Table of Contents

[DevOps 1](#_Toc139277345)

[**虚拟开发环境(VirtualBox, Vagrant, Docker)** 4](#_Toc139277346)

[**VirtualBox** 4](#_Toc139277347)

[Vagrant 5](#_Toc139277348)

[Docker 14](#_Toc139277349)

[Containerize app 14](#_Toc139277350)

[Dockerfile指令 17](#_Toc139277351)

[Docker命令 18](#_Toc139277352)

[Docker镜像和仓库 20](#_Toc139277353)

[范例：测试静态网站，web服务， 22](#_Toc139277354)

[实战：java程序，Spring Boot, 30](#_Toc139277355)

[Bundle 33](#_Toc139277356)

[实践: Knowledge module, mysql, meadia wiki 35](#_Toc139277357)

[Deployment and Orchestration 37](#_Toc139277358)

[Kubernetes 38](#_Toc139277359)

[Docker Swarm [1.12 RC] 39](#_Toc139277360)

[Tilt (microservice dev env) 41](#_Toc139277361)

[Tiltfile (written in Starlark, subset of Python) 41](#_Toc139277362)

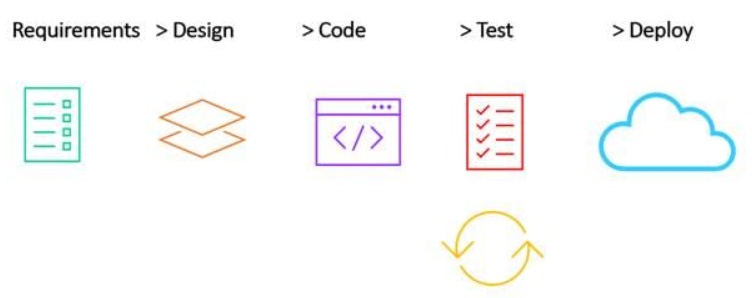
[Tilt CLI 43](#_Toc139277363)

[Remote Development (本地vscode,远程服务器要装gcc, gdb) 44](#_Toc139277364)

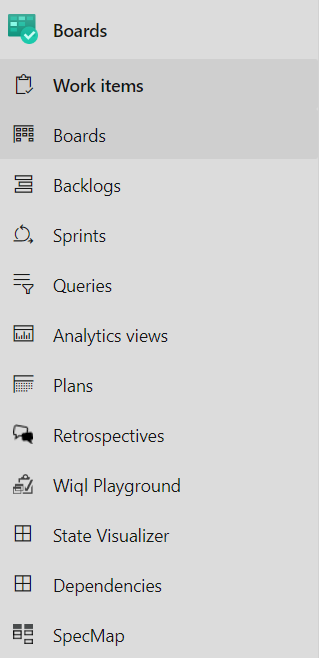
# DevOps

典型的软件开发团队由软件开发人员、测试人员、DevOps 工程师、架构师和项目经理组成。

Azure DevOps 是一个云平台,它提供了在云中管理整个软件开发生命周期的工具。



Azure Boards:



Azure Repos: 源代码版本控制

A screenshot of a computer

Description automatically generated with medium confidence

Azure Repos 实际上就是 Repository，代码的仓库，也就是类似于 Github，SVN，TFVC，Subversion 等等，对代码进行统一托管的服务。

使用策略保护分支: 将代码提交到 master 分之前需要先通过 CodeReview

Git设置

通过 Pull Request 修改；最少需要一个 Code reviewer；PR 每次发生更改都重置代码审阅者的投票。

Branches -> master 分支 -> More … -> Branch policies -> When new changes are pushed:

Minimum number of reviewers: 1

Reviewers: xxx, xxx, …

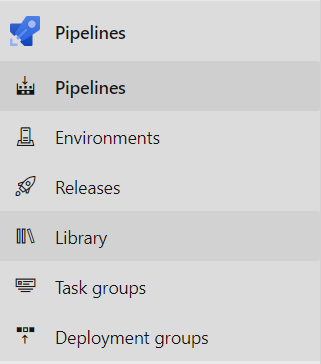
用户修改代码

新建分支, 修改代码 -> submit all and push

Git -> Pull requests -> Create a pull request

1. 在创建 Pull request 的页面可以看到这个 PR 有 1 个提交并修改了 文件，系统已经贴心帮我填好 Title，并关联了两个工作项。点击“Create”创建完成 Pull request 的创建。
2. Pull request 创建后，在 PR 的详细页面可以看到它的内容、是否冲突、关联的工作项、历史记录等。这时候只需要等待一个 code reviewer 审核通过.
3. reviewer 会收到通知要做 review，然后他就可以来看看这个 Pull request 做了些什么，没问题的话他就可以 Approve 这个 Pull request。
4. 通过后右上角的蓝色按钮会变成“Complete”，点击即可完成这个 PR 并将代码合并到 master 分支。

Azure Pipelines (构建和发布服务的工具): 软件项目的持续集成 (CI) 和持续交付 (CD) 是一项自动化的主要任务, 其中工具用于获取代码的最新版本、构建代码并部署到目的地。

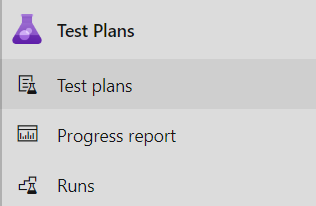


Azure Pipelines 支持构建、测试和部署 Node.js、Python、Java、PHP、Ruby、C/C++、.NET、Android 和 iOS 应用。在 Linux、macOS 和 Windows 上并行运行。

Azure Pipelines 支持容器和 Kubernetes, 允许团队轻松构建映像并将其推送到 Docker Hub 和 Azure Container Registry 等容器注册表。

将容器部署到单个主机或 Kubernetes。

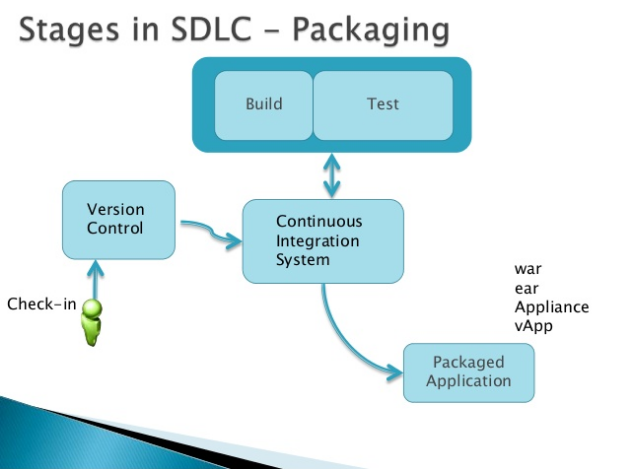
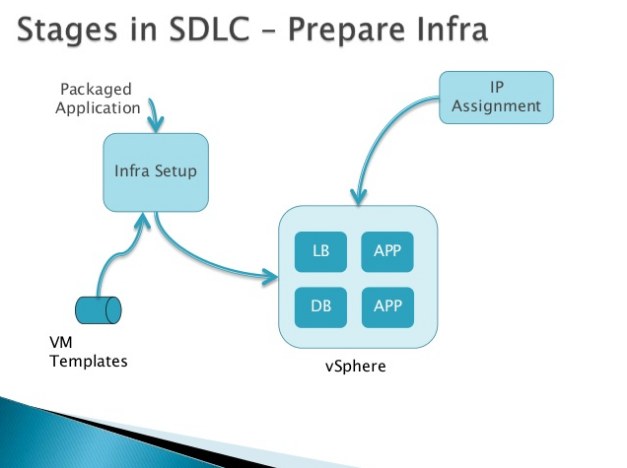
Azure Test Plans: 软件测试和 QA 团队创建测试用例、测试和报告结果以及生成报告



Azure Artifacts: 允许团队共享来自公共和私有源的包,例如 Maven、npm、NuGet 等,并将包共享集成到 CI/CD 管道中

Check-in -> Build and Test -> Infrastructure setup -> Release -> Monitoring and scaling

Git/SVN, Gradle, gulp, yeoman (yo, bower and grunt), jHipster, angular-fullstack, Docker/vagrant+puppet,

# **虚拟开发环境(Hyper-V, VirtualBox, Vagrant, Docker)**

虚拟化平台，用于在单台物理计算机上运行多个虚拟机（VM）。每个虚拟机都像一台完整的计算机一样运行操作系统和程序

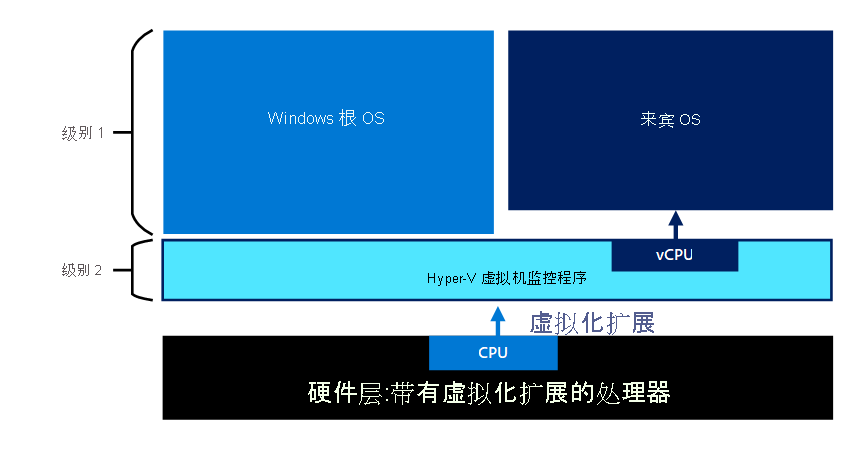
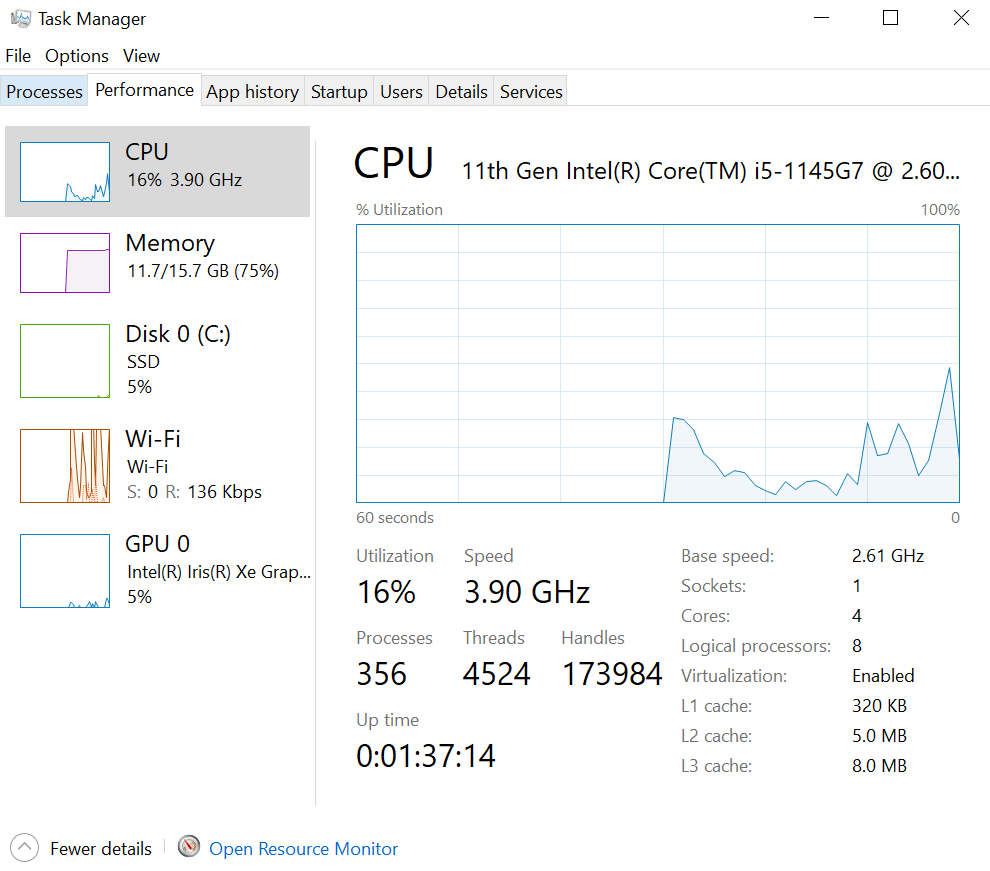
虚拟开发环境

