1. 服务器分区

**1.根目录**

根的数据确实在磁盘上，但根不在磁盘上，而是在内核自身中。内核在装载之前，是没有根的概念的。分区机制是软件提供的，文件系统也是软件提供的。调用bootloader从某个分区中将内核加载到内存完成初始化后，自己创造了1个根，然后挂载分区，从而使根作为访问其他分区的入口。

**2.和windows的区别**

Linux和windows都采用的是树型结构。最上层是根目录，其他的所有目录都是从根目录出发而生成的。在linux中，无论操作系统管理几个磁盘分区，目录树只有一个。而windows树型结构的根是磁盘分区的盘符，有几个分区就有几个树型结构，他们之间的关系是并列的。

**3.分区规划**

（1）默认安装下的分区



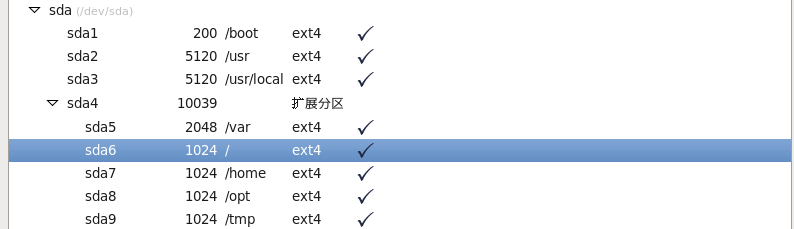
如果选择系统默认安装，会分为/和/boot两个分区。

（2）简单分区

只要分区/及swap即可，并预留一个备用的剩余磁盘容量作为将来练习。

（3）目录和分区的关系

在系统安装时自定义分区，需要指定分区的挂载目录。然后系统安装会将对应目录的文件安装到该分区下。



（4）服务器分区方案（安装时的下拉列表中就这些项）

将需要较大容量的目录、读写较为频繁的目录单独出来，这样如果出错不会影响到/分区。

1）/：为除了保存到其他分区下的目录之外的目录所需要的空间，FHS建议/所在分区应越小越好，通常2G以上，LVM格式。

2）/boot：100M即可。

3）/usr：默认安装程序目录，通常10G。

4）/usr/local：用户安装程序目录，通常10G以上。

5）/opt：附加应用程序，1G最多。

6）/home：通常每个用户100M即可，用户不多2G即可。

7）/tmp：最多1G即可。

8）/var：视服务器用途而定，剩下全部空间，一般2G左右。

8）/swap：通常分配和物理内存一样的空间，如果需要大的内存数据库，至少4G。

1. 编译安装

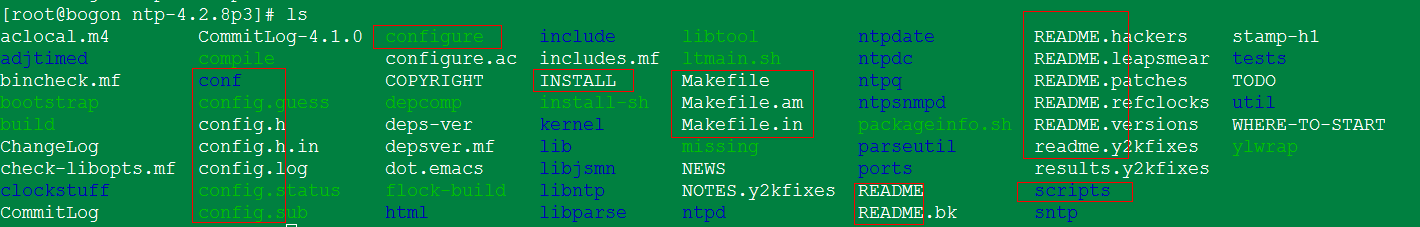
以安装npt为例。

**1.下载、解包**

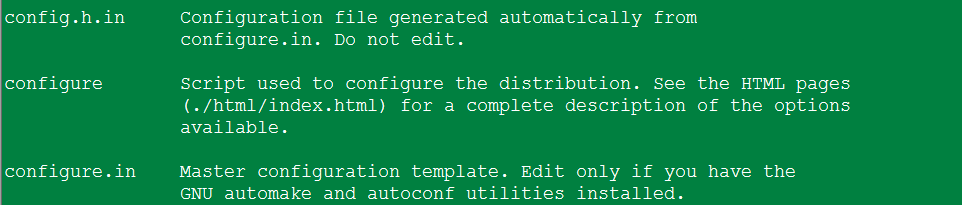
wget <http://linux.vbird.org/linux_basic/0520source/ntp-4.2.8p3.tar.gz>

