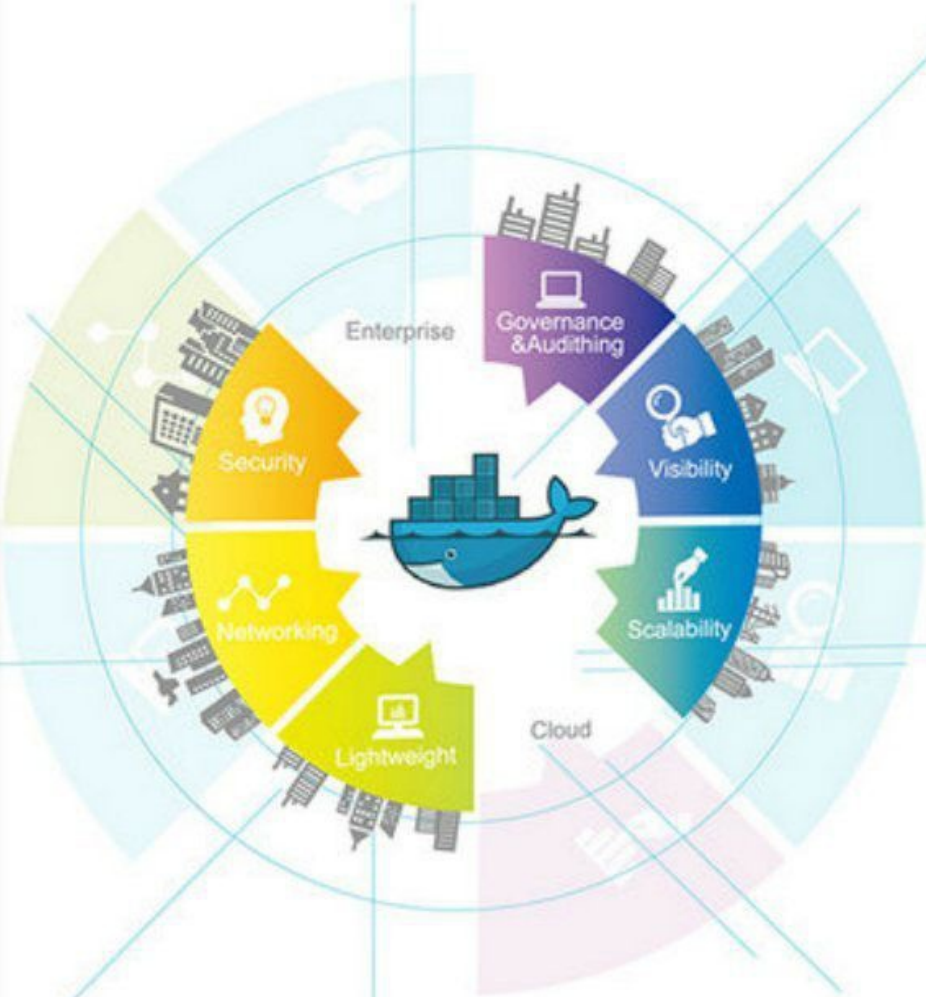
Broadview
www.broadview.com.cn

Docker全攻略

集作者多年Docker实战经验，内容具有较高的实战价值和深度

张涛 编著



中国工信出版集团
電子工業出版社

Docker全攻略

電子工業出版社

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

Docker全攻略

张涛 编著

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

电子书寻找看手相 钉钉或微信pythontesting

目　录

内容简介
前言
第一篇 Docker简介
第1章 Docker的前世今生
1.1 什么是LXC
1.2 Docker为什么选择了AUFS
1.3 Docker是如何产生的
第2章 Docker现状
2.1 Docker应用范围
2.2 Docker的优缺点
第3章 Docker将如何改变未来
第二篇 Docker基础
第4章 如何安装Docker
4.1 安装Docker前需要知道的事
4.2 Linux平台安装Docker
4.3 MAC/OS平台安装Docker
4.4 Windows平台如何支持Docker
4.5 在其他平台中安装Docker
第5章 Docker基本命令
5.1 Docker操作命令
5.2 Docker网络命令
5.3 Docker安全命令
5.4 Docker资源命令
5.5 Docker RestFul命令

5.6 Docker组件命令
第6章 Docker命令剖析
6.1 Docker操作命令
6.2 Docker网络命令
6.3 Docker安全命令
6.4 Docker资源命令
第7章 Dockerfile介绍
7.1 Dockerfile有什么用
7.2 如何编写Dockerfile
第8章 Dockerfile最佳实践
第9章 Docker部署案例
第三篇 Docker进阶
第10章 Docker运行剖析
10.1 Docker的生命周期
10.2 Docker Daemon
10.3 Docker CLI
第11章 Docker内核讲解
11.1 Docker背后的Namespace
11.2 Docker的文件系统
11.3 Docker的image管理
第12章 Docker资源调度
12.1 Docker如何管理资源
12.2 Docker资源管理器
第四篇 Docker生态圈
第13章 Docker的云生态环境
13.1 Docker的开发语言
13.2 支持Docker的开源组件

13.3 CoreOS、Vagrant和Amazon如何支持Docker
第五篇 Docker案例
第14章 基于Amazon的Docker部署案例
第15章 基于CoreOS的Docker部署案例

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

图书在版编目（**CIP**）数据

Docker全攻略 / 张涛编著.—北京：电子工业出版社，2016.4

ISBN 978-7-121-28238-6

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

中国版本图书馆CIP数据核字（2016）第042573号

责任编辑：安 娜

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：787×1092 1/16 印张：33.75 字数：952千字

版 次：2016年4月第1版

印 次：2016年4月第1次印刷

印 数：3000册 定价：89.00元

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

电子书寻找看手相 钉钉或微信pythontesting

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

服务热线：（010）88258888。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

内容简介

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

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

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

电子书寻找看手相 钉钉或微信pythontesting

前言

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

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

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

但随着Docker的不断发展和完善，我们真真切切地感受到了Docker是如何从一个鲜有耳闻的名词变成了虚拟化首选工具。基于Docker的云平台解决方案如雨后春笋般涌现，基于Docker的中国本地化解决方案也

开始逐步出现，各类国内镜像加速器和仓库也开始层出不穷。

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

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

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

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

第三部分包括第10章至第第15章。第三部分深入讲解了Docker内核机制。第10章，第11章和第12章分别从Docker生命周期、namespace和资源管理等方面入手，完完整整地讲解了Docker的内核运行方式。第13章、第14章和第15章则通过基于Amazon和Coreos的实际部署案例介绍

了Docker生态圈。

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

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

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

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

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

张涛

2016年2月21日

北京

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

第一篇　Docker简介

本篇包括

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

电子书寻找看手相 钉钉或微信pythontesting

第1章　Docker的前世今生

1.1　什么是LXC

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

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

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

Namespaces则是另外一个重要的资源隔离机制。Namespaces将进程、进程组、IPC、网络、内存等资源都变得不再是全局性资源，而是将这些资源从内核层面属于某个特定的Namespace。在不同的Namespace之间，这些资源是相互透明、不可见的。比如说，A用户登

录系统后，可以查看到B用户的进程PID。虽说A用户不能杀死B用户的进程，但A和B却能相互感知。但假如A用户在Namespace-A中，B用户在Namespace-B中，虽然A和B仍然共存于同一个Linux操作系统当中，但A却无法感知到B。在这种情况下，Linux内核不但将Namespace相互隔离，而且将所分配的资源牢牢固定在各自空间之中。

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

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

1.2 Docker为什么选择了AUFS

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

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

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

这句话不长，但理解起来颇需一些脑力。没关系，我们慢慢来分析这句话。

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

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

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

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

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

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

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

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

1.3 Docker是如何产生的

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

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

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

山穷水尽疑无路，柳暗花明又一村。

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

哇！开发界，测试界和生产环境三界统一了！

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

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

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

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

第2章　Docker现状

2.1　Docker应用范围

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

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

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

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

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

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

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

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

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

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

综上所述，结合目前Docker在实际环境之中的运用，如果一个企业需要将产品发布在云平台之上，同时也想提高开发效率，最好还能方便

管理产品，那么使用Docker绝对是一个好的选择。

2.2 Docker的优缺点

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

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

1. Docker资源利用率比传统虚拟机要高

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

2. Docker支持跨节点部署

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

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

3. 版本可控，组件可复用

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

4. 共享镜像

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

5. 轻量，易维护

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

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

电子书寻找看手相 钉钉或微信pythontesting

1. 宿主资源没有完成做到隔离

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

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

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

2. Golang语言尚未成熟

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

3. Docker虽已开源，但事实上被一家公司所控制

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

目前dotCloud公司已经推出了面向企业用户的咨询、支持和服务这

些企业收费服务，所以这种担心看来也不是杞人忧天。

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

第3章　Docker将如何改变未来

Docker将“云”由虚幻变得有些现实

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

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

而这也是Docker风靡世界、炙手可热的一个原因。

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

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

然改变云平台的基础架构。

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

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

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

Docker虽然前景广阔，但除了Google、AWS等有雄厚的技术实力可以保障大规模Docker集群运行之外，市面上还缺乏大规模运行和维护Docker集群的经验。与此同时，也缺乏跨平台管理Docker的工具。同时

Docker自身也处于快速生长期，无论是在功能性还是在稳定性上面，离大家的期望值都还有一些距离。目前的Docker正如幼年的Java一样，虽然年轻，但却已经得到了大家的一致认可，只是缺乏历练和经验。

书籍下载qq群6089740 钉钉群21734177
IT书籍 http://t.cn/RDIAj5D

第二篇　Docker基础

本篇包括

- 第4章　如何安装Docker
- 第5章　Docker基本命令
- 第6章　Docker命令剖析
- 第7章　Dockerfile介绍
- 第8章　Dockerfile最佳实践
- 第9章　Docker部署案例

电子书寻找看手相 钉钉或微信pythontesting

第4章　如何安装Docker

4.1　安装Docker前需要知道的事

安装Docker是一件很容易的事情，但容易不代表随时随地就可以安装。下面是在安装Docker之前需要了解的注意事项。

1. Docker只能运行在**Linux**环境中

虽然我们可以在Windows和Mac操作系统中安装和使用Docker，但截止到目前，Docker仍然只能运行在Linux内核中。这一限制来源于LXC技术，因为Linux是开源操作系统，所以我们可以得知LXC机制，从而抽象出Docker。反观Windows和Mac，这两大操作系统仍然尚未开源，而是否支持LXC，我们同样也不得而知。所以现阶段Docker只能运行在Linux内核中，或许在不久的未来Docker能原生态的出现在Windows和Mac系统中，但目前还无法做到这一点。

2. Docker可以在**Windows**和**Mac**系统中使用

上面我们提到Docker仅支持Linux操作系统，但不意味着Docker不能在Windows和Mac上面使用。在开源社区中，有一个产品变相地解决了这个问题，就是boot2Docker。使用这个产品，可以在Windows和Mac上面运行Docker（暂且我们可以先理解成Docker可以在Windows和Mac上面运行，实际情况却并非如此），boot2Docker的安装使用及其原理，我们会在后面的章节介绍。

3. 选择一个主机操作系统很重要

这是什么意思呢？我们知道Linux内核版本相对固定，但Linux操作系统就多了，如Ubuntu、CentOS、Fedora、RedHat，等等。这些操作系统可能使用的是同一个版本的Linux内核，但在内核之外，又各有千秋。

Docker官方建议使用Ubuntu操作系统作为宿主系统，应该是看重了Ubuntu默认支持AUFS文件系统的缘故。其他系统同样可以运行Docker，只不过需做一些特殊配置。

4. 准备一个大硬盘

以现在的硬盘发展速度而言，500G、1T基本是电脑的标配。如果这些硬盘空间都分配给Docker，那肯定是没有问题的。但现实情况，很可能读者会使用一个VMWare或Virtual box这类的虚拟软件，虚拟出一个Linux来运行Docker，如果是这样的话，就提前建议读者多给这个虚拟机分配些空间。如果读者用一台装有Linux的实体机来运行Docker，那么就建议读者多给根目录分配一些空间。

究其原因，是因为Docker默认使用AUFS文件系统来存储数据，这个文件系统会额外的产生很多文件来维护文件分层信息，因此其保存的数据量会很大。比如说我们看到的image可能仅有1个G，但为了维护这个image的数据分层信息，AUFS内部产生的数据量可能会有2G，或者3G。所以准备个大硬盘，有备无患。

5. 读者应该具备一些**Linux**操作知识

本书中，我们先假定读者是具有一定Linux操作知识的人员。

无论是准备进行Docker应用开发的中高级开发人员，还是可能会用到Docker技术的运维人员，都应具有一定的Linux基础知识。例如可以熟练使用Linux常用命令，可以熟练编写Shell脚本，有一定的代码阅读功底，能编写或者看懂makefile文件等。

本书的侧重点是讲解Docker的使用，以及Docker的运行机制。所以不会用很大的篇幅来介绍Linux的基础知识，这些知识在阅读本书之前应该有一些了解。至于代码的阅读功底，如果是读者是运维人员，那么不做强求了解即可。但因为Docker是使用Golang语言开发的，所以如果读者是开发人员，有Golang开发使用经验将是最好的；若没有Golang使用经验的话，那么建议至少有看懂C/C++的能力，有使用面向对象语言（例如Java）的经验。

4.2 Linux平台安装Docker

本节将介绍如何在Linux平台上安装Docker。我们假定主机操作系统是Ubuntu，所以后续所有操作都将默认在Ubuntu系统上面执行。而在本节的最后，我们将简要介绍一下如何在其他Linux系统中安装Docker。

1. 安装**Docker**

在安装之前，需要确认当前Ubuntu版本。Docker目前支持Ubuntu以下版本：

- Ubuntu Wily 15.10
- Ubuntu Vivid 15.04

- Ubuntu Trusty 14.04（TLS）
- Ubuntu Precise 12.04（TLS）
- Ubuntu Saucy 13.10

同时Docker要求必须为64位操作系统，同时Linux内核版本最小为3.10。（3.10以下的内核存在一些bug，在某些特定场景中有可能会引起容器中的数据丢失）

如果读者不确定当前所使用的操作系统版本号，那么可以通过下面的方式查询：

```
在终端中执行：cat /etc/issue
或者
在终端中执行：sudo lsb_release -a（执行此命令要求具有管理员权限）
如何查询当前Linux内核版本号？在终端中执行uname -r即可看到内核版本号。
```

在上述几个Ubuntu版本中，我们建议使用Ubuntu Trusty 14.04. 因为这个版本是目前最为稳定的版本，其中Docker运行所需的各种动态库和内核也最为完整，即在这个版本上面安装Docker，不需要任何其他多余的步骤，安装最为方便简单。

①首先登录一个具有sudo权限的账户，建议使用root创建一个Docker用户，并赋予其sudo权限。

②执行下面的命令，检查是否安装wget工具。

```
$ which wget
```

如果安装了wget，那么将看到wget的路径。反之，则会提示没有wget。如果没有安装wget，则执行下面的命令安装wget：

```
$ sudo apt-get update
$ sudo apt-get install wget
```

③安装完wget之后，我们就可以使用wget来获取Docker安装包了：

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

执行完上面的命令，系统就会下载Docker安装包及其依赖包，并自动安装。

④当安装完成后，执行下面的命令可验证是否安装成功：

```
$ sudo Docker run hello-world
```

执行这条命令后，如果Docker安装成功，将看到屏幕输出如下信息：

```
Hello from Docker.
This message shows that your installation appears to be working c
……
```

这就表示Docker已经安装成功，后面就可以开始Docker使用之旅了。

2. 更新**Docker**

Docker版本迭代速度很快，如果发现有新的Docker版本了，并且想尝试新的Docker功能时，可以执行下面的命令升级Docker：

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

3. 卸载**Docker**

当你不再需要Docker的时候，可以执行下面的命令卸载Docker。

①卸载Docker安装包：

```
$ sudo apt-get purge lxc-Docker
```

②卸载Docker安装包和Docker所有的依赖模块：

```
$ sudo apt-get autoremove --purge lxc-Docker
```

以上两条命令，仅仅是删除了安装包和依赖包，Docker所存储的image、container以及Docker所创建的配置文件，都还保留在系统目录下。执行下面的命令可以彻底清除Docker所保留的数据：

```
$ rm -rf /var/lib/Docker
```

> 注意
>
> 执行rm操作时一定要谨慎，千万要确认后面的目录路径是否正确！

4. 配置**Docker**

这部分内容属于Docker高级安装选项了，通常情况下Docker默认的参数就可以满足使用，如果读者想对Docker进行定制化安装配置，那么建议参阅以下内容。

①创建Docker组

默认情况下，Docker　Daemon会监听本地的一个Socket文件。而这个socket文件通常是由root用户来创建的，其他用户对其并没有读写权限。因而在执行Docker命令时，就需要使用sudo来进行操作。也正因为

这个原因，Docker Daemon需要使用root用户来启动。

为了回避这个问题，建议读者创建一个Docker group，然后将Docker用户添加到这个group中。这样再启动Docker Daemon时，就可以回避掉socket文件没有读写权限的问题。

下面的命令执行后，就会创建一个Docker组，并且将指定用户添加到组内。

```
$ groupadd Docker
$ usermod -aG Docker Ubuntu #ubuntu 表示的是用户名
```

②调整内存和SWAP参数

在使用Docker过程中，可能会出现下面的警告：

```
WARNING: Your kernel does not support cgroup swap limit. WARNING
kernel does not support swap limit capabilities. Limitation disca
```

启用系统中内存和SWAP统计功能后，就可以解决这个问题。

使用GNU GRUB（GNU Grand Unified Bootloader）来启用这项功能的步骤如下。

1）首先使用具有系统管理员权限的用户登录到系统中；

2）编辑/etc/default/grub文件；

3）修改GRUB_CMDLINE_LINUX参数：

```
GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

4）保存并退出编辑；

5）执行sudo update-grub来更新grub；

6）重启系统

③调整UFW(Ubuntu Fire Wall)参数

如果你的Ubuntu系统启用了UFW，那么就需要单独配置一下Fire Wall的参数。Docker默认使用桥接转发方式来管理container之间的网络连接。在默认情况下，UFW是拒绝所有转发请求的，因此如果你想从另外一个主机访问到这台主机上面的容器，那么就需要UFW允许外来的网络请求访问到Docker Daemon服务端口（此端口默认是2375）。

按照以下步骤可以调整UFW参数：

1）首先使用具有系统管理员权限的用户登录到系统中；

2）执行$sudo ufw status，确认UFW已经安装并且启用；

3）编辑/etc/default/ufw文件；

4）修改DEFAULT_FROWARD_POLICY参数：

```
DEFAULT_FORWARD_POLICY="ACCEPT"
```

5）保存并且退出；

6）重新加载UFW；

```
$ sudo ufw reload
```

7）允许2375端口可以被外网访问：

```
$ sudo ufw allow 2375/tcp
```

4.3 MAC/OS平台安装Docker

在4.1节中，我们提到过Docker只能运行在Linux平台中，无法直接运行在MacOS平台。如果想要在Mac OS平台运行Docker，就需要借助一个工具—Boot2Docker。

通过Boot2Docker，用户即可直接在终端运行Docker的各项命令，就如同Docker直接安装在Mac OS上面一样。如果读者想对Boot2Docker进行深入研究，可以在Github（https://github.com/boot2docker）下载Boot2Docker工程。但Boot2Docker仅仅是一个在Mac OS中用来开发、测试Docker的小工具，所以不建议将其部署在产品生产环境中。

在安装Boot2Docker之前，需要确认当前版本是否为OS X 10.6版本或者之上。如果在10.6版本之下，请自行升级至10.6。

完成版本确认工作之后，我们首先简要介绍一下Boot2Docker。

Boot2Docker是一个包含VirtualBox虚拟软件、Docker程序和Boot2Docker Management Tool的一个集合软件。其中，Boot2Docker Management Tool是一个运行在Mac OS操作系统上面的轻量级Linux虚拟机。这个轻量级虚拟机完全运行在内存中，启动特别快（耗时大概5秒钟），同时体积也非常小（不超过24MB）。

使用Boot2Docker安装Docker和直接在Linux上面安装Docker有什么不同呢？

Docker运行时，分为Docker客户端和Docker服务端。如果是本地环境，那么Linux服务器既是Docker客户端也是Docker服务端；如果是网络环境，那么Docker客户端指的是你自己运行Docker命令的那台计算机环境，而Docker服务端指的是容器所运行的那台计算机环境。

在Docker默认的使用方式中，Docker客户端、Docker服务端和容器都是运行在同一个Linux环境中的，这就意味着你可以使用localhost:8000或者0.0.0.0:8376这样的方式访问Docker container，它们之间的联系图如图4-1所示。

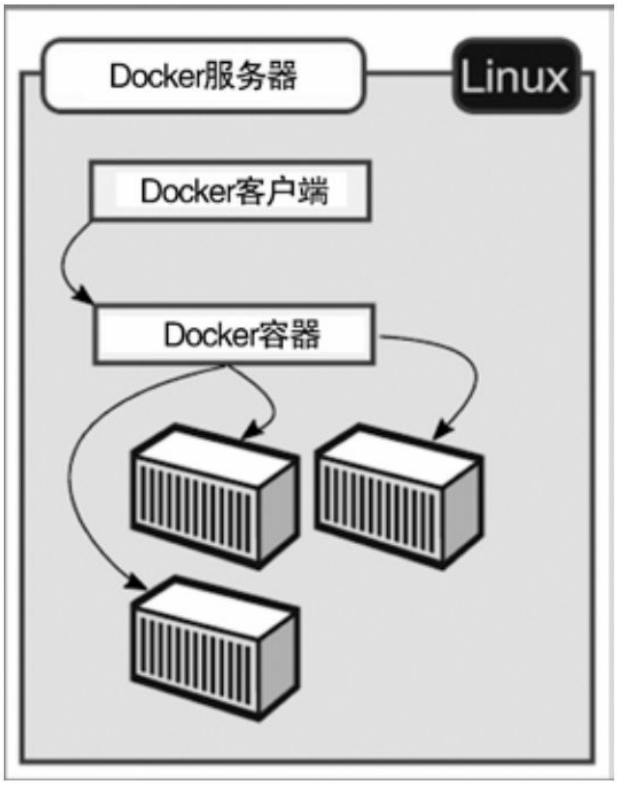


图4-1 联系图

而通过Boot2Docker的方式安装Docker时，Docker就变为运行在一个VirtualBox虚拟机中，而Docker客户端则运行在Mac OS上，如图4-2所示。

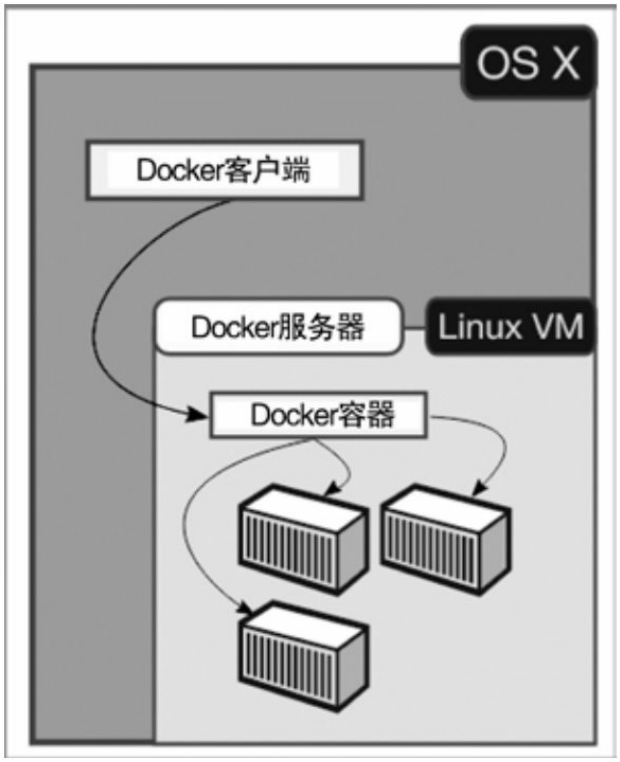


图4-2　通过Boot2Docker

这种情况下，Docker服务器的IP地址就变为虚拟机的IP地址。当启动Boot2Docker进程时，VM将会自动获取一个IP。Boot2Docker会自动将Docker命令路由到这个IP的虚拟机中（其中的技术实现，有兴趣的读者请参考Boot2Docker项目源码）。

在简单介绍完Boot2Docker之后，下面就开始实际安装Boot2Docker。安装过程非常简单，只需要三步即可。

①单击https://github.com/boot2docker/osx-installer/releases，下载Mac版本的Boot2Docker安装包。

②在Downloads页面中选择合适版本的安装包。建议下载最新版本。

③双击pkg安装文件，即可完成安装。

安装完成之后，Boot2Docker和VirtualBox将会添加到Applications目录中，应用程序目录默认存在于/usr/local/bin目录。

现在是不是迫不及待地想创建一个Docker容器呢？下面就开始创建一个容器吧。

若想创建Docker容器，首先必须要运行boot2Docker程序来创建一个VM，然后才能执行Docker命令来创建、加载和管理容器。你可以从Applications中运行Boot2Docker，同样也可以从命令行中运行Boot2Docker。

如果选择从Applications中运行Boot2Docker，那么将看到Boot2Docker完成了以下工作：

①打开一个终端窗口；

②创建$HOME/.boot2Docker目录；

③创建VirtualBox ISO和密钥证书（certs）；

④启动了运行Docker Daemon的虚拟机。

当以上4步创建完成之后，你就可以直接再Mac的终端窗口中执行Docker命令了。现在读者可以运行经典的“hello world”容器了。

执行下面的命令：

```
$ Docker run hello-world
```

然后，如果你看到下面的输出，那么就表示Boot2Docker在Mac安装成功了。

```
$ Docker run hello-world
Unable to find image 'hello-world:latest' locally
……

For more examples and ideas, visit:
http://docs.Docker.com/userguide/
```

如果需要从命令行执行程序，那么可以按照下面的步骤从命令行启动Boot2Docker。

①首先创建一个Boot2Docker虚拟机。这个命令会初始化一个全新的虚拟机，所以这个命令仅需执行一次：

```
$ boot2Docker init
```

②启动Boot2Docker虚拟机：

```
$ boot2Docker start
```

③显示Boot2Docker客户端的环境变量：

```
$ boot2Docker shellinit
Writing /Users/mary/.boot2Docker/certs/boot2Docker-vm/ca.pem
Writing /Users/mary/.boot2Docker/certs/boot2Docker-vm/cert.pem
Writing /Users/mary/.boot2Docker/certs/boot2Docker-vm/key.pem
    export Docker_HOST=tcp://192.168.73.12:2376
    export Docker_CERT_PATH=/Users/mary/.boot2Docker/certs/boot2D
    export Docker_TLS_VERIFY=1
```

有些环境变量数据会因为用户环境不同而有所差异。

④设置刚才所看到的环境变量：

```
$ eval "$(boot2Docker shellinit)"
```

也可以手工执行第三条命令所回显出的export命令。

⑤执行hello-world命令，如果正常，将看到如下输出：

```
$ Docker run hello-world
Unable to find image 'hello-world:latest' locally
……
```

```
For more examples and ideas, visit:
http://docs.Docker.com/userguide/
```

建议读者在运行Boot2Docker之前，最好确认Docker客户端的环境变量已经初始化。通过下面的命令，可以确认是否成功初始化。

```
$ boot2Docker status
$ Docker version
#如果可以看到Docker版本，就说明环境变量成功初始化。
```

下面我们再介绍一些Boot2Docker的基本用法。

1. 访问**container**网络端口

a）我们假设需要使用Docker创建一个Nginx服务器，那么执行下面的命令：

```
$ Docker run -d -P --name web nginx
```

这里请直接执行上面的命令，其中的参数我们现在不需要关心。在后面的章节，将会详细讲解这些参数的使用。

b）使用Docker ps命令显示新创建的Nginx容器的详细信息：

```
CONTAINER ID  IMAGE  COMMAND      CREATED STATUS PORTS NAMES
5fb65ff765e9      nginx:latest    "nginx  -g  'daemon  of     3
0.0.0.0:49156->443/tcp, 0.0.0.0:49157->80/tcp   web
```

c）获取以下容器的端口信息：

```
$ Docker port web
443/tcp -> 0.0.0.0:49156
80/tcp -> 0.0.0.0:49157
```

后面的信息表示，本地的49156端口映射容器的443端口，本地的49157端口映射容器的80端口。如果需要访问容器的80端口，那么就可以通过访问本地的49157端口实现。

我们通过49157端口来访问Nginx容器，如图4-3所示。

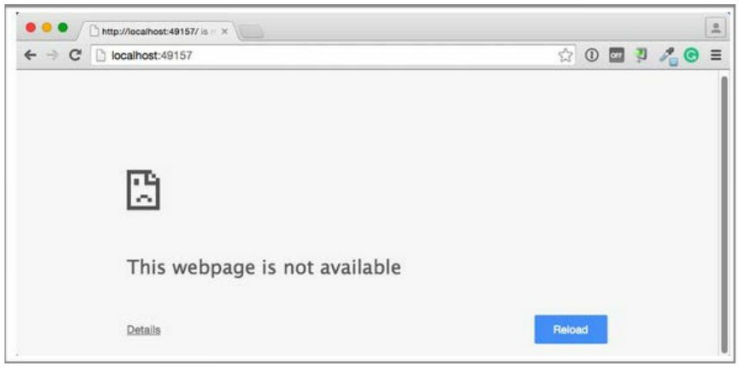


图4-3 访问Nginx容器

访问失败！这是因为虚拟机的49157和nginx容器的80相互映射。所

以我们需要访问http://虚拟机IP:49157。

d）下面是获取虚拟机IP的命令：

```
$ boot2Docker ip
192.168.73.12
```

重新访问Nginx容器，如图4-4所示。



图4-4　重新访问Nginx容器

2. 向**container**挂载目录

当用户启动Boot2Docker时，它会自动将/Users目录挂载到虚拟机中，而我们可以利用这一点挂载更多的目录到虚拟机中。下面就介绍一下如何挂载多个目录到虚拟机中。

a）切换到用户HOME目录：

```
$ cd $HOME
```

b）创建一个site目录：

```
$ mkdir site
```

c）进入site目录：

```
$ cd site
```

d）在site目录中，创建一个index.html文件：

```
$ echo "my new site" > index.html
```

e）创建一个Nginx服务器，并且使用新创建的index.html文件替换默认的index.html文件：

```
$ Docker run -d -P -v $HOME/site:/usr/share/nginx/html --name mys
```

f）获取刚才创建的mysite服务器的端口信息：

```
$ Docker port mysite
80/tcp -> 0.0.0.0:49166
```

```
443/tcp -> 0.0.0.0:49165
```

> 注意
>
> 主机端口，可能会因为个人的环境而有所差异。

g）打开浏览器，访问Ngnix服务器，如图4-5所示。

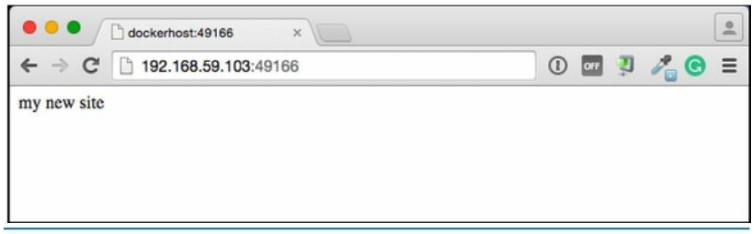


图4-5　访问Ngnix服务器

h）我们在site目录中再次创建一个cool.html文件：

```
$ echo "This is cool" > cool.html
```

i）我们在浏览器中访问新创建的cool.html文件，如图4-6所示。



图4-6　访问cool.html文件

j）结束我们的实验，清除资源：

```
$ Docker stop mysite
$ Docker rm mysite
```

3. 升级**Boot2Docker**

当Boot2Docker推出新版本后，你可以通过command line方式进行升级，也可以通过installer方式进行升级。

使用command line方式进行升级时，方法如下。

a）打开一个终端窗口。

b）停止boot2Docker应用程序：

```
$ boot2Docker stop
```

c）执行升级命令：

```
$ boot2Docker upgrade
```

使用installer方式进行升级时，方法如下。

a）同样打开一个终端窗口。

b）停止boot2Docker应用程序：

```
$ boot2Docker stop
```

c）在https://github.com/boot2Docker/osx-installer/releases页面中下载最新版本。

d）双击下载的pkg文件，进行安装。

4. 卸载**Boot2Docker**

卸载相对于安装来说，有些复杂。

a）首先需要到https://github.com/boot2docker/osx-installer/releases点击Source code来下载源码，zip和tar.gz都可以。

b）解压源码包到任意目录。

c）在本地打开一个终端窗口。

d）切换到源码包所在的解压目录。

```
$ cd <path to extracted source code>
```

e）给uninstall.sh赋予可执行权限。

```
$ chmod +x uninstall.sh
```

f）执行uninstall.sh脚本进行卸载：

```
$ ./uninstall.sh
```

4.4 Windows平台如何支持Docker

前面我们介绍了如何在Linux平台和Mac OS平台上面安装使用Docker，本节我们介绍如何在Windows平台安装Docker。其实有了前面几节的介绍，相信你也感觉到了Docker团队为我们做的已经挺人性化了，只要前提条件准备好，那么安装就是一件非常容易的事情。

我们言归正传，介绍如何在Windows平台中安装Docker。在正式安装之前，我们照例先说明一下安装的前提条件。

目前Docker已经可以完美地在Windows 7和Windows 8.1 64位操作系统中运行，建议读者也在这两个版本的操作系统上面进行实验，当然理论上只要读者机器的CPU支持硬件虚拟化，同时Windows版本在XP以上，那么都可以运行Docker。

正如前面所言，为了在Windows平台运行Docker，我们也需要Boot2Docker这个工具。Boot2Docker在Windows平台上面也会虚拟出一个运行Docker服务端的虚拟机，然后运行在Windows平台的Docker客户端与虚拟机中的Docker服务端完成交互，结构图如图4-7所示。

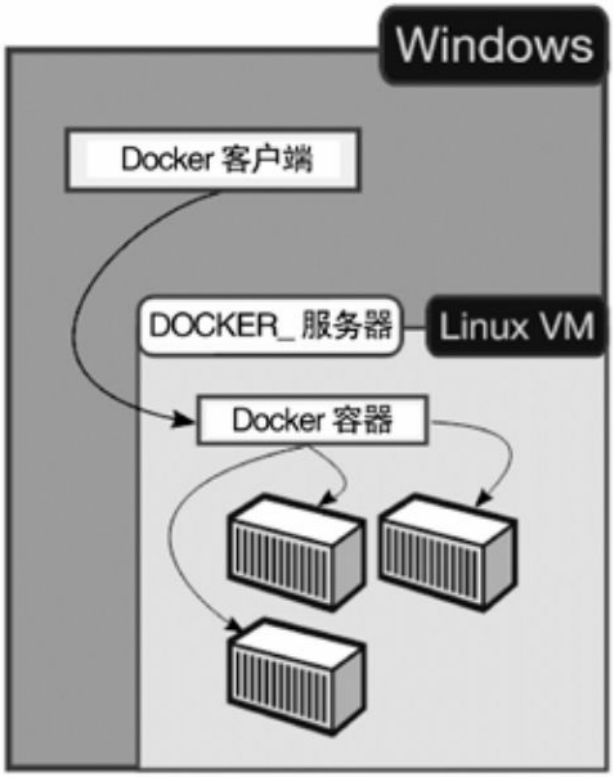


图4-7　结构图

现在开始安装步骤如下。

①首先请读者在https://github.com/boot2docker/windows-installer/releases中下载最新版本的Boot2Docker安装包。

②双击exe安装包，Boot2Docker会自动安装Docker client For Windows、VirtualBox、Git for Windows（MSYS-git），还有boot2Docker的Linux版本的iso镜像，最后就是Boot2Docker management tool。

③安装过程很简单，如果都采用默认参数的话，一直单击Next按钮就可以了，如图4-8所示。



图4-8　安装Boot2Docker

④安装完成后，请手动添加SSH到环境变量PATH中。

⑤添加环境变量后，从桌面双击Boot2Docker　Start这个快捷方式，将自动启动VirtualBox，并且完成初始化，同时也会生成certs。

⑥以上步骤都完成之后，就可以在打开的CMD窗口内执行Docker version来确认Docker是否启动成功了，如图4-9所示。

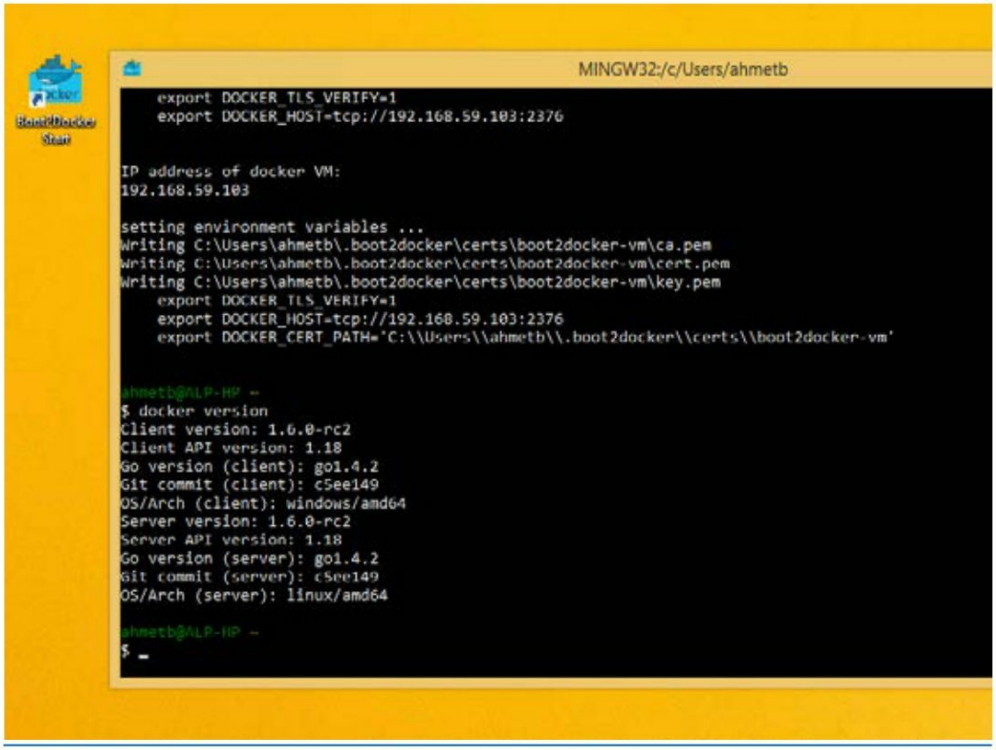


图4-9　确认Docker是否启动成功

下面我们执行一个经典的hello-world来看看Windows版本的Docker吧：

```
$ Docker run hello-world
```

如果能看到下面的信息，就表示hello-world container创建成功了。

```
Hello from Docker.
This message shows that your installation appears to be working c
……
```

```
For more examples and ideas, visit:
 http://docs.Docker.com/userguide/
```

因为Docker client没有GUI，所以它所有的执行都是从命令行发起的。下面介绍一下如何在CMD和PowerShell上面使用Boot2Docker。

首先看一下CMD里面如何使用Boot2Docker。Boot2Docker需要通过ssh这个工具来登录到虚拟机中，所以第一步就要在当前CMD窗口环境中，将SSH工具添加到PATH中。

执行下面命令，添加PATH变量：

```
set PATH=%PATH%;"c:\Program Files (x86)\Git\bin"
```

这样，我们就可以在当前CMD窗口内通过执行Boot2Docker start来启动虚拟机了。如果执行命令后，Boot2Docker提示当前还没有创建虚拟机的话，就执行Boot2Docker init来新创建一个虚拟机。

Boot2Docker start执行完之后，将看到当前环境变量。将set开头的环境变量放在当前cmd窗口中执行，然后就可以执行所有的Docker命令了。

上面介绍的是CMD中如何使用Boot2Docker，下面我们看看如何在PowerShell中使用Boot2Docker。

同样第一步需要把SSH添加到PATH当中去，只不过这时执行的命令和上面的命令不太一样，这次的命令如下：

```
$Env:Path = "${Env:Path};c:\Program Files (x86)\Git\bin"
```

然后就可以依次执行Boot2Docker　init和Boot2Docker　start了。最后，你将看到在当前窗口仍然提示应该设定的环境变量，此时，你需要把所有以$env开头的环境变量在当前窗口中执行一次，然后就可以使用Docker的所有命令了。

特别补充一下：如何在Windows平台使用putty登录到Docker虚拟机中。如果想用putty登录到Docker虚拟机，那么就需要使用Boot2Docker生成的certs。

首先我们将certs中的私钥证书转换成putty可以识别的证书格式：

①打开puttygen.exe，然后加载%USERPROFILE%\.ssh\id_boot2Docker。%USERPOFILE%，当前用户默认的HOME目录，一般在C:/User/下面。

②单击Save Private Key保存新生成的key。

③putty使用刚才保存的private　key登录Docker@127.0.0.1:2022，这样就可以登录到运行Docker服务端的虚拟机了。

最后看看如何卸载Boot2Docker。在Mac平台需要一个卸载脚本才可以卸载。在Windows平台卸载就很简单了，读者可以在Windows的程序管理器中查看Boot2Docker，这样就可以单击卸载了。但需要注意的是，Docker-install.exe不会被删掉，需要用户自行手动删除。

4.5 在其他平台中安装Docker

Docker主要运行在Ubuntu平台中，但在其他Linux系统中也可以安装使用。下面我们简要介绍一下如何在CentOS、Debian、Fedora和RedHat系统中安装Docker。

1. CentOS

如果想在CentOS系统上面安装Docker，那么推荐用CentOS 7.X版本。而CentsOS 6.5或者更低在满足下面条件时也可以运行Docker。

Docker运行需要64位操作系统，同时需要不低于3.10的Linux内核版本。CentOS 7.X内核版本是3.10，所以CentOS 7.X 64位操作系统可以运行Docker。而CentOS 6.5并没有使用3.10的Linux内核，因此在CentOS 6.5上面运行Docker时，需要确认内核版本不低于2.6.3-431。

可以用uname–r命令来显示当前系统的Linux内核版本。

如果需要在CentOS 6.X中安装Docker，则为了在安装和使用Docker过程中不出现错误，建议在安装之前首先安装所有的系统更新包，这里顺便提一句：尽量将目前已有的bug全部修复是日常使用操作系统的好习惯。

在CentOS 6.X和7.X上面安装Docker的步骤相同，只需要下载正确的Docker安装包。不同版本的CentOS对应不同的安装包，请大家在下载时特别留意一下，具体如表4-1所示。

表4-1 不同版本的LentOS对应不同版本的安装包

CentOS版本号	**Docker**安装包名称
6.5或者更高	https://get.docker.com/rpm/1.7.0/centos-6/RPMS/x86_64/docker-engine-1.7.0-1.el6.x86_64.rpm https://get.docker.com/rpm/1.7.0/centos-6/SRPMS/docker-engine-1.7.0-1.el6.src.rpm
7.X	https://get.docker.com/rpm/1.7.0/centos-7/RPMS/x86_64/docker-engine-1.7.0-1.el7.centos.x86_64.rpm https://get.docker.com/rpm/1.7.0/centos-7/SRPMS/docker-engine-1.7.0-1.el7.centos.src.rpm

好，我们现在开始安装吧。

首先使用一个有sudo或者root权限的用户登录到CentOS中执行下面命令更新操作系统：

```
$ sudo yum update
```

更新完系统后，就可以下载Docker安装包了，注意不要下载错版本。在本次安装中，我们以CentOS 6.5为例进行安装。

```
$ curl -O -sSL
https://get.Docker.com/rpm/1.7.0/centos-6/RPMS/x86_64/Docker-engi
```

使用yum命令安装Docker：

```
$ sudo yum localinstall --nogpgcheck
Docker-engine-1.7.0-1.el6.x86_64.rpm
```

如果读者本地的CentOS没有问题的话，那么这个安装过程将非常简单。在等待几分钟后，我们就可以看到安装完成，现在开始启动Docker Daemon：

```
$ sudo service Docker start
```

启动成功后，再使用经典的hello-world来验证一下Docker吧：

```
$ sudo Docker run hello-world
Unable to find image 'hello-world:latest' locally
……

For more examples and ideas, visit:
 http://docs.Docker.com/userguide/
```

看到这些信息，就表示Docker安装成功。

在CentOS系统中，同样建议读者创建一个Docker group，然后将Docker用户添加到Docker组中，这样可以省略使用sudo执行Docker命令。关于如何创建Docker group，以及如何将Docker添加到group中，读

者可以参考4.2节中的相关介绍，这里不再赘述。

如果希望Docker Daemon能够随系统启动而启动，那么可以通过执行：

```
$ sudo chkconfig Docker on
```

将Docker Daemon添加到系统初始化列表中。当系统在执行应用程序初始化时，将Docker Daemon作为守护进程而启动起来。

最后，我们简要介绍一下如何在CentOS中卸载Docker。卸载过程简单且方便，首先找出Docker rpm名称：

```
$ yum list installed | grep Docker
yum list installed | grep Docker
Docker-engine.x86_64                1.7.0-1.el6
```

然后使用yum执行删除即可：

```
$ sudo yum -y remove Docker-engine.x86_64
```

执行完毕之后，Docker应用程序就会被删掉，但Docker所创建的容器信息、所存储的镜像和用户其他配置信息并不会被删掉，这些仍然需要手动删除。默认情况下，删除/var/lib/Docker目录即可。如果读者自行

将存储目录修改为其他目录，那么删除相对应的存储目录即可。

```
$ rm -rf /var/lib/Docker
```

以上就是如何在CentOS 6.X和7.X中安装、使用和卸载Docker。下面我们看一下如何在Debian中安装Docker。

2. Debian

Docker目前仅支持在Debian 8.0 Jessie 64bit和Debian 7.7 Wheezy 64bit中安装使用，所以读者需要确认当前Debian版本是否符合要求。

在Debian系统中安装Docker和在Ubuntu系统中安装较为类似，但Debian 8.0和Debian 7.7因为内核版本的问题，安装步骤并不一样。下面我们一一介绍如何在8.0和7.7上面安装Docker。

Debian 8.0使用的是Linux 3.10内核版本，其中jessie-backports资料库已经内置了Docker.io安装包。由于历史原因，Debian中存在一个名为Docker的比较旧的KED3/GNOME2软件包，所以我们需要使用的Docker在debian中就改名为Docker.io了。

安装相对就简单多了，直接执行命令进行安装：

```
$ sudo apt-get update
$ sudo apt-get install Docker.io
```

apt-get install执行完之后，我们就可以使用hello-world进行验证了。

```
$ sudo Docker run --rm hello-world
```

这里的--rm表示当hello-world容器退出时，Docker会将此容器的相关信息删掉。关于run相关命令参数，本书后面章节会详细讲解。

上面是在Debian 8.0中安装Docker，下面简单提一下如何卸载Docker。

在Debian中卸载Docker只需要一条命令，如果只需卸载Docker应用程序，那么执行：

```
$ sudo apt-get purge Docker-io
```

如果想把应用程序和依赖一并卸载掉，那么执行：

```
$ sudo apt-get autoremove --purge Docker-io
```

和在其他系统卸载Docker一样，上述两条命令仅仅是把Docker应用程序卸载掉了，如果读者想把Docker所保存的数据都删掉，那么执行：

```
$ rm -rf /var/lib/Docker
```

3. Debian 7.7

Docker在Debian系统上面运行，最低要求是Linux 3.8的内核版本。而Debian 7.7大多使用的是Linux 3.2内核。所以在Debian 7.7上面安装Docker，首先需要更新内核版本。

我们将wheezy-backports添加到源中。在/etc/apt/sources.list文件中添加下面一行：

```
deb http://http.debian.net/debian wheezy-backports main
```

然后安装新内核：

```
$ sudo apt-get update

$ sudo apt-get install -t wheezy-backports  linux-image-amd64
```

> 注意
>
> 要记得使用-t wheezy-backports。

安装完成后，重启你的Debian系统，以便使用新安装的内核。后面就很简单了，使用下面的命令，直接安装：

```
curl -sSL https://get.Docker.com/ | sh
```

在Debian 7.7中卸载Docker和在Debian 8.0中卸载过程完全相同，此处不再赘述。

最后，我们附加一个如何使用非root权限的用户操作Docker的步骤。Docker启动后需要绑定本地的UNIX Socket管道文件，所以Docker一般都是使用root权限来启动的。当操作Docker时，也都需要使用sudo来提升权限。

如果你创建了一个Docker group，并且将Docker添加到了Docker group中。那么当Docker Daemon启动时，会将绑定的UNIX Socket文件读写权限赋予Docker group。这样一来，虽然Docker是使用root权限启动的，但你在使用Docker用户操作它时，却不需要再使用sudo提升权限了。

下面是将Docker用户添加到Docker group和重启Docker的命令：

```
# Add the Docker group if it doesn't already exist.
$ sudo groupadd Docker

# Add the connected user "${USER}" to the Docker group.
# Change the user name to match your preferred user.
# You may have to logout and log back in again for
# this to take effect.
$ sudo gpasswd -a ${USER} Docker
```

```
# Restart the Docker daemon.
$ sudo service Docker restart
```

4. Fedora

Docker目前支持Fedora 20、Fedora 21和Fedora22三个版本的Fedora系统，读者在安装Docker之前，请先确认是否为上述三个版本之一。

Docker在Fedora系统中运行同样最低要求Linux 3.10版本的内核，读者可以使用uname–r命令来检查当前Fedora内核版本。如果内核版本过低，请先升级内核版本。

同CentOS一样，不同版本的Fedora使用不同的Docker安装包Fedora各版本所对应的Docker安装包名称如表4-2所示。

表4-2 Fedora各版本对应的Docker安装包

Fedora版本号	**Docker**安装包名称
Fedora 20	docker-engine-1.7.0-1.fc20.x86_64.rpm docker-engine-1.7.0-1.fc20.src.rpm
Fedora 21	docker-engine-1.7.0-1.fc21.x86_64.rpm docker-engine-1.7.0-1.fc21.src.rpm
Fedora 22	docker-engine-1.7.0-1.fc22.x86_64.rpm docker-engine-1.7.0-1.fc22.src.rpm

在下面的安装过程中，我们以Fedora 21为例进行讲解。如果读者使用的是其他版本，那么仅仅是安装包名称不同而已，步骤相同。

首先使用具有sudo或者root权限的用户登录到Fedora中，然后检查一下当前系统中是否有已安装的旧版本的Docker：

```
$ yum list installed | grep Docker
```

如果存在旧版本的Docker，那么需要使用yum–r remove<packagename>来移除旧Docker。

如果没有旧版本的话，则可以下载最新版本的Docker：

```
$ curl -O –sSL
 https://url_to_package/Docker-engine-1.7.0-0.1.fc21.x86_64.rpm
```

等下载完成后，就可以使用yum来安装了：

```
$ sudo yum localinstall –nogpgcheck
 Docker-engine-1.7.0-0.1.fc21.x86_64.rpm
```

安装过程正常情况下会非常顺利，按照惯例，我们启动Docker Daemon，然后使用hello-world来验证安装是否成功。

```
$ sudo service Docker start
$ sudo Docker run hello-world
```

```
……

For more examples and ideas, visit:
 http://docs.Docker.com/userguide/
```

在Fedora上面卸载Docker，和在Debian上面卸载Docker完全一致，这里不再赘述。

5. Red Hat

虽然RedHat市场占有率不错，但Docker目前也仅支持RedHat Enterprise Linux 6.6和RedHat Enterprise Linux 7或者更高级。读者在安装之前仍然需要确认操作系统版本。

Docker在Red Hat上面运行时，依赖于64位操作系统和最小为3.10的Linux内核版本，而Red Hat 7直接就包含3.10的Linux内核，所以在Red Hat 7上面安装运行Docker是没有问题的。而Red Hat 6.X没有使用3.10内核，为了在Red Hat 6上面运行Docker，经过多次测试，最好使用版本为2.6.32-431或者更高版本号的Linux内核。

读者可以使用uname–r来检查当前系统内核版本号。如果内核版本没有问题，就可以安装Docker了。

不同的RedHat版本同样对应不同的Docker安装包，对应关系如表4-3所示。

表4-3 不同的RedHat版本的Docker安装包

RedHat版本	**Docker**安装包名称

6.6 and higher	https://get.docker.com/rpm/1.7.0/centos-6/RPMS/x86_64/docker-engine-1.7.0-1.el6.x86_64.rpm https://get.docker.com/rpm/1.7.0/centos-6/SRPMS/docker-engine-1.7.0-1.el6.src.rpm
7.X	https://get.docker.com/rpm/1.7.0/centos-7/RPMS/x86_64/docker-engine-1.7.0-1.el7.centos.x86_64.rpm https://get.docker.com/rpm/1.7.0/centos-7/SRPMS/docker-engine-1.7.0-1.el7.centos.src.rpm

下面，我们用稍微麻烦点的RedHat 6.6来安装Docker，如果使用Red Hat 7的话，则步骤完全一样。

首先使用具有sudo或者root权限的用户登录到RedHat中，然后下载Docker安装包到当前目录：

```
$ curl -O –sSL
http://get.Docker.com/Docker/1.7.0/rpms/centos-6/RPMS/x86_64/Dock
```

下载完成后，使用yum来安装这个软件包：

```
$ sudo yum localinstall –nogpgcheck Docker-engine-1.7.0-0.1.el6.x
```

启动Docker Daemon，然后使用hello-world来验证Docker是否安装成

功：

```
$ sudo service Docker start
$ sudo Docker run hello-world
……

For more examples and ideas, visit:
 http://docs.Docker.com/userguide/
```

在RedHat上卸载Docker和在Debian、Fedora上卸载Docker完全相同，这里不再赘述。

第5章　Docker基本命令

5.1　Docker操作命令

Daemon命令

Daemon是Docker的服务端执行程序，也是Docker接受处理CL命令请求的模块，所以它的命令参数很多，使用非常灵活。

命令使用讲解如表5-1所示。

表5-1　命令使用讲解

命令	讲解
--api-cors-header=""	设定远程访问 API 所需的 cors-header
-b, --bridge=""	将容器网络访问方式设置为网络桥模式
--bip=""	设定网络桥 IP
-D, --debug=false	启动 Debug 模式
-d, --daemon=false	启动守护进程模式
--default-gateway=""	所有容器默认使用 IPv4 地址
--default-gateway-v6=""	所有容器默认使用 IPv6 地址
--dns=[]	使用指定的 DNS 服务器
--dns-search=[]	使用指定的 DNS 检索服务器
--default-ulimit=[]	设定容器默认的 ulimit 参数
-e, --exec-driver="native"	设定 Docker 使用的 exec-driver
--exec-opt=[]	设定 exec-driver 参数
--exec-root="/var/run/Docker"	设定 exec-driver 默认根目录地址
--fixed-cidr=""	设定 IPv4 子网段
--fixed-cidr-v6=""	设定 IPv6 子网段
-G, --group="Docker"	设定 UNIX Socket 文件用户所属组
-g, --graph="/var/lib/Docker"	设定 Docker runtime 时的根目录地址
-H, --host=[]	设定 UNIX Socket 文件监听位置
-h, --help=false	输出帮助信息
--icc=true	允许容器之间相互通信
--insecure-registry=[]	设定可信任仓库地址
--ip=0.0.0.0	设定绑定容器端口信息时的默认 IP
--ip-forward=true	允许转发 IPv4 数据包
--ip-masq=true	允许 IP 伪装

续表

命令	讲解
--iptables=true	允许修改 IPtables 数据
--ipv6=false	允许使用 IPv6 网络
-l, --log-level="info"	设定 log 输出级别
--label=[]	按照 Key=Value 格式设定 Daemon 标签
--log-driver="json-file"	设定日志记录方式
--log-opt=[]	设定日志记录驱动参数
--mtu=0	设定容器 MTU 值
-p, --pidfile="/var/run/Docker.pid"	设定 Docker.pid 文件路径
--registry-mirror=[]	设定优先访问的仓库镜像地址
-s, --storage-driver=""	设定数据存储驱动方式
--selinux-enabled=false	启用 selinux
--storage-opt=[]	设定数据存储驱动参数
--tls=false	启用 TLS 安全认证
--tlscacert="~/.Docker/ca.pem"	设定 CA 证书地址
--tlscert="~/.Docker/cert.pem"	设定 TLS 证书地址
--tlskey="~/.Docker/key.pem"	设定 TLS 密钥地址
--tlsverify=false	启用 TLS 来验证远程访问请求
--userland-proxy=true	启用指定的网络代理
-v, --version=false	输出 Docker 版本信息

如果参数后面有“[]”标记，例如“--dns=[]”，则表示此参数可以使用多次。

Daemon是Docker操作容器最重要的模块，它既是一个客户端，也是一个服务端。当Daemon以服务端形式执行时，需要在Docker后面添加“-d”参数（Docker–d）；如果需要Daemon输出执行其他CL命令过程所产生的调试信息，则需要在Docker后面添加“-D”参数（Docker–d–D）。

因为Daemon非常重要，所以下面将按照如下顺序重点讲解Daemon：

①Daemon网络配置

②Daemon存储驱动配置

③Daemon执行驱动配置

④Daemon DNS配置

⑤Daemon其他配置

1. Daemon网络配置

Docker通过Restful API的形式对外提供服务，Daemon是处理访问请求最核心的模块，因此Daemon必须实现对外监听功能。目前Daemon主要有三种途径来实现监听功能：

- 通过socket文件
- 通过tcp协议
- 通过fd文件

默认情况下，Daemon会通过socket文件的方式达到监听目的。这个socket文件默认存放在/var/run/Docker.socket。如果要监听这个文件，则用户需要具有root操作权限或者在Docker成员组内。

如果当前环境需要远程跨访问另外节点中的Docker Daemon，那么就需要Daemon使用tcp监听方式。此时用户需要注意，tcp监听方式分为两种：无加密访问和加密安全访问。

在默认情况下，2375是无安全加密访问的tcp端口，而2376是加密安全访问的tcp端口。如果用户需要启用https安全访问，那么就需要使用TLS 1.0或者更高版本的TLS支持。因为安全原因，Docker Daemon已经不再支持SSLV3或者更低的SSL协议了。

而监听fd文件则是Docker为使用systemd的操作系统特别准备的，在此类操作系统中使用Docker–d–H fd://，就可以启动监听功能。

在Daemon的命令参数列表中，可以看到“-H,—host=[]”参数，此参数表示-H可以指定多个网络地址。所以当我们需要监听多个网络地址时，可以通过下面的方式来完成：

```
Docker -d -H unix:///var/run/Docker.sock -H tcp://192.168.73.12 -
```

当Docker Daemon作为客户端使用时，也可以通过指定网络地址的方式来访问对应的Daemon。例如，本地地址为192.168.73.12，但我们需要访问192.168.73.12的Daemon，那么我们就可以通过Docker–H tcp://192.168.73.12:2376……来访问对方节点的Daemon。

如果不想每次都敲入对方的网络地址，那么也可以通过设置环境变量的方式来访问，例如：

```
$ export Docker_HOST=“tcp://192.168.73.12:2376"

$ Docker ps –a #显示59.103节点的Docker环境中的容器信息
```

那么当环境变量和命令行输入不一致时，哪个优先级更高呢？答案是命令行。例如：

```
$ export Docker_HOST="tcp://192.168.73.12:2376"
$ Docker ps –a
$ Docker –H 192.168.59.108:2376 ps –a
```

此时，Daemon作为客户端将会访问192.168.59.108节点中的Daemon。

如果对方节点中的Daemon启用了TLS安全认证。此时，需要使用—tlsverify来提交秘钥文件。通过设置环境变量也可以达到此效果，以下两种方式是等效的。

```
$ Docker --tlsverify ps
等效于
$ export Docker_TLS_VERIFY=1
$ Docker ps
```

2. Daemon存储驱动配置

Docker Daemon提供了五种存储驱动方式，分别是AUFS、Devicemapper、btrfs、zfs和overlay。

AUFS五种存储驱动中最老牌的驱动方式，但很可惜其一直作为Linux的分支使用，从来没有被纳入到Linux的主干版本当中。AUFS内部有一些可能会引起Linux内核崩溃的问题，因为这些问题一直没有很好的解决方案，所以这些问题在很大程度上影响了AUFS的推广使用。

但幸运的是，AUFS上面的问题并没有影响到Docker的使用，同时，AUFS允许Docker容器之间共享数据资源。因此，如果需要在主机中运行成百上千个容器，并且这些容器有可能会复用很多数据资源时，那么使用AUFS将会是一个非常好的选择。

Devicemapper可以使用非常少的前置条件就能达到写时复制（CoW）的效果。如果采用Devicemapper存储数据的话，那么Docker Daemon会基于两个块设备来创建devicemapper：一块用于存储数据，另一块用于存储元数据。

btrfs执行速度较快，尤其是在用Dockerfile构建image时。但与Devicemapper一样，btrfs不能在各容器之间提供数据共享功能。如果需要使用btrfs，则通过Docker–d–s btrfs–g/mnt/btrfs_partition来进行切换。

而zfs与btrfs相比，速度没有btrfs快，但提供了非常好的日志记录功能。借助于Single Copy ARC功能，各容器可以共享一部分块设备。如果需要切换到zfs，则通过Docker–d–s zfs来切换。

最后是overlay驱动，这个驱动是一种执行速度非常快的联合堆栈文件系统。而且是最近刚刚被加入到Linux内核主干版本（3.18.0）的功能。因为这种技术是最近刚刚发布，还没有大规模使用经验，而且根据目前已知情况，overlay随着image数量的增加，有可能会大量消耗文件链接数，因此不建议在生产环境中使用。如果用户需要使用，则通过Docker–d–s overlay可以切换到overlay驱动。

上述存储驱动，都可以通过--storage-opt+各驱动调整参数来进行设置。例如，devicemapper驱动的调整参数是以dm为前缀，Zfs驱动的调整参数是以zfs为前缀。

下面我们讲解一下其中的重点参数。

1）dm.thinpooldev：用户可以自定义一个块设备，用于建立devicemapper数据池。如果devicemapper使用块设备来存储数据，那么建议使用lvm来创建和管理数据池卷组。Docker将使用这个卷组来存储所有image和container的快照。

可以参考以下示例来使用dm.thinpooldev：

```
Docker -d --storage-opt dm.thinpooldev=/dev/mapper/thin-pool
```

2）dm.basesize：用户可以指定每个块设备大小，这个值将直接限制image和container的尺寸，默认值为10GB。需要特别注意的是：devicemapper所存储的数据是离散分布的，其存储的数据量如果为10GB，不代表其只使用了数据池中的10GB大小，而是会占用比10GB更多的空间。在使用pull操作时，Docker往往会初始化比image所显示尺寸更大的空间。

可以参考以下示例使用dm.basesize：

```
$ Docker -d --storage-opt dm.basesize=20G
```

3）dm.loopdatasize：此值将会影响到Docker的data回环文件，不建议在生产环境中修改。devicemapper可以使用这个文件来模拟一个文件系统。此文件默认大小为100GB，但因为此文件也是离散稀疏的，所以

实际文件尺寸将比100GB大。

可以使用下面示例来修改dm.loopdatasize：

```
$ Docker -d --storage-opt dm.loopdatasize=200G
```

4）dm.loopmetadatasize：此值将会影响到Docker的metadata回环文件，不建议在生产环境中修改。devicemapper可以使用这个文件来模拟一个文件系统。此文件默认大小为2GB，但因为此文件也是离散稀疏的，因此实际文件尺寸将比2GB大。

可以使用下面示例来修改dm.loopmetadatasize：

```
$ Docker -d --storage-opt dm.loopmetadatasize=4G
```

5）dm.fs：用户可指定devicemapper所使用的文件系统。目前支持的文件系统有“ext4”和“xfs”，默认为“ext4”。

可以通过下面示例来修改dm.fs：

```
$ Docker -d --storage-opt dm.fs=xfs
```

6）dm.mkfsarg：当在创建基本块设备时，可以使用这个参数来扩展mkfs的参数，例如：

```
$ Docker -d --storage-opt "dm.mkfsarg=-O has_journal"
```

7）dm.mountopt：当挂载额外的块设备时，可以用通过这个参数来扩展mount参数，例如：

```
$ Docker -d --storage-opt dm.mountopt=nodiscard
```

8）dm.blocksize：用户可指定块设备中的块大小，默认块大小为64KB。可使用下面命令进行调整：

```
$ Docker -d --storage-opt dm.blocksize=512K
```

9）dm.blkdiscard：用户可指定开启或关闭文件系统的回收功能，默认是开启。当删除image或者container时，devicemapper将回收删除的文件系统。如果选择关闭此功能，那么当删除container时，删除时间将缩短好几倍。可删除后的文件空间将不会返回给主机的文件操作系统。

可使用下面命令进行调整：

```
Docker -d --storage-opt dm.blkdiscard=false
```

10）dm.override_udev_sync_check：用户可以通过这个参数来覆盖

devicemapper和udev之间的同步检查标志位。为了确认当前Docker Daemon是否支持udev同步检查，可以参考以下命令来确认：

```
$ Docker info
[...]
Udev Sync Supported: true
[...]
```

只有当前环境支持udev同步检查时，才可使用此功能。可以通过下面命令进行调整：

```
$ Docker -d –storage-opt dm.override_udev_sync_check=true
```

11）zfs.fsname：用户可自行设定Docker所创建的数据集路径。默认情况下，Docker将会在/var/lib/Docker中创建zfs的数据集，可使用下面命令进行调整：

```
$ Docker -d -s zfs --storage-opt zfs.fsname=zroot/Docker
```

3. Daemon执行驱动配置

Docker默认将其自行开发的针对namespaces、cgroups和SELinux的接口包（native）作为虚拟化执行驱动，但Docker也支持使用lxc作为其虚拟化执行驱动。

但需要明确的是，Docker支持lxc仅仅是为了保持兼容性，有一些新功能和新特性，在lxc上面并不存在。所以将lxc作为虚拟化执行驱动，并不是开发人员的优先选择。

如果需要使用lxc，则可在Daemon启动时指定-e lxc，将执行驱动切换为LXC。

Docker允许用户使用--exec-opt来调整执行驱动的参数，所有执行驱动的参数都以native开头。但目前Docker仅推出了native.cgroupdriver这一个参数（希望Docker后续版本可以支持更多的参数）。

native.cgroupdriver参数用来指定容器组的管理驱动，目前可选项为cgroupfs和systemd。

默认情况下，这个选项是空值。此时Docker将首先尝试使用systemd，如果不可用，则使用cgroupfs。使用下面的命令可以指定：

```
$ sudo Docker -d --exec-opt native.cgroupdriver=cgroupfs
```

4. Daemon DNS配置

目前所有容器都会共用同一份DNS设置，因此当修改DNS设置后，所有容器将会实时更新使用。

使用Docker-d--dns 8.8.8.8可以指定DNS服务器地址，而使用Docker-d--dns-search example.com则可以修改DNS搜索服务器信息。

5. Daemon其他配置

Docker允许用户使用—default-ulimit来设置所有容器的资源访问限额。当启动容器时，如果没有限制资源限额，那么容器将会继承Daemon中的默认资源限额值。用户在容器运行过程中，也可以在使用Docker run时使用—ulimit来设置新的资源限额。

Docker还提供了一套IP地址翻译模块，用于容器访问外部主机资源。使用这种翻译模块后，Docker将会自动维护各容器之间的网络拓扑结构，可使用–ip-masq=false禁用此翻译模块。

此外Docker支持对数据目录/var/lib/Docker和数据临时目录/var/lib/Docker/tmp的软连接，使用Docker_TMPDIR也可以达到同样的效果。

如果需要使用Docker_TMPDIR，可参考下面的示例：

```
Docker_TMPDIR=/mnt/disk2/tmp  /usr/local/bin/Docker  -d  -D  -g
/var/lib/boot2Docker/Docker.log 2>&1
# 或者
export Docker_TMPDIR=/mnt/disk2/tmp
/usr/local/bin/Docker -d -D -g /var/lib/Docker -H unix:// > /var/
```

1）attach命令

在实际生产环境当中，用户经常会以后台模式创建并且启动容器。在后台模式下，用户无法操控容器或者获取容器中应用的输出内容。

只有当容器运行时，指定了前端交互式模式的同时也重定向了stdout之后，用户才能与容器进行交互。但如果在运行时指定了前端交互模式，那么就意味着这个pty不能退出，否则这个容器也将随之被关闭。这种情况在生产环境中肯定是无法被接受的。

那么是否意味着当容器处于后台模式运行时，用户就无法与容器进行交互呢？答案是否定的。通过attach命令，用户就可以挂载到正在运行的容器中，并且可以在容器中执行任何命令。

下面我们就看一下attach命令如何使用。

命令使用讲解：

首先我们看一下attach的用法：

```
Usage: Docker attach [OPTIONS] CONTAINER

Attach to a running container

  --no-stdin=false    Do not attach STDIN
  --sig-proxy=true    Proxy all received signals to the process
```

通过attach命令，用户可以凭借container id或者container name挂载到任意一个正在运行的容器之中，查看容器中应用输出内容或者与容器进行交互式操作，并且Docker也支持多个用户同时挂载到同一个容器之中进行操作。

当用户操作完毕之后，可以使用组合键Ctrl+P+Q退出容器。如果需要指定容器退出时的返回码，则可以通过exit来实现。

下面我们来看一个attach的操作实例。首先启动一个ubuntu的容器，为了保持此容器不退出，在启动容器时我们指定此容器运行top–b：

```
$ Docker run -d --name topdemo ubuntu /usr/bin/top -b
```

下一步就是通过attach命令挂载到这个容器中：

```
$ Docker attach topdemo

top - 02:05:52 up  3:05,  0 users,  load average: 0.01, 0.02, 0.0
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zom
Cpu(s):  0.1%us,  0.2%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.
Mem:    373572k total,   355560k used,    18012k free,    27872k
Swap:   786428k total,        0k used,   786428k free,   221740k

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAN
 1 root      20   0 17200 1116  912 R    0  0.3   0:00.03 top

 top - 02:05:55 up  3:05,  0 users,  load average: 0.01, 0.02, 0.
 Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zo
 Cpu(s):  0.0%us,  0.2%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0
 Mem:    373572k total,   355244k used,    18328k free,    27872k
```

```
Swap:   786428k total,        0k used,   786428k free,   221776k

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COM
      1 root      20   0 17208 1144  932 R    0  0.3   0:00.03 t



top - 02:05:58 up  3:06,  0 users,  load average: 0.01, 0.02, 0
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zo
Cpu(s):  0.2%us,  0.3%sy,  0.0%ni, 99.5%id,  0.0%wa,  0.0%hi,  0
Mem:    373572k total,   355780k used,    17792k free,    27880k
Swap:   786428k total,        0k used,   786428k free,   221776k

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMA
     1 root      20   0 17208 1144  932 R    0  0.3   0:00.03 to
```

因为我们指定此容器运行top命令，因此挂载到容器中后，就会看到top命令的输出结果。我们通过Ctrl+P+Q退出容器，然后再看一下容器是否还在运行：

```
$ Docker ps -a | grep topdemo

7998ac8581f9        ubuntu:14.04        "/usr/bin/top -b"   1 min
topdemo
```

通过Up About a minute可以看到，容器不会因为attach命令退出而停

止。如果我们选择使用exit退出容器，又会是什么样的效果呢？让我们再次使用attach命令挂载到容器中。

```
$ Docker attach topdemo

top - 02:05:52 up  3:05,  0 users,  load average: 0.01, 0.02, 0.0
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zom
Cpu(s):  0.1%us,  0.2%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.
Mem:    373572k total,   355560k used,    18012k free,    27872k
Swap:   786428k total,        0k used,   786428k free,   221740k

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAN
 1 root      20   0 17200 1116  912 R    0  0.3   0:00.13 top

 top - 02:05:55 up  3:05,  0 users,  load average: 0.01, 0.02, 0.
 Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zo
 Cpu(s):  0.0%us,  0.1%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0
 Mem:    373572k total,   355244k used,    18328k free,    27872k
 Swap:   786428k total,        0k used,   786428k free,   221776k

   PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COM
       1 root      20   0 17208 1144  932 R    0  0.3   0:00.03 t



 top - 02:05:58 up  3:06,  0 users,  load average: 0.02, 0.02, 0.
 Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zo
```

```
Cpu(s):  0.21%us,  0.3%sy,  0.0%ni, 99.5%id,  0.0%wa,  0.0%hi,
Mem:    373572k total,   355780k used,    17792k free,    27880k
Swap:   786428k total,        0k used,   786428k free,   221776k

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMA
     1 root      20   0 17208 1144  932 R    0  0.3   0:00.14 to
```

我们输入exit 10如下：

```
root@7998ac8581f9:/# exit 10
```

Attach命令退出之后，容器也会同步被关闭。我们查看一下当前此容器的状态：

```
$ Docker ps -a | grep topdemo

7998ac8581f9        ubuntu:14.04        "/usr/bin/top -b"   11 mi
ago                         topdemo
```

可以看到容器的退出响应码变成了10，符合我们的预期结果；这也就验证了用户可以通过exit命令来指定容器退出码。

2）build命令

在前面章节中，我们曾经提到所有的容器都是以image为基础进行创建的。image可以算作Docker运行的基础，由此可见，image对于Docker来说非常重要。下面，我们介绍如何创建image。

本节重点在于讲解build命令如何使用。与build命令紧密相关的一个文件是Dockerfile，这个文件是image的组织文件。在Dockerfile中定义了image的内容、命令以及配置信息。关于Dockerfile的相关内容，我们将在第7章中详细介绍。

下面我们就开始讲解build命令如何使用。

build命令的参数如下：

```
Usage: Docker build [OPTIONS] PATH | URL | -

Build a new image from the source code at PATH

-f, --file=""           Name of the Dockerfile (Default is 'PATH/
--force-rm=false        Always remove intermediate containers
--no-cache=false        Do not use cache when building the image
--pull=false            Always attempt to pull a newer version of
-q, --quiet=false       Suppress the verbose output generated by
--rm=true               Remove intermediate containers after a su
-t, --tag=""            Repository name (and optionally a tag) fo
-m, --memory=""         Memory limit for all build containers
--memory-swap=""        Total memory (memory + swap), `-1` to dis
-c, --cpu-shares        CPU Shares (relative weight)
```

```
--cpuset-mems=""        MEMs in which to allow execution, e.g. `0
--cpuset-cpus=""        CPUs in which to allow execution, e.g. `0
--cgroup-parent=""      Optional parent cgroup for the container
--ulimit=[]             Ulimit options
```

在build命令参数中，[OPTIONS] PATH|URL|-表示OPTIONS是可选项。如果不输入，则各参数保持默认。但PATH、URL和-是必输项，至少需要输入一个。

Docker在构建一个image时，需要读取一个配置文件。这个配置文件就是前面我们所提到的Dockerfile。这个文件默认文件名是Dockerfile，但也可以是其他文件名，不一定必须是Dockerfile（Dockerfile首字母需大写）。而且用户也可以通过PATH|URL|-来指定Dockerfile。

在Dockerfile中指定了此image中的数据、命令和配置数据，当需要使用其他文件来构建image时，只要这个文件满足Dockerfile的数据格式，Docker Daemon就可以正确读取并进行构建。

```
Docker build - < Dockerfile
```

在PATH|URL|-这三个参数中，“-”是最简单的一种方式。“-”表示build命令将会从STDIN读入数据。比如，我们可以通过下面的命令将Dockerfile中的数据通过STDIN传入到build中。

而PATH参数表示的是Dockerfile文件全路径。比如我们将Dockerfile

放置到$(HOME)/file目录，那么就可以按照下面命令来使用了：

```
Docker build $(HOME)/file/Dockerfile
```

通过上面的命令，build就会到指定的目录读取Dockerfile。如果文件名不是Dockerfile，例如叫做myimagefile（前提是myimagefile符合Dockerfile的语法结构，否则就会报错），则将路径名稍作改动就可以：

```
Docker build $(HOME)/file/myimagefile
```

这三个参数中，“-”和“PATH”都比较好理解，最后剩一个URL参数，下面来看URL如何使用。

如果需要使用URL参数，那么URL参数必须指向一个git地址（本地或者远程的git都可以）。Docker会通过git clone--depth 1–recursive命令将URL中指定的git仓库克隆到本地，然后将此仓库中的Dockerfile内容传送给Docker Daemon。

当给URL参数指定Git URL时，需要遵循GIT的URL语法。例如，Git URL中包含“:”时，Git将会clone URL中“:”前面的部分。因此“:”前面的部分必须是一个分支（branch），一个标签（tag）或者是一个SHA值。而“:”后面的部分将被作为Dockerfile的内容。例如，笔者在Github上面新创建了一个branch分支。

在此branch中，有一个名为Docker的目录。此目录中存储着构建image使用的Dockerfile，此Dockerfile的内容如下：

```
FROM 192.168.73.12:5001/mysql
LABEL version=2.0
LABEL build=0727
LABEL comment=GIT
```

我们可以使用下面的命令，通知Docker Daemon加载上面github中的Dockerfile来构建image：

```
Docker build https://github.com/andy-zhangtao/Docker-image.git#co
```

当执行这条命令时，Docker Daemon首先会从github上面克隆repositories，也就是后面的https://github.com/andy-zhangtao/Docker-image.git。当克隆成功后，再将当前分支切换到container分支中。

这两步完成后，build命令会使用“:”后面的路径作为Dockerfile进行镜像构建。

使用URL方式时需要注意，此种情况下Docker Daemon只会寻找以Dockerfile命名的文件，不支持自定义文件名。

使用URL模式时，GitURL的几种有效表达方式如表5-2所示。

表5-2 GitURL的几种有效表达方式

Git URL 前缀	构建使用的路径	Docker 路径
myrepo.git	refs/heads/master	/
myrepo.git#mytag	refs/tags/mytag	/
myrepo.git#mybranch	refs/heads/mybranch	/
myrepo.git#abcdef	sha1 = abcdef	/
myrepo.git#:myfolder	refs/heads/master	/myfolder
myrepo.git#master:myfolder	refs/heads/master	/myfolder
myrepo.git#mytag:myfolder	refs/tags/mytag	/myfolder
myrepo.git#mybranch:myfolder	refs/heads/mybranch	/myfolder
myrepo.git#abcdef:myfolder	sha1 = abcdef	/myfolder

为了提高构建效率，建议读者无论使用PATH还是URL，都将Dockerfile和构建过程中涉及的文件放到一个单独目录中。同时还可以将不需要的文件或者目录定义到Dockerignore文件中。这样Docker在构建image时，就会只加载使用到的文件。

因为构建image是一个比较消耗时的操作，所以在构建过程中，Docker client和Docker Daemon需要保持连接。一旦连接失败，构建将会自动取消。

当Docker Daemon构建成功后，build命令会正常退出。如果构建过程中出现错误，那么在client端也会看到相应的错误输出。例如下面我们演示的失败案例：

```
$ Docker build -t fail .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM busybox
 ---> 4986bf8c1536
Step 1 : RUN exit 13
```

```
 ---> Running in e26670ec7a0a
INFO[0000] The command [/bin/sh -c exit 13] returned a non-zero c
```

现在Docker build中的三个参数介绍完了，下面我们通过几个实例来演示Docker build的使用技巧。

①如果当前目录中存在Dockerfile，那么就可以不输入文件名。（因为Docker默认使用Dockerfile），例如：

```
$ Docker build .

Uploading context 10240 bytes
Step 1 : FROM busybox
Pulling repository busybox
 ---> e9aa60c60128MB/2.284 MB (100%) endpoint: https://cdn-regist
Step 2 : RUN ls -lh /
 ---> Running in 9c9e81692ae9
total 24
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 bin
drwxr-xr-x    5 root     root        4.0K Oct 19 00:19 dev
drwxr-xr-x    2 root     root        4.0K Oct 19 00:19 etc
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 lib
lrwxrwxrwx    1 root     root           3 Mar 12  2013 lib64 -> l
dr-xr-xr-x  116 root     root           0 Nov 15 23:34 proc
lrwxrwxrwx    1 root     root           3 Mar 12  2013 sbin -> bi
dr-xr-xr-x   13 root     root           0 Nov 15 23:34 sys
```

```
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 tmp
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 usr
 ---> b35f4035db3f
Step 3 : CMD echo Hello world
 ---> Running in 02071fceb21b
 ---> f52f38b7823e
Successfully built f52f38b7823e
Removing intermediate container 9c9e81692ae9
Removing intermediate container 02071fceb21b
```

请注意最后两行输出：

```
Removing intermediate container 9c9e81692ae9
Removing intermediate container 02071fceb21b
```

这两行表示当image构建成功后，Docker Daemon会把构建过程中所创建的临时image删掉。这是默认操作，如果想保留这些临时image，可以通过--rm=false来保留。

那么什么情况下可以删除临时image，什么情况下不需要删除呢？

一般来说，如果这个image需要经常性的进行构建，并且每次发生的变化都不大，那么建议保留临时image。因为Docker Daemon会复用临时image，以提高构建效率。如果不经常构建，或者变动特别大，那么建议删掉image。因为下次Daemon构建image时，发现变化非常大，也会重新生成临时image，旧的临时image此时就没有任何用途了，留着只

会浪费硬盘空间。

如果我们选择保留这些临时image，就需要修改上面的build命令：

```
$ Docker build  --rm=false .
Uploading context 18.829 MB
Uploading context
Step 0 : FROM busybox
 ---> 769b9341d937
Step 1 : CMD echo Hello world
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
$ echo ".git" > .Dockerignore
$ Docker build .
Uploading context  6.76 MB
Uploading context
Step 0 : FROM busybox
 ---> 769b9341d937
Step 1 : CMD echo Hello world
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
```

这样就不会删除临时image了。

②使用自定义的Dockerfile。下面的命令是使用自定义Dockerfile来执行build命令：

```
$ Docker build -f Dockerfile.debug .
```

Dockerfile.debug是我们假设在当前目录中的一个Dockerfile，通过-f可以强制使用自定义的构建文件。

在构建过程中还有一个常用的命令“-t”，通过-t可以指定镜像tag（tag和name的区别在后面章节会详细命令），通过下面的命令，可以给构建成功后的image添加tag：

```
$Docker build –t myImage .
```

实际生产环境中，经常会把自定义Dockerfile和指定tag联合起来使用，例如：

```
$ Docker build -f Dockerfiles/Dockerfile.debug -t myapp_debug .
$ Docker build -f Dockerfiles/Dockerfile.prod  -t myapp_prod .
```

在当前目录中，可以将不同用途的Dockerfile都放置在Dockerfiles目录中，然后根据不同的情况构建不同的image。

3）commit命令

上面我们介绍了如何构建image，当image构建完成之后，我们就可以使用这些image来创建相对应的容器。可是如果容器运行的结果并非我们所预期的呢？也就是说，如果image里面的数据构建错误了怎么办呢？

有的读者会想到，再重新构建一个镜像。对，这固然是一个方法，但仅仅适用于有非常明显错误时使用。如果错误一时很难排查，需要在容器中进行debug才能排查解决时，用户或许就会希望如果Docker能把debug完之后的容器保存成一个image，那该有多好呀。

在Docker remoteAPI中有一个API可以完成这样的事情。在Docker commandline中也有这样一个命令完成这个功能，即Docker commit命令。通过Docker commit，用户可以以某个容器为基础保存成一个新的image，同时调整这个image的某些配置参数。

我们看一下Docker commit的用法：

```
Usage: Docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

  -a, --author=""       Author (e.g., "John Hannibal Smith <hanni
  -c, --change=[]       Apply specified Dockerfile instructions w
  -m, --message=""      Commit message
  -p, --pause=true      Pause container during commit
```

Commit命令中参数比较少，这也从侧面说明了这个命令简单好

用。下面我们逐一讲解各参数如何使用。

```
-a, --author="":
```

这个参数的目标从字面上就很好理解，就是保存新的作者名称。虽说有的image可能使用的人不多，但也属于开发成果，所以加上作者名称，也算是对开发者脑力劳动的尊重。

```
-c, --change=[]:
```

这里的change如果你从字面理解是保存所有变更过的文件那就错了。这里的change是修改image里面的配置参数。因为我们新创建的image是基于某个容器而创建的，而所有的容器又都是以image为模板创建的。

因此我们通过commit命令所创建的image其实就是继承此容器的模板image，关系如图5-1所示。

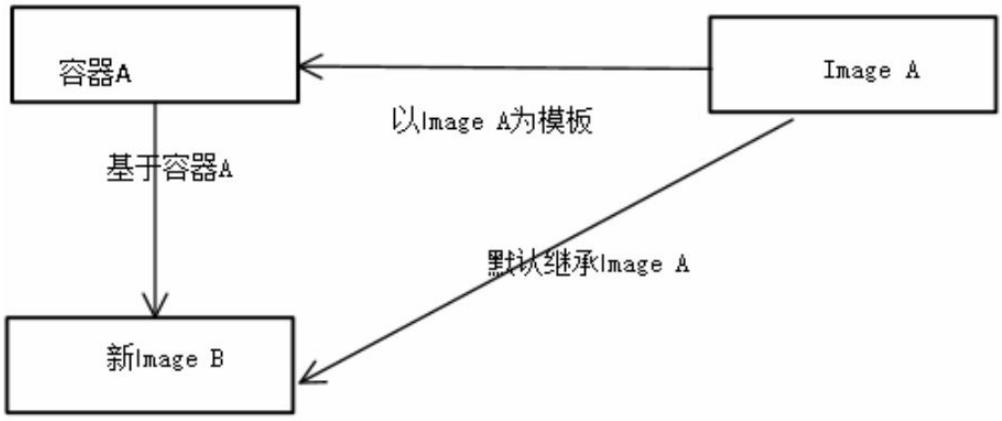


图5-1　创建的image与继承的image的关系

如果我们想调整Image B的参数，就需要使用-change了。比如Image A中没有debug所需的相关环境变量，那么在Image　B中就需要新增一个debug开关标志位，我们可以这样操作：

```
$Docker commit --change "ENV DEBUG true" c3f279d17e0a  SvenDowide
```

注意

这里的c3f279d17e0a是一个正在运行的容器。

当我们再次查询新image信息时，就会发现已经有DEBUG这个环境变量了：

```
$ Docker inspect -f "{{ .Config.Env }}" f5283438590d
[HOME=/ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/s
```

注意

f5283438590d是刚才commit成功后，Daemon所返回的新image ID。

目前Docker commit支持的可修改的参数为：

```
CMD|ENTRYPOINT|ENV|EXPOSE|LABEL|ONBUILD|USER|VOLUME|WORKDIR
```

目前只有这9项参数可以修改，其余的参数均继承自Image A。我们期待Docker后续版本可以支持更多参数。

```
-m, --message="":
```

这个参数是提交一个comment备注信息，用于说明此image用途，修改内容等其他备注信息。

当使用Docker inspect或者images/{ID}/json API时，可以显示此信息。此信息属性名为：Comment。

最后一个参数是：-p,--pause=true：

默认情况下，在执行commit操作时，容器内所有的进程都是处于暂停状态的。如果此时这个容器还在对外提供服务，例如，Web服务、db服务等，那么其他容器就会受到影响。

我们可以通过将-pause=false，来通知Daemon，在执行commit时需要保持容器内进程继续工作。

但如果采取-pause=false，那么commit的效率就会下降。因此要结合当时具体情况来决定是否暂停进程。

4）cp命令

cp是容器和host主机之间传输文件的命令。因为在Linux系统中，一切都是文件。所以这里的文件既可以指单个数据文件，也可以指某个目录。同时cp命令是双向传输的，可以从容器向主机复制文件，同时也可以从主机向容器复制文件。

在什么场合会使用到cp呢？结合笔者的经验，有以下场合比较适合使用cp命令：

a. debug环境中

比如容器A处于debug过程中，如果在debug过程中需要某个文件或者目录，虽然可以通过在Dockerfile文件中使用ADD命令解决问题，但耗时费力，性价比低。此时就可以通过cp命令，先将文件导进去，如果验证无误后再修改Dockerfile。

b. 自定义配置文件

假设容器A是一个提供Web服务的容器，里面运行的是play frameword的应用。如果正处于开发阶段，那么需要使用阶段的application.conf；如果正处于测试阶段，那么就需要使用测试阶段的application.conf。两个阶段中只有这个配置文件不同而已，其他所有的文件都相同。那么在启动容器后，可以根据阶段的不同，单独添加不同的配置文件，这样也就减少了image的维护量。

c. 需要分析容器中数据时

例如，容器A运行的结果并非预期中的结果，用户需要分析容器A中应用运行日志，或者将容器A中的配置文件单独取出进行测试。这时

就需要使用cp命令将文件从容器中单独拷出了。

> 注意
>
> 在Docker 1.8之前，cp命令仅支持从容器向主机的文件拷贝。从主机向容器的文件拷贝是从1.8才开始支持的。读者在使用cp命令时请注意Docker版本。

以上三种是笔者结合日常经验所使用cp命令的场景，读者可根据实际情况灵活使用cp命令，而不必拘泥于上述三种场景之中。

为了保持讲解时版本的统一性，下面我们看一下cp命令在Docker 1.7中是如何使用的。

命令使用讲解：

首先看一下cp都有哪些参数：

```
Usage: Docker cp [OPTIONS] CONTAINER:PATH HOSTDIR|-
--help  Print usage statement
```

可以看到cp命令其实很简单，因为在Docker 1.7中仅支持从容器向主机的复制，所以只有上述这一种用法。

细心的读者可能发现在用法中，没有对OPTIONS的介绍。按照经验，这些OPTIONS应该是对cp命令的一些补充，但很可惜在Docker 1.7

中，这些OPTIONS没有任何实际意义，在Docker 1.8中同样也没有实际意义。这可能是Docker的预留用法，至少目前来说OPTIONS里面没有任何参数。

剩下的就是CONTAINER:PATH、HOSTDIR和—这三个参数了。

CONTAINER:PATH指的是目标文件在容器中的路径。CONTAINER可以是容器名称或者容器ID，PATH则是文件在容器中的路径，默认情况是相对于容器根目录的相对路径，如果输入全路径也不会有问题。

HOSTDIR指的是文件在宿主机中的路径，也就是我们准备将此文件拷贝到主机什么位置。这里需要注意：在Docker 1.7中，此路径是文件所在的目录路径。即假设我们准备将文件/tmp/A.txt从容器拷贝到主机中的$(HOME)/Docker目录中，那么此时会有如下两种情况：

①$(HOME)/Docker目录已存在。

此时cp命令会将A.txt直接拷贝到$(HOME)/Docker中，执行完毕后，在$(HOME)/Docker会出现A.txt文件。执行的命令如下：

```
Docker cp <container id>:/tmp/A.txt $(HOME)/Docker
```

②$(HOME)/Docker目录不存在。

此时cp命令会首先创建Docker目录，然后再执行拷贝动作，执行的命令同样是：

```
Docker cp <container id>:/tmp/A.txt $(HOME)/Docker
```

从上面可以看出，在执行cp命令时，不需要指定主机的文件名。Docker默认情况下，会直接将A.txt从容器拷贝到主机中，文件名不会发生改变。但如果此时指定了文件名，例如：

```
Docker cp <container id>:/tmp/A.txt $(HOME)/Docker/A.txt
```

那么Docker会创建一个名为A.txt的目录，然后再将A.txt文件拷贝到A.txt目录中。

最后一个参数是“-”。这个参数表示Docker Daemon会把A.txt归档成tar文件，然后将此文件以二进制流的形式通过STDOUT发回给client端。此时如果在“-”后面没有任何处理的话，将会接收到二进制流。所以通常的处理方式是将此二进制流保存成tar文件。例如：

```
Docker cp <container id>:/tmp/A.txt - > /tmp/A.tar
```

在Docker 1.8中，cp命令支持双向拷贝，因此用法比较灵活。但万变不离其宗，主要用法和1.7中一致。所以在1.7上面，理解cp命令后，在1.8上面也会很快掌握cp命令。

5）create命令

create命令是Docker用来创建容器的一个命令。这个命令和后面会

讲到的Run命令有一些区别。Run命令执行完之后，容器就会被创建并且开始运行。而create命令执行完之后，仅仅是创建了一个容器而已。

从Docker的角度来分析的话，create命令是以image为模板创建的一个容器，并且在容器文件层的最上面一层添加一个读写层，容器里面所发生的所有数据变化都会被保存在最上面的读写层中。（读者是否还记得Docker使用的是AUFS文件系统，每个容器都是由不同的只读文件层叠加而成，只有最上面的一层是读写层）。

create命令有一项任务就是初始化这个读写层（更细节的内容会在第6章和第11章中介绍）命令。当create命令执行成功后，再通过start命令就可以启动这个容器了。

所以说RUN命令就是create和start两个命令的组合体。

下面我们介绍create命令的使用方法。

```
Usage: Docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
-a, --attach=[]                         Attach to STDIN, STDOUT or ST
--add-host=[]                           Add a custom host-to-IP mappi
--blkio-weight=0                        Block IO weight (relative wei
-c, --cpu-shares=0                      CPU shares (relative weight)
--cap-add=[]                            Add Linux capabilities
--cap-drop=[]                           Drop Linux capabilities
--cgroup-parent=""                      Optional parent cgroup for th
--cidfile=""                            Write the container ID to the
--cpu-period=0                          Limit CPU CFS (Completely Fai
```

```
--cpu-quota=0                      Limit CPU CFS (Completely Fai
--cpuset-cpus=""                   CPUs in which to allow execut
--cpuset-mems=""                   Memory nodes (MEMs) in which
--device=[]                        Add a host device to the cont
--dns=[]                           Set custom DNS servers
--dns-search=[]                    Set custom DNS search domains
-e, --env=[]                       Set environment variables
--entrypoint=""                    Overwrite the default ENTRYPO
--env-file=[]                      Read in a file of environment
--expose=[]                        Expose a port or a range of p
-h, --hostname=""                  Container host name
--help=false                       Print usage
-i, --interactive=false            Keep STDIN open even if not a
--ipc=""                           IPC namespace to use
-l, --label=[]                     Set metadata on the container
                                   value)
--label-file=[]                    Read in a line delimited file
--link=[]                          Add link to another container
--log-driver=""                    Logging driver for container
--log-opt=[]                       Log driver specific options
--lxc-conf=[]                      Add custom lxc options
-m, --memory=""                    Memory limit
--mac-address=""                   Container MAC address (e.g. 9
--memory-swap=""                   Total memory (memory + swap),
--memory-swappiness=""             Tune a container's memory swa
                                   between 0 and 100.
--name=""                          Assign a name to the containe
```

```
--net="bridge"                    Set the Network mode for the
--oom-kill-disable=false          Whether to disable OOM Killer
-P, --publish-all=false           Publish all exposed ports to
-p, --publish=[]                  Publish a container's port(s)
--pid=""                          PID namespace to use
--privileged=false                Give extended privileges to t
--read-only=false                 Mount the container's root fi
--restart="no"                    Restart policy (no, on-failur
--security-opt=[]                 Security options
-t, --tty=false                   Allocate a pseudo-TTY
--disable-content-trust=true      Skip image verification
-u, --user=""                     Username or UID
--ulimit=[]                       Ulimit options
--uts=""                          UTS namespace to use
-v, --volume=[]                   Bind mount a volume
--volumes-from=[]                 Mount volumes from the specif
-w, --workdir=""                  Working directory inside the
```

因为Create命令是用来创建和初始化容器的，所以其自身带有很多调整参数。在这么多的参数里面，经常使用的是下面5个参数。

```
-i, --interactive=false       Keep STDIN open even if not attache
-t, --tty=false               Allocate a pseudo-TTY
-P, --publish-all=false       Publish all exposed ports to random
-p, --publish=[]              Publish a 容器's port(s) to the hos
```

```
-v, --volume=[]                Bind mount a volume
```

其他参数均保持默认值即可。各个参数的详细说明请参阅本章最后一部分。

-i是将host的Stdin同容器的Stdin相连。通常情况下，-i和-t是配合使用的，通过-t会创建一个pty终端。当-t和-i配合使用时，就相当于在host环境中启动了一个链接容器的pty伪终端。而因为通过-i，已经将host的Stdin同容器的Stdin相互连接，所以在host的pty中输入的数据就会传输到容器之中。

-P/-p用来映射网络端口。在实际生产环境中，绝大多数的应用都需要访问其他应用，或被其他应用访问，因此开放一个端口是非常实际的需求。

当使用-p后，Docker可以映射用户所指定的端口。例如，-p 3306/3306就表示将容器的3306端口映射到主机的3306端口。如果使用-P，则表示容器中所有需要暴露的端口会映射到主机随机一个端口（端口范围为49153～65535）。例如，容器A使用的image在构建（build）阶段指定暴露8080和8084端口，那么当使用-P之后，8080端口就可能会随机映射到主机的49190端口或者另外一个空闲端口上，8084端口则会被映射到另外一个空闲端口上。

-v是挂载参数，即将主机的一个文件或者一个目录挂载到容器中。例如，我们将主机的目录A挂载到容器中的/tmp，并且命名为CA，就可以通过–v/A:/tmp/CA来实现。

下面我们使用这五个参数来创建一个容器：

```
$ Docker create -t -i fedora bash

6d8af538ec541dd581ebc2a24153a28329acb5268abe5ef868c1f1a261221752
```

-t和-i还可以连写成–it，效果是一样的。

```
$ Docker create -v .:/data --name data ubuntu

240633dfbb98128fa77473d3d9018f6123b99c454b3251427ae190a7d951ad57
```

当使用-v时，“:”左边表示的是host文件路径，右边则是容器中的文件路径。

```
$ Docker create  -P  --name data ubuntu
```

上面的命令表示将ubuntu默认端口全部映射到host中，执行成功之后，可以通过Docker port来查看具体端口号。

常用的参数就是上面的用法，表5-3是其他参数的使用方法。

表5-3 其他参数的使用方法

参数名称	参数含义
-a, --attach=[]	选择重定向流，可选项为 STDIN、STDOUT、STDERR
--add-host=[]	在容器中添加一项新的主机名，注意是主机名:IP 格式
--blkio-weight=0	I/O 阻塞权重，在 10~1000 之间取值
-c, --cpu-shares=0	用来设定此容器使用 CPU 的权重。比如 CpuShares 设定为 512，同时另外一个容器的 CpuShares 设定为 1024，则此容器使用 CPU 的时间为 512/(512+1024)=1/3
--cap-add=[]	用来为此容器添加内核权限
--cap-drop=[]	用来为此容器删除内核权限
--cgroup-parent=""	指定此容器的 cgroup 目录路径，如果此值为空，则默认将 Daemon 进程所在的 cgroup 目录路径作为此容器的 cgroup 目录路径；如果指定的路径不存在，则 Daemon 会创建此目录
--cidfile=""	将此容器的 ID 值输出到指定文件中
--cpu-period=0	用来设定此容器每次使用 CPU 的时间长度，以微秒为单位
--cpu-quota=0	用来设定此容器每次被 CPU 调度时的权重
--cpuset-cpus=""	用来设定此容器运行在哪个 CPU 中，例如设定为“0,1”，表示此容器只会在第一个和第二个 CPU 上面运行
--cpuset-mems=""	用来设定内存访问级别，只在 NUMA 架构中生效
--device=[]	JSON 对象，用来将 host 的设备挂载到此容器中。例如：{ "PathOnHost": "/dev/deviceName", "PathIn 容器": "/dev/deviceName", "CgroupPermissions": "mrw"}
--dns=[]	用来设定此容器的 DNS 信息
--dns-search=[]	用来设定此容器的 DNS 查询域信息
-e, --env=[]	用来设定此容器的环境变量

续表

参数名称	参数含义
--entrypoint=""	用来设定此容器启动后，后续的 command 将如何执行。如果 EntryPoint 值不为空，那么后续所有的 command 都将以参数的形式，交由 EntryPoint 中的命令执行
--env-file=[]	容器启动时会将此文件中的数据加载到环境变量中
--expose=[]	用来设定此容器中向外暴露的网络端口
-h, --hostname=""	用来设定此容器的主机名
--help=false	是否显示帮助信息
--ipc=""	设定 IPC 名称
-l, --label=[]	设定此容器的标签名称
--label-file=[]	当容器启动时，会将此文件中的数据初始为标签内容
--link=[]	设定此容器需要链接的其他容器 ID
--log-driver=""	设定此容器的日志存储驱动方式
--log-opt=[]	用来指定日志存储驱动所需的特定参数
--lxc-conf=[]	设定 execdriver，只有当 execdriver 设定为 LXC 时，此参数才有效
-m, --memory=""	指定限制 memory 使用量，以字节为单位。如果为 0，则表示由 Daemon 来控制
--mac-address=""	指定此容器的 MAC 地址
--memory-swap=""	限制 memory+swap 总使用量，以字节为单位。如果为-1，则表示 swap 禁用，MemroySwap>=Memory。如果为 0，则表示由 Daemon 来控制
--memory-swappiness=""	设定此容器的 swappiness 值，在 0～100 之间调节
--name=""	设定容器名称
--net="bridge"	设置此容器网络连接方式，可选值为：bridge、host 和容器:<namelid>
--oom-kill-disable=false	当主机出现 OOM 错误时，是否需要强制 kill 此容器
--pid=""	设定此容器的 PID 值
--privileged=false	设定此容器是否需要内核特权
--read-only=false	设定此容器的 root 文件层是否为只读层
--restart="no"	设定此容器的重启策略，可选项为 no、on-failure[:max-retry]、always。 当为 on-failure 时，需要指定重启次数
--security-opt=[]	在 MLS 系统中设置自定义标签，例如，可设置 SELinux
--disable-content-trust=true	设定是否启动 image 验证
-u, --user=""	设定此容器默认以哪个用户身份执行应用
--volumes-from=[]	用来设定此容器从其他容器继承而来的挂载信息
-w, --workdir=""	设定此容器的默认工作目录

6）diff

diff命令，最为人所熟知的功能是比较数据差异。diff最早出现在UNIX操作系统中的一个文本差异比较工具，后来随着SVN、GIT、CVS这些版本管理系统的流行，逐渐被其他人所熟知。

在Docker中，为什么需要diff功能呢？根据前面讲解的内容我们知道，Docker使用的是AUFS文件系统（也会使用其他文件系统，但本质是一样的），一个image就是多个文件层的叠加，每次容器所产生的数

据变化都会被反映并被保存在最上面的读写层中，因此每个文件层之间就会产生数据差异，而通过diff命令可以获取这些差异。

简而言之，diff就是用来比较容器文件层数据差异的命令。

下面是diff的使用方法：

