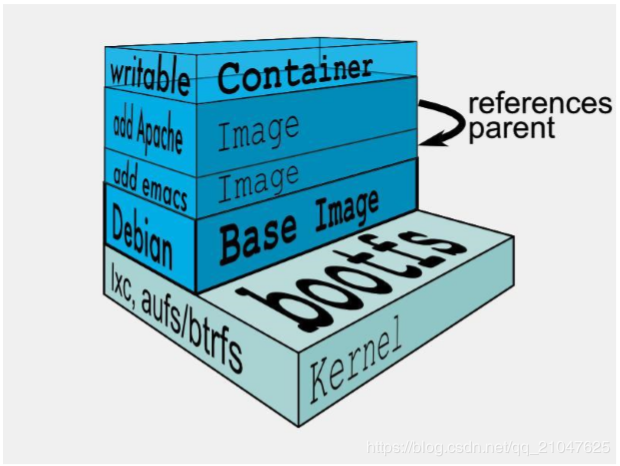
Docker 容器互联

1  基于Docker Volum的容器互联

1.1 Docker的文件存储

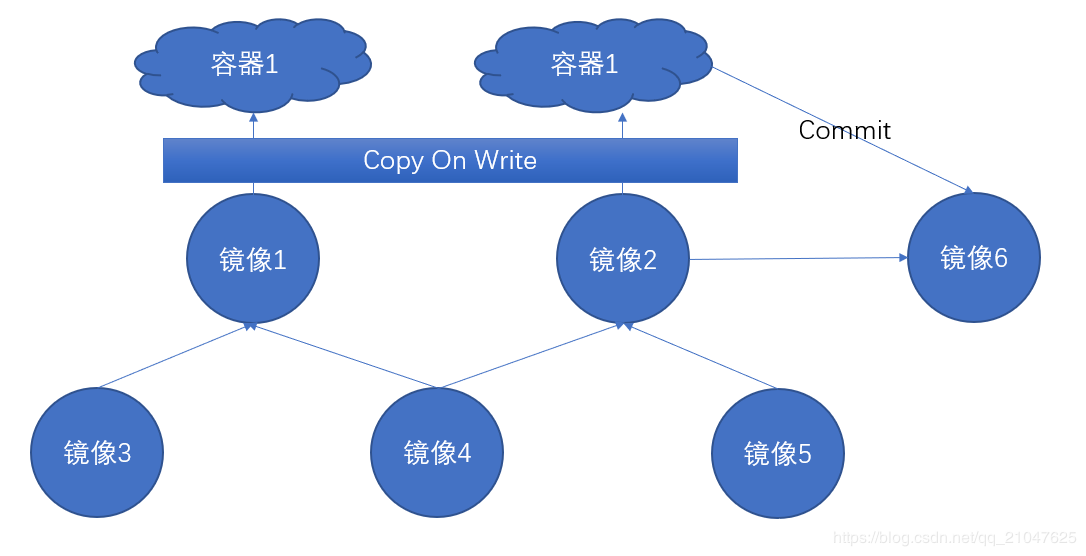
docker的文件系统是copy on write方式的, 文件是一层层往上叠加的, 最下的一层是只读的, 要修改的时候会复制这层只读的覆盖在只读的上面作为一层可写的, 这个时候原来的只读文件依然是存在的这是看不到了

docker 中镜像的存储就是使用的上述方式,分层结构如下



上图Docker的容器就是通过只读的镜像上面覆盖的可写层, 要想把这个可写层持久化, 就要用到之前的讲的 docker commit 了

但是镜像的层次又可以共享:

