Docker容器架构概述

文档说明：

- 架构示例的Docker版本：***Docker 1.13.1***
- Docker架构从1.11版本开始全面调整模块架构，在高版本中实现多个Docker组件间的解耦，

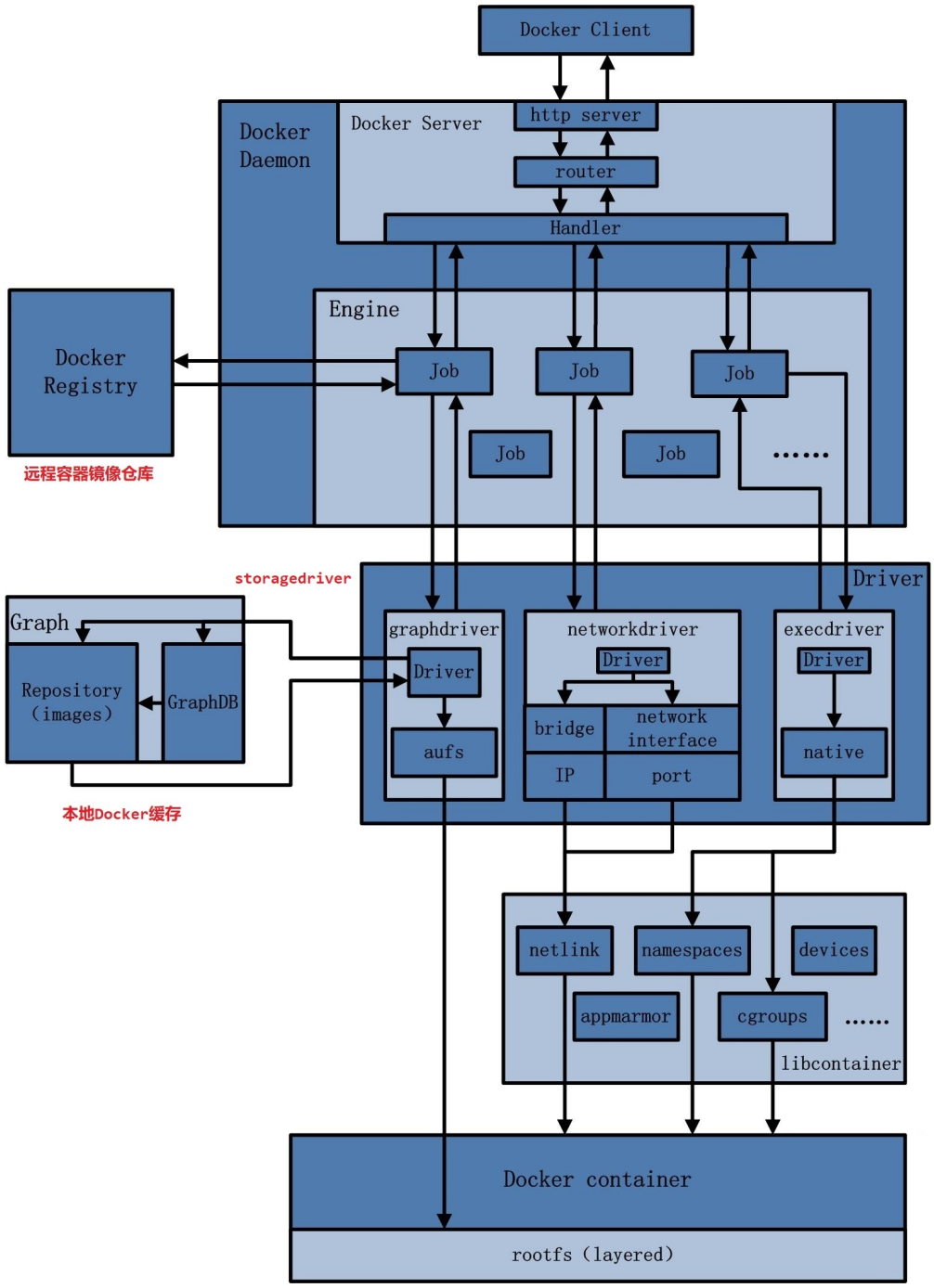
  解除Docker的绑定！

Docker架构概览：

- Docker对使用者来说是一个C/S模式的架构，S端采用**松耦合架构**，各模块有机组合并支撑

  Docker运行。

- Docker架构示意：



1. 用户使用Docker Client与Docker Daemon建立通信，并发送请求给后者。

2. 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。

3. 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。

4. ***libcontainer*** 是一项独立的容器管理包，networkdriver以及execdriver都是通过它来

实现具体对容器进行的操作。

5. 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。

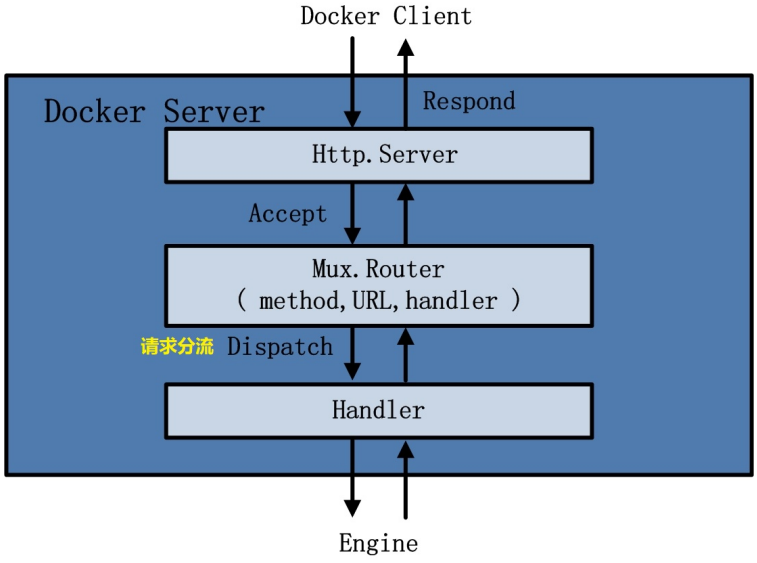
Docker功能模块：

- **Docker Client：**

1. Docker Client可以通过以下三种方式和Docker Daemon建立通信：
   a. tcp://<*host_ip*>:<*port*>
   b. unix://<*path_to_socket*>
   c. fd://<*socketfd*>
2. Docker Client可以通过设置命令行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。