```
Usage: Docker diff CONTAINER

Inspect changes on a container's filesystem
```

diff命令参数很少，只需要容器名称或者容器ID就可以，不需要其他参数，因而使用起来非常简单。

下面我们看一下diff命令的使用。首先我们在一个容器里面做一些增删改操作，例如在Ubuntu容器中，添加a.txt，删除b.txt，修改c.txt，如下所示：

```
root@95eaa193be4d:/#
root@95eaa193be4d:/# touch /tmp/a.txt
root@95eaa193be4d:/# echo "This is a add test" > /tmp/a.txt
root@95eaa193be4d:/# touch /tmp/b.txt
root@95eaa193be4d:/# rm /tmp/b.txt
root@95eaa193be4d:/# touch /tmp/c.txt
root@95eaa193be4d:/# echo "This is a change test">/tmp/c.txt
root@95eaa193be4d:/# echo "I add a new line ">>/tmp/c.txt
```

```
root@95eaa193be4d:/#
```

然后我们执行以下diff命令，看看会有什么变化：

```
root@boot2Docker:~# Docker diff 95eaa193be4d
C /tmp
A /tmp/a.txt
A /tmp/c.txt
```

这个结果表示什么意思呢？

- A表示新增（Add）。
- D表示删除（Delete）。
- C表示变化（Change）。

我们创建了b.txt，又删除了b.txt。但上面并没有反映出b.txt文件的变化。上面的结果好像不是我们所预期的结果。为什么会这样呢？这是因为我们所产生的数据是保存在容器最上面的读写层中，diff比较的是最上层和其他层的差异。虽然我们创建并删除了b.txt，但这个b.txt原本就不存在于其他任何层中，因此b.txt变化就不会体现出来了。

如果我们需要体现出删除一个文件的变化，该如何操作呢？读者也许明白应该如何进行后续操作了吧。对，就是将读写层保存起来，然后再执行删除操作。

保存最上面读写层，我们选择使用commit命令：

```
root@boot2Docker:~# Docker commit 95eaa193be4d newcontainer
3dc0cb6b08ca85929bebe7503c64eeb38cad5255123c91b5044dc9717f143086
```

然后基于最新的image，再创建一个容器。这个新容器就会包含我们刚才的文件操作。在这个新容器中，我们执行删除a.txt操作，然后看一下diff结果：

```
root@boot2Docker:~# Docker run -it 3dc0cb6b08ca85929bebe7503c64e
/bin/bash
root@0dd39f3ba6c6:/# rm /tmp/a.txt

Docker@boot2Docker:~$ Docker diff 0dd39f3ba6c6
C /tmp
D /tmp/a.txt
Docker@boot2Docker:~$
```

我们来分析一下两次的运行结果。因为我们创建或者删除的文件都保存在tmp目录中，所以diff之后只会显示tmp目录发生变化。在第一次diff中，因为a.txt和c.txt属于新增，所以被标记为A；而在第二次创建容器之后，我们删除了a.txt文件，所以a.txt就被标记为了D。

如果把Diff命令用在一个搭建编译环境的容器之中，那么作用就会非常明显了。例如，下面的容器是一个Golang编译环境的容器，使用diff命令，可以获知每次都有哪些文件发生了变更：

```
$ Docker diff 7bb0e258aefe

C /dev
A /dev/kmsg
C /etc
A /etc/mtab
A /go
A /go/src
A /go/src/github.com
A /go/src/github.com/Docker
A /go/src/github.com/Docker/Docker
A /go/src/github.com/Docker/Docker/.git
```

7）events命令

当容器运行时，尤其是以后台方式运行时，Docker client是无法得知此容器内部运行情况的。但在实际的生产环境中，有很多场合都需要获取到容器内部的运行情况。

为此，Docker提供了几种方法来获取容器内部运行情况：挂载（attach）、日志（logs）和事件（events）。这几种方式都可以获取到容器内部情况，但又各自有不同的使用场景。在本节中，我们介绍events的使用方法。

events顾名思义，就是获取容器所报告的事件。Docker容器在运行过程中会向Docker Daemon报告如下事件：

```
create, destroy, die, export, kill, oom, pause, restart, start, s
```

除容器之外，image也会在使用过程中向Docker　Daemon报告untag和delete两种事件。

下面我们看一下events命令的使用方法：

```
Usage: Docker events [OPTIONS]

Get real time events from the server

  -f, --filter=[]    Filter output based on conditions provided
  --since=""         Show all events created since timestamp
  --until=""         Stream events until this timestamp
```

events命令共有三个参数选项，这三个选项都用于结果筛选。其中--since和--until会根据时间来筛选结果。这里的时间既可以是UNIX时间戳，也可以是遵从RFC3339中规定的时间格式（Docker　1.8版本中，也可接受诸如-10m、1h30m这样的时间表达式）。

而--filter可用来筛选指定容器或者指定image的事件类型。--filter后面需要一个key=value这样的表达式。比如：--filter"foo=bar"--filter"bif=baz"。目前可用的筛选项是container、event、image。后面我们将演示如何使用--filter筛选数据。

我们首先演示如何使用--since和--until来筛选事件。--since是指从某一个时间点开始Docker Daemon所接收到的所有事件。--unit是指从某一个时间点开始到某一个时间点为止，Daemon所接收到的所有事件。换言之，--since可以单用，但--unit必须和--since联合使用。例如下面的命令：

```
Docker@boot2Docker:~$ Docker events --since 1378216169
```

表示从1378216169开始到现在Daemon所接收的所有事件。我们节选了部分返回数据用作演示：

```
2015-08-17T06:36:31.000000000Z 192.168.73.12:5001/mysql:latest: p
2015-08-17T06:36:31.000000000Z 74a862cd919d: (from 74a862cd919d)
2015-08-17T06:36:31.000000000Z 74a862cd919d: (from 74a862cd919d)
2015-08-17T06:36:31.000000000Z 74a862cd919d: (from 74a862cd919d)
2015-08-18T01:13:44.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T01:13:44.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T10:03:28.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T10:03:28.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T10:03:46.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T10:04:11.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T10:04:11.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T10:05:03.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T10:05:16.000000000Z 74a862cd919d: (from 192.168.73.12
2015-08-18T10:05:16.000000000Z 74a862cd919d: (from 192.168.73.12
```

如果换做使用--until，那么需要和since一起配合使用：

```
Docker@boot2Docker:~$ Docker events --since 1439964402 --until 1
2015-08-19T00:44:57.000000000Z 95eaa193be4d: (from 192.168.73.12
2015-08-19T00:44:57.000000000Z 95eaa193be4d: (from 192.168.73.12
2015-08-19T00:45:36.000000000Z 95eaa193be4d: (from 192.168.73.12
cat /tmp/a.txt
2015-08-19T00:45:36.000000000Z 95eaa193be4d: (from 192.168.73.12
cat /tmp/a.txt
2015-08-19T00:46:40.000000000Z 95eaa193be4d: (from 192.168.73.12
cat /tmp/c.txt
2015-08-19T00:46:40.000000000Z 95eaa193be4d: (from 192.168.73.12
cat /tmp/c.txt
2015-08-19T00:51:20.000000000Z 95eaa193be4d: (from 95eaa193be4d)
2015-08-19T00:51:20.000000000Z 95eaa193be4d: (from 95eaa193be4d)
2015-08-19T00:51:34.000000000Z 95eaa193be4d: (from 192.168.73.12
```

如果单独使用--until，那么将不会输出任何数据。如果感觉时间戳不方便的话，那么可以使用RFC3339中规定的时间格式，例如下面的例子：

```
$ Docker events --since '2013-09-03'
2014-09-03T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-09-03T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
```

```
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8
2014-09-03T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8

$ Docker events --since '2013-09-03T15:49:29'
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8
```

看完--since和--until的示例后，我们看一下--filter的示例，前面提到--filter接收的是key=value这样的键值对。目前有效的key是container、event、image。下面我们逐一演示：

```
$ Docker events --filter 'event=stop'
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-09-03T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8

$ Docker events --filter 'image=ubuntu-1:14.04'
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1

$ Docker events --filter 'container=7805c1d35632'
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8
```

```
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8

$ Docker events --filter 'container=7805c1d35632' --filter 'conta
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8

$ Docker events --filter 'container=7805c1d35632' --filter 'event
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8

$ Docker events --filter 'container=container_1' --filter 'contai
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8
```

8）exec命令

exec是在容器内部执行命令的命令。在Docker使用过程中，很少会出现启动容器时就直接创建pty的情况。因为在启动容器时就创建了pty，那么当这个终端关闭时，容器也会随之关闭，这在生产环境中是不合理的。所以在绝大多数情况下，容器都是以后台模式来运行的。

既然容器是以后台模式在运行，就意味着通常情况下我们是无法获取容器内部情况的。虽然我们可以通过logs、attach和event来获取此容器内部部分信息，但这些命令都有一个共同点，即只能获取到容器内已

经发生的事情。例如，只有容器中的应用产生了日志，才能用logs获取到；同样只有容器中产生了事件才能使用events获取到。如果应用没有产生任何数据，那么使用上述三个命令也没有多大效果。

而exec命令则不同，它可以在容器中运行其他命令。使用exec命令可以在容器中执行脚本、命令或者任何容器中合法的工具。

通过exec命令我们可以很容易的针对容器进行debug，比如在容器中修改执行脚本，或者在容器中执行debug程序等操作。

Exec的使用非常简单：

```
Usage: Docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

Run a command in a running container

  -d, --detach=false          Detached mode: run command in the b
  -i, --interactive=false     Keep STDIN open even if not attache
  -t, --tty=false             Allocate a pseudo-TTY
  -u, --user=                 Username or UID (format: <name|uid>
```

在大多数情况下，后面四个参数都不需要使用，例如，执行ps并且筛选进程：

```
Docker@boot2Docker:~$ Docker exec  0dd39f3ba6c6 ps -s|grep XXXX
```

或者创建一个新文件：

```
Docker@boot2Docker:~$ Docker exec  0dd39f3ba6c6 touch /tmp/1.txt
```

在四个参数中，-t和-i经常联合使用，例如修改文件时：

```
Docker@boot2Docker:~$ Docker exec  -it 0dd39f3ba6c6 vi /tmp/1.tx
```

但这种情况下，如果不申请一个pty和绑定stdin的话，那么仅仅能编辑文本却无法保存。因为没有pty，容器无法接收到控制字符。

9）history命令

Docker有一个diff命令，这个命令可以查看容器中文件系统的变更历史。通过diff命令，就可以追溯容器的文件变更。在image层面同样也有一个命令可以追踪它的历史变更，这个命令就是history命令。

在使用history命令时，Docker Daemon将返回此image在构建过程中所有执行过的命令。也就是说，history命令可以把image的历史完全展现出来。比如image的baseimage是哪个，image中的应用如何安装，image中的目录如何被创建，image中CMD、Entrypoint都是如何设置的，等等。这些信息都可以通过history一一展现出来。

所以说history正如其意，将image的历史完全暴露无遗。

History的应用很简单，示例如下：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
e9453316c630    2015-03-24T06:04:38Z    /bin/sh -c #(nop) EXPOSE
365f2af56f23    2015-03-24T06:04:34Z    /bin/sh -c apt-get update
b835f321160b    2015-03-19T09:00:54Z    /bin/sh -c #(nop) MAINTAI
2103b00b3fdf    2015-03-10T17:21:33Z    /bin/sh -c #(nop) CMD [/b
4faa69f72743    2015-03-10T17:21:32Z    /bin/sh -c sed -i 's/^#\s
76b658ecb564    2015-03-10T17:21:30Z    /bin/sh -c echo '#!/bin/s
f0dde87450ec    2015-03-10T17:21:24Z    /bin/sh -c #(nop) ADD fil
511136ea3c5a    2013-06-13T21:03:50Z                      0
```

通过这两个例子可以看出，当--human=true之后。可以把时间和大小转化成容易理解的格式。除此之外，就没有太大区别了。

同--human相比，--no-trunc对用户更为重要。当--no-trunc默认设为false，并且--no-trunc设为true之后，会展现下面的结果：

```
Docker@boot2Docker:~$ Docker history  --no-trunc=true 192.168.73
IMAGE   CREATED   CREATED BY               SIZE               COM
e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3c17836d84c735b
#(nop) EXPOSE 22/tcp 80/tcp 8080/tcp 8088/tcp
365f2af56f23411f84b8c9573b165b3fa666baebe56f165ae13a8f8bdc9bc100
apt-get update && apt-get install -y openjdk-7-jdk && apt-get ins
zip libstdc++5 libc6-i386 python git curl openssh-server   572.2
b835f321160b6b78d7fea9eab390750db8a19d34f3578beaf2bd207c813af7af
#(nop) MAINTAINER TIBCO Software Inc. <dev@tibco-support.com>
2103b00b3fdf1d26a86aded36ae73c1c425def0f779a6e69073b3b77377df348
#(nop) CMD [/bin/bash]
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

history才会把所有信息都返回，如下所示：

```
Docker@boot2Docker:~$ Docker history  --quiet=true 192.168.73.12
e9453316c630
365f2af56f23
b835f321160b
2103b00b3fdf
4faa69f72743
76b658ecb564
f0dde87450ec
511136ea3c5a

Docker@boot2Docker:~$ Docker history  --quiet=false 192.168.73.1
IMAGE           CREATED       CREATED BY
           COMMENT
e9453316c630   4 months ago  /bin/sh -c #(nop) EXPOSE 22/tcp 80/t
365f2af56f23   4 months ago  /bin/sh -c apt-get update && apt-get
b835f321160b   5 months ago  /bin/sh -c #(nop) MAINTAINER TIBCO S
2103b00b3fdf   5 months ago  /bin/sh -c #(nop) CMD [/bin/bash]
4faa69f72743   5 months ago  /bin/sh -c sed -i 's/^#\s*\(deb.*uni
76b658ecb564   5 months ago  /bin/sh -c echo '#!/bin/sh' > /usr/s
f0dde87450ec   5 months ago  /bin/sh -c #(nop) ADD file:a2d97c73f
511136ea3c5a   2 years ago                                     0 B
```

通过上面三个参数的演示，可以发现使用history默认的参数设置就

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
 Dirs: 44
 Dirperm1 Supported: true
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 4.0.7-boot2Docker
Operating System: Boot2Docker 1.7.0 (TCL 6.3); master : f2d5ca6 -
CPUs: 1
Total Memory: 1.956 GiB
Name: boot2Docker
ID: ET6I:D32F:DF37:M5BI:ACKC:TT57:O7VS:ICYW:JZT6:FMCW:ADTV:ZOS7
Debug mode (server): true
File Descriptors: 10
Goroutines: 17
System Time: 2015-08-21T07:16:01.712506139Z
EventsListeners: 0
Init SHA1:
Init Path: /usr/local/bin/Docker
Docker Root Dir: /var/lib/Docker
```

里面的属性，就不一一介绍了。大部分通过属性名就可以分辨出功能。但有一些属性，只有启用Debug模式时才会显示，否则不显示，如表5-4所示。

表5-4　启用Debug模式时才会显示的属性

Debug mode (server)	File Descriptors

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

1. 基本信息类，包括名称，创建时间，当前状态（运行、停止或者暂停）等；

2. 运行参数类，包括网络信息、环境变量、主机名称等；

3. 底层数据类，包括CPU设置、内存设置、虚拟化设置等。

下面我们看一下inspect如何使用：

```
Usage: Docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE

Return low-level information on a container or image

  -f, --format=       Format the output using the given go templa
  --help=false        Print usage
```

通过上面的使用方式可以看到，inspect支持返回多个目标信息，只需要输入多个ID，并且中间使用空格隔开即可。下面是一个使用inspect返回的示例：

```
Docker@Docker-cloud-r720:~$ Docker inspect 5d35dd929e49
[
{
    "Id": "5d35dd929e49558b06505a940026c0eaadd5a454f10f4edb3b319
    "Created": "2015-07-29T05:07:33.152756431Z",
    "Path": "Docker-registry",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "NetworkID": "d252225ae1fe82740ff295635c0e98ffb4a84583ea2
        "PortMapping": null,
        "Ports": {
            "5000/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "5002"
                }
            ]
        },
        "SandboxKey": "/var/run/Docker/netns/5d35dd929e49",
        "SecondaryIPAddresses": null,
        "SecondaryIPv6Addresses": null
    },
    "ResolvConfPath":
"/home/Docker/.Docker/containers/5d35dd929e49558b06505a940026c0ea
1/resolv.conf",
    "HostnamePath":
"/home/Docker/.Docker/containers/5d35dd929e49558b06505a940026c0ea
1/hostname",
    "HostsPath":
"/home/Docker/.Docker/containers/5d35dd929e49558b06505a940026c0ea
1/hosts",
    "LogPath":
"/home/Docker/.Docker/containers/5d35dd929e49558b06505a940026c0ea
1/5d35dd929e49558b06505a940026c0eaadd5a454f10f4edb3b3197fcf20f986
    "Name": "/pa-registry",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "OomKillDisable": false,
        "Privileged": false,
        "PortBindings": {
            "5000/tcp": [
                {
                    "HostIp": "",
                    "HostPort": "5002"
                }
            ]
        },
        "Links": null,
        "PublishAllPorts": false,
        "Dns": null,
        "DnsSearch": null,
        "ExtraHosts": null,
        "VolumesFrom": null,
        "Devices": [],
        "NetworkMode": "bridge",
        "IpcMode": "",
        "PidMode": "",
        "UTSMode": "",
        "CapAdd": null,
        "CapDrop": null,
        "RestartPolicy": {
            "Name": "no",
            "MaximumRetryCount": 0
        },
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            "SETTINGS_FLAVOR=dev"
        ],
        "Cmd": [
            "Docker-registry"
        ],
        "Image": "registry",
        "Volumes": null,
        "VolumeDriver": "",
        "WorkingDir": "",
        "Entrypoint": null,
        "NetworkDisabled": false,
        "MacAddress": "",
        "OnBuild": null,
        "Labels": {}
    }
}
]
```

可以看到inspect返回的数据非常丰富。上面的信息是当前目标最完整的信息，但某些时候我们并不需要这么多的信息，而仅仅需要其中的一部分。从inspect返回的结果可以看出，上述信息是一个JSON格式的字符串。这时候，就需要使用-f参数来自定义输出内容了。-f其实就是选择JSON中的一部分输出。

那-f参数如何使用呢？我们看一下上面JSON字符串的结构：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "IPPrefixLen": ,
        "IPv6Gateway": “",
        "LinkLocalIPv6Address": "",
        "LinkLocalIPv6PrefixLen": ,
        "MacAddress": “",
       "NetworkID": "",
        "PortMapping": ,
        "Ports": {},
        "SandboxKey": "",
        "SecondaryIPAddresses": ,
        "SecondaryIPv6Addresses":
    },
    "ResolvConfPath": "",
    "HostnamePath": "",
    "HostsPath": "",
    "LogPath": "",
    "Name": "",
    "RestartCount": ,
    "Driver": "",
    "ExecDriver": "",
    "MountLabel": "",
    "ProcessLabel": "",
    "Volumes": {},
    "VolumesRW": {},
    "AppArmorProfile": "",
    "ExecIDs": ,
    "HostConfig": {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "CapAdd": ,
        "CapDrop": ,
        "RestartPolicy": {},
        "SecurityOpt": ,
        "ReadonlyRootfs": ,
        "Ulimits": ,
        "LogConfig": {},
        "CgroupParent": ""
    },
    "Config": {
        "Hostname": "",
        "Domainname": "",
        "User": "",
        "AttachStdin": ,
        "AttachStdout": ,
        "AttachStderr": ,
        "PortSpecs": ,
        "ExposedPorts": {},
        "Tty": ,
        "OpenStdin": ,
        "StdinOnce": ,
        "Env": [],
        "Cmd": [],
        "Image": "",
        "Volumes": ,
        "VolumeDriver": "",
        "WorkingDir": "",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "ExecDriver": "",
    "MountLabel": "",
    "ProcessLabel": "",
    "Volumes": {},
    "VolumesRW": {},
    "AppArmorProfile": "",
    "ExecIDs": ,
    "HostConfig": { },
    "Config": {}
}
```

这些是第一层级的属性值，然后State、NetworkSettings、HostsPat还包含一些第二层级的属性值。所以上面的JSON就是由一层一层的属性封装起来的。

当我们弄清JSON的层级关系之后，在需要使用-f来定义输出时，其实就是按照层级来输出数据。例如，我们需要输出容器的IPAddress，则首先定位IPAddress所处的层级：

```
NetworkSettings –> IPAddress
```

那么-f的输入值就是{{.NetworkSettings.IPAddress}},这里面{{}}是固定格式，现在不需要深究，如果读者感兴趣的话，可以参考Golang中对于JSON的处理方式。

下面的命令会输出容器IP地址：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

在使用inspect定义输出内容时，一定要注意大小写匹配。Docker是严格按照字符进行匹配的。如果大小写不一致，就会出现下面的错误：

```
Docker@Docker-cloud-r720:～$ Docker inspect -f {{.logPath}} 5d35d
<no value>
```

在使用-f时还有一个隐藏的小技巧，比如我们可以看到Ports的数据：

```
"Ports": {
            "5000/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "5002"
                }
            ]
        }
```

熟悉JSON语法的读者可以看出ports包含多个对象，每个对象又是一个对象数组。假如我们仅仅需要ports第一个对象的HostPort属性值时，应该怎么办呢？

这种情况下，可以使用index函数。Index函数可以返回符合匹配参数的对象数据，例如我们只获取ports的第一个对象：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

除index函数外，inspect还有一个JSON函数，这个函数可以将输出数据以JSON格式输出。例如，上面的config数据，如果原样输出，那么是这样的：

```
Docker@Docker-cloud-r720:~$ Docker inspect -f '{{.Config}}' 5d35
{5d35dd929e49   false false false [] map[5000/tcp:{}] false false
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
Docker_REGISTRY_CONFIG=/Docker-registry/config/config_sample.yml
registry map[]   <nil> false  [] map[]}
```

如果不熟悉config的输出结构，就很难处理。这时我们添加上JSON函数之后，情况就大不一样了：

```
Docker@Docker-cloud-r720:~$ Docker inspect -f '{{json .Config}}'
{
    "Hostname": "5d35dd929e49",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "PortSpecs": null,
    "ExposedPorts": {
        "5000/tcp": {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

了说明问题，将那个字符串做了JSON格式化。其实只是展现有所不同，本质还是一样的。

当输出这样的数据后，无论人眼识别还是代码处理都会很容易。所以在使用inspect命令时，建议添加上JSON函数，百利而无一害。

12）kill命令

kill是强制停止容器的命令，在正常情况下不会用到。当需要关掉某个容器时，优先使用stop命令。只有容器里面的应用出现异常情况无法正常停止时，例如容器内进程出现死循环，或者容器内资源耗尽已经无法按照正常流程关闭时，才会考虑使用kill命令。

使用kill时，仅需要提供容器名称即可，例如：

```
Usage: Docker kill [OPTIONS] CONTAINER [CONTAINER...]

Kill a running container using SIGKILL or a specified signal

  --help=false           Print usage
  -s, --signal=KILL     Signal to send to the container
```

如果需要同时关掉多个容器，则只需要输入多个容器名称，中间用空格隔开。

默认情况下，kill命令会向指定的容器发送SIGKILL信号，如果需要向容器发送特定的信号，则可以使用--signal参数。例如：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  -f, --follow=false                Follow log output
  --help=false                      Print usage
  --since=                          Show logs since timestamp
  -t, --timestamps=false            Show timestamps
  --tail=all                        Number of lines to show from t
```

Logs命令除help之外有4个参数，我们通过演示来逐一讲解这些参数如何使用。首先看一下--follow参数。这个参数默认为false，当设置为true时，Docker logs命令将实时显示STDOUT追加的日志。例如：

```
Docker@Docker-cloud-r720:～$ Docker logs -f 6e0e2d6ccfb5
[2015-07-06 02:32:00 +0000] [1] [INFO] Starting gunicorn 19.1.1
[2015-07-06 02:32:00 +0000] [1] [INFO] Listening at: http://0.0.0
[2015-07-06 02:32:00 +0000] [1] [INFO] Using worker: gevent
[2015-07-06 02:32:00 +0000] [11] [INFO] Booting worker with pid:
[2015-07-06 02:32:00 +0000] [12] [INFO] Booting worker with pid:
[2015-07-06 02:32:00 +0000] [13] [INFO] Booting worker with pid:
[2015-07-06 02:32:00 +0000] [14] [INFO] Booting worker with pid:
06/Jul/2015:02:32:01 +0000 WARNING: Cache storage disabled!
06/Jul/2015:02:32:01 +0000 WARNING: LRU cache disabled!
06/Jul/2015:02:32:01 +0000 DEBUG: Will return Docker-registry.dri
<中间略过部分日志>
192.168.73.12 - - [17/Aug/2015:06:35:00 +0000] "GET /v1/images/
1eba0810df1fbb18cb2316308cd6eb50ee3d61f97b8a5b45391b24ac6ebfdf5d/
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
192.168.73.12:8081/repository/library/mysql/" "Mozilla/5.0 (Windo
537.36 (KHTML, like Gecko) Chrome/44.0.2403.125 Safari/537.36"
03/Aug/2015:00:36:44 +0000 DEBUG: args = {'namespace': u'library
03/Aug/2015:00:36:44 +0000 DEBUG: [get_tags] namespace=library; r
172.17.0.10 - - [03/Aug/2015:00:36:44 +0000] "GET /v1/repositorie
78  "http://192.168.73.12:8081/repository/library/mysql/"  "Mozil
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.125 Safar
03/Aug/2015:00:36:44 +0000 DEBUG: args = {'image_id':
u'1eba0810df1fbb18cb2316308cd6eb50ee3d61f97b8a5b45391b24ac6ebfdf5
172.17.0.10 - - [03/Aug/2015:00:36:44 +0000] "GET /v1/images/
1eba0810df1fbb18cb2316308cd6eb50ee3d61f97b8a5b45391b24ac6ebfdf5d/
192.168.73.12:8081/repository/library/mysql/" "Mozilla/5.0 (Windo
537.36 (KHTML, like Gecko) Chrome/44.0.2403.125 Safari/537.36"
03/Aug/2015:00:36:45 +0000 DEBUG: args = {'namespace': u'library
03/Aug/2015:00:36:45 +0000 DEBUG: [get_tags] namespace=library; r
```

RFC3339协议规定的时间格式相对于UNIX时间戳而言，再容易理解。下面就是使用RFC3339协议的示例：

```
Docker@Docker-cloud-r720:～$ Docker logs --since="2015-08-01" 6e0
172.17.0.10  -  -  [03/Aug/2015:00:36:41  +0000]  "GET  /v1/searc
192.168.73.12:8081/repositories" "Mozilla/5.0 (Windows NT 6.3; WO
Gecko) Chrome/44.0.2403.125 Safari/537.36"
172.17.0.10  -  -  [03/Aug/2015:00:36:44  +0000]  "GET  /v1/searc
192.168.73.12:8081/repository/library/mysql/" "Mozilla/5.0 (Windo
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.12
2015-08-03T00:36:44.117930497Z 03/Aug/2015:00:36:44 +0000 DEBUG:
'repository': u'mysql'}
2015-08-03T00:36:44.119017658Z 03/Aug/2015:00:36:44 +0000 DEBU
repository=mysql
```

加黑部分就是每条日志的时间戳。其实每条日志在记录时都会有一个时间戳。--since命令就是通过这个时间戳来筛选日志的。

最后是--tail命令。这个命令用来显示最后N条记录。例如，我们只想看最后一条日志，就可以这样操作：

```
Docker@Docker-cloud-r720:～$ Docker logs --tail=1 6e0e2d6ccfb5
192.168.73.12 - - [17/Aug/2015:06:35:00 +0000] "GET /v1/images/
1eba0810df1fbb18cb2316308cd6eb50ee3d61f97b8a5b45391b24ac6ebfdf5d/
"Docker/1.7.1 go/go1.4.2 git-commit/786b29d-dirty kernel/4.0.7-bo
```

--tail命令可以同--f、--since命令组合使用。当与上述两个命令组合使用时，会将上述两个命令返回的结果集中到一起，然后截取最后N条记录进行展示。

14）ps命令

ps命令是Docker CLI中最常用的一个命令了，在介绍Docker的文档或者书籍中，第一个使用的命令大多是Docker images，第二个就是Docker ps。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

首先是--all=false。默认情况下，Docker只会返回当前处于running状态的容器，其他状态的容器默认不显示。如果我们需要查看所有状态的容器，或者说查看所有容器的信息。那么就需要将--all置为true，或者直接输入-a都是同样的效果。例如，下例为默认的输出：

```
Docker@Docker-cloud-r720:~$ Docker ps
CONTAINER ID      IMAGE      COMMAND       CREATED      STATUS
8546477baf70      konradkleine/Docker-registry-frontend   "/bin/s
Up 3 weeks        443/tcp, 0.0.0.0:8081->80/tcp   my-registry-ui
2660e91327f6      konradkleine/Docker-registry-frontend   "/bin/s
Up 3 weeks        443/tcp, 0.0.0.0:8082->80/tcp   pa-registry-ui
5d35dd929e49      registry                                "Docker
Up 3 weeks        0.0.0.0:5002->5000/tcp          pa-registry
6e0e2d6ccfb5      registry                                "Docker
Up 3 weeks        0.0.0.0:5001->5000/tcp          dreamy_almeida
```

当我们将--all置为true后，输出如下：

```
Docker@Docker-cloud-r720:~$ Docker ps -a
CONTAINER ID   IMAGE              COMMAND           CREATED
69d3bafc83da   ubuntu:14.04     "/bin/bash"      4 days ago
33611f7600b4   192.168.73.12:5001/library/ubuntu-dev    "/bin/bas
days ago         berserk_babbage
8546477baf70     konradkleine/Docker-registry-frontend    "/bin/s
Up 3 weeks                443/tcp, 0.0.0.0:8081->80/tcp
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
$Docker ps –before 95eaa193be4d
CONTAINER ID    IMAGE         COMMAND        CREATED      STATUS
2d3167ccecd6    ubuntu-dev   "/bin/bash"    5 days ago     Exited
4788a0283b14    ubuntu-dev   "/bin/bash"    5 days ago     Exited
1dc8d6956f99    ubuntu-dev   "/bin/bash"    5 days ago     Exited
```

可以看出，在95eaa193be4d创建之前的三个容器都筛选出来了，符合我们的预期。在使用--before时需要注意，此参数会将指定容器之前所有状态的容器都显示出来。如果我们需要的这些容器中筛选出特定的容器，那么就需要使用--filter参数。

--filter可以用来筛选哪些容器需要被显示。--filter可以从五个维度进行筛选，分别是：id、label、name、exited和status。除了exited只能在使用--all时生效外，其他四个维度可以和ps任意参数相互组合使用。比如我们需要在上面的结果中筛选出当前处于running状态的容器，配合--filter参数：

```
$Docker ps --before 95eaa193be4d   -f 'status=running
```

这样ps命令就会在--before返回的结果中筛选出status=running的容器，并且展现出来。除running外，status还支持筛选created、restarting、paused和exited这四种状态。

--latest参数是用来展现最后一个被创建的容器。当使用这个参数时，无论这个容器当前处于什么状态，都将被展现出来。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
Docker@boot2Docker:~$ Docker ps --no-trunc -q
6cd3885a6a4085775f87aa23508f364b5e3bc26dd6db8ad2c1c36bbb06a6e488
```

为什么说这个参数经常用到呢？这是因为我们在编写一些脚本时，例如批量stop或者kill时，往往只关心容器ID值，其他的属性值我们并不关心。如果没有-q参数，那么我们就必须通过其他手段将容器ID截取出来。现在有了-q参数，这些工作就会变得很简单，例如下面批量删除容器的命令：

```
Docker rm $(Docker ps -a -q)
```

--size参数可以展现目前此容器所消耗的硬盘空间，例如：

```
CONTAINER ID      IMAGE        COMMAND     CREATED     STATUS   P
6cd3885a6a40        192.168.73.12:5001/library/ubuntu-dev   "/bi
28 hours (Paused)  22/tcp, 80/tcp, 8080/tcp, 8088/tcp   desperate
```

这个参数在实际生产环境中使用率并不高，但在容器监控方面，聊胜于无，有总比没有强。

最后一个参数是--since。这个参数和--before相反，是用来展现从指定容器创建开始到现在为止，被创建的容器信息。我们继续使用--before那三个容器为例说明，当我们使用了--since=容器A，那么就只会

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

这个命令和tag命令容易混淆，这两个命令都是进行改名操作的命令。tag是对image进行重命名，而rename是对容器进行重命名，所以使用时需要注意区分。

命令使用讲解：

```
Usage: Docker rename [OPTIONS] OLD_NAME NEW_NAME

Rename a container
```

这应该是Docker命令中最简单的命令之一了。对待如此简单的命令，我们使用一个示例来结束对它的讲解吧：

```
Docker@boot2Docker:~$ Docker ps
CONTAINER ID    IMAGE         COMMAND     CREATED    STATUS     P
6cd3885a6a40        192.168.73.12:5001/library/ubuntu-dev   "/bin
(Paused)   22/tcp, 80/tcp, 8080/tcp, 8088/tcp   desperate_lumiere

Docker@boot2Docker:~$ Docker rename desperate_lumiere mynewconta
#使用容器ID或者容器名称都可以
Docker@boot2Docker:~$ Docker ps
CONTAINER ID   IMAGE        COMMAND      CREATED     STATUS     P
6cd3885a6a40      192.168.73.12:5001/library/ubuntu-dev   "/bin/b
(Paused)   22/tcp, 80/tcp, 8080/tcp, 8088/tcp   mynewcontainer
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

SIGKILL信号的方式来关闭容器。

在Docker实际使用过程中，使用restart参数时，建议设置--time参数，以防止出现死锁情况。

17）Run命令

Run命令是Docker　CLI命令中最重要的一个命令。在Docker当中，只有通过Run命令才能创建并运行一个单独的容器。当Run创建容器时，会给目标容器分配独立运行资源，这些资源包括独立的文件系统资源、独立的网络访问资源和独立的进程资源。

在前面我们介绍过，构建镜像时可以指定此镜像默认的执行程序，默认导出网络端口和其他默认配置。但在使用Run命令创建容器时，这些默认值都会被Run命令所忽略；Run命令可以凭借对容器的最终“控制权”，在创建容器时直接修改此镜像所拥有的所有默认值。

也正因为Run命令可以取得容器最终控制权，所以Run命令拥有比其他命令更多的使用参数。

命令使用讲解：

```
Usage: Docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

-a, --attach=[]                 Attach to STDIN, STDOUT or STDERR
--add-host=[]                   Add a custom host-to-IP mapping (
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
--lxc-conf=[]                    Add custom lxc options
-m, --memory=                    Memory limit
--mac-address=                   Container MAC address (e.g. 92:d0
--memory-swap=                   Total memory (memory + swap), '-1
--name=                          Assign a name to the container
--net=bridge                     Set the Network mode for the cont
—oom-kill-disable=false          Disable OOM Killer
-P, --publish-all=false          Publish all exposed ports to rand
-p, --publish=[]                 Publish a container's port(s) to
--pid=                           PID namespace to use
--privileged=false               Give extended privileges to this
--read-only=false                Mount the container's root filesy
--restart=no                     Restart policy to apply when a co
--rm=false                       Automatically remove the containe
--security-opt=[]                Security Options
--sig-proxy=true                 Proxy received signals to the pro
-t, --tty=false                  Allocate a pseudo-TTY
-u, --user=                      Username or UID (format: <name|ui
--ulimit=[]                      Ulimit options
--uts=                           UTS namespace to use
-v, --volume=[]                  Bind mount a volume
--volumes-from=[]                Mount volumes from the specified
-w, --workdir=                   Working directory inside the cont
```

上面是run命令所有的参数。在使用这些参数时，需遵循下面用法：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

运行时，不能同时搭配使用--rm参数。

容器后台运行的参数为：

```
-d=false: Detached mode: Run container in the background, print r
```

```
Docker@boot2Docker:～$ Docker run -d 192.168.73.12:5001/ubuntu
3ce665663ee16c7d3f8adc1b47d8eea739ada529a77e58ab87a9c705c32496e2
```

当-d=true时，此容器将运行在后台模式，我们将Ubuntu容器放到后台运行：

当我们需要容器在前台运行时，只要不指定-d值就可以实现。此时Docker会在容器初始化时将控制台重定向至容器的标准输入（STDIN）、标准输出（STDOUT）和标准错误（STDERR）。Docker甚至可以模拟出一个TTY来传送信号量。而我们提到的这些事情，都是通过下面的参数来实现的：

```
-a=[]             : Attach to `STDIN`, `STDOUT` and/or `STDERR`
-t=false          : Allocate a pseudo-tty
--sig-proxy=true  : Proxify all received signal to the process (r
-i=false          : Keep STDIN open even if not attached
```

-a参数是用来控制标准输入/输出的。如果不使用-a参数，Docker默

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

容器运行时态设置包括：容器标识、PID设置、UTS设置、IPC设置、容器重启策略和容器清除六个方面。

标识一个容器通常有三种方式：

- 使用长UUID，例如f78375b1c487e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e74
- 使用短UUID，例如：f78375b1c487。
- 使用名称，例如：my_ubuntu。

UUID是由Docker Daemon自动生成的，用户无法修改，用户可以设置的只有容器名称。用户可以通过--name来设定，如下所示：

```
Docker@boot2Docker:~$ Docker run –it –name mybusy busybox
```

如果用户没有设定容器名称，Docker Daemon会自动生成一个随机字符串来标识此容器。

Docker考虑到生产环境中的自动化部署，提供了将容器ID输出到指定文本的功能，此功能通过--cidfile参数来指定写入的文本。当指定cidfile后，Docker run会在容器创建成功之后，将容器ID写入cidfile文本之中，然后就可以通过解析cidfile来获取容器ID，进行后续操作。下面就是使用cidfile的示例：

```
Docker@boot2Docker:~$ Docker run -d --cidfile=/tmp/busyboxID.log
68b1ee5961d7435ef4241dbe2e5f1f3a3940464860f408b3b50b2099cf255aa1
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
--uts=""  : Set the UTS namespace mode for the container,
                'host': use the host's UTS namespace inside the c
```

默认情况下，每个容器（包括那些不需要和外部网络进行链接的容器）都拥有其独自的UTS命名空间。如果uts="host"，那么就意味着此容器的uts命令设置将和主机的uts设置完全相同。

一般情况下，尤其是在生产环境中，不推荐使用--uts="host"。因为一旦使用了host选项，那么此容器就具有了修改主机配置文件的权限，这种情况对于生产环境来说十分危险。

IPC命令空间的设置用法如下：

```
--ipc=""  : Set the IPC mode for the container,
            'container:<name|id>': reuses another container's IPC
            'host': use the host's IPC namespace inside the conta
```

IPC(POSIX/SysV　IPC)命名空间提供了相互隔离的命名共享内存、信号灯变量和消息队列。

共享内存可以提高进程数据交互速度。共享内存一般用在database和高性能应用（C/OpenMPI, C++/using boost libraries）上，或者金融服务上。如果需要容器里面部署上述类型的应用，那么就应在多个容器中共享内存。

Docker允许每个容器通过--restart设定失败重启策略。如果容器发生

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

下面我们就讲解一下--rm参数。

默认情况下，当容器退出时，Docker Daemon不会删除此容器所保存的数据，这样有利于用户检查容器退出时刻所保持的数据信息。但如果用户不需要容器退出时还保存其内部信息，则可以通过--rm参数来通知Docker Daemon删除容器数据。但需注意，--rm参数和-d、--restart参数不能同时使用。以下是--rm参数的用法：

```
--rm=false: Automatically remove the container when it exits (inc
```

c）容器网络设置参数

容器的网络设置参数共包括：--dns、--net、--add-host和--mac-address四个参数，其用法分别如下：

```
--dns=[]              : Set custom dns servers for the container
--net="bridge"        : Set the Network mode for the container
            'bridge'  : creates a new network stack for the conta
            'none'    : no networking for this container
            'container:<name|id>': reuses another container netwo
            'host'    : use the host network stack inside the cor
--add-host=""         : Add a line to /etc/hosts (host:IP)
--mac-address=""      : Sets the container's Ethernet device's MA
```

Docker允许通过Docker run--net none来关闭网络接口，此时将关闭

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

<name|id>。

例如：当前有一个绑定了本地地址localhost的容器名为Redis，如果另外一个容器需要复用这个网络堆栈，则需要如下操作：

```
$ sudo Docker run -d --name redis example/redis --bind 127.0.0.1
# 通过redis的网络访问外部网络
$ sudo Docker run --rm -ti --net container:redis example/redis-cl
```

③host（主机模式）

当网络模式设置为host时，这个容器将完全共享host的网络堆栈。host所有的网络接口将完全对容器开放。容器的主机名也会存在于host的hostname中。这时，容器所有对外暴露的port和对其他容器的link，将完全失效。

④none

将网络模式设置为none时，这个容器将不允许访问任何外部网络。这个容器内部只会有一个loopback接口，而且不存在任何可以访问外部网络的路由信息。

d）容器运行时态CPU和内存设置参数

以下参数可以用来调整容器在运行时态下的CPU和内存资源，如表5-7所示。

表5-7　参数

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
$ Docker run -ti ubuntu:14.04 /bin/bash
```

如果设定ubuntu容器内存为300MB，但允许其尽可能多地使用内存：

```
$ Docker run -ti -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

在Docker中，如果对--memory-swap没有限制，那么默认是memory+swap=memory*2. 如下所示：

```
$ Docker run -ti -m 300M ubuntu:14.04 /bin/bash
```

此容器允许使用的内存上限为300MB，但内存+交换分区总量为300MB*2=600MB，则允许使用的交换分区上限为300MB。

我们还可以通过--memory-swap来设定允许使用的交互分区上限值，如下所示：

```
$ Docker run -ti -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash
```

此时设定ubuntu容器允许使用的内存为300MB，但允许使用的交换分区+内存上限为1GB，所以此容器中允许使用的交换分区为700MB。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

通过--memory-swappiness设定高权重的值，可以在一定程度上提高交互分区工作效率。

②CPU权重和时间片约束

默认情况下，所有容器都具有平等的CPU使用权重。但用户可以通过Run命令来为每个容器设定CPU使用权重。CPU权重取值从0到1024，可以通过-c或者-shares来修改默认值。

但这种权重值只有在多核CPU体系中才会生效。其原理是，当一个容器中的进程都处于IDLE状态时，其他容器可以使用此容器目前空闲的CPU时间片。但实际可用的CPU时间片则取决于当前主机系统中所运行的容器总数（假设宿主机当中没有执行其他进程）。

例如，当前主机系统中运行3个容器：容器A、容器B和容器C。容器A的CPU使用权重为1024，容器B和容器C的CPU使用权重分别为512。假设三个容器中的进程都处于100%使用CPU状态，那么容器A将得到50%的CPU时间片，而容器B和容器C将分别得到25%的CPU时间片。

如果此时增加了第四个容器：容器D，并且容器D的CPU使用权重为1024。那么容器A将得到33%的CPU时间片，容器B和容器C将分别得到16.5%的CPU时间片，而容器D同样得到33%的CPU时间片。

在一个多核CPU体系中，CPU时间片是所有CPU核总共时间片的总和。即便一个容器被设定使用小于100%的CPU，其也有可能会使用到单核CPU的100%，如下例所示。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
$ Docker run -ti --cpuset-cpus="1,3" ubuntu:14.04 /bin/bash
```

通过--cpuset-cpus可以设定此容器仅能使用1号CPU和3号CPU。如果设定值为：

```
$ Docker run -ti --cpuset-cpus="0-2" ubuntu:14.04 /bin/bash
```

这样的设定的含义是：可以使用0、1和2，这三个CPU。

e）容器运行时态内核权限和LXC参数设置

Docker可以通过run命令的--blkio-weight参数（取值范围为0到1000），为每个容器读写块设备的I/O资源进行限制（blkio）。默认情况下，每个容器的I/O限制都是相同的，均为500。同时Docker仅支持对块设备的I/O进行限制，不支持其他类型设备的I/O限制。

通过下面的命令，可以创建两个不同blkio阀值的容器：

```
$ Docker run -ti --name c1 --blkio-weight 300 ubuntu:14.04 /bin/b
$ Docker run -ti --name c2 --blkio-weight 600 ubuntu:14.04 /bin/b
```

通常情况下，Docker中的容器都是运行在用户态下，没有内核访问权限。例如，不能在上面所创建的容器中运行Docker Daemon。如果需要容器访问内核中的设备，那么就需要给此容器赋予内核特权。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

通过--privileged参数，配合--cap-add和--cap-drop参数，就可以增加或者删除容器所拥有的内核特权。默认情况下，每个容器都拥有对表5-8所示清单中的内核权限。

表5-8　容器的内核权限

SETPCAP	任意修改进程权限
SYS_MODULE	允许普通用户插入和删除内核模块
SYS_RAWIO	允许用户打开端口，并读取修改端口数据
SYS_PACCT	允许配置进程记账（审计）
SYS_ADMIN	允许执行系统管理任务，如挂载/卸载文件系统、设置磁盘配额、开/关交换设备和文件
SYS_NICE	允许提升优先级，设置其他进程的优先级
SYS_RESOURCE	忽略资源限制
SYS_TIME	允许改变系统时钟
SYS_TTY_CONFIG	允许配置 TTY 设备
MKNOD	允许使用 mknod 系统调用
AUDIT_WRITE	允许在内核安全日志中记录事件
AUDIT_CONTROL	允许控制内核安全模块
MAC_OVERRIDE	允许修改内核 MAC（强制访问控制）配置
MAC_ADMIN	允许控制内核 MAC（强制访问控制）模块
NET_ADMIN	允许执行网络管理任务：接口、防火墙和路由
SYSLOG	允许事件记录到系统日志中
CHOWN	允许改变文件的所有权
NET_RAW	允许使用原始（raw）套接字
DAC_OVERRIDE	忽略对文件的所有 DAC 访问限制
FOWNER	以最后操作的 UID 覆盖文件先前的 UID
DAC_READ_SEARCH	忽略所有对读、搜索操作的限制
FSETID	确保在文件被修改后不修改 setuid/setgid 位
KILL	允许对不属于自己的进程发送信号
SETGID	允许改变组 ID
SETUID	允许改变用户 ID
LINUX_IMMUTABLE	允许修改文件的不可修改（IMMUTABLE）和只添加（APPEND-ONLY）属性
NET_BIND_SERVICE	允许绑定到小于 1024 的端口
NET_BROADCAST	允许网络广播和多播访问（未使用）
IPC_LOCK	允许锁定共享内存片段
IPC_OWNER	忽略 IPC 所有权检查
SYS_CHROOT	允许使用 chroot()系统调用
SYS_PTRACE	允许跟踪任何进程
SYS_BOOT	允许重新启动系统
LEASE	允许在文件上建立租借锁
SETFCAP	允许在指定的程序上授权能力给其他程序
WAKE_ALARM	允许进程发送系统激活事件
BLOCK_SUSPEND	允许系统挂起

下面几个示例，则演示了如何增添系统内核权限。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
-rw-rw-r-- 1 1000 1000     16 Oct  8 00:09 .Dockerignore
-rwxrwxr-x 1 1000 1000    464 Oct  8 00:09 .drone.yml
drwxrwxr-x 1 1000 1000   4096 Dec  4 06:11 .git
-rw-rw-r-- 1 1000 1000    461 Dec  4 06:08 .gitignore
....
```

通过添加SYS_ADMIN权限，在容器内部可以挂载FUSE文件系统。

如果Docker启动时选择了lxc-driver（Docker-d--exec-driver=lxc），那么就可以使用--lxc-conf来设定LXC参数。但需要注意的是，未来host上面的Docker Daemon有可能不会使用LXC，所以这些参数有可能会包含一些没有实现的配置功能，所以用户在操作这些参数时必须要十分熟悉LXC。

> 注意
>
> 当使用--lxc-conf修改container参数后，Docker Daemon将不再维护这些参数。那么必须由用户自行进行维护。比如说，用户使用--lxc-conf修改了容器的IP地址，那么在/etc/hosts里面是不会自动体现的，需要用户自行维护。

f）其他可修改的参数

Docker run除了可以重新设定以上参数外，还可以重新设定日志驱动和Dockerfile中的一些默认值。下面我们就简单看一下如何重新设定这两项参数。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

以作为参数追加到ENTRYPOINT中。

②重置ENTRYPOINT参数：

```
--entrypoint="": Overwrite the default entrypoint set by the imag
```

ENTRYPOINT和COMMAND类似，它指定了当容器执行时，需要启动哪些进程。

ENTRYPOINT可以为容器设定默认启动行为，所以当容器启动时，可以执行任何一个二进制可执行程序。用户也可以通过COMMAND给这个ENTRYPOINT传递参数。如果需要在容器中执行其他进程，用户也可以指定其他ENTRYPOINT。

在下面的例子中，容器可以在启动时自动执行shell，然后启动其他进程。

```
$ sudo Docker run -i -t --entrypoint /bin/bash example/redis
#or two examples of how to pass more parameters to that ENTRYPOIN
$ sudo Docker run -i -t --entrypoint /bin/bash example/redis -c l
$ sudo Docker run -i -t --entrypoint /usr/bin/redis-cli example/r
```

③重置EXPOSE参数

Dockefile在网络方面除了提供一个EXPOSE之外，没有提供其他选项。下面这些参数可以覆盖Dockefile的EXPOSE默认值：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

问到这个container，同时Docker会在客户端的rq中设定一些环境变量来记录绑定的IP和PORT。

④重置ENV参数

HOME	用户基本环境变量
HOSTNAME	容器主机名
PATH	容器PATH变量：/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
TERM	如果容器启动了TTY，则会有TERM变量

当容器启动时，会自动在容器中初始化上面四个环境变量。

用户也可以通过-e来设定任意的环境变量，甚至覆盖已经存在的环境变量，也可以在Dockerfile中通过ENV来设定环境变量。

```
$ sudo Docker run -e "deep=purple" --rm ubuntu /bin/bash -c expor
declare -x HOME="/"
declare -x HOSTNAME="85bc26a0e200"
declare -x OLDPWD
declare -x PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bi
declare -x PWD="/"
declare -x SHLVL="1"
declare -x container="lxc"
declare -x deep="purple"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
declare -x PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bi
declare -x PWD="/"
declare -x REDIS_ALIAS_NAME="/distracted_wright/redis"
declare -x REDIS_ALIAS_PORT="tcp://172.17.0.32:6379"
declare -x REDIS_ALIAS_PORT_6379_TCP="tcp://172.17.0.32:6379"
declare -x REDIS_ALIAS_PORT_6379_TCP_ADDR="172.17.0.32"
declare -x REDIS_ALIAS_PORT_6379_TCP_PORT="6379"
declare -x REDIS_ALIAS_PORT_6379_TCP_PROTO="tcp"
declare -x SHLVL="1"
declare -x container="lxc"
#使用上面的内容，就可以链接客户端了。
$ sudo Docker run -i -t --rm --link redis-name:redis_alias --entr
-c '/redis-stable/src/redis-cli -h $REDIS_ALIAS_PORT_6379_TCP_ADD
172.17.0.32:6379>
```

Docker也会将这个alias的IP地址写入到/etc/hosts文件中，然后容器就可以通过别名来访问link后的容器。

```
$ sudo Docker run -d --name servicename busybox sleep 30
$ sudo Docker run -i -t --link servicename:servicealias busybox p
```

如果用户重启了源容器（servicename），相关联的容器也会同步更新/etc/hosts。

⑤重置volume参数

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

在Docker中执行rmi命令时，Docker需要查找出指定镜像的所有父文件层，然后逐一确认父类文件层是否被其他容器所复用。如果正在被使用，那么此镜像的删除动作将会被终止。只有当此镜像所有的父文件层都被确认没有被复用之后，此镜像才会被彻底删除。

rmi的使用方式同rm类似，如下所示：

```
Usage: Docker rmi [OPTIONS] IMAGE [IMAGE...]

Remove one or more images

  -f, --force=false     Force removal of the image
  --help=false          Print usage
  --no-prune=false      Do not delete untagged parents
```

--force表示强制删除。此处的强制仅限于一个镜像有多个标签的情况，例如下面的示例：

```
REPOSITORY       TAG                 IMAGE ID            CREATED
newcontainer     latest              3dc0cb6b08ca        6 days a
mycontainer      latest              3dc0cb6b08ca        6 days a
```

可看出这是同一个文件层，通过标签被标记成了两个不同的image。尽管从文件的角度上分析这是同一个文件，只是名称不同。但从Docker的角度会认为这是两个毫无关系的镜像。因此在使用Docker

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

19）start命令

start命令和restart命令有时容易混淆。Restart是对一个处于running状态的容器进行重启操作，同时包含两个动作：关闭和启动。而start则是对一个处于stop状态的容器进行操作，仅包含一个动作：启动。

那么是不是restart可以使用的场景中都可以使用start呢？其实不然，start和restart还是有些区别的。Restart只是单纯的关闭再启动，而start可以在启动时提供额外的功能。这个额外的功能就是可以完成容器与主机I/O流之间的重定向。

所以要分情况，根据实际需要调用restart或start命令。

下面我们看一下start的使用方式：

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

Start one or more stopped containers

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

通过参数可以看到，使用--attach和--interactive两个参数可以完成I/O重定向。下面我们通过几个示例来说明问题。

当我们不使用其他参数，直接使用start启动容器时，Docker输出如

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
2015-08-25 05:46:21 5 [Note] InnoDB: Database was not shutdown no
2015-08-25 05:46:21 5 [Note] InnoDB: Starting crash recovery.
2015-08-25 05:46:21 5 [Note] InnoDB: Reading tablespace informati
2015-08-25 05:46:21 5 [Note] InnoDB: Restoring possible half-writ
2015-08-25 05:46:21 5 [Note] InnoDB: from the doublewrite buffer.
2015-08-25 05:46:21 5 [Note] InnoDB: 128 rollback segment(s) are
2015-08-25 05:46:21 5 [Note] InnoDB: Waiting for purge to start
2015-08-25 05:46:21 5 [Note] InnoDB: 5.6.25 started; log sequence
2015-08-25 05:46:21 5 [Note] Server hostname (bind-address): '*';
2015-08-25 05:46:21 5 [Note] IPv6 is available.
```

可以看到使用了--attach参数后，Docker会把容器中的Stderr和Stdout重定向到主机的Stderr和Stdout流中。所以我们就可以看到MySQL容器所输出的启动日志。

如果我们使用了--interactive参数，情况就变成了下面的样子：

```
Docker@Docker-cloud-r720:~$ Docker start -i goofy_galileo
2015-08-25 05:49:56 0 [Note] mysqld (mysqld 5.6.25) starting as p
2015-08-25 05:49:56 5 [Note] Plugin 'FEDERATED' is disabled.
2015-08-25 05:49:56 5 [Note] InnoDB: Using atomics to ref count b
2015-08-25 05:49:56 5 [Note] InnoDB: The InnoDB memory heap is di
2015-08-25 05:49:56 5 [Note] InnoDB: Mutexes and rw_locks use GCC
2015-08-25 05:49:56 5 [Note] InnoDB: Memory barrier is not used
2015-08-25 05:49:56 5 [Note] InnoDB: Compressed tables use zlib 1
2015-08-25 05:49:56 5 [Note] InnoDB: Using Linux native AIO
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

的虚拟机；但从主机操作系统角度来看，所有正在运行的容器都是宿主环境中相互独立的进程，这些容器进程都在共享使用主机资源。因此在Docker使用过程中，用户有必要时刻关注容器所消耗的资源，以免因为资源耗尽而造成容器运行失败。

Docker提供了一个可以监控查看容器资源的命令，就是stats命令。目前stats命令可以统计CPU使用率、内存使用率和网络吞吐量这三方面信息。

使用stats命令可参考下面的使用方法：

```
Usage: Docker stats [OPTIONS] CONTAINER [CONTAINER...]

Display a live stream of one or more containers' resource usage s

  --help=false         Print usage
  --no-stream=false    Disable streaming stats and only pull the
```

stats命令支持统计多个容器的资源状况。当需要统计多个容器时，只需输入多个容器ID或者容器名称，中间使用空格分隔即可，例如：

```
Docker@Docker-cloud-r720:~$ Docker stats  8546477baf70  2660e913
CONTAINER     CPU %        MEM USAGE/LIMIT     MEM %           NE
2660e91327f6  0.05%        13.61 MB/33.69 GB   0.04%           1
8546477baf70  0.06%        9.363 MB/33.69 GB   0.03%           28
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

以支持更多的统计项。

21）stop命令

stop命令和start命令是功能相反的命令。在讲解restart命令时，也提到过restart=stop+start，所以restart中一部分用法和stop命令是重复的。掌握了restart命令的读者，也就一并掌握了stop命令的用法。

所以在本节中，我们仅讲解一下stop和kill的区别。

stop和kill命令的区别仅限于所发送的信号和超时时间的设置。kill命令默认发送SIGKILL信号，但也可以发送指定的信号。而stop命令，则只能发送SIGTERM和SIGKILL信号，不能发送其他信号。

stop命令可以指定等待时间，当超过等待时间后，将通过发送SIGKILL信号强行停止容器运行，而这点是kill命令所不具备的。

下面我们看一下stop命令的用法：

```
Usage: Docker stop [OPTIONS] CONTAINER [CONTAINER...]

Stop a running container by sending SIGTERM and then SIGKILL afte
grace period

  --help=false       Print usage
  -t, --time=10      Seconds to wait for stop before killing it
```

stop命令的用法和restart非常相似，也是通过空格分隔多个容器名

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

我们看一下tag的使用方法：

```
Usage: Docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/

Tag an image into a repository

  -f, --force=false    Force
  --help=false         Print usage
```

上面的用法有些抽象，下面我们通过几种情况来演示如何使用tag命令。

①直接命名：

```
#下面是busybox 镜像信息
busybox      latest              8c2e06607696        4 months ago
#直接命名，同时新名称中不包含仓库信息
Docker@boot2Docker:~$ Docker tag busybox newbusybox
#除名称不同外，其他信息完全相同
newbusybox   latest              8c2e06607696        4 months ago
busybox      latest              8c2e06607696        4 months ago
```

②包含仓库信息：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

④给无名镜像命名：

```
#47e39a22c47e是一个没有名称的镜像
<none>      <none>              47e39a22c47e        8 days ago
#通过tag命令可以给它命名
Docker@boot2Docker:~$ Docker tag 47e39a22c47e myname
myname     latest               47e39a22c47e        8 days ago
```

以上案例都只是修改了名称，如果需要修改tag标签，那么就像下面命令一样执行：

```
Docker@boot2Docker:~$ Docker tag newbusybox newbusybox:v1
newbusybox    latest               91c95931e552        4 months a
newbusybox    v1                   91c95931e552        4 months a
```

可以看到我们成功添加了tag为v1的镜像。

当我们给镜像命名成功之后，就可以通过pull命令向仓库中上传镜像了。但需要注意的是：如果镜像名称中不包含仓库信息，那么在执行pull时，Docker将默认向公共仓库上传镜像。

23）top命令

top命令属于统计容器资源状态的命令，但和stats命令有所区别。

stats命令统计的是CPU、内存和网络吞吐数据三项数据，而top统计

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
--max-requests  100  -k  gevent  --graceful-timeout  3600  -t  36
Docker_registry.wsgi:application
root                5666                8016                1
00:00:29            /usr/bin/python /usr/local/bin/gunicorn --acc
--max-requests  100  -k  gevent  --graceful-timeout  3600  -t  36
Docker_registry.wsgi:application
root                5756                8016                1
00:00:25            /usr/bin/python /usr/local/bin/gunicorn --acc
--max-requests  100  -k  gevent  --graceful-timeout  3600  -t  36
Docker_registry.wsgi:application
root                8016                3445                0
00:00:01            /usr/bin/python /usr/local/bin/gunicorn --acc
--max-requests  100  -k  gevent  --graceful-timeout  3600  -t  36
Docker_registry.wsgi:application
```

如果需要定制输出项，可以在容器名称后面输入ps命令相关选项，例如：

```
Docker@Docker-cloud-r720:～$ Docker top 5d35dd929e49 -e
PID                 TTY                 TIME                CMD
5610                ?                   00:00:30            gunic
5645                ?                   00:00:30            gunic
5666                ?                   00:00:30            gunic
5756                ?                   00:00:26            gunic
8016                ?                   00:00:01            gunic
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

在Linux操作系统中，凡是ps命令后面可使用的参数，都可以附加到容器名称之后，用来定制输出项。

通过top命令和stats命令配合，就可以完成对容器的基本监控。

24）version命令

version命令用来输出Docker当前版本信息。同info命令相比，version输出的内容仅侧重于当前Docker Daemon版本、Docker remote API版本等信息，不会显示内核版本等核心信息，因此需要读者根据实际情况灵活运用。

我们看一下version的使用方法：

```
Usage: Docker version [OPTIONS]

Show the Docker version information.

  --help=false       Print usage
```

执行version命令后，Docker将会返回两部分版本信息：server信息和client信息（因为Docker本身既是server端，也是client端），如下所示：

```
Client version: 1.7.1
Client API version: 1.19
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
Usage: Docker pull [OPTIONS] NAME[:TAG|@DIGEST]

Pull an image or a repository from the registry

  -a, --all-tags=false    Download all tagged images in the repos
  --help=false              Print usage
```

默认情况下，只要给pull命令提供一个有效的image　name就可以。例如：

```
$Docker pull ubuntu
```

通过这条命令，Docker就会从官方仓库中下载Ubuntu镜像。

在5.1节中我们提到过镜像名称与镜像标签之间的区别。当用户提供的镜像名称不包含私有仓库信息时，Docker将会在官方仓库中检索是否存在相同名称的镜像。如果在目标仓库中此镜像存在多个标签，那么默认将下载标签为latest的镜像。

私有仓库是什么意思呢？简单来说，如果镜像名称中存在类似于XXX.XXXX.XXX（也有可能是域名）这样格式的字符串，就可以认为目标仓库是私有仓库。例如，我们在本书中经常使用到的192.168.73.12:5001/library/ubuntu-dev就是私有仓库中的一个镜像，其中仓库地址是：192.168.73.12:5001。如果镜像名称中没有XXX.XXX.XXX格式的字符串，那么Docker就认为用户需要从公有仓库中下载镜像。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
Push an image or a repository to the registry
```

在使用push命令时，如果镜像名称中包含私有仓库信息，那么push命令会将此镜像上传到私有仓库。如果不包含私有仓库信息，那么push命令会将镜像默认上传到Docker官方仓库。

push命令中的私有仓库判断标准和pull命令中的私有仓库判断标准完全一致。此处不再赘述。

下面我们用一个示例来结束对push命令的讲解：

```
Docker@Docker-cloud-r720:～$ Docker push 192.168.73.12:5001/libra
#当执行完这个命令后，Docker将向私有仓库192.168.73.12:5001上传

Docker@Docker-cloud-r720:～$ Docker push library/ubuntu-dev
#当执行完这个命令，Docker将向公有仓库上传
```

3. Search命令

Docker官方目前存储了相当多的镜像，而且随着Docker的迅猛发展，各大主流产品厂商也在不断持续提供官方版本的产品镜像。

由此就带来一个问题：用户如何在如此庞大的镜像仓库中找到所需要的镜像呢？还好Docker官方提供了一个镜像搜索工具，就是search命令。

通过search命令，用户可以在指定的仓库中检索到所需的镜像。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
docker@docker-cloud-r720:~$ docker search untu
NAME                                    DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
ubuntu                                  Ubuntu is a Debian-based Linux operating s...   2124      [OK]
ansible/ubuntu14.04-ansible             Ubuntu 14.04 LTS with ansible                   64                   [OK]
ubuntu-upstart                          Upstart is an event-based replacement for ...   31        [OK]
torusware/speedus-ubuntu                Always updated official Ubuntu docker imag...   25                   [OK]
dorowu/ubuntu-desktop-lxde-vnc          Ubuntu with openssh-server and NoVNC on po...   23                   [OK]
sequenceiq/hadoop-ubuntu                An easy way to try Hadoop on Ubuntu             20                   [OK]
tleyden5iwx/ubuntu-cuda                 Ubuntu 14.04 with CUDA drivers pre-installed    17                   [OK]
ubuntu-debootstrap                      debootstrap --variant=minbase --components...   12        [OK]
neurodebian                             NeuroDebian provides neuroscience research...   11        [OK]
rastasheep/ubuntu-sshd                  Dockerized SSH service, built on top of of...   5                    [OK]
fike/ubuntu-postgresql                  PostgreSQL 9.4 beta until 9.0 version runn...   4                    [OK]
nimmis/ubuntu                           This is a docker images different LTS vers...   3                    [OK]
n3ziniuka5/ubuntu-oracle-jdk            Ubuntu with Oracle JDK. Check tags for ver...   3                    [OK]
maxexcloo/ubuntu                        Docker base image built on Ubuntu with Sup...   2                    [OK]
armbuild/ubuntu-debootstrap             ARMHF port of ubuntu-debootstrap                2                    [OK]
nuagebec/ubuntu                         Simple always updated Ubuntu docker images...   2                    [OK]
seetheprogress/ubuntu                   Ubuntu image provided by seetheprogress us...   1                    [OK]
densuke/ubuntu-jp-remix                 Ubuntu Linux鉢璽鞴璽remix槎ム慟鉢c企                                1                   [OK]
isuper/base-ubuntu                      This is just a small and clean base Ubuntu...   1                    [OK]
dreamcat4/tvh.ubuntu.build.deps         Builder for tvheadend's build dependancies...   1                    [OK]
gbevan/ubuntu-foreman                   Ubuntu based Puppet Foreman with Ansible        0                    [OK]
densuke/ubuntu-supervisor               densuke/ubuntu-jp-remix:trusty 涓嬪緢 supe...          0                    [OK]
jianhuiz/ubuntu-docker                  ubuntu images that runs in nova-docker          0                    [OK]
vicamo/ubuntu-phablet-jiexi             Dockerfile for developing Ubuntu Jiexi PDK.     0                    [OK]
redswarm/ubuntu-dronesim-autobuild      This repository contains an Ubuntu image w...   0                    [OK]
```

读者可能会发现上面两个结果中都是25个image，那为什么说search untu会匹配出更多的结果呢？这是因为search命令最多只会返回25条结果。如果两个结果数量都大于25，那么会显示相同的结果。但从比较结果范围来看，会发现匹配ubuntu的结果集是untu结果集的子集，从这方面可以说明，使用untu进行匹配可以获取到更多的结果。

通过上面的结果，也可以看到search返回的数据包含有STARS、OFFICIAL和AUTOMATED三列。这三列分别表示星级、是否官方版本、是否为Docker自动化构建。通过--stars和--automated这两个参数可以分别针对STARS和AUTOMATED两列进行筛选。例如，我们需要检索星级比较高的镜像，如下所示：

```
Docker@Docker-cloud-r720:~$ Docker search -s 100 ubuntu
NAME      DESCRIPTION                                     STARS     OFFICIAL
ubuntu    Ubuntu is a Debian-based Linux operating s...   2124
```

我们只要星级高于100的记录（星级越高，代表镜像质量越高），因此只找到这一个镜像记录。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
baselibrary/ubuntu                ThoughtWorks Ubuntu Docker Imag
isuper/base-ubuntu                This is just a small and clean
cpuguy83/ubuntu
gbevan/ubuntu-foreman             Ubuntu based Puppet Foreman wit
zoni/ubuntu
partlab/ubuntu                    Simple Ubuntu Docker images
vicamo/ubuntu-phablet-jiexi       Dockerfile for developing Ubunt
densuke/ubuntu-supervisor         densuke/ubuntu-jp-remix:trusty
```

通过上面命令，我们就可以筛选出来属于Docker自动化构建的镜像。

最后的--no-trunc参数，在其他命令中也见过，就是为了方便用户查看会截取部分字段的数据。如果设置为true，那么输出如下：

```
ubuntu    Ubuntu is a Debian-based Linux operating system based c
ubuntu-upstart    Upstart is an event-based replacement for the /
processes at boot    31    [OK]
torusware/speedus-ubuntu    Always updated official Ubuntu Docker
acceleration software    25    [OK]
sequenceiq/hadoop-ubuntu    An easy way to try Hadoop on Ubuntu
tleyden5iwx/ubuntu-cuda    Ubuntu 14.04 with CUDA drivers pre-ins
ubuntu-debootstrap    debootstrap --variant=minbase --components=
--include=inetutils-ping,iproute2 <suite>    12    [OK]
guilhem/vagrant-ubuntu        10    [OK]
rastasheep/ubuntu-sshd    Dockerized SSH service, built on top of
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

5.3 Docker安全命令

1. login命令

这里的login并非是登录到Docker中，因为本地Docker不需要登录，只要有执行权限就可以正常使用。

当使用login时，Docker会将用户信息转发到目标仓库中。登录成功后，就可以正常的进行push操作了。

命令使用讲解：

```
Usage: Docker login [OPTIONS] [SERVER]

Register or log in to a Docker registry server, if no server is
specified "https://index.Docker.io/v1/" is the default.

  -e, --email=        Email
  --help=false        Print usage
  -p, --password=     Password
  -u, --username=     Username
```

从上面的使用方法可以看出，默认情况下，Docker会登录到https://index.Docker.io/v1/。如果需要登录到其他仓库，用户可以在命令后面通过指定仓库URL的方式来修改。例如，用户要登录到本地仓库中，就执行下面的命令：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

5.4 Docker资源命令

1. export命令

export是一个用来导出容器的命令。我们知道，在Docker体系当中，所有的应用都是通过镜像来进行分发的，因为每一个镜像都遵从Docker所定义的数据规范，因此每个镜像都可以顺利在Docker中运行。

但容器和镜像有所不同，因为镜像中所使用的文件层都是只读文件层，而容器中除了加载镜像所定义的只读文件层外，还需要额外添加一个读写文件层。因为每个容器所产生的数据都各不相同，所以无法按照分发镜像的方法去分发容器。

那么如果用户需要移植容器，又该如何操作呢？

虽然Docker不建议移植容器，但Docker也预留了export命令来导出容器。

命令使用讲解

通过export命令，Docker将指定的容器导出为tar包，再通过相配套的import命令将导出的tar包导入到另外一个Docker环境中，借此完成容器的移植。下面我们看一下export命令如何使用：

```
Usage: Docker export [OPTIONS] CONTAINER

  Export the contents of a filesystem to a tar archive (streamed
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

创建成功之后，查看一下容器中mountDIR目录内容：

```
root@e96e23805210:/mountDIR# ls
autoUpdate  config.json  config.json.bak  doc.go  email.json  mai
```

和当前目录的文件是一致的。

然后执行导出命令，并且将导出的tar包命名为export.tar：

```
Docker@Docker-cloud-r720:/tmp/Docker$ Docker export -o export.tar
```

当解压这个tar包后，会发现目录和容器之中的目录是一致的：

```
Docker@Docker-cloud-r720:/tmp/Docker$ ls
bin  boot  dev  etc  export.tar  home  lib  lib64  media  mnt  mo
srv  sys  tmp  usr  var
```

但是容器中挂载的卷组：

```
root@e96e23805210:/mountDIR# df -h
Filesystem      Size  Used Avail Use% Mounted on
rootfs          1.8T   49G  1.7T    3% /
none            1.8T   49G  1.7T    3% /
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
$ Docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar c
```

②当新容器启动时，将会把继承自dbdata容器的所有文件系统都归档到backup目录中。再创建一个用来保存备份数据的容器，叫做dbdata2，在启动时将当前目录挂载到容器中。

```
$ Docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

③最后一步是将备份的数据解压到dbdata2中。

```
$ Docker run --volumes-from dbdata2 -v $(pwd):/backup ubuntu cd /
```

经过上面这三步，就完成了数据“乾坤大挪移”。

2. images命令

images命令也是Docker　CLI中经常使用的一个命令。通过images，用户可以查看到目前所有的镜像信息，包括镜像何时被创建、镜像ID或者镜像tag、镜像大小等信息。

当用户通过images命令拿到镜像ID或者镜像tag之后，用户就可以使用这些信息来创建容器，进而进行后续操作。因此，images命令是Docker CLI命令中使用率非常高的一个命令。

images命令使用率非常高，但这个命令使用非常简单，例如：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

就可以将image中保存的comment信息显示出来。

--filter是筛选参数，采用key=value的方式。目前可使用的key只有dangling和label。dangling表示悬空，意思是查找没有父文件层的临时文件层。这些文件层是如何产生的呢？主要的产生方式是在构建image时，修改了Dockerfile中某一条命令，造成此命令之后所有临时文件层都失效。虽然Docker不会使用这些失效临时文件层，但又没有删除，长此以往就产生了很多没有父文件层的镜像数据。通过dangling，就可以筛选出这些处于悬空状态的镜像，然后一次性的删除，例如：

```
$ Docker images --filter "dangling=true"
REPOSITORY          TAG                 IMAGE ID            CREAT
<none>              <none>              8abc22fbb042        4 wee
<none>              <none>              48e5f45168b9        4 wee
<none>              <none>              bf747efa0e2f        4 wee
<none>              <none>              980fe10e5736        12 we
<none>              <none>              dea752e4e117        12 we
<none>              <none>              511136ea3c5a        8 mor
```

我们可以把镜像ID都找出来然后一一删除，也可以使用下面的组合命令一次性的删除：

```
$ Docker rmi $(Docker images -f "dangling=true" -q)

8abc22fbb042
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
busybox       latest          8c2e06607696bd4afb3d03b687e361cc43c
months ago        2.433 MB
192.168.73.12:5001/library/ubuntu-dev      latest
e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3c17836d84c735b
```

```
root@boot2Docker:~# Docker images --no-trunc=false
REPOSITORY                    TAG                IMAGE ID
newcontainer                  latest             3dc0cb6b08ca
<none>                        <none>             47e39a22c47e
baseconimage                  V1.1               f95964dc431d
baseconimage                  V1.0               d318c9e3d2e5
192.168.73.12:5001/mysql      latest             1eba0810df1f
hello-world                   latest             91c95931e552
busybox                       latest             8c2e06607696
192.168.73.12:5001/library/ubuntu-dev latest   e9453316c630
```

看出来了吧，--no-trunc可以用来截取镜像ID。全ID有的时候太长，不方便使用，这个时候使用--no-trunc将长ID截取成短ID，就方便多了。

3. import命令

import和export是功能相反的命令。export是将容器导出为一个tar文件格式的归档文件，当用户将这个归档文件拷贝到其他Docker环境以后，就需要使用import命令执行导入动作了。

导入归档文件只是import命令其中一项功能，import还可以通过网

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

了一个tar归档文件的导入。那tar文件是被作为镜像导入的呢，还是被作为容器导入的呢？其实，Docker是将tar归档文件视为镜像进行处理的。

因为容器相对于镜像而言，是一个动态活动的实体。Export也只是导出了容器当时的文件系统，容器内部没有落地的数据，因而并没有导出。所以Docker在执行import时，也只能把文件系统导入到Docker环境中。

当导入成功之后，用户就可以凭借其返回的镜像ID创建容器进行后续操作了。

在上面命令执行成功后，有一个问题就是Docker会创建新的镜像ID，但镜像tag却是<none>。这是因为在镜像所有属性中，ID是必填项，由Docker Daemon负责维护；而tag是选填项，由用户自行维护。如果用户没有给定tag值，则此属性默认为<none>。

通过下面的方式，用户可以指定image tag：

```
$ cat exampleimage.tgz | Docker import - exampleimagelocal:new
```

加黑部分的“exampleimagelocal:new”就是指定的tag。在使用“-”时，还可以结合Linux的一些小技巧，例如：

```
$ tar -c . | Docker import – exampleimagedir
```

读者可以充分发挥想象力，写出各种脚本来完成import。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

load命令是加载镜像归档文件的命令，这个命令和import命令相近，但又有一些区别。这两个命令都属于加载归档文件的命令，但import加载容器导出的归档文件，而load则加载镜像归档文件。

同时，使用export导出的归档文件不包括文件系统的历史记录，只有最近一次的读写层数据。所以import导入容器归档文件时，并不会维护文件系统历史，而load命令导入image归档文件时会维护历史记录。

因此虽说都是导入，但load和import适用范围不同。用户使用时需要注意。

下面的是load的使用方法：

```
Usage: Docker load [OPTIONS]

Load an image from a tar archive on STDIN

  --help=false         Print usage
  -i, --input=         Read from a tar archive file, instead of S
```

从使用方法可以看出，load支持直接读取文件，也支持从stdin读取文件。其实对于load来说，只要获取到归档文件的二进制流就可以，所以这两种方式没有本质性差异。

当通过STDIN加载image时，就像下面这样操作：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

5. pause命令

pause命令可以用来暂停容器进程。当用户暂时不需要容器继续工作，但又不能关闭此容器时，就可以使用pause暂时停止此容器工作。当容器处于pause状态时，其内部所有的进程也都同时处于pause状态，除非使用unpause唤醒，否则此容器将会一直处于pause状态。

Pause命令使用方式如下：

```
Usage: Docker pause [OPTIONS] CONTAINER [CONTAINER...]

Pause all processes within a container

  --help=false       Print usage
```

如果需要同时暂停多个容器时，使用空格隔开多个容器ID即可。

在执行pause命令时，Docker将向指定容器发送SIGSTOP信号。所以一般情况下，容器中部署的应用不应该对SIGSTOP进行捕获处理，这点需要读者在应用开发时多加注意。

当pause执行成功之后，用户可以通过ps查看到当前容器的状态已经变更为Paused：

```
Docker@boot2Docker:~$ Docker ps
CONTAINER ID        IMAGE        COMMAND        CREATED        ST
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

port命令返回两条记录，意思是test容器对外暴露了两个端口：7890端口和9876端口。

如果此容器暴露了非常多的端口，而用户只想知道指定的端口是否暴露，那么可以在port命令后面指定想知道的端口来完成筛选：

```
$ Docker port test 7890/tcp
0.0.0.0:4321
```

在Docker容器中，端口+协议共同标记了一个网络端口资源。所以筛选端口信息时，端口号和协议名称必须都要正确，否则会出现下面的错误：

```
$ Docker port test 7890/udp
2014/06/24 11:53:36 Error: No public port '7890/udp' published fo
```

如果不关心协议名称，只需要端口号，那么可以忽略后面的协议名称，例如：

```
$ Docker port test 7890
0.0.0.0:4321
```

通过port命令，用户可以随时获取到容器端口信息，方便其他容器网络通信。同时也可以通过port命令来验证此容器是否启动正常，应该

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

是最简单的删除命令：

```
$ Docker rm ubuntu
```

当执行完这条命令后，与Ubuntu容器相关的所有数据将在Docker环境中消失。

上面的命令和下面的命令是等效的：

```
$ Docker rm /ubuntu
```

这两条命令完全相同，可以通用（虽然不清楚Docker为什么要提供这样的输入方式，但既然提供了读者就可以放心的使用）。

上面那条是最简单的删除命令，但只能删除为非running状态的容器。如果已经确认当前容器可以删除，则可通过指定--force参数的方式直接强制删除，例如：

```
$ Docker rm -f ubuntu
```

这样无论Ubuntu容器处于什么状态都可以删除。

rm命令不仅能删除容器，还能附带删除网络连接和挂载的卷组数据。但是很遗憾，在Docker　1.7版本中--link存在bug，rm命令无法删除

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  --help=false       Print usage
  -o, --output=      Write to an file, instead of STDOUT
```

可以看出save的使用方式和export非常相近。

save命令可以导出多个镜像的文件系统，这里的多个镜像有两层含义：

a）多个互无关系的镜像。此时在输入参数时允许输入多个镜像名称或者ID，中间以空格隔开，例如：

```
$Docker save ubuntu mysql tomcat
```

上面三个镜像毫无关系，但可以同时导出。

b）同一个镜像的不同tag，例如：

```
$Docker save ubuntu
```

如果在当前环境中存在Ubuntu的多个tag，比如Ubuntu：latest、Ubuntu：v1等，这时Docker Daemon会把这些具有相同名称但tag不同的镜像同时导出。

--output参数的使用方式和export命令中--output参数的使用方式完全相同。因为save命令默认输出归档文件二进制流，因此可以也通过指定-

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

命令使用讲解：

```
Usage: Docker wait [OPTIONS] CONTAINER [CONTAINER...]

Block until a container stops, then print its exit code.
```

使用wait命令仅需提供容器名称或者容器ID即可，而且wait支持等待多个容器，只需使用空格分隔容器名称。

在实际使用过程中，经常会在脚本中使用wait命令等待指定的容器退出后，再执行后续的操作。wait命令也会输出指定容器的退出响应码，所以可以在脚本中通过获取响应码来判断容器退出是否成功。

5.5 Docker RestFul命令

Docker Remote API需要了解的知识点

在使用Docker Remote API之前，用户需要了解以下内容：

①默认情况下，Docker只会监听本地docker.socket文件（位置在unix:///var/run/docker.socket），客户端如果需要访问这个文件，则需要当前用户具有root权限，或者已经加入到Docker用户组中。

②Docker Daemon启用安全认证后，使用curl或者wget测试Rest API时，需要添加额外的认证参数。例如，通过curl访问boot2Docker的

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

下：

```
{"username": "string", "password": "string", "email": "string", "
```

在上面的JSON字符串中，auth必须留空。serveraddress是一个domain/IP格式的字符串集合，同时不需要添加诸如http的协议名称。上述引号使用的都是双引号。

⑥在v1.19版本的Remote API中，所有的API都可以归纳为三类操作：

- 对Containers的操作
- 对Images的操作
- 其他辅助操作

⑦Docker Daemon默认返回的都是JSON字符串。

```
http://192.168.73.12:2376/containers/json.
```

⑧Remote API使用方法：“GET/contaienrs/json”表示一个API，其中GET表示这个API的请求类型为GET。/contaienrs/json是路径，这是具体的命令。如果需要使用此API，则需要client将HTTP request类型设置为GET，然后发起如下请求：

Daemon接收到请求后，会根据后面的URL PATH，将此请求路由至具体的函数模块来进行处理，然后将响应发回给client。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

all	布尔值，接受 1/True/true，或者 0/False/false，默认为 false。 表示是否显示全部状态的 container。当为 false 时，只显示 running 状态的 container
limit	整型值，用于最后 *N* 个创建的 container
since	字符串，可输入 container ID、short ID 或者 container name。 表示查询在输入的 container 之后所创建的 container 信息
before	字符串，可输入 container ID、short ID 或者 container name。 作用与 since 参数相反，表示需要查询在此 container 之前所创建的 container 信息
size	布尔值，接受 1/True/true，或者 0/False/false，默认为 false。 表示是否在 container 信息中显示 size 数据
filters	JSON 格式字符串，用来筛选 Daemon 所返回的 container 信息。 目前有效筛选项为 exited = <int>、status=(restarting|running|paused|exited)、label=key 或者 key=value

③API示例

现在开始使用这个API来查询当前环境中所有的容器。首先演示all参数的使用，在前面的参数说明中提到，当all=false时，此API只显示running状态的容器；所以我们先发送如下rest请求，来查询所有running状态的container：

```
192.168.73.12:2376/containers/json
```

Daemon会返回如下信息：

```
[
    {
        "Id": "dd9f089baf26199ab305bf5b1f3213e8795ddb3a81ab609da
        "Names": [
            "/modeler_0"
        ],
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
                "PrivatePort": 3306,
                "PublicPort": 3306,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Up 44 hours"
    }
]
```

从加黑部分可以看出，这两个容器已经运行很长时间了。当all置为true时，又会出现什么情况呢？我们发送如下rest请求：

```
192.168.73.12:2376/containers/json?all=1
```

这次Daemon返回的数据多了一些，具体如下：

```
[
    {
        "Id": "960bc9c6d68cea120e183e82e504b5fb276984ac074036ca9a
        "Names": [
            "/happy_einstein"
        ],
        "Image": "hello-world",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Id": "890981b949fb5609da64620b243b4bfe8b3dd4784129b5370
        "Names": [
            "/api_mysql_0",
            "/modeler_0/api_mysql_0"
        ],
        "Image": "192.168.73.12:5001/mysql",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1436246719,
        "Ports": [
            {
                "IP": "0.0.0.0",
                "PrivatePort": 3306,
                "PublicPort": 3306,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Up 45 hours"
    }
]
```

由加黑部分可以看出，已经退出的容器可以通过all=true筛选出来。

--limit参数可以用来筛选最后N个创建的容器信息。使用此参数时，Docker会显示所有状态的容器，包括正在运行、正常退出的和非正常退出的。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
]
```

当需要显示多个container时，是否会按照顺序排列呢？我们输入limit=2来验证一下：

```
192.168.73.12:2376/containers/json?limit=2
```

可以看到Daemon返回了两个容器信息，并且也做了排序：

```
[
    {
        "Id": "4d937b882748bbe1d9a9ab5d56b79db8a16ec913a233cf197
        "Names": [
            "/romantic_pare"
        ],
        "Image": "1eba0810df1f",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437387832,
        "Ports": [],
        "Labels": {},
        "Status": "Exited (1) 6 minutes ago"
    },
    {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

--since和--before是两个具有相反功能的参数，--since参数可用来查询指定容器创建之后所创建的容器信息，例如发送如下rest请求：

```
192.168.73.12:2376/containers/json?since=678836752f08
```

此时Daemon将返回如下信息：

```
[
    {
        "Id": "3aeecf4236fb2c15686bcf5bfa98644e660718a3c9aa7edb6a
        "Names": [
            "/determined_rosalind"
        ],
        "Image": "1eba0810df1f",
        "Command": "/entrypoint.sh /bin/bash",
        "Created": 1437388790,
        "Ports": [
            {
                "PrivatePort": 3306,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Exited (0) 17 minutes ago"
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Labels": {},
        "Status": "Exited (0) 18 minutes ago"
    },
    {
        "Id": "678836752f086da8d4c90297f8e50a019645b5bae9a1ff4651
        "Names": [
            "/suspicious_poitras"
        ],
        "Image": "hello-world",
        "Command": "/bin/bash",
        "Created": 1437388765,
        "Ports": [],
        "Labels": {},
        "Status": ""
    },
    {
        "Id": "4d937b882748bbe1d9a9ab5d56b79db8a16ec913a233cf1972
        "Names": [
            "/romantic_pare"
        ],
        "Image": "1eba0810df1f",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437387832,
        "Ports": [],
        "Labels": {},
        "Status": "Exited (1) 14 hours ago"
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        ],
        "Labels": {},
        "Status": "Exited (0) 20 minutes ago"
    },
    {
        "Id": "678836752f086da8d4c90297f8e50a019645b5bae9a1ff465
        "Names": [
            "/suspicious_poitras"
        ],
        "Image": "hello-world",
        "Command": "/bin/bash",
        "Created": 1437388765,
        "Ports": [],
        "Labels": {},
        "Status": ""
    },
    {
        "Id": "4d937b882748bbe1d9a9ab5d56b79db8a16ec913a233cf197
        "Names": [
            "/romantic_pare"
        ],
        "Image": "1eba0810df1f",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437387832,
        "Ports": [],
        "Labels": {},
        "Status": "Exited (1) 14 hours ago"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    {
        "Id": "4d937b882748bbe1d9a9ab5d56b79db8a16ec913a233cf1972
        "Names": [
            "/romantic_pare"
        ],
        "Image": "1eba0810df1f",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437387832,
        "Ports": [],
        "Labels": {},
        "Status": "Exited (1) 14 hours ago"
    },
    {
        "Id": "81575c42517654a381b904328995701f42e5e6b636b47e213(
        "Names": [
            "/goofy_bartik"
        ],
        "Image": "hello-world",
        "Command": "/hello",
        "Created": 1437377225,
        "Ports": [],
        "Labels": {},
        "Status": "Exited (0) 17 hours ago"
    }
]
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Names": [
            "/romantic_pare"
        ],
        "Image": "1eba0810df1f",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437387832,
        "Ports": [],
        "SizeRootFs": 281786969,
        "Labels": {},
        "Status": "Exited (1) 14 hours ago"
    },
    {
        "Id": "81575c42517654a381b904328995701f42e5e6b636b47e213(
        "Names": [
            "/goofy_bartik"
        ],
        "Image": "hello-world",
        "Command": "/hello",
        "Created": 1437377225,
        "Ports": [],
        "SizeRootFs": 922,
        "Labels": {},
        "Status": "Exited (0) 17 hours ago"
    }
]
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Id": "3aeecf4236fb2c15686bcf5bfa98644e660718a3c9aa7edb6a
        "Names": [
            "/determined_rosalind"
        ],
        "Image": "1eba0810df1f",
        "Command": "/entrypoint.sh /bin/bash",
        "Created": 1437388790,
        "Ports": [
            {
                "PrivatePort": 3306,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Exited (0) 50 minutes ago"
    },
    {
        "Id": "81575c42517654a381b904328995701f42e5e6b636b47e213
        "Names": [
            "/goofy_bartik"
        ],
        "Image": "hello-world",
        "Command": "/hello",
        "Created": 1437377225,
        "Ports": [],
        "Labels": {},
        "Status": "Exited (0) 17 hours ago"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

status=(restarting|running|paused|exited)可查询状态为上述几项的容器信息，比如发送：

```
192.168.73.12:2376/containers/json?filters={ "status": [ "exited"
```

Daemon将返回：

```
[
    {
        "Id": "3aeecf4236fb2c15686bcf5bfa98644e660718a3c9aa7edb6a
        "Names": [
            "/determined_rosalind"
        ],
        "Image": "1eba0810df1f",
        "Command": "/entrypoint.sh /bin/bash",
        "Created": 1437388790,
        "Ports": [
            {
                "PrivatePort": 3306,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Exited (0) 56 minutes ago"
    },
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            "/goofy_bartik"
        ],
        "Image": "hello-world",
        "Command": "/hello",
        "Created": 1437377225,
        "Ports": [],
        "Labels": {},
        "Status": "Exited (0) 17 hours ago"
    }
]
```

而label=key或者key=value则可以查询指定label的容器信息。

此外，filter还支持按照name和id进行筛选。使用此两项筛选参数时，需要满足容器处于running状态，或者查询时指定all=true或limit值>0。

2. 创建容器

API	/containers/create
API 类型	POST
API 参数	无
附加参数	Config
API 返回值	201：创建成功 404：container 不存在 406：container 没有 running 导致无法加载 500：服务器处理错误

①API说明

第一个API可以列出所有当前环境中的容器信息。但用户又该如何

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
      "Cmd": [],
      "Entrypoint": "",
      "Image": "ubuntu",
      "Labels": {},
      "Volumes": { },
      "WorkingDir": "",
      "NetworkDisabled": false,
      "MacAddress": "",
      "ExposedPorts": {},
      "HostConfig": {
        "Binds": [],
        "Links": [],
        "LxcConf": {},
        "Memory": 0,
        "MemorySwap": 0,
        "CpuShares": 512,
        "CpuPeriod": 100000,
        "CpusetCpus": "0,1",
        "CpusetMems": "0,1",
        "BlkioWeight": 300,
        "OomKillDisable": false,
        "PortBindings": { },
        "PublishAllPorts": false,
        "Privileged": false,
        "ReadonlyRootfs": false,
        "Dns": [],
        "DnsSearch": [""],
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

Hostname		字符串，用来设定 container 的 hostname。 若为空，Daemon 将以 containerID 的前 12 位作为 hostname。注意不是 container 的 name
Domainname		字符串，用来设定 container 的域名，用来设置跨域访问
User		字符串，用来设定 container 执行时，以何用户来执行后续 command. 若为空，则以此 image build 时设定的 user 执行
AttachStdin		布尔值，用来设定是否将 container 中的 I/O 重定向至 Stdin
AttachStdout		布尔值，用来设定是否将 container 的 I/O 重定向至 Stdout
AttachStderr		布尔值，用来设定是否将 container 的 I/O 重定向至 Stderr
Tty		布尔值，用来设定是否需要在 container 中开启 TTY
OpenStdin		布尔值，用来设定是否需要在 container 中启用 Stdin
StdinOnce		布尔值，用来设定当此 container 的 client 关闭连接时，是否需要关闭 Stdin
Env		字符串数组，用来设定此 container 的环境变量
Cmd		字符串数组，用来设定此 container 启动时，需要执行的命令。如果 EntryPoint 值为空的话，container 将会执行 Cmd 中的命令。如果 EntryPoint 有值，Daemon 将会把 Cmd 中的值作为参数传入 EntryPoint 中执行
Entrypoint		字符串数组，用来设定此 container 启动后，后续的 command 如何执行。如果 EntryPoint 值不为空，那么后续所有的 command 都将以参数的形式，交由 EntryPoint 中的命令执行
Image		字符串，用来设定此 container 是以哪个 image 为基础而创建的。注意，此 image 只会在本地中查找，如果本地没有，则会返回错误，不会向 remote registry 查询
Labels		字符串数组，用来标示 container。 可以在 list API 中通过 filters 查询
Volumes		创建一个挂载点用于共享目录
WorkingDir		字符串，用来设定在 container 中，后续 command 的 HOME 目录
NetworkDisabled		布尔值，如果为 True，则将会屏蔽此 container 的所有网络连接
MacAddress		字符串，用来设定此 container 的 MAC 值。注意，Daemon 不会校验此 MAC 是否符合格式，需要用户自行保证此 MAC 是有效的 MAC 值
ExposedPorts		JSON 对象，用来设定此 container 中向外暴露的网络端口。 格式为："ExposedPorts": { "<port>/<tcp|udp>: {}" } 例如： "ExposedPorts":{"22/tcp": {},"80/tcp": {}}
HostConfig： Json 对象，用来设定 container 的运行配置信息。	Binds	JSON 对象，用来设定 container 中挂载的卷组信息。 此参数有三种方式： container_path：直接在 container 中创建一个挂载点。 host_path:container_path：将 host 的一个目录挂载到 container 中。 host_path:container_path:ro：将 host 的一个目录挂载到 container 中，同时在 container 中将此挂载点设置为只读（read only）
	Links	JSON 对象，用来将其他 container 链接到此 container 中。例如，["redis3:redis"]，表示将名为 redis3 的 container，LINK 到此 container，并且在此 container 中取别名为 redis

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
192.168.73.12:2376/containers/create?name=my_container
```

My_container就是本次新创建的容器名称，以此区别其他容器。

Config文件如下：

```
{
       "Hostname": "",
       "Domainname": "",
       "User": "",
       "AttachStdin": false,
       "AttachStdout": true,
       "AttachStderr": true,
       "Tty": false,
       "OpenStdin": false,
       "StdinOnce": false,
       "Env": null,
       "Cmd": [
              "date"
       ],
       "Entrypoint": "",
       "Image": "ubuntu",
       "Labels": {
              "com.example.vendor": "Acme",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "ReadonlyRootfs": false,
        "Dns": ["8.8.8.8"],
        "DnsSearch": [""],
        "ExtraHosts": null,
        "VolumesFrom": ["parent", "other:ro"],
        "CapAdd": ["NET_ADMIN"],
        "CapDrop": ["MKNOD"],
        "RestartPolicy": { "Name": "", "MaximumRetryCount": 0 },
        "NetworkMode": "bridge",
        "Devices": [],
        "Ulimits": [{}],
        "LogConfig": { "Type": "json-file", "Config": {} },
        "SecurityOpt": [""],
        "CgroupParent": ""
    }
}
```

Daemon处理完之后，会返回如下响应：

```
{
    "Id": "9fd53b99b62db7b16575a77db5adfac27f8e7176a3055941ae9286
    "Warnings": null
}
```

这个ID就是刚才创建成功的容器ID，然后用户就可以使用这个ID来

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
{
    "Id": "7a7c25864161119cdaa3b72542f7f43351ffe89b882846faed9fd7
    "Created": "2015-07-21T10:34:31.636978894Z",
    "Path": "/entrypoint.sh",
    "Args": [
        "mysqld"
    ],
    "State": {
        "Running": true,
        "Paused": false,
        "Restarting": false,
        "OOMKilled": false,
        "Dead": false,
        "Pid": 26417,
        "ExitCode": 0,
        "Error": "",
        "StartedAt": "2015-07-21T10:34:31.721290003Z",
        "FinishedAt": "0001-01-01T00:00:00Z"
    },
    "Image": "1eba0810df1fbb18cb2316308cd6eb50ee3d61f97b8a5b45391
    "NetworkSettings": {
        "Bridge": "",
        "EndpointID": "f357e787590ee36a6d437470db574569cc9e5457ee
        "Gateway": "172.17.42.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "HairpinMode": false,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
"/var/lib/Docker/containers/7a7c25864161119cdaa3b72542f7f43351ffe
ts",
    "LogPath":
“/var/lib/Docker/containers/7a7c25864161119cdaa3b72542f7f43351ffe
c25864161119cdaa3b72542f7f43351ffe89b882846faed9fd7aebd95e399-jso
    "Name": "/loving_swartz",
    "RestartCount": 0,
    "Driver": "aufs",
    "ExecDriver": “native-0.2",
    "MountLabel": "",
    "ProcessLabel": “",
    "Volumes": {
        "/var/lib/mysql":
"/var/lib/Docker/volumes/dbe72a658efa0c937b9eb29a3cad16ef1997c059
    },
    "VolumesRW": {
        "/var/lib/mysql": true
    },
    "AppArmorProfile": "",
    "ExecIDs": [
        "18152965506d9fb2168b5d834260c329093b848f666ad412a3896cf8
        "124d3d72802ece4ff9ddfd5360aa281247b30a80deb84073371fed5c
    ],
    "HostConfig": {
        "Binds": null,
        "ContainerIDFile": "",
        "LxcConf": [],
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "PidMode": "",
        "UTSMode": "",
        "CapAdd": null,
        "CapDrop": null,
        "RestartPolicy": {
            "Name": "no",
            "MaximumRetryCount": 0
        },
        "SecurityOpt": null,
        "ReadonlyRootfs": false,
        "Ulimits": null,
        "LogConfig": {
            "Type": "json-file",
            "Config": {}
        },
        "CgroupParent": ""
    },
    "Config": {
        "Hostname": "7a7c25864161",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "PortSpecs": null,
        "ExposedPorts": {
            "3306/tcp": {}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

可以看到Daemon返回的信息量相当大，此容器所有的数据基本都返回了。根据这些数据，用户就可以得知此容器目前状态，并且可以对这些数据进行进一步的整理、操作，从而达到动态监控、预警的目的。

但有的时候，容器ID不如容器名称方便记忆，因此下面我们来演示如何使用容器名称来查询信息。发送如下rest请求：

```
192.168.73.12:2376/containers/loving_swartz/json
```

加黑的地方，就是容器名称。Daemon处理后，会返回同样的信息。

4. 列出指定容器中正在运行的进程信息

API	/containers/(id)/top
API 类型	GET
API 参数	ID: container ID 或者 container name
附加参数	ps_args
API 返回值	200：处理成功 404：指定 container 不存在 500：服务器处理错误

①API说明

本节将要介绍的API可以列出指定容器中的进程信息。因为我们可以将Docker中的每一个容器理解成一个单独的VM，在这个容器中可以

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            "Jul21",
            "?",
            "00:00:36",
            "mysqld"
        ]
    ],
    "Titles": [
        "UID",
        "PID",
        "PPID",
        "C",
        "STIME",
        "TTY",
        "TIME",
        "CMD"
    ]
}
```

转换成比较容易理解的格式后，就是下面的样子：

```
UID    PID    PPID   C   STIME   TTY    TIME      CMD
999   26417  25766   0   Jul21    ?     00:00:36   mysqld
```

使用此条API时需要注意，需要查询的容器必须处于running状态才可以使用。如果查询不是running的容器，那么Daemon将会返回：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "Titles": [
        "USER",
        "PID",
        "%CPU",
        "%MEM",
        "VSZ",
        "RSS",
        "TTY",
        "STAT",
        "START",
        "TIME",
        "COMMAND"
    ]
}
```

转化成Linux格式后就是下面的数据：

```
root@736f34d290ac:/# ps -u
USER  PID  %CPU  %MEM  VSZ    RSS  TTY  STAT  START  TIME  COMMAN
root  26920  0.0    0.1   18172  3400  pts/3  Ss   06:21   0:00
```

此例也可说明ps_args就是ps命令后面的参数。

5. 获取指定容器运行日志

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

③API示例

假设环境中有一台安装了MySQL的container，现在需要获取到MySQL的启动日志，根据上面介绍的内容，发送下面的rest请求应该就可以：

```
192.168.73.12:2376/containers/7a7c25864161/logs
```

