Table of Contents

[DevOps 2](#_Toc156225834)

[**虚拟开发环境(Hyper-V, VirtualBox, Vagrant, Docker)** 4](#_Toc156225835)

[Microsoft Hyper-V 5](#_Toc156225836)

[VirtualBox 5](#_Toc156225837)

[Vagrant 6](#_Toc156225838)

[Docker 15](#_Toc156225839)

[Containerize app 15](#_Toc156225840)

[Dockerfile指令 18](#_Toc156225841)

[Docker命令 19](#_Toc156225842)

[Docker镜像和仓库 22](#_Toc156225843)

[范例：测试静态网站，web服务， 24](#_Toc156225844)

[实战：java程序，Spring Boot, 33](#_Toc156225845)

[Bundle 36](#_Toc156225846)

[实践: Knowledge module, mysql, meadia wiki 37](#_Toc156225847)

[Docker client: python-on-whales, Docker-py 40](#_Toc156225848)

[Deployment and Orchestration 41](#_Toc156225849)

[Kubernetes 41](#_Toc156225850)

[**kubectl 命令行接口（CLI）** 43](#_Toc156225851)

[**如何部署应用程序？** 45](#_Toc156225852)

[**部署托管的pod: rc, rs, ds, Job or Deployment** 49](#_Toc156225853)

[服务(与pod通信)，卷(挂载磁盘到容器)，ConfigMap/Secret(配置应用程序) 54](#_Toc156225854)

[Docker Swarm [1.12 RC] 62](#_Toc156225855)

[Tilt (microservice dev env) 64](#_Toc156225856)

[Tilt CLI 66](#_Toc156225857)

[Tiltfile (written in Starlark, subset of Python) 67](#_Toc156225858)

[Remote Development (本地vscode,远程服务器要装gcc, gdb) 72](#_Toc156225859)

[CI (continuous integration) CD (Continuous Delivery) 73](#_Toc156225860)

[理论知识 73](#_Toc156225861)

[Jenkins : automation server (build, test and deploy) 77](#_Toc156225862)

[Example: mmsim 83](#_Toc156225863)

[PMD Source Code Analyzer Project 85](#_Toc156225864)

[Source Code Management (SCM) 86](#_Toc156225865)

[Azure DevOps - Git 86](#_Toc156225866)

[Git command 90](#_Toc156225867)

[Binary & Artifact Management 100](#_Toc156225868)

[JFrog Artifactory 100](#_Toc156225869)

[Nexus Repository Manager 101](#_Toc156225870)

[BDD（Behavior Driven Development行为驱动开发） 103](#_Toc156225871)

[自动化运维：Ansible 105](#_Toc156225872)

[ad-doc 106](#_Toc156225873)

[Playbook 107](#_Toc156225874)

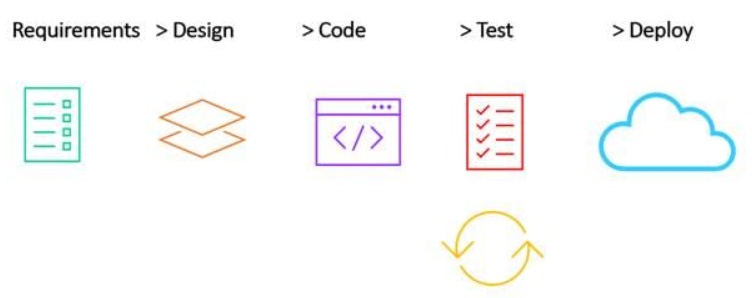
[Security Assessment 112](#_Toc156225875)

[CVSS: Common Vulnerability Scoring System通用漏洞评分系统 112](#_Toc156225876)

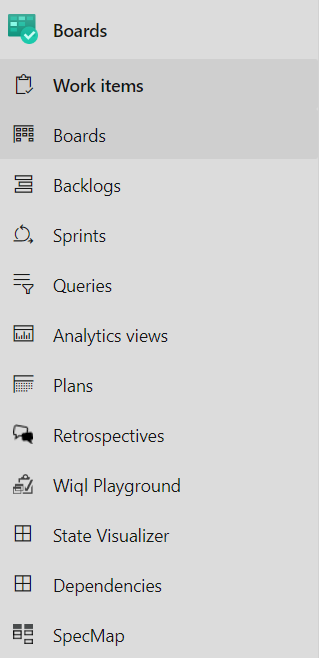
# DevOps

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, …

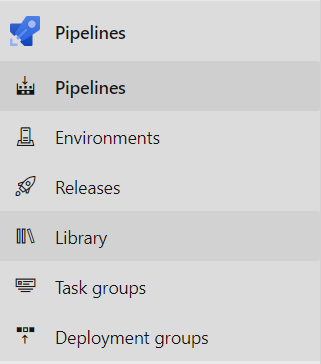
用户修改代码

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。

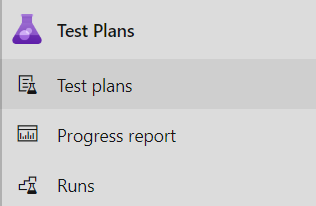


Azure Pipelines 支持构建、测试和部署 Node.js、Python、Java、PHP、Ruby、C/C++、.NET、Android 和 iOS 应用。在 Linux、macOS 和 Windows 上并行运行。

