

---

# 目錄

0. 介绍	1.1
1. 安装 docker	1.2
2. docker 的镜像和镜像源加速	1.3
3. docker 的容器	1.4
4. 理解 docker 镜像的层叠结构	1.5
5. 使用 Dockerfile 文件	1.6
6. docker 的数据卷	1.7
7. Docker Compose 的介绍与安装	1.8
8. 使用 compose 部署 GitLab 应用	1.9
9. 使用 compose 部署 Rocket.Chat 应用	1.10
10. docker 部署深入理解	1.11
11. 部署 owncloud 与 phpMyAdmin	1.12
12. 让 php-fpm 跑的 owncloud 应用 docker 化	1.13
13. docker 迁移 GitLab 项目	1.14

# docker 入门指南

零基础学习docker，从应用入手带你深入理解docker。

原文发布于我的个人博客：<https://www.rails365.net>

源码位于：<https://github.com/yinsigan/docker-tutorial>

电子版: [PDF](#) [Mobi](#) [ePbu](#)

联系我:

email: [hfpp2012@gmail.com](mailto:hfpp2012@gmail.com)

qq: 903279182

### 1. 介绍

docker有点像虚拟机技术那样，虚拟机是模拟了全部或部分的硬件，有一整套自己的操作系统，而docker不是，它只是一个进程，这个进程叫容器，这种叫容器技术，或隔离技术，它没有再启动一个操作系统，因为太耗资源，又太慢，它是隔离了linux内核，有自己的空间，比如说，自己的root账号，磁盘情况等。它很轻量级，启动很快，一个机器可以启动很多个docker容器进程。

说说几个应用场景，你没有理由不去用它。

- 前端工程师，不懂配置php或java的开发环境
- 持续集成测试
- 快速部署，一条命令，部署的环境一模一样
- 作为一个nodejs程序员，不懂php或ruby，想部署它们写的的应用，一条命令
- 方便地对程序进行资源配置，比如进程监控，内存限制
- 安装软件太慢，比如安装mongodb，要下载下来要好久，而docker只需要一条命令就可以跑起来

### 2. mac下安装

点击[这里](#)下载最新的mac版安装程序。

点击安装包下载完，运行即可。

### 3. linux下安装

一条命令即可。

```
curl -sSL https://get.daocloud.io/docker | sh
```

这条命令在**ubuntu 14.04**和**ubuntu 16.04**都可以成功安装**docker**。

安装成功后，可能会提示你这样的信息:

If you would like to use Docker as a non-root user, you should now consider

adding your user to the "docker" group with something like:

```
sudo usermod -aG docker vagrant
```

Remember that you will have to log out and back in for this to take effect!

`vagrant` 是你的用户名，可能你的用户名跟我的不一样。

意思就是说，你可以把当前用户加入到`docker`组，以后要管理`docker`就方便多了，不然你以后有可能要使用`docker`命令前，要在前面加 `sudo` 。

如果没加 `sudo` 就是类似这样的提示:

```
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.26/containers/json: dial unix /var/run/docker.sock: connect: permission denied
```

不过执行了 `sudo usermod -aG docker vagrant` 之后，你再重新登录(ssh)，就可以免去加 `sudo` 。

安装成功，需要把`docker`这个服务启动起来：

如果是`ubuntu 14.04`的系统，它会自动启动，你也可以使用下面的命令来启动。

```
$ sudo /etc/init.d/docker start
```

如果是`ubuntu 16.04`的系统，就用下面的命令：

```
$ sudo systemctl status docker.service
```

完结。

下一篇：[docker的镜像和镜像源加速\(二\)](#)



### 1. 介绍

上一篇: [安装docker\(一\)](#)

说镜像前，先说说容器，其实容器就是在镜像上运行的进程，docker容器就是一个进程，就好比，镜像是模板，存在于磁盘上，而容器就是模板生成来的实例，就有点类似，镜像是类，而对象是容器，对象是类生成的。

先来查看系统上的镜像，可以使用 `docker images` 命令。

你会发现是空的。

因为没有任何镜像。

我们来做个 `hello world` 。

首先要下载镜像，会使用 `docker pull` 命令，比如：

```
$ docker pull hello-world
```

你可能会发现，半天都没反应，更没下载下来，因为那个放镜像的地方在国外，反正在国内访问就有点慢，所以需要用到加速器。

### 2. 加速器

已经有人同步好了那些镜像，放在了国内的服务器，我们下载下来就会快很多。

这个就是加速器。

国内的加速器有好多，我们使用阿里云的加速器。

<https://cr.console.aliyun.com/#/imageList>

把下面的内容贴到终端上：

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<- 'EOF'
{
  "registry-mirrors": ["https://oyukeh0j.mirror.aliyuncs.com"]
}
EOF
```

然后再把docker重启，比如我是ubuntu 14.04，就可以这样：

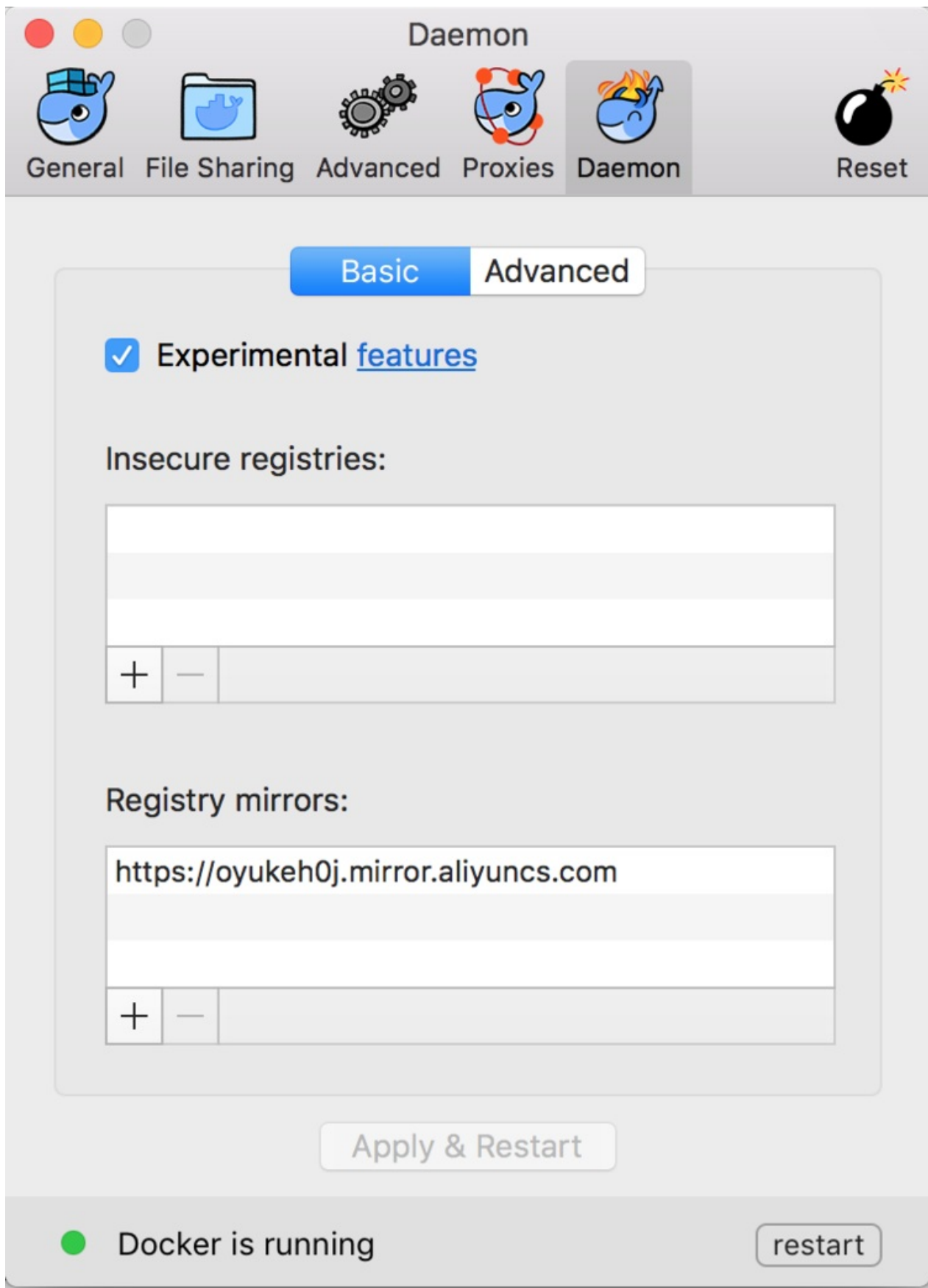
```
$ sudo /etc/init.d/docker restart
```

你再运行 `docker pull hello-world` 试试。

是不是很快？

如果看不出效果，可以试下 `docker pull sameersbn/gitlab`

上面说的是在ubuntu下进行操作，如果是mac呢？



完结。

下一篇：[docker的容器\(三\)](#)





## 1. 介绍

上一篇：[docker的镜像和镜像源加速\(二\)](#)

之前我们下载了 `hello-world` 这个镜像，现在把它运行起来。

使用 `docker run` 命令。

比如：

```
$ docker run hello-world
```

主要是输出一些文本信息，然后就退出了。

使用 `docker ps` 可以查看容器的进程。

不过是空的，因为一运行就退出，只是输出信息，输出完就结束了。

可以使用 `docker ps -a` 来查看退出过的进程。

比如：

```
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS		PORTS	
NAMES			
b25a96c17bf4	hello-world	"/hello"	About a minute ago
Exited (0)			
mystifying_saha			

## 2. 运行容器

上面只是演示hello world而已，我们运行些有用点的容器。

比如：

```
$ docker run -it ubuntu bash
```

运行的是ubuntu这个镜像，如果你的系统上没有这个镜像，会先下载下来。

`bash` 是表示进入那个ubuntu的镜像的容器的shell，`-t` 选项让Docker分配一个伪终端（`pseudo-tty`）并绑定到容器的标准输入上，`-i` 则让容器的标准输入保持打开。

这样就相当于进入了另一台ubuntu。

而且这个ubuntu跟你原来的系统是隔离的。

这样也没多大意义，我们经常会用docker来跑一些服务，比如web服务。

那就运行一个nginx容器试试。

```
$ docker run -d -p 80:80 --name webserver nginx
```

镜像名称是 `nginx`，`--name` 表示为这个容器取个名称叫 `webserver`。

运行之后，你用 `docker ps` 查看一下容器。

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
CREATED	STATUS	PORTS
NAMES		
a1d4b615709b	nginx	"nginx -g 'daemon ...'"
5 seconds ago	Up 4 seconds	0.0.0.0:80->80/tcp, 443
/tcp	webserver	