但是Daemon却没有返回日志，而是返回了如下信息：

```
Bad parameters: you must choose at least one stream
```

原来，我们没有指定返回的数据流，所以Daemon不知道应该返回stderr还是stdout了。我们指定之后重新发送请求：

```
192.168.73.12:2376/containers/7a7c25864161/logs?stdout=1
```

这次Daemon返回下面的数据：

```
Running mysql_install_db
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
+Note: new default config file not created.
-Please make sure your config file is current

DWARNING: Default config file /etc/mysql/my.cnf exists on the sys
6This file will be read by default by the MySQL server
=If you do not want to use this, either remove it, or use the
A--defaults-file argument to mysqld_safe when starting the server

Finished mysql_install_db
"MySQL init process in progress...
"MySQL init process in progress...
"MySQL init process in progress...
-MySQL init process done. Ready for start up.
```

作为一个运维人员或者开发人员，有时候会更关心有没有error或者warning，因此下面再看一下stderr的日志：

```
192.168.73.12:2376/containers/7a7c25864161/logs?stderr=1
```

这次Daemon返回的数据如下：

```
Y2015-07-21 10:34:31 0 [Note] /usr/sbin/mysqld (mysqld 5.6.25) st
i2015-07-21 10:34:31 29 [Warning] Buffered warning: Changed limit
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
)2015-07-21 10:34:31 29 [Note] Binlog end
C2015-07-21 10:34:31 29 [Note] InnoDB: FTS optimize thread exitir
;2015-07-21 10:34:31 29 [Note] InnoDB: Starting shutdown...
V2015-07-21 10:34:33 29 [Note] InnoDB: Shutdown completed; log se
Y2015-07-21 10:34:33 0 [Note] /usr/sbin/mysqld (mysqld 5.6.25) st
i2015-07-21 10:34:33 52 [Warning] Buffered warning: Changed limit
j2015-07-21 10:34:33 52 [Warning] Buffered warning: Changed limit
S2015-07-21 10:34:33 52 [Note] InnoDB: Using atomics to ref count
I2015-07-21 10:34:33 52 [Note] InnoDB: The InnoDB memory heap is
S2015-07-21 10:34:33 52 [Note] InnoDB: Mutexes and rw_locks use C
A2015-07-21 10:34:33 52 [Note] InnoDB: Memory barrier is not used
G2015-07-21 10:34:33 52 [Note] InnoDB: Compressed tables use zlib
=2015-07-21 10:34:33 52 [Note] InnoDB: Using Linux native AIO
G2015-07-21 10:34:33 52 [Note] InnoDB: Not using CPU crc32 instru
N2015-07-21 10:34:33 52 [Note] InnoDB: Initializing buffer pool,
N2015-07-21 10:34:33 52 [Note] InnoDB: Completed initialization o
R2015-07-21 10:34:33 52 [Note] InnoDB: Highest supported file for
J2015-07-21 10:34:33 52 [Note] InnoDB: 128 rollback segment(s) ar
A2015-07-21 10:34:33 52 [Note] InnoDB: Waiting for purge to start
R2015-07-21 10:34:33 52 [Note] InnoDB: 5.6.25 started; log sequer
)2015-07-21 10:34:33 52 [Note] Binlog end
C2015-07-21 10:34:33 52 [Note] InnoDB: FTS optimize thread exitir
;2015-07-21 10:34:33 52 [Note] InnoDB: Starting shutdown...
V2015-07-21 10:34:35 52 [Note] InnoDB: Shutdown completed; log se
O2015-07-21 10:34:35 0 [Note] mysqld (mysqld 5.6.25) starting as
i2015-07-21 10:34:35 75 [Warning] Buffered warning: Changed limit
j2015-07-21 10:34:35 75 [Warning] Buffered warning: Changed limit
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
=2015-07-21 10:34:36 75 [Note] Shutting down plugin 'ARCHIVE'
H2015-07-21 10:34:36 75 [Note] Shutting down plugin 'PERFORMANCE_
J2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_D
L2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_T
M2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_F
H2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_F
G2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_F
H2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_C
H2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_I
K2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_T
G2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_SYS_T
K2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_FT_IN
K2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_FT_IN
F2015-07-21 10:34:36 75 [Note] Shutting down plugin 'INNODB_FT_CO
```

还好，这里面显示的都是正常的warning。通过这些日志，可以判断出MySQL正常启动了。

下面再看一下，指定timestamps后会返回什么数据呢？下面的rest请求，添加了timestamps参数：

```
192.168.73.12:2376/containers/7a7c25864161/logs?stderr=1&timestar
```

Daemon将返回：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
w2015-07-21T10:34:31.776793191Z 2015-07-21 10:34:31 29 [Note] Inn
file full: wait...
t2015-07-21T10:34:31.778093710Z  2015-07-21  10:34:31  29  [Note]
file ./ib_logfile101 size to 48 MB
r2015-07-21T10:34:31.795549897Z 2015-07-21 10:34:31 29 [Note] Inn
size to 48 MB
x2015-07-21T10:34:31.811412254Z  2015-07-21  10:34:31  29  [Note]
file ./ib_logfile101 to ./ib_logfile0
i2015-07-21T10:34:31.811412311Z 2015-07-21 10:34:31 29 [Warning]
LSN=45781
p2015-07-21T10:34:31.812446065Z 2015-07-21 10:34:31 29 [Note] Inn
creating new
`2015-07-21T10:34:31.818984246Z 2015-07-21 10:34:31 29 [Note] Inn
i2015-07-21T10:34:31.822621298Z 2015-07-21 10:34:31 29 [Note] Inn
active.
w2015-07-21T10:34:31.823208979Z 2015-07-21 10:34:31 29 [Warning]
constraint system tables.
r2015-07-21T10:34:31.825310113Z 2015-07-21 10:34:31 29 [Note] Inn
tables created
u2015-07-21T10:34:31.825508248Z 2015-07-21 10:34:31 29 [Note] Inn
datafile system tables.
t2015-07-21T10:34:31.825577624Z 2015-07-21 10:34:31 29 [Note] Inn
tables created.
`2015-07-21T10:34:31.826142051Z 2015-07-21 10:34:31 29 [Note] Inn
k2015-07-21T10:34:31.876889583Z 2015-07-21 10:34:31 29 [Note] Inn
number 0
H2015-07-21T10:34:31.891897901Z 2015-07-21 10:34:31 29 [Note] Bin
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
is Barracuda.
i2015-07-21T10:34:33.782950678Z 2015-07-21 10:34:33 52 [Note] Inn
active.
`2015-07-21T10:34:33.783320405Z 2015-07-21 10:34:33 52 [Note] Inn
省略剩余日志内容….
```

在本案例中，MySQL输出的日志中自带时间。如果应用日志中没有时间，则可以通过timestamps参数来显示日志时间。

6. 获取指定容器文件系统变更信息

API	/containers/(id)/changes
API 类型	GET
API 参数	ID: container ID 或者 container name
附加参数	无
API 返回值	200：处理成功 404：指定 container 不存在 500：服务器处理错误

①API说明

Docker采用联合文件堆栈系统管理数据，因此容器每次发生文件变更时，都会将变更信息保存在最上面的读写层。如果用户想要知道容器每次都发生了哪些文件变更，就可以使用此API来获取。

②API参数用法：无

③API示例

假设需要获取MySQL　container的文件变更信息，则需要发送如下

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

1：新增

2：删除

此时在MySQL容器里面新创建一个文件，看看Daemon是否可以实时返回：

```
root@7a7c25864161:/# touch /tmp/1.txt
```

再次发送rest请求：

```
192.168.73.12:2376/containers/7a7c25864161/changes
```

Daemon返回了最新文件变更信息：

```
[
    {
        "Path": "/run",
        "Kind": 0
    },
    {
        "Path": "/run/mysqld",
        "Kind": 0
    },
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

②API参数用法：无

③API示例：

这条API使用非常简单，只需要输入容器ID或者容器name即可。假设用户需要导出94146b5b96ac，只需发送：

```
192.168.73.12:2376/containers/94146b5b96ac/export
```

然后将接收到的二进制数据保存成tar包就完成导出了。

8. 获取指定容器资源使用情况

API	/containers/(id)/stats
API 类型	GET
API 参数	ID: container ID 或者 container name
附加参数	stream
API 返回值	200：处理成功 404：指定 container 不存在 500：服务器处理错误

①API说明

此API可以获取指定ID的容器资源使用信息，包括内存、CPU、网络I/O这些信息。当需要监控容器资源时，就可以使用这个API来获取相关信息。

②API参数用法

	布尔值，可接受1/True/true、0/False/false等值。当为真时，

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "rx_bytes": 648,
        "rx_packets": 8,
        "rx_errors": 0,
        "rx_dropped": 0,
        "tx_bytes": 738,
        "tx_packets": 9,
        "tx_errors": 0,
        "tx_dropped": 0
    },
    "precpu_stats": {
        "cpu_usage": {
            "total_usage": 1146263264,
            "percpu_usage": [
                1146263264
            ],
            "usage_in_kernelmode": 800000000,
            "usage_in_usermode": 280000000
        },
        "system_cpu_usage": 283587150000000,
        "throttling_data": {
            "periods": 0,
            "throttled_periods": 0,
            "throttled_time": 0
        }
    },
   "cpu_stats": {
        "cpu_usage": {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            "pgmajfault": 0,
            "pgpgin": 72816,
            "pgpgout": 33520,
            "rss": 109961216,
            "rss_huge": 65011712,
            "swap": 0,
            "total_active_anon": 118972416,
            "total_active_file": 0,
            "total_cache": 115879936,
            "total_inactive_anon": 106651648,
            "total_inactive_file": 217088,
            "total_mapped_file": 208896,
            "total_pgfault": 48746,
            "total_pgmajfault": 0,
            "total_pgpgin": 72816,
            "total_pgpgout": 33520,
            "total_rss": 109961216,
            "total_rss_huge": 65011712,
            "total_swap": 0,
            "total_unevictable": 0,
            "total_writeback": 0,
            "unevictable": 0,
            "writeback": 0
        },
        "failcnt": 0,
        "limit": 2100011008
    },
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
mory_limit":9223372036854771712,"hierarchical_memsw_limit":92233
06651648,"inactive_file":217088,"mapped_file":208896,"pgfault":48
36,"pgpgout":42474,"rss":115228672,"rss_huge":106954752,"swap":0,
total_active_file":0,"total_cache":115879936,"total_inactive_anon
e":217088,"total_mapped_file":208896,"total_pgfault":48746,"total
2836,"total_pgpgout":42474,"total_rss":115228672,"total_rss_huge"
l_unevictable":0,"total_writeback":0,"unevictable":0,"writeback":
08},"blkio_stats":{"io_service_bytes_recursive":[],"io_serviced_r
e":[],"io_service_time_recursive":[],"io_wait_time_recursive":[],
me_recursive":[],"sectors_recursive":[]}}

{"read":"2015-07-23T09:19:05.607185857Z","network":{"rx_bytes":64
"rx_dropped":0,"tx_bytes":738,"tx_packets":9,"tx_errors":0,"tx_dr
usage":{"total_usage":1180489563,"percpu_usage":[1180489563],"usa
age_in_usermode":280000000},"system_cpu_usage":283702910000000,"t
hrottled_periods":0,"throttled_time":0}},"cpu_stats":{"cpu_usage"
cpu_usage":[1180736414],"usage_in_kernelmode":800000000,"usage_in
pu_usage":283703900000000,"throttling_data":{"periods":0,"throttl
0}},"memory_stats":{"usage":231186432,"max_usage":233336832,"stat
tive_file":0,"cache":115879936,"hierarchical_memory_limit":92233
sw_limit":9223372036854771712,"inactive_anon":106651648,"inactive
896,"pgfault":48746,"pgmajfault":0,"pgpgin":72836,"pgpgout":42474
954752,"swap":0,"total_active_anon":124239872,"total_active_file"
al_inactive_anon":106651648,"total_inactive_file":217088,"total_r
lt":48746,"total_pgmajfault":0,"total_pgpgin":72836,"total_pgpgou
"total_rss_huge":106954752,"total_swap":0,"total_unevictable":0,"
":0,"writeback":0},"failcnt":0,"limit":2100011008},"blkio_stats":
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

9. 调整指定容器的**TTY**

API	/containers/(id)/resize
API 类型	POST
API 参数	ID: container ID 或者 container name
附加参数	h w
API 返回值	200：处理成功 404：指定 container 不存在 500：服务器处理错误

①API说明

如果容器启用了TTY，则可以利用这个API来调整TTY的尺寸。在后面的执行实例API中也有一个类似的操作。

②API参数用法

h	高
w	宽

③API示例

这个API很简单，所以不再赘述，下面只展示一下请求rest：

```
192.168.73.12:2376/containers/4fa6e0f0c678/resize?h=40&w=80
```

只要Daemon返回200，就表示设置成功。但需要注意，新的TTY参数需要容器重启后才能生效。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "ReadonlyRootfs": false,
        "Dns": [],
        "DnsSearch": [""],
        "ExtraHosts": null,
        "CapAdd": [],
        "CapDrop": [],
        "RestartPolicy": {  },
        "NetworkMode": "",
        "Devices": [],
        "LogConfig": {  },
        "CgroupParent": ""
    }
```

所有的参数都是复用create container时的参数，参数意义完全一样，具体内容读者可以参考/containers/create一节。

③API示例

下面通过一个示例来说明如何启动指定container，首先组织一下request：

```
{
        "Binds": ["/tmp:/tmp"],
        "Memory": 0,
        "MemorySwap": 0,
        "CpuShares": 512,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
Docker@boot2Docker:~$ Docker ps
CONTAINER ID   IMAGE          COMMAND            CREATED
cb308d170d25  192.168.73.12:5001/mysql  "/entrypoint.sh mysq   5
PORTS                      NAMES
 0.0.0.0:11022->3306/tcp   tender_fermi
```

通过加黑部分可以看出，之前Docker是将容器的3306映射到host的3306端口，在重启容器时，我们将容器的3306端口修改映射到host的11022端口。

因此用户也可以在使用这个API重启容器时，修改一些容器运行参数。

11. 停止指定容器

API	/containers/(id)/stop
API 类型	POST
API 参数	ID: container ID 或者 container name
附加参数	t
API 返回值	204：启动成功 304：此 container 已经处于 stop 状态 404：指定 container 不存在 500：服务器处理错误

①API说明

上面的API是启动容器，这个API则是停止容器。与启动容器需要那么多的数据不同，停止容器只需要指定容器ID或者容器名称就可以。

②API参数用法

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Paused": false,
        "Restarting": false,
        "OOMKilled": false,
        "Dead": false,
        "Pid": 0,
        "ExitCode": 137,
        "Error": "",
        "StartedAt": "2015-07-23T10:09:28.110521923Z",
        "FinishedAt": "2015-07-23T10:22:08.646140026Z"
    },
    "Image": "1eba0810df1fbb18cb2316308cd6eb50ee3d61f97b8a5b45391
    "NetworkSettings": {
        "Bridge": "",
        "EndpointID": "",
        "Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "HairpinMode": false,
        "IPAddress": "",
        "IPPrefixLen": 0,
        "IPv6Gateway": "",
        "LinkLocalIPv6Address": "",
        "LinkLocalIPv6PrefixLen": 0,
        "MacAddress": "",
        "NetworkID": "",
        "PortMapping": null,
        "Ports": null,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "VolumesRW": {
        "/tmp": true,
        "/var/lib/mysql": true
    },
    "AppArmorProfile": "",
    "ExecIDs": null,
    "HostConfig": {
        "Binds": [
            "/tmp:/tmp"
        ],
        "ContainerIDFile": "",
        "LxcConf": null,
        "Memory": 0,
        "MemorySwap": 0,
        "CpuShares": 512,
        "CpuPeriod": 0,
        "CpusetCpus": "",
        "CpusetMems": "",
        "CpuQuota": 0,
        "BlkioWeight": 0,
        "OomKillDisable": false,
        "Privileged": false,
        "PortBindings": {
            "3306/tcp": [
                {
                    "HostIp": "",
                    "HostPort": "11022"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        },
        "SecurityOpt": null,
        "ReadonlyRootfs": false,
        "Ulimits": null,
        "LogConfig": {
            "Type": "json-file",
            "Config": {}
        },
        "CgroupParent": ""
    },
    "Config": {
        "Hostname": "cb308d170d25",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "PortSpecs": null,
        "ExposedPorts": {
            "3306/tcp": {}
        },
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": [
            "MYSQL_ROOT_PASSWORD=root",
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/b
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

掉它。

下面的示例设定最多等待5秒：

```
192.168.73.12:2376/containers/cb308d170d25/stop?t=5
```

12. 重启指定容器

API	/containers/(id)/restart
API 类型	POST
API 参数	ID: container ID 或者 container name
附加参数	t
API 返回值	200：重启成功 404：指定 container 不存在 500：服务器处理错误

①API说明

上面两个API是单独启动和停止容器。如果需要重启容器，则可以使用restart api。

②API参数用法

t	整型值，用来设定超时时间

③API示例

假设现在需要重启MySQL容器，发送下面的rest请求：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

执行stop时，Daemon会向容器发送SIGTERM信号，然后再发送SIGKILL信号。正常情况下，容器内部运行的进程在接收到SIGTREM信号后，都会保存数据并且正常退出，而后容器再正常关闭。

而执行kill时，Daemon只向容器发送SIGKILL信号，强制关闭容器。因此容器内部有些数据可能因为没有及时保存，而发生丢失。

所以stop是优雅地关闭容器，而kill则是暴力关闭容器。

②API参数用法

Signal	信号变量

③API示例

现在假设mysql出现问题，需要强制关闭，就可以发送如下rest请求

```
192.168.73.12:2376/containers/cb308d170d25/kill
```

当Daemon返回204响应后，就表示Daemon已经强制关闭了容器。

在默认情况下，发送kill请求时，Daemon将会发送SIGKILL信号给容器。但用户也可以通过signal参数发送其他信号。比如用户需要Daemon在关闭容器之前，给容器发送SIGINT信号，则可以发送如下rest请求：

```
192.168.73.12:2376/containers/cb308d170d25/kill?signal=SIGINT
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    {
        "Id": "cb308d170d255e3f6d7a6b400c9b5b4b0f196f2faf5f64e46c
        "Names": [
            "/tender_fermi"
        ],
        "Image": "192.168.73.12:5001/mysql",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437642849,
        "Ports": [
            {
                "IP": "0.0.0.0",
                "PrivatePort": 3306,
                "PublicPort": 11022,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Up 45 minutes"
    }
]
```

加黑部分就是MySQL容器的名称。现在假设用户需要对它进行重命名，将默认不好牢记的tender_fermi修改为mysql_3306，则发送如下rest请求：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
]
```

加黑部分可以看出名称已经改为mysql_3306了。

15. 暂停指定容器

API	/containers/(id)/pause
API 类型	POST
API 参数	ID: container ID 或者 container name
附加参数	无
API 返回值	204：成功 404：不存在 500：服务器处理错误

①API说明

暂停和停止（stop）、关闭（killl）不同。执行stop或者kill时，容器都会退出。从主机操作系统的角度看，执行此容器的进程已经退出了。而执行pause时，在主机操作系统层面执行此容器的进程仍然存在，没有退出，而此容器内部所有的进程处于冻结状态。

②API参数用法：无。

③API实例

首先查询目前哪些容器处于running状态：

```
[
    {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

验证一下：


```
[
    {
        "Id": "cb308d170d255e3f6d7a6b400c9b5b4b0f196f2faf5f64e46c
        "Names": [
            "/mysql_3306"
        ],
        "Image": "192.168.73.12:5001/mysql",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437642849,
        "Ports": [
            {
                "IP": "0.0.0.0",
                "PrivatePort": 3306,
                "PublicPort": 11022,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Up 30 hours (Paused)"
    }
]
```


从加黑部分可以看出，MySQL已经处于冻结状态。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            {
                "IP": "0.0.0.0",
                "PrivatePort": 3306,
                "PublicPort": 11022,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Up 30 hours (Paused)"
    }
]
```

假设需要让MySQL容器恢复工作，则只需要发送如下rest请求：

```
192.168.73.12:2376/containers/mysql_3306/unpause
```

如果Daemon返回204，则表示Daemon已经完成了恢复工作。下面查询验证一下：

```
[
    {
        "Id": "cb308d170d255e3f6d7a6b400c9b5b4b0f196f2faf5f64e46c
        "Names": [
            "/mysql_3306"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

API	/containers/(id)/attach
API 类型	POST
API 参数	ID: container ID 或者 container name
附加参数	logs stream stdin stdout stderr
API 返回值	200：挂载成功 400：参数错误 404：指定 container 不存在 500：服务器处理错误

①API说明

使用这个API，用户就可以挂载到指定容器中。用户挂载到容器中有什么用呢？假设目前生产环境中，MySQL容器出现异常情况，但又不允许关闭MySQL，那么用户就可以通过这个API挂载到MySQL容器中去，通过执行其他命令来排除故障。

②API参数用法

logs	布尔值，接受 1/True/true 或者 0/False/false，默认为 false。 如果为 true 的话，Daemon 将会实时返回此 container 的日志信息
stream	布尔值，接受 1/True/true 或者 0/False/false，默认为 false。 如果为 true 的话，Daemon 将返回此 container 的 I/O 流
stdin	布尔值，接受 1/True/true 或者 0/False/false，默认为 false。 如果为 stream 为 true 的话，Daemon 将允许客户端输入数据到此 container 的 stdin 中。 注意：当需要挂载 stdin 时，如果 stream=true，同时 logs=true，则 Daemon 将会关闭 stdin，所以不需要同时指定 logs 参数
stdout	布尔值，接受 1/True/true 或者 0/False/false，默认为 false。 如果为 logs=true 且 stdout=true，则 Daemon 将返回此 container 的 stdout 到客户端。如果 stream=true 且 stdout=true，则 Daemon 将允许客户端挂载到 container 的 stdout 中
stderr	布尔值，接受 1/True/true 或者 0/False/false，默认为 false。 如果为 logs=true 且 stderr =true，则 Daemon 将返回此 container 的 stderr 到客户端。如果 stream=true 且 stderr =true，则 Daemon 将允许客户端挂载到 container 的 stderr 中

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            {
                "PrivatePort": 22,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Up 3 minutes"
    },
    {
        "Id": "cb308d170d255e3f6d7a6b400c9b5b4b0f196f2faf5f64e46c
        "Names": [
            "/mysql_3306"
        ],
        "Image": "192.168.73.12:5001/mysql",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437642849,
        "Ports": [
            {
                "IP": "0.0.0.0",
                "PrivatePort": 3306,
                "PublicPort": 11022,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Up 5 minutes"
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
   20 ?        00:00:00 ps
]0;root@7c826ceb39d2: /root@7c826ceb39d2:/#
```

可以看到rest客户端接收到了数据。如果挂载了Stdin，那么在rest客户端输入数据之后，Daemon就会把接收的数据传送到Ubuntu容器的stdin。这里不再演示。

18. 通过**WebSocket**挂载到容器

API	/containers/(id)/attach/ws
API 类型	GET
API 参数	ID: container ID 或者 container name
附加参数	logs stream stdin stdout stderr
API 返回值	200：挂载成功 400：参数错误 404：指定 container 不存在 500：服务器处理错误

①API说明

这个API可以通过WebSocket协议（WebSocket是HTML5中新定义的协议，用来实现浏览器和服务器之间的全双工通信）来完成交互。只要客户端按照RFC6455的WebSocket协议发送数据，Daemon与客户端之间就可以完成全双工通信。

②API参数用法

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
INFO[175197] GET /containers/modest_wright/attach/ws?stream=1&s
DEBU[175197] attach: stdin: begin
DEBU[175197] attach: stdout: begin
DEBU[175197] attach: stderr: begin
```

可以看到默认情况下，Daemon端会同时初始化stdin、stdout和stderr，然后根据客户端的请求发回不同的I/O流。

如果在客户端输入简单的Linux命令，使用date来获取container的时间，如图5-2所示。

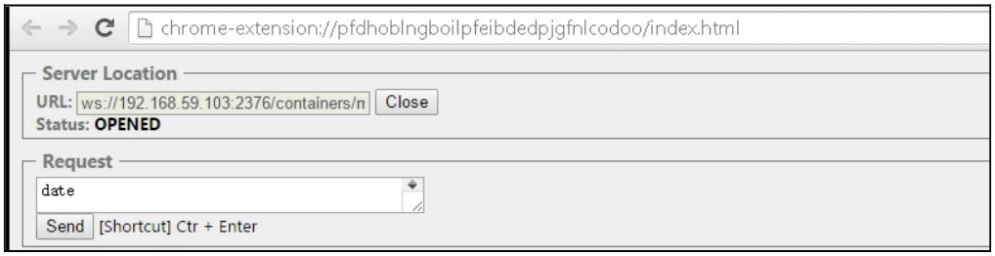


图5-2　使用date来获取container的时间

单击Send后，在Ubuntu容器中，就可以收到这条命令并且执行：

```
root@7c826ceb39d2:/# date
Sat Jul 25 08:18:53 UTC 2015
root@7c826ceb39d2:/#
```

同时因为是WebSocket协议，所以客户端也可以实时查看到结果，

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

假设用户需要获取Ubuntu关闭信息，并且判断Ubuntu容器是否是正常退出，就可以发送如下rest请求：

```
192.168.73.12:2376/containers/modest_wright/wait
```

mysql容器关闭之后，Daemon将返回如下信息：

```
{
    "StatusCode": 0
}
```

通过解析StatusCode，就可以做后续处理了。

20. 移除指定容器

API	/containers/(id)
API 类型	DELETE
API 参数	container ID 或者 container name
附加参数	v force
API 返回值	204：删除成功 400：参数输入错误 404：指定 container 不存在 500：服务器处理错误

①API说明

当用户不再使用这个容器时，可以使用这个API来移除容器记录，

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Names": [
            "/modest_wright"
        ],
        "Image": "192.168.73.12:5001/ubuntu-dev",
        "Command": "/bin/bash",
        "Created": 1437809744,
        "Ports": [],
        "Labels": {},
        "Status": "Exited (0) 9 minutes ago"
    },
    {
        "Id": "cb308d170d255e3f6d7a6b400c9b5b4b0f196f2faf5f64e46c
        "Names": [
            "/mysql_3306"
        ],
        "Image": "192.168.73.12:5001/mysql",
        "Command": "/entrypoint.sh mysqld",
        "Created": 1437642849,
        "Ports": [
            {
                "IP": "0.0.0.0",
                "PrivatePort": 3306,
                "PublicPort": 11022,
                "Type": "tcp"
            }
        ],
        "Labels": {},
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
                "PublicPort": 11022,
                "Type": "tcp"
            }
        ],
        "Labels": {},
        "Status": "Up About an hour"
}
]
```

返回信息中已经没有Ubuntu容器信息了，表示API执行成功。

21. 从指定容器中拷贝文件/目录

API	/containers/(id)/copy
API 类型	POST
API 参数	container ID 或者 container name
附加参数	Application/json 对象
API 返回值	200：拷贝成功 404：指定 container 不存在 500：服务器处理错误

①API说明：

这个API是最后一个操作容器的API，可用来从容器拷贝指定的文件或者目录。因为有时候用户需要从容器取出文件进行分析。

从前面的介绍可知，当容器启动时，可以通过共享的方式让容器获取到主机的文件或者目录。通过这个API，用户就可以从容器中实时取出文件。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

文件名为全路径，然后Daemon将会以application/x-tar的形式将文件返回给客户端。

22. 列出所有镜像信息

API	/images/json
API 类型	GET
API 参数	无
附加参数	all filters
API 返回值	200: 处理成功

①API说明：

操作镜像的API和操作容器的API基本类似，无非也是增删改查。在容器操作API中，有一个列出目前所有容器信息，这里也有一个与之相对应的API，就是查询目前所有的镜像信息。

②API参数用法

all	布尔值，接受1/True/true或者0/False/false，默认为false
filters	一个map[string][]string结构的JSON字符串，用来过滤image信息。目前有效的筛选参数为：dangling=true或者针对image label进行label=key/key=value筛选

③API示例

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "RepoTags": [
            "192.168.73.12:5001/mysql:latest"
        ],
        "RepoDigests": [],
        "Created": 1435878224,
        "Size": 0,
        "VirtualSize": 283489838,
        "Labels": {}
    },
    {
        "Id": "7a77eedb9e62030831507630a89515fae3c5fab5deef8835a6
        "ParentId": "57ccbdc390886ffafe292cb8aa9a8dc87adf8722566
        "RepoTags": [
            "<none>:<none>"
        ],
        "RepoDigests": [
            "<none>@<none>"
        ],
        "Created": 1435878224,
        "Size": 0,
        "VirtualSize": 283489838,
        "Labels": {}
    },
    {
        "Id": "57ccbdc390886ffafe292cb8aa9a8dc87adf872256645595
        "ParentId": "d7f9d4cf73af1624517053e9b621002f2e8aa7c6702
        "RepoTags": [
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "RepoTags": [
            "<none>:<none>"
        ],
        "RepoDigests": [
            "<none>@<none>"
        ],
        "Created": 1434415822,
        "Size": 1822,
        "VirtualSize": 283486901,
        "Labels": {}
    },
    {
        "Id": "e6897bf953062e9b0de5e4719949d420ac4ade194ec85b9d4b
        "ParentId": "8709a819fa6b56d6c916a78f46b3395c7afb9895a1e0
        "RepoTags": [
            "<none>:<none>"
        ],
        "RepoDigests": [
            "<none>@<none>"
        ],
        "Created": 1434415821,
        "Size": 166706780,
        "VirtualSize": 283485079,
        "Labels": {}
    },
    {
        "Id": "8709a819fa6b56d6c916a78f46b3395c7afb9895a1e061892
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Id": "857e8088dd2047ce7df41f40e8b616968e46871b48d7242f53
        "ParentId": "88b078de93806ba6f3ad3f31a2a6b7355fb9250ec8f9
        "RepoTags": [
            "<none>:<none>"
        ],
        "RepoDigests": [
            "<none>@<none>"
        ],
        "Created": 1434415803,
        "Size": 94898,
        "VirtualSize": 116778244,
        "Labels": {}
    },
    {
        "Id": "02a18be0147e7c5a93022fd19fbe03684fcf140180494c2218
        "ParentId": "857e8088dd2047ce7df41f40e8b616968e46871b48d7
        "RepoTags": [
            "<none>:<none>"
        ],
        "RepoDigests": [
            "<none>@<none>"
        ],
        "Created": 1434415803,
        "Size": 0,
        "VirtualSize": 116778244,
        "Labels": {}
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Id": "91c95931e552b11604fea91c2f537284149ec32fff0f700a47
        "ParentId": "a8219747be10611d65b7c693f48e7222c0bf54b5df84
        "RepoTags": [
            "hello-world:latest"
        ],
        "RepoDigests": [],
        "Created": 1429308098,
        "Size": 0,
        "VirtualSize": 910,
        "Labels": {}
    },
    {
        "Id": "e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3
        "ParentId": "365f2af56f23411f84b8c9573b165b3fa666baebe561
        "RepoTags": [
            "192.168.73.12:5001/ubuntu-dev:latest"
        ],
        "RepoDigests": [],
        "Created": 1427177078,
        "Size": 0,
        "VirtualSize": 760523598,
        "Labels": null
    }
]
```


由此可以看出，如果不需要all，或者将all置为false，则Daemon仅

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

然后查询hello-world镜像全量信息如下：


```
{
        "Id": "91c95931e552b11604fea91c2f537284149ec32fff0f700a47
        "ParentId": "a8219747be10611d65b7c693f48e7222c0bf54b5df84
        "RepoTags": [
            "hello-world:latest"
        ],
        "RepoDigests": [],
        "Created": 1429308098,
        "Size": 0,
        "VirtualSize": 910,
        "Labels": {}
    },
    {
        "Id": "a8219747be10611d65b7c693f48e7222c0bf54b5df8467d3f9
        "ParentId": "",
        "RepoTags": [
            "<none>:<none>"
        ],
        "RepoDigests": [
            "<none>@<none>"
        ],
        "Created": 1429308098,
        "Size": 910,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

Docker Daemon在运行hello-world镜像时，首先会加载a8219747be10611d65b7c693f48e7222-c0bf54b5df8467d3f99003611afa1fd8层，然后再加载91c95931e552b11604fea91c2f537284149ec32fff0-f700a4769cfd31d7696ae层。

我们进入hello-world后，使用df–h看到的所有文件目录也是这两层的所有文件重叠而显示出来的。

读者可以再用mysql和Ubuntu的镜像信息来体会一下联合分层文件系统的含义。因篇幅所限，这里不再单独分析mysql和Ubuntu的镜像文件信息。

除了all参数，这个API还有filters参数。filters参数是一个JSON格式的字符串，目前仅支持dangling=true筛选和针对镜像label进行的label=key/key=value筛选。

下面演示一下如何使用filters参数。

我们先发送一个dangling=true的rest请求。Dangling是悬空的意思，如果读者有单片机开发经验，就很容易理解悬空的意思了。在单片机中，悬空的概念是物理电路中，此器件不接任何电路也不接地。在这里悬空的意思，是没有tag并且名称为none，同时，也没有父文件层的临时文件层（估计开发这个API的工程师有过单片机开发经验，所以使用了悬空这个词汇）。

再直接点，就是用上dangling=true后，Daemon将返回所有标记为none的临时image。

例如，我们发送：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

这里面的filters={"dangling":["true"]}就是JSON格式的字符串。前面我们提到过，filters是一个map[string][]string格式的字符串，意思就是：filters是一个map，key是string类型的字符串，value是string数组。所以这里“dangling”就是key，而["true"]则是string数组。

同样，我们测试一下label=key/key=value。通过刚才查询出来的镜像，可以看到label一项都是空值。我们先来构建（build）一个包含label值的image。

构建使用的Dockerfile如下：

```
FROM 192.168.73.12:5001/mysql
LABEL version=1.0
LABEL build=0727
```

构建完成后，再查询一下：

```
[
    {
        "Id": "3ccda04e306b77fbe4f56174956c84608e0ec9bfdbe7a59c67
        "ParentId": "400b39e39c800d9ab60d1326979a82ecaeeb5e113878
        "RepoTags": [
            "mysql:label"
        ],
        "RepoDigests": [],
        "Created": 1437973221,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Size": 0,
        "VirtualSize": 910,
        "Labels": {}
    },
    {
        "Id": "e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3
        "ParentId": "365f2af56f23411f84b8c9573b165b3fa666baebe56f
        "RepoTags": [
            "192.168.73.12:5001/ubuntu-dev:latest"
        ],
        "RepoDigests": [],
        "Created": 1427177078,
        "Size": 0,
        "VirtualSize": 760523598,
        "Labels": null
    }
]
```

现在我们就拥有一个具有label属性的镜像了。

那么如何通过label=key/key=value来查询呢？很简单，我们发送如下rest请求就可以查询拥有version label属性的image了：

```
192.168.73.12:3600/images/json?filters={"label":["version"]}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

Daemon将会返回在目前镜像的label属性中，拥有version=1.0这个属性值的镜像信息，如下所示：

```
[
    {
        "Id": "3ccda04e306b77fbe4f56174956c84608e0ec9bfdbe7a59c67
        "ParentId": "400b39e39c800d9ab60d1326979a82ecaeeb5e113878
        "RepoTags": [
            "mysql:label"
        ],
        "RepoDigests": [],
        "Created": 1437973221,
        "Size": 0,
        "VirtualSize": 283489838,
        "Labels": {
            "build": "0727",
            "version": "1.0"
        }
    }
]
```

通过以上两个案例可以发现，通过label=key能查询所有镜像label属性中拥有key值的数据，而通过key=value，则可以进行精确筛选。

23. 根据**Dockerfile**创建**Docker**镜像

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

Dockerfile	字符串，用来指定 Daemon 构建时所使用的 Dockerfile。若为空，则默认为 Dockerfile
t	字符串，用来指定生成的 image tag。此属性必须满足正则表达式[a-z0-9-_.]
remote	字符串，用来指定远程 URL。指定此项后，Daemon 将会下载 URL 资源，然后执行后续构建操作
q	布尔值，可接受 1/True/true/yes 或者 0/False/false/no/none，表示是否输出全量构建（build）日志，默认为 false
nocache	布尔值，可接受 1/True/true/yes 或者 0/False/false/no/none，表示是否使用 image cache，默认为 false
pull	布尔值，可接受 1/True/true/yes 或者 0/False/false/no/none，表示是否需要更新 baseimage，默认为 false。若为 true，则每次构建（build）时，都会首先更新 baseimage。仅在 Docker 1.16 或更高版本中生效
rm	布尔值，可接受 1/True/true/yes 或者 0/False/false/no/none，表示是否需要删除构建（build）过程中所产生的中间临时 image。在 Docker v1.12 或更高版本中，此项默认为 true。其他版本中默认为 false
forcerm	布尔值，可接受 1/True/true/yes 或者 0/False/false/no/none，表示是否需要强制删除构建（build）过程中产生的临时 image 数据。此项比 rm 优先级高，只有在 forcerm 不存在或者 forcerm=false 的情况下，rm 属性才生效。在 Docker1.12 或者更高版本中，此项默认为 false
memory	整型数值，用来设置可用于构建（build）的内存大小
memswap	整型数值，用来设置可用于构建（build）的内存总大小（memory+swap），若为-1 表示禁用 swap
cpushares	整型数值，用来设置可用于构建（build）的 CPU 使用权重
cpusetcpus	整型数值，用来设置执行构建（build）的 CPU。例如，指定 0 号和 1 号 CPU 来执行构建

③API示例

首先确认一下当前环境中存在的镜像数量及信息，如下：

```
[
  {
    "Id": "3ccda04e306b77fbe4f56174956c84608e0ec9bfdbe7a59c6763b4
    "ParentId": "400b39e39c800d9ab60d1326979a82ecaeeb5e113878f1ec
    "RepoTags": [
      "mysql:label"
    ],
    "RepoDigests": [],
    "Created": 1437973221,
    "Size": 0,
    "VirtualSize": 283489838,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "Labels": {}
  },
  {
    "Id": "e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3c178
    "ParentId": "365f2af56f23411f84b8c9573b165b3fa666baebe56f165a
    "RepoTags": [
      "192.168.73.12:5001/ubuntu-dev:latest"
    ],
    "RepoDigests": [],
    "Created": 1427177078,
    "Size": 0,
    "VirtualSize": 760523598,
    "Labels": null
  }
]
```

当前环境仅存在上述几个image，我们现在开始利用RemoteAPI来创建一个镜像。

首先我们需要一个Dockerfile（有关于Dockerfile的知识点，会在后面逐一介绍，这里只需记住Dockerfile是用来组织image的一个配置文件即可）。

Dockerfile内容如下：

```
FROM 192.168.73.12:5001/mysql
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

作时只能发送一次请求，所以需要把所有涉及的文件做成tar包上传给Daemon。

Docker Daemon接受到tar包后，会进行解压缩操作，然后按照本地方式进行后续构建（build）操作。

针对tar包，Docker支持四种类型：

- 无压缩。
- 使用gzip压缩算法。
- 使用bzip2算法。
- 使用xz算法。

这里，我们选择bzip2算法。我们将做好的tar包命名为Docker.tar，然后发送rest请求。注意：一定设置Content-Type=application/tar，然后以二进制的形式将Docker.tar发送给server。

```
192.168.73.12:3600/build
```

如果Daemon返回200响应，则表示image create成功，同时也会将build log一并返回：

```
{"stream":"Step 0 : FROM 192.168.73.12:5001/mysql\n"}
{"stream":" ---\u003e 1eba0810df1f\n"}
{"stream":"Step 1 : LABEL version 1.0\n"}
{"stream":" ---\u003e Using cache\n"}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "build": "0727",
    "comment": "Remote",
    "version": "1.0"
  }
},
{
  "Id": "3ccda04e306b77fbe4f56174956c84608e0ec9bfdbe7a59c6763b4
  "ParentId": "400b39e39c800d9ab60d1326979a82ecaeeb5e113878f1ec
  "RepoTags": [
    "mysql:label"
  ],
  "RepoDigests": [],
  "Created": 1437973221,
  "Size": 0,
  "VirtualSize": 283489838,
  "Labels": {
    "build": "0727",
    "version": "1.0"
  }
},
{
  "Id": "1eba0810df1fbb18cb2316308cd6eb50ee3d61f97b8a5b45391b24
  "ParentId": "55361d8836d0576e189cb8b865643be2d56ec370c129510
  "RepoTags": [
    "192.168.73.12:5001/mysql:latest"
  ],
  "RepoDigests": [],
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "Labels": null
  }
]
```

这里哪个才是我们刚才新创建的镜像呢？通过和最开始的镜像信息进行比对，可以发现下面是多出来的镜像：

```
{
    "Id": "3932b3b157569be75fde18dcf4c93cc880ec9b915367b6923ecd5c
    "ParentId": "3ccda04e306b77fbe4f56174956c84608e0ec9bfdbe7a59c
    "RepoTags": [
      "<none>:<none>"
    ],
    "RepoDigests": [
      "<none>@<none>"
    ],
    "Created": 1438067586,
    "Size": 0,
    "VirtualSize": 283489838,
    "Labels": {
      "build": "0727",
      "comment": "Remote",
      "version": "1.0"
    }
  }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

再查询一次，看看现在有了哪些镜像：

```
[
  {
    "Id": "7e137307b71178527d136f2ddac836fe4c2fb4226a5778ef29d87(
    "ParentId": "3ccda04e306b77fbe4f56174956c84608e0ec9bfdbe7a59(
    "RepoTags": [
      "myremote-image:latest"
    ],
    "RepoDigests": [],
    "Created": 1438069214,
    "Size": 0,
    "VirtualSize": 283489838,
    "Labels": {
      "build": "0727",
      "comment": "Remote",
      "version": "1.0"
    }
  },
  {
    "Id": "3ccda04e306b77fbe4f56174956c84608e0ec9bfdbe7a59c6763b4
    "ParentId": "400b39e39c800d9ab60d1326979a82ecaeeb5e113878f1e(
    "RepoTags": [
      "mysql:label"
    ],
    "RepoDigests": [],
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "Created": 1429308098,
    "Size": 0,
    "VirtualSize": 910,
    "Labels": {}
  },
  {
    "Id": "e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3c178
    "ParentId": "365f2af56f23411f84b8c9573b165b3fa666baebe56f165a
    "RepoTags": [
      "192.168.73.12:5001/ubuntu-dev:latest"
    ],
    "RepoDigests": [],
    "Created": 1427177078,
    "Size": 0,
    "VirtualSize": 760523598,
    "Labels": null
  }
]
```

加黑部分就是我们新创建的镜像，而且在tag属性中，也有了刚才我们指定的myremote-image。以后就可以使用这个tag来操作镜像了。

通过这个示例可以看出，t参数可以指定镜像名称和标签。

Dockerfile参数可以用来指定需要加载的Dockerfile。前面我们说过，Docker默认会使用tar包里面的Dockerfile（这里需要指出，Dockerfile一般放在tar的顶层目录中）。如果我们有多个Dockerfile，并

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
FROM 192.168.73.12:5001/mysql
LABEL version=1.0
LABEL build=0727
LABEL comment=GIT
```

为了与区别之前创建的镜像，这里将comment修改为GIT，表示是由Github上面的Dockerfile所创建的。

我们将这个镜像命名为mygit-image，然后发送如下rest请求：（注意，Content-Type需要设置为application/tar）

```
192.168.73.12:3600/build?t=mygit-image&remote=https://github.com/
```

这里我们看一下Daemon返回的数据：

```
{"stream":"Step 0 : FROM 192.168.73.12:5001/mysql\n"}
{"stream":" ---\u003e 1eba0810df1f\n"}
{"stream":"Step 1 : LABEL version 1.0\n"}
{"stream":" ---\u003e Using cache\n"}
{"stream":" ---\u003e 400b39e39c80\n"}
{"stream":"Step 2 : LABEL build 0727\n"}
{"stream":" ---\u003e Using cache\n"}
{"stream":" ---\u003e 3ccda04e306b\n"}
{"stream":"Step 3 : LABEL comment GIT\n"}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "Labels": {
      "build": "0727",
      "comment": "GIT",
      "version": "1.0"
    }
  }
]
```

通过tag＝mygit-image和comment＝GIT，表明这的确是我们刚才通过Github上面的Dockerfile所创建的image。

除了使用Github之外，也可以通过普通的URL来创建，只不过Daemon会将URL所指向的资源下载到local，然后再加载使用。

当我们需要查看完整日志时，可以将q参数置为true，这样Daemon将返回构建过程中的全量日志。

--nocache参数表示是否使用缓存数据。通过上面简单的Dockerfile，我们可以看到Dockerfile里面就是一条条的命令。在构建镜像时，Daemon会将所有执行过的命令缓存起来。再次执行构件时，Daemon优先会查询缓存中是否已经有此命令。如果有的话，Daemon会直接将上次结果拿来使用。如果缓存中没有，Daemon才会执行这条命令。如果我们将nocache置为true，则Daemon将强制重新执行每一条命令。

--pull参数表示是否更新baseimage。在Dockerfile中，第一行FROM XXXX，表示以XXX为baseimage进行构建。如果不使用pull，Daemon

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

API	/images/create
API 类型	POST
API 参数	无
附加参数	fromImage fromSrc repo tag registry
API 返回值	200：创建成功 500：服务器处理错误

①API说明

从名称上来看，这个API也是来创建镜像的，那么和刚刚我们介绍的/build有什么区别呢？

Docker获取镜像有三种方式。

一、是构建镜像。就是自力更生，由Daemon重新创建一个镜像。

二、从外部获取image。这个外部可以是私有仓库或者公有仓库，也可以是通过导入/导出方式获取的tar包。

三、以容器为基础进行创建，这种方式会在后面的章节中介绍到。

上面我们介绍的/build就是通过自力更生的方式，使用Dockerfile来创建镜像。这种方式，完全自主可控，镜像里面的内容可以根据我们的需要进行自由裁剪。而通过外部获取的方式，只能是由第三方做好镜像之后，我们拿来使用。镜像里面的内容我们没办法修改。

我们现在将要介绍的/images/create，属于外部获取的方式。既然外部获取有两种方式，那么这两种方式又有什么区别呢？

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

fromImage优先级比fromSrc高。当fromImage有值时，即便这个属性值非法或者失效，fromSrc也无法使用。

repo仅在使用fromSrc时生效。这是因为如果采取pull的方式，那么镜像名称此时已经被确定了（如果镜像名称不确定的话，Daemon将无法知道要pull哪个镜像）。而fromSrc就不同了，用户需要告知Daemon这个镜像导入成功后，它的名字是什么。所以repo可以为空，此时Daemon将会默认赋值为none。

tag作用同repo类似。当选择fromImage时，Daemon将会pull指定tag的镜像。例如，当tag为latest时，Daemon只会pull　XXXXX:latest。如果tag为空，Daemon将会pull此镜像所有的版本。而当选择fromSrc时，Daemon导入成功后，将tag赋值给镜像。

Registry，预留项。

各个参数介绍完之后，我们首先演示如何使用fromImage。假定需要pull Ubuntu image，在目前Docker hub registry中，共存在ubuntu 20多个tag，我们这次只下载15.10。

首先在fromImage中赋值为ubuntu，在tag中赋值为15.10，然后发送如下rest请求：

```
192.168.73.12:2376/images/create?fromImage=ubuntu&tag=15.10
```

在等待若干时间后，Daemon将返回如下信息：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    ],
    "RepoDigests": [],
    "Created": 1438124332,
    "Size": 0,
    "VirtualSize": 749744460,
    "Labels": {
      "com.tibco.atmosphere.ci.buildNumber": "106",
      "com.tibco.atmosphere.ci.buildTypeId": "Atmosphere_IPaas_Ap
      "com.tibco.atmosphere.ci.url":
"http://llbuild2.na.tibco.com:8080/viewLog.html?buildNumber=106&b
Modeler"
    }
  },
  {
    "Id": "22ba5544a647176471ebf158dd79c131b5723d2459d39a989072d8
    "ParentId": "5026054de7d2cee6290c83c98ba2bfec2366cb575ccb6c31
    "RepoTags": [
      "ubuntu:15.10"
    ],
    "RepoDigests": [],
    "Created": 1436470202,
    "Size": 0,
    "VirtualSize": 134743114,
    "Labels": {}
  }
]
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "Id": "aac1ad7ac32feafbabdb79a3299f06420a053c5e1c81551322d281
    "ParentId": "",
    "RepoTags": [
      "ubuntu-debootstrap-local:14.04"
    ],
    "RepoDigests": [],
    "Created": 1438239965,
    "Size": 92753079,
    "VirtualSize": 92753079,
    "Labels": null
  },
  {
    "Id": "dad544a114c3c7374ec0bc2bad31143f62f3a7bab2ae34ab42c46a
    "ParentId": "6a010de58bb3fd6e1580edf017a67ea0aa0e00bc05bd1711
    "RepoTags": [
      "192.168.73.12:5002/ipaas/api-modeler:latest"
    ],
    "RepoDigests": [],
    "Created": 1438124332,
    "Size": 0,
    "VirtualSize": 749744460,
    "Labels": {
      "com.tibco.atmosphere.ci.buildNumber": "106",
      "com.tibco.atmosphere.ci.buildTypeId": "Atmosphere_IPaas_Ap
      "com.tibco.atmosphere.ci.url":
"http://llbuild2.na.tibco.com:8080/viewLog.html?buildNumber=106&b
Modeler"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

API	/images/(name)/json
API 类型	GET
API 参数	Name:字符串。image id 或者 image name
附加参数	无
API 返回值	200：处理成功 404：无此 image 500：服务器处理错误

①API说明

在介绍容器操作命令时，我们提到过获取容器详细信息的API，这里我们再介绍一个获取镜像详细信息的API。与容器相比，镜像信息要少很多。

因为镜像是作为模板来使用的，没有runtime数据，所以它的信息量就很小了。

②API参数用法：无

③API示例

我们用刚刚创建的Ubuntu来演示如何使用这个API。首先我们需要获取到镜像id，就通过/images/json来获取。

```
[
  {
    "Id": "aac1ad7ac32feafbabdb79a3299f06420a053c5e1c81551322d281
    "ParentId": "",
    "RepoTags": [
      "ubuntu-debootstrap-local:14.04"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "ParentId": "5026054de7d2cee6290c83c98ba2bfec2366cb575ccb6c31
    "RepoTags": [
      "ubuntu:15.10"
    ],
    "RepoDigests": [],
    "Created": 1436470202,
    "Size": 0,
    "VirtualSize": 134743114,
    "Labels": {}
  }
]
```

加黑部分是镜像id。当然我们不需要输入这么长的id，输入short id即可。我们发送如下rest请求：

```
192.168.73.12:3600/images/22ba5544a64/json
```

Daemon返回如下信息：

```
{
  "Id": "22ba5544a647176471ebf158dd79c131b5723d2459d39a989072d802
  "Parent": "5026054de7d2cee6290c83c98ba2bfec2366cb575ccb6c3196b0
  "Comment": “",
  "Created": "2015-07-09T19:30:02.407197018Z",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "Labels": {}
  },
  "DockerVersion": "1.6.2",
  "Author": "",
  "Config": {
    "Hostname": "6edadd5399cf",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "PortSpecs": null,
    "ExposedPorts": null,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": null,
    "Cmd": [
      "/bin/bash"
    ],
    "Image": "5026054de7d2cee6290c83c98ba2bfec2366cb575ccb6c3196b
    "Volumes": null,
    "Volumes": null,
    "VolumeDriver": "",
    "WorkingDir": "",
    "Entrypoint": null,
    "NetworkDisabled": false,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

每个镜像都是由若干文件层叠加形成的，所以当镜像发生变化时，其实是最上面的文件层发生了变化。

通过这个API，我们就可以查询到镜像的文件变更记录

②API参数用法：无

③API示例

我们继续使用Ubuntu这个镜像来演示。通过上面几个实例，我们已经得知Ubuntu的image id是22ba5544a647176471ebf158dd79c131b5723d2459d39a989072d8020599a3d1 image name是ubuntu:15.10。

下面我们分别使用这两个来发送rest请求，首先使用image id来发送rest请求：

```
192.168.73.12:3600/images/22ba5544a6471764/history
```

Daemon返回如下信息：

```
[
  {
    "Id": "22ba5544a647176471ebf158dd79c131b5723d2459d39a989072d8
    "Created": 1436470202,
    "CreatedBy": "/bin/sh -c #(nop) CMD [\"/bin/bash\"]",
    "Tags": [
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
\"gz\";' > /etc/apt/apt.conf.d/Docker-gzip-indexes",
    "Tags": null,
    "Size": 701,
    "Comment": ""
  },
  {
    "Id": "c0c20a188a7be02c7b84606d0e7204efb366c416533eaebefb331a
    "Created": 1436470199,
    "CreatedBy": "/bin/sh -c #(nop) ADD file:8e75593669277501b976
in /",
    "Tags": null,
    "Size": 134740550,
    "Comment": ""
  }
]
```

我们比对一下ubuntu:15.10的Dockerfile，其内容如下：

```
FROM scratch
ADD ubuntu-wily-core-cloudimg-amd64-root.tar.gz /

# a few minor Docker-specific tweaks
# see https://github.com/Docker/Docker/blob/master/contrib/mkimag
RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
    && echo 'exit 101' >> /usr/sbin/policy-rc.d \
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

通过history查询出Ubuntu image有四次文件变更历史，而在Dockerfile中也有四次文件操作，分别是：向image复制文件、执行软件安装、执行脚本命令和设置CMD。

这四次正好与history所查询出的四次文件变更历史相对应。这就说明了，Dockerfile中每执行一次命令，都会产生一个临时image。当所有命令都执行完毕后，最后使用的image就是所有临时image数据叠加后的产物。

27. 向**registry**上传镜像

API	/images/(name)/push
API 类型	POST
API 参数	Name:字符串。image name，用来解析 registry 地址。
附加参数	tag X-Registry-Auth
API 返回值	200：上传成功 404：指定 image 不存在 500：服务器处理错误

①API说明

Docker是通过符合镜像数据规范的image进行应用分发的，而分发最好的途径就是通过各种registry。

但如何将image上传到registry就是一个值得关注的问题了。

这里的API就是为了解决这个问题而产生的。通过这个API，我们可以通过发送rest请求的方式将Daemon Server所在环境中的镜像上传至指定registry中，而团队中的其他成员或者其他用户就可以使用你所上传的image了。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

中去。后面就很简单了，Daemon逐层将镜像的文件信息发送给registry，registry接收之后再逐层进行处理。

有的读者会问到，有一些镜像名称没有这些信息，难道这些镜像就没法上传了吗？答案是否定的。当没有上述那些hostname和port信息时，Daemon将会把这些镜像上传到hub registry中去。

由此可以看出，Daemon在上传镜像时会优先判断对端是否是private registry，如果是则上传，如果不是则上传到hub registry。

把这些解释清楚之后，这个push API就很容易理解了。我们先上传一个镜像到private registry，这次就以上面我们拿来讲解registry的192.168.73.12:5001/ubuntu:latest为例。

我们直接上传它：（注意一定要设置X-Registry-Auth。即便不需要认证，也要赋值为{}）

```
192.168.73.12:3600/images/192.168.73.12:5001/ubuntu/push?tag=late
```

等待片刻后，Daemon返回如下信息：

```
{"status":"The push refers to a repository [192.168.73.12:5001/Ub
{"status":"Sending image list"}
{"status":"Pushing repository 192.168.73.12:5001/Ubuntu (1 tags)"
{"status":"Pushing","progressDetail":{},"id":"aac1ad7ac32f"}{"sta
此处略过进度信息和状态信息,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

但是上面介绍的API中，却没有设置镜像名称的API，这样又如何达到自由控制镜像的目的呢？

为了解决这个问题，/tag API诞生了，使用这个API可以设置镜像名称。镜像名称和镜像标签有什么区别呢？这里有一个范围的问题，例如：192.168.73.12:5001/Ubuntu:15.10，这整个字符串叫做镜像名称，其中92.168.73.12:5001/Ubuntu称为repository，15.10称为tag。

但是当我们提到需要设置标签时，这个标签就不单单指的是15.10了。这时的标签可以是15.10，也可以指192.168.73.12:5001/Ubuntu:15.10。

所以当提到标签时，需要根据当时的场景来决定到底指的是名称中的标签，还是整个名称。

②API参数用法

repo	字符串，用来表示repository，合法字符范围为：[a-z0-9-_.]
force	布尔值，接受1/True/true或者0/False/false，默认为false
tag	字符串，表示新的tag值

③API示例

我们仍以hello-world为例演示此API，首先看一下目前hello-world的镜像名称：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
[
  {
    "Id": "91c95931e552b11604fea91c2f537284149ec32fff0f700a4769c
    "ParentId": "a8219747be10611d65b7c693f48e7222c0bf54b5df8467d
    "RepoTags": [
      "hello-world:latest",
      "192.168.73.12:5001:remote"
    ],
    "RepoDigests": [],
    "Created": 1429308098,
    "Size": 0,
    "VirtualSize": 910,
    "Labels": {}
  }
]
```

加黑部分表示刚才的重命名操作成功，后续就可以使用push API进行上传。

细心的读者可能会发现，/tag API中有三个参数，但我们只使用了repo和tag这两个参数，还有一个force参数没使用。望文生义，感觉force应该是强制的意思，到底是不是呢？我们再发送一次刚才的rest请求：

```
192.168.73.12:3600/images/hello-world:latest/tag?repo=192.168.73
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

库中查找镜像。

②API参数用法

term	字符串，合法字符范围为[a-z0-9-_.]。用来指定需要查询的image name。如果image name中包含private registry信息，则会在private registry中查询；如果没有，则会在Docker hub registry中查询

③API示例

在上面介绍term时，已经提到通过term参数可以指定查询范围。假设我们需要在private registry中查询Ubuntu image是否存在，而这个private registry是192.168.73.12:5001，那么我们就需要发送如下rest请求：

```
192.168.73.12:3600/images/search?term=192.168.73.12:5001/library/
```

注意

在term参数中不能输入tag信息，例如：latest。

Daemon查询到信息后，会返回如下信息：

```
[
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "star_count": 2001,
    "is_official": true,
    "name": "Ubuntu",
    "is_trusted": false,
    "is_automated": false,
    "description": "Ubuntu is a Debian-based Linux operating syst
  },
  {
    "star_count": 28,
    "is_official": true,
    "name": "Ubuntu-upstart",
    "is_trusted": false,
    "is_automated": false,
    "description": "Upstart is an event-based replacement for the
processes at boot"
  },
  {
    "star_count": 11,
    "is_official": true,
    "name": "ubuntu-debootstrap",
    "is_trusted": false,
    "is_automated": false,
    "description": "debootstrap --variant=minbase --components=ma
ping,iproute2 <suite> /"
  },
  {
    "star_count": 2,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

但有时此API并不能查询到结果，因为当Daemon执行查询时，只是充当了另外一个客户端而已，真正执行查询任务的是对方远程的registry，所以能否查询到结果，或者能否执行这个查询任务，其实更依赖于远程的registry。比如，当远程registry关闭了rest API之后，我们的查询就会报错，返回如下：

```
Unexpected status code 404
```

当读者使用这个API时，需要注意处理异常情况。建议在使用之前最好能确认一下对方远程registry是否允许执行此API。

30. 验证认证信息

API	/auth
API 类型	POST
API 参数	无
附加参数	JSON 认证字符串
API 返回值	200：用户新认证成功/用户信息创建成功 500：服务器处理错误

①API说明

如果registry开启了用户认证，那么当用户需要向registry申请资源时，就需要提交认证信息。认证信息通常是在发送API时单独提交，例如，执行/images/(name)/push操作时，可以在X-Registry-Auth参数中附加认证信息。

也可以通过本节将要介绍的认证API提交，通过这个API可以完成用户认证，用户注册功能。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

需要注意serveraddress属性，这个属性用来表示registry地址，默认是https://index.docker.io/v1/。

然后将上述JSON字符串添加到request body中，同时将Content-Type置为application/json，然后发送如下rest请求：

```
192.168.73.12:2376/auth
```

如果上述信息都正确的话，Daemon将会返回200响应码，同时返回如下信息：

```
{
  "Status": "Login Succeeded"
}
```

如果认证信息中输入的username和email在Docker hub registry中都不存在，则registry便会使用上面的信息创建一个新用户，然后返回200响应码和如下信息：

```
{
  "Status": "Account created. Please use the confirmation link we
}
```

用户需要登录邮箱中，激活此账户后方可使用registry。如果在没有

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

运行过程中随时都可能会变更的信息。

②API参数用法：无。

③API示例

我们现在可以查询本章当中一直用来演示的Daemon信息，看看会返回哪些信息。我们发送如下rest请求：

```
192.168.73.12:3600/info
```

Daemon处理后，返回如下信息：

```
{
  "ID": "ET6I:D32F:DF37:M5BI:ACKC:TT57:O7VS:ICYW:JZT6:FMCW:ADTV:Z
  "Containers": 0,
  "Images": 15,
  "Driver": "aufs",
  "DriverStatus": [
    [
      "Root Dir",
      "/var/lib/Docker/aufs"
    ],
    [
      "Backing Filesystem",
      "tmpfs"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "InsecureRegistryCIDRs": [
      "127.0.0.0/8"
    ],
    "IndexConfigs": {
      "192.168.73.12:5001": {
        "Name": "192.168.73.12:5001",
        "Mirrors": [],
        "Secure": false,
        "Official": false
      },
      "192.168.73.12:5002": {
        "Name": "192.168.73.12:5002",
        "Mirrors": [],
        "Secure": false,
        "Official": false
      },
      "Docker.io": {
        "Name": "Docker.io",
        "Mirrors": null,
        "Secure": true,
        "Official": true
      }
    }
  },
  "InitSha1": "",
  "InitPath": "/usr/local/bin/Docker",
  "NCPU": 1,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "crv": "P-256",
    "d": "IcAz-nQYMdV7Y9k0VLT1Oepq3GD1d4x43YsUqSRb8JU",
    "kid": "ET6I:D32F:DF37:M5BI:ACKC:TT57:O7VS:ICYW:JZT6:FMCW:AD
    "kty": "EC",
    "x": "HR0YfbvhznPRz1RpUGKZVOpaH88mCVKsWGxXoBCEZSw",
    "y": "5lEYUWa0upEtMLZfr8ZH5EB8_8eXQMCV0ATcpMojhJ0"
}
```

这个key.json是Docker安全认证的基础。Docker采用两种算法来产生秘钥，一种是RSA，另外一种是EC。

什么是EC算法呢？EC（椭圆曲线算法）是代替RSA的强有力的竞争者。目前SET协议的制定者已把它作为下一代SET协议中默认的公钥密码算法使用，所以EC有望取代RSA，成为通用的公钥加密算法。感兴趣的读者可以查阅相关资料加深理解。与RSA相比，RSA虽然使用非常广泛，但EC在加密算法计算量、安全性和存储空间上都比RSA要高效得多，所以Docker默认采取EC加密算法。

我们再回头看一下key.json文件，在key.json中，kty属性表示的就是EC算法，crv属性表示使用EC算法时的加密长度，P-256表示的是256位加密，可选参数还有P-384、P-512。

d属性表示的是采用EC算法时，大数（数学意义上面的大数）是多少。这个值是Daemon随机产生的大数，然后按位取值后进行base64编码得到的。

x和y表示采取EC算法时，X轴和Y轴离散数据点坐标值。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

上面的API侧重显示Docker运行环境的相关信息，而本节的API侧重显示Docker自身的版本信息。通过此API可以获取到版本信息、内核版本、Golang版本信息和其所支持的API版本信息。

②API参数用法：无

③API示例

我们发送如下rest请求来获取Docker版本信息：

```
192.168.73.12:3600/version
```

Daemon返回200响应码，同时返回如下信息：

```
{
  "Version": "1.7.1",
  "ApiVersion": "1.19",
  "GitCommit": "786b29d",
  "GoVersion": "go1.4.2",
  "Os": "linux",
  "Arch": "amd64",
  "KernelVersion": "4.0.7-boot2Docker"
}
```

这里面ApiVersion需要关注一下。有些API存在于不同的版本中，

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

Daemon返回200响应码，同时也会返回OK：

```
OK
```

这个API虽简单却很实用，不仅可以用来做通信链路检测，还可以兼职用来做Docker server的负载能力测试。当远程Docker server负载压力大时，其返回OK时间就会相应变长，所以可以用此值简单判断负载能力。

但管理Docker集群最好使用Docker官方提供的swarm工具。

34. 基于容器创建镜像

API	/commit
API 类型	POST
API 参数	无
附加参数	container, repo, tag, comment, author, config
API 返回值	204：创建成功 404：指定容器不存在 500：服务器处理错误

①API说明

前面我们介绍过一个创建镜像的API，叫做/images/create，其中提到创建镜像有三种方法，此处的API就是上面提到的第三种方法：以容器为基础创建镜像。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
[
  {
    "Id": "c5ea97b5c597346bf39c5d4691c9a78bfa03071adbf31ba263b0ec
    "Names": [
      "/happy_jones"
    ],
    "Image": "192.168.73.12:5001/library/ubuntu-dev:latest",
    "Command": "/bin/bash",
    "Created": 1438735562,
    "Ports": [
      {
        "PrivatePort": 8088,
        "Type": "tcp"
      },
      {
        "PrivatePort": 8080,
        "Type": "tcp"
      },
      {
        "PrivatePort": 80,
        "Type": "tcp"
      },
      {
        "PrivatePort": 22,
        "Type": "tcp"
      }
    ],
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
[
  {
    "Id": "6622ec20ffa5881d83b597622b44679a2e24cf713c5e03c3891db7
    "ParentId": "e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2
    "RepoTags": [
      "baseconimage:V1.0"
    ],
    "RepoDigests": [],
    "Created": 1438737080,
    "Size": 0,
    "VirtualSize": 760523598,
    "Labels": {}
  },
  {
    "Id": "e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3c178
    "ParentId": "365f2af56f23411f84b8c9573b165b3fa666baebe56f165a
    "RepoTags": [
      "192.168.73.12:5001/library/ubuntu-dev:latest"
    ],
    "RepoDigests": [],
    "Created": 1427177078,
    "Size": 0,
    "VirtualSize": 760523598,
    "Labels": null
  }
]
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
      "8080/tcp": {},
      "8088/tcp": {}
    },
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
    ],
    "Cmd": [
      "/bin/bash"
    ],
    "Image": "",
    "Volumes": null,
    "VolumeDriver": "",
    "WorkingDir": "",
    "Entrypoint": null,
    "NetworkDisabled": false,
    "MacAddress": "",
    "OnBuild": null,
    "Labels": {}
  },
  "Architecture": "amd64",
  "Os": "linux",
  "Size": 0,
  "VirtualSize": 760523598
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
     "WorkingDir": "",
     "NetworkDisabled": false,
     "ExposedPorts": {
     }
}
```

我们这里再次以c5ea97b5c597为基础创建一个新镜像，新镜像名称仍然为baseconimage，tag变为V1.1，同时增加对外9999端口映射，CMD变为date，Hostname设定为newConfigImage。

为了完成上述操作，我们首先准备config参数：

```
{
     "Hostname": "newConfigImage",
     "Domainname": "",
     "User": "",
     "AttachStdin": false,
     "AttachStdout": true,
     "AttachStderr": true,
     "PortSpecs": null,
     "Tty": false,
     "OpenStdin": false,
     "StdinOnce": false,
     "Env": null,
     "Cmd": [
             "date"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
{
  "Id": "f95964dc431d4d239dcd9734d14a5cc32e4cf1180398d7983e414053
  "Parent": "e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3c1
  "Comment": "",
  "Created": "2015-08-05T01:44:40.673277319Z",
  "Container": "c5ea97b5c597346bf39c5d4691c9a78bfa03071adbf31ba26
  "ContainerConfig": {
    "Hostname": "c5ea97b5c597",
    "Domainname": "",
    "User": "",
    "AttachStdin": true,
    "AttachStdout": true,
    "AttachStderr": true,
    "PortSpecs": null,
    "ExposedPorts": {
      "22/tcp": {},
      "80/tcp": {},
      "8080/tcp": {},
      "8088/tcp": {}
    },
   "Tty": true,
    "OpenStdin": true,
    "StdinOnce": true,
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
    ],
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
      "8088/tcp": {},
      "9999/tcp": {}
    },
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
    ],
    "Cmd": [
      "date"
    ],
    "Image": "",
    "Volumes": null,
    "VolumeDriver": "",
    "WorkingDir": "",
    "Entrypoint": null,
    "NetworkDisabled": false,
    "MacAddress": "",
    "OnBuild": null,
    "Labels": {}
  },
  "Architecture": "amd64",
  "Os": "linux",
  "Size": 5,
  "VirtualSize": 760523603
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "Type": "tcp"
      },
      {
        "PrivatePort": 8080,
        "Type": "tcp"
      },
      {
        "PrivatePort": 80,
        "Type": "tcp"
      },
      {
        "PrivatePort": 22,
        "Type": "tcp"
      }
    ],
    "Labels": {},
    "Status": "Up 2 seconds"
  }
]
```

可以看到新映射的9999端口已经启动了，但cmd的值为什么不是date呢？这是因为为了不让容器退出，我们在启动时指定了/bin/bash。如果不指定cmd，那么此处的cmd将会是我们在config中指定的date命令。

35. 监控**Docker**事件

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

until	UNIX时间戳，筛选截止到这个时间戳的事件
filters	JSON字符串，筛选指定类的事件

③API示例：

在使用这个API时，UNIX时间戳是必不可少的条件。我们首先通过date+%s获取当前环境中的时间戳。在笔者的环境中，当前UNIX时间戳是1438741502。

下面我们逐个参数来演示。

Since参数：我们将时间戳向前提前两天，这样才能获取到更多的事件，所以就变为since=1438568700。将这个参数发送给Daemon：

```
GET 192.168.73.12:2376/events?since=1438568700
```

Daemon处理这个请求时会不断返回数据。如果此时使用postman之类的工具进行测试，则无法显示结果。我们换用curl来发送请求，Daemon处理后返回如下信息：

```
D:\>curl -X GET 192.168.73.12:2376/events?since=1438568700
{"status":"pull","id":"hello-world:latest","time":1438755921}
{"status":"untag","id":"91c95931e552b11604fea91c2f537284149ec32f
:1438756068}
{"status":"delete","id":"91c95931e552b11604fea91c2f537284149ec32
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

- event：特定的事件
- image：筛选特定image发出的事件
- container：筛选特定container发出的事件。

这里我们使用event来演示filters参数，image、container的使用和event类似。我们发送如下请求来筛选delete事件：

```
192.168.73.12:2376/events?until=1438756490&since=0&filters={"ever
```

Daemon处理结束后，返回如下数据：

```
{"status":"delete","id":"91c95931e552b11604fea91c2f537284149ec32f
":1438756068}
{"status":"delete","id":"a8219747be10611d65b7c693f48e7222c0bf54b5
":1438756068}
```

可见Daemon已经把不在filters内的其他事件都剔除出去了。

36. Image Tar包文件格式

Tar包说明

在介绍导出容器时提到过，Docker可以将容器导出为一个tar包，然后用户将tar包分发给其他人。

同样镜像也可以导出为一个tar包。如何导入/导出tar包，在后面的

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

37. 打包指定镜像

API	/images/(name)/get
API 类型	GET
API 参数	image name
附加参数	无
API 返回值	200：打包成功 500：服务器处理错误

①API说明

容器可以导出，镜像同样也可以导出，通过这个API就可以导出指定的镜像。导出后的tar包，就是上面所介绍的tar包格式。

导出之后就可以通过其他方式分发此tar包，虽然效率比使用registry要低很多，但效果是一样的。

②API参数用法

name	字符串，只能输入image name。 如果输入的是完整的image name，如ubuntu:latest，则Daemon就会将对应的image打包输出。 如果输入的image name不带tag信息，比如只输入ubuntu，那么Daemon会将所有的ubuntu image都打包输出

③API示例

假设我们要将hello-world打包输出，那么只需要发送：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

而/images/get API则解决了这个问题。使用这个API可以一次性打包多个镜像，其他用户导入这个tar包后会一次性地导入多个镜像，这样极大地提高了效率。

②API参数用法

names	字符串，输入image name。 如果输入的是完整的image name，如ubuntu:latest，则Daemon就会将对应的image打包输出。 如果输入的image name不带tag信息，比如只输入ubuntu，那么Daemon会将所有的ubuntu image都打包输出

③API示例

如果我们尝试一次性地将ubuntu和busybox这两个镜像所有的版本都打包导出，那么我们仅需多次使用name赋值就可以。需要发送的rest请求如下：

```
curl -X GET 192.168.73.12:2376/images/get?names=hello-world:lates
```

其中，ubuntu:latest表示仅打包latest版本的镜像，而后面的busybox，则表示将所有tag版本的busybox都打包导出。

导出后的tar包结构如下：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

API	/images/load
API 类型	POST
API 参数	无
附加参数	无
API 返回值	200：加载成功 500：服务器处理错误

①API说明

有进有出才算正常。前面两个API都是介绍如何导出，我们这里介绍如何将tar包内的镜像导入到Docker环境中。

读者看到这个API时，应该看到此API没有任何参数。那么数据如何传入到Docker里面呢？如果读者能联想到导入容器时的API，就应该明白我们不是依靠参数将tar包传入到Docker内部的，而是将tar包内的数据以二进制的方式放入request body中传给Docker Daemon的。

剩下的工作就很简单了，将拿到手的tar包以二进制的方式发给Docker Daemon就行。因为tar包有Repositories文件，所以Daemon通过解析这个文件就可以完成导入工作了。

②API参数用法：无。

③API示例

我们现在开始演示如何导入tar包。首先需要将环境中已有的busybox和hello-world两个镜像都删掉，如下所示：

```
[
  {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
      "192.168.73.12:5001/library/ubuntu-dev:latest"
    ],
    "RepoDigests": [],
    "Created": 1427177078,
    "Size": 0,
    "VirtualSize": 760523598,
    "Labels": null
  }
]
```

目前环境中仅仅保存有baseconimage和Ubuntu-dev两个镜像，然后我们发送如下rest请求：

```
192.168.73.12:2376/images/load
```

最后以二进制的形式将刚才导出的tar包添加到body中，如图5-5所示。



图5-5　将tar包添加到body中

当Daemon返回200响应码后，我们再次查看一下当前环境中的镜像列表：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "Id": "91c95931e552b11604fea91c2f537284149ec32fff0f700a4769c1
    "ParentId": "a8219747be10611d65b7c693f48e7222c0bf54b5df8467d3
    "RepoTags": [
      "hello-world:latest"
    ],
    "RepoDigests": [],
    "Created": 1429308098,
    "Size": 0,
    "VirtualSize": 910,
    "Labels": {}
  },
  {
    "Id": "8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e391
    "ParentId": "6ce2e90b0bc7224de3db1f0d646fe8e2c4dd37f17939282
    "RepoTags": [
      "busybox:latest"
    ],
    "RepoDigests": [],
    "Created": 1429308073,
    "Size": 0,
    "VirtualSize": 2433303,
    "Labels": {}
  },
  {
    "Id": "e9453316c6301900e3c445a798a816ea5a345c4d3a0835f2e3c178
    "ParentId": "365f2af56f23411f84b8c9573b165b3fa666baebe56f165a
    "RepoTags": [
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

执行实例创建时，Daemon会把执行实例与容器做映射，映射成功后，执行实例就处于待执行状态。

需要注意的是，执行实例仅仅是保存在内存中的配置信息，一旦与之映射的容器关闭、销毁或者Daemon关闭，所有创建成功的执行实例都将消失。

②API参数用法

attachStdin	布尔值，接受 1/True/true 或者 0/False/false，默认为 false。 表示是否需要在执行的命令中开启标准输入
attachStdout	布尔值，接受 1/True/true 或者 0/False/false，默认为 false 表示是否需要在执行的命令中开启标准输出
attachStderr	布尔值，接受 1/True/true 或者 0/False/false，默认为 false 表示是否需要在执行的命令中开启标准错误
tty	布尔值，接受 1/True/true 或者 0/False/false，默认为 false 表示是否需要在执行的命令中申请伪终端
cmd	字符串，必输项。用来输入准备在此执行实例中执行的命令

注意

上述所有参数都需要通过JSON字符串形式提交到Daemon。

③API示例

假设我们需要在目前已经存在的Ubuntu-dev容器中执行一条date命令，所以需要创建对应此命令的执行实例。

首先组织一下所需的参数字符串，如下所示：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

号，如下所示：

```
{
  "Id": "cf2cb1073c18eeab01ab516399832453ba12f3816f7615f2f79dc49e
}
```

这个ID号和镜像ID，容器ID类似，都是64位UUID值。作用同样是用于保证在整个Docker Daemon生存周期内唯一。

当拿到这个ID值后，我们就可以使用下面即将介绍到的API来执行此命令。

41. 启动执行实例

API	/exec/(id)/start
API 类型	POST
API 参数	执行实例全 ID
附加参数	detach tty
API 返回值	201：执行成功 404：指定的执行实例不存在

①API说明

当使用上面的API创建好执行实例后，我们就可以使用这个API来启动此执行实例了。在执行实例启动时，Daemon会将此实例中的命令放入到与之关联的容器中。然后通过容器中的环境来执行，并且将结果输出到stdout中。

②API参数用法

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
 "Tty": false
}
```

然后使用执行实例id来发送rest请求就可以了。注意：这里不能使用短id，必须输入全id。

```
192.168.73.12:2376/exec/ cf2cb1073c18eeab01ab516399832453ba12f381
00dfc8/start
```

因为我们设定为stdout=true，因此Daemon返回了date执行后的结果，如下所示：

```
Thu Aug  6 01:37:50 UTC 2015
```

42. 重置执行实例中的**tty**尺寸

API	/exec/(id)/resize
API 类型	POST
API 参数	执行实例全 ID
附加参数	h w
API 返回值	201：重置成功 404：指定的执行实例不存在

①API说明

在介绍容器API时，也提到过一个调整TTY尺寸的API。此处的API

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

API	/exec/(id)/json
API 类型	POST
API 参数	执行实例全 id
附加参数	无
API 返回值	200：获取成功 404：指定的执行实例不存在 500：服务器处理错误

①API说明

与镜像和容器一样，我们也可以获取每个执行实例的详细信息。在详细信息中，我们会看到执行实例当前状态，需要执行的命令，与之关联的容器信息，等等。

②API参数用法：无。

③API示例

我们尝试获取d2806e37f737d702c0f277a5a4af48771b0bc47aaa1b01f5fde10511d3225ef8这个执行实例的详细信息，直接发送如下rest请求：

```
192.168.73.12:2376/exec/d2806e37f737d702c0f277a5a4af48771b0bc47aa
```

```
{
  "ID": "d2806e37f737d702c0f277a5a4af48771b0bc47aaa1b01f5fde1051
  "Running": false,
  "ExitCode": 0,
  "ProcessConfig": {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    "ID": "ba30f5e01c61e0e8519b335b2130c7f5432adf9cd9d3c85fd68b7c
    "Created": "2015-08-06T00:24:39.400200388Z",
    "Path": "/bin/bash",
    "Args": [],
    "Config": {
      "Hostname": "ba30f5e01c61",
      "Domainname": "",
      "User": "",
      "AttachStdin": true,
      "AttachStdout": true,
      "AttachStderr": true,
      "PortSpecs": null,
      "ExposedPorts": {
        "22/tcp": {},
        "80/tcp": {},
        "8080/tcp": {},
        "8088/tcp": {},
        "9999/tcp": {}
      },
      "Tty": true,
      "OpenStdin": true,
      "StdinOnce": true,
      "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
      ],
      "Cmd": [
        "/bin/bash"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
      "PortMapping": null,
      "Ports": {
        "22/tcp": null,
        "80/tcp": null,
        "8080/tcp": null,
        "8088/tcp": null,
        "9999/tcp": null
      },
      "SandboxKey": "/var/run/Docker/netns/ba30f5e01c61",
      "SecondaryIPAddresses": null,
      "SecondaryIPv6Addresses": null
    },
    "ResolvConfPath":
"/var/lib/Docker/containers/ba30f5e01c61e0e8519b335b2130c7f5432ad
olv.conf",
    "HostnamePath":
"/var/lib/Docker/containers/ba30f5e01c61e0e8519b335b2130c7f5432ad
tname",
    "HostsPath":
"/var/lib/Docker/containers/ba30f5e01c61e0e8519b335b2130c7f5432ad
ts",
    "LogPath":
"/var/lib/Docker/containers/ba30f5e01c61e0e8519b335b2130c7f5432ad
0f5e01c61e0e8519b335b2130c7f5432adf9cd9d3c85fd68b7d6bd161a70d-jso
    "Name": "/nostalgic_jones",
    "Driver": "aufs",
    "ExecDriver": "native-0.2",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

可以让Docker在集群中高效运行。

Docker-compose属于应用层的工具，主要用于Docker容器在集群中快速部署。利用Docker-compose，用户可以自定义哪个容器运行哪个应用。与此同时，Docker-compose也支持热部署，而且可以根据负载情况随时扩展，如图5-6所示。

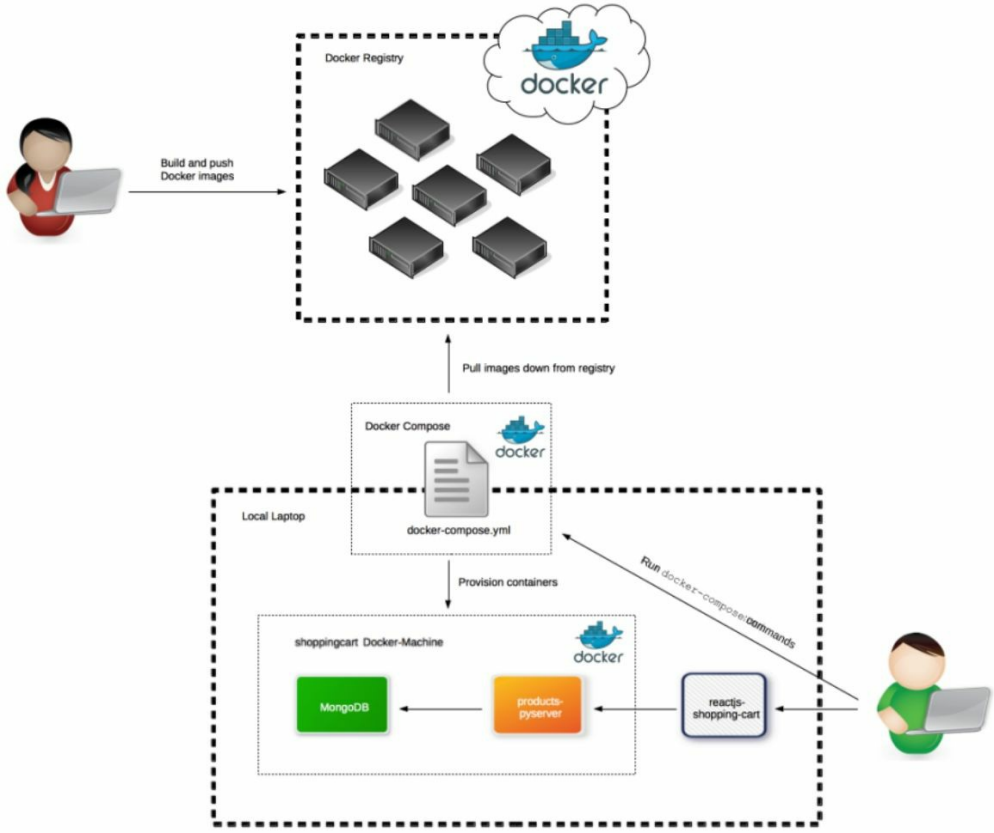


图5-6　Docker工作原理

安装Docker时，默认是不提供Docker-compose工具的（MAC版本默认提供）。因此在使用Docker-compose之前，我们需要先安装它。

1. 安装**Docker-compose**

Docker-compose仅支持在MAC OS X和64位Linux操作系统中运行，

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
$ chmod +x /usr/local/bin/Docker-compose
```

在安装完成之后，执行下面的命令测试安装是否成功：

```
$ Docker-compose --version
Docker-compose version: 1.4.0
```

2. 卸载**Docker-compose**

当需要卸载Docker-compose时，直接删除Docker-compose即可：

```
$ rm /usr/local/bin/Docker-compose
```

如果用户是使用pip安装的Docker-compose，那么可执行下面的命令执行删除：

```
$ pip uninstall Docker-compose
```

3. 使用**Docker-compose**

当Docker-compose安装成功后，我们就可以在当前环境中使用这个工具了。下面看一下Docker-compose如何使用。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  up                  Create and start containers
  migrate-to-labels   Recreate containers to add labels
  version             Show the Docker-Compose version information
```

Docker-compose的参数分为两类：选择参数和命令参数。选择参数主要用于指定Docker-compose配置文件和容器所属的工程名称，而命令参数则是用来管理服务的。

4. 选择参数

在选择参数中，-f是用来指定Docker-compose工具配置文件的，这个参数属于可选项参数。如果不指定-f，那么Docker-compose将在当前工作目录中搜索名为“Docker-compose.yaml”的配置文件。如果当前工作目录没有此文件，则会在父目录中搜索此文件，直到检索到此文件为止。

如果使用“-”代替文件名，那么Docker-compose将会从STDIN中读取数据作为配置文件中的内容加载使用，如下所示：

```
##直接使用Docker-compose.yaml作为配置文件
$Docker@boot2Docker:～$ Docker-compose  up –d
Creating Docker_tomcat_1...
     Name              Command       State            Ports
------------------------------------------------------------------
Docker_tomcat_1   catalina.sh run   Up      0.0.0.0:32769->8080/t
##通过-f指定配置文件
Docker@boot2Docker:～$ Docker-compose -f Docker-compose-tomcat.ym
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

工程名称的使用，我们将在下面给出示例。

5. 配置文件

Docker-compose中的命令参数会用到配置文件中的一些概念，所以在讲解命令参数之前，先了解一下配置文件。

从上面的示例可以看出，Docker-compose使用的配置文件其实是一个yaml文件。在这个yaml文件中，按照KEY-VALUE格式定义各项参数，然后Docker-compose通过解析yaml文件来获取用户的操作意图，从而达到自动化操作的目的。

yaml配置文件中的参数可多可少，最少只需要包括镜像名称，最多可以调整容器所有的运行参数。

在配置yaml文件时，用户需要把握住yaml文件的本质。yaml文件只是帮助用户达到快速部署的目的，如果容器参数在构建镜像时已经被指定了，并且部署时不需要再调整，那么就没有必要体现在yaml文件中。

下面我们看一下yaml文件的格式：

```
tomcat:
  image: 192.168.73.12:5001/tomcat
```

这是一个最简单的Docker-compose配置文件，其中，Tomcat被称为service name，是必填项。在Docker-compose配置文件中，有两项必须要指定：一项是service name，另外一项是image或者build。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

在Docker-compose配置文件中，只有服务名称和服务来源是必填项，其他都是可选项。

表5-11中列出了Docker-compose配置文件中可以修改的参数项。

表5-9　可修改的参数项

参数名称	参数用法	参数示例
Dockerfile	当服务来源选择为 build 时，指定 Dockerfile	Dockerfile: Dockerfile-tomcat
Command	覆盖镜像中所设置的 COMMAND 参数值	command: bundle exec thin -p 3000
Links	当此服务需要链接其他服务时，可以通过此项来实现	links: - db - db:database - redis 表示此服务需要链接其他三个服务。当链接成功后，在此服务的/etc/hosts 中会出现三项记录： 172.17.2.186　db 172.17.2.186　database 172.17.2.187　redis
external_links	和 links 不同，此项表示此服务需要链接此配置文件之外的容器	用法同 links 相同，如下： external_links: - redis_1 - project_db_1:mysql - project_db_2:postgresql
extra_hosts	维护容器中的 hosts 文件，可以直接添加主机信息	extra_hosts: - "somehost:162.242.195.82" - "otherhost:50.31.209.229" 维护成功后，在/etc/hosts 中会出现如下记录： 162.242.195.82　somehost 50.31.209.229　otherhost
ports	维护端口映射， 规则为<主机端口>:<容器端口>。 当只有容器端口时，将在主机空闲端口中随机映射一个	ports: - "3000" - "8000:8000" - "49100:22" - "127.0.0.1:8001:8001"

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

续表

参数名称	参数用法	参数示例
labels	覆盖此容器的 labels 信息	labels: com.example.description: "Accounting webapp" com.example.department: "Finance" com.example.label-with-empty-value: "" labels: - "com.example.description=Accounting webapp" - "com.example.department=Finance" - "com.example.label-with-empty-value"
container_name	定义容器名称	container_name: my-web-container 如果自定义容器名称之后，那么将无法使用 scale 命令
log driver	覆盖日志驱动参数	log_driver: "syslog" log_opt: address: "tcp://192.168.0.42:123"
net	覆盖网络访问方式	net: "bridge" net: "none" net: "container:[name or id]" net: "host"
pid	启用共享模式	pid: "host"。 当 pid 置为 host 后，此服务所有容器将共用同一个主机进程，此时每个容器都有访问其他容器命名空间和虚拟化环境的权限
dns	设置容器的 DNS 设置	dns: 8.8.8.8 dns: - 8.8.8.8 - 9.9.9.9
cap_add, cap_drop	设置容器的内核权限	cap_add: - ALL cap_drop: - NET_ADMIN - SYS_ADMIN
dns_search	设置容器的 DNS_SEARCH	dns_search: example.com dns_search: - dc1.example.com - dc2.example.com
devices	设置容器可加载的块设备	devices: - "/dev/ttyUSB0:/dev/ttyUSB0"
security_opt	设置容器的安全参数	security_opt: - label:user:USER - label:role:ROLE

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

build参数用于构建镜像。在介绍配置文件时，我们提到过服务来源有两处：一是来自于已有镜像，二是构建生成的镜像。

在选择构建生成镜像时，每次执行Docker-compose此工具都会自动构建一次。如下所示：

```
Docker@boot2Docker:～$ Docker-compose -f Docker-compose-build.yml
Building tomcat...
Step 0 : FROM 192.168.73.12:5001/tomcat
 ---> 6fe1972e6b08
Successfully built 6fe1972e6b08
```

当选择--no-cache=true后，会放弃cache中的镜像数据。

（2）kill参数

```
Usage: kill [options] [SERVICE...]
Options:
    -s SIGNAL         SIGNAL to send to the container.
                      Default signal is SIGKILL.
```

kill参数用于强制关闭服务。如果需要向服务发送指定信号量，则可以通过-s来指定，例如：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

理。如果添加--no-color=true后，将放弃着色。Logs参数示例如下：

```
Docker@boot2Docker:~$ Docker-compose logs postgresql
Attaching to Docker_postgresql_2, Docker_postgresql_1
postgresql_2 | 2015-09-06 09:59:36 GMT LOG:  database system was
GMT
postgresql_2 | 2015-09-06 09:59:36 GMT LOG:  database system is r
postgresql_2 | 2015-09-06 09:59:36 GMT LOG:  autovacuum launcher
postgresql_1 | 2015-09-06 09:59:25 GMT LOG:  database system was
GMT
postgresql_1 | 2015-09-06 09:59:25 GMT LOG:  database system is r
postgresql_1 | 2015-09-06 09:59:25 GMT LOG:  autovacuum launcher
```

（4）port参数

```
Usage: port [options] SERVICE PRIVATE_PORT

Options:
    --protocol=proto  tcp or udp [default: tcp]
    --index=index     index of the container if there are multipl
                      instances of a service [default: 1]
```

port参数可用来查看此服务所映射的主机端口号。例如下例中，Tomcat服务有两个实例：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
Usage: ps [options] [SERVICE...]

Options:
    -q    Only display IDs
```

ps参数用来显示服务状态信息，包括服务中实例名称、实例命令、实例当前状态和实例端口映射信息。

在前面我们曾经提到过-p会有其他用途，这里就可以讲解-p的用途了。首先看下面的例子：

```
Docker@boot2Docker:~$ Docker-compose -f Docker-compose-tomcat.ym
Creating mytomcat_tomcat_1...
Docker@boot2Docker:~$ Docker-compose -f Docker-compose-tomcat.ym
Creating and starting 2... done
```

首先使用Docker-compose创建一个Tomcat服务，并且将其实例扩展为两个。我们使用ps查看一下：

```
Docker@boot2Docker:~$ Docker-compose ps
Name   Command   State   Ports
------------------------------
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

回顾创建服务的命令：

```
Docker-compose -f Docker-compose-tomcat.yml -p mytomcat up -d
```

我们指定了工程名为mytomcat，使用的配置文件为Docker-compose-tomcat.yaml。所以当执行ps命令时，同样需要给Docker-compose指定正确的工程名和所使用的配置文件。因此，正确的ps命令应该是下面的样子：

```
Docker@boot2Docker:~$ Docker-compose -f Docker-compose-tomcat.ym
      Name               Command       State            Ports
-------------------------------------------------------------------
mytomcat_tomcat_1   catalina.sh run   Up      0.0.0.0:32785->8080
mytomcat_tomcat_2   catalina.sh run   Up      0.0.0.0:32786->8080
```

而之前演示的例子之所以没有使用-f和-p，是为了简化命令复杂度突出重点而使用的默认工程和默认配置文件。

在下面的例子中，同样为了突出命令重点而使用了默认值。

（6）pull参数

pull参数用来从指定的镜像仓库下载镜像。如果我们在配置文件中指定了服务镜像来源为image方式。那么执行pull时，Docker-compose会检查本地是否存在指定的镜像。如果不存在则会根据镜像名称解析出仓

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
Status: Image is up to date for 192.168.73.12:5001/tomcat:latest
```

（7）restart参数

```
Usage: restart [options] [SERVICE...]

Options:
  -t, --timeout TIMEOUT      Specify a shutdown timeout in second
                             (default: 10)
```

restart是重启服务的命令。使用restart命令，可以将指定配置文件中的部分服务或者全部服务下的所有实例全部关闭再启动。

同Docker CLI中的restart命令一样，这里的restart命令也有-t参数。-t用来设置等待关闭时间，当超过此时间后服务实例仍未关闭，那么Docker-compose将会强制关闭此实例。

使用restart参数时请注意，此命令仅对当前运行状态为up的实例有效。restart参数操作示例如下所示：

```
##当前Tomcat服务中存在两个实例
Docker@boot2Docker:~$ Docker-compose -f Docker-compose-tomcat.ym
      Name               Command       State            Ports
-------------------------------------------------------------------
mytomcat_tomcat_1   catalina.sh run   Up      0.0.0.0:32785->8080
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
mytomcat_tomcat_2   catalina.sh run   Exit 143
##restart失败，当前状态仍未Exit
```

（8）rm参数

```
Usage: rm [options] [SERVICE...]

Options:
    -f, --force   Don't ask to confirm removal
    -v            Remove volumes associated with containers
```

rm参数是用来移除服务所有实例的命令，使用方式与Docker CLI中的rm基本一致，当需要强制删除时，需要使用--force参数。如果需要在移除时，一并移除其生成的挂载数据，则使用-v参数。

rm参数使用方式如下所示：

```
Docker@boot2Docker:~$ Docker-compose -f Docker-compose-tomcat.ym
Going to remove mytomcat_tomcat_2, mytomcat_tomcat_1
Are you sure? [yN] y
Removing mytomcat_tomcat_2... done
Removing mytomcat_tomcat_1... done
```

如果仅需要删除指定的服务，则在rm参数后面输入服务名称即可。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

而run就是用来覆盖参数，以达到手工部署的目的。例如，我们希望Tomcat实例可以执行bash，方便在bash中执行其他工具：

```
$ Docker-compose run tomcat bash
```

当执行Docker-compose　run后，将会覆盖配置文件中所定义的部分参数，转而执行run命令中所设定的参数。

虽然run命令可以覆盖配置文件中的部分参数，但还是有两点需要注意：

①不同run命令所创建的服务是相互独立的。比如在Tomcat服务中执行了bash后，也可以同一时间在同一个配置文件的Tomcat服务中执行Python。因此下面的两个操作是合法的：

```
$ Docker-compose run tomcat bash
$ Docker-compose run tomcat python test.py
```

②run命令不会自动创建端口映射。如果手工创建的服务需要与外部网络进行通信，那么就需要添加--service-ports来建立端口映射，例如：

```
$ Docker-compose run tomcat bash
##不会创建端口映射
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
##创建服务
Docker@boot2Docker:~$ Docker-compose ps
     Name              Command       State            Ports
-------------------------------------------------------------------
Docker_tomcat_1   catalina.sh run   Up      0.0.0.0:32795->8080/t
Docker@boot2Docker:~$ Docker-compose scale tomcat=3
Creating and starting 2... done
Creating and starting 3... done
##将实例数扩展为3个
Docker@boot2Docker:~$ Docker-compose ps
     Name              Command       State            Ports
Docker_tomcat_1   catalina.sh run   Up      0.0.0.0:32795->8080/t
Docker_tomcat_2   catalina.sh run   Up      0.0.0.0:32796->8080/t
Docker_tomcat_3   catalina.sh run   Up      0.0.0.0:32797->8080/t
```

scale不仅可以正向增加，同样还可以反向减少，例如：

```
Docker@boot2Docker:~$ Docker-compose scale tomcat=2
Stopping Docker_tomcat_3... done
Removing Docker_tomcat_3... done
##将实例数缩减为2个
Docker@boot2Docker:~$ Docker-compose ps
     Name              Command       State            Ports
-------------------------------------------------------------------
Docker_tomcat_1   catalina.sh run   Up      0.0.0.0:32795->8080/t
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
Docker@boot2Docker:~$ Docker-compose ps
     Name              Command        State               Ports
-------------------------------------------------------------------
Docker_tomcat_1   catalina.sh run   Up         0.0.0.0:32794->808
Docker_tomcat_2   catalina.sh run   Up         0.0.0.0:32793->808
Docker_tomcat_3   catalina.sh run   Up         0.0.0.0:32792->808
```

（12）stop参数

```
Usage: stop [options] [SERVICE...]

Options:
  -t, --timeout TIMEOUT      Specify a shutdown timeout in second
                             (default: 10)
```

stop参数同start相反，是用来停止服务的。stop可以同时停止配置文件中的某些服务或者全部服务。当需要停止特定服务时，需要在stop后面给定服务名，例如：

```
Docker@boot2Docker:~$ Docker-compose stop tomcat
Stopping Docker_tomcat_3... done
Stopping Docker_tomcat_2... done
Stopping Docker_tomcat_1... done
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

相反，无论实例是否存在，都必须重新创建。

--no-build用在如果服务镜像为build时，在创建服务阶段可以不用执行build操作。在实际使用过程中，很少会用到。而最后的-t，同样是关闭超时时间。

上述所有参数中，最常用的是-d，使用方式可参考上面命令示例中的up命令。

第6章 Docker命令剖析

在开始剖析Docker Cli命令之前，首先需要讲解Daemon。因为Daemon在整个Docker命令体系中承担着承上启下的作用。一方面，Daemon需要处理客户端发来的指令；另一方面，Daemon还负责维护整个Docker环境，可谓身兼数职于一身。

承担着如此重担的Daemon本身并不庞大，甚至可以说小得可怜。如此小的身躯是如何扛起这万重的任务呢？这正是我们后面将要阐述的内容。

在本章，我们重点讲解Daemon如何处理客户端指令。而在高级篇中，我们会讲解Daemon如何维护Docker环境。

通过前面章节的介绍，我们得知，Docker既可以当作服务端，例如docker–d，还可以当作客户端，例如docker run XXX。所以Docker本身既是客户端也是服务端。完事开头难，我们要想在纷繁复杂的Docker内核中理出头绪，没有一个入口可不行。Docker作为一切命令的发起者，无疑是一个非常好的入口，所以下面的Docker剖析就从Docker自己开始。

下面是Docker函数的大致流程：

①初始化运行参数。

②判断运行模式（服务端/客户端）。

③如果是服务端，则处理API请求。

④如果是客户端，则执行相对应的命令。

从上面流程中可看到，当Docker函数执行时，会首先判断自己是客户端还是服务端，然后再去执行不同的业务逻辑。这个模式判断尤为重要，下面是其模式判断的代码片段：

```
if *flDaemon {
    if *flHelp {
      flag.Usage()
      return
    }
    mainDaemon()
    return
  }
```

当*flDaemon是true时，表示其为服务模式，然后就去执行mainDaemon进行服务初始化。如果*flDaemon为false，则表示为客户端，跳过初始化进行后面的命令解析。

而*flDaemon是在什么地方初始化的呢？*flDaemon这个参数是在上面流程图的参数初始化模块进行处理的。下面为此阶段参数初始化的代码片段：

```
var (
  flVersion   = flag.Bool([]string{"v", "-version"}, false, "Prin
  flDaemon    = flag.Bool([]string{"d", "-daemon"}, false, "Enabl
```

```
  flDebug     = flag.Bool([]string{"D", "-debug"}, false, "Enable
  flLogLevel  = flag.String([]string{"l", "-log-level"}, "info",
  flTls       = flag.Bool([]string{"-tls"}, false, "Use TLS; impl
  flHelp      = flag.Bool([]string{"h", "-help"}, false, "Print u
  flTlsVerify = flag.Bool([]string{"-tlsverify"}, dockerTlsVerify

  // these are initialized in init() below since their default va
isn't fully initialized until init() runs
  flTrustKey *string
  flCa       *string
  flCert     *string
  flKey      *string
  flHosts    []string
)
```

可以看到不止初始化了*flDaemon参数，还初始化了很多其他参数。这里捎带提一下Golang中是如何对参数进行处理的，例如下面的一行代码：

