这个 PDF 算是学习 docker 的一个小总结，所有文章摘自我在 csdn 的博客专栏：

http://blog.csdn.net/column/details/docker.html

第一章到第八章摘自 docker 官方网站，翻译了一些个人认为比较重要的文章，后面实战部分是自己一些实验的过程以及目前在公司部署 docker 过程中的一些感受。

个人水平有限，如果您发现问题，请直接发邮件给我 dwj_wz@163.com，或加 QQ 讨论群 341410255 我会认真回复您！

WaitFish

2014-09-03

注：

下文以黄色标记的内容是一些提示和注意事项。

以红色字体标注的都是一些需要执行的命令行。如：

```
root@ubuntudocker:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
58b043aa05eb        desk_hz:v1          "/startup.sh"       5 days ago          Up 2 seconds        5900/tcp, 6080/tcp, 22/tcp   yanlx
使用这个命令来查看当前运行的容器列
```

内容目录

Docker 学习手册-v1.0

一、为什么要使用 docker?

1、快速交付应用程序

- 开发者使用一个标准的 image 来构建开发容器，开发完成之后，系统管理员就可以使用这个容器来部署代码
- docker 可以快速创建容器，快速迭代应用程序，并让整个过程可见，使团队中的其他成员更容易理解应用程序是如何创建和工作的。
- docker 容器很轻！很快！容器的启动时间是次秒级的，节约开发、测试、部署的时间

2、更容易部署和扩展

- docker 容器可以在几乎所有的环境中运行，物理机、虚拟机、公有云、私有云、个人电脑、服务器等等。
- docker 容器兼容很多平台，这样就可以把一个应用程序从一个平台迁移到另外一个。

3、效率更高

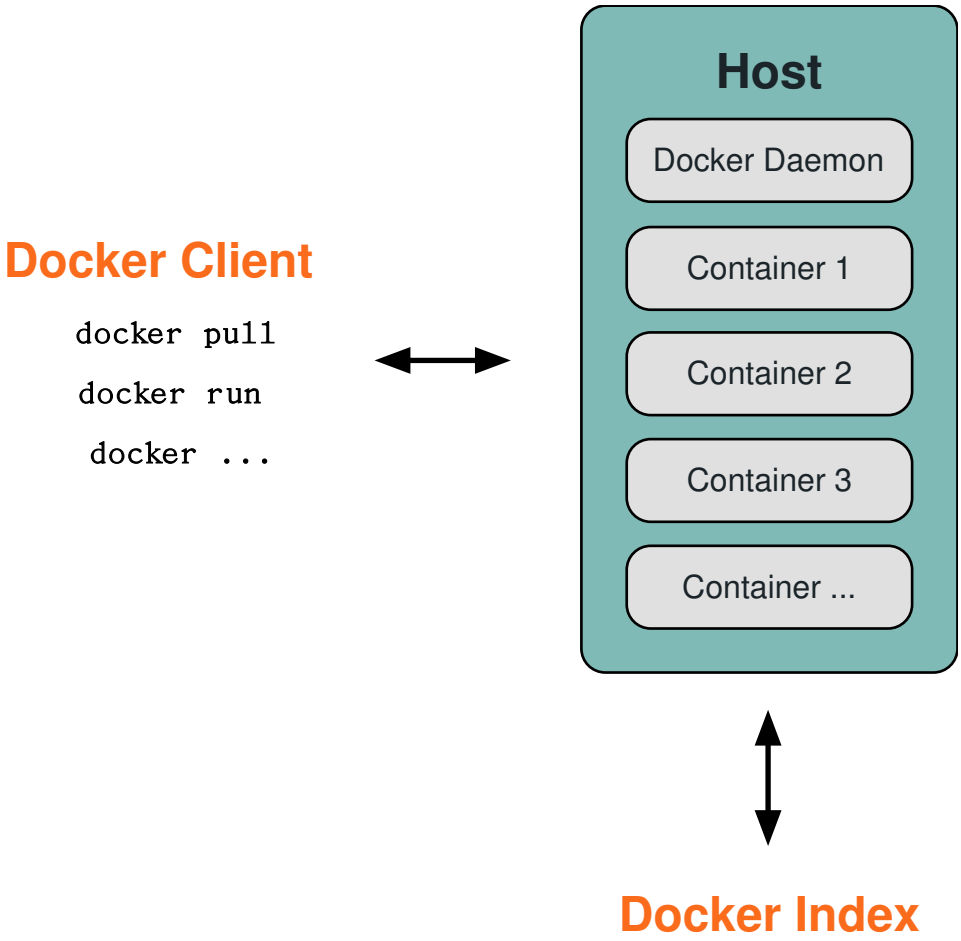
- docker 容器不需要 hypervisor，他是内核级的虚拟化。

4、快速部署也意味着更简单的管理

- 通常只需要小小的改变就可以替代以往巨型和大量的更新工作。

二、Docker 的体系结构

docker 使用 C/S 架构，docker daemon 作为 server 端接受 client 的请求，并处理（创建、运行、分发容器），他们可以运行在一个机器上，也通过 sockerts 或者 RESTful API 通信。



Docker daemon 一般在宿主主机后台运行，用户使用 client 而直接跟 daemon 交互。Docker client 以系统做 bin 命令的形式存在，用户用 docker 命令来跟 docker daemon 交互。

1、Docker 的内部组件

docker 有三个内部组件

- docker images
- docker registries
- docker containers

Docker images

docker images 就是一个只读的模板。比如：一个 image 可以包含一个 ubuntu 的操作系统，里面安装了

apache 或者你需要的应用程序。images 可以用来创建 docker containers，docker 提供了一个很简单的机制来创建 images 或者更新现有的 images，你甚至可以直接从其他人那里下载一个已经做好的 images

Docker registries

Docker registries 也叫 docker 仓库，它有公有仓库和私有仓库 2 种形式，他们都可以用来让你上传和下载 images。公有的仓库也叫 Docker Hub。它提供了一个巨大的 image 库可以让你下载，你也可以在自己的局域网内建一个自己的私有仓库。

Docker containers

Docker containers 也叫 docker 容器，容器是从 image 镜像创建的。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、安全的平台。

2、Docker image 的工作原理

每个 docker 都有很多层次构成，docker 使用 union file systems 将这些不同的层结合到一个 image 中去。

AUFS (AnotherUnionFS) 是一种 Union FS, 简单来说就是支持将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)的文件系统, 更进一步的理解, AUFS 支持为每一个成员目录(类似 Git Branch)设定 readonly、readwrite 和 whiteout-able 权限, 同时 AUFS 里有一个类似分层的概念, 对 readonly 权限的 branch 可以逻辑上进行修改(增量地, 不影响 readonly 部分的)。通常 Union FS 有两个用途, 一方面可以实现不借助 LVM、RAID 将多个 disk 挂到同一个目录下, 另一个更常用的就是将一个 readonly 的 branch 和一个 writeable 的 branch 联合在一起，Live CD 正是基于此方法可以允许在 OS image 不变的基础上允许用户在其上进行一些写操作。Docker 在 AUFS 上构建的 container image 也正是如此。

3、Docker 仓库

docker 仓库用来保存我们的 images，当我们创建了自己的 image 之后我们就可以使用 push 命令将它上传到公有或者私有仓库，这样下次要在另外一台机器上使用这个 image 时候，只需要从仓库上 pull 下来就可以了。

4、Docker 容器

当我们运行 docker run -i -t ubuntu /bin/bash 命令时，docker 在后台运行的操作如下：

- 如果本地有 ubuntu 这个 image 就从它创建容器，否则从公有仓库下载
- 从 image 创建容器
- 分配一个文件系统，并在只读的 image 层外面挂载一层可读写的层
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去

- 从地址池配置一个 ip 地址给容器
- 执行你指定的程序，在这里启动一个/bin/bash 进程
- -i -t 指定标准输入和输出

5、Docker 底层技术

docker 底层的 2 个核心技术分别是 Namespaces 和 Control groups

以下内容摘自 InfoQ Docker，自 1.20 版本开始 docker 已经抛开 lxc，不过下面的内容对于理解 docker 还是有很大帮助。

1)pid namespace

不同用户的进程就是通过 pid namespace 隔离开的，且不同 namespace 中可以有相同 pid。所有的 LXC 进程在 docker 中的父进程为 docker 进程，每个 lxc 进程具有不同的 namespace。同时由于允许嵌套，因此可以很方便的实现 Docker in Docker。

2) net namespace

有了 pid namespace, 每个 namespace 中的 pid 能够相互隔离，但是网络端口还是共享 host 的端口。网络隔离是通过 net namespace 实现的， 每个 net namespace 有独立的 network devices, IP addresses, IP routing tables, /proc/net 目录。这样每个 container 的网络就能隔离开来。docker 默认采用 veth 的方式将 container 中的虚拟网卡同 host 上的一个 docker bridge: docker0 连接在一起。

3) ipc namespace

container 中进程交互还是采用 linux 常见的进程间交互方法(interprocess communication - IPC), 包括常见的信号量、消息队列和共享内存。然而同 VM 不同的是，container 的进程间交互实际上还是 host 上具有相同 pid namespace 中的进程间交互，因此需要在 IPC 资源申请时加入 namespace 信息 - 每个 IPC 资源有一个唯一的 32 位 ID。

4) mnt namespace

类似 chroot，将一个进程放到一个特定的目录执行。mnt namespace 允许不同 namespace 的进程看到的文件结构不同，这样每个 namespace 中的进程所看到的文件目录就被隔离开了。同 chroot 不同，每个 namespace 中的 container 在/proc/mounts 的信息只包含所在 namespace 的 mount point。

5) uts namespace

UTS("UNIX Time-sharing System") namespace 允许每个 container 拥有独立的 hostname 和 domain name, 使其在网络上可以被视作一个独立的节点而非 Host 上的一个进程。

6) user namespace

每个 container 可以有不同的 user 和 group id, 也就是说可以在 container 内部用 container 内部的用户执行程序而非 Host 上的用户。

Control groups 主要用来隔离各个容器和宿主主机的资源利用。

三、Docker 安装

官方网站上有各个 linux 发行版的安装指南，这里就写下 centos 和 ubuntu 的安装。

1、ubuntu14.04 安装 docker

```
$ sudo apt-get update

$ sudo apt-get install docker.io

$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker

$ sudo sed -i '$acomplete -F _docker docker' /etc/bash_completion.d/docker.io
```