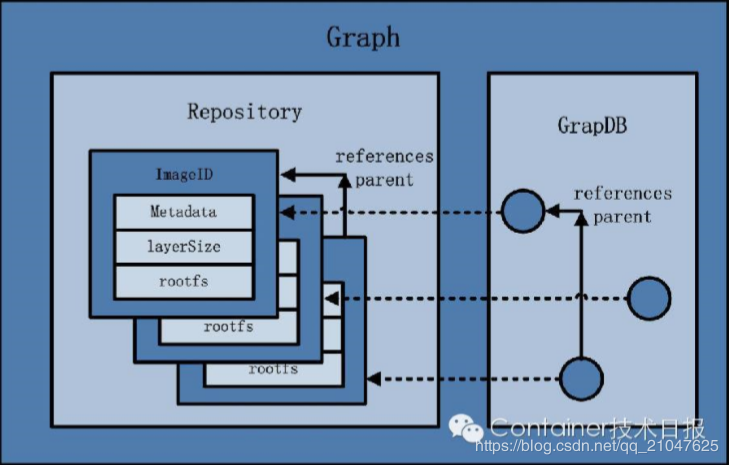


镜像是分层叠加的, 顶层的镜像依赖于底层的镜像, 容器是通过镜像 Copy On Write 一个可写层, 这个可写层 Commit 就完成了持久化又成了一个新的镜像.

Docker镜像的存储位置:

docker的镜像是分层的,一个镜像往往是多个镜像分层叠加出来的, 镜像的分层信息是存储在 Docker Graph 里面

这里记录了镜像有些那些层,   每一层的父层和大小等信息

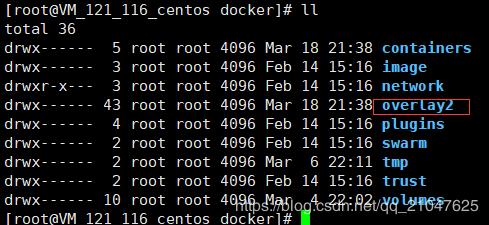


其中 GrapDB 存储的是分层结构 , Repository 里面存储是镜像的信息, 两者结合得到镜像的分层信息

但是这些里面存储都是镜像的信息, 并不是镜像的实际存储位置,

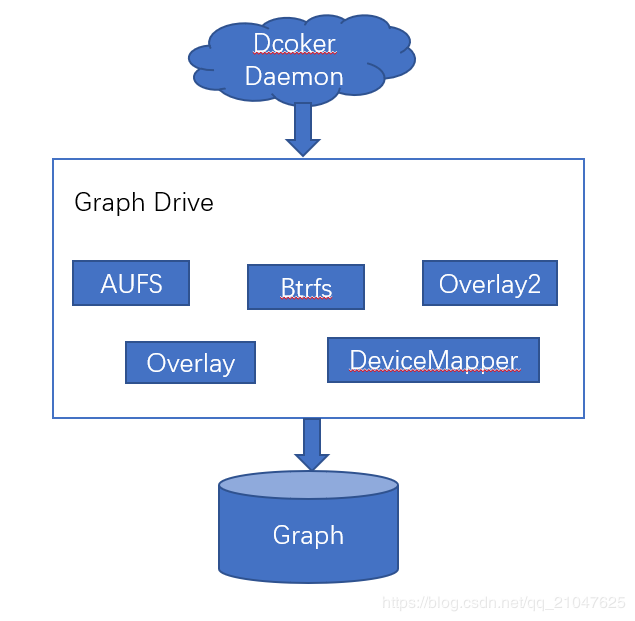
镜像的二进制文件存储位置是. /var/lib/docker/<storage-driver>

<storage-driver> 是  Docker Graph Driver , 不同的 Docker Graph Driver 会存储在不同的目录, 比如本机采用的是 overlay2



Docker Graph Driver 是Docker 使用来管理存储镜像每层内容及可读写的容器的驱动, 目前主要有 DeviceMapper、AUFS、Overlay、Overlay2、Btrfs、ZFS 等，不同的存储驱动实现方式有差异，镜像组织形式可能也稍有不同，但都采用栈式存储，并采用 Copy-on-Write策略。且存储驱动采用热插拔架构，可动态调整。

Graph Driver 就类似于Java连接数据库的 JDBC, 只不过这个驱动是用于 docker Deamon 来对镜像个容器进行读写.



