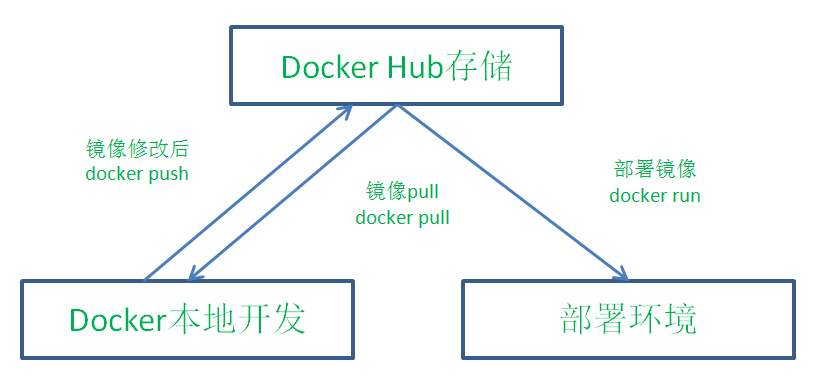
第七章 Docker实战

Docker其实就是轻量级虚拟机，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镜像基本操作

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对镜像的基本操作。

### 创建基础镜像

创建基础镜像的过程依赖该镜像所基于的Linux发行版，可以使用Linux tar命令创建基础镜像。制作基础镜像的主要工作包括获取根文件系统、获取应用程序及其依赖的包文件、将所有文件打包并将打包文件导入制作成Docker镜像。

通常情况下，如果想使用某种特定的Linux发行版制作基础镜像，需要启动一台运行相应发行版系统的机器。而在非Debian/Ubuntu机器上可以使用Debian debootstrap工具构建Debian/Ubuntu镜像。debootstrap是一款用来引导基础Debian系统的工具，可以在不使用安装光盘的情况下在系统中安装Debian，也可以用来在chroot环境中安装不同规格的Debian系统。

Docker源码项目中提供了一些用来制作基础镜像的脚本，位于源码目录中contrib文件夹下，该目录在GitHub中的地址为：

https://github.com/docker/docker/tree/master/contrib

用户可以根据自己的需求使用contrib目录下相应的mkimage-\*.sh脚本制作基础镜像。

### Docker的镜像操作：

Docker提供如下命令对镜像文件进行操作：

|  |  |
| --- | --- |
| 命令 | 描述 |
| build | 根据Dockerfile构建镜像 |
| commit | 将指定容器提交成为新的镜像 |
| history | 查看镜像的历史信息 |
| import | 从Linux tar文件中导入新的镜像文件系统 |
| inspect | 查看镜像文件的底层信息 |
| images | 列出当前系统包含的镜像 |
| load | 加载Linux tar文件生成镜像文件 |
| pull | 从Docker Registry中获取镜像或仓库 |
| push | 将镜像或仓库上传到Docker Registry中 |
| rmi | 从本地存储中删除指定镜像 |
| save | 经镜像文件保存到tar文件归档 |
| search | 在Docker Hub或指定Registry中搜索镜像 |
| tag | 将镜像标记成仓库 |

用户可以使用docker images命令查看当前主机上的所有镜像：

$ docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

ubuntu 14.04 99ec81b80c55 4 weeks ago 266 MB

从docker images命令返回的结果中可以得到镜像所属的仓库、tag、镜像ID、创建时间和镜像的大小。如果本地没有所需要的镜像文件，可以使用docker pull命令来获取。docker pull命令默认从官方Docker Hub上获取镜像，用户也可以显式指定私有的Docker Registry作为镜像源。docker pull命令的使用方式如下：

$ docker pull [$PRIV\_REGISTRY]$REPOSITORY:$TAG

Docker Hub上提供了基于多种Linux发行版的镜像，有些镜像中包含专有的组件可以用来提供特定的服务。用户可以通过浏览器访问Docker Hub网站搜索所需要的镜像：

<https://hub.docker.com/>

Docker提供了login命令来登陆Docker Hub，docker Hub具有账户管理功能。用户可以使用docker search命令来执行镜像搜索。search命令默认在Docker Hub上搜索指定镜像，也可以指定私有的Registry，docker search命令的使用方式如下：

$ docker search [$PRIV\_REGISTRY]$SEARCH\_ITEM

Docker提供了很多不同功能镜像，但是当用户没有发现满足自己需求功能的镜像时，可以选择自己制作镜像。Docker提供了两种制作镜像的方式：

1、更新现有的容器中内容，并使用docker commit命令生成一个新的镜像

2、使用Dockerfile来指定基础镜像和依赖信息来创建镜像

使用Dockerfile制作镜像的详细介绍可以参考XXXXX节的内容，下面简单介绍如何使用docker commit命令创建镜像：

首先使用docker run命令启动一个基础镜像，这里以Ubuntu为例：

$ docker run –ti ubuntu:latest apt-get install vim

上述命令使用ubuntu:latest镜像创建了一个ID为0d58f534a02e的容器，在容器中执行apt-get install vim命令安装vim。vim安装成功后，容器会自动退出。然后使用docker commit命令将这个容器的拷贝提交成一个新的镜像文件：

