目 录

[目 录 1](#_Toc81463612)

[第一章 初识Docker 1](#_Toc81463613)

[1.什么是Docker 1](#_Toc81463614)

[2.为什么要使用Docker 3](#_Toc81463615)

[3.虚拟化与Docker 5](#_Toc81463616)

[4.Docker的应用场景 6](#_Toc81463617)

[5.Docker的优点 6](#_Toc81463618)

[1、快速，一致地交付您的应用程序 6](#_Toc81463619)

[2、响应式部署和扩展 7](#_Toc81463620)

[3、在同一硬件上运行更多工作负载 7](#_Toc81463621)

[相关链接 7](#_Toc81463622)

[第二章 Docker的体系结构 7](#_Toc81463623)

[1. Docker的内部组件 8](#_Toc81463624)

[2. Docker Image的工作原理 9](#_Toc81463625)

[3. Docker仓库 9](#_Toc81463626)

[4. Docker容器 9](#_Toc81463627)

[5. Docker底层技术 9](#_Toc81463628)

[第二章 Docker安装和使用 11](#_Toc81463629)

[1.内部服务器安装docker 11](#_Toc81463630)

[2. Centos7系列安装docker 13](#_Toc81463631)

[3. Docker创建 13](#_Toc81463632)

[4. 守护态运行 13](#_Toc81463633)

[5. 终止、删除容器 14](#_Toc81463634)

[第四章 Docker images的获取和使用 15](#_Toc81463635)

[1. 获取镜像 15](#_Toc81463636)

[2. 查找镜像 16](#_Toc81463637)

[3. 下载镜像 17](#_Toc81463638)

[4. 创建自己的镜像 17](#_Toc81463639)

[第五章 Docker中的网络介绍 22](#_Toc81463640)

[1. 端口映射实现访问容器 22](#_Toc81463641)

[2. Docker容器互联 23](#_Toc81463642)

[第六章 Docker高级网络配置 27](#_Toc81463643)

[1. 配置DNS 27](#_Toc81463644)

[2. 容器之间的通信 29](#_Toc81463645)

[3. 创建一个点到点的连接 30](#_Toc81463646)

[4. Docker的4种网络模式 31](#_Toc81463647)

[第七章 Docker的数据管理 33](#_Toc81463648)

[1. 数据卷 33](#_Toc81463649)

[2. 数据卷容器 35](#_Toc81463650)

[第八章 容器的安全性能 37](#_Toc81463651)

[第九章 学会部署局域网docker 38](#_Toc81463652)

[第十章 Docker实现多台物理机之间的容器互联 39](#_Toc81463653)

[第十一章 Docker中使用Supervisor管理进程 40](#_Toc81463654)

[第十二章 Docker创建tomcat/weblogic集群 41](#_Toc81463655)

[第十三章 Docker安装Redis 41](#_Toc81463656)

[第十四章 Docker安装MySQL 41](#_Toc81463657)

[第十五章 基于Docker的Azure DevOps实现 41](#_Toc81463658)

[1. Docker和持续集成（CI） 50](#_Toc81463659)

[2. 使用Azure DevOps来完成CI 59](#_Toc81463660)

1. 初识Docker

如果说个人主机时代大家比拼的关键是CPU主频的高低和内存的大小，那么在云计算时代，虚拟化技术无疑是整座信息技术大厦最核心的一块基石。

伴随着信息技术产业的发展，虚拟化技术已经应用到各种关键场景中。从最早上世纪60年代IBM推出的大型主机虚拟化到后来x86平台上的虚拟化，虚拟化技术自身也在不断丰富和创新。

虚拟化既可以通过硬件模拟来实现，也可以通过操作系统来实现。而近些年出现的容器虚拟化方案，更是充分利用了操作系统本身已有的机制和特性，可以实现轻量级的虚拟化，甚至有人把它称为新一代的虚拟化技术。Docker毫无疑问就是其中的佼佼者。

那么，什么是Docker？它会带来什么好处？它跟现有虚拟化技术又有何关系呢？

本章在介绍Docker项目的起源和发展之后，会剖析Docker和Linux容器技术的密切联系，以及在开发和运维中使用Docker的突出优势。最后，还将阐述Docker在整个虚拟化领域中的定位。

1.什么是Docker

Docker开源项目

Docker 是一个开源的应用容器引擎，基于Go语言并遵从Apache 2.0协议开源。Docker是一个开放平台，用于开发应用、交付（shipping）应用、运行应用。 Docker允许用户将基础设施（Infrastructure）中的应用单独分割出来，形成更小的颗粒（容器），从而提高交付软件的速度。

Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、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。

容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）,更重要的是容器性能开销极低。

Docker 从 17.03 版本之后分为 CE（Community Edition: 社区版） 和 EE（Enterprise Edition: 企业版），我们用社区版就可以了。

Docker的主要目标是“Build,Ship and Run Any App,Anywhere"，即通过对应用组件的封装（Packaging）、分发（Distribution）、部署（Deployment)、运行（Runtime）等生命周期的管理，达到应用组件级别的“一次封装，到处运行”。这里的应用组件，既可以是一个Web应用，也可以是一套数据库服务，甚至是一个操作系统或编译器。

Docker基于Linux的多项开源技术提供了高效、敏捷和轻量级的容器方案，并且支持在多种主流云平台（Paas）和本地系统上部署。可以说Docker为应用的开发和部署提供了“站式"的解决方案。

Docker容器与虚拟机类似，但二者在原理上不同。容器是将操作系统层虚拟化，虚拟机则是虚拟化硬件，因此容器更具有便携性、高效地利用服务器。 容器更多的用于表示 软件的一个标准化单元。由于容器的标准化，因此它可以无视基础设施（Infrastructure）的差异，部署到任何一个地方。另外，Docker也为容器提供更强的业界的隔离兼容。

Docker 利用Linux核心中的资源分离机制，例如cgroups，以及Linux核心名字空间（namespaces），来创建独立的容器（containers）。这可以在单一Linux实体下运作，避免引导一个虚拟机造成的额外负担[3]。Linux核心对名字空间的支持完全隔离了工作环境中应用程序的视野，包括行程树、网络、用户ID与挂载文件系统，而核心的cgroup提供资源隔离，包括CPU、存储器、block I/O与网络。从0.9版本起，Dockers在使用抽象虚拟是经由libvirt的LXC与systemd - nspawn提供界面的基础上，开始包括libcontainer库做为以自己的方式开始直接使用由Linux核心提供的虚拟化的设施，依据行业分析公司“451研究”：“Dockers是有能力打包应用程序及其虚拟容器，可以在任何Linux服务器上运行的依赖性工具，这有助于实现灵活性和便携性，应用程序在任何地方都可以运行，无论是公用云端服务器、私有云端服务器、单机等。”

Docker作为基于Go语言实现的云开源项目，诞生于2013年初，最初发起者是dotCloud公司。Docker自开源后受到广泛的关注和讨论，目前已有多个相关项目，逐渐形成了围绕Docker的生态体系。dotCloud公司后来也改名为Docker Inc，专注于Docker相关技术和产品的开发。

Docker项目目前已加人了Linux基金会，遵循Apache 2.0协议，全部开源代码均在https://github.com/docker/docker上进行维护。在最近一次Linux基金会的调查中，Docker是仅次于Openstack的最受欢迎的云计算开源项目。

现在主流的Linux操作系统都已经支持Docker。例如，Redhat RHEL 6.5/CentOS 6.5往上的操作系统、Ubuntu 14.04操作系统，都已经默认带有Docker软件包。Google公司宣称在其PaaS（Platform as a Service）平台及服务产品中广泛应用了Docker。微软公司宣布和Docker公司合作，以加强其云平台Azure对Docker的支持。公有云提供商亚马逊近期也推出了AWS EC2 Container，提供对Docker的支持。

Linux容器技术

Docker引擎的基础是Linux容器（Linux Containers，LXC）技术。IBM Developer Works上给出了关于容器技术的准确描述：

容器有效地将由单个操作系统管理的资源划分到孤立的组中，以便更好地在孤立的组之间平衡有冲突的资源使用需求。与虚拟化相比，这样既不需要指令级模拟，也不需要即时编译。容器可以在核心CPU本地运行指令，而不需要任何专门的解释机制。此外，也避免了准虚拟化（paravirtualization）和系统调用替换中的复杂性。

Linux容器其实不是一个全新的概念。最早的容器技术可以追溯到1982年Unix系列操作系统上的chroot工具（直到今天，主流的Unix、Linux操作系统仍然支持和带有该工具）。早期的容器实现技术包括SUN Solaris操作系统上的Solaris Containers（2004年发布），FreeBSD操作系统上的FreeBSD jail（2000年左右出现），以及GNU/Linux上的Linux-VServer(http://linux-vserver.org/)（2001年10月）和OpenVZ(http://openvz.org)（2005年）。

虽然这些技术经过多年的演化已经十分成熟，但是由于种种原因，这些容器技术并没有被集成到主流的Linux内核中，使用起来并不方便。例如，如果用户要使用OpenVZ技术，就需要先给操作系统打上特定的内核补丁方可使用。

后来LXC项目借鉴了前人成熟的容器设计理念，并基于一系列新的内核特性实现了更具扩展性的虚拟化容器方案。更加关键的是，LXC被集成到了主流Linux内核中，进而成为Linux系统轻量级容器技术的事实标准。

从Linux容器到Docker

在LXC的基础上，Docker进一步优化了容器的使用体验。Docker提供了各种容器管理工具（如分发、版本、移植等）让用户无需关注底层的操作，可以简单明了地管理和使用容器。用户操作Docker容器就像操作一个轻量级的虚拟机那样简单。

读者可以简单地将容器理解为一种沙盒（Sandbox)。每个容器内运行一个应用，不同的容器相互隔离，容器之间也可以建立通信机制。容器的创建和停止都十分快速，容器自身对资源的需求也十分有限，远远低于虚拟机。很多时候，甚至直接把容器当作应用本身也没有任何问题。

有理由相信，随着Docker技术的进一步成熟，它将成为更受欢迎的容器虚拟化技术实现，得到更广泛的应用。

2.为什么要使用Docker

Docker效率更高

每个容器共享宿主系统的内核。然而各个容器都运行着一个Linux的副本，这意味着没有Hypervisor，而且不需要额外的启动。容器引擎负责启动或者停止容器，这与虚拟机实现中的Hypervisor类似，但是容器是内核级的虚拟化。

Docker容器虚拟化的好处

Docker项目的发起人和Docker Inc.的CTO——Solomon Hykes认为，Docker在正确的地点、正确的时间顺应了正确的趋势——即高效地构建应用。现在开发者需要能方便地创建运行在云平台上的应用，也就是说应用必须能够脱离底层机器，而且同时必须是“任何时间任何地点"可获取的。因此，开发者们需要一种创建分布式应用程序的方式，这也是Docker所能够提供的。

举个简单的应用场景的例子。假设用户试图基于最常见的LAMP（Linux+Apache+MySQL+PHP)组合来运维一个网站。按照传统的做法，首先，需要安装Apache、MySQL和PHP以及它们各自运行所依赖的环境；之后分别对它们进行配置（包括创建合适的用户、配置参数等）；经过大量的操作后，还需要进行功能测试，看是否工作正常；如果不正常，则意味着更多的时间代价和不可控的风险。可以想象，如果再加上更多的应用，事情会变得更加难以处理。

更为可怕的是，一旦需要服务器迁移（例如从阿里云迁移到腾讯云），往往需要重新部署和调试。这些琐碎而无趣的“体力活"，极大地降低了工作效率。

而Docker提供了一种更为聪明的方式，通过容器来打包应用，意味着迁移只需要在新的服务器上启动需要的容器就可以了。这无疑将节约大量的宝贵时间，并降低部署过程出现问题的风险。

Docker在开发和运维中的优势

对开发和运维（DevOps）人员来说，可能最梦寐以求的就是一次性地创建或配置，可以在任意环境、任意时间让应用正常地运行。而Docker恰恰是可以实现这一终极目标的瑞士军刀。

具体说来，Docker在开发和运维过程中，具有如下几个方面的优势。

* 更快速的交付和部署。使用Docker，开发人员可以使用镜像来快速构建一套标准的开发环境；开发完成之后，测试和运维人员可以直接使用相同环境来部署代码。
* Docker可以快速创建和删除容器，实现快速迭代，大量节约开发、测试、部署的时间。并且，各个步骤都有明确的配置和操作，整个过程全程可见，使团队更容易理解应用的创建和工作过程。
* 更高效的资源利用。Docker容器的运行不需要额外的虚拟化管理程序（Virtual Machine Manager，VMM以及Hypervisor)支持，它是内核级的虚拟化，可以实现更高的性能，同时对资源的额外需求很低。Docker可以快速创建容器，Docker使用的是宿主机的系统内核，所以在创建容器的时候是毫秒级的，但是VM有自己的内核和硬件，开机需要自检，浪费了更多的时间，Docker就节约了很多的开发、测试和部署时间。Docker不需要像VM那样使用Hypervisor进行硬件虚拟化，容器就不需要进行硬件虚拟以及运行完整的操作系统，因此Docker对系统资源的利用率就更高。Docker在运行速度、内存消耗、文件的存储速度上都要比VM更高效。
* 更轻松的迁移和扩展。Docker容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑、服务器等。这种兼容性让用户可以在不同平台之间轻松地迁移应用。由于团队的不同人员使用相同的镜像创建容器，因此容器的执行环境是一致的，目前很多的平台都已经支持docker应用，无论是物理机、虚拟机、公有云还是私有云甚至是笔记本上运行的结果多事一致的。因此用户可以很容易将一个平台上的容器迁移到另外一个平台上。由于Docker使用的是分层存储以及镜像技术，第四章在下周镜像的过程中可以看到镜像下载是分层下载的，因此复用重复的部分就更容易，应用的维护和更新就更简单，基于基础镜像的进一步扩展也就很简单，还可以自己定制镜像。
* 更简单的更新管理。使用Dockerfile，只需要小小的配置修改，就可以替代以往大量的更新工作。并且所有修改都以增量的方式进行分发和更新，从而实现自动化并且高效的容器管理。

