**深度探索Docker容器化**

**Inside the Docker Containerization**

Version： 1.0.0

Author : Jxva

**Change Records**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| File No | Version | Author/Modifier | Created/Modifed | Reason for change | Change history |
| 1 | 1.0.0 | Jxva | 2015-01-14 |  | Draft |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Note 1:  You change the archive must fill in the table every time.  Note 2:  Both "reason for change" and "change content" need to fill in each archive. | | | | | |

**Table of Contents**

[1 Introduction 6](#_Toc421018961)

[1.1 What is Docker? 6](#_Toc421018962)

[1.2 The Docker Ecosysem 6](#_Toc421018963)

[1.3 The Implementation of Docker 7](#_Toc421018964)

[1.3.1 Namespaces 7](#_Toc421018965)

[1.3.2 Cgroups 8](#_Toc421018966)

[1.3.3 LXC 9](#_Toc421018967)

[1.3.4 AUFS 10](#_Toc421018968)

[1.3.5 BTRFS 11](#_Toc421018969)

[1.4 Containers vs VMs 11](#_Toc421018970)

[1.4.1 Virtual Machines 11](#_Toc421018971)

[1.4.2 Docker Container 12](#_Toc421018972)

[1.4.3 Similarities and Differences 12](#_Toc421018973)

[1.5 Future Development 12](#_Toc421018974)

[2 Getting Started 14](#_Toc421018975)

[2.1 Upgrade Kernel 14](#_Toc421018976)

[2.2 Create User for Docker 14](#_Toc421018977)

[2.3 Install Docker 15](#_Toc421018978)

[2.4 Install Basic Dependencies 15](#_Toc421018979)

[2.4.1 Install LXC 15](#_Toc421018980)

[2.4.2 Install Cgroup 16](#_Toc421018981)

[2.5 The Usage of Docker 16](#_Toc421018982)

[2.5.1 Command Line Help 18](#_Toc421018983)

[2.5.2 Background Process 20](#_Toc421018984)

[2.6 Quick Start 21](#_Toc421018985)

[2.6.1 Download the Image File 21](#_Toc421018986)

[2.6.2 Import the Image File 22](#_Toc421018987)

[2.7 Install OpenSSH 22](#_Toc421018988)

[2.8 Install Supervisord 23](#_Toc421018989)

[2.8.1 Configure Supervisord 23](#_Toc421018990)

[2.8.2 Configure Servers 24](#_Toc421018991)

[2.9 Image Persistence and Restore 24](#_Toc421018992)

[2.9.1 Export Image 24](#_Toc421018993)

[2.9.2 Import Image 25](#_Toc421018994)

[2.9.3 Save Image 25](#_Toc421018995)

[2.9.4 Load Image 25](#_Toc421018996)

[3 Docker Image Management 26](#_Toc421018997)

[3.1 Made by Oneself Docker Image 26](#_Toc421018998)

[3.1.1 Make Image in CentOS6 26](#_Toc421018999)

[3.1.2 Make Image in CentOS7 27](#_Toc421019000)

[3.2 Use Docker in Ali Cloud 28](#_Toc421019001)

[3.2.1 Remove Default Route 28](#_Toc421019002)

[3.2.2 Create a Net Bridge 28](#_Toc421019003)

[3.2.3 Bind a CIDR 28](#_Toc421019004)

[3.3 Daemon Service 28](#_Toc421019005)

[3.4 The Use of Docker in Practice 29](#_Toc421019006)

[3.4.1 Running Practice 29](#_Toc421019007)

[3.4.2 The Commonly Used Skill 32](#_Toc421019008)

[3.5 Dockerfile 32](#_Toc421019009)

[4 Docker Hub 33](#_Toc421019010)

[5 Docker Registry 34](#_Toc421019011)

[5.1 Why do you need a Docker Registry? 34](#_Toc421019012)

[5.2 Set Up Our Own Registry Service 35](#_Toc421019013)

[5.3 Deploy and Configure the Registry (for registry v1.0) 35](#_Toc421019014)

[5.3.1 Use registry to start private warehouse container 35](#_Toc421019015)

[5.3.2 Push mirror to private warehouse 35](#_Toc421019016)

[5.3.3 Private warehouse query method 38](#_Toc421019017)

[5.3.4 Access and download the mirror of the private warehouse on the other machines 38](#_Toc421019018)

[5.4 Run a local registry mirror 38](#_Toc421019019)

[5.5 Why? 38](#_Toc421019020)

[5.6 How does it work? 38](#_Toc421019021)

[5.7 How do I set up a local registry mirror? 39](#_Toc421019022)

[5.7.1 Step 1: Configure your Docker daemons to use the local registry mirror 39](#_Toc421019023)

[5.7.2 Step 2: Run the local registry mirror 39](#_Toc421019024)

[5.8 Test it out 39](#_Toc421019025)

[6 Docker API 41](#_Toc421019026)

[7 Docker Advanced Usage 42](#_Toc421019027)

[7.1 Pull Docker via Proxy 42](#_Toc421019028)

[7.2 Initialize the Hosts File for Docker Containers 42](#_Toc421019029)

[7.3 Bind a Static IP for the Docker Containers 43](#_Toc421019030)

[7.4 Mapping Ports for Docker Containers 43](#_Toc421019031)

[7.5 Customizing Permissions for Docker Containers 43](#_Toc421019032)

[8 What is Next? 44](#_Toc421019033)

[9 Reference Documentation 45](#_Toc421019034)

# https://d3oypxn00j2a10.cloudfront.net/0.13.0/img/homepage/docker-whale-home-logo.png?76ac00d5f24bIntroduction

Docker是PaaS提供商dotCloud开源的一个基于 LXC 的高级容器引擎，源代码托管在 Github上, 基于[Go语言](http://baike.baidu.com/view/2976233.htm)并遵从Apache2.0协议开源。

Docker是一个开源的应用容器引擎，让开发者可以将他们的应用和依赖包打包到一个可移植的容器中，然后发布到任何流行的Linux机器上，也可以实现虚拟化。容器完全使用沙箱机制，相互之间不会有任何接口。几乎没有性能开销，可以很容易地在机器和数据中心中运行。最重要的是，它不依赖于任何语言、框架或包装系统。

Docker的官方网址为：<http://www.docker.com/>

## What is Docker?

Build, Ship and RunAny App, Anywhere!

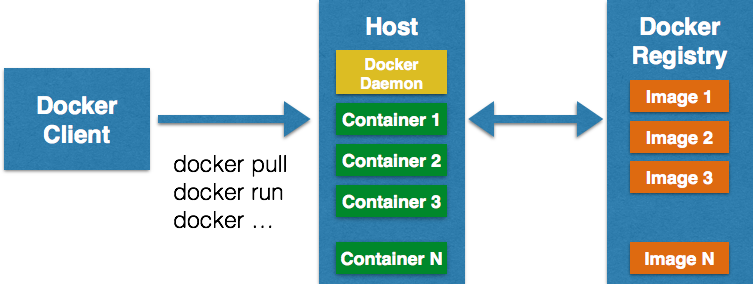
Docker - An open platform for distributed applications for developers and sysadmins.

Docker Engine - A portable, lightweight application runtime and packaging tool.

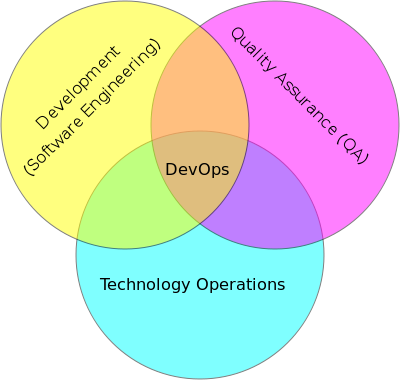
Docker Hub - A cloud service for sharing applications and automating workflows.

## The Docker Ecosysem

Docker 主要有 Docker Hub 和 Docker 引擎组成。前者是Docker 官方提供的容器镜像仓库；后者运行在宿主机上，可分为服务器端和客户端两部分。服务器端负责构建、运行和分发 Docker 容器等重要工作，客户端负责接收用户的命令和服务程序进行通信。



除了这两部分，Dockerfile也是不得不提的，它虽然不能算作一个独立的组件，但是却是 Docker 中很重要的部分。通过 Dockerfile，技术人员可以创建自己的Docker容器镜像。Dockerfile起到了连接开发与运维的桥梁的作用，非常符合现在DevOps(开发软件工程、技术运营和质量保障QA三者的交集)的潮流。



## The Implementation of Docker

Docker 所使用的几个重要技术：namespaces、cgroups、LXC 、AUFS、BTRFS。

### Namespaces

*注：本小节内容参考：*[*http://blog.dotcloud.com/under-the-hood-linux-kernels-on-dotcloud-part*](http://blog.dotcloud.com/under-the-hood-linux-kernels-on-dotcloud-part)

Linux 容器通过 Kernel 的 namespaces 技术，为一个或一组进程创建独立的pid、net、ipc、mnt、uts 等 namespaces，从而与其它进程相互隔离。下面将介绍 namespaces 都会对哪些资源进行分组控制以实现相互隔离：

1. **pid namespace**

不同容器中进程是通过 pid namespace 隔离开的，且不同容器中可以有相同 pid，具有以下特征:

