docker swarm join \ --token SWMTKN-1-1036et9x170gcou2ayq6xppu18eqoweqdhgabin406rdpvn0kf-4u18rdnvgk9bkab6nekmqskkg \192.168.132.2377

<http://gitbook.domeos.org/>

安装DOCKER最新版本

https://docs.docker.com/install/linux/docker-ce/centos/#set-up-the-repository

1：sudo apt-get update 更新文件源库

d:\Documents\Tencent Files\124276316\Image\C2C\[{I@IK6960W9GBN$YR8B3)9.png

<http://gitbook.domeos.org/>

1：Cannot connect to the Docker daemon. Is the docker daemon running on this host?

多方查找，解决如下：

sudo su -                       //切换到root

service docker start      //启动docker service

docker images              //显示所有images

docker run hello-world  //重新运行

2：centos上不了网，vi /etc/sysconfig/network-scripts/ifcfg-ens33命令进入ifcfg-ens33文档，改ONBOOT=YES

install Docker CE for centos：

https://docs.docker.com/install/linux/docker-ce/centos/#set-up-the-repository

安装GITLAB服务器：

<https://www.gitlab.com.cn/installation/#centos-7>

修改IP地址：

 /etc/gitlab/gitlab.rb:

<https://docs.gitlab.com/omnibus/settings/nginx.html#enable-https>

*# Start all GitLab components*

sudo gitlab-ctl start

*# Stop all GitLab components*

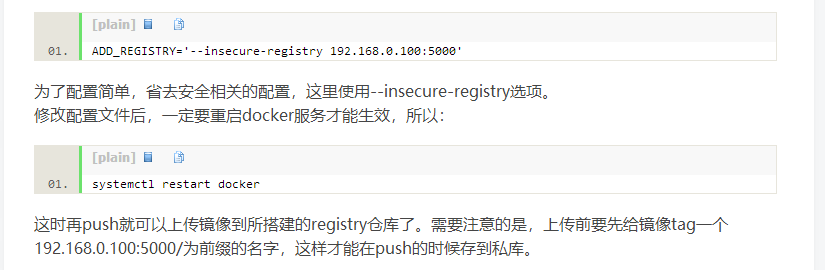
sudo gitlab-ctl stop

*# Restart all GitLab components*

sudo gitlab-ctl restart

安装私人仓库：

[http://blog.csdn.net/felix\_yujing/article/details/51564739](qq://txfile/)



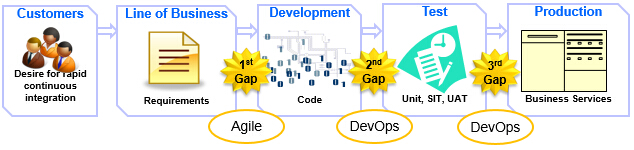
之后要如下：

到目录 /etc/docker

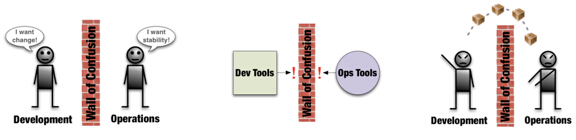
这里新建一个名为daemon.json的文件  
  
内容是{ "insecure-registries":["192.168.2.24:5000"] }  
{ "insecure-registries":["你的IP地址:5000"] }最后重启docker  service docker restart

DevOps介绍

DevOps（Deveplopment和Operations的简称），中译为开发运维一体化，可定义为是一种过程、方法、文化、运动或实践，主要是为了通过一条高度自动化的流水线来加强开发和其他IT职能部门之间的沟通和协作，加速软件和服务的交付。

[](http://dockerone.com/uploads/article/20160622/8876dba778684a1d6ac733da168f72b7.jpg)

在一个较成熟的软件和服务交付的团队里，就技术层面来说主要分为三个组成部分：开发、测试和运维。DevOps的作用就是将这三个部分紧密的连接起来，提供一条从软件开发到质量保障到技术运营的自动化流水线，加强不同角色之间的沟通和协作，基于用户需求实现软件和服务的快速交付。

[](http://dockerone.com/uploads/article/20160622/78f4671f05d233f4d42bd1eb9df9ae4c.jpg)

"这是运维的盘..."  
  
描述得略显浮夸……但这种踢皮球的事情在IT公司里面真的是随处可见。无谓对错，也无锅可背，都是由开发和运维的基因所决定，但是最终的受害者却是用户。偏偏比较有意思的就是，开发和运维人员所做的这些也都是为了用户，开发人员必须根据用户的需求对应用的功能进行不停的变更，运维人员也必须根据用户的需求提供稳定和持续的服务。各司其职的同时也在两者之间形成了一面无形的墙，阻碍了开发和运维之间的沟通和协作，而DevOps的出现就是为了击碎这堵无形之墙。

DevOps落地的思考

技术层面

DevOps不是一个工具，但它需要被工具来实现，好在现今已经有了很多商业版和开源版的软件来形成一个有效的工具链来作为DevOps技术层面的支撑。但是光有工具还不够，再好的工具没人会用也没意义，所以需要有熟悉这个工具链的IT人员来提供技术支持，利用工具实现DevOps的高度自动化。

流程层面

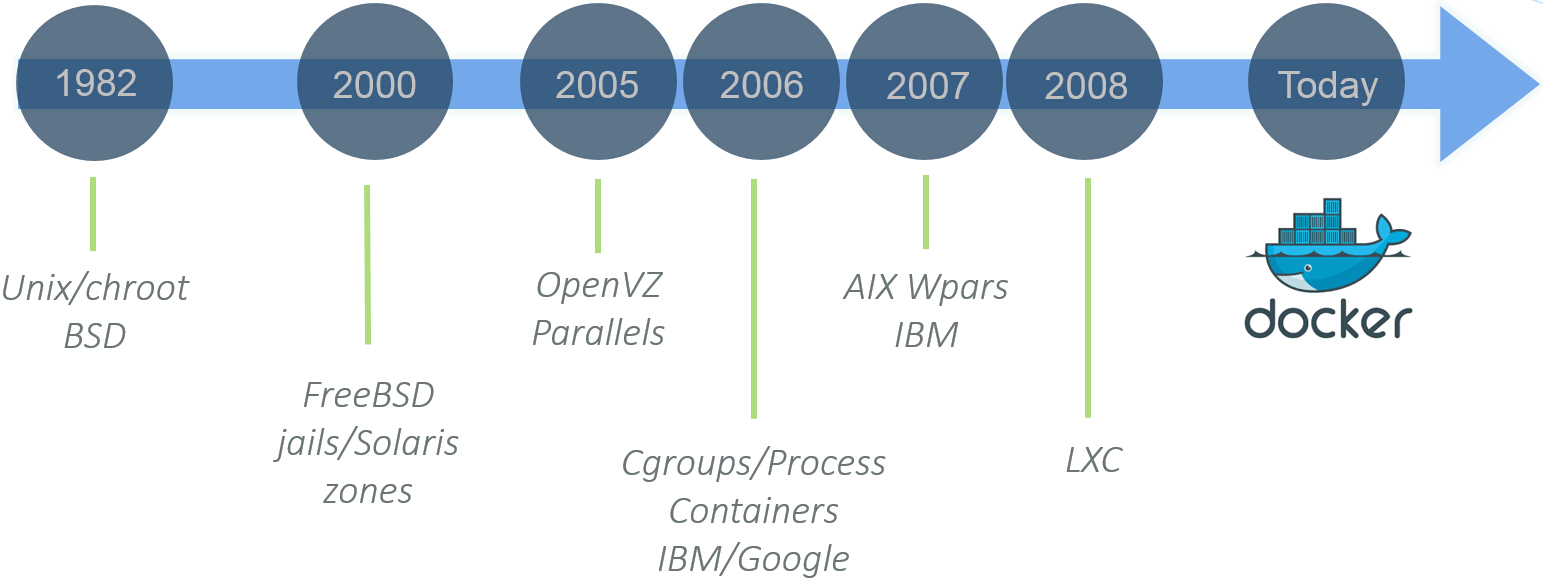
DevOps是一条从开发到运维的流水线，想要流水线能够高效的自动运行，必须要设定一系列的流程和规范来进行管控。IT的管理者需要有基于软件或服务交付的全局观，能够清晰的认识到交付周期中不同角色的痛点在哪里，进而定制出合适的协作流程。

组织层面

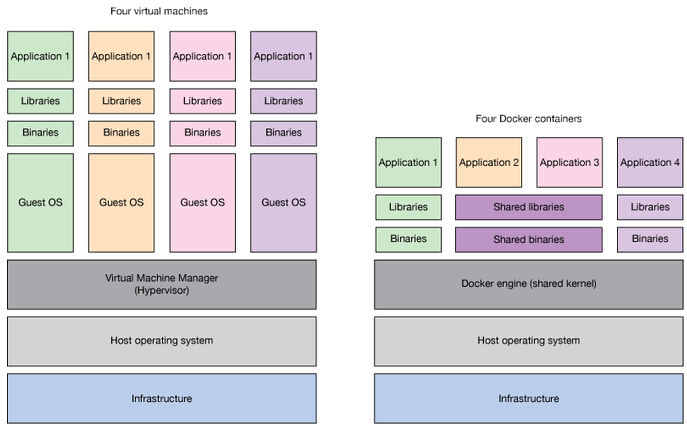
DevOps并不是简单的将开发部门和运维部门合并，而是加强开发部门和运维部门之间的协作和沟通。这需要管理者们对企业的IT部门有着足够的重视并且愿意去推动DevOps这种开发和运维间高效协作的模式，并且开发和运维的人员之间也需要有开放、接纳和协作的意识。   
  
DevOps是一个虚无缥缈的玩意儿，它并不能被工具或软件来简单的定义或量化。但工具或软件却是实现DevOps的一个重要组成部分，而Docker就是实现DevOps最合适的工具之一。

Docker介绍

Docker是一个分布式应用构建、迁移和运行的开放平台，它允许开发或运维人员将应用和运行应用所依赖的文件打包到一个标准化的单元（容器）中运行。

[](http://dockerone.com/uploads/article/20160622/b3b23ea1112d7253d5652bed397c56d7.jpg)

容器是一个非常早期的技术，Unix的Chroot功能可以说是容器的雏形，而后到大家所熟知的基于Namespace和Cgroups技术的LXC（Linux Container），最后到现在如日中天的Docker。站在前人的肩膀之上，Docker最妙的地方就是将容器的使用简单化和标准化，再配合一波开源、互联网、云计算、大数据的浪潮，可谓是时代的宠儿。

[](http://dockerone.com/uploads/article/20160622/9d5a7727de35f035ebc7d004770b828a.jpg)

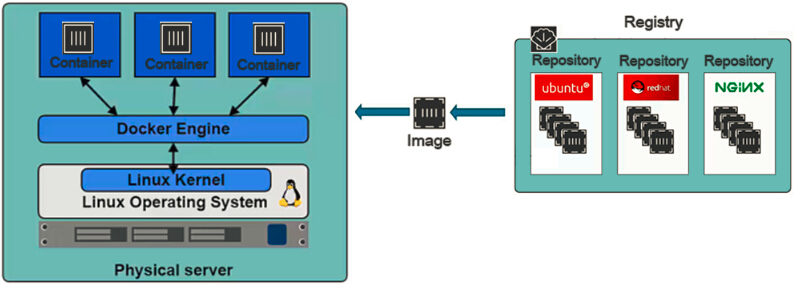
很多人都喜欢拿容器和虚拟机对比，其实容器和虚机都是属于虚拟化技术的一种实现。两种架构在底层上相同，需要物理硬件和操作系统的支持。不同的是虚拟机场景中，Hypervisor（如KVM）作为操作系统到虚拟机的中间层，而容器场景中Docker Engine（以Docker为例）作为操作系统到容器的中间层。虚机封装操作系统和应用，而容器则直接封装应用，这也是为什么容器要比虚机轻量的原因。

[](http://dockerone.com/uploads/article/20160622/07998f6516230e4cfbe033abc1a714c5.jpg)

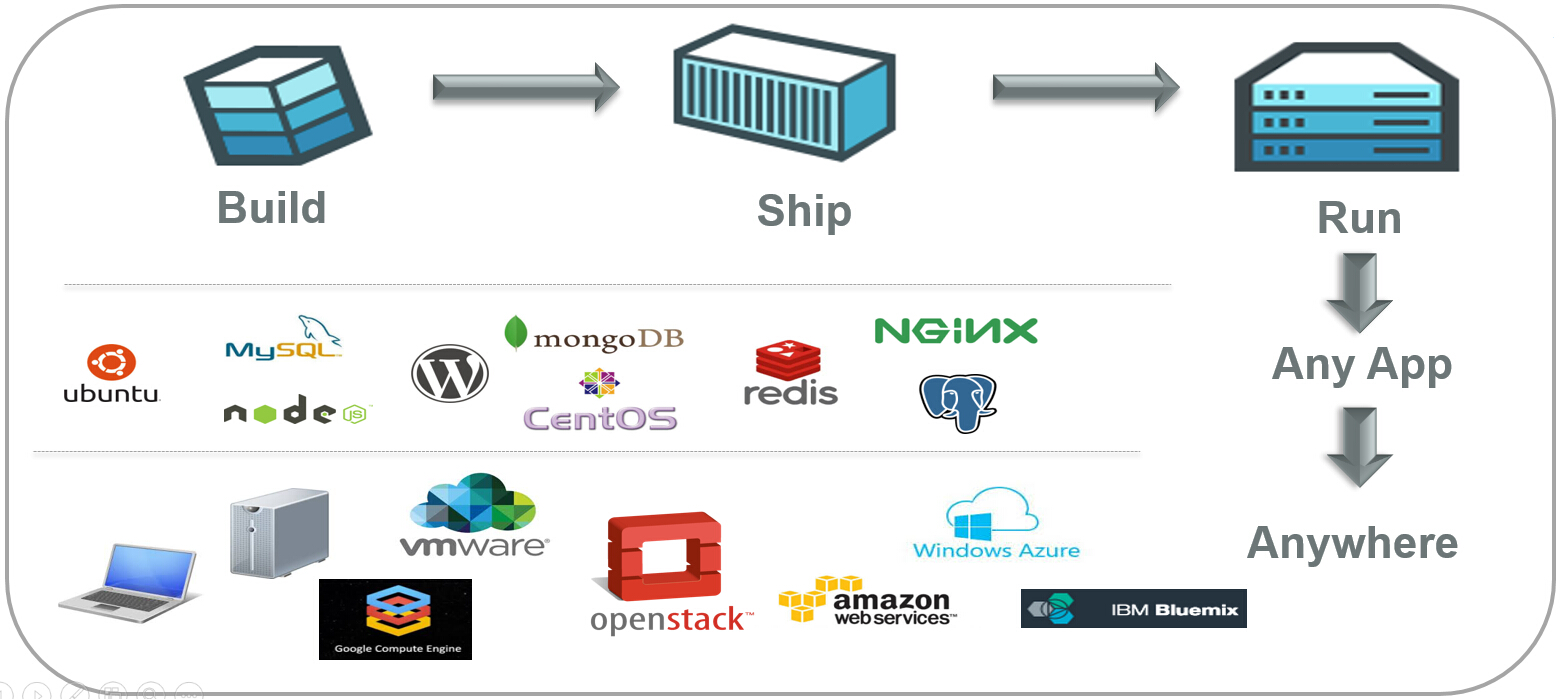
上图中将虚拟机和容器的特性进行了对比，可以看出容器相对于虚拟机比较有优势的地方就是轻量、灵活、资源利用率高。缺点主要就是隔离性不如虚拟机，也就是一直被无限放大的容器的安全性问题。但偏偏就是因为容器没有完全被隔离到一个密封的小黑屋里面，所以才能带来比虚拟机更好的资源利用率。   
  
个人认为容器在短期之内还取代不了虚机，在未来很长一段时间内会是容器和虚机并存的情况。而到最终谁替代谁，取决的不是技术本身，而是用户体验时代的需求。

PS：希望有朋友能够发现此图中的一点小漏洞。

Docker基本组件介绍

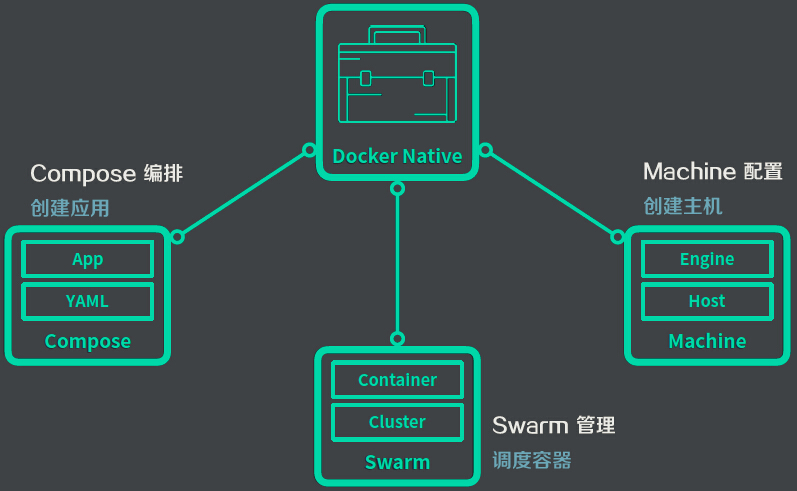
[](http://dockerone.com/uploads/article/20160622/a76141ceed65c8433e79e4117167d42d.jpg)