Docker与虚拟机比较

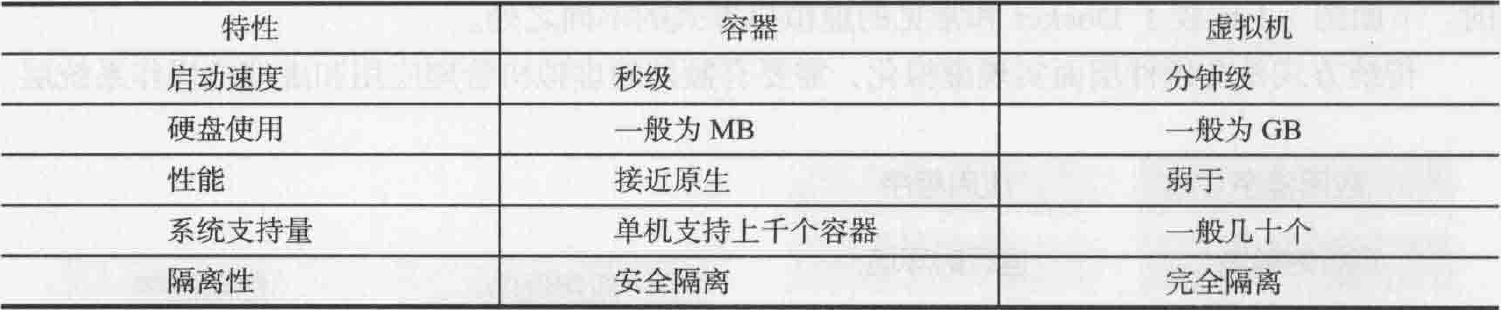
作为一种轻量级的虚拟化方式，Docker在运行应用上跟传统的虚拟机方式相比具有显著优势：

* Docker容器很快，启动和停止可以在秒级实现，这相比传统的虚拟机方式要快得多。
* Docker容器对系统资源需求很少，一台主机上可以同时运行数千个Docker容器。
* Docker通过类似Git的操作来方便用户获取、分发和更新应用镜像，指令简明，学习成本较低。
* Docker通过Dockerfile配置文件来支持灵活的自动化创建和部署机制，提高工作效率。

Docker容器除了运行其中的应用之外，基本不消耗额外的系统资源，保证应用性能的同时，尽量减小系统开销。传统虚拟机方式运行N个不同的应用就要启动N个虚拟机（每个虚拟机需要单独分配独占的内存、磁盘等资源），而Docker只需要启动N个隔离的容器，并将应用放到容器内即可。

当然，在隔离性方面，传统的虚拟机方式多了一层额外的隔离。但这并不意味着Docker就不安全。Docker利用Linux系统上的多种防护机制实现了严格可靠的隔离。从1.3版本开始，Docker引人了安全选项和镜像签名机制，极大地提高了使用Docker的安全性。

下表总结了使用Docker容器技术与传统虚拟机技术的特性比较。



3.虚拟化与Docker

虚拟化技术是一个通用的概念，在不同领域有不同的理解。在计算领域，一般指的是计算虚拟化（Computing Virtualization)，或通常说的服务器虚拟化。维基百科上的定义如下：

在计算机技术中，虚拟化（Virtualization）是一种资源管理技术，是将计算机的各种实体资源，如服务器、网络、内存及存储等，予以抽象、转换后呈现出来，打破实体结构间的不可切割的障砰，使用户可以用比原本的组态更好的方式来应用这些资源。

可见，虚拟化的核心是对资源进行抽象，目标往往是为了在同一个主机上运行多个系统或应用，从而提高系统资源的利用率，同时带来降低成本、方便管理和容错容灾等好处。

从大类上分，虚拟化技术可分为基于硬件的虚拟化和基于软件的虚拟化。其中，真正意义上的基于硬件的虚拟化技术不多见，少数如网卡中的单根多10虚拟化（Single Root I/O Virtualization and Sharing Specification，SR-IOV)等技术，也超出了本书的讨论范畴。

基于软件的虚拟化从对象所在的层次，又可以分为应用虚拟化和平台虚拟化（通常说的虚拟机技术即属于这个范畴）。其中，前者一般指的是一些模拟设备或Wine这样的软件。后者又可以细分为如下几个子类：

* 完全虚拟化。虚拟机模拟完整的底层硬件环境和特权指令的执行过程，客户操作系统无需进行修改。例如VMware Workstation、VirtualBox、QEMU等。
* 硬件辅助虚拟化。利用硬件（主要是CPU）辅助支持（目前x86体系结构上可用的硬件辅助虚拟化技术包括Intel-VT和AMD-V）处理敏感指令来实现完全虚拟化的功能，客户操作系统无需修改，例如VMware Workstation、xen、KVM。
* 部分虚拟化。只针对部分硬件资源进行虚拟化，客户操作系统需要进行修改。现在有些虚拟化技术的早期版本仅支持部分虚拟化。
* 超虚拟化（Paravirtualization）。部分硬件接口以软件的形式提供给客户机操作系统，客户操作系统需要进行修改，例如早期的xeno
* 操作系统级虚拟化。内核通过创建多个虚拟的操作系统实例（内核和库）来隔离不同的进程。容器相关技术即在这个范畴。

可见，Docker以及其他容器技术都属于操作系统的虚拟化这个范畴。

Docker虚拟化方式之所以拥有众多优势，这跟操作系统的虚拟化自身的特点是分不开的。下面图1-1比较了Docker和常见的虚拟机方式的不同之处。

传统方式是在硬件层面实现虚拟化，需要有额外的虚拟机管理应用和虚拟机操作系统层。

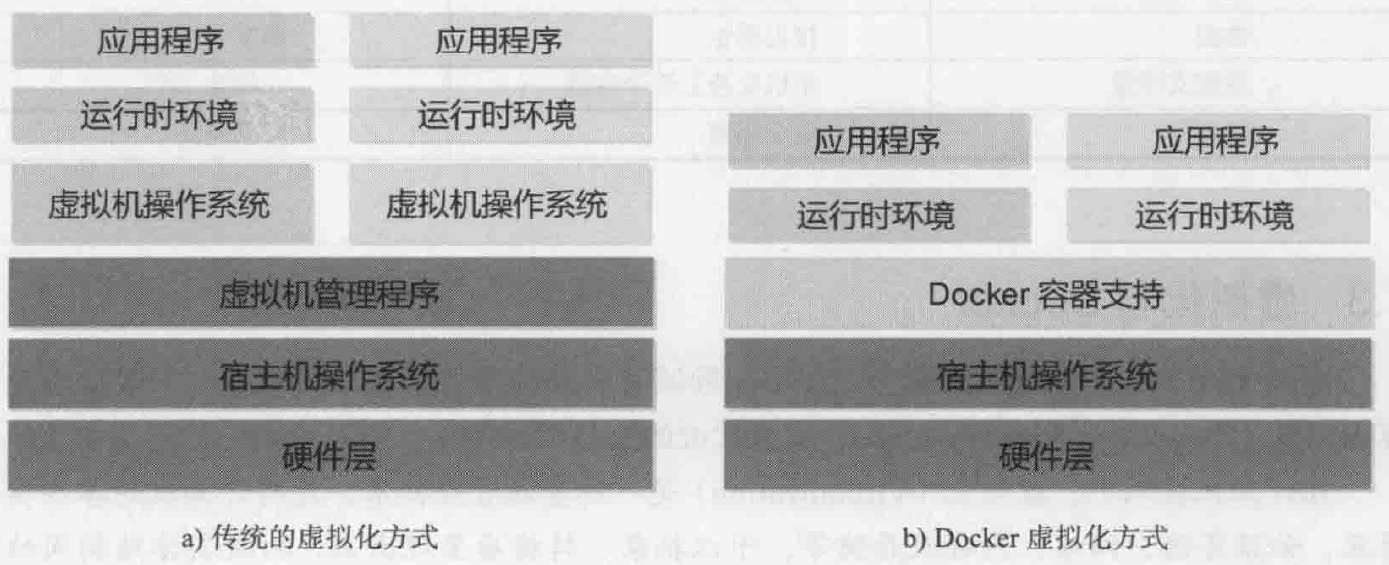


图1-1 Docker和传统的虚拟机方式的不同之处

Docker容器是在操作系统层面上实现虚拟化，直接复用本地主机的操作系统，因此更加轻量级。

4.Docker的应用场景

* Web 应用的自动化打包和发布。
* 自动化测试和持续集成、发布。
* 在服务型环境中部署和调整数据库或其他的后台应用。
* 从头编译或者扩展现有的OpenShift或Cloud Foundry平台来搭建自己的 PaaS 环境。

5.Docker的优点

Docker 是一个用于开发，交付和运行应用程序的开放平台。Docker 使您能够将应用程序与基础架构分开，从而可以快速交付软件。借助 Docker，您可以与管理应用程序相同的方式来管理基础架构。通过利用Docker的方法来快速交付，测试和部署代码，您可以大大减少编写代码和在生产环境中运行代码之间的延迟。

### 1、快速，一致地交付您的应用程序

Docker 允许开发人员使用您提供的应用程序或服务的本地容器在标准化环境中工作，从而简化了开发的生命周期。

容器非常适合持续集成和持续交付（CI / CD）工作流程，请考虑以下示例方案：

* 您的开发人员在本地编写代码，并使用 Docker 容器与同事共享他们的工作。
* 他们使用 Docker 将其应用程序推送到测试环境中，并执行自动或手动测试。
* 当开发人员发现错误时，他们可以在开发环境中对其进行修复，然后将其重新部署到测试环境中，以进行测试和验证。
* 测试完成后，将修补程序推送给生产环境，就像将更新的镜像推送到生产环境一样简单。

### 2、响应式部署和扩展

Docker 是基于容器的平台，允许高度可移植的工作负载。Docker 容器可以在开发人员的本机上，数据中心的物理或虚拟机上，云服务上或混合环境中运行。

Docker 的可移植性和轻量级的特性，还可以使您轻松地完成动态管理的工作负担，并根据业务需求指示，实时扩展或拆除应用程序和服务。

### 3、在同一硬件上运行更多工作负载

Docker 轻巧快速。它为基于虚拟机管理程序的虚拟机提供了可行、经济、高效的替代方案，因此您可以利用更多的计算能力来实现业务目标。Docker非常适合于高密度环境以及中小型部署，而您可以用更少的资源做更多的事情。

## 相关链接

Docker 官网：[https://www.docker.com](https://www.docker.com/)

Github Docker 源码：<https://github.com/docker/docker-ce>

第二章 Docker的体系结构

Docker使用的是C/S结构，即客户机client和服务器server架构，Docker daemon是Docker最核心的后台进程，主要运行在宿主机的后天，它负责响应来自Docker client的请求，然后将这些请求翻译成系统调用完成容器管理操作包括创建、运行、分发容器等管理操作。它可以运行在一个机器上也可以通过Socket通信。Docker client的命令在系统中以bin命令的形式存在,用户通过docker命令来跟docker daemon进行交互。Docker C/S体系架构如下图：



1. Docker的内部组件

Docker主要有三个内部组件，分别是：Docker Images，Docker repository，Docker container。

Docker Images是一个只读的模板。例如一个Images可以包含了一个CentOS系统，里面安装了Apache等应用程序，使用Images可以创建你需要的Container，你可以下载镜像或者自己制作一个镜像。

Docker Repository是Docker仓库是集中存放镜像的地方，分为公共仓库和私有仓库。可以在仓库中下载你自己需要的Images。

Docker Container即Docker容器，容器是从镜像创建的，它可以启动、开始、创建、终止、删除等，每一个容器之间是相互隔离的。

1. Docker Image的工作原理

在下载镜像的过程中可以看到镜像是分层的，Images一般是由若干层组成，字母和数字组成的串是每层的唯一的ID，使用docker pull命令下载镜像的时候会输出每一层的信息，当不同的镜像包括相同的层的时候，本地存储只保存层的一份内容，减少了存储空间。

1. Docker仓库

Docker是存放镜像的地方，分为公有仓库和私有仓库，公有仓库是docker hub，本地的私有仓库是通过registry镜像来创建的。

1. Docker容器

Docker容器是通过镜像来创建的，当使用命令docker run -it centos /bin/bash时，docker在后台运行操作如下：

①如果本地没有centos系统镜像，就从docker hub下载，由于公司业务安全，服务器不可以访问外网，需要从docker.oa.com下载所需要的镜像，使用docker pull命令下载；

②从images创建容器；

③分配一个文件系统，并在只读的镜像层外面挂在一层可读写层；

④从宿主机配置的网桥接口中桥接一个虚拟的接口到容器上；

⑤从网桥的地址池中配置一个IP地址给容器；

⑥执行用户指定的应用程序。

1. Docker底层技术

Docker底层有两个核心的技术Namespace和Control groups。

命名空间是 Linux 内核一个强大的特性。每个容器都有自己单独的命名空间，运行在其中的应用都像是在独立的操作系统中运行一样。命名空间保证了容器之间彼此互不影响。

①pid命名空间

不同用户的进程是通过pid命名空间隔离开的，且不同的命名空间中可以有相同的pid。所有的LXC进程在docker中的父进程为docker进程，每个LXC进程具有不同的命名空间。

②net命名空间

有了pid的命名空间，每个命名空间中的pid是相互隔离的，但是网络还是共享宿主机host的端口，网络隔离是通过net命名空间来实现的，，每个net命名空间有独立的网络设备、IP地址、路由表、/proc/net目录，这样每个容器的网络都可以隔离。Docke采用的是veth的方式，将容器中的虚拟网卡同host上的一个docker网桥docker0连接在一起。

③ipc命名空间

容器中进程交互采用了linux中使用的进程间交互方式（interprocess communication -IPC），主要暴扣信号量、消息队列和共享内存等。

④mnt命名空间

将一个进程放到一个特定的目录执行。mnt命名空间允许不同命名空间的进程看到的文件结构不同，这样每个命名空间中的进程所看到的文件目录就被隔离开了。

⑤uts命名空间

UTS("UNIX Time-sharing System") 命名空间允许每个容器拥有独立的 hostname和domain name,使其在网络上可以被视作一个独立的节点而非主机上的一个进程。