问题：在开发机上面开发完毕程序，放到正式环境之后会出现各种奇怪的问题：描述符少了、nginx配置不正确、MySQL编码不对、php缺少模块、glibc版本太低等。

解决：需要虚拟开发环境，在本机跑虚拟机，例如VMware、VirtualBox等。在本机可以运行自己喜欢的OS（Windows、Ubuntu、Mac等），开发的程序运行在虚拟机中，这样迁移到生产环境可以避免环境不一致导致的莫名错误。

## Microsoft Hyper-V

与其他虚拟化平台相比，Hyper-V的功能相对较少。例如，它不支持远程连接，不能直接使用USB设备等

**** 

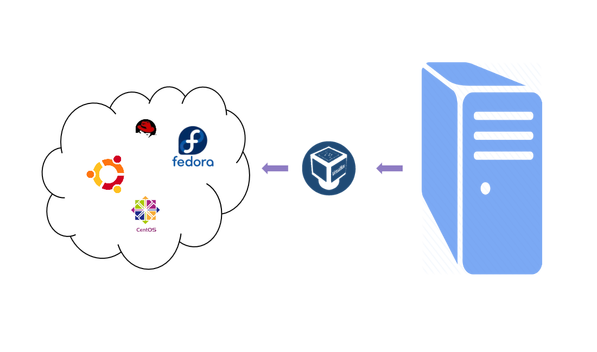
可以在任务管理器的性能选项卡看到自己的电脑是否开启了CPU虚拟化

Apps & features -> Programs and Features -> Turn Windows Features on or off -> Hyper-V

## **VirtualBox**

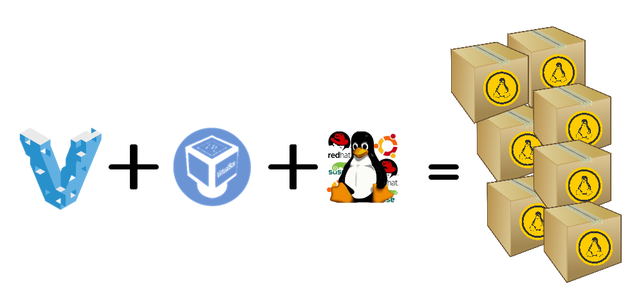
Hypervisor (2007) ，一种运行在基础物理服务器和操作系统之间的中间软件层,可允许多个操作系统和应用共享硬件。也可叫做VMM（ virtual machine monitor ），即虚拟机监视器。

Hypervisors是一种在虚拟环境中的“元”操作系统。他们可以访问服务器上包括磁盘和内存在内的所有物理设备。Hypervisors不但协调着这些硬件资源的访问，也同时在各个虚拟机之间施加防护。**当服务器启动并执行Hypervisor时，它会加载所有虚拟机客户端的操作系统同时会分配给每一台虚拟机适量的内存，CPU，网络和磁盘。**



Vagrant (2010)

Written in Ruby, **Vagrant is a scripting engine for VirtualBox**. So you can setup multiple virtual machines. Vagrant gives you multiple options for provisioning (automate installation) your virtual machine, from simple shell scripts to more complex tools like Chef and Puppets



## Vagrant

* 方便建立虚拟环境，而且可以**模拟多台虚拟机**，这样我们平时还可以在开发机模拟分布式系统。
* Vagrant还会创建一些**共享文件夹**，用来给你在主机和虚拟机之间共享代码用。这样就使得我们可以**在主机上写程序，然后在虚拟机中运行**。如此一来团队之间就可以共享相同的开发环境，就不会再出现类似“只有你的环境才会出现的bug”这样的事情。
* 提升工作效率：团队新员工加入，常常会遇到花一天甚至更多时间来从头搭建完整的开发环境，而有了Vagrant，只需要直接将已经打包好的package（里面包括开发工具，代码库，配置好的服务器等）拿过来就可以工作了。**通过 Vagrant 封装一个 Linux 的开发环境，分发给团队成员。成员可以在自己喜欢的桌面系统（Mac/Windows/Linux）上开发程序，代码却能统一在封装好的环境里运行**。
* 对于自动部署，vagrant支持多种机制：可以使用puppet，chef或者用于在vagrant配置文件中定义的所有虚拟机上自动安装软件程序和配置的shell脚本等。

Install

1. VirtualBox安装
2. Vagrant安装
3. Vagrant配置
4. 下载VagrantBox
5. 添加box

$ vagrant box add base 远端的box地址或者本地的box文件名

base是box的名称，可以是任意的字符串，base是默认名称，主要用来标识一下你添加的box

box中的镜像文件被放到了：/Users/astaxie/.vagrant.d/boxes/，如果在window系统中应该是放到了： C:\Users\当前用户名\.vagrant.d\boxes\目录下。

box是一个配置好的基础虚拟机映像，Vagrant使用盒子来创建开发环境，支持VirtualBox、VMware、KVM等虚拟机。

$vagrant box list #显示当前已经添加的box列表

单机单虚拟机

1. 初始化

$ vagrant init

如果你添加的box名称不是base，那么需要在初始化的时候指定名称vagrant init haha。实质是在当前目录生成一个 Vagrantfile的文件，里面有很多配置信息

1. 启动虚拟机

$vagrant up

1. 连接到虚拟机

$vagrant ssh

这样我们就可以像连接到一台服务器一样进行操作了。

打包分发

$ vagrant package

打包完成后会在当前目录生成一个 package.box 的文件，将这个文件传给其他用户，其他用户只要添加这个 box 并用其初始化自己的开发目录就能得到一个一模一样的开发环境了。

部署自定义虚拟主机

1) 在待部署的主机上安装 Vagrant 软件环境

2) 将打包 xxx.box 映像复制到待部署主机上

3) 在待部署的主机上使用 Vagrant 添加 xxxx.box

4) 执行 vagrant up启动虚拟主机（部署完成）

$vagrant box list #显示当前已经添加的box列表

$vagrant box remove #删除相应的box

$vagrant destroy #停止当前正在运行的虚拟机并销毁所有创建的资源

$vagrant halt #关机

$vagrant package #打包命令，可以把当前的运行的虚拟机环境进行打包

$vagrant reload #重新启动虚拟机，主要用于重新载入配置文件

$vagrant resume #恢复前面被挂起的状态

$vagrant ssh-config #输出用于ssh连接的一些信息

$vagrant status #获取当前虚拟机的状态

$vagrant suspend #挂起当前的虚拟机

Vagrantfile配置文件

配置这个虚拟主机网络连接方式，端口转发，同步文件夹，以及怎么和puppet，chef结合的一个配置文件

* VM的名称

config.vm.box = "base"

* 网络设置

1. host-only(主机模式)，意思是主机和虚拟机之间的网络互访

config.vm.network :forwarded\_port, guest: 80, host: 8080

虚拟机80端口映射到物理机8080端口

1. Bridge(桥接模式)，该模式下的VM就像是局域网中的一台独立的主机，也就是说需要VM到你的路由器要IP，这样的话局域网里面其他机器就可以访问它了

桥接网络（公共网络，局域网DHCP服务器自动分配IP）

config.vm.network :public\_network

1. 私有网络：允许多个虚拟机通过主机通过网络互相通信，vagrant允许用户分配一个静态IP，然后使用私有网络设置。

config.vm.network :private\_network, ip: "192.168.50.4"

* hostname设置

config.vm.hostname = "go-app"

* 同步目录

/vagrant目录默认就是当前的开发目录，这是在虚拟机开启的时候默认挂载同步的。我们还可以通过配置来设置额外的同步目录：

默认的，vagrant将共享你的工作目录（即Vagrantfile所在的目录）到虚拟机中的/vagrant

config.vm.synced\_folder "src/", "/srv/website"

"src/"：物理机目录;"/srv/website"虚拟机目录

* vagrant和puppet

config.vm.provision :puppet #这里没有配置pp文件等的路径，全部采用默认

end

默认配置的目录结构：

Vagrantfile

manifests

default.pp

单机多虚拟机 （分布式集群系统）

* 快速建立产品网络的多机器环境，例如web服务器、db服务器
* 建立一个分布式系统，学习他们是如何交互的
* 测试API和其他组件的通信
* 容灾模拟，网络断网、机器死机、连接超时等情况