选择策略:

若内核支持多种存储驱动，且没有显式配置，Docker 会根据它内部设置的优先级来选择。优先级为 AUFS > Btrfs/ZFS > Overlay2 > Overlay > DeviceMapper。若使用 DeviceMapper 的话，在生产环境，一定要选择 direct-lvm, loopback-lvm 性能非常差。

选择会受限于 Docker 版本、操作系统、系统版本等。例如，AUFS 只能用于 Ubuntu 或 Debian 系统，Btrfs 只能用于 SLES （SUSE Linux Enterprise Server, 仅 Docker EE 支持）。

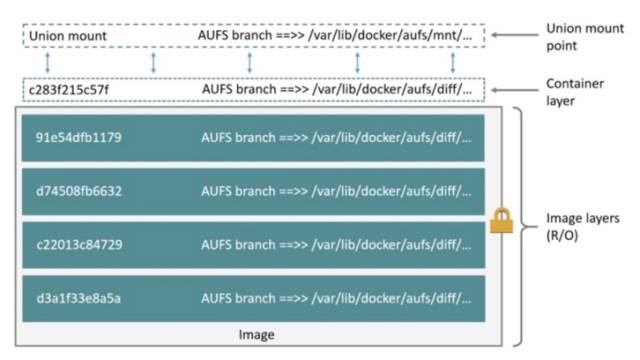
有些存储驱动依赖于后端的文件系统。例如，Btrfs 只能运行于后端文件系统 Btrfs 上。

不同的存储驱动在不同的应用场景下性能不同。例如，AUFS、Overlay、Overlay2 操作在文件级别，内存使用相对更高效，但大文件读写时，容器层会变得很大；DeviceMapper、Btrfs、ZFS 操作在块级别，适合工作在写负载高的场景；容器层数多，且写小文件频繁时，Overlay 效率比 Overlay2 更高；Btrfs、ZFS 更耗内存。

## 存储驱动

****AUFS****

AUFS（AnotherUnionFS）是一种Union FS，是文件级的存储驱动。AUFS能透明覆盖一或多个现有文件系统的层状文件系统，把多层合并成文件系统的单层表示。简单来说就是支持将不同目录挂载到同一个虚拟文件系统下的文件系统。这种文件系统可以一层一层地叠加修改文件。无论底下有多少层都是只读的，只有最上层的文件系统是可写的。当需要修改一个文件时，AUFS创建该文件的一个副本，使用CoW将文件从只读层复制到可写层进行修改，结果也保存在可写层。在Docker中，底下的只读层就是image，可写层就是



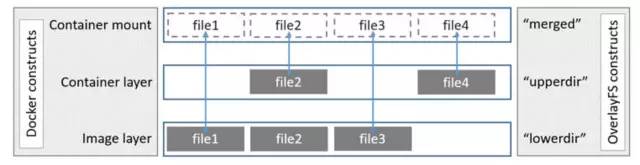
### Overlay2

容器镜像就是容器运行时所需要的二进制文件与依赖包的集合，存储在只读的Dockerfile中。镜像基于分层的联合文件系统UnionFS来实现，Ubuntu里基于AUFS实现，新推出OverlayFS比AUFS性能更好，部署更简单，两者核心原理是类似的。图5为容器镜像基于OverlayFS（推荐更高效稳定的Overlay2）的分层架构，其文件处理如表2。