⑥user命名空间

每个容器可以有不同的用户和组id,也就是说非主机上的用户也可以在容器内执行程序。

Control groups

CGroups是Linux内核提供的一种机制，这种机制可以根据特定的行为，把一系列系统任务及其子任务整合（或分隔）到按资源划分等级的不同组内，从而为系统资源管理提供一个统一的框架。CGroups可以限制、记录、隔离进程组所使用的物理资源（包括：CPU、memory、IO等），为容器实现虚拟化提供了基本保证。

Cgroups提供了以下四大功能：资源限制、优先级分配、资源统计、进程控制。

①资源限制（Resource Limitation）：Cgroups可以对进程组使用的资源总额进行限制。如设定应用运行时使用内存的上限，一旦超过这个配额就发出OOM（Out of Memory）。

②优先级分配（Prioritization）：通过分配的CPU时间片数量及硬盘IO带宽大小，实际上就相当于控制了进程运行的优先级。

③资源统计（Accounting）：Cgroups可以统计系统的资源使用量，如CPU使用时长、内存用量等等，这个功能非常适用于计费。

④进程控制（Control）：Cgroups可以对进程组执行挂起、恢复等操作。

1. Docker安装和使用

在Linux安装命令：yum install -y docker-io

查看docker版本：docker version

启动docker服务: service docker start

停止docker服务:service docker stop

查看docker存储位置:docker info

查看docker镜像: docker images

查看docker容器: docker ps

启动docker容器：docker start 容器名称(containerID)

停止docker容器：docker stop 容器名称(containerID)

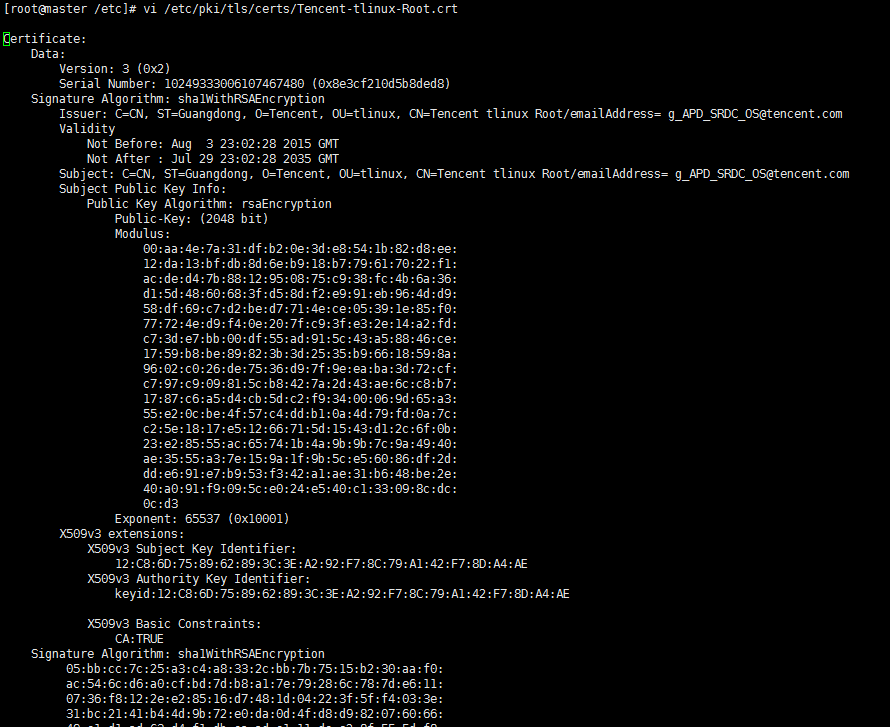
卸载docker容器：docker rmi 容器名称(containerID)

1.内部服务器安装docker

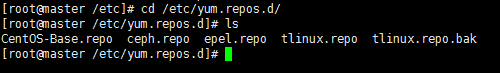
由于公司内部的服务器不可以链接外网，所以安装docker需要使用公司内部的源，因此需要进行Linux的yum配置：

①确认yum服务SSL根证书，确认文件

/etc/pki/tls/certs/Tencent-tlinux-Root.crt；



②确认yum源的配置，见/etc/yum.repos.d/，应该包epel.repo和tlinux.repo两个文件；



③确认yum服务器ip，可以查看/etc/hosts文件，yum服务器域名ip对应关系如下：

10.240.99.238 tlinux-mirrorlist.tencent-cloud.com

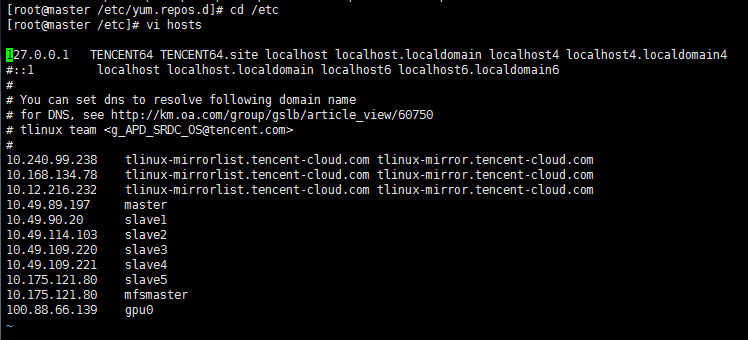
tlinux-mirror.tencent-cloud.com

10.168.134.78 tlinux-mirrorlist.tencent-cloud.com

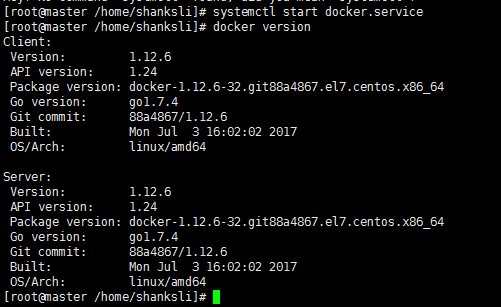
tlinux-mirror.tencent-cloud.com

10.12.216.232 tlinux-mirrorlist.tencent-cloud.com

tlinux-mirror.tencent-cloud.com



以上的配置完成后，就可以使用yum install docker命令直接安装docker了，安装完成后使用systemstl start docker.service命令来启动docker服务，然后使用 docker version命令可以查看docker的版本：



每次启动docker服务后，都需要查看docker版本信息，确保服务已经正常运行。

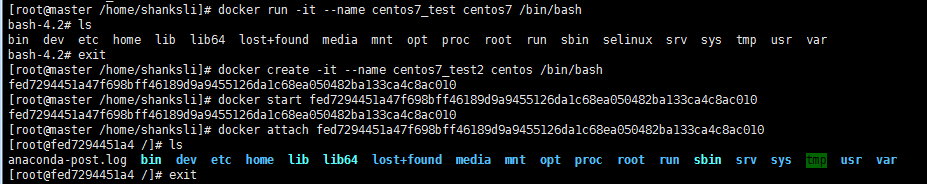
1. Centos7系列安装docker

我自己笔记本虚拟机安装的是Centos7版本的操作系统，可以连接上外网，所以可以使用EPEL软件仓库安装docker，可以直接使用yum install docker-io就可以了。如果是Centos6版本需要先升级到Centos7版本，再进行以上的操作。

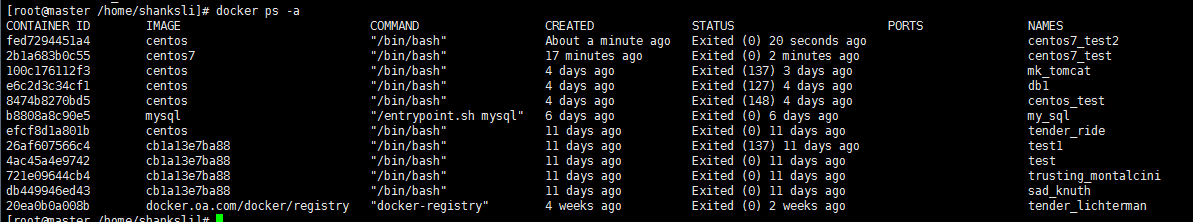
完成docker安装后，如果需要docker服务跟随服务器开机一起启动，可以使用chkconfig docker on命令。

1. Docker创建

首先我们创建一个基于centos7镜像的容器，使用docker run -it --name centos7\_test centos7 /bin/bash命令，创建容器成功后直接运行，也可以使用docker create创建容器，会返回容器的ID，此时需要使用docker start命令启动容器，启动后使用docker attach命令进入该容器：

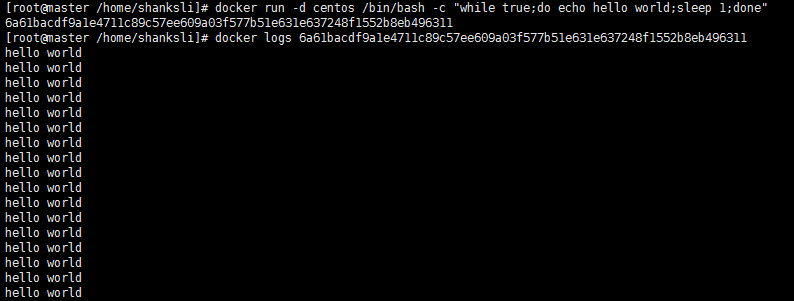


退出容器后可以使用docker ps -a查看在本服务器上创建的容器：



1. 守护态运行

Docker在后台以守护态（Daemonized）形式运行，可以使用—d参数实现。例如在服务器上后台运行一个间隔一秒输出一个hello world的容器：



此时如果需要获取容器的输出信息我们可以使用docker logs命令。

1. 终止、删除容器

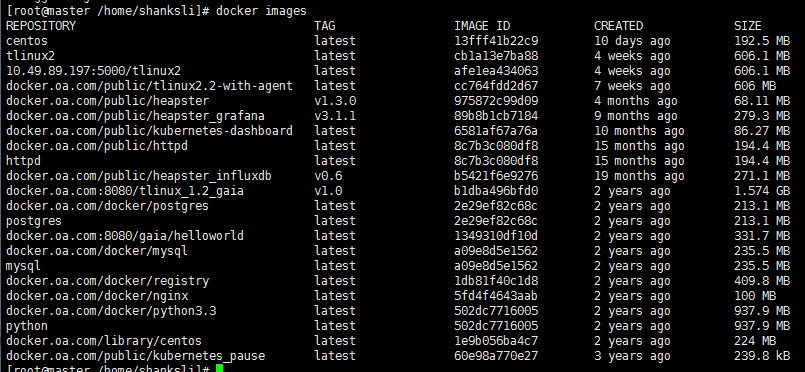
当需要终止一个容器的时候可以使用docker stop和docker kill命令。两个命令是有区别的，docker stop首先是向容器发送SIGTERM信号，等待一段时间后在发送SIGKILL来终止容器；docker kill命令会直接发送SIGKILL信号来强行终止容器。容器不再使用之后可以使用docker rm命令删除，docker rm只可以删除处于终止或者退出的状态的容器，不可以删除处于运行态的容器，如果要删除一个运行态的容器可以加上-f参数。

第四章 Docker images的获取和使用

Docker images是docker的三大组件之一。Docker运行容器之前需要本地存在对应的镜像，如果本地没有，则docker会尝试从默认的镜像仓库中下载，因为公司服务器无法连接到外网，因此只可以通过配置在公司的镜像仓库中下载，公司所有的镜像都保存在docker.oa.com中。

1. 获取镜像

首先是查看本地有哪些镜像，使用docker images命令：



通过列出的信息我们可以看到以下信息：

①镜像来自哪一个仓库，例如docker.oa.com/public/httpd存放的是httpd的镜像；

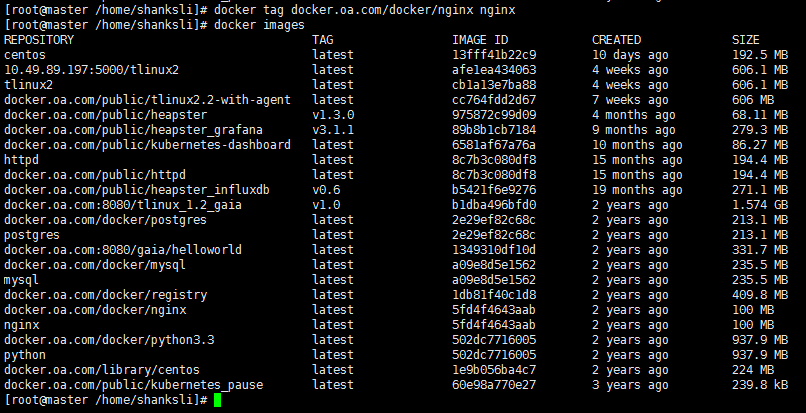
②镜像的标签信息TAG，例如latest、v3.1.1用来标记不同的版本信息，标签只是标记，不可以唯一识别一个镜像；

③IMAGE ID是唯一的镜像标识，例如httpd和docker.oa.com/public/httpd的镜像ID都是一样的8c7b3c080df8，说明他们俩都指向同一个镜像；

④CREATED创建时间说明镜像的最后更新时间；

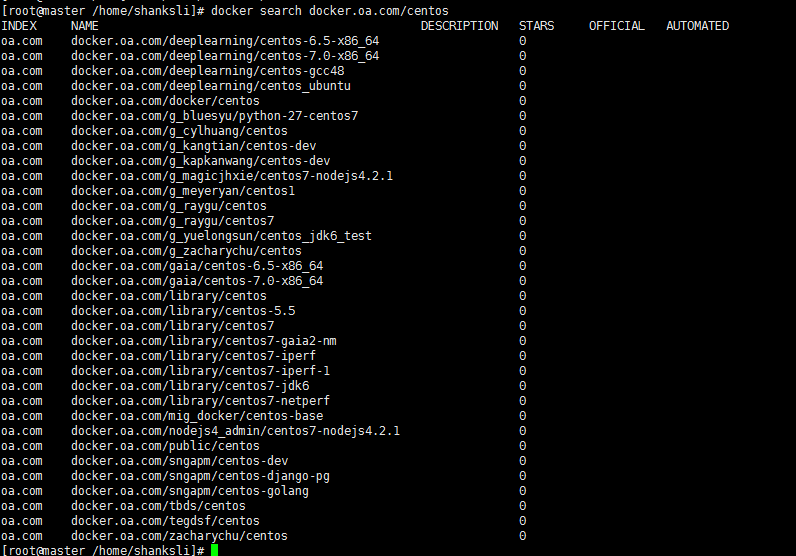
⑤SIZE镜像大小，优秀的镜像往往体积都比较小。

上面我们看到了同一个镜像有两个名字，因为使用了tag给镜像加标签了，例如我们使用docker tag docker.oa.com/docker/nginx nginx表示给镜像docker.oa.com/docker/nginx加一个标签nginx，我们再次使用docker images查看多了一个nginx镜像，但是ID和docker.oa.com/docker/nginx是一致的。



