生在几个进程在同一台机器上运行。下面是一个 Docker 简化并解决这个冲突问题的绝佳示例。在不同的容器运行它们，就像这样：

```
docker run -d --name webA nginx:latest
docker logs webA
docker run -d --name webB nginx:latest
docker logs webB
```

启动第一个 nginx 容器

验证是否工作，输出应为空

启动第二个 nginx 容器

验证是否工作，输出应为空

为了概括程序可能会相互冲突的方式，让我们用停车来比喻。

有偿停车场有几个基本特征：一个支付系统，少数保留车位，以及编过号的停车空间。

将这些特征变成一个支付系统，则需要与特定接口共享一些资源。支付系统可能会接受现金、信用卡或两者兼有。只携带现金的人将无法使用只接受信用卡的车库，人们没有钱支付这笔费用也将无法在车库停放车辆。

同样，某些程序依赖共享组件，如编程语言库的特定版本将不能在这样的环境（已经有一个不同的版本库或者缺乏该库）运行。就像两个人分别使用不同付款方式，希望在只接受一种方式的同一个车库停放，当你想在同一个系统中使用不同版本的库，冲突就会出现。

在这个比喻中预留的空间代表稀缺的资源。试想一下，在车库值班的人员将同一个保留车库分配给两辆车。只要一次只有一个司机想用该车库，就没有问题。但如果两个司机想同时使用该空间，第一个如愿则第二个只能失败。正如你在 2.7 节看到的冲突，当两个程序尝试绑定到同一网络端口上时，则会发生同一类型的冲突。

最后，如果有人在停车场调换了已停车的空间编号，考虑一下会发生什么情况。当车主返回，并试图找到自己的车，他可能无法做到。虽然这是一个愚蠢的示例，但却是一个在共享环境中，环境变量的改变会导致什么问题，一个很好的比喻。应用程序经常使用环境变量或注册表项，以找到它们所需要的其他资源。这些资源可能是库或其他程序。当程序之间发生冲突，它们可能会以不兼容的方式修改这些变量。

下面是一些常见问题的冲突：

- 两个程序想要绑定到相同的网络端口。
- 两个程序都使用相同的临时文件名和文件锁。
- 两个程序想要使用不同版本且全局已安装的库。
- 同一程序的两个副本要使用相同的 PID 文件。
- 第二个安装的程序修改了另一个程序正在使用的环境变量，导致第一个程序中断。

当一个或多个项目都有一个共同的依赖，但不能共享或有不同的需要时，就会产生这些冲突。就像前面提到端口冲突的示例，Docker 通过 Linux 的命名空间、根文件系统和虚拟网络组件等工具解决了这些软件冲突。所有这些工具都用来为每个容器提供隔离。

2.4 消除元数据冲突：构建一个网站农场

下面，你会看到 Docker 如何帮助你避免进程隔离所引发的软件冲突。但是，如果你不小心，最终构建的系统将会引发元数据冲突，或在 Docker 那一层引发容器之间的冲突。

考虑另一个示例，一个客户要求你建立一个系统，为客户承载可变数量的网站，他们也想利用本章前面构建的监控技术。简单地扩展之前你所构建的系统，会是完成这个任务最简单的方式，而不是去定制 NGINX 配置。在这个示例中，你将构建多个容器运行 Web 服务器以及针对每个服务器的监视器。该系统会比较像如图 2-2 所示的架构。

我们的第一反应可能是简单地启动更多的 Web 容器，但并不是那么简单。要认识到容器数量的增加将会使整个问题变得复杂。

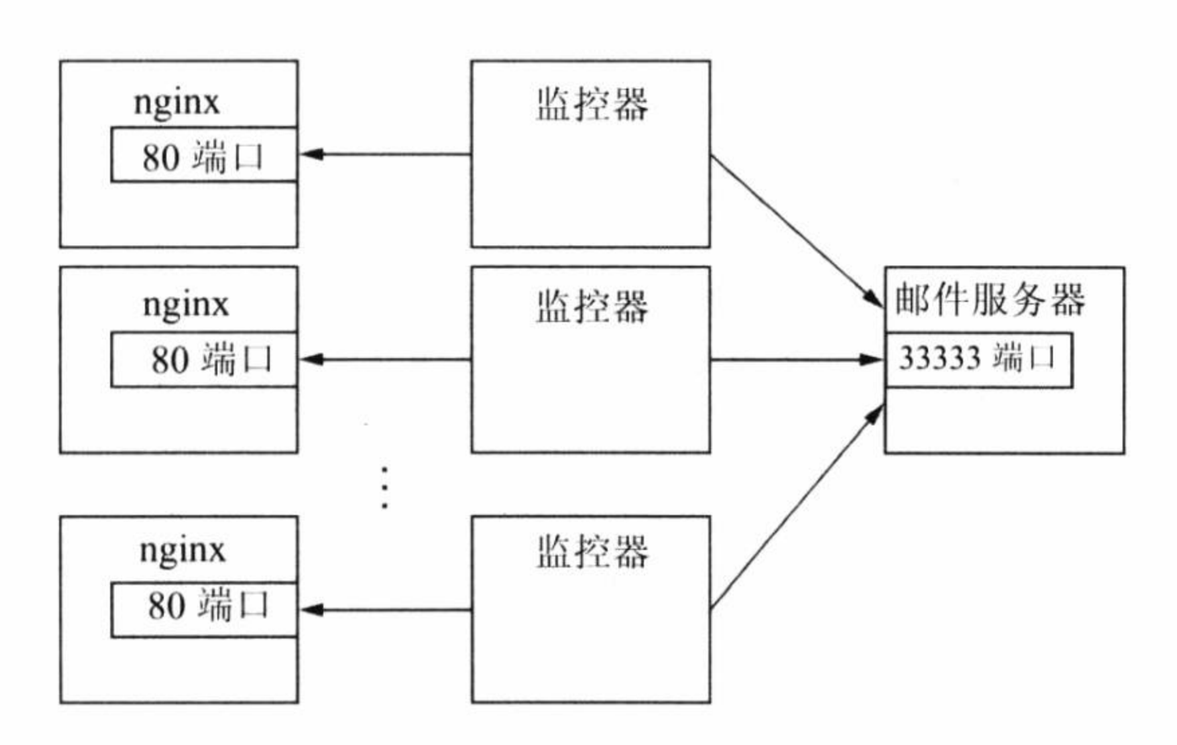


图 2-2 一组 Web 服务器及相应的监控器

2.4.1 灵活的容器标识

在最后一个示例中，简单地创建更多的 NGINX 容器副本是一个坏主意，而要究其原因则最好是去亲身体验：

```
docker run -d --name webid nginx
docker run -d --name webid nginx
```

创建一个名为 Webid 的容器

创建另一个名为 Webid 的容器

这里第二个命令将会失败，并出现冲突的错误：

```
FATA[0000] Error response from daemon: Conflict. The name "webid" is
already in use by container 2b5958ba6a00. You have to delete (or rename)
that container to be able to reuse that name.
```

使用固定的容器名，如 web，在用于实验和文档时很有用，但在多容器的系统中，像这样使用固定的名称，会产生冲突。在默认情况下，Docker 分配一个唯一的（人性化）的名字给它创建的每个容器。该--name 标志只是简单地重写一个已知值的进程。如果出现这样的情况，容器的名称需要改变，你可以随时通过 docker rename 命令重命名该容器：

```
docker rename webid webid-old
docker run -d --name webid nginx
```

重命名当前 Web 容器为 Webid-old

创建另一个名为 Webid 的容器

重命名容器可以减轻一次性命名冲突，但无法用来避免首要问题。除了名称，Docker 分配一个唯一的标识符，在第一个示例中提到过。这是个十六进制编码的 1024 位数字，如下所示：

```
7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

当容器以守护模式启动，其标识符将被打印到终端。你可以在任何命令中使用这些标识符，而不是用容器名来指代特定的容器。例如，你可以使用该 ID 执行 stop 或 exec 命令：

```
docker exec \
   7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5 \
ps
docker stop \
   7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

所生成 ID 在极高概率下是唯一的，这意味着 ID 冲突几乎不可能。也不可能存在该 ID 的前 12 个字符在同一计算机上发生碰撞。所以在大多数 Docker 界面上，你会看到 Docker ID

被截断成为前 12 个字符。这使得生成的 ID 更加人性化。在需要容器标识时，可以使用它们。因此，前面的两个命令可以这样写：

```
docker exec 7cb5d2b9a7ea ps
docker stop 7cb5d2b9a7ea
```

没有一种 ID 特别适于人来使用，但它们很适合脚本和自动化技术。Docker 有多种方法来获取一个容器的 ID，使自动化成为可能。在这些情况下，就可使用完整或截断的数字 ID。

获得容器 ID 的第一种方式是简单地启动或创建一个新容器，将命令的结果赋值给一个 shell 变量。正如前面看到的，当一个新容器以守护模式启动时，容器 ID 将被写入到终端（标准输出）。如果只想在创建容器时得到容器 ID，交互式容器是无法做到的。幸运的是，你可以用另外一个命令创建一个容器而无须启动它。`docker create` 命令和 `docker run` 很类似，主要区别在于该容器是被停止状态创建：

```
docker create nginx
```

结果应该如下所示：

```
b26a631e536d3caae348e9fd36e7661254a11511eb2274fb55f9f7c788721b0d
```

如果你使用的是 Linux shell 命令，类似 sh 或 bash，你可以简单地将结果分配给一个 shell 变量，以后可再次使用它：

```
CID=$(docker create nginx:latest)
echo $CID
```

在 POSIX 兼容的 shell 中才能工作

shell 变量为冲突创造了新的机会，但冲突的范围仅限于终端会话或脚本启动时的进程环境。这些冲突通过环境的管理，应该是容易避免的。这种方法的问题是，如果多个用户或自动过程需要共享该信息，它不会有什么帮助。在这种情况下，你可以使用一个容器 ID（CID）文件。

无论是 `docker run` 还是 `docker create` 命令，都提供了另一种标志，可将新容器 ID 写入到已知文件中：

```
docker create --cidfile /tmp/web.cid nginx
cat /tmp/web.cid
```

创建一个尚未启动的容器

检查该文件

就像 shell 变量的使用，这个功能增加了冲突的机会。该 CID 文件的名字（在 `--cidfile` 之后提供）必须是已知的或具备某些已知结构。就像手动命名容器，这种做法在全局（Docker 范围内）的命名空间使用已知的名称。好消息是，如果该文件已经存在，Docker 将不会使

用所提供的 CID 文件创建一个新的容器。仅当你创建两个同名的容器时，该命令将失败。

使用 CID 文件而不是文件名的另一个原因是，CID 文件可以很容易地与容器一起共享并重命名该容器。Docker 这个功能，称为卷，这会在第 4 章讨论。

提示 处理 CID 文件命名冲突的一种策略是利用已知的或可预见的路径约定来分割空间。例如，在这种情况下，你可以指定一个已知目录作为父目录，进一步通过客户 ID 将这个已知目录划分成多个子目录来存放 web 容器。这将会产生诸如 /containers/web/customer1/web.cid 或 /containers/web/customer8/web.cid 的路径。

你可以使用其他的命令，像 `docker ps` 来获得容器 ID。例如，你想获得最后创建的那个容器的截断 ID，可以这么做：

```
CID=$(docker ps --latest --quiet)
echo $CID

CID=$(docker ps -l -q)
echo $CID
```

在 POSIX 兼容的 shell 中才能工作

用缩写标记再运行一次

提示 如果你想获得完整的容器 ID，你可以使用 `docker ps` 命令的 `--no-trunc` 选项。

到目前为止，这个功能主要用于自动化。但是，即使截断有帮助，这些容器的 ID 还是难于阅读或记住。由于这个原因，Docker 也为每个容器生成了一个可读的名称。

命名约定使用个性化的形容词、下画线，以及一个有影响力的科学家、工程师、发明家或其他类似的思想领导者的姓氏。生成的名称，如 compassionate_swartz、hungry_goodall 和 distracted_turing。这些更具可读性和方便记忆。当你用 Docker 工具直接工作时，你总是可以使用 `docker ps` 命令来查找人性化的名称。

容器的识别可能会非常棘手，但你可以通过使用 Docker ID 和名称生成功能来解决问题。

2.4.2 容器的状态和依赖

有了这个新的知识，新的系统可能看起来是这样的：

```
MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)
WEB_CID=$(docker create nginx)

AGENT_CID=$(docker create --link $WEB_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)
```

确认第一个示例中的邮件服务仍在运行中

这个命令片段可以用来完成一个新的脚本，为每个客户启动新的 NGINX 和监控器实例。你可以使用 docker ps 看到，它们已经建好了：

```
docker ps
```

无论是 NGINX 或是代理，容器状态都不被列入在输出中。 Docker 容器总是会在四个状态中，并按照如图 2-3 所示通过命令转变。

你启动的新容器没有出现在容器的列表中，因为 docker ps 仅显示默认运行的容器。那些被 docker create 指定创建的容器从未启动（已退出状态）。想看到所有的容器（包括那些在退出状态的），使用-a 选项：

```
docker ps -a
```

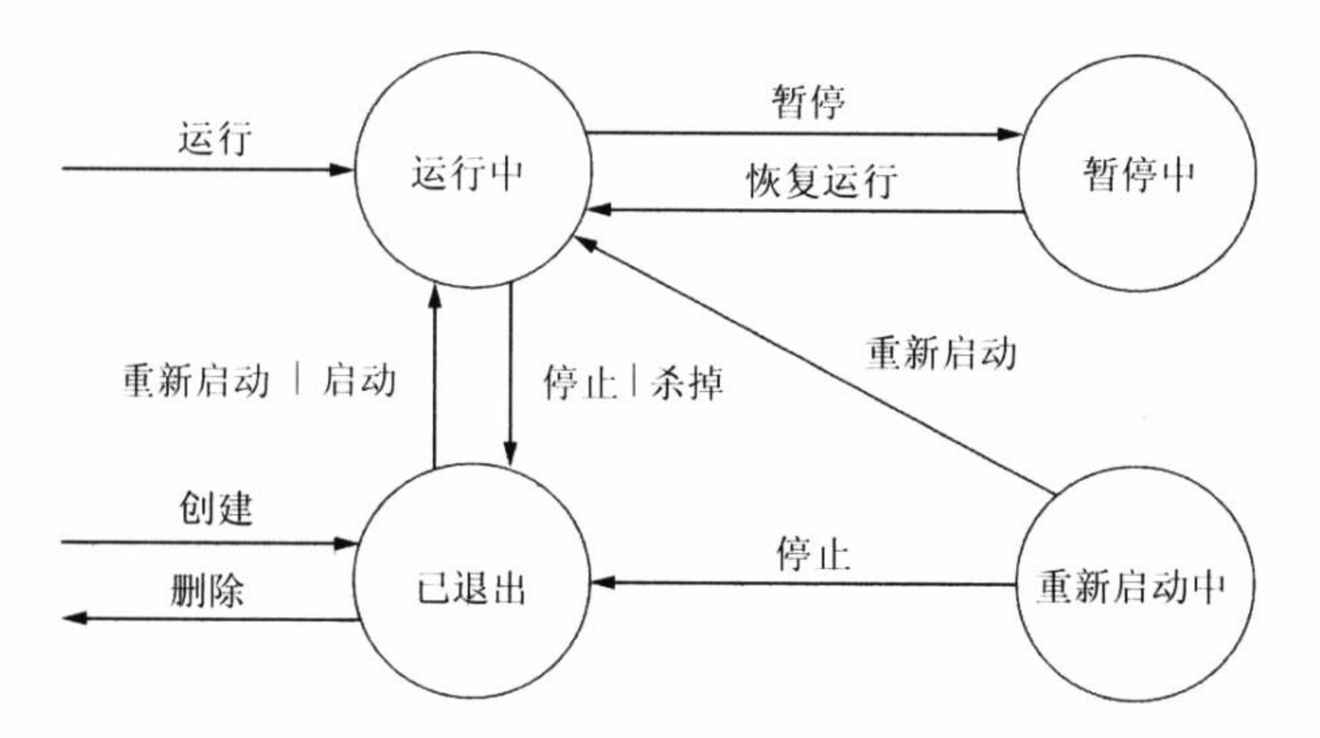


图 2-3 Docker 容器的状态转移图

现在，你已经确定创建了这两个容器，需要启动它们，可以使用 docker start 命令：

```
docker start $AGENT_CID
docker start $WEB_CID
```

运行这些命令将导致一个错误。容器需要以其依赖关系链相反的顺序来启动。因为你试图在 web 容器之前启动监控器容器，Docker 将报告这样的消息：

```
Error response from daemon: Cannot start container 03e65e3c6ee34e714665a
8dc4e33fb19257d11402b151380ed4c0a5e38779d0a: Cannot link to a non running
container: /clever_wright AS /modest_hopper/ insideweb
FATA[0000] Error: failed to start one or more containers
```

在本示例中，监控器容器依赖于 web 容器。你需要首先启动 web 容器：

```
docker start $WEB_CID
```

```
docker start $AGENT_CID
```

当你考虑链接的内在机理时，这很有意义。链接的机制将 IP 地址注入所依赖的容器，在运行的容器中得到该 IP 地址。如果你尝试启动一个容器，其依赖于另一个并未运行的容器，Docker 将不会将该容器的 IP 地址注入到未运行的容器中。Docker 链接机制将在第 5 章讲解，但对容器的启动而言，还是很有用的。

无论你使用 docker run 或 docker create，由此产生的容器需要以其依赖关系链相反的顺序来启动。这意味着，循环依赖是不可能使用 Docker 容器关系来构建的。

在这一点上，你可以把这一切融合在一起，成为一个简洁的脚本，如下所示：

```
MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)

WEB_CID=$(docker run -d nginx)

AGENT_CID=$(docker run -d \
    --link $WEB_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)
```

现在，这个脚本能够无一例外地为每个客户创建可运行的新网站，你该充满信心了吧。客户已经回来了，感谢你到目前为止完成的网络和监测工作，但是事情发生了变化。

他们已经决定把重点放在通过 WordPress（一个流行的开源内容管理和博客程序）建立自己的网站。幸运的是，WordPress 已通过 Docker Hub 名为 WordPress:4 的仓库发布。你需要交付的是一组命令，可以提供一个新的 WordPress 网站，和已交付的相比，它具有同样的监控命令和报警功能。

关于内容管理系统和其他状态系统，有趣的是，它们使用的数据使各种运行的程序变得专门化。Adam 的 WordPress 博客和 Betty 的 WordPress 博客是不同的，即使它们正在运行相同的软件。唯一的不同是内容。即使内容是一样的，因为它们在不同的网站上运行，还是不同的。

如果你建立的系统或软件，知道了太多关于它们的环境——如地址或依赖服务的固定位置，改变环境或再使用这个软件将变得艰难。在合同完成之前，你需要交付一个最大限度减少环境依赖的系统。

2.5 构建与环境无关的系统

很多的工作和软件安装或维护机器有关，这些工作还处理环境的特殊性。这些特殊性

作为全局范围的依赖关系（如已知主机文件系统的位置）、硬编码的部署架构（代码或配置的环境检查），或数据局部性（存储在特定的不在部署体系结构以内的机器上的数据）。认识到这一点，如果你的目标是建立低维护的系统，你应该努力减少这些事情。

Docker 有三个特定的功能，以帮助建立与环境无关的系统：

- 只读文件系统
- 环境变量注入
- 存储卷

处理卷是一个大主题，也是第 4 章的主题。为了学习前两个功能，在本章的其余部分将改变对示例的需求。

WordPress 使用一个名为 MySQL 的数据库程序来存储大部分数据，所以先确保运行 WordPress 的容器是只读文件系统，是一个好主意。

2.5.1 只读文件系统

使用只读文件系统产生以下两个积极效果。首先，你对容器不能更改它所包含的文件产生信心；其次，你也会进一步树立信念：容器中的攻击者无法破坏文件。

为了在客户系统上使用`--read-only`标志，从 WordPress 镜像创建和启动一个容器：

```
docker run -d --name wp --read-only wordpress:4
```

完成这些步骤后，检查容器是否正在运行。你可使用任何先前介绍的方法，也可以直接检查容器元数据。如果指定的 wp 容器正在运行，以下命令将输出为真，否则为假。

```
docker inspect --format "{{.State.Running}}" wp
```

`docker inspect` 命令将显示 Docker 为该容器保留的所有元数据（一个 JSON 文件）。格式选项会改变元数据。除了该容器的运行状态，本例中其会滤除元数据的所有字段。这个命令将简单地输出为错误。

在这种情况下，容器没有运行。为了找出原因，检查容器中的日志：

```
docker logs wp
```

输出是这样的：

```
error: missing WORDPRESS_DB_HOST and MYSQL_PORT_3306_TCP environment
variables
Did you forget to --link some_mysql_container:mysql or set an external db
```

```
with -e WORDPRESS_DB_HOST=hostname:port?
```

看起来 WordPress 有一个 MySQL 数据库的依赖关系。数据库是一个程序，根据检索和搜索方式存储数据。好消息是，你可以使用 Docker 安装 MySQL，就像安装 WordPress：

```
docker run -d --name wpdb \
    -e MYSQL_ROOT_PASSWORD=ch2demo \
    mysql:5
```

一旦启动后，会创建一个不同的 WordPress 容器，并链接到这个新的数据库容器中（链接机制在第 5 章深入讨论）：

```
docker run -d --name wp2 \
    --link wpdb:mysql \
    -p 80 --read-only \
    wordpress:4
```

为数据库创建一个链接

使用唯一的容器名

再检查一次，WordPress 是否正常运行：

```
docker inspect --format "{{.State.Running}}" wp2
```

你可以得知 WordPress 失败后重新启动。检查日志以确定原因：

```
docker logs wp2
```

应该有类似以下的日志：

```
... Read-only file system: AH00023: Couldn't create the rewrite-map mutex
(file /var/lock/apache2/rewrite-map.1)
```

同样是 WordPress 失败后重新启动，但这次的问题是，它试图写一个锁定文件到特定的位置。这是启动的必要组成部分，并无特殊。在这个示例中，需要为只读文件系统增加异常处理。你需要使用卷来做这个异常处理。使用下面的命令启动 WordPress，不会有任何问题：

```
# Start the container with specific volumes for read only exceptions
docker run -d --name wp3 --link wpdb:mysql -p 80 \
    -v /run/lock/apache2/ \
    -v /run/apache2/ \
    --read-only wordpress:4
```

为可写空间创建指定的存储卷

更新版本后你能用的脚本看上去像是这样：

```
SQL_CID=$(docker create -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5)


docker start $SQL_CID
```

```
MAILER_CID=$(docker create dockerinaction/ch2_mailer)
docker start $MAILER_CID

WP_CID=$(docker create --link $SQL_CID:mysql -p 80 \
    -v /run/lock/apache2/ -v /run/apache2/ \
    --read-only wordpress:4)

docker start $WP_CID

AGENT_CID=$(docker create --link $WP_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)

docker start $AGENT_CID
```

恭喜你，现在你应该有了一个正在运行的 WordPress 容器！通过使用只读文件系统，以及链接 WordPress 到另一个运行着数据库的容器，可以确保运行 WordPress 镜像的容器永远不会改变。这意味着，如果运行客户的 WordPress 博客程序的机器出了问题，可轻松在其他地方启动该容器的另一个副本。

但是这一设计有两个问题。首先，数据库和 WordPress 的容器运行在同一个机器上。其次，WordPress 对重要的设置，如数据库名称、管理用户、管理密码、数据库加盐等使用默认值。为了解决这个问题，你可以创建多个版本的 WordPress 软件，每一个客户都有一个特殊的配置。这样做会使简单的配置脚本变成一个在创建镜像时写入文件的怪物。通过使用环境变量来注入配置则是一个更好的方式。

2.5.2 环境变量的注入

环境变量是通过其执行上下文提供给程序的键值对。它可以让你在改变一个程序的配置时，无须修改任何文件或更改用于启动该程序的命令。

Docker 使用环境变量来传达相关信息，包括容器的守护选项、容器的主机名，以及其他在容器中运行程序的实用信息。Docker 还为用户提供了一个机制，可将环境变量注入到一个新的容器。那些期望通过环境变量获取重要信息的程序，可在容器创建时就进行配置。幸运的是，WordPress 就是这样一个程序。

在深入 WordPress 的细节之前，尝试注入和观察你的环境变量。UNIX 命令 env 显示当前执行上下文（终端）中的所有环境变量。要查看注入的环境变量，请使用以下命令：

--env 标志或-e 缩写，可用于注入任何环境变量。如果变量已经由镜像或 Docker 设置，则该值将被覆盖。容器内部运行程序的方式，可以依赖于这种变量。WordPress 遵循以下环境变量：

- WORDPRESS_DB_HOST
- WORDPRESS_DB_USER
- WORDPRESS_DB_PASSWORD
- WORDPRESS_DB_NAME
- WORDPRESS_AUTH_KEY
- WORDPRESS_SECURE_AUTH_KEY
- WORDPRESS_LOGGED_IN_KEY
- WORDPRESS_NONCE_KEY
- WORDPRESS_AUTH_SALT
- WORDPRESS_SECURE_AUTH_SALT
- WORDPRESS_LOGGED_IN_SALT
- WORDPRESS_NONCE_SALT

提示 这个示例忽略了 KEY 和 SALT 变量，但任何实际的生产系统绝对应该设置这些值。

开始之前，你应该解决数据库和 WordPress 容器在同一个机器上运行的问题。不是使用链接来满足 WordPress 的数据库依赖，而是注入 WORDPRESS_DB_HOST 变量的值：

```
docker create --env WORDPRESS_DB_HOST=<my database hostname> wordpress:4
```

这个示例会创建（并不启动）WordPress 容器。不管你在<my database hostname>中指定的是什么，该容器都会试着连上一个 MySQL 数据库。

由于远程数据库可能不会使用任何默认的用户名和密码，你就必须同时注入这些设置。假设数据库管理员是养猫爱好者且痛恨强密码：

```
docker create \
```

```
--env WORDPRESS_DB_HOST=<my database hostname> \
--env WORDPRESS_DB_USER=site_admin \
--env WORDPRESS_DB_PASSWORD=MeowMix42 \
wordpress:4
```

使用环境变量注入这一方法，能帮助你区分 WordPress 容器和 MySQL 容器之间的物理联系。由于数据库和客户的 WordPress 网站都在同一台机器上，你仍然需要解决前面提到的第二个问题——所有的网站都使用相同的默认数据库名称。你需要为每一个独立的站点设置数据库名称以环境变量的方式注入：

```
docker create --link wpdb:mysql \
    -e WORDPRESS_DB_NAME=client_a_wp wordpress:4      ← 针对 client A

docker create --link wpdb:mysql \
    -e WORDPRESS_DB_NAME=client_b_wp wordpress:4      ← 针对 client B
```

你已经解决了这些问题，现在可以修改配置脚本了。首先，设置机器只能运行一个 MySQL 容器：

```
DB_CID=$(docker run -d -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5)
MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)
```

那么网站配置脚本将是这样的：

```
if [ ! -n "$CLIENT_ID" ]; then                  ← 假定$CLIENT_ID 变量已设置
    echo "Client ID not set"                      为脚本的输入
    exit 1
fi

WP_CID=$(docker create \
    --link $DB_CID:mysql \                      ← 使用 DB_CID 创建链接
    --name wp_$CLIENT_ID \
    -p 80 \
    -v /run/lock/apache2/ -v /run/apache2/ \
    -e WORDPRESS_DB_NAME=$CLIENT_ID \
    --read-only wordpress:4)

docker start $WP_CID

AGENT_CID=$(docker create \
    --name agent_$CLIENT_ID \
    --link $WP_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)

docker start $AGENT_CID
```

这个新脚本将为每一位客户启动 WordPress 实例和监控器，并将这些容器以及一个单

独的邮件程序和 MySQL 数据库彼此连接。WordPress 容器在被销毁、重新启动、升级的同时，完全不用担心数据会丢失。如图 2-4 所示就是这种架构。

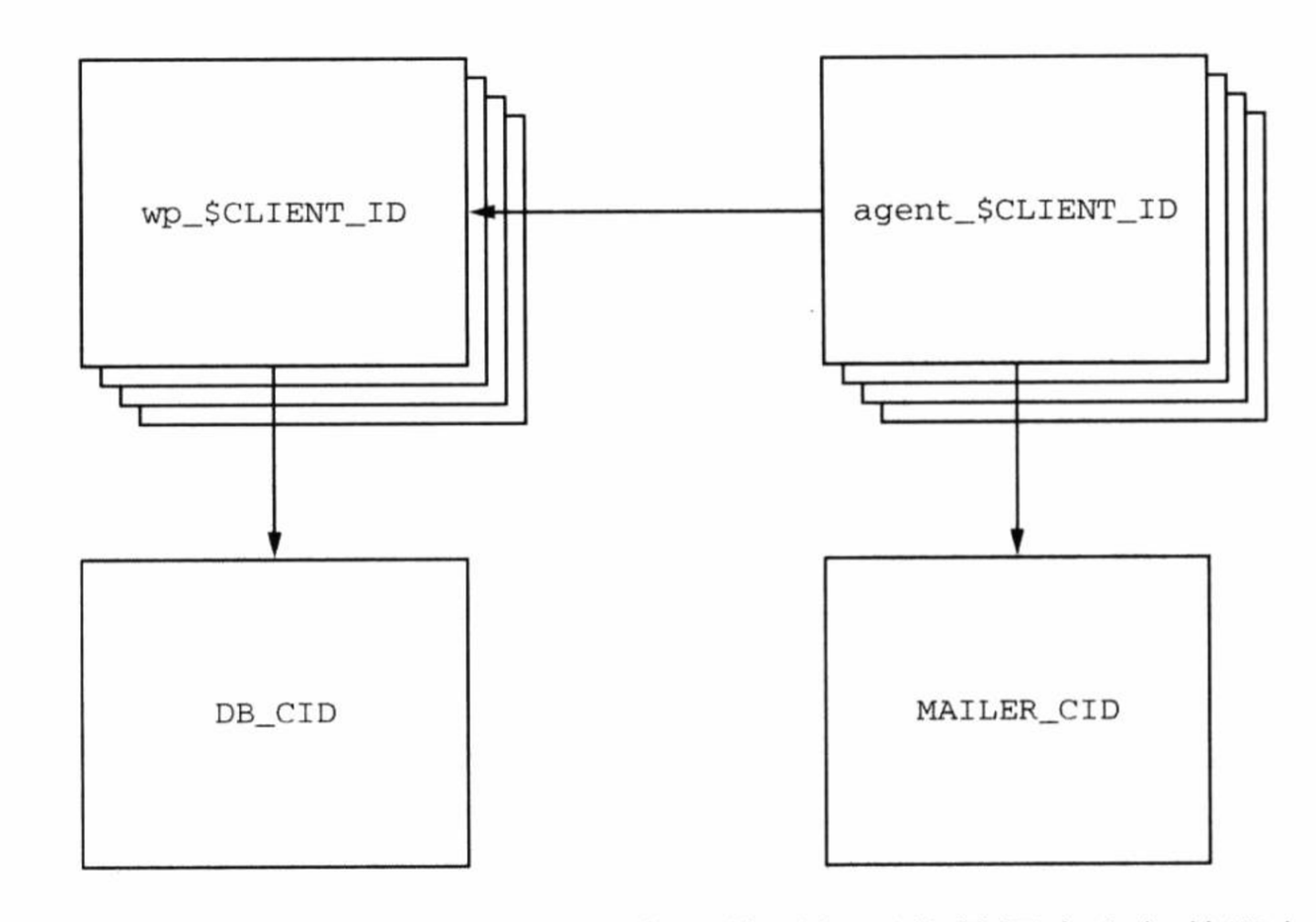


图 2-4 每个 WordPress 和监控器容器使用相同的数据库和邮件服务

客户对当前交付的所有工作成果很满意。但有一件事可能会比较烦人。在早期的测试中，你发现监视器在该网站不可用时，的确可以准确通知邮件服务器，但重新启动该网站和监控器需要手动实现。要是系统在检测到故障时，能尝试自动恢复就更好了。Docker 对此提供了重启的策略，但仍不够稳定。

2.6 建立持久化的容器

有些软件故障事实上仅在极少见情况下出现。当这些条件发生时，告警很重要，但至少也要做到尽可能快地恢复服务。你在本章中建立的监控系统是一个很好的开端，使得系统所有者保持对系统故障的警觉，但它丝毫无助于恢复服务。

当容器中的所有进程都退出后，该容器将进入退出状态。请记住，一个 Docker 容器可以处于以下四种状态：

- 运行
- 已暂停
- 重新启动
- 已退出（容器尚未启动时也适用）

从暂时故障中恢复的基本策略是，在其退出或失败时能有一个自动重新启动的过程。

Docker 提供了用于监控和重新启动容器的几个选项。

2.6.1 自动重启容器

Docker 基于重新启动策略提供了这一功能。创建容器时使用`--restart`标志，就可以通知 Docker 完成以下操作：

- 从不重新启动（默认）
- 检测到故障时尝试重新启动
- 当检测到故障时，在一段预定的时间后重新开始尝试重启
- 不管任何条件，始终重新启动容器

Docker 并不总是试图立即重新启动容器。那样只会适得其反，将导致更多的问题。想象这样一个容器，除了打印时间和退出，什么也不做。如果容器配置为始终重新启动，Docker 总是立即重新启动它，那么系统除了重新启动该容器，就什么都做不了了。相反，Docker 针对定时尝试重新启动，采取了指数回退策略。

回退策略决定了连续尝试重新启动所需要的时间间隔。指数回退策略会将花在前一次等待连续尝试的时间加倍。例如，如果第一次容器重新启动 Docker 需要等待 1 秒钟，然后第二次尝试将等待 2 秒，第三次等待 4 秒，第四次等待 8 秒，以此类推。具有较少初始等待时间的指数回退策略是一种常见的服务修复技巧。下面，你可以看到 Docker 使用这一策略，构建了一个总是重新启动并简单打印时间的容器：

```
docker run -d --name backoff-detector --restart always busybox date
```

接着在几秒钟后，使用尾随日志功能来观察回退和重新启动：

```
docker logs -f backoff-detector
```

日志会显示已重新启动的所有次数，并会等到下次重新启动，打印出当前时间，然后退出。为监控器和 WordPress 容器添加这一标志，就能解决修复问题。

不希望直接采用这个方法的唯一理由是，在补偿期间，容器并没有运行。容器等待重新启动的时候，仍是在重新启动的状态。在回退探测容器时，尝试运行另一个进程：

```
docker exec backoff-detector echo Just a Test
```

运行该命令会导致的错误信息：

```
docker exec backoff-detector echo Just a Test
```

这意味着那些处于运行状态时才能做到的事情此时都无法做到，比如在容器中执行其

他命令。如果你需要在一个故障的容器中运行诊断程序，这可能就是一个问题。更好的策略是容器中运行 init 或 supervisor 进程。

2.6.2 使用 init 和 supervisor 进程维持容器的运行状态

init 或 supervisor 进程，用于启动和维护其他程序状态。在 Linux 系统中，PID #1 是 init 进程。它启动所有其他系统进程，并在出现意外故障时重新启动它们。容器中使用类似的模式来启动和管理进程，是一个常见的做法。

容器中的 supervisor 进程用来保持容器始终运行，即使目标进程（如一个 web 服务器），出现故障并重新启动。一个容器中可能有多个这样程序，最流行的包括 `init`、`systemd`、`runit`、`upstart` 和 `supervisord`。使用这些程序发布的软件会在第 8 章讲解。现在，来看看使用 `supervisord` 的容器。

一个名为 Tutum 的公司在单个容器中提供了完整的 LAMP（Linux、Apache、MySQL 和 PHP）栈。创建的容器使用 `supervisord` 来确保所有相关的进程持续运行。启动一个示例容器：

```
docker run -d -p 80:80 --name lamp-test tutum/lamp
```

使用 `docker top` 命令，该容器中能看到哪些进程正运行着：

```
docker top lamp-test
```

top 子命令显示的是主机为每一个容器中的进程所分配的 PID。你会看到 `supervisord`、`MySQL` 和 `Apache` 都在运行程序清单中。现在，容器运行起来了，你可以通过手动停止容器内的某一进程，来测试 `supervisord` 的重启功能。

问题是，在容器内杀死进程，你需要知道它在容器中的 PID 值。要获得该列表，运行以下 exec 子命令：

```
docker exec lamp-test ps
```

生成的进程列表在 CMD 列中显示了 `apache2` 的进程：

```
PID TTY  TIME CMD
1   ?    00:00:00 supervisord
433 ?    00:00:00 mysqld_safe
835 ?    00:00:00 apache2
842 ?    00:00:00 ps
```

当你运行该命令时，PID 列中的值会有所不同。找到 PID 上 `apache2` 那列，然后在下面的命令中插入`<PID>`：

```
docker exec lamp-test kill <PID>
```

运行此命令，会在 lamp-test 容器内运行 Linux 的 kill 程序，通知 apache2 进程关闭。当 apache2 停止时，supervisord 进程会记录该事件，并重启该进程。容器日志将清晰地显示这些事件：

```
...
... exited: apache2 (exit status 0; expected)
... spawned: 'apache2' with pid 820
... success: apache2 entered RUNNING state, process has stayed up for >
     than 1 seconds (startsecs)
```

使用 init 或 supervisor 程序的一个常见替代方法是使用一个启动脚本，该脚本至少会检查软件成功启动的先决条件。这些脚本有时会用作容器的默认命令。例如，可以通过在启动 WordPress 进程前，运行脚本验证和设置默认环境变量，来启动你所创建的 WordPress 容器。你可以通过覆盖容器启动脚本，例如，使用命令来查看启动脚本内容：

```
docker run wordpress:4 cat /entrypoint.sh
```

运行命令将导致如下错误信息：

```
error: missing WORDPRESS_DB_HOST and MYSQL_PORT_3306_TCP environment
variables
...
```

之所以失败，是因为即使你将命令设置为 cat /entrypoint.sh，Docker 容器仍会在运行该命令之前会先执行入口点命令。在入口点命令处，可放置用来验证容器启动条件的代码。虽然这会在本书第 2 部分深入讨论，你并不需要知道如何在命令行中覆盖或专门设置一个容器的入口点。再次尝试运行最后一个命令，但这次使用--entrypoint 标志来运行指定程序，并传递参数：

```
docker run --entrypoint="cat" \
    wordpress:4 /entrypoint.sh
```

使用 cat 命令作为容器执行的入口

将/entrypoint.sh 作为 cat 命令的参数

如果你运行了这个脚本，就会看到它如何摆脱对软件依赖的环境变量，并设置相应的默认值。一旦这个脚本验证了 WordPress 能合法执行，它将在启动之前接受请求或执行默认的命令。

启动脚本是建立持久化容器的一个重要组成部分，始终可以与 Docker 重启的策略相结合来互补优势。因为无论是 MySQL 还是 WordPress 容器，都已经使用启动脚本，你只需简

单地为每个示例脚本，更新版本并设置重启策略。

最终修改完，你就建立了一个完整的 WordPress 站点系统，学习了使用 Docker 管理容器的基础知识，做了相当多的实验。你的机器上可能有几个不再需要的容器，为了收回这些容器所占用的资源，你必须暂停并从系统中删除它们。

2.7 清理

易于清理也是使用容器和 Docker 的主因之一。容器提供的隔离，能简化你在不得不停止进程和删除文件时的所有步骤。用了 Docker，整个清理过程被压缩成几个简单的命令。在任何清除任务中，你必须先确定要停止或删除的容器。记住，要列出计算器上所有的容器，使用 `docker ps` 命令：

```
docker ps -a
```

因为本章示例所创建的容器将不再使用，你应该能够安全地停止并删除所有列出的容器。如果有任何为自己创建的容器，删除容器时请务必小心。

所有的容器使用硬盘空间，来存储已写入到容器文件系统的日志、容器的元数据和文件。所有的容器也会消耗机器上的资源：容器名称、主机端口映射等全局命名空间。在大多数情况下，需要删除这些不再用的容器。

若要从机器中删除一个容器，使用 `docker rm` 命令。例如，要删除名为 wp 的已停止的容器：

```
docker rm wp
```

你应该对 `docker ps -a` 产生的列表中的所有容器扫过一遍，删除处于已退出状态的所有容器。如果你尝试删除正在运行、暂停、重新启动的容器，Docker 会显示如下错误：

```
Error response from daemon: Conflict, You cannot remove a running
container.Stop the container before attempting removal or use -f
FATA[0000] Error: failed to remove one or more containers
```

在容器文件被删除之前，容器中运行的进程应当停止。你可以使用 `docker stop` 命令或在 `docker rm` 中使用`-f` 标志做到这一点。关键的区别是，当你使用`-f` 标志停止一个进程，Docker 发送 `SIG_KILL` 信号，立即终止接收过程。相比之下，使用 `docker stop` 将发送 `SIG_HUP` 信号。`SIG_HUP` 的收件人有时间进行最后的退出和清理任务。`SIG_KILL` 信号没有这样的允许时间，并可能导致文件损坏或较差的网络体验。你可以通过 `docker kill` 直接给容器发出 `SIG_KILL` 命令。但是，只有在必须小于标准 30 秒（最大停止时间）

内停止容器时，才需要使用 docker kill 或 docker rm -f。

未来，如果你在做短暂的容器试验，可以通过在命令中指定--rm 来避免清理工作的负担。这样做，当容器进入退出状态时，就会被自动删除。例如，下面的命令会将一个新的 BusyBox 容器显示的消息写到屏幕上，并在容器退出时立即将其删除：

```
docker run --rm --name auto-exit-test busybox:latest echo Hello World
docker ps -a
```

在这种情况下，你既可以使用 docker stop 也可以使用 docker rm 来妥善清理，此外使用单步式 docker rm -f 命令同样合适。鉴于在第 4 章中提到的原因，你也应该使用-v 标志。Docker CLI 很容易构建这样一个快速清除命令：

```
docker rm -vf $(docker ps -a -q)
```

本章介绍了容器中运行软件的基础知识。第 1 部分的其余各章将重点放在用容器工作的某个具体方面。下一章将侧重于安装和卸载镜像、镜像是如何与容器关联起来，以及如何与容器文件系统协同工作的内容。

2.8 小结

Docker 项目的核心要点是使用户能够在容器上运行软件。本章介绍如何使用 Docker 实现这一目的。本章所涵盖的理念和功能包括以下内容：

- 容器可以基于虚拟终端，连接到用户 shell 或以守护模式的形式运行。
- 在默认情况下，每一个 Docker 容器都有自己的 PID 命名空间，隔离了每个容器的进程信息。
- Docker 用产生的容器 ID、简写的容器 ID，或者其人性化的名称来标识每个容器。
- 所有容器都在四个不同状态中：正在运行、暂停、重新启动或退出。
- docker exec 命令可以在正运行的容器内运行一个额外的进程。
- 用户可以通过输入或在容器创建时给进程指定环境变量的方式，提供额外的配置。
- 使用--read-only 标志创建容器时，会将挂载的容器文件系统设置为只读，防止容器被修改。
- 容器重启策略，即在容器创建时设置--restart 标志，将有助于系统出现故障时进行自动恢复。
- Docker 使用 docker rm 命令清理容器，和创建时一样简单。

第 3 章　软件安装的简化

本章介绍

- 选择所需的软件
- 使用 Docker Hub 查找和安装软件
- 从其他来源安装软件
- 了解文件系统的隔离
- 镜像和文件系统分层是如何工作的
- 使用分层镜像的优点

前两章介绍了 Docker 全新的概念和封装技术。本章将深入到容器文件系统和软件的安装。软件安装分成三个步骤，如图 3-1 所示。

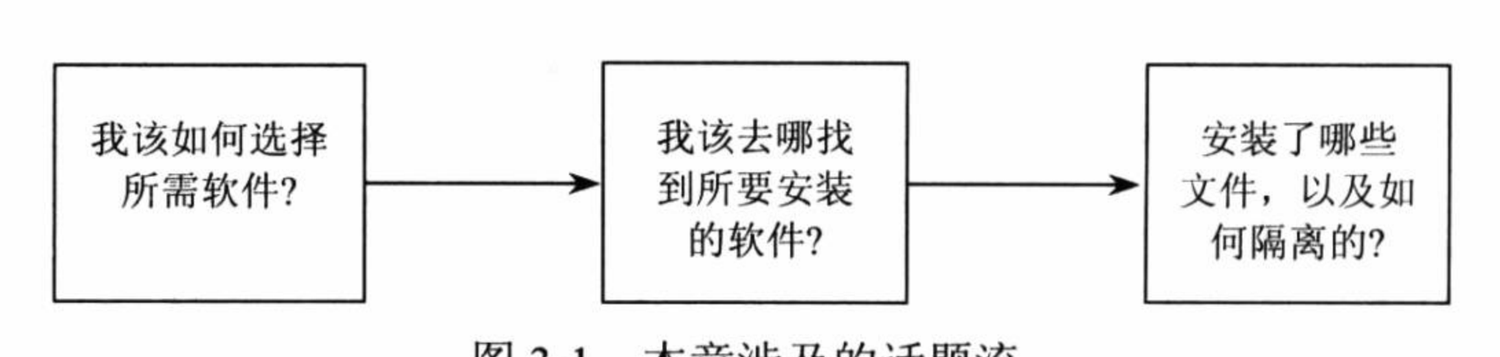


图 3-1　本章涉及的话题流

安装任何软件的第一步是确定你要安装的软件。你知道软件是使用镜像发布的，但你需要知道如何确切地告诉 Docker 所要安装的镜像。我已经提到过，仓库存储着镜像，但在本章中我将展示仓库和标记以选择安装所需的软件镜像。

本章将涉及安装 Docker 镜像的细节，主要有三种方式：

- Docker Hub 和其他注册服务器
- 使用 `docker save` 和 `docker load` 命令加载、导出镜像文件
- 用 Dockerfiles 构建镜像

阅读这些材料，你将学习到如何用 Docker 隔离已安装的软件，你会接触到一个新的名词——分层。用镜像的时候，分层是一个重要的概念，它会对软件用户产生重要影响。本章将讨论有关镜像是如何工作的部分。这些知识将帮助你评估镜像质量，为本书的第 2 部分建立一个基准技能。

3.1 选择所需的软件

假设你要安装一个叫作 TotallyAwesomeBlog 2.0 的程序，你会怎样告诉 Docker 要安装什么呢？需要一种方法来命名该程序，并指定要使用的版本，以及指定你想安装的来源。学习如何选择特定的软件是软件安装的第一步，如图 3-2 所示。

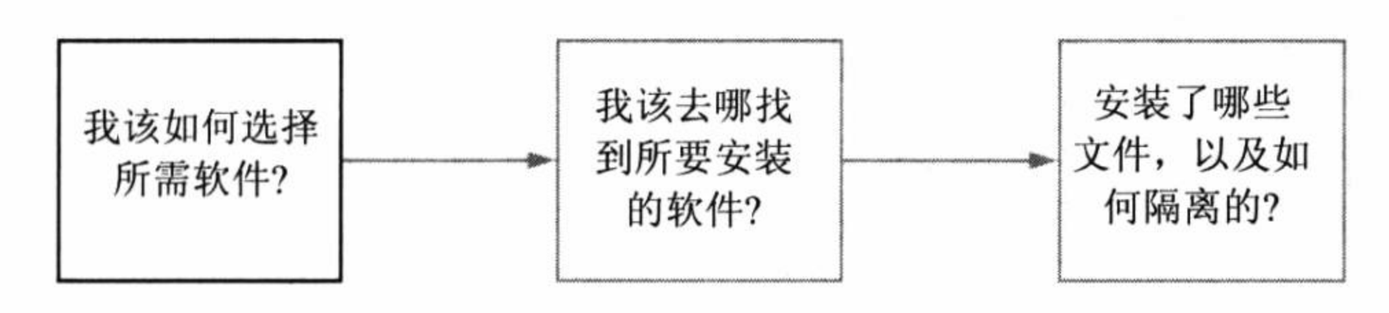


图 3-2 步骤一：选择所需的软件

你已经知道，Docker 通过镜像来创建容器。一个镜像是一个文件，包含了创建容器所需的文件和镜像元数据。该元数据包含关于镜像之间的关联信息、命令历史、暴露的端口、卷的定义等。

镜像有标识符，因此可以被用作软件的名称和版本，但在实际中很少被用作原生镜像的识别符。识别符是唯一的长序列，包括字母和数字。每次更改镜像时，识别符也会被修改。镜像识别符很难奏效，因为它是不可预测的。相反，用户应该用仓库工作。

3.1.1 什么是仓库

仓库是一个有名字的镜像桶，名字类似于 URL。仓库名由该镜像所在的主机、拥有该镜像的用户账户和一个简短的名称组成。例如在下文你将安装来自 quay.io/dockerinaction/ch3_hello_registry 仓库的镜像。

正如软件可能有多个版本，仓库可以容纳多个镜像。每一个库中的镜像可由标签来唯一标识。如果发布 quay.io/dockerinaction/ch3_hello_registry 的新版本，可以将其标记为“v2”，将老版本标记为“v1”。如果想下载旧版本，可用其 v1 标签来明确指定镜像。

你在第 2 章安装的镜像来自 Docker Hub 上的 Nginx 仓库，指定的是“最新”的标签。仓库名称和标签形成复合键，或形成由非唯一组件组成的唯一参考。在该示例中，镜像被指定为 nginx:latest。虽然这种方式的标识符可能偶尔会比原始镜像的标识符显得更长，但它们可预测并可获得镜像的意图。

3.1.2 使用标签

标签是唯一指定镜像的重要途径，也是一种创建有用别名的便利方法。尽管一个标签只能在一个仓库中被应用到单个镜像，但一个镜像可以有多个标签。这使得仓库管理者可以构建有用的版本或功能标签。

例如，Docker Hub 在 Java 库维护着以下标签：7、7-jdk、7u71、7u71-jdk、openjdk-7 和 openjdk-7u71。所有这些标签被施加到相同的镜像。但随着 Java 7 当前小版本的增加，7u72 标签发布了，7u71 标签可能会消失，并被 7u72 标签所取代。如果你关心正在运行 Java 7 中的小版本，你必须跟上这些变化。如果你只是想确保运行的是最新的 Java 7 版本，只需使用是 7 的标签。这样总是能被分配到 Java 7 的最新小版本镜像。这些标签为用户提供了极大的灵活性。

不同的镜像标签结合不同的软件配置也是很常见的，举例来说，我为一个名为 geoip 的开源程序发布了两个镜像。这是一个 web 应用，能通过关联网络地址来获得粗略的地理位置信息。第一个镜像使用默认配置，即软件本身直连网络。第二个镜像则放在 web 负载均衡器之后运行。每个镜像具有不同的标记，使用户很容易地指定所需要的那个镜像。

提示 当你正在寻找软件进行安装，要始终密切注意仓库中提供的标签。如果你不知道需要哪一个，你可以在从仓库 pull 时简单地忽略标签选项，从仓库中下载所有已标记的镜像。我有时候不小心会这么做，还是蛮恼人的，不过这很容易清理。

用于 Docker 选择软件的知识都在这了。有了这些，你就可以开始通过 Docker 寻找和安装软件了。

3.2 查找和安装软件

你可以通过仓库名来指定软件，但怎么发现你想要安装的仓库呢？发现可信软件是复杂的，接下来是学习如何使用 Docker 安装软件，如图 3-3 所示。

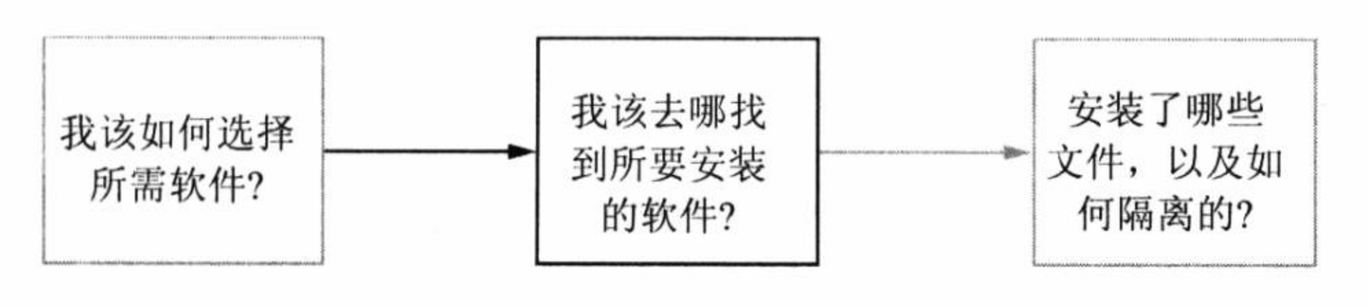


图 3-3 步骤二：定位仓库

为了找到仓库，你既可以靠运气来猜，或者使用索引。索引就是仓库的搜索引擎。有好几个公共 Docker 索引库，但默认情况下 Docker 集成的是一个名为 Docker Hub 的索引库。

Docker Hub 是由 Docker 公司运营的注册服务器和索引库，也是 Docker 默认的注册服务器和索引库。当你发出 `docker pull` 或 `docker run` 命令，而没有指定一个替代的注册服务器，Docker 将默认为在 Docker Hub 里面寻找。Docker Hub 使得 Docker 更加好用。

Docker 公司已做出努力，确保 Docker 是一个开放的生态系统，并发布了公开的镜像，这样你就可以运行属于自己的注册服务器，`Docker` 命令行工具也很容易配置，来使用替代的注册服务器。下文中，我将讨论包括 Docker 替代镜像的安装和发布工具。但下面，首先会讲解如何使用 Docker Hub，让你可以从默认的工具中获得最多。

3.2.1 命令行使用 Docker Hub

Docker 几乎所有的东西都可以通过命令行来完成，包括在 Docker Hub 中搜索仓库。

`Docker` 命令行会为你搜索 Docker Hub 索引库，显示结果包括像每个仓库被打星的次数，标记一个官方支持的特定仓库（`OFFICIAL` 列），标记一个特定仓库是值得信赖的（`TRUSTED` 列）。Docker Hub 网站允许注册用户为仓库打星，类似于其他社交网站，如 GitHub。仓库的星星数可以作为镜像质量和社区知名度或可信赖程度的重要指标。Docker Hub 也提供了一套由 Docker 公司或当前软件维护者维持的官方仓库。这些通常被称为库。

有迹象表明，镜像作者可以在 Docker Hub 以两种方式发布镜像：

- 使用命令行来发布独立系统构建的镜像。这种方式被认为是不能信任的，因为并不清楚它们究竟是如何构建的。
- 公开 Dockerfile，并使用 Docker Hub 的持续构建系统。Dockerfiles 是构建镜像的脚本。首选自动创建的镜像版本，因为在安装之前，Dockerfile 可用于检查。第二种方式构建的镜像将被标记为可信赖的。

使用私有的 Docker Hub 注册服务器，或将镜像推送到 Docker Hub 上你的账号中，需要你的认证授权。在这种情况下，你可以使用 `docker login` 命令登录到 Docker Hub。一旦登录后，你就可以从私有仓库下载，推送到你控制的任何镜像库，并在库中标记你的镜像。第 7 章会介绍推送和标记镜像。

运行 `docker login` 将提醒你输入 Docker Hub 的账户密码。一旦你提供了，你的命令行客户端将被验证，就可以访问你的私人仓库。完成时，你可以使用 `docker logout` 命令注销。

如果你想找到要安装的软件，则需要知道从哪里开始搜索。下面的示例演示了如何使用 `docker search` 命令搜索。该命令可能需要几秒钟，但它内置了一个超时，所以最终都会返回。当运行该命令时，它只会搜索索引，并且什么都不会安装。

假设 Bob 是一名软件开发者，决定他从事的项目需要一个数据库。他曾经听说过一个蛮受欢迎的数据库，名为 postgres。他想在 Docker Hub 上是否会有呢？所以他执行了下面的命令：

```
docker search postgres
```

几秒钟后返回了几个结果。在列表的顶部，有一个非常受欢迎的仓库，带了数百颗星星。他也很喜欢，因为这是一个官方的仓库，意味着 Docker Hub 维护者已经精心挑选了仓库的管理者。他用 `docker pull` 来安装镜像，他的项目也能继续前进。

这是如何使用 `docker` 命令行搜索的简单示例。该命令将在 Docker Hub 上搜索所有含 `postgres` 字样的仓库。由于 Docker Hub 是一个免费的公共服务，用户往往会建立大量公开的个人副本。Docker Hub 可以让用户为仓库打星，类似于 Facebook 的“Like”一样。这是镜像质量的一个合理指标，但应注意不要用来作为代码值得信赖的标志。

试想一下，如果有人构建了一个拥有数百颗星星的仓库，提供了一些高质量的开源软件。有一天，他们的仓库被恶意黑客获得控制权，且发布的镜像包含病毒。虽然容器可以有效隔离恶意代码，但在恶意镜像中就不成立了。如果攻击者控制了镜像的构建或有针对性地攻击且专挑薄弱的镜像本身，这可能会导致严重的危害。出于这个原因，正在使用公开脚本构建的镜像被认为是更值得信赖的。运行 `docker search` 的搜索结果，你可以看

到镜像是从公开的脚本构建的，在 AUTOMATED 列寻找一个 [OK] 标记。

现在你已经知道如何通过终端找到 Docker Hub 上的软件。虽然你可以从终端完成大多数，不过有一些事，你只能通过网站来做到。

3.2.2 通过网站访问 Docker Hub

如果你在访问 docker.com 的时候偶然碰到 Docker Hub，你应该花点时间看看 https://hub.docker.com。Docker Hub 可以搜索仓库、组织或特定用户。用户和组织的资料页会列出账户所持有的仓库、最近活动的仓库，以及该账户被打星的仓库。在仓库页面，你可以看到以下内容：

- 镜像提供商提供的镜像常规信息
- 仓库中的标签列表
- 仓库的创建日期
- 下载次数
- 用户评论

Docker Hub 可以免费加入，本书中你将需要一个账户。当你登录后，就可以打星，对仓库发表评论。你可以创建和管理自己的仓库。我们将在第 2 部分讲述。现在，仅仅是感受一下这个网站和它所提供的内容。

活动：Docker Hub 寻宝游戏

这是个很好的练习，使用第 2 章中所学到的技能，在 Docker Hub 上找软件。这个活动的目的是鼓励你使用 Docker Hub 和练习创建容器。下面将会介绍在 docker run 命令中的三个新的选项。

在这个活动中，你将用 Docker Hub 上的两个可用的镜像来创建容器。第一个来自 dockerinaction/ch3_ex2_hunt 仓库。在这种镜像中，你会发现一个小程序，提示输入密码。只能通过查找 Docker Hub 上第二个神秘仓库运行的一个容器来找到密码。这些镜像使用的程序需要将终端连接到容器。下面的命令演示了如何做到这一点，停止运行时将自动删除该容器：

```
docker run -it --rm dockerinaction/ch3_ex2_hunt
```

当你运行该命令时，寻宝游戏程序会提示你输入密码。如果你已经知道答案，继续前进，输入密码。如果没有的话，只需输入任何东西，它会给你一个提示。此时，你应

该有需要完成活动的工具。如图 3-4 所示为你需要做什么。

还卡着？我可以给你一个提示。神秘仓库是为这本书创建的。也许你应该尝试寻找这本书的 Docker Hub。请记住，仓库以用户名/仓库的模式命名。

当你得到答案，给自己鼓鼓劲，再使用 docker rmi 命令删除该镜像。具体而言，运行的命令看起来应该像这样的：

```
docker rmi dockerinaction/ch3_ex2_hunt
docker rmi <mystery repository>
```

如果你一直紧跟示例，可以使用 docker run 命令的--rm 选项，就不需要对容器进行清理。在本例中，你已经学到了很多。也已经找到了 Docker Hub 的新镜像，并以新的方式使用 docker run 命令。还有很多交互式容器的东西需要了解。下一节会介绍更多的细节。

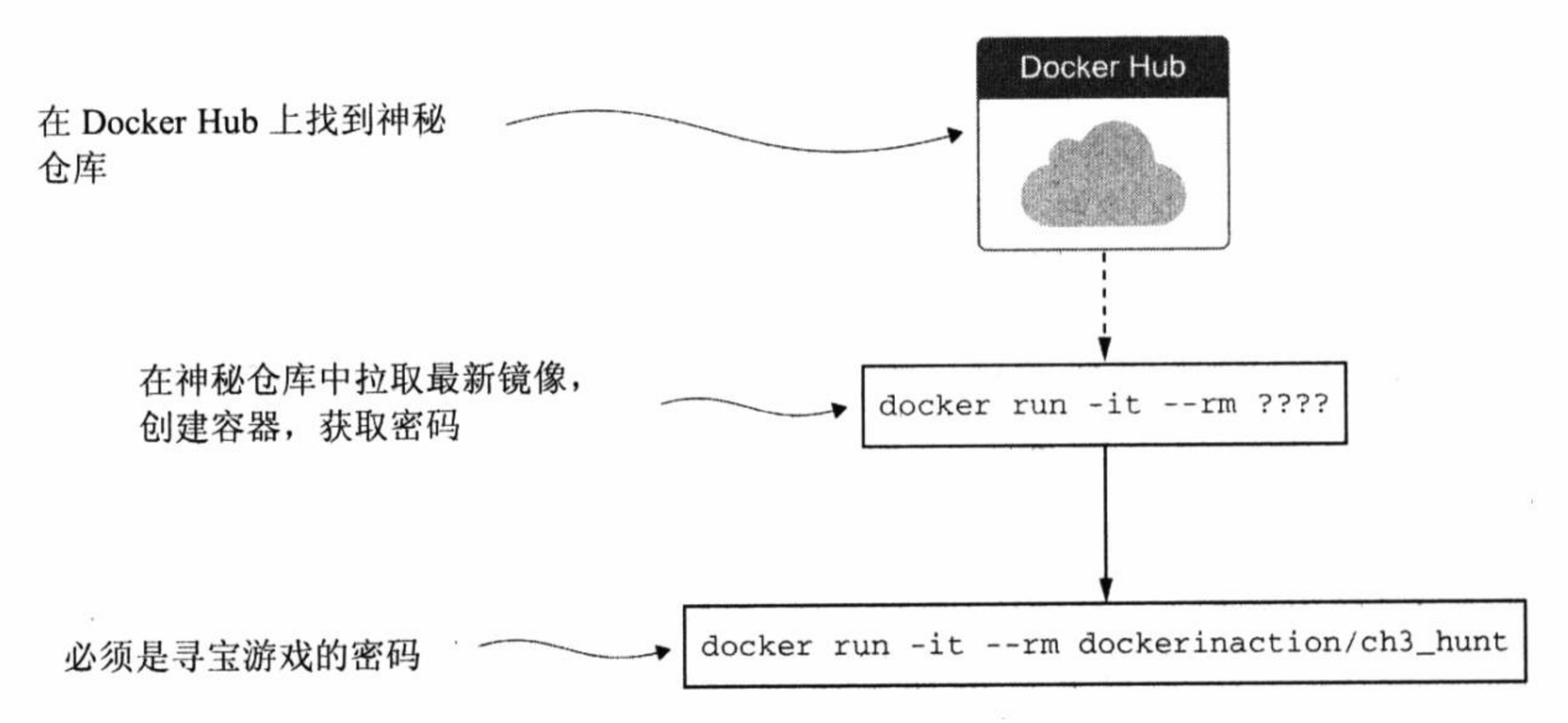


图 3-4 完成 Docker Hub 寻宝游戏的必要步骤。在 Docker Hub 上找到神秘仓库。并从该仓库安装最新的镜像，交互运行得到密码

Docker Hub 不是软件的唯一来源，这取决于软件发行商的目标和角度，Docker Hub 可能不是适当的发布点。封闭源代码或专有项目可能不想冒险通过第三方发布他们的软件。还有其他三种方式来安装软件：

- 可以使用替代仓库的注册服务器或运行自己的注册服务器。
- 可以手动从文件加载镜像。
- 可以从其他来源下载项目，并利用提供的 Dockerfile 自建镜像。

所有这三个选项可用于私有项目或企业基础设施。接下来的几个小节会介绍如何从替

代来源安装软件。

3.2.3 使用替代注册服务器

正如前文提到的，Docker 注册服务器可供任何人运行。托管公司已将其集成到自己的产品，而且已经开始运行他们自己的内部注册服务器。我不打算谈注册服务器，第 8 章会涉及，但是你早点学会如何使用它们是非常重要的。

使用替代注册服务器很简单，不需要额外的配置。所有你需要的是注册服务器的地址。以下命令将会从替代注册服务器下载另一个“Hello World”的示例，如下所示：

```
docker pull quay.io/dockerinaction/ch3_hello_registry:latest
```

注册服务器地址，是仓库规范的一部分。完整格式如下：

```
[REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

Docker 知道如何和注册服务器沟通，所以唯一的区别是，你指定了注册服务器。在某些情况下，注册服务器将需要认证的步骤。如果你遇到这个情况，请查阅文档或从注册服务器的配置组中找到更多信息。当你结束使用已安装的 hello-registry 镜像，用以下的命令将其删除：

```
docker rmi quay.io/dockerinaction/ch3_hello_registry
```

注册服务器功能强大。它们使得用户可不用考虑镜像存储和运输的问题。但是运行你自己的注册服务器，就会比较复杂，也可能成为基础设施部署潜在的单点故障。如果为你的应用场景运行定制的注册服务器，听起来有点复杂，第三方分发工具也可能出问题，你也许可以考虑直接从文件加载镜像。

3.2.4 镜像文件

Docker 提供了一个命令，可将镜像由文件加载到 Docker。有了这个工具，你也可以加载通过其他渠道获得的镜像文件。也许你的公司选择通过中心文件服务器或者版本控制系统来分发镜像。如果镜像足够小，也可以请你的朋友通过邮件发给你或者通过 U 盘分享。获得了这个文件，你可以通过 `docker load` 命令加载到 Docker 中去。

我告诉 `docker load` 命令之前，你需要一个镜像文件。不可能在你旁边就会躺着一个镜像文件。我会告诉你如何从一个已加载的镜像保存下一个文件。为了这个示例的目的，

你会下载 busybox:latest.，该镜像小，易于使用。可用 docker save 命令把该镜像保存到文件。如图 3-5 所示，docker save 从 BusyBox 容器导出一个文件。

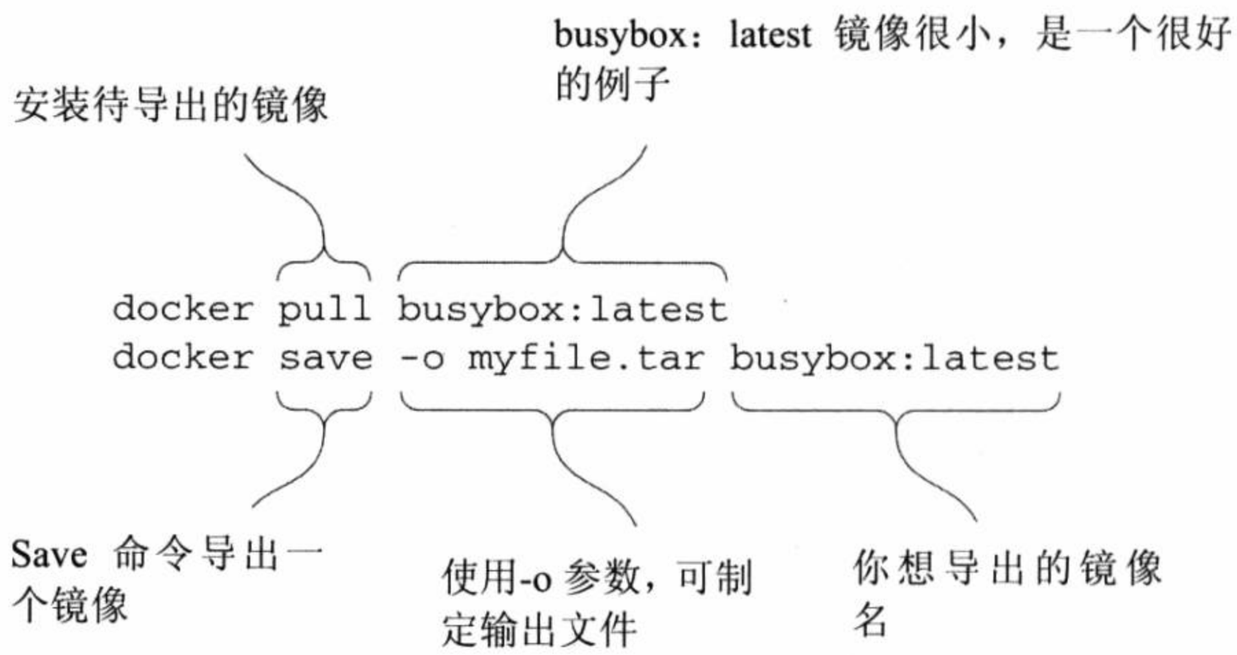


图 3-5 pull 和 save 命令部分

在这个示例中，我用.tar 的文件名后缀，因为 docker save 命令创建的是 TAR 归档文件。你可以使用任何你想要的文件名。如果省略-o 标志，生成的文件将被传输到终端。

提示 因打包使用 TAR 存档文件的其他生态系统定义了自己的文件扩展名。例如 Java 使用.jar、.war 和.ear。在这样的情况下，使用自定义的文件扩展名，可对目的和归档的内容提供帮助提示。虽然 Docker 没法设置默认值，这个事上面也没有官方指导，如果你经常使用这些文件工作的话，你会发现使用自定义扩展挺有用。

运行 save 命令后，docker 程序将被强制终止。检查当前工作目录。如果指定的文件存在，使用此命令从 Docker 删除该镜像：

```
docker rmi busybox
```

去除镜像后，使用 docker load 将创建的文件再次加载。像 docker save，如果运行 docker load 命令而不使用-i 参数，Docker 会使用标准输入流，而不是从文件读取归档：

```
docker load -i myfile.tar
```

一旦执行了 docker load 命令，镜像就会被加载。你可以通过运行 docker images 命令来再次验证这一点。如果一切顺利，BusyBox 也应包括在列表中。

把镜像作为文件来用，和注册服务器一样简单易用，但是你错过了所有注册服务器提供的分发设施。如果你想建立自己的分发工具，或者你已经有工具了，也应该尝试着通过

这些命令和 Docker 集成起来。

另一种流行的项目分发模式，是使用一组文件和安装脚本。这种方法适用于需要使用公开的版本控制库分发的开源项目。在这些情况下，会使用文件，但该文件不是镜像，它只是一个 Dockerfile。

3.2.5 从 Dockerfile 安装

Dockerfile 是 Docker 用来描述新镜像构建步骤的脚本。这个文件会和作者想要放入镜像的软件一起发布。这样，你不用从技术上安装镜像。相反，你只要跟着说明构建镜像。Dockerfiles 会在第 7 章深入讨论。

分发 Dockerfile 类似于分发镜像文件。可选择适合自己的分发机制。一个常见的模式是通过如 Git 或 Mercurial 的版本控制系统来分发 Dockerfile。如果你已经安装了 Git，那么可以试试用这个来运行公开库：

```
git clone https://github.com/dockerinaction/ch3_dockerfile.git
docker build -t dia_ch3/dockerfile:latest ch3_dockerfile
```

在这个示例中，你将公共源代码库的项目复制到机器上，然后使用项目的 Dockerfile 构建 Docker 镜像。`docker build` 命令的`-t` 选项的值设置成要安装镜像的仓库。从 Dockerfiles 构建镜像是一种将项目融入现有工作流的轻量级方法。采取这种方法有两个不足：首先，根据项目的具体情况，构建过程可能需要一些时间；第二，依赖关系可能会从撰写 Dockerfile 到镜像构建这段时间发生变化。这些问题会影响发布，但不太会影响用户体验。尽管有这些缺点，仍然很流行。

当你完成这个示例，确保清理自己的工作区：

```
docker rmi dia_ch3/dockerfile
rm -rf ch3_dockerfile
```

阅读本节之后，你应该对所有用 Docker 安装软件的可选项有一个完整的画面了。但是，当安装了软件，你应该会想对它做一些改动吧。

3.3 安装文件和隔离

了解了镜像如何识别、发现和安装是一个 Docker 用户最基本的水平。如果你了解哪些文件实际要安装，这些文件是如何构建并在运行时隔离，利用这些经验，你就能回答更复

杂一些的问题，比如这些：

- 什么样的镜像因素会影响下载和安装速度？
- 当我用 docker images 命令时，列举了哪些未命名的镜像？
- 为何 docker pull 命令的输出包括有关相关依赖层的下载消息？
- 我写入容器文件系统中的那些文件放在哪？

这是第三步，也是最后一步来了解如何使用 Docker 安装软件，如图 3-6 所示。

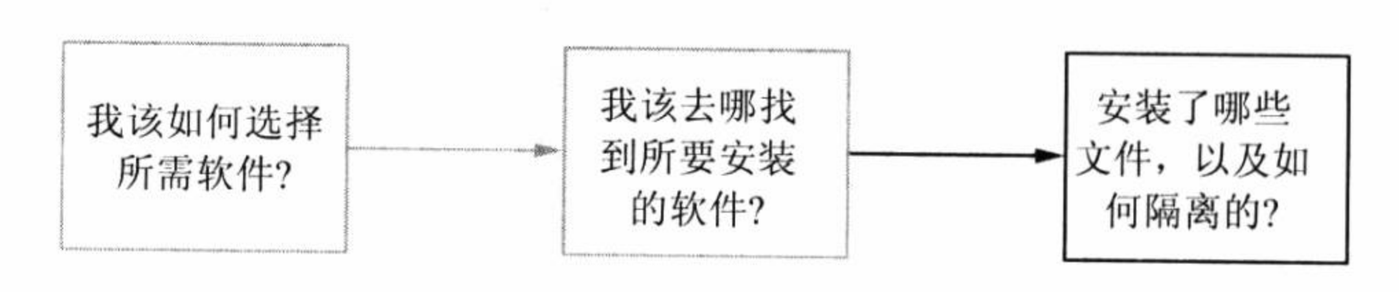


图 3-6 步骤三：了解软件是如何安装的

到目前为止，当我写有关安装软件部分，均用了镜像这个词。这是为了推断你要使用的软件是在单独一个镜像上，并且该镜像包含在单个文件中。虽然这偶尔可能是准确的，但大部分时候，我一直称之为镜像的概念，实际上是镜像层的集合。一个镜像层至少和一个其他的镜像关联。这是比较容易理解的，特别是当你在实战中看到过它们。

3.3.1 镜像层实战

在这个示例中你要安装两个镜像，都依赖于 Java 6。应用程序本身是 Hello World 风格的。我希望你留意的是，当你安装每一个镜像时 Docker 做了什么。你应该注意到它需要多久才能安装好，第一个和第二个之间的比较，还需阅读打印到终端的输出。当镜像正在安装，你可以观察 Docker 是如何确定它需要下载哪些依赖，然后查看各个镜像层的下载进度。Java 是个很好的示例，因为该层是相当大的，这会给你一段时间真正看到 Docker 的行为。

你要安装两个镜像 dockerinaction/ch3_myapp 和 dockerinaction/ch3_myotherapp。你只要使用 docker pull 命令，因为你只需要看到镜像的安装，而不是从镜像启动容器。下面是应该要运行的命令：

```
docker pull dockerinaction/ch3_myapp
docker pull dockerinaction/ch3_myotherapp
```

你看见了吗？除非你的网络连接远远比我好，或者你已经安装了 Java 6 以及其他一些镜像的依赖，下载 dockerinaction/ch3_myapp 应该已经比 dockerinaction/ch3_myotherapp 慢得多。

当你安装 ch3_myapp 时，Docker 决定了它需要安装 openjdk 6 镜像，因为它是请求镜

像的直接依赖（父层）。当 Docker 去安装依赖时，才发现原来该层的依赖关系，而且还是第一次下载。一旦安装了该层的所有依赖，该层安装才结束。最后，openjdk-6 层会被安装，然后再安装 ch3_myapp 层。

当你发出安装 ch3_myotherapp 命令，Docker 确定了 openjdk 6 已经安装并立即安装 ch3_myotherapp 镜像。这算简单的，由于要传输的数据小于一兆字节，这就更快了。但同样的，对用户而言这都是理想的过程。

从用户角度来看，这种能力还是不错的，也许不希望优化。只要看看他们碰巧还可以。从软件或镜像作者的角度来看，这种能力应该在镜像设计中发挥主要的因素。我在第 7 章将进行更多的介绍。

如果你现在运行 `docker images`，就会看到下面的仓库中列出了：

- dockerinaction/ch3_myapp
- dockerinaction/ch3_myotherapp
- java:6

在默认情况下，`docker image` 命令将只显示你的仓库。类似于其他命令，如果指定 `-a` 标志，该列表将包括每一个安装的中间镜像层。运行 `docker images -a` 将显示，包括<none>多个仓库的列表。唯一指定镜像的办法是使用在镜像 ID 列的值。

在这个示例中，你直接安装两个镜像，但也会安装第三方父仓库。你需要清理所有三种。你这样做更容易：

```
docker rmi \
    dockerinaction/ch3_myapp \
    dockerinaction/ch3_myotherapp \
    java:6
```

`docker rmi` 命令允许你指定要删除的镜像，列表用空格分隔。你需要在这个示例后用这种方式很方便地删除一小部分镜像。我将在本书后面的示例中使用这个命令。

3.3.2 分层关系

镜像维护着父/子依赖关系。在这些依赖关系中，从父层构建形成新的一层。容器中的文件是镜像所创建容器的所有层合集。镜像可以与任何其他镜像有依赖关系，包括为不同的所有者提供不同仓库的镜像。第 3.3.1 节中的两个镜像使用 Java 6 镜像作为它们的父层。两个镜像完整的渊源关系如图 3-7 所示。

图 3-7 中的分层就是 java 6 镜像的示例。镜像作者可以打标签、发布，也可命名。用户可以创建别名，就像第 2 章使用 `docker tag` 命令。在镜像打好标签之前，指代镜像的唯一方法是创建镜像时所生成的唯一标识符（UID）。在图 3-7 中，常见的 Java 6 镜像的父层使用其 UID 的前 12 位数字来标示。这些层包含公共库和 Java 6 软件的依赖。考虑到用户的使用，Docker 将 UID 从 65（16 进制）个数字截断为 12 个。内部通过 API 访问，Docker 采用全 65 个数字。当你安装了类似未命名的镜像，这点要特别注意。当你使用 `docker images` 命令看到这些数字的话，我不希望你觉得有什么不好的事情或者一些恶意软件已经植入到你的机器中。

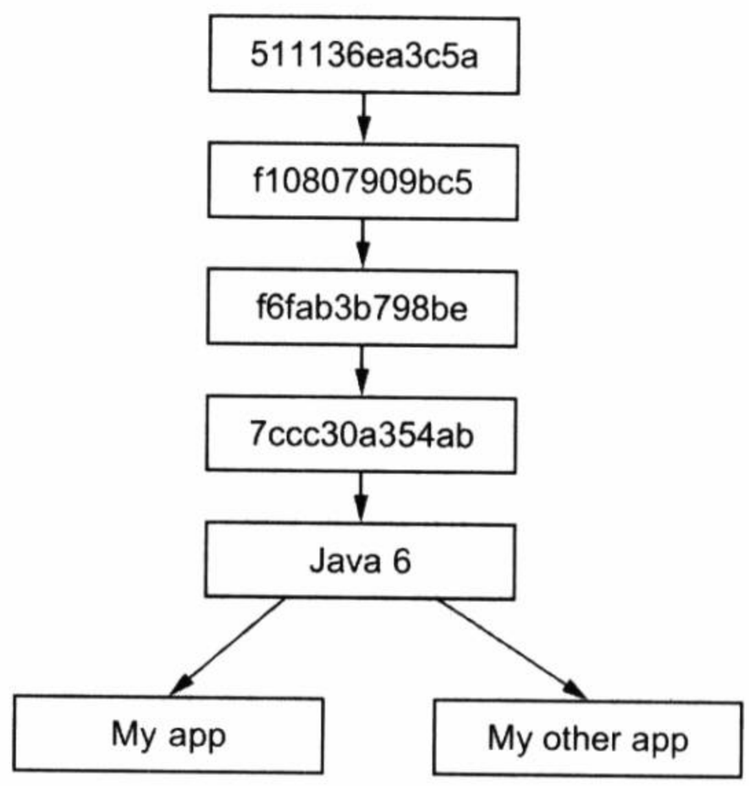


图 3-7 在 3.3.1 节中使用的两个 Docker 镜像的渊源关系

Java 镜像是相当大的。在写本节时，openjdk-6 镜像大小是 348 MB，而在 openjdk-7 镜像大小是 590 MB。当你运行镜像时，会节省一些空间，但即便如此，openjre-6 的大小也有 200 MB。再次，这里选择 Java，因为它是一个基础依赖。

3.3.3 容器文件系统抽象和隔离

容器中正在运行的程序对镜像分层一无所知，仿佛该操作不是在容器中或操作镜像上运行的。从容器的角度看，它具有由镜像所提供文件的独占副本。这就是所谓 Union 文件系统。Docker 支持多种 Union 文件系统，并给出适合你的最佳选择。至于 Union 文件系统是如何工作的，对于如何有效地使用 Docker 而言，意义不大。

Union 文件系统是创建有效文件系统隔离极为关键的一套工具，其他工具还有 MNT 命名空间和 chroot 系统调用。

该文件系统在你的主机系统上创建挂载点，对分层使用进行了封装。所创建的分层都被捆绑到 Docker 镜像层。同样，在安装 Docker 镜像时，分层进行解压缩并由文件提供商为你的系统进行合理配置。

Linux 内核提供了 MNT 系统命名空间。当 Docker 创建一个容器，这个新的容器会有自己的 MNT 命名空间，以及为镜像创建新的挂载点。

最后，`chroot` 通过容器上下文来构建镜像的根文件系统。这可以防止运行于容器内的任何程序与主机系统的其他部分有所关联。

`chroot` 和 MNT 命名空间是常见的容器技术。再加上 Union 文件系统，给 Docker 容器带来很多好处。

3.3.4 分层文件系统及其工具的优点

第一，也许是最重要的优点，即公共层仅需安装一次。如果想安装任何数目的镜像，它们都依赖于公共层，即公共层以及它的所有父层，都只需被下载或安装一次。这意味着你只要安装一个程序，不用保存冗余文件或者下载冗余分层。相比之下，大多数虚拟机技术会多次存储相同的文件，因为一台机器上会有多个虚拟机。

第二，分层提供了用于依赖管理和隔离的工具。软件作者特别得心应手，第 7 章会谈得更多一些。从用户的角度来看，这样做的好处是帮助你通过检查镜像和分层来识别运行的软件。

最后，很容易地构建专业的软件，因为你只要在某个基本镜像上做些细微的变化就可以了，这会在第 7 章有详细介绍。因此提供专门的镜像，可帮助用户从最小的定制开始，获取他们想要的，这就是使用 Docker 的最好原因之一。

3.3.5 Union 文件系统的不足

Docker 可选择运行最佳的文件系统，但对每个工作负载而言，并没有完美的实现。事实上，有一些具体的使用情况，你应该停下来考虑使用其他 Docker 功能。

不同的文件系统对于文件属性、大小、名称和字符都有不同的规则。Union 文件系统经常需要在不同的文件系统规则之间进行转换。在情况好的时候，它们能够提供可接受的转换。不好的时候，某些功能则无法工作。例如，btrfs 和 OverlayFS 虽然提供了扩展属性，但会导致 SELinux 无法工作。

Union 文件系统使用一种称为写时复制的模式，这使得内存映射文件（`mmap()`）的系统

调用）的实现比较困难。一些 Union 文件系统提供了在适当的条件下的某种实现方式，但其实避免镜像中使用内存映射文件，这会是一个好主意。

备份文件系统是 Docker 另一种可插拔的功能。你可以通过使用 info 命令来决定安装哪些文件系统。如果你想明确告诉 Docker 使用哪个文件系统，在启动 Docker 守护进程时，就加上`--storage-driver`或`-s`选项。写入 Union 文件系统出现问题，大多可以在不改变存储供应商的前提下来解决，如可用存储卷，这是第 4 章的主题。

3.4 小结

软件的安装和管理任务带来了一些独特的挑战。本章介绍了如何使用 Docker 来解决这些问题。本章涵盖的核心理念和内容如下：

- Docker 用户可使用仓库名称来确定他们想通过 Docker 安装的软件。
- Docker Hub 是默认的注册服务器，可以通过网站或 `docker` 命令行工具在 Docker Hub 上找到需要的软件。
- docker 命令行工具可以很方便地通过其他注册服务器或其他形式来分发安装软件。
- 镜像仓库配置中会包括注册服务器的主机信息。
- `docker load` 和 `docker save` 命令可以将 TAR 档案文件用来加载和保存镜像。
- 分发一个项目的 Dockerfile 可简化用户机器上构建镜像的过程。
- 镜像通常和其他镜像都有父 / 子的关联。这些关系构成分层。当我们说，我们已经安装了一个镜像，我们是说已经安装了一个目标镜像，及其依赖的每个镜像层。
- 使用分层构建镜像，可重用分层，并可节省分发带宽和机器上的存储空间。

第 4 章　持久化存储和卷间状态共享

4

本章介绍

- 存储卷的简介
- 存储卷的两种类型
- 宿主机和容器之间如何共享数据
- 容器之间如何共享数据
- 存储卷的生命周期
- 存储卷之间的数据管理和控制模式

此时此刻，你已经安装和运行了一些程序。然而你看到的这些程序示例，并不能反映出真实的世界。前三章的示例和现实的区别在于，现实中的程序需要数据才能运行。本章介绍了 Docker 存储卷以及容器之间如何管理数据的方法。

就像在容器中运行一个数据库程序一样，你可以将这个软件打包在镜像中，当启动容器时，它将会初始化一个空数据库。当其他程序接入该数据库并存入数据，这些数据要如何才能保存下来呢？是以容器中一个文件的形式吗？当你暂停一个容器或者删除这个容器，这些数据要怎么办？如果你想要升级数据库程序，这些数据怎么搬迁？

考虑一下另一种情况，一组不同的 web 应用程序运行在在不同的容器中。为了让它们的日志文件保存在容器之外，你准备把它们写到哪儿去呢？你将如何取得这些日志来排除故障呢？如何让其他程序（比如日志摘要工具）来访问这些文件呢？这些问题的答案就涉及存储卷的使用。

4.1 存储卷的简介

一个主机或容器的目录树是由一组挂载点创建而成，这些挂载点描述了如何能构建出一个或多个文件系统 。存储卷是容器目录树上的挂载点，其中一部分主机目录树已经被挂载了。大多数人只熟悉一点点文件系统和挂载点，很少对其进行定制。比起任何其他的 Docker 话题，存储卷会更难一些。这与对挂载点的缺乏了解有关。

如果没有存储卷，Docker 用户会受限于 Union 文件系统，仅提供镜像挂载。如图 4-1 所示容器中运行着的一个程序，正写数据到文件中。第一个文件写入到了根文件系统。操作系统控制根文件系统将改变的部分装入 Union 文件系统的顶层。第二个文件则写入到已经挂载于容器目录树/data 中。改动会通过存储卷直接影响到主机文件系统上。

虽然 Union 文件系统适用于构建和分享镜像，但对持久化或共享数据而言，并不是理想的方法。存储卷填补了这些用例，并在容器化系统设计中发挥了关键作用。

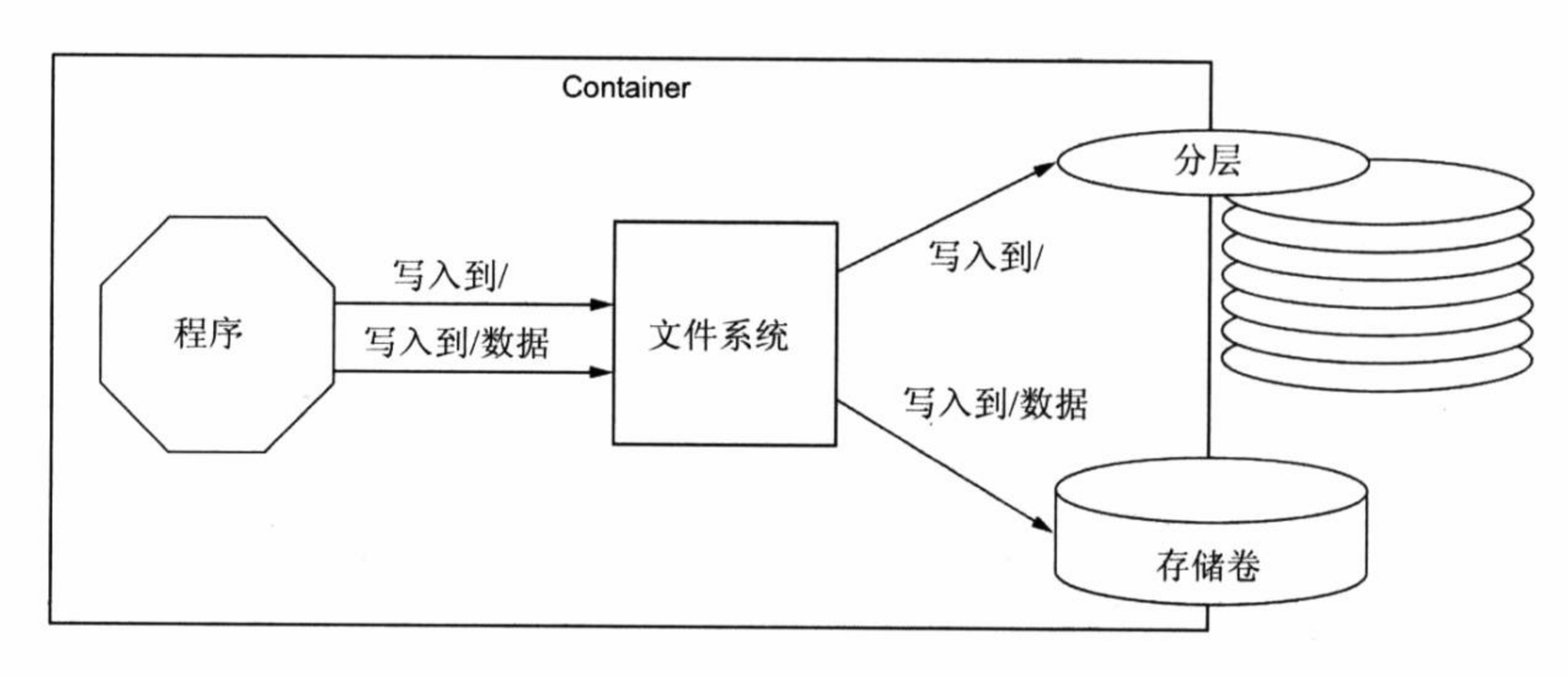


图 4-1 容器挂载的存储卷是 Union 文件系统顶层可写的部分

4.1.1 存储卷提供容器无关的数据管理方式

从语义上来说，存储卷是一个数据分割和共享的工具，有一个与容器无关的范围或生命周期。这使得存储卷成为了容器化系统设计中关于文件分享或写入最重要的一部分。数据示例根据其范围或者接入容器方式的不同，分成以下几种：

- 数据库软件与数据库中的数据
- web 应用程序与日志数据
- web 数据处理应用程序的输入和输出数据
- web 服务器与静态内容
- 产品与支持工具

存储卷可分离关注点，并为架构组件创建模块化。这种模块化设计帮助你更轻松地了解、构建、支持和重用大型系统的部件。

按以下方式来思考：镜像适合打包和分发相对静态的文件，如程序；存储卷则持有动态或专门数据。这种区别使得镜像可重用，数据也可以简单分享。这种相对静态和动态文件空间的分离，允许应用程序或镜像的作者，实现高级模式，例如多态和可组合工具。

多态工具维护一致的接口，但可能有多种实现，分别做不同的事情。考虑这样一个应用程序，如一个通用的应用服务器。例如，Apache Tomcat，在网上提供了一个 HTTP 接口并分发其接收到的任何请求给可插拔的各类程序。因此 Tomcat 具有多态行为。使用存储卷可以将其行为配置插入到容器中，而无须修改镜像。另外，考虑 MongoDB 或 MySQL 这样的数据库。数据库的值由其包含的数据所定义。数据库程序总呈现相同的接口，但其值则完全不同，这取决于插入卷的数据。多态容器模式是第 4.5.3 节的主题。

更为基本的是，存储卷可以隔离应用程序和主机的关系。镜像被装载到主机，创建出一个容器。Docker 不知道主机在哪里运行，只能判断哪些文件在容器中可用。这意味着 Docker 本身就没有办法利用主机上的设施，如装载的网络存储，或混合光纤和固态硬盘。但有主机知识的用户可以使用存储卷，在容器中将这些目录映射到主机的存储上。

现在，你已经熟悉了存储卷及其重要性，可以在一个真实的示例中开始使用它们。

4.1.2 NoSQL 数据库使用存储卷

Apache Cassandra 项目提供了一个具有内置集群，最终一致性和线性写入可伸缩的列数据库。这是现代系统设计的热门选择，其正式的镜像也可在 Docker Hub 找到。Cassandra 就像其他的数据库一样，在磁盘文件上存储其数据。在本节中，你将使用官方的 Cassandra 镜像创造一个单节点集群，并创建一个键空间，删除容器，然后在另一个容器中恢复这个新节点上的键空间。

我们通过创建已定义存储卷的单个容器来开始，这被称为存储卷容器。存储卷容器是本章后面讨论的高级模式之一。

