用Docker部署一个Web应用

tonnie
编程 话题的优秀回答者
关注他

本文将以个人（开发）的角度，讲述如何使用Docker技术在线上单机模式下部署一个Web应用，如有错误欢迎指出。

上次在这篇文章提到了Docker，这次打算把这个坑展开来讲。

首先，什么是Docker?根据官网描述，我们可以得知，Docker是一个软件/容器平台，使用了虚拟化技术(cgroups，namespaces)来实现操作系统的资源隔离和限制，对于开发人员来说，容器技术为应用的部署提供了沙盒环境，我们可以在独立的容器运行和管理应用程序进程，Docker提供的抽象层使得开发人员之间可以保持开发环境相对的一致，避免了冲突。

下面体验下Docker的使用：

使用下面的shell命令安装Docker

```
$ curl -sSL https://get.docker.com/ | sh
```

安装成功后，使用下面的命令应该能显示Docker的版本信息，说明Docker已经被安装了

```
$ docker -v
Docker version 17.04.0-ce, build 4845c56
```

接着我们使用Docker创建一个nginx的容器:

```
$ docker run -d --name=web -p 80:80 nginx:latest
```

这条命令表示Docker基于nginx:alpine这个Docker镜像，创建一个名称为web的容器，并把容器内部的80端口与宿主机上的80端口做映射，使得通过宿主机80端口的流量转发到容器内部的80端口上。

使用docker ps命令，可以列出正在运行的容器，可以看到，刚才基于nginx镜像创建的容器已经处于运行状态了：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
a89d281829f9        nginx:latest        "nginx -g 'daemon ..."   2 minutes ago
```

现在访问宿主机地址的80端口，看到nginx的欢迎页面。

Docker容器本质上是一个运行的进程以及它需要的一些依赖，而Docker镜像则是定义这个容器的一个"模版"。

使用docker images能看到目前的镜像：

```
$ docker images
REPOSITORY
nginx                 latest              bedece1f06cc        10 minutes ago
```

了解到这个事实之后，我们使用下面的命令进入刚才创建的容器内部

```
$ docker exec -i -t web bash
```

现在处于的是容器内部的根文件系统(rootfs)，它跟宿主机以及其他容器的环境是隔离开的，看起来这个容器就是一个独立的操作系统环境一样。使用ps命令可以看到容器内正在运行的进程：

```
$ ps -l
PID   USER     TIME   COMMAND
    1 root       0:00 nginx: master process nginx -g daemon off;
    5 nginx      0:00 nginx: worker process
   23 root       0:00 ps -l
```

使用exit命令可以从容器中退出，回到宿主机的环境：

```
$ exit
```

使用docker inspect命令我们可以看到关于这个容器的更多详细信息：

```
$ docker inspect web
```

结果是用json格式表示的容器相关信息，拉到下面的**Networks**一列可以看到这个容器的网络环境信息：

```
"Networks": {
  "bridge": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "716496983db3eef8257dae57f4e0084c8242d8f5277da8a35b5ce265ccb4
    "EndpointID": "e3ab409f152e87594fe2f07e32cea2577983b352f3bba8cc99de6092682
    "Gateway": "172.17.0.1",
    "IPAddress": "172.17.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
```

```
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:11:00:02"
  }
}
```

内容显示了这个容器使用了bridge桥接的方式通信，它是docker容器默认使用的网络驱动（使用docker network ls可以看到所有的驱动），从上面可以看到这个容器的IP地址为172.17.0.2，网关地址为172.17.0.1。

现在回想刚才的例子，访问宿主机的80端口，宿主机是怎么跟容器打交道，实现转发通信的呢?

要解决这个问题，我们首先要知道，docker在启动的时候会在宿主机上创建一块名为docker0的网卡，可以用ifconfig查看：

