搭建高性能分布式docker系统

一、目录

1、docker环境及其网络

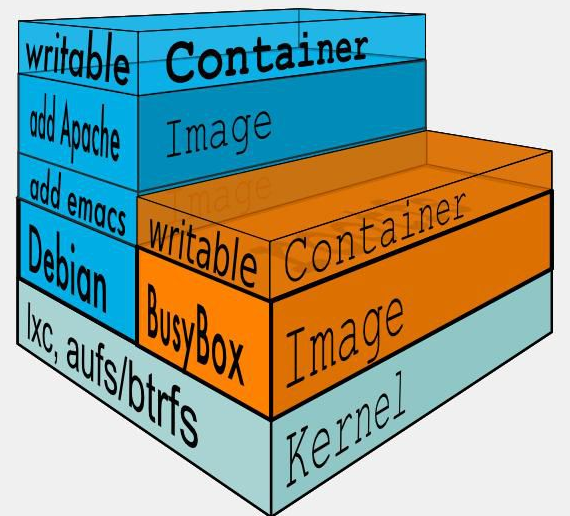
2、搭建分布式系统

3、常用命令

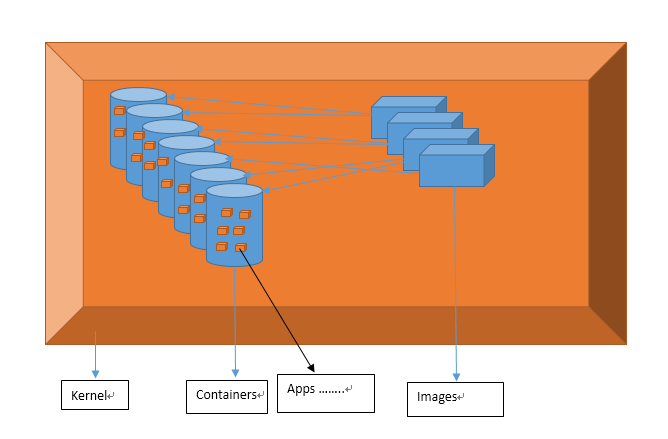
二、内容

1、docker环境及其网络

关于docker，在任何搜索引擎上随便一搜就会发现很多关于docker的介绍。这里仅稍作介绍。docker是虚拟机吗，是容器吗.... 都不是，docker是一种虚拟化解决方案，或者说是一种容器引擎，一种轻量级的虚拟化方案。类似下面这张图的相信很多人都看过：



其实这张图有点混乱，尤其是对刚刚学习docker的人来讲容易混淆一些概念，比如kernel，image，container等等.... 我重新画一张图，应该是这样的()：



它的轻量级体现在哪里，能看到它的kernel那一层实际上是公用的，也就是在OS层面是公共的，上面建立的所有container都公用一个kernel，这也是和一般的虚拟机不一样的地方（一般意义上来讲虚拟机是从上到下app到os都是相互独立的自成体系的系统）。同样，image也是遵从同样的思想，不同容器可以公用相同的image，这样节省了存储空间、便于快速部署（省的copy来copy去）、便于升级... 但是容器之间是完全隔离的。当在一个容器中做一些修改时并不会影响到其他容器，因为docker的修改是有一个layer的概念，也就是当你要写入时会新加一层layer来记录你的写入而不会直接在底层的layer上修改，所以多个容器公用一个image的时候相互之间是隔离的，他们的修改只会在自己的容器中生效。启动一个vm是要花费很大代价的，会占用大量的cpu内存等资源，但是启动一个docker容器是非常快的，几乎和启动一个进程似的，最重要的是占用资源少，这也是为什么docker能在这么短时间出名的最主要原因。理解docker重要的就是要理解它的层级部署、隔离性以及AUFS文件系统，基本概念不再赘述了。