3. Docker Client发送容器管理请求后，由Docker Daemon接受并处理请求，当Docker Client
接收到返回的请求响应并简单处理后，Docker Client一次完整的生命周期就结束了。
4. 当需要继续发送容器管理请求时，用户必须再次通过docker命令创建Docker Client。

- **Docker Daemon：**

1. 接受并处理Docker Client发送的请求。
2. 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。
3. **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。**
4. Docker Daemon的大致可以分为三部分：
   a. **Docker Server：**
      1）专门服务于Docker Client的Server，接受并调度分发Docker Client发送的请求。
      2）通过包 ***gorilla/mux***，创建了一个 ***mux.Router***，提供请求的路由功能。
      3）在Golang中，gorilla/mux 是一个强大的URL路由器以及调度分发器。
      4）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。



   b. **Docker Engine：**
      Engine是Docker架构中的运行引擎，同时也是Docker运行的核心模块。
   c. **Job：**

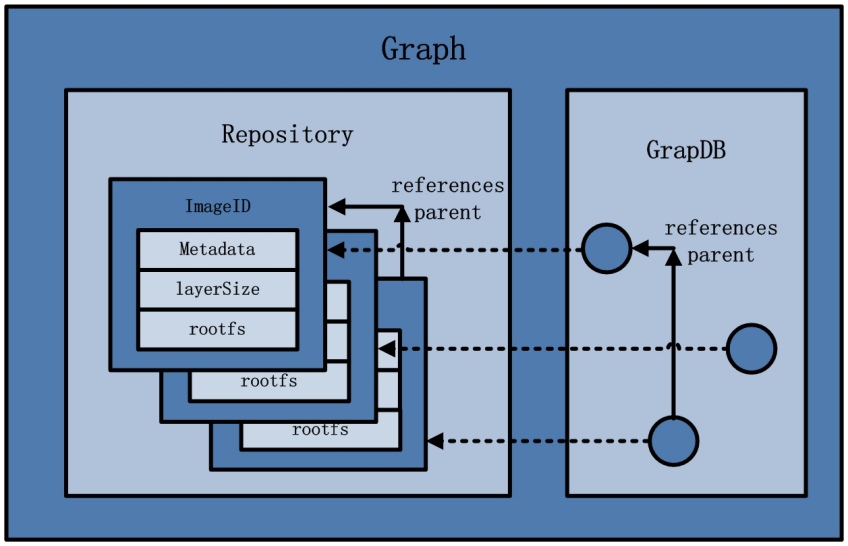
1）一个Job可以认为是Docker架构中Engine内部最基本的工作执行单元。
2）**Docker可以做的每一项工作，都可以抽象为一个Job。**