$ docker commit -m "Install vim" -a "TestUser" 0d58f534a02e ubuntu:vim

7cb8e07efd63fd1be0a7ed0667bd256cc49411cfe0f9b754a70ff1e025fa5501

上例中”-m”参数用来指定提交信息，用来描述对镜像所做的改动；”-a”参数指定制作者信息。“0d58f534a02e” 是我们要提交的容器，也就是成功安装vim的ubuntu，。用户也可以指定生成镜像的仓库名和tag，本例中使用”ubuntu:vim”。最终docker commit命令生成了新的镜像文件，并在终端打印出镜像ID，docker commit执行结束后可以使用docker images命令查看本地是否新增”ubuntu:vim”镜像。

用户可以使用docker history命令可以查看ubuntu:vim镜像的历史信息：

$ docker history ubuntu:vim

IMAGE CREATED CREATED BY SIZE COMMENT

7cb8e07efd63 2 days ago /bin/sh apt-get intall vim 2.2 MB Install vim

d0955f21bf24 5 months ago /bin/sh -c #(nop) CMD [/bin/bash] 0 B

9fec74352904 5 months ago /bin/sh -c sed -i 's/^#\s\*\(deb.\*universe\)$/ 1.895 kB

a1a958a24818 5 months ago /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic 194.5 kB

f3c84ac3a053 5 months ago /bin/sh -c #(nop) ADD file:777fad733fc954c0c1 188.1 MB

511136ea3c5a 2 years ago 0 B Imported from -

docker history命令的返回结果中可以看到我们最新提交的”Install vim”信息，实际使用中可以使用该命令对出现错误的镜像进行回溯，查找错误原因。

制作新的镜像后，用户可以将镜像上传到公有的Docker Hub中来实现镜像的共享。可以使用docker push命令来实现镜像的上传：

$ docker push $IMAGE\_NAME[:$IMAGE\_TAG]

用户也可以选择将镜像上传到自己搭建的私有Registry中，首先需要使用docker tag命令为镜像重新命名成“$PRIV\_REGISTRY/$NAMESPACE/$REPO\_NAME:$TAG”的形式，然后执行docker push操作，Docker 会检测出镜像名中的私有Registry地址，并将镜像上传到该Registry中：

$docker tag ubuntu:vim $PRIV\_REGISTRY/TestUser/ubuntu:vim

$docker push $PRIV\_REGISTRY/TestUser/ubuntu:vim

除了上面介绍的使用docker push和docker pull命令外，docker还提供了其他方式来实现镜像的共享和迁移。用户可以使用docker save命令将镜像打包成Linux tar文件归档，拷贝到另外一台主机后通过docker load命令将tar文件加载为docker镜像；使用docker load和docker save命令可以保存镜像文件的元数据信息。用户也可以通过docker export和docker import命令实现容器的保存和导入，但是这样会丢失镜像和容器的元数据。

Docker提供了docker rmi命令来删除本地存储的镜像，用户可以使用如下命令删除本地无用的镜像：

$docker rmi $IAMGE\_NAME:$IMAGE\_TAG

注：上述介绍的命令和示例都基于registry V1的镜像。对于V2及更高版本Registry，可以使用digest信息来确定镜像。

## 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。Docker中容器操作的命令如下：

|  |  |
| --- | --- |
| 命令 | 描述 |
| Attach | 连接到一个正在运行的容器中 |
| Cimmit | 将容器的内容提交成新的镜像 |
| Cp | 在容器和主机之间进行文件拷贝 |
| Create | 创建新的容器 |
| Diff | 查看容器中文件系统内容的更新 |
| Exec | 在处于运行状态的容器中执行指定命令 |
| Export | 将容器的内容使用Linux tar归档 |
| Inspect | 查看容器的底层信息 |
| Kill | 杀掉运行中的容器 |
| Logs | 获取指定容器的日志输出 |
| Pause | 停止容器中的所有进程 |
| Port | 查看指定容器中的公共端口 |
| Ps | 查看当前系统的容器 |
| Rename | 重命名指定容器 |
| Restart | 重启指定容器 |
| Rm | 删除指定容器 |
| Run | 创建一个新的容器并执行指定命令 |
| Start | 启动指定容器（处于停止状态） |
| Stats | 显示容器的资源占用情况 |
| Stop | 停止指定容器（处于运行状态） |
| Top | 查看容器中运行的进程 |
| Unpause | 恢复一个处于暂停状态的容器 |
| Wait | 等待容器执行结束，并打印退出码 |

本节将根据上述Docker容器命令介绍Docker对容器的一些常见操作，包括容器的状态控制、容器状态查看、容器的数据管理和容器之间建立链接等内容，然后简单介绍怎样使用Docker编排工具docker-copose进行多容器管理。