然后你可以使用 `curl http://127.0.0.1` 查看一下，是否运行了nginx服务。

使用 `docker stop` 命令可以停止这个容器的运行。

比如：

```
$ docker stop webserver
```

停止之后输入 `docker ps` 发现是空的，这个时候要输入 `docker ps -a` 才能查看退出的容器。

可以使用下面这条命令，清除所有已退出的容器。

```
$ docker rm -f $(docker ps -a | grep Exit | awk '{ print $1 }')
```

之前运行nginx容器的时候，有使用 `-p 80:80` 这个参数。

这个表示端口映射。

解释前，我们先来改一下。

```
$ docker run -d -p 8080:80 --name webserver nginx
```

查看一下运行状态：

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
CREATED	STATUS	PORTS
NAMES		
45e47ed889f8	nginx	"nginx -g 'daemon ...'"
4 seconds ago	Up 3 seconds	443/tcp, 0.0.0.0:8080->
80/tcp	webserver	

现在你可以需要使用 `curl http://127.0.0.1:8080` 才能访问了。

也就是说，`-p` 参数中，第一个端口是暴露在外面的端口，外面可以访问的端口。

至于 `-d` 嘛，就是以守护态运行。

有些应用根本就不需要暴露接口在外面，比如下面这个：

```
$ docker ps
```

mposenginx_gitlab_1		
7e4493b736d2	sameersbn/postgresql:9.5-4	"/sbin/entrypoi
nt.sh"	2 days ago	Up 2 days
		5432/tcp
		gitlabcomposenginx_pos
tgresql_1		
8cd6c90f9a52	sameersbn/redis:latest	"/sbin/entrypoi
nt.sh "	2 days ago	Up 2 days
		6379/tcp

这两个端口在外部是无法访问的，没有绑定在0.0.0.0这个ip上，这个有什么用呢。

其实有个参数是 `--link`，供单机的容器之间打通一个网络通道，这样不必通过ip来访问，而是一个别名。

比如，先运行两个容器：

```
$ docker run --name gitlab-postgresql -d \  
  --env 'DB_NAME=gitlabhq_production' \  
  --env 'DB_USER=gitlab' --env 'DB_PASS=password' \  
  --env 'DB_EXTENSION=pg_trgm' \  
  --volume /srv/docker/gitlab/postgresql:/var/lib/postgresql \  
  sameersbn/postgresql:9.6-2  
  
$ docker run --name gitlab-redis -d \  
  --volume /srv/docker/gitlab/redis:/var/lib/redis \  
  sameersbn/redis:latest
```

这两个容器有两个名称，分别是 `gitlab-postgresql` 和 `gitlab-redis`。

然后再启动一个暴露接口的容器，去连接这两个容器。

```
$ docker run --name gitlab -d \  
  --link gitlab-postgresql:postgresql --link gitlab-redis:redisio \  
  --publish 10022:22 --publish 10080:80 \  
  --env 'GITLAB_PORT=10080' --env 'GITLAB_SSH_PORT=10022' \  
  --env 'GITLAB_SECRETS_DB_KEY_BASE=long-and-random-alpha-numeric-string' \  
  --env 'GITLAB_SECRETS_SECRET_KEY_BASE=long-and-random-alpha-numeric-string' \  
  --env 'GITLAB_SECRETS_OTP_KEY_BASE=long-and-random-alpha-numeric-string' \  
  --volume /srv/docker/gitlab/gitlab:/home/git/data \  
  sameersbn/gitlab:8.16.6
```

其中 `postgresql` 和 `redisio` 是容器的主机别名。

使用 `docker restart` 命令可以重启容器。

还有一个参数比较常用，就是 `-rm`，这个参数是说容器退出后随之将其删除。默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 `docker rm`。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 `-rm` 可以避免浪费空间。

完结。

下一篇: [理解docker镜像的层叠结构\(四\)](#)

### 1. 介绍

上一篇: [docker的容器\(三\)](#)

可能你使用 `docker pull sameersbn/gitlab` 命令的时候，会发现要下载好多 images，因为 docker 的 images 是层叠结构的。

这个有点像 git。就是最上一层的 images，可能是下一层的 images，为基础构建的。

### 2. docker commit

要先理解这个，先来看个例子。

之前我们使用过下面的命令来运行过一个 nginx 服务。

```
$ docker run --name webserver -d -p 80:80 nginx
```

现在我们进入到这个容器，改点东西。

```
$ docker exec -it webserver bash
```

然后：

```
root@3729b97e8226:/# echo '<h1>Hello, Docker!</h1>' > /usr/share  
/nginx/html/index.html  
root@3729b97e8226:/# exit
```

你再使用 `curl http://localhost` 看看，是不是页面变了。

然后你再使用 `stop` 命令把这个容器停掉，然后再启动，你会发现又变回原来的样子。

我们之前修改了容器的内容。

来看看改了哪些东西：

```
$ docker diff webserver
C /root
A /root/.bash_history
C /run
A /run/nginx.pid
C /usr
C /usr/share
C /usr/share/nginx
C /usr/share/nginx/html
C /usr/share/nginx/html/index.html
C /var
C /var/cache
C /var/cache/nginx
A /var/cache/nginx/client_temp
A /var/cache/nginx/fastcgi_temp
A /var/cache/nginx/proxy_temp
A /var/cache/nginx/scgi_temp
A /var/cache/nginx/uwsgi_temp
```

我们可以把这个容器的改变保存下来，成为一个镜像，下次就能直接运行了。

```
$ docker commit \
  --author "hfpp2012 <hfpp2012@aliyun.com>" \
  --message "修改了默认网页" \
  webserver \
  nginx:v2
```

现在来查看一下nginx的镜像，发现有两个，如下：

```
$ docker images nginx
```

REPOSITORY	TAG	IMAGE ID	CREATED
nginx	v2	f51e5af097aa	5 minutes ago
nginx	latest	db079554b4d2	11 days ago



因为v2那个版本的nginx镜像，才只是改一点点内容，所以他们两个镜像的占用磁盘的容量只是182MB多点，而不是182 \* 2 MB。

现在可以把v2这个版本的nginx镜像运行起来。

```
$ docker run --name web2 -d -p 81:80 nginx:v2
```

### 3. 慎用 docker commit

使用 `docker commit` 虽然能保存镜像或创建镜像，但是一般情况下我们不会这么用，我们会使用 `Dockerfile` 来构建镜像。

`docker commit` 有一些不好的地方，上面的例子也可以看出来，我只是改了一个文件，上面却显示改了很多，比如临时文件，记录命令的文件(`.bash_history`)，还有pid文件啥的。

而且使用 `docker commit` 创建的镜像，别人也无法知道你是如何创建的，万一里面藏着什么东西，别人都不知道，很不安全。

所以嘛，我们接下来会使用 `Dockerfile` 来创建镜像。

完结。

下一篇：[使用Dockerfile文件\(五\)](#)

### 1. 介绍

上一篇：[理解docker镜像的层叠结构\(四\)](#)

Dockerfile是一个纯文本文件，内容是可读的，就是把构建镜像的指令一条条放在这个文件中，然后就可以build出来一个镜像。

### 2. 使用

我们来实践一下。

创建一个最简单的Dockerfile文件。

```
$ mkdir mynginx  
$ cd mynginx  
$ touch Dockerfile
```

内容如下：

```
FROM nginx  
RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index  
.html
```

我们以此Dockerfile文件，来生成了一个镜像，使用 `docker build` 命令。

只要

```
$ docker build .
```

会显示这样的信息：

```
Sending build context to Docker daemon 2.048 kB
Step 1/2 : FROM nginx
---> db079554b4d2
Step 2/2 : RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx
/html/index.html
---> Running in 64127d3ff089
---> 6fb1fd56e2dd
Removing intermediate container 64127d3ff089
Successfully built 6fb1fd56e2dd
```

build成功之后可以查看一下：

```
$ docker images nginx
REPOSITORY          TAG                 IMAGE ID            CREA
TED                 SIZE
<none>              <none>             6fb1fd56e2dd       4 se
conds ago           182 MB
nginx               v2                 f51e5af097aa       44 m
inutes ago          182 MB
```

其中可看出，`REPOSITORY` 和 `TAG` 部分都为none，其实我们build的时候可以给个名称标识一下。

```
$ docker build -t nginx:v3 .
```

现在就有了。

```
$ docker images nginx
REPOSITORY          TAG                 IMAGE ID            CREA
TED                 SIZE
nginx              v3                 6fb1fd56e2dd       4 mi
nutes ago           182 MB
nginx              v2                 f51e5af097aa       48 m
inutes ago          182 MB
```

## 3. 参数解释

回到之前的Dockerfile文件的内容。

```
FROM nginx
RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
```

只有两个指令，分别是 `FROM` 和 `RUN` 。

`FROM` 代表的是基础镜像，`RUN` 表示的是运行的命令。

值得注意的是，每条指令都会创建一层镜像，不要单独写太多的 `RUN` 语句，而是要串起来。

例如，类似下面的作法是不对的。

```
FROM ruby:2.2.2
# update
RUN apt-get update -qq
# RUN apt-get upgrade -qq
# RUN apt-get dist-upgrade -qq
# RUN apt-get autoremove -qq
# for build essential lib
RUN apt-get install -y build-essential
# for postgres
RUN apt-get install -y libpq-dev
# for nokogiri
RUN apt-get install -y libxml2-dev libxslt1-dev
# for a JS runtime
RUN apt-get install -y nodejs
# for imagemagick
RUN apt-get install -y imagemagick
# for vim
RUN apt-get install -y vim
# set gem sources
RUN gem sources -r https://rubygems.org/
RUN gem sources -a https://ruby.taobao.org/
ENV APP_HOME /var/www/coomo_store
RUN mkdir -p $APP_HOME
WORKDIR $APP_HOME
ADD Gemfile* $APP_HOME/
RUN bundle install --jobs=4
ADD . $APP_HOME
```

还有很多命令，会被经常用到，比如 `ARG`，`ENV`，`EXPOSE`，`WORKDIR`，`CMD`。

下面来介绍一下。

`ARG` 表示的是设置一个参数，在 `build` 的时候由外面通过命令行参数的形式传过来。

比如：

```
$ docker build --build-arg PORT=3000 --build-arg NPM_TOKEN=e43d3f2c-e7b7-4522-bf74-b7d2863eddf7 .
```

传了两个参数分别为 `PORT` 和 `NPM_TOKEN` 。

`ENV` 是设置环境变量，以后这个变量可以传到docker容器内部中去。

`EXPOSE` 表示暴露的端口号。

`WORKDIR` 就是指定工作目录啦。

`CMD` 是容器启动的命令。

来看一个实例，具体体会一下这些命令。

```
from node:6.9

ARG PORT
ARG NPM_TOKEN

ENV PORT ${PORT}
ENV NPM_TOKEN ${NPM_TOKEN}

EXPOSE ${PORT}

ADD . /code
WORKDIR /code

RUN npm install
RUN npm run build
CMD npm run docker
```