1. 查找镜像

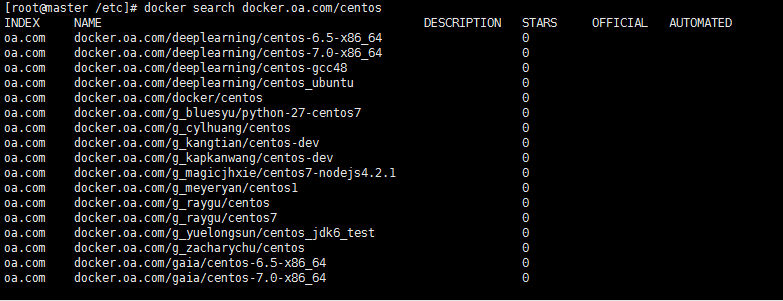
公司内部的所有镜像都在docker.oa.com上，我们在查找镜像的时候使用docker search命令可以找到自己需要的镜像，例如我需要查找一个centos的镜像：docker search docker.oa.com/centos



查找镜像的时候出现了一点小插曲，在输入命令之后出现了问题，返回为问题是https的协议有问题没有返回，估计是在之前的操作中无意修改了什么文件，因此需要安装https的证书。

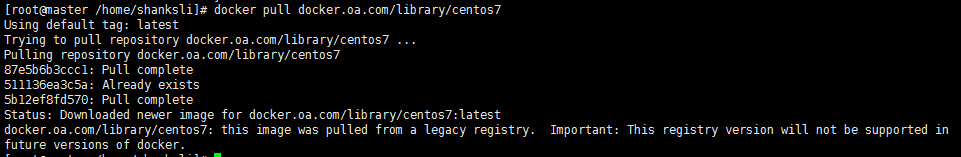


配置完成以后就可以使用docker search命令了：



1. 下载镜像

找到自己需要的镜像docker.oa.com/deeplearning/centos-6.5-x86\_64，然后使用docker pull命令就可以下载了：



我们可以很清晰的看到，下载的镜像是分层保存的。我们可以使用docker images看到刚才下载的镜像，给刚才的镜像加一个标签centos7，然后就可以使用这个镜像了。

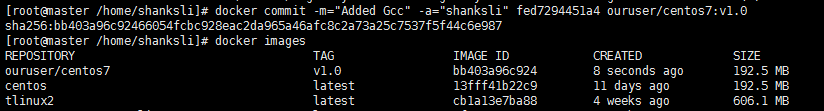
1. 创建自己的镜像

自己创建镜像有两种方法：①使用docker commit来扩展一个image，②从dockerfile来创建images。

①使用docker commit来扩展一个image

先使用images启动一个容器，添加应用程序后退出，然后使用docker commit命令提交相应的副本，然后就可以得到新的镜像，这种方法主要用在给镜像扩展功能上。

例如我们使用刚才的容器fed7294451a4给它安装GCC功能退出后，使用 docker commit -m="Added Gcc" -a="shanksli" fed7294451a4 ouruser/centos7:v1.0创建一个名为ouruser/centos7标签为v1.0的新镜像，使用docker ps -a可以查看：



②从dockerfile来创建images

使用docker commit来扩展一个image比较简单，但它不容易在一个团队中分享它。我们使用docker build 来创建一个新的image。首先需要创建一个dockerfile，包含一些如何创建我们的image的指令。例如我是想制作一个centos的镜像，首先我去docker hub的官网上下载centos的镜像，https://github.com/CentOS/sig-cloud-instance-images/blob/e37f28d8f4a2a5970ee9dec81d49d483ac1e4ae8/docker/centos-7-docker.tar.xz下载成功后，先把该文件上传到跳板机，然后再通过scp命令上传到服务器刚才新建的文件夹中，此处略去了把镜像上传到10.49.89.197：/home/shanksli/workspace/docker\_image下的步骤，然后进入服务器进入刚才的目录/home/shanksli/workspace/docker\_image，新建一个文件Dockerfile，之后编写dockerfile文件，dockerfile中添加的内容如下：

FROM scratch

ADD centos-7-docker.tar.xz /

LABEL name="CentOS Base Image" \

vendor="CentOS" \

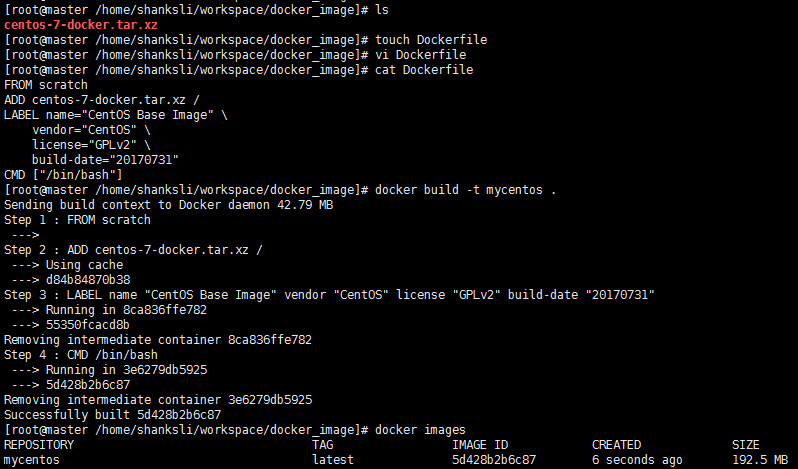
license="GPLv2" \

build-date="20170731"

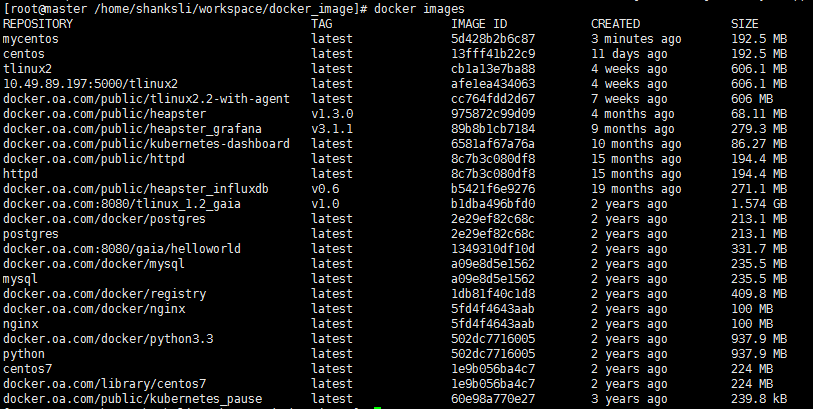
CMD ["/bin/bash"]

上面dockerfile中的内容**FROM**命令会指定镜像是基于哪一个基础镜像创建的，该命令可以使用多次，表示会创建多个镜像；**ADD**用于复制构建环境中的文件或目录到镜像中；**LABEL**用于为镜像添加元数据，元数以键值对的形式指定，包含了镜像的名称版本信息以及创建日期等；**CMD**用于指定在容器启动时所要执行的命令。

添加内容成功后保存退出，然后使用docker build命令创建新的镜像。截图从镜像放到文件下开始：



创建成功后使用docker images 就可以查看到我们刚才新建的镜像mycentos，



并且可以使用docker inspect查看镜像的详细信息：



也可以使用docker inspect -f 指定参数：



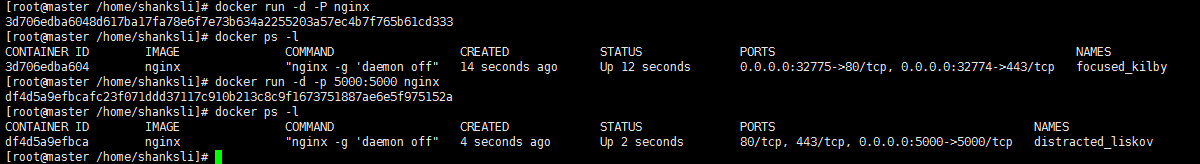
我们可以在上面的图中看到在build进程执行的过程中，它要做的第一件事情就是上传这个Dockerfile文件中的内容，因为所有的操作都是按照Dockerfile中的文件来执行的。我们看到Dockerfile中的指令是一条一条的执行的，每一步都创建一个容器，在容器中执行指令并且按照前面docker commit方法扩展一个镜像，当所有的命令都执行完毕之后，返回一个镜像的ID，并且把中间步骤产生的容器都删除掉。

比较新建的镜像和之前在docker.oa.com上下载的镜像大小可以看到docker hub上的镜像要比docker.oa.com上的镜像小，说明了docker hub上的镜像要好一些。

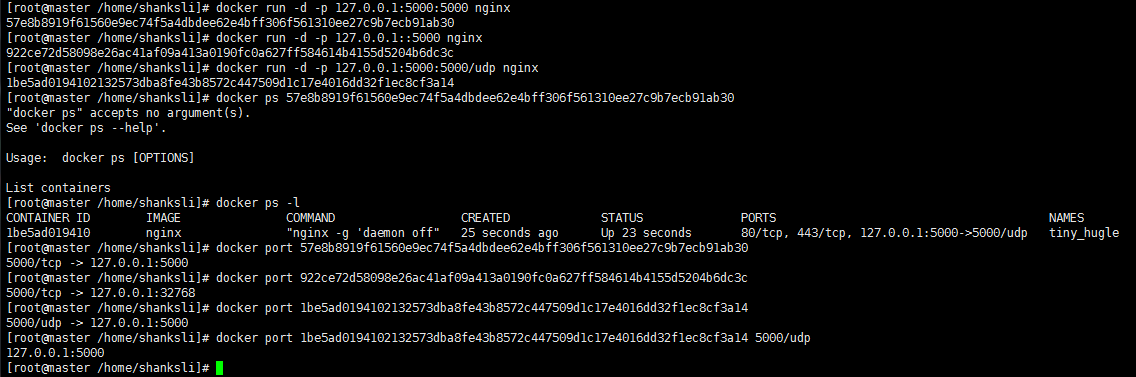
第五章 Docker中的网络介绍

1. 端口映射实现访问容器

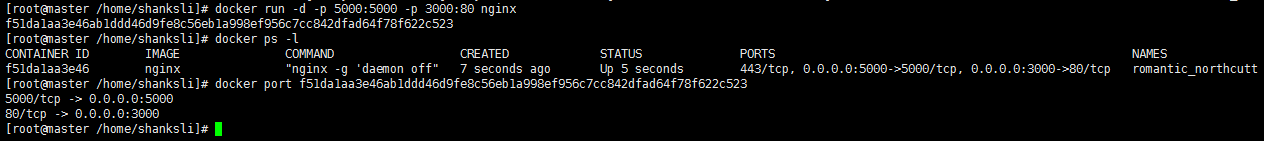
启动容器的时候，如果不指定对应的参数，在外部是无法通过网络来访问容器内的网络应用和服务的。当容器中需要运行一些网络应用，让外部访问这些应用时，可以使用-p或者-P参数来指定端口的映射。例如我们使用-P标记的时候，docker会随机的映射一个从49000到49900的端口到内部容器的端口上，可以使用docker ps -l查看当前容器端口映射情况；如果使用-p可以指定我们想要映射的端口，在一个端口上只可以绑定一个容器：



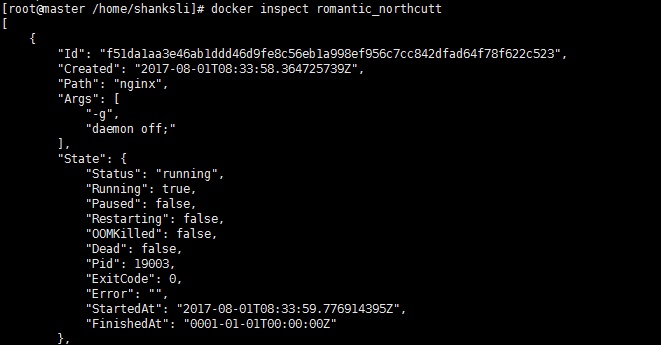
-p默认会绑定会绑定本地所欲的接口地址，我们在做端口映射的时候可以指定一个地址或者是或者是任意的端口到容器的5000端口，还可以上使用UDP标记指定UDP端口，可以使用docker port来查看当前绑定的端口配置，也可以查看绑定的地址：



也可以多次使用-p来绑定多个端口：



每一个容器都有自己的内部网络和IP地址，可以使用docker inspect查看所有的变量，可以查看到当前容器的内部IP地址等信息。



1. Docker容器互联

容器的互联机制（linking）是为了让多个容器中应用进行快速交互的方式，他会在源和接收容器之间创建连接关系，接收容器可以通过容器名快速访问到源容器，而不需要指定IP地址。

1. 容器的命名系统

Linking系统根据容器的名称来执行的，当我们创建容器的时候系统会随机的分配一个名字，我们也可以使用--name来自己命名，有两个好处：

①自定义的名字比较好记，比如一个web应用的容器，我们就起名字web，一目了然；

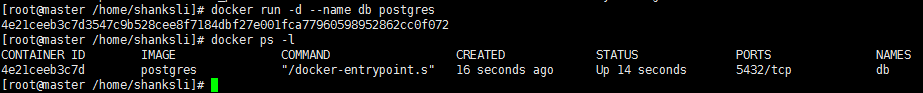
②连接到其他的容器的时候，即使是重启容器也可以使用容器名而不用改变。

注意：容器的命名是唯一的，如果系统中已经有一个同名的容器，就需要使用docker rm删除之前同名的容器。

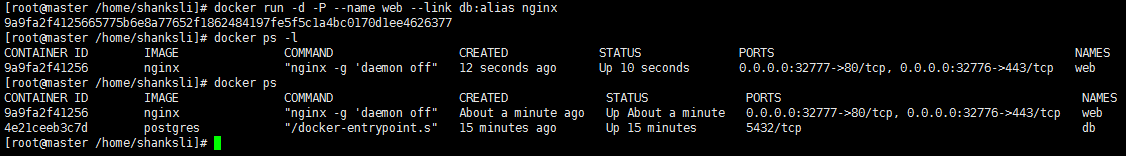
1. 容器互联

使用--link可以让容器之间安全的进行交互。

首先创建一个新的数据库的容器，给该容器命名为db：



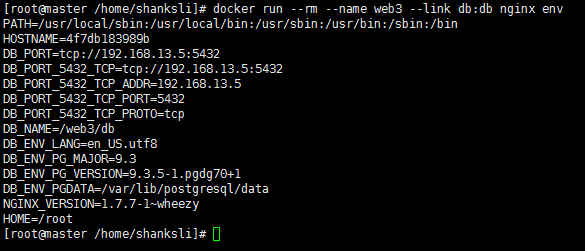
然后再创建一个web容器，并且把它连接到db容器上：



可以使用docker ps查看容器的连接情况。Docker相当于在两个互联的容器之间创建了一个虚拟的通道，而不需要映射它们的端口到宿主机上，在创建db容器的时候没有使用-p或者-P标记，使用link之后我们就可以不用暴露数据库容器的端口到网络上。

