ocker源码分析（四）：Docker Daemon之NewDaemon实现

08月04日 22:12:51 孙宏亮 阅读数：1240 标签： Docker 更多

【摘要】

架构中Docker Daemon支撑着整个后台的运行，同时也统一化管理着Docker架构中graph、graphdriver、execdriver、volumes、Docker container等众多资源。可以说，Docker aemon对象来调度，而newDaemon的实现恰巧可以帮助大家了解这一切的来龙去脉。

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。然而，对于广大开发者而言，使用Docker这项技术已然不来的技术福利已不是困难。如今，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，们的思考与实践。

《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。

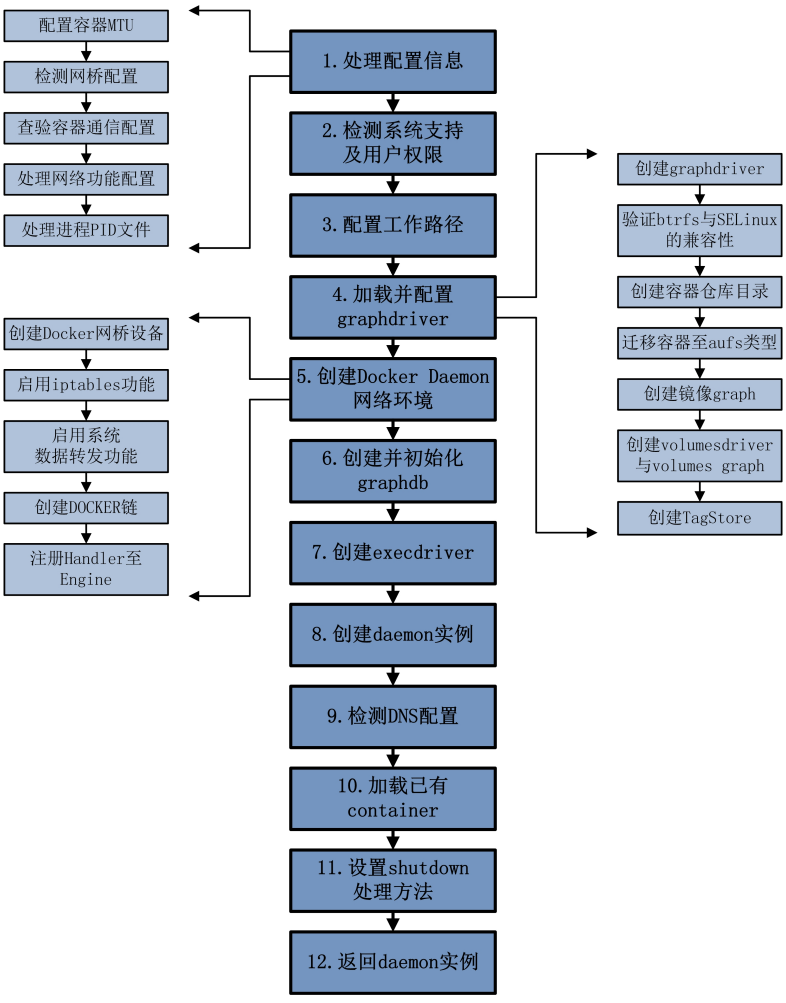
Daemon作用简介

er架构中有很多重要的概念，如：graph，graphdriver，execdriver，networkdriver，volumes，Docker containers等。Docker的实现过程中，需要将以_而Docker Daemon中的daemon实例就是设计来完成这一任务。

的角度，NewDaemon函数的执行出色的完成了Docker Daemon创建并加载daemon的任务，最终实现统一管理Docker Daemon的资源。

Daemon源码分析内容安排

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：



Daemon具体实现

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

代码分析如下：

数名：NewDaemon；

数调用具体实现所处的包位置：**./docker/daemon**；

数具体实现源文件：**./docker/daemon/daemon.go**；

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；

数返回类型：d，具体的Daemon对象实例；err，错误状态。

ocker/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
}
```

在实现NewDaemon的过程中，主要依靠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New
篇幅分析其实现细节。

用配置信息

DaemonFromDirectory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
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的运行更贴近用户所期待的运行场景。

息的处理包含4部分：

置Docker容器的MTU；

则网桥配置信息；

险容器通信配置；

里PID文件配置。

一分析配置信息的处理。

配置Docker容器的MTU

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()
    }
```

nfig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/da**
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/confi**

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

faultNetworkMtu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；MTU值defaultNetworkMtu，值为1500。

检测网桥配置信息

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

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

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Bri互斥类型，只能至多指定其中之一。在**默认配置文件**中，BridgeIface和BridgeIP均为空。

查验容器通信配置

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Communication的作用是启动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 --iptab
    }
```

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。

处理网络功能配置

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
```

nfig中的DisableNetwork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。后续的”init_networ

处理PID文件配置

D文件配置，主要工作为：为Docker Daemon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shutdo 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)
    })
    }
```

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**始值为” /var/run/docker.pid”。

是关于配置信息处理的分析。

测系统支持及用户权限

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
处理器的支持。

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())
}
```

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
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”。

