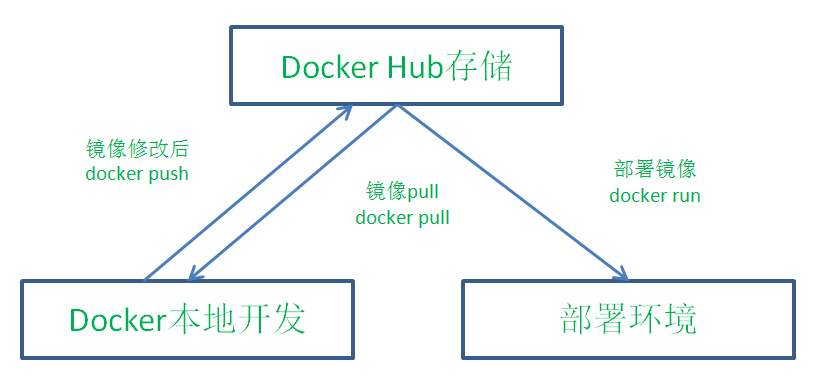
第七章 Docker实战

我们已经知道Docker是个轻量级虚拟机，那么一台虚拟机脱离不了运行环境，而这个运行环境或者说载体就是Docker 的镜像，我们先看看docker镜像的使用环境：



从上图我们可以看到，Docker Hub是用来存储Docker镜像的（详见第x章），Docker本地开发环境是用来做开发镜像用的，一般的开发流程是先从Docker Hub中获取到基础镜像，之后修改使新的镜像满足一定的需求或提供某种服务，然后就可以push到Docker Hub中，而本地也无需保留镜像的备份（相关的源码，Dockerfile还是需要留的）。这样开发工作就完成了，部署的时候直接将开发好的镜像pull下来，然后直接通过docker run命令部署即可，或者借助其他部署工具（比如docker-compose详见后续章节）管理部署镜像。

在本章，前两节首先向读者介绍Docker镜像容器相关的基本命令，后续章节将创建一个基于Web服务器的工程，并为该Web前端挂上后台服务器，后台也是个有几个独立的模块组成，会通过docker-compose来同时管理几个容器。读者可以亲身体验下Docker开发流程项目组织等。

## Docker基本操作

镜像是Docker启动容器的基础，也是应用发布的标准格式。Docker用户能够在任意主机上获取到特定的镜像，并利用该镜像启动容器运行指定服务。Docker支持很多对镜像的操作命令来提供对镜像的灵活处理，包括镜像的制作、查看、重命名、导入、导出、删除等。下面介绍docker对镜像操作的命令。

* build

docker build命令使用指定路径的源码构建一个新的镜像。

Usage: docker build [OPTIONS] PATH | URL | -

docker build命令从指定的路径获取Dockerfile，然后根据Dockerfile的内容制作镜像。Dockerfile的每一行对应一条操作指令，会生成镜像的一层。docker build制作镜像时，会将当前目录下所有内容发送到docker daemon。因此，制作镜像时要删除无用的文件，尽量保证工作目录整洁。如果URL设置为Dockerfile或者是指向压缩包的URL，docker client不会向daemon发送数据内容；如果设置为Git仓库地址，docker会先把指定Git仓库克隆到本地，然后将内容发送到docker daemon。

build命令支持以下参数：

* -f,--file=PATH/Dockerfile

指定Dockerfile，可以是本地路径（绝对路径或相对路径），也可以设置为一个指向远端tarball或Git仓库的URL。不论哪种方式，构建的内容中必须包含Dockerfile文件。

* --force-rm=true|false

不论镜像制作成功与否，结束后删除制作过程中生成的中间容器，默认值为false。

* --no-cache=true|false

制作镜像时不使用缓存，默认值为false。

* --pull=true|false

制作镜像时尝试获取新版本的镜像，默认值为false。

* -q, --quiet=true|false

忽略镜像制作过程中的输出打印，默认值为false。如果不希望终端打印内容过多，可以使能该选项。

* --rm=true|false

镜像制作成功后删除制作过程中生成的中间容器，默认值为true。

* -t, --tag=””

为制作的镜像指定名字和Tag，如果该名字已经存在会覆盖原有镜像。

* -m, --memory=MEMORY

设置镜像内存使用限制， 默认使用系统所有内存。

* --memory-swap=MEMORY-SWAP

设置内存总量，包括内存和交换分区。如果设置为-1，表示禁用交换分区。

* -c, --cpu-shares=0

设置容器（运行该镜像启动的容器）占用CPU时间的比重。默认情况下，所有容器占用相同的CPU时钟比重，--cpu-shares参数用于修改当前容器相对其他所有运行容器的比重，设置范围为0-1024。如果当前系统运行三个容器C1、C2、C3，其cpu-shares分别设置为1024、512、512，那么他们占用的CPU时钟比重分别是50%、25%、25%。

注意：所设置的比重只有在容器运行CPU密集型进程时才会生效。如果一个容器执行任务为空，不论它占用的CPU比重是多少，剩余的容器会使用所有cpu时钟。在多核系统中，CPU时间是在多核之间分配的；即便容器占用CPU比重少于100%，也有可能占用某一个核100%的CPU时钟。

* --cpu-period=0

限制容器运行时的CPU CFS（Completely Fair Scheduler，完全公平调度）周期，内核会根据该参数的内容限制容器的CPU使用率。

* --cpu-quota=0

设置容器运行时的CPU CFS限额，默认占用所有的CPU资源。

* --cpuset-cpus=CPUSET-CPUS

设置允许容器运行的CPU，可以通过(0-2, 0,1,2)的形式设置。

* --cpuset-mems=CPUSET-MEMS

设置NUMA系统中允许容器使用的内存节点，可以通过(0-2, 0,1,2)的形式设置。

* --cgroup-parent=CGROUP-PARENT

设置创建容器cgroup的路径。如果使用相对路径，是指相对系统init进程的cgroups目录的路径，而非当前目录。

* import

docker import命令创建一个空的文件系统镜像，并将打包文件的内容导入。docker import命令生成的镜像不保留原有打包文件的分层结构。

Usage: docker import [OPTIONS] URL|- [REPOSITORY[:TAG]]

使用docker import命令导入文件时，可以使用如下方式为制作的镜像指定tag：

cat example.tar | docker import – example:test

说明：docker import命令导入文件支持的格式包括：

“.tar”，”.tar.gz”，”.tgz”，”.bzip”，”.tar.xz”，”.txz”

import命令支持以下参数：

* -c, --change=[]

导入镜像时使用Dockerfile指令修改镜像属性，如可以使用如下方式设置镜像的http\_proxy环境变量：

cat example.tar | docker import -c “ENV http\_proxy [example@example.com](mailto:example@example.com)” – example:test

说明： change参数可以使用的指令包括：

“CMD”，”ENTRYPOINT”，”ENV”，”EXPOSE”，”ONBUILD”，”USER”，”VOLUME”，”WORKDIR”

