Docker镜像与容器存储结构分析

发表于 2014-11-20 15:51 | 5049次阅读 | 来源 http://www.cnblogs.com | 2 条评论 | 作者 刁金明

数据存储 Structure Docker

摘要：本文依据已有的条件分析了Docker的镜像与容器的存储结构，并通过一系列小实验深入分析了device mapper和aufs这两种存储结构。希望为大家提供一些有价值的参考，并引出有意义的讨论。

编者按： Docker是一个开源的应用容器引擎，主要利用Linux内核namespace实现沙盒隔离，用Cgroup实现资源限制。 Docker 支持三种镜像层次的存储driver：aufs、device mapper、Btrfs。 广州Linux爱好者 刁金明，通过一系列小实验对Docker的device mapper和aufs这两个存储结构进行了深入分析。以下问原文：

aufs:

AUFS (AnotherUnionFS) 是一种Union FS，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。Aufs driver是Docker最早支持的driver，但是aufs只是Linux内核的一个补丁集，而且不太可能会被加入到Linux内核中。但是由于aufs是唯一一个可以实现容器间共享可执行代码和运行库的storage driver，所以当你跑成千上百个拥有相同程序代码或者运行库的时候，aufs是个相当不错的选择。

device mapper:

Device mapper是Linux 2.6内核中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。

Device mapper driver会创建一个100G的简单文件包含你的镜像和容器。每一个容器被限制在10G大小的卷内， 可以调整。

你可以在启动Docker daemon时用参数-s 指定driver：docker -d -s devicemapper。

Btrfs:

Btufs driver 在Docker build时可以很高效。但是跟device mapper一样不支持设备间共享存储。

下面笔者就已有的条件去分析Docker的镜像与容器的存储结构。

环境：

openSUSE 13.10 + Docker version 1.2.0，build fa7b24f

Ubuntu 14.10 + Docker version 1.0.1，build 990021a

在没有aufs支持的Linux发行版本上（CentOS、openSUSE等），安装Docker可能就使用了device mapper driver。

查看你的Linux发行版有没有aufs支持：lsmod | grep aufs

笔者openSUSE 13.10里是没有加载这个模块的：

而虚拟机里的Ubuntu 14.10 是加载了这个模块的：

而我们列出/var/lib/docker这个目录的内容也可以看出你那个Docker是使用了哪个storage driver：

openSUSE 13.10 上的/var/lib/docker

这里应该看出是使用了device mapper这个driver。

然后再来看看虚拟机Ubuntu 14.10上/var/lib/docker 目录：

这里也可以看出笔者Ubuntu里Docker 是使用了aufs 这个driver，下文就这两个不同的driver作对比。请注意分析的是哪一个。

那么镜像文件在本地存放在哪里呢？

笔者在openSUSE和Ubuntu里把Docker彻底重新安装了一遍，删除了所有镜像，并只Pull下来一个Ubuntu:14.10的镜像，这样分析起来会比较简单明了，现在两个系统都只有一个Ubuntu:14.10的镜像：