tar -zxvf ntp-4.2.8p3.tar.gz

**2.文件列表**

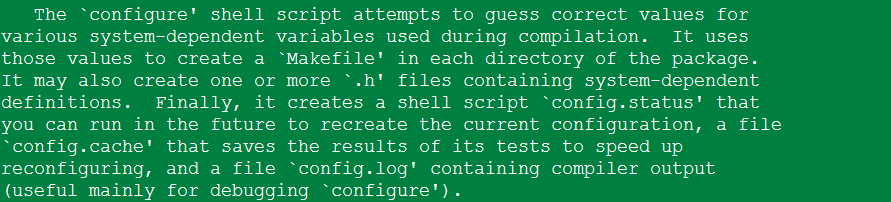


conf为源码配置，include为引用lib，scripts为一些脚本（也有源码）。README文件为介绍文件/目录内容：

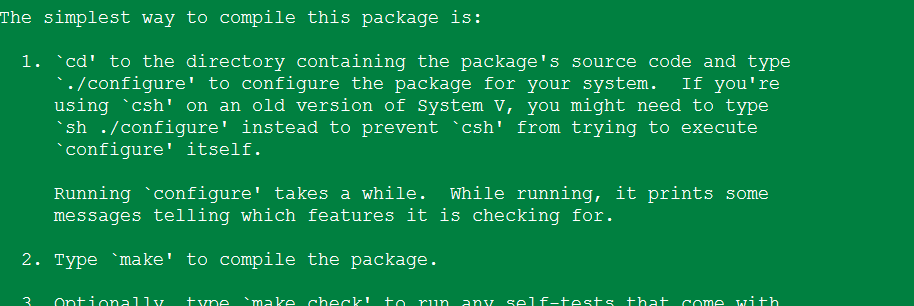


INSTALL为安装说明，里面有：

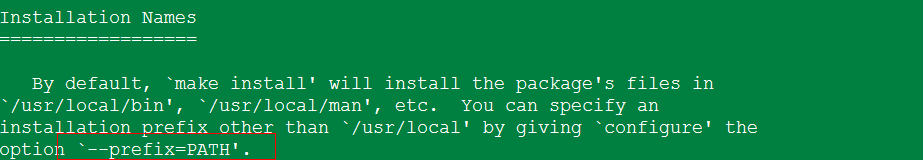
（1）configure作用说明

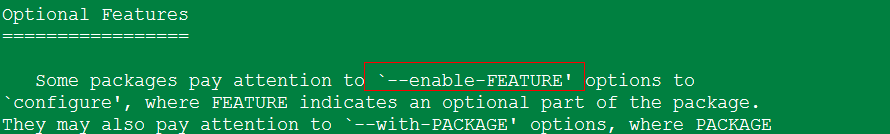


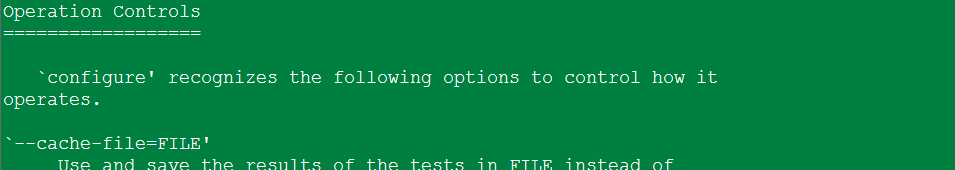
（2）编译说明



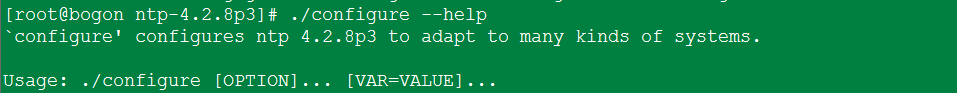
（3）Installation Names/ Optional Features/ Operation Controls等