```
$ ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:a4:e4:10:80
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1414 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1778 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:181802 (181.8 KB)  TX bytes:142440 (142.4 KB)
```

这个网卡的ip地址为172.17.0.1，看到这里你是否想起了刚才我们创建的容器使用的网关地址即为172.17.0.1? 我们是否可以大胆地猜测，docker容器就是通过这张名为docker0的网卡进行通信呢? 确实如此，以单机环境为例，Docker Daemon启动时会创建一块名为docker0的虚拟网卡，在Docker初始化时系统会分配一个IP地址绑定在这个网卡上，docker0的角色就是一个宿主机与容器间的网桥，作为一个二层交换机，负责数据包的转发。当使用docker创建一个容器时，如果使用了bridge模式，docker会创建一个vet对，一端绑定到docker0上，而另一端则作为容器的eth0虚拟网卡。

使用ifconfig也可以看到这个veth对的存在：

```
veth8231e5b Link encap:Ethernet  HWaddr 16:e8:f2:1d:e1:4d
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:29 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4033 (4.0 KB)  TX bytes:3741 (3.7 KB)
```

我找了一张图，可以很好地表示veth对的存在方式：

而真正实现端口转发的魔法的是nat规则。如果容器使用-p指定映射的端口时，docker会通过iptables创建一条nat规则，把宿主机打到映射端口的数据包通过转发到docker0的网关，docker0再通过广播找到对应ip的目标容器，把数据包转发到容器的端口上。反过来，如果docker要跟外部的网络进行通信，也是通过docker0和iptables的nat进行转发，再由宿主机的物理网卡进行处理，使得外部可以不知道容器的存在。

使用iptables -t nat命令可以看到添加的nat规则：

```
$ iptables -t nat -xvL
Chain PREROUTING (policy ACCEPT 376 packets, 21292 bytes)
    pkts      bytes target     prot opt in     out     source               des
   78606  4609864 DOCKER     all  --  any    any     anywhere             anywh

Chain INPUT (policy ACCEPT 376 packets, 21292 bytes)
    pkts      bytes target     prot opt in     out     source               des

Chain OUTPUT (policy ACCEPT 286 packets, 21190 bytes)
    pkts      bytes target     prot opt in     out     source               des
       0        0 DOCKER     all  --  any    any     anywhere            !127.(

Chain POSTROUTING (policy ACCEPT 290 packets, 21430 bytes)
    pkts      bytes target     prot opt in     out     source               des
       0        0 MASQUERADE  all  --  any    !docker0  172.17.0.0/16        ar
       0        0 MASQUERADE  tcp  --  any    any     172.17.0.2           172.

Chain DOCKER (2 references)
    pkts      bytes target     prot opt in     out     source               des
       0        0 RETURN     all  --  docker0 any     anywhere             anyv
       4      240 DNAT       tcp  --  !docker0 any     anywhere             any
```

从上面的最后一行可以观察到流量转发到了172.17.0.2的80端口上，这个地址就是刚才创建容器使用的IP地址。

现在知道在刚才的例子中宿主机是怎么跟容器通信了吧，那么容器跟容器之间通信呢？类似地，也是通过这个docker0交换机进行广播和转发。

扯的有点多，开始进入正题，先写一个Web应用压压惊。

一般情况下，如果你要编写一个Web项目，你会做什么呢？反正对于我来说，如果我要写一个python web项目的话，我会先用virtualenv建立一个隔离环境，进入环境内，使用pip安装Django，最后用django-admin startproject创建一个项目，搞定。

但是如果用容器化的方式思考，我们大可直接借助于容器的隔离性优势，更好地控制环境和版本的隔离，通常情况下你都不需要再关心用pyenv，virtualenv这种方式来初始化python环境的了，一切交给docker来完成吧。

甚至把安装django这个步骤也省了，直接通过一句命令来拉取一个安装了django的Python环境的镜像。

```
$ docker pull django
```

现在通过这个镜像运行django容器，同时进入容器Shell环境：

```
$ docker run -it --name=app -p 808
```

在/usr/src这个目录下新建一个app目录，然后用django-admin命令新建一个django项目：