如果使用操作系统自带包安装 docker ，使用上面的办法，安装的版本是 0.9.1 (不建议，因为 1.0 生产版本已经发布，下面介绍安装方法）

如果要安装最新的 docker 版本，那么需要安装 https 支持

```
$apt-get install apt-transport-https

$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
36A1D7869245C8950F966E92D8576A8BA88D21E9

$ sudo sh -c "echo deb https://get.docker.io/ubuntu docker main\

> /etc/apt/sources.list.d/docker.list"

$ sudo apt-get update

$ sudo apt-get install lxc-docker
```

这样就安装完毕了。

2、ubuntu12.04 安装 docker

如果是更低版本的 ubuntu

```
$ sudo apt-get update

$ sudo apt-get install linux-image-generic-lts-raring linux-headers-generic-lts-raring

# reboot

$ sudo reboot
```

然后重复上面的步骤即可

3、centos6\7 系列安装 docker

使用 EPEL 软件仓库可以安装 docker，版本必须在 centos6 以后

如果是 centos6

```
#wget http://mirrors.hustunique.com/epel/6/i386/epel-release-6-8.noarch.rpm

#rpm -ivhepel-release-6-8.noarch.rpm

#yum install docker-io
```

用上面这个命令安装就可以了

centos7 直接安装就可以了

如果之前的系统中存在 docker 这个软件，最好先删除掉这个包，一个老旧的包

```
$ service docker start

$ chkconfig docker on
```

四、Docker image 详细介绍

在之前的介绍中，我们知道 docker images 是 docker 的三大组件之一。

docker 把下载的 images 存储到 docker 主机上，如果一个 image 不在主机上，docker 会从一个镜像仓库下载，默认的仓库是 DOCKER HUB 公共仓库。

接下来将介绍更多关于 docker images 的内容，包括：

- 使用和管理本地主机上的 images
- 创建一个基础的 images
- 上传 images 到 docker hub（公共 images 仓库）
- 列出本地主机上已经存在的 images

使用 docker images 显示本机上的 images

```
$ sudo docker images
REPOSITORY      TAG     IMAGE ID     CREATED     VIRTUAL SIZE
training/webapp  latest   fc77f57ad303  3 weeks ago  280.5 MB
ubuntu          13.10    5e019ab7bf6d  4 weeks ago  180 MB
ubuntu          saucy    5e019ab7bf6d  4 weeks ago  180 MB
ubuntu          12.04    74fe38d11401  4 weeks ago  209.6 MB
ubuntu          precise  74fe38d11401  4 weeks ago  209.6 MB
ubuntu          12.10    a7cf8ae4e998  4 weeks ago  171.3 MB
ubuntu          quantal  a7cf8ae4e998  4 weeks ago  171.3 MB
ubuntu          14.04    99ec81b80c55  4 weeks ago  266 MB
ubuntu          latest   99ec81b80c55  4 weeks ago  266 MB
ubuntu          trusty   99ec81b80c55  4 weeks ago  266 MB
ubuntu          13.04    316b678ddf48  4 weeks ago  169.4 MB
ubuntu          raring   316b678ddf48  4 weeks ago  169.4 MB
ubuntu          10.04    3db9c44f4520  4 weeks ago  183 MB
ubuntu          lucid    3db9c44f4520  4 weeks ago  183 MB
```

当我们启动一个使用这个 image 的容器时，docker 会从 docker hub 下载它。在列出信息中，我们可以看到 3 个字段信息

- 来自于哪个仓库，比如 ubuntu
- mage 的标记，比如 14.04
- 它的 ID 号

一个仓库可能有一个 images 的都个发行版，比如 ubuntu，他们有 10.04 12.04 12.10 13.04 14.04, 每个发行版的标记都不同，可以使用 tag 命令来指定 images

使用一个 images 的标记来启动容器

```
$ sudo docker run -t -i ubuntu:14.04 /bin/bash
$ sudo docker run -t -i ubuntu:12.04 /bin/bash
```

如果你不指定具体的发行版，比如仅使用 ubuntu，那么 docker 会使用最新的发行版 ubuntu：latest

提示：建议最好指定发行版，只有这样你才可以保证你真正使用的 image 是那个

1、获取 images

我们如何获取新的 images 呢？当我们启动容器使用的 image 不再本地主机上时，docker 会自动下载他们。这很耗时，我们可以使用 docker pull 命令来预先下载我们需要的 image。下面的例子下载一个 centos 镜像。

```
$ sudo docker pull centos
Pulling repository centos
b7de3133ff98: Pulling dependent layers
5cc9e91966f7: Pulling fs layer
511136ea3c5a: Download complete
ef52fb1fe610: Download complete
```

我们可以看到下载的 image 的每一个层次，这样当我们使用这个 image 来启动容器的时候，它就可以马上启动了。

```
$ sudo docker run -t -i centos /bin/bash
bash-4.1#
```

2、查找 images

docker 的一个特点是很多人因为各种不同的用途创建了各种不同的 images。它们都被上传到了 docker hub 共有仓库上，我们可以在 docker hub 的网站上来查找它们。使用 docker search 命令。比如，当我们的团队需要 ruby 和 sinatra 作为 web 应用程序的开发时，我们使用 docker search 来搜索合适的 image，使用关键字 sinatra

```
$ sudo docker search sinatra
NAME                                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
training/sinatra                    Sinatra training image                          0                    [OK]
marceldegraaf/sinatra               Sinatra test app                                0
mattwarren/docker-sinatra-demo                                                      0                    [OK]
luisbebop/docker-sinatra-hello-world                                                0                    [OK]
bmorearty/handson-sinatra           handson-ruby + Sinatra for Hands on with D...   0
subwiz/sinatra                                                                      0
bmorearty/sinatra                                                                   0
```

我们看到返回了很多包含 sinatra 的 images。其中包括 image 名字、描述、星级（表示该 image 的受欢迎程度）、是否官方创建、是否自动创建。官方的 images 是 stackbrew 项目组创建和维护的，autimated 资源允许你验证 image 的来源和内容。

现在我们已经回顾了可用的 images，并决定使用 training/sinatra 镜像。到目前为止，我们看到了 2 种 images 资源。比如 ubuntu，被称为基础或则根镜像。这些基础镜像是 docker 公司创建、验证、支持、提供。他们往往使用一个单词作为他们的名字。

还有一种类型，比如我们选择的 training/sinatra 镜像。它是由 docker 的用户创建并维护的，你可以通过指定 image 名字的前缀来指定他们，比如 training。

3、下载 images

现在我们指定了一个 image，training/sinatra，我们可以使用 docker pull 命令来下载它

```
$ sudo docker pull training/sinatra
```

然后我们就可以使用这个 image 来启动容器了

```
$ sudo docker run -t -i training/sinatra /bin/bash
root@a8cb6ce02d85:/#
```

4、创建我们自己的 images

别人的镜像虽然好，但不一定适合我们。我们可以对他们做一些改变，有 2 个方法：

1)第一个方法：使用 docker commit 来扩展一个 image

先使用 image 启动容器，更新后提交结果到新的 image。

```
$ sudo docker run -t -i training/sinatra /bin/bash
root@0b2616b0e5a8:/#
```

注意：记住容器的 ID，稍后我们还会用到

```
这里我们在容器中添加 json gem
root@0b2616b0e5a8:/# gem install json
```

当结束后，我们使用 exit 来退出，现在我们的容器已经被我们改变了，使用 docker commint 命令来提交相应的副本。

```
$ sudo docker commit -m="Added json gem" -a="Kate Smith" 0b2616b0e5a8 ouruser/sinatra:v2
4f177bd27a9ff0f6dc2a830403925b5360bfe0b93d476f7fc3231110e7f71b1c
```

-m 来指定提交的信息，跟我们使用的版本控制工具一样。

-a 可以指定我们更新的用户信息指定我们要从哪个容器 ID 来创建我们的副本，最后指定目标 image 的名字。

这个例子里面，我们指定了一个新用户，ouruser，使用了 sinatra 的 image，最后指定了 image 的标记 v2。

使用 docker images 来查看我们创建的新 image。

```
$ sudo docker images
REPOSITORY          TAG     IMAGE ID      CREATED       VIRTUAL SIZE
training/sinatra    latest  5bc342fa0b91  10 hours ago  446.7 MB
ouruser/sinatra     v2      3c59e02ddd1a  10 hours ago  446.7 MB
ouruser/sinatra     latest  5db5f8471261  10 hours ago  446.7 MB
使用新的 image 来启动容器
$ sudo docker run -t -i ouruser/sinatra:v2 /bin/bash
root@78e82f680994:/#
```

2)第二个办法：从 dockerfile 来创建 image

使用 docker commit 来扩展一个 image 比较简单，但它不容易在一个团队中分享它。我们使用 docker build 来创建一个新的 image。为此，我们需要创建一个 dockerfile，包含一些如何创建我们的 image 的指令

现在，我们来创建一个目录和一个 dockerfile

```
$ mkdir sinatra
$ cd sinatra
$ touch Dockerfile
```

每一条指令都创建一个 image 的新的一层，下面是一个简单的例子：

```
# This is a comment
FROM ubuntu:14.04
MAINTAINER Kate Smith <ksmith@example.com>
RUN apt-get -qq update
RUN apt-get -qqy install ruby ruby-dev
RUN gem install sinatra
```

- 使用#来注释
- FROM 指令告诉 docker 使用哪个 image 源，
- 接着是维护者的信息
- 最后，我们指定了 3 条 run 指令。每一条 run 指令在 image 执行一条命令，比如安装一个软件包，在这里我们使用 apt 来安装了一些软件

现在，让我们来使用 docker build 来通过 dockerfile 创建 image

```
$ sudo docker build -t="ouruser/sinatra:v2" .
Uploading context  2.56 kB
Uploading context
Step 0 : FROM ubuntu:14.04
 ---> 99ec81b80c55
Step 1 : MAINTAINER Kate Smith <ksmith@example.com>
 ---> Running in 7c5664a8a0c1
 ---> 2fa8ca4e2a13
Removing intermediate container 7c5664a8a0c1
Step 2 : RUN apt-get -qq update
 ---> Running in b07cc3fb4256
 ---> 50d21070ec0c
Removing intermediate container b07cc3fb4256
```

```
Step 3 : RUN apt-get -qqy install ruby ruby-dev
 ---> Running in a5b038dd127e
Selecting previously unselected package libasan0:amd64.
(Reading database ... 11518 files and directories currently installed.)
Preparing to unpack .../libasan0_4.8.2-19ubuntu1_amd64.deb ...
Setting up ruby (1:1.9.3.4) ...
Setting up ruby1.9.1 (1.9.3.484-2ubuntu1) ...
Processing triggers for libc-bin (2.19-0ubuntu6) ...
 ---> 2acb20f17878
Removing intermediate container a5b038dd127e
Step 4 : RUN gem install sinatra
 ---> Running in 5e9d0065c1f7
. . .
Successfully installed rack-protection-1.5.3
Successfully installed sinatra-1.4.5
4 gems installed
 ---> 324104cde6ad
Removing intermediate container 5e9d0065c1f7
Successfully built 324104cde6ad
```