Vagrantfile

Vagrant.configure("2") do |config|

config.vm.define :web do |web|

web.vm.provider "virtualbox" do |v|

v.customize ["modifyvm", :id, "--name", "web", "--memory", "512"]

end

web.vm.box = "vagrant-ubuntu14"

web.vm.hostname = "web"

web.vm.network :private\_network, ip: "11.11.1.1"

end

config.vm.define :db do |db|

db.vm.provider "virtualbox" do |v|

v.customize ["modifyvm", :id, "--name", "db", "--memory", "512"]

end

db.vm.box = "vagrant-ubuntu14"

db.vm.hostname = "db"

db.vm.network :private\_network, ip: "11.11.1.2"

end

end

$vagrant up

vagrant up启动后，stdout会打印ip and 端口号

登录虚机

双击Putty.exe

登录虚机web 127.0.0.1 2222

登录虚机db 127.0.0.1 2200

虚拟机之间通信

vagrant@web:~$ ssh 11.11.1.2

vagrant@db:~$

通常情况下Box只做最基本的设置，而不是设置好所有的环境，因此Vagrant通常使用Chef或者Puppet来做进一步的环境搭建

Puppet

Puppet 是一种 Linux、Unix、Windows 平台的集中配置管理系统，使用自有的 Puppet 描述语言，可管理配置文件、用户、cron任务、软件包、系统服务等。Puppet把这些系统实体称之为资源，其设计目标是简化对这些资源的管理以及妥善处理资源之间的依赖关系。

* **将基础设施作为代码进行保存与版本化.**
* 无论是在笔记本上的开发环境，还是在生产环境上，开发人员和运维人员都能够使用相同的清单对系统进行管理
* 将配置作为代码处理，系统管理员就能够为开发人员提供独占的测试环境
* 甚至可以将Puppet代码交付给审记，如今有许多审记都接收Puppet清单，以进行一致性验证
* 能够将Puppet代码签入到某个共享的版本控制工具中，这将为你的基础设施提供一个可控的历史记录。

若为vagrant设置代理

export http\_proxy="http://user:password@host:port"

vagrant plugin install vagrant-proxyconf

export VAGRANT\_HTTP\_PROXY="http://user:password@host:port"

vagrant up

VAGRANT\_HTTP\_PROXY设置可以由Vagrantfile里的proxy替代

Vagrantfile文件

Vagrant.configure("2") do |config|

config.proxy.http = "http://165.225.96.34:10015"

config.proxy.https = "http://165.225.96.34:10015"

config.proxy.no\_proxy = "localhost,127.0.0.1"

config.vm.provision :puppet

config.vm.network :public\_network

config.vm.network :forwarded\_port, guest:80, host:8080

config.vm.define :web do |web|

web.vm.provider "virtualbox" do |v|

v.customize ["modifyvm", :id, "--name", "web", "--memory", "512"]

end

web.vm.box = "vagrant-ubuntu14"

web.vm.hostname = "web"

web.vm.network :private\_network, ip: "11.11.1.1"

end

end

Vagrantfile

manifests

default.pp

Puppet清单

# Basic Puppet Apache manifest

class apache {

exec { 'apt-get update':

command => '/usr/bin/apt-get update'

}

package { "apache2":

ensure => present,

}

service { "apache2":

ensure => running,

require => Package["apache2"],

}

file { '/var/www':

ensure => link,

target => "/vagrant",

notify => Service['apache2'],

force => true

}

}

include apache

vagrant@web:~$ ls /vagrant

$ls /var/www

以上都是显示的是Vagrantfile文件父目录的子目录和文件

已经安装好ubunbu and apache2, apache2的静态站点根目录是/var/www/html

所以本地开发目录是

vagrant

Vagranfile

manifests

defaults.pp

html

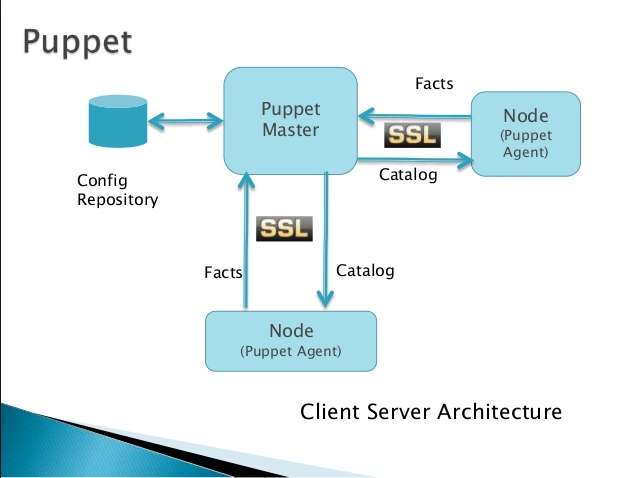
index.html

登录虚拟机$curl 127.0.0.1

或者host机浏览器: localhost:8080 （因为puppet已将虚拟机端口转发）

主机-代理部署

1. 主机puppet master集中存储所有清单，通过/etc/puppet/manifests/site.pp找到条目确定每个代理执行的Puppet代码
2. 主机将清单编译为编目并将其传输给代理执行
3. 代理向主机报告Puppet运行状态



实战

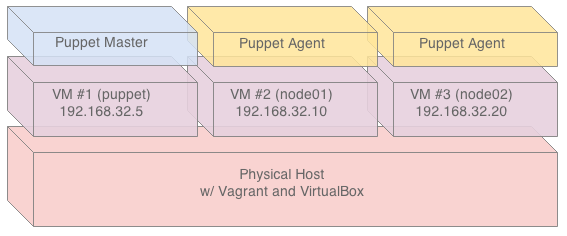
Refer:

<http://gold.xitu.io/entry/569d9cff816dfa005958feb5>

<https://programmaticponderings.wordpress.com/2014/12/14/installing-puppet-master-and-agents-on-multiple-vm-using-vagrant-and-virtualbox/>

<https://github.com/garystafford/multi-vagrant-puppet-vms>

Installing Puppet Master and Agents on Multiple VM Using Vagrant and VirtualBox



one server VM running Puppet Master, and two agent node VMs running Puppet Agent. Both agent nodes should have successfully been registered with Puppet Master, and configured themselves based on the Puppet Master’s main manifest. Agent node configuration includes installing ntp, git, Fig, and Docker.

multi-vagrant-puppet-vms

Vagrantfile

nodes.json

bootstrap-master.sh

bootstrap-node.sh

site.pp

创建三个虚拟机，以及虚拟机的安装脚本

nodes.json

{

"nodes": {

"puppet.example.com": {

":ip": "192.168.32.5",

"ports": [],

":memory": 1024,

":bootstrap": "bootstrap-master.sh"

},

"node01.example.com": {

":ip": "192.168.32.10",

"ports": [],

":memory": 1024,

":bootstrap": "bootstrap-node.sh"

},

"node02.example.com": {

":ip": "192.168.32.20",

"ports": [],

":memory": 1024,

":bootstrap": "bootstrap-node.sh"

}

}

}

Vagrantfile

# read vm and chef configurations from JSON files

nodes\_config = (JSON.parse(File.read("nodes.json")))['nodes']

VAGRANTFILE\_API\_VERSION = "2"

Vagrant.configure(VAGRANTFILE\_API\_VERSION) do |config|

config.proxy.http = "http://165.225.96.34:10015"

config.proxy.https = "http://165.225.96.34:10015"

config.proxy.no\_proxy = "localhost,127.0.0.1"

config.vm.box = "ubuntu/trusty64"

nodes\_config.each do |node|

node\_name = node[0] # name of node

node\_values = node[1] # content of node

config.vm.define node\_name do |config|

# configures all forwarding ports in JSON array

ports = node\_values['ports']

ports.each do |port|

config.vm.network :forwarded\_port,

host: port[':host'],

guest: port[':guest'],

id: port[':id']

end

config.vm.hostname = node\_name

config.vm.network :private\_network, ip: node\_values[':ip']

config.vm.provider :virtualbox do |vb|

vb.customize ["modifyvm", :id, "--memory", node\_values[':memory']]

vb.customize ["modifyvm", :id, "--name", node\_name]

end

config.vm.provision :shell, :path => node\_values[':bootstrap']

end

end

end

Bootstrapping Puppet Master Server (搭建Puppet服务器)

# Install Puppet Master （安装puppetmaster）

sudo apt-get install -yq puppetmaster

# Configure /etc/hosts file

echo "192.168.32.5 puppet.example.com puppet" | sudo tee --append /etc/hosts

echo "192.168.32.10 node01.example.com node01" | sudo tee --append /etc/hosts

echo "192.168.32.20 node02.example.com node02" | sudo tee --append /etc/hosts

# Add optional alternate DNS names to /etc/puppet/puppet.conf

sudo sed -i 's/.\*\[main\].\*/&\ndns\_alt\_names = puppet,puppet.example.com/' /etc/puppet/puppet.conf

# symlink manifest from Vagrant synced folder location

ln -s /vagrant/site.pp /etc/puppet/manifests/site.pp

Bootstrapping Puppet Agent Nodes （搭建Puppet客户端）

sudo apt-get install -yq puppet

#向Puppet master进程发起一个连接，这个进程会监听puppet server的8140端口

sudo puppet resource cron puppet-agent ensure=present user=root minute=30 \ command='/usr/bin/puppet agent --onetime --no-daemonize --splay'

sudo puppet resource service puppet ensure=running enable=true

# Configure /etc/hosts file

echo "192.168.32.5 puppet.example.com puppet" | sudo tee --append /etc/hosts

echo "192.168.32.10 node01.example.com node01" | sudo tee --append /etc/hosts

echo "192.168.32.20 node02.example.com node02" | sudo tee --append /etc/hosts

# Add agent section to /etc/puppet/puppet.conf

#编辑文件，告诉puppet代理puppet服务器的域名

echo "" && echo "[agent]\nserver=puppet" | sudo tee --append /etc/puppet/puppet.conf

sudo puppet agent --enable

登录puppet client 1: node01

登录puppet client 2: node02

//客户端创建的SSL认证请求，并将其发送给服务器, 同理node01

vagrant@node02:~$ sudo puppet agent –test –waitforcert=60

登录puppet master

//查看还未被签署的认证请求列表。

vagrant@puppet:~$sudo puppet cert –list

//签署请求

$ sudo puppet cert --sign puppetclient

Once the certificate signing process is complete, the Puppet Agent retrieves the client configuration from the Puppet Master and applies it to the local agent node. The Puppet Agent will execute all applicable steps in the site.pp manifest on the Puppet Master server, designated for that specific Puppet Agent node

site.pp manifest on the Puppet Master server, applied by Puppet Agent on the agent nodes.

site.pp

node default {

notify { "Debug output on ${hostname} node.": }

include ntp, git

}

node 'node01.example.com', 'node02.example.com' {

notify { "Debug output on ${hostname} node.": }

include ntp, git, docker, fig

}

Docker

The difference between **Docker and Vagrant**

Once difference is that Docker is light-weight and shared-kernel Linux containers between all instances but user spaces are different for each instance.

While Vagrant is automatically provisioning multiple virtual machines with their own configurations; manage VMs can be created and dispose on demand via commands; Moreover, Vagrant can be a Docker provider to manager Docker builds and deployments.