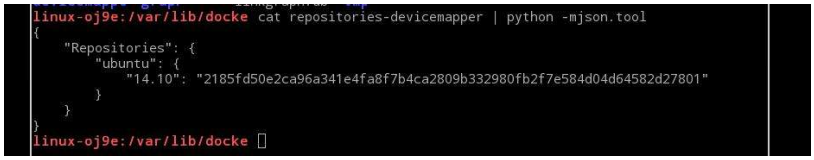
openSUSE：

Ubuntu：

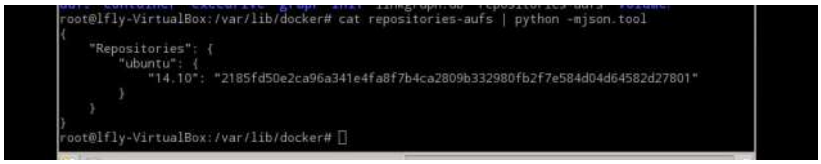
首先现在我们来看看/var/lib/docker里都是什么文件。

1、首先用**Python**的**json.tool**工具查看下**repositories-***里的内容。

openSUSE：



里面的json数据记录的正是本地上存放的镜像的名称及其64位长度的ID。这个ID可以有其12位的简短模式。 Ubuntu上也是一样的：

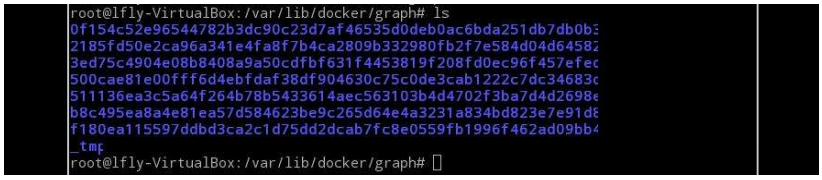


而且我们可以发现这两个ID是一样。这时我们其实可以猜想到：这个ID是全局性的，就是说你这个镜像在镜像仓库上的ID也是这个。被其它机器上ID也是这个。这样的好处无疑是方便管理镜像。

2./var/lib/docker/graph 目录里的内容：

openSUSE：

Ubuntu:

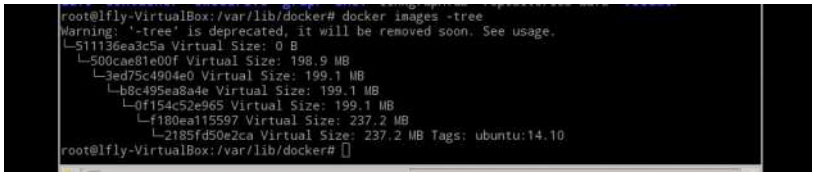


Graph目录里有7个长ID命名的目录，其中第二个长ID是我们所pull下来的Ubuntu14.10镜像的对应的长ID。那么其它6个是怎么来的呢？

这里我们用docker images -tree列出镜像树形结构：

可以看到最下层的镜像是我们的Ubuntu14.10。那么上面对应的是6个layer。就是说在这个树中第n+1个层是基于第n个层上改动的。而第n个层在graph目录里都对应着一个长ID目录。

我们来看看虚拟机里Ubuntu14.10 里的docker images -tree:



大小数量一致。但是到了最后一个层的大小不一样（这里原因可能会是系统问题，也可能是Docker版本问题，具体原因需要另外考察）。

再分析一下各个层的大小，第一个为0B，第二个层就应该为198.9MB，第三个层大小为0.2MB(199.1-198.9)...如此类推下去。

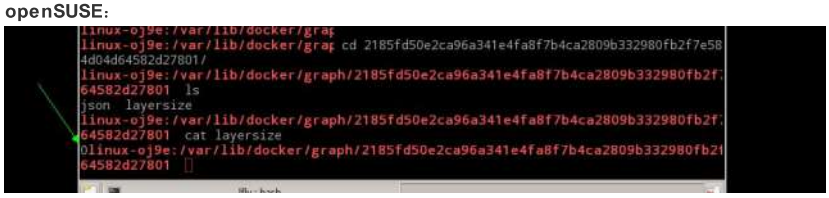
上层的image依赖下层的image（注：这里的逻辑上层是上图树形结构的下层），因此Docker中把下层的image称作父image，没有父image的image称作base image；

例如我要用这里的Ubuntu:14.10为模板启动一个容器时，Docker会加载树形结构中的最下层（2185fd5...），然后加载其父层（f180ea...），这样一直加载到第一层（511136...）才算加载这个rootfs。那么每一层在哪里保存它的父层信息呢？在下面长ID目录里的json文件其实也可以看到这个信息。

