Docker(一) 文件系统

什么是容器

先来说说什么是容器，我们都跟刚接触 docker 的新手讲，你就先把 docker 当虚拟机。因为我们怕一开始就讲容器巴拉巴拉的会让人懵逼。反正从使用者的角度来看，大多数情况下也感觉不出来它跟虚拟机有什么区别。 所以不了解容器并不会对我们使用 docker run， docker build， docker ps
这些命令搭建服务有丝毫的阻碍。就如我们开车的时候不用非得知道汽车里面是怎么构造的。所以如果读者之前没有使用过 docker 的话，我强烈建议各位先去体验一番，有一个基本的了解之后再来继续读本篇文章，这样你能够更好的理解下面的内容。

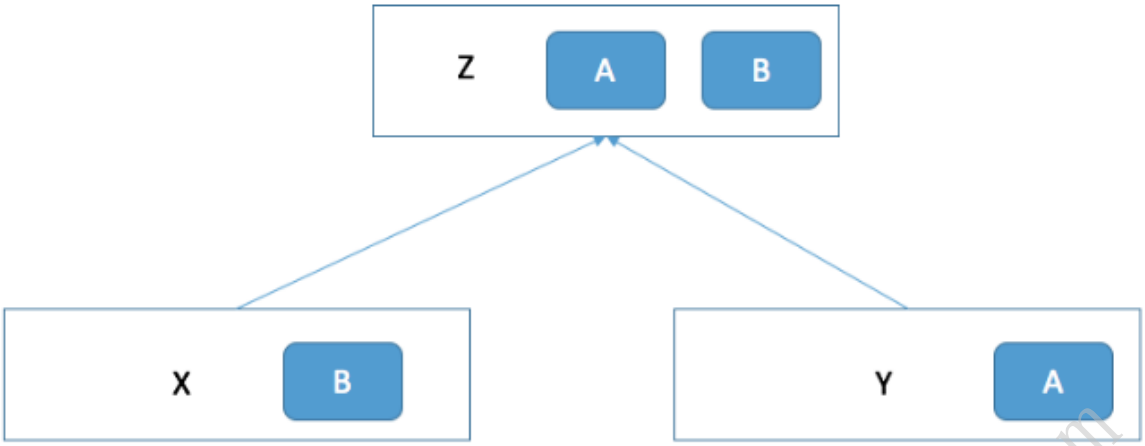
那么正题开始，不懂容器原理虽然并不影响我们作为一个 docker 的使用者。但我们的最终目的是通过 docker+k8s 构建并维护一个容器集群并提供给公司内部大量使用，所以光学会开车是不行的，我们还得学会修车。所以接下来我们看看容器到底是个什么东东。 首先当我们运行 docker run 的时候，我们知道是根据一个镜像构建出了一个容器， 我们看看这个容器里有什么 (假设这里我使用的是一个 centos7.2 的镜像)。 你可以通过 docker exec -it 容器名称 bash 来进入容器。

```
gaofei@docker02:~/test/release25$ prophet2.5 dgo -n release25
Python info: 2.7.12
OpenSSL info: OpenSSL 1.0.2g  1 Mar 2016
[2017-07-24 12:02:33.995797] DEBUG: prophet: export KUBECONFIG=$HOME/admin.conf;kubectl exec -it release25-deploy-880910729-m52t8 b
sh
[root@release25-deploy-880910729-m52t8 /]# pwd
/
[root@release25-deploy-880910729-m52t8 /]# ll
total 38
-rw-r--r--    1 root root 15747 Mar 15 20:00 anaconda-post.log
lrwxrwxrwx    1 root root     7 Mar 15 19:58 bin -> usr/bin
drwxr-xr-x    5 root root   360 Jul 24 09:21 dev
drwxr-xr-x   65 root root     6 Jul 24 09:21 etc
drwxr-xr-x    6 root root     3 Jul 24 09:21 home
lrwxrwxrwx    1 root root     7 Mar 15 19:58 lib -> usr/lib
lrwxrwxrwx    1 root root     9 Mar 15 19:58 lib64 -> usr/lib64
drwx------    2 root root     2 Mar 15 19:58 lost+found
drwxr-xr-x    2 root root     2 Nov  5  2016 media
drwxr-xr-x    2 root root     2 Nov  5  2016 mnt
drwxr-xr-x    2 root root     2 Nov  5  2016 opt
dr-xr-xr-x 1887 root root     0 Jul 24 09:21 proc
dr-xr-x---    4 root root     3 May 15 10:44 root
drwxr-xr-x   11 root root     4 Jul 24 09:21 run
lrwxrwxrwx    1 root root     8 Mar 15 19:58 sbin -> usr/sbin
drwxr-xr-x    2 root root     2 Nov  5  2016 srv
dr-xr-xr-x   13 root root     0 Jun 12 02:27 sys
drwxrwxrwt   22 root root    18 Jul 24 09:48 tmp
drwxr-xr-x   20 root root     9 Mar 28 09:36 usr
drwxr-xr-x   27 root root     3 Jul 24 09:21 var
[root@release25-deploy-880910729-m52t8 /]#
```