* Docker Image  
  Docker镜像是一个运行容器的只读模板。
* Docker Container  
  Docker容器是一个运行应用的标准化单元。
* Docker Registry  
  Docker注册服务器用来存放镜像。
* Docker Engine  
    
  Docker引擎用来在主机上创建，运行和管理容器。

[](http://dockerone.com/uploads/article/20160622/3073aae8949faf13c75dc2bc99592739.jpg)

了解Docker的朋友都知道，Docker将自身最主要的特点以下面这一句话来描述"Build,Ship and Run Any App Anywhere"。Build出Image，然后使用Registry来Ship镜像，最终使用Engine将Container和包含的App在任意平台（Anywhere）上运行起来。

Docker原生工具介绍

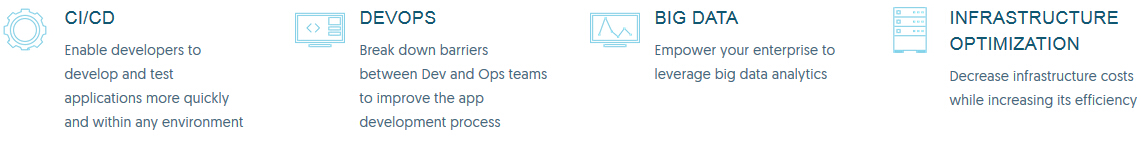
[](http://dockerone.com/uploads/article/20160622/c65506febd46873ef5650d2b615e4f79.jpg)

* Docker Machine：让用户在基础架构平台快速部署Docker宿主机；
* Docker Swarm：让用户在集群环境中调度和运行容器；
* Docker Compose：让用户在集群环境中编排和部署应用。

这三个工具构成了Docker的原生环境，加上比较火的Kubernetes、Mesos、Rancher、etcd等外部生态，构建出了一个比较完整的Docker容器生态圈。对于原生工具和外部工具，个人觉得工具或技术并没有好坏之分，主要还是看适用场景和客户需求。而正是有这些生态的合作和竞争造就的乱世，才促进了容器技术的高速发展和逐步成熟。

Docker适用的场景

* 持续集成和持续交付
* 开发运维一体化
* 容器云
* 大数据

[](http://dockerone.com/uploads/article/20160622/362af071799e0b06b9291bcab46bf701.jpg)

Docker官方给的Use Case是CI/CD、DevOps、Big Data和Infrastructure Optimization（Cloud）。   
  
这里比较有意思的就是，这几个使用场景似乎正好描绘着Docker当前的发展史。   
  
起初Docker的出现主要面向的对象是开发者，为开发者提供应用快速开发和测试的环境，这就是CI/CD所在的场景。   
  
随后的发展使得Docker不再仅仅只关注开发层面的东西，而在向运维层面迈进，就出现了DevOps的场景。  
  
既然有了运维，那肯定避免不了接触到基础架构的东西，而现今的基础架构基本都是围绕着云计算来展开，所以Docker又涉及到了基础架构优化的层面，也就是Container Cloud。   
  
基础架构的容器云有了，那么势必需要对云中的应用提供服务，加上Docker自身的许多优势，自然而然的又涉及到了Big Data的使用场景。   
  
而Docker自身的解决方案Docker Cloud和Docker Data Center的先后推出也侧面反应了从开发到运维场景的逐步支持。DDC的出现更是将目标直接瞄准了企业内部容器云。   
  
难以分清是新技术成就了Docker，还是Docker承载了新技术。至少就目前来看，Docker的发展方向是顺应这个时代的。这只是三岁多的Docker，不敢想象它在将来会有多大的能量爆发出来，将这些留给时间去述说。

Docker实现DevOps的优势

优势一

开发、测试和生产环境的统一化和标准化。镜像作为标准的交付件，可在开发、测试和生产环境上以容器来运行，最终实现三套环境上的应用以及运行所依赖内容的完全一致。

优势二

解决底层基础环境的异构问题。基础环境的多元化造成了从Dev到Ops过程中的阻力，而使用Docker Engine可无视基础环境的类型。不同的物理设备，不同的虚拟化类型，不同云计算平台，只要是运行了Docker Engine的环境，最终的应用都会以容器为基础来提供服务。

优势三

易于构建、迁移和部署。Dockerfile实现镜像构建的标准化和可复用，镜像本身的分层机制也提高了镜像构建的效率。使用Registry可以将构建好的镜像迁移到任意环境，而且环境的部署仅需要将静态只读的镜像转换为动态可运行的容器即可。

优势四

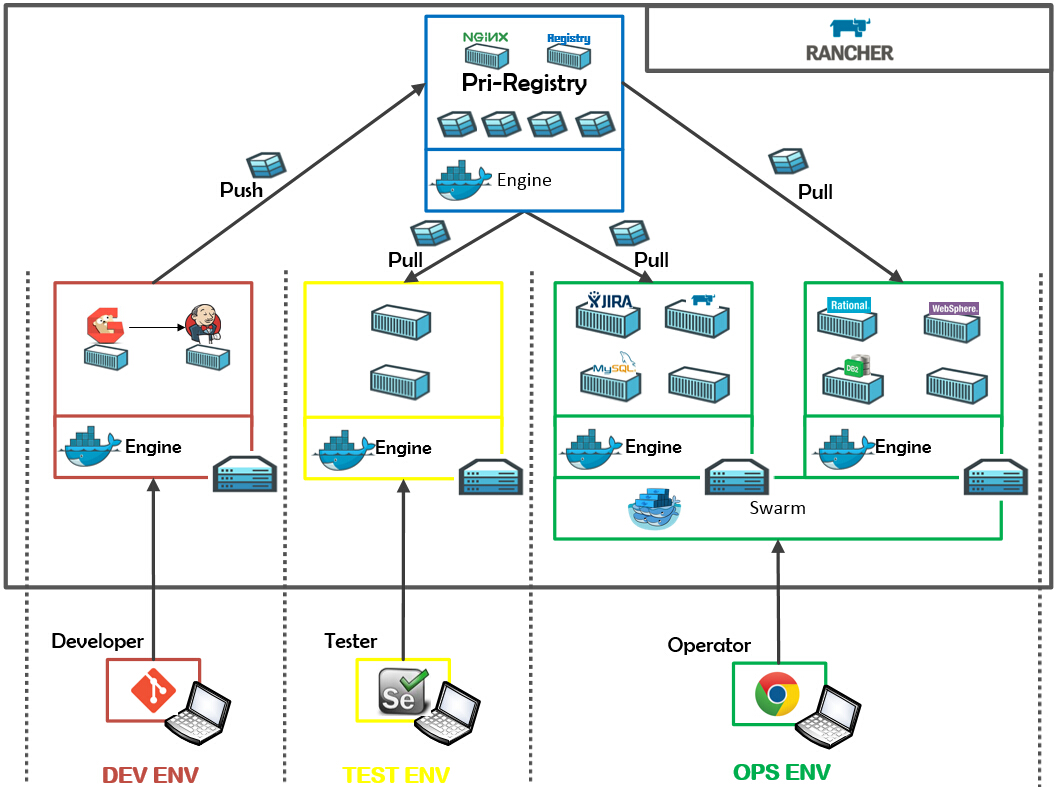
轻量和高效。和需要封装操作系统的虚拟机相比，容器仅需要封装应用和应用需要的依赖文件，实现轻量的应用运行环境，且拥有比虚拟机更高的硬件资源利用率。

优势五

工具链的标准化和快速部署。将实现DevOps所需的多种工具或软件进行Docker化后，可在任意环境实现一条或多条工具链的快速部署。

实例分享

架构介绍

[](http://dockerone.com/uploads/article/20160622/19dde8f3a78d6da4c886646f8d1861b3.jpg)

该DevOps环境基于开源产品Rancher进行管理，分为三套环境和一套横跨各环境的私有Registry。基于RancherUI和Docker宿主机，每套环境对不同的角色配置权限管理，每个角色仅能访问对应的环境。开发、测试和运维人员可自由选择Web UI或Docker CLI的方式去管理自己的环境。

DEV ENV

定义为开发环境，包含开发人员客户端的笔记本和服务端的一台Docker主机。

TEST ENV

定义为测试环境，包含测试人员客户端笔记本和服务端的一台Docker主机。

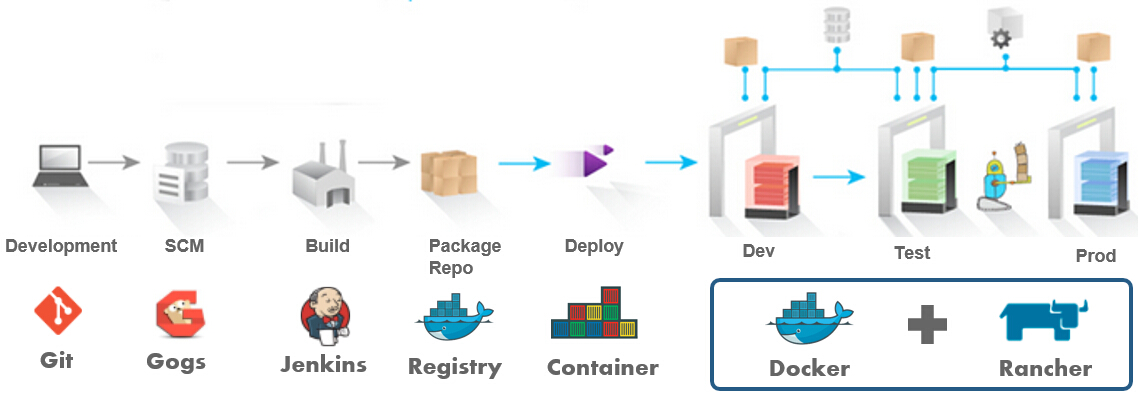
OPS ENV

定义为运维环境，包含运维人员客户端笔记本和服务端的两台Docker主机构建的Swarm集群。

Pri-Registry

私有镜像服务器，整个DevOps流水线中的核心。将镜像作为最终的交付件在不同的环境中Ship和Run，以实现从Dev到Ops应用环境的一致性部署。

运作流程

[](http://dockerone.com/uploads/article/20160622/7baf2d29052adea4fb46aa09dc4b7375.jpg)

* 开发者通过本地的Git客户端向服务端的Gogs提交代码，Jenkins将代码构建成镜像放到本地。开发人员将对应的镜像启动容器来预览的开发结果。如果确认已满足预期，则将该镜像推送到私有的Docker注册服务器中进行存储。
* 测试人员将私有镜像仓库中开发人员新提交的镜像运行成容器，并进行手动或者自动的功能性测试，测试通过后镜像会被打上一个新的Tag以被其他环境使用。
* 运维人员基于私有镜像仓库中被打过Tag的镜像启动为容器，最终交付给客户。

目前该环境主要用于Docker Image的构建和发布，Ops（Prod）环境中跑了一些项目内部使用的应用，但更多的是作为Demo环境提供给客户访问而并非真正意义上的生产环境。  
  
由于时间有限，此次交流就到这里了。内容比较粗燥，旨在抛砖引玉，谢谢大家的参与。开源的不仅仅是技术，而是思想，整套环境搭建的过程和技术细节会在本人的微信订阅号和DockOne官方网站（dockone.io）中陆续发出，有兴趣的朋友可关注“小张烤茄”订阅号，再次感谢!

Q&A

**Q：对于一些中小互联网企业尤其DevOps二次开发能力不强的，在使用Docker集群方面有什么建议？**

A：一条DevOps流水线对集群环境的需求不高，我们更应该关心的是对每一个步骤的单个工具怎么去使用和管理，集群环境适用于都会用到大规模的容器部署中。还是那句话，技术不分好坏，选择最合适的就行。如果把容器当作虚拟机来跑，可以解决你的一些问题，那就这样跑又如何呢，技术或者工具是为需求服务的。

**Q：运维投拆开发应用消耗太大，为什么用了容器就能解决？只是开发和运行用了同一套环境，性能低下一样得靠优化，客器只是让大家在同一个平台上对话了。**

A：对的，如果应用本身的代码写得不好，不论在开发环境和运维环境都会造成应用消耗大的问题。使用DevOps这样的流水线，一个方面就是解决环境异构的问题。好比我是一个开发者，我在Windows里面开发，Java用的是1.5。但是在生产环境中，APP服务器用的是Linux，Java是1.6，那这样可能会引起功能性的问题。

**Q：您好，有些传统应用比较难拆分，上云的话难免会把容器做虚拟机用，请问在这一块有好的实践么？**

A：传统应用拆分确实是个难题，从可用性和性能方面考虑把传统应用放到容器当作虚拟机跑是可以的。这种场景实现了硬件的资源的高利用率，但是由于传统应用的紧耦合，很难以利用到容器的灵活迁移和弹性部署。而且需要关心的是，传统应用放到容器里面跑，数据保护这方面到底如何去做。目前我这边没有很好的实践方案，而且这种案例确实是国内整个Docker界需要面临的一个问题。

**Q：虚机或容器，传统双机节点模式部署，出问题通过双机切换主备应用还有意义么？**

A：有，双机热备或互备的方案是在传统IT环境中经历了时间的考验的。容器的出现并不是要颠覆以前的所有，而是让客户的应用场景拥有一个多的选择。

**Q：目前容器化的都是应用APP之类的，可以使用容器化一个类Unix系统么？比如容器化一个苹果系统能实现么？**

A：容器的定位就是在于APP的封装，就算有CentOS这样的镜像也只是为了拉取运行应用需要的Bin文件和Lib文件。你这个问题有点是想把容器当作虚拟机跑了，可以去了解一下RancherVM或HyperContainer，或许能够满足你的需求。

Docker介绍：

Docker 三剑客

**三剑客简介**

**docker-machine**

docker技术是基于Linux内核的cgroup技术实现的，那么问题来了，在非Linux平台上是否就不能使用docker技术了呢？答案是可以的，不过显然需要借助虚拟机去模拟出Linux环境来。

docker-machine就是docker公司官方提出的，用于在各种平台上快速创建具有docker服务的虚拟机的技术，甚至可以通过指定driver来定制虚拟机的实现原理（一般是virtualbox）。

**docker-compose**

docker镜像在创建之后，往往需要自己手动pull来获取镜像，然后执行run命令来运行。当服务需要用到多种容器，容器之间又产生了各种依赖和连接的时候，部署一个服务的手动操作是令人感到十分厌烦的。

dcoker-compose技术，就是通过一个.yml配置文件，将所有的容器的部署方法、文件映射、容器连接等等一系列的配置写在一个配置文件里，最后只需要执行docker-compose up命令就会像执行脚本一样的去一个个安装容器并自动部署他们，极大的便利了复杂服务的部署。

**docker-swarm**

swarm是基于docker平台实现的集群技术，他可以通过几条简单的指令快速的创建一个docker集群，接着在集群的共享网络上部署应用，最终实现分布式的服务。

相比起zookeeper等集群管理框架来说，swarm显得十分轻量，作为一个工具，它把节点的加入、管理、发现等复杂的操作都浓缩为几句简单的命令，并且具有自动发现节点和调度的算法，还支持自定制。虽然swarm技术现在还不是非常成熟，但其威力已经可见一般。

**浅谈docker服务架构和远程API**

在正式使用docker技术部署集群应用时，我们应该先来了解一下docker工作的一些底层原理，和docker远程调用的API，这样才能大体了解集群究竟是如何运作的。

**daemon**

之前的docker入门文章中讲过，docker的基础服务，比如容器的创建、查看、停止、镜像的管理，其实都是由docker的守护进程(daemon)来实现的。

每次执行的docker指令其实都是通过向daemon发送请求来实现的。

daemon的运作（通信模式）主要有两种，一种是通过unix套接字（默认，但只能在本地访问到，比较安全），一种是通过监听tcp协议地址和端口来实现（这个可以实现在远程调用到docker服务）。

**远程API**

除了通过远程tcp协议访问远程主机上的docker服务外，docker还提供了一套基于HTTP的API，可以使用curl来实现操作远程主机上的docker服务，这为开发基于WEB的docker服务提供了便利。

**远程docker使用示例**

最终实现集群的时候实际是使用docker的远程调用来将不同的docker主机连接成一个整体的（通过tcp协议）。

我们不妨先来手动模拟尝试一下docker服务的远程调用吧。

首先需要在提供服务的主机上将docker的运行方式改为tcp，具体方法为修改/etc/default/docker中的DOCKER\_OPTS为如下内容

-H tcp://127.0.0.1:4243 -H unix:///var/run/docker.sock

-H 后的参数是自己定义的要绑定的tcp地址和端口，成功绑定后重启docker服务就可以在该端口访问到docker的daemon服务。

不幸的是：

* 在OSX平台上，并没有找到docker的daemon配置文件
* 在OSX平台上，使用docker -H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock -d这样的命令来尝试以tcp的方式启动docker daemon也是失败的，并没有任何作用
* 目前推测除了Linux平台上，其他平台上的配置方法都不太一样，但是在网络上暂时没有找到解决方案，所以后面的操作我只能通过在本地创建多个docker-machine的方式来模拟实现远程调用。

假设我们在192.168.1.123这台主机上开启了docker服务，监听了2375端口，那么我们就可以在同一网段的其他主机上（比如192.168.1.233）通过docker -H tcp://192.168.1.123:2345 <command>的方式调用到该主机上的docker服务。

比如

docker -H tcp://192.168.1.123:2345 ps

docker -H tcp://192.168.1.123:2345 images

docker -H tcp://192.168.1.123:2345 run ...