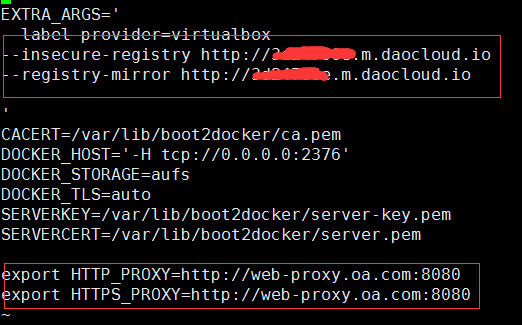
其实怎么搭建环境是取决于你要用docker来做什么，想要一个什么样的环境的，所以也没法一概而论，比如做web服务的，可能需要搭建一个docker集群，搭好的容器有做web服务的、有做负载均衡的、有做计算的、有提供cgi服务、有做log统计的等等等.....如果是整个部门中要搭建一套开发测试环境那重点可能就在于把镜像仓库做好，开发和测试分别搭建好自己本地的base环境，开发做好了就commit一下，测试直接把镜像pull下来本地重建个容器就OK了，不用把精力浪费在环境搭建上。如果是自己搭建一套只有自己使用的docker环境那就更简单了，一般根据不同操作系统在官网上都会有相应的安装包，如果是在windows系统上搭建自己的开发环境建议还是用virtualbox加boot2docker（后面例子将以此为例来进行）（安装时不要忘记要到BIOS里把虚拟化打开），Windows上装的docker，不要试图在宿主机上用apt-get yum等方式安装软件，因为宿主机是在Windows系统的一个虚拟主机，只有核心的几个内核命令以及git命令是可以使用的，其他的命令都相当于到Windows下去执行Linux命令一样，所以注定是失败的。

至于如何pull 镜像、如何自己制作image，如何commit修改过的容器等等这些基本操作不再赘述，可参考第5部分贴出的常用命令。

docker的镜像库是个非常丰富的image库，里面有来自世界各地的docker爱好者们提交的各种各样的镜像文件，但是悲剧的是被墙了，即使能连上pull的速度也是非常慢，下面针对我司网络状况介绍下网络配置方式。

首先，我们肯定是需要pull镜像的，所以这条路无论如何也要打通，否则就享受不到docker丰富多彩的image仓库了。（其实，自己用ftp上传需要安装的东西也是可以的，比如要自己安装一个ubuntu系统的话就用ftp工具上传你下载好的ubuntu包ubuntu-14.04-x86\_64.tar.gz，然后cat ubuntu-14.04-x86\_64.tar.gz |docker import - ubuntu:ubuntu14，但是肯定不如一个docker pull命令方便）。无论是开发网还是办公网要想直接使用docker hub是不可能的（当然也可使用我司的gaia，只是镜像实在太少...），国内目前也有不少镜像加速平台阿里云、daoCloud等等.... 所以可以使用本地代理配合镜像加速平台的方式来使用dockerhub。代理服务器可以在proxy\_devnet.pac里找个自己喜欢的，镜像加速可以到国内的加速平台申请一个（绝大部分是免费的）。