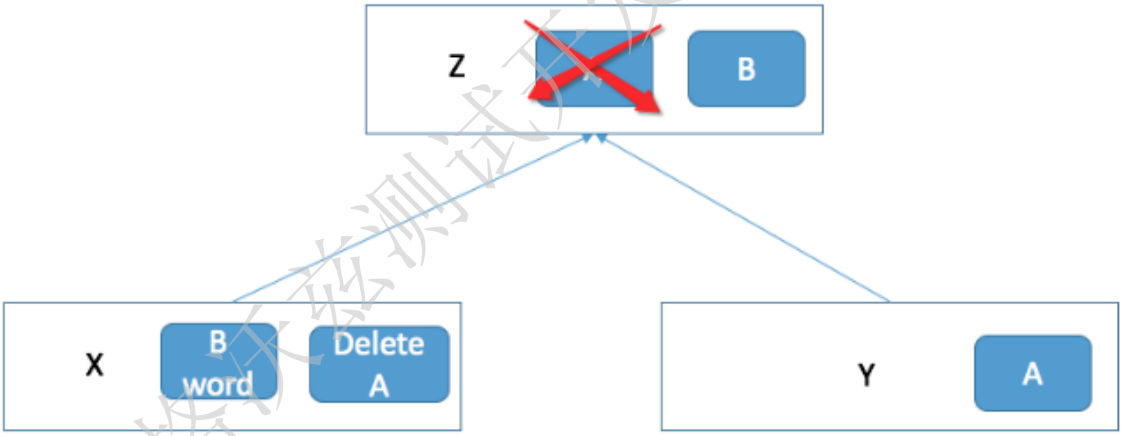

可以看到我们通过命令进入了容器中，我们发现这是一个完整的操作系统，有 root 目录，有 bin，有 sbin，有 proc。运行一切 centos 的命令也都是正常的。好像我们就是在一个虚拟机一样。好像跟宿主机没有半毛钱关系 (我这里的实验宿主机是 ubantu)。 但这是个假象，我们在容器里运行的任何命令其实就是在宿主机上运行的，是 docker
耍了一些手段让我们以为是在一个跟宿主机不相关的完整的操作系统上，但其实我们在容器里运行的任何命令在本质上就是在宿主机上运行的。 这就是容器和虚拟机最大的区别。 虚拟机是个完整的操作系统，与宿主机完全隔离，在虚拟机上运行的任何东西都不会影响宿主机。
而容器不是， 容器就是个骗局，它让你以为你是运行在一个完整的操作系统上，其实你们都是在宿主机上玩而已。
所以我们之前一定听过类似这样的话：容器是节省资源的，是共享宿主机内核的，容器上运行的东西是有可能把宿主机搞挂的。 那么 docker 到底耍了什么手段，我们先从联合文件系统说起

联合文件系统：aufs

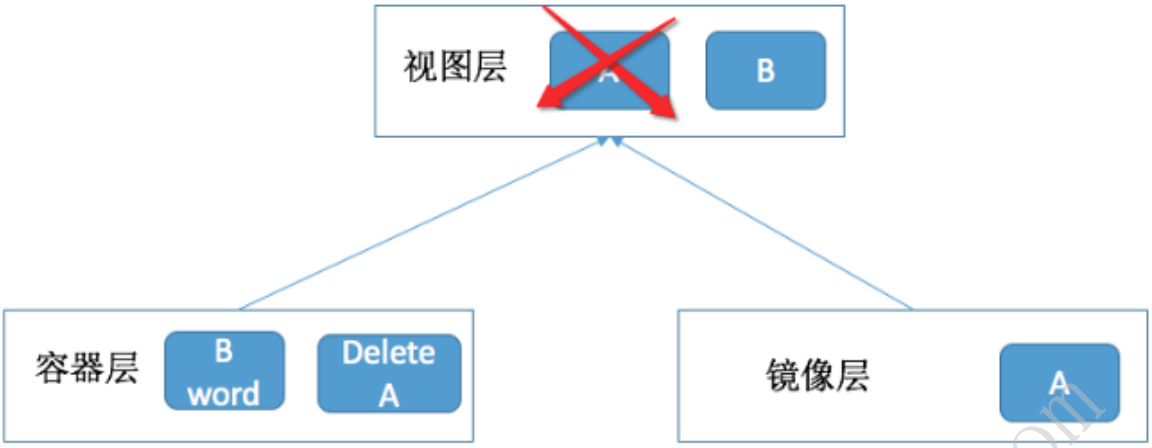
aufs，是联合文件系统的简写。是 docker 最早版本的文件系统。 通过它的特性 docker
实现了镜像层的重叠，容器层的存储和展示层的显示。



