Docker 容器技术使用指南

目录

第一部分 Docker 容器技术基础及其应用场景介绍

1.1 Docker 的基本概念

Docker 容器是资源分割和调度的基本单位，封装整个服务的运行时环境，用于构建、发布和运行分布式应用的一个框架。它是一个跨平台、可移植并且简单易用的容器解决方案。Docker 的源代码托管在 GitHub 上，基于 Go 语言开发并遵从 Apache 2.0 协议。

Docker 容器可以快速自动化地部署应用，并通过操作系统内核技术(namespaces、cgroups 等)为容器提供资源隔离与安全保障。Docker 作为轻量级的虚拟化方式，实现了 PaaS 平台的高效部署、运行和维护。

1.2 为什么使用 Docker

（1）、持续部署与测试

Docker 消除了线上线下的环境差异，保证了应用生命周期的环境一致性和标准化。开发人员使用镜像实现标准开发环境的构建，开发完成后通过封装着完整环境和应用的镜像进行迁移，由此，测试和运维人员可以直接部署软件镜像来进行测试和发布，大大简化了持续集成、测试和发布的过程。

Docker 是革命性的，它重新定义了软件开发、测试、交付和部署的流程。我们交付的东西不再只是代码、配置文件、数据库定义等，而是整个应用服务及其运行环境。

（2）、优异的跨平台性

Docker 在原有 Linux 容器的基础上进行大胆革新，为容器设定了一 整套标准化的配置方法，将应用及其依赖的运行环境打包成镜像。Docker 是可移植(或者说跨平台)的，可以在各种主流操作系统上使用。Java 可以做到“一次编译，到处运行”，而 Docker 可以 “构建一次，在各平台上运行”(Build once，run anywhere)。越来越多的云平台都支持 Docker，用户再也无需担心受到云平台的捆绑，同时也让应用多平台混合部署成为可能。

（3）、高资源利用率与隔离

Docker 容器没有管理程序的额外开销，与底层共享操作系统，性能更加优良，系统负载更低，在同等条件下可以运行更多的应用实例，可以更充分地利 用系统资源。同时，Docker 拥

有不错的资源隔离与限制能力，可以精确地对应用分配 CPU、 内存等资源，保证了应用间不会相互影响。Docker 是轻量级虚拟化技术。与传统的 VM 相比，它更轻量，启动速度更快，单台硬件上可以同时跑成百上千个容器，所以非常适合在业务高峰期通过启动大量容器进行横向扩展。 Docker 容器技术的直接虚拟化不仅在技术方面使 CPU 利用率得到显著提升，还因 80：20 法则可在业务上更大程度发挥 CPU 利用率，真正体现了虚拟化精髓。

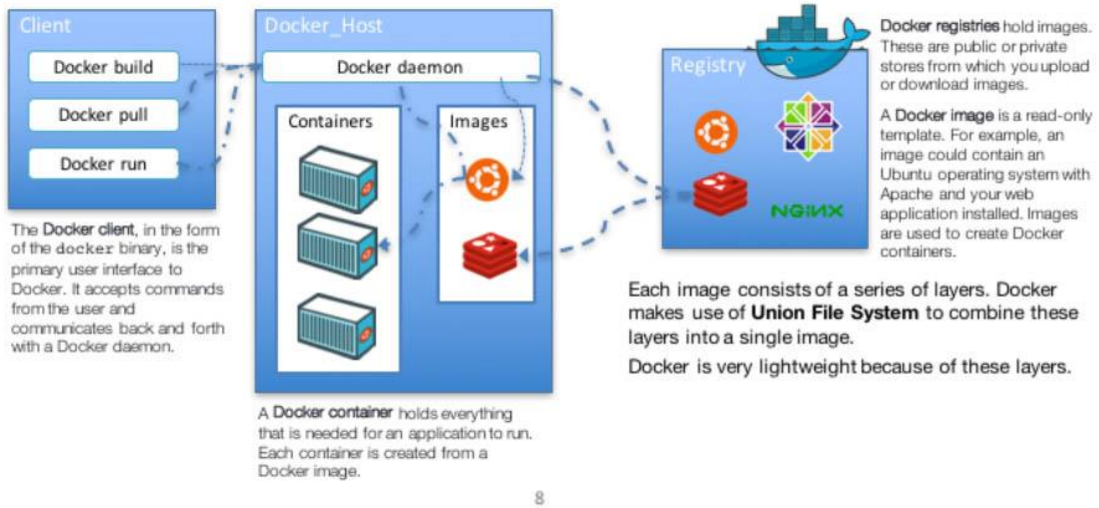
（4）、环境标准化和版本控制

可以使用 Git 等工具对 Docker 镜像进行版本控制，相比基于代码的版本控制来说，能够对整个应用运行环境实现版本控制，一旦出现故障可以快速回滚。相比以前的虚拟机镜像，Docker 压缩和备份速度更快，镜像启动也像启动一个普通进程一样快速

（5）、应用镜像仓库

Docker 官方构建了一个镜像仓库，组织和管理形式类似于 GitHub，其上 已累积了成千上万的镜像。因为 Docker 的跨平台适配性，相当于为用户提供了一个非常 有用的应用商店，所有人都可以自由地下载微服务组件，这为开发者提供了巨大便利。

1.3 Docker 体系结构简介



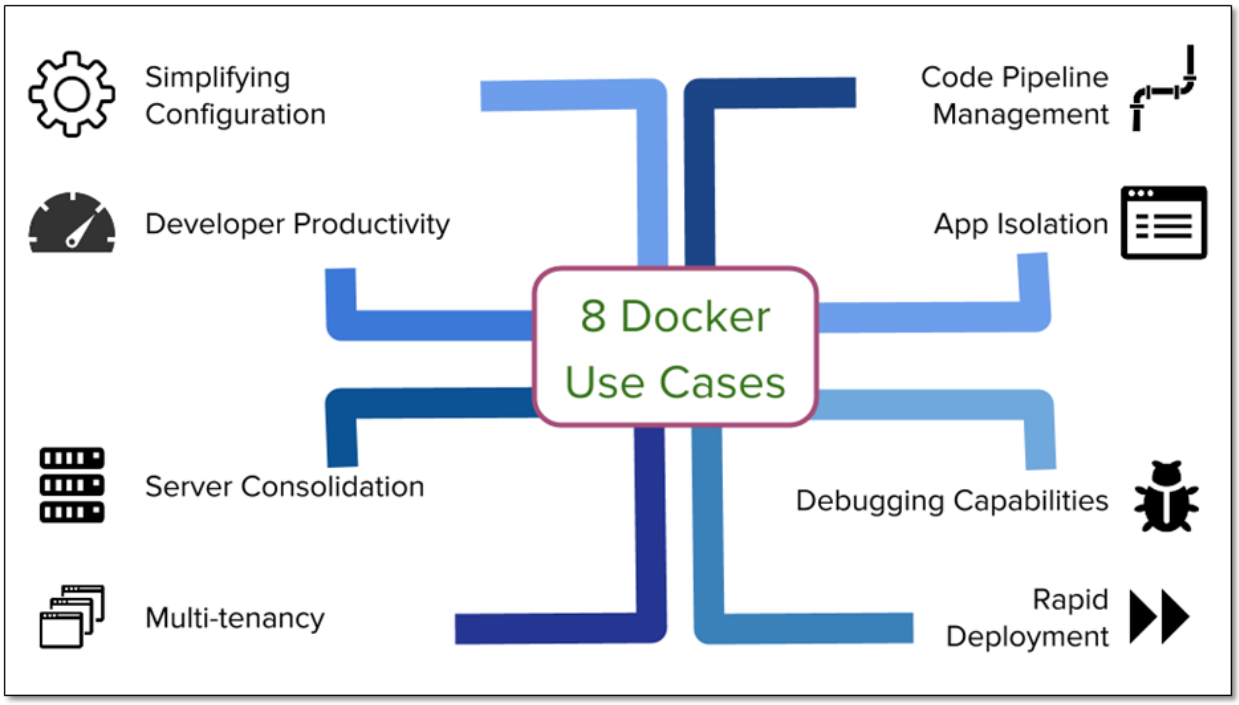
Docker 是一个客户/服务器（CIient/Server，CS）架构（见上图）。Docker 客户端是远程控制器，可通过 TCP REST 向 Docker Host 发送请求，包括创建容器、运行容器、保存容器、删除容器等请求。Docker 服务端的 Daemon 对客户端的请求进行相应的管理，随后通过 driver 转发至容器中的 libcontainer 执行环境。libcontainer 提供与不同 Linux 内核隔离的接口，类似命名空间及控制组。这种架构允许多个容器在共享同一个 Linux 内核的情况下完全隔离地

运行。

Docker 镜像(Images)	Docker 镜像是用于创建 Docker 容器的模板。
Docker 容器(Container)	容器是独立运行的一个或一组应用。
Docker 客户端(Client)	Docker 客户端通过命令行或者其他工具使用 Docker API (https://docs.docker.com/reference/api/docker_remote_api)与 Docker 的守护进程通信。
Docker 主机(Host)	一个物理或者虚拟的机器用于执行 Docker 守护进程和容器。
Docker 仓库(Registry)	Docker 仓库用来保存镜像，可以理解为代码控制中的代码仓库。Docker Hub(https://hub.docker.com) 提供了庞大的镜像集合供使用。
Docker Machine	Docker Machine 是一个简化 Docker 安装的命令行工具，通过一个简单的命令行即可在相应的平台上安装 Docker,比如 VirtualBox、Digital Ocean、Microsoft Azure。

1.4 Docker 容器技术的应用场景

一般认为 Docker 技术有以下 8 个主要的应用场景，参见下图：



（1）、应用场景 1：简化配置

这是 Docker 公司宣传的 Docker 的主要使用场景。Docker 能将运行环境和配置放在代码中然后部署，同一个 Docker 的配置可以在不同的环境中使用，这样就降低了硬件要求和应用环境之间耦合度。

（2）、应用场景 2：代码流水线（Code Pipeline）管理

代码从开发者的机器到最终在生产环境上的部署，需要经过很多的中间环境。而每一个中间环境都有微小的差别，Docker 给应用提供了一个从开发到上线均一致的环境，让代码的流水线变得简单不少。

（3）、应用场景 3：提高开发效率

Docker 能提升开发者的开发效率。不同的开发环境中，Docker 都可以把两件事做好，一是可以在开发环境、生产环境之间直接迁移，二是可以让我们快速搭建开发环境。开发环境的机器通常内存比较小，之前使用虚拟的时候，我们经常需要为开发环境的机器加内存，而现在 Docker 可以轻易的让几十个服务在 Docker 中跑起来。

（4）、应用场景 4：隔离应用

有很多种原因会让我们选择在一个机器上运行不同的应用，Docker 非常适合在较低的成本下实现多种应用的隔离。

（5）、应用场景 5：整合服务器

Docker 隔离应用的能力使得 Docker 可以整合多个服务器以降低成本。由于没有操作系统的内存占用，以及能在多个实例之间共享没有使用的内存，Docker 可以比虚拟机提供更好的服务器整合解决方案。通常数据中心的服务器资源利用率只有 30%，通过使用 Docker 并进行有效的资源分配可以大幅提高服务器资源的利用率。

（6）、应用场景 6：调试能力

Docker 提供了很多的工具，包括可以为容器设置检查点、设置版本和查看两个容器之间的差别，这些特性可以帮助调试 Bug。

（7）、应用场景 7：多租户环境

另外一个 Docker 的使用场景是在多租户的应用中，它可以避免关键应用的重写。我们一个特别的关于这个场景的例子是为物联网的应用开发一个快速、易用的多租户环境。这种多租户的基本代码非常复杂，很难处理，重新规划这样一个应用不但消耗时间，也浪费金钱。

使用 Docker，可以为每一个租户的应用层的多个实例创建隔离的环境，这不仅简单而且成本低廉，当然这一切得益于 Docker 环境的启动速度和其高效的 diff 命令。

（8）、应用场景 8：快速部署

在虚拟机之前，购入部署新的硬件资源需要消耗几天的时间。虚拟化技术（Virtualization）将这个时间缩短到了分钟级别。而 Docker 通过为进程仅仅创建一个容器而无需启动一个操作系统，再次将这个过程缩短到了秒级。这正是 Google 和 Facebook 都看重的特性。我们可以创建销毁 Docker 容器而无需担心重新启动带来的开销。

第二部分 核心概念与安装配置

本部分首先介绍 Docker 的三大核心概念。

- 镜像(Image)
- 容器(Container)
- 仓库(Repository)

只有理解了这三个核心概念，才能顺利地理解 Docker 容器的整个生命周期。随后将介绍如何在常见的操作系统平台上安装 Docker，包括 Redhat Linux、Windows、Centos 等主流操作系统平台。

2.1 核心概念

Docker 的大部分操作都围绕着它的三大核心概念——镜像、容器和仓库而展开。因此，准确把握这三大核心概念对于掌握 Docker 技术尤为重要。

1．Docker 镜像

Docker 镜像类似于虚拟机镜像，可以将它理解为一个只读的模板。例如，一个镜像可以包含一个基本的操作系统环境，里面仅安装了 Apache 应用程序(或用户需要的其他软件)。可以把它称为一个 Apache 镜像。

镜像是创建 Docker 容器的基础。通过版本管理和增量的文件系统，Docker 提供了一套十分简单的机制来创建和更新现有的镜像，用户甚至可以从网上下载一个已经做好的应用镜像，并直接使用。

2．Docker 容器

Docker 容器类似于一个轻量级的沙箱，Docker 利用容器来运行和隔离应用。容器是从镜像创建的应用运行实例。可以将其启动、开始、停止、删除，而这些容器都是彼此相互隔离的、互不可见的。

可以把容器看做是一个简易版的 Linux 系统环境(包括 root 用户权限、进程空间、用户空间和网络空间等)以及运行在其中的应用程序打包而成的盒子。

3．Docker 仓库

Docker 仓库类似于代码仓库，它是 Docker 集中存放镜像文件的场所。

有时候会看到有资料将 Docker 仓库和仓库注册服务器(Registry)混为一谈，并不严格区分。实际上，仓库注册服务器是存放仓库的地方，其上往往存放着多个仓库。每个仓库集中存放某一类镜像，往往包括多个镜像文件，通过不同的标签(tag)来进行区分。例如存放 ubuntu 操作系统镜像的仓库称为 ubuntu 仓库，其中可能包括 14.04、12.04 等不同版本的镜像。