* load

docker load命令从指定的 tar文件或标准输入装载镜像。

Usage: docker load [OPTIONS]

docker load命令所装载文件中包含的镜像信息会恢复到新镜像中。

load命令支持以下参数：

* -i, --input=””

使用linux tar文件作为输入，input参数用于指定tar文件的路径。

* pull

docker pull命令用于从registry中获取镜像或仓库。

Usage: docker pull [OPTIONS] NAME[:TAG|@DIGEST]

pull命令支持以下参数：

* -a, --all-tags=false

获取指定仓库中的所有镜像，默认值为false，只获取tag为latest的镜像。

docker pull命令在没有显式指定REGISTRY\_HOST时，会默认从Docker的公有registry地址registry-1.docker.io获取镜像；也可以在pull的过程中指定私有的registry，下面例子表示从registry-host主机的5000端口运行的registry服务上获取镜像：

docker pull registry-host:5000/official/busybox:latest

* images

docker images命令用来列出本地docker仓库存储的镜像列表。

Usage: docker images [OPTIONS] [REPOSITORY]

默认情况下，docker images命令列出的镜像中不包括制作镜像过程中生成的中间镜像。列出的信息包括仓库（镜像名）、tag、镜像ID、创建日期和大小。

images命令支持以下参数：

* -a, --all=false

显示本地镜像仓库的所有镜像，包括中间镜像，默认值为false。

* --digests=false

显示镜像摘要信息，默认值为false。

* -f, --filter=[]

对输出内容进行过滤，如”-f=dangling=true”会过滤出所有无用的镜像；”-f=label=com.foo=amd64”会过滤出所有com.foo标签为amd64的镜像。

* --no-trunc=false

输出信息不做截断处理，信息更完整，但是会使打印信息变得混乱，默认值为false。

* -q, --quiet=false

只显示镜像ID，默认值为false。

* tag

docker tag 命令将镜像标记成仓库。

Usage: docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

docker tag命令为指定registry中的镜像重新命名，其中REGISTRYHOST代表registry的地址，使用”HOSTNAME:PORT”这种形式；USERNAME为镜像的所有者；NAME为镜像名；TAG是镜像的标记，用来区分具有相同镜像名的不同版本的镜像。

tag命令支持以下参数：

* -f, --force=true|false

强制执行命名操作，默认值为false。

* push

docker push命令用于将镜像上传到registry。

Usage: docker push [OPTIONS] NAME[:TAG]

使用push命令上传镜像时，如果不指定REGISTRY\_HOST，就会把镜像上传到Docker的公有registry “registry-1.docker.io”中。也可以通过docker tag命令将镜像标记成相应私有registry的镜像，然后执行docker push将该镜像上传到私有registry。以registry-host:5000为例，下例将busybox镜像上传到registry-host上5000端口运行的私有registry中：

# docker tag busybox:latest registry-host:5000/myadmin/busybox:latest

# docker push registry-host:5000/myadmin/busybox:latest

* save

docker save命令将镜像保存成tar文件归档。

Usage: docker save [OPTIONS] IMAGE [IMAGE...]

使用docker save命令归档镜像文件，默认输出到stdout，也可以输出到”—output”参数指定的输出文件中。Save命令生成的tar文件中包含镜像所有的层级信息和Tag及Version信息。

如果不指定镜像tag，docker save会把本地仓库所有对应镜像都进行归档；如果指定仓库和tag只归档特定的镜像。如下例所示，第一条命令会将本地busybox对应的所有tag的镜像都打包到busybox-all.tar文件中；第二条命令只打包busybox:latest镜像：

# docker save busybox > busybox-all.tar

# docker save --output=busybox-latest.tar busybox:latest

save命令支持以下参数：

* -o, --output=””

将镜像内容归档到指定的文件中。

## Docker容器操作

容器是docker运行的实例，docker为容器管理提供很多命令和参数，使用户能够以很灵活的方式对容器进行创建、查询、修改和删除等操作。本节对docker中的容器管理命令进行介绍。

* run

docker run命令用于启动一个新的容器并运行指定命令。

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

docker run命令启动一个新的容器，并在容器中启动一个新的进程。如果指定的镜像文件不存在，docker run会像docker pull一样先获取指定镜像。

run命令支持以下参数：

* -a, --attach=[]

连接容器的stdin、stdout或stderr。在不使用-d参数启动容器时，run命令默认会启动容器中的进程并将终端连接到进程的stdin、stdout、和stderr。使用-a参数可以指定三者中的任意一个。

* --add-host=[]

向容器的/etc/hosts文件中增加一个host到IP的映射对，可以在容器中通过host访问IP。

* --blkio-weight=0

设置block IO权重（相对权重），赋值范围为10-1000。

* -c, --cpu-shares=0

设置容器占用CPU时间的比重。默认情况下，所有容器占用相同的CPU时钟比重，--cpu-shares参数用于修改当前容器相对其他所有运行容器的比重，设置范围为0-1024。如果当前系统运行三个容器C1、C2、C3，其cpu-shares分别设置为1024、512、512，那么他们占用的CPU时钟比重分别是50%、25%、25%。

注意：所设置的比重只有在容器运行CPU密集型进程时才会生效。如果一个容器执行任务为空，不论它占用的CPU比重是多少，剩余的容器会使用所有cpu时钟。在多核系统中，CPU时间是在多核之间分配的；即便容器占用CPU比重少于100%，也有可能占用某一个核100%的CPU时钟。

* --cap-add=[]

向容器添加指定的linux capabilities。

* --cap-drop=[]

从容器中删除指定linux capabilities。

* --cid-file=””

将容器ID写入到指定文件中。

* --cgroup-parent=””

设置创建容器cgroup的路径。

注意： 如果使用相对路径，是指相对系统init进程的cgroups目录的路径，而不是相对当前目录。

* --cpu-period=0

限制容器运行时的CPU 调度周期，内核会根据该参数限制容器的CPU使用率。

* --cpu-quota=0

设置容器运行时的CPU 调度限额，默认会占用所有的CPU资源，可以通过该参数设置容器的CPU占用率。

* --cpuset-cpus=””

设置允许容器运行的CPU，可以通过(0-2, 0,1,2)的形式设置。

* --cpuset-mems=””

设置NUMA系统中允许容器使用的内存节点，可以通过(0-2, 0,1,2)的形式设置。

* -d, --detach=true|false

使能后台模式，在后台运行容器并向终端打印容器ID，默认值为false。

说明： 使用后台模式启动容器后，可以使用docker ps命令查看运行中的容器，并使用docker attach命令重新登录到该容器中。在连接模式下，可以使用键盘CTRL-P+CTRL-Q键从运行的容器中退出，且不会影响容器中运行的进程。使用后台模式时，不能够使用--rm选项。

* --device=[]

将指定的主机设备添加到容器中。

* --dns-search=[]

设定容器的DNS搜索域。

* --dns=[]

设置容器的DNS服务器。

* -e, --env=[]

设定容器的环境变量。

* --entrypoint=””

指定容器运行的ENTRYPOINT，该参数会覆盖镜像文件中设置的入口命令。

* --env-file=[]