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。

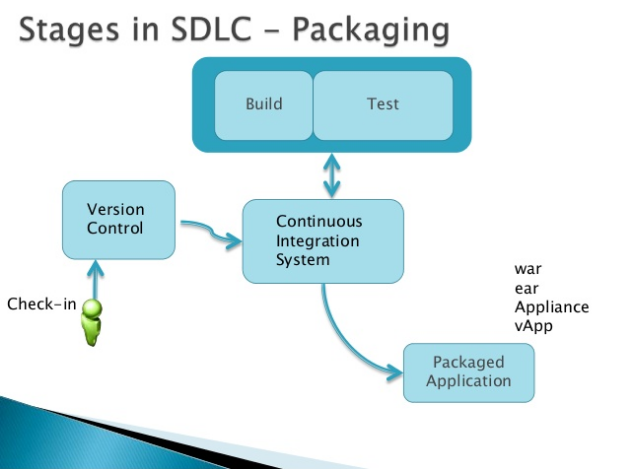
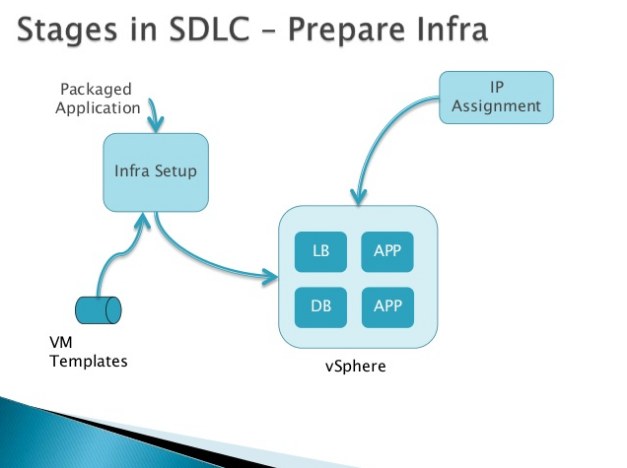
Azure Test Plans: 软件测试和 QA 团队创建测试用例、测试和报告结果以及生成报告



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。

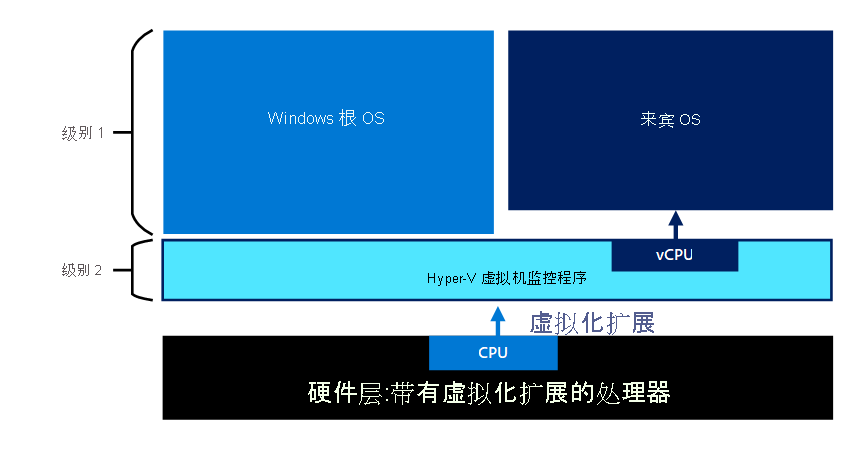
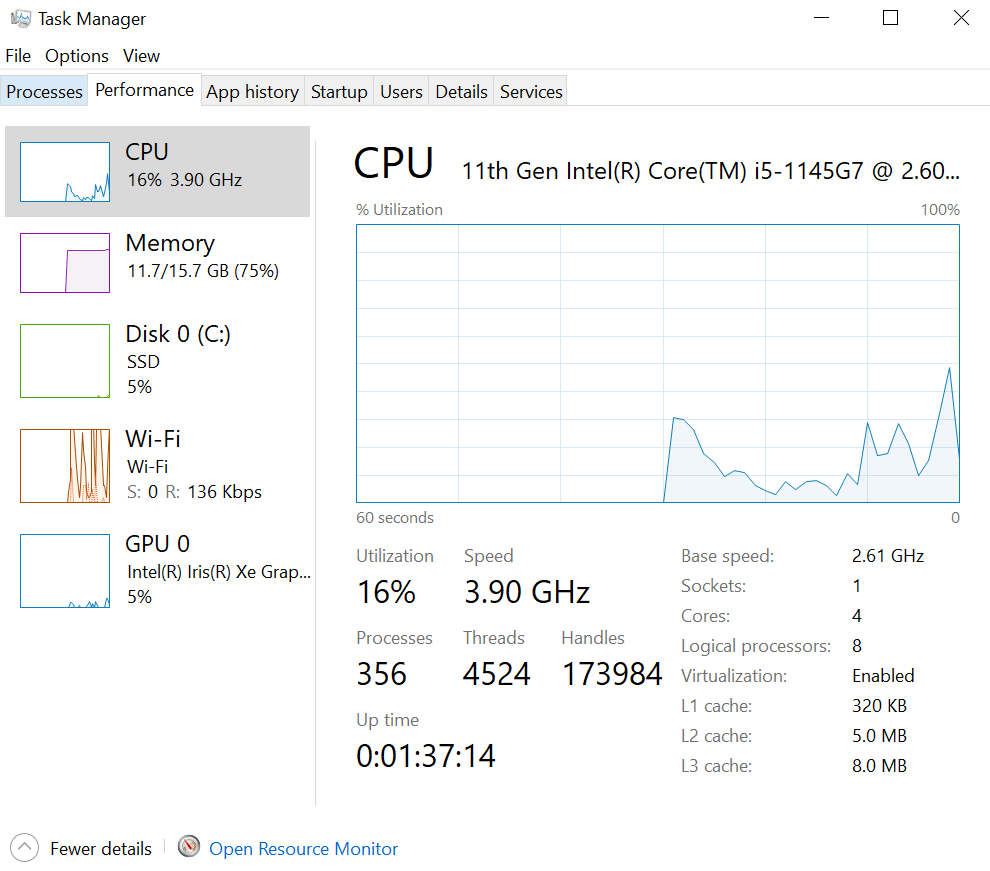
解决：需要虚拟开发环境，在本机跑虚拟机，例如VMware、VirtualBox等。在本机可以运行自己喜欢的OS（Windows、Ubuntu、Mac等），开发的程序运行在虚拟机中，这样迁移到生产环境可以避免环境不一致导致的莫名错误。

Limitations:

Programs that depend on specific hardware will not work well in a virtual machine. For example, games or applications that require processing with **GPUs might not work well**