上面这个图就是 AUFS 的工作原理。 假设我们现在有两个目录 X 和 Y， 他们分别有 A 和 B 两个文件。 那么 aufs 的作用就是可以将这两个目录合并，并挂载到一个新的设备上也就是 Z 上。 在 Z 这个目录上我们可以同时看到 A 和 B 这两个文件。这就是为什么叫联合文件系统。它的意思就是可以把多个文件联合在一起成为一个统一的视图。 那么如果我们在 Z 目录上 (也就是联合文件系统的视图上) 修改 A 和 B 会发生什么事情？ X 和 Y 这两个目录下的 A 和 B 文件也会相应修改么？ 我们来试试， 假如我们删除 A 文件， 并在 B 文件中添加一个单词 Hello。我们会发现出现了下面的变化。



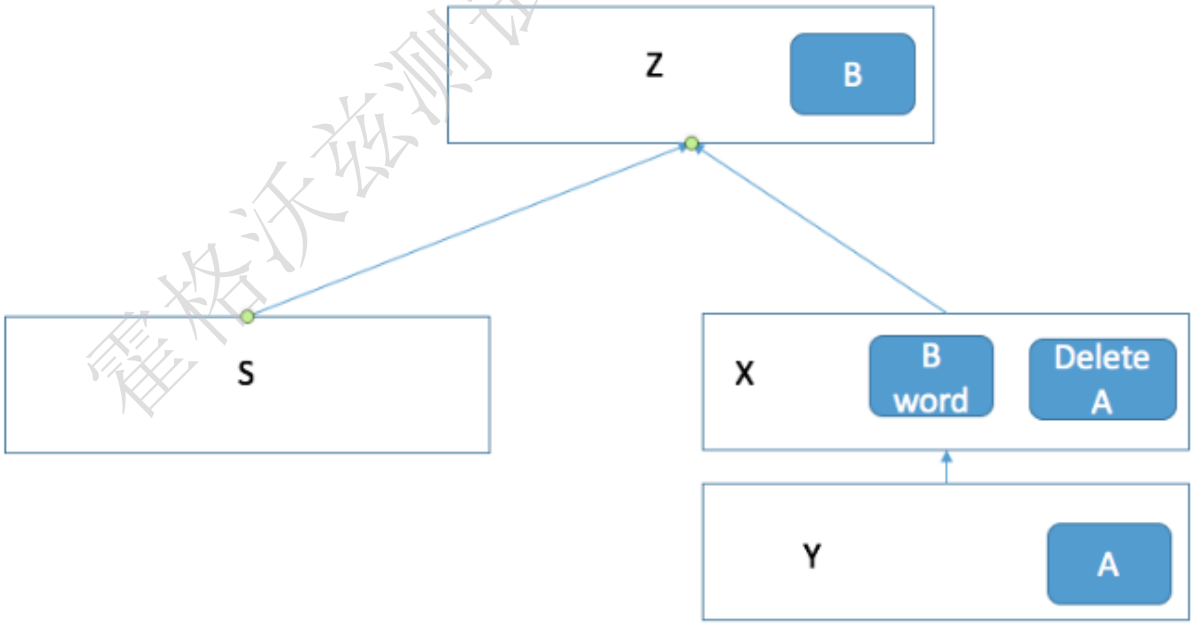
我们在 Z 目录中对 A 和 B 做出了修改。发现原本拥有 B 文件的 X 发生了改动。 但是有用 Y 中的 A 文件并没有发生改变。 相反的，我们发现在 X 目录中多了一个删除 A 文件的记录。 这是为什么？ 这就是 aufs 的一个特性。 在所有联合起来的目录中，只有第一个目录是有写权限的。 也就是说 Z 不论修改什么都只能对第一个联合进来的 X 做更改，它是没有权限对其他目录 (例如 Y) 做任何改动的。 如果我们在 Z 目录中对 A 做出了更改，它是没有权限修改 A 的，所以相应的在 X 中添加一条记录，记录了更改 A 的内容。 这就是 aufs。大家仔细想一下这样的特性会产生什么效果? 看下面的图。