从指定文件中获取容器环境变量。

* --expose=[]

将容器中的一个或一系列端口暴露到主机中，使用啊“--expose=5000-5500“这种形式来暴露连续端口。

* --group-add=[]

向容器中添加额外的用户组运行进程。

* -h, --hostname=””

设置容器的主机名，该主机名在容器内部有效。

* -i, --interactive=true|false

连接容器的标准输入，默认值为false。

* --ipc=””

指定容器的IPC策略，默认会创建新的私有IPC命名空间。可选的IPC策略包括：

1. “container:<name|id>”：重用其他容器的共享内存，信号量和消息队列。
2. “host”：使用主机的共享内存，信号量和消息队列；这种模式下容器可以访问主机所有的共享内存，会影响主机其他进程或容器的安全性，不建议使用。

* -l, --label=””

为容器添加标签，用于完善容器元数据信息。

* --label-file=[]

从指定文件中加载label信息，每一个label以EOL结尾。

* --link=[]

将容器与指定容器建立连接，使用“<name|id>:alias”的方式指定新的连接。在新容器中可以通过别名来访问建立连接的容器。

* --lxc-conf=[]

从指定文件中读取lxc配置，作为lxc驱动参数，仅在使用lxc驱动时有效。

* --log-driver=json-file|syslog|journald|gelf|fluent|none

指定容器的日志驱动器，默认是用docker daemon指定的驱动方式；docker logs命令仅在json-file日志驱动模式下有效。

* --log-opt=[]

日志驱动选项，用于配置日志驱动的参数。

* -m, --memory=””

设置容器使用内存的上限，可以指定单位b、k、m、g。如果主机支持内存交换，内存设置值可以大于主机物理内存大小，默认情况下内存不做限制。

* --memory-swap=””

设置内存总量，包括内存和交换分区的总和。如果设置为-1，表示禁用交换分区。

* --mac-address=””

设置容器的MAC地址。

* --name=””

设置容器名。

* --net=””

设置容器的网络模式，包括以下四种模式：

1. “bridge”: 在docker网桥上创建新的网络栈
2. “none”: 不设置网络
3. “container:<name|id>”: 重用其他容器的网络栈
4. “host”: 使用主机的网络栈，容器可以访问主机所有的本地网络服务，影响主机业务安全性，不建议使用。

* --oom-kill-disable=true|false

禁用容器的内存耗尽时停止进程功能。

* -P, --publish-all=true|false

将容器中的所有端口暴露到主机的随机端口上，默认值为false。

* -p, --publish=[]

将容器的一个端口或端口段发布到主机上，可以使用如下格式来指定端口的映射关系：

format: ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort | containerPort

说明： 可以使用docker port命令查看容器的实际端口映射。

* --pid=host

设置容器的pid模式。如果设置为host模式，容器能够访问主机的PID namespace，会影响主机安全性。

* --uts=host

设置容器的UTS模式，如果设置为host模式，容器能够访问主机的UTS namespace，会影响主机安全性。

* --privileged=true|false

为容器分配额外权限，默认为false。使能该参数后，容器可以访问到主机上所有的设备，以及修改主机的安全配置，不建议使用该参数。

* --read-only=true|false

以只读的方式挂载容器的根文件系统。

* --restart=”no”

设置容器退出时的处理策略，可以设置为no、on-failure[:max-retry]、always。

* --rm=true|false

当容器内进程执行结束后自动删除容器，与-d参数不能互相兼容，默认值为false。

* --security-opt=[]

设置容器的安全策略，包括以下设置项：

1. “label:user:USER”：设置容器的安全标签用户。
2. “label:role:ROLE”：设置容器的安全标签角色。
3. “label:type:TYPE”：设置容器的安全标签类型。
4. “label:level:LEVEL”：设置容器的安全标签级别。
5. “label:disable”：关闭容器的安全标签限制。

--sig-proxy=true|false

将接受到的信号转发到容器中的进程，对非TTY模式有效，转发的信号不包括SIGCHLD、SIGKILL、SIGSTOP信号。默认值为true。

* --memory-swappiness=””

调整容器的内存交换行为，赋值范围为0-100。

* -t, --tty=true|false

为容器分配一个虚拟终端，默认值为false。

* -u, --user=””

使用指定的用户或用户组执行容器中命令，可以使用如下格式：

--user [user | user:group | uid | uid:gid | user:gid | uid:group ]

如果不指定，会默认使用容器的root用户执行该命令。

* --ulimit=[]

设置容器ulimit。

* -v, --volume=[]

为容器挂载数据卷，可以多次指定-v参数为容器挂载多个数据卷。挂载的数据卷可以通过”:rw”和”:rw”后缀来指定只读或读写模式，默认情况下使用读写模式。

在SELinux这样的标签系统中，需要为挂载的卷设置适当的标签才能够访问卷上的数据。默认情况下，docker不会修改操作系统设置的标签；如果需要修改标签内容，可以在挂载数据卷时加上”:z”或”:Z”后缀；加上该标记后docker会为数据卷上的文件重新加载标签。“z”表示会在容器间共享数据卷的内容；”Z”表示非共享模式，只有当前容器能够访问数据卷的内容。

挂载数据卷时，多种模式之间可以使用”,”隔开。

* --volumes-from=[]

将指定容器中挂载的数据卷重新挂载到当前容器中。不论指定容器是否处于运行状态，都可以使用”volumes-from”参数挂载数据卷。默认会按照数据卷原有的模式挂载，也可以通过添加后缀的方式来修改挂载模式。当数据卷与容器中路径冲突时，会隐藏容器中相应的文件路径。

* -w, --workdir=”

指定容器的工作目录，默认工作目录为根目录”/”

* logs

docker logs命令用于获取指定容器的日志。

Usage: docker logs [OPTIONS] CONTAINER

logs命令获取容器当前日志。只有在使用json-driver日志驱动时有效，其他任何日志驱动下都不可用。

logs命令支持以下参数：

* -f, --follow=true|false

持续跟踪并打印容器日志，默认值为false。

* --since=””

打印指定时间戳以后的日志。

* -t, --timestamps=true|false

显示时间戳，默认值为false。

* --tail=”all”

显示指定行数的最新日志信息。

* ps

docker ps命令列出本地仓库的容器信息，默认只显示运行中的容器。

Usage: docker ps [OPTIONS]

ps命令支持以下参数：

* -a, --all=true|false

显示所有的容器，包括处于停止状态的容器。

* --before=””

只显示在指定容器之前创建的容器，包括运行中和已经停止的容器。

* -f, --filter=[]

提供过滤功能，有效的过滤字段包括：

1. exited=<int> 具有指定退出状态码的容器。
2. label=<key>或label=<key>=<value>：具有指定标签的容器。
3. status=(created|restarting|running|paused|exited)：处于指定状态的容器。
4. name=<string>：使用指定容器名或ID的容器。