对于已经建好的docker环境只要在/var/lib/boot2docker/profile中加入以下内容：



然后在自己家目录.docker/machine/machines/yourmachinename/config.json中加入以下内容：



其中的http://\*\*\*.m.daocloud.io就是你自己的镜像加速器地址。

然后重启docker就可以了。

当然如果你是新建的，那可以这样，在新建时加上一个--engine-registry-mirror参数来改变一下镜像地址：

clipboard.png

代理按照上面的第一种方式添加。

然后重启docker。

然后就可以尽情的使用docker了。

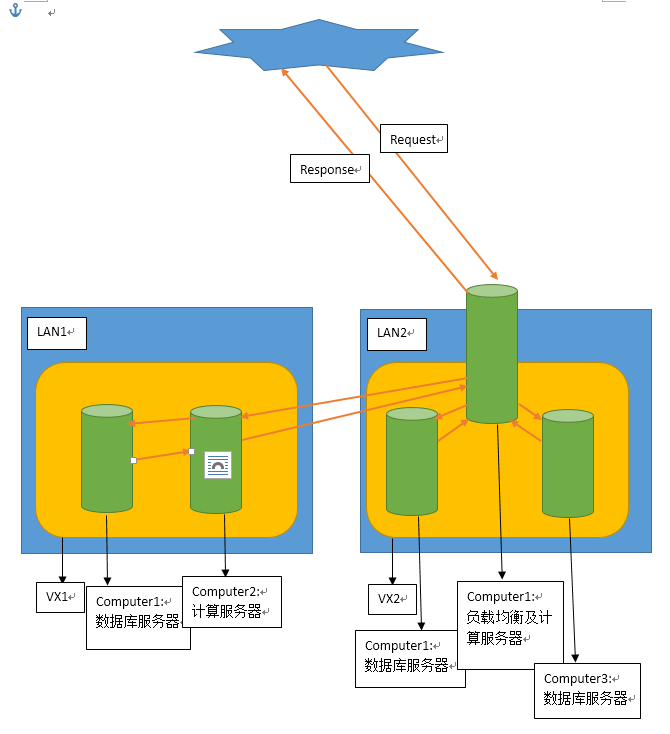
那么容器直接如何通信呢，不同宿主机中的容器如何通信呢，如何从外部访问容器呢。

同一个宿主机中的容器其实是相当于在同一个局域网中的，使用内网ip是可以直接相互连接的，所以如果只搭建一个宿主机的话只需要配置好容器与宿主机、宿主机与外界的网络就可以了，容器相互之间的网络直接用内网IP；但大多数情况是需要搭建一套集群环境，也就是需要N个宿主机，每个宿主机N个容器的情况。docker的网络配置主要有host、container、bridge以及借助第三方工具这几种方式。host模式其实就是借用宿主机的网络配置的意思，在新建一个容器时不会为这个容器虚拟出新的网卡、ip等，容器也不会获得自己的network namespace，所以也就不用使用NAT了，因为这时候它的网络和宿主机的就是同一个网络（但是也仅限网络，容器的其他资源比如文件系统等都还是独立的）； container模式就是和一个已存在的container共享网络，这时候容器也不会创建新的网络而是和这个已存在的容器共享ip 端口等等，此时容器之间可以通过网卡IO直接通信（同样也是遵从容器的隔离性的）；bridge是docker默认的方式也是平时用的最多的，此模式会为每一个容器分配 network namespace、设置 IP 等，并将一个主机上的 Docker 容器连接到一个虚拟网桥上。当 Docker server 启动时，会在主机上创建一个名为 docker0 的虚拟网桥，此主机上启动的 Docker 容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中了。这样就可以在容器内自己分配IP和组网了；另外还有一些第三方的工具比如weave之类的可以很方便的建立起需要的网络来，但是目前来看虽然有些工具配置时方便但是网络效率却是不高的（个人感觉它把docker命令封装这一点也会让人用起来不那么方便，尤其是对于很多习惯了docker原始命令的人来讲）。当然肯定也还有些其他方式也可以完成这种配置比如把不同宿主机加到自己路由表中的这种方式也是可以达到相同效果的。下面结合第二部分主要利用bridge方式来搭建。

2、搭建分布式系统

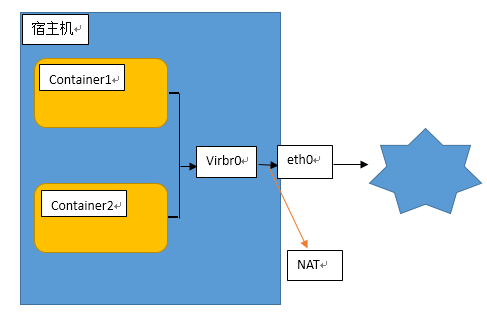
下面利用上面介绍的网桥的方式来搭建一套简单的分布式系统（暂不涉及：容灾、数据恢复、paxos等各种复杂算法以及zookeeper等各种开源框架的使用）

效果图如下：



其中LAN1和LAN2分别是两个docker局域网，由两个虚拟机构成，其中的每一个服务器都是一个container。

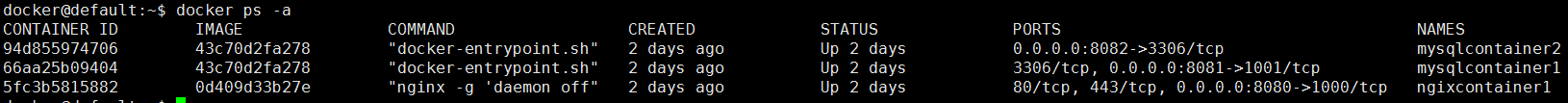
其中容器与宿主机以及与外界的连接如下所示：

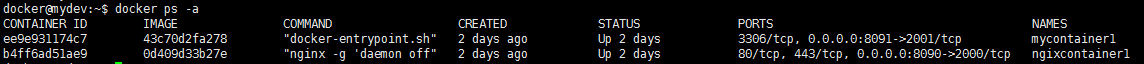


default代表Lan2，mydev代表Lan1:

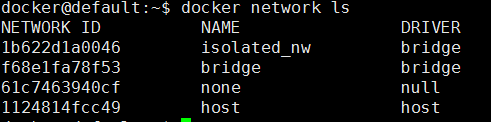


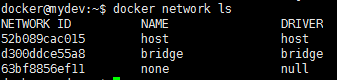
default和mydev的内容如下：

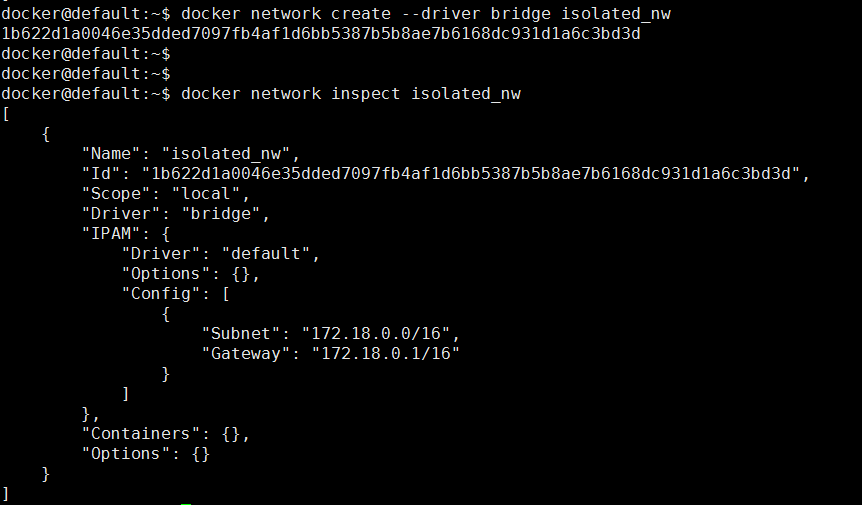




网络内容如下：

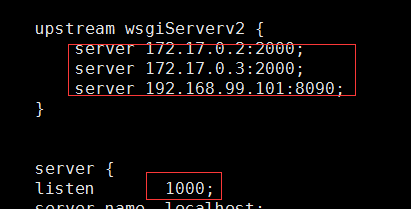






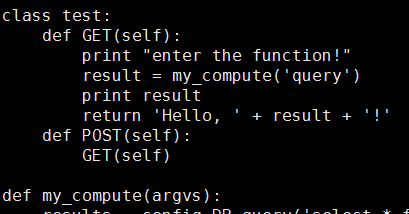
首先，default中将宿主机的8080端口映射到容器的1000端口，在nginx中监听1000端口，起一个python进程来做简单的计算和log记录，nginx收到的请求也会转发到这个python进程上来。然后用wsgi转发到三个server上做下一步计算处理。

nginx监听及转发配置：

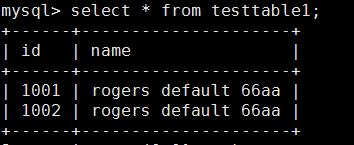


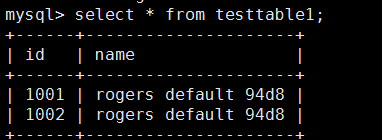
因为VX2中的两个server是在同一个LAN中的，所以相当于是局域网内的转发，直接使用内网地址就可以了，这样还可以减少一次网卡间的转发和NAT转换；最后一个地址因为是在另一个LAN中的，所以要用外网地址端口映射的方式来转发请求。