### 容器状态管理

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执行指定命令，相当于执行”docker create + docker start”命令。使用docker run启动容器时可以指定相关参数对容器进行配置，具体参数及使用说明可以参考docker run命令的介绍：

https://github.com/docker/docker/blob/master/man/docker-run.1.md

下面用一个简单的例子介绍容器的启动，我们使用本地registry镜像启动一个名为myregistry的容器：

$docker run -d -p 5000:5000 --name myregistry registry

0fc253e0c8d1b21f4014c9d9ecfee1d87bc5d80d3c5f7dc44182f8fe79275599

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服务。通过使用”-p 5000:5000”参数将容器的5000端口映射到主机的5000端口，用户可以在其他机器上使用当前主机的5000端口访问容器提供的registry服务。示例中docker将容器端口暴露到主机所有接口的5000端口上，也可以使用“-p 127.0.0.1:5000:5000“这种形式的参数指定将容器端口映射到localhost的5000端口。我们同样可以指定绑定端口所使用的协议，”-p 5000:5000/udp”参数表示用户使用udp协议暴露5000端口。可以通过docker port命令来查看容器的端口映射关系。

docker中容器有相应的状态，包括运行，停止，暂停等。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进行转换和恢复。如果要停止上面启动的myregistry容器，可以使用docker stop或者docker kill命令。二者的差别在于docker stop可以指定一定的时间延迟，而docker kill命令可以指定发送给docker内进程的信号类型。

$docker stop myregistry

$docker kill myregistry

处于停止状态的容器无法使用docker ps命令显示，需要加上”-a”参数才能看到。用户可以使用docker start命令启动处于停止状态的容器：

$docker start myregistry

如果需要重启容器，可以使用docker restart命令，该命令相当于执行”docker stop + docker start”，restart命令会先停止指定容器，然后重新开启：

$docker restart myregistry

用户也可以使用pause、unpause命令来暂停或恢复容器中的进程：

$ docker pause myregistry

$ docker unpause myregistry

执行pause命令后，容器处于Paused状态，使用unpause命令可以恢复容器的执行状态。处于Paused状态的容器，无法使用start、stop、rm、restart等命令修改容器状态，需要先执行unpause恢复容器的执行状态。

对于已经停止运行的容器，可以使用docker rm命令移除容器：

$docker rm myregistry

注意：如果容器处于运行状态，使用docker rm执行删除操作会失败，应该先执行docker stop停止容器然后使用rm删除，或者使用“-f“参数强行删除运行中容器。

### 容器状态查看

docker ps命令可以用来查看当前系统中正在运行的容器：

$docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

0fc253e0c8d1 registry "docker-registry" 3 seconds ago Up 2 seconds 0.0.0.0:5050->5000/tcp myregistry

docker ps命令的返回信息中包括容器ID、启动容器所使用的镜像、当前容器执行的启动命令、容器的启动时间、容器当前的状态、容器的端口信息和容器名。ps命令默认显示的是当前系统中处于运行状态的容器，如果想要查看处于停止状态的容器，需要使用”-a”参数。

如果想要查看当前容器中运行进程的输出内容，可以使用docker logs命令：

$docker logs –f myregistry

15/Aug/2015:06:43:49 +0000 DEBUG: Will return docker-registry.drivers.file.Storage

[2015-08-15 06:43:50 +0000] [1] [INFO] Starting gunicorn 19.1.1

[2015-08-15 06:43:50 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)

[2015-08-15 06:43:50 +0000] [1] [INFO] Using worker: gevent

[2015-08-15 06:43:50 +0000] [13] [INFO] Booting worker with pid: 13

“-f“参数会持续监听容器的输出内容，可以使用CTRL-C停止。

用户也可以使用docker top命令查看容器中运行的进程：

$ docker top myregistry

UID PID PPID C STIME TTY TIME CMD

root 15800 26293 0 14:43 ? 00:00:00 /usr/bin/python /usr/local/bin/gunicorn --access-logfile - --error-logfile - --max-requests 100 -k gevent --graceful-timeout 3600 -t 3600 -w 4 -b 0.0.0.0:5000 --reload --preload docker\_registry.wsgi:application

… …

top命令的返回结果中包括当前myregistry容器中为启动registry服务运行的所有进程。另外docker也提供了inspect命令来查看容器的底层信息，：

$docker inspect myregistry

{

"Id": "0fc253e0c8d1b21f4014c9d9ecfee1d87bc5d80d3c5f7dc44182f8fe79275599 ",

"Created": "2015-08-15T06:43:45.538760576Z",

"Path": "docker-registry",

"Args": [],

"State": {

"Running": true,

"Paused": false,

"Restarting": false,

"OOMKilled": false,

"Dead": false,

"Pid": 15800,

"ExitCode": 0,

"Error": "",

"StartedAt": "2015-08-15T06:43:46.082877891Z",

"FinishedAt": "0001-01-01T00:00:00Z"