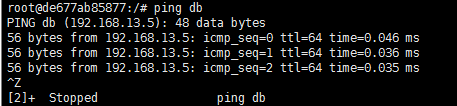
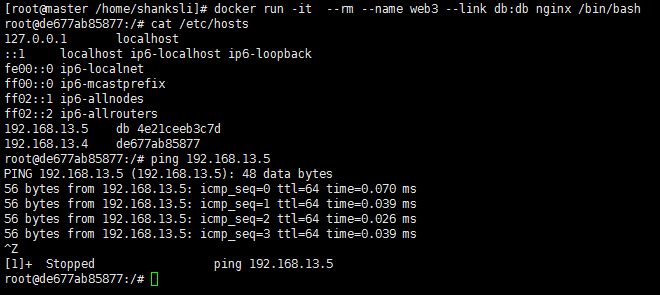
Docker通过两种方式公开容器的连接信息：

①更新环境变量，使用env方式查看容器的环境变量；



其中DB\_开头的的环境变量是提供web3容器连接db容器使用的。

②更新/etc/hosts文件，docker还添加host信息到父容器的/etc/hosts文件中，以下是一个父容器web的hosts文件：

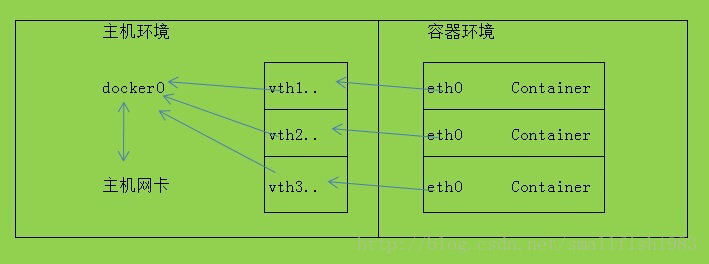


首先我们在hosts文件中可以看到两个IP，第一个是db容器的IP地址别名以及主机名，第二个是新建的web容器的IP地址和主机名，因为我们没有使用-p或者-P命令将db容器的端口暴露出来，我们可以使用ping命令验证在web容器中是否可以ping通db容器，我们在ping db的时候，它会把db的地址解析成192.168.13.5这与hosts文件中的一致。

注意：可以链接多个子容器到父容器，比如我们可以链接多个web到db容器上。

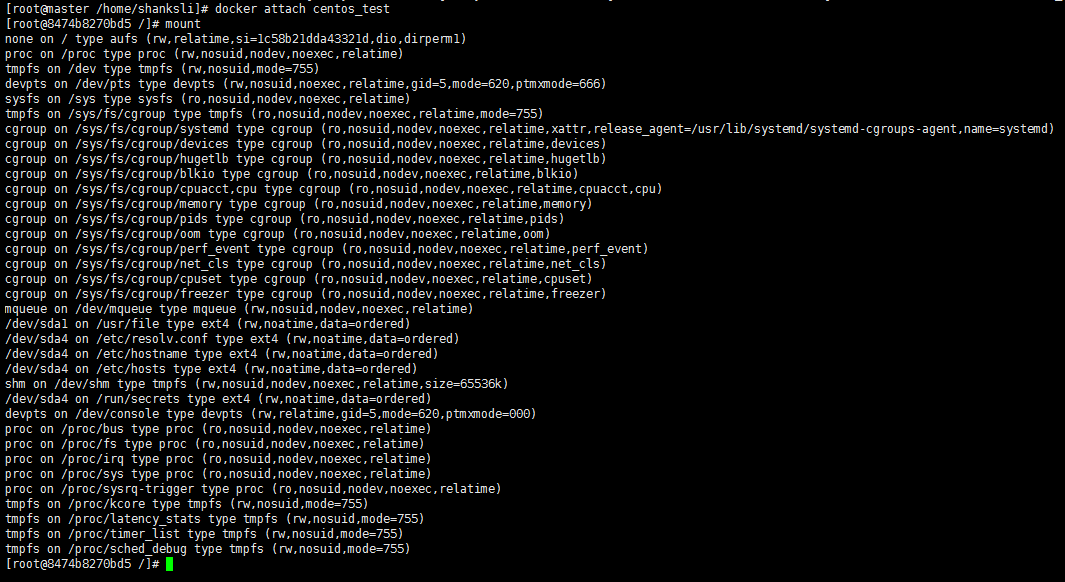
第六章 Docker高级网络配置

当docker启动的时候，会在主机上创建一个docekr0的虚拟网卡，docker0的网卡不是普通的网卡，他是桥接到其他网卡的虚拟机网卡，容器需要使用他来和主机进行通信。当创建一个docker容器的时候，它就创建了一个对接口，当数据包发送到一个接口时，另外一个接口也可以收到相同的数据包，它们是绑在一起的一对孪生接口。这对接口在容器中那一端的的名字是eth0，宿主主机端的会指定一个唯一的名字，比如vethAQI2QT这样的名字，这种接口名字不再主机的命名空间中。所有的veth的接口都会桥接到docker0，这样docker就创建了在主机和所有容器之间一个虚拟共享网络。



1. 配置DNS

Docker没有给每一个容器定制镜像，，但是容器会有主机的主机名和DNS配置信息，他主要是使用主机上的配置文件来覆盖容器的配置文件，在容器中使用mount命令就可以看到：



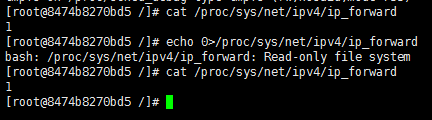
这样的话可以在宿主主机更改了dns信息后，马上可以更新docker的dns配置信息。

1. 容器之间的通信

判断两个容器是否可以通信，在操作系统层方面，取决于3个因素：

①网络拓扑是否连接到容器的网络端口，默认的情况下是docker会将所有的容器都连接到docker0这一个网桥来提供数据包通信。

②主机是否开启了ip转发，需要查看ip\_forward参数是否为1，为1的时候可以提供数据包的转发，通常docker设定ip\_forward=true,docker就会在启动的时候设定ip\_forward参数为1，可以自己手动检查并且设置参数。



可以使用cat /proc/sys/net/ipv4/ip\_forward命令来查看ip\_forward的当前值，当我使用echo 0>proc/sys/net/ipv4/ip\_forward命令把ip\_forward值修改为0的时候，提示我/proc/sys/net/ipv4/ip\_forward是一个只读文件，因此不可以修改该文件的值。

③iptables（iptables，一个运行在[用户空间](https://zh.wikipedia.org/wiki/%E4%BD%BF%E7%94%A8%E8%80%85%E7%A9%BA%E9%96%93)的应用软件，通过控制[Linux内核](https://zh.wikipedia.org/wiki/Linux%E5%85%A7%E6%A0%B8)[netfilter](https://zh.wikipedia.org/wiki/Netfilter)模块，来管理网络数据包的流动与转送。在大部分的Linux系统上面，iptables是使用/usr/sbin/iptables来操作）是否允许建立特殊的连接，当docker设定--iptables=false的时候，docker不会改变系统的iptables设定，否则它会在--ice=true的时候添加一条默认的ACCERT策略到forward链，基本所有人都会开启ip\_forward来启用容器间的通信。

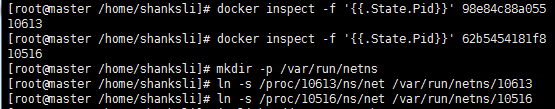
1. 创建一个点到点的连接

默认docker会将所有容器连接到由docker0提供的虚拟子网，你也可以使用自己创建的网桥。但如果你想要2个特殊的容器之间可以直连通信，而不用去配置复杂的主机网卡桥接。解决办法很简单：创建一对接口，把2个容器放到这对接口中，配置成点到点链路类型。这2个容器就可以直接通信了。配置如下：

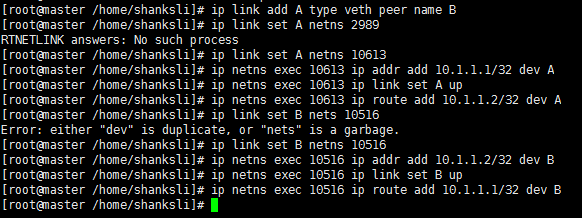
①在2个终端中启动2个容器



②找到他们的process IDs ，然后创建他们的名称空间namespace entries



③创建管道peer接口，并且给每一个容器配置路由信息



④接下来就可以在各自的容器内部ping通互联的容器，因为是点到点的链路不需要子网和子网掩码，使用ip route来连接单个ip地址到指定的网络接口。

1. Docker的4种网络模式

我们在使用docker run创建Docker容器时，可以用--net选项指定容器的网络模式，Docker有以下4种网络模式：

①host模式，使用--net=host指定。

②container模式，使用--net=container:NAME\_or\_ID指定。

③none模式，使用--net=none指定。

④bridge模式，使用--net=bridge指定，默认设置。

⑴host模式

Docker使用了Linux的Namespaces技术来进行资源隔离，如PID Namespace隔离进程，Mount Namespace隔离文件系统，Network Namespace隔离网络等。一个Network Namespace提供了一份独立的网络环境，包括网卡、路由、Iptable规则等都与其他的Network Namespace隔离。一个Docker容器一般会分配一个独立的Network Namespace。但如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。

例如，我们在10.10.101.105/24的机器上用host模式启动一个含有web应用的Docker容器，监听tcp80端口。当我们在容器中执行任何类似ifconfig命令查看网络环境时，看到的都是宿主机上的信息。而外界访问容器中的应用，则直接使用10.10.101.105:80即可，不用任何NAT转换，就如直接跑在宿主机中一样。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。

⑵container模式

这个模式指定新创建的容器和已经存在的一个容器共享一个Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的IP，而是和一个指定的容器共享IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的，两个容器的进程可以通过网卡设备通信。

⑶none模式

在这种模式下，Docker容器拥有自己的Network Namespace，但是，并不为Docker容器进行任何网络配置。也就是说，这个Docker容器没有网卡、IP、路由等信息，需要我们自己为Docker容器添加网卡、配置IP等。

⑷bridge模式

bridge模式是Docker默认的网络设置，此模式会为每一个容器分配Network Namespace、设置IP等，并将一个主机上的Docker容器连接到一个虚拟网桥上。下面着重介绍一下此模式。

第七章 Docker的数据管理

在docker容器中以及容器之间的数据管理的主要方式有两种：数据卷和数据卷容器。数据卷是把容器内的数据直接映射到本地的主机环境；数据卷容器是使用特定的容器来维护数据卷。

1. 数据卷

数据卷是一个由UFS文件系统专门设计的的特殊目录，可以提供很多有用的特性：

①数据卷可以在容器之间共享和重用；

②对数据卷的改变是立马生效；

③当你更新数据卷中的数据的时候，不会被包含到image中；

④卷会一直存在直到没有容器使用他们。

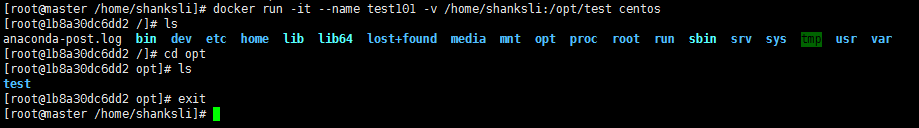
1. 添加一个数据卷

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



1. 添加一个主机目录作为数据卷

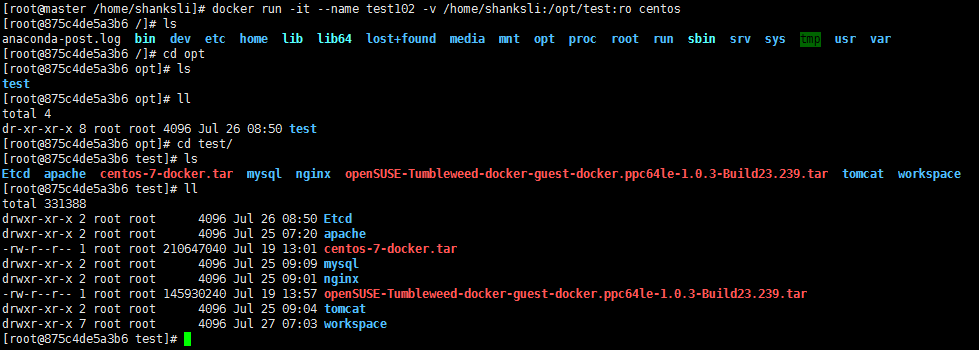
使用-v标记也可以挂载一个主机的目录到容器中去



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

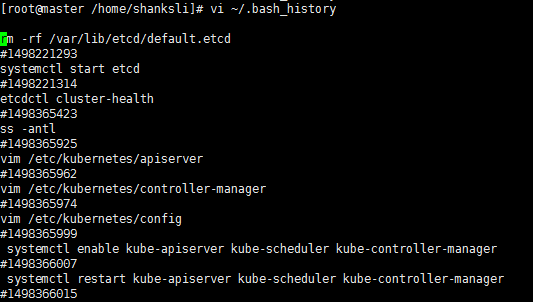
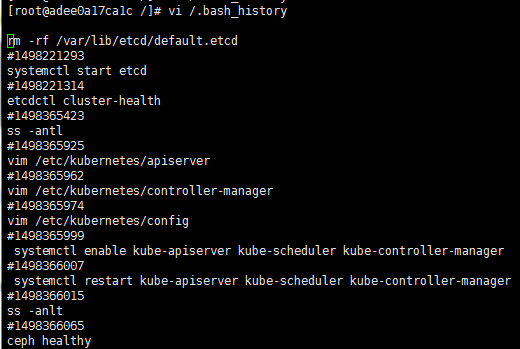
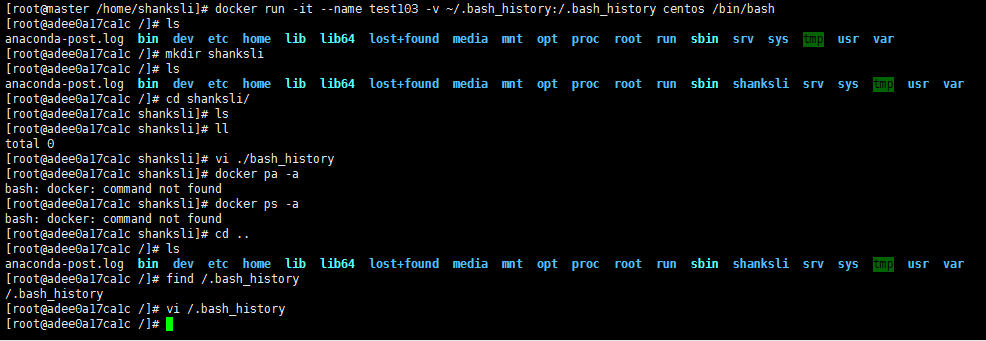
docker 加载的数据卷默认是读写权限，但我们可以把它加载为只读。docker