使用-t 标记来指定新的 image 的用户信息和命令

使用了.来指出 dockerfile 的位置在当前目录

注意：你也可以指定一个 dockfile 的路径

我们可以看到 build 进程在执行操作。它要做的第一件事情就是上传这个 dockfile 内容，因为所有的操作都要依据它来进行。

然后，我们看到 dockfile 中的指令被一条一条的执行。每一步都创建了一个新的容器，在容器中执行指令并提交就跟之前介绍过的 docker commit 一样。当所有的指令都执行完毕之后，返回了一个 image id，并且所有的中间步骤所产生的容器都被删除和清理了。

注意：一个 image 不能超过 127 层

从我们新建的 images 开启容器

```
$ sudo docker run -t -i ouruser/sinatra:v2 /bin/bash
```

```
root@8196968dac35:/#
$ sudo docker tag 5db5f8471261 ouruser/sinatra:devel
用 tag 命令标记新的 images
$ sudo docker images ouruser/sinatra
REPOSITORY          TAG     IMAGE ID      CREATED       VIRTUAL SIZE
ouruser/sinatra     latest  5db5f8471261  11 hours ago  446.7 MB
ouruser/sinatra     devel   5db5f8471261  11 hours ago  446.7 MB
ouruser/sinatra     v2      5db5f8471261  11 hours ago  446.7 MB
```

5、使用 docker push 上传 images

```
$ sudo docker push ouruser/sinatra
The push refers to a repository [ouruser/sinatra] (len: 1)
Sending image list
Pushing repository ouruser/sinatra (3 tags)
```

6、用 dcoker rmi 移除本地 images

```
$ sudo docker rmi training/sinatra
Untagged: training/sinatra:latest
Deleted: 5bc342fa0b91cabf65246837015197eecfa24b2213ed6a51a8974ae250fedd8d
Deleted: ed0fffdcdae5eb2c3a55549857a8be7fc8bc4241fb19ad714364cbfd7a56b22f
Deleted: 5c58979d73ae448df5af1d8142436d81116187a7633082650549c52c3a2418f0
```

注意：在删除 images 之前要先用 docker rm 删掉依赖于这个 images 的容器

五、Docker 中的网络介绍

1、端口映射

当我们使用-P 标记时，docker 会随机映射一个 49000 到 49900 的端口到内部容器的端口。

```
使用 docker ps 可以看到  这次是 49155 映射到了 5000

$ sudo docker run -d -P training/webapp python app.py

$ sudo docker ps nostalgic_morse

CONTAINER ID  IMAGE                COMMAND       CREATED       STATUS       PORTS
NAMES

bc533791f3f5  training/webapp:latest  python app.py 5 seconds ago  Up 2 seconds  0.0.0.0:49155-
>5000/tcp  nostalgic_morse

-p（小写的 P）可以指定我们要映射的端口，但是，在一个指定端口上只可以绑定一个容器。

$ sudo docker run -d -p 5000:5000 training/webapp python app.py

-p 默认会绑定本地所有接口地址，所以我们一般指定一个地址，比如 localhost

$ sudo docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py

或者绑定 localhost 的任意端口到容器的 5000 端口

$ sudo docker run -d -p 127.0.0.1::5000 training/webapp python app.py

还可以使用 upd 标记来指定 udp 端口

$ sudo docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py

使用 dicker port 来查看当前绑定的端口配置，也可以查看到绑定的地址

$ docker port nostalgic_morse 5000

127.0.0.1:49155.
```

注意：

- *容器有自己的内部网络和 ip 地址（使用 docker inspect 可以获取所有的变量，docker 还可以有一个可变的网络配置。）*
- *-p 标记可以多次使用来绑定多个端口*

比如：

```
$ sudo docker run -d -p 5000:5000  -p 3000:80 training/webapp python app.py
```

2、docker 中的容器互联-linking 系统

docker 有一个 linking 系统可以连接多个容器。它会创建一对父子关系，父容器可以看到所选择的子容器的信息。

1)容器的命名系统

linking 系统依据容器的名称来执行。当我们创建容器的时候，系统会随机分配一个名字。当然我们也可以自己来命名容器，这样做有 2 个好处：

- 当我们自己指定名称的时候，比较好记，比如一个 web 应用我们可以给它起名叫 web
- 当我们要连接其他容器时候，可以作为一个有用的参考点，比如连接 web 容器到 db 容器

```
使用--name 标记可以为容器命名

$ sudo docker run -d -P --name web training/webapp python app.py

使用 docker -ps 来验证我们设定的命名

$ sudo docker ps -l

CONTAINER ID  IMAGE                COMMAND       CREATED       STATUS       PORTS
NAMES

aed84ee21bde  training/webapp:latest python app.py  12 hours ago  Up 2 seconds 0.0.0.0:49154->5000/tcp  web

也可以使用 docker inspect 来查看容器的名字

$ sudo docker inspect -f "{{ .Name }}" aed84ee21bde

/web
```

注意：容器的名称是唯一的。如果你命名了一个叫 web 的容器，当你要再次使用 web 这个名称的时候，你需要用 docker rm 来删除之前创建的容器，也可以再执行 docker run 的时候 加—rm 标记来停止旧的容器，并删除，rm 和-d 参数是不兼容的。

2)容器互联

```
links 可以让容器之间安全的交互，使用--link 标记。下面先创建一个新的数据库容器，

$ sudo docker run -d --name db training/postgres

删除之前创建的 web 容器

$ docker rm -f web

创建一个新的 web 容器，并将它 link 到 db 容器

$ sudo docker run -d -P --name web --link db:db training/webapp python app.py
```

--link 标记的格式：--link name:alias，name 是我们要链接的容器的名称，alias 是这个链接的别名。

```
使用 docker ps 来查看容器的链接

$ docker ps

CONTAINER ID  IMAGE                    COMMAND           CREATED          STATUS
PORTS                 NAMES

349169744e49  training/postgres:latest  su postgres -c '/usr  About a minute ago  Up About a minute
5432/tcp              db, web/db

aed84ee21bde  training/webapp:latest    python app.py       16 hours ago       Up 2 minutes
0.0.0.0:49154->5000/tcp  web
```

我们可以看到我们命名的容器，db 和 web，db 容器的 names 列有 db 也有 web/db。这表示 web 容器链接到 db 容器，他们是一个父子关系。在这个 link 中，2 个容器中有一对父子关系。docker 在 2 个容器之间创建了一个安全的连接，而且不用映射他们的端口到宿主主机上。在启动 db 容器的时候也不用-p 和-P 标记。**使用 link 之后我们就可以不用暴露数据库端口到网络上。**

docker 通过 2 种方式为父子关系的容器公开连接信息：

- 环境变量
- 更新/etc/hosts 文件

```
使用 env 命令来查看容器的环境变量

$ sudo docker run --rm --name web2 --link db:db training/webapp env

    . . .

    DB_NAME=/web2/db

    DB_PORT=tcp://172.17.0.5:5432

    DB_PORT_5000_TCP=tcp://172.17.0.5:5432

    DB_PORT_5000_TCP_PROTO=tcp

    DB_PORT_5000_TCP_PORT=5432

    DB_PORT_5000_TCP_ADDR=172.17.0.5

    . . .

除了环境变量，docker 还添加 host 信息到父容器的/etc/hosts 的文件。下面是父容器 web 的 hosts 文件

$ sudo docker run -t -i --rm --link db:db training/webapp /bin/bash

root@aed84ee21bde:/opt/webapp# cat /etc/hosts
```

```
172.17.0.7  aed84ee21bde
. . .
172.17.0.5  db
```