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，可就是说若返回不为0，则说明不是以
出Fatal日志。

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，
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
需的内核版本若过低，则必须升级至3.8.0。

置工作路径

ocker 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
er”。

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
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

载并配置存储驱动graphdriver

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

Graphdriver的创建

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)
```

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GraphOpti
aphDriver中的new函数实现加载graph的存储驱动。

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

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
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)
    }
}
```

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
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
}
```

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
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，
r来选择具体的graphdriver。

验证btrfs与SELinux的兼容性

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Fat

```
// 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!")
}
```

创建容器仓库目录

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
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
}
```

迁移容器至aufs类型

hdriver的类型为aufs时，需要将现有的graph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
}
```

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
}
```

e 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。

e 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。

e 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containe

创建镜像graph

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。

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

aph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
}
```

ot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
前缀检索镜像与容器的ID；最后driver表示具体的graphdriver类型。

创建volumesdriver以及volume graph

er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
volumes时，Docker需要使用文件系统driver来管理它，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

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

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。

创建TagStore

re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)
```

gStore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{}
}
```

述的是TagStore类型中的多个属性的含义：

h：TagStore中记录镜像的仓库文件位置；

aph：相应的Graph实例对象；

positories：记录具体的镜像的仓库的数据结构；

ıc.Mutex：TagStore的互斥锁

lingPool：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；

shingPool：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；

建Docker Daemon网络环境

ocker 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，同样也关系这容器间的通信。

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
    }
}
```

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
work属性赋值为false，故判断语句结果为真，执行相应的代码块。

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：

境变量EnableIptables，使用config.EnableIptables来赋值，为true；

境变量InterContainerCommunication，使用config.InterContainerCommunication来赋值，为true；

境变量EnableIpForward，使用config.EnableIpForward来赋值，值为true；

境变量BridgeIface，使用config.BridgeIface来赋值，为空字符串""；

境变量BridgeIP，使用config.BridgeIP来赋值，为空字符串""；

境变量DefaultBindingIP，使用config.DefaultIp.String()来赋值，为"0.0.0.0"。

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，具体的实现位于**./docker/daemon/net**
r.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对象注册了4个Handler,如
release_interface"，"allocate_port"，"link"。

创建Docker网络设备

ocker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。

river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，则继续使
建docker0网桥。具体实现为**createBridge(bridgeIP)**，以及**createBridgeIface(bridgeIface)**。

Bridge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
的网桥设备，并设置MAC地址，通过libcontainer中netlink包的CreateBridge来实现。

启用iptables功能并配置

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
cker/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)
    }
}
```

tupIPtables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s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
```

上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，如果需要继续法网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
```

P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
```

F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
```

启用系统数据包转发功能

x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s/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)
    }
}
```

创建DOCKER链

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/bridg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)
}
```

注册Handler至Engine

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_interainer分配一个专属网卡； * realease_interface：释放网卡资源； * allocate_port：为Docker container分配一个端口； * link：实现Docker container间的link

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。

建graphdb并初始化

db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graph

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

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h.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**hdb/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)
}
```

建execdriver

iver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)
```

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：

nfig.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 cont

nfig.Root:Docker运行时的root路径，默认配置文件中为"/var/lib/docker";

sInitPath:系统上存放dockerinit文件的路径，一般为"/var/lib/docker/init/dockerinit-1.2.0";

sInfo: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。

execdrivers.NewDriver之前，首先通过以下代码，获取期望目标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)
```

行以上代码，localCopy为"/var/lib/docker/init/dockerinit-1.2.0"，而sysInitPath为当前Docker运行时中dockerinit-1.2.0实际所处的路径，utils.DockerInitPat**tils/util.go**。若localCopy与sysInitPath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

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
}
```

emoryLimit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通过宿主机是否存在/sys/kernel/security/app则置为true，否则置为false。

ecdrivers.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
执行代码，返回最终的execdriver，如以下，其中native.NewDriver实现位于**./docker/daemon/execdriver/native/driver.go**：

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

建daemon对象

· 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,
}
```

析Daemon类型的属性：

测DNS配置

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**./doc**
o，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",
      daemon.config.Dns = DefaultDns
    }
    return nil
}
```

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
8.8.8.8，8.8.4.4，并将其赋值给config文件中的Dns属性。

动时加载已有Docker containers

er 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 Daemo
器，将容器信息收集，并做相应的维护。

置shutdown的处理方法

已有Docker container之后，Docker Daemon设置了多项在shutdown操作中需要执行的handler。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())
        }
    })
```

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
； * 通过portallocator.ReleaseAll()，释放所有之前占用的资源； * 通过daemon.driver.Cleanup()，通过graphdriver实现unmount所有layers中的挂载点；
Graph.Close()关闭graphdb的连接。

返回daemon对象

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。

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

ker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NewI
on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。

简介

，**DaoCloud**初创团队成员，软件工程师，浙江大学计算机科学专业应届毕业研究生。

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

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。

流，邮箱：**allen.sun@daocloud.io**

文献

o Programming Language-Packages]
s matches]
sys/net/ipv4/* Variables:]
st packages of docker]

注Docker源码分析公众号