},

"Image": "fce0875119375894384df6df6b50d91cc4e989a52dda4789d71bf0e8a56d0b61",

… …

inspect命令显示指定容器的底层相关信息。信息内容较多，默认会以JSON格式呈现返回结果，也可以通过“-f/--format“参数指定信息呈现格式，例如可以通过如下方式获取到inspect信息中的IP地址：

docker inspect –format=’{{.NetworkSettings.IPAdress}}’ myregistry

说明： 更多关于如何使用Go模板的信息请参考<http://golang.org/pkg/text/template/>

Docker提供了其他方式来查看容器的运行状态， 如用户可以使用stats命令来查看容器资源占用情况；也可以通过docker attach或docker exec命令进入容器查看运行状态，这两个命令的差别在于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但因信息，而exec命令会在容器中创建新的进程执行指定的命令，用户可以使用exec命令执行top或ps命令查看当前容器的资源占用和进程执行情况，也可以执行”/bin/bash”命令进入到容器中的bash shell查看整个系统的状态。

### 容器数据管理

Docker镜像和容器的管理使用分层的联合文件系统。为了能够持久化保存和共享容器的数据，Docker提出了Volume的概念，简单来讲Volume就是目录或者文件，Volume可以绕过默认的联合文件系统，以正常的文件或目录的形式存放在宿主机上。Docker提供了两种管理数据的方式：数据卷和数据卷容器。

### 新增数据卷：

用户可以在执行docker create或docker run命令时使用“-v“参数来添加数据卷，可以通过多次指定该参数来挂载多个数据卷，仍然以创建myregistry容器为例：

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

“-v”参数会在容器的/tmp/registry目录下创建一个新的数据卷。新的数据卷默认会以可读写的方式创建。

用户可以通过docker inspect命令定位容器中数据卷在主机中的位置：

$docker inspect myregistry

… …

"Volumes": {

"/tmp/registry": "/var/lib/docker/volumes/08c5670c709f53dd10313e4309a4bb19e883102c0ed99207c4833eb1d1abc8e1/\_data"

},

"VolumesRW": {

"/tmp/registry": true

},

… …

从上面inspec的返回结果中可以看到，”/tmp/registry”数据卷对应的主机目录位置为"/var/lib/docker/volumes/08c5670c709f53dd10313e4309a4bb19e883102c0ed99207c4833eb1d1abc8e1/\_data"。

### 将主机目录挂载为数据卷

“-v”参数除了可以用于创建数据卷外，还可以用来将docker daemon所在主机上的文件或文件夹挂载到容器中，比如：

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

上述命令可以将docker daemon所在主机的”/host/registry”目录挂载到容器的”/tmp/registry”路径下。将主机目录挂载为数据卷的功能在有些场景下非常有用。比如，我们可以将主机上的源码目录挂载到容器中，然后使用改动后的源码重新编译程序。使用volume挂载主机目录时必须使用绝对路径，如果指定路径不存在，docker会自动创建该目录。

注意：如果容器中”/tmp/registry”路径已经存在，docker会使用”/src/registry”的内容覆盖该目录，这样可以与mount命令的功能保持一致。

### 创建数据卷容器：

如果用户需要在容器之间共享一些需要永久存储的数据，或者想要使用一个临时容器中的相关数据，可以创建一个数据卷容器，然后使用该容器进行数据共享。

假如我们想要创建多个Postgres数据库，并且希望这些数据库之间能够共享数据，可以先创建一个数据卷容器，该容器中并不运行任何应用：

$ docker create -v /dbdata --name dbdata training/postgres /bin/true

然后启动Postgres数据库服务，使用”—volumes-from”参数将上面生成的数据卷挂载进来，可以启动多个容器，各个容器之间都通过dbdata数据卷共享数据：

$ docker run -d --volumes-from dbdata --name db1 training/postgres

$ docker run -d --volumes-from dbdata --name db2 training/postgres

用户也可以使用”—volume-from db1”或者”—volume-from=db2”的方式挂载dbdata数据卷：

$ docker run -d --name db3 --volumes-from db1 training/postgres

使用数据卷容器存储的数据不会轻易丢失，即便删除db1、db2容器甚至是初始化该数据卷的dbdata容器，该数据卷也不会被删除。只有在删除最后一个使用该数据卷的容器时显式地指定“docker rm -v“才会删除该数据卷。

数据卷的备份、转储和迁移

使用数据卷的方式管理容器数据时可以很方便地进行数据卷中数据的备份、转储和迁移。我们可以使用如下命令将数据卷中的数据打包，并将打包后文件拷贝到主机当前目录中：

$ docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

上述命令创建了一个容器，该容器挂载了dbdata数据卷，并将主机的当前目录挂载到容器的”/backup”目录中；然后在容器中使用tar命令将dbdata数据卷中的内容打包存放到”/backup”目录的backup.tar文件中。当容器执行结束后，备份文件就会出现在主机的当前目录。之后可以将该备份文件恢复到当前容器或新创建的容器中，完成数据的备份和迁移工作。