```
docker run -d \
    --volume /var/lib/cassandra/data \
    --name cass-shared \
    alpine echo Data Container
```

在容器中指定存储卷的挂载点

存储卷容器将立即停止，这符合了本示例的目的。不要删除它。你会在创建运行 Cassandra 新容器时，使用这个存储卷：

```
docker run -d \
    --volumes-from cass-shared \
    --name cass1 \
    cassandra:2.2
```

继承了存储卷的定义

Docker 从 Docker Hub 下载 cassandra:2.2 镜像，创建一个新容器，并复制存储卷容器的卷定义。然后，容器的存储卷挂载在/var/lib/cassandra/data，指向主机目录树相同的位置。接下来，从 cassandra:2.2 镜像启动容器，运行 Cassandra 客户端工具，并连接到正在运行的服务器：

```
docker run -it --rm \
    --link cass1:cass \
    cassandra:2.2 cqlsh cass
```

现在，你可以从 CQLSH 命令行检查或修改 Cassandra 数据库。首先，查找一个名为 docker_hello_world 的键空间：

```
select *
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

Cassandra 应该返回一个空列表。这意味着该示例的数据库尚未进行修改。接下来，用以下命令创建键空间：

```
create keyspace docker_hello_world
with replication = {
    'class' : 'SimpleStrategy',
    'replication_factor': 1
};
```

现在，你已经修改了数据库，再发相同的查询，应该可以看到返回结果，也可确认你的更改被数据库已接受。下面的命令和前面是一样的：

```
select *
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

这一次 Cassandra 应该返回刚才所创建的键空间的下一个条目。如果你确信已经连接并修改了 Cassandra 节点，就可退出 CQLSH 程序并停止客户端容器：

```
# Leave and stop the current container
quit
```

客户端容器创建时，使用--rm 标志，在命令停止会被自动删除。然后通过停止和去除所创建的 Cassandra 节点，清理该示例的第一部分：

```
docker stop cass1
docker rm -vf cass1
```

你创建的 Cassandra 客户端和服务器，都将在执行这些命令后被删除。如果你所做的修改需要持久化，唯一的办法是存储卷容器。这样数据的范围已扩大到两个容器，它的生命周期已经延长到数据生成时所在的容器之外了。

你可以重复这些步骤进行测试。创建一个新的 Cassandra 节点，连接客户端，并且查询键空间。图 4-2 描述了整个系统，即你已经建立的一切。

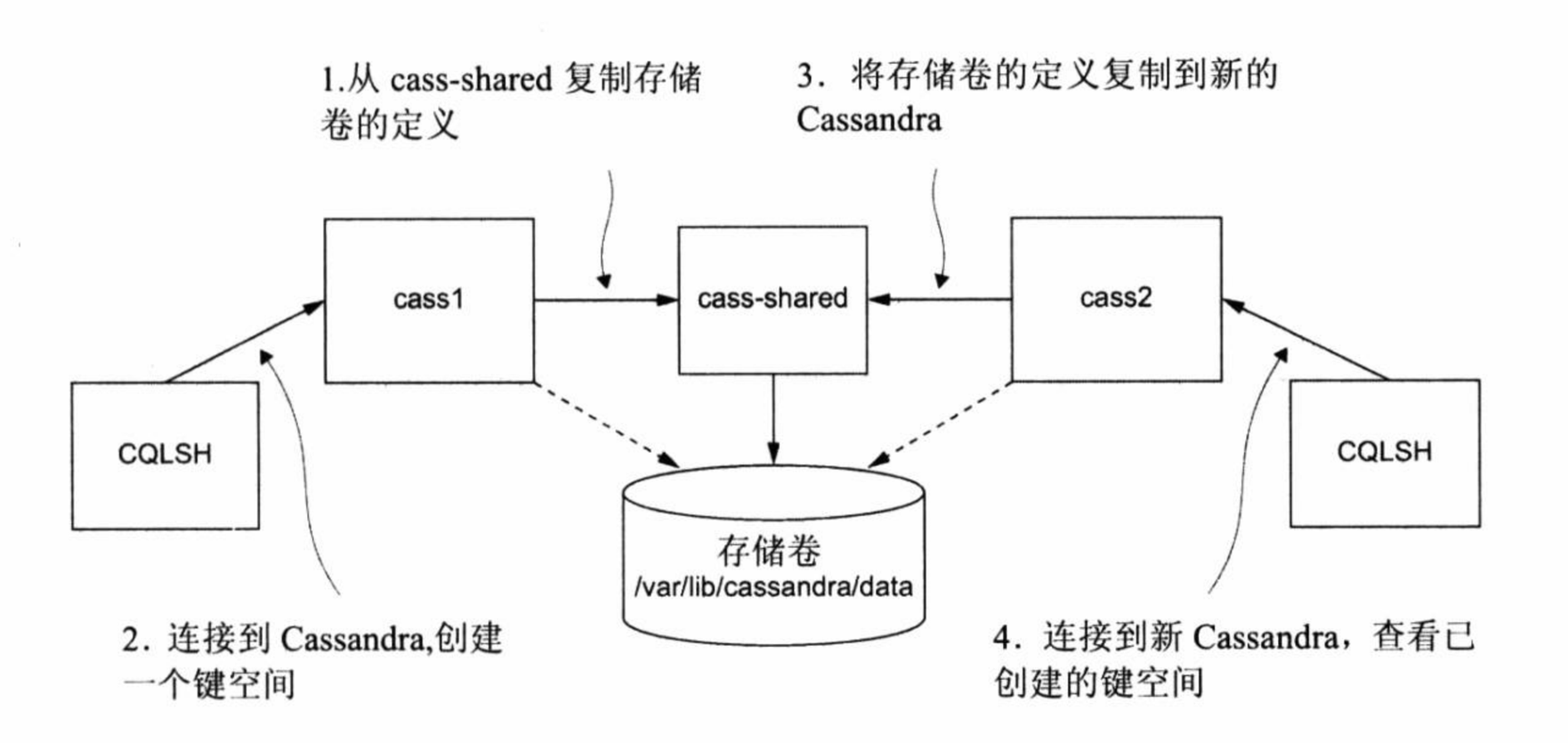


图 4-2 创建和恢复 Cassandra 上存储数据的关键步骤

接下来的三个命令将测试数据的恢复：

```
docker run -d \
   --volumes-from cass-shared \
   --name cass2 \
   cassandra:2.2

docker run -it --rm \
   --link cass2:cass \
   cassandra:2.2 \
   cqlsh cass

select *
```

```
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

最后一个命令返回单个条目，将会和你前面一个容器中创建的键空间相匹配。这证实了以前的要求，说明了存储卷如何被用于创建持久化的系统。往下继续之前，先退出 CQLSH 程序并清理工作区。确保删除该存储卷容器：

```
quit
docker rm -vf cass2 cass-shared
```

这个示例演示了存储卷使用的一种方式，但尚未了解它们如何工作、使用的模式，或者如何管理存储卷的生命周期。这一章的其他部分将深入到存储卷的各个方面，先从其提供可用的不同类型开始。

4.2 存储卷的类型

存储卷有两种类型。每一个存储卷就是容器目录树的挂载点在主机目录树中的位置，但不同的存储卷类型在主机的位置是不同的。第一种类型是绑定挂载存储卷。绑定挂载存储卷使用用户提供的主机目录或文件。第二类型是管理存储卷。管理存储卷使用由 Docker 守护进程控制的位置，被称为 Docker 管理空间。存储卷类型如图 4-3 所示。

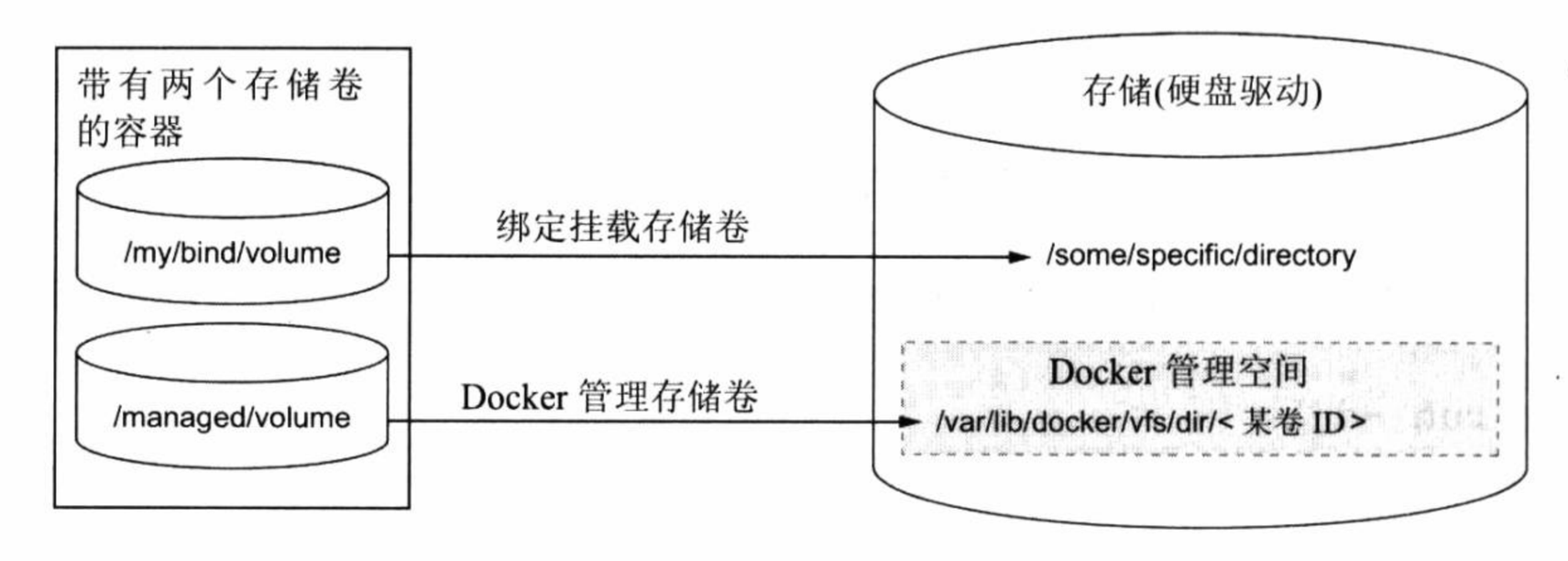


图 4-3 Docker 提供了绑定挂载存储卷和管理存储卷

每种存储卷类型都有优缺点。须根据你的具体使用情况选择使用。本节将深入探讨每一种类型。

4.2.1 绑定挂载卷

绑定挂载卷是一种存储卷，指向主机文件系统上用户指定的位置。绑定挂载卷在主机

提供的文件或目录需要挂载到容器目录的特定位置时，非常有用，如图 4.4 所示。

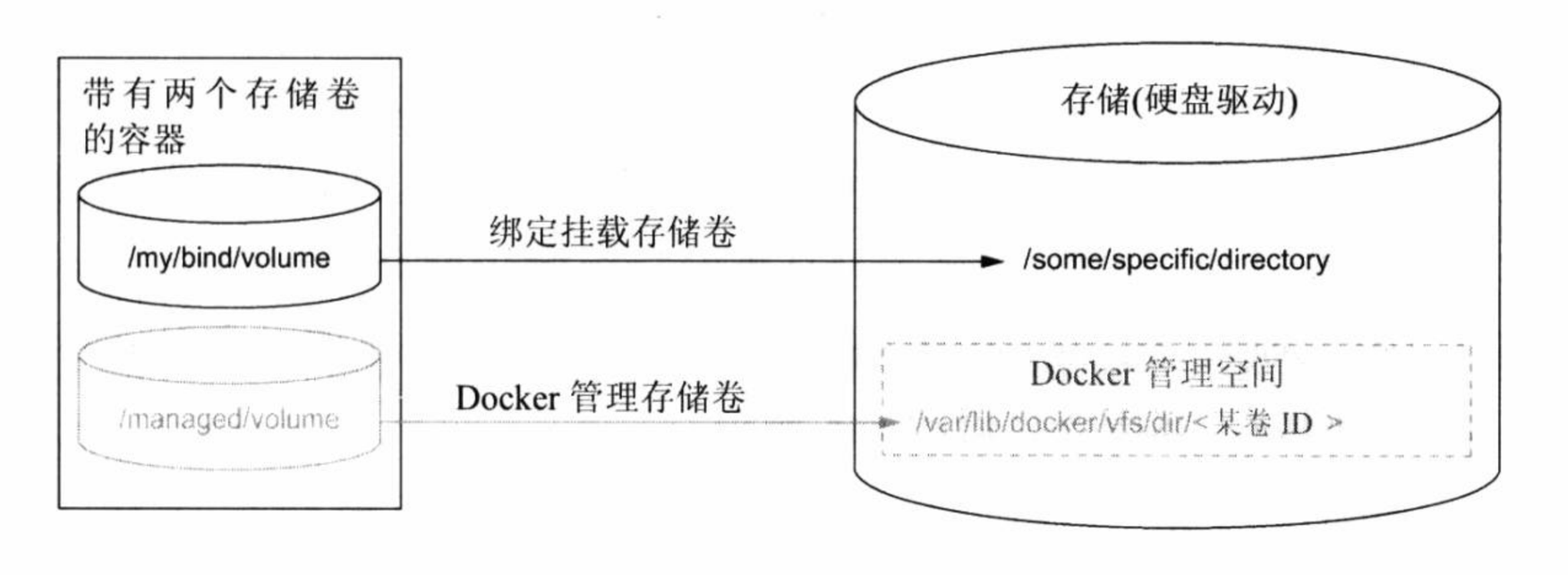


图 4-4 主机目录作为绑定挂载卷

如果你想将数据分享给运行在容器外的进程，比如主机系统组件，绑定挂载卷是很有用的。如果你想将主机数据通过已知的主机目录分享给容器中运行的特定程序，绑定挂载卷也管用。

例如，假设你在本地计算机上处理文档或网页，并希望与朋友分享你的工作。一个方法是使用 Docker 启动 web 服务器，并将工作内容复制到 web 服务器的镜像中。虽然这个方法可行，甚至也可能是生产环境的最佳实践，但这很麻烦，每次你要更新共享文件的版本，都得重建镜像。

相反，你可以使用 Docker 启动 web 服务器，并绑定挂载你的文档地址在 web 服务器新容器的文档根目录上。在主目录创建一个名为 example-docs 的新目录。在该目录中创建一个名为 index.html 的文件。在这个文件中，为你的朋友，写入一个不错的消息。下面的命令将启动一个 Apache HTTP 服务器，你的新目录会被绑定挂载在服务器的文档根目录中：

```
docker run -d --name bmweb \
    -v ~/example-docs:/usr/local/apache2/htdocs \
    -p 80:80 \
    httpd:latest
```

有了这个运行的容器，能够将 web 浏览器指向 Docker 引擎运行的 IP 地址，能够看到你创建的文件。

在这个示例中，你使用了-v 选项和位置映射来创建绑定挂载卷。该映射以冒号分隔（这是 Linux 命令行工具的常见风格）。映射键（冒号之前的路径）是主机文件系统上的一个绝对路径，该键值（冒号后的路径）是容器中挂载的目标存储位置。你必须使用绝对路径指定该位置。

这个示例展示了存储卷的一个重要属性或特征。当你在一个容器文件系统挂载存储卷，

它取代了镜像在该位置提供的内容。本示例中，httpd:latest 的镜像在/usr/local/apache2/htdocs 提供某些默认的 HTML 内容，但当存储卷挂载在该位置时，镜像提供的内容被主机上的内容所覆盖。这种行为是本章后面将讨论的多态容器模式的基础。

扩展一下该示例，假设你想确保在 Apache HTTP web 服务器不能更改此卷的内容。即使是最值得信赖的软件也可能包含漏洞，且最好尽量减少网站攻击的影响。幸运的是，Docker 提供了一种机制将挂载卷和存储卷设为只读权限。你可以通过在存储卷映射规则后追加：ro 来完成。例如，你应该会修改 run 命令：

```
docker rm -vf bmweb

docker run --name bmweb_ro \
    --volume ~/example-docs:/usr/local/apache2/htdocs/:ro \
    -p 80:80 \
    httpd:latest
```

通过挂载只读卷，可以避免容器内的任何进程修改该卷的内容。你可以通过运行一个快速测试，看到：

```
docker run --rm \
    -v ~/example-docs:/testspace:ro \
    alpine \
    /bin/sh -c 'echo test > /testspace/test'
```

此命令启动一个 web 服务器的容器，绑定挂载卷为只读。运行时，试图将单词“test”添加到卷上一个名为 test 的文件中。由于卷被挂载成只读，该命令失败。

最后要注意的是，如果你指定了一个不存在的主机目录，Docker 会为你创建相应的目录。虽然这可以派上用场，但依靠这个功能并不是个好主意，最好在这个目录上对权限设置有更多的管控。

```
ls ~/example-docs/absent                                          ← 验证 absent 目录不存在

docker run --rm -v ~/example-docs/absent:/absent alpine:latest \
        /bin/sh -c 'mount | grep absent'                          ← 检查挂载卷的定义

ls ~/example-docs/absent                                          ← 检查已创建的目录
```

绑定挂载卷并不仅限于目录，但目录是它们常用的方式。你可以使用绑定挂载卷装入单个文件。在创建或链接资源时，避免了与其他资源的冲突，提供了灵活性。考虑一下当你要安装一个特定的文件到一个包含其他文件的目录。具体而言，假设你只想在镜像分发的 web 内容外添加一个额外的文件。如果你用一整个目录绑定挂载到该位置，那么其他文

件都将丢失。通过使用一个特定的文件作为存储卷，可以只覆盖或插入单个文件。

在这种情况下，需要注意的重点是，文件必须在创建容器之前就存在于主机上。否则 Docker 会认为你想用一个目录，并在主机上创建它，把它挂载在需要的位置（即使该位置由另一个文件占用）。

使用绑定挂载卷的第一个问题是是它们将可移植容器绑定到特定主机的文件系统。如果容器的定义取决于主机文件系统特定位置上的内容，无论所在位置的内容可用或不可用，该定义无法跨主机移植。

第二个问题是，创造了与其他容器发生冲突的机会。启动 Cassandra 的多个实例，都使用相同的主机位置挂载存储卷，这将是一个糟糕的主意。在这种情况下，每个实例将竞争相同的一组文件。如果没有其他工具，比如文件锁定，这将有可能导致数据库损坏。

绑定挂载卷比较适合需要使用特殊挂载点的工作站或机器。最好在通用平台或硬件池避免这类特定的绑定。你可以使用 Docker 管理卷，以与主机无关和便携的方式利用该存储卷。

4.2.2 Docker 管理卷

管理卷不同于绑定挂载卷，这是因为 Docker 守护程序会在主机文件系统中创建存储卷，并由 Docker 管理，如图 4-5 所示。

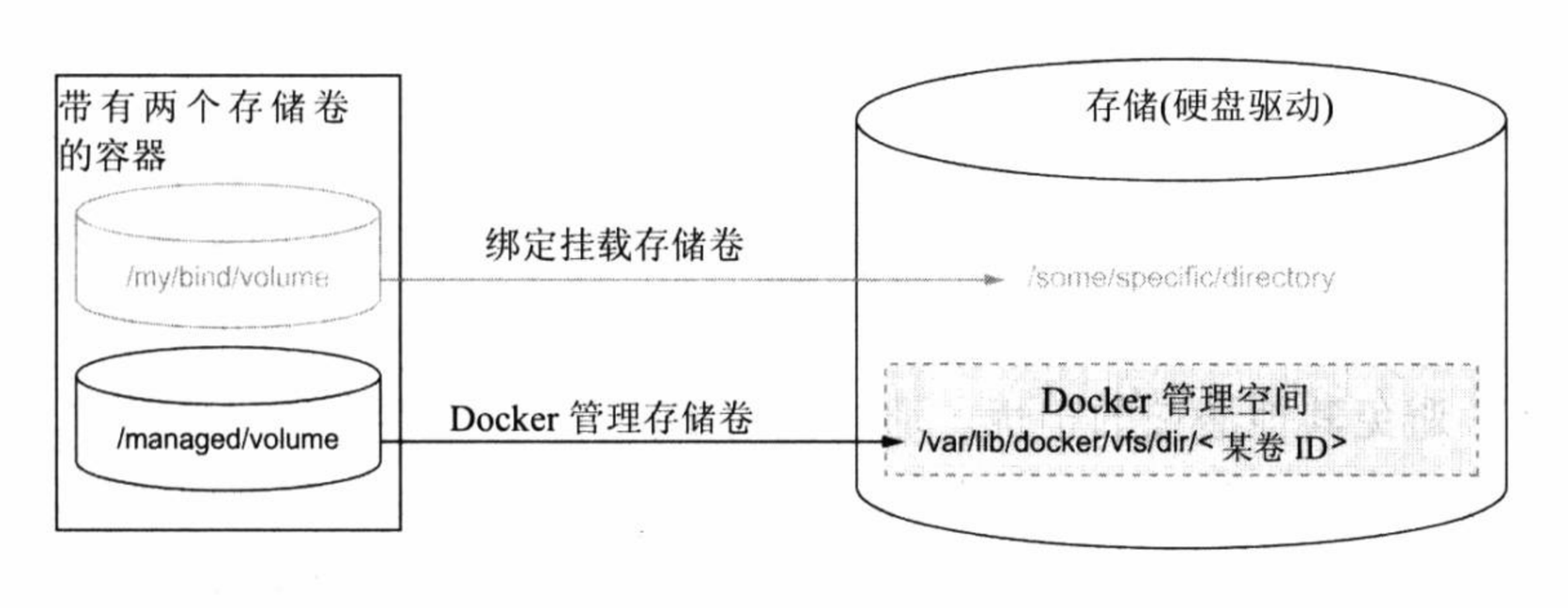


图 4-5 Docher 管理空间的一个目录挂载为存储卷

使用管理卷是一种在文件系统特定位置解耦卷的方法。当你执行 `docker run` 使用`-v`选项（或`--volume`），只要指定容器目录树中的挂载点，管理卷即可创建。你已经为 4.1.2 节中的 Cassandra 示例创建了管理卷。名为 cass-shared 的容器，在/var/lib/Cassandra/data 指定了一个存储卷：

```
docker run -d \
    -v /var/lib/cassandra/data \
    --name cass-shared \
    alpine echo Data Container
```

容器中指定存储卷的挂载点

当你创建了这个容器时，Docker 守护程序在主机文件系统中创建了目录，来存储这三个存储卷的内容。为了找到这个目录的确切位置，你可以使用 docker inspect 命令过滤卷键。这个输出结果说明了，Docker 为每个存储卷创建的目录是由主机上的 Docker 守护进程控制的：

```
docker inspect -f "{{json .Volumes}}" cass-shared
```

inspect 子命令将输出容器挂载点的列表，以及在主机目录树中的相应路径。输出将是这样的：

```
{"/var/lib/cassandra/data":"/mnt/sda1/var/lib/docker/vfs/dir/632fa59c...
"}
```

卷键指向这个映射的键值。在这个映射中，每个键都是容器中的挂载点，键值是主机文件系统上的目录位置。在这里，我们已经检视了容器的一个卷。该映射对键的字典式排序，和创建容器时所指定的顺序无关。

提示 VirtualBox（Docker Machine 或 Boot2Docker）用户应记住，在每个键值中指定的主机路径是相对于虚拟机的根文件系统，而不是其主机的。管理卷是正在运行的 Docker 守护程序在虚拟机上创建的，但 VirtualBox 绑定挂载卷所引用的确实是该主机上的目录或文件。

Docker 管理卷，在桌面上手动构建或链接工具时似乎很难用，但在较大的系统中数据的具体位置是不太重要的，所示管理卷以一个更有效的方式来组织你的数据。使用它们，可解耦存储卷，避免其他潜在的担忧。通过使用 Docker 管理卷，你只要简单地指出，“我需要一个地方放一些我工作的数据。”这是可以在任何计算机安装和使用 Docker 的前提和要求。此外，当一个卷使用完，你告诉 Docker 清理这些东西，Docker 可以放心删除不再由容器使用的任何目录或文件。以这种方式，使用管理卷可以帮忙管理混乱。随着 Docker 的中间件或插件的发展，管理卷用户将能够获得更高级的功能，如可移植存储卷。

数据的共享访问是存储卷的一个重要特征。如果你已经从文件系统已知位置解耦了存储卷，你需要知道容器之间如何共享存储卷，且不会暴露容器的确切位置。下一节介绍两个容器间存储卷数据共享的方法。

4.3 共享存储卷

假设你有一个跑在容器中的 web 服务器，所有接收到的请求都记录在/logs/access。如果要将这些日志从你的 web 服务器转移到持久化存储上，你可能会想通过另一个容器中的脚本来做到。这时候容器间的卷分享的价值就更加明显。正如有两种类型的存储卷，这儿也有两种方法来共享容器之间的存储卷。

4.3.1 主机依赖的共享

你已经知道实现主机依赖的共享所需的工具了。两个或多个容器使用主机依赖的共享，即每个容器在主机文件系统的已知位置有一个绑定挂载卷。这是容器之间存储空间共享的最明显的方法。你可以看到下面的示例：

```
mkdir ~/web-logs-example                          ← 创建以知目录

docker run --name plath -d \
    -v ~/web-logs-example:/data \                 绑定挂载该目录为日志可写容器
    dockerinaction/ch4_writer_a

docker run --rm \
    -v ~/web-logs-example:/reader-data \          绑定挂载该目录为日志只读容器
    alpine:latest \
    head /reader-data/logA

cat ~/web-logs-example/logA                       ← 查看主机上的日志

停止可写容器 → docker stop plath
```

在这个示例中，你创建了两个容器：一个名为 plath 的容器写文件，另一个则读该文件。这些容器都有一个共同的绑定挂载卷。容器之外，你可以通过列出目录中的内容或查看新的文件来观察这些变化。

探索一下容器可能以这种方式连接在一起。接下来的示例启动四个容器，两个日志写入者和两个读取者：

```
docker run --name woolf -d \
    --volume ~/web-logs-example:/data \
    dockerinaction/ch4_writer_a

docker run --name alcott -d \
    -v ~/web-logs-example:/data \
    dockerinaction/ch4_writer_b
```

```
docker run --rm --entrypoint head \
    -v ~/web-logs-example:/towatch:ro \
    alpine:latest \
    /towatch/logA

docker run --rm \
    -v ~/web-logs-example:/toread:ro \
    alpine:latest \
    head /toread/logB
```

在这个示例中，你创建了四个容器，每一个都有绑定挂载卷。前两个容器写入到卷的不同文件中。第三和第四个容器则在不同的位置挂载卷，并作为只读。这是一个玩具示例，但它清楚地展示了一个功能，即有多种方法可构建镜像和软件。

主机依赖的共享要求你使用绑定挂载卷，如果你有大量的机器，主机依赖的共享可能会导致问题或维护起来过于昂贵。下一节展示了在一组容器中共享管理卷和绑定挂载卷的快捷方式。

4.3.2 共享和 volumes-from 标志

docker run 命令提供了一个标志，可将卷从一个或多个容器复制到新的容器中。标志--volumes，可设定多次，可指定多个源容器。

你可以在 4.1.2 节使用该标志，将由存储卷容器定义的管理卷复制到 Cassandra 容器中。这个示例很现实，但是并没有说明--volumes-from 标志和管理卷容器的几个特定行为：

```
docker run --name fowler \
    -v ~/example-books:/library/PoEAA \
    -v /library/DSL \
    alpine:latest \
    echo "Fowler collection created."
docker run --name knuth \
    -v /library/TAoCP.vol1 \
    -v /library/TAoCP.vol2 \
    -v /library/TAoCP.vol3 \
    -v /library/TAoCP.vol4.a \
    alpine:latest \
    echo "Knuth collection created"
```

```
docker run --name reader \
    --volumes-from fowler \
    --volumes-from knuth \
    alpine:latest ls -l /library/

docker inspect --format "{{json .Volumes}}" reader
```

列出所有复制到新容器的存储卷

检查新容器的卷列表

在这个示例中，你创建了两个容器，一个定义了 Docker 管理卷，另一个定义了绑定挂载卷。想将这些分享给没有--volumes-from 标志的第三个容器，你需要检查先前创建的容器，然后绑定挂载卷到 Docker 管理的主机目录。当你使用--volumes-from 标志，Docker 会为你做到这一切。复制任何本卷所引用的源容器到新的容器中。在这种情况下，名为 reader 的容器，复制由 fowler 和 knuth 定义的所有卷。

你可以直接或传递地复制卷。这意味着，如果你能从另一个容器复制卷，你还可以复制从其他容器复制得来的存储卷。使用上一个示例中所创建的容器，如下：

```
docker run --name aggregator \
    --volumes-from fowler \
    --volumes-from knuth \
    alpine:latest \
    echo "Collection Created."

docker run --rm \
    --volumes-from aggregator \
    alpine:latest \
    ls -l /library/
```

创建这样一个容器

复制了其他两个容器的存储卷，使用该容器的存储卷，查看指定目录

复制卷始终具有相同的挂载点。这意味着，你不能在三种情况下使用--volumes-from。

第一种情况，如果你构建的容器需要共享卷挂载到不同的位置，你不能使用--volumes-from。没有工具也无法提供重新映射该挂载点。它只会通过指定的容器中指定的挂载点进行复制。例如，如果在上一个示例中，student 容器想挂载该库到像/school/library 的位置，它们将不能够这样做。

第二种情况，源卷之间彼此冲突，或者有新的卷规格。如果一个或多个源创建的管理卷具有相同挂载点，将只能接收其中之一：

```
docker run --name chomsky --volume /library/ss \
   alpine:latest echo "Chomsky collection created."
docker run --name lamport --volume /library/ss \
   alpine:latest echo "Lamport collection created."

docker run --name student \
   --volumes-from chomsky --volumes-from lamport \
   alpine:latest ls -l /library/

docker inspect -f "{{json .Volumes}}" student
```

当你运行该示例时，因为有两个源卷使用相同的挂载点，docker inspect 会显示最后一个容器只能有一个存储卷挂载在/library/ss。每个源容器定义了相同的挂载点，复制到新容器的两个卷会产生一个竞争状态。两个复制操作中，只有一个能够成功。

一个实际的例子，会受到以下限制，如果你正复制多个 web 服务器的存储卷到一个容器中，而这些服务器都运行相同软件或共享公共配置（在容器化系统中不太可能），那么所有这些服务器可能使用相同的挂载点。在这种情况下，挂载点会发生冲突，而你只能够访问到所需数据的子集。

第三种情况，如果你需要更改卷的写权限，就不能使用--volumes-from。这是因为--volumes-from 复制了卷的定义。例如，如果你的源卷挂载了具有读/写访问的卷，想要给一个容器只读访问的权限，使用--volumes-from 将无法正常工作。

使用--volumes-from 标志共享存储卷是构建可移植的应用程序架构的重要工具，但它确实引入了一些限制。使用 Docker 托管卷解耦容器数据和主机文件系统，这对大多数生产环境至关重要。Docker 为托管卷创建的文件和目录，还需要进行核算和维护。要了解 Docker 如何与这些文件打交道，以及如何保持 Docker 环境的整洁，你需要了解托管卷的生命周期。

4.4 管理卷的生命周期

现在你应该有相当多的容器和存储卷需要进行清理。到目前为止，本书中没有讨论清理的方法，这节你会有丰富的工具可以使用。管理卷的生命周期独立于任何容器，但截至目前，你只能通过容器来引用它们。

4.4.1 管理卷的权限

管理卷是二等实体。你没有办法分享或删除特定的管理卷，因为你没有办法指定一个管理卷。

如果你不使用绑定挂载卷，只创建了管理存储卷的话，那么只能通过它们的容器来区分。

区分存储卷最好的方法是为每个管理卷定义一个容器。这样你可以很具体地了解用了哪些存储卷。更重要的是，这样做可以帮助你删除特定的存储卷。否则你需要在容器中检查卷的映射，并手动清理 Docker 管理空间。删除卷需要被引用的容器，重要的是了解哪些容器拥有这个管理卷。如图 4-6 所示。

一个容器拥有所有挂载在其文件系统上的管理卷，并且多个容器可以拥有一个像示例中 fowler、kunth、reader 这样的存储卷。Docker 跟踪管理卷的引用，以确保没有删除当前引用的存储卷。

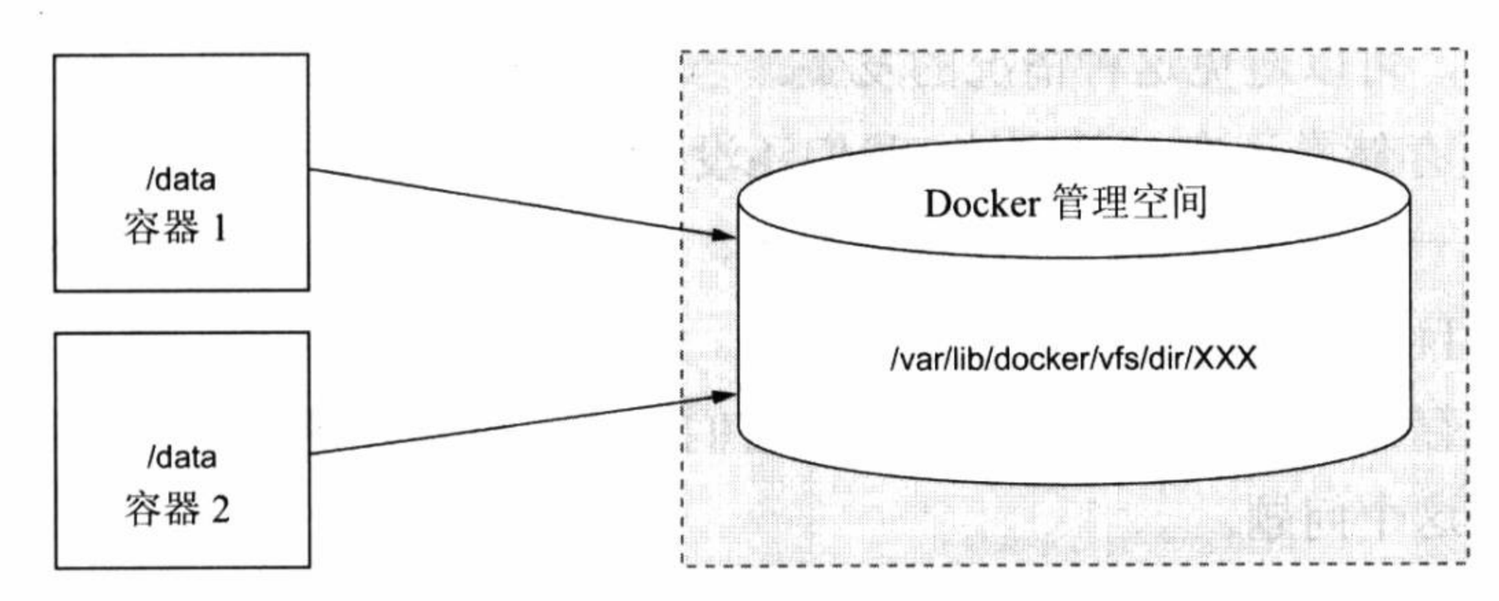


图 4-6 这两个容器对单个管理卷的所有权关系

4.4.2 存储卷的清理

清理管理卷是一个手动的任务。此默认功能可以防止意外损坏潜在的有价值的数据。Docker 守护程序无法删除绑定挂载卷，因为源卷不在 Docker 管理范围内。这样做可能会导致冲突、不稳定和数据意外丢失。

删除容器时，Docker 可以删除管理卷。运行带有-v 选项的 `docker rm` 命令将试图删除目标容器中引用的任何管理卷。任何有其他容器引用的管理卷将被忽略，但内部引用计数器仍会递减。这是一种默认的安全机制，但是它会导致如图 4-7 所示的问题。

如果删除每一个已引用管理卷的容器，但没有使用-v 标志，就会产生孤立卷。移除孤立卷，需要一系列手工操作，根据卷的大小，可能有时候这么做也不会很花时间。此外，你可能会考虑使用脚本清理一些孤立卷。在运行之前你应该仔细检查这些脚本，因为需要以特权用户来运行，如果它们包含恶意软件，你可能会失去系统的完全控制权。

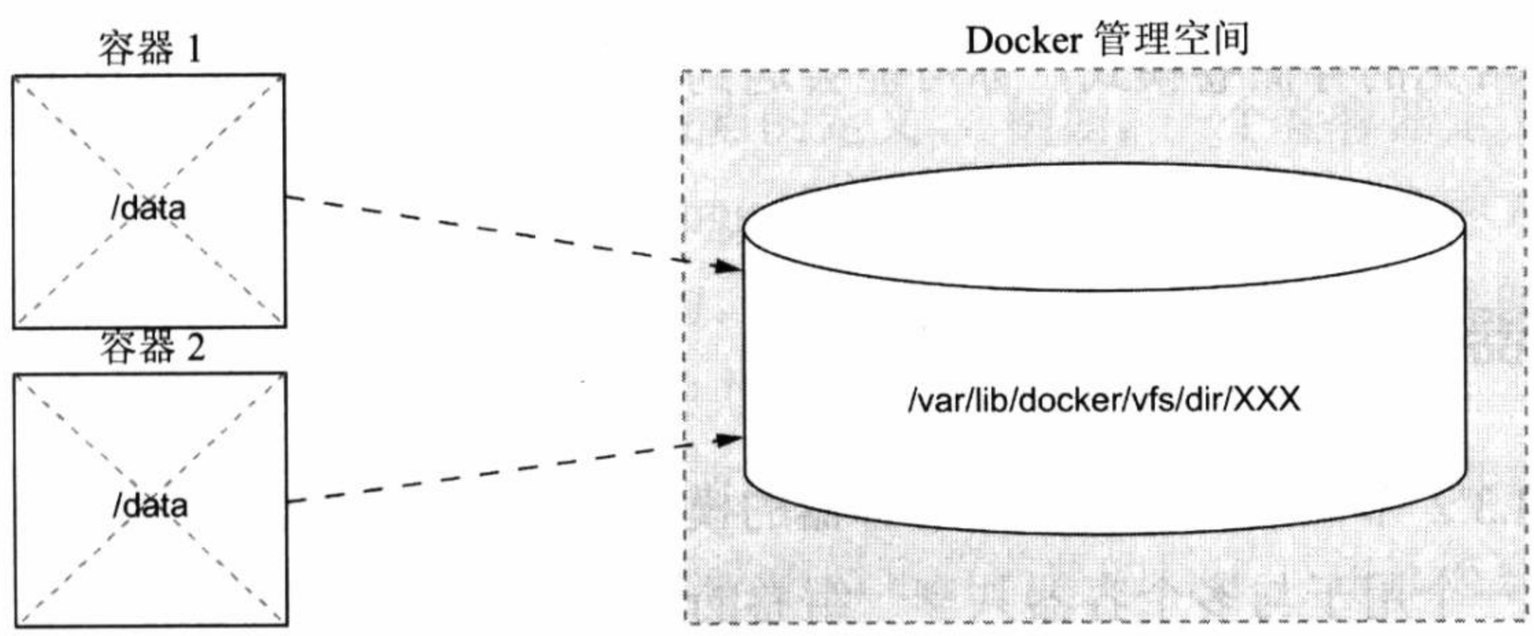


图 4-7 删除两个容器时，未告知 Docker 删除挂载于上述两个容器中的存储卷，因此该存储卷变成了一个孤立卷

还有一个更好的主意，通过使用-v 选项，并使用 4.5 节中讨论的卷容器模式，来获取

关键数据的习惯，可以避免这种情况的发生。

Docker 创建存储卷还有一种方法，我们还没有提到过。镜像元数据可以提供卷的规格，第七章包括了这一机制的细节。在这种情况下，你可能甚至不知道新容器创建的那些存储卷，这也是训练自己使用-v 选项最主要的原因。

孤立卷会使存储空间不可用，直到清除它们为止。你可以通过时常记得清理并使用卷容器模式来减少这个问题。

清理 进一步阅读之前，需要一些时间来清理你所创建的容器。使用 docker ps -a 得到这些容器的列表，并记得使用 docker rm 的-v 标志，防止孤立卷。

以下是一个较早的删除容器的具体示例：

```
docker rm -v student
```

另外，如果使用的是兼容 POSIX 的 shell，你可以用以下命令，删除所有停止的容器和卷：

```
docker rm -v $(docker ps -aq)
```

卷的清理是资源管理的重要组成部分。现在你已经了解了卷的生命周期、共享机制和用例，应该准备好学习高级的存储卷模式了。

4.5 存储卷的高级容器模式

在现实世界中，存储卷可用来实现多种文件系统的定制和容器的交互。本节主要介绍了几种高级但常见的存储卷模式，你可能会遇到或有充分的理由在你自己的系统中采纳这些模式。

4.5.1 卷容器模式

4.1.3 和 4.3.2 节中使用一种称为卷容器的模式，这是一个容器，只是提供卷的句柄。如果你遇到了一个用于与多个容器共享一组卷的情况，或者如果需要对一组适合常见用例的存储卷进行分类，那么这将很有用，如图 4-8 所示。

卷容器并不需要运行，因为停止时容器仍能保证存储卷的引用。最近的几个示例，都是使用卷容器模式。这个示例容器 cass-shared、fowler、knuth、chomsky 和 lamport 都运行了一个简单的 echo 命令来打印一些东西到终端，然后退出。当创建新容器时，你会使用

已停止的容器作为--volumes-from 标志来源。

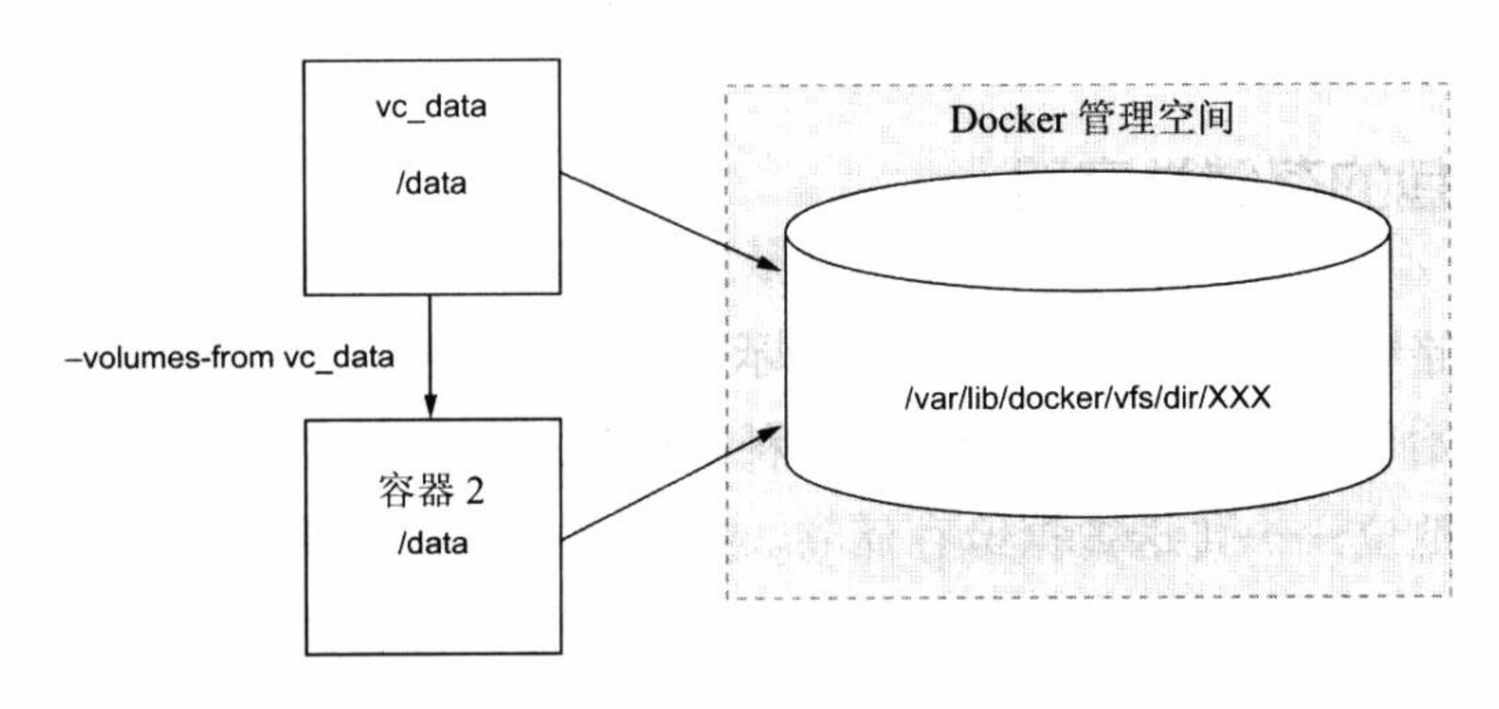


图 4-8 容器 2 复制了 vc_data 的存储卷句柄

卷容器对于数据句柄的维护非常重要，即使单个容器对数据具有一定的独占访问权限，这些句柄仍使其轻松地备份、恢复和迁移数据。

假设你想更新数据库软件（使用新的镜像）。如果你的数据库容器将其状态写入到存储卷，该卷是由一个卷容器定义的，迁移时将关闭原来的数据库容器，并启动新的卷容器作为新数据库容器的存储卷来源，很简单。备份和恢复操作可以类似地处理。当然，假定新数据库软件能够读取旧软件的存储格式，它只要在相同位置上寻找数据即可。

提示 使用容器名前缀（如 vc_）将是人或脚本在删除容器时不使用-v 选项的一个很好的提示。但特定的前缀没有那些建立的约定来得正式和重要，因为你的团队和构建的工具会依赖于这些约定。

当你控制并能够使挂载点的命名惯例标准化，那么存储卷容器将会最有用。这是因为每一个容器从卷容器复制卷并继承了挂载点的定义。 例如，一个卷容器定义了挂载在/logs 的存储卷，只对能够访问该存储卷（仅/logs 挂载点）的容器有效。以这种方式，一个存储卷和它的挂载点成为容器之间的一种合约。出于这个原因，具有特定要求的镜像应清楚地传达其文档中的内容或找到一种可编程的方法。

例如，卷容器挂载在/log，但新容器使用--volumes-from 标签，期望在/var/logs 目录发现日志。在这种情况下，新容器将无法访问它所需要的日志，并且该系统失效。

考虑另一个例子，名为 vc_data 的卷容器贡献了两个存储卷：/data 和/app。一个新容器依赖于 vc_data 提供的/data，却使用了/app，如果都以这种方式来复制这两个存储卷就会失败。因为这两个容器是不相关的，但 Docker 无法确定。错误无法发现，直到新的容器被创建，并以某种方式失败后才会觉察到。

存储卷容器模式，更简单。它是 Docker 处理数据的基础工具，并且可以以多种有趣的方式来扩展。

4.5.2 数据打包的存储卷容器

通过扩展存储卷容器模式，如图 4-9 所示，将容器中的数据打包以此增加其价值。一旦你调整容器开始使用存储卷，你会发现各种各样的场合都需要共享存储卷。存储卷容器处于一个独特的地位——传送数据给存储卷。数据打包的存储卷容器这种扩展模式因此名副其实。它描述了镜像是如何被用于分发静态资源，比如在由其他镜像构建的容器中使用的配置或代码。

数据打包的卷容器将镜像中的静态内容复制到其定义的存储卷。这些容器可用于分发关键架构信息，如配置，密钥材料和代码。

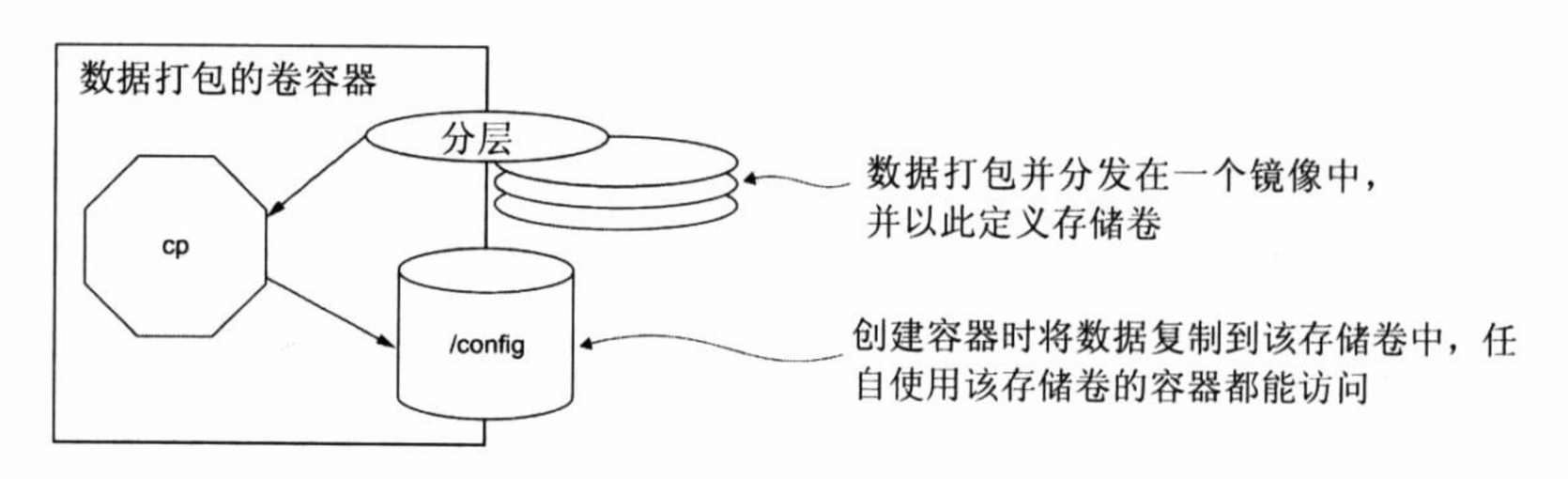


图 4-9 数据打包的存储卷容器，在/config 挂载点，共享和填充了一个存储卷

对于你手动构建的镜像中包含的可用数据，你需要复制出来，可通过运行和定义存储卷，并在容器启动时执行 cp 命令：

复制镜像内容到一个存储卷

```
docker run --name dpvc \
    -v /config \
    dockerinaction/ch4_packed /bin/sh -c 'cp /packed/* /config/'

docker run --rm --volumes-from dpvc \
    alpine:latest ls /config

docker run --rm --volumes-from dpvc \
    alpine:latest cat /config/packedData

docker rm -v dpvc
```

列出共享资料

查看共享资料

清理时记得使用-v参数

此代码中的命令共享了由单个镜像分发的文件。此例你创建了三个容器：一个数据打包的卷容器和两个容器，复制该卷并检查了卷内容。同样也说明了，你需要考虑配置分发的方法。利用数据打包的卷容器，将共享内容注入新容器，将是下一节讨论多态容器模式

的基础。

4.5.3 多态容器模式

正如我在本章前面提到的，多态工具就是以一致性的方式进行交互，但可能有几个实现，分别做不同的事情。使用存储卷，你可以注入不同的行为到容器中，而无需修改其镜像。一个多态容器提供了一些功能，很容易通过存储卷实现。例如，你可能有一个包含 Node.js 的二进制镜像，并默认执行运行 Node.js 位于/app/app.js 的程序。该镜像可能包含一些默认的实现，例如，简单打印“This is a Node.js application”到终端。

你可以通过挂载在/app/app.js 的存储卷，注入自己的 app.js 实现，改变该镜像构建的容器行为。在新镜像中分层添加新功能可能更有意义，但在某些情况下，这是最好的解决方案：第一个是在开发期间，可能不想在每次迭代时构建新镜像；第二个是在运维事件发生期间。

考虑一种场景，操作故障发生了。为了分类问题，需要镜像中某些可用的工具，可是在制作镜像时你没有预料到。但是，如果你挂载一个存储卷，其中放了这些工具，就可以使用 docker exec 命令运行容器中的其他进程：

```
docker run --name tools dockerinaction/ch4_tools

docker run --rm \
    --volumes-from tools \
    alpine:latest \
    ls /operations/*

docker run -d --name important_application \
    --volumes-from tools \
    dockerinaction/ch4_ia

docker exec important_application /operations/tools/someTool

docker rm -vf important_application

docker rm -v tools
```

创建一个存储容器，包含相应的工具

列出这些共享的工具

利用这些新工具，启动一个新的容器

在该容器中使用这些共享工具

关闭该工具

清理这些工具

你可以将文件注入其他静态容器来更改所有类型的行为。最常见的是，你将使用多态容器来注入应用程序配置。考虑一个多状态部署管道，其中应用程序的配置将根据部署它的位置而改变。你可以使用数据压缩卷容器，在每个阶段提供特定于环境的配置，然后应用程序将在某个已知位置查找其配置：

```
docker run --name devConfig \
    -v /config \
```

```
    dockerinaction/ch4_packed_config:latest \
    /bin/sh -c 'cp /development/* /config/'

docker run --name prodConfig \
    -v /config \
    dockerinaction/ch4_packed_config:latest \
    /bin/sh -c 'cp /production/* /config/'

docker run --name devApp \
    --volumes-from devConfig \
    dockerinaction/ch4_polyapp

docker run --name prodApp \
    --volumes-from prodConfig \
    dockerinaction/ch4_polyapp
```

在这个示例中，你启动同一应用程序两次，但采用不同的配置文件。使用这种模式，你可以建立一个简单的版本控制的配置分发系统。

4.6 小结

学习如何使用 Docker 的第一个重大障碍是理解存储卷和文件系统。本章做了深入介绍，其中包括：

- 存储卷允许容器与主机或其他容器共享文件。
- 存储卷是主机文件系统的一部分，Docker 将主机文件系统挂载到容器中指定位置。
- 两种类型的存储卷：Docker 管理卷挂载主机文件系统的 Docker 目录，绑定挂载卷可挂载主机文件系统的任何位置。
- 存储卷有生命周期，且独立于任何特定的容器，但是一个用户只能通过容器句柄引用 Docker 管理卷。
- 孤立卷问题会导致磁盘空间难以恢复。可在 `docker rm` 命令中使用`-v` 选项来避免此问题。
- 卷容器模式对保证存储卷有序组织，并避免孤立卷问题非常有用。
- 数据打包的卷容器模式对给其他容器分发静态内容非常有用。
- 多态容器模式是一种组成最小功能组件并最大化重用的方法。

第 5 章　网络访问

本章介绍

- 网络容器原型
- Docker 如何与计算机的网络一同工作
- Docker 如何构建网络容器
- 如何自定义容器网络
- 如何使容器对网络可见
- 发现网络上的其他容器

在上一章中，我们学习了如何在一个容器中使用数据卷和访问其中的文件。本章将探讨输入与输出的另一种常见形式：网络访问。

如果你想要在一个 Docker 容器中运行一个网站、一个数据库、一个邮件服务器或者任何依赖于网络的软件，比如一个 web 浏览器，你就必须了解如何将这个容器连接到网络。阅读完本章后，你能学会创建一个暴露在网络上且适合你所运行的应用的容器，能学会在一个容器中使用另外一个容器的网络软件，还能理解容器是如何与主机还有主机网络进行交互的。

本章关注于单主机网络（single-host networking）。多主机网络（multi-host networking）是第 12 章的主题。第 12 章描述了服务发现的策略和容器链接在那种情况下起到的作用。在你学习任何以上内容之前，本章的内容是必需学习且有意义的。

5.1 网络相关的背景知识

简要地概括下相关的网络概念对理解本章的主题非常有帮助。注意，本节仅仅涉及高层的概念；因此，如果你是这方面的专家，请随意跳过这部分内容。

网络都是关于进程间通信的内容，这些进程可能会，也可能不会共享相同的本地资源。为了理解本章的主要内容，你只需要掌握一小部分基础的、常被进程使用的网络抽象概念。你对网络理解得越深，就能对 Docker 的工作原理学得越好。但是深入的理解对于使用 Docker 提供的工具不是必须的。即便这种情况发生了，本节包含的内容也能够帮助你独立地调研相关的主题。这些被用于进程的基础抽象概念包括协议、网络接口和端口。

5.1.1 基础：协议，接口和端口

就通信和网络而言，协议就是一类语言。支持同一协议的双方就能够理解对方传递的信息，这是有效通信的关键所在。Hypertext Transfer Protocol（HTTP）是一个非常流行的网络协议，很多人都听说过它，正是这个协议，构建了万维网（World Wide Web）。大量的网络协议和多个通信层都是由那些基础协议创建的。不过，就目前来说，更重要的是你知道了协议是什么，接下来你就能够理解网络接口和端口了。

网络接口拥有一个地址，代表了一个位置，你可以把接口类比成真实世界中带有地址的位置。网络接口就像一个信箱，发送给收件人的信息会被投递到带有收件人接口地址的信箱中，同时信息也会从该地址的信箱中取出，投递到其他地址的信箱。

类似于一个信箱拥有一个邮政地址，一个网络接口则拥有一个 IP 地址，IP 地址由互联网协议（Internet Protocl）定义的。关于 IP 协议的细节内容非常的有趣，但是超出了本书的范围。更重要的是，你要明白 IP 地址在网络中是独一无二的，并且包含了它们在网络中位置的信息。

通常来说，计算机拥有两种类型的接口：一类是以太网接口，另一类是本地回环接口。以太网接口就是你最熟悉的那类网络接口，它用来连接其他的接口和进程。本地回环接口则不会连接到其他任何接口。乍看起来，这类接口好像毫无用处，但是利用网络协议来与同一台计算机上其他程序进行通信时，本地回环接口通常是非常有用的。

类似于前面信箱的隐喻，一个端口就像是一个收件人或发件人。在一个地址上可能会有多个收件人。类似地，一个 IP 地址可以接收分别发向 Wendy web 服务器、Deborah 数据

库和 Casey 缓存的信息，如图 5-1 所示阐明了这个过程。每个接收信息的人应当只能看到自己的信息。

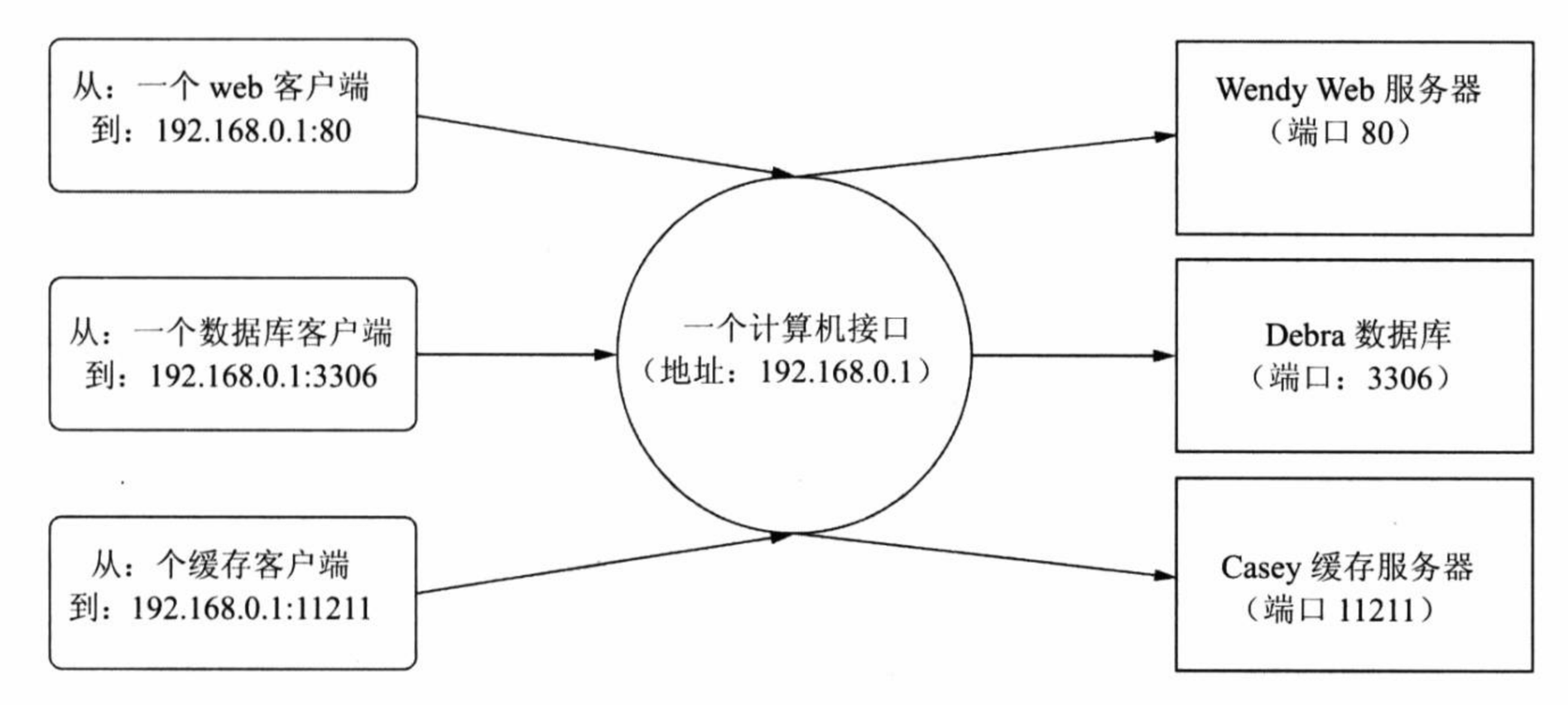


图 5-1 多个进程使用了同一个网络接口，类似于多个使用同一个信箱的人能够被区分，这些进程也能够被唯一标识

实际上，端口仅仅是数字，是 Transmission Control Protocol（TCP）协议的一部分。协议的详细内容同样超出了本书的范围，但是我鼓励你去学习这些内容。创建了某个协议标准的人，或者是拥有某一个具体产品的公司，拥有端口的决定权。比如说，web 服务器提供的 HTTP 服务，端口默认为 80。MySQL，一个数据库产品，协议的默认端口是 3306。Memcached，是一种高速缓存技术，默认在 11211 端口上提供服务。记录在 TCP 上的端口信息就相当于写在信封上的名字，能够用来区分收件人。

协议、接口和端口对于软件和使用者来说是首先要关心的内容。通过学习这些知识，你能够更好地理解程序间通信的方式和更好地理解你的计算机是如何接入到更大的网络中的。

5.1.2 高级：网络，NAT 和端口转发

每个接口代表了网络中一个单一节点。接口连接在一起就定义了网络，并且这个连接决定了接口的 IP 地址。

有的时候，信息的接收接口并不与发送接口直接相连，这时信息会被传输到一个中间节点，这个节点知道如何将信息传输到最终目的接口。现在，让我们再回头来看信箱这个

隐喻，我们会发现以上过程和真实世界邮局的操作过程非常相似。

当你在发件箱放置了一个信息，负责发送信息的程序就会取出这个信息，并且将它发送到一个本地路由设备。这个设备本身就是一个接口，它会接收这个信息，然后沿着路由的下一个节点传输到接收接口。一个本地路由设备可能会将信息转发到一个区域性网络路由设备，经过该区域性网络路由设备再传输到接收接口的本地路由设备，最终才到达接收接口。在网络路由中这类传输模式是非常普遍的。如图 5-2 所示了以上描述的路由过程，并且画出了现实世界信件投递和网络路由之间的关系。

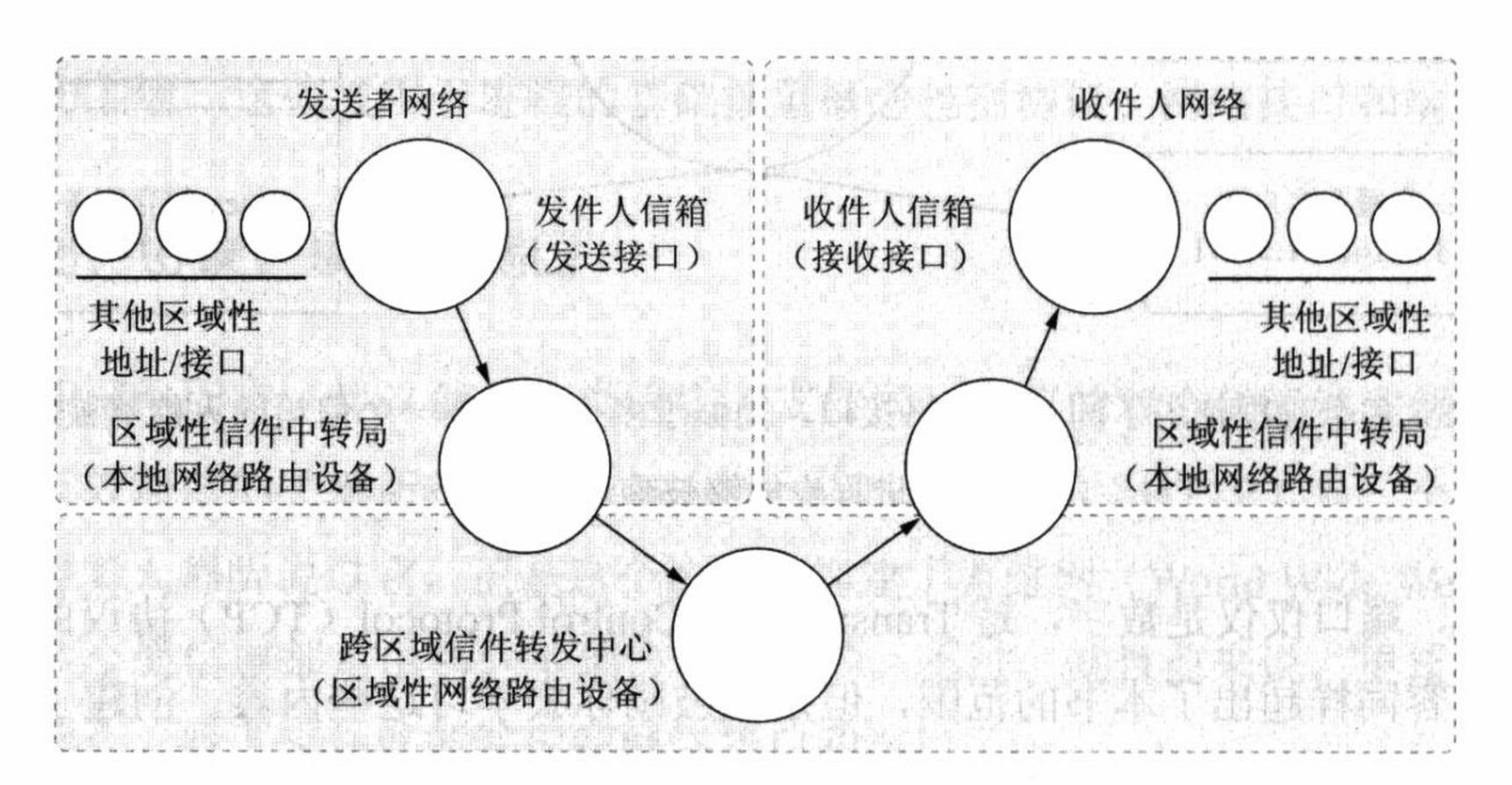


图 5-2 信息在邮政系统和计算机网络中的传输路径

本章只关注单台计算机上的接口，因此我们考虑的网络和路由并不会像上面的那么复杂。实际上，本章讲述的是两个特殊的网络和容器连接到这两个网络的方式。第一个网络就是你的计算机现在连接的网络。第二个是 Docker 创建的一种虚拟网络，创建它的目的是让所有正在运行的容器能够连接到第一个网络，这种虚拟网络被称为网桥。

如同名字所暗示的，一个网桥就是一个能够连接多个网络的接口，它使得多个网路就能像单个网络一样工作，如图 5-3 所示，其形象地描绘了这种网络。连接在网桥上的网络基于不同类型的网络地址，网桥通过选择性地转发这些网络之间的流量来工作。也许这有些难以理解，没有关系，你只需要去适应这个抽象的概念，就足以理解本章的主要内容了。

以上仅仅是对一些具有细微差别的概念的简要介绍。事实上，我也只是简要地介绍了这些概念，目的就是帮助你去理解如何使用 Docker 和被它简化后的网络设备。

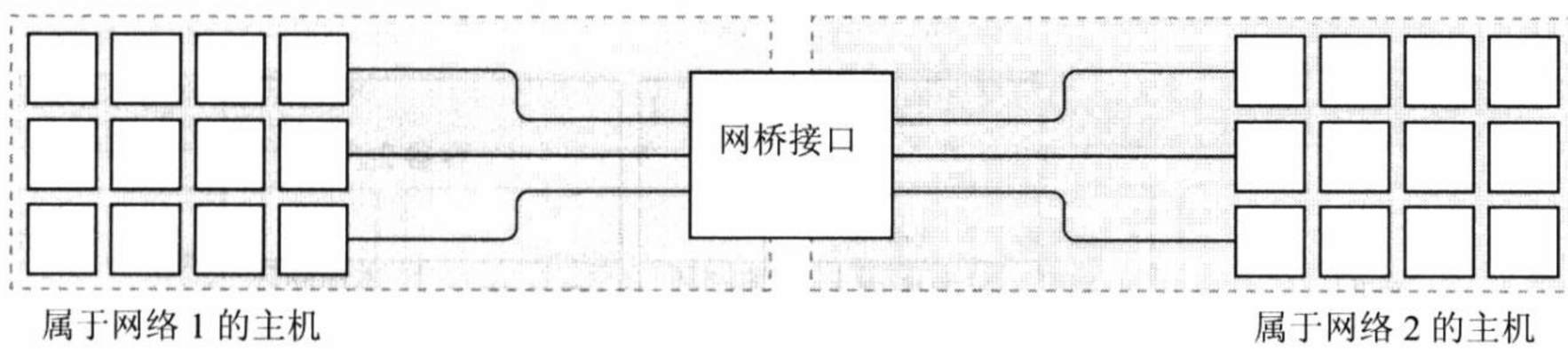


图 5-3 连接着两个不同网络的网桥接口

5.2 Docker 的网络

Docker 关心两种类型的网络：单主机虚拟网络和多主机虚拟网络。本地虚拟网络用来提供容器的隔离。多主机虚拟网络构建了一个抽象的覆盖网络，在这个网络中，任何容器相对于网络上的其他容器都拥有独立的、可路由的 IP 地址。

本章深入讨论了单主机虚拟网络。理解 Docker 如何在网络上将容器隔离，对于那些关心安全的人是非常重要的。同时，那些构建网络应用的人需要知道容器化将会如何影响他们的部署要求。

在我写本章时，多主机网络依旧处于测试状态。实现它不仅需要理解单主机网络的那部分内容，还需要对其他的生态系统工具有深刻的理解。因此，在多主机网络功能被开发出来前，将理解 Docker 如何构建本地虚拟网络作为学习的起点是非常正确的。

5.2.1 本地 Docker 网络的拓扑结构

Docker 使用操作系统的底层特性构建了一个特殊的、可定制的虚拟网络拓扑结构。这个虚拟网络只在安装有 Docker 的机器上有效，并且它由主机上的容器和主机所连接的网络之间的路由构成。你可以改变这个网络结构的行为，甚至在某些情况下，可以使用启动 Docker 后台进程和容器的命令行的选项来改变网络结构本身。如图 5-4 所示描绘了两个连接到虚拟网络和对应组件的容器。

每个容器各自拥有一个本地回环接口和一个分离的以太网接口，其中以太网接口连接着在主机命名空间上的另一个虚拟接口。这两个互连的接口在主机网络栈和每个容器的网络栈之间建立了连接。就像经典的家庭网络，每一个容器都被赋予了一个唯一的私有 IP 地址，从外部的网络不能直接连接到该私有 IP。网络连接需要经过 Docker 网桥接口路由到另一网络，这个网桥接口被称为 docker 0。你可以把 docker 0 想象成家庭中的路由器。为每个容器创建的虚拟接口都会连接到 docker 0，这样它们就构成了一个网络。最后，这个网桥接口 docker 0 会连接到主机所连接的网络上。

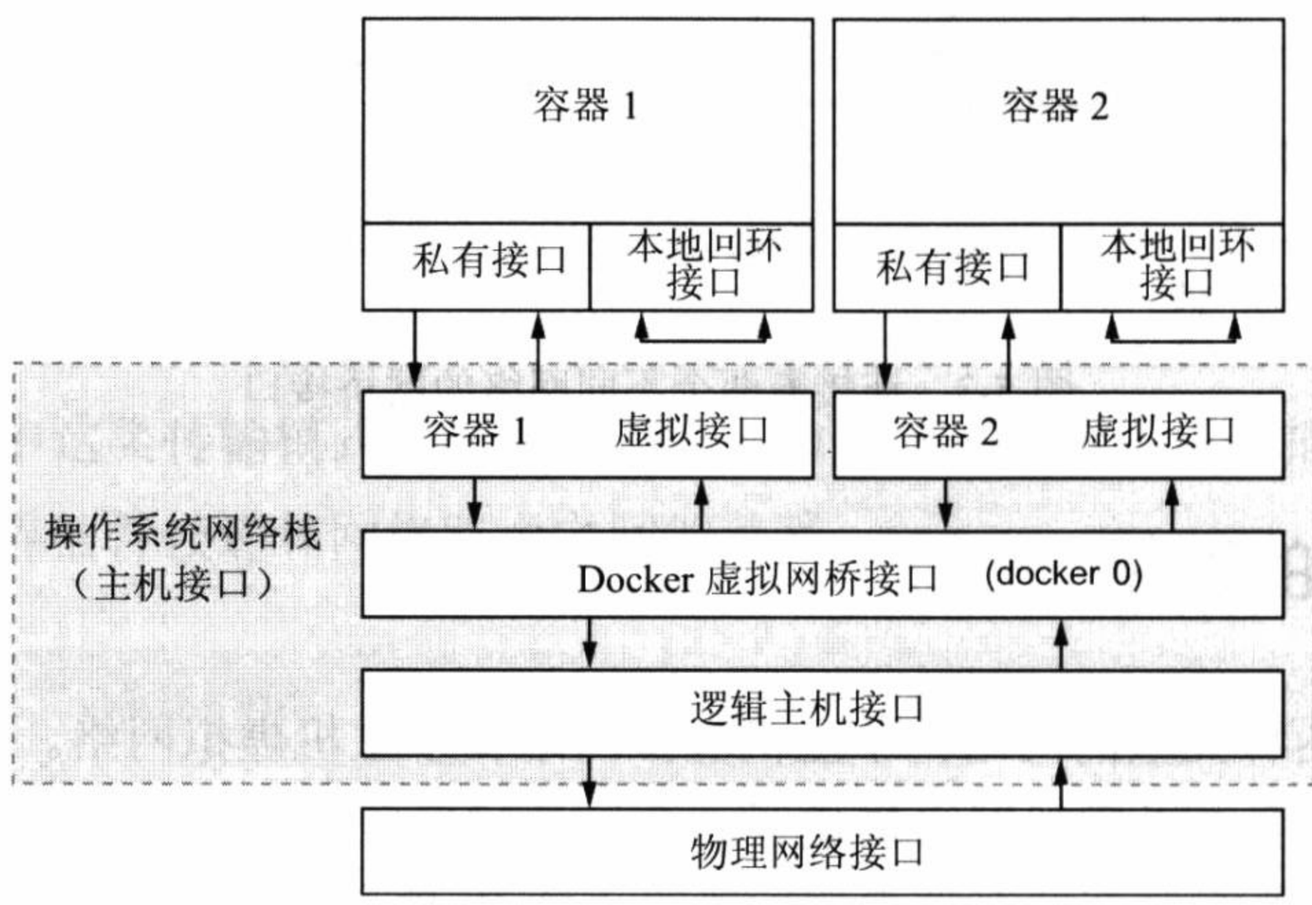


图 5-4 默认的本地 Docker 网络拓扑结构和两个连接在网络上的容器

使用 docker 命令行工具，你可以自定义 IP 地址、网桥接口 docker 0 连接的主机接口、容器之间通信的方式。接口之间的连接决定了容器如何隔离或者暴露在网络中。Docker 使用内核命名空间来创建这些私有的虚拟接口，但是命名空间本身并不提供网络的隔离。网络暴露或者隔离是通过主机的防火墙规则（每一个主流的 Linux 发行版都运行有一个防火墙）来实现的。Docker 的命令行选项提供了四种网络容器原型。

5.2.2 四种网络容器原型

所有的 Docker 容器都要符合这四种原型中的一种。这些原型定义了一个容器如何与其他的本地容器、主机网络进行通信。每一种原型有不同的目的，你可以认为它们拥有不同程度的隔离。当使用 Docker 创建容器时，仔细思考你的目标是非常重要的，然后在不影响目标情况下，尽可能地使用最强力的容器。如图 5-5 所示，其形象地描绘了每一个原型，最强大（也意味着隔离程度最高）的在最左边，最脆弱的在最右边。

四种原型如下：

- Closed 容器
- Joined 容器
- Bridged 容器
- Open 容器

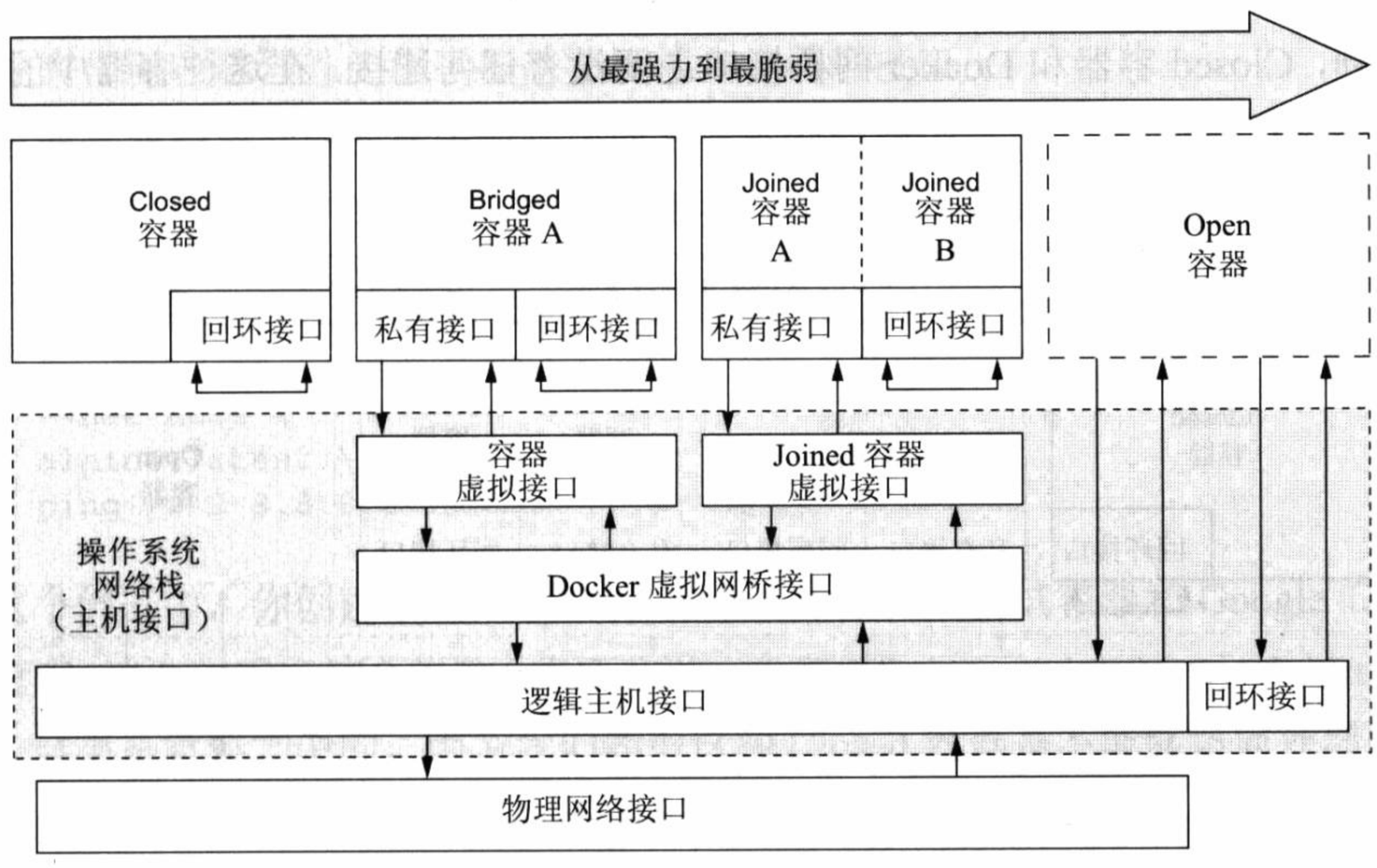


图 5-5　四种网络容器原理以及它们与 Docekr 网络拓扑结构的关系

在接下来的内容，我将会逐一介绍四种网络容器原型。需要使用到全部原型的读者只占很小一部分，因此在学习如何构建、何时使用哪一种原型的内容后，你就有能力区分这四种原型，选择你应该使用的一种来构建容器。

5.3 Closed 容器

最强大的网络容器意味着不允许任何的网络流量。这一种被称为 Closed 容器。运行在这一种容器中的进程只能够访问本地回环接口。如果进程只需要和本身或者和其他本地进程通信的话，选择这一种是非常合适的。但是，如果容器中有任何进程想访问这种容器不支持的网络时，比如说软件想从互联网下载更新，因为进程不能访问互联网，因此使用这种原型就是不合适的。

大多数的读者从事的领域主要在服务器软件或者是 web 应用后端，在这种情形下，很难想象不能访问网络的容器会有什么实际的应用。现在有很多的方式能够让使用者很轻松地使用 Docker，而不用考虑数据卷容器、备份工作、脱机批处理、诊断工具的存在。你面临的挑战并不是证明 Docker 每一个功能存在的必要性，而是了解哪一个功能最适合你的用户场景。

Docker 在创建这一种容器时，跳过了创建外部可访问网络接口的步骤。可以在如图 5-6 所示中看到，Closed 容器和 Docker 网桥接口之间没有任何连接。在这种容器中的程序只能和本地程序进行通信。