最终swarm构建集群的时候，就是通过这样的远程服务调用来调度各个节点的。

**集群和分布式运算**

在正式开始实践集群之前，我们有必要了解究竟什么是集群，什么是分布式计算。

首先，这两者有一个共同点，就是他们都是使用了多个服务节点的，通俗的说，就是要用到多台服务器协同工作（不一定是实体，也可能是虚拟机）。

而两者的区别在于：

* 集群是多台机器执行同一个业务，每次根据调度算法寻找最合适的节点来执行该业务
* 分布式计算是将一个业务拆分成多个独立的部分，由多台机器共同协作完成

集群的优点在于，当业务的需要的资源比较大时，可以避免由一个服务器去独自承担压力，而且即便有一个节点宕机了，业务仍然可以继续正常运行。这有点类似于负载均衡

分布式的优点则是在计算上，可以协同多台机器发挥计算的威力，进行需要超高性能的运算。

**构建集群**

说现在我们正式开始构建集群。

**使用docker-machine创建节点**

由于实体机器的缺乏以及在osx上无法正常开启tcp的docker服务，我们基于docker-machine来创建多个虚拟机，作为集群中的节点。

执行下面的命令就可以创建一个新的docker-machine虚拟机manager1

docker-machine create --driver virtualbox manager1

在创建了虚拟机后，可以使用docker-machine env manager1来查看虚拟机manager1的相关信息，包括IP地址

现在我们继续执行命令创建worker1和worker2两个节点，使用docker-machine ls命令可以查看到所有正在工作的虚拟机：

docker-machine ls

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

manager1 - virtualbox Running tcp://192.168.99.100:2376 v17.06.1-ce

worker1 - virtualbox Running tcp://192.168.99.101:2376 v17.06.1-ce

worker2 - virtualbox Running tcp://192.168.99.102:2376 v17.06.1-ce

创建docker machine后，可以通过docker-machine ssh manager1 <command>的方式来访问虚拟机，执行指令。

**创建swarm集群**

初始化一个swarm集群的命令为：

docker swarm init --listen-addr <MANAGER-IP>:<PORT> --advertise-addr <IP>

--listen-addr参数是管理者节点的docker服务所在的IP:PORT，也就是说，可以通过这个组合访问到该节点的docker服务。

--advertise-addr是广播地址，也就是其他节点加入该swarm集群时，需要访问的IP

现在我们在manager1节点里创建swarm网络，执行

docker-machine ssh manager1 docker swarm init --listen-addr 192.168.99.100:2377 --advertise-addr 192.168.99.100

返回响应：

Swarm initialized: current node (23lkbq7uovqsg550qfzup59t6) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \

--token SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-c036gwrakjejql06klrfc585r \

192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

这样便创建了一个swarm集群，并且manager1节点目前是以管理者的身份加入在节点中的。

现在我们把worker1和worker2两个节点加入到swarm集群中去，分别在两个节点的虚拟机中执行docker swarm join --token ..即可：

docker-machine ssh worker1 docker swarm join --token \

SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-c036gwrakjejql06klrfc585r \

192.168.99.100:2377

This node joined a swarm as a worker.

docker-machine ssh worker2 docker swarm join --token \

SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-c036gwrakjejql06klrfc585r \

192.168.99.100:2377

This node joined a swarm as a worker.

在任何一个节点上执行docker node ls都可以查看到当前整个集群中的所有节点：

docker-machine ssh manager1 docker node ls

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

manager1 - virtualbox Running tcp://192.168.99.100:2376 v1.12.3

worker1 - virtualbox Running tcp://192.168.99.101:2376 v1.12.3

worker2 - virtualbox Running tcp://192.168.99.102:2376 v1.12.3

**创建跨主机网络**

集群建立完毕之后我们要做的就是在集群上部署我们的服务。但是首先应该让所有的节点处在一个共享的网络中，这样当我们把服务部署在这个共享网络中，就相当于部署在整个集群中了。

使用docker network ls可以查看到当前主机所参与的所有网络：

docker-machine ssh manager1 docker network ls

NETWORK ID NAME DRIVER SCOPE

764ff31881e5 bridge bridge local

fbd9a977aa03 host host local

6p6xlousvsy2 ingress overlay swarm

e81af24d643d none null local

其中SCOPE为swarm，DRIVER为overlay的即为集群节点中的共享网络。集群建立后会有一个默认的ingress共享网络，现在我们来再创建一个：

docker-machine ssh manager1 docker network create --driver overlay swarm\_test

**在跨主机网络上部署服务**

在集群上部署应用，就是在共享网络上部署**服务(service)**。

但首先要保证每个节点上都已经有所需的镜像和环境了，这点便可以通过将同一份docker-compose配置文件共享到每个主机上，使用docker-compose在每个节点上下载镜像和搭建环境的工作。

由于judge\_server的服务架构很简单，就一个镜像，所以我在这里直接在每台主机上把它pull下来就好了：

docker-machine ssh manager1 docker pull registry.cn-qingdao.aliyuncs.com/marklux/judge\_server:1.0

docker-machine ssh worker1 docker pull registry.cn-qingdao.aliyuncs.com/marklux/judge\_server:1.0

docker-machine ssh worker2 docker pull registry.cn-qingdao.aliyuncs.com/marklux/judge\_server:1.0

接下来便是重头戏，我们使用manager1节点，在共享网络上启动我们的服务

docker service create --replicas 3 --name judge\_swarm -p 8090:8090 --network=swarm\_test registry.cn-qingdao.aliyuncs.com/marklux/judge\_server:1.0

这个命令看起来是不是很像docker run？没错，swarm最终的目的就是把操作集群变得像操作单一的docker服务端一样简单！

--replicas 用于指定服务需要的节点数量，也就是集群的规模，这个值是弹性的，你可以在后续动态的更改它。

当服务中某个节点挂掉时，swarm将会搜寻集群中剩余的可用节点，顶替上去。也就是说，swarm会动态的调度，总是保持服务是由3个节点运行着的。

-p 用于暴露端口到宿主机，这样我们就能访问到了。

--network用于指定部署service的网络是哪一个

现在在manager1节点中使用docker service ls来查看集群中的服务：

docker service ls

ID NAME MODE REPLICAS IMAGE PORTS

kofcno637cmq judge\_swarm replicated 3/3 registry.cn-qingdao.aliyuncs.com/marklux/judge\_server:1.0 \*:8090->8090/tcp

现在我们尝试在本地访问192.168.99.100:8090/ping，就可以得到响应了,事实上，现在无论将ip换为worker1或者worker2的，响应的结果都是一样，因为此时所有节点已经处在一个共同的集群网络下了

经过大量的访问测试，可以看到hostname是在变化着的，这说明每次请求，都由swarm动态的调度，选择了不同的节点来进行处理。

**遗留问题**

至此集群的部署已经完成，但是我们还遗留了几个问题没有解决：

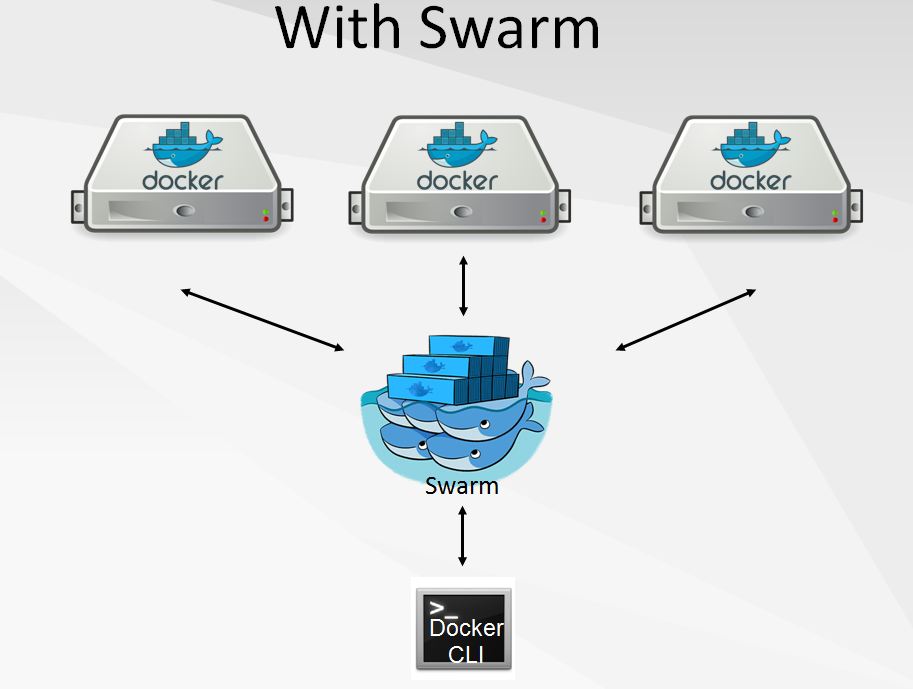
* 集群节点的动态添加删除不是很方便，这导致在web端管理判题服务机有一定的难度，当然可以通过docker的REMOTE API来实现，不过复杂度比较高
* 集群节点间的文件同步不太好实现，可能需要自己写脚本同步或是使用rsync之类的服务来实现
* swarm非常适合快速构建大量集群来实现业务的处理，不过对于只有几台机器的情况而言，有些"杀鸡用牛刀"的感觉

Swarm背景

现实中我们的应用可能会有很多，应用本身也可能很复杂，单个Docker Engine所能提供的资源未必能够满足要求。而且应用本身也会有可靠性的要求，希望避免单点故障，这样的话势必需要分布在多个Docker Engine。在这样一个大背景下，Docker社区就产生了Swarm项目。

Swarm是什么

Swarm这个项目名称特别贴切。在Wiki的解释中，Swarm behavior是指动物的群集行为。比如我们常见的蜂群，鱼群，秋天往南飞的雁群都可以称作Swarm behavior。

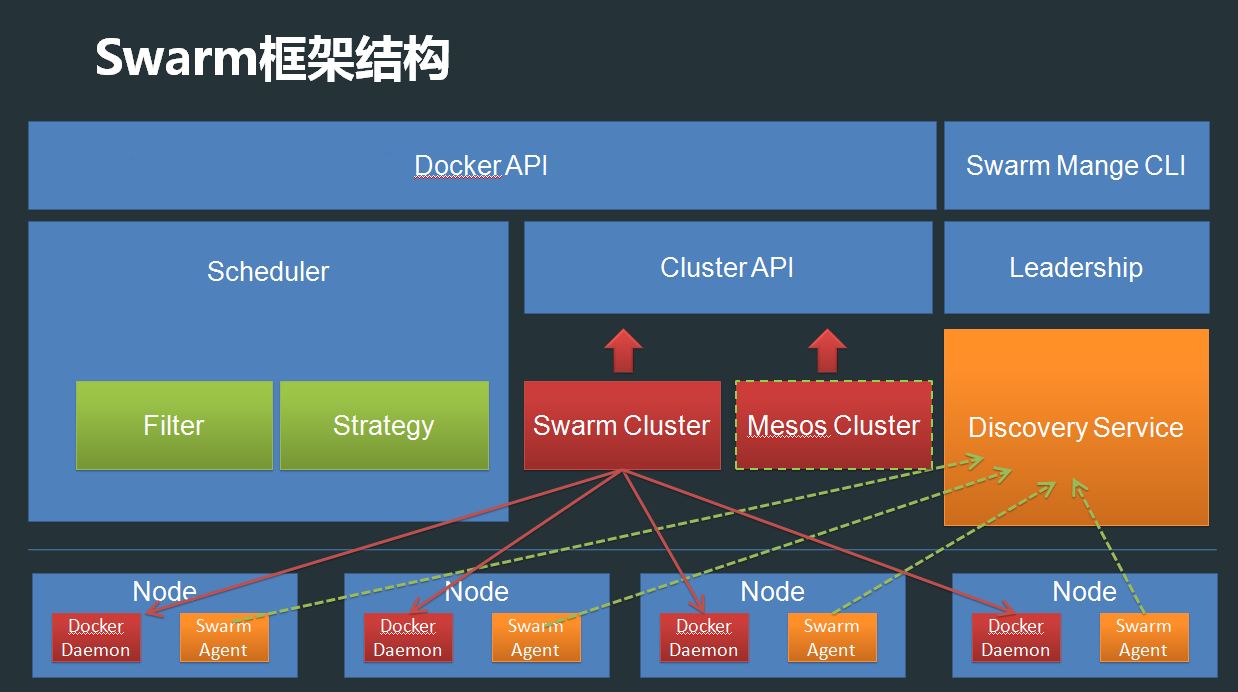
[](http://dockerone.com/uploads/article/20150908/dfc9566b84ce7ca9a2e60c3d6542178e.JPG)

Swarm项目正是这样，通过把多个Docker Engine聚集在一起，形成一个大的docker-engine，对外提供容器的集群服务。同时这个集群对外提供Swarm API，用户可以像使用Docker Engine一样使用Docker集群。

Swarm 特点

* 对外以Docker API接口呈现，这样带来的好处是，如果现有系统使用Docker Engine，则可以平滑将Docker Engine切到Swarm上，无需改动现有系统。
* Swarm对用户来说，之前使用Docker的经验可以继承过来。非常容易上手，学习成本和二次开发成本都比较低。同时Swarm本身专注于Docker集群管理，非常轻量，占用资源也非常少。 \*“Batteries included but swappable”，简单说，就是插件化机制，Swarm中的各个模块都抽象出了API，可以根据自己一些特点进行定制实现。
* Swarm自身对Docker命令参数支持的比较完善，Swarm目前与Docker是同步发布的。Docker的新功能，都会第一时间在Swarm中体现。

Swarm框架结构

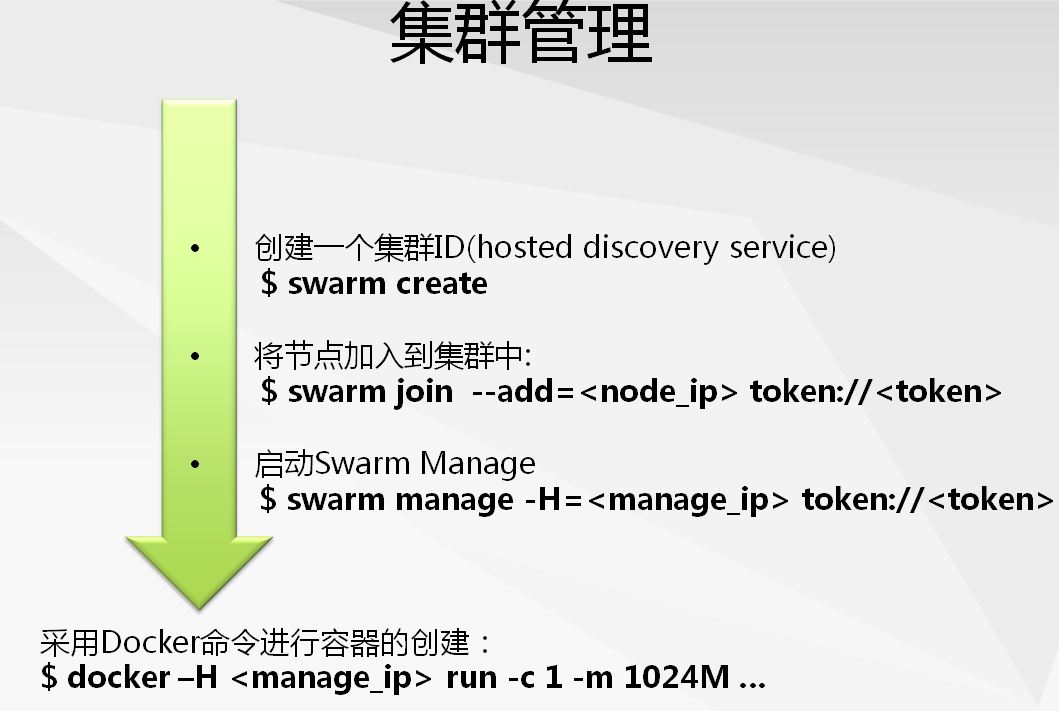
[](http://dockerone.com/uploads/article/20150908/11f6d762768d27db4553924173c83adb.JPG)

* Swarm对外提供两种API， 一种是Docker API，用于负责容器镜像的生命周期管理， 另外一种是Swarm集群管理CLI，用于集群管理。
* Scheduler模块，主要实现调度功能。在通过Swarm创建容器时，会经过Scheduler模块选择出一个最优节点，里面包含了两个子模块，分别是Filter和Strategy， Filter用来过滤节点，找出满足条件的节点（比如资源足够，节点正常等等），Strategy用来在过滤出的节点中根据策略选择一个最优的节点（比如对找出的节点进行对比，找到资源最多的节点等等）, 当然Filter/Strategy用户可以定制。
* Swarm对集群进行了抽象，抽象出了Cluster API，Swarm支持两种集群，一种是Swarm自身的集群，另外一种基于Mesos的集群。
* LeaderShip模块用于Swarm Manager自身的HA，通过主备方式实现。
* Discovery Service 服务发现模块，这个模块主要用来提供节点发现功能。
* 在每一个节点上，都会有一个Agent，用于连接Discovery Service，上报Docker Daemon的IP端口信息，Swarm Manager会直接从服务发现模块中读取节点信息。