这里有 2 个 hosts，第一个是 web 容器，web 容器用 id 作为他的主机名，第二个是 db 容器的 ip 和主机名

```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping db
PING db (172.17.0.5): 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

用 ping 来 ping db 容器，它会解析成 172.17.0.5

注意：官方的 ubuntu 镜像默认没有安装 ping

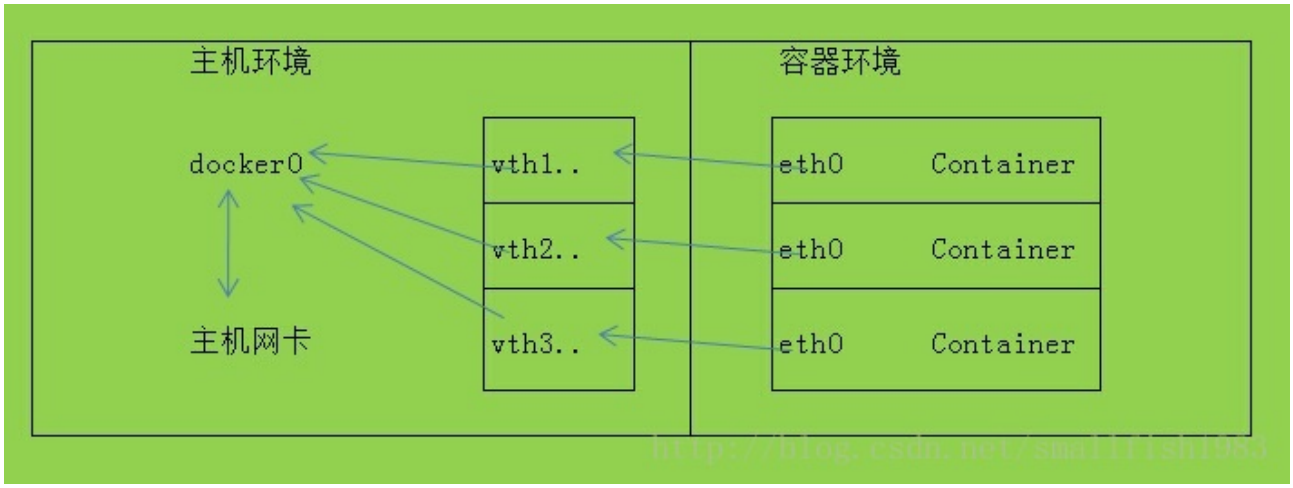
注意：你可以链接多个子容器到父容器，比如我们可以链接多个 web 到 db 容器上。

六、docker 高级网络配置

当 docker 启动时，会在主机上创建一个 docker0 的虚拟网卡。他随机挑选 RFC1918 私有网络中的一段地址给 docker0。比如 172.17.42.1/16,16 位掩码的网段可以拥有 65534 个地址可以使用，这对主机和容器来说应该足够了。

注意：本章介绍 docker 的高级网络配置，一般情况下你不需要知道这些也可以使 docker 正常工作。简单的网络配置和介绍请看第五章内容。

docker0 不是普通的网卡，他是桥接到其他网卡的虚拟网卡，容器使用它来和主机相互通信。当创建一个 docker 容器的时候，它就创建了一个对接口，当数据包发送到一个接口时，另外一个接口也可以收到相同的数据包，它们是绑在一起的一对孪生接口。这对接口在容器中那一端的的名字是 eth0，宿主主机端的会指定一个唯一的名字，比如 vethAQI2QT 这样的名字，这种接口名字不再主机的命名空间中。所有的 veth*的接口都会桥接到 docker0，这样 docker 就创建了在主机和所有容器之间一个虚拟共享网络。



接下来的部分将介绍在一些场景中，docker 所有的网络定制配置。linux 的原生命令将调整、补充、甚至替换 docker 默认的网络配置。

1、快速配置指南

下面是一个跟 docker 网络相关的命令列表，希望可以让你快速找到需要的信息。有些命令选项只有在 docker 服务启动的时候才可以执行，而且不能马上生效。

- -b BRIDGE or --bridge=BRIDGE — 桥接配置
- --bip=CIDR — 定制 docker0 的掩码
- -H SOCKET... or --host=SOCKET... — 它告诉 docker 从哪个通道来接收 run container stop container 这样的命令，也是 docker api 的地址

- --icc=true|false — 请看下文容器之间的通信
- --ip-forward=true|false — 请看下文容器之间的通信
- --iptables=true|false — 请看下文容器之间的通信
- --mtu=BYTES —请看下文定制 docker0

下面 2 个可以在 docker 服务启动和 docker run 执行的时候指定，服务启动的时候指定则会为 docker run 设定默认值，docker run 后面指定可以覆盖默认值。

- --dns=IP_ADDRESS... — 请看下文 dns 配置
- --dns-search=DOMAIN... — 请看下文 dns 配置

最后这些选项只有在 docker run 后执行，因为它是针对容器的特性内容。

- -h HOSTNAME or --hostname=HOSTNAME — 主机名配置
- --link=CONTAINER_NAME:ALIAS — link 系统
- --net=bridge|none|container:NAME_or_ID|host —桥接配置
- -p SPEC or --publish=SPEC — 映射容器端口到宿主主机
- -P or --publish-all=true|false — 映射容器端口到宿主主机

2、配置 DNS

docker 没有定制为每一个容器定制 image，是怎么提供容器的主机名和 dns 配置呢？秘诀就是它用主机上的 3 个配置文件来覆盖容器的这 3 个文件，在容器中使用 mount 命令可以看到：

```
$ mount
...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
tmpfs on /etc/resolv.conf type tmpfs ...
...
```

这种机制可以让宿主主机从 dhcp 更新 dns 信息后，马上更新所有 docker 容器的 dns 配置。如果要保持 docker 中这些文件固定不变，你可以不覆盖容器中的这些配置文件，然后使用下面的选项来配置它们。

配置容器 dns 服务的方法

-h HOSTNAME or --hostname=HOSTNAME

设定容器的主机名，它会被写到/etc/hostname，/etc/hosts 中的 ip 地址自动写成分配的 ip 地址，在/bin/bash 中显示该主机名。但它不会在 docker ps 中显示，也不会在其他的容器的/etc/hosts 中显示。

--link=CONTAINER_NAME:ALIAS

这选项会在创建容器的时候添加一个其他容器 CONTAINE_NAME 的主机名到/etc/hosts 文件中，让新容器的进程可以使用主机名 ALIAS 就可以连接它。--link=会在容器之间的通信中更详细的介绍

--dns=IP_ADDRESS

添加 dns 服务器到容器的/etc/resolv,conf 中，让容器用这 ip 地址来解析所有不在/etc/hosts 中的主机名。

--dns-search=DOMAIN

设定容器的搜索域，当设定搜索域为.example.com 时，会在搜索一个 host 主机名时，dns 不仅搜索 host，还会搜索 host.example.com

注意：如果没有上述最后 2 个选项，docker 会用主机上的/etc/resolv.conf 来配置容器，它是默认配置。

3、容器之间的通信

判断 2 个容器之间是否能够通信，在操作系统层面，取决于 3 个因素：

- 网络拓扑是否连接到容器的网络接口？默认 docker 会将所有的容器连接到 docker0 这网桥来提供数据包通信。其他拓扑结构将在稍后的文档中详细介绍。
- 主机是否开启 ip 转发，ip_forward 参数为 1 的时候可以提供数据包转发。通常你只需要为 docker 设定 --ip-forward=true,docker 就会在服务启动的时候设定 ip_forward 参数为 1。下面是手工检查并手工设定该参数的方法。

```
# Usually not necessary: turning on forwarding,
# on the host where your Docker server is running
$ cat /proc/sys/net/ipv4/ip_forward
0
$ sudo echo 1 > /proc/sys/net/ipv4/ip_forward
$ cat /proc/sys/net/ipv4/ip_forward
1
```

- 你的 iptables 是否允许这条特殊的连接被建立？当 docker 的设定--iptables=false 时，docker 不会改变系统的 iptables 设定，否则它会在--icc=true 的时候添加一条默认的 ACCEPT 策略到

FORWARD 链，当—icc=false 时，策略为 DROP。几乎所有的人都会开启 ip_forward 来启用容器间的通信。但是否要改变 icc-true 配置是一个战略问题。这样 iptable 就可以防止其他被感染容器对宿主主机的恶意端口扫描和访问。

当你选择更安全的设定--icc=false 后，如何保持你希望的容器之间通信呢?

答案就是--link=CONTAINER_NAME:ALIAS 选项，在之前的 dns 服务设定中提及过。如果 docker 使用 icc=false and --iptables=true 2 个参数，当 docker run 使用--link=选型时，docker 会为 2 个容器在 iptable 中参数一对 ACCEPT 规则，开放的端口取决与 dockerfile 中的 EXPOSE 行，详见第五章。

注意：--link= 中的 CONTAINER_NAME 必须是自动生成的 docker 名字比如 stupefied_pare，或者你用--name 参数指定的名字，主机名在--link 中不会被识别。

你可以使用 iptables 命令来检查 FORWARD 链是 ACCEPT 还是 DROP

```
当--icc=false 时，默认规则应该是这样
$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  0.0.0.0/0            0.0.0.0/0
...
当添加了--link 后，ACCEPT 规则被改写了，添加了新的端口和 IP 规则
$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  172.17.0.2           172.17.0.3           tcp spt:80
ACCEPT     tcp  --  172.17.0.3           172.17.0.2           tcp dpt:80
DROP       all  --  0.0.0.0/0            0.0.0.0/0
```

4、映射一个容器端口到宿主主机

默认情况下，容器可以建立到外部网络的连接，但是外部网络无法连接到容器。所有到外部的连接，源地址都会被伪装成宿主主机的 ip 地址，iptables 的 masquerading 来做到这一点。

```
# 查看主机的 masquerading 规则
$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16      !172.17.0.0/16
...
```

当你希望容器接收外部连接时，你需要在 docker run 执行的时候就指定对应选项，第五章详细介绍了 2 种方法：

- 指定-P --publish-all=true|false 选项会映射 dockerfile 中 expose 的所有端口，主机端口在 49000-49900 中随机挑选。当你的另外一个容器需要学习这个端口时候，很不方便。
- 指定-p SPEC 或则 --publish=SPEC,可以指定任意端口从主机映射容器内部

不管用那种办法，你可以通过查看 iptable 的 nat 表来观察 docker 在网络层做了什么操作。

```
#使用-P 时：
$ iptables -t nat -L -n
...
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:49153 to:172.17.0.2:80
#使用-p 80:80 时：
$ iptables -t nat -L -n
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:80 to:172.17.0.2:80
```

注意：

- *这里看到 docker 映射了 0.0.0.0.它接受主机上的所有接口地址。可以通过-p IP:host_port:container_port 或则 -p IP::port 来指定主机上的 ip、接口，制定更严格的规则。*
- 如果你希望永久改变绑定的主机 ip 地址，可以 在 dcoker 配置中指定--ip=IP_ADDRESS. 记得重启服务。

5、定制 docker0

docker 服务默认会创建一个 docker0 接口，它在 linux 内核层桥接所有物理或虚拟网卡，这就将所有容器和主机接口都放到同一个物理网络。

Docker 指定了 docker0 的 ip 地址和子网掩码，让主机和容器之间可以通过网桥相互通信，它还给出了 MTU-接口允许接收的最大传输单元，通常是 1500bytes 或宿主主机网络路由上支持的默认值，这 2 个都需要在服务启动的时候配置。

- --bip=CIDR — 192.168.1.5/24.ip 地址加掩码 使用这种格式
- --mtu=BYTES — 覆盖默认的 docker mtu 配置

你可以在配置文件中配置 DOCKER_OPTS，然后重启来改变这些参数。

```
# 当容器启动后，你可以使用 brctl 来确认他们是否已经连接到 docker0 网桥
$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.3a1d7362b4ee       no              veth65f9
                                                        vethdda6
#如果 brctl 命令没安装的话，在 ubuntu 中你可以使用 apt-get install bridge-utils 这个命令来安装
```

docker0 网桥设置会在每次创建新容器的时候被使用。docker 从可用的地址段中选择一个空闲的 ip 地址给容器的 eth0 端口，子网掩码使用网桥 docker0 的配置，docker 主机本身的 ip 作为容器的网关使用。

```
$ sudo docker run -i -t --rm base /bin/bash
$ ip addr show eth0
24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
```