```
$ cd /usr/src
$ mkdir app
$ cd app
$ django-admin startproject django_app
```

然后使用下面的命令，在容器8000端口上运行这个应用：

```
$ python manage.py makemigartions
$ python manage.py migrate
$ python manage.py runserver 0.0.0.0:8000 &
```

由于之前已经将容器的8000端口与宿主机的8080端口做了映射，因此我们可以通过访问宿主机的8080端口访问这个应用。

```
$ exit #退出容器
$ curl -L http://127.0.0.1:8080/
```

注意了，对这个容器的所有修改仅仅只对这个容器有效，不会影响到镜像和基于镜像创建的其他容器，当这个容器被销毁之后，所做的修改也就随之销毁。

下面新建一个应用ping，作用是统计该应用的访问次数，每次访问页面将次数累加1，返回响应次数给前端页面，并把访问次数存到数据库中。

使用redis作为ping的数据库，与之前类似，拉取redis的镜像，运行容器。

```
$ docker pull redis
$ docker run -d --name=redis -p 6379 redis
```

由于django容器需要与redis容器通信的话首先要知道它的ip地址，但是像刚才那样，每次都手工获取容器的ip地址显然是一件繁琐的事情，于是我们需要修改容器的启动方式，加入—link参数，建立django容器与redis容器之间的联系。

删除掉之前的容器，现在重新修改django容器的启动方式：

```
$ docker run -it --name=app -p 8080:8000 -v /code:/usr/src/app --link=redis:db
```

这次加入了两个参数：

- ***-v /code:/usr/src/app*** 表示把宿主机上的/code目录挂载到容器内的/usr/src/app目录，可以通过直接管理宿主机上的挂载目录来管理容器内部的挂载目录。

- ***--link=redis:db*** 表示把redis容器以db别名与该容器建立关系，在该容器内以db作为主机名表示了redis容器的主机地址。

现在进入到django容器，通过ping命令确认django容器能访问到redis容器：

```
$ ping db
PING db (192.168.32.12): 56 data bytes
64 bytes from 192.168.32.12: icmp_seq=0 ttl=64 time=0.463 ms
64 bytes from 192.168.32.12: icmp_seq=1 ttl=64 time=0.086 ms
```

像之前一样，建立一个项目，接着使用djan

```
$ django-admin startapp ping
```

编写ping的视图，添加到项目的urls.py：

```
from django.shortcuts import render
from django.http import HttpResponse
import redis

rds = redis.StrictRedis('db', 6379)

def ping(request):
    rds.incr('count', 1)
    cnt = rds.get('count')
    cnt = b'0' if cnt is None else cnt
    return HttpResponse(cnt.decode())

''' urls.py
from pingtest.views import ping

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', ping)
]
'''
```

别忘了安装redis的python驱动：

```
$ pip install redis
```

运行django应用，访问应用的根地址，如无意外便能看到随着页面刷新累加的数字。

```
$ curl http://127.0.0.1/
3243
$ curl http://127.0.0.1/
3244
```

你或许会想，每次创建一个容器都要手工做这么多操作，好麻烦，有没有更方便的方式地来构建容器，不需要做那么多额外的环境和依赖安装呢？

仔细一想，其实我们创建的容器都是建立在基础镜像上的，那么有没有办法，把修改好的容器作为基础镜像，以后需要创建容器的时候都使用这个新的镜像呢？当然可以，使用docker commit [CONTAINER]的方式可以将改动的容器导出为一个Docker镜像。

当然，更灵活的方式是编写一个Dockerfile来构建镜像，正如Docker镜像是定义Docker容器的模版，Dockerfile则是定义Docker镜像的文件。下面我们来编写一个Dockerfile，以定义出刚才我们进行改动后的容器导出的镜像。

下面加入supervisor和gunicorn以更好地监控和部署应用进程：

gunicorn的配置文件：