1) lowerdir：镜像层，保存只读的容器镜像文件，多个容器共享；

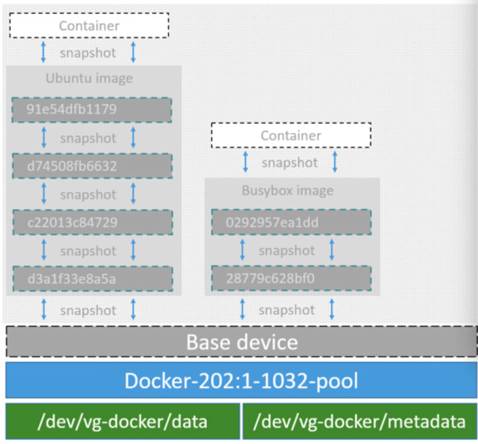
2) upperdir：容器层，可读可写，采用写时复制COW，每个容器独有。

3) merged：统一视图层，整合各层内容，挂载的容器rootfs下，统一呈现。



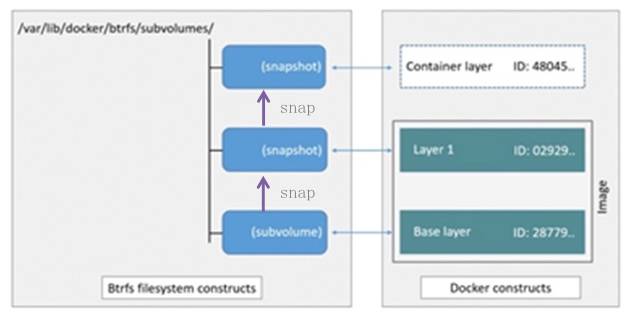
#### Device mapper

Device mapper是Linux内核2.6.9后支持的，提供的一种从逻辑设备到物理设备的映射框架机制，在该机制下，用户可以很方便的根据自己的需要制定实现存储资源的管理策略。前面讲的AUFS和OverlayFS都是文件级存储，而Device mapper是块级存储，所有的操作都是直接对块进行操作，而不是文件。Device mapper驱动会先在块设备上创建一个资源池，然后在资源池上创建一个带有文件系统的基本设备，所有镜像都是这个基本设备的快照，而容器则是镜像的快照。所以在容器里看到文件系统是资源池上基本设备的文件系统的快照，并不有为容器分配空间。当要写入一个新文件时，在容器的镜像内为其分配新的块并写入数据，这个叫用时分配。当要修改已有文件时，再使用CoW为容器快照分配块空间，将要修改的数据复制到在容器快照中新的块里再进行修改。Device mapper 驱动默认会创建一个100G的文件包含镜像和容器。每一个容器被限制在10G大小的卷内，可以自己配置调整。结构如下图所示：



#### Btrfs

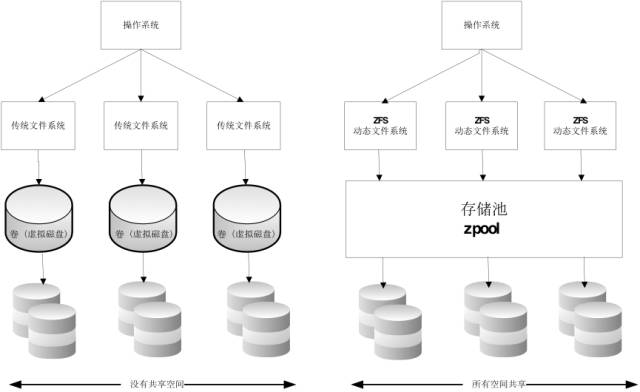
Btrfs被称为下一代写时复制文件系统，并入Linux内核，也是文件级级存储，但可以像Device mapper一直接操作底层设备。Btrfs把文件系统的一部分配置为一个完整的子文件系统，称之为subvolume 。那么采用 subvolume，一个大的文件系统可以被划分为多个子文件系统，这些子文件系统共享底层的设备空间，在需要磁盘空间时便从底层设备中分配，类似应用程序调用 malloc()分配内存一样。为了灵活利用设备空间，Btrfs 将磁盘空间划分为多个chunk 。每个chunk可以使用不同的磁盘空间分配策略。比如某些chunk只存放metadata，某些chunk只存放数据。这种模型有很多优点，比如Btrfs支持动态添加设备。用户在系统中增加新的磁盘之后，可以使用Btrfs的命令将该设备添加到文件系统中。Btrfs把一个大的文件系统当成一个资源池，配置成多个完整的子文件系统，还可以往资源池里加新的子文件系统，而基础镜像则是子文件系统的快照，每个子镜像和容器都有自己的快照，这些快照则都是subvolume的快照。



