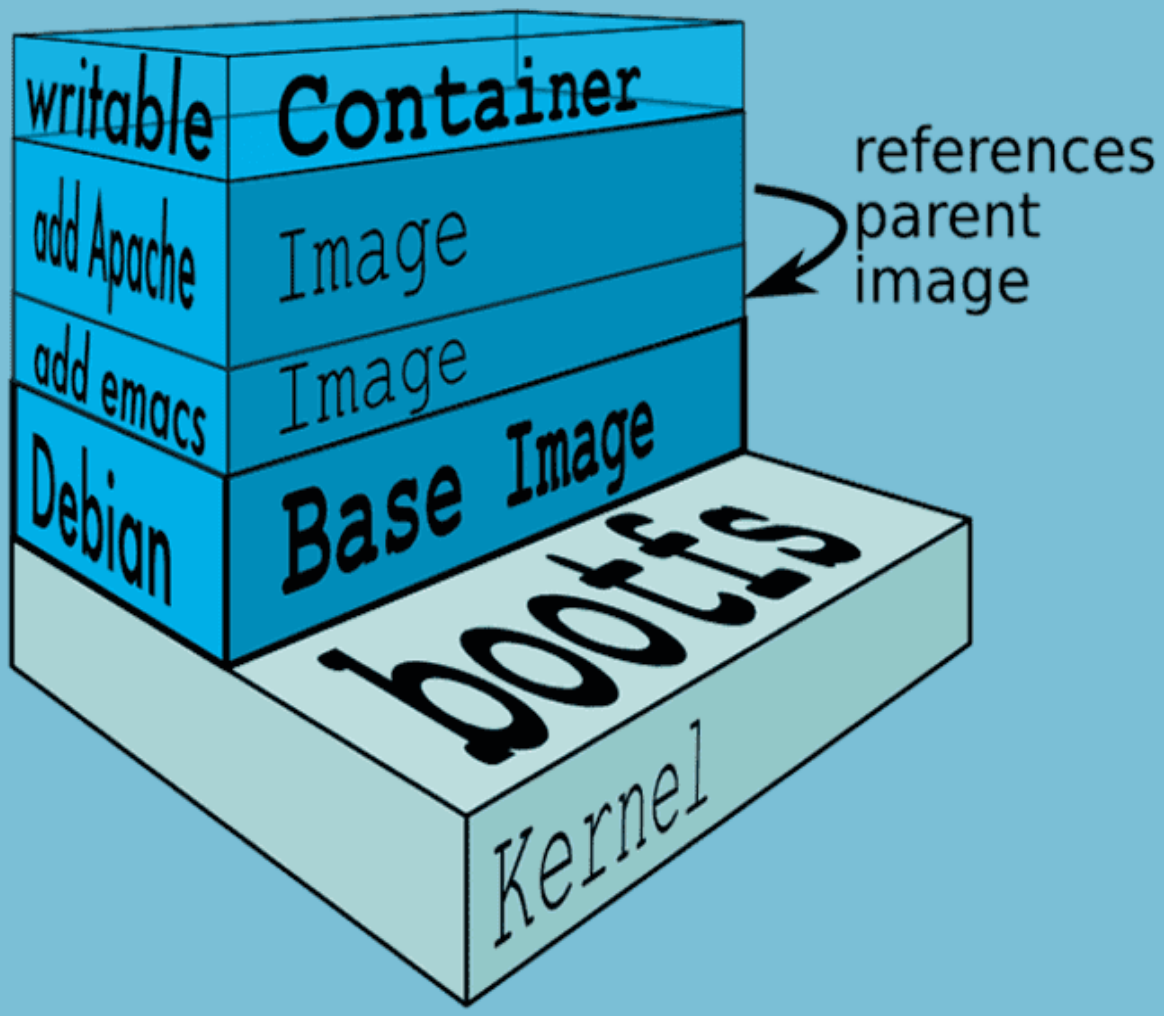

writable
Container
add Apache
Image
add emacs
Image
Debian
Base Image
bootfs
Kernel
references
parent
image

Docker 源码分析

（一）：Docker架构

> 原文出处：infoq中文站
> 作者：孙宏亮

在《深入浅出Docker》系列文章的基础上，InfoQ推出了《Docker源码分析》系列文章。《深入浅出Docker》系列文章更多的是从使用角度出发，帮助读者了解Docker的来龙去脉，而《Docker源码分析》系列文章通过分析解读Docker源码，来让读者了解Docker的内部实现，以更好的使用Docker。总之，我们的目标是促进Docker在国内的发展以及传播。另外，欢迎加入InfoQ Docker技术交流群，QQ群号：272489193。

- 1 背景
  - 1.1 Docker简介
  - 1.2 Docker版本信息
- 2 Docker架构分析内容安排
- 3 Docker总架构图
- 4 Docker架构内各模块的功能与实现分析
  - 4.1 Docker Client
  - 4.2 Docker Daemon
  - 4.3 Docker Registry
  - 4.4 Graph
  - 4.5 Driver
  - 4.6 libcontainer
  - 4.7 Docker container
- 5 Docker运行案例分析
  - 5.1 docker pull
  - 5.2 docker run
- 6 总结
- 7 作者简介
- 8 参考文献

1 背景

1.1 Docker简介

Docker是Docker公司开源的一个基于轻量级虚拟化技术的容器引擎项目,整个项目基于Go语言开发，并遵从Apache 2.0协议。目前，Docker可以在容器内部快速自动化部署应用，并可以通过内核虚拟化技术

（namespaces及cgroups等）来提供容器的资源隔离与安全保障等。由于Docker通过操作系统层的虚拟化实现隔离，所以Docker容器在运行时，不需要类似虚拟机（VM）额外的操作系统开销，提高资源利用率，并且提升诸如IO等方面的性能。

由于众多新颖的特性以及项目本身的开放性，Docker在不到两年的时间里迅速获得诸多厂商的青睐，其中更是包括Google、Microsoft、VMware等业界行业领导者。Google在今年六月份推出了Kubernetes，提供Docker容器的调度服务，而今年8月Microsoft宣布Azure上支持Kubernetes，随后传统虚拟化巨头VMware宣布与Docker强强合作。今年9月中旬，Docker更是获得4000万美元的C轮融资，以推动分布式应用方面的发展。

从目前的形势来看，Docker的前景一片大好。本系列文章从源码的角度出发，详细介绍Docker的架构、Docker的运行以及Docker的卓越特性。本文是Docker源码分析系列的第一篇——Docker架构篇。

1.2 Docker版本信息

本文关于Docker架构的分析都是基于Docker的源码与Docker相应版本的运行结果，其中Docker为最新的1.2版本。

2 Docker架构分析内容安排

本文的目的是：在理解Docker源代码的基础上，分析Docker架构。分析过程中主要按照以下三个步骤进行：

- Docker的总架构图展示
- Docker架构图内部各模块功能与实现分析
- 以Docker命令的执行为例，进行Docker运行流程阐述

3 Docker总架构图

学习Docker的源码并不是一个枯燥的过程，反而可以从中理解Docker架构的设计原理。Docker对使用者来讲是一个C/S模式的架构，而Docker的后端是一个非常松耦合的架构，模块各司其职，并有机组合，支撑Docker的运行。

在此，先附上Docker总架构，如图3.1。

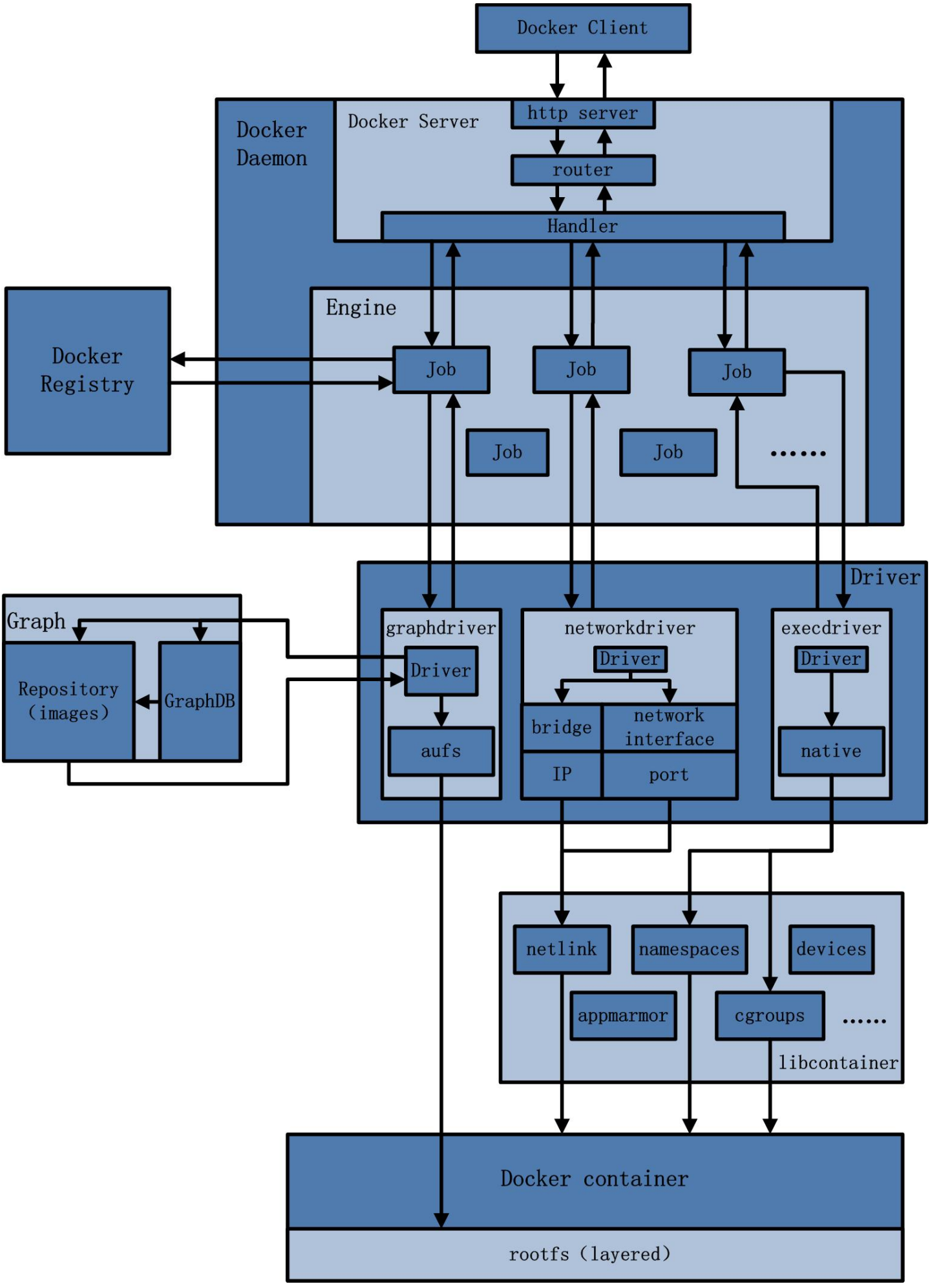


图3.1 Docker总架构图

如图3.1，不难看出，用户是使用Docker Client与Docker Daemon建立通信，并发送请求给后者。

而Docker Daemon作为Docker架构中的主体部分，首先提供Server的功能使其可以接受Docker Client的请求；而后Engine执行Docker内部的一系列工作，每一项工作都是以一个Job的形式的存在。

Job的运行过程中，当需要容器镜像时，则从Docker Registry中下载镜像，并通过镜像管理驱动graphdriver将下载镜像以Graph的形式存储；当需要为Docker创建网络环境时，通过网络管理驱动networkdriver创建并配置Docker容器网络环境；当需要限制Docker容器运行资源或执行用户指令等操作时，则通过execdriver来完成。

而libcontainer是一项独立的容器管理包，networkdriver以及execdriver都是通过libcontainer来实现具体对容器进行的操作。

当执行完运行容器的命令后，一个实际的Docker容器就处于运行状态，该容器拥有独立的文件系统，独立并且安全的运行环境等。

4 Docker架构内各模块的功能与实现分析

接下来，我们将从Docker总架构图入手，抽离出架构内各个模块，并对各个模块进行更为细化的架构分析与功能阐述。主要的模块有：Docker Client、Docker Daemon、Docker Registry、Graph、Driver、libcontainer以及Docker container。

4.1 Docker Client

Docker Client是Docker架构中用户用来和Docker Daemon建立通信的客户端。用户使用的可执行文件为docker，通过docker命令行工具可以发起众多管理container的请求。

Docker Client可以通过以下三种方式和Docker Daemon建立通信：tcp://host:port，unix://path_to_socket和fd://socketfd。为了简单起见，本文一律使用第一种方式作为讲述两者通信的原型。与此同时，与Docker Daemon建立连接并传输请求的时候，Docker Client可以通过设置命令行flag参数的形式设置安全传输层协议(TLS)的有关参数，保证传输的安全性。

Docker Client发送容器管理请求后，由Docker Daemon接受并处理请求，当Docker Client接收到返回的请求相应并简单处理后，Docker Client一次完整的生命周期就结束了。当需要继续发送容器管理请求时，用户必须再次通过docker可执行文件创建Docker Client。

4.2 Docker Daemon

Docker Daemon是Docker架构中一个常驻在后台的系统进程，功能是：接受并处理Docker Client发送的请求。该守护进程在后台启动了一个Server，Server负责接受Docker Client发送的请求；接受请求后，Server通过路由与分发调度，找到相应的Handler来执行请求。

Docker Daemon启动所使用的可执行文件也为docker，与Docker Client启动所使用的可执行文件docker相同。在docker命令执行时，通过传入的参数来判别Docker Daemon与Docker Client。

Docker Daemon的架构，大致可以分为以下三部分：Docker Server、Engine和Job。Daemon架构如图4.1。

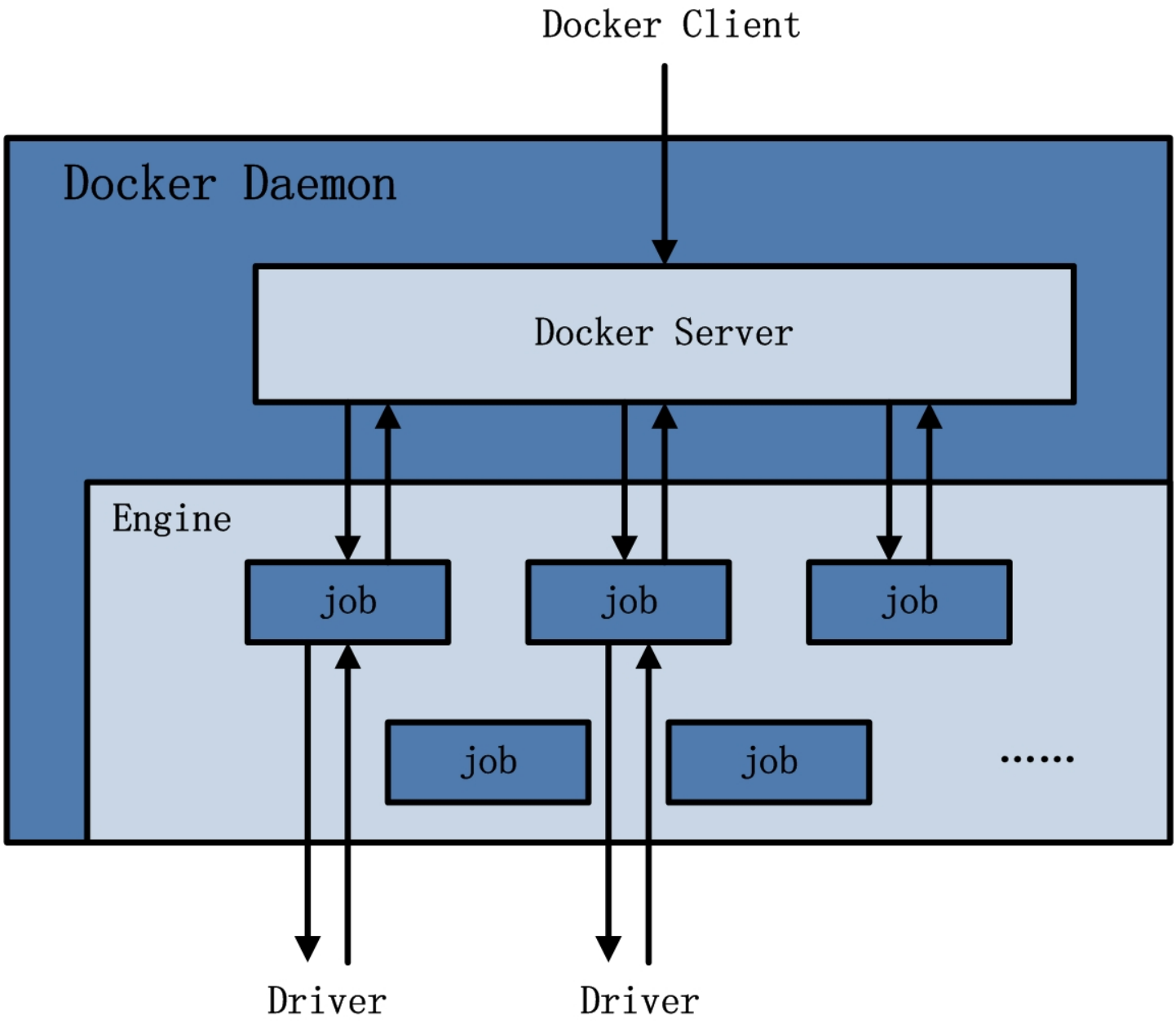


图4.1 Docker Daemon架构示意图

4.2.1 Docker Server

Docker Server在Docker架构中是专门服务于Docker Client的server。该server的功能是：接受并调度分发Docker Client发送的请求。Docker Server的架构如图4.2。

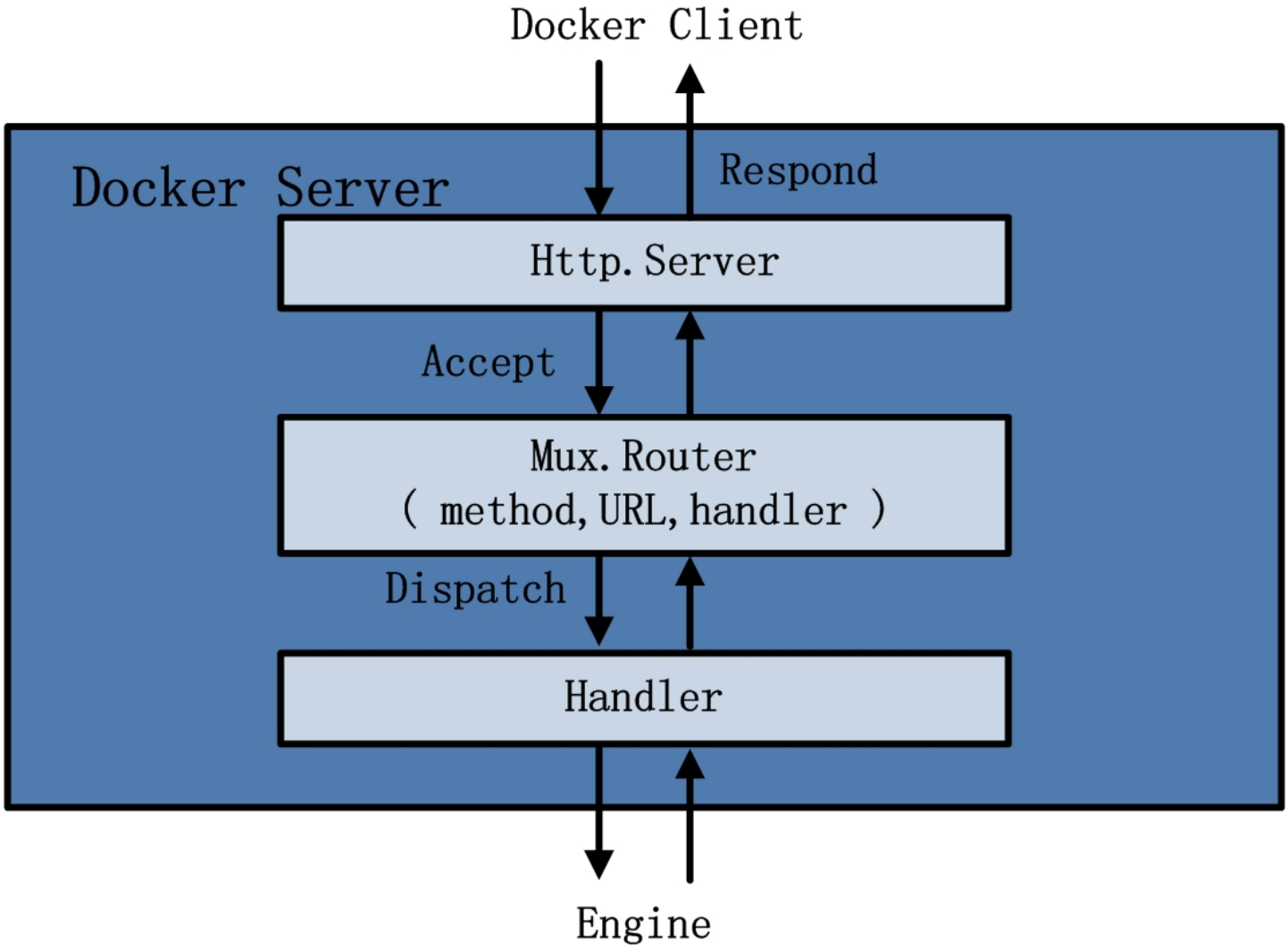


图4.2 Docker Server架构示意图

在Docker的启动过程中，通过包gorilla/mux，创建了一个mux.Router，提供请求的路由功能。在Golang中，gorilla/mux是一个强大的URL路由器以及调度分发器。该mux.Router中添加了众多的路由项，每一个路由项由HTTP请求方法（PUT、POST、GET或DELETE）、URL、Handler三部分组成。

若Docker Client通过HTTP的形式访问Docker Daemon，创建完mux.Router之后，Docker将Server的监听地址以及mux.Router作为参数，创建一个httpSrv=http.Server{}，最终执行httpSrv.Serve()为请求服务。

在Server的服务过程中，Server在listener上接受Docker Client的访问请求，并创建一个全新的goroutine来服务该请求。在goroutine中，首先读取请求内容，然后做解析工作，接着找到相应的路由项，随后调用相应的Handler来处理该请求，最后Handler处理完请求之后回复该请求。

需要注意的是：Docker Server的运行在Docker的启动过程中，是靠一个名为"serveapi"的job的运行来完成的。原则上，Docker Server的运行是众多job中的一个，但是为了强调Docker Server的重要性以及为后续job服务的重要特性，将该"serveapi"的job单独抽离出来分析，理解为Docker Server。

4.2.2 Engine

Engine是Docker架构中的运行引擎，同时也Docker运行的核心模块。它扮演Docker container存储仓库

的角色，并且通过执行job的方式来操纵管理这些容器。

在Engine数据结构的设计与实现过程中，有一个handler对象。该handler对象存储的都是关于众多特定job的handler处理访问。举例说明，Engine的handler对象中有一项为：{"create": daemon.ContainerCreate,}，则说明当名为"create"的job在运行时，执行的是daemon.ContainerCreate的handler。

4.2.3 Job

一个Job可以认为是Docker架构中Engine内部最基本的工作执行单元。Docker可以做的每一项工作，都可以抽象为一个job。例如：在容器内部运行一个进程，这是一个job；创建一个新的容器，这是一个job，从Internet上下载一个文档，这是一个job；包括之前在Docker Server部分说过的，创建Server服务于HTTP的API，这也是一个job，等等。

Job的设计者，把Job设计得与Unix进程相仿。比如说：Job有一个名称，有参数，有环境变量，有标准的输入输出，有错误处理，有返回状态等。

4.3 Docker Registry

Docker Registry是一个存储容器镜像的仓库。而容器镜像是在容器被创建时，被加载用来初始化容器的文件架构与目录。

在Docker的运行过程中，Docker Daemon会与Docker Registry通信，并实现搜索镜像、下载镜像、上传镜像三个功能，这三个功能对应的job名称分别为"search"，"pull" 与 "push"。

其中，在Docker架构中，Docker可以使用公有的Docker Registry，即大家熟知的Docker Hub，如此一来，Docker获取容器镜像文件时，必须通过互联网访问Docker Hub；同时Docker也允许用户构建本地私有的Docker Registry，这样可以保证容器镜像的获取在内网完成。

4.4 Graph

Graph在Docker架构中扮演已下载容器镜像的保管者，以及已下载容器镜像之间关系的记录者。一方面，Graph存储着本地具有版本信息的文件系统镜像，另一方面也通过GraphDB记录着所有文件系统镜像彼此之间的关系。Graph的架构如图4.3。

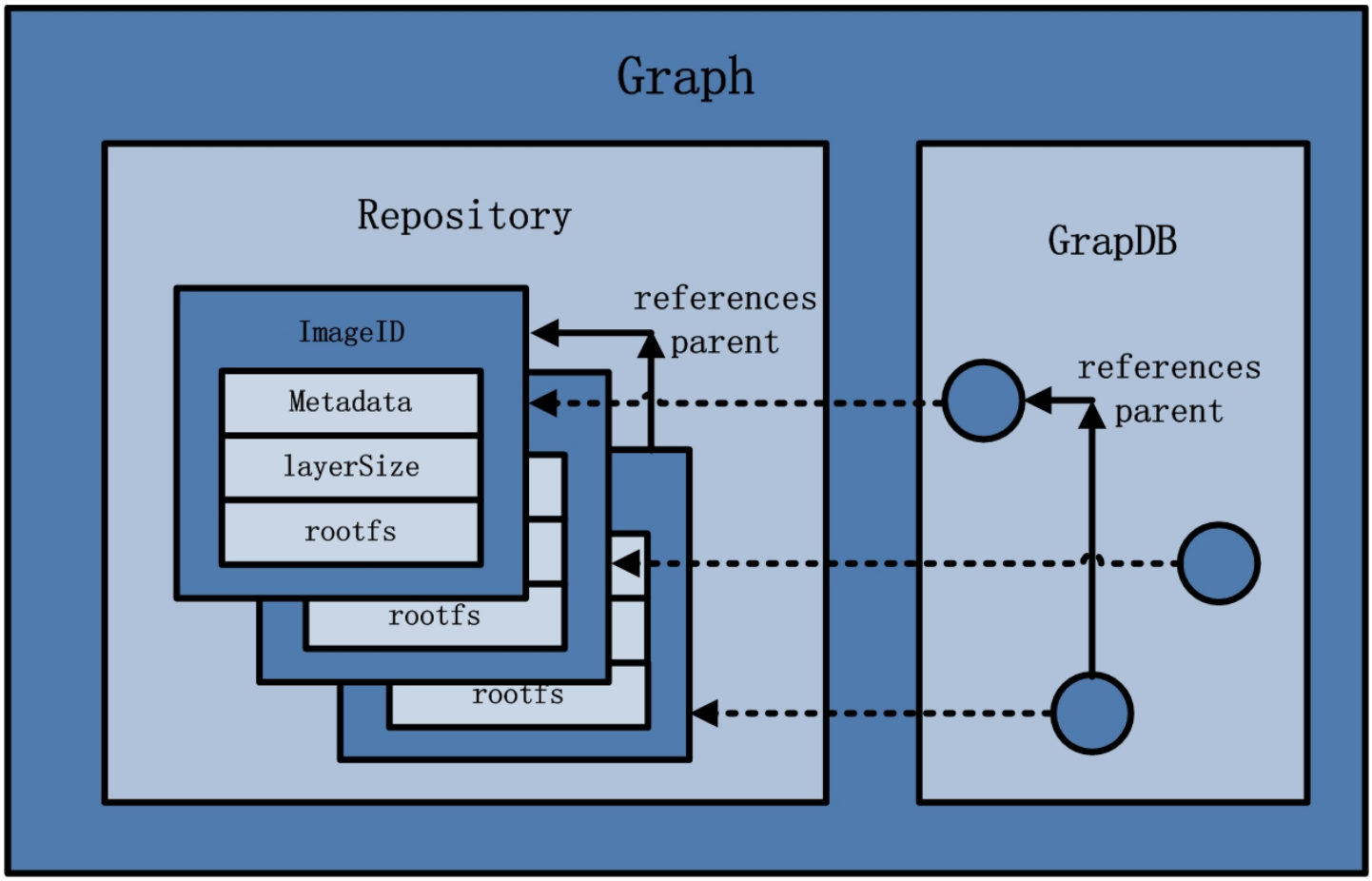


图4.3 Graph架构示意图

其中，GraphDB是一个构建在SQLite之上的小型图数据库，实现了节点的命名以及节点之间关联关系的记录。它仅仅实现了大多数图数据库所拥有的一个小的子集，但是提供了简单的接口表示节点之间的关系。

同时在Graph的本地目录中，关于每一个的容器镜像，具体存储的信息有：该容器镜像的元数据，容器镜像的大小信息，以及该容器镜像所代表的具体rootfs。

4.5 Driver

Driver是Docker架构中的驱动模块。通过Driver驱动，Docker可以实现对Docker容器执行环境的定制。由于Docker运行的生命周期中，并非用户所有的操作都是针对Docker容器的管理，另外还有关于Docker运行信息的获取，Graph的存储与记录等。因此，为了将Docker容器的管理从Docker Daemon内部业务逻辑中区分开来，设计了Driver层驱动来接管所有这部分请求。

在Docker Driver的实现中，可以分为以下三类驱动：graphdriver、networkdriver和execdriver。

graphdriver主要用于完成容器镜像的管理，包括存储与获取。即当用户需要下载指定的容器镜像时，graphdriver将容器镜像存储在本地的指定目录；同时当用户需要使用指定的容器镜像来创建容器的rootfs时，graphdriver从本地镜像存储目录中获取指定的容器镜像。

在graphdriver的初始化过程之前，有4种文件系统或类文件系统在其内部注册，它们分别是aufs、btrfs、vfs和devmapper。而Docker在初始化之时，通过获取系统环境变量”DOCKER_DRIVER”来提取所使用driver的指定类型。而之后所有的graph操作，都使用该driver来执行。

graphdriver的架构如图4.4：

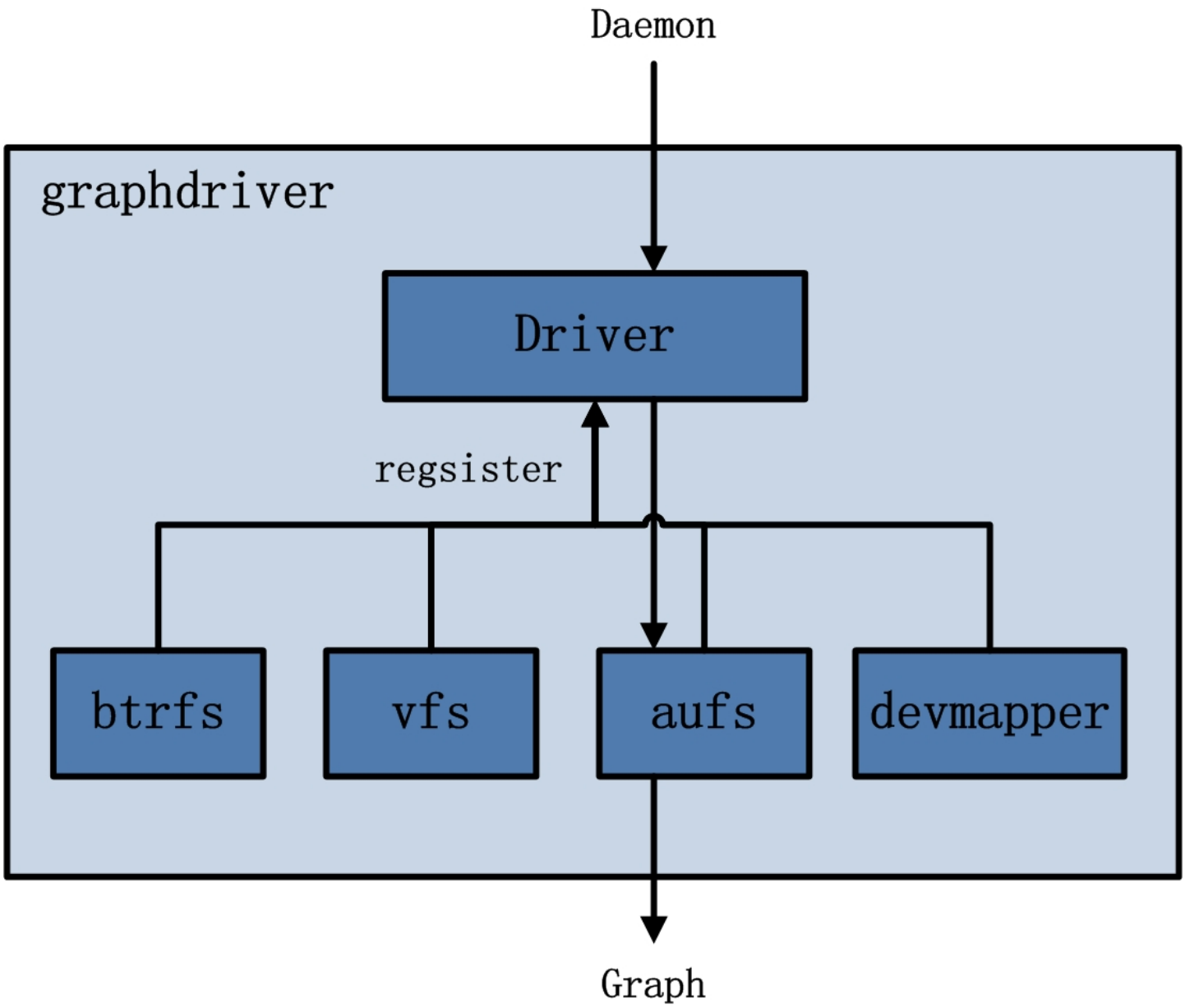


图4.4 graphdriver架构示意图

networkdriver的用途是完成Docker容器网络环境的配置，其中包括Docker启动时为Docker环境创建网桥；Docker容器创建时为其创建专属虚拟网卡设备；以及为Docker容器分配IP、端口并与宿主机做端口映射，设置容器防火墙策略等。networkdriver的架构如图4.5：

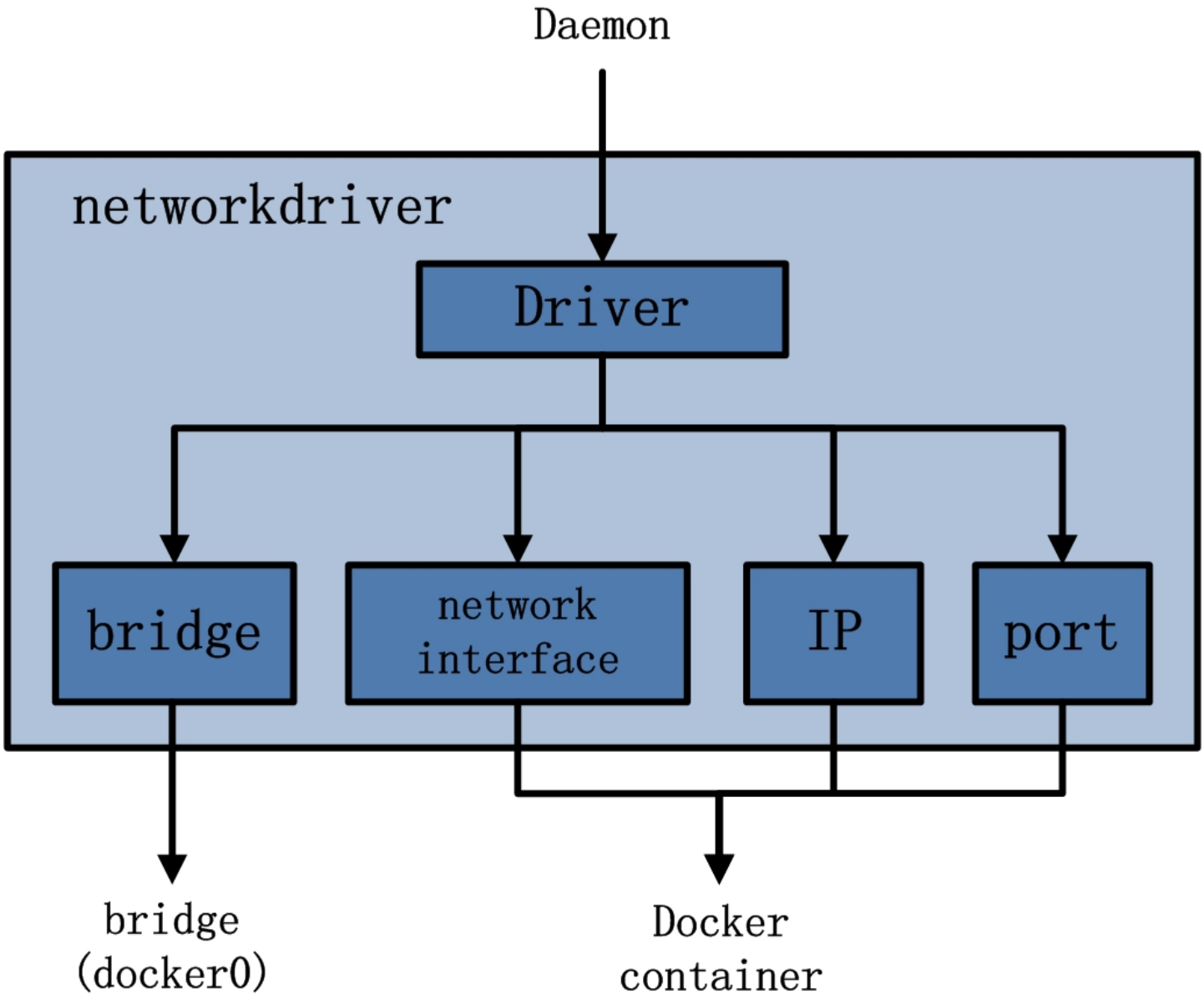


图4. 5 networkdriver架构示意图

execdriver作为Docker容器的执行驱动，负责创建容器运行命名空间，负责容器资源使用的统计与限制，负责容器内部进程的真正运行等。在execdriver的实现过程中，原先可以使用LXC驱动调用LXC的接口，来操纵容器的配置以及生命周期，而现在execdriver默认使用native驱动，不依赖于LXC。具体体现在Daemon启动过程中加载的ExecDriverflag参数，该参数在配置文件已经被设为"native"。这可以认为是Docker在1.2版本上一个很大的改变，或者说Docker实现跨平台的一个先兆。execdriver架构如图4.6：

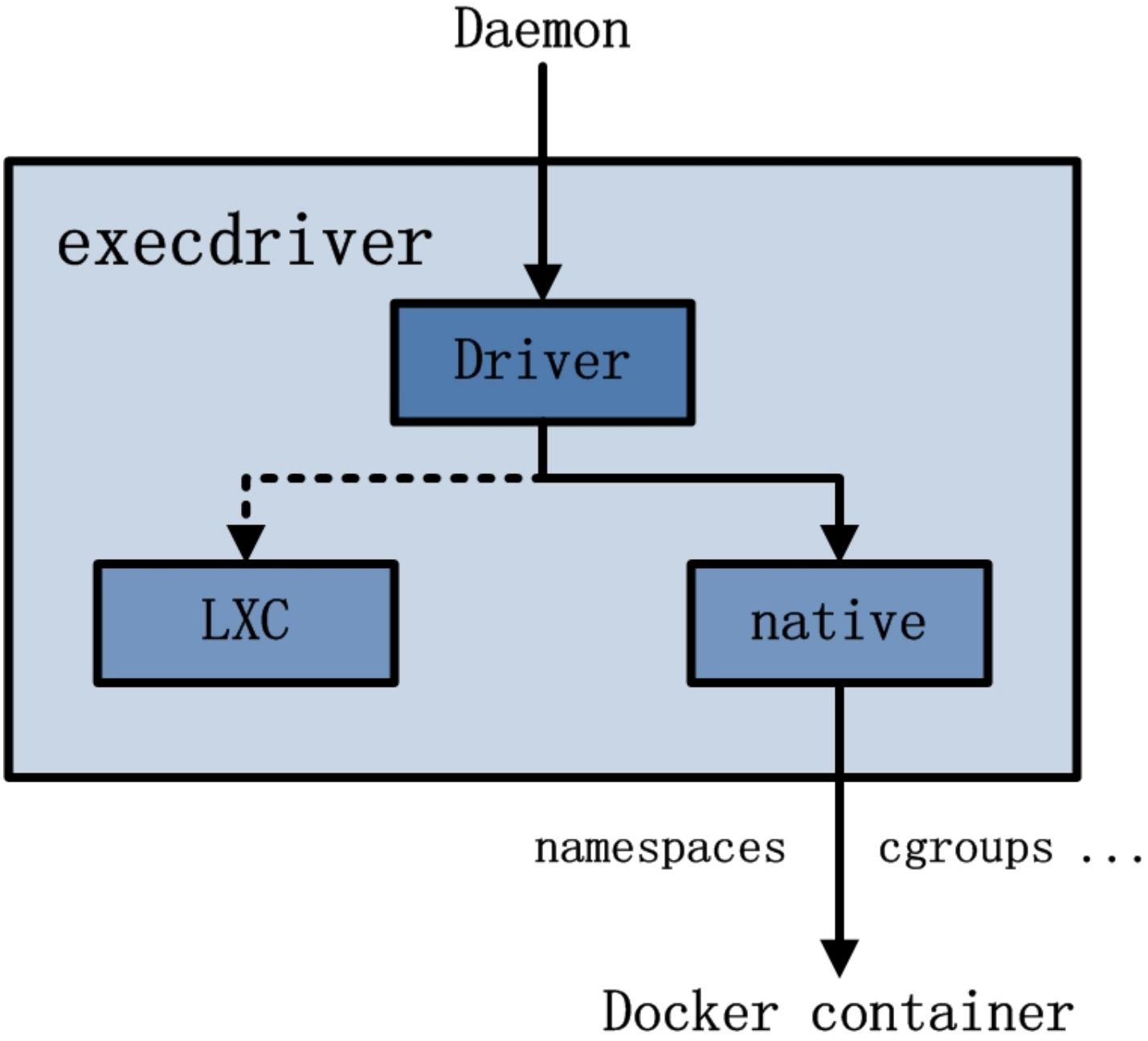


图4.6 execdriver架构示意图

4.6 libcontainer

libcontainer是Docker架构中一个使用Go语言设计实现的库，设计初衷是希望该库可以不依靠任何依赖，直接访问内核中与容器相关的API。

正是由于libcontainer的存在，Docker可以直接调用libcontainer，而最终操纵容器的namespace、cgroups、apparmor、网络设备以及防火墙规则等。这一系列操作的完成都不需要依赖LXC或者其他包。libcontainer架构如图4.7：

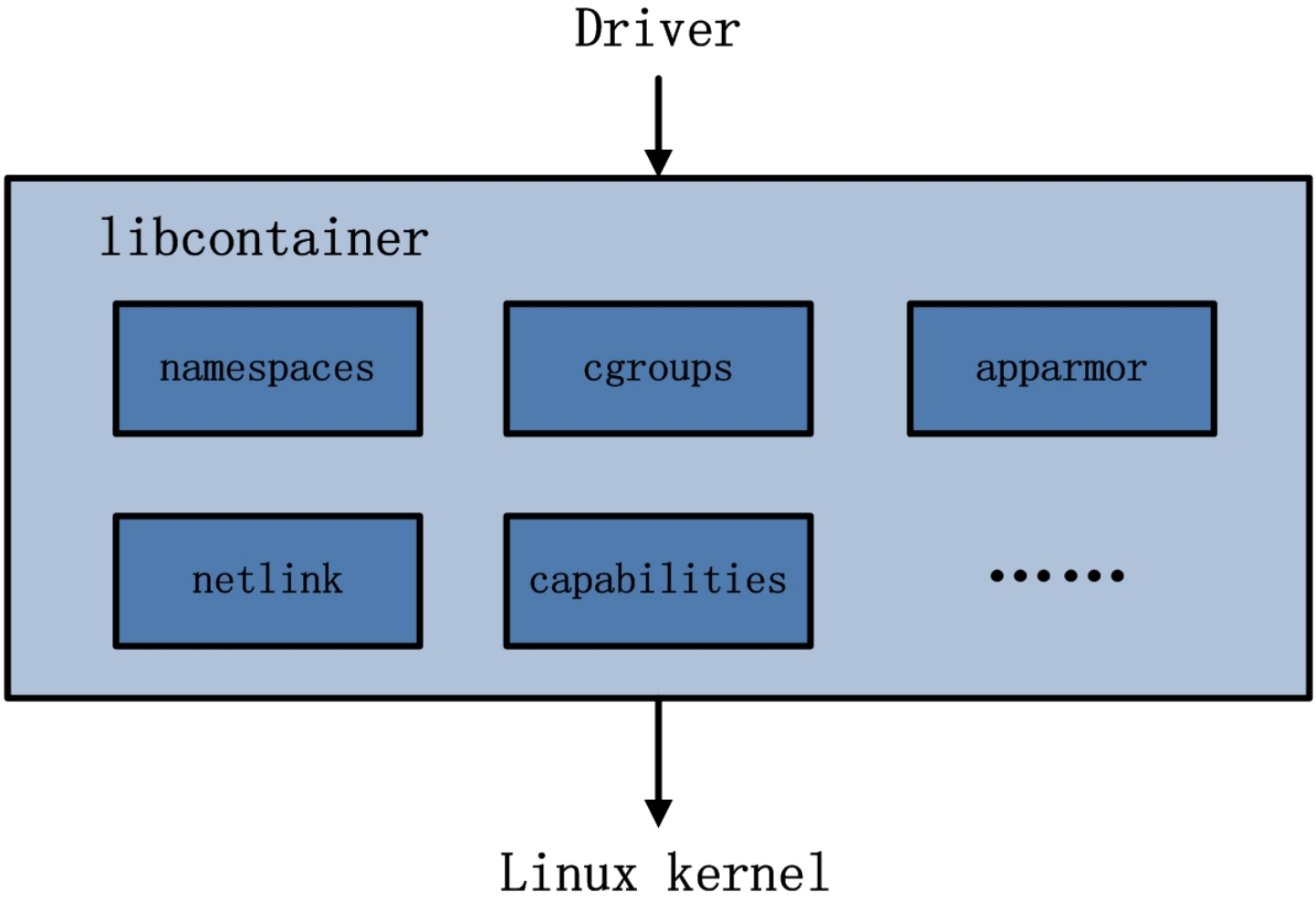


图4.7 libcontainer示意图

另外，libcontainer提供了一整套标准的接口来满足上层对容器管理的需求。或者说，libcontainer屏蔽了Docker上层对容器的直接管理。又由于libcontainer使用Go这种跨平台的语言开发实现，且本身又可以被上层多种不同的编程语言访问，因此很难说，未来的Docker就一定会紧紧地和Linux捆绑在一起。而于此同时，Microsoft在其著名云计算平台Azure中，也添加了对Docker的支持，可见Docker的开放程度与业界的火热度。

暂不谈Docker，由于libcontainer的功能以及其本身与系统的松耦合特性，很有可能会在其他以容器为原型的平台出现，同时也很有可能催生出云计算领域全新的项目。

4.7 Docker container

Docker container（Docker容器）是Docker架构中服务交付的最终体现形式。

Docker按照用户的需求与指令，订制相应的Docker容器：

- 用户通过指定容器镜像，使得Docker容器可以自定义rootfs等文件系统；
- 用户通过指定计算资源的配额，使得Docker容器使用指定的计算资源；
- 用户通过配置网络及其安全策略，使得Docker容器拥有独立且安全的网络环境；
- 用户通过指定运行的命令，使得Docker容器执行指定的工作。

Docker容器示意图如图4.8：

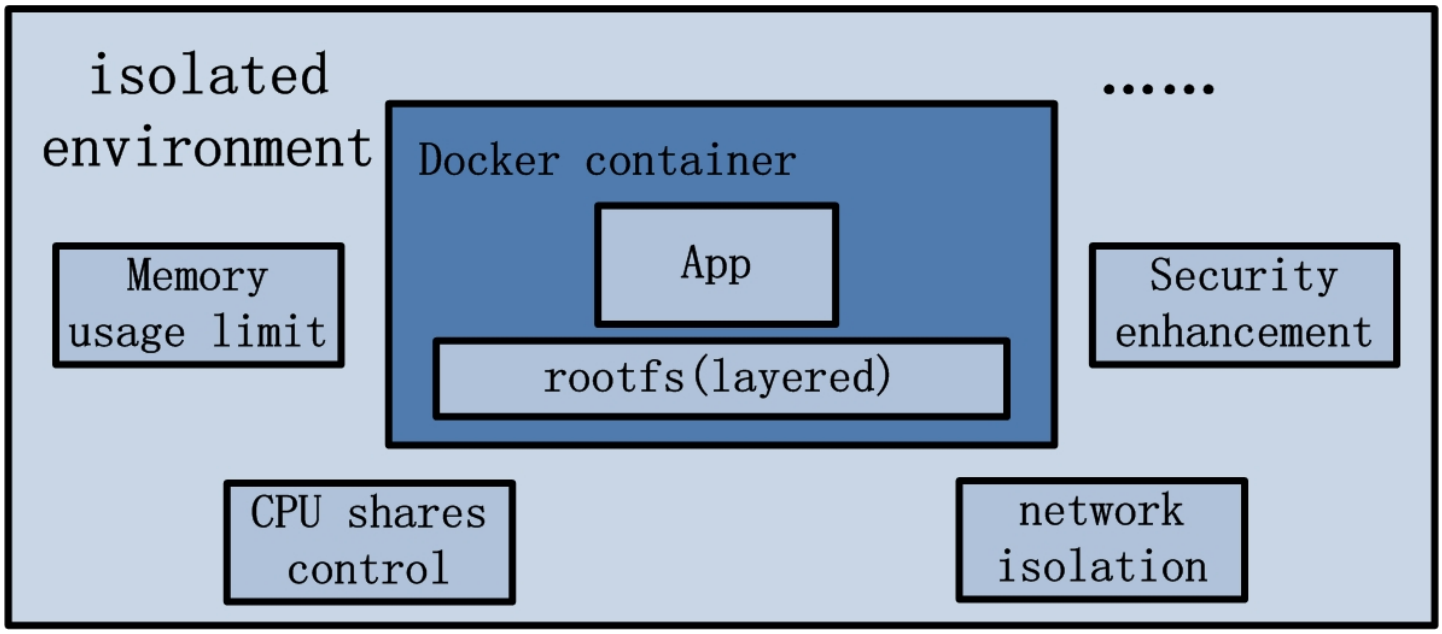


图4.8 Docker容器示意图

5 Docker运行案例分析

上一章节着重于Docker架构中各个部分的介绍。本章的内容，将以串联Docker各模块来简要分析，分析原型为Docker中的docker pull与docker run两个命令。

5.1 docker pull

docker pull命令的作用为：从Docker Registry中下载指定的容器镜像，并存储在本地的Graph中，以备后续创建Docker容器时的使用。docker pull命令执行流程如图5.1。

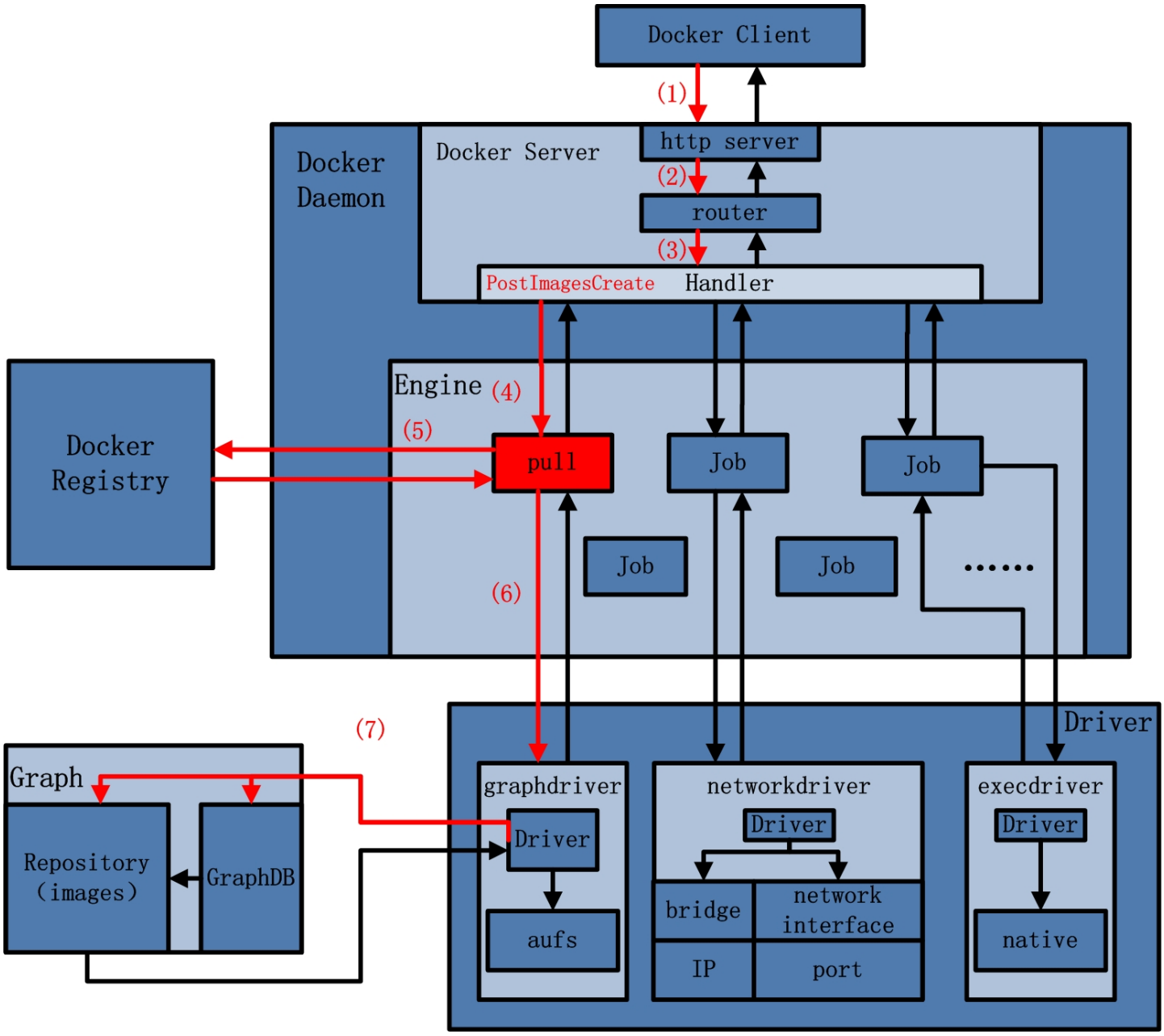


图5.1 docker pull命令执行流程示意图

如图，图中标记的红色箭头表示docker pull命令在发起后，Docker所做的一系列运行。以下逐一分析这些步骤。

(1) Docker Client接受docker pull命令，解析完请求以及收集完请求参数之后，发送一个HTTP请求给Docker Server，HTTP请求方法为POST，请求URL为"/images/create? "+"xxx"；