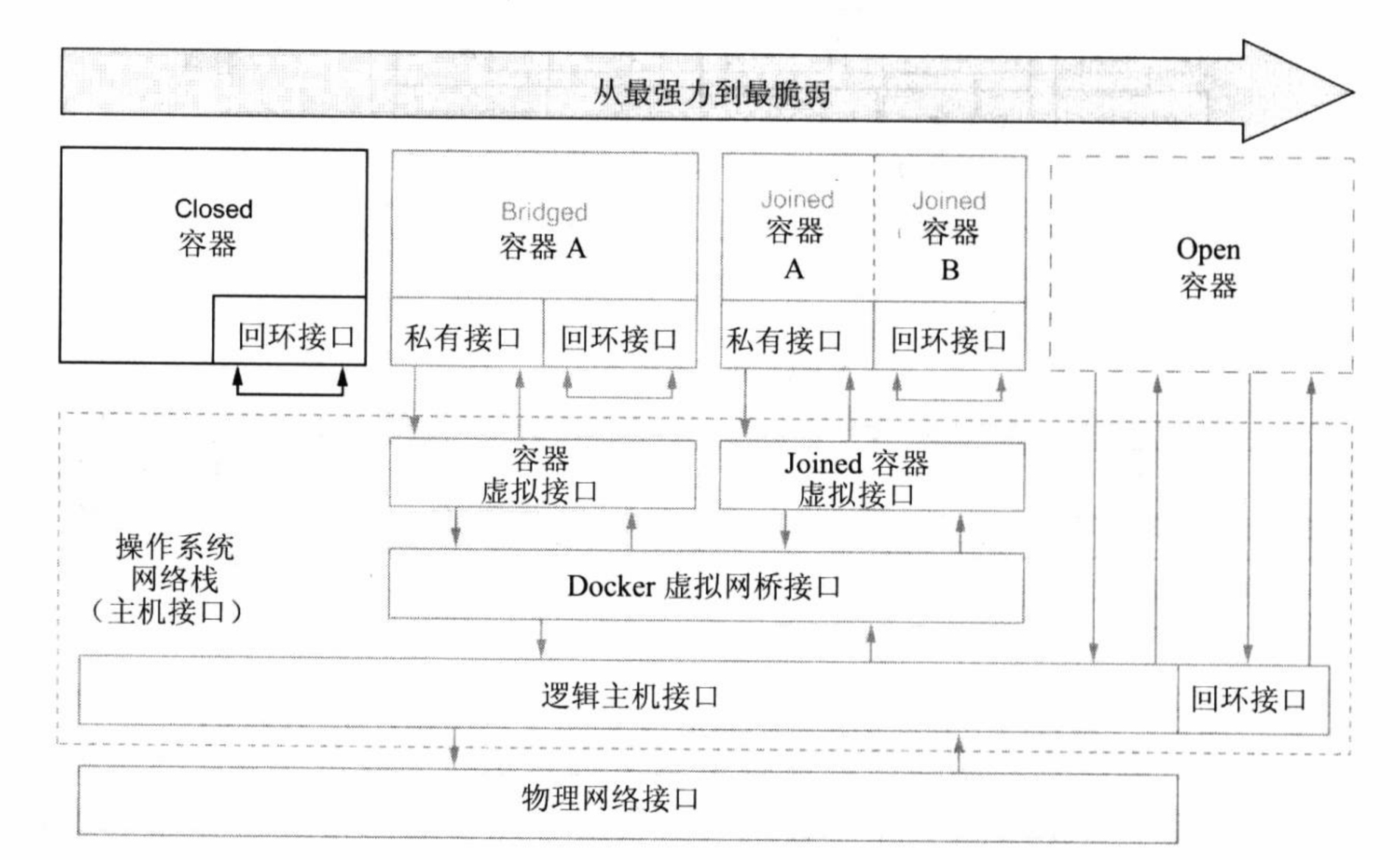


图 5-6 Closed 容器以及相关的组件

所有的 Docker 容器，包括 Closed 容器，都有权限访问一个私有的本地回环接口。你可能已经有过访问回环接口的经验。对于开发者来说，使用 localhost 或者 127.0.0.1 作为 URL 中的地址是非常常见的。在这些情况下，你可以让程序去绑定本地回环接口或者去连接已经绑定在该接口的服务。

Docker 为每一个容器创建私有本地回环接口，目的是让运行在容器中的程序能够通过网络进行通信，注意这些通信不能离开容器。

你可以在 `docker run` 命令后添加`--net none` 作为参数来告诉 Docker 创建一个 Closed 容器:

```
docker run --rm \
    --net none \
    alpine:latest \
    ip addr
```

创建一个 Closed 容器

列出所有的接口

运行以上的例子，你可以看到本地回环接口是唯一可用的网络接口，并且绑定到 IP 地址 127.0.0.1 上。以上的配置意味着三件事：

- 任何运行在该容器中的程序可以等待该接口上的网络连接或向该接口发起网络连接。
- 没有任何在容器外的程序都能够连接到该网络接口上。
- 没有任何在容器内的程序能够访问到容器外的网络。

最后一点非常重要，并且非常容易被证明。比如说，尝试访问一些总是有效的网络服务，这里，我们尝试访问 Google 的公共 DNS 服务：

```
docker run --rm \
    --net none \
    alpine:latest \
    ping -w 2 8.8.8.8
```

创建一个 Closed 容器

尝试访问谷歌 DNS 服务器

在这个例子中，你创建了一个 Closed 容器，并且尝试测试容器和 Google DNS 服务器的连接速度。这个尝试应该会失败，并且返回一个类似“ping:send-to:Network is unreachable”的信息。结果是意料之中的，因为我们知道这种 Closed 容器是不能够访问外部网络的。

什么时候使用 Closed 容器

如果对网络隔离程度要求非常高，或者程序不需要网络访问权限时，Closed 容器是最好的选择。比如说，运行一个终端文本编辑器不需要外部网络连接；为了防止信息被窃取，生成随机密码的程序也不需要外部网络连接。

对于一个 Closed 容器，并没有太多的方式来自定义网络配置。尽管这种容器看起来被限制过度了，但是它是四种原型中最安全的，并且也能够被扩展来适应特殊情况。这并不是 Docker 容器的默认选项，但是在实际生产环境中，在使用其他更脆弱的选项前，你必须找到这么做的充分理由。Bridged 容器才是 Docker 的默认选项，下面我们将介绍这个默认选项。

5.4 Bridged 容器

Bridged 容器放开了网络的隔离程度，因此这种容器入手更加容易。这种原型可定制性最高，并且被认为是最佳实践。Bridged 容器拥有两个接口，一个是私有的本地回环接口，另外一个私有接口通过网桥连接到主机的其他容器。

本节是本章中最长的部分。Bridged 容器是最常见的网络容器原型（如图 5-7 所示），本节介绍了几个其他原型也能使用的新选项。在 5.6 节之前，我们讨论的内容都跟 Bridged 容器有关。

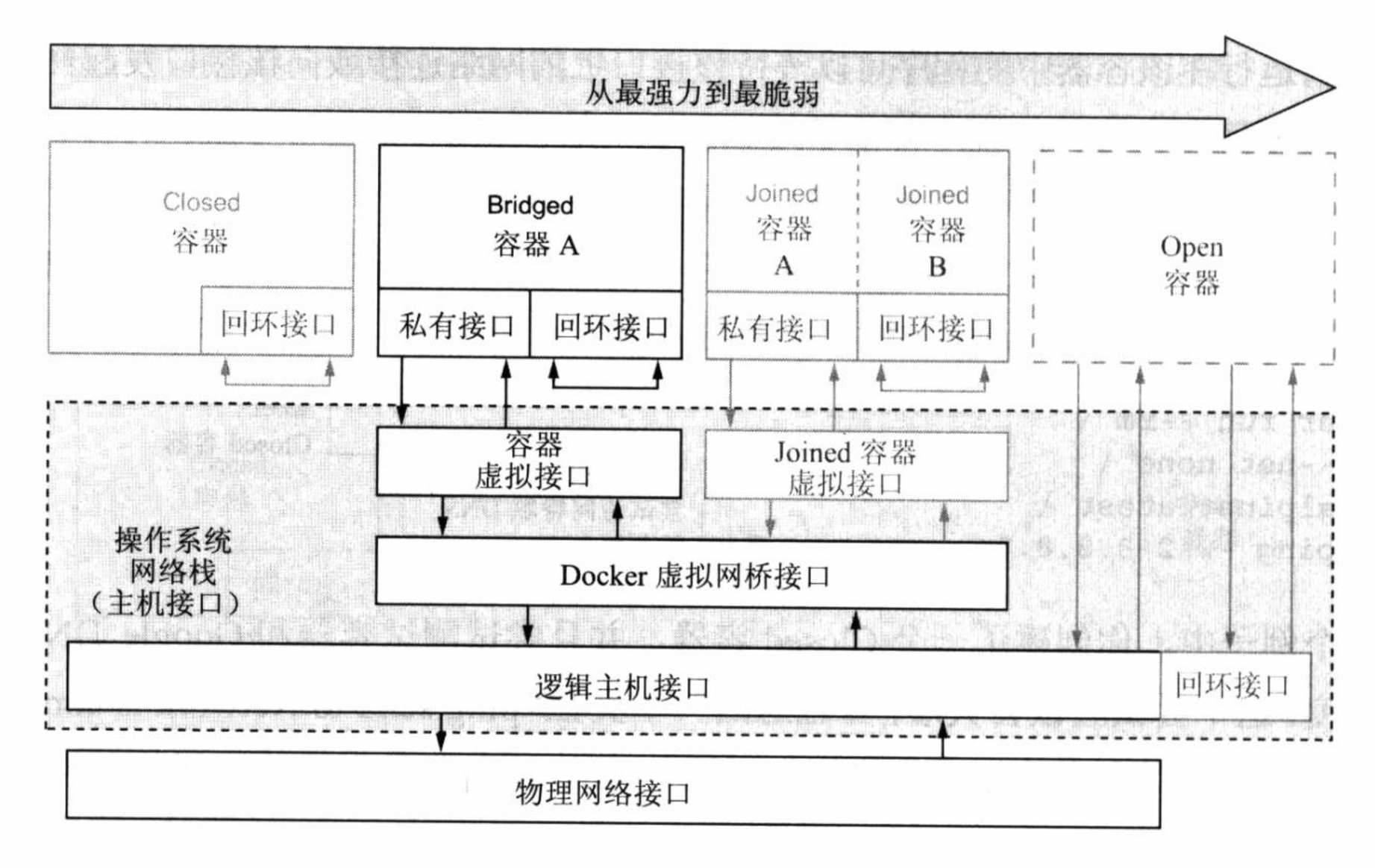


图 5-7 Bridged 容器和相关组件

所有连接到 docker 0 接口的接口都是同一个虚拟子网的一部分。这意味着它们能够互相通信，并且能通过 docker 0 接口和外部的网络进行通信。

5.4.1 访问外部网络

选择 Bridged 容器的最常见理由就是进程需要访问外部网络。为了创建一个 Bridge 容器，你可以忽略 `docker run` 命令中的`--net` 选项，或者将`--net` 的值设置为 `bridge`。下面的例子各自使用了一种方式来创建 Brided 容器：

```
docker run --rm \
    --net bridge \
    alpine:latest \
    ip addr
```

加入网桥网络

列出所有的网络接口

就像 Closed 容器中的第一个例子，这个命令会从最新的 alpine 镜像中创建一个新的容器并且列出所有可用的网络接口。这一次，它将会列出两个接口：一个以太网接口和一个本地回环接口。输出结果包括 IP 地址、子网掩码、最大传输单元（MTU）和不同接口的权值。

你已经验证了该容器拥有另外一个带有IP地址的接口，现在再一次尝试访问外部网络。这一次，我们忽略`--net` 选项来验证 Bridge 容器是否是 Docker 网络容器的默认选项：

```
docker run --rm \
    alpine:latest \
    ping -w 2 8.8.8.8
```

忽略了--net 选项

尝试访问谷歌 DNS 服务器

从 Bridged 容器中尝试访问 Google 的公开 DNS 服务器可以工作，并且不需要设置额外的选项。运行这个命令后，你将会看到你的容器执行了 2 秒的 ping 测试，并且输出了相关网络统计信息的报告。

现在，你应该知道了，如果你有一些软件需要访问外部网络，或者访问私有网络中的其他计算机，那么你可以使用 Bridged 容器。

5.4.2 自定义命名解析

域名系统（DNS）是一种能够将主机名映射成 IP 地址的协议。使用这种协议，客户端就能从某一个 IP 地址的依赖中解耦出来，转而依赖于一个固定的主机名，不管主机的 IP 地址如何改变，主机名都会负责映射到这个 IP。一个改变对外通信的最基础方式就是为 IP 地址创建名字。

网桥网络上的容器和其他在该网络上的计算机，拥有不具备公共可路由能力的 IP 地址是非常典型的。这意味着除非你运行有自己的 DNS 服务器，否则你不能够通过名字来映射它们。Docker 提供了不同的选项来自定义 DNS 配置。

首先，`docker run` 命令有一个`--hostname` 选项，你可以使用这个选项来设置一个新容器的主机名。这个选项会在该容器中的 DNS 覆盖系统中添加一条记录。这条记录会将提供的主机名映射成该容器的桥接 IP 地址。

```
docker run --rm \
    --hostname barker \
    alpine:latest \
    nslookup barker
```

设置容器的主机名

将主机名解析为 IP 地址

这个例子创建了一个主机名为 bakker 的容器，并且执行了一个程序来查找该主机名对应的 IP 地址。运行这个例子将会生成类似以下格式的信息：

```
Server:    10.0.2.3
Address 1: 10.0.2.3

Name:      barker
Address 1: 172.17.0.22 barker
```

最后一行的 IP 地址就是上面新创建的容器的桥接 IP 地址。第一行带有 Server 标签的

IP 地址表示的是提供映射功能的服务器地址。

为容器设置主机名字是非常有用的，比如说容器中的程序需要查询它自己的 IP 地址或者必须自我识别时。因为其他的容器不知道这个主机名，因此它的功能是有限的。但是如果你使用了一个外部 DNS 服务器，你就能够共享这些主机名了。

第二个自定义 DNS 配置的选项能够用来指定一个或者多个 DNS 服务器。为了证明它的作用，下面的例子创建了一个新的容器，并且将它的 DNS 服务器设置为 Google 的公开 DNS 服务器：

```
docker run --rm \
    --dns 8.8.8.8 \
    alpine:latest \
    nslookup docker.com
```

设置主 DNS 服务器

解析 docher.com 的 IP 地址

如果你在笔记本上运行 Docker，并且经常在不同网络供应商之间移动，那么使用一个特定的 DNS 服务器能够提供一致性。对于构建服务和网络，这是一个非常重要的工具。下面有一些关于设置 DNS 服务器的重要笔记：

- 值必须是 IP 地址。原因非常明显，容器需要一个 DNS 服务器来查找某一个名字的 IP 地址。
- `--dns=[]`选项可以被使用多次来设置多个 DNS 服务器（防止一个或者多个服务器不可用）。
- `--dns=[]`选项可以在你启动后台进程 Docker daemon 时进行设置。这么做之后，这些 DNS 服务器会默认配置到每一个容器上。如果跟该后台程序相关的容器依旧运行，这时候你停止该后台程序并且修改默认的 DNS 服务器，当你重新启动该后台程序时，运行中的容器依旧会保留老的 DNS 服务器设置。你需要重新启动这些容器来让 DNS 服务器改动生效。

第三个 DNS 相关的选项——`--dns-search=[]`，允许你指定一个 DNS 查找域，这个查找域就像 host 名的一个默认后缀。当该选项被设置，在查询时，任何不包括已知顶级域名（比如.com 或者.net）的主机名都会自动加上该后缀名。

```
docker run --rm \
    --dns-search docker.com \
    busybox:latest \
    nslookup registry.hub
```

设置查找域

registry.hub.docker.com 解析的快捷方法

这个命令会解析 registry.hub.docker.com 的 IP 地址，因为指定的 DNS 查找域会自动补全主机名。

这个功能最常用来解决烦琐事情，比如说公司内部网络的快捷名称。你的公司可能维护一个内部的 wiki 文档，你能够简单地通过 http://wiki/来引用。但是 Docker 提供的这个功能可以有更多强大的应用。

假设你分别为你的开发和测试环境维护了一个 DNS 服务器。与其构建环境感知（environment-aware）有关的软件（硬编码，如 myservice.dev.mycompany.com），你可以考虑使用 DNS 查找域和无环境感知（environment-aware）的名字（如 myservice）：

```
docker run --rm \
    --dns-search dev.mycompany \
    busybox:latest \
    nslookup myservice
```

注意 dev 这个前缀

解析 myservice.dev.mycompany 的 IP 地址

```
docker run --rm \
    --dns-search test.mycompany \
    busybox:latest \
    nslookup myservice
```

注意 test 这个前缀

解析 myservice.test.mycompany 的 IP 地址

使用这种模式，唯一需要改变的就是程序运行的上下文。和提供自定义的 DNS 服务器类似，你可以为同一个容器提供多个自定义的查找域，你只需要简单地设置多次--dns-search 选项就行了。举个例子：

```
docker run --rm \
    --dns-search mycompany \
    --dns-search myothercompany ...
```

这个选项也可以在启动 Docker 后台进程时进行设置，为每一个新创建的容器提供默认的查找域。再一次提醒你，只有创建容器时，这些选项才会生效。如果一个容器正在运行，你改变了默认值，那么这个容器会保留旧的值。

最后一个 DNS 相关的选项提供了覆盖 DNS 系统的能力。这个系统和`--hostname` 选项中提到的系统是一样的。`--add-host=[]`选项能自定义从主机名到 IP 地址的映射关系：

```
docker run --rm \
    --add-host test:10.10.10.255 \
    alpine:latest \
    nslookup test
```

注意映射关系

解析到 10.10.10.225

类似于`--dns` 和`--dns-search` 选项，这个选项能够被设置多次。但是不同于其他的选项，这个选项不能够在启动 Docker 后台进程时设置默认值。

这个功能就像一把主机名解析手术刀。它能为单独的容器提供特定的主机名映射，可

能是最细粒度的自定义了。你可以使用这个功能将特定的主机名映射到一个已知的 IP 地址上，比如说 127.0.0.1，以此来有效地阻止特定的主机名。你也可以使用它来将指定目的 IP 的网络流量，通过代理进行路由。这经常被用来将不安全的网络流量转入到安全通道，比如 SSH，进行路由。这种添加覆盖的技巧已经被 web 开发者使用很多年了，他们经常使用这种方式来运行 web 应用的本地副本。如果你花费一定的时间来思考这些接口，我确信你一定能想到各种各样的应用情景。

所有的自定义转换关系都保存在容器中的/etc/hosts 文件中。如果你想要看看有哪些覆盖内容，你所要做的就是查找这个文件。这个文件的编辑和解析规则有些超出本书的范围，这里不再深入探讨，想要学习的话，这些内容都可以在网络上找到。

```
docker run --rm \
    --hostname mycontainer \
    --add-host docker.com:127.0.0.1 \
    --add-host test:10.10.10.2 \
    alpine:latest \
    cat /etc/hosts
```

设置 host 名

创建映射关系

创建另一个入口

列出所有的入口

图上的命令将会输出类似以下内容的信息：

```
172.17.0.45  mycontainer
127.0.0.1    localhost
::1          localhost ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
10.10.10.2   test
127.0.0.1    docker.com
```

DNS 是一个能够改变行为的强大系统。主机名到 IP 地址的解析提供了一个简单的接口，使得开发者和程序能够从某一个具体的网络地址中解耦出来。如果说 DNS 是你改变出站流量行为的最好工具，那么防火墙和网络拓扑结构就是你控制入站流量的最好工具。

5.4.3 开放对容器的访问

Bridged 容器在默认情况下不能够被主机网络访问。容器被主机的防火墙保护了起来。默认的网络拓扑结构没有提供任何从主机外部接口到容器接口的路由。这意味着想要从主机外部访问到容器是不可能的。入站流量的流动情况，如图 5-8 所示。

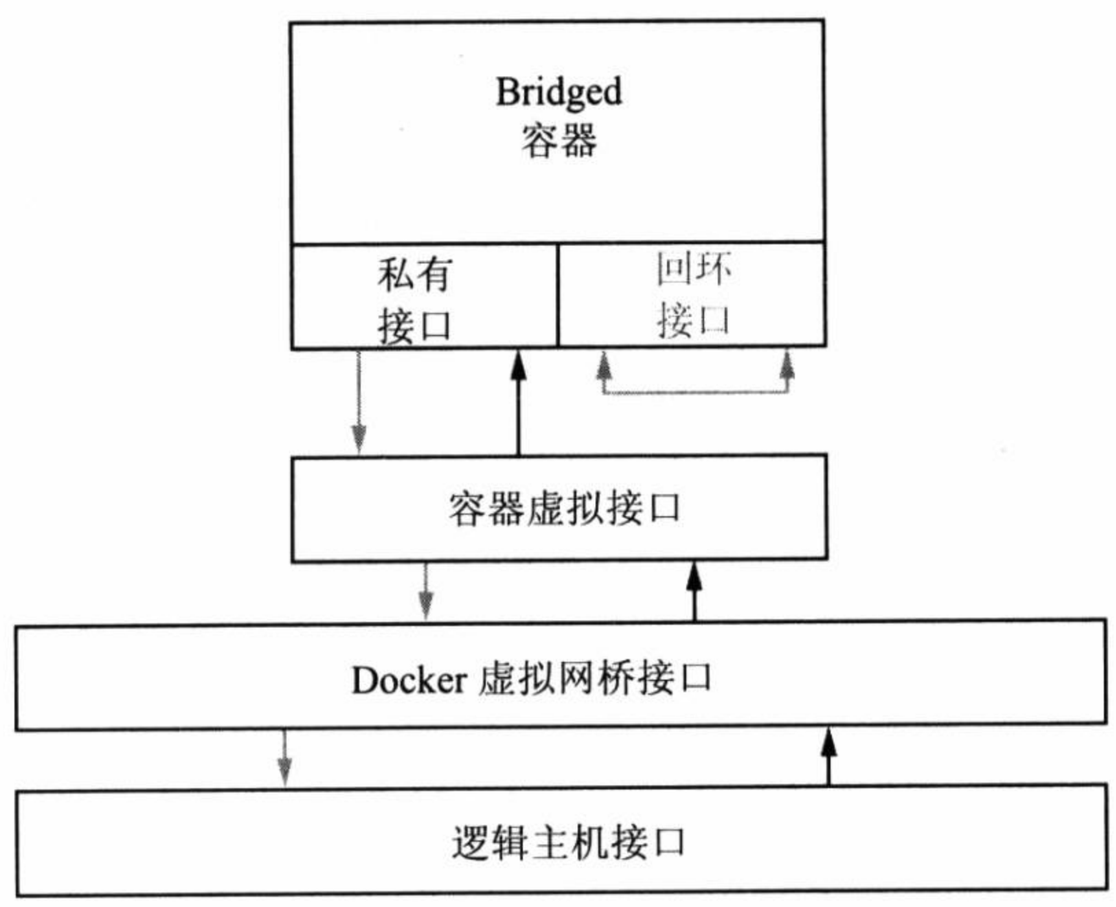


图 5-8 Bridged 容器的入站流量

如果不能够通过网络访问容器，那么容器的功能会变得非常有限。幸运的是，事实并非如此。docker run 命令提供了一个-p --publish=[]选项，它能够在主机网络栈上的端口和容器端口之间创建映射关系。你已经在前面多次使用过这个功能了，但是它值得我们再一次探讨。映射的格式有如下四种：

- 这种格式会在将容器端口绑定到所有主机接口的一个动态端口上。

```
docker run -p 3333 …
```

- 这种格式会将一个具体的容器端口绑定到每一个主机接口的某一个具体接口上。

```
docker run -p 3333:3333 …
```

- 这种格式会将容器端口绑定到拥有指定 IP 地址的主机接口的动态端口上。

```
docker run -p 192.168.0.32::2222 …
```

- 这种格式会将容器端口绑定到拥有指定 IP 地址的主机接口的一个具体的端口上。

```
docker run -p 192.168.0.32:1111:1111 …
```

以上的例子假设你的主机 IP 地址是 192.168.0.32。这个地址是随机的，但是用来说明这个功能是非常有用的。以上的每一个命令都会创建从主机接口的端口到容器接口的端口的路由。不同的格式提供了不同程度的粒度和控制。当你想要提供多个映射关系时，这个选项同样也能够被设置多次。

docker run 命令提供了另外一个可替代的方式来打开网路通道。如果你能够接受主

机上动态或者短暂的端口，那么你可以使用-P --publish-all 选项。这个选项会告诉 Docker daemon 去创建端口映射关系，效果类似于-p 命令的第一种格式作用于容器所有的端口，将容器的端口都暴露出去。镜像自带了一组开放端口，一方面为了简单实用，另一方面可以提醒使用者容器中的服务监听了哪些端口。举个例子来说，dockerinaction/ch5_expose 开放了 5000、6000 和 7000 端口，下面的每一个命令都能起到同样的效果：

```
docker run -d --name dawson \
    -p 5000 \
    -p 6000 \
    -p 7000 \
    dockerinaction/ch5_expose
```

暴露所有的端口

```
docker run -d --name woolery \
    -P \
    dockerinaction/ch5_expose
```

暴露相关的端口

你能很清楚地看到，以上的命令能够省去用户一部分的输入，但是它也带来了两个问题。第一，如果没有开放你想要的端口，怎么办？第二，如何发现哪一个动态端口被映射到容器端口？

docker run 命令提供了另外一个--expose 选项，它能设置容器想要开发的端口。这个选项能够被设置多次，一个端口设置一次：

```
docker run -d --name philbin \
    --expose 8000 \
    -P \
    dockerinaction/ch5_expose
```

暴露另一个端口

暴露所有的端口

以上命令中的--expose 选项会将 8000 端口添加到-P 选项的端口列表中。运行以上的例子后，你可以通过使用 docker ps、docker inspect 或者一个新的 docker port 命令来查看端口是如何被映射的。port 子命令接受一个容器名字或者 ID 作为参数，并且会输出一个列表，每一行对应一个端口映射：

```
docker port philbin
```

运行以上命令应该会产生类似以下格式的信息：

```
5000/tcp -> 0.0.0.0:49164
6000/tcp -> 0.0.0.0:49165
7000/tcp -> 0.0.0.0:49166
8000/tcp -> 0.0.0.0:49163
```

有了本节介绍的工具，你就能够将入站流量正确地路由到某一个 Bridged 容器上了。跨容器通信是另一种精妙的通信方式。下一节将介绍这个内容。

5.4.4 跨容器通信

提醒一下，到目前为止，所有的容器都是使用 Docker 桥接网络来与其他容器或者主机网络进行通信。所有的本地 Bridged 容器都是在同一个桥接网络上，并且默认情况能够互相通信。如图 5-9 所示同一个主机上，五个容器之间的网络关系。

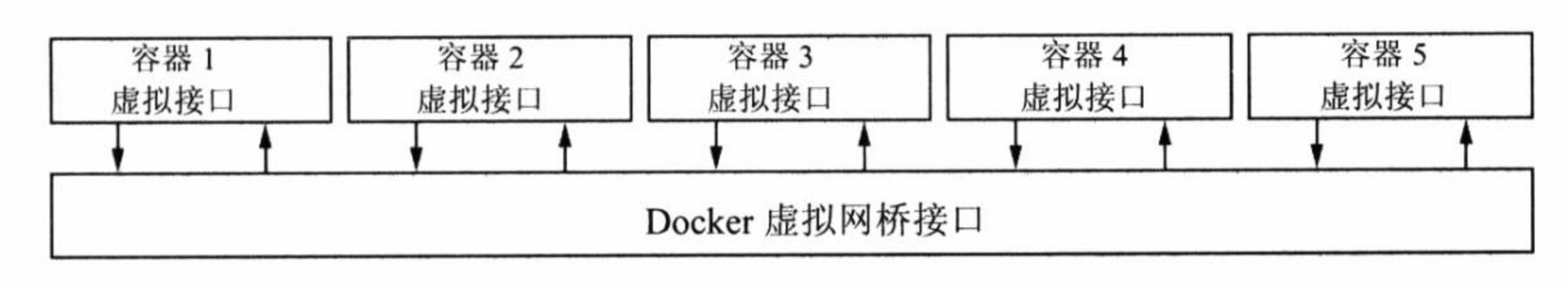


图 5-9 连接在同一个 Docker 网桥（docker 0）上的五个容器

为了确保你对以上结构的开放性有充分理解，下面的这个命令证明了以上容器是如何在网络上通信的。

```
docker run -it -rm dockerinaction/ch5_nmap -sS -p 3333 172.17.0.0/24
```

这个命令会运行一个叫做 nmap 的程序来扫描所有连接在网桥网络上的接口，并且查找那些能够在 3333 端口上接受网络连接的接口。如果你在某一个容器上运行这样一个服务，那么这个命令会发现这个容器，这样你就能够对这个容器发起网络连接了。

这种方式使得构建互相合作的容器变得非常简单，在容器之间构造通信通道不需要任何额外的工作。这就像一个开放的网络一样，尽管这可能存在一定的风险，但却是可以容忍的。因为使用一些安全程度较低的功能，比如默认密码、明文等，是非常常见的。天真的使用者可能期望网络拓扑结构或者本地防火墙能够对容器的访问权限进行控制，在一定程度上，这么说是对的，但是在默认情况下，容器对本地的其他容器是完全开放的。

当你启动 Docker 后台进程时，你可以选择关闭容器之间的网络连接。在多租户环境（multi-tenant environments）下，这么做是最佳实践方式。它能够最小化攻击面（attack surface），也就是减少攻击者可能造成的危害范围。你可以在启动 Docker 后台进程时使用 `--icc=false` 选项来达到这个效果：

```
docker -d -icc=false …
```

当跨容器通信被禁止了，除非被显式允许的流量，否则任何从容器到容器的网络流量都会被主机上的防火墙阻止。这些例外情况包含在 5.4 节中。

禁止跨容器通信在任何 Docker 环境中都是重要的一步。禁止之后，在这个环境中，想要正常工作，必须显式地声明容器的依赖。如果允许跨容器通信，在最好的情况下，一个混乱的配置也能够允许在依赖还没有准备好时启动容器。而在最糟糕的情况下，允许跨容器通信使得恶意软件能够攻击其他本地容器。

5.4.5 修改网桥接口的配置

在介绍下一个 archetype 之前，现在似乎是介绍网桥接口配置选项的最佳时机。除了本节，其他节的例子总是假设使用的是默认的网桥配置。

Docker 提供了三个选项来自定义网桥接口，这个接口在 Docker daemon 首次启动时就创建了。这些选项能够让使用者做以下事情：

- 定义网桥的地址和子网
- 定义容器所能够获取的 IP 地址的范围
- 定义最大传输单元（MTU）

为了定义网桥的 IP 地址和子网范围，可以在启动 Docker 后台进程时使用--bip 选项。至于为什么想要使用不同的 IP 地址范围，理由是各种各样的。当你想要使用它时，只需要使用这个选项就行了。

使用`--bip`选项（bip 是 bridge IP 的缩写，表示网桥 IP），你可以设置 Docker 创建的网桥接口的 IP 地址，也可以使用无类域内路由（CIDR）地址来设置子网的大小。无类域内路由概念提供了指定 IP 地址和路由前缀的方式。你可以阅读附录 B 中无类域内路由的简要介绍。网上有很多关于设置无类域内格式的地址的细节，但是如果你对掩码比较熟悉，那么下面的例子足以让你开始使用它了。

假设你想要将你的网桥IP地址设置为192.168.0.128，并且只想分配这个子网最后的128个地址。在这种情况下，你需要将`--bip`选项的值设置为 192.168.0.128/25。更详细地说，这个值会创建一个 IP 地址为 192.168.0.128 的 docker0 接口，并且子网 IP 地址的范围为 192.168.0.128 到 192.168.0.255。命令类似于：

```
docker -d -bip "192.168.0.128" …
```

当指定了网桥的网络后，接下来你可以自定义该网络中的容器的 IP 地址的范围。为了达到这个效果，`--fixed-cidr`选项提供了一个类似无类域内路由概念的配置。

和之前的情况类似，如果你想要保留最后的 64 个地址，那么你可以使用 192.168.0.192/26。当这个值被设置，Docker daemon 启动后，新的容器的 IP 地址限制于 192.168.0.192 到 192.168.0.255。唯一需要注意的就是，指定的范围必须在网桥网络的子网范围内（如果你感到困惑，网络上有很多非常好的文档或者工具能够帮你解决这个问题）：

```
docker -d -fixed-cidr "192.168.0.192/26"
```

我将不会在最后一个设置上花太多时间。网络接口对一个数据包的最大大小有一个限制（一个数据包就是网络通信的原子单位）。根据协议，以太网接口拥有 1500 字节的最大

数据包大小。这是默认的配置。在一些特殊的情况下，你需要更改 Docker 网桥的最大传输单元（MTU）。当你需要时，你可以使用--mtu 选项来设置这个大小（字节）：

```
docker -d -mtu 1200
```

熟悉 Linux 网络基础的人可能知道其可以使用自定义的网桥接口，而非默认的网桥接口。为了达到这个效果，配置你自己的网桥接口，然后当你启动 Docker 后台进程时，告诉它使用自定义的网桥接口来替代 docker0，使用的选项是-b 或--bridge。

如果你已经配置了一个网桥接口，叫做 mybridge，那么你可以使用以下命令来启动 Docker：

```
docker -d -b mybridge …
docker -d -bridge mybridge …
```

构建自定义的网桥需要对 Linux 内核工具有比较深的理解，但这部分内容不是本书必需的。但是你要明白，如果你的研究工作的确需要这部分内容，这些知识是可学习的。

5.5 Joined 容器

下一个隔离度更低的原型叫做 Joined 容器。这些容器共享一个网络栈，在这种情况下，容器之间没有任何的隔离。这意味着更少的控制和安全。尽管这不是最不安全的原型，但它是第一个打破容器之间界限的。

这种类型的原型通过将某一个容器接口的访问权提供给另外一个新的容器来构建。在这种情况下，接口就类似于共享的数据卷。如图 5-10 所示两个 Joined 容器的网络架构。

亲身体会 Joined 容器的最简单方式就是创建一个具体的容器，然后将它和一个新的容器连接起来。第一个命令启动了一个在本地回环接口监听的服务器，第二个命令列出了当前容器所有的开放端口。你会发现，第二个命令会将第一个命令创建的开放端口也列出来，因为两个容器共享相同的网络接口：

```
docker run -d -name brady
    -net none alpine:latest
    nc -l 127.0.0.1:3333

docker run -it
    -net container:brady
    alpine:latest netstat -al
```

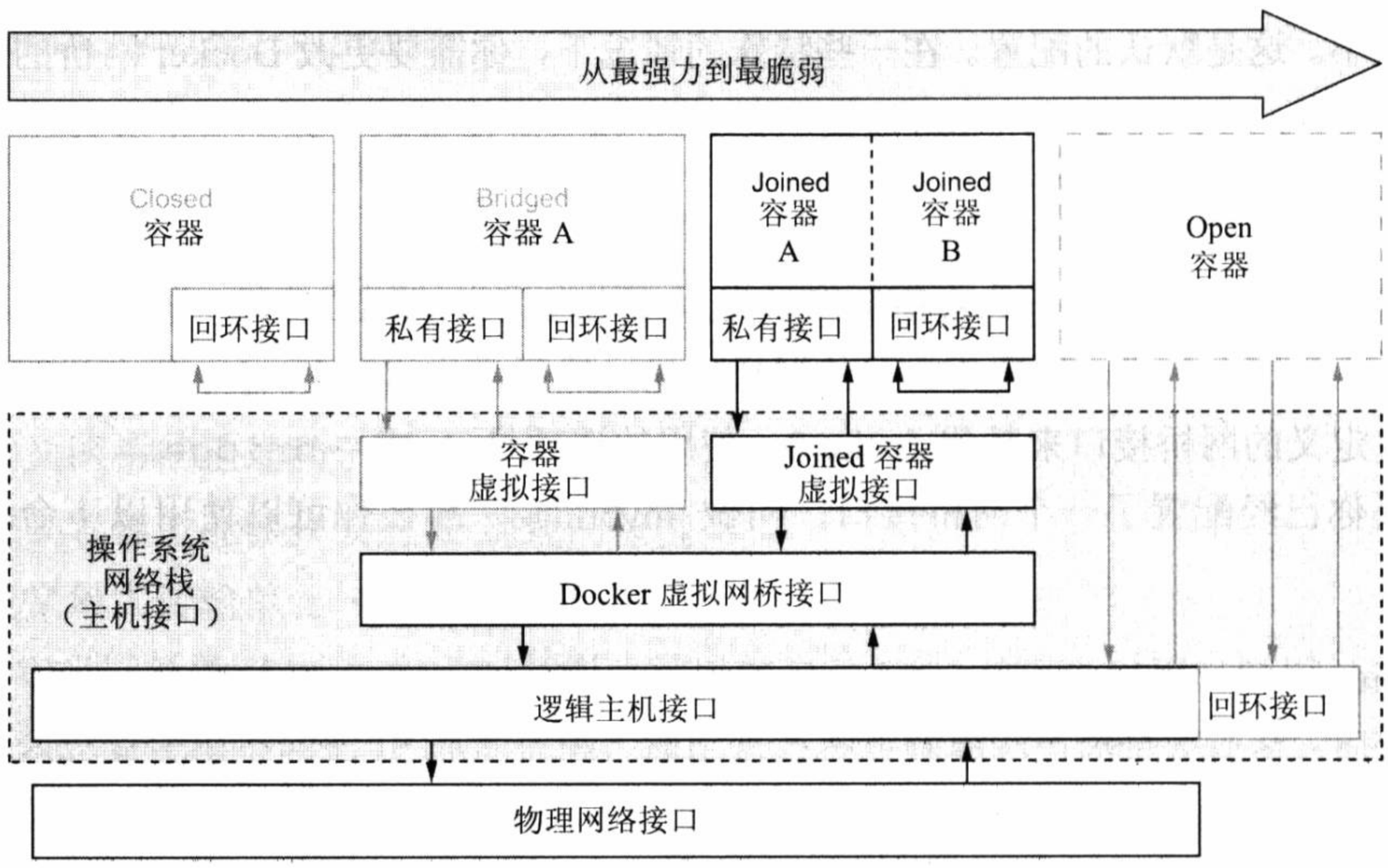


图 5-10 两个容器共享同一个网桥和同一个本地回环接口

以上两个命令将会创建两个共享相同网络接口的容器。因为第一个命令创建了一个 Closed 容器，因此第二个容器只共享那个本地回环接口。`--net` 选项中容器的值决定了新容器要和哪一个容器进行连接，容器名字或者 ID 都能标识容器的身份。

在这种情况下的容器维持了另一种形式的隔离。它们各自维持有不同的文件系统、不同的内存等，但是它们的确共享了同一个网络组件。这可能听起来令人有些担心，但是这种容器的确是非常有用的。

在上一个例子中，我们连接两个容器到同一个网络接口，并且这个网络接口不能够访问外部网络。在这个例子中，你扩展了 Closed 容器的有效性。当两个不同的程序各自有不同资源的访问权，它们想要互相通信，但又不想赋予对方对自己资源的直接访问权限时，这种模式是非常有用的。或者，当你有网络服务需要互相通信，但是网络访问或者服务发现机制（如 DNS）不可用，那么这种模式也是非常有用的。

暂时先把安全问题放在一边，使用 Joined 容器会再引入端口冲突问题。无论何时，当使用者创建 Joined 容器时他们需要意识到这个问题。比如说，如果被连接的容器运行有类似的服务，那么它们就会产生冲突。在这些情况下，冲突可以通过更加传统的方式来解决，比如说修改应用的配置。这些冲突可能在任何共享的接口上发生。当程序运行在容器外，它们和计算机上的其他程序同样共享一个主机接口，因此这种特定范围的增长依旧算是当前现状的一个改进。

当两个容器被连接，每一个接口都被共享，因此冲突可能在任何中的一个发生。乍一看，连接两个需要网桥访问权的容器是非常愚蠢的，毕竟，它们已经能够通过 Docker 网桥

子网进行通信了。但是考虑到以下情况，当一个进程需要通过其他的安全通道来监控另外一个进程，此时，容器间的通信会受到防火墙规则的限制。如果其中一个进程想要与另外一个进程在一个未开放的端口上进行通信，最佳的方式就是连接这两个容器。

我们为什么使用 Joined 容器

当你想要不同容器上的程序通过本地回环接口进行通信时，请使用 Joined 容器。

当一个容器中的程序将要改变 Joined 网络栈，而另外一个程序将要使用那个被改变的网络栈时，请使用 Joined 容器。

当你想要监控另外一个容器中某个程序的网络流量时，请使用 Joined 容器。

在你因为 Docker 允许任何新的容器连接正在运行的容器，而开始讨论 Docker 如何不安全之前，请记住给 Docker 发送任何命令都是需要特殊权限的，而那些拥有特权的攻击者可以做任何他们想做的事情，包括攻击任何容器中的代码和数据。在那种情况下，这种网络栈的共享行为就显得不那么严重了。

当你构建了多租户系统时，这会成为一个大问题。当你正在构建或者思考使用这么一个服务时，你首先要做的就是创建多个账户，并且尝试着从一个账户获取另外一个账户的权限。如果你做到了，那么对这个服务，你需要三思而行。因为，连接其他账户的网络栈或者挂载其他账户的数据卷都是非常严重的问题。

Joined 容器是有些脆弱，但它不是最脆弱的网络容器。这个名头要归 Open 容器所有。

5.6 Open 容器

Open 容器非常的危险。它没有网络容器，并且对主机网络有完全的访问权。包括对重要主机服务的访问权。Open 容器没有提供任何隔离，当你没有其他选择时它才应该被考虑。唯一一个补救的特性就是，没有特权的容器依旧不能够重新配置网络栈。如图 5-11 所示的一个 Open 容器的网络架构。

当你指定 host 作为 `docker run` 命令的`--net` 选项的值时，这种类型的容器会被创建：

option on the `docker run` command:

```
docker run --rm \
    --net host \
    alpine:latest ip addr
```

创建一个 Open 容器

运行这个命令会从最新的 alpine 镜像创建一个容器，并且没有任何的网络隔离。当你在这个容器中执行 `ip addr` 命令时，你可以看到所有主机上的网络接口，包括 docker 0。

如同你看到的，这个例子创建了一个容器，在容器中执行了一个命令，然后立马删除了这个容器。

使用这种配置，进程能够绑定低于 1024 的受保护的网络端口。

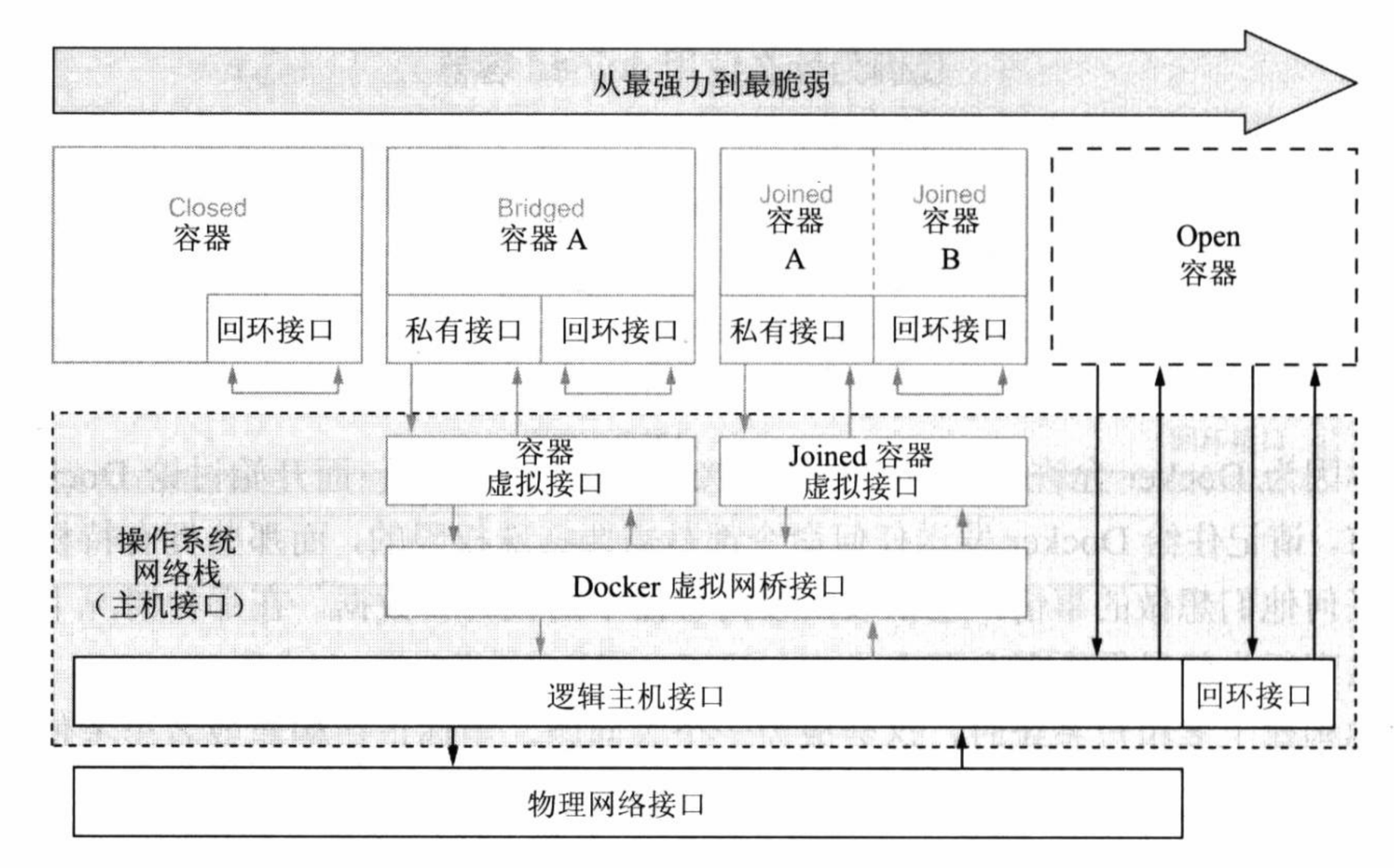


图 5-11 Open 容器对主机网络资源有完全访问权

5.7 跨容器依赖

既然你已经了解了哪些网络容器你能够创建，并且这些容器是如何和网络进行交互的，那么接下来你需要知道如何在一个容器中使用另外一个容器的网络软件。你已经了解了容器如何通过网桥网络进行通信，并且你可能已经开始思考如何拼接和构建这样一个小系统了。当你考虑到网桥网络在容器创建时动态赋予它 IP 地址，如何做到本地服务发现似乎成为了一个复杂的问题。

解决这个问题的一个方法是使用一个本地的 DNS 服务器，并且容器启动时在 DNS 服务器中进行注册。另外一个方法是你自己写一个程序扫描本地网络上监听某些已知端口的 IP 地址。这两个方法都能在动态环境中工作，但是需要比较大的工作量和额外的工具。当任意的跨容器通信被禁止时，这两个方法都将失效。当然，你可以强制所有的网络流量进出都通过已知的、已公开的端口。但是在有些情况下，你不可避免地需要某个网络端口的访问特权。Docker 提供了另外一个工具来处理这种情况。

5.7.1 链接——本地服务发现

当你创建了一个新的容器，你可以告诉 Docker，将它和另外一个容器相链接。当新容器被创建时，目标容器必须正在运行。原因非常简单，只有容器正在运行，它们才能维持其 IP 地址。如果它们停止了，那么它们将会失去 IP 地址的租约。

为新容器添加一条链接会发生以下三件事：

- 描述目标容器的环境变量会被创建。
- 链接的别名和对应的目标容器的 IP 地址会被添加到 DNS 覆盖列表中。
- 最有趣的是，如果跨容器通信被禁止了，Docker 会添加特定的防火墙规则来允许被链接的容器间的通信。

前两个特性对基本的服务发现非常有用，但是第三个特性在不牺牲容器间通信的情况下，加固了本地容器网络。

能够用来通信的端口就是那些已经被目标容器公开的端口。因此，当跨容器通信（ICC）被允许时，`--expose` 选项为容器端口到主机端口的映射提供了捷径。当跨容器通信（ICC）被禁止时，`--expose` 选项成了定义防火墙规则和在网络上显式声明容器接口的一个工具。在同样的情况下，链接成为了本地运行时服务的依赖的静态声明。这里有一个简单的例子，注意这些镜像并不是真实存在的：

```
docker run -d --name importantData \
    --expose 3306 \
    dockerinaction/mysql_noauth \
    service mysql_noauth start

docker run -d --name importantWebapp \
    --link imporantData:db \
    dockerinaction/ch5_web startapp.sh -db tcp://db:3306

docker run -d --name buggyProgram \
    dockerinaction/ch5_buggy
```

被链接的目的容器的名字

创建一条链接并且将别名设置为 db

这个容器没有到 importantData 容器的路由

从这个例子你能看到，我已经启动一个 MySQL 服务器（一个著名的数据库服务器），服务器配置看起来有些愚蠢。它的名字暗示说，这个服务器禁止了认证要求，因此任何可以连接到服务器的人都能够访问数据。之后，我又启动了一个重要的 web 应用，它需要访问 importantData 容器中的数据。我添加了一条从 importantWebapp 容器到 importantData 容器的链接。Docker 会为新容器添加能够描述如何连接 importantData 容器的信息。在这种情况下，当 web 应用向 **tcp://db:3306** 发起数据库连接请求时，它将会连接到数据库。最后，我又启动了另外一个容器，它包含了有 Bug 的代码。如果这段程序被攻击了，即便它运行

在非特权账户下，攻击者也能够探测该网桥网络。

如果跨容器通信被允许，那么攻击者很容易就能够从数据库中偷取信息。他们可以通过简单的网络扫描来锁定具体的开放端口，然后通过发起数据库链接就能够获得访问权了。即便网络管理员偶然看到了这条数据库链接，他也会认为这条链接是正确的，毕竟没有任何容器依赖被强制性构建。

如果跨容器通信被禁止，攻击者就不能够从被攻击的容器访问到其他容器了。

说实话，这是一个非常愚蠢的例子。请不要认为，简单地禁止跨容器通信就能够保护那些不能自保的资源。只有正确恰当的配置、强大的网络规则设置，加上服务依赖的声明，你才能构造一个深度安全防御的系统。

5.7.2 链接别名

链接是单向的网络依赖，当一个容器被创建并且指定了链接对象时，该依赖被创建。在之前已经提到过，`--link` 选项接受一个参数来实现这个目的。这个参数是容器名字或 ID 到别名的映射。别名只要求在被创建容器范围内是唯一的。因此，如果三个名为 a、b、c 的容器已经存在并且正在运行，那么我可以运行以下命令：

```
docker run -link a:alias-a -link b:alias-b -link c:alias-c
```

但是如果我犯错了，把一些或者全部的容器都赋予了同一个别名，那么这个别名只能包含其中一个容器的链接信息。在这种情况下，防火墙规则依旧会被创建，但是没有了链接信息，这些规则几乎是无用的。

链接别名带来了一个高层次的问题。运行在一个容器中的软件想要知道它所链接的容器或者主机的别名，之后它才能执行查询操作。类似于主机名，链接别名成为了一个多方必须都同意且保持一致的符号，这样系统才能够运行正确。链接别名就相当于合同。

开发者可能会假设数据库拥有 database 的别名，该名字也会在 DNS 覆盖系统中存在，基于这个假设，他们会构建应用，并且总是向 tcp://database:3306 发起查询链接。只要构建容器的人或进程创建了数据库的链接别名，或者使用`--add-host` 选项创建了主机名，之前假设主机名的方法才能够正确工作。也可以考虑另外一种方法，应用从名为 `DATABASE_PORT` 的环境变量上读取连接信息。这种环境变量的方法只有当带有该别名的链接被创建时才有效。

问题的关键在于不存在依赖声明或者运行时依赖的检测。对于构建容器的人来说，是可以这么做的，即便不提供所需的链接信息。而 Docker 的使用者必须依赖文档来获取这些

依赖信息，或者包含自定义的依赖检测和在容器启动时实行早报错（fail-fast）机制。我建议先构建依赖检测代码。举个例子，下面这段包含在 dockerinaction/ch5_ff 中的脚本验证了在容器启动时是否存在一个叫做 database 的链接别名。

```
!/bin/sh
if [ -z ${DATABASE_PORT+x} ] then echo "Link alias 'database' was not set!"
exit else exec "$@" fi
```

你可以通过运行以下的命令来查看脚本的效果：

```
docker run -d --name mydb --expose 3306 \
    alpine:latest nc -l 0.0.0.0:3306

docker run -it --rm \
    dockerinaction/ch5_ff echo This "shouldn't" work.

docker run -it --rm \
    --link mydb:wrongalias \
    dockerinaction/ch5_ff echo Wrong.

docker run -it --rm \
    --link mydb:database \
    dockerinaction/ch5_ff echo It worked.

docker stop mydb && docker rm mydb
```

创建有效的链接目标

测试不建立链接的情况

测试不正确的链接别名的情况

测试错误别名

停止并删除链接目标容器

Docker 创建链接时会改变环境变量，而这段脚本就是通过检测这个环境变量来判断链接别名是否存在。当你在第 7 章开始构建你自己的镜像时，会发现这非常有用。

5.7.3 环境变量的改动

我已经提到过，创建一条链接会在新容器中添加链接信息。这些连接信息一方面存储在环境变量中，另一方面通过在 DNS 覆盖系统中添加主机名的映射来将链接信息注入新容器中。让我们通过一个例子来观察这些行为吧：

```
docker run -d --name mydb \
    --expose 2222 --expose 3333 --expose 4444/udp \
    alpine:latest nc -l 0.0.0.0:2222

docker run -it --rm \
    --link mydb:database \
    dockerinaction/ch5_ff env

docker stop mydb && docker rm mydb
```

创建有效的链接目标

创建链接并且列出所有环境变量

这将会输出一整块类似下面格式的信息：

```
DATABASE_PORT=tcp://172.17.0.23:3333
DATABASE_PORT_3333_TCP=tcp://172.17.0.23:3333
DATABASE_PORT_2222_TCP=tcp://172.17.0.23:2222
DATABASE_PORT_4444_UDP=udp://172.17.0.23:4444
DATABASE_PORT_2222_TCP_PORT=2222
DATABASE_PORT_3333_TCP_PORT=3333
DATABASE_PORT_4444_UDP_PORT=4444
DATABASE_PORT_3333_TCP_ADDR=172.17.0.23
DATABASE_PORT_2222_TCP_ADDR=172.17.0.23
DATABASE_PORT_4444_UDP_ADDR=172.17.0.23
DATABASE_PORT_2222_TCP_PROTO=tcp
DATABASE_PORT_3333_TCP_PROTO=tcp
DATABASE_PORT_4444_UDP_PROTO=udp
DATABASE_NAME=/furious_lalande/database
```

可以看到，有多个环境变量由于链接的创建而创建。所有跟某一具体链接相关的变量都会使用该链接别名作为前缀。其中还会有一个以_NAME 结尾的变量，它由当前容器的名字，一个斜杠和链接的别名组成。对于每个被链接的容器开放的端口，都会有四个单独的环境变量，并且环境变量的名字包含了对应的开放端口。模式大概如下：

- <ALIAS>_PORT_<PORT NUMBER>_<PRO TOCOL TCP or UDP>_PORT

这个变量仅仅包含了端口数字。这看起来非常古怪，因为端口已经被包含在变量名中了。当你在过滤那些包含有 TCP_PORT 字符串的环境变量时，这是非常有用的。这么做会得到所有端口的列表。

- <ALIAS>_PORT_<PORT NUMBER>_<PRO TOCOL TCP or UDP>_ADDR

这种变量的值表示接收网络连接的容器的 IP 地址。如果别名一样，那么它们会拥有一样的值。

- <ALIAS>_PORT_<PORT NUMBER>_<PRO TOCOL TCP or UDP>_PROTO

就像以_PORT 为后缀的变量一样，值的信息也被包含在变量名中。认识到协议不仅仅可以是 TCP 是非常重要的。UDP 也是支持的。

- <ALIAS>_PORT_<PORT NUMBER>_<PRO TOCOL TCP or UDP>

这种变量包含了上面所有的信息，并且以 URL 格式表示。

还有一个额外的环境变量，模式为<ALIAS>_<PORT>，它包含其中一个开放端口的连接信息，格式为 URL。

这些环境变量包含了应用开发者在向被链接容器发起网络链接时所需要的任何信息。但是，如果开发者已经提前定义好了端口和协议，那么主机名解析才是他们所需要的，他们可以通过 DNS 来完成这个目标。

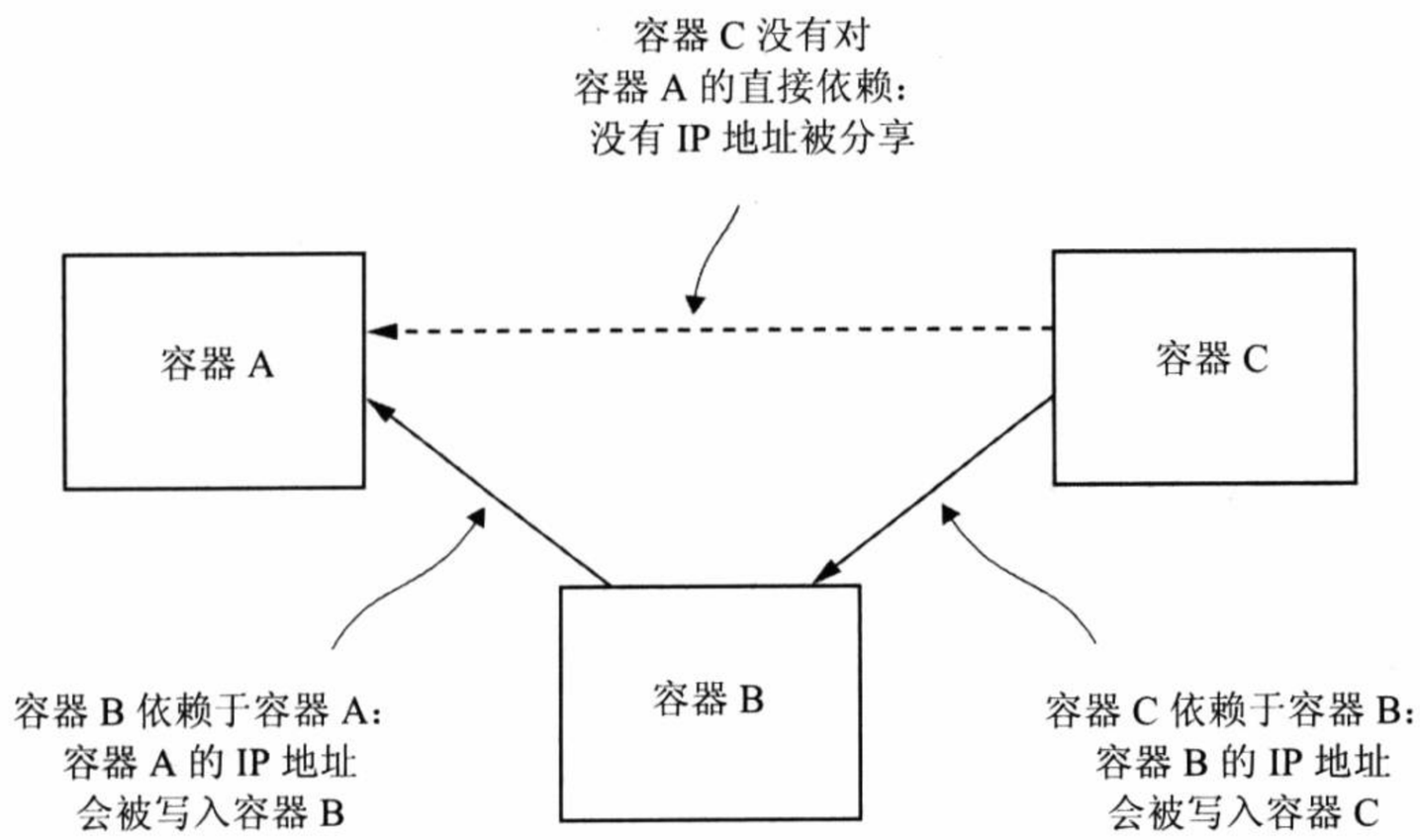


图 5-12　链接不具有传递性。容器 C 和容器 A 之间没有链接关系

5.7.4　链接的本质和缺点

链接的本质就是静态的、具有方向性和无传递性的依赖。无传递性意味着被链接的容器不会继承链接。更具体地说，如果我链接了 B 到 A，并且链接了 C 到 B，那么 C 和 A 之间不存在链接。如图 5-12 所示这三个容器的关系。

链接通过检测目的容器的网络信息（IP 地址和开放端口），然后将这些信息注入新容器中。由于这个操作过程是在容器创建期间进行的，并且在一个容器运行之间，Docker 不知道该容器的 IP 地址，因此链接只能在新容器到已存在的容器之间构建。当然，这并不是说通信是单向的，而是说服务发现是单向的。这也意味着如果某个依赖由于某些原因停止了，则这条链接也会被破坏。记住，只有容器在运行状态，它才能够维持 IP 地址的租约。因此一旦容器被停止或者重启了，那么它将失去 IP 地址租约并且任何链接到该容器的容器保留的都是过期的链接信息了。

这个特性使得有些人痛批链接的价值，问题在于，失败的容器越深，需要重启的容器越多，产生的多米诺效应就越大。这也许对于某些情况是非常大的问题，但是你必须考虑具体的影响。

如果一个非常重要的服务，比如数据库出问题了，可用性事件已经发生。所选中的服务发现方法会影响故障恢复程序。不可用的服务可能会在同一个或者不同的 IP 地址上进行恢复。链接会在 IP 地址发生改变时才会断裂，并且需要重启。这导致一些人转移到更加动态的查询系统，比如说 DNS。

但是 DNS 系统会有生存时间（TTL），它会降低 IP 地址改变后的传播速度。如果一个 IP 地址在恢复期间改变了，那么使用 DNS 是更加简单方便的，但是只有在数据库链接能够连接失败，数据库重连允许超时，并且 DNS TTL 过期时间比重启容器的时间更短的情况下，系统恢复才会发生得更迅速。在禁止容器链接时，你必须允许跨容器通信。

如果你的应用启动很慢，并且你需要在服务发现上处理 IP 地址的改变，那么你可能会考虑 DNS。否则，考虑由容器链接构建的静态依赖链。构建一个系统，它能够在依赖失败时重启那些失败的容器，这是能够做到的。

本章专注于单主机 Docker 网络，在这个范围内，链接是非常有用的工具。大多数的环境跨越多台计算机。服务可移植性的思想是，一个服务能够运行在任何机器、任何容器中。它认为存在一个系统，系统中的进程可能会运行在任何地方，这个系统比具有严格的本地限制的系统更加的健壮。我认为这是正确的，但是展示 Docker 在这两种情况中是如何使用的，也是非常重要的。

5.8 小结

网络是一个比较宽泛的概念，需要花费好几本书才能够覆盖全面。Docker 提供的单主机网络工具涉及一些网络的基础知识，本章应该能够帮助读者对这些知识有一个基础的认识。在阅读本章的过程中，你能够学习到以下知识：

- Docker 提供了四个网络容器原型：Close 容器、Bridged 容器、Joined 容器和 Open 容器。
- Docker 创建的网桥网络会将参与的容器互相绑定到一起，并且绑定到主机所依附的网络上。
- 当 Docker 后台进程启动时，可以使用 `docker` 命令行选项用自定义的网桥接口替代 Docker 创建的网桥接口。
- `docker run` 命令的选项能够被用来开放容器接口的端口，绑定容器开放端口到主机的网络接口上，还能链接其他的容器。
- 禁止任意的跨容器通信是非常简单的，并且能够构建一个深度防御的系统。
- 使用链接能够提供一个低负载的本地服务发现机制，并且映射具体的容器依赖。

第 6 章　隔离——限制危险

本章介绍

- 限制资源
- 共享容器内存
- 用户、权限和管理员特权
- 授权访问某个具体的 Linux 功能
- 加强 Linux 隔离和安全的工具：SELinux 和 AppArmor

容器提供的是进程上下文的隔离，而不是整个系统的虚拟化。这之间的语义区别可能看起来很微妙，但是产生的影响确实巨大。第 1 章稍微提了一下区别。第 2 到 5 章，每章都覆盖了一部分 Docker 容器的隔离特性。本章覆盖剩余的部分，并且包含了加强系统安全的内容。

本章包含的特性主要专注于管理或者限制正在运行的软件的危险性。你将会学习到如何授予容器资源的访问权，如何访问共享的内存，如何在特殊用户下运行程序，如何控制容器对你的计算机所能做的改变，如何集成其他的 Linux 隔离工具。以上有些内容包含了超出本书范围的 Linux 特性。在这些情况下，我会尝试让你了解它们的目的，并且给你一些基础的使用案例，然后你就能够将它们和 Docker 集成在一起了。如图 6-1 所示，列出了八个命名空间（namespace）和用来构建 Docker 容器的特性。

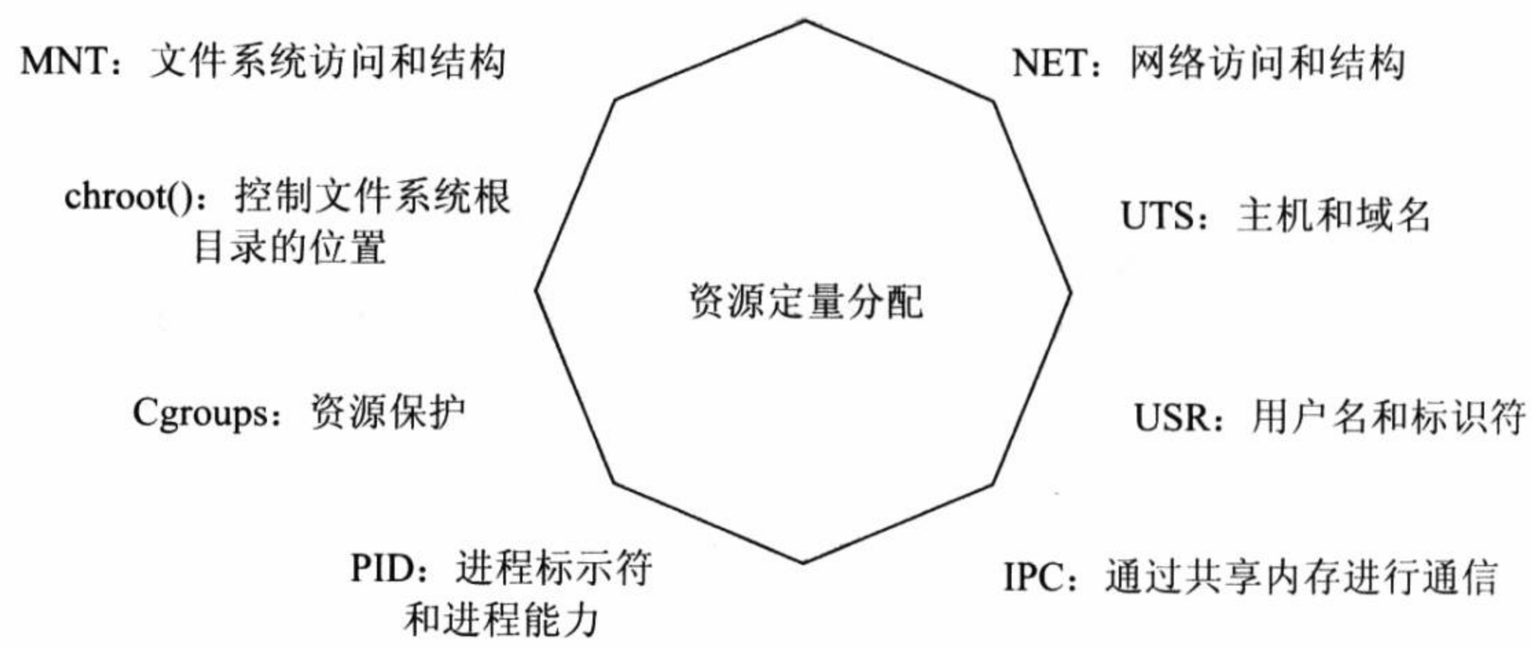


图 6-1 八边形容器

最后一次提醒，Docker 和它使用的技术都在不断地迭代更新。因此当你使用本章提到的工具前，记得去检查下它们的开发状况，看看有没有加强的功能和最新的最佳实践方案。

6.1 资源分配

物理系统资源非常稀少，比如说内存和 CPU 时间。如果进程的资源消耗超过可用的物理资源，那么进程将会遭遇性能问题并且可能会停止运行。构建强隔离系统的内容包含对单独的容器提供资源的定量分配。

如果你想要确保一个程序不会因使用资源过多而压垮其他程序，最简单的办法就是限制这个进程所能够使用的资源。Docker 为 `docker run` 和 `docker create` 命令提供了三个选项来管理三种不同类型资源的分配。这三种类型为内存、CPU 和设备。

6.1.1 内存限制

内存限制是你能够对容器所做的最基础的限制。它限制了容器中的进程能够使用的内存大小。内存限制对确保一个容器不会因使用资源过多而影响到运行在同一台计算机上的其他容器非常有用。你可以通过在 `docker run` 或 `docker create` 命令上使用`-m` 或`--memory` 选项来设置内存限制。这个选项会接受一个值和一个基础单元作为参数。格式如下：

```
where unit = b, k, m or g
```

在这个命令中，`b` 表示字节，`k` 表示千字节，`m` 表示兆字节，`g` 表示千兆字节。利用这个知识，在下面的例子中启动一个数据库应用：

```
docker run -d --name ch6_mariadb \
    --memory 256m \
    --cpu-shares 1024
    --user nobody \
    --cap-drop all \
    dockerfile/mariadb
```

设置一个内存限制

运行以上的命令，你会启动一个内存限制为 256 兆字节的容器，容器中安装有一个名为 MariaDB 的数据库应用。你可能还注意到这个命令还有一些额外的选项。本章会覆盖这些内容，但是你应该能够根据选项的名字猜测到它们的作用。还需要注意，以上命令并没有开放任何端口或者绑定任何端口到主机接口。连接到这个数据库的最简单方式就是从其他容器链接该容器。在我们开始学习这些内容前，我想要确认你已经对上面发生的内容和如何使用内存限制有了全面的理解。

理解内存限制最重要的是明白内存限制并不是内存保留。并不是说它保证具体大小的内存是可用的，而是说它会防止容器使用超出具体大小的内存资源。

在你分配内存之前，你应该考虑两件事情。第一，你正在运行的软件能够在预期的内存限制下正常工作么？第二，运行软件的系统能够支持分配的内存数量么？

第一个问题一般来说比较难回答。对于开源软件来说，提供最低运行配置需求是不常见的。即便提供了，你也需要基于你交给程序处理的数据大小来判断程序所需要的内存大小。不管怎样，人们更趋向于高估资源，然后通过实验和错误来调整资源大小。内存敏感的软件，如数据库；熟练的专家，如数据库管理员，能够提供更好的估计和建议。即便如此，这个问题常常需要由另外一个问题来回答：你有多少内存资源？因此这带来了第二个问题。

运行软件的系统能够支持分配的内存数量么？设置一个比系统可用内存资源更大的内存限制是有可能发生的。在拥有内存交换空间（虚拟内存）的计算机上，容器还可能会意识到内存限制。但是设置一个比任何物理内存资源都要大的内存限制总是可能的，在这种情况下，系统的物理资源限制会一直限制着容器。

最后，你需要理解当内存资源耗尽时，软件可能以多种方式失败。一些程序可能会由于内存访问错误而失败，其他的可能不会失败，而是向日志文件中记录内存溢出错误。Docker 既不检测这种错误，也不尝试缓解这种问题。它所能做的就是应用你通过 `--restart` 选项指定的重启逻辑，这部分内容在第 2 章中描述过。

6.1.2 CPU

CPU 时间和内存一样稀少，但是 CPU 时间少的后果不是失败，而是性能降低。一个等待 CPU 时间的暂停进程依旧工作正常，但是如果进程在进行一些重要的数据处理任务，或

者是正在进行收入回收（revenue-generating）的 web 应用或者 App 的后端服务，那么进程的缓慢可能比进程失败更加糟糕。Docker 允许你从两种方式来限制容器 CPU 资源。

第一，你可以指定容器的相对权重。Linux 使用这个权重来决定该容器应该占用的 CPU 时间百分比。这个百分比是相对于所有对容器可用的处理器的 CPU 周期的总和来计算的。

为了设置一个容器的 CPU 时间分配和建立它的相对权重，docker run 和 docker create 命令都提供了一个--cpu-shares 选项。选项的值应该是一个整数（这意味着你不应该加上引号）。之后，启动另外一个容器来查看 CPU 时间分配是如何工作的：

```
docker run -d -P --name ch6_wordpress \
--memory 512m \
--cpu-shares 512 \
--user nobody \
--cap-drop net_raw \
--link ch6_mariadb \
wordpress:4.1
```

设置一个相对权重

这个命令会下载并且运行 4.1 版本的 WordPress。WordPress 由 PHP 语言写成，这是一个软件如何适应安全威胁挑战的极佳例子。现在，我们已经启动它了，并且添加了一些额外的防范措施。如果你想要看到它运行在你的计算机上，可以使用 `docker port ch6_wordpress` 命令来获得服务所运行在的端口（我称之为`<port>`），并且在你的 web 浏览器中打开 http://localhost:<port>。记住，如果你在使用 Boot2Docker，你需要使用 `boot2docker ip` 命令来获取 Docker 所运行在的虚拟机的 IP 地址。当你获得这个 IP 地址，在之前的 URL 中用这个值替代 localhost。

当你启动了 MariaDB 容器，你将它的相对权重（`cpu-shares`）设置为 `1024`，同时你将 WordPress 容器的相对权重设置为 512。这些设置构建了这样一个系统：每当 WordPress 容器获得一个 CPU 周期，MariaDB 容器就能够获得两个。如果你启动了第三个容器，并且设置它的`--cpu-shares` 值为 2048，那么它将获得一半数量的 CPU 周期，MariaDB 和 WordPress 容器将会按照之间的比例分割剩下的一半。如图 6-2 所示，描绘了全部权重的变化对应的比例变化。

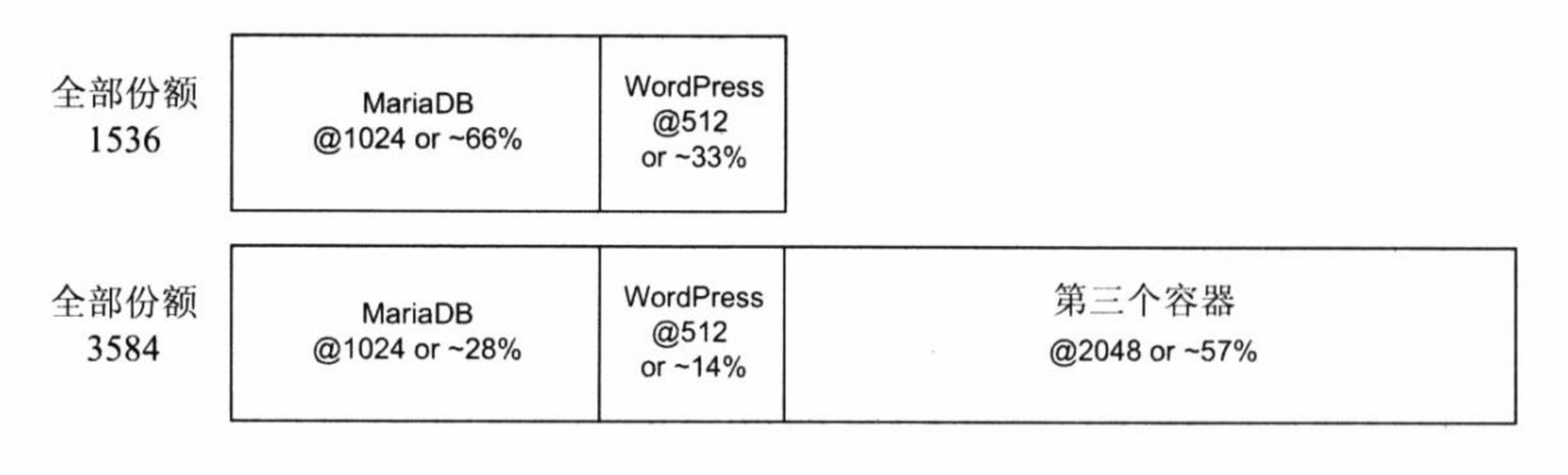


图 6-2 相对权重和 CPU 分配比例

CPU 资源限制不同于内存限制，只有在 CPU 时间上存在竞争时，它才会被强制执行。如果其他的进程和容器处于空闲中，那么被限制的容器可能会超出 CPU 资源限制。这是可取的，因为这种策略确保 CPU 不会被浪费，而且当其他进程需要 CPU 时，被限制的进程会让出 CPU。这个工具的目的就是为了防止一个或者多个进程使用过多资源而压垮计算机，而不是为了阻碍这些进程的性能。默认配置不会限制容器，因此容器能够使用 100%的 CPU。

Docker 提供的第二个特性就是为一个容器指定一个具体的、可使用的 CPU 集合。大多数现代硬件使用了多核 CPU。粗略地说，一个 CPU 能够并行处理的指令的数量和它拥有的核的数量一致，这对在同一台计算机上运行多个进程是非常有用的。

上下文切换就是从执行一个进程切换到执行另一个进程的任务。这个操作十分昂贵，因此可能对你系统的性能产生显著的影响。确保某些重要的进程不会在同一个 CPU 核集合中执行能够减少上下文的切换，在某些情况，这种行为是有意义的。你可以使用 `docker run` 或 `docker create` 命令的`--cpuset-cpus` 选项来限制容器只能在某一指定的 CPU 核集合中执行。

想要看到 CPU 集合限制的效果，你可以对你计算机上某一个 CPU 核加压并且测试 CPU 负载：

```
# Start a container limited to a single CPU and run a load generator
docker run -d \
    --cpuset-cpus 0 \                                  ← 限制只能使用标号为 0 的 CPU
    --name ch6_stresser dockerinaction/ch6_stresser

# Start a container to watch the load on the CPU under load
docker run -it --rm dockerinaction/ch6_htop
```

一旦你允许了第二个命令，你将会看到 `htop` 命令显示了正在运行的进程和当前可用 CPU 的负载。`ch6_stresser` 容器会在 30 秒后停止运行，因此当你在运行这个实验时，不要拖延，不然你将看不到结果。

你可以按下【Q】键来退出 htop 程序。在继续学习下面的内容之前，记得停止并删除 `ch6_stresser` 容器：

```
docker rm -vf ch6_stresser
```

当我第一次使用这个特性时，我认为这是令人兴奋的。为了加深理解，请对 `--cpuset-cpus` 选项使用不同的值，重复多次实验。如果你做了，你将会看到在不同 CPU 核或者不同 CPU 核集合中的进程的运行状况。值的格式可以是列表或者范围：

- `0，1，2` —— 一个列表包含了 CPU 的前三个核
- `0-2` —— 一个范围包含了 CPU 的前三个核

6.1.3 设备的访问权

设备是最后一种资源类型。不同于 CPU 和内存限制，设备的访问并不是一种限制。这更像是一种资源认证控制。

Linux 系统有各种各样的设备，包括硬盘、光驱、USB 设备、鼠标、键盘、声卡和网络摄像头。在默认情况下，容器对某些设备具有访问权，此外，Docker 为每个容器创建了剩下的设备（比如虚拟终端）。

有时，在主机和特定容器之间共享设备可能非常重要。考虑这种情况，当你正在运行某些计算机视觉软件，这些软件需要网络摄像头的访问权。在这种情况下，你需要授予运行有这些软件的容器对网络摄像头的访问权；你可以使用--device 选项来指定一个设备的集合，这些设备会被挂载进新容器中。下面这个例子会将在/dev/video0 位置的网络摄像头映射到新容器的同一个位置上。注意，当且仅当在/dev/video0 上有一个网络摄像头设备时，这个例子才能正确运行：

```
docker -it --rm \
    --device /dev/video0:/dev/video0 \
    ubuntu:latest ls -al /dev
```

挂在 video0 设备

--device 选项的值必须是主机操作系统上的设备文件到新容器中位置的映射。这个选项能够被设置多次，授予容器不同设备的访问权。

那些拥有自定义硬件或者专属驱动的人会发现这个特性非常有用。相对于修改他们的操作系统，这种特性是更加可取的。

6.2 共享内存

Linux 提供了一些工具，用来在同一台计算机上的进程之间共享内存。这种形式的跨进程通信（IPC）能够以内存的速度执行。当由于网络延迟或者基于管道的 IPC 阻碍了软件的性能，使之达不到需求时，共享内存的方式就常常被使用。基于共享内存 IPC 的最佳使用案例发生在科学计算领域和一些流行的数据库技术中，比如 PostgreSQL。

在默认情况下，Docker 为每一个容器创建了一个独立且唯一的 IPC 命名空间。Linux 的 IPC 命名空间分区共享内存单元，比如说命名的共享内存块、信号量和消息队列。如果你不确定这些是什么，没有关系，你只需要知道 Linux 程序使用这些工具来协调处理的。IPC 命名空间的作用就是防止一个容器中的进程访问主机或者其他容器的内存。

6.2.1 跨容器的进程间通信

我已经创建了一个名为 dockerinactionch6_ipc 的镜像，它同时包含了一个生产者进程和消费者进程。它们之间通过共享内存来通信。下面的命令会帮助你理解在不同容器中运行这些进程会有什么问题：

```
docker -d -u nobody --name ch6_ipc_producer \
    dockerinaction/ch6_ipc -producer
docker -d -u nobody --name ch6_ipc_consumer \
    dockerinaction/ch6_ipc -consumer
```

启动生产者进程

启动消费者进程

以上的命令启动了两个容器。第一个创建了一个消息队列并且在上面广播信息。第二个从消息队列中拉取信息，并且将信息写入到日志文件中。你可以通过下面的命令查看两个容器的日志文件，来获取两个容器正在做什么：

```
docker logs ch6_ipc_producer
docker logs ch6_ipc_consumer
```

你会发现，你启动的容器似乎出现了问题。消费者容器永远不会在消息队列中看到消息。每一个进程都使用相同的 key 来标识共享的内存资源，但是它们指向的却是不同的内存。原因在于，每一个容器都拥有它们自己的共享内存命名空间。

如果你想要在不同容器中使用共享内存进行通信，那么你需要使用--ipc 选项来连接它们的 IPC 命令空间。--ipc 选项支持这样一种容器模式，它创建的新容器的 IPC 命名空间和另外一个目标容器是一样的。这就像第 5 章提到的--net 选项。如图 6-3 所示了容器和它们的命名空间中的共享内存池的关系。

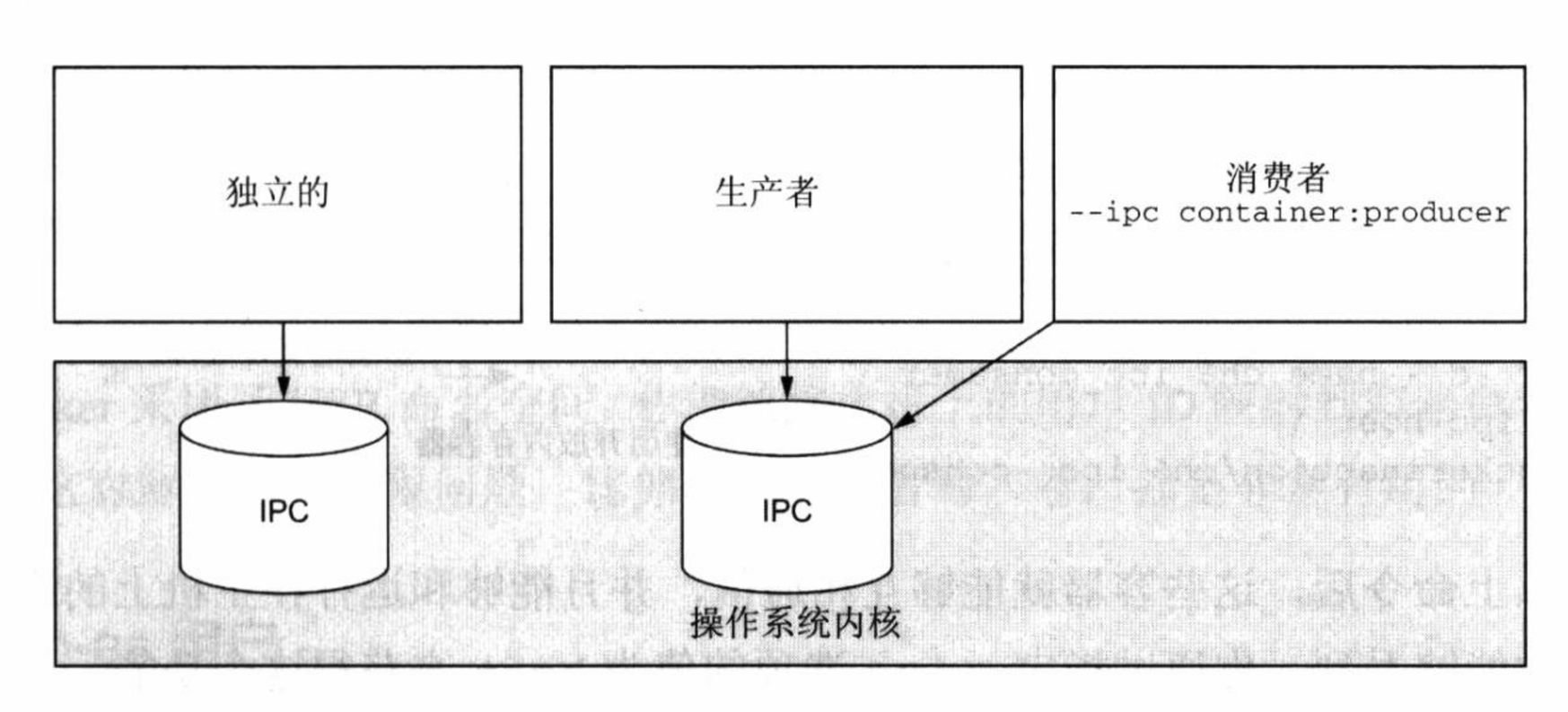


图 6-3 三个容器和它们的共享内存池，消费者和生产者共享同一个内存池

使用下面的命令来测试被连接的 IPC 命名空间效果：

```
docker rm -v ch6_ipc_consumer
docker -d --name ch6_ipc_consumer \
    --ipc container:ch6_ipc_producer \
    dockerinaction/ch6_ipc -consumer
```

删除原始的消费进程

启动新的消费者进程

连接 IPC 命名空间

以上的命令删除了原有的消费者容器，复用 ch6_ipc_producer 容器的 IPC 命名空间，重建了消费者容器。现在，消费者进程应该能够访问相同的内存位置了。你可以通过以下的命令查看每个容器的日志文件，来查看它们是否正确工作：

```
docker logs ch6_ipc_producer
docker logs ch6_ipc_consumer
```

在继续学习后面内容之前，记得清理正在运行的容器：

- 选项 v 会清理卷
- 如果容器正在运行，选项 f 会停止它们
- rm 命令支持多个容器

```
docker rm -vf ch6_ipc_producer ch6_ipc_consumer
```

复用共享内存命名空间，会导致明显的安全隐患。但是如果你需要这个功能，你能够简单地通过这个选项就能获得。在容器间共享内存是比主机上共享内存更加安全的替代方案。

6.2.2 开放内存容器

内存隔离是一种优良特性。如果你想要和主机运行在同一个命名空间中，你可以使用开放内存容器（open memory container）：

```
docker -d --name ch6_ipc_producer \
    --ipc host \
    dockerinaction/ch6_ipc -producer

docker -d --name ch6_ipc_consumer \
    --ipc host \
    dockerinaction/ch6_ipc -consumer
```

启动生产者进程

使用开放内存容器

启动消费者进程

使用开放内存容器

运行以上命令后，这些容器就能够互相通信，并且能够和运行在主机上的进程通信。从例子中你能够看到，你通过指定--ipc 选项的值为 host 来获得这个功能。当你想要和一个必须运行在主机上的进程进行通信时，你可能会使用这个功能，但是，在一般情况下，应该避免使用它。