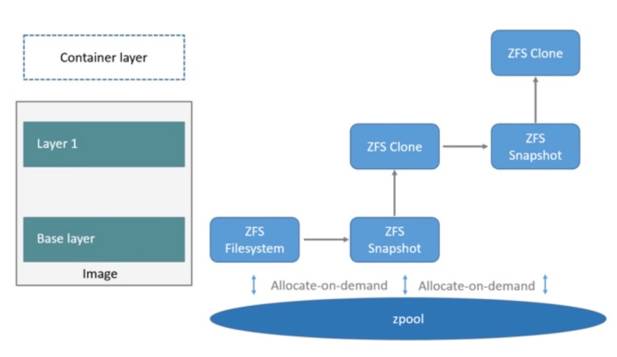
当写入一个新文件时，为在容器的快照里为其分配一个新的数据块，文件写在这个空间里，这个叫用时分配。而当要修改已有文件时，使用CoW复制分配一个新的原始数据和快照，在这个新分配的空间变更数据，变结束再更新相关的数据结构指向新子文件系统和快照，原来的原始数据和快照没有指针指向，被覆盖。

#### ZFS

ZFS 文件系统是一个革命性的全新的文件系统，它从根本上改变了文件系统的管理方式，ZFS 完全抛弃了“卷管理”，不再创建虚拟的卷，而是把所有设备集中到一个存储池中来进行管理，用“存储池”的概念来管理物理存储空间。过去，文件系统都是构建在物理设备之上的。为了管理这些物理设备，并为数据提供冗余，“卷管理”的概念提供了一个单设备的映像。而ZFS创建在虚拟的，被称为“zpools”的存储池之上。每个存储池由若干虚拟设备（virtual devices，vdevs）组成。这些虚拟设备可以是原始磁盘，也可能是一个RAID1镜像设备，或是非标准RAID等级的多磁盘组。于是zpool上的文件系统可以使用这些虚拟设备的总存储容量。



下面看一下在Docker里ZFS的使用。首先从zpool里分配一个ZFS文件系统给镜像的基础层，而其他镜像层则是这个ZFS文件系统快照的克隆，快照是只读的，而克隆是可写的，当容器启动时则在镜像的最顶层生成一个可写层。如下图所示：



当要写一个新文件时，使用按需分配，一个新的数据快从zpool里生成，新的数据写入这个块，而这个新空间存于容器（ZFS的克隆）里。  
当要修改一个已存在的文件时，使用写时复制，分配一个新空间并把原始数据复制到新空间完成修改。



五种存储驱动对比：

AUFS在读的方面性能相比Overlay要差一些，但在写的方面性能比Overlay要好。

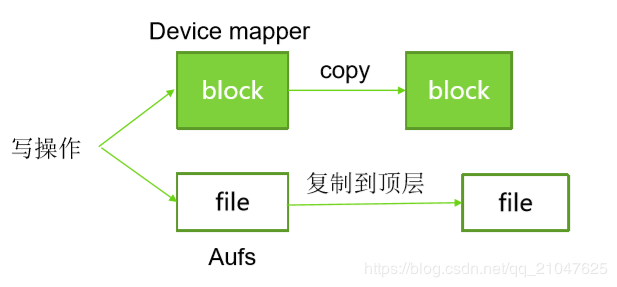
device mapper在512M以上文件的读写性能都非常的差，但在512M以下的文件读写性能都比较好。

btrfs在512M以上的文件读写性能都非常好，但在512M以下的文件读写性能相比其他的存储驱动都比较差。

ZFS整体的读写性能相比其他的存储驱动都要差一些。 简单的测试了一些数据，对测试出来的数据原理还需要进一步的解析。

## 1.2Docker Volume

Docker Copy on Write



和刚刚说的一样, Docker 的写操作主要分为两种, 一个是 块级别的, 一个是文件级别的,  块级别的在修改文件的时候会把修改部分所在的块 copy 到 write层用于写, 文件级别则是直接复制真个文件到 write 层,   所以块级别能够节省空间, 但是块级别 copy的操作次数多, 文件级别直接copy 文件就会是的容器变大, 但是直接copy 文件就会大大减少copy操作的次数.

以上操作对容器中中需要频繁读写的大文件就很不利,  块级别的 COW 操作太频繁,效率低, 文件级别的又会使得容器变得很大, 比如mysql数据库的数据库存储文件.

这个时候就引入了 Volume, Volme 可以将宿主机的文件映射到容器直接操作, 不必经过 COW, 常见的高频写文件有日志系统和数据存储文件.