```
import multiprocessing

bind = "0.0.0.0:8000"
workers = multiprocessing.cpu_count() * 2 + 1
worker_class = 'meinheld.gmeinheld.MeinheldWorker'
user = 'root'
loglevel = 'warning'
reload = True
```

```
accesslog = '-' #to supervisord's stdout
#errorlog
```

supervisord的配置文件：

```
[supervisord]
nodaemon=true
logfile_maxbytes=10MB
loglevel=debug

[program:ping]
command=gunicorn -c /etc/gunicorn_conf.py django_app.wsgi:application
directory=/usr/src/app
user=root
process_name=root
numprocs=1
autostart=true
autorestart=true
redirect_stderr=True
```

以supervisord作为web应用容器的启动进程，supervisord来管理gunicorn的进程。这里说明一下的是，由于使用docker logs命令来打印容器的日志时默认是从启动进程（supervisord）的stdout和stderr里收集的，而gunicorn又作为supervisord的派生进程存在，因此要正确配置gunicorn和supervisord的日志选项，才能从docker logs中看到有用的信息。

把上面所做的修改混杂在一起，终于得出了第一个Dockerfile：

```
FROM django:latest

COPY ./app /usr/src/app

COPY supervisord.conf /etc/supervisord.conf

COPY gunicorn_conf.py /etc/gunicorn_conf.py

RUN apt-get update && \
    apt-get install -y supervisor && \
    rm -rf /var/lib/apt/lists/*

RUN pip install meinheld && \
    pip install gunicorn && \
    cd /usr/src/app && \
    pip install -r requirement.txt && \
    python manage.py makemigrations && \
    python manage.py migrate

WORKDIR /usr/src/app

CMD supervisord -c /etc/supervisord.conf
```

上面的Dockerfile的说明如下：

- FROM指令制定了该镜像的基础镜像为django:latest。
- 三行COPY指令分别将宿主机的代码文件和配置文件复制到容器环境的对应位置。
- 接着两行RUN指令，一条指令安装supervisor，另一条指令安装python的依赖以及初始化django应用。

- 最后运行supervisord，配置为刚才复制的supervisor的配置文件。

上面每一条指令都会由docker容器执行然后提交为一个镜像，叠在原来的镜像层的上方，最后得到一个拥有许多镜像层叠加的最终镜像。

完成Dockerfile的编写后，只需要用docker build命令就能构建出一个新的镜像：

```
docker build -t test/app .
```

接着就可以根据这个镜像来创建和运行容器了：

```
$ docker run -d --name=app -p 8080:8000 -v /code:/usr/src/app --link=redis:db
```

目前为止，项目的应用结构图如下：

现在，如果Redis这个节点出现故障的话会怎么样?

答案是，整个服务都会不可用了，更糟糕的是，数据备份和恢复同步成为了更棘手的问题。

很明显，我们不能只依赖一个节点，还要通过建立主从节点防止数据的丢失。再创建两个redis容器，通过slaveof指令为Redis建立两个副本。

```
$ docker run -d --name=redis_slave_1 -p 6380:6379 --link=redis:master redis re
$ docker run -d --name=redis_slave_2 -p 6381:6379 --link=redis:master redis re
```

现在写入到Redis主节点的数据都会在从节点上备份一份数据。

现在看起来好多了，然而当Redis master挂掉之后，服务仍然会变的不可用，所以当master宕机时还需要通过选举的方式把新的master节点推上去（故障迁移），Redis Sentinel正是一个合适的方式，我们建立Sentinel集群来监控Redis master节点，当master节点不可用了，再由Sentinel集群根据投票选举出slave节点作为新的master。

下面为Sentinel编写Dockerfile，在redis镜像的基础上作改动：