加载的数据卷默认是读写权限，但我们可以把它加载为只读。



1. 挂在一个宿主主机文件作为数据卷

-v标记也可以从主机挂载单个文件到容器中。



使用上面的命令把 ~/.bash\_history文件挂在到容器中/.bash\_history文

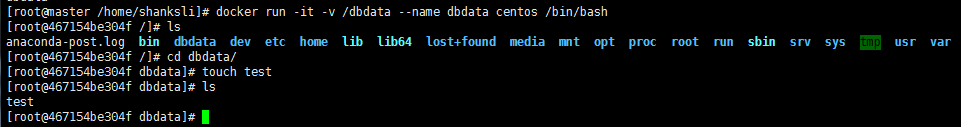
件，使得在容器中使用的命令在主机上都可以看到。

1. 数据卷容器

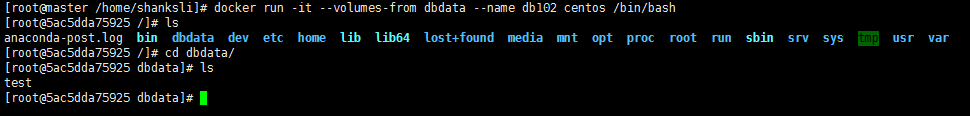
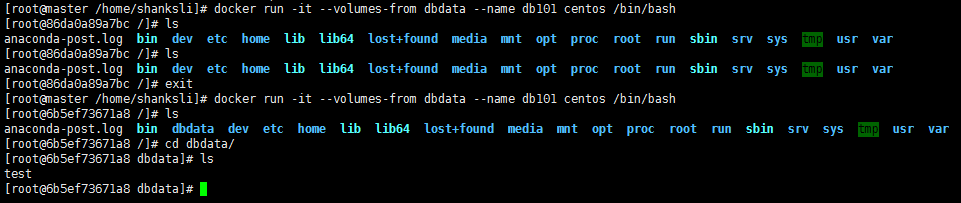
如果你有一些持续更新的数据需要在容器之间共享，最好创建Data Volume Container，然后加载它。现在就来创建一个命名的数据卷容器：

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

①首先创建一个数据卷容器dbdata，并且在容器里面创建一个数据卷挂在到/dbdata：



②然后可以在其他容器中使用--volumes-from来挂载dbdata容器中的数据卷，例如创建db101和db102两个容器，并且从dbdata中挂载数据卷：



③这个时候db101和db102都挂载同一个数据卷到相同的/dbtata目录下，三个容器只要一个容器写入数据，其他容器都会发生变化，可以使用--volumes-from挂载多个数据卷。

可以利用数据卷容器对其中的数据卷进行备份、恢复，以实现数据数据的迁移。

第八章 容器的安全性能

第九章 学会部署局域网docker

第十章 Docker实现多台物理机之间的容器互联

第十一章 Docker中使用Supervisor管理进程

第十二章 Docker创建tomcat/weblogic集群

第十三章 Docker安装Redis

输入docker search redis

然后选择你需要的版本 如：docker pull redis:5.0

下载完后输入docker run -d -p 6379:6379 redis:5.0

启动服务

接着就可以直接用客户端连接redis啦

Docker学习总结

写这篇学习总结主要包含了两个层面的意义:第一方面是为了把最近学习docker相关的内容做一个总结,便于在以查看相关的材料,方便学习和总结,这也是主要的一方面;第二方面是为了完成导师安排的任务。

本文主要是参考docker学习手册的内容完成的，目前计划主要主要包括十二章的内容：第一章为什么需要使用docekr；第二章docker的体系结构；第三章docker的安装和使用；第四章docker images的获取和使用；第五章docker中的网络介绍；第六章docker高级网络配置；第七章docker的数据管理；第八章容器的安全性能；第九章学会部署局域网docker；第十章docker实现多台物理机之间的容器互联；第十一章在docker中使用Supervisor管理进程；第十二章docker创建tomcat/weblogic集群。这篇总结是根据自己在学习docker过程中遇到的一些问题和实际的操作展开的，很多的内容都是自己理解的，如果有认识上的错误，请指正。

注意：本文中我自己认为需要注意的地方会使用黄色标记出来。

第十四章 Docker安装MySQL

从docker检索mysql镜像：docker search mysql

从镜像中下载：docker pull mysql 可以指定版本:mysql:5.7

查看镜像：docker images

启动mysql镜像：docker run --restart=always --name first-mysql -p 3306:3306 -eMYSQL\\_ROOT\\_PASSWORD=123456 -d mysql:latest

启动mysql服务输入docker start containerID（可以输入docker ps -a查看镜像的containerID）

输入docker exec -it mysql（containerID） bash进入容器

接着输入mysql -uroot -p可以进入MySQL服务;

开放3306端口就可以使用工具连接Mysql啦。

如果你安装的mysql8.0连接的时候会出现Authentication plugin ‘caching\_sha2\_password’ cannot be loaded;

第十五章 基于Docker的Azure DevOps实现

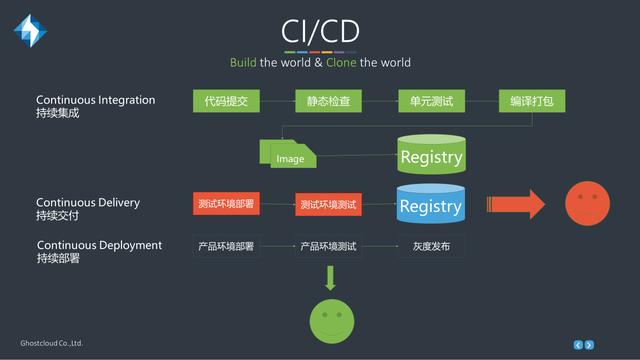
众所周知，传统开发模式已经面临了诸多难题。首先，在代码集成方面，因为没有合适粒度的代码合并，大规模的合并会有很大的风险，且传统开发模式中没有自动化测试，以至于测试周期特别长，人力成本高昂。其次，传统开发中的单体应用，通常都很庞大，单体应用把所有模块都包含在一个应用中，升级单个模块也需要对整个应用进行升级，所以升级和创新都很不方便，常见的比如银行系统就是如此。

同时，传统的开发模式中单独采用微服务的情况也会由于服务数量多而没有有效管理，在大批量的部署和测试的时候容易出现问题。除此之外，传统开发模式更面临着开发和测试的环境不一致，以及由于没有有效的升级方式导致业务停止的问题。



（如何一步步实现DevOps）

为了解决传统开发模式中的问题，目前一个比较流行和彻底的方案是：DevOps流程+微服务理论+使用容器和容器编排工具。在这里展示给大家的是一个理论上的基于容器的CI/CD流程，实际上，DevOps的前身就是CI/CD，实现了CI/CD后，再加上一些发布、部署等标准和管理就构成了DevOps。



（基于容器的CI/CD流程）

实现DevOps之自动化测试

那么如何来完整的实现DevOps呢？通常情况下，传统开发模式转向DevOps的第一步是解决自动化问题。要想持续的集成代码，没有自动化测试来保证快速地进行合并后的验证，风险是很高的，而且没有自动化测试，测试环境很有可能成为整个开发环节的瓶颈。只要是经常使用的测试用例，需要尽量自动化每一个操作。

自动化工具很多，对自动化工具和测试框架的选择是需要根据具体应用来决定的，这里只列举其中常用的一小部分——Jenkins、Python、Robot Framework、Shell Script、Selenium、Ansible和Docker Container Orchestration——这些都是我们面对客户需求的时候经常用到的。然而，不是每次集成都需要跑完所有的测试用例，因而对测试用例进行管理，可提高持续集成的效率。



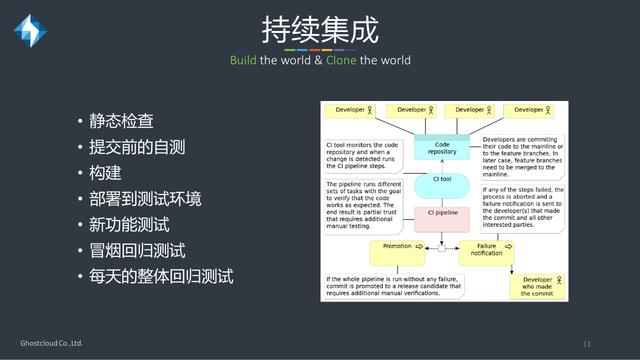
（自动化测试）

如何来判断自动化测试用例和框架是否有效？常见的判断依据有三个，首先是自动化测试的覆盖率。如果通过率再高，覆盖率低，那么自动化测试就不是一个有效的，目前企业级比较认可的覆盖率是75%左右，再提高也比较困难。其次是看漏测率，有时候自动化用例本身也可能有Bug，前期阶段通过比较手动测试自动化测试的结果来判断自动化测试是否有效。最后，当产品发布后根据从客户来源的bug数目来判断自动化测试用例是否有效。另外，要稳定一套自动化用例，一般需要2个版本周期或者更长。

实现DevOps之持续集成和持续交付

持续集成一个主要的功能是让每个工程师的代码提交都不会影响到Mainline，以保证Mainline的可发布状态。实施持续集成时，需要注意的地方：

1. 指定规则，提交代码时要一并提交新功能的测试用例。
2. 集成的粒度和频度也很关键。一般一个小模块，不超过1周的时间。



（持续集成）

持续集成通过后，根据应用程序的特点，在经过系统集成测试、性能测试、稳定的自动化测试通过率以及管理层的批准后，才是可持续交付和部署的应用程序。

持续交付有两种方式，一种就是基于DevOps的自动持续发布，一种是多个功能一并发布。在持续交付的过程中需要注意三个问题：

1.部署到生产环境后也要有相应的测试；

2.使用Toggle控制功能是否生效；

3.要有回滚的手段（灰度发布）。

实现DevOps之微服务化

有了自动化测试、持续集成和持续交付三块，已经基本实现了DevOps的粗略流程，而为了提高DevOps的效率， 往往需要结合微服务。一个微服务理论上只做一件事，并能用任何语言编写。微服务是松耦合的，意味着一个应用的微服务可以被部署到不同机器上并通过resAPI/RPI来通信，当定义好微服务的API之后，每个team便能独立开发。因此，微服务更容易被测试和实现CI/CD。



（微服务的最佳实践）

在微服务的最佳实践中，首先不得不提容器。容器的轻量化让微服务启动很快，同时容器的跨平台性保证了微服务可以在不同的平台启动起来。第二种是使用代理服务器来访问微服务，现在最常见的方式是前端连接一个代理服务器，后端再连接运行同一个微服务的几个相同容器。一个大的应用会使用几十上百个微服务，和微服务不相关的库文件不建议放在容器中。实践微服务中，建议使用配置管理工具（ansible, puppet等）和容器服务编排工具（K8s，Swarm，EcOS等）。

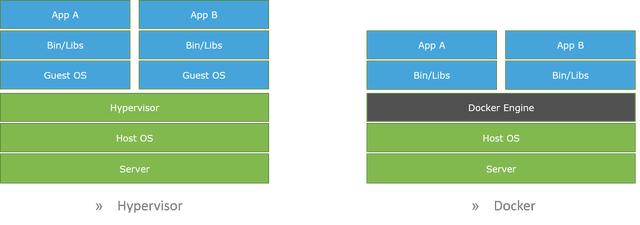


（康威定律）

在开发微服务中康威定律起到了很大的作用。康威定律指出任何软件代码都是用来反映组织机构而产生的，如果要采用微服务的开发方法，就需要是把团队划分成多个小团队，由每个小团队负责一个或多个微服务。所以如果要转成DevOps和CI/CD的开发模式，就需要采用这种敏捷开发模式，一个团队7-8个人比较合适。

实现DevOps之容器技术

另外一个实现DevOps的重要手段是Docker容器技术。和传统的Hypervisor相比，Docker没有自己的操作系统，它使用宿主机的操作系统，而Hypervisor需要建立虚拟机，每个虚拟机需要装一个操作系统，因此Docker效率更高更节约资源。如果一台物理机可以操作20个虚拟机，便至少可以启动200个容器，且启动容器的时间是秒级。



（Docker和Hypervisor的对比）

使用容器编排工具可以实现对容器的健康检查、动态伸缩、灰度发布和蓝绿发布等功能。而我们提到的容器编排技术，比如K8s，Mesos和Swarm，都是开源的工具，这里我们把精灵云自研的容器编排工具EcOS和开源工具进行了简单对比。