- **Docker Registry：**

1．registry是存储容器镜像的仓库。
2．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。
3．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。
4．**这三个功能对应的Job名称分别为：search、pull、push**

- **Graph：**

1．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。
2．一方面，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 ***GrapDB*** 记录
着所有文件系统镜像彼此之间的关系。



a．**GrapDB：**
1）构建在SQLite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。
2）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。
b．**Repository：**
关于每一个的容器镜像，具体存储的信息包括：
该容器镜像的元数据、容器镜像的大小信息、该容器镜像所代表的具体rootfs

- **Driver：**

1．通过Driver驱动，Docker可以实现对Docker容器执行环境的定制。
2．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。
3．**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。
4．在Docker Driver的实现中，可以分为以下三类驱动：

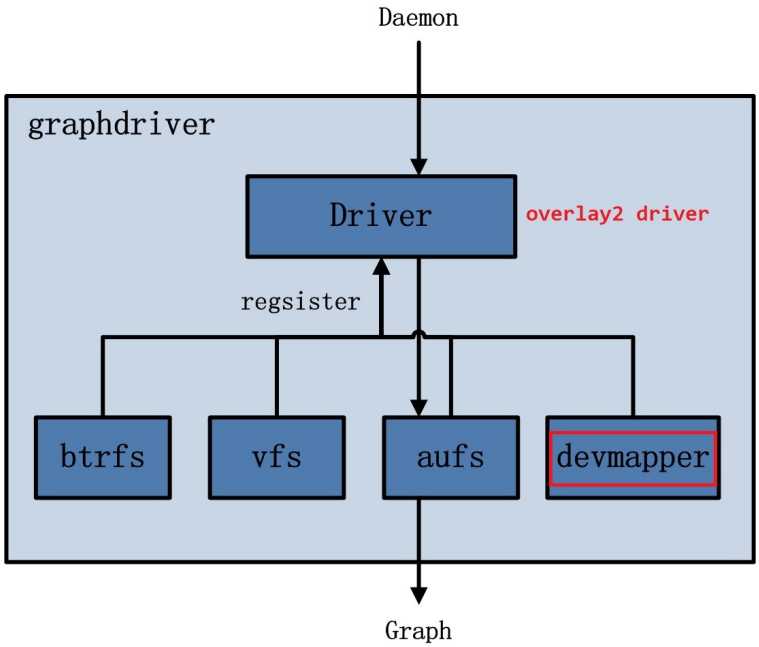
a. **graphdriver**：***rootfs***