(2) Docker Server接受以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法来确定执行该请求的具体handler；

(3) mux.Router将请求路由分发至相应的handler，具体为PostImagesCreate；

(4) 在PostImageCreate这个handler之中，一个名为"pull"的job被创建，并开始执行；

(5) 名为"pull"的job在执行过程中，执行pullRepository操作，即从Docker Registry中下载相应的一个或者多个image；

(6) 名为"pull"的job将下载的image交给graphdriver；

(7) graphdriver负责将image进行存储，一方创建graph对象，另一方面在GraphDB中记录image之间的关系。

5.2 docker run

docker run命令的作用是在一个全新的Docker容器内部运行一条指令。Docker在执行这条命令的时候，所做工作可以分为两部分：第一，创建Docker容器所需的rootfs；第二，创建容器的网络等运行环境，并真正运行用户指令。因此，在整个执行流程中，Docker Client给Docker Server发送了两次HTTP请求，第二次请求的发起取决于第一次请求的返回状态。Docker run命令执行流程如图5.2。

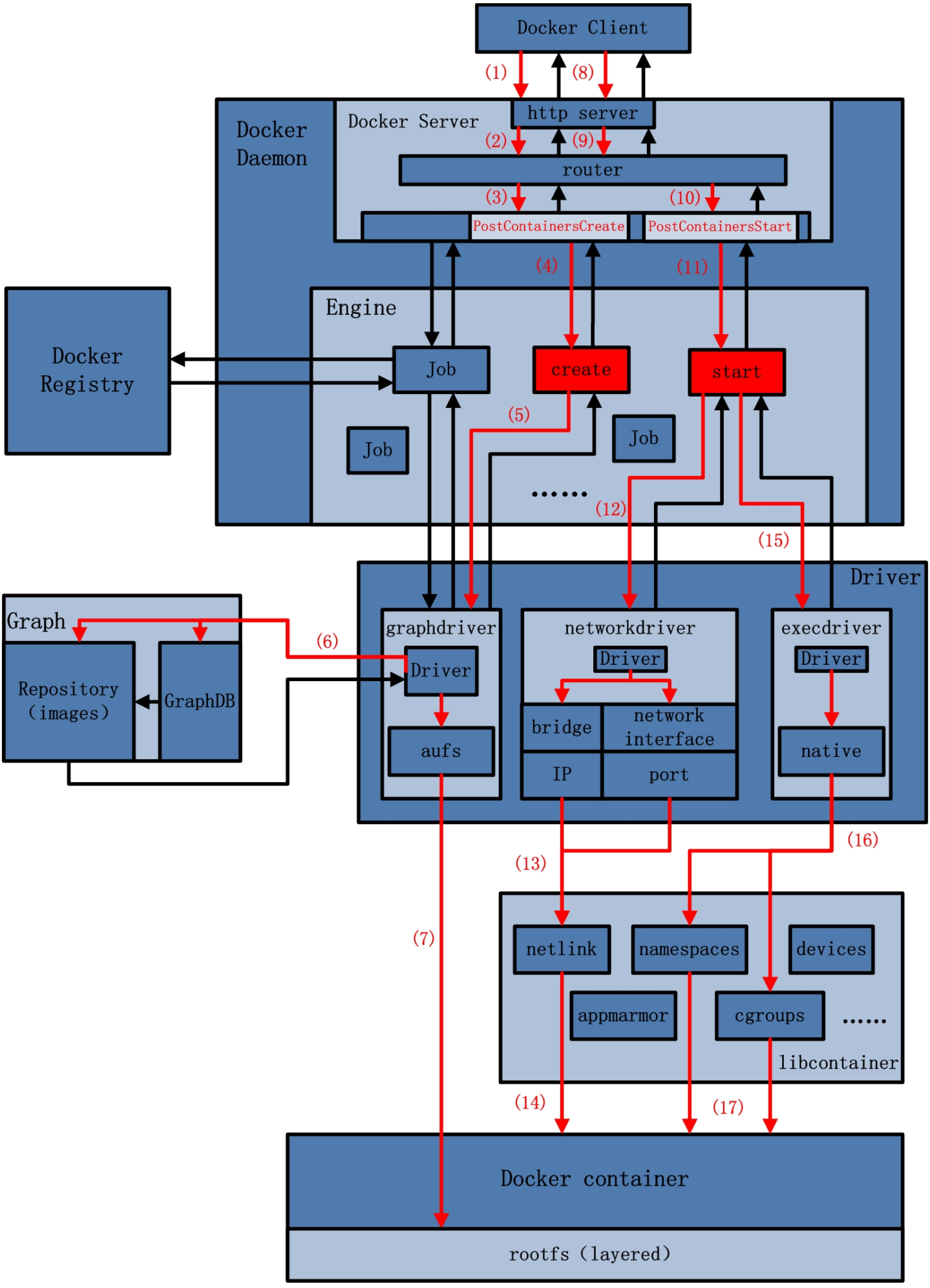


图5.2 docker run命令执行流程示意图

如图，图中标记的红色箭头表示docker run命令在发起后，Docker所做的一系列运行。以下逐一分析这些步骤。

(1) Docker Client接受docker run命令，解析完请求以及收集完请求参数之后，发送一个HTTP请求给Docker Server，HTTP请求方法为POST，请求URL为"/containers/create? "+"xxx"；

(2) Docker Server接受以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法来确定执行该请求的具体handler；

(3) mux.Router将请求路由分发至相应的handler，具体为PostContainersCreate；

(4) 在PostImageCreate这个handler之中，一个名为"create"的job被创建，并开始让该job运行；

(5) 名为"create"的job在运行过程中，执行Container.Create操作，该操作需要获取容器镜像来为Docker容器创建rootfs，即调用graphdriver；

(6) graphdriver从Graph中获取创建Docker容器rootfs所需要的所有的镜像；

(7) graphdriver将rootfs所有镜像，加载安装至Docker容器指定的文件目录下；

(8) 若以上操作全部正常执行，没有返回错误或异常，则Docker Client收到Docker Server返回状态之后，发起第二次HTTP请求。请求方法为"POST"，请求URL为"/containers/"+container_ID+"/start"；

(9) Docker Server接受以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法来确定执行该请求的具体handler；

(10)mux.Router将请求路由分发至相应的handler，具体为PostContainersStart；

(11)在PostContainersStart这个handler之中，名为"start"的job被创建，并开始执行；

(12)名为"start"的job执行完初步的配置工作后，开始配置与创建网络环境，调用networkdriver；

(13)networkdriver需要为指定的Docker容器创建网络接口设备，并为其分配IP，port，以及设置防火墙规则，相应的操作转交至libcontainer中的netlink包来完成；

(14)netlink完成Docker容器的网络环境配置与创建；

(15)返回至名为"start"的job，执行完一些辅助性操作后，job开始执行用户指令，调用execdriver；

(16)execdriver被调用，初始化Docker容器内部的运行环境，如命名空间，资源控制与隔离，以及用户命令的执行，相应的操作转交至libcontainer来完成；

(17)libcontainer被调用，完成Docker容器内部的运行环境初始化，并最终执行用户要求启动的命令。

6 总结

本文从Docker 1.2的源码入手，分析抽象出Docker的架构图，并对该架构图中的各个模块进行功能与实现的分析，最后通过两个docker命令展示了Docker内部的运行。

通过对Docker架构的学习，可以全面深化对Docker设计、功能与价值的理解。同时在借助Docker实现用户定制的分布式系统时，也能更好地找到已有平台与Docker较为理想的契合点。另外，熟悉Docker现有架构以及设计思想，也能对云计算PaaS领域带来更多的启发，催生出更多实践与创新。

7 作者简介

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

8 参考文献

http://en.wikipedia.org/wiki/Docker_(software)">http://en.wikipedia.org/wiki/Docker_(software)

http://www.slideshare.net/rajdeep/docker-architecturev2

https://github.com/docker/libcontainer

http://www.infoq.com/cn/articles/docker-core-technology-preview

https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/

https://crosbymichael.com/the-lost-packages-of-docker.html

（二）：Docker Client创建与命令执行

1. 前言

如今，Docker作为业界领先的轻量级虚拟化容器管理引擎，给全球开发者提供了一种新颖、便捷的软件集成测试与部署之道。在团队开发软件时，Docker可以提供可复用的运行环境、灵活的资源配置、便捷的集成测试方法以及一键式的部署方式。可以说，Docker的优势在简化持续集成、运维部署方面体现得淋漓尽致，它完全让开发者从持续集成、运维部署方面中解放出来，把精力真正地倾注在开发上。

然而，把Docker的功能发挥到极致，并非一件易事。在深刻理解Docker架构的情况下，熟练掌握Docker Client的使用也非常有必要。前者可以参阅《Docker源码分析》系列之Docker架构篇，而本文主要针对后者，从源码的角度分析Docker Client，力求帮助开发者更深刻的理解Docker Client的具体实现，最终更好的掌握Docker Client的使用方法。即本文为《Docker源码分析》系列的第二篇——Docker Client篇。

2. Docker Client源码分析章节安排

本文从源码的角度，主要分析Docker Client的两个方面：创建与命令执行。整个分析过程可以分为两个部分：

第一部分分析Docker Client的创建。这部分的分析可分为以下三个步骤：

- 分析如何通过docker命令，解析出命令行flag参数，以及docker命令中的请求参数；
- 分析如何处理具体的flag参数信息，并收集Docker Client所需的配置信息；
- 分析如何创建一个Docker Client。

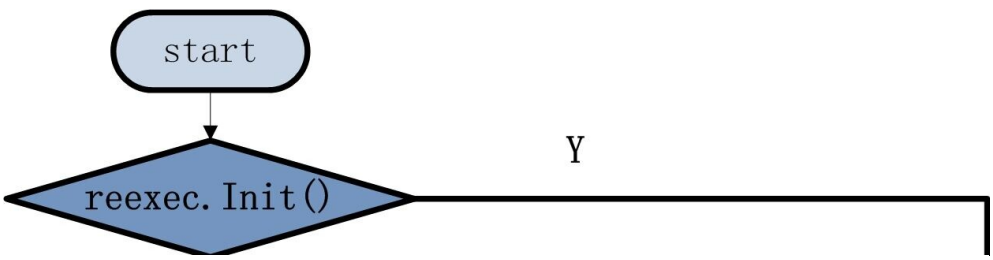
第二部分在已有Docker Client的基础上，分析如何执行docker命令。这部分的分析又可分为以下两个步骤：

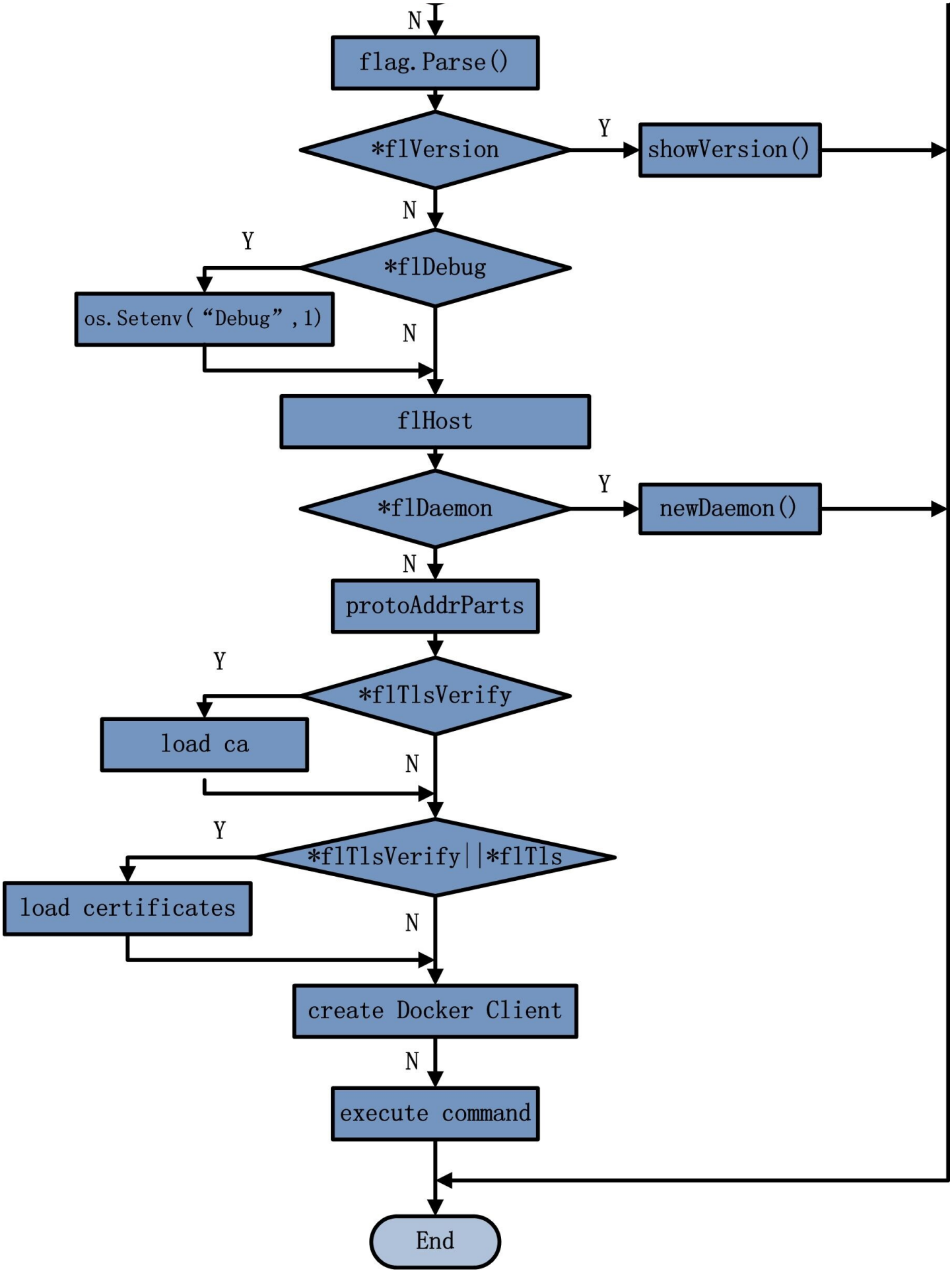
- 分析如何解析docker命令中的请求参数，获取相应请求的类型；
- 分析Docker Client如何执行具体的请求命令，最终将请求发送至Docker Server。

3. Docker Client的创建

Docker Client的创建，实质上是Docker用户通过可执行文件docker，与Docker Server建立联系的客户端。以下分三个小节分别阐述Docker Client的创建流程。

以下为整个docker源代码运行的流程图：





上图通过流程图的方式，使得读者更为清晰的了解Docker Client创建及执行请求的过程。其中涉及了诸多源代码中的特有名词，在下文中会一一解释与分析。

3.1. Docker命令的flag参数解析

众所周知，在Docker的具体实现中，Docker Server与Docker Client均由可执行文件docker来完成创建并启动。那么，了解docker可执行文件通过何种方式区分两者，就显得尤为重要。

对于两者，首先举例说明其中的区别。Docker Server的启动，命令为docker -d或docker --daemon=true；而Docker Client的启动则体现为docker --daemon=false ps、docker pull NAME等。

可以把以上Docker请求中的参数分为两类：第一类为命令行参数，即docker程序运行时所需提供的参数，如: -D、--daemon=true、--daemon=false等；第二类为docker发送给Docker Server的实际请求参数，如：ps、pull NAME等。

对于第一类，我们习惯将其称为flag参数，在go语言的标准库中，同时还提供了一个flag包，方便进行命令行参数的解析。

交待以上背景之后，随即进入实现Docker Client创建的源码，位于./docker/docker/docker.go，该go文件包含了整个Docker的main函数，也就是整个Docker（不论Docker Daemon还是Docker Client）的运行入口。部分main函数代码如下：

```
func main() {
    if reexec.Init() {
      return
    }
    flag.Parse()
    // FIXME: validate daemon flags here
    ......
}
```

在以上代码中，首先判断reexec.Init()方法的返回值，若为真，则直接退出运行，否则的话继续执行。查看位于./docker/reexec/reexec.go中reexec.Init()的定义，可以发现由于在docker运行之前没有任何的Initializer注册，故该代码段执行的返回值为假。

紧接着，main函数通过调用flag.Parse()解析命令行中的flag参数。查看源码可以发现Docker在./docker/docker/flag.go中定义了多个flag参数，并通过init函数进行初始化。代码如下：

```
var (
  flVersion     = flag.Bool([]string{"v", "-version"}, false, "Print version information and quit")
  flDaemon      = flag.Bool([]string{"d", "-daemon"}, false, "Enable daemon mode")
  flDebug       = flag.Bool([]string{"D", "-debug"}, false, "Enable debug mode")
  flSocketGroup = flag.String([]string{"G", "-group"}, "docker", "Group to assign the unix socket specifi
ed by -H when running in daemon mode use '' (the empty string) to disable setting of a group")
  flEnableCors  = flag.Bool([]string{"#api-enable-cors", "-api-enable-cors"}, false, "Enable CORS header
s in the remote API")
  flTls         = flag.Bool([]string{"-tls"}, false, "Use TLS; implied by tls-verify flags")
  flTlsVerify   = flag.Bool([]string{"-tlsverify"}, false, "Use TLS and verify the remote (daemon: verify clie
nt, client: verify daemon)")

  // these are initialized in init() below since their default values depend on dockerCertPath which isn't
 fully initialized until init() runs
  flCa    *string
  flCert  *string
  flKey   *string
  flHosts []string
)

func init() {
  flCa = flag.String([]string{"-tlscacert"}, filepath.Join(dockerCertPath, defaultCaFile), "Trust only remot
es providing a certificate signed by the CA given here")
  flCert = flag.String([]string{"-tlscert"}, filepath.Join(dockerCertPath, defaultCertFile), "Path to TLS certi
ficate file")
  flKey = flag.String([]string{"-tlskey"}, filepath.Join(dockerCertPath, defaultKeyFile), "Path to TLS key fil
e")
  opts.HostListVar(&flHosts, []string{"H", "-host"}, "The socket(s) to bind to in daemon mode\nspecifie
d using one or more tcp://host:port, unix:///path/to/socket, fd://* or fd://socketfd.")
}
```

这里涉及到了Golang的一个特性，即init函数的执行。在Golang中init函数的特性如下：

- init函数用于程序执行前包的初始化工作，比如初始化变量等；
- 每个包可以有多个init函数；
- 包的每一个源文件也可以有多个init函数；
- 同一个包内的init函数的执行顺序没有明确的定义；
- 不同包的init函数按照包导入的依赖关系决定初始化的顺序；
- init函数不能被调用，而是在main函数调用前自动被调用。

因此，在main函数执行之前，Docker已经定义了诸多flag参数，并对很多flag参数进行初始化。定义的命令行flag参数有：flVersion、flDaemon、flDebug、flSocketGroup、flEnableCors、flTls、flTlsVerify、flCa、flCert、flKey等。

以下具体分析flDaemon：

- 定义：flDaemon = flag.Bool([]string{"d", "-daemon"}, false, "Enable daemon mode")
- flDaemon的类型为Bool类型
- flDaemon名称为”d”或者”-daemon”，该名称会出现在docker命令中

- flDaemon的默认值为false
- flDaemon的帮助信息为”Enable daemon mode”
- 访问flDaemon的值时，使用指针* flDaemon解引用访问

在解析命令行flag参数时，以下的语言为合法的：

- -d, --daemon
- -d=true, --daemon=true
- -d=”true”, --daemon=”true”
- -d='true', --daemon='true'

当解析到第一个非定义的flag参数时，命令行flag参数解析工作结束。举例说明，当执行docker命令docker --daemon=false --version=false ps时，flag参数解析主要完成两个工作：

- 完成命令行flag参数的解析，名为-daemon和-version的flag参数flDaemon和flVersion分别获得相应的值，均为false；
- 遇到第一个非flag参数的参数ps时，将ps及其之后所有的参数存入flag.Args()，以便之后执行Docker Client具体的请求时使用。

如需深入学习flag的解析，可以参见源码命令行参数flag的解析。

3.2. 处理flag信息并收集Docker Client的配置信息

有了以上flag参数解析的相关知识，分析Docker的main函数就变得简单易懂很多。通过总结，首先列出源代码中处理的flag信息以及收集Docker Client的配置信息，然后再一一对此分析：

- 处理的flag参数有：flVersion、flDebug、flDaemon、flTlsVerify以及flTls；
- 为Docker Client收集的配置信息有：protoAddrParts(通过flHosts参数获得，作用为提供Docker Client与Server的通信协议以及通信地址)、tlsConfig(通过一系列flag参数获得，如*flTls*、flTlsVerify，作用为提供安全传输层协议的保障)。

随即分析处理这些flag参数信息，以及配置信息。

在flag.Parse()之后的代码如下：

```
if *flVersion {
  showVersion()
  return
}
```

不难理解的是，当经过解析flag参数后，若flVersion参数为真时，调用showVersion()显示版本信息，并从main函数退出；否则的话，继续往下执行。

```
if *flDebug {
  os.Setenv("DEBUG", "1")
}
```

若flDebug参数为真的话，通过os包的中Setenv函数创建一个名为DEBUG的系统环境变量，并将其值设为”1”。继续往下执行。

```
if len(flHosts) == 0 {
  defaultHost := os.Getenv("DOCKER_HOST")
  if defaultHost == "" || *flDaemon {
    // If we do not have a host, default to unix socket
    defaultHost = fmt.Sprintf("unix://%s", api.DEFAULTUNIXSOCKET)
  }
  if _, err := api.ValidateHost(defaultHost); err != nil {
    log.Fatal(err)
  }
  flHosts = append(flHosts, defaultHost)
}
```

以上的源码主要分析内部变量flHosts。flHosts的作用是为Docker Client提供所要连接的host对象，也为Docker Server提供所要监听的对象。

分析过程中，首先判断flHosts变量是否长度为0，若是的话，通过os包获取名为DOCKER_HOST环境变量的值，将其赋值于defaultHost。若defaultHost为空或者flDaemon为真的话，说明目前还没有一个定义的host对象，则将其默认设置为unix socket，值为api.DEFAULTUNIXSOCKET，该常量位于./docker/api/common.go，值为"/var/run/docker.sock"，故defaultHost为”unix:///var/run/docker.sock”。验证该defaultHost的合法性之后，将defaultHost的值追加至flHost的末尾。继续往下执行。

```
if *flDaemon {
  mainDaemon()
  return
}
```

若flDaemon参数为真的话，则执行mainDaemon函数，实现Docker Daemon的启动，若mainDaemon函数执行完毕，则退出main函数，一般mainDaemon函数不会主动终结。由于本章节介绍Docker Client的启动，故假设flDaemon参数为假，不执行以上代码块。继续往下执行。

```
if len(flHosts) > 1 {
  log.Fatal("Please specify only one -H")
}
protoAddrParts := strings.SplitN(flHosts[0], "://", 2)
```

以上，若flHosts的长度大于1的话，则抛出错误日志。接着将flHosts这个string数组中的第一个元素，进

行分割，通过”://”来分割，分割出的两个部分放入变量protoAddrParts数组中。protoAddrParts的作用为解析出与Docker Server建立通信的协议与地址，为Docker Client创建过程中不可或缺的配置信息之一。

```
  var (
    cli       *client.DockerCli
    tlsConfig tls.Config
  )
tlsConfig.InsecureSkipVerify = true
```

由于之前已经假设过flDaemon为假，则可以认定main函数的运行是为了Docker Client的创建与执行。在这里创建两个变量：一个为类型是client.DockerCli指针的对象cli，另一个为类型是tls.Config的对象tlsConfig。并将tlsConfig的InsecureSkipVerify属性设置为真。TlsConfig对象的创建是为了保障cli在传输数据的时候，遵循安全传输层协议(TLS)。安全传输层协议(TLS) 用于两个通信应用程序之间保密性与数据完整性。tlsConfig是Docker Client创建过程中可选的配置信息。

```
  // If we should verify the server, we need to load a trusted ca
  if *flTlsVerify {
    *flTls = true
    certPool := x509.NewCertPool()
    file, err := ioutil.ReadFile(*flCa)
    if err != nil {
      log.Fatalf("Couldn't read ca cert %s: %s", *flCa, err)
    }
    certPool.AppendCertsFromPEM(file)
    tlsConfig.RootCAs = certPool
    tlsConfig.InsecureSkipVerify = false
  }
```

若flTlsVerify这个flag参数为真的话，则说明需要验证server端的安全性，tlsConfig对象需要加载一个受信的ca文件。该ca文件的路径为*flCA参数的值，最终完成tlsConfig对象中RootCAs属性的赋值，并将InsecureSkipVerify属性置为假。

```
// 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 {
        log.Fatalf("Couldn't load X509 key pair: %s. Key encrypted?", err)
      }
      tlsConfig.Certificates = []tls.Certificate{cert}
    }
  }
```

如果flTls和flTlsVerify两个flag参数中有一个为真，则说明需要加载以及发送client端的证书。最终将证书内容交给tlsConfig的Certificates属性。

至此，flag参数已经全部处理，并已经收集完毕Docker Client所需的配置信息。之后的内容为Docker Client如何实现创建并执行。

3.3. Docker Client的创建

Docker Client的创建其实就是在已有配置参数信息的情况，通过Client包中的NewDockerCli方法创建一个实例cli，源码实现如下：

```
  if *flTls || *flTlsVerify {
    cli = client.NewDockerCli(os.Stdin, os.Stdout, os.Stderr, protoAddrParts[0], protoAddrParts[1], &tlsC
onfig)
  } else {
    cli = client.NewDockerCli(os.Stdin, os.Stdout, os.Stderr, protoAddrParts[0], protoAddrParts[1], nil)
  }
```

如果flag参数flTls为真或者flTlsVerify为真的话，则说明需要使用TLS协议来保障传输的安全性，故创建Docker Client的时候，将TlsConfig参数传入；否则的话，同样创建Docker Client，只不过TlsConfig为nil。

关于Client包中的NewDockerCli函数的实现，可以具体参见./docker/api/client/cli.go。

```
func NewDockerCli(in io.ReadCloser, out, err io.Writer, proto, addr string, tlsConfig *tls.Config) *Docke
rCli {
  var (
    isTerminal = false
    terminalFd uintptr
    scheme     = "http"
  )

  if tlsConfig != nil {
    scheme = "https"
  }

  if in != nil {
    if file, ok := out.(*os.File); ok {
      terminalFd = file.Fd()
      isTerminal = term.IsTerminal(terminalFd)
    }
  }

  if err == nil {
    err = out
  }
  return &DockerCli{
    proto:      proto,
    addr:       addr,
    in:         in,
    out:        out,
    err:        err,
    isTerminal: isTerminal,
    terminalFd: terminalFd,
    tlsConfig:  tlsConfig,
    scheme:     scheme,
  }
}
```

总体而言，创建DockerCli对象较为简单，较为重要的DockerCli的属性有proto：传输协议；addr：host的目标地址，tlsConfig：安全传输层协议的配置。若tlsConfig为不为空，则说明需要使用安全传输层协议，DockerCli对象的scheme设置为“https”，另外还有关于输入，输出以及错误显示的配置，最终返回该对象。

通过调用NewDockerCli函数，程序最终完成了创建Docker Client，并返回main函数继续执行。

4. Docker命令执行

main函数执行到目前为止，有以下内容需要为Docker命令的执行服务：创建完毕的Docker Client，docker命令中的请求参数（经flag解析后存放于flag.Arg()）。也就是说，需要使用Docker Client来分析docker 命令中的请求参数，并最终发送相应请求给Docker Server。

4.1. Docker Client解析请求命令

Docker Client解析请求命令的工作，在Docker命令执行部分第一个完成，直接进入main函数之后的源码部分：

```
if err := cli.Cmd(flag.Args()...); err != nil {
    if sterr, ok := err.(*utils.StatusError); ok {
      if sterr.Status != "" {
        log.Println(sterr.Status)
      }
      os.Exit(sterr.StatusCode)
    }
    log.Fatal(err)
  }
```

查阅以上源码，可以发现，正如之前所说，首先解析存放于flag.Args()中的具体请求参数，执行的函数为cli对象的Cmd函数。进入./docker/api/client/cli.go的Cmd函数：

```
// Cmd executes the specified command
func (cli *DockerCli) Cmd(args ...string) error {
  if len(args) > 0 {
    method, exists := cli.getMethod(args[0])
    if !exists {
      fmt.Println("Error: Command not found:", args[0])
      return cli.CmdHelp(args[1:]...)
    }
    return method(args[1:]...)
  }
  return cli.CmdHelp(args...)
}
```

由代码注释可知，Cmd函数执行具体的指令。源码实现中，首先判断请求参数列表的长度是否大于0，若不是的话，说明没有请求信息，返回docker命令的Help信息；若长度大于0的话，说明有请求信息，则首先通过请求参数列表中的第一个元素args[0]来获取具体的method的方法。如果上述method方法不存在，则返回docker命令的Help信息，若存在的话，调用具体的method方法，参数为args[1]及其之后所有的请求参数。

还是以一个具体的docker命令为例，docker –daemon=false –version=false pull Name。通过以上的分析，可以总结出以下操作流程：

(1) 解析flag参数之后，将docker请求参数” pull”和“Name”存放于flag.Args();

(2) 创建好的Docker Client为cli，cli执行cli.Cmd(flag.Args()...);

在Cmd函数中，通过args[0]也就是” pull”,执行cli.getMethod(args[0])，获取method的名称；

(3) 在getMothod方法中，通过处理最终返回method的值为” CmdPull”；

(4) 最终执行method(args[1:]...)也就是CmdPull(args[1:]...)。

4.2. Docker Client执行请求命令

上一节通过一系列的命令解析，最终找到了具体的命令的执行方法，本节内容主要介绍Docker Client如何通过该执行方法处理并发送请求。

由于不同的请求内容不同，执行流程大致相同，本节依旧以一个例子来阐述其中的流程，例子为：docker pull NAME。

Docker Client在执行以上请求命令的时候，会执行CmdPull函数，传入参数为args[1:]...。源码具体为./docker/api/client/command.go中的CmdPull函数。

以下逐一分析CmdPull的源码实现。

(1) 通过cli包中的Subcmd方法定义一个类型为Flagset的对象cmd。

```
cmd := cli.Subcmd("pull", "NAME[:TAG]", "Pull an image or a repository from the registry")
```

(2) 给cmd对象定义一个类型为String的flag，名为”#t”或”#-tag”，初始值为空。

```
tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image in a repository")
```

(3) 将args参数进行解析，解析过程中，先提取出是否有符合tag这个flag的参数，若有，将其给赋值给tag参数，其余的参数存入cmd.NArg();若无的话，所有的参数存入cmd.NArg()中。

```
if err := cmd.Parse(args); err != nil {
return nil }
```

(4) 判断经过flag解析后的参数列表，若参数列表中参数的个数不为1，则说明需要pull多个image，pull命令不支持，则调用错误处理方法cmd.Usage()，并返回nil。

```
if cmd.NArg() != 1 {
cmd.Usage()
return nil
    }
```

(5) 创建一个map类型的变量v，该变量用于存放pull镜像时所需的url参数；随后将参数列表的第一个值赋给remote变量，并将remote作为键为fromImage的值添加至v；最后若有tag信息的话，将tag信息作为键为”tag”的值添加至v。

```
var (
  v      = url.Values{}
  remote = cmd.Arg(0)
)
v.Set("fromImage", remote)
if *tag == "" {
  v.Set("tag", *tag)
}
```

(6) 通过remote变量解析出镜像所在的host地址，以及镜像的名称。

```
  remote, _ = parsers.ParseRepositoryTag(remote)
    // Resolve the Repository name from fqn to hostname + name
    hostname, _, err := registry.ResolveRepositoryName(remote)
    if err != nil {
      return err
    }
```

(7) 通过cli对象获取与Docker Server通信所需要的认证配置信息。

```
cli.LoadConfigFile()
    // Resolve the Auth config relevant for this server
    authConfig := cli.configFile.ResolveAuthConfig(hostname)
```

(8) 定义一个名为pull的函数，传入的参数类型为registry.AuthConfig，返回类型为error。函数执行块中最主要的内容为：cli.stream(……)部分。该部分具体发起了一个给Docker Server的POST请求，请求的url为"/images/create?"+v.Encode()，请求的认证信息为：map[string][]string{"X-Registry-Auth": registryAuthHeader,}。

```
    pull := func(authConfig registry.AuthConfig) error {
      buf, err := json.Marshal(authConfig)
      if err != nil {
        return err
      }
      registryAuthHeader := []string{
        base64.URLEncoding.EncodeToString(buf),
      }
      return cli.stream("POST", "/images/create?"+v.Encode(), nil, cli.out, map[string][]string{
      "  X-Registry-Auth": registryAuthHeader,
      })
    }
```

(9) 由于上一个步骤只是定义pull函数，这一步骤具体调用执行pull函数，若成功则最终返回，若返回错误，则做相应的错误处理。若返回错误为401，则需要先登录，转至登录环节，完成之后，继续执行pull函数，若完成则最终返回。

```
 if err := pull(authConfig); err != nil {
  if strings.Contains(err.Error(), "Status 401") {
    fmt.Fprintln(cli.out, "\nPlease login prior to pull:")
    if err := cli.CmdLogin(hostname); err != nil {
      return err
    }
        authConfig := cli.configFile.ResolveAuthConfig(hostname)
        return pull(authConfig)
  }
  return err
}
```

以上便是pull请求的全部执行过程，其他请求的执行在流程上也是大同小异。总之，请求执行过程中，大多都是将命令行中关于请求的参数进行初步处理，并添加相应的辅助信息，最终通过指定的协议给Docker Server发送Docker Client和Docker Server约定好的API请求。

5. 总结

本文从源码的角度分析了从docker可执行文件开始，到创建Docker Client，最终发送给Docker Server请求的完整过程。

笔者认为，学习与理解Docker Client相关的源码实现，不仅可以让用户熟练掌握Docker命令的使用，还可以使得用户在特殊情况下有能力修改Docker Client的源码，使其满足自身系统的某些特殊需求，以达到定制Docker Client的目的，最大发挥Docker开放思想的价值。

6. 作者简介

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

7. 参考文献

1. http://www.infoq.com/cn/articles/docker-command-line-quest
2. http://docs.studygolang.com/pkg/
3. http://blog.studygolang.com/2013/02/%E6%A0%87%E5%87%86%E5%BA%93-%E5%91%BD%E4%BB%A4%E8%A1%8C%E5%8F%82%E6%95%B0%E8%A7%A3%E6%9E%90flag/
4. https://docs.docker.com/reference/commandline/cli/

（三）：Docker Daemon启动

- 1 前言
- 2 Docker Daemon简介
- 3 Docker Daemon源码分析内容安排
- 4 Docker Daemon的启动流程
- 5 mainDaemon( )的具体实现
  - 5.0 配置初始化
  - 5.1 flag参数检查
  - 5.2 创建engine对象
  - 5.3 设置engine的信号捕获
  - 5.4 加载builtins
  - 5.5 使用goroutine加载daemon对象并运行
  - 5.5.1 创建daemon对象
  - 5.6 打印Docker版本及驱动信息
  - 5.7 Job之serveapi的创建与运行
- 6 总结
- 7 作者简介
- 8 参考文献

1 前言

Docker诞生以来，便引领了轻量级虚拟化容器领域的技术热潮。在这一潮流下，Google、IBM、Redhat等业界翘楚纷纷加入Docker阵营。虽然目前Docker仍然主要基于Linux平台，但是Microsoft却多次宣布对Docker的支持，从先前宣布的Azure支持Docker与Kubernetes，到如今宣布的下一代Windows Server原生态支持Docker。Microsoft的这一系列举措多少喻示着向Linux世界的妥协，当然这也不得不让世人对Docker的巨大影响力有重新的认识。

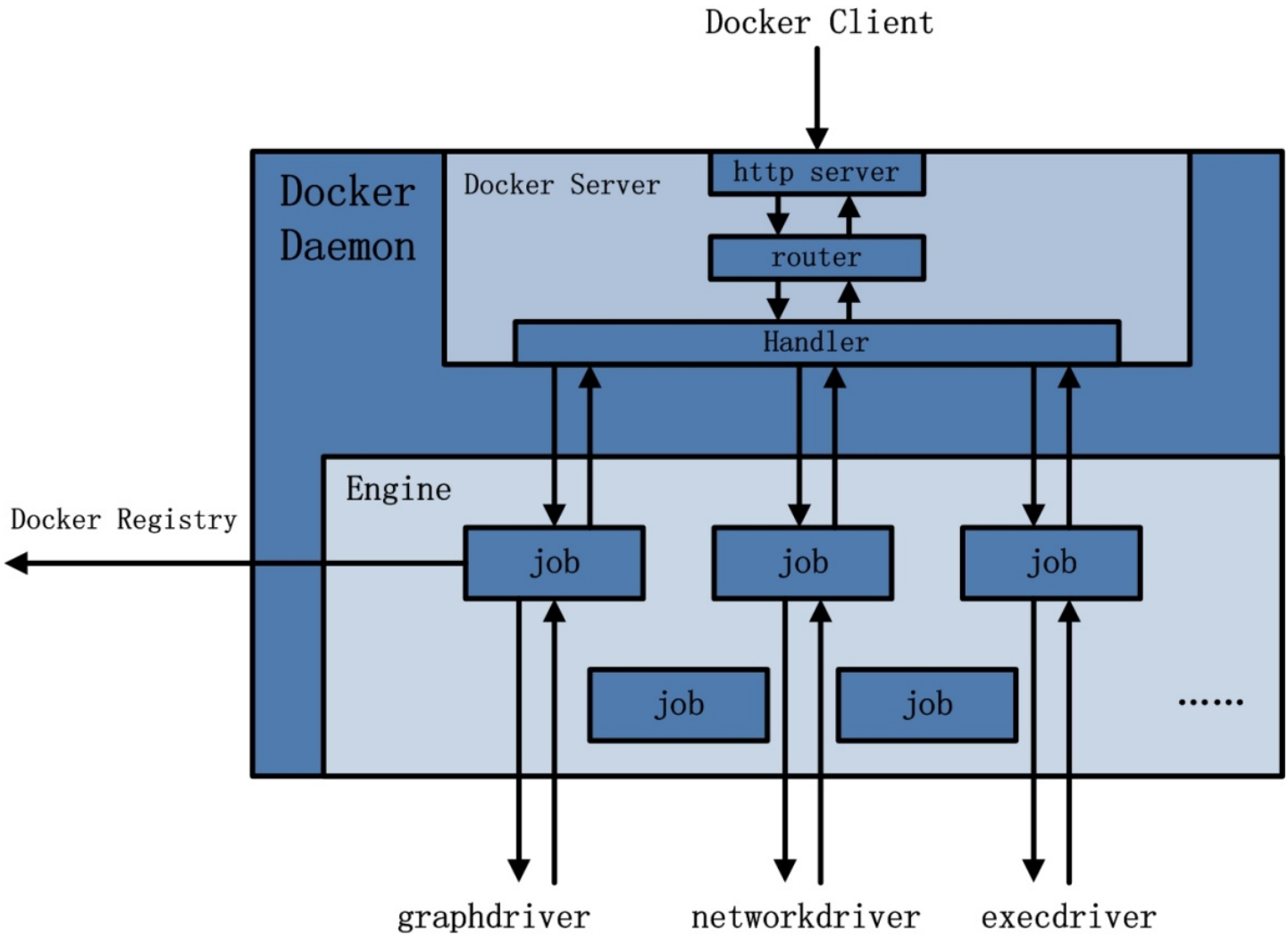
Docker的影响力不言而喻，但如果需要深入学习Docker的内部实现，笔者认为最重要的是理解Docker Daemon。在Docker架构中，Docker Client通过特定的协议与Docker Daemon进行通信，而Docker Daemon主要承载了Docker运行过程中的大部分工作。本文即为《Docker源码分析》系列的第三篇——Docker Daemon篇。

2 Docker Daemon简介

Docker Daemon是Docker架构中运行在后台的守护进程，大致可以分为Docker Server、Engine和Job三部分。Docker Daemon可以认为是通过Docker Server模块接受Docker Client的请求，并在Engine中处理请求，然后根据请求类型，创建出指定的Job并运行，运行过程的作用有以下几种可能：向Docker

Registry获取镜像，通过graphdriver执行容器镜像的本地化操作，通过networkdriver执行容器网络环境的配置，通过execdriver执行容器内部运行的执行工作等。

以下为Docker Daemon的架构示意图：



3 Docker Daemon源码分析内容安排

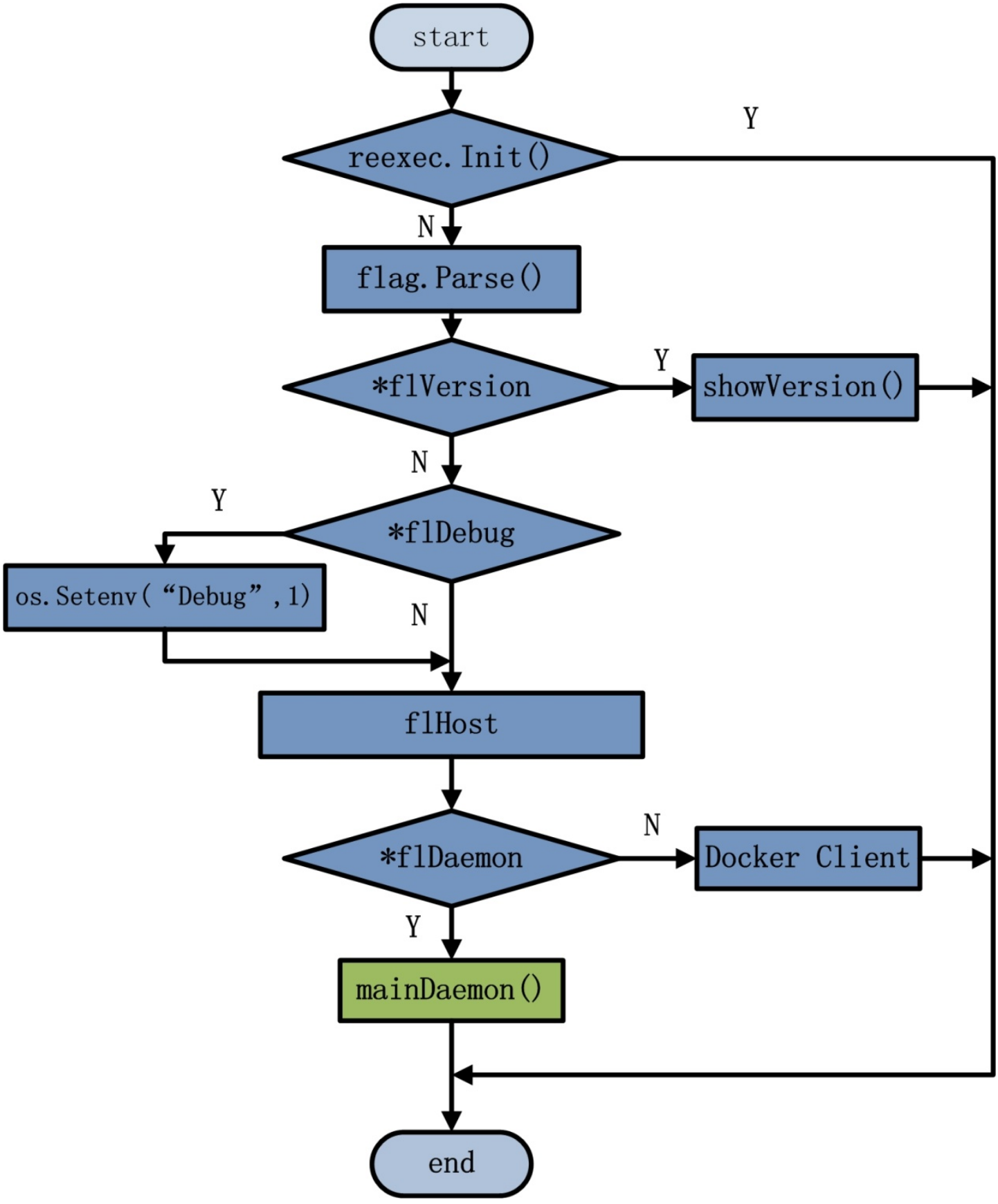
本文从源码的角度，主要分析Docker Daemon的启动流程。由于Docker Daemon和Docker Client的启动流程有很大的相似之处，故在介绍启动流程之后，本文着重分析启动流程中最为重要的环节：创建daemon过程中mainDaemon()的实现。

4 Docker Daemon的启动流程

由于Docker Daemon和Docker Client的启动都是通过可执行文件docker来完成的，因此两者的启动流程非常相似。Docker可执行文件运行时，运行代码通过不同的命令行flag参数，区分两者，并最终运行两者各自相应的部分。

启动Docker Daemon时，一般可以使用以下命令：docker --daemon=true; docker –d; docker –d=true等。接着由docker的main()函数来解析以上命令的相应flag参数，并最终完成Docker Daemon的启动。

首先，附上Docker Daemon的启动流程图：



由于《Docker源码分析》系列之Docker Client篇中，已经涉及了关于Docker中main()函数运行的很多前续工作（可参见Docker Client篇），并且Docker Daemon的启动也会涉及这些工作，故本文略去相同部分，而主要针对后续仅和Docker Daemon相关的内容进行深入分析，即mainDaemon()的具体源码实现。

5 mainDaemon( )的具体实现

通过Docker Daemon的流程图，可以得出一个这样的结论：有关Docker Daemon的所有的工作，都被包含在mainDaemon()方法的实现中。

宏观来讲，mainDaemon()完成创建一个daemon进程，并使其正常运行。

从功能的角度来说，mainDaemon()实现了两部分内容：第一，创建Docker运行环境；第二，服务于Docker Client，接收并处理相应请求。

从实现细节来讲，mainDaemon()的实现过程主要包含以下步骤：

- daemon的配置初始化（这部分在init()函数中实现，即在mainDaemon()运行前就执行，但由于这部分内容和mainDaemon()的运行息息相关，故可认为是mainDaemon()运行的先决条件）；
- 命令行flag参数检查；
- 创建engine对象；
- 设置engine的信号捕获及处理方法；
- 加载builtins；
- 使用goroutine加载daemon对象并运行；
- 打印Docker版本及驱动信息；
- Job之”serveapi”的创建与运行。

下文将一一深入分析以上步骤。

5.0 配置初始化

在mainDaemon()运行之前，关于Docker Daemon所需要的config配置信息均已经初始化完毕。具体实现如下，位于./docker/docker/daemon.go：

```
var (
    daemonCfg = &daemon.Config{}
)
func init() {
    daemonCfg.InstallFlags()
}
```

首先，声明一个为daemon包中Config类型的变量，名为daemonCfg。而Config对象，定义了Docker Daemon所需的配置信息。在Docker Daemon在启动时，daemonCfg变量被传递至Docker Daemon并被使用。

Config对象的定义如下（含部分属性的解释），位于./docker/daemon/config.go：

```
type Config struct {
    Pidfile                 string   //Docker Daemon所属进程的PID文件
    Root                    string   //Docker运行时所使用的root路径
    AutoRestart             bool     //已被启用，转而支持docker run时的重启
    Dns                     []string  //Docker使用的DNS Server地址
    DnsSearch               []string  //Docker使用的指定的DNS查找域名
    Mirrors                 []string  //指定的优先Docker Registry镜像
    EnableIptables          bool     //启用Docker的iptables功能
    EnableIpForward         bool     //启用net.ipv4.ip_forward功能
    EnableIpMasq            bool      //启用IP伪装技术
    DefaultIp               net.IP     //绑定容器端口时使用的默认IP
    BridgeIface             string      //添加容器网络至已有的网桥
    BridgeIP                string     //创建网桥的IP地址
    FixedCIDR               string     //指定IP的IPv4子网，必须被网桥子网包含
    InterContainerCommunication   bool  //是否允许相同host上容器间的通信
    GraphDriver             string      //Docker运行时使用的特定存储驱动
    GraphOptions            []string   //可设置的存储驱动选项
    ExecDriver              string    // Docker运行时使用的特定exec驱动
    Mtu                     int      //设置容器网络的MTU
    DisableNetwork          bool     //有定义，之后未初始化
    EnableSelinuxSupport    bool     //启用SELinux功能的支持
    Context                 map[string][]string   //有定义，之后未初始化
}
```

已经有声明的daemonCfg之后，init()函数实现了daemonCfg变量中各属性的赋值，具体的实现为：daemonCfg.InstallFlags()，位于./docker/daemon/config.go，代码如下：

```
func (config *Config) InstallFlags() {
    flag.StringVar(&config.Pidfile, []string{"p", "-pidfile"}, "/var/run/docker.pid",
 "Path to use for daemon PID file")
    flag.StringVar(&config.Root, []string{"g", "-graph"}, "/var/lib/docker",
"Path to use as the root of the Docker runtime")
    ......
    opts.IPVar(&config.DefaultIp, []string{"#ip", "-ip"}, "0.0.0.0", "Default IP address to
use when binding container ports")
    opts.ListVar(&config.GraphOptions, []string{"-storage-opt"}, "Set storage driver options")
    ......
}
```

在InstallFlags()函数的实现过程中，主要是定义某种类型的flag参数，并将该参数的值绑定在config变量的指定属性上，如：

flag.StringVar(&config.Pidfile, []string{"p", "-pidfile"}, " /var/run/docker.pid", "Path to use for daemon PID file")

以上语句的含义为：

- 定义一个为String类型的flag参数；
- 该flag的名称为”p”或者”-pidfile”；

- 该flag的值为” /var/run/docker.pid”,并将该值绑定在变量config.Pidfile上；
- 该flag的描述信息为"Path to use for daemon PID file"。

至此，关于Docker Daemon所需要的配置信息均声明并初始化完毕。

5.1 flag参数检查

从这一节开始，真正进入Docker Daemon的mainDaemon()运行分析。

第一个步骤即flag参数的检查。具体而言，即当docker命令经过flag参数解析之后，判断剩余的参数是否为0。若为0，则说明Docker Daemon的启动命令无误，正常运行；若不为0，则说明在启动Docker Daemon的时候，传入了多余的参数，此时会输出错误提示，并退出运行程序。具体代码如下：

```
if flag.NArg() != 0 {
    flag.Usage()
    return
}
```

5.2 创建engine对象

在mainDaemon()运行过程中，flag参数检查完毕之后，随即创建engine对象，代码如下：

```
eng := engine.New()
```

Engine是Docker架构中的运行引擎，同时也是Docker运行的核心模块。Engine扮演着Docker container存储仓库的角色，并且通过job的形式来管理这些容器。

在./docker/engine/engine.go中,Engine结构体的定义如下：

```
type Engine struct {
    handlers   map[string]Handler
    catchall   Handler
    hack       Hack // data for temporary hackery (see hack.go)
    id         string
    Stdout     io.Writer
    Stderr     io.Writer
    Stdin      io.Reader
    Logging    bool
    tasks      sync.WaitGroup
    l          sync.RWMutex // lock for shutdown
    shutdown   bool
    onShutdown []func() // shutdown handlers
}
```

其中，Engine结构体中最为重要的即为handlers属性。该handlers属性为map类型，key为string类型，value为Handler类型。其中Handler类型的定义如下：

```
type Handler func(*Job) Status
```

可见，Handler为一个定义的函数。该函数传入的参数为Job指针，返回为Status状态。

介绍完Engine以及Handler，现在真正进入New()函数的实现中：

```
func New() *Engine {
    eng := &Engine{
        handlers: make(map[string]Handler),
        id:       utils.RandomString(),
        Stdout:   os.Stdout,
        Stderr:   os.Stderr,
        Stdin:    os.Stdin,
        Logging:  true,
    }
    eng.Register("commands", func(job *Job) Status {
        for _, name := range eng.commands() {
            job.Printf("%s\n", name)
        }
        return StatusOK
    })
    // Copy existing global handlers
    for k, v := range globalHandlers {
        eng.handlers[k] = v
    }
    return eng
}
```

分析以上代码，可以知道New()函数最终返回一个Engine对象。而在代码实现部分，第一个工作即为创建一个Engine结构体实例eng；第二个工作是向eng对象注册名为commands的Handler，其中Handler为临时定义的函数func(job *Job) Status{ }，该函数的作用是通过job来打印所有已经注册完毕的command名称，最终返回状态StatusOK；第三个工作是：将已定义的变量globalHandlers中的所有的Handler，都复制到eng对象的handlers属性中。最后成功返回eng对象。

5.3 设置engine的信号捕获

回到mainDaemon()函数的运行中，执行后续代码：

```
signal.Trap(eng.Shutdown)
```

该部分代码的作用是：在Docker Daemon的运行中，设置Trap特定信号的处理方法，特定信号有SIGINT，SIGTERM以及SIGQUIT；当程序捕获到SIGINT或者SIGTERM信号时，执行相应的善后操作，最后保证Docker Daemon程序退出。

该部分的代码的实现位于./docker/pkg/signal/trap.go。实现的流程分为以下4个步骤：

- 创建并设置一个channel，用于发送信号通知；

- 定义signals数组变量，初始值为os.SIGINT, os.SIGTERM;若环境变量DEBUG为空的话，则添加os.SIGQUIT至signals数组；
- 通过gosignal.Notify(c, signals...)中Notify函数来实现将接收到的signal信号传递给c。需要注意的是只有signals中被罗列出的信号才会被传递给c，其余信号会被直接忽略；
- 创建一个goroutine来处理具体的signal信号，当信号类型为os.Interrupt或者syscall.SIGTERM时，执行传入Trap函数的具体执行方法，形参为cleanup(),实参为eng.Shutdown。