## 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

**** 

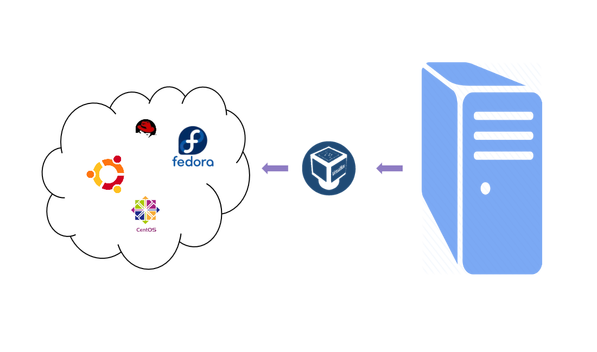
可以在任务管理器的性能选项卡看到自己的电脑是否开启了CPU虚拟化

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

## VirtualBox

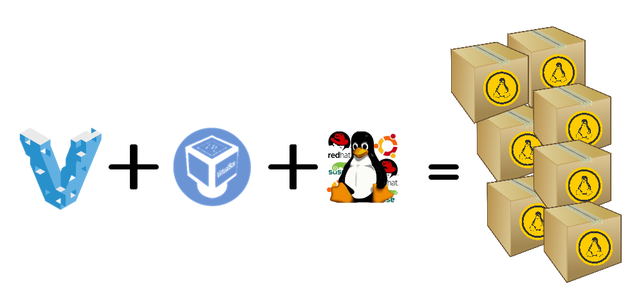
Hypervisor (2007) ，一种运行在基础物理服务器和操作系统之间的中间软件层,可允许多个操作系统和应用共享硬件。也可叫做VMM（ virtual machine monitor ），即虚拟机监视器。

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

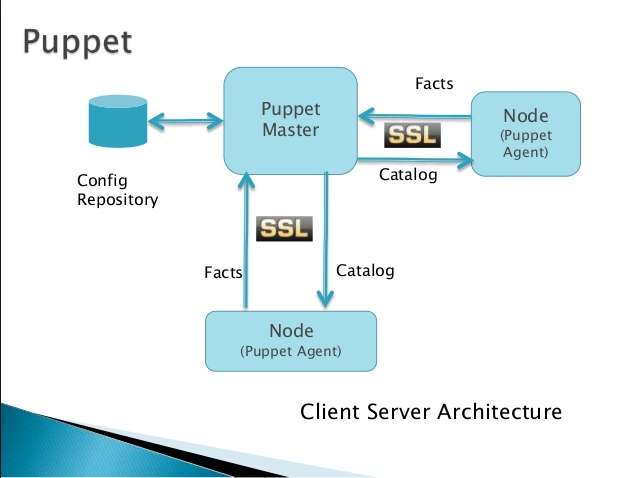
index.html

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

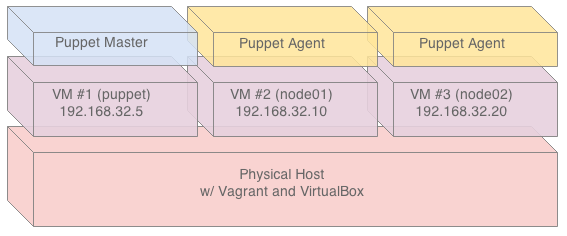
Refer:

<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》

Need to install MongoDB, RabbitMQ, or MySQL? Use Docker to simplify dev and test

Almost every interesting application uses at least one infrastructure service such as a database or a message broker. For example, if you tried to build and/or run the Spring Boot-based user registration service you would have discovered that it needs both MongoDB and RabbitMQ.

One option, of course, is to install both of those services on your machine. Unfortunately, installing a service is not always easy. Also, different projects might need different incompatible versions. Moreover, I’m not a fan of cluttering my machine with random services. Fortunately, there is a great way to solve this problem: Docker. You install Docker and use it to run the services that you need as containers.

Docker only directly runs on Linux so if you are using Mac OSX or Windows the first step is to install Boot2Docker (Mac OSX, Window). Boot2Docker installs the Docker command line locally but runs the Docker daemon in a Virtual Box VM

### Containerize app

* Install Docker

$ 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

若是Mac or Win10, Docker将会自己启动一个VM并在VM中运行Docker守护进程。Docker客户端可执行文件可以在宿主操作系统中使用，并可以与VM中的守护进程通信。

**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）

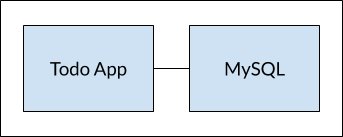
--mount type=bind, src="$(pwd)", target=/app \ //同步工作目录与容器目录/app

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

$ docker pull 用户名/镜像名:tag //默认tag为latest

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 bob\_the\_container 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

**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。

[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**

注意：从Docker 1.13版本开始使用新语法$ docker container run

**//进入正在运行中的后台容器，对于调试容器内运行的应用非常重要**

$ docker exec <container-id> cat /data.txt

$ docker exec -it <container-id> bash

[root@de28...:/#](mailto:root@de28...:/) ps aux // 容器内的进程运行在主机操作系统上

$ ps aux | grep app.js

会发现容器内的进程实际上运行在主机操作系统上，但是容器使用独立的PID Linux命名空间并且有着独立的系列号，完全独立于进程树

[root@de28...:/#](mailto:root@de28...:/) ls // 容器拥有完整的文件系统

容器内的应用不仅拥有独立的文件系统，还有进程，用户，主机名和网络接口

// 查看container 环境变量

$ docker exec <container-id> printenv

$ 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 start bob\_the\_container //启动容器.

$ docker stop daemon\_dave // 停止容器.

$ docker rm daemon\_dave // 删除容器

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

* **Docker 内部网络docker0**

$ 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

* **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资源管理**

$ 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 container ls // 默认只列出正在运行的容器，-a列出包括停止的所有容器

$ docker ps -a //列出现有全部容器， -a列出包括停止的所有容器

//列出镜像列表，-a列出包括中间层, 本地镜像都保存在/var/lib/docker

$ docker images ls

$ docker volume ls // 列出数据卷

$ docker network ls // 列出network

$ df -h // 查看磁盘使用情况 （服务器内存大小）

$ du -sh \* // 查看当前目录下各个文件及目录占用空间大小

$ df -i // 查看inode使用情况

$ df -h /var/lib/docker // 查看docker中剩余空间

$ docker system df // 查看Docker磁盘使用情况

TYPE TOTAL ACTIVE SIZE RECLAIMABLE

Images 88 28 149.3GB 127.2GB (85%)

Containers 102 45 911.4MB 103.7MB (11%)

Local Volumes 0 0 0B 0B

Build Cache 408 0 0B 0B

Images: 所有镜像占用的空间，包括拉取的镜像、本地构建的镜像

Containers: 运行中的容器所占用的空间（没运行就不占空间）

Local Volumes: 本地数据卷的空间

Build Cache: 镜像构建过程串，产生的缓存数据.