Vagrant

## Docker

Books:

《第一本Docker书》

《第一本Docker书>修订版》

### Containerize app

* Install Docker

install virtualBox and Ubuntu

refer to: <https://docs.docker.com/linux/step_one/>

ctrl + alt + t打开终端

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

若出现is Docker Daemon running? 将qzlin加入docker用户组 and Require reboot Ubuntu

sudo groupadd docker

sudo gpasswd -a ${USER} docker

sudo service docker restart

newgrp docker or log out/in to activate the changes to groups.

$sudo service docker restart //restart the docker service

**Containerize an application**

a container: 1) is a runnable instance of an image. You can create, start, stop, move, or delete a container using the DockerAPI or CLI. 2) Can be run on local machines, virtual machines or deployed to the cloud. 3) Is portable (can be run on any OS). 4) Is isolated from other containers and runs its own software, binaries, and configurations.

app

spec

src

package.json

yarn.lock

DockerFile // create Dockfile

Dockerfile

FROM node:18-alpine //base image

WORKDIR /app

COPY . .

RUN yarn install --production

CMD ["node", "src/index.js"]

EXPOSE 3000

$docker build -t todo-app . //构建image `getting-started`

Note: 若无法connection，添加--build-arg http\_proxy=<http://165.225.96.34:10015/>

$docker run **-dp 127.0.0.1:3000:3000** todo-app // Start an app container

-p 将容器端口3000发布到本机127.0.0.1:3000

浏览器访问<http://localhost:3000>

**Update the application and container image**

src/static/js/app.js // Update the source code

docker build -t todo-app . //update image

docker rm -f <the-container-id> // stop and remove old container

docker run -dp 127.0.0.1:3000:3000 todo-app //start updated app container

local development setups

$ docker run -dp 127.0.0.1:3000:3000 \

-w /app \ // 设置工作目录（working directory）

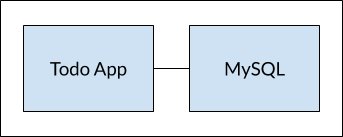
--mount type=bind,src="$(pwd)",target=/app \ //同步工作目录与容器目录/app

node:18-alpine sh -c "yarn install && yarn run dev"

修改当前目录，比如src/static/js/app.js，则同步容器目录/app

刷新网页，更新结果 （容器内nodemon process会监视文件变化，重启应用）

**App connect to MySQL (容器之间通过网络通信)**



* Docker Networking可以将容器连接到不同宿主机上的容器
* 通过Docker Networking连接的容器可以在无需更新连接的情况下，对停止、启动或者重启容器。
* 使用Docker Networking，不必事先创建容器再去连接它。同样，也不必关心容器的运行顺序

创建网络

$ docker network create todo-app //创建桥接网络，命名为todo-app

docker run -d \

--network todo-app --network-alias mysql \ //在Docker网络todo-app中启动

-v todo-mysql-data:/var/lib/mysql \

-e MYSQL\_ROOT\_PASSWORD=secret \

-e MYSQL\_DATABASE=todos \

mysql:8.0

docker run -dp 127.0.0.1:3000:3000 \

-w /app -v "$(pwd):/app" \

--network todo-app \ //在Docker网络todo-app中启动

-e MYSQL\_HOST=mysql \

-e MYSQL\_USER=root \

-e MYSQL\_PASSWORD=secret \

-e MYSQL\_DB=todos \

node:18-alpine \

sh -c "yarn install && yarn run dev"

**Docker Compose (for multi-container apps with YAML文件)**

docker-compose.yml

services:

app: // Define the app service

image: node:18-alpine

command: sh -c "yarn install && yarn run dev"

ports:

- 127.0.0.1:3000:3000

working\_dir: /app

volumes:

- ./:/app

environment:

MYSQL\_HOST: mysql

MYSQL\_USER: root

MYSQL\_PASSWORD: secret

MYSQL\_DB: todos

mysql: // Define the MySQL service

image: mysql:8.0

volumes:

- todo-mysql-data:/var/lib/mysql

environment:

MYSQL\_ROOT\_PASSWORD: secret

MYSQL\_DATABASE: todos

volumes:

todo-mysql-data:

$ docker compose up -d // Run the application stack

### Dockerfile指令

RUN //指定镜像被构建时要运行的命令

CMD //指定容器被启动时要运行的命令

举例：

$docker run -it qizhonglin/static\_web:v0 /bin/true //等价于CMD [“/bin/true”]

docker run命令可以覆盖CMD指令。如果我们在Dockerfile里指定了CMD指令，而同时在docker run命令行中也指定了要运行的命令，命令行中指定的命令会覆盖Dockerfile中的CMD指令。

在Dockerfile中只能指定一条CMD指令。如果指定了多条CMD指令，也只最后一条CMD指令会被使用。

ENTRYPOINT //与CMD指令类似，但不容易在启动容器时被覆盖。实际上docker run命令行中指定的任何参数都会被当做参数再次传递给ENTRYPOINT指令中指定的命令。

WORKDIR //设置工作目录，ENTRYPOINT和CMD指定程序会在此目录下执行

ENV //设置环境变量

USER //指定该镜像会以什么样的用户去运行

VOLUME //添加卷。卷可以是容器内目录，提供共享数据或数据持久化

* 卷可以在容器间共享和重用，一个容器可以不是必须和其他容器共享卷
* 对卷的修改是立时生效的，对卷的修改不会对更新镜像产生影响
* 卷会一直存在直至没有任何容器再使用它

卷功能让我们可以**将数据（源代码）、数据库或者其他内容添加到镜像中而不是将这些内容提交到镜像中**，并且允许我们在多个容器间共享这些内容。我们可以利用此功能来**测试容器和内容的应用程序代码**，管理日志，或者处理容器内部的数据库

ADD //将构建环境下的文件和目录复制到镜像中.

COPY //构建环境下的文件和目录复制到镜像中，但不会做文件提取和解压工作

注意：复制时，文件源路径必须是一个与当前构建环境相对的文件或者目录，本地文件都放到和Dockerfile同一个目录下，不能复制该目录之外的任何文件。

ONBUILD //为镜像添加触发器。当一个镜像被用做其他镜像的基础镜像时，该镜像中的触发器将会被执行。触发器会在构建过程中插入新指令，我们可以认为这些指令是紧跟在FROM之后指定的。触发器可以是任何构建指令

### Docker命令

$docker info //返回容器和镜像数量，执行驱动和存储驱动，基本配置

docker ps -a //列出现有全部容器 -a表示所有容器包括运行和停止

$docker start bob\_the\_container //启动容器

$docker attach bob\_the\_container //重新附着到该容器的会话上

$docker logs -ft daemon\_dave // 获取容器的日志, -f监控Docker的日志

$docker top daemon\_dave // 查看容器内部运行的进程

$docker rm daemon\_dave // 删除容器

$docker rm `docker ps -a -q` // 删除所有容器

$docker rmi - f bob\_the\_container

$docker inspect daemon\_dave //返回配置信息，包括名称命令网络配置等

$docker images //列出镜像列表，本地镜像都保存在/var/lib/docker

$docker search puppet //查找镜像, 远程镜像保存在仓库Registr即Docker Hub

// Tag & push your image to DockerHub

$ docker tag imageID qizhonglin/docker-whale:latest

$ docker login - - username=qizhonglin - - [email=qizhong.lin@philips.com](mailto:email%3Dqizhong.lin@philips.com)

$ docker push qizhonglin/docker-whale

$docker pull 用户名/镜像名:tag

拉取顶层仓库或用户仓库中的所有内容，用户仓库镜像是由Docker用户创建，顶层仓库是由Docker公司和选定的厂商创建

$docker run -it --name next\_container Ubuntu /bin/bash

docker run命令从镜像启动一个容器时，如果该镜像不在本地，Docker会先从Docker Hub下载该镜像。如果没有指定具体的镜像标签，那么Docker会自动下载latest标签的镜像。

// remove all Exited containers

sudo docker ps -a | grep Exit | cut -d ' ' -f 1 | xargs sudo docker rm

$docker run - - name bob\_the\_container -it ubuntu /bin/bash //运行

--name: 指定容器名称(替代容器ID), 命名唯一。

- i 标志保证容器中STDIN是开启的

- t 标志告诉Docker为要创建的容器分配一个伪tty终端

ubuntu镜像告诉Docker基于ubuntu镜像来创建容器，该容器拥有自己的网络、IP地址、以及一个用来和宿主机进行通信的桥接网络接口

/bin/bash告诉Docker在新容器中运行的命令，在这里指启动一个bash shell

[root@de28...:/#](mailto:root@de28...:/) apt-get update && apt-get install vim

若无法连接网络，方法一：见官方<https://docs.docker.com/config/daemon/systemd/>

方法二： $sudo gedit /etc/default/docker

export http\_proxy=”http://HOST:PORT/”

方法三：

$ export http\_proxy=http://165.225.96.34:10015/

export https\_proxy=https://165.225.96.34:10015/

$docker run - - name daemon\_dave **-d** ubuntu /bin/sh \

-c “while true; do echo hello world; sleep 1; done”

除了交互式运行容器(interactive container),我们也可以创建长期运行的容器（守护式容器daemonized container）没有交互式会话，非常适合运行应用程序和服务。**-d标识告诉Docker将容器入到后台运行**

$docker exec <container-id> cat /data.txt // 进入正在运行中的后台容器

$docker run **-dp 127.0.0.1:3000:3000** getting-started // Start an app container

-p 将容器端口3000发布到本机127.0.0.1:3000

Docker容器公开端口 -> 本地网络

Docker内部网络: 在安装Docker时，会创建一个新的网络接口，名字是docker0.每个容器都会在这个接口上分配一个IP地址，如$ip a show docker0

接口docker0是一个虚拟的以太网桥，用于连接容器和本地宿主网络

Docker每创建一个容器就会创建一组互联的网络接口。这组接口就像管道的两端。这组接口其中一端作为容器的eth0接口，而另一端统一命令为类似vethec6a这种名字，作为宿主机的一个端口。

容器默认是无法访问的，从宿主网络与容器通信时，必须明确指定打开的端口

$docker inspectgetting-started //查看网关地址docker0, IP是”IPAddress”, 端口是”Ports”

$docker port webapp 4567 //获取映射端口，假定为49161

$docker network ls //命令列出当前系统中的所有网络

$docker network rm app //删除一个Docker网络

$docker network inspect app //查看网络更详细的信息

$docker net connect app db2 //将正在运行的容器db2加入docker网络app中

$docker network disconnect app db2 //容器db2断开docker网络app

$docker run - -restart=always -name daemon\_dave –d Ubuntu /bin/sh –c “while true;…”

restart=always标志表明无论容器的退出代码是什么，Docker都会自动重启该容器

restart=on-failure标志表明只有当容器的退出代码为非0值的时候，才会自动重启

$docker-compose up -d //启动多容器

$docker-compose ps //列出docker-compose.yml正在运行的所有服务

$docker-compose logs //显示Docker Compose服务的日志

$docker-compose stop

$docker-compose start

$docker-compose kill

$docker-compose rm

### Docker镜像和仓库

Docker镜像是由文件系统叠加而成，最底端是一个引导文件系统bootfs,第二层是root文件系统rootfs,它位于引导文件系统之上，rootfs可以是一种或多种操作系统如Debian or Ubuntu。