下面这个链接是官方给的关于Dockerfile的文档：

<https://docs.docker.com/engine/reference/builder/#dockerfile-examples>

完结。

下一篇：[docker的数据卷\(六\)](#)

### 1. 介绍

上一篇：[使用Dockerfile文件\(五\)](#)

之前有说过，在docker容器上进行的更改并不会主动被保存下来，除非你commit了，不然你重新运行这个镜像，生成新的容器，之前容器的内容就会没有了，因为容器只是一个进程而已。

所以说，容器运行时应该尽量保持容器存储层不发生写操作。

但是，有时候我们需要存储持久化的数据，比如数据库，你的数据都在容器中，肯定是不行的，因为一退出就没有了。

这个时候需要用到 数据卷 。

数据卷 就是可以让你把主机上的数据以挂载的方式链接到容器中，这样不同的容器也能共享，而且数据也不会因为容器的退出而丢失。

这个 数据卷 会被经常使用。

### 2. 使用

下面我们来体会一下 数据卷 的功能。

```
$ docker run -d -v ~/mynginx:/a -p 80:80 --name webserver nginx
```

这里命令会挂载主机的目录 ~/mynginx 到容器中的目录 /a 。

你可以试验一下，分别在两端更改内容，比如新建一个文件，看是不是都变化了。

我们也可以创建数据卷容器，数据卷容器也是一个正常的容器，这种容器可以为其他容器提供和共享数据。

比如，下面创建了一个数据卷容器：

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

然后其他容器要使用这个数据卷容器的话，只要使用 --volumes-from 参数即可。

```
$ sudo docker run -d --volumes-from dbdata --name db1 training/postgres
$ sudo docker run -d --volumes-from dbdata --name db2 training/postgres
```

完结。

下一篇：[Docker Compose 的介绍与安装 \(七\)](#)



### 1. 介绍

上一篇：[docker的数据卷\(六\)](#)

如果别人做了一个docker镜像服务，我最喜欢的就是它提供一个docker compose的配置文件给我，我一改，就能跑起整个应用。

docker compose能够运行容器，也就是说来实现部署，不仅如此，有时候你的应用不止一个容器，compose也能把多个容器联接起来。

### 2. 安装

这里有官方的安装指南：<https://docs.docker.com/compose/install/>

我选择用python的pip来安装，只需要一条命令即可。

```
$ sudo pip install -U docker-compose
```

如果你没有pip，那可以用下面的命令先安装一下：

```
$ sudo apt-get install python-pi
```

之后就可以直接使用 `docker-compose` 命令了。

除此之外，还可以从这里下载dokcer-compose的二进制文件。

<https://github.com/docker/compose/releases>

复制好后要给他加上可执行权限。

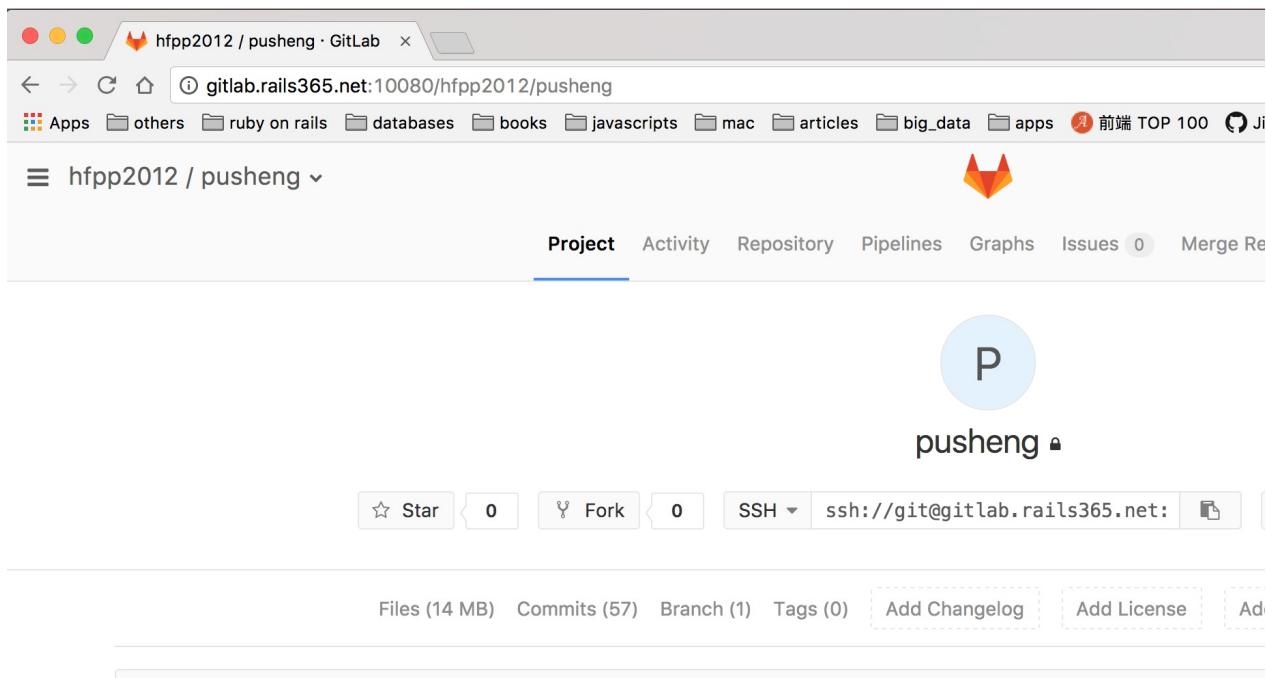
完结。

下一篇：[使用compose部署gitlab应用\(八\)](#)

### 1. 介绍

上一篇：[Docker Compose的介绍与安装\(七\)](#)

之前介绍过compose，现在来使用它，直接来部署一个gitlab应用。



### 2. 下载镜像

我们部署的是[sameersbn/docker-gitlab](#)这个镜像。

首先把它下载下来。

```
$ docker pull sameersbn/gitlab
```

### 3. 配置文件

我们不需要去run它，只需要先下载一个compose的配置文件。

```
$ wget https://raw.githubusercontent.com/sameersbn/docker-gitlab/master/docker-compose.yml
```

打开这个文件，把它的内容按照你自己的需要修改，比如改改github登录的配置信息，邮件发送的配置信息等。

比如我的配置如下：

```
version: '2'
services:
  redis:
    restart: always
    image: sameersbn/redis:latest
    command:
      - --loglevel warning
    volumes:
      - /srv/docker/gitlab/redis:/var/lib/redis:Z
  postgresql:
    restart: always
    image: sameersbn/postgresql:9.5-4
    volumes:
      - /srv/docker/gitlab/postgresql:/var/lib/postgresql:Z
    environment:
      - DB_USER=gitlab
      - DB_PASS=password
      - DB_NAME=gitlabhq_production
      - DB_EXTENSION=pg_trgm
  gitlab:
    restart: always
    image: sameersbn/gitlab:8.15.2
    depends_on:
      - redis
      - postgresql
    ports:
      - "10080:80"
      - "10022:22"
    volumes:
      - /srv/docker/gitlab/gitlab:/home/git/data:Z
    environment:
      - DEBUG=false
      - DB_ADAPTER=postgresql
      - DB_HOST=postgresql
      - DB_PORT=5432
      - DB_USER=gitlab
      - DB_PASS=password
      - DB_NAME=gitlabhq_production
```

```
- REDIS_HOST=redis
- REDIS_PORT=6379
- TZ=Asia/Beijing
- GITLAB_TIMEZONE=Beijing
- GITLAB_HTTPS=false
- SSL_SELF_SIGNED=false
- GITLAB_HOST=gitlab.rails365.net
- GITLAB_PORT=10080
- GITLAB_SSH_PORT=10022
- GITLAB_RELATIVE_URL_ROOT=
- GITLAB_SECRETS_DB_KEY_BASE=xxx
- GITLAB_SECRETS_SECRET_KEY_BASE=xxx
- GITLAB_SECRETS_OTP_KEY_BASE=JBSWY3DPEHPK3PXP
- GITLAB_ROOT_PASSWORD=
- GITLAB_ROOT_EMAIL=
- GITLAB_NOTIFY_ON_BROKEN_BUILDS=true
- GITLAB_NOTIFY_PUSHER=false
- GITLAB_EMAIL=hfpp2012@rails365.net
- GITLAB_EMAIL_REPLY_TO=noreply@example.com
- GITLAB_INCOMING_EMAIL_ADDRESS=reply@example.com
- GITLAB_BACKUP_SCHEDULE=daily
- GITLAB_BACKUP_TIME=01:00
- SMTP_ENABLED=true
- SMTP_DOMAIN=192.168.33.10
- SMTP_HOST=smtpdm.aliyun.com
- SMTP_PORT=25
- SMTP_USER=hfpp2012@rails365.net
- SMTP_PASS=xxxxxx
- SMTP_STARTTLS=true
- SMTP_AUTHENTICATION=plain
. . . .
- OAUTH_GITHUB_API_KEY=8910bb915c112971520b
- OAUTH_GITHUB_APP_SECRET=dc04e2fa55626bd0b1433c35adb57d8551
1d5772
```

有些地方被我省略了，还有些地方被我用 `xxx` 隐藏了。

## 4. compose up

现在改完之后就可以直接运行了。我们使用下面的命令来运行。

```
$ docker-compose up
```

运行的时候会发现会下载其他的镜像，比如postgresql，还有redis。因为compose是允许不同的镜像容器互相链接的，从上面的配置文件中的内容也可以看出来。

如果要以守护态运行，可以加 `-d` 参数。

## 5. nginx配置

这个镜像默认是会启用10080端口访问，我们可以用nginx反向代理到这个端口上。

比如：gitlab.conf。

```
upstream gitlab {
    server                127.0.0.1:10080;
}
server {
    listen                80;
    server_name           gitlab.rails365.net;
    server_tokens         off;
    root                  /dev/null;
    location / {
        proxy_read_timeout    300;
        proxy_connect_timeout 300;
        proxy_redirect        off;
        proxy_set_header      X-Forwarded-Proto $scheme;
        proxy_set_header      Host                $http_host;
        proxy_set_header      X-Real-IP           $remote_addr;
        proxy_set_header      X-Forwarded-For     $proxy_add_x_forwa
rded_for;
        proxy_set_header      X-Frame-Options    SAMEORIGIN;
        proxy_pass             http://gitlab;
    }
}
```

完结。

下一篇：[使用compose部署Rocket.Chat应用\(九\)](#)