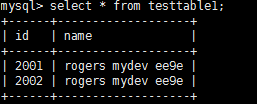
python进程做的事情也很简单，如下所示：



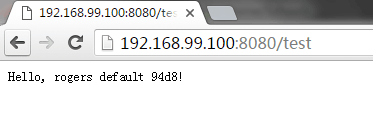
几个数据库server的存储内容如下：

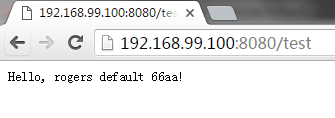


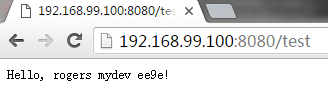




然后，把各个server启动，用浏览器就可以访问到搭建起来的系统了，当不停的刷新页面时就会看到来自不同server的数据了：







目前只是用docker来做简单的分布式计算和web服务，真实的server集群能做的事情它都能做，所以对于服务部署或数据挖掘或机器学习（后续再用docker集群来做机器学习的处理）或云平台服务来讲也是节约成本和提高效率的一个有效途径。上文也提到过，整个团队的开发、测试、生产环境的部署都可以在docker下进行（比如可以使用Docker＋Kubernetes＋Jenkins来实现），不用将精力浪费在繁杂的环境搭建上，一个部门搭建好一个基础的环境后制作成一个base image，然后各个部门可以基于此来构建自己的开发环境，部署好的运行系统也可以打包成image，测试只要把它pull下来新建container就可以跑测试用例了。大家都知道云平台要求按照需求伸缩扩展和按模型付费，其中的各个用户的实例可能底层服务是相似的，但上层服务与资源是千差万别的，并且各个用户之间是完全隔离的，这与docker的理念完全吻合，可以轻而易举的将一台真实的server按不同需求拆分为N个不同功能的server服务于广大用户，并且回收和新建都是非常便捷的，这也是docker的魅力所在吧。

3、常用命令

docker cmd:

docker version 查看docker版本

docker info 显示docker系统的信息

docker images 列出镜像列表

docker search image\_name 检索image

docker pull image\_name 下载image

docker rmi image\_name 删除一个或者多个镜像

docker history image\_name 显示一个镜像的历史

docker run -i -t image\_name /bin/bash 启动

docker run image\_name apt-get install -y app\_name 在容器中安装新的程序

docker ps -a 列出所有的container

docker ps 列出当前所有正在运行的container

docker ps -l 列出最近一次启动的container

docker rm e22aa73710fb 删除容器

docker commit ID new\_image\_name 保存对容器的修改

$docker rm `docker ps -a -q` 删除所有容器

$docker rm Name/ID 删除单个容器

$docker stop Name/ID 停止一个容器

$docker start Name/ID 启动一个容器

$docker kill Name/ID 杀死一个容器

$docker logs Name/ID 从一个容器中取日志

$docker diff Name/ID 列出一个容器里面被改变的文件或者目录，list列表会显示出三种事件，A 增加的，D 删除的，C 被改变的

$docker top Name/ID 显示一个运行的容器里面的进程信息

$docker cp Name:/container\_path to\_path 从容器里面拷贝文件/目录到本地一个路径

$docker cp ID:/container\_path to\_path

$docker restart Name/ID 重启一个正在运行的容器

$docker attach ID 附加到一个运行的容器上面

$docker save image\_name -o file\_path 保存镜像到一个tar包;

$docker load -i file\_path 加载一个tar包格式的镜像

$docker save image\_name > /home/save.tar

$docker load < /home/save.tar 使用scp将save.tar拷到机器b上

docker push new\_image\_name 发布docker镜像

docker build -t image\_name Dockerfile\_path 根据dockerfile构建容器

sudo docker export <CONTAINER ID> > /home/export.tar 导出容器

docker network ls

docker network inspect bridge

route cmd:

sudo ifconfig docker0 172.17.1.1 netmask 255.255.255.0

sudo route add -net 172.17.0.0 netmask 255.255.255.0 gw 192.168.99.102
sudo iptables -t nat -F POSTROUTING
sudo iptables -t nat -A POSTROUTING -s 172.17.1.0/24 ! -d 172.17.0.0/16 -j MASQUERADE

sudo route add -net 172.17.1.0 netmask 255.255.255.0 gw 192.168.99.100
sudo iptables -t nat -F POSTROUTING
sudo iptables -t nat -A POSTROUTING -s 172.17.0.0/24 ! -d 172.17.0.0/16 -j MASQUERADE