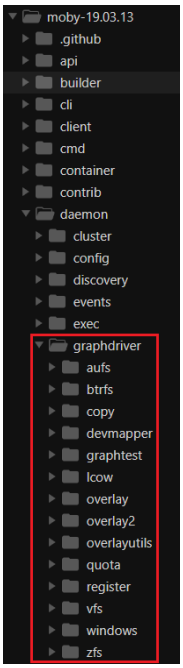
1）graphdriver主要用于完成容器镜像的管理，包括存储与获取。

2）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。





3）在graphdriver的初始化过程之前，有5种文件系统或类文件系统在其内部注册。

4）分别是 aufs、btrfs、vfs、**devmapper**、**overlay2**。

5）Docker在初始化时，通过获取系统环境变量 ***DOCKER_DRIVER*** 来提取所使用driver的

指定类型，之后所有的graph操作，都使用该driver来执行。

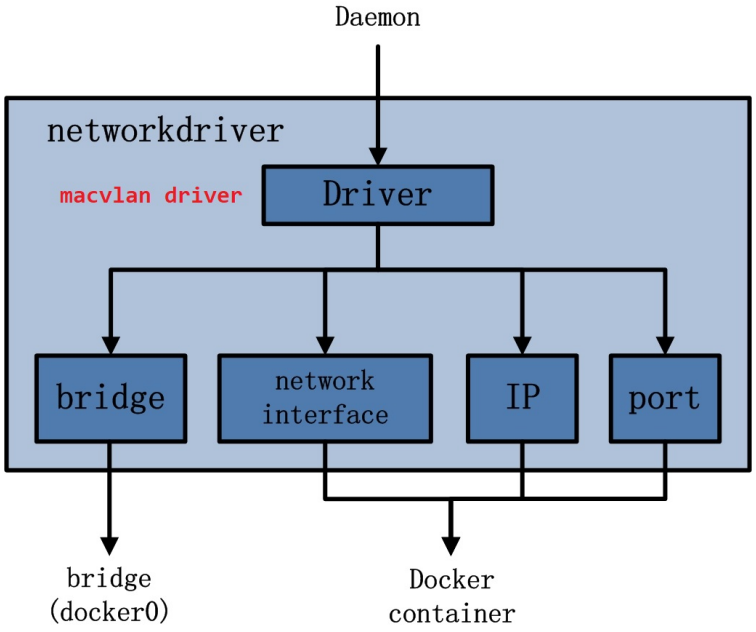
b. **networkdriver**：

1）networkdriver的用途是完成Docker容器网络环境的配置。

2）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。

* 注意：
高版本的networkdriver被containerd的CNI容器网络接口所取代，支持不同的
网络模式。

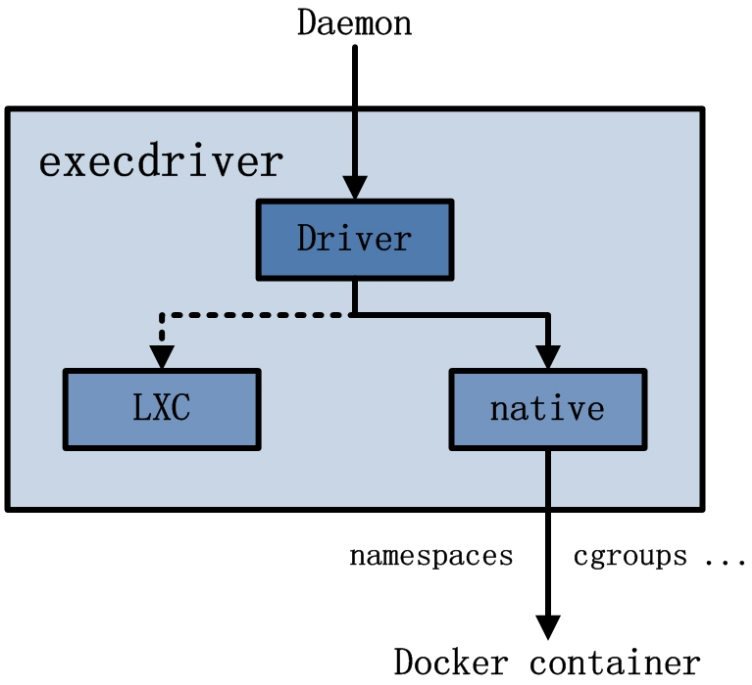


c. **execdriver**：*namespace*与*cgroup*

1）**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。

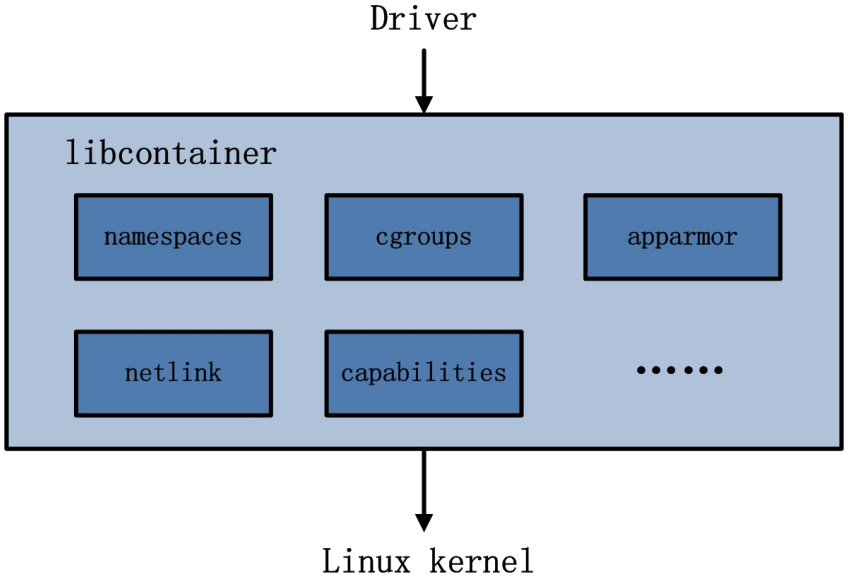
2）在execdriver的实现过程中，原先可以使用LXC驱动调用LXC的接口，来操纵容器
的配置以及生命周期，而现在execdriver默认使用 ***native*** 驱动，不依赖于LXC。