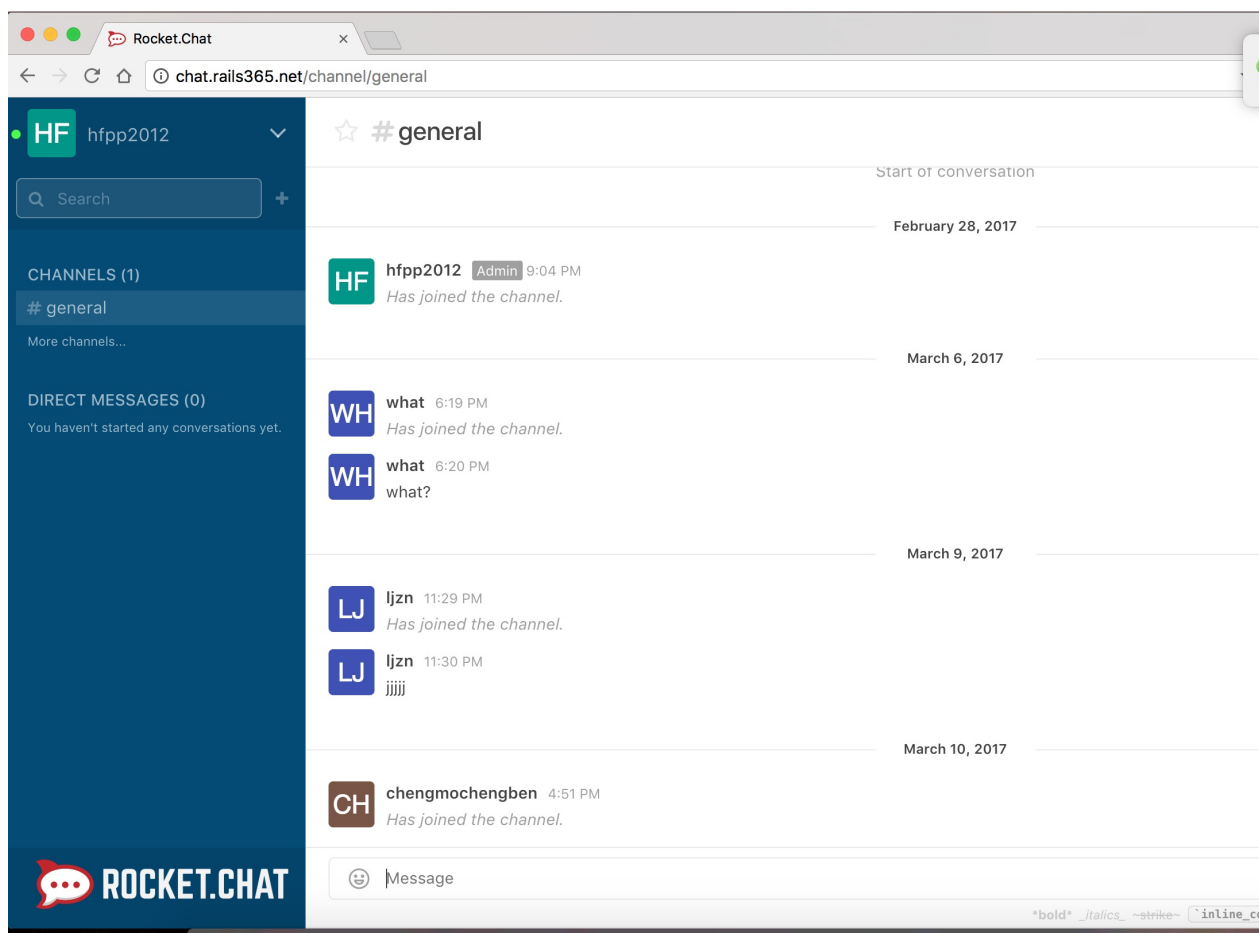
### 1. 介绍

上一篇：[使用compose部署gitlab应用\(八\)](#)

**Rocket.Chat**是类似一个**slack**的聊天室应用，要部署它很简单，官方就有文档，只要照着那文档来即可。

它的文档地址是：

<https://rocket.chat/docs/installation/paas-deployments/aliyun/>



### 2. 部署

最好切换到**root**账号下部署。

第一步如下：

```
mkdir /home/rocketchat
cd /home/rocketchat
mkdir data
mkdir dump
```

接着新建docker-compose.yml文件：

```
db:
  image: mongo
  volumes:
    - $PWD/data:/data/db
    - $PWD/dump:/dump
  command: mongod --smallfiles
web:
  image: rocketchat/rocket.chat
  environment:
    - MONGO_URL=mongodb://db:27017/meteor
    - ROOT_URL=http://your-ip-address:8818
  links:
    - db:db
  ports:
    - 8818:3000
```

其中 `your-ip-address` 改成你自己的外网的域名或ip。

用的是官方提供的docker镜像[RocketChat](#)。

然后部署：

```
$ docker-compose up
```

如果没什么问题，就使用 `8818` 端口访问。

最后在线上可以加上 `-d` 开启守护态。

### nginx配置

最后加上nginx的反向代理配置。



```
upstream chat {
    server 127.0.0.1:8818;
}
server {
    listen                80;
    server_name           chat.rails365.net;
    error_log /var/log/nginx/rocketchat.access.log;
    location / {
        proxy_pass http://chat/;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forward-For $proxy_add_x_forwarded_for;

        proxy_set_header X-Forward-Proto http;
        proxy_set_header X-Nginx-Proxy true;
        proxy_redirect off;
    }
}
```

完结。

下一篇：[docker部署深入理解之数据库\(十\)](#)

### 1. 介绍

上一篇：[使用compose部署Rocket.Chat应用\(九\)](#)

之所以来介绍关于数据库的部署，是因为数据库很重要，经常会被用到，也是为了下面两篇文章作铺垫。

应该说最主要是为了深入理解docker的几个概念，比如匿名卷，数据卷，网络，还有 `docker-compose` 的写法。

学了这些，和理解了这些知识，应该说，以后部署一个别人给你的，或网络上存在的成熟应用，是没有问题的。

### 2. 端口映射

数据库会以mysql作为例子，postgresql等数据库是一样的。

<https://github.com/docker-library/docs/tree/master/mysql>

这里有它的使用说明。

可以先看看的。

首先来思考一下，要运行一个mysql服务最基本的需求，可能要先设置一个账号名和密码，不然如何使用呢？

所以：

运行一个最简单的mysql镜像，可以这样：

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql
```

然后用 `docker ps` 查看一下，是这样的：

CONTAINER ID	IMAGE	COMMAND
CREATED	STATUS	PORTS
NAMES		
6679faa19a49	mysql	"docker-entrypoint..."
3 seconds ago	Up 3 seconds	3306/tcp
some-mysql		

以 `mysql` 镜像为基础，创建了一个容器，名为 `some-mysql`，并设置 `root` 密码为 `my-secret-pw`。

但是这个容器是没有暴露接口的，没有暴露接口的容器也是能登录的，不过需要特殊的手段，没有暴露接口，你就不能用外部的 `mysql` 等客户端来登录。

如果要暴露接口可以这么做：

```
docker run --name some-mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql
```

注意，可以使用 `docker stop some-mysql && docker rm -f $(docker ps -a | grep Exit | awk '{ print $1 }')` 来停止使用这个容器

现在使用 `docker ps` 来查看一下。

CONTAINER ID	IMAGE	COMMAND
CREATED	STATUS	PORTS
NAMES		
feab724df3b6	mysql	"docker-entrypoint..."
3 seconds ago	Up 2 seconds	0.0.0.0:3306->3306/tcp
some-mysql		

可见，端口那个部分由 `3306/tcp` 变成了 `0.0.0.0:3306->3306/tcp`。

`0.0.0.0` 表示外部的主机能访问到。

### 3. 环境变量

之前有说过，启动这个 `mysql` 容器的时候，已经设置了 `root` 的密码，是通过环境变量的方式设置的，也就是一个参数 `-e`。

除此之外，你还可以设置其他变量：

- `MYSQL_DATABASE` 指定使用的数据库，默认会创建
- `MYSQL_USER` 如果不想使用`root`账号，可以新建一个账号来使用
- `MYSQL_PASSWORD` 账号的密码

也就是说，`mysql`镜像会利用这些传过来的变量，来做一些操作，比如说创建数据库，创建`root`密码等。这样你才能去连接这个`mysql`服务。

## 4. 匿名卷

我们有一个重要的功能需要注意，就是把数据库的数据保存下来，不会因为`container`停止了，数据就没了。

这个要用到一个叫 匿名卷 的问题。

```
# https://github.com/docker-library/mysql/blob/ee989d0666458d08d
d79c55f7abb62be29a9cb3d/5.5/Dockerfile
VOLUME /var/lib/mysql
```

这个 `mysql` 镜像已经做好了匿名卷，就是 `/var/lib/mysql` 这个目录，意思就是说任何向 `/var/lib/mysql` 中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化。

注意：`/var/lib/mysql`是容器中的路径

当然，我们可以使用自己的目录来覆盖这个挂载设置。

使用 `-v` 参数即可：

```
docker run --name some-mysql -p 3306:3306 -v /home/ubuntu/ownclo
ud/mysql_data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw
-d mysql
```

当然你可以挂载你任何想持久化的目录和文件。

## 5. 深入环境变量

开启了`mysql`服务，总要被连接或使用吧。

比如我要进去把数据导出来，可以这么做：

```
$ docker exec some-mysql sh -c 'exec mysqldump --all-databases -u root -p"$MYSQL_ROOT_PASSWORD"' > /some/path/on/your/host/all-databases.sql
```

`$MYSQL_ROOT_PASSWORD` 表示引用容器内的变量。

`MYSQL_ROOT_PASSWORD` 这个变量之前有提过，在 `Dockerfile` 或一些其他启动脚本就可以先定义好，你再传进去就好了。

整条命令，表示进入容器中执行 `mysqldump` 命令，这个命令使用的 `mysql` 用户是 `root`，连接密码，是使用是容器内部的变量 `MYSQL_ROOT_PASSWORD`。

那如何来证明呢？

可以这么做：

```
$ docker exec some-mysql sh -c 'echo "$MYSQL_ROOT_PASSWORD"'
my-secret-pw
```

它会输出你之前启动 `mysql` 服务器端容器时传过去的密码：`my-secret-pw`。

我们再传一个自己的变量，然后重启开启容器：

```
$ docker run --name some-mysql -p 3306:3306 -v /home/ubuntu/owncloud/mysql_data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -e MYENV=MYVALUE -d mysql
```

传的变量名是 `MYENV`，值是 `MYVALUE`。

结果输出如下：