Shutdown()函数的定义位于./docker/engine/engine.go，主要做的工作是为Docker Daemon的关闭做一些善后工作。

善后工作如下：

- Docker Daemon不再接收任何新的Job；
- Docker Daemon等待所有存活的Job执行完毕；
- Docker Daemon调用所有shutdown的处理方法；
- 当所有的handler执行完毕，或者15秒之后，Shutdown()函数返回。

由于在signal.Trap(eng.Shutdown)函数的具体实现中执行eng.Shutdown，在执行完eng.Shutdown之后，随即执行os.Exit(0)，完成当前程序的立即退出。

5.4 加载builtins

为eng设置完Trap特定信号的处理方法之后，Docker Daemon实现了builtins的加载。代码实现如下：

```
if err := builtins.Register(eng); err != nil {
    log.Fatal(err)
}
```

加载builtins的主要工作是为：为engine注册多个Handler，以便后续在执行相应任务时，运行指定的Handler。这些Handler包括：网络初始化、web API服务、事件查询、版本查看、Docker Registry验证与搜索。代码实现位于./docker/builtins/builtins.go,如下：

```
func Register(eng *engine.Engine) error {
    if err := daemon(eng); err != nil {
        return err
    }
    if err := remote(eng); err != nil {
        return err
    }
    if err := events.New().Install(eng); err != nil {
        return err
    }
    if err := eng.Register("version", dockerVersion); err != nil {
        return err
    }
    return registry.NewService().Install(eng)
}
```

以下分析实现过程中最为主要的5个部分：daemon(eng)、remote(eng)、events.New().Install(eng)、eng.Register(“version”,dockerVersion)以及registry.NewService().Install(eng)。

5.4.1 注册初始化网络驱动的Handler

daemon(eng)的实现过程，主要为eng对象注册了一个key为”init_networkdriver”的Handler，该Handler的值为bridge.InitDriver函数，代码如下：

```
func daemon(eng *engine.Engine) error {
    return eng.Register("init_networkdriver", bridge.InitDriver)
}
```

需要注意的是，向eng对象注册Handler，并不代表Handler的值函数会被直接运行，如bridge.InitDriver，并不会直接运行，而是将bridge.InitDriver的函数入口，写入eng的handlers属性中。

Bridge.InitDriver的具体实现位于./docker/daemon/networkdriver/bridge/driver.go ，主要作用为：

- 获取为Docker服务的网络设备的地址；
- 创建指定IP地址的网桥；
- 配置网络iptables规则；
- 另外还为eng对象注册了多个Handler,如 ” allocate_interface” ， ” release_interface” ， ” allocate_port” ，” link” 。

5.4.2 注册API服务的Handler

remote(eng)的实现过程，主要为eng对象注册了两个Handler，分别为” serveapi” 与” acceptconnections” 。代码实现如下：

```
func remote(eng *engine.Engine) error {
    if err := eng.Register("serveapi", apiserver.ServeApi); err != nil {
        return err
    }
    return eng.Register("acceptconnections", apiserver.AcceptConnections)
}
```

注册的两个Handler名称分别为”serveapi”与”acceptconnections”,相应的执行方法分别为apiserver.ServeApi与apiserver.AcceptConnections，具体实现位于./docker/api/server/server.go。其中，ServeApi执行时，通过循环多种协议，创建出goroutine来配置指定的http.Server，最终为不同的协议请求服务；而AcceptConnections的实现主要是为了通知init守护进程，Docker Daemon已经启动完毕，可以让Docker Daemon进程接受请求。

5.4.3 注册events事件的Handler

events.New().Install(eng)的实现过程，为Docker注册了多个event事件，功能是给Docker用户提供API，使得用户可以通过这些API查看Docker内部的events信息，log信息以及subscribers_count信息。具体的代码位于./docker/events/events.go，如下：

```
func (e *Events) Install(eng *engine.Engine) error {
    jobs := map[string]engine.Handler{
        "events":            e.Get,
        "log":               e.Log,
        "subscribers_count": e.SubscribersCount,
    }
    for name, job := range jobs {
        if err := eng.Register(name, job); err != nil {
            return err
        }
    }
    return nil
}
```

5.4.4 注册版本的Handler

eng.Register(“version”,dockerVersion)的实现过程，向eng对象注册key为”version”，value为”dockerVersion”执行方法的Handler，dockerVersion的执行过程中，会向名为version的job的标准输出中写入Docker的版本，Docker API的版本，git版本，Go语言运行时版本以及操作系统等版本信息。dockerVersion的具体实现如下：

```
func dockerVersion(job *engine.Job) engine.Status {
    v := &engine.Env{}
    v.SetJson("Version", dockerversion.VERSION)
    v.SetJson("ApiVersion", api.APIVERSION)
    v.Set("GitCommit", dockerversion.GITCOMMIT)
    v.Set("GoVersion", runtime.Version())
    v.Set("Os", runtime.GOOS)
    v.Set("Arch", runtime.GOARCH)
    if kernelVersion, err := kernel.GetKernelVersion(); err == nil {
        v.Set("KernelVersion", kernelVersion.String())
    }
    if _, err := v.WriteTo(job.Stdout); err != nil {
        return job.Error(err)
    }
    return engine.StatusOK
}
```

5.4.5 注册registry的Handler

registry.NewService().Install(eng)的实现过程位于./docker/registry/service.go，在eng对象对外暴露的API信息中添加docker registry的信息。当registry.NewService()成功被Install安装完毕的话，则有两个调用能够被eng使用：” auth” ，向公有registry进行认证；” search” ，在公有registry上搜索指定的镜像。

Install的具体实现如下：

```
func (s *Service) Install(eng *engine.Engine) error {
    eng.Register("auth", s.Auth)
    eng.Register("search", s.Search)
    return nil
}
```

至此，所有builtins的加载全部完成，实现了向eng对象注册特定的Handler。

5.5 使用goroutine加载daemon对象并运行

执行完builtins的加载，回到mainDaemon()的执行，通过一个goroutine来加载daemon对象并开始运行。这一环节的执行，主要包含三个步骤：

- 通过init函数中初始化的daemonCfg与eng对象来创建一个daemon对象d；
- 通过daemon对象的Install函数，向eng对象中注册众多的Handler；
- 在Docker Daemon启动完毕之后，运行名为” acceptconnections” 的job，主要工作为向init守护进程发送” READY=1” 信号，以便开始正常接受请求。

代码实现如下：

```
go func() {
    d, err := daemon.MainDaemon(daemonCfg, eng)
    if err != nil {
        log.Fatal(err)
    }
    if err := d.Install(eng); err != nil {
        log.Fatal(err)
    }
    if err := eng.Job("acceptconnections").Run(); err != nil {
        log.Fatal(err)
    }
}()
```

以下分别分析三个步骤所做的工作。

5.5.1 创建daemon对象

daemon.MainDaemon(daemonCfg, eng)是创建daemon对象d的核心部分。主要作用为初始化Docker Daemon的基本环境，如处理config参数，验证系统支持度，配置Docker工作目录，设置与加载多种driver，创建graph环境等，验证DNS配置等。

由于daemon.MainDaemon(daemonCfg, eng)是加载Docker Daemon的核心部分，且篇幅过长，故安排《Docker源码分析》系列的第四篇专文分析这部分。

5.5.2 通过daemon对象为engine注册Handler

当创建完daemon对象，goroutine执行d.Install(eng)，具体实现位于./docker/daemon/daemon.go:

```
func (daemon *Daemon) Install(eng *engine.Engine) error {
    for name, method := range map[string]engine.Handler{
        "attach":            daemon.ContainerAttach,
        ......
        "image_delete":      daemon.ImageDelete,
    } {
        if err := eng.Register(name, method); err != nil {
            return err
        }
    }
    if err := daemon.Repositories().Install(eng); err != nil {
        return err
    }
    eng.Hack_SetGlobalVar("httpapi.daemon", daemon)
    return nil
}
```

以上代码的实现分为三部分：

- 向eng对象中注册众多的Handler对象；
- daemon.Repositories().Install(eng)实现了向eng对象注册多个与image相关的Handler，Install的实

现位于./docker/graph/service.go；

- eng.Hack_SetGlobalVar("httpapi.daemon", daemon)实现向eng对象中map类型的hack对象中添加一条记录，key为”httpapi.daemon”，value为daemon。

5.5.3 运行acceptconnections的job

在goroutine内部最后运行名为”acceptconnections”的job，主要作用是通知init守护进程，Docker Daemon可以开始接受请求了。

这是源码分析系列中第一次涉及具体Job的运行，以下简单分析”acceptconnections”这个job的运行。

可以看到首先执行eng.Job("acceptconnections")，返回一个Job，随后再执行eng.Job("acceptconnections").Run()，也就是该执行Job的run函数。

eng.Job(“acceptconnections”)的实现位于./docker/engine/engine.go，如下：

```
func (eng *Engine) Job(name string, args ...string) *Job {
    job := &Job{
        Eng:    eng,
        Name:   name,
        Args:   args,
        Stdin:  NewInput(),
        Stdout: NewOutput(),
        Stderr: NewOutput(),
        env:    &Env{},
    }
    if eng.Logging {
        job.Stderr.Add(utils.NopWriteCloser(eng.Stderr))
    }
    if handler, exists := eng.handlers[name]; exists {
        job.handler = handler
    } else if eng.catchall != nil && name != "" {
        job.handler = eng.catchall
    }
    return job
}
```

由以上代码可知，首先创建一个类型为Job的job对象，该对象中Eng属性为函数的调用者eng，Name属性为”acceptconnections”，没有参数传入。另外在eng对象所有的handlers属性中寻找键为”acceptconnections”记录的值，由于在加载builtins操作中的remote(eng)中已经向eng注册过这样的一条记录，key为”acceptconnections”，value为apiserver.AcceptConnections。因此job对象的handler为apiserver.AcceptConnections。最后返回已经初始化完毕的对象job。

创建完job对象之后，随即执行该job对象的run()函数。Run()函数的实现位于./docker/engine/job.go，该函数执行指定的job，并在job执行完成前一直阻塞。对于名为”acceptconnections”的job对象，运行代码为job.status = job.handler(job)，由于job.handler值为apiserver.AcceptConnections，故真正执行的是job.status = apiserver.AcceptConnections(job)。

进入AcceptConnections的具体实现，位于./docker/api/server/server.go,如下：

```
func AcceptConnections(job *engine.Job) engine.Status {
    // Tell the init daemon we are accepting requests
    go  systemd.SdNotify("READY=1")
    if activationLock != nil {
        close(activationLock)
    }
    return engine.StatusOK
}
```

重点为go systemd.SdNotify("READY=1")的实现，位于./docker/pkg/system/sd_notify.go，主要作用是通知init守护进程Docker Daemon的启动已经全部完成，潜在的功能是使得Docker Daemon开始接受Docker Client发送来的API请求。

至此，已经完成通过goroutine来加载daemon对象并运行。

5.6 打印Docker版本及驱动信息

回到mainDaemon()的运行流程中，在goroutine的执行之时，mainDaemon()函数内部其它代码也会并发执行。

第一个执行的即为显示docker的版本信息，以及ExecDriver和GraphDriver这两个驱动的具体信息，代码如下：

```
log.Printf("docker daemon: %s %s; execdriver: %s; graphdriver: %s",
    dockerversion.VERSION,
    dockerversion.GITCOMMIT,
    daemonCfg.ExecDriver,
    daemonCfg.GraphDriver,
)
```

5.7 Job之serveapi的创建与运行

打印部分Docker具体信息之后，Docker Daemon立即创建并运行名为” serveapi”的job，主要作用为让Docker Daemon提供API访问服务。实现代码位于./docker/docker/daemon.go#L66，如下：

```
job := eng.Job("serveapi", flHosts...)
job.SetenvBool("Logging", true)
job.SetenvBool("EnableCors", *flEnableCors)
job.Setenv("Version", dockerversion.VERSION)
job.Setenv("SocketGroup", *flSocketGroup)

job.SetenvBool("Tls", *flTls)
job.SetenvBool("TlsVerify", *flTlsVerify)
job.Setenv("TlsCa", *flCa)
job.Setenv("TlsCert", *flCert)
job.Setenv("TlsKey", *flKey)
job.SetenvBool("BufferRequests", true)
if err := job.Run(); err != nil {
    log.Fatal(err)
}
```

实现过程中，首先创建一个名为”serveapi”的job，并将flHosts的值赋给job.Args。flHost的作用主要是为Docker Daemon提供使用的协议与监听的地址。随后，Docker Daemon为该job设置了众多的环境变量，如安全传输层协议的环境变量等。最后通过job.Run()运行该serveapi的job。

由于在eng中key为”serveapi”的handler，value为apiserver.ServeApi，故该job运行时，执行apiserver.ServeApi函数，位于./docker/api/server/server.go。ServeApi函数的作用主要是对于用户定义的所有支持协议，Docker Daemon均创建一个goroutine来启动相应的http.Server，分别为不同的协议服务。

由于创建并启动http.Server为Docker架构中有关Docker Server的重要内容，《Docker源码分析》系列会在第五篇专文进行分析。

至此，可以认为Docker Daemon已经完成了serveapi这个job的初始化工作。一旦acceptconnections这个job运行完毕，则会通知init进程Docker Daemon启动完毕，可以开始提供API服务。

6 总结

本文从源码的角度分析了Docker Daemon的启动，着重分析了mainDaemon()的实现。

Docker Daemon作为Docker架构中的主干部分，负责了Docker内部几乎所有操作的管理。学习Docker Daemon的具体实现，可以对Docker架构有一个较为全面的认识。总结而言，Docker的运行，载体为daemon，调度管理由engine，任务执行靠job。

7 作者简介

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

8 参考文献

1. http://www.infoq.com/cn/news/2014/10/windows-server-docker
2. http://www.freedesktop.org/software/systemd/man/sd_notify.html
3. http://www.ibm.com/developerworks/cn/linux/1407_liuming_init3/index.html
4. http://docs.studygolang.com/pkg/os/
5. https://docs.docker.com/reference/commandline/cli/

（四）：Docker Daemon之NewDaemon实现

- 1. 前言
- 2. NewDaemon作用简介
- 3. NewDaemon源码分析内容安排
- 4. NewDaemon具体实现
  - 4.1. 应用配置信息
  - 4.2. 检测系统支持及用户权限
  - 4.3. 配置工作路径
  - 4.4. 加载并配置graphdriver
  - 4.5. 创建Docker Daemon网络环境
  - 4.6. 创建graphdb并初始化
  - 4.7. 创建execdriver
  - 4.8. 创建daemon实例
  - 4.9. 检测DNS配置
  - 4.10. 启动时加载已有Docker containers
  - 4.11. 设置shutdown的处理方法
  - 4.12. 返回daemon对象实例
- 5. 总结
- 6. 作者简介
- 7. 参考文献

1. 前言

Docker的生态系统日趋完善，开发者群体也在日趋庞大，这让业界对Docker持续抱有极其乐观的态度。如今，对于广大开发者而言，使用Docker这项技术已然不是门槛，享受Docker带来的技术福利也不再是困难。然而，如何探寻Docker适应的场景，如何发展Docker周边的技术，以及如何弥合Docker新技术与传统物理机或VM技术的鸿沟，已经占据Docker研究者们的思考与实践。

本文为《Docker源码分析》第四篇——Docker Daemon之NewDaemon实现，力求帮助广大Docker爱好者更多得理解Docker 的核心——Docker Daemon的实现。

2. NewDaemon作用简介

在Docker架构中有很多重要的概念，如：
graph，graphdriver，execdriver，networkdriver，volumes，Docker containers等。Docker在实现过程中，需要将以上实体进行统一化管理，而Docker Daemon中的daemon实例就是设计用来完成这一任务的实体。

从源码的角度，NewDaemon函数的执行完成了Docker Daemon创建并加载daemon的任务，最终实现统一管理Docker Daemon的资源。

3. NewDaemon源码分析内容安排

本文从源码角度，分析Docker Daemon加载过程中NewDaemon的实现，整个分析过程如下图：

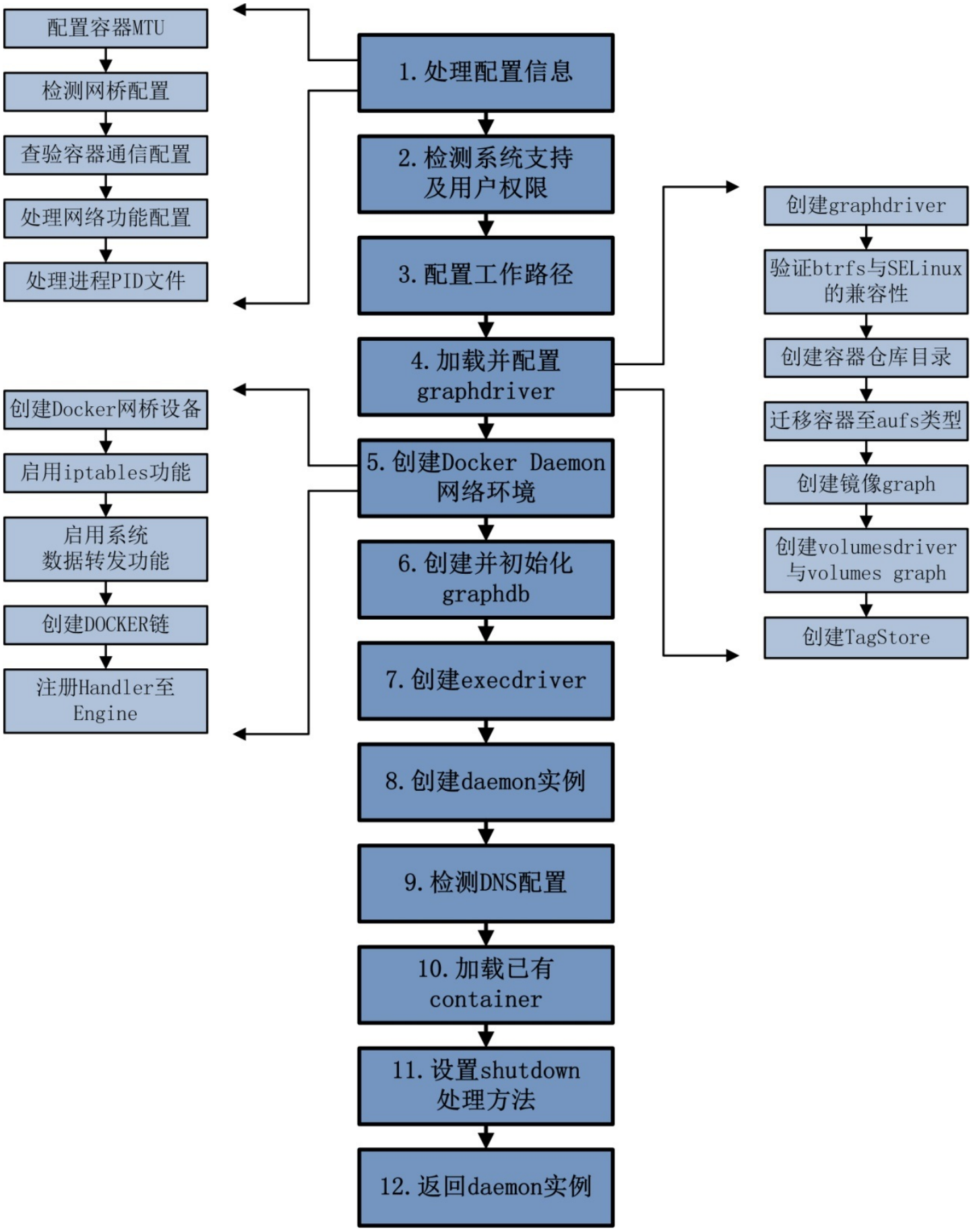


图3.1 Docker Daemon中NewDaemon执行流程图

由上图可见，Docker Daemon中NewDaemon的执行流程主要包含12个独立的步骤：处理配置信息、检测系统支持及用户权限、配置工作路径、加载并配置graphdriver、创建Docker Daemon网络环境、创建并初始化graphdb、创建execdriver、创建daemon实例、检测DNS配置、加载已有container、设置

shutdown处理方法、以及返回daemon实例。

下文会在NewDaemon的具体实现中，以12节分别分析以上内容。

4. NewDaemon具体实现

在《Docker源码分析》系列第三篇中，有一个重要的环节：使用goroutine加载daemon对象并运行。在加载并运行daemon对象时，所做的第一个工作即为：

```
d, err := daemon.NewDaemon(daemonCfg, eng)
```

该部分代码分析如下：

- 函数名：NewDaemon；
- 函数调用具体实现所处的包位置：./docker/daemon；
- 函数具体实现源文件：./docker/daemon/daemon.go；
- 函数传入实参：daemonCfg，定义了Docker Daemon运行过程中所需的众多配置信息；eng，在mainDaemon中创建的Engine对象实例；
- 函数返回类型：d，具体的Daemon对象实例；err，错误状态。

进入./docker/daemon/daemon.go中NewDaemon的具体实现，代码如下：

```
func NewDaemon(config *Config, eng *engine.Engine) (*Daemon, error) {
    daemon, err := NewDaemonFromDirectory(config, eng)
    if err != nil {
        return nil, err
    }
    return daemon, nil
}
```

可见，在实现NewDaemon的过程中，通过NewDaemonFromDirectory函数来实现创建Daemon的运行环境。该函数的实现，传入参数以及返回类型与NewDaemon函数相同。下文将大篇幅分析NewDaemonFromDirectory的实现细节。

4.1. 应用配置信息

在NewDaemonFromDirectory的实现过程中，第一个工作是：如何应用传入的配置信息。这部分配置信息服务于Docker Daemon的运行，并在Docker Daemon启动初期就初始化完毕。配置信息的主要功能是：供用户自由配置Docker的可选功能，使得Docker的运行更贴近用户期待的运行场景。

配置信息的处理包含4部分：

- 配置Docker容器的MTU；
- 检测网桥配置信息；
- 查验容器通信配置；

- 处理PID文件配置。

4.1.1. 配置Docker容器的MTU

config信息中的Mtu应用于容器网络的最大传输单元（MTU）特性。有关MTU的源码如下：

```
if config.Mtu == 0 {
config.Mtu = GetDefaultNetworkMtu()
```

可见，若config信息中Mtu的值为0的话，则通过GetDefaultNetworkMtu函数将Mtu设定为默认的值；否则，采用config中的Mtu值。由于在默认的配置文件./docker/daemon/config.go（下文简称为默认配置文件）中，初始化时Mtu属性值为0，故执行GetDefaultNetworkMtu。

GetDefaultNetworkMtu函数的具体实现位于./docker/daemon/config.go:

```
func GetDefaultNetworkMtu() int {
    if iface, err := networkdriver.GetDefaultRouteIface(); err == nil {
        return iface.MTU
    }
    return defaultNetworkMtu
}
```

GetDefaultNetworkMtu的实现中，通过networkdriver包的GetDefaultRouteIface方法获取具体的网络设备，若该网络设备存在，则返回该网络设备的MTU属性值；否则的话，返回默认的MTU值defaultNetworkMtu，值为1500。

4.1.2. 检测网桥配置信息

处理完config中的Mtu属性之后，马上检测config中BridgeIface和BridgeIP这两个信息。BridgeIface和BridgeIP的作用是为创建网桥的任务”init_networkdriver”提供参数。代码如下：

```
if config.BridgeIface != "" && config.BridgeIP != "" {
    return nil, fmt.Errorf("You specified -b & --bip, mutually exclusive options.
Please specify only one.")
}
```

以上代码的含义为：若config中BridgeIface和BridgeIP两个属性均不为空，则返回nil对象，并返回错误信息，错误信息内容为：用户同时指定了BridgeIface和BridgeIP，这两个属性属于互斥类型，只能至多指定其中之一。而在默认配置文件中，BridgeIface和BridgeIP均为空。

4.1.3. 查验容器通信配置

检测容器的通信配置，主要是针对config中的EnableIptables和InterContainerCommunication这两个属性。EnableIptables属性的作用是启用Docker对iptables规则的添加功能；InterContainerCommunication的作用是启用Docker container之间互相通信的功能。代码如下：

```
if !config.EnableIptables && !config.InterContainerCommunication {
    return nil, fmt.Errorf("You specified --iptables=false with --icc=
false. ICC uses iptables to function. Please set --icc or --iptables to true.")
}
```

代码含义为：若EnableIptables和InterContainerCommunication两个属性的值均为false，则返回nil对象以及错误信息。其中错误信息为：用户将以上两属性均置为false，container间通信需要iptables的支持，需设置至少其中之一为true。而在默认配置文件中，这两个属性的值均为true。

4.1.4. 处理网络功能配置

接着，处理config中的DisableNetwork属性，以备后续在创建并执行创建Docker Daemon网络环境时使用，即在名为”init_networkdriver”的job创建并运行中体现。

```
config.DisableNetwork = config.BridgeIface == DisableNetworkBridge
```

由于config中的BridgeIface属性值为空，另外DisableNetworkBridge的值为字符串”none”，因此最终config中DisableNetwork的值为false。后续名为”init_networkdriver”的job在执行过程中需要使用该属性。

4.1.5. 处理PID文件配置

处理PID文件配置，主要工作是：为Docker Daemon进程运行时的PID号创建一个PID文件，文件的路径即为config中的Pidfile属性。并且为Docker Daemon的shutdown操作添加一个删除该Pidfile的函数，以便在Docker Daemon退出的时候，可以在第一时间删除该Pidfile。处理PID文件配置信息的代码实现如下：

```
if config.Pidfile != "" {
    if err := utils.CreatePidFile(config.Pidfile); err != nil {
        return nil, err
    }
    eng.OnShutdown(func() {
        utils.RemovePidFile(config.Pidfile)
    })
}
```

代码执行过程中，首先检测config中的Pidfile属性是否为空，若为空，则跳过代码块继续执行；若不为空，则首先在文件系统中创建具体的Pidfile，然后向eng的onShutdown属性添加一个处理函数，函数具体完成的工作为utils.RemovePidFile(config.Pidfile)，即在Docker Daemon进行shutdown操作的时候，删除Pidfile文件。在默认配置文件中，Pidfile文件的初始值为” /var/run/docker.pid”。

以上便是关于配置信息处理的分析。

4.2. 检测系统支持及用户权限

初步处理完Docker的配置信息之后，Docker对自身运行的环境进行了一系列的检测，主要包括三个方面：

- 操作系统类型对Docker Daemon的支持；
- 用户权限的级别；
- 内核版本与处理器的支持。

系统支持与用户权限检测的实现较为简单，实现代码如下：

```
if runtime.GOOS != "linux" {
    log.Fatalf("The Docker daemon is only supported on linux")
}
if os.Geteuid() != 0 {
    log.Fatalf("The Docker daemon needs to be run as root")
}
if err := checkKernelAndArch(); err != nil {
    log.Fatalf(err.Error())
}
```

首先，通过runtime.GOOS，检测操作系统的类型。runtime.GOOS返回运行程序所在操作系统的类型，可以是Linux，Darwin，FreeBSD等。结合具体代码，可以发现，若操作系统不为Linux的话，将报出Fatal错误日志，内容为“Docker Daemon只能支持Linux操作系统”。

接着，通过os.Geteuid()，检测程序用户是否拥有足够权限。os.Geteuid()返回调用者所在组的group id。结合具体代码，也就是说，若返回不为0，则说明不是以root用户的身份运行，报出Fatal日志。

最后，通过checkKernelAndArch()，检测内核的版本以及主机处理器类型。checkKernelAndArch()的实现同样位于./docker/daemon/daemon.go。实现过程中，第一个工作是：检测程序运行所在的处理器架构是否为“amd64”，而目前Docker运行时只能支持amd64的处理器架构。第二个工作是：检测Linux内核版本是否满足要求，而目前Docker Daemon运行所需的内核版本若过低，则必须升级至3.8.0。

4.3. 配置工作路径

配置Docker Daemon的工作路径，主要是创建Docker Daemon运行中所在的工作目录。实现过程中，通过config中的Root属性来完成。在默认配置文件中，Root属性的值为”/var/lib/docker”。

在配置工作路径的代码实现中，步骤如下：

(1) 使用规范路径创建一个TempDir，路径名为tmp；

(2) 通过tmp，创建一个指向tmp的文件符号连接realTmp；

(3) 使用realTemp的值，创建并赋值给环境变量TMPDIR；

(4) 处理config的属性EnableSelinuxSupport；

(5) 将realRoot重新赋值于config.Root，并创建Docker Daemon的工作根目录。

4.4. 加载并配置graphdriver

加载并配置存储驱动graphdriver，目的在于：使得Docker Daemon创建Docker镜像管理所需的驱动环境。Graphdriver用于完成Docker容器镜像的管理，包括存储与获取。

4.4.1. 创建graphdriver

这部分内容的源码位于./docker/daemon/daemon.go#L743-L790，具体细节分析如下：

```
graphdriver.DefaultDriver = config.GraphDriver
driver, err := graphdriver.New(config.Root, config.GraphOptions)
```

首先，为graphdriver包中的DefaultDriver对象赋值，值为config中的GraphDriver属性，在默认配置文件中，GraphDriver属性的值为空；同样的，属性GraphOptions也为空。然后通过graphDriver中的new函数实现加载graph的存储驱动。

创建具体的graphdriver是相当重要的一个环节，实现细节由graphdriver包中的New函数来完成。进入./docker/daemon/graphdriver/driver.go中，实现步骤如下：

第一，遍历数组选择graphdriver，数组内容为os.Getenv(“DOCKER_DRIVER”)和DefaultDriver。若不为空，则通过GetDriver函数直接返回相应的Driver对象实例，若均为空，则继续往下执行。这部分内容的作用是：让graphdriver的加载，首先满足用户的自定义选择，然后满足默认值。代码如下：

```
for _, name := range []string{os.Getenv("DOCKER_DRIVER"), DefaultDriver} {
    if name != "" {
        return GetDriver(name, root, options)
    }
}
```

第二，遍历优先级数组选择graphdriver，优先级数组的内容为依次为”aufs”，”brtfs”，”devicemapper”和”vfs”。若依次验证时，GetDriver成功，则直接返回相应的Driver对象实例，若均不成功，则继续往下执行。这部分内容的作用是：在没有指定以及默认的Driver时，从优先级数组中选择Driver，目前优先级最高的为“aufs”。代码如下：

```
for _, name := range priority {
    driver, err = GetDriver(name, root, options)
    if err != nil {
        if err == ErrNotSupported || err == ErrPrerequisites || err == ErrIncompatibleFS {
            continue
        }
        return nil, err
    }
    return driver, nil
}
```

第三，从已经注册的drivers数组中选择graphdriver。

在”aufs”，”btrfs”，”devicemapper”和”vfs”四个不同类型driver的init函数中，它们均向graphdriver的drivers数组注册了相应的初始化方法。分别位于./docker/daemon/graphdriver/aufs/aufs.go，以及其他三类driver的相应位置。这部分内容的作用是：在没有优先级drivers数组的时候，同样可以通过注册的driver来选择具体的graphdriver。

4.4.2. 验证btrfs与SELinux的兼容性

由于目前在btrfs文件系统上运行的Docker不兼容SELinux，因此当config中配置信息需要启用SELinux的支持并且driver的类型为btrfs时，返回nil对象，并报出Fatal日志。代码实现如下：

```
// As Docker on btrfs and SELinux are incompatible at present, error on both being enabled
if config.EnableSelinuxSupport && driver.String() == "btrfs" {
return nil, fmt.Errorf("SELinux is not supported with the BTRFS graph driver!")
}
```

4.4.3. 创建容器仓库目录

Docker Daemon在创建Docker容器之后，需要将容器放置于某个仓库目录下，统一管理。而这个目录即为daemonRepo，值为：/var/lib/docker/containers，并通过daemonRepo创建相应的目录。代码实现如下：

```
daemonRepo := path.Join(config.Root, "containers")
if err := os.MkdirAll(daemonRepo, 0700); err != nil && !os.IsExist(err) {
    return nil, err
}
```

4.4.4. 迁移容器至aufs类型

当graphdriver的类型为aufs时，需要将现有graph的所有内容都迁移至aufs类型；若不为aufs，则继续往下执行。实现代码如下：

```
if err = migrateIfAufs(driver, config.Root); err != nil {
return nil, err
}
```

这部分的迁移内容主要包括Repositories，Images以及Containers，具体实现位于./docker/daemon/graphdriver/aufs/migrate.go。

```
func (a *Driver) Migrate(pth string, setupInit func(p string) error) error {
    if pathExists(path.Join(pth, "graph")) {
        if err := a.migrateRepositories(pth); err != nil {
            return err
        }
        if err := a.migrateImages(path.Join(pth, "graph")); err != nil {
            return err
        }
        return a.migrateContainers(path.Join(pth, "containers"), setupInit)
    }
    return nil
}
```

migrate repositories的功能是：在Docker Daemon的root工作目录下创建repositories-aufs的文件，存储所有与images相关的基本信息。

migrate images的主要功能是：将原有的image镜像都迁移至aufs driver能识别并使用的类型，包括aufs所规定的layers，diff与mnt目录内容。

migrate container的主要功能是：将container内部的环境使用aufs driver来进行配置，包括，创建container内部的初始层（init layer），以及创建原先container内部的其他layers。

4.4.5. 创建镜像graph

创建镜像graph的主要工作是：在文件系统中指定的root目录下，实例化一个全新的graph对象，作用为：存储所有标记的文件系统镜像，并记录镜像之间的关系。实现代码如下：

```
g, err := graph.NewGraph(path.Join(config.Root, "graph"), driver)
```

NewGraph的具体实现位于./docker/graph/graph.go，实现过程中返回的对象为Graph类型，定义如下：

```
type Graph struct {
    Root    string
    idIndex *truncindex.TruncIndex
    driver  graphdriver.Driver
}
```

其中Root表示graph的工作根目录，一般为”/var/lib/docker/graph”；idIndex使得检索字符串标识符时，允许使用任意一个该字符串唯一的前缀，在这里idIndex用于通过简短有效的字符串前缀检索镜像与容器的ID；最后driver表示具体的graphdriver类型。

4.4.6. 创建volumesdriver以及volumes graph

在Docker中volume的概念是：可以从Docker宿主机上挂载到Docker容器内部的特定目录。一个volume可以被多个Docker容器挂载，从而Docker容器可以实现互相共享数据等。在实现volumes时，Docker需

要使用driver来管理它，又由于volumes的管理不会像容器文件系统管理那么复杂，故Docker采用vfs驱动实现volumes的管理。代码实现如下：

```
volumesDriver, err := graphdriver.GetDriver("vfs", config.Root, config.GraphOptions)
volumes, err := graph.NewGraph(path.Join(config.Root, "volumes"), volumesDriver)
```

主要完成工作为：使用vfs创建volumesDriver；创建相应的volumes目录，并返回volumes graph对象。

4.4.7. 创建TagStore

TagStore主要是用于存储镜像的仓库列表（repository list）。代码如下：

```
repositories, err := graph.NewTagStore(path.Join(config.Root, "repositories-"+driver.String()), g)
```

NewTagStore位于./docker/graph/tags.go，TagStore的定义如下：

```
type TagStore struct {
    path         string
    graph        *Graph
    Repositories   map[string]Repository
    sync.Mutex
    pullingPool    map[string]chan struct{}
    pushingPool    map[string]chan struct{}
}
```

需要阐述的是TagStore类型中的多个属性的含义：

- path：TagStore中记录镜像仓库的文件所在路径；
- graph：相应的Graph实例对象；
- Repositories：记录具体的镜像仓库的map数据结构；
- sync.Mutex：TagStore的互斥锁
- pullingPool ：记录池，记录有哪些镜像正在被下载，若某一个镜像正在被下载，则驳回其他Docker Client发起下载该镜像的请求；
- pushingPool：记录池，记录有哪些镜像正在被上传，若某一个镜像正在被上传，则驳回其他Docker Client发起上传该镜像的请求；

4.5. 创建Docker Daemon网络环境

创建Docker Daemon运行环境的时候，创建网络环境是极为重要的一个部分，这不仅关系着容器对外的通信，同样也关系着容器间的通信。

在创建网络时，Docker Daemon是通过运行名为”init_networkdriver”的job来完成的。代码如下：

```
if !config.DisableNetwork {
    job := eng.Job("init_networkdriver")

    job.SetenvBool("EnableIptables", config.EnableIptables)
    job.SetenvBool("InterContainerCommunication", config.InterContainerCommunication)
    job.SetenvBool("EnableIpForward", config.EnableIpForward)
    job.Setenv("BridgeIface", config.BridgeIface)
    job.Setenv("BridgeIP", config.BridgeIP)
    job.Setenv("DefaultBindingIP", config.DefaultIp.String())

    if err := job.Run(); err != nil {
        return nil, err
    }
}
```

分析以上源码可知，通过config中的DisableNetwork属性来判断，在默认配置文件中，该属性有过定义，却没有初始值。但是在应用配置信息中处理网络功能配置的时候，将DisableNetwork属性赋值为false，故判断语句结果为真，执行相应的代码块。

首先创建名为”init_networkdriver”的job，随后为该job设置环境变量，环境变量的值如下：

- 环境变量EnableIptables，使用config.EnableIptables来赋值，为true；
- 环境变量InterContainerCommunication，使用config.InterContainerCommunication来赋值，为true；
- 环境变量EnableIpForward，使用config.EnableIpForward来赋值，值为true；
- 环境变量BridgeIface，使用config.BridgeIface来赋值，为空字符串”“；
- 环境变量BridgeIP，使用config.BridgeIP来赋值，为空字符串”“；
- 环境变量DefaultBindingIP，使用config.DefaultIp.String()来赋值，为”0.0.0.0”。

设置完环境变量之后，随即运行该job，由于在eng中key为”init_networkdriver”的handler，value为bridge.InitDriver函数，故执行bridge.InitDriver函数，具体的实现位于./docker/daemon/networkdriver/bridge/dirver.go，作用为：

- 获取为Docker服务的网络设备的地址；
- 创建指定IP地址的网桥；
- 启用Iptables功能并配置；
- 另外还为eng实例注册了4个Handler,如”allocate_interface”，”release_interface”，”allocate_port”，”link”。

4.5.1. 创建Docker网络设备

创建Docker网络设备，属于Docker Daemon创建网络环境的第一步，实际工作是创建名为“docker0”的网桥设备。

在InitDriver函数运行过程中，首先使用job的环境变量初始化内部变量；然后根据目前网络环境，判断是否创建docker0网桥，若Docker专属网桥已存在，则继续往下执行；否则的话，创建docker0网桥。具体

实现为createBridge(bridgeIP)，以及createBridgeIface(bridgeIface)。

createBridge的功能是：在host主机上启动创建指定名称网桥设备的任务，并为该网桥设备配置一个与其他设备不冲突的网络地址。而createBridgeIface通过系统调用负责创建具体实际的网桥设备，并设置MAC地址，通过libcontainer中netlink包的CreateBridge来实现。

4.5.2. 启用iptables功能

创建完网桥之后，Docker Daemon为容器以及host主机配置iptables，包括为container之间所需要的link操作提供支持，为host主机上所有的对外对内流量制定传输规则等。代码位于./docker/daemon/networkdriver/bridge/driver/driver.go#L133-L137,如下：

```
// Configure iptables for link support
if enableIPTables {
    if err := setupIPTables(addr, icc); err != nil {
        return job.Error(err)
    }
}
```

其中setupIPtables的调用过程中，addr地址为Docker网桥的网络地址，icc为true，即为允许Docker容器间互相访问。假设网桥设备名为docker0，网桥网络地址为docker0_ip,设置iptables规则，操作步骤如下：

(1) 使用iptables工具开启新建网桥的NAT功能，使用命令如下：

```
iptables -I POSTROUTING -t nat -s docker0_ip ! -o docker0 -j MASQUERADE
```

(2) 通过icc参数，决定是否允许container间通信，并制定相应iptables的Forward链。Container之间通信，说明数据包从container内发出后，经过docker0，并且还需要在docker0处发往docker0，最终转向指定的container。换言之，从docker0出来的数据包，如果需要继续发往docker0，则说明是container的通信数据包。命令使用如下：

```
iptables -I FORWARD -i docker0 -o docker0 -j ACCEPT
```

(3) 允许接受从container发出，且不是发往其他container数据包。换言之，允许所有从docker0发出且不是继续发向docker0的数据包，使用命令如下：

```
iptables -I FORWARD -i docker0 ! -o docker0 -j ACCEPT
```

(4) 对于发往docker0，并且属于已经建立的连接的数据包，Docker无条件接受这些数据包，使用命令如下：

```
iptables -I FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

4.5.3. 启用系统数据包转发功能

在Linux系统上，数据包转发功能是被默认禁止的。数据包转发，就是当host主机存在多块网卡的时，如果其中一块网卡接收到数据包，并需要将其转发给另外的网卡。通过修改/proc/sys/net/ipv4/ip_forward的值，将其置为1，则可以保证系统内数据包可以实现转发功能，代码如下：

```
if ipForward {
    // Enable IPv4 forwarding
    if err := ioutil.WriteFile("/proc/sys/net/ipv4/ip_forward", []byte{'1', '\n'}, 0644); err != nil {
        job.Logf("WARNING: unable to enable IPv4 forwarding: %s\n", err)
    }
}
```

4.5.4. 创建DOCKER链

在网桥设备上创建一条名为DOCKER的链，该链的作用是在创建Docker container并设置端口映射时使用。实现代码位于./docker/daemon/networkdriver/bridge/driver/driver.go，如下：

```
if err := iptables.RemoveExistingChain("DOCKER"); err != nil {
    return job.Error(err)
}
if enableIPTables {
    chain, err := iptables.NewChain("DOCKER", bridgeIface)
    if err != nil {
        return job.Error(err)
    }
        portmapper.SetIptablesChain(chain)
}
```

4.5.5. 注册Handler至Engine

在创建完网桥，并配置完基本的iptables规则之后，Docker Daemon在网络方面还在Engine中注册了4个Handler，这些Handler的名称与作用如下：

- allocate_interface：为Docker container分配一个专属网卡；
- realease_interface：释放网络设备资源；
- allocate_port：为Docker container分配一个端口；
- link：实现Docker container间的link操作。

由于在Docker架构中，网络是极其重要的一部分，因此Docker网络篇会安排在《Docker源码分析》系列的第六篇。

4.6. 创建graphdb并初始化

Graphdb是一个构建在SQLite之上的图形数据库，通常用来记录节点命名以及节点之间的关联。Docker Daemon使用graphdb来记录镜像之间的关联。创建graphdb的代码如下：

```
graphdbPath := path.Join(config.Root, "linkgraph.db")
graph, err := graphdb.NewSqliteConn(graphdbPath)
if err != nil {
    return nil, err
}
```

以上代码首先确定graphdb的目录为/var/lib/docker/linkgraph.db；随后通过graphdb包内的NewSqliteConn打开graphdb，使用的驱动为”sqlite3”，数据源的名称为”/var/lib/docker/linkgraph.db”；最后通过NewDatabase函数初始化整个graphdb，为graphdb创建entity表，edge表，并在这两个表中初始化部分数据。NewSqliteConn函数的实现位于./docker/pkg/graphdb/conn_sqlite3.go，代码实现如下：

```
func NewSqliteConn(root string) (*Database, error) {
    ......
    conn, err := sql.Open("sqlite3", root)
    ......
    return NewDatabase(conn, initDatabase)
}
```

4.7. 创建execdriver

Execdriver是Docker中用来执行Docker container任务的驱动。创建并初始化graphdb之后，Docker Daemon随即创建了execdriver，具体代码如下：

```
ed, err := execdrivers.NewDriver(config.ExecDriver, config.Root, sysInitPath, sysInfo)
```

可见，在创建execdriver的时候，需要4部分的信息，以下简要介绍这4部分信息：

- config.ExecDriver:Docker运行时中指定使用的exec驱动类别，在默认配置文件中默认使用”native”,也可以将这个值改为”lxc”，则使用lxc接口执行Docker container内部的操作;
- config.Root:Docker运行时的root路径，默认配置文件中为”/var/lib/docker”；
- sysInitPath:系统上存放dockerinit二进制文件的路径，一般为”/var/lib/docker/init/dockerinit-1.2.0”；
- sysInfo:系统功能信息，包括：容器的内存限制功能，交换区内存限制功能，数据转发功能，以及AppArmor安全功能等。

在执行execdrivers.NewDriver之前，首先通过以下代码，获取期望的目标dockerinit文件的路径localPath，以及系统中dockerinit文件实际所在的路径sysInitPath：

```
localCopy := path.Join(config.Root, "init", fmt.Sprintf("
dockerinit-%s", dockerversion.VERSION))
sysInitPath := utils.DockerInitPath(localCopy)
```

通过执行以上代码，localCopy为”/var/lib/docker/init/dockerinit-1.2.0”，而sysyInitPath为当前Docker运行时中dockerinit-1.2.0实际所处的路径，utils.DockerInitPath的实现位于./docker/utils/util.go。若localCopy与sysyInitPath不相等，则说明当前系统中的dockerinit二进制文件，不在localCopy路径下，需要将其拷贝至localCopy下，并对该文件设定权限。

设定完dockerinit二进制文件的位置之后，Docker Daemon创建sysinfo对象，记录系统的功能属性。SysInfo的定义，位于./docker/pkg/sysinfo/sysinfo.go，如下：

```
type SysInfo struct {
    MemoryLimit            bool
    SwapLimit              bool
    IPv4ForwardingDisabled bool
    AppArmor               bool
}
```

其中MemoryLimit通过判断cgroups文件系统挂载路径下是否均存在memory.limit_in_bytes和memory.soft_limit_in_bytes文件来赋值，若均存在，则置为true，否则置为false。SwapLimit通过判断memory.memsw.limit_in_bytes文件来赋值，若该文件存在，则置为true，否则置为false。AppArmor通过host主机是否存在/sys/kernel/security/apparmor来判断，若存在，则置为true，否则置为false。

执行execdrivers.NewDriver时，返回execdriver.Driver对象实例，具体代码实现位于./docker/daemon/execdriver/execdrivers/execdrivers.go，由于选择使用native作为exec驱动，故执行以下的代码，返回最终的execdriver，其中native.NewDriver实现位于./docker/daemon/execdriver/native/driver.go：

return native.NewDriver(path.Join(root, "execdriver", "native"), initPath)

至此，已经创建完毕一个execdriver的实例ed。

4.8. 创建daemon实例

Docker Daemon在经过以上诸多设置以及创建对象之后，整合众多内容，创建最终的Daemon对象实例daemon，实现代码如下：

```
daemon := &Daemon{
    repository:     daemonRepo,
    containers:     &contStore{s: make(map[string]*Container)},
    graph:          g,
    repositories:   repositories,
    idIndex:        truncindex.NewTruncIndex([]string{}),
    sysInfo:        sysInfo,
    volumes:        volumes,
    config:         config,
    containerGraph: graph,
    driver:         driver,
    sysInitPath:    sysInitPath,
    execDriver:     ed,
    eng:            eng,
}
```

以下分析Daemon类型的属性：

属性名	作用
repository	部署所有Docker容器的路径
containers	用于存储具体Docker容器信息的对象
graph	存储Docker镜像的graph对象
repositories	存储Docker镜像元数据的文件
idIndex	用于通过简短有效的字符串前缀定位唯一的镜像
sysInfo	系统功能信息
volumes	管理host主机上volumes内容的graphdriver，默认为vfs类型
config	Config.go文件中的配置信息，以及执行产生的配置DisableNetwork
containerGraph	存放Docker镜像关系的graphdb
driver	管理Docker镜像的驱动graphdriver，默认为aufs类型
sysInitPath	系统dockerinit二进制文件所在的路径
execDriver	Docker Daemon的exec驱动，默认为native类型
eng	Docker的执行引擎Engine类型实例

4.9. 检测DNS配置

创建完Daemon类型实例daemon之后，Docker Daemon使用daemon.checkLocaldns()检测Docker运行环境中DNS的配置， checkLocaldns函数的定义位于./docker/daemon/daemon.go，代码如下：

```
func (daemon *Daemon) checkLocaldns() error {
    resolvConf, err := resolvconf.Get()
    if err != nil {
        return err
    }
    if len(daemon.config.Dns) == 0 && utils.CheckLocalDns(resolvConf) {
        log.Infof("Local (127.0.0.1) DNS resolver found in resolv.conf and
containers can't use it. Using default external servers : %v", DefaultDns)
        daemon.config.Dns = DefaultDns
    }
    return nil
}
```

以上代码首先通过resolvconf.Get()方法获取/etc/resolv.conf中的DNS服务器信息。若本地DNS 文件中有127.0.0.1，而Docker container不能使用该地址，故采用默认外在DNS服务器，为8.8.8.8，8.8.4.4，并将其赋值给config文件中的Dns属性。

4.10. 启动时加载已有Docker containers

当Docker Daemon启动时，会去查看在daemon.repository，也就是在/var/lib/docker/containers中的内容。若有存在Docker container的话，则让Docker Daemon加载这部分容器，将容器信息收集，并做相应的维护。

4.11. 设置shutdown的处理方法

加载完已有Docker container之后，Docker Daemon设置了多项在shutdown操作中需要执行的handler。也就是说：当Docker Daemon接收到特定信号，需要执行shutdown操作时，先执行这些handler完成善后工作，最终再实现shutdown。实现代码如下：

```
eng.OnShutdown(func() {
    if err := daemon.shutdown(); err != nil {
        log.Errorf("daemon.shutdown(): %s", err)
    }
    if err := portallocator.ReleaseAll(); err != nil {
        log.Errorf("portallocator.ReleaseAll(): %s", err)
    }
    if err := daemon.driver.Cleanup(); err != nil {
        log.Errorf("daemon.driver.Cleanup(): %s", err.Error())
    }
    if err := daemon.containerGraph.Close(); err != nil {
        log.Errorf("daemon.containerGraph.Close(): %s", err.Error())
    }
})
```

可知，eng对象shutdown操作执行时，需要执行以上作为参数的func(){……}函数。该函数中，主要完成4部分的操作：

- 运行daemon对象的shutdown函数，做daemon方面的善后工作；
- 通过portallocator.ReleaseAll()，释放所有之前占用的端口资源；
- 通过daemon.driver.Cleanup()，通过graphdriver实现unmount所有layers中的挂载点；
- 通过daemon.containerGraph.Close()关闭graphdb的连接。

4.12. 返回daemon对象实例

当所有的工作完成之后，Docker Daemon返回daemon实例，并最终返回至mainDaemon()中的加载daemon的goroutine中继续执行。

5. 总结

本文从源码的角度深度分析了Docker Daemon启动过程中daemon对象的创建与加载。在这一环节中涉及内容极多，本文归纳总结daemon实现的逻辑，一一深入，具体全面。

在Docker的架构中，Docker Daemon的内容是最为丰富以及全面的，而NewDaemon的实现而是涵盖了Docker Daemon启动过程中的绝大部分。可以认为NewDaemon是Docker Daemon实现过程中的精华所在。深入理解NewDaemon的实现，即掌握了Docker Daemon运行的来龙去脉。

6. 作者简介

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

7. 参考文献

http://docs.studygolang.com/pkg/

http://www.iptables.info/en/iptables-matches.html

https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

http://crosbymichael.com/the-lost-packages-of-docker.html

（五）：Docker Server的创建

- 1.Docker Server简介
- 2. Docker Server源码分析内容安排
- 3.Docker Server创建流程
  - 3.1创建名为”serveapi”的job
  - 3.2配置job环境变量
  - 3.3 运行job
- 4.ServeApi运行流程
- 5.ListenAndServe实现
  - 5.1 创建router路由实例
  - 5.2 创建listener监听实例
  - 5.3 创建http.Server
  - 5.4 启动API服务
- 6.总结
- 7.作者简介
- 8.参考文献

1.Docker Server简介

Docker架构中，Docker Server是Docker Daemon的重要组成部分。Docker Server最主要的功能是：接受用户通过Docker Client发送的请求，并按照相应的路由规则实现路由分发。

同时，Docker Server具备十分优秀的用户友好性，多种通信协议的支持大大降低Docker用户使用Docker的门槛。除此之外，Docker Server设计实现了详尽清晰的API接口，以供Docker用户选择使用。通信安全方面，Docker Server可以提供安全传输层协议（TLS），保证数据的加密传输。并发处理方面，Docker Daemon大量使用了Golang中的goroutine，大大提高了服务端的并发处理能力。

本文为《Docker源码分析》系列的第五篇——Docker Server的创建。

2. Docker Server源码分析内容安排

本文将从源码的角度分析Docker Server的创建，分析内容的安排主要如下：

(1) “serveapi”这个job的创建并执行流程，代表Docker Server的创建；

(2) “serveapi”这个job的执行流程深入分析；

(3) Docker Server创建Listener并服务API的流程分析。

3.Docker Server创建流程

《Docker源码分析（三）：Docker Daemon启动》主要分析了Docker Daemon的启动，而在mainDaemon()运行的最后环节，实现了创建并运行名为”serveapi”的job。这一环节的作用是：让Docker Daemon提供API访问服务。实质上，这正是实现了Docker架构中Docker Server的创建与运行。

从流程的角度来说，Docker Server的创建并运行，代表了”serveapi”这个job的整个生命周期：创建Job实例job，配置job环境变量，以及最终执行该job。本章分三节具体分析这三个不同的阶段。

3.1创建名为”serveapi”的job

Job是Docker架构中Engine内部最基本的任务执行单位，故创建Docker Server这一任务的执行也不例外，需要表示为一个可执行的Job。换言之，需要创建Docker Server，则必须创建一个相应的Job。具体的Job创建形式位于./docker/docker/daemon.go，如下：

```
job := eng.Job("serveapi", flHosts...)
```

以上代码通过Engine实例eng创建一个Job类型的实例job，job名为”serveapi”，同时用flHost的值来初始化job.Args。flHost的作用是：配置Docker Server监听的协议与监听的地址。

需要注意的是，《Docker源码分析（三）：Docker Daemon启动》mainDaemon()具体实现过程中，在加载builtins环节已经向eng对象注册了key为”serveapi”的Handler，而该Handler的value为api.ServeApi。因此，在运行名为”serveapi”的job时，会执行该job的Handler，即api.ServeApi。

3.2配置job环境变量

创建完Job实例job之后，Docker Daemon为job配置环境参数。在Job实现过程中，为Job配置参数有两种方式：第一，创建Job实例时，用指定参数直接初始化Job的Args属性；第二，创建完Job后，给Job添加指定的环境变量。以下代码则实现了为创建的job配置环境变量：