3）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。



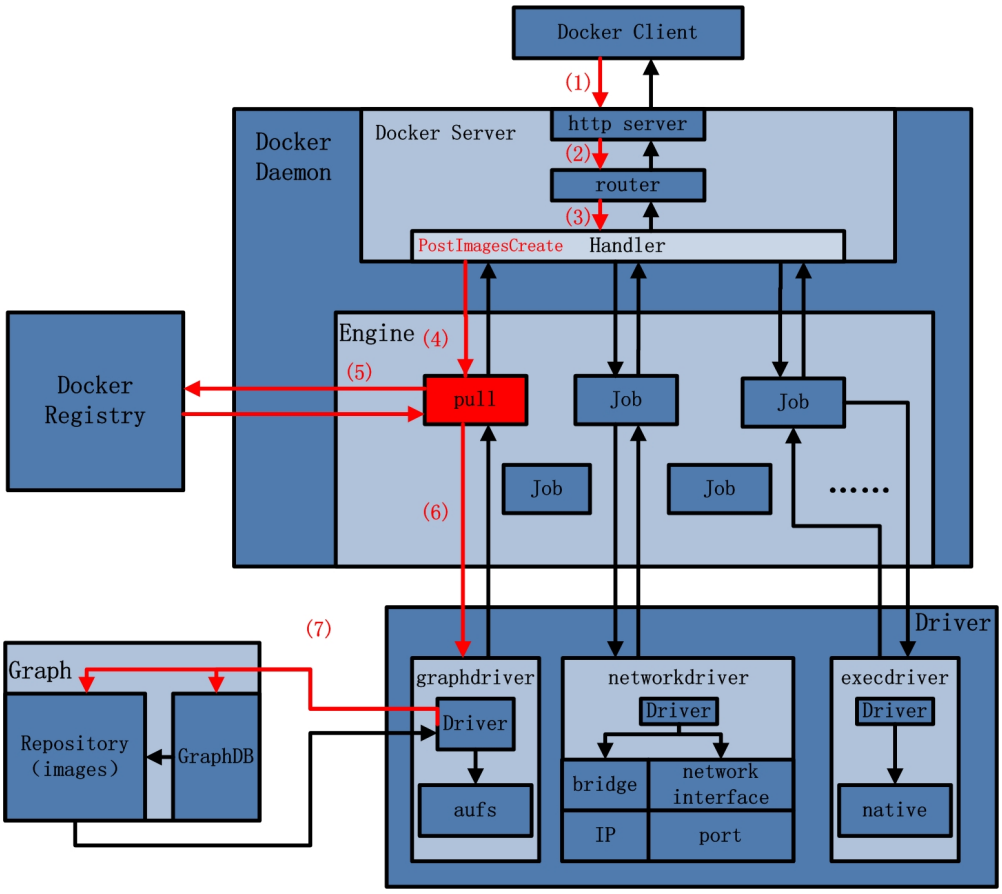
- **libcontainer**：

1. 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。
2. 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、**cgroup**、**apparmor**、网络设备以及防火墙规则等。
3. 这一系列操作的完成都不需要依赖LXC或者其他包。

