# 镜像来源

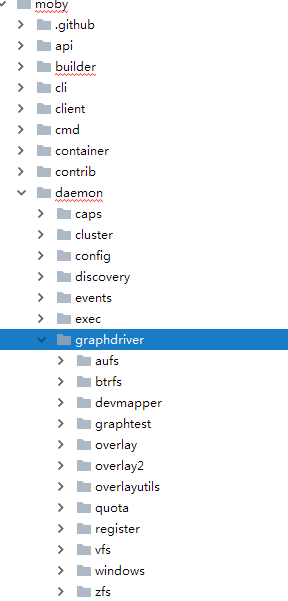
镜像需要存在于本地仓库中才能用其启动容器，镜像通常有以下三种来源:

* 使用dockerfile构建
* 导入从其它仓库save的镜像
* 从远端仓库pull镜像

其它还有对容器进行commit等，但它们的原理都包含在了以上3种方式之中。

无论采用哪种方式，镜像的最初来源一般都是通过dockerfile构建而来，因此首先分析dockerfile构建镜像的过程，进而帮助我们了解镜像是如何存储和使用的.

docker可以采用不同的存储驱动来存储和使用镜像，目前内置的驱动有:



我们采用的是overlay2,以此为例进行分析的讲解，基本原理大同小异。

# 目录结构

docker的根工作目录一般是/var/lib/docker

首先看一下涉及到的相关目录：

/var/lib/docker/image/<graphdriver>:存储镜像管理数据的目录，以使用的存储驱动命名

/distribution:pull的镜像相关元数据

/imagedb:镜像数据库

/content:构成镜像的每层次的配置数据

/sha256/<IMAGE-ID>:每镜像层次的配置digest,也就是镜像ID.(**参考源码:github.com/docker/docker/image/store.go:store.Create(config []byte)(ID, error)**)

/metadata:

/sha256/<IMAGE-ID>:具有父镜像的层次ID,没有父镜像的基础镜像在此目录没有内容

/parent:父镜像ID(**参考源码:github.com/docker/docker/daemon/build.go:Daemon.CreateImage(config []byte,parent string,platform string)(builder.Image, error)**)

/layerdb:镜像每layer元数据

/sha256

/<chainID>:每个layer的chainID

/cache-id:本layer在下面所对应的cache-id

/diff:本层次的diffID

/size:本层次的大小

/parent: 父layer chainID (**moby/daemon/commit.go:Daemon.Commit** **moby/layer/ro\_layer.go:storeLayer**)

/mounts:容器的RW　layer信息

/<container-id>:

/init-id:读写层的cache-id

/cache-id:容器的读写层mount-id

/parent:父layer的chainID

/var/lib/docker/<graphdriver>:镜像的所有layer和容器rwlayer的位置

/l:符号链接目录，每一个符号链接文件链接向下面的cache-id,一一对应，使用这个符号链接的目的是因为mount args最大限制为一个pagesize.

/cache-id: layer的cache-id为一个随机值.(**参考源码:moby/layer/layer\_unix.go:layerStore.mountID(name string)string**)

/diff:本layer所包含的实际文件系统数据

link:存储本cache-id所对应的符号链接

lower:本layer的所有父layer所对应的符号链接

/megerd:本layer及所有父layer共同呈现的目录

/work:overlay2文件系统使用的目录(**参考源码:moby/daemon/graphdriver/overlay2/overlay.go:Driver.CreateReadWrite(id, parent,opts)erro**r)

/cache-id-init:容器的可写层目录:(**moby/layer/layer\_store.go:layerStore.initMount(graphID, parent, mountLabel, initFunc,storageOpt)(string,error)**)

# 镜像构建



# 镜像导出

使用docker save命令导出镜像

镜像内容:

.

├── 816c0fa43179255d36592e0ede6ed020793130645eaf063fa27c5544ae46bb6b

│ ├── json

│ ├── layer.tar

│ └── VERSION

├── b8efb18f159bd948486f18bd8940b56fd2298b438229f5bd2bcf4cedcf037448.json

├── bcb8dea8dbd93bef252214259890b19a6a4886bc333d0b16f98a40b5fd063c27

│ ├── json

│ ├── layer.tar

│ └── VERSION

├── e1a9983e063a540bd4072c352ab6bc72b63ceebf311255e9d16de34eee018471

│ ├── json

│ ├── layer.tar

│ └── VERSION

└── manifest.json

b8efb18f159bd948486f18bd8940b56fd2298b438229f5bd2bcf4cedcf037448.json:

<IMAGE-ID>.json:镜像配置

manifest.json:清单文件

**type** manifestItem **struct** {  
 Config string  
 RepoTags []string  
 Layers []string  
 Parent image.ID **`json:",omitempty"`** LayerSources **map**[layer.DiffID]distribution.Descriptor **`json:",omitempty"`**}

816c0fa43179255d36592e0ede6ed020793130645eaf063fa27c5544ae46bb6b:

镜像每一层次的内容，这个ID是在导出时用原有层次信息生成的临时镜像所做的摘要。对应清单文件中的Layers.(**参考代码:docker/docker/image/tarexport/save.go**)

json: V1Image结构体

VERSION:版本信息,1.0

layer.tar:实际的文件系统内容

# 镜像导入

镜像导出的逆过程。

参考代码:(image/tarexport/load.go:Load)

1. 创建临时目录(/tmp/XRFMG/docker-import-)，对镜像压缩包进行解压.
2. 读取manifest.json文件,获取config文件名，文件名为<IMAGE-ID>.json
3. 从配置文件中获取所有层的diffIDS,遍历所有diffIDS,依次加载

如果layer的chainID已经存在，则不再导入.

如果layer的chainID不存在，则导入.

导入后判断导入layer的diffID与配置文件中是否相等，不相等则报错.