```
        valid_lft forever preferred_lft forever
    inet6 fe80::306f:e0ff:fe35:5791/64 scope link
        valid_lft forever preferred_lft forever
$ ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.3
$ exit
```

转发数据包需要在主机上设定 ip_forward 参数为 1,上文介绍过。

6、创建自己的桥接

如果希望完全使用自己的桥接设置，可以在启动 docker 服务的时候，使用 -b BRIDGE or --bridge=BRIDGE 来告诉 docker 使用你的网桥。如果服务已经启动，旧的网桥还在使用中，那需要先停止服务，再删除旧的网桥

```
#停止旧网桥并删除
$ sudo service docker stop
$ sudo ip link set dev docker0 down
$ sudo brctl delbr docker0
```

然后在开启服务前，创建你自己希望的网桥接口，这里建立一个网桥的配置：

```
# 创建自己的网桥
$ sudo brctl addbr bridge0
$ sudo ip addr add 192.168.5.1/24 dev bridge0
$ sudo ip link set dev bridge0 up
# 确认网桥启动
$ ip addr show bridge0
4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default
    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.1/24 scope global bridge0
        valid_lft forever preferred_lft forever
```

```
# 告诉 docker 桥接设置，并启动 docker 服务（在 ubuntu 上）
$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
$ sudo service docker start
```

docker 服务启动成功并绑定容器到新的网桥，新建一个容器，你会看到它的 ip 是我们的设置的新 ip 段，docker 会自动检测到它。用 brctl show 可以看到容器启动或则停止后网桥的配置变化，在容器中使用 ip a 和 ip r 来查看 ip 地址配置和路由信息。

7、Docker 如何连接到容器?

让我们回顾一些基础知识：

机器需要一个网络接口来发送和接受数据包，路由表来定义如何到达哪些地址段。这里的网络接口可以不是物理接口。事实上，每个 linxu 机器上的 lo 环回接口（docker 容器中也有）就是一个完全的 linux 内核虚拟接口，它直接复制发送缓存中的数据包到接收缓存中。docker 让宿主主机和容器使用特殊的虚拟接口来通信--通信的 2 端叫“peers“，他们在主机内核中连接在一起，所以能够相互通信。创建他们很简单，前面介绍过了。

docker 创建容器的步骤如下：

- 创建一对虚拟接口
- 其中宿主主机一端使用一个名字比如 veth65f9，他是唯一的,另外一端桥接到默认的 docker0,或其它你指定的桥接网卡。
- 主机上的 veth65f9 这种接口映射到新的新容器中的名称通常是 eth0,在容器这个隔离的 network namespace 中，它是唯一的，不会有其他接口名字和它冲突。
- 从主机桥接网卡的地址段中获取一个空闲地址给 eth0 使用，并设定默认路由到桥接网卡。
- 完成这些之后，容器就可以使用这 eth0 虚拟网卡来连接其他容器和其他网络。

你也可以为特殊的容器设定特定的参数，在 docker run 的时候使用--net，它有 4 个可选参数：

--net=bridge — .默认连接到 docker0 网桥。

--net=host — 告诉 docker 不要将容器放到隔离的网络堆栈中。从本质上讲，这个选项告诉 docker 不要容器化容器的网络！尽管容器还是有自己的文件系统、进程列表和资源限制。但使用 ip addr 命令这样命令就可以知道实际上此时的的容器处于和 docker 宿主主机的一样的网络级别，它拥有完全的宿主主机接口访问权限。虽然它不允许容器重新配置主机的网络堆栈，除非--privileged=true — 但是容器进程可以跟其他 root 进程一样可以打开低数字的端口，可以访问本地网络服务比如 D-bus，还可以让容器做一些意想不到的事情，比如重启主机，使用这个选项的时候要非常小心！

--net=container:NAME_or_ID — 告诉 docker 将新容器的进程放到一个已经存在的容器的网络堆栈中，新容器进程有它自己的文件系统、进程列表和资源限制，但它会和那个已经存在的容器共享 ip 地址和端口，他们之间来可以通过环回接口通信。

--net=none — 告诉 docker 将新容器放到自己的网络堆栈中，但是不要配置它的网络,类似于 vmware 的 host-only。这可以让你创建任何自定义的配置，本文最后一段将介绍 他们。

下面通过配置一个以--net=none 启动的容器，使他达到跟平常一样具有访问网络的权限。来介绍 docker 是如何连接到容器中的。

```
# 启动一个/bin/bash 指定--net=none
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@63f36fc01b5f:/#
#  再开启一个新的终端，查找这个容器的进程 id，然后创建它的命名空间，后面的 ip netns 会用到
$ sudo docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
$ pid=2778
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid
#检查桥接网卡的 ip 和子网掩码
$ ip addr show docker0
21: docker0: ...
inet 172.17.42.1/16 scope global docker0
...
# 创建一对”peer“接口 A 和 B，绑定 A 到网桥，并启用它
$ sudo ip link add A type veth peer name B
$ sudo brctl addif docker0 A
$ sudo ip link set A up
# 将 B 放到容器的网络命名空间，命名为 eth0,配置一个空闲的 ip
$ sudo ip link set B netns $pid
$ sudo ip netns exec $pid ip link set dev B name eth0
```

```
$ sudo ip netns exec $pid ip link set eth0 up
$ sudo ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0
$ sudo ip netns exec $pid ip route add default via 172.17.42.1
#自此，你又可以像平常一样使用网络了
```

当你退出 shell 后，docker 清空容器，容器的 eth0 随网络命名空间一起被摧毁，A 接口也被自动从 docker0 取消注册。不用其他命令，所有东西都被清理掉了！

注意 ip netns exec 命令，它可以让我们像 root 一样配置网络命名空间。但在容器内部无法使用，因为统一的安全策略，docker 限制容器进程配置自己的网络。使用 ip netns exec 可以让我们不用设置--privileged=true 就可以完成一些可能带来危险的操作。

8、工具和示例

在介绍自定义网络拓扑之前，你可能会对一些外部工具和例子感兴趣：

https://github.com/jpetazzo/pipework

Jérôme Petazzoni 创建了一个叫 pipework 的 shell 脚本来帮助我们在复杂的场景中完成网络连接

https://github.com/brandon-rhodes/fopnp/tree/m/playground

Brandon Rhodes 创建了一个完整的 docker 容器网络拓扑，包含 nat 防火墙，服务包括 HTTP, SMTP, POP, IMAP, Telnet, SSH, and FTP:

工具使用的网络命令跟我们之前看到非常相似。

9、创建一个点到点连接

默认 docker 会将所有容器连接到由 docker0 提供的虚拟子网，你也可以使用自己创建的网桥。但如果你想要 2 个特殊的容器之间可以直连通信，而不用去配置复杂的主机网卡桥接。

解决办法很简单：创建一对接口，把 2 个容器放到这对接口中，配置成点到点链路类型。这 2 个容器就可以直接通信了。配置如下：

```
# 在 2 个终端中启动 2 个容器
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@1f1f4c1f931a:/#
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@12e343489d2f:/#
```

```
#找到他们的 process IDs，然后创建他们的 namespace entries

$ sudo docker inspect -f '{{.State.Pid}}' 1f1f4c1f931a
2989
$ sudo docker inspect -f '{{.State.Pid}}' 12e343489d2f
3004
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/2989/ns/net /var/run/netns/2989
$ sudo ln -s /proc/3004/ns/net /var/run/netns/3004

# 创建”peer“接口，然后配置路由

$ sudo ip link add A type veth peer name B

$ sudo ip link set A netns 2989
$ sudo ip netns exec 2989 ip addr add 10.1.1.1/32 dev A
$ sudo ip netns exec 2989 ip link set A up
$ sudo ip netns exec 2989 ip route add 10.1.1.2/32 dev A

$ sudo ip link set B netns 3004
$ sudo ip netns exec 3004 ip addr add 10.1.1.2/32 dev B
$ sudo ip netns exec 3004 ip link set B up
$ sudo ip netns exec 3004 ip route add 10.1.1.1/32 dev B
```

现在这 2 个容器就可以相互 ping 通，并成功建立连接。点到点链路不需要子网和子网掩码，使用 ip route 来连接单个 ip 地址到指定的网络接口。

如果没有特殊需要你不需要指定--net=none 来创建点到点链路。

还有一个办法就是创建一个只跟主机通信的容器，除非有特殊需求，你可以仅用--icc=false 来限制主机间的通信。

七、Docker 数据管理

这一章介绍如何在 docker 内部以及容器之间管理数据

在容器中管理数据的 2 个主要方式：

- Data volumes
- Data volume containers.

1、Data volumes 数据卷

数据卷是一个由 UFS 文件系统专门设计的的特殊目录，它可以提供很多有用的特性：

- 数据卷可以在容器之间共享和重用
- 对数据卷的改变是立马生效
- 当你更新数据卷中的数据的时候，不会被包含到 image 中
- 卷会一直存在直到没有容器使用他们

1)添加一个数据卷

在用 docker run 命令的时候，使用-v 标记来添加一个数据卷。在一次 run 中多次使用可以挂载多个数据卷，下面加载一个卷到 web 容器上。

```
$ sudo docker run -d -P --name web -v /webapp training/webapp python app.py

#创建一个新的卷到容器的/webapp
```

注意：也可以在 dockerfile 中使用 volume 来添加一个或者多个新的卷到由该 image 创建的任意容器

2)挂载一个主机目录作为数据卷

使用-v 标记也可以挂载一个主机的目录到容器中去

```
$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp training/webapp python app.py

#上面的命令加载主机的/src/webapp 到容器的/opt/webapp 目录。这个在测试的时候特别好用，比如我们可以加载我们的源码到容器中，来查看他们是否正常工作。目录的路径必须是主机上的绝对路径，如果目录不存在 docker 会自动为你创建它。
```

注意:dockerfile 中不能用，各种操作系统的文件路径格式不一样，所以不一定适合所有的主机。

```
#docker 加载的数据卷默认是读写权限，但我们可以把它加载为只读。

$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp:ro training/webapp python app.py

#加了 ro 之后，就挂载为只读了。
```

3)挂载一个宿主主机文件作为数据卷

```
#-v 标记也可以从主机挂载单个文件到容器中

$ sudo docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu /bin/bash

#这样就可以记录在容器输入过的命令了。
```

注意：很多工具子在使用 vi 或者 sed --in-place 的时候会导致 inode 的改变，从 docker 1.1.0 起，它会报错，所以最简单的办法就直接 mount 父目录。

2、Data Volume Container 数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建 Data Volume Container，然后加载它。现在就来创建一个命名的数据卷容器：