Docker将这样的文件系统称为镜像。一个镜像可以放到另一个镜像的顶部。位于下面的镜像称为父镜像，可以依次类推，直至镜像栈的最底部，最底部的镜像称为基础镜像。最后，当从一个镜像启动容器时，Docker会在该镜像的最顶层加载一个读写文件系统。我们想在Docker中运行的程序就是在这个读写层中执行（见page: 50）

当Docker第一次启动一个容器时，初始的读写层是空的。当文件系统发生变化时，这些变化都会应用到这一层上。比如如果想修改一个文件，这个文件首先会从该读写层下面的只读层复制到该读写层。该文件的只读版本依然存在，但是已经被读写层中的该文件副本所隐藏。

这种机制称为写时复制copy on write.每个只读镜像层都是只读的，并且以后永远不会变化。当创建一个新容器时，Docker会构建出一个镜像栈，并在栈的最顶端添加一个读写。这个读写层再加上其下面的镜像层以及一些配置数据，就构成了一个容器。

**发布镜像to Docker registry (相当于Share application)**

构建镜像：一般基于一个已有的基础镜像，如ubuntu,构建新镜像

Dockerfile使用基本的基于DSL语法的指令来构建一个Docker镜像

目录结构

static\_web

Dockerfile

static\_web目录就是我们的构建环镜build environment, Docker则称此环境为上下文context or 构建上下文。Docker会在构建镜像时将构建上下文和该上下文中的文件和目录上传到Docker守护进程。这样Docker守护进程就能直接访问你想在镜像中存储的任何代码、文件或者其他数据

Dockerfile

# Version: 0.0.1

FROM ubuntu:14.04 //基本镜像

MAINTAINER QiZhong Lin [qizhong.lin@philips.com](mailto:qizhong.lin@philips.com) //镜像作者

RUN apt-get update //镜像中运行命令(默认情况下在shell里使用/bin/sh -c执行)

RUN apt-get install –y nginx

RUN echo ‘Hi, I am in your container’ > /usr/share/nginx/html/index.html

EXPOSE 80 //容器内应用程序使用容器端口。

#开头的行都会被认为是注释

Docker中的指令会按顺序从上而下执行，每条指令都会创建一个新的镜像层并对镜像进行提交。Docker大体上按照如下流程执行Dockerfile中的指令

1. Docker从基础镜像运行一个容器
2. 执行一条指令，对容器做出修改
3. 执行类似docker commit的操作，提交一个新的镜像层
4. Docker再基于刚提交的镜像运行一个新容器
5. 执行Dockerfile中的下一条指令，直至所有指令都执行完毕

如果你的Dockerfile由于某些原因如某条指令失败了没有正常结束，那么你将得到一个可以使用的镜像。这对调试非常有帮助

$docker build -t=”qizhonglin/static\_web:v0” . //构建镜像， -t设置仓库和名称

最后的`.`告诉Docker到本地目录中去找Dockerfile文件。也可以指定一个Git仓库的源地址来指定Dockerfile的位置。

如果在构建上下文的根目录下存在以.dockerignore命名的文件的话，那么该文件内容会被按行进行分割，每一行都是一条文件过滤匹配模式。非常像.gitignore文件。该文件用来设置哪些文件不会被上传到构建上下文中去。

Docker会将之前构建时创建的镜像当做缓存并作为新的开始点。

$docker history qizhonglin/static\_web:v0 // 查看镜像是如何构建出来

$docker run -d -p 80 --name static\_web qizhonglin/static\_web:v0 \

nginx -g “daemon off;”

启动容器qizhonglin/static\_web:v0

-d选项告诉Docker以分离detached的方式在后台运行。这种方式非常适合运行类似nginx守护进程这样的需要长时间运行的过程。

-p标志用来控制Docker在运行时应该公开哪些网络端口给外部（宿主机）

nginx -g “daemon off;” 容器里运行的命令，表明将以前台运行的方式启动nginx作来web服务器

$docker ps -l // 查看容器中的80端口被映射到宿主机的随机端口号

$docker run -d -p 8080:80 --name static\_web qizhonglin/static\_web:v0 nginx –g “daemon off;”

将容器80端口映射到宿主机的指定端口8080

$curl localhost:映射端口号

$docker rmi qizhonglin/static\_web:v0 //删除一个镜像

该操作只会删除本地的镜像，如果想删除Docker Hub上的镜像仓库，DockerHub -> Delete repository

$docker rmi `docker images –a –q`

运行自己的Docker Registry，见page90

Or push to Docker Hub 见docker网站

### 范例：测试静态网站，web服务，

**范例1：使用Docker测试静态网站**

将Docker作为本地Web开发环境，这个环境可以完全重现生产环境，保证开发环境和部署环境一致

sample

Dockerfile

nginx

global.conf

nginx.conf

website

index.html

$docker build -t qizhonglin/nginx .

$docker run -dp 80 --name website -v $PWD/website:/var/www/html/website qizhonglin/nginx nginx

-v这个选项允许我们将宿主机的目录$PWD/website作卷，挂载到容器里/var/www/html/website, 而容器会将该目录/var/www/html/website作为nginx服务器的根目录root，从而挂载了正在开发的本地网站

卷是在一个或者多个容器内被选定的目录，可以绕过分层的联合文件系统，为Docker提供持久数据或者共享数据。这意味着对卷的修改会直接生效，并绕过镜像。当提交或者创建镜像时，卷不被包含在镜像里

$docker ps -l //容器website正在运行，其80端口被映射到本地…端口port

$curl localhost:port //访问容器里运行的服务

修改本地网站内容

$curl localhost:port //刷新镜像容器里挂载该网站，自动更新

**范例2：构建Redis镜像和容器**

Dockerfile

FROM ubuntu

RUN apt-get update

RUN apt-get -y install redis-server redis-tools

EXPOSE 6379

ENTRYPOINT ["/usr/bin/redis-server"]

CMD []

$docker build -t qizhonglin/redis .

$docker run -d -p 6379 --name redis qizhonglin/redis

$docker port redis 6379 //查看redis端口6379映射到host端口是多少(假定是49161)？

Host机安装Redis客户端，连接容器redis, 验证Redis服务器是否工作

$apt-get -y install redis-tools

$redis-cli -h 127.0.0.1 -p 49161

Redis 127.0.0.1:49161>

**范例3：Docker容器互连**

$docker run -d --name redis qizhonglin/redis 启动redis容器

$docker run -p 4567 --name webapp --link redis:db -it -v $PWD/webapp:/opt/webapp qizhonglin/Sinatra /bin/bash

启动webapp容器，对外暴露4567端口，内连redis容器

--link标志创建了两个容器间的父子连接，连接的容器名字:连接后容器的别名

通过把容器连接在一起，可以让父容器直接访问任意子容器的公开端口（比如，父容器webapp可以连接到子容器redis的6379端口）。容器化的应用程序限制了可被攻击的界面，减少了公开暴露的网络

被连接的容器必须运行在同一个Docker宿主机上，不同Docker宿主机上运行的容器无法连接

Docker在父容器中的以下两个地方写入了连接信息

/etc/hosts文件中

包含连接信息的环境变量中

$docker run -p 4567 --name webapp ---link redis:db …

[root@...:/#](mailto:root@...:/) env

HOSTNAME=…

DB\_NAME=…

DB\_PORT=…

Docker在连接webapp and redis容器时，自动创建了这些以DB开头的环境变量，这些连接信息可以让容器内的应用程序使用相同的方法与别的容器进行连接，而不用关心被连接的容器的具体细节

可以给Sinatra应用程序加入一些连接信息，以便与Redis通信。

require ‘uri’

…

uri = URI.parse(ENV[‘DB\_PORT’])

redis = Redis.new(:host => uri.host, :port -> uri.port)

…

**范例4：使用Docker构建服务**

Jekyll镜像：容器存放通过卷挂载的网站源代码

Apache镜像：容器利用包含编译后的网站的卷，并为其服务

Jekyll

Dockerfile

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install ruby ruby-dev make nodejs

RUN gem install --no-rdoc --no-ri jekyll

VOLUMN /data/

VOLUMN /var/www/html

WORKDIR /data

ENTRYPOINT ["jekyll", "build", "--destination=/var/www/html"]

/data/：用来存放网站的源代码

/var/www/html: 用来存放编译后的jekyll网站码

ENTRYPOINT: 自动构建，将工作目录/data/中的所有jekyll网站代码构建到/var/www/html/目录中

Apache镜像

apache

Dockerfile

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install apache2

VOLUMN ["/var/www/html"]

WORKDIR /var/www/html

EXPOSE 80

ENTRYPOINT ["/usr/sbin/apache2"]

CMD ["-D", "FOREGRAOUND"]

使用VOLUMN指令创建一个卷/var/www/html，用来存放编译后的jekyll网站，然后将/var/www/html设为工作目录

使用ENTRYPOINT and CMD指令组合在容器启动时默认运行Apache

$docker build -t qizhonglin/Jekyll .

$docker build -t qizhonglin/apache .

$docker run -v /home/qizhonglin/blog:/data/ --name blog qizhonglin/Jekyll

把本地的/home/qizhonglin/blog目录作为/data/卷挂载到容器里。容器已经拿到网站的源代码，并将其构建到已编译的网站，存放到/var/www/html/目录

卷是在一个或多个容器中特殊指定的目录，卷会绕过联合文件系统，为持久化数据和数据提供几个有用的特性。

* 卷可以在容器间共享和重用
* 共享卷时不一定要运行相应的容器
* 对卷的修改会直接在卷上反映出来
* 更新镜像时不会包含对卷的修改
* 卷会一直存在，直至没有容器使用它们

利用卷，可以在不用提交镜像修改的情况下，向镜像里加入数据（如源代码、数据或者其他内容），并且可以在容器间共享这些数据

$docker run -d -P -volumns-from blog qizhonglin/apache

--volumns-from标志把指定容器里的所有卷都加入新创建的容器里。这意味着Apache容器可以访问之前创建的blog容器里/var/www/html卷中存放的编译后的jekyll网站。即使blog容器没有运行，apache容器也可以访问这个卷。

$docker port containerID 80

假定为49160

localhost:49160

**范例5: 使用Docker构建并测试Web应用程序**

sinatra

Dockerfile

webapp

bin

webapp

lib

app.rb

webapp\_redis

bin

webapp

lib

app.rb

redis

Dockerfile

Dockerfile

FROM Ubuntu:14.04

RUN apt-get update -yqq && apt-get -yqq install ruby ruby-dev build-essential redis-tools

RUN gem install --no-rdoc --no-ri Sinatra json redis

RUN mkdir -p /opt/webapp

EXPOSE 4567

CMD [“/opt/webapp/bin/webapp” ]

$docker build -t qizhonglin/sinatra .