Vloume使用是通过docker run  的 -v 参数来实现的

[docker@VM\_121\_116\_centos ~]$ docker run -it -p 6379:6379 -v ~/redis/date:/data redis /bin/bash

root@b385ca73cad3:/data# ls

appendonly.aof root tomcat

root@b385ca73cad3:/data# mkdir testdir

root@b385ca73cad3:/data# exit

exit

[docker@VM\_121\_116\_centos ~]$ ll

total 580364

drwxrwxr-x 2 docker docker 4096 Mar 6 22:00 dockerFile

-rw-rw-r-- 1 docker docker 594276824 Mar 6 17:31 ideaIU-191.6014.8.exe

drwxr-xr-x 3 root root 4096 Mar 19 21:26 redis

[docker@VM\_121\_116\_centos ~]$ cd redis/date/

[docker@VM\_121\_116\_centos date]$ ll

total 16

-rw-r--r-- 1 polkitd ssh\_keys 443 Mar 19 21:27 appendonly.aof

-rw-r--r-- 1 polkitd ssh\_keys 249 Mar 19 21:27 root

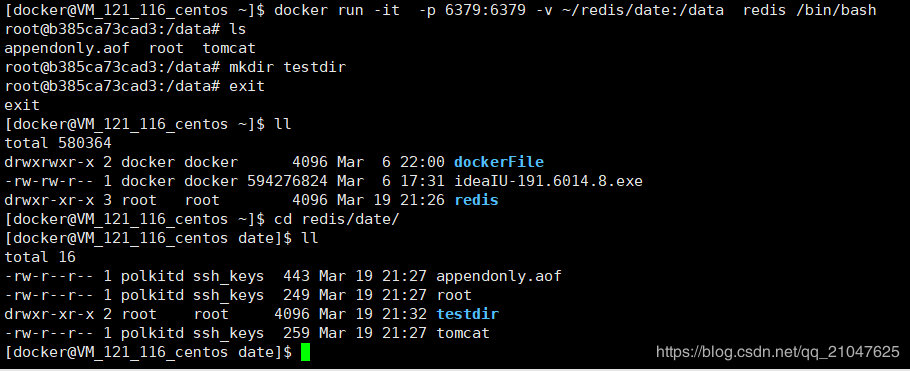
drwxr-xr-x 2 root root 4096 Mar 19 21:32 testdir

-rw-r--r-- 1 polkitd ssh\_keys 259 Mar 19 21:27 tomcat

[docker@VM\_121\_116\_centos date]$

 如命令所示 使用 -v 把宿主机的  ~/redis/date 挂载到 容器的 /data , 之后在容器的 /data  建立一个  testdir 文件夹. 之后在宿主机的 ~/redis/data 里面看见了刚刚建的文件夹. 如不指定宿主机目录, 那么docker就会在docker得volume目录中创建一个目录挂载到容器内, 当容器删除的时候, 这个目录也就随之删除了.

这样容器的 /data 目录就能避免大量的 COW 操作了, 直接进行读写.



在次看看容器信息 使用 docker inspect

[docker@VM\_121\_116\_centos date]$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

b385ca73cad3 redis "docker-entrypoint..." 8 minutes ago Exited (0) 7 minutes ago sad\_murdock

e8abf8c09a26 mysql:5.6 "docker-entrypoint..." 3 weeks ago Exited (0) 2 weeks ago mysql5.6

[docker@VM\_121\_116\_centos date]$ docker inspect b385ca73cad3

[

{

"Id": "b385ca73cad3607f78fc87214dadf23435eebeaf5ee5551a17f160a432b08264",

"Created": "2019-03-19T13:31:55.437267785Z",

"Path": "docker-entrypoint.sh",

"Args": [

"/bin/bash"

],

"State": {

"Status": "exited",

"Running": false,

"Paused": false,

"Restarting": false,

"OOMKilled": false,

"Dead": false,

"Pid": 0,

"ExitCode": 0,

"Error": "",

"StartedAt": "2019-03-19T13:31:55.783433554Z",

"FinishedAt": "2019-03-19T13:32:42.591792526Z"