graph长**ID**目录内容（对于**Ubuntu**里是一样的，这里以**openSUSE**为例）：

我们进入长ID目录里看看里面的内容：

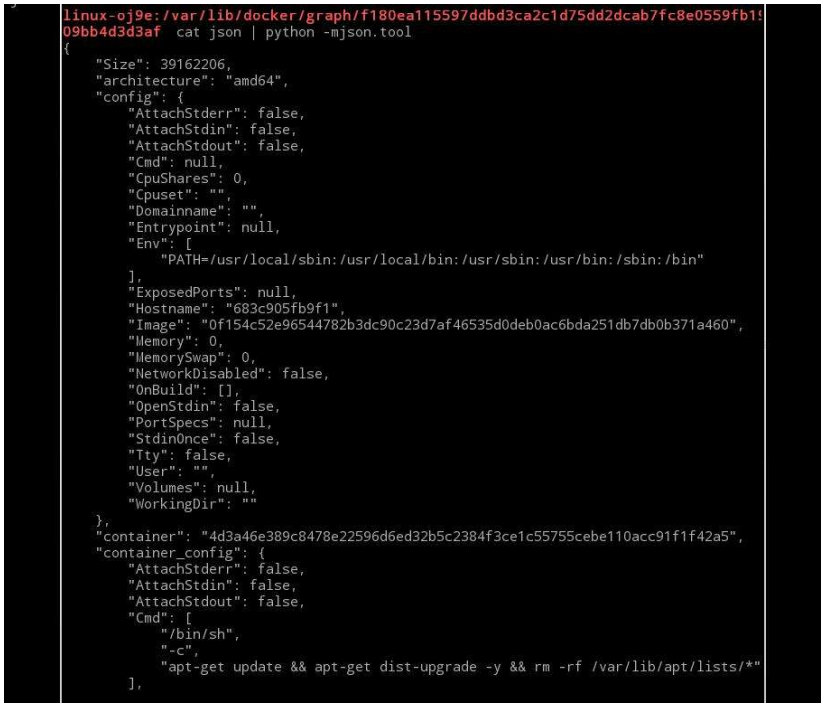
openSUSE：



我们进入最后一个层长ID目录里。里面有一个json文件及一个名为layersize的文件。用cat查看layersize里的内容，里面记录的数字是指这个层的大小。这里(绿色前头)是0。而我们从上面的目录树可以算出最后一个层确实是0。如 果还不相信。我们再算算倒数第二个层的大小(openSUSE里的树形图里短id为f180ea115597的层)应该为37.8M。现在进入对应长ID 目录：

可以看到是是37816084(B)，约37.8M，与我们计算的刚刚吻合。

而另一个文件json又是什么呢？用python工具看看：（内容有点多，没有截完）



可以看到json这个文件保存的是这个镜像的元数据。

拉到底部可以看到有个parent的值：

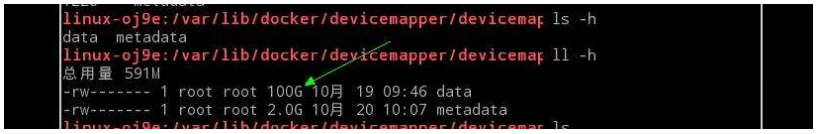
这个就是保存了其父层长ID的值。对照树形结构看f180ea115597 的父层是不是0f154c52e965。

但是注意在graph这个目录里并没有找到我们想找到的镜像内容存放地。只是一些镜像相关的信息数据。

镜像里的内容存放在哪里

openSUSE：

在openSUSE下的/var/lib/docker/devicemapper/devicemapper/这个目录下找到两个文件，并列出其大小。



其中一个data的文件大小为100G(非真实占用)。真实占用的情况如下：

100G的只占用了590M。

上面我们讲到：Device mapper driver会创建一个100G的简单文件包含你的镜像和容器。每一个容器被限制在10G大小的卷内。那么看来这个100G的简单文件正是这个名为data 的文件，那么镜像和容器下是存放在这里的。