* -l, --latest=true|false

只显示最近创建的容器，默认值为false。

* -n=N

显示最近创建的N个容器，包括处于非运行状态的。

* --no-trunc=true|false

不对输出做截断操作，默认值为false。

* -q, --quiet=true|false

只显示容器ID，默认值为false。

* -s, --size=true|false

显示容器中总的文件大小，默认值为false。

* --since=””

只显示在指定容器之后创建的容器。

* commit

docker commit命令用于使用指定容器创建一个新的镜像文件。

Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

commit命令指定一个容器，并使用该容器生成镜像。新的镜像文件中包含容器文件系统的所有内容，但是不包括容器中挂载的数据卷信息。

commit命令支持以下参数：

* -a, --author=””

指定作者信息。

* -c, --change=[]

制作镜像的时候可以使用标准的Dockerfile指令修改镜像信息，如可以使用如下方式设置镜像的http\_proxy环境变量：

cat example.tar | docker import -c “ENV http\_proxy [example@example.com](mailto:example@example.com)” – example:test

说明： change参数可以使用的指令包括：

“CMD”，”ENTRYPOINT”，”ENV”，”EXPOSE”，”ONBUILD”，”USER”，”VOLUME”，”WORKDIR”

* -m, --message=””

用于添加提交信息。

* -p, --pause=true|false

执行commit时暂停容器，默认值为true。

注意： 虽然使用docker commit制作镜像比较方便，但是不推荐使用这种方式制作用于发布和共享的镜像，应该使用Dockerfile和build命令替代。

* export

docker export命令将容器文件系统的内容导出成tar文件。

Usage: docker export [OPTIONS] CONTAINER

export命令将指定容器对应的内容导出到标准输出或output参数指定的文件中。

export命令支持以下参数：

* -o, --output=””

指定导出文件路径。

* stop

docker stop命令用于停止处于运行状态的容器。

Usage: docker stop [OPTIONS] CONTAINER [CONTAINER...]

stop命令支持以下参数：

* -t, --time=10

在指定时间后停止容器，默认值为10秒。

* restart

docker restart命令用于重启运行中的容器。

Usage: docker restart [OPTIONS] CONTAINER [CONTAINER...]

restart命令支持以下参数：

* -t, --time=10

在指定时间之后重启容器，默认值为10秒。

## 管理docker容器

本节将介绍docker中对容器的一些常见操作，包括容器的创建、查询和修改等。然后介绍使用docker编排工具docker-copose进行多容器管理。

### 1.3.1 使用docker管理容器

用户可以使用docker run命令启动容器，该命令首先创建一个容器然后启动，相当于执行docker create + docker start命令。docker run命令启动容器时可以指定相关参数对容器进行配置，具体参数可以参考第二节中对docker run命令的介绍。下面用一个简单的例子介绍docker run的用法：使用registry镜像运行一个名为myregistry的容器；将容器的5000端口暴露到主机，并将主机的/tmp/registry挂载到容器中：

docker run -d -p 5000:5000 -v /tmp/registry:/tmp/registry --name myregistry registry

上例中-d参数表示新创建的镜像会在后台执行，并在终端上打印容器ID。上述命令参数中镜像名后没有指定任何命令，所以容器会执行registry镜像的默认入口命令，即执行docker-registry命令启动registry。

容器启动后，可以通过docker ps命令查看正在运行中的容器。如果想要查看容器的运行状态，可以使用logs、stats或top命令查看容器的日志和资源占用情况。也可以通过docker attach或docker exec命令进入容器；二者的差别在于attach命令会连接到容器的初始进程，并在当前终端显示初始进程的输出；而exec命令会在容器中执行指定的命令，可以执行top或ps命令查看当前容器的资源占用情况，或者执行bash命令进入到容器中查看整个系统的状态。

docker中容器有相应的状态，包括运行，停止，暂停和退出。docker提供相应的命令，如start、stop、pause、unpause、restart和kill，在容器的各个状态之间转换。另外， docker提供commit命令将容器文件系统的改变提交成一个新的镜像，相当于对原有的镜像做了增量改变。使用commit命令生成镜像的方式非常方便，但是在生产环境中不建议使用，因为这样生成的镜像文件并不规范而且不太方便通过Dockerfile的方式共享。生产环境下建议使用Dockerfile和docker build命令来创建镜像。

### 1.3.2使用docker-compose管理容器

前面介绍的是对单个容器进行操作的一些常见命令，而实际使用中系统可能是由多个服务组成，每一个服务都可能需要一个或多个容器来支撑，而且各个服务之间相互关联。这种情况下，使用docker命令来管理整套系统的所有容器是非常复杂的。Docker提供了一种编排工具docker-compose，docker-compose可以简化容器的管理。下面简单介绍docker-compose的使用方法。

docker-compose是用来管理多容器应用的工具。通过一个配置文件来定义多服务应用中各个服务的配置，以及各个服务之间的相互关联关系，然后使用单条指令来部署并启动该应用。

docker-compose的基本使用方式包含三个步骤：

1. 确定应用的执行环境，制作Dockerfile或镜像，以便能够复制重现业务。
2. 确定应用中各服务的配置及相互关系，生成配置文件docker-compose.yml。
3. 使用”docker-compose up“命令来启动并运行该应用。

Dockerfile的语法及使用方法前面已经介绍，不再赘述。下面简单介绍docker-compose.yml文件的一些常见语法和docker-compose管理容器的命令，关于docker-compose更详细的说明请参考项目文档。

docker-compose.yml文件定义的服务必须指定image或build参数，这两个参数用来确定容器运行所依赖的镜像文件。配置中的其他参数可选，而且与docker run命令的参数具有对应关系。

下面通过简单的例子介绍docker-compose.yml文件的使用。如下示例定义了两个容器test\_busybox和test\_ubuntu，其中test\_busybox使用busybox作为基础镜像，而test\_ubuntu则基于当前目录下的Dockerfile文件重新制作镜像文件；test\_ubuntu容器将5000端口暴露到主机，并与test\_busybox容器之间建立链接。

test\_ubuntu:

build: ./

ports： “5000:5000”

links： “test\_busybox:busybox”

test\_busybox:

images: busybox

docker-compose.yml中有些参数也可以通过如下方式指定多个值：

test\_ubuntu:

images: ubuntu

ports：

- “5000:5000”

- “5050:5050”

这样便可以指定test\_ubuntu镜像向主机暴露5000和5050两个端口。

说明： docker-compose.yml文件支持docker运行容器的所有参数，参数的设置规则不再详细介绍，有兴趣的读者可以在docker-compose工程的docs/yml.md文件中查看（<https://github.com/docker/compose/blob/master/docs/yml.md>）。

在docker-compose.yml文件中完成应用的配置后，便可以通过docker-compose命令启动并管理应用。docker-compose提供很多命令实现应用中容器的启动、停止和查询等操作，可以很灵活地管理应用。下面简要介绍docker-compose命令：

* build

Usage: docker-compose build [options] [SERVICE...]