```
flVersion   = flag.Bool([]string{"v", "-version"}, false, "Print
```

表示此程序可以接受一个-v参数或者--version参数。如果参数中没有-v或者--version，那么flVersion默认值就是false，反之就是true，例如：

```
docker –v ##此时flVersion = true
docker ##此时flVersion = false
docker –v=false ##此时flVersion = false
```

因此，在上面参数中，仅初始化了-v、-d、-D、-l、-tls、-tlsverify这几个参数。因为在Docker函数处理流程中，仅需要判断是否为服务模式，所以初始化其他参数也没有用途。只要参数中有-d，就表示是服务模式。如果没有-d，则表示是客户端模式。

说到这里，我们把模式判断之前的流程梳理一番，如图6-1所示。

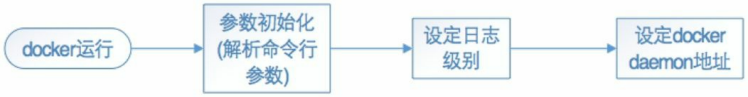


图6-1　模式之前的流程

其代码如下：

```
func main() {
  if reexec.Init() {
    return
  }
  // Set terminal emulation based on platform as required.
  stdin, stdout, stderr := term.StdStreams()

  initLogging(stderr)
  flag.Parse() //解析参数
```

```
// FIXME: validate daemon flags here

if *flVersion {
  showVersion()
  return
}
if *flLogLevel != "" {
  lvl, err := logrus.ParseLevel(*flLogLevel)
  if err != nil {
    fmt.Fprintf(os.Stderr, "Unable to parse logging level: %s\n
    os.Exit(1)
  }
  setLogLevel(lvl)
} else {
  setLogLevel(logrus.InfoLevel)
}
if *flDebug {
  os.Setenv("DEBUG", "1")
  setLogLevel(logrus.DebugLevel)
}

if len(flHosts) == 0 {
  defaultHost := os.Getenv("DOCKER_HOST")
  if defaultHost == "" || *flDaemon {
    if runtime.GOOS != "windows" {
      // If we do not have a host, default to unix socket
      defaultHost = fmt.Sprintf("unix://%s", opts.DefaultUnixSo
```

```
      } else {
        // If we do not have a host, default to TCP socket on Wi
        defaultHost = fmt.Sprintf("tcp://%s:%d", opts.DefaultHTT
      }
    }
    defaultHost, err := opts.ValidateHost(defaultHost)
    if err != nil {
      if *flDaemon {
        logrus.Fatal(err)
      } else {
        fmt.Fprint(os.Stderr, err)
      }
      os.Exit(1)
    }
    flHosts = append(flHosts, defaultHost)
  }

  setDefaultConfFlag(flTrustKey, defaultTrustKeyFile)
  <后续继续讲解>
}
```

参数解析主要代码如下：

```
var (
  flVersion   = flag.Bool([]string{"v", "-version"}, false, "Pri
```

```
  flDaemon    = flag.Bool([]string{"d", "-daemon"}, false, "Enabl
  flDebug     = flag.Bool([]string{"D", "-debug"}, false, "Enable
  flLogLevel  = flag.String([]string{"l", "-log-level"}, "info",
  flTls       = flag.Bool([]string{"-tls"}, false, "Use TLS; impl
  flHelp      = flag.Bool([]string{"h", "-help"}, false, "Print u
  flTlsVerify = flag.Bool([]string{"-tlsverify"}, dockerTlsVerify

  // these are initialized in init() below since their default va
isn't fully initialized until init() runs
  flTrustKey *string
  flCa       *string
  flCert     *string
  flKey      *string
  flHosts    []string
)
//对参数进行初始化
func init() {
  var placeholderTrustKey string
  // TODO use flag flag.String([]string{"i", "-identity"}, "", "P
  flTrustKey = &placeholderTrustKey

  flCa = flag.String([]string{"-tlscacert"}, filepath.Join(docker
certs signed only by this CA")
  flCert = flag.String([]string{"-tlscert"}, filepath.Join(docker
to TLS certificate file")
  flKey = flag.String([]string{"-tlskey"}, filepath.Join(dockerCe
TLS key file")
```

```
  opts.HostListVar(&flHosts, []string{"H", "-host"}, "Daemon sock

  flag.Usage = func() {
    fmt.Fprint(os.Stdout, "Usage: docker [OPTIONS] COMMAND [arg.
linux containers.\n\nOptions:\n")

    flag.CommandLine.SetOutput(os.Stdout)
    flag.PrintDefaults()

    help := "\nCommands:\n"

    sort.Sort(byName(dockerCommands))

    for _, cmd := range dockerCommands {
      help += fmt.Sprintf("    %-10.10s%s\n", cmd.name, cmd.descr
    }

    help += "\nRun 'docker COMMAND --help' for more information o
    fmt.Fprintf(os.Stdout, "%s\n", help)
  }
}
```

可能有的读者会对参数如何进行初始化、如何进行解析不清楚。这里的参数解析使用的是Golang官方提供的parse包。开放时只需按照上面的规则赋值，然后调用flag.Parse()就可以完成参数解析。

赋值时既可以在var声明区赋值，也可以在init函数中进行赋值，flag.Parse()都可以完成解析。

Docker作为服务端时后续处理流程非常多。我们暂时先放到一旁，优先按照客户端的思路往下进行。

当Docker作为客户端运行时，后续的步骤相对就简单多了，主要逻辑就是解析命令。代码如下：

```
var (
    cli       *client.DockerCli
    tlsConfig tls.Config
  )

中间略过部分代码

  if err := cli.Cmd(flag.Args()...); err != nil {
    if sterr, ok := err.(client.StatusError); ok {
      if sterr.Status != "" {
        fmt.Fprintln(cli.Err(), sterr.Status)
        os.Exit(1)
      }
      os.Exit(sterr.StatusCode)
    }
    fmt.Fprintln(cli.Err(), err)
    os.Exit(1)
}
```

首先声明了一个Docker的客户端结构体。代码运行到此处时，就可以使用flag包取出命令，例如images命令、pull命令，等等。然后通过Cli的Cmd函数来处理。我们看一下Cmd函数是如何执行的：

```
func (cli *DockerCli) getMethod(args ...string) (func(...string)
  camelArgs := make([]string, len(args))
  for i, s := range args {
    if len(s) == 0 {
      return nil, false
    }
    camelArgs[i] = strings.ToUpper(s[:1]) + strings.ToLower(s[1:]
  }
  methodName := "Cmd" + strings.Join(camelArgs, "")
  method := reflect.ValueOf(cli).MethodByName(methodName)
  if !method.IsValid() {
    return nil, false
  }
  return method.Interface().(func(...string) error), true
}

// Cmd executes the specified command.
func (cli *DockerCli) Cmd(args ...string) error {
  if len(args) > 1 {
    method, exists := cli.getMethod(args[:2]...)
    if exists {
```

```
      return method(args[2:]...)
    }
  }
  if len(args) > 0 {
    method, exists := cli.getMethod(args[0])
    if !exists {
      return fmt.Errorf("docker: '%s' is not a docker command.\nS
    }
    return method(args[1:]...)
  }
  return cli.CmdHelp()
}
```

通过上面的代码，我们可以知道：当调用Cmd代码时，Docker会使用Cmd+方法名的方式，通过反射在所有代码中查找相匹配的函数，如果找到了函数：

```
method, exists := cli.getMethod(args[:2]...)
if exists {
  return method(args[2:]...)
}
```

即Exists=true的时候，会转而进入到函数内部执行代码。

但在上面的代码中，我们漏掉了一部分代码。那部分代码在做什么

呢？其实那部分代码是用于设置tls证书的。关于安全认证，完全可以再写一本书了，所以这里就简单略过。

因此，我们就可以完整地提炼出当Docker以客户端模式执行时的逻辑，如图6-2所示。

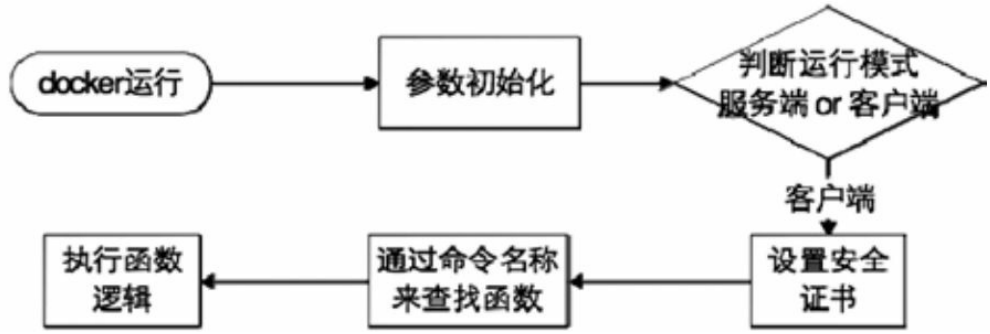


图6-2　以客户端模式执行时的逻辑

现在回头再看Docker代码，就可以理清其中的逻辑了。请看下面的完整代码：

```
func main() {
  if reexec.Init() {
    return
  }
  // Set terminal emulation based on platform as required.
  stdin, stdout, stderr := term.StdStreams()

  initLogging(stderr)
  flag.Parse()
  // FIXME: validate daemon flags here
```

```
if *flVersion {
  showVersion()
  return
}
if *flLogLevel != "" {
  lvl, err := logrus.ParseLevel(*flLogLevel)
  if err != nil {
    fmt.Fprintf(os.Stderr, "Unable to parse logging level: %s\n
    os.Exit(1)
  }
  setLogLevel(lvl)
} else {
  setLogLevel(logrus.InfoLevel)
}

if *flDebug {
  os.Setenv("DEBUG", "1")
  setLogLevel(logrus.DebugLevel)
}

if len(flHosts) == 0 {
  defaultHost := os.Getenv("DOCKER_HOST")
  if defaultHost == "" || *flDaemon {
    if runtime.GOOS != "windows" {
      // If we do not have a host, default to unix socket
      defaultHost = fmt.Sprintf("unix://%s", opts.DefaultUnixSo
    } else {
```

```
      // If we do not have a host, default to TCP socket on Wi
      defaultHost = fmt.Sprintf("tcp://%s:%d", opts.DefaultHTT
    }
  }
if len(flHosts) == 0 {
  defaultHost := os.Getenv("DOCKER_HOST")
  if defaultHost == "" || *flDaemon {
    if runtime.GOOS != "windows" {
      // If we do not have a host, default to unix socket
      defaultHost = fmt.Sprintf("unix://%s", opts.DefaultUnixSo
    } else {
      // If we do not have a host, default to TCP socket on Wi
      defaultHost = fmt.Sprintf("tcp://%s:%d", opts.DefaultHTT
    }

  }
  defaultHost, err := opts.ValidateHost(defaultHost)
  if err != nil {
    if *flDaemon {
      logrus.Fatal(err)
    } else {
      fmt.Fprint(os.Stderr, err)
    }
    os.Exit(1)
  }
  flHosts = append(flHosts, defaultHost)
}
```

```
setDefaultConfFlag(flTrustKey, defaultTrustKeyFile)

if *flDaemon {
  if *flHelp {
    flag.Usage()
    return
  }
  mainDaemon()
  return
}

if len(flHosts) > 1 {
  fmt.Fprintf(os.Stderr, "Please specify only one -H")
  os.Exit(0)
}
protoAddrParts := strings.SplitN(flHosts[0], "://", 2)
var (
  cli       *client.DockerCli
  tlsConfig tls.Config
)
tlsConfig.InsecureSkipVerify = true

// Regardless of whether the user sets it to true or false, if
// specify --tlsverify at all then we need to turn on tls
if flag.IsSet("-tlsverify") {
  *flTls = true
```

```
  }

  // If we should verify the server, we need to load a trusted ca
  if *flTlsVerify {
    certPool := x509.NewCertPool()
    file, err := ioutil.ReadFile(*flCa)
    if err != nil {
      fmt.Fprintf(os.Stderr, "Couldn't read ca cert %s: %s\n", *f
      os.Exit(1)
    }
    certPool.AppendCertsFromPEM(file)
    tlsConfig.RootCAs = certPool
    tlsConfig.InsecureSkipVerify = false
  }

  // If tls is enabled, try to load and send client certificates
  if *flTls || *flTlsVerify {
    _, errCert := os.Stat(*flCert)
    _, errKey := os.Stat(*flKey)
    if errCert == nil && errKey == nil {
      *flTls = true
      cert, err := tls.LoadX509KeyPair(*flCert, *flKey)
      if err != nil {
        fmt.Fprintf(os.Stderr, "Couldn't load X509 key pair: %q.
err)
        os.Exit(1)
      }
```

```
      tlsConfig.Certificates = []tls.Certificate{cert}
    }
    // Avoid fallback to SSL protocols < TLS1.0
    tlsConfig.MinVersion = tls.VersionTLS10

  }

  if *flTls || *flTlsVerify {
    cli = client.NewDockerCli(stdin, stdout, stderr, *flTrustKey,
&tlsConfig)
  } else {
    cli = client.NewDockerCli(stdin, stdout, stderr, *flTrustKey,
nil)
  }

  if err := cli.Cmd(flag.Args()...); err != nil {
    if sterr, ok := err.(client.StatusError); ok {
      if sterr.Status != "" {
        fmt.Fprintln(cli.Err(), sterr.Status)
        os.Exit(1)
      }
      os.Exit(sterr.StatusCode)
    }
    fmt.Fprintln(cli.Err(), err)
    os.Exit(1)
  }
}
```

当Docker作为客户端时，其主要工作就是通过反射找到命令的入口函数，然后执行函数。因此每个命令都可以做相互隔离、低耦合。下面看看如果Docker作为服务端时是如何执行的。

目前我们已知Docker作为服务端时，会调用mainDaemon来进行服务初始化，因此下面我们探究一下mainDaemon函数的内部。

在mainDaemon函数当中，其主要完成了两件事情：初始化Rest API路由表，通过这个路由表初始化一个API Server并且运行这个服务，其流程图如图6-3所示。

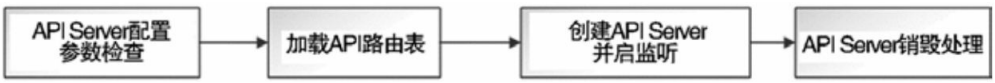


图6-3 流程图

在上述四个流程中，加载API路由表无疑是重点。因为Docker所有的命令，包括Cli和remoteAPI，都是由客户端发送rest API命令到Daemon。然后Daemon通过API路由表匹配到请求相对应的处理函数，再去执行处理函数。因此，如果需要在Docker基础上进行定制开发，那么配置API路由表就是开发的第一步。

我们看一下Docker Daemon是如何加载API路由表的，代码如下：

```
serverConfig := &apiserver.ServerConfig{
        Logging:     true,
        EnableCors:  daemonCfg.EnableCors,
        CorsHeaders: daemonCfg.CorsHeaders,
```

```
        Version:     dockerversion.VERSION,
        SocketGroup: daemonCfg.SocketGroup,
        Tls:         *flTls,
        TlsVerify:   *flTlsVerify,
        TlsCa:       *flCa,
        TlsCert:     *flCert,
        TlsKey:      *flKey,
    }

    api := apiserver.New(serverConfig)
```

这里的serverConfig，其实就是当前Docker Daemon的基础配置信息，包括是否启动log功能、当前内核版本号，以及是否启用TLS安全证书等。

当代码执行到apiserver.New时就会完成加载路由表功能。这里的New并不是Java或者C++中创建新对象时执行的构造函数，而是apiserver结构体的一个普通函数，如下：

```
func New(cfg *ServerConfig) *Server {
    srv := &Server{
        cfg:   cfg,
        start: make(chan struct{}),
    }
    r := createRouter(srv)
    srv.router = r
```

```
    return srv
}
```

apiserver通过New代码使用上面传入的serverConfig，构造了一个完整的Daemon　Server实例。这个实例不仅包括服务配置信息、API路由表，同时还包括一个channel变量（用于多个协程之间的通信）。

而其中的createRoute就是创建API路由表的函数，代码如下：

```
func createRouter(s *Server) *mux.Router {
  r := mux.NewRouter()
  if os.Getenv("DEBUG") != "" {
    ProfilerSetup(r, "/debug/")
  }
  m := map[string]map[string]HttpApiFunc{
    "GET": {
      "/_ping":                          s.ping,
      "/events":                         s.getEvents,
      "/info":                           s.getInfo,
      "/version":                        s.getVersion,
      "/images/json":                    s.getImagesJSON,
      "/images/search":                  s.getImagesSearch,
      "/images/get":                     s.getImagesGet,
      "/images/{name:.*}/get":           s.getImagesGet,
      "/images/{name:.*}/history":       s.getImagesHistory,
      "/images/{name:.*}/json":          s.getImagesByName,
```

```
      "/containers/ps":                   s.getContainersJSON,
      "/containers/json":                 s.getContainersJSON,
      "/containers/{name:.*}/export":     s.getContainersExport,
      "/containers/{name:.*}/changes":    s.getContainersChanges,
      "/containers/{name:.*}/json":       s.getContainersByName,
      "/containers/{name:.*}/top":        s.getContainersTop,
      "/containers/{name:.*}/logs":       s.getContainersLogs,
      "/containers/{name:.*}/stats":      s.getContainersStats,
      "/containers/{name:.*}/attach/ws":  s.wsContainersAttach,
      "/exec/{id:.*}/json":               s.getExecByID,
    },
    "POST": {
      "/auth":                         s.postAuth,
      "/commit":                       s.postCommit,
      "/build":                        s.postBuild,
      "/images/create":                s.postImagesCreate,
      "/images/load":                  s.postImagesLoad,
      "/images/{name:.*}/push":        s.postImagesPush,
      "/images/{name:.*}/tag":         s.postImagesTag,
      "/containers/create":            s.postContainersCreate,
      "/containers/{name:.*}/kill":    s.postContainersKill,
      "/containers/{name:.*}/pause":   s.postContainersPause,
      "/containers/{name:.*}/unpause": s.postContainersUnpause,
      "/containers/{name:.*}/restart": s.postContainersRestart,
      "/containers/{name:.*}/start":   s.postContainersStart,
      "/containers/{name:.*}/stop":    s.postContainersStop,
      "/containers/{name:.*}/wait":    s.postContainersWait,
```

```
    "/containers/{name:.*}/resize":   s.postContainersResize,
    "/containers/{name:.*}/attach":   s.postContainersAttach,
    "/containers/{name:.*}/copy":     s.postContainersCopy,
    "/containers/{name:.*}/exec":     s.postContainerExecCreate
    "/exec/{name:.*}/start":          s.postContainerExecStart,
    "/exec/{name:.*}/resize":         s.postContainerExecResize
    "/containers/{name:.*}/rename":   s.postContainerRename,
  },
  "DELETE": {
    "/containers/{name:.*}": s.deleteContainers,
    "/images/{name:.*}":     s.deleteImages,
  },
  "OPTIONS": {
    "": s.optionsHandler,
  },
}

// If "api-cors-header" is not given, but "api-enable-cors" is
// otherwise, all head values will be passed to HTTP handler
corsHeaders := s.cfg.CorsHeaders
if corsHeaders == "" && s.cfg.EnableCors {
  corsHeaders = "*"
}

for method, routes := range m {
  for route, fct := range routes {
    logrus.Debugf("Registering %s, %s", method, route)
```

```
      // NOTE: scope issue, make sure the variables are local and
      localRoute := route
      localFct := fct
      localMethod := method

      tools.Logger("In the server.go createRouter")
      // build the handler function
      f  :=  makeHttpHandler(s.cfg.Logging,  localMethod,  localF
version.Version(s.cfg.Version))

      // add the new route
      if localRoute == "" {
        r.Methods(localMethod).HandlerFunc(f)
      } else {
        r.Path("/v{version:[0-9.]+}" + localRoute).Methods(localM
        r.Path(localRoute).Methods(localMethod).HandlerFunc(f)
      }
    }
  }

  return r
}
```

看到这里，总算看到了一些熟悉的面孔。以前在Remote API中介绍的那些API，在这里都可以找到处理原型。例如，GET/version这条API对应的处理函数是s.getVersion函数。当Daemon接收到/version的GET请

求时，就会执行下面的getVersion代码：

```
func (s *Server) getVersion(version version.Version, w http.Respo
map[string]string) error {
    v := &types.Version{
        Version:    dockerversion.VERSION,
        ApiVersion: api.Version,
        GitCommit:  dockerversion.GITCOMMIT,
        GoVersion:  runtime.Version(),
        Os:         runtime.GOOS,
        Arch:       runtime.GOARCH,
    }

    if version.GreaterThanOrEqualTo("1.19") {
        v.Experimental = utils.ExperimentalBuild()
    }

    if kernelVersion, err := kernel.GetKernelVersion(); err == ni
        v.KernelVersion = kernelVersion.String()
    }

    return writeJSON(w, http.StatusOK, v)
}
```

执行完毕之后，Docker Daemon的API版本号就会返回给客户端。

```
serveAPIWait := make(chan error)
  go func() {
    tools.Logger(flHosts)
    if err := api.ServeApi(flHosts); err != nil {
      logrus.Errorf("ServeAPI error: %v", err)
      serveAPIWait <- err
      return
    }
    serveAPIWait <- nil
  }()

  if err := migrateKey(); err != nil {
    logrus.Fatal(err)
  }
  daemonCfg.TrustKeyPath = *flTrustKey

  registryService := registry.NewService(registryCfg)
  d, err := daemon.NewDaemon(daemonCfg, registryService)
  if err != nil {
    if pfile != nil {
      if err := pfile.Remove(); err != nil {
        logrus.Error(err)
      }
    }
    logrus.Fatalf("Error starting daemon: %v", err)
  }
```

```
logrus.Info("Daemon has completed initialization")

logrus.WithFields(logrus.Fields{
  "version":     dockerversion.VERSION,
  "commit":      dockerversion.GITCOMMIT,
  "execdriver":  d.ExecutionDriver().Name(),
  "graphdriver": d.GraphDriver().String(),
}).Info("Docker daemon")

signal.Trap(func() {
  api.Close()
  <-serveAPIWait
  shutdownDaemon(d, 15)
  if pfile != nil {
    if err := pfile.Remove(); err != nil {
      logrus.Error(err)
    }
  }
})
api.AcceptConnections(d)
errAPI := <-serveAPIWait
shutdownDaemon(d, 15)
if errAPI != nil {
  if pfile != nil {
    if err := pfile.Remove(); err != nil {
      logrus.Error(err)
```

```
        }
    }
    logrus.Fatalf("Shutting down due to ServeAPI error: %v", errA
}
```

当apiserver加载完API路由表之后，后面的事情就相对简单了。启动apiserver，并且当接收到停止信号（通过apiserver的channel来传送信号）后执行资源回收工作，代码如下：

至此Docker Daemon服务模式初始化完毕，可以正式处理来自于命令行或者远程的处理请求。

因为Daemon是Docker处理请求最为核心的部件，所以我们详细讲解了一下Daemon的处理流程。从Docker处理流程中来说，Daemon就是其大门，所有的请求都必须经过Daemon的筛选和分发才能得到有效的处理，如图6-4所示。

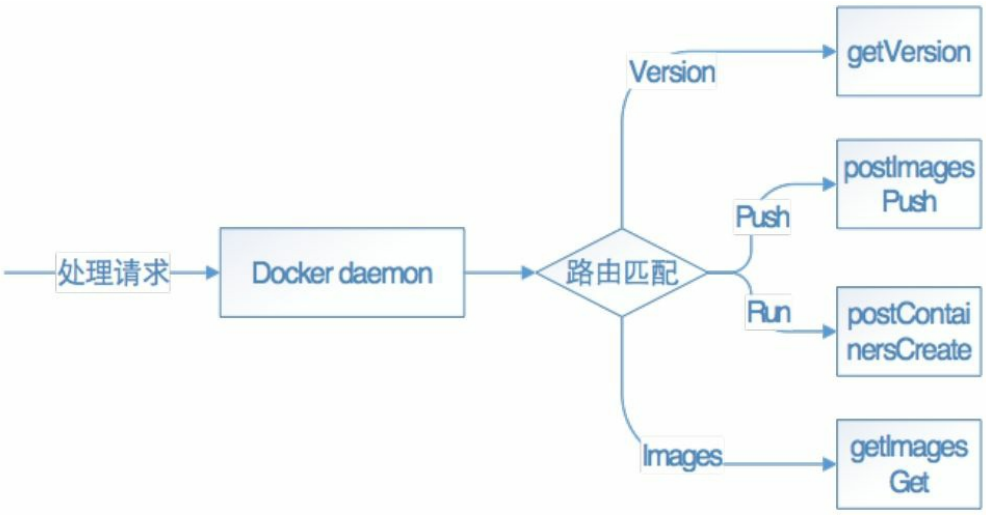


图6-4 Docker Daemon的处理流程

从下节开始，我们会依次讲解Cli中各个命令是如何执行的。

6.1 Docker操作命令

1. attach命令

attach是挂载命令，通过这个命令，我们可以挂载到另外一个正在运行的容器之中。

当执行attach命令时，Docker Daemon会将命令行的STDOUT、STDERR和STDIN（可选）与容器的STDOUT、STDERR和STDIN相关联。这样从用户的角度来看，就好像是登录到了容器内部。但其实用户还在容器外部，只不过用户每次执行的命令都会实时输入到容器的STDIN中，所以看起来是进入了内部。下面我们就梳理一下attach的执行流程，看看它是如何实现I/O流绑定的。

attach的客户端代码简洁、层次分明、结构清晰，例如：

```
func (cli *DockerCli) CmdAttach(args ...string) error {
  var (
    cmd     = cli.Subcmd("attach", "CONTAINER", "Attach to a runn
    noStdin = cmd.Bool([]string{"#nostdin", "-no-stdin"}, false,
    proxy   = cmd.Bool([]string{"#sig-proxy", "-sig-proxy"}, true
process")
  )
```

```
cmd.Require(flag.Exact, 1)

  cmd.ParseFlags(args, true)
  name := cmd.Arg(0)

  stream, _, err := cli.call("GET", "/containers/"+name+"/json",
  if err != nil {
    return err
  }

  var c types.ContainerJSON
  if err := json.NewDecoder(stream).Decode(&c); err != nil {
    return err
  }

  if !c.State.Running {
    return fmt.Errorf("You cannot attach to a stopped container,
  }

  if err := cli.CheckTtyInput(!*noStdin, c.Config.Tty); err != ni
    return err
  }

  if c.Config.Tty && cli.isTerminalOut {
    if err := cli.monitorTtySize(cmd.Arg(0), false); err != nil
      logrus.Debugf("Error monitoring TTY size: %s", err)
    }
```

```
  }

  var in io.ReadCloser

  v := url.Values{}
  v.Set("stream", "1")
  if !*noStdin && c.Config.OpenStdin {
    v.Set("stdin", "1")
    in = cli.in
  }

  v.Set("stdout", "1")
  v.Set("stderr", "1")
  if *proxy && !c.Config.Tty {

    sigc := cli.forwardAllSignals(cmd.Arg(0))
    defer signal.StopCatch(sigc)
  }
  tools.Logger("cmd.Arg[0]=", cmd.Arg(0))
  if err := cli.hijack("POST", "/containers/"+cmd.Arg(0)+"/attach
cli.out, cli.err, nil, nil); err != nil {
    return err
  }
_, status, err := getExitCode(cli, cmd.Arg(0))
  if err != nil {
    return err
  }
```

```
  if status != 0 {
    return StatusError{StatusCode: status}
  }
  return nil
}
```

attach客户端代码共完成了三件事情：验证输入的容器名称是否存在，验证输入的容器是否满足挂载要求，向Daemon发送挂载请求。

在上段代码中，有一段代码需要我们注意：

```
    v.Set("stream", "1")
    if !*noStdin && c.Config.OpenStdin {
        v.Set("stdin", "1")
        in = cli.in
    }

    v.Set("stdout", "1")
    v.Set("stderr", "1")
```

从这段代码中我们看到，发往Daemon的URL参数中一定会存在stream、stdout和stderr三个参数。这三个参数stream=true表示需要将两者I/O流关联起来，stdout和stderr都为true，表示这两个I/O流会进行关联。而stdin则需要根据用户的输入来决定是否关联。

当这些参数都准备好之后，Docker就会作为客户端向Daemon发往

一个POST方法的API：

```
if err := cli.hijack("POST", "/containers/"+cmd.Arg(0)+"/attach?"
cli.out, cli.err, nil, nil); err != nil {
        return err
    }
```

剩下的就是服务端代码的事情了。我们看一下服务端代码的情况：

```
"/containers/{name:.*}/attach":  s.postContainersAttach
```

根据这个路由，我们可以得知attach对应的服务端处理函数为：postContainersAttach。这个函数只做了一件事情，就是关联指定的I/O流。函数代码很简单，具体如下：

```
func (s *Server) postContainersAttach(version version.Version, w
vars map[string]string) error {
    if err := parseForm(r); err != nil {
        return err
    }
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    }
```

```
    cont, err := s.daemon.Get(vars["name"])
    if err != nil {
        return err
    }
    inStream, outStream, err := hijackServer(w)
    if err != nil {
        return err
    }
    defer closeStreams(inStream, outStream)

    if _, ok := r.Header["Upgrade"]; ok {
        fmt.Fprintf(outStream, "HTTP/1.1 101 UPGRADED\r\nContent-
stream\r\nConnection: Upgrade\r\nUpgrade: tcp\r\n\r\n")
    } else {
        fmt.Fprintf(outStream, "HTTP/1.1 200 OK\r\nContent-Type:
r\n\r\n")
    }

    attachWithLogsConfig := &daemon.ContainerAttachWithLogsConfig
        InStream:  inStream,
        OutStream: outStream,
        UseStdin:  boolValue(r, "stdin"),
        UseStdout: boolValue(r, "stdout"),
        UseStderr: boolValue(r, "stderr"),
        Logs:      boolValue(r, "logs"),
        Stream:    boolValue(r, "stream"),
        Multiplex: version.GreaterThanOrEqualTo("1.6"),
```

```
    }
    if err := s.daemon.ContainerAttachWithLogs(cont, attachWithLo
        fmt.Fprintf(outStream, "Error attaching: %s\n", err)
    }

    return nil
}
```

这段代码貌似没有什么特别之处，而玄机就在ContainerAttachWithLogs函数中。在ContainerAttachWithLogs函数中，将指定容器的I/O流与当前环境的I/O流做了映射关系，代码如下：

```
func (daemon *Daemon) ContainerAttachWithLogs(container *Containe
error {
    var errStream io.Writer

    if !container.Config.Tty {
        errStream = stdcopy.NewStdWriter(c.OutStream, stdcopy.Std
        c.OutStream = stdcopy.NewStdWriter(c.OutStream, stdcopy.S
    } else {
        errStream = c.OutStream
    }

    var stdin io.ReadCloser
    var stdout, stderr io.Writer
```

```
    if c.UseStdin {
        stdin = c.InStream
    }
    if c.UseStdout {
        stdout = c.OutStream
    }
    if c.UseStderr {
        stderr = errStream
    }

    return container.AttachWithLogs(stdin, stdout, stderr, c.Logs
}
```

这段代码里面，声明了三个I/O流：

```
var stdin io.ReadCloser
var stdout, stderr io.Writer
```

然后，这三个I/O流与容器之中的I/O流进行关联（这里的关联仅仅是将文件句柄赋给对方，后面有真正的关联操作）：

```
if c.UseStdin {
        stdin = c.InStream
```

```
    }
    if c.UseStdout {
        stdout = c.OutStream
    }
    if c.UseStderr {
        stderr = errStream
    }
```

而后将这些I/O流传入container.AttachWithLogs函数中进行后续的关联操作：

```
func (c *Container) AttachWithLogs(stdin io.ReadCloser, stdout, s
error {

    if logs {
        logDriver, err := c.getLogger()
        if err != nil {
            logrus.Errorf("Error obtaining the logger %v", err)
            return err
        }
        if _, ok := logDriver.(logger.Reader); !ok {
            logrus.Errorf("cannot read logs for [%s] driver", log
        } else {
            if cLog, err := logDriver.(logger.Reader).ReadLog();
                logrus.Errorf("Error reading logs %v", err)
```

```
            } else {
                dec := json.NewDecoder(cLog)
                for {
                    l := &jsonlog.JSONLog{}

                    if err := dec.Decode(l); err == io.EOF {
                        break
                    } else if err != nil {
                        logrus.Errorf("Error streaming logs: %s",
                        break
                    }
                    if l.Stream == "stdout" && stdout != nil {
                        io.WriteString(stdout, l.Log)
                    }
                    if l.Stream == "stderr" && stderr != nil {
                        io.WriteString(stderr, l.Log)
                    }
                }
            }
        }
    }

    c.LogEvent("attach")
    if stream {
        var stdinPipe io.ReadCloser
        if stdin != nil {
            r, w := io.Pipe()
```

```
            go func() {
                defer w.Close()
                defer logrus.Debugf("Closing buffered stdin pipe"
                io.Copy(w, stdin)
            }()
            stdinPipe = r
        }
        <-c.Attach(stdinPipe, stdout, stderr)
        if c.Config.StdinOnce && !c.Config.Tty {
            c.WaitStop(-1 * time.Second)
        }
    }
    return nil
}
```

函数中只有两个逻辑判断，即需要log时如何操作，以及需要stream输出时如何操作。Log比较简单，读者可以自行分析。下面我们分析一下stream=true的情况。

首先代码在容器中创建了一个管道（io.Pipe()），然后将这个管道的write端与当前环境的STNID绑定到一起（io.copy），这样，在STDIN中写入的数据就可以传递到容器内部（这时还缺少容器里面的I/O流，后面会讲到）。然后开始处理stdout和stderr，读者可能要问，为什么这里不像处理stdin那样，直接调用io.copy呀。

这是因为stdin是可选参数，而stdout和stderr一定会绑定，所以处理流程不同。另外一个原因是attach需要阻塞，不能绑定之后什么都不处

理就直接退出，所以需要一个阻塞函数，而stdin不能成为阻塞函数。如果stdin变为阻塞函数，那么在命令行就什么都无法输入了，所以只能将stdout和stderr变为阻塞函数。命令行的stdout和stderr一直等待容器的输出，如果有输出，数据就会实时输出；如果没有输出，那么就一直阻塞在那里等待数据到达。

搞清楚这个问题之后，我们继续看后面是如何处理stdout和stderr这两个I/O流的，代码如下：

```
func (c *Container) Attach(stdin io.ReadCloser, stdout io.Writer,
    return attach(&c.StreamConfig, c.Config.OpenStdin, c.Config.S
stdout, stderr)
}
```

c.Attach函数什么都没做，而是直接调用了另外一个内部的attach函数：

```
func attach(streamConfig *StreamConfig, openStdin, stdinOnce, tty
io.Writer, stderr io.Writer) chan error {
  var (
    cStdout, cStderr io.ReadCloser
    cStdin           io.WriteCloser
    wg               sync.WaitGroup
    errors           = make(chan error, 3)
  )
```

```
  if stdin != nil && openStdin {
    cStdin = streamConfig.StdinPipe()
    wg.Add(1)
  }

  if stdout != nil {
    cStdout = streamConfig.StdoutPipe()
    wg.Add(1)
  }

  if stderr != nil {
    cStderr = streamConfig.StderrPipe()
    wg.Add(1)
  }

  // Connect stdin of container to the http conn.
go func() {
    if stdin == nil || !openStdin {
      return
    }
    logrus.Debugf("attach: stdin: begin")
    defer func() {
      if stdinOnce && !tty {
        cStdin.Close()
      } else {
        // No matter what, when stdin is closed (io.Copy unblock)
```

```
        if cStdout != nil {
          cStdout.Close()
        }
        if cStderr != nil {
          cStderr.Close()
        }
      }
      wg.Done()
      logrus.Debugf("attach: stdin: end")
    }()

    var err error
    if tty {
      _, err = copyEscapable(cStdin, stdin)
    } else {
      _, err = io.Copy(cStdin, stdin)

    }
    if err == io.ErrClosedPipe {
      err = nil
    }
    if err != nil {
      logrus.Errorf("attach: stdin: %s", err)
      errors <- err
      return
    }
  }()
```

```
attachStream := func(name string, stream io.Writer, streamPipe i
   if stream == nil {
     return
   }
   defer func() {
     // Make sure stdin gets closed
     if stdin != nil {
       stdin.Close()
     }
     streamPipe.Close()
     wg.Done()
     logrus.Debugf("attach: %s: end", name)
   }()
   logrus.Debugf("attach: %s: begin", name)
   _, err := io.Copy(stream, streamPipe)
   if err == io.ErrClosedPipe {
     err = nil
   }
   if err != nil {
     logrus.Errorf("attach: %s: %v", name, err)
     errors <- err
   }
 }

 go attachStream("stdout", stdout, cStdout)
 go attachStream("stderr", stderr, cStderr)
```

```
  return promise.Go(func() error {
    wg.Wait()
    close(errors)
    for err := range errors {
      if err != nil {
        return err
      }
    }
    return nil
  })
}
```

这段代码完成了attach最后所有的功能。首先将刚才管道流的Reader端与容器内操作系统的Stdin进行对接：

```
io.Copy(cStdin, stdin)
```

再将命令行的stdout和stderr分别与容器的stdout和stderr进行对接，其中，对stdout和stderr的操作则借助于Golang的函数式编程特性，代码如下：

```
attachStream := func(name string, stream io.Writer, streamPipe io
        if stream == nil {
            return
```

```
        }
        defer func() {
            // Make sure stdin gets closed
            if stdin != nil {
                stdin.Close()
            }
            streamPipe.Close()
            wg.Done()
            logrus.Debugf("attach: %s: end", name)
        }()

        logrus.Debugf("attach: %s: begin", name)
        _, err := io.Copy(stream, streamPipe)
        if err == io.ErrClosedPipe {
            err = nil
        }
        if err != nil {
            logrus.Errorf("attach: %s: %v", name, err)
            errors <- err
        }
    }
```

这里返回了attachStream函数，然后再将stdout和stderr作为参数传入函数中：

```
go attachStream("stdout", stdout, cStdout)
go attachStream("stderr", stderr, cStderr)
```

因为io.copy在内部处理时是一个阻塞函数，所以这里借助Golang强大的并发能力可以同时处理stdout和stderr阻塞请求。

attach就通过上述代码逻辑通过I/O流共享的方式，达到容器挂载的目的。为了方便大家理解这段关于I/O流的处理，下面给出如图6-5所示的示意图。

通过此示意图，也可以看出，attach这个命令其实就是完成了一件I/O流相互关联的事情。但是在这个命令过程中会遇到信号处理、阻塞等待以及判断是否需要处理STDIN这三个问题。因此在进行I/O流关联之前，需要先做一些额外的操作处理。

2. build命令

build命令是用Dockerfile中的指令来构建Docker镜像的命令，因此，build代码的主要逻辑都是围绕着Dockerfile文件、所涉及文件和镜像参数这三个要素进行操作的。

build命令执行时，其主要逻辑是：

①获取Dockerfile文件。

②加载Dockerfile文件所在目录下的所有文件。

③解析Dockerfile文件指令。

④当执行文件内指令时，判断是否存在有效缓存镜像。

⑤执行指令。

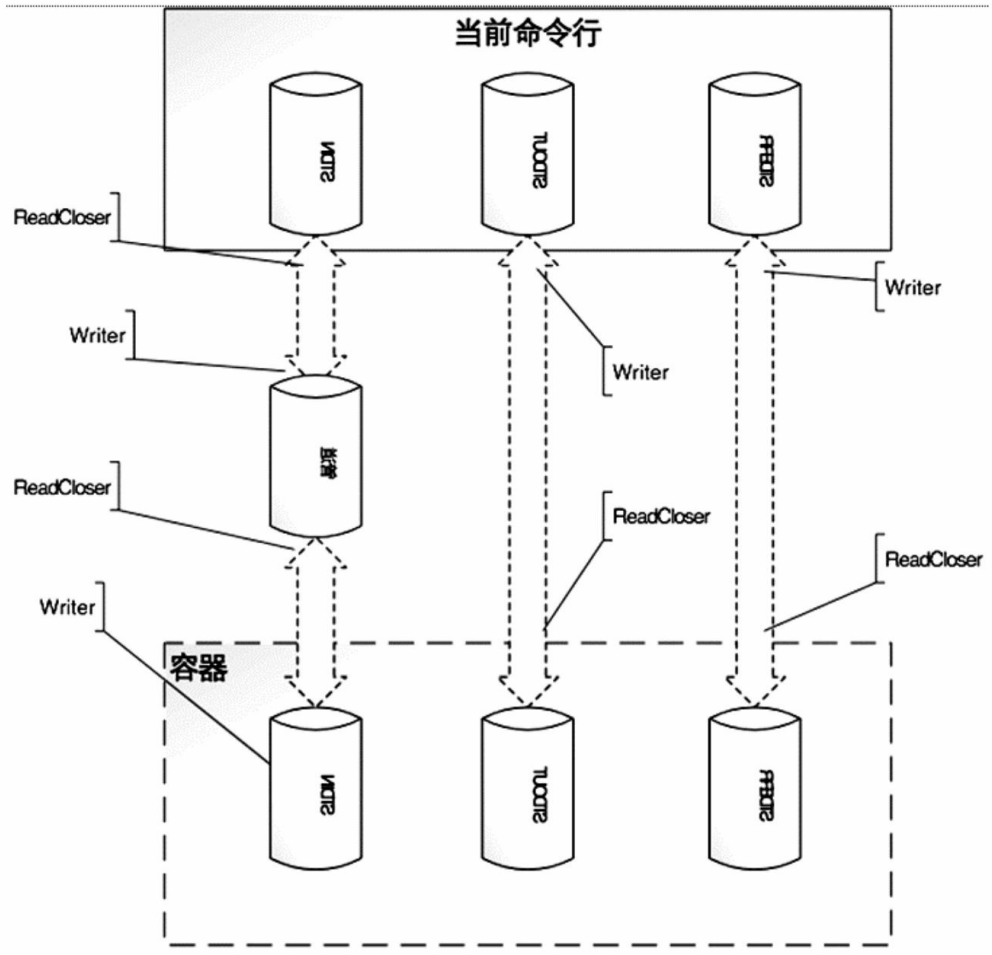


图6-5　I/O流处理示意图

因此在剖析build代码时，我们也就按照这个顺序开始。

上述步骤的第一步和第二步是在build客户端执行的。首先获取Dockerfile文件，获取的方式有三种：通过标准输入传入（通过参数“-”来实现）、通过远程方式获取文件（通过git方式获取）和通过指定路径方式获取。

在代码中也会分出三种处理方式，具体如下：

```
if cmd.Arg(0) == "-" {
    // As a special case, 'docker build -' will build from either
    // contents of stdin as a Dockerfile, or a tar-ed context fro
    buf := bufio.NewReader(cli.in)
    magic, err := buf.Peek(tarHeaderSize)
    if err != nil && err != io.EOF {
      return fmt.Errorf("failed to peek context header from STDIN
    }
    if !archive.IsArchive(magic) {
      dockerfile, err := ioutil.ReadAll(buf)
      if err != nil {
        return fmt.Errorf("failed to read Dockerfile from STDIN:
      }

      // -f option has no meaning when we're reading it from stdi
      // so just use our default Dockerfile name
      *dockerfileName = api.DefaultDockerfileName
      context, err = archive.Generate(*dockerfileName, string(doc
    } else {
      context = ioutil.NopCloser(buf)
    }
  } else if urlutil.IsURL(cmd.Arg(0)) && (!urlutil.IsGitURL(cmd.A
    isRemote = true
  } else {
```

```
    root := cmd.Arg(0)
    if urlutil.IsGitURL(root) {
      root, err = utils.GitClone(root)
      if err != nil {
        return err
      }
      defer os.RemoveAll(root)
    }
    if _, err := os.Stat(root); err != nil {
      return err
    }

    absRoot, err := filepath.Abs(root)
    if err != nil {
      return err
    }

    filename := *dockerfileName // path to Dockerfile

    if *dockerfileName == "" {
      // No -f/--file was specified so use the default
      *dockerfileName = api.DefaultDockerfileName
      filename = filepath.Join(absRoot, *dockerfileName)

      // Just to be nice ;-) look for 'dockerfile' too but only
      // use it if we found it, otherwise ignore this check
      if _, err = os.Lstat(filename); os.IsNotExist(err) {
```

```
        tmpFN := path.Join(absRoot, strings.ToLower(*dockerfileNa
        if _, err = os.Lstat(tmpFN); err == nil {
          *dockerfileName = strings.ToLower(*dockerfileName)
          filename = tmpFN
        }
      }
    }

    origDockerfile := *dockerfileName // used for error msg
    if filename, err = filepath.Abs(filename); err != nil {
      return err
    }

    // Verify that 'filename' is within the build context
    filename, err = symlink.FollowSymlinkInScope(filename, absRoo
    if err != nil {
      return fmt.Errorf("The Dockerfile (%s) must be within the b
root)
    }

    // Now reset the dockerfileName to be relative to the build c
    *dockerfileName, err = filepath.Rel(absRoot, filename)
    if err != nil {
      return err
    }
    // And canonicalize dockerfile name to a platform-independent
    *dockerfileName, err = archive.CanonicalTarNameForPath(*docke
```

```
    if err != nil {
      return fmt.Errorf("Cannot canonicalize dockerfile path %s:
    }

    if _, err = os.Lstat(filename); os.IsNotExist(err) {
      return fmt.Errorf("Cannot locate Dockerfile: %s", origDocke
    }
    var includes = []string{"."}

    excludes, err := utils.ReadDockerIgnore(path.Join(root, ".doc
    if err != nil {
      return err
    }
    keepThem1, _ := fileutils.Matches(".dockerignore", excludes)
    keepThem2, _ := fileutils.Matches(*dockerfileName, excludes)
    if keepThem1 || keepThem2 {
      includes = append(includes, ".dockerignore", *dockerfileNar
    }

   if err := utils.ValidateContextDirectory(root, excludes); err
      return fmt.Errorf("Error checking context is accessible: '%
again.", err)
    }
    options := &archive.TarOptions{
      Compression:     archive.Uncompressed,
      ExcludePatterns: excludes,
      IncludeFiles:    includes,
```

```
    }
    context, err = archive.TarWithOptions(root, options)
    if err != nil {
      return err
    }
  }
```

可以看到，build代码首先判断第一个参数是否为“-”。如果是“-”，就直接从标准输入中读取所有的数据内容作为Dockerfile内容。当第一个参数不是“-”时，就会判断是否为远程获取方式，判断规则如下：

```
if urlutil.IsURL(cmd.Arg(0)) && (!urlutil.IsGitURL(cmd.Arg(0)) |
        isRemote = true
    }
```

如果是URL或者git任意一种，则视为远程方式。我们看一下具体是如何比较的，因为搞明白比对规则更有利于使用build命令，代码如下：

```
var (
    validPrefixes = map[string][]string{
        "url":       {"http://", "https://"},
        "git":       {"git://", "github.com/", "git@"},
        "transport": {"tcp://", "udp://", "unix://"},
    }
```

```
    urlPathWithFragmentSuffix = regexp.MustCompile(".git(?:#.+)?$
)

// IsURL returns true if the provided str is an HTTP(S) URL.
func IsURL(str string) bool {
    return checkURL(str, "url")
}

// IsGitURL returns true if the provided str is a git repository
func IsGitURL(str string) bool {
    if IsURL(str) && urlPathWithFragmentSuffix.MatchString(str) {
        return true
    }
    return checkURL(str, "git")
}

// IsGitTransport returns true if the provided str is a git trans
// the prefix of the string for known protocols used in git.
func IsGitTransport(str string) bool {
    return IsURL(str) || strings.HasPrefix(str, "git://") || str
}

// IsTransportURL returns true if the provided str is a transport
func IsTransportURL(str string) bool {
    return checkURL(str, "transport")
}
func checkURL(str, kind string) bool {
```

```
    for _, prefix := range validP
        if strings.HasPrefix(str,
            return true
        }
    }
    return false
}
```

原来是通过正则表达式来进行匹
http://或https://字符，则认为是网络连
git@，则认为是通过git方式。以后读
Dockerfile，就清楚应该怎么输入git U

这里还有一个hasGit，这个参数
断当前环境是否安装git。如果当前环
也不会执行。Docker用这个参数来处

下面继续顺着build的代码往下走
Dockerfile，也不是远程方式获取时，
定路径的方式来获取。所以后面就开

但在这里，Docker顺便把git方式

```
        root := cmd.Arg(0)
        if urlutil.IsGitURL(root)
            root, err = utils.Git
            if err != nil {
```

```
                return err
            }
            defer os.RemoveAll(root)
        }
```

Docker判断是否为git方式，如果是git方式，则把git上面的目录都取回本地。Docker为什么这么做呢？笔者理解的是，Docker认为git上面的目录是可以整个取下来的，那么与其放到服务端取下来再加载。不如在客户端取下来，视为本地文件一起加载。这样从逻辑上比较清晰，代码相互不会耦合，有利于代码维护。

我们看一下Docker是如何克隆git数据的。

```
func GitClone(remoteURL string) (string, error) {
  if !urlutil.IsGitTransport(remoteURL) {
    remoteURL = "https://" + remoteURL
  }
  root, err := ioutil.TempDir("", "docker-build-git")
  if err != nil {
    return "", err
  }

  u, err := url.Parse(remoteURL)
  if err != nil {
    return "", err
  }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  return append(args, remoteURL.String(), root)
}

func checkoutGit(fragment, root string) (string, error) {
  refAndDir := strings.SplitN(fragment, ":", 2)

  if len(refAndDir[0]) != 0 {
    if output, err := gitWithinDir(root, "checkout", refAndDir[0]
      return "", fmt.Errorf("Error trying to use git: %s (%s)", e
    }
  }

  if len(refAndDir) > 1 && len(refAndDir[1]) != 0 {
    newCtx, err := symlink.FollowSymlinkInScope(filepath.Join(roo
    if err != nil {
      return "", fmt.Errorf("Error setting git context, %q not wi
err)
    }

    fi, err := os.Stat(newCtx)
    if err != nil {
      return "", err
    }
    if !fi.IsDir() {
      return "", fmt.Errorf("Error setting git context, not a dir
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        if *dockerfileName == "" {
            // No -f/--file was specified so use the default
            *dockerfileName = api.DefaultDockerfileName
            filename = filepath.Join(absRoot, *dockerfileName)

            // Just to be nice ;-) look for 'dockerfile' too but
            // use it if we found it, otherwise ignore this check
            if _, err = os.Lstat(filename); os.IsNotExist(err) {
                tmpFN := path.Join(absRoot, strings.ToLower(*dock
                if _, err = os.Lstat(tmpFN); err == nil {
                    *dockerfileName = strings.ToLower(*dockerfile
                    filename = tmpFN
                }
            }
        }

#    DefaultDockerfileName string = "Dockerfile"
```

后续获取Dockerfile实际路径，然后根据处理平台对Dockerfile实际路径进行合法转换。这里的转换是为了以后兼容Windows平台和Darwin平台考虑的，目前没有作用。

当把这部分处理完之后，就到了文件处理的最后环节了：过滤文件。Docker通过.dockerignore来过滤文件。Docker的处理流程是，首先通过读取.dockerignore来获取需要过滤的文件列表，代码如下：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

然后单独判断是否需要过滤.dockerignore文件和Dockerfile文件。如果不需要过滤，则将这两个文件放到加载文件切片中。当这些都做完后，Docker为了保证后续文件处理不会有异常情况发生，最后还会做一次文件验证。这次的验证，主要是验证文件是否存在、文件权限是否正确和文件是否可读等。

这些步骤都完成后，Docker就开始将这些文件压缩打包，然后将压缩包发送给Daemon处理：

```
        options := &archive.TarOptions{
            Compression:     archive.Uncompressed,
            ExcludePatterns: excludes,
            IncludeFiles:    includes,
        }
        context, err = archive.TarWithOptions(root, options)
        if err != nil {
            return err
        }
```

看到这里，就明白Docker为什么建议用户尽量把Dockerfile单独放到一个目录中，并且目录中只放置所涉及的文件了吧。因为当这样做的时候，这个压缩包体积将会最小，并且目录里面文件发生变化的概率也会最小，这样就有利于Docker尽可能地复用缓存中的数据，提高构建效

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return err
  }
  memory = parsedMemory
}
var memorySwap int64
if *flMemorySwap != "" {
  if *flMemorySwap == "-1" {
    memorySwap = -1
  } else {
    parsedMemorySwap, err := units.RAMInBytes(*flMemorySwap)
    if err != nil {
      return err
    }
    memorySwap = parsedMemorySwap
  }
}
// Send the build context
v := &url.Values{}
if *tag != "" {
  repository, tag := parsers.ParseRepositoryTag(*tag)
  if err := registry.ValidateRepositoryName(repository); err !=
    return err
  }
  if len(tag) > 0 {
    if err := tags.ValidateTagName(tag); err != nil {
      return err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
v.Set("cpusetcpus", *flCPUSetCpus)
v.Set("cpusetmems", *flCPUSetMems)
v.Set("cpushares", strconv.FormatInt(*flCPUShares, 10))
v.Set("cpuquota", strconv.FormatInt(*flCpuQuota, 10))
v.Set("cpuperiod", strconv.FormatInt(*flCpuPeriod, 10))
v.Set("memory", strconv.FormatInt(memory, 10))
v.Set("memswap", strconv.FormatInt(memorySwap, 10))
v.Set("cgroupparent", *flCgroupParent)

v.Set("dockerfile", *dockerfileName)

headers := http.Header(make(map[string][]string))
buf, err := json.Marshal(cli.configFile.AuthConfigs)
if err != nil {
  return err
}
headers.Add("X-Registry-Config", base64.URLEncoding.EncodeToSt

if context != nil {
  headers.Set("Content-Type", "application/tar")
}
sopts := &streamOpts{
  rawTerminal: true,
  in:          body,
  out:         cli.out,
  headers:     headers,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    Remove          bool
    ForceRemove     bool
    Pull            bool
    Memory          int64
    MemorySwap      int64
    CpuShares       int64
    CpuPeriod       int64
    CpuQuota        int64
    CpuSetCpus      string
    CpuSetMems      string
    CgroupParent    string
    AuthConfigs     map[string]cliconfig.AuthConfig

    Stdout  io.Writer
    Context io.ReadCloser
    // When closed, the job has been cancelled.
    // Note: not all jobs implement cancellation.
    // See Job.Cancel() and Job.WaitCancelled()
    cancelled  chan struct{}
    cancelOnce sync.Once
}
```

这个结构体中主要保存了镜像构建过程中所用到的数据，例如，Dockerfile文件，缓存数据和镜像参数等这些来自于客户端所传递过来的数据。这个结构体封装好之后，就通过下面的函数开始进行真正的构建：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err != nil {
        return err
    }
    decompressedStream, err := archive.DecompressStream(context)
    if err != nil {
        tools.Logger("archive.DecompressStream:", err)
        return err
    }

    if b.context, err = tarsum.NewTarSum(decompressedStream, true
        return err
    }

    if err := chrootarchive.Untar(b.context, tmpdirPath, nil); er
        return err
    }

    b.contextPath = tmpdirPath
    return nil
}
```

可以看到，Docker将压缩包解压到了Docker自己的tmp目录中，并且将此目录的路径返回给了buildConfig（这个路径会在后面用到）。

现在目录和构建所需的文件都有了，Docker开始解析Dockerfile文件中的指令，为后续执行内部指令做准备。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            if b.ForceRemove {
                b.clearTmp()
            }
            return "", err
        }
        fmt.Fprintf(b.OutStream, " ---> %s\n", stringid.Truncatel
        if b.Remove {
            b.clearTmp()
        }
    }
```

b.dockerfile就是解析后的Dockerfile指令语法树，此刻开始循环处理Dockerfile中的每一条指令。执行指令循环的函数是一dispatch函数：

```
func (b *Builder) dispatch(stepN int, ast *parser.Node) error {
  cmd := ast.Value
  attrs := ast.Attributes
  original := ast.Original
  flags := ast.Flags
  strs := []string{}
  msg := fmt.Sprintf("Step %d : %s", stepN, strings.ToUpper(cmd))

  if len(ast.Flags) > 0 {
    msg += " " + strings.Join(ast.Flags, " ")
  }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    var str string
    str = ast.Value
    if _, ok := replaceEnvAllowed[cmd]; ok {
      var err error
      str, err = ProcessWord(ast.Value, b.Config.Env)
      if err != nil {
        return err
      }
    }
    strList[i+l] = str
    msgList[i] = ast.Value
    i++
  }

  msg += " " + strings.Join(msgList, " ")
  fmt.Fprintln(b.OutStream, msg)

  // XXX yes, we skip any cmds that are not valid; the parser sho
  // picked these out already.
  if f, ok := evaluateTable[cmd]; ok {
    b.BuilderFlags = NewBuilderFlags()
    b.BuilderFlags.Args = flags
    return f(b, strList, attrs, original)
  }

  return fmt.Errorf("Unknown instruction: %s", strings.ToUpper(cm
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    Workdir    = "workdir"
    Run        = "run"
    Cmd        = "cmd"
    Entrypoint = "entrypoint"
    Expose     = "expose"
    Volume     = "volume"
    User       = "user"
)



func init() {
    evaluateTable = map[string]func(*Builder, []string, map[strin
        command.Env:        env,
        command.Label:      label,
        command.Maintainer: maintainer,
        command.Add:        add,
        command.Copy:       dispatchCopy, // copy() is a go built
        command.From:       from,
        command.Onbuild:    onbuild,
        command.Workdir:    workdir,
        command.Run:        run,
        command.Cmd:        cmd,
        command.Entrypoint: entrypoint,
        command.Expose:     expose,
        command.Volume:     volume,
        command.User:       user,
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
if name == NoBaseImageSpecifier {
    b.image = ""
    b.noBaseImage = true
    return nil
}

image, err := b.Daemon.Repositories().LookupImage(name)
if b.Pull {
    image, err = b.pullImage(name)
    if err != nil {
        return err
    }
}
if err != nil {
    if b.Daemon.Graph().IsNotExist(err, name) {
        image, err = b.pullImage(name)
    }

    // note that the top level err will still be !nil here if
    // not the error. This approach just simplifies the logic
    if err != nil {
        return err
    }
}

return b.processImageFrom(image)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

图6-6　示意图

最后，有一类函数需要特别注意，请看下面的代码：

```
func run(b *Builder, args []string, attributes map[string]bool, c
   //中间略过部分代码
    // Ensure that we keep the container mounted until the commit
    // to avoid unmounting and then mounting directly again
    c.Mount()
    defer c.Unmount()

    err = b.run(c)
    if err != nil {
        return err
    }
    if err := b.commit(c.ID, cmd, "run"); err != nil {
        return err
    }

    return nil
}
```

在run代码中，存在Mount函数和commit函数，当这些函数执行时，就会生成一个新的文件层。所以当不确定某些Dockerfile指令是否会生成文件层时，就可以在指令代码中查看是否有上述两个函数。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    buildConfig.Remove = boolValue(r, "rm")
  }
  if boolValue(r, "pull") && version.GreaterThanOrEqualTo("1.16")
    buildConfig.Pull = true
  }
output := ioutils.NewWriteFlusher(w)
  buildConfig.Stdout = output
  buildConfig.Context = r.Body

  buildConfig.RemoteURL = r.FormValue("remote")
  buildConfig.DockerfileName = r.FormValue("dockerfile")
  buildConfig.RepoName = r.FormValue("t")
  buildConfig.SuppressOutput = boolValue(r, "q")
  buildConfig.NoCache = boolValue(r, "nocache")
  buildConfig.ForceRemove = boolValue(r, "forcerm")
  buildConfig.AuthConfigs = authConfigs
  buildConfig.MemorySwap = int64ValueOrZero(r, "memswap")
  buildConfig.Memory = int64ValueOrZero(r, "memory")
  buildConfig.CpuShares = int64ValueOrZero(r, "cpushares")
  buildConfig.CpuPeriod = int64ValueOrZero(r, "cpuperiod")
  buildConfig.CpuQuota = int64ValueOrZero(r, "cpuquota")
  buildConfig.CpuSetCpus = r.FormValue("cpusetcpus")
  buildConfig.CpuSetMems = r.FormValue("cpusetmems")
  buildConfig.CgroupParent = r.FormValue("cgroupparent")
  if closeNotifier, ok := w.(http.CloseNotifier); ok {
    finished := make(chan struct{})
    defer close(finished)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
)

repoName, tag = parsers.ParseRepositoryTag(buildConfig.RepoName
tools.Logger("[job.go] Build repoName:", repoName)
if repoName != "" {
  if err := registry.ValidateRepositoryName(repoName); err != n
    return err
  }
  if len(tag) > 0 {
    if err := tags.ValidateTagName(tag); err != nil {
      return err
    }
  }
}
if buildConfig.RemoteURL == "" {
  tools.Logger("Here RemoteURL is null")
  context = ioutil.NopCloser(buildConfig.Context)
} else if urlutil.IsGitURL(buildConfig.RemoteURL) {
  tools.Logger("Here RemoteURL is a git url")
  root, err := utils.GitClone(buildConfig.RemoteURL)
  if err != nil {
    return err
  }
  defer os.RemoveAll(root)

  c, err := archive.Tar(root, archive.Uncompressed)
  if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
sf := streamformatter.NewJSONStreamFormatter()
builder := &Builder{
  Daemon: d,
  OutStream: &streamformatter.StdoutFormater{
    Writer:          buildConfig.Stdout,
    StreamFormatter: sf,
  },
  ErrStream: &streamformatter.StderrFormater{
    Writer:          buildConfig.Stdout,
    StreamFormatter: sf,
  },
  Verbose:          !buildConfig.SuppressOutput,
  UtilizeCache:      !buildConfig.NoCache,
  Remove:           buildConfig.Remove,
  ForceRemove:       buildConfig.ForceRemove,
  Pull:             buildConfig.Pull,
  OutOld:           buildConfig.Stdout,
  StreamFormatter:   sf,
  AuthConfigs:       buildConfig.AuthConfigs,
  dockerfileName:    buildConfig.DockerfileName,
  cpuShares:         buildConfig.CpuShares,
  cpuPeriod:         buildConfig.CpuPeriod,
  cpuQuota:          buildConfig.CpuQuota,
  cpuSetCpus:        buildConfig.CpuSetCpus,
  cpuSetMems:        buildConfig.CpuSetMems,
  cgroupParent:      buildConfig.CgroupParent,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  return "", err
}
b.Config = &runconfig.Config{}

b.TmpContainers = map[string]struct{}{}

for i, n := range b.dockerfile.Children {
  select {
  case <-b.cancelled:
    logrus.Debug("Builder: build cancelled!")
    fmt.Fprintf(b.OutStream, "Build cancelled")
    return "", fmt.Errorf("Build cancelled")
  default:
    // Not cancelled yet, keep going...
  }
  if err := b.dispatch(i, n); err != nil {
    if b.ForceRemove {
      b.clearTmp()
    }
    return "", err
  }
  fmt.Fprintf(b.OutStream, " ---> %s\n", stringid.TruncateID(b.
  if b.Remove {
    b.clearTmp()
  }
}
if b.image == "" {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    cmd.Var(&flChanges, []string{"c", "-change"}, "Apply Dockerf
image")
```

下面代码会通过判断各个变量是否为空，组织post命令参数。当这些参数都判断完之后，commit客户端代码就会向Daemon发送commit请求，具体如下：

```
    stream, _, err := cli.call("POST", "/commit?"+v.Encode(), co
    if err != nil {
        return err
    }
```

所以commit的重头戏在服务端，下面是服务端中commit处理函数与Rest请求的对应关系：

```
"/commit":                          s.postCommit,
```

接下来我们深入到postCommit函数一探究竟。

在postCommit函数中，Docker首先构建了一个commitConfig结构体。这个结构体是为后面处理具体文件层准备参数，处理代码如下：

```
    containerCommitConfig := &daemon.ContainerCommitConfig{
        Pause:   pause,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    imgID, err := builder.Commit(s.daemon, cont, containerCommitC
    if err != nil {
        return err
    }
```

Docker运行到这行代码时只完成了解析参数的工作，还没有真正开始创建镜像，所以在commit函数之中，Docker首先获取了源容器的配置参数，代码如下：

```
container, err := d.Get(name)
```

这行代码非常简练，但却通过容器名获取到了容器配置参数。本着打破砂锅问到底的态度，我们把这个函数也剖开，看看究竟：

```
func (daemon *Daemon) Get(prefixOrName string) (*Container, error
    if containerByID := daemon.containers.Get(prefixOrName); cont
        // prefix is an exact match to a full container ID
        return containerByID, nil
    }
    // GetByName will match only an exact name provided; we ignor
    if containerByName, _ := daemon.GetByName(prefixOrName); cont
        // prefix is an exact match to a full container Name
        return containerByName, nil
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

那c.Lock和c.Unlock是用来做什么的呢？顾名思义，这两个是添加同步锁的原子函数。因为Docker Daemon是异步执行命令，所以当获取容器信息时，需要保证当前没有其他事务会改变容器信息。因此通过添加原子同步锁的方式来保证数据同步。

Lock和Unlock的代码，其原理就是当需要添加锁时，会在goroutine中添加一个阻塞函数，来阻塞住其他请求，只有操作结束时，才会释放锁，代码如下：

```
func (m *Mutex) Lock() {
  // Fast path: grab unlocked mutex.
  if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
    if raceenabled {
      raceAcquire(unsafe.Pointer(m))
    }
    return
  }

  awoke := false
  for {
    old := m.state
    new := old | mutexLocked
    if old&mutexLocked != 0 {
      new = old + 1<<mutexWaiterShift
    }
    if awoke {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    panic("sync: unlock of unlocked mutex")
  }

  old := new
  for {
    // If there are no waiters or a goroutine has already
    // been woken or grabbed the lock, no need to wake anyone.
    if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken)
      return
    }
    // Grab the right to wake someone.
    new = (old - 1<<mutexWaiterShift) | mutexWoken
    if atomic.CompareAndSwapInt32(&m.state, old, new) {
      runtime_Semrelease(&m.sema)
      return
    }
    old = m.state
  }
}
```

让我们再回到刚才的思路，通过全ID可以获取到容器信息。但在实际操作过程当中，很少会使用全ID。因为它太长，太不方便记忆。更多的是使用短ID或者容器名称，因此后面的两个判断就是针对使用短ID和容器名称的情况。

deamon.GetByName用来处理当参数为容器名称时，如何查询与之

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

名称就可以查询到ID。Docker也正是这样做的，而保存这个对应关系的则是数据库。

Docker使用的是sqlite3，这个轻量级的文件数据库。当Docker Daemon启动时，会初始化两张表：entity和edge。entity表用来存储全ID；edge表用来存储容器名称和容器全ID的对应关系。两张表的数据库定义代码如下：

```
const (
    createEntityTable = `
    CREATE TABLE IF NOT EXISTS entity (
        id text NOT NULL PRIMARY KEY
    );`

    createEdgeTable = `
    CREATE TABLE IF NOT EXISTS edge (
        "entity_id" text NOT NULL,
        "parent_id" text NULL,
        "name" text NOT NULL,
        CONSTRAINT "parent_fk" FOREIGN KEY ("parent_id") REFERENC
        CONSTRAINT "entity_fk" FOREIGN KEY ("entity_id") REFERENC
        );
    `

    createEdgeIndices = `
    CREATE UNIQUE INDEX IF NOT EXISTS "name_parent_ix" ON "edge"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

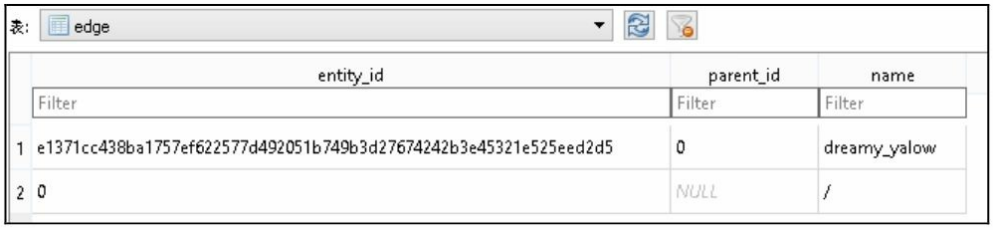
```
        e = next
    }
    return e, nil

}
```

这里可以看到代码中使用“/”进行字符串分隔，然后对分隔后的容器名分别进行查询。其实这里也只有一个容器名而已。Db.child函数会通过名称在edge表中查询其所对应的ID，查询代码如下：

```
func (db *Database) child(parent *Entity, name string) *Entity {
    var id string
    if err := db.conn.QueryRow("SELECT entity_id FROM edge WHERE
name).Scan(&id); err != nil {
        return nil
    }
    return &Entity{id}
}
```

图6-7是edge表数据的截图，方便读者参考里面的数据：



书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func (idx *TruncIndex) Get(s string) (string, error) {
    if s == "" {
        return "", ErrEmptyPrefix
    }
    var (
        id string
    )
    subTreeVisitFunc := func(prefix patricia.Prefix, item patric
        if id != "" {
            // we haven't found the ID if there are two or more I
            id = ""
            return ErrAmbiguousPrefix
        }
        id = string(prefix)
        return nil
    }

    idx.RLock()
    defer idx.RUnlock()
    if err := idx.trie.VisitSubtree(patricia.Prefix(s), subTreeVi
        return "", err
    }
    if id != "" {
        return id, nil
    }
    return "", fmt.Errorf("no such id: %s", s)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

当Docker创建容器时，会在Docker目录中创建一个以容器全ID为名称的目录。容器短ID则是全ID的前12位，这个目录里面放置了容器的所有信息，同时容器之间也存在继承关系。而Docker就从根目录开始逐层递归查找（walk），当匹配到相对应ID值的文件时，Docker就会把文件名作为全ID返回给最外层。

Docker之所以这样做，完全是考虑到全ID和短ID都是全局唯一的。在当前环境中，不可能出现重复值，因此才使用这种方式。但笔者认为，既然已经使用sqlite数据库，那么当创建容器时，将短ID值存放到表中，然后再使用索引检索，这样从效率和代码量上面应该都会比目前的方式要好一些。但这仅仅是笔者的一种想法，还没有经过测试，因而仅供参考。

让我们回归到主线上面来，现在已经获取到了容器信息，那么后面该做什么了呢？我们看一下Docker的后续处理：

```
    if c.Config == nil {
        c.Config = &runconfig.Config{}
    }

    newConfig, err := BuildFromConfig(d, c.Config, c.Changes)
    if err != nil {
        return "", err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    for i, n := range ast.Children {
        if err := builder.dispatch(i, n); err != nil {
            return nil, err
        }
    }

    return builder.Config, nil
}
```

这个函数里面封装了build对象，而最后返回的是build.Config，看来这个Config才是重点。看代码，就要看重点；找到重点，事半功倍：

```
type Config struct {
    Hostname     string
    Domainname   string
    User             string
    AttachStdin      bool
    AttachStdout     bool
    AttachStderr     bool
    ExposedPorts     map[nat.Port]struct{}
    PublishService   string
    Tty              bool // Attach standard streams to a tty, ir
    OpenStdin        bool // Open stdin
    StdinOnce        bool // If true, close stdin after the 1 att
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func Merge(userConf, imageConf *Config) error {
    if userConf.User == "" {
        userConf.User = imageConf.User
    }
    if len(userConf.ExposedPorts) == 0 {
        userConf.ExposedPorts = imageConf.ExposedPorts
    } else if imageConf.ExposedPorts != nil {
        if userConf.ExposedPorts == nil {
            userConf.ExposedPorts = make(nat.PortSet)
        }
        for port := range imageConf.ExposedPorts {
            if _, exists := userConf.ExposedPorts[port]; !exists
                userConf.ExposedPorts[port] = struct{}{}
            }
        }
    }

    if len(userConf.Env) == 0 {
        userConf.Env = imageConf.Env
    } else {
        for _, imageEnv := range imageConf.Env {
            found := false
            imageEnvKey := strings.Split(imageEnv, "=")[0]
            for _, userEnv := range userConf.Env {
                userEnvKey := strings.Split(userEnv, "=")[0]
                if imageEnvKey == userEnvKey {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
    return nil
}
```

代码量大但逻辑很简单，就是用增量数据覆盖了旧数据。通过上面的代码，我们也可以发现，合并时只合并了User、Expose、Env、Entrypoint、WorkDir和Volume这几项。也就是说，这几项是可以体现在新镜像中的，而其他参数即便修改了也不会体现在新镜像中。

万里长征到了最后一步，就是要创建文件层了。我们看一下最后的commit函数：

```
    img, err := d.Commit(container, c.Repo, c.Tag, c.Comment, c.A
    if err != nil {
        return "", err
    }

func (daemon *Daemon) Commit(container *Container, c *ContainerCo
{
    if c.Pause && !container.IsPaused() {
        container.Pause()
        defer container.Unpause()
    }

    rwTar, err := container.ExportRw()
    if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if c.Repo != "" {
        if err := daemon.repositories.Tag(c.Repo, c.Tag, img.ID,
            return img, err
        }
    }
    container.LogEvent("commit")
    return img, nil
}
```

commit函数的逻辑大致可以分为两步：导出源数据，创建新数据。

在导出源数据时，Docker采用将源数据打包压缩输出为二进制流的方式。然后将这个二进制流放到Docker目录下面进行恢复，最后命名为新镜像的ID值。说出来非常简单，但实际操作过程非常复杂，请看下面：

```
func (graph *Graph) Create(layerData archive.ArchiveReader, conta
author string, containerConfig, config *runconfig.Config) (*Image
    img := &Image{
        ID:              stringid.GenerateRandomID(),
        Comment:         comment,
        Created:         time.Now().UTC(),
        DockerVersion:  dockerversion.VERSION,
        Author:          author,
        Config:          config,
        Architecture:   runtime.GOARCH,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  // We need this entire operation to be atomic within the engine
  // this doesn't mean Register is fully safe yet.
  graph.imageMutex.Lock(img.ID)
  defer graph.imageMutex.Unlock(img.ID)

  // The returned `error` must be named in this function's signat
  // `err` is not shadowed in this deferred cleanup.
  defer func() {
    // If any error occurs, remove the new dir from the driver.
    // Don't check for errors since the dir might not have been c
    if err != nil {
      graph.driver.Remove(img.ID)
    }
  }()
  if graph.Exists(img.ID) {
    return fmt.Errorf("Image %s already exists", img.ID)
  }

if err := os.RemoveAll(graph.imageRoot(img.ID)); err != nil && !o
    return err
  }

  // If the driver has this ID but the graph doesn't, remove it f
  // (the graph is the source of truth).
  // Ignore errors, since we don't know if the driver correctly r
  // (FIXME: make that mandatory for drivers).
  graph.driver.Remove(img.ID)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

据中合并新增数据。

当这一切都完成之后，Docker会将这个临时目录改为新镜像的id值。此时从AUFS文件系统的角度来看，已经新添加了一个文件层。但这个镜像目前还只存在于AUFS当中，如果想要Docker可以识别这个新镜像，则需要将新镜像注册到Docker Daemon之中，也就是整个commit环节中的最后一步：graph.idIndex.Add(img.ID)

```
func (idx *TruncIndex) Add(id string) error {
    idx.Lock()
    defer idx.Unlock()
    if err := idx.addID(id); err != nil {
        return err
    }
    return nil
}
```

当这些步骤都完成之后，新镜像就诞生了。至此commit操作结束，Daemon将向客户端返回新镜像的ID值。

通过对commit命令的分析，我们明白了Docker会通过多种方式来保存容器信息（map集合和sqlite数据库），同时也明白了Docker如何产生一个镜像，其本质就是通过AUFS文件系统来创建目录。如果需要Docker识别并且可以使用这个新镜像，则需要在Docker Daemon中注册新镜像。

4. cp命令

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    cfg := &types.CopyConfig{
        Resource: info[1],
    }
    stream, statusCode, err := cli.call("POST", "/containers/"+ir
    if stream != nil {
        defer stream.Close()
    }
    if statusCode == 404 {
        return fmt.Errorf("No such container: %v", info[0])
    }
    if err != nil {
        return err
    }

    hostPath := cmd.Arg(1)
    if statusCode == 200 {
        if hostPath == "-" {
            _, err = io.Copy(cli.out, stream)
        } else {
            err = archive.Untar(stream, hostPath, &archive.TarOpt
        }
        if err != nil {
            return err
        }
    }
    return nil
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if cfg.Resource == "" {
        return fmt.Errorf("Path cannot be empty")
    }

    data, err := s.daemon.ContainerCopy(vars["name"], cfg.Resourc
    if err != nil {
        if strings.Contains(strings.ToLower(err.Error()), "no suc
            w.WriteHeader(http.StatusNotFound)
            return nil
        }
        if os.IsNotExist(err) {
            return fmt.Errorf("Could not find the file %s in cont
        }
        return err
    }
    defer data.Close()

    w.Header().Set("Content-Type", "application/x-tar")
    if _, err := io.Copy(w, data); err != nil {
        return err
    }

    return nil
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    return container.Copy(res)
}
```

在ContainerCopy中，首先判断容器是否存在。这一点非常有必要，因为容器ID值是用户自行指定的，所以无法确定此容器是否一定存在。这一步几乎在每个命令的处理中都能看到。虽然烦琐，但不能省略。如果读者需要在Docker内核中定制开发其他命令，那么这一步必须要添加。

然后ContainerCopy又通过如下代码获取了目标文件的相对路径：

```
    if res[0] == '/' {
        res = res[1:]
    }
```

读者同时也就明白了Docker在进行复制操作时，所有的文件路径都是基于“/”的相对路径。即使提供了全路径，这里也会处理得到相对路径。res=res[1:]是指从res这个切片第二个元素开始一直截取到最后一个元素，然后将新生成的切片赋给原值（也就是去掉首元素）。

最后调用container的copy函数来进行最后的打包工作，copy函数实现方法如下：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  var stat os.FileInfo
  stat, err = os.Stat(m.Source)
  if err != nil {
    return nil, err
  }
  if err = fileutils.CreateIfNotExists(dest, stat.IsDir()); err
    return nil, err
  }
  if err = mount.Mount(m.Source, dest, "bind", "rbind,ro"); err
    return nil, err
  }
}
basePath, err := container.GetResourcePath(resource)
if err != nil {
  return nil, err
}
stat, err := os.Stat(basePath)
if err != nil {
  return nil, err
}
var filter []string
if !stat.IsDir() {
  d, f := filepath.Split(basePath)
  basePath = d
  filter = []string{f}
} else {
  filter = []string{filepath.Base(basePath)}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

container.Lock()，这是一个同步锁。在前面的命令中，我们提到过这个函数功能。因为我们需要从容器中将文件复制出来，所以在复制过程中，目标文件不能发生更改。在复制开始之前需要添加一个同步锁，而为了保证这个同步锁可以被释放掉，后面紧跟着defer container.Unlock()。defer是Golang中用于函数退出时执行的语句，作用相当于Java中的final语句。通过这句代码，就可以保证无论copy函数执行成功与否，同步锁都可以被释放掉。

接下来的代码是查找容器挂载点，这步操作貌似和打包目标文件没有多少关联。真实情况是这样吗？我们先看一下这段代码实现了什么功能：

```
    mounts, err := container.setupMounts()
    if err != nil {
        return nil, err
    }
```

这段代码获取了当前容器所有的挂载点信息。比如说，在MySQL容器中，我们通过mount命令或者df–h命令查看当前挂载信息，可以看到：

```
root@8f02e3e34665:/# df -h
Filesystem      Size  Used  Avail  Use%   Mounted on
none            1.8G  1.8G   41M     98%    /
tmpfs          1002M     0 1002M     0%     /dev
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        var stat os.FileInfo
        stat, err = os.Stat(m.Source)
        if err != nil {
            return nil, err
        }
        if err = fileutils.CreateIfNotExists(dest, stat.IsDir());
            return nil, err
        }
        if err = mount.Mount(m.Source, dest, "bind", "rbind,ro");
            return nil, err
        }
    }
```

就是将每个挂载点中的数据同步到Docker当前容器文件系统中。这么做的目的在于Docker最后打包时，都是在容器文件系统中操作。挂载点数据如果保存在host文件目录中，那么最终将无法打包，因此需要同步过去。

这个同步过程简要描述如下，假设目前同步的挂载点是/etc/resolv.conf文件，那么此文件的源文件保存在主机：

```
/var/lib/docker/containers/8f02e3e346657c531df1c4505cea98104a010a
lv.conf
```

而目标路径，也就是容器中的文件系统真实路径则是：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
    archive, err := archive.TarWithOptions(basePath, &archive.Tar
        Compression:  archive.Uncompressed,
        IncludeFiles: filter,
    })
```

Resource是目标文件在容器中的相对路径，basePath则是此文件在主机中的绝对路径。当通过container.GetResourcePath(resource)获取到绝对路径之后，就调用archive.TarWithOptions进行压缩操作，最后通过：

```
reader := ioutils.NewReadCloserWrapper(archive, func() error {
        err := archive.Close()
        container.CleanupStorage()
        container.UnmountVolumes(true)
        container.Unmount()
        return err
    })
    container.LogEvent("copy")
    return reader, nil
```

逐层返回压缩包的二进制数据，直至返回到客户端层，最后客户端通过err=archive.Untar(stream, hostPath,&archive.TarOptions{NoLchown:true})进行解压缩，拿到最终的文件。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    config, hostConfig, cmd, err := runconfig.Parse(cmd, args)
```

但却可以完成下面长长的参数解析：

```
Create a new container
  -a, --attach=[]            Attach to STDIN, STDOUT or STDERR
  --add-host=[]              Add a custom host-to-IP mapping (h
  --blkio-weight=0           Block IO (relative weight), betwee
  -c, --cpu-shares=0         CPU shares (relative weight)
  --cap-add=[]               Add Linux capabilities
  --cap-drop=[]              Drop Linux capabilities
  --cgroup-parent=           Optional parent cgroup for the con
  --cidfile=                 Write the container ID to the file
  --cpu-period=0             Limit CPU CFS (Completely Fair Sch
  --cpu-quota=0              Limit the CPU CFS quota
  --cpuset-cpus=             CPUs in which to allow execution (
  --cpuset-mems=             MEMs in which to allow execution (
  --device=[]                Add a host device to the container
  --dns=[]                   Set custom DNS servers
  --dns-search=[]            Set custom DNS search domains
  -e, --env=[]               Set environment variables
  --entrypoint=              Overwrite the default ENTRYPOINT o
  --env-file=[]              Read in a file of environment vari
  --expose=[]                Expose a port or a range of ports
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    ulimits   = make(map[string]*ulimit.Ulimit)
    flUlimits = opts.NewUlimitOpt(ulimits)

    flPublish     = opts.NewListOpts(nil)
    flExpose      = opts.NewListOpts(nil)
    flDns         = opts.NewListOpts(opts.ValidateIPAddress)
    flDnsSearch   = opts.NewListOpts(opts.ValidateDNSSearch)
    flExtraHosts  = opts.NewListOpts(opts.ValidateExtraHost)
    flVolumesFrom = opts.NewListOpts(nil)
    flLxcOpts     = opts.NewListOpts(nil)
    flEnvFile     = opts.NewListOpts(nil)
    flCapAdd      = opts.NewListOpts(nil)
    flCapDrop     = opts.NewListOpts(nil)
    flGroupAdd    = opts.NewListOpts(nil)
    flSecurityOpt = opts.NewListOpts(nil)
    flLabelsFile  = opts.NewListOpts(nil)
    flLoggingOpts = opts.NewListOpts(nil)
    flNetwork   = cmd.Bool([]string{"#n", "#-networking"}, true,
    flPrivileged  = cmd.Bool([]string{"#privileged", "-privileged
to this container")
    flPidMode         = cmd.String([]string{"-pid"}, "", "PID nam
    flUTSMode         = cmd.String([]string{"-uts"}, "", "UTS nam
    flPublishAll      = cmd.Bool([]string{"P", "-publish-all"}, f
random ports")
    flStdin           = cmd.Bool([]string{"i", "-interactive"}, f
attached")
    flTty             = cmd.Bool([]string{"t", "-tty"}, false, "A
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
execution (0-3, 0,1)")
    flCpusetMems      = cmd.String([]string{"-cpuset-mems"}, "",
(0-3, 0,1)")
    flBlkioWeight     = cmd.Int64([]string{"-blkio-weight"}, 0, "
10 and 1000")
    flNetMode         = cmd.String([]string{"-net"}, "default", "
container")
    flMacAddress      = cmd.String([]string{"-mac-address"}, "",
92:d0:c6:0a:29:33)")
    flIpcMode         = cmd.String([]string{"-ipc"}, "", "IPC nam
    flRestartPolicy   = cmd.String([]string{"-restart"}, "no", "R
container exits")
    flReadonlyRootfs  =  cmd.Bool([]string{"-read-only"},  false,
filesystem as read only")
    flLoggingDriver   = cmd.String([]string{"-log-driver"}, "", "
    flCgroupParent    = cmd.String([]string{"-cgroup-parent"}, ""
container")
  )

  cmd.Var(&flAttach, []string{"a", "-attach"}, "Attach to STDIN,
  cmd.Var(&flVolumes, []string{"v", "-volume"}, "Bind mount a vol
  cmd.Var(&flLinks, []string{"#link", "-link"}, "Add link to anot
  cmd.Var(&flDevices, []string{"-device"}, "Add a host device to
  cmd.Var(&flLabels, []string{"l", "-label"}, "Set meta data on a
  cmd.Var(&flLabelsFile, []string{"-label-file"}, "Read in a line
  cmd.Var(&flEnv, []string{"e", "-env"}, "Set environment variabl
  cmd.Var(&flEnvFile, []string{"-env-file"}, "Read in a file of e
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  Domainname:        domainname,
  ExposedPorts:      ports,
  User:             *flUser,
  Tty:              *flTty,
  NetworkDisabled:   !*flNetwork,
  OpenStdin:         *flStdin,
  AttachStdin:       attachStdin,
  AttachStdout:      attachStdout,
  AttachStderr:      attachStderr,
  Env:              envVariables,
  Cmd:              runCmd,
  Image:            image,
  Volumes:          flVolumes.GetMap(),
  MacAddress:        *flMacAddress,
  Entrypoint:        entrypoint,
  WorkingDir:        *flWorkingDir,
  Labels:           convertKVStringsToMap(labels),
}

hostConfig := &HostConfig{
  Binds:              binds,
  ContainerIDFile:    *flContainerIDFile,
  LxcConf:            lxcConf,
  Memory:             flMemory,
  MemorySwap:         MemorySwap,
  CpuShares:          *flCpuShares,
  CpuPeriod:          *flCpuPeriod,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  CgroupParent:       *flCgroupParent,
}

applyExperimentalFlags(expFlags, config, hostConfig)

// When allocating stdin in attached mode, close stdin at clien
if config.OpenStdin && config.AttachStdin {
  config.StdinOnce = true
}
return config, hostConfig, cmd, nil
```

我们省略了函数解析和这段代码之间的逻辑，那部分逻辑是用来给各项参数赋值的。上面所说的另外一个功能，就是这里大家所看到的：Parse函数返回了一个config结构体和hostConfig结构体。

这两个结构体分别是用来做什么的呢？hostConfig结构体是容器的虚拟化运行参数，包括CPU、内存、I/O、网络等命名空间参数。而config是容器的运行参数，包括主机名、用户名、环境变量等运行参数。

Parse函数返回这两个结构体准备做什么用呢？这需要我们返回到create主流程来一探究竟。让我们继续研究create后面的逻辑：

```
  config, hostConfig, cmd, err := runconfig.Parse(cmd, args)
  if err != nil {
      cmd.ReportError(err.Error(), true)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    mergedConfig := runconfig.MergeConfigs(config, hostConfig)

    var containerIDFile *cidFile
    if cidfile != "" {
        var err error
        if containerIDFile, err = newCIDFile(cidfile); err != nil
            return nil, err
        }
        defer containerIDFile.Close()
    }

    //create the container
    stream, statusCode, err := cli.call("POST", "/containers/crea
mergedConfig, nil)
    //if image not found try to pull it
    if statusCode == 404 && strings.Contains(err.Error(), config.
        repo, tag := parsers.ParseRepositoryTag(config.Image)
        if tag == "" {
            tag = tags.DEFAULTTAG
        }
        fmt.Fprintf(cli.err, "Unable to find image '%s' locally\n
        if err = cli.pullImageCustomOut(config.Image, cli.err); e
            return nil, err
        }
        if  stream,  _,  err  =  cli.call("POST",  "/containers/c
mergedConfig, nil); err != nil {
            return nil, err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

第二件事：调用服务端的CreateAPI。在Docker体系中，所有的客户端都是用来收集参数的，真正负责运行的都是服务端，而两者之间通信的桥梁就是RestFulAPI。

第三件事：当Create函数使用的镜像在本地不存在时，Docker需要从仓库（包括远程仓库或者本地仓库）下载指定的镜像。

到目前为止，我们已经读完了Create客户端的业务逻辑。如果想要理解一个容器是如何由镜像变化而来的，那么就要深入到Create的服务端去。下面就是Create服务端的代码分析。

按照惯例，我们需要拿到Create服务端真正的处理函数，因此需要从RestfulAPI和函数的对应入手：

```
"/containers/create":           s.postContainersCreate,
```

顺藤摸瓜，我们进入postContainersCreate函数的内部：

```
func (s *Server) postContainersCreate(version version.Version, w
vars map[string]string) error {
    if err := parseForm(r); err != nil {
        return err
    }
    if err := checkForJson(r); err != nil {
        return err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

了Docker命令处理的流程。客户端负责收集参数并且发送到服务端，服务端首先要做的就是解析客户端发来的参数，并且封装成所需的结构体传入后面真正的处理函数，Create函数也不例外。

postContainerCreate第二件事就是调用s.daemon.ContainerCreate函数来执行真正的创建过程。

而postContainerCreate最后一件事就是返回最终的处理结果。

我们省略掉参数解析和返回处理结果两步，重点看一下s.daemon.ContainerCreate这个函数。

```
func (daemon *Daemon) ContainerCreate(name string, config *
*runconfig.HostConfig) (string, []string, error) {
    if config == nil {
        return "", nil, fmt.Errorf("Config cannot be empty in orc
    }

    warnings, err := daemon.verifyHostConfig(hostConfig)
    if config == nil {
        return "", nil, fmt.Errorf("Config cannot be empty in orc
    }

    warnings, err := daemon.verifyHostConfig(hostConfig)
    if err != nil {
        return "", warnings, err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

在ContainerCreate函数中，首先是调用verifyHostConfig来确认各项参数是否在合法区间内。为了让读者可以了解各项参数的合法范围，因此特提供verifyHostConfig判断代码如下：

```
func (daemon *Daemon) verifyHostConfig(hostConfig *runconfig.Host
    var warnings []string

    if hostConfig == nil {
        return warnings, nil
    }

    for port := range hostConfig.PortBindings {
        _, portStr := nat.SplitProtoPort(string(port))
        if _, err := nat.ParsePort(portStr); err != nil {
            return warnings, fmt.Errorf("Invalid port specificati
        }
    }

    if hostConfig.LxcConf.Len() > 0 && !strings.Contains(daemon.E
        return  warnings,  fmt.Errorf("Cannot  use  --lxc-conf  w
ExecutionDriver().Name())
    }
    if hostConfig.Memory != 0 && hostConfig.Memory < 4194304 {
        return warnings, fmt.Errorf("Minimum memory limit allowed
    }
    if hostConfig.Memory > 0 && !daemon.SystemConfig().MemoryLimi
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
{
        return warnings, fmt.Errorf("Range of blkio weight is fro
    }
    if hostConfig.OomKillDisable && !daemon.SystemConfig().OomKil
        hostConfig.OomKillDisable = false
        return warnings, fmt.Errorf("Your kernel does not support
    }
    return warnings, nil
}
```

待所有的参数都检测完成之后，就开始调用Create函数创建容器。在执行创建之前，同样存在一些合法性检查。首先检查的就是config.Image，如下：

```
if config.Image != "" {
        img, err = daemon.repositories.LookupImage(config.Image)
        if err != nil {
            return nil, nil, err
        }
        if err = img.CheckDepth(); err != nil {
            return nil, nil, err
        }
        imgID = img.ID
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
(host:port:port) has been deprecated. Use -p to publish the ports
    }
    if img != nil && img.Config != nil {
        if err := runconfig.Merge(config, img.Config); err != nil
            return nil, err
        }
    }
    if config.Entrypoint.Len() == 0 && config.Cmd.Len() == 0 {
        return nil, fmt.Errorf("No command specified")
    }
    return warnings, nil
}
```

这段代码主要使用自定义容器配置信息覆盖镜像中的配置信息，并将最后的配置信息作为容器运行配置信息。与此同时，此段代码也会检查容器的端口信息是否合法（这里的端口合法指的是端口写法是否合法，建议使用Docker目前通用的-p写法来发布端口）。如果容器的entrypoint和cmd为空，则也会报错。因为这两项为空，就意味着容器没有入口函数，那么创建起来也毫无价值。

而后Daemon会继续进行IP检查和一些参数配置，相对比较简单，下面只提供代码实现：

```
    if !config.NetworkDisabled && daemon.SystemConfig().IPv4Forwa
        warnings = append(warnings, "IPv4 forwarding is disabled.
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        err error
        id  = stringid.GenerateRandomID()
    )

    if name == "" {
        if name, err = daemon.generateNewName(id); err != nil {
            return "", "", err
        }
        return id, name, nil
    }

    if name, err = daemon.reserveName(id, name); err != nil {
        return "", "", err
    }

    return id, name, nil
}
```

可以看到ID值其实是一个随机字符串，而这个随机字符串是由下面代码生成的：

```
func GenerateRandomID() string {
    for {
        id := make([]byte, 32)
        if _, err := io.ReadFull(rand.Reader, id); err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            name = "/" + name
        }

        if _, err := daemon.containerGraph.Set(name, id); err !=
            if !graphdb.IsNonUniqueNameError(err) {
                return "", err
            }
            continue
        }
        return name, nil
    }

    name = "/" + stringid.TruncateID(id)
    if _, err := daemon.containerGraph.Set(name, id); err != nil
        return "", err
    }
    return name, nil
}
```

name，也就是容器名称，是由namesgenerator.GetRandomName(i)生成的，而ID仅仅用来向Docker内置的SQLite数据库插入数据时使用。在执行Set函数时，Daemon首先会使用ID和name来查询当前数据库中是否有相同记录，如果没有的话，再执行插入。如果存在相同记录，则会再次生成一个name，重新执行插入。

而namesgenerator.GetRandomName(i)每次都会随机生成一个名称，

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "backstabbing",
        "berserk",
        "boring",
        "clever",
        "cocky",
        "compassionate",
        "condescending",
        "cranky",
        "desperate",
        "determined",
        "distracted",
        "dreamy",
        "drunk",
        "ecstatic",
        "elated",
        "elegant",
        "evil",
        "fervent",
        "focused",
        "furious",
        "gloomy",
        "goofy",
        "grave",
        "happy",
        "high",
        "hopeful",
        "hungry",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
right = [...]string{
        //  Muhammad  ibn  Jābir  al-Ḥarrānī  al-Battānī  was  a
https://en.wikipedia.org/wiki/Mu%E1%B8%A5ammad_ibn_J%C4%81bir_al-
tt%C4%81n%C4%AB
        "albattani",

        // June Almeida - Scottish virologist who took the first
https://en.wikipedia.org/wiki/June_Almeida
        "almeida",

        // Archimedes was a physicist, engineer and mathematician
them here. https://en.wikipedia.org/wiki/Archimedes
        "archimedes",

        // Maria Ardinghelli - Italian translator, mathematician
https://en.wikipedia.org/wiki/Maria_Ardinghelli
        "ardinghelli",

        // Aryabhata - Ancient Indian mathematician-astronomer du
https://en.wikipedia.org/wiki/Aryabhata
        "aryabhata",

        // Charles Babbage invented the concept of a programmable
https://en.wikipedia.org/wiki/Charles_Babbage.
        "babbage",
        // Stefan Banach - Polish mathematician, was one of the f
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
and Leibniz by over half a millenium - https://en.wikipedia.org/w
        "bhaskara",

    // Elizabeth Blackwell - American doctor and first American w
https://en.wikipedia.org/wiki/Elizabeth_Blackwell
        "blackwell",

        // Niels Bohr is the father of quantum theory. https://en
        "bohr",

        // Satyendra Nath Bose - He provided the foundation for B
of the Bose–Einstein condensate. - https://en.wikipedia.org/wiki/
        "bose",

        // Brahmagupta - Ancient Indian mathematician during 598-
zero - https://en.wikipedia.org/wiki/Brahmagupta#Zero
        "brahmagupta",

        // Emmett Brown invented time travel. https://en.wikipedi
Brian Goff)
        "brown",

        // Rachel Carson - American marine biologist and conserva
other writings are credited with advancing the global env
https://en.wikipedia.org/wiki/Rachel_Carson
        "carson",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
https://en.wikipedia.org/wiki/Leonardo_da_Vinci.
        "davinci",

        // Albert Einstein invented the general theory of relativ
https://en.wikipedia.org/wiki/Albert_Einstein
        "einstein",

        // Gertrude Elion - American biochemist, pharmacologist a
Prize in Medicine - https://en.wikipedia.org/wiki/Gertrude_Elion
        "elion",

        // Douglas Engelbart gave the mother of all demos:
https://en.wikipedia.org/wiki/Douglas_Engelbart
        "engelbart",

        // Euclid invented geometry. https://en.wikipedia.org/wik
        "euclid",

        // Pierre de Fermat pioneered several aspects of modern m
https://en.wikipedia.org/wiki/Pierre_de_Fermat
        "fermat",

        // Enrico Fermi invented the first nuclear reactor.
https://en.wikipedia.org/wiki/Enrico_Fermi.
        "fermi",

        //  Richard  Feynman  was  a  key  contributor  to  quant
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        // Dorothy Hodgkin was a British biochemist, credited wit
crystallography. She was awarded the Nobel Prize in Chemistry in
https://en.wikipedia.org/wiki/Dorothy_Hodgkin
        "hodgkin",

        // Erna Schneider Hoover revolutionized modern communicat
telephon switching method. https://en.wikipedia.org/wiki/Erna_Sch
        "hoover",

        // Grace Hopper developed the first compiler for a comput
credited with popularizing the term "debugging" for fixing comput
https://en.wikipedia.org/wiki/Grace_Hopper
        "hopper",

        // Hypatia - Greek Alexandrine Neoplatonist philosopher i
mothers of mathematics - https://en.wikipedia.org/wiki/Hypatia
        "hypatia",

        // Yeong-Sil Jang was a Korean scientist and astronomer d
the first metal printing press and water gauge. https://en.wikipe
        "jang",

        // Karen Spärck Jones came up with the concept of inverse
in most search engines today. https://en.wikipedia.org/wiki/Karen
        "jones",

        // Jack Kilby and Robert Noyce have invented silicone int
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        // Mary Leakey - British paleoanthropologist who discover
skull - https://en.wikipedia.org/wiki/Mary_Leakey
        "leakey",

        // Ada Lovelace invented the first algorithm. https://en.
(thanks James Turnbull)
        "lovelace",

        // Auguste and Louis Lumière - the first filmmakers in hi
https://en.wikipedia.org/wiki/Auguste_and_Louis_Lumi%C3%A8re
        "lumiere",
        // Mahavira - Ancient Indian mathematician during 9th cen
identities - https://en.wikipedia.org/wiki/Mah%C4%81v%C4%ABra_(ma
        "mahavira",

        // Maria Mayer - American theoretical physicist and Nobel
the nuclear shell model of the atomic nucleus - https://en.wikipe
        "mayer",

        // John McCarthy invented LISP: https://en.wikipedia.org/
        "mccarthy",

        // Barbara McClintock - a distinguished American cytogene
Physiology or Medicine for discovering transposons. https://en.wi
        "mcclintock",

        //  Malcolm  McLean  invented  the  modern  shipping  con
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "panini",

        // Cecilia Payne-Gaposchkin was an astronomer and astroph
her Ph.D. thesis an explanation for the composition of stars in t
hydrogen and helium. https://en.wikipedia.org/wiki/Cecilia_Payne-
        "payne",

        // Ambroise Pare invented modern surgery. https://en.wiki
        "pare",

        // Louis Pasteur discovered vaccination, fermentation and
https://en.wikipedia.org/wiki/Louis_Pasteur.
        "pasteur",

        // Radia Perlman is a software designer and network engin
of the spanning-tree protocol (STP). https://en.wikipedia.org/wik
        "perlman",

        // Rob Pike was a key contributor to Unix, Plan 9, the X
programming language. https://en.wikipedia.org/wiki/Rob_Pike
        "pike",

        //  Henri  Poincaré  made  fundamental  contributions  in
https://en.wikipedia.org/wiki/Henri_Poincar%C3%A9
        "poincare",

        // Laura Poitras is a director and producer whose work, m
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        // Meghnad Saha - Indian astrophysicist best known for hi
used to describe chemical and physical conditions in stars -
https://en.wikipedia.org/wiki/Meghnad_Saha
        "saha",

        // Jean E. Sammet developed FORMAC, the first widely usec
manipulation of mathematical formulas. https://en.wikipedia.org/w
        "sammet",

        // Françoise Barré-Sinoussi - French virologist and Nobel
Medicine; her work was fundamental in identifying HIV as
https://en.wikipedia.org/wiki/Fran%C3%A7oise_Barr%C3%A9-Sinoussi
        "sinoussi",

        // Richard Matthew Stallman - the founder of the Free Sof
Free Software Foundation, and the League for Programming Freedom.
copyleft to protect the ideals of this movement, and enshrined th
(General Public License) for software. https://en.wikiquote.org/w
        "stallman",

        // Aaron Swartz was influential in creating RSS, Markdown
of the internet as we know it today. He was devoted to freedom of
https://en.wikiquote.org/wiki/Aaron_Swartz
        “swartz",

       // Nikola Tesla invented the AC electric system and every
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
first successful airplane and making the first controlled, powere
human flight - https://en.wikipedia.org/wiki/Wright_brothers
        "wright",

        // Rosalyn Sussman Yalow - Rosalyn Sussman Yalow was an A
co-winner of the 1977 Nobel Prize in Physiology or Medicine for c
technique. https://en.wikipedia.org/wiki/Rosalyn_Sussman_Yalow
        "yalow",

        // Ada Yonath - an Israeli crystallographer, the first wo
Nobel prize in the sciences. https://en.wikipedia.org/wiki/Ada_Yo
        "yonath",
    }
```

left数组是一些常用的人名，而right则是近现代科技界的名人，并且每个名人后面还跟着一段个人简介。通过这么有创意的代码，马上就令Docker容器名称有了一些文化底蕴。