```
job.SetenvBool("Logging", true)
job.SetenvBool("EnableCors", *flEnableCors)
job.Setenv("Version", dockerversion.VERSION)
job.Setenv("SocketGroup", *flSocketGroup)

job.SetenvBool("Tls", *flTls)
job.SetenvBool("TlsVerify", *flTlsVerify)
job.Setenv("TlsCa", *flCa)
job.Setenv("TlsCert", *flCert)
job.Setenv("TlsKey", *flKey)
job.SetenvBool("BufferRequests", true)
```

对于以上配置，环境变量的归纳总结如下表：

环境变量名	flag参数	默认值	作用值
Logging		true	使用日志输出
EnableCors	flEnableCors	false	在远程API中提供CORS头
Version			显示Docker版本号
SocketGroup	flSocketGroup	“docker”	在daemon模式中unix domain socket分配用户组名
Tls	flTls	false	使用TLS安全传输协议
TlsVerify	flTlsVerify	false	使用TLS并验证远程Client
TlsCa	flCa		指定CA文件路径
TlsCert	flCert		TLS证书文件路径
TlsKey	flKey		TLS密钥文件路径
BufferRequest		true	缓存Docker Client请求

3.3 运行job

配置完毕job的环境变量，随即执行job的运行函数，具体实现代码如下：

```
if err := job.Run(); err != nil {
    log.Fatal(err)
}
```

在eng对象中已经注册过key为”serveapi”的Handler，故在运行job的时候，执行这个Handler的value值，相应Handler的value为api.ServeApi。至此，名为”serveapi”的job的生命周期已经完备。下文将深入分析job的Handler，api.ServeApi执行细节的具体实现。

4.ServeApi运行流程

本章将深入分析Docker Server提供API服务的部分，从源码的角度剖析Docker Server的架构设计与实现。

作为一个监听请求、处理请求的服务端，Docker Server首先明确自身需要为多少种通信协议提供服务，在Docker这个C/S模式的架构中，可以使用的协议无外乎三种:TCP协议，Unix Socket形式，以及fd的形式。随后，Docker Server根据协议的不同，分别创建不同的服务端实例。最后，在不同的服务端实例中，创建相应的路由模块，监听模块，以及处理请求的Handler，形成一个完备的server。

”serveapi”这个job在运行时，将执行api.ServeApi函数。ServeApi的功能是：循环检查所有Docker Daemon当前支持的通信协议，并对于每一种协议都创建一个goroutine，在这个goroutine内部配置一个服务于HTTP请求的server端。ServeApi的代码实现位于./docker/api/server/server.go#L1339：

第一，判断job.Args的长度是否为0，由于通过flHosts来初始化job.Args，故job.Args的长度若为0的话，说明没有Docker Server没有监听的协议与地址，参数有误，返回错误信息。代码如下：

```
if len(job.Args) == 0 {
    return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]", job.Name)
}
```

第二，定义两个变量，protoAddrs代表flHosts的内容；而chError定义了和protoAddrs长度一致的error类型channel管道，chError的作用在下文中会说明。同时还定义了activationLock，这是一个用来同步”serveapi”和”acceptconnections”这两个job执行的channel。在serveapi运行时ServeFd和ListenAndServe的实现中，由于activationLock这个channel中没有内容而阻塞，而当运行”acceptionconnections”这个job时，会首先通知init进程Docker Daemon已经启动完毕，并关闭activationLock，同时也开启了serveapi的继续执行。正是由于activationLock的存在，保证了”acceptconnections”这个job的运行起到通知”serveapi”开启正式服务于API的效果。代码如下：

```
var (
    protoAddrs = job.Args
    chErrors   = make(chan error, len(protoAddrs))
)
activationLock = make(chan struct{})
```

第三，遍历protoAddrs，即job.Args，将其中的每一项都按照字符串“://”进行分割，若分割后protoAddrParts的长度不为2，则说明协议加地址的书写形式有误，返回job错误；若不为2，则分割获得每一项中的协议protoAddrPart[0]与地址protoAddrParts[1]。最后分别创建一个goroutine来执行ListenAndServe的操作。goroutine的运行主要依赖于ListenAndServe(protoAddrParts[0], protoAddrParts[1], job)的运行结果，若返回error，则chErrors中有error，当前goroutine执行完毕；若没有返回error，则该goroutine持续运行，持续提供服务。其中最为重要的是ListenAndServe的实现，该函数具体实现了如何创建listener、router以及server，并协调三者进行工作，最终服务于API请求。代码如下：

```
for _, protoAddr := range protoAddrs {
    protoAddrParts := strings.SplitN(protoAddr, "://", 2)
    if len(protoAddrParts) != 2 {
        return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]", job.Name)
    }
    go func() {
        log.Infof("Listening for HTTP on %s (%s)", protoAddrParts[0], protoAddrParts[1])
        chErrors <- ListenAndServe(protoAddrParts[0], protoAddrParts[1], job)
    }()
}
```

第四，根据chErrors的值运行，若chErrors这个channel中有错误内容，则ServeApi该函数返回；若无错误内容，则循环被阻塞。代码如下：

```
for i := 0; i < len(protoAddrs); i += 1 {
    err := <-chErrors
    if err != nil {
        return job.Error(err)
    }
}

return engine.StatusOK
```

至此， ServeApi的运行流程已经详细分析完毕，其中核心部分ListenAndServe的实现，下一章开始深入。

5.ListenAndServe实现

ListenAndServe的功能是：使Docker Server监听某一指定地址，接受该地址上的请求，并对以上请求路由转发至相应的处理函数Handler处。从实现的角度来看，ListenAndServe主要实现了设置一个服务于HTTP的server，该server将监听指定地址上的请求，并对请求做特定的协议检查，最终完成请求的路由与分发。代码实现位于./docker/api/server/server.go。

ListenAndServe的实现可以分为以下4个部分：

(1) 创建router路由实例；

(2) 创建listener监听实例；

(3) 创建http.Server；

(4) 启动API服务。

ListenAndServe的执行流程如下图：

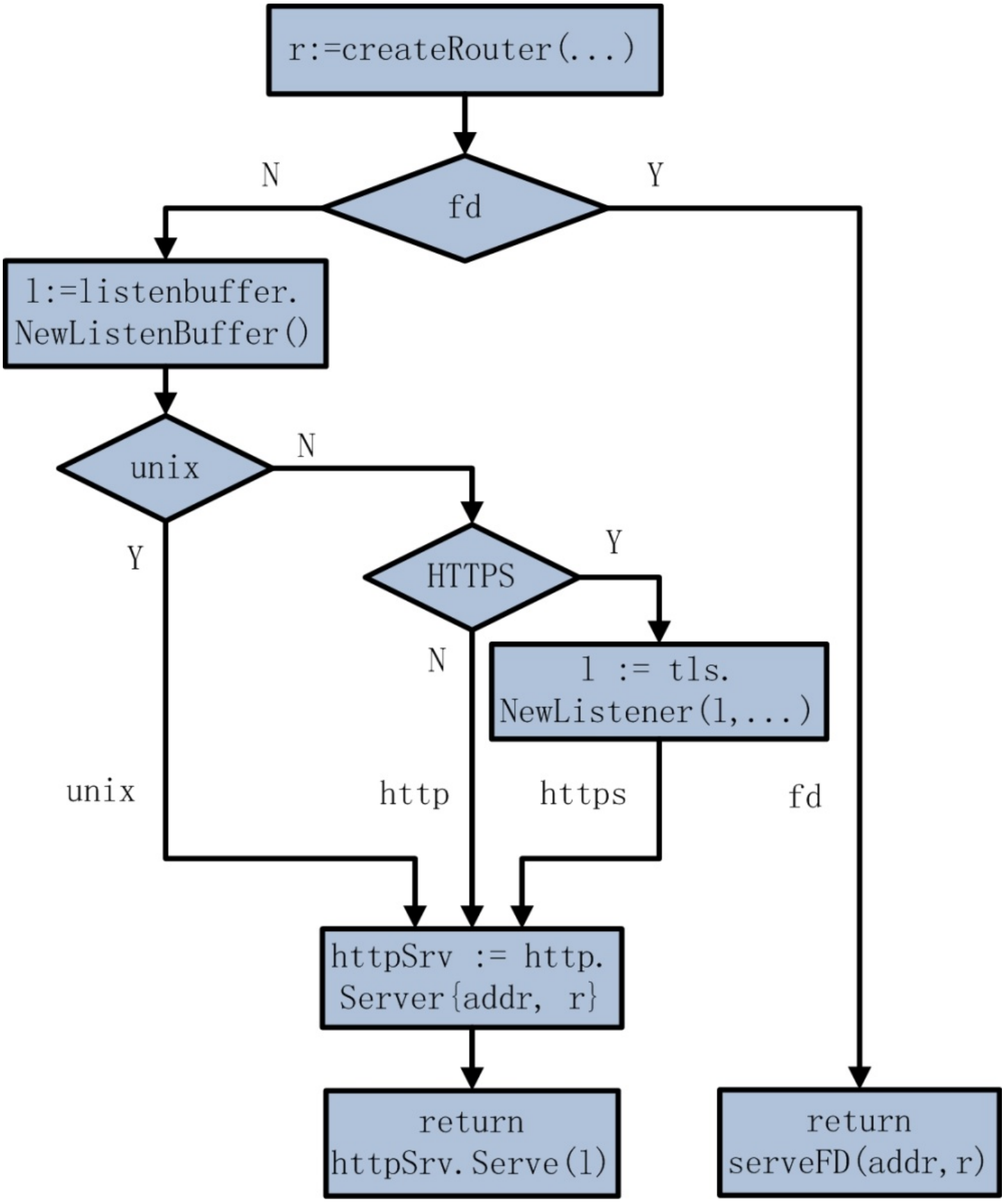


图5.1 ListenAndServer执行流程图

下文将按照ListenAndServe执行流程图一一深入分析各个部分。

5.1 创建router路由实例

首先，ListenAndServe的实现中通过createRouter创建了一个router路由实例。代码实现如下：

```
rr, err := createRouter(job.Eng, job.GetenvBool("Logging"), job.GetenvBool("EnableCors"), job.Getenv(
"Version"))
if err != nil {
    return err
}
```

createRouter的实现位于./docker/api/server/server.go#L1094。

创建router路由实例是一个重要的环节，路由实例的作用是：负责Docker Server对请求进行路由以及分发。实现过程中，主要两个步骤：第一，创建全新的router路由实例；第二，为router实例添加路由记录。

5.1.1 创建空路由实例

实质上，createRouter通过包gorilla/mux实现了一个功能强大的路由器和分发器。如下：

```
r := mux.NewRouter()
```

NewRouter()函数返回了一个全新的router实例r。在创建Router实例时，给Router对象的两个属性进行赋值，这两个属性为nameRoutes和KeepContext。其中namedRoutes属性为一个map类型，其中key为string类型，value为Route路由记录类型；另外，KeepContext属性为false，表示Docker Server在处理完请求之后，就清除请求的内容，不对请求做存储操作。代码位于./docker/vendor/src/github.com/gorilla/mux/mux.go#L16，如下：

```
func NewRouter() *Router {
    return &Router{namedRoutes: make(map[string]*Route), KeepContext: false}
}
```

可见，以上代码返回的类型为mux.Router。mux.Router会通过一系列已经注册过的路由记录，来为接受的请求做匹配，首先通过请求的URL或者其他条件，找到相应的路由记录，并调用这条路由记录中的执行Handler。mux.Router有以下这些特性：

- 请求可以基于URL 的主机名、路径、路径前缀、shemes、请求头和请求值、HTTP请求方法类型或者使用自定义的匹配规则；
- URL主机名和路径可以拥有一个正则表达式来表示；
- 注册的URL可以被直接运用，也可以被保留，这样可以保证维护资源的使用；
- 路由记录可以被用以子路由器：如果父路由记录匹配，则嵌套记录只会被用来测试。当设计一个组内的路由记录共享相同的匹配条件时，如主机名、路劲前缀或者其他重复的属性，子路由的方式很有帮助；
- mux.Router实现了http.Handler接口，故和标准的http.ServeMux兼容。

5.1.2 添加路由记录

Router路由实例r创建完毕，下一步工作是为Router实例r添加所需要的路由记录。路由记录存储着用来匹配请求的信息，包括对请求的匹配规则，以及匹配之后的Handler执行入口。

回到createRouter实现代码中，首先判断Docker Daemon的启动过程中有没有开启DEBUG模式。通过docker可执行文件启动Docker Daemon，解析flag参数时，若flDebug的值为false，则说明不需要配置DEBUG环境；若flDebug的值为true，则说明需要为Docker Daemon添加DEBUG功能。具体的代码实现如下：

```
if os.Getenv("DEBUG") != "" {
    AttachProfiler(r)
}
```

AttachProiler(r)的功能是为路由实例r添加与DEBUG相关的路由记录，具体实现位于./docker/api/server/server.go#L1083，如下：

```
func AttachProfiler(router *mux.Router) {
    router.HandleFunc("/debug/vars", expvarHandler)
    router.HandleFunc("/debug/pprof/", pprof.Index)
    router.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
    router.HandleFunc("/debug/pprof/profile", pprof.Profile)
    router.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
    router.HandleFunc("/debug/pprof/heap", pprof.Handler("heap").ServeHTTP)
    router.HandleFunc("/debug/pprof/goroutine", pprof.Handler("goroutine").ServeHTTP)
    router.HandleFunc("/debug/pprof/threadcreate", pprof.Handler("threadcreate").ServeHTTP)
}
```

分析以上源码，可以发现Docker Server使用两个包来完成DEBUG相关的工作：expvar和pprof。包expvar为公有变量提供标准化的接口，使得这些公有变量可以通过HTTP的形式在”/debug/vars”这个URL下被访问，传输时格式为JSON。包pprof将Docker Server运行时的分析数据通过”/debug/pprof/”这个URL向外暴露。这些运行时信息包括以下内容：可得的信息列表、正在运行的命令行信息、CPU信息、程序函数引用信息、ServeHTTP这个函数三部分信息使用情况（堆使用、goroutine使用和thread使用）。

回到createRouter函数实现中，完成DEBUG功能的所有工作之后，Docker Docker创建了一个map类型的对象m，用于初始化路由实例r的路由记录。简化的m对象，代码如下：

```
m := map[string]map[string]HttpApiFunc{
    "GET": {
        ......
        "/images/{name:.*}/get":           getImagesGet,
        ......
    },
    "POST": {
        ......
        "/containers/{name:.*}/copy":    postContainersCopy,
    },
    "DELETE": {
        "/containers/{name:.*}": deleteContainers,
        "/images/{name:.*}":     deleteImages,
    },
    "OPTIONS": {
        "": optionsHandler,
    },
}
```

对象m的类型为map，其中key为string类型，代表HTTP的请求类型，
如”GET”，”POST”，”DELETE”等，value为另一个map类型，该map代表的是URL与执行Handler的映射。在第二个map类型中，key为string类型，代表是的请求URL，value为HttpApiFunc类型，代表具体的执行Handler。其中HttpApiFunc类型的定义如下：

```
type HttpApiFunc func(eng *engine.Engine, version version.Version,
 w http.ResponseWriter, r *http.Request, vars map[string]string) error
```

完成对象m的定义，随后Docker Server通过该对象m来添加路由实例r的路由记录。对象m的请求方法，请求URL和请求处理Handler这三样内容可以为对象r构建一条路由记录。实现代码。如下：

```
for method, routes := range m {
    for route, fct := range routes {
        log.Debugf("Registering %s, %s", method, route)
        localRoute := route
        localFct := fct
        localMethod := method

            f := makeHttpHandler(eng, logging, localMethod,
localRoute, localFct, enableCors, version.Version(dockerVersion))

        if localRoute == "" {
            r.Methods(localMethod).HandlerFunc(f)
        } else {
            r.Path("/v{version:[0-9.]+}" + localRoute).
Methods(localMethod).HandlerFunc(f)
            r.Path(localRoute).Methods(localMethod).HandlerFunc(f)
        }
    }
}
```

以上代码，在第一层循环中，按HTTP请求方法划分，获得请求方法各自的路由记录，第二层循环，按匹配请求的URL进行划分，获得与URL相对应的执行Handler。在嵌套循环中，通过makeHttpHandler返回一个执行的函数f。在返回的这个函数中，涉及了logging信息，CORS信息（跨域资源共享协议），以及版本信息。以下举例说明makeHttpHandler的实现，从对象m可以看到，对于”GET”请求，请求URL为”/info”，则请求Handler为”getInfo”。执行makeHttpHandler的具体代码实现如下：

```
func makeHttpHandler(eng *engine.Engine, logging bool, localMethod string,
localRoute string, handlerFunc HttpApiFunc, enableCors bool, dockerVersion version.Version) http.Ha
ndlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // log the request
        log.Debugf("Calling %s %s", localMethod, localRoute)

        if logging {
            log.Infof("%s %s", r.Method, r.RequestURI)
        }

        if strings.Contains(r.Header.Get("User-Agent"), "Docker-Client/") {
            userAgent := strings.Split(r.Header.Get("User-Agent"), "/")
            if len(userAgent) == 2 && !dockerVersion.Equal(version.Version(userAgent[1])) {
                log.Debugf("Warning: client and server don't have the same version
(client: %s, server: %s)", userAgent[1], dockerVersion)
            }
        }
        version := version.Version(mux.Vars(r)["version"])
        if version == "" {
            version = api.APIVERSION
        }
        if enableCors {
            writeCorsHeaders(w, r)
        }

        if version.GreaterThan(api.APIVERSION) {
            http.Error(w, fmt.Errorf("client and server don't have same version
(client : %s, server: %s)", version, api.APIVERSION).Error(), http.StatusNotFound)
            return
        }

        if err := handlerFunc(eng, version, w, r, mux.Vars(r)); err != nil {
            log.Errorf("Handler for %s %s returned error: %s", localMethod, localRoute, err)
            httpError(w, err)
        }
    }
}
```

可见makeHttpHandler的执行直接返回一个函数func(w http.ResponseWriter, r *http.Request) 。在这个func函数的实现中，判断makeHttpHandler传入的logging参数，若为true，则将该Handler的执行通过日志显示，另外通过makeHttpHandler传入的enableCors参数判断是否在HTTP请求的头文件中添加跨域资源共享信息，若为true，则通过writeCorsHeaders函数向response中添加有关CORS的HTTP Header，代码实现位于./docker/api/server/server.go#L1022，如下：

```
func writeCorsHeaders(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Access-Control-Allow-Origin", "*")
    w.Header().Add("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept
")
    w.Header().Add("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT, OPTIONS")
}
```

最为重要的执行部分位于handlerFunc(eng, version, w, r, mux.Vars(r))，如以下代码：

```
if err := handlerFunc(eng, version, w, r, mux.Vars(r)); err != nil {
        log.Errorf("Handler for %s %s returned error: %s", localMethod, localRoute, err)
        httpError(w, err)
}
```

对于”GET”请求类型，”/info”请求URL的请求，由于Handler名为getInfo，也就是说handlerFunc这个形参的值为getInfo，故执行部分直接运行getInfo(eng, version, w, r, mux.Vars(r))，而getInfo的具体实现位于./docker/api/server/serve.go#L269，如下：

```
func getInfo(eng *engine.Engine, version version.Version, w http.ResponseWriter,
r *http.Request, vars map[string]string) error {
        w.Header().Set("Content-Type", "application/json")
        eng.ServeHTTP(w, r)
        return nil
}
```

以上makeHttpHandler的执行已经完毕，返回func函数，作为指定URL对应的执行Handler。

创建完处理函数Handler，需要向路由实例中添加新的路由记录。如果URL信息为空，则直接为该HTTP请求方法类型添加路由记录；若URL不为空，则为请求URL路径添加新的路由记录。需要额外注意的是，在URL不为空，为路由实例r添加路由记录时，考虑了API版本的问题，通过r.Path("/v{version:[0-9.]+}" + localRoute).Methods(localMethod).HandlerFunc(f)来实现。

至此，mux.Router实例r的两部分工作工作已经全部完成：创建空的路由实例r，为r添加相应的路由记录，最后返回路由实例r。

现Ⅰ时er路由记录。需要额外的利次循环中，都有不同的组合1083lla/mux/mux.go,

5.2 创建listener监听实例

路由模块，完成了请求的路由与分发这一重要部分，属于ListenAndServe实现中的第一个重要工作。对于请求的监听功能，同样需要模块来完成。而在ListenAndServe实现中，第二个重要的工作就是创建Listener。Listener是一种面向流协议的通用网络监听模块。

在创建Listener之前，先判断Docker Server允许的协议，若协议为fd形式，则直接通过ServeFd来服务请求；若协议不为fd形式，则继续往下执行。

在程序执行过程中，需要判断”serveapi”这个job的环境中”BufferRequests”的值，是否为真，若为真，则通过包listenbuffer创建一个Listener的实例l，否则的话直接通过包net创建Listener实例l。具体的代码位于./docker/api/server/server.go#L1269，如下：

```
if job.GetenvBool("BufferRequests") {
    l, err = listenbuffer.NewListenBuffer(proto, addr, activationLock)
} else {
    l, err = net.Listen(proto, addr)
}
```

由于在mainDaemon()中创建”serveapi”这个job之后，给job添加环境变量时，已经给”BufferRequets”赋值为true，故使用包listenbuffer创建listener实例。

Listenbuffer的作用是：让Docker Server可以立即监听指定协议地址上的请求，但是将这些请求暂时先缓存下来，等Docker Daemon全部启动完毕之后，才让Docker Server开始接受这些请求。这样设计有一个很大的好处，那就是可以保证在Docker Daemon还没有完全启动完毕之前，接收并缓存尽可能多的用户请求。

若协议的类型为TCP，另外job中环境变量Tls或者TlsVerify有一个为真，则说明Docker Server需要支持HTTPS服务，需要为Docker Server配置安全传输层协议（TLS）的支持。为实现TLS协议，首先需要建立一个tls.Config类型实例tlsConfig，然后在tlsConfig中加载证书，认证信息等，最终通过包tls中的NewListener函数，创建出适应于接收HTTPS协议请求的Listener实例l，代码如下：

```
l = tls.NewListener(l, tlsConfig)
```

至此，创建网络监听的Listener部分已经全部完成。

5.3 创建http.Server

Docker Server同样需要创建一个Server对象来运行HTTP服务端。在ListenAndServe实现中第三个重要的工作就是创建http.Server：

```
httpSrv := http.Server{Addr: addr, Handler: r}
```

其中addr为需要监听的地址，r为mux.Router路由实例。

5.4 启动API服务

创建http.Server实例之后，Docker Server立即启动API服务，使Docker Server开始在Listener监听实例l上接受请求，并对于每一个请求都生成一个新的goroutine来做专属服务。对于每一个请求，goroutine会读取请求，查询路由表中的路由记录项，找到匹配的路由记录，最终调用路由记录中的执行Handler，执行完毕后，goroutine对请求返回响应信息。代码如下：

```
return httpSrv.Serve(l)
```

至此，ListenAndServer的所有流程已经分析完毕，Docker Server已经开始针对不同的协议，服务API请求。

6.总结

Docker Server作为Docker Daemon架构中请求的入口，接管了所有Docker Daemon对外的通信。通信API的规范性，通信过程的安全性，服务请求的并发能力，往往都是Docker用户最为关心的内容。本文基于源码，分析了Docker Server大部分的细节实现。希望Docker用户可以初探Docker Server的设计理念，并且可以更好的利用Docker Server创造更大的价值。

7.作者简介

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

8.参考文献

http://guzalexander.com/2013/12/06/golang-channels-tutorial.html

http://www.gorillatoolkit.org/pkg/mux

http://docs.studygolang.com/pkg/expvar/

http://docs.studygolang.com/pkg/net/http/pprof/

（六）：Docker Daemon网络

- 1. 前言
- 2. Docker Daemon网络分析内容安排
- 3. Docker Daemon网络配置
  - 3.1 Docker Daemon网络配置接口
  - 3.2 Docker Daemon网络初始化
  - 3.3 创建Docker网桥
- 4 总结
- 5 作者介绍
- 6 参考文献

1. 前言

Docker作为一个开源的轻量级虚拟化容器引擎技术，已然给云计算领域带来了新的发展模式。Docker借助容器技术彻底释放了轻量级虚拟化技术的威力，让容器的伸缩、应用的运行都变得前所未有的方便与高效。同时，Docker借助强大的镜像技术，让应用的分发、部署与管理变得史无前例的便捷。然而，Docker毕竟是一项较为新颖的技术，在Docker的世界中，用户并非一劳永逸，其中最为典型的便是Docker的网络问题。

毋庸置疑，对于Docker管理者和开发者而言，如何有效、高效的管理Docker容器之间的交互以及Docker容器的网络一直是一个巨大的挑战。目前，云计算领域中，绝大多数系统都采取分布式技术来设计并实现。然而，在原生态的Docker世界中，Docker的网络却是不具备跨宿主机能力的，这也或多或少滞后了Docker在云计算领域的高速发展。

工业界中，Docker的网络问题的解决势在必行，在此环境下，很多IT企业都开发了各自的新产品来帮助完善Docker的网络。这些企业中不乏像Google一样的互联网翘楚企业，同时也有不少初创企业率先出击，在最前沿不懈探索。这些新产品中有，Google推出的容器管理和编排开源项目Kubernetes，Zett.io公司开发的通过虚拟网络连接跨宿主机容器的工具Weave，CoreOS团队针对Kubernetes设计的网络覆盖工具Flannel，Docker官方的工程师Jérôme Petazzoni自己设计的SDN网络解决方案Pipework，以及SocketPlane项目等。

对于Docker管理者与开发者而言，Docker的跨宿主机通信能力固然重要，但Docker自身的网络架构也同样重要。只有深入了解Docker自身的网络设计与实现，才能在这基础上扩展Docker的跨宿主机能力。

Docker自身的网络主要包含两部分：Docker Daemon的网络配置，Docker Container的网络配置。本文主要分析Docker Daemon的网络。

2. Docker Daemon网络分析内容安排

本文从源码的角度，分析Docker Daemon在启动过程中，为Docker配置的网络环境，章节安排如下：

(1) Docker Daemon网络配置；

(2) 运行Docker Daemon网络初始化任务；

(3) 创建Docker网桥。

本文为《Docker源码分析》系列第六篇——Docker Daemon网络篇，第七篇将安排Docker Container网络篇。

3. Docker Daemon网络配置

Docker环境中，Docker管理员完全有权限配置Docker Daemon运行过程中的网络模式。 关于Docker的网络模式，大家最熟知的应该就是“桥接”的模式。下图为桥接模式下，Docker的网络环境拓扑图（包括Docker Daemon网络环境和Docker Container网络环境）：

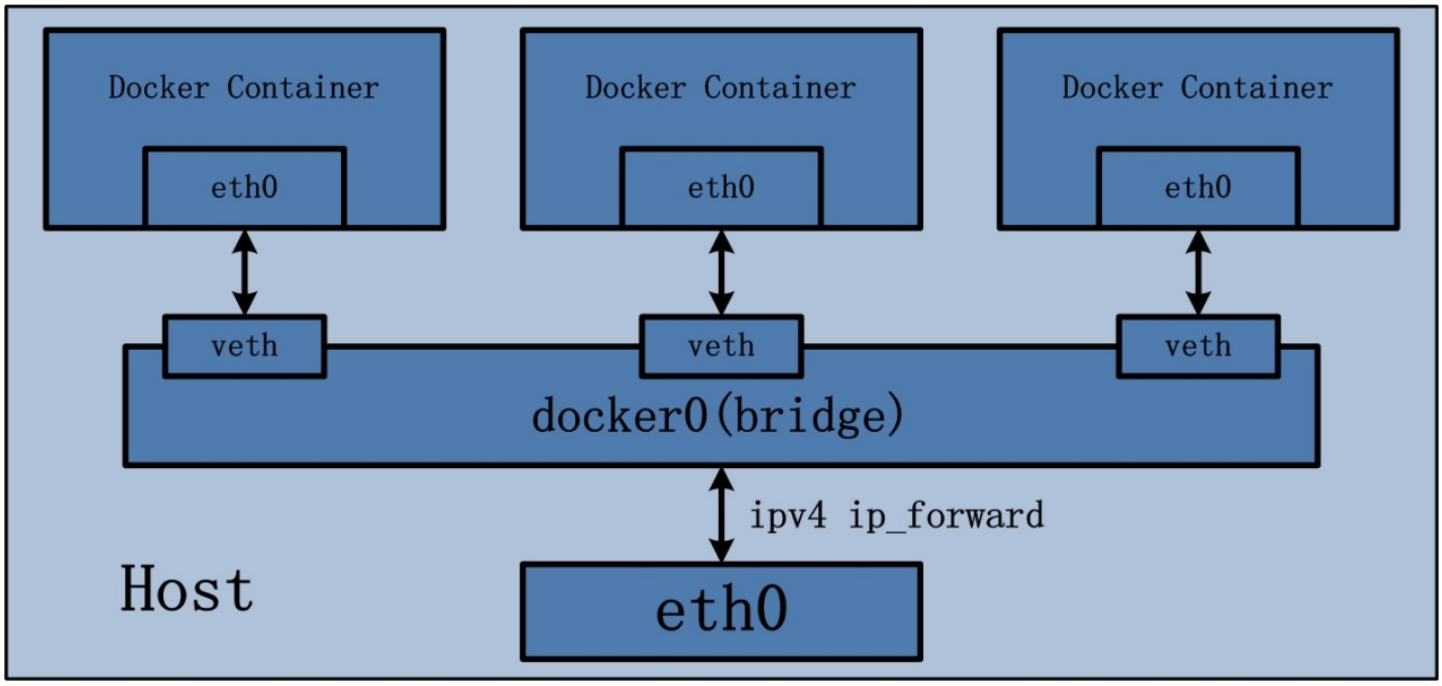


图3.1 Docker网络桥接示意图

然而，“桥接”是Docker网络模式中最为常用的模式。除此之外，Docker还为用户提供了更多的可选项，下文将对此一一说来。

3.1 Docker Daemon网络配置接口

Docker Daemon每次启动的过程中，都会初始化自身的网络环境，这样的网络环境最终为Docker Container提供网络通信服务。

Docker管理员配置Docker的网络环境，可以在Docker Daemon启动时，通过Docker提供的接口来完成。换言之，可以使用docker二进制可执行文件，运行docker -d并添加相应的flag参数来完成。

其中涉及的flag参数有EnableIptables、EnableIpForward、BridgeIface、BridgeIP以及

InterContainerCommunication。该五个参数的定义位于./docker/daemon/config.go，具体代码如下：

```
flag.BoolVar(&config.EnableIptables, []string{"#iptables", "-iptables"}, true, "Enable Docker's addition of iptables rules")
flag.BoolVar(&config.EnableIpForward, []string{"#ip-forward", "-ip-forward"}, true, "Enable net.ipv4.ip_forward")
flag.StringVar(&config.BridgeIP, []string{"#bip", "-bip"}, "", "Use this CIDR notation address for the network bridge's IP, not compatible with -b")
flag.StringVar(&config.BridgeIface, []string{"b", "-bridge"}, "", "Attach containers to a pre-existing network bridge\nuse 'none' to disable container networking")
flag.BoolVar(&config.InterContainerCommunication, []string{"#icc", "-icc"}, true, "Enable inter-container communication")
```

以下介绍这5个flag的作用：

- EnableIptables：确保Docker对于宿主机上的iptables规则拥有添加权限；
- EnableIpForward：确保net.ipv4.ip_forward可以使用，使得多网络接口设备模式下，数据报可以在网络设备之间转发；
- BridgeIP：在Docker Daemon启动过程中，为网络环境中的网桥配置CIDR网络地址；
- BridgeIface：为Docker网络环境指定具体的通信网桥，若BridgeIface的值为”none”，则说明不需要为Docker Container创建网桥服务，关闭Docker Container的网络能力；
- InterContainerCommunication：确保Docker容器之间可以完成通信。

除了Docker会使用到的5个flag参数之外，Docker在创建网络环境时，还使用一个DefaultIP变量，如下：

```
opts.IPVar(&config.DefaultIp, []string{"#ip", "-ip"}, "0.0.0.0", "Default IP address to use when binding container ports")
```

该变量的作用是：当绑定容器的端口时，将DefaultIp作为默认使用的IP地址。

具备了以上Docker Daemon的网络背景知识，以下着重举例分析使用BridgeIP和BridgeIface，在启动Docker Daemon时进行网络配置：

启动Docker Daemon使用命令	**用途注释**
docker -d	启动Docker Daemon，使用默认网桥docker0，不指定CIDR网络地址
docker -d -b=”xxx”	启动Docker Daemon，使用网桥xxx，不指定CIDR网络地址
docker -d --bip=”172.17.42.1”	启动Docker Daemon，使用默认网桥docker0，使用指定CIDR网络地址”172.17.42.1”
docker -d --bridge=”xxx” --bip=”10.0.42.1”	报错，出现兼容性问题，不能同时指定”BridgeIP”和”BridgeIface”

启动Docker Daemon使用命令	用途注释
docker -d --bridge=”none”	启动Docker Daemon，不创建Docker网络环境

深入理解BridgeIface与BridgeIP，并熟练使用相应的flag参数，即做到了如何配置Docker Daemon的网络环境。需要特别注意的是，Docker Daemon的网络与Docker Container的网络存在很大的区别。Docker Daemon为Docker Container创建网络的大环境，Docker Container的网络需要Docker Daemon的网络提供支持，但不唯一。举一个形象的例子，Docker Daemon可以创建docker0网桥，为之后Docker Container的桥接模式提供支持，然而Docker Container仍然可以根据用户需求创建自身网络，其中Docker Container的网络可以是桥接模式的网络，同时也可以直接共享使用宿主机的网络接口，另外还有其他模式，会在《Docker源码分析》系列的第七篇——Docker Container网络篇中详细介绍。

3.2 Docker Daemon网络初始化

正如上一节所言，Docker管理员可以通过与网络相关的flag参数BridgeIface与BridgeIP，来为Docker Daemon创建网路环境。最简单的，Docker管理员通过执行”docker -d”就已经完成了运行Docker Daemon，而Docker Daemon在启动的时候，根据以上两个flag参数的值，创建相应的网络环境。

Docker Daemon网络初始化流程图如下：

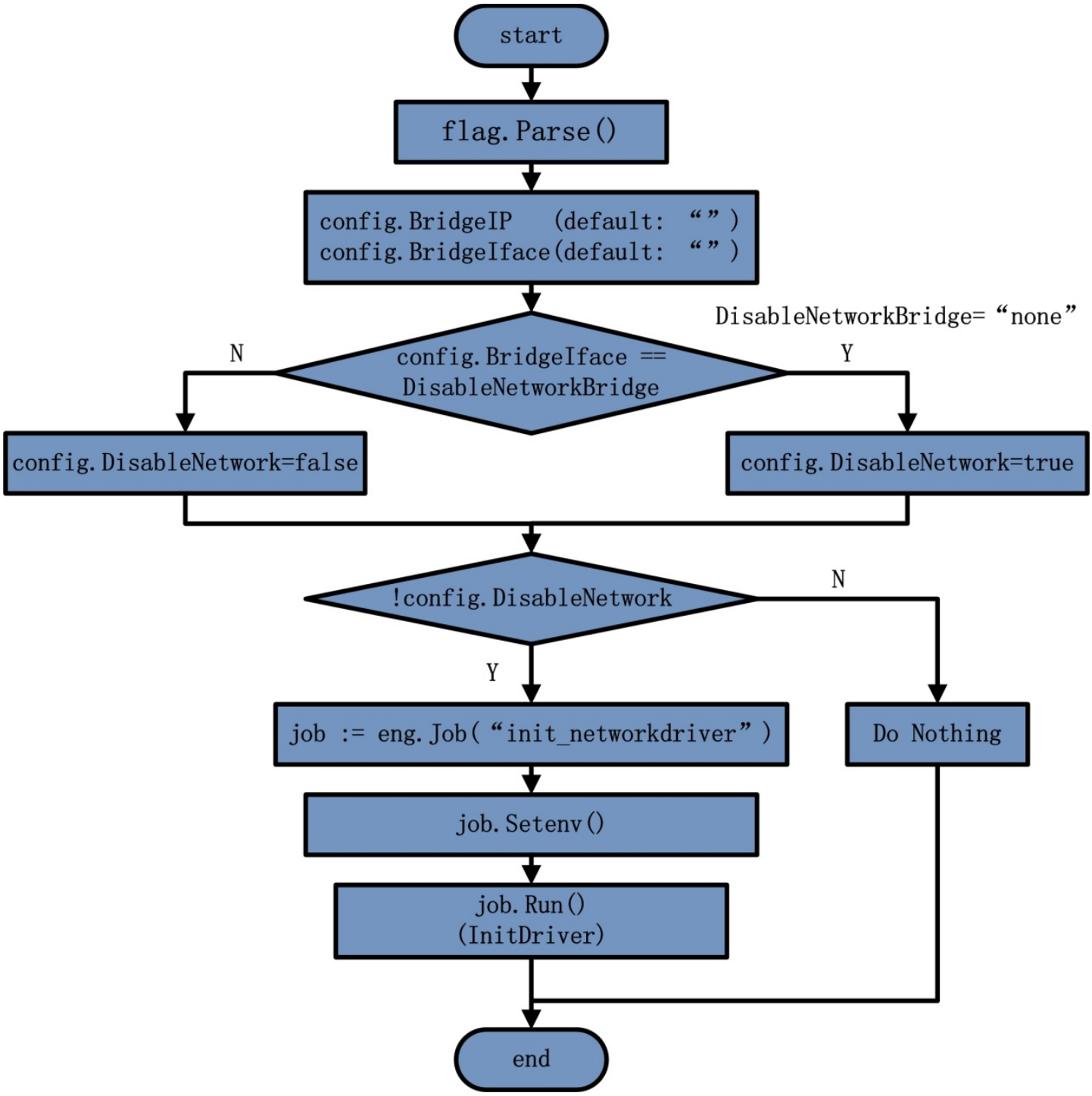


图 3.2 Docker Daemon网络初始化流程图

Docker Daemon网络初始化的流程总体而言，主要是根据解析flag参数来决定到底建立哪种类型的网络环境。从流程图中可知，Docker Daemon创建网络环境时有两个分支，不难发现分支代表的分别是：为Docker创建一个网络驱动、以及对Docker的网络不做任何的操作。

以下参照Docker Daemon网络初始化流程图具体分析实现步骤。

3.2.1 启动Docker Daemon传递flag参数

用户启动Docker Daemon，并在命令行中选择性的传入所需要的flag参数。

3.2.2 解析网络flag参数

flag包对命令行中的flag参数进行解析，其中和Docker Daemon网络配置相关的flag参数有5个，分别

是：EnableIptables、EnableIpForward、BridgeIP、BridgeIface以及InterContanierCommunication，各个flag参数的作用上文已有介绍。

3.2.3 预处理flag参数

预处理与网络配置相关的flag参数信息，包括检测配置信息的兼容性、以及判断是否创建Docker网络环境。

首先检验是否会出现彼此不兼容的配置信息，源码位于./docker/daemon/daemon.go#L679-L685。

这部分的兼容信息有两种。第一种是BridgeIP和BridgeIface配置信息的兼容性，具体表现为用户启动Docker Daemon时，若同时指定了BridgeIP和BridgIface的值，则出现兼容问题。原因为这两者属于互斥对，换言之，若用户指定了新建网桥的设备名，那么该网桥已经存在，无需指定网桥的IP地址BridgeIP；若用户指定了新建网桥的网络IP地址BridgeIP，那么该网桥肯定还没有新建成功，则Docker Daemon在新建网桥时使用默认网桥名“docker0”。具体如下：

```
// Check for mutually incompatible config options
if config.BridgeIface != "" && config.BridgeIP != "" {
    return nil, fmt.Errorf("You specified -b & --bip, mutually exclusive options. Please specify only one."
)
}
```

第二种是EnableIptables和InterContainerCommunication配置的兼容性，具体是指不能同时指定这两个flag参数为false。原因很简单，如果指定InterContainerCommunication为false，则说明Docker Daemon不允许创建的Docker容器之间互相进行通信。但是为了达到以上目的，Docker正是使用iptables过滤规则。因此，再次设定EnableIptables为false，关闭iptables的使用，即出现了自相矛盾的结果。代码如下：

```
if !config.EnableIptables && !config.InterContainerCommunication {
    return nil, fmt.Errorf("You specified --iptables=false with --icc=false. ICC uses iptables to function. P
lease set --icc or --iptables to true.")
    }
```

检验完系统配置信息的兼容性问题，Docker Daemon接着会判断是否需要为Docker Daemon配置网络环境。判断的依据为BridgeIface的值是否与DisableNetworkBridge的值相等，DisableNetworkBridge在./docker/daemon/config.go#L13中被定义为const量，值为字符串”none”。因此，若BridgeIface为”none”，则DisableNetwork为true，最终Docker Daemon不会创建网络环境；若BridgeIface不为”none”，则DisableNetwork为false，最终Docker Daemon需要创建网络环境（桥接模式）。

3.2.4 确定Docker网络模式

Docker网络模式由配置信息DisableNetwork决定。由于在上一环节已经得出DisableNetwork的值，故这一环节可以确定Docker网络模式。该部分的源码实现位于./docker/daemon/daemon.go#L792-L805，如下：

```
if !config.DisableNetwork {
    job := eng.Job("init_networkdriver")

    job.SetenvBool("EnableIptables", config.EnableIptables)
    job.SetenvBool("InterContainerCommunication", config.InterContainerCommunication)
    job.SetenvBool("EnableIpForward", config.EnableIpForward)
    job.Setenv("BridgeIface", config.BridgeIface)
    job.Setenv("BridgeIP", config.BridgeIP)
    job.Setenv("DefaultBindingIP", config.DefaultIp.String())

    if err := job.Run(); err != nil {
        return nil, err
    }
}
```

若DisableNetwork为false，则说明需要创建网络环境，具体的模式为创建Docker网桥模式。创建网络环境的步骤为：

(1) 创建名为”init_networkdriver”的job；

(2) 为该job配置环境变量，设置的环境变量有EnableIptables、InterContainerCommunication、EnableIpForward、BridgeIface、BridgeIP以及DefaultBindingIP；

(3) 运行job。

运行”init_network”即为创建Docker网桥，这部分内容将会在下一节详细分析。

若DisableNetwork为true。则说明不需要创建网络环境，网络模式属于none模式。

以上便是Docker Daemon网络初始化的所有流程。

3.3 创建Docker网桥

Docker的网络往往是Docker开发者最常提起的话题。而Docker网络中最常使用的模式为bridge桥接模式。本小节将详细分析创建Docker网桥的创建流程。

创建Docker网桥的实现通过”init_network”这个job的运行来完成。”init_network”的实现为InitDriver函数，位于./docker/daemon/networkdriver/bridge/driver.go#L79，运行流程如下：

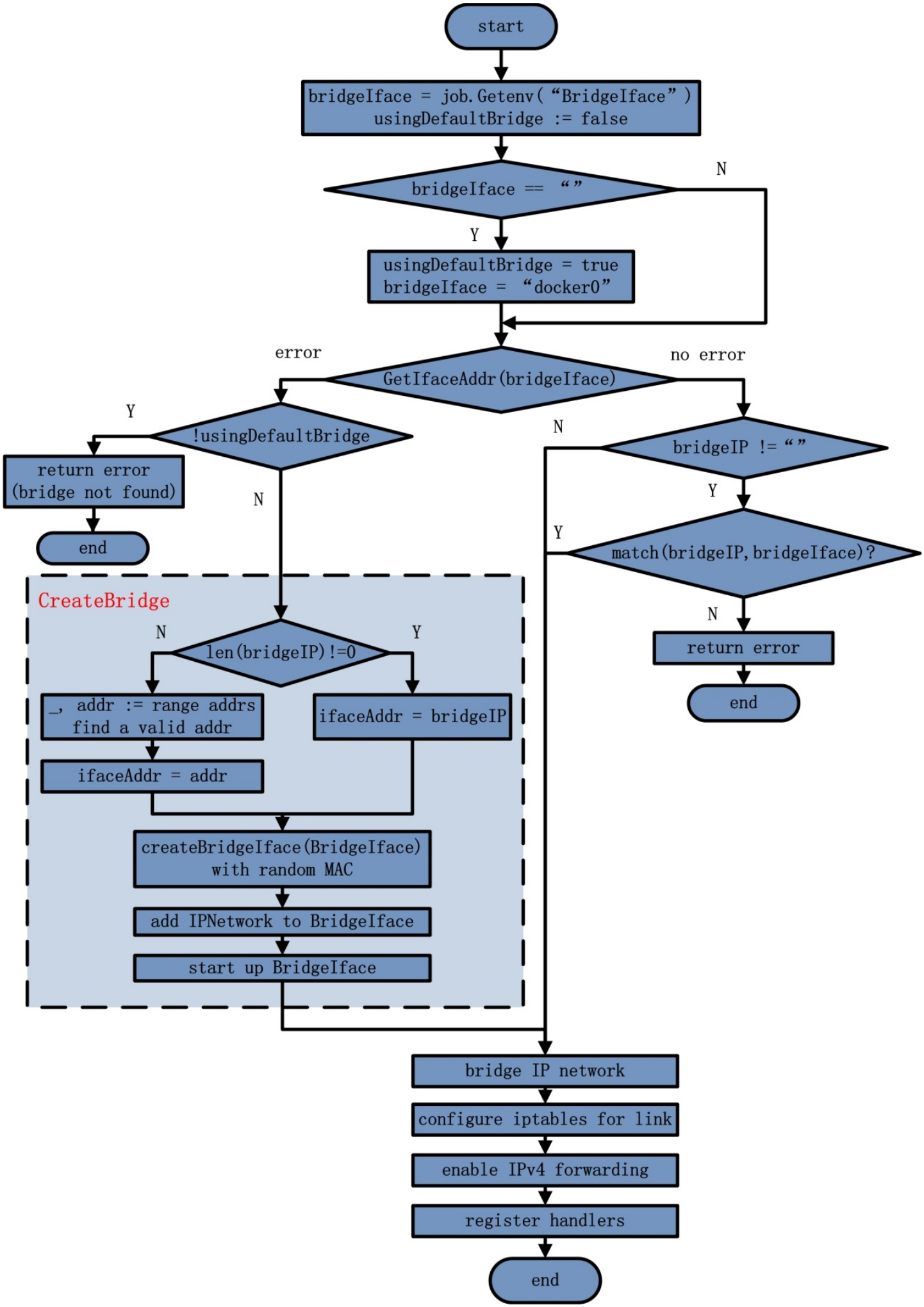

start
bridgeIface = job.Getenv("BridgeIface")
usingDefaultBridge := false
bridgeIface == ""
N
Y
usingDefaultBridge = true
bridgeIface = "docker0"
error
GetIfaceAddr(bridgeIface)
no error
Y
!usingDefaultBridge
return error
(bridge not found)
end
N
CreateBridge
N
len(bridgeIP)!=0
Y
_, addr := range addrs
find a valid addr
ifaceAddr = addr
ifaceAddr = bridgeIP
createBridgeIface(BridgeIface)
with random MAC
add IPNetwork to BridgeIface
start up BridgeIface
N
bridgeIP !=""
Y
Y
match(bridgeIP,bridgeIface)?
N
return error
end
bridge IP network
configure iptables for link
enable IPv4 forwarding
register handlers
end

图3.3 Docker Daemon创建网桥流程图

3.3.1 提取环境变量

在InitDriver函数的实现过程中，Docker首先提取”init_networkdriver”这个job的环境变量。这样的环境变量共有6个，各自的作用在上文已经详细说明。具体的实现代码为：

```
var (
    network        *net.IPNet
    enableIPTables = job.GetenvBool("EnableIptables")
    icc            = job.GetenvBool("InterContainerCommunication")
    ipForward      = job.GetenvBool("EnableIpForward")
    bridgeIP       = job.Getenv("BridgeIP")
)

if defaultIP := job.Getenv("DefaultBindingIP"); defaultIP != "" {
    defaultBindingIP = net.ParseIP(defaultIP)
}

bridgeIface = job.Getenv("BridgeIface")
```

3.3.2 确定Docker网桥设备名

提取job的环境变量之后，Docker随即确定最终使用网桥设备的名称。为此，Docker首先创建了一个名为usingDefaultBridge的bool变量，含义为是否使用默认的网桥设备，默认值为false。接着，若环境变量中bridgeIface的值为空，则说明用户启动Docker时，没有指定特定的网桥设备名，因此Docker首先将usingDefaultBridge置为true，然后使用默认的网桥设备名DefaultNetworkBridge，即docker0；若bridgeIface的值不为空，则判断条件不成立，继续往下执行。这部分的代码实现为：

```
usingDefaultBridge := false
if bridgeIface == "" {
    usingDefaultBridge = true
    bridgeIface = DefaultNetworkBridge
}
```

3.3.3 查找bridgeIface网桥设备

确定Docker网桥设备名bridgeIface之后，Docker首先通过bridgeIface设备名在宿主机上查找该设备是否真实存在。若存在，则返回该网桥设备的IP地址，若不存在，则返回nil。实现代码位于./docker/daemon/networkdriver/bridge/driver.go#L99，如下：

```
addr, err := networkdriver.GetIfaceAddr(bridgeIface)
```

GetIfaceAddr的实现位于./docker/daemon/networkdriver/utils.go，实现步骤为：首先通过Golang中net包的InterfaceByName方法获取名为bridgeIface的网桥设备，会得出以下结果：

- 若名为bridgeIface的网桥设备不存在，直接返回error；
- 若名为bridgeIface的网桥设备存在，返回该网桥设备的IP地址。

需要强调的是：GetIfaceAddr函数返回error，说明当前宿主机上不存在名为bridgeIface的网桥设备。而这样的结果会有两种不同的情况：第一，用户指定了bridgeIface，那么usingDefaultBridge为false，而该bridgeIface网桥设备在宿主机上不存在；第二，用户没有指定bridgeIface，那么usingDefaultBridge为true，bridgeIface名为docker0，而docker0网桥在宿主机上也不存在。

当然，若GetIfaceAddr函数返回的是一个IP地址，则说明当前宿主机上存在名为bridgeIface的网桥设备。这样的结果同样会有两种不同的情况：第一，用户指定了bridgeIface，那么usingDefaultBridge为false，而该bridgeIface网桥设备在宿主机上已经存在；第二，用户没有指定bridgeIface，那么usingDefaultBridge为true，bridgeIface名为docker0，而docker0网桥在宿主机上也已经存在。第二种情况一般是：用户在宿主机上第一次启动Docker Daemon时，创建了默认网桥设备docker0，而后docker0网桥设备一直存在于宿主机上，故之后在不指定网桥设备的情况下，重启Docker Daemon，会出现docker0已经存在的情况。

以下两小节将分别从bridgeIface已创建与bridgeIface未创建两种不同的情况分析。

3.3.4 bridgeIface已创建的情况

Docker Daemon所在宿主机上bridgeIface的网桥设备存在时，Docker Daemon仍然需要验证用户在配置信息中是否为网桥设备指定了IP地址。

用户启动Docker Daemon时，假如没有指定bridgeIP参数信息，则Docker Daemon使用名为bridgeIface的原有的IP地址。

当用户指定了bridgeIP参数信息时，则需要验证：指定的bridgeIP参数信息与bridgeIface网桥设备原有的IP地址信息是否匹配。若两者匹配，则验证通过，继续往下执行；若两者不匹配，则验证不通过，抛出错误，显示“bridgeIP与已有网桥配置信息不匹配”。该部分内容位于./docker/daemon/networkdriver/bridge/driver.go#L119-L129，代码如下：

```
network = addr.(*net.IPNet)
// validate that the bridge ip matches the ip specified by BridgeIP
if bridgeIP != "" {
    bip, _, err := net.ParseCIDR(bridgeIP)
    if err != nil {
        return job.Error(err)
    }
    if !network.IP.Equal(bip) {
        return job.Errorf("bridge ip (%s) does not match existing bridge configuration %s", network.IP, bi
p)
    }
}
```

3.3.5 bridgeIface未创建的情况

Docker Daemon所在宿主机上bridgeIface的网桥设备未创建时，上文已经介绍将存在两种情况：

l 用户指定的bridgeIface未创建；

l 用户未指定bridgeIface，而docker0暂未创建。

当用户指定的bridgeIface不存在于宿主机时，即没有使用Docker的默认网桥设备名docker0，Docker打印日志信息“指定网桥设备未找到”，并返回网桥未找到的错误信息。代码实现如下：

```
if !usingDefaultBridge {
    job.Logf("bridge not found: %s", bridgeIface)
    return job.Error(err)
}
```

当使用的默认网桥设备名，而docker0网桥设备还未创建时，Docker Daemon则立即实现创建网桥的操作，并返回该docker0网桥设备的IP地址。代码如下：

```
// If the iface is not found, try to create it
job.Logf("creating new bridge for %s", bridgeIface)
if err := createBridge(bridgeIP); err != nil {
    return job.Error(err)
}

job.Logf("getting iface addr")
addr, err = networkdriver.GetIfaceAddr(bridgeIface)
if err != nil {
    return job.Error(err)
}
network = addr.(*net.IPNet)
```

创建Docker Daemon网桥设备docker0的实现，全部由createBridge(bridgeIP)来实现，createBridge的实现位于./docker/daemon/networkdriver/bridge/driver.go#L245。

createBridge函数实现过程的主要步骤为：

(1) 确定网桥设备docker0的IP地址；

(2) 通过createBridgeIface函数创建docker0网桥设备，并为网桥设备分配随机的MAC地址；

(3) 将第一步中已经确定的IP地址，添加给新创建的docker0网桥设备；

(4) 启动docker0网桥设备。

以下详细分析4个步骤的具体实现。

首先Docker Daemon确定docker0的IP地址，实现方式为判断用户是否指定bridgeIP。若用户未指定bridgeIP，则从Docker预先准备的IP网段列表addrs中查找合适的网段。具体的代码实现位于./docker/daemon/networkdriver/bridge/driver.go#L257-L278，如下：