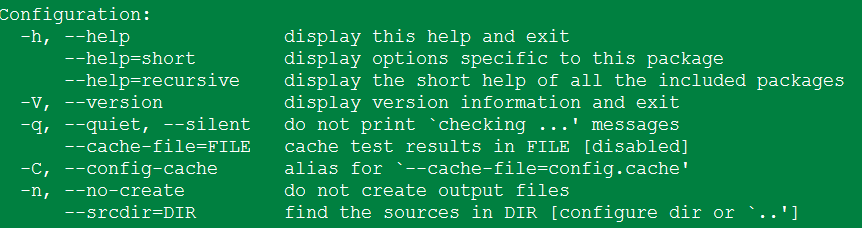


**3.查看configure帮助**

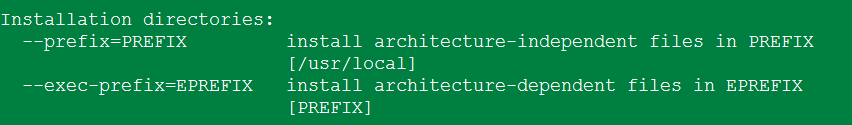


里面有README中的全部说明，包括：

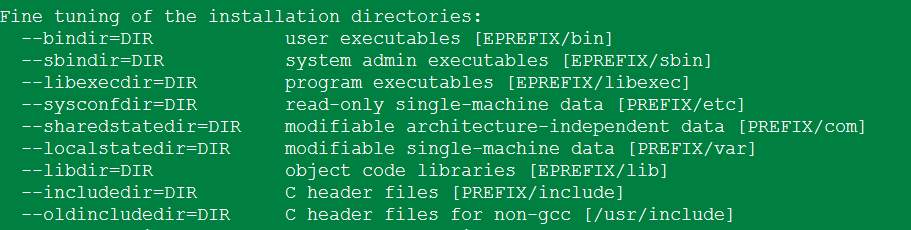
（1）配置



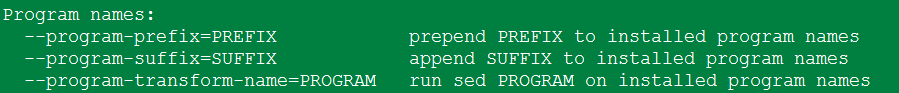
（2）安装目录



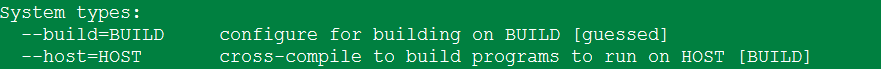
还有更细致目录控制



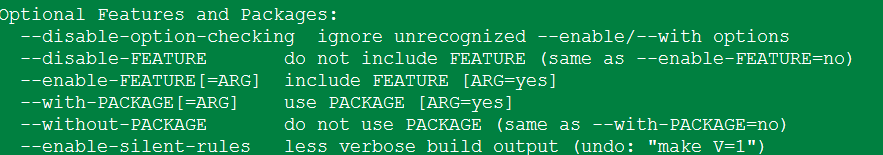
（3）程序名称设置



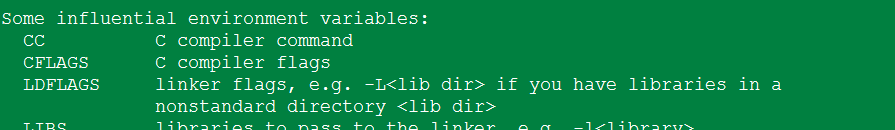
（4）系统类型



（5）功能控制



（6）环境变量



**4.正式安装**





1. PAM模块

PAM是一套应用程序接口，提供了一连串的验证机制，只要使用者将验证阶段的需求告知PAM后，PAM就能够回报使用者验证的结果（成功或失败）。PAM可以提供给其他程序调用。PAM用来进行验证的数据称为模块（Modules），每个PAM模块的功能都不太相同。

**1.PAM模块设置语法**

以passwd指令调用PAM为例，执行流程如下：

（1）使用者执行/usr/bin/passwd程序，输入密码。

（2）passwd调用PAM模块进行验证。

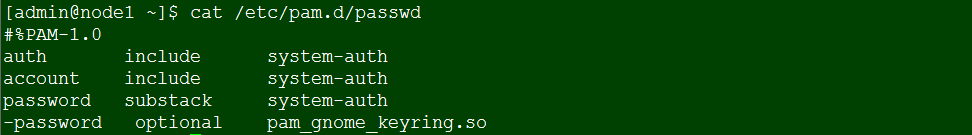
（3）PAM模块会到/etc/pam.d/找寻与程序（passwd）同名的配置文件。