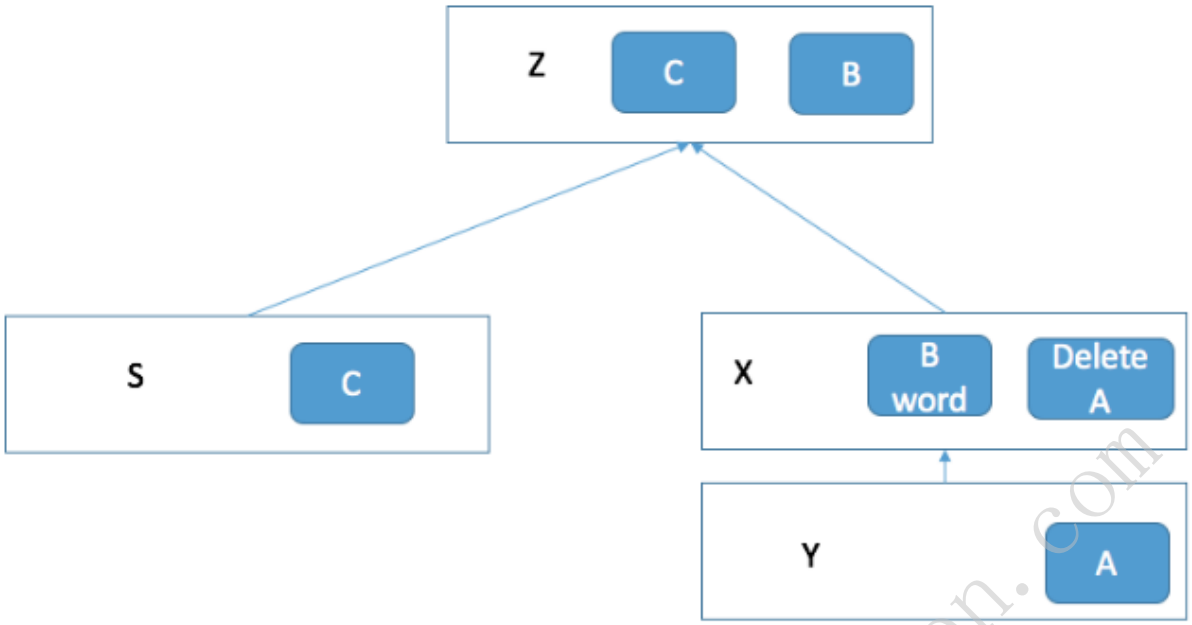
我们之前的 Y 就是镜像层，X 就是容器层，而联合文件目录
Z，就是我们进入容器以后展示给我们看的视图层。不知道大家到了这里脑袋能否转过弯来。Y 是永远无法更改的，所以它是镜像层。X
是可以修改的，并且它可以记录对镜像层的修改来达到定制自己的文件的目的，这样在视图层上，我们就看到的是经过容器层修改后的视图了
。 这就是 docker 使用 aufs 对文件系统耍的花样。 我们下载各种操作系统的镜像其实是做的很像是操作系统的镜像层。这个镜像层里有
root，有 bin，有 sbin，有 proc，有一切你想要的命令脚本。 docker
会创建一个空的容器层并和镜像层联合起来，成为一个视图给我们看，并修改了各种环境变量让所有命令都指向镜像层去。
这样就有了在一开始我们进入容器后的那个截图。 我们看到的甚至不是容器层，而是联合起来的视图层。
所以我们其实还是在宿主机上的用户， 只不过我们看不出来。