```
if len(bridgeIP) != 0 {
    _, _, err := net.ParseCIDR(bridgeIP)
    if err != nil {
        return err
    }
    ifaceAddr = bridgeIP
} else {
    for _, addr := range addrs {
        _, dockerNetwork, err := net.ParseCIDR(addr)
        if err != nil {
            return err
        }
        if err := networkdriver.CheckNameserverOverlaps(nameservers, dockerNetwork); err == nil {
            if err := networkdriver.CheckRouteOverlaps(dockerNetwork); err == nil {
                ifaceAddr = addr
                break
            } else {
                log.Debugf("%s %s", addr, err)
            }
        }
    }
}
```

其中为网桥设备准备的候选网段地址addrs为：

```
addrs = []string{
        "172.17.42.1/16", // Don't use 172.16.0.0/16, it conflicts with EC2 DNS 172.16.0.23
        "10.0.42.1/16",   // Don't even try using the entire /8, that's too intrusive
        "10.1.42.1/16",
        "10.42.42.1/16",
        "172.16.42.1/24",
        "172.16.43.1/24",
        "172.16.44.1/24",
        "10.0.42.1/24",
        "10.0.43.1/24",
        "192.168.42.1/24",
        "192.168.43.1/24",
        "192.168.44.1/24",
}
```

通过以上的流程的执行，可以确定找到一个可用的IP网段地址，为ifaceAddr；若没有找到，则返回错误日志，表明没有合适的IP地址赋予docker0网桥设备。

第二个步骤通过createBridgeIface函数创建docker0网桥设备。createBridgeIface函数的实现如下：

```
func createBridgeIface(name string) error {
    kv, err := kernel.GetKernelVersion()
    // only set the bridge's mac address if the kernel version is > 3.3
    // before that it was not supported
    setBridgeMacAddr := err == nil && (kv.Kernel >= 3 && kv.Major >= 3)
    log.Debugf("setting bridge mac address = %v", setBridgeMacAddr)
    return netlink.CreateBridge(name, setBridgeMacAddr)
}
```

以上代码通过宿主机Linux内核信息，确定是否支持设定网桥设备的MAC地址。若Linux内核版本大于3.3，则支持配置MAC地址，否则则不支持。而Docker在不小于3.8的内核版本上运行才稳定，故可以认为内核支持配置MAC地址。最后通过netlink的CreateBridge函数实现创建docker0网桥。

Netlink是Linux中一种较为特殊的socket通信方式，提供了用户应用间和内核进行双向数据传输的途径。在这种模式下，用户态可以使用标准的socket API来使用netlink强大的功能，而内核态需要使用专门的内核API才能使用netlink。

Libcontainer的netlink包中CreateBridge实现了创建实际的网桥设备，具体使用系统调用的代码如下：

```
syscall.Syscall(syscall.SYS_IOCTL, uintptr(s), SIOC_BRADDBR, uintptr(unsafe.Pointer(nameBytePtr)))
```

创建完网桥设备之后，为docker0网桥设备配置MAC地址，实现函数为setBridgeMacAddress。

第三个步骤是为创建docker0网桥设备绑定IP地址。上一步骤仅完成了创建名为docker0的网桥设备，之后仍需要为docker0网桥设备绑定IP地址。具体代码实现为：

```
if netlink.NetworkLinkAddIp(iface, ipAddr, ipNet); err != nil {
        return fmt.Errorf("Unable to add private network: %s", err)
}
```

NetworkLinkAddIP的实现同样位于libcontainer中的netlink包，主要的功能为：通过netlink机制为一个网络接口设备绑定一个IP地址。

第四个步骤是启动docker0网桥设备。具体实现代码为：

```
if err := netlink.NetworkLinkUp(iface); err != nil {
        return fmt.Errorf("Unable to start network bridge: %s", err)
    }
```

NetworkLinkUp的实现同样位于libcontainer中的netlink包，功能为启动docker网桥设备。

至此，docker0网桥历经确定IP、创建、绑定IP、启动四个环节，createBridge关于docker0网桥设备的工作全部完成。

3.3.6 获取网桥设备的网络地址

创建完网桥设备之后，网桥设备必然会存在一个网络地址。网桥网络地址的作用为：Docker Daemon在创建Docker Container时，使用该网络地址为Docker Container分配IP地址。

Docker使用代码network = addr.(*net.IPNet)获取网桥设备的网络地址。

3.3.7 配置Docker Daemon的iptables

创建完网桥之后，Docker Daemon为容器以及宿主机配置iptables，包括为container之间所需要的link操作提供支持，为host主机上所有的对外对内流量制定传输规则等。该部分详情可以参看《Docker源码分析（四）：Docker Daemon之NewDaemon实现》。代码位于./docker/daemon/networkdriver/bridge/driver/driver.go#L133，如下：

```
// Configure iptables for link support
if enableIPTables {
        if err := setupIPTables(addr, icc); err != nil {
            return job.Error(err)
        }
}

// We can always try removing the iptables
if err := iptables.RemoveExistingChain("DOCKER"); err != nil {
        return job.Error(err)
}

if enableIPTables {
        chain, err := iptables.NewChain("DOCKER", bridgeIface)
        if err != nil {
            return job.Error(err)
        }
        portmapper.SetIptablesChain(chain)
}
```

3.3.8 配置网络设备间数据报转发功能

在Linux系统上，数据包转发功能是被默认禁止的。数据包转发，就是当host主机存在多个网络设备时，如果其中一个接收到数据包，并需要将其转发给另外的网络设备。通过修改/proc/sys/net/ipv4/ip_forward的值，将其置为1，则可以保证系统内数据包可以实现转发功能，代码如下：

```
if ipForward {
        // Enable IPv4 forwarding
        if err := ioutil.WriteFile("/proc/sys/net/ipv4/ip_forward", []byte{'1', '\n'}, 0644); err != nil {
            job.Logf("WARNING: unable to enable IPv4 forwarding: %s\n", err)
        }
}
```

3.3.9 注册网络Handler

创建Docker Daemon网络环境的最后一个步骤是：注册4个与网络相关的Handler。这4个Handler分别是

allocate_interface、release_interface、allocate_port和link，作用分别是为Docker Container分配网络设备，回收Docker Container网络设备、为Docker Container分配端口资源、以及为Docker Container间执行link操作。

至此，Docker Daemon的网络环境初始化工作全部完成。

4 总结

在工业界，Docker的网络问题备受关注。Docker的网络环境可以分为Docker Daemon网络和Docker Container网络。本文从Docker Daemon的网络入手，分析了大家熟知的Docker 桥接模式。

Docker的容器技术以及镜像技术，已经给Docker实践者带来了诸多效益。然而Docker网络的发展依然具有很大的潜力。下一篇Docker Container网络篇，将会带来更为灵活的Docker网络配置。

5 作者介绍

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

6 参考文献

http://www.cnblogs.com/iceocean/articles/1594195.html

http://docs.studygolang.com/pkg/net/

（七）：Docker Container网络（上）

- 1.前言(什么是Docker Container)
- 2.Docker Container网络分析内容安排
- 3.Docker Container网络模式
  - 3.1 bridge桥接模式
  - 3.2 host模式
  - 3.3 other container模式
  - 3.4 none模式
- 4.作者介绍
- 5.下期预告

1.前言(什么是Docker Container)

如今，Docker技术大行其道，大家在尝试以及玩转Docker的同时，肯定离不开一个概念，那就是“容器”或者“Docker Container”。那么我们首先从实现的角度来看看“容器”或者“Docker Container”到底为何物。

逐渐熟悉Docker之后，大家肯定会深深得感受到：应用程序在Docker Container内部的部署与运行非常便捷，只要有Dockerfile，应用一键式的部署运行绝对不是天方夜谭； Docker Container内运行的应用程序可以受到资源的控制与隔离，大大满足云计算时代应用的要求。毋庸置疑，Docker的这些特性，传统模式下应用是完全不具备的。然而，这些令人眼前一亮的特性背后，到底是谁在“作祟”，到底是谁可以支撑Docker的这些特性？不知道这个时候，大家是否会联想到强大的Linux内核。

其实，这很大一部分功能都需要归功于Linux内核。那我们就从Linux内核的角度来看看Docker到底为何物，先从Docker Container入手。关于Docker Container，体验过的开发者第一感觉肯定有两点：内部可以跑应用（进程），以及提供隔离的环境。当然，后者肯定也是工业界称之为“容器”的原因之一。

既然Docker Container内部可以运行进程，那么我们先来看Docker Container与进程的关系，或者容器与进程的关系。首先，我提出这样一个问题供大家思考“容器是否可以脱离进程而存在”。换句话说，能否创建一个容器，而这个容器内部没有任何进程。

可以说答案是否定的。既然答案是否定的，那说明不可能先有容器，然后再有进程，那么问题又来了，“容器和进程是一起诞生，还是先有进程再有容器呢？”可以说答案是后者。以下将慢慢阐述其中的原因。

阐述问题“容器是否可以脱离进程而存在”的原因前，相信大家对于以下的一段话不会持有异议：通过Docker创建出的一个Docker Container是一个容器，而这个容器提供了进程组隔离的运行环境。那么问题在于，容器到底是通过何种途径来实现进程组运行环境的“隔离”。这时，就轮到Linux内核技术隆重登场了。

说到运行环境的“隔离”，相信大家肯定对Linux的内核特性namespace和cgroup不会陌生。namespace主要负责命名空间的隔离，而cgroup主要负责资源使用的限制。其实，正是这两个神奇的内核特性联合使用，才保证了Docker Container的“隔离”。那么，namespace和cgroup又和进程有什么关系呢？问题的答案可以用以下的次序来说明：

(1) 父进程通过fork创建子进程时，使用namespace技术，实现子进程与其他进程（包含父进程）的命名空间隔离；

(2) 子进程创建完毕之后，使用cgroup技术来处理子进程，实现进程的资源使用限制；

(3) 系统在子进程所处namespace内部，创建需要的隔离环境，如隔离的网络栈等；

(4) namespace和cgroup两种技术都用上之后，进程所处的“隔离”环境才真正建立，这时“容器”才真正诞生！

从Linux内核的角度分析容器的诞生，精简的流程即如以上4步，而这4个步骤也恰好巧妙的阐述了namespace和cgroup这两种技术和进程的关系，以及进程与容器的关系。进程与容器的关系，自然是：容器不能脱离进程而存在，先有进程，后有容器。然而，大家往往会说到“使用Docker创建Docker Container（容器），然后在容器内部运行进程”。对此，从通俗易懂的角度来讲，这完全可以理解，因为“容器”一词的存在，本身就较为抽象。如果需要更为准确的表述，那么可以是：“使用Docker创建一个进程，为这个进程创建隔离的环境，这样的环境可以称为Docker Container（容器），然后再在容器内部运行用户应用进程。”当然，笔者的本意不是想否定很多人对于Docker Container或者容器的认识，而是希望和读者一起探讨Docker Container底层技术实现的原理。

对于Docker Container或者容器有了更加具体的认识之后，相信大家的眼球肯定会很快定位到namespace和cgroup这两种技术。Linux内核的这两种技术，竟然能起到如此重大的作用，不禁为之赞叹。那么下面我们就从Docker Container实现流程的角度简要介绍这两者。

首先讲述一下namespace在容器创建时的用法，首先从用户创建并启动容器开始。当用户创建并启动容器时，Docker Daemon 会fork出容器中的第一个进程A（暂且称为进程A，也就是Docker Daemon的子进程）。Docker Daemon执行fork时，在clone系统调用阶段会传入5个参数标志CLONE_NEWNS、CLONE_NEWUTS、CLONE_NEWIPC、CLONE_NEWPID和CLONE_NEWNET（目前Docker 1.2.0还没有完全支持user namespace）。Clone系统调用一旦传入了这些参数标志，子进程将不再与父进程共享相同的命名空间（namespace），而是由Linux为其创建新的命名空间（namespace），从而保证子进程与父进程使用隔离的环境。另外，如果子进程A再次fork出子进程B和C，而fork时没有传入相应的namespace参数标志，那么此时子进程B和C将会与A共享同一个命令空间（namespace）。如果Docker Daemon再次创建一个Docker Container，容器内第一个进程为D，而D又fork出子进程E和F，那么这三个进程也会处于另外一个新的namespace。两个容器的namespace均与Docker Daemon所在的namespace不同。Docker关于namespace的简易示意图如下：

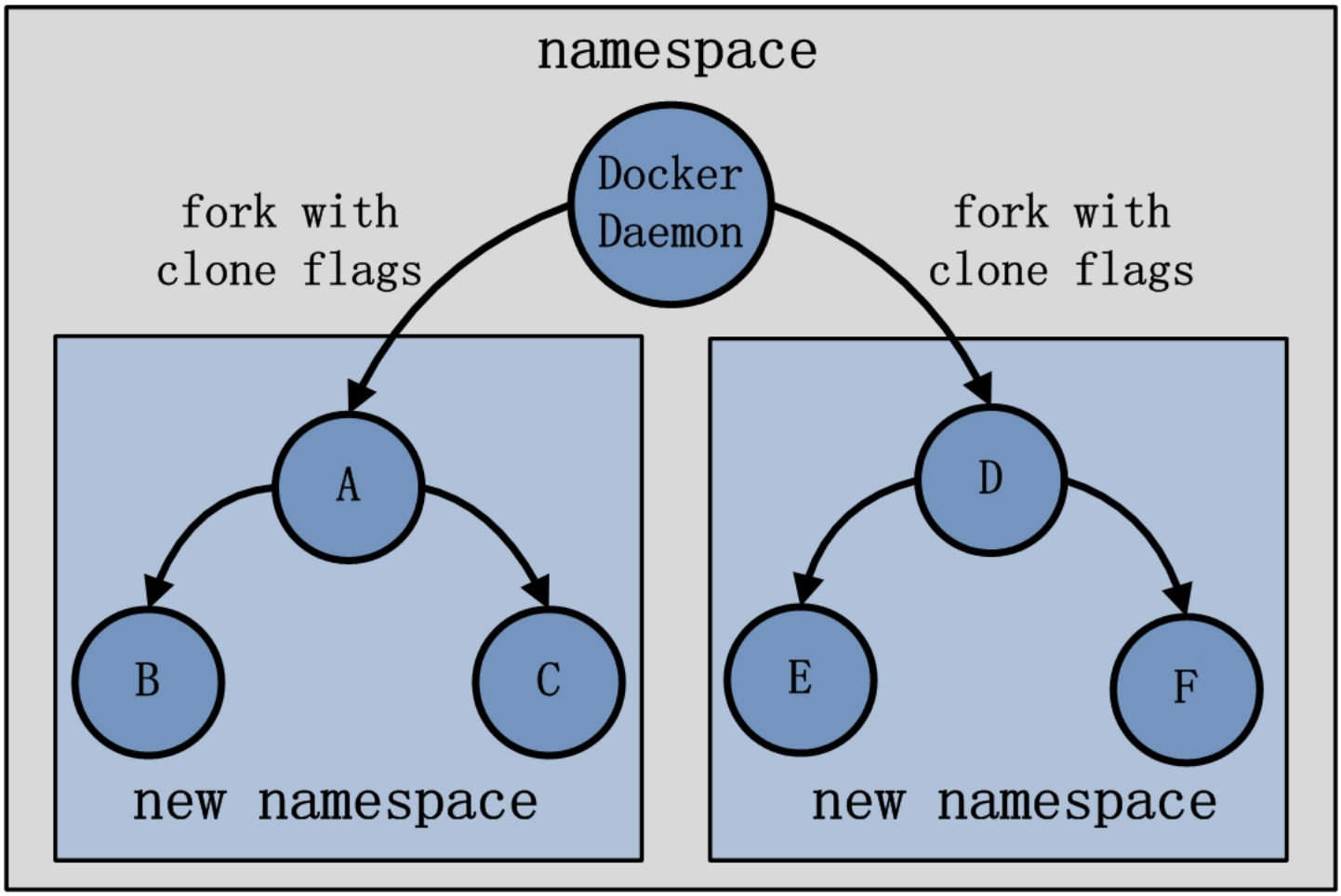


图1.1 Docker中namespace示意图

再说起cgroup，大家都知道可以使用cgroup为进程组做资源的控制。与namespace不同的是，cgroup的使用并不是在创建容器内进程时完成的，而是在创建容器内进程之后再使用cgroup，使得容器进程处于资源控制的状态。换言之，cgroup的运用必须要等到容器内第一个进程被真正创建出来之后才能实现。当容器内进程被创建完毕，Docker Daemon可以获知容器内进程的PID信息，随后将该PID放置在cgroup文件系统的指定位置，做相应的资源限制。

可以说Linux内核的namespace和cgroup技术，实现了资源的隔离与限制。那么对于这种隔离与受限的环境，是否还需要配置其他必需的资源呢。这回答案是肯定的，网络栈资源就是在此时为容器添加。当为容器进程创建完隔离的运行环境时，发现容器虽然已经处于一个隔离的网络环境（即新的network namespace），但是进程并没有独立的网络栈可以使用，如独立的网络接口设备等。此时，Docker Daemon会将Docker Container所需要的资源一一为其配备齐全。网络方面，则需要按照用户指定的网络模式，配置Docker Container相应的网络资源。

2.Docker Container网络分析内容安排

Docker Container网络篇将从源码的角度，分析Docker Container从无到有的过程中，Docker Container网络创建的来龙去脉。Docker Container网络创建流程可以简化如下图：

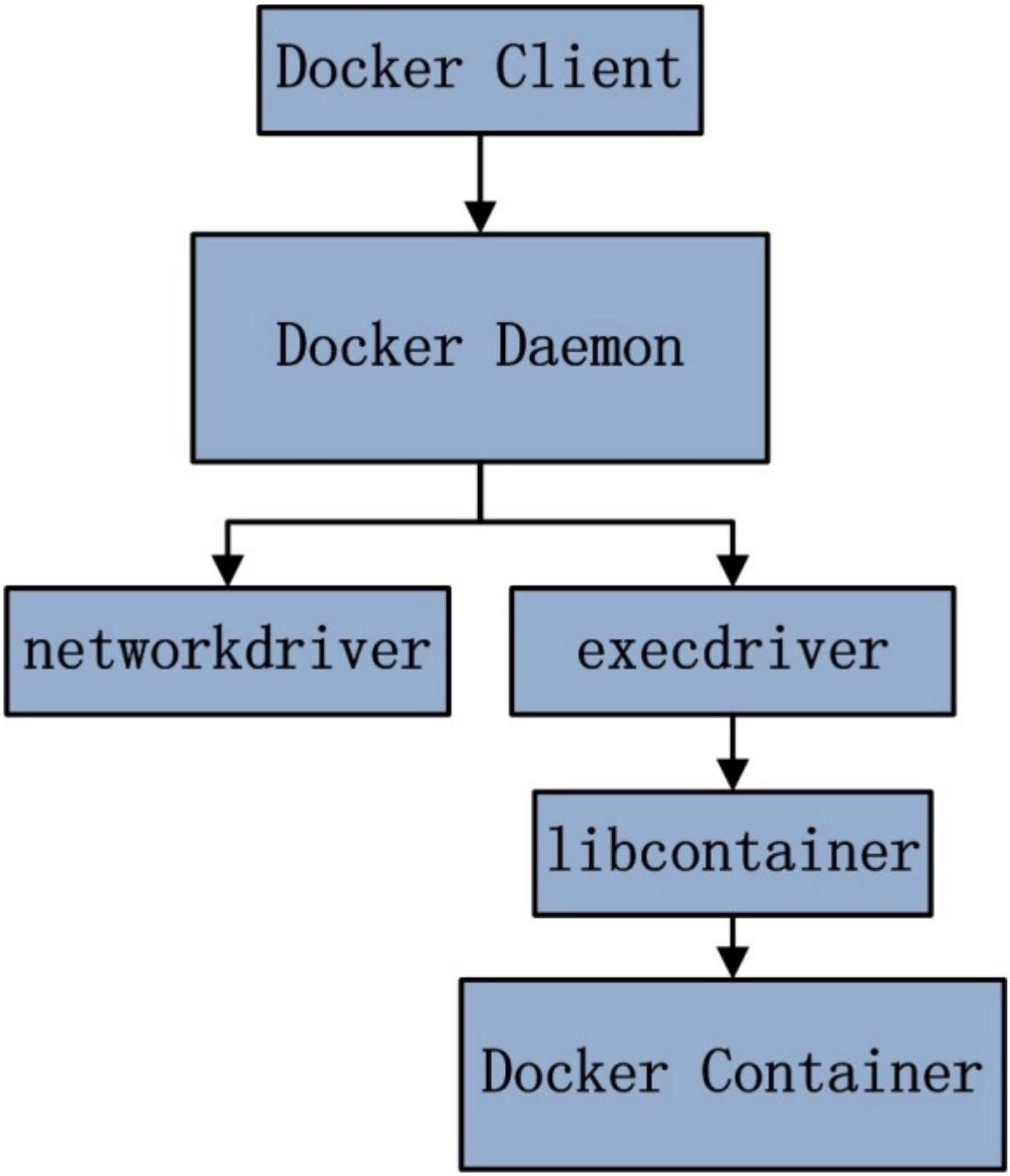


图2.1 Docker Container网络创建流程图

Docker Container网络篇分析的主要内容有以下5部分：

(1) Docker Container的网络模式；

(2) Docker Client配置容器网络；

(3) Docker Daemon创建容器网络流程；

(4) execdriver网络执行流程；

(5) libcontainer实现内核态网络配置。

Docker Container网络创建过程中，networkdriver模块使用并非是重点，故分析内容中不涉及networkdriver。这里不少读者肯定会有疑惑。需要强调的是，networkdriver在Docker中的作用：第一，为Docker Daemon创建网络环境的时候，初始化Docker Daemon的网络环境（详情可以查看

《Docker源码分析》系列第六篇），比如创建docker0网桥等；第二，为Docker Container分配IP地址，为Docker Container做端口映射等。而与Docker Container网络创建有关的内容极少，只有在桥接模式下，为Docker Container的网络接口设备分配一个可用IP地址。

本文为《Docker源码分析》系列第七篇——Docker Container网络（上）。

3.Docker Container网络模式

正如在上文提到的，Docker可以为Docker Container创建隔离的网络环境，在隔离的网络环境下，Docker Container独立使用私有网络。相信很多的Docker开发者也是体验过Docker这方面的网络特性。

其实，Docker除了可以为Docker Container创建隔离的网络环境之外，同样有能力为Docker Container创建共享的网络环境。换言之，当开发者需要Docker Container与宿主机或者其他容器网络隔离时，Docker可以满足这样的需求；而当开发者需要Docker Container与宿主机或者其他容器共享网络时，Docker同样可以满足这样的需求。另外，Docker还可以不为Docker Container创建网络环境。

总结Docker Container的网络，可以得出4种不同的模式：bridge桥接模式、host模式、other container模式和none模式。以下初步介绍4中不同的网络模式。

3.1 bridge桥接模式

Docker Container的bridge桥接模式可以说是目前Docker开发者最常使用的网络模式。Brdige桥接模式为Docker Container创建独立的网络栈，保证容器内的进程组使用独立的网络环境，实现容器间、容器与宿主机之间的网络栈隔离。另外，Docker通过宿主机上的网桥(docker0)来连通容器内部的网络栈与宿主机的网络栈，实现容器与宿主机乃至外界的网络通信。

Docker Container的bridge桥接模式可以参考下图：

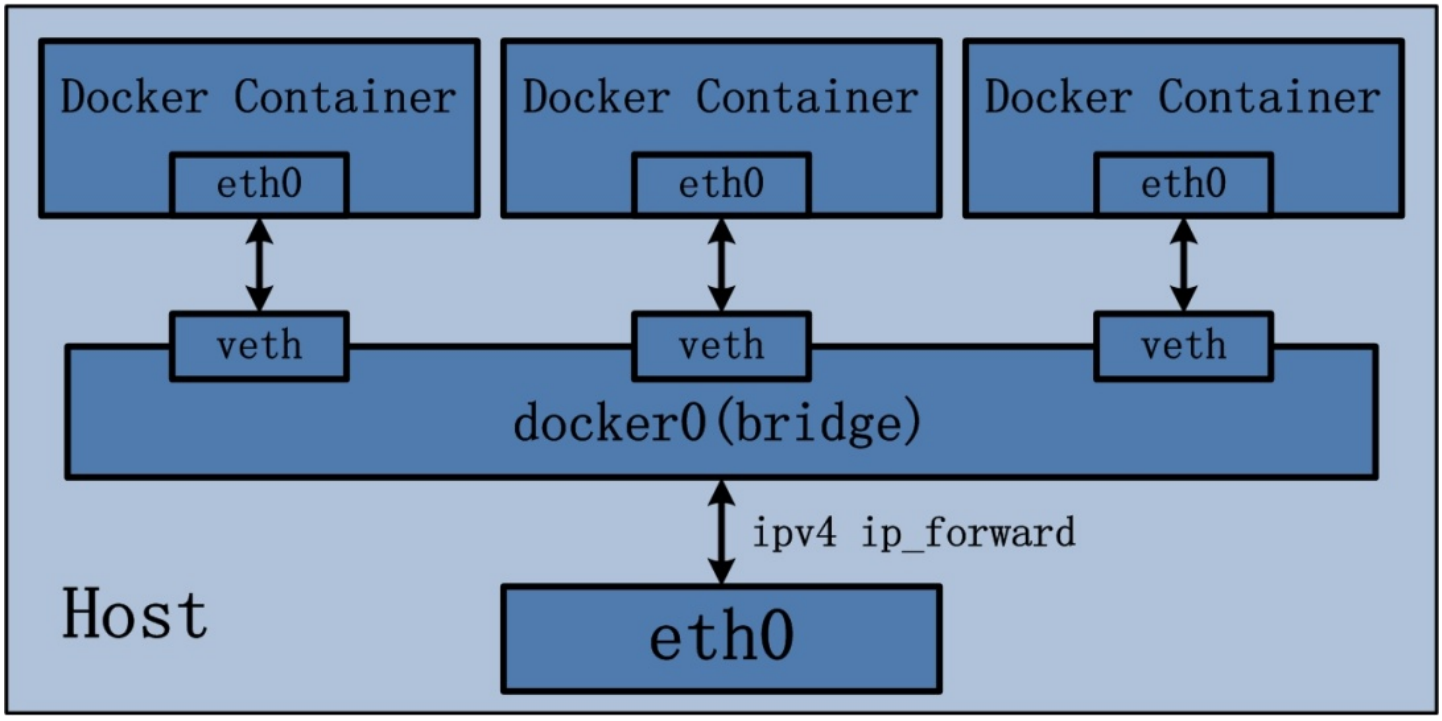


图3.1 Docker Container Bridge桥接模式示意图

Bridge桥接模式的实现步骤主要如下：

(1) Docker Daemon利用veth pair技术，在宿主机上创建两个虚拟网络接口设备，假设为veth0和veth1。而veth pair技术的特性可以保证无论哪一个veth接收到网络报文，都会将报文传输给另一方。

(2) Docker Daemon将veth0附加到Docker Daemon创建的docker0网桥上。保证宿主机的网络报文可以发往veth0；

(3) Docker Daemon将veth1添加到Docker Container所属的namespace下，并被改名为eth0。如此一来，保证宿主机的网络报文若发往veth0，则立即会被eth0接收，实现宿主机到Docker Container网络的联通性；同时，也保证Docker Container单独使用eth0，实现容器网络环境的隔离性。

Bridge桥接模式，从原理上实现了Docker Container到宿主机乃至其他机器的网络连通性。然而，由于宿主机的IP地址与veth pair的 IP地址均不在同一个网段，故仅仅依靠veth pair和namespace的技术，还不足以是宿主机以外的网络主动发现Docker Container的存在。为了使得Docker Container可以让宿主机以外的世界感知到容器内部暴露的服务，Docker采用NAT（Network Address Translation，网络地址转换）的方式，让宿主机以外的世界可以主动将网络报文发送至容器内部。

具体来讲，当Docker Container需要暴露服务时，内部服务必须监听容器IP和端口号port_0，以便外界主动发起访问请求。由于宿主机以外的世界，只知道宿主机eth0的网络地址，而并不知道Docker Container的IP地址，哪怕就算知道Docker Container的IP地址，从二层网络的角度来讲，外界也无法直接通过Docker Container的IP地址访问容器内部应用。因此，Docker使用NAT方法，将容器内部的服务监听的端口与宿主机的某一个端口port_1进行“绑定”。

如此一来，外界访问Docker Container内部服务的流程为：

(1) 外界访问宿主机的IP以及宿主机的端口port_1；

(2) 当宿主机接收到这样的请求之后，由于DNAT规则的存在，会将该请求的目的IP（宿主机eth0的IP）和目的端口port_1进行转换，转换为容器IP和容器的端口port_0;

(3) 由于宿主机认识容器IP，故可以将请求发送给veth pair；

(4) veth pair的veth0将请求发送至容器内部的eth0，最终交给内部服务进行处理。

使用DNAT方法，可以使得Docker宿主机以外的世界主动访问Docker Container内部服务。那么Docker Container如何访问宿主机以外的世界呢。以下简要分析Docker Container访问宿主机以外世界的流程：

(1) Docker Container内部进程获悉宿主机以外服务的IP地址和端口port_2，于是Docker Container发起请求。容器的独立网络环境保证了请求中报文的源IP地址为容器IP（即容器内部eth0），另外Linux内核会自动为进程分配一个可用源端口（假设为port_3）；

(2) 请求通过容器内部eth0发送至veth pair的另一端，到达veth0，也就是到达了网桥（docker0）处；

(3) docker0网桥开启了数据报转发功能（/proc/sys/net/ipv4/ip_forward），故将请求发送至宿主机的eth0处；

(4) 宿主机处理请求时，使用SNAT对请求进行源地址IP转换，即将请求中源地址IP（容器IP地址）转换为宿主机eth0的IP地址；

(5) 宿主机将经过SNAT转换后的报文通过请求的目的IP地址（宿主机以外世界的IP地址）发送至外界。

在这里，很多人肯定会问：对于Docker Container内部主动发起对外的网络请求，当请求到达宿主机进行SNAT处理后发给外界，当外界响应请求时，响应报文中的目的IP地址肯定是Docker宿主机的IP地址，那响应报文回到宿主机的时候，宿主机又是如何转给Docker Container的呢？关于这样的响应，由于port_3端口并没有在宿主机上做相应的DNAT转换，原则上不会被发送至容器内部。为什么说对于这样的响应，不会做DNAT转换呢。原因很简单，DNAT转换是针对容器内部服务监听的特定端口做的，该端口是供服务监听使用，而容器内部发起的请求报文中，源端口号肯定不会占用服务监听的端口，故容器内部发起请求的响应不会在宿主机上经过DNAT处理。

其实，这一环节的内容是由iptables规则来完成，具体的iptables规则如下：

```
iptables -I FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

这条规则的意思是，在宿主机上发往docker0网桥的网络数据报文，如果是该数据报文所处的连接已经建立的话，则无条件接受，并由Linux内核将其发送到原来的连接上，即回到Docker Container内部。

以上便是Docker Container中bridge桥接模式的简要介绍。可以说，bridger桥接模式从功能的角度实现了两个方面：第一，让容器拥有独立、隔离的网络栈；第二，让容器和宿主机以外的世界通过NAT建立通信。

然而，bridge桥接模式下的Docker Container在使用时，并非为开发者包办了一切。最明显的是，该模式下Docker Container不具有一个公有IP，即和宿主机的eth0不处于同一个网段。导致的结果是宿主机以外的世界不能直接和容器进行通信。虽然NAT模式经过中间处理实现了这一点，但是NAT模式仍然存在问题与不便，如：容器均需要在宿主机上竞争端口，容器内部服务的访问者需要使用服务发现获知服务的外部端口等。另外NAT模式由于是在三层网络上的实现手段，故肯定会影响网络的传输效率。

3.2 host模式

Docker Container中的host模式与bridge桥接模式有很大的不同。最大的区别当属，host模式并没有为容器创建一个隔离的网络环境。而之所以称之为host模式，是因为该模式下的Docker Container会和host宿主机共享同一个网络namespace，故Docker Container可以和宿主机一样，使用宿主机的eth0，实现和外界的通信。换言之，Docker Container的IP地址即为宿主机eth0的IP地址。

Docker Container的host网络模式可以参考下图：

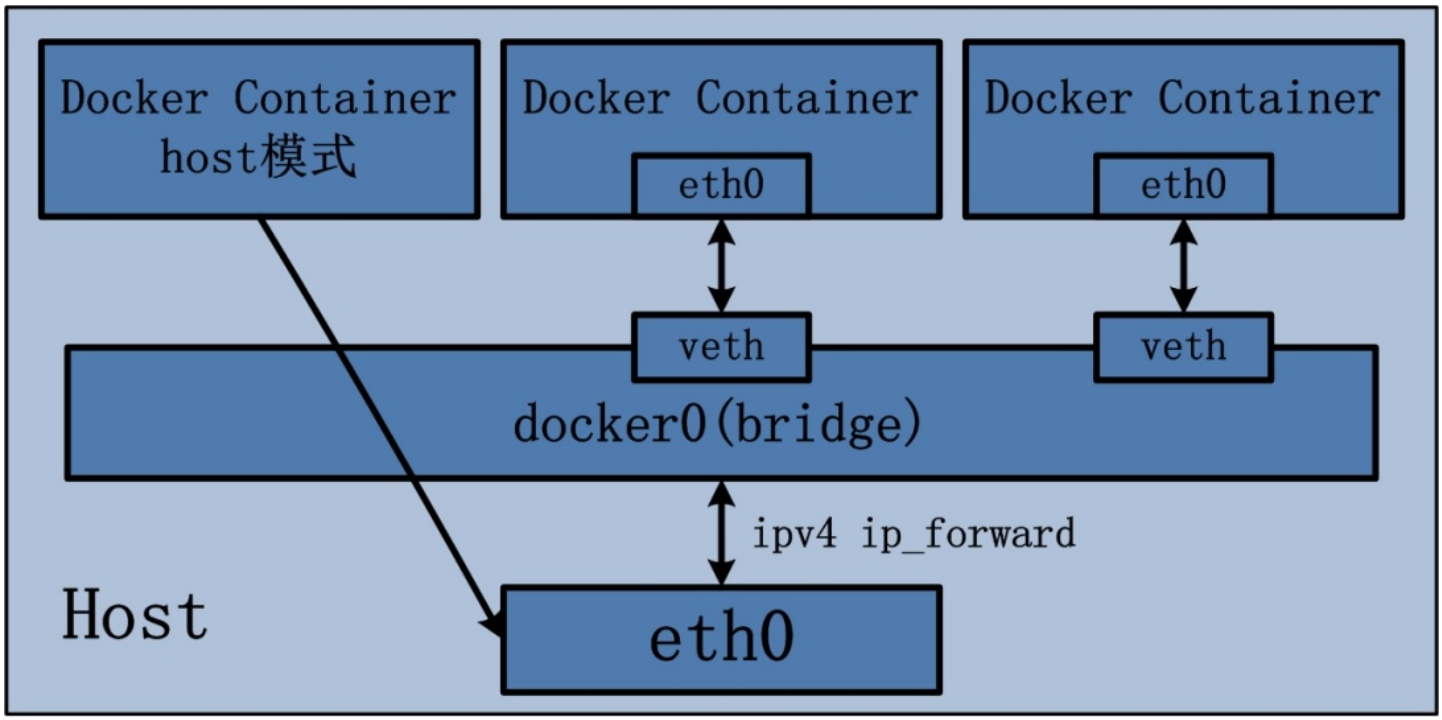


图3.2 Docker Container host网络模式示意图

上图最左侧的Docker Container，即采用了host网络模式，而其他两个Docker Container依然沿用brdige桥接模式，两种模式同时存在于宿主机上并不矛盾。

Docker Container的host网络模式在实现过程中，由于不需要额外的网桥以及虚拟网卡，故不会涉及docker0以及veth pair。上文namespace的介绍中曾经提到，父进程在创建子进程时，如果不使用CLONE_NEWNET这个参数标志，那么创建出的子进程会与父进程共享同一个网络namespace。Docker就是采用了这个简单的原理，在创建进程启动容器的过程中，没有传入CLONE_NEWNET参数标志，实现Docker Container与宿主机共享同一个网络环境，即实现host网络模式。

可以说，Docker Container的网络模式中，host模式是bridge桥接模式很好的补充。采用host模式的Docker Container，可以直接使用宿主机的IP地址与外界进行通信，若宿主机的eth0是一个公有IP，那么容器也拥有这个公有IP。同时容器内服务的端口也可以使用宿主机的端口，无需额外进行NAT转换。当然，有这样的方便，肯定会损失部分其他的特性，最明显的是Docker Container网络环境隔离性的弱化，即容器不再拥有隔离、独立的网络栈。另外，使用host模式的Docker Container虽然可以让容器内部的服务和传统情况无差别、无改造的使用，但是由于网络隔离性的弱化，该容器会与宿主机共享竞争网络栈的使用；另外，容器内部将不再拥有所有的端口资源，原因是部分端口资源已经被宿主机本身的服务占用，还有部分端口已经用以bridge网络模式容器的端口映射。

3.3 other container模式

Docker Container的other container网络模式是Docker中一种较为特别的网络的模式。之所以称为“other container模式”，是因为这个模式下的Docker Container，会使用其他容器的网络环境。之所以称为“特别”，是因为这个模式下容器的网络隔离性会处于bridge桥接模式与host模式之间。Docker Container共享其他容器的网络环境，则至少这两个容器之间不存在网络隔离，而这两个容器又与宿主机以及除此之外其他的容器存在网络隔离。

Docker Container的other container网络模式可以参考下图：

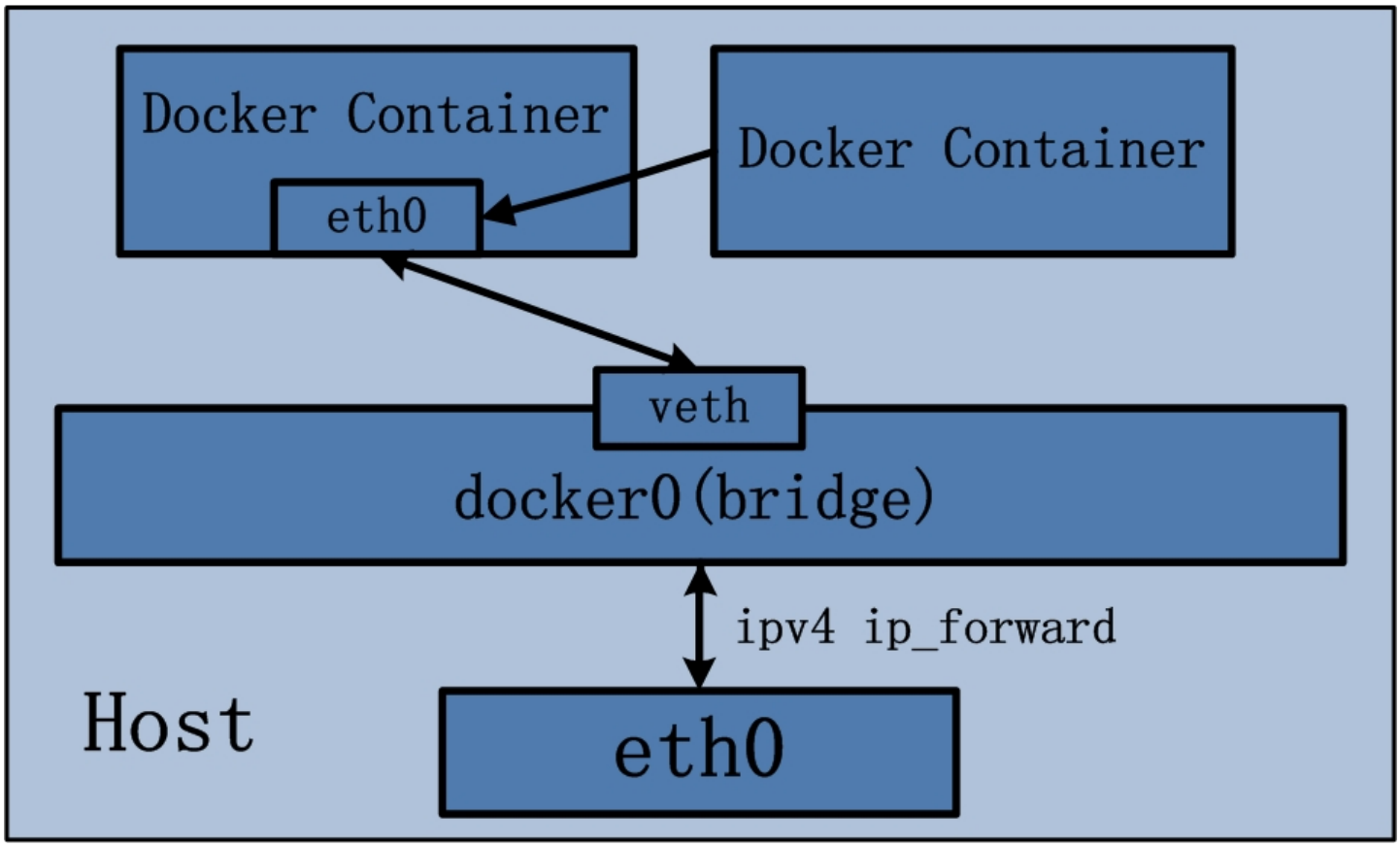


图3.3 Docker Container other container网络模式示意图

上图右侧的Docker Container即采用了other container网络模式，它能使用的网络环境即为左侧Docker Container brdige桥接模式下的网络。

Docker Container的other container网络模式在实现过程中，不涉及网桥，同样也不需要创建虚拟网卡veth pair。完成other container网络模式的创建只需要两个步骤：

(1) 查找other container（即需要被共享网络环境的容器）的网络namespace；

(2) 将新创建的Docker Container（也是需要共享其他网络的容器）的namespace，使用other container的namespace。

Docker Container的other container网络模式，可以用来更好的服务于容器间的通信。

在这种模式下的Docker Container可以通过localhost来访问namespace下的其他容器，传输效率较高。虽然多个容器共享网络环境，但是多个容器形成的整体依然与宿主机以及其他容器形成网络隔离。另外，这种模式还节约了一定数量的网络资源。但是需要注意的是，它并没有改善容器与宿主机以外世界通信的情况。

3.4 none模式

Docker Container的第四种网络模式是none模式。顾名思义，网络环境为none，即不为Docker Container任何的网络环境。一旦Docker Container采用了none网络模式，那么容器内部就只能使用loopback网络设备，不会再有其他的网络资源。

可以说none模式为Docker Container做了极少的网络设定，但是俗话说得好“少即是多”，在没有网络配置的情况下，作为Docker开发者，才能在这基础做其他无限多可能的网络定制开发。这也恰巧体现了Docker设计理念的开放。

4.作者介绍

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

5.下期预告

下期内容为：Docker源码分析（八）：Docker Container网络（下）

（八）：Docker Container网络（下）

- 1.Docker Client配置容器网络模式
  - 1.1 Docker Client使用
  - 1.2 runconfig包解析
  - 1.3 CmdRun执行
- 2.Docker Daemon创建容器网络流程
  - 2.1创建容器并配置网络参数
  - 2.2启动容器之网络配置
- 3.execdriver网络执行流程
  - 3.1创建libcontainer的Config对象
  - 3.2 调用libcontainer的namespaces启动容器
- 4.libcontainer实现内核态网络配置
  - 4.1创建exec.Cmd
  - 4.2启动exec.Cmd创建进程
  - 4.3为容器进程初始化网络环境
- 5.总结
- 6.作者介绍
- 7.参考文献

1.Docker Client配置容器网络模式

Docker目前支持4种网络模式，分别是bridge、host、container、none，Docker开发者可以根据自己的需求来确定最适合自己应用场景的网络模式。

从Docker Container网络创建流程图中可以看到，创建流程第一个涉及的Docker模块即为Docker Client。当然，这也十分好理解，毕竟Docker Container网络环境的创建需要由用户发起，用户根据自身对容器的需求，选择网络模式，并将其通过Docker Client传递给Docker Daemon。本节，即从Docker Client源码的角度，分析如何配置Docker Container的网络模式，以及Docker Client内部如何处理这些网络模式参数。

需要注意的是：配置Docker Container网络环境与创建Docker Container网络环境有一些区别。区别是：配置网络环境指用户通过向Docker Client传递网络参数，实现Docker Container网络环境参数的配置，这部分配置由Docker Client传递至Docker Daemon，并由Docker Daemon保存；创建网络环境指，用户通过Docker Client向Docker Daemon发送容器启动命令之后，Docker Daemon根据之前保存的网络参数，实现Docker Container的启动，并在启动过程中完成Docker Container网络环境的创建。

以上的基本知识，理解下文的Docker Container网络环境创建流程。

1.1 Docker Client使用

Docker架构中，用户可以通过Docker Client来配置Docker Container的网络模式。配置过程主要通过docker run命令来完成，实现配置的方式是在docker run命令中添加网络参数。使用方式如下（其中NETWORKMODE为四种网络模式之一，ubuntu为镜像名称，/bin/bash为执行指令）：

```
docker run -d --net NETWORKMODE ubuntu /bin/bash
```

运行以上命令时，首先创建一个Docker Client，然后Docker Client会解析整条命令的请求内容，接着解析出为run请求，意为运行一个Docker Container，最终通过Docker Client端的API接口，调用CmdRun函数完成run请求执行。（详情可以查阅《Docker源码分析》系列的第二篇——Docker Client篇）。

Docker Client解析出run命令之后，立即调用相应的处理函数CmdRun进行处理关于run请求的具体内容。CmdRun的作用主要可以归纳为三点：

- 解析Docker Client传入的参数，解析出config、hostconfig和cmd对象等；
- 发送请求至Docker Daemon，创建一个container对象，完成Docker Container启动前的准备工作；
- 发送请求至Docker Daemon，启动相应的Docker Container（包含创建Docker Container网络环境创建）。

1.2 runconfig包解析

CmdRun函数的实现位于./docker/api/client/commands.go。CmdRun执行的第一个步骤为：通过runconfig包中ParseSubcommand函数解析Docker Client传入的参数，并从中解析出相应的config，hostConfig以及cmd对象，实现代码如下：

```
config, hostConfig, cmd, err := runconfig.ParseSubcommand
(cli.Subcmd("run", "[OPTIONS] IMAGE [COMMAND] [ARG...]",
"Run a command in a new container"), args, nil)
```

其中，config的类型为Config结构体，hostConfig的类型为HostConfig结构体，两种类型的定义均位于runconfig包。Config与HostConfig类型同用以描述Docker Container的配置信息，然而两者之间又有着本质的区别，最大的区别在于两者各自的作用范畴：

- Config结构体：描述Docker Container独立的配置信息。独立的含义是：Config这部分信息描述的是容器本身，而不会与容器所在host宿主机相关；
- HostConfig结构体：描述Docker Container与宿主机相关的配置信息。

1.2.1 Config结构体

Config结构体描述Docker Container本身的属性信息，这些信息与容器所在的host宿主机无关。结构体的定义如下：

```
type Config struct {
    Hostname        string
    Domainname      string
    User            string
    Memory          int64  // Memory limit (in bytes)
    MemorySwap      int64  // Total memory usage (memory + swap); set `-1' to disable swap
    CpuShares       int64  // CPU shares (relative weight vs. other containers)
    Cpuset          string // Cpuset 0-2, 0,1
    AttachStdin     bool
    AttachStdout    bool
    AttachStderr    bool
    PortSpecs       []string // Deprecated - Can be in the format of 8080/tcp
    ExposedPorts    map[nat.Port]struct{}
    Tty             bool // Attach standard streams to a tty, including stdin if it is not closed.
    OpenStdin       bool // Open stdin
    StdinOnce       bool // If true, close stdin after the 1 attached client disconnects.
    Env             []string
    Cmd             []string
    Image           string // Name of the image as it was passed by
the operator (eg. could be symbolic)
    Volumes         map[string]struct{}
    WorkingDir      string
    Entrypoint      []string
    NetworkDisabled bool
    OnBuild         []string
}
```

Config结构体中各属性的详细解释如下表：

Config结构体属性名	**类型**	**代表含义**
Hostname	string	容器主机名
Domainname	string	域名名称
User	string	用户名
Memory	int64	容器内存使用上限（单位：字节）
MemorySwap	int64	容器所有的内存使用上限（物理内存+交互区），关闭交互区支持置为-1
CpuShares	int64	容器CPU使用share值，其他容器的相对值
Cpuset	string	CPU核的使用集合
AttachStdin	bool	是否附加标准输入
AttachStdout	bool	是否附加标准输出
AttachStderr	bool	是否附加标准错误输出
PortsSpecs	[]string	目前已被遗弃

Config结构体属性名	类型	代表含义
ExposedPorts	map[nat.Port]struct{}	容器内部暴露的端口号
Tty	bool	是否分配一个伪终端tty
OpenStdin	bool	在没有附加标准输入时，是否依然打开标准输入
StdinOnce	bool	若为真，表示第一个客户关闭标准输入后关闭标准输入功能
Env	[]string	容器进程运行的环境变量
Cmd	[]string	容器内通过ENTRYPOINT运行的指令
Image	string	容器rootfs所依赖的镜像名称
Volumes	map[string]struct{}	容器需要从host宿主机上挂载的目录
WorkingDir	string	容器内部的指定工作目录
Entrypoint	[]string	覆盖镜像属性中默认的ENTRYPOINT
NetworkDisabled	bool	是否关闭容器网络功能
OnBuild	[]string	

1.2.2 HostConfig结构体

HostConfig结构体描述Docker Container与宿主机相关的属性信息，结构体的定义如下：

```
type HostConfig struct {
    Binds           []string
    ContainerIDFile string
    LxcConf         []utils.KeyValuePair
    Privileged      bool
    PortBindings    nat.PortMap
    Links           []string
    PublishAllPorts bool
    Dns             []string
    DnsSearch       []string
    VolumesFrom     []string
    Devices         []DeviceMapping
    NetworkMode     NetworkMode
    CapAdd          []string
    CapDrop         []string
    RestartPolicy   RestartPolicy
}
```

Config结构体中各属性的详细解释如下表：

HostConfig结构体属性名	类型	代表含义
Binds	[]string	从宿主机上绑定到容器的volumes
ContainerIDFile	string	文件名，文件用以写入容器的ID
LxcConf	[]utils.KeyValuePair	添加自定义的lxc选项
Privileged	bool	是否赋予该容器扩展权限
PortBindings	nat.PortMap	容器绑定到host宿主机的端口
Links	[]string	添加其他容器的链接到该容器
PublishAllPorts	bool	是否向宿主机暴露所有端口信息
Dns	[]string	自定义的DNS服务器地址
DnsSearch	[]string	自定义的DNS查找服务器地址
VolumesFrom	[]string	从指定的容器中挂载到该容器的volumes
Devices	[]DeviceMapping	为容器添加一个或多个宿主机设备
NetworkMode	NetworkMode	为容器设置的网络模式
CapAdd	[]string	为容器用户添加一个或多个Linux Capabilities
CapDrop	[]string	为容器用户禁用一个或多个Linux Capabilities
RestartPolicy	RestartPolicy	当一个容器异常退出时采取的重启策略

1.2.3 runconfig解析网络模式

讲述完Config与HostConfig结构体之后，回到runconfig包中分析如何解析与Docker Container网络模式相关的配置信息，并将这部分信息传递给config实例与hostConfig实例。

runconfig包中的ParseSubcommand函数调用parseRun函数完成命令请求的分析，实现代码位于./docker/runconfig/parse.go#L37-L39，如下：

```
func ParseSubcommand(cmd *flag.FlagSet, args []string,
   sysInfo *sysinfo.SysInfo) (*Config, *HostConfig, *flag.FlagSet, error) {
       return parseRun(cmd, args, sysInfo)
}
```

进入parseRun函数即可发现：该函数完成了四方面的工作：

- 定义与容器配置信息相关的flag参数；
- 解析docker run命令后紧跟的请求内容，将请求内容全部保存至flag参数中，余下的内容一个为镜像image名，另一个为需要在容器内执行的cmd命令；
- 通过flag参数验证参数的有效性，并处理得到Config结构体与HostConfig结构体需要的属性值；
- 创建并初始化Config类型实例config、HostConfig类型实例hostConfig，最总返回config、hostConfig与cmd。

本文主要分析Docker Container的网络模式，而parseRun函数中有关容器网络模式的flag参数有flNetwork与flNetMode，两者的定义分别位于./docker/runconfig/parse.go#L62与./docker/runconfig/parse.go#L75，如下：

> flNetwork = cmd.Bool([]string{"#n", "#-networking"}, true, "Enable networking for this container")
>
> flNetMode = cmd.String([]string{"-net"}, "bridge", "Set the Network mode for the container\n'bridge': creates a new network stack for the container on the docker bridge\n'none': no networking for this container\n'container:': reuses another container network stack\n'host': use the host network stack inside the container. Note: the host mode gives the container full access to local system services such as D-bus and is therefore considered insecure.")