（4）依据/etc/pam.d/passwd内的设置，引用相关的PAM模块逐步进行验证分析。

（5）将验证结果（成功、失败以及其他讯息）回传给passwd程序。

（6）passwd程序根据PAM回传结果决定下一个动作。

因此，/etc/pam.d/passwd中的配置最重要。



第1行#后为说明PAM版本，其余以#开头的均为注释。

每1行为1个独立的验证流程，分为3个字段，分别是验证类别（type）、控制标准（flag）、PAM的模块与该模块的参数。

1）验证类别

有如下4类：

auth：认证，主要用来检验使用者的身份验证，通常需要密码，后接模块是用来检验使用者的身份。

account：帐号，大部分是在进行authorization（授权），主要是在检验使用者是否具有正确的使用权限。

session：期间，session管理的就是使用者在这次登陆（或使用这个指令）期间，PAM所给予的环境设置。通常在记录使用者登陆与登出时的信息。如使用su或sudo指令时， /var/log/secure里会有相应pam的说明，而且记载的数据是session open、session close信息。

password：密码，主要在提供验证的修订工作，如修改/变更密码。

4种验证类型也通常按上述顺序进行。

2）验证的控制旗标

即验证通过的标准，管控该验证的放行方式，也有如下4类：

required：如果成功则带有success标志，失败则带有failure标志，但不论成功或失败都会继续后续的验证流程。由于后续的验证流程可以继续进行，通常用于日志记录。

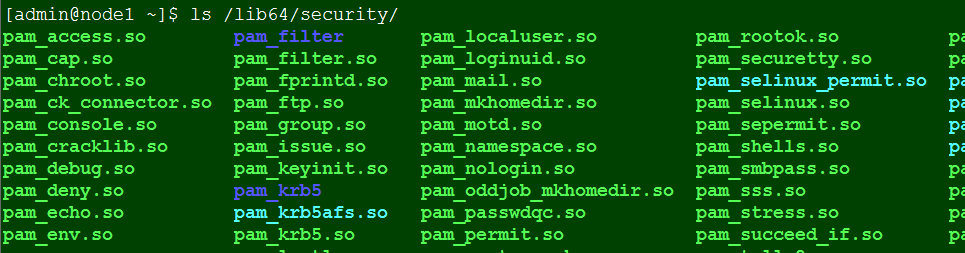
requisite：若验证失败则立刻回报原程序failure标志，并终止后续的验证流程。若验证成功则带有success标志并继续后续的验证流程。与required最大的区别在于失败时是否继续验证下去。

sufficient：若验证成功则立刻回传success给原程序，并终止后续的验证流程；若验证失败则带有failure标志并继续后续的验证流程。

optional：通常显示一些通知，并不是用于验证。

（2）常用模块简介

PAM模块放置在/lib64/security/下：



常用模块说明如下：

pam\_securetty.so：限制系统管理员只能够从安全的终端机登陆（tty）。

pam\_nologin.so：这个模块可以限制一般使用者是否能够登陆主机。

pam\_selinux.so：SELinux是个针对程序设置管理权限的功能。

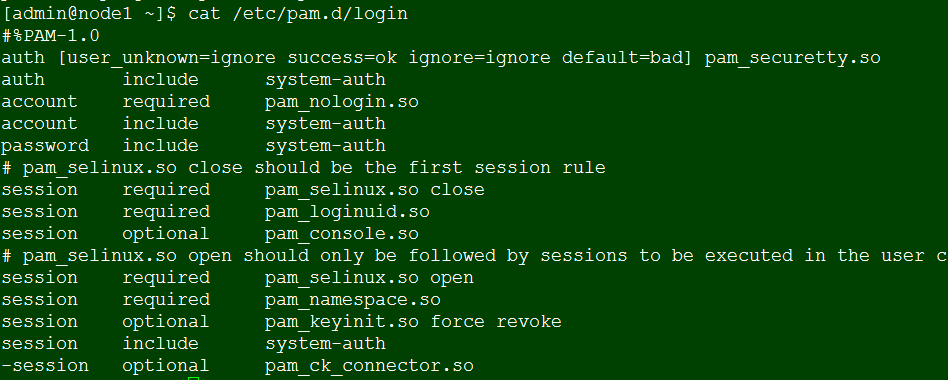
pam\_console.so：当系统出现某些问题，让使用者可以通过特殊终端接口顺利的登陆系统。