* 每个 namespace 中的 pid 是有自己的 pid=1 的初始进程
* 每个 namespace 中的进程只能影响自己的同一个 namespace 或子 namespace 中的进程
* 因为 /proc 包含正在运行的进程，因此在容器中的 /proc 目录只能看到自己 namespace 中的进程
* 因为 namespace 允许嵌套，父 namespace 可以影响子 namespace 的进程，所以子 namespace 的进程可以在父 namespace 中看到，但是具有不同的 pid

1. **net namespace**

如果仅仅隔离了进程空间，还是会有问题。比如如果你在多个容器中运行 Apache 服务器，那只能有一个服务器使用 80 端口。对此你有两个选择，让每个 Apache 服务器使用不同的端口，或者隔离网络空间。

net namespace 使得每个容器都有自己的 lo loopback 接口。同时，还有一个通常被命名为 eth0 的网络接口，通过这个接口，容器可以和 host 或其它容器进行通信。eth0 interface 会被分配一个 172.17.0.XXX 的 IP 地址，容器之间可以通过这个 IP 地址相互通信。

1. **ipc namespace**

ipc namepace 对于不熟悉 Unix 的人（包括之前的我）吸引力不是很大。毕竟，现在进程之间的通信多数是通过网络实现的。但实际上，Unix ipc 有着十分广泛的应用，比如管道就是 ipc 的一种。对 ipc 就不做过多介绍了，总之 Linux Kernel namespaces 可以让不同容器的 ipc 相互隔离。

1. **mnt namespace**

如名所示，mnt namespace 是处理挂载点的。mnt namespace 可以使不同容器拥有不同的挂载的文件系统和 root 目录。在一个 mnt namespace 挂载的文件系统只能被同一个 namespace 里的进程所见。

1. **uts namespace**

uts namespace 用于控制 hostname 的隔离。

有了以上几种隔离，一个容器就可以对外展现出一个独立计算机的能力，并且不同容器内的资源在操作系统层面实现了隔离。 然而不同 namespace 之间资源还是相互竞争的，仍然需要类似 ulimit 来管理每个容器所能使用的资源 - Docker 采用的是 cgroup。

### Cgroups

namespaces 对进程分组以实现资源隔离，但这隔离还是不够的。一个进程可以通过占用过度的硬件资源的方式去影响另一个分组中的进程。所以，要想实现完善的资源隔离，不仅要对资源分组，还要能对这一组内的进程所使用的资源进行约束。cgroups 就是用来实现这个目的的。

cgroups 的全称是 Control Groups，在 2003 年由 Google 的工程师实现，在 2007 年加入 Linux Kernel。cgroups 可限制进程对 CPU、内存、块存储和网络的使用。这里不对 cgroups 的使用方法作介绍，感兴趣的同学可以参考如下：

* <https://sysadmincasts.com/episodes/14-introduction-to-linux-control-groups-cgroups>
* <https://docs.docker.com/articles/runmetrics/>

虽然 cgroups 提供了对 IO 资源使用的约束的功能，但 Docker 目前(1.3)尚未提供支持。

1. **内存**

cgroups 可以控制进程所能使用的内存和 swap 空间的大小。通过 -m 参数，Docker 可以限制容器所能使用的最大内存数：

|  |
| --- |
| #docker run -m 128m -d container\_img cmd\_name |

|  |  |
| --- | --- |
|  |  |

1. **CPU**

Docker 可以通过 cgroups 限制容器所能使用的 CPU 资源。在执行 docker run 命令时可以通过参数 --cpu-shares (-c)、--cpuset 对 CPU 做出限制。

**--cpu-shares (-c)**

-c 设置的是一个相对值，这个值将影响到此容器内的进程所能使用的 CPU 时间片。新运行的 Docker 容器默认使用 1024，对一个单独的 Docker 的容器来说，这个值没有任何意义。当启动两个 Docker 容器的时候，这两个容器将平分 CPU 时间片。如果两个容器，一个不指定 -c，另一个设置为 -c 512，那前者将使用大致 2/3 的计算能力，另一个将使用大致 1/3 的计算能力。但是 -c 对容器所能使用 CPU 的运行频率等没有任何影响。

**--cpuset**

可让你指定 Docker 容器中的进程运行在第几块 CPU 上。后面跟一个数组或用逗号分隔的多个数字 0,1,2。比如下列命令将会使你的容器运行在第一个 CPU 核心上。

|  |
| --- |
| $ docker run -i -t --cpuset 0 ubuntu:14.04 |

|  |  |
| --- | --- |
|  |  |

*注：cgroups 使用的是一种伪文件系统形式的接口。这个伪文件系统实际存在于内存中，但映射在目录中。用户通过在这个目录中写入文件来对 cgroups 进行操作。*

### LXC

LXC为Linux Container的简写。Linux Container是一种轻量级的内核虚拟化的手段与技术，可以提供轻量级的虚拟化，以便隔离进程和资源，而且不需要提供指令解释机制以及全虚拟化的其他复杂性。

与传统[虚拟化技术](http://baike.baidu.com/view/13605.htm)相比，它的优势在于：

* 与宿[主机](http://baike.baidu.com/view/23880.htm)使用同一个内核，性能损耗小；
* 不需要指令级模拟；
* 不需要即时(Just-in-time)编译；
* 容器可以在CPU核心的本地[运行指令](http://baike.baidu.com/view/2290127.htm)，不需要任何专门的解释机制；
* 避免了准虚拟化和系统调用替换中的复杂性；
* 轻量级隔离，在隔离的同时还提供共享机制，以实现容器与宿主机的资源共享。

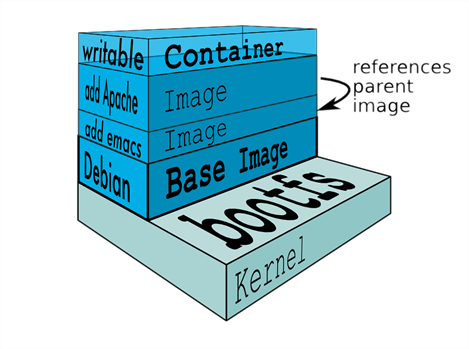
Linux Container提供了在单一可控[主机](http://baike.baidu.com/view/23880.htm)节点上支持多个相互隔离的server container同时执行的机制。Linux Container有点像chroot，提供了一个拥有自己进程和[网络空间](http://baike.baidu.com/view/420399.htm)的虚拟环境，但又有别于[虚拟机](http://baike.baidu.com/view/1132.htm)，因为LXC是一种操作系统层次上的资源的虚拟化。

LXC在资源管理方面依赖于Linux内核的cgroups子系统，cgroups子系统是Linux内核提供的一个基于进程组的资源管理的框架，可以为特定的进程组限定可以使用的资源。

*注：从0.9版本开始,Docker自身就具有一个管理容器的库,名字为 libcontainer。libcontainer中的nsinit工具允许用户直接访问linux名字空间和cgroup内核。*

### AUFS

*注：最新的 Docker 使用 BTRFS 替代 AUFS，但所要实现的功能相同*   
AUFS 的全称是 Another Union File System。AUFS（包括其它 UFS）的一个重要能力是能使两个目录结构合二为一。这有什么用呢？Docker 的镜像都是有多个层组成的，最上层是一个可读写的，而下面的层则是只读的。通过 AUFS 的目录融合的能力，实现了既可随意读写，又保证了下层的内容安全的目的。见下图可以有一个形象的认识。下图中的 bootfs 层包含了 Linux Kernel。在其上是某个特定的 Linux 发行版本的不同于 Kernel 的文件层，在下图中是 Debian。再往上有包含 emacs 和 Apache 的两个层。这些层在容器运行时都是只读的。最上面就是容器运行时可读写的层了。上面的层可以只读访问下面的层里的文件。这样的层次结构可以通过 Dockerfile 来创建。



那这样的一个结构有什么样的好处呢？主要有下面几点：

1. **节省磁盘和内存的存储空间**

节省磁盘存储空间是因为不同的容器镜像之间可以共享相同的层。例如，两个不同的 Java 应用的容器镜像，它们都是基于 Ubuntu 14.04 和 JDK7。那在同一台 Host 中，这两个镜像就会使用相同的 Ubuntu 14.04 和 JDK7 的层。

节省内存空间是因为 Linux 为了加快磁盘访问，会将一些磁盘上的文件加载到内存中。所以，节省磁盘空间的同时也就可以间接地解释内存的使用。

1. **加快部署速度**

同样，可共享的镜像层能加快部署速度。因为相同的层不用被重复下载部署。

1. **允许对文件任意改动**

上传可读写的层可以对下面的只读层中的文件做任意修改，但这其实是 copy-on-write，所以，这种修改对下面的层其实是安全的。

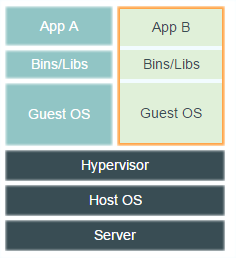
### BTRFS

Btrfs（通常念成Butter FS），由Oracle于2007年宣布并进行中的COW(copy-on-write式)文件系统。目标是取代Linux目前的ext3文件系统，改善ext3的限制，特别是单一文件大小的限制，总文件系统大小限制以及加入[文件校验](http://baike.baidu.com/view/5094912.htm)和特性。加入目前ext3/4未支持的一些功能，例如可写的[磁盘快照](http://baike.baidu.com/view/4190243.htm)(snapshots)，以及支持[递归](http://baike.baidu.com/view/96473.htm)的[快照](http://baike.baidu.com/view/16510.htm)(snapshots of snapshots)，内建[磁盘阵列](http://baike.baidu.com/view/63423.htm)（RAID）支持，支持子卷(Subvolumes)的概念，允许在线调整文件系统大小。

## Containers vs VMs

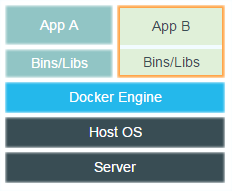
Docker Container和普通的虚拟机Image相比, 最大的区别是它并不包含操作系统内核。

### Virtual Machines



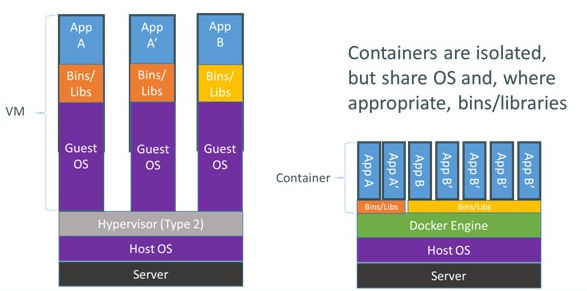
Virtual Machines - Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.