},

====================省略==================================

"GraphDriver": {

"Name": "overlay2",

"Data": {

"LowerDir": "/var/lib/docker/overlay2/f2806e52d83beca84bd0bb5a137a4a70974fcecdede03e5bc1b3f5c3711619c9-init/diff:/var/lib/docker/overlay2/61d911e09963bc0e77b7020b7bdbf798a5c35dbd90514c810f3aebb6875610f7/diff:/var/lib/docker/overlay2/fdddcb2b85492183f28660470d17c554e5bdb3a01cc8089e9c2112da7eefb4d5/diff:/var/lib/docker/overlay2/9476ee9fd3b0ccad81bdfa469c4188bda49230dff3598e9018cb763a0ba683c1/diff:/var/lib/docker/overlay2/65cfb70d00faeb7477439d635dc0a6f78f4f3d7bb6c5d28b29c6428744a58ee3/diff:/var/lib/docker/overlay2/55e18ec0a5caafed7907512ee5f2dd3e4db5cd701a1adfe5f4fe3bec2911e922/diff:/var/lib/docker/overlay2/ed7f62d6e611f876fb9eb55b06653b23198a915971fd1efdffbb61c33626f759/diff",

"MergedDir": "/var/lib/docker/overlay2/f2806e52d83beca84bd0bb5a137a4a70974fcecdede03e5bc1b3f5c3711619c9/merged",

"UpperDir": "/var/lib/docker/overlay2/f2806e52d83beca84bd0bb5a137a4a70974fcecdede03e5bc1b3f5c3711619c9/diff",

"WorkDir": "/var/lib/docker/overlay2/f2806e52d83beca84bd0bb5a137a4a70974fcecdede03e5bc1b3f5c3711619c9/work"

}

},

"Mounts": [

{

"Type": "bind",

"Source": "/home/docker/redis/date",

"Destination": "/data",

"Mode": "",

"RW": true,

"Propagation": "rprivate"

}

],

=============================省略================================

}

}

}

]

[docker@VM\_121\_116\_centos date]$



可以看到 将  /home/docker/redis/date 挂载到 /data

讲了这么多怎么基于 Volume 互联呢?  我们刚刚把宿主机一个指定的目录挂载到了容器内部, 那么要是我们把这个目录挂载到多个容器有什么效果呢?

我们再次创建一个容器还是挂在这个目录

[docker@VM\_121\_116\_centos date]$ docker run -it -p 26379:6379 -v ~/redis/date:/data --name redistwo redis /bin/bash

root@cf0ca7e833bb:/data# ls

appendonly.aof root testdir tomcat

root@cf0ca7e833bb:/data# mkdir testdirtwo

root@cf0ca7e833bb:/data# exit

exit

为了区别刚刚那个, 我们加了 --name 参数, 进入容器发现有刚刚的建的  testdir, 我们再次建一个 testdirtwo, 再次回到上个容器看看

[docker@VM\_121\_116\_centos date]$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

cf0ca7e833bb redis "docker-entrypoint..." 3 minutes ago Exited (0) 2 minutes ago redistwo

b385ca73cad3 redis "docker-entrypoint..." 26 minutes ago Exited (0) 25 minutes ago sad\_murdock

e8abf8c09a26 mysql:5.6 "docker-entrypoint..." 3 weeks ago Exited (0) 2 weeks ago mysql5.6

[docker@VM\_121\_116\_centos date]$ docker start b385ca73cad3

b385ca73cad3

[docker@VM\_121\_116\_centos date]$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

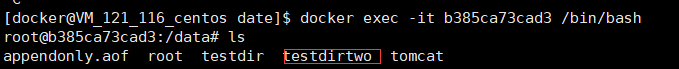
b385ca73cad3 redis "docker-entrypoint..." 28 minutes ago Up 7 seconds 0.0.0.0:6379->6379/tcp sad\_murdock

[docker@VM\_121\_116\_centos date]$ docker exec -it b385ca73cad3 /bin/bash

root@b385ca73cad3:/data# ls

appendonly.aof root testdir testdirtwo tomcat

root@b385ca73cad3:/data#



有第二个容器创建的文件夹, 这样就容器之间的互联

对于跨越宿主机的容器互联, 可是在linux采取共享目录等技术, 或者分布式文件系统

有 iscsi nfs ceph 等

1.3 数据容器共享解决方案(volumes-form)

将本地目录挂载到容器, 在 DockerFile 中是不推荐使用的, 因为这样的容器就不可移植了.