根据所存储的镜像公开分享与否，Docker 仓库可以分为公开仓库(Public)和私有仓库(Private)两种形式。目前，最大的公开仓库是官方提供的 Docker Hub，其中存放了数量庞大的镜像供用户下载。国内不少云服务提供商(如时速云、阿里云等)也提供了仓库的本地源，可以提供稳定的国内访问。

当然，用户如果不希望公开分享自己的镜像文件，Docker 也支持用户在本地网络内创建一个只能自己访问的私有仓库。当用户创建了自己的镜像之后就可以使用 push 命令将它上传到指定的公有或者私有仓库。这样用户下次在另外一台机器上使用该镜像时，只需要将其从仓

库上 pull 下来就可以了。

可以看出，Docker 利用仓库管理镜像的设计理念与 Git 非常相似，实际上在理念设计上借鉴了 Git 的很多优秀思想。

2.2 安装 Docker

Docker 在主流的操作系统和云平台上都可以使用，包括 Linux 操作系统(如 ubuntu、Debian、CentOS、Redhat 等)、MacOS 操作系统和 Windows 操作系统，以及 AWS 等云平台。

用户可以访问 Docker 官网的 Get Docker(https://www.docker.com/products/overview)页面，查看获取 Docker 的方式，以及 Docker 支持的平台类型，如图 2-2 所示。

在 Get Docker 页面，我们可以看到目前 Docker 支持 Docker Platform、Docker Hub、Docker
Cloud 和 Docker DataCenter。

- Docker Platform：支持在桌面系统或云平台安装 Docker；
- DockerHub：官方提供的云托管服务，可以提供公有或私有的镜像仓库；
- DockerCloud：官方提供的容器云服务，可以完成容器的部署与管理，可以完整地支持容器化项目，还有 CI、CD 功能；
- Docker DataCenter：提供企业级的简单安全弹性的容器集群编排和管理。

我们推荐尽量使用 Linux 操作系统来运行 Docker，因为目前 Linux 操作系统对 Docker 的支持是原生的，使用体验最好

2.2.1 在 Red Hat Enterprise Linux 上安装 Docker

以下是支持 Docker 的 RHEL 版本：

Red Hat Enterprise Linux 7 (64-bit)

Red Hat Enterprise Linux 6.5 (64-bit) 或更高版本

如果你的 RHEL 运行的是发行版内核。那就仅支持通过 extras 渠道或者 EPEL 包来安装 Docker。如果我们打算在非发行版本的内核上运行 Docker ，内核的改动可能会导致出错

1.Red Hat Enterprise Linux 7 安装 Docker

Red Hat Enterprise Linux 7 （64 位） 自带 Docker。我们可以在发行日志中找到概述和指

南。

Docker 包含在 extras 镜像源中，使用下面的方法可以安装 Docker:

启用 extras 镜像源:

```
$ sudo subscription-manager repos --enable=rhel-7-server-extras-rpms
```

安装 Docker :

```
$ sudo yum install docker
```

2.Red Hat Enterprise Linux 6.5 安装 Docker

需要在 64 位 的 RHEL 6.5 或更高的版本上来安装 Docker，Docker 工作需要特定的内核补丁，因此 RHEL 的内核版本应为 2.6.32-431 或者更高。

Docker 已经包含在 RHEL 的 EPEL 源中。该源是 Extra Packages for Enterprise Linux (EPEL) 的一个额外包，社区中正在努力创建和维护相关镜像。

首先，你需要安装 EPEL 镜像源，在 EPEL 中已经提供了 docker-io 包。

下一步，我们将要在我们的主机中安装 Docker,也就是 docker-io 包:

```
$ sudo yum -y install docker-io
```

更新 docker-io 包:

```
$ sudo yum -y update docker-io
```

现在 Docker 已经安装好了，我们来启动 docker 进程:

```
$ sudo service docker start
```

设置开机启动:

```
$ sudo chkconfig docker on
```

现在，让我们确认 Docker 是否正常工作：

```
$ sudo docker run -i -t fedora /bin/bash
```

现在 Docker 已经安装好了，让我们来启动 Docker 进程

```
$ sudo service docker start
```

如果我们想要开机启动 Docker ，我们需要执行如下的命令：

```
$ sudo chkconfig docker on
```

现在测试一下是否正常工作:

```
$ sudo docker run -i -t fedora /bin/bash
```

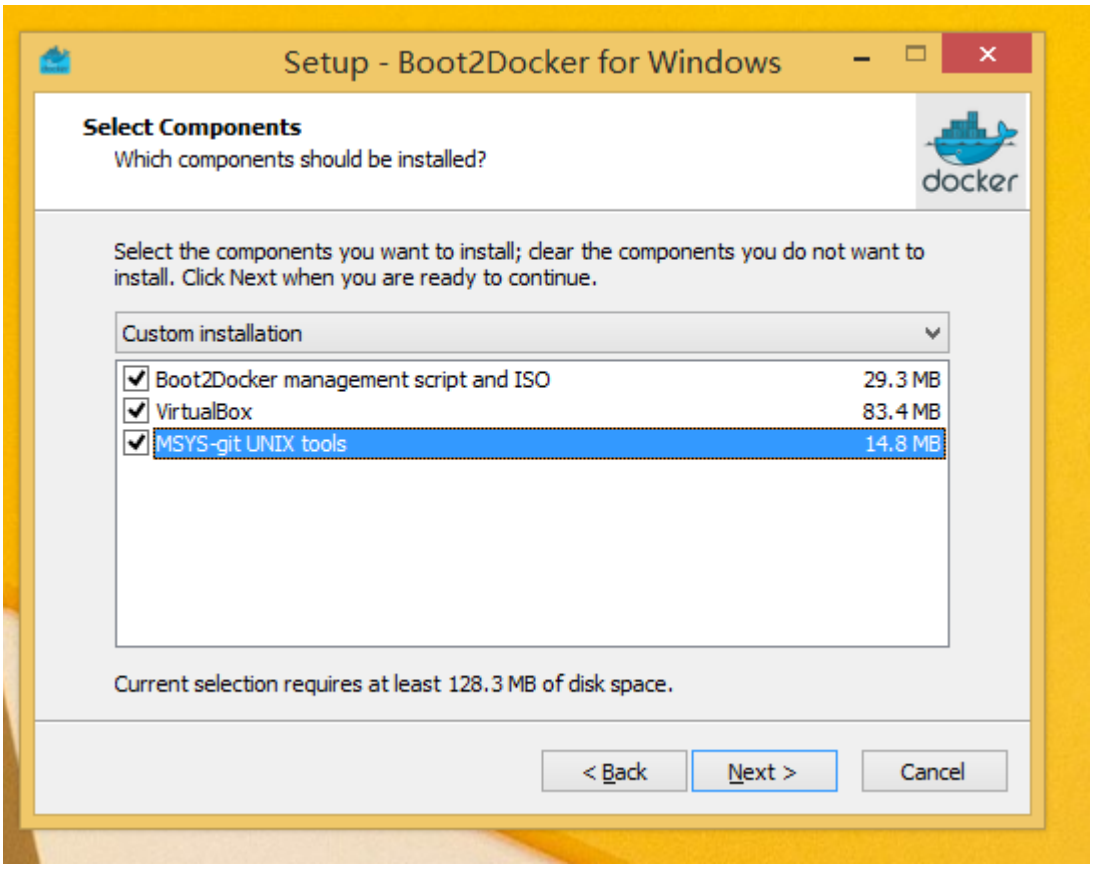
注意：如果运行的时候提示一个 Cannot start container 的错误，错误中提到了 SELINUX 或者 权限不足。我们需要更新 SELINUX 规则。可以使用 sudo yum upgrade selinux-policy 然后重启。

2.2.2 在 Windows 上安装 Docker

1. 安装

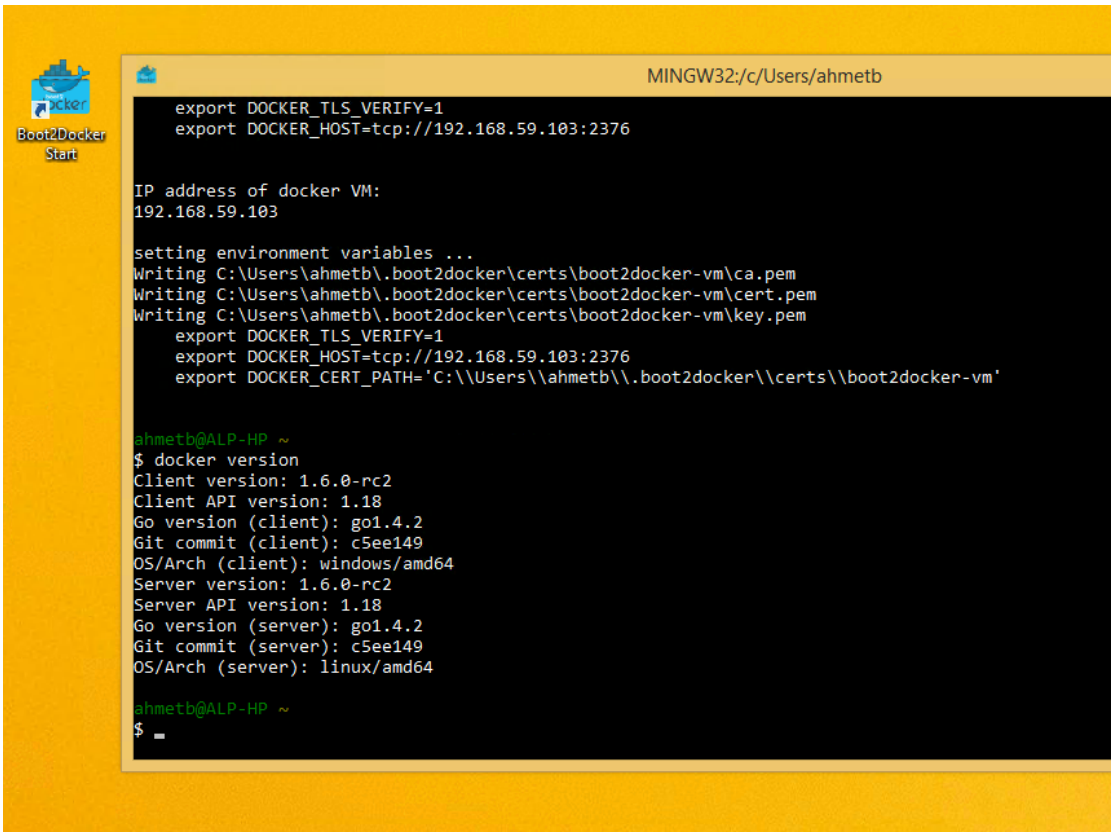
1)、下载最新版本的 Docker for Windows Installer

2)、运行安装文件，它将会安装 virtualbox、MSYS-git boot2docker Linux 镜像和 Boot2Docker 的管理工具。



3)、从桌面上或者 Program Files 中找到 Boot2Docker for Windows，运行 Boot2Docker Start 脚本。这个脚本会要求你输入 ssh 密钥密码 - 可以简单点（但是起码看起来比较安全），然后只需要按[Enter]按钮即可。

4)、Boot2Docker Start 将启动一个 Unix shell 来配置和管理运行在虚拟主机中的 Docker，运行 docker version 来查看它是否正常工作。



2.运行 Docker

注意：如果使用的是一个远程的 Docker 进程，像 Boot2docker ，就不需要像前边的文档实例中那样在输入 Docker 命令之前输入 sudo。

Boot2docker start 将会自动启动一个 shell 命令框并配置好环境变量，以便我们可以马上使用 Docker ：

让我们尝试运行 hello-world 例子。 运行：

$ docker run hello-world

这将会下载一个非常小的 hello-world 镜像，并且打印出 Hello from Docker 的信息。

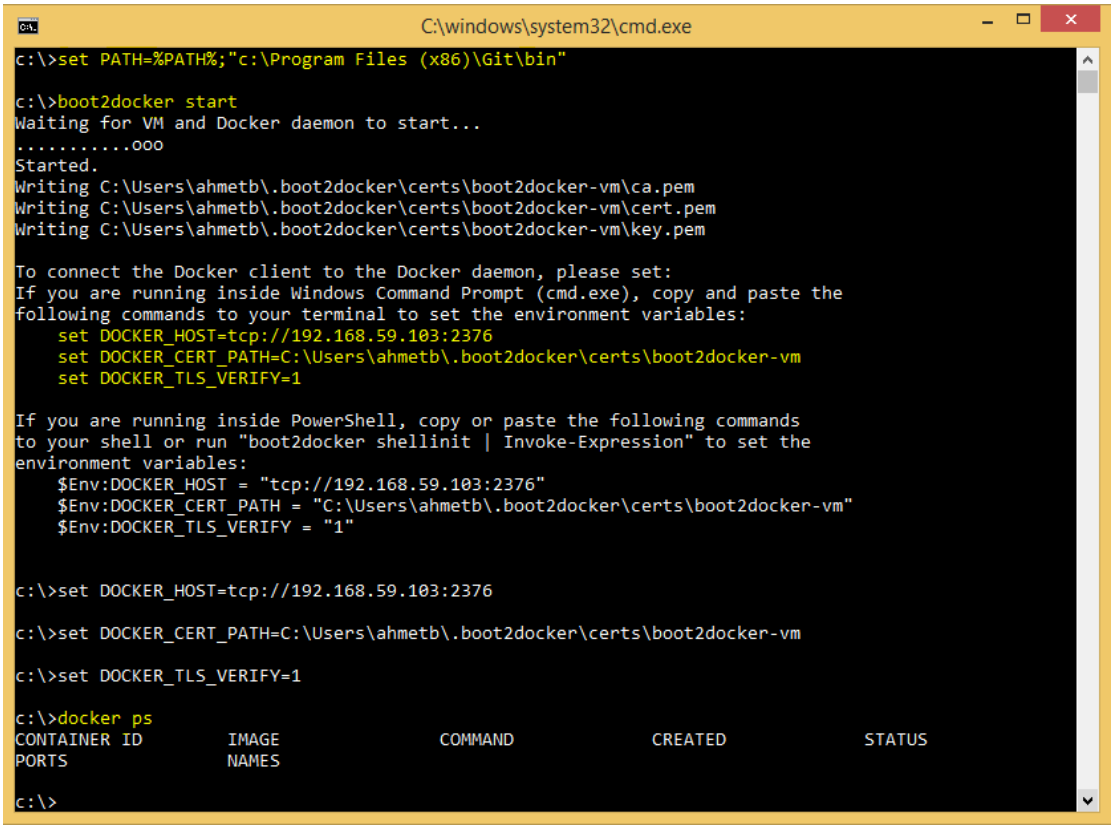
3.使用 Windows 的命令行(cmd.exe) 来管理运行 Docker

启动一个 Windows 命令行（cmd.exe），运行 Boot2docker 命令，这需要 Windows PATH 环境变量中包含了 ssh.exe。因此我们需要将安装的 Git 的 bin 目录 （其中包含了 ssh.exe）配置到我们的 %PATH% 环境变量中，运行如下命令：

set PATH=%PATH%;"c:\Program Files (x86)\Git\bin"