请随意查看例子的源代码，尽管这是一段丑陋且简单的 C 程序代码。你可以通过在 Docker Hub 上的镜像网页上找出源仓库的链接来获得这段代码。

你可以使用和 6.2.1 节一样的代码来清理你创建的容器：

```
docker rm -vf ch6_ipc_producer ch6_ipc_consumer
```

开放内存容器非常危险，但是它还是比直接在容器外面运行这些进程更加安全。

6.3 理解用户

在默认情况下，Docker 以容器中的 root 用户来启动容器。root 用户几乎拥有对当前容器的所有访问特权。任何以 root 用户运行的进程都会继承它的权限。如果其中一个进程有漏洞，那么它可能就会危害整个容器。有两种方法能够限制这种危害，但是防止这种问题的最有效办法就是不使用 root 用户。

但也存在一些合理的意外情况，这时使用 root 用户是最佳的选择。比如，当构建镜像时，或者没有其他选项的运行时（runtime），你需要使用 root 用户。当你想要在容器中运行系统管理软件时，也有一些其他的类似情况。在这些情况下，进程不仅需要容器的访问特权，还需要主机操作系统的特权。本节包含了对这些问题的一系列解决办法。

6.3.1 Linux 用户命令空间

Linux 最近发布了一个新的用户（USR）命名空间，它允许一个命名空间中的用户被映射到另一个命名空间的用户上。这个新的命名空间类似于进程标识符（PID）命名空间。

Docker 还没有集成 USR 命名空间。这意味着，如果一个容器的用户 ID（数字，不是名字）和主机上的一个用户一样，那么该容器中的用户和主机上的用户拥有相同的主机文件权限。这并不是一个问题，因为容器中的文件系统的改动只会保留在容器的文件系统中，不会影响到主机。但是，这能够影响到卷（volume）。

当 Docker 采用了 USR 命名空间，你能够将主机上的用户 ID 映射到容器命名空间的用户 ID 上。这对解决文件权限问题，比如说读取或者写入卷，都是非常有用的。

6.3.2 run-as 用户

在你创建一个容器之前，如果你能够说明哪一个用户名（和用户 ID）默认被使用，这

是极好的。在默认情况下，这个由镜像指定。目前没有办法能够检测出一个镜像指定的默认用户。这个信息也不包含在 Docker Hub 中。并且也没有命令能够检测镜像的元数据。

最接近预期并且可用的是 docker inspect 命令。inspect 子命令能够显示出一个容器的元数据。容器元数据包含了镜像的元数据。如果你创建了一个容器——让我们称之为 bob 吧，你就能够获取这个容器正在使用的用户名：

```
docker create --name bob busybox:latest ping localhost

docker inspect bob

docker inspect --format "{{.Config.User}}" bob
```

显示 bob 镜像的所有元数据

只显示 bob 镜像定义的 run-as 用户

如果显示结果空白，那么容器会默认使用 root 用户来启动。如果结果不为空，要么镜像作者指定了一个默认的 run-as 用户，要么就是当你创建这个容器时，你指定了 run-as 用户。第二个命令中的-f 或--format 选项允许你为输出结果指定模板。在这个例子中，你选择文档的 Config 属性的 User 域。这个值可以是任何有效的 Go 语言模板，如果你感觉能够掌握它，那么你就能够获得更多创新性的结果。

这个方法存在一些问题。第一，run-as 用户可能会被镜像使用的脚本所更改。这些常常是启动或者初始化的脚本。而 docker inspect 返回的元数据仅仅是容器启动时的配置。因此，如果用户被脚本改变了，改动的结果并不会反馈到 docker inspect 命令的输出上。第二，你必须用镜像创建一个容器来获得这些信息，这是非常危险的。

现在，唯一能够解决这两个问题的方法就是检测镜像的内容。当你下载好镜像后，你能够打开镜像文件，手动查看它的元数据和初始化脚本。但是这么做非常地消耗时间，并且很容易犯错。从目前来看，运行一个简单的实验来检测默认用户是更好的方式。下面的命令能够解决第一个问题，但不能解决第二个问题：

```
docker run --rm --entrypoint "" busybox:latest whoami

docker run --rm --entrypoint "" busybox:latest id
```

输出结果：root

输出结果：uid=0(root)gid=0(root) groups=10(wheel)

结果证明这两个命令能够获得镜像默认用户的信息。拥有 whoami 和 id 这两个命令在 Linux 发行版中是非常常见的，因此它们能够在任何给定的镜像中运行。第二个命令更加强大，它能够同时显示 run-as 用户的名字和 ID 信息。注意，这两个脚本都很谨慎地重置了容器的 entrypoint，这确保了以上命令中跟随在镜像名字后的命令能够被容器执行。一流的镜像元数据工具已经包含了这个功能，因此以上的命令仅仅是这些工具的简陋替代品。接下来，考虑如图 6-4 所示的中两个 root 用户之间的简单交流。

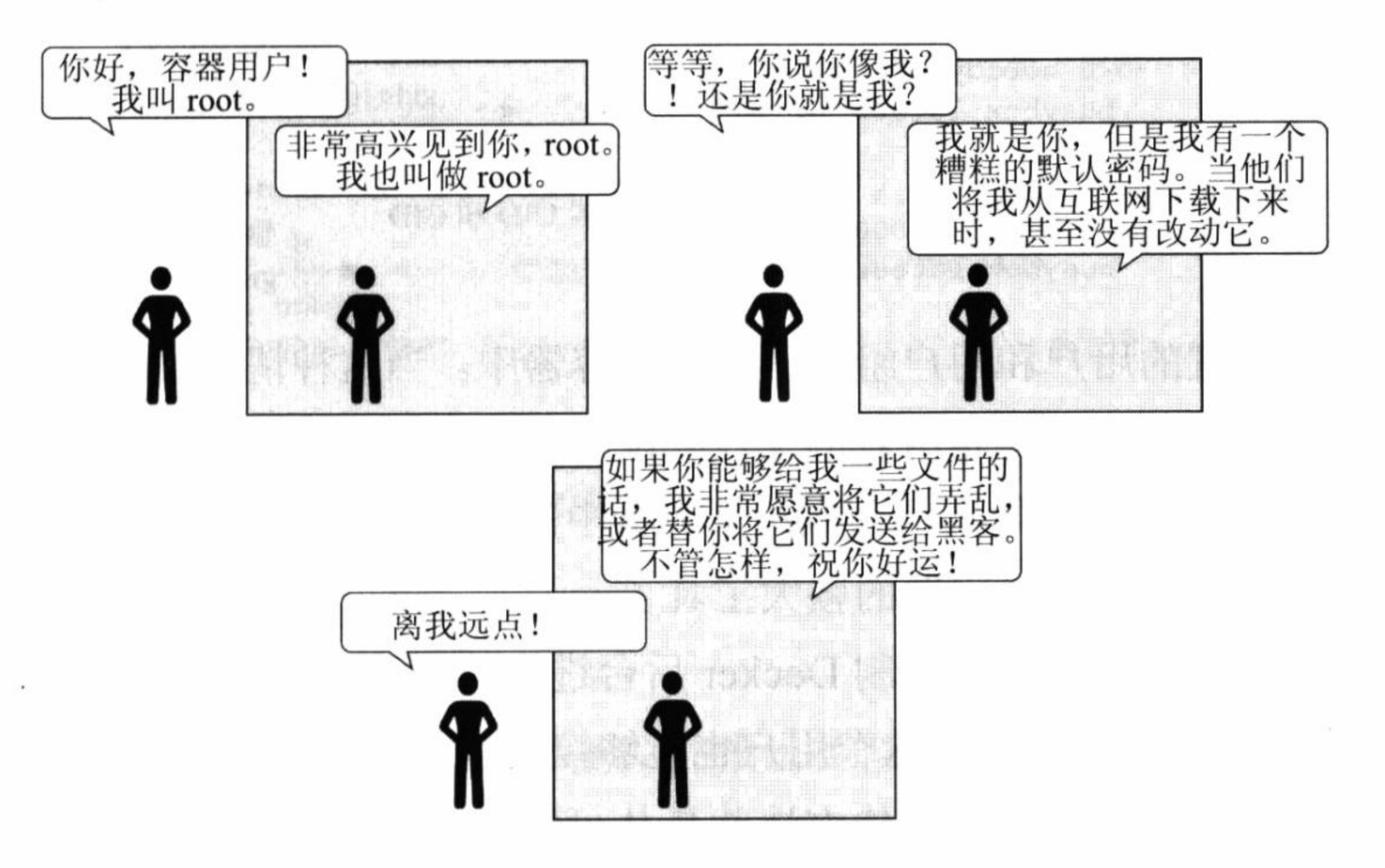


图 6-4 root vs root —— 一场安全戏剧

当你创建容器时，如果你设置了 run-as 用户，那么你就能够完全避免默认用户带来的问题。注意，指定的用户名必须在你使用的镜像中存在。不同的 Linux 发行版有不同的用户定义，并且一些镜像作者的个人因素导致了这个集合的扩大或缩小。你能够使用以下命令来获得镜像中的可用用户的列表：

```
docker run -rm busybox:latest awk -F: '$0=$1' /etc/passwd
```

我并不想太深入地讲解这个命令，你只需了解 Linux 的用户数据库就存储在/etc/passwd 文件中。这个命令会读取这个文件，并且获取其中的用户名列表。一旦你找到了你想要使用的用户，你就能够使用这个用户作为 run-as 用户来启动容器了。Docker 为 `docker run` 和 `docker create` 命令提供了`-u` 或`--user` 选项来设置这个 run-as 用户。下面这个命令将 run-as 用户设置为 nobody：

```
docker run --rm \
    --user nobody \
    busybox:latest id
```

设置 nobody 为 run-as 用户

输出：uid=99(nobody) gid=99(nobody)

这个命令使用了 nobody 用户。这个用户非常常见，常常在权限受限的场景中使用，比如运行应用。这仅仅是一个例子，你可以设置 run-as 用户为镜像中定义的任何用户，包括 root 用户。这仅仅是`-u` 或`--user` 选项功能的冰山一角。该选项的值能够是一对用户和用户组，还同时支持用户和用户组的名字或 ID。下面的例子使用 ID 替代名字：

```
docker run --rm \
    -u nobody:default \
    busybox:latest id

docker run --rm \
    -u 10000:20000 \
    busybox:latest id
```

将 run-as 用户设置为 robody，用户组设置为 default

输出：uid=99(nobody) gid=100(default)

设置 UID 和 GID

输出：uid=10000 gid=20000

第二个命令设置的用户和用户组都不存在于容器中。当这种情况发生时，ID 不会被解析成用户或者用户组名字，但是所有的文件权限会以该用户和用户组存在的情况来进行工作。改动 run-as 用户是否会导致问题取决于容器中的软件是如何配置的。否则，它将是一个能够很简单地解决文件权限问题的强大工具了。

你应该对你系统上那些能够控制 Docker 后台进程的用户非常小心。如果一个用户能够控制你的 Docker 后台进程，那么这个用户也能够控制你系统上的 root 用户。

保证运行时配置不出问题的最佳方法就是从可信赖的源拉取镜像或者自己构建镜像。想要做些恶意的行为是完全可能的，比如说将默认的非 root 用户转变为 root 用户，或者不经过认证就能开放对 root 用户的访问权。第 7 章简要地包含了这部分内容。现在，我更集中于另一个有趣的例子：

```
docker run -it --name escalation -u nobody busybox:latest \
    /bin/sh -c "whoami; su -c whoami"
```

输出："nobody" 和 "root"

这看起来太简单了吧！BusyBox 官方镜像没有为 root 用户（或其他用户）设置密码。Ubuntu 的官方镜像默认将 root 用户锁定；因此你不得不以 root 用户或者通过 SUID 提权（更多内容在第 7 章讨论）来启动容器，获得 root 权限。BusyBox 如此脆弱的认证方式导致在容器中，运行在任何用户下的进程能够自提权到 root。看到这些情况，你可能得到的教训是，学习 Ubuntu 或其他软件，开始设置密码或者禁止 root 用户。这意味着修改镜像，因此即便你不是一个软件开发者，你也能够从本书的第 2 部分受益。

6.3.3 用户和卷

既然你已经了解了容器中的用户如何和主机上的用户共享同一个用户 ID 空间，接下来你需要了解这两个用户如何互相影响。产生影响的主要理由就是卷（volume）中文件的文件权限问题。举个例子，如果你正在运行一个 Linux 终端，你能够直接使用下面的命令；否则，你需要使用 `boot2docker ssh` 命令从 Boot2Docker 虚拟机中获取一个 shell：

```
echo "e=mc^2" > garbage

chmod 600 garbage

sudo chown root:root garbage

docker run --rm -v "$(pwd)"/garbage:/test/garbage \
    -u nobody \
    ubuntu:latest cat /test/garbage

docker run --rm -v "$(pwd)"/garbage:/test/garbage \
    -u root ubuntu:latest cat /test/garbage
# Outputs: "e=mc^2"

# cleanup that garbage
sudo rm -f garbage
```

在主机上创建一个新文件

使这个文件只能被文件的所有者读取

将文件的所有者改为 root（假设你拥有 sudo 权限）

尝试用 nobody 用户读取文件

尝试用容器 root 用户读取文件

倒数第二个 `docker` 命令应该会失败，并报出类似于“Permission denied”的错误信息。但是最后一个 `docker` 命令应该会成功，并且会将输出的第一个命令输入到文件中的内容。这意味着卷（volume）中的文件的文件权限在容器中也是有效的。但这也意味着用户 ID 空间被共享了。主机上的 root 用户和容器中的 root 用户的 ID 都是 0。因此，尽管容器中 ID 为 65534 的 nobody 用户不能够访问主机上 root 用户创建的文件，但是容器中的 root 用户可以。

除非你想要主机的文件能够被容器访问，否则不要将文件以卷的形式挂载到容器上。

关于这个例子的好消息是，你已经了解到文件权限是如何在容器中生效的，因此能够解决更多实际的操作性问题。比如说，如何处理写入到卷中的一个日志文件？

最可取的方法是使用第 4 章描述的数据卷。但即便如此，你还是需要考虑文件权限和所有权问题。如果一个进程运行在 1001 用户下，并且它将日志文件写入到卷中，而另外一个容器中的进程想要以 1002 用户去访问这个文件，那么文件权限会阻止这个操作。

解决这个难题的一个方法是管理运行中用户的 ID。你可以提前修改镜像的用户 ID 为运行容器的用户 ID，或者你可以使用想要的用户和用户组去启动容器：

```
mkdir logFiles

sudo chown 2000:2000 logFiles

docker run --rm -v "$(pwd)"/logFiles:/logFiles \
    -u 2000:2000 ubuntu:latest \
    /bin/bash -c "echo This is important info > /logFiles/important.log"

docker run --rm -v "$(pwd)"/logFiles:/logFiles \
    -u 2000:2000 ubuntu:latest \
    /bin/bash -c "echo More info >> /logFiles/important.log"

sudo rm -r logFiles
```

设置目录的所有权为预期的用户和用户组

写入重要的 log 文件

设置 UID：GID 为 2000:2000

从另一个容器上添加 log 文件的内容

也设置 UID：GID 为 2000:2000

运行完这个例子后，你将会发现文件能够被写入到归用户 2000 所属的目录下。不仅如

此，任何容器的用户或用户组拥有这个目录的写入权限，那么它们就能够在该目录下写入文件或者直接修改同一个文件，当然前提是存在这种权限。这个方法适用于读取、写入和执行文件。

6.4 能力——操作系统功能的授权

Docker 能够支持容器中进程的功能授权。在 Linxu 中这些功能授权被称为能力（capabilities），不过作为对其他操作系统的原生支持，其他的后端实现也需要被提供。无论什么时候，一个进程尝试使用一个关卡系统调用，这个进程的能力就会被检测，看看是否存在需要的能力。如果存在，调用成功，否则调用失败。

当你创建了一个新容器，在默认情况下，Docker 删减了一部分的能力。这么做是为了更进一步地将正在运行的进程和操作系统的管理功能隔离开来。在阅读下面的能力列表时，你可能能够猜出它们被去除的原因。在写本书时，被去除的能力集合如下：

- SETPCAP —— 修改进程
- SYS_MODULE —— 插入或移除内核模块
- SYS_RAWIO —— 修改内核内存
- SYS_PACCT —— 配置进程计数
- SYS_NICE —— 修改进程优先级
- SYS_RESOURCE —— 覆盖资源限制
- SYS_TIME —— 修改系统时钟
- SYS_TTY_CONFIG —— 配置 TTY 设备
- AUDIT_WRITE —— 审计日志文件的写入
- AUDIT_CONTROL —— 配置审计子系统
- MAC_OVERRIDE —— 忽略内核 MAC 策略
- MAC_ADMIN —— 配置 MAC
- SYSLOG —— 修改内核打印行为
- NET_ADMIN —— 配置网络
- SYS_ADMIN —— 一系列管理功能的集合

对提供给 Docker 容器的能力集合进行删减是合理的，但未来你可能还需要对它进一步添加或删减。比如说，NET_RAW 能力非常危险。如果你想要比默认配置更谨慎小心，那

么你应该从能力列表中去除 NET_RAW。你可以通过使用 docker create 或 docker run 上的--cap-drop 选项来为容器去除能力。

```
docker run --rm -u nobody \
    ubuntu:latest \
    /bin/bash -c "capsh --print | grep net_raw"

docker run --rm -u nobody \
    --cap-drop net_raw \
    ubuntu:latest \
    /bin/bash -c "capsh --print | grep net_raw"
```

去除 NET_RAW 能力

在 Linux 的文档中，你经常能够看到能力的名字都是大写，并且前缀是 CAP_，但是如果这种格式的能力被用作能力管理的选项，这个前缀将不能正确工作。因此，请使用没有前缀且小写的名字。

类似于--cap-drop 选项，--cap-add 选项能够增添能力。如果你需要增添 SYS_ADMIN 能力，那么你可以使用下面的命令：

```
docker run --rm -u nobody \
    ubuntu:latest \
    /bin/bash -c "capsh --print | grep sys_admin"

docker run --rm -u nobody \
    --cap-add sys_admin \
    ubuntu:latest \
    /bin/bash -c "capsh --print | grep sys_admin"
```

SYS_ADMIN 能力没有被包含

添加 SYS_ADMIN 能力

类似于其他容器创建选项，--cap-drop 选项和--cap-add 选项都能够被设置多次来添加或删减多个能力。这些选项能够被用来构建这样一个容器，容器中的进程只能执行那些正确行为所需要的操作。

6.5 运行特权容器

当你需要在容器中运行系统管理任务时，你可以授予这些容器访问你计算机的特权。特权容器维持它们自己的文件系统和网络隔离，但却拥有对设备和共享内存的全部访问权，还具备全部的系统能力。你可以尝试几个有趣的任务，比如说在特权容器中运行 Docker。

大部分特权容器主要用于系统管理。比如说，root 文件系统只读的环境，或者在容器外安装容器被禁止，或者没有主机上 shell 的直接访问权。如果你想要运行一个程序来修改作业系统（比如说负载均衡），并且你有权在主机上运行容器，那么你就能够非常简单地在一个特权容器中运行这个程序。

如果你遇到了一个只需要特权容器就能解决的问题，那么你可以使用 docker run 或 docker create 命令的--privileged 选项来开启这种模式：

```
docker run --rm \
    --privileged \
    ubuntu:latest id
```

查看用户 ID

```
docker run --rm \
    --privileged \
    ubuntu:latest capsh -print
```

查看 Linux 能力

```
docker run --rm \
    --privileged \
    ubuntu:latest ls /dev
```

查看已被挂载的设备

```
docker run --rm \
    --privileged \
    ubuntu:latest ifconfig
```

检测网络配置

特权容器部分内容依旧被隔离。比如说，网络命名空间依旧受到影响。如果你需要破开这个命名空间，那么你需要同时使用`--net host`命令。

6.6 使用加强工具创建更健壮的容器

Docker 使用合理的默认配置和一系列工具集使之易被采用，并提升实际使用体验。但如果你使用了额外的工具，你就能加强这些默认容器。能够加强容器的工具包括 AppArmor 和 SELinux。如果你使用 LXC 容器供应商，你也能够提供自定义的 LXC 配置，并因此能够微调容器。如果你使用 LXC，你甚至能够使用一个名为 seccomp-bpf（通过系统调用过滤来保证计算机安全）的 Linux 特性。

关于这些工具，多本书已经出版了，分别讲述了这些工具各自的细微差别、好处和所需的技能集。毫无疑问，它们的功能值得这些努力。不同的 Linux 发行版对这些工具支持也不同，因此你需要花费一些时间在里面。但是一旦你调整好了主机配置，那么 Docker 的集成就会更加简单。

安全研究

信息安全领域非常的复杂且在无时无刻地变化。阅读 InfoSec 专家们的对话很容易使人感到不知所措。这些人常常是对开发和使用都有深入研究的人。你能够从这些对话中获取到的有用信息就是，想要平衡系统的安全和用户的需求是非常复杂的。

如果你对这个领域很陌生，你所能做的就是在跳入这个圈子前，先阅读相关的文章、论文、博客和书籍。这能够让你在思考其他观点之前，消化某一个观点并且获得一些深入理解的机会。当你已经能够形成你自己的见解和洞察时，那些这些对话会变得更加有

价值。

阅读一篇论文或者学习一个东西就找到更好解决问题的方法是非常困难的。无论你的情况如何，系统会从多个源头提升进步。因此你能做的最好的事就是逐个学习这些工具。不要害怕某些工具所需要的深度，这一切努力都是值得的，你最终将更好地理解你的系统。

Docker 并不是一个完美的解决方案。有些人会争论它甚至不是一个安全的工具。但是它所提供的改进比那些由于感知消耗而放弃任何隔离的替代品是要好得多。如果你能读到这里，可能你会愿意去跟进学习这些辅助的内容。

6.6.1 指定额外的安全选项

Docker 提供了一个在容器创建或者运行时指定 Linux 安全模块（LSM）的选项。LSM 是 Linux 采用的一个框架，用作操作系统和安全供应商之间的接口层。

AppArmor 和 SELinux 都是 LSM 的供应商。它们都提供了强制访问控制（MAC——定义访问规则的系统）来取代标准的 Linux 自主访问控制（文件所有者定义访问规则）。

可以通过 `docker run` 或 `docker create` 命令上的`--security-opt` 选项来设置这些内容。这个选项能够被设置多次。值可以是以下六种格式中的一种：

- 使用格式 `label:user:<USERNAME>` 来设置 SELinux 用户标签，其中`<USERNAME>`表示你想要使用的用户的名字。
- 使用格式 `label:role:<ROLE>`来设置 SELinux 角色标签，其中`<ROLE>`表示你想要赋予给容器中进程的角色的名字。
- 使用格式 `label:type:<TYPE>`来设置 SELinux 类型标签，其中`<TYPE>`表示容器中进程的类型的名字。
- 使用格式 `label:level:<LEVEL>`来设置 SELinux 级别标签，其中`<LEVEL>`表示容器中进程应该运行的级别。级别格式为低-高。如果只缩写为低级别，那么 SELinux 会将这个范围解析为单个级别。
- 使用格式 `label:disable` 禁止 SELnux 标签限制。
- 使用格式 `label:apparmor:<PROFILE>`来在容器上应用一个 AppArmor 配置，其中`<PROFILE>`就是配置文件的名字。

就像你看到的，SELinux 是一个标签系统。标签的集合被称为上下文（context），被应用于每个文件和系统对象上。一个类似的集合则被应用于每个用户和进程上。当一个进程

尝试与某一个文件或系统资源进行交互时，标签的集合就会根据被允许的规则进行评估。评估的结果决定了这个交互被允许还是被阻碍。

最后一个选项会设置一个 AppArmor 的配置文件。AppArmor 常常被用来替代 SELinux，因为它可以根据文件路径来工作而不是标签，其次还拥有一个训练模型，你能够基于观察到的应用行为来被动地构建配置文件。这些不同常常是 AppArmor 更容易被采用和维持的理由。

6.6.2 微调 LXC

Docker 最开始被创建是为了使用名为 Linux 容器（LXC）的软件。LXC 是一个容器运行时（container runtime）供应商的一个工具，它能够工作在 Linux 上创建命名空间和构建一个容器所需要的组件。

随着 Docker 的逐渐成熟，可移植性成为了关注的焦点，由此，一个新的、用来取代 LXC 的容器运行时，libcontainer 被构建了。Docker 默认自带 libcontainer，但 Docker 使用了一个接口层，因此用户可以改变容器执行（container execution）供应商。LXC 是一个比 libcontainer 更加成熟的库，且提供了很多与 Docker 目标背道而驰的额外特性。如果你可以并且想要使用 LXC，你可以改变容器供应商，并且利用这些额外的特性。在大力投资这个方向前，请你明白，某些额外的特性能够极大地降低容器的可移植性。

为了使用 LXC，你需要安装它，并且确保 Docker 后台进程程序启动时，LXC 驱动被允许。当启动 Docker 后台进程时，使用`--exec-driver=lxc`来使用 LXC。Docker 后台进程常常被作为系统服务来启动，因此请根据 www.docker.com 上的安装指导来找到你使用的发行版的操作细节。

一旦 Docker 被配置为 LXC，你可以使用 `docker run` 或 `docker create` 命令的`--lxc-conf`选项来设置 LXC 的配置：

```
docker run -d \
    --lxc-conf="lxc.cgroup.cpuset.cpus=0,1" \
    --name ch6_stresser dockerinaction/ch6_stresser
```

通过 LXC 将容器限制在两个 CPU 核上

```
docker run -it --rm dockerinaction/ch6_htop

docker rm -vf ch6_stresser
```

就和本章之前的一个类似例子一样，你可以按下【Q】键来退出 htop 程序。

如果你决定使用 LXC 并且设置了 LXC 特有的配置选项，那么 Docker 将不会意识到这

些配置。某些配置会和标准容器改动产生冲突。因此，你应该总是小心地验证那些会对容器产生实际影响的配置。

6.7 因地制宜地构建容器

容器是一个横切关注点（cross-cutting concern）。人们使用容器有各种各样的理由和方式，这远比我曾经罗列的多得多。因此，当你使用 Docker 构建容器来实现你的目的时，花一些时间保证容器能够以契合你所运行的软件的方式来运行是非常重要的。

最安全的策略就是以隔离度最高的容器作为起始点，当你想要去除某些限制时，请为之找到合适的理由。在实际中，人们更趋向于临时反应而不是提前预防。因此，我认为 Docker 的默认容器配置击中了要点。这为用户提供了一份合理的、不会阻碍用户产品性的默认配置。

在默认情况下，Docker 容器隔离程度并不是最高。Docker 并不要求你加强这些默认配置。如果你想这么做的话，它反而会让你在实际生产中做出愚蠢的事情。这使得 Docker 更像一个工具而不是一个负担，更像某些人们想用的东西而不是他们不得不用的东西。至于那些不愿在实际生产中做出愚蠢事情的人，Docker 提供了一个简单的接口让他们能够加强容器的隔离度。

6.7.1 应用

应用是我们使用计算机的全部原因。大多数应用都是其他人编写的程序，这些程序需要和具有潜在恶意的数据进行交互。想想你的 web 浏览器吧。

web 浏览器是一类几乎安装在每一台计算机上的应用。它与网页、图像、脚本、嵌入式视频、Flash、Java 应用等进行交互。你当然没有创建这些内容，并且大多数人并不是 web 浏览器项目的参与者。你如何能够信任你的 web 浏览器能够正确地处理所有这些内容呢？

有些漫不经心的读者可能会忽略这个问题。毕竟，即便最坏的情况发生了又会怎样呢？好吧，如果一个攻击者控制了你的浏览器（或其他应用），那么他们会获得那个应用的所有能力和运行应用的用户的权限。他们能够弄乱你的计算机，删除你的文件，安装其他的恶意软件，甚至攻击你的其他计算机。因此，忽略这个问题并不太好。问题依旧存在：当这成为你需要承担的风险，你如何保护你自己？

最佳的办法就是隔离这些风险。第一，确保运行应用的用户具有有限的权限。这种情况下，即便发生了问题，攻击者也不能够改变你计算机上的文件。第二，限制浏览器的系

统能力。这种情况下，你确保了你的系统配置是安全的。第三，限制应用的 CPU 和内存资源。资源限制能够为系统保留一些资源，使系统还能够响应。最后，指定应用能够访问的设备的白名单是一个好主意。这能够防止摄像头、USB 等类似设备被窥探。

6.7.2 高层的系统服务

高层的系统服务和应用有些不同。它们不是操作系统的一部分，但是你的计算机需要确保它们被启动并且保持运行状态。这些工具常常处于操作系统的外层，而伴随于应用，但它们又常常需要对操作系统的访问特权来正确地运转。它们为用户和其他软件提供了重要的功能。如 `cron`，`syslogd`，`dbus`，`sshd`，`docker`。

如果你对这些工具不熟悉（希望不是所有都不熟悉），没有关系。它们的工作类似于维持系统日志，运行安排好的命令，在网络上获得一个安全的 shell，还有 `docker` 负责管理容器。

尽管以 root 用户运行这些服务非常常见，但实际上，很少部分需要全部的特权。使用能力（capability）让它们的权限只包含实际需要的特权。

6.7.3 低层的系统服务

低层的系统服务控制的东西类似于设备或系统网络栈。它们需要对系统组件具有访问特权（如防火墙软件需要对网络栈有管理权）。

很少看到这类服务运行在容器中。文件系统管理、设备管理和网络管理等任务都是主机关注的核心。大多数运行在容器中的软件被期望具有可移植性。因此与计算器有关的任务很难适用于通用的容器环境。

最好的例外就是短期的配置容器。比如说，在这样一个环境中，所有的部署都和 Docker 镜像和容器有关，你想要像提交软件一样提交网络栈的改动。在这种情况下，你可能会提交一个带有配置文件的镜像到主机上，然后使用一个特权容器来做改动。在这种情况下，风险已经被降低，因为你授权配置文件能够被提交，这个容器不会长久运行，并且类似这种变动很容易被监控审计。

6.8 小结

本章介绍了 Linux 提供的隔离特性，并讨论了 Docker 如何使用这些特性来构造可配置的容器。有了这些知识，你就能够自定义容器隔离度，使得 Docker 能够适用于任何用户场

景。本章包含了以下内容：

- Docker 使用 cgroups 让用户能够设置内存限制、CPU 权重、CPU 核限制，还有设备访问限制。
- 每个 Docker 容器都有各自的 IPC 命名空间，它们能够被其他容器或主机共享，以此建立基于共享内存的高速通信。
- Docker 还不支持 USR 命名空间，因此容器中的用户和用户组 ID 和主机上具有相同 ID 的用户和用户组是相等的。
- 你可以并且应该使用 `docker run` 和 `docker create` 命令的`-u` 选项使得容器以非 root 用户运行。
- 尽可能避免容器以特权模式运行。
- Linux 能力提供了操作系统特性的授权功能。Docker 去除某些能力是为了提供合理的默认隔离配置。
- 使用`--cap-add` 和`--cap-drop` 选项来赋予或去除容器的能力。
- Docker 提供了配置选项来集成加强隔离技术，比如 SELinux 和 AppArmor。这些都是非常强大的工具，任何认真严肃的 Docker 采用者都应该研究它们。

第 2 部分

镜像发布：如何打包软件

尽管这可能不是一个频繁发生的事情，但一个 Docker 使用者不可避免地需要创建一个镜像。比如，你需要的软件没有被打包在一个镜像中，或者你需要的一个特性在镜像中没有被允许，这些情况你都需要自建镜像。本部分会帮助你理解如何初始化、自定义和特殊化你想要部署或共享的镜像。

第 7 章　在镜像中打包软件

本章介绍

- 手动的镜像构建和练习
- 从打包的角度看待镜像
- 扁平镜像
- 镜像版本控制的最佳实践

本章的目的是为了帮助你理解镜像设计的关注点，帮助你学习构建镜像的工具，帮助你探索高级镜像模式。你会通过亲手实现一个现实世界的例子来学到这些知识。在开始本章之前，你应该牢牢掌握本书第一部分讲述的概念。

创建容器有两种方法，要么手工修改现有容器中的镜像，要么定义、执行一个名为 Dockerfile 的自动化构建脚本。本章关注于手工修改镜像的过程、镜像操作的基本机制、还有最终人工产生的镜像。Dockerfile 和构建自动化内容包含在第 8 章中。

7.1　从容器构建镜像

如果你对如何使用容器已经非常熟悉了，那么学习如何构建容器会变得非常简单。记住，联合文件系统（UFS）挂载提供了容器的文件系统。任何对容器内文件系统的改动都会被写入到新的文件层中，这个文件层归创建它的容器所有。

在你接触真实的软件之前，下一节详细地叙述了一个 Hello World 程序的经典工作流。

7.1.1 打包 Hello World

从一个容器构建一个镜像的基础工作流包含三部分：

第一，你需要从一个已存在的镜像创建一个容器。至于选什么镜像，这需要根据你最终想要将哪些东西包含到新镜像中，以及需要哪些修改镜像的工具来决定。

第二，修改这个容器的文件系统。这些改动会被保存在容器的联合文件系统的新文件层。在本章后面的内容中，我们会再次回顾镜像、文件层（layer）、还有仓库（repository）之间的关系。

第三，一旦改动完成，那么就要将这些改动提交（commit）。一旦改动被提交，你就能够从新镜像创建新的容器了。如图 7-1 所示描绘了这个工作流。

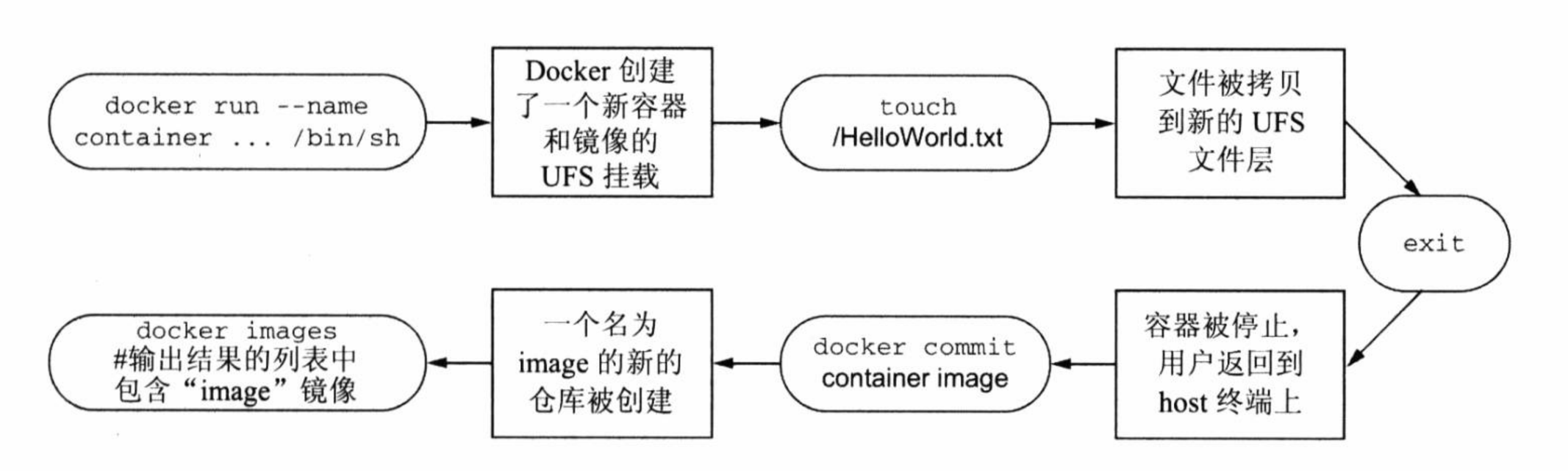


图 7-1 从容器中构建镜像

在心里记住这些步骤，跟随下面的命令来创建一个名为 hw_image 的新镜像：

```
docker run --name hw_container \
    ubuntu:latest \
    touch /HelloWorld

docker commit hw_container hw_image

docker rm -vf hw_container

docker run --rm \
    hw_image \
    ls -l /HelloWorld
```

修改容器中的文件

将改动提交到新镜像中

去除被改动的文件

检测容器中的文件

这似乎显得过于简单，但你应该明白，随着构建的镜像越来越复杂，这个过程会变得有些微妙差别，但是基础的步骤总是一样的。

现在你已经对工作流有了基本了解，你应该尝试创建带有真实世界软件的新镜像。下面，你将会打包一个名为 Git 的程序。

7.1.2 打包 Git

Git 是一个非常流行的分布式版本控制工具。关于 Git 的书已经出版了很多，如果你对它不熟悉，我建议你花一些时间学习如何使用 Git。但现在，你只需要知道这是一个程序，你要将这个程序安装到一个 Ubuntu 镜像中。

你需要从一个合适的基础镜像创建一个容器：

```
docker run -it -name image-dev ubuntu:latest /bin/bash
```

这条命令会启动一个运行 bash shell 的新容器。基于这个 shell，你就能够输入命令来自定义你的容器了。Ubuntu 提供了一个软件管理工具，叫做 `apt-get`。它能够获取你想要在镜像中打包的软件，记住它，这迟早是用的上的。现在你有了一个交互 shell，接下来，你需要使用下面的命令在容器中安装 Git：

```
apt-get -y install git
```

这个命令会告诉 APT 在容器的文件系统上下载并且安装 Git 和其所有的依赖。安装完成后，你可以通过以下命令来测试 Git 是否安装：

```
git version
#输出结果如下：
#git version 1.9.1
```

类似 apt-get 这种打包工具替代了你需要手动完成的所有事情，使得安装和卸载软件更加容易。但它们并没有为软件提供隔离，并且依赖冲突常常发生。不过，可以肯定的是，你安装在容器外的其他软件并不会影响到当前容器中 Git 的版本。

既然 Git 已经被安装在你的 Ubuntu 容器上，你可以简单地使用下面命令退出容器：

```
exit
```

现在，容器会被停止，但依然保留在你的计算机上。Git 被安装在 ubuntu:latest 镜像的最上面的新文件层中，你如何能够准确地知道哪些东西被改动了？当你在打包软件时，审查容器中被改动的文件列表是非常有用的，Docker 提供了一个命令来达到这个目的。

7.1.3 审查文件系统的改动

Docker 提供了一个命令，它能够显示容器中文件系统的所有改动。这些改动包括添加、修改、删除文件和目录。你可以通过 `diff` 子命令来查看你安装 Git 时的改动：

```
docker diff image-dev
```

输出结果是一个非常长的文件改动列表

以 A 开头的行表示文件被添加。以 C 开头表示修改，以 D 开头表示删除。安装 Git 会包含多个改动，不利于区分，因此，我们使用一些更加特殊的例子会更好理解些：

```
docker run --name tweak-a busybox:latest touch /HelloWorld
docker diff tweak-a
# Output:
#    A /HelloWorld
```

添加新文件到 busybox 镜像中

```
docker run --name tweak-d busybox:latest rm /bin/vi
docker diff tweak-d
# Output:
#    C /bin
#    D /bin/vi
```

从 busyboc 镜像中移除现有文件

```
docker run --name tweak-c busybox:latest touch /bin/vi
docker diff tweak-c
# Output:
#    C /bin
#    C /bin/busybox
```

修改 busybox 镜像中现有文件

记得清理容器：

```
docker rm -vf tweak-a
docker rm -vf tweak-d
docker rm -vf tweak-c
```

现在你已经看到你对文件系统的改动了，是时候提交这些改动到新镜像中了。与之前大多数情况一样，存在一个命令来完成多个操作。

7.1.4 Commit——创建新镜像

你可以使用 docker commit 命令从被修改的容器上创建新镜像。最好能够使用-a 选项为新镜像指定作者的信息。同时你也应该总是使用-m 选项，它能够设置关于提交的信息。前面你已经在 image-dev 容器中安装了 Git，可以使用下面的命令从 image-dev 容器创建并标记一个名为 ubuntu-git 的新镜像：

```
docker commit -a "@dockerinaction" -m "Added git" image-dev ubuntu-git
# 输出内容是一个新的、唯一的镜像 ID:
# bbf1d5d430cdf541a72ad74dfa54f6faec41d2c1e4200778e9d4302035e5d143
```

一旦你提交了这个镜像，它就会显示在你计算机的已安装镜像列表中。运行 docker images 命令应该会包含下面这一行：

```
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE ubuntu-git latest bbf1d5d430cd
5 seconds ago 226 MB
```

可以从这个新镜像中创建一个容器，并且在其中测试 Git 来确保新镜像正确工作：

```
docker run -rm ubuntu-git git version
```

现在你已经基于 Ubuntu 镜像创建了一个安装有 Git 的新镜像。这是一个非常好的开端，但如果你省略了命令覆盖（command override），会发生什么事情呢？尝试运行命令去找到答案：

```
docker run -rm ubuntu-git
```

当你运行这个命令时似乎什么都不会发生。这是因为你启动原始容器时附带的命令会被提交到新镜像中，而之前你启动创建新镜像的容器时附带的命令是/bin/bash。因此，当你使用这个默认命令从新镜像中创建一个容器时，它会启动一个 shell 并且立马停止它。显然，这并不是一个非常有用的默认命令。

我对 ubuntu-git 镜像的用户希望每次手动启动 Git 表示怀疑。在镜像上设置一个指向 git 的入口点（entrypoint）是更好的做法。一个入口点就是一个程序，它会在容器启动时被执行。如果入口点没有被设置，那么默认的命令会被直接执行。如果入口点被设置，那么默认的命令和它的参数就会作为参数传递给入口点。

为了设置入口点，你需要使用--entrypoint 选项重新创建一个容器，并且从这个容器创建新的镜像：

```
docker run --name cmd-git --entrypoint git ubuntu-git      ← 显示标准的 git 帮助命令，然后退出

docker commit -m "Set CMD git" \
    -a "@dockerinaction" cmd-git ubuntu-git                ← 提交新镜像并保持名字不变

清除 → docker rm -vf cmd-git

docker run --name cmd-git ubuntu-git version               ← 测试
```

现在入口点被设置为 Git，用户再也不需要在最后输入 git 命令了。这看起来似乎是一个无关紧要、边缘的节约，但人们使用的很多工具并不是都像这个例子一样那么简洁。设置入口点仅仅是使得镜像更加易于使用和易于集成进项目的一部分。

7.1.5 可配置的镜像属性

当你使用 docker commit 命令，你就向镜像提交了一个新的文件层。但并不是只有文件系统快照被提交。每一层都包含描述执行上下文（execution context）的元数据。其中包括一些容器启动时设置的参数，下面罗列的内容会被记录进新镜像：

- 所有的环境变量
- 工作目录

- 开放端口集合
- 所有的卷定义
- 容器入口点
- 命令和参数

如果这些值没有被明确地指定，那么这些值会从原始镜像继承。本书的第一部分逐一讲述了这些内容，因此我不再赘述。但测试两个详细的例子可能对你有帮助。第一个，一个容器明确地指定了两个环境变量：

```
docker run --name rich-image-example \
    -e ENV_EXAMPLE1=Rich -e ENV_EXAMPLE2=Example \
    busybox:latest

docker commit rich-image-example rie

docker run --rm rie \
    /bin/sh -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"
```

创建环境变量

提交镜像

输出结果：Rich Example

第二个，在第一个例子的容器上，明确地指定了入口点和命令：

```
docker run --name rich-image-example-2 \
    --entrypoint "/bin/sh" \
    rie \
    -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"

docker commit rich-image-example-2 rie

docker run --rm rie
```

设置默认入口点

设置默认命令

提交镜像

不同的命令有相同的输出结果

这两个例子在 BusyBox 镜像上构建了两层额外的文件层。两个例子中的文件都没改变，但是行为却改变了，这是因为上下文元数据被改变了。这些改动包括第一层上的两个新环境变量。这些环境变量很明显被第二层继承了，而第二层又设置了入口点和默认命令来显示这些变量的值。尽管最后一个命令在使用最终的镜像创建容器时，没有指定任何替换的行为，但可以很清楚地看到，之前定义的行为被继承了。

现在你已经理解了如何改动一个镜像，接下来花些时间深入到镜像和层的运行机制中。这么做会帮助你在实际情况中构建高质量的镜像。

7.2 深入 Docker 镜像和层

到目前为止，你已经构建了一些镜像。在这些例子中，你首先从镜像创建容器，比如说 ubuntu:latest 或 busybox:latest。然后你对这个容器的文件系统或者上下文进行改动。最后，当你使用 `docker commit` 命令创建新的镜像时，所有的一切仿佛才开始工作。理解容器文件系统如何工作，理解 `docker commit` 实际做了什么，都能够帮助你成为一个更好的镜像制作者。本节深入这个主题，并且证明这对制作者的影响。

7.2.1 深入联合文件系统

理解联合文件系统的细节对于镜像制作者非常重要，理由有两个：

- 制作者需要知道添加、修改、删除文件会给新镜像带来什么影响。
- 作者需要对层之间的关系，还有层和镜像、仓库、标签（tag）的关系有一个扎实了解。

让我们先从一个简单的例子开始。假设你想要对一个已存在的镜像修改，这个镜像是 ubuntu:latest，你想要在根目录上添加一个名为 mychange 的文件。你应该使用以下命令来达到目的：

```
docker run -name mod_ubuntu ubuntu:latest touch /mychange
```

生成的容器（叫做 mod_ubuntu）最后会停止，但是改动已经被写入到它的文件系统了。如第 3 和第 4 章讨论的那样，根文件系统由镜像提供，这个文件系统由联合文件系统实现。

联合文件系统由多个层组成。每当对联合文件系统改动一次，改动会被记录到一个新的层中，这个新层放置于所有层的最上面。容器（和用户）访问文件系统所看到的，就是所有这些层的“联合”，或者说是自上而下的观察角度。如图 7-2 所示，从两个角度描绘了这个例子。

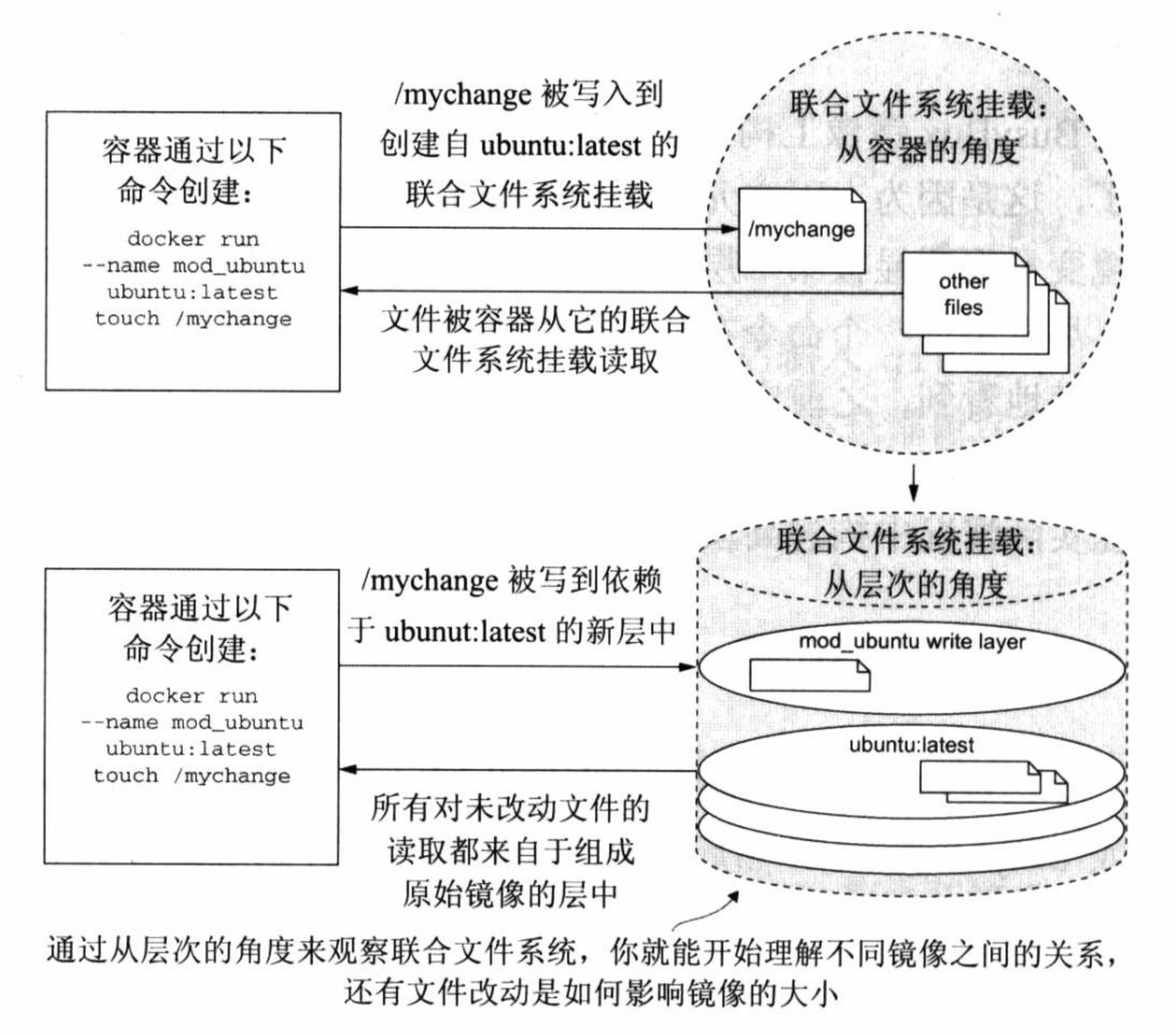


图 7-2 从两个角度来描述一个简单的文件写入例子时在联合文件系统上的表现

当你从联合文件系统读取一个文件时，系统会从存在该文件的、最上面的一层中读取。如果文件没有在最顶层被创建或者改动，那么读取操作就会沿着层不断向下找，直到找到存在这个文件的层。这个过程描绘在如图 7-3 所示中。

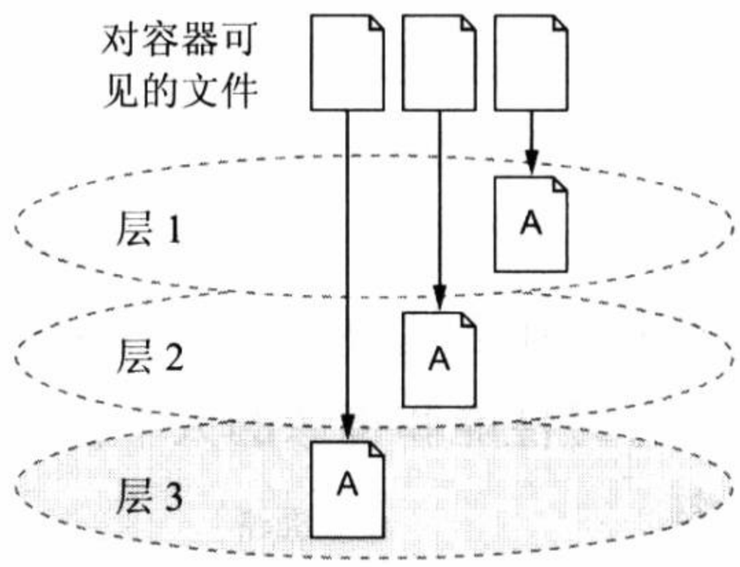


图 7-3 读取分布在不同层的文件——对容器可见的文件

以上展示的层功能都被联合文件系统隐藏起来了。运行在容器上的软件并不需要采取任何具体的操作来利用这些特性。添加文件仅覆盖了三种文件系统写入类型的一种，另外两种分别是修改和删除。

和添加文件类似，文件修改和删除也通过修改最顶层来工作的。当一个文件被删除，一个删除记录就被写入到最顶层，它遮挡了底层该文件的所有版本。当一个文件被修改，修改也被写入到最顶层，它也同样遮挡了底层所有该文件的版本。对容器文件系统的改动可以通过之前已经使用过的命令来获得：

```
docker diff mod_ubuntu
```

这个命令会输出以下结果：

```
A /mychange
```

A 在这里表示文件被添加。运行下面两个命令，能够看到删除文件时如何被记录的：

```
docker run -name mod_busybox_delete busybox:latest rm /etc/profile
docker diff mod_busybox_delete
```

输出结果如下：

```
C /etc D /etc/profile
```

D 表示文件被删除。该文件的父目录也被包含，C 表示该目录被修改。

接下来的两个命令能够看到文件修改后的效果：

```
docker run -name mod_busybox_change busybox:latest touch /etc/profile docker
diff mod_busybox_change
```

结果如下：

```
C /etc
C /etc/profile
```

同样，C 意味着修改，其中一个是文件，另一个是该文件所在的目录。如果嵌套了 5

层的文件被修改，那么输出结果对目录树的每一层都会有对应的一行。文件修改机制是理解联合文件系统最重要的东西。

大多数的联合文件系统采用了名为写时复制（copy-on-write）的技术，如果你将它理解为改动时复制（copu-on-change）会更好理解。当只读层（read-only layer）上一个文件被修改了，那么这整个文件会在改动发生之前被复制到最上面的可写层（writable layer）。这对运行时性能和镜像的大小会有负面影响。7.2.3 节包含了这种方法会如何影响镜像设计的内容。

下面将花费一些时间，通过测试更加通用的场景（描绘在如图 7-4 所示中）来巩固你对联合文件系统的理解。在图 7-4 中，文件会在三层的镜像上被多次添加、修改、删除。

知道文件系统修改如何被记录，你就能开始理解当你使用 `docker commit` 命令创建新镜像时会发生什么了。

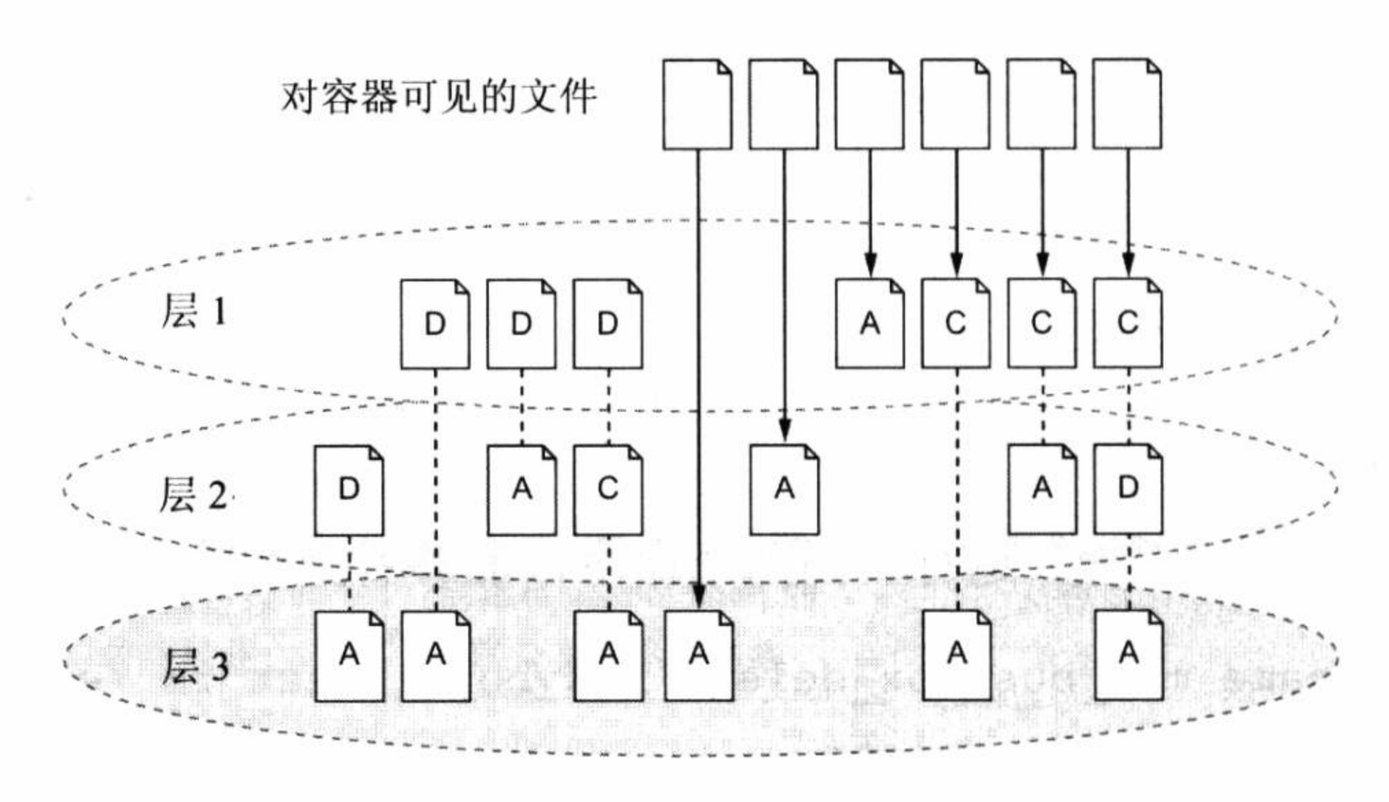


图 7-4 在具有三层的镜像上进行各式各样的文件添加、修改、删除组合操作

7.2.2 重新认识镜像、层、仓库和标签

你已经用 docker commit 命令创建了一个新镜像，并且理解了这个命令相当于提交了最上层的改动。但到目前为止，我们还没有定义过提交（`commit`）。

记住，联合文件系统由多个层以栈的形式组成，并且新的层会被添加到栈的最上方。这些层会被独立存储，每层包含这一层的改动信息和元数据。当你向容器的文件系统提交容器的改动时，你也是以同样的方式保存了最顶层的一个副本。

当你提交一层时，一个新的 ID 会为这一层创建，所有文件改动的副本都会被保存。准确地说，这如何发生取决于你计算机使用的存储引擎。对于你来说，理解通用的方法比理

解其中的细节更加重要。新层的元数据包含了之前生成的 ID，还有更低一层的层 ID（父层），还有新层被创建时的执行上下文（execution context）。层 ID 和元数据形成了一个图，Docker 和联合文件系统（UFS）使用这个图来构造镜像。

如图 7-5 所示，一个镜像由多个层以栈的形式组成，首先给出一个顶层作为起始点，然后根据每层元数据中的父层 ID 将多个层自上而下地连接起来。

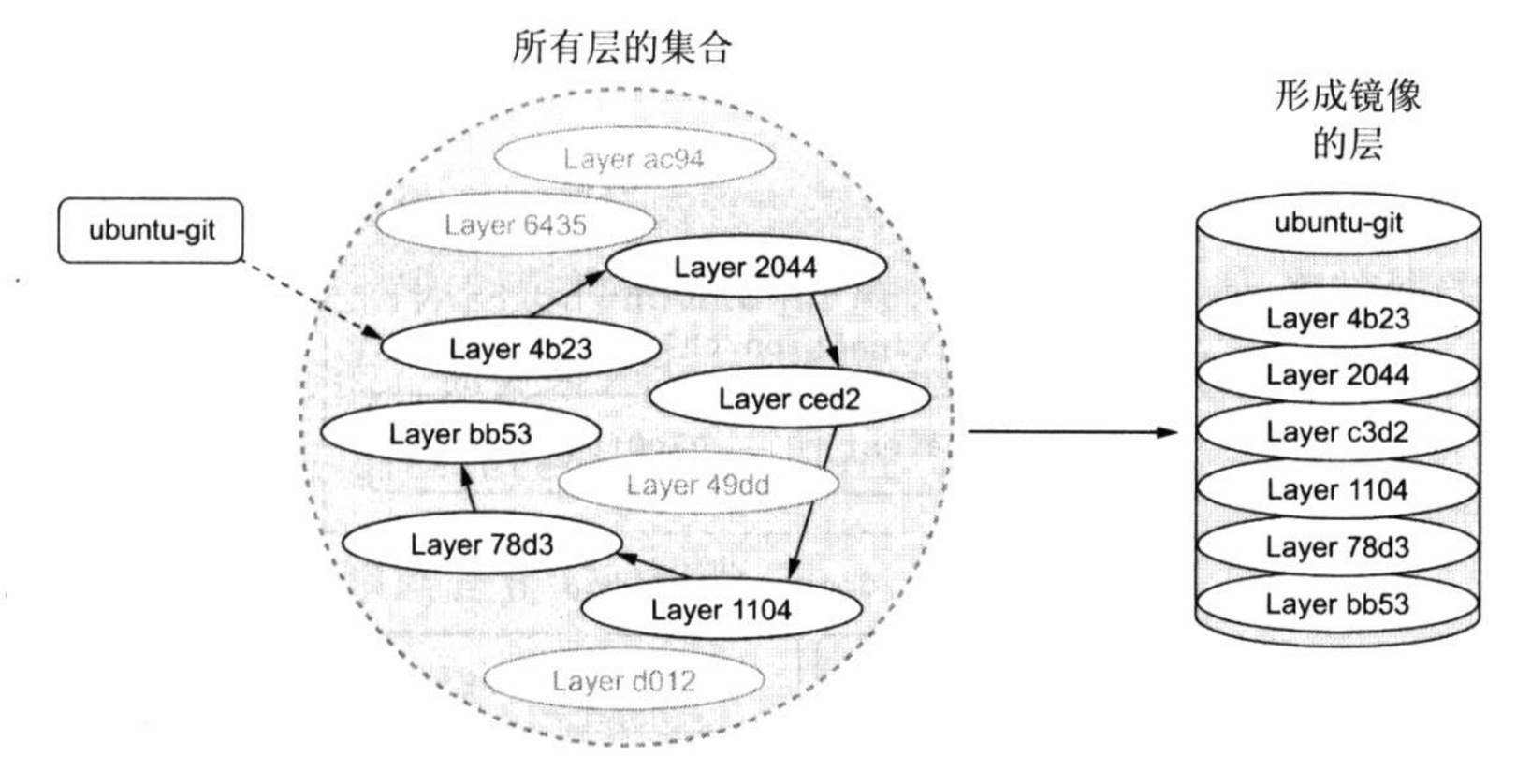


图 7-5 从顶层开始，遍历所有的父层，然后这些层组成的集合形成一个镜像

从某些起始层开始，遍历寻找到它们的依赖层，然后这些层以栈的形式构造成镜像。遍历从栈顶层开始。这意味着这个层的 ID 也就是它和其依赖层所形成的镜像的 ID。花费一些时间来实际体验下这个效果，你可以提交之前你创建的 mod_ubuntu 容器来看看效果：

```
docker commit mod_ubuntu
```

输出结果会包含一个新镜像的 ID：

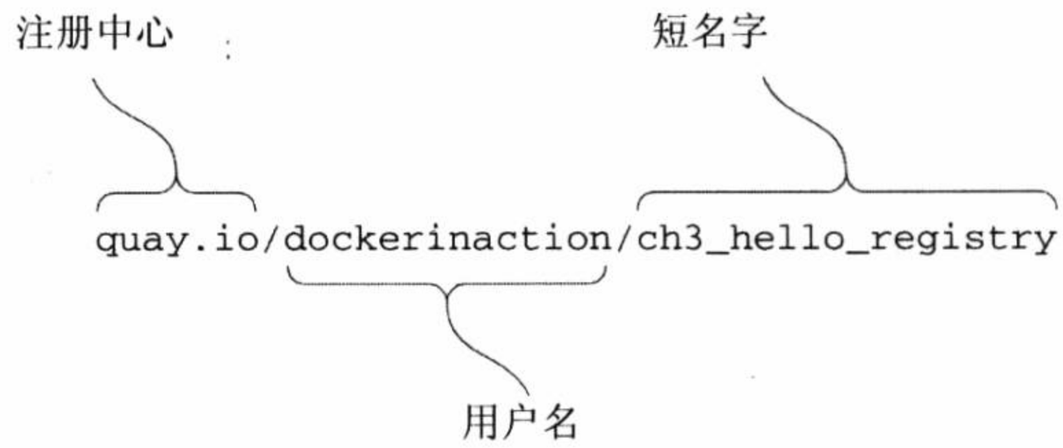
```
6528255cda2f9774a11a6b82be46c86a66b5feff913f5bb3e09536a54b08234d
```

你可以使用这个镜像 ID 来从这个镜像创建一个新容器。和容器类似，层 ID 也由大量十六进制数字组成，不利于用户直接使用。因此，Docker 提供了仓库（repository）的概念。

在第 3 章中，仓库被简单地定义为镜像的命名桶。更具体些，仓库的格式为位置/名字，它指向了特定层 ID 的集合。每个仓库至少包含一个标签，该标签指向一个层 ID，这也就形成了镜像的 ID。让我们重新回顾下第 3 章用到的例子：

这个仓库存储于 quay.io 的注册中心，它指定了用户名（dockerinaction）和一个唯一的短名字（ch3_hello_registry）。拉取这个仓库将会把该仓库中每个标签定义的镜像都拉取下来。在这个例子中，该仓库只有一个标签——latest。如图 7-6 所示，这个标签指向了短

ID 为 07c0f84777ef 的层。



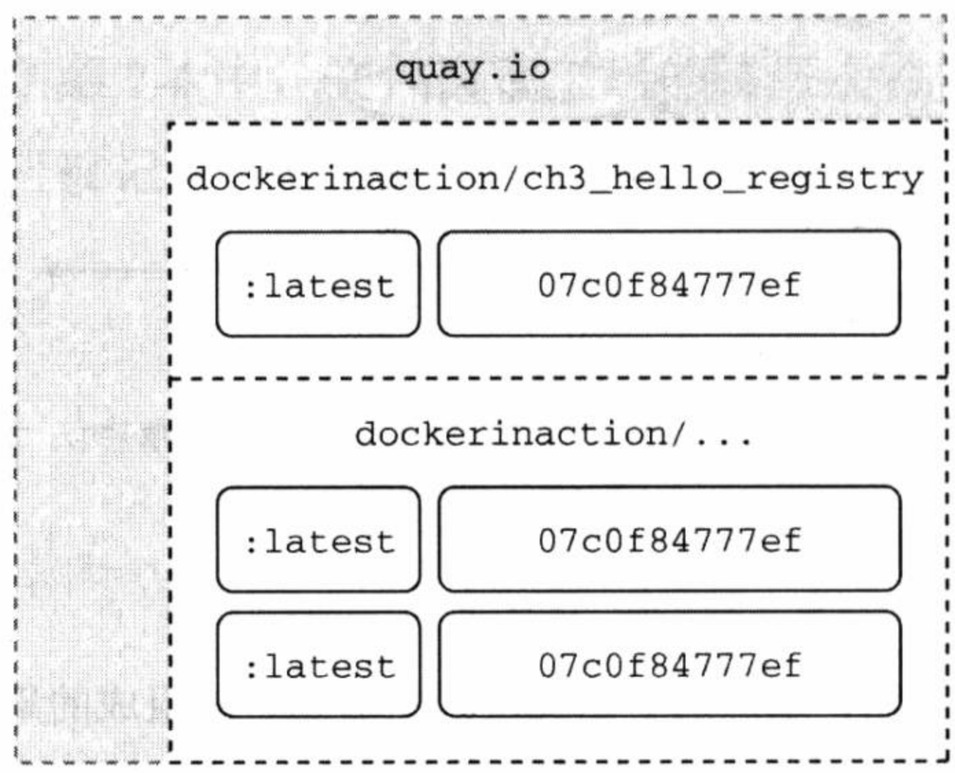


图 7-6 对仓库的可视化展示

仓库和标签通过 docker tag、docker commit、docker build 命令创建。下面我们再次利用 mod_ubuntu 容器，将它放入一个仓库中，并且带有一个标签：

```
docker commit mod_ubuntu myuser/myfirstrepo:mytag
# 输出结果：
# 82ec7d2c57952bf57ab1ffdf40d5374c4c68228e3e923633734e68a11f9a2b59
```

可以看到，一个层的新副本被创建，因此生成的新 ID 和之前的不一样了。有了这个新的、友好的名字，从新镜像创建容器会变得更加简单。如果你想要复制一个镜像，那么你只需要从现有的镜像创建一个新的标签或仓库。你可以使用 docker tag 命令来完成。每个仓库都拥有一个默认的 latest 标签。如果标签被省略，那么这个默认标签会被使用：

```
docker tag myuser/myfirstrepo:mytag myuser/mod_ubuntu
```

到此为止，你应该对基础的 UFS 基本原理和 Docker 如何创建和管理层、镜像、仓库有了一个深入理解。记住这些，下面让我们来思考它们可能怎样影响镜像设计。

创建容器会创建一个可写层，所有在可写层下面的层都是不可变的，这意味着它们永远不会被改变。这个特性使得共享镜像访问权变得更加可行，而不是为每个容器创建独立的副本。它也使得每层变得高度可复用。另一方面，当你对镜像进行改动时，你仅仅需要添加一个新的层，老的层永远不需要被改动。镜像不可避免地需要被改动，你需要意识到任何镜像的限制，并且将改动如何影响镜像大小牢记在心。

7.2.3 镜像体积和层数限制

如果像大多数人管理文件系统那样去管理镜像，那么 Docker 镜像会立马变得不可用。举个例子，假设你想要对之前你创建的 ubuntu-git 镜像生成新的版本，直接修改那个 ubuntu-git 镜像似乎是非常自然的方法。你会为老版本分配新标签，为新版本分配 latest 标签：

```
docker tag ubuntu-git:latest ubuntu-git:1.9
```

创建新的标签：1.9

构建新镜像的第一件事就是将 Git 卸载：

```
docker run --name image-dev2 \
    --entrypoint /bin/bash \
    ubuntu-git:latest -c "apt-get remove -y git"

docker commit image-dev2 ubuntu-git:removed

docker tag -f ubuntu-git:removed ubuntu-git:latest

docker images
```

执行 bash 命令

移除 Git

提交镜像

重新分配一个 latest 标签

测试镜像大小

输出的镜像列表和大小如下所示：

```
REPOSITORY    TAG         IMAGE ID        CREATED            VIRTUAL SIZE
ubuntu-git    latest      826c66145a59    10 seconds ago     226.6 MB
ubuntu-git    removed     826c66145a59    10 seconds ago     226.6 MB
ubuntu-git    1.9         3e356394c14e    41 hours ago       226 MB
...
```

注意，你卸载了 Git，实际上镜像的大小却增大了。尽管你能够通过 docker diff 命令观察到具体的改动，但你应该能够很快意识到增长的原因在于联合文件系统。

记住，联合文件系统实际上会在最顶层添加一个文件来标记一个文件被删除。原始的文件和任何文件副本依旧保留在镜像的其他层中。减少镜像大小对于使用镜像的用户和系统都是非常重要的。如果你能够避免下载时间过长，或通过智能镜像创建来提高硬盘使用率，那么你的用户就能够受益。你还应该意识到这个方法存在另一种风险。

联合文件系统可能有一个层数量的限制。这个限制取决于文件系统，但 42 层限制在使用 AUFS 系统的计算机上是非常常见的。这个数字看起来很大，但并不是不能够达到的。你

可以通过使用 docker history 命令来查看一个镜像的所有层。输出内容包含以下内容：

- 缩写的层 ID
- 层的年龄
- 创建容器时的初始命令
- 这一层的全部文件大小

通过查看 ubuntu-git:removed 镜像的历史，你能够看到有三层已经被添加到原始 ubuntu:latest 镜像的最上方：

```
docker history ubuntu-git:removed
```

输出结果如下：

```
IMAGE          CREATED          CREATED BY                      SIZE
826c66145a59   24 minutes ago   /bin/bash -c apt-get remove     662 kB
3e356394c14e   42 hours ago     git                             0 B
bbf1d5d430cd   42 hours ago     /bin/bash                       37.68 MB
b39b81afc8ca   3 months ago     /bin/sh -c #(nop) CMD [/bin     0 B
615c102e2290   3 months ago     /bin/sh -c sed -i 's/^#\s*\     1.895 kB
837339b91538   3 months ago     /bin/sh -c echo '#!/bin/sh'     194.5 kB
53f858aaaf03   3 months ago     /bin/sh -c #(nop) ADD file:     188.1 MB
511136ea3c5a   22 months ago                                    0 B
```

你能够通过导出镜像，再导入镜像来获得扁平镜像，减少镜像大小。这不是一个好主意，因为你会丢失改动的历史信息，还有那些使用者下载镜像后可能获得的东西。更加聪明的做法是创建一个分支。

与其跟层系统（layer system）进行斗争，不如通过使用层系统来创建分支，这能够同时解决镜像大小和层增长问题。层系统使其可以返回到某个镜像历史节点，并且使创建新分支变得更加简单方便。每当你从同一个镜像创建一个容器，你就潜在地创建了一个新的分支。

在重新思考对新 ubunut-git 镜像的构建策略后，你应该基于 ubunut:latest 镜像开始构建。有了从 ubuntu:latest 创建的新容器，你就能够安装任何你想要的 Git 版本了。产生的结果将会是原始的 ubunut-git 镜像和你新建立的镜像共享相同的父层，并且新的镜像不会有任何不相关的无用内容。

分支很有可能会导致你重复在同等分支中已经完成的步骤。手动做这部分内容很可能导致错误。通过 Dockerfile 自动化构建镜像是一个更好的选择。

有时需要从零开始构建一个全新的镜像。如果你的目标是保持镜像足够小或者你需要的技术有比较少的依赖，那么这种做法是有益的。其他时候，你可能想通过修改一个镜像的历史来减少镜像大小。不论哪种情况，你都需要一个导入和导出全部文件系统的方式。

7.3 导出和导入扁平文件系统

为了满足这个需求，Docker 提供了两个命令来导入和导出文件归档（archives of files）。socker export 命令会将扁平的联合文件系统的所有内容导出到标准输出或者一个压缩文件上。输出信息包含了所有从容器角度能够观察到的文件。如果你需要在容器上下文外使用镜像中的文件系统，这是非常有帮助的。你也可以使用 docker cp 命令来完成这个目标，但如果你想要多个文件，导出整个文件系统可能是更直接的办法。

创建一个新容器并且使用 export 子命令来获得新容器文件系统的扁平复制：

```
docker run --name export-test \
   dockerinaction/ch7_packed:latest ./echo For Export
docker export --output contents.tar export-test
docker rm export-test
tar -tf contents.tar
```

导出文件系统内容

显示归档内容

这些命令会在当前目录生成一个名为 contents.tar 的文件。这个文件应该包含两个文件。现在，你能够解压、查看或改动这些文件。如果你省略了--output（或-o）选项，那么文件系统的内容就会以压缩文件格式导流到标准输出上。导流到标准输出有利于和其他支持压缩文件格式的 shell 程序一起工作。

docker import 命令会将压缩格式的内容导入到一个新镜像中。import 命令能够识别多种压缩或未压缩的压缩文件格式。在文件系统被导入的过程中，一个可选的 Dockfile 指令也能够被应用。导入文件系统是一个将最小文件集合导入到新镜像的简单方法。

为了体验它是如何有用的，我们可以考虑一个静态链接的、Go 版本的 Hello World。创建一个空目录，将下面的代码复制到一个名为 hello-world.go 的新文件中：

```
package main
import "fmt"
func main() {
        fmt.Println("hello, world!")
}
```

你可能没有在计算机上安装 Go，但这对 Docker 用户来说不是一个问题。通过运行下面的命令，Docker 会拉取一个包含有 Go 编译器的镜像，编译并且静态链接这个代码（这意味着编译后的可执行文件能够完全独立运行），然后将生成的程序放回到你的目录中：

```
docker run -rm -v "$(pwd)":/usr/src/hello
-w /usr/src/hello golang:1.3 go build -v
```

如果一切运行顺利，你会在同一个目录下发现一个可执行文件（二进制文件）。这意味着这段静态链接版本的 Hello World 能够在容器中独立运行，不需要依赖于任何文件。下一步，将这个程序放到压缩文件中：

```
tar -cf static_hello.tar hello
```

现在这个程序已经被打包进压缩文件中，你可以使用 docker import 命令将它导入到镜像中：

```
docker import -c "ENTRYPOINT [\"/hello\"]" - \
    dockerinaction/ch7_static < static_hello.tar
```

通过 UNIX 管道将 tar 文件重定向

在这个命令中，你使用-c 选项来设置一个 Dockerfile 命令。使用的命令设置了新镜像的入口点。Dockerfile 命令的具体语法将在第 8 章讲述。在这个命令中更加有趣的是第一行最后的连字符-。这个连字符表示压缩文件的内容会通过标准输入导入。如果你不从本地文件系统获取压缩文件，而是从远程 Web 服务器抓取压缩文件，你也可以在这个位置指定一个 URL 来实现。

你将生成的镜像标记为 dockerinaction/ch7_static_repository。花一些时间去研究它的结果：

```
docker run dockerinaction/ch7_static
docker history dockerinaction/ch7_static
```

输出结果：hello，world

你会发现这个镜像的历史只存在一层：

```
IMAGE           CREATED          CREATED BY      SIZE
edafbd4a0ac5    11 minutes ago                   1.824 MB
```

在这个情况下，生成的镜像非常小，有两个理由：第一，我们生成的程序仅有 1.8MB 大小，并且我们没有包含任何操作系统文件或者辅助程序到镜像中，因此，这是一个最小化的镜像。第二，它仅有一个层，因此并不存在被删除或被无用的文件被包含进镜像的底层中。使用单一层（或扁平）镜像的缺点是，你的系统并没有从层复用中受益。当然，如果你所有的镜像都足够小，那么这可能不是一个问题。但是当镜像由多层组成，或语言不提供静态链接时，考虑负载变得非常重要。

每个镜像设计的决定都需要平衡，包括是否使用扁平镜像。无论你使用什么机制来构建镜像，你的镜像使用者都需要一个一致的、可预测的方式来识别不同的版本。

7.4 版本控制的最佳实践

实用的版本管理能够帮助用户更好地利用镜像。一个实用的版本控制框架的目标就是能够清楚地交流并且提供采用灵活性。

除非这是你的第一个软件，否则仅保持或构建一个版本的软件是不够的。如果你正在发布软件的第一个版本，你应该立马考虑到用户的使用体验。版本控制为什么重要，原因在于它们确定了软件采用者依赖的交流规则。意外的、不符合规则的软件改动会带来很多问题。

在 Docker 中，维护同一个软件的多个版本的关键是设置正确的仓库标签。每个仓库包含多个标签，多个标签能够指向同一个镜像，这两点是实用标签框架的核心。

不同于用来创建标签的其他两个命令，`docker tag` 是唯一一个能够应用于已存在的镜像的命令。为了理解如何使用标签，和它们是如何影响用户使用体验的，让我们来思考如图 7-7 所示的两个标签框架。

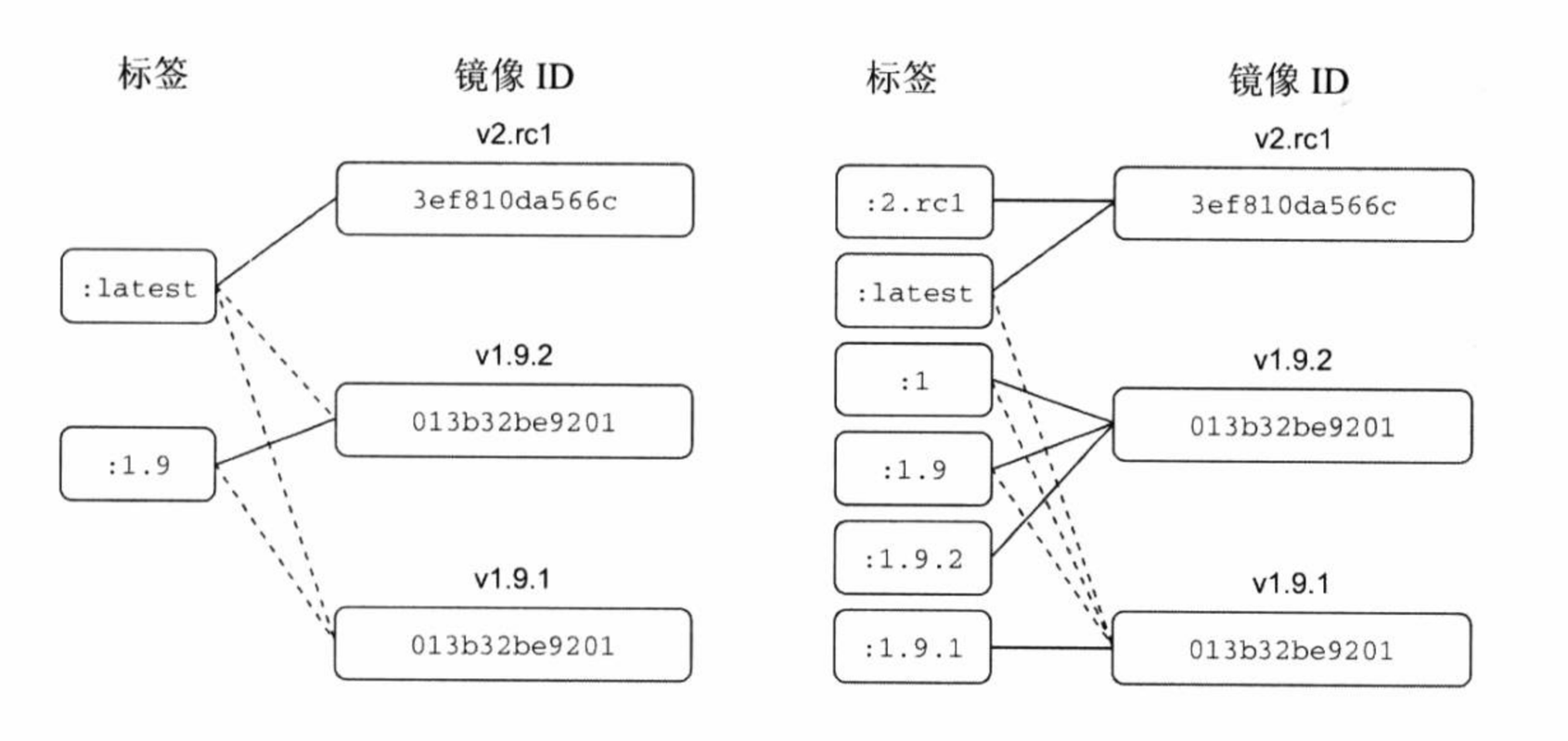


图 7-7 对于包含三个镜像的同一个仓库，左右分别是两个不同的标签框架
虚线表示一个标签和一个镜像间的旧关系

图 7-7 中左边的标签框架有两个问题。第一个问题，提供了很差的采用灵活性。用户能够选择依赖于 1.9 或者 latest。当用户采用 1.9 版本，它的实际实现是 1.9.1 版本，因此开发可能依赖于 1.9.1 版本定义的行为。但是当系统升级到 1.9.2 版本时，标签 1.9 指向 1.9.2 版本，由于没有办法显示声明依赖关系，因此开发代码可能会出现问题。

消除这个问题的最佳方式就是定义一个具有一致性的标签版本。这并不是说要引入一个第三方的版本控制系统。它的意思是，版本控制系统的最小单元要和实际软件的迭代的最小单元保持一致。通过这种方式提供多种标签，你可以让用户自己决定他们想要使用的版本。

我们再来看图 7-7 右边的系统。采用标签 1 的用户，总会使用版本 1.x 分支的最新版本（the highest minor）来构建系统。采用标签 1.9 的用户，则总会使用版本 1.9.x 分支的最新版本。那些需要小心谨慎地在不同版本之间迁移的采用者，就能通过以上的方式来控制版本。

第二个问题跟 `latest` 标签有关。在左边框架中，现在 latest 指向了一个不另外标记的镜像，因此采用者没有办法知道它指向了哪个版本的标签。在这个例子中，我们可以看到它指向了软件下一个主要版本的候选发行版。一个不加怀疑的用户可能就会采用 latest 标签，并且认为它指向的是一个被另外标记过的最新版本。

关于 `latest` 标签，还有另外一个问题。这个标签实际被采用的频率比它原本应该被采用的频率更高。这是因为 `latest` 标签是默认标签，并且 Docker 还是一个非常年轻的社区。这带来的影响就是，一个负责任的仓库为维持者总应该确保仓库的 `latest` 标签指向最新的稳定版，而不是最新的测试版。

最后一个需要牢记在心的就是容器的上下文。版本控制并不仅仅是你的软件，还包括所有被打包的软件依赖。举个例子，如果你将软件打包在一个特殊的 Linux 发行版中，比如 Debian，那么这些额外的包会成为镜像交流规则的一部分。你的用户会基于你的镜像来进行开发，在某些时候，可能会依赖于你镜像中的某一个特殊的 shell 或者脚本。如果你突然将你的软件重新打包到其他系统，比如 CentOS，但不对软件做出相应改动，那么你的用户开发的系统可能会出现问题。

如果软件依赖变动了，或者软件需要基于多个底层系统发布，那么这些依赖应该要被包含到你的标签框架中。

Docker 的官方仓库是非常值得学习的例子。下面是官方的 golang 仓库，每一行都代表一个不同的镜像：

```
1.3.3,              1.3
1.3.3-onbuild,      1.3-onbuild
1.3.3-cross,        1.3-cross
1.3.3-wheezy,       1.3-wheezy
1.4.2,              1.4,            1,              latest
1.4.2-onbuild,      1.4-onbuild,    1-onbuild,      onbuild
1.4.2-cross,        1.4-cross,      1-cross,        cross
1.4.2-wheezy,       1.4-wheezy,     1-wheezy,       wheezy
```

所有列根据版本范围整齐排列，左边是处于构建的版本，右边是主要版本。每一个版本还有一个额外的基础镜像模块，这个信息以注释的方式添加在标签中。

用户知道实际最新版本是 1.4.2。如果一个用户需要基于 debian:wheezy 平台构建的最新版本，他可以使用 wheezy 标签。需要带有 ONBUILD 的 1.4 版本镜像，可以使用 1.4-onbuild 标签。这个框架将版本升级的控制和责任交给了镜像的使用者。

7.5 小结

这章包含 Docker 镜像创建、标签管理和其他发布所需要关心内容。这些内容能够帮助你构建镜像和成为一个更加优秀的容器使用者。下面是本章的关键内容：

- 当使用 `docker commit` 命令提交容器时，新的镜像被创建。
- 当一个容器被提交，启动容器时的配置也会被编码进新镜像的配置文件中。
- 一个镜像由多层以栈形式组成，且镜像由其中的最顶层来标识。
- 镜像的磁盘大小就是组成镜像的层的大小总和。
- 可以使用 `docker export` 和 `docker import` 命令将镜像导出为压缩文件格式，或将压缩文件导入到镜像。
- `docker tag` 命令能够被用来对同一个仓库赋予多个标签。
- 仓库维护者应该保持标签的实用性，让用户更容易采用和迁移控制。
- 将软件的最新稳定版标记为 `latest`。
- 提供细粒度、重叠的标签，这有利于用户掌控软件的版本进展。

第 8 章　构建自动化和高级镜像设置

本章介绍

- 使用 Dockerfile 自动化打包
- 元数据指令
- 文件系统指令
- 多进程和持久的容器
- 可信的基础镜像
- 用户相关的内容
- 降低镜像的攻击面

Dockerfile 是一个文件，它由构建镜像的指令组成。指令由 Docker 镜像构建者自上而下排列，能够被用来修改镜像的任何信息。使用 Dockerfile 构建镜像使得很多任务变得非常简单，如同从计算机添加一个文件到容器中，只需要一行指令。本节包含了 Dockerfile 的基础知识、为什么使用 Dockerfile 是最佳方式的原因、对 Dockerfile 指令的简要概括、如何添加未来构建行为的内容。让我们从一个熟悉的例子开始吧。

8.1　使用 Dockerfile 打包 Git

让我们先回顾一下在 Ubuntu 上安装 Git 的例子。之前我们已经手工构建了这样一个镜像，你应该能意识到使用 Dockerfile 的很多细节和优点。

首先，创建一个新目录，并使用你最喜欢的文本编辑器在该目录中创建一个名为 Dockerfile 的文件。将下面五行文字复制到文件中，并且保存：

```
# An example Dockerfile for installing Git on Ubuntu
FROM ubuntu:latest
MAINTAINER "dockerinaction@allingeek.com"
RUN apt-get install -y git
ENTRYPOINT ["git"]
```