RECLAIMABLE: 指可清理的

// 查看单个image/container占用细节，以确定哪个镜像容器或本地卷占用过高空间

$ docker system df -v

// 清理磁盘，删除关闭的容器，无用的数据卷，网络和dangling镜像(即无tag的镜像)

$ docker system prune

$ docker system prune -a // 清理更彻底，连没有容器使用的image也删除

// 旧版docker1.13之前，没有docker system命令，需要手动清理image, container and volume

$ docker rmi - f <image> // -f强制删除镜像，即使有容器引用该镜像

//删除所有关闭的容器remove all Exited containers

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

$ docker rm daemon\_dave // 删除容器

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

$ docker kill <container-name>/<container-id> //杀掉一个多个正在运行的容器

* **docker-compose**

$ 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

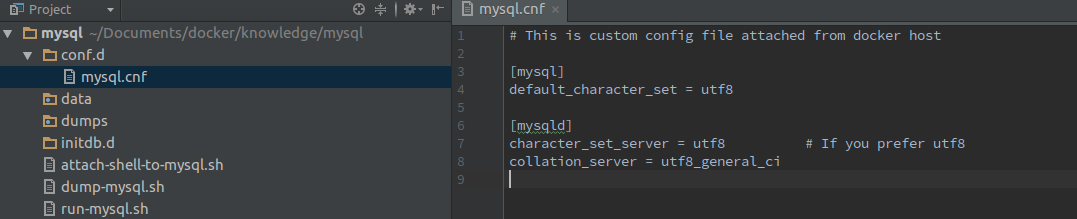
docker rm container-name 删除容器

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?

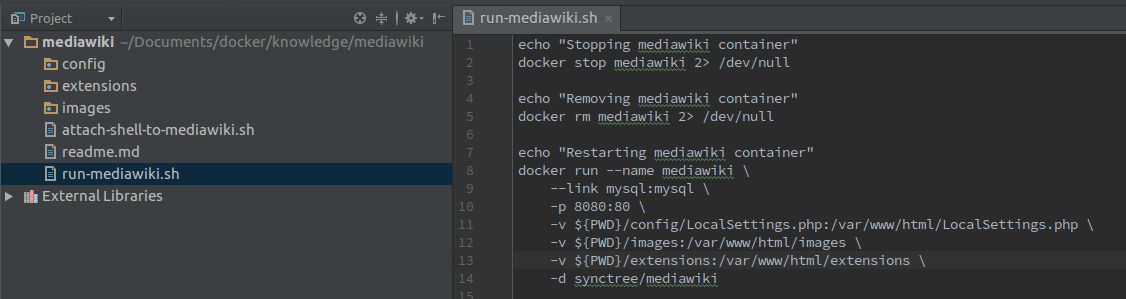
dump-mysql.sh

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

### Docker client: python-on-whales, Docker-py

<https://gabrieldemarmiesse.github.io/python-on-whales/>

**Python on Whales:** is to provide a 1-to-1 mapping between the Docker command line and Python

**Docker-py:** is a complete re-implementation of the Docker client binary (written in Go)

If you need to talk to the Docker engine directly, you need to do low level operations, use docker-py. If you wanted to call the docker command line from Python, do high level operations, use Python on Whales.

$ pip install python-on-whales

from python\_on\_whales import docker

docker.run("hello-world") # docker run hello-world

docker.pull("ubuntu") # docker pull ubuntu

docker.build("./") # docker build ./

docker.compose.up(["my\_service"]) # docker compose up my\_service

docker.image.list() # docker image ls

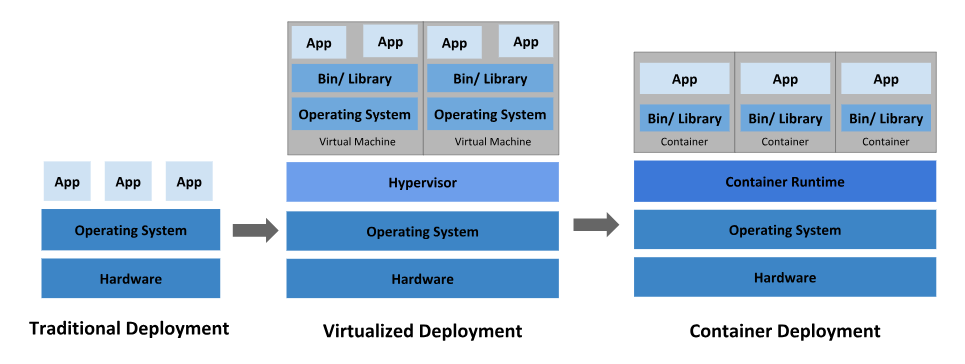
docker.ps() # docker ps

docker.copy() # docker cp

Docker object as Python objects: Container, Images, Volumes, Services... and their attributes

# Deployment and Orchestration

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



大型单体应用正被逐渐分解成小的可独立运行的组件。服务之间可以通过类似HTTP这样的同步协议通信，或者通过像AMQP这样的异步协议通信。

因为组件之间依赖的差异性，应用程序需要同一个库的不同版本是不可避免的。Kubernetes使用Linux容器技术来提供应用的隔离。

一个容器里运行的进程实际上运行在宿主机的操作系统上，但在容器里的进程仍然是和其他进程隔离的。不像虚拟机，进程是运行在不同的操作系统上。一个容器仅仅是运行在宿主机上被隔离的单个进程。

虚拟机的主要好处是提供完全隔离的环境，因为每个虚拟机运行在它自己的Linux内核上，而容器都是调用同一个内核。

## Kubernetes

Book: 《Kubernetes》in Action

容器是打包和运行应用程序的好方式。在生产环境中， 你需要管理运行着应用程序的容器，并确保服务不会下线。一个运行中的容器是一个运行在Docker主机上的进程。容器化的应用程序包含了运行所需的所有内容。

**容器镜像**：**软件包**包含运行应用程序所需的一切：代码和它需要的所有运行时、应用程序和系统库，以及一些基本设置的默认值。

Kubernetes(K8s) 部署和管理容器化，是管理容器的编排工具。Kubernetes provides many tools for scaling, networking, securing and maintaining your containerized applications, all containers in Kubernetes are scheduled as pods。