可见flag参数flNetwork表示是否开启容器的网络模式，若为true则开启，说明需要给容器创建网络环境；否则不开启，说明不给容器赋予网络功能。该flag参数的默认值为true，另外使用该flag的方式为：在docker run之后设定--networking或者-n，如：

```
docker run --networking true ubuntu /bin/bash
```

另一个flag参数flNetMode则表示为容器设定的网络模式，共有四种选项，分别是：bridge、none、container:和host。四种模式的作用上文已经详细解释，此处不再赘述。使用该flag的方式为：在docker run之后设定--net，如：

```
Docker run --net host ubuntu /bin/bash
```

用户使用docker run启动容器时设定了以上两个flag参数（--networking和--net），则runconfig包会解析出这两个flag的值。最终，通过flag参数flNetwork，得到Config类型实例config的属性NetworkDisabled；通过flag参数flNetMode，得到HostConfig类型实例hostConfig的属性NetworkMode。

函数parseRun返回config、hostConfig与cmd，代表着runconfig包解析配置参数工作的完成，CmdRun得到返回内容之后，继续向下执行。

1.3 CmdRun执行

在runconfig包中已经将有关容器网络模式的配置置于config对象与hostConfig对象，故在CmdRun函数的执行中，更多的是基于config对象与hostConfig参数处理配置信息，而没有其他的容器网络处理部分。

CmdRun后续主要工作是：利用Docker Daemon暴露的RESTful API接口，将docker run的请求发送至Docker Daemon。以下是CmdRun执行过程中Docker Client与Docker Daemon的简易交互图。

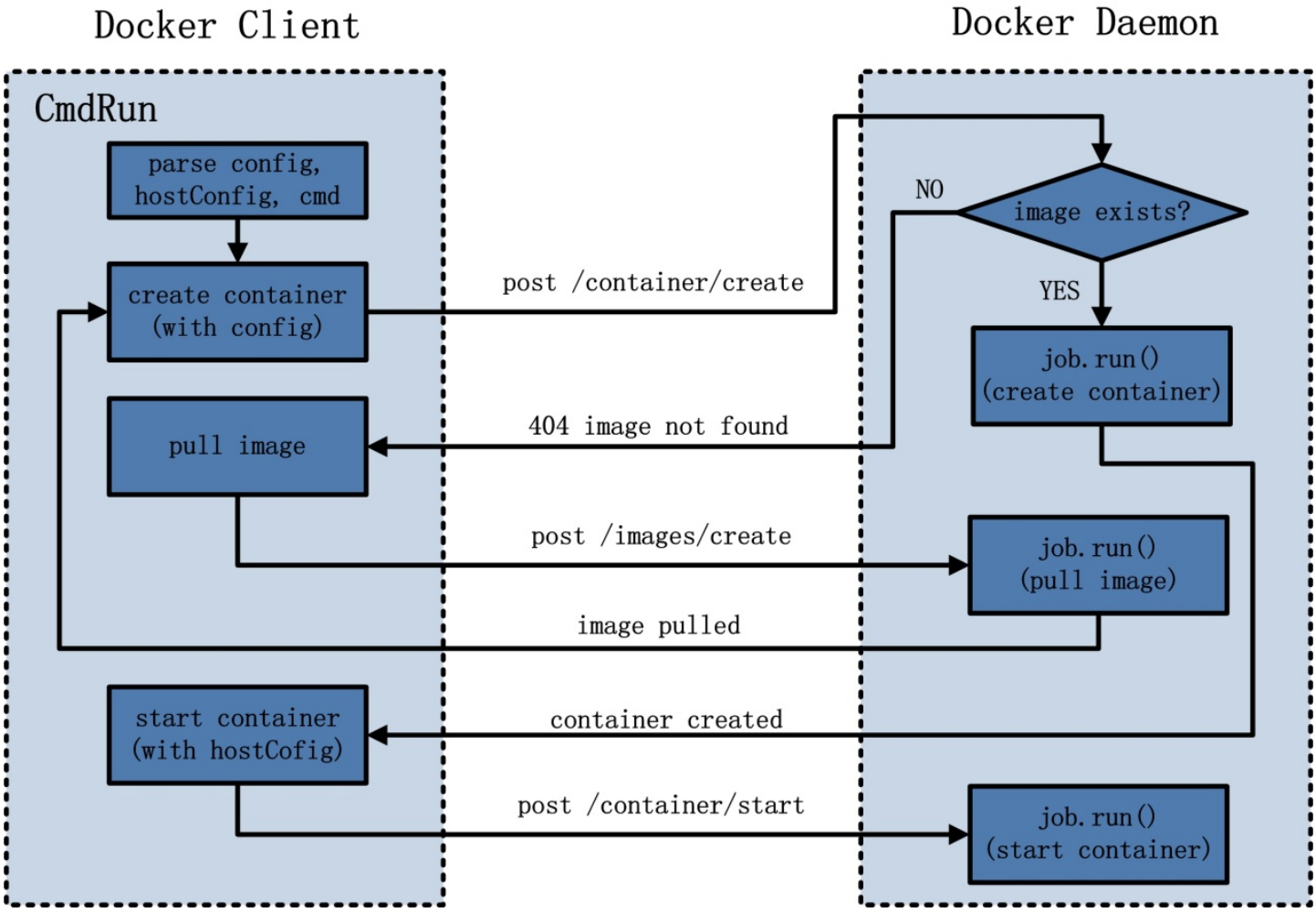


图1.1 CmdRun中Docker Client与Docker Daemon交互图

具体分析CmdRun的执行流程可以发现：在解析config、hostConfig与cmd之后，Docker Client首先发起请求create container。若Docker Daemon节点上已经存在该容器所需的镜像，则立即执行create container操作并返回请求响应；Docker Client收到响应后，再发起请求start container。若容器镜像还不存在，Docker Daemon返回一个404的错误，表示镜像不存在；Docker Client收到错误响应之后，再发起一个请求pull image，使Docker Daemon首先下载镜像，下载完毕之后Docker Client再次发起请求create container，Docker Daemon创建完毕之后，Docker Client最终发起请求start container。

总结而言，Docker Client负责创建容器请求的发起。关于Docker Container网络环境的配置参数存储于config与hostConfig对象之中，在请求create container和start container发起时，随请求一起发送至Docker Daemon。

2.Docker Daemon创建容器网络流程

Docker Daemon接收到Docker Client的请求大致可以分为两次，第一次为create container，第二次为start container。这两次请求的执行过程中，都与Docker Container的网络相关。以下按照这两个请求的执行，具体分析Docker Container网络模式的创建。Docker Daemon如何通过Docker Server解析RESTful请求，并完成分发调度处理，在《Docker源码分析》系列的第五篇——Docker Server篇中已经详细分析过，本文不再赘述。

2.1创建容器并配置网络参数

Docker Daemon首先接收并处理create container请求。需要注意的是：create container并非创建了一个运行的容器，而是完成了以下三个主要的工作：

- 通过runconfig包解析出create container请求中与Docker Container息息相关的config对象；
- 在Docker Daemon内部创建了与Docker Container对应的container对象；
- 完成Docker Container启动前的准备化工作，如准备所需镜像、创建rootfs等。

创建容器过程中，Docker Daemon首先通过runconfig包中ContainerConfigFromJob函数，解析出请求中的config对象，解析过程代码如下：

config := runconfig.ContainerConfigFromJob(job)

至此，Docker Client处理得到的config对象，已经传递至Docker Daemon的config对象，config对象中已经含有属性NetworkDisabled具体值。

处理得到config对象之后，Docker Daemon紧接着创建container对象，并为Docker Container作相应的准备工作。具体的实现代码位于./docker/daemon/create.go#L73-L78，如下：

```
if container, err = daemon.newContainer(name, config, img); err != nil {
    return nil, nil, err
}
if err := daemon.createRootfs(container, img); err != nil {
    return nil, nil, err
}
```

与Docker Container网络模式配置相关的内容主要位于创建container对象中。newContainer函数的定义位于./docker/daemon/daemon.go#L516-L550，具体的container对象如下：

```
container := &Container{
    ID:              id,
    Created:         time.Now().UTC(),
    Path:            entrypoint,
    Args:            args, //FIXME: de-duplicate from config
    Config:          config,
    hostConfig:      &runconfig.HostConfig{},
    Image:           img.ID, // Always use the resolved image id
    NetworkSettings: &NetworkSettings{},
    Name:            name,
    Driver:          daemon.driver.String(),
    ExecDriver:      daemon.execDriver.Name(),
    State:           NewState(),
}
```

在container对象中，config对象直接赋值给container对象的Config属性，另外hostConfig属性与NetworkSeeetings属性均为空。其中hostConfig对象将在start container请求执行过程中被赋

值，NetworkSettings类型的作用是描述容器的网络具体信息，定义位于./docker/daemon/network_settings.go#L11-L18，代码如下：

```
type NetworkSettings struct {
        IPAddress   string
        IPPrefixLen int
        Gateway     string
        Bridge      string
        PortMapping map[string]PortMapping // Deprecated
        Ports       nat.PortMap
}
```

Networksettings类型的各属性的详细解释如下表：

NetworkSettings属性名称	类型	含义
IPAddress	string	IP网络地址
IPPrefixLen	int	网络标识位长度
Gateway	string	网关地址
Bridge	string	网桥地址
PortMapping	map[string]PortMapping	端口映射
Ports	nat.PortMap	端口号

总结而言，Docker Daemon关于create container请求的执行，先实现了容器配置信息从Docker Client至Docker Daemon的转移，再完成了启动容器前所需的准备工作。

2.2启动容器之网络配置

创建容器阶段，Docker Daemon创建了容器对象container，container对象内部的Config属性含有NetworkDisabled。创建容器完成之后，Docker Daemon还需要接收Docker Client的请求，并执行start container的操作，即启动容器。

启动容器过程中，Docker Daemon首先通过runconfig包中ContainerHostConfigFromJob函数，解析出请求中的hostConfig对象，解析过程代码如下：

```
hostConfig := runconfig.ContainerHostConfigFromJob(job)
```

至此，Docker Client处理得到的hostConfig对象，已经传递至Docker Daemon的hostConfig对象，hostConfig对象中已经含有属性NetworkMode具体值。

容器启动的所有工作，均由以下的Start函数来完成，代码位于./docker/daemon/start.go#L36-L38，如下：

```
if err := container.Start(); err != nil {
        return job.Errorf("Cannot start container %s: %s", name, err)
    }
```

Start函数实现了容器的启动。更为具体的描述是：Start函数实现了进程的启动，另外在启动进程的同时为进程设定了命名空间（namespace），启动完毕之后为进程完成了资源使用的控制，从而保证进程以及之后进程的子进程都会在同一个命名空间内，且受到相同的资源控制。如此一来，Start函数创建的进程，以及该进程的子进程，形成一个进程组，该进程组处于资源隔离和资源控制的环境，我们习惯将这样的进程组环境称为容器，也就是这里的Docker Container。

回到Start函数的执行，位于./docker/daemon/container.go#L275-L320。Start函数执行过程中，与Docker Container网络模式相关的部分主要有三部分：

- initializeNetwork()，初始化container对象中与网络相关的属性；
- populateCommand，填充Docker Container内部需要执行的命令，Command中含有进程启动命令，还含有容器环境的配置信息，也包括网络配置；
- container.waitForStart()，实现Docker Container内部进程的启动，进程启动之后，为进程创建网络环境等。

2.2.1初始化容器网络配置

容器对象container中有属性hostConfig，属性hostConfig中有属性NetworkMode，初始化容器网络配置initializeNetworking()的主要工作就是，通过NetworkMode属性为Docker Container的网络作相应的初始化配置工作。

Docker Container的网络模式有四种，分别为：host、other container、none以及bridge。initializeNetworking函数的执行完全覆盖了这四种模式。

initializeNetworking()函数的实现位于./docker/daemon/container.go#L881-L933。

2.2.1.1 初始化host网络模式配置

Docker Container网络的host模式意味着容器使用宿主机的网络环境。虽然Docker Container使用宿主机的网络环境，但这并不代表Docker Container可以拥有宿主机文件系统的视角，而host宿主机上有很多信息标识的是网络信息，故Docker Daemon需要将这部分标识网络的信息，从host宿主机添加到Docker Container内部的指定位置。这样的网络信息，主要有以下两种：

- host宿主机的主机名（hostname）；
- host宿主机上/etc/hosts文件，用于配置IP地址以及主机名。

其中，宿主机的主机名hostname用于创建container.Config中的Hostname与Domainname属性。

另外，Docker Daemon在Docker Container的rootfs内部创建hostname文件，并在文件中写入Hostname与Domainname；同时创建hosts文件，并写入host宿主机上/etc/hosts内的所有内容。

2.2.1.2 初始化other container网络模式配置

Docker Container的other container网络模式意味着：容器使用其他已经创建容器的网络环境。

Docker Daemon首先判断host网络模式之后，若不为host网络模式，则继续判断Docker Container网络模式是否为other container。如果Docker Container的网络模式为other container（假设使用的-net参数为--net=container:17adef，其中17adef为容器ID）。Docker Daemon所做的执行操作包括两部分。

第一步，从container对象的hostConfig属性中找出NetworkMode，并找到相应的容器，即17adef的容器对象container，实现代码如下：

```
nc, err := container.getNetworkedContainer()
```

第二步，将17adef容器对象的HostsPath、ResolveConfPath、Hostname和Domainname赋值给当前容器对象container，实现代码如下：

```
container.HostsPath = nc.HostsPath
container.ResolvConfPath = nc.ResolvConfPath
container.Config.Hostname = nc.Config.Hostname
container.Config.Domainname = nc.Config.Domainname
```

2.2.1.3 初始化none网络模式配置

Docker Container的none网络模式意味着不给该容器创建任何网络环境，容器只能使用127.0.0.1的本机网络。

Docker Daemon通过config属性的DisableNetwork来判断是否为none网络模式。实现代码如下：

```
if container.daemon.config.DisableNetwork {
        container.Config.NetworkDisabled = true
        return container.buildHostnameAndHostsFiles("127.0.1.1")
    }
```

2.2.1.4 初始化bridge网络模式配置

Docker Container的bridge网络模式意味着为容器创建桥接网络模式。桥接模式使得Docker Container创建独立的网络环境，并通过“桥接”的方式实现Docker Container与外界的网络通信。

初始化bridge网络模式的配置，实现代码如下：

```
if err := container.allocateNetwork(); err != nil {
        return err
}
return container.buildHostnameAndHostsFiles(container.NetworkSettings.IPAddress)
```

以上代码完成的内容主要也是两部分：第一，通过allocateNetwork函数为容器分配网络接口设备需要的

资源信息（包括IP、bridge、Gateway等），并赋值给container对象的NetworkSettings；第二，为容器创建hostname以及创建Hosts等文件。

2.2.2创建容器Command信息

Docker在实现容器时，使用了Command类型。Command在Docker Container概念中是一个非常重要的概念。几乎可以认为Command是Docker Container生命周期的源头。Command的概念会贯穿以后的《Docker源码分析》系列，比如Docker Daemon与dockerinit的关系，dockerinit和entrypoint.sh的关系，entrypoint.sh与CMD的关系，以及namespace在这些内容中扮演的角色。

简单来说，Command类型包含了两部分的内容：第一，运行容器内进程的外部命令exec.Cmd；第二，运行容器时启动进程需要的所有环境基础信息：包括容器进程组的使用资源、网络环境、使用设备、工作路径等。通过这两部分的内容，我们可以清楚，如何启动容器内的进程，同时也清楚为容器创建什么样的环境。

首先，我们先来看Command类型的定义，位于./docker/daemon/execdriver/driver.go#L84，通过分析Command类型以及其他相关的数据结构类型，可以得到以下简要类型关系图：

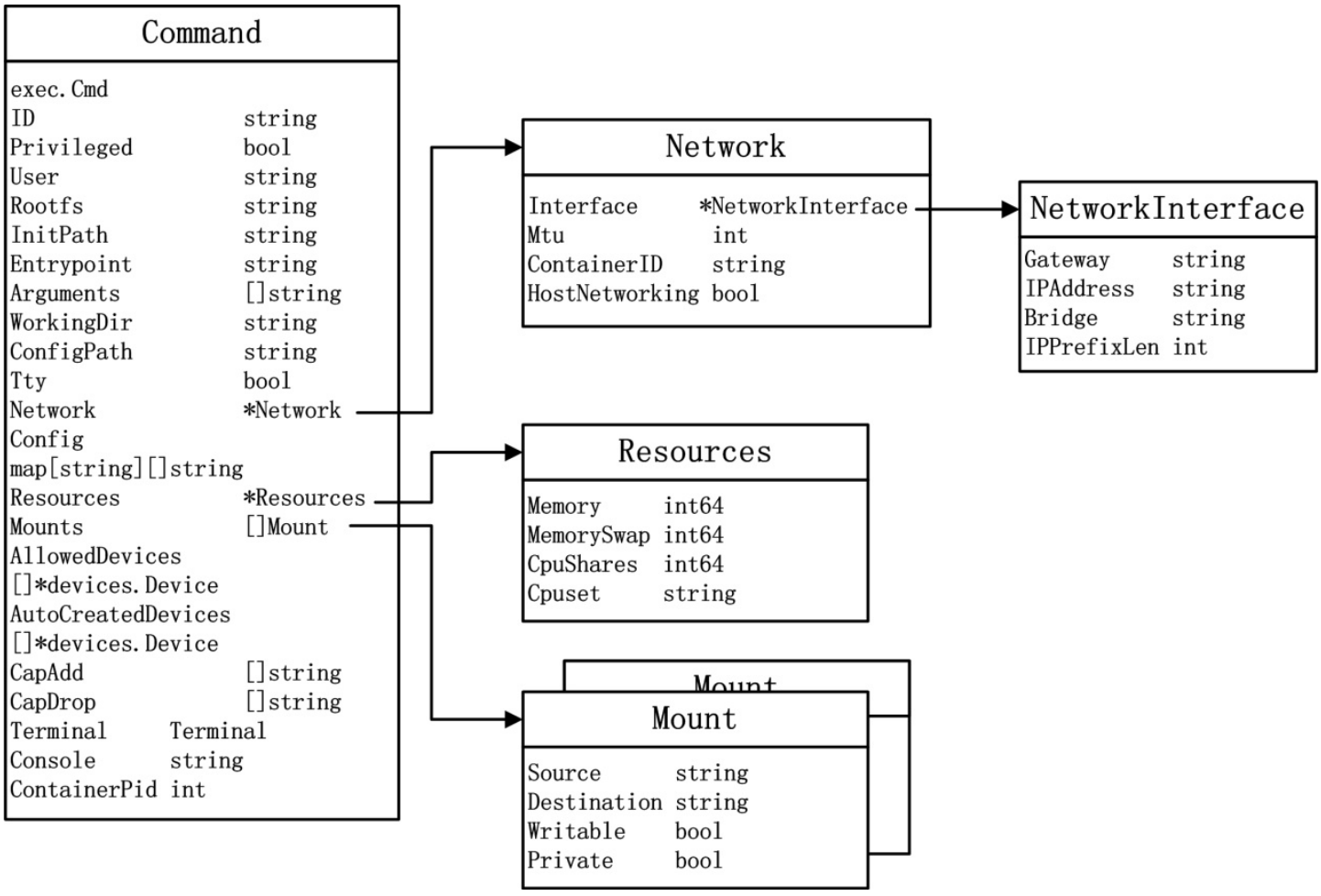


图 2.1 Command类型关系图

从Command类型关系图中可以看到，Command类型中重新包装了exec.Cmd类型，即代表需要创建的进程具体的外部命令；同时，关于网络方面的属性有Network，Network的类型为指向Network类型的指针；关于Docker Container资源使用方面的属性为Resources，从Resource的类型来看，Docker目前能

做的资源限制有4个维度，分别为内存，内存+Swap，CPU使用，CPU核使用；关于挂载的内容，有属性Mounts；等等。

简单介绍Command类型之后，回到Docker Daemon启动容器网络的源码分析。在Start函数的执行流程中，紧接initializeNetworking()之后，与Docker Container网络相关的是populateCommand环节。populateCommand的函数实现位于./docker/daemon/container.go#L191-L274。上文已经提及，populateCommand的作用是创建包execdriver的对象Command，该Command中既有启动容器进程的外部命令，同时也有众多为容器环境的配置信息，包括网络。

本小节，更多的分析populateCommand如何填充Command对象中的网络信息，其他信息的分析会在《源码分析系列》的后续进行展开。

Docker总共有四种网络模式，故populateCommand自然需要判断容器属于哪种网络模式，随后将具体的网络模式信息，写入Command对象的Network属性中。查验Docker Container网络模式的代码位于./docker/daemon/container.go#L204-L227，如下：

```
parts := strings.SplitN(string(c.hostConfig.NetworkMode), ":", 2)
    switch parts[0] {
    case "none":
    case "host":
        en.HostNetworking = true
    case "bridge", "": // empty string to support existing containers
        if !c.Config.NetworkDisabled {
            network := c.NetworkSettings
            en.Interface = &execdriver.NetworkInterface{
                Gateway:     network.Gateway,
                Bridge:      network.Bridge,
                IPAddress:   network.IPAddress,
                IPPrefixLen: network.IPPrefixLen,
            }
        }
    case "container":
        nc, err := c.getNetworkedContainer()
        if err != nil {
            return err
        }
        en.ContainerID = nc.ID
    default:
        return fmt.Errorf("invalid network mode: %s", c.hostConfig.NetworkMode)
    }
```

populateCommand首先通过hostConfig对象中的NetworkMode判断容器属于哪种网络模式。该部分内容涉及到execdriver包中的Network类型，可参见Command类型关系图中的Network类型。若为none模式，则对于Network对象（即en，*execdriver.Network）不做任何操作。若为host模式，则将Network对象的HostNetworking置为true；若为bridge桥接模式，则首先创建一个NetworkInterface对象，完善该对象的Gateway、Bridge、IPAddress和IPPrefixLen信息，最后将NetworkInterface对象作为Network对象的Interface属性的值；若为other container模式，则首先通过getNetworkedContainer()函数获知

被分享网络命名空间的容器，最后将容器ID，赋值给Network对象的ContainerID。由于bridge模式、host模式、none模式以及other container模式彼此互斥，故Network对象中Interface属性、ContainerID属性以及HostNetworking三者之中只有一个被赋值。当Docker Container的网络查验之后，populateCommand将en实例Network属性的值，传递给Command对象。

至此，populateCommand关于网络方面的信息已经完成配置，网络配置信息已经成功赋值于Command的Network属性。。

2.2.3启动容器内部进程

当为容器做好所有的准备与配置之后，Docker Daemon需要真正意义上的启动容器。根据Docker Daemon启动容器流程涉及的Docker模块中可以看到，这样的请求，会被发送至execdriver，再经过libcontainer，最后实现真正启动进程，创建完容器。

回到Docker Daemon的启动容器，daemon包中start函数的最后一步即为执行container.waitForStart()。waitForStart函数的定义位于./docker/daemon/container.go#L1070-L1082，代码如下：

```
func (container *Container) waitForStart() error {
	container.monitor = newContainerMonitor(container, container.hostConfig.RestartPolicy)

	select {
	case <-container.monitor.startSignal:
	case err := <-utils.Go(container.monitor.Start):
	    return err
	}

	return nil
}
```

以上代码运行过程中首先通过newContainerMonitor返回一个初始化的containerMonitor对象，该对象中带有容器进程的重启策略（RestartPolicy）。这里简单介绍containerMonitor对象。总体而言，containerMonitor对象用以监视容器中第一个进程的执行。如果containerMonitor中指定了一种进程重启策略，那么一旦容器内部进程没有启动成功，Docker Daemon会使用重启策略来重启容器。如果在重启策略下，容器依然没有成功启动，那么containerMonitor对象会负责重置以及清除所有已经为容器准备好的资源，例如已经为容器分配好的网络资源（即IP地址），还有为容器准备的rootfs等。

waitForStart()函数通过container.monitor.Start来实现容器的启动，进入./docker/daemon/monitor.go#L100，可以发现启动容器进程位于./docker/daemon/monitor.go#L136，代码如下：

```
exitStatus, err = m.container.daemon.Run(m.container, pipes, m.callback)
```

以上代码实际调用了daemon包中的Run函数，位于./docker/daemon/daemon.go#L969-L971，代码如下：

```
func (daemon *Daemon) Run(c *Container, pipes *execdriver.Pipes, startCallback execdriver.StartCallb
ack) (int, error) {
	return daemon.execDriver.Run(c.command, pipes, startCallback)
}
```

最终，Run函数中调用了execdriver中的Run函数来执行Docker Container的启动命令。

至此，网络部分在Docker Daemon内部的执行已经结束，紧接着程序运行逻辑陷入execdriver，进一步完成容器启动的相关步骤。

3.execdriver网络执行流程

Docker架构中execdriver的作用是启动容器内部进程，最终启动容器。目前，在Docker中execdriver作为执行驱动，可以有两种选项：lxc与native。其中，lxc驱动会调用lxc工具实现容器的启动，而native驱动会使用Docker官方发布的libcontainer来启动容器。

Docker Daemon启动过程中，execdriver的类型默认为native，故本文主要分析native驱动在执行启动容器时，如何处理网络部分。

在Docker Daemon启动容器的最后一步，即调用了execdriver的Run函数来执行。通过分析Run函数的具体实现，关于Docker Container的网络执行流程主要包括两个环节：

(1) 创建libcontainer的Config对象

(2) 通过libcontainer中的namespaces包执行启动容器

将execdriver.Run函数的运行流程具体展开，与Docker Container网络相关的流程，可以得到以下示意图：

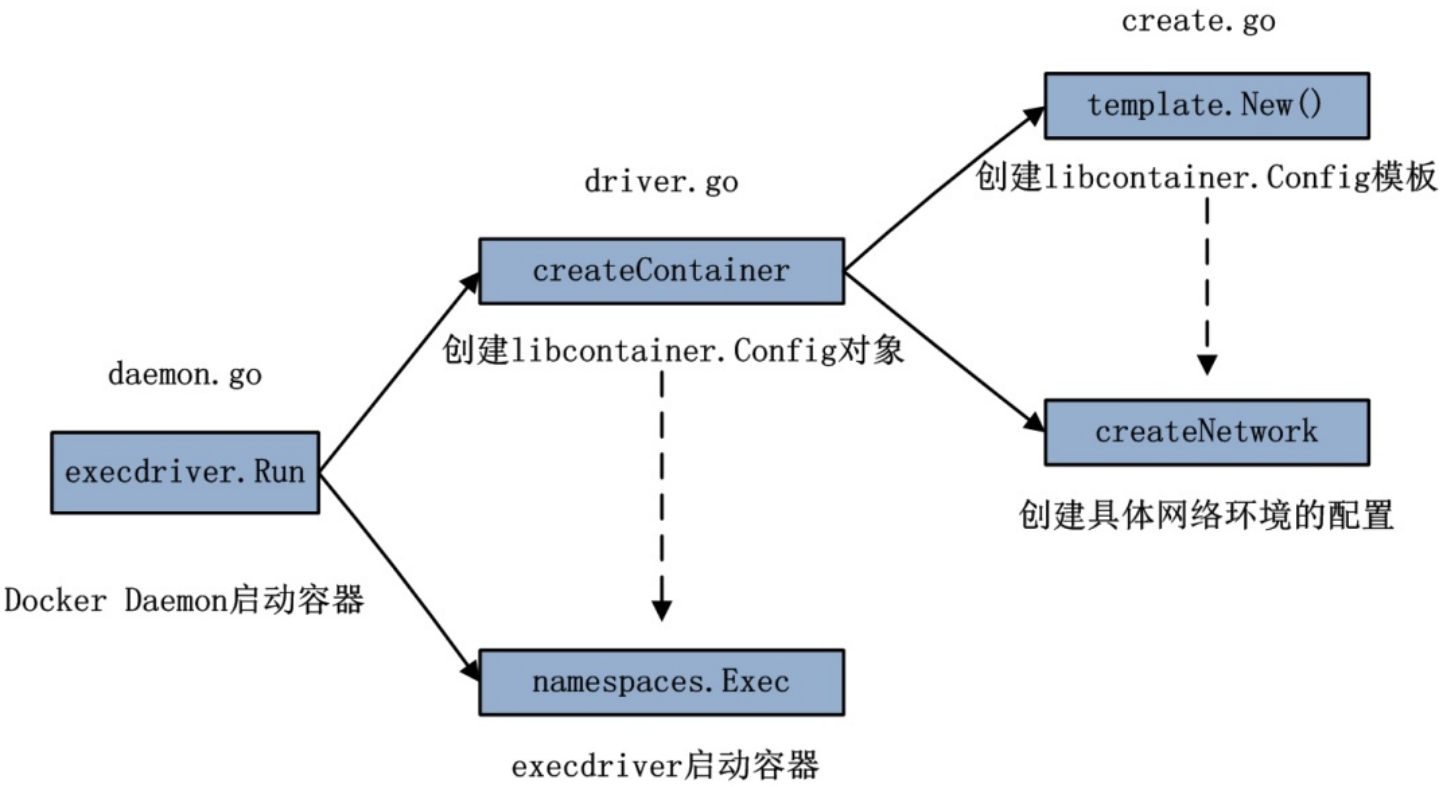


图3.1 execdriver.Run执行流程图

3.1创建libcontainer的Config对象

Run函数位于./docker/daemon/execdriver/native/driver.go#L62-L168，进入Run函数的实现，立即可以发现该函数通过createContainer创建了一个container对象，代码如下：

```
container, err := d.createContainer(c)
```

其中c为Docker Daemon创建的execdriver.Command类型实例。以上代码的createContainer函数的作用是：使用execdriver.Command来填充libcontainer.Config。

libcontainer.Config的作用是，定义在一个容器化的环境中执行一个进程所需要的所有配置项。createContainer函数的存在，使用Docker Daemon层创建的execdriver.Command，创建更下层libcontainer所需要的Config对象。这个角度来看，execdriver更像是封装了libcontainer对外的接口，实现了将Docker Daemon认识的容器启动信息转换为底层libcontainer能真正使用的容器启动配置选项。libcontainer.Config类型与其内部对象的关联图如下：

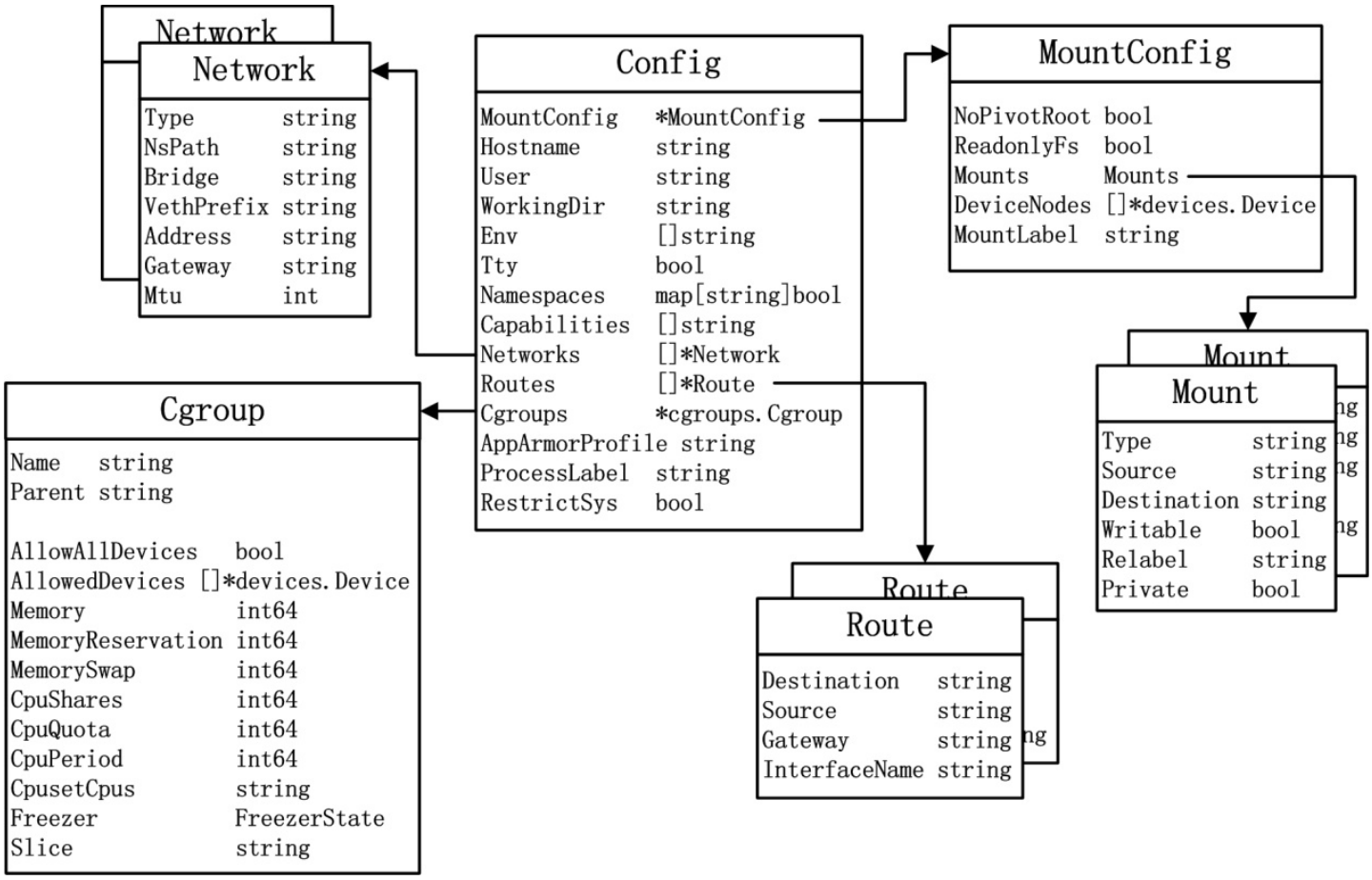


图3.2 libcontainer.Config类型关系图

进入createContainer的源码实现部分，位于./docker/daemon/execdriver/native/create.go#L23-L77，代码如下：

```
func (d *driver) createContainer(c *execdriver.Command) (*libcontainer.Config, error) {
    container := template.New()
    ......
    if err := d.createNetwork(container, c); err != nil {
        return nil, err
    }
    ......
    return container, nil
}
```

3.1.1 libcontainer.Config模板实例

从createContainer函数的实现以及execdriver.Run执行流程图中都可以看到，createContainer所做的第一个操作就是即为执行template.New()，意为创建一个libcontainer.Config的实例container。其中，template.New()的定义位于./docker/daemon/execdriver/native/template/default_template.go，主要的作用为返回libcontainer关于Docker Container的默认配置选项。

Template.New()的代码实现如下：

```
func New() *libcontainer.Config {
    container := &libcontainer.Config{
        Capabilities: []string{
            "CHOWN",
            "DAC_OVERRIDE",
            "FSETID",
            "FOWNER",
            "MKNOD",
            "NET_RAW",
            "SETGID",
            "SETUID",
            "SETFCAP",
            "SETPCAP",
            "NET_BIND_SERVICE",
            "SYS_CHROOT",
            "KILL",
            "AUDIT_WRITE",
        },
        Namespaces: map[string]bool{
            "NEWNS":  true,
            "NEWUTS": true,
            "NEWIPC": true,
            "NEWPID": true,
            "NEWNET": true,
        },
        Cgroups: &cgroups.Cgroup{
            Parent:          "docker",
            AllowAllDevices: false,
        },
        MountConfig: &libcontainer.MountConfig{},
    }

    if apparmor.IsEnabled() {
        container.AppArmorProfile = "docker-default"
    }

    return container
}
```

关于该libcontainer.Config默认的模板对象，从源码实现中可以看到，首先设定了Capabilities的默认项，如CHOWN、DAC_OVERRIDE、FSETID等；其次又将Docker Container所需要的设定的namespaces添加默认值，即需要创建5个NAMESPACE，如NEWNS、NEWUTS、NEWIPC、NEWPID和NEWNET，其中不包括user namespace，另外与网络相关的namespace为NEWNET；最后设定了一些关于cgroup以及apparmor的默认配置。

Template.New()函数最后返回类型为libcontainer.Config的实例container，该实例中只含有默认配置项，其他的配置项的添加需要createContainer的后续操作来完成。

3.1.2 createNetwork实现

在createContainer的实现流程中，为了完成container对象（类型为libcontainer.Config）的完善，最后有很多步骤，如与网络相关的createNetwork函数调用，与Linux内核Capabilities相关的setCapabilities函数调用，与cgroups相关的setupCgroups函数调用，以及与挂载目录相关的setupMounts函数调用等。本小节主要分析createNetwork如何为container对象完善网络配置项。

createNetwork函数的定义位于./docker/daemon/execdriver/native/create.go#L79-L124，该函数主要利用execdriver.Command中Network属性中的内容，来判断如何创建libcontainer.Config中Network属性（关于两中Network属性，可以参见图3.1和图3.2）。由于Docker Container的4种网络模式彼此互斥，故以上Network类型中Interface、ContainerID与HostNetworking最多只有一项会被赋值。

由于execdriver.Command中Network的类型定义如下：

```
type Network struct {
    Interface      *NetworkInterface
    Mtu            int
    ContainerID    string
    HostNetworking bool
}
```

分析createNetwork函数，其具体实现可以归纳为4部分内容：

(1) 判断网络是否为host模式；

(2) 判断是否为bridge桥接模式；

(3) 判断是否为other container模式；

(4) 为Docker Container添加loopback网络设备。

首先来看execdriver判断是否为host模式的代码：

```
if c.Network.HostNetworking {
    container.Namespaces["NEWNET"] = false
    return nil
}
```

当execdriver.Command类型实例中Network属性的HostNetworking为true，则说明需要为Docker Container创建host网络模式，使得容器与宿主机共享同样的网络命名空间。关于host模式的具体介绍中，已经阐明，只须在创建进程进行CLONE系统调用时，不传入CLONE_NEWNET参数标志即可实现。这里的代码正好准确的验证了这一点，将container对象中NEWNET的Namespace设为false，最终在libcontainer中可以达到效果。

再看execdriver判断是否为bridge桥接模式的代码：

```
if c.Network.Interface != nil {
        vethNetwork := libcontainer.Network{
            Mtu:         c.Network.Mtu,
            Address:    fmt.Sprintf("%s/%d", c.Network.Interface.IPAddress,
c.Network.Interface.IPPrefixLen),
            Gateway:    c.Network.Interface.Gateway,
            Type:       "veth",
            Bridge:     c.Network.Interface.Bridge,
            VethPrefix: "veth",
        }
        container.Networks = append(container.Networks, &vethNetwork)
    }
```

当execdriver.Command类型实例中Network属性的Interface不为nil值，则说明需要为Docker Container创建bridge桥接模式，使得容器使用隔离的网络环境。于是这里为类型为libcontainer.Config的container对象添加Networks属性vethNetwork，网络类型为“veth”，以便libcontainer在执行时，可以为Docker Container创建veth pair。

接着来看execdriver判断是否为other container模式的代码：

```
if c.Network.ContainerID != "" {
        d.Lock()
        active := d.activeContainers[c.Network.ContainerID]
        d.Unlock()

        if active == nil || active.cmd.Process == nil {
            return fmt.Errorf("%s is not a valid running container to join", c.Network.ContainerID)
        }
        cmd := active.cmd

        nspath := filepath.Join("/proc", fmt.Sprint(cmd.Process.Pid), "ns", "net")
        container.Networks = append(container.Networks, &libcontainer.Network{
            Type:   "netns",
            NsPath: nspath,
        })
    }
```

当execdriver.Command类型实例中Network属性的ContainerID不为空字符串时，则说明需要为Docker Container创建other container模式，使得创建容器使用其他容器的网络环境。实现过程中，execdriver需要首先在activeContainers中查找需要被共享网络环境的容器active；并通过active容器的启动执行命令cmd找到容器第一进程在宿主机上的PID；随后在proc文件系统中找到该进程PID的关于网络namespace的路径nspath；最后为类型为libcontainer.Config的container对象添加Networks属性，Network的类型为“netns”。

此外，createNetwork函数还实现为Docker Container创建一个loopback回环设备，以便容器可以实现内部通信。实现过程中，同样为类型libcontainer.Config的container对象添加Networks属性，Network的类型为“loopback”，代码如下：

```
container.Networks = []*libcontainer.Network{
        {
            Mtu:     c.Network.Mtu,
            Address: fmt.Sprintf("%s/%d", "127.0.0.1", 0),
            Gateway: "localhost",
            Type:    "loopback",
        },
    }
```

至此，createNetwork函数已经把与网络相关的配置，全部创建在类型为libcontainer.Config的container对象中了，就等着启动容器进程时使用。

3.2 调用libcontainer的namespaces启动容器

回到execdriver.Run函数，创建完libcontainer.Config实例container，经过一系列其他方面的处理之后，最终execdriver执行namespaces.Exec函数实现启动容器，container对象依然是namespace.Exec函数中一个非常重要的参数。这一环节代表着execdriver把启动Docker Container的工作交给libcontainer，以后的执行陷入libcontainer。

调用namespaces.Exec的代码位于./docker/daemon/execdriver/native/driver.go#L102-L127，为了便于理解，简化的代码如下：

```
namespaces.Exec(container, c.Stdin, c.Stdout, c.Stderr, c.Console,
 c.Rootfs, dataPath, args, parameter_1, parameter_2)
```

其中parameter_1为定义的函数，如下：

```
func(container *libcontainer.Config, console, rootfs, dataPath,
init string, child *os.File, args []string) *exec.Cmd {
        c.Path = d.initPath
        c.Args = append([]string{
            DriverName,
            "-console", console,
            "-pipe", "3",
            "-root", filepath.Join(d.root, c.ID),
            "--",
        }, args...)

        // set this to nil so that when we set the clone flags anything else is reset
        c.SysProcAttr = &syscall.SysProcAttr{
            Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
        }
        c.ExtraFiles = []*os.File{child}

        c.Env = container.Env
        c.Dir = c.Rootfs

        return &c.Cmd
    }
```

同样的，parameter_2也为定义的函数，如下：

```
func() {
        if startCallback != nil {
            c.ContainerPid = c.Process.Pid
            startCallback(c)
        }
    }
```

Parameter_1以及parameter_2这两个函数均会在libcontainer的namespaces中发挥很大的重要。

至此，execdriver模块的执行部分已经完结，Docker Daemon的程序运行逻辑陷入libcontainer。

4.libcontainer实现内核态网络配置

libcontainer是一个Linux操作系统上容器的实现包。libcontainer指定了创建一个容器时所需要的配置选项，同时它利用Linux namespace和cgroup等技术为使用者提供了一套Golang原生态的容器实现方案，并且没有使用任何外部依赖。用户借助libcontainer，可以感受到众多操纵namespaces，网络等资源的便利。

当execdriver调用libcontainer中namespaces包的Exec函数时，libcontainer开始发挥其实现容器功能的作用。Exec函数位于./libcontainer/namespaces/exec.go#L24-L113。本文更多的关心Docker Container的网络创建，因此从这个角度来看Exec的实现可以分为三个步骤：

(1) 通过createCommand创建一个Golang语言内的exec.Cmd对象；

(2) 启动命令exec.Cmd，执行容器内第一个进程；

(3) 通过InitializeNetworking函数为容器进程初始化网络环境。

以下详细分析这三个部分，源码的具体实现。

4.1创建exec.Cmd

提到exec.Cmd，就不得不提Go语言标准库中的包os以及包os/exec。前者提供了与平台无关的操作系统功能集，后者则提供了功能集里与命令执行相关的部分。

首先来看一下在Go语言中exec.Cmd的定义，如下：

```
type Cmd struct {
        Path string                 //所需执行命令在系统中的路径
        Args []string                //传入命令的参数
        Env []string                 //进程运行时的环境变量
          Dir string                  //命令运行的工作目录
          Stdin io.Reader
        Stdout io.Writer
        Stderr io.Writer
        ExtraFiles []*os.File            //进程所需打开的文件描述符资源
        SysProcAttr *syscall.SysProcAttr   //可选的操作系统属性
        Process *os.Process           //代表Cmd启动后，操作系统底层的具体进程
        ProcessState *os.ProcessState    //进程退出后保留的信息
}
```

清楚Cmd的定义之后，再来分析namespaces包的Exec函数中，是如何来创建exec.Cmd的。在Exec函数的实现过程中，使用了以下代码实现Exec.Cmd的创建：

```
command := createCommand(container, console, rootfs, dataPath,
os.Args[0], syncPipe.Child(), args)
```

其中createCommand为namespace.Exec函数中传入的倒数第二个参数，类型为CreateCommand。而createCommand只是namespaces.Exec函数的形参，真正的实参则为execdriver调用namespaces.Exec时的参数parameter_1，即如下代码：

```
func(container *libcontainer.Config, console, rootfs, dataPath,
 init string, child *os.File, args []string) *exec.Cmd {
        c.Path = d.initPath
        c.Args = append([]string{
            DriverName,
            "-console", console,
            "-pipe", "3",
            "-root", filepath.Join(d.root, c.ID),
            "--",
        }, args...)

        // set this to nil so that when we set the clone flags anything else is reset
        c.SysProcAttr = &syscall.SysProcAttr{
            Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
        }
        c.ExtraFiles = []*os.File{child}

        c.Env = container.Env
        c.Dir = c.Rootfs

        return &c.Cmd
    }
```

熟悉exec.Cmd的定义之后，分析以上代码的实现就显得较为简单。为Cmd赋值的对象有Path，Args，SysProcAttr，ExtraFiles，Env和Dir。其中需要特别注意的是Path的值d.initPath，该路径下存放的是dockerinit的二进制文件，Docker 1.2.0版本下，路径一般为“/var/lib/docker/init/dockerinit-1.2.0”。另外SysProcAttr使用以下的代码来赋值：

```
&syscall.SysProcAttr{
        Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
    }
```

syscall.SysProAttr对象中的Cloneflags属性中，即保留了libcontainer.Config类型的实例container中的Namespace属性。换言之，通过exec.Cmd创建进程时，正是通过Cloneflags实现Clone系统调用中传入namespace参数标志。

回到函数执行中，在函数的最后返回了c.Cmd，命令创建完毕。

4.2启动exec.Cmd创建进程

创建完exec.Cmd，当然需要将该执行命令运行起来，namespaces.Exec函数中直接使用以下代码实现进程的启动：

```
if err := command.Start(); err != nil {
        return -1, err
    }
```

这一部分的内容简单直接， Start()函数用以完成指定命令exec.Cmd的启动执行，同时并不等待其启动完毕便返回。Start()函数的定义位于os/exec包。

进入os/exec包，查看Start()函数的实现，可以看到执行过程中，会对command.Process进行赋值，此时command.Process中会含有刚才启动进程的PID进程号，该PID号属于在宿主机pid namespace下，而并非是新创建namespace下的PID号。

4.3为容器进程初始化网络环境

上一环节实现了容器进程的启动，然而却还没有为之配置相应的网络环境。namespaces.Exec在之后的InitializeNetworing中实现了为容器进程初始化网络环境。初始化网络环境需要两个非常重要的参数：container对象以及容器进程的Pid号。类型为libcontainer.Config的实例container中包含用户对Docker Container的网络配置需求，另外容器进程的Pid可以使得创建的网络环境与进程新创建的namespace进行关联。

namespaces.Exec中为容器进程初始化网络环境的代码实现位于./libcontainer/namespaces/exec.go#L75-L79，如下：

```
if err := InitializeNetworking(container, command.Process.Pid, syncPipe, &networkState); err != nil {
        command.Process.Kill()
        command.Wait()
        return -1, err
    }
```

InitializeNetworing的作用很明显，即为创建的容器进程初始化网络环境。更为底层的实现包含两个步骤：

(1) 先在容器进程的namespace外部，创建容器所需的网络栈；

(2) 将创建的网络栈迁移进入容器的net namespace。

IntializeNetworking的源代码实现位于./libcontainer/namespaces/exec.go#L176-L187，如下：

```
func InitializeNetworking(container *libcontainer.Config, nspid int, pipe *syncpipe.SyncPipe, networkS
tate *network.NetworkState) error {
    for _, config := range container.Networks {
        strategy, err := network.GetStrategy(config.Type)
        if err != nil {
            return err
        }
        if err := strategy.Create((*network.Network)(config), nspid, networkState); err != nil {
            return err
        }
    }
    return pipe.SendToChild(networkState)
}
```

以上源码实现过程中，首先通过一个循环，遍历libcontainer.Config类型实例container中的网络属性Networks；随后使用GetStrategy函数处理Networks中每一个对象的Type属性，得出Network的类型，这里的类型有3种，分别为“loopback”、“veth”、“netns”。除host网络模式之外，loopback对于其他每一种网络模式的Docker Container都需要使用；veth针对bridge桥接模式，而netns针对other container模式。

得到Network类型的类型之后，libcontainer创建相应的网络栈，具体实现使用每种网络栈类型下的Create函数。以下分析三种不同网络栈各自的创建流程。

4.3.1 loopback网络栈的创建

Loopback是一种本地环回设备，libcontainer创建loopback网络设备的实现代码位于./libcontainer/network/loopback.go#L13-L15，如下：

```
func (l *Loopback) Create(n *Network, nspid int, networkState *NetworkState) error {
        return nil
}
```

令人费解的是，libcontainer在loopback设备的创建函数Create中，并没有作实质性的内容，而是直接返回nil。

其实关于loopback设备的创建，要回到Linux内核为进程新建namespace的阶段。当libcontainer执行command.Start()时，由于创建了一个新的网络namespace，故Linux内核会自动为新的net namespace创建loopback设备。当Linux内核创建完loopback设备之后，libcontainer所做的工作即只要保留loopback设备的默认配置，并在Initialize函数中实现启动该设备。

4.3.2 veth网络栈的创建

Veth是Docker Container实际使用的网络策略之一，其使用网桥docker0并创建veth pair虚拟网络设备对，最终使一个veth配置在宿主机上，而另一个veth安置在容器网络namespace内部。

libcontainer中实现veth策略的代码非常通俗易懂，代码位于./libcontainer/network/veth.go#L19-L50，如下：

```
name1, name2, err := createVethPair(prefix)
    if err != nil {
        return err
    }
    if err := SetInterfaceMaster(name1, bridge); err != nil {
        return err
    }
    if err := SetMtu(name1, n.Mtu); err != nil {
        return err
    }
    if err := InterfaceUp(name1); err != nil {
        return err
    }
    if err := SetInterfaceInNamespacePid(name2, nspid); err != nil {
        return err
    }
```

主要的流程包含的四个步骤：

(1) 在宿主机上创建veth pair；

(2) 将一个veth附加至docker0网桥上；

(3) 启动第一个veth；

(4) 将第二个veth附加至libcontainer创建进程的namespace下。

使用Create函数实现veth pair的创建之后，在Initialize函数中实现将网络namespace中的veth改名为“eth0”，并设置网络设备的MTU等。

4.3.3 netns网络栈的创建

netns专门为Docker Container的other container网络模式服务。netns完成的工作是将其他容器的namespace路径传递给需要创建other container网络模式的容器使用。

libcontainer中实现netns策略的代码位于./libcontainer/network/netns.go#L17-L20，如下：

```
func (v *NetNS) Create(n *Network, nspid int, networkState *NetworkState) error {
        networkState.NsPath = n.NsPath
        return nil
}
```

使用Create函数先将NsPath赋给新建容器之后，在Initialize函数中实现将网络namespace的文件描述符交由新创建容器使用，最终实现两个Docker Container共享同一个网络栈。

通过Create以及Initialize的实现之后，Docker Container相应的网络栈环境即已经完成创建，容器内部的应用进程可以使用不同的网络栈环境与外界或者内部进行通信。关于Initialize函数何时被调用，需要清楚Docker Daemon与dockerinit的关系，以及如何实现Docker Daemon进程与dockerinit进程跨

namespace进行通信，这部分内容会在《Docker源码分析》系列后续专文分析。

5.总结

如何使用Docker Container的网络，一直是工业界倍加关心的问题。本文将从Linux内核原理的角度阐述了什么是Docker Container，并对Docker Container 4种不同的网络模式进行了初步的介绍，最终贯穿Docker 架构中的多个模块，如Docker Client、Docker Daemon、execdriver以及libcontainer，深入分析Docker Container网络的实现步骤。

目前，若只谈论Docker，那么它还是只停留在单host宿主机的场景上。如何面对跨host的场景、如何实现分布式Docker Container的管理，目前为止还没有一个一劳永逸的解决方案。再者，一个解决方案的存在，总是会适应于一个应用场景。Docker这种容器技术的发展，大大改善了传统模式下使用诸如虚拟机等传统计算单位存在的多数弊端，却在网络方面使得自身的使用过程中存在瑕疵。希望本文是一个引子，介绍Docker Container网络，以及从源码的角度分析Docker Container网络之后，能有更多的爱好者思考Docker Container网络的前世今生，并为Docker乃至容器技术的发展做出贡献。

6.作者介绍

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

7.参考文献

http://docs.studygolang.com/pkg/os/exec/#Cmd

https://github.com/docker/libcontainer/tree/v1.2.0

https://github.com/docker/libcontainer/issues/323

（九）：Docker镜像

- 1.前言
- 2.Docker镜像介绍
  - 2.1 rootfs
  - 2.2 Union mount
  - 2.3 image
  - 2.4 layer
- 3.总结
- 4.作者介绍
- 5.参考文献
- 6.下期预告

1.前言

回首过去的2014年，大家可以看到Docker在全球刮起了一阵又一阵的“容器风”，工业界对Docker的探索与实践更是一波高过一波。在如今的2015年以及未来，Docker似乎并不会像其他昙花一现的技术一样，在历史的舞台上热潮褪去，反而在工业界实践与评估之后，显现了前所未有的发展潜力。

究其本质，“Docker提供容器服务”这句话，相信很少有人会有异议。那么，既然Docker提供的服务属于“容器”技术，那么反观“容器”技术的本质与历史，我们又可以发现什么呢？正如前文所提到的，Docker使用的“容器”技术，主要是以Linux的cgroup、namespace等内核特性为基础，保障进程或者进程组处于一个隔离、安全的环境。Docker发行第一个版本是在2013年的3月，而cgroup的正式亮相可以追溯到2007年下半年，当时cgroup被合并至Linux内核2.6.24版本。期间6年时间，并不是“容器”技术发展的真空期，2008年LXC（Linux Container）诞生，其简化了容器的创建与管理；之后业界一些PaaS平台也初步尝试采用容器技术作为其云应用的运行环境；而与Docker发布同年，Google也发布了开源容器管理工具lmctfy。除此之外，若抛开Linux操作系统，其他操作系统如FreeBSD、Solaris等，同样诞生了作用相类似的“容器”技术，其发展历史更是需要追溯至千禧年初期。

可见，“容器”技术的发展不可谓短暂，然而论同时代的影响力，却鲜有Docker的媲美者。不论是云计算大潮催生了Docker技术，抑或是Docker技术赶上了云计算的大时代，毋庸置疑的是，Docker作为领域内的新宠儿，必然会继续受到业界的广泛青睐。云计算时代，分布式应用逐渐流行，并对其自身的构建、交付与运行有着与传统不一样的要求。借助Linux内核的cgroup与namespace特性，自然可以做到应用运行环境的资源隔离与应用部署的快速等；然而，cgroup和namespace等内核特性却无法为容器的运行环境做全盘打包。而Docker的设计则很好得考虑到了这一点，除cgroup和namespace之外，另外采用了神奇的“镜像”技术作为Docker管理文件系统以及运行环境的强有力补充。Docker灵活的“镜像”技术，在笔者看来，也是其大红大紫最重要的因素之一。

2.Docker镜像介绍

大家看到这，第一个问题肯定是“什么是Docker镜像”？

据Docker官网的技术文档描述，Image（镜像）是Docker术语的一种，代表一个只读的layer。而layer则具体代表Docker Container文件系统中可叠加的一部分。