### Docker Container



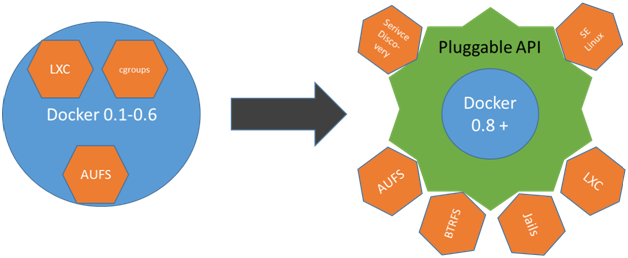
The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

### Similarities and Differences



## Future Development

虽然Docker目前默认使用LXC和AUFS，但是Docker的核心思想本身，并不强制绑定这两者，0.8版本已经可以使用BTRFS，而整个Docker框架也改成了插件式的架构，便于添加替换各个功能模块



 例如更多的Storage方案的支持，规避AUFS当前的问题，除了LXC以外更多的虚拟化方案等。

# Getting Started

Docker推荐的Linux内核为3.8及以上，在低于这个内核版本的系统上也可以运行，但是可能会有一些功能受限，笔者推荐大家使用CentOS7。

## Upgrade Kernel

对于内核低于3.8的CentOS6来说，笔者建议还是升级内核来完全支持Docker，内核的升级分为两种：

**(a). 在线升级**

|  |
| --- |
| $ rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org  $ rpm -ivh http://www.elrepo.org/elrepo-release-6-5.el6.elrepo.noarch.rpm  $ yum --enablerepo=elrepo-kernel install kernel-lt -y |

**(b). 离线升级**

可以从<http://jxva.com/down/rpm/upgrade_kernel_for_centos6> 上直接下载最新的内核进行安装，

|  |
| --- |
| $ rpm -Uvh http://jxva.com/down/rpm/upgrade\_kernel\_for\_centos6/kernel-lt-3.10.60-1.el6.elrepo.x86\_64.rpm  $ rpm -Uvh http://jxva.com/down/rpm/upgrade\_kernel\_for\_centos6/kernel-lt-devel-3.10.60-1.el6.elrepo.x86\_64.rpm  $ rpm -Uvh http://jxva.com/down/rpm/upgrade\_kernel\_for\_centos6/kernel-lt-firmware-3.10.60-1.el6.elrepo.noarch.rpm  $ rpm -Uvh http://jxva.com/down/rpm/upgrade\_kernel\_for\_centos6/kernel-lt-headers-3.10.60-1.el6.elrepo.x86\_64.rpm  $ rpm -Uvh http://jxva.com/down/rpm/upgrade\_kernel\_for\_centos6/perf-3.10.60-1.el6.elrepo.x86\_64.rpm |

内核升级成功之后，我们还需要进行如下操作

|  |
| --- |
| $ vim /etc/grub.conf  //设置default值（从0开始）  $ reboot  $ uname -r |

## Create User for Docker

为docker的操作建立一个docker专用组与用户对于安全方面来说，通常是一个好的办法

|  |
| --- |
| $ groupadd docker  //添加用户  $ adduser --system --shell /sbin/nologin --comment 'Docker' --create-home --home-dir /home/docker/ docker  //把docker用户加到docker与root组：  $ usermod -a -G docker,root docker |

后继操作可以直接通过su docker来对docker服务进行操作与管理。

## Install Docker

Docker是由GO语言编写的，编译docker是一件比较痛苦的事情，一般我们直接从官网上下载已经编译好的二进制文件格式，下载最新的docker二进制格式地址为：

|  |
| --- |
| $ wget https://get.docker.com/builds/Linux/x86\_64/docker-latest -O docker  $ mv docker /usr/bin/docker  $ chmod +x /usr/bin/docker  //查看docker版本信息  $ docker version  //启动docker服务  $ docker –d  //关闭docker服务  $ killall docker |

## Install Basic Dependencies

在安装一些基本依赖时，经常需要使用到epel源，安装epel源的方法如下：

针对CentOS6

|  |
| --- |
| $ rpm -Uvh http://mirrors.aliyun.com/centos/6/extras/x86\_64/Packages/epel-release-6-8.noarch.rpm  $ yum install docker-io --enablerepo=epel |

针对CentOS7

|  |
| --- |
| $ wget http://mirrors.aliyun.com/centos/7/extras/x86\_64/Packages/epel-release-7-2.noarch.rpm  $ rpm -ivh epel-release-7-2.noarch.rpm |

如果你的Linux内核为3.8或高于3.8可以略过此小节，安装docker的依赖环境主要是在比较低的Linux内核版本系统上，比如CentOS6等。

### Install LXC

lxc是在epel源中，其未带centos系统模板，在centos原始源中并不存在，需要 为centos增加epel源地址，命令如下：

|  |
| --- |
| $ wget http://mirrors.aliyun.com/centos/6/extras/x86\_64/Packages/epel-release-6-8.noarch.rpm  $ rpm -ivh epel-release-6-8.noarch.rpm  $ yum install lxc |

也可以直接从<http://jxva.com/down/rpm/lxc> 上直接下载进行安装，

|  |
| --- |
| $ rpm -Uvh http://jxva.com/down/rpm/lxc/lua-alt-getopt-0.7.0-1.el6.noarch.rpm  $ rpm -Uvh http://jxva.com/down/rpm/lxc/lua-filesystem-1.4.2-1.el6.x86\_64.rpm  $ rpm -Uvh http://jxva.com/down/rpm/lxc/lua-lxc-1.0.6-1.el6.x86\_64.rpm  $ rpm -Uvh http://jxva.com/down/rpm/lxc/lxc-1.0.6-1.el6.x86\_64.rpm  $ rpm -Uvh http://jxva.com/down/rpm/lxc/lxc-libs-1.0.6-1.el6.x86\_64.rpm |

检查lxc运行环境

|  |
| --- |
| $ service lxc start  $ lxc-checkconfig |

### Install Cgroup

Cgroup直接yum安装即可

|  |
| --- |
| $ yum install libcgroup libcap-devel  $ chkconfig cgconfig on  $ service cgconfig start |

也可以直接从<http://jxva.com/down/rpm/>上直接下载进行安装

|  |
| --- |
| $ rpm -Uvh http://jxva.com/down/rpm/libcap-devel-2.16-5.5.el6.x86\_64.rpm  $ rpm -Uvh http://jxva.com/down/rpm/libcgroup-0.40.rc1-12.el6.x86\_64.rpm |

## The Usage of Docker

Docker官方为了让用户快速了解Docker，提供了一个[交互式教程](https://www.docker.com/tryit/#0)，旨在帮助用户掌握Docker命令行的使用方法。但是由于Docker技术的快速发展，此交互式教程已经无法满足Docker用户的实际使用需求，所以让我们一起开始一次真正的命令行学习之旅。首先，Docker的命令清单可以通过运行 docker ，或者 docker help 命令得到。