**// install minikube**

$ curl -Lo minikube

<https://storage.googleapis.com/minikube/releases/v0.23.0/minikube-darwin-amd64>

&& chmod +x minikube

&& sudo mv minikube /usr/local/bin

$ minikube start

启动集群需要花费超过一分钟的时间

**// install kubectl**

$ curl -LO <https://storage.googleapis.com/kubernetes-releases/release>

$(curl -s <https://storage.googleapis.com/kubernetes-releases/release/stable.txt)/bin/>

darwin/amd64/kubectl

&& chmod +x kubectl

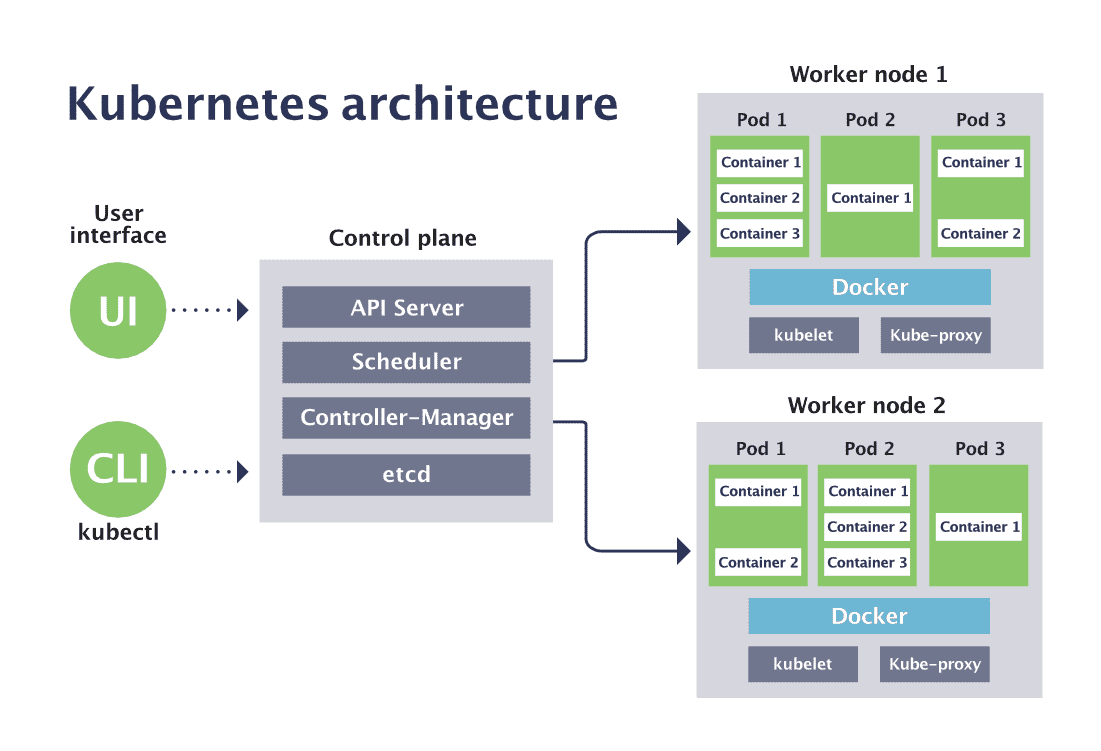
&& sudo mv kubectl /usr/local/bin

**K3s**是轻量级的 Kubernetes 版本，适合在资源受限的硬件上使用，易于设置和使用，因此非常适合用于本地开发集群。

// 下载二进制文件并注册一个系统服务自动启动 K3s(每次电脑启动时)和kubectl CLI

// 二进制文件<100M, 安装和部署<30s

$ curl -sFL https://get.k3s.io | sh – // install K3s



开发者提交应用描述到Kubernetes API Server, Kubernetes会将它们部署到集群的工作节点。 节点上的Kubelets将通知Docker从镜像仓库中拉取容器镜像并运行容器。

Kubernetes的原则是：开发者描述应用程序需要的资源，Kubernetes会根据应用程序的资源需求描述，基于每个节点上的可用资源选择最合适的节点来运行应用程序。Kubernetes监控你的应用程序组件和它们运行的节点，并在节点出现故障时自动将它们重新调度到其他节点。

**Pod**：一个或多个应用容器（如 Docker）. 这些容器**共享存储、网络**、以及怎样运行这些容器的声明，在同一个 Pod 内，所有容器共享一个 IP 地址和端口空间，并且可以通过 localhost发现对方。

为什么引入pod含多个容器比单个容器含多个进程好？如果在单个容器中运行多个不相关的进程，那么保持所有进程运行、管理它们的日志等将会是我们的责任，比如进程崩溃能够自动重启，所有进程记录到相同的标准输出。

同一个pod中的所有容器运行于相同的network and UTS命名空间下运行，因此它们共享相同的主机名和网络接口（IP地址和端口空间），因此容器可以通过localhost与同一个pod中的其他容器进行通信。

Pod是逻辑主机，其行为与非容器世界中的物理主机或虚拟机非常相似。此外运行在同一个pod中的进程与运行在同一物理机或虚拟机上的进程相似，只是每个进程都封装在一个容器之中。

**Deployment/Job(工作负载资源):** 创建和管理一组Pod.

* Deployment管理集群**无状态应用**，Deployment中的Pod等价可替换。
* Job用来执行一次性任务，而 CronJob 用来执行的根据时间规划反复运行的任务。

**kubectl 命令行接口（CLI）**

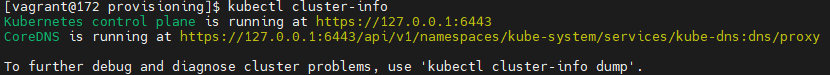
kubectl 使用 Kubernetes API 与集群进行交互

<https://jamesdefabia.github.io/docs/user-guide/kubectl/kubectl/>

$ kubectl version

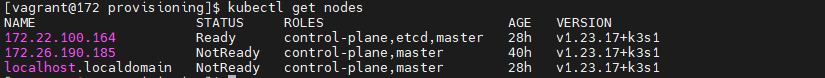
$ kubectl --help

$ kubectl cluster-info // 查看集群是否正常工作



**// kubectl get 列出各种Kubernetes对象**

$ kubectl get nodes // 列出集群节点.