笔者如此介绍Docker镜像，相信众多Docker爱好者理解起来依旧是云里雾里。那么理解之前，先让我们来认识一下与Docker镜像相关的4个概念：rootfs、Union mount、image以及layer。

2.1 rootfs

Rootfs：代表一个Docker Container在启动时（而非运行后）其内部进程可见的文件系统视角，或者是Docker Container的根目录。当然，该目录下含有Docker Container所需要的系统文件、工具、容器文件等。

传统来说，Linux操作系统内核启动时，内核首先会挂载一个只读（read-only）的rootfs，当系统检测其完整性之后，决定是否将其切换为读写（read-write）模式，或者最后在rootfs之上另行挂载一种文件系统并忽略rootfs。Docker架构下，依然沿用Linux中rootfs的思想。当Docker Daemon为Docker Container挂载rootfs的时候，与传统Linux内核类似，将其设定为只读（read-only）模式。在rootfs挂载完毕之后，和Linux内核不一样的是，Docker Daemon没有将Docker Container的文件系统设为读写（read-write）模式，而是利用Union mount的技术，在这个只读的rootfs之上再挂载一个读写（read-write）的文件系统，挂载时该读写（read-write）文件系统内空无一物。

举一个Ubuntu容器启动的例子。假设用户已经通过Docker Registry下拉了Ubuntu:14.04的镜像，并通过命令docker run –it ubuntu:14.04 /bin/bash将其启动运行。则Docker Daemon为其创建的rootfs以及容器可读写的文件系统可参见图2.1：

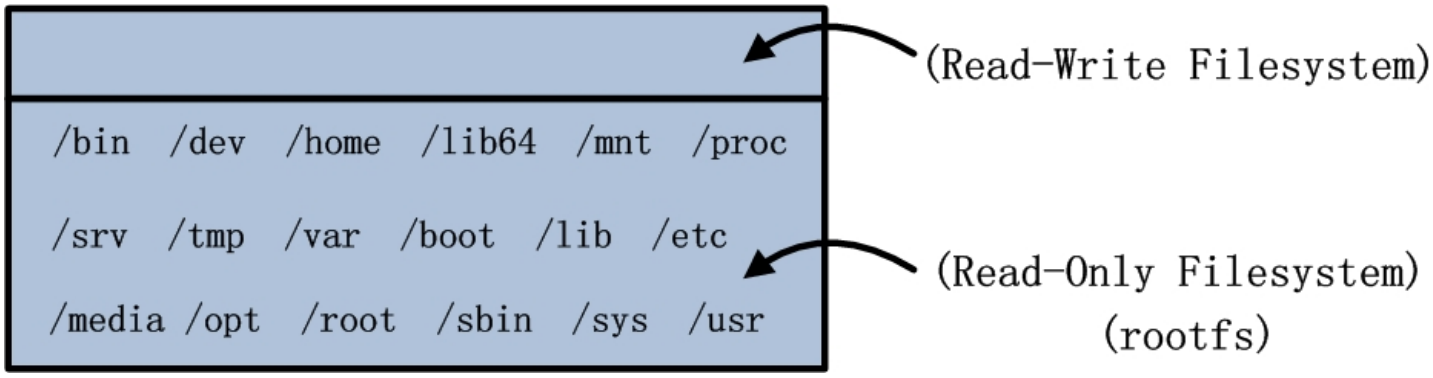


图2.1 Ubuntu 14.04容器rootfs示意图

正如read-only和read-write的含义那样，该容器中的进程对rootfs中的内容只拥有读权限，对于read-write读写文件系统中的内容既拥有读权限也拥有写权限。通过观察图2.1可以发现：容器虽然只有一个文件系统，但该文件系统由“两层”组成，分别为读写文件系统和只读文件系统。这样的理解已然有些层级（layer）的意味。

简单来讲，可以将Docker Container的文件系统分为两部分，而上文提到是Docker Daemon利用Union

Mount的技术，将两者挂载。那么Union mount又是一种怎样的技术？

2.2 Union mount

Union mount：代表一种文件系统挂载的方式，允许同一时刻多种文件系统挂载在一起，并以一种文件系统的形式，呈现多种文件系统内容合并后的目录。

一般情况下，通过某种文件系统挂载内容至挂载点的话，挂载点目录中原先的内容将会被隐藏。而Union mount则不会将挂载点目录中的内容隐藏，反而是将挂载点目录中的内容和被挂载的内容合并，并为合并后的内容提供一个统一独立的文件系统视角。通常来讲，被合并的文件系统中只有一个会以读写（read-write）模式挂载，而其他的文件系统的挂载模式均为只读（read-only）。实现这种Union mount技术的文件系统一般被称为Union Filesystem，较为常见的有UnionFS、AUFS、OverlayFS等。

Docker实现容器文件系统Union mount时，提供多种具体的文件系统解决方案，如Docker早版本沿用至今的的AUFS，还有在docker 1.4.0版本中开始支持的OverlayFS等。

更深入的了解Union mount，可以使用AUFS文件系统来进一步阐述上文中ubuntu:14.04容器文件系统的例子。如图2.2：

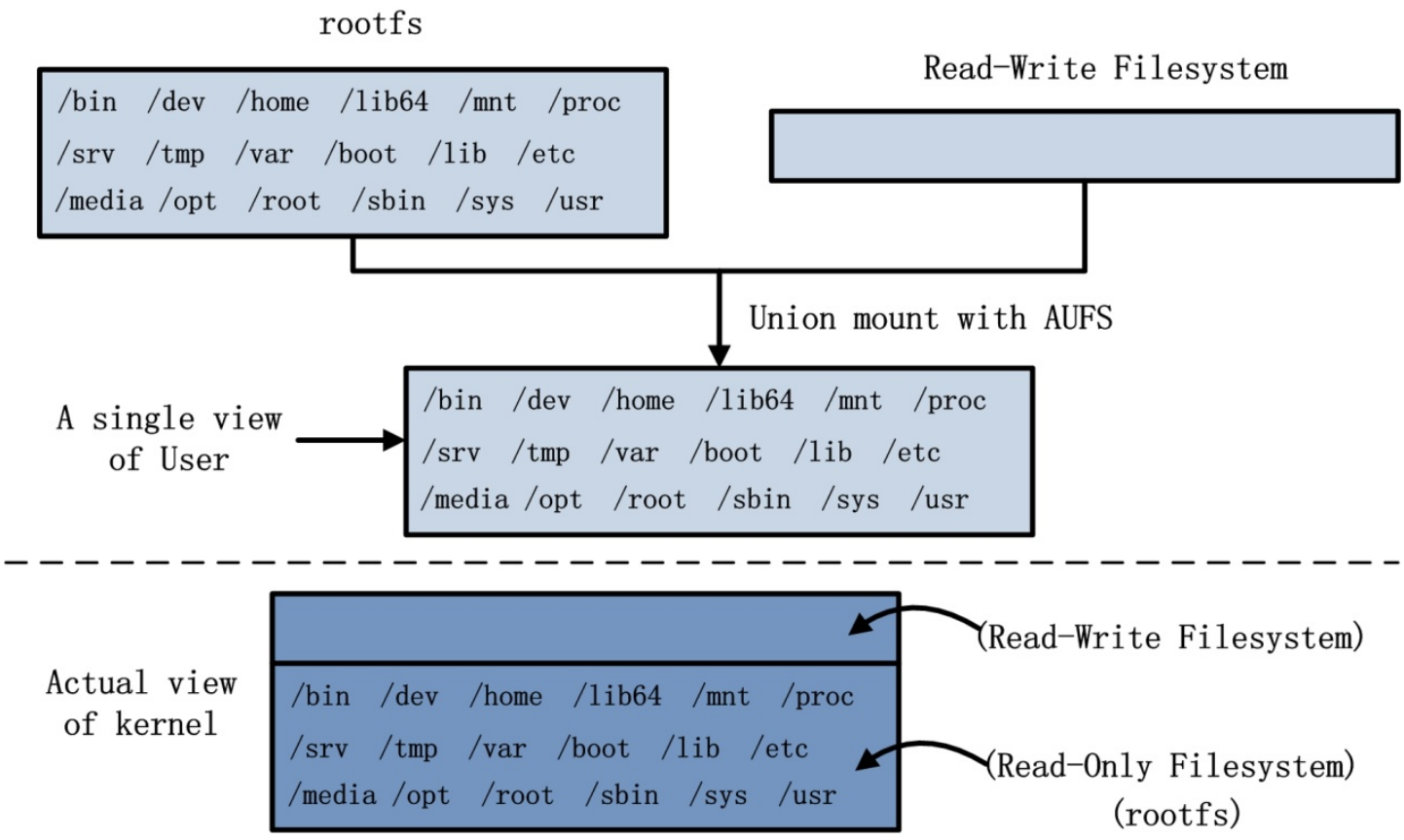


图2.2 AUFS挂载Ubuntu 14.04文件系统示意图

使用镜像ubuntu:14.04创建的容器中，可以暂且将该容器整个rootfs当成是一个文件系统。上文也提到，挂载时读写（read-write）文件系统中空无一物。既然如此，从用户视角来看，容器内文件系统和rootfs完全一样，用户完全可以按照往常习惯，无差别的使用自身视角下文件系统中的所有内容；然而，从内核的角度来看，两者在有着非常大的区别。追溯区别存在的根本原因，那就不得不提及AUFS等文件系统的

COW（copy-on-write）特性。

COW文件系统和其他文件系统最大的区别就是：从不覆写已有文件系统中已有的内容。由于通过COW文件系统将两个文件系统（rootfs和read-write filesystem）合并，最终用户视角为合并后的含有所有内容的文件系统，然而在Linux内核逻辑上依然可以区别两者，那就是用户对原先rootfs中的内容拥有只读权限，而对read-write filesystem中的内容拥有读写权限。

既然对用户而言，全然不知哪些内容只读，哪些内容可读写，这些信息只有内核在接管，那么假设用户需要更新其视角下的文件/etc/hosts，而该文件又恰巧是rootfs只读文件系统中的内容，内核是否会抛出异常或者驳回用户请求呢？答案是否定的。当此情形发生时，COW文件系统首先不会覆写read-only文件系统中的文件，即不会覆写rootfs中/etc/hosts，其次反而会将该文件拷贝至读写文件系统中，即拷贝至读写文件系统中的/etc/hosts，最后再对后者进行更新操作。如此一来，纵使rootfs与read-write filesystem中均由/etc/ hosts，诸如AUFS类型的COW文件系统也能保证用户视角中只能看到read-write filesystem中的/etc/hosts，即更新后的内容。

当然，这样的特性同样支持rootfs中文件的删除等其他操作。例如：用户通过apt-get软件包管理工具安装Golang，所有与Golang相关的内容都会被安装在读写文件系统中，而不会安装在rootfs。此时用户又希望通过apt-get软件包管理工具删除所有关于MySQL的内容，恰巧这部分内容又都存在于rootfs中时，删除操作执行时同样不会删除rootfs实际存在的MySQL，而是在read-write filesystem中删除该部分内容，导致最终rootfs中的MySQL对容器用户不可见，也不可访。

掌握Docker中rootfs以及Union mount的概念之后，再来理解Docker镜像，就会变得水到渠成。

2.3 image

Docker中rootfs的概念，起到容器文件系统中基石的作用。对于容器而言，其只读的特性，也是不难理解。神奇的是，实际情况下Docker的rootfs设计与实现比上文的描述还要精妙不少。

继续以ubuntu 14.04为例，虽然通过AUFS可以实现rootfs与read-write filesystem的合并，但是考虑到rootfs自身接近200MB的磁盘大小，如果以这个rootfs的粒度来实现容器的创建与迁移等，是否会稍显笨重，同时也会大大降低镜像的灵活性。而且，若用户希望拥有一个ubuntu 14.10的rootfs，那么是否有必要创建一个全新的rootfs，毕竟ubuntu 14.10和ubuntu 14.04的rootfs中有很多一致的内容。

Docker中image的概念，非常巧妙的解决了以上的问题。最为简单的解释image，就是 Docker容器中只读文件系统rootfs的一部分。换言之，实际上Docker容器的rootfs可以由多个image来构成。多个image构成rootfs的方式依然沿用Union mount技术。

多个Image构成rootfs的示意图如图2.3（图中，rootfs中每一层image中的内容划分只为了阐述清楚rootfs由多个image构成，并不代表实际情况中rootfs中的内容划分）：

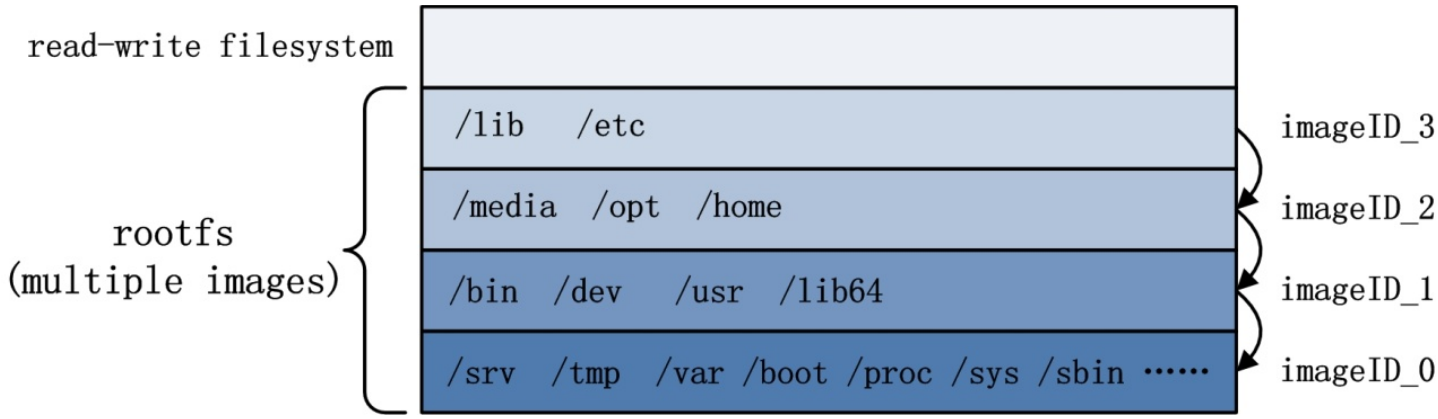


图2.3容器rootfs多image构成图

从上图可以看出，举例的容器rootfs包含4个image，其中每个image中都有一些用户视角文件系统中的一部分内容。4个image处于层叠的关系，除了最底层的image，每一层的image都叠加在另一个image之上。另外，每一个image均含有一个image ID，用以唯一的标记该image。

基于以上的概念，Docker Image中又抽象出两种概念：Parent Image以及Base Image。除了容器rootfs最底层的image，其余image都依赖于其底下的一个或多个image，而Docker中将下一层的image称为上一层image的Parent Image。以图2.3为例，imageID_0是imageID_1的Parent Image，imageID_2是imageID_3的Parent Image，而imageID_0没有Parent Image。对于最下层的image，即没有Parent Image的镜像，在Docker中习惯称之为Base Image。

通过image的形式，原先较为臃肿的rootfs被逐渐打散成轻便的多层。Image除了轻便的特性，同时还有上文提到的只读特性，如此一来，在不同的容器、不同的rootfs中image完全可以用来复用。

多image组织关系与复用关系如图2.4（图中镜像名称的举例只为将image之间的关系阐述清楚，并不代表实际情况中相应名称image之间的关系）：

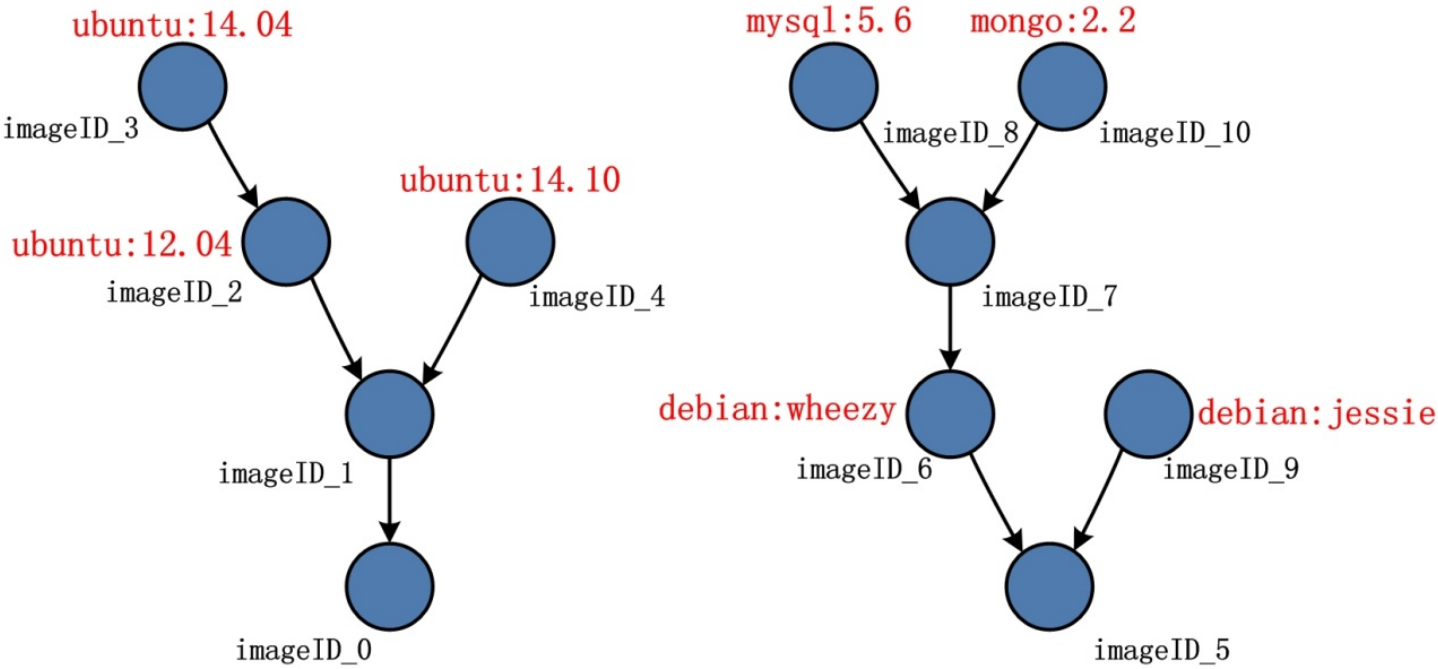


图2.4 多image组织关系示意图

图2.4中，共罗列了11个image，这11个image之间的关系呈现一副森林图。森林中含有两棵树，左边树

中包含5个节点，即含有5个image；右边树中包含6个节点，即含有6个image。图中，有些image标记了红色字段，意味该image代表某一种容器镜像rootfs的最上层image。如图中的ubuntu:14.04，代表imageID_3为该类型容器rootfs的最上层，沿着该节点找到树的根节点，可以发现路径上还有imageID_2，imageID_1和imageID_0。特殊的是，imageID_2作为imageID_3的Parent Image，同时又是容器镜像ubuntu:12.04的rootfs中的最上层，可见镜像ubuntu:14.04只是在镜像ubuntu:12.04之上，再另行叠加了一层。因此，在下载镜像ubuntu:12.04以及ubuntu:14.04时，只会下载一份imageID_2、imageID_1和imageID_0，实现image的复用。同时，右边树中mysql:5.6、mongo:2.2、debian:wheezy和debian:jessie也呈现同样的关系。

2.4 layer

Docker术语中，layer是一个与image含义较为相近的词。容器镜像的rootfs是容器只读的文件系统，rootfs又是由多个只读的image构成。于是，rootfs中每个只读的image都可以称为一层layer。

除了只读的image之外，Docker Daemon在创建容器时会在容器的rootfs之上，再mount一层read-write filesystem，而这一层文件系统，也称为容器的一层layer，常被称为top layer。

因此，总结而言，Docker容器中的每一层只读的image，以及最上层可读写的文件系统，均被称为layer。如此一来，layer的范畴比image多了一层，即多包含了最上层的read-write filesystem。

有了layer的概念，大家可以思考这样一个问题：容器文件系统分为只读的rootfs，以及可读写的top layer，那么容器运行时若在top layer中写入了内容，那这些内容是否可以持久化，并且也被其它容器复用？

上文对于image的分析中，提到了image有复用的特性，既然如此，再提一个更为大胆的假设：容器的top layer是否可以转变为image？

答案是肯定的。Docker的设计理念中，top layer转变为image的行为（Docker中称为commit操作），大大释放了容器rootfs的灵活性。Docker的开发者完全可以基于某个镜像创建容器做开发工作，并且无论在开发周期的哪个时间点，都可以对容器进行commit，将所有top layer中的内容打包为一个image，构成一个新的镜像。Commit完毕之后，用户完全可以基于新的镜像，进行开发、分发、测试、部署等。不仅docker commit的原理如此，基于Dockerfile的docker build，其追核心的思想，也是不断将容器的top layer转化为image。

3.总结

Docker风暴席卷全球，并非偶然。如今的云计算时代下，轻量级容器技术与灵活的镜像技术相结合，似乎颠覆了以往的软件交付模式，为持续集成（Continuous Integration, CI）与持续交付（Continuous Delivery, CD）的发展带来了全新的契机。

理解Docker的“镜像”技术，有助于Docker爱好者更好的使用、创建以及交付Docker镜像。基于此，本文从Docker镜像的4个重要概念入手，介绍了Docker镜像中包含的内容，涉及到的技术，还有重要的特性。Docker引入优秀的“镜像”技术时，着实使容器的使用变得更为便利，也拓宽了Docker的使用范

畴。然而，于此同时，我们也应该理性地看待镜像技术引入时，是否会带来其它的副作用。关于镜像技术的其它思考，《Docker源码分析系列》将在后续另文分析。

4.作者介绍

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

5.参考文献

http://www.csdn.net/article/2014-09-24/2821832

http://en.wikipedia.org/wiki/Cgroups

http://www.infoq.com/cn/articles/docker-future

https://docs.docker.com/terms/image/

https://docs.docker.com/terms/layer/#layer

http://en.wikipedia.org/wiki/Union_mount

https://www.usenix.org/legacy/publications/library/proceedings/neworl/full_papers/mckusick.a

http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Ffsys_COW_filesystem.html

6.下期预告

Docker源码分析（十）：Docker镜像下载

Docker源码分析（十一）：Docker镜像存储

（十）：Docker镜像下载

- 1.前言
- 2.本文分析内容安排
- 3.Docker镜像下载流程
- 4.Docker Client
  - 4.1 解析镜像参数
  - 4.2 配置认证信息
  - 4.3 发送API请求
- 5.Docker Server
  - 5.1 解析请求参数
  - 5.2 创建并配置job
  - 5.3 触发执行job
- 6.Docker Daemon
  - 6.1 解析job参数
  - 6.2 创建session对象
  - 6.3 执行镜像下载
- 7.总结
- 8.作者介绍
- 9.参考文献

1.前言

说Docker Image是Docker体系的价值所在，没有丝毫得夸大其词。Docker Image作为容器运行环境的基石，彻底解放了Docker容器创建的生命力，也激发了用户对于容器运用的无限想象力。

玩转Docker，必然离不开Docker Image的支持。然而“万物皆有源”，Docker Image来自何方，Docker Image又是通过何种途径传输到用户机器，以致用户可以通过Docker Image创建容器？回忆初次接触Docker的场景，大家肯定对两条命令不陌生：docker pull和docker run。这两条命令中，正是前者实现了Docker Image的下载。Docker Daemon在执行这条命令时，会将Docker Image从Docker Registry下载至本地，并保存在本地Docker Daemon管理的graph中。

谈及Docker Registry，Docker爱好者首先联想到的自然是Docker Hub。Docker Hub作为Docker官方支持的Docker Registry，拥有全球成千上万的Docker Image。全球的Docker爱好者除了可以下载Docker Hub开放的镜像资源之外，还可以向Docker Hub贡献镜像资源。在Docker Hub上，用户不仅可以享受公有镜像带来的便利，而且可以创建私有镜像库。Docker Hub是全国最大的Public Registry，另外Docker还支持用户自定义创建Private Registry。Private Registry主要的功能是为私有网络提供Docker

镜像的专属服务，一般而言，镜像种类适应用户需求，私密性较高，且不会占用公有网络带宽。

2.本文分析内容安排

本文作为《Docker源码分析》系列的第十篇——Docker镜像下载篇，主要从源码的角度分析Docker下载Docker Image的过程。分析流程中，docker的版本均为1.2.0。

分析内容的安排如以下4部分：

(1) 概述Docker镜像下载的流程，涉及Docker Client、Docker Server与Docker Daemon；

(2) Docker Client处理并发送docker pull请求；

(3) Docker Server接收docker pull请求，并创建镜像下载任务并触发执行；

(4) Docker Daemon执行镜像下载任务，并存储镜像至graph。

3.Docker镜像下载流程

Docker Image作为Docker生态中的精髓，下载过程中需要Docker架构中多个组件的协作。Docker镜像的下载流程如图3.1：

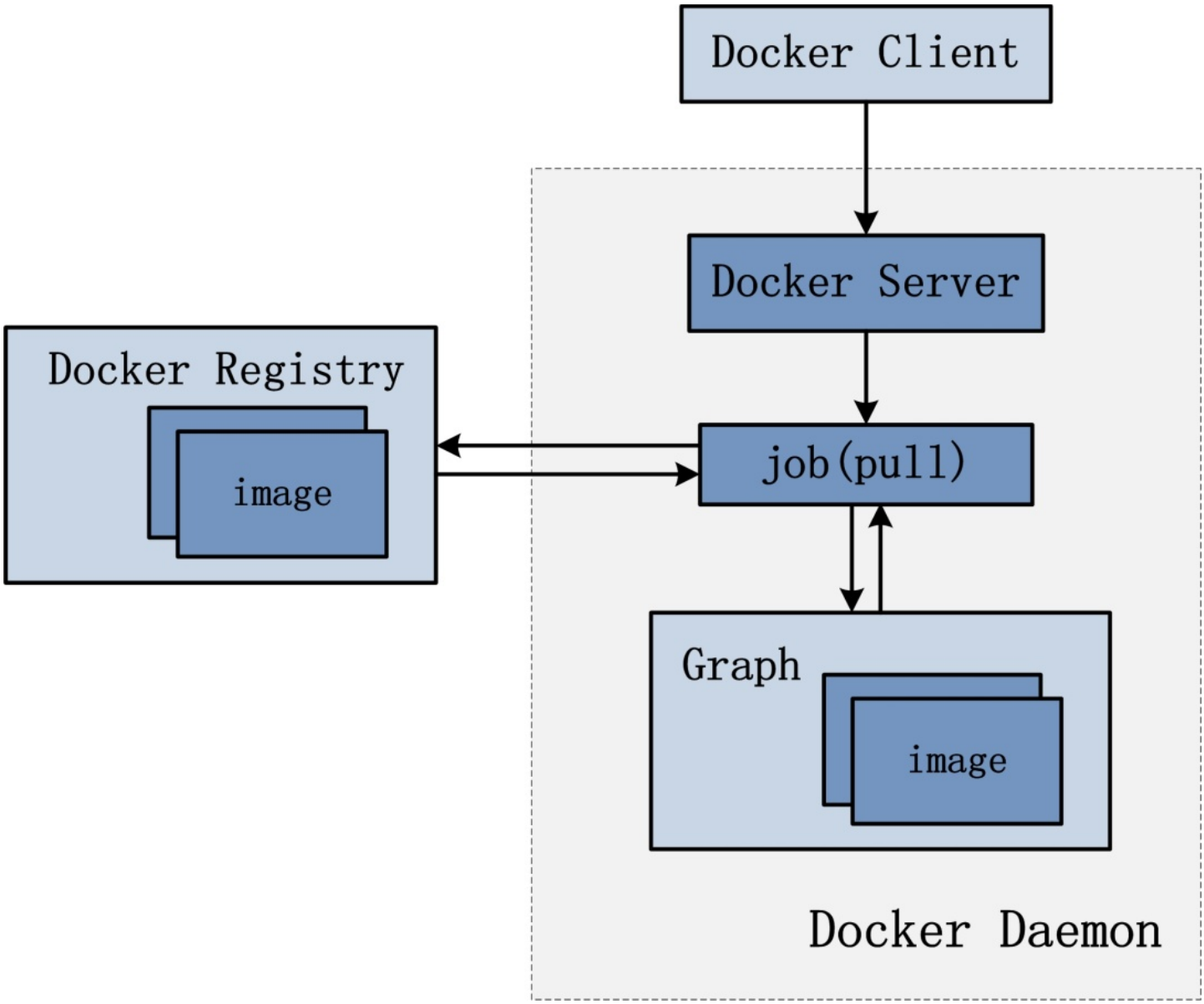


图3.1 Docker镜像下载流程图

如上图，下载流程，可以归纳为以上3个步骤：

(1) 用户通过Docker Client发送pull请求，作用为：让Docker Daemon下载指定名称的镜像；

(2) Docker Daemon中负责Docker API请求的Docker Server，接收Docker镜像的pull请求，创建下载镜像任务并触发执行；

(3) Docker Daemon执行镜像下载任务，从Docker Registry中下载指定镜像，并将其存储与本地的graph中。

下文即从三个方面分析docker pull请求执行的流程。

4.Docker Client

Docker架构中，Docker用户的角色绝大多数由Docker Client来扮演。因此，用户对Docker的管理请求全部由Docker Client来发送，Docker镜像下载请求自然也不例外。

为了更清晰的描述Docker镜像下载，本文结合具体的命令进行分析，如下：

```
docker pull ubuntu:14.04
```

以上的命令代表：用户通过docker二进制可执行文件，执行pull命令，镜像参数为ubuntu:14.04，镜像名称为ubuntu，镜像标签为14.04。此命令一经触发，第一个接受并处理的Docker组件为Docker Client，执行内容包括以下三个步骤：

(1) 解析命令中与Docker镜像相关的参数；

(2) 配置Docker下载镜像时所需的认证信息；

(3) 发送RESTful请求至Docker Daemon。

4.1 解析镜像参数

通过docker二进制文件执行docker pull ubuntu:14.04 时，Docker Client首先会被创建，随后通过参数处理分析出请求类型pull，最终执行pull请求相应的处理函数。关于Docker Client的创建与命令执行可以参见《Docker源码分析》系列第二篇——Docker Client篇。

Docker Client执行pull请求相应的处理函数，源码位于./docker/api/client/command.go#L1183-L1244，有关提取镜像参数的源码如下：

```
func (cli *DockerCli) CmdPull(args ...string) error {
    cmd := cli.Subcmd("pull", "NAME[:TAG]", "Pull an image or a repository from the registry")
    tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image in a repository")
    if err := cmd.Parse(args); err != nil {
        return nil
    }

    if cmd.NArg() != 1 {
        cmd.Usage()
        return nil
    }
    var (
        v      = url.Values{}
        remote = cmd.Arg(0)
    )

    v.Set("fromImage", remote)

    if *tag == "" {
        v.Set("tag", *tag)
    }

    remote, _ = parsers.ParseRepositoryTag(remote)
    // Resolve the Repository name from fqn to hostname + name
    hostname, _, err := registry.ResolveRepositoryName(remote)
    if err != nil {
        return err
    }
    ......
}
```

结合命令docker pull ubuntu:14.04，来分析CmdPull函数的定义，可以发现，该函数传入的形参为args，实参只有一个字符串ubuntu:14.04。另外，纵观以上源码，可以发现Docker Client解析的镜像参数无外乎4个：tag、remote、v和hostname，四者各自的作用如下：

- tag：带有Docker镜像的标签；
- remote：带有Docker镜像的名称与标签；
- v：类型为url.Values，实质是一个map类型，用于配置请求中URL的查询参数；
- hostname：Docker Registry的地址，代表用户希望从指定的Docker Registry下载Docker镜像。

4.1.1 解析tag参数

Docker镜像的tag参数，是第一个被Docker Client解析的镜像参数，代表用户所需下载Docker镜像的标签信息，如：docker pull ubuntu:14.04请求中镜像的tag信息为14.04，若用户使用docker pull ubuntu请求下载镜像，没有显性指定tag信息时，Docker Client会默认该镜像的tag信息为latest。

Docker 1.2.0版本除了以上的tag信息传入方式，依旧保留着代表镜像标签的flag参数tag，而这个flag参数在1.2.0版本的使用过程中已经被遗弃，并会在之后新版本的Docker中被移除，因此在使用docker 1.2.0版本下载Docker镜像时，不建议使用flag参数tag。传入tag信息的方式，建议使用docker pull

NAME[:TAG]的形式。

Docker 1.2.0版本依旧保留的flag参数tag，其定义与解析的源码位于：./docker/api/client/commands.go#1185-L1188，如下：

```
tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image in a repository")
if err := cmd.Parse(args); err != nil {
        return nil
}
```

以上的源码说明：CmdPull函数解析tag参数时，Docker Client首先定义一个flag参数，flag参数的名称为”#t”或者 “#-tag” ，用途为：指定Docker镜像的tag参数，默认值为空字符串；随后通过cmd.Parse(args)的执行，解析args中的tag参数。

4.1.2 解析remote参数

Docker Client解析完tag参数之后，同样需要解析出Docker镜像所属的repository，如请求docker pull ubuntu:14.04中，Docker镜像为ubuntu:14.04，镜像的repository信息为ubuntu，镜像的tag信息为14.04。

Docker Client通过解析remote参数，使得remote参数携带repository信息和tag信息。Docker Client解析remote参数的第一个步骤，源码如下：

```
remote = cmd.Arg(0)
```

其中，cmd的第一个参数赋值给remote，以docker pull ubuntu:14.04为例，cmd.Arg(0)为ubuntu:14.04，则赋值后remote值为ubuntu:14.04。此时remote参数即包含Docker镜像的repository信息也包含tag信息。若用户请求中带有Docker Registry的信息，如docker pull localhost.localdomain:5000/docker/ubuntu:14.04，cmd.Arg(0)为localhost.localdomain:5000/docker/ubuntu:14.04，则赋值后remote值为localhost.localdomain:5000/docker/ubuntu:14.04，此时remote参数同时包含repository信息、tag信息以及Docker Registry信息。

随后，在解析remote参数的第二个步骤中，Docker Client通过解析赋值完毕的remote参数，从中解析中repository信息，并再次覆写remote参数的值，源码如下：

```
remote, _ = parsers.ParseRepositoryTag(remote)
```

ParseRepositoryTag的作用是：解析出remote参数的repository信息和tag信息，该函数的实现位于./docker/pkg/parsers/parsers.go#L72-L81，源码如下:

```
func ParseRepositoryTag(repos string) (string, string) {
        n := strings.LastIndex(repos, ":")
        if n < 0 {
            return repos, ""
        }
        if tag := repos[n+1:]; !strings.Contains(tag, "/") {
            return repos[:n], tag
        }
        return repos, ""
}
```

以上函数的实现过程，充分考虑了多种不同Docker Registry的情况，如：请求docker pull ubuntu:14.04中remote参数为ubuntu:14.04，而请求docker pull localhost.localdomain:5000/docker/ubuntu:14.04中用户指定了Docker Registry的地址localhost.localdomain:5000/docker，故remote参数还携带了Docker Registry信息。

ParseRepositoryTag函数首先从repos参数的尾部往前寻找”:” ，若不存在，则说明用户没有显性指定Docker镜像的tag，返回整个repos作为Docker镜像的repository；若”:”存在，则说明用户显性指定了Docker镜像的tag，”:”前的内容作为repository信息，”:”后的内容作为tag信息，并返回两者。

ParseRepositoryTag函数执行完，回到CmdPull函数，返回内容的repository信息将覆写remote参数。对于请求docker pull localhost.localdomain:5000/docker/ubuntu:14.04，remote参数被覆写后，值为localhost.localdomain:5000/docker/ubuntu，携带Docker Registry信息以及repository信息。

4.1.3 配置url.Values

Docker Client发送请求给Docker Server时，需要为请求配置URL的查询参数。CmdPull函数的执行过程中创建url.Value并配置的源码实现位于./docker/api/client/commands.go#L1194-L1203，如下：

```
var (
        v      = url.Values{}
        remote = cmd.Arg(0)
    )

    v.Set("fromImage", remote)

    if *tag == "" {
        v.Set("tag", *tag)
    }
```

其中，变量v的类型url.Values，配置的URL查询参数有两个，分别为”fromImage”与”tag” ，” fromImage”的值是remote参数没有被覆写时值，” tag”的值一般为空，原因是一般不使用flag参数tag。

4.1.4 解析hostname参数

Docker Client解析镜像参数时，还有一个重要的环节，那就是解析Docker Registry的地址信息。这意味

着用户希望从指定的Docker Registry中下载Docker镜像。

解析Docker Registry地址的代码实现位于./docker/api/client/commands.go#L1207，如下：

```
hostname, _, err := registry.ResolveRepositoryName(remote)
```

Docker Client通过包registry中的函数ResolveRepositoryName来解析hostname参数，传入的实参为remote，即去tag化的remote参数。ResolveRepositoryName函数的实现位于./docker/registry/registry.go#L237-L259，如下：

```
func ResolveRepositoryName(reposName string) (string, string, error) {
        if strings.Contains(reposName, "://") {
            // It cannot contain a scheme!
            return "", "", ErrInvalidRepositoryName
        }
        nameParts := strings.SplitN(reposName, "/", 2)
        if len(nameParts) == 1
|| (!strings.Contains(nameParts[0], ".") && !strings.Contains(nameParts[0], ":") &&
        nameParts[0] != "localhost") {
            // This is a Docker Index repos (ex: samalba/hipache or ubuntu)
            err := validateRepositoryName(reposName)
            return IndexServerAddress(), reposName, err
        }
        hostname := nameParts[0]
        reposName = nameParts[1]
        if strings.Contains(hostname, "index.docker.io") {
            return "", "", fmt.Errorf("Invalid repository name, try \"%s\" instead", reposName)
        }
        if err := validateRepositoryName(reposName); err != nil {
            return "", "", err
        }

        return hostname, reposName, nil
}
```

ResolveRepositoryName函数首先通过”/”分割字符串reposName，如下：

```
nameParts := strings.SplitN(reposName, "/", 2)
```

如果nameParts的长度为1，则说明reposName中不含有字符”/”，意味着用户没有指定Docker Registry。另外，形如”samalba/hipache”的reposName同样说明用户并没有指定Docker Registry。当用户没有指定Docker Registry时，Docker Client默认返回IndexServerAddress()，该函数返回常量INDEXSERVER，值为”https://index.docker.io/v1”。也就是说，当用户下载Docker镜像时，若不指定Docker Registry，默认情况下，Docker Client通知Docker Daemon去Docker Hub上下载镜像。例如：请求docker pull ubuntu:14.04，由于没有指定Docker Registry，Docker Client默认使用全球最大的Docker Registry——Docker Hub。

当不满足返回默认Docker Registry时，Docker Client通过解析reposNames，得出用户指定的Docker Registry地址。例如：请求docker pull localhost.localdomain:5000/docker/ubuntu:14.04中，解析出的Docker Registry地址为localhost.localdomain:5000。

至此，与Docker镜像相关的参数已经全部解析完毕，Docker Client将携带这部分重要信息，以及用户的认证信息，构建RESTful请求，发送给Docker Server。

4.2 配置认证信息

用户下载Docker镜像时，Docker同样支持用户信息的认证。用户认证信息由Docker Client配置；Docker Client发送请求至Docker Server时，用户认证信息也被一并发送；随后，Docker Daemon处理下载Docker镜像请求时，用户认证信息在Docker Registry被验证。

Docker Client配置用户认证信息包含两个步骤，实现源码如下：

```
cli.LoadConfigFile()
    // Resolve the Auth config relevant for this server
    authConfig := cli.configFile.ResolveAuthConfig(hostname)
```

可见，第一个步骤是使cli（Docker Client）加载ConfigFile，ConfigFile是Docker Client用来存放有关Docker Registry的用户认证信息的对象。DockerCli、ConfigFile以及AuthConfig三种数据结构之间的关系如图4.1：

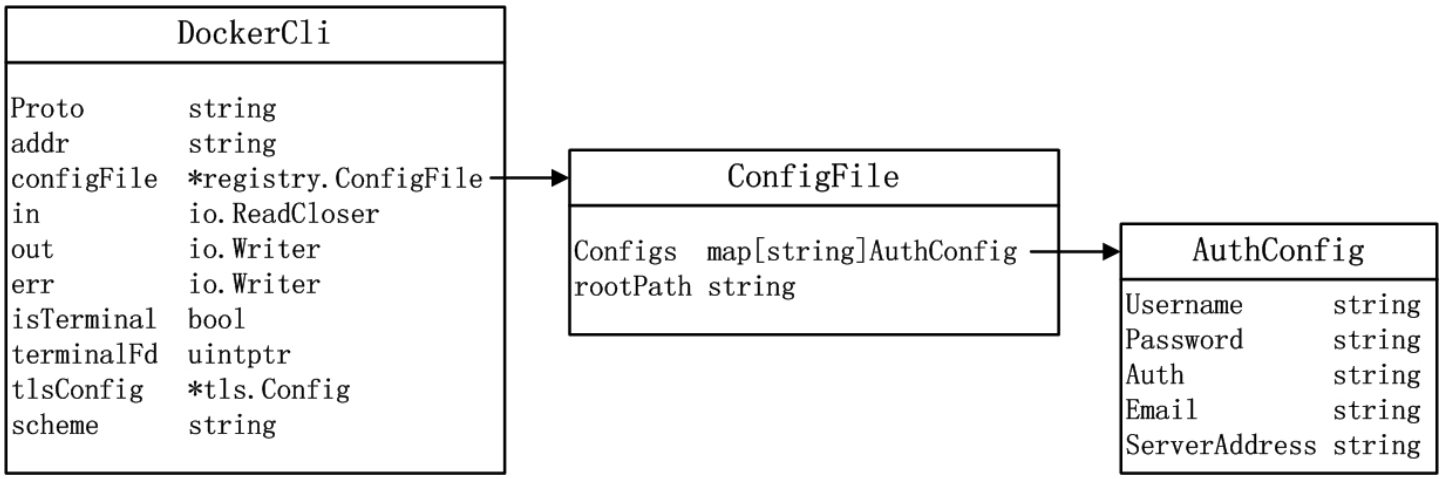


图4.1 DockerCli、ConfigFile以及AuthConfig关系图

DockerCli结构体的属性configFile为一个指向registry.ConfigFile的指针，而ConfigFile结构体的属性Configs属于map类型，其中key为string，代表Docker Registry的地址，value的类型为AuthConfig。AuthConfig类型具体含义为用户在某个Docker Registry上的认证信息，包含用户名，密码，认证信息，邮箱地址等。

加载完用户所有的认证信息之后，Docker Client第二个步骤是：通过用户指定的Docker Registry，即之前解析出的hostname参数，从用户所有的认证信息中找出与指定hostname相匹配的认证信息。新创建的authConfig，类型即为AuthConfig，将会作为用户在指定Docker Registry上的认证信息，发送至Docker Server。

4.3 发送API请求

Docker Client解析完所有的Docker镜像参数，并且配置完毕用户的认证信息之后，Docker Client需要使用这些信息正式发送镜像下载的请求至Docker Server。

Docker Client定义了pull函数，来实现发送镜像下载请求至Docker Server，源码位于./docker/api/client/commands.go#L1217-L1229，如下：

```
pull := func(authConfig registry.AuthConfig) error {
        buf, err := json.Marshal(authConfig)
        if err != nil {
            return err
        }
        registryAuthHeader := []string{
            base64.URLEncoding.EncodeToString(buf),
        }

        return cli.stream("POST", "/images/create?"+v.Encode(), nil, cli.out, map[string][]string{
            "X-Registry-Auth": registryAuthHeader,
        })
    }
```

pull函数的实现较为简单，首先通过authConfig对象，创建registryAuthHeader，最后发送POST请求，请求的URL为"/images/create?"+v.Encode()，在URL中传入查询参数包括”fromImage”与”tag”，另外在请求的HTTP Header中添加认证信息registryAuthHeader，。

执行以上的pull函数时，Docker镜像下载请求被发送，随后Docker Client等待Docker Server的接收、处理与响应。

5.Docker Server

Docker Server作为Docker Daemon的入口，所有Docker Client发送请求都由Docker Server接收。Docker Server通过解析请求的URL与请求方法，最终路由分发至相应的handler来处理。Docker Server的创建与请求处理，可以参看《Docker源码分析》系列之Docker Server篇。

Docker Server接收到镜像下载请求之后，通过路由分发最终由具体的handler——postImagesCreate来处理。postImagesCreate的实现位于./docker/api/server/server.go#L466-L524，的、其执行流程主要分为3个部分：

(1) 解析HTTP请求中包含的请求参数，包括URL中的查询参数、HTTP header中的认证信息等；

(2) 创建镜像下载job，并为该job配置环境变量；

(3) 触发执行镜像下载job。

5.1 解析请求参数

Docker Server接收到Docker Client发送的镜像下载请求之后，首先解析请求参数，并未后续job的创建与运行提供参数依据。Docker Server解析的请求参数，主要有：HTTP请求URL中的查询参数”fromImage”、” repo”以及”tag” ，以及有HTTP请求的header中的”X-Registry-Auth”。

请求参数解析的源码如下：

```
var (
        image = r.Form.Get("fromImage")
        repo  = r.Form.Get("repo")
        tag   = r.Form.Get("tag")
        job   *engine.Job
    )
    authEncoded := r.Header.Get("X-Registry-Auth")
```

需要特别说明的是：通过”fromImage”解析出的image变量包含镜像repository名称与镜像tag信息。例如用户请求为docker pull ubuntu:14.04，那么通过”fromImage”解析出的image变量值为ubuntu:14.04，并非只有Docker镜像的名称。

另外，Docker Server通过HTTP header中解析出authEncoded，还原出类型为registry.AuthConfig的对象authConfig，源码实现如下：

```
authConfig := ®istry.AuthConfig{}
    if authEncoded != "" {
        authJson := base64.NewDecoder(base64.URLEncoding, strings.NewReader(authEncoded))
        if err := json.NewDecoder(authJson).Decode(authConfig); err != nil {
            // for a pull it is not an error if no auth was given
            // to increase compatibility with the existing api it is defaulting to be empty
            authConfig = ®istry.AuthConfig{}
        }
    }
```

解析出HTTP请求中的参数之后，Docker Server对于image参数，再次进行解析，从中解析出属于repository与tag信息，其中repository有可能暂时包含Docker Registry信息，源码实现如下：

```
if tag == "" {
        image, tag = parsers.ParseRepositoryTag(image)
    }
```

Docker Server的参数解析工作至此全部完成，在这之后Docker Server将创建镜像下载任务并开始执行。

5.2 创建并配置job

Docker Server只负责接收Docker Client发送的请求，并将其路由分发至相应的handler来处理，最终的请求执行还是需要Docker Daemon来协作完成。Docker Server在handler中，通过创建job并触发job执行的形式，把控制权交于Docker Daemon。

Docker Server创建镜像下载job并配置环境变量的源码实现如下：

```
job = eng.Job("pull", image, tag)
    job.SetenvBool("parallel", version.GreaterThan("1.3"))
    job.SetenvJson("metaHeaders", metaHeaders)
    job.SetenvJson("authConfig", authConfig)
```

其中，创建的job名为pull，含义是下载Docker镜像，传入参数为image与tag，配置的环境变量有parallel、metaHeaders与authConfig。

5.3 触发执行job

Docker Server创建完Docker镜像下载job之后，需要触发执行该job，实现将控制权交于Docker Daemon。

Docker Server触发执行job的源码如下：

```
if err := job.Run(); err != nil {
        if !job.Stdout.Used() {
            return err
        }
        sf := utils.NewStreamFormatter(version.GreaterThan("1.0"))
        w.Write(sf.FormatError(err))
    }
```

由于Docker Daemon在启动时，已经配置了名为”pull”的job所对应的handler，实际为graph包中的CmdPull函数，故一旦该job被触发执行，控制权将直接交于Docker Daemon的CmdPull函数。Docker Daemon启动时Engine的handler注册，可以参见《Docker源码分析》系列的第三篇——Docker Daemon启动篇。

6.Docker Daemon

Docker Daemon是完成job执行的主要载体。Docker Server为镜像下载job准备好所有的参数配置之后，只等Docker Daemon来完成执行，并返回相应的信息，Docker Server再将响应信息返回至Docker Client。Docker Daemon对于镜像下载job的执行，涉及的内容较多：首先解析job参数，获取Docker镜像的repository、tag、Docker Registry信息等；随后与Docker Registry建立session；然后通过session下载Docker镜像；接着将Docker镜像下载至本地并存储于graph；最后在TagStore标记该镜像。

Docker Daemon对于镜像下载job的执行主要依靠CmdPull函数。这个CmdPull函数与Docker Client的CmdPull函数完全不同，前者是为了代替用户发送镜像下载的请求至Docker Daemon，而Docker Daemon的CmdPull函数则是实现代替用户真正完全镜像下载的任务。调用CmdPull函数的对象类型为TagStore，其源码实现位于./docker/graph/pull.go。

6.1 解析job参数

正如Docker Client与Docker Server，Docker Daemon执行镜像下载job时的第一个步骤也是解析参数。解析工作一方面确保传入参数无误，另一方面按需为job提供参数依据。表6.1罗列Docker Daemon解析的job参数，如下：

表6.1 Docker Daemon解析job参数列表

参数名称	参数描述
localName	代表镜像的repository信息，有可能携带Docker Registry信息
tag	代表镜像的标签信息，默认为latest
authConfig	代表用户在指定Docker Registry上的认证信息
metaHeaders	代表请求中的header信息
hostname	代表Docker Registry信息，从localName解析获得，默认为Docker Hub地址
remoteName	代表Docker镜像的repository名称信息，不携带Docker Registry信息
endpoint	代表Docker Registry完整的URL，从hostname扩展获得

参数解析过程中，Docker Daemon还添加了一些精妙的设计。如：在TagStore类型中设计了pullingPool对象，用于保存正在被下载的Docker镜像，下载完毕之前禁止其他Docker Client发起相同镜像的下载请求，下载完毕之后pullingPool中的该记录被清除。Docker Daemon一旦解析出localName与tag两个参数信息，则立即检测pullingPool，实现源码位于./docker/graph/pull.go#L36-L46，如下：

```
c, err := s.poolAdd("pull", localName+":"+tag)
    if err != nil {
        if c != nil {
            // Another pull of the same repository is already taking place; just wait for it to finish
            job.Stdout.Write(sf.FormatStatus("", "Repository %s already being pulled by another client. Wa
iting.", localName))
            <-c
            return engine.StatusOK
        }
        return job.Error(err)
    }
    defer s.poolRemove("pull", localName+":"+tag)
```

6.2 创建session对象

下载Docker镜像，Docker Daemon与Docker Registry需要建立通信。为了保障两者通信的可靠性，Docker Daemon采用了session机制。Docker Daemon每收到一个Docker Client的镜像下载请求，都会创建一个与相应Docker Registry的session，之后所有的网络数据传输都在该session上完成。包registry定义了session，位于./docker/registry/registry.go，如下：

```
type Session struct {
        authConfig    *AuthConfig
        reqFactory    *utils.HTTPRequestFactory
        indexEndpoint string
        jar           *cookiejar.Jar
        timeout       TimeoutType
}
```

CmdPull函数中创建session的源码实现如下：

```
r, err := registry.NewSession(authConfig, registry.HTTPRequestFactory
(metaHeaders), endpoint, true)
```

创建的session对象为r，在下一阶段的镜像下载过程中，多数与镜像相关的数据传输均在r这个seesion的基础上完成。

6.3 执行镜像下载

Docker Daemon之前所有的操作，都属于配置阶段，从解析job参数，到建立session对象，而并未与Docker Registry建立实际的连接，并且也还未真正传输过有关Docker镜像的内容。

完成所有的配置之后，Docker Daemon进入Docker镜像下载环节，实现Docker镜像下载的源码位于./docker/graph/pull.go#L69-L71，如下：

```
if err = s.pullRepository(r, job.Stdout, localName, remoteName, tag, sf,
job.GetenvBool("parallel")); err != nil {
        return job.Error(err)
    }
```