```
$ docker exec some-mysql sh -c 'echo "$MYENV"'
MYVALUE
```

证明我们的猜想是正确的：所以传给容器的变量都被保存着，进入容器都能使用。

除此之外，我们还可以开启一个客户端的容器进程去连接服务器端的容器进程。

```
$ docker run -it --link some-mysql:mysql --rm mysql sh -c 'exec  
mysql -h"$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT"  
-uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'
```

你会发现，开启一个客户端，并且进入其中了。

这部分 `--link some-mysql:mysql` 先不看，我们先来看看后面的环境变量。

怎么又多出了几个变量，不过这些变量都能查看到：

```
$ docker run -it --link some-mysql:mysql --rm mysql sh -c 'env'
```

输出如下：

```
MYSQL_ENV_MYENV=MYVALUE  
HOSTNAME=811b319ce670  
MYSQL_MAJOR=5.7  
SHLVL=0  
HOME=/root  
MYSQL_ENV_MYSQL_MAJOR=5.7  
TERM=xterm  
MYSQL_PORT_3306_TCP_ADDR=172.17.0.2  
MYSQL_ENV_MYSQL_ROOT_PASSWORD=my-secret-pw  
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  
MYSQL_VERSION=5.7.17-1debian8  
MYSQL_ENV_GOSU_VERSION=1.7  
MYSQL_PORT_3306_TCP_PORT=3306  
MYSQL_PORT_3306_TCP_PROTO=tcp  
MYSQL_PORT=tcp://172.17.0.2:3306  
MYSQL_PORT_3306_TCP=tcp://172.17.0.2:3306  
MYSQL_ENV_MYSQL_VERSION=5.7.17-1debian8  
GOSU_VERSION=1.7  
PWD=/  
MYSQL_NAME=/competent_lamport/mysql
```

之前使用 `MYSQL_ROOT_PASSWORD` 这种变量好好的，为何又要使用 `MYSQL_ENV_MYSQL_ROOT_PASSWORD` 这个变量呢，原因是因为用了 `--link`。

把 `--link` 这部分去掉再来试试。

```
$ docker run -it --rm mysql sh -c 'env'
HOSTNAME=744912e7a7f6
MYSQL_MAJOR=5.7
SHLVL=0
HOME=/root
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MYSQL_VERSION=5.7.17-1debian8
GOSU_VERSION=1.7
PWD=/
```

看到没？少了好多变量。

想要用什么，拿来用就好了。

其中有一个变量叫 `MYSQL_PORT_3306_TCP_ADDR`。

这个变量跟docker的网络接口有关。

## 6. link和网络接口

先来看看 `--link some-mysql:mysql` 这个参数。

以 `:` 分隔分为两部分，第一部分是容器的名称，第二部是网络连接接口的别名。

表示要连接 `some-mysql` 这个容器，这个容器是一个mysql服务器程序，并且取了一个别名叫 `mysql`，这个别名有点类似于 `localhost`，有点儿像ip地址的作用，只是在容器或容器之间用。

这个别名会发挥着很大的作用的，连接这个mysql服务器(`some-mysql`)的容器，都可以使用这个别名来找到这台mysql服务器的ip地址。

下面的章节我们会介绍这个别名的使用。

我们先把这个别名改一下。

```
$ docker run -it --link some-mysql:db --rm mysql sh -c 'exec env'

HOSTNAME=ab6048fd4ccd
DB_PORT=tcp://172.17.0.2:3306
MYSQL_MAJOR=5.7
SHLVL=0
DB_PORT_3306_TCP=tcp://172.17.0.2:3306
DB_ENV_MYSQL_VERSION=5.7.17-1debian8
HOME=/root
DB_NAME=/clever_goldberg/db
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MYSQL_VERSION=5.7.17-1debian8
DB_ENV_MYSQL_MAJOR=5.7
DB_PORT_3306_TCP_ADDR=172.17.0.2
GOSU_VERSION=1.7
DB_ENV_MYSQL_ROOT_PASSWORD=my-secret-pw
PWD=/
DB_ENV_GOSU_VERSION=1.7
DB_PORT_3306_TCP_PORT=3306
DB_PORT_3306_TCP_PROTO=tcp
```

看下输出，之前类似以 `MYSQL` 开头的变量 `MYSQL_PORT_3306_TCP_ADDR` 的变量都变成了以 `DB` 开头的变量 `DB_PORT_3306_TCP_ADDR`。

`DB_PORT_3306_TCP_ADDR` 是个ip地址，它代表的是这台被连接的mysql服务器的容器的ip地址。

其实用一台命令也可以查出这个ip地址。

```
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' container_name_or_id
```

`container_name_or_id` 表示容器的hash值，可以写上mysql服务器那个容器的名称。

比如：



```
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}' some-mysql
172.17.0.2
```

果然这个地址一样的。

我们来验证一下。

进入 `some-mysql` 那个容器：

```
$ docker exec -it some-mysql bash
```

然后：

```
$ cat /etc/hosts
127.0.0.1    localhost
::1         localhost ip6-localhost ip6-loopback
fe00::0     ip6-localnet
ff00::0     ip6-mcastprefix
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters
172.17.0.2   9acca0e7b7f8
```

看最后一行 `172.17.0.2 9acca0e7b7f8`。 `9acca0e7b7f8` 是 `some-mysql` 这个容器的hash值。

总结一下，也就是说，如果容器要被外部使用，就要暴露接口，如果容器间使用就可以不必要暴露接口。

下一篇文章会写关于在 `docker-compose` 中使用 `--link` 和 `docker network` 这个关于网络连接的指令。

完结。

下一篇：[部署owncloud与phpMyAdmin\(十一\)](#)

## 1. 介绍

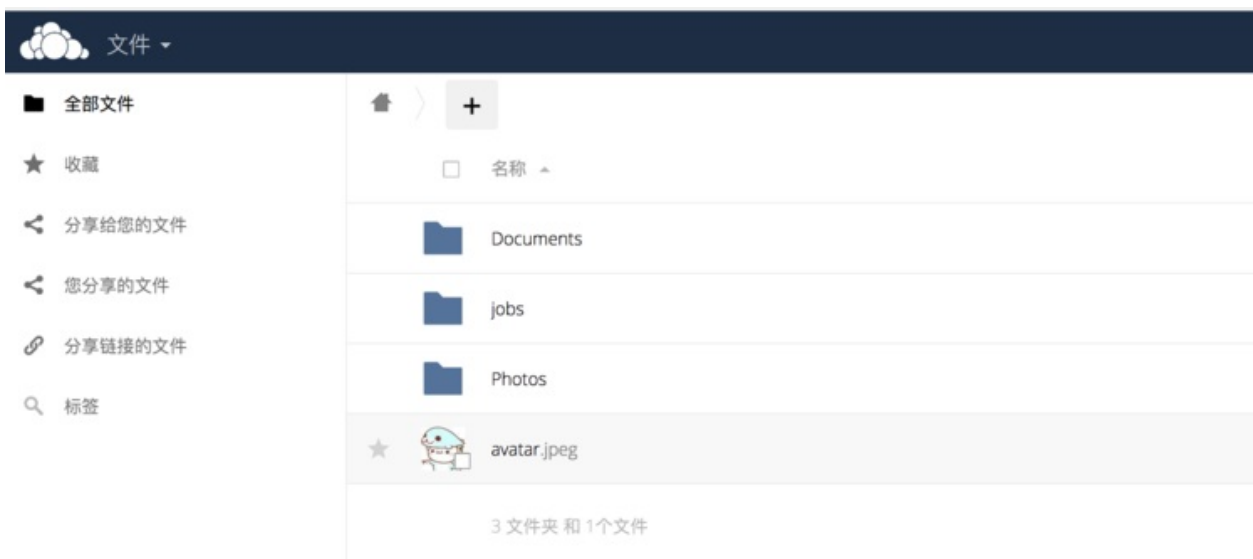
上一篇：[docker部署深入理解\(十\)](#)

- [owncloud 主页](#)
- [owncloud 客户端](#)
- [owncloud docker 文档](#)
- [phpMyAdmin docker 镜像](#)
- [docker-compose.yml 2 写法](#)

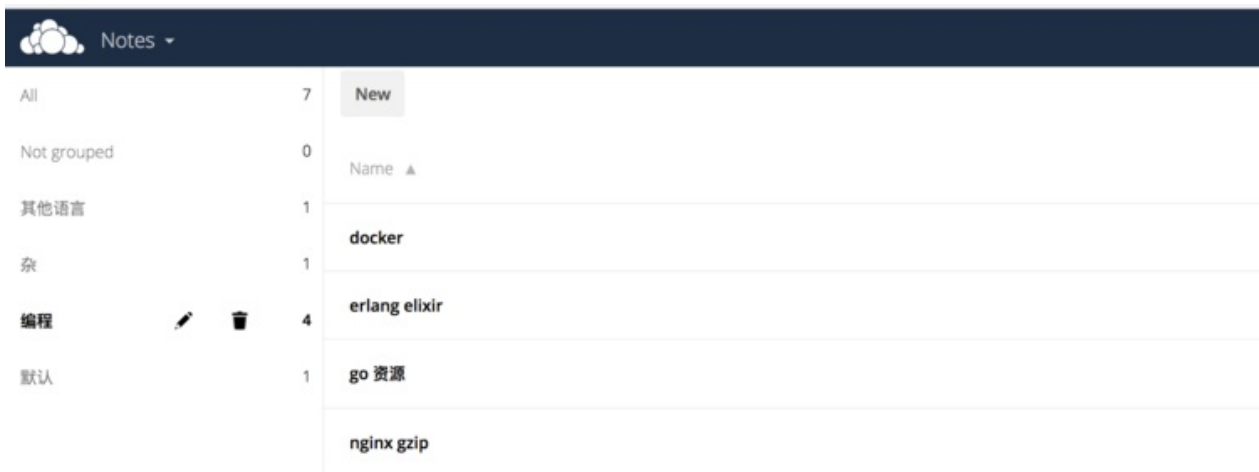
[owncloud](#)是一款放文件和图片等数据的应用，它有点类似于网盘这样的东西，以后再也不用担心哪个网盘被封杀了，自己搭建一个，上传和下载速度又快，资料也安全，我最喜欢的是它有两个特点，第一个是有插件，比如说，我安装了一个可以写笔记的插件，还有另外一个，就是它有很多平台的客户端，在mac或windows，甚至智能手机下我可以安装一个软件，然后能够通过操作文件资源管理器那样随意同步文件。

下面是一些截图：

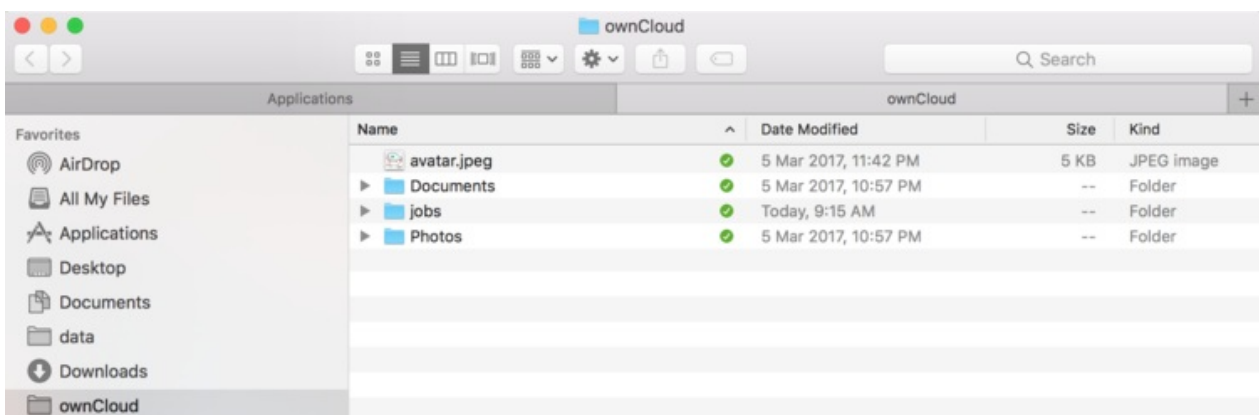
放文件的主页面：



放笔记的主页面：



mac下的客户端



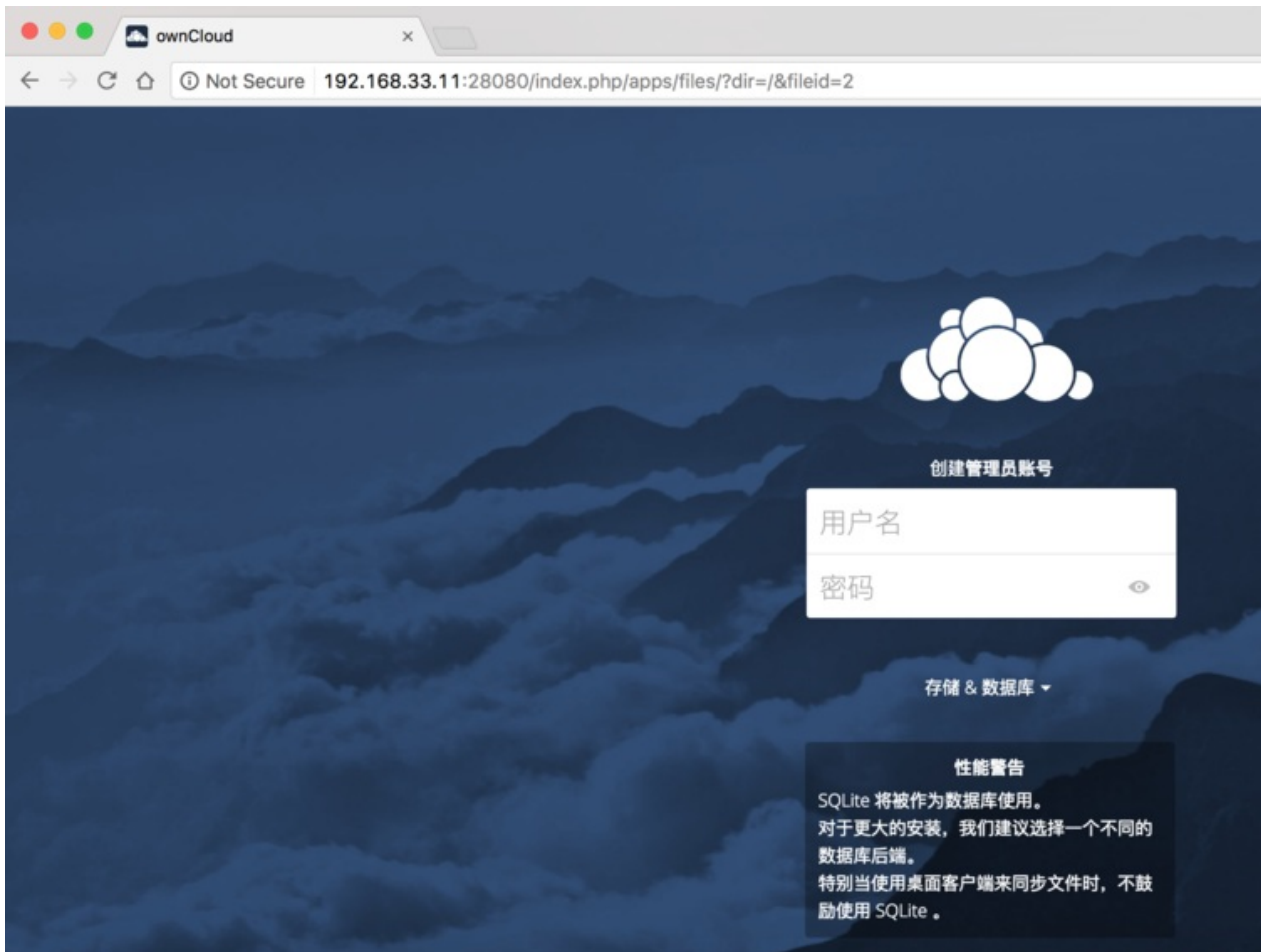
这节主要还是介绍owncloud的部署，之所以会介绍phpMyAdmin的部署，是为了去研究docker的其他知识点，也能轻易地用phpMyAdmin去连接已经存在的数据库，不管是不是docker部署的。

## 2. 部署owncloud

最简单的部署可能是这样的：