### 容器连接

Docker提供了一种link机制，该机制可以用来在多个容器之间建立连接，并将连接信息发送到相应的容器中。Docker中为容器建立连接的方式很简单。比如我们首先创建一个数据库服务，并为其指定容器名为postgresdb：

$ docker run -d --name postgresdb training/postgres

注意：如果创建容器时没有指定容器名，Docker会为容器分配一个随机的名字。实际使用场景中最好根据容器的用途为容器指定特定的名字，这样可以用于区分不同服务；Docker提供了”--name”参数来指定容器名。

然后创建另外一个容器，并将其与postgresdb容器之间建立连接：

$docker run –-name link-test –link postgresdb:db ubuntu /bin/bash

从上面命令的参数中可以看到，postgresdb容器在新创建的link-test容器中的别名为db，我们可以在link-test容器中使用db这一别名来访问posgresdb容器提供的服务。

### 多容器管理

前面介绍的是对单个容器进行操作的一些常见命令，而实际使用过程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，而且各个服务之间存在依赖关系。这种情况下，仅仅使用Docker命令来管理整套系统的所有容器是非常复杂的，可以使用Docker提供的编排工具docker-compose来简化多容器的管理。

docker-compose是一种用来管理多容器应用的工具。用户可以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，然后可以使用一条命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或镜像文件，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。

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。

说明：docker-compose项目在GitHub上的地址为：https://github.com/docker/compose

docker-compose.yml文件中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，其中image参数指定使用当前系统已经存在的镜像，build参数指定使用Dockerfile重新构建新的镜像。配置中的其他参数可选，而且与docker run命令支持的参数具有对应关系。docker-compose.yml文件支持的参数包括：

|  |  |
| --- | --- |
| 参数 | 功能 |
| image | 指定启动服务所使用的镜像。 |
| build | 用来制作docker镜像的路径，需要包含Dockerfile |
| dockerfile | 使用指定Dockerfile制作镜像 |
| command | 覆盖默认的执行命令 |
| links | 指定容器和其他容器之间的连接 |
| external\_links | 与当前docker-compose.yml指定工程之外的容器建立连接 |
| extra\_hosts | 添加新的主机映射，与docker的add-host参数类似 |
| ports | 指定容器的端口映射 |
| expose | 将指定端口暴露给建立连接的容器 |
| volumes | 将指定路径挂载为数据卷 |
| volumes\_from | 从其他服务或容器中挂载数据卷 |
| environment | 添加环境变量 |
| envfile | 从指定文件中添加环境变量 |
| extends | 从当前配置文件或其他配置文件中扩展新的服务 |
| labels | 使用docker labels为容器添加元数据 |
| container\_name | 指定容器名 |
| log\_driver | 指定容器使用的日志驱动类型 |
| net | 指定容器网络模式 |
| pid | 指定容器的PID模式 |
| dns | 为容器指定DNS服务器 |
| cap\_add, cap\_drop | 为容器新增或删除linux Capabilities |
| dns\_search | 设置容器DNS搜索域 |
| devices | 指定容器的设备映射信息 |
| security\_opt | 设置容器的安全策略 |
| working\_dir | 指定容器的工作路径 |
| entrypoint | 指定容器的入口 |
| user | 指定运行容器所使用的用户 |
| hostname | 指定容器的主机名 |
| domainname | 指定容器域名 |
| mac\_address | 指定容器中网卡MAC地址 |
| mem\_limit | 指定容器中内存占用上限 |
| memswap\_limit | 指定容器中所有内存的使用上限，包含swap分区 |
| privileged | 指定以特权模式启动容器 |
| restart | 指定容器的重启策略 |
| stdin\_open | 标出输入开关 |
| tty | tty开关 |
| readonly | 以只读的方式挂载根文件系统 |
| cpu\_shares | 设置容器占用CPU时间的比重 |
| cpu\_set | 设置容器运行时的CPU亲和性 |

说明： docker-compose.yml文件支持docker运行容器的所有参数，各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>）。

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使用系统中的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。

当多个应用程序需要复用一个共同的服务时，可以使用docker-compose提供的extends参数来共享或复用其他配置文件甚至其他工程中定义的服务，这意味着我们可以在一个配置文件中定义一个基础服务，并在其他任何地方引用该服务，并且在复用该服务时可以修改原有的配置参数或添加新的配置。extends参数的使用方法如下例所示，我们可以在一个名为common-service.yml的配置文件中定义一个名为webapp的服务：

webapp:

build: .

ports:

- "8000:8000"

volumes:

- "/data"

然后在当前工程中引用webapp服务：

Web:

Extends:

File: common-service.yml

Service: webapp

Ports:

- “5000:8000“

Environment:

- DEBUG=1

复用webapp时，可以修改或新增相关配置，上面配置文件中新定义的web服务就修改了webapp的端口映射信息，并新增了环境变量”DEBUG=1”。

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各服务的启动、停止和查询等操作，可以很灵活地管理应用。docker-compose包括以下命令：

|  |  |
| --- | --- |
| 命令 | 描述 |
| build | 构建新的服务 |
| kill | 杀掉运行中的容器 |
| logs | 查看容器的输出信息 |
| port | 显示指定服务的公共端口 |
| ps | 显示运行中的容器 |
| pull | 获取服务所需要的镜像文件 |
| restart | 重启指定的服务 |
| rm | 删除处于停止状态的容器 |
| run | 运行指定服务并执行指定命令 |
| scale | 设定服务可扩展的容器数量 |
| start | 启动服务 |
| stop | 停止服务 |
| up | 创建启动服务对应 容器 |
| migrate-to-labels | 使用label的方式重新创建容器 |

docker-compose支持”-f/--file”参数来指定所使用的配置文件，默认使用当前目录下docker-compose.yml文件。如果当前目录下不存在该文件，docker-compose会尝试访问所有的父目录，直到找到docker-compose.yml文件为止。用户也可以通过”-p/--project-name”参数来指定工程名，如果不显式指定该参数会默认使用目录名作为工程名。用户在确定当前目录下配置文件docker-compose.yml中各个服务的配置和关联正确后，可以通过下面的简单命令一键启动整个应用：

$docker- compose up –d

该命令会根据配置文件获取或创建所需的镜像，并根据各个服务的配置启动容器运行服务。也可以通过上面介绍的”-f/--file”和”-p/--project-name”参数指定该工程的配置文件和工程名。

如果工程中某一个服务的代码有改动，可以使用如下命令更新指定服务：

$docker-compose build $SERVICE

该命令可以重新构建$SERVICE指定服务的镜像，停止并删除该服务原有的容器，然后使用新的镜像重新创建容器。用户可以使用上表中列出的命令来对修改或查看当前工程的运行状态，各个命令的使用方法与前文介绍的docker相应命令类似，可以使用”docker-compose $COMMAND --help”查看各命令的详细说明。

注意：以上介绍的docker-compose命令和配置参数都是基于docker-compose 1.4.0版本。

## Dockerfile简介

前面我们已经了解如何制作镜像，也提到了使用Dockerfile制作镜像。接下来我们将一起来探讨一下Dockerfile的构成和语法。通过本节的学习，您可以轻松的写出自己需要的Dockerfile来。本节由浅入深，由一个简单例子入手，之后会一一介绍Dockerfile的各种命令。不会让读者觉得很难入手。

### 1.4.1一个简单的例子

Dockerfile的注释都是以“#”开始的，每一行是一个命令。一般情况下，Dockerfile由四部分组成：基础镜像信息，维护者信息，镜像操作指令（命令）和容器启动指令。下面是一个Nginx服务镜像Dockerfile文件内容：

# This Dockerfile uses the ubuntu image

# VERSION 2 - EDITION 1

# Author: tester

# Command format: Instruction [arguments / command] ..

# Base image to use, this must be set as the first line

FROM ubuntu

# Maintainer: tester < tester at email.com> (@docker\_user)

MAINTAINER tester tester@huawei.com

# Commands to update the image

RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >> /etc/apt/sources.list

RUN apt-get update && apt-get install -y nginx

RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf

# Commands when creating a new container

CMD /usr/sbin/nginx

需要说明的是，如果使用Dockerfile来构建镜像，Dockerfile的第一条有效信息（注释除外）必须是基础镜像信息，维护者信息紧随其后。镜像操作指令在维护信息之后，因为操作指令的不同，也就构建出千差万别的镜像来。最后是镜像启动命令，它被用作镜像的默认启动指令。

### 1.4.2 Dockerfile的语法命令

从前面的例子中，我们可以看到几个熟悉的命令：“FROM ubuntu”，“RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf”等等，不难看出，Docker命令的一般格式为 INSTRUCTION arguments，而这些命令包括 FROM、MAINTAINER、RUN 等。下面我们将把最为常用的几个命令介绍下：

* FROM指令

格式为 FROM <image>或FROM <image>:<tag>

Dockerfile的第一条必须是FROM指令，用来指定要制作的镜像继承自哪个镜像。注意，FROM指令可以跟多个基础镜像（每个镜像一次）。

* MAINTAINER

格式为 MAINTAINER <name>

用来指定维护者信息。

* RUN

格式为 RUN <command> 或 RUN ["executable", "param1", "param2"…]。

该命令是用来执行shell命令，当解析Dockerfile时，遇到RUN命令，Docker会将该命令翻译为“/bin/sh –c “xxx””，其中xxx为RUN后边的shell命令。