```
$ sudo docker run -d -v /dbdata --name dbdata training/postgres echo Data-only container for postgres

#然后，你可以在其他容器中使用--volumes-from 来挂载/dbdata 卷

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

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

#还可以使用多个--volumes-from 参数来从多个容器挂载多个数据卷

#也可以从其他已经挂载了容器卷的容器来挂载数据卷

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

如果你移除了挂载的容器，包括初始容器，或者后来的 db1 db2，这些卷在有容器使用它的时候不会被删除。这可以让我们在容器之间升级和移动数据。

3、利用 Data Volume Container 来备份、恢复、移动数据卷

数据卷另外一个功能是使用他们来备份、恢复、移动数据。使用--volume 标记来创建一个加载了卷的新的容器，命令如下：

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

这里我们创建了一个容器，先从 dbdata 容器来挂载数据卷。然后从本地主机挂载当前到容器的/backup 目录。最后，使用 tar 命令来将 dbdata 卷备份为 back.tar。当命令执行完、容器停止之后，我们就备份了 dbdata 数据卷。

```
你可以使用这个备份来恢复这个容器。
$ sudo docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
#然后使用 untar 解压这个备份文件到新容器卷中。
$ sudo docker run --volumes-from dbdata2 -v $(pwd):/backup busybox tar xvf /backup/backup.tar
你可以用上述技术实现数据卷的备份、移动、恢复。
```

八、容器安全

评估 docker 的安全性时，主要考虑 3 个方面:

- 由内核中 namespace 和 cgruoups 提供的容器的内在安全
- docker 程序本身的抗攻击性
- 加固内核安全性来影响容器的安全性

1、Kernel Namespaces

docker 容器和 lxc 容器很相似，他们提供的安全特性也差不多。当你用 docker run 启动一个容器时，在后台 docker 为容器创建了一个 namespace 和 contril groups 的集合。

Namespaces 提供了最初也是最直接的隔离，在容器中运行的进程不会被运行在主机上的进程和容器发现，他们之间相互影响也就小了。

每个容器都有自己的网络堆栈，他们不能访问其他容器的 sockets 接口。不过，如果在主机系统上做了相应的设置，他们还是可以像跟主机交互一样的和其他容器交互通信。当你指定公共端口或则使用 links 来连接 2 个容器时，他们就可以相互通信了。（相互 ping、udp、tcp 都没问题，也可以根据需要设定更严格的策略）从网络架构上来看，所有的容器通过主机的网桥接口相互通信，就像物理机器通过物理交换机通信一样。

内核提供的 namesapce 和私有网络的代码有多成熟?

内核 namesapce 从内核 2.6.15 之后被引入，距今已经 5 年了，在很多大型生产系统中被验证。他们的设计和灵感提出的时间更早，openvz 项目利用 namespace 重新封装他们的内核，并合并到主流内核中。openvz 最早的版本在 2005，所以他们的设计和实现都很成熟。

2、Control Groups

Control Groups 是 LXC 容器的另外一个关键组件，由它来实现资源的审计和限制。他们提供了很多有用的特性，还可以用来确保每个容器可以公平分享主机的内存、cpu、磁盘 IO 等资源，更重要的是，它可以保证当一个容器耗尽其中一个资源的时候不会连累主机宕机。

尽管他们不阻止容器之间相互访问、处理数据和进程，但他们在防止拒绝服务攻击方面是必不可少的。在多用户的平台比如共有或则私有的 paas 上更加重要，当某些应用程序表现不好的时候，可以保证一直的 uptime 和性能。Control Groups 始于 2006 年，从 2.6.24 之后被引入。

3、Docker Daemon Attack Surface

运行一个容器或则应用程序意味着运行一个 docker 服务。docker 服务要求 root 权限，所以你需要了解一些重要的细节。

首先，确保只有可信的用户可以访问 docker 服务，因为这会直接导致很严重的后果。因为，docker 允许你在主机和容器之间共享文件夹，这就容易让容器突破资源限制。比如当你在启动容器的时候将主机的/映射到容器的/host 目录中，那么容器就可以对主机做任何更改了。这听起来很疯狂？不过，你要知道几乎所有虚拟机系统都有在物理主机和虚拟机之间共享资源的限制，所以需要你自己来考虑这一层的安全性。

比如，当你使用一个 web api 来提供容器创建服务时，要比平常更加注意参数的检查，防止恶意的用户用精心准备的参数来创建带有任意参数的容器

因此，REST API 在 docker0.5.2 之后使用 unix socket 替代了绑定在 127.0.0.1 上的 tcp socket（后者容易遭受跨站脚本攻击）。现在你可以使用增强的 unix sockt 权限来限制对控制 socket 的访问。

你依然可以将 REST API 发布到 http 服务上。不过一定要小心确认这里的安全机制，确保只有可信的网络或则 vpn 或则受保护的 stunnel 和 ssl 认证可以对 REST API 进行访问。还可以使用 https 和认证 HTTPS and certificates.

最近改进的 linux namespace 将很快可以实现使用非 root 用户来运行全功能的容器。这解决了因在容器和主机共享文件系统而引起的安全问题。

docker 的终极目标是改进 2 个安全特性：

- 将 root 用户的容器映射到主机上的非 root 用户，减轻容器和主机之间因权限提升而引起的安全问题
- 允许 docker 服务在非 root 权限下运行，委派操作请求到那些经过良好审计的子进程，每个子进程拥有非常有限的权限：虚拟网络设定，文件系统管理、配置等等。

最后，如果你在一个服务器上运行 docker，建议去掉 docker 之外的其他服务，除了一些管理服务比如 ssh 监控和进程管理工具 nrpe clllectd 等等。

4、Linux Kernel Capabilities

默认情况下，docker 启动的容器只严格使用一部分内核 capabilities。这代表什么呢?

这是一个 root 或非 root 二分法粒度管理的访问控制系统。比如 web 服务进程只需要绑定一个低于 1024 的端口，不需要用 root 来允许：那么它只需要给它授权 net_bind_service 功能就可以了。还有很多其他的 capabilities，几乎所有需要 root 权限的仅需要指定一个部分 capabilities 就可以了。

这对容器的安全有很多好处，通常的服务器需要允许一大堆 root 进程，通常有 ssh cron syslogd; 模块和网络配置工具等等。容器则不同，因为大部分这种人物都被容器外面的基础设施处理了：

- ssh 可以被主机上 ssh 服务替代
- 硬件管理也无关紧要，容器中也就无需执行 udevd 或则其他类似的服务
- 网络管理也都在主机上设置，除非特殊需求，ifconfig、route、ip 也不需要了。

这意味这大部分情况下，容器完全不需要“真正的”root 权限。因此，容器可以运行一个减少的 capabilities 集，容器中的 root 也比“真正的 root"拥有更少的 capabilities,比如：

- 完全禁止任何 mount 操作
- 禁止直接访问宿主主机的 socket
- 禁止访问一些文件系统的操作，比如创建新的设备 node 等等
- 禁止模块加载
- 还有一些其他的

就算攻击者在容器中取得了 root 权限，他能做的破坏也少了，也不能获得主机的更高权限。

然而这不会影响普通的 web apps，恶意的用户会想各种办法来对你！默认情况下，docker 丢弃了它需要的功能之外的其余部分。这里有一个白名单和黑名单，在 Linux manpages 可以看到完整的清单列表。当然，你还可以启用你需要的额外 capabilities。默认 docker 容器仅使用白名单的内 capabilities。

5、Other Kernel Security Features

Capabilities 是现代 linux 内核提供的诸多安全特性中的一个，docker 可以利用现有的如 TOMOYO, AppArmor, SELinux, GRSEC 来增强安全性。为什么 docker 当前只启用 capabilities,而不介入其他系统。

因为这样他就还可以有很多方法来加固 docker 主机，下面是一些例子。

- 你可以在内核中加载 GRSEC 和 PAX，这会增加很多安全检查。
- 你可以使用一些有增强安全特性的发行版的模板，比如带 apparmor 的模板和 redhat 系列带 selinux dcoker 策略，这些模板提供了额外的安全特性。
- 使用你自己喜欢的访问控制机制来定义你自己的安全策略。
- 像其他添加到 docker 容器的第三方工具一样（比如网络拓扑和文件系统共享），有很多这样的工具，利用他们可以不用改变 docker 内核就可以加固现有的 docker 容器

6、结论

docker 容器默认还是比较安全的，特别是你如果注意在容器中使用非 root 权限来允许进程的话。你还可以添加额外的比如 Apparmor, SELinux, GRSEC 等你熟悉的加固方法。最后，如果你对其他容器系统中的安全特性感兴趣，你也可以在 docker 中实现它，毕竟，所有的东西都已经在内核中了。

九、Docker 实战—从无到有部署局域网 docker（解决墙的问题）

由于 GFW 的关系，国内用户在使用 docker 的时候，pull 一个基本的镜像都拉下来，更不用说使用官方的 index 镜像了。

本节介绍如何在被墙的情况下创建属于自己特色的 docker

1、安装 docker

参见本文第三小节

2、从文件系统创建一个 image 镜像

创建镜像有很多方法，官方的推荐是 pull 一个，不过在墙内，想 pull 一个基本的 ubuntu 都没办法完成。

这里推荐一个办法就是从一个文件系统 import 一个镜像，个人推荐可以使用 opvz 的模板来创建：（openvz 可以说是容器虚拟化的先锋吧）

openvz 的模板下载地址如下：

http://openvz.org/Download/templates/precreated

下载完之后，比如：下载了一个 ubuntu14.04 的镜像

使用以下命令：

```
sudo cat ubuntu-14.04-x86_64-minimal.tar.gz  |docker import - ubuntu:14.04
```

然后用 docker images 看下：

```
docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              14.04               05ac7c0b9383        17 seconds ago      215.5 MB
```

就多了一个我们的 ubuntu 镜像

3、创建私有仓库

官方指南称最简单的办法是 docker run -p 5000:5000 registry，如果被墙了，也无法下载该 images。感谢 CSDN，我有一个 1M 的腾讯云服务器，上面搭建了一个私有仓库大家可以使用

```
docker pull 203.195.193.251:5000/registry
```