```
$ docker run -d -p 28080:80 owncloud
```

然后访问这个应用，界面如下：



不过这样并没有多大的用处，因为这样只能使用sqlite作为数据库，其实它可以使用mysql或postgresql作为数据库的。

我们把上一篇的知识点结合起来，最终用docker-compose来部署，写了一个部署文件：

```
version: '2'

services:

  owncloud:
    restart: always
    image: owncloud
    ports:
      - 18080:80
    volumes:
      - /home/hfpp2012/owncloud/html:/var/www/html:Z
    depends_on:
      - db
    dns:
      - 10.202.72.118
      - 10.202.72.116
      - 8.8.8.8

  db:
    restart: always
    image: mariadb
    environment:
      - MYSQL_ROOT_PASSWORD=my-secret-pw
      - MYSQL_DATABASE=owncloud_pro
    volumes:
      - /home/hfpp2012/owncloud/datadir:/var/lib/mysql
```

`volumes` 是挂载数据卷，这个上篇文章也说过，至于`dns`是为了让owncloud装插件的时候不报超时的错误。

重点来看的是 `depends_on`，这个跟上篇文章所提的link是对应的。

整个意思是说这样的：这个 `compose` 文件包含两个服务，主要是owncloud这个服务，它又依赖于下面的db服务。也就是说这个owncloud服务可以去连接这个db数据库服务。因为这个owncloud服务是需要连接数据库的，必须要有一个数据库服务，要么是外部的，要么是内部，这里用docker开启了db服务，供owncloud来连接。

运行 `docker-compose up` 把这个服务跑起来。

注意，测试的时候，建议启动前，把两个镜像卷的内容清一下

```
$ rm -rf /home/hfpp2012/owncloud/html  
$ rm -rf /home/hfpp2012/owncloud/datadir
```

启动起来后，不要再选择默认的sqlite数据库，我们要选择mysql数据库，如下所示，填上必要的信息：



The image shows the 'Create administrator account' step of the OwnCloud installation. It features a yellow form for the username 'gitlab' and a password field with a strength indicator. A tooltip points to the password field, stating '非常弱的密码' (Very weak password). Below this is a 'Storage & Database' section with a dropdown menu set to '数据目录' (Data directory), showing the path '/var/www/html/data'. Under '配置数据库' (Configure database), 'MySQL/MariaDB' is selected among 'SQLite' and 'PostgreSQL'. The database configuration fields show 'root' as the user, a masked password, 'owncloud\_pro' as the database name, and 'db' as the table prefix. A large blue button at the bottom says '安装完成' (Installation complete).

创建管理员账号

gitlab

.....

非常弱的密码

存储 & 数据库 ▾

数据目录

/var/www/html/data

配置数据库

SQLite MySQL/MariaDB PostgreSQL

root

.....

owncloud\_pro

db

安装完成

至于最上面的管理员账号，你自己指定账号名和密码就可以了。

下面的连接数据库的配置要参照 `docker-compose.yml` 文件中的 `db` 部分，用户名是 `root`，密码是 `my-secret-pw`，数据库是 `owncloud_pro`，最重要的是数据库主机这部分，这部分应该填写 `ip` 地址或域名的，不过这里只要填写 `db` 就可以了，它是一个别名，因为配置中 `owncloud` 中有一个项是 `depends_on`，它代表能够连接到 `db` 那个容器上。

这样点击 `创建完成` 就能连接好数据库，整个应用也能跑起来，如下图所示：



### 3. 换别的数据库postgresql

owncloud能正常跑起来，也能正常使用了，不过这里要介绍一下，如果不用mysql数据库呢，我要用postgresql，也是可以的。

把compose配置文件中的 `db` 那部分换一下，如下：



```
version: '2'

services:

  owncloud:
    restart: always
    image: owncloud
    ports:
      - 18080:80
    volumes:
      - /home/ubuntu/owncloud/html:/var/www/html:Z
    depends_on:
      - postgresql
    dns:
      - 8.8.8.8
      - 10.0.2.3

  postgresql:
    restart: always
    image: sameersbn/postgresql:9.5-4
    volumes:
      - /home/ubuntu/owncloud/postgresql:/var/lib/postgresql:Z
    environment:
      - DB_USER=gitlab
      - DB_PASS=password
      - DB_NAME=gitlabhq_production
```

注意，此时，在owncloud安装页面中的 数据库主机 就要填成 postgresql 了。如下图所示：



The image shows the OwnCloud installation wizard interface. At the top, the title '创建管理员账号' (Create administrator account) is displayed. Below it, a yellow box contains the username 'gitlab' and a password field with dots and an eye icon. A red progress bar is at the bottom of this box. A black tooltip with the text '非常弱的密码' (Very weak password) points to the password field. Below the password field, the text '存储 & 数据库' (Storage & Database) is followed by a dropdown arrow. Underneath, the title '数据目录' (Data directory) is shown above a white box containing the path '/var/www/html/data'. Below this, the title '配置数据库' (Configure database) is shown above three tabs: 'SQLite', 'MySQL/MariaDB', and 'PostgreSQL'. The 'PostgreSQL' tab is selected. Below the tabs, a white box contains the database name 'gitlab', a password field with dots and an eye icon, the database name 'gitlabhq\_production', and the database type 'postgresql'. At the bottom, a large blue button with the text '安装完成' (Installation complete) is visible.

创建管理员账号

gitlab

.....

非常弱的密码

存储 & 数据库 ▾

数据目录

/var/www/html/data

配置数据库

SQLite MySQL/MariaDB PostgreSQL

gitlab

.....

gitlabhq\_production

postgresql

安装完成

按照之前的步骤，最后也是能正常跑起owncloud的。

## 4. 使用 **phpmyadmin** 来连接 **mysql** 数据库

首先，我们不来连接我们之前建立的mysql服务，而是先新建一个mysql服务的容器，然后来连接。

### 4.1 先跑起来

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql
```

然后使用新建一个连接此mysql容器的phpmyadmin的容器：

```
$ docker run --name myadmin -d --link some-mysql:db -p 8080:80 phpmyadmin/phpmyadmin
```

打开8080端口，界面如下图所示：



The image shows the phpMyAdmin login page. At the top is the phpMyAdmin logo, which features a stylized sailboat and the text 'phpMyAdmin'. Below the logo is the text '欢迎使用 phpMyAdmin'. There are two main sections: '语言 - Language' and '登录'. The '语言 - Language' section has a dropdown menu showing '中文 - Chinese simplified'. The '登录' section has fields for '用户名:' (Username) with the value 'root' and '密码:' (Password) with a masked password '.....'. There is a '执行' (Execute) button at the bottom right.