这段代码分析得有些远，让我们继续回到原先的轨道上来。等容器名称和容器ID创建成功之后，Daemon就开始注册这个容器了，而使用的函数就是Register函数，代码如下：

```
    if err := daemon.Register(container); err != nil {
        return nil, nil, err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
// don't update the Suffixarray if we're starting up
// we'll waste time if we update it for every container
daemon.idIndex.Add(container.ID)

if err := daemon.verifyVolumesInfo(container); err != nil {
    return err
}

if err := container.prepareMountPoints(); err != nil {
    return err
}

if container.IsRunning() {
    logrus.Debugf("killing old running container %s", contain

    container.SetStopped(&execdriver.ExitStatus{ExitCode: 0})

    // use the current driver and ensure that the container i
    cmd := &execdriver.Command{
        ID: container.ID,
    }
    daemon.execDriver.Terminate(cmd)

    if err := container.Unmount(); err != nil {
        logrus.Debugf("unmount error %s", err)
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
// Add adds a new ID to the TruncIndex
func (idx *TruncIndex) Add(id string) error {
    idx.Lock()
    defer idx.Unlock()
    if err := idx.addID(id); err != nil {
        return err
    }
    return nil
}
```

下一步就是处理数据卷组：

```
if err := daemon.verifyVolumesInfo(container); err != nil {
        return err
    }
func (daemon *Daemon) verifyVolumesInfo(container *Container) err
    // Inspect old structures only when we're upgrading from old
    // to versions >= 1.7 and the MountPoints has not been popula
    if len(container.MountPoints) == 0 && len(container.Volumes)
        for destination, hostPath := range container.Volumes {
            vfsPath := filepath.Join(daemon.root, "vfs", "dir")
            rw := container.VolumesRW != nil && container.Volumes

            if strings.HasPrefix(hostPath, vfsPath) {
                id := filepath.Base(hostPath)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            continue
        }
        dataPath := l.(*local.Root).DataPath(m.Name)
        volumePath := filepath.Dir(dataPath)

        d, err := ioutil.ReadDir(volumePath)
        if err != nil {
            // If the volume directory doesn't exist yet it w
            // so we only return the error when there is a di
            if !os.IsNotExist(err) {
                return err
            }
            // Do not check when the volume directory does no
            continue
        }
        if validVolumeLayout(d) {
            continue
        }

        if err := os.Mkdir(dataPath, 0755); err != nil {
            return err
        }

        // Move data inside the data directory
        for _, f := range d {
            oldp := filepath.Join(volumePath, f.Name())
            newp := filepath.Join(dataPath, f.Name())
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

信息，并且在容器所属的文件系统中依次创建这些挂载点目录。

最后会调用Todisk函数来使所创建的卷组信息落地。我们看一下Todisk函数的真面目：

```
func (container *Container) toDisk() error {
    data, err := json.Marshal(container)
    if err != nil {
        return err
    }

    pth, err := container.jsonPath()
    if err != nil {
        return err
    }

    if err := ioutil.WriteFile(pth, data, 0666); err != nil {
        return err
    }

    return container.WriteHostConfig()
}
```

可以看到，其实ToDisk函数就是将刚才所做的卷组变更都写入hostConfig文件当中。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
    return nil
}
```

可以看到，创建根文件层没有非常神秘的操作。第一步是在Docker所有的文件目录中创建此容器的根文件，也就是85996fe4c6afd9f1a4b8534df3b4aa0c10ecab92fc70b6170160204a4d6f9f12-init这样的文件。第二步是调用目前Daemon所使用的文件驱动创建文件层。假如我们使用AUFS文件系统，那么就使用AUFS的Create函数来创建文件层，代码如下：

```
func (a *Driver) Create(id, parent string) error {
    if err := a.createDirsFor(id); err != nil {
        return err
    }
    // Write the layers metadata
    f, err := os.Create(path.Join(a.rootPath(), "layers", id))
    if err != nil {
        return err
    }
    defer f.Close()

    if parent != "" {
        ids, err := getParentIds(a.rootPath(), parent)
        if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "/dev/shm":          "dir",
        "/proc":             "dir",
        "/sys":              "dir",
        "/.dockerinit":     "file",
        "/.dockerenv":      "file",
        "/etc/resolv.conf": "file",
        "/etc/hosts":       "file",
        "/etc/hostname":    "file",
        "/dev/console":     "file",
        "/etc/mtab":        "/proc/mounts",
    } {
        parts := strings.Split(pth, "/")
        prev := "/"
        for _, p := range parts[1:] {
            prev = filepath.Join(prev, p)
            syscall.Unlink(filepath.Join(initLayer, prev))
        }

        if _, err := os.Stat(filepath.Join(initLayer, pth)); err
            if os.IsNotExist(err) {
                if err := system.MkdirAll(filepath.Join(initLayer
                    return err
                }
                switch typ {
                case "dir":
                    if err := system.MkdirAll(filepath.Join(initL
                        return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

权限的文件。

此时在主流程中剩余的代码逻辑就是处理挂载点了：

```
for spec := range config.Volumes {
        var (
            name, destination string
            parts             = strings.Split(spec, ":")
        )
        switch len(parts) {
        case 2:
            name, destination = parts[0], filepath.Clean(parts[1]
        default:
            name = stringid.GenerateRandomID()
            destination = filepath.Clean(parts[0])
        }
        // Skip volumes for which we already have something mount
        // destination because of a --volume-from.
        if container.isDestinationMounted(destination) {
            continue
        }
        path, err := container.GetResourcePath(destination)
        if err != nil {
            return nil, nil, err
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
return container.ID, warnings, nil
```

其他程序依次返回，直至容器ID被返回至客户端。至此，一个容器可以说完成创建了。但有的读者也许会问，这段代码中只创建了容器的根文件层，其他文件层都没有被处理呀？

其实在create时只创建了根文件层，但是我们已经把此根文件层的父级目录写入了维护文件，因而等此容器真正运行时，Daemon才会逐层加载文件层，最终呈现给我们一个完整的文件操作系统。

6. diff命令

diff命令可以用来展现某个容器的差异，而这些差异都是以文件的形式展现的。这句话该如何理解呢？比如说，我们创建了一个运行Oracle的容器，并且在这个容器中，我们创建了一张表，在表中插入了几条数据，但并没有做commit提交。此时如果我们使用diff命令查看容器变化，那么可以看到这张表所在的数据文件，但却看不到刚才插入的那几条数据（假如我们能看到数据的话）。

为什么会有这样的结果呢？这是因为当我们创建表时使用的是DML语句。DDL语句执行完之后，就会生成数据表所使用的文件。而我们插入记录，使用的是DDL语句。这些语句只有执行commit时，才会写入文件中。没有执行commit的时候，这些记录只是保存在内存中（这里，我们为简化问题暂时忽略了回滚段的存在）。

当执行diff命令时，可以看到当前容器只有一些数据文件发生了变化。这时因为内存中保存的数据是无法比较的，最终反映出来的差异都

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if cmd.Arg(0) == "" {
        return fmt.Errorf("Container name cannot be empty")
    }

    rdr, _, err := cli.call("GET", "/containers/"+cmd.Arg(0)+"/ch
    if err != nil {
        return err
    }

    changes := []types.ContainerChange{}
    if err := json.NewDecoder(rdr).Decode(&changes); err != nil {
        return err
    }

    for _, change := range changes {
        var kind string
        switch change.Kind {
        case archive.ChangeModify:
            kind = "C"
        case archive.ChangeAdd:
            kind = "A"
        case archive.ChangeDelete:
            kind = "D"
        }
        fmt.Fprintf(cli.out, "%s %s\n", kind, change.Path)
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return writeJSON(w, http.StatusOK, changes)
}
```

而按照我们之前的分析经验来看，Daemon一定会首先判断指定的容器是否存在。这个判断就在daemon.ContainerChanges函数中：

```
func (daemon *Daemon) ContainerChanges(name string) ([]archive.Ch
    container, err := daemon.Get(name)
    if err != nil {
        return nil, err
    }

    return container.Changes()
}
```

这个判断一举两得，一方面判断了指定容器是否存在；另一方面当容器存在时，直接使用此容器结构体中的数据。在container.Changes函数中，Daemon对当前容器上锁，保证获取文件变更期间，文件层数据不会发生变化，代码如下：

```
func (container *Container) Changes() ([]archive.Change, error) {
    container.Lock()
    defer container.Unlock()
    return container.changes()
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

这段代码是获取到当前容器最上面的读写层所有的父类文件层。为什么要这么做？正如之前所言，我们需要通过比较所有文件层之后才能得知文件具体的变更类型，而上面代码就是获取所有的文件层。例如，我们当前的容器id是1b1580fceb452979eff90321e4e8214ec370a7bc5f4f049853807c20d555bdab，那么此文件在容器创建时，会初始化其所有的父级文件层，内容如下：

```
1b1580fceb452979eff90321e4e8214ec370a7bc5f4f049853807c20d555bdab
91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d920813e267c
d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb82
c22013c8472965aa5b62559f2b540cd440716ef149756e7b958a1b2aba421e87
d3a1f33e8a5a513092f01bb7eb1c2abf4d711e5105390a3fe1ae2248cfde1391
```

当上面代码执行完成之后，返回的layers就是上面的5条记录，而后Daemon就开始调用archive.Changes(layers, path.Join(a.rootPath(),"diff", id))来判断差异了。请继续往下看：

```
func Changes(layers []string, rw string) ([]Change, error) {
    var (
        changes     []Change
        changedDirs = make(map[string]struct{})
    )

    err := filepath.Walk(rw, func(path string, f os.FileInfo, err
        if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        // Find out what kind of modification happened
        file := filepath.Base(path)
        // If there is a whiteout, then the file was removed
        if strings.HasPrefix(file, ".wh.") {
            originalFile := file[len(".wh."):]
            change.Path = filepath.Join(filepath.Dir(path), origi
            change.Kind = ChangeDelete
        } else {
            // Otherwise, the file was added
            change.Kind = ChangeAdd

            // ...Unless it already existed in a top layer, in wh
            for _, layer := range layers {
                stat, err := os.Stat(filepath.Join(layer, path))
                if err != nil && !os.IsNotExist(err) {
                    return err
                }
                if err == nil {
                    // The file existed in the top layer, so that

                    // However, if it's a directory, maybe it was
                    // If you modify /foo/bar/baz, then /foo will
because it's the parent of bar
                    if stat.IsDir() && f.IsDir() {
                        if f.Size() == stat.Size() && f.Mode() ==
stat.ModTime()) {
                            // Both directories are the same, don
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

可以看到changes函数中最关键的逻辑就是通过调用filepath.Walk来循环判断每个文件。此时rw则是容器最上层的读写文件层。所以这段函数翻译过来就是循环判断读写文件层中出现的每个文件的变更类型，而判断标准则是本节开头所介绍的标准。

下面这段代码是用来过滤AUFS自身文件的。在AUFS文件系统中，会用.wh..wh.来标记其自身文件，包括.wh..wh.orph、.wh..wh.plnk、.wh..wh.aufs和.wh..wh..opq这四个文件。这四个文件中，.wh..wh.orph和.wh..wh.plnk是目录，后两个是文件。读者如果有兴趣了解这四个文件，可以参阅AUFS相关文档。

```
// Rebase path
        path, err = filepath.Rel(rw, path)
        if err != nil {
            return err
        }

        // As this runs on the daemon side, file paths are OS spe
        path = filepath.Join(string(os.PathSeparator), path)

        // Skip root
        if path == string(os.PathSeparator) {
            return nil
        }

        // Skip AUFS metadata
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
                return nil
            }
        }
        change.Kind = ChangeModify
        break
    }
}
```

Layers保存着所有的父级文件层，因为文件路径在所有文件层都是一致的，不会变更。因此这里只需在父级文件层中判断此文件是否存在就可以。如果都不存在，那么就是新增。如果存在，那么就判断文件属性是否发生变化；只要有一项属性发生变化，就会认为属于新增。

当所有文件都判断结束后，服务端依次返回判断结果给客户端。客户端再依次解析展现，至此diff命令执行完毕。

通过对diff命令的讲解，我们明白了在AUFS文件系统中，所有的删除都是被标记为删除，真实的文件还保存在主机文件系统中。文件发生变更之后，会在读写层保存最新的文件，而父级文件层中的文件不会发生任何变更。这也是为什么Docker需要大量硬盘空间的原因，因为每一次的变更看似是变更了，其实只是在最上层发生了变更，其他文件层还是冗余地保留了文件的备份。

7. events命令

events命令可以返回容器的创建、销毁、导出、关闭、异常、暂停等事件，所以events命令经常被用来做Docker容器的动态监控。但作为

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

件源中获取所有的事件，然后对这些事件进行筛选，最后把实时将筛选出来的信息输出到客户端。我们先来看events客户端的执行逻辑：

```
func (cli *DockerCli) CmdEvents(args ...string) error {
    cmd := cli.Subcmd("events", "", "Get real time events from th
    since := cmd.String([]string{"#since", "-since"}, "", "Show a
    until := cmd.String([]string{"-until"}, "", "Stream events un
    flFilter := opts.NewListOpts(nil)
    cmd.Var(&flFilter, []string{"f", "-filter"}, "Filter output b
    cmd.Require(flag.Exact, 0)

    cmd.ParseFlags(args, true)

    var (
        v               = url.Values{}
        eventFilterArgs = filters.Args{}
    )

    // Consolidate all filter flags, and sanity check them early.
    // They'll get process in the daemon/server.
    for _, f := range flFilter.GetAll() {
        var err error
        eventFilterArgs, err = filters.ParseFlag(f, eventFilterAr
        if err != nil {
            return err
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

点还是在server端。

我们首先在分发路由中找到events在server端对应的处理函数：

```
        "/events":                          s.getEvents
```

然后转入到getEvents函数中：在getEvents函数中，它首先解析URL中的参数，获取到客户端所传来的筛选项：

```
if err := parseForm(r); err != nil {
    return err
}
var since int64 = -1
if r.Form.Get("since") != "" {
    s, err := strconv.ParseInt(r.Form.Get("since"), 10, 64)
    if err != nil {
        return err
    }
    since = s
}
var until int64 = -1
if r.Form.Get("until") != "" {
    u, err := strconv.ParseInt(r.Form.Get("until"), 10, 64)
    if err != nil {
        return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

而parseForm最终会把URL中的参数复制到这个map集合中，而后代码就可以从r.Form中获取到相应参数值。parseForm函数主要代码如下：

```
func (r *Request) ParseForm() error {
    var err error
    if r.PostForm == nil {
        if r.Method == "POST" || r.Method == "PUT" || r.Method ==
            r.PostForm, err = parsePostForm(r)
        }
        if r.PostForm == nil {
            r.PostForm = make(url.Values)
        }
    }
    if r.Form == nil {
        if len(r.PostForm) > 0 {
            r.Form = make(url.Values)
            copyValues(r.Form, r.PostForm)
        }
        var newValues url.Values
        if r.URL != nil {
            var e error
            newValues, e = url.ParseQuery(r.URL.RawQuery)
            if err == nil {
                err = e
            }
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

Daemon所有接收到的事件。那么它又如何使用呢？请看Docker后面的代码：

```
    current, l := es.Subscribe()
```

通过这么一句简单的代码，Docker就获取到了一个当前所有事件的完整备份和一个channel。下面是Subscribe函数实现：

```
func (e *Events) Subscribe() ([]*jsonmessage.JSONMessage, chan in
    e.mu.Lock()
    current := make([]*jsonmessage.JSONMessage, len(e.events))
    copy(current, e.events)
    l := e.pub.Subscribe()
    e.mu.Unlock()
    return current, l
}
```

到目前为止，Docker已经获取到了事件源和筛选项。Docker现在可以进行筛选了，筛选代码如下：

```
    for _, ev := range current {
        if ev.Time < since {
            continue
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

别；而在sendEvent函数中，则对传入的单个事件进行合法判断。例如，是否为指定容器所发出的事件，是否为指定的事件类型，是否为指定镜像所发出的事件，等等。如果通过合法性判断，那么就会将这个事件返回给客户端。如果没有通过合法性判断，那么就开始判断下一个事件，直至所有的事件都完成判断。

而在进行合法性判断时，最关键的函数就是isFiltered函数了，下面是isFiltered函数的代码：

```
isFiltered := func(field string, filter []string) bool {
    if len(filter) == 0 {
        return false
    }
    for _, v := range filter {
        if v == field {
            return false
        }
        if strings.Contains(field, ":") {
            image := strings.Split(field, ":")
            if image[0] == v {
                return false
            }
        }
    }
    return true
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

的事件判断完全一致了。下面是处理接收事件的代码：

```
for {
    select {
    case ev := <-l:
        jev, ok := ev.(*jsonmessage.JSONMessage)
        if !ok {
            continue
        }
        if err := sendEvent(jev); err != nil {
            return err
        }
    case <-timer.C:
        return nil
    case <-closeNotify:
        logrus.Debug("Client disconnected, stop sending event
        return nil
    }
}
```

使用for循环，是为了达到死循环的效果。select是go语言中专门用来处理多个channel的语法。使用select之后，代码执行到这里就会阻塞等待channel。如果channel中产生了新消息，代码就会判断此消息属于何种类型的消息。如果读者感觉select不好理解，那么可以把select理解成一个可以阻塞的switch语法。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

如果不仔细研读代码，这个问题还真不容易解答。因为执行events命令时，是客户端在实时显示命令。因此这个问题的答案十有八九需要在客户端代码中找寻，所以让我们回到客户端代码中，重新审视一遍。

在客户端代码中，有两行不起眼的代码或许可以给我们一些提示，代码如下：

```
sopts := &streamOpts{
    rawTerminal: true,
    out:         cli.out,
}
```

sopts是steamOpts结构体的实例对象，其内部属性out被赋值为cli.out，而这个cli.out则是一个io.Writer（此结构体定义请看本节最后的附录代码）。

所以在Docker Daemon中，当需要发送事件时，其实是将数据发送到了客户端，而客户端接收到json数据之后，就将解析之后的数据直接写入到cli.out(io.Writer)中。而“实时”就更容易做到了，只要客户端和服务端的TCP链接不断，就可以一直实时接收数据。

通过上面的代码逻辑，我们使用events命令就达到了实时获取Docker事件的效果。

为了让读者可以全面地理解events的服务端逻辑，特提供如下代码：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        dur := time.Unix(until, 0).Sub(time.Now())
        timer = time.NewTimer(dur)
    }

    ef, err := filters.FromParam(r.Form.Get("filters"))
    if err != nil {
        return err
    }
    isFiltered := func(field string, filter []string) bool {
        if len(filter) == 0 {
            return false
        }
        for _, v := range filter {
            if v == field {
                return false
            }
            if strings.Contains(field, ":") {
                image := strings.Split(field, ":")
                if image[0] == v {
                    return false
                }
            }
        }
        return true
    }

    d := s.daemon
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if since == -1 {
        current = nil
    }
    defer es.Evict(l)
    for _, ev := range current {
        if ev.Time < since {
            continue
        }
        if err := sendEvent(ev); err != nil {
            return err
        }
        tools.Logger("ev=", ev)
    }
    var closeNotify <-chan bool
    if closeNotifier, ok := w.(http.CloseNotifier); ok {
        closeNotify = closeNotifier.CloseNotify()
    }
    for {
        select {
        case ev := <-l:
            jev, ok := ev.(*jsonmessage.JSONMessage)
            if !ok {
                continue
            }
            if err := sendEvent(jev); err != nil {
                return err
            }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    tlsConfig *tls.Config
    // scheme holds the scheme of the client i.e. https.
    scheme string
    // inFd holds the file descriptor of the client's STDIN (if v
    inFd uintptr
    // outFd holds file descriptor of the client's STDOUT (if val
    outFd uintptr
    // isTerminalIn indicates whether the client's STDIN is a TTY
    isTerminalIn bool
    // isTerminalOut dindicates whether the client's STDOUT is a
    isTerminalOut bool
    // transport holds the client transport instance.
    transport *http.Transport
}
```

8. exec命令

下面讲解exec命令是如何实现的。在第5章中，我们简要介绍了exec命令的用法，通过exec命令，用户可以在Docker容器中执行任意的命令，并且获取到相应结果。这样的效果是通过什么技术实现的呢？下面我们就看一下exec命令的实现奥秘。

exec命令的实现逻辑与其他命令并无差异，首先是客户端负责收集命令行的执行参数，然后将此参数通过Rest API的方式发送到Docker Daemon，而后服务端解析参数并且执行特定的逻辑。因此exec命令的实现重点在服务端，客户端仅负责收集参数并且调用服务端API。但先易后难，我们先看一下客户端逻辑。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    )
    cmd.Require(flag.Min, 2)
    if err := cmd.ParseFlags(args, true); err != nil {
        return nil, err
    }
    container = cmd.Arg(0)
    parsedArgs := cmd.Args()
    execCmd = parsedArgs[1:]

    execConfig := &ExecConfig{
        User: *flUser,
        // TODO(vishh): Expose 'Privileged' once it is supported
        // +        //Privileged:   job.GetenvBool("Privileged"),
        Tty:       *flTty,
        Cmd:       execCmd,
        Container: container,
        Detach:    *flDetach,
    }

    // If -d is not set, attach to everything by default
    if !*flDetach {
        execConfig.AttachStdout = true
        execConfig.AttachStderr = true
        if *flStdin {
            execConfig.AttachStdin = true
        }
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err != nil {
        return err
    }
```

从API名称上面，我们可以推测出此API就是在某一个容器（execConfig.Container）中执行命令，执行命令所需的具体信息则通过execConfig来获取。按照此逻辑，后面就应该是服务端的工作。我们看看服务端又是如何处理此请求的，首先找到服务端此API所对应的处理函数：

```
            "/containers/{name:.*}/exec":    s.postContainerExec
```

然后进入到postContainerExecCreate函数当中：

```
func  (s  *Server)  postContainerExecCreate(version  version.Vers
*http.Request, vars map[string]string) error {
    if err := parseForm(r); err != nil {
        return err
    }
    name := vars["name"]

    execConfig := &runconfig.ExecConfig{}
    if err := json.NewDecoder(r.Body).Decode(execConfig); err !=
        return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

经提到过execConfig对象包含了执行命令相关信息，所以服务端要想执行命令，必然要获取此对象。而客户端是通过JSON的形式将execConfig对象发送到Docker Daemon的。所以在服务端，Docker Daemon通过json.NewDecoder(r.Body).Decode(execConfig)将JSON字符串重新解析为execConfig对象。

这些工作都做完之后，Docker Daemon在postContainerExecCreate中做了最后一件事情：调用ContainerExecCreate函数来创建此命令的执行实例。

Docker为什么要这么做呢？我们先按耐住这份好奇心，慢慢来看。下面是ContainerExecCreate函数的实现代码：

```
func (d *Daemon) ContainerExecCreate(config *runconfig.ExecConfig
    // Not all drivers support Exec (LXC for example)
    if err := checkExecSupport(d.execDriver.Name()); err != nil {
        return "", err
    }

    container, err := d.getActiveContainer(config.Container)
    if err != nil {
        return "", err
    }

    cmd := runconfig.NewCommand(config.Cmd...)
    entrypoint, args := d.getEntrypointAndArgs(runconfig.NewEntry
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return execConfig.ID, nil

}
```

ContainerExecCreate函数非常有意思，在函数开始时，首先判断虚拟化驱动是否支持在容器中运行命令，并且在注释中也明确指出了LXC不支持exec命令。目前Docker的虚拟化驱动只有事实上的两类：LXC和native。LXC是Linux内核版本所提供的容器虚拟化驱动，而native则是Docker为了达到跨平台的目的，由Docker官方所推出的一套虚拟化驱动标准。按照Docker的计划，只要实现了native驱动中所定义的接口，那么就一定可以运行Docker以达到跨平台的目标。所以建议读者在启动Docker时，虚拟化驱动选用native。

言归正传，如果启动Docker Daemon时选用了native驱动，那么此项检查就会通过，然后Docker Daemon会获取到目标容器的entrypoint。为什么要获取entrypoint呢？在前面的章节中我们说过，如果在镜像或者容器中设定了entrypoint，那么以后所有的命令或者参数都将变为entrypoint的参数。

所以当需要在容器中运行命令时，一定要判断此容器有没有设定entrypoint。而后Docker Daemon就开始封装ProcessConfig对象了。在ProcessConfig对象中，可以看到此时已经没有了客户端所传入的命令信息，取而代之的则是entrypoint。

```
    processConfig := execdriver.ProcessConfig{
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        args = cmdSlice[1:]
    }
    return entrypoint, args
}
```

读者看到这里，想必会加深对entrypoint指令的理解了吧。我们继续对exec服务端代码的解析，当Daemon封装完ProcessConfig对象之后，又封装了execConfig对象。这个execConfig对象虽然名称上和客户端代码中的execConfig对象很相似，但却是完全不同的对象。此时的execConfig对象包含了一个执行所需的所有信息的命令，对象定义如下：

```
type execConfig struct {
    sync.Mutex
    ID            string
    Running       bool
    ExitCode      int
    ProcessConfig execdriver.ProcessConfig
    StreamConfig
    OpenStdin  bool
    OpenStderr bool
    OpenStdout bool
    Container  *Container
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

读者会好奇，执行队列是一个什么样的东西，有何作用呢？其实，执行队列只是为了解释方便而称之为队列，其本质是一个map集合：

```
type execStore struct {
    s map[string]*execConfig
    sync.RWMutex
}
```

这个队列（map集合）的key值是执行实例的ID值，而value则是执行实例的执行配置信息。当需要执行时，只需通过ID获取到配置信息就可以随时执行这个实例了。

看到这里，应该说大部分读者都有一个疑问，为什么需要保存这个执行实例呢？既然都创建了执行实例，直接执行不就可以了吗？Docker这么做，是为了方便remote API调用。当看完后面如何执行此执行实例，就可以找到此问题的答案了。

Docker Daemon注册完执行实例之后，就将执行实例的exec ID返回给了客户端，如下所示：

```
            stream, _, err := cli.call("POST", "/containers
execConfig, nil)
    if err != nil {
        return err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        // For now don't print this - wait for when we support ex
        // fmt.Fprintf(cli.out, "%s\n", execID)
        return nil
    }
```

如果在执行exec命令时没有使用其他参数，那么execConfig.Detach将等于false。客户端会通过执行CheckTtyInput函数来检查客户端是否满足输出条件，判断如下：

```
func (cli *DockerCli) CheckTtyInput(attachStdin, ttyMode bool) er
    if ttyMode && attachStdin && !cli.isTerminalIn {
        return errors.New("cannot enable tty mode on non tty inpu
    }
    return nil
}
```

如果检查没有错误之后，客户端就会调用start API：

```
    if execConfig.AttachStdin {
        in = cli.in
    }
    if execConfig.AttachStdout {
        out = cli.out
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

负责启动执行实例的函数是postContainerExecStart，它是如何处理的：

```
func  (s  *Server)  postContainerExecStart(version  version.Versi
*http.Request, vars map[string]string) error {
    if err := parseForm(r); err != nil {
        return err
    }
    var (
        execName = vars["name"]
        stdin    io.ReadCloser
        stdout   io.Writer
        stderr   io.Writer
    )

    execStartCheck := &types.ExecStartCheck{}
    if err := json.NewDecoder(r.Body).Decode(execStartCheck); err
        return err
    }

    if !execStartCheck.Detach {
        // Setting up the streaming http interface.
        inStream, outStream, err := hijackServer(w)
        if err != nil {
            return err
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    w.WriteHeader(http.StatusNoContent)

    return nil
}
```

在postContainerExecStart函数中，我们首先看到Daemon获取了执行实例ID。因为注册执行实例时，此ID是Key，所以需要执行它时必然要先获取到此Key，而后看到：

```
    if !execStartCheck.Detach {
        // Setting up the streaming http interface.
        inStream, outStream, err := hijackServer(w)
        if err != nil {
            return err
        }
        defer closeStreams(inStream, outStream)
        var errStream io.Writer

        if _, ok := r.Header["Upgrade"]; ok {
            fmt.Fprintf(outStream, "HTTP/1.1 101 UPGRADED\r\nCont
raw-stream\r\nConnection: Upgrade\r\nUpgrade: tcp\r\n\r\n")
        } else {
            fmt.Fprintf(outStream, "HTTP/1.1 200 OK\r\nContent-Ty
stream\r\n\r\n")
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

首先获取execConfig，上面代码中的execName其实就是执行实例ID。然后，执行如下代码：

```
func() {
    execConfig.Lock()
    defer execConfig.Unlock()
    if execConfig.Running {
        err = fmt.Errorf("Error: Exec command %s is already r
    }
    execConfig.Running = true
}()
if err != nil {
    return err
}
```

将此执行实例置为running状态，防止同时并发执行。再后，执行如下代码：

```
if execConfig.OpenStdin {
    r, w := io.Pipe()
    go func() {
        defer w.Close()
        defer logrus.Debugf("Closing buffered stdin pipe")
        io.Copy(w, stdin)
    }()
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

做完这些铺垫之后，就到了真正执行的时刻：

```
    go func() {
        if err := container.Exec(execConfig); err != nil {
            execErr <- fmt.Errorf("Cannot run exec command %s in
container.ID, err)
        }
    }()
```

通过go并发去启动执行实例。为什么需要并发执行呢？因为此处的代码仍然是由Daemon在执行，如果执行实例需要很长时间才能执行完毕，那么Daemon阻塞在此处不是一个明智的选择，所以选择了并发执行。

下面再看一下container.Exec函数。

```
func (container *Container) Exec(execConfig *execConfig) error {
    container.Lock()
    defer container.Unlock()

    waitStart := make(chan struct{})

    callback := func(processConfig *execdriver.ProcessConfig, pi
        if processConfig.Tty {
            // The callback is called after the process Start()
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

container.monitorExec的执行结果。

```
func (container *Container) monitorExec(execConfig *execConfig, c
error {
    var (
        err      error
        exitCode int
    )

    pipes  :=  execdriver.NewPipes(execConfig.StreamConfig.stdin,
execConfig.StreamConfig.stderr, execConfig.OpenStdin)
    exitCode, err = container.daemon.Exec(container, execConfig,
    if err != nil {
        logrus.Errorf("Error running command in existing containe
    }

    logrus.Debugf("Exec task in container %s exited with code %d"
    if execConfig.OpenStdin {
        if err := execConfig.StreamConfig.stdin.Close(); err != n
            logrus.Errorf("Error closing stdin while running in %
        }
    }
    if err := execConfig.StreamConfig.stdout.Clean(); err != nil
        logrus.Errorf("Error closing stdout while running in %s:
    }
    if err := execConfig.StreamConfig.stderr.Clean(); err != nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

令，代码如下：

```
func (d *driver) Exec(c *execdriver.Command, processConfig
*execdriver.Pipes, startCallback execdriver.StartCallback) (int,
    active := d.activeContainers[c.ID]
    if active == nil {
        return -1, fmt.Errorf("No active container exists with ID
    }

    p := &libcontainer.Process{
        Args: append([]string{processConfig.Entrypoint}, processC
        Env:  c.ProcessConfig.Env,
        Cwd:  c.WorkingDir,
        User: processConfig.User,
    }

    config := active.Config()
    if err := setupPipes(&config, processConfig, p, pipes); err
        return -1, err
    }

    if err := active.Start(p); err != nil {
        return -1, err
    }

    if startCallback != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

行完毕之后，返回执行结果。至此，一条命令就在容器中执行完毕了。

当我们在客户端命令行输入一条Docker exec时，感觉很快就返回了结果，没想到后面却有如此多的操作。通过对exec命令的解析，可以初步体会到Docker是如何管理容器的，又是如何从容器中获取数据的。

在本节的最后，我们提供了exec客户端的代码和hijack的代码供读者参阅。

```
func (cli *DockerCli) CmdExec(args ...string) error {
    cmd := cli.Subcmd("exec", "CONTAINER COMMAND [ARG...]", "Run
true)

    execConfig, err := runconfig.ParseExec(cmd, args)
    // just in case the ParseExec does not exit
    if execConfig.Container == "" || err != nil {
        return StatusError{StatusCode: 1}
    }

    stream, _, err := cli.call("POST", "/containers/"+execConfig
    if err != nil {
        return err
    }

    var response types.ContainerExecCreateResponse
    if err := json.NewDecoder(stream).Decode(&response); err != n
        return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    // Interactive exec requested.
    var (
        out, stderr io.Writer
        in          io.ReadCloser
        hijacked    = make(chan io.Closer)
        errCh       chan error
    )

// Block the return until the chan gets closed
    defer func() {
        logrus.Debugf("End of CmdExec(), Waiting for hijack to fi
        if _, ok := <-hijacked; ok {
            fmt.Fprintln(cli.err, "Hijack did not finish (chan st
        }
    }()

    if execConfig.AttachStdin {
        in = cli.in
    }
    if execConfig.AttachStdout {
        out = cli.out
    }
    if execConfig.AttachStderr {
        if execConfig.Tty {
            stderr = cli.out
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    if err := <-errCh; err != nil {
        logrus.Debugf("Error hijack: %s", err)
        return err
    }
    var status int
    if _, status, err = getExecExitCode(cli, execID); err != nil
        return err
    }

    if status != 0 {
        return StatusError{StatusCode: status}
    }
    return nil
}

func (cli *DockerCli) hijack(method, path string, setRawTerminal
stderr io.Writer, started chan io.Closer, data interface{}) erro
    defer func() {
        if started != nil {
            close(started)
        }
    }()

    params, err := cli.encodeData(data)
    if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        tcpConn.SetKeepAlivePeriod(30 * time.Second)
    }
    if err != nil {
        if strings.Contains(err.Error(), "connection refused") {
            return fmt.Errorf("Cannot connect to the Docker daemo
        }
        return err
    }
    clientconn := httputil.NewClientConn(dial, nil)
    defer clientconn.Close()

    // Server hijacks the connection, error 'connection closed' e
    clientconn.Do(req)
    rwc, br := clientconn.Hijack()
    defer rwc.Close()

    if started != nil {
        started <- rwc
    }

    var receiveStdout chan error
    var oldState *term.State
    if in != nil && setRawTerminal && cli.isTerminalIn && os.Gete
        oldState, err = term.SetRawTerminal(cli.inFd)
        if err != nil {return err}
        defer term.RestoreTerminal(cli.inFd, oldState)
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            logrus.Debugf("[hijack] End of stdin")
        }
        if conn, ok := rwc.(interface {
            CloseWrite() error
        }); ok {
            if err := conn.CloseWrite(); err != nil {
                logrus.Debugf("Couldn't send EOF: %s", err)
            }
        }
        return nil
    })
    if stdout != nil || stderr != nil {
        if err := <-receiveStdout; err != nil {
            logrus.Debugf("Error receiveStdout: %s", err)
            return err
        }
    }
    if !cli.isTerminalIn {
        if err := <-sendStdin; err != nil {
            logrus.Debugf("Error sendStdin: %s", err)
            return err
        }
    }
    return nil
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
535020c3e8ad   9 weeks ago /bin/sh -c #(nop) COPY file:4abd3bff6(
```

首先在Docker中执行一个docker history hello-world命令，看一下执行效果。

通过执行的history命令可以看出，history输出了指定镜像在build过程中执行的所有指令。按照常理分析，既然Docker可以获取到一个镜像所有的构建执行指令，那么不外乎：

①所有的构建执行指令都保存在本地，Docker从本地某地获取。

②所有的构建执行指令保存在镜像中，然后Docker从镜像中获取。

这两个获取途径，我们分析一下哪种有可能是Docker所采用的。第一种方法中，如果镜像所有的构建指令通过某种方式都保存在本地，如保存在SQLite中，那么对于在本地执行构建的镜像而言，这个是可行的。但对于由第三方所构建的镜像而言，这个方案就很不现实了。所以此种方法，我们可以忽略。

而第二种方法是所有的执行指令保存在镜像中。如果这样做的话，那么无论是本地构建还是异地构建，只要构建指令成功保存在镜像中，那么这些信息就可以伴随着镜像一起分发。这样只要下载到的镜像数据是完整的，那么Docker就可以从镜像中读取到这些信息。

第一种方法已经排除了，那么我们按照第二种方法的思路来分析一下history的逻辑。首先我们看一下history的客户端代码：

```
func (cli *DockerCli) CmdHistory(args ...string) error {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        }
        if !*quiet {
            if *human {
                fmt.Fprintf(w,  "\t%s  ago\t",  units.HumanDurati
(entry.Created, 0))))
            } else {
                fmt.Fprintf(w, "\t%s\t", time.Unix(entry.Created,
            }

            if *noTrunc {
                fmt.Fprintf(w, "%s\t", entry.CreatedBy)
            } else {
                fmt.Fprintf(w, "%s\t", stringutils.Truncate(entry
            }

            if *human {
                fmt.Fprintf(w, "%s\t", units.HumanSize(float64(en
            } else {
                fmt.Fprintf(w, "%d\t", entry.Size)
            }

            fmt.Fprintf(w, "%s", entry.Comment)
        }
        fmt.Fprintf(w, "\n")
    }
    w.Flush()
    return nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return writeJSON(w, http.StatusOK, history)
}
```

从上面代码可以看出，getImagesHistory函数其实主要调用的是s.daemon.Repositories().History函数，只不过在getImagesHistory函数中解析了镜像名称，因此我们下一步的工作自然而然地需要跟进到s.daemon.Repositories().History(name)中。

```
func (s *TagStore) History(name string) ([]*types.ImageHistory, e
    foundImage, err := s.LookupImage(name)
    if err != nil {
        return nil, err
    }

    lookupMap := make(map[string][]string)
    for name, repository := range s.Repositories {
        for tag, id := range repository {
            // If the ID already has a reverse lookup, do not upd
            if _, exists := lookupMap[id]; !exists {
                lookupMap[id] = []string{}
            }
            lookupMap[id] = append(lookupMap[id], utils.ImageRefe
        }
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    lookupMap := make(map[string][]string)
    for name, repository := range s.Repositories {
        for tag, id := range repository {
            if _, exists := lookupMap[id]; !exists {
                lookupMap[id] = []string{}
            }
            lookupMap[id] = append(lookupMap[id], utils.ImageRefe
        }
    }
```

lookupMap则保存着当前Docker环境中所有镜像的tag标签信息，保存数据如下：

假设当前环境中存在的镜像数据如下：

```
REPOSITORY        TAG                 IMAGE ID            CREATED
docker-dev        latest              a045e605f194        3 minut
hello-world       latest              af340544ed62        9 weeks
docker-dev        1.7.1               a7d1af33a73b        3 month
```

则lookupMap保存的数据如下：

```
map[
    afd3532992bf79eddb5457df1396bbf7165864f6abc6c24896e39c117d0b2
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

要想理解透这段代码，我们就需要深入到foundImage.WalkHistory函数中，因此请出foundImage.WalkHistory的源码：

```
func (img *Image) WalkHistory(handler func(*Image) error) (err er
    currentImg := img
    for currentImg != nil {
        if handler != nil {
            if err := handler(currentImg); err != nil {
                return err
            }
        }
        currentImg, err = currentImg.GetParent()
        if err != nil {
            return fmt.Errorf("Error while getting parent image:
        }
    }
    return nil
}
```

在WalkHistory函数中，currentImg就是调用WalkHistory的实例本身。如果读者对Golang不熟悉，可能感觉很困惑，没有类似于Java的对象参数，这里的img是从哪里来的？其实这就是Golang的语言特点，读者可以参阅一下Golang的语法简介资料，了解相关知识点。这里只需记住，currentImg现在就是外层的foundImage对象。

如果currentImg和handler都不为空，则将会把currentImg作为参数放

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

父镜像的构建信息（一个镜像只能是由一条构建指令形成），最后将获取到的构建信息存储起来返回给客户端。

通过上面的流程，我们就获取到了指定镜像的构建指令。但对History命令的分析并没有结束，因为还有三个问题没有搞清楚：

①构建信息是何时保存的？

②构建信息保存在什么地方？

③History命令是从哪里，通过什么方式获取到构建指令的？

前两个问题是在build命令和commit命令中有意的忽略的细节问题，下面依次解答。

首先构建信息是何时保存的？

按照常理分析，应该是在build命令执行阶段保存的。因为build就是用来构建镜像的命令，build命令可以获取到所有的构建指令，因此放在此环节合情合理。下面是build命令中用来执行具体指令的代码：

```
for i, n := range b.dockerfile.Children {
    select {
    case <-b.cancelled:
        logrus.Debug("Builder: build cancelled!")
        fmt.Fprintf(b.OutStream, "Build cancelled")
        return "", fmt.Errorf("Build cancelled")
    default:
        // Not cancelled yet, keep going...
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            }
        }
        strList[i+l] = str
        msgList[i] = ast.Value
        i++
    }
```

strList数组保存着当前指令的具体执行信息。例如，在Dockerfile中如果定义了一句WORKDIR/tmp，那么strList保存的则是/tmp，而后就会通过一个map来查找具体的执行函数（读者可以参阅build一节获取相关资料）：

```
    if f, ok := evaluateTable[cmd]; ok {
        b.BuilderFlags = NewBuilderFlags()
        b.BuilderFlags.Args = flags
        return f(b, strList, attrs, original)
    }
```

可以看到strList已经作为参数传入到f函数中。我们仍然以workdir指令为例，继续下面的分析。在workdir指令函数中，其函数签名为：

```
func workdir(b *Builder, args []string, attributes map[string]boo
```

其实不只workdir的函数签名是这样的，所有的指令函数签名都是

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        if err != nil {
            return err
        }
        if hit {
            return nil
        }

        container, err := b.create()
        if err != nil {
            return err
        }
        id = container.ID

        if err := container.Mount(); err != nil {
            return err
        }
        defer container.Unmount()
    }
    container, err := b.Daemon.Get(id)
    if err != nil {
        return err
    }

    // Note: Actually copy the struct
    autoConfig := *b.Config
    autoConfig.Cmd = autoCmd
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

下面是b.Daemon.Commit的代码：

```
func (daemon *Daemon) Commit(container *Container, repository, ta
bool, config *runconfig.Config) (*image.Image, error) {
    if pause && !container.IsPaused() {
        container.Pause()
        defer container.Unpause()
    }

    if err := container.Mount(); err != nil {
        return nil, err
    }
    defer container.Unmount()

    rwTar, err := container.ExportRw()
    if err != nil {
        return nil, err
    }
    defer func() {
        if rwTar != nil {
            rwTar.Close()
        }
    }()

    // Create a new image from the container's base layers + a ne
    var (
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

containerID，parentImageID，comment，author，containerConfig，config）这句。

```
func (graph *Graph) Create(layerData archive.ArchiveReader, conta
author string, containerConfig, config *runconfig.Config) (*image
    img := &image.Image{
        ID:            stringid.GenerateRandomID(),
        Comment:       comment,
        Created:       time.Now().UTC(),
        DockerVersion: dockerversion.VERSION,
        Author:        author,
        Config:        config,
        Architecture:  runtime.GOARCH,
        OS:            runtime.GOOS,
    }

    if containerID != "" {
        img.Parent = containerImage
        img.Container = containerID
        img.ContainerConfig = *containerConfig
    }

    if err := graph.Register(img, layerData); err != nil {
        return nil, err
    }
    return img, nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err := os.RemoveAll(graph.ImageRoot(img.ID)); err != nil &
        return err
    }

    graph.driver.Remove(img.ID)

    tmp, err := graph.Mktemp("")
    defer os.RemoveAll(tmp)
    if err != nil {
        return fmt.Errorf("Mktemp failed: %s", err)
    }

    if err := graph.driver.Create(img.ID, img.Parent); err != nil
        return fmt.Errorf("Driver %s failed to create image rootf
    }

    img.SetGraph(graph)
    if err := image.StoreImage(img, layerData, tmp); err != nil {
        return err
    }

    if err := os.Rename(tmp, graph.ImageRoot(img.ID)); err != nil
        return err
    }
    graph.idIndex.Add(img.ID)
    return nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    defer f.Close()
    return json.NewEncoder(f).Encode(img)
}
```

在StoreImage函数中，Docker创建了一个名为JSON的文件。细心的读者会问：没看到JSON文件的定义呀？这个定义JSON文件的代码是jsonPath：

```
func jsonPath(root string) string {
    return filepath.Join(root, "json")
}
```

此时的JSON文件是在一个临时目录中，待都创建成功之后，Docker会将此临时目录正式命名为此镜像ID。然后将镜像的所有元数据都写入到此JSON文件中。都执行完之后，JSON文件的数据将会是下面的样子：

```
{
    "id": "fd1ed22e7edf1168e3c0acf8ed3e46789f6cc4df6b6ebdd84cff42
    "parent": "216c4dc07ff24e7ab6aa29ba3a94d7e60182ebe7295c77762f
    "created": "2015-07-14T03:00:42.55169846Z",
    "container": "5d6a992c1a53314945108cb060dca4060f90956cbc753b4
    "container_config": {
        "Hostname": "dd360632d03c",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        "NetworkDisabled": false,
        "MacAddress": "",
        "OnBuild": [],
        "Labels": {}
    },
    "docker_version": "1.7.0",
    "author": "Tianon Gravi <admwiggin@gmail.com> (@tianon)",
    "config": {
        "Hostname": "dd360632d03c",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "ExposedPorts": null,
        "PublishService": "",
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": [
            "PATH=/go/bin:/usr/local/go/bin:/usr/local/sbin:/usr/
sbin:/bin",
            "LXC_VERSION=1.0.7",
            "GO_VERSION=1.4.2"
        ],
        "Cmd": [
            "/bin/bash"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

行构建指令，最后再将此容器保存成一个镜像文件层。因此，可以将build命令理解成通过多次针对容器调用commit命令来生成的镜像。

还剩最后一个问题：history命令是从哪里、通过什么方式获取到构建指令的？这个问题现在非常容易回答了。History获取到镜像的父文件层信息，然后迭代读取每个文件层的JSON文件数据，输出JSON文件中的ContainerConfig.Cmd.Slice()。

至此，我们真正的将History服务端代码解析完毕。

回到History客户端代码，还剩下最后一部分代码：将接收到的JSON数据输出到终端。首先通过tabwriter.NewWriter(cli.out, 20, 1, 3, ' ', 0)定制了输出的样式，其中20表示每列最小宽度，1表示\t所占的宽度，3表示最多补齐3个宽度，'　'表示使用空格补齐，而0表示不启用格式控制。后面就简单了，使用for循环将history数组的元素都输出出来。此时也可以看到并没有tag的位置，所以说那个获取tag的代码有些多余。

通过对History命令的解析，我们看到了Docker如何镜像以及如何对文件层进行处理。同时还看到了Docker源码还有很多值得优化的地方。History命令虽然用起来简单，但其背后的流程却不简单，从底层到上层几乎都涉及了。

读者朋友如果对本节中的知识点存在疑问，可以结合build命令和commit命令两节一起阅读。

10. info命令

info用来输出Docker目前的环境信息，此命令的性质和help命令相似，都是用来辅助用户了解当前Docker环境的。但从代码实现层面而

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            fmt.Fprintf(cli.out, " %s: %s\n", pair[0], pair[1])
        }
    }
    ioutils.FprintfIfNotEmpty(cli.out, "Execution Driver: %s\n",
    ioutils.FprintfIfNotEmpty(cli.out, "Logging Driver: %s\n", in
    ioutils.FprintfIfNotEmpty(cli.out, "Kernel Version: %s\n", in
    ioutils.FprintfIfNotEmpty(cli.out, "Operating System: %s\n",
    fmt.Fprintf(cli.out, "CPUs: %d\n", info.NCPU)
    fmt.Fprintf(cli.out, "Total Memory: %s\n", units.BytesSize(fl
    ioutils.FprintfIfNotEmpty(cli.out, "Name: %s\n", info.Name)
    ioutils.FprintfIfNotEmpty(cli.out, "ID: %s\n", info.ID)

    if info.Debug {
        fmt.Fprintf(cli.out, "Debug mode (server): %v\n", info.De
        fmt.Fprintf(cli.out, "File Descriptors: %d\n", info.NFd)
        fmt.Fprintf(cli.out, "Goroutines: %d\n", info.NGoroutines
        fmt.Fprintf(cli.out, "System Time: %s\n", info.SystemTime
        fmt.Fprintf(cli.out, "EventsListeners: %d\n", info.NEvent
        fmt.Fprintf(cli.out, "Init SHA1: %s\n", info.InitSha1)
        fmt.Fprintf(cli.out, "Init Path: %s\n", info.InitPath)
        fmt.Fprintf(cli.out, "Docker Root Dir: %s\n", info.Docker
    }

    ioutils.FprintfIfNotEmpty(cli.out, "Http Proxy: %s\n", info.H
    ioutils.FprintfIfNotEmpty(cli.out, "Https Proxy: %s\n", info.
    ioutils.FprintfIfNotEmpty(cli.out, "No Proxy: %s\n", info.NoP
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return nil
}
```

info客户端调用服务端提供的API，然后解析服务端返回的JSON字符串。此字符串包含着Docker服务端所保存的系统参数信息，info客户端在依次读取JSON中的各项属性之后输出到终端。

因此从逻辑上来分析，info命令属于非常简单的命令。我们看一下info服务端代码是如何读取Docker系统信息的。首先看一下info服务端处理函数：

```
func (s *Server) getInfo(version version.Version, w http.Response
map[string]string) error {
    info, err := s.daemon.SystemInfo()
    if err != nil {
        return err
    }

    return writeJSON(w, http.StatusOK, info)
}
```

在getInfo函数中，只调用了一个s.daemon.SystemInfo，然后将此函数返回的数据封装成json数据返回给客户端。因此info命令所有的奥秘就都在s.daemon.SystemInfo函数中。下面看一下s.daemon.SystemInfo函

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            operatingSystem += " (containerized)"
        }
    }

    meminfo, err := system.ReadMemInfo()
    if err != nil {
        logrus.Errorf("Could not read system memory info: %v", er
    }

    // if we still have the original dockerinit binary from befor
the path to that, since that's more intuitive (the copied path is
VERSION)
    initPath := utils.DockerInitPath("")
    if initPath == "" {
        // if that fails, we'll just return the path from the dae
        initPath = daemon.SystemInitPath()
    }

    v := &types.Info{
        ID:                 daemon.ID,
        Containers:         len(daemon.List()),
        Images:             imgcount,
        Driver:             daemon.GraphDriver().String(),
        DriverStatus:       daemon.GraphDriver().Status(),
        MemoryLimit:        daemon.SystemConfig().MemoryLimit,
        SwapLimit:          daemon.SystemConfig().SwapLimit,
        CpuCfsPeriod:       daemon.SystemConfig().CpuCfsPeriod,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if httpsProxy := os.Getenv("https_proxy"); httpsProxy != "" {
        v.HttpsProxy = httpsProxy
    }
    if noProxy := os.Getenv("no_proxy"); noProxy != "" {
        v.NoProxy = noProxy
    }
    if hostname, err := os.Hostname(); err == nil {
        v.Name = hostname
    }

    return v, nil
}
```

在SystemInfo函数中，所有的操作都围绕着types.Info这个结构体展开。Docker通过s.daemon.Graph().Map()获取到了当前Docker环境中已经存在的镜像集合。

s.daemon.Graph().Map()函数在之前已经分析过了，这个函数会通过walkAll函数来查找所有的镜像。因此通过获取s.daemon.Graph().Map()返回的集合元素个数就可以得知目前所有的镜像个数。需要注意的是，s.daemon.Graph().Map()中将每个文件层都视为一个单独的镜像，因而其返回的镜像数量层较之镜像文件层数量更为准确。例如，hello-world镜像可能包含5个文件层，虽然我们认为hello-world只是一个镜像，但通过s.daemon.Graph().Map()函数返回的镜像数量却是5。因此读者在处理统计数据时，需要多加留意。

而后连续调用了kernel.GetKernelVersion()和

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

上面代码中的系统调用发生在uname函数中：

```
func uname() (*syscall.Utsname, error) {
    uts := &syscall.Utsname{}

    if err := syscall.Uname(uts); err != nil {
        return nil, err
    }
    return uts, nil
}
```

而最终调用返回的数据同调用uname-a返回的数据一致，这里也是通过系统调用获取数据。当kernel.GetKernelVersion()执行完成后，返回的数据为：Kernel Version:4.0.9-boot2docker（假设Docker运行在Boot2docker之中）。

获取操作系统版本号则没有通过系统调用的方式，而是直接读取/etc/os-release文件，代码如下：

```
var (
    // file to use to detect if the daemon is running in a conta
    proc1Cgroup = "/proc/1/cgroup"
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    defer file.Close()
    return parseMemInfo(file)
}
```

而后依此通过读取Docker本身的配置文件信息、读取操作系统配置文件信息和调用系统命令的方式，获取types.Info结构体中的所有信息，并且将获取到的信息都封装给了types.Info结构体，最后将此结构体返回给Docker Daemon。

Docker Daemon接收到s.Daemon.SystemInfo()返回的JSON字符串后，没有任何其他多余的操作，直接将此JSON字符串返回给了客户端。客户端代码在接收到此JSON字符串后，对此进行解析，然后逐个获取属性并展现出来。

Info命令实现非常简单，我们在本节的代码分析中需要熟悉一下Docker各项参数的保存位置。在后面的章节中，我们会介绍Docker是在何时将这些参数写入到文件中的，以及这些参数除了展现之外还有什么其他的作用。

11. inspect命令

在Docker基本命令操作章节中，我们已经知道inspect命令用来显示指定镜像或者指定容器的底层信息。本节当中，我们会详细分析inspect是如何获取底层信息以及如何进行信息筛选的。

从前面history和info命令的代码分析中，应该可以看出来：这些获取信息类的命令大部分都是从一个或者多个文件中读取到信息，然后封装成JSON字符串返回给客户端。那么inspect是不是也是采用这样的做

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    for _, name := range cmd.Args() {
        obj, _, err := readBody(cli.call("GET", "/containers/"+na
        if err != nil {
            obj, _, err = readBody(cli.call("GET", "/images/"+nam
            isImage = true
            if err != nil {
                if strings.Contains(err.Error(), "No such") {
                    fmt.Fprintf(cli.err, "Error: No such image or
                } else {
                    fmt.Fprintf(cli.err, "%s", err)
                }
                status = 1
                continue
            }
        }

        if tmpl == nil {
            if err = json.Indent(indented, obj, "", "    "); err
                fmt.Fprintf(cli.err, "%s\n", err)
                status = 1
                continue
            } } else {
            rdr := bytes.NewReader(obj)
            dec := json.NewDecoder(rdr)
            if isImage {
                inspPtr := types.ImageInspect{}
                if err := dec.Decode(&inspPtr); err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
                        return err
                    }
                }
            }
            cli.out.Write([]byte{'\n'})
        }
        indented.WriteString(",")
    }
    if indented.Len() > 1 {
        indented.Truncate(indented.Len() - 1)
    }
    indented.WriteString(“]\n")
    if tmpl == nil {
        if _, err := io.Copy(cli.out, indented); err != nil {
            return err
        }
    }
    if status != 0 {
        return StatusError{StatusCode: status}
    }
    return nil
}
```

inspect的客户端代码有些长，但逻辑层次还是比较明显的。inspect的客户端逻辑为：循环获取用户指定的镜像或者容器底层信息，然后逐个输出属性值。这里我们假设用户只输入了一个镜像名称来看看其背后

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return writeJSON(w, http.StatusOK, containerJSON)
}
```

和其他服务端API不同的是，在/containers/{name:.*}/json中首次出现了版本比较逻辑。如果客户端使用的Remote API版本小于1.19，则Docker Daemon会使用ContainerInspectRaw来为客户端封装数据；如果使用1.19，那么就会使用ContainerInspect来为客户端封装数据。

我们看一下这两个函数的区别，然后再来说说为什么这里会出现一个版本判断。

```
func (daemon *Daemon) ContainerInspect(name string) (*types.Conta
    container, err := daemon.Get(name)
    if err != nil {
        return nil, err
    }

    container.Lock()
    defer container.Unlock()

    base, err := daemon.getInspectData(container)
    if err != nil {
        return nil, err
    }

    return &types.ContainerJSON{base, container.Config}, nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

这两个函数从逻辑上没有大的区别，唯一的区别就是ContainerInspectRaw多封装了一个types.ContainerConfig对象，同时，ContainerInspectRaw返回的是types.ContainerJSONRaw对象，而ContainerInspect返回的则是types.ContainerJSON对象。

将这两个对象对比可以发现，types.ContainerJSON包含的数据更为丰富。

```
type ContainerJSON struct {
    *ContainerJSONBase
    Config *runconfig.Config
}
type ContainerJSONRaw struct {
    *ContainerJSONBase
    Config *ContainerConfig
}

type ContainerConfig struct {
    *runconfig.Config

    // backward compatibility, they now live in HostConfig
    Memory     int64
    MemorySwap int64
    CpuShares  int64
    Cpuset     string
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

本开始，则提供了大量的数据。Docker此处判断版本是为了兼容旧版本。但笔者认为完全没有这个必要，因为如果要兼容旧版本，那么应该所有的API都应该兼容旧版本而不是某几个API兼容。而且Inspect不属于使用频率非常高的API，因此兼容它没有太大的实际意义。因此此处判断版本的代码完全可以删掉。

假设目前使用的客户端版本是1.19，则会通过ContainerInspect来获取容器信息。关于daemon.Get(name)这个函数，我们很熟悉了。此函数可以通过容器ID或者容器名称来获取到对应容器对象，然后通过daemon.getInspectData(container)来获取到此时的容器信息。Docker此时既然已经获取到了容器对象信息，而且在容器对象中也存在config这一属性，那么为什么不直接返回这个属性，而是要再使用daemon.getInspectData(container)来返回另外一个对象呢？

这是因为从Docker 1.19开始，inspect命令想尽可能的返回更多的底层信息，而不仅仅满足于container.config这一部分数据，所以在daemon.getInspectData(container)中只将container.config作为其中一个属性来处理，又添加了其他很多属性。

daemon.getInspectData(container)的代码不难理解，在本节的最后作为附录提供给读者参阅，这里不再解析。

```
func (s *Server) getImagesByName(version version.Version, w http.
vars map[string]string) error {
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    } imageInspect, err := s.daemon.Repositories().Lookup(vars["n
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

在这个函数中，所有的逻辑处理都在s.daemon.Repositories().Lookup函数中：

```
func (s *TagStore) Lookup(name string) (*types.ImageInspect, erro
    image, err := s.LookupImage(name)
    if err != nil || image == nil {
        return nil, fmt.Errorf("No such image: %s", name)
    }

    imageInspect := &types.ImageInspect{
        Id:              image.ID,
        Parent:          image.Parent,
        Comment:         image.Comment,
        Created:         image.Created,
        Container:       image.Container,
        ContainerConfig:  &image.ContainerConfig,
        DockerVersion:    image.DockerVersion,
        Author:          image.Author,
        Config:          image.Config,
        Architecture:    image.Architecture,
        Os:              image.OS,
        Size:            image.Size,
        VirtualSize:     image.GetParentsSize(0) + image.Size,
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if img != nil {
        return img, err
    }

    // name must be an image ID.
    store.Lock()
    defer store.Unlock()
    if img, err = store.graph.Get(name); err != nil {
        return nil, err
    }

    return img, nil
}
```

inspect的服务端处理还是非常简单的。处理逻辑就是获取到指定对象，然后基于此对象封装某一些属性。当服务端将封装好的JSON字符串返回给客户端时，客户端就可以针对这些数据进一步处理了，比如筛选。

inspect命令的看点在于客户端是如何对JSON字符串进行筛选的。在讲解inspect命令如何使用时，只提到了通过docker inspect-f‘{{.XXXX}}’的方式就可以获取到指定的属性值。这里我们就深入地探讨一下Docker是如何做到的。

其实实现这个功能是一件挺简单的事情，Docker就是借助Golang的模板语法来实现这个功能的。模板语法，相信大部分读者都听说过，目

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

则：

```
{{.}} 输出当前对象的值
{{.username}} 输出当前对象中字段或方法名称为 username 的值，也可以访问其
{{.username.config}} ，如果“config” 返回一个 Struct 对象，则还可以继续
{{.username.config.xxxx}}
```

以上是变量基本使用规则，还有一种稍高级的使用规则：

```
{{Method1 "参数值1" “参数值2"}} 定义方法“Method1”，参数值1和参数值2作为
```

在Inspect中只用到了上述三种用法，所以我们就只介绍到这里。如果读者还想深入研究，请访问Golang官网获取相关资料。下面我们看看inspect是如何使用模板语法的，首先第一步获取template对象：

```
    var tmpl *template.Template
    if *tmplStr != "" {
        var err error
        if tmpl, err = template.New("").Funcs(funcMap).Parse(*tmp
            return StatusError{StatusCode: 64,
                Status: "Template parsing error: " + err.Error()
        }
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
and，call，html，index，js，len，not，or，print，printf，println，ur
```

关于这些内置函数，用户可以直接在-f参数中使用。上面仅仅是完成了模板语法的第一步，而后执行的语句是：

```
                if err := tmpl.Execute(cli.out, inspPtr); err !=
                    rdr.Seek(0, 0)
                    var raw interface{}
                    if err := dec.Decode(&raw); err != nil {
                        return err
                    }
                    if err = tmpl.Execute(cli.out, raw); err != r
                        return err
                    }
                }
```

目标字符串inspPtr保存的是服务端返回的JSON字符串。这样连贯起来，就是在inspPtr中查找Config属性，然后将Config属性值返回给JSON函数，最后输出到终端（cli.out）。再结合我们前面所说的三个变量使用方式，inspect就可以提供筛选功能了。

因为inspect需要分辨服务端返回的是镜像的JSON数据还是容器的JSON数据，所以需要一些判断。这些判断非常简单，读者可以看本节最开始的inspect代码自行分析。在本节，我们需要熟悉如何在Golang中使用模板语法，同时需要熟悉Docker中保存镜像底层数据和容器底层数

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        ExitCode:    container.State.ExitCode,
        Error:       container.State.Error,
        StartedAt:   container.State.StartedAt,
        FinishedAt:  container.State.FinishedAt,
    }
    volumes := make(map[string]string)
    volumesRW := make(map[string]bool)

    for _, m := range container.MountPoints {
        volumes[m.Destination] = m.Path()
        volumesRW[m.Destination] = m.RW
    }
    contJSONBase := &types.ContainerJSONBase{
        Id:              container.ID,
        Created:         container.Created,
        Path:            container.Path,
        Args:            container.Args,
        State:           containerState,
        Image:         container.ImageID,
        NetworkSettings:  container.NetworkSettings,
        ResolvConfPath:   container.ResolvConfPath,
        HostnamePath:     container.HostnamePath,
        HostsPath:        container.HostsPath,
        LogPath:          container.LogPath,
        Name:            container.Name,
        RestartCount:    container.RestartCount,
        Driver:          container.Driver,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    cmd.ParseFlags(args, true)

    var errNames []string
    for _, name := range cmd.Args() {
        if _, _, err := readBody(cli.call("POST", fmt.Sprintf("/c
*signal), nil, nil)); err != nil {
            fmt.Fprintf(cli.err, "%s\n", err)
            errNames = append(errNames, name)
        } else {
            fmt.Fprintf(cli.out, "%s\n", name)
        }
    }
    if len(errNames) > 0 {
        return fmt.Errorf("Error: failed to kill containers: %v",
    }
    return nil
}
```

没错，上面就是Kill客户端的所有代码。从这段代码可以看出，Kill客户端所有的工作就是调用服务端所提供的/containers/{name:.*}/kill这个API，用户所指定的信号量会作为参数传入到服务端进行处理。Kill命令的客户端除此之外就没有其他操作了。下面我们将分析的视角转到服务端，看看服务端的处理逻辑。

```
func (s *Server) postContainersKill(version version.Version, w ht
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err := s.daemon.ContainerKill(name, sig); err != nil {
        return err
    }

    w.WriteHeader(http.StatusNoContent)
    return nil
}
```

Kill的服务端处理函数同样也不复杂，主要的处理函数为s.daemon.ContainerKill。在调用此函数之前，Docker会解析用户所传来的信号量参数。在这段处理函数中，如果用户提供的是信号量值，例如“9”，那么就直接将其转化为数字。如果用户输入的是SIGKILL，那么Docker就通过signal.SignalMap进行匹配，例如，“KILL”对应syscall.SIGKILL，而syscall.SIGKILL则等于Signal （0x9）。

当信号量完成解析之后，就开始调用s.daemon.ContainerKill强行关闭容器。下面我们就深入到这个函数当中看看它是如何完成这项任务的。

```
func (daemon *Daemon) ContainerKill(name string, sig uint64) erro
    container, err := daemon.Get(name)
    if err != nil {
        return err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return nil
    }

    // 1. Send SIGKILL
    if err := container.killPossiblyDeadProcess(9); err != nil {
        if container.IsRunning() {
            container.WaitStop(2 * time.Second)
            if container.IsRunning() {
                return err
            }
        }
    }

    // 2. Wait for the process to die, in last resort, try to kil
    if err := killProcessDirectly(container); err != nil {
        return err
    }

    container.WaitStop(-1 * time.Second)
    return nil
}

func (container *Container) killPossiblyDeadProcess(sig int) erro
    err := container.KillSig(sig)
    if err == syscall.ESRCH {
        logrus.Debugf("Cannot  kill  process  (pid=%d)  with  sig
container.GetPid(), sig)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    container.monitor.ExitOnNext()

    if container.Restarting {
        return nil
    }

    return container.daemon.Kill(container, sig)
}
```

在KillSig函数中存在一些判断，例如，判断容器是否出于暂停状态、容器是否出于运行状态或者容器是否处于重启状态等。这里的判断并不是因为处于这些状态，执行Kill命令会出错，而是因为使用Kill命令就是为了关闭容器。如果容器当前已经不处于运行状态了，那么就没有再运行Kill命令的必要了。所以这里加了一些判断。

如果这些检查全部通过，则说明指定的容器是我们必须要关闭的，Docker可在后续调用container.daemon.Kill(container, sig)来关闭。这里的Kill命令可不是我们在上面的介绍的Kill命令了，Docker源码中有很多这样的名称相近的函数，需要读者特别注意。

```
func (daemon *Daemon) Kill(c *Container, sig int) error {
    return daemon.execDriver.Kill(c.command, sig)
}
```

在这个container.daemon.Kill(container, sig)函数之中，可以看到调用

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

这个结构体的名称有些误导人，笔者在第一次看到时，还错以为是需要执行的命令，再深入一看结构体定义时，才得知原来Command是容器信息定义。Command定义如下：

```
type Command struct {
    ID                 string            `json:"id"`
    Rootfs             string            `json:"rootfs"` // roo
    ReadonlyRootfs     bool              `json:"readonly_rootfs
    InitPath           string            `json:"initpath"` // c
    WorkingDir         string            `json:"working_dir"`
    ConfigPath         string           `json:"config_path"` //
when the lxc template is moved into the driver
    Network            *Network          `json:"network"`
    Ipc                *Ipc              `json:"ipc"`
    Pid                *Pid              `json:"pid"`
    UTS                *UTS              `json:"uts"`
    Resources          *Resources        `json:"resources"`
    Mounts             []Mount           `json:"mounts"`
    AllowedDevices     []*configs.Device `json:"allowed_devices"
    AutoCreatedDevices []*configs.Device `json:"autocreated_devi
    CapAdd             []string          `json:"cap_add"`
    CapDrop            []string          `json:"cap_drop"`
    ContainerPid       int      `json:"container_pid"`  // the p
    ProcessConfig      ProcessConfig  `json:"process_config"` //
container.
    ProcessLabel       string            `json:"process_label"`
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return err
    }
```

看到这里，读者应该明白了，Kill命令本质就是向容器初始化进程发送指定信号，而为了找到这个进程，Kill命令做了很多额外的工作。但是不是向此进程发送完信号，Kill命令就算执行完毕了呢？当然不是，因为在s.daemon.ContainerKill中还有一个步骤，我们还没分析，那就是

killProcessDirectly函数是在执行killPossiblyDeadProcess之后调用的，是为了确保指定的容器一定会被关掉。而它又是什么样的逻辑呢？请看下面代码：

```
func killProcessDirectly(container *Container) error {
    if _, err := container.WaitStop(10 * time.Second); err != nil
        // Ensure that we don't kill ourselves
        if pid := container.GetPid(); pid != 0 {
            logrus.Infof("Container %s failed to exit within 10 s
stringid.TruncateID(container.ID))
            if err := syscall.Kill(pid, 9); err != nil {
                if err != syscall.ESRCH {
                    return err
                }
                logrus.Debugf("Cannot kill process (pid=%d) with
            }
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func wait(waitChan <-chan struct{}, timeout time.Duration) error
    if timeout < 0 {
        <-waitChan
        return nil
    }
    select {
    case <-time.After(timeout):
        return fmt.Errorf("Timed out: %v", timeout)
    case <-waitChan:
        return nil
    }
}
```

通过上面两个函数可以得知：每个容器关闭时，都会向Docker Daemon发送一个消息。如果Docker Daemon在指定时间内没有收到这个关闭消息，则认为容器关闭失败。此后Docker Daemon会执行强制关闭流程。

分析到这里，我们才算完整分析完Kill命令。对于本节所分析的Kill命令来说，客户端和服务端的逻辑都不难，重点是理解容器初始化进程的概念，以及如何向容器发送指定信号。在Docker模块定制开发中，如何管理容器是重中之重，掌握如何向容器发送信号又是其中很重要的一个方式，如果读者有此方面的开发需求，建议在Docker源码基础上亲自调试和修改，这样才能更加深刻理解Docker是如何管理容器的。

13. Logs命令

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    var c types.ContainerJSON
    if err := json.NewDecoder(stream).Decode(&c); err != nil {
        return err
    }

    if logType := c.HostConfig.LogConfig.Type; logType != "json-f
        return fmt.Errorf("\"logs\" command is supported only for
(got: %s)", logType)
    }

    v := url.Values{}
    v.Set("stdout", "1")
    v.Set("stderr", "1")

    if *since != "" {
        v.Set("since", timeutils.GetTimestamp(*since))
    }

    if *times {
        v.Set("timestamps", "1")
    }

    if *follow {
        v.Set("follow", "1")
    }
    v.Set("tail", *tail)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

假设当前配置的Docker日志保存方式为json-file，那么Logs后续就开始封装参数，然后调用/containers/{name:.*}/logs。除刚才遗留的问题外，Logs的客户端代码应该也算比较简单的。下面我们就转向Los服务端代码：

```
func (s *Server) getContainersLogs(version version.Version, w htt
vars map[string]string) error {
    if err := parseForm(r); err != nil {
        return err
    }
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    }

    // Validate args here, because we can't return not StatusOK a
    stdout, stderr := boolValue(r, "stdout"), boolValue(r, "stder
    if !(stdout || stderr) {
        return fmt.Errorf("Bad parameters: you must choose at lea
    }

    var since time.Time
    if r.Form.Get("since") != "" {
        s, err := strconv.ParseInt(r.Form.Get("since"), 10, 64)
        if err != nil {
            return err
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    Tail                string
    Since               time.Time
    UseStdout, UseStderr bool
    OutStream           io.Writer
}
```

由此可以看出，其实ContainerLogsConfig结构体就是为了容器获取日志而准备的，在Docker源码之中，使用了非常多的结构体。如果读者对此理解有困难，可以将这些结构体理解为Java中的对象。如果读者有Golang的使用经验，那么理解起来就非常容易了。

Docker封装好ContainerLogsConfig结构体之后，就开始调用s.daemon.ContainerLogs函数，而参数就是刚才封装好的ContainerLogsConfig结构体。这么做的目的，是为了使用刚才传入的参数（类似于生成一个Java对象，然后将此对象作为参数调用另外一个对象的方法一样）。

下面是s.daemon.ContainerLogs函数代码：

```
func (daemon *Daemon) ContainerLogs(name string, config *Containe
    var (
        lines  = -1
        format string
    )
    if !(config.UseStdout || config.UseStderr) {
        return fmt.Errorf("You must choose at least one stream")
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    logDriver, err := container.getLogger()
    cLog, err := logDriver.GetReader()
    if err != nil {
        logrus.Errorf("Error reading logs: %s", err)
    } else {
        // json-file driver
        if config.Tail != "all" {
            var err error
            lines, err = strconv.Atoi(config.Tail)
            if err != nil {
                logrus.Errorf("Failed to parse tail %s, error: %v
                lines = -1
            }
        }

        if lines != 0 {
            if lines > 0 {
                f := cLog.(*os.File)
                ls, err := tailfile.TailFile(f, lines)
                if err != nil {
                    return err
                }
                tmp := bytes.NewBuffer([]byte{})
                for _, l := range ls {
                    fmt.Fprintf(tmp, "%s\n", l)
                }
                cLog = tmp
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        }
    }

    if config.Follow && container.IsRunning() {
        chErr := make(chan error)
        var stdoutPipe, stderrPipe io.ReadCloser

        // write an empty chunk of data (this is to ensure that t
        // HTTP Response is sent immediatly, even if the containe
        // not yet produced any data)
        outStream.Write(nil)

        if config.UseStdout {
            stdoutPipe = container.StdoutLogPipe()
            go func() {
                logrus.Debug("logs: stdout stream begin")
                chErr <- jsonlog.WriteLog(stdoutPipe, outStream,
                logrus.Debug("logs: stdout stream end")
            }()
        }
        if config.UseStderr {
            stderrPipe = container.StderrLogPipe()
            go func() {
                logrus.Debug("logs: stderr stream begin")
                chErr <- jsonlog.WriteLog(stderrPipe, errStream,
                logrus.Debug("logs: stderr stream end")
            }()
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

是解析参数，这些参数都是由客户端传来，并且在getContainersLogs当中封装到ContainerLogsConfig结构体之中。第二层是获取到容器中保存的日志数据，获取代码很简单：

```
logDriver, err := container.getLogger()
cLog, err := logDriver.GetReader()
```

通过这两行代码就完成了获取日志的操作，我们稍后会分析这两行背后的工作。

第三层，就是使用第一层解析后的参数来提取日志。这一层的代码量占据了本函数代码量的60%。虽然代码量大，但理解起来并不困难。

这样我们就把s.daemon.ContainerLogs这个函数分解开来，下面我们逐层分析这个函数。

首先是第一层，参数解析：

```
var (
    lines  = -1
    format string
)
if !(config.UseStdout || config.UseStderr) {
    return fmt.Errorf("You must choose at least one stream")
}
if config.Timestamps {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

定义了，定义如下：

```
const (
    RFC3339NanoFixed = "2006-01-02T15:04:05.000000000Z07:00"
    JSONFormat = `"` + time.RFC3339Nano + `"`
)
```

由此也就看出了RFC3339NanoFixed其实是Docker官方自己定义的一个时间戳显示格式。当用户通过指定timestamps来显示日志时间戳时，默认显示的就是这种格式。

再往后看，Docker通过daemon.Get(name)获取到了容器实例，而获此实例是为了后面获取容器日志数据做准备。

这些工作都做完之后，就开始了第二层的处理工作：获取容器日志数据。

Docker获取容器日志时，首先调用的是：logDriver, err:=container.getLogger()。下面我们看看此函数的逻辑：

```
func (container *Container) getLogger() (logger.Logger, error) {
    cfg := container.getLogConfig()
    c, err := logger.GetLogDriver(cfg.Type)
    if err != nil {
        return nil, fmt.Errorf("Failed to get logging factory: %v
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

为什么这么说呢？因为container.getLogConfig()函数就是直接返回LogConfig对象，代码如下：

```
func (container *Container) getLogConfig() runconfig.LogConfig {
    cfg := container.hostConfig.LogConfig
    if cfg.Type != "" { // container has log driver configured
        return cfg
    }
    // Use daemon's default log config for containers
    return container.daemon.defaultLogConfig
}
```

```
"LogConfig": {
            "Type": "json-file",
            "Config": {}
        }
```

只有在cfg.Type为空的情况下，才会返回默认配置参数。而默认的配置参数同样也是

Docker随后通过c， err:=logger.GetLogDriver(cfg.Type)来获取日志驱动。这里Docker获取的日志驱动又为何物呢？继续往下看logger.GetLogDriver的代码。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

动名称获取到其所对应的驱动函数而已。

Docker一共支持几种日志驱动类型呢？目前Docker支持三种，分别是：JSON-file、syslog和journald。在这三种驱动方式中，JSON-file是支持度最高、最稳定的，另外两种都不太成熟和稳定。所以如果用户没有特别需要，建议用户保持默认参数，选择JSON-file这种驱动方式。

当Docker初始化容器时，会根据容器中的log-driver指定的参数名称调用相对应的注册函数，完成注册。因此，在每个驱动结构体中都会存在如下一段代码：

```
func init() {
    if err := logger.RegisterLogDriver(name, New); err != nil {
        logrus.Fatal(err)
    }
}
```

而这里的New函数就是我们刚才所说的驱动（以json-file为例）：

```
func New(ctx logger.Context) (logger.Logger, error) {
    log, err := os.OpenFile(ctx.LogPath, os.O_RDWR|os.O_APPEND|os
    if err != nil {
        return nil, err
    }
    return &JSONFileLogger{
        f:   log,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

情。

我们可以在后面的代码中看到Docker需要对lines进行判断。此时的lines有什么作用呢？我们可以看下面的代码：

```
        if config.Tail != "all" {
            var err error
            lines, err = strconv.Atoi(config.Tail)
            if err != nil {
                logrus.Errorf("Failed to parse tail %s, error: %v
                lines = -1
            }
        }
```

如果用户通过--tail指定了行数，那么Docker就会通过strconv.Atoi将用户所输入的数字转成整型，并赋值给lines。由此我们可以得知，lines就是用户所期望查看的日志行数。了解这些，下面的这段代码就变得很容易理解了：

```
            if lines > 0 {
                f := cLog.(*os.File)
                ls, err := tailfile.TailFile(f, lines)
                if err != nil {
                    return err
                }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            if l.Stream == "stderr" && config.UseStderr {
                io.WriteString(errStream, logLine)
            }
        }
```

这里从cLog，也就是我们所获取的保存有日志内容的结构体中截取出指定行数的数据。再将cLog转换成json结构体，循环将cLog保存的数据输出到outStream或者errStream当中去。这两个I/O流还记得吗？这就是在Logs开始时所初始化的那两个I/O流。

如果用户没有指定其他选项，那么Logs命令执行到这里就完成了。但如果用户指定了--follow选项，那么Docker就会通过channel实时将容器所产生的数据发送给用户。完成这个任务的代码很简单，具体如下：

```
    if config.UseStdout {
        stdoutPipe = container.StdoutLogPipe()
        go func() {
            logrus.Debug("logs: stdout stream begin")
            chErr <- jsonlog.WriteLog(stdoutPipe, outStream,
            logrus.Debug("logs: stdout stream end")
        }()
    }
```

Docker获取到容器的输出管道，然后将jsonlog.WriteLog函数返回的日志内容发送到chErr这个channel中。在jsonlog.WriteLog函数中，凭借

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

看到这里，Logs命令就全部分析完了。Logs命令在日常使用当中出现的频率比较高，查看容器日志几乎是生产环境中例行的一项工作了。

在本节，我们重点分析了Docker是如何处理日志驱动的，包括注册、获取和使用。在分析过程中，我们也注意到Docker中的Logs命令并不是十全十美的，例如，syslog和journald这两种驱动方式还不稳定。但完全开发一种新的日志驱动还是比较简单的事情，只要满足Logger的接口要求就可以。

14. ps命令

下面我们来分析Docker是如何通过读取环境数据来筛选容器的。用户通过调用ps命令，并附加一些参数就可以筛选出指定的容器。从用户的角度来说，不需要关心容器运行数据保存在什么地方。但对于Docker来说，则需要随时随地的维护容器信息。下面就来看看Docker是从什么地方取出这些容器数据，并加以筛选的。

下面是ps的客户端代码：

```
func (cli *DockerCli) CmdPs(args ...string) error {
    var (
        err error

        psFilterArgs = filters.Args{}
        v            = url.Values{}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
if *last != -1 {
    v.Set("limit", strconv.Itoa(*last))
}

if *since != "" {
    v.Set("since", *since)
}

if *before != "" {
    v.Set("before", *before)
}

if *size {
    v.Set("size", "1")
}

// Consolidate all filter flags, and sanity check them.
// They'll get processed in the daemon/server.
for _, f := range flFilter.GetAll() {
    if psFilterArgs, err = filters.ParseFlag(f, psFilterArgs)
        return err
    }
}

if len(psFilterArgs) > 0 {
    filterJSON, err := filters.ToParam(psFilterArgs)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    stripNamePrefix := func(ss []string) []string {
        for i, s := range ss {
            ss[i] = s[1:]
        }

        return ss
    }
    for _, container := range containers {
        ID := container.ID

        if !*noTrunc {
            ID = stringid.TruncateID(ID)
        }

        if *quiet {
            fmt.Fprintln(w, ID)

            continue
        }
        var (
            names   = stripNamePrefix(container.Names)
            command = strconv.Quote(container.Command)
        )

        if !*noTrunc {
            command = stringutils.Truncate(command, 20)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        }

        fmt.Fprint(w, "\n")
    }

    if !*quiet {
        w.Flush()
    }

    return nil
}
```

从ps客户端代码的层次可以看出，ps命令属于典型的三段式代码。即第一部分代码用来解析并且封装用户所指定的命令行参数，第二部分用来调用服务端提供的API，第三部分则用来输出服务端返回的结果。

第一部分和第三部分的逻辑，在之前的命令中已经多次遇到了。所以本节我们直接看一下服务端是如何处理ps请求的。

首先看一下服务端API：

```
func (s *Server) getContainersJSON(version version.Version, w htt
vars map[string]string) error {
    if err := parseForm(r); err != nil {
        return err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

所有的参数封装为ContainersConfig结构体，而代码的最后则是输出s.daemon.Containers的执行结果，因此我们需要将视线转到s.daemon.Containers函数。

```
func (daemon *Daemon) Containers(config *ContainersConfig) ([]*ty
    var (
        foundBefore bool
        displayed   int
        all         = config.All
        n           = config.Limit
        psFilters   filters.Args
        filtExited  []int
    )
    containers := []*types.Container{}

    psFilters, err := filters.FromParam(config.Filters)
    if err != nil {
        return nil, err
    }
    if i, ok := psFilters["exited"]; ok {
        for _, value := range i {
            code, err := strconv.Atoi(value)
            if err != nil {
                return nil, err
            }
            filtExited = append(filtExited, code)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            return nil, err
        }
    }
    errLast := errors.New("last container")
    writeCont := func(container *Container) error {
        container.Lock()
        defer container.Unlock()
        if !container.Running && !all && n <= 0 && config.Since =
            return nil
        }
        if !psFilters.Match("name", container.Name) {
            return nil
        }

        if !psFilters.Match("id", container.ID) {
            return nil
        }

        if !psFilters.MatchKVList("label", container.Config.Label
            return nil
        }

        if config.Before != "" && !foundBefore {
            if container.ID == beforeCont.ID {
                foundBefore = true
            }
            return nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            ID:    container.ID,
            Names: names[container.ID],
        }
        newC.Image = container.Config.Image
        if len(container.Args) > 0 {
            args := []string{}
            for _, arg := range container.Args {
                if strings.Contains(arg, " ") {
                    args = append(args, fmt.Sprintf("'%s'", arg))
                } else {
                    args = append(args, arg)
                }
            }
            argsAsString := strings.Join(args, " ")

            newC.Command = fmt.Sprintf("%s %s", container.Path, a
        } else {
            newC.Command = fmt.Sprintf("%s", container.Path)
        }
        newC.Created = int(container.Created.Unix())
        newC.Status = container.State.String()

        newC.Ports = []types.Port{}

        for port, bindings := range container.NetworkSettings.Por
            p, _ := nat.ParsePort(port.Port())
            if len(bindings) == 0 {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    for _, container := range daemon.List() {
        if err := writeCont(container); err != nil {
            if err != errLast {
                return nil, err
            }
            break
        }
    }
    return containers, nil
}
```

如果我们对s.daemon.Containers(config)函数进行分层，就可以看出这个函数是按照下面的逻辑顺序来执行的：

①获取当前环境中所有的容器列表；

②依次对列表中的容器进行筛选；

③将符合筛选规则的容器返回给客户端。

在上面的源码中，Docker通过下面的代码获取当前所有的容器：

```
    daemon.ContainerGraph().Walk("/", func(p string, e *graphdb.E
        names[e.ID()] = append(names[e.ID()], p)
        return nil
    }, 1)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            if err != errLast {
                return nil, err
            }
            break
        }
    }
```

daemon.List()返回的是当前环境中所有的容器ID，因此通过for循环依次取出作为writeCont的参数进行合法性验证。在writeCont函数当中，又分为两部分逻辑。首先是根据用户提交的筛选参数进行筛选，例如，是否是指定容器名称、是否是指定容器ID、是否是指定运行状态，等等。如果无法通过筛选，则跳过此容器。反之则通过容器筛选，Docker会将这个通过筛选的容器封装成特定的结构体，如下：

```
        newC := &types.Container{
            ID:    container.ID,
            Names: names[container.ID],
        }
        newC.Image = container.Config.Image


type Container struct {
    ID         string `json:"Id"`
    Names      []string
    Image      string
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

很大的用处。所以如果读者需要基于容器数据做额外的扩展功能，那么适时的运用SQLite中的数据，会大大简化代码逻辑。

15. Rename命令

在使用Rename命令时，需要特别留意Rename只能针对容器进行重命名操作，不能针对镜像进行重命名。那么如果需要对镜像进行重命名呢？很简单，使用Tag命令。Tag和Rename命令都可以完成重命名的操作，但是两者的处理逻辑却完全不同。本节我们先解析Rename命令，看看Docker通过哪些操作可以完成对一个容器的重命名。

解析Rename命令，首先从客户端代码入手：

```
func (cli *DockerCli) CmdRename(args ...string) error {
    cmd := cli.Subcmd("rename", "OLD_NAME NEW_NAME", "Rename a co
    cmd.Require(flag.Exact, 2)
    cmd.ParseFlags(args, true)

    oldName := cmd.Arg(0)
    newName := cmd.Arg(1)

    if _, _, err := readBody(cli.call("POST", fmt.Sprintf("/conta
newName), nil, nil)); err != nil {
        fmt.Fprintf(cli.err, "%s\n", err)
        return fmt.Errorf("Error: failed to rename container name
    }
    return nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err := s.daemon.ContainerRename(name, newName); err != nil
        return err
    }
    w.WriteHeader(http.StatusNoContent)
    return nil
}
```

在postContainerRename函数中，Docker通过s.daemon.ContainerRename完成主要逻辑。在此之前的代码都是为了获取参数，可略过不看。下面是s.daemon.ContainerRename的代码：

```
func (daemon *Daemon) ContainerRename(oldName, newName string) er
    if oldName == "" || newName == "" {
        return fmt.Errorf("usage: docker rename OLD_NAME NEW_NAME
    }

    container, err := daemon.Get(oldName)
    if err != nil {
        return err
    }

    oldName = container.Name

    container.Lock()
    defer container.Unlock()
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

两个问题。通过container, err:=daemon.Get(oldName)，Docker找到了oldName对应的容器对象。此对象包含容器ID属性，因此Docker就通过oldName找到了容器ID。这是第一个问题的答案。

然后Docker又通过daemon.reserveName(container.ID, newName)修改了指定容器ID所对应的容器名称。虽然目前我们还不知道Docker是如何实现的，但这就是第二个问题的答案。

ContainerRename函数后面的步骤则实现了一个回滚操作。这样在新容器名称落地前如果遇到异常，可以保证回滚到旧容器名称。仔细观察回滚代码，可以发现其也使用了daemon.reserveName(container.ID, oldName)函数，只不过将名称替换成了旧容器名称而已。那么daemon.reserveName函数到底长什么样呢？下面就来揭晓daemon.reserveName的庐山真面目：

```
func (daemon *Daemon) reserveName(id, name string) (string, erro
    if !validContainerNamePattern.MatchString(name) {
        return "", fmt.Errorf("Invalid container name (%s),
validContainerNameChars)
    }

    if name[0] != '/' {
        name = "/" + name
    }

    if _, err := daemon.containerGraph.Set(name, id); err != nil
        if !graphdb.IsNonUniqueNameError(err) {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

有遇到过，它究竟是谁呢？

通过翻看Daemon结构体定义可以知道daemon.containerGraph原型，如下：

```
    containerGraph   *graphdb.Database
```

如果看到这里读者没有恍然大悟的感觉，那么笔者建议读者温习一下Create命令。graphdb.Database是操作SQLite数据库的结构体。由此我们也就知道了containerGraph.Set其实是在对SQLite数据库进行Set操作。下面是Set操作代码：

```
func (db *Database) Set(fullPath, id string) (*Entity, error) {
    db.mux.Lock()
    defer db.mux.Unlock()

    tx, err := db.conn.Begin()
    if err != nil {
        return nil, err
    }

    var entityID string
    if err := tx.QueryRow("SELECT id FROM entity WHERE id = ?;",
        if err == sql.ErrNoRows {
            if _, err := tx.Exec("INSERT INTO entity (id) VALUES(
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if parent.id == e.id {
        return fmt.Errorf("Cannot set self as child")
    }

    if _, err := tx.Exec("INSERT INTO edge (parent_id, name, enti
name, e.id); err != nil {
        return err
    }
    return nil
}
```

Set操作就是当查询的纪录对象不存在时，插入一条新记录。这样在reserveName函数中出现的daemon.containerGraph.Set(name, id)，我们就明白是什么含义了。Docker先确保此容器存在，然后再插入一条新记录用来记录ID和新名称的对应关系。

那已经存在的旧记录怎么办？Docker考虑到一个容器ID不能同时拥有两个名称，因此在调用reserveName成功之后，通过daemon.containerGraph.Delete(oldName)删除已存在的旧名称。这些做完后，是不是就完成Rename工作了呢？

没有，还有最后一步。因为通过Inspect命令可以查询到容器详细信息，而在分析Inspect时，我们看到这个命令是通过解析容器的JSON参数文件来获取数据的，并非通过读取SQLite来获取数据。因此在Rename的最后，Docker通过container.toDisk()来实现落地。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return err
    }

    return ioutil.WriteFile(pth, data, 0666)
}
```

而在操作时如果出现任何异常，则Docker会再次恢复回旧的容器名称，这就是Rename命令所有的逻辑。

在本节，我们看到了保存容器信息的两个地点：一个是JSON文件，另外一个则是SQLite数据库。如果读者需要修改容器信息，那么就需要同时修改这两个地方。而至于Docker为什么需要将容器数据分别保存在不同的地方，我们会在后续章节中深入讲解。

16. Run命令

Run命令是Docker所有客户端命令当中，使用频率最高的命令。也是Docker所有客户端命令当中，参数最多、最复杂的命令。按照常理来说，Run命令的实现应该是最复杂的。

但是，Run命令的实现其实并不复杂，因为Run命令是根据不同的参数调用了不同的API，而这些API我们在讲解其他命令时基本都涉及到了，所以Run命令本质就是其他命令的一个结合体。

然而在Run命令众多调用的API里面，最重要的是/containers/create和/containers/{name:.*}/start。这两个API，读者眼熟吗？这两个API就是Create和Start命令所使用的API。在Run命令当中，Docker将用户提供的所有与容器相关的参数都封装成了hostConfig文件传递给

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        os.Exit(1)
    }
    if len(hostConfig.Dns) > 0 {
        for _, dnsIP := range hostConfig.Dns {
            if dns.IsLocalhost(dnsIP) {
                fmt.Fprintf(cli.err,  "WARNING:  Localhost  DNS
containers.\n", dnsIP)
                break
            }
        }
    }
    if config.Image == "" {
        cmd.Usage()
        return nil
    }

    if !*flDetach {
        if err := cli.CheckTtyInput(config.AttachStdin, config.Tt
            return err
        }
    } else {
        if fl := cmd.Lookup("-attach"); fl != nil {
            flAttach = fl.Value.(*opts.ListOpts)
            if flAttach.Len() != 0 {
                return ErrConflictAttachDetach
            }
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    )
    if !config.AttachStdout && !config.AttachStderr {
        waitDisplayID = make(chan struct{})
        go func() {
            defer close(waitDisplayID)
            fmt.Fprintf(cli.out, "%s\n", createResponse.ID)
        }()
    }
    if *flAutoRemove && (hostConfig.RestartPolicy.IsAlways() || h
        return ErrConflictRestartPolicyAndAutoRemove
    }
    hijacked := make(chan io.Closer)

    defer func() {
        logrus.Debugf("End of CmdRun(), Waiting for hijack to fin
        if _, ok := <-hijacked; ok {
            fmt.Fprintln(cli.err, "Hijack did not finish (chan st
        }
    }()
    if config.AttachStdin || config.AttachStdout || config.Attach
        var (
            out, stderr io.Writer
            in          io.ReadCloser
            v           = url.Values{}
        )
        v.Set("stream", "1")
        if config.AttachStdin {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    case err := <-errCh:
        if err != nil {
            logrus.Debugf("Error hijack: %s", err)
            return err
        }
    }
    defer func() {
        if *flAutoRemove {
            if _, _, err = readBody(cli.call("DELETE", "/containe
nil)); err != nil {
                fmt.Fprintf(cli.err, "Error deleting container: %
            }
        }
    }()
    if _, _, err = readBody(cli.call("POST", "/containers/"+creat
err != nil {
        return err
    }

    if  (config.AttachStdin  ||  config.AttachStdout  ||  config.
cli.isTerminalOut {
        if err := cli.monitorTtySize(createResponse.ID, false); e
            fmt.Fprintf(cli.err, "Error monitoring TTY size: %s\n
        }
    }

    if errCh != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        } else {
            if _, status, err = getExitCode(cli, createResponse.I
                return err
            }
        }
    }
    if status != 0 {
        return StatusError{StatusCode: status}
    }
    return nil
}
```

可以看到Run命令的客户端代码是Docker所有客户端代码中最多的，但这些代码都是为了调用/containers/create和/containers/{name:.*}/start而服务的。

大部分代码是在解析用户所提供的命令参数，并将其封装成hostConfig，然后通过cli.createContainer(config, hostConfig, hostConfig.ContainerIDFile,*flName)来创建容器。最后，Docker又调用cli.call("POST","/containers/"+createResponse.ID+"/start", nil, nil)来启动所创建成功的容器。

因为Run命令是分别调用API函数来实现的，所以Run命令并没有相对应的服务端函数。即对于Run命令来说，只需关注客户端代码即可。

17. Rmi命令

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func (cli *DockerCli) CmdRmi(args ...string) error {
    var (
        cmd     = cli.Subcmd("rmi", "IMAGE [IMAGE...]", "Remove c
        force   = cmd.Bool([]string{"f", "-force"}, false, "Force
        noprune = cmd.Bool([]string{"-no-prune"}, false, "Do not
    )
    cmd.Require(flag.Min, 1)
    cmd.ParseFlags(args, true)

    v := url.Values{}
    if *force {
        v.Set("force", "1")
    }
    if *noprune {
        v.Set("noprune", "1")
    }

    var errNames []string
    for _, name := range cmd.Args() {
        rdr, _, err := cli.call("DELETE", "/images/"+name+"?"+v.E
        if err != nil {
            fmt.Fprintf(cli.err, "%s\n", err)
            errNames = append(errNames, name)
        } else {
            dels := []types.ImageDelete{}
            if err := json.NewDecoder(rdr).Decode(&dels); err !=
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err := parseForm(r); err != nil {
        return err
    }
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    }

    name := vars["name"]
    force := boolValue(r, "force")
    noprune := boolValue(r, "noprune")

    list, err := s.daemon.ImageDelete(name, force, noprune)
    if err != nil {
        return err
    }

    return writeJSON(w, http.StatusOK, list)
}
```

在deleteImages函数当中，所有的逻辑都是调用s.daemon.ImageDelete来完成的。调用此函数之前的工作是为了解析参数。

```
func (daemon *Daemon) ImageDelete(name string, force, noprune boo
    list := []types.ImageDelete{}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err != nil {
        if r, _ := daemon.Repositories().Get(repoName); r != nil
            return fmt.Errorf("No such image: %s", utils.ImageRef
        }
        return fmt.Errorf("No such image: %s", name)
    }

    if strings.Contains(img.ID, name) {
        repoName = ""
        tag = ""
    }

    byParents, err := daemon.Graph().ByParent()
    if err != nil {
        return err
    }
    repos := daemon.Repositories().ByID()[img.ID]

    deleteByID := repoName == “"
    if deleteByID {
        for _, repoAndTag := range repos {
            parsedRepo, parsedTag := parsers.ParseRepositoryTag(r
            if repoName == "" || repoName == parsedRepo {
                repoName = parsedRepo
                if parsedTag != "" {
                    repoAndTags[repoName] = append(repoAndTags[re
                }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            return err
        }
    }

    for repoName, tags := range repoAndTags {
        for _, tag := range tags {
            tagDeleted, err := daemon.Repositories().Delete(repoN
            if err != nil {
                return err
            }
            if tagDeleted {
                *list = append(*list, types.ImageDelete{
                    Untagged: utils.ImageReference(repoName, tag)
                })
                daemon.EventsService.Log("untag", img.ID, "")
            }
        }
    }
    tags = daemon.Repositories().ByID()[img.ID]

    if (len(tags) <= 1 && repoName == "") || len(tags) == 0 {
        if len(byParents[img.ID]) == 0 {
            if err := daemon.Repositories().DeleteAll(img.ID); er
                return err
            }
            if err := daemon.Graph().Delete(img.ID); err != nil {
                return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

被多个节点使用）而结束。

而imgDeleteHelper函数是如何判断当前镜像是否可删除的呢？我们来看一下imgDeleteHelper的逻辑。

```
    byParents, err := daemon.Graph().ByParent()
    if err != nil {
        return err
    }

func (graph *Graph) ByParent() (map[string][]*image.Image, error)
    byParent := make(map[string][]*image.Image)
    err := graph.walkAll(func(img *image.Image) {
        parent, err := graph.Get(img.Parent)
        if err != nil {
            return
        }
        if children, exists := byParent[parent.ID]; exists {
            byParent[parent.ID] = append(children, img)
        } else {
            byParent[parent.ID] = []*image.Image{img}
        }
    })
    return byParent, err
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

状态。

当imgDeleteHelper函数生成此拓扑图之后，Docker需要将此镜像所有的标签去除掉，去除标签则是为了将此镜像数据从Docker Daemon中移除。

然后Docker开始进行镜像删除前的判断，此判断应该是imgDeleteHelper函数最重要的一个判断了。

```
if (len(tags) <= 1 && repoName == "") || len(tags) == 0 {
    if len(byParents[img.ID]) == 0 {
        if err := daemon.Repositories().DeleteAll(img.ID); er
            return err
        }
        if err := daemon.Graph().Delete(img.ID); err != nil {
            return err
        }
        *list = append(*list, types.ImageDelete{
            Deleted: img.ID,
        })
        tools.Logger("Delete -> ",img.ID)
        daemon.EventsService.Log("delete", img.ID, "")
        if img.Parent != "" && !noprune {
            err := daemon.imgDeleteHelper(img.Parent, list, f
            if first {
                return err
            }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

有太多出彩的地方，反而是如何判断此镜像是否被其他子镜像所调用是一个亮点。Docker通过构建实时父子镜像拓扑图的形式，来判断此镜像是否处于被调用状态。在实际生产环境当中，可以通过UI将这个实时拓扑图展现给用户，这样更有利于镜像维护。

18. Start命令

通过Start命令用户可以启动指定的容器，但这个容器必须处于Stop状态才可被启动。

在开始讲解Start命令之前，我们先来简单回顾一下创建一个容器所需的条件。首先需要文件系统，容器运行所需的文件系统需要从最底层文件开始，逐层叠加而成。然后需要虚拟化驱动，也就是LXC或者native，用于创建此容器的init初始化进程。待此容器的init进程初始化成功之后，init进程再依次加载文件系统中的文件，这样一个容器就开始运行了。