$ kubectl get namespace // 查看名字空间.

$ kubectl get deployments -A

-A 表示--all-namespaces

-n mmsim 表示--namespaces mmsim，仅获取指定名字空间内的deployments

$ kubectl get rc -A // 查看ReplicationController

$ kubectl get rs -A // 查看ReplicaSet

$ kubectl get ds -A // 查看DaemonSet

$ kubectl get jobs -A // 查看Job

$ kubectl get pods -A

$ kubectl get po -A

$ kubectl get po -A -a // -a 等于--show-all, 列出所有pods, 包括运行结束的

$ kubectl get services -A

$ kubectl get svc -A

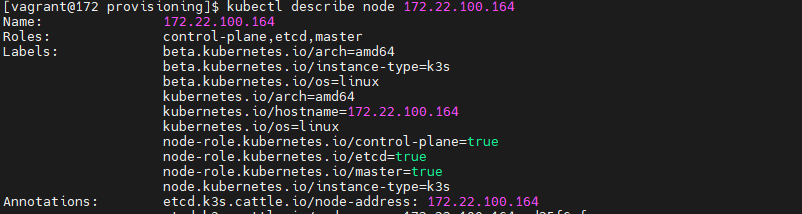
$ kubectl get pv // PersistentVolume

$ kubectl get pvc // PersistentVolumeClaim

**// kubectl describe 显示对象的详细信息**

$ kubectl describe nodes // 所有节点信息.

$ kubectl describe nodes <node-name> // 节点状态, CPU,内存,系统信息,…



$ kubectl describe rc <rc-name>

$ kubectl describe pods

$ kubectl describe pods -n <namespace> <pod-name>

$ kubectl describe services

$ kubectl describe services -n <namespace> <service-name>

**// kubectl explain pods 查看API对象支持哪些属性**

$ kubectl explain pods

$ kubectl explain pod.spec

$ kubectl logs -n <namespace> <pod-name> // 打印 Pod当前容器的日志

$ kubectl logs -n <namespace> <pod-name> --previous // 打印pod前一个容器的日志

$ kubectl exec -n <namespace> <pod-name> -- env // 在 Pod 容器执行命令，列出环境变量

$ kubectl exec -n <namespace> <pod-name> -- touch /var/ready

$ kubectl exec -it -n <namespace> <pod-name> -- bash // Pod容器中启动bash 会话

[p3rtp@<pod-name> /]$ ls

[p3rtp@<pod-name> /]$ $ cat /opt/rt/patient-list-service/rt.py

$ kubectl delete po <pod-name> // 按名称删除pod.

在删除pod的过程中，实际上我们在指示Kubernetes终止该pod中的所有容器。Kubernetes向进程发送一个SIGTERM信号并等待一定的秒数（默认为30s）,使其正常关闭。

$ kubectl delete po -l env=debug // 通过标签选择器删除pod

$ kubectl delete po ns custom-namespace // 通过命名空间来删除pod.

**删除pod, 会重启一个新的pod**, 因为ReplicationController会保证pod的运行

$ kubectl delete all --all

第一个all指定正在删除所有资源类型

第二个--all指定将删除所有资源实例

**如何部署应用程序？**

kubia

app.js

Dockerfile

**app.js**

const http = require('http')

const os = require('os')

console.log("Kubia server starting …")

var handler = function(request, response) {

console.log("Received request from " + request.connection.remoteAddress);

response.writeHead(200);

response.end("’You have hit " + os.hostname() + "\n");

};

var www = http.createServer(handler);

[www.listen(8080)](http://www.listen(8080));

**Dockerfile**

FROM node:7

ADD app.js /app.js

ENTRYPOINT ["node", "app.js"]

$ docker build -t kubia . // build image `kubia`

**// 运行应用app.js via docker container**

$ docker run --name kubia-container -p 8080:8080 -d kubia // 运行容器镜像

$ curl localhost:8080

$ docker logs kubia-container // 查看应用程序日志

$ docker ps // 列出运行中的容器

$ docker exec -it kubia-container bash // 在已有的容器内部运行shell

root@xxx:/# ps aux // 从容器内列出进程

$ ps aux | grep app.js // 运行在主机操作系统上的容器进程

$ docker stop kubia-container // 停止容器

$ docker rm kubia-container // 删除容器

**// 部署应用app.js via docker container in k8s pod**

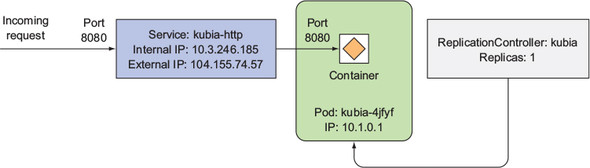
$ kubectl run kubia --image=kubia --port=8080 --generator=run/v1

--image=kubia 指定要运行的容器镜像

--port=8080 告诉Kubernetes应用正在监听8080端口

--generator 创建ReplicationController,而不是Deployment

当运行kubectl 命令时，1) 它通过向Kubernetes API服务器发送一个REST HTTP请示，在集群中创建一个新的ReplicationController对象。2) 然后ReplicationController创建一个新的pod, 调度器将其调度到一个工作节点上。3) Kubelet看到pod被调用度到节点上，就会通知docker从镜像中心拉取指定的镜像并运行容器。



// 访问web应用

$ kubectl expose rc kubia --type=LoadBalancer --name kubia-http

要让pod能够从外部访问，需要通过服务对象公开它，要创建LoadBalancer类型的服务，通过负载均衡的公共IP访问pod.