（几种常见容器编排技术的比较）

K8s是由谷歌发起的开源框架，最大的问题是太笨重，对使用者来说操作很复杂，学习周期很长。Swarm是Docker公司开发的工具，Docker本身不能支持的功能，Swarm也是无法支持的。如图所示，EcOS是精灵云自主开发的容器编排技术，最大的特点是结合了开源工具的优点，在应用编排上完全可视化。EcOS内置的自研调度框架Newben，在网络、应用迁移、负载均衡、弹性伸缩、调度规则等方面也比开源框架有比较大的优势，在支撑过数万用户和数十万的容器调度考验后，Newben是目前国内定制能力最强，支撑功能最丰富的调度框架。

实现DevOps之灰度发布

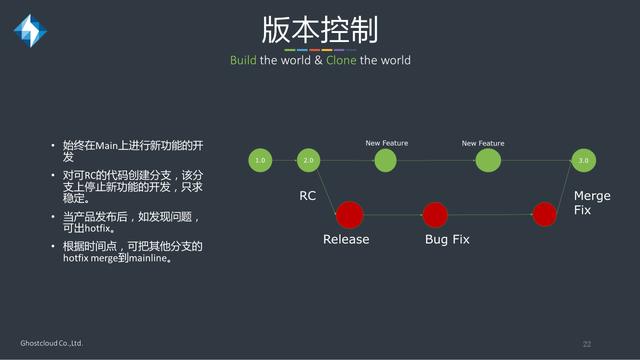
如果一个服务由多个相同的容器运行，灰度发布则先对其中的部分容器先进行升级，可混合让老版本和新版本的容器同时提供服务。如发现新服务没有什么问题，则可以把所有剩下的微服务再全部进行升级。



（灰度发布）

实现DevOps之版本控制

DevOps下版本控制的原则是始终在Mainline上进行新功能的开发，并经由持续集成的自动化测试对代码进行验证。当功能开发到一定阶段的时候，对可RC的代码创建分支，该分支上停止新功能的开发，只求稳定。当产品发布后，如发现问题，可出hotfix。根据时间点和具体需要，可把其他分支的hotfix merge到Mainline上。



（版本控制原理）

1. Docker和持续集成（CI）

**什么是持续集成？**

我们先得了解持续集成的相关概念，才能更好地指导开发和使用Docker来改进我们的工作流。和其他教程不一样，笔者更喜欢将必要的知识点围绕理论、流程（工作流程）、方法、实践来进行讲解，而不是单纯的为讲解知识点而进行讲解。也就是说，笔者希望为大家打通任督二脉，能够将理论、知识、思想和指导应用到工作的实际场景和实践之中，而不是拿着字典写文章，抱着宝典写代码。至于很多具体的语法、技术细节，除了常用的知识点，笔者更希望大家阅读官方文档——毕竟看官网比看书靠谱多了，官网会一直更新和改进，而书和教程自出版或发布之后，基本上就“死“了。

好了，我们回到正题。持续集成是一种软件开发实践，即团队开发成员经常集成他们的工作，通常每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建（包括编译，发布，自动化测试）来验证，从而尽早地发现集成错误。

徒弟一脸崇拜道：“师父，为什么我做出来的飞剑，一念咒语不是碎了就是爆了呢？”。

师父摸了摸胡子道：“徒儿莫急，冰冻三尺非一日之寒！为师我刻了3年的阵法，练习了3年的咒语，然后又花了3年一起练习，才让第一把飞剑飞上了太空。我看你天资聪慧，顶多20年就够了”。

2年后，徒弟边刻阵法边念咒，突然飞剑的剑身嗖的一下不见了，只余剑柄。

师父：“徒儿，你的飞剑怎么飞了一截出去了！”

徒弟握着剑柄行礼道：“师父勿怪，这段时间我对飞剑的制作过程进行了改良，一边刻阵法一边念咒，现在我对阵法和咒语的掌控都达到了70%，所以只有前半截飞出去了！“

**注意：集成软件的过程不是新问题，如果项目开发的规模比较小，比如一个人的项目，如果它对外部系统的依赖很小，那么软件集成不是问题，但是随着软件项目复杂度的增加（即使增加一个人），就会对集成和确保软件组件能够在一起工作提出了更多的要求-要早集成，常集成。早集成，频繁的集成帮助项目在早期发现项目风险和质量问题，如果到后期才发现这些问题，解决问题代价很大，很有可能导致项目延期或者项目失败。**

**核心价值**



要素

1.统一的代码库

2.自动构建

3.自动测试

4.每个人每天都要向代码库主干提交代码

5.每次代码递交后都会在持续集成服务器上触发一次构建

6.保证快速构建

7.模拟生产环境的自动测试

8.每个人都可以很容易的获取最新可执行的应用程序

9.每个人都清楚正在发生的状况

10.自动化的部署

原则

1. 所有的开发人员需要在本地机器上做本地构建，然后再提交的版本控制库中，从而确保他们的变更不会导致持续集成失败。

2. 开发人员每天至少向版本控制库中提交一次代码。

3. 开发人员每天至少需要从版本控制库中更新一次代码到本地机器。

4. 需要有专门的集成服务器来执行集成构建,每天要执行多次构建。

5. 每次构建都要100%通过。

6. 每次构建都可以生成可发布的产品。

7. 修复失败的构建是优先级最高的事情。

8. 测试是未来，未来是测试

持续集成我们就先说到这里，建议大家也可以了解下敏捷开发，毕竟持续集成是敏捷开发的基石，但是敏捷开发是一个大命题，这里我们顺带提一下，然后我们还是先继续本篇教程：



师父：“徒儿，你真的在短短3年就让飞剑飞起来了？”。

徒弟：“弟子愚钝，在刻剑的过程中倍觉无聊，又不喜欢哼歌，于是索性边练咒边刻剑。后面徒儿发现，如果刻错了或者念错了，飞剑就会提前直接爆炸，虽然每次炸的内裤都没了，但是能够尽早发现错误。所以徒弟才能一日千里”。

师父摸了摸胡须道：“然来如此！不过，这就是你大庭广众之下裸奔的借口！！？”

相比其他技术，Docker在持续集成（CI）这块有着先天的优势。在通常的情况下，我们要实现持续集成往往会遇到以下问题：

l **复杂的依赖关系**

不同的项目环境，不同的语言，不同的程序包依赖，甚至是操作系统的依赖等等，都会影响到我们持续集成的自动化脚本的执行。而且依赖包之间的兼容性，版本的兼容性，间接依赖或者多重依赖等问题等等，对于开发和运维来说，都是一个噩梦。就如以下对话：

徒弟：“师父，我按照您教的方式念咒，为什么飞剑飞起来了之后就收不回来了？”。

师父直接一巴掌，说：“兔崽子，上次就和你说了，咒语现在最低的兼容级别是——普通话二级乙等！谁教你说长沙话的！”

l **不一致的环境**

在通常的环境中，我们需要准备好开发、测试和生产环境，往往开发环境随便开发人员折腾，有时候操作系统或者依赖软件的版本的区别、组件的不同、配置不一样，都足够让开发环境正常运行的程序在测试环境上跑不起来，造成测试人员和开发人员的故意伤害事件，导致“行凶人员”后悔终生，感悟到“冲动就是魔鬼”的箴言。我们还是以对话来阐述这个问题：

徒弟拿出普通话二级乙等证书道：“师父，我苦学普通话，终于达到普通话二级乙等。然后按照您教的方式念咒了，之后为什么飞剑飞起来了之后还是没法收回来？”。

师父又是一巴掌，说：“兔崽子，你没看到下雨了么？”

徒弟弱弱的问：“这个和下雨有关系么？是不是雨天法术受雨滴干扰，咒语的效果受到影响呢？”

师父指着外面道：“瞎了？你丫的不赶紧把被子收回来烘干，你的飞剑就甭想要了！”

l **应用架构的复杂性和配置的多样性**

现在的系统架构越来越复杂，甚至由多种开发语言组成，而且包含前后端等多方面内容。这些可能会导致其部署方式的不同以及配置的复杂性。并且一个系统维护到后面，往往有很多历史遗留问题，比如那各种配置文件和配置方式，各种补丁，各种脚本等等。这些因素会导致自动化流程会非常麻烦和艰难。我们继续来一段对话：

徒弟：“师父，被子收好了，但是飞剑越飞越远了，是不是可以教我收回我的飞剑啦！”。

师父张开一只眼：“小崽子，普通话念完后，用长沙话再念一遍收剑咒！前几天，为师对收剑咒又进行了改造。”

徒弟用长沙话念完，飞剑还是再天空中乱窜，并没有降下来的意思。徒弟赶紧问道：“师父，为啥还是不行呢？”

师父弹了弹手指，远处一根若隐若现的细线展现出来，师父指着那根线说：“看到那边那根线没？还不赶紧去追！”

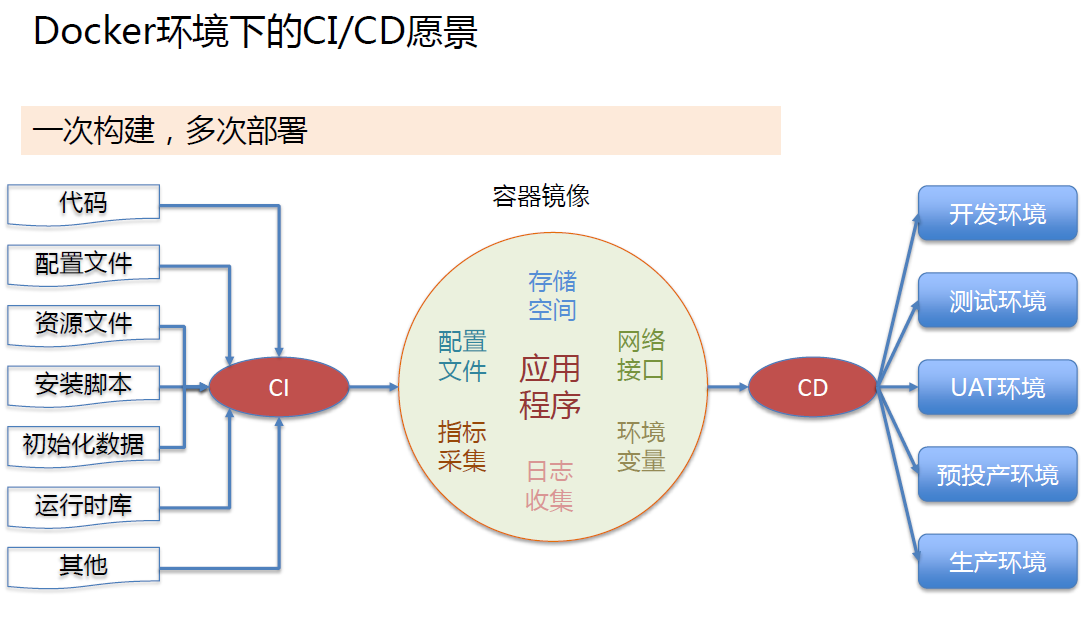
相比这些问题，Docker实现持续集成（CI）就方便多了。

首先，Docker可以让我们非常容易和方便地以“容器化”的方式去部署应用。它就像集装箱一样，打包了所有依赖，再在其他服务器上部署很容易，不至于换服务器后发现各种配置文件散落一地，这样就解决了编译时依赖和运行时依赖的问题。

其次，Docker的隔离性使得应用在运行时就像处于沙箱中，每个应用都认为自己是在系统中唯一运行的程序，这样就可以很方便地在一个系统中部署多种不同环境来解决依赖复杂度的问题。

正因为Docker是以应用为中心，镜像中打包了应用及应用所需的环境，一次构建，处处运行。这种特性完美解决了传统模式下应用迁移后面临的环境不一致问题。

因此使用Docker实现持续集成，我们可以使用一些简单的免费的工具即可实现，也可以非常方便的自己搭建集成环境或者编写脚本实现。比如Azure DevOps、Tencent Hub、Jenkins和TeamCity，接下来我们会逐步进行介绍。

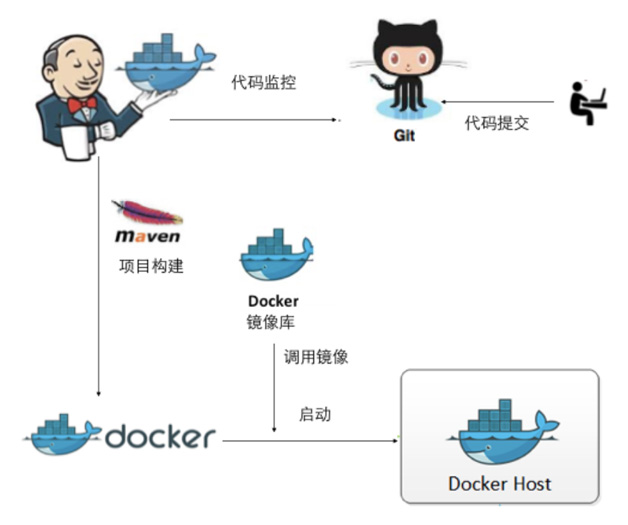


持续集成工作流程

一般情况下，持续集成的流程如下所示：



下面是一个参考流程：



代码版本管理，我们推荐使用Git。关于git版本库的使用，我这里就不啰嗦了，如果有朋友感兴趣，我也可以分享一些内容。

后续，我们将会分享使用相关工具来实施我们的CI流程。

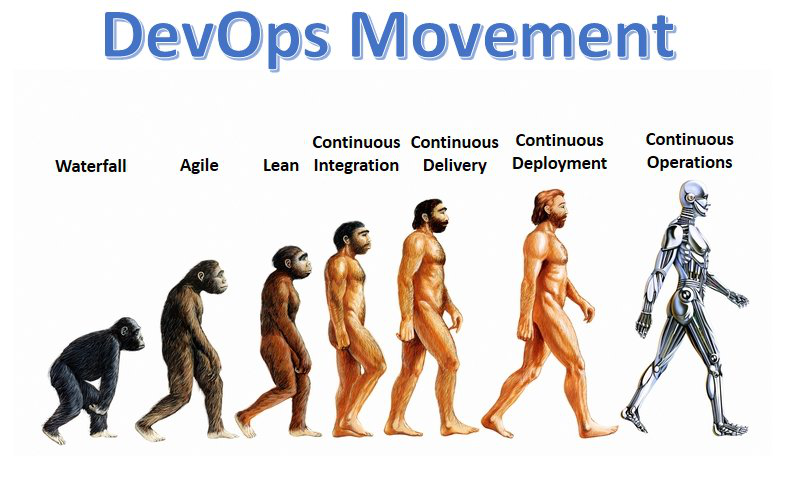
1. 使用Azure DevOps来完成CI

Azure DevOps，以前叫VSTS，现在被微软改名部正式更名为Azure DevOps。和VSTS一样，微软都提供了免费的使用额度，对于小团队和个人开发者来说，完全是足够了。