|  |
| --- |
| [root@iZ23u5zzzcpZ ~]# docker  Usage: docker [OPTIONS] COMMAND [arg...]  A self-sufficient runtime for linux containers.  Options:  --api-enable-cors=false Enable CORS headers in the remote API  -b, --bridge="" Attach containers to a pre-existing network bridge  use 'none' to disable container networking  --bip="" Use this CIDR notation address for the network bridge's IP, not compatible with -b  -D, --debug=false Enable debug mode  -d, --daemon=false Enable daemon mode  --dns=[] Force Docker to use specific DNS servers  --dns-search=[] Force Docker to use specific DNS search domains  -e, --exec-driver="native" Force the Docker runtime to use a specific exec driver  --fixed-cidr="" IPv4 subnet for fixed IPs (ex: 10.20.0.0/16)  this subnet must be nested in the bridge subnet (which is defined by -b or --bip)  -G, --group="docker" Group to assign the unix socket specified by -H when running in daemon mode  use '' (the empty string) to disable setting of a group  -g, --graph="/var/lib/docker" Path to use as the root of the Docker runtime  -H, --host=[] The socket(s) to bind to in daemon mode or connect to in client mode, specified using one or more tcp://host:port, unix:///path/to/socket, fd://\* or fd://socketfd.  --icc=true Allow unrestricted inter-container and Docker daemon host communication  --insecure-registry=[] Enable insecure communication with specified registries (no certificate verification for HTTPS and enable HTTP fallback) (e.g., localhost:5000 or 10.20.0.0/16)  --ip=0.0.0.0 Default IP address to use when binding container ports  --ip-forward=true Enable net.ipv4.ip\_forward  --ip-masq=true Enable IP masquerading for bridge's IP range  --iptables=true Enable Docker's addition of iptables rules  -l, --log-level="info" Set the logging level  --label=[] Set key=value labels to the daemon (displayed in `docker info`)  --mtu=0 Set the containers network MTU  if no value is provided: default to the default route MTU or 1500 if no default route is available  -p, --pidfile="/var/run/docker.pid" Path to use for daemon PID file  --registry-mirror=[] Specify a preferred Docker registry mirror  -s, --storage-driver="" Force the Docker runtime to use a specific storage driver  --selinux-enabled=false Enable selinux support. SELinux does not presently support the BTRFS storage driver  --storage-opt=[] Set storage driver options  --tls=false Use TLS; implied by --tlsverify flag  --tlscacert="/root/.docker/ca.pem" Trust only remotes providing a certificate signed by the CA given here  --tlscert="/root/.docker/cert.pem" Path to TLS certificate file  --tlskey="/root/.docker/key.pem" Path to TLS key file  --tlsverify=false Use TLS and verify the remote (daemon: verify client, client: verify daemon)  -v, --version=false Print version information and quit  Commands:  attach Attach to a running container  build Build an image from a Dockerfile  commit Create a new image from a container's changes  cp Copy files/folders from a container's filesystem to the host path  create Create a new container  diff Inspect changes on a container's filesystem  events Get real time events from the server  exec Run a command in a running container  export Stream the contents of a container as a tar archive  history Show the history of an image  images List images  import Create a new filesystem image from the contents of a tarball  info Display system-wide information  inspect Return low-level information on a container  kill Kill a running container  load Load an image from a tar archive  login Register or log in to a Docker registry server  logout Log out from a Docker registry server  logs Fetch the logs of a container  port Lookup the public-facing port that is NAT-ed to PRIVATE\_PORT  pause Pause all processes within a container  ps List containers  pull Pull an image or a repository from a Docker registry server  push Push an image or a repository to a Docker registry server  restart Restart a running container  rm Remove one or more containers  rmi Remove one or more images  run Run a command in a new container  save Save an image to a tar archive  search Search for an image on the Docker Hub  start Start a stopped container  stop Stop a running container  tag Tag an image into a repository  top Lookup the running processes of a container  unpause Unpause a paused container  version Show the Docker version information  wait Block until a container stops, then print its exit code  Run 'docker COMMAND --help' for more information on a command. |

### Command Line Help

在Docker容器技术不断演化的过程中，Docker的子命令已经达到34个之多，其中核心子命令(例如：run)还会有复杂的参数配置。笔者通过结合功能和应用场景方面的考虑，把命令行划分为4个部分，方便我们快速概览Docker命令行的组成结构：

|  |  |
| --- | --- |
| **功能划分** | **命令** |
| 环境信息相关 | info  version |
| 系统运维相关 | attach  build  commit  cp  diff  export  images  import / save / load  inspect  kill  port  pause / unpause  ps  rm  rmi  run  start / stop / restart  tag  top  wait |
| 日志信息相关 | events  history  logs |
| Docker Hub服务相关 | login  pull / push  search |

(a) 参数约定

单个字符的参数可以放在一起组合配置，例如

|  |
| --- |
| $ docker run -t -i --name test busybox sh |

可以用这样的方式等同：

|  |
| --- |
| $ docker run -ti --name test busybox sh |

(b) Boolean

Boolean参数形式如： -d=false。注意，当你声明这个Boolean参数时，比如 docker run -d=true，它将直接把启动的Container挂起放在后台运行。

(c) 字符串和数字

参数如 --name=“”定义一个字符串，它仅能被定义一次。同类型的如-c=0 定义一个数字，它也只能被定义一次。

### Background Process

Docker后台进程是一个常驻后台的系统进程，值得注意的是Docker使用同一个文件来支持客户端和后台进程，其中角色切换通过-d来实现。这个后台进程是用来管理容器的，使用Docker --help可以得到更详细的功能参数配置

Docker后台进程参数清单如下表：

|  |  |
| --- | --- |
| **参数** | **解释** |
| --api-enable-cors=false | 开放远程API调用的 [CORS 头信息](https://en.wikipedia.org/wiki/Cross-Origin_Resource_Sharing)。这个接口开关对想进行二次开发的上层应用提供了支持。 |
| -b, --bridge="" | 挂载已经存在的网桥设备到 Docker 容器里。注意，使用 none 可以停用容器里的网络。 |
| --bip="" | 使用 [CIDR](https://en.wikipedia.org/wiki/CIDR_notation) 地址来设定网络桥的 IP。注意，此参数和 -b 不能一起使用。 |
| -D, --debug=false | 开启Debug模式。例如：docker -d -D |
| -d, --daemon=false | 开启Daemon模式。 |
| --dns=[] | 强制容器使用DNS服务器。例如： docker -d --dns 8.8.8.8 |
| --dns-search=[] | 强制容器使用指定的DNS搜索域名。例如： docker -d --dns-search example.com |
| -e, --exec-driver="native" | 强制容器使用指定的运行时驱动。例如：docker -d -e lxc |
| -G, --group="docker" | 在后台运行模式下，赋予指定的Group到相应的unix socket上。注意，当此参数 --group 赋予空字符串时，将去除组信息。 |
| -g, --graph="/var/lib/docker" | 配置Docker运行时根目录 |
| -H, --host=[] | 在后台模式下指定socket绑定，可以绑定一个或多个 tcp://host:port, unix:///path/to/socket, fd://\* 或 fd://socketfd。例如：  $ docker -H tcp://0.0.0.0:2375 ps 或者  $ export DOCKER\_HOST="tcp://0.0.0.0:2375" $ docker ps |
| --icc=true | 启用内联容器的通信。 |
| --ip="0.0.0.0" | 容器绑定IP时使用的默认IP地址 |
| --ip-forward=true | 启动容器的 net.ipv4.ip\_forward |
| --iptables=true | 启动Docker容器自定义的iptable规则 |
| --mtu=0 | 设置容器网络的MTU值，如果没有这个参数，选用默认 route MTU，如果没有默认route，就设置成常量值 1500。 |
| -p, --pidfile="/var/run/docker.pid" | 后台进程PID文件路径。 |
| -r, --restart=true | 重启之前运行中的容器 |
| -s, --storage-driver="" | 强制容器运行时使用指定的存储驱动，例如,指定使用devicemapper, 可以这样：  docker -d -s devicemapper |
| --selinux-enabled=false | 启用selinux支持 |
| --storage-opt=[] | 配置存储驱动的参数 |
| --tls=false | 启动TLS认证开关 |
| --tlscacert="/Users/dxiao/.docker/ca.pem" | 通过CA认证过的的certificate文件路径 |
| --tlscert="/Users/dxiao/.docker/cert.pem" | TLS的certificate文件路径 |
| --tlskey="/Users/dxiao/.docker/key.pem" | TLS的key文件路径 |
| --tlsverify=false | 使用TLS并做后台进程与客户端通讯的验证 |
| -v, --version=false | 显示版本信息 |

注意，其中带有[] 的启动参数可以指定多次，例如

|  |
| --- |
| $ docker run -a stdin -a stdout -a stderr -i -t ubuntu /bin/bash |

通过以上Docker命令行的详细解释，可以强化对Docker命令的全面理解。考虑到Docker命令行的发展变化非常快，读者可以参考官方的[命令行解释](http://docs.docker.com/reference/commandline/cli/)文档更新相应的命令行解释。另外，通过以上Docker命令行的分析，可以知道Docker命令行架构设计的特点在于客户端和服务端的运行文件是同一个文件，内部实现代码应该是重用的设计。笔者希望开发者在开发类似的命令行应用时参考这样的设计，减少前后台容错的复杂度。

## Quick Start

### Download the Image File

我们可以自制一个CentOS或Ubuntu等镜像文件，但是过程比较麻烦，这里直接从Docker官方下载一个已经制作好的CentOS7镜像文件，下载地址为：

<https://github.com/CentOS/sig-cloud-instance-images/raw/c7bfde2d0e3ed621ecfbf02aae7b50d4c912b0f1/docker/centos-7-20150102_1408-docker.tar.xz>

### Import the Image File

在导入镜像文件之前，需要确保docker服务已经启动，否则无法进行导入操作，导入镜像的命令可以采用如下任一一个

|  |
| --- |
| $ cat centos-7-20150102\_1408-docker.tar.xz | docker import - centos7  $ docker import - centos7 < centos-7-20150102\_1408-docker.tar.xz  $ xz -d -k < centos-7-20150102\_1408-docker.tar.xz | docker import - centos7 |

|  |  |
| --- | --- |
|  |  |

导入成功之后，可以查看docker服务之中已有的镜像

|  |
| --- |
| $ docker images |

|  |  |
| --- | --- |
|  |  |

//TODO 增加一个截图

执行docker里面的shell命令

|  |
| --- |
| $ docker run centos7 /bin/echo hello world  $ docker run -i -t centos7 /bin/bash |

|  |  |
| --- | --- |
|  |  |

## Install OpenSSH

在Docker中安装SSH服务不是Docker所主张的，Docker提倡的是单个Docker服务只提供一个应用级服务，即应用服务容器化，但是在平台的使用Docker中，我们还是更倾向于Docker做为一个类虚拟机使用，所以笔者通常会在Docker中安装一个SSH服务，并通过SSH来管理与部署应用。

|  |
| --- |
| $ yum update  $ yum -y install openssh-server passwd net-tools  $ mkdir /var/run/sshd  $ vi /etc/pam.d/sshd  //找到下面文件，注释掉这一行  #session required pam\_loginuid.so  $ vi /etc/ssh/sshd\_config  PermitRootLogin yes  UseDNS no  //修改root密码  $ passwd  //生成默认的key  $ ssh-keygen -t rsa -f /etc/ssh/ssh\_host\_rsa\_key  $ ssh-keygen -t dsa -f /etc/ssh/ssh\_host\_dsa\_key  $ ssh-keygen -t ecdsa -f /etc/ssh/ssh\_host\_ecdsa\_key  //启动ssh服务  $ /usr/sbin/sshd |

|  |  |
| --- | --- |
|  |  |

## Install Supervisord

利用Supervisord我们可以在单个Docker容器中启动多个应用服务。Supervisord是运行在python环境下的服务监控程序，所以在安装supervisord之前必须有python环境。