docker 的分层镜像

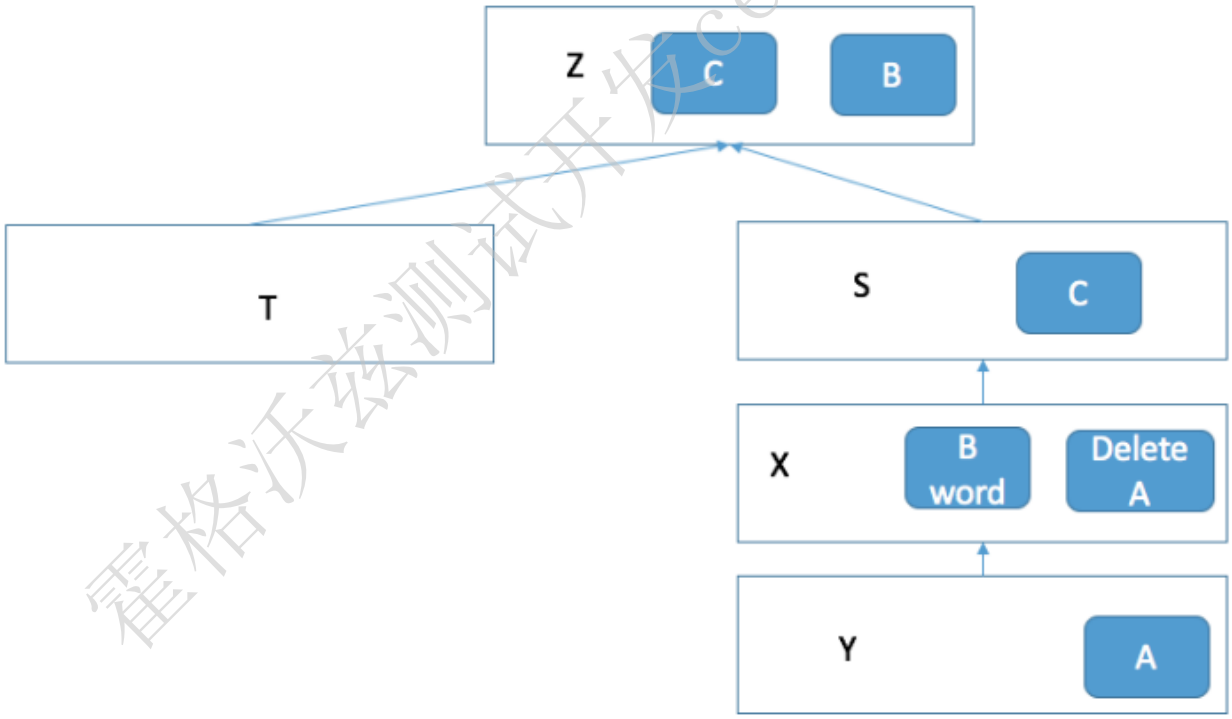
了解了 AUFS 之后我们就可以来看看 docker 的分层镜像了。还是上面的例子，假设现在我们想根据现在对 X 和 Y
的改动构建一个新的镜像，并用这个新的镜像运行一个容器会是什么样子的？ 看了下面的图你应该就能了解了。



X 和 Y 变成了一个两层的镜像层。新创建出的 S 作为容器层继续接受视图层的更改请求。如果你想继续更改，例如添加一个文件 C。
那么就会变成下面这样。



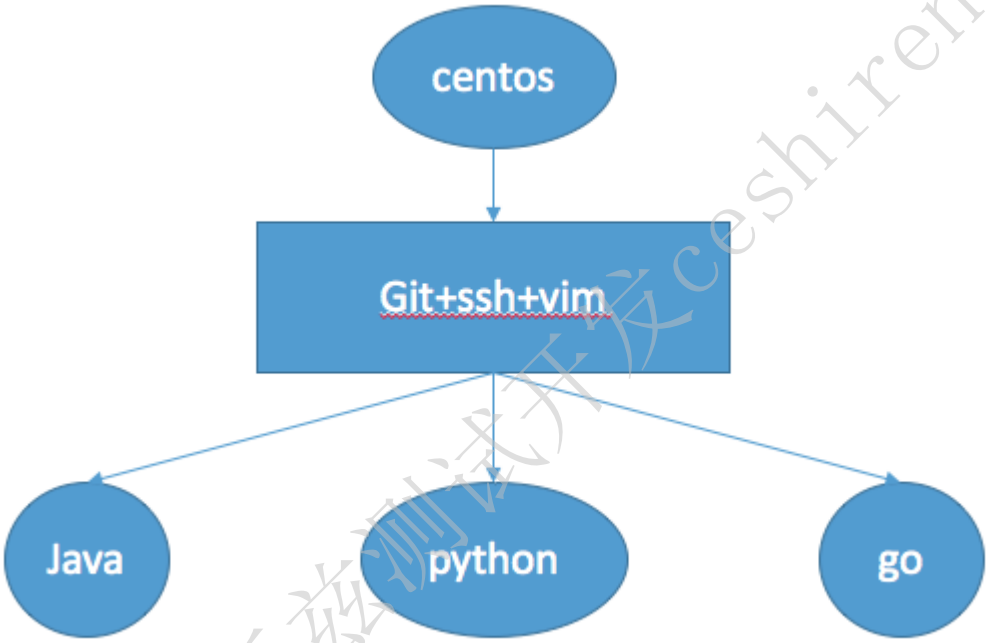
你想把这个最新的修改也做成镜像，并再跑一个容器出来，那么就会变成下面的这个样子。