到我的服务器下载 速度虽然慢点，但有保证！

另外的方法是使用刚才的创建的 ubuntu 来创建，官方有个 docker 仓库的源码地址

https://github.com/dotcloud/docker-registry 下载私有仓库的源码，可以根据上面的 docker file 来创建。

也可以参考：

http://www.vpsee.com/2013/11/build-your-own-docker-private-regsitry-service/

4、在私有仓库上传、下载、搜索 images

创建好自己的私有仓库之后，可以使用 docker tag 一个镜像，然后 push，然后在别的机器上 pull 下来就好了。这样我们的局域网私有 docker 仓库就搭建好了。

步骤如下：

使用 docker run -p 5000:5000 registry 在局域网的一台机器上开启一个容器之后，我的局域网私有仓库地址为 192.168.7.26:5000

先在本机看下现有的 images

```
WaitFish ~ # docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              latest              ba5877dc9bec        6 weeks ago         192.7 MB
ubuntu              14.04               ba5877dc9bec        6 weeks ago         192.7 MB
```

使用 docker tag 将 ba58 这个 image 标记为 192.168.7.26:5000/test

```
WaitFish ~ # docker tag ba58  -t 192.168.7.26:5000/test
WaitFish ~ # docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              14.04               ba5877dc9bec        6 weeks ago         192.7 MB
ubuntu              latest              ba5877dc9bec        6 weeks ago         192.7 MB
192.168.7.26:5000/test   latest              ba5877dc9bec        6 weeks ago         192.7 MB
```

使用 docker push 上传我们标记的新 image，这里因为我的服务器上已经有这个 images，所有在上传文件层的时候，都跳过了，但是标记还是不一样的。

```
WaitFish ~ # docker push 192.168.7.26:5000/test
The push refers to a repository [192.168.7.26:5000/test] (len: 1)
Sending image list
```

```
Pushing repository 192.168.7.26:5000/test (1 tags)
Image 511136ea3c5a already pushed, skipping
Image 9bad880da3d2 already pushed, skipping
Image 25f11f5fb0cb already pushed, skipping
Image ebc34468f71d already pushed, skipping
Image 2318d26665ef already pushed, skipping
Image ba5877dc9bec already pushed, skipping
Pushing tag for rev [ba5877dc9bec] on {http://192.168.7.26:5000/v1/repositories/test/tags/latest}
WaitFish ~ # curl http://192.168.7.26:5000/v1/search
The program 'curl' is currently not installed. You can install it by typing:
apt-get install curl
现在的私有仓库只支持这样简陋的搜索方式，如果没有安装 curl，可以先安装后再使用
WaitFish ~ # apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  curl
0 upgraded, 1 newly installed, 0 to remove and 570 not upgraded.
Need to get 123 kB of archives.
After this operation, 313 kB of additional disk space will be used.
Get:1 http://192.168.7.26/ubuntu/ trusty/main curl amd64 7.35.0-1ubuntu2 [123 kB]
Fetched 123 kB in 0s (7457 kB/s)
Selecting previously unselected package curl.
(Reading database ... 184912 files and directories currently installed.)
Preparing to unpack .../curl_7.35.0-1ubuntu2_amd64.deb ...
Unpacking curl (7.35.0-1ubuntu2) ...
Processing triggers for man-db (2.6.7.1-1) ...
```

```
Setting up curl (7.35.0-1ubuntu2) ...
WaitFish ~ # curl http://192.168.7.26:5000/v1/search
{"num_results": 7, "query": "", "results": [{"description": "", "name": "library/miaxis_j2ee"}, {"description": "", "name": "library/tomcat"}, {"description": "", "name": "library/ubuntu"}, {"description": "", "name": "library/ubuntu_office"}, {"description": "", "name": "library/desktop_ubu"}, {"description": "", "name": "dockerfile/ubuntu"}, {"description": "", "name": "library/test"}]}
这里我们可以看到 {"description": "", "name": "library/test"} 表示我们的 image 已经被成功上传了。
现在我们到另外一台机器上下载这个 images
[root@opnvz ~]# docker pull 192.168.7.26:5000/test
Pulling repository 192.168.7.26:5000/test
ba5877dc9bec: Download complete
511136ea3c5a: Download complete
9bad880da3d2: Download complete
25f11f5fb0cb: Download complete
ebc34468f71d: Download complete
2318d26665ef: Download complete
[root@opnvz ~]# docker images
REPOSITORY                    TAG         IMAGE ID        CREATED         VIRTUAL SIZE
192.168.7.26:5000/test        latest      ba5877dc9bec    6 weeks ago     192.7 MB
```

这样我们就可以在新的机器上使用这个 images 了！

十、Docker 实战--在 Docker 中使用 Supervisor 来管理进程

docker 容器在启动的时候开启单个进程，比如，一个 ssh 或则 apache 的 daemon 服务。但我们经常需要在一个机器上开启多个服务，这可以有很多方法，最简单的就是把多个启动命令方到一个启动脚本里面，启动的时候直接启动这个脚本，另外就是安装进程管理工具。.

本小节将使用进程管理工具 supervisor 来管理容器中的多个进程。使用 Supervisor 可以更好的控制、管理、重启我们希望运行的进程。在这里我们演示一下如何同时使用 ssh 和 apache 服务。

1、dockerfile

```
FROM ubuntu:13.04

MAINTAINER examples@docker.com

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

RUN apt-get update

RUN apt-get upgrade -y


安装 supervisor

安装    ssh apache supervisor


RUN apt-get install -y openssh-server apache2 supervisor

RUN mkdir -p /var/run/sshd

RUN mkdir -p /var/log/supervisor

这里安装 3 个软件，还创建了 2 个用来允许 ssh 和 supervisor 的目录

COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf

添加 supervisor‘s 的配置文件

添加配置文件到对应目录下面

映射端口，开启 supervisor

使用 dockerfile 来映射指定的端口，使用 cmd 来启动 supervisord

EXPOSE 22 80

CMD ["/usr/bin/supervisord"]

这里我们映射了 22 和 80 端口，使用 supervisord 的可执行路径启动服务。
```

2、supervisor 配置文件内容

```
[supervisord]

nodaemon=true

[program:sshd]

command=/usr/sbin/sshd -D


[program:apache2]

command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

配置文件包含目录和进程，第一段 supervsord 配置软件本身，使用 nodaemon 参数来运行。下面 2 段包含我们要控制的 2 个服务。每一段包含一个服务的目录和启动这个服务的命令

3、使用方法

```
创建 image

$ sudo docker build -t test/supervisord .

启动我们的 supervisor 容器

$ sudo docker run -p 22 -p 80 -t -i test/supervisords

2013-11-25 18:53:22,312 CRIT Supervisor running as root (no user in config file)

2013-11-25 18:53:22,312 WARN Included extra file "/etc/supervisor/conf.d/supervisord.conf" during parsing

2013-11-25 18:53:22,342 INFO supervisord started with pid 1

2013-11-25 18:53:23,346 INFO spawned: 'sshd' with pid 6

2013-11-25 18:53:23,349 INFO spawned: 'apache2' with pid 7

使用 docker run 来启动我们创建的容器。使用多个-p 来映射多个端口，这样我们就能同时访问 ssh 和 apache 服务了。
```

4、可以使用这个方法创建一个只有 ssh 服务基础 image

之后创建 image 可以以这个 image 为基础来创建

十一、Docker 实战—创建 tomcat/weblogic 集群

1、安装 tomcat 镜像

准备好需要的 jdk tomcat 等软件放到 home 目录下面，启动一个虚拟机

```
docker run -t -i -v /home:/opt/data  --name mk_tomcat ubuntu /bin/bash
```

这条命令挂载本地 home 目录到虚拟机的/opt/data 目录，虚拟机内目录若不存在，则会自动创建。接下来就是 tomcat 的基本配置，jdk 环境变量设置好之后，将 tomcat 程序放到/opt/apache-tomcat 下面

编辑/etc/supervisor/conf.d/supervisor.conf 文件，添加 tomcat 项

```
[supervisord]
nodaemon=true

[program:tomcat]
command=/opt/apache-tomcat/bin/startup.sh

[program:sshd]
command=/usr/sbin/sshd -D
docker commit  ac6474aeb31d  tomcat
```

新建 tomcat 文件夹，新建 Dockerfile

```
FROM tomcat
EXPOSE  22 8080
CMD ["/usr/bin/supervisord"]
```

根据 dockerfile 创建 image

```
docker build tomcat tomcat
```

2、安装 weblogic 镜像

步骤和 tomcat 基本一致，这里贴一下配置文件

supervisor.conf

```
[supervisord]
nodaemon=true



[program:weblogic]
command=/opt/Middleware/user_projects/domains/base_domain/bin/startWebLogic.sh


[program:sshd]
command=/usr/sbin/sshd -D
```

dockerfile

```
FROM weblogic
EXPOSE  22 7001
CMD ["/usr/bin/supervisord"]
```

3、tomcat/weblogic 镜像的使用

1)存储的使用

在启用 docker run 的时候，使用 -v 参数

-v, --volume=[] Bind mount a volume (e.g. from the host: -v /host:/container, from docker: -v /container)

将本地磁盘映射到虚拟机内部，它在主机和虚拟机容器之间是实时变化的，所以我们更新程序、上传代码只需要更新物理主机的目录就可以了，数据存储的详细介绍请参见本文第七小节

2)tomcat 和 weblogic 集群的实现

tomcat 只要开启多个容器即可

```
docker run -d -v -p 204:22 -p 7003:8080 -v /home/data:/opt/data --name tm1 tomcat
```

```
/usr/bin/supervisord

docker run -d -v -p 205:22 -p 7004:8080 -v /home/data:/opt/data --name tm2 tomcat
/usr/bin/supervisord

docker run -d -v -p 206:22 -p 7005:8080 -v /home/data:/opt/data --name tm3 tomcat
/usr/bin/supervisord
```

这里说一下 weblogic 的配置，大家知道 weblogic 有一个域的概念。如果要使用常规的 administrator +node 的方式部署，就需要在 supervisord 中分别写出 administartor server 和 node server 的启动脚本，这样做的优点是：

- 可以使用 weblogic 的集群，同步等概念
- 部署一个集群应用程序，只需要安装一次应用到集群上即可

缺点是：

- docker 配置复杂了
- 没办法自动扩展集群的计算容量，如需添加节点，需要在 administrator 上先创建节点，然后再配置心的容器 supervisor 启动脚本，然后再启动容器

另外种方法是将所有的程序都安装在 adminiserver 上面，需要扩展的时候，启动多个节点即可，它的优点和缺点和上一中方法恰恰相反。（目前我使用这种方式来部署开发和测试环境）