所以一个容器运行只需要两步：创建文件系统，分配运行资源。

之前介绍的Create命令中，其实只完成了第一步创建文件系统。而Start命令，则同时完成了这两件事情。下面我们就看看Start命令是如何运行的：

```
func (cli *DockerCli) CmdStart(args ...string) error {
    var (
        cErr chan error
        tty  bool
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            sigc := cli.forwardAllSignals(cmd.Arg(0))
            defer signal.StopCatch(sigc)
        }

        var in io.ReadCloser

        v := url.Values{}
        v.Set("stream", "1")

        if *openStdin && c.Config.OpenStdin {
            v.Set("stdin", "1")
            in = cli.in
        }

        v.Set("stdout", "1")
        v.Set("stderr", "1")

        hijacked := make(chan io.Closer)
        // Block the return until the chan gets closed
        defer func() {
            logrus.Debugf("CmdStart() returned, defer waiting for
            if _, ok := <-hijacked; ok {
                fmt.Fprintln(cli.err, "Hijack did not finish (cha
            }
            cli.in.Close()
        }()
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            }
        } else {
            if !*attach && !*openStdin {
                fmt.Fprintf(cli.out, "%s\n", name)
            }
        }
    }
    if len(errNames) > 0 {
        encounteredError = fmt.Errorf("Error: failed to start cor
    }
    if encounteredError != nil {
        return encounteredError
    }
    if *openStdin || *attach {
        if tty && cli.isTerminalOut {
            if err := cli.monitorTtySize(cmd.Arg(0), false); err
                fmt.Fprintf(cli.err, "Error monitoring TTY size:
            }
        }
        if attchErr := <-cErr; attchErr != nil {
            return attchErr
        }
        _, status, err := getExitCode(cli, cmd.Arg(0))
        if err != nil {
            return err
        }
        if status != 0 {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            return err
        }

        hostConfig = c
    }

    if err := s.daemon.ContainerStart(vars["name"], hostConfig);
        if err.Error() == "Container already started" {
            w.WriteHeader(http.StatusNotModified)
            return nil
        }
        return err
    }
    w.WriteHeader(http.StatusNoContent)
    return nil
}
```

在postContainersStart函数中，可以看到Docker预留了hostConfig这个结构体，此结构体可以从客户端中读取而来，但Start命令的客户端却没有此参数。当用户通过其他方式来调用此API时，可以使用此结构体来定制容器参数。

下面我们看看s.daemon.ContainerStart函数。

```
func (daemon *Daemon) ContainerStart(name string, hostConfig *run
    container, err := daemon.Get(name)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

在ContainerStart函数当中，Docker判断了目标容器是否满足执行Start的要求，也就是目标容器不能处于Pause或者Running状态。为什么不能是这两种状态呢？在前面，我们提到了容器运行需要两个条件：文件系统和所需资源。

而处于Pause和Running状态的容器，都已经拥有了运行资源。虽然Pause状态的容器运行资源暂时不能使用，但通过Unpause之后，此部分资源就被激活了。如果对处于此两种状态的容器继续分配资源，那么就会发生资源冲突。因此只能对处于Stop状态的容器执行Start命令了。

通过状态检查之后，Docker开始调用container.Start()来启动容器。我们看看container.Start()的逻辑。

```
func (container *Container) Start() (err error) {
    container.Lock()
    defer container.Unlock()

    if container.Running {
        return nil
    }

    if container.removalInProgress || container.Dead {
        return fmt.Errorf("Container is marked for removal and ca
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    env := container.createDaemonEnvironment(linkedEnv)
    if err := populateCommand(container, env); err != nil {
        return err
    }

    mounts, err := container.setupMounts()
    if err != nil {
        return err
    }

    container.command.Mounts = mounts
    return container.waitForStart()
}
```

在container.Start()函数中，共分为两部分逻辑。Docker通过调用container.Mount()将此容器所使用的文件层挂载上去，然后再通过container.setupMounts()将其余文件层挂载上去。而container.initializeNetworking()、container.setupLinkedContainers()、container.setupWorkingDirectory()和container.createDaemonEnvironment(linkedEnv)这几个函数都是根据所传入的hostConfig配置参数重新生成目标容器的配置文件，也就是config.json、hostconfig.json、hostname等这几个文件。

Docker做这些是为了准备文件系统，而后Docker调用container.waitForStart()开始创建资源。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        if afterRun {
            m.container.Lock()
            m.container.setStopped(&exitStatus)
            defer m.container.Unlock()
        }
        m.Close()
    }()

    m.container.RestartCount = -1

    for {
        m.container.RestartCount++

        if err := m.container.startLogging(); err != nil {
            m.resetContainer(false)

            return err
        }

        pipes  :=  execdriver.NewPipes(m.container.stdin,  m.cont
m.container.Config.OpenStdin)

        m.container.LogEvent("start")

        m.lastStartTime = time.Now()

        if exitStatus, err = m.container.daemon.Run(m.container,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            continue
        }
        if exitStatus.OOMKilled {
            m.container.LogEvent("oom")
        }
        m.container.LogEvent("die")
        m.resetContainer(true)
        return err
    }
}
```

在container.monitor.Start函数中，最主要的是其中的for循环。在for循环当中，Docker通过调用m.container.daemon.Run(m.container, pipes, m.callback)来创建和分配资源。下面是Run函数代码：

```
func (d *driver) Run(c *execdriver.Command, pipes *execdriver.Pip
StartCallback) (execdriver.ExitStatus, error) {
    container, err := d.createContainer(c)
    if err != nil {
        return execdriver.ExitStatus{ExitCode: -1}, err
    }

    p := &libcontainer.Process{
        Args: append([]string{c.ProcessConfig.Entrypoint}, c.Proc
        Env:  c.ProcessConfig.Env,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            p.Wait()
            return execdriver.ExitStatus{ExitCode: -1}, err
        }
        startCallback(&c.ProcessConfig, pid)
    }

    oom := notifyOnOOM(cont)
    waitF := p.Wait
    if nss := cont.Config().Namespaces; !nss.Contains(configs.NEW
        waitF = waitInPIDHost(p, cont)
    }
    ps, err := waitF()
    if err != nil {
        execErr, ok := err.(*exec.ExitError)
        if !ok {
            return execdriver.ExitStatus{ExitCode: -1}, err
        }
        ps = execErr.ProcessState
    }
    cont.Destroy()
    _, oomKill := <-oom
    return execdriver.ExitStatus{ExitCode: utils.ExitStatus(ps.Sy
OOMKilled: oomKill}, nil
}
```

在Run函数中，Docker调用execDriver（目前默认是native）的

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    m.container.setRunning(pid)

    select {
    case <-m.startSignal:
    default:
        close(m.startSignal)
    }

    if err := m.container.ToDisk(); err != nil {
        logrus.Debugf("%s", err)
    }
}
```

当执行close(m.startSignal)之后，所有监听m.startSignal的协程都将接收到容器初始化成功的消息，包括container.waitForStart函数：

```
func (container *Container) waitForStart() error {
    container.monitor = newContainerMonitor(container, container.

    select {
    case <-container.monitor.startSignal:
    case err := <-promise.Go(container.monitor.Start):
        return err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    if err := d.createPid(container, c); err != nil {
        return nil, err
    }

    if err := d.createUTS(container, c); err != nil {
        return nil, err
    }

    if err := d.createNetwork(container, c); err != nil {
        return nil, err
    }

    if c.ProcessConfig.Privileged {
        for i := range container.Mounts {
            if container.Mounts[i].Destination == "/sys" {
                container.Mounts[i].Flags &= ^syscall.MS_RDONLY
            }
        }
        container.ReadonlyPaths = nil
        container.MaskPaths = nil
        if err := d.setPrivileged(container); err != nil {
            return nil, err
        }
    } else {
        if err := d.setCapabilities(container, c); err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
    if err := l.Validator.Validate(config); err != nil {
        return nil, newGenericError(err, ConfigInvalid)
    }
    containerRoot := filepath.Join(l.Root, id)
    if _, err := os.Stat(containerRoot); err == nil {
        return nil, newGenericError(fmt.Errorf("Container with id
    } else if !os.IsNotExist(err) {
        return nil, newGenericError(err, SystemError)
    }
    if err := os.MkdirAll(containerRoot, 0700); err != nil {
        return nil, newGenericError(err, SystemError)
    }
    return &linuxContainer{
        id:            id,
        root:          containerRoot,
        config:        config,
        initPath:      l.InitPath,
        initArgs:      l.InitArgs,
        cgroupManager: l.NewCgroupsManager(config.Cgroups, nil),
    }, nil
}
```

19. Stats命令

Stats命令大多使用在监控场景下，通过Stats命令，用户可以实时监

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    for _, n := range names {
        s := &containerStats{Name: n}
        cStats = append(cStats, s)
        go s.Collect(cli, !*noStream)
    }
    // do a quick pause so that any failed connections for contai
be
    // evicted before we display the initial or default values.
    time.Sleep(1500 * time.Millisecond)
    var errs []string
    for _, c := range cStats {
        c.mu.Lock()
        if c.err != nil {
            errs = append(errs, fmt.Sprintf("%s: %v", c.Name, c.e
        }
        c.mu.Unlock()
    }
    if len(errs) > 0 {
        return fmt.Errorf("%s", strings.Join(errs, ", "))
    }
    for range time.Tick(500 * time.Millisecond) {
        printHeader()
        toRemove := []int{}
        for i, s := range cStats {
            if err := s.Display(w); err != nil && !*noStream {
                toRemove = append(toRemove, i)
            }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        go s.Collect(cli, !*noStream)
    }
```

names保存的是需要收集信息的容器名称。在for循环中，通过go这个关键字，创建了一系列的协程来并发执行s.Collect函数。从名称上来看，这个函数应该就是负责收集容器信息的函数。

而在上面代码的最后一部分同样是一个for循环：

```
    for range time.Tick(500 * time.Millisecond) {
        printHeader()
        toRemove := []int{}
        for i, s := range cStats {
            if err := s.Display(w); err != nil && !*noStream {
                toRemove = append(toRemove, i)
            }
        }
        for j := len(toRemove) - 1; j >= 0; j-- {
            i := toRemove[j]
            cStats = append(cStats[:i], cStats[i+1:]...)
        }
        if len(cStats) == 0 {
            return nil
        }
        w.Flush()
        if *noStream {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
    defer stream.Close()
    var (
        previousCPU    uint64
        previousSystem uint64
        dec            = json.NewDecoder(stream)
        u              = make(chan error, 1)
    )
    go func() {
        for {
            var v *types.Stats
            if err := dec.Decode(&v); err != nil {
                u <- err
                return
            }
            var (
                memPercent = float64(v.MemoryStats.Usage) / float
                cpuPercent = 0.0
            )
            previousCPU = v.PreCpuStats.CpuUsage.TotalUsage
            previousSystem = v.PreCpuStats.SystemUsage
            cpuPercent = calculateCPUPercent(previousCPU, previou
            s.mu.Lock()
            s.CPUPercentage = cpuPercent
            s.Memory = float64(v.MemoryStats.Usage)
            s.MemoryLimit = float64(v.MemoryStats.Limit)
            s.MemoryPercentage = memPercent
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        if !streamStats {
            return
        }
    }
}
        case err := <-u:
            if err != nil {
                s.mu.Lock()
                s.err = err
                s.mu.Unlock()
                return
            }
        }
        if !streamStats {
            return
        }
    }
}
```

可以看到调用服务端API的动作就发生在s.Collect函数中。但有一点很奇怪，在s.Collect函数中只调用了一次API，而后就没有再调用过。调用完API之后，就是解析服务端所传回的数据，然后等待主程序500毫秒一次的显示。

从这里来看，客户端貌似只能显示最近一次的容器信息，那么通过Stats命令所看到的实时刷新效果又是如何实现的呢？客户端找不到答

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

数。

```
func (daemon *Daemon) ContainerStats(name string, stream bool, ou
    updates, err := daemon.SubscribeToContainerStats(name)
    if err != nil {
        return err
    }

    var preCpuStats types.CpuStats
    getStat := func(v interface{}) *types.Stats {
        update := v.(*execdriver.ResourceStats)
        // Retrieve the nw statistics from libnetwork and inject
        if nwStats, err := daemon.getNetworkStats(name); err == n
            update.Stats.Interfaces = nwStats
        }
        ss := convertToAPITypes(update.Stats)
        ss.PreCpuStats = preCpuStats
        ss.MemoryStats.Limit = uint64(update.MemoryLimit)
        ss.Read = update.Read
        ss.CpuStats.SystemUsage = update.SystemUsage
        preCpuStats = ss.CpuStats
        return ss
    }

    enc := json.NewEncoder(out)
    if !stream {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        if err := enc.Encode(s); err != nil {
            // TODO: handle the specific broken pipe
            daemon.UnsubscribeToContainerStats(name, updates)
            return err
        }
    }

func (daemon *Daemon) SubscribeToContainerStats(name string) (cha
    c, err := daemon.Get(name)
    if err != nil {
        return nil, err
    }
    ch := daemon.statsCollector.collect(c)
    return ch, nil
}
```

看来揭开谜题的关键就在这个for循环了。从循环条件中分析，是从updates中取出数据。而updates则是通过updates, err:=daemon.SubscribeToContainerStats(name)来定义的。

从SubscribeToContainerStats函数的返回值，可以得知原来updates是一个channel类型的变量值。那么从for v:=range updates这句代码，我们就明白是从这个channel中取数据。如果channel为空，则阻塞在这里，直到里面放入数据之后才运行。

而后的getStat则是一个函数指针：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

可以看到Stats结构体就是最终呈现给用户的数据。但是很奇怪，在服务端代码中并没有看到返回数据的逻辑，那么客户端又是如何接收到服务端数据的呢？

再切回到客户端，可以发现客户端通过dec.Decode(&v)将接收到的数据进行解码。而服务端则有一个enc.Encode(s)进行编码的过程。我们看看enc是如何获取的：

```
enc := json.NewEncoder(out)
```

enc原来是通过对out进行封装而生成的，out又是在getContainersStats函数中通过下面代码声明的：

```
out = ioutils.NewWriteFlusher(w)
```

这里的w是http.ResponseWriter。看到这里，我就明白了，虽然客户端只调用了一次API，但这个API和我们前面经常遇到的API不同，此处的API并没有断开链接，也就是此处的API是一个TCP长链接。

客户端一旦和服务端建立链接之后，服务端就通过这个链路源源不断地将数据返回给客户端。而客户端也从这个链路中接收数据，然后在对此数据进行解析，定时显示给用户。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

20. Stop命令

在Start命令一节，则我们已经知道启动一个容器只需要文件和资源两个要素就可以了。那么对于Start的逆向命令Stop，则只需将此容器运行资源销毁即可。那文件系统是否需要销毁呢？因为Stop只是暂时让容器停止运行，而不是销毁此容器。如果文件系统和运行资源同时都没有了，那么此容器也就彻底在Docker环境当中抹去了，所以Stop命令只需要销毁资源即可。

下面我们就来看看Docker是如何销毁容器资源的。

```
func (cli *DockerCli) CmdStop(args ...string) error {
    cmd := cli.Subcmd("stop", "CONTAINER [CONTAINER...]", "Stop a
and then SIGKILL after a\ngrace period", true)
    nSeconds := cmd.Int([]string{"t", "-time"}, 10, "Seconds to w
    cmd.Require(flag.Min, 1)

    cmd.ParseFlags(args, true)

    v := url.Values{}
    v.Set("t", strconv.Itoa(*nSeconds))

    var errNames []string
    for _, name := range cmd.Args() {
        _, _, err := readBody(cli.call("POST", "/containers/"+nam
        if err != nil {
            fmt.Fprintf(cli.err, "%s\n", err)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err := s.daemon.ContainerStop(vars["name"], seconds); err
        if err.Error() == "Container already stopped" {
            w.WriteHeader(http.StatusNotModified)
            return nil
        }
        return err
    }
    w.WriteHeader(http.StatusNoContent)

    return nil
}
```

而在服务端的postContainersStop函数当中，重要的逻辑是通过s.daemon.ContainerStop函数来完成的。下面我们看一下s.daemon.ContainerStop函数。

```
func (daemon *Daemon) ContainerStop(name string, seconds int) er
    container, err := daemon.Get(name)
    if err != nil {
        return err
    }
    if !container.IsRunning() {
        return fmt.Errorf("Container already stopped")
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    return nil
}
```

因为ContainerStop函数逻辑很简单，所以我们跳过ContainerStop函数直接来看Stop函数。在Stop函数当中，Docker先礼后兵，首先向容器发送SIGTERM信号。如果容器进程无法响应此信号，则再发送SIGKILL信号。这段逻辑本质是复用Kill命令的逻辑，所以读者可以翻看前面讲解Kill命令的章节来熟悉相关内容。

而后Docker通过调用container.WaitStop(time.Duration(seconds)*time.Second)来等待容器关闭。为什么需要等待呢？可以回想一下Kill命令的执行过程，容器资源销毁不是一朝一夕可以完成的事情，所以Docker需要等待一段时间，而container.WaitStop采用的方式是监听waitChan。

```
func (s *State) WaitStop(timeout time.Duration) (int, error) {
    s.Lock()
    if !s.Running {
        exitCode := s.ExitCode
        s.Unlock()
        return exitCode, nil
    }
    waitChan := s.waitChan
    s.Unlock()
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
select {
    case <-time.After(timeout):
        return fmt.Errorf("Timed out: %v", timeout)
    case <-waitChan:
        return nil
    }
```

所以说Stop和Kill命令从本质上并无太大差别。

21. Restart命令

在讲解完Start和Stop命令之后，Restart命令就非常容易理解了，Docker通过依次调用Stop命令和Start命令就完成了Restart命令。

因此Restart命令，我们就不深入讲解了，只提供一下相关源码供读者参考。

Restart客户端代码如下：

```
func (cli *DockerCli) CmdRestart(args ...string) error {
    cmd := cli.Subcmd("restart", "CONTAINER [CONTAINER...]", "Res
    nSeconds := cmd.Int([]string{"t", "-time"}, 10, "Seconds to w
container")
    cmd.Require(flag.Min, 1)

    cmd.ParseFlags(args, true)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return err
    }
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    }

    timeout, _ := strconv.Atoi(r.Form.Get("t"))

    if err := s.daemon.ContainerRestart(vars["name"], timeout); e
        return err
    }

    w.WriteHeader(http.StatusNoContent)

    return nil
}

func (daemon *Daemon) ContainerRestart(name string, seconds int)
    container, err := daemon.Get(name)
    if err != nil {
        return err
    }
    if err := container.Restart(seconds); err != nil {
        return fmt.Errorf("Cannot restart container %s: %s\n", na
    }
    container.LogEvent("restart")
    return nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

分析Tag命令仍然从客户端入手，Tag命令的客户端看起来没有什么特别之处：

```
func (cli *DockerCli) CmdTag(args ...string) error {
    cmd := cli.Subcmd("tag", "IMAGE[:TAG] [REGISTRYHOST/][USERNA
a repository", true)
    force := cmd.Bool([]string{"f", "#force", "-force"}, false, "
    cmd.Require(flag.Exact, 2)

    cmd.ParseFlags(args, true)

    var (
        repository, tag = parsers.ParseRepositoryTag(cmd.Arg(1))
        v               = url.Values{}
    )
    //Check if the given image name can be resolved
    if err := registry.ValidateRepositoryName(repository); err !=
        return err
    }
    v.Set("repo", repository)
    v.Set("tag", tag)

    if *force {
        v.Set("force", "1")
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return err
    }
    s.daemon.EventsService.Log("tag", utils.ImageReference(repo,
    w.WriteHeader(http.StatusCreated)
    return nil
}
```

在服务端的postImagesTag函数中，Docker通过解析参数，然后调用s.daemon.Repositories(). Tag(repo, tag, name, force)的方式来完成镜像的重命名。因此下一步我们就需要看看s.daemon. Repositories().Tag(repo, tag, name, force)函数了：

```
func (store *TagStore) Tag(repoName, tag, imageName string, force
    return store.SetLoad(repoName, tag, imageName, force, nil)
}

func (store *TagStore) SetLoad(repoName, tag, imageName string, f
    img, err := store.LookupImage(imageName)
    store.Lock()
    defer store.Unlock()
    if err != nil {
        return err
    }
    if tag == "" {
        tag = tags.DEFAULTTAG
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        }
    } else {
        repo = make(map[string]string)
        store.Repositories[repoName] = repo
    }
    repo[tag] = img.ID
    return store.save()
}
```

原来s.daemon.Repositories().Tag函数本质就是SetLoad函数。而在SetLoad函数中，Docker首先通过store.LookupImage(imageName)找到需要修改的镜像对象，然后通过store.Repositories [repoName]来判断，新的镜像名称是否已经在当前环境中存在了。如果不存在，也就是exists等于false的时候，直接通过repo[tag]=img.ID和return store.save()这两句代码完成重命名。

如果存在呢？Docker就会多一步操作，通过store.Repositories[repoName][tag]来判断新的镜像名称（新的标签）是否存在。如果不存在，则Docker可以认为这是给一个镜像命名一个新标签，也会通过调用epo[tag]=img.ID和return store.save()这两句代码来完成重命名。如果这个判断也存在，那么Docker就需要用户更换一个名称，除非用户通过force选项跳过这个验证。

而最后的store.save()函数就是使新数据落地的函数，我们看一下Docker是如何让这个新名称落地的。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err := json.NewDecoder(f).Decode(&store); err != nil {
        return err
    }
    return nil
}
```

在修改镜像名称之前，Docker先读取了此镜像所有的数据，然后等待名称修改完成之后，再把这个镜像所有的数据重新写回原文件当中。Docker通过这种方式就达到了给一个镜像重新命名的过程。

从Tag命令的实现过程来看，Tag的逻辑比Rename简单多了，只需通过修改JSON文件就可以完成所有的工作，而Rename需要修改JSON文件和SQLite数据库。

读者也可以借鉴Tag命令的实现过程，动态修改镜像的其他参数。

23. Top命令

我们现在已经知道通过Top命令可以获取到指定容器中正在执行的进程号，但还不知道Docker是如何获取到这些进程信息的。通过本节，读者就可以明白Docker是如何获取到容器内部进程号的，同时也可以深入理解LXC的本质。

下面，就从客户端入手来分析Top命令。

```
func (cli *DockerCli) CmdTop(args ...string) error {
    cmd := cli.Subcmd("top", "CONTAINER [ps OPTIONS]", "Display t
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return nil
}
```

Top客户端通过调用/containers/{name:.*}/top来获取容器内部进程信息，然后再将服务端返回的数据按照如下结构体进行解析展现给用户。

```
type ContainerProcessList struct {
    Processes [][]string
    Titles    []string
}
```

下面我们来看服务端API函数。

```
func (s *Server) getContainersTop(version version.Version, w http
vars map[string]string) error {
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    }

    if err := parseForm(r); err != nil {
        return err
    }

    procList, err := s.daemon.ContainerTop(vars["name"], r.Form.G
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    output, err := exec.Command("ps", strings.Split(psArgs, " ").
    if err != nil {
        return nil, fmt.Errorf("Error running ps: %s", err)
    }

    procList := &types.ContainerProcessList{}

    lines := strings.Split(string(output), "\n")
    procList.Titles = strings.Fields(lines[0])

    pidIndex := -1
    for i, name := range procList.Titles {
        if name == "PID" {
            pidIndex = i
        }
    }
    if pidIndex == -1 {
        return nil, fmt.Errorf("Couldn't find PID field in ps out
    }

    for _, line := range lines[1:] {
        if len(line) == 0 {
            continue
        }
        fields := strings.Fields(line)
        p, err := strconv.Atoi(fields[pidIndex])
        if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

逻辑层面将容器和主机系统分离开，而做的一系列包装。用Oracle数据的视图概念可以解释这些情况，我们在Docker容器中所看到的所有进程都是基于主机系统所创建出的视图信息。在容器当中运行的每一个进程都可以在主机系统当中看到相对应的实际进程。

而如何将这些进程和相对应的容器进行一一对应呢？就是凭借容器中的init根进程。在容器内部，所有的进程都是init根进程的派生子进程。虽然在主机系统中，进程号不相同，但这个派生关系仍然成立。所以只要获取到容器中init根进程所对应的实际进程号，此进程所有的子进程就都归属于此容器了。

而Top命令所完成的，就是在主机系统中匹配出所有归属于指定容器的进程。

如果理解了上面所说的这些内容，那么s.daemon.ContainerTop函数的流程就非常简单了，我们来看一下它的处理流程。

```
container, err := daemon.Get(name)
if err != nil {
    return nil, err
}

if !container.IsRunning() {
    return nil, fmt.Errorf("Container %s is not running", nam
}

pids, err := daemon.ExecutionDriver().GetPidsForContainer(con
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
"/sys/fs/cgroup/cpuacct/docker/996bb48a943762d1796a11c1f2eb263f75
,
        "cpuset":
"/sys/fs/cgroup/cpuset/docker/996bb48a943762d1796a11c1f2eb263f75a
        "devices":
"/sys/fs/cgroup/devices/docker/996bb48a943762d1796a11c1f2eb263f75
,
        "freezer":
"/sys/fs/cgroup/freezer/docker/996bb48a943762d1796a11c1f2eb263f75
,
        "memory":
"/sys/fs/cgroup/memory/docker/996bb48a943762d1796a11c1f2eb263f75a
        "perf_event":
"/sys/fs/cgroup/perf_event/docker/996bb48a943762d1796a11c1f2eb263
f2"
    },
    "namespace_paths": {
        "NEWIPC": "/proc/2748/ns/ipc",
        "NEWNET": "/var/run/docker/netns/996bb48a9437",
        "NEWNS": "/proc/2748/ns/mnt",
        "NEWPID": "/proc/2748/ns/pid",
        "NEWUSER": "/proc/2748/ns/user",
        "NEWUTS": "/proc/2748/ns/uts"
    },
```


2748就是此容器中init进程在主机环境中相对应的实际进程。但

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func (cli *DockerCli) CmdVersion(args ...string) error {
    cmd := cli.Subcmd("version", "", "Show the Docker version inf
    cmd.Require(flag.Exact, 0)

    cmd.ParseFlags(args, true)

    if dockerversion.VERSION != "" {
        fmt.Fprintf(cli.out, "Client version: %s\n", dockerversio
    }
    fmt.Fprintf(cli.out, "Client API version: %s\n", api.Version)
    fmt.Fprintf(cli.out, "Go version (client): %s\n", runtime.Ver
    if dockerversion.GITCOMMIT != "" {
        fmt.Fprintf(cli.out, "Git commit (client): %s\n", dockerv
    }
    fmt.Fprintf(cli.out, "OS/Arch (client): %s/%s\n", runtime.GOO
    if utils.ExperimentalBuild() {
        fmt.Fprintf(cli.out, "Experimental (client): true\n")
    }

    stream, _, err := cli.call("GET", "/version", nil, nil)
    if err != nil {
        return err
    }

    var v types.Version
    if err := json.NewDecoder(stream).Decode(&v); err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

信息就是远程服务端的信息了。这就是Docker的Version命令需要调用Restful API的原因。

在Version命令当中，获取客户端的代码都比较简单，基本都是直接获取，所以此处略过不提，重点看一下服务端是如何处理的。

```
func (s *Server) getVersion(version version.Version, w http.Respo
map[string]string) error {
    v := &types.Version{
        Version:    dockerversion.VERSION,
        ApiVersion: api.Version,
        GitCommit:  dockerversion.GITCOMMIT,
        GoVersion:  runtime.Version(),
        Os:         runtime.GOOS,
        Arch:       runtime.GOARCH,
    }

    if version.GreaterThanOrEqualTo("1.19") {
        v.Experimental = utils.ExperimentalBuild()
    }

    if kernelVersion, err := kernel.GetKernelVersion(); err == ni
        v.KernelVersion = kernelVersion.String()
    }

    return writeJSON(w, http.StatusOK, v)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

获取内核参数。但这个内核指的是操作系统的内核呢，还是Docker的内核呢？答案是操作系统的内核。

```
func GetKernelVersion() (*KernelVersionInfo, error) {
    var (
        err error
    )

    uts, err := uname()
    if err != nil {
        return nil, err
    }

    release := make([]byte, len(uts.Release))

    i := 0
    for _, c := range uts.Release {
        release[i] = byte(c)
        i++
    }

    release = release[:bytes.IndexByte(release, 0)]
    return ParseRelease(string(release))
}

func uname() (*syscall.Utsname, error) {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        Minor:  minor,
        Flavor: flavor,
    }, nil
}
```

GetKernelVersion函数通过调用syscall.Uname来获取当前主机运行环境的utsname，结构体如下：

```
type Utsname struct {
    Sysname    [65]int8
    Nodename   [65]int8
    Release    [65]int8
    Version    [65]int8
    Machine    [65]int8
    Domainname [65]int8
}
```

然后再通过ParseRelease解析utsname的Release属性，最终得到的是类似于4.0.9-boot2docker这样的属性值（此值根据不同的操作系统而变化）。

但是Docker最终获取的KernelVersion并没有在客户端展现给用户，Docker为什么不输出给用户，原因暂时还不得知。但通过分析Version命令可以看到，如果在Version中获取到不同层次的数据，只要对客户端代码稍加扩展就可以输出这些数据。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    // Resolve the Repository name from fqn to RepositoryInfo
    repoInfo, err := registry.ParseRepositoryInfo(remote)
    if err != nil {
        return err
    }
    // Resolve the Auth config relevant for this server
    authConfig := registry.ResolveAuthConfig(cli.configFile, repo
    // If we're not using a custom registry, we know the restrict
    // applied to repository names and can warn the user in advan
    // Custom repositories can have different rules, and we must
    // allow pushing by image ID.
    if repoInfo.Official {
        username := authConfig.Username
        if username == "" {
            username = "<user>"
        }
        return fmt.Errorf("You cannot push a \"root\" repository.
<user>/<repo> (ex: %s/%s)", username, repoInfo.LocalName)
    }

    v := url.Values{}
    v.Set("tag", tag)

    _, _, err = cli.clientRequestAttemptLogin("POST", "/images/"+
cli.out, repoInfo.Index, "push")
    return err
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    repoInfo.Index, err = config.NewIndexInfo(indexName)
    if err != nil {
        return nil, err
    }

    if repoInfo.Index.Official {
        normalizedName := repoInfo.RemoteName
        if strings.HasPrefix(normalizedName, "library/") {
            // If pull "library/foo", it's stored locally under '
            normalizedName = strings.SplitN(normalizedName, "/",
        }

        repoInfo.LocalName = normalizedName
        repoInfo.RemoteName = normalizedName
        // If the normalized name does not contain a '/' (e.g. "f
        // then it is an official repo.
        if strings.IndexRune(normalizedName, '/') == -1 {
            repoInfo.Official = true
            // Fix up remote name for official repos.
            repoInfo.RemoteName = "library/" + normalizedName
        }

        // *TODO: Prefix this with 'docker.io/'.
        repoInfo.CanonicalName = repoInfo.LocalName
    } else {
        // *TODO: Decouple index name from hostname (via registry
        repoInfo.LocalName = repoInfo.Index.Name + "/" + repoInfo
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
map[string]string) error {
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    }

    metaHeaders := map[string][]string{}
    for k, v := range r.Header {
        if strings.HasPrefix(k, "X-Meta-") {
            metaHeaders[k] = v
        }
    }
    if err := parseForm(r); err != nil {
        return err
    }
    authConfig := &cliconfig.AuthConfig{}

    authEncoded := r.Header.Get("X-Registry-Auth")
    if authEncoded != "" {
        // the new format is to handle the authConfig as a header
        authJson := base64.NewDecoder(base64.URLEncoding, strings
        if err := json.NewDecoder(authJson).Decode(authConfig); e
            // to increase compatibility to existing api it is de
            authConfig = &cliconfig.AuthConfig{}
        }
    } else {
        // the old format is supported for compatibility if there
        if err := json.NewDecoder(r.Body).Decode(authConfig); err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

通读服务端代码可以得知，Docker上传镜像调用的是s.daemon.Repositories().Push函数。而调用Push函数之前的代码，都是为了调用Push而做的准备。那么都有哪些准备呢？我们依次来看。

首先是封装Headers，Docker通过解析客户端所传来的Headers数组，将客户端中上传的Header都封装到metaHeaders当中。

```
metaHeaders := map[string][]string{}
for k, v := range r.Header {
    if strings.HasPrefix(k, "X-Meta-") {
        metaHeaders[k] = v
    }
}
```

而后，如果对方仓库需要进行用户身份认证，那么Docker也会特别封装authConfig结构体。

```
authConfig := &cliconfig.AuthConfig{}

authEncoded := r.Header.Get("X-Registry-Auth")
if authEncoded != "" {
    // the new format is to handle the authConfig as a header
    authJson := base64.NewDecoder(base64.URLEncoding, strings
    if err := json.NewDecoder(authJson).Decode(authConfig); e
        // to increase compatibility to existing api it is de
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    var (
        sf = streamformatter.NewJSONStreamFormatter()
    )

    repoInfo, err := s.registryService.ResolveRepository(localNar
    if err != nil {
        return err
    }

    if _, err := s.poolAdd("push", repoInfo.LocalName); err != ni
        return err
    }
    defer s.poolRemove("push", repoInfo.LocalName)

    endpoint, err := repoInfo.GetEndpoint(imagePushConfig.MetaHea
    if err != nil {
        return err
    }
    tr := transport.NewTransport(
        registry.NewTransport(registry.NoTimeout, endpoint.IsSecu
        registry.DockerHeaders(imagePushConfig.MetaHeaders)...,
    )
    client := registry.HTTPClient(tr)
    r, err := registry.NewSession(client, imagePushConfig.AuthCon
    if err != nil {
        return err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return err
    }
    s.eventsService.Log("push", repoInfo.LocalName, "")
    return nil
}
```

这个函数逻辑上有些复杂，需要细心进行分析。首先来看下面这一行代码：

```
repoInfo, err := s.registryService.ResolveRepository(localName)
```

此行代码返回的repoInfo非常重要，可以说此变量完成主导了Push函数后面所有的逻辑。repoInfo变量表示的是RepositoryInfo结构体，此结构体定义如下：

```
type RepositoryInfo struct {
    Index         *IndexInfo
    RemoteName    string
    LocalName     string
    CanonicalName string
    Official      bool
}

type IndexInfo struct {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

这个函数是直接返回另外一个函数的结果，如下所示：

```
func (s *Service) ResolveRepository(name string) (*RepositoryInfo
    return s.Config.NewRepositoryInfo(name)
}
```

所以需要分析s.Config.NewRepositoryInfo(name)函数才能得知具体逻辑，继续往下看：

```
func (config *ServiceConfig) NewRepositoryInfo(reposName string)
    if err := validateNoSchema(reposName); err != nil {
        return nil, err
    }

    indexName, remoteName := splitReposName(reposName)
    if err := validateRemoteName(remoteName); err != nil {
        return nil, err
    }

    repoInfo := &RepositoryInfo{
        RemoteName: remoteName,
    }

    var err error
    repoInfo.Index, err = config.NewIndexInfo(indexName)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    return repoInfo, nil
}
```

validateNoSchema用来验证给定的镜像名称是否含有“://”。然后又通过splitReposName函数对镜像名称进行分割，如果用户提供的镜像名称为“192.168.73.12:5001/ubuntu:14.04”这种格式，则返回的indexName就是192.168.73.12:5001，remoteName是ubuntu。而如果用户提供的镜像名称为“<user>/ubuntu:14.04”这种格式时，返回的indexName则是<user>/，remoteName仍然是ubuntu。

Docker这么做是为了区分目标仓库是官方仓库还是私有仓库，区分的方法则是通过比较indexName是否与index.docker.io相等来完成的：

```
    if val == "index."+IndexServerName() {
        val = IndexServerName()
    }
```

Docker为什么会关心目标仓库类型呢？因为这涉及后面Docker如何来生成endpoint。下面来看Docker是如何来生成endpoint的。

生成endpoint是通过endpoint, err:=repoInfo.GetEndpoint(imagePushConfig.MetaHeaders)来完成的。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        trimmedAddress string
        err            error
    )

    if !strings.HasPrefix(address, "http") {
        address = "https://" + address
    }

    trimmedAddress, endpoint.Version = scanForAPIVersion(address)

    if endpoint.URL, err = url.Parse(trimmedAddress); err != nil
        return nil, err
    }
    endpoint.IsSecure = secure
    tr := NewTransport(ConnectTimeout, endpoint.IsSecure)
    endpoint.client = HTTPClient(transport.NewTransport(tr, Docke
    return endpoint, nil
}
```

上面四个函数中，newEndpoint(index.GetAuthConfigKey(), index.Secure, metaHeaders)是最重要的一个函数。index.GetAuthConfigKey()如果是官方仓库，则返回https://index.docker.io/v1/；如果是私有仓库时则返回类似于192.168.73.12:5001这样的URL。

在newEndpoint函数中，会使用这个URL地址来解析endpoint的URL

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            alwaysSetBasicAuth = true
        }
    }

    client.Transport = AuthTransport(client.Transport, authConfig

    jar, err := cookiejar.New(nil)
    if err != nil {
        return nil, errors.New("cookiejar.New is not supposed to
    }
    client.Jar = jar

    return r, nil
}
```

如果是私有仓库，则Docker会执行Endpoint.Ping()，此函数用来确定私有仓库版本号：

```
func (e *Endpoint) Ping() (RegistryInfo, error) {
    switch e.Version {
    case APIVersion1:
        return e.pingV1()
    case APIVersion2:
        return e.pingV2()
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if repoInfo.Index.Official || endpoint.Version == registry.AP
        err := s.pushV2Repository(r, localRepo, imagePushConfig.O
Tag, sf)
        if err == nil {
            s.eventsService.Log("push", repoInfo.LocalName, "")
            return nil
        }

        if err != ErrV2RegistryUnavailable {
            return fmt.Errorf("Error pushing to registry: %s", er
        }
    }

    if err := s.pushRepository(r, imagePushConfig.OutStream, repo
Tag, sf); err != nil {
        return err
    }
```

如果版本为V2，则调用pushV2Repository，反之则调用pushRepository。这两个函数在实现方式上变化较大，所以需要分开来说。首先说pushRepository函数：

```
func (s *TagStore) pushRepository(r *registry.Session, out
RepositoryInfo, localRepo map[string]string,tag string, sf *strea
{
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
sf, r); err != nil {
            return err
        }
    }
    _, err = r.PushImageJSONIndex(repoInfo.RemoteName, imageIndex
    return err
}
```

在pushRepository函数中，通过s.getImageList获取指定镜像的所有子镜像文件层。例如，我们用做实例的ubuntu镜像，通过s.getImageList返回的数据是：

```
d3a1f33e8a5a513092f01bb7eb1c2abf4d711e5105390a3fe1ae2248cfde1391
c22013c8472965aa5b62559f2b540cd440716ef149756e7b958a1b2aba421e87
d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb82
91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d920813e267c
```

这么做的目的是因为在Docker中，一个镜像是由不同子镜像叠加组成的，而这些子镜像又都是可以被复用的。为了减少上传数据量，提高上传效率，Docker在上传镜像之前会将子镜像列表提供给目标仓库，然后只上传仓库中明确告之不存在的镜像层。

这些操作都封装在pushImageToEndpoint函数中：

```
func (s *TagStore) pushImageToEndpoint(endpoint string, out io.Wr
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        imageData <- imagePushData{
            id:       id,
            endpoint: endpoint,
            tokens:   repo.Tokens,
        }
    }
    // close the channel to notify the workers that there will be
    close(imageData)
    wg.Wait()
    close(imagesToPush)
    // wait for all the images that require pushes to be collecte
    shouldPush := <-pushes
    // finish by pushing any images and tags to the endpoint.  Th
    // is very important that is why we are still iterating over
    for _, id := range imageIDs {
        if _, push := shouldPush[id]; push {
            if _, err := s.pushImage(r, out, id, endpoint, repo.T
                return err
            }
        }
        for _, tag := range tags[id] {
            out.Write(sf.FormatStatus("", "Pushing tag for rev [%
endpoint+"repositories/"+remoteName+"/tags/"+tag))
            if err := r.PushRegistryTag(remoteName, id, tag, endp
                return err
            }
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

Docker循环取出待上传的镜像文件层，然后依次放入imageData这个channel中，worker会在lookupImageOnEndpoint中取出imageData中的数据。如果workder都在忙，那么for循环中放入imageData就会发生阻塞。只有worker产生空闲之后，阻塞才会消失。

通过这种方式，可让worker不停地从imageData中取出数据，直到所有需要上传的数据都查询完。最后Docker调用close(imageData)来通知所有的worker已经没有需要查询的数据了，然后workder才可以退出。

当所有的镜像文件层都查询完毕后，Docker也就知道了哪些镜像需要进行上传，而后Docker就开始依次进行上传操作：

```
    for _, id := range imageIDs {
        if _, push := shouldPush[id]; push {
            if _, err := s.pushImage(r, out, id, endpoint, repo.T
                return err
            }
        }
        for _, tag := range tags[id] {
            out.Write(sf.FormatStatus("", "Pushing tag for rev [%
endpoint+"repositories/"+remoteName+"/tags/"+tag))
            if err := r.PushRegistryTag(remoteName, id, tag, endp
                return err
            }
        }
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    layerData, err := s.graph.TempLayerArchive(imgID, sf, out)
    if err != nil {
        return "", fmt.Errorf("Failed to generate layer archive:
    }
    defer os.RemoveAll(layerData.Name())

    // Send the layer
    logrus.Debugf("rendered layer for %s of [%d] size", imgData.I

    checksum, checksumPayload, err := r.PushImageLayerRegistry(im
        progressreader.New(progressreader.Config{
            In:        layerData,
            Out:       out,
            Formatter: sf,
            Size:      int(layerData.Size),
            NewLines:  false,
            ID:        stringid.TruncateID(imgData.ID),
            Action:    "Pushing",
        }), ep, jsonRaw)
    if err != nil {
        return "", err
    }
    imgData.Checksum = checksum
    imgData.ChecksumPayload = checksumPayload
    // Send the checksum
    if err := r.PushImageChecksumRegistry(imgData, ep); err != ni
        return "", err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return fmt.Errorf("error getting registry endpoint: %s",
    }

    tags, err := s.getImageTags(localRepo, tag)
    if err != nil {
        return err
    }
    if len(tags) == 0 {
        return fmt.Errorf("No tags to push for %s", repoInfo.Loca
    }

    auth, err := r.GetV2Authorization(endpoint, repoInfo.RemoteNa
    if err != nil {
        return fmt.Errorf("error getting authorization: %s", err)
    }

    for _, tag := range tags {
        logrus.Debugf("Pushing repository: %s:%s", repoInfo.Canon

        layerId, exists := localRepo[tag]
        if !exists {
            return fmt.Errorf("tag does not exist: %s", tag)
        }

        layer, err := s.graph.Get(layerId)
        if err != nil {
            return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
// Schema version 1 requires layer ordering from top to r
for i, layer := range layers {
    logrus.Debugf("Pushing layer: %s", layer.ID)

    if layer.Config != nil && metadata.Image != layer.ID
        if err := runconfig.Merge(&metadata, layer.Config
            return err
        }
    }
    jsonData, err := layer.RawJson()
    if err != nil {
        return fmt.Errorf("cannot retrieve the path for %
    }

    checksum, err := layer.GetCheckSum(s.graph.ImageRoot(
    if err != nil {
        return fmt.Errorf("error getting image checksum:
    }
    var exists bool
    if len(checksum) > 0 {
        dgst, err := digest.ParseDigest(checksum)
        if err != nil {
            return fmt.Errorf("Invalid checksum %s: %s",
        }

        // Call mount blob
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        if err := validateManifest(m); err != nil {
            return fmt.Errorf("invalid manifest: %s", err)
        }

        logrus.Debugf("Pushing %s:%s to v2 repository", repoInfo.
        mBytes, err := json.MarshalIndent(m, "", "   ")
        if err != nil {
            return err
        }
        js, err := libtrust.NewJSONSignature(mBytes)
        if err != nil {
            return err
        }

        if err = js.Sign(s.trustKey); err != nil {
            return err
        }

        signedBody, err := js.PrettySignature("signatures")
        if err != nil {
            return err
        }
        logrus.Infof("Signed manifest for %s:%s using daemon's ke
s.trustKey.KeyID())

        // push the manifest
        digest, err := r.PutV2ImageManifest(endpoint, repoInfo.Re
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            }

            if layersSeen[layer.ID] {
                break
            }
            layers = append(layers, layer)
            layersSeen[layer.ID] = true
        }
```

Docker逐层获取每个文件层的父级，经过循环处理之后layers保存的数据变为：

```
d3a1f33e8a5a513092f01bb7eb1c2abf4d711e5105390a3fe1ae2248cfde1391
c22013c8472965aa5b62559f2b540cd440716ef149756e7b958a1b2aba421e87
d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb82
91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d920813e267c
```

达到的效果和getImages是一样的。如果Docker复用getImages函数那么逻辑会更为简单一些。

而后的处理就和V1中的处理大同小异了。首先通过layer.GetCheckSum(s.graph.ImageRoot (layer.ID))来计算校验码，而后再向目标仓库提出查询请求，也就是判断此文件层是否存在于目标仓库。如果不存在则调用pushV2Image上传指定的文件层。而上传的方式，也是通过将文件层数据做成压缩包，然后将压缩包上传到目标仓库中。当

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func (cli *DockerCli) CmdPull(args ...string) error {
    cmd := cli.Subcmd("pull", "NAME[:TAG|@DIGEST]", "Pull an imag
true)
    allTags := cmd.Bool([]string{"a", "-all-tags"}, false, "Downl
repository")
    cmd.Require(flag.Exact, 1)

    cmd.ParseFlags(args, true)

    var (
        v         = url.Values{}
        remote    = cmd.Arg(0)
        newRemote = remote
    )
    taglessRemote, tag := parsers.ParseRepositoryTag(remote)
    if tag == "" && !*allTags {
        newRemote = utils.ImageReference(taglessRemote, tags.DEFA
    }
    if tag != "" && *allTags {
        return fmt.Errorf("tag can't be used with --all-tags/-a")
    }

    v.Set("fromImage", newRemote)

    // Resolve the Repository name from fqn to RepositoryInfo
    repoInfo, err := registry.ParseRepositoryInfo(taglessRemote)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
    var (
        image = r.Form.Get("fromImage")
        repo  = r.Form.Get("repo")
        tag   = r.Form.Get("tag")
    )
    authEncoded := r.Header.Get("X-Registry-Auth")
    authConfig := &cliconfig.AuthConfig{}
    if authEncoded != "" {
        authJson := base64.NewDecoder(base64.URLEncoding, strings
        if err := json.NewDecoder(authJson).Decode(authConfig); e
            authConfig = &cliconfig.AuthConfig{}
        }
    }
    var (
        err    error
        output = ioutils.NewWriteFlusher(w)
    )
    w.Header().Set("Content-Type", "application/json")
    if image != "" { //pull
        if tag == "" {
            image, tag = parsers.ParseRepositoryTag(image)
        }
        metaHeaders := map[string][]string{}
        for k, v := range r.Header {
            if strings.HasPrefix(k, "X-Meta-") {
                metaHeaders[k] = v
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
  if err != nil {
    if !output.Flushed() {
      return err
    }
    sf := streamformatter.NewJSONStreamFormatter()
    output.Write(sf.FormatError(err))
  }

  return nil

}
```

从上面代码可以看出，其中有两个处理流程：import和pull。现在只关心如何执行Pull命令，所以转入Pull函数中。

```
func (s *TagStore) Pull(image string, tag string, imagePullConfig
    var (
        sf = streamformatter.NewJSONStreamFormatter()
    )
    // Resolve the Repository name from fqn to RepositoryInfo
    repoInfo, err := s.registryService.ResolveRepository(image)
    if err != nil {
        return err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            logrus.Errorf("Error configuring mirrors: %s", err)
            return err
        }

        if v2mirrorEndpoint != nil {
            logrus.Debugf("Attempting to pull from v2 mirror: %s"
            return s.pullFromV2Mirror(v2mirrorEndpoint, v2mirrorR
logName)
        }
    }

    logrus.Debugf("pulling  image  from  host  %q  with  remote
repoInfo.RemoteName)

    endpoint, err := repoInfo.GetEndpoint(imagePullConfig.MetaHea
    if err != nil {
        return err
    }

    tr := transport.NewTransport(
        registry.NewTransport(registry.ReceiveTimeout, endpoint.I
        registry.DockerHeaders(imagePullConfig.MetaHeaders)...,
    )
    client := registry.HTTPClient(tr)
    r, err := registry.NewSession(client, imagePullConfig.AuthCon
    if err != nil {
        return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return nil
}
```

第一感觉是不是认为Pull函数和我们在分析Push命令时遇到的Push函数非常类似？

首先对Pull函数进行分层，这种方法对于研读第三方源码尤其有效。

Pull函数首先获取目标仓库信息，这一点非常容易理解，因为无论是上传还是下载，首要信息自然是获取目标地址。然后Pull函数开始根据目标仓库信息构建endpoint，这个流程好像和Push函数流程类似。最后Pull函数按照V2版本、V1版本的顺序尝试下载镜像。

Pull的执行流程看来和Push的流程大同小异，只不过到了最后一步，将push的上传动作替换成了下载动作而已。如果真是如此，那么Pull函数我们就相当于已经掌握十之八九了。下面就验证一下Pull和Push的相似度。

首先获取目标仓库地址，Pull函数通过repoInfo, err:=s.registryService.ResolveRepository(image)来获取目标仓库信息。然后再通过validateRepoName(repoInfo.LocalName)来验证仓库信息。通过翻看Push函数的逻辑，发现Push函数也是使用相同的函数来完成相同的事情。

继续往下看，Pull函数判断如果目标仓库是Docker官方镜像仓库，并且Docker官方仓库使用的是V2版本，那么就直接调用pullFromV2Mirror来下载镜像。Docker为什么需要进行这个判断呢？这

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    for tag, id := range tagsList {
        repoData.ImgList[id] = &registry.ImgData{
            ID:       id,
            Tag:      tag,
            Checksum: "",
        }
    }

    logrus.Debugf("Registering tags")
    // If no tag has been specified, pull them all
    if askedTag == "" {
        for tag, id := range tagsList {
            repoData.ImgList[id].Tag = tag
        }
    } else {
        // Otherwise, check that the tag exists and use only that
        id, exists := tagsList[askedTag]
        if !exists {
            return fmt.Errorf("Tag %s not found in repository %s"
        }
        repoData.ImgList[id].Tag = askedTag
    }
    errors := make(chan error)

    layersDownloaded := false
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            out.Write(sf.FormatProgress(stringid.TruncateID(img.I
from %s", img.Tag, repoInfo.CanonicalName), nil))
            success := false
            var lastErr, err error
            var isDownloaded bool
            for _, ep := range repoInfo.Index.Mirrors {
                // Ensure endpoint is v1
                ep = ep + "v1/"
                out.Write(sf.FormatProgress(stringid.TruncateID(i
(%s) from %s, mirror: %s", img.Tag, repoInfo.CanonicalName, ep),
                if isDownloaded, err = s.pullImage(r, out, img.ID
                    // Don't report errors when pulling from mirr
                    logrus.Debugf("Error pulling image (%s) from
repoInfo.CanonicalName, ep, err)
                    continue
                }
                layersDownloaded = layersDownloaded || isDownloac
                success = true
                break
            }
            if !success {
                for _, ep := range repoData.Endpoints {
                    out.Write(sf.FormatProgress(stringid.Truncate
image (%s) from %s, endpoint: %s", img.Tag, repoInfo.CanonicalNam
                    if isDownloaded, err = s.pullImage(r, out, im
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        if err := <-errors; err != nil {
            lastError = err
        }
    }
    if lastError != nil {
        return lastError
    }

    for tag, id := range tagsList {
        if askedTag != "" && tag != askedTag {
            continue
        }
        if err := s.Tag(repoInfo.LocalName, tag, id, true); err
            return err
        }
    }

    requestedTag := repoInfo.CanonicalName
    if len(askedTag) > 0 {
        requestedTag = utils.ImageReference(repoInfo.CanonicalNar
    }
    WriteStatus(requestedTag, out, sf, layersDownloaded)
    return nil
}
```

在Push当中，我们是获取到指定镜像的子镜像列表，然后逐个把子

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    for tag, id := range tagsList {
        repoData.ImgList[id] = &registry.ImgData{
            ID:       id,
            Tag:      tag,
            Checksum: "",
        }
    }

    logrus.Debugf("Registering tags")
    // If no tag has been specified, pull them all
    if askedTag == "" {
        for tag, id := range tagsList {
            repoData.ImgList[id].Tag = tag
        }
    } else {
        // Otherwise, check that the tag exists and use only that
        id, exists := tagsList[askedTag]
        if !exists {
            return fmt.Errorf("Tag %s not found in repository %s"
        }
        repoData.ImgList[id].Tag = askedTag
    }

    errors := make(chan error)
```

这段代码从目标仓库中取回了镜像所有标签信息。而后再判断如果

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    // FIXME: Launch the getRemoteImage() in goroutines

    layersDownloaded := false
    for i := len(history) - 1; i >= 0; i-- {
        id := history[i]

        // ensure no two downloads of the same layer happen at th
        if c, err := s.poolAdd("pull", "layer:"+id); err != nil {
            logrus.Debugf("Image (id: %s) pull is already running
            <-c
        }
        defer s.poolRemove("pull", "layer:"+id)

        if !s.graph.Exists(id) {
            out.Write(sf.FormatProgress(stringid.TruncateID(id),
            var (
                imgJSON []byte
                imgSize int
                err     error
                img     *image.Image
            )
            retries := 5
            for j := 1; j <= retries; j++ {
                imgJSON, imgSize, err = r.GetRemoteImageJSON(id,
                if err != nil && j == retries {
                    out.Write(sf.FormatProgress(stringid.Truncate
layers", nil))
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
                if uerr, ok := err.(*url.Error); ok {
                    err = uerr.Err
                }
                if terr, ok := err.(net.Error); ok && terr.Timeou
                    time.Sleep(time.Duration(j) * 500 * time.Mill
                    continue
                } else if err != nil {
                    out.Write(sf.FormatProgress(stringid.Truncate
layers", nil))
                    return layersDownloaded, err
                }
                layersDownloaded = true
                defer layer.Close()

                err = s.graph.Register(img,
                    progressreader.New(progressreader.Config{
                        In:        layer,
                        Out:       out,
                        Formatter: sf,
                        Size:      imgSize,
                        NewLines:  false,
                        ID:        stringid.TruncateID(id),
                        Action:    "Downloading",
                    }))
                if terr, ok := err.(net.Error); ok && terr.Timeou
                    time.Sleep(time.Duration(j) * 500 * time.Mill
                    continue
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

如果s.graph.Exists(id)返回false，就说明此文件层在本地不存在，Docker就会明白此文件层是需要下载到本地的。Docker会调用imgJSON, imgSize, err=r.GetRemoteImageJSON(id, endpoint)来获取文件层的元数据信息，然后再调用layer, err:=r.GetRemoteImageLayer(img.ID, endpoint, int64(imgSize))来获取此文件层的压缩包，最后通过s.graph.Register完成文件层的注册工作。

Docker同时考虑到了网络不佳的场景，因此代码中设置了一旦上述下载环节出错的话，会间隔500毫秒重新执行下载。如果重试5次后仍然失败，就直接报错退出。上面的步骤也印证了我们对于Pull命令执行流程的第二步和第三步。

当分析完Pull命令之后，就可以看到Pull命令其实就是Push命令的翻版，只不过执行顺序和Push命令完全相反。Push是先获取目标仓库中存在的子镜像，然后再打包上传。而Pull则是判断本地不存在的子镜像，再依次下载。

在实际生产环境中，Pull和Push命令是使用最多最广的命令，同时也是优化空间最大的一组命令。当前Docker在此两个命令中，对于用户认证、多协程并发、断点续传都仅仅只满足了最基本的使用需求，无法应对实际使用过程中的复杂场景。

当读者需要对Pull和Push命令进行扩展时，再翻看本节内容或许就更为容易理解。

3. Search命令

Search命令是在远程仓库中检索镜像的命令。通过Search命令，用户可以查询远程仓库中是否存在指定镜像，然后再配合Pull或者Push命

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    rdr, _, err := cli.clientRequestAttemptLogin("GET", "/images/
repoInfo.Index, "search")
    if err != nil {
        return err
    }

    results := ByStars{}
    if err := json.NewDecoder(rdr).Decode(&results); err != nil {
        return err
    }

    sort.Sort(sort.Reverse(results))

    w := tabwriter.NewWriter(cli.out, 10, 1, 3, ' ', 0)
    fmt.Fprintf(w, "NAME\tDESCRIPTION\tSTARS\tOFFICIAL\tAUTOMATED
    for _, res := range results {
        if ((*automated || *trusted) && (!res.IsTrusted && !res.I
res.StarCount) {
            continue
        }
        desc := strings.Replace(res.Description, "\n", " ", -1)
        desc = strings.Replace(desc, "\r", " ", -1)
        if !*noTrunc && len(desc) > 45 {
            desc = stringutils.Truncate(desc, 42) + "..."
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        authEncoded = r.Header.Get("X-Registry-Auth")
        headers     = map[string][]string{}
    )

    if authEncoded != "" {
        authJson := base64.NewDecoder(base64.URLEncoding, strings
        if err := json.NewDecoder(authJson).Decode(&config); err
            // for a search it is not an error if no auth was giv
            // to increase compatibility with the existing api it
            config = &cliconfig.AuthConfig{}
        }
    }
    for k, v := range r.Header {
        if strings.HasPrefix(k, "X-Meta-") {
            headers[k] = v
        }
    }
    query, err := s.daemon.RegistryService.Search(r.Form.Get("ter
    if err != nil {
        return err
    }
    return json.NewEncoder(w).Encode(query.Results)
}
```

在服务端当中，遵循Docker标准编程模型，采取先解析参数再调用处理函数的方式，将Search命令所有的逻辑都封装到了

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

中都讲解过，这里不再赘述。下面看看r.SearchRepositories函数：

```
func (r *Session) SearchRepositories(term string) (*SearchResults
    logrus.Debugf("Index server: %s", r.indexEndpoint)
    u := r.indexEndpoint.VersionString(1) + "search?q=" + url.Que
    res, err := r.client.Get(u)
    if err != nil {
        return nil, err
    }
    defer res.Body.Close()
    if res.StatusCode != 200 {
        return  nil,  httputils.NewHTTPRequestError(fmt.Sprintf("
res.StatusCode), res)
    }
    result := new(SearchResults)
    return result, json.NewDecoder(res.Body).Decode(result)
}
```

在SearchRepositories函数中，Docker封装好访问对方仓库的URL，来调用远程仓库提供的Restful API。目前来说，如果访问官方远程仓库，那么封装好的URL就是：https://index.docker.io/v1/search?q=。如果是访问第三方仓库，那么Docker会按照Pull命令中相同的处理流程，优先使用V2版本的API；如果V2版本不适用，再切换使用V1版本的API。

远程仓库则会根据上传的镜像名称进行模糊匹配，然后将匹配的结果按照如下结构体返回给Docker Daemon：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

1. Login命令

在日常使用Docker过程中，需要登录到Docker官方提供的Hub仓库。这时想必大家都明白可以通过Login命令来登录，但在Login命令的用法中有这样一行提示：

```
Register or log in to a Docker registry server, if no server is s
```

从这行提示中，可以得知Login不仅可以用来登录到Docker官方仓库中，也可以登录到用户指定的仓库中。那么Login是如何做到的呢？在本节，我们来看看Docker是如何使用用户认证信息的。

按照我们对于Login的使用理解，Login无非就是收集用户名、用户口令和用户E-mail地址。然后发送到指定的仓库进行校验。那么Login的客户端应该不会很复杂，真实情况又是什么样的呢？请看下面Login的客户端代码：

```
func (cli *DockerCli) CmdLogin(args ...string) error {
    cmd := cli.Subcmd("login", "[SERVER]", "Register or log in to
is\nspecified \""+registry.IndexServerAddress()+"\" is the defaul
    cmd.Require(flag.Max, 1)

    var username, password, email string

    cmd.StringVar(&username, []string{"u", "-username"}, "", "Use
    cmd.StringVar(&password, []string{"p", "-password"}, "", "Pas
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if !ok {
        authconfig = cliconfig.AuthConfig{}
    }

    if username == "" {
        promptDefault("Username", authconfig.Username)
        username = readInput(cli.in, cli.out)
        username = strings.Trim(username, " ")
        if username == "" {
            username = authconfig.Username
        }
    }
    if username != authconfig.Username {
        if password == "" {
            oldState, err := term.SaveState(cli.inFd)
            if err != nil {
                return err
            }
            fmt.Fprintf(cli.out, "Password: ")
            term.DisableEcho(cli.inFd, oldState)

            password = readInput(cli.in, cli.out)
            fmt.Fprint(cli.out, "\n")

            term.RestoreTerminal(cli.inFd, oldState)
            if password == "" {
                return fmt.Errorf("Error : Password Required")
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        delete(cli.configFile.AuthConfigs, serverAddress)
        if err2 := cli.configFile.Save(); err2 != nil {
            fmt.Fprintf(cli.out, "WARNING: could not save config
        }
        return err
    }
    if err != nil {
        return err
    }
         var response types.AuthResponse
    if err := json.NewDecoder(stream).Decode(&response); err != n
        // Upon error, remove entry
        delete(cli.configFile.AuthConfigs, serverAddress)
        return err
    }

    if err := cli.configFile.Save(); err != nil {
        return fmt.Errorf("Error saving config file: %v", err)
    }
    fmt.Fprintf(cli.out, "WARNING: login credentials saved in %s\

    if response.Status != "" {
        fmt.Fprintf(cli.out, "%s\n", response.Status)
    }
    return nil
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            fmt.Fprintf(cli.out, "%s: ", prompt)
        } else {
            fmt.Fprintf(cli.out, "%s (%s): ", prompt, configDefau
        }
    }

    readInput := func(in io.Reader, out io.Writer) string {
        reader := bufio.NewReader(in)
        line, _, err := reader.ReadLine()
        if err != nil {
            fmt.Fprintln(out, err.Error())
            os.Exit(1)
        }
        return string(line)
    }
```

而后Login定义了两个函数：promptDefault和readInput。

这两个函数组合起来就可以完成从命令行读入用户手工输入数据的功能。但是在Login命令开始时，已经定义了三个变量来读取数据，为什么这里还会允许用户在命令行输入数据呢？这是因为Docker允许读者自行保存私有仓库的用户信息，如果用户在使用Login时没有传入用户认证参数，那么Docker就会尝试读取用户自定义保存的数据，完成这个步骤是依靠下面的一行代码：

```
    authconfig, ok := cli.configFile.AuthConfigs[serverAddress]
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
type ConfigFile struct {
    AuthConfigs map[string]AuthConfig `json:"auths"`
    HttpHeaders map[string]string     `json:"HttpHeaders,omitempt
    filename    string                // Note: not serialized - t
}
```

在config.json文件中还保存有AuthConfig结构体，此结构体定义如下：

```
type AuthConfig struct {
    Username      string `json:"username,omitempty"`
    Password      string `json:"password,omitempty"`
    Auth          string `json:"auth"`
    Email         string `json:"email"`
    ServerAddress string `json:"serveraddress,omitempty"`
}
```

将上述json结构体翻译为容易理解的格式，就是下面的样子：

```
{
    "auths": {
        "RegistryName": {
            "username": "XXXXXXX",
            "password": "XXXXXXX",
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if err != nil {
        return err
    }
    status, err := s.daemon.RegistryService.Auth(config)
    if err != nil {
        return err
    }
    return writeJSON(w, http.StatusOK, &types.AuthResponse{
        Status: status,
    })
}
```

可以看到Docker在处理Login时主要逻辑都放在了s.daemon.RegistryService.Auth(config)这个函数上。继续跟踪代码，看看Docker是如何进行Auth的。

```
func (s *Service) Auth(authConfig *cliconfig.AuthConfig) (string,
    addr := authConfig.ServerAddress
    if addr == "" {
        addr = IndexServerAddress()
    }
    index, err := s.ResolveIndex(addr)
    if err != nil {
        return "", err
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

在Login函数中，又出现了一次版本判断。Docker会根据endpoint中的版本信息，选择登录方式。loginV1和loginV2不仅仅是版本发生了变化，判断逻辑也完全不同。在V1版本中，客户端（也就是Docker Daemon）只需将用户认证信息通过post方式发往仓库校验，然后判断仓库返回的http响应码即可。但在V2版本中，有两种认证方式：基本认证和令牌认证。基本认证仍然是发送请求，然后判断HTTP响应码。在令牌认证中，是将获取到的令牌发送给仓库，然后判断仓库返回的认证码。

因为这两种认证方式都是在Docker仓库应用中完成的，所以我们这里就不深入讲解了。感兴趣的读者，可以前往Docker的Github地址查看registry认证相关资料。

Login函数虽然想起来逻辑非常简单，但真正实施起来就会发现，一个Login会涉及很多方面。在本节当中，我们学习了如何将仓库认证信息保存到config.json，同时学习了Docker Login命令如何读取此配置文件。在服务端方面，我们看到Docker仓库中的V1版本和V2版本各有哪些认证方式。

这些内容对于需要基于Docker仓库进行认证定制化开发的读者会有一些帮助。如果需要深入了解Docker仓库是如何进行认证以及如何工作的，请参阅Docker关于仓库的文档资料。

2. Logout命令

在本节，我们会看到Docker CLI命令当中最简单的命令##Logout命令。分析了Docker这么多的CLI命令之后，Logout命令绝对是最简单的一个命令。下面看Logout命令的所有的源码：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

Login命令的分析过程，我们得知Login的过程其实就是Docker Daemon向指定的Docker仓库进行用户认证的过程。如果认证通过，那么Docker Daemon就允许用户向Docker仓库上传或者下载指定镜像数据。

但是Logout并没有向Docker仓库做迁出。一方面是Docker仓库没有此类的API，另一方面是没必要这么做。因为用户和Docker仓库之间的交互都是无状态的，所以在Docker仓库中，并没有保存用户的状态信息。用户是否处于登录状态，仓库不知道也不关心。因此Docker仓库也就没有再提供类似于签出的API。

那么Logout命令做了哪些事情呢？观察上面的代码可发现：当用户执行Logout时，其实就是删除了Login成功后所保存的用户认证信息。当前环境中没有了这些认证信息，再向Docker仓库上传数据时就会报错。这样就达到了签出的目的。

Logout的命令逻辑非常简单，留给读者自己分析吧。

6.4 Docker资源命令

1. Export命令

Export命令，顾名思义可以用来导出容器。这个命令与其他命令相比，实现起来非常简单。在前面的基础命令章节中，介绍过使用Export可以将目前Docker环境中已有的容器导出为tar格式的压缩包。当我们拿到tar格式压缩包之后，就可以通过其他方式分发给其他用户。

如果简要给Export命令下一个定义，就是将指定容器所存储的数据打包传递到客户端。因此，Export命令的实现逻辑大致可分为以下几

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        }
    } else if cli.isTerminalOut {
        return errors.New("Cowardly refusing to save to a termina
    }

    image := cmd.Arg(0)
    sopts := &streamOpts{
        rawTerminal: true,
        out:         output,
    }
    if err := cli.stream("GET", "/containers/"+image+"/export", s
        return err
    }

    return nil
}
```

通过前面几个命令的代码解析，我们大致可以得到Docker命令的实现思路：客户端负责解析命令行传来的参数，然后调用服务端提供的remote API来实现具体功能。因此一个命令是否复杂，可以通过其调用服务端remote API的数量来判断。

在Export的客户端命令中，可以看到只在最后调用了一次remote API，因此说明了Export命令实现逻辑较为简单。在客户端中通过cmd包解析命令行参数，然后封装了一个streamOpts对象。这个对象之前我们见过，它封装了客户端的输出流，可以用来接收数据。在客户端代码的

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        } else {
            _, err = stdcopy.StdCopy(stdout, stderr, body)
        }
        logrus.Debugf("[stream] End of stdout")
        return err
    }
    return nil
}
```

通过上面的代码，可以清晰看到cli.steam和其他两个函数的不同。cli.stream函数主要是接收服务端所传来的二进制流，也就是I/O流。而cli.call则是发起一个同步短链接，接收字符串数据。

为什么Export会选用cli.steam来接收二进制流呢？回看上面列出的Export实现步骤，在Export执行的最后，服务端需要发送一个tar格式的压缩包给客户端，这个压缩包必然是通过二进制流的形式发送给客户端的，所以Export在客户端才会调用cli.steam函数来接收并处理服务端所传回的二进制流。

客户端现在算是分析完了，我们看一下服务端逻辑。因为客户端仅仅调用了一个remote　api，因此找到这个api所对应的处理函数就可以找到Export所有的处理逻辑。

```
            "/containers/{name:.*}/export":    s.getContainersExp

func (s *Server) getContainersExport(version version.Version, w h
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

为什么Docker不使用Java开发，放到虚拟机中运行由虚拟机来管理呢？因为Docker需要调用Linux的内核态函数，如果使用Java来书写代码，复杂度就会以指数级上升，即便抛开复杂度不谈，效率也会大打折扣。所以综合考虑了代码复杂度和效率之后，Docker团队采用Golang来编写。

但因为Golang自身也在不断地发展和完善，Golang的异常处理并不像Java那么成熟，所以Docker在服务端中需要强制性的检测参数合法性。

说完参数检测的问题后，我们深入到s.daemon.ContainerExport函数中一探究竟。

```
func (daemon *Daemon) ContainerExport(name string, out io.Writer)
    container, err := daemon.Get(name)
    if err != nil {
        return err
    }
    data, err := container.Export()
    if err != nil {
        return fmt.Errorf("%s: %s", name, err)
    }
    defer data.Close()

    // Stream the entire contents of the container (basically a v
    if _, err := io.Copy(out, data); err != nil {
        return fmt.Errorf("%s: %s", name, err)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            err := archive.Close()
            container.Unmount()
            return err
        }),
        nil
}
```

在Export函数中，Docker先通过Mount函数将此容器挂载上去。为什么要执行Mount函数？这是为了兼容处理已经关闭的容器。因为从用户的角度分析，只要这个容器没有被删除，那么无论是正在运行的容器，还是已经处于关闭状态的容器，都应该可以被导出。

此处的Mount和Linux系统中的Mount挂载不是同一个意思，在Linux系统中，通过Mount命令，可以让操作系统识别出一个块设备，并可以读写这个块设备。但在上面的代码中，并不是通过Mount来挂载块设备的意思。

这里的Mount是给容器指定一个根路径。但具体做法并不是直接挂载一个目录，而是会逐层将mnt目录和diff目录挂载到容器中。当根目录、mnt目录和diff目录都挂载完毕之后，这个容器才算是真正地完成了数据初始化。

上面代码中，err:=container.Mount()仅有一个error返回值，说明container.Mount函数会将主机多个目录挂载到容器根路径。如果挂载失败（其实也就是当前容器的目录数据Docker维护的容器目录数据不同时），就报错退出。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func (cli *DockerCli) CmdImages(args ...string) error {
    cmd := cli.Subcmd("images", "[REPOSITORY]", "List images", tr
    quiet := cmd.Bool([]string{"q", "-quiet"}, false, "Only show
    all := cmd.Bool([]string{"a", "-all"}, false, "Show all image
images)")
    noTrunc := cmd.Bool([]string{"#notrunc", "-no-trunc"}, false,
    showDigests := cmd.Bool([]string{"-digests"}, false, "Show di

    flFilter := opts.NewListOpts(nil)
    cmd.Var(&flFilter, []string{"f", "-filter"}, "Filter output b
    cmd.Require(flag.Max, 1)
    cmd.ParseFlags(args, true)

    // Consolidate all filter flags, and sanity check them early.
    // They'll get process in the daemon/server.
    imageFilterArgs := filters.Args{}
    for _, f := range flFilter.GetAll() {
        var err error
        imageFilterArgs, err = filters.ParseFlag(f, imageFilterAr
        if err != nil {
            return err
        }
    }

    matchName := cmd.Arg(0)
    v := url.Values{}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if !*quiet {
        if *showDigests {
            fmt.Fprintln(w, "REPOSITORY\tTAG\tDIGEST\tIMAGE ID\tC
        } else {
            fmt.Fprintln(w, "REPOSITORY\tTAG\tIMAGE ID\tCREATED\t
        }
    }

    for _, image := range images {
        ID := image.ID
        if !*noTrunc {
            ID = stringid.TruncateID(ID)
        }

        repoTags := image.RepoTags
        repoDigests := image.RepoDigests
        if  len(repoTags)  ==  1  &&  repoTags[0]  ==  "<none>:<n
repoDigests[0] == "<none>@<none>" {
            // dangling image - clear out either repoTags or repo
            repoDigests = []string{}
        }

        // combine the tags and digests lists
        tagsAndDigests := append(repoTags, repoDigests...)
        for _, repoAndRef := range tagsAndDigests {
            repo, ref := parsers.ParseRepositoryTag(repoAndRef)
            // default tag and digest to none - if there's a valu
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

Images客户端命令和之前分析的客户端命令从逻辑上来看，大致是相似的：首先是处理命令行传来的参数，然后调用服务端提供的/images/json api，最后将服务端返回的数据按照一定的格式输出出来。

Images使用cmd包来处理命令行的参数，没有特别的难点和需要讲解的地方，读者若有不懂的地方，可以参考之前的命令解析。当Images处理完命令行参数之后，就会通过cli.call调用服务端提供的API。

而cli.call函数在前面也介绍过。这个函数会发起一个同步短链接，当服务端返回数据之后，链接就会断开。Images客户端调用的API是/images/json。毫无疑问，下面的分析重点就是这个API了。

直接进入此API对应的处理函数getImagesJSON中：

```
func (s *Server) getImagesJSON(version version.Version, w http.Re
map[string]string) error {
    if err := parseForm(r); err != nil {
        return err
    }

    imagesConfig := graph.ImagesConfig{
        Filters: r.Form.Get("filters"),
        // FIXME this parameter could just be a match filter
        Filter: r.Form.Get("filter"),
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
imageFilters, err := filters.FromParam(config.Filters)
if err != nil {
    return nil, err
}
for name := range imageFilters {
    if _, ok := acceptedImageFilterTags[name]; !ok {
        return nil, fmt.Errorf("Invalid filter '%s'", name)
    }
}

if i, ok := imageFilters["dangling"]; ok {
    for _, value := range i {
        if strings.ToLower(value) == "true" {
            filtTagged = false
        }
    }
}
_, filtLabel = imageFilters["label"]

if config.All && filtTagged {
    allImages, err = s.graph.Map()
} else {
    allImages, err = s.graph.Heads()
}
if err != nil {
    return nil, err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
                continue
            }
            if filtTagged {
                newImage := new(types.Image)
                newImage.ParentId = image.Parent
                newImage.ID = image.ID
                newImage.Created = int(image.Created.Unix())
                newImage.Size = int(image.Size)
                newImage.VirtualSize = int(image.GetParentsSi
                newImage.Labels = image.ContainerConfig.Label
                if utils.DigestReference(ref) {
                    newImage.RepoTags = []string{}
                    newImage.RepoDigests = []string{imgRef}
                } else {
                    newImage.RepoTags = []string{imgRef}
                    newImage.RepoDigests = []string{}
                }
                lookup[id] = newImage
            }
        }
    }
}
s.Unlock()
images := []*types.Image{}
for _, value := range lookup {
    images = append(images, value)
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

稍后会在分析过程中解答。

下面就从这三个步骤开始服务端代码分析，首先是解析筛选参数：

```
var (
    allImages  map[string]*image.Image
    err        error
    filtTagged = true
    filtLabel  = false
)

imageFilters, err := filters.FromParam(config.Filters)
if err != nil {
    return nil, err
}
for name := range imageFilters {
    if _, ok := acceptedImageFilterTags[name]; !ok {
        return nil, fmt.Errorf("Invalid filter '%s'", name)
    }
}

if i, ok := imageFilters["dangling"]; ok {
    for _, value := range i {
        if strings.ToLower(value) == "true" {
            filtTagged = false
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
    return args, nil
}
```

而这里Docker为什么要调用这个函数来解析filter呢？如果读者对此有疑问，则说明对images命令的使用有些陌生了。在Docker基本操作命令中讲到Images命令时曾提到，此命令的filter参数是筛选参数，采用的是key=value的方式。目前有效的key只有dangling和label，而key＝value这种方式，如果采用json进行解析效率就非常高。

而后返回的是一个Map集合，所以对它进行循环处理会很方便。

```
    for name := range imageFilters {
        if _, ok := acceptedImageFilterTags[name]; !ok {
            return nil, fmt.Errorf("Invalid filter '%s'", name)
        }
    }
```

上面的代码主要判断用户输入的筛选参数是否都是有效参数。其判断逻辑是：如果name在参数列表中，其就是有效参数；反之，name不在参数列表中，那么就是无效的。acceptedImageFilterTags就是存放参数列表的Map集合：

```
var acceptedImageFilterTags = map[string]struct{}{
    "dangling": {},
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }
    if err != nil {
        return nil, err
    }
```

可以明显看到这里的allImages变量值和用户输入的筛选条件有关，下面分别看一下s.graph.Map()和s.graph.Heads()有什么区别？先看s.graph.Map函数：

```
func (graph *Graph) Map() (map[string]*image.Image, error) {
    images := make(map[string]*image.Image)
    err := graph.walkAll(func(image *image.Image) {
        images[image.ID] = image
    })
    if err != nil {
        return nil, err
    }
    return images, nil
}
```

在Map函数中，首先创建了一个map集合，然后调用了graph.walkAll：

```
func (graph *Graph) walkAll(handler func(*image.Image)) error {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func (graph *Graph) Heads() (map[string]*image.Image, error) {
    heads := make(map[string]*image.Image)
    byParent, err := graph.ByParent()
    if err != nil {
        return nil, err
    }
    err = graph.walkAll(func(image *image.Image) {
        // If it's not in the byParent lookup table, then
        // it's not a parent -> so it's a head!
        if _, exists := byParent[image.ID]; !exists {
            heads[image.ID] = image
        }
    })
    return heads, err
}
```

从代码来看，Heads函数也是通过walkAll函数来迭代处理传入的每一个image数据的。但和Map函数不同的是，Heads处理的数据是来自于graph.ByParent。下面是graph.ByParent函数的代码：

```
func (graph *Graph) ByParent() (map[string][]*image.Image, error)
    byParent := make(map[string][]*image.Image)
    err := graph.walkAll(func(img *image.Image) {
        parent, err := graph.Get(img.Parent)
        if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        }
    })
```

在Heads函数的walkAll函数中，对传入的image数据依次判断是否存在于byParent集合中。我们知道byParent集合保存的是所有具有父子关系的镜像数据，即在byParent集合中不存在的数据，就是单独存在且没有任何父子关系的孤立的image数据。

看到这里就明白了：Heads函数返回的是孤立镜像数据，而Map函数返回的是所有镜像数据。

Docker这么做有何用意呢？回到这个if判断中来：

```
if config.All && filtTagged {
    allImages, err = s.graph.Map()
} else {
    allImages, err = s.graph.Heads()
}
```

config.All表示的是显示所有数据，filtTagged表示是否显示悬空数据。当dangling为true时，filtTagged会被置为fasle。也就是说，当filtTagged等于false时，只显示孤立处于悬空状态的数据。当filtTagged等于true时，将显示所有的数据。这样就可以明白上面if判断语句了。

默认情况下，Docker会显示所有镜像数据；而当用户使用了dangling参数时，只会显示处于悬空状态的孤立镜像数据。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

s.Repositories是一个Map集合，在这个集合中，保存的是Docker Daemon在启动时初始化的镜像数据。Map中的key保存的是镜像名称，例如，hello-world这样的名称；而value保存的则是此名称下面所有的标签及其ID值。下面是s.Repositories中一条记录所保存的数据：

```
stirng=:[ docker-dev ]
repository=:[
                map[
                       latest:8a049c2b03aa89b052a5221cc121df5cf3c
                       1.7.1:a7d1af33a73b84ad41d296939ff4396ff18b
                      ]
                  ]
```

Docker循环处理s.Repositories时，先执行了下面的判断：

```
        if config.Filter != "" {
            if match, _ := path.Match(config.Filter, repoName);
                continue
            }
        }
```

config.Filter变量保存的是什么呢？这个值又是从何而来呢？其实这个值，最初是从客户端代码中处理的。在客户端代码中有如下一句：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

在这个循环中，Docker需要获取到当前标签下的镜像消息，因此使用了下面的代码来获取数据：

```
            imgRef := utils.ImageReference(repoName, ref)
            image, err := s.graph.Get(id)
            if err != nil {
                log.Printf("Warning: couldn't load %s from %s: %s
                continue
            }
```

imgRef保存的是镜像名称＋当前标签。例如，docker-dev:1.7.1、hello-world:latest这样的变量数据。

然后通过s.graph.Get(id)获取到指定ID的镜像数据。获取这些数据是为了填充最终需要返回的lookup集合。下面看看Docker是如何填充lookup集合的：

```
if lImage, exists := lookup[id]; exists {
    if filtTagged {
        if utils.DigestReference(ref) {
            lImage.RepoDigests = append(lImage.RepoDigests, imgRe
        } else { // Tag Ref.
            lImage.RepoTags = append(lImage.RepoTags, imgRef)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

首先判断此ID是否已经被填充到了lookup中。存在一个ID对应多个标签的情况吗？存在！例如可以通过Docker　tag命令将一个镜像重新指定标签。如果此ID已经在lookup集合中存在了，那么就判断filtTagged。如果filtTagged等于true，则表示需要显示所有的数据，此时就可以将此标签放入结果集中。

如果此ID不在lookup集合之中，那么表示此ID属于首次处理。首先在allImages中删除此ID值（为什么需要删除后面会提到），然后再判断用户是否指定了标签筛选。如果指定了，就判断此ID的标签是否满足用户筛选条件。

最后，提取此镜像的相关信息封装成Image结构体，放入lookup集合中，这样就完成了一次ID镜像的判断。如此循环往复，直至所有的ID都判断完毕。

但这还没有结束。因为还没有回答为什么需要删除allImages中的数据。我们知道allImages中保存的是本次所有需要处理的镜像数据。但在for循环中，处理数据却是根据镜像名称来查找的ID。因此那些没有镜像名称的镜像就会被遗漏掉。这些镜像最后如何处理呢？最简单的方式是在代码的最后，统一给这些镜像添加<none:none>这样的默认值。

而如何保证只给这些没有名称的镜像添加默认值呢？Docker就想到了每处理过一个镜像，就把此镜像在allImages中删除，最后allImages中剩下的数据就是没有名称的镜像，所以才出现了需要执行delete(allImages, id)这样的逻辑。

最后代码，Docker将所有排好序的镜像数据返回给客户端，然后客户端再定义输出样式，循环取出服务端返回的镜像数据，逐个输出就完

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

①创建一个空白的文件系统；

②将指定的内容导入到新创建的文件系统之中；

③将导入内容之后的文件系统命名为指定的名称；

④将此文件系统在Docker环境中完成注册。

我们看一下Import命令是如何实现上述四步的，首先看看客户端代码：

```
func (cli *DockerCli) CmdImport(args ...string) error {
    cmd := cli.Subcmd("import", "URL|- [REPOSITORY[:TAG]]", "Crea
import  the  contents  of  the\ntarball  (.tar,  .tar.gz,  .tgz,
then\noptionally tag it.", true)
    flChanges := opts.NewListOpts(nil)
    cmd.Var(&flChanges, []string{"c", "-change"}, "Apply Dockerfi
image")
    cmd.Require(flag.Min, 1)

    cmd.ParseFlags(args, true)

    var (
        v          = url.Values{}
        src        = cmd.Arg(0)
        repository = cmd.Arg(1)
    )
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        rawTerminal: true,
        in:          in,
        out:         cli.out,
    }

    return cli.stream("POST", "/images/create?"+v.Encode(), sopts
}
```

Import命令的客户端和其他命令不同，这里捎带做了一些逻辑判断，这个逻辑判断非常简单：

```
    if repository != "" {
        //Check if the given image name can be resolved
        repo, _ := parsers.ParseRepositoryTag(repository)
        if err := registry.ValidateRepositoryName(repo); err != n
            return err
        }
    }
```

仅仅判断给定的镜像名称是否可用。但笔者认为为了保持逻辑风格一致，可以将这个判断放到服务端代码中进行判断。在客户端的最后仍然调用了服务端提供的API。

在服务端的处理函数（postImagesCreate）中，其主要有两个操作：执行Pull操作或者执行import操作。下面看看postImagesCreate是如何判

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        output = ioutils.NewWriteFlusher(w)
    )

    w.Header().Set("Content-Type", "application/json")

    if image != "" { //pull
        if tag == "" {
            image, tag = parsers.ParseRepositoryTag(image)
        }
        metaHeaders := map[string][]string{}
        for k, v := range r.Header {
            if strings.HasPrefix(k, "X-Meta-") {
                metaHeaders[k] = v
            }
        }

        imagePullConfig := &graph.ImagePullConfig{
            MetaHeaders: metaHeaders,
            AuthConfig:  authConfig,
            OutStream:   output,
        }

        err = s.daemon.Repositories().Pull(image, tag, imagePullC
    } else { //import
        if tag == "" {
            repo, tag = parsers.ParseRepositoryTag(repo)
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    }

    return nil

}
```

可以看到在postImagesCreate函数中，首先解析客户端传来的参数。然后通过fromImage是否为空来决定后续是执行pull操作还是执行import操作。此时假设fromImage不为空，来看看Docker如何执行pull操作的：

```
    if image != "" { //pull
        if tag == "" {
            image, tag = parsers.ParseRepositoryTag(image)
        }
        metaHeaders := map[string][]string{}
        for k, v := range r.Header {
            if strings.HasPrefix(k, "X-Meta-") {
                metaHeaders[k] = v
            }
        }

        imagePullConfig := &graph.ImagePullConfig{
            MetaHeaders: metaHeaders,
            AuthConfig:  authConfig,
            OutStream:   output,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

Docker为什么需要分别获取镜像名称和镜像标签名称呢？这是因为后面需要调用Pull函数来执行下载操作。Pull函数声明如下：

```
func (s *TagStore) Pull(image string, tag string, imagePullConfig
```

为了满足Pull函数的声明方式，需要分别获取镜像名称和镜像标签。而imagePullConfig保存的是Docker在执行Pull操作时所需的Http header信息。Pull命令在前面已经分析过了，所以这里就不进行代码分析了。

但需要注意的是，Import客户端命令代码中并没有对fromImage做特殊处理。言外之意就是，从客户端发起的Import命令是不会运行到Pull逻辑中的。究其原因，是Docker考虑到客户端命令中已经存在Pull命令了，因而就没必要在Import命令中支持Pull操作了。因此在Import客户端代码中屏蔽了此部分代码处理。

下面在分析Import命令时，重点就放在了另外一个逻辑分支上面：从指定数据源导入镜像。其逻辑如下：

```
if tag == "" {
    repo, tag = parsers.ParseRepositoryTag(repo)

src := r.Form.Get("fromSrc")
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
if src == "-" {
    archive = imageImportConfig.InConfig
} else {
    u, err := url.Parse(src)
    if err != nil {
        return err
    }
    if u.Scheme == "" {
        u.Scheme = "http"
        u.Host = src
        u.Path = ""
    }
    imageImportConfig.OutStream.Write(sf.FormatStatus("", "Do
    resp, err = httputils.Download(u.String())
    if err != nil {
        return err
    }
    progressReader := progressreader.New(progressreader.Confi
        In:        resp.Body,
        Out:       imageImportConfig.OutStream,
        Formatter: sf,
        Size:      int(resp.ContentLength),
        NewLines:  true,
        ID:        "",
        Action:    "Importing",
    })
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if containerID != "" {
        img.Parent = containerImage
        img.Container = containerID
        img.ContainerConfig = *containerConfig
    }

    if err := graph.Register(img, layerData); err != nil {
        return nil, err
    }
    return img, nil
}
```

在Create函数中，最终调用的仍然是graph.Register函数。而为了调用graph.Register函数又构造了一个img结构体，此结构体封装了镜像相关信息。这里需要特别注意的是containerID这个变量，如果此变量为空，那么将不会封装img结构体中关于父镜像的信息。在Import命令中，恰恰是containerID为空的情况。因此，通过Import命令导入的镜像是没有父镜像的。

在Create函数的最后，调用的是graph.Register函数。这个函数在Build、Commit命令中都有涉及，此刻又再次涉及。graph.Register函数逻辑较多，与Import命令有关的是下面的代码：

```
    if err := image.StoreImage(img, layerData, tmp); err != nil {
        return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}
```

通过这段逻辑，Docker将layerData解压到了指定的目录中。当layerData被完全解压之后，其内部保存的镜像数据也就相当于保存到了Docker文件系统中。这么说是因为压缩包里面包括了一个镜像的所有数据，读者可以参阅第4章中有关压缩包内容的相关说明。

而StoreImage仅仅是将镜像数据保存到了文件系统之中，Docker本身还没有“接纳”此镜像。因为在Docker Daemon中还没有意识到此镜像的存在，所以在调用完StoreImage之后，又调用了graph.idIndex.Add(img.ID)函数。这个函数将此镜像ID存入到Docker Daemon之中。这样Docker才意识到了新添加了一个镜像，而镜像ID则是刚才所传入的img.ID。

此时graph.Register函数运行完毕，Create函数也运行完毕。在最外围的Import命令所调用的img, err:=s.graph.Create(archive,"","","Imported from"+src,"", nil, imageImportConfig. ContainerConfig)也执行完毕。

后面Import需要做的就是注册镜像名称和标签。当然这些不是必选项，因为即便没有这些数据，Docker也能查询出来（原因参阅Images命令关于<none:none>的处理）。此部分处理代码如下：

```
    if repo != "" {
        if err := s.Tag(repo, tag, img.ID, true); err != nil {
            return err
        }
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

首先仍然从客户端代码开始：

```
func (cli *DockerCli) CmdLoad(args ...string) error {
    cmd := cli.Subcmd("load", "", "Load an image from a tar archi
    infile := cmd.String([]string{"i", "-input"}, "", "Read from
    cmd.Require(flag.Exact, 0)

    cmd.ParseFlags(args, true)

    var (
        input io.Reader = cli.in
        err   error
    )
    if *infile != "" {
        input, err = os.Open(*infile)
        if err != nil {
            return err
        }
    }
    sopts := &streamOpts{
        rawTerminal: true,
        in:          input,
        out:         cli.out,
    }
    if err := cli.stream("POST", "/images/load", sopts); err != n
        return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    var (
        repoDir = filepath.Join(tmpImageDir, "repo")
    )

    if err := os.Mkdir(repoDir, os.ModeDir); err != nil {
        return err
    }
    images, err := s.graph.Map()
    if err != nil {
        return err
    }
    excludes := make([]string, len(images))
    i := 0
    for k := range images {
        excludes[i] = k
        i++
    }
    if err := chrootarchive.Untar(inTar, repoDir, &archive.TarOpt
err != nil {
        return err
    }
    dirs, err := ioutil.ReadDir(repoDir)
    if err != nil {
        return err
    }
    for _, d := range dirs {
        if d.IsDir() {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

在Load函数中还能隐约发现Import的一些痕迹，而之前对于Import的代码分析也有利于理解Load的一些行为逻辑。从Load开篇的一行代码：tmpImageDir, err:=ioutil.TempDir ("","docker-import-")，大多数读者就能猜测出后面应该是什么逻辑。对，先将压缩包中的内容解压缩到这里的临时目录，然后再从这个临时目录中逐层判断当前Docker环境是否已经存在此文件层，最后将不存在的文件层从临时目录中转移到指定目录。这个逻辑和Import的逻辑大致相同，只是处理过程会更为简单。

```
var (
    repoDir = filepath.Join(tmpImageDir, "repo")
)

if err := os.Mkdir(repoDir, os.ModeDir); err != nil {
    return err
}
images, err := s.graph.Map()
if err != nil {
    return err
}
excludes := make([]string, len(images))
i := 0
for k := range images {
    excludes[i] = k
    i++
}
if err := chrootarchive.Untar(inTar, repoDir, &archive.TarOpt
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        }
        img, err := image.NewImgJSON(imageJson)
        if err != nil {
            logrus.Debugf("Error unmarshalling json", err)
            return err
        }
        if err := image.ValidateID(img.ID); err != nil {
            logrus.Debugf("Error validating ID: %s", err)
            return err
        }

        if c, err := s.poolAdd("pull", "layer:"+img.ID); err != n
            if c != nil {
                logrus.Debugf("Image (id: %s) load is already run
                <-c
                return nil
            }

            return err
        }

        defer s.poolRemove("pull", "layer:"+img.ID)

        if img.Parent != "" {
            if !s.graph.Exists(img.Parent) {
                if err := s.recursiveLoad(img.Parent, tmpImageDir
                    return err
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
imageJson, err := ioutil.ReadFile(filepath.Join(tmpImageD
if err != nil {
    logrus.Debugf("Error reading json", err)
    return err
}

layer, err := os.Open(filepath.Join(tmpImageDir, "repo",
if err != nil {
    logrus.Debugf("Error reading embedded tar", err)
    return err
}
```

我们看到后面Docker获取了每个文件层的json文件和layer.tar。json文件保存的是其所在文件层的元数据，而layer.tar保存的则是文件层中具体文件数据。

而后发现了一个在Import中没有遇见的判断：

```
if c, err := s.poolAdd("pull", "layer:"+img.ID); err != n
    if c != nil {
        logrus.Debugf("Image (id: %s) load is already run
        <-c
        return nil
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

最后调用s.graph.Register(img, layer)来执行导入工作。s.graph.Register函数，在之前的命令中已经分析过了，这里不再赘述。

```
    reposJSONFile, err := os.Open(filepath.Join(tmpImageDir, "rep
    if err != nil {
        if !os.IsNotExist(err) {
            return err
        }
        return nil
    }
    defer reposJSONFile.Close()

    repositories := map[string]Repository{}
    if err := json.NewDecoder(reposJSONFile).Decode(&repositories
        return err
    }

    for imageName, tagMap := range repositories {
        for tag, address := range tagMap {
            if err := s.SetLoad(imageName, tag, address, true, ou
                return err
            }
        }
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return err
    }
    var repo Repository
    repoName = registry.NormalizeLocalName(repoName)
    if r, exists := store.Repositories[repoName]; exists {
        repo = r
        if old, exists := store.Repositories[repoName][tag]; exis

            if !force {
                return fmt.Errorf("Conflict: Tag %s is already se
it, please use -f option", tag, old)
            }

            if old != img.ID && out != nil {

                fmt.Fprintf(out, "The image %s:%s already exists,
empty string\n", repoName, tag, old[:12])

            }
        }
    } else {
        repo = make(map[string]string)
        store.Repositories[repoName] = repo
    }
    repo[tag] = img.ID
    return store.save()
}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    repo[tag] = img.ID
```

Docker首先在当前已经注册的镜像信息中查询是否有指定镜像。请读者注意，此时已经不是通过ID来匹配了，而是通过类似于hello-world这样的名称来匹配。如果当前Docker环境中没有hello-world这样的镜像信息，那么很简单通过repo[tag]=img.ID就可以完成注册。如果存在，那么Docker就开始判断指定的标签是不是存在，如果标签不存在，则会被认为这是一个新镜像。最糟糕的情况就是指定的镜像名称和指定的标签都存在，那么Docker就会认为此镜像已经存在了，需要用户来决定是强行覆盖还是指定为另外的镜像。

当SetLoad函数执行完毕之后，一个全新的镜像就导入到Docker环境之中了，而后用户就可以通过Docker images查询到刚才所导入的镜像。

通过对本节中关于Load命令的解析，读者也可以感受到其实Load和Import区别并不大，都是对在一个临时目录压缩包进行解压，然后再判断目录中的数据是否已经存在，如果不存在再导入到Docker目录中。而唯一不同的是，Load和Import判断的位置不同，一个是判断镜像数据，另外一个是判断容器数据。除此之外，没有大的区别。

5. Pause命令

Pause命令用来暂停指定容器的运行，与Inspect和Image命令相比，Pause命令需要调用Docker内核才能完成功能，所以Pause命令属于Docker比较底层的命令，我们在分析时会更加详细。

虽说Pause命令属于底层命令，但其客户端的代码并不复杂：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func (s *Server) postContainersPause(version version.Version, w h
vars map[string]string) error {
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    }
    if err := parseForm(r); err != nil {
        return err
    }

    if err := s.daemon.ContainerPause(vars["name"]); err != nil {
        return err
    }

    w.WriteHeader(http.StatusNoContent)

    return nil
}
```

很出乎意料，服务端代码中并没有想象的那么复杂，只调用了一个s.daemon.ContainerPause (vars[“name”])就完成了容器暂停工作。接下来，我们看看s.daemon.ContainerPause函数的逻辑：

```
func (daemon *Daemon) ContainerPause(name string) error {
    container, err := daemon.Get(name)
    if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    if !container.Running {
        return fmt.Errorf("Container %s is not running", containe
    }

    if err := container.daemon.execDriver.Pause(container.commanc
        return err
    }
    container.Paused = true
    return nil
}
```

在container.Pause函数中，我们看到Docker是如何判断指定容器是否满足执行Pause的条件：首先容器不能处于Pause状态，其次容器一定要处于Running状态。只有满足这两个条件，Docker才会允许对此容器执行Pause操作。如果读者需要对Pause命令进行定制化操作，那么就可以在此处添加其他判断。

当满足执行Pause的条件之后，Docker调用了container.daemon.execDriver.Pause函数来真正执行Pause操作。对于execDriver，我们之前接触过。execDriver是Docker用来管理容器的一套标准驱动接口，目前使用最多的实现驱动是Docker官方开发的native实现库。因此，接下来看一下在native中是如何实现Pause操作的。

```
func (d *driver) Pause(c *execdriver.Command) error {
    d.Lock()
    active := d.activeContainers[c.ID]
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
func (c *linuxContainer) Pause() error {
    c.m.Lock()
    defer c.m.Unlock()
    return c.cgroupManager.Freeze(configs.Frozen)
}
```

可以看到上面的Pause函数调用了c.m.Lock函数，但在调用active.Pause函数之前已经调用过d.lock，按道理这个容器对象已经上锁了，那么这里再调用c.m.Lock应该会报错的，Docker为什么还要这么做呢？

这是因为在active.Pause之前调用的d.Lock是对driver对象的上锁。driver对象是Docker架构中负责管理所有容器的一个对象，对这个对象上锁可以保证此时driver对象只执行客户端所指定的Pause操作。而这里的c.m.Lock其实是对即将要操作的容器对象上锁，其目的是为了保证只修改客户端所指定的容器。

而为什么不会报错，则是因为这两个锁的使用范围不同。看似这个问题好像没有关联关系，但读者如果需要针对Docker内核进行开发，那么就需要掌握好这些锁的使用场合和时机了。在锁定指定容器之后，Docker调用Freeze函数来“冰冻”容器。这个Freeze函数才是最终的容器“杀手”，Freeze函数如下：

```
func (m *Manager) Freeze(state configs.FreezerState) error {
    d, err := getCgroupData(m.Cgroups, 0)
    if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return err
    }

    dir, err := d.path("freezer")
    if err != nil {
        return err
    }
```

Docker通过调用这些代码可以获取到cgroup目录路径。cgroup还有印象吗？在第1章我们曾经提到过，没有cgroup就不会产生LXC，cgroup是LXC的基础。cgroup是Linux系统中用来限制、记录、隔离进程组物理资源的一种框架。cgroup是由几个子系统组成的：blkio、cpu、cpuacct、cpuset、devices、freezer、memory、net_cls和ns。在执行Pause命令时，底层所调用的则是cgroup中的freezer子系统。读者可以使用mount命令查看当前主机中cgroup配置目录：

```
cgroup on /sys/fs/cgroup type tmpfs (rw,relatime,mode=755)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,relatime,cpuset)
cgroup on /sys/fs/cgroup/cpu type cgroup (rw,relatime,cpu)
cgroup on /sys/fs/cgroup/cpuacct type cgroup (rw,relatime,cpuacct
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,relatime,blkio)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,relatime,memory)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,relatime,devices
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,relatime,freezer
cgroup on /sys/fs/cgroup/net_cls type cgroup (rw,relatime,net_cls
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

FROZEN（冻结）、THAWED（解冻）和FREEZING（冻结中）。在Docker中只用到了前两种状态，没有FREEZING这种状态。而最后的freezer:=subsystems[“freezer”]则是Docker中分别对应cgroup几种子系统的实际处理代码，subsystems的代码如下：

```
    subsystems = map[string]subsystem{
        "devices":    &DevicesGroup{},
        "memory":     &MemoryGroup{},
        "cpu":        &CpuGroup{},
        "cpuset":     &CpusetGroup{},
        "cpuacct":    &CpuacctGroup{},
        "blkio":      &BlkioGroup{},
        "hugetlb":    &HugetlbGroup{},
        "perf_event": &PerfEventGroup{},
        "freezer":    &FreezerGroup{},
    }
```

因此后面所调用的freezer.Set(dir, m.Cgroups)就是FreezerGroup.Set。

```
func (s *FreezerGroup) Set(path string, cgroup *configs.Cgroup) e
    switch cgroup.Freezer {
    case configs.Frozen, configs.Thawed:
        if err := writeFile(path, "freezer.state", string(cgroup.
            return err
        }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

看似这个操作没有必要，但其实这一步是Docker为了确保容器状态变更成功而做的检查。因为freezer.state允许存在三种状态，而最后的状态FREEZING就是处于冻结中的状态。如果长时间出现冻结中的状态，可能意味着此容器状态变更失败。但FREEZING状态是由内核进行操作，一旦长时间出现FREEZING状态，那么Docker将无法处理此种异常情况，只能循环等待。反映到用户那里就是Pause命令长时间等待。所以有必要在这里添加一个超时设定，如果超时则向用户提示状态容器冻结失败。