Swarm各个模块介绍

集群管理

Swarm Manager CLI用于集群管理。大家可以看这张图，通过三步就可以将集群创建起来。

[](http://dockerone.com/uploads/article/20150908/7b9d0aa76737467b2abfa12341f2221e.JPG)

Swarm容器集群创建完成后，就可以采用Docker命令，像使用Docker Engine一样使用Swarm集群创建容器了。

服务发现

服务发现，在Swarm中主要用于节点发现，每一个节点上的Agent会将docker-egine的IP端口注册到服务发现系统中。Manager会从服务发现模块中读取节点信息。Swarm中服务发现支持已下3种类型的后端：  
第一种，是hosted discovery service，是Docker Hub提供的服务发现服务，需要连接外网访问。   
第二种，是KV分布式存储系统，现在已支持etcd、ZooKeeper、Consul三种。   
第三种，是静态IP。可以使用本地文件或者直接指定节点IP，这种方式不需要启动额外使用其他组件,一般在调试中会使用到。   
Scheduler

调度模块主要用户容器创建时，选择一个最优节点。在选择最优节点过程中，分为了两个阶段：   
第一个阶段，是过滤。根据条件过滤出符合要求的节点，过滤器有以下5种：

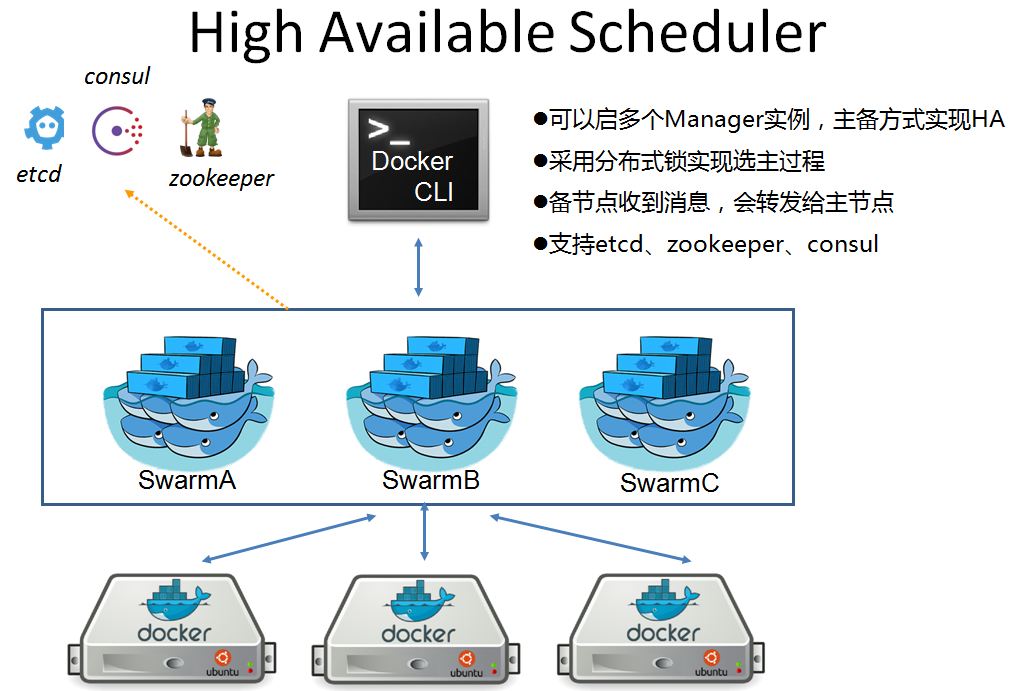
1. Constraints，约束过滤器，可以根据当前操作系统类型、内核版本、存储类型等条件进行过滤，当然也可以自定义约束，在启动Daemon的时候，通过Label来指定当前主机所具有的特点。
2. Affnity，亲和性过滤器，支持容器亲和性和镜像亲和性，比如一个web应用，我想将DB容器和Web容器放在一起，就可以通过这个过滤器来实现。
3. Dependency，依赖过滤器。如果在创建容器的时候使用了**--volume-from/--link/--net**某个容器，则创建的容器会和依赖的容器在同一个节点上。
4. Health filter，会根据节点状态进行过滤，会去除故障节点。
5. Ports filter，会根据端口的使用情况过滤。

调度的第二个阶段是根据策略选择一个最优节点。有以下三种策略：

1. Binpack，在同等条件下，选择资源使用最多的节点，通过这一个策略，可以将容器聚集起来。
2. Spread，在同等条件下，选择资源使用最少的节点，通过这一个策略，可以将容器均匀分布在每一个节点上。
3. Random，随机选择一个节点。

Leadership

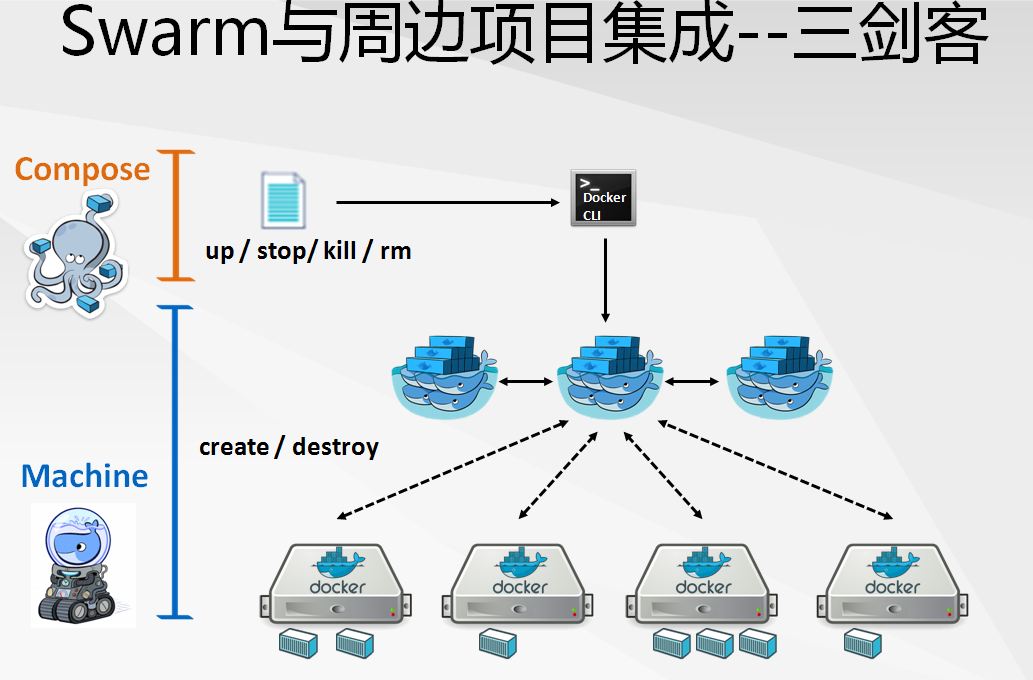
Leadership模块，这个模块主要用来提供Swarm Manager自身的HA。

[](http://dockerone.com/uploads/article/20150908/6290a7079c6505b6cb4978802846ab5a.JPG)

为了防止Swarm Manager单点故障，引入了HA机制，Swarm Manager自身是无状态的，所以还是很容易实现HA的。 实现过程中采用主备方式，当主节点故障以后，会从新选主提供服务，选主过程中采用分布式锁实现，现在支持etcd、ZooKeeper、Consul三种类型的分布式存储，用来提供分布式锁。 当备节点收到消息后，会将消息转发给主节点。   
  
以上就是框架中各个模块的相关介绍，下来和大家一起再看一下，Swarm与周边项目的集成。  
  
首先看一下，与三剑客之间的集成。

Swarm与周边项目集成

三剑客是Docker公司去年底发布的三个项目，这三者是可以紧密协作的。可以看一下这张图：

[](http://dockerone.com/uploads/article/20150908/58074f17fc5aeaf39cc985d38b807c11.JPG)

最下面是Machine，通过Machine可以在不同云平台上创建出包含docker-engine的主机。Machine通过driver机制，目前支持多个平台的docker-egine环境的部署，比如亚马逊、OpenStack等。 Docker Engine创建完以后，就该Swarm上场了，Swarm将每一个主机上的docker-egnine管理起来，对外提供容器集群服务。最上面是Compose项目，Compose项目主要用来提供基于容器的应用的编排。用户通过yml文件描述由多个容器组成的应用，然后由Compose解析yml，调用Docker API，在Swarm集群上创建出对应的容器。   
  
我们知道现在围绕Docker已经产生了很大的一个生态圈。 因此Swarm不仅在和自家兄弟集成，也能积极和周边的一些项目集成。比如，Swarm现在已经可以和Mesos进行集成。Swarm与Mesos集成时，也是以Framework方式集成，实现了Framework所需的接口。这个大特性处在experiment阶段。

Swarm社区的现状

Swarm项目去年底发布，发展了短短半年时间，已经到了0.4版本，目前还处于正在快速演进阶段。 Swarm发布版本周期目前是跟着Docker一起发布，基本上两个月一个版本，在开发过程中，采用迭代方式开发，基本上每两个星期完成一轮迭代。参与社区的方法基本上和其他社区一致。当遇到问题时，可以在社区创建issue，然后描述问题，最好能上环境信息以及问题重现的步骤，这样有利于问题的定位。当然也可也直接通过IRC或者邮件直接交流。 Swarm社区很欢迎大家的参与，不论是使用中遇到的问题以及Bug，还是Swarm功能上目前无法满足大家的地方。都欢迎大家提出来，一起讨论。  
  
如果对代码感兴趣的话，可以参考Docker社区的提交代码流程来提交代码，也非常欢迎大家能够参与到Swarm社区中提交代码。

Swarm未来规划

1. 首先是支持所有的Docker API，现在支持率大概在95%，其中一些实现还存在问题，需要改进。
2. 第二块是网络部分，通过Libnetwork项目，实现overlay network。
3. 第三块是Self healing，通过这一个功能可以实现，当一个节点故障以后，会将故障节点上的容器在另外一些节点上创建。
4. 第四块是Global Scheduler。这个特性主要用来将一个容器在每一个节点上进行创建。比如，想将一个log容器在每一个节点创建，用来记录日志，则可以通过这一特性实现。
5. 最后是volume，这一块社区最近在讨论。

Q&A

**Q：Kubernetes和Swarm 相比较来看如何选择呢？**  
A：一个很Open的话题，根据特点，选择适合自己的就OK。Swarm对外提供Docker API，自身轻量、学习成本、二次开发成本都比较低，自身是插件式框架。从功能上讲，Swarm是Q：Kubernetes的一个子集，个人感觉，Compose+Swarm =Kubernetes。

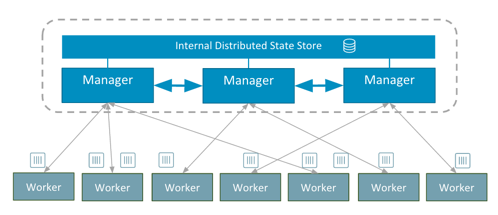
 docker 1.12 版本 的新特性

（1）docker swarm：集群管理，子命令有init, join, leave, update

（2）docker service：服务创建，子命令有create, inspect, update, remove, tasks

（3）docker node：节点管理，子命令有accept, promote, demote, inspect, update, tasks, ls, rm

（4）docker stack/deploy：试验特性，用于多应用部署， 类似与 docker-compose 中的特性。



我在使用的时候还是 测试版本，所以使用

wget -qO- https://test.docker.com/ | sh

进行安装配置

安装完毕以后:

[复制代码](javascript:void(0);)

[root@swarm-manager ~]# docker version

Client:

Version: 1.12.0-rc2

API version: 1.24

Go version: go1.6.2

Git commit: 906eacd

Built:

OS/Arch: linux/amd64

Server:

Version: 1.12.0-rc2

API version: 1.24

Go version: go1.6.2

Git commit: 906eacd

Built:

OS/Arch: linux/amd64

[复制代码](javascript:void(0);)

一、 我们首先来看看 1.12 中 新特性里面的 内置 swarm 命令 （swarmkit采用raft协议构建集群）

[复制代码](javascript:void(0);)

[root@swarm-manager ~]# docker swarm --help

Usage: docker swarm COMMAND

Manage Docker Swarm

Options:

--help Print usage

Commands:

init Initialize a Swarm

join Join a Swarm as a node and/or manager

update Update the Swarm

leave Leave a Swarm

inspect Inspect the Swarm

Run 'docker swarm COMMAND --help' for more information on a command.

[复制代码](javascript:void(0);)

这里 命令有 5个

[复制代码](javascript:void(0);)

Commands:

init Initialize a Swarm

join Join a Swarm as a node and/or manager

update Update the Swarm

leave Leave a Swarm

inspect Inspect the Swarm

[复制代码](javascript:void(0);)

1. init 初始化，一个 Swarm 集群，执行效果：

[root@swarm-manager ~]# docker swarm init

Swarm initialized: current node (0vwpni05mew2j84i6gjet44iu) is now a manager.

这里显示 初始化了一个 Swarm 集群，并把这机器做为 管理节点。

[root@swarm-manager ~]#docker node ls

ID NAME MEMBERSHIP STATUS AVAILABILITY MANAGER STATUS

0vwpni05mew2j84i6gjet44iu \* swarm-manager Accepted Ready Active Leader

执行 docker node ls 可以查看 Swarm 的集群情况 （只能在 manager 中执行）

[root@swarm-manager ~]# netstat -lan|grep 2377

可以看到 群集开放了一这个 2377 的端口。

默认绑定 0.0.0.0:2377 ，当然我们也可以使用 docker swarm init --listen-addr <MANAGER-IP>:<PORT> 进行绑定ip

2377 这个端口是用于 Swarm 中 node 节点加入使使用的。

2. join 加入 Swarm 集群, 可加入做为 node 节点，也可加入 作为 管理节点。

[root@swarm-node-1 ~]#docker swarm join 10.6.0.140:2377

This node joined a Swarm as a worker.

这里在 node-1 里面执行了 join 命令，加入了 10.6.0.140 这个 manager 这个 Swarm 集群里

[root@swarm-manager ~]#docker node ls

ID NAME MEMBERSHIP STATUS AVAILABILITY MANAGER STATUS

0vwpni05mew2j84i6gjet44iu \* swarm-manager Accepted Ready Active Leader

4mqsmp0gzlqeicit98ce8wh2q swarm-node-1 Accepted Ready Active

这里可以看到 node-1 已经加入到 swarm 的集群里面来了。

3. update 命令, 只提示 updated ，具体作用尚不明白，应该是立刻刷新 swarm 的群集信息

[root@swarm-manager ~]#docker swarm update

Swarm updated.

4. leave 命令， 离开 Swarm 集群， 一个docker 只能加入一个node.

[root@swarm-node-1 ~]#docker swarm leave 10.6.0.140:2377

5. inspect 命令， 查询 Swarm 集群 的整体信息。 （只能在 manager 中执行）

[复制代码](javascript:void(0);)

[root@swarm-manager ~]#docker swarm inspect

[

{

"ID": "c052zw5ll0ugw08shg2xf7ajp",

"Version": {

"Index": 11

},

"CreatedAt": "2016-06-23T02:09:18.935434519Z",

"UpdatedAt": "2016-06-23T02:09:19.155114277Z",

"Spec": {

"Name": "default",

"AcceptancePolicy": {

"Policies": [

{

"Role": "worker",

"Autoaccept": true

},

{

"Role": "manager",

"Autoaccept": false

}

]

},

"Orchestration": {

"TaskHistoryRetentionLimit": 10

},

"Raft": {

"SnapshotInterval": 10000,

"LogEntriesForSlowFollowers": 500,

"HeartbeatTick": 1,

"ElectionTick": 3

},

"Dispatcher": {

"HeartbeatPeriod": 5000000000

},

"CAConfig": {

"NodeCertExpiry": 7776000000000000

}

}

}

]

[复制代码](javascript:void(0);)

二、 接下来我们来看看 service 命令。

[复制代码](javascript:void(0);)

[root@swarm-manager ~]#docker service --help

Usage: docker service COMMAND

Manage Docker services

Options:

--help Print usage

Commands:

create Create a new service

inspect Inspect a service

tasks List the tasks of a service

ls List services

rm Remove a service

scale Scale one or multiple services

update Update a service

Run 'docker service COMMAND --help' for more information on a command.