注：RUN命令也是在制作镜像的容器中执行的。

* EXPOSE

格式为 EXPOSE <port> [<port>...]。

该命令用来将容器中的端口号暴露出来，只有暴露出来的端口号，才可以在容器启动（docker run –p）的时候和服务器端口做映射。在后续我们会用到这个命令。

* CMD

有三种格式：

CMD ["executable","param1","param2"] 使用 exec 执行，推荐方式；

CMD command param1 param2 在 /bin/sh 中执行，提供给需要交互的应用；

CMD ["param1","param2"] 提供给 ENTRYPOINT 的默认参数；

指定启动容器时执行的命令，每个 Dockerfile 只能有一条 CMD 命令。如果指定了多条命令，只有最后一条会被执行。

如果用户启动容器时候指定了运行的命令，则会覆盖掉 CMD 指定的命令。读者可以和 ENTRYPOINT命令对比。

* ENTRYPOINT

两种格式：

ENTRYPOINT ["executable", "param1", "param2"]

ENTRYPOINT command param1 param2（shell中执行）。

每个 Dockerfile 中只能有一个 ENTRYPOINT，当指定多个时，只有最后一个起效。

该命令配置容器启动后执行的命令，并且不可被 docker run 提供的参数覆盖。所以建议使用CMD命令，而不是ENTRYPOINT命令。

* VOLUME

格式为 VOLUME ["/data"]。

创建一个可以从本地主机或其他容器挂载的挂载点，一般用来存放数据库和需要保持的数据等。如果和host共享目录，Dockerfile中必须先创建一个挂载点，然后在启动容器的时候通过 （docker run –v HOSTPATH:IMAGEPATH）来挂载，其中IMAGEPATH就是创建的挂载点。

* ENV

格式为 ENV <key> <value>。

指定一个环境变量，会被后续 RUN 指令使用，并在容器运行时保持。

* ADD

格式为 ADD <src> <dest>。

该命令将复制指定的 <src> 到容器中的 <dest>。 其中 <src> 可以是Dockerfile所在目录的一个相对路径；也可以是一个 URL；还可以是一个 tar 文件（自动解压为目录）。

* COPY

格式为 COPY <src> <dest>。

复制本地主机的 <src>（为 Dockerfile 所在目录的相对路径）到容器中的 <dest>。

当使用本地目录为源目录时，推荐使用 COPY。

### 1.4.3 再谈Docker镜像制作

前面我们已经介绍过，使用docker build命令并给定一个Dockerfile就可以制作一个镜像。那么Docker是怎么把这个镜像制作出来的呢？因为本书的读者并不是开发Docker或者阅读Docker源码的人，所以本节就以实例的方式将Docker制作镜像的过程简单的呈现出来。

先准备一个基础镜像busybox:latest，这个镜像可以从官方pull下来。然后写一个简单的Dockerfile如下：

# This Dockerfile uses the busybox image

# VERSION 1 - EDITION 1

# Author: tester

FROM busybox:latest

RUN date;sleep 100;date

RUN echo “abc” > /mytest

RUN date;sleep 100;date

CMD /bin/sh

我们用build命令去制作镜像，名称为busybox:v1，如下：

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

现在Docker正在制作这个镜像，我们可以通过ps看下到底发生了什么：

root@ubuntu:/tmp # docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

c5db3d7ca9b9 d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057acb8817 "/bin/sh -c 'date;sl 9 seconds ago Up 8 seconds mad\_yalow

root@ubuntu:/tmp #

从中可以看到，在镜像制作的时候，Docker会启动一个容器，并在该容器里制作这个镜像。接下来我们通过exec命令，去容器中看看，到底发生了什么：

root@ubuntu:/tmp/dockerBuild# docker exec -ti c5db3d7ca9b9 sh

/ # ps -ef

PID USER COMMAND

1 root /bin/sh -c date;sleep 100;date

6 root sleep 100

16 root sh

21 root ps -ef

/ #

从上边的命令中，我们看到，docker 在制作镜像的RUN命令都是在容器中执行的。如果我们等上100秒，不停的ps,我们会顺序的抓到RUN的每一条命令。

从上面我们可以看出，Docker在制作镜像的时候的操作顺序：

**1. 解析Dockerfile，并找到基础镜像（可能会从DockerHub上Pull）**

**2. 以基础镜像为基础创建一个容器**

**3. 在容器中顺序执行Dockerfile中的命令。**

**4. 如果不是RUN命令，比如ENV命令，记录下来以便启动的时候执行**

**5. 属性命令记录在Image的属性中。**

**6. 当所有命令执行完后，commit该容器为新的镜像。**

当然这里说的几个步骤，只是简单的阐述了镜像制作的过程，其实中间设计到很多复杂操作，比如分层存储等。有兴趣的读者，可以去阅读相关源码。

## 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的源码中提取总结而得，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。