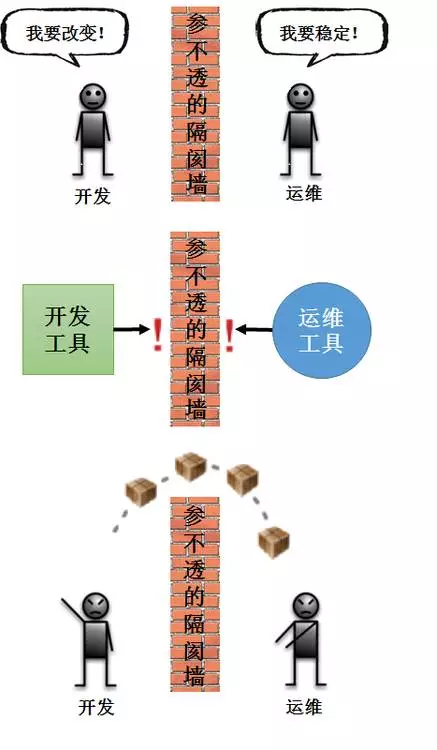


**什么是DevOps？**

DevOps（Development和Operations的组合词）是一组过程、方法与系统的统称，用于促进开发（应用程序/软件工程）、技术运营和质量保障（QA）部门之间的沟通、协作与整合。它是一种重视“软件开发人员（Dev）”和“IT运维技术人员（Ops）”之间沟通合作的文化、运动或惯例。透过自动化“软件交付”和“架构变更”的流程，来使得构建、测试、发布软件能够更加地快捷、频繁和可靠。它的出现是由于软件行业日益清晰地认识到：**为了按时交付软件产品和服务，开发和运营工作必须紧密合作。**



DevOps的引入能对产品交付、测试、功能开发和维护（包括──曾经罕见但如今已屡见不鲜的──“热补丁”）起到意义深远的影响。在缺乏DevOps能力的组织中，开发与运营之间存在着信息“鸿沟”──例如运营人员要求更好的可靠性和安全性，开发人员则希望基础设施响应更快，而业务用户的需求则是更快地将更多的特性发布给最终用户使用。这种信息鸿沟就是最常出问题的地方。



DevOps经常被描述为“开发团队与运营团队之间更具协作性、更高效的关系”。由于团队间协作关系的改善，整个组织的效率因此得到提升，伴随频繁变化而来的生产环境的风险也能得到降低。

总之，通过DevOps，各专业团队之间的协调和协作得到改善，缩短了将更改提交到系统与将更改投入到生产之间的时间。它还可确保此过程符合安全性和可靠性标准。结果：产品质量改善、交付速度加快、客户满意度提升。

**DevOps对应用程序发布的影响**

在很多企业中，应用程序发布是一项涉及多个团队、压力很大、风险很高的活动。然而在具备DevOps能力的组织中，应用程序发布的风险很低，原因如下：

1. **减少变更范围**

与传统的瀑布式开发模型相比，采用敏捷或迭代式开发意味着更频繁的发布、每次发布包含的变化更少。由于部署经常进行，因此每次部署不会对生产系统造成巨大影响，应用程序会以平滑的速率逐渐生长。

1. **加强发布协调**

靠强有力的发布协调人来弥合开发与运营之间的技能鸿沟和沟通鸿沟；采用电子数据表、电话会议、即时消息、企业门户（wiki、sharepoint）等协作工具来确保所有相关人员理解变更的内容并全力合作。

1. **自动化**

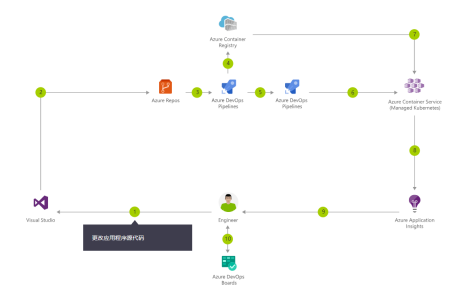
强大的部署自动化手段确保部署任务的可重复性、减少部署出错的可能性。

与传统开发方法那种大规模的、不频繁的发布（通常以“季度”或“年”为单位）相比，敏捷方法大大提升了发布频率（通常以“天”或“周”为单位）。减少变更范围与传统的瀑布式开发模型相比，采用敏捷或迭代式开发意味着更频繁的发布、每次发布包含的变化更少。由于部署经常进行，因此每次部署不会对生产系统造成巨大影响，应用程序会以平滑的速率逐渐生长。加强发布协调靠强有力的发布协调人来弥合开发与运营之间的技能鸿沟和沟通鸿沟；采用电子数据表、电话会议、即时消息、企业门户（wiki、sharepoint）等协作工具来确保所有相关人员理解变更的内容并全力合作。强大的自动化部署手段能够确保部署任务的可重复性、减少部署出错的可能性。

**适用于容器的 CI/CD 流程**

使用容器，可轻松地持续生成和部署应用程序。

Azure DevOps 可以通过设置持续版本以生成容器映像和业务流程，让我们能更快、更可靠地进行部署。以下是一个适用于容器和Azure的CI/CD 流程：



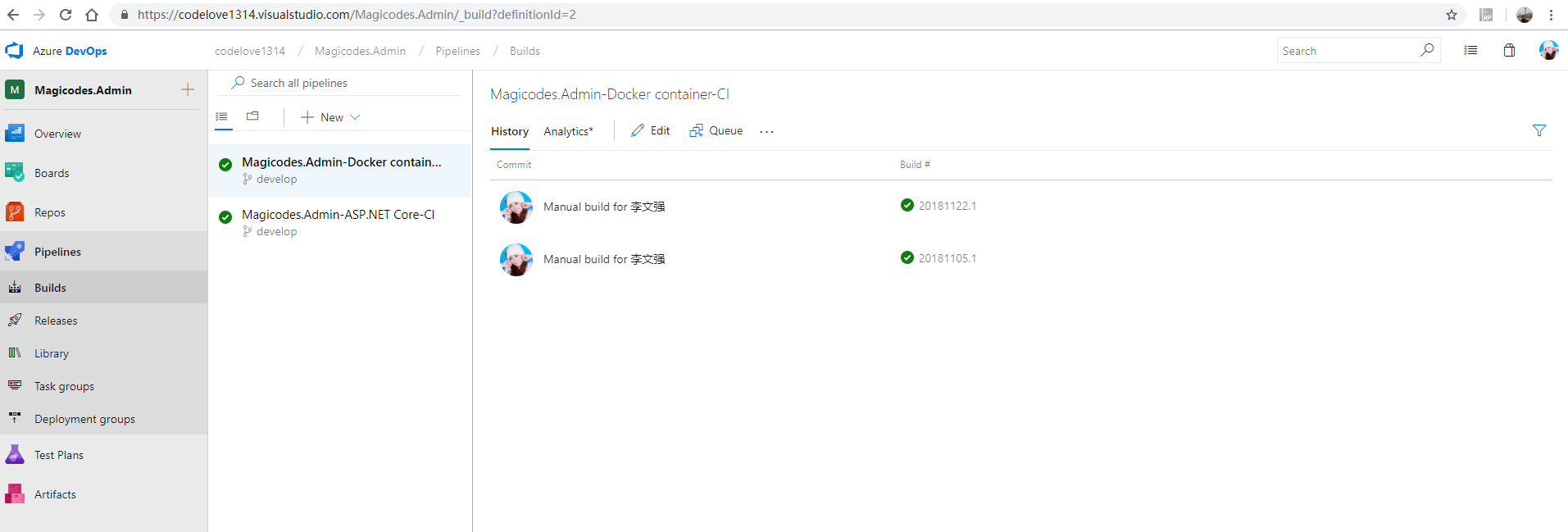
步骤说明：



**使用Azure DevOps来配置一个简单的CI流程**

Azure DevOps服务涵盖了整个开发生命周期，可帮助开发人员更快地高质量地交付软件，其提供了Azure Pipelines、Azure Boards、Azure Artifacts、Azure Repos和Azure Test Plans。关于Azure DevOps我们就介绍到这里，毕竟是免费介绍。

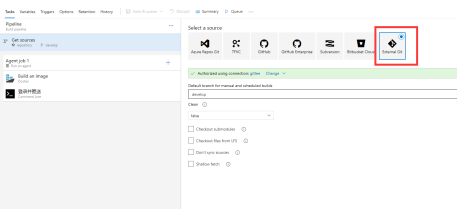
现在，我们需要侧重介绍的是Pipelines，也就是代码流水线。看，多形象，所以以前自诩为码农是错误的，我们应该是码工，广大流水线工人的一环，无产阶级之一，共产主义接班人。不好意思，又偏题了，我们继续：



首先，我们需要定义一个流水线，为了便于演示，我这里就定义一些针对Docker的简单步骤，大家可以按需添加步骤，比如单元测试步骤等等。



如图所示，步骤很简单，首先设置代码源，这里我们直接对接Magicodes.Admin框架的git库地址。Git库地址大家可以在这里找到：<https://gitee.com/xl_wenqiang/Magicodes.Admin.Core>。

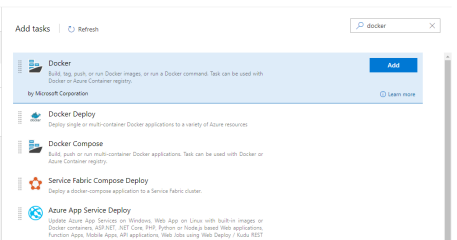


因为代码是托管再码云，所以我们选择如上图所示的最后一种方式，并且选择对应的分支。

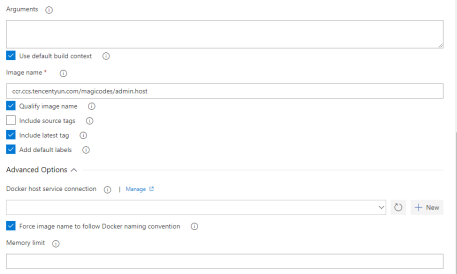
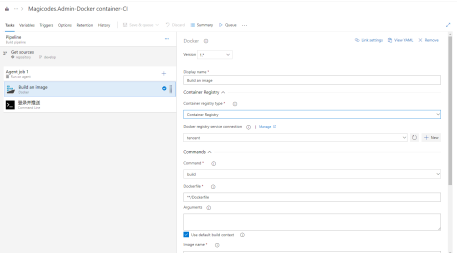
接下来，我们需要添加job和task。job添加一个默认的即可，无需设置什么条件和参数。接下来我们添加task，实际上就是步骤。

第一步，构建镜像。

我们需要添加一个docker task:



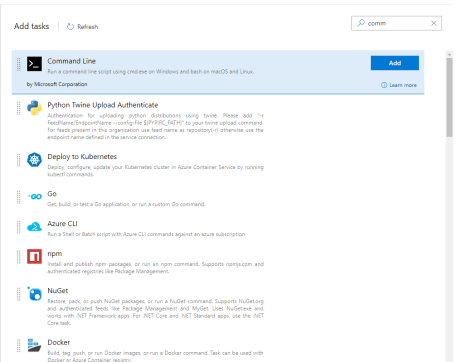
然后设置command命令为build，也就是构建：



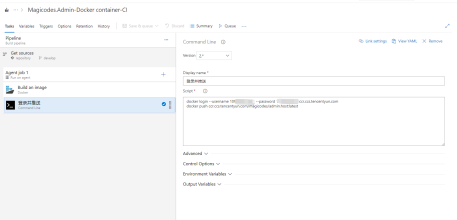
构建配置我们可以根据自己的需求来设置，比如根据分支设置镜像版本等等。

第二步，登录腾讯云镜像仓库并且推送。

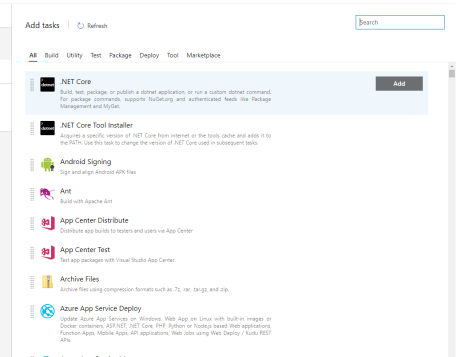
这一步，就有点门槛了，原生的docker命令并不好使，因为task之间的上下文是断开的，也就是login了你也没法push。这时候，还是命令行靠谱，简单粗暴。所以我们需要添加一个Command line task：



然后编写命令脚本：



简单粗暴的两个步骤就搞定了，大家可以根据自己的持续集成流程来定制，毕竟微软在开发者服务这块淫荡多年，还是相当给力的。我们可以初步看看支持的task：

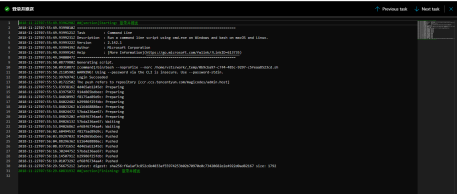
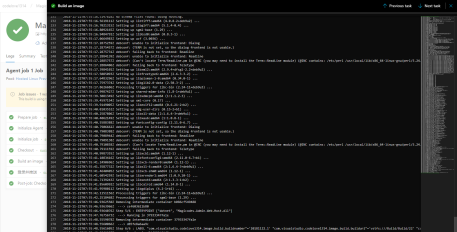


非常之多，足够我们随便玩了。而且玩坏了还不用赔钱。

接下来，跑起来：



点开还能看到详细的过程：



 激不激动，简单不简单？就这么几下就搞定了。产品很强大，就是拉取代码有点慢，看起来是托管在国外。顺手一查，额，美国：



因此，我们不是很推荐使用Azure DevOps来完成CI，网络的延迟足够拖垮我们焦虑的神经。但是如果我们的代码托管在Github，那么使用Azure DevOps是不错的选择。在接下来的教程中，我们会讲解如何打造自己的Github开源库的CI流程——不仅完全自动化，而且还支持在readme页面添加各种动态图标。