[复制代码](javascript:void(0);)

1. docker create 命令， 既 创建 一个 服务。

[复制代码](javascript:void(0);)

[root@swarm-manager ~]#docker service create --help

Usage: docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]

Create a new service

Options:

--constraint value Placement constraints (default [])

--endpoint-mode string Endpoint mode(Valid values: VIP, DNSRR)

-e, --env value Set environment variables (default [])

--help Print usage

-l, --label value Service labels (default [])

--limit-cpu value Limit CPUs (default 0.000)

--limit-memory value Limit Memory (default 0 B)

--mode string Service mode (replicated or global) (default "replicated")

-m, --mount value Attach a mount to the service

--name string Service name

--network value Network attachments (default [])

-p, --publish value Publish a port as a node port (default [])

--replicas value Number of tasks (default none)

--reserve-cpu value Reserve CPUs (default 0.000)

--reserve-memory value Reserve Memory (default 0 B)

--restart-condition string Restart when condition is met (none, on\_failure, or any)

--restart-delay value Delay between restart attempts (default none)

--restart-max-attempts value Maximum number of restarts before giving up (default none)

--restart-window value Window used to evalulate the restart policy (default none)

--stop-grace-period value Time to wait before force killing a container (default none)

--update-delay duration Delay between updates

--update-parallelism uint Maximum number of tasks updated simultaneously

-u, --user string Username or UID

-w, --workdir string Working directory inside the container

[复制代码](javascript:void(0);)

docker service create 里面有非常多的 参数。 这里有很详细的使用说明。

下面我们来 创建一个 service 试试看

首先pull 一个 nginx 镜像 下来 用于 测试

[root@swarm-manager ~]#docker pull nginx

创建 2 个 nginx :

[root@swarm-manager ~]#docker service create --name nginx --replicas 2 -p 80:80/tcp nginx

使用 docker service ls 可查看 服务 启动情况。

[root@swarm-manager ~]#docker service ls

ID NAME REPLICAS IMAGE COMMAND

1b9a58mlz330 nginx 1/2 nginx

使用 tasks 命令 可查看 nginx 的情况。

[root@swarm-manager ~]#docker service tasks nginx

ID NAME SERVICE IMAGE LAST STATE DESIRED STATE NODE

56er48j3hin9ysdi3sb1chbn1 nginx.1 nginx nginx Preparing 2 minutes Running swarm-node-1

e7vtvpkbstznoi8ogihaao1f5 nginx.2 nginx nginx Running 2 minutes Running swarm-manager

这里 显示 swarm-node-1 节点中 last state 的状态为 preparing .

原因是 swarm-node-1 节点并没有 nginx 的镜像

在 node-1 节点pull nginx 镜像。

[root@swarm-node-1 ~]#docker pull nginx

node-1 节点pull 完毕以后，在使用 service ls 查看， 服务已经自动 启动了。

Swarm模式下的引擎拥有自组织与自修复特性，意味着它们能够识别我们定义的应用，并在出现差错时持续检查并修复环境。

举例来说，如果大家关闭某台运行有Nginx实例的设备，则另一节点上会自动启动一套新的容器。

如果关闭Swarm内半数设备所使用的网络交换机，则另外一半设备会顶替而上，接管对应工作负载。

[root@swarm-manager ~]#docker service ls

ID NAME REPLICAS IMAGE COMMAND

1b9a58mlz330 nginx 2/2 nginx

[root@swarm-manager ~]#docker service tasks nginx

ID NAME SERVICE IMAGE LAST STATE DESIRED STATE NODE

56er48j3hin9ysdi3sb1chbn1 nginx.1 nginx nginx Running 32 minutes Running swarm-node-1

e7vtvpkbstznoi8ogihaao1f5 nginx.2 nginx nginx Running 32 minutes Running swarm-manager

2. docker service scale 命令， 这个命令是批量生成已有容器。

直接对 nginx=10 既可让 nginx 的容器生成10个。

[root@swarm-manager ~]#docker service scale nginx=10

nginx scaled to 10

使用 tasks 可以看到，已经在 2个 节点中生成了10个 nginx 容器

[复制代码](javascript:void(0);)

[root@swarm-manager ~]#docker service tasks nginx

ID NAME SERVICE IMAGE LAST STATE DESIRED STATE NODE

56er48j3hin9ysdi3sb1chbn1 nginx.1 nginx nginx Running 43 minutes Running swarm-node-1

e7vtvpkbstznoi8ogihaao1f5 nginx.2 nginx nginx Running 43 minutes Running swarm-manager

9vqxcmskj1nawo8wl0fqr32j2 nginx.3 nginx nginx Preparing 20 seconds Running swarm-manager

0vbqoyestm7ob6r1zq9jwj6il nginx.4 nginx nginx Running 20 seconds Running swarm-node-1

13jf9mkl4k5e57pq4hoeb68ru nginx.5 nginx nginx Running 20 seconds Running swarm-node-1

a0tk6ni6a02diuo5u3t870qk7 nginx.6 nginx nginx Running 20 seconds Running swarm-manager

cwplvo5wfqp3rn5ynvxv9wv90 nginx.7 nginx nginx Running 20 seconds Running swarm-manager

7feil5xqc5hdkseasthkq2nyx nginx.8 nginx nginx Running 20 seconds Running swarm-node-1

8jt5yovxoz7t89edinb9ydao1 nginx.9 nginx nginx Starting 20 seconds Running swarm-node-1

dst4ydun1upham0o7e8a9hj3w nginx.10 nginx nginx Running 20 seconds Running swarm-manager

[复制代码](javascript:void(0);)

当我们想 缩容 时间， 也可以使用 scale nginx=2 让容器变成2个。

[root@swarm-manager ~]#docker service scale nginx=2

nginx scaled to 2

在运行 nginx=2 时可以看到 容器已经缩小为 2个 。

当我们使用 docker ps 查看，会发现容器被 stop 而非 rm 。

当我们使用 docker service rm nginx 的时候，所有的容器都会被 删除，请注意。

[root@swarm-manager ~]#docker service tasks nginx

ID NAME SERVICE IMAGE LAST STATE DESIRED STATE NODE

0vbqoyestm7ob6r1zq9jwj6il nginx.4 nginx nginx Running 12 minutes Running swarm-node-1

13jf9mkl4k5e57pq4hoeb68ru nginx.5 nginx nginx Running 12 minutes Running swarm-node-1

3. docker service update 命令。 可对 服务的启动 参数 进行 更新/修改。

上面我们新建了一个服务，命令为：

[root@swarm-manager ~]#docker service create --name nginx --replicas 2 -p 80:80/tcp nginx

如果我们先新加入了一个 node 想让 nginx 分布在 3个 node 上面， 我们可以使用 update 命令。

[root@swarm-manager ~]#docker service update --replicas 3 nginx

nginx

更新完毕以后 我们可以查看到 REPLICAS 已经变成 3/3

[root@swarm-manager ~]#docker service ls

ID NAME REPLICAS IMAGE COMMAND

1b9a58mlz330 nginx 3/3 nginx

docker service update 命令，也可用于直接 升级 镜像等。

[root@swarm-manager ~]#docker service tasks nginx

ID NAME SERVICE IMAGE LAST STATE DESIRED STATE NODE

0vbqoyestm7ob6r1zq9jwj6il nginx.4 nginx nginx Running 41 minutes Running swarm-node-1

340e1u31vadq3jtebzeddmatt nginx.5 nginx nginx Preparing 5 seconds Running swarm-manager

上面我们使用了 nginx 镜像启动了 任务。 使用 update --image 可直接对 image 进行更新。

[root@swarm-manager ~]#docker service update --image nginx:new nginx

nginx

可以看到 IMAGE 已经变成 nginx:new

[root@swarm-manager ~]#docker service tasks nginx

ID NAME SERVICE IMAGE LAST STATE DESIRED STATE NODE

2ba3utpk6icf0w449kcwgxmnm nginx.4 nginx nginx:new Running 49 seconds Running swarm-manager

5wmmneiueeool09fs8d2g1ncq nginx.5 nginx nginx:new Running 49 seconds Running swarm-node-1

挂载目录, mount

[复制代码](javascript:void(0);)

docker service create --mount type=bind,target=/container\_data/,source=/host\_data/

例 - 本地目录： target = 容器里面的路径， source = 本地硬盘路径

docker service create --name nginx --mount type=bind,target=/usr/share/nginx/html/,source=/opt/web/ --replicas 2 --publish 80:80/tcp nginx

docker service create --mount type=volume,source=<VOLUME-NAME>,target=<CONTAINER-PATH>,volume-driver=<DRIVER>,

例 - 挂载volume卷： source = volume 名称 , traget = 容器里面的路径

docker service create --name nginx --mount type=volume,source=myvolume,target=/usr/share/nginx/html,volume-driver=local --replicas 2 --publish 80:80/tcp nginx

[复制代码](javascript:void(0);)

三、 下面是 node , node命令用于节点管理:

[复制代码](javascript:void(0);)

[root@swarm-manager ~]#docker node --help

Usage: docker node COMMAND

Manage Docker Swarm nodes

Options:

--help Print usage

Commands:

accept Accept a node in the swarm

demote Demote a node from manager in the swarm

inspect Inspect a node in the swarm

ls List nodes in the swarm

promote Promote a node to a manager in the swarm

rm Remove a node from the swarm

tasks List tasks running on a node

update Update a node

Run 'docker node COMMAND --help' for more information on a command.

[复制代码](javascript:void(0);)

1. 首先是 docker node accept 命令， 这个命令用于 同意 申请加入 swarm 集群。

在使用 docker swarm init 的时候，如果使用了 --auto-accept none 的话，需要使用 docker node accept 来通过申请。

在没有通过申请之前，节点 MEMBERSHIP 状态为 Pending 状态。

--auto-accept 可以设置三种角色 分别为 (worker, manager, or none) 。

使用 docker node accept + 节点 ID 既可通过申请。

2. docker node promote 与 docker node demote 命令。

docker node promote 是 将 worker 普通节点，提升为 manager 节点。

docker node demote 是 将 manager 管理节点，降级为 worker 节点。

3. docker node inspect 可查看节点的具体信息  
docker node rm 可删除一个节点   
docker node tasks 可查看节点中运行的 service 任务。

四、 docker stack/deploy 目前 stack 还处于 测试阶段。

https://github.com/docker/docker/blob/master/experimental/docker-stacks-and-bundles.md

目前 docker 1.12.0-rc2 版本还没有这个命令， 等待 正式版 的到来。

K8s介绍·：

主要功能体现在如下：

–使用Docker对应用程序包装、实例化

–以集群的方式运行、管理跨机器的容器

–解决Docker跨机器容器之间的通讯问题

–Kubernetes的自我修复机制使得容器集群总是运行在用户期望的状态

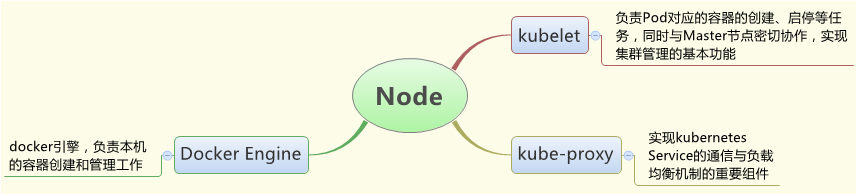
基本概念：

        Kubernetes中的大部分概念Node、Pod、Replication Controller、Service等都可以看作一种“资源对象”，几乎所有的资源对象都可以通过kubectl工具（API调用）执行增、删、改、查等操作并将其保存在etcd中持久化存储。从这个角度来看，kubernetes其实是一个高度自动化的资源控制系统，通过跟踪对比etcd库里保存的“资源期望状态”与当前环境中的“实际资源状态”的差异来实现自动控制和自动纠错的高级功能。

Master：集群控制管理节点，所有的命令都经由master处理。

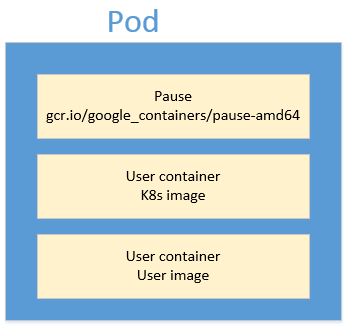


Node：是kubernetes集群的工作负载节点。Master为其分配工作，当某个Node宕机时，Master会将其工作负载自动转移到其他节点。



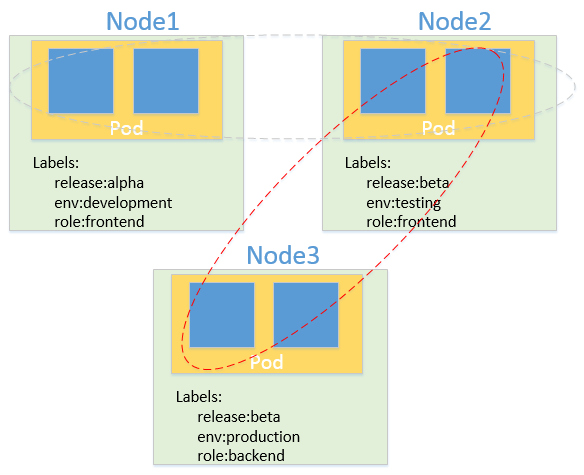
        Node节点可动态增加到kubernetes集群中，前提是这个节点已经正确安装、配置和启动了上述的关键进程，默认情况下，kubelet会向Master注册自己，这也kubernetes推荐的Node管理方式。一旦Node被纳入集群管理范围，kubelet会定时向Master汇报自身的情况，以及之前有哪些Pod在运行等，这样Master可以获知每个Node的资源使用情况，并实现高效均衡的资源调度策略。如果Node没有按时上报信息，则会被Master判断为失联，Node状态会被标记为Not Ready，随后Master会触发工作负载转移流程。

Pod：是kubernetes最重要也是最基本的概念。每个Pod都会包含一个 “根容器”，还会包含一个或者多个紧密相连的业务容器。



        Kubernetes为每个Pod都分配了唯一的IP地址，称之为PodIP，一个Pod里的多个容器共享PodIP地址。要求底层网络支持集群内任意两个Pod之间的直接通信，通常采用虚拟二层网络技术来实现（Flannel）。

Label：是一个key=value的键值对，其中key与value由用户自己指定。可以附加到各种资源对象上，一个资源对象可以定义任意数量的Label。可以通过LabelSelector（标签选择器）查询和筛选资源对象。



RC：Replication Controller声明某个Pod的副本数在任意时刻都符合某个预期值。定义包含如下：

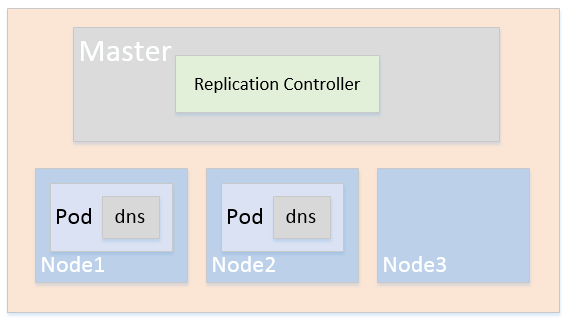
   （1）Pod期待的副本数（replicas）

   （2）用于筛选目标Pod的Label Selector

   （3）当Pod副本数小于期望时，用于新的创建Pod的模板template

  （4）通过改变RC里的Pod副本数量，可以实现Pod的扩容或缩容功能

  （5）通过改变RC里Pod模板中的镜像版本，可以实现Pod的滚动升级功能



思考：如果Node2上的pod死掉怎么办？

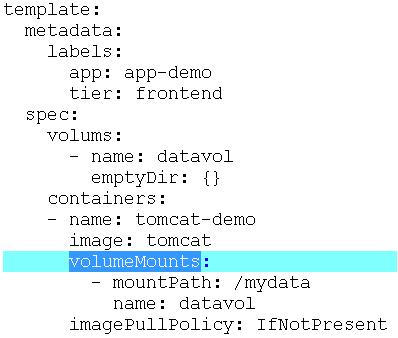
Service：“微服务”，kubernetes中的核心。通过分析、识别并建模系统中的所有服务为微服务，最终系统有多个提供不同业务能力而又彼此独立的微服务单元所组成，服务之间通过TCP/IP进行通信。每个Pod都会被分配一个单独的IP地址，而且每个Pod都提供了一个独立的Endpoint以被客户端访问。

思考：客户端如何访问？

        部署负载均衡器，为Pod开启对外服务端口，将Pod的Endpoint列表加入转发列表中，客户端通过负载均衡器的对外IP+Port来访问此服务。每个Service都有一个全局唯一的虚拟ClusterIP，这样每个服务就变成了具备唯一IP地址的“通信节点”，服务调用就变成了最基础的TCP网络通信问题。

Volume：是Pod中能够被多个容器访问的共享目录。定义在Pod之上，被一个Pod里的多个容器挂载到具体的文件目录之下；与Pod生命周期相同。

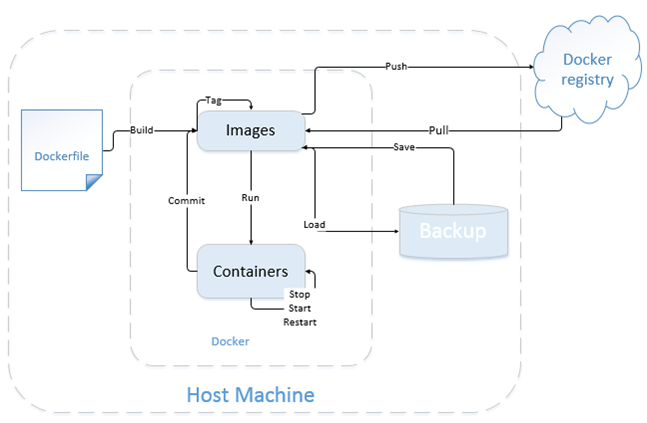
可以让一个Pod里的多个容器共享文件、让容器的数据写到宿主机的磁盘上或者写文件到 网络存储中，具体如下图所示：