启动服务

$docker run -d -p 4567 --name webapp -v $PWD/webapp:/opt/webapp qizhonglin/sinatra

将本地的webapp目录挂载到容器里创建的目录/opt/webapp

容器启动时，执行/opt/webapp/bin/webapp这个文件

$docker port webapp 4567 (假定映射端口是49160)

测试

$curl -i -H ‘Accept:application/json’ -d ‘name=Foo&status=Bar’ <http://localhost:49160/json>

若web应用程序使用Redis后端数据库，则在应用程序文件app.rb里创建到Redis连接redis=Redis.new(:host=>’db’, :port=>’6379’)

构建启动Redis容器

Redis

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install redis-server redis-tools

EXPOSE 6379

ENTRYPOINT [ “/usr/bin/redis-server” ]

CMD []

$docker build -t qizhonglin/redis .

$docker run -d -p 6379 --name redis qizhonglin/redis

$docker port redis 6379 （假定映射端口是49161）

在本地安装redis客户端，访问redis容器这个服务端

$apt-get -y install redis-tools

$redis-cli -h 127.0.0.1 -p 49161

绑定sinatra容器与redis容器

从Docker1.9版本开始，容器之间可以通过Docker Networking创建的网络互相通信。

* Docker Networking可以将容器连接到不同宿主机上的容器
* 通过Docker Networking连接的容器可以在无需更新连接的情况下，对停止、启动或者重启容器。
* 使用Docker Networking，不必事先创建容器再去连接它。同样，也不必关心容器的运行顺序

创建网络

$**docker network create** app //创建桥接网络，命名为app

在Docker网络app中启动Redis容器

$docker run -d **--net=app** --name db qizhonglin/redis //--net指定db在网络app中运行

在Docker网络app中启动sinatra容器

$docker run -p 4567 --net=app --name webapp\_redis -it -v $PWD/webapp\_redis:/opt/webapp qizhonglin/sinatra

**在同一个Docker网络app启动的容器，会将它们的IP地址信息自动更新在/etc/hosts文件中。通过主机名和端口可以互相访问。比如webapp访问数据库redis,可以通过db:6379**

redis = Redis.new(:host=> ‘db’, :port=> ‘6379’)

**范例6：使用Docker构建一个Java应用服务**

fetcher

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install wget

VOLUME [ “/var/lib/tomcat7/webapps/” ]

WORKDIR /var/lib/tomcat7/webapps/

ENTRYPOINT [ “wget”]

CMD [ “-?” ]

使用wget从指定的URL获取文件，并把文件保存在/var/lib/tomcat7/webapps/目录。（之前把这个卷共享给Tomcat服务器并且运行里面的内容

若没有指定URL, 指令ENTRYPOINT and CMD会返回wget帮助

$docker build -t qizhonglin/fetcher .

$docker run -I -t --name sample\_war qizhonglin/fetcher <https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war>

下载sample.war使存于共享卷中

tomcat7

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install tomcat7 default-jdk

ENV CATALINA\_HOME /usr/share/tomcat7

ENV CATALINA\_BASE /var/lib/tomcat7

ENV CATALINA\_PID /var/run/tomcat7.pid

ENV CATALINA\_SH /usr/share/tomcat7/bin/catalina.sh

ENV CATALINA\_TMPDIR /tmp/tomcat7-tomcat7-tmp

RUN mkdir -p $CATALINA\_TMPDIR

VOLUME [ “/var/lib/tomcat7/webapps/” ]

EXPOSE 8080

ENTRYPOINT [ “/usr/share/tomcat7/bin/catalina.sh”, “run” ]

安装jdk and tomcat

创建应用程序的共享卷

启动tomcat

$docker build -t qizhonglin/tomcat7 .

$docker run --name sample\_app --volumes-from sample\_war -d -P qizhonglin/tomcat7

使容器sample\_app与sample\_war共享卷，（即存储在/var/lib/tomcat7/webapps/卷里的WAR文件会从sample容器挂载到sample\_app容器，最终被Tomcat加载并执行。

$docker port sample\_app 8080 (假定映射端口是49154)

localhost:49154/sample

**范例7：多容器的应用栈**

Express + redis

Node容器 + 主Redis/2从Redis + 日志容器

nodejs

nodeapp

package.json

server.js

Dockerfile

redis\_base

Dockerfile

redis\_primary

Dockerfile

redis\_replica

Dockerfile

logstash

Dockerfile

主从Redis服务

redis\_base

FROM ubuntu:14.04

RUN apt-get -yqq update && apt-get -yqq install redis-server redis-tools

VOLUME [ "/var/lib/redis", "/var/log/redis/" ]

EXPOSE 6379

CMD []

$docker build -t qizhonglin/redis\_base .

redis\_primary

FROM qizhonglin/redis\_base

ENTRYPOINT [ "redis-server", "--logfile /var/log/redis/redis-server.log" ]

$docker build -t qizhonglin/redis\_primary

redis\_replica

FROM qizhonglin/redis\_base

ENTRYPOINT [ "redis-server", "--logfile /var/log/redis/redis-replica.log", "--slaveof redisprimary 6379" ]

$docker build -t qizhonglin/redis\_replica

启动Redis主从服务器

由镜像qizhonglin/redis\_primary and qizhonglin/redis\_replica运行主从redis服务

$docker network create express

$docker run -d -h redisprimary --net express --name redisprimary qizhonglin/redis\_primary

$docker run -d -h redispreplica1 --net express --name redisreplica1 qizhonglin/redis\_replica

$docker run -d -h redispreplica2 --net express --name redisreplica2 qizhonglin/redis\_replica

查看redis容器的日志

$docker run -it --rm --volumes-from redisprimary ubuntu cat /var/log/redis/redis-server.log

启动应用程序

$docker build -t qizhonglin/nodejs .

$docker run -d **--net express** -p 3000:3000 --name nodeapp qizhonglin/nodejs

host机测试

localhost:3000

同一网络express内的虚机可以互通

$docker inspect redisprimary （假定IP地址: 172.18.0.2）

$docker exec -it nodeapp bash

[root@...:/opt/nodeapp#](mailto:root@...:/opt/nodeapp) ping 172.18.0.2

虚拟机open tcp 3000,从而可以在window甚至外网连接

捕获应用日志

在生产环境里需要确保可以捕获日志并将日志保存到日志服务器。一般使用Logstash来完成

Dockerfile

FROM Ubuntu:14.04

RUN apt-get -yqq install logstash

ADD logstash.conf /etc/

WORKDIR /opt/logstash

ENTRYPOINT [ “bin/logstash” ]

CMD [ “--config=/etc/logstash.conf” ]

Logstash配置文件logstash.conf

input {

file {

type => "syslog"

path => ["/var/log/nodeapp/nodeapp.log", "/var/log/redis/redis-server.log"]

}

}

output {

stdout {

codec => rubydebug

}

}

input: Logstash会监视这两个文件，将其中新的内容发送给Logstash.

output: 接受所有Logstash输入的内容将其输出到标准输出上。现实中，一般会将Logstash配置到Elasticsearch集群或者其他的目的。

$docker build -t qizhonglin/logstash .

$docker run -d --name logstash --volumes-from redisprimary --volumes-from nodeapp qizhonglin/logstash

分别挂载了redisprimary and nodeapp容器的卷，这样就可以访问Redis and Node的日志文件了，任何加到这些日志文件里的内容都会反映在logstash容器的卷里，即input.file.path, logstash容器启动时由命令bin/logstash --config=/etc/logstash.conf 启动Logstash并加载/etc/logstash.conf配置文件

$docker logs -f logstash

查看logstash容器的日志（因此Logstash配置文件将输出写至stdout）

### 实战：java程序，Spring Boot,

**镜像Jdk8**

jdk8-image-ex

Dockerfile

docker-compose.yml

Main.java

**Dockerfile**

FROM frolvlad/alpine-oraclejdk8:slim

WORKDIR /mnt

ADD Main.java .

ENTRYPOINT [ "sh" ]

CMD [ "-c", "javac Main.java && java Main" ]

**docker-compose.yml**

jdk8-ex:

build: .

jdk8-image-ex$ docker-compose up

**Spring Boot with Docker**

Refer:

<https://spring.io/guides/gs/spring-boot-docker/>

<https://hub.docker.com/r/frolvlad/alpine-oraclejdk8/>

Docker is a Linux container management toolkit with a "social" aspect, allowing users to publish container images and consume those published by others. A Docker image is a recipe for running a containerized process

gs-spring-boot-docker

src

main

java

com.philips.bootdocker

Application.java

docker

build.gradle

build.gradle

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.5.RELEASE")

**classpath("se.transmode.gradle:gradle-docker:1.2")**

}

}

**group = "qizhonglin" //docker hub 帐号**

apply plugin: 'java'

apply plugin: 'eclipse'

apply plugin: 'idea'

apply plugin: 'spring-boot'

**apply plugin: 'docker'**

jar {

baseName = 'gs-spring-boot-docker'

version = '0.1.0'

}

repositories {

mavenCentral()

}

sourceCompatibility = 1.8

targetCompatibility = 1.8

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

testCompile("org.springframework.boot:spring-boot-starter-test")

}

task wrapper(type: Wrapper) {

gradleVersion = '2.3'

}

// docker任务

task buildDocker(type: Docker, dependsOn: build) {

push = true

applicationName = jar.baseName

dockerfile = file('src/main/docker/Dockerfile')

doFirst {

copy {

from jar

into stageDir

}

}

}

Application.java

@SpringBootApplication

@RestController

public class Application {

@RequestMapping("/")

public String home() {

return "Hello Docker World";

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Dockerfile

FROM frolvlad/alpine-oraclejdk8:slim

VOLUME /tmp

ADD **gs-spring-boot-docker-0.1.0.jar** app.jar

RUN sh -c "touch /app.jar"

ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar" ]

run application, the result **gs-spring-boot-docker-0.1.0.jar will be put to /tmp**

gs-spring-boot-docker$gradle build buildDocker

$docker run -p 8080:8080 -t qizhonglin/gs-spring-boot-docker

$curl localhost:8080

### Bundle

Docker 1.12引入了一种新的文件格式，名为Distributed Application Bundle（即分布式应用包，目前尚处于实验阶段）。Bundle是一种立足于服务之上的新型抽象机制，主要面向全堆栈应用。