如果[系统](http://www.2cto.com/os/)没有python, 键入: yum install python (CentOS)或者apt-get install python (Ubuntu)。

安装supervisord命令如下：

|  |
| --- |
| $ yum -y python-setuptools  $ easy\_install supervisor  $ mkdir -p /var/log/supervisor |

### Configure Supervisord

|  |  |
| --- | --- |
|  |  |

新建supervisord的默认配置文件

|  |
| --- |
| $ vi /etc/supervisord.conf |

|  |  |
| --- | --- |
|  |  |

内容如下：

|  |
| --- |
| [supervisord]  nodaemon=true  [program:sshd]  command=/usr/sbin/sshd -D  #[program:hosts]  #command=/bin/bash -c "/bin/echo '10.204.79.222 mirror.centos.org' >> /etc/hosts"  ##autostart=true  #redirect\_stderr=true  #exitcodes=0  #startretries=0  #[program:redis-server]  #command=/data/program/redis-2.8.12/bin/redis-server /data/program/redis-2.8.12/etc/redis.conf  #add another service  #[program:hello]  #command=/bin/bash -c "/root/test.sh" |

可能直接采用如下命令测试一下/etc/supervisord.conf文件的启动过程

|  |
| --- |
| $ supervisord |

|  |  |
| --- | --- |
|  |  |

### Configure Servers

采用supervisord方式启动应用服务器时，应用服务器不能使用守护方式启动，否则supervisord进程无法监控到应用服务器进程的存在，所以我们通常会将nginx与redis等的进程启动改为非守护方式。

针对redis，我们需要将redis.conf中的daemonize 参数改为no;

针对nginx，我们需要将nginx.conf中的daemon 参数改为off;

对于supervisord管理的进程使用即退出的，我们要在管理的进程中加入参数startretries为0，否则supervisord会对其进行多次探测，甚至导致docker系统CPU跑满。

## Image Persistence and Restore

在管理镜像与容器方面，docker提供了两个成对的命令：

import与export : 导入与导出容器

save与load：保存与加载镜像

导出后再导入(exported-imported)的容器会丢失所有的历史，而保存后再加载（saveed-loaded）的镜像没有丢失历史和层(layer)。 这意味着使用导出后再导入的方式，你将无法回滚到之前的层(layer)，同时，使用保存后再加载的方式持久化整个镜像，就可以做到层回滚。

可以执行docker tag <LAYER ID> <IMAGE NAME>来回滚之前的层。

### Export Image

采用export导出的镜像文件将无法保存镜像的历史层，仅保存最后的单个容器状态，笔者一般采用docker官方原始镜像加入sshd与supervisord服务之后，再部署特定的应用服务，然后直接采用export导出备份好镜像，不保存历史层的镜像文件会更小，方便后续迁移。

笔者通常采用的镜像管理方式如下：

* 从docker官网下载centos-7-20150102\_1408-docker.tar.xz;
* 通过centos-7-20150102\_1408-docker.tar制作成centos7\_sshd\_20150111.tar.xz(带有sshd及supervisord服务)；
* 通过centos7\_sshd\_20150111.tar.xz来制作gitlab、redmine等镜像，如：centos7\_sshd\_gitlab\_gitlab-ci\_20150112.tar.xz、centos7\_sshd\_redmine\_20150112.tar.xz。

导出容器的命令如下：

|  |
| --- |
| $ docker export <CONTAINER ID> | xz -z -k -6 > /data/centos7\_sshd\_20150111.tar.xz |

|  |  |
| --- | --- |
|  | xz是一个解压软件，-6表示压缩比率，数字越大，压缩比越高。 |

*注：需要确保宿主系统中已经安装好了xz解压软件*

也可以使用如下命令之一：

|  |
| --- |
| $ docker export <CONTAINER ID> | bzip2 -6 -c> /data/centos7\_sshd\_20150111.tar.bz2  $ docker export <CONTAINER ID> > /data/centos7\_sshd\_20150111.tar |

### Import Image

导入镜像的命令可以采用如下任一一个

|  |
| --- |
| $ cat centos-7-20150102\_1408-docker.tar.xz | docker import - centos7  $ docker import - centos7 < centos-7-20150102\_1408-docker.tar.xz  $ xz -d -k < centos-7-20150102\_1408-docker.tar.xz | docker import - centos7 |

|  |  |
| --- | --- |
|  |  |

### Save Image

使用save命令保存镜像，可以保存镜像的所有层，采用如下命令可以查看镜像的树型层结构

|  |
| --- |
| $ docker images --tree |

*注： --tree参数现在不推荐使用，在以后的版本中可能会被废弃*

保存镜像的命令可以采用如下任一一个：

|  |
| --- |
| $ docker save <IMAGE NAME> | bzip2 -6 -c > centos-7-20150102\_1408-docker.tar.bz2  $ docker save <IMAGE NAME> | xz -z -k -6 > centos-7-20150102\_1408-docker.tar.xz |

### Load Image

加载镜像的命令可以采用如下任一一个：

|  |
| --- |
| $ bzip2 -d -c < centos-7-20150102\_1408-docker.tar.bz2 | docker load  $ xz -d -k < centos-7-20150102\_1408-docker.tar.xz | docker load |

# Docker Image Management

本章主要讲解如何自制镜像文件、在阿里云上如何使用docker服务、如何使用docker容器化布署、管理、分发应用等高级进阶用法。

## Made by Oneself Docker Image

虽然直接采用docker官方提供的原始镜像是笔者强烈建议的，　并且自制镜像文件是一个比较折腾的工作，不过自己动手完成镜像的制作，还是挺有意思与挑战的。

### Make Image in CentOS6

在centos6下可以通过febootstrap这个工具来制作最小的镜像文件，安装febootstrap采用如下命令

|  |
| --- |
| $ yum -y install febootstrap  //采用docker用户来进行操作  $ su docker  $ mkdir -p /home/docker/images  $ cd /home/docker/images  $ febootstrap -i bash -i wget -i yum -i iputils -i iproute centos6 centos6-image http://mirrors.163.com/centos/6/os/x86\_64/ |

(-i 安装package， centos6 操作系统版本，centos6-images安装目录，最后是源地址)

上面成功后：

|  |
| --- |
| $ cd /home/docker/images/centos6-images  $ tar -c . | docker import - centos65-bash |

(centos65-bash为docker镜像名字)

|  |
| --- |
| $ docker images  REPOSITORY       TAG           IMAGE ID         CREATED             VIRTUAL SIZE  centos65-bash   latest        d2ecedd2b5ed      8 minutes ago       306.7 MB |

测试docker镜像，创建一个容器：

|  |
| --- |
| $ docker run -t -i centos65-bash /bin/bash |

退出后

|  |
| --- |
| $ docker ps -a  CONTAINER         ID  IMAGE     COMMAND        CREATED  STATUS PORTS     NAMES  14c9afaf2f06 centos65-bash:latest  /bin/bash 7 minutes ago Exited (0) 6 minutes ago insane\_lalande |

启动刚才创建的容器：

|  |
| --- |
| $ docker start 14c9afaf2f06 |

进入启动后的容器：

|  |
| --- |
| $ docker attach 14c9afaf2f06 |

一个简单的镜像就制作完成了。

导出镜像和导入镜像方便分享镜像文件

导出镜像：

|  |
| --- |
| $ docker save <IMAGENAME> | bzip2 -6 -c>img.tar.bz2 |

导入镜像(换一台机器)：

|  |
| --- |
| $ bzip2 -d -c < img.tar.bz2 | docker load |

采用febootstrap命令，我们还可以制作centos7等镜像，下面的命令列举了常用的镜像制作方法

|  |
| --- |
| $ febootstrap -i bash -i yum -i iputils centos7 centos7-image http://mirror.centos.org/centos/7.0/  $ febootstrap -i bash -i yum -i iputils centos6 centos6-image http://mirror.centos.org/centos/6.5/  $ febootstrap -i bash -i wget -i yum -i iputils -i iproute centos6 centos6-image http://mirrors.163.com/centos/6/os/x86\_64/ |

### Make Image in CentOS7

在CentOS7下需要通过supermin这个工具来制作最小的镜像文件，febootstrap的后续版本改名为supermin，supermin官网为：<http://people.redhat.com/~rjones/supermin/>