在详细解释这个例子前，让我们先在包含 Dockerfile 文件的目录中使用 `docker build` 命令，从 Dockerfile 文件创建一个新镜像，并将新镜像的标签设为 `auto`：

```
docker build -tag ubuntu-git:auto .
```

输出结果包含了关于步骤的多行信息，以及 apt-get 命令的输出结果，最终还会输出类似以下格式的信息：

```
Successfully built 0bca8436849b
```

运行以上的命令就会启动构建进程。当进程结束，你就能够获得一个全新的镜像。使用下面的命令，能够看到所有 ubuntu-git 镜像，包括最新产生的镜像：

```
docker images
```

带有 auto 标签的镜像也显示在输出结果中：

```
REPOSITORY    TAG         IMAGE ID        CREATED              VIRTUAL SIZE
ubuntu-git    auto        0bca8436849b    10 seconds ago       225.9 MB
ubuntu-git    latest      826c66145a59    10 minutes ago       226.6 MB
ubuntu-git    removed     826c66145a59    10 minutes ago       226.6 MB
ubuntu-git    1.9         3e356394c14e    41 hours ago         226 MB
...
```

现在，你能够使用这个新镜像来运行 Git 命令了：

```
docker run -rm ubuntu-git:auto
```

从输出结果可以看出，使用 Dockerfile 构建的镜像能够正常工作，并且功能上和之前手工创建的没有区别。让我们来具体分析一下这个例子：

第一步，你先创建了带有四个指令的 Dockerfile：

- `FROM ubuntu:latest` —— 和手工创建类似，告诉 Docker 从最新的 Ubuntu 镜像创建新镜像。
- `MAINTAINER` —— 设置镜像维护者的名字和邮箱。当用户遇到问题时，这些信息能够帮助这些人联系维护者。设置这些信息之前都是通过调用 `commit` 子命令来完成的。
- `RUN apt-get install -y git` —— 告诉 Docker 运行该命令来安装 Git。

- ENTRYPOINT ["git"] —— 将镜像的入口点设置为 git。

和大多数脚本一样，Dockerfile 也支持注释。任何以#开头的行都会被忽略。详细地对 Dockerfile 进行注释是非常重要的。除了能够提高 Dockerfile 的可维护性，注释还能够帮助用户更好地审计他们关心的镜像，这样易于用户判断是否采用和传播。

Dockerfile 唯一一条特殊的规则就是第一个指令必须是 FROM。如果你从一个空镜像开始，且想要打包的软件没有依赖，或者你能够自己提供所有的依赖，那么你可以从一个特殊的空镜像开始，它的名字是 scratch。

当写好 Dockerfile 并保存，你就能够通过 docker build 命令来启动构造程序了。该命令有一个选项和一个参数。--tag（或-t）选项的值指定了你想要使用的完整仓库设计。在这个例子中，你使用了 ubuntu-git:auto。最后的参数则指定了 Dockerfile 的位置，表示在当前目录寻找文件。

docker build 命令还有另外一个选项--file（或-f），这个选项让你能够设置 Dockerfile 的名字。Dockerfile 是默认的文件名字，但你能够使用这个选项让构建程序去寻找名为 BuildScript 的文件。注意，这个选项只能设置文件的名字，而不能设置文件的位置。最后一个参数是设置位置的唯一方式。

构造程序会将你之前手动构建镜像做的任务自动化。每个指令都会创建一个新容器，然后做指定的修改。当修改完成，构建程序会提交新容器，然后以同样的方式执行下一条指令。

构造程序首先会检查 FROM 指令指定的镜像是否被安装。如果没有，Docker 会自动尝试拉取镜像。让我们来看看之前运行的 build 子命令的第 0 步的输出结果：

```
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:latest
 ---> b39b81afc8ca
```

你可以看到，FROM 指令指定的镜像是 ubuntu:latest，这个镜像已经被安装在计算机上。该镜像的短 ID 也包含在输出中。

下一个指令指定了维护者的信息。这会创建一个新容器，然后提交它。下面是第 1 步的输出结果：

```
Step 1 : MAINTAINER "dockerinaction@allingeek.com"
 ---> Running in 938ff06bf8f4
 ---> 80a695671201
Removing intermediate container 938ff06bf8f4
```

输出结果包含了新容器的 ID 和被提交层的 ID。这个层就是下一条指令 RUN 使用的镜

像的最顶层。RUN 指令的输出结果包含了 `apt-get install -y git` 命令的输出结果。如果你对这个输出不感兴趣，你可以在 `docker build` 命令上使用`–quiet` 或`-q` 选项。这个选项会使得构造程序运行在安静模式下，所有中间容器的输出都会被消除。RUN 指令在安静模式下的输出结果如下：

```
Step 2 : RUN apt-get install -y git
 ---> Running in 4438c3b2c049
 ---> 1c20f8970532
Removing intermediate container 4438c3b2c049
```

尽管这一步通常需要花费更长的时间来完成，你可以看到指令和输入，还有正在运行命令的容器的 ID 和产生的新层的 ID。最后，ENTRYPOINT 指令执行了完全相同的步骤，输出结果并不意外，和之前的非常类似：

```
Step 3 : ENTRYPOINT git
 ---> Running in c9b24b0f035c
 ---> 89d726cf3514
Removing intermediate container c9b24b0f035c
```

在构建过程中的每一步都会有一个新层被加入到要产生的镜像中。这不仅意味着你能够从任意一步开始创建分支，更重要的是构建程序能够缓存每一步的结果，当运行完几个指令，下一条指令出现问题时，构建程序能够在问题被修复后，从同一步重新启动。你可以故意破坏 Dockerfile 来亲身体验下这个效果。

将下面这一行加入到 Dockerfile 的最后：

```
RUN This will not work
```

然后重新运行构造命令：

```
docker build -tag ubuntu-git:auto .
```

输出结果会显示出哪些步骤由于被缓存而能够被构成程序跳过：

```
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:latest
 ---> b39b81afc8ca
Step 1 : MAINTAINER "dockerinaction@allingeek.com"
 ---> Using cache
 ---> 80a695671201
Step 2 : RUN apt-get install -y git
 ---> Using cache
 ---> 1c20f8970532
Step 3 : ENTRYPOINT git
 ---> Using cache
```

注意缓存使用

```
 ---> 89d726cf3514
Step 4 : RUN This will not work
 ---> Running in f68f0e0418b5
/bin/sh: 1: This: not found
INFO[0001] The command [/bin/sh -c This will not work] returned a non-zero
  code: 127
```

第 1 步到第 3 步都被跳过，这是因为它们已经在上一次构建中被构建过。第 4 步失败了，这是由于在容器中不存在名为 This 的程序。这些输出是非常有用的，因为它们能告诉你 Dockerfile 具体在哪里出现了问题。如果你修复了这个问题，再次运行构造程序，同样的步骤会被跳过，并且构造会最终成功，输出结果类似于 Successfully built d7a8ee0cebd4。

如果构建过程需要下载，或者包含某些需要消耗大量时间的任务，那么缓存会节省很多时间。如果你需要完整地从零开始构建，请使用--no-cache 选项来禁止缓存的使用。但请注意，确保完全需要时才禁止缓存。

这个小例子仅使用了 14 个 Dockerfile 指令中的 4 个。其中所有的被添加进新镜像的文件受限于只能从网络下载，以受限制的方式来修改环境，因此这提供了一个非常通用的工具。下一个例子将带有更加具体的目的，并且会提供更加完整的 Dockerfile 命令解析。

8.2 Dockerfile 入门

Dockerfile 具有信息表达性，且易于理解，这些要归功于 Dockerfle 支持注释的简洁语法。你能够使用任何版本控制系统来跟踪 Dockerfile 文件的改动。维护多个版本的镜像就和管理多个 Dockerfile 一样简单。Dockerfile 构建程序自身使用缓存技术来解决快速开发和迭代带来的问题。这个构建过程可追踪且可重用。它们能够很简单地和现有的构建系统、持续构建和集成工具一起工作。所有以上的情况都说明 Dockerfile 比手工构建镜像更可取，因此学习 Dockerfile 非常重要。

除了一个指令，其他所有的 Dockerfile 指令都包含在本节的例子中。这个特殊指令是 ONBUILD，它针对特殊的用户情景，因此包含在下一节中。所有包含在这里的指令都只能被简要地介绍，如果想要深入研究每个指令，在线的 Docker 文档（https://docs.docker.com/reference/builder/）是最好的去处。Docker 还在在线文档（https://docs.docker.com/reference/builder/）中提供了一个最佳实践章节。

8.2.1 元数据指令

第一个例子构建了一个基础镜像和另外两个打包有不同版本邮件程序的镜像，这些邮件程序在第 2 章被使用过。这些程序的目的是在一个 TCP 端口上监听信息，然后将这些信

息发送到指定的收件人那里。第一个版本会监听所有的信息，但是只会在日志文件中记录这些信息。第二个则会将这些信息以 HTTP POST 的方式发送到指定的 URL 上。

之所以选择 Dockerfile，其中一个原因就是它能够简化将文件从计算机复制到镜像的过程。但并不是所有文件都需要被复制到新镜像。当你启动一个新项目，第一件事就是定义哪些文件永远不应该被复制进镜像中。你可以在名为.dockerignore 的文件中指定这些信息。在这个例子中，你将会创建三个 Dockerfile 文件，这些文件并不需要复制到新镜像中。

使用你最喜欢的文本编辑器，创建一个名为.dockerignore 的文件，然后将以下内容复制到文件中：

```
.dockerignore
mailer-base.df
mailer-logging.df
mailer-live.df
```

当复制完成，保存并关闭文件。这个文件会防止.dockerignore 文件和名为 mailer-base.df、mailer-log.df、mailer-live.df 的文件在构建过程中被复制到新镜像中。完成这些工作，下面你就能开始构建基础镜像了。

构建基础镜像就是创建另外两个容器共用的层。每个不同版本的邮件程序的镜像将会基于一个名为 mailer-base 的镜像来构建。当你创建了一个 Dockerfile，你需要记住，每个 Dockerfile 指令都会导致一个新层被创建。指令应该尽可能合并，这是因为构建程序不会进行任何的优化。接下来让我们亲身实践一下，创建一个名为 mailer-base 的文件，并将下面的内容复制到文件中：

```
FROM debian:wheezy
MAINTAINER Jeff Nickoloff "dia@allingeek.com"
RUN groupadd -r -g 2200 example && \
    useradd -rM -g example -u 2200 example
ENV APPROOT="/app" \
    APP="mailer.sh" \
    VERSION="0.6"
LABEL base.name="Mailer Archetype" \
      base.version="${VERSION}"
WORKDIR $APPROOT
ADD . $APPROOT
ENTRYPOINT ["/app/mailer.sh"]
EXPOSE 33333
# 不要在基础镜像中设置默认用户，否则
# 接下来的实现将不能够更新镜像
# USER example:example
```

这个文件不存在

在包含有 mailer-base 文件的目录下使用 docker build 命令就能够开始构建镜像。

-f 选项指定了文件的名字：

```
docker build -t dockerinaction/mailer-base:0.6 -f mailer-base.df .
```

五个新的指令在这个 Dockerfile 文件中被引入。第一个指令是 ENV，类似于 docker run 或 docker create 命令的--env 选项，ENV 指令设置了镜像的环境变量。在这个例子中，一个 ENV 指令被用来设置了三个不同的环境变量。这也能通过三个 ENV 指令来完成，但这么做会导致三个层被创建。和 shell 脚本类似，你可以使用一个反斜杠来转义换行符，使得文档看起来结构清晰：

```
Step 3 : ENV APPROOT "/app" APP "mailer.sh" VERSION "0.6"
 ---> Running in 05cb87a03b1b
 ---> 054f1747aa8d
Removing intermediate container 05cb87a03b1b
```

Dockerfile 文件中声明的环境变量不仅对产生的镜像有效，它们还能够在其他 Dockerfile 指令中使用。在这个例子中，环境变量 VERSION 在下一个指令 LABEL 中被使用：

```
Step 4 : LABEL base.name "Mailer Archetype" base.version "${VERSION}"
 ---> Running in 0473087065c4
 ---> ab76b163e1d7
Removing intermediate container 0473087065c4
```

LABEL 指令用来定义键值对，这些键值对被记录为镜像或容器的额外元数据。这和 docker run 或 docker create 命令的--label 选项在功能上一致。和之前的 ENV 指令一样，多个 label 可以且应该放入到一个指令中。在这个例子中，VERSION 环境变量的值替换了 base.version 的值。通过这种方式来使用环境变量，VERSION 的值不仅能够用于 label，对运行在容器中的进程也是可用的。这增加了 Dockerfile 的可维护性，这是因为当变动发生时，只需修改一个地方即可，其他使用了这个变量的地方会自动更新，保证了改动的一致性。

接下来的两个指令是 WORKDIR 和 EXPOSE。同样，它们和 docker run 或 docker create 命令上对应选项的功能是一致的。一个环境变量作为一个参数在 WORKDIR 命令中被使用：

```
Step 5 : WORKDIR $APPROOT
 ---> Running in 073583e0d554
 ---> 363129ccda97
Removing intermediate container 073583e0d554
```

使用 WORKDIR 指令的会生成一个默认工作目录为/app 的新镜像。和命令行选项类似，当指定的 WORKDIR 目录不存在，那么这个目录会被自动创建。最后，EXPOSE 指令创建了一个层，对外开放 TCP 的 33333 端口：

```
Step 7 : EXPOSE 33333
 ---> Running in a6c4f54b2907
 ---> 86e0b43f234a
Removing intermediate container a6c4f54b2907
```

在这个 Dockerfile 中，你应该能够识别 FROM 指令、MAINTAINER 指令和 ENTRYPOINT 指令。简单来说，FROM 指令使得栈从 debian:wheezy 镜像顶部开始，任何被构建的新层都会被放置在这个镜像的最上层。MAINTAINER 指令设置了镜像元数据 Author 的值。ENTRYPOINT 指令则设置了在容器启动时需要被运行的可执行程序。比如说入口点被设置为 exec ./mailer.sh，这使用了该指令的 shell 格式。

ENTRYPOINT 指令有两种格式：shell 格式和 exec 格式。shell 格式类似于一个 shell 命令，其中的参数以空格为界限分隔开来。exec 格式是一个字符串的数组，其中第一个值是要执行的命令，剩下的值则是参数。shell 格式指定的命令将会被作为默认 shell 的一个参数来执行。具体点说，指定的命令在运行时会以/bin/sh -c 'exec ./mailer.sh'的形式执行。最重要的是，如果 ENTRYPOINT 使用了 shell 格式，那么 CMD 指令提供的所有其他参数，或 docker run 命令在运行时指定的额外参数都会被忽略。这使得 ENTRYPOINT 的 shell 格式不那么灵活可变动了。

你从输出构建的输出结果可以看到，ENV 和 LABEL 指令都生成了一个新的步骤和层。但是，输出结果并没有说明环境变量是否被正确替换使用。为了验证这个问题，你需要使用下面命令来检查镜像：

```
docker inspect dockerinaction/mailer-base:0.6
```

要点：记住，docker inspect 命令只能够被用来查看容器或镜像的元数据。在这个例子中，你使用它来检查镜像。

相关的输出如下：

```
"Env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "APPROOT=/app",
    "APP=mailer.sh",
    "VERSION=0.6"
],
...
"Labels": {
    "base.name": "Mailer Archetype",
    "base.version": "0.6"
},
...
"WorkingDir": "/app"
```

从输出结果可以很清楚地看到，环境变量被成功替换使用。你同样可以在 ENV、ADD、COPY、WORKDIR、VOLUME、EXPOSE 和 USER 指令中使用这种形式的环境变量替换使用。

最后被注释的行是一个元数据指令 USER。它为接下来的构建步骤和从这个镜像创建的容器指定了用户和用户组。在这个例子中，在基础镜像中设置这个选项能够防止任何下游的 Dockerfile 安装软件。这意味着，如果要安装软件，这些 Dockerfile 需要将默认用户转回 root 用户，重新获得权限。这么做将会创建至少两个额外的层。更好的方法是在基础镜像创建用户和用户组账户，然后让具体的实现者在他们完成构建时再设置默认的用户。

在这个 Dockerfile 中，最令人好奇的还是 ENTRYPOINT 指向了一个不存在的文件，这会带了什么后果？答案是，当你尝试从这样一个基础镜像运行一个容器时，这个入口点会失败。但是由于这个基础镜像设置了入口点，邮件程序的具体实现就又少了一个需要被重复的层。接下来的两个 Dockerfile 将会为 mailer.sh 提供不同的实现。

8.2.2 文件系统指令

拥有自定义功能的镜像需要修改文件系统。Dockerfile 定义了三个指令来修改文件系统：COPY、VOLUME 和 ADD。第一个实现的 Dockerfile 写在名为 mailer-logging.df 的文件中：

```
FROM dockerinaction/mailer-base:0.6
COPY ["./log-impl", "${APPROOT}"]
RUN chmod a+x ${APPROOT}/${APP} && \
    chown example:example /var/log
USER example:example
VOLUME ["/var/log"]
CMD ["/var/log/mailer.log"]
```

在这个 Dockerfile 中，你基于 mailer-base 基础镜像来产生新镜像。出现的三个新指令分别是 COPY、VOLUME 和 CMD。COPY 指令将会从镜像被创建的文件系统上复制文件到容器中。COPY 指令至少需要两个参数。最后一个参数是目的目录，其他所有的参数则为源文件。这个指令只拥有一个意外的特性：任何被复制的文件的所有权都会被设置为 root 用户。无论在 COPY 指令前面设置的默认用户是什么，这种情况都会发生。因此，最好在所有需要更新的文件都复制到镜像后，再使用 RUN 指令来修改文件的所有权。

和 ENTRYPOINT 等指令类似，COPY 指令同样支持 shell 格式和 exec 格式。但是如果任何一个参数包含了空格，那么你必须要使用 exec 格式。

要点：尽可能使用 exec（或字符串数组）格式是一个最佳实践。这样至少能够使得 Dockerfile 保持一致，避免不同风格的写法混合在一起。这将会使得你的 Dockerfile 更加容易阅读，并且不需要深入了解这两个指令之间的区别，就能够确保指令按照你所期望的那样去运行。

第二个新指令是 VOLUME。如果你理解了 docker run 或 docker create 命令的 --volume 选项，那么这个指令将会按照你所期望的那样运行。在字符串数组参数中的每一个值都会在产生的新层中被创建为一个新的卷定义。在镜像构建时定义卷比在运行时更加受到限制。你没有办法在镜像构建时指定一个绑定-挂载（bind-mount）卷或者只读卷。这个指令只能够在文件系统中创建一个指定的位置，然后将一个卷定义添加到镜像元数据中。

在这个 Dockerfile 的最后一个指令是 CMD。CMD 和 ENTRYPOINT 很相关。它们都能够支持 shell 格式和 exec 格式，并且都能够被用来在容器中启动一个进程。但它们依旧存在一些很重要的区别。

CMD 指令表示入口点的一个参数列表。一个容器的默认入口点是/bin/sh。如果一个容器的入口点没有被设置，这个默认值会被使用。但是，如果入口点被设置了，并且使用的是 exec 格式，那么你将使用 CMD 指令来设置默认参数。这个 Dockerfile 使用的基础镜像将 ENTRYPOINT 定义为邮件程序。同时，这个 Dockerfile 注入了一个 mailer.sh 的实现，并且指定了一个默认参数。这个参数是日志文件的位置。

在构建这个镜像前，你需要实现邮件程序的 logging 版本。创建一个/log-impl 目录，在目录中，创建一个名为 mailer.sh 的文件，并且将下面的脚本复制到文件中：

```
#!/bin/sh
printf "Logging Mailer has started.\n"
while true
do
    MESSAGE=$(nc -l -p 33333)
    printf "[Message]: %s\n" "$MESSAGE" > $1
    sleep 1
done
```

这个脚本的结构细节并不重要。你所要知道的是，这个脚本将会在 33333 端口启动一个邮件程序后台进程，并且将它收到的每一个信息写入到邮件程序第一个参数指定的文件中。使用下面的命令从包含 mailer-logging.df 文件的目录中构建 mailer-logging 镜像：

```
docker build -t dockerinaction/mailer-logging -f mailer-logging.df .
```

镜像构建的结果应该是平淡的、不令人意外的。下面让我们使用这个新镜像来启动一个命名容器：

```
docker run -d -name logging-mailer dockerinaction/mailer-logging
```

现在这个邮件程序被构建且正在运行。链接到这个容器的其他容器发送的信息将会记

录到/var/log/mailer.log 文件中。尽管这在实际情况中并不是那么有趣或有用，但它利于测试。发送 Email 可能更适合于操作监测。

下一个实现例子使用了 Amazon Web Servies 提供的 Email Service。让我们从另一个名为 mailer-live.df 的 Dockerfile 开始吧：

```
FROM dockerinaction/mailer-base:0.6
ADD ["./live-impl", "${APPROOT}"]
RUN apt-get update && \
    apt-get install -y curl python && \
    curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py" && \
    python get-pip.py && \
    pip install awscli && \
    rm get-pip.py && \
    chmod a+x "${APPROOT}/${APP}"
RUN apt-get install -y netcat
USER example:example
CMD ["mailer@dockerinaction.com", "pager@dockerinaction.com"]
```

这个 Dockerfile 包含了一个新的指令 ADD。ADD 指令类似于 COPY 指令，但它们有以下两点重要区别。ADD 指令：

- 如果指定了一个 URL，会拉取远程源文件。
- 会将被判定为存档文件的源中的文件提取出来。

自动提取存档文件更为有用。使用 ADD 指令的远程拉取功能并不是一个好的实践。原因在于尽管这个特性非常方便，但是它没有提供任何机制来清理不被使用的文件，这会导致额外的层。作为替代品，你应该使用链状的 RUN 指令，就像 mailer-live.df 的第三个指令。

在这个 Dockerfile 中，需要被注意的其他指令还有 CMD，它接收了两个参数，分别指定了你要发送的邮件的发件人和收件人。而 mailer-logging.df 仅仅指定了一个参数，这是它们的不同之处。

接下来，在包含有 mailer-live.df 文件的目录下创建一个名为 live-impl 的子目录，并在这个子目录下，将下面的脚本复制到名为 mailer.sh 的文件中：

```
#!/bin/sh
printf "Live Mailer has started.\n"
while true
do
  MESSAGE=$(nc -l -p 33333)
  aws ses send-email --from $1 \
    --destination {\"ToAddresses\":[\"$2\"]} \
    --message "{\"Subject\":{\"Data\":\"Mailer Alert\"},\
                \"Body\":{\"Text\":{\"Data\":\"$MESSAGE}\"}}}"
  sleep 1
done
```

这个脚本的关键信息在于，和其他实现一样，它也会在 33333 端口等待连接，然后对接收到的任何信息进行操作，接下来睡眠一段时间，最后重新开始等待其他信息。而在这个实现中，脚本会使用 Simple Email Service 命令行工具发送一个邮件。使用下面两个命令构建新镜像，然后启动一个新容器：

```
docker build -t dockerinaction/mailer-live -f mailer-live.df .
docker run -d -name live-mailer dockerinaction/mailer-live
```

如果你对这些容器链接了一个观察者容器，你会发现 logging 邮件程序按照预期运行，但是 live 邮件程序似乎在连接 Simple Email Service 来发送信息上遇到了问题。经过一些调查，你最终意识到容器没有配置正确。aws 程序需要设置指定的环境变量。

为了让这个例子正常工作，你需要设置环境变量 AWS_ACCESS_KEY_ID、AWS_SECRET_ ACCESS_KEY 和 AWS_DEFAULT_REGION。以这种方式发现执行的先决条件对用户来说是困惑的。8.4.1 节详细介绍了一个镜像设计模式，它能够减少这种问题带来的阻力，促进采用。

在学习设计模式前，你需要学习最后一个 Dockerfile 指令。记住，并不是所有的镜像都包含应用。有一些是作为下游镜像的平台而被构建的。这些情况能够从注入下游构建时（build-time）行为的能力中受益。

8.3 注入下游镜像在构建时发生的操作

只有一个 Dockerfile 指令没有包含在以上的初级教程中。这个指令就是 ONBUILD。如果生成的镜像被作为另一个构建的基础镜像，则 ONBUILD 指令定义了需要被执行的那些指令。举个例子，你可以使用 ONBUILD 指令来编译下游层提供的程序，上游的 Dockerfile 将构建目录的内容复制到一个已知目录，然后在这个目录中编译代码。上游的 Dockerfile 一般会使用类似以下形式的指令：

```
ONBUILD COPY [".", "/var/myapp"]
ONBUILD RUN go build /var/myapp
```

跟随在 ONBUILD 后的指令不会在包含它们的 Dockerfile 被构建时被执行。这些指令会被记录在生成镜像的元数据 ContainerConfig.OnBuild 下。上面的指令将会产生以下元数据：

```
...
"ContainerConfig": {
...
    "OnBuild": [
        "COPY [\".\", \"/var/myapp\"]",
        "RUN go build /var/myapp"
    ],
    ...
```

这个元数据会一直被保留，直到生成的镜像被另外的 Dockerfile 作为基础镜像。当一个下游的 Dockerfile 通过 FROM 指令使用了上游的镜像（带有 ONBUILD 指令的 Dockerfile 产生的镜像），那么这些在 ONBUILD 后跟随的指令将会在 FROM 指令后，下一条指令前被执行。

考虑下面的例子，你将能够准确地看到 ONBUILD 指令什么时候被注入到构建中。你需要创建两个 Dockerfile，并且执行两个构建命令来获得完整体验。首先，创建一个上游 Dockerfile，它使用了 ONBUILD 指令。将这个文件命名为 base.df，然后将下面的指令复制到该文件中：

```
FROM busybox:latest
WORKDIR /app
RUN touch /app/base-evidence
ONBUILD RUN ls -al /app
```

你将会看到，这个 base.df 构建的镜像会在/app 目录下创建一个名为 base-evidence 的空文件。ONBUILD 指令将会在构建时列出/app 目录下的内容，因此如果你想要准确地看到文件系统发生了哪些改动，你就不应该将构建运行在安静模式下。

接下来要创建的文件是下游的 Dockerfile。当这个文件被构建，你将会准确地看到产生的镜像发生了哪些改动。将这个文件命名为 downstream.df，并将以下内容复制到文件中：

```
FROM dockerinaction/ch8_onbuild
RUN touch downstream-evidence
RUN ls -al .
```

这个 Dockerfile 会使用 dockerinaction/ch8_onbuild 作为基础镜像，因此你在构建基础镜像时需要使用这个仓库名。接下来，你可以看到它又创建了一个文件，然后再一次列出/app 目录下的内容。

有了这两个 Dockerfile 文件，构建镜像已经一切就绪。运行以下命令来创建上游镜像：

```
docker build -t dockerinaction/ch8_onbuild -f base.df .
```

构建的输出结果类似于以下格式：

```
Sending build context to Docker daemon 3.072 kB
Sending build context to Docker daemon
Step 0 : FROM busybox:latest
---> e72ac664f4f0
Step 1 : WORKDIR /app
---> Running in 4e9a3df4cf17
---> a552ff53eedc
Removing intermediate container 4e9a3df4cf17
Step 2 : RUN touch /app/base-evidence
---> Running in 352819bec296
---> bf38c3e396b2
Removing intermediate container 352819bec296
Step 3 : ONBUILD run ls -al /app
---> Running in fd70cef7e6ca
---> 6a53dbe28364
Removing intermediate container fd70cef7e6ca
Successfully built 6a53dbe28364
```

接下来使用以下命令创建下游镜像：

```
docker build -t dockerinaction/ch8_onbuild_down -f downstream.df .
```

输出结果很清楚地显示出（基础镜像中）ONBUILD 指令什么时候被执行的：

```
Sending build context to Docker daemon 3.072 kB
Sending build context to Docker daemon
Step 0 : FROM dockerinaction/ch8_onbuild
# Executing 1 build triggers
Trigger 0, RUN ls -al /app
Step 0 : RUN ls -al /app
---> Running in dd33ddea1fd4
    total 8
    drwxr-xr-x    2 root     root  4096 Apr 20 23:08 .
    drwxr-xr-x   30 root     root  4096 Apr 20 23:08 ..
    -rw-r--r--    1 root     root     0 Apr 20 23:08 base-evidence
---> 92782cc4e1f6
Removing intermediate container dd33ddea1fd4
Step 1 : RUN touch downstream-evidence
---> Running in 076b7e110b6a
---> 92cc1250b23c
Removing intermediate container 076b7e110b6a
Step 2 : RUN ls -al .
---> Running in b3fe2daac529
    total 8
    drwxr-xr-x    2 root     root  4096 Apr 20 23:08 .
    drwxr-xr-x   31 root     root  4096 Apr 20 23:08 ..
    -rw-r--r--    1 root     root     0 Apr 20 23:08 base-evidence
    -rw-r--r--    1 root     root     0 Apr 20 23:08 downstream-evidence
---> 55202310df7b
Removing intermediate container b3fe2daac529
Successfully built 55202310df7b
```

你可以看到，在基础镜像构建的第 3 步，构成程序将 ONBUILD 指令注册到容器元数据中。接下来，下游镜像在构建过程的输出显示了是哪个指令触发了从基础镜像继承而来的 ONBUILD 指令。构建程序发现这个触发，并且立马在第 0 步（FROM 指令）后处理了它。从输出结果可以看到，ONBUILD 指令指定的 RUN 指令的结果包含在输出中，并且只能看到基础镜像构建的证据。接下来，构建程序继续执行下游 Dockerfile 的指令，它会再一次列出/app 目录下的内容。这一次，两个改动的证据都被列出了。

这个例子并不那么实用，但是却适合作为案例分析。你应该在 Docker Hub 中查找那些带有 onbuild 前缀的标签，了解这些在实际情况中是如何使用的。下面列出了我偏爱的一些镜像：

- https://registry.hub.docker.com/_/python/
- https://registry.hub.docker.com/_/golang/
- https://registry.hub.docker.com/_/node/

8.4 使用启动脚本和多进程容器

无论你选择哪个工具，你总是需要考虑一部分镜像设计方面的东西。你需要问自己运行在你容器中的软件是否需要启动援助、管理、监督或与容器中其他进程进行合作。如果是，那么你需要在镜像中包含一个启动脚本或者初始化程序，然后将它设置成入口点。

8.4.1 验证环境相关的先决条件

失败模式不利于传达信息，并且如果错误随时能够发生，这会导致用户不知所措。如果容器配置问题总是在启动时就能够触发错误，那么用户就能确信一个启动了的容器会保持正常运行。

在软件设计领域，越早触发错误和先决条件检测都是最佳实践。这对镜像设计也是同样有效的。应该被检测的先决条件就是上下文的假设。

Docker 容器对创建它们的环境没有控制权。但是它们对自己的执行有完全控制权。一个镜像作者能够通过在执行主要任务之前引入对环境和依赖的验证来提高镜像的用户体验。如果容器使用的镜像更早地失败并且显示出具有描述性的错误信息，那么一个容器用户能够更好地被告知镜像的需求。

举个例子，WordPress 需要设置特定的环境变量，或需要定义容器链接。没有这些上下文，WordPress 将不能够连接到存储博客数据的数据库。在一个容器中启动 WordPress，却

不能够访问它应该获得的数据，这是不合理的。WordPress 镜像使用一个脚本作为容器的入口点。这个脚本验证了容器上下文配置是否和当前包含版本的 WordPress 兼容。

如果任何需求没有被满足（一个链接没有被定义或者一个变量没有设置），那么这个脚本会在启动 WrodPress 前退出，容器也会意外地停止。

这类的启动脚本一般来说不通用，只适用于特殊场景（use-case specific）。如果你正在一个镜像中打包一个特定软件，那么你需要针对这个软件写一个脚本来验证先决条件。你的脚本需要尽可能地验证假设的上下文。这应该包含以下内容：

- 假定的链接（和别名）
- 环境变量
- 网络访问
- 网络端口可用性
- 根文件系统挂载参数（可读写或只读）
- 卷
- 当前用户

你可以使用任何脚本或程序语言来完成这个任务。但本着构建最小镜像的精神，最好使用已经包含在镜像中的语言或脚本工具来实现。大多数的基础镜像带有 shell，比如说/bin/sh 或/bin/bash。由于这个原因，shell 脚本成为了完成这个任务的最常用工具。

让我们来思考一下下面的 shell 脚本，它验证的程序依赖于一个 web 服务。在容器启动时，这个脚本确保要么另外一个容器已经链接到 web 别名，并且开放了 80 端口，要么 WEB_HOST 环境变量已经被定义：

```
#!/bin/bash
set -e

if [ -n "$WEB_PORT_80_TCP" ]; then
  if [ -z "$WEB_HOST" ]; then
    WEB_HOST='web'
  else
    echo >&2 '[WARN]: Linked container, "web" overridden by $WEB_HOST.'
    echo >&2 "===> Connecting to WEB_HOST ($WEB_HOST)"
  fi
fi

if [ -z "$WEB_HOST" ]; then
  echo >&2 '[ERROR]: specify a linked container, "web" or WEB_HOST environ-
   ment variable'
  exit 1
fi
exec "$@" # run the default command
```

如果你对 shell 脚本不熟悉，现在并不是一个恰当的时机来学习这个内容。这些知识是可获得的，并且存在多个适合自学的极佳资源。这个具体的脚本使用了一个环境变量和容

器链接同时被测试的模式。如果环境变量被设置了，那么容器链接会被忽略。在脚本最后，默认命令被执行。

镜像使用了启动脚本来验证配置的正确性，如果使用者没有正确使用这个镜像，那么容器应该会很快地失败，尽管这些相同的容器可能会在后面因为其他原因而失败。你可以组合启动脚本和容器重启策略来构建稳定的容器。但是容器重启策略并不是完美的解决办法。失败后正在等待重启的容器并不在运行，这意味着如果容器处于退避窗口（backoff）的中间，那么操作者不能够在这个容器中执行另外一个进程。这个问题的解决办法与确保容器不会停止运行有关。

8.4.2 初始化进程

基于 UNIX 的计算机通常会先启动一个初始化（init）进程。这个 init 进程负责启动所有其他的系统服务，让它们持续运行，然后负责关闭它们。使用一个 init 风格的系统来启动、管理、重启、关闭容器进程通常是恰当的。

经典的 init 进程使用一个文件或多个文件来描述被初始化后的系统的理想状态。这些文件描述了哪些程序需要被启动，什么时候启动它们，当它们停止时哪些操作需要被执行。使用一个 init 进程是启动多个程序、清理遗弃的进程、监控进程和自动化重启失败进程的最佳方式。

如果你决定采用这个模式，你应该将 init 进程设置为以应用为导向的容器的入口点。你可能要使用启动脚本来提前定义好环境变量，这取决于你使用的 init 程序。

举个例子，runit 程序不会传递环境变量到它启动的程序中。如果你的服务使用了一个启动脚本来验证环境变量，那么它将不会拥有其所需要的环境变量的访问权。解决这个问题的最佳方法可能是为 runit 程序使用一个启动脚本。这个脚本可能会将环境变量写入到一些文件中，因此你的应用的启动脚本就能够访问它们了。

现在已经有多个开源的 init 程序可用。功能全面的 Linux 发行版自带了功能全面的重量级 init 系统，比如说 SysV、Upstart 和 systemmd。Linux Docker 镜像，如 Ubuntu、Debian 和 Cent OS，都安装有自己的 init 程序，但是功能却不能立即使用。这些程序的配置复杂，且常对需要 root 权限的资源有重度的依赖。由于以上的原因，社区更倾向于使用轻量级的 init 程序。

主流的选择包括 runit、Busybox init、Supervisord 和 DAEMON 工具。这些工具都尝试解决类似的问题，但是每一个都有其各自的优缺点。使用 init 进程对于应用容器来说是最

佳实践，但是并不存在一个适合所有情况的完美 init 程序。

当考虑在容器中使用 init 程序时，你需要考虑以下因素：

- init 程序会将额外的依赖带入到镜像中
- 文件大小
- init 程序如何将信号量传递到它的子进程（如果它做了的话）
- 需要的用户权限
- 监控和重启功能（backoff-on-restart 特性是加分项）
- 僵尸进程清理功能

无论你选择哪个 init 程序，请确保你的镜像能够利用它来提高采用者对从你的镜像创建的容器的信心。如果容器需要更早的失败以便传递配置错误的信息，请确保 init 程序不会隐藏这些失败信息。

这些供你任意使用的工具构建的镜像能够产生持久耐用的容器。持久性并不代表安全，尽管持久容器的采用者可能相信它们能够尽可能地持续运行，但是他们不应该完全相信你的镜像直到这些镜像被加固。

8.5 加固应用镜像

作为一个镜像作者，提前预想到所有可能的使用场景是非常困难的。所以，尽可能加固你所构建的镜像。加固一个镜像就是塑造镜像，使得基于这个镜像创建的任何 Docker 容器的攻击面减少的过程。

加固应用镜像的一个通用策略就是最小化包含在其中的软件。按照常理推断，包含越少的组件就能够减少潜在漏洞的数量。更进一步说，构建最小化的镜像能够使得镜像的下载时间更短，并且能够帮助使用者更快地部署和构建容器。

除了上面提到的通用策略，还有三件事能够用来加固镜像。第一，你可以强制基于某个特定的镜像来构建镜像。第二，你能够确保无论容器如何基于你的镜像来构建，它们都会拥有一个合适的默认用户。第三，你应该去除 root 用户提权的通用途径。

8.5.1 内容可寻址镜像标识符

本书到目前为止，镜像 ID 都被设计成用来让镜像作者以对使用者透明的方式来更新镜像。一个镜像作者可以选择基于哪个镜像来工作，但层的透明机制使得基础镜像被安全检查后有没有被改动过变得不可知。从 Docker 1.6 开始，镜像 ID 包含了一个可选的摘要组件。

包含有摘要组件的镜像 ID 被称为内容可寻址镜像标识符（CAIID）。与简单地引入一个具体的，且可能被改动过的层不同，它引入的是一个包含特殊内容的特殊层。

现在，只要一个镜像处于一个版本为 2 的仓库中，那么镜像作者就能够强制从一个特定的，且未改动的起点开始构建。在标准的 tag 位置后面添加一个@符号，符号后面跟随的就是摘要。

使用 `docker pull` 命令，并且观察输出中标记为 `digest` 的行，你就能从一个远程仓库发现这个镜像的摘要。一旦你拥有了这个摘要，你可以将它作为 ID 在 Dockerfile 中的 `FROM` 指令中使用。举个例子，下面的例子使用了一个具体的快照 `debian:jessie` 来作为基础镜像：

```
docker pull debian:jessie
# Output:
# ...
# Digest: sha256:d5e87cfcb730...

# Dockerfile:
FROM debian@sha256:d5e87cfcb730...
...
```

无论这个 Dockerfile 被使用了多少次或在任何时间使用，它们都会使用被该 CAIID 标识的镜像作为基础镜像。这尤其有利于将基础镜像的更新整合到你的镜像中，并且有利于标识运行在你计算机上的某个具体的软件构造。

尽管这不能直接限制镜像的攻击面，但是使用 CAIID 能够防止镜像在你无意识的状态下被改动。接下来的两个例子将会解决镜像攻击面的问题。

8.5.2 用户权限

著名的容器逃离手段都依赖于获得容器中的管理员权限。第 6 章包含了用来加固容器的工具。那一章对用户管理进行了深入探讨，并且对 Linux 的 USR 命名空间进行了简要的讨论。本节涉及的内容包括为镜像建立合理的用户默认配置的标准实践。

首先，请理解当一个 Docker 用户创建容器时，他们总能够覆盖镜像的默认配置。因此，并不存在某个方法来防止容器以 root 用户运行。镜像作者所能做的就是创建其他的非 root 用户和建立一个非 root 默认用户和用户组。

Dockerfile 包含了一个 `USER` 指令，和 `docker run` 或 `docker create` 命令一样，它能够设置用户和用户组。这个指令本身已经在之前的 Dockerfile 入门指南介绍过了。本节关注于注意事项和最佳实践。

最佳实践和通用指南就是尽可能地削减特权。你可以通过 `USER` 指令在任何容器被创

建前，或通过一个在容器启动时的启动脚本来完成这个任务。对于镜像作者，难点在于如何确定何时才是恰当的最早时间。

如果你过早地削减特权，那么活动用户（active user）可能没有权限来完成 Dockerfile 的其他指令。举个例子，下面的 Dockerfile 将不能够被正确构建：

```
FROM busybox:latest
USER 1000:1000
RUN touch /bin/busybox
```

构建这个 Dockerfile 将会在第 2 步失败，错误信息类似于 touch: /bin/busybox: Permission denied。用户的改变明显地影响到了文件的访问权。在这个例子中，UID 1000 没有改动文件/bin/busybox 的所有者的权限。那个文件当前的所有者是 root。将第二行和第三行对换一下就能够修复这个问题。

第二个关于时间的考虑就是运行时所需要的权限和能力（capability）。如果镜像在运行时启动了一个需要管理员权限的进程，那么在这个行为发生前将用户改为非 root 用户是没有意义的。举个例子，任何需要系统端口范围（1～1024）的进程都需要被拥有管理员权限（至少需要 CAP_NET_ADMIN）的用户启动。思考当你尝试以非 root 用户使用 Netcat 绑定 80 端口时，会发生什么？将下面的内容复制到名为 UserPermissionDenied.df 的文件中：

```
FROM busybox:latest
USER 1000:1000
ENTRYPOINT ["nc"]
CMD ["-l", "-p", "80", "0.0.0.0"]
```

构建这个 Dockerfile 生产新镜像，并且使用这个镜像创建一个容器。在这个例子，UID 为 1000 的用户将会缺少需要的权限，导致命令失败：

```
docker build
    -t dockerinaction/ch8_perm_denied
    -f UserPermissionDenied.df

docker run dockerinaction/ch8_perm_denied
# 输出结果:
# nc: bind: Permission denied
```

像这种情况，在 Dockerfile 中修改默认用户并没有什么好处。反而任何你构建的启动脚本应该负责尽快地削减权限。最后一个问题是，应该使用哪一个用户？

Docker 目前缺乏对 Linux USER 命名空间的支持。这意味着容器中 UID 为 1000 的用

户和主机上 UID 为 1000 的用户是一样的。除了 UID 和 GID，其他的内容类似多台计算机，都是分离的。比如说，你的计算机上有一个 UID 为 1000 的用户，这个可能是你的用户名，但是在一个 BusyBox 容器中，UID 为 1000 的用户是默认用户。

最终，在 Docker 采用 USER 命名空间之前，镜像作者很难决定哪一个 UID/GID 适合使用。唯一你能够确定的事情就是使用常见的或系统级别的 UID/GID 是不合适的。尽管有了这条策略，使用原始的 UID/GID 数字依旧是艰难的。直接使用原始数字会降低脚本和 Dockerfile 可读性。因此，较为经典的做法是使用 RUN 指令创建镜像所要使用的用户和用户组。下面的内容就是 Postgres Dockerfile 的第二个指令：

```
# 首先，添加我们自己的用户和用户组，以此确保它们的 ID 一直存在，
# 无论添加了哪些依赖。
RUN groupadd -r postgres && useradd -r -g postgres postgres
```

这个指令简单地创建了一个 postgres 用户和用户组，它们的 UID 和 GID 都是自动生成的。这条指令在早期就放入到 Dockerfile 中，因此在重新构建的过程中它的内容总是能够被缓存，并且不管构建过程中其他被添加进来的用户，这些 ID 将会保持一致。然后这些用户和用户组就能够在 USER 指令中使用了。这么做能够使得默认用户更加安全。但是 Postgres 容器要求在启动期间提升权限，这个具体的镜像使用了一个类似 su 或 sudo 的程序在 postgres 用户下启动 Postgres 进程。这么做确保了这个容器中的进程不以管理员权限运行。

用户权限是构建镜像中更具有微妙区别的一方面。通用的规则是，如果你构建的镜像被设计用来运行某些特定的应用程序，那么默认配置应该尽可能地削减用户权限。

一个功能运行正确的系统应该拥有合理的默认配置来保证安全。记住，应用或代码很少是完美的，并且它们的恶意行为可能是故意为之。因此，你应该采用额外的步骤来减少镜像的攻击面。

8.5.3 SUID 和 SGID 权限

最后一个加固方法就是缓解 SUID 和 SGID 的权限。常见的文件系统权限（读，写，执行）仅仅是 Linux 定义的权限集合中的一部分。除了这些，还有两个特别有意思的权限：SUID 和 SGID。

这两个本质上来说是类似的。一个设置有 SUID 位的可执行文件总是会以它的所有者用户来执行。比如说，/usr/bin/passwd 的所有者是 root 用户，并且它拥有 SUID 权限。如果

一个非 root 用户，比如 bob，执行了这个程序，那么 bob 将会以 root 用户来执行程序。你可以通过构建以下的 Dockerfile 来亲身体验下以上内容：

```
FROM ubuntu:latest
# 设置 whoami 程序的 SUID 位
RUN chmod u+s /usr/bin/whoami
# 创建一个 example 用户，并且将它设置为默认用户
RUN adduser --system --no-create-home --disabled-password --disabled-login \
    --shell /bin/sh example
USER example
# 设置默认命令，比较容器用户和
# 执行 whoami 程序的有效用户
CMD  printf "Container running as:           %s\n" $(id -u -n) && \
     printf "Effectively running whoami as: %s\n" $(whoami)
```

当你创建完这个 Dockerfile，你需要使用下面命令来构建镜像，创建容器，并且执行容器中的默认命令：

```
docker build -t dockerinaction/ch8_whoami
docker run dockerinaction/ch8_whoami
```

输出结果类似以下内容：

```
Container running as: example
Effectively running whoami as: root
```

从默认命令的输出结果可以看出，尽管你用 example 用户来执行 `whoami` 程序，但是该程序却运行在 root 用户的上下文上。SGID 具有类似的功能。区别在于，SGID 会从拥有该程序的用户组的上下文执行，而不是程序所有者。

在你的基础镜像上运行一个快速的查找就能够知道拥有这些权限的文件有多少，分别是什么：

```
docker run -rm debian:wheezy find / -perm +6000 -type f
```

输出结果如下：

```
/sbin/unix_chkpwd
/bin/ping6
/bin/su
/bin/ping
/bin/umount
/bin/mount
/usr/bin/chage
```

```
/usr/bin/passwd
/usr/bin/gpasswd
/usr/bin/chfn
/usr/bin/newgrp
/usr/bin/wall
/usr/bin/expiry
/usr/bin/chsh
/usr/lib/pt_chown
```

下面的命令将会找出所有的 SGID 文件：

```
docker run -rm debian:wheezy find / -perm +2000 -type f
```

输出的列表数目减少了很多：

```
/sbin/unix_chkpwd
/usr/bin/chage
/usr/bin/wall
/usr/bin/expiry
```

每个列出的文件在这个具体的镜像中都拥有 SGID 或 SUID 权限，任何其中一个出现 Bug 都能够被用来 root 提权。好消息是，拥有 SUID 或 SGID 权限的文件通常在镜像构建过程中很有用，但是应用场景很少需要用到它们。如果你的镜像将被用来运行来自外部的软件，那么缓解这种提权的危险是最佳实践。

要解决这个问题，要么删除这些文件，要么将文件的 SUID 和 SGID 权限去除。这两种方法都能够减少镜像的攻击面。下面的 Dockerfile 指令会将镜像中所有文件的 SUID 和 SGID 权限都去除：

```
RUN for i in $(find / -type f ( -perm +6000 -o -perm +2000 ));
do chmod ug-s $i; done
```

加固镜像能够帮助用户构建加固的容器。尽管这些加固措施不能够防止用户故意构建脆弱的容器，但是以上讨论到的措施能够帮助到那些常见的，毫无戒心的用户。

8.6 小结

大多数 Docker 镜像都是从 Dockerfile 自动构建的。本章包含了 Docker 提供的构建自动化方法，还有 Dockerfile 的最佳实践。在学习下一章内容前，请确保你已经理解了本章的关键内容：

- Docker 提供了一个镜像自动化构建程序，它会从 Dockerfile 中读取指令来构建镜像。
- 每一个 Dockerfile 指令都会创建一个镜像层。
- 尽可能地合并指令，这样能够减少镜像的大小和层的数量。
- Dockerfiles 包含了能够设置镜像元数据的指令，比如说默认用户、开放端口、默认命令和入口点。
- 其他的 Dockerfile 指令能从本地文件系统或远程目录复制文件到构建的镜像中。
- 下游的构建会继承上游 Dockerfile 中 `ONBUIILD` 指令设置的构建触发。
- 启动脚本应该用来在启动主要应用前验证容器的执行上下文。
- 一个有效的执行上下文应该拥有正确的环境变量集合，网络依赖的可用性和一个合适的用户配置。
- init 程序能够被用来启动多个进程、监控这些进程、清除孤立的进程和转发信号量到子进程中。
- 应该使用内容可寻址镜像标识符，创建非 root 的默认用户和禁止或去除任何带有 SUID 或 SGID 权限的可执行文件来加固镜像。

第 9 章　公有和私有软件分发

本章介绍

- 选择一个项目分发方法
- 使用托管基础设施
- 运行和使用你自己的 Registry
- 理解镜像手动分发工作流程
- 分发镜像资源

你可以通过自己编写软件、定制化或者从互联网拉取拥有自己的镜像，但是，如果没有人可以安装镜像，那么它有什么好处呢？Docker 不同于其他容器管理工具的原因就是因为它提供了镜像分发功能。

有几种方法可以得到你的镜像，本章将探讨镜像分发模式，并提供了一个框架，可以为你的项目构建或选择一个或者多个镜像。

托管 Registry 提供了公有和私有仓库，以及自动构建工具。运行一个私有的 Registry 让你可以隐藏和定制镜像分发基础设施。分发工作流程的深度定制也许会要求你放弃 Docker 官方的镜像分发设施而构建你自己的镜像分发设施。有些系统甚至可能会放弃把镜像作为分发单元，而是分发镜像源代码。

本章将教你如何选择和使用一个分发镜像到外部或者工作区域的方法。

9.1 选择一个分发方法

选择一个分发方法最困难的事情还是针对你的情形选择适当的方法，为了帮助解决这个问题，在本章中提出的每个方法都用同一组选择标准来审查。

首先要认识到 Docker 镜像分发是没有万能的解决方案的。分发需求变化的原因有很多，也有很多种方法。每种方法的核心都是基于 Docker 工具，所以总是可以用最小的代价从一个方法迁移到另外一个方法。最好的开始方式是在一个较高的层面来审查各种选项。

9.1.1 分发选项图谱

镜像分发的选项图谱是一个弹性和复杂性的平衡，提供了最多弹性的方法往往是使用起来最复杂的，而那些最易于使用的方法通常也是最受限的。如图 9-1 所示的完整的图谱。

图谱中的方法范围涵盖了从托管 Registry 比如（Docker Hub）到完全自定义的分发架构或者源代码分发方法。本章将比其他章节更详细地介绍这些主题，其中需要特别关注私有 Registry，因为它们会提供两者之间最佳的平衡。

拥有一个选择的图谱说明了你的选择范围，但你需要一组一致的选择标准，以确定你应该使用哪一个。

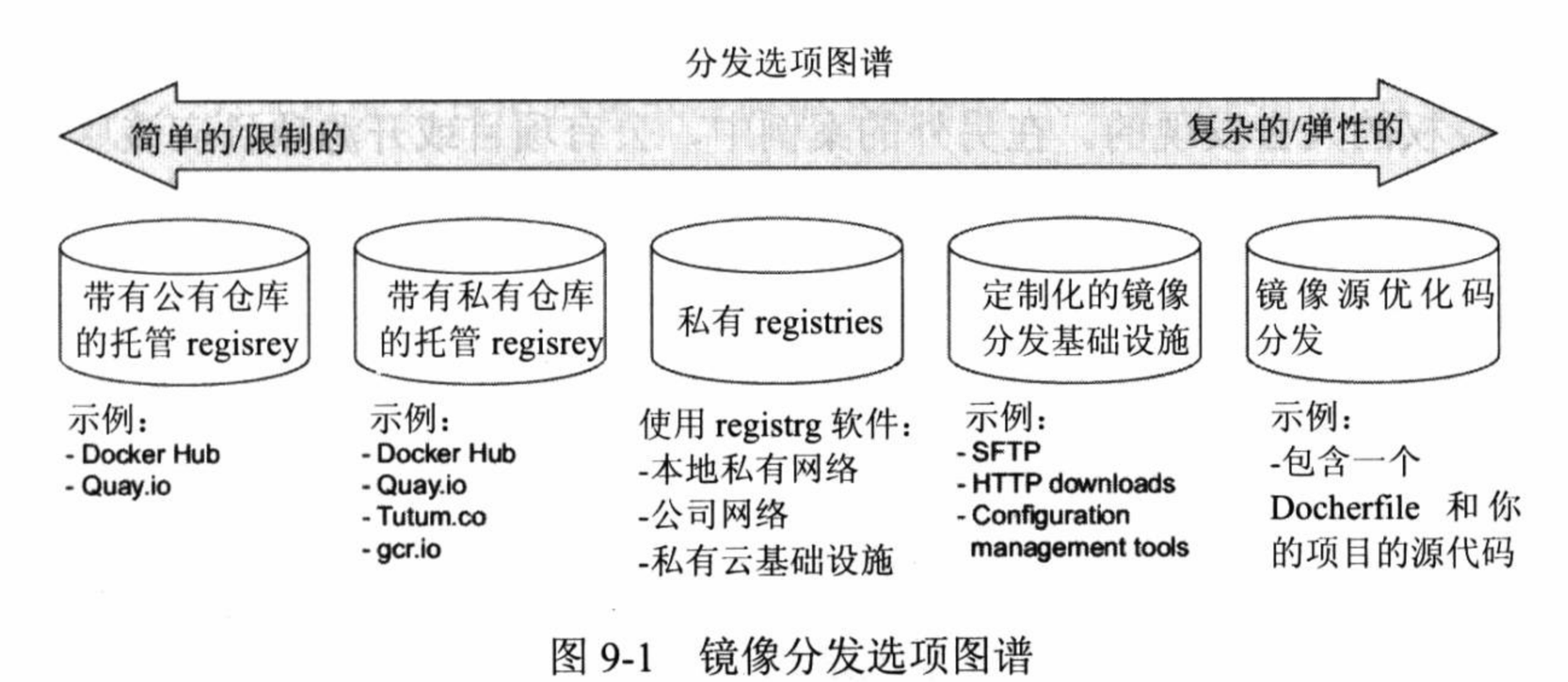


图 9-1 镜像分发选项图谱

9.1.2 选择标准

从众多的选项中为你的需求选择最好的分发方法似乎是令人生畏的。在这种情况下你应该花时间去理解选项，为进行选择而识别标准，避免做出快速决定或解决方案的冲动。

以下为已确认的基于整个图谱差异和常见业务问题的选择标准，当做出决定时，要考

虑的每一个标准在你的情形中是如何的重要：

- 成本
- 可见性
- 传输速度和带宽开销
- 生命周期控制
- 可用性控制
- 访问控制
- 产品完整性
- 产品保密性
- 必要的专业知识

本章其余的相关部分会涵盖每个分发方法是如何叠加起来实现这些标准的。

成本

成本是最明显的标准，成本的图谱分布范围可以从免费到非常昂贵，甚至“很复杂”。越低的成本通常是更好的，但是成本通常是最弹性的标准。例如，大多数人如果情况要求的话，会将成本与产品保密性挂钩。

可见性

可见性是下一个最明显的分发方法标准。如果可能的话秘密项目或内部工具，应该是很难让未经授权的人去发现的。在另外的案例中，公有项目或开源项目应该尽可能保持可见来推广采用。

传输

传输速度和带宽开销是下一个最弹性的标准。各种方法都会影响文件大小和镜像安装速度，比如利用镜像层、并行下载、镜像预编译和那些使用合并镜像文件或者依赖部署时的镜像构建。高传输速度或低安装延迟对于使用即时部署来服务同步请求的系统是至关重要的，而对于开发环境或者异步处理系统是相反的。

生命周期

生命周期控制是一个业务问题，不只是一个技术问题。托管分发方法取决于人们或者公司的业务问题。一位高管面对使用一个托管 Registry 的选项也许会问：“如果他们破产或者偏离了仓库托管业务，会发生什么？”，问题也会缩减到“在我们之前第三方的业务需求是否会改变”，如果这是一个你关心的问题，那么生命周期控制是很重要的。Docker 使得

切换这些方法非常容易，其他的标准（比如必要的专业知识或者成本）实际上可能胜过这个问题。由于这些理由，生命周期控制是另一个更弹性的标准。

可用性

可用性控制是能够控制有关你的仓库可用性问题的解决方案。托管解决方案没有提供可用性控制。如果你是一个付费用户的话，企业通常在可用性方面提供一些服务水平协议（SLA），但没有什么途径可以直接解决这个问题。在图谱的另一端，私有 Registry 或自定义解决方案把控制和责任都交到你自己手中。

访问控制

访问控制保护你的镜像不被修改或者未经授权方的访问。存在有不同程度的访问控制，一些系统只提供修改一个特定仓库的访问控制，而其他一些系统提供了整个 Registry 的过程控制。尽管如此，其他系统可能包括付费墙或数字版权管理控制。项目通常有特定的由产品或者业务决定的访问控制需求。这使得访问控制成为最缺少弹性，但也是最需要考虑的需求。

完整性

产品的完整性和保密性都带来了较少弹性和更技术化的图谱。产品完整性是可信赖的，确保文件和镜像的一致性。违反完整性可能包括中间人攻击，攻击者拦截镜像的下载，并用自己的内容替换。另外可能还包括恶意或者破解 Registry 来欺骗返回的有效载荷。

保密性

产品的保密性是一种常见的公司开发商业机密或专有软件的需求。举例来说，如果你使用 Docker 分发加密材料，那么保密将是一个大问题。产品的完整性和保密性是不同的图谱特性。总的来说，开箱即用的分发安全特性不会提供最佳的机密性或完整性。如果这是你的一个需求，需要一个信息安全专业人员来实施和复审解决方案。

选择一个分发方法时要考虑的最后一件事就是必要的专业知识水平。使用托管方法可以非常简单，只需要多一点的对于工具机制的理解就可以了。构建自定义的镜像或者镜像资源分发管道需要一系列相关技术的专业知识。如果你没有专业知识或无法找到具备这些知识的人，那么使用更复杂的解决方案将是一个挑战。在这种情况下，你可以协调额外成本的差距。

通过这组强大的选择标准，你就可以开始学习和评估不同的分发方法，最好的起点是在最左边的图谱，那里有托管 Registry。

9.2 通过托管 Registry 发布

提醒一下，Docker 的 Registry 服务可以使得仓库可以通过 Docker pull 命令进行访问，Registry 托管了仓库，分发你的镜像最简单的方法是使用托管 Registry。

一个托管 Registry 是一个由第三方供应商拥有和运营的 Docker Registry 服务。Docker Hub、Quay.io、Tutum.co 和 Google Container Registry 都是托管 Registry 供应商的例子。在默认情况下，Docker 将镜像发布到 Docker Hub，Docker Hub 和大多数其他托管 Registry 都提供了公有和私有的 Registry，如图 9-2 所示。

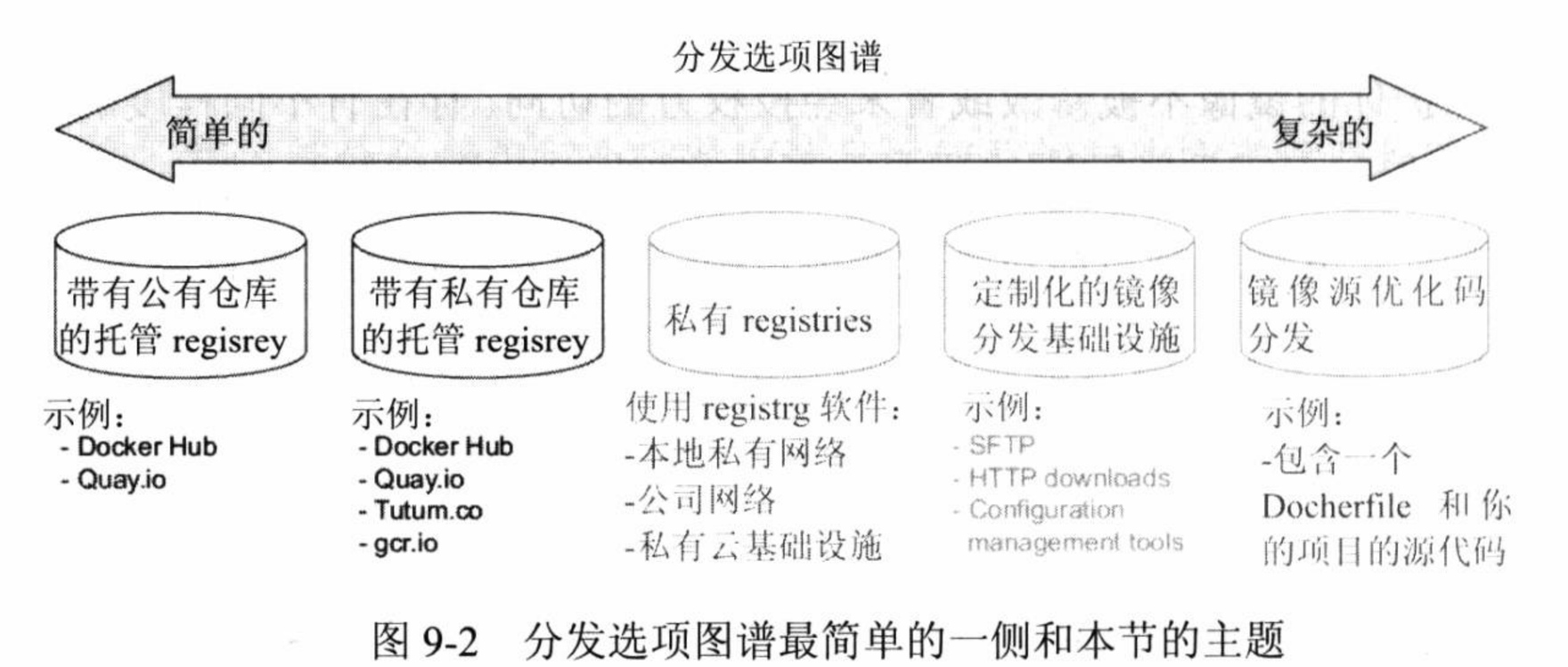


图 9-2 分发选项图谱最简单的一侧和本节的主题

在本书中使用的示例镜像是通过托管在 Docker Hub 和 Quay.io 的公共仓库来分发的，在本章结尾你将会理解怎么样使用托管 Registry 来发布你自己的镜像以及这些托管 Registry 是如何符合选择标准的。

9.2.1 通过公有仓库发布：你好！Docker Hub

在托管 Registry 上开始使用公有仓库最简单的方法就是将镜像推送到自己在 Docker Hub 的仓库。要做到这一点，你所需要的是一个 Docker Hub 账号和一个要发布的镜像，如果你还没有，现在就注册一个 Docker Hub 账号。

一旦你有了账号，你就需要创建一个要发布的镜像。创建一个新的名为 HelloWorld.df 的 Dockerfile，添加如下指令：

```
FROM busybox:latest
CMD echo Hello World
```

来自 HelloWorld.df 文件

第 8 章讲述了 Dockerfile 指令，提醒一下，FROM 指令告诉 Docker 镜像构建器从哪个

现有的镜像开始构建新的镜像，CMD 指令为新的镜像设置默认命令。这个镜像创建的容器将会显示“Hello World”然后退出。使用接下来的命令来构建你的新镜像：

```
docker build \
    -t <insert Docker Hub username>/hello-dockerfile \
    -f HelloWorld.df \
    .
```

插入你的用户名

务必要将上述命令中的用户名替换为你自己的 Docker Hub 用户名，授权访问和修改仓库都是基于 Docker Hub 上的仓库名称的用户名部分的。如果你不是用自己的用户名创建一个仓库的话，你将无法发布镜像。

使用 docker 命令行工具在 Docker Hub 上发布一个镜像需要你在客户端与 Docker Hub 间建立一个已认证的会话，你可以用 login 命令：

```
docker login
```

这个命令会提示你输入用户名、电子邮件地址和密码，每一个都可以作为参数传递给命令，使用--username、--email 和--password 标记即可。当你登录后，Docker 客户端维护一个你自己的证书映射表，这些证书在一个文件中进行验证。它将专门存储用户名和身份认证 token，而不是你的密码。

一旦你登录了后，你就可以推送你的仓库到托管 Registry，使用 docker push 命令：

```
docker push <insert Docker Hub username>/hello-dockerfile
```

插入你的用户名

运行这个命令应该输出如下：

```
The push refers to a repository
[dockerinaction/hello-dockerfile] (len: 1)
7f6d4eb1f937: Image already exists
8c2e06607696: Image successfully pushed
6ce2e90b0bc7: Image successfully pushed
cf2616975b4a: Image successfully pushed
Digest:
  sha256:ef18de4b0ddf9ebd1cf5805fae1743181cbf3642f942cae8de7c5d4e375b1f20
```

命令的输出包括上传状态和生成的仓库内容指纹。推送操作将在远端 Registry 创建仓库，上传每一个新层，然后创建相应的标签。

一旦推送操作完成，你的公共仓库将会对外界可用，可以通过搜索你的用户名和你的新仓库验证是否如此。例如使用以下命令找到属于用户 dockerinaction 的例子：

```
docker search dockerinaction/hello-dockerfile
```

将用户名 dockerinaction 替换为你自己的，在 Docker Hub 找到你的新仓库。你也可以登录到 Docker Hub 网站查看你的仓库，然后查找和修改你的新仓库。

用 Docker Hub 分发你的第一个镜像，你应该考虑这种方法如何符合选择标准，如表 9-1 所示。

表 9-1 公有托管仓库的性能

标 准	评级	备 注
成本	最佳	如果你正在使用一个共有的 GitHub 仓库，那么是没有成本的
可见性	最佳	GitHub 对于开源工具是一个非常醒目的位置，它提供了优秀的社交和搜索组件，可以简单地发现项目
传输速度和大小	好	通过分发镜像源代码，你可以利用其他 Registry 的基本层，这样做可以减少运输和存储的负担。GitHub 还提供一个内容分发网络（CDN），CDN 使得世界各地的客户可以低网络延迟来访问 GitHub 项目
可用性控制	最糟糕	依靠 GitHub 或者其他的托管版本控制供应商消除了任何的可用性控制权
生命周期控制	糟糕	虽然 Git 作为一种流行的工具应该有一段时间了，通过集成 GitHub 或者其他托管版本控制供应商，你放弃了任何的生命周期控制权
访问控制	好	GitHub 或其他托管版本控制供应商针对私有仓库提供了访问控制工具
产品完整性	好	这个解决方案对于作为构建流程一部分产生的镜像或者克隆到客户端机器的源代码没有提供完整性保证，但完整性是版本控制系统的重点。任何完整性问题都应该通过标准的 Git 流程来显现出来并很容易地恢复
保密性	最糟糕	公有项目不提供源代码保密性
必要的经验	好	镜像生产者和消费者都需要熟悉 Dockerfile、Docker 构建器和 Git 工具

基于托管 Registry 的公有仓库对于开源项目所有者或者那些仅仅刚刚使用 Docker 的用户是最佳的选择。人们仍然应该对于从互联网下载并运行的软件持怀疑态度，所以不公开源代码的公有仓库要获得用户信任是很困难的。托管（受信的）构建可以在一定程度上解决这个问题。

9.2.2 使用自动构建发布公有项目

几个不同的托管 Registry 都提供自动化构建，自动化构建是由 Registry 供应商使用你可用的镜像源代码来构建镜像。镜像消费者对于这些构建有着更高程度的信任，因为 Registry 所有者是从可审核的源代码构建镜像的。

使用自动构建分发你的镜像需要两个组件：一个托管镜像仓库和一个托管 Git 仓库，你的镜像源代码就是由 Git 仓库发布的。Git 是一个流行的分布式版本控制系统，一个 Git 仓库存储了你的项目的变化历史。尽管分布式版本控制系统（如 Git）没有集中式架构，一些受欢迎的公司提供 Git 仓库托管服务。Docker Hub 为自动构建功能集成了 GitHub.com 和 Bitbucket.org。

这两种托管 Git 仓库工具都提供所谓的 Webhook。在这种背景下，`Webhook` 可以用来给你的镜像仓库通知你的 Git 仓库源代码发生变化了。当 Docker Hub 接收到 Git 仓库的

Webhook 时，将会在 Docker Hub 仓库里启动一个自动化构建，如图 9-3 所示。

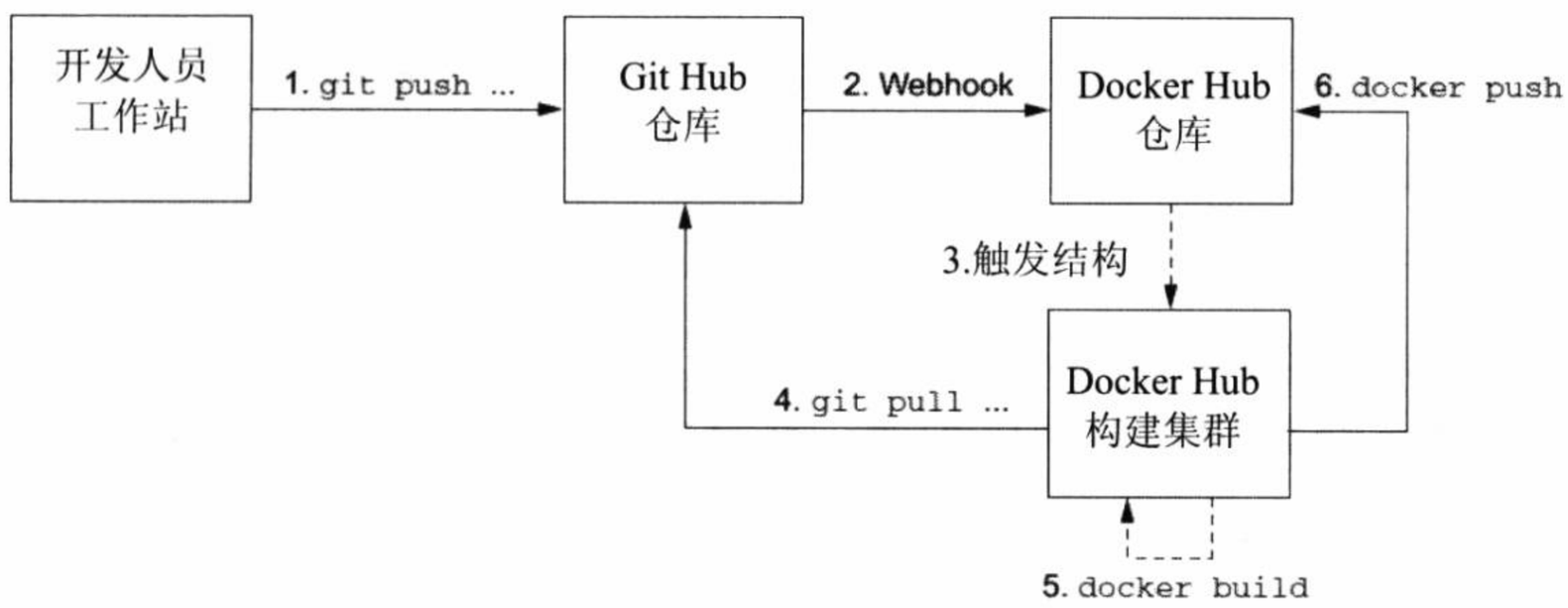


图 9-3 Docker Hub 自动化构建工作流程

自动化构建过程将为您的项目从你注册的 Git 仓库拉取源代码，包括 Dockerfile。Docker Hub 构建集群使用 `docker build` 命令从那些源代码中构建一个新的镜像，并按照仓库配置来打上标签，然后再推送到你的 Docker Hub 仓库。

创建一个 Docker Hub 自动化构建

下面的例子将指导你完成所需的步骤把自己的 Docker Hub 设置为自动构建，这个示例使用了 Git。整本书写的都是有关 Git 的，所以我们不会过多地涉及详细内容。如今 Git 随好几个操作系统一起发行了，但是如果你的计算机上没有安装或者你需要一般的帮助，请访问网站 https://git-scm.com。对于本示例，Docker Hub 和 GitHub 账号都是需要的。

登录你的 GitHub 账号，然后创建一个新的仓库，命名为 hello-docker，确保这个代码库是公有的。不要使用许可证或者.gitignore 文件初始化这个代码库。一旦在 GitHub 上创建了这个代码仓库，回到你的命令行终端并且创建一个新的名为 hello-docker 的工作目录。

创建一个新的命名为 Dockerfile 的文件，并且包含如下几行：

```
FROM busybox:latest
CMD echo Hello World
```

这个 Dockerfile 将会产生一个简单的 Hello World 镜像，为了将这个镜像放到一个新的 Docker Hub 仓库里，你需要做的第一件事就是将其添加到你的 Git 仓库。接下来的 Git 命令将会创建一个本地仓库并添加 Dockerfile 到其中，提交这个变动，然后推送这个变动到你的 GitHub 仓库。一定要用你的 GitHub 用户名替换掉<your username>：

```
git init
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
git remote add origin \
    https://github.com/<your username>/hello-docker.git
```

使用你的 email 地址

使用你的用户名

使用你的 GitHub 用户名

然而不要添加或提交你的文件到仓库。在你推送你的工作到 GitHub 之前，你应该在 Docker Hub 上创建一个新的自动化构建和仓库。你必须通过网站 https://hub.docker.com 执行这一步骤。一旦你登录后，在网页头部单击创建按钮，从下拉式菜单中选择自动化构建，该网站将指导您完成自动化构建的设置。

步骤包括 GitHub 身份认证和授予 Docker Hub 受限地访问你的账号的权限。访问权限是必需的，这样 Docker Hub 就可以找到你的 GitHub 仓库并注册合适的 Webhook。接下来，你将被提示使用 GitHub 仓库用于自动化构建。选择之前你创建的 hello-docker 仓库，一旦你完成了创建向导，你将会直接来到仓库页面，现在回到你的命令行终端，添加并推送你的工作到 GitHub 仓库。

```
git add Dockerfile
git commit -m "first commit"
git push -u origin master
```

执行最后一个命令时，你可能会被提示 Github 登录凭证，在提供之后，你的工作将被上传到 GitHub，你可以在线查看 Dockerfile。现在你的镜像源代码在 GitHub 上可用了，应该由你的 Docker Hub 仓库触发一个构建。让我们返回仓库页面并单击构建详情选项卡。你将会看到一个从你最新推送到 GitHub 仓库的触发而来的构建列表，一旦完成之后，返回命令行查找你的仓库：

```
docker search <your username>/hello-docker
```

插入你的 Docker Hub 用户名

自动化构建是镜像消费者的首选，在大多数情况下简化了镜像维护。有时候你不想让你的源代码提供给公众，好消息是，大多数托管仓库供应商提供了私有仓库。

9.2.3 私有托管仓库

私有仓库从操作和产品角度来看类似于公有仓库，大多数 Registry 都提供这两个选项，通过他们的网站做任何配置的差异都是很小的。因为 Docker Registry API 并没有区分这两种类型的仓库，提供这两种类型的 Registry 供应商通常需要你通过他们的网站、应用程序或者 API 提供私有 Registry 服务。

用于私有仓库与那些公有仓库的工作工具是一模一样的，只有一个例外。在你可以使用 `docker pull` 或者 `docker run` 命令从一个私有仓库安装一个镜像之前，你需要从托管仓库获得身份认证。为此，你将使用 `docker login` 命令，就像如果你正使用 `docker push` 命令上传镜像一样。

以下命令将提示你对由 Docker Hub、quay.io 以及 tutum.co 提供的 Registry 进行身份认证，在创建账号和身份认证之后，你将会在三个 Registry 里拥有对于你的公有和私有仓库的完全的权限，代码体子命令可以带上一个可选的服务参数：

```
docker login
# Username: dockerinaction
# Password:
# Email: book@dockerinaction.com
# WARNING: login credentials saved in /Users/xxx/.dockercfg.
# Login Succeeded

docker login tutum.co
# Username: dockerinaction
# Password:
# Email: book@dockerinaction.com
# WARNING: login credentials saved in /Users/xxx/.dockercfg.
# Login Succeeded

docker login quay.io
# Username: dockerinaction
# Password:
# Email: book@dockerinaction.com
# WARNING: login credentials saved in /Users/xxx/.dockercfg.
# Login Succeeded
```

在你决定将私有的托管仓库作为你的分发解决方案前，考虑如何满足你的选择标准；如表 9-2 所示。

表 9-2 私有托管仓库的性能

标　准	评级	备　注
成本	最佳	私有仓库的成本通常会随着你需要的仓库数量而伸缩，成本计划通常范围是从 5 个仓库每月几美元到 50 个仓库每月 50 美元，在这里存储和每月虚拟服务器托管的价格压力是驱动因素。需要超过 50 个仓库的用户或组织可能会发现它更适合运行自己的私有 Registry
可见性	最佳	私有仓库是定义为私有的，它们通常被排除在仓库索引之外，并且需要在 Registry 确认仓库存在之前进行身份认证。私有仓库不适合发布一些软件或者分发开源镜像。相反，他们对于小型的私有项目或者组织是很棒的工具，因为他们不想承担自己运行 Registry 的开销
传输速度和大小	更好	任何托管 Registry 比如 Docker Hub 都会减少用于传输镜像的带宽并且启用镜像的分层并行传输。忽视通过互联网传输文件的潜在延迟，托管 Registry 应该对于其他非 Regisgry 解决方案可以完美执行
可用性控制	最糟糕	托管 Registry 没有提供任何可用性控制，然而与共有仓库不同的是，使用私有仓库使你成为一个付费客户，这些付费用户可能有更强的 SLA 保证或客户服务支持
生命周期控制	好	你对于托管 Registry 没有生命周期控制权，但 Registry 将符合所有的 Docker Registry API，并且从一个托管主机迁移到另一个应该是一个低成本的操作

续表

标　准	评级	备　注
访问控制	更好	读取和写入私有仓库的权限于已授权用户
产品完整性	最佳	有理由期望所有托管 Registry 支持 Registry V2 和内容可寻址镜像
保密性	最糟糕	尽管这些仓库提供了隐私，它们还是不适合存储明文的机密材料或者商业秘密代码，虽然 Registry 对于请求的资源需要用户身份认证和授权，这些机制有几个潜在的问题。供应商可能使用弱凭证的存储、丢失证书或者在闲时留下未加密的组件。最后，你的机密材料不应该被 Registry 供应商的员工访问
必要的经验	最佳	就像共有仓库，使用托管 Registry 的私有仓库只要求你最低程度地熟悉 Docker 并可以通过网站设置一个账户，这个解决方案是任何 Docker 用户触手可及的

个人和小型团队会在私有托管仓库里发现最实用的工具，他们的低成本和基本身份认证功能对于有着最小安全需求的低成本项目或者私人项目是非常友好的。大公司或者需要更高程度的保密性以及合适预算的项目，他们的需求可以通过运行自己的私有 Registry 来更好地满足。

9.3 私有 Registry 介绍

当你有一个困难的可用性控制、生命周期控制或者安全需求时，那么运行一个私有 Registry 可能是你最好的选择。这样做的话，你可以不用牺牲与 Docker 的 pull 和 push 机制的互操作性，就可以获得控制权，或者不会增加对于环境的学习曲线。人们可以像与托管 Registry 完全一样与私有 Registry 进行交互。

Docker Registry 软件（称之为 Distribution）是开源软件并且按照 Apache 2 许可证分发，这款软件的可用性和宽容的许可证让运行自己的 Registry 的工程成本非常低廉。可以通过 Docker Hub 运行，易于在非生产环境下使用。如图 9-4 所示，私有 Registry 居于分发选项图谱的当中位置。

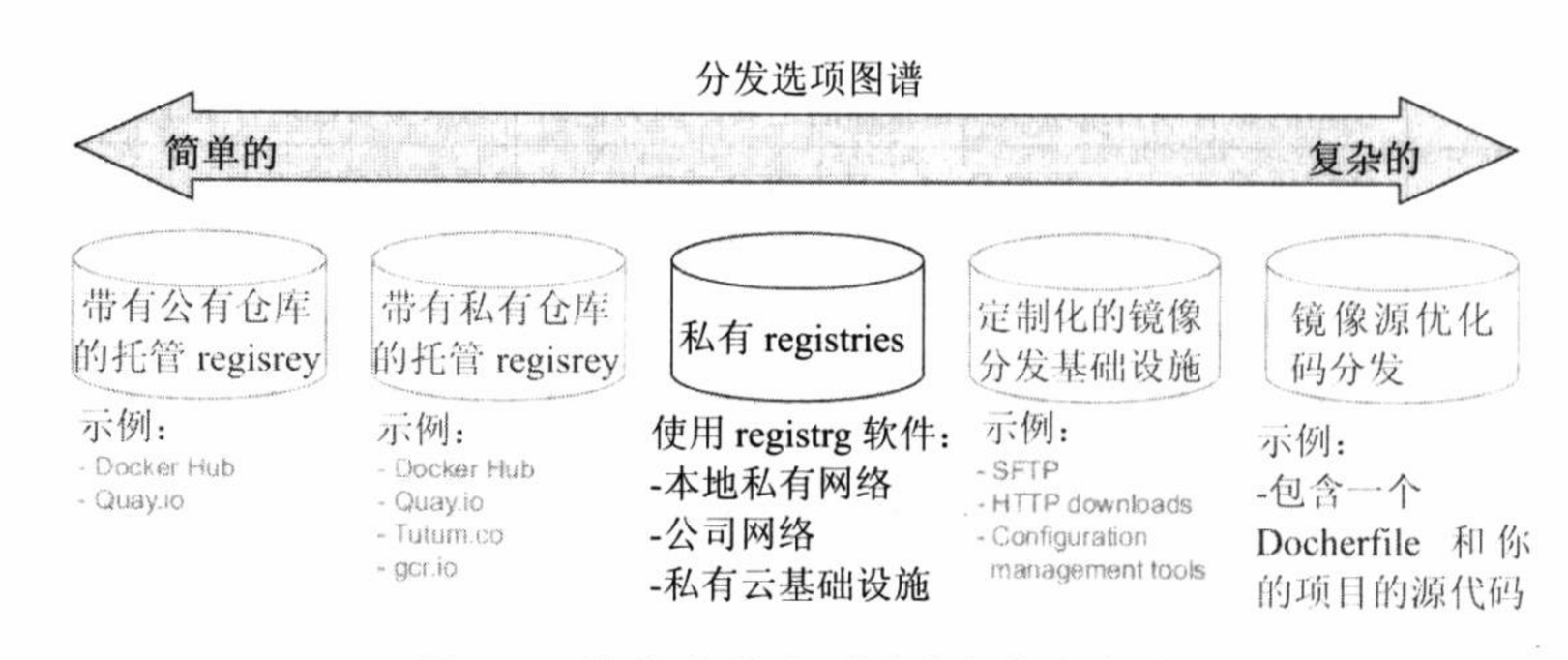


图 9-4 镜像分发选项图谱中的私有 Registry

如果你有类似如下的特殊的基础设施使用案例，那么运行一个私有 Registry 是很好的：

- 区域镜像缓存
- 团队特定的镜像分发位置或可见性
- 环境或者部署特定阶段的镜像池
- 公司的镜像审批流程
- 外部镜像的生命周期控制

在决定这是你最好的选择之前，考虑这些选择标准的成本详细说明，如表 9-3 所示。

表 9-3 私有 Registry 的性能

标 准	评级	备 注
成本	好	至少，一个私有 Registry 增加了硬件的开销（虚拟或其他），支持费用以及失败风险，但社区通过构建开源软件已经投资了大部分需要部署私有 Registry 的项目。成本的伸缩将比托管 Registry 具有更多的维度，托管 Registry 的成本随着原生仓库数量伸缩，然而私有 Registry 的成本伸缩是随着事务和存储使用率的，如果你建立了一个具有高事务率的系统，你需要按需扩大 Registry 主机的数量，同样的，Registry 服务于一些小镜像，比相同数量的大镜像会有低得多的存储成本。
可见性	好	私有 Registry 的可见性一样可由你来决定，但即使你拥有的 Registry 对全世界开放，仍然会比像 Docker Hub 这样的广受欢迎的 Registry 曝光度要少得多
传输速度和大小	最佳	任何客户端与任何 Registry 之间的延迟会根据网络上这些节点之间的距离、网络速度以及 Registry 上的拥堵而有不同，由于那些变量的差异，私有 Registry 可能会快于或者慢于托管 Registry。但是私有 Registry 将会吸引大多数人或者机构来用于内部基础设施。消除互联网或者数据中心网络间的依赖会成比例地改善延迟。因为这个解决方案是使用 Docker Registry 的，它将共享与托管 Registry 解决方案相同的并行性传输收益
可用性控制	最佳	你作为 Registry 所有者拥有完全的可用性控制权
生命周期控制	最佳	你作为 Registry 所有者拥有完全的生命周期控制权
访问控制	好	Registry 软件不包含任何身份认证或者开箱即用的授权功能，但是实现这些功能可以获得最小的工程操作开销
产品完整性	最佳	Registry API V2 支持内容可寻址的镜像，并且这款开源软件支持可插拔的存储后端。对于额外的完整性保护，你可以在网络上强制使用 TLS，并且使用后端静止存储
保密性	好	私有 Registry 在分发选项图谱上是第一个适合存储商业机秘密或者机密材料的解决方案，你控制身份认证和授权机制，你也控制网络以及在传输中的安全机制。最重要的是,你控制静止存储。这是在你的权力范围之内确保系统以这样一种保密的方式配置。
必要的经验	好	开始和运行本地 Registry 只需要基本的 Docker 经验，但运行和维护一个高可用的生产环境 Registry 需要多个技术领域的经验，具体的设置取决于你想利用的特性。一般来说，你要熟悉搭建 Nginx 代理、LDAP 或者 Kerberos 提供的身份认证，以及 Redis 缓存

从托管 Registry 到私有 Registry 的最大权衡是获得弹性和控制能力，同时要求更深度

和广度的工程经验来构建和维护解决方案。本节的其余部分讨论你需要实现的，除了最复杂的 Registry 部署设计，并且在您的环境中突显出定制化的机会。

9.3.1 使用 Registry 镜像

不管你这样做的理由是什么，开始使用 Docker Registry 软件是很容易的。在 Docker Hub 的名为 registry 的仓库有可用的分发软件，可以用一个单独的命令在容器中启动一个本地 registry：

```
docker run -d -p 5000:5000 \
        -v "$(pwd)"/data:/tmp/registry-dev \
        --restart=always --name local-registry registry:2
```

Docker Hub 分发的镜像可以在运行 Docker Daemon 客户端的计算器上进行不安全访问的配置。当你启动 Registry 时，你可以像其他 Registry 一样使用 `docker pull`、`run`、`tag` 和 `push` 命令。在这种情况下，Registry 位置是 localhost:5000，你的系统架构现在应该匹配如图 9-5 所示内容。