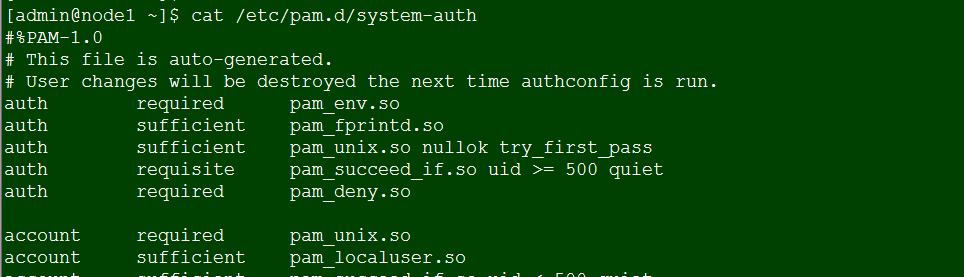
pam\_env.so：用来设置环境变量的1个模块。

pam\_unix.so：很复杂且重要的1个模块，可以用在验证阶段的认证功能，可以用在授权阶段的帐号授权管理，可以用在会议阶段的登录文件记录等。

pam\_pwquality.so：用来检验密码的强度。

以login为例，其PAM验证机制流程如下：





1）验证阶段

先通过pam\_securetty.so判断是否root。再通过pam\_env.so设置额外的环境变量。再通过pam\_unix.so检验密码。继续往下以pam\_succeed\_if.so判断UID是否大于1000。

2）授权阶段

先以pam\_nologin.so判断/etc/nologin是否存在，若存在则不许一般使用者登陆。再以pam\_unix.so及pam\_localuser.so进行帐号管理。再以pam\_succeed\_if.so判断UID是否小于1000，若小于1000则不记录登录信息。最后以pam\_permit.so允许该帐号登陆。

3）密码阶段

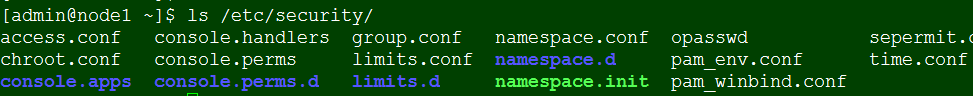
先以pam\_pwquality.so设置密码仅能尝试错误3次。再以pam\_unix.so通过sha512、shadow等功能进行密码检验。

4）会议阶段

先以pam\_selinux.so暂时关闭SELinux。再使用pam\_limits.so设置好使用者能够操作的系统资源。再以pam\_loginuid.so规范不同的UID权限等。

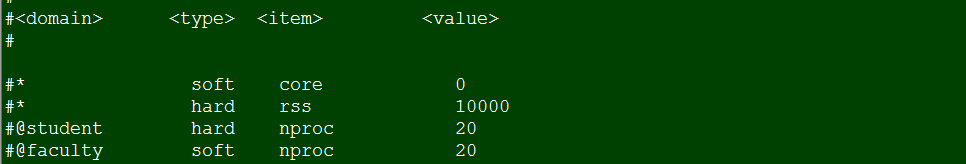
6.其他相关文件

其他PAM相关的文件主要在/etc/security目录下。



（1）limits.conf

对各个用户进行限制，包括一些文件大小、登录次数等。



（2）/var/log/secure、/var/log/messages

登录日志。

1. docker

参考

http://blog.csdn.net/chengly0129/article/details/68947265

https://cr.console.aliyun.com/?spm=5176.1971733.2.28.2fa25aaaxFMFBu#/accelerator

* 1. 概要

Docker系统包括服务端和客户端。其中，服务端是一个服务进程，管理着所有的容器。客户端则扮演着docker服务端的远程控制器，用来控制docker的服务端进程。通常服务端和客户端运行在一台机器上。

Docker包括三个基本概念：镜像、容器、仓库。

* + 1. 镜像

（1）操作系统可分为内核和用户空间。对于Linux而言，内核启动后，会挂载root文件系统为其提供用户空间支持。Docker镜像就相当于是一个root文件系统。如ubuntu官方镜像就包含了完整的一套Ubuntu1最小系统的root文件系统。

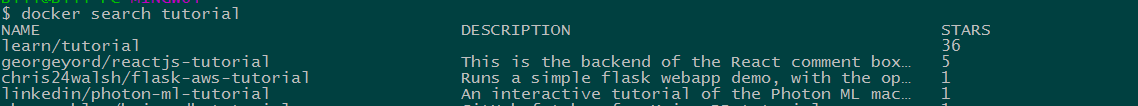
（2）Docker镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