通过这种上层镜像只记录对下层镜像的更改的方式，我们可以把镜像一层一层的重叠起来。这就是 docker 的分层镜像系统.

```
gaofei@docker02:~/test/release26$ docker push [redacted] compile:2.5.0
The push refers to a repository [redacted]
a8ad3ed4dbad: Layer already exists
aee97157e1c7: Layer already exists
d976d14dd915: Layer already exists
911632b06082: Layer already exists
69c728abd204: Layer already exists
560d3f602160: Layer already exists
d626432497cc: Layer already exists
ee54ea887c00: Layer already exists
d275bb60cf29: Layer already exists
cdc030ccc219: Layer already exists
2.5.0: digest: sha256:49f0312152e31cd908a84633f99ebef2664b2686b4a9349ff30b41f7858df7ce size: 2428
```

这是我再测试环境中 push 的一个镜像。 可以看到它有很多的 layer， 是一个拥有 10 层的镜像。这就是 docker 对文件系统的玩法，也是有效节省资源的方法。 所以一般我们的在一台机器上的镜像都会是树结构。



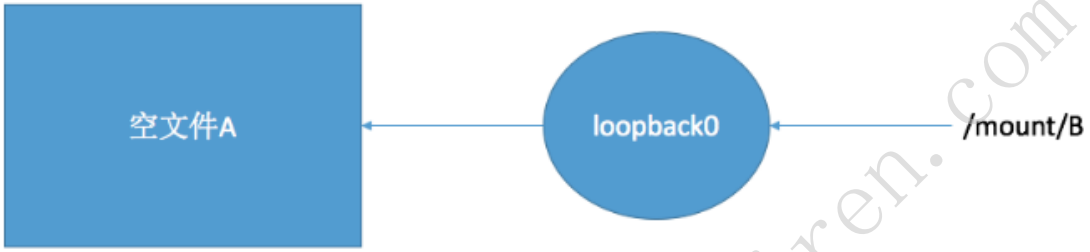
这也是我们的 dockerfile 里第一个指定就是 FROM。如下：

```
FROM selenium/node-chrome-debug:3.4.0

USER root

RUN apt-get update \
    && apt-get -y install ttf-wqy-microhei ttf-wqy-zenhei \
    && apt-get clean \
    && x11vnc -storepasswd work123 /home/seluser/.vnc/passwd
```

FROM 指令的意思就是从哪一级镜像层开始继承，也就是从哪一级镜像层开始联合。 RUN 指令会在镜像层的基础上运行。但并不对镜像层更改，而是创建一个容器层并记录对镜像层的更改。 最后当 RUN 指令运行完毕以后才会把容器层制作成新的一层镜像。 这里需要注意的是，一个 RUN 指令就是一层。 避免一个命令使用一个 RUN。 如果你用了 100 个 RUN 指定来跑 shell 的话， 会造成 100 层的镜像深度。这会变成一个噩梦的。 合理的利用镜像分层是 docker 使用者的必修课。