Build命令构建服务并以”project\_service”的形式标记该服务，如果服务已经存在就重新构建。若服务的配置或镜像内容有变化，可以使用docker-compose build命令重新构建。

build命令支持以下参数：

* --no-cache

构建镜像时不使用cache。

* stop

Usage: docker-compose stop [options] [SERVICE...]

stop命令用于停止指定服务对应的容器。

支持以下参数：

* -t, --timeout=TIMEOUT

在指定时间后停止容器，默认值为10秒。

* up

Usage: docker-compose up [options] [SERVICE...]

up命令用于启动指定服务，相当于对服务对应的容器执行build、create、start和attach操作。如果服务对应的容器正在运行，该命令会停止并重新启动该容器，这样可以同步配置或镜像的更新。

up命令支持以下参数：

* --allow-insecure-ssl

允许建立面向docker registry的不安全连接。

* -d

后台模式，在后台启动容器，并将容器名打印到终端输出。

* --no-color

以单色的模式打印输出内容。

* --x-smart-recreate

只重新创建配置或镜像文件发生变化的容器。

* --no-create

不重新创建已经存在的容器。

* --no-build

不论容器依赖的镜像存在与否，都不制作镜像。

* -t, --timeout=TIMEOUT

在指定时间后停止与容器的连接，默认值为10秒。

## 镜像命名规则

因为镜像的名称和镜像操作息息相关，所以作者打算拿一节来专门介绍下Docker镜像的命名规则。我们先看看镜像名称的几个部分：

* Registry 地址： Registry服务器的地址，可以为域名也可以是IP
* 用户命名空间： 用户的名称空间，Registry服务器用户区分不同用户的镜像
* 镜像名称： 镜像的名字
* 镜像Tag： 镜像的tag，用于版本控制

当Docker尝试去push一个镜像到服务器的时候，首先会去解析这个镜像的名称，把Registry（Docker Hub）地址解析出来，这个地址可以是URL也可以是IP<:port> 形式。然后Docker会把Registry地址作为目的地发送REST API（HTTP请求），将镜像push到服务器上；用户命名空间则是Registry为了区分不同的用户而设立的。由此可见，如果一个镜像的名字如果不正确，那么根本不可能成功的push到Registry服务器中。

* Registry命名规范
* 不能包含http请求的Schema“://”
* 如果是URL形式，必须包含“.”
* 可以是IP[:port]形式，比如1.1.1.1:90
* 命名空间的命名规范
* 2-255个字符
* 只允许小写字母和数字以及“\_”或“-”
* 不允许以“-”开头或者结尾，可以在中间
* 不允许出现“--”
* 不允许空格等特殊字符
* 镜像命名规范
* 只允许小写字母和数字以及“\_”和“.”“-”
* 不允许以“-”开头或者结尾，可以在中间
* 不允许空格特殊字符
* 不能包含 ”/”

注意：该规则是通过Docker 1.8.0的源码中提取总结出来的，和1.8.0的规则相同，但是随着Docker代码的开发，不保证和后续Docker版本100%吻合。对于其他版本，如有疑问请自行参看Docker 源码。

## 基于Docker的Web应用和发布

读者已经有了Docker的基本命令，本节将基于Docker部署一个HTTPS的WEB服务器为例子，为您展示Docker的工作方式。我们将把一个web的工程项目部署为一个以Tomcat为服务器的HTTPS的Web站点。其原理和在普通服务器上部署类似，只需要将自己开发的软件包放到Tomcat工程目录下即可。不同的是，要把自己开发的东西放到镜像中，Tomcat服务器要启在容器中。

### 1.5.1 选择基础镜像

部署一个WEB站点，其实有很多选择，比如使用Nginx、Apache等，在这里我们以Tomcat为例。首先需要从镜像库中选择Tomcat的基础镜像，如果没有自己的镜像库，可以从Docker Hub中获取。Docker Hub为我们提供了丰富的版本，包括不同的JRE版本：

root@anio-Tecal-BH622-V2:~ # docker images | grep tomcat

tomcat 8-jre7 71093fb71661 6 days ago 347.7 MB

tomcat 7-jre8 4dcef5c50d60 6 days ago 494.3 MB

tomcat 7.0.63-jre8 4dcef5c50d60 6 days ago 494.3 MB

tomcat 7.0-jre8 4dcef5c50d60 6 days ago 494.3 MB

tomcat 7.0-jre7 f1fb45bb5af9 6 days ago 348.2 MB

tomcat 7 f1fb45bb5af9 6 days ago 348.2 MB

tomcat 7.0.63 f1fb45bb5af9 6 days ago 348.2 MB

tomcat 7-jre7 f1fb45bb5af9 6 days ago 348.2 MB

tomcat 7.0.63-jre7 f1fb45bb5af9 6 days ago 348.2 MB

tomcat 7.0 f1fb45bb5af9 6 days ago 348.2 MB

tomcat 6.0.44-jre8 221207f53791 6 days ago 491.9 MB

tomcat 6.0-jre8 221207f53791 6 days ago 491.9 MB

tomcat 6-jre8 221207f53791 6 days ago 491.9 MB

tomcat 6-jre7 8c4e1a4ca737 6 days ago 345.8 MB

tomcat 6.0.44-jre7 8c4e1a4ca737 6 days ago 345.8 MB

tomcat 6.0.44 8c4e1a4ca737 6 days ago 345.8 MB

tomcat 6.0 8c4e1a4ca737 6 days ago 345.8 MB

tomcat 6.0-jre7 8c4e1a4ca737 6 days ago 345.8 MB

tomcat 6 8c4e1a4ca737 6 days ago 345.8 MB

tomcat 7.0.62-jre8 55f2cc1815fa 4 weeks ago 494.5 MB

tomcat 7.0.62 611a2cc9d3d0 4 weeks ago 348.4 MB

tomcat 7.0.62-jre7 611a2cc9d3d0 4 weeks ago 348.4 MB

tomcat 7.0.61-jre7 1e8d1fcedf1f 11 weeks ago 349.7 MB

tomcat 7.0.61 1e8d1fcedf1f 11 weeks ago 349.7 MB

tomcat 6.0.43-jre7 1e36cc8da5d2 11 weeks ago 347.3 MB

tomcat 6.0.43 1e36cc8da5d2 11 weeks ago 347.3 MB

tomcat 7.0.61-jre8 9e11bd76affb 11 weeks ago 496.4 MB

tomcat 6.0.43-jre8 b8d56384a231 11 weeks ago 494 MB

tomcat 7.0.59-jre8 c54c272df5dc 3 months ago 493.1 MB

tomcat 7.0.59 93f4bcb2ca91 3 months ago 346.1 MB

tomcat 7.0.59-jre7 93f4bcb2ca91 3 months ago 346.1 MB

root@anio-Tecal-BH622-V2:~ #

我们选择Tomcat 7.0-jre8.版本。并在本地将这个镜像Pull下来：

root@anio-Tecal-BH622-V2:~ # docker pull tomcat:7.0-jre8

或者

root@anio-Tecal-BH622-V2:~ # docker pull –a tomcat

小提示：后者会把tomcat的所有版本都pull下来，笔者就是通过pull –a 参数把所以tomcat镜像拉到本地的。

小提示： 如果镜像前面不加namespace和registry server，那么docker默认从官方的Docker Hub服务器 (<https://index.docker.io/v1/>) pull该镜像，所以确保您的服务器可以连接到Docker Hub或者用自己的私有Registry服务器。