不用填写主机，因为已经用了连接器(--link)db那个别名，用了这个别名，就会导入一些变量给myadmin这个实例，而这些变量的值，又是由 some-mysql 这个容器传过来的，例如 some-mysql 这个容器的ip地址，端口等。关于这部分的详细内容可以查看上一篇文章。

你可以试下把 db 那个名称改一下，换成别的，你会发现会用不了的，因为 myadmin 这个容器已经找不到对应的 some-mysql 容器的ip地址了。

#### 4.2 使用--net参数

使用phpmyadmin的重点不是去连接一个自己的新建的mysql服务器，这样没有意义，而是要连接我们目前存在于系统上的，且是用docker-compose部署的。

现在我们就来连接之前使用docker-compose部署owncloud时创建的mysql容器。

按照之前的经验，你可以会先找mysql容器的名称，然后把它link起来。

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
CREATED	STATUS	PORTS
NAMES		
a1a069dfd65c	owncloud	"/entrypoint.sh ap..."
18 seconds ago	Up 17 seconds	0.0.0.0:18080->80/tcp
owncloud_owncloud_1		
8ce43555ef6f	mariadb	"docker-entrypoint..."
18 seconds ago	Up 18 seconds	3306/tcp
owncloud_db_1		

可见，容器名称为 `owncloud_db_1` 。

于是我们把phpmyadmin容器启动起来。

```
$ b3f23a98d638cd2d3a416f4692a77f56541cff44bc9c54527e9df858c348eb
26
docker: Error response from daemon: Cannot link to /owncloud_db_
1, as it does not belong to the default network.
```

出错了，意思是说不能连接到owncloud\_db\_1，网络不对。

确实是，之前运行owncloud容器的时候，如果你有留意到的话，它会先创建一个网络：

```
Creating network "owncloud_default" with the default driver
Creating owncloud_db_1
Creating owncloud_owncloud_1
Attaching to owncloud_db_1, owncloud_owncloud_1
...
```

我们使用 `docker network ls` 命令查看一下网络的情况。

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
80b102ef3785	bridge	bridge	local
925fbce49cb8	host	host	local
3cc70e86c6eb	none	null	local
29f2f9c181ea	owncloud_default	bridge	local
53afc10ba142	root_default	bridge	local

有一个适配器是由owncloud创建的，名为 `owncloud_default`。我们把它利用起来。

我们在运行phpmyadmin容器的时候，加入一个参数 `--net`。

```
$ docker run --name myadmin -d --link owncloud_db_1:db -p 8080:80 --net owncloud_default phpmyadmin/phpmyadmin
```

现在就可以运行起来了，记住运行前，要把以前的myadmin容器进程stop和kill掉。

### 4.3 使用docker-compose部署

这样用命令来敲，我不太喜欢，因为难记，我喜欢把它写到一个docker-compose.yml文件中，然后一条命令就可以跑起来。

```
version: '2'

services:

  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    container_name: phpmyadmin
    network_mode: "owncloud_default"
    environment:
      - PMA_ARBITRARY=1
    restart: always
    ports:
      - 8080:80
    volumes:
      - /sessions
```

在compose配置文件中使用 `network_mode` 代替 `--net` 参数。

`PMA_ARBITRARY=1` 这个是phpmyadmin使用的变量，意思就是说，可以自己输入ip地址。如下图所示：



The image shows the phpMyAdmin login page. At the top is the phpMyAdmin logo, which consists of a stylized sailboat above the text 'phpMyAdmin'. Below the logo is the text '欢迎使用 phpMyAdmin'. There are two main sections: '语言 - Language' and '登录'. The '语言 - Language' section has a dropdown menu currently set to '中文 - Chinese simplified'. The '登录' section has three input fields: '服务器:' with the value 'db', '用户名:' with the value 'root', and '密码:' with a masked password '.....'. A '执行' button is located at the bottom right of the login section.

服务器 那里要输入 db 。

现在可以成功地用docker来运行phpmyadmin了，并且可以连接到自己需要的数据库进行查看。

至于部署到线上环境，可以参考下面这两篇文章，使用nginx反向代理过去。

- [使用compose部署gitlab应用\(八\)](#)
- [使用compose部署Rocket.Chat应用\(九\)](#)

完结。

下一篇：[让 php-fpm 跑的 owncloud 应用 docker 化 \(十二\)](#)





### 1. 介绍

之前我们介绍过owncloud的部署([部署 owncloud 与 phpMyAdmin \(十一\)](#))，不管是owncloud还是phpMyAdmin都是php语言写的应用，owncloud这个容器默认是跑在apache下，我不太喜欢，想改成nginx+php-fpm来跑。

### 2. 一个简单的实例

我们先来尝试一下用nginx结合php-fpm来跑php应用。

首先创建一个docker-compose.yml文件，内容如下：

```
version: '2'

services:
  web:
    image: nginx:latest
    ports:
      - 8080:80
    volumes:
      - ./code:/code
      - ./site.conf:/etc/nginx/conf.d/default.conf
  php:
    image: php:fpm
    volumes:
      - ./code:/code
```

数据卷 `./code` 这个目录是放php代码的，还有一个文件 `site.conf` 放的是nginx的配置。

它的内容如下：

```
server {
    index index.php index.html;
    server_name localhost;
    error_log /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root /code;

    location ~ /\.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass php:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
}
```

nginx中使用fastcgi这个模块去连接php-fpm。这些是nginx中fastcgi的基本配置啦。

然后我们试着写些php的代码，让它跑起来：

在当前目录下的code目录下新建 index.php 文件，内容如下:

```
<?php
echo phpinfo();
?>
```

然后一行 `docker-compose up` 命令就可以跑起来，使用8080端口查看效果。



# PHP Version 5.6.30

System	Linux 0334a9997ab4 4.4.0-66-generic #87-Ubuntu SMP Fri M
Build Date	Feb 28 2017 17:36:23
Configure Command	'./configure' '--with-config-file-path=/usr/local/etc/php' '--with-c cgi' '--enable-ftp' '--enable-mbstring' '--enable-mysqlnd' '--with enable-fpm' '--with-fpm-user=www-data' '--with-fpm-group=ww fpie' '-O2' 'LDLFLAGS=-Wl,-O1 -Wl,--hash-style=both' '-pie' 'C
Server API	FPM/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/etc/php
Loaded Configuration File	(none)
Scan this dir for additional .ini files	/usr/local/etc/php/conf.d
Additional .ini files parsed	(none)
PHP API	20131106

### 3. 从apache迁移到php-fpm

现在尝试把owncloud用php-fpm来跑。

之前我们的docker-compose.yml是这么写的：

```
version: '2'

services:

  owncloud:
    restart: always
    image: owncloud
    ports:
      - 18080:80
    volumes:
      - /home/hfpp2012/owncloud/html:/var/www/html:Z
    depends_on:
      - db
    dns:
      - 10.202.72.118
      - 10.202.72.116
      - 8.8.8.8

  db:
    restart: always
    image: mariadb
    environment:
      - MYSQL_ROOT_PASSWORD=my-secret-pw
      - MYSQL_DATABASE=owncloud_pro
    volumes:
      - /home/hfpp2012/owncloud/datadir:/var/lib/mysql
```

需要改造一下，把 owncloud 中的 image 和 ports 部分变一下：

```
image: owncloud:9.1.4-fpm
ports:
  - 9000:9000
```