现在，我们可以运行 boot2docker start 命令来启动 Boot2docker 虚拟机。（如果有虚拟主机不存在的错误提示，需要运行 boot2docker init 命令）。复制上边的指令到 cmd.exe 来设

置 windows 控制台的环境变量，然后就可以运行 docker 命令了，譬如 docker ps ：

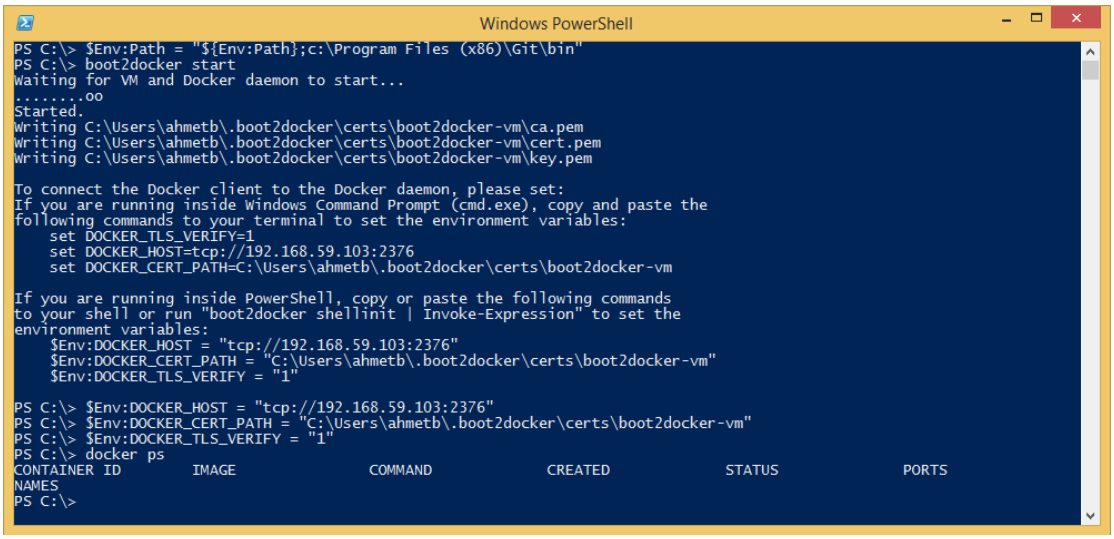


4.PowerShell 中使用 Docker

启动 PowerShell，需要将 ssh.exe 添加到 PATH 中。

$Env:Path = "${Env:Path};c:\Program Files (x86)\Git\bin"

之后，运行 boot2docker start 命令行，它会打印出 PowerShell 命令，这些命令是用来设置环境变量来连接运行在虚拟机中 Docker 的。运行这些命令，然后就可以运行 docker 命令了，譬如 docker ps ：



提示:可以使用 boot2docker shellinit | Invoke-Expression 来设置环境变量来代替复制粘贴 Powershell 命令。

2.2.3 在 CentOS 环境下安装 Docker

Docker 支持以下的 CentOS 版本：

CentOS 7 (64-bit)

CentOS 6.5 (64-bit)或更高的版本

前提条件

Docker 运行在 CentOS 7 上，要求系统为 64 位、系统内核版本为 3.10 以上。

Docker 运行在 CentOS-6.5 或更高的版本的 CentOS 上，要求系统为 64 位、系统内核版本为 2.6.32-431 或者更高版本。

使用 yum 安装（CentOS 7 下）

Docker 要求 CentOS 系统的内核版本高于 3.10 ，查看本页面的前提条件来验证你的 CentOS 版本是否支持 Docker 。

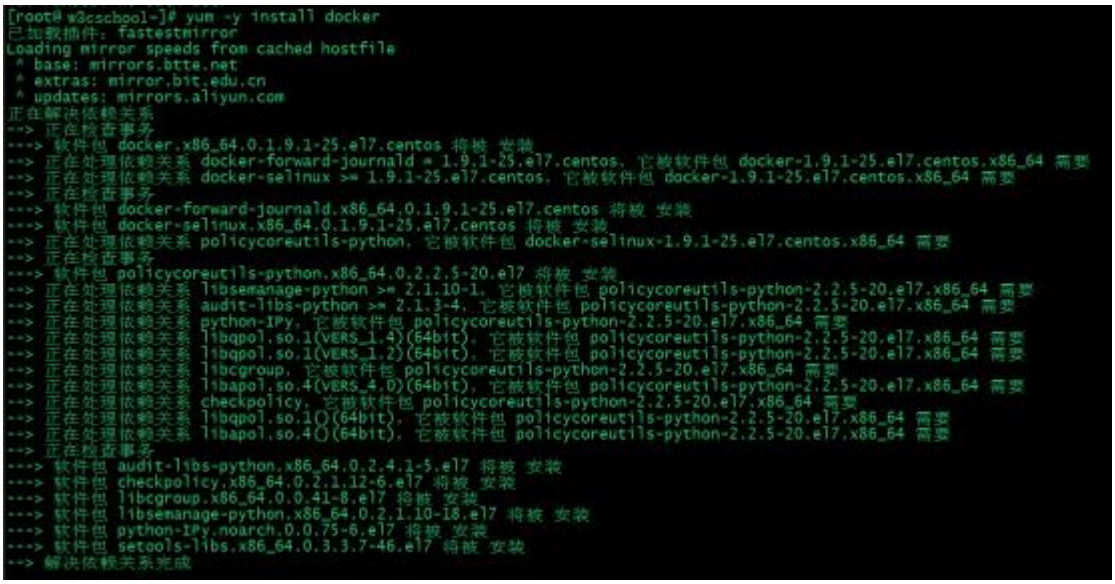
通过 uname -r 命令查看你当前的内核版本

uname -r 3.10.0-327.el7.x86_64

安装 Docker

Docker 软件包和依赖包已经包含在默认的 CentOS-Extras 软件源里，安装命令如下：

yum -y install docker



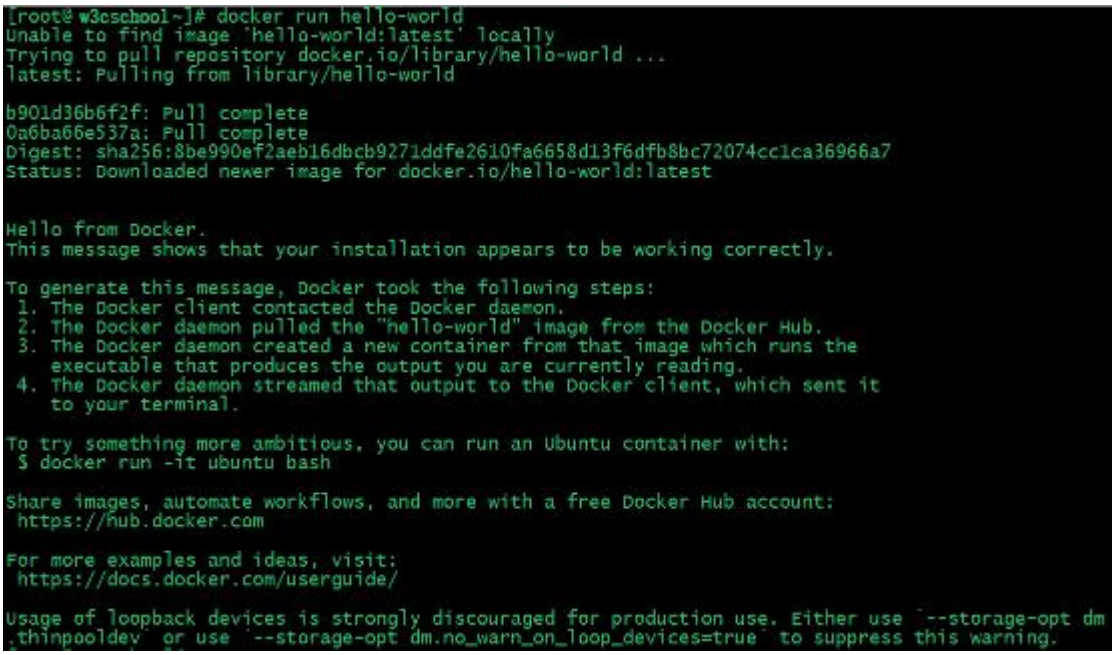
安装完成。

启动 Docker 后台服务

service docker start

测试运行 hello-world

docker run hello-world



由于本地没有 hello-world 这个镜像，所以会下载一个 hello-world 的镜像，并在容器内运行。

使用脚本安装 Docker

1、使用 sudo 或 root 权限登录 Centos。

2、确保 yum 包更新到最新。

$ sudo yum update

3、执行 Docker 安装脚本。

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

执行这个脚本会添加 docker.repo 源并安装 Docker。

4、启动 Docker 进程。

$ sudo service docker start

5、验证 docker 是否安装成功并在容器中执行一个测试的镜像。

$ sudo docker run hello-world

到此，docker 在 CentOS 系统的安装完成。

第三部分　使用 Docker 镜像

镜像(image)是 Docker 三大核心概念中最为重要的，自 Docker 诞生之日起“镜像”就是相关社区最为热门的关键词。

Docker 运行容器前需要本地存在对应的镜像，如果镜像没保存在本地，Docker 会尝试先从默认镜像仓库下载(默认使用：Docker Hub 公共注册服务器中的仓库)，用户也可以通过配置，使用自定义的镜像仓库。

本部分将介绍围绕镜像这一核心概念的具体操作，包括如何使用 pull 命令从 Docker Hub 仓库中下载镜像到本地，如何查看本地已有的镜像信息和管理镜像标签，如何在远端仓库使用 search 命令进行搜索和过滤，如何删除镜像标签和镜像文件，如何创建用户定制的镜像并且保存为外部文件。最后，还介绍如何往 Docker Hub 仓库中推送自己的镜像。

3.1 获取镜像

镜像是运行容器的前提，官方的 Docker Hub 网站已经提供了数十万个镜像供大家开放下载。

可以使用 docker pull 命令直接从 Docker Hub 镜像源来下载镜像。该命令的格式为 docker pull NAME[:TAG]。其中，NAME 是镜像仓库的名称(用来区分镜像)，TAG 是镜像的标签(往往用来表示版本信息)。通常情况下，描述一个镜像需要包括‘名称+标签’信息。

例如，获取一个 Ubuntu 14.04 系统的基础镜像可以使用如下的命令：

```
$docker pull ubuntu：14.04
14.04：Pulling from library/ubuntu
6c953ac5d795: Pull complete
3eed5ff2 0a90: Pull complete
f8419ea7Cib5: Pull complete
51900bc9e720: Pull complete
a3ed95caeb02: Pull complete
Digest: sha2 56: 97421885f3da3b23f52eeddcaa9f8f91172a8ac3cd5d3cd40b5lc7aad09f66cc
Status: Downloaded newer image for ubuntu：14.04
```

对于 Docker 镜像来说，如果不显式指定 TAG，则默认会选择 latest 标签，这会下载仓库中最新版本的镜像。

下面的例子将从 Docker Hub 的 Ubuntu 仓库下载一个最新的 Ubuntu 操作系统的镜像。

```
$docker pull ubuntu
```

Using default tag: latest
latest: Pulling from library/ubuntu
5ba4 f3 0e5bea: Pull complete
9d7d19C9dc56: Pull complete
ac6ad7efdof9: Pull complete
e749la747824: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256: 46fb5d0 01b88ad904c5c732b086b596b92cfb4a4840a3abdoe35dbb6870585e4
Status: Downloaded newer image for Ubuntu: latest

该命令实际上下载的就是 ubuntu: latest 镜像。

下载过程中可以看出，镜像文件一般由若干层(layer)组成，6c953ac5d795 这样的串是层的唯一 id(实际上完整的 id 包括 256 比特，由 64 个十六进制字符组成)。使用 docker pull 命令下载时会获取并输出镜像的各层信息。当不同的镜像包括相同的层时，本地仅存储层的一份内容，减小了需要的存储空间。

我们可能会想到，在使用不同的镜像仓库服务器的情况下，可能会出现按镜像重名的情况。

严格地讲，镜像的仓库名称中还应该添加仓库地址（即 registry，注册服务器）作为前缀，只是我们默认使用的是 Docker Hub 服务，该前缀可以忽略。

例如，docker pull Ubuntu:14.04 命令相当于 docker pull registry.hub.docker.com/Ubuntu:14.04 命令，即从默认的注册服务器 Docker Hub Registry 中的 ubuntu 仓库来下载标记为 14.04 的镜像。

如果从非官方的仓库下载，则需要在仓库名称前指定完整的仓库地址。例如从网易蜂巢的镜像源来下载 ubuntu:14.04 镜像，可以使用如下命令，此时下载的镜像名称为 hub.c.163.com/public/Ubuntu:14.04：

$ docker pull hub.c.163.com/public/Ubuntu:14.04

pull 子命令支持的选项主要包括：

-a,--all- tags=true| false：是否获取仓库中的所有镜像，默认为否。

下载镜像到本地后，即可随时使用该镜像了，例如利用该镜像创建一个容器，在其中运行 bash 应用，执行 ping localhost 命令：

$ docker run -it ubuntu:14.04 bash

root@9c74026dfl2a:/#ping 10calhost
PING localhost(127.0.0.1)56(84)bytes of data.
64 bytes from localhost(127.0.0.1):icmp_seq=1 ttl=64 time=0.058 ms
64 bytes from localhost(127.0.0.1):icmp_seq=2 ttl=64 time=0.023 ms

```
64 bytes from localhost(127.0.0.1):icmp_seq=3 ttl=64 time=0.018 ms
^C
————localhost Ping statistics————
3 packets transmitted，3 received，0%packet loss，time 1999ms
rtt min/avg/max/mdev=0.018/0.033/0.058/0.017 ms
root@9c7402 6dfl2a:/# exit
exit
```