（3）Docker设计时，就充分利用UnionFS的技术，将文件系统设计为分层存储的架构。镜像构建时，会一层层构建。前一层是后一层的基础。每一层构建完就不会再发生改变。后一层上的任何改变只发生在自己这一层。如删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。

所以严格来说，镜像并非是像一个ISO那样的打包文件，镜像只是一个虚拟的概念。它并非由一个文件组成，而是由多层文件系统联合组成。

（4）分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

（5）官网index.docker.io页面展示所有可用的镜像，可通过search+镜像名检索。



镜像通常按“用户名/镜像名”存储。对于如ubuntu这类基础镜像，经过官方验证值得信任，可以直接用镜像名来检索。

* + 1. 容器

（1）镜像和容器的关系，就像面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

（2）容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的命名空间。因此容器可以拥有自己的root文件系统、网络配置、进程空间，甚至用户ID空间。容器内的进程是运行在一个隔离的环境里。

（3）容器也是采用分层存储设计。每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层。容器存储层用于为容器运行时的读写操作而准备。容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。按照Docker最佳实践的要求，容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用数据卷或绑定宿主目录，他们的的生存周期独立于容器，容器消亡，数据卷不会消亡。

* + 1. Docker Registry

（1）Docker Registry用于集中存储、分发镜像。一个Docker Registry中可以包含多个仓库。每个仓库可以包含多个标签。每个标签对应一个镜像。可以通过<仓库名>:<标签>的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以latest作为默认标签。

（2）配置国内加速器

执行docker info，查看RegistryMirrors的值是否是自己配置的。

* 1. 使用

（3）获取镜像



默认仓库地址是DockerHub。

（4）从下载过程中可以看到我们之前提及的分层存储的概念，镜像是由多层存储所构成。下载也是一层层的去下载，并非单一文件。下载过程中给出了每一层的ID的前12位。并且下载结束后，给出该镜像完整的sha256的摘要，以确保下载一致性。



（5）运行

以镜像为基础启动并运行一个容器。

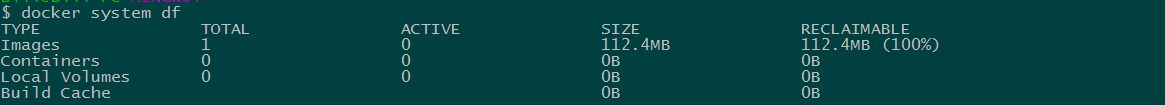


（6）列出镜像

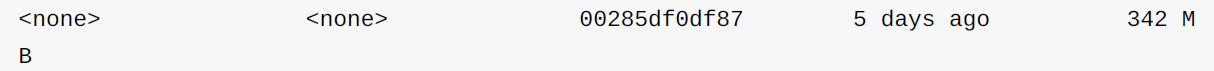


DockerHub中显示的体积是压缩后的体积。这里的是解压后的大小。

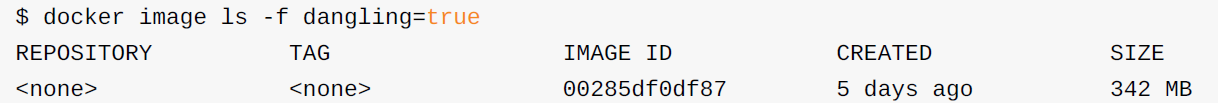
列表中的镜像体积总和并非是所有镜像实际硬盘消耗。由于Docker镜像是多层存储结构，并且可以继承、复用，因此不同镜像可能会因为使用相同的基础镜像，从而拥有共同的层。可以通过以下命令来便捷的查看镜像、容器、数据卷所占用的空间。



（7）有一种特殊的镜像，既没有仓库名，也没有标签，均为<none>。

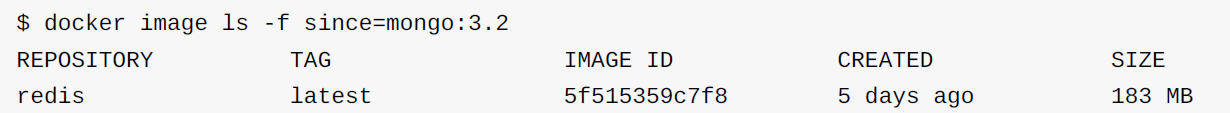


出现这种情况是因为由于新旧镜像同名，旧镜像名称被取消导致，可以直接删除。可以用下面的命令专门显示这类镜像：



（8）为了加速镜像构建、重复利用资源，Docker使用了中间层镜像。默认的dockerimagels列表中只会显示顶层镜像，如果希望显示包括中间层镜像在内的所有镜像的话，需要加-a参数。