device mapper

刚才我们了解了 docker 的分层镜像系统和它的实现原理 aufs。 但其实 docker 的分层文件系统有很多的实现方式，aufs 只是其中一种。它是 docker 最早期实现的 storge driver，也是目前最稳定的存储驱动。 但它有一个很不爽的缺线就是 aufs 一直没能进入 linux 内核的主干。 也就导致了很多 linux 的发行版是不支持 aufs 的。 例如我们最常用的 centos。 早期的 docker 只能运行在 ubantu 上。据说是因为 aufs 的代码写的实在太烂，linux 的作者一直坚决的反对把 aufs 添加到内核主干来。 但是大家对于 docker 的使用越来越迫切， 这时候 linux 社区的大牛们出马使用 device mapper 支持了 docker 的分层镜像系统。 之后我还会介绍 overlayFS 和 overlayFS2， 但他们都是对 aufs 的模仿，实现的都是 docker 的分层镜像系统。所以我就不再具体介绍他们原理了。 这里先简单介绍一下 device mapper， linux 运维的同学一定对它不陌生，lvm 逻辑卷的主要实现技术。
通过快照的方式备份数据，以前很多数据库的热备都是通过类似这样的技术实现的。快照的特点也是只对原始数据修改的部分做记录，而不是真正的修改原始数据。 这一点跟 AUFS 一样，这样 device mapper 也就可以跟 aufs 一样去制作一个分层镜像系统。 device mapper 的优点是支持几乎所有 linux 的发行版本。它弥补了 aufs 只能在 ubantu 上运行的缺陷，让很多使用者可以很 happy 的在 centos 上安装并使用 docker。 但是它的缺点也是很令人诟病的。 device mapper
在构建自己的存储设备的时候是通过下面这个流程（图画的实在太 low 了请大家见谅）：



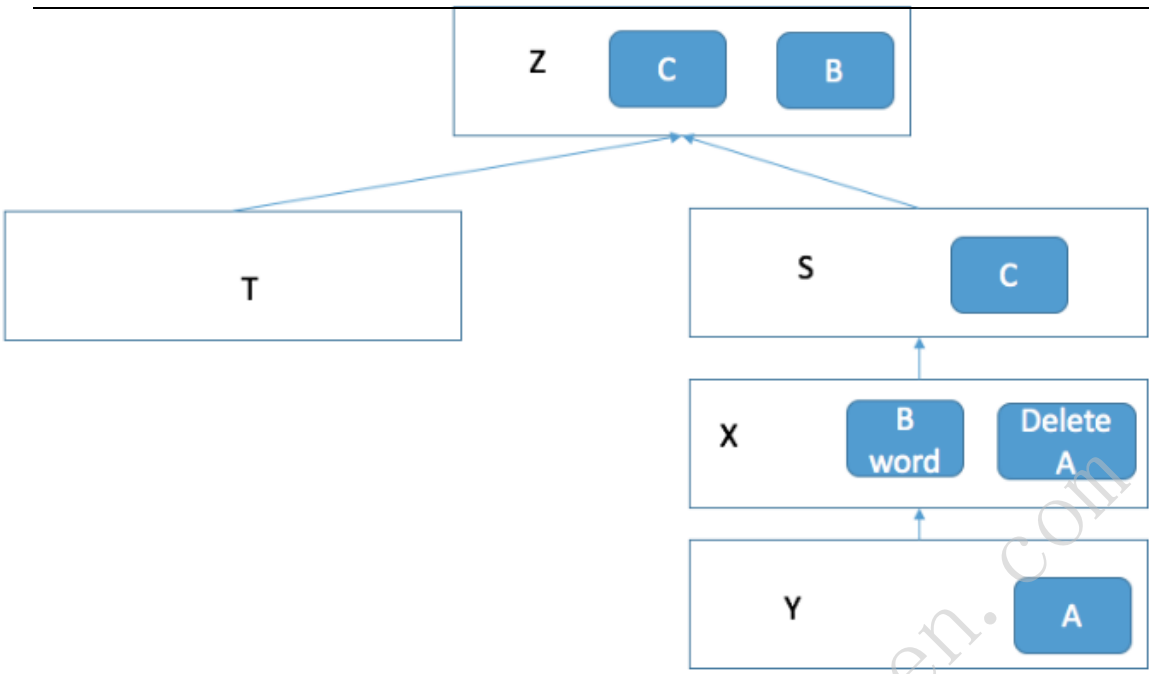
首先会先创建一个空文件 A(docker 默认创建一个 100G 的空文件), 这个文件有 100G 大，但实际上是一个空文件。
然后创建一个本地回环设备 loopback0， 这个设备的特点是可以关联这个空文件 A，并且可以把这个 loopback0 挂载出一个设备 B。
这样任何对 B 的改动都会通过 loopback0 保存在这个文件 A 中。 这就是 device mapper 做实际存储的方式
(时间有点久了具体细节有点记不起来)。 这种机制有两个主要的缺点

1. 一开始的那个空文件 A 是固定大小的，一旦创建没有任何扩容的可能。docker 默认创建的是 100G 大小的空间。如果随着使用的增加，我们的容器和镜像越来越多，100G 的硬盘空间极可能是不够的。如果发生了这种情况，没有任何办法。只能 rm -r /var/lib/docker 来解决，也就是整个把 docker 干掉再通过事先创建文件和回环设备的方式扩容。 我们曾经经历过这种痛，这也是我们放弃 device mapper 的原因之一。
2. 这种实现方式依赖本地回环设备进行 io 读写。 这种方式是不稳定的，在 google 查看过很多人的评论，都说 device mapper 的 io 不稳定，负载高。