一个Docker Bundle文件属于一组服务的声明性规范，负责说明：

运行哪套具体镜像版本

创建怎样的网络

各服务中的容器如何联网并运行

Bundle文件具备全面的可移植性，且可通过软件交付通道实现部署，这是因为其允许大家对多容器Docker应用进行规范指定与版本控制。

Consul 见page202

每个Docker容器将其中运行的服务注册到服务发现工具里。注册的信息可以是IP地址和端口，以便服务之间进行交互

Consul是一种服务发现工具，使用一致性算法的特殊数据存储器，使用Raft一致性算法来提供确定的写入机制。Consul暴露了键值存储系统和服务分类系统，并提供高可用性、高容错能力，并保证强一致性。服务可以将自己注册到Consul，并以高可用且分布式的方式共享这些信息。

consul

Dockerfile

consul.json

FROM ubuntu:14.04

RUN apt-get -qqy update

RUN apt-get -qqy install curl unzip

//安装consul

ADD https://dl.bintray.com/mitchellh/consul/0.3.1\_linux\_amd64.zip /tmp/consul.zip

RUN cd /usr/sbin && unzip /tmp/consul.zip && chmod +x /usr/sbin/consul && rm /tmp/consul.zip

//安装webui

ADD https://dl.bintray.com/mitchellh/consul/0.3.1\_web\_ui.zip /tmp/webui.zip

RUN cd /tmp/ && unzip webui.zip && mv dist/ /webui/

ADD consul.json /config/

//consul的默认端口

EXPOSE 53/udp 8300 8301 8301/udp 8302 8302/udp 8400 8500

VOLUME [ "/data" ]

//启动consul服务

ENTRYPOINT [ "/usr/sbin/consul", "agent", "-config-dir=/config" ]

consul.json

{

"data\_dir": "/data", //保存consul数据

"ui\_dir": "/webui", //指定网页界面文件的位置

"client\_addr": "0.0.0.0", //将consul绑定到容器内的所有网页界面

"ports": { "dns": 53 }, //配置consul服务运行时需要的端口

"recursor": "8.8.8.8" //指定DNS服务器

}

$docker build –t qizhonglin/consul .

$docker run -p 8500:8500 -p 53:53/udp -h node1 qizhonglin/consul -server -bootstrap

-server告诉Consul代理以服务器的模式运行

-bootstrap告诉Consul本节点可以自选举为集群领导者

若出现错误指”53端口已被占用“，可以用杀死占用该端口的进程

$sudo kill 'sudo lsof -t -i:53

测试

localhost:8500/ui/#dc1/services

### 实践: Knowledge module, mysql, meadia wiki

image and container的保存，导入，导出

docker images

docker save one-image > one-image.tar 保存image

docker ps -a 查看容器ID

docker commit <CONTAINER ID> my-container-image 容器 -> image

docker export <CONTAINER ID> container.tar 导出容器

cat container.tar | docker import - container:latest

docker load < one-image.tar 导入image

docker stop container-name

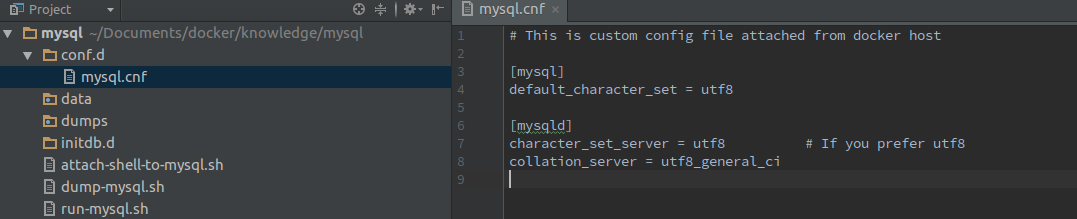
docker rm container-name 删除容器

docker rmi one-image 删除image

**mysq**l

<https://hub.docker.com/_/mysql/>

<https://github.com/besnik/tutorials/tree/master/docker-mysql>



conf.d/ - be used to store mysql configuration files (.cnf) that overrides default configuration

data/ - store mysql data

dumps/ - store sql dumps we will make

initdb.d/ - store scripts (.sh, .sql, .sql.gz) to be executed when container is started for the first time

Method:

**run-mysql.sh** //first time: Starting mysql container，重新由image创建，数据全无

**docker start mysql** //若已经有mysql container，运行即可

open mysql client on container?

attach-shell-to-mysql.sh

#mysql --user=root --password=$MYSQL\_ROOT\_PASSWORD

create full dump of mysql database?

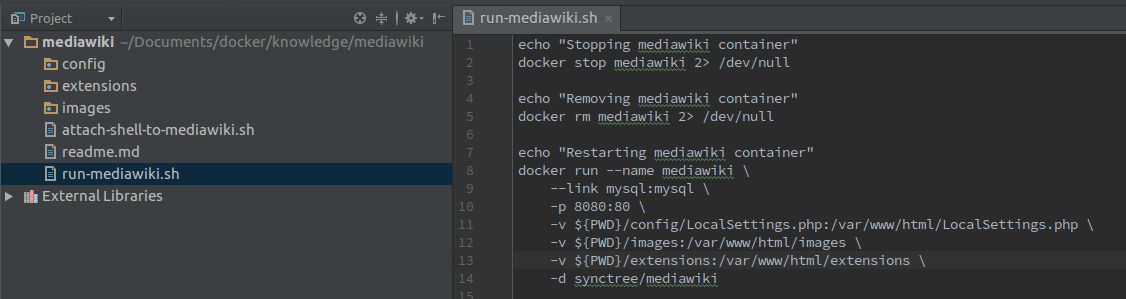
dump-mysql.sh

restore data from full dump?

copy \*.sql to initdb.d/

**synctree-mediawiki**

<https://github.com/besnik/tutorials/tree/master/docker-mediawiki>



Install mediawiki

第一次，需要创建LocalSettings.php, 同时初始化Mediawiki数据库

docker run --name mediawiki \

--link mysql:mysql \

-p 8080:80 \

-d synctree/mediawiki

浏览器访问： localhost/mediawiki

mysql:root@%/mediawiki 连接数据库

wiki名称：knowledge-wiki

admin:admin123 wiki管理员

download LocalSettings.php and mv LocalSettings.php to config/

mediawiki$ docker cp mediawiki:/var/www/html/extensions . copy content from container

mediawiki$ docker cp mediawiki:/var/www/html/images .

./run-mediawiki.sh (第二次一直运行这个 或者 docket start mediawiki)

docker cp config/wiki.png mediawiki:/var/www/html/resources/assets/wiki.png

note: in office with proxy

test the above method, the performance of web-server is very slow…

after dissect, the reason is because of volume

so please use docker cp instead of -v

the recommend method is knowledge/qzlin/readme.md

mediawiki

<https://www.mediawiki.org/wiki/Manual:Running_MediaWiki_on_Debian_or_Ubuntu>

Dockerfile

FROM ubuntu:16.04

MAINTAINER Qizhong Lin <qizhong.lin@philips.com>

ENV MEDIAWIKI\_VERSION 1.27

ENV MEDIAWIKI\_FULL\_VERSION 1.27.1

RUN apt-get update

RUN apt-get install -y apache2 php php-mysql libapache2-mod-php php-xml php-mbstring

RUN apt-get install -y php-apcu php-intl imagemagick inkscape php-gd php-cli

WORKDIR /var/www/html

启动redis server

echo "stopping redis-knowledge container"

docker stop redis-knowledge 2> /dev/null

echo "removing redis-knowledge container"

docker rm redis-knowledge 2> /dev/null

echo "re-starting redis-knowledge container"

docker run --name redis-knowledge -d redis redis-server

flask 连接redis

docker stop flask 2> /dev/null

docker rm flask 2> /dev/null

docker run \

--name flask \

--link mysql-knowledge:mysql \

--link redis-knowledge:redis \

-e DEV\_DATABASE\_URL=mysql://root:q@mysql/knowledge \

-e DATABASE\_URL=mysql://root:q@mysql/knowledge \

-d \

-p 5000:5000 \

-v $SCRIPT\_DIR:/app \

-e MEDIA\_WIKI\_DOMAIN=http://$localIP/mediawiki \

qizhonglin/flask \

python manage.py runserver -h 0.0.0.0 –threaded

如何进入已运行的容器，从而查看flask连接redis的环境变量

docker exec -it flask bash

root…/app# env

# Deployment and Orchestration

Tools to manage, scale, and maintain containerized applications are called orchestrators.

## Kubernetes

Kubernetes provides many tools for scaling, networking, securing and maintaining your containerized applications, All containers in Kubernetes are scheduled as pods,

bb.yaml

apiVersion: apps/v1

kind: **Deployment**

metadata:

name: bb-demo

namespace: default

spec:

replicas: 1

selector:

matchLabels:

bb: web

template:

metadata:

labels:

bb: web

spec:

containers:

- name: bb-site

image: getting-started

imagePullPolicy: Never

---

apiVersion: v1

kind: Service

metadata:

name: bb-entrypoint

namespace: default

spec:

type: **NodePort**

selector:

bb: web

ports:

- port: 3000 // route traffic from port 30001 on your host to port 3000

targetPort: 3000

nodePort: 30001

$ kubectl apply -f bb.yaml //deploy app to Kubernetes

## Docker Swarm [1.12 RC]

Docker1.12通过swarm实现了容器集群，在集群之上可以service命令创建一个服务，实现服务的多实例。在服务之上，通过stack可以描述一组服务，最终实现一个大系统的部署

Docker Engine 1.12 includes swarm mode for natively managing a cluster of Docker Engines called a Swarm. Use the Docker CLI to create a swarm, deploy application services to a swarm, and manage swarm behavior.

* Service discovery
* Load balancing
* Multi-host networking
* Scaling

orchestration and cluster management

The cluster management and orchestration features embedded in the Docker Engine

You enable swarm mode for the Engine by either initializing a swarm or joining an existing swarm.

To deploy your application to a swarm, you submit a service definition to a manager node. The manager node dispatches units of work called tasks to worker nodes.

Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes receive and execute tasks dispatched from manager nodes. By default manager nodes are also worker nodes, but you can configure managers to be manager-only nodes. The agent notifies the manager node of the current state of its assigned tasks so the manager can maintain the desired state.

replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes

global services, the swarm runs one task for the service on every available node

The swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm. The swarm manager can automatically assign the service a PublishedPort;The swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service

操作过程：

1. initializing a cluster of Docker Engines in swarm mode

$docker swarm init --listen-addr <MANAGER-IP>:<PORT> //ssh终端连上manager node

1. adding nodes to the swarm

$docker swarm join <MANAGER-IP>:<PORT> //ssh终端连上worker node

1. deploying application services to the swarm

//ssh终端连上manager node

$ docker service create --replicas 1 --name helloworld alpine ping docker.com

1. managing the swarm once you have everything running

//ssh终端连上manager node

see which nodes are running the service:

$ docker service tasks helloworld 可能是worker1, worker2,甚至是manager node

Scale the service in the swarm

$ docker service scale helloworld=5

see the updated task list

$ docker service tasks helloworld

drain a node that had a task assigned to it, 从而worker1 node的任务会被别的worker node接手

$docker node update --availability drain worker1

操作命令：

（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 下面在一个三节点环境中，针对各个命令进行试用，详细展开说明

Docker Swarm概念解析

容器编排

将容器从只能部署在单一主机之上，升级为能够将多种复杂的容器应用广泛部署于大量设备当中.要实现编排方案，我们需要一套独立于基础设施之外的分布式平台，贯穿整个应用生命周期始终，且能够在硬件发生故障或软件更新时继续正常起效。

$docker swarm init

这条命令会创建一个Raft节点协作组。作为首个节点，其负责实现管理功能，意味着它可以接收命令并调度任务。随着向Swarm中添加更多节点，后续节点将默认作为工作节点存在，分别执行由管理节点分派的任务。管理节点属于Raft协作组的组成部分。我们利用一套经过优化的Raft存储体系，其直接读取内存内容以提升调度工作的性能表现。

$docker service create –name frontend –replicas 5 -p 80:80/tcp nginx:latest

由5套Nginx容器构成的Swarm作为单一内部负载均衡型服务，且于Swarm内任意节点的端口80上进行交付

在创建服务时，我们可以选择创建复制型或者全局服务。复制型服务意味着我们定义的任意数量的容器都可扩散至全部可用主机。相比之下，全局服务则代表调度Swarm当中每台主机上同一容器的一个实例。

Docker如何实现弹性?

warm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。

安全性

Docker 1.12提供经过严格认证的TLS、身份验证、授权与加密机制来保护Swarm中的各相关节点，且全部采取开箱即用的实现方式。



在建立首个管理节点时，Docker Engine会生成新的认证中心（简称CA）以及一组初始证书。在这一初始步骤完成后，加入该Swarm的每个节点都会自动被分配予一份新证书、随机生成的ID以及Swarm中的当前角色（管理节点或者工作节点）。这些证书将被作为其加密安全节点的身份且贯穿于整个生命周期，而管理节点则利用证书确保任务与更新以安全方式进行。

以自动化方式解决了TLS证书中最难搞定的部分：证书轮换。

从深层角度看，参与至Swarm内的各个节点都会持续不断地刷新自身证书，确保可能存在的泄露或者违规证书不会长久有效。用户可以对各证书的轮换频率进行设定，且最高可设置为每30分钟更换一次

# Tilt (microservice dev env)

<https://tilt.dev/>

Kubernetes for Prod, Tilt for Dev = Dockerfile + Kubernetes YAML.

Tilt is a **microservice development environment** for teams that deploy to Kubernetes. Tilt automates all the steps from a code change to a new process: watching files, building container images, and bringing your environment up-to-date.

Kubernetes defines building blocks for running servers together – such as containers, pods, and services.

Kubernetes Workloads is a Kubernetes object that has a container = image + logs + …

Resources: a Docker image to build + Kubernetes YAML to apply, or a command to run locally.

## Tiltfile (written in Starlark, subset of Python)

def microservice\_yaml(name): // 定义函数

read\_file('config/%s.yaml' % name)

return local('./config/generate.py %s' % name) // 执行命令`generate.py {name}`

services = ['frontend', 'backend', 'users', 'graphql'] //变量

[k8s\_yaml(microservice\_yaml(service)) for service in services]

#等价docker build -t companyname/frontend -f frontend/Dockerfile.dev frontend

docker\_build(

"companyname/frontend", // image name

"frontend", // build from dir frontend

dockerfile="frontend/Dockerfile.dev",

live\_update=[

sync('frontend', '/app'), # Sync dir frontend into container dir /app.

# if start-time.txt was changed, the server reloads

run('touch /app/app.py', trigger='./start-time.txt')]

)

# Configuring Kubernetes Resources

k8s\_resource(workload='frontend', port\_forwards='9000:8000' ) # localhost:9000 <- 8000

k8s\_resource('frontend', resource\_deps=['database']) // frontend depends on database

// custom\_build as subprocesses of Tilt, build image, push image to image registry

custom\_build(

'frontend',

'docker build -t $EXPECTED\_REF frontend',

['./frontend'],

)

**// many Tiltfiles and Repos**

load('../common/Tiltfile', 'VERSION', 'common\_config\_yaml') // load shared functions

k8s\_yaml(common\_config\_yaml())

include('./frontend/Tiltfile') // 同目录下load sub-Tiltfiles

include('./backend/Tiltfile')

USE\_OAUTH2 = os.path.exists('../.secrets/values-dev.yaml') // load services conditionally

if USE\_OAUTH2:

symbols = load\_dynamic('../oauth2-proxy/Tiltfile')

// Run only some services

config.define\_string\_list("to-run", args=True)

cfg = config.parse()

groups = {

'consumer': ['a', 'b', 'c'], // some services as feature consumer.

'enterprise': ['a', 'b', 'd'],

}

resources = []

for arg in cfg.get('to-run', []):

if arg in groups: resources += groups[arg]

else: resources.append(arg) # individual services `tilt up a b d`

config.set\_enabled\_resources(resources)

config.define\_string\_list("to-edit")

to\_edit = cfg.get('to-edit', [])

if 'a' in to\_edit: docker\_build('a', './a')

if 'b' in to\_edit: docker\_build('b', './b')

if 'c' in to\_edit: docker\_build('c', './c')

if 'd' in to\_edit: docker\_build('d', './d')

## Tilt CLI

$ tilt get uiresources // list all resources

$ tile up // run all services

$ tilt up consumer // run services in consumer

范例1: flask web服务

app

templates

index.html

app.py

requirements.txt

**Dockerfile**

**kubernetes.yaml**

**Tiltfile**

Dockerfile

FROM python:3.6

WORKDIR /app

ADD . .

RUN pip install -r requirements.txt

ENTRYPOINT ["python", "/app/app.py"]

Kubernetes.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: **example-python // resource name**

labels:

app: example-python

spec:

selector:

matchLabels:

app: example-python

template:

metadata:

labels:

app: example-python

spec:

containers:

- name: example-python

image: **example-python-image //使用docker image**

ports:

- containerPort: 8000

Tiltfile

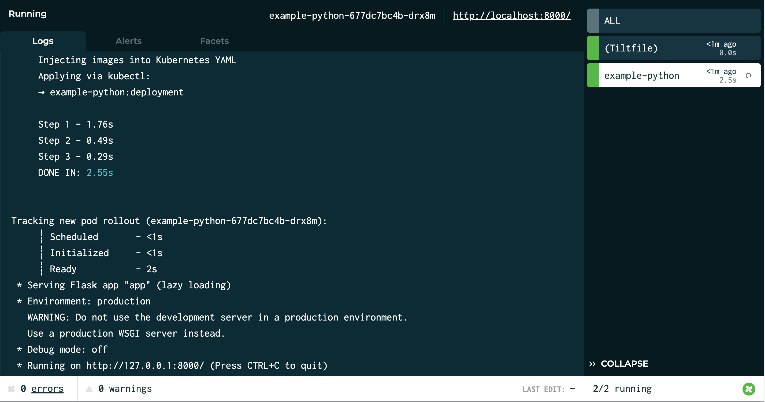
docker\_build('example-python-image', '.') // build image with name and current Dir

k8s\_yaml('kubernetes.yaml') //load Kubernetes deployment YAML,

k8s\_resource('example-python', port\_forwards=8000) //port-forwarding localhost:8000

$ tilt up

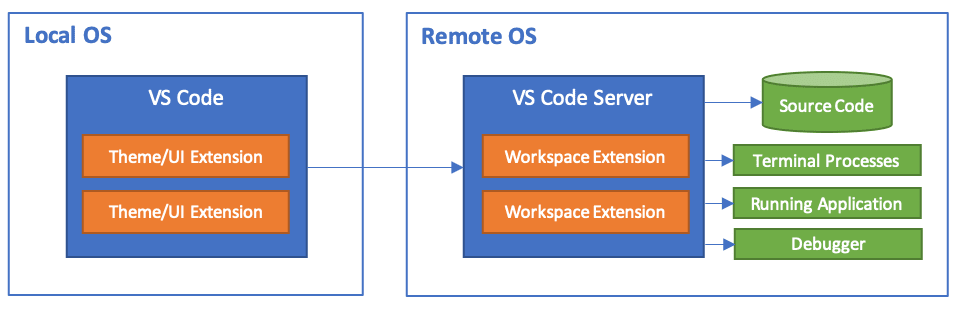
Tilt打开浏览器，显示app状态和日志



if the image has changed, the containers restart.

# Remote Development (本地vscode,远程服务器要装gcc, gdb)

<https://code.visualstudio.com/docs/remote/remote-overview>



安装vscode on win

1. 远程登录Linux系统

$sudo apt-get install openssh-server -y //OpenSSH服务器for linux

$ ps –e | grep ssh //查看ssh服务是否启动

$ service ssh start //开启sshd服务

OpenSSH客户端 for win, win10自带

>Get-WindowsCapability -Online | ? Name -like ‘OpenSSH\*’ //测试是否装OpenSSH

1. 配置免密远程登录（本地私密与远程公密）

>ssh-keygen -t rsa -b 4096 //win生成c:/users/xxx/.ssh/id\_rsa.pub, id\_rsa

//公钥内容添加到远程主机.ssh/authorized\_keys, open PowerShell

>scp .ssh\id\_rsa.pub [qzlin@192.168.1.7:/home/qzlin/.ssh/id\_rsa.pub](mailto:qzlin@192.168.1.7:/home/qzlin/.ssh/id_rsa.pub)

>ssh qzlin@192.168.1.7

$ cat id\_rsa.pub >> authorized\_keys

1. 配置VS Code远程开发插件

VS code, ctrl+shift+x -> install Remote – SSH

ssh -> connect to host -> Add New SSH Host: ssh user@ip

选择ssh的config:

Host 192.168.1.1

HostName 192.168.1.1 #填写远程服务器ip

User root #填写远程服务器user

连接远程服务器

连接成功后，通过File->Open->File/Folder打开文件或者文件夹，然后就可以像编辑本地文件一样编辑远程文件了