```
FROM redis:latest

COPY run-sentinel.sh /run-sentinel.sh

COPY sentinel.conf /etc/sentinel.conf

RUN chmod +x /run-sentinel.sh

ENTRYPOINT ["/run-sentinel.sh"]
```

Sentinel的配置文件：

```
port 26379

dir /tmp

sentinel monitor master redis-master 6379 2

sentinel down-after-milliseconds master 30000

sentinel parallel-syncs master 1

sentinel failover-timeout master 180000
```

run-sentinel.sh：

```
#!/bin/bash

exec redis-server /etc/sentinel.conf --sentinel
```

构建出Sentinel的镜像文件，容器运行的方式类似于redis：

```
$ docker run -d --name=sentinel_1 --link=redis:redis-master [build_sentinel_ima
$ docker run -d --name=sentinel_2 --link=redis:redis-master [build_sentinel_ima
$ docker run -d --name=sentinel_3 --link=redis:redis-master [build_sentinel_ima
```

这下Sentinel的容器也搭建起来了，应用的结构图如下：

简单验证一下当redis主节点挂掉后sentinel怎么处理：

```
$ docker pause redis-master
$ docker logs -f --tail=100 sentinel_1
1:X 17 Apr 14:32:51.633 # +sdown master master 192.168.32.12 6379
1:X 17 Apr 14:32:52.006 # +new-epoch 1
1:X 17 Apr 14:32:52.007 # +vote-for-leader 35ff9e1686f3425f4cbe5680a741e366a18
1:X 17 Apr 14:32:52.711 # +odown master master 192.168.32.12 6379 #quorum 3/2
1:X 17 Apr 14:32:52.711 # Next failover delay: I will not start a failover bef
1:X 17 Apr 14:32:53.221 # +config-update-from sentinel 35ff9e1686f3425f4cbe568
1:X 17 Apr 14:32:53.221 # +switch-master master 192.168.32.12 6379 192.168.32.
1:X 17 Apr 14:32:53.221 * +slave slave 192.168.32.6:6379 192.168.32.6 6379 @ ma
1:X 17 Apr 14:32:53.221 * +slave slave 192.168.32.12:6379 192.168.32.12 6379 @
$ docker exec -it sentinel_1 redis-cli -p 26379 info Sentinel
# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
sentinel_simulate_failure_flags:0
master0:name=master,status=ok,address=192.168.32.5:6379,slaves=2,sentinels=3
```

修改代码用Sentinel获取redis实例：

```
sentinel = Sentinel([('sentinel', 26379)])
rds = sentinel.master_for('master')
```

下面再来考虑这种情况：

假设我们对django_app容器进行伸缩，扩展出三个一模一样的django应用容器，这时候怎么办，该访问哪个？显然，这时候需要一个负载均衡的工具作为web应用的前端，做反向代理。

nginx是一个非常流行的web服务器，用它完成这个当然没问题，这里不说了。

下面说一说个人尝试过的两种选择：

LVS（Linux Virtual Server）作为最外层的服务，负责对系统到来的请求做负载均衡，转发到后端的服务器（Real Server）上，DR（Direct Route）算法是指对请求报文的数据链路层进行修改mac地址的方式，转发到后端的一台服务器上，后端的服务器集群只需要配置和负载均衡服务器一样的虚拟IP（VIP），请求就会落到对应mac地址的服务器上，跟NAT模式相比，DR模式不需要修改目的IP地址，因此在返回响应时，服务器可以直接将报文发送给客户端，而无须转发回负载均衡服务器，因此这种模式也叫做三角传输模式。

Haproxy是一个基于TCP/HTTP的负载均衡工具，在负载均衡上有许多精细的控制。下面简单地使用Haproxy来完成上面的负载均衡和转发。

首先把haproxy的官方镜像下载下来：

```
$ docker pull haproxy
```

这类的镜像的Dockerfile都可以在Docker Hub上找到。

这次同样选择编写Dockerfile的方式构建自定的haproxy镜像：

```
FROM haproxy:latest

COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
```

暂时只需要把配置文件复制到配置目录就可以了，因为通过看haproxy的Dockerfile可以看到最后有这么一行，于是乎偷个懒~

