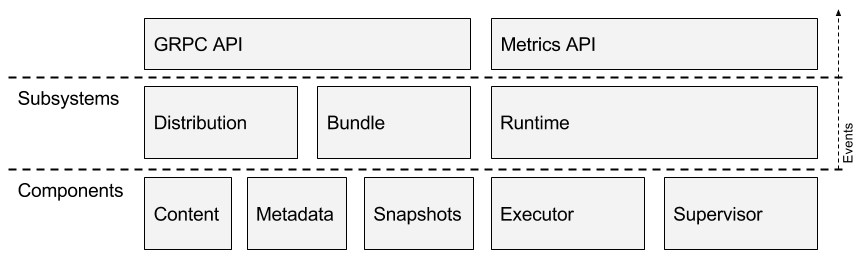
# containerd

## 简介

containerd并不是直接面向最终用户的，而是主要用于集成到更上层的系统里，比如Swarm,Kubernetes,Mesos等容器编排系统。containerd以daemon形式运行在系统上，通过unix domain socket暴露底层的gRPC API,上层系统可以通过这些API管理机器上的容器。

## 架构



Distribution:和Docker Registry打交道，拉取镜像

Bundle:管理本地磁盘上镜像的子系统

Runtime:创建容器，管理容器的子系统

containerd gRPC:

containers

content

diff

events

images

namesapces

snapshot

tasks

version

# 容器创建

## 必要参数检查

## 查找创建容器的image

## 创建容器的读写层:

layer/layer\_store.go:CreateRWLayer(containerID, parent chainID, opts \*CreateRWLayerOpts)

containerID:容器ID

parent:这里就是镜像的img.RootFS.ChainID

参数选项:包括mount参数，mount初始化函数，存储配置。

### mount初始化:

* 调用driver创建init层和可写层:

graphdriver/overlay2/overlay.go:CreateReadWrite(id,parent,opts)

id: <**cacheID-init**> 对于linux系统cacheID为一个随机ID，对于windows系统就是容器ID.

1. 创建init层
2. 获取设置的UID,GID MAP，以映射的UID,GID创建init目录,目录权限为0700(不同的graphdriver权限不同)/var/lib/docker/overlayer2/<cacheID-init>
3. 如果配置了限额，则设置限额
4. 创建diff, work,megerd目录和link,,lower文件，在../l中创建符号链接指向diff目录，并将符号链接名称写入link文件。将parent cacheID写入lower文件.
5. 创建可写层
6. 同A,B,C再创建容器可写层<**cacheID**>

* 将所有layers和init层mount到cacheID目录下的merged:

graphdriver/overlay2/overlay.go:Get(id,mountLable)

* 调用初始化函数初始化init层

daemon/initlayer/setup\_unix.go:Setup(initLayer,rootUID,rootGid)

创建如下目录，文件和符号链接

**"/dev/pts"**: **"dir"**,  
**"/dev/shm"**: **"dir"**,  
**"/proc"**: **"dir"**,  
**"/sys"**: **"dir"**,  
**"/.dockerenv"**: **"file"**,  
**"/etc/resolv.conf"**: **"file"**,  
**"/etc/hosts"**: **"file"**,  
**"/etc/hostname"**: **"file"**,  
**"/dev/console"**: **"file"**,  
**"/etc/mtab"**: **"/proc/mounts"**

* 将所有layers再umount

## 设置主机配置

### volume配置和tmpfs配置

volume的优先级为:之前配置--🡪volumes-from-🡪volume

* mount类型:

**const** (  
 *// TypeBind is the type for mounting host dir* ***TypeBind*** Type = **"bind"** *// TypeVolume is the type for remote storage volumes* ***TypeVolume*** Type = **"volume"** *// TypeTmpfs is the type for mounting tmpfs* ***TypeTmpfs*** Type = **"tmpfs"** *// TypeNamedPipe is the type for mounting Windows named pipes* ***TypeNamedPipe*** Type = **"npipe"**)

* 读写方式

*// read-write modes***var** rwModes = **map**[string]bool{  
 **"rw"**: ***true***,  
 **"ro"**: ***true***,  
}

默认为ro

* label模式

*// label modes***var** linuxLabelModes = **map**[string]bool{  
 **"Z"**: ***true***,  
 **"z"**: ***true***,  
}

* 挂载关系

从linux2.6.15开始,mount加入了shared,private,slave,unbindable ４种挂载关系。

shared:互相传播事件

slave:只能从master接收事件

private:没有相互传播能力

unbindable:不能够通过bind操作clone

*// PropagationRPrivate RPRIVATE****PropagationRPrivate*** Propagation = **"rprivate"***// PropagationPrivate PRIVATE****PropagationPrivate*** Propagation = **"private"***// PropagationRShared RSHARED****PropagationRShared*** Propagation = **"rshared"***// PropagationShared SHARED****PropagationShared*** Propagation = **"shared"***// PropagationRSlave RSLAVE****PropagationRSlave*** Propagation = **"rslave"***// PropagationSlave SLAVE****PropagationSlave*** Propagation = **"slave"**

默认使用rprivate.