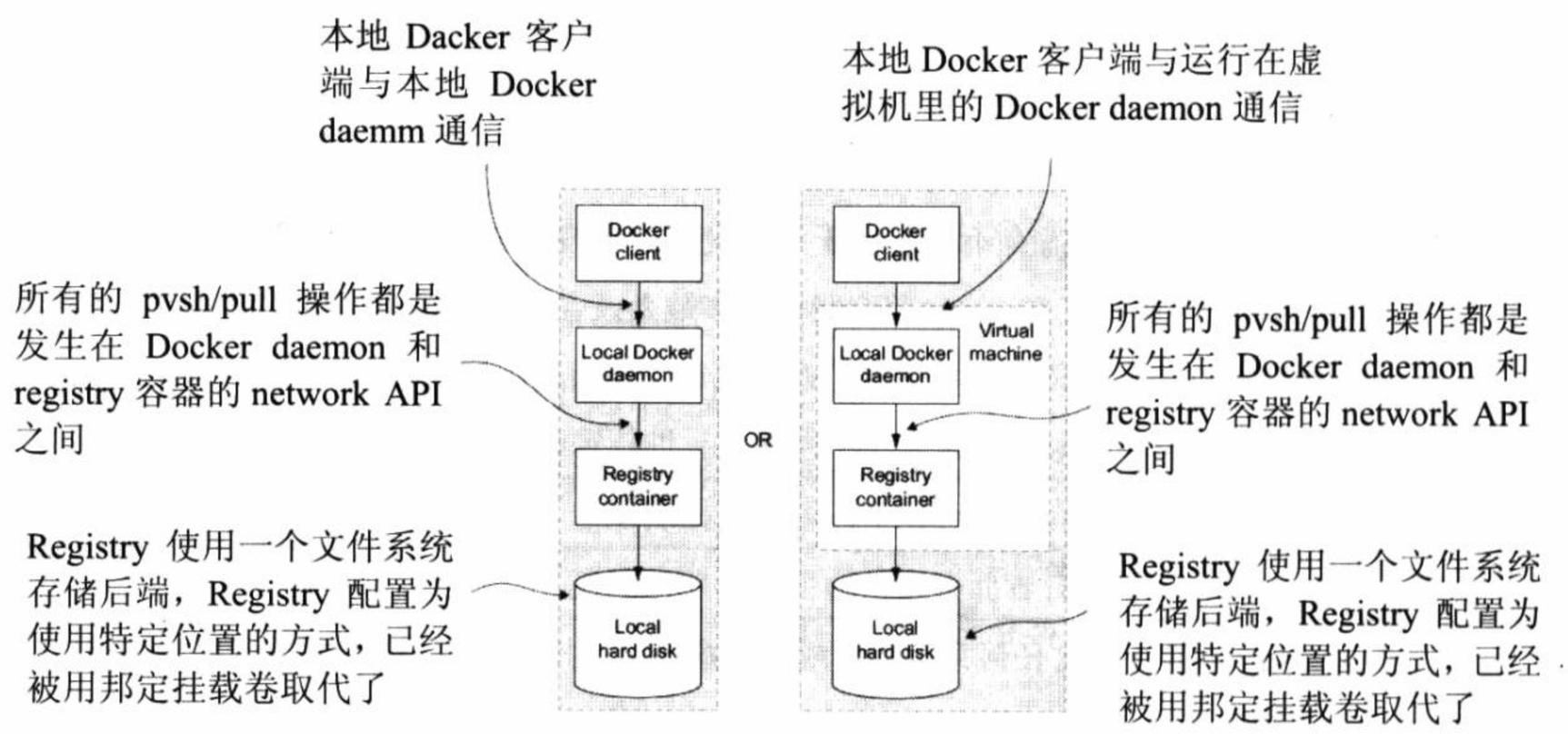


图 9-5 Docker 客户端、Daemon、本地 Registry 和本地存储之间的交互

需要在他们的外部镜像依赖关系上有着严格的版本控制的公司将会从外部资源（如 Docker Hub）上拉取镜像并且复制到他们自己的 Registry。如果想了解你的 Registry 是怎么工作的，可以考虑一下将镜像从 Docker Hub 复制到你的新 Registry 的工作流程：

从 Docker Hub 拉取 demo 镜像

```
docker pull dockerinaction/ch9_registry_bound

docker images -f "label=dia_excercise=ch9_registry_bound"

docker tag dockerinaction/ch9_registry_bound \
    localhost:5000/dockerinaction/ch9_registry_bound
docker push localhost:5000/dockerinaction/ch9_registry_bound
```

通过标签过滤器验证镜像可发现

推送 demo 镜像到你的私有 registrg

在运行这四个命令时，你从 Docker Hub 复制一个示例仓库到你的本地 Registry。如果你从启动 Registry 的同一位置开始执行这些命令，你会发现新创建的数据子目录包含新的 Registry 数据。

9.3.2 从 Registry 使用镜像

与 Docker 生态系统的紧密集成可以让你觉得就像正在使用已经安装在你计算机上的软件一样。当网络延迟消除后，就如使用一个本地 Registry 一样，甚至感觉不到是在分布式组件上工作。因此，推送数据到一个本地仓库的运用不是很令人兴奋。

下一组的命令应该给你留下深刻印象，你正在使用一个真正的 Registry。这些命令将从你的 Docker Damon 本地缓存删除示例仓库来展示它们消失了，然后重新从你的个人 Registry 安装：

```
docker rmi \
    dockerinaction/ch9_registry_bound \
    localhost:5000/dockerinaction/ch9_registry_bound

docker images -f "label=dia_excercise=ch9_registry_bound"

docker pull localhost:5000/dockerinaction/ch9_registry_bound

docker images -f "label=dia_excercise=ch9_registry_bound"

docker rm -vf local-registry
```

移除标记的引用

再次从 registry 拉取

演示镜像又回来了

清除本地 registry

你可以在本地尽可能多地使用这个 Registry，但不安全的默认配置将阻止远程 Docker 客户端使用你的 Registry（除非他们明确允许不安全的访问）。这是在生产环境中为数不多的需要在部署 Registry 前需要解决的问题。第 10 章将会深入地探讨 Registry 软件。

这是最灵活的分发方法，涉及了 Docker Registry，如果你需要更好的流量控制、存储和组件管理，你应该考虑在一个手动的分发系统中直接处理镜像。

9.4 镜像的手动发布和分发

镜像是文件，你可以像任何其他文件那样分发。常见的可供软件下载的有网站、文件传输协议（FTP）服务器、企业存储网络或通过 P2P 网络。你可以使用任何一种渠道用于镜像的分发，在你知道镜像收件人的情况下甚至可以使用电子邮件或 USB Key。

当你将镜像作为文件时，你只能使用 Docker 来管理本地镜像和创建文件，所有其他问题都留给你来解决。这个功能的缺失使得镜像的手动发布和分发成为第二个最灵活的，但

是也是非常复杂的分发方法。本节讨论定制镜像分发基础设施，如图 9-6 所示的选项图谱。

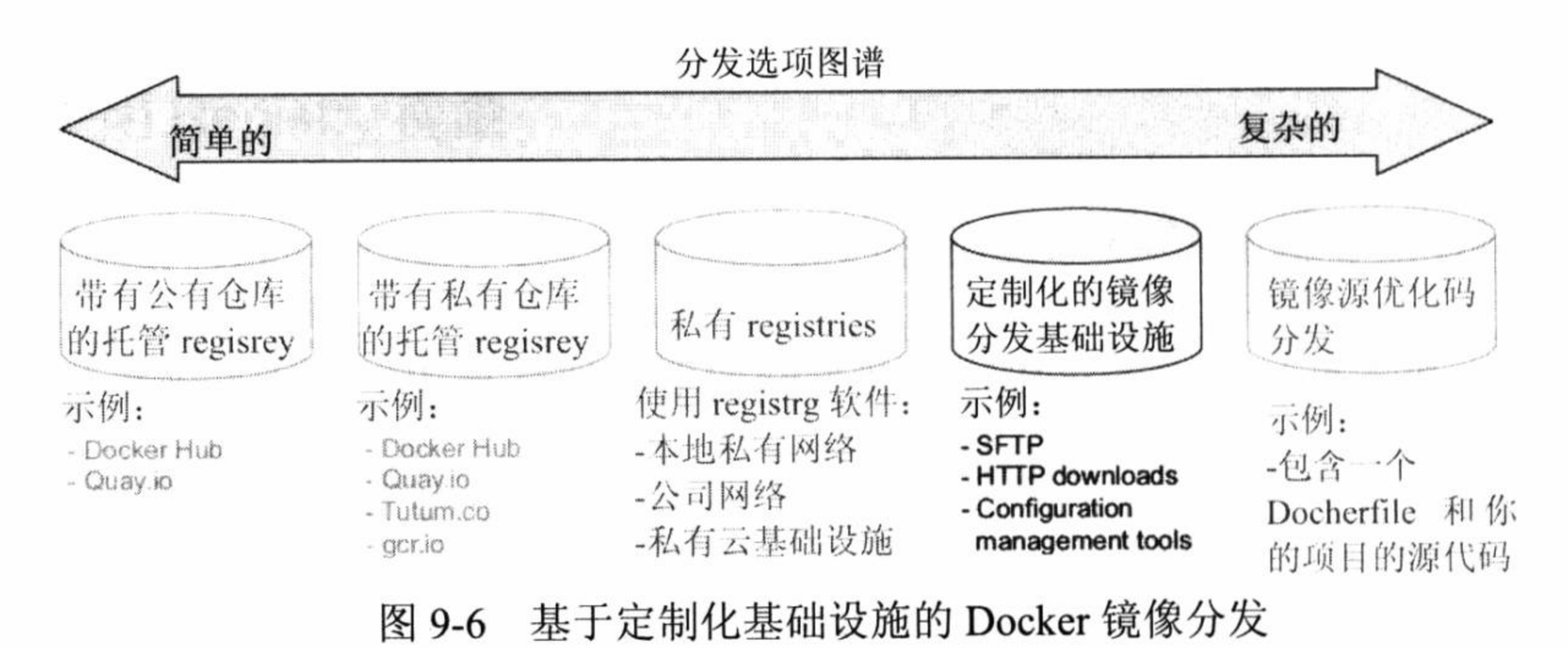


图 9-6 基于定制化基础设施的 Docker 镜像分发

我们已经讨论了所有的将镜像作为文件来处理的方法。第 3 章讨论了加载镜像到 Docker 并且保存镜像到硬盘中。第 7 章讨论了将完整的文件系统导出和导入为合并的镜像。这些技术是构建分发工作流程的基础，如图 9-7 所示。

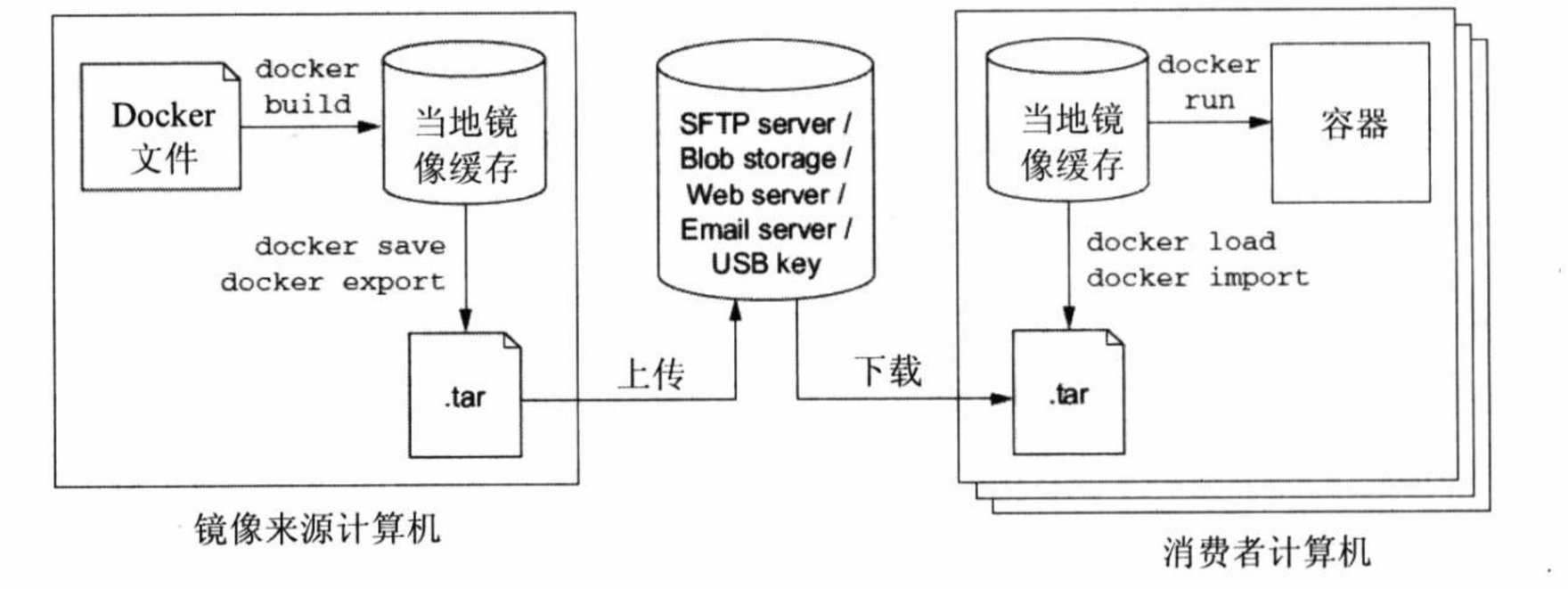


图 9-7 生产者、传输和消费者之间的一个典型的手动分发工作流程

如图 9-7 所示的工作流程是一个了解你如何使用 Docker 创建一个镜像并准备分发的概括。你应该熟悉使用 `docker build` 命令来创建一个镜像，用 `docker save` 命令或者 `docker export` 命令来创建一个镜像文件，你可以使用一个单独的命令来执行每一个操作。

一旦有了一个文件格式的镜像，你就可以使用任何传输协议。在图 9-7 中没有显示的一个定制组件是上传一个镜像的传输机制，这个机制可能是一个被文件共享工具（如 Dropbox 监视的文件夹），也可能是一段定期运行的自定义代码，或者针对一个新文件，并使用 FTP 或 HTTP 将文件推送到远程服务器。无论是什么机制，该通用组件都要求努力集成。

该图还展示了客户端如何接收镜像并在镜像分发后使用它来构建容器，类似于镜像的起源，客户端需要一些过程或机制从远程源代码获得镜像。一旦客户端有了镜像文件，他们就可以使用 `docker load` 或者 `import` 命令完成传输。

针对个别的选择标准来衡量人工镜像分发是没有意义的，使用一个非 Docker 的分发渠道会给你完整的控制，它将由你来决定你的选项是如何权衡标准的，如表 9-4 所示。

表 9-4 自定义镜像分发基础设施的性能

标　准	评级	备　注
成本	好	至少，一个私有 Registry 增加了硬件的开销（虚拟或其他），支持费用以及失败风险，但社区通过构建开源软件已经投资了大部分需要部署私有 Registry 的项目。成本的伸缩将比托管 Registry 具有更多的维度，托管 Registry 的成本随着原生仓库数量伸缩，然而私有 Registry 的成本伸缩是随着事务和存储使用率的，如果你建立了一个具有高事务率的系统，你需要按需扩大 Registry 主机的数量，同样的，Registry 服务于一些小镜像比相同数量的大镜像会有低得多的存储成本
可见性	好	私有 Registry 的可见性一样可由你来决定，但即使你拥有的 Registry 对全世界开放，仍然会比像 Docker Hub 这样的广受欢迎的 Registry 曝光度要少得多
传输速度和大小	最佳	任何客户端与任何 Registry 之间的延迟会根据网络上这些节点之间的距离、网络速度以及 Registry 上的拥堵而有不同，由于那些变量的差异，私有 Registry 可能会快于或者慢于托管 Registry。但是私有 Registry 将会吸引大多数人或者机构来用于内部基础设施。消除互联网或者数据中心网络间的依赖会成比例的改善延迟。因为这个解决方案是使用 Docker Registry 的，它将共享与托管 Registry 解决方案相同的并行性传输收益
可用性控制	最佳	你作为 Registry 所有者拥有完全的可用性控制权
生命周期控制	最佳	你作为 Registry 所有者拥有完全的生命周期控制权
访问控制	好	Registry 软件不包含任何身份认证或者开箱即用的授权功能，但是实现这些功能可以获得最小的工程操作开销
产品完整性	最佳	Registry API V2 支持内容可寻址的镜像，并且这款开源软件支持可插拔的存储后端。对于额外的完整性保护，你可以在网络上强制使用 TLS，并且使用后端静止存储
保密性	好	私有 Registry 在分发选项图谱上是第一个适合存储商业机秘密或者机密材料的解决方案，你控制身份认证和授权机制，你也控制网络以及在传输中的安全机制。最重要的是,你控制静止存储。这是在你的权力范围之内确保系统以这样一种保密的方式配置
必要的经验	好	开始和运行本地 Registry 只需要基本的 Docker 经验，但运行和维护一个高可用的生产环境 Registry 需要多个技术领域的经验，具体的设置取决于你想利用的特性。一般来说，你要熟悉搭建 Nginx 代理、LDAP 或者 Kerberos 提供的身份认证，以及 Redis 缓存

所有的标准同样适用于手动分发，但没有特定的传输方法的内容就很难展开讨论。

使用文件传输协议的分发基础设施示例

搭建一个功能齐全的示例将帮助你正确地了解探究什么是手动分发基础设施，这一节将会使用 FTP 协议搭建一个这样的基础设施。

相比以往，FTP 更加不受欢迎了。这个协议没有提供保密性，需要通过网络传输凭证进行身份认证，但是软件是随手可得的，客户端适用于大部分平台，这使得 FTP 可以成为一个很棒的搭建分发基础设施的工具，如图 9-8 所示了你需要搭建的内容。

本节中的示例使用了两个现有的镜像。*dockerinaction/ch9*ftpd 是第一个，它是一个基于 centos:6 镜像的专门化版本，其中安装了 vsftpd（一个 FTP Daemon），并且配置为可以匿名写入访问。第二个是 dockerinaction/ch9*ftp*client，是一个基于流行的 Apline Linux 简化版本的专门化版本，其中安装了一个名叫 LFTP 的 FTP 客户端，并且设置为镜像的 Entrypoint。

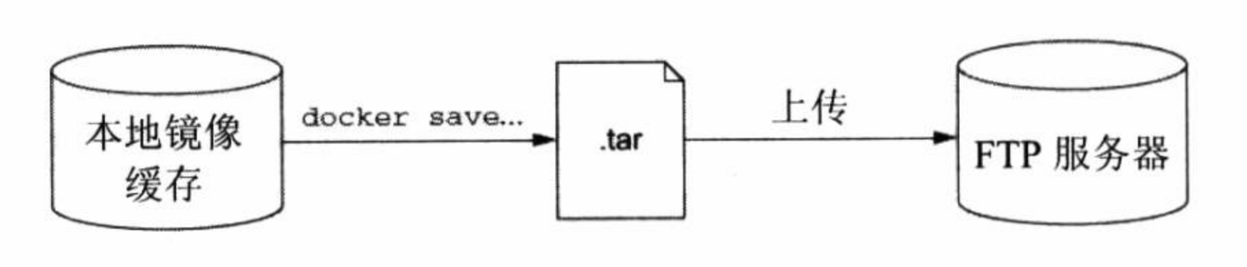


图 9-8 FTP 发布基础设施

为了准备实验，需要从 Dockder Hub 拉取一个已知的镜像用来作为分发，在这个例子中使用了 registry:2 镜像：

```
docker pull registry:2
```

一旦你有一个需要分发的镜像，就可以开始了，第一步是构建你的镜像分发基础设施，在本例中，意味着要运行一个 FTP 服务器：

```
docker run -d --name ftp-transport -p 21:12 dockerinaction/ch9_ftpd
```

这个命令将会启动一个在 TCP 端口 21（默认端口）上允许 FTP 连接的 FTP 服务，不要在生产环境中使用这个镜像。这个服务将被配置为允许匿名并在 pub/incoming 目录下写入访问，你的分发基础设施将会使用这个目录作为镜像分发接入点。

下一步你需要做的就是导出一个为文件格式的镜像，可以使用如下命令：

```
docker save -o ./registry.2.tar registry:2
```

运行这个命令将会在你的当前目录下导出这个 registry:2 镜像，其为一个结构化的镜像文件，文件将保留所有的元数据以及镜像相关的历史数据，在这个时刻，你可以注入各种阶段，比如校验和生成或者文件加密。这个基础设施没有这样的要求，你应该向着分发的目标前进。

dockerinaction/ch9*ftp*client 镜像有一个安装好的 FTP 客户端，可以用来上传你的新镜像到你的 FTP 服务器。记住，你是在一个容器中启动一个名为 ftp-transport 的 FTP 服务器。如果你在你的计算机上运行这个容器，你可以使用容器链接机制从客户端来引用一个 FTP 服务器，否则，你将会使用主机命名注入（参见第 5 章）、服务器 DNS 命名或者一个 IP 地址：

```
docker run --rm --link ftp-transport:ftp_server \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e 'cd pub/incoming; put registry.2.tar; exit' ftp_server
```

这个命令创建了一个容器，绑定一个卷到你的本地目录，并连接到你的 FTP 服务器容器，该命令将会使用 LFTP 来上传一个名为 registry.2.tar 的文件到位于 ftpserver 的服务器。你可以通过罗列 FTP 服务器目录下的内容来验证你上传的镜像：

```
docker run --rm --link ftp-transport:ftp_server \
   -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e "cd pub/incoming; ls; exit" ftp_server
```

这个 registry 镜像现在可以下载到任何可以通过网络访问服务器的 FTP 客户端，但该文件可能永远不会被当前的 FTP 服务器配置所覆盖。如果你需要在生产环境中使用类似的工具，需要提供自己的版本管理机制。

在这个场景中推广镜像可用性，需要客户端通过使用你运行的最后一个命令定期轮询服务器。你可以另外建立一些网站或者发送电子邮件告知客户端有关镜像的消息，不过所有这些都是跟标准的 FTP 传输工作流程无关的了。

在继续评估这个分发方法之前，使用 registry 镜像从 FTP 服务器获得客户端如何集成的信息。

第一步，从你的本地镜像缓存中删除 registry 镜像，并从你的本地目录删除文件：

```
rm registry.2.tar
docker rmi registry:2
```

首先需要移出任何 registry 容器

然后从你的 FTP 服务器下载镜像文件：

```
docker run --rm --link ftp-transport:ftp_server \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e 'cd pub/incoming; get registry.2.tar; exit' ftp_server
```

在这时，你应该再次在你的本地目录里拥有 registry.2.tar 文件，可以使用 docker load 命令重新加载镜像到你的本地镜像缓存。

```
docker load -i registry.2.tar
```

这是一个有关镜像手动发布和分发的基础设施如何搭建的最小的例子，借助于一些扩展，你可以搭建一个符合生产环境质量要求的基于 FTP 的分发中心，本示例中的当前配置符合这些选择标准，如表 9-5 所示。

表 9-5 基于 FTP 的分发基础设施示例的性能

标 准	评级	备 注
成本	好	分发成本是由带宽、存储和硬件需求驱动的，托管分发解决方案（如云存储）将把这些成本打包，并且当你的使用量增加时，通常会降低单位价格，但是托管解决方案是将人员成本和多个你不需要的益处一起打包了，相对于你自己运行 Registry 抬高了价格
可见性	好	就像私有 Registry，绝大多数的手动分发方法都需要比有名气的 Registry 付出更多的努力来推广，这方面的例子可能包括使用流行的网站或者其他知名的文件分发中心
传输速度和大小	好	然而传输速度取决于传输，文件大小取决于你选择使用的分层镜像或者合并镜像。记住，分层镜像维持了镜像历史、容器的创建、元数据以及可能已被删除或重写的旧文件。合并的镜像只包含文件系统上的当前文件集
可用性控制	最佳	如果可用性控制权是你的场景中的一个重要因素，你可以使用你自己的传输机制
生命周期控制	糟糕	使用专有的协议、工具或其他技术，无论是开放的，还是不在你的控制下的，都会影响生命周期控制。例如，与托管文件共享服务（如 Dropbox 搭配的分发镜像文件）就不会赋予你生命周期控制权，另一方面，与你的朋友交换 USB 驱动器将只会在你们决定使用 USB 驱动器时持续生命周期
访问控制	糟糕	你可以在传输中使用你需要的有关文件加密的访问控制功能。如果你建立了一个系统，用一个特定的密钥加密你的镜像文件，你可以确保只有一个人有正确的密钥可以来访问镜像
产品完整性	糟糕	完整性验证是一个更昂贵的可以实现广泛分发的功能，不过，至少你需要一个可信的通信信道用于推广加密文件签名
保密性	好	您可以使用廉价的加密工具实现内容保密。如果你需要元保密（交易本身是秘密的）以及内容保密，那么你应该避免使用托管工具，确保你使用的传输机制提供了保密性（HTTPS、SFTP、SSH 或离线）
必要的经验	好	托管工具通常被设计为易于使用的，并需要较小程度的集成工作流程的经验。但在大多数情况下，你可以轻松地使用你所有的简单工具

简而言之，几乎没有真正的场景显示这个传输配置是适当的，但它有助于说明当你将镜像作为文件来处理时不同的关注问题以及可以创建的基本工作流程，唯一更有弹性和潜在复杂的镜像发布和分发方法涉及的是如何分发镜像源代码。

9.5 镜像源代码分发工作流程

当分发镜像源代码而不是镜像时，你可以关闭所有的 Docker 分发工作流程，仅仅依靠

Docker 镜像构建器。镜像手动发布和分发时，源代码分发工作流程应该按照一个特定实现内容的选择标准来评估。

使用一个托管源代码控制系统（如 GitHub 上的 Git）将与使用像 `rsync` 这样的文件备份工具相比有很大的不同，在某种程度上，源代码分发的工作流程是那些镜像手动发布和分发工作流程关注问题的超集。你必须构建你自己的工作流，但是没有 `docker save`、`load`、`export` 或者 `import` 命令的帮助。生产者需要确定他们如何打包他们的源代码，消费者需要了解这些源代码是如何打包的，就像他们自己如何构建一个镜像一样。这些扩展接口使源代码分发的工作流程成为最有弹性的和潜在最复杂的分发方法。如 9-9 所示了展示了镜像源代码分发位于选项图谱中最复杂的一端。

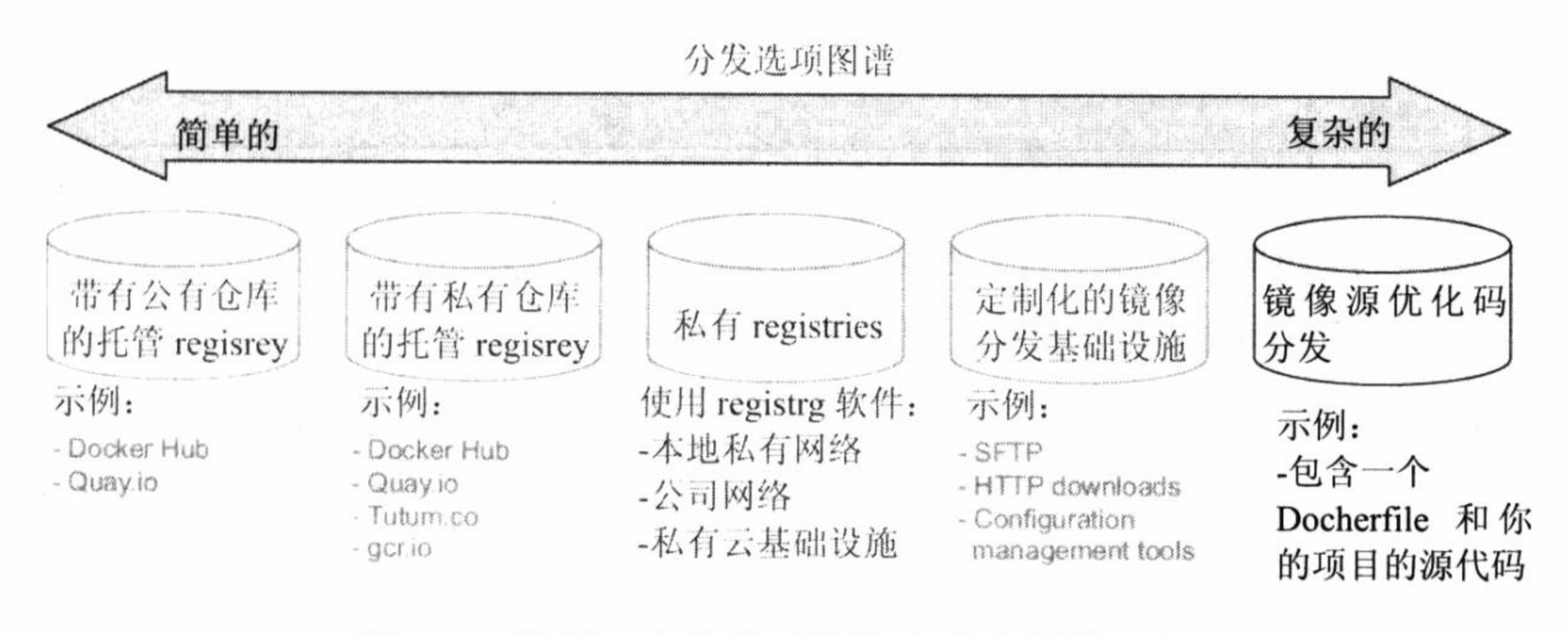


图 9-9 使用已有的基础设施来分发镜像源代码

镜像源代码分发是最常见的一种方法，尽管也是潜在的最复杂的方法，原因是扩展接口已经被流行的版本控制软件标准化了。

在 GitHub 上使用 Docker 来分发一个项目

使用 Dockerfile 和 GitHub 来分发镜像源代码几乎等同于在一个托管 Docker 镜像仓库上设置一个自动化构建。使用 Git 来整合你的本地和 GitHub 上的仓库的所有步骤是相同的，唯一的区别是你不要创建一个 Docker Hub 账号或者仓库。相反，你的镜像消费者将直接克隆你的 GitHub 仓库并在本地使用 `docker build` 命令来构建你的镜像。

假设生产者有一个现有的项目、Dockerfile 和 GitHub 仓库，它们的分发工作流程将会看起来如下所示：

```
git init
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
git add Dockerfile
# git add *whatever other files you need for the image*
git commit -m "first commit"
git remote add origin https://github.com/<your username>/<your repo>.git
git push -u origin master
```

与此同时，消费者会使用这样的一组通用命令集：

```
git clone https://github.com/<your username>/<your repo>.git
cd <your-repo>
docker build -t <your username>/<your repo> .
```

这些所有的步骤对于常规的 Git 或者 GitHub 用户而言是非常熟悉的，如表 9-6 所示：

表 9-6 基于 GitHub 的镜像源代码分发性能

标 准	评级	备 注
成本	最佳	如果你正在使用一个共有的 GitHub 仓库，那么是没有成本的
可见性	最佳	GitHub 对于开源工具是一个非常醒目的位置，它提供了优秀的社交和搜索组件，可以简单地发现项目
传输速度和大小	好	通过分发镜像源代码，你可以利用其他 Registry 的基本层，这样做可以减少运输和存储的负担。GitHub 还提供一个内容分发网络（CDN），CDN 使得世界各地的客户可以低网络延迟来访问 GitHub 项目
可用性控制	最糟糕	依靠 GitHub 或者其他的托管版本控制供应商消除了任何的可用性控制权
生命周期控制	糟糕	虽然 Git 作为一种流行的工具应该有一段时间了，通过集成 GitHub 或者其他托管版本控制供应商，你放弃了任何的生命周期控制权
访问控制	好	GitHub 或其他托管版本控制供应商针对私有仓库提供了访问控制工具
产品完整性	好	这个解决方案对于作为构建流程一部分产生的镜像或者克隆到客户端机器的源代码没有提供完整性保证，但完整性是版本控制系统的重点。任何完整性问题都应该通过标准的 Git 流程来显现出来并很容易地恢复
保密性	最糟糕	公有项目不提供源代码保密性
必要的经验	好	镜像生产者和消费者都需要熟悉 Dockerfile、Docker 构建器和 Git 工具

镜像源代码分发与所有的 Docker 分发工具是脱离的，仅仅依靠镜像构建器，你可以采用任何可用的分发工具集。如果你为分发或者源代码版本控制锁定了一个特定的工具集，这也许是唯一的符合标准的选择。

9.6 小结

本章涵盖了各种软件分发的机制以及 Docker 在每个分发机制方面的价值贡献，最近实施了分发渠道或者目前正在这样做的读者可能需要额外的洞察力来审视他们的解决方案，而其他的读者会了解更多有效的选择。在任何情况下，重要的是在继续往下阅读之前，要确保你已经获得了以下的知识：

- 有一个选择的选项图谱显示了你的选择范围。
- 你应该总是使用一组一致的选择标准，以评估你的分发选项并确定应该使用哪个方法。
- 托管的公有仓库提供了出色的项目可见性，是免费的，并且只需要非常少的经验就可以采用。
- 因为镜像是由一个受信任的第三方构建的，所以消费者将对其自动构建产生的镜像有更高程度的信任。
- 托管的私有仓库对于小型团队是划算的，提供了令人满意的访问控制。
- 运行自己的 Registry 使你能够构建适合特殊使用案例的基础设施，并且不需要放弃 Docker 的分发设施。
- 将镜像分发为文件，可以用任何文件共享系统来完成。
- 镜像源代码分发是非常弹性的，但是在你运用的时候会非常复杂，使用流行的源代码分发工具和模式会让事情变得简单。

第 10 章　运行自定义 Registry

本章介绍

- 直接使用 Registry API
- 搭建一个中央 Registry
- Registry 认证工具
- 大规模配置 Registry
- 通过通知集成

第 9 章讨论了几个分发 Docker 镜像的方法，其中一个方法涉及了运行 Docker Registry，这是一个弹性的镜像分发组件，对于这个方法自身或者作为更大的复杂系统的一部分是非常有用的。出于这个原因，了解如何配置自己的 Registry 将帮助你最有效地使用 Docker。

有些集成了 Docker Registry 做软件开发的人员，可能要运行一个本地实例来做开发，他们也可能使用它作为他们项目的预生产环境，一个开发团队可能会部署自己的中央 Registry 来分享他们的工作和简化集成，一个公司可能会运行一个或者多个拥有持久化存储的集中式 Registry。这些 Registry 可以用来控制外部镜像的依赖关系或者用来管理部署，如图 10-1 所示的这些配置。本章将会讨论所有的这些使用案例、扩展方法和对于 Registry API 本身的介绍。在本章末尾，你将会启动一个针对任何使用案例做了适当配置的 Registry。

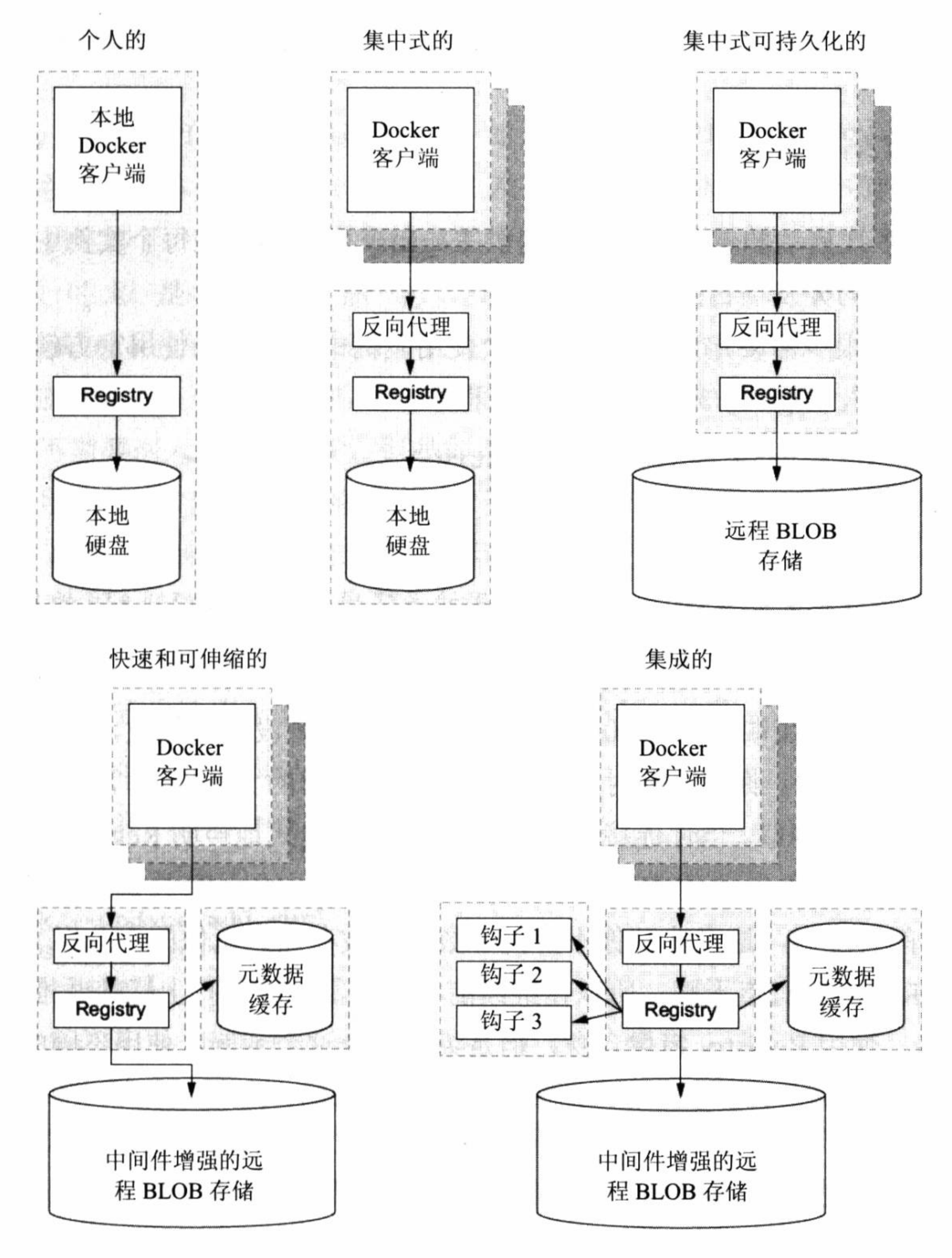


图 10-1 从个人的到可靠的、可伸缩的和可集成的 Registry 配置

任何实现 Registry API 的程序是一个注册表，本章使用分发（docker/distribution）项目，这个项目可以在 Docker Hub 上的 Registry 仓库找到。

10.1 运行个人 Registry

如果你刚开始使用或者出于开发目的使用，个人 Registry 是很好的。介绍材料在第 9

章，但这一节演示了如何使用 Registry API 来访问你的新 Registry。

10.1.1 再度介绍镜像

在本章你会多次启动和重新启动一个提供 Registry 的容器，在每个实例中你将使用由 registry:2 仓库提供的分发项目的专门化版本。

个人 Registry 很少需要定制，在这样一个使用案例中，你可以使用官方镜像，拉取镜像并且启动一个个人 Registry 来开始我们的旅程：

```
docker run -d --name personal_registry \
    -p 5000:5000 --restart=always \
    registry:2
```

小贴士 在写作本文的时候，latest 标签引用的是实现了 Registry V1 API 的 Registry 最后一个版本，除非另有注明，比如本章中的示例需要 V2 API。

这个分发项目运行在端口号 5000，但是客户端对于位置没有做假设，并且尝试连接到默认的 80 端口（HTTP）。你可以映射主机的 80 端口到容器的 5000 端口，但是本例你应该直接映射主机的 5000 端口。当你连接到 Registry，你需要显式地声明 Registry 运行的端口。

你从 Registry 镜像启动的容器将会存储你发送到此的仓库数据到一个挂载点为 /var/lib/registry 的管理卷，这意味着你不必担心数据会被存储到镜像的主分层文件系统。

一个空的 Registry 有点无聊，所以在继续前行之前需要给其打上标签并推送一个镜像到这个 Registry。使用 Registry 镜像自身，但是为了区分这个示例，使用不同的仓库名称：

```
docker tag registry:2 localhost:5000/distribution
docker push localhost:5000/distribution:2
```

推送命令将会为上传到 Registry 的镜像中的每一层都输出一行，最后输出镜像的指纹。如果你愿意，你可以删除本地标签为 localhost:5000/distribution:2 的镜像，然后试着从你的 Registry 拉取：

```
docker rmi localhost:5000/distribution:2
docker pull localhost:5000/distribution:2
```

所有这些命令在第 2 章和第 7 章都有所涉及，区别在于那些章节是有关托管 Registry 的。这个示例强调当运行你自己的 Registry 时，在托管解决方案和定制化基础设施之间你的知识、脚本和自动化基础设施是可移植的。使用像这样的命令行工具对于脚本和用户是非常棒的，但是如果你开发软件来与 Registry 作集成，你就需要直接使用 API 了。

10.1.2 介绍 V2 API

Registry V2 的 API 是 RESTful 风格的，如果你不熟悉 RESTful API，只要知道 RESTful API 是一个遵循超文本传输协议（HTTP）以及按照 HTTP 协议原语来访问和操纵远程资源来使用的也就足够了，关于这方面有很多优秀的在线资源和书籍。对于 Docker Registry，这些资源可以是标签、清单、BLOBs 和上传的 BLOB。你可以在 https://docs.docker.com/registry /spec/api/上找到完整的 Registry 规范。

因为 RESTful API 使用 HTTP，你可能需要一些时间来熟悉协议的细节。本章的示例是完整的，你不需要深入了解 HTTP，但是如果你知道幕后发生了什么，那么你会受益更多。

为了运用任何 HTTP 服务，你需要一个 HTTP 客户端，真正聪明的读者可能知道如何使用原生的 TCP 连接来处理 HTTP 请求，但大多数都不愿意处理麻烦的低层细节协议。尽管 web 浏览器能够处理 HTTP 请求，命令行工具会给你完全操作 RESTful API 的能力。

在本章的例子使用 cURL 命令行工具，因为本书是有关 Docker 的，你应该在容器内使用 cURL，这种方式也适用于 Docker 原生和 Boot2Docker 用户，为此需要准备一个镜像（借此也可以练习 Dockerfile 技能）：

```
FROM gliderlabs/alpine:latest
LABEL source=dockerinaction
LABEL category=utility
RUN apk --update add curl
ENTRYPOINT ["curl"]
CMD ["--help"]
```

来自 curl.df 文件

```
docker build -t dockerinaction/curl -f curl.df .
```

通过这个新的 dockerinaction/curl 镜像，你可以执行示例中的 cURL 命令，而不用担心 cURL 是否需要在你计算机上安装或者安装什么版本。为了庆祝你的新镜像的诞生，可以通过给你的正在运行的 Registry 发送一个简单的请求来开始使用 Registry API：

```
docker run --rm --net host dockerinaction/curl -Is
    http://localhost:5000/v2/
```

注意/v2/

该请求将导致以下输出：

```
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json; charset=utf-8
Docker-Distribution-Api-Version: registry/2.0
```

这个命令用来验证 Registry 运行的是 V2 API，并在 HTTP 响应头部中返回特定的 API 版本，请求的最后组成部分/v2/，是每一个基于 V2 API 的资源的前缀。

如果你不小心向一个运行 V1 API 的 Registry 发出该请求，输出会看起来像这样：

```
HTTP/1.1 404 NOT FOUND
Server: gunicorn/19.1.1
Connection: keep-alive
Content-Type: text/html
Content-Length: 233
```

HTTP 详细解释 这个命令使用了一个 HTTP DEAD 请求只获得响应头部。针对该资源的一个成功的 GET 请求将会返回一个相同的头部和一个携带空的 DOM 的响应 body。

现在你已经使用 cURL 命令来验证 Registry 确实使用 V2 API 了，那么应该做更多有趣的东西。下一个命令将在你的 Registry 里的分发仓库里获得标签列表：

```
docker run --rm -u 1000:1000 --net host \
    dockerinaction/curl -s http://localhost:5000/v2/distribution/tags/list
```

运行这个命令应该显示结果如下：

```
{"name":"distribution","tags":["2"]}
```

在这里你可以看到 Registry 以 JSON 文档格式响应你的请求，该 JSON 文档列出了你的分发仓库中的标签，在本例中只有一个。JSON 文档是一个键-值对的结构化文档，它使用括号来表示对象（其中包含另一组键值对），方括号代表列表，引号代表字符串值。

再次运行这个示例，但是这次添加了另外一个标签到你的仓库，产生一个更有趣的结果：

```
docker tag \
    localhost:5000/distribution:2 \
    localhost:5000/distribution:two

docker push localhost:5000/distribution:two

docker run --rm \
    -u 1000:1000 \
    --net host \
    dockerinaction/curl \
    -s http://localhost:5000/v2/distribution/tags/list
```

创建 tag 名称

以非特权模式运行

以无 network 命令空间方式运行

curl 命令返回的输出如下所示：

```
{"name":"distribution","tags":["2","two"]}
```

在输出中可以清楚地看到仓库中包含了你定义的标签，推送到你的 Registry 里的仓库的每一个不同的标签都会被罗列出来。

在本节中你可以使用创建的个人 Registry 用于任何测试或者你可能需要的个人生产力

需求，但在使用方面有些限制，如果没有任何配置更改的话。即使你没有计划部署一个集中式 Registry，你也可以从一些采用在 10.5 节讨论的 Registry 通知功能的定制方案中受益匪浅。不过在你作出这些改变之前，你需要知道如何定制 Registry 镜像。

10.1.3 定制镜像

本章的其余部分将解释如何立足于 Registry 镜像来扩展个人 Registry，并且深入探讨更高级的使用案例，不过在此之前你需要更多地了解 Registry 镜像本身。

本章将用 Dockerfiles 来专门定制 Registry 镜像，如果你不熟悉 Dockerfile 语法并且没有读过第 8 章，那么你现在应该就去阅读该章或者在 https://docs.docker.com/reference/builder 阅读在线文档。

你需要知道的有关镜像的第一件事是关键组件：

- `registry` 的基础镜像是基于 Debian 的，已经更新了依赖关系。
- 主程序被命名为 `registry`，并在 PATH 路径上可用。
- 默认的配置文件为 config.yml。

不同的项目维护人员都有不同的对于最佳基础镜像的想法，对于 Docker，Debian 是一个惯常的选项。`Debian` 有一个对于分发功能齐全的最小封装，只需要占用大约 125MB 硬盘空间，它还附带一个广受欢迎的包管理器，所以安装或升级依赖关系已经不是一个问题了。

主程序命名为 registry，并且设置为镜像的 Entrypoint，这意味着从镜像启动一个容器时，你可以省略任何命令行参数获得默认行为，或者直接添加自己的参数到 `docker run` 命令的后面部分。

像所有的 Docker 项目一样，这个分发项目旨在提供一个合理的缺省配置，这个缺省配置位于 config.yml 文件中。顾名思义，这个配置是用 YAML 写的，如果你不熟悉 YAML，也不需要担心。YAML 是被设计为最大化的可读性的。如果你有兴趣，你可以在 http://yaml.org 上发现很多 YAML 的资源。

这个配置文件是本章的真正核心，有几种在镜像中注入你自己的配置的方法。你可以直接修改包含的文件，一个绑定的挂载卷可以用来覆盖一个你已经写入的文件。在这种情况下，你会将你自己的文件复制到另一个位置，并为你的新镜像设置一个新的默认命令。

这个配置文件包含 9 个顶级字段，每个字段定义了 Registry 的主要功能组件：

- `version`——这是一个必需的字段，指定了配置版本（不是软件版本）。
- `log`——本节中的这个配置控制由分发项目产生的日志输出。

- storage——存储配置控制在何处，以及如何进行镜像存储和维护。
- auth——这个配置控制 Registry 中的身份认证机制。
- middleware——中间件配置是可选的，它用于配置存储、注册表或者使用中的仓库中间件。
- reporting——某些报告工具已经整合到了分发项目里，这些工具包括 Bugsnag 和 NewRelic，这个字段配置这些工具集。
- http——这一字段指定分发系统应该如何在网络上可用。
- notification——在 notification 字段中的配置为以 Webhook 风格与其他项目集成。
- redis——最后在 redis 字段中提供 Redis 缓存的配置。

记住这些组件，你应该通过定制 Registry 镜像来准备好构建高级 Registry 的使用案例。记住，分发项目是快速发展的，如果你在本书的指令方面遇到棘手的问题，你可以咨询在线文档以供参考或者在 https://github.com/docker/distribution 检出项目本身。

10.2 集中式 Registry 的增强

启动 Registry 镜像的一个本地副本，可以不用做修改就可以很好地运行于个人用途或者测试。当不止一个人需要访问相同的 Registry 时，就被称为一个集中式 Registry。这一节解释了如何通过定制官方 Registry 镜像来实现一个集中式的 Registry，如图 10-2 所示了从个人 Registry 发展到集中式 Registry 所涉及的变化。

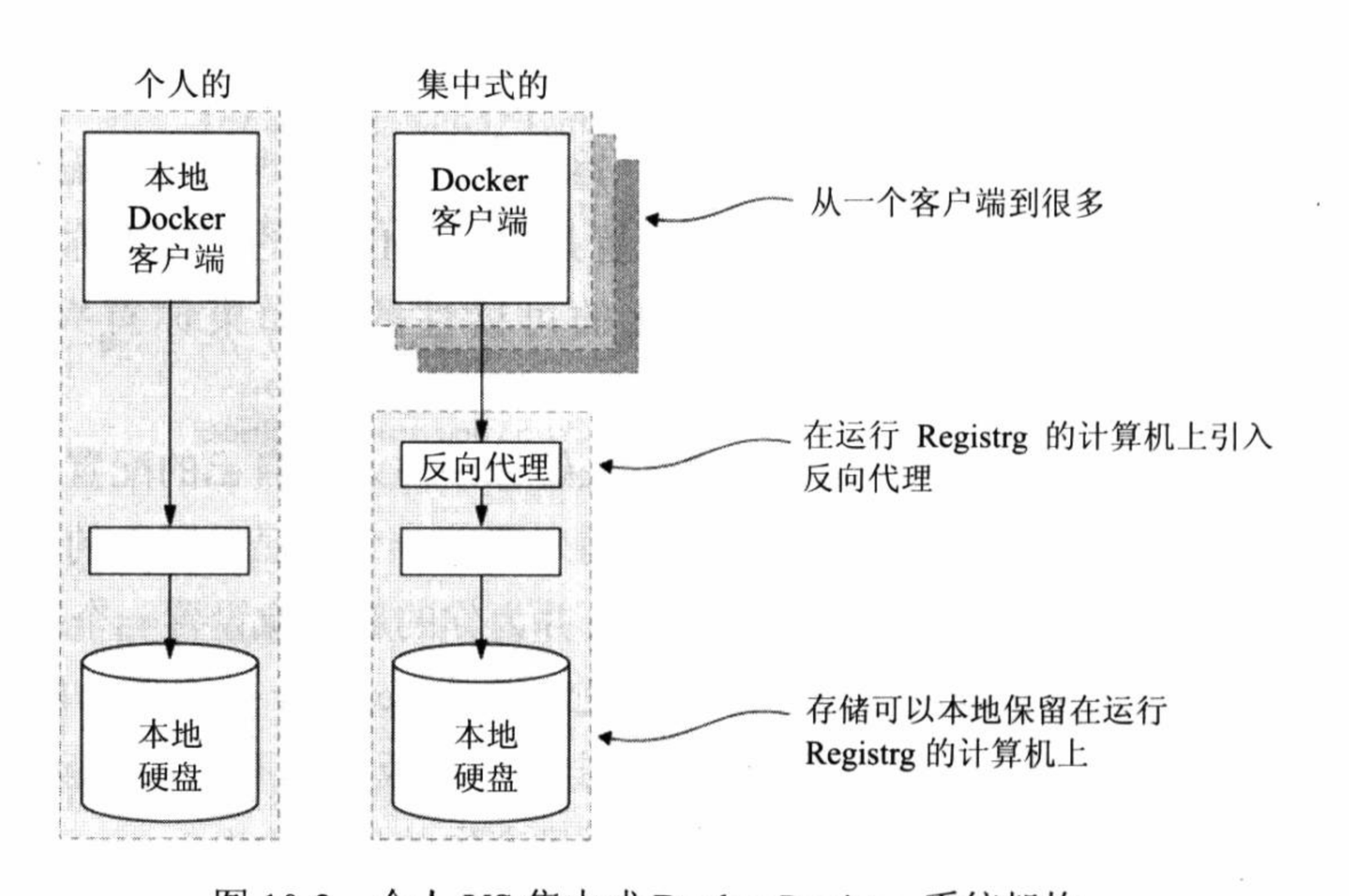


图 10-2 个人 VS 集中式 Docker Registry 系统架构

对于不止一个人来访问的 Registry，需要保证在网络上可用。你可以很容易地达成此项目标，只需要映射 Registry 容器端口到正在运行（`docker run ...-p 80:5000 ...`）的计算机网络接口的 80 端口上。引入网络依赖带来了一整套新的安全漏洞。从窥探到破坏镜像传输，中间人攻击可以引发几个问题。添加一个传输层安全（TLS）将会保护你的系统免于这些或者那些攻击。

一旦客户端可以访问 Registry，你要确保只有正确的用户可以访问它，这就是所谓的身份认证（在 10.2.3 节讨论）。

任何服务所有者遇到的最不可避免的问题是客户端兼容性，鉴于有多个客户端会连接到你的 Registry，你需要考虑他们使用的 Docker 是什么版本，支持多个客户端版本可能会非常棘手，在 10.2.4 节展示了处理这种问题的一种方法。

在开始下一个 Registry 类型之前，在 10.2.5 节讨论了在生产环境中 Registry 配置的最佳实践，包括强化和预防性维护步骤。

这些问题大多集中在与客户端的互动中，虽然分发软件有一些工具可以满足你的需求，在系统中添加一个反向代理可以引入所需的弹性，从而实现所有这些增强功能。

10.2.1 创建一个反向代理

当你使用正确的基础镜像和 Dockerfile 做定制化时，创建反向代理是一个快速的任务，如图 10-3 所示的反向代理和你的 Registry 之间的关系。

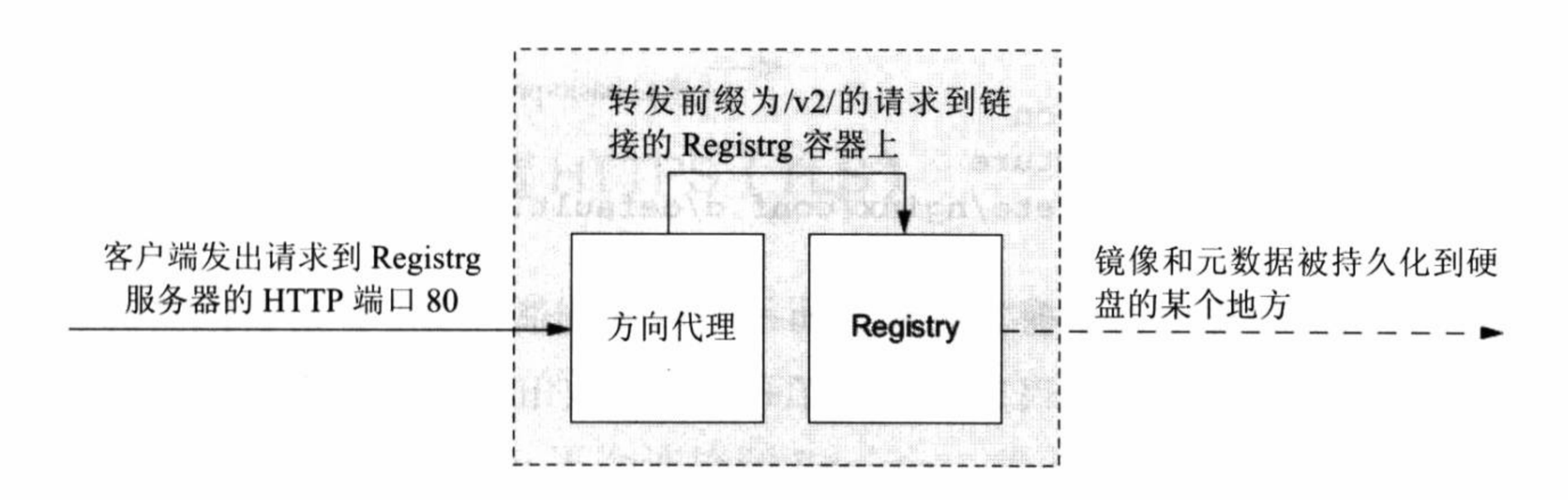


图 10-3 在客户端和 Registry 之间插入一个反向代理

你的反向代理配置将包括两个容器，第一个运行 Nginx 反向代理，第二个运行你的 Registry，反向代理容器将会通过别名 `registry` 链接到主机上的 Registry 容器。创建一个名为 basic-proxy.conf 的新文件，包含如下的配置：

```
upstream docker-registry {
  server registry:5000;
}

server {
  listen 80;
  # Use the localhost name for testing purposes
  server_name localhost;
  # A real deployment would use the real hostname where it is deployed
  # server_name mytotallyawesomeregistry.com;

  client_max_body_size 0;
  chunked_transfer_encoding on;

  # We're going to forward all traffic bound for the registry
  location /v2/ {
    proxy_pass                          http://docker-registry;
    proxy_set_header  Host              $http_host;
    proxy_set_header  X-Real-IP         $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout                  900;
  }
}
```

来自 basic-proxy.conf 文件

链接别名需求

容器端口需求

注意/v2/前缀

Upstream 解析

正如前文所述，Nginx 是一个复杂的软件，整本书一直致力于它的使用，所以我不会在这里试图解释它的种种复杂之处，你需要理解的一些配置已经标注了。这个配置将转发所有在 HTTP 主机 localhost80 的端口上的流量，根据前缀/v2 转发到 http://registry:5000。这个配置将成为接下来反向代理修改的基础。

一旦你有了反向代理配置，你需要构建一个新的镜像。最小的 Dockerfile 应该足够了，应该从最新的 Nginx 镜像启动，其中包含了你的新配置。Nginx 基础镜像负责所有的标准事项，比如公开端口。创建一个新的名为 basic-proxy.df 的文件，粘贴下面 Dockerfile 内容：

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ./basic-proxy.conf /etc/nginx/conf.d/default.conf
```

来自 basic-proxy.df 文件

小贴士 在生产环境中，你确实应该添加一个环境验证脚本（参见 8.4.1 节）来确保容器已经正确连接。但是因为你使用了一个 Nginx upstream 指令，它将可以为你验证主机是否可用。

此时，你应该准备好构建镜像，这样使用下面的 `docker build` 命令：

```
docker build -t dockerinaction/basic_proxy -f basic-proxy.df .
```

现在一切准备就绪了，你的个人 Registry 应该已经按照 10.1.1 节的示例运行了，请再次使用它，因为已经准备好了内容，接下来的命令将会创建你的新的反向代理并且测试连接：

```
docker run -d --name basic_proxy -p 80:80 \
    --link personal_registry:registry \
    dockerinaction/basic_proxy

docker run --rm -u 1000:1000 --net host \
    dockerinaction/curl \
    -s http://localhost:80/v2/distribution/tags/list
```

启动反向代理

链接到 Registry

通过反向代理运行 cURL 命令查询你的 Registry

一些事情正在迅速发生，所以在继续前进之前，请花些时间来确保你理解正在发生的那些事。

前面你创建了一个个人 Registry，它运行在一个名为 `personalregistry` 的容器里，并且公开了端口号 5000，你也添加了一些打了标签的镜像到名为 distribution 的由你个人 Registry 托管的仓库里。这组中的第一个命令创建了一个新的运行反向代理监听 80 端口的容器，这个反向代理容器链接到你的 Registry 容器。反向代理在 80 端口上收到的同时请求路径前缀为/v2/的任何流量都将被转发到你的 Registry 容器的 5000 端口上。

最后，你可以从同一台运行 Docker Daemon 的主机运行一个 `curl` 命令，该命令发出一个请求到 localhost（这种情况下你需要主机名）的 80 端口，这个请求被反向代理到你的 Registry（因为路径以/v2/开始），然后这个 Registry 的响应是分发仓库包含的标签列表。

注意 这个端到端测试非常类似于如果将你的反向代理和 Registry 容器部署到一个不同的已知名称的主机时会发生的状况。basic-proxy.conf 配置文件的注释解释了如何为生产环境的部署设置主机名称。

你正在搭建的反向代理不会在系统中添加任何东西，除了网络中的另外一个跳跃点以及强化的 HTTP 接口，接下来的三节将会解释如何修改基础配置来添加 TLS、认证和多版本客户端支持。

10.2.2 在反向代理上配置 HTTPS（TLS）

安全传输层协议（TLS）提供了端点标识、消息完整性和消息隐私，其实在 HTTP 的下一层，提供了 HTTPS 协议中的 S。使用 TLS 来给你的 Registry 加密不仅仅是一个最佳实践。没有 TLS，Docker Daemon 不会连接到 Registry，除非 Registry 运行在本地主机，这使得任何人运行集中式 Registry 都必须强制操作这些步骤。

SSH 隧道是什么？

有过 TLS 经验的读者或许已经知道提供公钥基础设施的能力同时也带来了昂贵的费用和复杂性，加密 Registry 网络流量的一个便宜又可以说不太复杂的方式是使连接只通

过 secure Shell（SSH）。

SSH 使用类似于 TLS 的安全技术，但是缺乏第三方的信任机制，这使得 TLS 可以扩展到大量的用户，不过 SSH 可以对网络流量提供隧道协议。

为了确保你的 Registry 使用 SSH，需要在作为 Registry 的同一台机器上安装 SSH 服务器（OpenSSH）。SSH 服务器到位之后，只要映射 Registry 到机器的环回接口（localhost）上，这样就可以限制 Registry 的入站流量流经 SSH 服务器。

当客户端想要在这个配置中使用你的 Registry 时，他们将创建一个 SSH 隧道。这个隧道绑定了一个本地的 TCP 端口并通过你的计算机和远程 SSH 服务器之间的 SSH 连接转发流量到此隧道中，隧道出口是一些目的主机及端口。在此背景下，客户端创建一个隧道，允许它们对待你的 Registry 就好像是在本地运行一样。客户端将使用这样的命令行创建隧道：

```
ssh -f -i my_key user@ssh-host -L 4000:localhost:5000 -N
```

隧道创建后，客户端将会使用 Registry，就好像其在本地 4000 端口运行一样。

如果你为一个小型团队运行一个集中式 Registry，那么使用 SSH 隧道可能会很有用。不过使用的前提是有用户账户管理和认证（因此你可以同时解决身份认证问题）。但这种做法并不好扩展，因为有账户管理的开销以及通常需要比 HTTPS 更高程度的用户复杂度。

HTTPS 端点在三个方面不同于 HTTP 端点：首先，监听端口是 TCP 443 端口；其次，需要签署证书和私钥文件；最后，服务器主机名和代理配置必须和创建证书的机器匹配。在本示例中，你将为本地主机创建一个自签名证书。这样的证书不适合真正的 Registry，但是有很多可用的指南可以帮助你更换为由证书颁发机构颁发的证书。如图 10-4 所示的涉及代理和 Registry 的新的 HTTPS 保护机制。

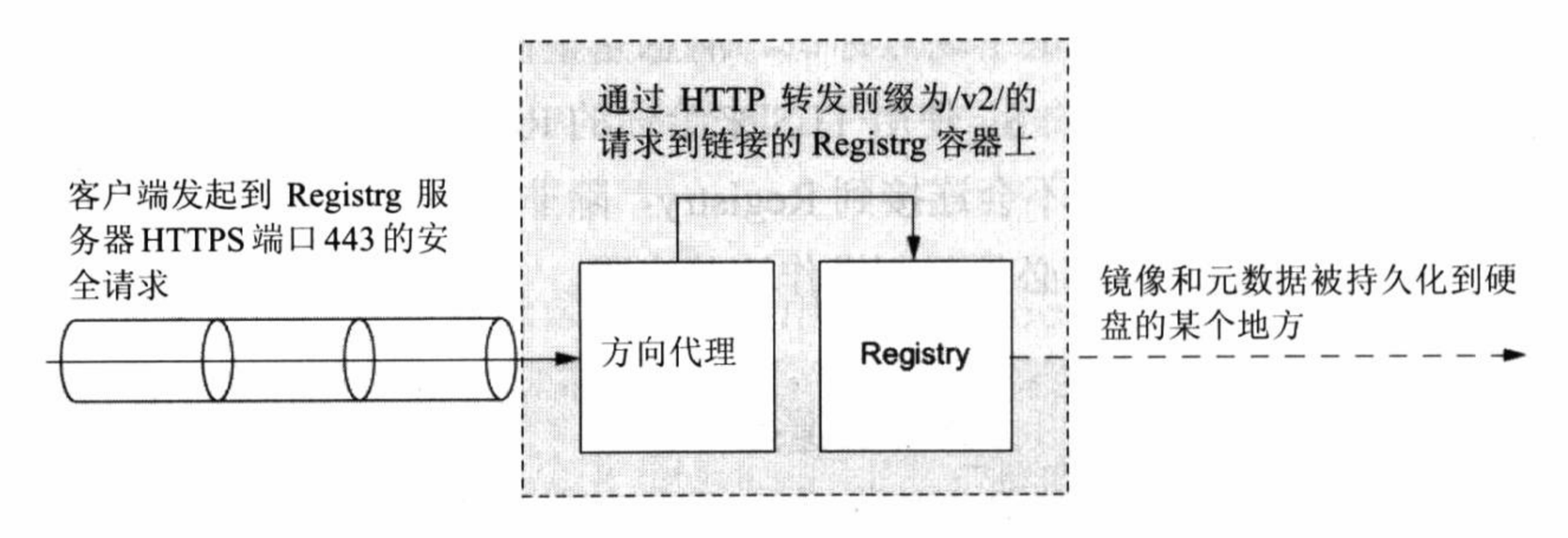


图 10-4 在代理添加一个 HTTP（TLS）端点

这个设计的第一步是生成私钥和公钥对以及自签名证书。没有 Docker 的话，你需要安装 OpenSSL 并运行三个复杂的命令。有了 Docker，以及一个由 CenturyLink 创建的公有镜像，你可以用一个命令做整件事情：

```
docker run --rm -e COMMON_NAME=localhost -e KEY_NAME=localhost \
    -v "$(pwd)":/certs centurylink/openssl
```

此命令将生成一个 4096 比特位的 RSA 密钥对，并且在你当前的工作目录中存储私钥文件和自签名证书。镜像在 Docker Hub 是公开可用的，并且由自动化构建维护。它是完全可审核的，所以必要时越是偏执，就越是可以免费验证（或重建）镜像的。在创建的三个文件中，你将使用两个，第三个是证书签名请求（CSR），可以被删掉。

下一步是创建你的反向代理配置文件，创建一个新的名为 tls-proxy.conf 的文件，拷贝如下的配置，再一次说明，相关行是注释：

```
upstream docker-registry {                          ← 来自 tls-proxy.conf 文件
  server registry:5000;
}

server {
  listen 443 ssl;                                   ← 注意端口 443 和 SSL 的使用
  server_name localhost                             ← 命名为 localhost

  client_max_body_size 0;
  chunked_transfer_encoding on;

  ssl_certificate /etc/nginx/conf.d/localhost.crt;  ← 注意 SSL 配置
  ssl_certificate_key /etc/nginx/conf.d/localhost.key;

  location /v2/ {
    proxy_pass                          http://docker-registry;
    proxy_set_header  Host              $http_host;
    proxy_set_header  X-Real-IP         $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout                  900;
  }
}
```

这个配置和基础反向代理的配置的差异包括以下几点：

- 监听 443 端口
- 注册一个 SSL 证书
- 注册一个 SSL 证书密钥

你应该注意到这个代理配置被设置为在端口 5000 上使用相同的 Registry，在相同的 Registry 上运行多个代理跟从 Registry 的角度看运行多个客户端是没有区别的。

在你把这些汇总之前，最后一步是创建一个 Dockerfile，这一次，除了复制代理配置外，

你还需要将证书和密钥文件复制到镜像里。以下的 Dockerfile 使用复制多个源的形式的指令，这样做不需要多层文件系统，否则可能会因为多个复制指令创建多个镜像层。创建一个名为 tls-proxy.df 的新文件，并插入以下行：

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ["./tls-proxy.conf", \
      "./localhost.crt", \
      "./localhost.key", \
      "/etc/nginx/conf.d/"]
```

复制新证书

复制私钥

使用以下的 docker build 命令构建新镜像：

```
docker build -t dockerinaction/tls_proxy -f tls-proxy.df .
```

现在进行汇总，使用 curl 命令启动你的反向代理并测试：

```
docker run -d --name tls-proxy -p 443:443  \
    --link personal_registry:registry \
    dockerinaction/tls_proxy

docker run --rm \
    --net host \
    dockerinaction/curl -ks \
    https://localhost:443/v2/distribution/tags/list
```

注意端口 443

注意 link

注意“k”flag

注意“https”和“443”

这个命令应该列出在你的个人 Registry 里分发仓库的标签：

```
{"name":"distribution","tags":["2","two"]}
```

本示例中的 curl 命令使用了-k 选项，该选项将指示 curl 忽略请求端点的任何证书错误，在使用一个自签名证书的场景下需要使用这个选项，除了这个细微差别外，你可以通过 HTTPS 成功地发出对 Registry 的请求。

10.2.3 添加身份认证层

有三种机制进行身份认证，包括分发项目本身，它们可以被恰当地命名为 silly、token 和 htpasswd。作为直接在分发项目配置身份认证的替代方案，你可以在反向代理层（参见 10.2.2 节）配置各种不同的身份认证机制。

第一，silly 是完全不安全的，应该被忽略掉，它仅仅适用于开发目的，在稍后的软件版本中可能会被移除掉。

第二，token 使用 JSON web token（JWT），这是与 Docker Hub 相同的认证机制。这是一个复杂的身份认证方法，使得 Registry 可以验证一个调用者是否已经用第三方服务来进行身份认证了，并且不需要任何后端通信。从中剥离的关键细节是用户不直接与你的 Registry 进行认证，使用此机制要求你部署一个单独的身份认证服务。

有几个开源 JWT 认证服务是可用的，但没有一个是可以推荐用于生产环境的。在 JWT 生态系统成熟之前，最好的做法是使用第三个身份认证机制 htpasswd。

htpasswd 是以一个 Apache web 服务器附带的工具命名的开源项目，htpasswd 用于生成编码后的用户名和密码对，其中密码已经用 bcrypt 算法进行了加密。采用 htpasswd 身份认证方式时，你应该意识到从客户端发送到你的 Registry 的密码是未加密的，这叫做 HTTP 基本身份认证，因为 HTTP 基本认证是通过网络发送密码的，这是很重要的一个方面，你可以用这个身份认证方式配合 HTTPS。

有两种方法可以添加 htpasswd 认证到你的 Registry，分别是在反向代理层和 Registry 本身，这两种情况您都需要使用 htpasswd 创建一个密码文件。如果你没有安装 htpasswd，可以使用 Docker 来这么做。按照以下的 Dockerfile（命名为 htpasswd.df）和构建命令创建一个镜像：

```
FROM debian:jessie
LABEL source=dockerinaction
LABEL category=utility
RUN apt-get update && \
    apt-get install -y apache2-utils
ENTRYPOINT ["htpasswd"]
```

一旦你有了 Dockerfile，就可以构建你的镜像：

```
docker build -t htpasswd -f htpasswd.df .
```

有了你的新镜像，你就可以为密码文件创建一个新条目，如下所示：

```
docker run -it --rm htpasswd -nB <USERNAME>
```

重要的是将<USERNAME>换成你想要创建的用户名，并为 htpasswd 使用 flag-nB。这样会使用 bcrypt 算法并在终端显示输出，程序会提示你输入密码两次，然后生成密码文件条目，将结果复制到一个名为 registry.password 的文件，结果应该是这样的：

```
registryuser:$2y$05$mfQjXkprC94Tjk4IQz4vOOK6q5VxUhsxC6zajd35ys1O2J2x1aLbK
```

一旦你有了一个密码文件，你可以在 Nginx 中通过添加 10.2.2 节展现的两行配置实现 HTTP 基本身份认证，创建 tls-authproxy.conf 文件并添加这些行：

```
# filename: tls-auth-proxy.conf
upstream docker-registry {
  server registry:5000;
}

server {
  listen 443 ssl;
  server_name localhost

  client_max_body_size 0;
  chunked_transfer_encoding on;

  # SSL
  ssl_certificate /etc/nginx/conf.d/localhost.crt;
  ssl_certificate_key /etc/nginx/conf.d/localhost.key;

  location /v2/ {
    auth_basic "registry.localhost";                          ← 认证领域
    auth_basic_user_file /etc/nginx/conf.d/registry.password;
    ↑ 密码文件

    proxy_pass                          http://docker-registry;
    proxy_set_header  Host              $http_host;
    proxy_set_header  X-Real-IP         $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout                  900;
  }
}
```

现在创建一个新的名为 tls-auth-proxy.df 的 Dockerfile：

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY [./tls-auth-proxy.conf", \
      "./localhost.crt", \
      "./localhost.key", \
      "./registry.password", \
      "/etc/nginx/conf.d/"]
```

通过这些改变，你可以使用在 10.2.2 节中的其余指令来重新构建你的 Registry 来处理 HTTP 基本身份认证，而不用重复以往的工作，通过配置分发规则来使用 TLS 和 HTTP 基本身份认证这种做法是更值得的。如果你想增强个人 Registry 的安全性，直接添加 TLS 和基本身份认证到 Registry 是有用的，在生产环境中，更加适合的是在反向代理层中终止 TLS 连接。

以下的配置文件（名为 tls-auth-registry.yml）添加了 TLS 和 HTTP 基本身份认证到另外一个默认的分发容器中：

```
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    filesystem:
        rootdirectory: /var/lib/registry
    cache:
        layerinfo: inmemory
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    tls:
        certificate: /localhost.crt
        key: /localhost.key
    debug:
        addr: localhost:5001
auth:
    htpasswd:
        realm: registry.localhost
        path: /registry.password
```

TLS 配置

认证配置

带注释的文本显示了配置中两个被修改的分段，首先是 http 分段，添加了一个名为 tls 的子分段，这个子分段有两个属性，分别是 certificate 和 key，它们的值分别是按照 10.2.2 节生成的证书和密钥文件的路径。你需要将这些文件复制到镜像里或者在运行时用卷挂载到容器里，不过将密钥文件复制到镜像中通常总是一个坏主意，除非是为了测试目的。

第二个新的分段是 auth，正如你所见，在其中新的 htpasswd 分段使用了两个属性。首先是 realm，简单定义了 HTTP 认证领域，这只是一个字符串。其次是 path，是你用 htpasswd 创建的 registry.password 文件的位置路径。可以用一个便捷的 Dockerfile（命名为 tle-auth-registry.df）来打包处理：

```
# Filename: tls-auth-registry.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
# Set the default argument to specify the config file to use

# Setting it early will enable layer caching if the
# tls-auth-registry.yml changes.
CMD ["/tls-auth-registry.yml"]
COPY [".//tls-auth-registry.yml", \
      "./localhost.crt", \
      "./localhost.key", \
      "./registry.password", \
      "/"]
```

再一次强调，在生产环境中将你的密钥文件复制到镜像里是一个坏主意，应该使用卷来代替。前面的 Dockerfile 出于演示目的将所有的专门化材料复制到了根目录。使用 docker build 和 docker run 命令来构建和启动新的安全的 Registry：

```
docker build -t dockerinaction/secure_registry -f tls-auth-registry.df  .

docker run -d --name secure_registry \
    -p 5443:5000 --restart=always \
    dockerinaction/secure_registry
```

如果你用 TLS 给 Registry 自身做了安全处理，那么当你安装一个反向代理可能会遇到问题，原因是应用级反向代理软件（比如 Nginx 或者 Apache httpd）是在 HTTP 层面运行的，它需要检查请求流量以便知道需要如何被路由或者从一个指定的客户端路由到一台相同的上游主机，只有在 TLS 会话被 Registry 终止的情况下，这样的代理才会看到加密的流量。一个可以起作用的解决方案要么是在反向代理层（正如你前面所做的）终止 TLS 会话，或者在较低的网络层（比如第四层）使用代理（负载均衡）。想了解网络分层的更多信息，请查看 OSI 模型。

如果你需要支持多个客户端版本，那么反向代理为了正确路由需要检查请求内容。Registry API 随着 Docker 1.6 的发布做了些改变，如果你想都支持这两个 Registry API，你需要实现 API 感知的反向代理层。

10.2.4 客户端兼容性

Registry 协议在版本 V1 和 V2 之间发生了翻天覆地的变化，Docker 1.6 版本之前的客户端不能与 Registry V2 对话。区分 V1 和 V2 兼容的客户端以及随后的定向请求到兼容的 Registry 服务对于我们的反向代理是非常简单的。

为了清晰所见，本节中的示例省略了任何 HTTPS 或身份认证过程，但是我们鼓励你结合相关特性来搭建一个反向代理来满足你的需求。你将搭建的支持多个客户端版本的反向代理和路由配置，如图 10-5 所示。

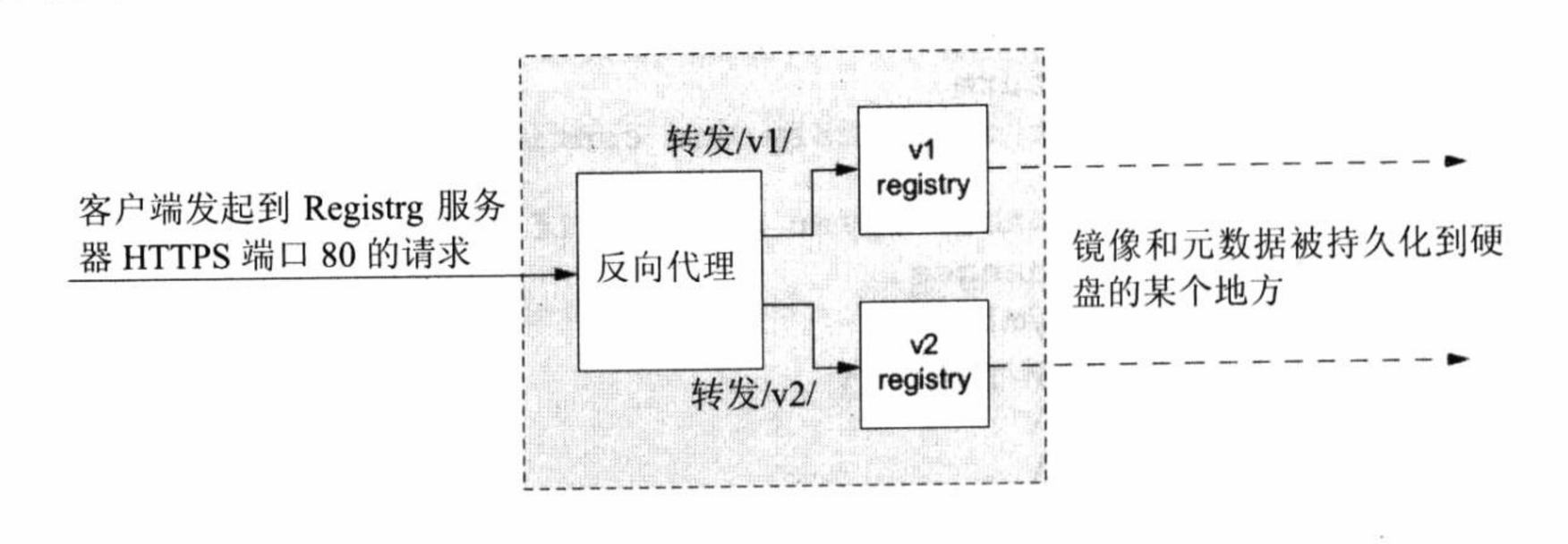


图 10-5 基于请求的 Registry API 版本进行客户端路由

正如之前你搭建的反向代理，这个修改需要三个步骤：

- 创建一个 Nginx 配置文件（dual-client-proxy.conf）
- 创建一个简洁的 Dockerfile（dual-client-proxy.df）
- 构建一个新的镜像

在名为 dual-client-proxy.conf 的文件中找到新的反向代理配置，包含如下所示：

```
upstream docker-registry-v2 {
  server registry2:5000;
}
upstream docker-registry-v1 {
  server registry1:5000;
}

server {
  listen 80;
  server_name localhost;

  client_max_body_size 0;
  chunked_transfer_encoding on;

  location /v1/ {
    proxy_pass                      http://docker-registry-v1;
    proxy_set_header  Host          $http_host;
    proxy_set_header  X-Real-IP     $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout              900;
  }

  location /v2/ {
    proxy_pass                      http://docker-registry-v2;
    proxy_set_header  Host          $http_host;
    proxy_set_header  X-Real-IP     $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout              900;
  }
}
```

V2 registry upstream

V1 registry upstream

V1 URL 前缀

V2 upstream 路由

V1 URL 前缀

V2 upstream 路由

与基本的反向代理配置相比，唯一的显著区别是包含了第二个 upstream 服务和第二个 location 规范，指定了 URL 以/v1/开始。接下来，创建一个新的名为 dual-client-proxy.df 的 Dockerfile，包括以下指令：

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ./dual-client-proxy.conf /etc/nginx/conf.d/default.conf
```

最后，创建新镜像：

```
docker build -t dual_client_proxy -f dual-client-proxy.df .
```

在你启动一个为 V1 和 V2 Registry API 的请求转发流量的反向代理之前，你需要有一个运行的 V1 Registry。以下命令将拉取 0.9.1 版本的 Registry，并在容器中启动：

```
docker run -d --name registry_v1 registry:0.9.1
```

有了这两个运行的 Registry 版本，你终于可以创建你的支持双重 API 的反向代理了。以下命令将创建反向代理并链接这两个 Registry，然后测试 V1 和 V2 版本的 API：

```
docker run -d --name dual_client_proxy \
    -p 80:80 \
    --link personal_registry:registry2 \
    --link registry_v1:registry1 \
    dual_client_proxy

docker run --rm -u 1000:1000 \
    --net host \
    dockerinaction/curl -s http://localhost:80/v1/_ping

docker run --rm -u 1000:1000 \
    --net host \
    dockerinaction/curl -Is http://localhost:80/v2/
```

从主机测试 V1

从主机测试 V2

随着时间的推移，越来越少的客户端需要 V1 版本的 Registry 支持了，但是未来总会有可能出现另外一个 API 变化的，反向代理是处理这种场景的最佳方式。

10.2.5 应用于生产环境之前

生产环境的配置在一些特定方面通常应该不同于开发或者测试环境的配置，最突出的差异是有关机密材料的管理，其次是日志优化、调试端点以及可靠的存储。

机密材料（如私钥）不应该提交到镜像里，因为移动或操纵镜像很容易泄露这些机密信息。相反地，任何对机密安全性严肃对待的系统都应该只在具有写保护机制或者更加强大的软件库或者硬件安全模块中存储机密。

分发项目使用了一些不同的机密材料：

- TLS 私钥
- SMTP 用户名和密码
- Redis 机密
- 各种远程存储账户 ID 和密钥对
- 客户端状态签名密钥

这一点很重要，也就是上面这些材料不应该提交到你的生产环境 Registry 配置中，或

者包含在任何你创建的镜像里。相反，应该考虑通过绑定加载卷注入秘密文件，这些卷可以挂载在 tmpfs 或者 RAMDisk 设备上，并设置受限的文件权限。直接源自配置文件的机密可以使用环境变量来注入。

前缀为 REGISTRY-的环境变量将被用作覆盖由分发项目加载的配置，配置变量是完全合格的，并且用下画线分割作为缩进级别。例如，客户端显示了配置文件中如下的机密：

```
http:
    secret: somedefaultsecret
```

可以用名为 REGISTRY-*HTTP*-SECRET 的环境变量来覆盖。如果你想在生产环境中启动一个运行分发项目的容器，你应该用-e flag 在 docker run 命令中注入机密：

```
docker run -d -e REGISTRY_HTTP_SECRET=<MY_SECRET> registry:2
```

出现了越来越多的集中式机密管理和机密分发项目，为了你的生产环境基础设施，你应该花一些时间来选择一个。

在生产环境中，日志配置设置得过于敏感会让磁盘或日志处理基础设施不堪重负，通过设置一个环境变量可以降低日志级别，设置 REGISTRY-*LOG*-LEVEL 到 error 或者 warn 级别：

```
docker run -d -e REGISTRY_LOG_LEVEL=error registry:2
```

下一个生产环境配置的区别就很简单了，禁用调试端点即可，可以使用环境变量的配置完成，设置 REGISTRY-*HTTP*-DEBUG 为一个空格字符串就可以确保分发不会启动一个调试端点了：

```
docker run -d -e REGISTRY_HTTP_DEBUG='' registry:2
```

当你部署 Registry 到生产环境时，你可能会需要移除本地文件系统的存储。本地文件系统存储的最大问题是特殊化。使用本地存储的 Registry 中的每一个镜像都是专门运行在某台计算机上的，这个特殊化降低了 Registry 的持久性，如果存储 Registry 数据的硬盘崩溃了，或者机器变得不能用了，那么你可能会经历数据丢失或者在最好的情况下可用性也会降低。

当波动是可接受的，或者涉及可复制镜像的使用案例时，在生产环境中使用本地存储是非常适当的。除了这些特定情况，你还需要数据持久性，所以你需要一个持久存储的后端。

10.3 持久化的 BLOB 存储

BLOB 是二进制大对象的缩写，Registry 处理的是镜像层（BLOB）和元数据，这个术语是相关的，因为有几个流行的 BLOB 存储服务和项目。为了采用持久的 BLOB 存储，你需要修改你的 Registry 配置文件并构建一个新的镜像。如何从一个集中式 Registry 发展到一个持久化的集中式 Registry，如图 10-6 所示。

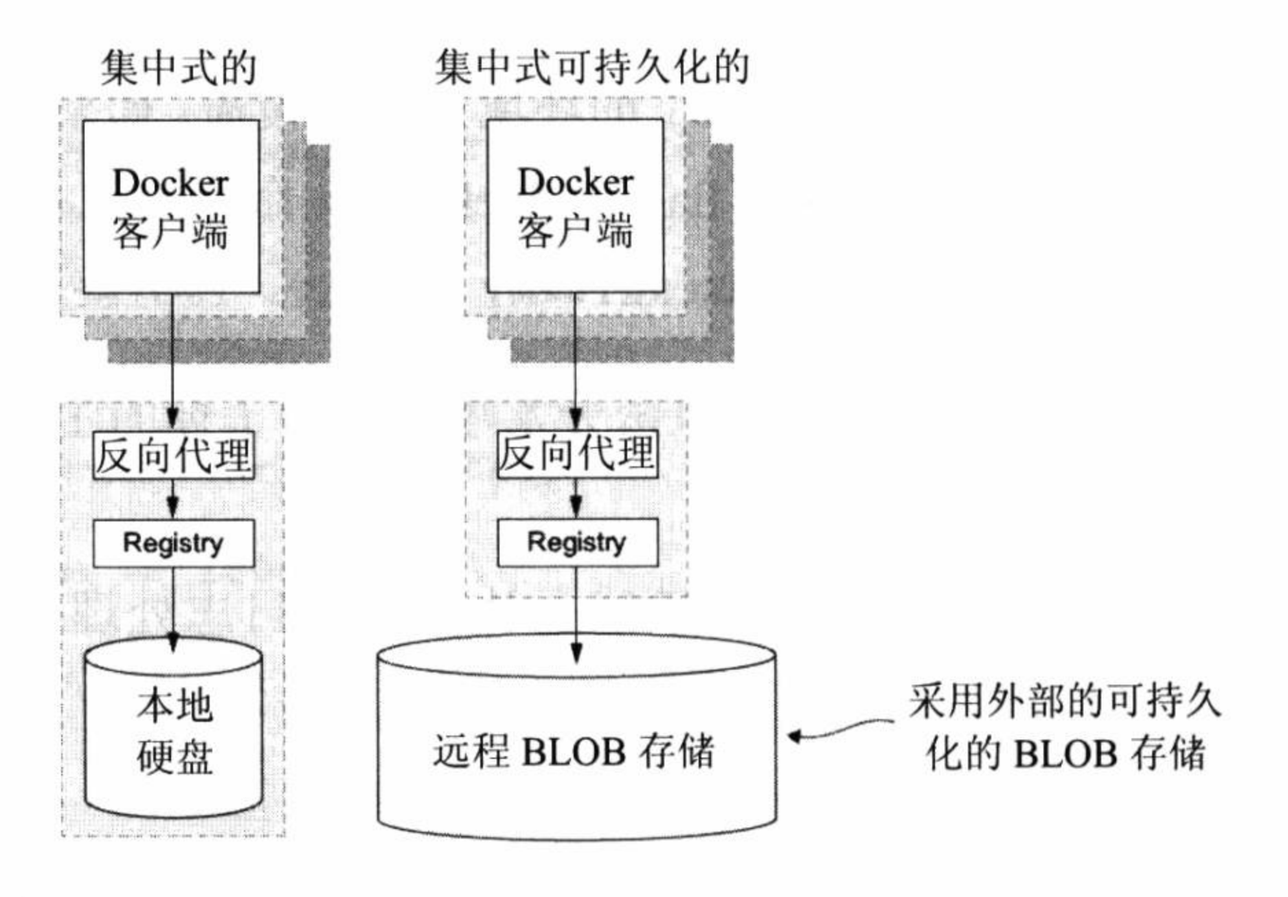


图 10-6 如何提高集中式 Registry 的持久性

除了本地文件系统，分发项目目前支持流行的 BLOB 存储后端。配置文件中处理存储的分段被恰如其分地命名为 `storage`，该映射有四个互相冲突的属性来定义存储后端，有效的 Registry 配置中只会出现其中一个属性：