### 1.5.2 制作HTTPS服务器镜像

默认的基础镜像是提供的HTTP服务器，我们需要部署的是一个HTTPS Web站点，所以第一步需要先将HTTP的Tomcat支持HTTPS。我们知道，HTTPS是需要证书的，因为本书不是介绍HTTPS或者WEB安全，所以这里只是简单给出生成证书的方法，并不详细介绍，关键是在容器中如何使用这些证书，

* 生成HTTPS需要的证书

因为本书不是专门介绍web开发以及https或者证书相关的书籍，这里只简单的教读者一种生产证书的方法：

root@ubuntu:~/work# mkdir ssl

root@ubuntu:~/work# cd ssl/

root@ubuntu:~/work/ssl# keytool -genkey -alias tomcat -keyalg RSA -keystore tomcat.keystore

Enter keystore password:

Re-enter new password:

What is your first and last name?

[Unknown]: Tester

What is the name of your organizational unit?

[Unknown]: Huawei

What is the name of your organization?

[Unknown]: Huawei

What is the name of your City or Locality?

[Unknown]: Hangzhou

What is the name of your State or Province?

[Unknown]: Zhejiang

What is the two-letter country code for this unit?

[Unknown]: CN

Is CN=Tester, OU=Huawei, O=Huawei, L=Hangzhou, ST=Zhejiang, C=CN correct?

[no]: yes

Enter key password for <tomcat>

(RETURN if same as keystore password):

Re-enter new password:

root@ubuntu:~/work/ssl# ls

tomcat.keystore

root@ubuntu:~/work/ssl#

* 把证书导入到镜像中并commit

root@ubuntu:~/work/ssl# docker run -ti -v $(pwd):/tmp rnd-dockerhub.huawei.com/official/tomcat:7.0-jre8 bash

root@887a6ec5aeec:/usr/local/tomcat# ls

LICENSE NOTICE RELEASE-NOTES RUNNING.txt bin conf lib logs temp webapps work

root@887a6ec5aeec:/usr/local/tomcat# ls /tmp/

tomcat.keystore

root@887a6ec5aeec:/usr/local/tomcat# mkdir keys

root@887a6ec5aeec:/usr/local/tomcat# cp /tmp/tomcat.keystore keys/

root@887a6ec5aeec:/usr/local/tomcat#

docker run –v参数可以将Host的目录动态挂载到容器中的某一路径，本例中就通过-v将当前目录挂载到容器中/tmp目录，来达到目录共享的目的，实现文件从服务器到容器的复制。

* 修改Tomcat的配置并commit

修改端口号为8080的tomcat配置项：

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443"

SSLEnabled="true" scheme="https" secure="true" clientAuth="false" keystoreFile="/usr/local/tomcat/keys/tomcat.keystore" keystorePass="test" sslProtocol="TLS"/>

修改后切勿急于退出容器，可以再打开一个窗口，然后

root@ubuntu:~/work/ssl# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

50f2c2d7e873 official/tomcat: 7.0-jre8 “bash” 10 minutes ago Up 8080/tcp condescending\_brown

root@ubuntu:~/work/ssl# docker commit 50f2c2d7e873 tomcat:https

d798c341b5d01c8e6beb860ac3ccb766d2de98c3c8758c916791b7d8dfee86e5

root@ubuntu:~/work/ssl#

当我们修改完容器后，可以直接选择用commit命令（注意，这个时候容器一定不要退出，因为一旦退出，那么所有的修改都被删掉了）

* 验证基础镜像是否可以正常工作

要想验证基础镜像是否能正常工作，只需要运行起来，看看Tomcat首页是否能正常打开即可。

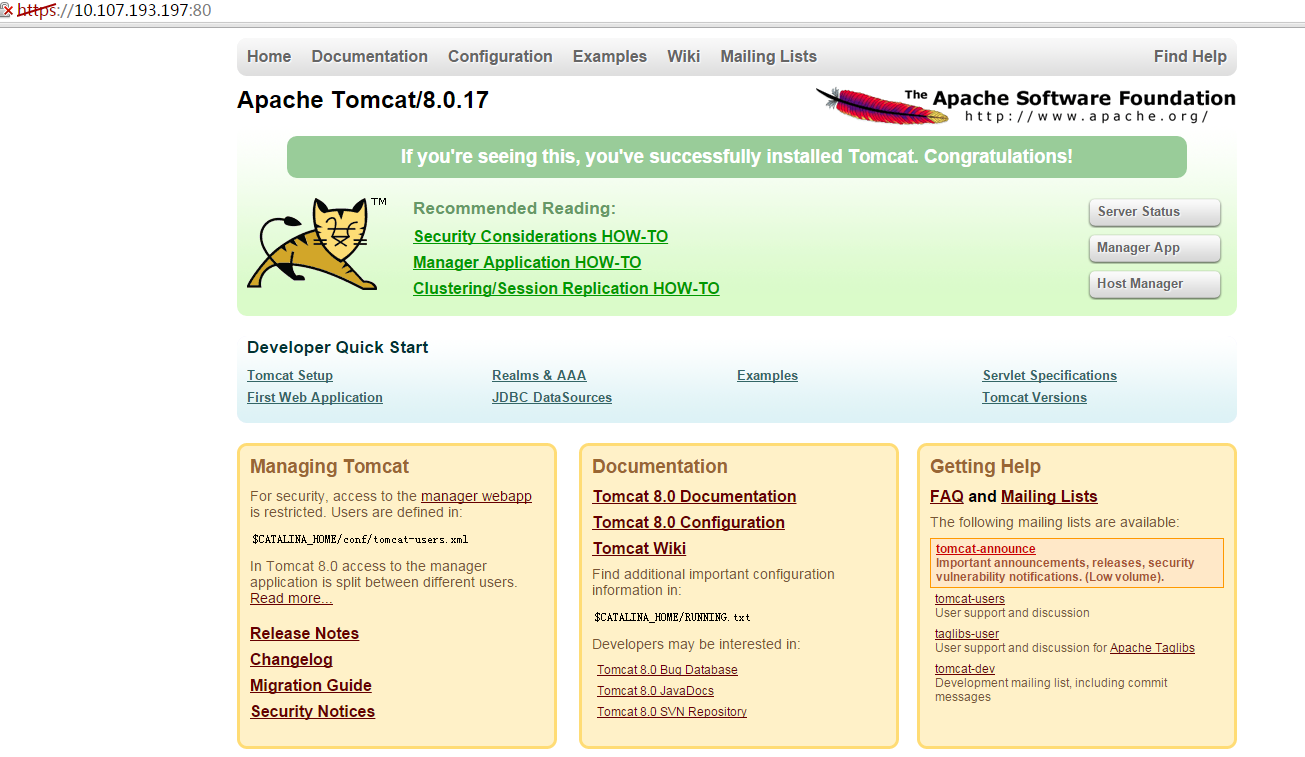
root@ubuntu:~/work/ssl# docker run –p 80:8080 tomcat:https