* 一致性

**const** (  
 *// ConsistencyFull guarantees bind mount-like consistency* ***ConsistencyFull*** Consistency = **"consistent"** *// ConsistencyCached mounts can cache read data and FS structure* ***ConsistencyCached*** Consistency = **"cached"** *// ConsistencyDelegated mounts can cache read and written data and structure* ***ConsistencyDelegated*** Consistency = **"delegated"** *// ConsistencyDefault provides "consistent" behavior unless overridden* ***ConsistencyDefault*** Consistency = **"default"**)

### 设置links

1. 在新容器的/etc/hosts中添加link的容器DNS信息
2. 保存配置到hostconfig.json

/var/lib/docker/<container-id>/hostconfig.json

### 将容器配置保存

container.ToDisk

/var/lib/docker/<container-id>/config.v2.json

### 创建平台特定的信息

### 设置网络模式并更新网络配置

daemon.SetDefaultNetModeIfBlan

deamon.updateContainerNetworkSettings

### 再次保存容器配置

container.ToDisk

# 容器启动

* checkpoint模式只能在实验版中指定
* 调用graphdriver mount容器的FS:

daemon/graphdriver/overlay2/overlay.go:Get(id,mountLabel)

* 初始化网络

1. 如果是container方式,则设置相同的Hostname,DomainName,hosts,resolvconf
2. 如果是host方式，则设置hostname为主机
3. 分配网络
   1. 删除原同名namespace(containerID)
   2. 依次连接配置的network(CNM系统)
   3. 写入配置文件hostconfig.json
4. 构造容器hostname文件

* OCI createSpec

1. populateCommSpec:设置link,workingdir,init,
2. 判断daemon是否设置了”exec-opt:native.cgroupdriver”为systemd
3. 设置资源blkio,memory,cpu,oom
4. 设置devices, setDevices(specs.Spec,\*container.Container). --privileged设置主机所有devices rwm权限，否则按--device,--device-cgroup-rule设置对应的cgroup
5. 设置Rlimits
6. 设置UID,GID
7. 设置namespaces:user,network,ipc,pid,uts
8. 设置capabilities,如果--privileged设置所有
9. 设置seccomp,如果指定--privileged则不设置

设置shm目录(/var/lib/docker/<container-ID>/shm),

syscall.Mount("shm",shmPath, "tmpfs", uintptr(syscall.MS\_NOEXEC|syscall.MS\_NOSUID|syscall.MS\_NODEV)

，并mount

1. 设置secretDir,swarm使用
2. 设置mounts.
3. tmpfs mount
4. apparmor设置

* 获取配置的runtime,如果使用systemd,添加”--systemd-cgroup-true”参数。
* 设置restart策略
* 保存Apparmor配置
* 调用containerd的create接口:

docker/docker/libcontainerd/client\_daemon.go:create

1. 准备bundle的目录

/var/run/docker/libcontainerd/containerd/<container-id>

1. 写入spec配置:config.json
2. 启动容器:

libcontainerd/client\_daemon.go:Start

* 创建FIFO,用于容器的终端输入输出.不有终端的容器才创建init-stderr.(libcontainerd/client\_daemon.go:createIO)

初始化容器终端docker/docker/container/container.go:InitializeStdio

* 创建NewTask(containerd/services/tasks/service.go:Create)

调用runtime(containerd/containerd/linux/runtime.go:Create)

调用shim(containerd/containerd/linux/shim/service.go:Create)

初始化新进程(containerd/containerd/linux/shim/init.go:newInitProcess)

1. mount rootfs
2. 调用runc创建(containerd/go-runc/runc.go:Create)

* t.start:启动用户指定进程(containerd/containerd/services/tasks/service.go:Start)

(containerd/go-runc/runc.go:runOrError)

容器网络

daemon/start.go:--🡪containerStart--🡪initializeNetworking---🡪allocateNetwork-🡪connectToNetwork--🡪CreateEndpoint

BuildCreateEndpointOptions

BuildJoinOptions