好了。这时我在openSUSE上再pull下一个Ubuntu:12.10 镜像看看这个文件大小有什么变化： 这次一下子截了三个命令的信息：

```
linux-oj9e:/ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZ
E
ubuntu              14.10               2185fd50e2ca        6 days ago          236.9 MB
ubuntu              12.10               c5881f11ded9        4 months ago        172.1 MB
linux-oj9e:/ docker images -tree
Warning: '-tree' is deprecated, it will be removed soon. See usage.
└─511136ea3c5a Virtual Size: 0 B
  ├─500cae81e00f Virtual Size: 198.9 MB
  │ └─3ed75c4904e0 Virtual Size: 199.1 MB
  │   └─b8c495ea8a4e Virtual Size: 199.1 MB
  │     └─0f154c52e965 Virtual Size: 199.1 MB
  │       └─f180ea115597 Virtual Size: 236.9 MB
  │         └─2185fd50e2ca Virtual Size: 236.9 MB Tags: ubuntu:14.10
  └─bac448df371d Virtual Size: 100.9 MB
    └─dfaad36d8984 Virtual Size: 101.1 MB
      └─5796a7edb16b Virtual Size: 101.1 MB
        └─c5881f11ded9 Virtual Size: 172.1 MB Tags: ubuntu:12.10
linux-oj9e:/
linux-oj9e:/ du -sh /var/lib/docker/devicemapper/devicemapper/*
787M    /var/lib/docker/devicemapper/devicemapper/data
1.4M    /var/lib/docker/devicemapper/devicemapper/metadata
linux-oj9e:/
linux-oj9e:/
```

Pull下来的Ubuntu是172.1M，树形结构可以看到各个层的关系。而data的大小变成了787M。没pull Ubuntu:12.10之前是590M，增加了197M，跟pull下来的172.1M有点差距。这里可认为是存储了额外的某些信息。

那么容器是不是也存放在这里呢？

我们用Ubuntu14.10启动一个模板看看情况如何：

```
linux-oj9e:/
linux-oj9e:/ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
        PORTS               NAMES
a9b35d72fcd4        ubuntu:14.10        "/bin/bash"         2 minutes ago       Up 2 minute
s                           trusting_hoover
linux-oj9e:/
linux-oj9e:/ du -sh /var/lib/docker/devicemapper/devicemapper/*
789M    /var/lib/docker/devicemapper/devicemapper/data
1.5M    /var/lib/docker/devicemapper/devicemapper/metadata
linux-oj9e:/
linux-oj9e:/ ll /var/lib/docker/containers/
总用量 4
drwx------ 2 root root 4096 10月  20 10:40 a9b35d72fcd433ac91e9aa83bec002f13fbd7827422f
04b3866a639a18
linux-oj9e:/
linux-oj9e:/ ll /var/lib/docker/containers/a9b35d72fcd433ac91e9aa83bec002f13fbd7827422ff9
14704b3866a639a187/
总用量 20
-rw------- 1 root root    0 10月  20 10:40 a9b35d72fcd433ac91e9aa83bec002f13fbd7827422ff9147
04b3866a639a187-json.log
-rw-r--r-- 1 root root 1490 10月  20 10:40 config.json
-rw-r--r-- 1 root root  284 10月  20 10:40 hostconfig.json
-rw-r--r-- 1 root root   13 10月  20 10:40 hostname
-rw-r--r-- 1 root root  174 10月  20 10:40 hosts
-rw-r--r-- 1 root root  834 10月  20 10:40 resolv.conf
linux-oj9e:/
```

这次我也是一下子截了几个命令，可以看到了一个基于Ubuntu:14.10镜像的容器在运行中，简短ID是a9b35d72fcd4，

第二个命令du列出了data的大小为789M，增加了2M。

第三个命令列出了container目录内出现一个长ID的目录，ID就是运行的容器的ID。但是里面的文件应该都是些配置文件。并没有我们想要的内容目录。

这样的话我们进一步做测试：在运行的容器内使用dd if=/dev/zero of=test.txt bs=1M count=8000 创建一个8G大小的文件后：

```
linux-oj9e:/var/lib/docker/devicemapper/devicemap ls
data  metadata
linux-oj9e:/var/lib/docker/devicemapper/devicemap du -sh *
8.6G    data
9.4M    metadata
linux-oj9e:/var/lib/docker/devicemapper/devicemap
```

这里data变成了8.6G，增长了接近8G，这样也证实了容器里的内容是保存在data这个简单文件内的。

这样的话证实了device mapper driver是把镜像和容器的文件都存储在data这个文件内。

Ubuntu 的aufs driver又如何呢：

Ubuntu上由于是aufs driver，所以/var/lib/docker 目录下有aufs目录而不是devicemapper 目录：

```
root@lfly-VirtualBox:/var/lib/docker# ls
aufs       execdrive  init          repositories-aufs
container  graph      linkgraph.db  volumes
root@lfly-VirtualBox:/var/lib/docker# cd aufs
root@lfly-VirtualBox:/var/lib/docker/aufs# ls
diff  layers  mnt
root@lfly-VirtualBox:/var/lib/docker/aufs#
root@lfly-VirtualBox:/var/lib/docker/aufs#
```