（9）ls后可以进行路径匹配，也可以使用--filter或-f过滤。



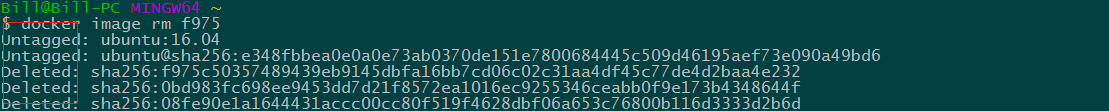
since/before：之后、之前。

（10）-q：只显示镜像对于的ID列表。

（11）删除本地镜像



其中，<镜像>可以是镜像短ID、镜像长ID、镜像名或者镜像摘要。



删除行为分为两类：Untagged和Deleted。删除镜像实际是删除某个标签的镜像。因为一个镜像可以对应多个标签，因此首先取消所指定的所有镜像标签，但可能还有别的标签指向了这个镜像。只要当该镜像所有的标签都被取消了，该镜像很可能会失去了存在的意义，因此会触发删除Deleted行为。

（12）利用commit理解镜像构成

不要使用docker commit定制镜像，定制镜像应该使用Dockerfile来完成。

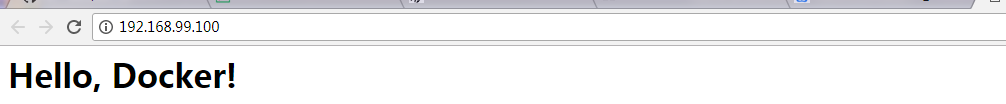
如定制1个Web服务器。如下，如果本地没有nginx，会自动从仓库中pull。



使用exec进入容器，使用如下内容写入欢迎页。



此时，服务器欢迎页改为如下：

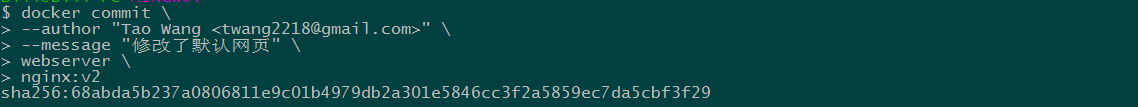


使用diff命令查看具体改动。

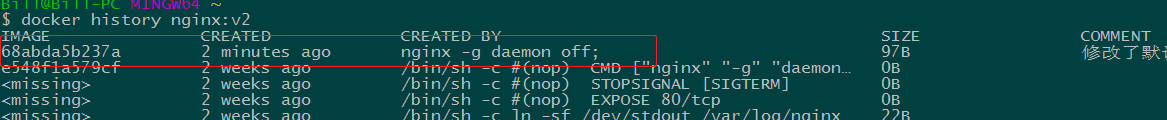


目前所做的任何文件修改都记录于容器存储层里。可commit命令，将容器的存储层保存下来成为镜像。





可以用dockerhistory具体查看镜像内的历史记录。如下，可以查看到刚才修改的那一层。



commit存在严重的问题：1）所有镜像的操作都是黑箱操作，生成的镜像也被称为黑箱镜像，很难了解执行过什么命令。2）没制作1次镜像，就更加臃肿，即使后面删除一些东西，因为分层，实际并没有删除。

**操作Docker容器**

（1）启动容器

（1）使用run命令。参数-t让Docker分配一个伪终端并绑定到容器的标准输入上。-i则让容器的标准输入保持打开。-d让Docker在后台运行而不是直接把执行命令的结果输出在当前宿主机下。