只是给owncloud换个版本，而且php-fpm默认是使用9000端口的。

使用 `docker-compose up` 或 `docker-compose restart` 重新把owncloud这个应用跑起来。

## 4. nginx配置

接下来，我们把nginx的配置加上。

我们不用上面的那个很普通的关于fastcgi的配置，而owncloud的官方提供了一套比较标准的。

官方提供的那套nginx配置的网址

是[https://doc.owncloud.org/server/9.0/admin\\_manual/installation/nginx\\_examples.html](https://doc.owncloud.org/server/9.0/admin_manual/installation/nginx_examples.html)

```
upstream php-handler {
    server 127.0.0.1:9000;
    #server unix:/var/run/php5-fpm.sock;
}

server {
    listen 80;
    server_name file.rails365.net;
    # enforce https
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl;
    server_name file.rails365.net;

    ssl_certificate          /home/hfpp2012/owncloud/ssl/file.r
ails365.net.key.pem;
    ssl_certificate_key      /home/hfpp2012/owncloud/ssl/file.r
ails365.net.key;
    # ssl_dhparam
    ssl_dhparam /home/hfpp2012/owncloud/ssl/dhparam.pem;

    # Add headers to serve security related headers
    # Before enabling Strict-Transport-Security headers please r
ead into this topic first.
    #add_header Strict-Transport-Security "max-age=15552000; inc
ludeSubDomains";
    add_header X-Content-Type-Options nosniff;
```

```
add_header X-Frame-Options "SAMEORIGIN";
add_header X-XSS-Protection "1; mode=block";
add_header X-Robots-Tag none;
add_header X-Download-Options noopen;
add_header X-Permitted-Cross-Domain-Policies none;

# Path to the root of your installation
root /home/hfpp2012/owncloud/html;

location = /robots.txt {
    allow all;
    log_not_found off;
    access_log off;
}

# The following 2 rules are only needed for the user_webfinger
# app.
# Uncomment it if you're planning to use this app.
#rewrite ^/.well-known/host-meta /public.php?service=host-meta last;
#rewrite ^/.well-known/host-meta.json /public.php?service=host-meta-json last;

location = /.well-known/carddav {
    return 301 $scheme://$host/remote.php/dav;
}
location = /.well-known/caldav {
    return 301 $scheme://$host/remote.php/dav;
}

location /.well-known/acme-challenge { }

# set max upload size
client_max_body_size 512M;
fastcgi_buffers 64 4K;

# Disable gzip to avoid the removal of the ETag header
gzip off;

# Uncomment if your server is build with the ngx_pagespeed m
```

```

module
    # This module is currently not supported.
    #pagespeed off;

    error_page 403 /core/templates/403.php;
    error_page 404 /core/templates/404.php;

    location / {
        rewrite ^ /index.php$uri;
    }

    location ~ ^/(?:(?:build|tests|config|lib|3rdparty|templates|data)/ {
        return 404;
    }
    location ~ ^/(?!(?:\.(?:autotest|occ|issue|indie|db_|console)) {
        return 404;
    }

    location ~ ^/(?!(?:index|remote|public|cron|core/ajax/update|status|ocs/v[12]|updater/.+|ocs-provider/.+|core/templates/40[34])\.(?:php|?)$|/) {
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME /var/www/html$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
        fastcgi_param HTTPS on;
        fastcgi_param modHeadersAvailable true; #Avoid sending the security headers twice
        fastcgi_param front_controller_active true;
        fastcgi_pass php-handler;
        fastcgi_intercept_errors on;
        fastcgi_request_buffering off; #Available since nginx 1.7.11
    }

    location ~ ^/(?!(?:updater|ocs-provider)(?:$|/)) {
        try_files $uri $uri/ =404;
        index index.php;
    }

```



```

}

# Adding the cache control header for js and css files
# Make sure it is BELOW the PHP block
location ~* \.(?:css|js)$ {
    try_files $uri /index.php$uri$is_args$args;
    add_header Cache-Control "public, max-age=7200";
    # Add headers to serve security related headers (It is i
ntended to have those duplicated to the ones above)
    # Before enabling Strict-Transport-Security headers plea
se read into this topic first.
    #add_header Strict-Transport-Security "max-age=15552000;
includeSubDomains";
    add_header X-Content-Type-Options nosniff;
    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-XSS-Protection "1; mode=block";
    add_header X-Robots-Tag none;
    add_header X-Download-Options noopen;
    add_header X-Permitted-Cross-Domain-Policies none;
    # Optional: Don't log access to assets
    access_log off;
}

location ~* \.(?:svg|gif|png|html|ttf|woff|ico|jpg|jpeg)$ {
    try_files $uri /index.php$uri$is_args$args;
    # Optional: Don't log access to other assets
    access_log off;
}
}

```

在官方提供的配置的基础上，我做出了一些改变，除了证书和域名部分是一定要跟你的实际情况一致之外，还有如下改变：

- `root` 这一行变成 `root /home/hfpp2012/owncloud/html;`
- `fastcgi_param SCRIPT_FILENAME`  
`$document_root$fastcgi_script_name;` 变成了 `fastcgi_param`  
`SCRIPT_FILENAME /var/www/html$fastcgi_script_name;`

最后一个改变是为了解决一个nginx的报错问题而变的。我参考这篇文章进行了解决：

<http://lovelace.blog.51cto.com/1028430/1314565>

除此之外，有一个地方值得注意，如果不使用https的话，除了改变433端口为80端口一些必要的改变，还需要把下面一行注释掉：

```
fastcgi_param HTTPS on;
```

完结。

## 1. 介绍

现在我需要把docker部署的gitlab应用迁移到另一台主机上。

如果不知道如何用**docker**来搭建**gitlab**服务的，可以参照我以前的一篇文章[使用 compose 部署 GitLab 应用 \(八\)](#)

## 2. 流程

首先gitlab是会每天做一次备份的，备份文件位于 `/srv/docker/gitlab/gitlab/backups` 。

```
hfpp2012@iz94dujl81mZ:/srv/docker/gitlab/gitlab/backups$ pwd
/srv/docker/gitlab/gitlab/backups
hfpp2012@iz94dujl81mZ:/srv/docker/gitlab/gitlab/backups$ ls
1492045227_2017_04_13_gitlab_backup.tar  1492218048_2017_04_15_gitlab_backup.tar  1492477226_2017_04_18_gitlab_backup.tar
1492131651_2017_04_14_gitlab_backup.tar  1492304447_2017_04_16_gitlab_backup.tar  1492563646_2017_04_19_gitlab_backup copy.tar
1492131655_2017_04_14_gitlab_backup.tar  1492390847_2017_04_17_gitlab_backup.tar  1492563646_2017_04_19_gitlab_backup.tar
```

这里有好多个备份，选一个日期最新的。

再把它上传到服务器上。

如果觉得都不新，可以创建一个最新的，使用下面的命令：

```
$ docker run --name gitlab -it --rm \
  sameersbn/gitlab:9.0.5 app:rake gitlab:backup:create
```

不管怎样，就是要找一个备份，然后上传到新的服务器上。

注意：这个备份所使用**gitlab**的版本和新的服务器上使用的**gitlab**的版本要一致，不然不能成功迁移的

新的服务器的备份的文件存放的位置跟之前的一样，也

是： `/srv/docker/gitlab/gitlab/backups` 。

执行下面的命令可以恢复备份：

```
$ docker-compose run --rm gitlab app:rake gitlab:backup:restore
```

```

[root@iZ94xgvnfq8Z:~/gitlab-compose-nginx# docker-compose run --rm gitlab app:rake gitlab:backup:restore
Initializing logdir...
Initializing datadir...
Installing configuration templates...
Configuring gitlab...
Configuring gitlab::database
Configuring gitlab::redis
Configuring gitlab::secrets...
Configuring gitlab::sidekiq...
Configuring gitlab::gitlab-workhorse...
Configuring gitlab::unicorn...
Configuring gitlab::timezone...
Configuring gitlab::rack_attack...
Configuring gitlab::ci...
Configuring gitlab::artifacts...
Configuring gitlab::lfs...
Configuring gitlab::mattermost...
Configuring gitlab::project_features...
Configuring gitlab::smtp_settings...
Configuring gitlab::oauth...
Configuring gitlab::oauth::github...
Configuring gitlab::ldap...
Configuring gitlab::backups...
Configuring gitlab::backups::schedule...
Configuring gitlab::registry...
Configuring gitlab::pages...
Configuring gitlab-shell...
Configuring nginx...
Configuring nginx::gitlab...

date: invalid date '@1492563646_2017_04_19'
▶ 1492563646_2017_04_19_gitlab_backup.tar (created at )

Select a backup to restore: 1492563646_2017_04_19_gitlab_backup.tar

```

之后可能会报错：

```

Unpacking backup ... done
Before restoring the database we recommend removing all existing
tables to avoid future upgrade problems. Be aware that if you have
custom tables in the GitLab database these tables and all data will be
removed.

Do you want to continue (yes/no)? yes
Removing all tables. Press `Ctrl-C` within 5 seconds to abort
Cleaning the database ...
rake aborted!
ActiveRecord::StatementInvalid: PG::UndefinedTable: ERROR: relation "schema_migrations" does not exist
: TRUNCATE schema_migrations
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/post
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/post
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/abst
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/abst
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/abst
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/post
/home/git/gitlab/lib/tasks/gitlab/db.rake:35:in `block (3 levels) in <top (required)>'
/home/git/gitlab/lib/tasks/gitlab/backup.rake:51:in `block (3 levels) in <top (required)>'
PG::UndefinedTable: ERROR: relation "schema_migrations" does not exist
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/post
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/post
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/abst
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/abst
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/abst
/home/git/gitlab/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.8/lib/active_record/connection_adapters/post
/home/git/gitlab/lib/tasks/gitlab/db.rake:35:in `block (3 levels) in <top (required)>'
/home/git/gitlab/lib/tasks/gitlab/backup.rake:51:in `block (3 levels) in <top (required)>'
Tasks: TOP => gitlab:db:drop_tables
(See full trace by running task with --trace)

```

没关系，你再把 `docker-compose up` 运行一下，再开一个终端去运行之前的恢复命令。


```
Restoring repositories ...
* hfpp2012/boat_manager ... [DONE]
* hfpp2012/store_data ... [DONE]
* hfpp2012/pusheng ... [DONE]
* hfpp2012/new_rails_base_app ... [DONE]
* hfpp2012/nginx-docker-conf ... [DONE]
* hfpp2012/old_docker_compose ... [DONE]
* hfpp2012/gitlab-compose-nginx ... [DONE]
* hfpp2012/h4_template ... [DONE]
* hfpp2012/rails365 ... [DONE]
* bc/portal ... [DONE]
* hfpp2012/idso2017 ... [DONE]
* hfpp2012/QOLM-P-app ... [DONE]
* hfpp2012/docker-for-discuz ... [DONE]
* hfpp2012/koa-app-start ... [DONE]
* hfpp2012/tinyshop-docker ... [DONE]
* hfpp2012/vbot ... [DONE]
* hfpp2012/accounts-react ... [DONE]
* hfpp2012/es6-react-webpack ... [DONE]
* hfpp2012/docker-flarum ... [DONE]
* hfpp2012/yarn_project ... [DONE]
* hfpp2012/ownnote ... [DONE]
* hfpp2012/powerpaste ... [DONE]
* hfpp2012/cttp_16_x_lbd ... [DONE]
* hfpp2012/cttp_9_dashboard-free ... [DONE]
* hfpp2012/cttp_6_dashboard ... [DONE]
* hfpp2012/dgfp_42_hr ... [DONE]
* hfpp2012/p2p ... [DONE]
* hfpp2012/1477875142_p2p ... [DONE]
* hfpp2012/inxedu ... [DONE]
* hfpp2012/diming ... [DONE]
* hfpp2012/vote ... [DONE]
Put GitLab hooks in repositories dirs [DONE]
done
```

数据库和仓库都会帮我恢复

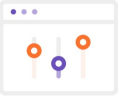
所有数据都回来了，good job!



## 13. docker 迁移 GitLab 项目

 Projects

Search



**Customize your experience**  
Change syntax themes, default project pages, and more in preferences.  
[Check it out](#)

**Your projects**

Starred projects

Explore projects

Filter by name...

Last updated

New Project

S	hfpp2012 / store_data	★ 0	🔒
R	hfpp2012 / rails365 墨剑定制	★ 0	🔒
N	hfpp2012 / nginx-docker-conf	★ 0	🔒
V	hfpp2012 / vote 答题网站	★ 0	🔒
D	hfpp2012 / docker-flarum	★ 0	🔄

完结。