这里的aufs 目录有三个目录，diff 、layers 、mnt 三个目录。

这里layers目录是保存了layers层次信息，并不是layers里面的内容。

而diff 目录时有数个长ID目录：

```
root@lfly-VirtualBox:/var/lib/docker/aufs/diff#
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# docker images -tree
Warning: '-tree' is deprecated, it will be removed soon. See usage.
└─511136ea3c5a Virtual Size: 0 B
  └─500cae81e00f Virtual Size: 198.9 MB
    └─3ed75c4904e0 Virtual Size: 199.1 MB
      └─b8c495ea8a4e Virtual Size: 199.1 MB
        └─0f154c52e965 Virtual Size: 199.1 MB
          └─f180ea115597 Virtual Size: 237.2 MB
            └─2185fd50e2ca Virtual Size: 237.2 MB Tags: ubuntu:14.10
root@lfly-VirtualBox:/var/lib/docker/aufs/diff#
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# du -sh *
24K     0f154c52e96544782b3dc90c23d7af46535d0deb0ac6bda251db7db0b371a460
12K     2185fd50e2ca96a341e4fa8f7b4ca2809b332980fb2f7e584d04d64582d27801
300K    3ed75c4904e08b8408a9a50cdfbf631f4453819f208fd0ec96f457efedddaa28
217M    500cae81e00fff6d4ebfdaf38df904630c75c0de3cab1222c7dc34683dc4ae41
4.0K    511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
28K     b8c495ea8a4e81ea57d584623be9c265d64e4a3231a834bd823e7e91d831d1f3
42M     f180ea115597ddbd3ca2c1d75dd2dcab7fc8e0559fb1996f462ad09bb4d3d3af
root@lfly-VirtualBox:/var/lib/docker/aufs/diff#
```

列出这几个目录的大小可以看出基本与上面树形结构的所能计算的大小相对应（相关部分可能是由于压缩或者其它原因造成，这里纯属猜测）。

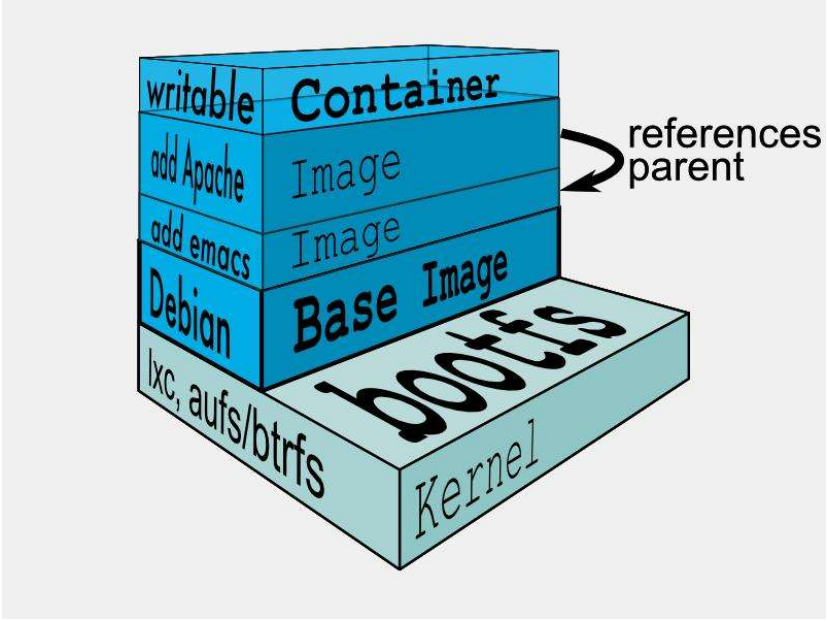
那我们进入f180ea115597这个ID对应的目录看看里面是什么：

```
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# cd f180ea115597ddbd3ca2c1d75dd
2dcab7fc8e0559fb1996f462ad09bb4d3d3af/
root@lfly-VirtualBox:/var/lib/docker/aufs/diff/f180ea115597ddbd3ca2c1d75dd2dca
b7fc8e0559fb1996f462ad09bb4d3d3af# ls
etc  tmp  usr  var
root@lfly-VirtualBox:/var/lib/docker/aufs/diff/f180ea115597ddbd3ca2c1d75dd2dca
```