(a) 直接安装

|  |
| --- |
| $ yum install supermin |

(b) 自己编译supermin

|  |
| --- |
| $ wget http://libguestfs.org/download/supermin/supermin-5.1.11.tar.gz  $ yum install ocaml ocaml-findlib-devel e2fsprogs-devel  $ yum install rpm-devel  //supermin是采用静态编译的，所以需要安装静态依赖库  $ yum install -y glibc-static libstdc++-static  $ tar -xvf supermin-5.1.11.tar.gz  $ cd supermin-5.1.11  $ ./configure  $ make  $ make install  $ supermin -V |

使用supermin的命令如下

|  |
| --- |
| //for supermin v4  $ supermin --names bash coreutils -o centos7-basic http://mirrors.aliyun.com/centos/7.0/  //for supermin v5  $ supermin --prepare bash coreutils -o centos7-basic http://mirrors.aliyun.com/centos/7.0/ |

## Use Docker in Ali Cloud

### Remove Default Route

由于阿里云路由与Docker启动的默认路由有冲突，需要删除如下一条阿里云的默认路由

|  |
| --- |
| $ route del -net 172.16.0.0 netmask 255.240.0.0 dev eth0 |

### Create a Net Bridge

如果不希望删除路由，可以为docker增加一个网桥，命令如下：

|  |
| --- |
| $ brctl addbr mydockerbridge $ ip addr add 192.168.1.1/24 dev mydockerbridge $ ip link set dev mydockerbridge up $ docker -d -b mydockerbridge |

### Bind a CIDR

或者使用 --bip 参数来手工分配给 CIDR 地址即可

|  |
| --- |
| $ docker -d --bip=192.168.100.1/24 |

CIDR（无类别域间路由，Classless Inter-Domain Routing）是一个在Internet上创建附加地址的方法，这些地址提供给服务提供商（ISP），再由ISP分配给客户。CIDR将路由集中起来，使一个IP地址代表主要骨干提供商服务的几千个IP地址，从而减轻Internet[路由器](http://baike.baidu.com/view/1360.htm)的负担。

## Daemon Service

Docker后台进程是一个常驻后台的系统进程，值得注意的是Docker使用同一个文件来支持客户端和后台进程，其中角色切换通过-d来实现。这个后台进程是用来管理容器的，使用Docker --help可以得到更详细的功能参数配置。

Docker的模式这个与rsync的模式类似，启动Docker的守护方式有很多种，最简单的就是如下方式：

|  |
| --- |
| $ docker -d  $ docker --daemon=true |

-d, --daemon=false，开启Daemon模式。

默认情况，Docker的工作目录为/var/lib/docker，我们可以通过指定-g或--graph参数来指定工作目录，通常笔者采用的工作目录为/data/docker

|  |
| --- |
| $ docker --graph="/data/docker" -d |

如果采用守护方式，可以采用如下命令

|  |
| --- |
| $ nohup docker --graph="/data/docker" -d > /dev/null &  $ nohup docker --graph="/data/docker" --bip="172.17.42.1/24" -d > /dev/null &  $ nohup docker --graph="/data/docker" --bip="192.168.42.1/24" -d > /dev/null & |

为Docker服务指定一个工作目录及IP簇通常是一个好的方式。

## The Use of Docker in Practice

以实践的方式介绍Docker的使用技巧

### Running Practice

这一小节主要讲解常用的docker实践命令

1. 对CPU资源的计算能力进行了限制，同时限制了内存最大使用为2G

|  |
| --- |
| $ docker run -c 512 -m 2G -i -t centos7 /bin/bash |

1. 将docker容器内的22与80端口映射为宿主的8022与8080端口

|  |
| --- |
| $ docker run -d -p 8022:22 -p 8080:80 centos7\_sshd /usr/sbin/sshd -D |

1. 为docker容器名为jxva-docker，docker内的系统主机名为jxva-docker,

|  |
| --- |
| $ docker run --hostname="jxva-docker" --name="jxva-docker" -p 22022:22 -p 22080:80 centos7\_sshd /etc/init.d/init.sh |

1. 在(c)的基础上增加了文件目录映射

|  |
| --- |
| $ docker stop jxva-docker  $ docker rm jxva-docker  $ docker run --hostname="jxva-docker" --name="jxva-docker" --memory="4g" -d -v /data/home/jxva:/data -p 22022:22 -p 22080:80 -p 22443:443 -p 3000:3000 centos7\_sshd supervisord -c /etc/supervisord.conf |

将主机目录挂载到docker容器内,通过docker run命令的-v/--volume参数，文件目录映射的格式为：[host-dir]:[container-dir]:[rw|ro]，默认为rw可读写。

1. 目前生产环境采用的比较全的

|  |
| --- |
| $ docker run --hostname="jxva-docker" --name="jxva-docker" -c 128 --memory="4g" -d -v /data/home/jxva:/data -v /data/docker/jdk1.8.0\_25:/data/program/jdk1.8.0\_25:ro -v /data/docker/mysql-5.6.19-linux-x86\_64:/usr/local/mysql:ro -p 22022:22 -p 22080:80 -p 22443:443 -p 22336:3306 -p 22300:3000 centos7\_sshd supervisord -c /etc/supervisord.conf |

为了管理docker服务方便，我们编写了一个docker容器管理脚本，方便管理各自容器，脚本如下：

|  |
| --- |
| #! /bin/sh  declare -a users  users=([11]='liumy' [13]='guosy' [15]='wujx' [17]='linzy' [19]='lisa' [21]='gaoyb' [22]='jxva')  start () {  if [[ $1 = 'all' ]]; then  echo -e $"Starting all docker services \n"  for key in ${!users[@]}  do  value=${users[$key]}  docker run --hostname="${value}-docker" --name="$value-docker" --memory="4g" -d -v /data/home/${value}:/data -p ${key}022:22 -p ${key}080:80 -p ${key}443:443 -p ${key}336:3306 centos7\_sshd supervisord -c /etc/supervisord.conf  echo $key '--' $value 'start'  done  else  exist=0  for key in ${!users[@]}  do  #echo $key '--' ${users[$key]}  value=${users[$key]}  if [[ $value = $1 ]]; then  exist=1  docker run --hostname="${value}-docker" --name="$value-docker" --memory="4g" -d -v /data/home/${value}:/data -p ${key}022:22 -p ${key}080:80 -p ${key}443:443 -p ${key}336:3306 centos7\_sshd supervisord -c /etc/supervisord.conf  echo $key '--' $value 'start'  fi  done  if [[ $exist = 0 ]]; then  echo 'No such user'  fi  fi  RETVAL=$?  echo  [ $RETVAL -eq 0 ]  }  stop () {  if [[ $1 = 'all' ]]; then  echo -e $"Stoping all docker services \n"  for key in ${!users[@]}  do  value=${users[$key]}  docker stop ${value}-docker  # docker rm ${value}-docker  echo $key '--' $value 'stop'  done  else  exist=0  for key in ${!users[@]}  do  value=${users[$key]}  if [[ $value = $1 ]]; then  exist=1  docker stop ${value}-docker  # docker rm ${value}-docker  echo $key '--' $value 'stop'  fi  done  if [[ $exist = 0 ]]; then  echo 'No such user'  fi  fi  RETVAL=$?  echo  [ $RETVAL -eq 0 ]  }  restart () {  stop $1  start $1  }  # See how we were called.  case "$1" in  start)  start $2  ;;  stop)  stop $2  ;;  restart)  restart $2  ;;  \*)  echo $"Usage: $0 {start|stop|restart} xxx"  RETVAL=2  ;;  esac  exit $RETVAL |

### The Commonly Used Skill

1. 查看容器IP

|  |
| --- |
| $ docker inspect --format='{{.NetworkSettings.IPAddress}}' <CONTAINER ID>  $ docker inspect <CONTAINER NAME|ID> | grep IPAddress |

## Dockerfile

//TODO

# Docker Hub

//TODO

# Docker Registry

[Docker Registry](https://github.com/dotcloud/docker-registry) 是一个存储和分享 [Docker](http://docker.io/) 镜像的服务, Docker 不用自己动手制作镜像，官方已经提供了很多版本的 Linux 镜像，直接从官方仓库（Public Repositories）下载就可以了。如果考虑到安全性和速度，我们可能会想在自己局域网里架设一个私有仓库（Private Repositories）来放我们自己的镜像，[Docker-Registry](https://github.com/dotcloud/docker-registry) 正是我们需要的工具。

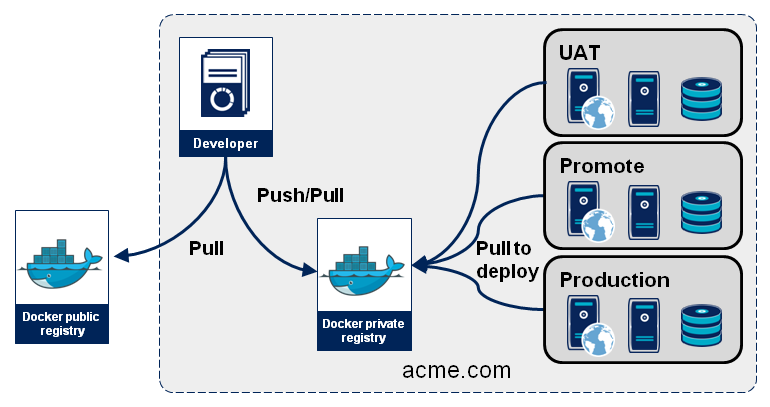
Docker registry 就是管理 docker 镜像的服务， Docker 公司维护的 registry 就是 [http://hub.docker.com](http://hub.docker.com/), 它可以让我们方便的下载预先做好的镜像。

|  |
| --- |
| $ docker pull centos7 |

上面的命令就是缺省的从这个Docker官方源下载。在国内为了加快访问，你也可以使用 [docker.cn](http://docker.cn/) 的服务，他们同步了常用的镜像，使用也非常方便，如：

|  |
| --- |
| $ docker pull docker.cn/docker/centos7 |

大部分公司在推广使用 docker 时，都会为了使用方便，在公司内部自己架设一个，不仅仅是为了安全、节省大量的带宽，而且也可以有效推动内部对docker的有效利用，如下图（源自[http://blog.octo.com/en/docker-registry-first-steps](http://blog.octo.com/en/docker-registry-first-steps/%EF%BC%89%E3%80%82) ）。