        在kubernetes1.2的时候，RC就由Replication Controller升级成Replica Set，“下一代RC”。命令兼容适用，Replica Set主要被Deployment这个更高层的资源对象所使用，从而形成一套Pod创建、删除、更新的编排机制。当我们使用Deployment时，无需关心它是如何创建和维护ReplicaSet的，这一切是自动发生的。

docker

        既然k8s是基于容器的，那么就不得不提到docker。2013年初，docker横空出世，孕育着新思想的“容器”，Docker选择容器作为核心和基础，以容器为资源分割和调度的基本单位，封装整个软件运行时环境，为开发者和系统管理员设计，用于构建、发布和运行分布式应用的平台。是一个跨平台、可移植并且简单易用的容器解决方案。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。



        上图是一个image的简单使用。我们可以通过一个dockerfile来build自己的image。可以把image上传（push）到自己的私有镜像仓库，也可以从私有仓库pull到本地进行使用。可以单独使用命令行，直接run container，可以对container进行stop、start、restart操作。也可以对image进行save保存操作以及加载load操作，大家具体可以根据自己的使用，选择不同的操作即可。

docker资源隔离技术

        2013年初，docker横空出世，孕育着新思想的“容器”，Docker选择容器作为核心和基础，以容器为资源分割和调度的基本单位，封装整个软件运行时环境，为开发者和系统管理员设计，用于构建、发布和运行分布式应用的平台。是一个跨平台、可移植并且简单易用的容器解决方案。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。（关于这两种资源隔离技术，本人只能从功能上说明两者相当于就是分组，隔离，但是具体的内部原理还不甚了解，所以此处只提下概念，如皋想要深入了解，可搜集其他资料）

docker监控

        cAdvisor（Container Advisor）是Google开发的用于分析运行中容器的资源占用和性能指标的开源工具。cAdvisor是一个运行时的守护进程，负责收集、聚合、处理和输出运行中容器的信息。对于每个容器，cAdvisor都有资源隔离参数、资源使用历史情况以及完整的历史资源使用和网络统计信息的柱状图。

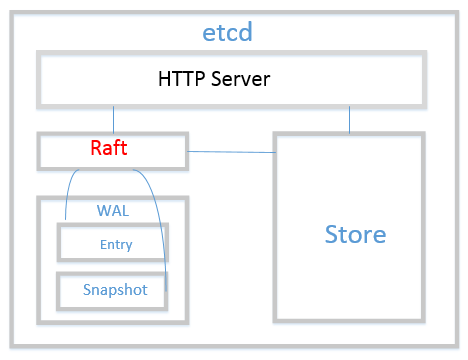
        cAdvisor不但可以为用户提供监控服务，还可以结合其他应用为用户提供良好的服务移植和定制。包括结合InfluxDB对数据进行存储，以及结合Grafana提供web控制台，自定义查询指标，并进行展示。



etcd

     etcd是一个键值存储仓库，用于配置共享和服务发现。etcd受Zookeeper与doozer启发而催生的项目。

etcd架构



etcd存储

        etcd的存储分为内部存储和持久化（硬盘）存储两部分。内存中的存储除了顺序化地记录所有用户对节点数据变更的记录外，还会对用户数据进行索引、建堆等方便查询的操作。而持久化则使用WAL进行记录存储。在k8s中，所有数据的存储以及操作记录都在etcd中进行存储，所以对于k8s集群来说，etcd是相当重要的，一旦故障，可能导致整个集群的瘫痪或者数据丢失。

   在WAL体系中，所有的数据在提交之前都会进行日志记录。持久化存储的目录分为两个：snap和wal。snapshot相当于数据压缩，默认会将10000条wal操作记录merge成snapshot，节省存储，又保证数据不会丢失。

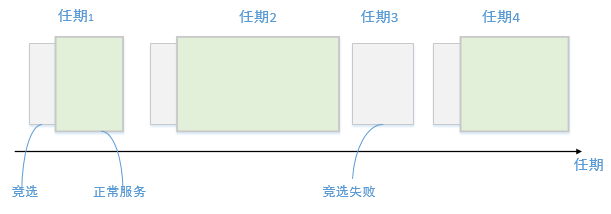
   WAL：存储所有事务的变化记录

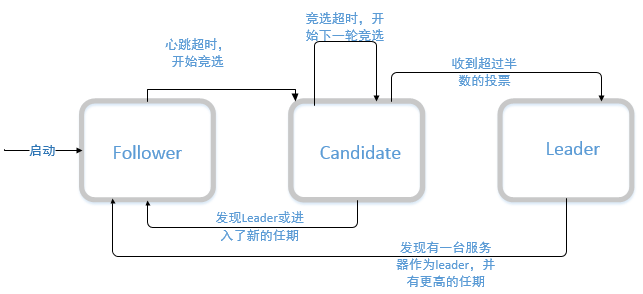
   Snapshot：用于存放某一时刻etcd所有目录的数据

思考：数据损坏或者机器故障怎么办？？？

etcd核心算法

etcd的核心算法是raft算法，强一致性算法。具体如下图所示

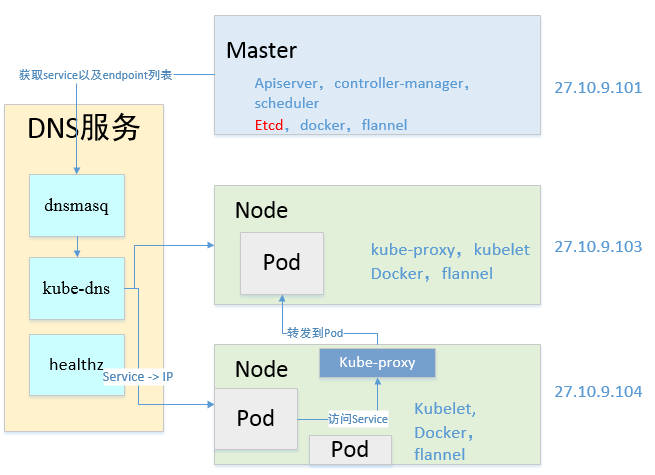




注意：由于etcd是负责存储，所以不建议搭建单点集群，如zookeeper一样，由于存在选举策略，所以一般推荐奇数个集群，如3，5，7。只要集群半数以上的结点存活，那么集群就可以正常运行，否则集群可能无法正常使用。

k8s集群部署方案

如下是我的集群部署策略，1个master + 2个node（minion1.2之前的叫法）。我的存储集群etcd是单点集群，不推荐此做法，你懂得，哈哈。网络使用的是flannel虚拟二次网络。



如何验证？

搭建完成之后，命令行执行：kubectl  get  no    查看节点状态是否ready。

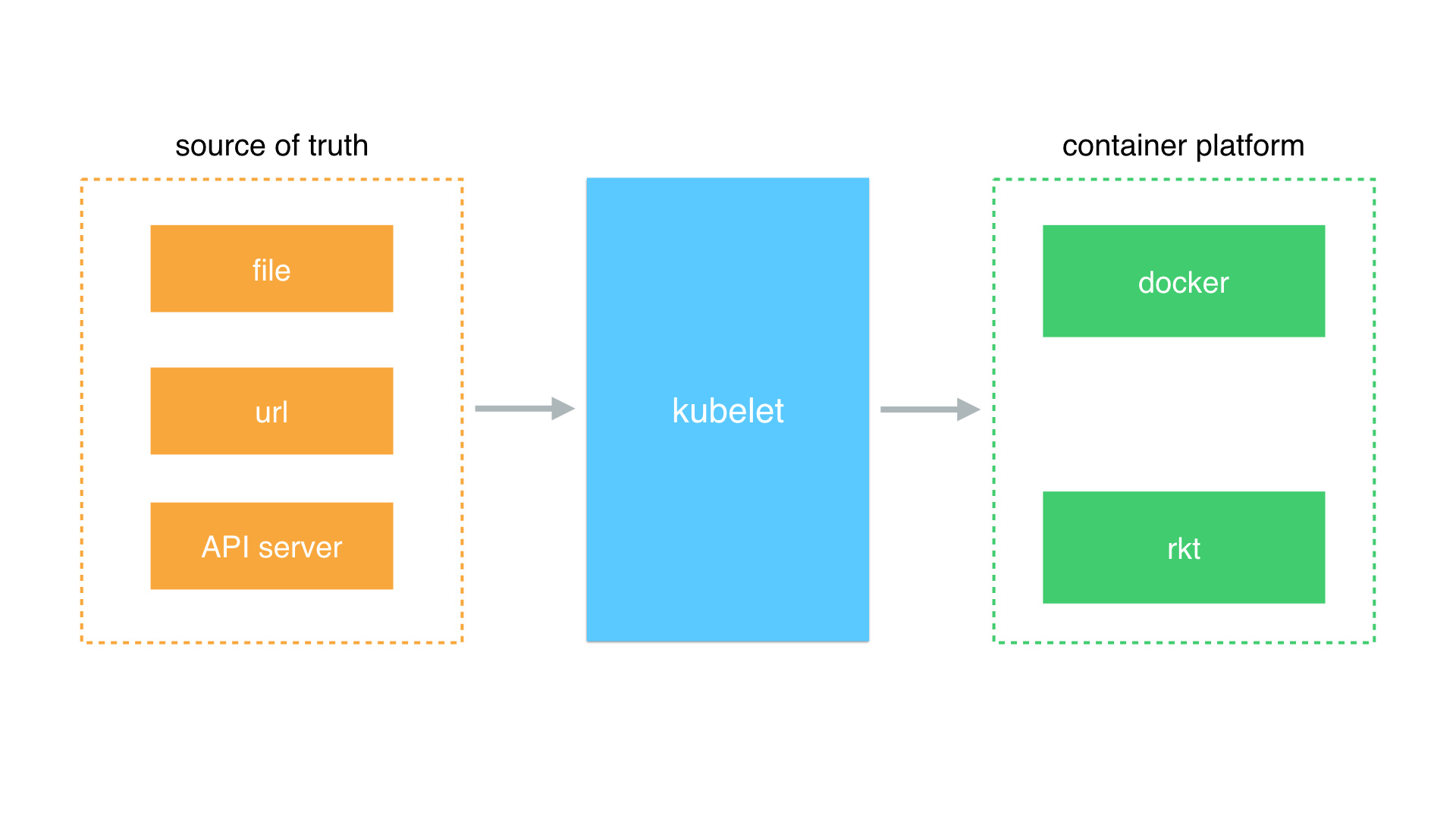
其他的大家可以通过代码来进行验证，看是否可以成功访问各个服务以及成功运行。

**简介**

[kubernetes](http://cizixs.com/2016/07/12/kubernetes-intro) 是一个分布式的集群管理系统，在每个节点（node）上都要运行一个 worker 对容器进行生命周期的管理，这个 worker 程序就是 kubelet。

简单地说，kubelet 的主要功能就是定时从某个地方获取节点上 pod/container 的期望状态（运行什么容器、运行的副本数量、网络或者存储如何配置等等），并调用对应的容器平台接口达到这个状态。

集群状态下，kubelet 会从 master 上读取信息，但其实 kubelet 还可以从其他地方获取节点的 pod 信息。目前 kubelet 支持三种数据源：



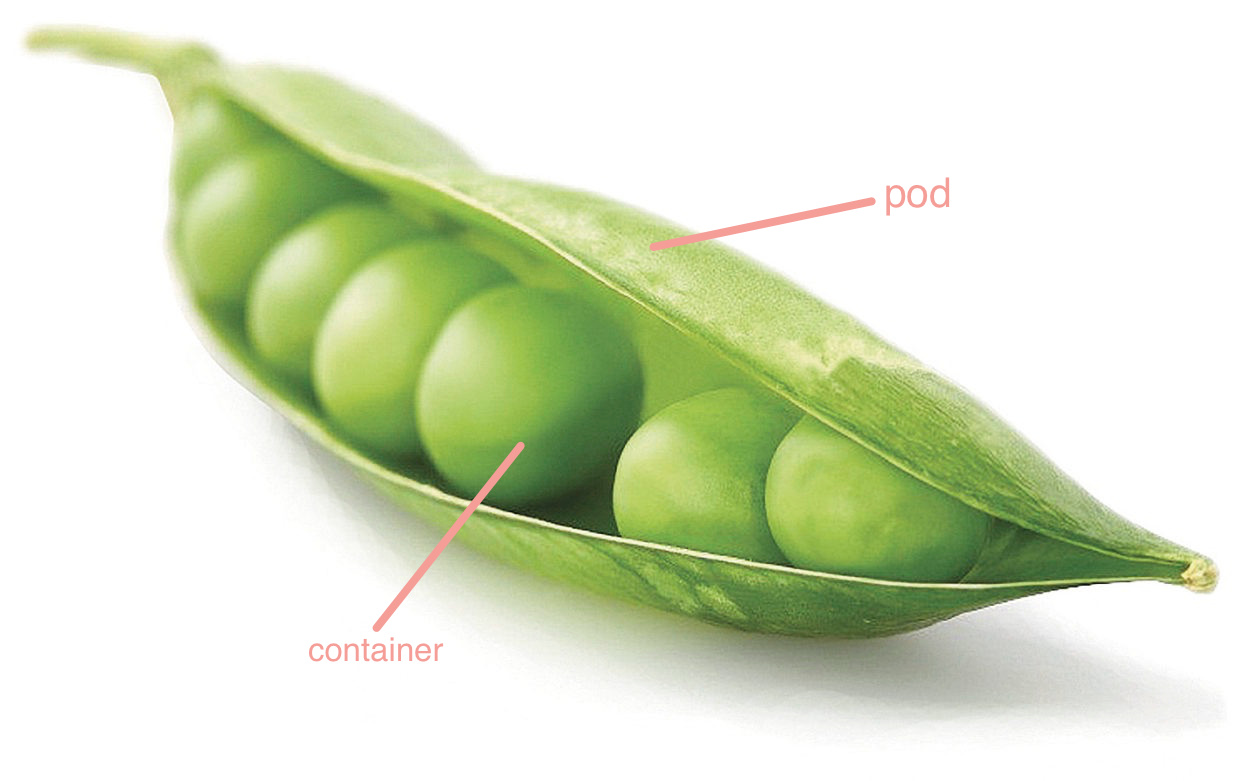
1. 本地文件
2. 通过 url 从网络上某个地址来获取信息
3. API Server：从 kubernetes master 节点获取信息

从管理的对象来说，kubelet 目前支持 [docker](https://www.docker.com/) 和 [rkt](https://coreos.com/rkt/)，默认情况下使用的 docker。

**kubelet 主要功能**

**pod 管理**

在 kubernetes 的设计中，最基本的管理单位是 pod，而不是 container。pod 是 kubernetes 在容器上的一层封装，由一组运行在同一主机的一个或者多个容器组成。如果把容器比喻成传统机器上的一个进程（它可以执行任务，对外提供某种功能），那么 pod 可以类比为传统的主机：它包含了多个容器，为它们提供共享的一些资源。



之所以费功夫提供这一层封装，主要是因为容器推荐的用法是里面只运行一个进程，而一般情况下某个应用都由多个组件构成的。

pod 中所有的容器最大的特性也是最大的好处就是共享了很多资源，比如网络空间。pod 下所有容器共享网络和端口空间，也就是它们之间可以通过 localhost 访问和通信，对外的通信方式也是一样的，省去了很多容器通信的麻烦。

除了网络之外，定义在 pod 里的 volume 也可以 mount 到多个容器里，以实现共享的目的。

最后，定义在 pod 的资源限制（比如 CPU 和 Memory） 也是所有容器共享的。

**容器健康检查**

创建了容器之后，kubelet 还要查看容器是否正常运行，如果容器运行出错，就要根据设置的重启策略进行处理。检查容器是否健康主要有两种方式：在容器中执行命令和通过 HTTP 访问预定义的 endpoint。

先来说说执行命令的方式，简单来说就是在容器中执行某个 shell 命令，根据它的 exit code 来判断容器是否正常工作：

livenessProbe:

**exec**:

**command**:

- cat

- /tmp/health

initialDelaySeconds: 15

timeoutSeconds: 1

另外一种就是 HTTP 的方式，通过向某个 url 路径发送 HTTP GET 请求，根据 response code 判断容器是否正常工作：

livenessProbe:

httpGet:

path: /healthz

port: 8080

httpHeaders:

- name: X-Custom-Header

value: Awesome

initialDelaySeconds: 15

timeoutSeconds: 1 可以看到，你也可以自定义发送的 HTTP 请求的头部。

不管用什么方式，如果检测到容器不健康，kubelet 会删除该容器，并根据容器的重启策略进行处理（比如重启，或者什么都不做）。

**容器监控**

kubelet 还有一个重要的责任，就是监控所在节点的资源使用情况，并定时向 master 报告。知道整个集群所有节点的资源情况，对于 pod 的调度和正常运行至关重要。

kubelet 使用 [cAdvisor](https://github.com/google/cadvisor) 进行资源使用率的监控。cAdvisor 是 google 开源的分析容器资源使用和性能特性的工具，在 kubernetes 项目中被集成到 kubelet 里，无需额外配置。默认情况下，你可以在 http://<host\_ip>:4194 地址看到 cAdvisor 的管理界面。