以上代码中pullRepository函数包含了镜像下载整个流程的林林总总，该流程可以参见图6.1：

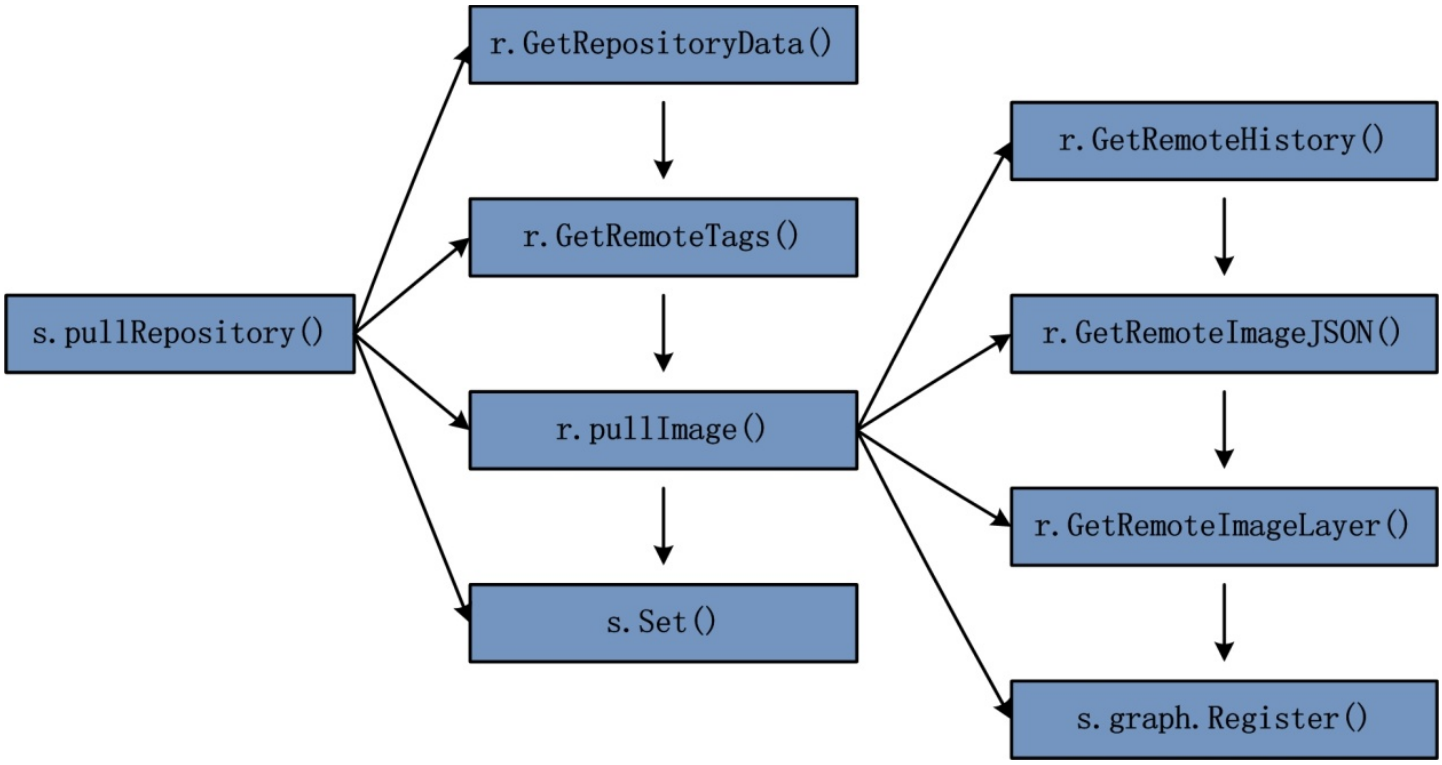


图6.1 pullRepository流程图

关于上图的各个环节，下表给出简要的功能介绍：

表6.2 pullRepository各环节功能介绍表

函数名称	功能介绍
r.GetRepositoryData()	获取指定repository中所有image的id信息
r.GetRemoteTags()	获取指定repository中所有的tag信息
r.pullImage()	从Docker Registry下载Docker镜像
r.GetRemoteHistory()	获取指定image所有祖先image id信息
r.GetRemoteImageJSON()	获取指定image的json信息
r.GetRemoteImageLayer()	获取指定image的layer信息
s.graph.Register()	将下载的镜像在TagStore的graph中注册
s.Set()	在TagStore中添加新下载的镜像信息

分析pullRepository的整个流程之前，很有必要了解下pullRepository函数调用者的类型TagStore。TagStore是Docker镜像方面涵盖内容最多的数据结构：一方面TagStore管理Docker的Graph，另一方面TagStore还管理Docker的repository记录。除此之外，TagStore还管理着上文提到的对象pullingPool以及pushingPool，保证Docker Daemon在同一时刻，只为一个Docker Client执行同一镜像的下载或上传。TagStore结构体的定义位于./docker/graph/tags.go#L20-L29，如下：

```
type TagStore struct {
        path         string
        graph        *Graph
        Repositories map[string]Repository
        sync.Mutex
        // FIXME: move push/pull-related fields
        // to a helper type
        pullingPool map[string]chan struct{}
        pushingPool map[string]chan struct{}
}
```

以下将重点分析pullRepository的整个流程。

6.3.1 GetRepositoryData

使用Docker下载镜像时，用户往往指定的是Docker镜像的名称，如：请求docker pull ubuntu:14.04中镜像名称为ubuntu。GetRepositoryData的作用则是获取镜像名称所在repository中所有image的 id信息。

GetRepositoryData的源码实现位于./docker/registry/session.go#L255-L324。获取repository中image的ID信息的目标URL地址如以下源码：

```
repositoryTarget := fmt.Sprintf("%srepositories/%s/images", indexEp, remote)
```

因此，docker pull ubuntu:14.04请求被执行时，repository的目标URL地址为https://index.docker.io/v1/repositories/ubuntu/images，访问该URL可以获得有关ubuntu这个repository中所有image的 id信息，部分image的id信息如下：

```
[{"checksum": "", "id": "
2427658c75a1e3d0af0e7272317a8abfaee4c15729b6840e3c2fca342fe47bf1"},
{"checksum": "", "id":
"81fbd8fa918a14f4ebad9728df6785c537218279081c7a120d72399d3a5c94a5"
}, {"checksum": "", "id":
"ec69e8fd6b0236b67227869b6d6d119f033221dd0f01e0f569518edabef3b72c"
}, {"checksum": "", "id":
"9e8dc15b6d327eaac00e37de743865f45bee3e0ae763791a34b61e206dd5222e"
}, {"checksum": "", "id":
"78949b1e1cfdcd5db413c300023b178fc4b59c0e417221c0eb2ffbbd1a4725cc"
},.....]
```

获取以上信息之后，Docker Daemon通过RepositoryData和ImgData类型对象来存储ubuntu这个repository中所有image的信息，RepositoryData和ImgData的数据结构关系如图6.2：

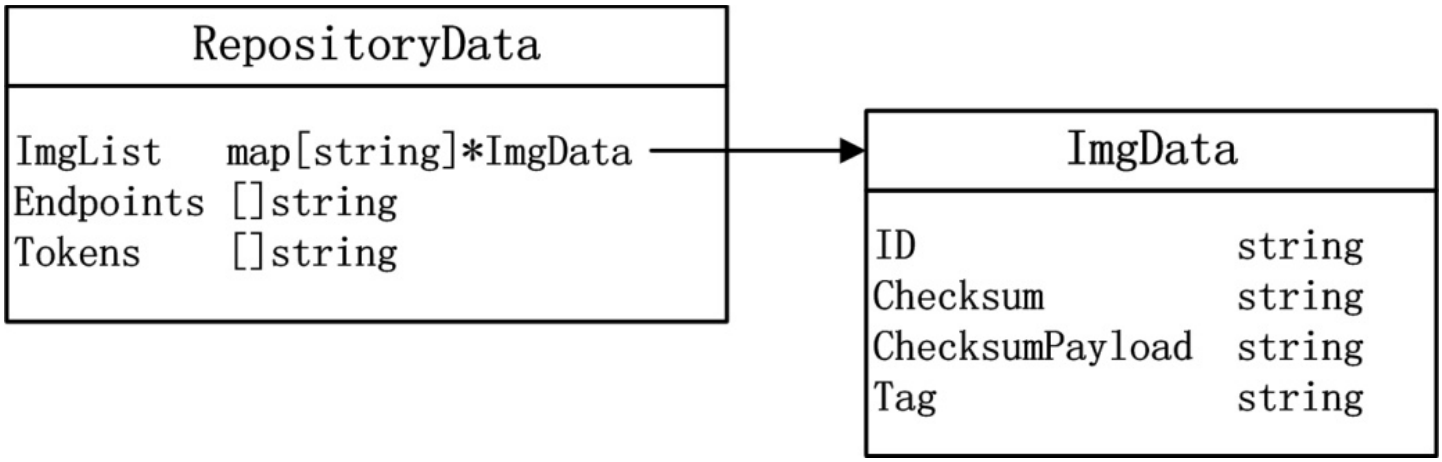


图6.2 RepositoryData和ImgData的数据结构关系图

GetRepositoryData执行过程中，会为指定repository中的每一个image创建一个ImgData对象，并最终将所有ImgData存放在RepositoryData的ImgList属性中，ImgList的类型为map，key为image的ID，value指向ImgData对象。此时ImgData对象中只有属性ID与Checksum有内容。

6.3.2 GetRemoteTags

使用Docker下载镜像时，用户除了指定Docker镜像的名称之外，一般还需要指定Docker镜像的tag，如：请求docker pull ubuntu:14.04中镜像名称为ubuntu，镜像tag为14.04，假设用户不显性指定tag，则默认tag为latest。GetRemoteTags的作用则是获取镜像名称所在repository中所有tag的信息。

GetRemoteTags的源码实现位于./docker/registry/session.go#L195-234。获取repository中所有tag信息的目标URL地址如以下源码：

```
endpoint := fmt.Sprintf("%srepositories/%s/tags", host, repository)
```

获取指定repository中所有tag信息之后，Docker Daemon根据tag对应layer的ID，找到ImgData，并对填充ImgData中的Tag属性。此时，RepositoryData的ImgList属性中，有的ImgData对象有Tag内容，有的ImgData对象中没有Tag内容。这也和实际情况相符，如下载一个ubuntu:14.04镜像，该镜像的rootfs中只有最上层的layer才有tag信息，这一层layer的parent Image并不一定存在tag信息。

6.3.3 pullImage

Docker Daemon下载Docker镜像时是通过image id来完成。GetRepositoryData和GetRemoteTags则成功完成了用户传入的repository和tag信息与image id的转换。如请求docker pull ubuntu:14.04中，repository为ubuntu，tag为14.04，则对应的image id为2d24f826。

Docker Daemon获得下载镜像的image id之后，首先查验pullingPool，判断是否有其他Docker Client同样发起了该镜像的下载请求，如果没有的话Docker Daemon才继续下载任务。

执行pullImage函数的源码实现位于./docker/graph/pull.go#L159，如下：

```
s.pullImage(r, out, img.ID, ep, repoData.Tokens, sf)
```

而pullImage函数的定义位于./docker/graph/pull.go#L214-L301。图6.1中，可以看到pullImage函数的执行可以分为4个步骤：GetRemoteHistory、GetRemoteImageJson、GetRemoteImageLayer与s.graph.Register()。

GetRemoteHistory的作用很好理解，既然Docker Daemon已经通过GetRepositoryData和GetRemoteTags找出了指定tag的image id，那么Docker Daemon所需完成的工作为下载该image 及其所有的祖先image。GetRemoteHistory正是用于获取指定image及其所有祖先iamge的id。

GetRemoteHistory的源码实现位于./docker/registry/session.go#L72-L101。

获取所有的image id之后，对于每一个image id，Docker Daemon都开始下载该image的全部内容。Docker Image的全部内容包括两个方面：image json信息以及image layer信息。Docker所有image的json信息都由函数GetRemoteImageJSON来完成。分析GetRemoteImageJSON之前，有必要阐述清楚什么是Docker Image的json信息。

Docker Image的json信息是一个非常重要的概念。这部分json唯一的标志了一个image，不仅标志了image的id，同时也标志了image所在layer对应的config配置信息。理解以上内容，可以举一个例子：docker build。命令docker build用以通过指定的Dockerfile来创建一个Docker镜像；对于Dockerfile中所有的命令，Docker Daemon都会为其创建一个新的image，如：RUN apt-get update, ENV path=/bin, WORKDIR /home等。对于命令RUN apt-get update，Docker Daemon需要执行apt-get update操作，对应的rootfs上必定会有内容更新，导致新建的image所代表的layer中有新添加的内容。而如ENV path=/bin, WORKDIR /home这样的命令，仅仅是配置了一些容器运行的参数，并没有镜像内容的更新，对于这种情况，Docker Daemon同样创建一层新的layer，并且这层新的layer中内容为空，而命令内容会在这层image的json信息中做更新。总结而言，可以认为Docker的image包含两部分内容：image的json信息、layer内容。当layer内容为空时，image的json信息被更新。

清楚了Docker image的json信息之后，理解GetRemoteImageJSON函数的作用就变得十分容易。GetRemoteImageJSON的执行代码位于./docker/graph/pull.go#L243，如下：

```
imgJSON, imgSize, err = r.GetRemoteImageJSON(id, endpoint, token)
```

GetRemoteImageJSON返回的两个对象imgJSON代表image的json信息，imgSize代表镜像的大小。通过imgJSON对象，Docker Daemon立即创建一个image对象，创建image对象的源码实现位于./docker/graph/pull.go#L251，如下：

```
img, err = image.NewImgJSON(imgJSON)
```

而NewImgJSON函数位于包image中，函数返回类型为一个Image对象，而Image类型的定义而下：

```
type Image struct {
    ID              string            `json:"id"`
    Parent          string            `json:"parent,omitempty"`
    Comment         string            `json:"comment,omitempty"`
    Created         time.Time         `json:"created"`
    Container       string            `json:"container,omitempty"`
    ContainerConfig  runconfig.Config  `json:"container_config,omitempty"`
    DockerVersion    string            `json:"docker_version,omitempty"`
    Author          string            `json:"author,omitempty"`
    Config          *runconfig.Config `json:"config,omitempty"`
    Architecture     string            `json:"architecture,omitempty"`
    OS              string            `json:"os,omitempty"`
    Size            int64

    graph Graph
}
```

返回img对象，则说明关于该image的所有元数据已经保存完毕，由于还缺少image的layer中包含的内容，因此下一个步骤即为下载镜像layer的内容，调用函数为GetRemoteImageLayer，函数执行位于./docker/graph/pull.go#L270，如下：

```
layer, err := r.GetRemoteImageLayer(img.ID, endpoint, token, int64(imgSize))
```

GetRemoteImageLayer函数返回当前image的layer内容。Image的layer内容指的是：该image在parent image之上做的文件系统内容更新，包括文件的增添、删除、修改等。至此，image的json信息以及layer内容均被Docker Daemon获取，意味着一个完整的image已经下载完毕。下载image完毕之后，并不意味着Docker Daemon关于Docker镜像下载的job就此结束，Docker Daemon仍然需要对下载的image进行存储管理，以便Docker Daemon在执行其他如创建容器等job时，能够方便使用这些image。

Docker Daemon在graph中注册image的源码实现位于./docker/graph/pull.go#L283-L285，如下：

```
err = s.graph.Register(imgJSON,utils.ProgressReader(layer, imgSize,
out, sf, false, utils.TruncateID(id), "Downloading"),img)
```

Docker Daemon通过graph存储image是一个很重要的环节。Docker在1.2.0版本中可以通过AUFS、DevMapper以及BTRFS来进行image的存储。在Linux 3.18-rc2版本中，OverlayFS已经被内核合并，故从1.4.0版本开始，Docker 的image支持OverlayFS的存储方式。

Docker镜像的存储在Docker中是较为独立且重要的内容，故将在《Docker源码分析》系列的第十一篇专文分析。

6.3.4 配置TagStore

Docker镜像下载完毕之后，Docker Daemon需要在TagStore中指定的repository中添加相应的tag。每当用户查看本地镜像时，都可以从TagStore的repository中查看所有含有tag信息的image。

Docker Daemon配置TagStore的源码实现位于./docker/graph/pull.go#L206，如下：

```
if err := s.Set(localName, tag, id, true); err != nil {
        return err
    }
```

TagStore类型的Set函数定义位于./docker/graph/tags.go#L174-L205。Set函数的指定流程与简要介绍如图6.3：

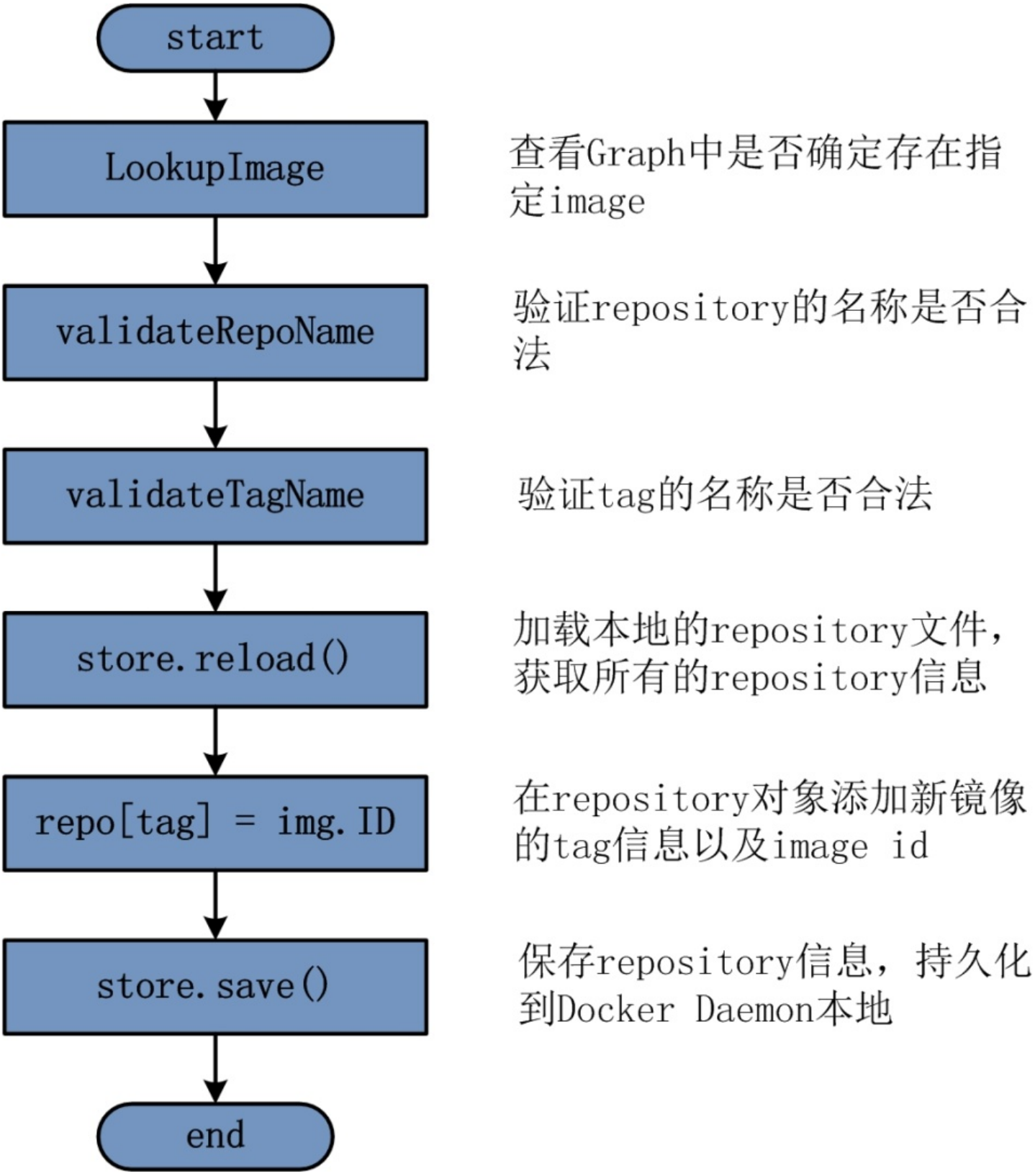


图6.3 TagStore中Set函数执行流程图

当Docker Daemon将已下载的Docker镜像信息同步到repository之后，Docker下载镜像的job就全部完成，Docker Daemon返回响应至Docker Server，Docker Server返回相应至Docker Client。注：本地的repository文件位于Docker的根目录，根目录一般为/var/lib/docker，如果使用aufs的graphdriver，则repository文件名为repositories-aufs。

7.总结

Docker镜像给Docker容器的运行带来了无限的可能性，诸如Docker Hub之类的Docker Registry又使得Docker镜像在全球的开发者之间共享。Docker镜像的下载，作为使用Docker的第一个步骤，Docker爱好者若能熟练掌握其中的原理，必定能对Docker的很多概念有更为清晰的认识，对Docker容器的运行、管理等均是有百利而无一害。

Docker镜像的下载需要Docker Client、Docker Server、Docker Daemon以及Docker Registry四者协同合作完成。本文从源码的角度分析了四者各自的扮演的角色，分析过程中还涉及多种Docker概念，如repository、tag、TagStore、session、image、layer、image json、graph等。

8.作者介绍

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

9.参考文献

https://docs.docker.com/terms/image/

https://docs.docker.com/terms/layer

http://docs.studygolang.com/pkg/

（十一）：镜像存储

- 1.前言
- 2.镜像注册
- 3.验证镜像ID
- 4.创建镜像路径
  - 4.1创建mnt、diff和layers
  - 4.2 mount祖先镜像并返回根目录
- 5.存储镜像内容
  - 5.1解压镜像内容
  - 5.2收集镜像大小并记录
  - 5.3存储jsonData信息
- 6.注册镜像ID
- 7.总结
- 8.作者介绍
- 参考文献

1.前言

Docker Hub汇总众多Docker用户的镜像，极大得发挥Docker镜像开放的思想。Docker用户在全球任意一个角度，都可以与Docker Hub交互，分享自己构建的镜像至Docker Hub，当然也完全可以下载另一半球Docker开发者上传至Docker Hub的Docker镜像。

无论是上传，还是下载Docker镜像，镜像必然会以某种形式存储在Docker Daemon所在的宿主机文件系统中。Docker镜像在宿主机的存储，关键点在于：在本地文件系统中以如何组织形式，被Docker Daemon有效的统一化管理。这种管理，可以使得Docker Daemon创建Docker容器服务时，方便获取镜像并完成union mount操作，为容器准备初始化的文件系统。

本文主要从Docker 1.2.0源码的角度，分析Docker Daemon下载镜像过程中存储Docker镜像的环节。分析内容的安排有以下5部分：

(1) 概述Docker镜像存储的执行入口，并简要介绍存储流程的四个步骤；

(2) 验证镜像ID的有效性；

(3) 创建镜像存储路径；

(4) 存储镜像内容；

(5) 在graph中注册镜像ID。

2.镜像注册

Docker Daemon执行镜像下载任务时，从Docker Registry处下载指定镜像之后，仍需要将镜像合理地存储于宿主机的文件系统中。更为具体而言，存储工作分为两个部分：

(1) 存储镜像内容；

(2) 在graph中注册镜像信息。

说到镜像内容，需要强调的是，每一层layer的Docker Image内容都可以认为有两个部分组成：镜像中每一层layer中存储的文件系统内容，这部分内容一般可以认为是未来Docker容器的静态文件内容；另一部分内容指的是容器的json文件，json文件代表的信息除了容器的基本属性信息之外，还包括未来容器运行时的动态信息，包括ENV等信息。

存储镜像内容，意味着Docker Daemon所在宿主机上已经存在镜像的所有内容，除此之外，Docker Daemon仍需要对所存储的镜像进行统计备案，以便用户在后续的镜像管理与使用过程中，可以有据可循。为此，Docker Daemon设计了graph，使用graph来接管这部分的工作。graph负责记录有哪些镜像已经被正确存储，供Docker Daemon调用。

Docker Daemon执行CmdPull任务的pullImage阶段时，实现Docker镜像存储与记录的源码位于./docker/graph/pull.go#L283-L285，如下：

```
err = s.graph.Register(imgJSON,utils.ProgressReader(layer,
 imgSize, out, sf, false, utils.TruncateID(id), “Downloading”),img)
```

以上源码的实现，实际调用了函数Register，Register函数的定义位于./docker/graph/graph.go#L162-L218:

```
func (graph *Graph) Register(jsonData []byte, layerData
archive.ArchiveReader, img *image.Image) (err error)
```

分析以上Register函数定义，可以得出以下内容：

(1) 函数名称为Register；

(2) 函数调用者类型为Graph；

(3) 函数传入的参数有3个，第一个为jsonData，类型为数组，第二个为layerData，类型为archive.ArchiveReader，第三个为img,类型为*image.Image；

(4) 函数返回对象为err，类型为error。

Register函数的运行流程如图11-1所示:

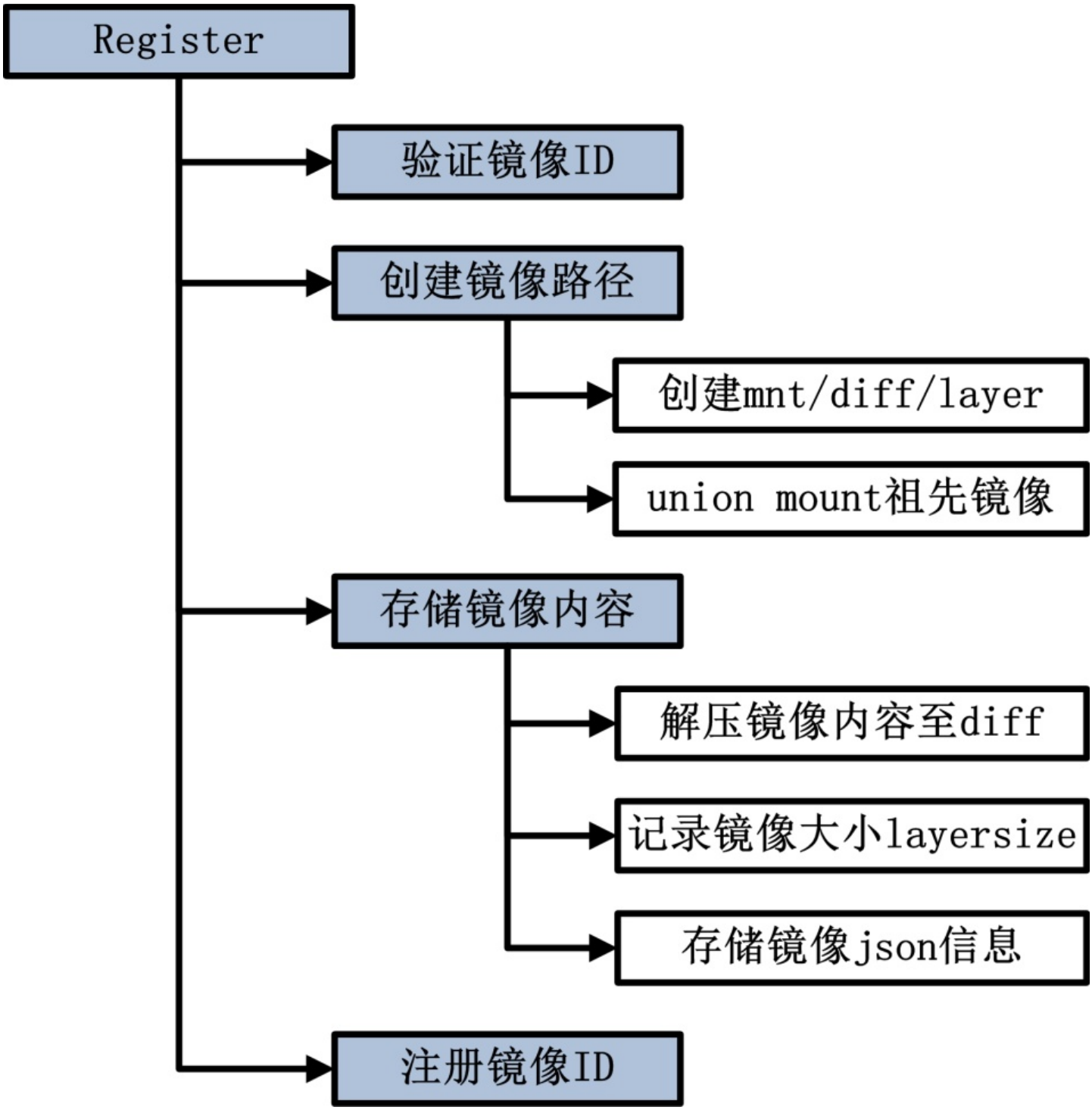


图11-1 Register函数执行流程图

3.验证镜像ID

Docker镜像注册的第一个步骤是验证Docker镜像的ID。此步骤主要为确保镜像ID命名的合法性。功能而言，这部分内容提高了Docker镜像存储环节的鲁棒性。验证镜像ID由三个环节组成。

(1) 验证镜像ID的合法性；

(2) 验证镜像是否已存在；

(3) 初始化镜像目录。

验证镜像ID的合法性使用包utils中的ValidateID函数完成，实现源码位

于./docker/graph/graph.go#L171-L173，如下：

```
if err := utils.ValidateID(img.ID); err != nil {
    return err
}
```

ValidateID函数的实现过程中，Docker Dameon检验了镜像ID是否为空，以及镜像ID中是否存在字符 ':' ，以上两种情况只要成立其中之一，Docker Daemon即认为镜像ID不合法，不予执行后续内容。

镜像ID的合法性验证完毕之后，Docker Daemon接着验证镜像是否已经存在于graph。若该镜像已经存在于graph，则Docker Daemon返回相应错误，不予执行后续内容。代码实现如下：

```
    if graph.Exists(img.ID) {
        return fmt.Errorf("Image %s already exists", img.ID)
    }
```

验证工作完成之后，Docker Daemon为镜像准备存储路径。该部分源码实现位于./docker/graph/graph.go#L182-L196，如下：

```
if err := os.RemoveAll(graph.ImageRoot(img.ID)); err != nil && !os.IsNotExist(err) {
        return err
    }

    // If the driver has this ID but the graph doesn't, remove it from the driver to start fresh.
    // (the graph is the source of truth).
    // Ignore errors, since we don't know if the driver correctly returns ErrNotExist.
    // (FIXME: make that mandatory for drivers).
    graph.driver.Remove(img.ID)

    tmp, err := graph.Mktemp("")
    defer os.RemoveAll(tmp)
    if err != nil {
        return fmt.Errorf("Mktemp failed: %s", err)
    }
```

Docker Daemon为镜像初始化存储路径，实则首先删除属于新镜像的存储路径，即如果该镜像路径已经在文件系统中存在的话，立即删除该路径，确保镜像存储时不会出现路径冲突问题；接着还删除graph.driver中的指定内容，即如果该镜像在graph.driver中存在的话，unmount该镜像在宿主机上的目录，并将该目录完全删除。以AUFS这种类型的graphdriver为例，镜像内容被存放在/var/lib/docker/aufs/diff目录下，而镜像会被mount至目录/var/lib/docker/aufs/mnt下的指定位置。

至此，验证Docker镜像ID的工作已经完成，并且Docker Daemon已经完成对镜像存储路径的初始化，使得后续Docker镜像存储时存储路径不会冲突，graph.driver对该镜像的mount也不会冲突。

4.创建镜像路径

创建镜像路径，是镜像存储流程中的一个必备环节，这一环节直接让Docker使用者了解以下概念：镜像以何种形式存在于本地文件系统的何处。创建镜像路径完毕之后，Docker Daemon首先将镜像的所有祖先镜像通过aufs文件系统mount至mnt下的指定点，最终直接返回镜像所在rootfs的路径，以便后续直接在该路径下解压Docker镜像的具体内容（只包含layer内容）。

4.1创建mnt、diff和layers

创建镜像路径的源码实现位于./docker/graph/graph.go#L198-L206， 如下：

```
// Create root filesystem in the driver
if err := graph.driver.Create(img.ID, img.Parent); err != nil {
        return fmt.Errorf("Driver %s failed to create image rootfs %s: %s", graph.driver, img.ID, err)
}
// Mount the root filesystem so we can apply the diff/layer
rootfs, err := graph.driver.Get(img.ID, "")
if err != nil {
        return fmt.Errorf("Driver %s failed to get image rootfs %s: %s", graph.driver, img.ID, err)
}
```

以上源码中Create函数在创建镜像路径时起到举足轻重的作用。那我们首先分析graph.driver.Create(img.ID, img.Parent)的具体实现。由于在Docker Daemon启动时，注册了具体的graphdriver，故graph.driver实际的值为具体注册的driver。方便起见，本章内容全部以aufs类型为例，即在graph.driver为aufs的情况下，阐述Docker镜像的存储。在ubuntu 14.04系统上，Docker Daemon的根目录一般为/var/lib/docker,而aufs类型driver的镜像存储路径一般为/var/lib/docker/aufs。

AUFS这种联合文件系统的实现，在union多个镜像时起到至关重要的作用。首先来关注，Docker Daemon如何为镜像创建镜像路径，以便支持通过aufs来union镜像。Aufs模式下，graph.driver.Create(img.ID, img.Parent)的具体源码实现位于./docker/daemon/graphdriver/aufs/aufs.go#L161-L190,如下：

```
// Three folders are created for each id
// mnt, layers, and diff
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 {
            return err
        }

        if _, err := fmt.Fprintln(f, parent); err != nil {
            return err
        }
        for _, i := range ids {
            if _, err := fmt.Fprintln(f, i); err != nil {
                return err
            }
        }
    }
    return nil
}
```

在Create函数的实现过程中，createDirsFor函数在Docker Daemon根目录下的aufs目录/var/lib/docker/aufs中，创建指定的镜像目录。若当前aufs目录下，还不存在mnt、diff这两个目录，则会首先创建mnt、diff这两个目录，并在这两个目录下分别创建代表镜像内容的文件夹，文件夹名为镜像ID，文件权限为0755。假设下载镜像的镜像ID为image_ID，则创建完毕之后，文件系统中的文件为/var/lib/docker/aufs/mnt/image_ID与/var/lib/docker/aufs/diff/image_ID。回到Create函数中，执行完createDirsFor函数之后，随即在aufs目录下创建了layers目录，并在layers目录下创建image_ID文件。

如此一来，在aufs下的三个子目录mnt，diff以及layers中，分别创建了名为镜像名image_ID的文件。继续深入分析之前，我们直接来看Docker对这三个目录mnt、diff以及layers的描述，如图11-2所示：

```
aufs driver directory structure

.
├── layers // Metadata of layers
│   ├── 1
│   ├── 2
│   └── 3
├── diff  // Content of the layer
│   ├── 1  // Contains layers that need to be mounted for the id
│   ├── 2
│   └── 3
└── mnt    // Mount points for the rw layers to be mounted
    ├── 1
    ├── 2
    └── 3
```

图11-2 aufs driver目录结构图

简要分析图11-2，图中的layers、diff以及mnt为目录/var/lib/docker/aufs下的三个子目录，1、2、3是镜像ID，分别代表三个镜像，三个目录下的1均代表同一个镜像ID。其中layers目录下保留每一个镜像的元数据，这些元数据是这个镜像的祖先镜像ID列表；diff目录下存储着每一个镜像所在的layer，具体包含的文件系统内容；mnt目录下每一个文件，都是一个镜像ID，代表在该层镜像之上挂载的可读写layer。因此，下载的镜像中与文件系统相关的具体内容，都会存储在diff目录下的某个镜像ID目录下。

再次回到Create函数，此时mnt，diff以及layer三个目录下的镜像ID文件已经创建完毕。下一步需要完成的是：为layers目录下的镜像ID文件填充元数据。元数据内容为该镜像所有的祖先镜像ID列表。填充元数据的流程如下：

(1) Docker Daemon首先通过f, err := os.Create(path.Join(a.rootPath(), "layers", id))打开layers目录下镜像ID文件；

(2) 然后，通过ids, err := getParentIds(a.rootPath(), parent)获取父镜像的祖先镜像ID列表ids；

(3) 其次，将父镜像镜像ID写入文件f；

(4) 最后，将父镜像的祖先镜像ID列表ids写入文件f。

最终的结果是：该镜像的所有祖先镜像的镜像ID信息都写入layers目录下该镜像ID文件中。

4.2 mount祖先镜像并返回根目录

Create函数执行完毕，意味着创建镜像路径并配置镜像元数据完毕，接着Docker Daemon返回了镜像的根目录，源码实现如下：

```
rootfs, err := graph.driver.Get(img.ID, "")
```

Get函数看似返回了镜像的根目录rootfs，实则执行了更为重要的内容——挂载祖先镜像文件系统。具体而言，Docker Daemon为当前层的镜像完成所有祖先镜像的Union Mount。Mount完毕之后，当前镜像的read-write层位于/var/lib/docker/aufs/mnt/image_ID。Get函数的具体实现位于./docker/daemon/graphdriver/aufs/aufs.go#L247-L278，如下：

```
func (a *Driver) Get(id, mountLabel string) (string, error) {
    ids, err := getParentIds(a.rootPath(), id)
    if err != nil {
        if !os.IsNotExist(err) {
            return "", err
        }
        ids = []string{}
    }

    // Protect the a.active from concurrent access
    a.Lock()
    defer a.Unlock()

    count := a.active[id]

    // If a dir does not have a parent ( no layers )do not try to mount
    // just return the diff path to the data
    out := path.Join(a.rootPath(), "diff", id)
    if len(ids) > 0 {
        out = path.Join(a.rootPath(), "mnt", id)

        if count == 0 {
            if err := a.mount(id, mountLabel); err != nil {
                return "", err
            }
        }
    }

    a.active[id] = count + 1

    return out, nil
}
```

分析以上Get函数的定义，可以得出以下内容：

(1) 函数名为Get；

(2) 函数调用者类型为Driver；

(3) 函数传入参数有两个：id与mountlabel；

(4) 函数返回内容有两部分：string类型的镜像根目录与错误对象error。

清楚Get函数的定义，再来看Get函数的实现。分析Get函数实现时，有三个部分较为关键，分别是Driver实例a的active属性、mount操作、以及返回值out。

首先分析Driver实例a的active属性。分析active属性之前，需要追溯到Aufs类型的graphdriver中Driver类型的定义以及graphdriver与graph的关系。两者的关系如图11-3所示：

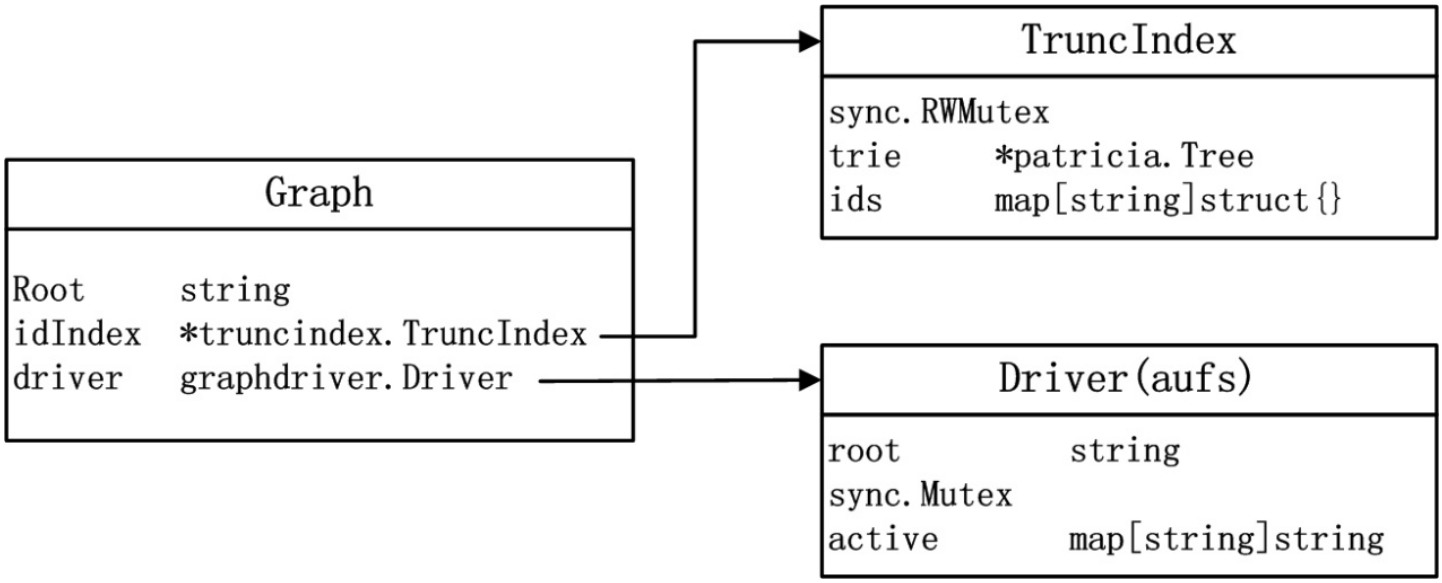


图11-3 graph与graphdriver关系图

Driver类型的定义位于./docker/daemon/graphdriver/aufs/aufs#L53-L57，如下：

```
type Driver struct {
    root      string
    sync.Mutex // Protects concurrent modification to active
    active    map[string]int
}
```

Driver结构体中root属性代表graphdriver所在的根目录，为/var/lib/docker/aufs。active属性为map类型，key为string，具体运用时key为Docker Image的ID，value为int类型，代表该层镜像layer被引用的次数总和。Docker镜像技术中，某一层layer的Docker镜像被引用一次，则active属性中key为该镜像ID的value值会累加1。用户执行镜像删除操作时，Docker Dameon会检查该Docker镜像的引用次数是否为0，若引用次数为0，则可以彻底删除该镜像，若不是的话，则仅仅将active属性中引用参数减1。属性sync.Mutex用于多个Job同时操作active属性时，确保active数据的同步工作。

接着，进入mount操作的分析。一旦Get参数传入的镜像ID参数不是一个Base Image，那么说明该镜像存在父镜像，Docker Daemon需要将该镜像所有的祖先镜像都mount到指定的位置，指定位置为/var/lib/docker/aufs/mnt/image_ID。所有祖先镜像的原生态文件系统内容分别位于/var/lib/docker/aufs/diff/。其中mount函数用以实现该部分描述的功能，mount的过程包含很多与aufs文件系统相关的参数配置与系统调用。

最后，Get函数返回out与nil。其中out的值为/var/lib/docker/aufs/mnt/image_ID，即使用该层Docker镜像时其根目录所在路径，也可以认为是镜像的RW层所在路径，但一旦该层镜像之上还有镜像，那么在mount后者之后，在上层镜像看来，下层镜像仍然是只读文件系统。

5.存储镜像内容

存储镜像内容，Docker Daemon的运行意味着已经验证过镜像ID，同时还为镜像准备了存储路径，并返回了其所有祖先镜像union mount后的路径。万事俱备，只欠“镜像内容的存储”。

Docker Daemon存储镜像具体内容完成的工作很简单，仅仅是通过某种合适的方式将两部分内容存储于本地文件系统并进行有效管理，它们是：镜像压缩内容、镜像json信息。

存储镜像内容的源码实现位于./docker/graph/graph.go#L209-L211，如下：

```
if err := image.StoreImage(img, jsonData, layerData, tmp, rootfs); err != nil {
    return err
}
```

其中，StoreImage函数的定义位于./docker/docker/image/image.go#L74，如下：

```
func StoreImage(img *Image, jsonData []byte, layerData
archive.ArchiveReader, root, layer string) error {
```

分析StoreImage函数的定义，可以得出以下信息：

(1) 函数名称：StoreImage；

(2) 函数传入参数名：img，jsonData，layerData，root，layer；

(3) 函数返回类型error。

简要分析传入参数的含义如表11-1所示：

表11-1 StoreImage函数参数表

参数名称	参数含义
img	通过下载的imgJSON信息创建出的Image对象实例
jsonData	Docker Daemon之前下载的imgJSON信息
layerData	镜像作为一个layer的压缩包，包含镜像的具体文件内容
root	graphdriver根目录下创建的临时文件”_tmp”,值为/var/lib/docker/aufs/_tmp
layer	Mount完所有祖先镜像之后，该镜像在mnt目录下的路径

掌握StoreImage函数传入参数的含义之后，理解其实现就十分简单。总体而言，StoreImage亦可以分为三个步骤：

(1) 解压镜像内容layerData至diff目录；

(2) 收集镜像所占空间大小，并记录；

(3) 将jsonData信息写入临时文件。

以下详细深入三个步骤的实现。

5.1解压镜像内容

StoreImage函数传入的镜像内容是一个压缩包，Docker Daemon理应在镜像存储时将其解压，为后续创建容器时直接使用镜像创造便利。

既然是解压镜像内容，那么这项任务的完成，除了需要代表镜像的压缩包之后，还需要解压任务的目标路径，以及解压时的参数。压缩包为StoreImage传入的参数layerData，而目标路径为/var/lib/docker/aufs/diff/。解压流程的执行源代码位于./docker/docker/image/image.go#L85-L120，如下：

```
// If layerData is not nil, unpack it into the new layer
    if layerData != nil {
        if differ, ok := driver.(graphdriver.Differ); ok {
            if err := differ.ApplyDiff(img.ID, layerData); err != nil {
                return err
            }

            if size, err = differ.DiffSize(img.ID); err != nil {
                return err
            }
        } else {
            start := time.Now().UTC()
            log.Debugf("Start untar layer")
            if err := archive.ApplyLayer(layer, layerData); err != nil {
                return err
            }
            log.Debugf("Untar time: %vs", time.Now().UTC().Sub(start).Seconds())

            if img.Parent == "" {
                if size, err = utils.TreeSize(layer); err != nil {
                    return err
                }
            } else {
                parent, err := driver.Get(img.Parent, "")
                if err != nil {
                    return err
                }
                defer driver.Put(img.Parent)
                changes, err := archive.ChangesDirs(layer, parent)
                if err != nil {
                    return err
                }
                size = archive.ChangesSize(layer, changes)
            }
        }
    }
```

可见当镜像内容layerData不为空时，Docker Daemon需要为镜像压缩包执行解压工作。以aufs这种

graphdriver为例，一旦aufs driver实现了graphdriver包中的接口Diff，则Docker Daemon会使用aufs driver的接口方法实现后续的解压操作。解压操作的源代码如下：

```
if differ, ok := driver.(graphdriver.Differ); ok {
        if err := differ.ApplyDiff(img.ID, layerData); err != nil {
            return err
        }

        if size, err = differ.DiffSize(img.ID); err != nil {
            return err
        }
    }
```

以上代码即实现了镜像压缩包的解压，与镜像所占空间大小的统计。代码differ.ApplyDiff(img.ID, layerData)将layerData解压至目标路径。理清目标路径，且看aufs这个driver中ApplyDiff的实现，位于./docker/docker/daemon/graphdriver/aufs/aufs.go#L304-L306，如下：

```
    func (a *Driver) ApplyDiff(id string, diff archive.ArchiveReader) error {
        return archive.Untar(diff, path.Join(a.rootPath(), "diff", id), nil)
}
```

解压过程中，Docker Daemon通过aufs driver的根目录/var/lib/docker/aufs、diff目录与镜像ID，拼接出镜像的解压路径，并执行解压任务。举例说明diff文件的作用，镜像27d474解压后的内容如图11-4所示：

```
root@vm:/var/lib/docker/aufs/diff/27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3# ll
total 100
drwxr-xr-x  21 root root  4096 Feb  1 16:11 ./
drwxr-xr-x 165 root root 20480 Mar 29 11:12 ../
drwxr-xr-x   2 root root  4096 Jan 29 00:28 bin/
drwxr-xr-x   2 root root  4096 Apr 11  2014 boot/
drwxr-xr-x   3 root root  4096 Jan 29 00:28 dev/
drwxr-xr-x  61 root root  4096 Jan 29 00:28 etc/
drwxr-xr-x   2 root root  4096 Apr 11  2014 home/
drwxr-xr-x  12 root root  4096 Jan 29 00:28 lib/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 lib64/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 media/
drwxr-xr-x   2 root root  4096 Apr 11  2014 mnt/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 opt/
drwxr-xr-x   2 root root  4096 Apr 11  2014 proc/
drwx------   2 root root  4096 Jan 29 00:28 root/
drwxr-xr-x   7 root root  4096 Jan 29 00:28 run/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 sbin/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 srv/
drwxr-xr-x   2 root root  4096 Mar 13  2014 sys/
drwxrwxrwt   2 root root  4096 Jan 29 00:29 tmp/
drwxr-xr-x  10 root root  4096 Jan 29 00:28 usr/
drwxr-xr-x  11 root root  4096 Jan 29 00:28 var/
root@vm:/var/lib/docker/aufs/diff/27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3#
```

图11-4镜像解压后示意图

回到StoreImage函数的执行流中，ApplyDiff任务完成之后，Docker Daemon通过DiffSize开启镜像磁盘空间统计任务。

5.2收集镜像大小并记录

Docker Daemon接管镜像存储之后，Docker镜像被解压到指定路径并非意味着“任务完成”。Docker Daemon还额外做了镜像所占空间大小统计的空间，以便记录镜像信息，最终将这类信息传递给Docker用户。

镜像所占磁盘空间大小的统计与记录，实现过程简单且有效，源代码位于./docker/docker/image/image.go#L122-L125，如下：

```
    img.Size = size
    if err := img.SaveSize(root); err != nil {
        return err
    }
```

首先Docker Daemon将镜像大小收集起来，更新Image类型实例img的Size属性，然后通过img.SaveSize(root)将镜像大小写入root目录，由于传入的root参数为临时目录_tmp，即写入临时目录_tmp下。深入SaveSize函数的实现，如以下源码：

```
func (img *Image) SaveSize(root string) error {
    if err := ioutil.WriteFile(path.Join(root, "layersize"), []
byte(strconv.Itoa(int(img.Size))), 0600); err != nil {
        return fmt.Errorf("Error storing image size in %s/layersize: %s", root, err)
    }
    return nil
}
```

SaveSize函数在root目录（临时目录/var/lib/docker/graph/_tmp）下创建文件layersize，并写入镜像大小的值img.Size。

5.3存储jsonData信息

Docker镜像中jsonData是一个非常重要的概念。在笔者看来，Docker的镜像并非只是Docker容器文件系统中的文件内容，同时还包括Docker容器运行的动态信息。这里的动态信息更多的是为了适配Dockerfile的标准。以Dockerfile中的ENV参数为例，ENV指定了Docker容器运行时，内部进程的环境变量。而这些只有容器运行时才存在的动态信息，并不会被记录在静态的镜像文件系统中，而是存储在以jsonData的形式先存储在宿主机的文件系统中，并与镜像文件系统做清楚的区分，存储在不同的位置。当Docker Daemon启动Docker容器时，Docker Daemon会准备好mount完毕的镜像文件系统环境；接着加载jsonData信息，并在运行Docker容器内部进程时，使用动态的jsonData内部信息为容器内部进程配置环境。

当Docker Daemon下载Docker镜像时，关于每一个镜像的jsonData信息均会被下载至宿主机。通过以上

jsonData的功能描述可以发现，这部分信息的存储同样扮演重要的角色。Docker Daemon如何存储jsonData信息，实现源码位于./docker/docker/image/image.go#L128-L139，如下：

```
    if jsonData != nil {
        if err := ioutil.WriteFile(jsonPath(root), jsonData, 0600); err != nil {
            return err
        }
    } else {
        if jsonData, err = json.Marshal(img); err != nil {
            return err
        }
        if err := ioutil.WriteFile(jsonPath(root), jsonData, 0600); err != nil {
            return err
        }
    }
```

可见Docker Daemon将jsonData写入了文件jsonPath(root)中，并为该文件设置的权限为0600。而jsonPath(root)的实现如下，即在root目录（/var/lib/docker/graph/_tmp目录）下创建文件json：

```
    func jsonPath(root string) string {
        return path.Join(root, "json")
}
```

镜像大小信息layersize信息统计完毕，jsonData信息也成功记录，两者的存储文件均位于/var/lib/docker/graph/_tmp下，文件名分别为layersize和json。使用临时文件夹来存储这部分信息并非偶然，11.6节将阐述其中的原因。

6.注册镜像ID

Docker Daemon执行完镜像的StoreImage操作，回到Register函数之后，执行镜像的commit操作，即完成镜像在graph中的注册。

注册镜像的代码实现位于./docker/docker/graph/graph.go#L212-L216，如下：

```
    // Commit
    if err := os.Rename(tmp, graph.ImageRoot(img.ID)); err != nil {
        return err
    }
    graph.idIndex.Add(img.ID)
```

11.5节StoreImage过程中使用到的临时文件_tmp在注册镜像环节有所体现。镜像的注册行为，第一步就是将tmp文件（/var/lib/docker/graph/_tmp ）重命名为graph.ImageRoot(img.ID)，实则为/var/lib/docker/graph/。使得Docker Daemon在而后的操作中可以通过img.ID在/var/lib/docker/graph目录下搜索到相应镜像的json文件与layersize文件。

成功为json文件与layersize文件配置完正确的路径之后，Docker Daemon执行的最后一个步骤为：添加

镜像ID至graph.idIndex。源代码实现是graph.idIndex.Add(img.ID)，graph中idIndex类型为*truncindex.TruncIndex， TruncIndex的定义位于./docker/docker/pkg/truncindex/truncindex.go#L22-L28，如下：

```
// TruncIndex allows the retrieval of string identifiers by any of their unique prefixes.
// This is used to retrieve image and container IDs by more convenient shorthand prefixes.
type TruncIndex struct {
	sync.RWMutex
	trie *patricia.Trie
	ids  map[string]struct{}
}
```

Docker用户使用Docker镜像时，一般可以通过指定镜像ID来定位镜像，如Docker官方的mongo:2.6.1镜像id为c35c0961174d51035d6e374ed9815398b779296b5f0ffceb7613c8199383f4b1，该ID长度为64。当Docker用户指定运行这个mongo镜像Repository中tag为2.6.1的镜像时，完全可以通过64为的镜像ID来指定，如下：

```
docker run –it c35c0961174d51035d6e374ed9815398b779296b5f0ffceb7613c8199383f4b1 /bin/bash
```

然而，记录如此长的镜像ID，对于Docker用户来说稍显不切实际，而TruncIndex的概念则大大帮助Docker用户可以通过简短的ID定位到指定的镜像，使得Docker镜像的使用变得尤为方便。原理是：Docker用户指定镜像ID的前缀，只要前缀满足在全局所有的镜像ID中唯一，则Docker Daemon可以通过TruncIndex定位到唯一的镜像ID。而graph.idIndex.Add(img.ID)正式完成将img.ID添加保存至TruncIndex中。

为了达到上一条命令的效果，Docker 用户完全可以使用TruncIndex的方式，当然前提是c35这个字符串作为前缀全局唯一，命令如下：

```
docker run –it c35 /bin/bash
```

至此，Docker镜像存储的整个流程已经完成。概括而言，主要包含了验证镜像、存储镜像、注册镜像三个步骤。

7.总结

Docker镜像的存储，使得Docker Hub上的镜像能够传播于世界各地变为现实。Docker镜像在Docker Registry中的存储方式与本地化的存储方式并非一致。Docker Daemon必须针对自身的graphdriver类型，选择适配的存储方式，实施镜像的存储。本章的分析，也在不断强调一个事实，即Docker镜像并非仅仅包含文件系统中的静态文件，除此之外还包含了镜像的json信息，json信息中有Docker容器的配置信息，如暴露端口，环境变量等。

可以说Docker容器的运行强依赖于Docker镜像，Docker镜像的由来就变得尤为重要。Docker镜像的下

载，Docker镜像的commit以及docker build新的镜像，都无法跳出镜像存储的范畴。Docker镜像的存储知识，也会有助于Docker其他概念的理解，如docker commit、docker build等。

8.作者介绍

孙宏亮，DaoCloud初创团队成员，软件工程师，浙江大学VLIS实验室应届研究生。读研期间活跃在PaaS和Docker开源社区，对Cloud Foundry有深入研究和丰富实践，擅长底层平台代码分析，对分布式平台的架构有一定经验，撰写了大量有深度的技术博客。2014年末以合伙人身份加入DaoCloud团队，致力于传播以Docker为主的容器的技术，推动互联网应用的容器化步伐。邮箱：allen.sun@daocloud.io

参考文献

http://aufs.sourceforge.net/aufs.html