$ kubectl get services // 查看外部IP

$ curl 104.155.74.57:8080

为什么需要服务？pod的存在是短暂的，消失的pod被ReplicationController替换为新的pod,导致pod IP地址不同，这就是需要服务的地方---解决不断变化的pod IP地址的问题。

// 服务请求切换到所有pod上

$ kubectl scale rc kubia –replicas=3 //增加期望的副本数

$ kubectl get rc

$ kubect get pods

$ curl 104.155.74.57:8080

$ curl 104.155.74.57:8080

**// 部署应用app.js via docker container in k8s pod via YAML or JSON描述文件**

$ kubectl get po kubia-xxx -o yaml // 列出已部署pod的完整yaml

**kubia-manual.yaml**

apiVersion: v1 # kubernetes API version

kind: Pod # Kubernets对象/资源类型

metadata: # 元数据（名称, 标签和注解等）

name: kubia-manual

spec: # 规格/内容（容器列表，volume等）

containers:

- image: kubia # 容器镜像

name: kubia # 容器名称

ports:

-containerPort: 8080 # 应用监听端口

protocol: TCP

$ kubectl create -f kubia-manual.yaml // 创建pod

$ kubectl get pod kubia-manual -o yaml

$ kubectl get pod kubia-manual -o json

容器化的应用程序通常会将日志记录到标准输出和标准错误流，而不是将其写入文件

$ kubectl logs kubia-manual // 查看pod日志

$ kubectl logs kubia-manual -c kubia // 获取多容器日志时指定容器名称

无service下与某个特定的pod进行通信（如对pod进行测试和调试）

$ kubectl port-forward kubia-manual 8888:8080 // 本地网络端口转发到pod端口

$ curl localhost::8888

**标签：**可以附加到资源的任意键值对，用来组织pod。

**kubia-manual.yaml**

metadata:

labels:

creation\_method: manual

env: prod

$ kubectl get po --show-labels

$ kubectl label po kubia-manual env=debug --overwrite // 修改pod标签

// 使用标签选择器列出pod子集

$ kubectl get po -l env

$ kubectl get po -l '!env' // 不含env标签的pod

$ kubectl get po -l env=prod

$ kubectl get po -l env=!prod

$ kubectl get po -l env in (prod, devel)

$ kubectl get po -l env notin (prod, devel)

// 使用标签和选择器来约束pod调度

$ kubectl label node <node-name> gpu=true // 使用标签分类工作节点

Kubia-pu.yaml

Spec:

nodeSelector: // 将pod调度到特定节点

gpu: "true"

**注解**：键值对，主要用于工具使用。大量使用注解可以为每个pod或其他API对象添加说明，以便每个使用该集群的人都可以快速查找有关每个单独对象的信息。

**kubia-manual.yaml**

metadata:

annotations:

kubernetes.io/created-by: {…}

**命名空间：**每个对象都可以有多个标签，对象组可以重叠。但**命名空间是将对象分割成完全独立且不重叠的组**。命名空间为资源名称提供了一个作用域

**custom-namespace.yaml**

apiVersion: v1

kind: Namespace // 定义一个命名空间

metadata:

name: custom-namespace // 命名空间名称

$ kubectl get ns

$ kubectl get po -n kube-system // 若不指定-n, 默认为命名空间default

**kubia-manual.yaml // 使用命名空间**

metadata:

namespace: custom-namespace

Kubernetes通过**存活探针liveness probe**检查容器是否还在运行。可以为pod中的每个容器单独指定存活探针，如果探测失败，Kubernetes将定期执行探针并重新启动容器。

Kubernetes有以下三种探测容器的机制：

* HTTP GET探针对容器IP地址执行请求，如果服务器返回错误响应状态码或者根本没有响应，那么探测就被认为是失败的，容器将被重新启动。
* TCP套接字探针尝试与容器指定端口建立TCP连接。如果连接成功建立，则探测成功。否则容器重新启动。
* Exec探针在容器内执行任意命令，并检查命令的退出状态码。如果状态码是0,则探测成功，那有其他状态码都被认为失败。

kubia-liveness-probe.yaml

apiVersion: v1

kind: Pod

metadata:

name: kubia-liveness

spec:

containers:

- image: kubia

name: kubia

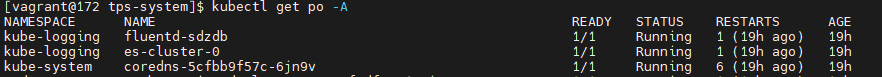
livenessProbe: # HTTP GET 存活探针

httpGet:

path:/

port: 8080

initialDelaySeconds: 15 # kubernetes会在第一次探测前等待15秒



RESTARTS列显示pod容器已被重启的次数

当容器被强行终止时，会创建一个全新的容器，而不是重启原来的容器。

同存活探针一样，Kubernetes通过**就绪探针**检查容器是否准备好接收客户端请求。

* Exec探针，执行进程的地方。容器的状态由进程的退出状态代码确定。
* HTTP GET探针，向容器发送HTTP GET请求，通过响应的HTTP状态代码判断容器是否准备好。
* TCP socket探针，它打开一个TCP连接到容器的指定端口，如果连接已建立，则认为容器已准备就绪。

存活探针通过杀死异常的容器并用新的正常容器替代它们来保持pod正常工作，而就绪探针确保只有准备好处理请求的pod才可以接收它们（请求）。

就绪探针确保客户端只与正常的pod交互，并且永远不会知道系统存在问题。

**部署托管的pod: rc, rs, ds, Job or Deployment**

不应该直接创建pod, 因为如果pod被错误地删除，它们正在运行的节点异常，或者它们从节点中被逐出时，它们将不会被重新创建。

通过创建ReplicationController (rc), ReplicaSet (rs), DaemonSet (ds), Job or Deployment等资源创建和管理pod。

rc旨在创建和管理一个pod的多个副本replicas, 这就是rc名字的由来。

rc三个关键部分: pod选择器、副本个数和pod模板

kubia-rc.yaml

apiVersion: v1

kind: **ReplicationController**

metadata:

name: kubia

spec:

replicas: 3 # 副本个数 （pod实例的目标数目）

selector: # pod选择器（决定rc的操作对象）

app: kubia

template: # pod模板（用于创建新pod）

metadata:

labels:

app: kubia

spec:

containers:

- image: kubia

name: kubia

ports:

-containerPort: 8080

$ kubectl create -f kubia-rc.yaml

由于没有任何pod有app=kubia标签，rc会根据pod模板启动三个新的pod.