## Why do you need a Docker Registry?

当在自己的环境中创建 [Docker](http://docker.io/) 镜像的时候，无论是装 [Redis](http://redis.io/)，[Hipache](https://github.com/dotcloud/hipache)，还是 IRC 协议的 [logbot](https://github.com/dannvix/Logbot) ，你都希望可以把镜像存到一个安全的地方。也许你项目中的 [Docker](http://docker.io/) 镜像需要安装 [Jenkins](http://buildbot.net/)，或者每次 commit 都跑一遍 [Buildbot](http://buildbot.net/)，又或者给镜像打上 bag 和 tag （相关阅读：[docker commit](http://docs.docker.io/en/latest/reference/commandline/cli/#commit)，[docker tag](http://docs.docker.io/en/latest/reference/commandline/cli/#tag)），再发送到 [Docker Registry](https://github.com/dotcloud/docker-registry)。可是如果镜像中的代码是私有的，你不想把镜像放到公共的 [Docker Registry](http://index.docker.io/) 上呢？[Docker](http://docker.io/) 公司已经想到了这一点，并因此建立了 [docker-registry](https://github.com/dotcloud/docker-registry) 项目。[docker-registry](https://github.com/dotcloud/docker-registry) 允许你把自己的镜像 [push](http://docs.docker.io/en/latest/reference/commandline/cli/#push) 到自己的 registry 中。

如果你想感受一下[docker registry](https://github.com/dotcloud/docker-registry)，可以用公共的 [registry](http://index.docker.io/) 来试试：

|  |
| --- |
| $ docker pull samalba/docker-registry  $ docker run -d -p 5000:5000 samalba/docker-registry  // 我们先pull下来一个简单的镜像（或者自己做一个也可以）  $ docker pull busybox  $ docker tag busybox localhost:5000/busybox  $ docker push localhost:5000/busybox |

对于 registry 入门，这个例子很有用，但是例子中仅用了一个简单的 HTTP 服务。任何知道服务器地址的人都可以随意 push 镜像，这不是个好方案。下面我们来建立自己的私有 registry 以供内部使用。

## Set Up Our Own Registry Service

在这里笔者将介绍如何部署一个带 SSL 加密、HTTP 验证并有防火墙防护的私有 [Docker Registry](https://github.com/dotcloud/docker-registry)， 架设Docker Registry 需要系统支持[Upstart](http://upstart.ubuntu.com/) ， 目前基本的Linux都支持，笔者这里采用 [Nginx](http://nginx.org/)作为 [Docker Registry](https://github.com/dotcloud/docker-registry) 的前端代理服务器，同时也用 [Nginx](http://nginx.org/) 完成 SSL 加密和基本的 HTTP 验证。同时采用 [Gunicorn](http://gunicorn.org/) 运行[Docker Registry](https://github.com/dotcloud/docker-registry) 并用 [Upstart](http://upstart.ubuntu.com/) 管理 [Gunicorn](http://gunicorn.org/)。我们还用 [Redis](http://redis.io/) 实现一个 LRU(Least Recently Used，近期最少使用算法) 缓存机制来减少 [Docker Registry](https://github.com/dotcloud/docker-registry) 和硬盘之间的数据存取。

## Deploy and Configure the Registry (for registry v1.0)

安装Docker后，可以通过官方提供的[registry](https://github.com/dotcloud/docker-registry)镜像来简单搭建一套本地私有仓库环境，这里记录简单的搭建过程，具体参考网址为：<https://docs.docker.com/registry/deploying/>

### Use registry to start private warehouse container

采用如下命令会可以在宿主机上创建并启动一个私有仓库的docker容器

|  |
| --- |
| $ mkdir -p /data/docker-registry  $ docker run -d -p 5000:5000 -v /data/docker-registry:/tmp/registry registry |

说明：若之前没有安装registry容器则会自动下载并启动一个registry容器，创建本地的私有仓库服务。默认情况下，会将仓库创建在容器的/tmp/registry目录下，可以通过 -v 参数来将镜像文件存放在本地的指定路径上（例如，放在本地目录/data/docker-registry下）。

### Push mirror to private warehouse

|  |
| --- |
| $ docker push 104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry:14.04 |

说明：根据第一步启动的registry容器所在宿主主机的IP和Port，push某环境的本地容器。

1. root@gerryyang:~# docker push 104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry:14.04
2. The push refers to a repository [104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry] (len: 1)
3. Sending image list
4. Pushing repository 104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry (1 tags)
5. 511136ea3c5a: Image successfully pushed
6. 3b363fd9d7da: Image successfully pushed
7. 607c5d1cca71: Image successfully pushed
8. f62feddc05dc: Image successfully pushed
9. 8eaa4ff06b53: Image successfully pushed
10. 894c0161121f: Image successfully pushed
11. a45787b0222f: Image successfully pushed
12. f0e3262ed661: Image successfully pushed
13. Pushing tag for rev [f0e3262ed661] on {http://104.131.173.242:5000/v1/repositories/ubuntu\_sshd\_gcc\_gerry/tags/14.04}

宿主主机my\_registry的目录结构

1. root@gerryyang:~/my\_registry# tree
2. .
3. ├── images
4. │   ├── 3b363fd9d7dab4db9591058a3f43e806f6fa6f7e2744b63b2df4b84eadb0685a
5. │   │   ├── ancestry
6. │   │   ├── \_checksum
7. │   │   ├── json
8. │   │   └── layer
9. │   ├── 511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
10. │   │   ├── ancestry
11. │   │   ├── \_checksum
12. │   │   ├── json
13. │   │   └── layer
14. │   ├── 607c5d1cca71dd3b6c04327c3903363079b72ab3e5e4289d74fb00a9ac7ec2aa
15. │   │   ├── ancestry
16. │   │   ├── \_checksum
17. │   │   ├── json
18. │   │   └── layer
19. │   ├── 894c0161121f105ac9b81bca7ac583ac1f29772625911db0fa2b6b475f5642fd
20. │   │   ├── ancestry
21. │   │   ├── \_checksum
22. │   │   ├── json
23. │   │   └── layer
24. │   ├── 8eaa4ff06b53ff7730c4d7a7e21b4426a4b46dee064ca2d5d90d757dc7ea040a
25. │   │   ├── ancestry
26. │   │   ├── \_checksum
27. │   │   ├── json
28. │   │   └── layer
29. │   ├── a45787b0222f955d68d9db34fb18033144b8a78015d9e306a1613894da0fd86e
30. │   │   ├── ancestry
31. │   │   ├── \_checksum
32. │   │   ├── json
33. │   │   └── layer
34. │   ├── f0e3262ed6617896b306852c923e4c0e1d359b58b29a02ef849c4b8978c73c65
35. │   │   ├── ancestry
36. │   │   ├── \_checksum
37. │   │   ├── json
38. │   │   └── layer
39. │   └── f62feddc05dc67da9b725361f97d7ae72a32e355ce1585f9a60d090289120f73
40. │       ├── ancestry
41. │       ├── \_checksum
42. │       ├── json
43. │       └── layer
44. └── repositories
45. └── library
46. └── ubuntu\_sshd\_gcc\_gerry
47. ├── \_index\_images
48. ├── tag\_14.04
49. └── tag14.04\_json
51. 12 directories, 35 files

关于https的问题

1. root@gerryyang:~# docker push 104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry:14.04
2. FATA[0002] Error: Invalid registry endpoint https://104.131.173.242:5000/v1/: Get https://104.131.173.242:5000/v1/\_ping: EOF. If this private registry supports only HTTP or HTTPS with an unknown CA certificate, please add `--insecure-registry 104.131.173.242:5000` to the daemon's arguments. In the case of HTTPS, if you have access to the registry's CA certificate, no need for the flag; simply place the CA certificate at /etc/docker/certs.d/104.131.173.242:5000/ca.crt

解决方法：  
修改Docker配置文件  
vim /etc/default/docker  
增加以下一行  
DOCKER\_OPTS="$DOCKER\_OPTS --insecure-registry=104.131.173.242:5000"  
重启Docker  
sudo service docker restart  
refer: <http://dockerone.com/question/21>

### Private warehouse query method

curl http://104.131.173.242:5000/v1/search  
说明：使用curl查看仓库104.131.173.242:5000中的镜像。在结果中可以查看到ubuntu\_sshd\_gcc\_gerry，说明已经上传成功了。

1. gerryyang@mba:personal\_repository$curl http://104.131.173.242:5000/v1/search
2. {"num\_results": 0, "query": "", "results": []}
3. gerryyang@mba:personal\_repository$curl http://104.131.173.242:5000/v1/search
4. {"num\_results": 1, "query": "", "results": [{"description": "", "name": "library/ubuntu\_sshd\_gcc\_gerry"}]}

### Access and download the mirror of the private warehouse on the other machines

docker pull 104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry:14.04

1. root@gerryyang:~# docker pull 104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry:14.04
2. Pulling repository 104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry
3. f0e3262ed661: Download complete
4. 511136ea3c5a: Download complete
5. 3b363fd9d7da: Download complete
6. 607c5d1cca71: Download complete
7. f62feddc05dc: Download complete
8. 8eaa4ff06b53: Download complete
9. 894c0161121f: Download complete
10. a45787b0222f: Download complete
11. Status: Image is up to date for 104.131.173.242:5000/ubuntu\_sshd\_gcc\_gerry:14.04

## Deploying a registry server (for registry v2.0)

1. Install Docker.
2. Run the hello-world image from the Docker public registry.

$ docker run hello-world

The run command automatically pulls a hello-world image from Docker's official images.