3.2 查看镜像信息

1. 使用 images 命令列出镜像

使用 docker images 命令可以列出本地主机上已有镜像的基本信息。

例如，下面的命令列出了上一小节中下载的镜像信息：

```
$docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	16.04	2fa927b5cdd3	2 weeks ago	122 MB
ubuntu	latest	2 fa927b5cdd3	2 weeks ago	122 MB
ubuntu	14.04	8fibd2ibd25c	2 weeks ago	188 MB

在列出的信息中，可以看到以下几个字段信息。

- 来自于哪个仓库，比如 ubuntu 仓库用来保存 ubuntu 系列的基础镜像；
- 镜像的标签信息，比如 14.04、latest 用来标注不同的版本信息。标签只是标记，并不能标识镜像内容；
- 镜像的 ID(唯一标识镜像)，如 ubuntu:latest 和 ubuntu:16.04 镜像的 ID 都是 2fa927b5cdd3，说明它们目前实际上指向同一个镜像；
- 创建时间，说明镜像最后的更新时间；
- 镜像大小，优秀的镜像往往体积都较小。

其中镜像的 ID 信息十分重要，它唯一标识了镜像。在使用镜像 ID 的时候，一般可以使用该 ID 的前若干个字符组成的可区分串来替代完整的 ID。

TAG 信息用来标记来自同一个仓库的不同镜像。例如 ubunm 仓库中有多个镜像，通过 TAG 信息来区分发行版本，包括 10.04、12.04、12.10、13.04、14.04、16.04 等标签：

镜像大小信息只是表示该镜像的逻辑体积大小，实际上由于相同的镜像层本地只会存储一份，物理上占用的存储空间会小于各镜像的逻辑体积之和。

images 子命令主要支持如下选项，用户可以自行进行尝试。

- -a，--all=true | false：列出所有的镜像文件(包括临时文件)，默认为否；

- --digests=true I false：列出镜像的数字摘要值，默认为否；
- -f，---filter=[ ]：过滤列出的镜像，如 dangling=true 只显示没有被使用的镜像；

也可指定带有特定标注的镜像等；

- --format="TEMPLATE"：控制输出格式，如.ID 代表 ID 信息，Repository 代表仓库信息等；
- --no-trunc=true l false：对输出结果中太长的部分是否进行截断，如镜像的 ID 信息，默认为是；
- -q，--quiet=true l false：仅输出 ID 信息，默认为否。

其中，对输出结果进行控制的选项如-f，--filter=[ ]、--no-trunc=true | false、-q，--quiet=true I false 等，大部分子命令都支持。

更多子命令选项还可以通过 man docker-images 来查看。

2．使用 tag 命令添加镜像标签

为了方便在后续工作中使用特定镜像，还可以使用 docker tag 命令来为本地镜像任意添加新的标签。例如添加一个新的 myubuntu:latest 镜像标签：

```
$ docker tag ubuntu:latest myubuntu:latest
```

再次使用 docker images 列出本地主机上镜像信息，可以看到多了一个拥有 myubuntu:latest 标签的镜像。之后，用户就可以直接使用 myubuntu:latest 来表示这个镜像了。

我们可能注意到，这些 myubuntu:latest 镜像的 ID 跟 ubuntu:latest 完全一致。它们实际上指向同一个镜像文件，只是别名不同而已。docker tag 命令添加的标签实际上起到了类似链接的作用。

3．使用 inspect 命令查看详细信息

使用 docker inspect 命令可以获取该镜像的详细信息，包括制作者、适应架构、各层的数字摘要等，可输入以下命令：

```
$docker inspect ubuntu:14.04
```

返回的是一个 JSON 格式的消息，如果我们只要其中一项内容时，可以使用参数-f 来指定，例如，获取镜像的 Architecture：

```
$docker inspect -f {{".Architecture"}}
    amd64
```

4．使用 history 命令查看镜像历史

既然镜像文件由多个层组成，那么怎么知道各个层的内容具体是什么呢?这时候可以使用 history 子命令，该命令将列出各层的创建信息。

例如，查看 ubuntu:14.04 镜像的创建过程，可以使用如下命令：

$docker history ubuntu:14.04

3.3 搜寻镜像

使用 docker search 命令可以搜索远端仓库中共享的镜像，默认搜索官方仓库中的镜像。用法为 docker search TERM，支持的参数主要包括：

- --automated=true I false：仅显示自动创建的镜像，默认为否；
- --no-trunc=true | false：输出信息不截断显示，默认为否；
- -s，--stars=X：指定仅显示评价为指定星级以上的镜像，默认为 0，即输出所有镜像。

例如，搜索所有自动创建的评价为 3+的带 nginx 关键字的镜像，如下所示：

$ docker search --automated -s 3 nginx

可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、星级(表示该镜像的受欢迎程度)、是否官方创建、是否自动创建等。

默认的输出结果将按照星级评价进行排序。

3.4 删除镜像

1．使用标签删除镜像

使用 docker rmi 命令可以删除镜像，命令格式为 docker rmi IMAGE [IMAGE…]，其中 IMAGE 可以为标签或 ID。

例如，要删除掉 myubuntu:latest 镜像，可以使用如下命令：

$ docker rmi myubuntu:latest

Untagged：myubuntu:latest

我们可能会担心，本地的 ubuntu:latest 镜像是否会受此命令的影响。无需担心，当同

一个镜像拥有多个标签的时候，docker rmi 命令只是删除该镜像多个标签中的指定标签而已，并不影响镜像文件。因此上述操作相当于只是删除了镜像 2fa927b5cdd3 的一个标签而已。

为保险起见，再次查看本地的镜像，发现 ubuntu:latest 镜像(准确地说是 2fa927b5cdd3 镜像)仍然存在。但当镜像只剩下一个标签的时候就要小心了，此时再使用 docker rmi 命令会彻底删除镜像。

例如删除标签为 ubuntu:14.04 的镜像，由于该镜像没有额外的标签指向它，执行 docker rmi 命令，它会删除这个镜像文件的所有层：

```
$ docker rmi ubuntu:14.04
```

2. 使用镜像 ID 删除镜像

当使用 docker rmi 命令，并且后面跟上镜像的 ID(也可以是能进行区分的部分 ID 串前缀)时，会先尝试删除所有指向该镜像的标签，然后删除该镜像文件本身。

注意，当有该镜像创建的容器存在时，镜像文件默认是无法被删除的，例如，先利用 ubuntu:14.04 镜像创建一个简单的容器来输出一段话：

```
$ docker run Ubuntu:14.04 echo 'hello! I am here!'
hello!I am here!
```

使用 docker ps -a 命令可以看到本机上存在的所有容器：

```
$ docker ps -a
```

可以看到，后台存在一个退出状态的容器，是刚基于 ubuntu:14.04 镜像创建的。

试图删除该镜像，Docker 会提示有容器正在运行，无法删除：

```
$ docker rmi Ubuntu:14.04
Error response from daemon: conflict: unable to remove repository reference "Ubuntu: 14.04" (must force)— container a21c0840213e is using it's referenced image 8f1bd21bd25c
```

如果要想强行删除镜像，可以使用 -f 参数。

```
$ docker rmi -f ubuntu:14.04
```

注意，通常并不推荐使用-f 参数来强制删除一个存在容器依赖的镜像。正确的做法是，先删除依赖该镜像的所有容器，再来删除镜像。首先删除容器 a21c0840213e：

$docker rm a21c0840213e

再使用 ID 来删除镜像，此时会正常打印出删除的各层信息：

$docker rmi 8f1bd21bd25c

3.5 创建镜像

创建镜像的方法主要有三种：基于已有镜像的容器创建、基于本地模板导入、基于 Dockerfile 创建。

1．基于已有镜像的容器创建

该方法主要是使用 docker commit 命令。命令格式为 docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]，主要选项包括：

- -a，--author=""：作者信息；
- -c，--change=[ ]：提交的时候执行 Dockerfile 指令，包括 CMD | ENTRYPOINT | ENV 1 EXPOSE | LABEL | ONBUILD | USER | VOLUME | WORKDIR 等；
- -m，--message=""：提交消息；-
- -p，--pause=true：提交时暂停容器运行。

下面将演示如何使用该命令创建一个新镜像。首先，启动一个镜像，并在其中进行修改操作，例如创建一个 test 文件，之后退出：

$ docker run -it Ubuntu:14.04/bin/bash

root@a925cb40b3f0:/# touch test

root@a925Cb40b3f0:/# exit

记住容器的 ID 为 a925cb40b3f0。

此时该容器跟原 ubuntu:14.04 镜像相比，已经发生了改变，可以使用 docker commit 命令来提交为一个新的镜像。提交时可以使用 ID 或名称来指定容器：

$ docker commit -m "Added a new file" -a "Docker Newbee" a925cb40b3f0 test:0.1

9e9C814023bcffc3e67e892a235afe61b02f66a947d2747f724bd317dda02f27

顺利的话，会返回新创建的镜像的 ID 信息，例如 9e9C814023bcffc3e67e892a235afe61b02f66a947d2747f724bd317dda02f27。此时查看本地镜像列表，会发现新创建的镜像已经存在了。

$docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

test 0.1 9e9c814023bc 4 seconds ago 188 MB

2．基于本地模板导入

用户也可以直接从一个操作系统模板文件导入一个镜像，主要使用 docker import 命令。命令格式为 docker import [OPTIONS] file |URL|-[REPOSITORY [:TAG]]。

要直接导入一个镜像，可以使用 OpenVZ 提供的模板来创建，或者用其他已导出的镜像模板来创建。OPENVZ 模板的下载地址为 http：//openvz.org/Download/templates/precreated。

例如，下载了 ubuntu -14.04 的模板压缩包，之后使用以下命令导入：

$cat Ubuntu -14.04 -x86_64 -minimal.tar.gz | docker import -ubuntu:14.04

然后查看新导入的镜像，会发现它已经在本地存在了：

$docker images

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04	05ac7cob9383	17 seconds ago	215.5 MB

3.6 存出和载入镜像

用户可以使用 docker save 和 docker load 命令来存出和载入镜像。

1．存出镜像

如果要导出镜像到本地文件，可以使用 docker save 命令。例如，导出本地的 ubuntu:14. 04 镜像为文件 ubuntu_14.04.tar，如下所示：

$docker images

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04	8f1bd21bd25c	2 weeks ago	188 MB

$docker save -o ubuntu_14.04.tar ubuntu:14.04

之后，用户就可以通过复制 ubuntu_14.04.tar 文件将该镜像分享给他人。

2．载入镜像

可以使用 docker load 将导出的 tar 文件再导入到本地镜像库，例如从文件 ubuntu_14.04.tar 导入镜像到本地镜像列表，如下所示：

$docker load --input ubuntu_14.04.tar

或：

$docker load < ubuntu_14.04.tar

这将导入镜像及其相关的元数据信息(包括标签等)。导入成功后，可以使用 docker images 命令进行查看。

3.7 上传镜像

可以使用 docker push 命令上传镜像到仓库，默认上传到 Docker Hub 官方仓库(需要登录)。命令格式为：

docker push NAME [:TAG] l [REGISTRY_HOST [:REGISTRY_PORT]/] NAME [:TAG]

用户在 Docker Hub 网站注册后可以上传自制的镜像。例如用户 user 上传本地的 test：latest 镜像. 可以先添加新的标签 user/test:latest，然后用 docker push 命令上传镜像：

$docker tag test:latest user/test:latest

$docker push user/test:latest

第一次上传时，会提示输入登录信息或进行注册。

第四部分　操作 Docker 容器

容器是 Docker 的另一个核心概念。简单来说，容器是镜像的一个运行实例。所不同的是，镜像是静态的只读文件，而容器带有运行时需要的可写文件层。如果认为虚拟机是模拟运行的一整套操作系统(包括内核、应用运行态环境和其他系统环境)和跑在上面的应用，那么 Docker 容器就是独立运行的一个(或一组)应用，以及它们必需的运行环境。容器是直接提供应用服务的组件，也是 Docker 实现快速启停和高效服务性能的基础。

在生产环境中，因为容器自身的轻量级特性，我们推荐使用容器时在一组容器前引入 HA(High Availability，高可靠性)机制。例如使用 HAProxy 工具来代理容器访问，这样在容器出现故障时，可以快速切换到功能正常的容器。此外，建议通过指定合适的容器重启策略，来自动重启退出的容器。

本部分具体介绍围绕容器的重要操作，包括创建一个容器、启动容器、终止一个容器、进入容器内执行操作、删除容器和通过导入导出容器来实现容器迁移等。

4.1 创建容器

从现在开始，可以忘掉虚拟机。对容器进行操作就跟直接操作应用一样简单、快速。Docker 容器实在太轻量级了，用户可以随时创建或删除容器。

1．新建容器

可以使用 docker create 命令新建一个容器，例如：

```
$docker create -it ubuntu:latest
af8f4f922dafee22c8fe6cd2aelldl6e25087d6lflblfa55b36e94db7ef45178
$docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
af8f4f922daf Ubuntu:latest "/bin/bash" 17 seconds ago Created silly_euler
```

使用 docker create 命令新建的容器处于停止状态，可以使用 docker start 命令来启动它。

create 命令和后续的 run 命令支持的选项都十分复杂，与容器运行模式相关、与容器和环境配置相关、与容器资源限制和安全保护相关。

2．启动容器

使用 docker start 命令来启动一个已经创建的容器，例如启动刚创建的 ubuntu 容器：

```
$ docker start af
af
```

此时，通过 docker ps 命令可以查看一个运行中的容器：

```
$docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
af8f4f922daf Ubuntu:latest "/bin/bash" 2 minutes ago Up 7 seconds silly euler
```

3．新建并启动容器

除了创建容器后通过 start 命令来启动，也可以直接新建并启动容器。所需要的命令主要为 docker run，等价于先执行 docker create 命令，再执行 docker start 命令。

例如，下面的命令输出一个 "Hello World"，之后容器自动终止：

```
$ docker run ubuntu /bin/echo 'Hello world'
Hello world
```

这跟在本地直接执行/bin/echo 'Hello world' 几乎感觉不出任何区别。

当利用 docker run 来创建并启动容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载；
- 利用镜像创建一个容器，并启动该容器；
- 分配一个文件系统给容器，并在只读的镜像层外面挂载一层可读写层；
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中；
- 从网桥的地址池配置一个 IP 地址给容器；
- 执行用户指定的应用程序；
- 执行完毕后容器被自动终止。

下面的命令启动一个 bash 终端，允许用户进行交互：

```
$ docker run -it Ubuntu:14.04/bin/bash
root@af8bae53bdd3:/#
```

其中，-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。更多的命令选项可以通过 man docker -run 命令来查看。

在交互模式下，用户可以通过所创建的终端来输入命令，例如：