*Analyzes resource usage and performance characteristics of running containers.*

除了系统使用的 CPU，Memory，存储和网络之外，cAdvisor 还记录了每个容器使用的上述资源情况。

**运行和验证 kubelet**

**参数介绍**

对于 kubelet 的配置，基本上都可以通过命令行启动时候的参数进行控制。因为 kubernetes 处于快速开发过程中，参数也可能会发生变化，这里给出 1.4 版本一些重要的参数含义：

| **参数** | **解释** | **默认值** |
| --- | --- | --- |
| –address | kubelet 服务监听的地址 | 0.0.0.0 |
| –port | kubelet 服务监听的端口 | 10250 |
| –read-only-port | 只读端口，可以不用验证和授权机制，直接访问 | 10255 |
| –allow-privileged | 是否允许容器运行在 privileged 模式 | false |
| –api-servers | 以逗号分割的 API Server 地址，用于和集群中数据交互 | [] |
| –cadvisor-port | 当前节点 cadvisor 运行的端口 | 4194 |
| –config | 本地 manifest 文件的路径或者目录 | ”” |
| –file-check-frequency | 轮询本地 manifest 文件的时间间隔 | 20s |
| –container-runtime | 后端容器 runtime，支持 docker 和 rkt | docker |
| –enable-server | 是否启动 kubelet HTTP server | true |
| –healthz-bind-address | 健康检查服务绑定的地址，设置成 0.0.0.0 可以监听在所有网络接口 | 127.0.0.1 |
| –healthz-port | 健康检查服务的端口 | 10248 |
| –hostname-override | 指定 hostname，如果非空会使用这个值作为节点在集群中的标识 | ”” |
| –log-dir | 日志文件，如果非空，会把 log 写到该文件 | ”” |
| –logtostderr | 是否打印 log 到终端 | true |
| –max-open-files | 允许 kubelet 打开文件的最大值 | 1000000 |
| –max-pods | 允许 kubelet 运行 pod 的最大值 | 110 |
| –pod-infra-container-image | 基础镜像地址，每个 pod 最先启动的容器，会配置共享的网络 | gcr.io/google\_containers/pause-amd64:3.0 |
| –root-dir | kubelet 保存数据的目录 | /var/lib/kubelet |
| –runonce | 从本地 manifest 或者 URL 指定的 manifest 读取并运行结束就退出，和 --api-servers 、--enable-server 参数不兼容 |  |
| –v | 日志 level | 0 |

**启动 kubelet**

在这篇教程中，我们会单独运行 kubelet，选择从本地目录中读取 pod manifest 信息去启动 pod。

[Unit]

Description=Kubelet Service

Documentation=http://kubernetes.com

After=network.target

Wants=network.target

[Service]

Type=simple

EnvironmentFile=-/etc/sysconfig/kubelet

ExecStartPre=/usr/bin/mkdir -p /etc/kubernetes/manifests

ExecStart=/usr/bin/kubelet \

--allow-privileged=true \

--config=/etc/kubernetes/manifests \

--pod-infra-container-image=172.16.1.41:5000/google\_containers/pause:0.8.0 \

--hostname-override=192.168.8.100

TimeoutStartSec=0

Restart=on-abnormal

[Install]

WantedBy=multi-user.target

在这篇文章的例子中，我们会运行一个 pod，这个 pod 有两个容器：nginx 和 busybox。 nginx 不做任何修改，只是提供简单的 HTTP 服务；busybox 运行命令，在终端打印出 nginx 的访问日志。为了实现两个容器之间共享数据，我们使用 kubernetes 提供的 Volume 功能，

apiVersion: v1

kind: Pod

metadata:

name: nginx-server

spec:

containers:

- name: nginx-server

image: 172.16.1.41:5000/nginx

ports:

- containerPort: 80

volumeMounts:

- mountPath: /var/**log**/nginx

name: nginx-logs

- name: **log**-output

image: 172.16.1.41:5000/busybox

**command**:

- bin/sh

args: [-c, 'tail -f /logdir/access.log']

volumeMounts:

- mountPath: /logdir

name: nginx-logs

volumes:

- name: nginx-logs

emptyDir: {}

把这个配置文件放到指定的目录，等待一段时间，我们就能 docker ps 看到有容器在运行了：

[root@localhost vagrant]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

66474b8cc336 172.16.1.41:5000/busybox "bin/sh -c 'tail -f /" 6 days ago Up 6 days k8s\_log-output.b93a0be3\_nginx-server-192.168.8.100\_default\_dae69c6f50364bf8fcf20bc67f42a416\_22c30949

90ab8ef86868 172.16.1.41:5000/nginx "nginx" 6 days ago Up 6 days k8s\_nginx-server.b88325ae\_nginx-server-192.168.8.100\_default\_dae69c6f50364bf8fcf20bc67f42a416\_e152bb97

db27cfa9f58f 172.16.1.41:5000/google\_containers/pause:0.8.0 "/pause" 6 days ago Up 6 days k8s\_POD.b66e0279\_nginx-server-192.168.8.100\_default\_dae69c6f50364bf8fcf20bc67f42a416\_2236d469

我们可以看到三个容器：pause、nginx 和 busybox，容器的名字是 kubelet 自动生成的。通过 docker inspect 可以看到 nginx 和 busybox 的网络模式：

[root@localhost vagrant]# docker inspect -f '' 90ab

container:db27cfa9f58f7b1047fb230da951453ca15e0961ca36ee7999ab3ed8ba5da814

[root@localhost vagrant]# docker inspect -f '' 6647

container:db27cfa9f58f7b1047fb230da951453ca15e0961ca36ee7999ab3ed8ba5da814

使用的是 container 模式，对应的 id 就是 pause 容器的。通过查看各个容器的网络配置也能验证这一点，所有的容器网络都是一样的：

[root@localhost vagrant]# docker exec 90ab ip addr

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group **default**

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

inet6 ::1/128 scope host

valid\_lft forever preferred\_lft forever

9: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP group **default**

link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff

inet 172.17.0.2/16 scope global eth0

valid\_lft forever preferred\_lft forever

inet6 fe80::42:acff:fe11:2/64 scope link tentative dadfailed

valid\_lft forever preferred\_lft forever

而 nginx 和 busybox 的配置信息中并没有 ip 地址，pause 容器中有：

[root@localhost vagrant]# docker inspect -f '' 6647

[root@localhost vagrant]# docker inspect -f '' 90ab

[root@localhost vagrant]# docker inspect -f '' db27

172.17.0.2

好了，分析就这么多。我们看测试一下启动的容器是否正常，先通过 curl 访问几次 nginx：

[root@localhost vagrant]# curl -s http://172.17.0.2 | head -n 5

***<!DOCTYPE html>***

<**html**>

<**head**>

<**title**>Welcome to nginx on Debian!</**title**>

<**style**>

然后通过 busybox 容器查看访问日志：

[root@localhost vagrant]# docker logs -f 6647

172.17.0.1 - - [08/Sep/2016:03:29:47 +0000] "GET / HTTP/1.1" 200 867 "-" "curl/7.29.0"

172.17.0.1 - - [08/Sep/2016:03:49:57 +0000] "GET / HTTP/1.1" 200 867 "-" "curl/7.29.0"

172.17.0.1 - - [08/Sep/2016:03:50:18 +0000] "GET / HTTP/1.1" 200 867 "-" "curl/7.29.0"

172.17.0.1 - - [08/Sep/2016:03:50:23 +0000] "GET / HTTP/1.1" 200 867 "-" "curl/7.29.0"

随着请求的发送，可以看到日志都打印出来了。

**API 接口**

前面也提到了，kubelet 自身提供了几个 HTTP API 接口，供用户查看信息。 最简单的就是 10248 的健康检查：

[root@localhost vagrant]# curl http://127.0.0.1:10248/healthz

ok

而 10255 端口比较重要，提供了 pod 和机器的信息：

➜ ~ http http://192.168.8.100:10255/pods

HTTP/1.1 200 OK

Content-Length: 1345

Content-Type: application/json

Date: Thu, 08 Sep 2016 04:03:38 GMT

{

"apiVersion": "v1",

"items": [

{

"metadata": {

"annotations": {

"kubernetes.io/config.hash": "bf570970380d322fa28f20a8fe4b61bc",

"kubernetes.io/config.seen": "2016-09-01T09:50:16.379538472+02:00",

"kubernetes.io/config.source": "file"

},

"creationTimestamp": null,

"name": "nginx-server-192.168.8.100",

"namespace": "default",

"selfLink": "/api/v1/pods/namespaces/nginx-server-192.168.8.100/default",

"uid": "dae69c6f50364bf8fcf20bc67f42a416"

},

"spec": {

"containers": [

{

"image": "172.16.1.41:5000/nginx",

"imagePullPolicy": "Always",

"name": "nginx-server",

"ports": [

{

"containerPort": 80,

"protocol": "TCP"

}

],

"resources": {},

"terminationMessagePath": "/dev/termination-log",

"volumeMounts": [

{

"mountPath": "/var/log/nginx",

"name": "nginx-logs"

}

]

},

{

"args": [

"-c",

"tail -f /logdir/access.log"

],

"command": [

"bin/sh"

],

"image": "172.16.1.41:5000/busybox",

"imagePullPolicy": "Always",

"name": "log-output",

"resources": {},

"terminationMessagePath": "/dev/termination-log",

"volumeMounts": [

{

"mountPath": "/logdir",

"name": "nginx-logs"

}

]

}

],

"dnsPolicy": "ClusterFirst",

"nodeName": "192.168.8.100",

"restartPolicy": "Always",

"securityContext": {},

"terminationGracePeriodSeconds": 30,

"volumes": [

{

"emptyDir": {},

"name": "nginx-logs"

}

]

},

"status": {

"conditions": [

{

"lastProbeTime": null,

"lastTransitionTime": "2016-09-01T07:52:36Z",

"status": "True",

"type": "PodScheduled"

}

],

"phase": "Pending"

}

}

],

"kind": "PodList",

"metadata": {}

}

pods 列出了节点上运行的 pod 信息，包括里面有多少容器、pod 的状态、一些 metadata 等。 /spec/ 路径返回节点的信息：

➜ ~ http http://192.168.8.100:10255/spec/

HTTP/1.1 200 OK

Content-Type: application/json

Date: Thu, 08 Sep 2016 04:06:53 GMT

Transfer-Encoding: chunked

{

"boot\_id": "48955926-11dd-4ad3-8bb0-2585b1c9215d",

"cloud\_provider": "Unknown",

"cpu\_frequency\_khz": 2194976,

"disk\_map": {

"253:0": {

"major": 253,

"minor": 0,

"name": "dm-0",

"scheduler": "none",

"size": 1107296256

},

...

"8:0": {

"major": 8,

"minor": 0,

"name": "sda",

"scheduler": "cfq",

"size": 42949672960

},

"8:16": {

"major": 8,

"minor": 16,

"name": "sdb",

"scheduler": "cfq",

"size": 21474836480

},

...

},

"filesystems": [

{

"capacity": 41293721600,

"device": "/dev/mapper/centos-root",

"inodes": 40345600,

"type": "vfs"

},

{

"capacity": 520794112,

"device": "/dev/sda1",

"inodes": 512000,

"type": "vfs"

},

{

"capacity": 21464350720,

"device": "/dev/sdb",

"inodes": 20971520,

"type": "vfs"

},

...

],

"instance\_id": "None",

"instance\_type": "Unknown",

"machine\_id": "b9597c4ae5f24494833d35e806e00b29",

"memory\_capacity": 514215936,

"network\_devices": [

{

"mac\_address": "08:00:27:c0:9e:5b",

"mtu": 1500,

"name": "enp0s3",

"speed": 1000

},

{

"mac\_address": "08:00:27:9b:5c:b5",

"mtu": 1500,

"name": "enp0s8",

"speed": 1000

}

],

"num\_cores": 1,

"system\_uuid": "823EB67A-057E-4EFF-AE7F-A758140CD2F7",

"topology": [

{

"caches": [

{

"level": 3,

"size": 3145728,

"type": "Unified"

}

],

"cores": [

{

"caches": [

{

"level": 1,

"size": 32768,

"type": "Data"

},

{

"level": 1,

"size": 32768,

"type": "Instruction"

},

{

"level": 2,

"size": 262144,

"type": "Unified"

}

],

"core\_id": 0,

"thread\_ids": [

0

]

}

],

"memory": 536403968,

"node\_id": 0

}

]

}

磁盘、网络、CPU、内存等信息，这些东西是 kubernetes 集群调度的重要依据，在集群模式中会更新到 master 节点。

**清理工作**

如果删除 manifests/ 路径下的文件，等一段时间就会发现，kubelet 已经清空了所有的容器。

$ curl -s http://localhost:10255/pods

{"kind":"PodList","apiVersion":"v1","metadata":{},"items":null}

**参考链接**

* [Deploy Kubernetes Master Node(s)](https://coreos.com/kubernetes/docs/latest/deploy-master.html)
* [What even is a kubelet?](http://kamalmarhubi.com/blog/2015/08/27/what-even-is-a-kubelet/)
* [Kubernetes Node Documentation](http://kubernetes.io/docs/admin/node/)
* [Overview of a Pod](https://coreos.com/kubernetes/docs/latest/pods.html)
* [Checking Pod Health](http://kubernetes.io/docs/user-guide/liveness/)

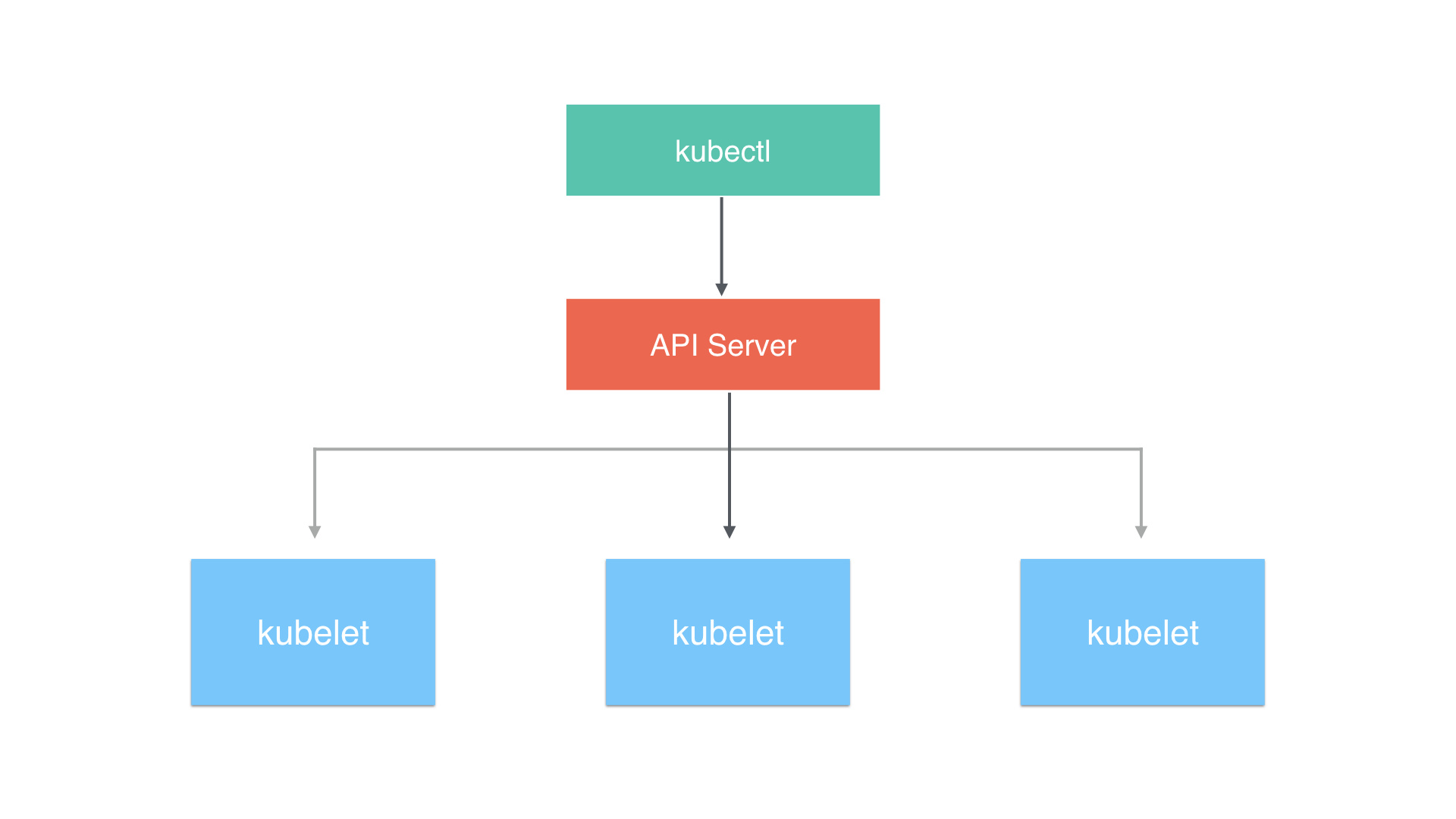
简介

这篇文章是整个系统的第二篇，[上一篇文章](http://cizixs.com/2016/10/25/kubernetes-intro-kubelet)讲解了 kubelet 的功能和使用。