（2）可以利用docker container start命令，直接将一个已经终止的容器启动运行。

（3）可以通过docker container ls命令来查看容器信息。

（4）可以通过docker container logs命令。获取容器的输出信息，

（5）可以使用docker container stop来终止一个运行中的容器。终止状态的容器可以用docker container ls -a命令看到。

（6）docker container restart重启容器。

（7）使用docker exec进入容器进行操作。使用-i和-t参数一起使用时，则可以看到我们熟悉的Linux命令提示符。

（8）使用docker export命令导出本地某个容器快照到本地文件。使用docker import从容器快照文件中再导入为镜像。

（9）使用docker container rm来删除一个处于终止状态的容器。使用docker container prune清理掉所有处于终止状态的容器。

**访问仓库**

（1）注册服务器是管理仓库的具体服务器，每个服务器上可以有多个仓库，而每个仓库下面有多个镜像。仓库可以被认为是一个具体的项目或目录。

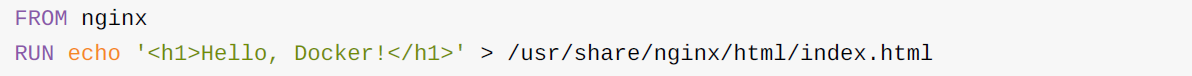
（2）通过执行docker login命令交互式的输入用户名及密码来完成在命令行界面登录DockerHub。可以通过docker logout退出登录。

（3）可以通过docker search命令来查找官方仓库中的镜像，并利用docker pull命令来将它下载到本地。

（4）通过docker push命令来将自己的镜像推送到DockerHub。

* 1. 使用Dockerfile定制镜像

Dockerfile里的每一条指令描述如何构建一层。



* + 1. 指令

**1.FROM**

指定基础镜像。如果为scratch，则表示一个空白的镜像。

**2.RUN**

（1）执行命令行命令。有shell格式和exec格式两种格式。



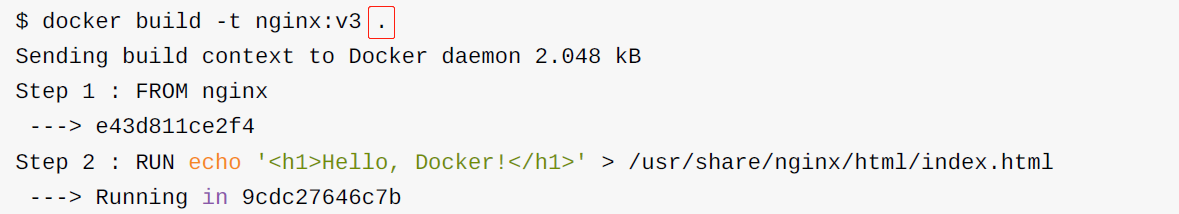


每一个RUN代表新建立一层，应该尽量减小层级，如下：



在使用Dockerfile构建镜像命令：





（14）镜像构建上下文

docker分为服务端守护进程和客户端工具两部分。为了让服务端获得本地文件，就需要设置镜像构建上下文。如下：



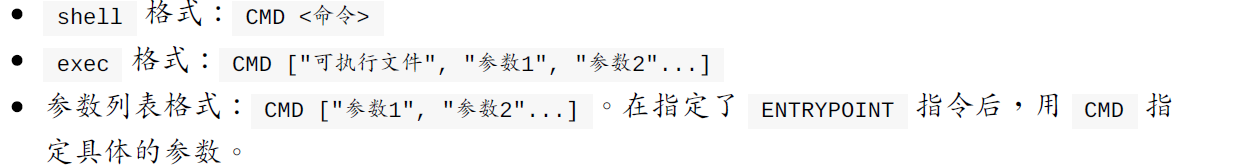
相对的的是上下文的路径，都是相对路径，不能是绝对路径。因此，还需要.dockerignore来剔除不需要作为上下文传递给Docker引擎的文件。如果不额外指定Dockerfile的话，会将上下文目录下的名为Dockerfile的文件作为Dockerfile。

（15）Dockerfile的其他命令



ADD指令在COPY基础上增加了一些功能。

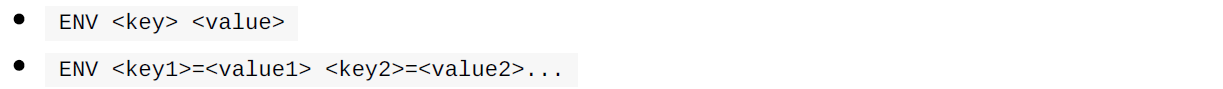
CMD指令的格式和RUN相似，也是两种格式：



Docker不是虚拟机，容器就是进程。既然是进程，那么在启动容器的时候，需要指定所运行的程序及参数。CMD指令就是用于指定默认的容器主进程的启动命令的。

Docker不是虚拟机，容器中的应用都应该以前台执行。

ENV设置环境变量



EXPOSE声明端口



USER指定当前用户

（16）保存镜像

使用docker save命令可以将镜像保存为归档文件。



之后，在另一个机器上就可以如下加载镜像。