如果读者需要修改Pause命令，那么建议完善此处的判断。以上就是Pause在Docker中的实现。在本节当中，分析重点是Docker是如何依托native调用cgroup框架完成容器物理资源管理的。在后续的Docker高级进阶篇中还会接触到cgroup，到时会详细分析Docker是如何和cgroup交互的。

6. Port命令

Port命令可以显示指定容器的端口映射信息。前面所介绍的Inspect命令也可以查询到端口映射信息，如下所示：

```
    "PortBindings": {
        "8080/tcp": [
            {
                "HostIp": "",
                "HostPort": "8080"
            }
        ],
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    var c struct {
        NetworkSettings struct {
            Ports nat.PortMap
        }
    }

    if err := json.NewDecoder(stream).Decode(&c); err != nil {
        return err
    }

    if cmd.NArg() == 2 {
        var (
            port  = cmd.Arg(1)
            proto = "tcp"
            parts = strings.SplitN(port, "/", 2)
        )

        if len(parts) == 2 && len(parts[1]) != 0 {
            port = parts[0]
            proto = parts[1]
        }
        natPort := port + "/" + proto
        if  frontends,  exists  :=  c.NetworkSettings.Ports[nat.P
frontends != nil {
            for _, frontend := range frontends {
                fmt.Fprintf(cli.out, "%s:%s\n", frontend.HostIp,
            }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
"NetworkSettings": {
    ....
        "Ports": {
            "8080/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "8080"
                }
            ],
            "8081/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "81"
                }
            ]
        }
    ....
    }

var c struct {
    NetworkSettings struct {
        Ports nat.PortMap
    }
}

type PortMap map[Port][]PortBinding
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

那么参数个数就为2，参数为ubuntu和8081。

这里先假设用户输入的命令是docker port ubuntu，那么cmd.NArg()返回1，Port命令跳过if判断直接运行最后的for判断。这里的for会循环输出此容器中的端口映射信息。输出的格式，就是下面的样式：

```
8081/tcp -> 0.0.0.0:81
8080/tcp -> 0.0.0.0:8080
```

如果cmd.NArg()等于2，那么Port就会运行后面的if判断。这个if判断其实是用来筛选具体端口的，关键代码就是下面的几行：

```
    if frontends, exists := c.NetworkSettings.Ports[nat.Port(port
nil {
            for _, frontend := range frontends {
                fmt.Fprintf(cli.out, "%s:%s\n", frontend.HostIp,
            }
            return nil
    }
```

这里面的port和proto都是从用户输入的第二个参数中截取出来的：

```
        var (
            port  = cmd.Arg(1)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

除容器数据的。

在讲解Rm命令之前，需要明确的是，当删除一个容器数据时都需要删除哪些数据。在Docker中，镜像数据本质是存储在文件系统中的文件数据，而容器则是基于镜像文件数据而运行的动态数据。因此一个正在运行的容器包括两部分数据：已经落地的文件数据和尚未落地的暂存在内存中的数据。

删除一个容器，本质而言是删除这两部分数据。那么这两部分数据应该如何删除呢？相对于落地的文件数据，未落地的内存数据比较好删除。因为Docker当中所有的容器都是基于Linux Container（也就是LXC）技术而虚拟化出来的Linux进程。只要遵循LXC中的标准定义，将此容器所有资源彻底回收，那么此容器运行时态中的所有数据也一并会被销毁掉。

而已经落地的文件数据又该如何删除呢？这些文件删掉即可。对！这些文件既然已经被保存到了主机文件系统当中，那么删掉即可。但问题在于，Docker是使用AUFS文件系统存储数据的，一个容器的数据会同时保存为若干文件层，也就是分散存储在不同位置，因而删除文件的关键在于找到此容器所有的文件数据然后删掉。

因为我们还没有涉及Docker在内核当中是如何使用AUFS文件系统来管理和使用容器的，所以我们先直接给出容器文件数据的存储位置。待在第11章讲解完Docker内核管理之后，读者就会明白容器所有文件数据都分布在哪里了。

简而言之，如果要删除一个容器已经落地的文件数据，那么需要删除以下目录中的容器数据：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    cmd.ParseFlags(args, true)

    val := url.Values{}
    if *v {
        val.Set("v", "1")
    }
    if *link {
        val.Set("link", "1")
    }

    if *force {
        val.Set("force", "1")
    }

    var errNames []string
    for _, name := range cmd.Args() {
        if name == "" {
            return fmt.Errorf("Container name cannot be empty")
        }
        name = strings.Trim(name, "/")

        _, _, err := readBody(cli.call("DELETE", "/containers/"+n
        if err != nil {
            fmt.Fprintf(cli.err, "%s\n", err)
            errNames = append(errNames, name)
        } else {
            fmt.Fprintf(cli.out, "%s\n", name)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    config := &daemon.ContainerRmConfig{
        ForceRemove:  boolValue(r, "force"),
        RemoveVolume: boolValue(r, "v"),
        RemoveLink:   boolValue(r, "link"),
    }

    if err := s.daemon.ContainerRm(name, config); err != nil {
        // Force a 404 for the empty string
        if strings.Contains(strings.ToLower(err.Error()), "prefix
            return fmt.Errorf("no such id: \"\"")
        }
        return err
    }

    w.WriteHeader(http.StatusNoContent)

    return nil
}
```

在服务端代码中，所有的逻辑都是在s.daemon.ContainerRm(name, config)函数中完成的，name保存的是容器名称或者容器ID，而config则保存的是删除选项。

```
func (daemon *Daemon) ContainerRm(name string, config *ContainerR
    container, err := daemon.Get(name)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        return nil
    }

    if err := daemon.rm(container, config.ForceRemove); err != ni
        return fmt.Errorf("Cannot destroy container %s: %v", name
    }

    container.LogEvent("destroy")

    if config.RemoveVolume {
        container.removeMountPoints()
    }
    return nil
}
```

可以看到在ContainerRm函数当中，Docker同时完成了三件事：删除网络映射、删除容器数据和删除挂载卷组。因为删除网络映射和删除挂载卷组比较简单，所以留给读者自行分析。只提示一点，Docker把网络映射存储在SQLite数据库当中，因此在删除网络映射时，需要维护SQLite数据库。

下面我们看一下daemon.rm(container, config.ForceRemove)函数：

```
func (daemon *Daemon) rm(container *Container, forceRemove bool)
    if container.IsRunning() {
        if !forceRemove {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    defer func() {
        if err != nil && forceRemove {
            daemon.idIndex.Delete(container.ID)
            daemon.containers.Delete(container.ID)
            os.RemoveAll(container.root)
        }
    }()

    if _, err := daemon.containerGraph.Purge(container.ID); err
        logrus.Debugf("Unable to remove container from link graph
    }

    if err = daemon.driver.Remove(container.ID); err != nil {
        return fmt.Errorf("Driver %s failed to remove root filesy
container.ID, err)
    }

    initID := fmt.Sprintf("%s-init", container.ID)
    if err := daemon.driver.Remove(initID); err != nil {
        return fmt.Errorf("Driver %s failed to remove init filesy
err)
    }

    if err = os.RemoveAll(container.root); err != nil {
        return fmt.Errorf("Unable to remove filesystem for %v: %v
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
            return fmt.Errorf("Could not kill running container,
        }
    }
```

然后Docker会通过daemon.statsCollector.stopCollection(container)关闭状态采集，在讲解Stats命令时提到了这部分。

而后Docker通过container.SetRemovalInProgress()来标记此容器已经处于删除状态，避免重复操作。Docker又通过container.Stop(3)来再次确认容器是否已经被关闭，而3指的是等待3秒，如果无法正常关闭，则会通过Kill信号强制关闭。

在调用container.Stop函数之后，Docker又连续调用container.SetDead()和container.ToDisk()。这两个函数将容器状态信息写入文件当中，这是为了避免用户通过Restart命令再次启动此容器，造成资源死锁。

上面是Docker为了删除文件数据而做的准备工作，下面Docker开始删除文件数据。

Docker通过调用daemon.containerGraph.Purge(container.ID)来删除此容器与其他容器之间的关系。

```
func (db *Database) Purge(id string) (int, error) {
    db.mux.Lock()
    defer db.mux.Unlock()
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    // Delete entity
    if _, err := tx.Exec("DELETE FROM entity where id = ?;", id);
        tx.Rollback()
        return -1, err
    }

    if err := tx.Commit(); err != nil {
        return -1, err
    }

    return int(changes + refsCount), nil
}
```

然后Docker通过daemon.driver.Remove(container.ID)和daemon.driver.Remove(initID)来删除容器位于aufs目录下的文件数据，也就是清除aufs/mnt、aufs/layers和aufs/diff这三个目录数据。

```
func (a *Driver) Remove(id string) error {
    // Protect the a.active from concurrent access
    a.Lock()
    defer a.Unlock()

    if a.active[id] != 0 {
        logrus.Errorf("Removing active id %s", id)
    }
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    return nil
}
```

为什么要调用两次？这是因为在aufs目录中，Docker还存储了每层的初始化数据，并且以“init”为结尾进行区分，所以需要调用两次才能删除完毕。

而后Docker通过调用os.RemoveAll(container.root)来彻底删除containers目录中的数据。

最后Docker通过daemon.execDriver.Clean(container.ID)来删除execdriver目录中的LXC配置数据。其实在execdriver目录中还有一层目录，这层目录以当前Docker所使用的execDriver名称来命名。目前使用最多的是native方式的execDriver，但不排除使用LXC方式的driver，所以此处目录名没有详细列出。

代码运行到此处时，此容器中所有的文件数据都已经被销毁了。剩下就是销毁此容器的LXC资源。在Docker当中销毁此资源非常简单，调用selinuxFreeLxcContexts(container.ProcessLabel)就可以回收资源。

剩下的两个函数daemon.idIndex.Delete(container.ID)和daemon.containers.Delete(container.ID)就是将此容器从Docker所维护的容器列表中删除。

至此，一个容器就彻底地从当前Docker环境中消失了。目前来说，此Rm命令除了删除网络映射方面还有待完善外，其他地方已经稳定了。在实际生产环境当中，对此命令的扩展需求非常少，而读者如果需要对此进行扩展，那么需要牢记要删除此容器所有的数据，避免出现孤

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    cmd.ParseFlags(args, true)

    var (
        output io.Writer = cli.out
        err    error
    )
    if *outfile != "" {
        output, err = os.Create(*outfile)
        if err != nil {
            return err
        }
    } else if cli.isTerminalOut {
        return errors.New("Cowardly refusing to save to a termina
    }

    sopts := &streamOpts{
        rawTerminal: true,
        out:         output,
    }

    if len(cmd.Args()) == 1 {
        image := cmd.Arg(0)
        if err := cli.stream("GET", "/images/"+image+"/get", sopt
            return err
        }
    } else {
        v := url.Values{}
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
    imageExportConfig := &graph.ImageExportConfig{Outstream: outp
    if name, ok := vars["name"]; ok {
        imageExportConfig.Names = []string{name}
    } else {
        imageExportConfig.Names = r.Form["names"]
    }

    if err := s.daemon.Repositories().ImageExport(imageExportConf
        if !output.Flushed() {
            return err
        }
        sf := streamformatter.NewJSONStreamFormatter()
        output.Write(sf.FormatError(err))
    }
    return nil

}
```

在getImagesGet函数中，Docker生成一个I/O输出流：output。最终在s.daemon.Repositories(). ImageExport函数中，将所生成的压缩包通过output返回给客户端。因此用户在使用Save命令时，需要将最终所获取到的二进制流重定向到一个文件当中。

下面看一下s.daemon.Repositories().ImageExport函数：

```
func (s *TagStore) ImageExport(imageExportConfig *ImageExportConf
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        } else {
            img, err := s.LookupImage(name)
            if err != nil {
                return err
            }

            if img != nil {
                repoName, repoTag := parsers.ParseRepositoryTag(r
                if len(repoTag) > 0 {
                    addKey(repoName, repoTag, img.ID)
                }
                if err := s.exportImage(img.ID, tempdir); err !=
                    return err
                }

            } else {
                if err := s.exportImage(name, tempdir); err != ni
                    return err
                }
            }
        }
        logrus.Debugf("End Serializing %s", name)
    }
    if len(rootRepoMap) > 0 {
        f, err := os.OpenFile(filepath.Join(tempdir, "repositorie
O_TRUNC, 0644)
        if err != nil {
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

文件是从rootRepo:=s.Repositories[name]这句代码开始的。

我们知道s.Repositories保存的是当前Docker环境当中所有的镜像数据，所以如果用户指定name＝hello-world这样格式的镜像名称，那么s.Repositories[name]返回的则是hello-world所有标签的镜像数据。而如果用户指定的是name＝hello-world:latest这样格式的镜像名称，那么s.Repositories[name]将返回空，然后Docker就会通过s.LookupImage(name)来获取此镜像。

但无论用户提供何种格式的镜像名称，Docker总会通过某种方式来准确定位目标镜像。当此镜像被定位到之后，Docker就会通过s.exportImage(img.ID, tempdir)来导出此镜像所有的文件。

```
func (s *TagStore) exportImage(name, tempdir string) error {
    for n := name; n != ""; {
        // temporary directory
        tmpImageDir := filepath.Join(tempdir, n)
        if err := os.Mkdir(tmpImageDir, os.FileMode(0755)); err
            if os.IsExist(err) {
                return nil
            }
            return err
        }

        var version = "1.0"
        var versionBuf = []byte(version)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
        }
        if err := s.ImageTarLayer(n, fsTar); err != nil {
            return err
        }

        // find parent
        img, err := s.LookupImage(n)
        if err != nil {
            return err
        }
        n = img.Parent
    }
    return nil
}
```

在exportImage函数中，Docker从最末端文件层开始将此文件层数据复制到开始所创建的临时目录中。而后通过os.Create(filepath.Join(tmpImageDir,“layer.tar"))创建一个空的压缩包，再经过s.ImageTarLayer(n, fsTar)将此文件层写入layer.tar之中。这一层写完之后，再通过n=img.Parent来获取此文件层的父文件层，依次循环压缩，直到完成所有文件层的压缩。

当所有文件层都压缩完成之后，Docker再调用archive.Tar(tempdir, archive.Uncompressed)将整个临时目录压缩成一个tar包。至此指定的镜像就都导出来了。

在ImageExport函数中还看到了如下一段代码：

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

而这些需求，是Stop和Kill命令无法实现的，因此Docker才增加了Wait命令。下面就来看看Wait命令是如何实现的。

Wait命令的客户端非常简洁：

```
func (cli *DockerCli) CmdWait(args ...string) error {
    cmd := cli.Subcmd("wait", "CONTAINER [CONTAINER...]", "Block
its exit code.", true)
    cmd.Require(flag.Min, 1)

    cmd.ParseFlags(args, true)

    var errNames []string
    for _, name := range cmd.Args() {
        status, err := waitForExit(cli, name)
        if err != nil {
            fmt.Fprintf(cli.err, "%s\n", err)
            errNames = append(errNames, name)
        } else {
            fmt.Fprintf(cli.out, "%d\n", status)
        }
    }
    if len(errNames) > 0 {
        return fmt.Errorf("Error: failed to wait containers: %v",
    }
    return nil
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
type ContainerWaitResponse struct {
    // StatusCode is the status code of the wait job
    StatusCode int `json:"StatusCode"`
}
```

而在服务端，Docker又是如何做的呢？

```
func (s *Server) postContainersWait(version version.Version, w ht
vars map[string]string) error {
    if vars == nil {
        return fmt.Errorf("Missing parameter")
    }

    status, err := s.daemon.ContainerWait(vars["name"], -1*time.S
    if err != nil {
        return err
    }

    return writeJSON(w, http.StatusOK, &types.ContainerWaitRespon
        StatusCode: status,
    })
}

func (daemon *Daemon) ContainerWait(name string, timeout time.Dur
    container, err := daemon.Get(name)
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

```
}

func wait(waitChan <-chan struct{}, timeout time.Duration) error
    if timeout < 0 {
        <-waitChan
        return nil
    }
    select {
    case <-time.After(timeout):
        return fmt.Errorf("Timed out: %v", timeout)
    case <-waitChan:
        return nil
    }
}
```

在WaitStop函数当中，Docker会判断指定容器是否处于Running状态，只有处于Running状态，此函数才有运行的必要。而阻塞则是通过读取waitChan这个channel来完成的。

waitChan是每个容器都会包含的一个channel，用来对外同步容器状态。只有容器被关闭时，此channel才会被销毁，因此Docker就通过监听waitChanl来判断容器是否被关闭。

```
func (daemon *Daemon) newBaseContainer(id string) CommonContainer
    return CommonContainer{
        ID:           id,
```

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

电子书寻找看手相 钉钉或微信pythontesting

第7章　Dockerfile介绍

7.1　Dockerfile有什么用

Docker为我们提供了非常多、非常炫的虚拟化功能，这些功能都是基于镜像技术和容器技术实现的。在前面的章节中，我们详细介绍了镜像如何使用和容器如何使用，但并没有回答镜像是如何来的。

镜像的由来与Dockerfile存在着很深的关联关系，当我们讲解完Dockefile的语法及使用后，镜像由来的问题就会迎刃而解，所以我们暂时先把镜像的由来放到一旁，看一下Dockerfile。

如果需要给Dockerfile下个定义，那么Dockerfile首先是一个文本文件，然后Dockerfile是一个Docker可以读懂的脚本文件，在这个脚本文件中记录着用户“创建”镜像过程当中需要执行的所有命令。

当Docker读取并执行Dockerfile中所定义的命令时，这些命令将会产生一些临时文件层。还记得我们提到过Docker使用的是AUFS文件系统吗？AUFS文件系统会将不同的文件层进行“叠加”从而形成一个完整的文件系统。这里在Dockerfile中执行命令后形成的临时文件层就是AUFS所使用的文件层。当Dockerfile所有命令都成功执行完之后，Docker会记录执行过程中所用到的所有文件层，并且会用一个名称来标记这一组文件层。

这一组文件层，就被称之为镜像。因此我们可以得出结论：镜像指的是一组特定的文件层。Docker中镜像的构建过程，就是Docker执行

Dockerfile中所定义命令而形成这组文件层的过程。

Docker中所有的容器都是基于镜像而创建的，而所有的镜像又都是通过Dockerfile而形成的。通过这样的逻辑，我们就能够明白Docker整套虚拟生态圈的起点就是Dockerfile。

在下面一节中，我们会详细介绍Dockefile的用法，以及里面的各个元素如何定义。在本节，我们先简单了解一下Dockerfile的语法，为下节做准备。

语法是什么呢？语法就是程序可以读懂的语义结构。程序通过解析特定的语义结构，可以明白用户的意图，从而执行特定的动作，产生用户所预期的结果。这些语义结构必须是符合规范、明确无歧义，并且方便用户书写的。

Dockerfile的语法非常简单，所以无论是编写，还是修改Dockerfile，都是一件非常容易的事情。下面是Dockerfile典型的语法示例：

```
# 在Dockerfile语法中，#表示单行注释。
command argument argument ..

#Dockerfile语法，就是命令+参数+参数……
#例如，我们想要在镜像构建过程中输出一个hello world
RUN echo "Hello Docker!"

#RUN是Dockerfile内置命令，表示在镜像中执行后面的命令，详细用法会在后面介绍
```

在Dockerfile中，可以使用的内置命令及其作用如表7-1所示。

表7-1　Dockerfile中的内置命令及其作用

内置命令	作用
FORM	指明基础镜像名称。必填项
MAINTAINER	可用于提供作者、版本及其他备注信息。可选项
RUN	用于执行后面的命令。当RUN执行完毕后，将产生一个新的文件层。可选项
CMD	指定此镜像启动时默认执行命令。可选项
LABEL	用于在镜像中添加元数据。例如版本号、构建日期等。可选项
EXPOSE	用于指定需要暴露的网络端口号。可选项
ENV	用于在镜像中添加环境变量。可选项
ADD	向镜像添加新文件或者新目录。可选项
COPY	从主句向镜像复制文件。可选项
ENTRYPOINT	在镜像中设定默认执行的二进制程序。可选项
VOLUME	向镜像中挂载一个卷组。可选项
USER	在镜像构建过程中，生成或者切换到另外一个用户。可选项
WORKDIR	设定此镜像后续操作的默认工作目录。可选项
ONBUILD	配置构建触发指令集。可选项

下面是Dockerfile典型示例：

```
##################################################################
# Dockerfile to build MongoDB container images
# Based on Ubuntu
##################################################################

# Set the base image to Ubuntu
FROM ubuntu

# File Author / Maintainer
MAINTAINER Example McAuthor
```

```
# Update the repository sources list
RUN apt-get update

################## BEGIN INSTALLATION ######################
# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-or

# Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv

# Add MongoDB to the repository sources list
RUN  echo  'deb  http://downloads-distro.mongodb.org/repo/ubuntu-
/etc/apt/sources.list.d/mongodb.list

# Update the repository sources list once more
RUN apt-get update

# Install MongoDB package (.deb)
RUN apt-get install -y mongodb-10gen

# Create the default data directory
RUN mkdir -p /data/db

##################### INSTALLATION END #####################

# Expose the default port
```

```
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

当我们编写好Dockerfile之后，就可以通过Docker CLI中的build命令来执行Dockerfile了。

```
#build命令，就是用来执行Dockerfile的，下面是其用法
# Example: sudo Docker build -t [name] .

Docker build -t my_ mongodb .
```

在等待Dockerfile中定义的所有命令都执行完毕之后，一个新的镜像my_mongodb就产生了。然后就可以通过Docker run或者Docker-compose工具来创建MongoDB的实例，从而对外提供服务了。

本节，我们介绍了Dockerfile有哪些用途以及Dockerfile的语法结构。在7.2节中，我们将会针对Dockerfile中的内置命令展开阐述。

7.2 如何编写Dockerfile

在7.1节中，我们列出了Dockerfile中的内置命令。在本节，我们将介绍这些内置命令如何使用。

1. FROM命令

```
FROM <image>
FROM <image>:<tag>
FROM <image>@<digest>
```

FROM是Dockerfile内置命令中唯一一个必填项，其有上述三种用法。FROM用来指定后续指令执行的基础镜像，所以在一个有效的Dockerfile中，FROM永远是第一个命令（注释除外）。

FROM指定的基础镜像既可以是本地已经存在的镜像，也可以是远程仓库中的镜像。当Dockerfile执行时，如果本地没有其指定的基础镜像，那么就会从远程仓库中下载此镜像。

在Dockerfile的语法中，并没有规定只能出现一次FROM，因而如果需要在一个Dockerfile中同时构建多个镜像时，也可以出现多次FROM，例如：

```
# Nginx
#
# VERSION               0.0.1

FROM      ubuntu
```

```
MAINTAINER Victor Vieux <victor@Docker.com>

LABEL Description="This image is used to start the foobar executa
Version="1.0"
RUN apt-get update && apt-get install -y inotify-tools nginx apac

# Firefox over VNC
#
# VERSION               0.3

FROM ubuntu
# Install vnc, xvfb in order to create a 'fake' display and firef
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir ~/.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the t
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]

# Multiple images example
#
# VERSION               0.1

FROM ubuntu
```

```
RUN echo foo > bar
# Will output something like ===> 907ad6c2736f

FROM ubuntu
RUN echo moo > oink
# Will output something like ===> 695d7793cbe4

# You'll now have two images, 907ad6c2736f with /bar, and 695d779
# /oink.
```

在上面的Dockerfile中一共出现了三次FROM命令，这就意味着当这个Dockerfile执行完毕之后，会同时生成三个镜像，但只会输出最后一个镜像的ID值，中间两个镜像只会被标记为<none>:<none>。所以是否需要在一个Dockerfile中同时生成多个镜像，需要用户根据情况自行决定。

在FROM用法中，tag和digest属于可选项，如果没有则默认取指定镜像的最新版，也就是latest版本。如果找不到latest版本，那么整个Dockerfile就会报错返回。

2. MAINTAINER命令

```
MAINTAINER authors_name
```

MAINTAINER是用来维护镜像作者的命令。一般情况下是放在FROM命令后面，当然也可以放在其他位置。

这里除了输入作者信息之外，还可以输入其他信息，同样不会报错。但入乡随俗，这里还是建议只输入作者信息，其他备注类的信息可以放到LABELS字段中。

3. RUN命令

```
RUN <command>
RUN ["executable", "param1", "param2"]
```

RUN命令是用来在镜像中执行命令的命令，是一个完整Dockerfile中使用频率最高的命令，其有上面两种用法，最基本的用法就是RUN<command>。

当使用RUN<command>用法时，后面的命令其实是由/bin/sh–C来负责执行的。所以RUN的这种用法就会存在一个限制，那就是在镜像中必须要有/bin/sh。

如果基础镜像中没有/bin/sh时，该怎么办呢？

如果基础镜像没有/bin/sh，那么就需要使用RUN的另外一个用法了。RUN　["executable","param1","param2"]，可以执行基础镜像中任意一个二进制程序。比如我们需要使用bash来执行程序，就可以这样编写RUN指令：

```
RUN ["/bin/bash", "-c", "echo hello"]
```

在使用这种用法时，[]中的数据都会被按照JSON字符串的格式进行解析，因此只能使用双引号“”，不能使用单引号或者其他符号，这点请读者注意。

同时使用这种用法时，还需要注意到环境变量的使用问题：

```
RUN [ "echo", "$HOME" ]
```

基础镜像中即便存在$HOME变量，上述示例仍然会失败。因为此时RUN执行时，不会加载环境变量中的数据。如果需要使用环境变量，那么只能通过下面的方式：

```
RUN [ "sh", "-c", "echo", "$HOME" ]
```

当RUN命令执行完毕之后，就会产生一个新的文件层。这个新产生的文件层会被保存在缓存中，并且将作为下个指令的基础镜像而存在，例如：

```
RUN apt-get dist-upgrade -y
```

这条命令产生的数据将会被后续所有指令复用。如果不需要在缓存中保存这些数据，那么需要使用--no-cache来禁用缓存保存功能，例如：

```
Docker build --no-cache XXXXX
```

在Docker目前版本中，存在一个已知的问题：所有镜像最多只能保存126个文件层。而执行一次RUN就会产生一个文件层，而且新产生的镜像会包括基础镜像的文件层，如图7-1所示。

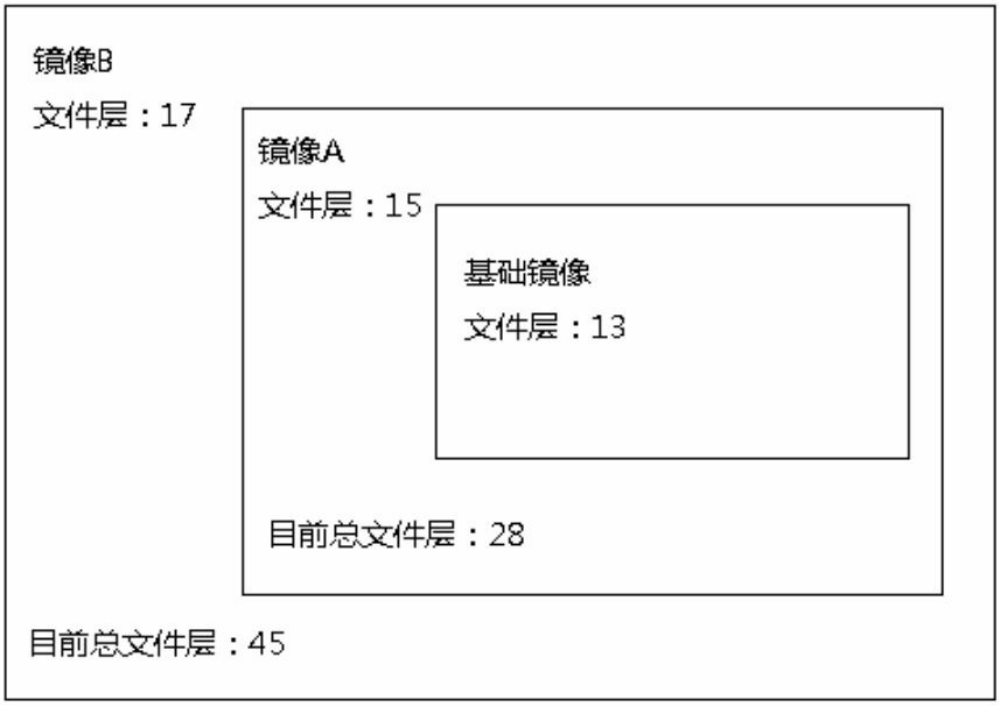


图7-1　文件层

虽然镜像B只产生了17个文件层，但其作为基础镜像的镜像A一共有28个文件层。这就导致了镜像B需要维护45个文件层。而当文件层数超过126时，Docker就无法再进行后续维护了。因此后续所有的命令都将失败。

所以使用RUN指令时，应尽量将多个命令放到一个RUN中去执行，在后续的Dockerfile最佳实践中，我们将谈到这个问题。

4. CMD命令

```
CMD ["executable","param1","param2"]
CMD ["param1","param2"]
CMD command param1 param2
```

CMD是用来设定镜像默认执行命令的命令。第一种用法是推荐的用法，当使用这种用法时，其设定的命令将作为容器启动时的默认执行命令，例如：

```
CMD  ["x11vnc", "-forever", "-usepw", "-create"]
```

当使用第二种用法时，其中的param将作为ENTERPOINT的默认参数使用。而第三种用法是将后面的命令作为shell命令，依靠/bin/sh–C来执行，例如：

```
CMD echo "This is a test." | wc -
```

如果用户需要脱离shell环境来执行命令，那么建议使用第一种用法来设定。

但无论使用哪种方法，都需要注意，在一个Dockerfile中可以同时出现多次CMD指令，可只有最后一次CMD命令会生效。同时在CMD中也只能出现双引号，不能使用单引号。与RUN命令一样，如果需要使用

环境变量，请使用sh–C。

CMD与RUN都是执行命令的命令，但RUN是在镜像构建过程中执行的命令。而CMD命令只是设定好需要执行的命令，只有等容器启动时才会真正执行。

5. LABEL命令

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

LABEL是一个采用键值对的形式来向镜像中添加元数据的命令，例如：

```
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

如果键值对中存在空格，那么就需要使用双引号来回避可能出现的错误，例如：

```
LABEL "com.example.vendor"="ACME Incorporated"
```

执行完LABEL命令之后，同样会产生一个新的文件层。因此为了

避免出现126层的问题，我们建议将多个LABEL放到一起，统一执行，例如：

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

如果新添加的LAEBL在旧的镜像中已经存在，那么新添加的LABEL将会覆盖旧值。

当镜像构建成功之后，可以通过Docker CLI中的inspec命令查询到LABEL值，如下所示：

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can s
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
}
```

6. EXPOSE命令

```
EXPOSE <port> [<port>...]
```

EXPOSE命令是当容器运行时，来通知Docker这个容器中哪些端口是应用程序用来监听的。例如，在一个tomcat的Dockerfile中，一定会出现下面的EXPOSE指令：

```
EXPOSE 8080
CMD ["catalina.sh", "run"]
```

当Tomcat容器运行时，Tomcat应用就会开始监听8080端口了。

但是使用EXPOSE命令不等同于这些端口就可以被外部网络访问。只有在容器启动时，配合-p或者-P，外部网络才可以访问到这些端口。如果没有使用-p或者-P，那么这些端口只能被主机中的其他容器访问，无法被外部网络所访问到。

7. ENV命令

```
ENV <key> <value>
ENV <key>=<value> ...
```

ENV命令用来在基础镜像中设定环境变量，有上面两种用法。同LABELS一样，ENV命令使用的也是key-value的方式存储数据。当使用第一种用法时，第一个字符串将被当作key来处理，后面的所有字符串将被当作value来处理，如下所示：

```
```

```
ENV myName John Doe
```

当采用第二种用法时，等号左边将是key，右边是value。如果value中存在空格时，需要使用“\”来进行转义，如下所示：

```
ENV myDog=Rex\ The\ Dog \
```

同时在使用第一种用法时，每个ENV命令只能维护一条环境变量，而第二种用法可以同时维护多条变量，如下所示：

```
##第一种用法
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
##第二种用法
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

虽然在最终的镜像中，环境变量中的结果是一样的，但使用第二种用法时，这些环境变量会被保存在同一个文件层中。

当容器运行时，可以通过Docker inspect来查看，同时也可以在执行Docker run时通过-e来重新设定环境变量。

8. ADD命令

```
ADD <src>... <dest>
ADD ["<src>",... "<dest>"]
```

ADD命令有两种用法，其实两种用法是相同的，第二种用法可以用来处理文件路径有空格的情况。

ADD命令是将src标记的文件，添加到容器中dest所标记的路径中去。src标记的文件可以是本地文件，也可以本地目录，甚至可以是远程URL链接。

当src标记的是本地文件或者目录时，其相对路径应该是相对于Dockerfile所在目录的路径，而dest则应该指向容器中的目录。如果这个目录不存在，那么当ADD命令执行时，将会在容器中自动创建此目录。

在src标记的路径中，允许使用通配符，例如：

```
ADD hom* /mydir/          # 添加所有以hom开头的文件
ADD hom?.txt /mydir/      # ?可以被任意单个字符所代替
```

而dest的路径则不允许使用通配符，并且其路径必须是绝对路径，或者相对于WORKDIR的相对路径，例如：

```
ADD test aDir/            #假设/是WORKDIR，那么就添加test到/aDir/
```

当ADD命令执行成功之后，所有新添加的文件或者目录的UID和GID默认都为0。

在使用ADD命令时，有以下几条规则需要遵守。

规则1

src指定的路径必须存在于Dockerfile所在目录。例如，下面所给定的路径就是非法路径：

```
ADD ../something /something
```

因为在Dockerfile执行时，Docker Daemon会读取Dockerfile所在目录的所有数据。如果ADD命令使用的文件在此目录中不存在，那么Daemon将找不到指定文件。

规则2

如果src指定的是URL，并且dest所指定的路径没有以“/”结尾，那么URL下载的数据将直接覆盖dest所给定的文件。

规则3

如果src指定的是URL，并且dest所指定的路径是以“/”结尾的。那么URL下载后的数据将直接写入dest所指定的目录中，例如：

```
ADD http://example.com/foobar /
```

ADD命令将会下载foobar目录，并且将此目录放入容器的根目录，结果是在容器中出现/footbar目录。

规则4

如果src指向的是一个目录，那么ADD指令将包括元数据在内的所有数据复制到容器中dest所指定的文件中，但src所指定的目录本身不会被复制进去，只会复制此目录下的文件。

规则5

如果src指向的是一个已知格式的压缩文件，例如，gzip、bzip2或者xz格式的文件。当添加到容器之后，会自动执行解压缩动作。而从URL中下载的压缩文件则不会执行解压缩。

规则6

如果src使用通配符指定了多个文件，那么此时dest必须是一个以“/”结尾的目录。

规则7

如果dest指向的路径没有以“/”结尾，那么这个路径指向的文件将会被src指定的文件覆盖。

规则8

如果dest指向的路径不存在，那么此路径中所涉及的父级目录都将会被创建。

规则9

当src指向的URL没有下载权限时，首先需要使用RUN wget或者RUN curl获取文件。

规则10

当ADD命令所标记的文件发生变化时，从变化的那个ADD命令开始，保存在缓存中的镜像将会失效，同时RUN命令产生的缓存镜像也会失效。

9. COPY命令

```
COPY <src>... <dest>
COPY ["<src>",... "<dest>"]
```

同ADD命令一样，COPY命令的第二种用法是用来处理路径中存在空格的情况。

COPY命令也是向容器中指定路径下添加文件。在添加文件时，同样需要确认此文件的确存在于Dockerfile所在目录中。与ADD命令相同，COPY命令也支持下例格式中的通配符：

```
COPY hom* /mydir/          # 添加所有以hom开头的文件
```

```
COPY hom?.txt /mydir/    # ?可以被任意单个字符所代替
```

在dest中的路径必须是全路径或者是相对于WORKDIR的相对路径。

在使用COPY命令时，需要遵循以下几条规则。

规则1

src指定的路径必须存在于Dockerfile所在目录。例如，下面所给定的路径就是非法路径：

```
ADD ../something /something
```

当Dockerfile执行时，Docker Daemon会读取Dockerfile所在目录的所有数据。所以如果COPY命令指定的文件在此目录中不存在，那么Daemon将找不到指定文件。

规则2

如果src指向的是一个目录，那么COPY命令将包括元数据在内的所有数据复制到容器中dest所指定的文件中。但src所指定的目录本身不会被复制进去，只会复制此目录下的文件。

规则3

如果src使用通配符指定了多个文件，那么此时dest必须是一个

以“/”结尾的目录。

规则4

如果dest指向的路径没有以“/”结尾，那么这个路径指向的文件将会被src指定的文件覆盖。

规则5

如果dest指向的路径不存在，那么此路径中所涉及的父级目录都将会被创建。

规则6

如果使用STDIN输入Dockerfile内容，那么COPY命令将失效，例如：

```
Docker build - < somefile
```

此时COPY命令将无法使用。

10. ENTRYPOINT命令

```
ENTRYPOINT ["executable", "param1", "param2"]
ENTRYPOINT command param1 param2
```

ENTRYPOINT是用来设定容器运行时默认执行程序的命令。第一

种用法是Docker官方推荐的用法。通过第一种用法，读者可以自行设定需要执行的二进制程序和参数。而第二种方法则将所设定的二进制程序限制在/bin/sh–C下执行。

下面我们通过一个实例来演示ENTRYPOINT命令的用法：

```
##首先截取Nginx镜像部分Dockerfile
RUN sed -Ei 's/^(bind-address|log)/#&/' /etc/mysql/my.cnf \
  && echo 'skip-host-cache\nskip-name-resolve' | awk '{ print } S
system("cat") }' /etc/mysql/my.cnf > /tmp/my.cnf \
  && mv /tmp/my.cnf /etc/mysql/my.cnf

VOLUME /var/lib/mysql

COPY Docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 3306
CMD ["mysqld"]
```

在MySQL官方提供的Dockerfile中，ENTRYPOINT命令使用的是第一种用法，设定含义为当mysql容器运行时，自动执行/entrypoint.sh，而参数则是mysqld。（为什么mysqld是参数，读者可参考CMD一节）

而/entrypoint.sh的功能就是启动mysql守护进程，并且创建默认用户。当entrypoint.sh执行成功之后，一个可供用户使用的MySQL数据库

就准备好了。

Docker run mysql param1 param2所提供的两个参数，都将作为参数传入entrypoint.sh中。

下面我们再看一下示例：

```
##启动一个Nginx容器
Docker run -i -t --rm -p 80:80 nginx
```

当这个命令执行成功之后，一个监听80端口同时对外提供Web应用的Nginx容器就准备好了。但我们只是创建了容器，并未初始化Nginx应用，那么nginx应用的初始化工作是谁做的呢？

其实这些初始化工作就是依靠Nginx容器中ENTRYPOINT设定的脚本执行的。如果我们在run命令后面添加了其他参数，这些参数就会传入ECTRYPOINT设定的脚本中，同时CMD中所设定的参数也将会失效。

而ENTRYPOINT所设定的值，可以在run命令中通过-entrypoint来修改，例如：

```
Docker run –entrypoint="/bin/sh"  -d mysql
##将entrypoint.sh替换为/bin/sh
```

与CMD命令一样，ENTRYPOINT命令可以在Dockerfile中出现多

次，但只有最后一次出现的ENTRYPOINT才会生效。

下面我们通过几个演示来加深对ENTRYPOINT命令的理解。

首先我们构建一个ubuntu容器，并且设定了ENTRYPOINT和CMD：

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

当直接创建这个ubuntu容器之后，就可以直接查看到top–b–H执行的结果：

```
$ Docker run -it --rm --name test  top -H
top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.0
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 z
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME
    1 root      20   0   19744   2336   2080 R  0.0  0.1   0:00.0
```

下面通过ps命令来验证结果：

```
$ Docker exec -it test ps aux

USER       PID %CPU %MEM    VSZ   RSS  TTY      STAT START   TIME
root             1  2.6  0.1  19752     2352 ?        Ss+     08
root             7  0.0  0.1  15572     2164 ?        R+      08
```

加黑的进程表示目前运行的进程就是top–b–H.

为什么是-H而不是-C呢？这是因为当run命令后没有添加其他参数时，CMD指定的-C将作为参数附加到top–b之后。如果run命令后面添加了其他参数，此时CMD指定的参数将会失效。

当我们使用ENTRYPOINT命令的第二种用法设定值时，又会怎样呢？

当使用第二种用法时，ENTRYPOINT命令设定的二进制程序将会忽略所有来自于CMD和RUN命令后面所添加的参数，只会运行ENTRYPOINT命令所设定的二进制程序。同时，为了确保容器可以正确处理stop命令发来的SIG信号，Docker建议使用exec来启动二进制程序。具体原因，我们看一下下面的示例：

首先我们构建一个同样执行top命令的ubuntu容器：

```
FROM ubuntu
ENTRYPOINT exec top –b
CMD ["-c"]
```

当我们运行这个容器时，会出现下面的情况：

```
$ Docker run -it --rm --name test top

Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167
CPU:   5% usr   0% sys   0% nic  94% idle   0% io   0% irq   0% s
Load average: 0.08 0.03 0.05 2/98 6
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     R     3164   0%   0% top -b
```

当前容器中只运行着top–b命令，CMD的参数和run后面添加的top都没有发挥作用。同时top–b变成了根进程，PID=1。

如果此时执行Docker stop，则可以正确关闭此容器。但如果我们在设定ENTRYPOINT时忘记使用exec了，那么就会出现下面的情况：

```
##Dockerfile
FROM ubuntu
ENTRYPOINT top –b

##运行容器
$ Docker run -it --name test
Mem: 1704184K used, 352484K free, 0K shrd, 0K buff, 140621524238
CPU:   9% usr   2% sys   0% nic  88% idle   0% io   0% irq   0% s
Load average: 0.01 0.02 0.05 2/101 7
```

```
  PID   PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1      0 root     S     3168   0%   0% /bin/sh -c top -b
    7      1 root     R     3164   0%   0% top -b
```

可以看到top–b不再是根进程了，而是变成了sh的子进程。此时执行Docker stop，因为sh不会处理Linux信号，所以容器不会正确关闭。只有过了所设定的超时时间后，通过SIGKILL信号才能强行关闭。

11. VOLUME命令

```
VOLUME ["/data"]
```

VOLUME可以在容器内部创建一个指定名称的挂载点。VOLUME命令参数可以采用类似VOLUME ["/var/log/"]这样的JSON格式数据（当使用JSON格式时，只能使用双引号）。也可以是使用空格分隔的字符串，例如：VOLUME/var/log/var/db。

如下所示：

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

我们创建了一个新的挂载点，并且在此挂载点中生成了greeting文件。

在使用VOLUME命令时需要注意，如果在Dockerfile中已经声明了某个挂载点，那么以后对此挂载点中文件的操作将不会生效。因此，一般来说，只会在Dockerfile的结尾处声明挂载点。

12. USER命令

```
USER daemon
```

USER命令是用来切换用户身份的。当执行完USER命令后，其后面所有的命令都将以新用户的身份来执行。

13. WORKDIR命令

```
WORKDIR /path/to/workdir
```

WORKDIR是用来切换当前工作目录的指令。WORKDIR命令中所切换的工作目录，可以影响到后续的RUN、CMD、ENTRYPOINT、COPY和ADD指令中的路径。

WORKDIR可以在Dockerfile中出现多次，但最终生效的路径是所有WORKDIR指定路径的叠加，例如：

```
WORKDIR /a
##当前目录为/a
WORKDIR b
##当前工作目录为/a/b
WORKDIR c
##当前工作目录为/a/b/c
RUN pwd
##最终结果为/a/b/c
```

如果需要切换到其他工作目录，那么应该使用全路径进行切换。如果使用相对路径，则默认是在当前目录中切换。

在WORKDIR中只可以使用ENV设定的环境变量值，例如下例：

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
```

因为DIRPATH在环境变量中存在，所以最终结果为：/path/$DIRNAME

14. ONBUILD命令

```
ONBUILD [INSTRUCTION]
```

ONBUILD是用来创建触发命令集的命令。由ONBUILD创建的触发命令集在当前Dockerfile执行过程中不会执行，而当此镜像被其他镜像当作基础镜像使用时，将会被触发执行。

ONBUILD不挑食，所有只要是在Dockerfile中属于合法的内置命令，都可以在此设定。

这是一个非常有意思的功能，例如，当前镜像A包含一个特殊应用，当此镜像被其他镜像当作基础镜像而引入时，镜像A中的ONBUILD指令集就会自动运行，创建初始化用户，或者初始化环境变量，以及自动创建挂载点等操作。

再比如，镜像A是一个Python应用程序的编译环境。在使用镜像A编译自己的代码时，可能需要将源代码复制到一个特定目录，然后执行编译脚本。这些如果都需要用户来做，那么会相当费时费力。而镜像A在构建时，可以把这些工作都封装成ONBUILD指令集，自动运行。

下面我们看一下ONBUILD的运行机制：

（1）当Dockerfile执行时，如果遇到ONBUILD标记的命令，其将会把这些命令作为镜像的元数据保存起来，并且在本次执行时不再执行这些命令。

（2）在Dockefile所有的命令都执行成功后，标记为ONBUILD的命令将会被保存到镜像的manifest文件中，后续可以通过Docker inspect来查看。

（3）当此镜像被用作基础镜像时，Docker首先会取出这些标记为ONBUILD的命令，然后按照其当初被标记的顺序执行。如果有一条执

行失败，则本次Dockefile整体失败返回。如果所有的ONBUILD命令执行成功，则FROM步骤成功，然后再执行后续的指令。

（4）这些命令不会被子镜像继承。

百闻不如一见，我们看一个使用ONBUILD命令的示例：

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

在ONBUILD命令中不允许嵌套，即ONBUILD ONBUILD是不允许的，同时在ONBUILD命令中也不允许执行FROM和MAINTAINER命令。

第8章　Dockerfile最佳实践

Dockerfile优化方案

在第7章中，我们讲解了Dockerfile的作用，以及如何书写Dockefile。编写一个Dockefile脚本看起来很容易，但要真正写好一个Dockerfile却不是一件很容易的事情。如果Dockerfile中结构混乱，或者命令太多，影响的不仅仅是Dockerfile的可读性，甚至会降低镜像的构建效率，拉长整个镜像的构建时间。

下面是Docker社区推荐的Dockerfile最佳实践，这些已经被证明是非常有效的最佳实践，希望可以帮助Docker开发者编写出高效、清晰、可靠的Dockerfile。

首先，我们介绍一下在编写Dockerfile时的通用法则。

1. 容器要尽可能的短小精悍

我们编写Dockerfile最终的目的是为了得到一个镜像，然后通过镜像得到一个可运行的容器。所以在编写Dockerfile之前，就应该确定这个容器将要提供哪些服务。这个容器不要大而全，而是尽可能的精悍。可以随时快速地启动和停止，同时可以用最少的步骤来配置容器服务。

2. 多使用**.Dockerignore**文件

.Dockerignore文件和.gitignore文件性质是一样的，都是当加载文件

时过滤某些文件，在介绍Dockerfile的COPY和ADD命令时介绍过。Docker在执行Dockerfile文件时，会将此文件所在目录中的所有文件都读入缓存中。但目录中这么多的文件，我们不一定都会用到，如何让Docker只加载我们需要的文件呢？

最通用的方式是使用Dockerignore文件。在这个文件中，我们定义出哪些文件不是我们所关心的。当Docker加载文件时，如果发现目录中存在. Dockerignore文件，那么此目录中符合. Dockerignore文件中定义规则的文件将会被过滤掉，不会被加载到缓存中。

但.Dockerignore文件必须放到Dockerfile所在目录下面。如果放到其他子目录，那么Docker会把这个文件当作普通文件来处理。.Dockerignore文件中的路径是以.Dockerignore所在目录为起点的。同时在.Dockerignore文件中，允许通配符的存在，例如：

```
*/temp*
*/*/temp*
temp?
*.md
!LICENSE.md
```

上面几条规则的含义如表8-1所示。

表8-1 Dockerfile通用规则的含义

规则	含义
/temp	二级子目录中，所有以 temp 开头的文件都将被忽略。 例如：/somedir/temporary.txt
//temp*	在三级子目录中，所有以 temp 开头的文件都将被忽略。 例如：/somedir/subdir/temporary.txt
temp?	在当前目录中，所有以 temp 开头的，并且文件名为 5 个字母的文件将被忽略。 例如：temp1、 tempa、tempb 等文件
*.md	在当前目录中，所有以.md 为结尾的文件都将被忽略
!LICENSE.md	忽略除 LICENSE.md 之外的所有文件。只加载 LICENSE.md 文件

在.Dockerignore文件中，后面的规则将覆盖上面的规则。如果规则有冲突，则以最后的规则为准。例如，我们将上面的规则稍加调整：

```
*/temp*
*/*/temp*
temp?
!LICENSE.md
*.md
```

我们将！LICENSE.md和*.md调换一下位置，意思就完全不一样了。在调换前，LICENSE.md是可以被加载到缓冲中去的，但调换位置之后，首先解析的是！LICENSE.md，这样会过滤掉除LICENSE.md之外的所有文件，然后会解析*.md，这样又会过滤掉所有的md文件，因此LICENSE.md也将处于过滤范围，所以，最终结果就是LICENSE.md将不会被加载到缓存中。

在.Dockerignore文件中，同样可以忽略.Dockerignore文件本身和Dockerfile文件，这个非常有必要。因为一旦Dockerfile文件内容发生了变化，那么缓存中相关的临时镜像都将失效，Dockerfile中所有的指令

都将重新执行一遍，这样势必会降低构建效率。

3. 不要安装非必需的软件包

为了达到最终镜像和容器短小精悍的目的，那么不会用到的软件包或者不适宜出现在此镜像中的软件包尽量做到不下载，不安装。

如何评判适不适宜呢？这就需要开发者自己凭借经验来判断了。例如，在数据库服务容器中安装一个文本编辑器就有些不合时宜，在一个Nginx容器中安装VNC，也同样不适合。

4. 一个容器尽量只运行一种服务

容器精悍，不只包括软件包上面的精悍，还包括运行时的精悍。为了达到容器可以快速部署的目的，应尽量每个容器只对外提供一种服务。当此容器需要运行其他服务时，通过LINK的方式从其他容器获取服务，而不是在当前容器中再启动另外一种服务。

5. 控制文件层数量

在介绍RUN命令时提到过一个镜像所能维护的文件层是有上限的。当文件层过多时，创建容器将会耗费更多的时间，这和我们推荐的简小精悍原则相违背。

在RUN命令一节中，我们提到可以将相似的命令放到同一个RUN指令中运行，这样可以减少文件层数量。但我们需要在文件层数量和合并命令之间找到一个平衡点，因为一旦命令合并的太厉害，势必会影响到Dockerfile中RUN指令的可读性。天长日久，这个Dockerfile将变得无法维护，或者维护成本很高。

所以需要开发者在文件层数量和可维护性之间找到平衡。

6. 对命令中的参数进行排序

这个规则看起来好像很不重要，但在日常实践中，它却可以发挥“四两拨千斤”的作用。我们看一下下面的示例：

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

通过对参数进行排序，我们可以避免下载到重复软件包，同时增添软件包时更为容易和控制，当前镜像中安装了哪些软件包也会一目了然。而且经过这样编排之后，无形中也提高了Dockerfile的可读性和可维护性。

按照默认的规则，每行会在最后一个字母后面添加一个空格，然后再添加“\”。建议读者也使用这样的规则，以保持风格统一。

7. 尽可能多地使用缓存中的数据

在Dockerfile执行过程中，Docker会将某些命令的执行结果保存成临时镜像，这些镜像都保存在缓存中。在下次执行Dockerfile时，Docker会在执行每一条指令之前检测此条指令缓存值是否有效。只有当

缓存没有或者缓存失效时，才会再次执行此条指令。

如果用户不希望Docker每次都使用缓存中的结果，那么可以通过--no-cache=true来关闭此项功能。

但是通过缓存功能，可以极大提高构建效率，所以用户有必要了解Docker是如何使用缓存的。Docker在什么情况下会加载缓存，Docker又是通过什么规则来匹配缓存的。

Docker加载完基础镜像之后，会比对由此基础镜像所派生出来的所有子镜像。比对规则就是看这些子镜像是否是由Dockerfile后面的命令所创建的，如果后面的命令发生了变化，那么缓存自动失效。如果命令没有发生变化，那么进行后续的比对。

大多数情况下，单纯比较指令是否发生变化就足以判定缓存是否有效，但某些命令需要特别检查，例如，ADD命令和COPY命令。

对于ADD和COPY命令来说，每次会检查其所涉及的文件是否发生了变更，而变更发生的标准就是计算文件的校验和。但在计算校验和时，文件最后修改日期和最后访问日期不会纳入计算范围。Docker每次通过计算本地目录中的文件校验和，并用这个校验和与缓存镜像中的文件校验和进行比对，如果校验和不一致，那么缓存自动失效。

当在验证ADD和COPY命令时，Docker只会检验镜像中的数据，不会检验容器中的数据。例如，如果在容器中运行了RUN apt-get-y update命令，即便更新了软件包，也不会影响到缓存镜像的有效性。

一旦缓存的镜像失效，那么此缓存镜像以后的所有镜像都将自动失效。言外之意就是，一旦某条命令发生变化导致镜像失效，那么此条命

令后面的所有命令将不再检查缓存，都会被一一执行。

以上是在使用Dockerfile过程中的最佳实践。但在Dockerfile内置命令中，有一些命令也存在优化的空间，下面就看一下命令如何进行优化。

（1）FROM命令

在使用FROM命令时，推荐使用官方提供的镜像作为基础镜像。同时尽可能地多使用Debian版本的Linux。因为Debian Linux易于操控，而且Debian Linux可以将系统尺寸控制的非常小，目前已经有小于100MB的镜像面世了。在Docker环境中，体积越小越利于镜像分发和部署。

（2）RUN命令

一般来说，为了Dockerfile具有更好的读写性和更好的可维护性在RUN命令中，使用“\”将命令分隔为多行。这一点在上面的最佳实践中已经提到过了。我们在优化RUN时，重点是优化apt-get命令。

不要在单行使用RUN apt-get update命令，这样做有可能会触发缓存问题，而将导致后面的apt-get install命令执行失败。

避免在Dockerfile中执行RUN apt-get upgrade或者执行RUN apt-get dist-upgrade，因为这样做毫无必要。如果用户确认某些软件包已经过时，的确需要更新，那么建议执行apt-get install–y，这样就会自动更新了。

RUN命令默认风格如下：

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo
```

建议大家遵守这样的风格，保持风格一致。通过这样的组合命令，在执行Dockerfile时，Docker会默认使用缓存数据，并且在不需要任何人工干预的情况下顺利安装所需软件的最新版本。

下面是一个实际的RUN命令示例，在这个示例中，Docker会在镜像中安装所需软件的最新版本，同时还会安装s3cmd的指定版本：

```
RUN apt-get update && apt-get install -y \
    aufs-tools \
    automake \
    btrfs-tools \
    build-essential \
    curl \
    dpkg-sig \
    git \
    iptables \
    libapparmor-dev \
    libcap-dev \
    libsqlite3-dev \
    lxc=1.0* \
```

```
    mercurial \
    parallel \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.0*
```

这种风格可以避免出现重复软件安装的问题，同时比下面这种风格更易维护：

```
RUN apt-get install -y package-foo && apt-get install -y package-
```

（3）CMD命令

CMD命令应该执行镜像中一定存在的二进制程序。CMD命令推荐使用CMD [“executable”,“param1”,“param2”…]这样的风格，尤其这个镜像是一个服务类镜像时，例如Apache、Rails或者其他服务。

比如运行Apache时，建议这样运行：

```
CMD ["apache2","-DFOREGROUND"]
```

在某种程度上，这种风格已经成为了服务类镜像的标准风格。

在其他类型的镜像中，大多将CMD中的二进制程序放到某种可交

互shell(bash，python，perl)中执行。例如，CMD[“perl”,“-de0”], CMD [“python”]或者CMD [“php”,“-a”]。这样当准备执行下面命令时，可以确保有一个可以执行程序的shell。

```
Docker run -it python
```

在CMD命令实际使用环境中，如果对ENTRYPOINT还不是十分熟悉，那么应该避免使用CMD [“param”,“param”]这样风格的指令，以免出现一些意料之外的错误。

（4）EXPOSE命令

EXPOSE命令用来设定容器启动之后将会监听哪些端口，所以在使用时，一般就是按照常规分配端口。比如，Apache Web应用应该监听80端口，MongoDB服务应该监听27017端口。

（5）ENV命令

为了确保容器运行时可以正确启动服务，一般在Dockerfile中会提前维护好环境变量。例如，在Nginx镜像中，会提前设置好ENV PATH/usr/local/nginx/bin:$PATH。这样当设置CMD [“nginx”]之后，容器运行Nginx时就会自动启动。

与此同时，ENV命令还可以用来维护一些版本号，方便维护，例如：

```
ENV PG_MAJOR 9.3
```

```
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

（6）ADD和COPY命令

尽管ADD和COPY命令极为相似，但一般来说还是比较推荐使用COPY命令。这是因为相对于ADD命令而言，COPY命令更为纯粹。

COPY仅仅提供了基本的从本地向镜像复制文件的功能，而ADD命令则提供了非常多的附加高级功能，如复制目录、复制远程文件等。而这些高级功能越多，其出错的概率就越高。

如果一定要给使用ADD命令的一个理由，那就是使用ADD命令的自动解压缩功能：

```
ADD rootfs.tar.xz /
```

如果在Dockerfile中，用户需要多次复制文件到镜像中。那么建议不要一次都复制完，而是分多次复制，例如：

```
COPY requirements.txt /tmp/
RUN pip install /tmp/requirements.txt
COPY . /tmp/
```

这样会尽可能地复用到缓存中的数据，某一次的文件变更不至于影响到全部缓存的数据。

Docker官方非常不建议使用ADD命令从远程服务获取文件的功能，因为这会导致镜像的体积变大。通过这种方式获取的文件，将无法删除，从而永远被遗留在镜像中。如果需要从远程服务器获取文件，那么就使用curl或者wget获取。当用户安装完成之后，就可以删除这些文件。下面是两种方式的比较：

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
##不建议这样做
RUN mkdir -p /usr/src/things \
    && curl -SL http://example.com/big.tar.xz \
    | tar -xJC /usr/src/things \
&& make -C /usr/src/things all
##建议这样做
```

除此之外，能用ADD命令的地方都可以换成COPY命令。

（7）ENTRYPOINT命令

ENTRYPOINT命令最好用于设定容器的默认主进程，同时将CMD命令中所设定的值当作ENTRYPOINT的参数使用。

假设我们在一个镜像中进行了下面的设定：

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

那么当我们使用下面命令启动这个容器时，就会输出s3cmd的help信息。

```
$ Docker run s3cmd
```

这是在实际生产环境中经常用到的一种用法，当用户不知道这个容器应该如何使用时，可以输出一些帮助信息来提示用户。

除此之外，ENTRYPOINT命令还用来设定执行一些脚本，这些脚本可以减少用户操作的复杂度。例如，在postgres提供的镜像中，ENTRYPOINT命令就设定执行下面的脚本：

```
#!/bin/bash
set -e

if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi
```

```
    exec gosu postgres "$@"
fi

exec "$@"
```

这个脚本被命名为Docker-entrypoint.sh，然后在Dockerfile中使用这个脚本：

```
COPY ./Docker-entrypoint.sh /
ENTRYPOINT ["/Docker-entrypoint.sh"]
```

当用户使用这个容器时，就可以有多种选择了：

```
$ Docker run postgres
##这样就可以初始化一个postgres服务

$ Docker run postgres postgres –help
##执行后面的命令

$ Docker run --rm -it postgres bash
##不初始化postgres，直接使用bash
```

（8）VOLUME命令

VOLUME命令创建的挂载点尽管可以挂载任何数据，但一般是用来导出数据库数据、配置数据，或者容器本身所产生的文件或目录。Docker官方推荐使用VOLUME命令来达到目录共享的目的。

（9）USER命令

如果容器中的服务的运行不需要特别的权限，那么建议使用USER命令切换到其他用户，以避免可能出现的权限问题。同时建议在Dockerfile最开始时，像下面这样首先创建一个用户：

```
RUN groupadd -r postgres && useradd -r -g postgres postgres
```

尽量不要使用root用户来安装软件，因为root有可能会出现无法使用TTY或者无法处理Linux信号等潜在问题。使用root给用户带来的潜在风险绝对大于使用root所带来的便利。

如果需要使用root权限来启动服务（例如以root身份启动一个守护进程，但不以root身份去执行），这时不必切换到root用户。使用gosu就可以解决这个问题（示例请参见上面的脚本）。

最后使用USER会产生一个新的文件层，因此为了控制稳健层数量和减少镜像复杂度，应尽量避免使用USER频繁切换用户。

（10）WORKDIR命令

为了Dockerfile整个文档的整洁，Docker鼓励多使用WORKDIR来设定当前工作目录。在使用WORKDIR时，建议使用绝对路径来设定。同时可以通过多次设定WORKDIR的方式来减少冗余的路径，例如：

```
RUN cd … && do-something
##这样的路径硬编码，可维护性很差，在Dockerfile中应减少这种情况的发生
```

（11）ONBUILD命令

ONBUILD不会在当前Dockerfile执行过程中执行，而是会在后续的子镜像中运行。可以将ONBUILD想象成父镜像送给子镜像的一个礼物，礼物的内容就是打包的指令集。

这样在编写子镜像的Dockerfile时，就可以少考虑很多问题。因为这些事情，父镜像都已经考虑到位，并且做了适当的处理。

ONBUILD命令主要用在编译类的镜像中，比如python镜像、ruby镜像。从具有ONBUILD属性的镜像中构建而来的子镜像，在命名上最好按照如下的风格：

```
ruby:1.9-onbuild或者ruby:2.0-onbuild
```

虽说ONBUILD命令可以包罗万象，但ADD、COPY命令最好不要囊括在内。因为如果子镜像加载的数据，不包括这些指令所使用的文件时，整个Dockerfile就会执行失败。

下面提供几个官方版本的Dockerfile，以供读者参考。

Go Dockerfile：

```
```

```
FROM buildpack-deps:wheezy-scm

# gcc for cgo
RUN apt-get update && apt-get install -y \
        gcc libc6-dev make \
        --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

ENV GOLANG_VERSION 1.5
ENV GOLANG_DOWNLOAD_URL https://golang.org/dl/go$GOLANG_VERSION.l
ENV GOLANG_DOWNLOAD_SHA1 5817fa4b2252afdb02e11e8b9dc1d9173ef3bd5a

RUN curl -fsSL "$GOLANG_DOWNLOAD_URL" -o golang.tar.gz \
    && echo "$GOLANG_DOWNLOAD_SHA1  golang.tar.gz" | sha1sum -c -
    && tar -C /usr/local -xzf golang.tar.gz \
    && rm golang.tar.gz

ENV GOPATH /go
ENV PATH $GOPATH/bin:/usr/local/go/bin:$PATH

RUN mkdir -p "$GOPATH/src" "$GOPATH/bin" && chmod -R 777 "$GOPATH
WORKDIR $GOPATH

COPY go-wrapper /usr/local/bin/
```

Perl Dockerfile：

```
FROM buildpack-deps
MAINTAINER Peter Martini <PeterCMartini@GMail.com>

RUN apt-get update \
    && apt-get install -y curl procps \
    && rm -fr /var/lib/apt/lists/*
RUN mkdir /usr/src/perl
COPY *.patch /usr/src/perl/
WORKDIR /usr/src/perl

RUN curl -SL https://cpan.metacpan.org/authors/id/S/SH/SHAY/perl-
tar.bz2 \
    && echo '63126c683b4c79c35008a47d56f7beae876c569f *perl-5.20
    && tar --strip-components=1 -xjf perl-5.20.2.tar.bz2 -C /usr/
    && rm perl-5.20.2.tar.bz2 \
    && cat *.patch | patch -p1 \
    && ./Configure -Dusethreads -Duse64bitall  -des \
    && make -j$(nproc) \
    && make test_harness \
    && make install \
    && cd /usr/src \
    && curl -LO https://raw.githubusercontent.com/miyagawa/cpanm:
    && chmod +x cpanm \
    && ./cpanm App::cpanminus \
    && rm -fr ./cpanm /root/.cpanm /usr/src/perl
```

```
WORKDIR /root

CMD ["perl5.20.2","-de0"]
```

Hylang Dockerfile：

```
# Base image
#
# VERSION   0.2
FROM        python:3
MAINTAINER  Paul R. Tagliamonte <paultag@hylang.org>

ADD . /opt/hylang/hy
RUN pip3 install -e /opt/hylang/hy

CMD ["hy"]
```

Rails Dockerfile：

```
FROM ruby:2.2

# throw errors if Gemfile has been modified since Gemfile.lock
RUN bundle config --global frozen 1
```

```
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

ONBUILD COPY Gemfile /usr/src/app/
ONBUILD COPY Gemfile.lock /usr/src/app/
ONBUILD RUN bundle install

ONBUILD COPY . /usr/src/app

RUN apt-get update && apt-get install -y nodejs --no-install-reco
RUN  apt-get  update  &&  apt-get  install  -y  mysql-client  pos
--no-install-recommends && rm -rf /var/lib/apt/lists/*

EXPOSE 3000
CMD ["rails", "server", "-b", "0.0.0.0"]
```

第9章　Docker部署案例

基于Docker部署和管理Nginx集群

据统计，世界上40%的商业网站在使用Nginx作为Web服务器。Nginx作为一款优秀的开源服务器，可同时提供负载均衡、Http缓冲等功能。伴随着Docker逐步步入商业化运作，Nginx官方为Docker hub提供了Nginx镜像。目前此镜像下载次数已经超过340多万次。

下面，我们就以Nginx为例，来看看实际生产环境中是如何使用Docker的。

结合我们在前面章节中所接触到的内容，创建和运行一个Nginx容器最简单的方式就是使用下面的命令：

```
# Docker run --name mynginx1 -P -d nginx
```

通过这个命令，我们可以实时基于最新的Nginx镜像创建一个名为mynginx1的容器实例。因为我们在run命令当中没有使用-i和-t，因此此容器将在分离（detached）模式下运行，只能通过Kill或者Stop命令才能停止。

这种方式虽然简单，但并不是我们在实际生产环境中使用的方式。因为，Nginx服务器需要被外部所访问，但上面的命令并没有提供任何

端口映射，所以不具有任何实际使用价值。在Nginx镜像当中，已经导出了80端口和443端口，因此我们可以直接映射这两个端口。

映射端口的方式有两种，-P和-p。-P的方式，是通知Docker Daemon在创建容器时，在宿主环境当中随机从49153和65535之间挑选一个没有被占用的端口，与容器导出的端口进行映射。这是一种在实际生产环境中经常采用的方案。因为如果我们需要同时创建多个Nginx容器，通过这种方式，就可以做到容器之间相互不干扰。但在使用-P时需要注意，当容器重启时，端口也会随之改变。不想发生这种改变，建议采用通用方式：-p。

因为后面还要继续创建几个Nginx容器，所以这里采用-P的方式，以避免端口冲突问题。

```
# Docker run -P --name mynginx1 -P -d nginx
# Docker ps

CONTAINER ID    IMAGE         COMMAND           CREATED     STATU
PORTS                NAMES

fcd1fb01b145   nginx:latest   "nginx -g 'daemon of 16 seconds ago
0.0.0.0:49166->443/tcp, 0.0.0.0:49167->80/tcp       mynginx1
```

通过Docker ps，我们可以看到随机挑选了49166和49167两个端口来映射容器当中的443和80端口。

为了验证Nginx容器当中的应用已经对外提供服务了，我们可以向

宿主环境的49176端口发起HTTP请求，如果出现下面的界面就表示Nginx服务启动成功。

到这里，我们只是创建了一个运行默认配置的Nginx服务器，这个容器还无法满足我们实际运行需求，所以我们需要对此容器进行一些配置。如果需要对容器进行配置，通过SSH直接访问容器无疑是一个非常好的方式。但很可惜，在实际生产环境当中，为了容器和宿主环境的安全，SSH服务一般来说是关闭的。我们只能采取Docker提供的其他方式来访问容器了。

下面我们按照难易程度，介绍生产环境当中经常使用的三种修改容器中配置文件的方式。

1. 通过修改宿主环境的文件来影响容器

这种方式是将Nginx容器需要使用到的配置文件，统一放到宿主环境的某一个目录当中。当Docker Daemon创建容器时，将此目录挂载到容器中。假设Nginx容器使用的配置文件都在/etc/nginx （容器当中的文件路径），Nginx默认的首页位置在/usr/share/nginx/html（容器当中的文件路径），而这些文件在宿主环境当中的存放位置为：/var/nginx/conf和/var/www。

那么需要执行的命令如下：

```
# Docker run --name mynginx2 -v /var/www:/usr/share/nginx/html:rc
-P -d nginx
```

因为我们使用了-P，所以不会发生端口冲突的问题。

-v表示挂载，而ro表示文件目录是只读权限，因而我们在本地目录/var/www和/var/nginx/conf当中所做的文件修改，会直接影响到Nginx容器。

读者或许会问，ro不是表示只读权限吗？那为什么在容器之中还能修改文件呢？这里的ro指的是此目录在容器当中是只读的。宿主环境当中的权限没有发生任何变化，只要当前用户在宿主环境中对此目录有写权限就可以。

通过此种方式，用户可以自由地在宿主环境当中修改配置文件。但每次启动容器，都需要挂载一次文件。如果跨节点运行Nginx容器，这些文件还要随之移动。这与Docker一次构建，随处运行的理念有些出入，所以就出现了下面的方案2。

2. 将配置文件构建到镜像当中

与方案1不同，方案2选择将需要的文件打包到镜像当中。具体来说，就是以官方的Nginx镜像为基础镜像，把需要用到的配置文件在构建阶段，替换基础镜像中的文件，从而形成一个新的镜像。这样的话，就符合Docker的理念了。只要配置文件不出错，那么通过这个镜像就可以随时创建一个符合我们预期的容器。

下面是构建新镜像所使用的Dockerfile：

```
FROM nginx
```

```
RUN rm /etc/nginx/conf.d/default.conf

RUN rm /etc/nginx/conf.d/examplessl.conf

COPY content /usr/share/nginx/html

COPY conf /etc/nginx
```

可以看出，新的镜像只是替换了部分文件，而基础镜像当中所有的参数都保留了下来。我们通过下面的命令来构建新镜像。

```
# Docker build -t mynginximage1 .
```

注意命令最后的“.”，这表示使用当前目录里面的Dockerfile来构建镜像。

最后，我们就可以通过修改之后的镜像来创建Nginx容器了。

```
# Docker run --name mynginx3 -P -d mynginximage1
```

方案2相对于方案1来说，摆脱了配置文件的干扰，可以自由地通过Pull和Push来分发自定义Nginx应用，但却丧失了修改文件的便利性。如果在方案2中需要修改文件，那么就需要挂载到Nginx容器当中对文件进行编辑。这种情况在生产环境是非常危险的，因为很有可能会修改其他

文件，此时就可以考虑方案3。

3. 通过专用的配置容器来创建容器

为了能自由分发镜像，同时也可以安全地修改配置文件，此时就需要一个专用的配置文件容器。这个容器只有一个用途：提供配置文件。

具体做法是：将Nginx所有需要使用的文件都集中到一个容器中，当需要创建容器时，就选择从此容器中挂载配置文件。下面我们看一下详细做法。

首先构建一个保存着配置文件的容器：

```
FROM nginx

COPY content /usr/share/nginx/html

COPY conf /etc/nginx

VOLUME /usr/share/nginx/html

VOLUME /etc/nginx
```

然后使用这个Dockerfile来构建镜像：

```
# Docker build -t mynginximage2 .
```

构建成功之后，我们就开始使用此镜像来创建容器：

```
# Docker run --name mynginx4 -P -d mynginximage2
```

此时，一个正在提供Nginx服务的容器就开始运行了。这些步骤和方案2没有太大区别，而真正有区别的是下面：

```
# Docker run -i -t --volumes-from mynginx4 --name mynginx4files c
 root@b1cbbad63dd1:/#
```

我们通过--volumes-from参数，从mynginx4“继承”了配置文件信息（/usr/share/nginx/html和/etc/nginx），此时在mynginx4files当中，只能看到继承的文件信息，而无法看到其他文件。所以在这个容器当中，就可以任意编辑这些配置文件了，并且不用担心会影响到其他文件，当然也不用担心会影响到其他容器。

因此方案3可以理解成是方案1和方案2的结合体。

这是对Nginx的配置环节，在实际生产环境当中，我们是无法保证Nginx服务器是7×24小时正常运转的。因此一旦Nginx出现问题后，第一选择是查看日志，那么我们如何通过Docker来查看Nginx的日志呢？

Docker作为一个可用于商业目的的开源工具，自然考虑到了日志的问题，所以Docker为每个容器都提供了一个保存应用日志的目录，默认地址是/var/lib/Docker/containers/<container id>/<container id>-json.log。

这里的container id是全ID，所以我们可以通过下面的命令来获取。

```
# Docker inspect --format '{{ .Id }}' <container name>
```

直接查看这个日志文件，是不容易看懂的，所以大多数人都会选择通过下面的方式来查看日志：

```
# Docker logs <container name>
```

当然也可以通过Rest API的方式来查看，但前提是Docker Daemon对外提供了Rest API。启用Rest API的方式如下：

```
DockerOPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/Docker.sock
```

设定此参数之后，重启Daemon才会生效。

后面就可以使用前面章节介绍的Rest API来获取日志了：

```
http://<Docker host>:4243/containers/<container name>/logs?stdout
```

以上这些都是基于Docker来管理日志，但在实际场景中，我们同样可能会提取日志，那么又该如何操作呢？结合管理Nginx的方案3，我们

提出了定制容器日志的方案。

通过下面的Dockerfile，构建一个新的镜像，当需要对日志进行分析时，就创建一个容器（包含日志分析工具）来单独处理日志。

```
FROM nginx

COPY content /usr/share/nginx/html

COPY conf /etc/nginx

VOLUME /var/log/nginx/log
```

上面说了很多修改配置文件的方案，但好像忽视了一个问题：如何重启Nginx，以便加载新的配置文件？

这就牵涉到如何控制容器了。目前来说，只能通过Docker restart的方式来重启容器。但Nginx团队为了迎合Docker，已经表示在不远的将来，会提供一个HUP信号来加载新配置文件，到那时通过Docker kill－s HUP就可以完成Nginx重启。

目前来说Nginx和Docker配合的已经非常不错了，通过Docker，用户可以跳过烦琐的配置，毫无压力地创建Nginx容器，也可以基于官方的Nginx镜像自由发挥，构建出符合特定需求的新镜像。

通过以上的演示，希望读者可以加深对Docker CLI的使用，同时也可以灵活掌握Docker在实际生产环境当中的一些用法。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

第三篇　Docker进阶

本篇包括

- 第10章　Docker运行剖析
- 第11章　Docker内核讲解
- 第12章　Docker资源调度

电子书寻找看手相 钉钉或微信pythontesting

第10章　Docker运行剖析

10.1　Docker的生命周期

在Docker体系结构中，最重要的组件有三个，分别是Docker Daemon、Docker Image（镜像）和Docker Container（容器）。

而这三个组件中，Docker　Daemon是负责维护Docker运行的守护进程，担负着资源管理、任务调度等多项功能。而Docker　Image则属于静态文件系统。最后的Docker　Container是基于Image真正提供应用服务的计算单元。

因此这三者的关系可以说是，Container基于Image被Daemon创建和管理，来实现提供服务的功能。也正因为Container负责提供服务，所以处于核心位置。因而当我们谈论到Docker生命周期时，更多的指的是Docker Container的生命周期。

图10-1展示了Docker Container从创建到销毁的完整的生命周期。

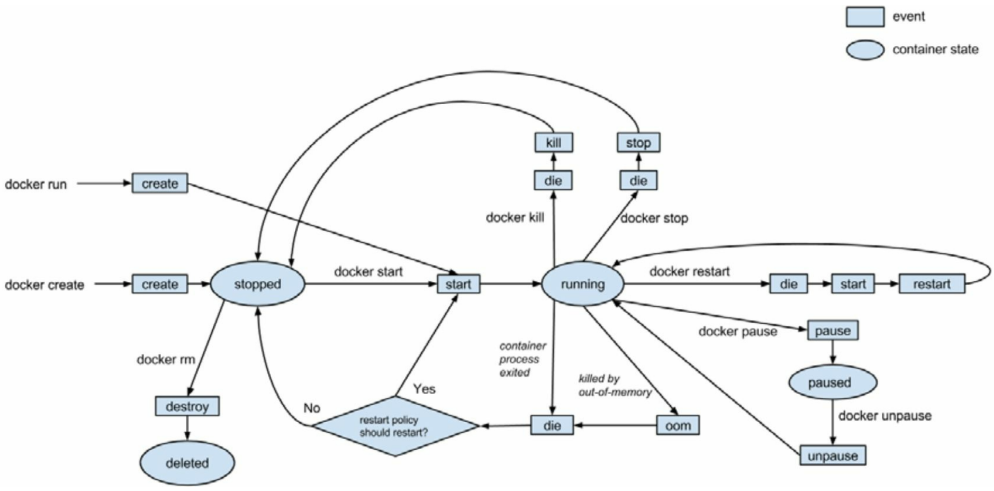


图10-1 Docker Container的生命周期

Docker Container的生命周期以创建为起点。Docker创建一个容器有两种途径：Create和Run。这两个命令都会创建一个容器，但从图10-1中可以看出，这两个命令最终的效果有所不同。使用Create创建的容器状态为Stopped，而使用Run创建的容器状态为Running。这是因为Create命令只会创建此容器所需要的文件系统，而不会给其分配运行资源，也就是不会分配CPU、Memory、Network和Process。只有在明确对此容器调用Start命令之后，Docker Daemon才会按照配置参数给其分配相应的运行资源，而这时此容器就会处于Running状态。

Run命令会连续调用Create和Start命令，对目标容器创建文件系统和分配运行资源。

从图10-1中也可以看出，容器会从Running状态转换到Stopped状态。此种状态转移分为两种情况：正常关闭和异常关闭。

正常关闭指的是用户调用Kill命令或者Stop命令，通过Docker Daemon对正在运行的容器进行资源回收。经过Docker Daemon的资源回收之后，此容器所有内存中的数据都会被销毁，但其文件系统仍然存

在。此时的状态与使用Create命令之后的效果相比，只是文件系统多了一部分数据。此后仍然可以通过Start再度启动，将状态转换至Running状态。

异常关闭发生的原因有很多，在实际运行当中主要有以下两种情况。

1. OOM（内存溢出）

为了保持主机环境和Docker中各个容器的运行稳定性，Docker在处理OOM事件时采取了“熔断器”和“耐压舱”的处理机制。

如果容器中的应用耗尽了主机系统分配给容器的内存限额，就会触发OOM事件。例如，在容器当中，部署了一个python脚本来统计文本中单词出现的频率。假设主机分配给此容器的内存上限为500MB，当脚本申请的内存大于500MB时，此容器就会触发OOM事件。而在这种情况下，此容器将会被强制关闭。

但需要注意的是，此时关闭容器的并非是Docker Daemon，而是宿主机操作系统。因为一个容器其实就是一组运行在宿主机操作系统当中的进程，宿主机操作系统通过cgroups对这组进程设定资源上限，当这些进程申请的资源到达上限时，触发的是宿主机操作系统的内核OOM事件，因此最终是由宿主机内核来关闭这些进程，这就是“熔断器”机制。只要目标达到一个阀值就销毁它，以免情况恶化。

如果用户不想关闭这个容器，那么可以选择--oom-kill-disable来禁用OOM-Killer。使用此参数时，仍需要注意，如果使用-m设置了此容器内存上限，那么当容器到达内存资源上限时，主机不会关闭容器，但也不会继续向此容器继续分配资源，此时容器将处于hung状态，这种机制

就是“耐压舱”。只需要将最坏的情况封闭在一定范围之内，而不至于蔓延出去。

如果用户使用了--oom-kill-disable，但也没有使用-m来设定上限，因而此时此容器将会尽可能多地使用主机内存资源。换言之，主机内存有多大，它就将用多大。

2. 进程意外退出

每个容器内部都存在一个Init进程，容器中其他所有进程都是此进程的子进程。一个容器运行不退出，表现在主机系统上面就是容器当中的Init进程和其子进程处于运行状态。

如果一个子进程因为某种原因造成了退出，那么其父进程也会同步退出，直至Init进程也退出。当Init进程退出时，也就代表着此容器被关闭。

与OOM不同，Docker目前没有任何机制可以监测到此时的进程退出属于正常退出还是异常退出。

但无论是正常关闭还是异常关闭，此容器的状态都会转换到Stopped。当出现容器关闭情况时，Docker Daemon会尝试再次重新将此容器由Stopped状态转为Running状态。当然不是所有的容器出现关闭时，都会有此待遇。只有设置了--restart参数的容器，Docker Daemon才会特别留意。

而在Docker容器的生命周期中，除了Running态和Stopped态之外，还有另外一种状态：Paused状态。

处于此种状态的容器，看似存在却无法做任何事情。究其原因，是

Docker“剥夺”了此容器的CPU资源。而其他资源，如Memory资源、Network资源等还保留未动。如此一来，失去了CPU资源的进程，是不会被主机内核系统所调度的，所以此容器就处于“冰封”状态。

而为其解封的方式非常容易，重新赋予其CPU资源就可以。

因此，一个容器完整的生命周期就是在Running→Paused→Stopped三种状态之间相互转换。而是否具有运行资源就是区分容器状态的唯一标准。

归根结底，Docker管理容器的过程就是Docker重新分配主机资源的过程，而主掌资源分配大权的角色就是Docker Daemon。

这样就引出了下一节我们要讨论的主题：Docker Daemon。

10.2 Docker Daemon

前面我们介绍了Docker中容器的完整生命周期。在Docker当中，负责管理容器生命周期的是Docker Daemon，但Daemon的作用并不限于管理容器，它可谓是Docker环境的大管家。

我们来看一下Docker Daemon主要负责哪些事情。

（1）Daemon负责任务调度，它需要将客户端传来的命令请求转化为特定的任务。

（2）Daemon负责维护镜像数据。一个完整的镜像由许多的子文件层组成，而这个镜像的依赖关系就是由Daemon来维护的。

（3）Daemon负责容器虚拟化。虚拟化是一个较大的概念，具体来说就是资源分配和资源隔离。

（4）Daemon负责容器生命周期。Daemon需要根据用户指令和容器自身状态来维护容器的生命周期。

这四项任务的关联关系如图10-2所示。

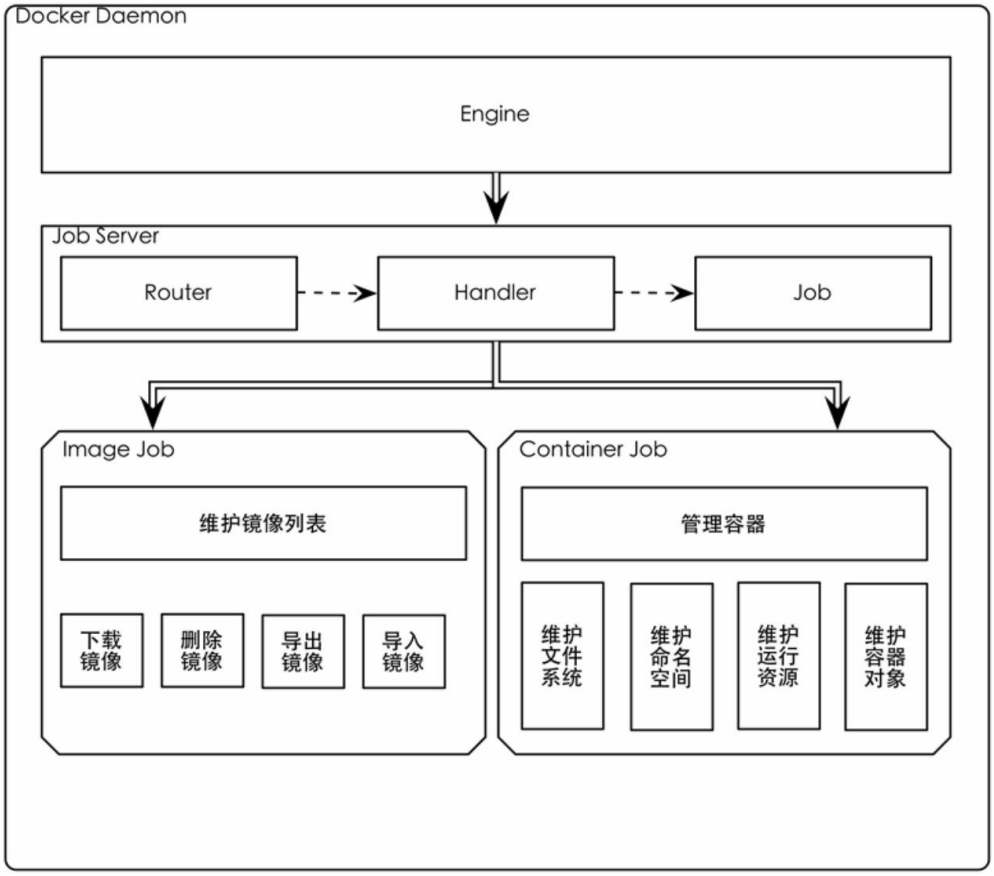


图10-2 关联关系

图中的Engine模块用来监听来自客户端的Restful API请求，Engine既可以通过unix://来监听本地socket文件（默认是Docker.sock），也可以通过tcp://的方式来监听某一个端口。当然Engine也可以选择两种方式同时监听。

Engine模块会将接收到的命令请求转化为具体的Job，而转化的依据就是存放在Daemon当中的API路由表。每一个API都有唯一与之相对应的Handler函数，在此Handler函数中，会有一系列的逻辑来处理客户端请求。而负责处理请求的逻辑函数，在Docker Daemon当中就以Job的形式运行。

Router、Handler和Job这三项共同组成了JobServer。而Engine每接收到一次来自于客户端的处理请求，就会生出一个JobServer。当Job完成作业，并将数据返回给客户端之后，此JobServer也会随之销毁。

在Docker Daemon中，只有两种Job类型：Image Job和Container Job。Image Job重点在于通过AUFS文件系统来维护镜像数据，而Container Job则侧重于管理资源。

我们通过图10-3来梳理一下Docker Daemon是如何通过LXC来管理资源的。

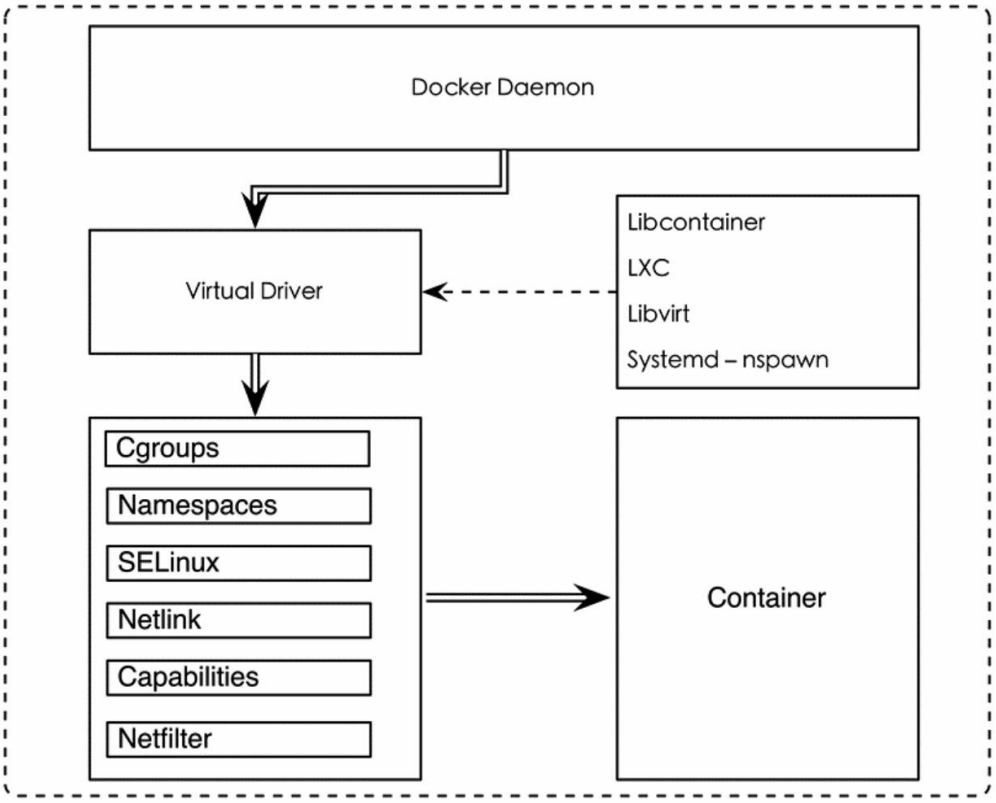


图10-3 Docker Daemon管理资源的过程

Daemon并不会参与资源分配和隔离的工作，这些工作都是由虚拟化驱动来完成的。目前在Docker当中，充当虚拟化驱动的有LXC、Libcontainer、Libvirt和nspawn。其中，Libcontainer是Docker自行开发用于管理资源的驱动库，也是目前默认使用的驱动库。

Daemon通过驱动库来管理Cgroups、namespaces、SELinux等各个组件，然后再基于所创建的资源使用方案产生一个可用的容器。

通过上面的介绍，我们应该明白了Daemon在Docker环境当中所扮演的角色。可以说Docker中所有的事情都需要Daemon参与并处理，虽然有些事情并非是由Daemon直接完成的，但一定是在Daemon监控下进行的。

因此可以说，Daemon就是运行在幕后的大管家。而站在前台，离用户最近的就是Docker CLI （Docker众多的客户端命令）。

10.3 Docker CLI

Docker的横空出世，对SaaS和PaaS的发展和普及提供了一种非常好的方案。在Docker推出之前，几乎所有的SaaS和PaaS都是使用定义良好的堆栈编写，不同平台之间无法相互迁移，有些应用甚至在同一平台之内也无法低代价的迁移。

Docker的出现，打破了这种局面。通过Docker，各种应用之间做到了相互隔离。同时也受益于Docker标准化的特点，用户可以将各自的应用封装成轻量、可移植和自解释型的镜像。并且借助于Docker的轻量化的特性，用户可基于镜像大量创建相互低耦合的应用。

凭借Docker的这些特性，创建SaaS和PaaS平台的门槛一下就降低了不少。但作为一种可商业化运行的解决方案，如何便利地运行和维护是其无法回避的一个问题。因此Docker提供了大量的CLI命令来提高运行效率，降低维护成本。

Docker客户端与Docker服务端交互的大致流程如图10-4所示。

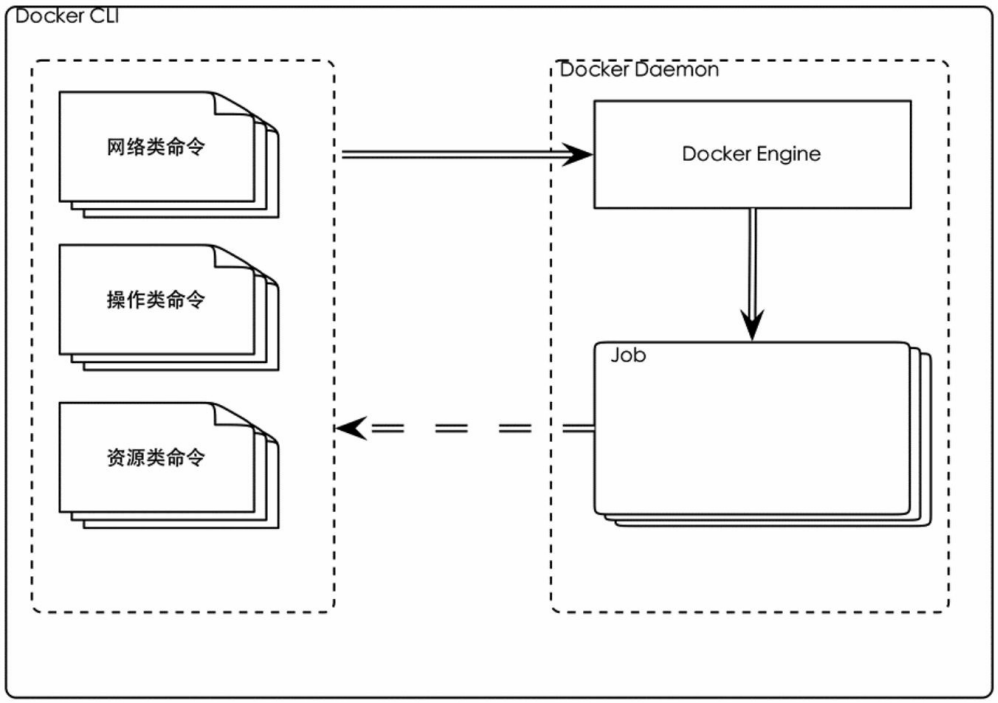


图10-4 Docker客户端与Docker服务端交互的大致流程

按照Docker的架构，每一个Docker CLI都必须通过Rest API的方式调用Docker Daemon完成某件特定的工作。对于Docker当中每一个CLI命令应该如何使用，同时又是如何实现的，我们在第5章和第6章当中已经详细讲解过了。但如何安全地使用Docker CLI，却一直没有提及。因此在本节当中，我们会补充如何安全调用Docker API。

首先需要明确一点：世界上没有绝对的安全，所有的安全都是相对的安全。再好的机制和方案，也离不开人的执行，所以有一套好的管理方式远比一套好的安全框架重要。

我们目前知道所有的命令都是通过RestAPI的方式来实现的，因此如何提防“别有用心”的人恶意调用Rest API就是防范的重点，失误操作不在考虑范围之内。

Docker CLI为了安全，做了两方面的防范：确保请求来自于可信任

的人和确保请求不包含恶意。

1. 确保请求来自于可信任的人

最可信任的来源自然是本地，因此Docker默认只支持来自于本地的处理请求。此时Docker只监听位于本地文件系统中的Docker.sock文件，同时也支持监听来自于外部的请求，这时Docker支持通过证书的方式来相互验证对方身份，如图10-5所示。

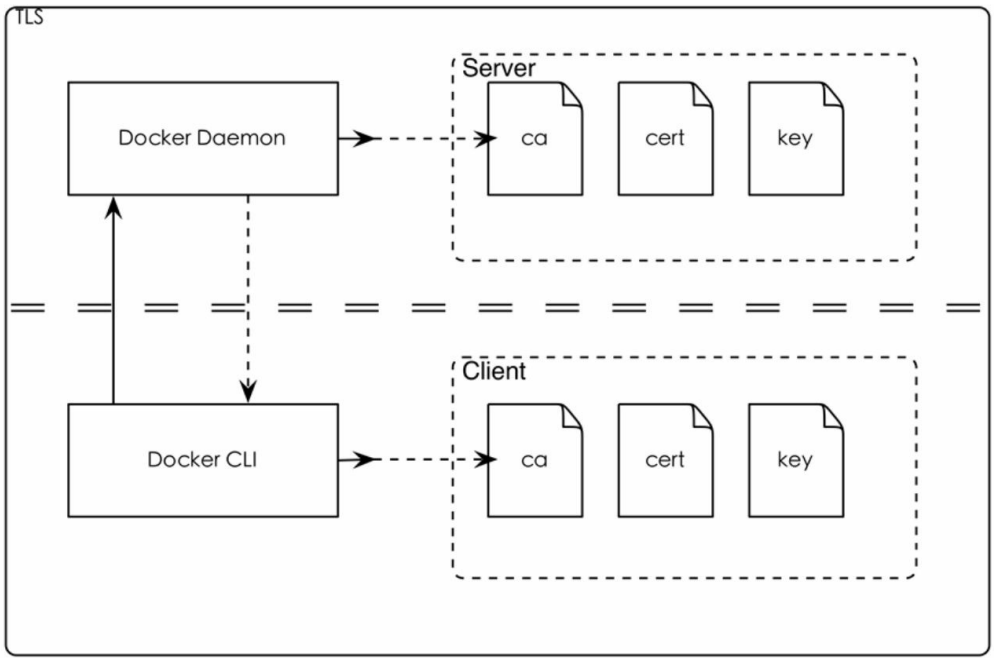


图10-5　通过证书的方式验证身份

当每一次Docker　Client需要调用Daemon时，都必须上传Client使用的证书。Daemon在处理请求之前会验证Client上传的证书是否是Server端所使用的证书。只有上传合法证书，Server才会处理请求。而Client也会验证Server返回的证书是否真实有效。

通过双向认证证书，Docker　Daemon可确保请求的确来自于可信任的对方。

2. 确保请求不包含恶意

在设计Rest API时，对于安全而言需要考虑许多环节。例如，防范重放攻击、防范信息篡改、防范信息丢失等。为了解决这些问题，可采取Oauth2认证，一次性Token等各种解决方案。

但Docker并没有这么做。因为Docker通过证书的方式确认了请求来源是合法的，那么Docker下一步需要做的就是检查客户发来的RestAPI请求中就否有恶意请求。

什么是恶意请求呢？

比如客户端请求Daemon创建一个运行Web应用的容器，那么在此请求中就不应该再出现创建其他容器的命令。

再比如客户端需要销毁一个容器，那么Daemon就必须十分小心地检查，确保此请求只能删除指定容器，而不能伤及无辜。

Docker Daemon针对每一个RestAPI，都有相应的参数检查逻辑。但不可否认的是，这方面目前还是Docker的弱项。Docker现在的检查仅仅只能满足运行需求，无法应对更为复杂的场景。如果需要基于Docker去构建一个面向商用的PaaS平台，那么对于Rest API的参数检查可能需要投入相当大的精力进行重构。

在本章当中，我们依次介绍了Docker Container、Docker Daemon和Docker CLI，这三个模块构成了Docker运行的主体。CLI是命令发起者，Daemon是命令执行者，而Container则是最终的服务提供者。这三者相互协作，才让用户能够得到其所期望的结果。

在下一章中，我们将会开始接触Docker背后的资源管理和虚拟化内

容，从内核的角度来看Docker是如何从无到有，创建一个容器的。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

第11章　Docker内核讲解

在Docker之前，如果需要实现虚拟化，KVM几乎是唯一可选的成熟方案。KVM是Kernel-based Virtual Machine的简称，它需要硬件支持，例如，Intel VT技术或者AMD V技术。对于KVM所实现的虚拟化环境，会拥有完整的进程调度器、存储管理器等内核模块，并且是直接基于硬件级的完全虚拟化。

而Docker选择了namespaces＋cgropus作为其实现虚拟化的方案。但namespaces和cgroups都是软件级的资源隔离和控制，仍然是对所在主机内核资源的分割和再分配，所以说Docker的虚拟化方案严格意义上来说其实是伪虚拟化。

既然namespaces＋cgroups是伪虚拟化的方案，那么Docker为什么还要采取这种方案呢？正所谓尺有所短，寸有所长。namespaces＋cgroups相对于KVM而言，性能损失最低，资源利用率最高，而这就是namespaces＋cgroups在私有云方面大行其道的原因。

Docker所采取的namespaces＋crgoups属于操作系统虚拟化方案，不像KVM那么形象直观，所以在本章当中，我们先从虚拟化的资源隔离性和数据便携性入手，讲解namespaces和文件系统，而在第12章，我们则会重点从资源可控制性入手讲解cgroups。

11.1　Docker背后的Namespace

电子书寻找看手相 钉钉或微信pythontesting

Namespace不是最近刚出来的新技术，早在Linux内核2.4.19中，就包含了最早的Mount Namespaces。但因为只有一个Mount Namespace，无法满足实际复杂场景的需要，所以在随后的内核版本里面，陆续添加了IPC、Network、PID、User和UTS。

Docker就是借助于这六个Namespace完成了资源隔离。下面，我们就来看看Docker是如何借助这六个namespaces完成轻量级资源隔离的。

1. IPC Namespace

IPC Namespace如图11-1所示。

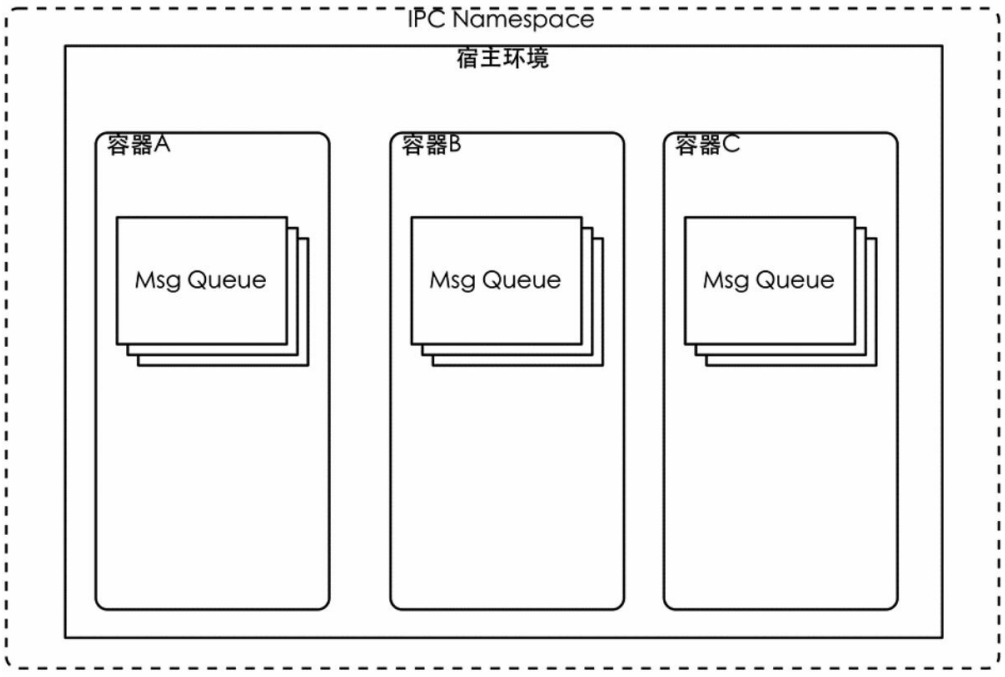


图11-1 IPC Namespace

Linux系统中的进程通信方式主要包括：信号量、消息队列和共享内存。对于一个资源隔离的容器来说，容器内部的所有进程都只能访问主机分配给它的资源。换言之，容器内部的进程通信，对于宿主机来说，就是具有相同PID的进程间通信。因此，Docker首先为此容器创建