当docker run命令不加任何command的时候，默认执行容器的ENTRYPOINT命令，该命令是在容器制作的时候指定的。所以建议当制作镜像的时候，为服务性镜像提供默认的ENTRYPOINT。

-p参数表示端口映射，80:8080 表示将host主机上的80端口，映射到container容器中8080端口，这里之所以不一样是为了让读者区分开这两个端口，实际上，也可以8080:8080。

另外我们的证书是自己生成的，所以浏览器会提示不安全，只需点击继续访问即可，真正部署上线，可以购买专门的证书。

好了，服务起来了，我们可以通过Web访问了，如下图，tomcat服务器已经完全启动成功。



这样一个基于https的tomcat服务器就ok了，是不是很容易？之所以容易，是因为基础镜像为我们做了很多事情：首先我们不需要自己去编译安装一个tomcat应用，只需要选择一个基础镜像。其次这个镜像默认启动命令就是启动tomcat，无需关心tomcat细节，当然了，有关web服务器软件的相关知识，还是需要多少懂一些的。

提示，如果公司内部有Registry服务器，可以将这些基础镜像积累下来，方便日后重复利用。

### 1.5.3 将WEB源码导入到Tomcat镜像中

在上一小节，已经成功将证书导入到镜像中，那是不是将源码导入到镜像也是一样的呢？其实，可以说也是，但是也不是。因为将WEB源码导入到镜像中，有很多方式，本节要介绍的是更通用，更便于开发的Dockerfile方式制作镜像。

总体上讲，将源码导入到镜像中有两种思路，一种是静态导入，就是将host主机上的源码（文件）拷贝一份到镜像中；还有一种可以动态将Host主机上的源码目录动态挂载到镜像中。我们分别砍下这两种方式：

* 静态导入：

假如我们的源码放在当前目录，并在当前目录创建个Dockerfile的文件：

# this Dockerfile is used to build HTTP Web Image.

FROM tomcat:https

MAINTAINER Wentao Zhang <[zhangwentao234@huawei.com](mailto:zhangwentao234@huawei.com)>

COPY ./websrc /usr/local/tomcat/webapps/myproj/

我的源码放在当前目录的websrc下面，Tomcat这个镜像将tomcat安装到了/usr/local/tomcat下，所以我们需要把源码拷贝到webapps下面。然后制作镜像：

root@ubuntu:~/work# docker build myweb:v1 .

检测镜像制作完成：

root@ubuntu:~/work/ssl# docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

myweb v1 4986bf8c1536 7 mimutes ago 509.8 MB

root@ubuntu:~/work/ssl#

* 动态挂载：

# This Dockerfile will use dynamic VOLUME to mount the Host SRC code

# to container.

FROM tomcat:https

MAINTAINER Wentao <[wentao@huawei.com](mailto:wentao@huawei.com)>

RUN mkdir –p /usr/local/tomcat/webapps/myproj

VOLUME /usr/local/tomcat/webapps/myproj

动态导入实际上在容器里只创建了一个用来放源代码volume卷，当执行docker run命令时，动态将源码挂载到容器中。

root@ubuntu:~/work/ssl# docker run -ti –v $(pwd)/websrc:/usr/local/tomcat/webapps/myproj myweb:v1

这种方式也可以将web部署成功，读者可以自行检验。

我们总结下，静态导入和动态挂载各有优缺点。静态导入将源码在制作镜像的时就导入到镜像中，干净利索，镜像独立，无论在什么地方只要有Docker环境就可以运行部署成功。但是如果在开发阶段，需要频繁的修改源码，就很不方便；而动态挂载的方式，Host上和容器中共享一份源码，在开发阶段，可以使用这种方式，当发现问题，随时修改，修改后立即生效，省去镜像制作的过程，提高调试效率。

小提示：可以在工程目录下创建两个Dockerfile，一个用于发布版本，用静态导入的方式；另一个用于调试，使用动态挂载方式。docker build是可以制定dockerfile的（-f选项）。

### 1.5.4 部署与验证

我们知道镜像制作成功后，既要部署这个容器应用。这里用到了之前介绍的docker run命令，容器启动方式：

docker run -ti –p 80:8080 myweb:v1

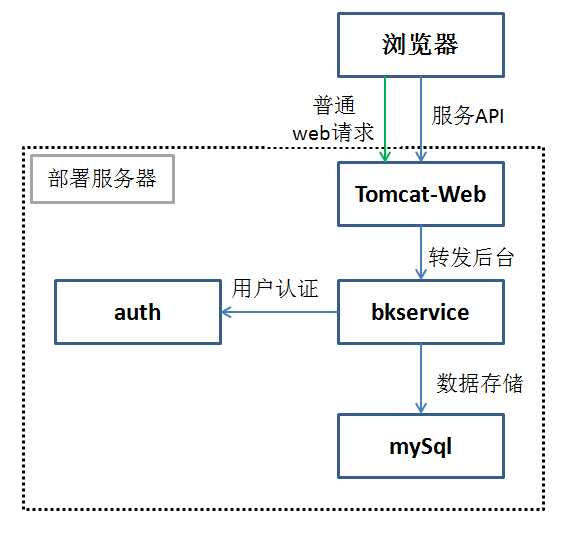
当然如果是动态挂载的，还需要加上-v参数。如果以上步骤部署成功，我们可以通过Chrome等浏览器访问该站点。

Tomcat中server.xml的配置里，我们配的是8080端口，如果读者想用其他端口，可自行修改；另外https证书是自己生产的，并非专门机构签发，所以浏览器会认为站点不安全，需要点击继续访问，方可正常访问该站点。

## 为Web站点添加后台服务

前面我们已经介绍过docker-compose工具，我们可以轻松的管理多个容器，每个容器完成不同的任务和功能。还是以我们的WEB站点为例，一个Web站点是不可能没有后台服务的。假如我们的WEB站点挂了两个后台服务：

* 后台service(bkservice)
* 认证服务器(auth)
* Mysql数据库(Mysql)

其中后台service是用来为Web站点服务的，具体功能由web的功能而定。认证服务器是为了解耦用户认证的服务器，两者通过发送HTTP请求API达到认证鉴权的目的。MySql是数据库，bkservice下挂的数据库。其中每个模块都是独立的，在本工程中，我们会将每个模块都制作成一个镜像。下面看看整体工程框架：

浏览器和前端交互，普通的前段请求均有tomcat服务器响应（比如请求web，显示控件等）。但是需要后台协作的API需要转发到后台，也就是bkservice模块。当然bkservice模块每次收到HTTP请求后，需要发送API到auth模块验证用户的身份以排除排除非法用户。验证通过后，bkservice需要将某些数据存储到mysql的数据库中。

因为本书是介绍docker的，而不是web前端或者后台服务器开发的书籍，所以这个项目只是设计了简单的用力场景，并未真正实现具体功能，只是将这框架搭起来而已。

### 1.6.1 代码组织结构

下面是作者做的工程目录结构：

root@ubuntu:~/work/web# tree

.