```
CMD ["haproxy", "-f", "/usr/local/etc/haproxy/haproxy.cfg"]
```

haproxy的配置文件如下：

```
global
    log 127.0.0.1 local0
    maxconn 4096
    daemon
    nbproc 4
```

```
defaults
    log 127.0.0.1 local3
    mode http
    option dontlognull
    option redispatch
    retries 2
    maxconn 2000
    balance roundrobin
    timeout connect 5000ms
    timeout client 5000ms
    timeout server 5000ms

frontend main
    bind *:6301
    default_backend webserver

backend webserver
    server app1 app1:8000 check inter 2000 rise 2 fall 5
    server app2 app2:8000 check inter 2000 rise 2 fall 5
    server app3 app3:8000 check inter 2000 rise 2 fall 5
```

这里的app即web应用容器的主机名，运行haproxy容器时用link连接三个web应用容器，绑定到宿主机的80端口。

```
$ docker run -d --name=lb -p 80:6301 --link app1:app1 --link app2:app2 --link a
```

这时候访问宿主机的80端口后，haproxy就会接管请求，用roundrobin方式轮询代理到后端的三个容器上，实现健康检测和负载均衡。

现在又有一个问题了，每次我们想增加或者减少web应用的数量时，都要修改haproxy的配置并重启haproxy，十分的不方便。

理想的方式是haproxy能自动检测到后端服务器的运行状况并相应调整配置，好在这种方式不难，我们可以使用etcd作为后端服务器的服务发现工具，把服务器的信息写入到etcd的数据库中，再由confd来间隔一段时间去访问etcd的api，将
件以及重启haproxy进程。

按官方的说法，etcd是一个可靠的分布式的KV存储系统，而confd则是使用模版和数据管理应用配置的一个工具，关于他俩我还没太多了解，所以不多说，下面把他们集成到上面的应用中。

创建一个etcd的容器：

```
docker run -d \
-e CLIENT_URLS=http://0.0.0.0:2379 \
-e PEER_URLS=http://0.0.0.0:2380 \
-p 2379:2379 \
-p 2380:2380 \
-p 4001:4001 \
-p 7001:7001 \
-v /etc/ssl/certs/:/etc/ssl/certs/ \
elcolio/etcd \
-name etcd \
-initial-cluster-token=etcd-cluster-1 \
-initial-cluster="etcd=http://etcd:2380"\
-initial-cluster-state=new \
-advertise-client-urls=http://etcd:2379 \
-initial-advertise-peer-urls http://etcd:2380
```

confd的处理比较简单，把confd的二进制文件和配置文件集成到之前haproxy的Dockerfile中：

```
FROM haproxy:latest

COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg

COPY confd .

RUN chmod +x confd

COPY haproxy.toml /etc/confd/conf.d/

COPY haproxy.tmpl /etc/confd/templates/

COPY boot.sh .

COPY watcher.sh .

CMD ["./boot.sh"]
```

通过之前haproxy的配置文件创建出新的模版文件，修改backend的配置，加入模版指令，表示confd从etcd的前缀为/app/servers的所有key中获取键值对，作为server的key的value，逐条追加到配置文件中去：

```
backend webserver
    {{range gets "/app/servers/*"}}
    server {{base .Key}} {{.Value}} check inter 2000 rise 2 fall 5
    {{end}}
```

下面是confd的配置文件：

```
[template]
src = "haproxy.tmpl"
dest = "/usr/local/etc/haproxy/haproxy.cfg"
keys = [
    "/app/servers"
]
owner = "root"
mode = "0644"
reload_cmd = "kill -HUP 1"
```

confd会把数据填入上面的模版文件，并把配置更新到haproxy配置的目标路径，再使用reload_cmd指定的命令重启haproxy。

修改后的haproxy镜像最后通过boot.sh启动进程：

```
#!/bin/bash

./watcher.sh &

exec /docker-entrypoint.sh haproxy -f /usr/local/etc/haproxy/haproxy.cfg
```

watcher.sh启动了confd间隔一段时间去访问etcd的地址，检查是否有更新：