IPC Namespace，允许容器内所有进程通过全局唯一的32位标识符访问共享资源。

需要注意的是，Docker自身通信都是通过TCP或者Socket进行通信的，所以IPC Namespace并不是为了容器自身使用，IPC Namespace更多是为了容器内部的应用而预留的。如果容器内部运行的应用需要使用Message Queue，就可以在同一宿主环境中创建多个Message Queue，而不会产生干扰，而后Docker为止容器创建PID Namespace。

2. PID Namespace

PID Namespace如图11-2所示。

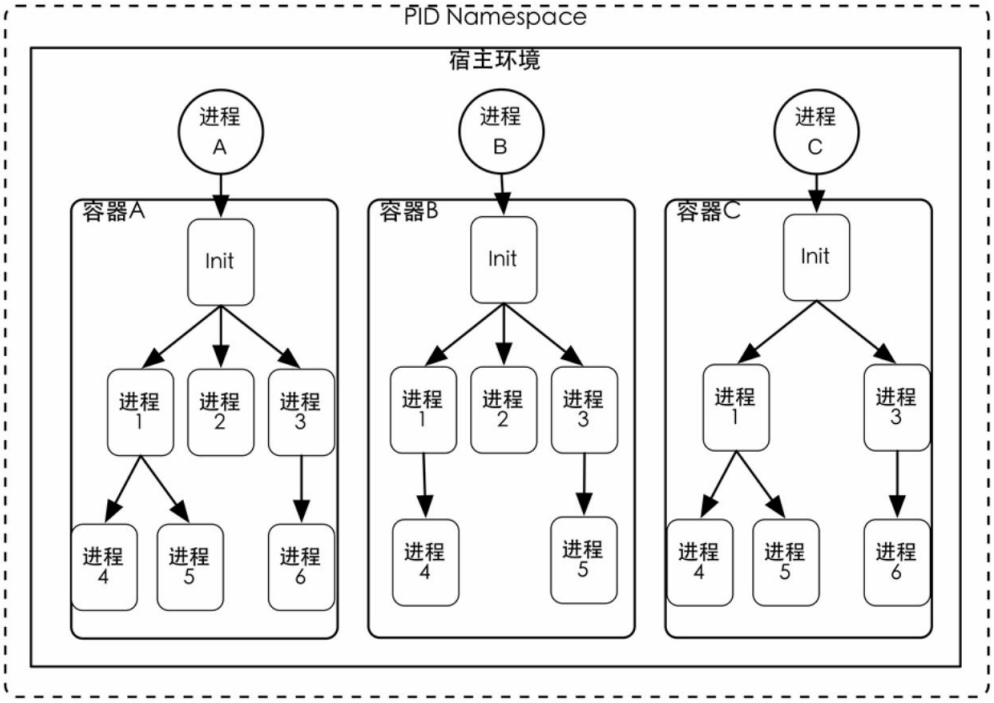


图11-2 PID Namespace

这个Namespace非常重要，它是容器资源隔离的一个重要标志，就是容器之间的进程树相互不可见。通过PID Namespace，每个容器中都

会有一个进程号计数器，容器内所有的进程号会被重新编号。宿主机的内核会维护各个容器中的进程树，在树最顶端的进程号变为1，也就是Init进程。此进程会作为容器内其他所有进程的父进程来执行容器环境的初始化工作。

但此Init进程毕竟不是实际操作系统中的Init进程，在宿主机环境中，此进程就是一个普通的进程。所以容器中的Init进程其实是一个伪Init进程，它无法像真正的Init进程那样“守护”整个容器操作环境。当Init进程下面所有的子进程都退出后，它也就无可避免地需要退出。而这也就是每个容器启动时都需要指定一个不可退出的进程作为CMD的原因。

同时Init进程作为容器中所有进程的父进程，还肩负着关闭和销毁容器的责任。因为借助于PID　Namespace在容器中所产生的进程树，处于树顶点的Init进程可以看到容器中所有的子进程，并且可以通过信号量来影响子进程的行为。所以Docker CLI中的Stop或者Kill命令的本质，就是向容器中的Init进程发送SIGSTOP信号或者SIGKILL信号。一旦容器处于顶点的Init进程被销毁，那么和其处于同一个PID　Namespace的所有进程都将会接收到内核发送的SIGSTOP或者SIGKILL信号而被销毁。

通过PID　Namespace，Docker为容器创建了独立的进程资源。但对于一个资源完全独立的容器而言，这还远远不够，因此Docker后面就为此容器开始创建UTS Namespace。

3. UTS Namespace

一个拥有独立进程树的容器，仅仅只能在宿主环境层面可以被视为独立节点。但Docker作为云平台部署工具来说，其所创建的容器必须要

在网络层面可以被设为一个独立节点。为了实现这个目标，Docker为每个容器都创建了UTS Namespace，如图11-3所示。

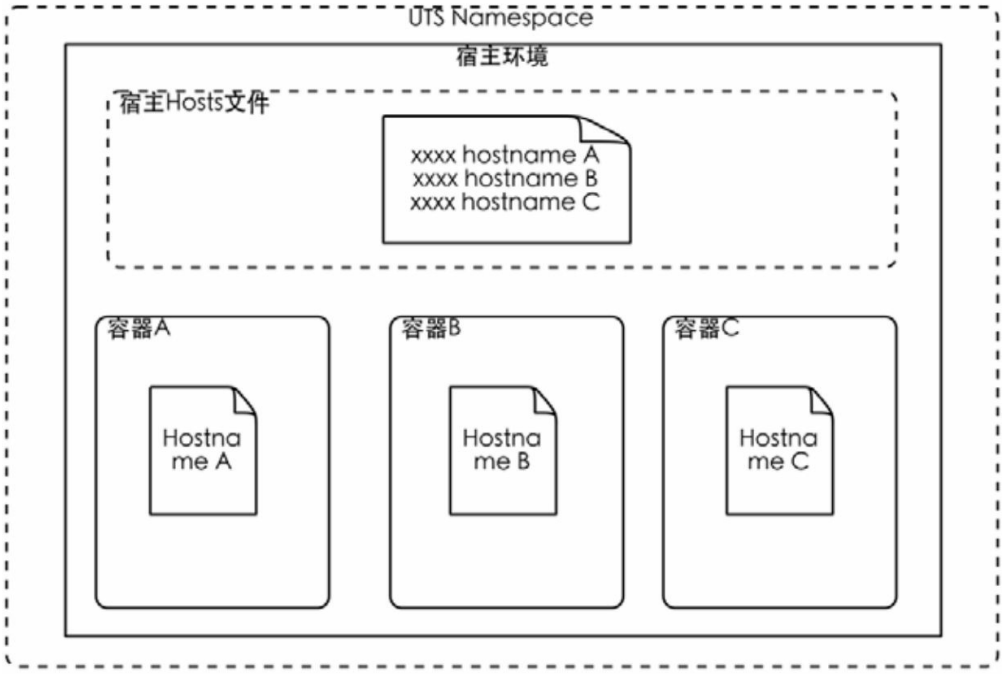


图11-3 UTS Namespace

通过UTS Namespace，每个容器都拥有独立的主机名和域名资源。虽然相对于其他Namespace而言，UTS Namespace只设置了容器主机名和容器域名，但正因为这两个属性在每个容器当中都是独立且唯一的，因此每个容器在网络中都可被设为单独节点，而非宿主机环境中的单独进程。

至此，Docker创建的容器通过PID Namespace拥有了独立的进程树，通过UTS Namespace作为独立节点而拥有了独立主机名和域名资源，同时每个容器也借助于IPC Namespace，可以毫无干扰地使用IPC资源。那么我们是不是就可以随心所欲地在容器当中部署应用了呢？答案是否定的，因为当我们在容器A中部署一个Tomcat应用监听8080端口，在容器B中再部署相同的Tomcat应用同样监听8080端口时，容器B中就

会报出“端口已占用”的错误。这是因为我们只是在进程资源方面完成了独立，还没有在网络层面完成资源独立。

4. Network Namespace

Network Namespace如图11-4所示。

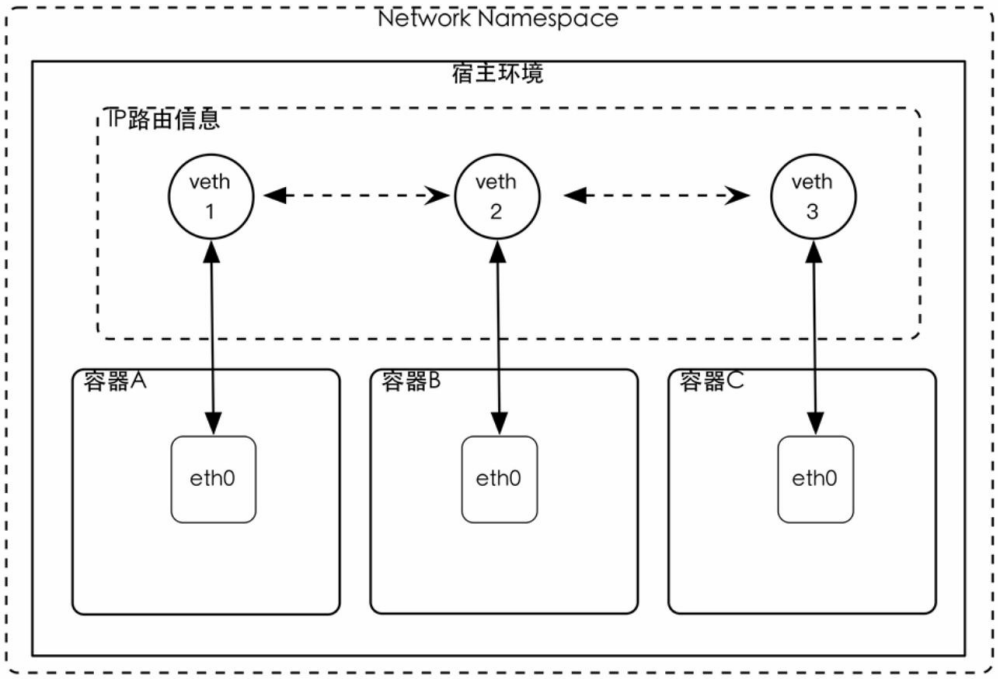


图11-4 Network Namespace

Docker通过Network Namespace为每个容器隔离网络资源。一个典型的Network Namespace包括IP协议栈（IPV4或者IPV6）、IP路由、端口信息和物理设备（网卡）。在Linux系统中，一个物理设备最多只能被包括在一个Network Namespace中，但我们所创建的容器却不止一个。为了解决这个问题，Docker为每个容器的Network创建一对虚拟网络设备，一个名为eth0，放置到容器当中；另外一个名为vethN，放置到宿主机环境当中。

这样从一个设备中流入数据，就可以从相对应的另外一个设备中读

取到数据。虽然从宿主机内核层面来看，其实这两个网卡只是管道的两端；但从容器中来看，这就是两块实际的网卡。通过这种方式，可以达到容器和内核之间数据流通的目的。但为了达到不同容器可以作为网络单独节点进行数据交互的目标，Docker在Network Namespace中还为每个容器设置了IP路由表。宿主机环境此时就充当了路由器的角色，因此毫不相干的两个容器就可以通过其内部的"网卡"互通数据。

从用户的角度来看，这个容器就完全具有了独立的网络资源。虽然从本质上来说，是通过宿主机环境的管道机制来完成的数据转发，但用户对此毫无感觉。用户只会觉得当前容器可以随意使用网络资源，包括监听任意端口和网络通信。

如果容器需要访问外部网络呢？也很简单，Docker只要为此容器增加一条路由规则即可。通过上面所讲述的Network Namespace原理，用户甚至可以为一个正在运行的容器动态增加虚拟网卡，但在实际情况中没有这个必要。

现在的容器又拥有了独立的网络资源，我们还需要为它再添加其他独立资源吗？是的，作为一个可以独立运行的操作系统而言，用户资源也必须是独立的。在容器中，用户应当可以在普通用户和root用户之间自由切换。这也就意味着一个在宿主机环境当中非常普通的用户，可以在其所创建的容器当中拥有root这个超级用户的权限，User Namespace可以用来解决这个问题。

5. User Namespace

User Namespace如图11-5所示。

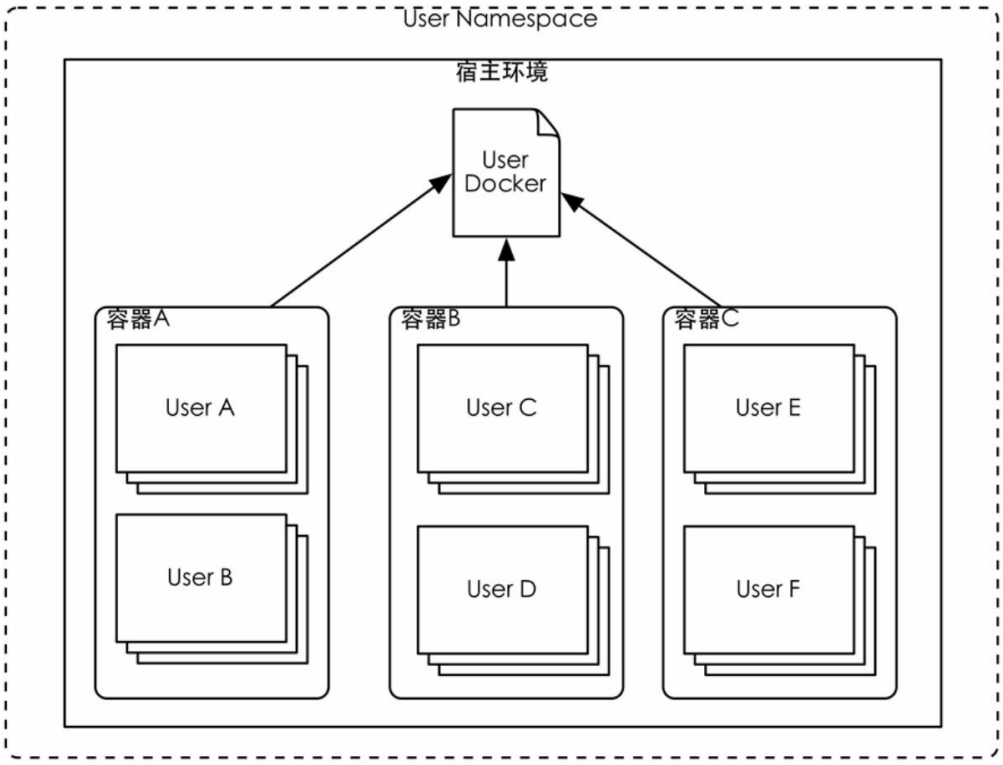


图11-5 User Namespace

Docker通过User Namespace隔离了所有与用户相关的资源，包括用户ID、用户组ID、用户权限。但无论容器中的用户怎么变，始终对应的是宿主机环境中所创建容器的用户，而且这种关联关系通过/proc/[pid]/uid_map和/proc/[pid]/gid_map这两个文件予以保存。

我们这里讨论了Docker通过这五种Namespace完成了资源隔离，但距离真正的资源隔离还差最后一步：文件系统的隔离。如果没有文件系统的隔离，那么即便通过PID Namespace实现了各自进程树，但通过ps命令仍然能看到宿主环境所有的进程，因为容器读取的是/proc中的数据，并且User Namespace也无法实现，因为同一个文件无法从属于多个用户。

6. Mount Namespace

Mount Namespace如图11-6所示。

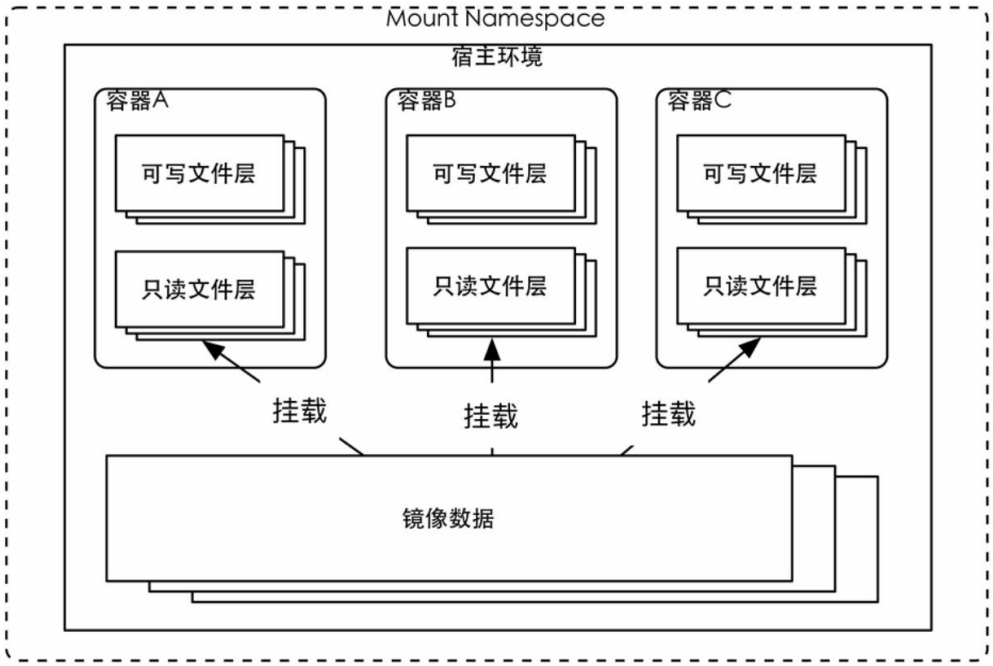


图11-6 Mount Namespace

Docker通过Mount namespace为每个容器分配了独立的文件系统。因为在Mount Namespace当中，可以通过隔离文件挂载点的方式提供隔离的文件系统，因此，Docker为每个容器创建一个其独有的目录，并且将此容器所依赖的镜像文件层按照先父后子的顺序，逐层挂载到此目录当中。隔离完成之后，不同的Mount Namespace之间的数据互不影响。同时，Docker会将当前目录设置为read-only模式，对此目录所做出的所有写操作都将体现到另外一个目录，而这个目录也就是我们一直所说的writable文件层。

讨论到这里，一个具有独立进程树、独立文件系统、独立网络资源、独立主机名和独立用户环境的容器才真正建立完成。而我们也可以看到，Docker通过Namespace提供的虚拟化与KVM相比，属于伪虚拟化。因为此时的资源独立仍是建立在共享资源的前提下，比如共享的内

核、共享的内存、共享的CPU以及共享的物理网卡，等等。

虽然Docker通过Namespace完成了资源独立，但仍然还有需要改善的地方。例如，PID Namespace当中，容器的Init进程仍然无法“守护”子进程；Network Namespace中没有考虑到网络防火墙的问题；而User Namespace作为刚刚纳入Linux内核的Namespace，在权限控制和兼容多种文件系统中，仍然还存有一些问题。

所以我们需要看到Docker有些地方不错，但有些地方仍需要用户根据实际应用场景进行完善和扩展。

11.2 Docker的文件系统

在本节之前，我们经常会谈论到AUFS文件系统，但却一直没有认真地讲解过AUFS。在本节，我们会弥补这个遗憾，认认真真地讨论一下Docker的文件系统。

在正式开始讲解AUFS之前，我们需要先熟悉一下Linux的启动过程，因为Docker启动容器的过程就是Linux启动过程的翻版，所以只要熟悉了Linux如何启动，理解Docker如何使用AUFS就易如反掌了。

Linux启动自然是从加电BIOS自检开始，但这一部分和我们的讲解重点相差甚远，因此直接跳过，我们直接进入到Grub加载bootfs阶段。

bootfs主要包括：bootloader和kernel（内核），而bootloader主要用来启动内核。内核看起来很神秘，其实只有三件工作：探测当前硬件设备，并且根据探测到的硬件设备初始化相关驱动，最后挂载根文件系统。这三项工作看起来简单，做起来却很难。探测硬件设备较为容易，

通过硬件中断就可以得知，但加载硬件驱动却涉及一个哲学问题：“先有鸡还是先有蛋”。

为什么这么说呢？因为现在的硬件设备更新换代速度特别快，而Linux内核不可能同时容纳所有的硬件驱动（这是一件不可能，同时也无法实现的工作，否则内核永远都无法发布了）。因此当前的做法就是，Linux内核当中只包含最基本的硬件驱动，例如硬盘、键盘、CPU、内存、I/O等，其他的驱动则放到文件系统当中，由内核根据需求再进行加载。

问题就出现在这个环节，驱动被放置在文件系统当中，但内核只有在驱动初始化之后才能挂载文件系统。为了解决这个“先有鸡还是先有蛋”的问题，Linux引入了initrd文件。

initrd文件全称是bootloader initialized RAM disk，本质是由bootloader初始化的一个内存文件系统，与initramfs、rootfs、tmpfs等同属ramfs范畴，都是内存文件系统。initrd文件当中包含有：lib、bin、sbin、usr、proc、sys、var、dev、boot等目录，所以我们可以认为initrd就是一个完整的虚拟根文件系统。initrd文件存在的目的就是为了在内核态和用户态之间建立一个桥梁，让内核加载initrd文件系统中的硬件驱动，再通过读写方式挂载用户态的根文件系统。

所以此时Linux启动顺序就变为：bootloader启动内核，并且将initrd文件中的内容加载到内存当中。内核就会通过initrd中所包含的init脚本完成各种硬件设备的初始化和加载工作，而后内核顺利加载根文件系统。

这一步做完之后，内核运行根文件系统当中的/sbin/init进程，创建

操作系统的1号进程，此后内核就会将操作系统控制权移交给用户空间的Init进程。而这一步也就标志着当前环境由内核态转移到了用户态。自此之后，用户态的init进程开始读取/etc/inittab脚本来执行相应的脚本，从而完成系统初始化，主要包括：设置时间、设置字体、设置随机数生成器、设置网络、设置防火墙，等等，如图11-7所示。

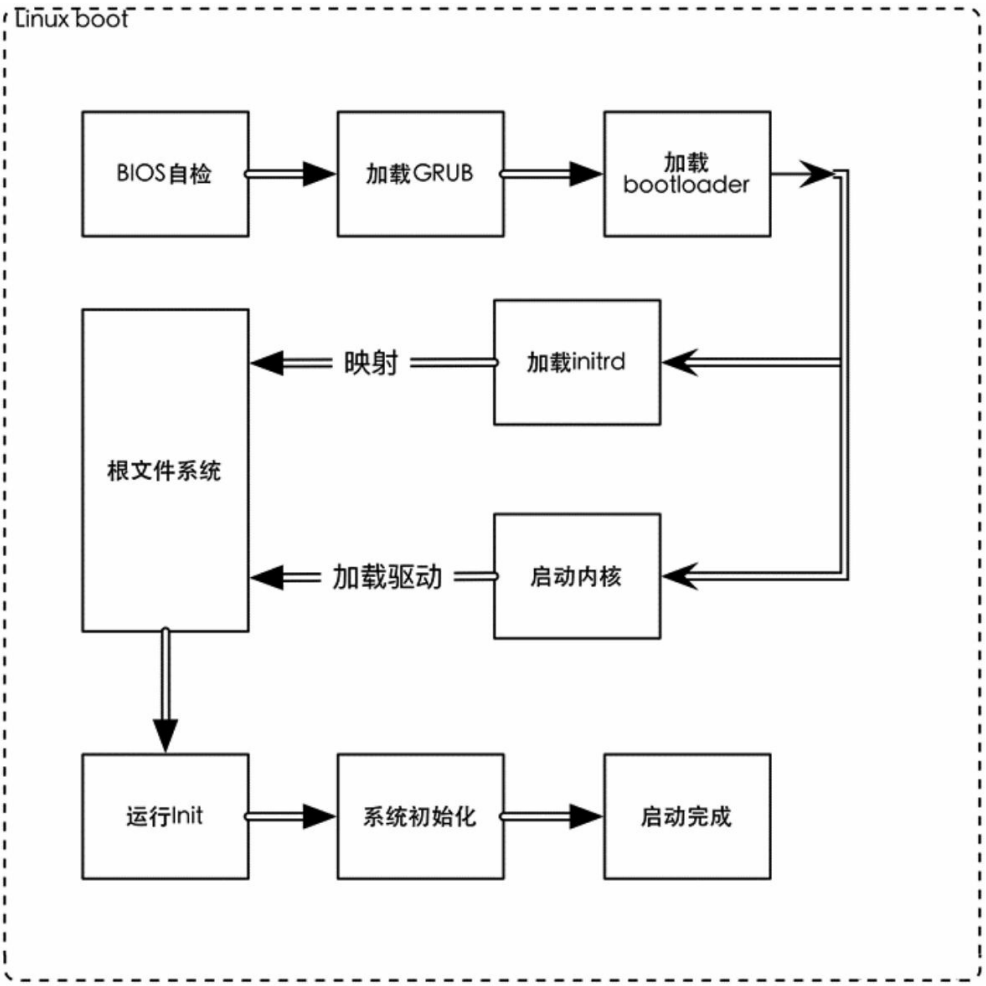


图11-7　Linux启动过程

图11-8就是Linux的启动过程，在Docker当中，也是按照这个顺序来启动容器的。

当Docker　Daemon为容器配置好各种Namespaces参数，准备启动容

器时，首先也是需要加载此容器的文件系统，并在此文件系统基础之上运行容器中的操作系统。目前Docker当中使用最多的文件系统是AUFS，这是一个分层文件系统，它可以将不同的目录挂载到同一个虚拟目录之中，并且形成“文件层”。

AUFS在将不同目录挂载到同一文件系统下时，还可以给不同目录设定只读、只写和写出三种权限。AUFS在挂载目录时，会严格按照先基础目录（父文件层）再增量目录（子文件层）的顺序进行挂载。所有目录都挂载完之后，AUFS会再挂载一个可读可写的目录，而以后所有的写操作都会体现在此目录中。

Docker为容器挂载文件系统的过程和Linux的启动顺序大致相似，但又有所简化。Docker在为容器挂载文件系统时，充当了内核的角色，因此就不需要再加载bootfs了。所以Docker会直接加载rootfs（根文件系统），通常这个rootfs处于整个镜像文件层的最顶层，也就是所有文件层的父级文件层。

但此时的rootfs并不是完整的根文件系统，但对于一个操作系统而言已经足够了，但对于我们的容器来说，还差很多文件。因为在rootfs文件系统之上，有很多的数据变化，此时还没有被体现出来。例如，下面的ubuntu:14.04镜像，其一共包含4个文件层：

```
Docker history ubuntu:14.04

IMAGE     CREATED  CREATED BY SIZE
91e54dfb1179 12 weeks ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0 B
d74508fb6632 12 weeks ago /bin/sh -c sed -i 's/^#\s*\(deb.*univer
```

```
c22013c84729 12 weeks ago /bin/sh -c echo '#!/bin/sh' > /usr/sbi
d3a1f33e8a5a 12 weeks ago /bin/sh -c #(nop) ADD file:5a3f9e9ab88e
```

Docker在第一个文件层（d3a1f33e8a5a）基础之上开始加载第二个文件层（c22013c84729），加载的方式就是通过调用宿主机环境的mount命令。例如，通过：mount-v-t aufs-o br=/var/lib/Docker/aufs/diff/<容器ID>-init=ro none/var/lib/Docker/aufs/mnt/<容器ID>-init的方式将第二层文件层挂载到容器文件系统中。这里面-t aufs表示目标文件系统为AUFS，-o指定我们所要挂载的目录，而br表示挂载的是目录，none表示这里挂载的不是设备文件，而是实际文件，如图11-8所示。

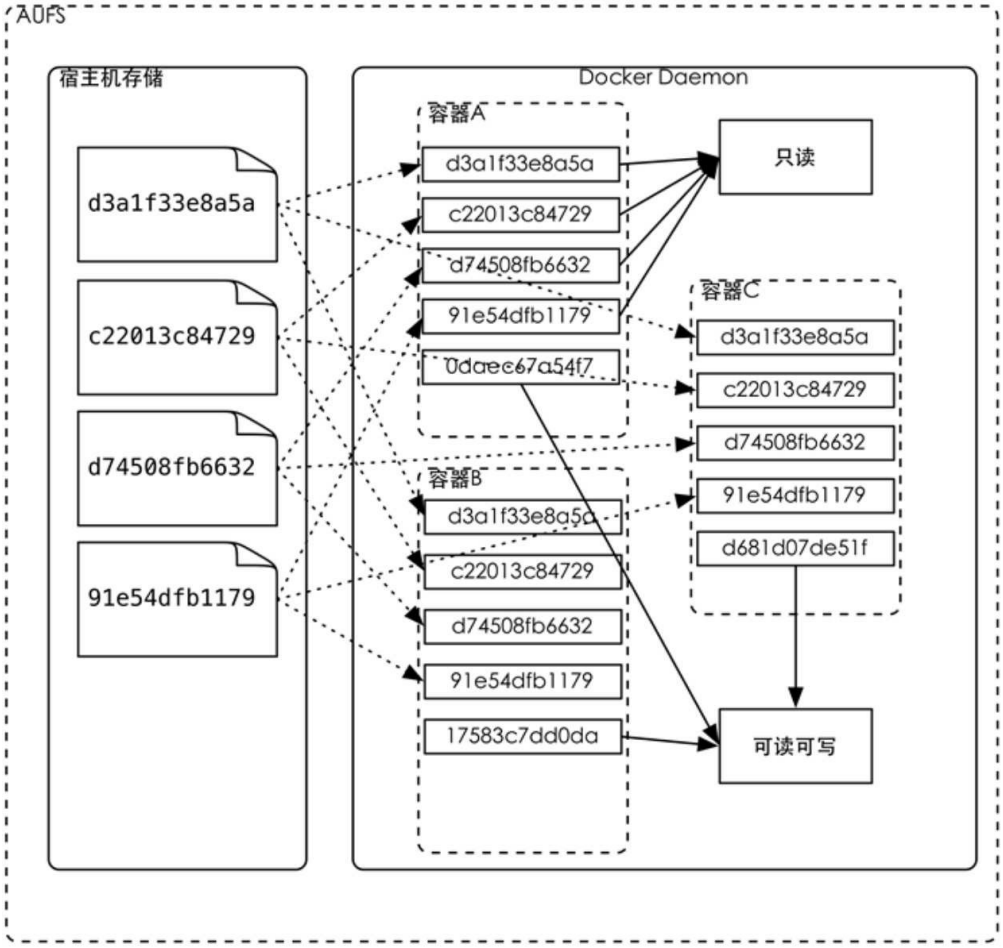


图11-8 AUFS挂载过程

Docker依次通过这种方式将所有的子文件层都加载到rootfs目录中，并且设定的目录权限全部是ro（只读）。这里加只读是为了防止数据冲突，因为除了每个容器最外的读写层是其独有的之外，所有的文件层都可能会被复用。如果不对其设定只读权限，那么一个容器改变了某个文件层的数据，这种改变势必会蔓延到其他容器之中，这样就会产生子文件层中的数据冲突。所以这里必须要设置为只读，一方面是为了隔离容器的文件系统，另外一方面也是为了维护容器稳定。

Docker最后会在当前容器的最后添加一个读写层，容器中对文件系统的写操作和删除操作都会被保存到这个目录中。此时容器中的rootfs目录才算完整，Docker剩下的工作就是运行此根文件系统中的Init脚本，而Init脚本则会调用相应的脚本完成操作系统的初始化工作，如同Linux中用户态初始化一样，如图11-9所示。

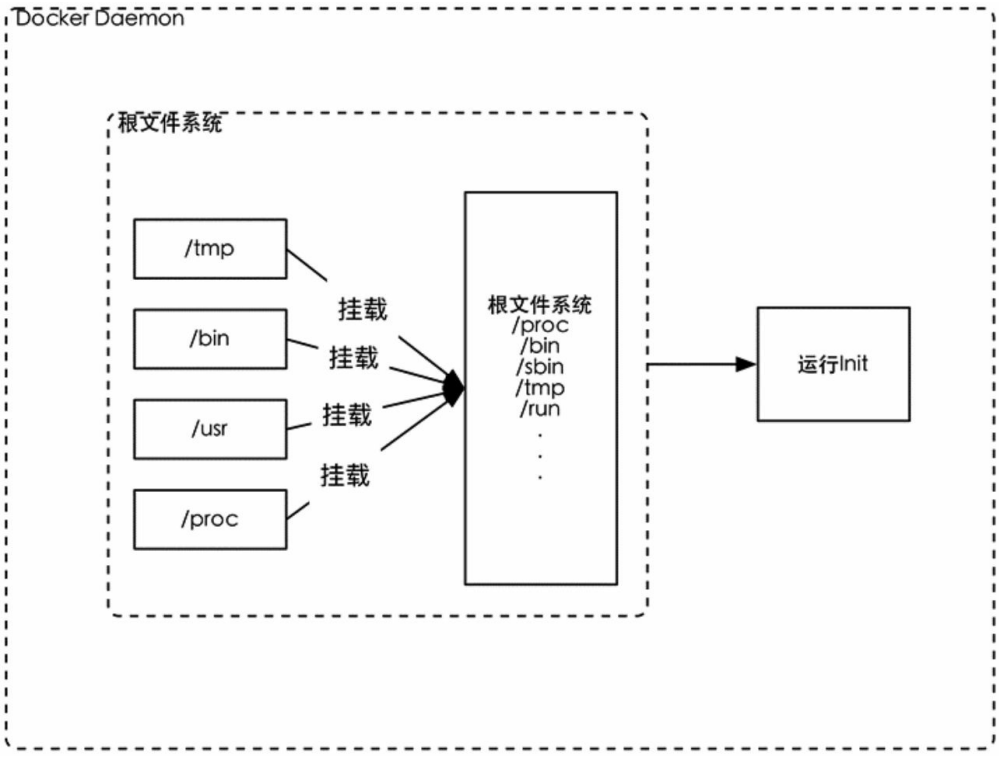


图11-9 Docker Daemon运行原理

当容器中的“操作系统”初始化完成之后，就会运行用户通过CMD所设定的脚本或者命令。

将Linux启动过程和容器启动过程对比，我们不难发现，Docker扮演了内核的角色，然后通过AUFS的联合挂载特性“组合”了一个rootfs，而后再挂载此rootfs。这与Linux启动过程中，bootlader加载initrd，再通过内核挂载根文件系统的处理方式几乎一致。唯一的区别就是Linux启动时，所使用的根文件系统是拿来就用，而容器中的根文件系统需要通过AUFS进行“组合”。

Docker除了AUFS外，还可以使用其他文件系统吗？当然可以，除了AUFS之外，还可以使用Devicemapper、Btrfs和Vfs三种文件系统。AUFS相对于这三种文件系统而言，实现起来非常简单，而且容易扩展，对宿主机存储机制几乎没有额外要求，但美中不足的是，AUFS迟迟未能加入到Linux内核当中，导致兼容性较差。除了Ubuntu之外，其他主流操作系统尚无官方支持计划。

AUFS还节省存储空间，而且AUFS实现机制可以最大程度地支持文件复用。这些是Docker选择AUFS的原因。在下一节，我们将会看到Docker是如何通过AUFS来管理和组织镜像的。

11.3 Docker的image管理

在11.2节当中，我们探讨了AUFS在Docker中的应用。Docker通过AUFS可以达到镜像共享文件层、共享数据的目的。但是Docker究竟是

如何通过AUFS来组织数据的呢？这个问题是我们本节需要继续深入探讨的内容。

为了回答这个问题，我们先从Docker存储目录入手。Docker默认将所有数据都存储在/var/lib/Docker中，但用户也可以自定义此路径，因此准确路径可以通过Docker info获取。

```
Docker info
Containers: 23
Images: 116
Storage Driver: aufs
 Root Dir: /var/lib/Docker/aufs
 Backing Filesystem: extfs
 Dirs: 162
 Dirperm1 Supported: true
Execution Driver: native-0.2
Logging Driver: json-file
EventsListeners: 0
Init SHA1:
Init Path: /usr/local/bin/Docker
Docker Root Dir:  /var/lib/Docker
Labels:
 provider=virtualbox
```

/var/lib/Docker目录中的文件结构如下：

```
drwxr-xr-x      5 root     root          4096 Oct  12 10:56 aufs
drwx------      2 root     root          4096 Oct  12 11:00 trust
-rw-------      1 root     root           468  Nov  17 05:38 repo
drwx------      6 root     root          4096 Nov  17 05:38 volum
drwx------      2 root     root          4096 Nov  18 01:17 init
drwx------      2 root     root          4096 Nov  18 01:17 tmp
-rw-r--r--      1 root     root         13312 Nov 18 01:49 linkgr
drwx------     25 root     root          4096 Nov  18 01:49 conta
drwx------    119 root     root         16384 Nov 18 02:03 graph
```

我们从repositories-aufs这个文件开始谈起。这个文件保存的是当前Docker环境所有具有Image Name的镜像信息，例如：

```
cat repositories-aufs
{
    "Repositories": {
        "192.168.73.12:5001/ubuntu": {
            "14.04": "91e54dfb11794fad694460162bf0cb0a4fa710cfa3f
        },
        "Docker-dev": {
            "1.7.1": "a7d1af33a73b84ad41d296939ff4396ff18bec18e96
            "latest": "907cda023cab17433a3725b6a1f744fd4fea5cae2b
        },
        "hello-world": {
            "latest": "975b84d108f1024953937167506c2935a081c8c55a
```

```
        },
        "ubuntu": {
            "latest": "91e54dfb11794fad694460162bf0cb0a4fa710cfa3
        }
    }
}
```

这个文件在Docker镜像管理和容器管理当中处于“线头”的位置，因为通过这个文件，Docker就可以获取到所有与镜像和容器有关的数据。这句话说的好像有点大，请耐住性子，我们继续往下分析。

通过repositories-aufs文件，我们（包括Docker Daemon）就可以知道每个镜像名称和与之对应的镜像ID。但镜像数据又保存在什么地方呢？我们以ubuntu:latest为例来找寻答案。

在graph目录中，我们找到了和ubuntu:latest的ID同名的目录（省略若干列）：

```
ls -ltr graph

drwx------  2 root    root   5b28e9246a2463cbe86a5f0eff13f70f187a
drwx------  2 root    root   91e54dfb11794fad694460162bf0cb0a4fa7
drwx------  2 root    root   c22013c8472965aa5b62559f2b540cd44071
drwx------  2 root    root   d3a1f33e8a5a513092f01bb7eb1c2abf4d71
```

在

91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d920813e267c目录中，有三个文件：

```
-rw-------    1 root     root             1    Oct 26 01:59 layer
-rw-------    1 root     root          1284 Oct 26 01:59 json
-rw-------    1 root     root            71   Oct 26 01:59 checks
```

layersize文件保存的是此文件层所占空间大小，而JSON文件则保存着当前文件层源数据信息：

```
cat json



{
    "id": "91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d9
    "parent": "d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309
    "created": "2015-08-20T20:21:15.767240511Z",
    "container": "74bb7db8d212f77ab6d467b710451e54d2c60533f641de8
    "container_config": {
        ...
    },
    "Docker_version": "1.7.1",
    "config": {
        ...
    },
```

```
    "architecture": "amd64",
    "os": "linux",
    "Size": 0
}
```

加黑的就是此文件层的父文件层。同样，我们在d74508fb6632491cea586a1fd7d748dfc5274-cd6fdfedee309ecdcbc2bf5cb82目录中也会看到checksum、JSON和layersize这三个文件。

其中layersize为1895，而JSON则为：

```
{
    "id": "d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdc
    "parent": "c22013c8472965aa5b62559f2b540cd440716ef149756e7b95
    "created": "2015-08-20T20:21:15.397405992Z",
    "container": "ce936b563c8c6bddfedd1923f308a125bc648b739cc5034
    "container_config": {
        ...
    },
    "Docker_version": "1.7.1",
    "config": {
        ...
    },
    "architecture": "amd64",
    "os": "linux",
    "Size": 1895
```

```
}
```

每个文件层中，JSON文件所保存的parent数据，和每个镜像所依赖的文件层顺序完全一致（如果读者是从第6章顺序看过来的，就会明白History命令也是通过JSON文件来获取数据的）。

```
Docker history ubuntu

IMAGE     CREATED   CREATED BY SIZE
91e54dfb1179   12 weeks ago /bin/sh -c #(nop) CMD ["/bin/bash"]
d74508fb6632    12 weeks ago /bin/sh -c sed -i 's/^#\s*\(deb.*uni
c22013c84729   12 weeks ago /bin/sh -c echo '#!/bin/sh' > /usr/sb
d3a1f33e8a5a    12 weeks ago /bin/sh -c #(nop) ADD file:5a3f9e9ab
```

通过JSON文件，我们（还有Docker Daemon）就会迭代获取到一个镜像所有的依赖文件层，然后我们就可以把目光转向aufs目录了。

在aufs目录中，所存储的就是所有镜像的真实数据，aufs目录中有三个子目录：

```
ls -ltr aufs

total 60
drwxr-xr-x  164  root     root         20480 Nov 18 02:03 mnt
drwxr-xr-x    2    root     root         20480 Nov 18 02:03 layer
```

```
drwxr-xr-x  164  root     root         20480 Nov 18 02:03 diff
```

而镜像数据就存储在diff目录中。diff目录中按照每个文件层ID保存着各自的数据，我们查看一下ubuntu镜像的数据：

```
ls d3a1f33e8a5a513092f01bb7eb1c2abf4d711e5105390a3fe1ae2248cfde13
bin    boot   dev    etc    home   lib    lib64  media  mnt    op
srv    sys    tmp    usr    var

ls c22013c8472965aa5b62559f2b540cd440716ef149756e7b958a1b2aba421e
etc   sbin  usr   var

ls d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb
etc

ls 91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d920813e26
```

这时，就可以看出ubuntu镜像所使用的四个文件层，每层存储的文件都互不相同。Docker通过JSON文件可以获取到一个镜像所使用的所有文件层，然后就在diff目录中判断所需的文件层是否存在，如果不存在则会下载此文件层。

我们在11.2节当中还提到了Docker通过AUFS联合挂载的特性，可以将不同目录挂载到同一个文件系统中，并将其作为容器的根文件系统而使用。但Docker具体是如何挂载的呢？也就是说，Docker是如何通过

AUFS来组织容器的呢？

我们现在已经明白一个容器是由镜像所有文件层和一个可写文件层所构成的。而这些又是如何被Docker所感知的呢？答案就是layers目录。在layers目录中，保存着一个容器所有依赖的文件层。

例如，我们通过Docker run ubuntu创建了一个ubuntu容器，容器ID为e0a6fd19c6f3。那么Docker在Create阶段，就会在layers目录中创建两个文件：

```
ls -ltr layers/|grep e0a6fd19c6f3
-rw-r--r--    1 root     root e0a6fd19c6f3f24503af0552be1bea26de8
-rw-r--r--    1 root     root e0a6fd19c6f3f24503af0552be1bea26de8
```

带有init的文件，就记录着创建此容器根文件系统所需要的文件层：

```
cat layers/e0a6fd19c6f3f24503af0552be1bea26de860058ec0c6a1770bc4a

91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d920813e267c
d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb82
c22013c8472965aa5b62559f2b540cd440716ef149756e7b958a1b2aba421e87
d3a1f33e8a5a513092f01bb7eb1c2abf4d711e5105390a3fe1ae2248cfde1391
```

而另外的文件，则记录着此容器所有的文件层：

```
cat layers/e0a6fd19c6f3f24503af0552be1bea26de860058ec0c6a1770bc4a

e0a6fd19c6f3f24503af0552be1bea26de860058ec0c6a1770bc4a5ada377f93
91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d920813e267c
d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb82
c22013c8472965aa5b62559f2b540cd440716ef149756e7b958a1b2aba421e87
d3a1f33e8a5a513092f01bb7eb1c2abf4d711e5105390a3fe1ae2248cfde1391
```

两个文件相比较，可以看出多出了一个以init结尾的文件层。这些文件层就形成了最终的根文件系统，并且保存在mnt目录之中：

```
ls mnt/1f6a90d63e1a2734d1492a03bc256cbe74eb1e9e0669974b2e266ea6c5
bin   boot  dev   etc   home  lib   lib64 media mnt   op
srv   sys   tmp   usr   var
```

Docker Daemon在Start阶段，就是把这个目录作为根文件系统挂载给容器，但那个可读可写的文件层在哪里呢？在diff目录之中！

```
ls -ltr diff/ |grep 1f6a90d63e1a2734d1492a03bc256cbe74eb1e9e06699

drwxr-xr-x    6 root     root 1f6a90d63e1a2734d1492a03bc256cbe74e
drwxr-xr-x    5 root     root 1f6a90d63e1a2734d1492a03bc256cbe74e
```

在容器当中，所有的变化都会体现在以容器ID命名的目录之中。

说了这么多，读者可能感觉关系特别乱，下面我们就通过图11-10来做一个总结。

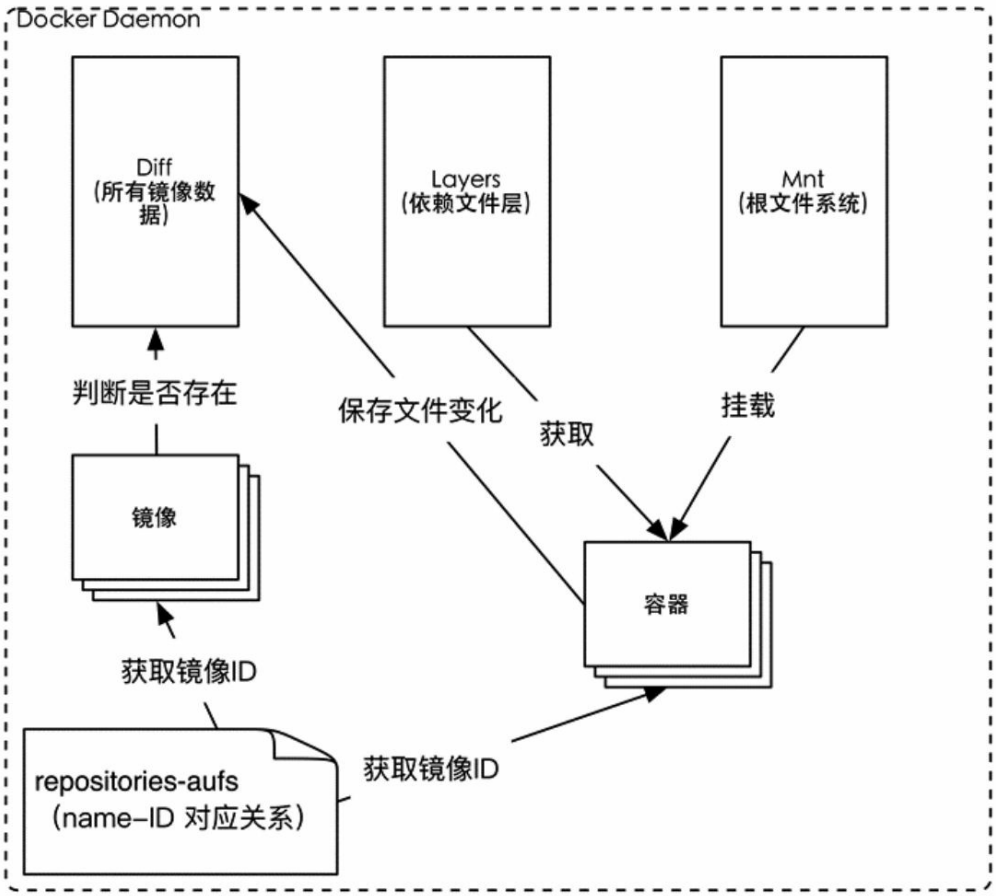


图11-10　Docker的image管理

Diff目录保存着所有文件层数据，而layers保存着所有容器的文件层依赖关系，并且随着容器销毁，里面的文件也会同步被销毁。而mnt目录则充当着容器的根文件系统，这个目录中的数据就是通过AUFS的特性而形成的，当容器处于stop或者exit状态时，此目录就会被卸载。

Docker Daemon在处理Create、Start、Diff等命令时，本质都是在处理这些目录中的数据。当Docker Daemon初始化时，会依次读取这些目

录中的数据，并且将这些数据都填充到Image结构体和Container结构体之中（Container结构体中的数据，有一部分来自于Container目录）。并且在每次发生Image和Container状态变化时，都会维护这些目录中的文件数据，并同步相应的结构体数据。

第12章　Docker资源调度

12.1　Docker如何管理资源

在前面章节，我们提到Docker底层是通过cgroups来管理资源的，但却没有深入讲解Docker是如何通过cgroups机制来分配和使用资源的。在本章，我们就来看看Docker是如何借助于cgroups机制来完成Docker环境中最为核心的资源管理。

我们首先了解一下cgroups机制。

cgroups最初叫Process Container，由Google工程师（Paul Menage和Rohit Seth）于2006年提出，后来更名为Control Groups。Control Groups，顾名思义就是把进程放到一个组里面进行统一控制。在Linux 2.6版本中，Control Groups被合并到Linux内核。

cgroups为资源管理提供了一个统一框架，可以把系统任务及其子任务整合（或分隔）到按资源等级而划分的不同任务组内，并且对这些任务组实施不同的资源分配方案。通俗来讲，cgroups可以限制、记录、隔离进程组所使用的物理资源（包括CPU、memory、I/O等）。基于这些原因，cgroups成为了Docker虚拟化的基础。

cgroups机制中有四个需要理解的概念。

（1）任务（task）。一个任务对应宿主机环境当中的一个进程。

（2）子系统（subsystem）。每一个子系统是对某一项具体物理资

源的控制器。例如，cpu子系统就是对CPU资源的控制，内存子系统就是对内存资源的控制。

（3）控制组（control group）。cgroups当中最基本的控制单元。一个group包含若干个任务（对应宿主环境的进程），并且此group也会包含若干子系统，用来控制group内的任务在指定子系统上面的资源使用。

（4）层级树（hierarchy）。cgroups的调度单位，由一个或多个group组成的树状结构。每个hierarchy通过绑定对应的子系统进行资源调度，同时子节点继承父节点的属性。整个系统可以有多个hierarchy。

如果觉得上面概念比较晦涩，那么可以通过图12-1来理解。

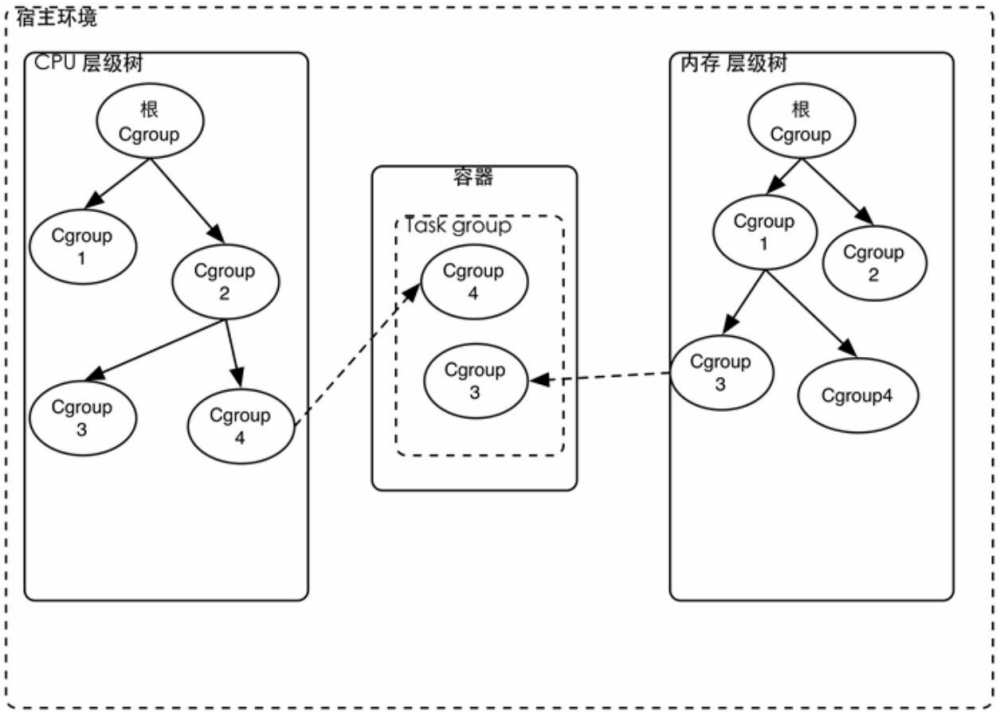


图12-1 cgroups的组成

目前cgroups共有10个子系统，具体如下：

- blkio：为块设备（比如磁盘，固态硬盘，USB等）设定I/O限制。
- cpuacct：为cgroup中任务生成CPU资源使用报告。
- cpuset：在多CPU系统中，为cgroup中的任务分配独立CPU和内存节点。
- devices：设置任务对物理设备的访问权限。
- freezer：挂起或者恢复cgroup中的任务。
- memory：设定cgroup中任务使用的内存限制，同时生成任务的内存资源使用报告。
- net_cls：使用等级识别符（classid）标记网络数据包，同时使用Linux流量控制程序（tc）识别从具体cgroup中生成的数据包。
- net_prio：对应用程序设置网络传输优先级，类似于socket选项的SO_PRIORITY。
- HugeTLB：HugeTLB页的资源控制功能。

通过查看/proc/cgroups，可以参看当前操作系统支持的子系统。当前Docker仅使用了上述子系统列表中的blkio、cpu、cpuacct、cpuset、device、freezer和memory这七种：

```
cat /proc/cgroups

#subsys_name    hierarchy     num_cgroups     enabled
cpuset          4             28              1
cpu             2             28              1
cpuacct         3             28              1
memory          8             28              1
devices         5             28              1
```

```
freezer         6               28              1
net_cls         9               28              1
blkio           1               28              1
perf_event      11              28              1
net_prio        10              28              1
hugetlb         7               28              1
```

用户可以通过Docker create命令或者Docker run命令来设置上述子系统的值。本质上来说，就是Docker Daemon将此容器所有的进程都添加到了一个group中，然后对这个group关联其他层级树。

从内核角度来看，cgroups就是内核在内核态进程中设置的钩子函数。当用户态任务运行过程中需要申请某项资源时，就会触发相应的子系统钩子函数进行检测，最终根据资源类别的不同，使用相对应的技术，进行资源限制和优先级分配。

但众所周知，Docker是基于Golang语言开发的，而Linux内核是通过C/C++语言开发的。虽然Golang可以调用C语言的API，但Docker是运行在用户态的应用，不应该直接调用内核态API接口，那么Docker又是如何调用cgroups来为其所用的呢？

其实这个问题不单单是Docker所遇到的，所有需要使用cgroups的应用都会遇到。为了方便用户态应用通过cgroups来达到资源调控的目的，cgroups通过VFS（Virtual Filesystem Switch，Linux虚拟文件系统转换器）的概念，实现了一套名为cgroup的文件系统。用文件系统中的目录、文件、创建、删除这些概念和操作，来映射对各个子系统的操作，并借此实现了cgroups中的层级概念。

Docker只需对cgroup文件系统中的各个文件进行修改，就可以达到对指定容器的资源管理，而操作的文件路径为/sys/fs/cgroup：

```
ls /sys/fs/cgroup/

blkio  cgmanager  cpu  cpuacct    cpuset    devices  freezer  hug
perf_event  systemd
```

例如，我们通过Docker run-it-m 512mg ubuntu bash创建了一个容器，如下所示：

```
Docker@default:~$ Docker run -it -m 512mb ubuntu bash
root@ca3be27a2284:/#
```

那么Docker Daemon在创建此容器时，就会在/sys/fs/cgroup/memory中创建与此容器相对应的group：

```
/sys/fs/cgroup/memory/Docker/ca3be27a22849c1ad78085d4da86bfcb490a
```

并在此目录中，创建一系列的初始化文件。每当此容器需要申请内存时，宿主机的内核进程就会根据此目录中的数值来决定是否需要分配资源。假如出现OOM的情况，内核就会根据memory.oom_control文件中的标志位进行相应处理。

在cgroups子系统中，每种子系统都提供多种策略供用户使用。下面我们就看一下Docker使用了七种子系统中的哪些策略。

1. Blkio：Docker使用两种策略：

- CFQ（完全公平队列策略）
- blkio.weight（权重比较）

默认是CFQ，即对所有容器一视同仁，不分彼此。如果使用权重比较的策略，内核会适当将I/O资源多分给权重大的容器。权重范围从100到1000。

2. CPU：对于CPU资源来说，也有两种策略：

- CFS（完全公平调度策略）
- RT（实时调度策略）

从CFS策略当中又衍生出按限额公平和按比例分配两种子策略，而RT则是按照周期给容器分配固定执行时间片。

Docker目前只使用CFS策略，通过cpu.cfs_period_us和cpu.cfs_quota_us设定周期和周期内最大可用时间，再通过cpu.shares设定容器使用的CPU权重。

3. CPUset：内核为容器指派运行代码的CPU，宿主机是多核机器为其生效前提。Docker目前使用cpuset.cpus和cpuset.mems来设置CPU编号和内存节点编号。

4. Device：cgroups可以对具体的物理设备设置黑名单或者白名单。Docker目前只使用白名单策略，设定语法为device_types:node_numbers

access_type。

- device_types有三种类型：b（块设备）、c（字符设备）、a（全部设备）
- access_type也有三种方式：r（读）、w（写）、m（创建）

5. Freezer：暂停或者恢复容器中的进程。只使用一个属性来表示进程的状态，例如，把state改为FROZEN，就可以暂停容器所有进程。Freezer status包括如下三种状态。

- FROZEN：暂停。
- FREEZING：正在暂停，这个是只读状态，只能由内核写入此值。
- THAWED：恢复。

Docker通过Pause和Unpause两个命令来修改此状态。

6. Memory：只有一种策略，即限额策略。Docker通过--memory="N"来强制设定最大使用量。

此外，cgroups中对内存还有软限制的子策略，当整体内存紧张的情况下，task获取的内存就被限制在软限制额度内，以保证不会有太多进程因内存挨饿。Docker通过--memory-reservation="N"来实现。

同时，Docker通过--memory-swap="N"来设定最大内存与swap区内存之和的用量限制。

对于内存告警（OOM），Docker通过--oom-kill-disable来解决。参数填0或1。0表示开启，当cgroup中的进程使用资源超过界限时立即杀死进程，1表示不启用（包含memory子系统的cgroup默认会启用）。当--oom-kill-disable不启用时，实际使用内存超过界限时，此进程会被暂

停，直到有空闲的内存资源。

通过本节，我们了解到Docker是如何通过cgroups完成资源管理的。但同大多数开源技术一样，Docker对于cgroups的支持还不是特别完善。例如，对于网络优先，就没有相应解决方案。但cgroups目前作为Linux内核的一部分，受到更多开发者的关注，这一点为Docker后续继续发展提供了技术保障。

12.2 Docker资源管理器

在生产环境中使用Docker，尤其是准备作为PaaS或者SaaS平台使用，那么安全是必不可少的考虑因素。

许多用户都认为Docker只是“沙盒”应用，Docker当中运行的应用并不会影响宿主机环境。这种直接将Docker类比为轻量级VM的观点是错误的。传统虚拟机中所有的操作都需要经过内核处理，但这时的内核仅仅指的是虚拟机内部运行的内核。即便此内核被黑客所攻破，那么也只有此虚拟机会受到影响。但Docker就不一样了，Docker当中所有的容器都运行在同一个宿主机内核当中，如果某个容器被黑客攻破，那么黑客就可以通过这个容器操作内核，继而控制所有的容器。

因此针对Docker来说，安全性就是两个问题：

（1）不对宿主环境产生影响。

（2）不对其他容器产生影响。

而这两个问题，我们也可以总结出，安全性问题就是资源隔离性问

题。在前面章节中，我们谈到Docker通过Namespaces来解决资源隔离问题。虽然通过Namespaces，Docker将大部分资源（网络、进程等）相互隔离，但仍有一些重要资源无法隔离，例如，共享内核，共享硬件设备，共享同一个root用户，共享/proc/sys等文件系统，共享SELinux、Time等资源。

虽然Linux在2006年开始吸收Mount Namespace作为其第一个Namespace，而后陆续将其余5个Namespace纳入内核当中。但内核仅仅是将Namespace作为一个增强功能来使用，Linux实际上并未因为吸收了Namespace，而将其所有内核资源完全按照Namespace来管理。

究其原因是考虑到，如果将内核所有资源全部按照Namespace机制来改造，那么Linux内核将变得非常复杂。所以目前仍没有被纳入Namespace管理的重要内核模块为：SELinux、CGroups、/proc中部分文件系统，以及/sys文件系统。这些“游离”于Namespaces之外的资源，就造成了Docker不安全的现状。

我们现在明白了Docker不安全的原因，既然无法从内核级别完善安全机制，那么只能依靠某些用户态的一些策略来提高Docker安全性。当前比较成功的安全策略有以下几种。

1. Capability策略

在Linux中，拥有最高权限的用户是root用户。而Docker的运行，恰恰又需要root用户。换言之，所有容器在宿主机环境当中对应的进程都是以root权限在运行，一旦容器被黑客攻陷，那么黑客就可以使用root权限操作内核。

Linux为了避免root用户权限被滥用的问题，从2.2内核开始，Linux

把原来和超级用户相关的高级权限划分成不同的单元，称为Capability，这样就可以独立对特定的Capability进行使用或禁止。

对于Docker来说，不合适的Capability列表有可能会导致容器中的应用崩溃、所以用户需要从功能性、安全性和稳定性三方面入手，综合判断才能给容器一个合适的Capability列表。但通常来说，Docker创建容器时所赋予的默认Capability列表，基本能满足容器的正常运行，如果需要额外增加其他Capability，用户一定要慎之又慎。

2. SELinux策略

SELinux作为Linux历史上最杰出的安全系统，被绝大多数Linux发行版本所采用。通过SELinux系统，管理员可以对文件、目录、端口诸如此类的资源达到彻底的控制，防止其他用户未经授权而进行修改。同时SELinux也对进程赋予最小的权限，每个进程都有各自的标签（TE），进程只能修改同标签的文件，无权修改其他标签文件，即便这些文件同属一个用户。而且SELinux也对用户权限有所限制，按照SELinux的策略，root用户在操作系统中也不能“无法无天”。

目前Docker还无法将SELinux应用于容器之中，但鉴于SELinux强大的安全控制力，建议用户在Docker所运行的宿主机环境当中，启用SELinux。这样即便容器被攻破，黑客也无法造成更大范围的破坏。

3. 缩小用户权限策略

通过SELinux，可以缩小用户权限。但此处的用户指的并非是宿主机环境中的用户，而是容器内部的用户。大多数容器为了方便，在容器内部都是以root用户来运行应用，因此有必要限制容器当中用户权限。除非十分必要，否则不建议使用root用户来启动应用。

4. 使用信任镜像策略

用户所使用的镜像，应该是用户自己通过Docker　build所产生的。如果需要使用第三方镜像，也尽量选择官方版本。来路不明的，或者没有听说过的镜像最好还是敬而远之。

同时，用户也要尽量通过Pull的方式从Docker仓库中下载，不要通过Load的方式加载第三方镜像。

5. CGroups策略

对于每一个容器，用户都应该通过CGroups策略对其使用的资源进行限制。虽然通过限制资源，并不能防止黑客攻陷容器。但容器安全并不仅限于黑客攻防，如果容器因为内存耗尽，或者I/O异常而导致其他容器无法正常运行，同样也属于不安全的范畴。

所以有必要对每一个容器都限制资源上限，按照“耐压舱”的原则，即便容器出现资源使用异常，也要将损害压缩在最小限度之内，不能蔓延到宿主机环境。

6. 最小文件系统策略

当容器中的应用需要挂载宿主机文件系统才能运行时，在挂载宿主机文件目录时，应该尽量剥离与应用无关的文件。同时没有特别需要，这些挂载的文件，应该只具有可读属性。而且除非有特别需要，同一个宿主机文件系统不应挂载到多个容器中，也就是容器之间不应通过共享宿主机文件系统的方式产生交叉影响。这样的机制避免了一个容器能够看到另一个容器的数据，而且容器也无法通过修改文件系统的内容来影响其他容器。

通过上述6条策略，可以在一定程度上保证容器和宿主机环境的安全。但无论多么牢固的安全策略，都抵不上用户的安全理念。所以需要用户时刻牢记，世界上没有绝对安全的事物，无论是虚拟机还是Docker，都有其各自的弱点，只有通过合适的，加固的Docker安全策略，辅助外部工具的监控和相关容错机制，才能保障Docker在生产环境中安全运行。

第四篇　Docker生态圈

本篇包括

- 第13章　Docker的云生态环境

第13章　Docker的云生态环境

13.1　Docker的开发语言

Docker受低层虚拟化技术限制，只能运行在Linux内核中，而且Docker运行需要频频调用与内核相关的API，因此也就决定了Docker只能使用C或者C++语系的开发语言。但众所都知，C的开发效率真是不高，幸好，我们还有Golang语言。

Golang语言同LXC、Namespaces一样，并不是新鲜事物。Golang最早是在2009年由Google所推出的一门开源语言。Golang兼顾了C/C++的执行高效性，也兼顾了Java、.Net等高级语言的开发效率，是一种拥有较高执行速度、编译速度和开发速度的系统编程语言。

Golang虽然相较于C/C++出现的时间要晚，但仍属于C语系家族。但其语法得到了很大的简化，使代码更简明、清楚，同时拥有动态语言的一些特点。同时Golang基于BSD完全开源，所以能免费地被任何人用于商业软件开发。而且Golang在并发方面做了相当大的改进，首次从语言层面就加入了对并发的支持，而并非像其他语言以库的形式进行支持。Golang虽然属于C语系家族，但却吸收了面向对象的思想。Golang没有类的概念，因而用户不用担心Golang的多态性，因为它仍然提供了接口机制。

说了这么多，Golang都应用在哪些方面呢？

Golang作为一门系统编程语言，首当其冲地适用于系统编程，其次

也适用于Webserver、分布式存储这样的分布式环境。当然，因为Golang对并发的语言级支持，使Golang特别适用于高并发的场合。但因为Golang的内存模型还不完善，因此对于实时性要求高的环境，暂时还不太适用。

上面说了这么多Golang的优点，那为什么Golang始终处于不温不火的局面呢？一方面是因为Golang的语法一直在完善演进当中，直到2014年才算稳定下来。另一方面是因为Golang语言还缺乏一个杀手级的应用。而2013年Docker的推出，扭转了这种无杀手级应用的局面。

因为Docker全部都是采用Golang语言编写，所以无论是对Docker的扩展，还是基于Docker做其他工具，势必都要掌握Golang语言。会Golang语言不一定会写Docker，但开发Docker的一定会Golang语言。

下面我们就简要看一下Golang语言的特点，注意，本节不会深入讲解Golang语法，如果需要深入了解Golang语言，请配合其他资料共同进行。

Golang程序是依靠包（package）来组织的，程序从main包开始执行。例如，Docker Daemon的程序就是从main包开始：

```
Docker/Docker.go

package main

import (
    "crypto/tls"
```

```
    "crypto/x509"
    "fmt"
    "io/ioutil"
    "os"
    "runtime"
    "strings"

    "github.com/Sirupsen/logrus"
    "github.com/Docker/Docker/api/client"
    "github.com/Docker/Docker/autogen/Dockerversion"
    "github.com/Docker/Docker/opts"
    flag "github.com/Docker/Docker/pkg/mflag"
    "github.com/Docker/Docker/pkg/reexec"
    "github.com/Docker/Docker/pkg/term"
    "github.com/Docker/Docker/utils"
)

const (
    defaultTrustKeyFile = "key.json"
    defaultCaFile       = "ca.pem"
    defaultKeyFile      = "key.pem"
    defaultCertFile     = "cert.pem"
)

func main() {
…..
}
```

import表示此程序需要依赖的包路径。在Golang当中，如果某一个包被导入，但却没有被使用，那么在编译阶段，编译器会报错。

在Golang当中，如果一个函数允许被其他包所调用，那么此函数首字母必须大写。否则，此函数只能被同包内函数所调用，例如：

```
func (s *State) WaitRunning(timeout time.Duration) (int, error) {
    s.Lock()
    if s.Running {
        pid := s.Pid
        s.Unlock()
        return pid, nil
    }
    waitChan := s.waitChan
    s.Unlock()
    if err := wait(waitChan, timeout); err != nil {
        return -1, err
    }
    return s.GetPid(), nil
}

func wait(waitChan <-chan struct{}, timeout time.Duration) error
    if timeout < 0 {
        <-waitChan
        return nil
```

```
    }
    select {
    case <-time.After(timeout):
        return fmt.Errorf("Timed out: %v", timeout)
    case <-waitChan:
        return nil
    }
}
```

WaitRunning可以被其他包所调用，而wait则只能在本包内使用。

在Golang中通过var来定义变量，并且可以在声明变量的同时进行初始化，例如：

```
var errNoDefaultRoute = errors.New("no default route was found")
```

Golang要求程序中不能出现“无用”资源，所以如果声明的变量没有被任何函数使用，那么在编译阶段，同样也会报错。

Golang的数据类型与C类语言的数据类型非常相像，共有以下几种：

- bool
- string
- int、int8、int16、int32、int64
- uint、uint8、uint16、uint32、uint64

- uintptr
- byte（等价于uint8）
- rune（等价于int32，用来表示一个unicode code point）
- float32 float64
- complex64 complex128

如果需要做类型转换，那么可以通过T(v)来完成，表示将v转换为类型T，例如：

```
var i int = 13
var f float64 = float64(i)
i := 55
f := float64(i)
```

在Golang当中，如果需要表示常量，可以通过const来声明，例如：

```
const (
    // Where we store the config file
    CONFIGFILE     = "config.json"
    OLD_CONFIGFILE = ".Dockercfg"

    // This constant is only used for really old config files whe
    // URL wasn't saved as part of the config file and it was jus
    // assumed to be this value.
    DEFAULT_INDEXSERVER = "https://index.Docker.io/v1/"
```

```
)
```

在Golang语法中，存在以下几种控制语句：for、if、switch、defer。用法如下：

For：

```
    for addr, ac := range configFile.AuthConfigs {
        ac.Username, ac.Password, err = DecodeAuth(ac.Auth)
        if err != nil {
            return &configFile, err
        }
        ac.Auth = ""
        ac.ServerAddress = addr
        configFile.AuthConfigs[addr] = ac
    }
```

使用for语句时不需要()，但需要{}（后面谈到的if、switch在此语法处理上也是如此）

If：

```
 if err := json.Unmarshal(b, &configFile.AuthConfigs); err !=
    arr := strings.Split(string(b), "\n")
    if len(arr) < 2 {
```

```
            return &configFile, fmt.Errorf("The Auth config file
        }
        authConfig := AuthConfig{}
        origAuth := strings.Split(arr[0], " = ")
        if len(origAuth) != 2 {
            return &configFile, fmt.Errorf("Invalid Auth config f
        }
        authConfig.Username, authConfig.Password, err = DecodeAut
        if err != nil {
            return &configFile, err
        }
        origEmail := strings.Split(arr[1], " = ")
        if len(origEmail) != 2 {
            return &configFile, fmt.Errorf("Invalid Auth config f
        }
        authConfig.Email = origEmail[1]
        authConfig.ServerAddress = DEFAULT_INDEXSERVER
        configFile.AuthConfigs[DEFAULT_INDEXSERVER] = authConfig
    } else {
        for k, authConfig := range configFile.AuthConfigs {
            authConfig.Username, authConfig.Password, err = Decod
            if err != nil {
                return &configFile, err
            }
            authConfig.Auth = ""
            authConfig.ServerAddress = k
            configFile.AuthConfigs[k] = authConfig
```

```
        }
    }
```

if语句可以在执行条件判断前带一个语句，此时语句中变量的生命周期会在if语句结束后结束。

Switch：

```
for _, change := range changes {
    var kind string
    switch change.Kind {
    case archive.ChangeModify:
        kind = "C"
    case archive.ChangeAdd:
        kind = "A"
    case archive.ChangeDelete:
        kind = "D"
    }
}
```

Golang中无须使用break语句来跳出switch，也可以省略判断条件，变为：switch{…}。

如果需要使用跳过某些条件的判断（类似于break的效果），则可以使用fallthrough，例如：

```
switch len(arr) {
    case 3:
        permissions = arr[2]
        fallthrough(等同于使用break)
    case 2:
        dst = arr[1]
        fallthrough
    case 1:
        src = arr[0]
    default:
        return DeviceMapping{}, fmt.Errorf("Invalid device specif
    }
```

Defer：

用于函数结束时执行，一般用来处理异常，释放资源。类似于try..catch..finally。

```
func (s *Server) postContainersAttach(version version.Version, w
vars map[string]string) error {
    …..
     defer closeStreams(inStream, outStream)


    …..
    attachWithLogsConfig := &daemon.ContainerAttachWithLogsConfig
```

```
        InStream:  inStream,
        OutStream: outStream,
        UseStdin:  boolValue(r, "stdin"),
        UseStdout: boolValue(r, "stdout"),
        UseStderr: boolValue(r, "stderr"),
        Logs:      boolValue(r, "logs"),
        Stream:    boolValue(r, "stream"),
        Multiplex: version.GreaterThanOrEqualTo("1.6"),
    }

    if err := s.daemon.ContainerAttachWithLogs(cont, attachWithLo
        fmt.Fprintf(outStream, "Error attaching: %s\n", err)
    }

    return nil
}
```

在Golang当中，虽然没有类的概念，但Golang提供了结构体（struct），通过结构体可以组织属性域和方法。初学者可以将结构体理解成类，例如：

```
type Server struct {
    daemon  *daemon.Daemon
    cfg     *ServerConfig
    router  *mux.Router
```

```
    start   chan struct{}
    servers []serverCloser
}
func (s *Server) postAuth(version version.Version, w http.Respons
map[string]string) error {
    var config *cliconfig.AuthConfig
    err := json.NewDecoder(r.Body).Decode(&config)
    r.Body.Close()
    if err != nil {
        return err
    }
    status, err := s.daemon.RegistryService.Auth(config)
    if err != nil {
        return err
    }
    return writeJSON(w, http.StatusOK, &types.AuthResponse{
        Status: status,
    })
}
```

postAuth就是Server的方法。可以在作用范围内，通过Server.postAuth来调用。

```
type HttpServer struct {
    srv *http.Server
    l   net.Listener
}

func (s *HttpServer) Serve() error {
```

```
    return s.srv.Serve(s.l)
}
server是HttpServer的方法，在其他包中，就可以通过HttpServer.Server来调用
```

Golang当中有一个数据类型会被经常用到：数组和切片。

数组在Golang当中是一个类型，[n]T表示一个长度为n的数组，其中元素类型全为T。数组一旦声明之后，长度就无法改变。

而切片，在Golang当中是一个数据结构，指向的是一个数组从某一位开始连续的部分，用[]T来表示。切片和数组看起来非常相像，但使用方式却完全不同。

以上是Golang基本语法的简要介绍，仅通过一节来完整介绍Golang语法是不现实的，而且Golang语法也不是本书的讲解重点，所以还是建议读者另外查找Golang语法相关资料，来加深对Golang语言的理解。

13.2 支持Docker的开源组件

Docker不是万能的，实际的生产环境非常复杂，仅凭Docker，是无法从容应对复杂场景的。因此如果需要将Docker应用到生产环境中，那么就必须配合其他开源工具一起使用。以下就是目前证明可以增强Docker功能的开源工具。

1. Swarm

Swarm是Docker官方提供的工具，理所当然是推荐首选。Swarm通

过调用Docker提供的Rest API来管理Docker集群。

Swarm由管理器和代理两部分组成，管理器运行在Master节点，而其他节点运行Swarm代理。由此将一群Docker宿主机变成一个单一的、虚拟的主机。因为Swarm是通过标准的Docker Rest API作为访问接口。因此理论上，各种Docker客户端都可以与Swarm进行通信。

当Swarm运行时，Swarm管理器会启动一个Scheduler（调度器）＋router（路由器），并用来监听客户端请求。当客户端需要创建一个新容器时，Swarm管理器就会通过Swarm代理调度合适的节点来创建容器。因此即便某些节点因为某些原因意外关闭，那么Swarm管理器也会及时感知到，并且尝试重新恢复此容器。

目前来说，Swarm适合Docker集群的初级管理。因为Swarm在网络层面仍然非常薄弱（此部分仍然依靠Docker Engine来实现），同时也不支持容器跨节点Link，而且Swarm在资源管理方面还仅处于起步阶段，相较于后面将要介绍的Kubernetes和Mesos而言，Swarm适合作为小规模集群管理。但不可否认的是，Swarm作为Docker官方提供的集群管理工具，可以快速支持Docker的最新功能，而且Swarm版本迭代速度特别快，几乎每两个月就会发布一个新版本，因此建议用户持续关注此项目。

2. Kubernetes

作为Google推出一个Docker容器管理工具，一经推出便引起了Docker社区的关注。

在这款开源工具中，Kubernetes引入了pod（容器仓）的概念，容器仓表示一个被当作单一逻辑服务来部署的容器。容器仓非常契合日趋流

行的“一个容器中运行一个服务”的部署思想。Kubernetes为保证较高的资源利用率，一般会在单一机器或者集群中运行多个容器仓。同时，Kubernetes也会持续监控容器仓以确保它们可以正常运行。

除了pod，Kubernetes还将真实应用服务抽象成Services，在每一个Services背后有很多对应的pod支撑。每一个Service对外表现为一个单一访问接口，外部不需要了解后端如何运行，这给扩展或维护后端带来很大的好处。

与Swarm通过Docker API来创建容器不同，Kubernetes通过Replication controllers来实例化pods。Replication controllers有若干策略，如Rescheduling、Scaling等。这些策略可以确保当前Kubernetes环境中，有足够数量的pod副本在运行。如果pods副本数量不足，Kubernetes会动态创建。如果数量太多，Kubernetes也会有选择性的杀死一部分。通过Replication controllers机制，Kubernetes集群总是运行在用户期望的状态。

而用来区分Pod、Service、Replication Controller的就是Labels。Labels是一个Key/Value键值对，Pod、Service、Replication Controller可以有多个label，但是每个label的key只能对应一个value。Kubernetes通过Label定位到目标容器，然后将访问Service的请求转发给目标容器。

现在各大厂商都宣布支持或者即将支持Kubernetes，所以在Docker开源社区当中，采用Kubernetes来管理Docker集群的方案越来越多。与此同时，在Kubernetes新版本中，Kubernetes也支持跨计算机集群部署容器。Kubernetes可以在不需要开启网络端口或执行其他操作的情况下，让容器彼此通信。

所以在集群规模不超过1万个节点的情况下，推荐使用Kubernetes来管理集群。而在集群规模达到万级规模的情况下，就可以采取Mesos的方案。

3. Mesos

Apache Mesos作为一个开源的集群管理工具，是为大规模集群而设计的。Mesos始终围绕高可用性和弹性而进行设计，所以在一个由Mesos管理的集群当中，允许混合运行Hadoop、Spark和Docker任务。

在一个典型的Mesos集群当中，一定会有Mesos-master、Mesos-slave、Frameworks和Zookeeper这四部分。Master负责收集集群资源，而Slave则负责管理本节点上面的任务。Frameworks是计算框架，通过MesosSchedulerDiver接入Mesos。Zookeeper用于选择和查找当前Master地址。在实际部署案例当中，为了容错，Zookeeper会管理Mesos Master节点以及备用Master节点（通常会有2个或4个备用Master节点时刻准备接替故障的Master节点）。

Master会维护一个现有资源列表，并且基于分配策略来决定提供多少资源。然后Master将此列表“提供”给Frameworks。Frameworks则由两部分组成：注册到Mesos中的调度器（scheduler）和运行在slave节点的任务执行进程（executor）。通过这两部分，Frameworks可以决定接受多少资源和决定哪个任务使用多少资源。

Mesos集群可以运行多种Frameworks，以适应不同种类的任务。Mesos的这种以资源为主的体系结构，大大提高了集群资源的利用率。在大规模集群（包含成千上万台服务器）中自动化运行和管理Docker容器时，Mesos会是一个非常合适的方案。

针对上述三种工具而言，Swarm使用Docker标准API来管理Docker集群，优点是简单方便，但在复杂应用场景下，Swarm的调度变得会有些困难。而Kubernetes对集群提供了容错、调度和扩展机制，可以用来管理一个中等规模的集群。但当集群规模很大时，Mesos作为一种底层级、并且久经沙场的调度器，而且可以同时支持包括Marathon、Kubernetes和Swarm在内的多种Frameworks，就几乎是不二选择了。

13.3 CoreOS、Vagrant和Amazon如何支持Docker

1. CoreOS

CoreOS是一个基于Docker的轻量级Linux发行版，通过轻量的系统架构和灵活的应用程序部署能力，可以简化数据中心的维护成本和复杂度。同其他Linux发行版相比，CoreOS有如下特点。

首先，CoreOS不采用安装软件包的方式，而是通过Docker的虚拟环境来提供应用服务。其次，借助于Docker，CoreOS内部所有应用程序可以共享系统内核和资源，但是彼此之间又相互隔离、不可见。这种方式降低了CoreOS内部应用程序与内核，运行环境之间的耦合度。最后，CoreOS只需满足Docker运行要求即可，因此CoreOS可以做的非常“瘦”。

相较于传统的部署方式而言，在CoreOS集群中部署应用程序更加灵活便捷，应用程序之间的干扰会更少，而且操作系统自身的维护也变得更加容易。

CoreOS作为一款支持云计算的操作系统，在设计时就考虑到了支持在线升级的情况，所以在CoreOS内部有主动、从动双分区。主动分区负责系统运行，从动分区负责系统升级。当CoreOS发布新版本之后，CoreOS会自动下载新版本到从动分区，同时通过cgroups对下载使用的网络和磁盘资源进行限制，借此降低自动升级对应用的干扰。当系统重启时，原来的从动分区将切换为主动分区，而之前的主动分区则被切换为从动分区，两个分区扮演的角色将相互对调，完成升级。

除此之外，CoreOS还支持集群管理。CoreOS通过fleet来进行集群管理，通过etcd来完成服务发现。

在每台CoreOS节点中，都运行着一个systemd服务和fleet代理。CoreOS通过systemd来组织和管理服务。而fleet则通过D-Bus API与systemd来通信。通过节点中的fleet代理，fleet可以查看集群中服务器的状态、启动或终止Docker container、读取日志内容等。当某个fleet代理从集群中脱离之后，其节点中的服务会被转移到其他符合条件的节点中。

Etcd则是一个分布式key/value存储服务，CoreOS集群中的程序和服务可以通过Etcd共享信息和服务发现。Etcd通过选举形式在服务器中选举Lead来同步数据，并以此确保集群内信息始终一致和可用。

2. Vagrant

Vagrant是一款基于Ruby的开源工具，通过Vagrant可以创建和部署虚拟化环境。Vagrant支持多种虚拟化方案，例如Virtualbox、VMware和Docker。

Vagrant以往更多的被作为一个工具用于管理虚拟机，例如，通过

Vagrant可以管理各种Linux机器。但现在Vagrant也开始支持Docker，支持的方式则是通过Provision。

Provision是Vagrant启动时可选的一种行为，通过Provision，Vagrant可以在启动虚拟机之前做一些初始化工作，例如，下载某些软件安装包和在虚拟机环境中执行特定shell脚本等。通过Provision，Vagrant可以自动安装Docker，创建容器和管理容器。

Vagrant可以通过执行特定的Vagrantfile，来创建一个标准的Docker运行环境，因此Vagrant在实际使用过程当中，更侧重于创建一个团队中统一的测试和开发环境。真正将Vagrant用于生产环境的，并不常见。

3. Amazon

Amazon所提供的AWS　EC2是一项高度可扩展的高性能容器管理服务。在其还未支持Docker之前，如果需要在EC2当中部署Docker，则用户需要自行在EC2中创建一个Linux实例，然后自行安装配置Docker服务。但现在这一切，AWS都可以自动完成了。

用户现在可在EC2实例集群上轻松地发布、管理和扩展Docker容器，还可以在几秒钟内启动、停止和管理这些容器。对于用户来说，这样使用Docker就更加方便了。用户只需构建Docker镜像，然后在AWS中运行这些镜像即可。

AWS不只支持Docker容器，还提供了一组强大的API来管理容器。通过这些API，用户可以自由发布容器集群并指定希望运行的任务，而EC2就会按照用户的意图执行后续的工作。

AWS会根据集群负载，用户所定制的隔离策略来组织和管理容器。将容器优先放置到资源最充足的EC2实例当中运行，从而让用户应用始终处于最优状态。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

第五篇　Docker案例

本篇包括

- 第14章　基于Amazon的Docker部署案例
- 第15章　基于CoreOS的Docker部署案例

电子书寻找看手相 钉钉或微信pythontesting

第14章　基于Amazon的Docker部署案例

在AWS中部署Docker应用，首先需要创建一个AWS账户。在本书编写期间，每一位新用户有一年的免费试用资源。

假设读者现在已经创建了AWS账户，并且登录成功，那么在服务控制台就可以看到EC2 Container Service，同时还可以看到EC2。EC2对应的是一台实例机器，这台机器除了可以部署Docker，还可以安装其他软件。所以对于用户来说，EC2相当于一台裸机。而ECS（EC2 Container Service）则是AWS专门用来部署Docker应用的。用户在选择时，不要选错，如图14-1所示。



图14-1　选择EC2 Container Service

单击进入之后，可以看到创建一个ECS服务共需要三步：创建Docker信息，配置服务和配置集群。

在第一步创建Docker信息中，我们需要输入容器名称、容器所依赖的镜像信息、此容器所需的内存数量和端口映射信息，如图14-2所示。

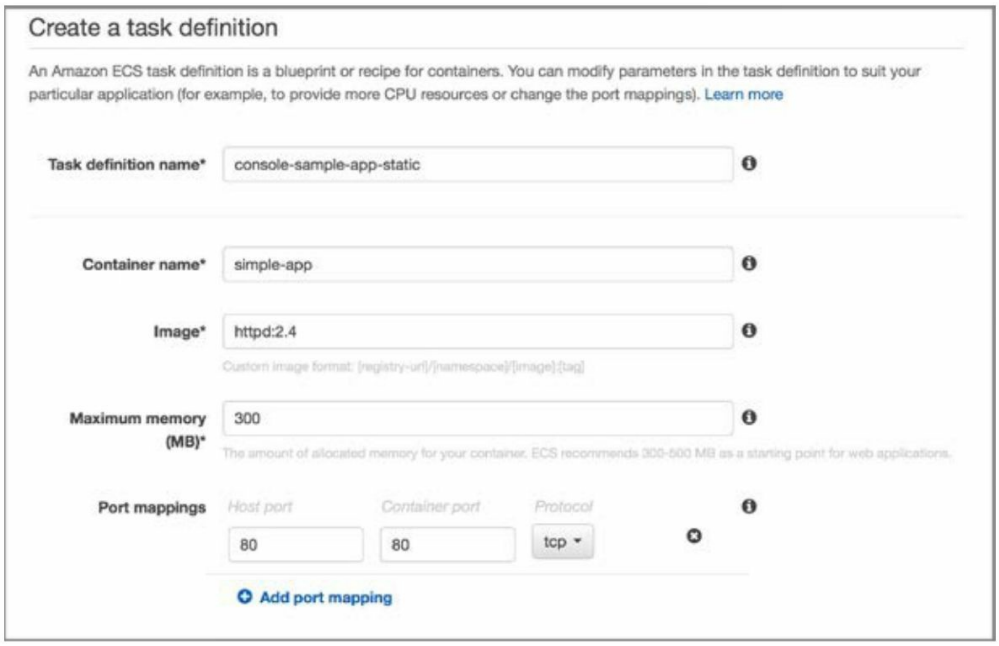


图14-2 创建Docker

第二步，配置服务信息。在这里，AWS将第一步所定义的任务打包为一项服务。所以我们需要为此项服务给定一个名称，同时在这里设定集群当中需要运行的容器数量。如果有必要，还可以为此项服务配置一个ELB（默认是不需要），如图14-3所示。

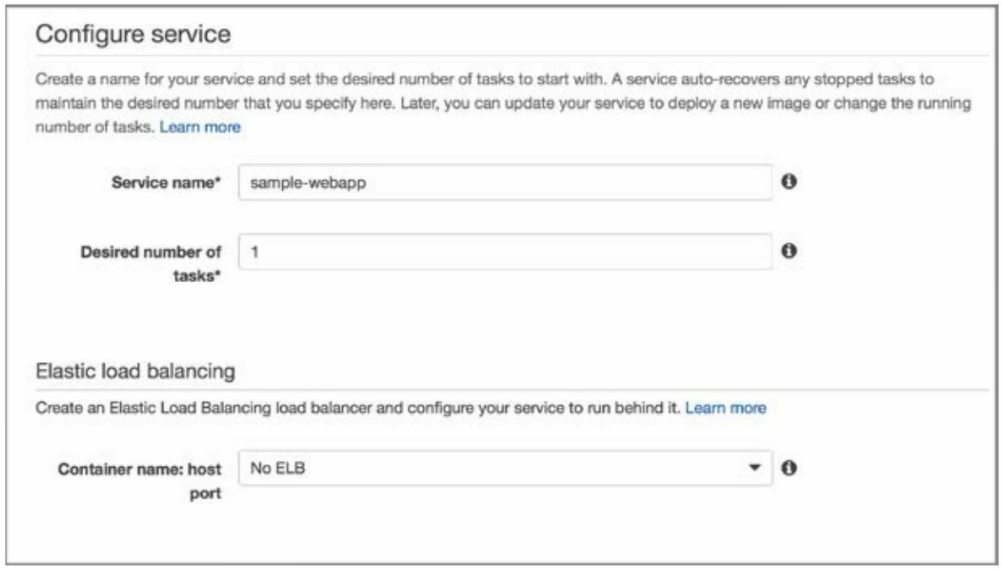


图14-3 配置服务信息

第三步，配置集群。这里需要配置容器运行的机器资源，从t2到m4各种资源都可以选择，费用也是依次增长。同时还可以配置是否开启SSH，以及外部可以访问的IP段，如图14-4所示。

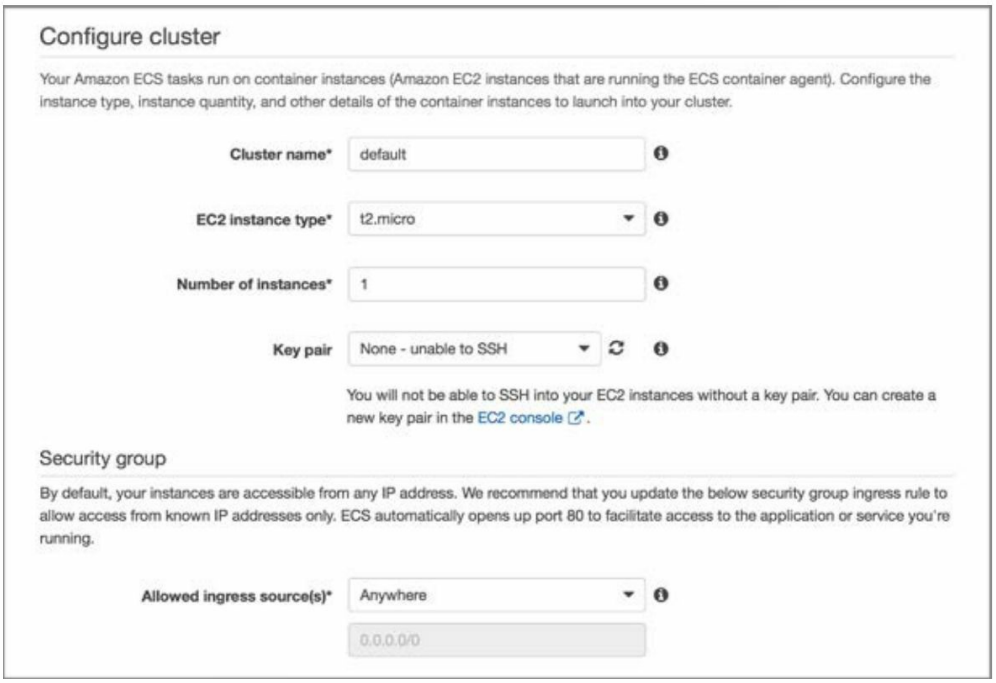


图14-4　配置集群

这三步都做完之后，AWS会呈现出一个信息配置清单，如果清单上面的信息核对无误之后，单击Launch instance，AWS就会开始创建服务，并且启动容器了。

目前来说，AWS提供的服务是最全面的，同时功能也是最强大的。但受限于国内访问速度和费用问题，国内使用规模一直不大。但作为目前支持Docker最好的PaaS平台，我们还是有必要接触一下。

书籍下载qq群6089740 钉钉群21734177 IT书籍 http://t.cn/RDIAj5D

第15章　基于CoreOS的Docker部署案例

CoreOS是一个使用Linux内核的轻量级操作系统。CoreOS将精力集中于云计算平台的基础设施，专注于轻量级、松耦合、高可用、集群化。CoreOS与其他操作系统不同的是，CoreOS当中的所有应用均通过Docker来部署。

我们来看一下如何在CoreOS当中部署Docker应用。本次选取的应用为Subgun，这是CoreOS官方提供的一个用于从mailgun（国外一家邮件服务提供商）订阅主题的应用。下面，我们就来看看如何通过CoreOS构建一个subgun集群。

运行Subgun需要两部分：Subgun应用程序（Subgun APP）和Subgun的守护进程（Subgun Daemon）。Subgun APP用来处理来自于http的请求即而Subgun Daemon用来更新处理之后的信息。具体工作细节，我们不需要关注，只需集中精力于如何部署即可。

我们将采取三节点的集群架构，并在集群之外使用AWS所提供的ELB作为负载均衡。完成之后的集群示意图如图15-1所示。

电子书寻找看手相 钉钉或微信pythontesting

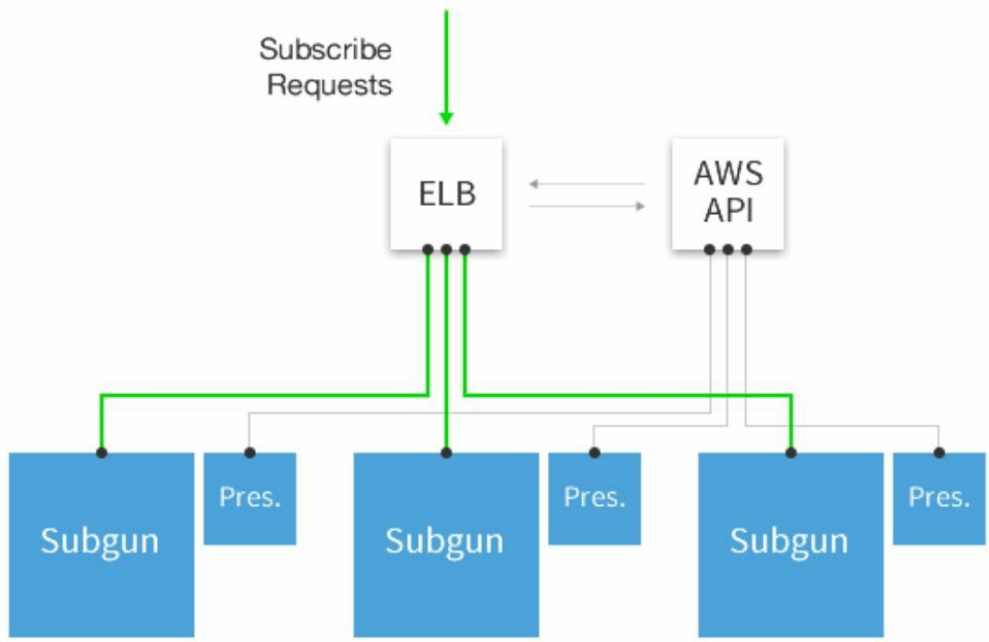


图15-1　集群示意图

因为CoreOS当中所有的应用都运行在Docker中，所以需要Subgun APP和Subgun Daemon的镜像。CoreOS官方已经向Docker hub提供了这两个应用的镜像，如果读者需要自己尝试，或者做一些定制化操作，可以参考下面的Dockerfile自行构建。

```
CoreOS/subgun

FROM stackbrew/ubuntu:precise
RUN apt-get install -y --allow-unauthenticated ca-certificates

ADD bin/subgun /bin/

EXPOSE 8080
ENV SUBGUN_LISTEN 127.0.0.1:8080
```

```
CMD /bin/subgun

quay.io/CoreOS/elb-presence

FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y python-boto

ADD elb-presence /bin/elb-presence

CMD /bin/elb-presence
```

除了Subgun镜像之外，我们还需要另外一些文件：Service文件。因为CoreOS是通过fleet来管理各节点服务的，所以我们需要为fleet准备一些Service文件。在Service文件当中，容器名称、服务名称这些是不能重复的，但因为我们这次是三节点部署，所以如果同时写三个不同的Service文件，那么就失去了CoreOS支持集群化的特点。所以，我们使用CoreOS提供的template语法来组织Service文件内容，最终的文件内容如下：

```
subgun-http@.service

[Unit]
Description=subgun
```

```
[Service]
ExecStartPre=-/usr/bin/Docker kill subgun-%i
ExecStartPre=-/usr/bin/Docker rm subgun-%i
ExecStart=/usr/bin/Docker  run  --rm  --name  subgun-%i  -e  SUBG
SUBGUN_LISTS=recv@sandbox2398.mailgun.org -e SUBGUN_API_KEY=key-7
-p 8080:8080 CoreOS/subgun
ExecStop=/usr/bin/Docker stop subgun-%i

[X-Fleet]
Conflicts=subgun-http@*.service

subgun-presence@.service

[Unit]
Description=subgun presence service
BindsTo=subgun-http@%i.service

[Service]
ExecStartPre=-/usr/bin/Docker kill subgun-presence-%i
ExecStartPre=-/usr/bin/Docker rm subgun-presence-%i
ExecStart=/usr/bin/Docker run --rm --name subgun-presence-%i -e A
-e AWS_SECRET_KEY=qxB5k7GhwZNweuRleclFGcvsqGnjVvObW5ZMKb2V -e AWS
bcwaldon-fleet-lb quay.io/CoreOS/elb-presence
ExecStop=/usr/bin/Docker stop subgun-presence-%i

[X-Fleet]
```

```
MachineOf=subgun-http@%i.service
```

在编写Service文件时，需要注意不能使用Docker run-d。因为使用-d之后，fleet将无法管理此容器，导致此容器运行后就会直接退出。

将编写好的Service文件放到任意一台CoreOS机器中，然后通过下面的命令将Service文件提交到CoreOS集群中。

```
$ fleetctl submit subgun-presence@.service subgun-http@.service
```

提交完成之后，可以通过下面的命令来验证是否提交成功。

```
$ fleetctl list-unit-files
UNIT                            HASH         DSTATE      STATE
subgun-http@.service            08be408     inactive    inactive
subgun-presence@.service        5180ed9     inactive    inactive
```

当看到Service状态为inactive时，就可以调用fleet来执行这些Service了：

```
$ fleetctl start subgun-presence@{1..3}.service subgun-http@{1..3
Unit subgun-http@1.service launched on 0e0a1f59.../172.17.8.102
Unit subgun-http@2.service launched on 30a73182.../172.17.8.103
Unit subgun-presence@1.service launched on 0e0a1f59.../172.17.8.1
```

```
Unit subgun-presence@2.service launched on 30a73182.../172.17.8.1
Unit subgun-http@3.service launched on 610a163d.../172.17.8.101
Unit subgun-presence@3.service launched on 610a163d.../172.17.8.1
```

最后，我们来验证一下Service是否都启动成功：

```
$ fleetctl list-units
UNIT                            MACHINE                     AC
subgun-http@1.service       0e0a1f59.../172.17.8.102       activ
subgun-http@2.service       30a73182.../172.17.8.103       activ
subgun-http@3.service       610a163d.../172.17.8.101       activ
subgun-presence@1.service   0e0a1f59.../172.17.8.102       activ
subgun-presence@2.service   30a73182.../172.17.8.103       activ
subgun-presence@3.service   610a163d.../172.17.8.101       activ
```

此时，可以看到所有Service都处于running状态了。上述命令都是通过人工来调用的，我们还可以把这些命令集成到vagrantfile中，通过vagrant来自动化创建CoreOS集群。