- `filesystem`
- `azure`
- `s3`
- `rados`

默认配置使用的 `filesystem` 属性只有一个子属性 `rootdirectory`，它指定用于本地存储的基本目录，例如，下面是一个默认配置的示例：

```
storage:
    filesystem:
        rootdirectory: /var/lib/registry
```

其他的 storage 分段属性可以配置分发项目集成各种分布式 BLOB 存储服务，本节的剩余部分将会从 Azure 开始讨论。

10.3.1 微软 Azure 托管远程存储

Azure 是微软云服务产品家族的名称，其中的一个服务就是命名为 Storage 的 BLOB 服务。如果有 Azure 账号，你可以为你的 Registry BLOB 存储使用这个 Storage 服务。可以从网站 http://azure.microsoft.com/services/storage/了解有关这个服务的更多的信息。

为了在 BLOB 存储中采用 Azure，你需要使用 `azure` 属性并且设置三个子属性：`accountname`、`accountkey` 和 `container`。在此上下文中，container 是指 Azure 存储容器，而不是 Linux 容器。

一个最小的 Azure 配置文件可能命名为 azure-config.yml，包括以下配置：

```
# Filename: azure-config.yml
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    azure:                                          Azure-特定的字段
        accountname: <your account name>
        accountkey: <your base64 encoded account key>
        container: <your container>
        realm: core.windows.net
    cache:
        layerinfo: inmemory
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

用你的账户替换配置中的带括号的文本，realm 属性应该被设置为你想要存储的镜像的范围，有关详细信息，请参阅官方 Azure 存储文档。`realm` 并不是一个必需的属性，默认设置为 `core.windows.net`。

你可以使用以下的 Dockerfile，把新的配置打包到原生 Registry 镜像的上一层，命名为 azure-config.df：

```
# Filename: azure-config.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
```

```
# Set the default argument to specify the config file to use
# Setting it early will enable layer caching if the
# azure-config.yml changes.
CMD ["/azure-config.yml"]
COPY ["./azure-config.yml","/azure-config.yml"]
```

接着你可以用如下的 docker build 命令构建镜像：

```
docker build -t dockerinaction/azure-registry -f azure-config.df .
```

有了 Azure 存储后端，你可以构建一个持久化的 Registry，可以根据费用而不是技术复杂性伸缩，这个权衡是远程托管 BLOB 存储的其中一个长处。如果你对于使用托管方案有兴趣，可以考虑更成熟的 AWS S3。

10.3.2 AWS S3 托管远程存储

AWS 的 Simple Storage Service（S3）除了提供 BLOB 存储，还提供了多种特性。你可以配置 BLOB 在闲时进行加密、版本化、访问审计或者 AWS 的 CDN（参见 10.4.2 节）是否可用。

可以使用 S3 这个存储属性来采用 S3 作为托管远程 BLOB 存储，有四个必需的子属性：accesskey、secretkey、region 和 bucket，这些都是对你的账户进行身份认证和设置 BLOB 读写位置必需的属性，其他子属性指定分发项目应该如何使用 BLOB 存储，包括 encrypt、secure、v4auth、chunksize 和 rootdirectory。

设置 encrypt 属性为 true 时，将会对于你的 Registry 保存到 S3 的数据启用数据闲时加密功能，这是一个增强你的服务安全性能的免费功能。

secure 属性控制与 S3 通信时 HTTPS 协议的使用，默认是 false，此时使用 HTTP。如果你存储私有镜像材料，应该设置为 true。

v4auth 属性告知 Registry 使用 AWS 认证协议的 V4 版本，一般来说这应该设置为 true，但默认值为 false。

大于 5GB 的文件必须切分成更小的文件，并在 S3 的服务器端重组，但分块上传的文件应该小于 5GB，并且大于 100MB。文件块可以并行上传，上传失败的单个文件块可以单独撤下重发。分发项目及其 S3 客户端可以自动执行文件分割，但 chunksize 属性设置了文件需要进行分割的大小，超过此数值，文件就应该被分割，最小的文件块大小是 5MB。

最后，rootdirectory 属性设置在你的 S3 bucket 内 Registry 数据的根目录，如果你想从相同的 bucket 运行多个 Registry，这个设置是很有用的：

```
# Filename: s3-config.yml
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    cache:
        layerinfo: inmemory
    s3:                                  ← S3 配置
        accesskey: <your awsaccesskey>
        secretkey: <your awssecretkey>
        region: <your bucket region>
        bucket: <your bucketname>
        encrypt: true
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /s3/object/name/prefix
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

如果你已经创建了一个名为 s3-config.yml 的配置文件，并提供你的账户访问密钥、机密、bucket 名字和区域，你可以将更新后的 Registry 配置打包到一个新镜像，就像你对 Azure 所做的，用如下 Dockerfile：

```
# Filename: s3-config.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
# Set the default argument to specify the config file to use
# Setting it early will enable layer caching if the
# s3-config.yml changes.
CMD ["/s3-config.yml"]
COPY ["./s3-config.yml","/s3-config.yml"]
```

接着你可以用如下的 `docker build` 命令构建新镜像：

```
docker build -t dockerinaction/s3-registry -f s3-config.df .
```

无论是 S3 还是 Azure，都是基于使用的成本模型提供服务的，开始没有前期投入，许多较小的 Registry 都可以在两个服务的免费区间内运行。

如果你对托管数据服务不感兴趣，并且面对技术复杂性也毫不犹豫想去尝试，那么你可以考虑运行一个 Ceph 存储集群和 RADOS BLOB 存储后端。

10.3.3 RADOS（Ceph）的内部远程存储

可靠的自主分布式对象存储（RADOS）由名为 Ceph（ http://ceph.com）的软件项目提供。Ceph 是一个用来构建类似 Azure Storage 或者 AWS S3 的分布式 BLOB 存储服务的软件，如果你有预算、时间和专业知识，你可以部署自己的 Ceph 集群，长期来看可以省钱。比金钱更重要的是，运行你自己的 BLOB 存储可以让你掌控自己的数据。

如果你决定走这条路，你可以使用 rados 存储属性来与你自己的存储做集成：

```
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    cache:
        layerinfo: inmemory
storage:
    rados:                          ← RADOS 配置
        poolname: radospool
        username: radosuser
        chunksize: 4194304
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

三个子属性分别是 poolname、username 和 chunksize。username 属性是不需要加以说明的，但是 poolname 和 chunksize 是非常有趣的，值得一说。

Ceph 在池中存储 BLOB，池是被配置为有一定的冗余、分布式和行为。池将会指示 BLOB 是如何通过你的 Ceph 存储集群来存储的，poolname 属性告知 Registry 哪一个池被用来作为 BLOB 存储。

Ceph 块与 S3 块相似但不相同，Ceph 块对于 Ceph 的内部数据展示是非常重要的。Ceph 架构的概述可以在这里找到：http://ceph.com/docs/master/architecture/。默认的块大小是 4MB，但是如果你需要覆盖该值，可以使用 chunksize 属性。

采用分布式 BLOB 存储系统是构建持久化 Registry 的一个重要组成部分。如果你打算向外公开你的 Registry，你将需要改进快速的以及可伸缩的 Registry，下一节将解释如何实现这些改进。

10.4 扩展访问和延迟的改进

反向代理和后端的持久化存储到位之后，你应该就可以横向扩展你的 Registry 了，但是这样做会引入额外的延迟开销。如果你需要为每秒数千次的交易服务扩展 Registry，那么你就需要实现一个缓存策略，甚至可以考虑使用 Amazon CloudFront 这样的内容分发网络。

如图 10-7 所示，本节介绍两个新组件，它们将帮助你获得低延迟的响应时间。

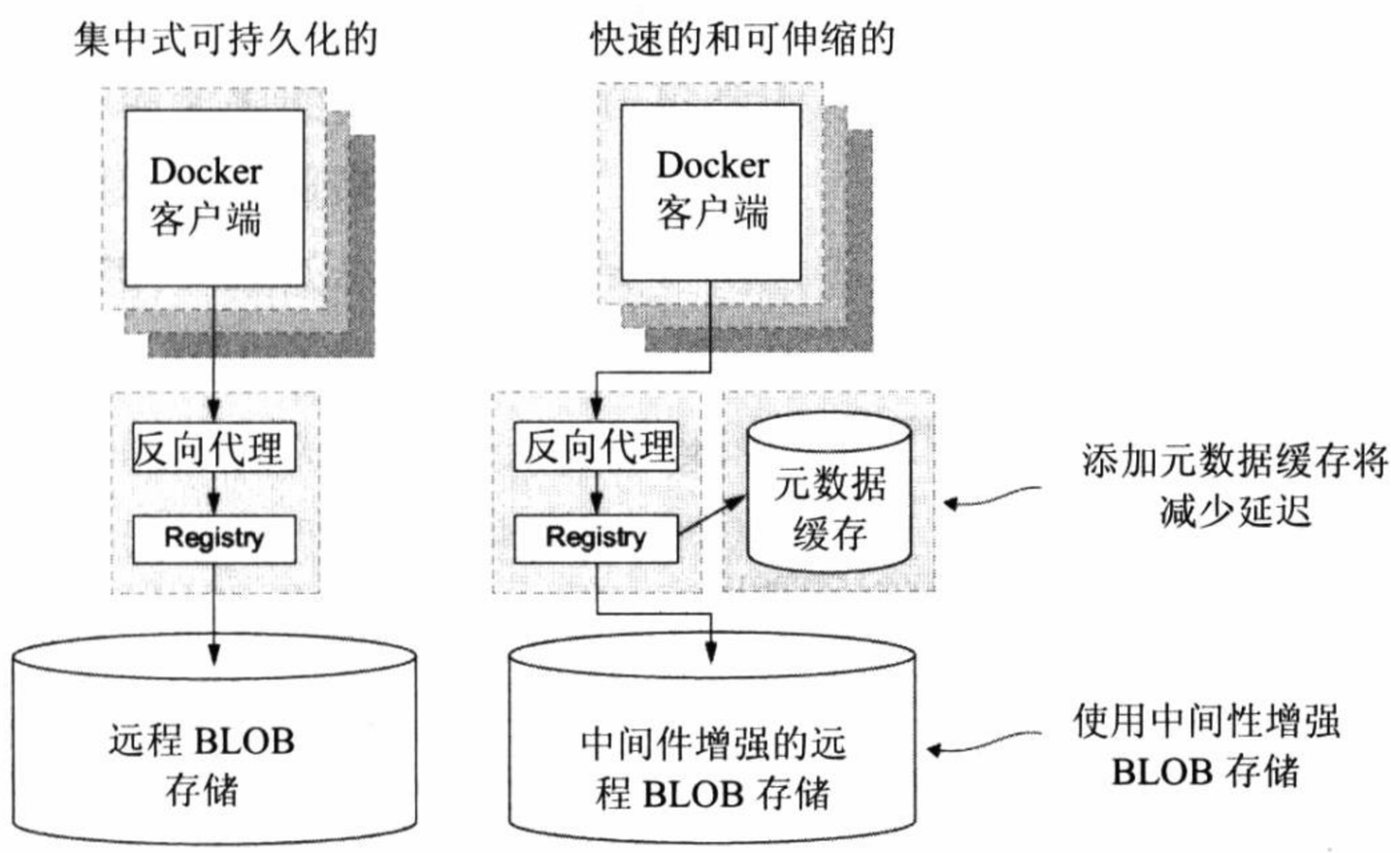


图 10-7 快速的和可伸缩的架构引入的元数据缓存和中间件

大多数读者不会为了运行这些练习而购买额外的机器，所以这些例子会让你使用独立的容器，而不是单独的机器。如果你在读本书的时候需要实现多机器 Registry，并且需要主机之间的链接软件的信息，请参考第 5 章。

10.4.1 与元数据缓存集成

当你需要低延迟的数据检索时，缓存是第一个拿手的工具，分发项目可以用内存映射或者 Redis 缓存仓库元数据。Redis（http://redis.io）是一个流行的开源键值对缓存和数据结构服务器。

内存映射选项比较适合较小的 Registry 或者开发用途，但使用一个专用的缓存项目，如 Redis，将有助于提高缓存的可靠性，并降低平均延迟。

分发项目中的元数据缓存配置可以用 `storage` 属性的 `cache` 子属性来设置，而 `cache` 有一个名为 `blobdescriptor` 的子属性，该属性有两个潜在的值，分别是

inmemory 和 redis。如果你使用 inmemory，那么设置该值是唯一需要的配置，但是如果你使用 redis，你需要提供额外的连接池配置。

顶层的 redis 属性只有一个必要的 addr 子属性，该属性指定了用于缓存的 Redis 服务器的位置，这个服务可以在同一台机器或者不同的机器上运行，但是如果你使用了本地主机名称，那么就必须在同一个容器或者加入网络的另外一个容器上运行。使用一个已知的主机别名可以让你灵活地代理一个在运行时配置的连接，在以下配置示例中，Registry 将尝试连接到一个 redis-host 端口为 6379 的 Redis 服务：

```
# Filename: redis-config.yml
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
storage:
    cache:                                  ← 缓存配置
        blobdescriptor: redis
    s3:
        accesskey: <your awsaccesskey>
        secretkey: <your awssecretkey>
        region: <your bucket region>
        bucket: <your bucketname>
        encrypt: true
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /s3/object/name/prefix
    maintenance:
        uploadpurging:
            enabled: false
redis:                                      ← Redis 特定详细信息
    addr: redis-host:6379
    password: asecret
    dialtimeout: 10ms
    readtimeout: 10ms
    writetimeout: 10ms
    pool:
        maxidle: 16
        maxactive: 64
        idletimeout: 300s
```

本示例的 redis 配置设置了数个可选的属性。passoword 属性定义了在连接时传给 Redis AUTH 命令的密码，dialtimeout、readtimeout 和 writetimeout 属性指定了连接、读取和写入 Redis 服务的超时值，最后一个属性 pool 有三个子属性，定义了连接池的属性。

池大小的最小值可以用 maxidle 属性指定，而最大值用 maxactive 属性设置。最后，从最后一次活跃连接的使用到候选进入闲置状态的时间可以用属性 idletimeout 指定，在当前的闲置连接数目达到最大值时，任何进入闲置状态的连接将会被关闭。

为了生成与环境无关的镜像，可以在机密的地方使用虚拟值，类似于 password 的属性应该在运行时用环境变量来覆盖。

缓存配置将有助于减低与 Registry 元数据服务相关的延迟，但镜像 BLOB 服务仍然是低效的。通过与远程 BLOB 存储，如 S3 集成，Registry 在镜像传输过程中成为流媒体瓶颈。流媒体连接是棘手的，因为相对于元数据查询而言，连接往往是长寿命。当长连接通过相同的负载均衡基础设施处理为短连接时，事情变得更为棘手。

你可以借助这个配置构建一个 Registry 并链接到一个 Redis 容器：

```
docker run -d --name redis redis
docker build -t dockerinaction/redis-registry -f redis-config.df .
docker run -d --name redis-registry \
    --link redis:redis-host -p 5001:5000 \
    dockerinaction/redis-registry
```

下一节将解释如何使用内容分发网络（CDN）和 Registry 中间件简化 BLOB 下载。

10.4.2 使用存储中间件简化 BLOB 传输

中间件在分发项目中所起的作用就像一个 Registry、仓库或者存储设备的装饰师。目前分发项目携带一个单一的存储中间件，它将你的 Registry 与后端的 S3 存储通过 AWS CloudFront 进行了集成，CloudFront 是一个内容分发网络。

CDN 是专为地理感知网络访问文件而设计的，这使得 CloudFront 对于剩余的伸缩性问题是一个完美的解决方案，这些问题是由于采用持久化和分布式 BLOB 存储而产生的。

启用 CloudFront 中间件和后端的 S3 存储可以使得文件的下载简化，如图 10-8 所示演示了数据如何通过集成配置流动。

相比于 BLOB 从 S3 回流到你的 Registry，随后回到发出请求的客户端，集成 CloudFront 可以让你直接重定向客户端到已认证的 CloudFront URL。这就消除了下载镜像到你的本地的网络开销，它也可以避免使用长连接，而是采用适当设计的 CloudFront 服务。

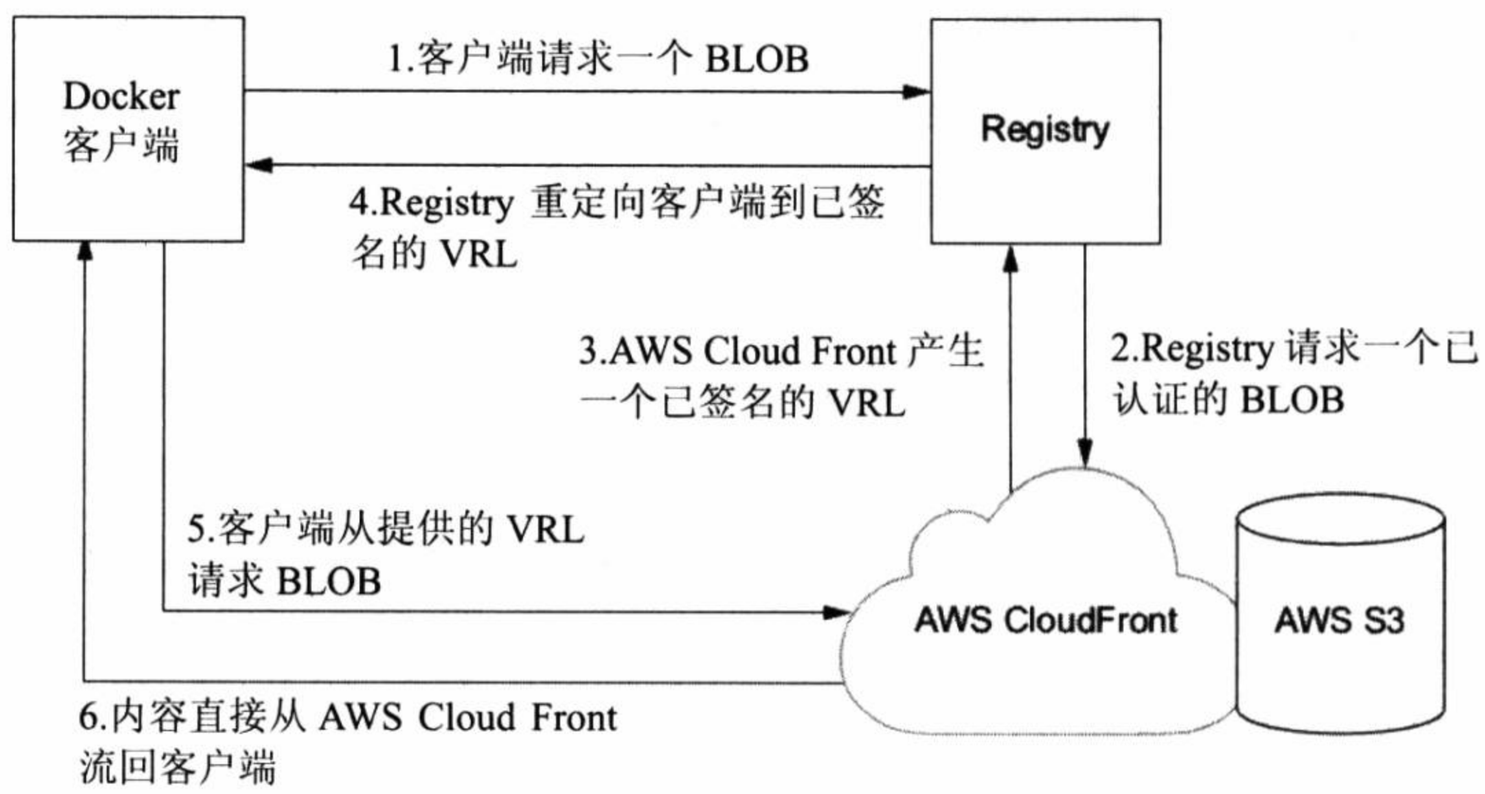


图 10-8 用 AWS CloudFront 存储中间件消除流处理 BLOB 流量

启用 CloudFront 中间件就像添加一个合适的配置那样简单，下面的示例使用 S3、Redis 和 CloudFront 完成：

```
# Filename: scalable-config.conf
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
storage:
    cache:
        blobdescriptor: redis
    s3:
        accesskey: <your awsaccesskey>
        secretkey: <your awssecretkey>
        region: <your bucket region>
        bucket: <your bucketname>
        encrypt: true
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /s3/object/name/prefix
    maintenance:
        uploadpurging:
            enabled: false
```

```
redis:
    addr: redis-host:6379
    password: asecret
    dialtimeout: 10ms
    readtimeout: 10ms
    writetimeout: 10ms
    pool:
        maxidle: 16
        maxactive: 64
        idletimeout: 300s
middleware:
    storage:
        - name: cloudfront
          options:
              baseurl: <https://my.cloudfronted.domain.com/>
              privatekey: </path/to/pem>
              keypairid: <cloudfrontkeypairid>
              duration: 3000
```

中间件配置

属性 middleware 以及子属性 storage 有点不同于目前为止你已经看到的其他配置，子属性 storage 包含一个名为 storage middleware 的列表，每个都有其自己的一组特定中间件的选项。

在这个示例中，你使用名为 cloudfront 的中间件，设置它的 baseurl、privatekey 路径、keypairid 名称和认证 URL 为有效的 duration。查阅 CloudFront 用户文档（http://aws.amazon. com/cloudfront）了解如何正确设置你的账户。

一旦你添加了一个特定于你的 AWS 账户和 CloudFront 分发的配置，你可以用 Dockerfile 打包配置，并部署任意数量的高性能 Registry 容器。借助于适当的硬件和配置，你可以扩展到支持每秒拉取或者推送数以千计的镜像的水准。

所有这些活动都生成有用的数据，类似于你的 Registry 的组件应该集成剩余的系统来对事件作出反应或者集中式地收集数据，分发项目使得集成通知成为可能。

10.5 通过通知集成

推出自己的 Registry 可以帮助你建立自己的镜像分发基础设施，但这样做你需要集成其他一些工具。通知是一个简单的 Webhook 类型的集成工具。当你在 Registry 配置文件中提供一个端点定义时，Registry 将会生成一个 HTTP 请求并为其中的每一次拉取或者推送上传一个 JSON 编码的事件，如图 10-9 所示，通知是如何与系统架构集成的。

当镜像分发系统配置为发送通知时，任何有效的推送或者拉取事件都会触发描述该事件的 JSON 文档传递到已配置的端点，这是分发系统主要的集成机制。

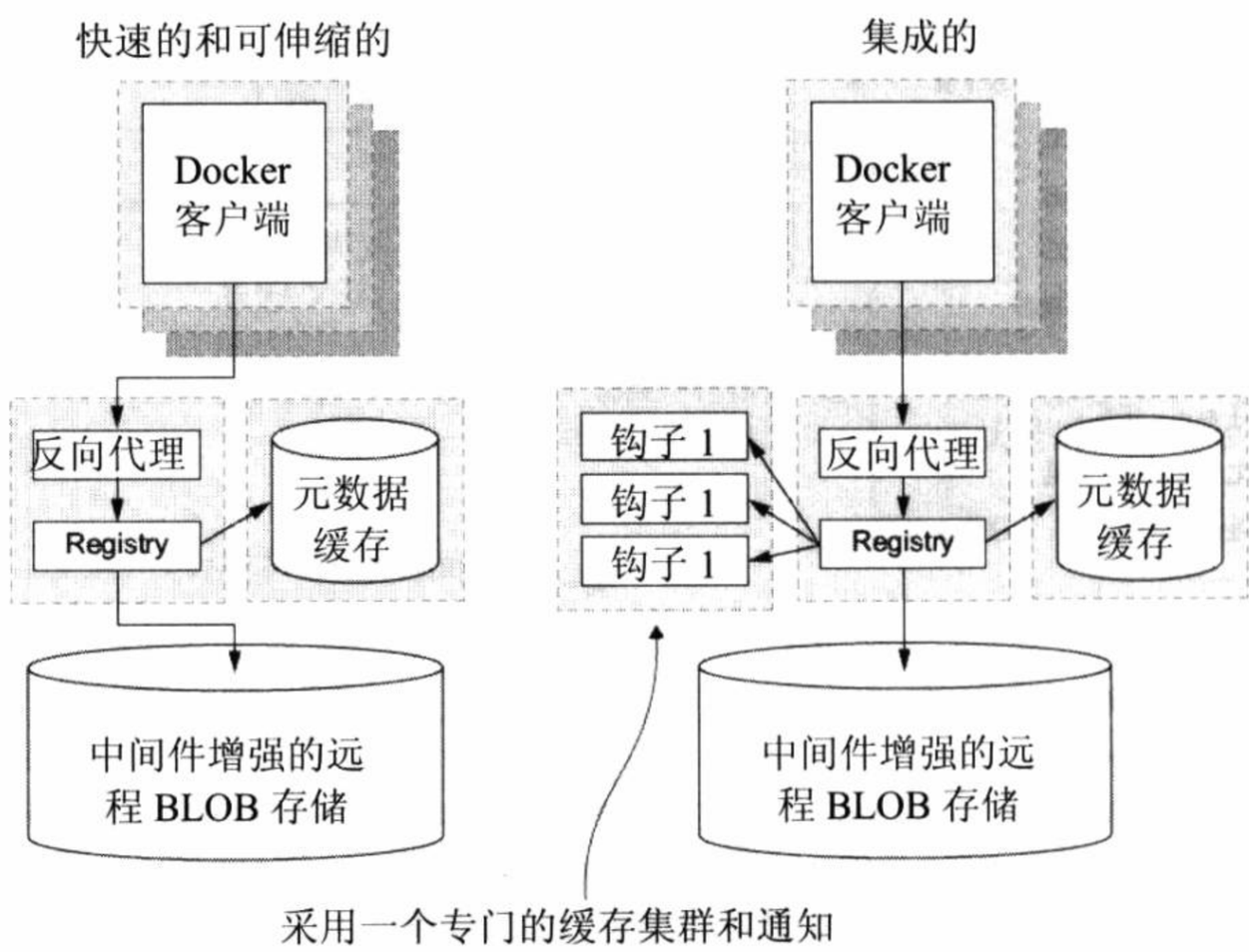


图 10-9 通知、报告和一个专用的元数据缓存

通知可以用来收集使用情况指标、触发部署、触发镜像重新构建、发送电子邮件或者做其他任何你能想到的事。你可以使用通知集成发布消息到你的 IRC 频道，重新生成用户文档或者触发扫描仓库的索引服务。在本章最后的这个例子中，你将把分发项目和 Elasticsearch 项目（https://github.com/elastic/elasticsearch）以及一个 web 接口集成来创建一个完全可搜索的 Registry 事件数据库。

Elasticsearch 是一个可伸缩的文档索引数据库，它提供了运行你自己的搜索引擎所有必需的功能，其中 Calaca 是一个流行的 Elasticsearch 开源 web 界面。在本例中，你将在自己的容器内运行每一个实例，如一个用 Node.js 实现的 pump 实例，一个分发 Registry 被配置为发送通知到这个 pump。如图 10-10 所示，本示例为你将构建的集成。

为了构建该系统，你将使用来自 Docker Hub 的官方 Elasticsearch 镜像，以及为本示例提供的两个镜像。所有的这些材料都是开源的，所以如果你对工作机制感兴趣，请检查仓库，集成和分发配置的细节也在这里有所涉及，请准备从 Docker Hub 拉取所需的镜像：

```
docker pull elasticsearch:1.6
docker pull dockerinaction/ch10_calaca
docker pull dockerinaction/ch10_pump
```

简单地说，`dockerinaction/ch10-calaca` 镜像包含一个基础的 Calaca 发行版，其被配置为使用一个在本地主机运行的 Elasticsearch 节点。在遵守跨域资源共享（CORS）规则的情况下，这个名字是很重要的。`dockerinaction/ch10-pump` 镜像包含一个小的 Node.js 服务，该服务监听通知，并将那些包含了仓库清单里的拉取或者推送动作的通知转发。这代表了一小部分由 Registry 发送的通知类型的子集。

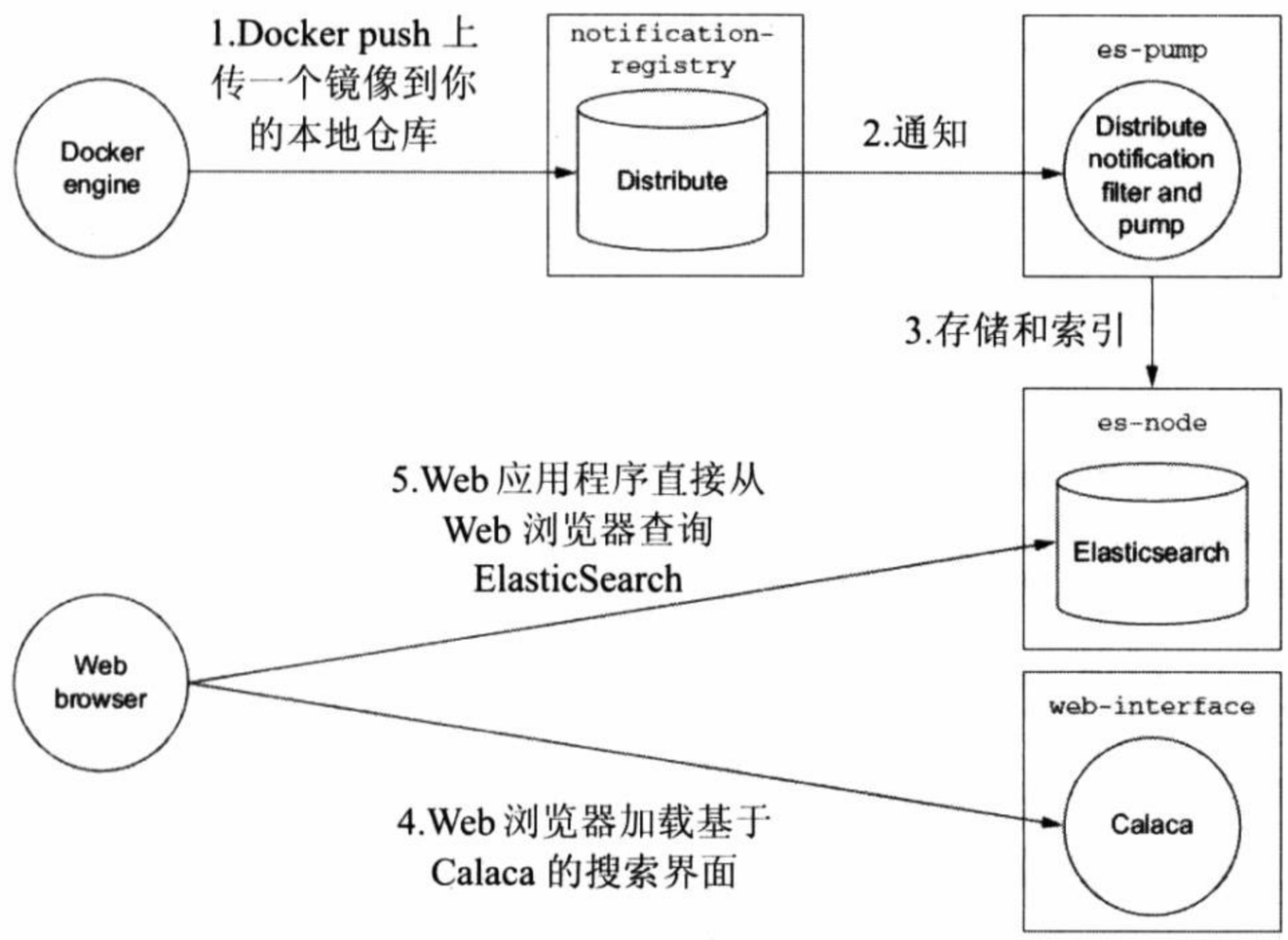


图 10-10 将分发系统与 ElasticSearch 集成

Registry 上的每个有效的动作都会导致一个通知，包括如下所示：

- 仓库清单上传和下载
- BLOB 元数据请求、上传和下载

通知以 JSON 对象的方式传递，每个通知包含一系列的事件，每个事件包含了描述事件的属性，下面显示了可用的属性：

```
{ "events": [{
  "id": "921a9db6-1703-4fe4-9dda-ea71ad0014f1",
  "timestamp": ...
  "action": "push",
  "target": {
    "mediaType": ...
    "length": ...
    "digest": ...
    "repository": ...
    "url": ...
  },
  "request": {
    "id": ...
    "addr": ...
    "host": ...
    "method": ...
    "useragent": ...
  },
  "actor": {},
  "source": {
    "addr": ...
    "instanceID": ...
  }
}]}
```

dockerinaction/ch10_pump 容器中的服务会检查事件列表中的每个元素，然后将合适的事件转发到 ElasticSearch 节点。

现在你已经了解 pump 是如何工作的了，启动 ElasticSearch 和 pump 容器：

```
docker run -d --name elasticsearch -p 9200:9200 \
    elasticsearch:1.6 -Des.http.cors.enabled=true

docker run -d --name es-pump -p 8000 \
    --link elasticsearch:esnode \
    dockerinaction/ch10_pump
```

可以通过传递环境变量到 Elasticsearch 程序本身，从而不需要创建一个完整的镜像就可以定制化由 Elasticsearch 镜像创建的容器，在前面的命令中启用 CORS 头部，这样你就可以将这个容器与 Calaca 集成。有了这些适当的组件，任意数量的分布式实例都可以发送通知到 es-pump 容器，所有相关的数据都存储在 Elasticsearch。

接下来，创建一个容器来运行 Calaca web 接口：

```
docker run -d --name calaca -p 3000:3000 \
    dockerinaction/ch10_calaca
```

注意运行 Calaca 的容器不需要链接到 Elasticsearch 容器，而是从 web 浏览器使用一个直接到了 Elasticsearch 节点的链接。在这种情况下，所提供的镜像配置为使用运行在本地主机上的 Elasticsearch 节点，如果你运行 VirtualBox，下一步可能会非常棘手。

VirtualBox 用户在技术上没有绑定 elasticsearch 容器的端口到本地主机，相反绑定到的是 VirtualBox 虚拟机的 IP 地址。你可以使用包含在 VirtualBox 里边的 VBoxManage 程序来解决这个问题，使用这个程序来创建你的主机和默认虚拟机之间的端口转发规则，你可以用两个命令创建你所需要的规则：

```
VBoxManage controlvm "$(docker-machine active)" natpf1 \
    "tcp-port9200,tcp,,9200,,9200"
VBoxManage controlvm "$(docker-machine active)" natpf1 \
    "tcp-port3000,tcp,,3000,,3000"
```

这些命令创建了两个规则：转发本地主机的 9200 端口到默认虚拟机的 9200 端口，同样的对于端口 3000 也一样。现在 VirtualBox 用户就可以跟原生的 Docker 用户一样以相同的方式与这些端口交互。

此时，你应该准备好配置和启动一个新的 Registry 了。对于这个示例，可以从默认的 Registry 配置开始并简单地添加一个 notification 分段，创建一个新文件并复制以下配置：

```
# Filename: hooks-config.yml
version: 0.1
log:
    level: debug
    formatter: text
    fields:
        service: registry
        environment: staging
storage:
    filesystem:
        rootdirectory: /var/lib/registry
    maintenance:
        uploadpurging:
            enabled: true
            age: 168h
            interval: 24h
            dryrun: false
http:
    addr: 0.0.0.0:5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
notifications:                          ← 通知配置
    endpoints:
        - name: webhookmonitor
          disabled: false
          url: http://webhookmonitor:8000/
          timeout: 500
          threshold: 5
          backoff: 1000
```

最后一节的 notification 指定了需要通知的端点的列表，每个端点的配置包括一个名称、URL、尝试超时时间、尝试阈值和重试时间。你也可以通过设置 disabled 属性为 false 来禁用单个端点，而不需要删除配置。在这种情况下，你已经在 webhookmonitor 的 8000 端口上定义了一个端点，如果你正在将其部署到分布式环境中，webhookmonitor 可能被设置为针对不同的主机。本示例中，webhookmonitor 是运行 pump 实例的容器的一个别名。

一旦你保存了配置文件，你就可以启动新的 Registry 容器并在运行过程中看到通知。下面的命令将创建一个 Registry，它使用一个基础镜像，使用绑定挂载卷注入配置，最后一个参数是对要使用的配置文件进行设置。该命令创建一个连接到 pump 容器，并分配为别名 webhookmonitor。最后，它将 Registry 绑定到本地主机（或者 Boot2Docker IP 地址）的 5555 端口：

```
docker run -d --name ch10-hooks-registry -p 5555:5000 \
    --link es-pump:webhookmonitor \
    -v "$(pwd)"/hooks-config.yml:/hooks-config.yml \
    registry:2 /hooks-config.yml
```

有了最后一个运行的组件，你就可以测试系统。首先测试 Calaca 容器。打开一个 web 浏览器并导航到 http://localhost:3000/，你应该看到一个简单的网页，上面的标题为 Calaca，里边有一个很大的搜索框。此时搜索不会有任何结果，因为还没有通知发送到 Elasticsearch，从你的仓库推送和拉取镜像可以看到 Elasticsearch 和通知是如何工作的。

给你现有的一个镜像打上标签并推送到你的新 Registry 会触发通知，你也许会考虑使用在前面章节创建的 dockerinaction/curl 镜像，这是一个很小的镜像，测试非常快：

```
docker tag dockerinaction/curl localhost:5555/dockerinaction/curl
docker push localhost:5555/dockerinaction/curl
```

如果你将 cURL 镜像命名为一个不一样的名字，你需要使用该名字而不是这里提供的，否则，你应该准备好搜索，在你的 web 浏览器中回到 Calaca 并在搜索框中输入 curl。

当你输入 curl 后，应该会出现一个搜索结果，这里列出的结果对应于 Registry 事件。任何有效的推送或者拉取都将在 Elasticsearch 触发创建这样的事件，Calaca 配置为显示仓库名称、事件类型、事件的时间戳，最后就是原生通知。如果你再次推送到相同的仓库，那么应该会有两个事件。相反，如果你从仓库拉取镜像，还有针对 curl 搜索词的第三个事件，但类型是 pull，自己试试看：

```
docker pull localhost:5555/dockerinaction/curl
```

原生通知都包含在搜索结果里，从而借助搜索事件可以帮助你获得一些有创造性的想法。Elasticsearch 可以对整个文档做索引，所以事件的任何字段都是一个潜在的搜索词，试着用这个例子搭建有趣的查询：

- 搜索 pull 或者 push，查看所有的拉取或者推送事件。
- 寻找一个特定的仓库前缀，获得带有这个前缀的所有事件的列表。
- 根据特定的镜像指纹追踪活动。
- 通过请求一个 IP 地址发现客户端。
- 发现客户端访问的所有仓库。

这个冗长的示例应该增强基于分发式的 Registry 作为你的发行版或者部署工作流程的关键组件的潜力，这个例子也应该提醒人们 Docker 是如何减少实现容器化技术的障碍的。

设置这个示例最复杂的部分在于创建容器，链接容器并通过卷注入配置，在第 11 章你将学习使用一个简单的 YAML 文件以及一个单一的命令是如何在这个示例上进行设置和迭代的。

10.6 小结

本章深入探讨了如何从分发项目构建一个 Docker Registry，这个信息对于打算部署自

己的 Registry 以及想要开发一个可以有深入了解的主镜像分发渠道的读者是非常重要的，该章具体涵盖了以下几点：

- 一个 Docker Registry 是由其公开的 API 定义的，分发项目是一个对于 Registry API V2 的开源实现。
- 运行你自己的 Registry 很简单，从 `registry:2` 镜像启动一个容器即可。
- 分发工程通过 YAML 文件配置。
- 实现有多个客户端的集中式的 Registry，通常需要实现一个反向代理，采用 TLS，并添加身份认证机制。
- 身份认证可以移到反向代理或者由 Registry 本身实现。
- 虽然有其他身份认证机制可用，HTTP 基础身份认证是最简单的配置，并最受欢迎。
- 反向代理层可以帮助解决 Registry API 对于多个客户端版本的兼容性问题。
- 在生产环境中通过绑定加载卷和环境变量配置覆盖来注入机密材料，不要提交机密材料到镜像里。
- 集中式 Registry 考虑采用远程 BLOB 存储，比如 Azure、S3 或者 Ceph。
- 分发项目可以通过创建一个元数据缓存（基于 Redis）或者采用 Amazon web 服务 CloudFront 存储中间件这两种方式配置为可伸缩的。
- 将分发项目与你剩下的部署项目、分布式系统和数据中心基础设施通过通知集成，非常简单。
- 通知以 JSON 格式推送事件数据到已配置的端点。

第 3 部分

多容器和多主机环境

如果说本书第 1 部分聚焦于容器提供的隔离，这部分将会聚焦于容器的组合，最有价值的系统是由两个或两个以上的组件组成的。由于大规模服务器软件、SOA 架构、微服务以及目前物联网的崛起，多组件的简单管理比以往任何时候都更重要。在构建这些系统时，我们采用诸如集群计算、编排和服务发现这样的工具。这些都是非常困难和微妙的问题。本书的最后一部分将介绍另外三个 Docker 工具并展示如何在实际环境中使用 Docker。

第 11 章　Docker Compose 声明式环境

本章介绍

- 使用 Docker Compose
- 操作环境和项目迭代
- 扩展服务和清理
- 构建声明式环境

你曾经有没有加入一个已有项目的团队，并努力设置你的开发环境或者配置 IDE？如果有人要你为他们的项目提供一个测试环境，你能列举所有你需要的问题来完成工作吗？你能想象当环境发生变化时，开发团队和系统管理员要重新同步是多么痛苦吗？这些都是常见的需要大量付出的任务，并且往往耗时很长而价值不太大。在最坏的情况下，他们引发的策略或规程限制了开发的灵活性，放慢了迭代周期，并带来阻力最小的前沿技术决策路径。

本章向你介绍 Docker Compose（也称为 Compose），以及如何使用它来解决这些常见的问题。

11.1　Docker Compose：第一天的启动并运行

Compose 是一个用于定义、启动和管理服务的工具，其中一个服务可以定义为 Docker 容

器的一个或多个副本。在 YAML 文件（http://yaml.org）中定义了服务和服务系统，并通过命令行 docker-compose 进行管理，有了 Compose，可以使用简单的命令来完成下面这些任务：

- 构建 Docke 镜像
- 启动容器化的应用及服务
- 启动完整的服务系统
- 管理系统中单个服务的状态
- 服务伸缩
- 查看生成服务的容器的收集日志

Compose 让你停止专注于单个的容器，而是描述完整的环境以及服务组件的交互。一个 Compose 文件可能会描述四到五个单独的服务，它们都是相互关联的，但应保持隔离和独立伸缩。这一层次的互动涵盖了大部分的系统管理日常使用案例，出于这个原因，大多数与 Docker 的交互都将通过 Compose。

此时几乎可以肯定你安装了 Docker，但是你可能没有安装 Compose。你可以在 https://docs.docker.com/compose/install/找到针对你的环境的最新的安装说明，在本文写作时，官方还没有实现支持 Windows。但很多用户已经通过 Pip（一个 Python 包管理器）成功地安装了 Compose。检查官方网站的最新信息，你会惊喜地发现 Compose 是一个二进制文件，安装说明很简单，现在花些时间安装 Compose。

对任何工具发挥优势的最佳方法是使用它，本节的其余部分将带你从一些情景化的例子开始。

11.1.1 用一个简单的开发环境入门

假设你以在成熟项目的团队中的一名软件开发人员的身份开始了一个新工作，如果在以往的这种类似情况下，你可能会觉得要花几天时间安装和配置 IDE，并在你的工作计算器上运行一个可操作的开发环境，但是在这里工作的第一天，你的同事给了你三个简单的指令就可以开始了：

- 安装 Docker
- 安装 Docker Compose
- 安装和使用 Git 来克隆开发环境

而不是要求你在这里克隆一个开发环境，我会让你创建一个名为 wp-example 的新目录，并复制以下的 docker-compose.yml 文件到这个目录：

```
# Filename: docker-compose.yml
wordpress:
  image: wordpress:4.2.2
  links:
    - db:mysql
  ports:
    - 8080:80
db:
  image: mariadb
  environment:
    MYSQL_ROOT_PASSWORD: example
```

定义命名为 wordpress 服务

使用官方 wordpress4.2.2 镜像

链接依赖到 db 服务上

映射容器端口 80 到宿主机端口 8080

定义命名为 db 的服务

使用官方的 mariadb:latest 镜像

通过环境变量设置数据库系统管理员密码

正如上述文件中告知的信息，将会启动一个 WordPress 服务和一个独立的数据库，这是第 2 章中一个基础示例的迭代，来到你创建 docker-compose.yml 文件的目录并用以下的命令启动：

```
docker-compose up
```

这将产生如下的输出：

```
Creating wpexample_db_1...
Creating wpexample_wordpress_1...
...
```

你应该能够在 web 浏览器中打开 http://localhost:8080/（或者将 localhost 替换为你的虚拟机的 IP 地址），就会发现一个新的 WordPress 安装界面。这个例子是相当简单的，但是可以描述多服务架构。想象一个典型的三层或四层 web 应用程序，包括 web 服务器、应用程序代码、数据库，也许还有一个缓存。启动一个这样环境的本地副本通常会花几天时间——如果负责整个工作的人对于一些组件还不熟悉的话，有了 Compose，你就可以简单地获取 docker-compose.yml，并运行 `docker-compose up` 命令。

当你开心地完成了 WordPress 实例后，你应该清理一下环境。你可以通过按【Ctrl】+【C】或【Control】+【C】组合键来关闭整个环境。在删除你创建的容器之前，花点时间用 `docker` 和 `docker-compose` 命令列出这些容器：

```
docker ps
docker-compose ps
```

使用 `docker` 以标准的方式显示两个（或更多）容器的列表，但是使用 `docker-compose` 命令列出的容器只包括那些在当前目录下 docker-compose.yml 文件中定义的容器，这种方式更加优雅和简洁。以这种方式过滤列表还能帮助你专注于那些组成你当前工作环境的容器。在继续学习本章之前，请花些时间去清理你的环境。

Compose 有个 `rm` 命令，非常类似于 `docker rm` 命令，区别在于 `docker-compose rm` 命令会删除所有的服务或者一个由环境定义的特定服务。另一个小的区别是，`-f` 选项

并不强迫删除正在运行的容器，相反，它会关闭用户确认阶段。

所以，清理的第一步是关闭环境，你可以使用 `docker-compose stop` 或者 `docker-compose kill` 命令达到此目的。使用 `stop` 优先于 `kill`，原因在第 1 部分有解释。就像其他的 Compose 命令，这两个命令可以传入一个服务名字作为要关闭的目标。

一旦你停止了服务，需要使用 `docker-compose rm` 命令来完成清理工作。记住，如果你省略-v 选项，卷可能成为孤立的：

```
docker-compose rm -v
```

Compose 会显示将被删除的容器列表，并提示你确认，按【Y】键继续。随着这些容器的删除，你将准备学习 Compose 是如何在管理状态以及迭代时避免孤立服务的技巧。

这个 WordPress 示例很简单，接下来，你将看到如何使用 Compose 来塑造更加复杂的环境。

11.1.2 一个复杂的架构：分布式系统和 Elasticsearch 的集成

在第 10 章的末尾，你创建了一个更加复杂的示例，你启动了由四个相关的组件组成的 Docker Registry，该 Registry 被配置为将事件数据打入一个 Elasticsearch 实例，并提供一个 web 界面搜索这些事件，如图 11-1 所示。

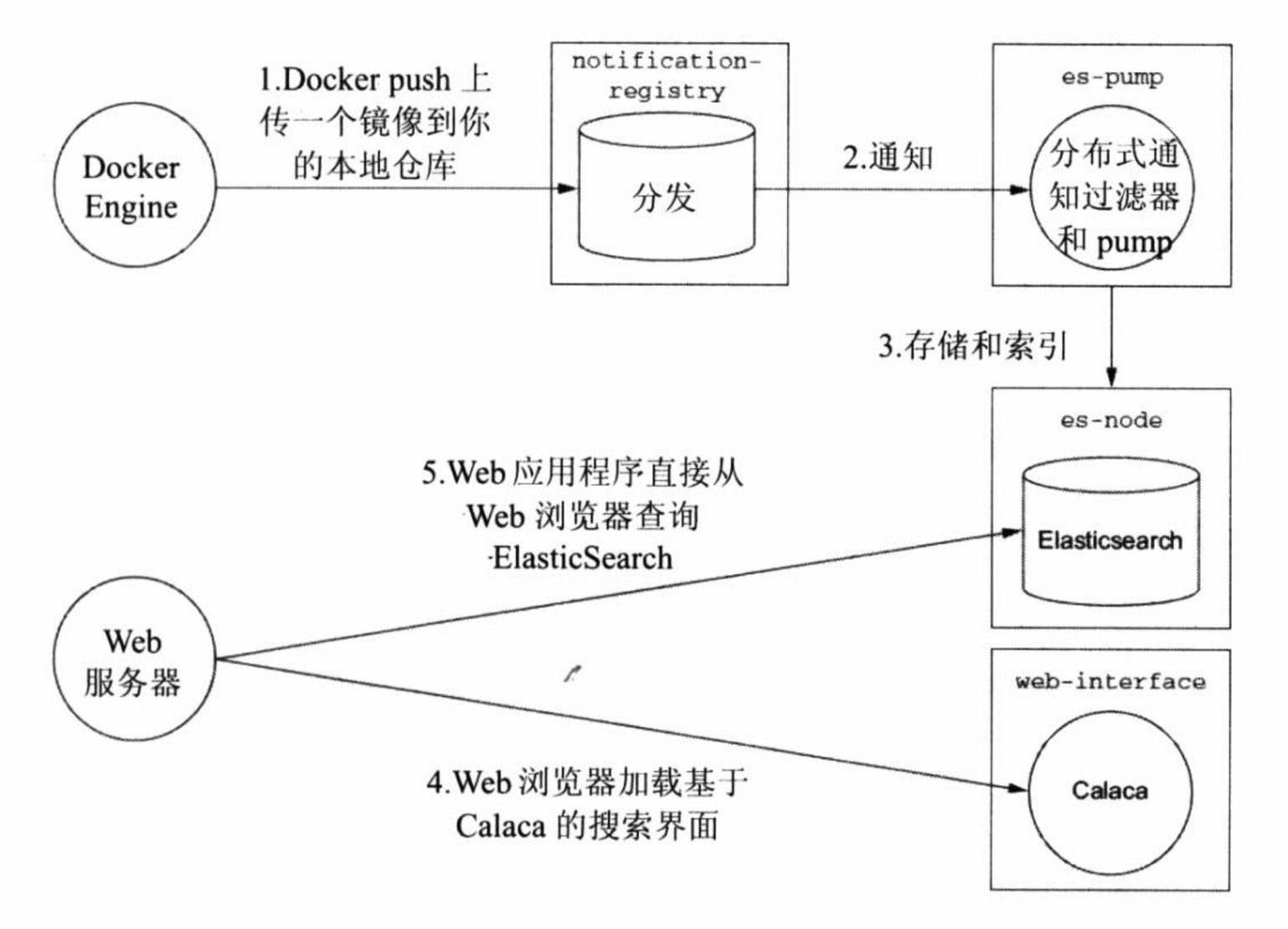


图 11-1 数据流经由四个容器组成的通知 Registry 的示例

设置这个示例需要镜像构建，并需要仔细衡量这些容器如何链接在一起的。你可以通过从版本控制系统克隆已有的环境快速重新创建这个示例，然后使用 Compose 启动：

```
git clone https://github.com/dockerinaction/ch11_notifications.git
cd ch11_notifications
docker-compose up -d
```

当你运行最后一个命令时，Docker 将会进入构建各种镜像和启动容器的生命周期，它与第一个例子的不同之处在于你使用-d 选项，这个选项在 datached 模式下启动容器，它的运作与用-d 选项运行 docker run 命令完全一样。当容器分离时，每个容器输出的日志将不会流向终端。

如果你需要访问这些数据，你可以针对特定的容器使用 docker log 命令，但如果你运行多个容器，这并不能很好地扩展。相反，应该使用 docker-compose logs 命令来获得所有容器或者 Compose 管理的一些服务子集聚合而成的日志流。例如，如果你想看到所有服务的所有日志，运行如下：

```
docker-compose logs
```

这个命令会自动跟踪日志，所以当你完成后，按【Ctrl】+【C】或【Control】+【C】组合键来退出。如果你想只看到一个或多个服务，那么就给出那些服务的命名：

```
docker-compose logs pump elasticsearch
```

在本例中，你用一个命令启动了完整的环境，并用一个命令查看，能够在如此高的层面运行很不错，但更强大的事实是你也拥有各种源代码并可以同样轻松地在本地迭代。

假设你有另一个想绑定在 3000 端口的服务，这与本示例的 calaca 服务冲突。改变非常简单，只要改变 ch11notifications/docker-compose.yml 文件，并再次运行 docker-compose up 命令，看一下文件：

```
registry:
  build: ./registry
  ports:
    - "5555:5000"
  links:
    - pump:webhookmonitor

pump:
  build: ./pump
  expose:
    - "8000"
  links:
    - elasticsearch:esnode

elasticsearch:
  image: elasticsearch:1.6
  ports:
    - "9200:9200"
```

映射 registry 到宿主机 5555 端口

链接 registry 到 pump 服务

公开端口 8000 到依赖的服务

链接 pump 到 elasticsearch 服务

使用官方 elasticsearch 镜像

映射 elasticsearch 服务的 9200 端口

```
calaca:
  build: ./calaca
  ports:
    - "3000:3000"
```

为 calaca 服务使用本地源码

映射主机端口 3000 到 calaca 服务端口 3000

改变最后一行，从读取 3000:3000 变为 3001:3000，并保存文件，随着这个变化，你可以很简单地通过再次运行 docker-compose up -d 命令重新构建环境。当你执行这个命令后，它将停止当前并删除运行的容器，接着创建新的容器，并重新附着前代环境挂载的数据卷，在可能的情况下，Compose 将限制那些发生更改的已重启的容器的范围。

如果你的服务源代码发生了变化，可以用一个命令重建一个或所有服务，运行如下命令：

```
docker-compose build
```

如果你只需要重建你的一个服务或者服务的一些子集，然后简单地命名服务。这个命令将重建 calaca 和 pump 服务：

```
docker-compose build calaca pump
```

此时此刻，停止和删除你为这些服务创建的容器：

```
docker-compose rm -vf
```

通过使用这些例子，你已经接触了大量的开发工作流程，并有一些惊喜：Docker Compose 让那些定义了环境的人员不用担心使用 Docker 工作的细节，解放了用户或者开发人员从而让他们可以专注于容器所含的应用程序。

11.2 环境内的迭代

学习 Compose 如何符合你的工作流程需要丰富的例子，本节将使用一个类似于你在真正的 API 产品所处的那种环境。你将会遇到很多场景并管理很多服务的整个生命周期，其中一个场景将指导你扩展独立的服务，另一个将教你状态管理相关的知识。不过现在不要过分关注环境是如何实现的，下一节将会介绍。

你在本节新遇到的环境是一个用来处理咖啡店元数据的 API，这是“一个热门的新启动的适合当地企业家和自由职业者的头脑风暴”，至少是这个例子的目的。环境结构如图 11-2 所示。

从 GitHub 仓库下载这个示例：

```
git clone https://github.com/dockerinaction/ch11_coffee_api.git
```

当你运行这个命令时，Git 将下载示例的最新副本并放到位于你当前目录下的一个名为

ch11-coffee-api 的新目录。当你一切就绪后，进入那个目录并开始使用环境。

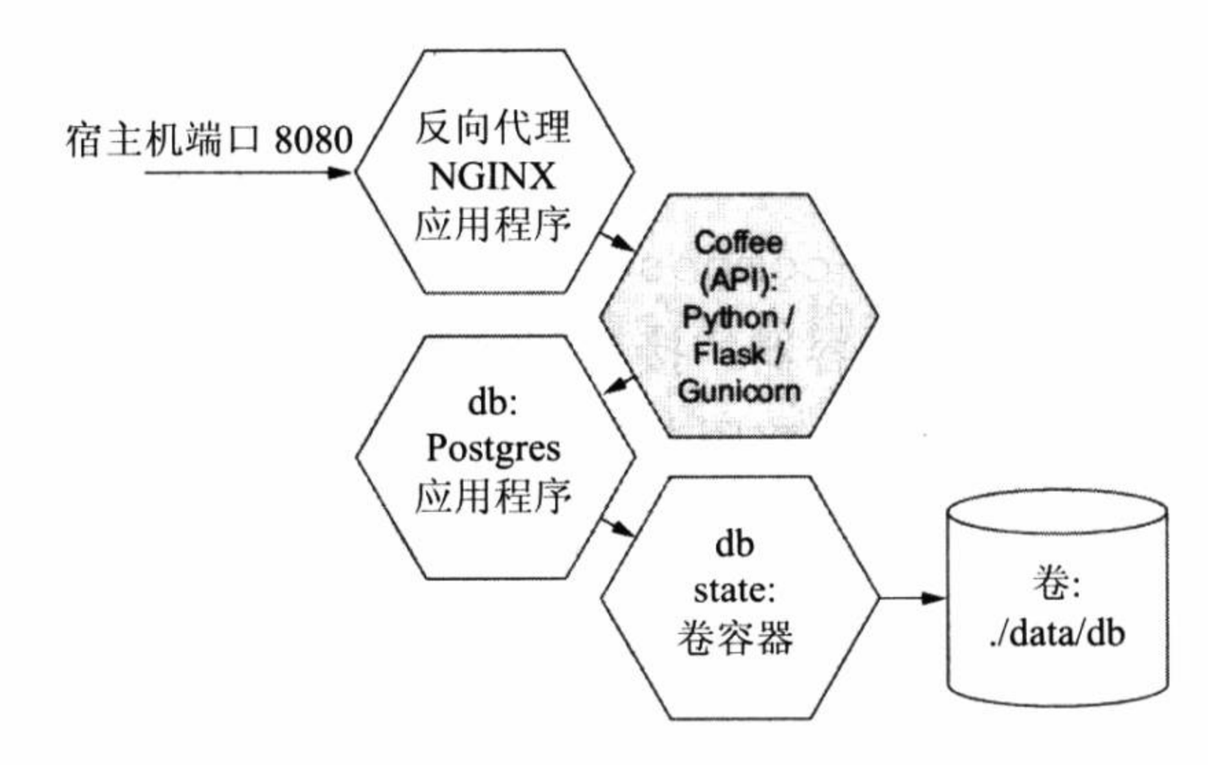


图 11-2 本章示例环境的服务和服务依赖

11.2.1 构建、启动和重新构建服务

借助于从版本控制系统复制源代码和环境描述，通过构建环境中声明的任何组件就可以启动开发流程了，你可以使用下面的命令：

```
docker-compose build
```

build 命令的输出将包含几行，表明特定的服务已经被跳过，因为它们使用了一个镜像。这个环境由四个组件组成，其中只有一个需要构建步骤：Coffee API。你应该从输出中看到当 Compose 构建此服务时，它触发了一个 Dockerfile 构建并创建了一个镜像。构建步骤为引用的服务运行一个 docker build 命令。

Coffee API 的源代码和 Dockerfile 包含在了 coffee 目录里，这是一个简单的基于 Flask 框架的监听端口为 3000 的 Python 应用程序，环境中的其他服务是源自于 Docker Hub 的开箱即用的组件。

随着环境构建，检出生成的已经加载到 Docker 中的镜像，运行 docker images 命令寻找一个名为 ch11coffeeapi_coffee 的镜像。Compose 使用标签和前缀名称来标识某个给定环境创建的镜像和容器。在这种情况下，为 coffee 服务生成的镜像前缀名称是 ch11coffeeapi，因为这是从环境派生的名字，这个名字来自 docker-compose.yml 文件所在的目录。

你已经为 Coffee API 构建了本地组件，但是环境可能需要引用系统中不存在的镜像，可以使用一个命令来拉取那些镜像：

```
docker-compose pull
```

这个命令将为环境中引用的标签拉取最新的镜像，此时此刻，所有必需的组件都应该在你的计算器上准备就绪了，现在你可以启动服务了，使用 db service 命令启动，并特别注意日志：

```
docker-compose up -d db
```

注意在 Compose 启动 db service 之前，先启动了 dbstate 服务。这是由于 Compose 会感知环境中所有已定义的服务，而 db 服务依赖于 dbstate 服务。当 Compose 启动任何特定的服务时，它将启动所有其依赖的服务，这意味着，当你进行迭代时，你只需要启动或重启你的环境的一部分，Compose 将确保它的所有依赖项启动。

现在你已经了解了 Compose 会感知服务依赖关系，然后启动完整的环境：

```
docker-compose up
```

当你使用一个未经限定的 `docker-compose up` 命令时，Compose 将创建或重新创建环境中的每一个服务并启动所有的服务，如果 Compose 检测到有任何还没构建或者使用了缺失镜像的服务，它会触发一个构建或获取合适的镜像（就像 `docker run` 命令）。在这种情况下，你可能已经注意到这个命令重新创建了 db 服务，尽管它已经运行了。这样做是为了确保一切都处于一个正常运转的状态。但如果你知道一个运行正确的某个特定服务的依赖关系，你可以不需要依赖关系就可以启动或重新启动一个服务。为此，需要引入 `flag--no-dep`。

例如，假设你为反向代理服务配置做了一个小调整（包含在 `docker-compose.yml` 文件里），想重新启动代理。你可能只是需要运行如下命令即可：

```
docker-compose up --no-dep -d proxy
```

这个命令将停止并删除任何可能运行的 proxy 容器，然后为 proxy 服务创建并启动一个新的容器。系统中的其他服务将不受影响。如果你省略了 `flag--no-dep`，那么每个服务都会被重新创建并重启，因为 `proxy` 服务要么直接依赖环境中的这些服务，要么是传递依赖的。

当你启动一个其中的组件，需要长时间启动过程的系统并且经历竞争条件时，`flag--no-dep` 就可以派上用场了。在这些情况下，你可能会在启动剩下的服务之前，先启动那些服务让它们完成初始化。

在环境运行时，你可以尝试体验和迭代项目。在 web 浏览器里打开 http://localhost:8080/api/coffeeshops（或使用你的虚拟机的 IP 地址），如果一切正常，你应该看到一个看起来像这样的 JSON 文档：

```
{
  "coffeeshops": []
}
```

这个端点列出了系统中所有的咖啡店，你可以看到列表为空。接下来，在其中添加一些内

容并成为一个你赖以工作的 API 且更熟悉的场景，使用以下 cURL 命令添加内容到你的数据库：

```
curl -H "Content-Type: application/json" \
     -X POST \
     -d '{"name":"Albina Press", "address": " 5012 Southeast Hawthorne
          Boulevard, Portland, OR", "zipcode": 97215, "price": 2,
          "max_seats": 40, "power": true, "wifi": true}' \
     http://localhost:8080/api/coffeeshops/
```

你可能需要用你的虚拟机的 IP 地址替换 localhost，新的咖啡店现在应该在你的数据库里边了。你可以重新在浏览器加载/api/coffeeshops/进行测试，结果应该类似于下面的响应：

```
{
  "coffeeshops": [
    {
      "address": " 5012 Southeast Hawthorne Boulevard, Portland, OR",
      "id": 35,
      "max_seats": 40,
      "name": "Albina Press",
      "power": true,
      "price": 2,
      "wifi": true,
      "zipcode": 97215
    }
  ]
}
```

现在，正如软件开发生命周期中很常见的，你应该在 Coffee API 中添加一个功能，当前的实现只允许你创建和展示咖啡店。从本地负载均衡添加一个基本的 ping 处理程序用于健康检查是非常棒的。在 web 浏览器中打开 http://localhost:8080/api/ping（或使用你的虚拟机的 IP 地址），可以看到当前应用程序的响应。

你要为这个路径添加一个处理程序，并让其返回正在运行 API 的主机名称，用你最喜欢的编辑器打开./coffee/api/api.py，并在文件末尾添加以下的代码：

```
@api.route('/ping')
def ping():
    return os.getenv('HOSTNAME')
```

如果你对本示例的下一步有疑问，或者如果你没有心情编辑文件，你可以从仓库捡出一个功能分支，该分支已经发生了一些更改：

```
git checkout feature-ping
```

一旦你进行了更改并保存文件（或捡出更新的分支），使用以下命令重新构建并创建服务：

```
docker-compose build coffee
docker-compose up -d
```

第一个命令将为 Coffee API 再次运行一个 docker build 命令，生成一个更新后的镜像。第二个命令将会重新创建环境。不必担心你创建的咖啡店数据。用于存储数据库的管理卷将会无缝地分离和附着到新的数据库容器上。当命令完成后，刷新你之前为/api/ping 加载的网页，应该显示一个熟悉风格的 ID，这是运行 Coffee API 的容器的 ID。记住，Docker 会将容器 ID 注入环境变量 HOSTNAME 中。

在本节里，你克隆了一个成熟的项目并用最小的学习曲线开始功能的迭代，下一步，你将会学习如何伸缩、停止和拆除服务。

11.2.2 服务伸缩和删除

Compose 最令人印象深刻并且有用的功能之一是能够支持服务的纵向伸缩，当你这么做时，Compose 创建了提供该服务的容器的多个副本。非常神奇的是这些副本将在你缩小规模时自动清理，但如你所料，当你关闭环境时正在运行的容器将会继续，直到环境被重新构建或者清理。在本节中，你将学习如何扩展、缩小和清理你的服务。

继续 Coffee API 示例，你应该拥有一个运行的环境，可以使用早先介绍的 docker-compose ps 命令来检查。记住，Compose 命令应该在你的 docker-compose.yml 文件所在的目录执行。如果环境没有处于运行状态（proxy、coffee 和 db 服务运行），那么用 docker-compose up -d 命令来运行。

假设你在管理一个测试或生产环境，需要提高 coffee 服务的并发性，要做到这一点，你只需要将你的机器指向目标环境（你将在第 12 章看到）并运行一个命令。在本示例的参数里，你正在使用你的开发环境。在扩展之前，先要获得提供 coffee 服务的容器列表：

```
docker-compose ps coffee
```

输出应该类似如下：

```
        Name                      Command        State            Ports
-------------------------------------------------------------------------
ch11coffeeapi_coffee_1     ./entrypoint.sh      Up      0.0.0.0:32807->3000/tcp
```

注意最右列详细说明了从主机到运行服务的单个容器间的端口映射。你可以使用公共端口（本示例为 32807）直接访问（不需要通过反向代理）由这个容器提供服务的 Coffee API，端口号将随你的计算机不同而不同。如果你加载了容器的 ping 处理程序，你将会看到运行这个服务的容器 ID。现在你已经为你的系统建立了一个基线，使用以下的命令扩展 coffee 服务：

```
docker-compose scale coffee=5
```

该命令将会记录创建的每一个容器的日志，再次使用 docker-compose ps 命令查看所有运行 coffee 服务的容器列表：

```
        Name                    Command        State           Ports
-----------------------------------------------------------------------------
ch11coffeeapi_coffee_1    ./entrypoint.sh     Up     0.0.0.0:32807->3000/tcp
ch11coffeeapi_coffee_2    ./entrypoint.sh     Up     0.0.0.0:32808->3000/tcp
ch11coffeeapi_coffee_3    ./entrypoint.sh     Up     0.0.0.0:32809->3000/tcp
ch11coffeeapi_coffee_4    ./entrypoint.sh     Up     0.0.0.0:32810->3000/tcp
ch11coffeeapi_coffee_5    ./entrypoint.sh     Up     0.0.0.0:32811->3000/tcp
```

正如你所看到的，现在有五个运行 Coffee API 的容器，除了容器 ID 和名称之外，其他都是相同的。这些容器甚至可以使用相同的主机端口映射配置。本示例可以正常工作的原因是 Coffee API 的外部端口 3000 已经被映射到了主机的临时端口（端口号为 0），当你绑定到端口 0 时，操作系统将在一个预定义范围内选择一个可用的端口。相反，如果总是绑定到了主机的 3000 端口，那么同时只能运行一个容器。

在继续下一节之前，先在几个不同的容器（使用容器专用的端口）上测试 ping 处理程序，本书的剩下部分将使用这个示例项目。然而此时除了缩小规模到一个单独的实例之外，没有别的事情可发出类似的命令来缩小：

```
docker-compose scale coffee=1
```

注意这里的“1”

该命令的日志指出了正在停止和删除哪些实例，再次使用 docker-compose ps 命令来验证你的环境的状态：

```
        Name                    Command        State           Ports
-----------------------------------------------------------------------------
ch11coffeeapi_coffee_1    ./entrypoint.sh     Up     0.0.0.0:32807->3000/tcp
```

在继续学习持久化状态之前，需要用 docker-compose rm 命令清理环境，这样你就可以重新开始一个崭新的环境。

11.2.3 迭代和持久化状态

你已经学会了使用 Compose 进行基本的环境状态管理，在上一节的结尾，你停止和删除了所有的服务以及任何的管理卷。在这之前，你也可以使用 Compose 来重新创建环境，有效地删除和重新构建所有的容器。本节重点是工作流程和边界情况的细微差别，这些边界情况有一些不受欢迎的影响。

首先，注意管理卷，卷是状态管理的主要问题。幸运的是，Compose 使得在迭代环境

中管理卷（如图 11-3 所示）的工作变得微不足道了。当服务重新构建时，附加的管理卷不会被删除。相反，它们重新附加到了那个服务更换后的容器上。这意味着你可以自由地迭代而不用担心丢失数据。当最后一个容器使用 docker-composerm 命令和 flag-v 删除时，管理卷最后也会被清理了。

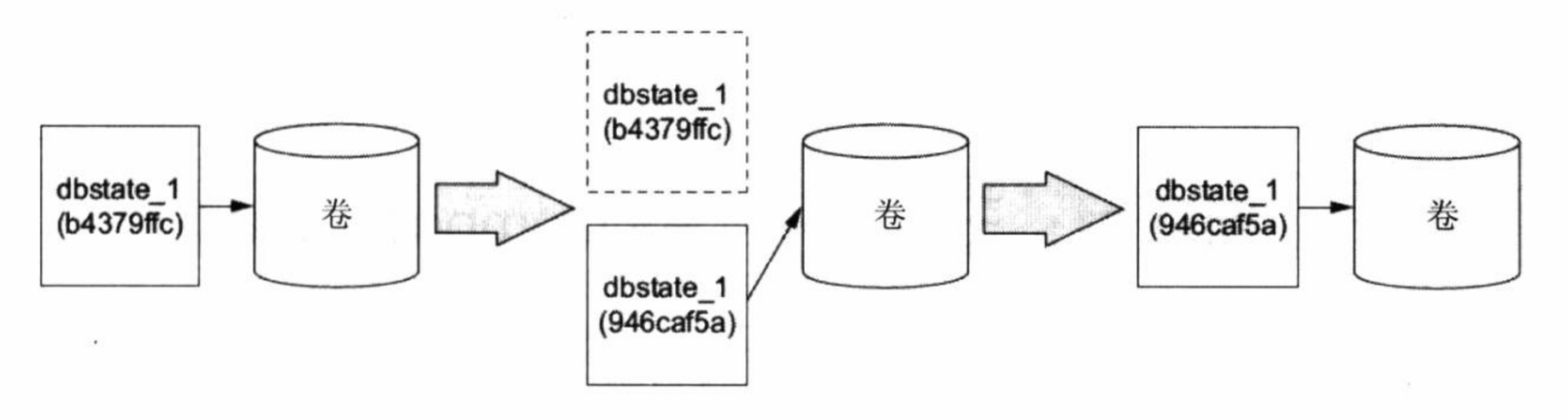


图 11-3 一个卷容器在重新创建后将会有同样一个重新附着的管理卷

与状态管理和 Compose 相关的最大的问题还是环境状态，在高度迭代的环境中你会变更一些东西，包括环境配置，某些类型的变更会带来一些问题。

例如，如果你在 docker-compose.yml 文件中重命名或删除一个服务定义，那么你就失去了使用 Compose 管理这个服务的能力。回到 Coffee API 示例中，coffee 服务在开发时被命名为 api。环境处于一种不断变化的状态中，在某些时候 api 服务处于运行中，该服务被重命名为 coffee。当这种情况发生时，Compose 将不再感知到 api 服务。重新构建并重启后，新的 coffee 服务将会工作，而 api 服务被孤立。

当你使用 docker ps 命令来列出正在运行的容器时你会发现这种状态，并且会注意到服务的旧版本容器仍然在运行。恢复很简单，你可以直接使用 docker 命令清理环境或者回到 docker-compose.yml 文件中添加孤立的服务定义，用 Compose 清理。

11.2.4 网络和连接问题

使用 Compose 来管理服务系统的最后一件需要注意的事情是记住容器连接局限性的影响。

在 Coffee API 示例项目中，proxy 服务对于 coffee 服务有一个连接的依赖。需要记住的是，Docker 构建的镜像是通过创建防火墙规则和注入服务发现信息到所依赖的容器的环境变量和/etc/hosts 文件中来建立连接关系的。

在高度迭代的环境中，用户可能只要重启特定的服务，如果另外一个服务依赖于它的话，这可能会导致一些问题。举例来说，如果你启动了 Coffee API 环境，然后选择性地重启 coffee 服务，proxy 服务将不再能够追溯到它的上游依赖。当容器重新创建或者重新

启动后，它们返回的是不同的 IP 地址。这一变化使得注入 proxy 服务的信息失效了。

这有时看起来烦琐，但是在没有动态服务发现机制的环境中处理这个问题最好的方法是重启整个环境，至少目标服务不会作为上游依赖。在使用动态服务发现机制或者 Overlay 网络的健壮系统中这不是一个问题，多主机网络将在 12 章简要讨论。

到目前为止，你已经在一个现有的项目上下文中使用 Compose，从零开始，你还有更多需要考虑的事情。

11.3 开始一个新项目：三个示例中的 Compose YAML

定义一个环境并不是一件轻松的事情，需要洞察力和深谋远虑。因为项目需求、流量整形、技术、财务约束和本地专业知识的变化都会成为你的项目环境需要考虑的因素。因此，保持环境和项目之间的关注点清晰的分离是至关重要的。未能这样做常常意味着迭代你的环境还需要迭代其中运行的代码。本节演示了 Compse YAML 功能是如何帮助你构建你需要的环境的。

本节的其余部分将研究 docker-compose.yml 文件中的部分内容，包括 Coffee API 示例，文本中包含相关的摘录。

11.3.1 启动前的构建、环境、元数据和网络

首先检查 coffee 服务，该服务使用了 Compose 管理构建、环境变量注入、链接依赖和特殊的网络配置，coffee 服务定义如下：

```
coffee:
  build: ./coffee
  user: 777:777
  restart: always
  expose:
    - 3000
  ports:
    - "0:3000"
  links:
    - db:db
  environment:
    - COFFEEFINDER_DB_URI=postgresql://postgres:development@db:5432/po...
    - COFFEEFINDER_CONFIG=development
    - SERVICE_NAME=coffee
  labels:
    com.dockerinaction.chapter: "11"
    com.dockerinaction.example: "Coffee API"
    com.dockerinaction.role: "Application Logic"
```

❶ 从位于./coffee 的 Dockerfile 构建

❷ 为使用数据库设置环境变量

❸ 为服务打上标签

❹ 公开和映射容器端口

当你有一个与特定镜像源代码密切相关的环境时，你可能想要使用 Compose 来自动化那些服务的构建阶段。在 Coffee API 示例项目中，对于 `coffee` 服务就是这么做的，不过使用案例超出了典型的开发环境的需求。

如果你的环境使用 data-packed 卷容器来注入环境配置，你可能会考虑为每个环境使用 Compose 来管理构建阶段。不管什么原因，这些都可以用一个简单的 YAML 键和结构来表达，参见前文的 Compose 文件中的❶。

`build` 键的值是用于构建的 Dockerfile 文件所在位置的目录，你可以使用从 YAML 文件位置开始的相对路径，你还可以使用 `dockerfile` 键来提供一个备选的 Dockerfile 文件的名称。

基于 Python 的应用程序需要设置一些环境变量来与数据库集成，通过 environment 键可以为一个服务设置环境变量，`environment` 键的值为内嵌列表或者字典，在❷中使用的是列表方式。

或者你可以提供一个或多个包含用 `env_file` 作为键的环境变量定义的文件，与环境变量相似，容器元数据可以设置为以 `labels` 为键，以内嵌列表为或者字典表为值，❸中使用的是字典形式。

使用详细的元数据使得用你的镜像和容器处理工作容易多了，但这仍然是一个可选的实践。Compse 使用标签为服务核算存储元数据。

最后，❹处显示了该服务在此外通过公开一个端口定制化了网络，绑定到一个主机端口，并声明一个链接依赖关系。

`expose` 键接受容器端口的一个列表，这些端口应该根据防火墙规则公开。`ports` 键按照与 `docker run` 命令中`-p` 选项一样的格式接受描述了端口映射关系的字符串列表。`links` 命令按照与 `docker run` 命令中 `flag --link` 一样的格式接受和定义了链接的列表。在阅读第 5 章后我们应该熟悉如何使用这些选项。

11.3.2 已知的组件和绑定挂载卷

Coffee API 示例的两个关键组件可以从 Docker Hub 下载镜像获得。一个是 `proxy` 服务，它使用一个官方 Nginx 库；另一个是 db 服务，使用官方 Postgres 库。官方仓库是合理可信的，不过在部署到敏感的环境中之前，拉取和检查第三方镜像是一个最佳实践。一旦你信任一个镜像，你应该使用内容可寻址的镜像，以确保没有部署不受信任的组件。

服务可以从任何带有 `image` 键的镜像启动，`proxy` 和 `db` 服务都是基于镜像，并且使用的都是内容可寻址的镜像：

```
db:
  image: postgres@sha256:66ba100bc635be17...
  volumes_from:
    - dbstate
  environment:
    - PGDATA=/var/lib/postgresql/data/pgdata
    - POSTGRES_PASSWORD=development
  labels:
    com.dockerinaction.chapter: "11"
    com.dockerinaction.example: "Coffee API"
    com.dockerinaction.role: "Database"

proxy:
  image: nginx@sha256:a2b8bef333864317...
  restart: always
  volumes:
    - ./proxy/app.conf:/etc/nginx/conf.d/app.conf
  ports:
    - "8080:8080"
  links:
    - coffee
  labels:
    com.dockerinaction.chapter: "11"
    com.dockerinaction.example: "Coffee API"
    com.dockerinaction.role: "Load Balancer"
```

为授信的Postgres版本使用内容可寻址镜像

使用数据容器模式

为授信的 NGINX 版本使用内容可寻址镜像

通过卷注入配置

Coffee API 项目使用一个数据库和负载平衡器，只需要最小配置即可，并且以卷的形式提供配置。

proxy 服务使用一个卷来绑定挂载一个本地的配置文件到 Nginx 的动态配置位置，这是一个简单的注入配置而不用构建一个完整的新镜像。

db 服务使用 volumesfrom 键来列出那些定义了必需卷的服务，在本示例中，db 在 dbstate 服务上声明了一个依赖关系，dbsate 服务是一个卷容器服务。

一般来说，YAML 中的键是与 docker run 命令已公开的功能密切相关，你可以在 https://docs.docker.com/compose/yml/里找到完整的参考。

11.3.3 卷容器和扩展服务

偶尔你会遇到一个通用的服务原型，示例可能包括一个 Node.js 服务、Java 服务、基于 Nginx 的负载均衡器或者一个卷容器。在这些情况下，可以表明这些原型作为父服务是合适的，并为特定的实例扩展和专门化。

Coffee API 项目定义了一个名为 data 的卷容器原型。原型像其他任何一种服务一样也是一种服务，在这种情况下，原型指定了一个要启动的镜像、一个要运行的命令、一个要运行的 UID 和标签元数据：

```
data:
  image: gliderlabs/alpine
  command: echo Data Container
  user: 999:999
  labels:
    com.dockerinaction.chapter: "11"
    com.dockerinaction.example: "Coffee API"
    com.dockerinaction.role: "Volume Container"
```

单独的服务除了定义卷容器的合理默认值外，其他什么也没有做。注意，它不定义任何卷，专门化工作是留给每个扩展自原型的卷容器的：

```
dbstate:
  extends:
    file: docker-compose.yml
    service: data
  volumes:
    - /var/lib/postgresql/data/pgdata
```

引用到另外一个文件中的父服务

dbstate 服务定义了一个扩展自 data 服务的卷容器，服务扩展必须指定文件和被扩展的服务名称，相关的键是 extends 以及内嵌的 file 和 service。服务扩展工作类似于 Dockerfile 构建的方式。首先构建原型容器，然后提交。子容器是一个由新生成的层构建的新容器。就像 Dockerfile 构建，这些子容器继承了父容器所有的属性，包括元数据。

dbstate 服务使用 volumes 键定义了挂载在/var/lib/postgresql/data/pgdata 的管理卷，volumes 键接受一系列由 flag docker run -v 所允许的卷规范，参见第 4 章了解卷类型、卷容器以及卷的细微差别。

Docker Compose 对于那些使用 Docker 作为他们基础设施核心组件的人员来说是一个关键的工具，有了它，你就可以减少迭代时间、版本控制环境以及与声明式文件交互的特定服务的编排工作。第 12 章将基于 Docker Compose 构建使用案例，并引入 Docker Machine 来帮助鉴定和自动化测试。

11.4 小结

本章聚焦于一个名为 Docker Compose 的 Docker 客户端的辅助工具，其提供了缓解单调沉闷的容器命令行管理工作的功能，本章包括以下几个方面：

- Docker Compose 是一个用于定义、启动和管理服务的工具，其中的服务被定义为 Docker 容器的一个或多个副本。
- Compose 使用 YAML 配置文件来提供环境定义。
- 使用 docker-compose 命令行程序，你可以构建镜像、启动和管理服务、扩展服

务并在任何运行 Docker Daemon 的主机上查看日志。

- 在项目中管理环境和迭代的 Compose 命令，类似于 `docker` 命令行。构建、启动、停止、删除和列出服务，所有这些都相当于单个容器关注的那些。
- 有了 Docker Compose，你可以通过运行一个 up 和 down 命令来伸缩容器数目。
- 借助于 YAML 格式的声明式环境配置可以启用环境版本控制、共享、迭代和一致性。

第 12 章　Docker Machine 和 Swarm 集群

本章介绍

- 创建虚拟机，以 Docker Machine 运行 Docker
- 集成并管理远程 Docker Daemon
- Docker Swarm 集群介绍
- 借助于 Docker Machine 提供完整的 Swarm 集群
- 在集群中管理容器
- 有关容器调度和服务发现的 Swarm 解决方案

本书的大部分内容是有关在一台计算器上的 Docker 如何进行互动的，而在实际项目中，你将会在同一时间使用多台机器。Docker 是一个很好的创建大规模服务系统的构建工具。在这样的环境中，你会遇到各种各样的新问题，它们不是可以直接由 Docker 引擎解决的。

用户如何启动一个不同的服务运行在不同的主机上的环境？服务将如何在这样一个分布式环境中定位服务依赖关系？用户如何以供应商无关的方式快速创建和管理大型成组的 Docker 主机？服务应该如何针对可用性和失效转移进行伸缩？随着服务扩展到了一组主机，系统将如何知道从负载均衡器管理流量？

容器提供的隔离的好处在于可以对于给定的计算器进行本地化，不过随着最初将容器比喻为集装箱，容器抽象使各种工具的实现成为可能。第 11 章谈到了 Docker Compose，一个定义服务和环境的工具。在本章中，你将会遇到 Docker Machine 和 Docker Swarm。这些

工具可以解决 Docker 用户在供应机器、编排部署和运行集群服务软件时遇到的各种问题。

Docker Engine 和 Docker Compose 通过从包含的软件中抽象主机这个概念让开发人员和操作人员的工作变得更简单，Docker Machine 和 Docker Swarm 帮助系统管理员和基础设施工程师将这些抽象扩展到集群环境里。

12.1 介绍 Docker Machine

学习和解决分布式系统问题的第一步是构建一个分布式系统，Docker Machine 可以在几秒钟内创建和移除启用了 Docker 的主机集群。学习如何使用这个工具对于那些想要了解如何在分布式云环境或者本地虚拟环境使用 Docker 的人是必不可少的。

驱动的选择

Docker Machine 附带了很多开箱即用的驱动，每个驱动程序都用不同的虚拟机技术或者基于云的虚拟计算供应商来与 Docker Machine 集成。每个云平台都有它的优点和缺点。从 Docker 客户端的角度来看，本地主机和远程主机是没有区别的。

使用本地虚拟机驱动（如 VirtualBox）将减低运行本章中的示例的成本，但是你应该考虑为你首选的云供应商选择一个驱动，驱动有强大的能力，可以知道你发出的命令实际上在管理真实世界的资源，并且你部署的将在互联网上运行。此时此刻，你离构建一个真实的生产环境只有少数特定领域的步骤了。

如果你决定为这些示例使用一个云提供商，你需要使用供应商特定的信息（比如访问密钥和安全密钥）来配置你的环境，这样可以在本章的任何命令中替换那些使用特定驱动的标识了。

你可以通过运行 docker-machine help create 命令或者咨询在线文档找到与特定驱动 flag 相关的详细信息。

12.1.1 构建和管理 Docker Machine

在 `docker` 命令行和 Compose 之间，将介绍几个命令，本节介绍了 `docker-machine` 命令行命令。因为这些工具在形式和功能上都如此相似，本节将通过一小组的示例作介绍，如果你想了解关于 `docker-machine` 命令行的更多信息，你可以随时使用 `help` 命令。

```
docker-machine help
```

第一个也是最重要的是要知道 Docker Machine 是如何创建一个 Docker 主机的。接下来的三个命令将创建三个使用 VirtualBox 驱动的主机，每个命令都会在你的计算机上创建一个新的虚拟机：

```
docker-machine create --driver virtualbox host1
docker-machine create --driver virtualbox host2
docker-machine create --driver virtualbox host3
```

在你运行这三个命令后（可能需要花几分钟时间），你会有三个由 Docker Machine 管理的 Docker 主机，Docker Machine 在你的 home 目录（位于~/.docker/machine/）通过一组文件来追踪这些主机。这些文件描述了你创建的主机，用于建立与主机安全通信的证书颁发机构以及用于基于 virtual Box 的主机的磁盘镜像。

Docker Machine 可以用来列出、检查和升级你的集群机器，使用 `ls` 子命令来获得被管理的机器列表：