里面是一些文件夹，但是只有几个，并不像我们平时常规Linux发行版里的那么齐全。

这里的话其实我们可以想到了因为一个层是基于另一个层之上的。Aufs文件系统可以做到增量修改，所以这里的几个文件夹是基于上一个层做的修改内容增量地保存在这里，因为上一个层对于这个层来说不可写：

在这里我需要先引用一张网上的图片：



这里我们可以看到一个我们想象中的运行中的container是包含了若干个read only的image层，然后最上面的writable层才是我们可写的层。第一个readonly的层会加载其父层。直到最下面的base image层。

我们所做的改动会被保存在最上面的那个writable层里。当我们用commit 把容器固化成镜像时那个层就会变成我们上面看到的“目录不齐全的”长ID目录。

为了证实这一点，我们在运行一个基于Ubuntu:14.10镜像的容器：

```
root@lfly-VirtualBox:/var/lib/docker/aufs#
root@lfly-VirtualBox:/var/lib/docker/aufs# docker run -it ubuntu:14.10
root@7b3c13323d8c:/#
root@7b3c13323d8c:/#
root@7b3c13323d8c:/#
```

可以看到运行的容器简短ID为7b3c13323d8c 。

这时再列出diff目录的内容：

```
root@lfly-VirtualBox:/var/lib/docker/aufs/diff/f180ea115597ddbd3ca2c1d75dd2dca
b7fc8e0559fb1996f462ad09bb4d3d3af# cd ..
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# du -sh *
24K     0f154c52e96544782b3dc90c23d7af46535d0deb0ac6bda251db7db0b371a460
12K     2185fd50e2ca96a341e4fa8f7b4ca2809b332980fb2f7e584d04d64582d27801
300K    3ed75c4904e08b8408a9a50cdfbf631f4453819f208fd0ec96f457efedddaa28
217M    500cae81e00fff6d4ebfdaf38df904630c75c0de3cab1222c7dc34683dc4ae41
4.0K    511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
12K     7b3c13323d8ce1280f904c5b1de6c344d2ab35b5bcc36afd144cd62953a13084
24K     7b3c13323d8ce1280f904c5b1de6c344d2ab35b5bcc36afd144cd62953a13084-init
28K     b8c495ea8a4e81ea57d584623be9c265d64e4a3231a834bd823e7e91d831d1f3
42M     f180ea115597ddbd3ca2c1d75dd2dcab7fc8e0559fb1996f462ad09bb4d3d3af
root@lfly-VirtualBox:/var/lib/docker/aufs/diff#
```

多了两个长ID目录，正是我们运行的容器的ID，列出内容：

```
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# ll 7b3c13323d8ce1280f904c5b1de
6c344d2ab35b5bcc36afd144cd62953a13084
total 16
drwxr-xr-x  4 root root 4096 Oct 20 10:20 ./
drwxr-xr-x 11 root root 4096 Oct 20 10:20 ../
-r--r--r--  1 root root    0 Oct 20 10:20 .wh..wh.aufs
drwx------  2 root root 4096 Oct 20 10:20 .wh..wh.orp/
drwx------  2 root root 4096 Oct 20 10:20 .wh..wh.pln/
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# ll 7b3c13323d8ce1280f904c5b1de
6c344d2ab35b5bcc36afd144cd62953a13084-init/
total 24
drwxr-xr-x  6 root root 4096 Oct 20 10:20 ./
drwxr-xr-x 11 root root 4096 Oct 20 10:20 ../
drwxr-xr-x  3 root root 4096 Oct 20 10:20 dev/
-rwxr-xr-x  1 root root    0 Oct 20 10:20 .dockeren*
-rwxr-xr-x  1 root root    0 Oct 20 10:20 .dockerini*
drwxr-xr-x  2 root root 4096 Oct 20 10:20 etc/
-r--r--r--  2 root root    0 Oct 20 10:20 .wh..wh.aufs
drwx------  2 root root 4096 Oct 20 10:20 .wh..wh.orp/
drwx------  2 root root 4096 Oct 20 10:20 .wh..wh.pln/
root@lfly-VirtualBox:/var/lib/docker/aufs/diff#
```