```
root@af8bae53bdd3:/#pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib iib64 media mnt opt proc root run sbin srv sys tmp usr var
root@af8bae53bdd3:/# ps
    PID TTY          TIME             CMD
    1     ?          00:00:00          bash
    1l     ?          00:00:00          ps
```

在容器内用 ps 命令查看进程，可以看到，只运行了 bash 应用，并没有运行其他无关的进程。用户可以按 Ctrl+d 或输入 exit 命令来退出容器：

```
root@af8bae53bdd3:/# exit
exit
```

对于所创建的 bash 容器，当使用 exit 命令退出之后，容器就自动处于退出(Exited)状态了。这是因为对 Docker 容器来说，当运行的应用退出后，容器也就没有继续运行的必要了。

某些时候，执行 docker run 会出错，因为命令无法正常执行容器会直接退出，此时可以查看退出的错误代码。

默认情况下，常见错误代码包括：

- 125：Dockerdaemon 执行出错，例如指定了不支持的 Docker 命令参数；
- 126：所指定命令无法执行，例如权限出错；
- 127：容器内命令无法找到。

命令执行后出错，会默认返回错误码。

4．守护态运行

更多的时候，需要让 Docker 容器在后台以守护态(Daemonized)形式运行。此时，可以通过添加 -d 参数来实现。

例如下面的命令会在后台运行容器：

```
$docker run -d ubuntu  /bin/sh -c "while true; do echo hello world; sleep 1; done"
ce554267d7a4c34eefc92C551705idc37b918b588736d0823e4C846596b04d83
```

容器启动后会返回一个唯一的 id，也可以通过 docker ps 命令来查看容器信息：

```
$docker ps
CONTAINER ID    IMAGE           COMMAND   CREATED   STATUS      PORTS     NAMES
ce554267d7a4    ubuntu:latest   "/bin/sh -c   'while t      About a minute ago  Up About a mlnute
determined_pik
```

此时，要获取容器的输出信息，可以如下使用 docker logs 命令：

```
$docker logs ce5
helio world
helio world
hello world
```

4.2 终止容器

可以使用 docker stop 来终止一个运行中的容器。该命令的格式为 docker stop [-t|--time[=10]] [CONTAINER…]。

首先向容器发送 SIGTERM 信号，等待一段超时时间(默认为 10 秒)后，再发送 SIGKILL 信号来终止容器：

```
$docker stop ce5
ce5
```

此外，当 Docker 容器中指定的应用终结时，容器也会自动终止。例如对于上一节中只启动了一个终端的容器，用户通过 exit 命令或 Ctrl+d 来退出终端时，所创建的容器立刻终止，处

于 stopped 状态。可以用 docker ps -qa 命令看到所有容器的 ID。例如：

```
$docker ps -qa
ce554267d7a4
d58050081fe3
e812 617b41f6
```

处于终止状态的容器，可以通过 docker start 命令来重新启动：

```
$ docker start ce5
```

此外，docker restart 命令会将一个运行态的容器先终止，然后再重新启动它：

```
$docker restart ce5
```

4.3 进入容器

Docker 从 1. 3. 0 版本起提供了一个方便的 exec 命令，可以在容器内直接执行任意命令。该命令的基本格式为：

docker exec [-d|--detach] [--detach-keys[=[ ]]] [-i|--interactive] [--privileged] [-t |--tty] [-u |--user[=USER]] CONTAINER COMMAND [ARG…]。

比较重要的参数有：

- -i,--interactive=true | false：打开标准输入接受用户输入命令，默认为 false；
- --privileged=true | false：是否给执行命令以高权限，默认为 false；
- -t,--tty= true | false：分配伪终端，默认为 false；
- -u,--user=" "：执行命令的用户名或 ID。

例如进入到刚创建的容器中，并启动一个 bash：

```
$docker exec -it 243c32535da7  /bin/bash
root@243c32535da7:/#
```

可以看到，一个 bash 终端打开了，在不影响容器内其他应用的前提下，用户可以很容易与容器进行交互。

4.4 删除容器

可以使用 docker rm 命令来删除处于终止或退出状态的容器，命令格式为 docker rm [-f

|--force] [-l |--link] [-v |--volumes] CONTAINER [CONTAINER…]。

主要支持的选项包括：

- -f，--force=false：是否强行终止并删除一个运行中的容器；
- -l，--link=false：删除容器的连接，但保留容器；
- -v，--volumes=false：删除容器挂载的数据卷。

例如，查看处于终止状态的容器，并删除：

```
$docker rm ce554267d7a4
Ce554267d7a4
```

默认情况下，docker rm 命令只能删除处于终止或退出状态的容器，并不能删除还处于运行状态的容器。

如果要直接删除一个运行中的容器，可以添加 -f 参数。Docker 会先发送 SIGKILL 信号给容器，终止其中的应用，之后强行删除，如下所示：

```
$ docker rm -f 2ae
2ae
```

4.5 导入和导出容器

某些时候，需要将容器从一个系统迁移到另外一个系统，此时可以使用 Docker 的导入和导出功能。这也是 Docker 自身提供的一个重要特性。

1. 导出容器

导出容器是指导出一个已经创建的容器到一个文件，不管此时这个容器是否处于运行状态，可以使用 docker export 命令，该命令的格式为 docker export [-o] --output[=" "]] CONTAINER。其中，可以通过-o 选项来指定导出的 tar 文件名，也可以直接通过重定向来实现。

示例：分别导出 ce554267d7a4 容器和 e812617b41f6 容器到文件 test_for_run.tar 文件和 test_for_stop.tar 文件：

```
$ docker export  -o test_for_run.tar ce5
$ls
test_for_run.tar
$ docker export e81 > test_for_stop.tar
$ls
```

test_for_run.tar test_for_stop.tar

之后，可将导出的 tar 文件传输到其他机器上，然后再通过导人命令导人到系统中，从而实现容器的迁移。

2．导入容器

导出的文件又可以使用 docker import 命令导入变成镜像，该命令格式为：

docker import [-c I --change[=[ ]]] [-m|--message [=MESSAGE]] file | URL | - [REPOSITORY [:TAG]]

用户可以通过-c，--change=[ ]选项在导入的同时执行对容器进行修改的 Dockerfile 指令。

下面将导出的 test_for_run.tar 文件导入到系统中：

$ docker import test_for_run.tar - test/Ubuntu:v1.0

之前镜像章节中我们曾介绍过使用 docker load 命令来导入一个镜像文件，与 docker import 命令十分类似。

实际上，既可以使用 docker load 命令来导入镜像存储文件到本地镜像库，也可以使用 docker import 命令来导人一个容器快照到本地镜像库。

这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息(即仅保存容器当时的快照状态)，而镜像存储文件将保存完整记录，体积也更大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。

4.6 实现容器的网络端口映射

下面我们来实现 Docker 容器的网络端口映射。我们先创建了一个 python 应用的容器。

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

fce072cc88cee71b1cdceb57c2821d054a4a59f67da6b416fceb5593f059fc6d

另外，我们可以指定容器绑定的网络地址，比如绑定 127.0.0.1。

我们使用 -P 参数创建一个容器，使用 docker ps 来看到端口 5000 绑定主机端口 32768。

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS

```
NAMES
fce072cc88ce        training/webapp     "python app.py"     4 minutes ago       Up 4 minutes
0.0.0.0:32768->5000/tcp   grave_hopper
```

我们也可以使用 -p 标识来指定容器端口绑定到主机端口。

两种方式的区别是:

-P :是容器内部端口随机映射到主机的高端口。

-p : 是容器内部端口绑定到指定的主机端口。

```
$ docker run -d -p 5000:5000 training/webapp python app.py
33e4523d30aaf0258915c368e66e03b49535de0ef20317d3f639d40222ba6bc0
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
NAMES
33e4523d30aa        training/webapp     "python app.py"     About a minute ago   Up About a minute
0.0.0.0:5000->5000/tcp    berserk_bartik
fce072cc88ce        training/webapp     "python app.py"     8 minutes ago       Up 8 minutes
0.0.0.0:32768->5000/tcp   grave_hopper
```

另外，我们可以指定容器绑定的网络地址，比如绑定 127.0.0.1。

```
$ docker run -d -p 127.0.0.1:5001:5002 training/webapp python app.py
95c6ceef88ca3e71eaf303c2833fd6701d8d1b2572b5613b5a932dfdfe8a857c
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
NAMES
95c6ceef88ca        training/webapp     "python app.py"     6 seconds ago       Up 6 seconds
5000/tcp, 127.0.0.1:5001->5002/tcp   adoring_stonebraker
33e4523d30aa        training/webapp     "python app.py"     3 minutes ago       Up 3 minutes
0.0.0.0:5000->5000/tcp               berserk_bartik
fce072cc88ce        training/webapp     "python app.py"     10 minutes ago      Up 10 minutes
0.0.0.0:32768->5000/tcp              grave_hopper
```

这样我们就可以通过访问 127.0.0.1:5001 来访问容器的 5002 端口。

上面的例子中，默认都是绑定 tcp 端口，如果要绑定 UPD 端口，可以在端口后面加上 /udp。

```
$ docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
6779686f06f6204579c1d655dd8b2b31e8e809b245a97b2d3a8e35abe9dcd22a
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
NAMES
6779686f06f6        training/webapp     "python app.py"     4 seconds ago       Up 2 seconds
5000/tcp, 127.0.0.1:5000->5000/udp   drunk_visvesvaraya
95c6ceef88ca        training/webapp     "python app.py"     2 minutes ago       Up 2 minutes
5000/tcp, 127.0.0.1:5001->5002/tcp   adoring_stonebraker
33e4523d30aa        training/webapp     "python app.py"     5 minutes ago       Up 5 minutes
0.0.0.0:5000->5000/tcp               berserk_bartik
fce072cc88ce        training/webapp     "python app.py"     12 minutes ago      Up 12 minutes
0.0.0.0:32768->5000/tcp              grave_hopper
```

docker port 命令可以让我们快捷地查看端口的绑定情况。

```
$ docker port adoring_stonebraker 5002
127.0.0.1:5001
```

第五部分 Docker 容器实现 Web 服务与应用

5.1 Docker 容器实现 Apache 服务

方法一、通过 Dockerfile 构建

创建 Dockerfile

首先，创建目录 apache,用于存放后面的相关东西。

```
$ mkdir -p  ~/apache/www ~/apache/logs ~/apache/conf
```

www 目录将映射为 apache 容器配置的应用程序目录，logs 目录将映射为 apache 容器的日志

目录，conf 目录里的配置文件将映射为 apache 容器的配置文件，进入创建的 apache 目录，创建 Dockerfile，内容如下：

```
FROM debian:jessie

# add our user and group first to make sure their IDs get assigned consistently, regardless of whatever dependencies get added
#RUN groupadd -r www-data && useradd -r --create-home -g www-data www-data

ENV HTTPD_PREFIX /usr/local/apache2
ENV PATH $PATH:$HTTPD_PREFIX/bin
RUN mkdir -p "$HTTPD_PREFIX" \
    && chown www-data:www-data "$HTTPD_PREFIX"
WORKDIR $HTTPD_PREFIX

# install httpd runtime dependencies
# https://httpd.apache.org/docs/2.4/install.html#requirements
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        libapr1 \
        libaprutil1 \
        libaprutil1-ldap \
        libapr1-dev \
        libaprutil1-dev \
        libpcre++0 \
        libssl1.0.0 \
    && rm -r /var/lib/apt/lists/*

ENV HTTPD_VERSION 2.4.20
ENV HTTPD_BZ2_URL https://www.apache.org/dist/httpd/httpd-$HTTPD_VERSION.tar.bz2

RUN buildDeps=' \
        ca-certificates \
```

```
        curl \
        bzip2 \
        gcc \
        libpcre++-dev \
        libssl-dev \
        make \
    ' \
    set -x \
    && apt-get update \
    && apt-get install -y --no-install-recommends $buildDeps \
    && rm -r /var/lib/apt/lists/* \
    \
    && curl -fSL "$HTTPD_BZ2_URL" -o httpd.tar.bz2 \
    && curl -fSL "$HTTPD_BZ2_URL.asc" -o httpd.tar.bz2.asc \
# see https://httpd.apache.org/download.cgi#verify
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys A93D62ECC3C8EA12DB220EC934EA76E6791485A8 \
    && gpg --batch --verify httpd.tar.bz2.asc httpd.tar.bz2 \
    && rm -r "$GNUPGHOME" httpd.tar.bz2.asc \
    \
    && mkdir -p src \
    && tar -xvf httpd.tar.bz2 -C src --strip-components=1 \
    && rm httpd.tar.bz2 \
    && cd src \
    \
    && ./configure \
        --prefix="$HTTPD_PREFIX" \
        --enable-mods-shared=reallyall \
    && make -j"$(nproc)" \
    && make install \
```

```
    \
    && cd .. \
    && rm -r src \
    \
    && sed -ri \
        -e 's!^(\s*CustomLog)\s+\S+!\1 /proc/self/fd/1!g' \
        -e 's!^(\s*ErrorLog)\s+\S+!\1 /proc/self/fd/2!g' \
        "$HTTPD_PREFIX/conf/httpd.conf" \
    \
    && apt-get purge -y --auto-remove $buildDeps

COPY httpd-foreground /usr/local/bin/
EXPOSE 80
CMD ["httpd-foreground"]
```

Dockerfile 文件中 COPY httpd-foreground /usr/local/bin/ 是将当前目录下的 httpd-foreground 拷贝到镜像里，作为 httpd 服务的启动脚本，所以我们要在本地创建一个脚本文件 httpd-foreground

```
#!/bin/bash
set -e
# Apache gets grumpy about PID files pre-existing
rm -f /usr/local/apache2/logs/httpd.pid
exec httpd -DFOREGROUND
```

赋予 httpd-foreground 文件可执行权限