```
./confd -interval 10 -node http://etcd:2379 -config-file /etc/confd/conf.d/hap
```

启动haproxy时建立与etcd容器间的连接：

```
$ docker run -d --name=lb -p 80:6301 --link app1:app1 --link app2:app2 --link a
```

下面通过调用etcd的api在/app/servers上新建一个服务器节点：

```
$ docker exec -it etcd etcdctl set /app/servers/app1 172.17.0.5:8000
```

观察haproxy容器的日志，可以看到配置被更新了：

```
$ docker logs -f lb
2017-04-16T16:24:00Z 78745b65a3d4 ./confd[7]: INFO Backend set to etcd
<7>haproxy-systemd-wrapper: executing /usr/local/sbin/haproxy -p /run/haproxy.p
2017-04-16T16:24:00Z 78745b65a3d4 ./confd[7]: INFO Starting confd
2017-04-16T16:24:00Z 78745b65a3d4 ./confd[7]: INFO Backend nodes set to http://
2017-04-16T16:24:00Z 78745b65a3d4 ./confd[7]: INFO /usr/local/etc/haproxy/hapro
2017-04-16T16:24:00Z 78745b65a3d4 ./confd[7]: INFO Target config /usr/local/etc
2017-04-16T16:24:00Z 78745b65a3d4 ./confd[7]: INFO Target config /usr/local/etc
<5>haproxy-systemd-wrapper: re-executing on SIGHUP.
<7>haproxy-systemd-wrapper: executing /usr/local/sbin/haproxy -p /run/haproxy.p
```

最终的应用结构图如下：

运行在机器上的服务时刻有可能有意外发生，因此我们需要一个服务来监控机器的运行情况和容器的资源占用。netdata是服务器的一个实时监测工具，利用它可以直观简洁地了解到服务器的运行情况。

当docker镜像和容器数量增多的情况下，手工去运行和定义docker容器以及其相关依赖无疑是非常繁琐和易错的工作。Docker Compose是由Docker官方提供的一个容器编排和部署工具，我们只需要定义好docker容器的配置文件，用compose的一条命令即可自动分析出容器的启动顺序和依赖，快速的部署和启动容器。

下面编写好compose的文件：

```
version: '2.1'
services:
  haproxy:
    container_name: lb
    build: ./builds/haproxy
    ports:
     - 80:6301
    restart: always
    links:
    - ping-app
    - etcd:etcd
    - netdata:netdata

  ping-app:
    build: .
    restart: always
    volumes:
    - ./app:/usr/src/app
    links:
    - sentinel
    - redis-master:db

  redis-master:
    image: redis:latest
    ports:
     - 6379:6379
    restart: always
```

```
  redis-slave:
    image: redis:latest
    command: redis-server --slaveof master 6379
    restart: always
    links:
    - redis-master:master

  sentinel:
    build: ./builds/sentinel
    restart: always
    links:
    - redis-master:redis-master
    - redis-slave

  etcd:
    image: elcolio/etcd
    command: -name etcd -initial-cluster-token=etcd-cluster-1 -initial-cluster-
    environment:
     - CLIENT_URLS=http://0.0.0.0:2379
     - PEER_URLS=http://0.0.0.0:2380
    ports:
     - 2379:2379
     - 2380:2380
     - 4001:4001
     - 7001:7001
    volumes:
     - /etc/ssl/certs/:/etc/ssl/certs/

  netdata:
    image: titpetric/netdata
    restart: always
    cap_add:
     - SYS_PTRACE
    volumes:
     - /proc:/host/proc:ro
     - /sys:/host/sys:ro
```

只需几条命令，就能启动和伸缩容器：

```
docker-compose -f docker-compose.yml up -d
docker-compose -f docker-compose.yml scale redis-slave=2
docker-compose -f docker-compose.yml scale sentinel=3
docker-compose -f docker-compose.yml scale ping-app=3
```

再通过一个脚本把web应用注册到etcd中去：