```
docker run -d -v -p 204:22 -p 7001:7001 -v /home/data:/opt/data --name node1 weblogic
/usr/bin/supervisord

docker run -d -v -p 205:22 -p 7002:7001 -v /home/data:/opt/data --name node2 weblogic
/usr/bin/supervisord

docker run -d -v -p 206:22 -p 7003:7001 -v /home/data:/opt/data --name node3 weblogic
/usr/bin/supervisord
```

这样在前端使用 nginx 来做负载均衡就可以完成配置了

十二、Docker 实战一多台物理主机之间的容器互联（暴露容器到真实网络中）

docker 默认的桥接网卡是 docker0。它只会在本机桥接所有的容器网卡，举例来说容器的虚拟网卡在主机上看一般叫做 veth*** 而 docker 只是把所有这些网卡桥接在一起，如下：

```
[root@opnvz ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.56847afe9799       no              veth0889
                                                        veth3c7b
                                                        veth4061
```

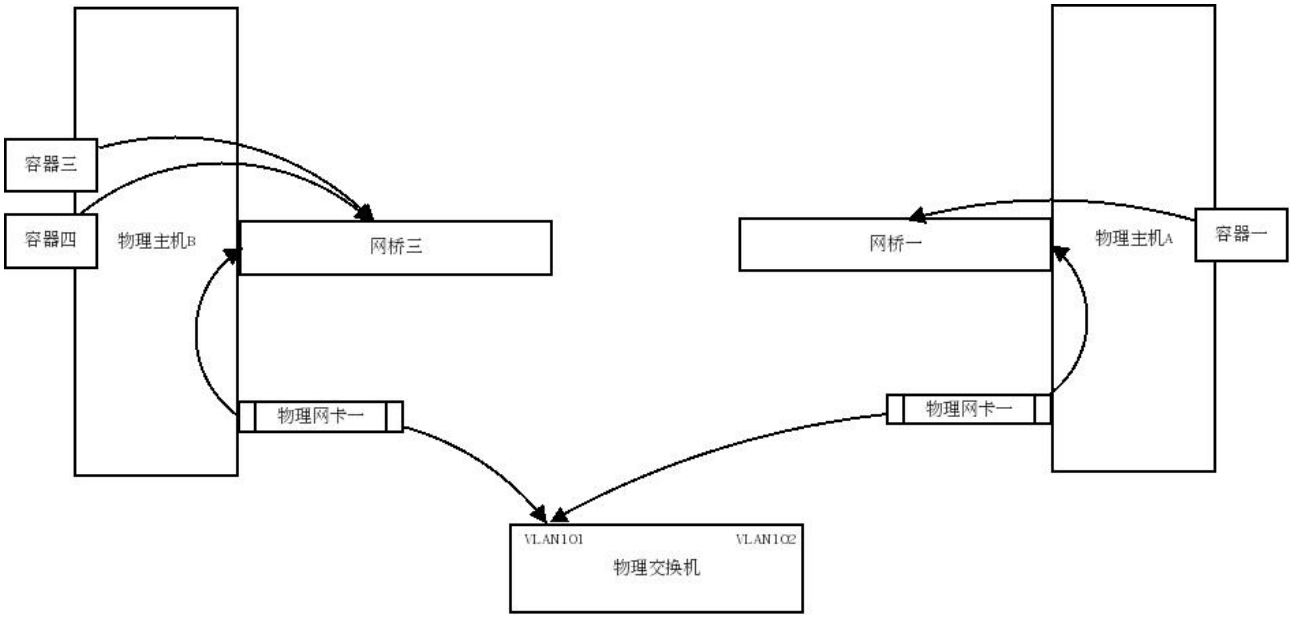
在容器中看到的地址一般是像下面这样的地址：

```
root@ac6474aeb31d:~# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
11: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 4a:7d:68:da:09:cf brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::487d:68ff:feda:9cf/64 scope link
       valid_lft forever preferred_lft forever
```

这样就可以把这个网络看成是一个私有的网络，通过 nat 连接外网，如果要让外网连接到容器中，就需要做端口映射，即-p 参数（更多原理参见本文第六小节）

如果在企业内部应用，或则做多个物理主机的集群，可能需要将多个物理主机的容器组到一个物理网络中来，那么就需要将这个网桥桥接到我们指定的网卡上。

1、拓扑图



主机 A 和主机 B 的网卡一都连着物理交换机的同一个 vlan 101,这样网桥一和网桥三就相当于在同一个物理网络中了，而容器一、容器三、容器四也在同一物理网络中了，他们之间可以相互通信，而且可以跟同一 vlan 中的其他物理机器互联。

2、ubuntu 示例

下面以 ubuntu 为例创建多个主机的容器联网:

```
创建自己的网桥,编辑/etc/network/interface 文件

auto br0

iface br0 inet static

address 192.168.7.31

netmask 255.255.240.0

gateway 192.168.7.254

bridge_ports em1

bridge_stp off

dns-nameservers 8.8.8.8 192.168.6.1

将 docker 的默认网桥绑定到这个新建的 br0 上面，这样就将这台机器上容器绑定到 em1 这个网卡所对应的物理网络上了。
```

ubuntu 修改/etc/default/docker 文件 添加最后一行内容

```
# Docker Upstart and SysVinit configuration file

# Customize location of Docker binary (especially for development testing).

#DOCKER="/usr/local/bin/docker"

# Use DOCKER_OPTS to modify the daemon startup options.

#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"


# If you need Docker to use an HTTP proxy, it can also be specified here.

#export http_proxy="http://127.0.0.1:3128/"


# This is also a handy place to tweak where Docker's temporary files go.

#export TMPDIR="/mnt/bigdrive/docker-tmp"


DOCKER_OPTS="-b=br0"
```

在启动 docker 的时候 使用-b 参数 将容器绑定到物理网络上。重启 docker 服务后，再进入容器可以看到它已经绑定到你的物理网络上了。

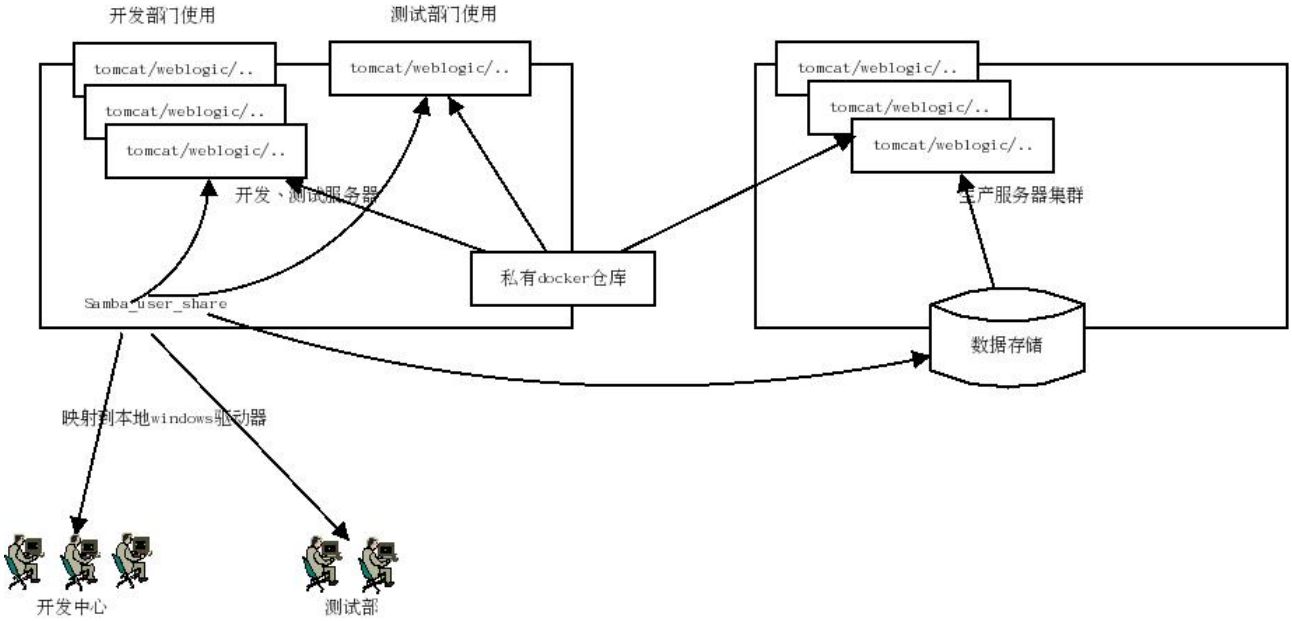
```
root@ubuntudocker:~# docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES

58b043aa05eb        desk_hz:v1          "/startup.sh"       5 days ago          Up 2 seconds        5900/tcp, 6080/tcp, 22/tcp   yanlx

root@ubuntudocker:~# brctl show

bridge name     bridge id               STP enabled     interfaces

br0             8000.7e6e617c8d53       no              em1

                                                        vethe6e5
```

这样就直接把容器暴露到你的物理网络上了，多台物理主机的容器也可以相互联网了。需要注意的是，这样就需要自己来保证容器的网络安全了。

十三、Docker 实战--中小企业 docker 环境搭建

docker 对于中小企业来说，搭建 paas 没有那个精力，也没那个必要，用做个人的 sandbox 用处又小了点，个人认为作为中小企业可以用 docker 来标准化开发、测试、生产环境。

画了简单的图：



docker 占用资源小，在一台 E5 128G 内存的服务器服务器上部署 100 个容器都绰绰有余，可以单独抽一个容器或则直接在宿主物理主机上部署 samba，利用 samba 的 home 分享方案将每个用户的 home 目录映射到开发中心和测试部门的 windows 机器上。可以针对项目组，由架构师搭建好一个标准的容器环境供项目组和测试部门使用，每个开发工程师可以拥有自己单独的容器，通过 docker run -v 将用户的 home 目录映射到容器中。需要提交测试时，只需要将代码移交给测试部门，然后分配一个容器使用-v 加载测试部门的 home 目录启动即可。这样，在公司内部的开发、测试基本就统一了，不会出现开发提交的代码，测试 部门部署不了的问题。

测试发布测试通过的报告后，架构师再一次检测容器环境，就可以直接交由部署工程师将代码和容器分别部署到生产环境中了。这种方式的部署横向性能的扩展性也极好。