```
$ chmod +x httpd-foreground
```

通过 Dockerfile 创建一个镜像，替换成你自己的名字

```
$ docker build -t httpd .
```

创建完成后，我们可以在本地的镜像列表里查找到刚刚创建的镜像

```
$ docker images httpd
REPOSITORY      TAG         IMAGE ID        CREATED           SIZE
httpd           latest      da1536b4ef14    23 seconds ago    195.1 MB
```

方法二、docker pull httpd

查找 Docker Hub 上的 httpd 镜像

$ docker search httpd

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
httpd	The Apache HTTP Server ..	524	[OK]	
centos/httpd		7		[OK]
rgielen/httpd-image-php5	Docker image for Apache...	1		[OK]
microwebapps/httpd-frontend	Httpd frontend allowing...	1		[OK]
lolhens/httpd	Apache httpd 2 Server	1		[OK]
publici/httpd	httpd:latest	0		[OK]
publicisworldwide/httpd	The Apache httpd webser...	0		[OK]
rgielen/httpd-image-simple	Docker image for simple...	0		[OK]
solsson/httpd	Derivatives of the offi...	0		[OK]
rgielen/httpd-image-drush	Apache HTTPD + Drupal S...	0		[OK]
learninglayers/httpd		0		[OK]
sohrabkhan/httpd	Docker httpd + php5.6 (...	0		[OK]
aintohvri/docker-httpd	Apache HTTPD Docker ext...	0		[OK]
alizarion/httpd	httpd on centos with mo...	0		[OK]

...

这里我们拉取官方的镜像

$ docker pull httpd

等待下载完成后，我们就可以在本地镜像列表里查到 REPOSITORY 为 httpd 的镜像。

使用 apache 镜像

运行容器

docker run -p 80:80 -v $PWD/www/:/usr/local/apache2/htdocs/ -v

$PWD/conf/httpd.conf:/usr/local/apache2/conf/httpd.conf -v

$PWD/logs/:/usr/local/apache2/logs/ -d httpd

命令说明：

-p 80:80 :将容器的 80 端口映射到主机的 80 端口

-v $PWD/www/:/usr/local/apache2/htdocs/ :将主机中当前目录下的 www 目录挂载到容器的 /usr/local/apache2/htdocs/

-v $PWD/conf/httpd.conf:/usr/local/apache2/conf/httpd.conf :将主机中当前目录下的 conf/httpd.conf 文件挂载到容器的/usr/local/apache2/conf/httpd.conf

-v $PWD/logs/:/usr/local/apache2/logs/ :将主机中当前目录下的 logs 目录挂载到容器的 /usr/local/apache2/logs/

查看容器启动情况

$ docker ps

```
CONTAINER ID  IMAGE   COMMAND              ... PORTS               NAMES
79a97f2aac37  httpd   "httpd-foreground"  ... 0.0.0.0:80->80/tcp  sharp_swanson
```

通过浏览器访问验证 Docker 中的 Apache 服务是否可访问。

5.2 Docker 容器实现 Nginx 服务

方法一、通过 Dockerfile 构建

创建 Dockerfile

首先，创建目录 nginx,用于存放后面的相关东西。

$ mkdir -p ~/nginx/www ~/nginx/logs ~/nginx/conf

www 目录将映射为 nginx 容器配置的虚拟目录，logs 目录将映射为 nginx 容器的日志目录，conf 目录里的配置文件将映射为 nginx 容器的配置文件，进入创建的 nginx 目录，创建 Dockerfile。

```
FROM debian:jessie

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

ENV NGINX_VERSION 1.10.1-1~jessie


RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62 \

        && echo "deb http://nginx.org/packages/debian/ jessie nginx" >> /etc/apt/sources.list \
```

```
        && apt-get update \
        && apt-get install --no-install-recommends --no-install-suggests -y \
                                                ca-certificates \
                                                nginx=${NGINX_VERSION} \
                                                nginx-module-xslt \
                                                nginx-module-geoip \
                                                nginx-module-image-filter \
                                                nginx-module-perl \
                                                nginx-module-njs \
                                                gettext-base \
        && rm -rf /var/lib/apt/lists/*

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
        && ln -sf /dev/stderr /var/log/nginx/error.log
EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

通过 Dockerfile 创建一个镜像，替换成你自己的名字。

docker build -t nginx .

创建完成后，我们可以在本地的镜像列表里查找到刚刚创建的镜像

$ docker images nginx

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	latest	555bbd91e13c	3 days ago	182.8 MB

方法二、docker pull nginx

查找 Docker Hub 上的 nginx 镜像。

$ docker search nginx

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
nginx	Official build of Nginx.	3260	[OK]	
jwilder/nginx-proxy	Automated Nginx reverse proxy for docker c...	674		[OK]

```
richarvey/nginx-php-fpm   Container running Nginx + PHP-FPM capable ...   207   [OK]
million12/nginx-php       Nginx + PHP-FPM 5.5, 5.6, 7.0 (NG), CentOS...   67    [OK]
maxexcloo/nginx-php       Docker framework container with Nginx and ...   57    [OK]
webdevops/php-nginx       Nginx with PHP-FPM                              39    [OK]
h3nrik/nginx-ldap         NGINX web server with LDAP/AD, SSL and pro...   27    [OK]
bitnami/nginx             Bitnami nginx Docker Image                      19    [OK]
maxexcloo/nginx           Docker framework container with Nginx inst...   7     [OK]
...
```

这里我们拉取官方的镜像

```
$ docker pull nginx
```

等待下载完成后，我们就可以在本地镜像列表里查到 REPOSITORY 为 nginx 的镜像。

使用 nginx 镜像，运行容器

```
$ docker run -p 80:80 --name mynginx -v $PWD/www:/www -v
$PWD/conf/nginx.conf:/etc/nginx/nginx.conf -v $PWD/logs:/wwwlogs  -d nginx
45c89fab0bf9ad643bc7ab571f3ccd65379b844498f54a7c8a4e7ca1dc3a2c1e
```

命令说明：

-p 80:80：将容器的 80 端口映射到主机的 80 端口

--name mynginx：将容器命名为 mynginx

-v $PWD/www:/www：将主机中当前目录下的 www 挂载到容器的/www

-v $PWD/conf/nginx.conf:/etc/nginx/nginx.conf：将主机中当前目录下的 nginx.conf 挂载到容器的/etc/nginx/nginx.conf

-v $PWD/logs:/wwwlogs：将主机中当前目录下的 logs 挂载到容器的/wwwlogs

查看容器启动情况

```
$ docker ps
CONTAINER ID        IMAGE         COMMAND                  ...   PORTS                    NAMES
45c89fab0bf9        nginx         "nginx -g 'daemon off"   ...   0.0.0.0:80->80/tcp, 443/tcp   mynginx
f2fa96138d71        tomcat        "catalina.sh run"        ...   0.0.0.0:81->8080/tcp     tomcat
```

通过浏览器访问

5.3 Docker 容器实现 Python 应用

方法一、通过 Dockerfile 构建

创建 Dockerfile

首先，创建目录 python,用于存放后面的相关东西。

$ mdkir -p ~/python ~/python/myapp

myapp 目录将映射为 python 容器配置的应用目录，进入创建的 python 目录，创建 Dockerfile。

```
FROM buildpack-deps:jessie

# remove several traces of debian python

RUN apt-get purge -y python.*


# http://bugs.python.org/issue19846

# > At the moment, setting "LANG=C" on a Linux system *fundamentally breaks Python 3*, and that's not OK.

ENV LANG C.UTF-8


# gpg: key F73C700D: public key "Larry Hastings <larry@hastings.org>" imported

ENV GPG_KEY 97FC712E4C024BBEA48A61ED3A5CA953F73C700D


ENV PYTHON_VERSION 3.5.1


# if this is called "PIP_VERSION", pip explodes with "ValueError: invalid truth value '<VERSION>'"

ENV PYTHON_PIP_VERSION 8.1.2
```

```
RUN set -ex \
        && curl -fSL "https://www.python.org/ftp/python/${PYTHON_VERSION%%[a-z]*}/Python-
$PYTHON_VERSION.tar.xz" -o python.tar.xz \
        && curl -fSL "https://www.python.org/ftp/python/${PYTHON_VERSION%%[a-z]*}/Python-
$PYTHON_VERSION.tar.xz.asc" -o python.tar.xz.asc \
        && export GNUPGHOME="$(mktemp -d)" \
        && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys "$GPG_KEY" \
        && gpg --batch --verify python.tar.xz.asc python.tar.xz \
        && rm -r "$GNUPGHOME" python.tar.xz.asc \
        && mkdir -p /usr/src/python \
        && tar -xJC /usr/src/python --strip-components=1 -f python.tar.xz \
        && rm python.tar.xz \
        \
        && cd /usr/src/python \
        && ./configure --enable-shared --enable-unicode=ucs4 \
        && make -j$(nproc) \
        && make install \
        && ldconfig \
        && pip3 install --no-cache-dir --upgrade --ignore-installed pip==$PYTHON_PIP_VERSION \
        && find /usr/local -depth \
                \( \
                    \( -type d -a -name test -o -name tests \) \
                    -o \
                    \( -type f -a -name '*.pyc' -o -name '*.pyo' \) \
                \) -exec rm -rf '{}' + \
        && rm -rf /usr/src/python ~/.cache

# make some useful symlinks that are expected to exist
RUN cd /usr/local/bin \
```

```
        && ln -s easy_install-3.5 easy_install \
        && ln -s idle3 idle \
        && ln -s pydoc3 pydoc \
        && ln -s python3 python \
        && ln -s python3-config python-config

CMD ["python3"]
```

通过 Dockerfile 创建一个镜像，替换成你自己的名字

```
$ docker build -t python:3.5 .
```

创建完成后，我们可以在本地的镜像列表里查找到刚刚创建的镜像

```
$ docker images python:3.5
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
python              3.5                 045767ddf24a        9 days ago          684.1 MB
```

方法二、docker pull python

查找 Docker Hub 上的 python 镜像

```
$ docker search python
NAME                      DESCRIPTION                     STARS     OFFICIAL   AUTOMATED
python                    Python is an interpreted,...    982       [OK]
kaggle/python             Docker image for Python...      33                   [OK]
azukiapp/python           Docker image to run Python ...  3                    [OK]
vimagick/python           mini python                               2          [OK]
tsuru/python              Image for the Python ...        2                    [OK]
pandada8/alpine-python    An alpine based python image              1          [OK]
1science/python           Python Docker images based on ...  1                 [OK]
lucidfrontier45/python-uwsgi  Python with uWSGI           1                    [OK]
orbweb/python             Python image                    1                    [OK]
pathwar/python            Python template for Pathwar levels 1                 [OK]
rounds/10m-python         Python, setuptools and pip.     0                    [OK]
ruimashita/python         ubuntu 14.04 python             0                    [OK]
```

tnanba/python Python on CentOS-7 image. 0 [OK]

这里我们拉取官方的镜像,标签为 3.5

$ docker pull python:3.5

等待下载完成后，我们就可以在本地镜像列表里查到 REPOSITORY 为 python,标签为 3.5 的镜像。

在~/python/myapp 目录下创建一个 helloworld.py 文件，代码如下：

#!/usr/bin/python

print("Hello, World!");

运行容器

$ docker run -v $PWD/myapp:/usr/src/myapp -w /usr/src/myapp python:3.5 python helloworld.py

命令说明：

-v $PWD/myapp:/usr/src/myapp :将主机中当前目录下的 myapp 挂载到容器的 /usr/src/myapp

-w /usr/src/myapp :指定容器的/usr/src/myapp 目录为工作目录

python helloworld.py :使用容器的 python 命令来执行工作目录中的 helloworld.py 文件

输出结果：

Hello, World!

5.4 Docker 容器实现 MySQL 服务

方法一、通过 Dockerfile 构建

创建 Dockerfile

首先，创建目录 mysql,用于存放后面的相关东西。

$ mkdir -p ~/mysql/data ~/mysql/logs ~/mysql/conf

data 目录将映射为 mysql 容器配置的数据文件存放路径，logs 目录将映射为 mysql 容器的日志目录，conf 目录里的配置文件将映射为 mysql 容器的配置文件。

进入创建的 mysql 目录，创建 Dockerfile

```
FROM debian:jessie
```

```
# add our user and group first to make sure their IDs get assigned consistently, regardless of whatever dependencies get added
RUN groupadd -r mysql && useradd -r -g mysql mysql

# add gosu for easy step-down from root
ENV GOSU_VERSION 1.7
RUN set -x \
    && apt-get update && apt-get install -y --no-install-recommends ca-certificates wget && rm -rf /var/lib/apt/lists/* \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architecture)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B42F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/bin/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
    && apt-get purge -y --auto-remove ca-certificates wget

RUN mkdir /docker-entrypoint-initdb.d

# FATAL ERROR: please install the following Perl modules before executing /usr/local/mysql/scripts/mysql_install_db:
# File::Basename
# File::Copy
# Sys::Hostname
# Data::Dumper
```

```
RUN apt-get update && apt-get install -y perl pwgen --no-install-recommends && rm -rf /var/lib/apt/lists/*

# gpg: key 5072E1F5: public key "MySQL Release Engineering <mysql-build@oss.oracle.com>" imported
RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys A4A9406876FCBD3C456770C88C718D3B5072E1F5

ENV MYSQL_MAJOR 5.6
ENV MYSQL_VERSION 5.6.31-1debian8

RUN echo "deb http://repo.mysql.com/apt/debian/ jessie mysql-${MYSQL_MAJOR}" >
/etc/apt/sources.list.d/mysql.list

# the "/var/lib/mysql" stuff here is because the mysql-server postinst doesn't have an explicit way to
disable the mysql_install_db codepath besides having a database already "configured" (ie, stuff in
/var/lib/mysql/mysql)
# also, we set debconf keys to make APT a little quieter
RUN { \
        echo mysql-community-server mysql-community-server/data-dir select ''; \
        echo mysql-community-server mysql-community-server/root-pass password ''; \
        echo mysql-community-server mysql-community-server/re-root-pass password ''; \
        echo mysql-community-server mysql-community-server/remove-test-db select false; \
    } | debconf-set-selections \
    && apt-get update && apt-get install -y mysql-server="${MYSQL_VERSION}" && rm -rf /var/lib/apt/lists/*
\
    && rm -rf /var/lib/mysql && mkdir -p /var/lib/mysql /var/run/mysqld \
    && chown -R mysql:mysql /var/lib/mysql /var/run/mysqld \
# ensure that /var/run/mysqld (used for socket and lock files) is writable regardless of the UID our mysqld
instance ends up having at runtime
    && chmod 777 /var/run/mysqld

# comment out a few problematic configuration values
```

```
# don't reverse lookup hostnames, they are usually another container
RUN sed -Ei 's/^(bind-address|log)/#&/' /etc/mysql/my.cnf \
    && echo 'skip-host-cache\nskip-name-resolve' | awk '{ print } $1 == "[mysqld]" && c == 0 { c = 1;
system("cat") }' /etc/mysql/my.cnf > /tmp/my.cnf \
    && mv /tmp/my.cnf /etc/mysql/my.cnf

VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /usr/local/bin/
RUN ln -s usr/local/bin/docker-entrypoint.sh /entrypoint.sh # backwards compat
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 3306
CMD ["mysqld"]
```