1. Start a registry on your localhost.

$ docker run -p 5000:5000 registry:2.0

This starts a registry on your DOCKER\_HOST running on port 5000.

1. List your images.

$ docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

registry 2.0 bbf0b6ffe923 3 days ago 545.1 MB

golang 1.4121a93c904635 days ago 514.9 MB

hello-world latest e45a5af57b00 3 months ago 910 B

Your list should include a hello-world image from the earlier run.

1. Retag the hello-world image for your local repoistory.

$ docker tag hello-world:latest localhost:5000/hello-mine:latest

The command labels a hello-world:latest using a new tag in the [REGISTRYHOST/]NAME[:TAG]format. The REGISTRYHOST is this case is localhost. In a Mac OSX environment, you'd substitute$(boot2docker ip):5000 for the localhost.

1. List your new image.

$ docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

registry 2.0 bbf0b6ffe923 3 days ago 545.1 MB

golang 1.4121a93c904635 days ago 514.9 MB

hello-world latest e45a5af57b00 3 months ago 910 B

localhost:5000/hello-mine latest ef5a5gf57b01 3 months ago 910 B

You should see your new image in your listing.

1. Push this new image to your local registry.

$ docker push localhost:5000/hello-mine:latest

The push refers to a repository [localhost:5000/hello-mine](len:1)

e45a5af57b00:Image already exists

31cbccb51277:Image successfully pushed

511136ea3c5a:Image already exists

Digest: sha256:a1b13bc01783882434593119198938b9b9ef2bd32a0a246f16ac99b01383ef7a

1. Use the curl command and the Docker Registry API v2 to list your image in the registry:

$ curl -v -X GET http://localhost:5000/v2/hello-mine/tags/list\*Hostname was NOT found in DNS cache

\*Trying127.0.0.1...\*Connected to localhost (127.0.0.1) port 5000(#0)> GET /v2/hello-mine/tags/list HTTP/1.1>User-Agent: curl/7.35.0>Host: localhost:5000>Accept:\*/\*

>

< HTTP/1.1 200 OK

< Content-Type: application/json; charset=utf-8

< Docker-Distribution-Api-Version: registry/2.0

< Date: Sun, 12 Apr 2015 01:29:47 GMT

< Content-Length: 40

<

{"name":"hello-mine","tags":["latest"]}

\* Connection #0 to host localhost left intact

You can also get this information by entering the http://localhost:5000/v2/hello-mine/tags/listaddress in your browser.

1. Remove all the unused images from your local environment:

$ docker rmi -f $(docker images -q -a )

This command is for illustrative purposes; removing the image forces any run to pull from a registry rather than a local cache. If you run docker images after this you should not see any instance ofhello-world or hello-mine in your images list.

$ docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

registry 2.0 bbf0b6ffe923 3 days ago 545.1 MB

golang 1.4121a93c904635 days ago 514.9 MB

1. Try running hello-mine.

$ docker run hello-mine

Unable to find image 'hello-mine:latest' locally

Pulling repository hello-mine

FATA[0001]Error: image library/hello-mine:latest not found

The run command fails because your new image doesn't exist in the Docker public registry.

1. Now, try running the image but specifying the image's registry:

$ docker run localhost:5000/hello-mine

If you run docker images after this you'll fine a hello-mine instance.

|  |
| --- |
| docker run -p 5000:5000 -v /data/docker-registry:/tmp/registry-dev -e PATH=/go/bin:/usr/src/go/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin -e GOLANG\_VERSION=1.4.2 -e GOPATH=/go/src/github.com/docker/distribution/Godeps/\_workspace:/go -e DISTRIBUTION\_DIR=/go/src/github.com/docker/distribution registry:2.0 /go/bin/registry /go/src/github.com/docker/distribution/cmd/registry/config.yml  root@e700c82c3444:/# find / -name 'registry'  /go/src/github.com/docker/distribution/registry  /go/src/github.com/docker/distribution/registry/middleware/registry  /go/src/github.com/docker/distribution/cmd/registry  /go/bin/registry  /var/lib/ucf/registry  root@e700c82c3444:/# find / -name 'config.yml'  /go/src/github.com/docker/distribution/cmd/registry/config.yml  /go/src/github.com/docker/distribution/docs/osx/config.yml  root@e700c82c3444:/# /go/src/github.com/docker/distribution/cmd/registry/config.yml |

## Run a local registry mirror

## Why?

If you have multiple instances of Docker running in your environment (e.g., multiple physical or virtual machines, all running the Docker daemon), each time one of them requires an image that it doesn't have it will go out to the internet and fetch it from the public Docker registry. By running a local registry mirror, you can keep most of the image fetch traffic on your local network.

## How does it work?

The first time you request an image from your local registry mirror, it pulls the image from the public Docker registry and stores it locally before handing it back to you. On subsequent requests, the local registry mirror is able to serve the image from its own storage.

## How do I set up a local registry mirror?

There are two steps to set up and use a local registry mirror.

### Step 1: Configure your Docker daemons to use the local registry mirror

You will need to pass the --registry-mirror option to your Docker daemon on startup:

sudo docker --registry-mirror=http://<my-docker-mirror-host> -d

For example, if your mirror is serving on http://10.0.0.2:5000, you would run:

sudo docker --registry-mirror=http://10.0.0.2:5000 -d

**NOTE:** Depending on your local host setup, you may be able to add the --registry-mirror options to theDOCKER\_OPTS variable in /etc/default/docker.

### Step 2: Run the local registry mirror

You will need to start a local registry mirror service. The [registry image](https://registry.hub.docker.com/_/registry/) provides this functionality. For example, to run a local registry mirror that serves on port 5000 and mirrors the content at registry-1.docker.io:

sudo docker run -p 5000:5000 \

-e STANDALONE=false \

-e MIRROR\_SOURCE=https://registry-1.docker.io \-e MIRROR\_SOURCE\_INDEX=https://index.docker.io \

registry

## Test it out

With your mirror running, pull an image that you haven't pulled before (using time to time it):

$ time sudo docker pull node:latest

Pulling repository node

[...]

real 1m14.078s

user 0m0.176s

sys 0m0.120s

Now, remove the image from your local machine:

$ sudo docker rmi node:latest

Finally, re-pull the image:

$ time sudo docker pull node:latest

Pulling repository node

[...]

real 0m51.376s

user 0m0.120s

sys 0m0.116s

The second time around, the local registry mirror served the image from storage, avoiding a trip out to the internet to refetch it.

# Docker API

//TODO

# Docker Advanced Usage

## Pull Docker via Proxy

有时候需要通过proxy代理上网，在使用docker pull时一直出现如下错误：

$docker run -d -p 5000:5000 -v /data/docker-registry:/tmp/registry registry:2.0

Unable to find image 'registry:2.0' locally

Pulling repository registry  
FATA[0000] Get https://index.docker.io/v1/repositories/library/registry/images: dial tcp: lookup index.docker.io: no DNS servers

在启动docker服务时，需要采用如下命令：

$ sudo HTTP\_PROXY=http://username:password@10.204.79.224:6428 docker --graph=/data/docker -d

同时修改/etc/sysconfig/docker

HTTP\_PROXY=http://186.100.4.107:808  
http\_proxy=$HTTP\_PROXY  
HTTPS\_PROXY=$HTTP\_PROXY  
https\_proxy=$HTTP\_PROXY  
export HTTP\_PROXY HTTPS\_PROXY http\_proxy https\_proxy

$ source /etc/sysconfig/docker

再次执行

$ docker run -d -p 5000:5000 -v /data/docker-registry:/tmp/registry registry:2.0

会显示如下结果：

Unable to find image 'registry:2.0' locally

Pulling repository registry

b39a445085a6: Pulling dependent layers

39bb80489af7: Download complete

df2a0347c9d0: Download complete

7a3871ba15f8: Download complete

## Initialize the Hosts File for Docker Containers

//TODO

## Bind a Static IP for the Docker Containers

//TODO

## Mapping Ports for Docker Containers

//TODO

## Customizing Permissions for Docker Containers

//TODO

# What is Next?

//TODO

# Reference Documentation

[1]<https://docs.docker.com/reference/>

[2] <http://kencochrane.net/blog/2013/08/the-docker-guidebook/>

[3] <http://dockerbook.com/>

[4] <http://paynedigital.com/articles/2013/11/introduction-to-docker>

[5] <http://blog.thoward37.me/articles/where-are-docker-images-stored/>