还有一些其他的缺点我没有太关注，但是光这两个原因就足以让我放弃 device mapper 了。

overlayFS

由于大家对 device mapper 的失望，所以新一代的联合文件系统 overlayFS 问世。它可以说是 aufs 的升级版， 并且成功进入 linux 内核主干。 这里面有个小插曲，网上曾经流传着一封给 linux 作者的邮件，希望将 overlayFS 加入到 linux 中，
回复的邮件说中说虽然我也不喜欢 overlayFS，但它总比 aufs 好太多了，考虑到大多数人的需要，我决定把它加入到 linux 内核中来 (这是对 aufs 有多大的怨念啊)。 overlayFS 作为 aufs 的升级版，他们实现的功能是一样的，架构也很像。



还记得上面这个是 aufs 实现分层镜像系统的图么。 它是通过把所有的镜像层通过 aufs 联合到一个视图上来实现的。 但是 overlayFS 对待镜像层的方式不是这样的，它只有一层镜像，通过在镜像层使用硬链接的方式指向深层的镜像。
当然镜像层仍然是不可更改的，容器层会记录针对镜像层的更改。 这种方式造就了 overlayFS 的一个重大缺陷， 那就是 linux inode 的消耗。 我们知道每个 linux 系统的 inode 是有限的。 而软链接是会创建新的 inode，所以 overlayFS 选择了不会消耗 inode 的硬链接来实现。但是硬链接有一个缺点就是只能对文件做硬链接，而不能对目录做硬链接。 所以 overlayFS 在实现分层镜像系统的时候，是 copy 了镜像层的所有目录结构，然后再使用硬链接。 这就使得这些 copy 的目录都消耗了 inode。 所以对使用 overlayFS 的 docker 而言，一旦镜像和容器的结构变的复杂，就很可能触发 inode 耗尽的情况 (我们曾经遇见过)。 这个缺点直接导致了很多人弃用了 overlayFS（包括我）。所以 overlayFS2 问世了。

overlayFS2

overlayFS2 作为第三代的联合文件系统，在架构上优化了很多。 当然其中最主要的变化就是在镜像层次在 128 层以下的时候仍然使用像 aufs 的联合文件视图的方式，到了 128 层以上的时候才会使用硬链接。这样避免了 inode 耗尽的情况。 当然了 overlayFS2 刚刚问世不久，你需要把内核版本升级到 4.0 以上才能够加载 overlayFS2 的内核模块。 目前针对 overlayFS2 是最主流也是最稳定的。 推荐使用这个。

aufs on centos

我们刚才说了 linux 社区的大牛们极其的厌恶 aufs，它这辈子是没可能进入 linux 内核主干了。 甚至 linux 社区曾经宣布早晚有一天 ubantu 也要放弃 aufs。 那么这个 aufs on centos 又是什么鬼？ 因为很多公司要么有运维要求，要么对 centos 情有独钟。总之他们就是想用 centos 跑 docker，而且他们还不想用 device mapper 和 overlayFS 这俩坑爹货。他们还不敢随便尝试刚出来的 overlayFS2。对，说的就是像我这样的 。 所以网上有个好心人自己维护了一个内核版本，在 centos 上安装了这个内核版本之后，就可以使用 aufs 来运行 docker 了。

设置 storage driver

OK，再介绍了常用的这几种虚拟文件系统后，我们来看看怎么设置 docker 来使用这些文件系统。以 systemd 管理举例，编辑 /etc/systemd/system/docker.service.d/docker.conf
添加--storage-driver=aufs 来选择你需要的文件系统吧。

总结

这些就是我接触过的 docker 的 strange driver 并对他们做了一些比较。 并且也讲了一下容器的本质并非完整的操作系统。
它只是跟你玩了点小花招，让你以为是在一个容器就是一个隔离的完整的操作系统。
实际上我们还是在宿主机上，只是他通过联合文件系统给我们的错觉。联合文件系统通过展示给我们的视图来隔离了容器间的文件系统。但光隔离文件系统是不行的，我们的网络，进程都需要隔离。 所以下一篇我将介绍 docker 的网络原理。通过下一篇的讲解，大家应该就知道容器到底是什么了。