通过 Dockerfile 创建一个镜像，替换成你自己的名字。

```
$ docker build -t mysql .
```

创建完成后，我们可以在本地的镜像列表里查找到刚刚创建的镜像

```
$ docker images |grep mysql
mysql               5.6                 2c0964ec182a        3 weeks ago         329 MB
```

方法二、docker pull mysql

查找 Docker Hub 上的 mysql 镜像。

```
$ docker search mysql
NAME                      DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
mysql                     MySQL is a widely used, open-source relati...   2529      [OK]
mysql/mysql-server        Optimized MySQL Server Docker images. Crea...   161                  [OK]
centurylink/mysql         Image containing mysql. Optimized to be li...   45                   [OK]
sameersbn/mysql                                                           36                   [OK]
google/mysql              MySQL server for Google Compute Engine          16                   [OK]
appcontainers/mysql       Centos/Debian Based Customizable MySQL Con...   8                    [OK]
```

```
marvambass/mysql          MySQL Server based on Ubuntu 14.04          6          [OK]
drupaldocker/mysql        MySQL for Drupal                            2          [OK]
azukiapp/mysql            Docker image to run MySQL by Azuki - http:...  2       [OK]
...
```

这里我们拉取官方的镜像,标签为 5.6。

```
$ docker pull mysql:5.6
```

等待下载完成后，我们就可以在本地镜像列表里查到 REPOSITORY 为 mysql,标签为 5.6 的镜像。

运行容器：

```
$ docker run -p 3306:3306 --name mymysql -v $PWD/conf/my.cnf:/etc/mysql/my.cnf -v $PWD/logs:/logs -v $PWD/data:/mysql_data -e MYSQL_ROOT_PASSWORD=123456 -d mysql:5.6
21cb89213c93d805c5bacf1028a0da7b5c5852761ba81327e6b99bb3ea89930e
```

命令说明：

-p 3306:3306：将容器的 3306 端口映射到主机的 3306 端口

-v $PWD/conf/my.cnf:/etc/mysql/my.cnf：将主机当前目录下的 conf/my.cnf 挂载到容器的/etc/mysql/my.cnf

-v $PWD/logs:/logs：将主机当前目录下的 logs 目录挂载到容器的/logs

-v $PWD/data:/mysql_data：将主机当前目录下的 data 目录挂载到容器的/mysql_data

-e MYSQL_ROOT_PASSWORD=123456：初始化 root 用户的密码

查看容器启动情况：

```
$ docker ps
CONTAINER ID    IMAGE         COMMAND                  ...  PORTS                    NAMES
21cb89213c93    mysql:5.6     "docker-entrypoint.sh"   ...  0.0.0.0:3306->3306/tcp   mymysql
```

第六部分　Docker 的运行监控

在运维体系中， 监控是非常重要的组成部分。通过监控可以实时掌握系统运行的状

态，对故障的提前预警，历史状态的回放等，还可以通过监控数据为系统的容量规划提供辅助决策，为系统性能优化提供真实的用户行为和体验。

6.1 容器的监控方案

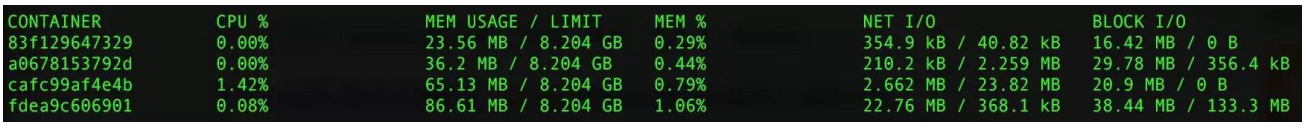
传统的监控系统大多是针对物理机或虚拟机设计的，物理机和虚拟机的特点是静态的，生命周期长，一个环境安装配置好后可能几年都不会去变动，那么对监控系统来说，监控对像是静态的，对监控对象做的监控配置也是静态的，系统上线部署好监控后基本就不再需要管理。

虽然物理机、虚拟机、容器对于应用进程来说都是 host 环境，容器也是一个轻量级的虚拟机， 但容器是动态的， 生命周期短，特别是在微服务的分布式架构下，容器的个数，IP 地址随时可能变化。如果还采用原来传统监控的方案，则会增加监控的复杂度。比如对于一个物理机或虚拟机，我们只要安装一个监控工具的 agent 就可以了，但如果在一个物理机上运行了无数个容器，也采用安装 agent 的方式，就会增加 agent 对资源的占用，但因为容器是与宿主机是共享资源，所以在容器内采集的性能数据会是宿主机的数据，那就失去在容器内采集数据的意义。

而且往往容器的数量比较多，采集到的数量也会非常多，容器可能启动几分钟就停止了，那么原来采集的数据就没有价值了，则会产生大量这样没有价值的监控数据，维护起来也会非常的复杂。那么应该如何对容器进行监控呢？答案是在容器外，宿主机上进行监控。这样不仅可以监控到每个容器的资源使用情况，还可以监控到容器的状态，数量等数据。

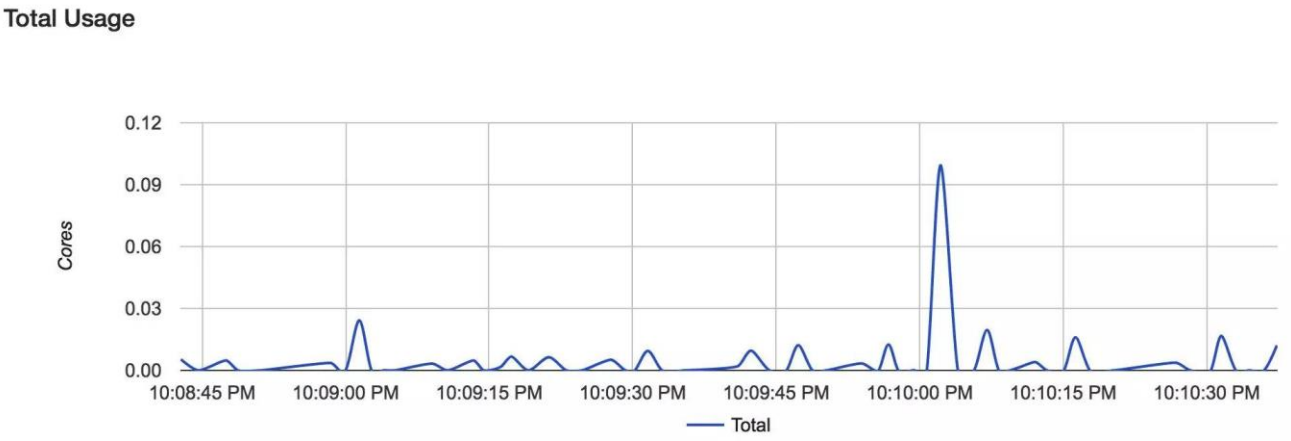
6.2 单台主机上容器的监控

单台主机上容器的监控实现最简单的方法就是使用命令 Docker stats，就可以显示所有容器的资源使用情况，如下输出：



CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O
83f129647329	0.00%	23.56 MB / 8.204 GB	0.29%	354.9 kB / 40.82 kB	16.42 MB / 0 B
a0678153792d	0.00%	36.2 MB / 8.204 GB	0.44%	210.2 kB / 2.259 MB	29.78 MB / 356.4 kB
cafc99af4e4b	1.42%	65.13 MB / 8.204 GB	0.79%	2.662 MB / 23.82 MB	20.9 MB / 0 B
fdea9c606901	0.08%	86.61 MB / 8.204 GB	1.06%	22.76 MB / 368.1 kB	38.44 MB / 133.3 MB

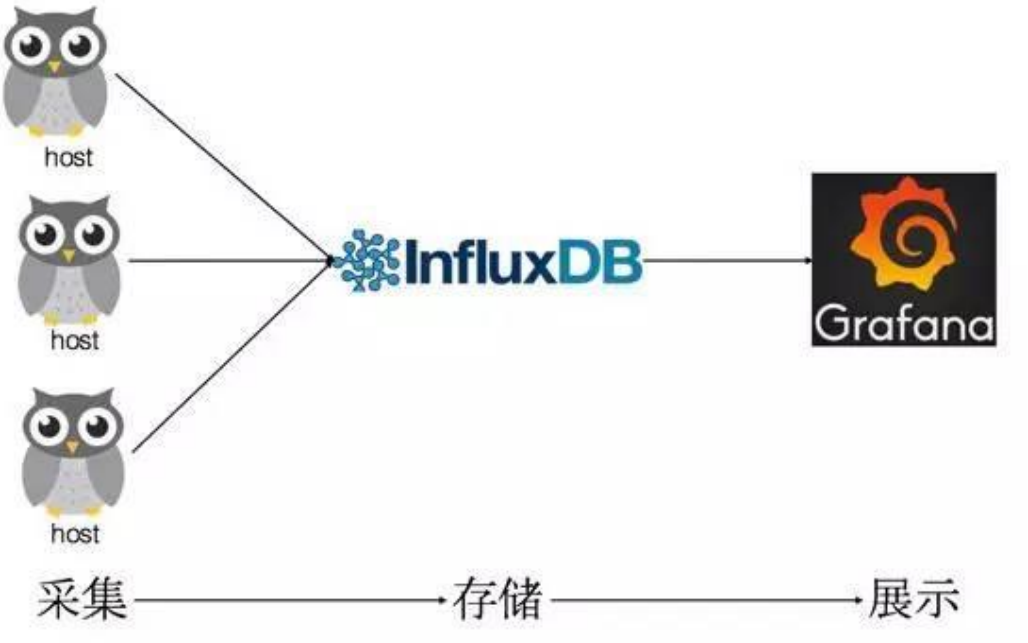
虽然可以很直观地看到每个容器的资源使用情况，但是显示的只是一个当前值，并不能看到变化趋势。而谷歌提供的图形化工具不仅可以看到每个容器的资源使用情况，还可以看到主机的资源使用情况，并且可以设置显示一段时间内的趋势。以下是 cAdvisor 的面板：



而且 cAdivsor 的安装非常简单，下载一个 cAdvisor 的容器启动后，就可以使用主机 IP 加默认端口 8080 进行访问了。

6.3 跨多台主机上容器的监控

cAdivsor 虽然能采集到监控数据，也有很好的界面展示，但是并不能显示跨主机的监控数据，当主机多的情况，需要有一种集中式的管理方法将数据进行汇总展示，最经典的方案就是 cAdvisor+ Influxdb+grafana，可以在每台主机上运行一个 cAdvisor 容器负责数据采集，再将采集后的数据都存到时序型数据库 influxdb 中，再通过图形展示工具 grafana 定制展示面板。结构如下：



这三个工具的安装也非常简单，可以直接启动三个容器快速安装。如下所示：

启动influxdb容器

```
#docker exec -ti influxsrv /bin/bash docker run -d -p 8083:8083 -p 8086:8086 --expose 8090 --expose 8099 --name influxsrv tutum/influxdb
```

通过主机IP+8083端口访问控制台

进入influxdb容器安装cadvisor数据库

```
#docker exec -it influxsrv /bin/bash
$ influx
➢ CREATE DATABASE cadvisor
➢ > use cadvisor
➢ > CREATE USER "root" WITH PASSWORD 'root' WITH ALL PRIVILEGES
```

启动cadvisor容器

```
#docker exec -ti influxsrv /bin/bash docker run --volume=/:/rootfs:ro --volume=/var/run:/var/run:rw --volume=/sys:/sys:ro --volume=/var/lib/docker/:/var/lib/docker:ro --publish=8080:8080 --detach=true --link influxsrv:influxsrv --name=cadvisor google/cadvisor:latest -storage_driver=influxdb -storage_driver_db=cadvisor -storage_driver_host=influxsrv:8086
```

通过主机IP+8080端口访问控制台

启动grafana容器

```
#docker exec -ti influxsrv /bin/bash docker run -d -p 3000:3000 -e INFLUXDB_HOST= 10.105.72.79 -e INFLUXDB_PORT=8086 -e INFLUXDB_NAME=cadvisor -e INFLUXDB_USER=root -e INFLUXDB_PASS=root --link influxsrv:influxsrv --name grafana grafana/grafana
```

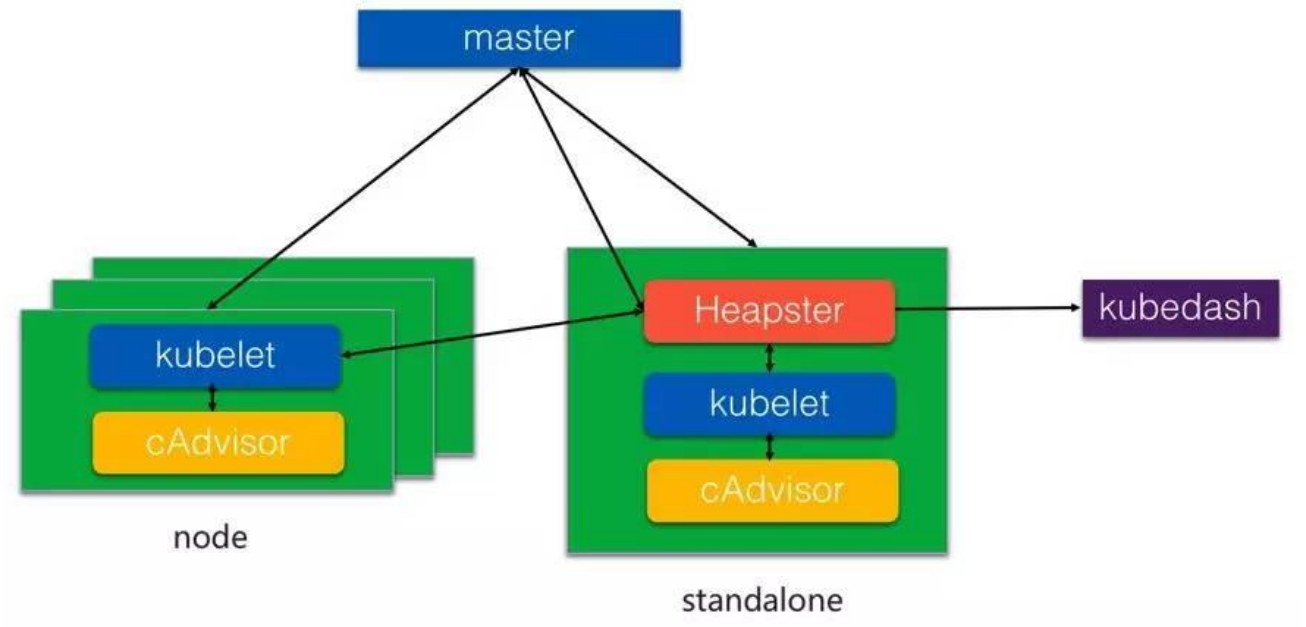
通过主机IP+3000端口访问控制台，用户名密码为admin/admin

在上面的安装步骤中，先是启动 influxdb 容器，然后进行到容器内部配置一个数据库给 cadvisor 专用，然后再启动 cadvisor 容器，容器启动的时候指定把数据存储到 influxdb 中，最后启动 grafana 容器，在展示页面里配置 grafana 的数据源为 influxdb，再定制要展