```
docker-machine ls
```

该命令将显示的结果类似如下：

```
NAME     ACTIVE   DRIVER       STATE     URL                         SWARM
host1             virtualbox   Running   tcp://192.168.99.100:2376
host2             virtualbox   Running   tcp://192.168.99.101:2376
host3             virtualbox   Running   tcp://192.168.99.102:2376
```

这个命令将列出每台 Docker 机器的名称、赖以创建的驱动、状态以及 Docker Daemon 可以被访问的 URL。如果你使用 Docker Machine 在本地运行 Docker，在列表里会有另一个条目，并且这个条目标记为 `active`。活跃的机器（在活跃列下面用星号表示）是当前配置为与你的环境进行通信的机器，任何用 `docker` 或者 `docker-compose` 命令行接口发布的命令都将会连接到活跃机器上的 Docker Daemon。

如果你想知道更多的关于一台特定的机器或查找它的配置的特定一部分，你可以使用 `inspect` 子命令：

```
docker-machine inspect host1
docker-machine inspect --format "{{.Driver.IPAddress}}" host1
```

JSON 文档描述了 machine

这就是 IP 地址

`docker-machine` 的 `inspect` 子命令非常类似于 `docker` 的 `inspect` 命令，你甚至可以使用相同的 Go 模板语法（http://golang.org/pkg/text/template/）来转换原生的描述这些机器的 JSON 文档。这个例子使用一个模板来检索计算器的 IP 地址。如果你需要在实践中使用，请使用 `ip` 子命令：

```
docker-machine ip host1
```

Docker Machine 可以相对轻松地构建一个集群，并且重要的是你可以同样轻松地维护这个集群。任何被管理的机器都可以用 upgrade 子命令升级：

```
docker-machine upgrade host3
```

这个命令的输出如下：

```
Stopping machine to do the upgrade...
Upgrading machine host3...
Downloading ...
Starting machine back up...
Starting VM...
```

在升级过程中机器将被关闭，下载一个软件更新后的版本并重启机器。使用这个命令，你可以用最小的代价对 Docker 队列执行滚动升级。

你偶尔会需要在其中一台机器上操作文件或直接访问一台机器的终端，可能是你需要获取或者准备绑定挂载卷的内容，有时你可能需要从主机来测试网络或者定制化主机配置。在这些情况下，你可以使用 ssh 和 scp 子命令。

当你用 Docker Machine 创建或注册一台机器，它会创建或导入一个 SSH 私有密钥文件，用来在一台机器上通过 SSH 协议以一个特权用户的身份来进行认证。docker-machine ssh 命令将与目标机器进行身份认证并绑定你的终端到该机器的 shell 上。

例如，如果你想要在一台名为 host1 的机器上创建一个文件，可以发出以下命令：

```
docker-machine ssh host1
touch dog.file
exit
```

绑定你的终端到 host 1 的 shell 上

退出远程 shell 并终止命令

使用一个完全绑定的终端来运行一个命令似乎有点没头没脑，如果你不需要一个完全交互式的终端，你可以指定传入额外的参数来运行 ssh 命令。运行以下命令来写入一只狗的名字到你刚刚创建的文件：

```
docker-machine ssh host1 "echo spot > dog.file"
```

如果你在一台机器上有需要复制到其他地方的文件，你可以使用 scp 子命令，这样操作比较安全。该命令有两个参数：一为源主机和文件，二为目标主机和文件。自己尝试一下，将你刚才创建的文件从 host1 复制到 host2，然后使用 ssh 子命令来查看 host2：

```
docker-machine scp host1:dog.file host2:dog.file
docker-machine ssh host2 "cat dog.file"
```

输出：spot

与 SSH 相关的命令是定制化配置的关键，获取卷的内容并执行其他与主机相关的维护工作。其余的你需要构建和维护队列的命令是可预测的。

用于启动、停止（或杀死）和删除机器的命令就像用于容器的那些命令，docker-machine 命令提供了四个子命令：start、stop、kill 和 rm：

```
docker-machine stop host2
docker-machine kill host3
docker-machine start host2
docker-machine rm host1 host2 host3
```

该节涵盖了使用 Docker Machine 进行构建和维护集群的大部分的基本机制，下一节将演示如何使用 Docker Machine 配置你的客户端环境以便使用这些机器以及如何直接访问机器。

12.1.2 配置 Docker 客户端与远程 Daemon 工作

Docker Machine 可以追踪其管理的机器的状态，你可以使用 Docker Machine 来升级远程主机上的 Docker，打开 SSH 连接并在主机间安全地复制文件。但是 Docker 客户端比如 docker 命令行接口或者 docker-compose 都被设计为每次只能连接单个 Docker 主机。出于这个原因，Docker Machine 的一个最重要的功能是为一台活跃的 Docker 主机生成环境配置。

Docker Machine、Docker 客户端以及环境之间的关系如图 12-1 所示。

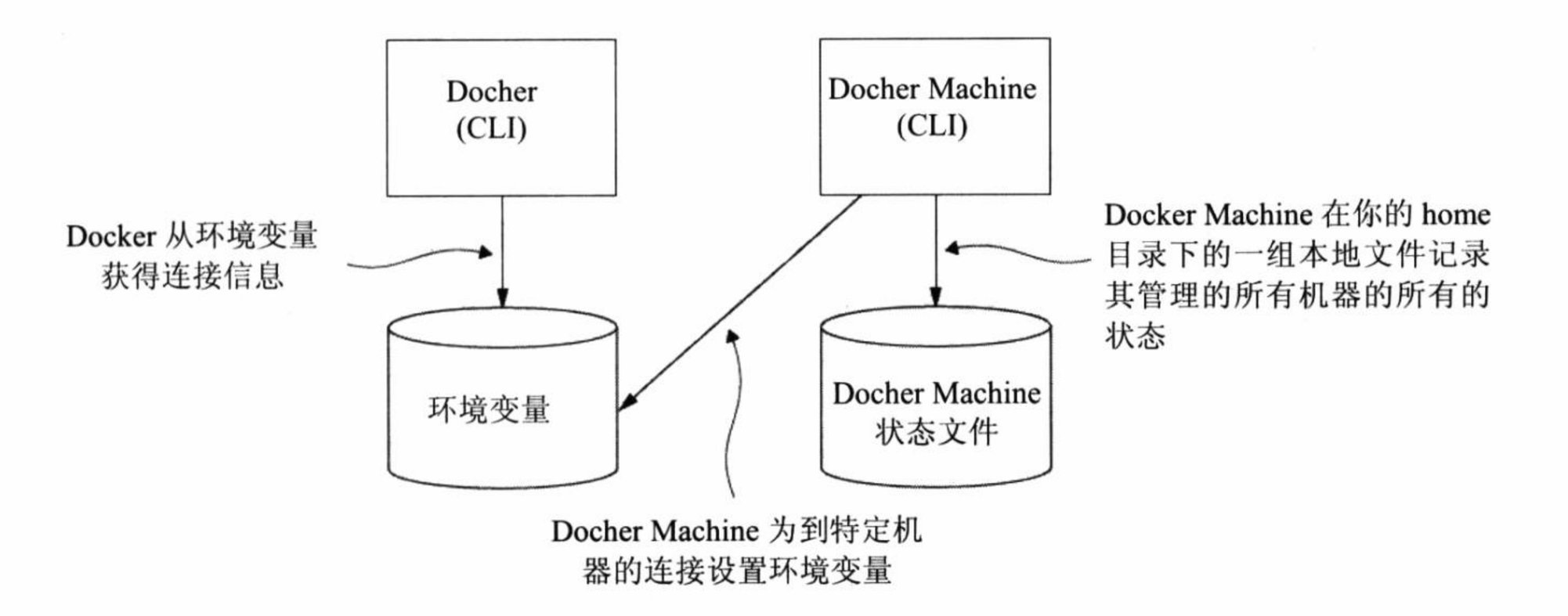


图 12-1 docker、docker-machine 和相关的状态资源之间的关系

通过创建数台新机器并激活其中一台来开始学习如何管理你的 Docker 环境，首先运行 create 命令：

```
docker-machine create --driver virtualbox machine1
docker-machine create --driver virtualbox machine2
```

为了激活这台新机器，你必须更新环境。Docker Machine 包含了一个名为 env 的子命令，该子命令试图自动检测用户的 shell 并打印命令来配置环境连接到一个特定的机器。如

果不能自动检测用户的 shell，你可以使用 flag--shell 来设置特定的 shell：

```
docker-machine env machine1
docker-machine env --shell powershell machine1
docker-machine env --shell cmd machine1
docker-machine env --shell fish machine1
docker-machine env --shell bash machine1
```

让 env 自动检测你的 shell

获得 PowerShell 配置

获得 CMD 配置

获得 fishll 配置

获得默认(POSIX)配置

这些命令将打印特定 Shell 的命令的列表，这些命令运行时需要带着如何调用自动执行 docker-machine 命令的注释。例如设置 machine1 作为活跃的机器，你可以在 POSIX Shell 中执行 docker-machine env 命令：

```
eval "$(docker-machine env machine1)"
```

如果你使用 Windows 并运行 PowerShell 运行，你可以运行一个如下的命令：

```
docker-machine env --shell=powershell machine1 | Invoke-Expression
```

你可以运行 active 子命令验证你已经激活了 machine1，或者你可以从 ls 子命令的输出中检查 ACTIVE 列：

```
docker-machine active
docker-machine ls
```

任何观察你本地机器环境配置的客户端将使用 Docker 远程 API，这些 API 是为在特定 URL 活跃机器提供的，当活跃的机器发生更改时，任何 Docker 客户端命令的执行目标也都会被更改。这种环境下的状态如图 12-2 所示。

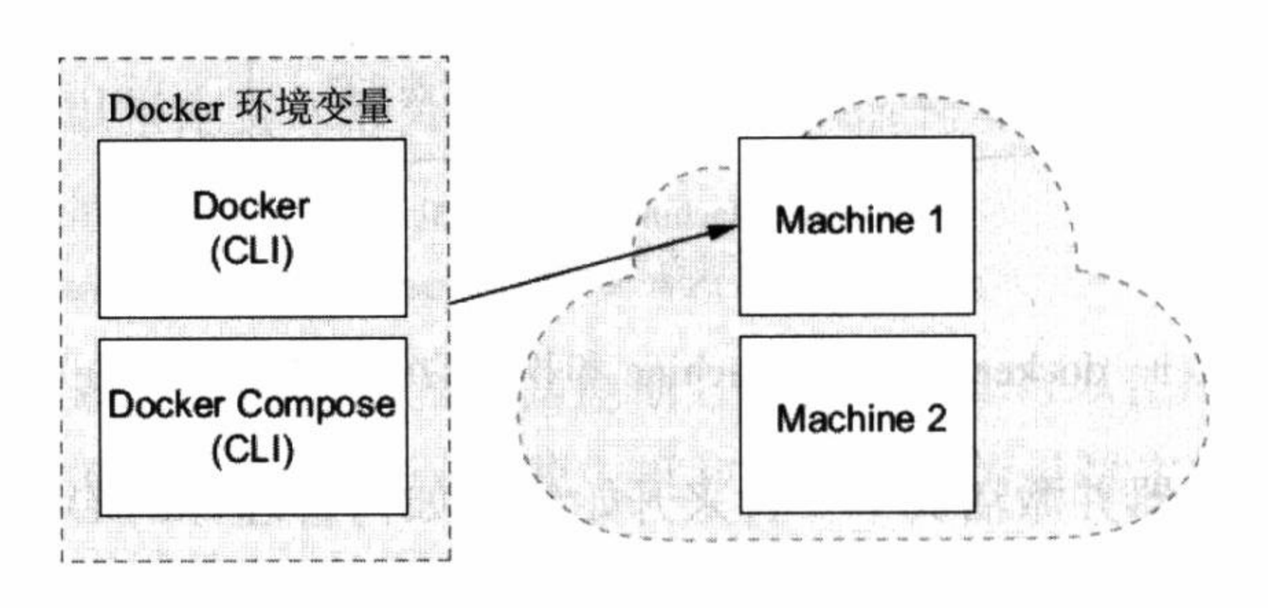


图 12-2 Docker Machine 创建的两台机器中的一台在本地环境中被激活了，Docker 客户端将使用该机器提供的 Docker API

创建一些容器并亲自体验多台机器如何工作是多么简单和微妙，可以从拉取一个镜像

到活跃机器开始：

```
docker pull dockerinaction/ch12_painted
```

拉取到活跃机器的镜像将只会被拉取到该机器上，这意味着如果你在集群中使用一些常见的镜像，你需要将镜像拉取到每一台机器上。如果最小化容器启动时间是重要的，那么理解这一点就很重要。在那些情况下，你会想在容器创建时间点以前尽可能多地并行拉取这些镜像。在你用这个镜像启动容器之前，将活跃机器从 machine1 变到 machine2，并列出这些镜像：

```
eval "$(docker-machine env machine2)"
docker images
```

用适合你的 shell 的等效命令取代

images 子命令的输出应该是空的，这个示例有助于说明独立地拉取镜像到每一台机器上的必要性，这台机器——machine2，从来没有安装任何镜像。

接下来，在 machine2 机器上拉取镜像并运行容器 dockerinaction/ch12*painted*：

```
docker run -t dockerinaction/ch12_painted \
    Tiny turtles tenderly touch tough turnips.
```

现在比较 machine1 和 machine2 上的容器列表：

```
docker ps -a
eval "$(docker-machine env machine1)"
docker ps -a
```

用适合你的 shell 的等效命令取代

列表再一次为空，因为 machine1 机器没有创建容器，这个简单的示例有助于说明多台机器工作是多么容易，它还应该说明手动连接、编排和调度相当大规模的机器集群是多么令人困惑。除非你明确地查询活动主机，不然随着使用数量的增加，会很难追踪你的 Docker 客户端是在哪里被管理的。

利用 Docker Compose 可以简化编排，但是 Compose 就像任何其他 Docker 客户端一样只能使用你的环境配置使用的 Docker Daemon。如果你想用 Compose 启动一个环境，在当前的配置中，machine1 是活跃机器，那么将会在 machine1 创建环境描述的所有服务。在继续之前，请删除 machine1 和 machine2 来清理环境：

```
docker-machine rm machine1 machine2
```

Docker Machine 是一个构建和管理基于 Docker 的集群机器的很棒的工具。Docker Compose 为基于容器的服务提供了编排，不过依然存在的主要问题是跨 Docker Machine 集群调度容器，但后来发现那些服务其实已经被部署在某些地方了。Docker Swarm 将会解决这个问题。

12.2 Docker Swarm 介绍

除非你曾经在一个分布式系统环境中工作过或者构建过动态部署拓扑，Docker Swarm 解决的那些问题都需要花一些功夫来理解。本节将深入这些并解释 Swarm 是如何从高层视角来解决每一个问题的。

当人们遇到的第一个问题时，他们可能会扪心自问："我应该选择哪台机器来运行给定的容器？"组织你需要的容器来跨集群机器运行并不是一个轻松的任务。以前我们会将不同的软件部署到不同的机器上。使用某台机器作为部署单元使得自动化会比用现有的工具考虑实现更加简单。当某台机器是部署单元时，要找出是哪台机器运行一个给定程序不是你需要回答的问题，答案总是"一个新的"。现在，随着 Linux 容器用于隔离和 Docker 用于容器工具化，剩下的主要问题就是资源的使用效率、每台机器硬件的性能特征和网络位置。基于这些因素选择一台机器就被称之为调度。

在有人找出第一个问题的解决方案后，他们会立刻问："现在我的服务已经部署在了网络上，其他服务如何找到它？"当你给自动化过程委托调度时，你不能事先知道服务部署在哪里。如果你不知道服务位于哪里，其他服务如何使用它呢？传统上服务端软件使用 DNS 来解决已知名字的服务在一组网络中的位置问题。DNS 提供一个适当的 lookup 接口，但写入数据是另一个问题。在一个特定位置推广服务的可用性称为注册，而解决已命名服务的位置问题被称为服务发现。

在本章你将学习借助于 Docker Machine 构建的一个 Swarm 集群是如何解决这些问题的，探索调度算法并部署 Coffee API 示例。

12.2.1 借助于 Docker Machine 构建 Swarm 集群

Swarm 集群是由两种类型的机器组成的，以管理模式运行 Swarm 的机器称为 manager，而运行 Swarm 代理的机器称为 node。

在其他方面，Swarm manager 和 node 就像任何其他的 Docker 机器，这些程序不需要特殊的安装或访问机器的权限。它们在 Docker 容器里边运行，如图 12-3 所示了一个典型的 Swarm manager 机器的计算栈，在其中运行了 manager 程序、Swarm 代理和另外的容器。

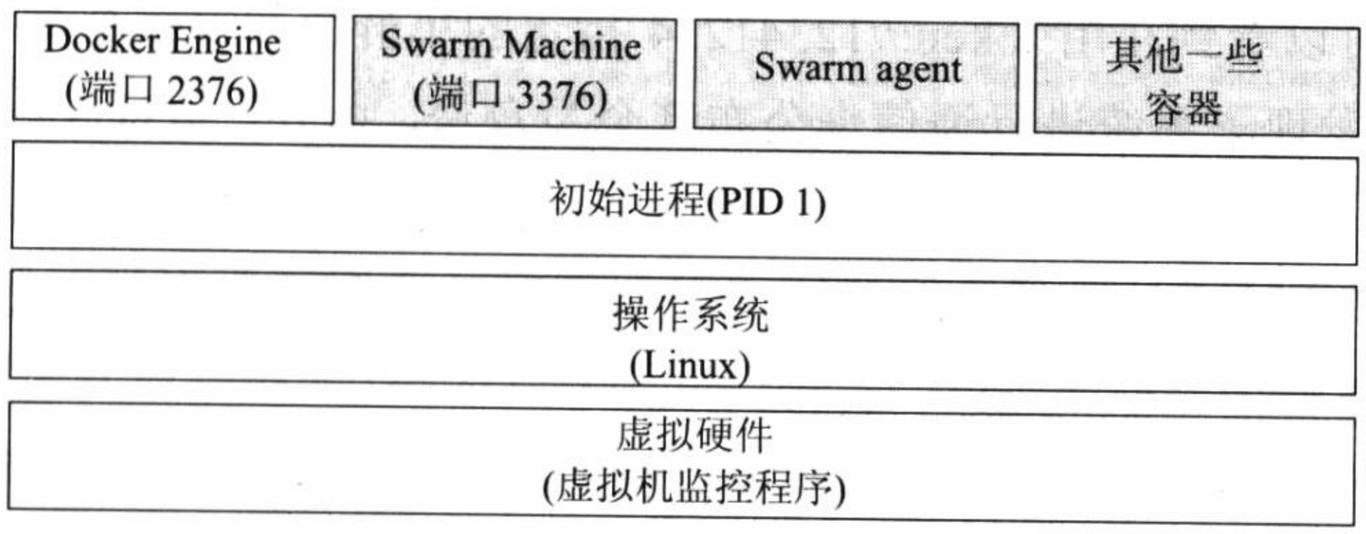


图 12-3 Swarm、Swarm manager 和另外的程序，这些程序都是在一个典型的虚拟机里的 Docker 引擎的 Docker 容器里运行的

Docker Machine 可以生成 Swarm 集群，就像操作独立的 Docker Machine 一样容易，唯一的区别是有一小组额外的命令行参数，这些参数包含在了 `create` 子命令里。

第一个是`--swarm`，表明正在创建的机器应该运行 Swarm 代理软件并加入一个集群。第二个是使用参数`--swarm-master`，将会指示 Docker Machine 配置新的机器作为 Swarm manager。第三个是 Swarm 集群的每一种机器类型都需要一种定位和识别其加入或者管理的集群的方法，`--swarm-discovery` 以一个额外的参数指定集群的唯一标识符。如图 12-4 所示了一个小的 machine 集群和一个单独的机器的对比。

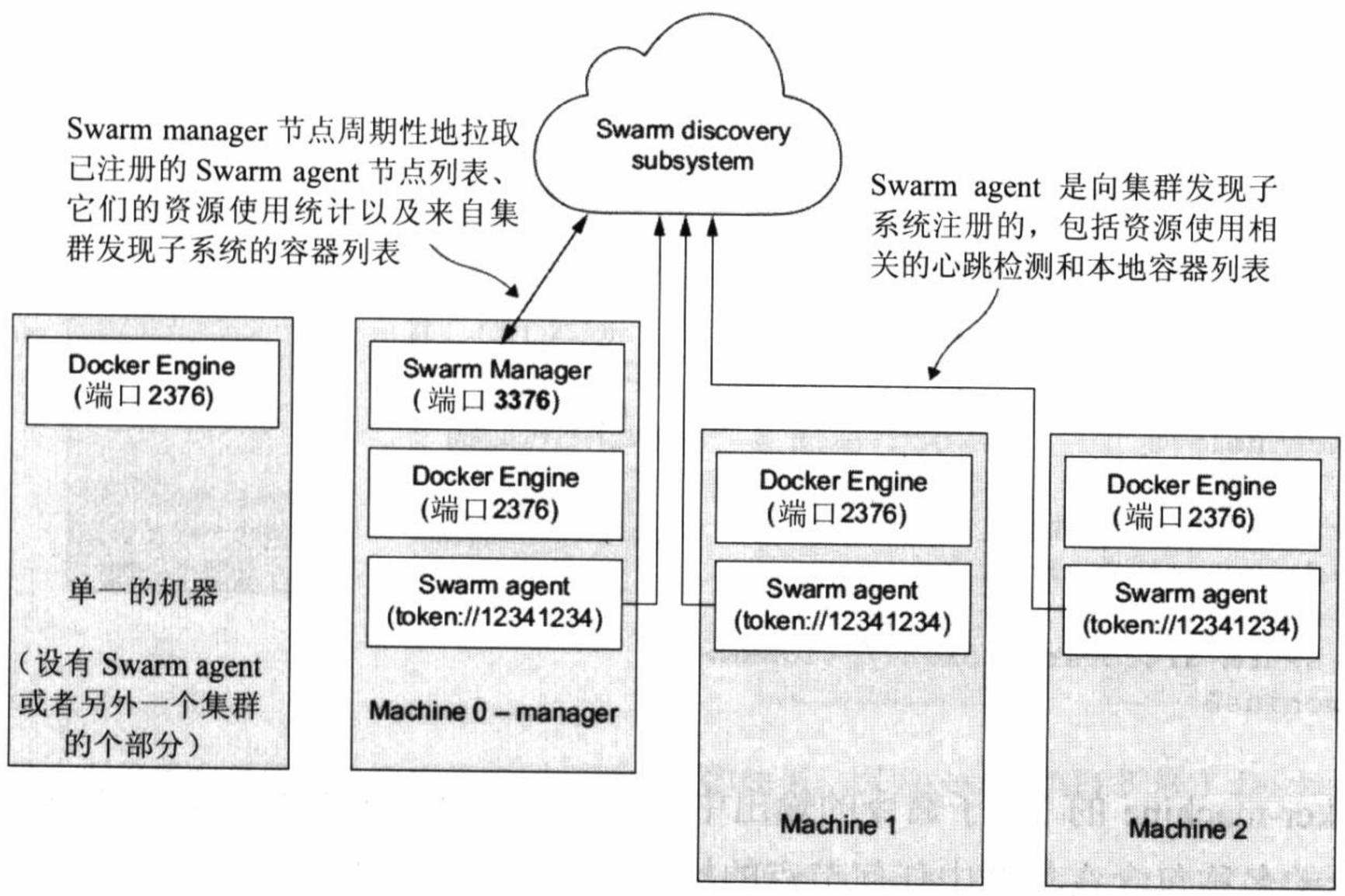


图 12-4 Swarm manager 和代理通过集群发现子系统交互

在这幅图上可以看到在用 token://12341234 指定的集群内部每一个节点上的 Swarm 代理可以与 Swarm 发现子系统进行通信来公布其会员身份，此外，运行 Swarm manager 的单台机器可以为集群中更新后的节点列表投票给 Swarm 发现子系统。在深入了解 Swarm 发现子系统和其他机制之前，我们会使用 Docker Machine 来创建一个你自己的 Swarm 集群。接下来的几个命令将指导你完成这一过程。

创建你自己的 Swarm 集群的第一步是创建一个集群标识符，类似于大多数的扩展自 Docker 的子系统，Swarm 发现子系统可以通过改变来适应你的环境。在默认情况下，Swarm 使用 Docker Hub 提供的一个免费托管解决方案，运行以下命令创建一个新的集群标识符：

创建一个新的本地 Docher

```
docker-machine create --driver virtualbox local
eval "$(docker-machine env local)"

docker run --rm swarm create
```

用适合你的 shell 的等效命令取代

最后一个命令应该输出一个十六进制的标识符，看起来像这样：

```
b26688613694dbc9680cd149d389e279
```

在接下来的三个命令中复制结果值并替换<TOKEN>，下面的命令将在你的计算机上用虚拟机创建一个有三个节点的 Swarm 集群。注意，第一个命令使用--swarm-master 参数，表明这台正在创建的 macher 应该管理新的 Swarm 集群：

```
docker-machine create \
    --driver virtualbox \
    --swarm \
    --swarm-discovery token://<TOKEN> \
    --swarm-master \
    machine0-manager

docker-machine create \
    --driver virtualbox \
    --swarm \
    --swarm-discovery token://<TOKEN> \
    machine1

docker-machine create \
    --driver virtualbox \
    --swarm \
    --swarm-discovery token://<TOKEN> \
    machine2
```

注意这个 flag

在 docker-machine 的 ls 子命令的输出中可以确认这些机器为 Swarm 集群的一部分，集群管理者的名称包含在集群中任何节点的标签为 SWARM 的列中：

```
NAME                ... URL                        SWARM
machine0-manager        tcp://192.168.99.106:2376 machine0-manager (manager)
machine1                tcp://192.168.99.107:2376 machine0-manager
machine2                tcp://192.168.99.108:2376 machine0-manager
```

Docker 客户端可以被配置为与任何这些机器单独工作，它们都可以通过一个公开的 TCP 套接字（就像任何其他的 Docker Machine）运行 Docker Daemon，但是，当你配置你的客户端在 master 节点上使用 Swarm 端点时，你可以就像使用一台大的 machine 那样开始用集群工作。

12.2.2 Swarm 扩展了 Docker 远程 API

Docker Swarm manager 端点公开了 Swarm API，Swarm 客户端可以使用那些 API 来控制或者检查集群。更重要的是 Swarm API 是对于 Docker 远程 API 的扩展。这意味着任何 Docker 客户端都可以直接连接到一个 Swarm 端点，并像一台单独的 macher 那样来操作一个集群。

Swarm manager 如何委派由 Docker 客户端指定的工作到集群中的节点，如图 12.5 所示。

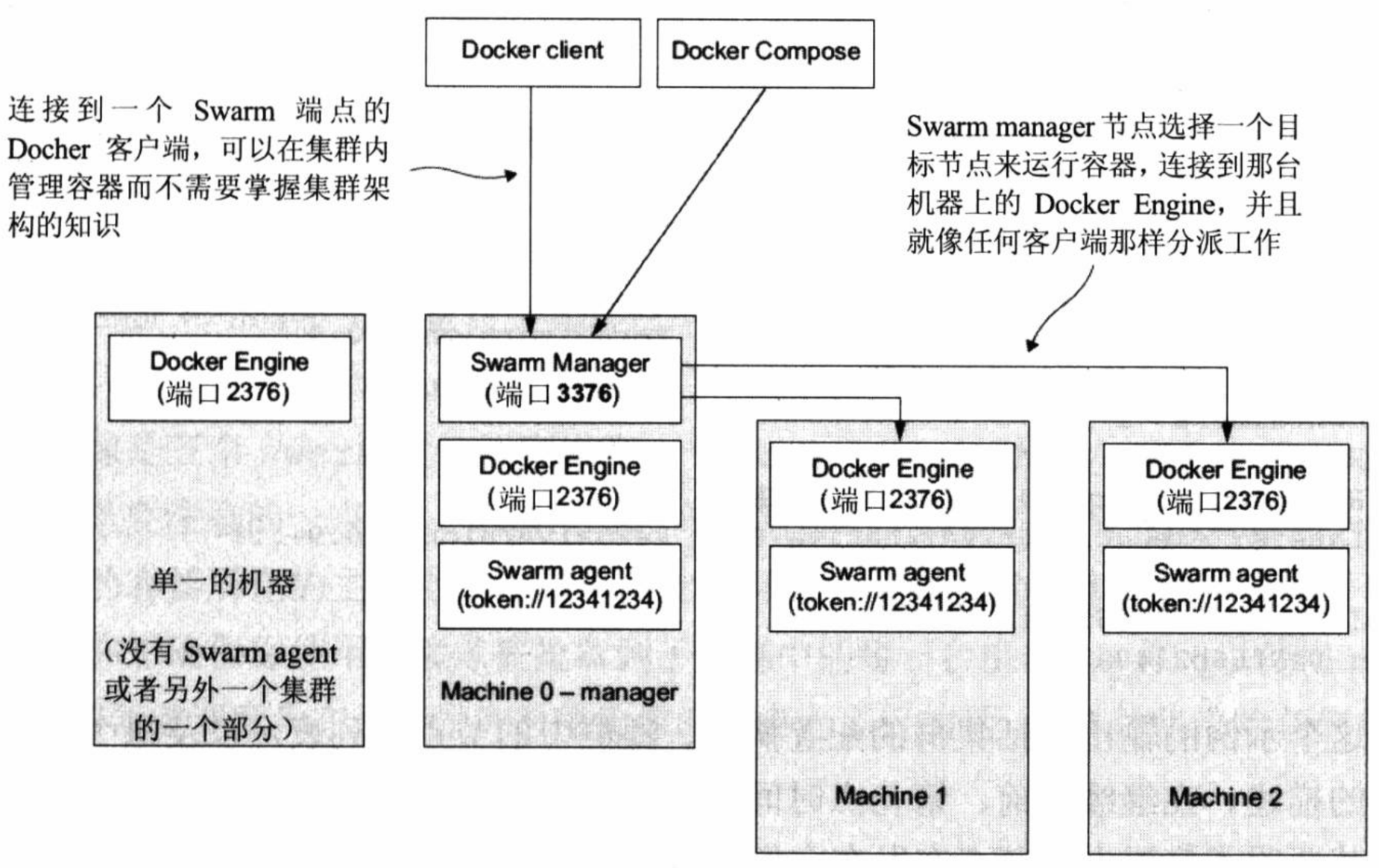


图 12-5 在集群中部署一个容器不需要集群方面的知识，因为 Swarm API 扩展了 Docker 远程 API。

由 Swarm 提供的 Docker 远程 API 的实现非常不同于 Docker Engine，根据特定的功能，来自客户端的单个请求可能会影响一个或多个 Swarm node。

使用你在上一节创建的 Swarm 集群来配置你的环境，为此添加--swarm 参数到 docker-machine env 子命令。如果你使用兼容 POSIX 的 Shell，运行以下命令：

```
eval "$(docker-machine env --swarm machine0-manager)"
```

如果你使用 PowerShell，运行如下：

```
docker-machine env --swarm machine0-master | Invoke-Expression
```

当你的环境被配置为访问一个 Swarm 端点时，docker 命令行接口将使用 Swarm 功能特性。例如使用 docker info 命令将会报告整个集群的信息而不是某个特定 Docker Daemon 的细节：

```
docker info
```

输出将类似于如下所示：

```
Containers: 4
Images: 3
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 machine0-manager: 192.168.99.110:2376
  ? Containers: 2
  ? Reserved CPUs: 0 / 1
  ? Reserved Memory: 0 B / 1.022 GiB
  ? Labels: executiondriver=native-0.2, kernelversion=4.0.9-...
 machine1: 192.168.99.111:2376
  ? Containers: 1
  ? Reserved CPUs: 0 / 1
  ? Reserved Memory: 0 B / 1.022 GiB
  ? Labels: executiondriver=native-0.2, kernelversion=4.0.9-...
 machine2: 192.168.99.112:2376
  ? Containers: 1
  ? Reserved CPUs: 0 / 1
  ? Reserved Memory: 0 B / 1.022 GiB
  ? Labels: executiondriver=native-0.2, kernelversion=4.0.9-...
CPUs: 3
Total Memory: 3.065 GiB
Name: 942f56b2349a
```

注意这个示例的输出包括集群的配置摘要、集群中的节点的列表，以及每个节点上的可用资源的描述。在继续之前，请花点时间回味一下这里发生了什么。这是你见过的第一个证据，其证明了集群中的节点在发布它们的端点信息，并且 manager 发现了这些节点，此外，所有这些信息都是特定于一个 Swarm 集群的，但是你可以通过 docker 命令获得，这个命令与在一台单独的 machine 操作是相同的。接下来，在集群内创建一个容器：

```
docker run -t -d --name hello-swarm \
    dockerinaction/ch12_painted \
    Hello Swarm
```

这次你以 detached 模式运行 ch12painted 容器，并将在日志中输出，容器也不会被自动删除。你可以用 logs 子命令查看容器的输出：

```
docker logs hello-swarm
```

```
 _   _        _ _          ____
| | | | ___| | | ___    / ___|_      ____ _ _ __ _ __ ___
| |_| |/ _ \ | |/ _ \   \___ \ \ /\ / / _` | '__| '_ ` _ \
|  _  |  __/ | | (_) |   ___) \ V  V / (_| | |  | | | | | |
|_| |_|\___|_|_|\___/   |____/ \_/\_/ \__,_|_|  |_| |_| |_|
```

这个命令看起来就像你是在访问一台单独的 machine，事实上，该协议是一样的，所以任何可以从 Docker 远程端点获得日志的 Docker 客户端都可以从 Swarm 端点获得日志，你可以用 ps 子命令发现哪一台机器正在运行容器。使用过滤来抓取名为 hello-swarm 的容器：

```
docker ps -a -f name=hello-swarm
```

注意在 NAMES 列中容器名字是以你的集群中的一台机器名字作为前缀的，这可从是任何节点。如果你创建了一个类似的但有一个略显不同的名字，就像 hello-world2 的容器，集群可能会在不同的主机上调度该容器。不过，在你这样做之前，花一点时间再次检查集群信息：

```
docker info
```

注意集群中的容器和镜像的数量：

```
Containers: 5
Images: 4
```

Container 和 Image 的数量对于集群是不太清晰的总和，因为在集群中有三个节点，你需要三个代理容器和一个 manager 容器，剩下的容器是你刚刚创建的 hello-swarm 容器。这四个镜像是三个 swarm 镜像的副本和一个 dockerinaction/ch12_painted 镜像的副本。这里要注意的一点是，当你用 run 命令创建一个容器时，所需的镜像将只会被拉取到被调度的容器所在的主机上。这就是为什么只有一个镜像的副本，而不是三个。

如果你想确保你使用的镜像被拉取到了集群中的每一台机器上，你可以使用 pull 子命令，如果容器被调度为在一个没有所需镜像的节点上运行的话，这样做将消除任何热身延迟：

```
docker pull dockerinaction/ch12_painted
```

这个命令将在每一个节点上启动拉取操作：

```
machine0-manager: Pulling dockerinaction/ch12_painted:latest... :
    downloaded
machine1: Pulling dockerinaction/ch12_painted:latest... : downloaded
machine2: Pulling dockerinaction/ch12_painted:latest... : downloaded
```

同样，删除容器将会在任何一台机器上删除指定的容器，而删除镜像将会从集群中的所有节点删除该镜像。Swarm 对于用户隐藏了所有这些复杂的细节，这样做使得用户可以解决更有趣的问题。

虽然并不是每一个决定都可以凭空而定，不同的容器调度算法对于集群效率和性能都有显著的影响。接下来，你将学习不同的算法，并提示大家如何操作这些 Swarm 调度器。

12.3 Swarm 调度

采用 Linux 容器作为你的部署单元的最有力的论据之一是你可以更有效地利用你的硬件以及削减硬件和能源成本，不过这样做需要在集群中聪明地放置容器。

Swarm 提供了三种不同的调度算法，每一个都有自己的优势和不足。当创建 Swarm manager 时就会设置调度算法，用户可以对于一个给定的 Swarm 集群通过为特定的容器提供约束来调整调度算法。

可以为每个容器设置约束，但是因为调度算法是在 Swarm manager 上设置的，所以创建集群时你需要指定这些设置。Docker Machine 的 `create` 子命令为这一目的提供了 `--swarm -strategy` 参数，默认选择是 `spread`。

12.3.1 Spread 算法

使用 Spread 算法的 Swarm 集群将试图在未充分利用的节点上调度容器，并对所有节点同样地传播工作负载，算法通过它们的资源利用率明确地为集群中的节点进行排名，然后根据每个节点运行的容器数量来对那些资源排名相同的节点进行排名。可用资源最多以及容器最少的机器将被选中来运行一个新的容器，如图 12-6 所示了在有三个相似的节点的集群中如何调度三个相等大小的容器。

在图 12-6 中可以看到，三个被调度的容器每一个都被放置在一台单独的主机上。此时此刻，这三台机器中的任何一个都会作为候选来运行第四个容器。当调度第四个容器时，选中的机器将运行两个容器，而调度第五个容器时将导致集群在两台只运行一个容器的机器中选择一台。机器的选择排名是由正在运行的容器数量相对于集群中的所有其他机器的容器比例来定的。那些运行更少容器的机器将有一个更高的排名，如果两台机器有相同的排名，那么将随机选择一台。

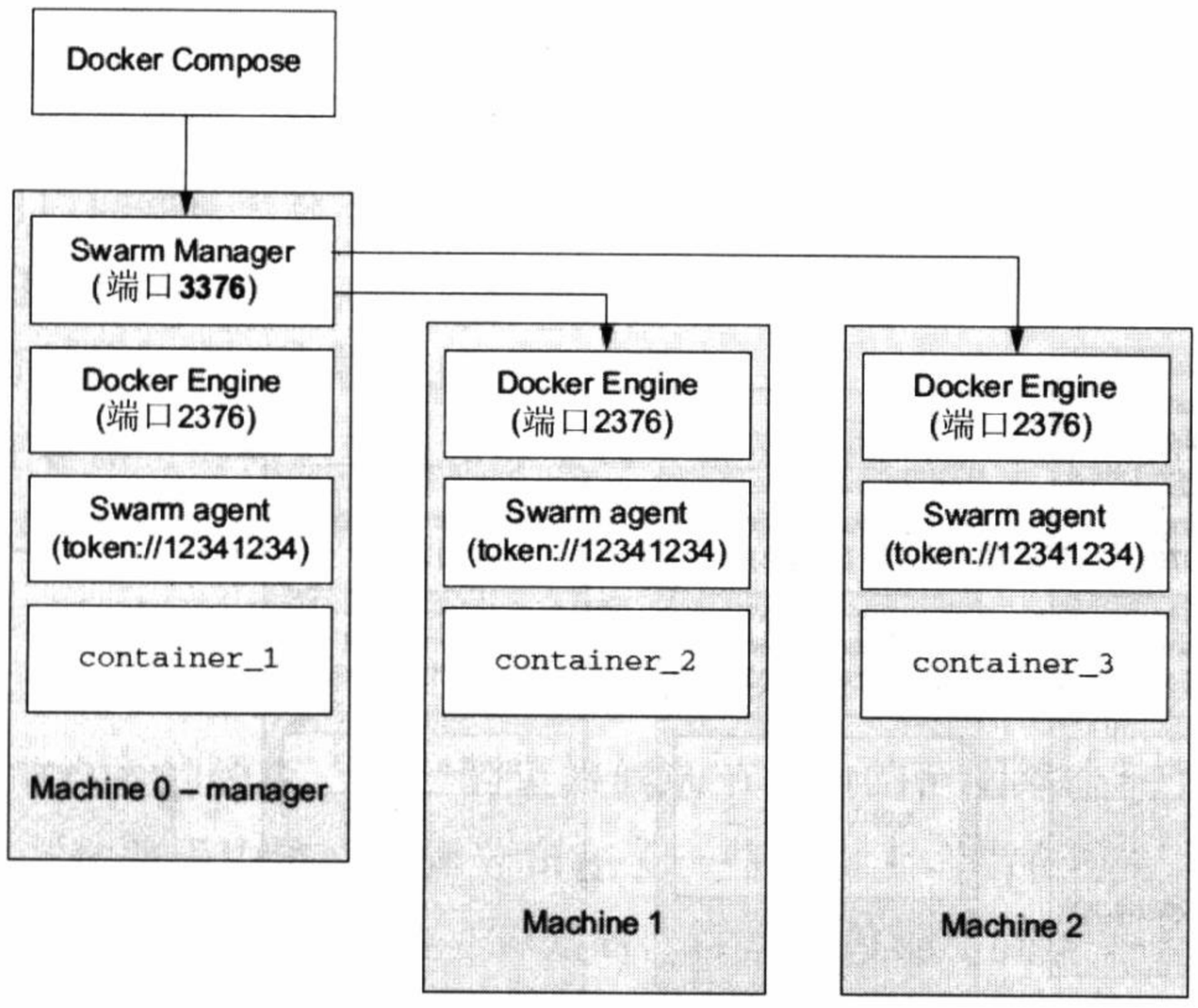


图 12-6 Spread 算法在集群中跨节点均匀地传播工作负载

Spread 算法将尝试均匀地使用整个集群，这样做会减少随机的机器故障的潜在影响，并将机器拥塞与集群拥塞绑定综合考虑，比如考虑一个运行大规模的、有着不同数量的容器副本的应用程序。

在 flock.json 文件中创建一个新的 Compse 环境描述，并定义一个名为 bird 的服务，该服务定期地按照标准输出绘制 bird：

```
bird:
  image: dockerinaction/ch12_painted
  command: bird
  restart: always
```

当你使用 Compose 向上扩展时，观察 Swarm 跨集群地分发了 10 个 bird 服务的副本：

创建一些 birds

```
docker-compose -f flock.yml scale bird=10
docker ps
```

检查容器分发

ps 命令的输出包含 10 个容器，每个容器的名字将会用部署该容器的机器的名字作为前缀。在集群中它们会被均匀地跨节点分布。在本实验中使用以下两个命令清理容器：

```
docker-compose -f flock.yml kill
docker-compose -f flock.yml rm -vf
```

该算法在容器已经设置了资源预订的情况下效果最好，那些限制有一些低程度的差异。由于容器所需的资源以及节点提供的资源是多样化的，Spread 算法会导致一些问题，如图

12-7 所示。

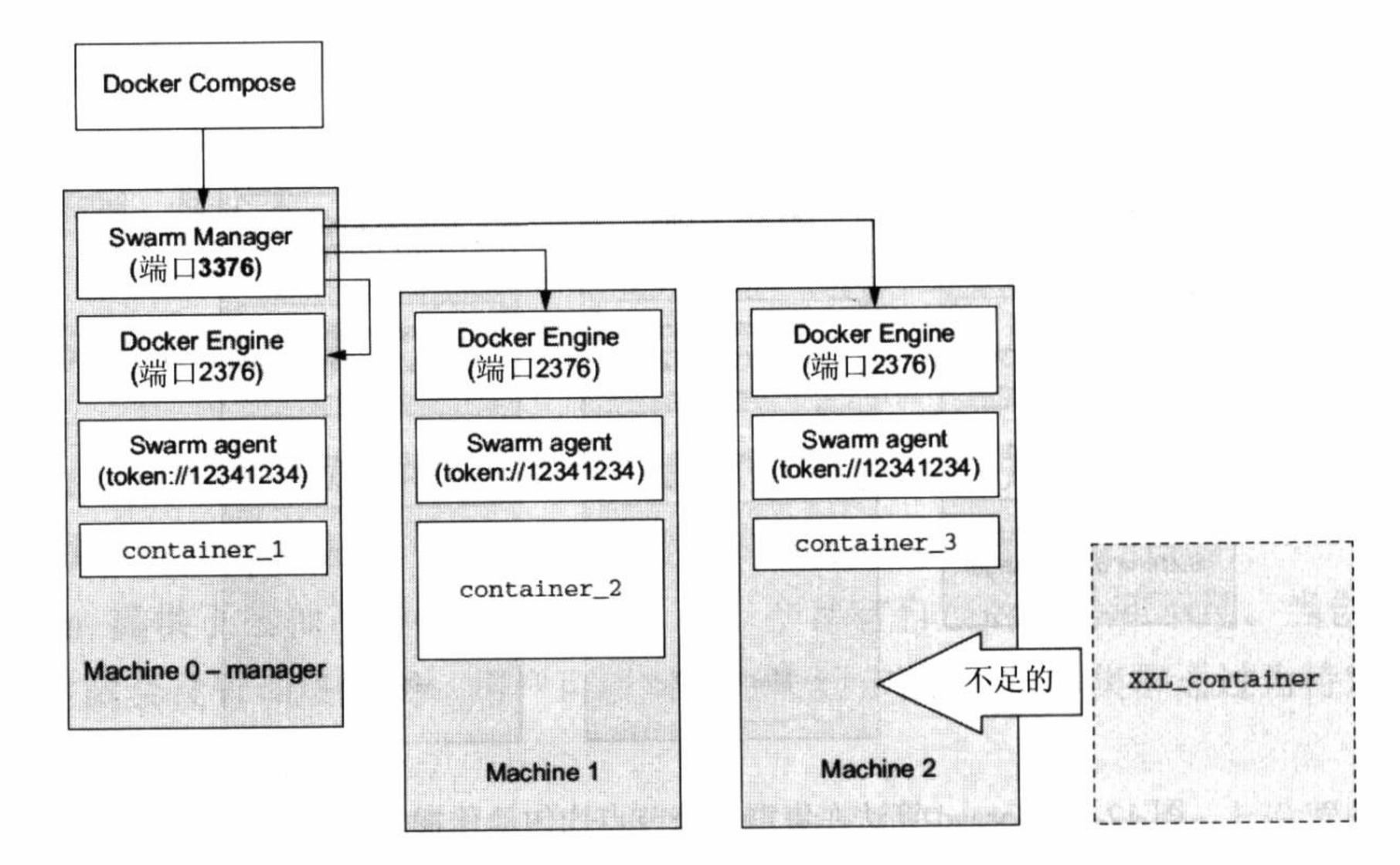


图 12-7 由于分布不均，新的容器无法被调度

图 12-7 演示了如何在容器所需的资源里边引入高阶差异，从而引发了 Spread 算法的糟糕性能。在这种情况下，因为没有机器有足够的资源，新的容器 `XXLcontainer` 不会被调度。现在回想起来，如果已经在 Machine 0 或者 Machine 1 机器上调度了 `container-3` 容器，很明显地，调度器就会避免发生这种情况。所以如果你使用过滤器，你就可以避免这种情况而不需要改变调度算法。

12.3.2 用过滤器调整调度

在 Swarm 调度器使用调度算法之前，它根据 Swarm 配置和容器的需求收集和过滤一组候选节点，每个候选节点都会经过为集群配置的每一个过滤器。可以使用 `docker info` 命令来发现集群使用的活跃过滤器。当 `docker` 客户端被配置为 Swarm 端点时，`info` 子命令的输出将会包含类似如下所示：

```
Filters: affinity, health, constraint, port, dependency
```

启用过滤器的集群将会减少候选节点的集合，以获得关联、约束、依赖性或者被调度的容器需要的端口以及健康节点的端口。

关联是用另一个容器或镜像进行托管的需求。约束是对于一些机器属性（比如内核版本、存储驱动器、网络速度、磁盘类型等）的需求。虽然你可以使用一组预定义的机器来定义约束，你还可以创建任何标签上的约束，这些标签已经应用到了节点的守护进程。依赖性是一个模拟的容器依赖关系，比如链接或共享卷。最后，端口过滤器将减少候选节点的集合，以获得可用的被请求主机端口。

可以用定义在容器上的标记节点和标签约束避免如图 12-7 所描述的不良分布，如图 12-8 所示了一个更好的配置系统的方法。

在图 12-8 中，Machine 0 和 Machine 1 都是用标签 `size=small` 创建的，Machine 2 被标记为 `size=xxl`。当容器 1～3 创建时，提供了一个环境变量指出被标记为 `size=small` 的节点上的约束，这一约束将候选节点池的范围缩小到了 Machine 0 或者 Machine 1 上。在其他例子中这些容器分布在了一些节点上。当在标记为 `size=xxl` 的节点上用一个约束创建 `xxl_container` 容器时，候选池被缩小到了单个节点上。只要其中任何一个节点由于其他任何原因没有做过滤，都将会被调度到 Machine 2。

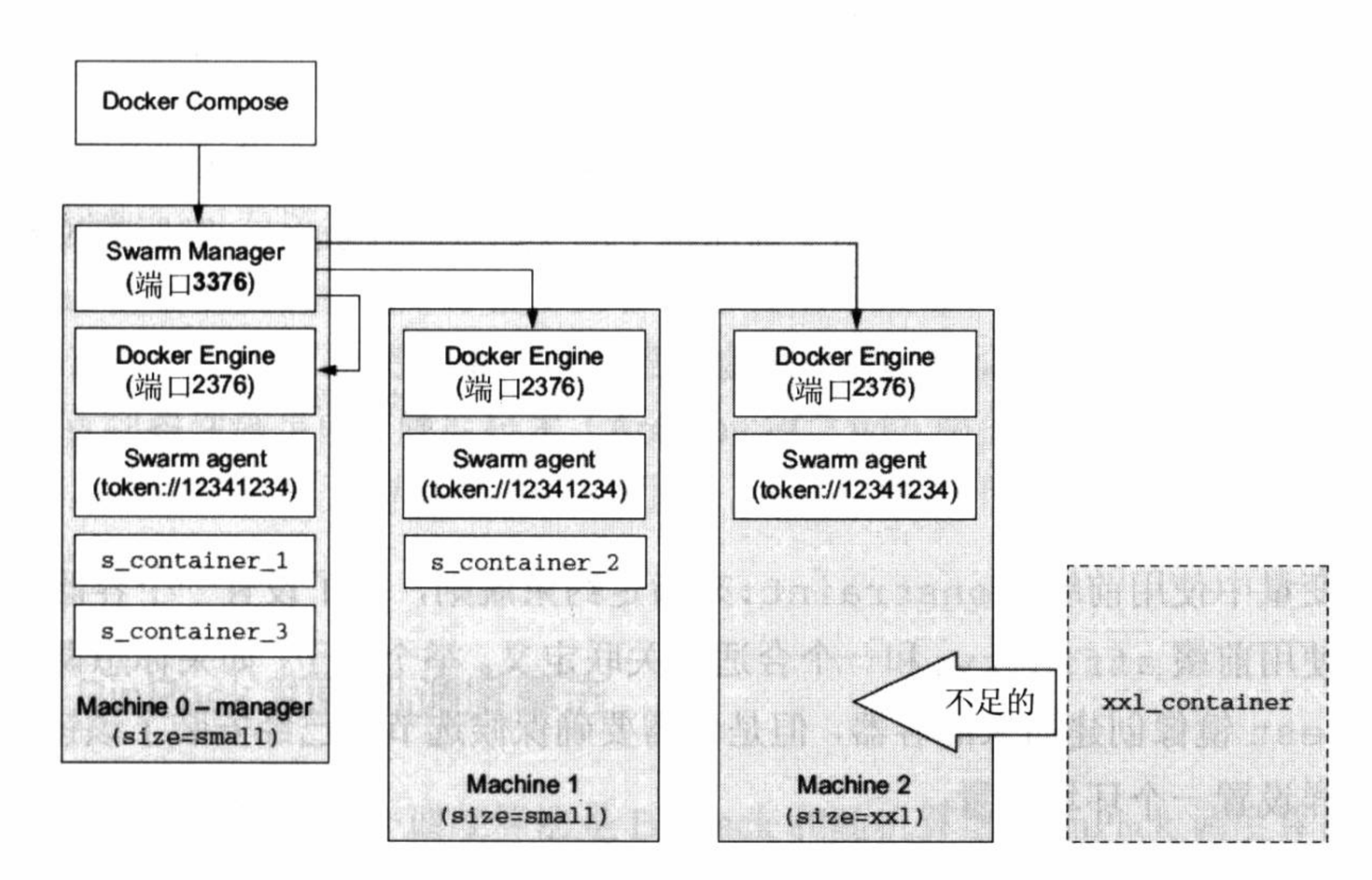


图 12-8 节点标签和约束的容器修剪调度候选人到适当的节点

你可以设置 `docker-machine create` 命令的`--enginelabel` 参数来给集群中的节点打上标签。例如，可以使用以下命令在集群中创建一个标签为 `size=small` 的新节点：

```
docker-machine create -d virtualbox \
    --swarm \
    --swarm-discovery token://<YOUR TOKEN> \
    --engine-label size=small \
    little-machine
```

应用一个 engine 标签

```
docker-machine create -d virtualbox \
    --swarm \
    --swarm-discovery token://<YOUR TOKEN> \
    --engine-label size=xxl \
    big-machine
```

应用一个 engine 标签

除了用于节点的任何标签，容器可以为所有节点默认指定的标准属性指定约束：

- node——集群中节点的名字或者 ID
- storagedriver——节点所使用的存储驱动器的名称
- executiondriver——节点使用的执行驱动程序的名称
- kernelversion——节点的 Linux 内核版本
- operatingsystem——节点上的操作系统名称

每个节点都会为 Swarm Master 提供这些值，你会发现集群中每个节点使用 docker info 子命令报告的内容。

容器使用环境变量来与它们的关联和约束需求通信，可以使用一个单独的环境变量来设置每个约束或关联规则。比如可能用于前面例子的 xxl_container 容器的规则看起来像以下这样：

```
docker run -d -e constraint:size==xxl \
    -m 4G \
    -c 512 \
    postgres
```

约束环境变量

在环境变量中使用前缀 constraint:来指定约束规则，为了设置一个容器关联，在环境变量中使用前缀 affinity:和一个合适的关联定义。举个例子，如果你想调度一个从 nginx:latest 镜像创建而来的容器，但是你需要确保候选节点已经安装了该镜像，你可以像以下这样设置一个环境变量：

```
docker run -d -e affinity:image==nginx \
    -p 80:80 \
    nginx
```

关联环境变量

任何安装了 nginx 镜像的节点都会在候选节点集合中。另外，如果你想运行一个类似的程序，比如 ha-proxy，相比较而言，需要确保程序运行在一个单独的节点，你可以创建一个否定关联：

```
docker run -d -e affinity:image!=nginx \
   -p 8080:8080 \
    haproxy
```

否定关联环境变量

这个命令将从候选节点集合中移除任何包含 nginx 镜像的节点，目前为止你所看到的这两条规则使用的是不同的规则操作符：==和!=，但规则语法的其他组成部分会使其具有很强的表现力。一个关联或者约束规则是由一个键、一个操作符和一个值做成的，而键必须是已知的并完全合规的，值可以有以下三种形式：

- 完全限定为字母、点、连字符、数字及下画线（比如 my-favorite.image-1）
- 全局语法指定模式（比如 my-favorite.image-*）
- 一个完整的 Golang 风格的正则表达式（比如/my-[a-z]+\.image-[0-9]+/）

创建有效规则的最后一个工具是软操作符，当你想做调度而不是规则时，在操作符的最后添加波浪字符。例如，如果你想建议 Swarm 调度一个 nginx 容器，该容器所在的节点已经安装了该镜像，不过如果条件不能满足，就进行调度，那么你会使用一个如下的规则：

```
docker run -d -e affinity:image==~nginx \
    -p 80:80 \
    nginx
```

建议关联环境变量

过滤器可以用来定制任何调度算法，借助于预见到你的预期工作量和基础设施的不同性质，过滤器可以发挥很大的效用。

Spread 算法的定义特性（即使是由容器卷使用的集群）将总是会被应用于过滤节点集，它始终与云的一个特征冲突。当只有一些节点闲置时，自动伸缩集群中的节点数目是可行的。不过直到条件满足前，底层技术（Amazon EC2 web 服务或任何其他服务）将无法缩小你的集群。这将导致超大型集群的低使用率，在这种情况下采用 BinPack 调度算法可以帮助解决这个问题。

12.3.3 BinPack 和随机调度算法

你应该理解的另外两个调度算法是 BinPack 和随机算法，BinPack 调度算法倾向于在调度工作负载到另外的节点前最有效地利用每一个节点。该算法使用最少数量的必需节点来支持工作负载，而随机算法提供了一个分配方案，其可以在 Spread 和 BinPack 之间达成妥协。候选池中的每个节点都有一个平等的被选中的机会，但该算法并不保证会在侯选池中实现均匀地分配工作负载。

在你采用任何一个算法之前，有几个注意事项值得回顾。如图 12-9 所示了 BinPack 算法是如何调度三个容器的。

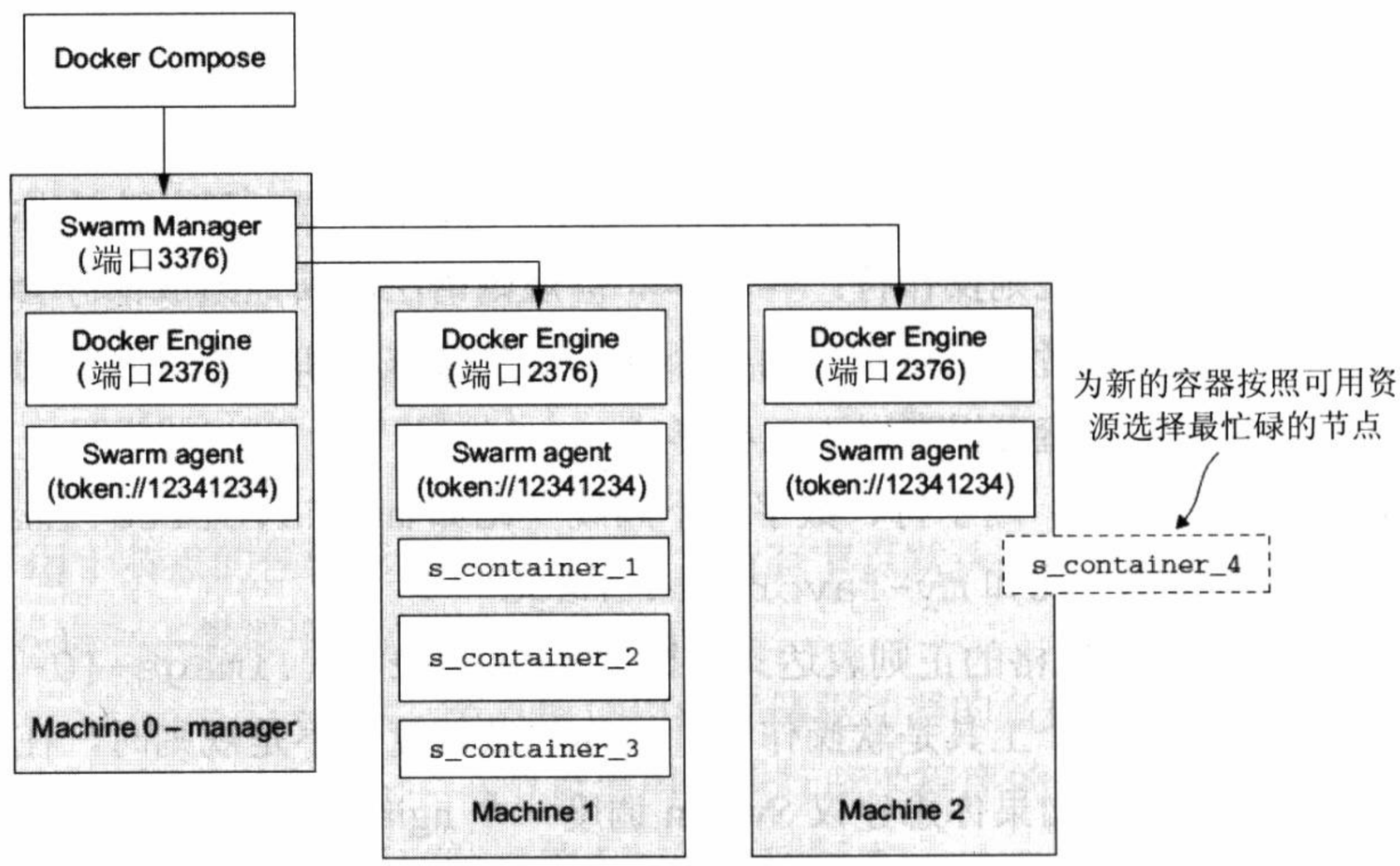


图 12-9 BinPack 只有在最忙碌的节点的资源不足以承担更多的工作的情况下才调度一个新节点

BinPack 可以在只知道被调度的容器的资源需求的情况下做出明智的有关有效打包的决定。出于这个原因，BinPack 只在你致力于为你系统中的容器创建资源预定的情况下有意义，使用资源隔离特性（如内存限制、CPU 权重和阻塞 IO 权重）会将你的容器从相邻的资源滥用中隔离出来。虽然这些限制不会创建本地资源预订，Swarm 调度器将把它们作为保留对象来处理，以防止任何一台主机负载过重。

BinPack 会解决你使用 Spread 算法遇到的最高的问题，其将通过优先考虑每个节点的有效使用来储备节点上的大量资源，该算法采用贪婪的方法来选择最忙碌的同时资源最少但仍足以满足容器需求的节点，这有时被称为一个最适合的节点。

如果你系统中的各个容器在资源需求方面有很大的差异性或者项目需要一个最小的集群以及自动缩小规模的选项，BinPack 尤其有用。而 Spread 算法在专用的集群系统中是最合理的方式，BinPack 在一个具备按需伸缩特性的完全虚拟化的集群中是最合理的方式。不过这种弹性是以付出可靠性为代价的。

BinPack 创建最小数量的最大关键节点，这个分配方案在几个忙碌的节点上增加了失败的可能性，增加了此类故障的影响。这也许是一个可接受的妥协，不过在构建一个关键的系统时要记住这个缺点。

随机算法在 Spread 和 BinPack 之间作了一个妥协，它仅依赖于概率从候选节点中选择。在实践中，这意味着可能在你的集群中同时包含忙碌节点和闲置节点。如图 12-10 所示了一个可能的由三个节点组成的候选集合的三个容器的分配方案。

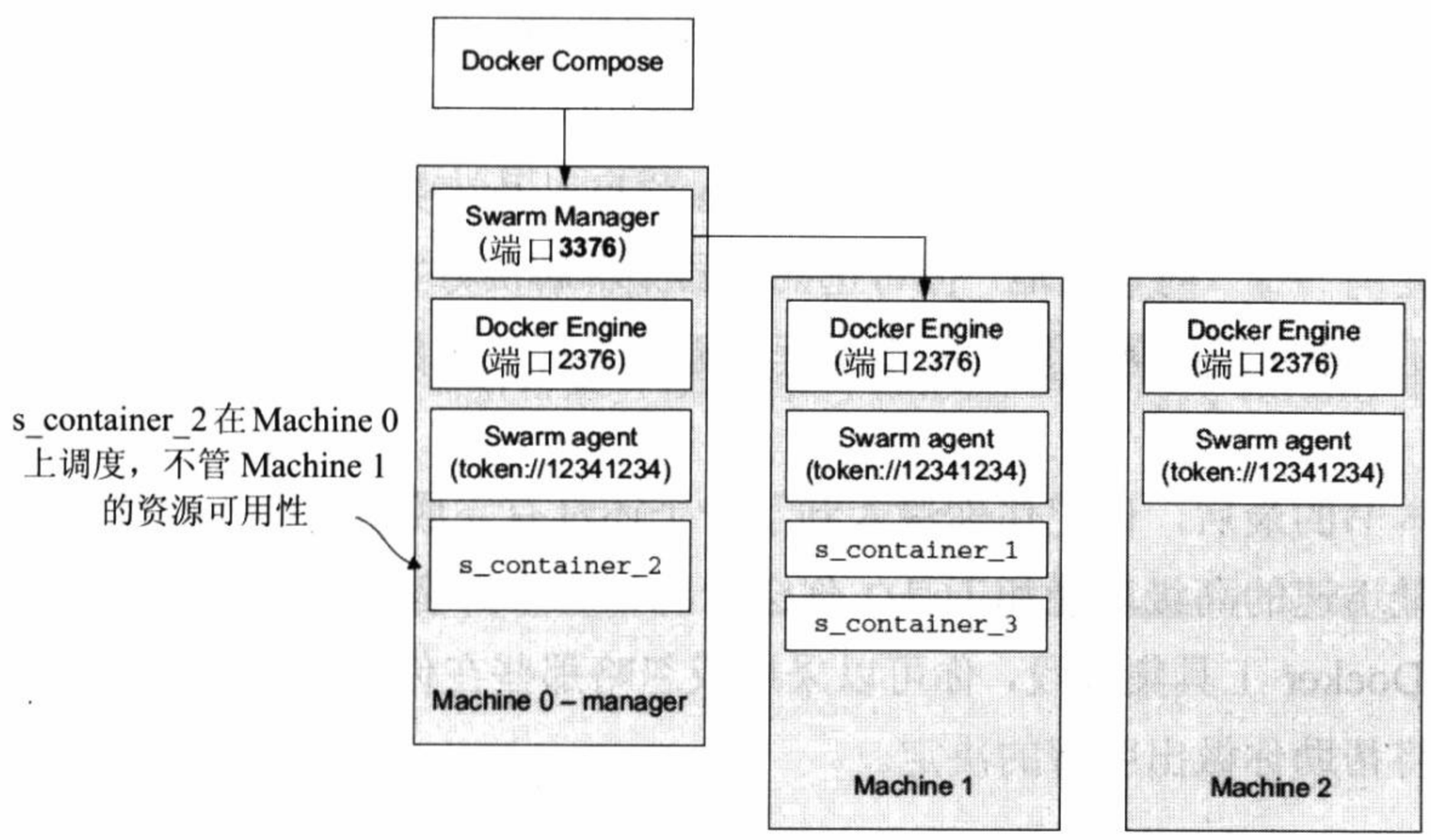


图 12-10　随机调度算法选择机器仅考虑概率

这个算法或许能会在集群中公平地分配工作，并且很可能适应你的容器集合中资源需求方面的巨大差异，但也可能意味着没有保证。这是可能的，该集群将包含热点或者使用效率低下的闲置节点，并且缺乏大型资源块以便提供给大型容器。一些工程师可能倾向于这个算法，所以更大的系统必须考虑这些可能性。

随着 Swarm 日渐成熟以及更多的数据节点可用，可能会出现新的调度算法。该项目本身还处于初期阶段，但这三个算法为健壮的系统提供了基本的构建块。调度可以让系统选择容器在哪里运行，而 Swarm 解决的最后一个问题是帮助那些容器用服务发现机制互相定位。

12.4　Swarm 服务发现

任何分布式系统都需要一些机制来定位，如果分布式系统是由在同一台机器上的多个进程组成的，它们需要在一些指定的共享内存池或者队列上达成一致。如果组件被设计为是通过网络进行交互的，它们需要为彼此的命名达成一致意见，并决定一个机制来解析这些名字。在大多数时候，网络应用程序依赖于 DNS 作为名字到 IP 地址的解析方案。

Docker 使用容器链接将静态配置注入一个容器的名字解析系统，在这一过程中，包含的应用程序不需要知道它们运行在一个容器中，但是一个单独的 Docker Engine 对于运行在其他主机的服务没有可见性，并且服务发现仅限于本地运行的容器。

另外，Docker 允许用户设置默认的 DNS 服务器，并为每一个容器或者每个主机上的容器搜索域名，DNS 服务器可以是公开 DNS 接口的任何系统，在过去的几年中，已经出

现了好几个这样的系统，其所形成的一个丰富的生态系统已经进化到了在多主机环境中解决服务发现问题。

随着这些系统的发展，Docker 宣布了 Swarm 项目以及该类型系统的通用接口。Swarm 项目的目标是提供一个“内置电池”但可选的容器集群解决方案。这个项目的一个重要里程碑是在多主机环境中实现服务发现，实现里程碑需要多种技术的发展以及 Docker Engine 的功能增强。

当你看到本书的最后，请记住你深入到了一个还没有完成的故事的结尾。这些都是系统中一些发展最迅速的高级功能和工具。你会看到每一个野蛮生长的服务发现方法，就像大多数其他的 Docker 工具集片段，你可以采用或忽略那些在你的情形中有意义的部分。本节的其余部分将帮助你做出明智的决定。

12.4.1 Swarm 和单主机网络

Docker Engine 在其被安装的每台机器上的桥接网络后面创建了本地网络，第 5 章深入探讨了这个话题。部署的容器位于一个 Docker 节点上，它已经超出了 Swarm agent 的范围来重组网络，该网络可以对于其他 Swarm 节点和集群中容器的发现做出反应。出于这个原因，如果一个 Swarm 集群部署到了运行单主机网络的 Docker 机器上，那么用 Swarm 部署的容器只能发现运行在相同主机上的其他容器。如图 12-11 所示了单主机网络如何限制调度器的候选集。

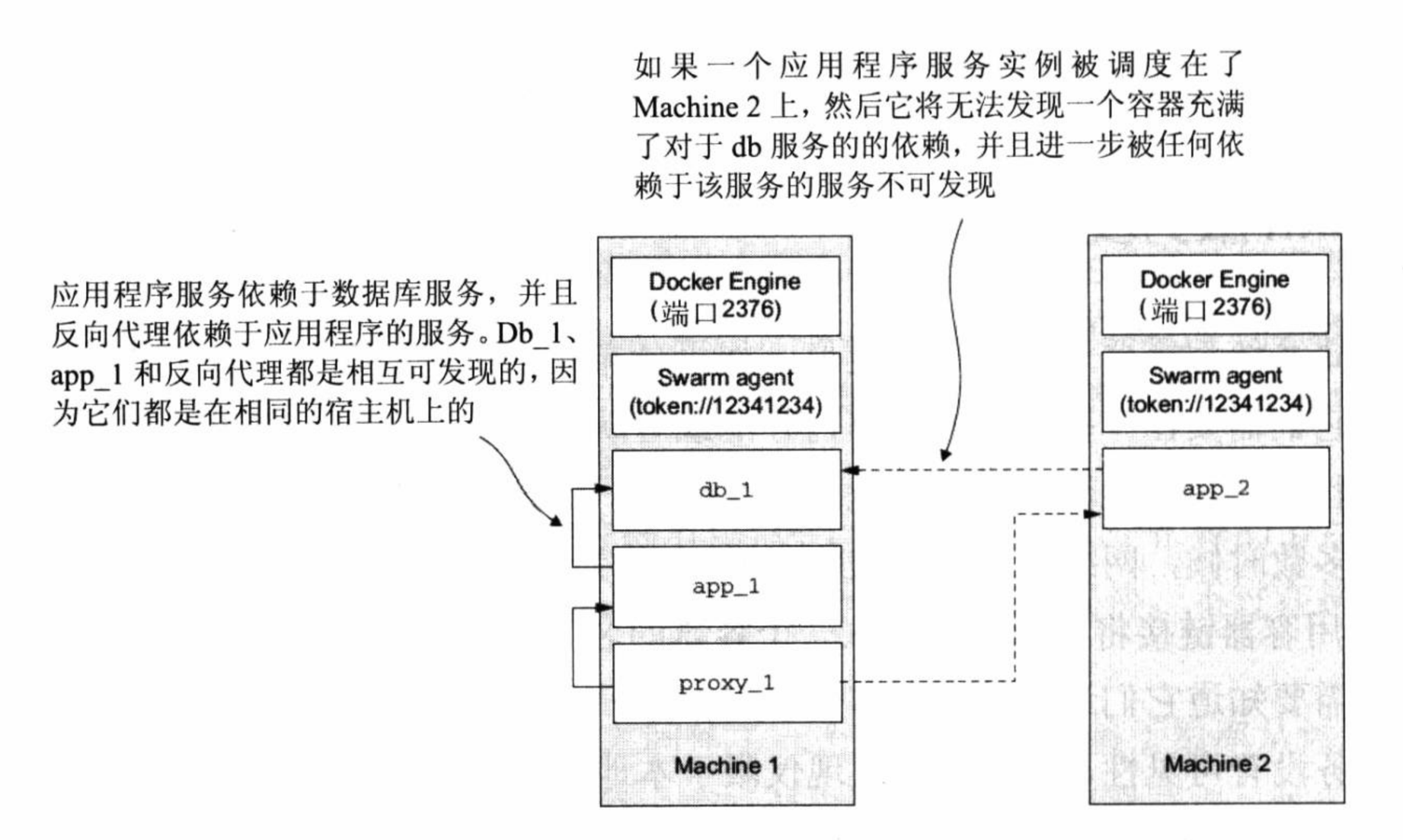


图 12-11 一个三层应用程序部署在一个单主机网络的集群中

这个依赖过滤器确保容器从来被不会调度到一台无从发现依赖关系的主机上，你可以通过部署 Coffee API 到现有的 Swarm 集群上尝试这个过滤器。

首先，使用 Git 为 Coffee API 示例克隆 Compose 环境描述：

```
git clone git@github.com:dockerinaction/ch12_coffee_api.git
```

运行这个命令后，切换到新目录并确保你的 Docker 环境指向 Swarm 集群：

```
cd ch12_coffee_api
eval "$(docker-machine env machine0-manager)"
```

一旦设置了你的环境，你就已经准备好启动 Docker Compose 的示例了，下面的命令将在 Swarm 集群中启动环境：

```
docker-compose up -d
```

现在使用 `docker` CLI 查看集群中新容器的分布，由于机器名是所有容器名字和别名的前缀，输出将会非常广泛。例如，运行在机器上的名为“bob”的容器将会被显示为“machine1/bob”，你可能想要重定向或复制输出到一个文件，以便可以在没有换行线的情况下查阅：

```
docker ps
```

如果你安装了 `less` 命令，你可以使用-S 参数来切分长行并用箭头导航：

```
docker ps | less -S
```

lsee 命令可能在你的系统不可用

在检查输出时，你会发现每个容器都是运行在同一台机器上的，即使你的 Swarm 配置为使用 Spread 算法。一旦第一个容器被调度了，并且该容器依赖另外一个容器，依赖性调度器将会从候选池中排除任何其他节点。

现在环境正在运行中，你应该可以查询服务。注意部署该环境的机器的名字，并且在如下的 cURL 命令中替换<MACHINE>：

```
curl http://$(docker-machine ip <MACHINE>):8080/api/coffeeshops/
```

如果你的系统没有安装 `cURL` 命令，你可以使用 web 浏览器来做相同的请求，请求的输出你应该很熟悉：

```
{
  "coffeeshops": []
}
```

请花些时间来对 Coffee 服务作伸缩性实验，检查 Swarm 是在哪里调度每个容器的。当完成示例后，用 `docker-machine stop` 命令关闭并用 `docker-compose rm -vf` 命令删除容器。

尽管有这个限制，集群化的应用程序在一些使用案例方面是可行的，但是最常见的使用案例是欠缺的。服务器端软件通常需要多主机分布和服务发现。社区已经构建和采用了一些新的和现有的工具来填补与 Docker 集成的解决方案的空白。

12.4.2 服务发现生态系统和权宜之计

网络服务发现的主要接口是 DNS，虽然提供 DNS 服务的软件已经出现了一段时间，传统的 DNS 服务器软件仍然使用很重的缓存，并努力提供高写入吞吐量，但通常缺乏成员监控。这些 DNS 系统不能在频繁部署的环境下动态伸缩，现代系统使用支持高写入吞吐量、会员管理甚至分布式锁设施的分布式键值对数据库。

服务发现软件的例子包括 etcd、Consul、ZooKeeper 和 Serf，它们在实现方面是根本不同的，但它们都是优秀的服务发现工具。每个工具都有细微差别和优缺点，不过这部分内容超出了本书的范围（可参见 http://dockone.io/article/667）。如图 12-12 所示了容器通过 DNS 或者其他的协议（如 HTTP）如何直接集成一个外部的服务发现工具。

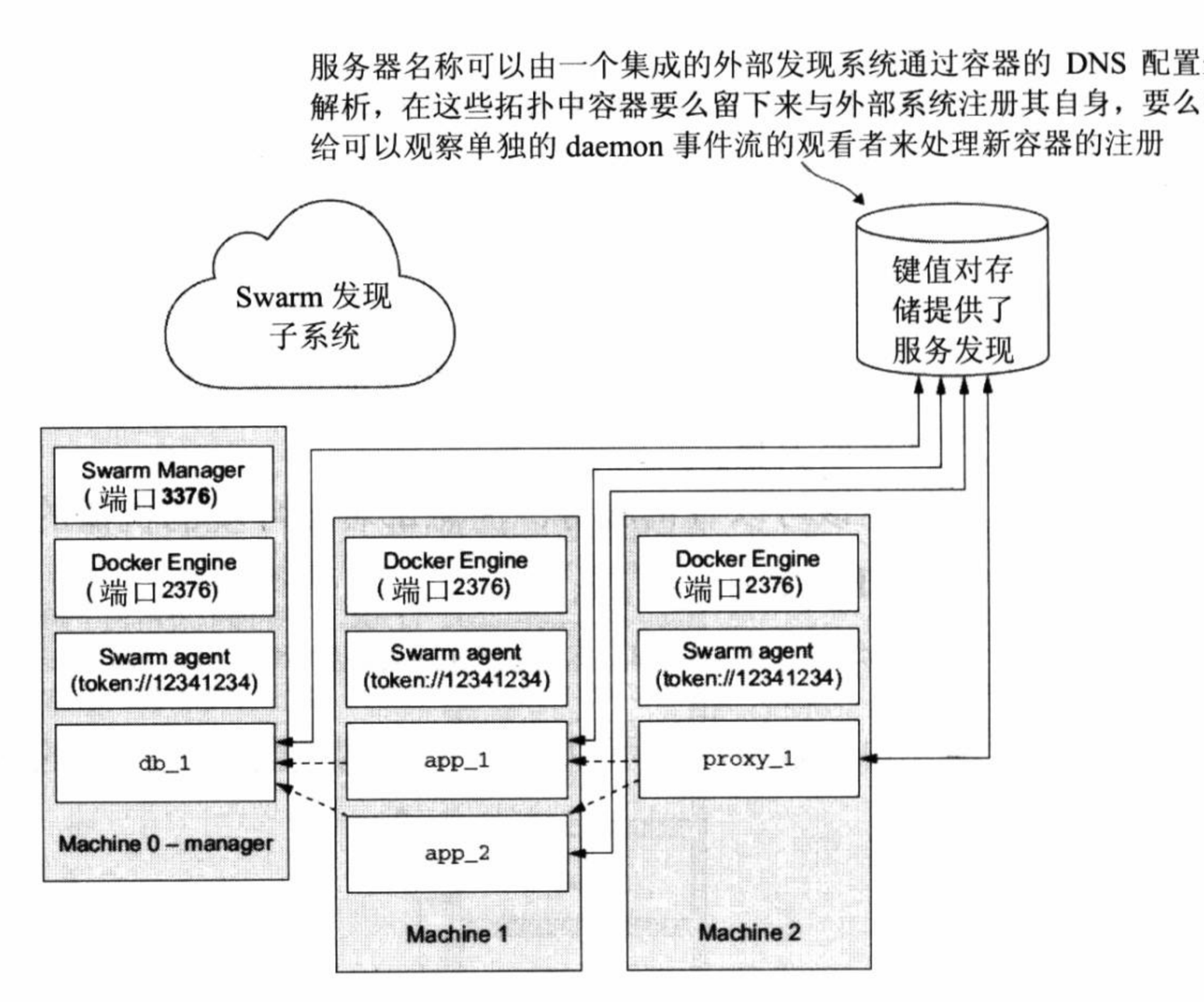


图 12-12 集群中的容器通过一个外部工具（如 etcd、Consul、ZooKeeper 或者 Serf）负责注册和服务发现

这些生态系统工具是易于理解和久经考验的，但整合容器层面的基础设施泄露了本应该隐藏的实现细节，将容器内部的应用程序与一个特定的服务发现机制做集成会减少该应

用程序的可移植性。理想的解决方案是在集群或者由 Docker Engine 和 Docker Swarm 提供的网络层集成服务注册和发现。借助于将多主机、网络使能的 Dock Engine 和 Overlay 网络技术与一个可插拔的键值对存储做集成，可以将梦想变成现实。

12.4.3 展望多主机网络

Docker Engine 的实验性分支在可插拔接口后抽象了网络设施，该接口可以被大量的网络驱动实现，包括桥接、主机和 Overlay。桥接和主机网络驱动实现了你已经很熟悉的单主机网络的特性，Overlay 驱动通过 IP 封装或者 VXLAN 实现了一个 overlay 网络，该网络为每个由 Docker Machine 管理的容器提供了可路由的 IP 地址，并被配置为使用相同的键值对存储。如图 12-13 所示了构建这样一个网络的交互过程。

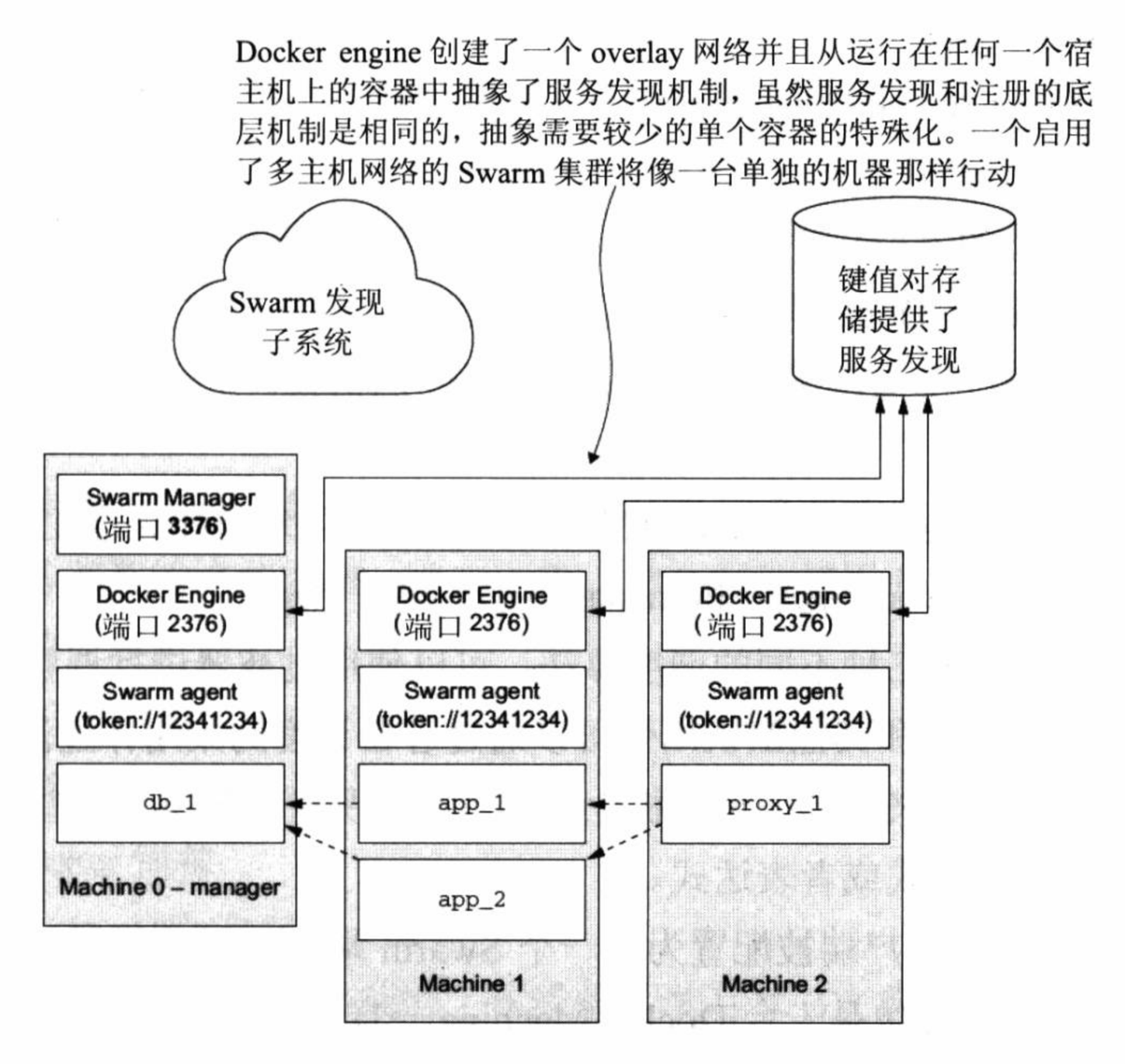


图 12-13 Docker 引擎上面的 Swarm 和多主机网络使得集群可以作为一个单独的机器，而容器充当网络中的平等成员

随着 Overlay 网络准备就绪，每一个容器都有一个唯一的 IP 地址，该 IP 可以在 Overlay 网络里的任何容器上路由。应用程序可以停止工作来确认或发布其主机 IP 地址，并将容器

端口映射到主机端口。所有这些工作都可以在由 Docker 和其集成的键值对存储一起提供的基础设施层中来执行。

当我们期待多主机网络落地成为 Docker Engine 的发行版以及由 Docker Machine 提供的多主机 Swarm 节点时，我们可以直接集成其他生态系统的项目。当这个落地时，这对于开发人员和系统架构师都将是一个安慰。

12.5 小结

Docker Machine 和 Docker Swarm 都为 Docker Engine 提供了增强应用程序的功能，这些和其他相关技术将帮助你运用在本书中所学到的从一开始在一台单独的计算机上使用 Docker 到在多台计算机上管理容器集群的知识。当你小有所成时，对于以下几点有一个全面深入的了解是非常重要的：

- 用户可以使用 Docker Machine 的 `create` 命令创建一个本地虚拟机或者在云上创建一台 Machine。
- Docker Machine 的 `env` 和 `config` 子命令可用于配置 Docker 客户端与 Docker Machine 提供的远程 Docker Engine 一起工作。
- Docker Swarm 是一个向后兼容 Docker 远程 API 的协议，并且可以在一组成员节点上提供集群设施。
- Swarm manager 程序实现了 Swarm API，并且为集群处理容器调度工作。
- Docker Machine 为 Swarm node 和 manager 都提供了 flag。
- DockerSawrm 提供了三种不同的调度算法，可以使用过滤器进行调整。
- Docker Engine 的标签和其他默认属性可以通过容器调度约束用作过滤标准。
- 容器调度关联可以用来在同一台主机上将容器置为另外的容器，这些容器或者镜像匹配一个已提供的模式或者表达式。
- 当任何一个 Docker 客户端被配置为与一个 Swarm 端点进行通信时，客户端将与整个 Swarm 互动，就好像是一台 Docker Machine 一样。
- Docker Swarm 目前调度在同一个节点上依赖的容器，直到多主机网络的发行版发布，或者你可以提供另一个服务发现机制并禁用 Swarm 的依赖性过滤器。
- 多主机网络将从 Docker 容器中的应用程序关注点抽象容器位置信息，每个容器在 Overlay 网络上都将是一台主机。

后　记

Docker 实战

Jeff Nickoloff

Docker 背后的想法是很简单的，创建一个轻量级的虚拟环境，称之为容器，其仅持有你的应用程序及其依赖。Docker Engine 使用主机操作系统来构建和管理这些容器。它们易于安装、管理和删除，容器内运行的应用程序共享资源，使得它们的足迹很微小。

《Docker 实战》教会读者如何创建、部署和管理 Docker 容器托管的应用程序，在以一个清晰透彻的对于 Docker 模式的开篇介绍之后，你将会学到如何在容器内打包应用，包括测试和分发应用的技术。你还将学习如何编排容器和应用程序以及它们的安装和卸载。本书使用精心设计的示例教你如何从安装、删除到编排容器和应用程序。从开发和测试机器到全面的云服务部署的这些领域一路走下来，你会发现使用 Docker 的各种技术。

本书包括的内容：

- 为部署打包容器
- 安装、管理和删除容器
- 使用 Docker 镜像
- 使用 Docker Hub 分发

读者只需要有 Linux 操作系统的工作经验，学习本书之前不需要知道 Docker。

软件工程师 Jeff Nickoloff 已经在沙漠代码营地、Amazon.com 和技术 meetup 上给数以百计的开发人员和系统管理员讲述了 Docker 和应用程序。