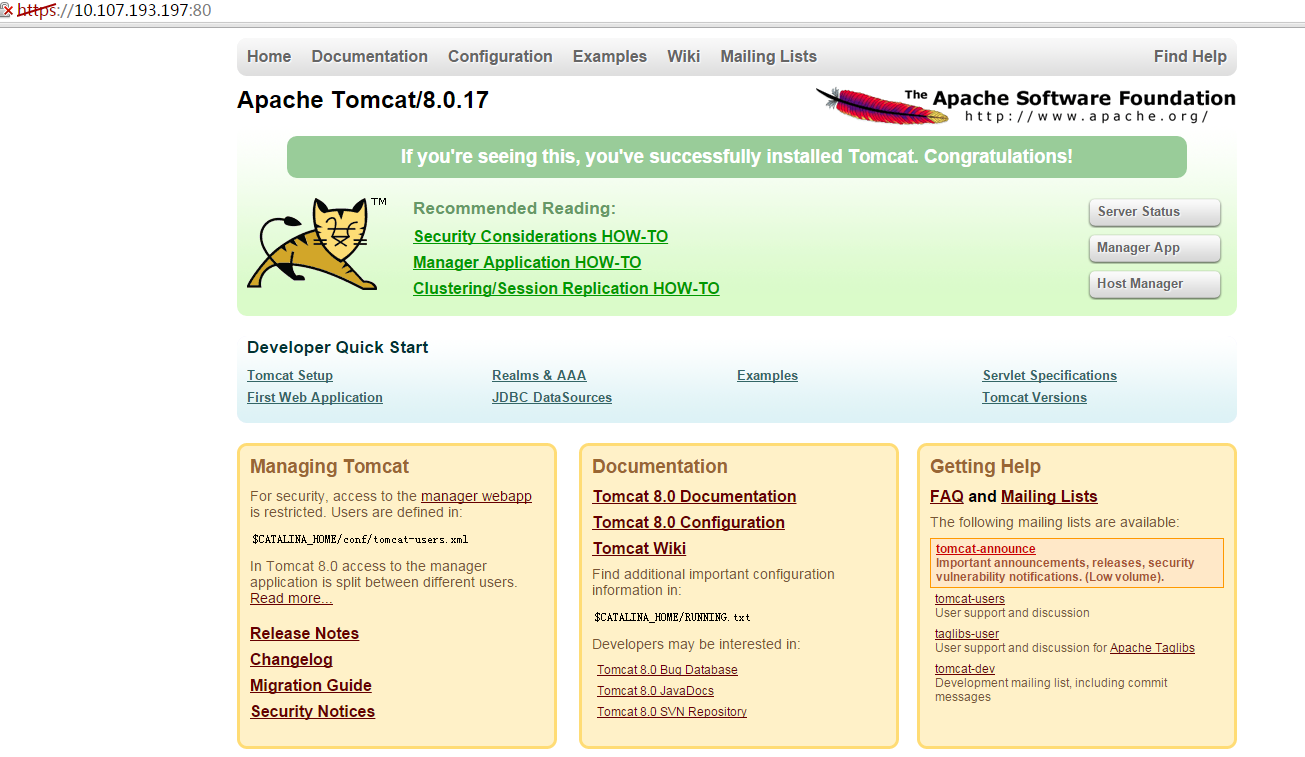
root@ubuntu:~/work/ssl# docker run –p 80:8080 tomcat:https

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

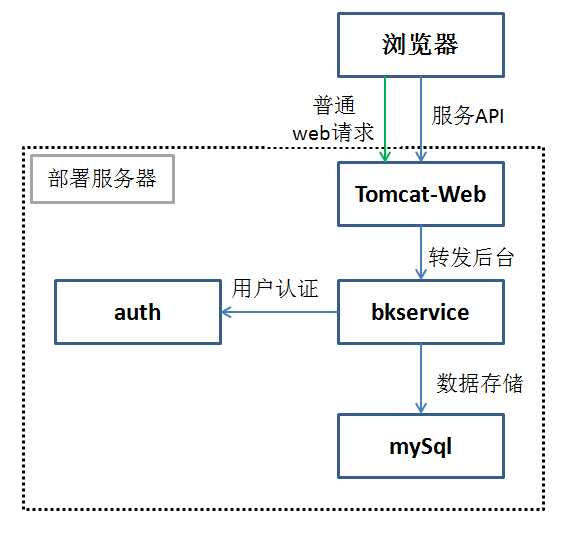
当然如果是动态挂载的，还需要加上-v参数。如果以上步骤部署成功，我们可以通过Chrome等浏览器访问该站点。

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

从上图可以看到，我们至少有四个功能上非常独立的模块，他们可以不用关心对方在什么地方，什么环境，以什么样的方式运行，只要能提供对应个API就可以一起工作。从组织结构上讲，也很独立。按照传统的开发部署模式，将不同的模块分给不同的组，然后各自为战，开发源码。等部署的时候，各自有自己的维护环境，部署的时候必须有四组人各自部署自己的服务，维护也一样。这样下来，部署，维护就成本很高。

如果把各个服务做成Docker镜像，并用docker-compose工具来管理工程，会大大简化工作。首先，将服务发布为镜像，服务所依赖的服务器环境就被封装在镜像中，Host上只需要能跑Docker即可。其次，使用docker-compose工具，可以简单的一次部署多个镜像服务。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镜像层存储

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。