在 kubernetes 集群中，API Server 有着非常重要的角色。API Server 负责和 etcd 交互（其他组件不会直接操作 etcd，只有 API Server 这么做），是整个 kubernetes 集群的数据中心，所有的交互都是以 API Server 为核心的。简单来说，API Server 提供了一下的功能：

* 整个集群管理的 API 接口：所有对集群进行的查询和管理都要通过 API 来进行
* 集群内部各个模块之间通信的枢纽：所有模块之前并不会之间互相调用，而是通过和 API Server 打交道来完成自己那部分的工作
* 集群安全控制：API Server 提供的验证和授权保证了整个集群的安全

在这篇教程中，我们的系统架构将变成下面这个样子：



我们把要配置的 pod 通过 kubectl 发送给 API Server，里面已经手动指定了要运行的节点。API Server 解析并保存对应的资源，对应的 kubelet 定时拉取数据时候发现 pod 是分配给自己的，会下载对应的配置并执行去生成 pod。

参数介绍

API Server 主要是和 etcd 打交道，并且对外提供 HTTP 服务，以及进行安全控制，因此它的命令行提供的参数也主要和这几个方面有关。下面是一些比较重要的参数以及说明（不同版本参数可能会有不同）：

| **参数** | **含义** | **默认值** |
| --- | --- | --- |
| –advertise-address | 通过该 ip 地址向集群其他节点公布 api server 的信息，必须能够被其他节点访问 | nil |
| –allow-privileged | 是否允许 privileged 容器运行 | false |
| –admission-control | 准入控制 | AlwaysAdmit |
| –authorization-mode | 授权模式 ，安全接口上的授权 | AlwaysAllow |
| –bind-address | HTTPS 安全接口的监听地址 | 0.0.0.0 |
| –secure-port | HTTPS 安全接口的监听端口 | 6443 |
| –cert-dir | TLS 证书的存放目录 | /var/run/kubernetes |
| –etcd-prefix | 信息存放在 etcd 中地址的前缀 | “/registry” |
| –etcd-servers | 逗号分割的 etcd server 地址 | [] |
| –insecure-bind-address | HTTP 访问的地址 | 127.0.0.1 |
| –insecure-port | HTTP 访问的端口 | 8080 |
| –log-dir | 日志存放的目录 |  |
| –service-cluster-ip-range | service 要使用的网段，使用 CIDR 格式，参考 kubernetes 中 service 的定义 |  |

安装和运行

API Server 是通过提供的 kube-apiserver 二进制文件直接运行的，下面的例子指定了 service 分配的 ip 范围，etcd 的地址，和对外提供服务的 ip 地址：

/usr/bin/kube-apiserver \

--service-cluster-ip-range=10.20.0.1/24 \

--etcd-servers=http://127.0.0.1:2379 \

--advertise-address=192.168.8.100 \

--bind-address=192.168.8.100 \

--insecure-bind-address=192.168.8.100 \

--v=4

这篇教程不会提供对 API Server 进行 HTTPS 的配置，所有的操作都是直接通过 8080 非安全端口访问的。

直接访问 8080 端口，API Server 会返回它提供了哪些接口：

[root@localhost vagrant]# curl http://192.168.8.100:8080

{

"paths": [

"/api",

"/api/v1",

"/apis",

"/apis/apps",

"/apis/apps/v1alpha1",

"/apis/autoscaling",

"/apis/autoscaling/v1",

"/apis/batch",

"/apis/batch/v1",

"/apis/batch/v2alpha1",

"/apis/extensions",

"/apis/extensions/v1beta1",

"/apis/policy",

"/apis/policy/v1alpha1",

"/apis/rbac.authorization.k8s.io",

"/apis/rbac.authorization.k8s.io/v1alpha1",

"/healthz",

"/healthz/ping",

"/logs/",

"/metrics",

"/swaggerapi/",

"/ui/",

"/version"

]

}

而目前最重要的路径是 /api/v1，里面包含了 kubernetes 所有资源的操作，比如下面的 nodes：

➜ ~ http http://192.168.8.100:8080/api/v1/nodes

HTTP/1.1 200 OK

Content-Length: 112

Content-Type: application/json

Date: Thu, 08 Sep 2016 08:14:45 GMT

{

"apiVersion": "v1",

"items": [],

"kind": "NodeList",

"metadata": {

"resourceVersion": "12",

"selfLink": "/api/v1/nodes"

}

}

API 以 json 的形式返回，会通过 apiVersion 来说明 API 版本号，kind 说明请求的是什么资源。不过这里面的内容是空的，因为目前还没有任何 kubelet 节点接入到我们的 API Server。对应的，pod 也是空的：

➜ ~ http http://192.168.8.100:8080/api/v1/pods

HTTP/1.1 200 OK

Content-Length: 110

Content-Type: application/json

Date: Thu, 08 Sep 2016 08:18:53 GMT

{

"apiVersion": "v1",

"items": [],

"kind": "PodList",

"metadata": {

"resourceVersion": "12",

"selfLink": "/api/v1/pods"

}

}

添加节点

添加节点也非常简单，启动　kubelet　的时候使用　--api-servers　指定要接入的　API Server 就行。kubelet 启动之后，会把自己注册到指定的 API Server，然后监听 API 对应 pod 的变化，根据 API 中 pod 的实际信息来管理节点上 pod 的生命周期。

现在访问 /api/v1/nodes 就能看到已经添加进来的节点：

➜ ~ http http://192.168.8.100:8080/api/v1/nodes

HTTP/1.1 200 OK

Content-Type: application/json

Date: Thu, 08 Sep 2016 08:27:44 GMT

Transfer-Encoding: chunked

{

"apiVersion": "v1",

"items": [

{

"metadata": {

"annotations": {

"volumes.kubernetes.io/controller-managed-attach-detach": "true"

},

"creationTimestamp": "2016-09-08T08:23:01Z",

"labels": {

"beta.kubernetes.io/arch": "amd64",

"beta.kubernetes.io/os": "linux",

"kubernetes.io/hostname": "192.168.8.100"

},

"name": "192.168.8.100",

"resourceVersion": "65",

"selfLink": "/api/v1/nodes/192.168.8.100",

"uid": "74e16eba-759d-11e6-b463-080027c09e5b"

},

"spec": {

"externalID": "192.168.8.100"

},

"status": {

"addresses": [

{

"address": "192.168.8.100",

"type": "LegacyHostIP"

},

{

"address": "192.168.8.100",

"type": "InternalIP"

}

],

"allocatable": {

"alpha.kubernetes.io/nvidia-gpu": "0",

"cpu": "1",

"memory": "502164Ki",

"pods": "110"

},

"capacity": {

"alpha.kubernetes.io/nvidia-gpu": "0",

"cpu": "1",

"memory": "502164Ki",

"pods": "110"

},

"conditions": [

{

"lastHeartbeatTime": "2016-09-08T08:27:36Z",

"lastTransitionTime": "2016-09-08T08:23:01Z",

"message": "kubelet has sufficient disk space available",

"reason": "KubeletHasSufficientDisk",

"status": "False",

"type": "OutOfDisk"

},

{

"lastHeartbeatTime": "2016-09-08T08:27:36Z",

"lastTransitionTime": "2016-09-08T08:23:01Z",

"message": "kubelet has sufficient memory available",

"reason": "KubeletHasSufficientMemory",

"status": "False",

"type": "MemoryPressure"

},

{

"lastHeartbeatTime": "2016-09-08T08:27:36Z",

"lastTransitionTime": "2016-09-08T08:24:56Z",

"message": "kubelet is posting ready status",

"reason": "KubeletReady",

"status": "True",

"type": "Ready"

}

],

"daemonEndpoints": {

"kubeletEndpoint": {

"Port": 10250

}

},

"images": [

{

"names": [

"172.16.1.41:5000/nginx:latest"

],

"sizeBytes": 425626718

},

{

"names": [

"172.16.1.41:5000/hyperkube:v0.18.2"

],

"sizeBytes": 207121551

},

{

"names": [

"172.16.1.41:5000/etcd:v3.0.4"

],

"sizeBytes": 43302056

},

{

"names": [

"172.16.1.41:5000/busybox:latest"

],

"sizeBytes": 1092588

},

{

"names": [

"172.16.1.41:5000/google\_containers/pause:0.8.0"

],

"sizeBytes": 241656

}

],

"nodeInfo": {

"architecture": "amd64",

"bootID": "48955926-11dd-4ad3-8bb0-2585b1c9215d",

"containerRuntimeVersion": "docker://1.10.3",

"kernelVersion": "3.10.0-123.13.1.el7.x86\_64",

"kubeProxyVersion": "v1.3.1-beta.0.6+fbf3f3e5292fb0",

"kubeletVersion": "v1.3.1-beta.0.6+fbf3f3e5292fb0",

"machineID": "b9597c4ae5f24494833d35e806e00b29",

"operatingSystem": "linux",

"osImage": "CentOS Linux 7 (Core)",

"systemUUID": "823EB67A-057E-4EFF-AE7F-A758140CD2F7"

}

}

}

],

"kind": "NodeList",

"metadata": {

"resourceVersion": "65",

"selfLink": "/api/v1/nodes"

}

}

我们可以看到，kubelet 收集了很多关于自身节点的信息，这些信息也会不断更新。这些信息里面不仅包含节点的系统信息（系统架构，操作系统版本，内核版本等）、还有镜像信息（节点上有哪些已经下载的 docker 镜像）、资源信息（Memory 和 Disk 的总量和可用量）、以及状态信息（是否正常，可以分配 pod等）。

和 API Server 通信

下面我们就通过 API Server 来创建 pod，而不是像上篇文章那样用拷贝文件的方式。我们把编写的 yaml 文件转换成 json 格式，保存到文件里。主要注意的是，我们指定了 nodeName 的名字，这个名字必须和之前通过 /api/v1/nodes 得到的结果中 metadata.labels.kubernetes.io/hostname 保持一致：

[root@localhost vagrant]# cat nginx\_pod.yml

apiVersion: v1

kind: Pod

metadata:

name: nginx-server

spec:

nodeName: 192.168.8.100

containers:

- name: nginx-server

image: 172.16.1.41:5000/nginx

ports:

- containerPort: 80

volumeMounts:

- mountPath: /var/**log**/nginx

name: nginx-logs

- name: **log**-output

image: 172.16.1.41:5000/busybox

**command**:

- bin/sh

args: [-c, 'tail -f /logdir/access.log']

volumeMounts:

- mountPath: /logdir

name: nginx-logs

volumes:

- name: nginx-logs

emptyDir: {}

使用 curl 执行 POST 请求，设置头部内容为 application/json，传过去文件中的 json 值，可以看到应答(其中 status 为 pending，表示以及接收到请求，正在准备处理)：

# curl -s -X POST -H "Content-Type: application/json" http://192.168.8.100:8080/api/v1/namespaces/default/pods --data @nginx\_pod.json

{

"kind": "Pod",

"apiVersion": "v1",

"metadata": {

"name": "nginx-server",

"namespace": "default",

"selfLink": "/api/v1/namespaces/default/pods/nginx-server",

"uid": "888e95d0-75a9-11e6-b463-080027c09e5b",

"resourceVersion": "573",

"creationTimestamp": "2016-09-08T09:49:28Z"

},

"spec": {

"volumes": [

{

"name": "nginx-logs",

"emptyDir": {}

}

],

"containers": [

{

"name": "nginx-server",

"image": "172.16.1.41:5000/nginx",

"ports": [

{

"containerPort": 80,

"protocol": "TCP"

}

],

"resources": {},

"volumeMounts": [

{

"name": "nginx-logs",

"mountPath": "/var/log/nginx"

}

],

"terminationMessagePath": "/dev/termination-log",

"imagePullPolicy": "Always"

}

],

"restartPolicy": "Always",

"terminationGracePeriodSeconds": 30,

"dnsPolicy": "ClusterFirst",

"nodeName": "192.168.8.100",

"securityContext": {}

},

"status": {

"phase": "Pending"

}

}

返回中包含了我们提交 pod 的信息，并且添加了 status、metadata 等额外信息。

等一段时间去查询 pod，就可以看到 pod 的状态已经更新了：

➜ http http://192.168.8.100:8080/api/v1/namespaces/default/pods

HTTP/1.1 200 OK

Content-Type: application/json

Date: Thu, 08 Sep 2016 09:51:29 GMT

Transfer-Encoding: chunked

{

"apiVersion": "v1",

"items": [

{

"metadata": {

"creationTimestamp": "2016-09-08T09:49:28Z",

"name": "nginx-server",

"namespace": "default",

"resourceVersion": "592",

"selfLink": "/api/v1/namespaces/default/pods/nginx-server",

"uid": "888e95d0-75a9-11e6-b463-080027c09e5b"

},

"spec": {

"containers": [

{

"image": "172.16.1.41:5000/nginx",

"imagePullPolicy": "Always",

"name": "nginx-server",

"ports": [

{

"containerPort": 80,

"protocol": "TCP"

}

],

"resources": {},

"terminationMessagePath": "/dev/termination-log",

"volumeMounts": [

{

"mountPath": "/var/log/nginx",

"name": "nginx-logs"

}

]

},

{

"args": [

"-c",

"tail -f /logdir/access.log"

],

"command": [

"bin/sh"

],

"image": "172.16.1.41:5000/busybox",

"imagePullPolicy": "Always",

"name": "log-output",

"resources": {},

"terminationMessagePath": "/dev/termination-log",

"volumeMounts": [

{

"mountPath": "/logdir",

"name": "nginx-logs"

}

]

}

],

"dnsPolicy": "ClusterFirst",

"nodeName": "192.168.8.100",

"restartPolicy": "Always",

"securityContext": {},

"terminationGracePeriodSeconds": 30,

"volumes": [

{

"emptyDir": {},

"name": "nginx-logs"

}

]

},

"status": {

"conditions": [

{

"lastProbeTime": null,

"lastTransitionTime": "2016-09-08T09:49:28Z",

"status": "True",

"type": "Initialized"

},

{

"lastProbeTime": null,

"lastTransitionTime": "2016-09-08T09:49:44Z",

"status": "True",

"type": "Ready"

},

{

"lastProbeTime": null,

"lastTransitionTime": "2016-09-08T09:49:44Z",

"status": "True",

"type": "PodScheduled"

}

],

"containerStatuses": [

{

"containerID": "docker://8b79eeea60f27b6d3f0a19cbd1b3ee3f83709bcf56574a6e1124c69a6376972d",

"image": "172.16.1.41:5000/busybox",

"imageID": "docker://sha256:8c566faa3abdaebc33d40c1b5e566374c975d17754c69370f78c00c162c1e075",

"lastState": {},

"name": "log-output",

"ready": true,

"restartCount": 0,

"state": {

"running": {

"startedAt": "2016-09-08T09:49:43Z"

}

}

},

{

"containerID": "docker://96e64cdba7b05d4e30710a20e958ff5b8f1f359c8d16d32622b36f0df0cb353c",

"image": "172.16.1.41:5000/nginx",

"imageID": "docker://sha256:51d764c1fd358ce81fd0e728436bd0175ff1f3fd85fc5d1a2f9ba3e7dc6bbaf6",

"lastState": {},

"name": "nginx-server",

"ready": true,

"restartCount": 0,

"state": {

"running": {

"startedAt": "2016-09-08T09:49:36Z"

}

}

}

],

"hostIP": "192.168.8.100",

"phase": "Running",

"podIP": "172.17.0.2",

"startTime": "2016-09-08T09:49:28Z"

}

}

],

"kind": "PodList",

"metadata": {

"resourceVersion": "602",

"selfLink": "/api/v1/namespaces/default/pods"

}

}

可以看到 pod 已经在运行，并且给分配了 ip：172.17.0.2，通过 curl 也可以访问它的服务：

[root@localhost vagrant]# curl -s http://172.17.0.2 | head -n 5

*<!DOCTYPE html>*

<**html**>

<**head**>

<**title**>Welcome to nginx on Debian!</**title**>

<**style**>

关于 API Server 提供的所有接口，可以参考官方的 [API 文档][http://kamalmarhubi.com/blog/2015/09/06/kubernetes-from-the-ground-up-the-api-server/]。

kubectl 命令

理论上，所有 kubernetes 提供的功能都能够直接通过 HTTP API 交互来实现，但是你也看到了，非常复杂。因此 kubernetes 提供了另外了 kubectl 命令行，它封装了 HTTP API 的交互过程，通过一系列的子命令来操作资源。比如我们上面创建 pod 的过程就可以通过

kubectl **create** -**f** nginx\_pod.yml

一行命令实现，查看 pod 的信息也很简单：

**kubectl** -s http://ip:8080 get pods

删除已经创建的 pod，可以使用 delete 命令：

kubectl **delete** pods/nginx-**server**

kubectl 的用法也有[详细的官方文档](http://kubernetes.io/docs/user-guide/kubectl-overview/)，这里就不多说了。

窗体顶端

窗体底端