示的数据，一个简单的跨多主机的监控系统就构建成功了。下图为 Grafana 的界面：

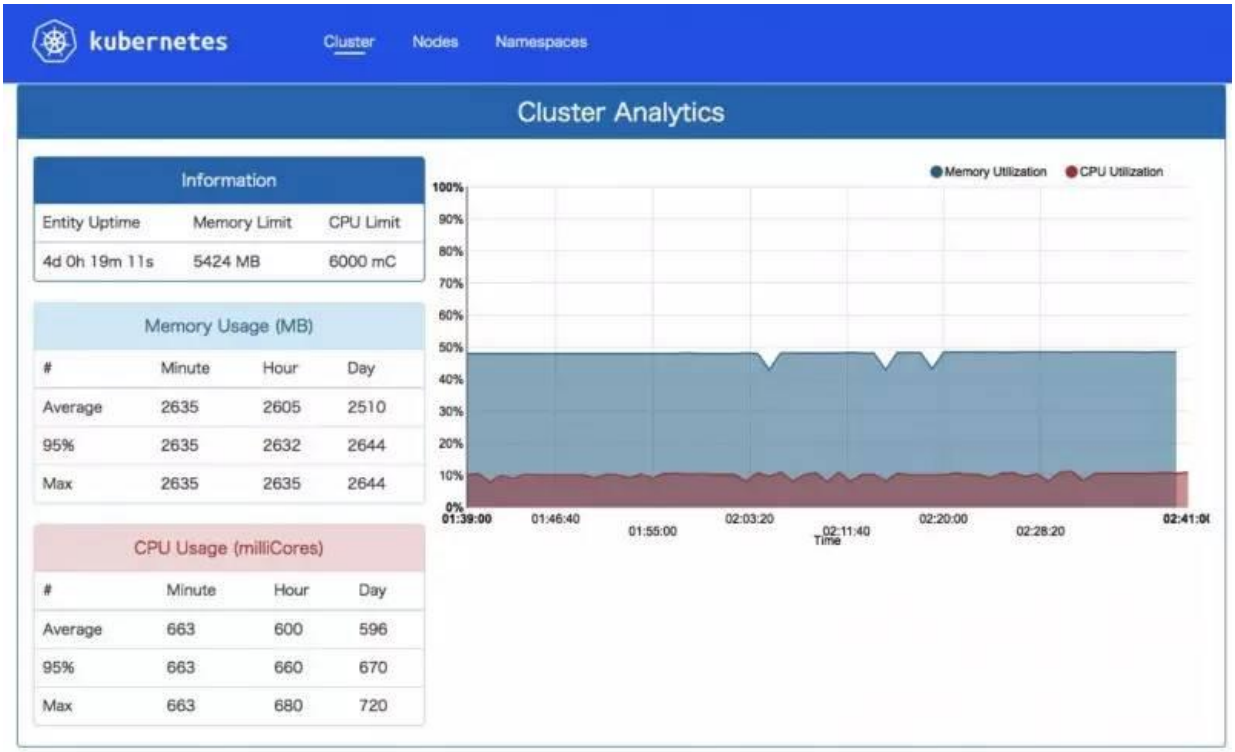
6.4 Kubernetes 上容器的监控

在 Kubernetes 的新版本中已经集成了 cAdvisor，所以在 Kubernetes 架构下，不需要单独再去安装 cAdvisor，可以直接使用节点的 IP 加默认端口 4194 就可以直接访问 cAdvisor 的监控面板。而 Kubernetes 还提供一个叫 heapster 的组件用于聚合每个 node 上 cAdvisor 采集的数据，再通过 Kubedash 进行展示，结构如下：



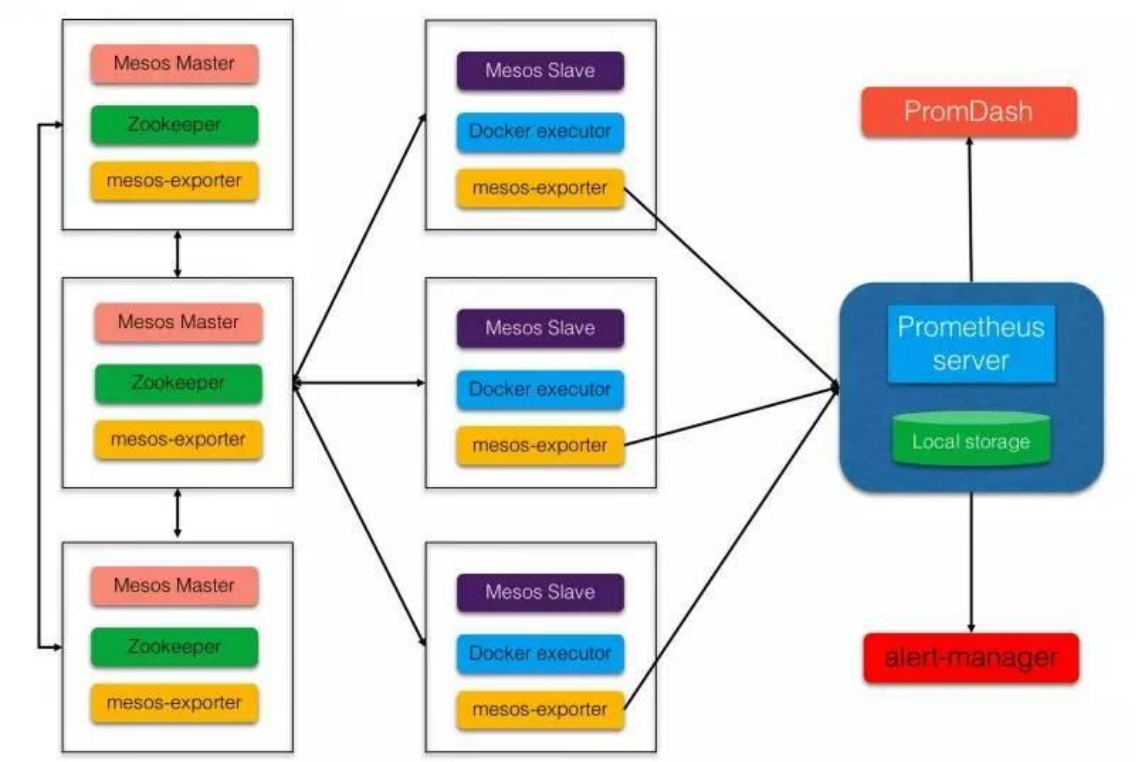
在 Kubernetes 的框架里，master 复杂调度后有的 node，所以在 heapster 启动时，当 heapster 配合 k8s 运行时，需要指定 kubernetes_master 的地址，heapster 通过 k8s 得到所

有 node 节点地址，然后通过访问对应的 node ip 和端口号(10250)来调用目标节点 Kubelet 的 HTTP 接口，再由 Kubelet 调用 cAdvisor 服务获取该节点上所有容器的性能数据，并依次返回到 heapster 进行数据聚合。再通过 kubedash 进行展示，界面如下：

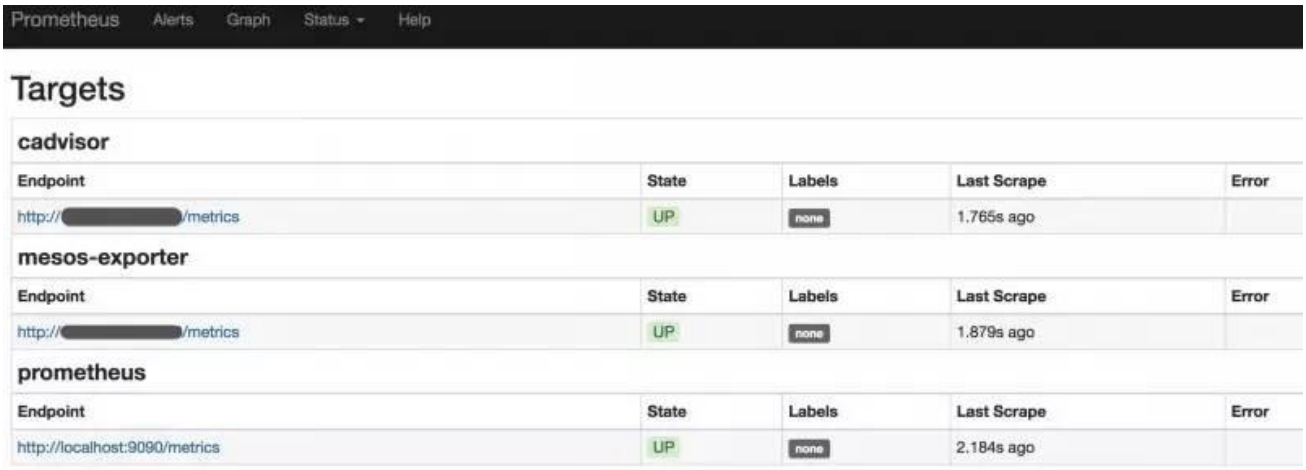


6.5 Mesos 的监控方案

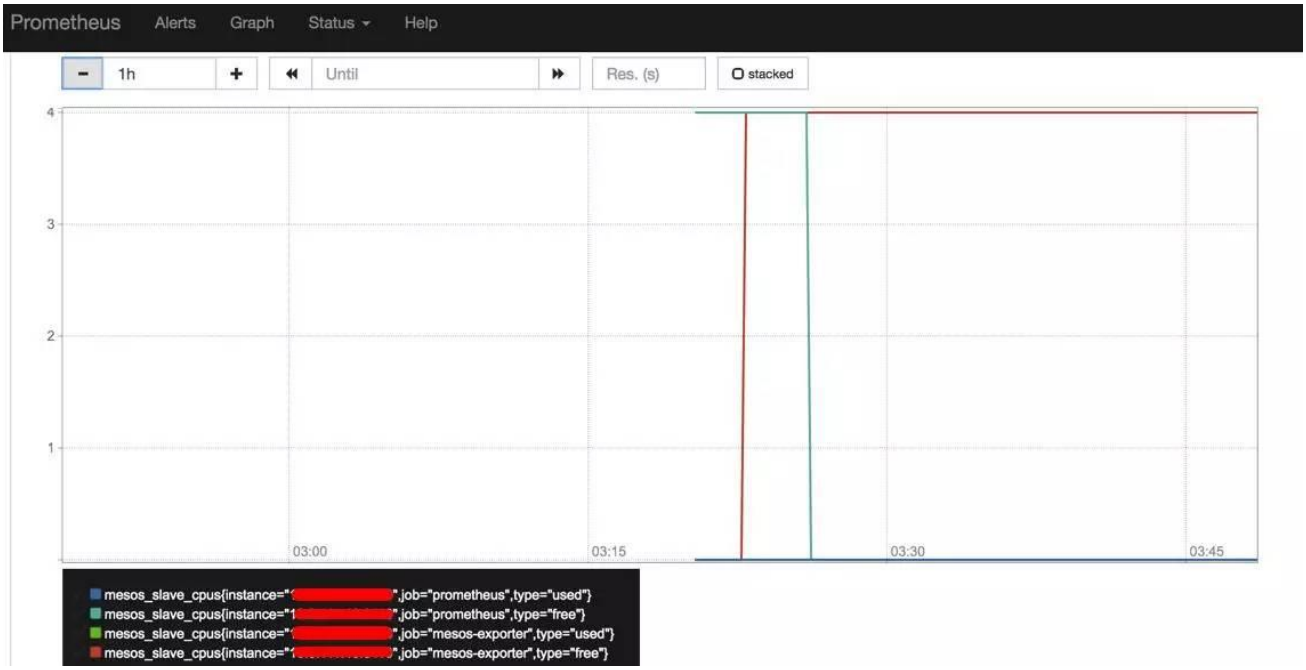
而 Mesos 提供一个 mesos-exporter 工具，用于导出 mesos 集群的监控数据 prometheus，而 prometheus 是个集 db、graph、statistic、alert 于一体的监控工具，安装也非常简单，下载包后做些参数的配置，比如监控的对象就可以运行了，默认通过 9090 端口访问。而 mesos-exporter 工具只需要在每个 slave 节点上启动一个进程，再 mesos-exporter 监控配置到 prometheus server 的监控目标中就可以获取到相关的数据。架构如下：



在 Prometheus 的面板上我们可以看到 Prometheus 的监控对象可以为 mesos-export，也可以为 cAdvisor。



下面为 Prometheus 的展示界面：



6.6 性能采集工具的对比

cAdvisor 可以采集本机以及容器的资源监控数据，如 CPU、 memory、filesystem and network usage statistics)。还可以展示 Docker 的信息及主机上已下载的镜像情况。因为 cAdvisor 默认是将数据缓存在内存中，在显示界面上只能显示 1 分钟左右的趋势，所以历史的数据还是不能看到，但它也提供不同的持久化存储后端，比如 influxdb 等。

Heapster 的前提是使用 cAdvisor 采集每个 node 上主机和容器资源的使用情况，再将所有 node 上的数据进行聚合，这样不仅可以看到整个 Kubernetes 集群的资源情况，还可以分别查看每个 node/namespace 及每个 node/namespace 下 pod 的资源情况。这样就可以从 cluster，node，pod 的各个层面提供详细的资源使用情况。默认也是存储在内存中，也提供不同的持久化存储后端，比如 influxdb 等。

mesos-exporter 的特点是可以采集 task 的监控数据。mesos 在资源调度时是在每个 slave 上启动 task executor，这些 task executor 可以是容器，也可以不是容器。而 mesos-exporter 则可以从 task 的角度来了解资源的使用情况，而不是一个一个没有关联关系的容器。

以上从几个典型的架构上介绍了一些 Docker 的运行监控，需要根据生产环境的特点结合每个监控产品的优势来达到监控的目的。比如 Grafana 的图表展示能力强，但是没有告警的

功能，那么可以结合 Prometheus 在数据处理能力改善数据分析的展示。