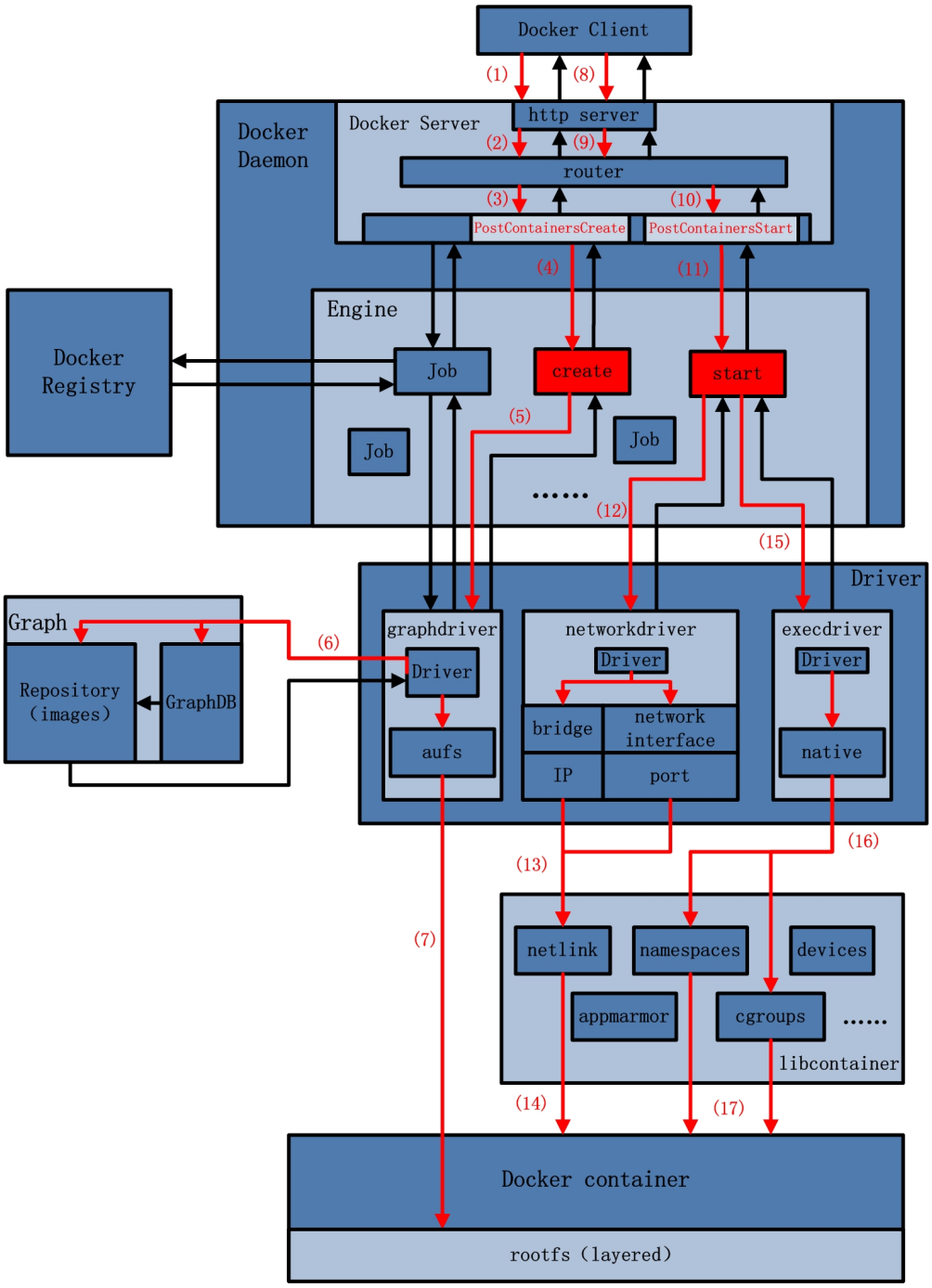


Docker工作流程示例：

- docker pull命令拉取容器镜像：



- docker run命令运行容器：


Docker Client
(1)
(8)
Docker Daemon
Docker Server
http server
(2)
(9)
router
(3)
(10)
PostContainersCreate
PostContainersStart
(4)
(11)
Engine
Docker Registry
Job
create
start
(5)
Job
Job
.....
(12)
(15)
Driver
Graph
(6)
graphdriver
networkdriver
execdriver
Repository (images)
GraphDB
Driver
Driver
Driver
aufs
bridge
network interface
IP
port
native
(16)
(13)
(7)
netlink
namespaces
devices
appmarmor
cgroups
.....
libcontainer
(14)
(17)
Docker container
rootfs (layered)