```
APP_SERVERS=$(docker-compose -f docker-compose.yml ps ping-app | awk '{print $
APP_SERVER_PORT=8000
INDEX=1
for ETCD_NODE in ${APP_SERVERS//\s/};
do
    docker-compose -f docker-compose.yml exec etcd etcdctl set /app/servers/ap
    INDEX=$(expr $INDEX + 1)
done
```

就这样，一个基于Docker构建并具有良好可用性的web应用就完成了。

编辑于 2017-04-18

编程 程序员 Docker

文章被以下专栏收录

四条鱼 关注专栏

推荐阅读

微服务为什么一定要上Docker?

慕容千语

入门系列之Kubernetes部署

腾讯云技术... 发表于腾讯云+社...

这4种云部署模式，说明已成为数字化社会中枢

云计算一切技术发展都是为弹性、安全可靠、低成本低简单自动化服务！云计算总未来，几乎所有的互联网应业应用都将承载于云上。A阿里云、百度、Google、

东南风

29 条评论 切换为时间排序

写下你的评论...

精选评论（1）

tonnie (作者) 1 年前

PS：我把示例代码放到了t.cn/RXxcPCY，可以下载下来运行脚本直接构建运行

4

评论（29）

AkaShiro 1 年前

这和IIS是一类吗

赞

充气 回复 AkaShiro 1 年前

这是nginx....

赞

吖猩 1 年前

写的很好，受教了

赞

Biao 1 年前

有意思。

赞

Alterac 1 年前

赞啊

赞

tonnie (作者) 1 年前

PS：我把示例代码放到了t.cn/RXxcPC

▲ 赞同 319 ▼ 29 条评论 分享 收藏 ···

4

小树叶 1 年前

赞一个

赞

KS Sun 1 年前

有心了

赞

abc look 1 年前

6666

赞

孙潇野 1 年前

最近正好在学docker，感觉这个可以作为一条学习路线来学了！谢谢分享

赞

吴化吉 1 年前

请教个问题：docker exec -i -t nginx bash 进去nginx环境之后，执行ps命令查看进程，但是ps命令没有啊？首先需要安装ps么，安装ps要更新软件源不然在国内太卡了，要更新软件源就要编辑sources.list文件，这时候发现vim也没有

2

潘可涵 回复 吴化吉 8 个月前

需要更新但是不用编辑source.list, apt-get update && apt-get install procps 就好了

赞

春哥大魔王 1 年前

有意思，关键点是服务注册发现和负载均衡

赞

Jason 1 年前

赞，不知答主有没有关注k8s，期待这块的文章

赞

一知半解 1 年前

mark一下

赞

Claymore 1 年前

很有心，谢谢

赞

Claymore 1 年前

实践中遇到个问题：-v /code:/usr/src/app 表示把宿主机上的/code目录挂载到容器内的/usr/src/app目录，可以通过直接管理宿主机上的挂载目录来管理容器内部的挂载目录。这里/code是我们的主机目录，一般是空的，挂载到容器内就会冲掉app里的文件啊，导致/code和/usr/src/app都是空的。求指点

赞

李广成 1 年前

从一个单机app，衍生到集群，负载均衡，全部使用docker搭建，真的很不错，学习学习

赞

李广成 1 年前

我感觉发现了一个宝，十分感谢作者深入浅出的引申出web应用的周边配置，很全

赞

李广成 1 年前

你好，我使用

sudo docker run -it --net:host --name=app -p 8080:8000 -v /home/lgc/pro/django/code/:/usr/src/app --link=redis:db django启动一个容器，进入容器之后，无法ping通一些地址，比方我要apt-get update 就不行，但是ping通宿主机是可以的。这个如何解决呢

赞

许辉 回复 **李广成** 4 个月前

请使用overlay网络进行service通讯 不要使用--link

赞

vincen 1 年前

厉害了我的哥

赞

1 2 下一页