├── build.sh

├── docker-compose.yml

├── keys

│   ├── server.xml

│   └── tomcat.keystore

├── misc

│   ├── auth

│   │   ├── Dockerfile

│   ├── bkservice

│   │   ├── Dockerfile

│   │   └── src

│   └── webui

│   ├── Dockerfile

│   └── src

├── release.sh

└── scripts

说明：

* build.sh 编译使用脚本程序
* docker-compose.yml 此文件docker-compse的配置文件
* keys 此文件夹存放用于https的keys，每次制作镜像，将key文件拷贝到tomcat镜像中
* misc 镜像目录，每个镜像有个单独folder，用于制作镜像
  + auth 认证模块容器的制作目录，使用目录下Dockerfile制作auth镜像
  + bkservice 后台服务容器制作目录，源码在src目录下
  + webui 前台容器制作目录，src目录为源码（可选，或者可以在Dockerfile里git clone下来源码，然后编译源码把输出二进制放在镜像中）
* release.sh 版本发布时执行的脚本
* scripts 工程中需要的一些工具脚本

首先看下 docker-compose.yml文件：

root@ubuntu:~/work/web# cat docker-compose.yml

dockerui:

build: misc/webui

volumes:

- misc/webui/src:/usr/local/tomcat/webapps/webui

links:

- dockerbkservice:dockerbkservice

ports:

- 8080:8080

restart: always

dockerbkservice:

build: misc/dockerconsole

links:

- dockerbuild:dockerbuild

volumes:

- misc/dockerconsole/console-docker:/usr/local/tomcat/webapps/console-docker

ports:

- "9090:9090"

environment:

- MACHINE\_CLIENT\_CERT\_PATH=/root/.docker/machine/certs

dockerauth:

build: misc/auth

environment:

- no\_proxy=127.0.0.1

volumes:

- misc/dockerauth/app.conf:/conf/app.conf

dockersql:

image: mysql:v1.0

以dockerbkservice为例，我们讲解一下，

* dockerbkservice: 顶级标签，表示一个镜像
* build: 是指当编译（docker-compose build）会去哪个目录去build这个image
* links: 和其他容器的link，可以以域名形式访问该容器
* volumes: 将host主机上的某个路径，挂载到容器中的某个路径 hostPath:containerPath
* ports: 将host的端口映射到容器中的某个端口，HostPort：containerPort
* environment: 设个一个环境变量

从上面不难推测到，其实docker-compose工具做的工作相当于解析配置文件，之后去按照配置文件的配置项去执行docker 命令而已。这里的四个镜像容器均有docker-compose管理。

小提示：如果觉得将不同的模块放在同一个工程下管理不方便，可以将各模块分别管理，比如auth模块，是一个工程，等执行的时候，可以在misc/auth/Dockerfile 里通过版本控制工具将代码下载下来，并编译运行。以git为例，可以再Dockerfile添加”RUN git clone [git@test/auth.git](mailto:git@test/auth.git); RUN cd auth/ ;make”即可。

### 1.6.2 整体部署服务

其实有了docker-compose工具，部署就会变的极为简单，首先可以考虑删掉之前的镜像，之后重新build，最后一次性启动所有容器。如下：

root@ubuntu:~/work/ssl# docker-compose rm --force

root@ubuntu:~/work/ssl# docker-compose build

root@ubuntu:~/work/ssl# docker-compose up –d

docker-compose命令的用法参看前几个章节。其中rm --force是用来删除镜像（主要是上次build遗留下来的）；build命令时用来制作这些镜像的；up –d是启动这几个镜像的。

## 用实例窥视docker存储

从之前的章节中，我们介绍过Docker存储镜像是分层的，这个其实很抽象，对于初学者来讲会觉得触不可及，本节就带在您不看源码的前提下窥视一下Docker镜像的存储原理。

首先看个例子，Docker支持save和load命令，这一对命令可以将一个镜像存储为一个压缩包（tarball），并且可以用load命令还原该镜像（包括镜像的层）。那么可以推断，save命令一定把镜像存储为压缩包的时候，一定把Docker镜像的存储结构也存下来了。下面我就把其存储的层挖出来。以busybox为例，首先将busybox存储在磁盘上，并解压这个压缩包，如下：

root@ubuntu:~/work/images# docker save busybox:latest > busybox.tar

root@ubuntu:~/work/images# ls

busybox.tar

root@ubuntu:~/work/images# ls

4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125 df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b

511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158 ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2

busybox.tar repositories

root@ubuntu:~/work/images#

从上面可以看出，其实“docker save”把每一层都分开存储，每个文件夹名称就是其层ID。里边存储了该镜像的名称，Tag和ID。我们再看看每个层下面有哪些内容：

root@ubuntu:~/work/images# ls df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b/

json layer.tar VERSION

每个层下面都存储了一个json文件，里边有自己的层id，和父层ID（实际上是一个Layer结构体），像个链表串联在一起。该文件夹下还有个layer.tar存储了该层的数据。

有了这些基础，我们继续探究，假如我们在busybox中增加一个文件（比如在镜像中创建个文件，touch a）

root@ubuntu:~/work/images# docker run -ti busybox sh

/ # touch a

/ #echo hello > a

然后在另一个窗口中commit这个容器为 busybox:v2

root@ubuntu:~/work/build-service# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

6cb9f44e2050 busybox "sh" 7 seconds ago Up 7 seconds fervent\_poincare

root@ubuntu:~/work/build-service# docker commit 6cb9f44e2050 busybox:v2

d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817

root@ubuntu:~/work/build-service#

我们看到commit后生成了一个的层，这个层ID为：d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817

然后在将busybox:v2 save到成busybox2.tar

root@ubuntu:~/work/images# docker save busybox:v2 > busybox2.tar

root@ubuntu:~/work/images/y# tar -xf busybox2.tar

root@ubuntu:~/work/images/y# ls

4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125 d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817 repositories

511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158 df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b

b.tar ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2

我们在tarball里找到了这个层：

d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817

然后我们到这个层下解开layer.tar

root@ubuntu:~/work/images/y/d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817# tar -xf layer.tar

root@ubuntu:~/work/images/y/d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817# ls

a json layer.tar root VERSION

root@ubuntu:~/work/images/y/d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817# cat a

hello

root@ubuntu:~/work/images/y/d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817#

这个层下面多了一个文件a，内容是hello。而且这个tarball里只增加了这一个层。

通过这个实例我们更形象的理解Docker存储镜像方式，首先一个镜像都有一个或多个层，每个层都有自己的层ID，每个层有自己的内容，当把镜像中所有层叠加起来，就是一个镜像。如果两个镜像引用了同一个层，其实Docker可以做到只存一份数据，做到最小存储的效果。