然后我们在运行的容器中创建一个/test 目录，并在里面用dd命令创建一个8G的test.txt文件：完成这些后再列出这两个目录内容：

```
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# du -sh *
24K     0f154c52e96544782b3dc90c23d7af46535d0deb0ac6bda251db7db0b371a460
12K     2185fd50e2ca96a341e4fa8f7b4ca2809b332980fb2f7e584d04d64582d27801
300K    3ed75c4904e08b8408a9a50cdfbf631f4453819f208fd0ec96f457efedddaa28
217M    500cae81e00fff6d4ebfdaf38df904630c75c0de3cab1222c7dc34683dc4ae41
4.0K    511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
7.5G    7b3c13323d8ce1280f904c5b1de6c344d2ab35b5bcc36afd144cd62953a13084
24K     7b3c13323d8ce1280f904c5b1de6c344d2ab35b5bcc36afd144cd62953a13084-init
28K     b8c495ea8a4e81ea57d584623be9c265d64e4a3231a834bd823e7e91d831d1f3
42M     f180ea115597ddbd3ca2c1d75dd2dcab7fc8e0559fb1996f462ad09bb4d3d3af
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# ll 7b3c13323d8ce1280f904c5b1de
6c344d2ab35b5bcc36afd144cd62953a13084
total 20
drwxr-xr-x  5 root root 4096 Oct 20 11:23 ./
drwxr-xr-x 11 root root 4096 Oct 20 10:20 ../
drwxr-xr-x  2 root root 4096 Oct 20 11:24 test/
-r--r--r--  1 root root    0 Oct 20 10:20 .wh..wh.aufs
drwx------  2 root root 4096 Oct 20 10:20 .wh..wh.orp/
drwx------  2 root root 4096 Oct 20 10:20 .wh..wh.pln/
root@lfly-VirtualBox:/var/lib/docker/aufs/diff# ll 7b3c13323d8ce1280f904c5b1de
6c344d2ab35b5bcc36afd144cd62953a13084-init/
total 24
drwxr-xr-x  6 root root 4096 Oct 20 10:20 ./
drwxr-xr-x 11 root root 4096 Oct 20 10:20 ../
drwxr-xr-x  3 root root 4096 Oct 20 10:20 dev/
-rwxr-xr-x  1 root root    0 Oct 20 10:20 .dockeren*
-rwxr-xr-x  1 root root    0 Oct 20 10:20 .dockerini*
drwxr-xr-x  2 root root 4096 Oct 20 10:20 etc/
-r--r--r--  2 root root    0 Oct 20 10:20 .wh..wh.aufs
drwx------  2 root root 4096 Oct 20 10:20 .wh..wh.orp/
drwx------  2 root root 4096 Oct 20 10:20 .wh..wh.pln/
root@lfly-VirtualBox:/var/lib/docker/aufs/diff#
```

多出来的目录

可以看到其中一个目录（没有init后缀）变成了7.5G，而另一个目录还是24K。

在长ID目录里还多了一个test文件夹，正是我们在容器里创建的，这样的话里面无疑问就是test.txt文件了。容器通过这种方法在writable层里记录了修改过的内容（增量记录）。（这里有个小问题笔者也还不清楚：怎么记录删除了东西呢？这个问题以后再考察）