就要不指定宿主机的目录, 使用容器存储位置的目录.

如下:

[docker@VM\_121\_116\_centos date]$ docker run -it -p 6379:6379 -v /data --name oneredis redis /bin/bash

root@6636fb7d293c:/data# mkdir redis

root@6636fb7d293c:/data# ls

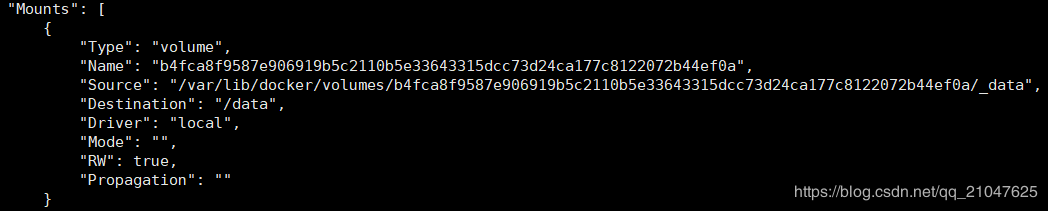
redis

root@6636fb7d293c:/data# exit

exit

[docker@VM\_121\_116\_centos date]$ docker inspect oneredis

查看挂载信息结果: 将 /var/lib/docker/volumes/b4fca8f9587e906919b5c2110b5e33643315dcc73d24ca177c8122072b44ef0a/\_data 挂载到了 /data



我们没有实现约定好共享目录,就不能体检知道挂载的目录,  其他容器怎么从这里读取呢?

这个时候要用 --volumes-from

示例:

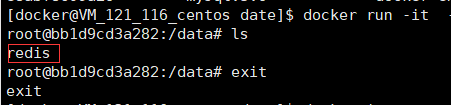
[docker@VM\_121\_116\_centos date]$ docker run -it -p 26379:6379 --volumes-from=oneredis --name tworedis redis /bin/bash

root@bb1d9cd3a282:/data# ls

redis

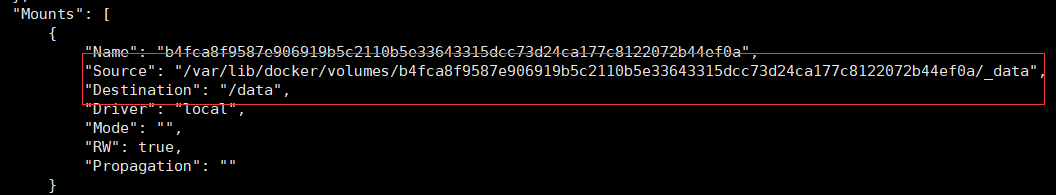
root@bb1d9cd3a282:/data# exit

exit



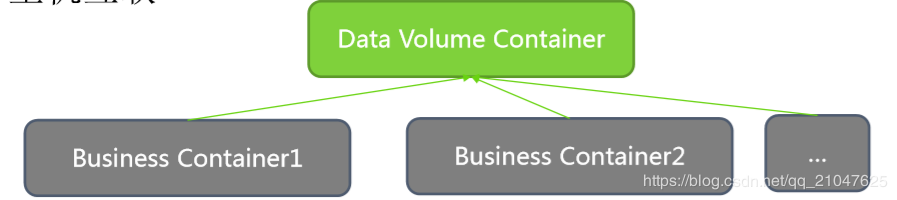
能看到第一个容器创建的目录

再看看容器挂载



和第一个一模一样

这样就解决的移植性的问题



一般的解决方案就是 建议一个 数据共享的管理容器, 其他业务容器 volumes-from 数据管理容器, 这样实现业务容器互联