从上面我们可以知道容器的writable 层是保存在以容器ID为名的长ID目录里的，而ID+init后缀目录是保存容器的初始信息的。

好了，现在我们进行最后一个实验：把容器固化成镜像。

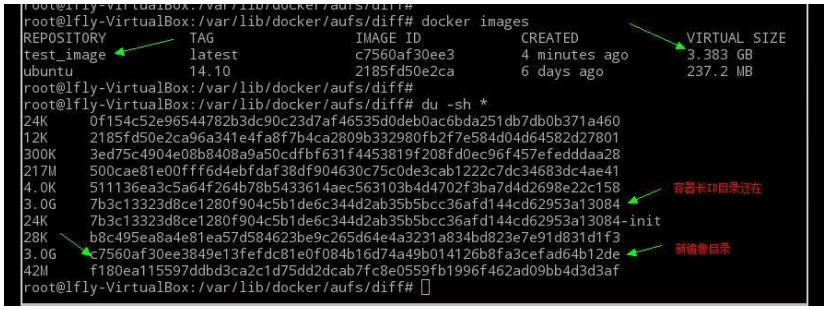
（这里要做个小小调整。把上面8G的文件删除了再建一个3G大小的文件test_3G.txt代替）

```
root@7b3c13323d8c:/test# rm test.txt
root@7b3c13323d8c:/test# dd if=/dev/zero of=test_3G.txt bs=1M count=3000
3000+0 records in
3000+0 records out
3145728000 bytes (3.1 GB) copied, 10.3978 s, 303 MB/s
root@7b3c13323d8c:/test#
root@7b3c13323d8c:/test#
```

```
root@7b3c13323d8c:/test# exit
exit
root@lfly-VirtualBox:/# docker commit 7b3c13323d8c test_image
c7560af30ee3849e13fefdc81e0f084b16d74a49b014126b8fa3cefad64b12de
root@lfly-VirtualBox:/#
```

Commit 后把容器固化成了test_image的镜像，得到那个镜像的长ID。

现在看看变化：



那个窗口目录还在，原因是我们还没用rm 命令删除那个容器。而多出来的镜像目录正是我们固化所得到的，其大小与上面容器writable层大小一致为3GB。现在看看里面是什么内容：

里面有一个test目录，目录下对应我们创建的3GB大小的test_3G.txt文件，这就是我们改动过的内容保存了在这个目录内。

现在我们用rm命令删除容器看看结果：

容器被删除了，其对应的长目录ID也被删除了。而那个固化的得到的镜像（c7560af30）被保存了下来。

通过上面的小实验基本可以看清Docker在devicemapping和aufs这两个driver的存储结构，但是这些目录是怎样灵活地在运行容器时被加载到一起，就需要读者去了解更深层的关于aufs及devicemapping相关的知识。

参考文献：

Docker官方文档： https://docs.docker.com/reference/commandline/cli/

Docker存储结构： http://blog.thoward37.me/articles/where-are-docker-images-stored/

原文链接： docker镜像与容器存储结构分析（责编：周小璐）

台，大数据实践，大数据产业资讯等服务。

顶 4 踩 0

推荐阅读相关主题： 存储 结构 linux内核 大数据技术大会 文件系统 项目管理

相关文章 最新报道

已有2条评论

还可以再输入500个字

有什么感想，你也来说说吧！

黑洞 欢迎您！ 发表评论

最新评论 最热评论

LiangZhang0924 2014-11-21 16:46
感谢分享，感谢分享！

回复

sayume 2014-11-21 10:01
恭喜作者在写了这么长的一篇文章之后终于发现了docker的增量文件系统原理，其实这一点在官方文档里面早就明确指出了。即使要做实验，也应该是在同一个镜像上，使用dockerfile配置不同的应用程序，来看看docker是怎么做的

回复

共1页 首页 上一页 1 下一页 末页

请您注意
·自觉遵守：爱国、守法、自律、真实、文明的原则
·尊重网上道德，遵守《全国人大常委会关于维护互联网安全的决定》及中华人民共和国其他各项有关法律法规
·严禁发表危害国家安全，破坏民族团结、国家宗教政策和社会稳定，含侮辱、诽谤、教唆、淫秽等内容的作品
·承担一切因您的行为而直接或间接导致的民事或刑事法律责任
·您在CSDN新闻评论发表的作品，CSDN有权在网站内保留、转载、引用或者删除
·参与本评论即表明您已经阅读并接受上述条款