$ kubectl get rc

$ kubectl get pods

$ kubectl delete pod kubia-xxx # 手动删除pod, rc会立即启动新容器

更改pod标签，那么它不再与rc的pod选择器匹配，从而rc不再管这个pod了，rc会启动新的pod。

$ kubectl label pod <pod-name> app=foo --overwrite

修改pod模板，在启动新容器时会用上更改后的模板，主要用于升级pod

$ kubectl edit rc <rc-name>

修改副本数目

$ kubectl scale rc <rc-name> --replicas=3

$ kubectl delete rc <rc-name> # 删除rc, 会导致托管的pod也会被删除

$ kubectl delete rc <rc-name> --cascade=false #仅删除rc,但保留pod

推荐使用ReplicaSet而不是ReplicationController

kubia-rs.yaml

apiVersion: apps/v1beta2 # 注意不是v1版本

kind: **ReplicaSet**

metadata:

name: kubia

spec:

replicas: 3 # 副本个数

selector: # pod选择器

matchLabels:

app: kubia

template: # pod模板

…

$ kubectl create -f kubia-rs.yaml

$ kubectl get rs

$ kubectl delete rs <rs-name>

使用RC,RS and Deployment在Kubernetes集群上运行部署特定数量的pod。使用ds在每个节点上运行一个pod，比如希望在每个节点上运行日志收集器和资源监控器。

ssd-monitor-daemonset.yaml

apiVersion: apps/v1beta2

kind: **DaemonSet**

metadata:

name: ssd-monitor

spec:

selector: # pod选择器

matchLabels:

app: ssd-monitor

template: # pod模板

…

$ kubectl create -f ssd-monitor-daemonset.yaml

$ kubectl get ds

rc, rs and ds会持续运行任务，永远达不到完成态，这些pod中的进程在退出时会重新启动。Job资源托管pod（执行批处理任务）, 该pod在内部进程成功结束时，不重启容器，一旦任务完成，pod就被认为处于完成状态。

exporter.yaml

apiVersion: batch/v1

kind: **Job**

metadata:

name: batch-job

spec: # 没有指定pod选择器，直接根据pod模板中的标签创建

template:

metadata:

labels:

app: batch-job

spec:

restartPolicy: OnFailure # Job不能使用Always为默认的重新启动策略

containers:

-name: main

Image: batch-job

$ kubectl create -f exporter.yaml

$ kubectl get jobs

$ kubectl get po -a

-a 等于--show-all, 列出所有pods, 包括运行结束的

完成后pod未被删除的原因是允许你查阅其日志

$ kubectl logs batch-job-xxx

Job资源在创建时会立即运行pod. CronJob资源支持cron任务，指定时间或重复运行

cronjob.yaml

apiVersion: batch/v1beta2

kind: **CronJob**

metadata:

name: batch-job-every-fifteen-minutes

spec:

schedule: "0, 15, 30, 45, \* \* \* \*" 每天每小时0,15, 30, 45分钟运行

startingDeadlineSeconds: 15 pod在预定时间15秒后开始运行

jobTemplate:

spec:

template: # pod模板

…

$ kubectl create -f cronjob.yaml

//创建 Deployment 时, Kubernetes 创建了一个 Pod 来托管你的应用实例。

$ kubectl create deployment nginx --image nginx

$ kubectl create deployment k8s-bc --image=gcr.io/google-samples/kubernetes-bootcamp:v1

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

$ kubectl create -f bb.yaml # 生成bb.yaml, 描述对象期望状态

apiVersion: apps/v1

kind: Deployment# 资源类型Deployment, Job, Ingress, Service

metadata:

name: nginx-demo # 资源名字

# 将资源划分为组，default, kube-node-lease, kube-public, kube-system

namespace: default

# 标签指定对用户相关对象的标识属性，以便选择和查找对象

labels:

environment: production

app: nginx

# 为对象附加非标识元数据，客户端（如工具和库）能够获取这些元数据信息

annotations:

imageregistry: "https://hub.docker.com/"

spec: # 指定资源内容

replicas: 1

selector:

matchLabels:

nginx: web

//工作负载资源的控制器通常使用Pod Template来替你创建 Pod 并管理它们

template:

metadata:

labels:

nginx: web

spec:

containers:

- name: nginx # 容器名字

image: nginx:1.14.2 # 容器镜像

imagePullPolicy: Never # 每次启动时检查和更新image策略

// 默认情况下，Pod 只能通过 Kubernetes 集群中的内部 IP 地址访问。 **要外部访问容器,须将 Pod 暴露为Service**，定义了 Pod 的逻辑集和访问 Pod 的协议

apiVersion: v1

kind: **Service** # 创建service提供对外访问的接口

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

kube-node-lease: 该名字空间包含用于与各个节点关联的 Lease（租约）对象。 节点租约允许 kubelet 发送心跳， 由此控制面能够检测到节点故障。

kube-system(该名字空间用于 Kubernetes 系统创建的对象)

Pod 示例，它由一个运行镜像 nginx:1.14.2 的容器组成

**pods/simple-pod.yml**

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

$ kubectl apply -f pods/simple-pod.yaml

Kubernetes 集群中的 Pod 主要有两种用法：

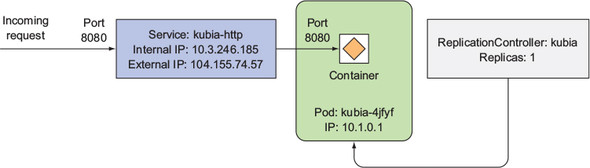
* 运行单个容器的 Pod。"每个 Pod 一个容器" 模型是最常见的 Kubernetes 用例；
* 运行多个协同工作的容器的 Pod

### 服务(与pod通信)，卷(挂载磁盘到容器)，ConfigMap/Secret(配置应用程序)

**服务**

客户端通过IP地址和端口号建立连接，这些连接会被路由到提供该服务的任意一个pod上。

通过为前端pod创建服务，并且将其配置成可以在集群外部访问，可以暴露一个单一不变的IP地址让外部的客户端连接pod.同理可以为后台数据库pod创建服务，能够让前端的pod通过环境变量或DNS以及服务名来访问后端服务。



如果pod的标签与服务的pod选择器相匹配，那么pod就将作为服务的后端。

**# 通过kubectl expose创建服务**

$ kubectl run kubia --image=kubia --port=8080 --generator=run/v1

$ kubectl expose rc kubia --type=LoadBalancer --name kubia-http

$ curl 104.155.74.57:8080

**# 通过YAML描述文件来创建服务**

kubia-svc.yaml

apiVersion: v1

kind: Service

metadata:

name: kubia

spec:

ports:

* name: http

port: 80 # pod的8080端口映射成80端口

targetPort: 8080

* name: https

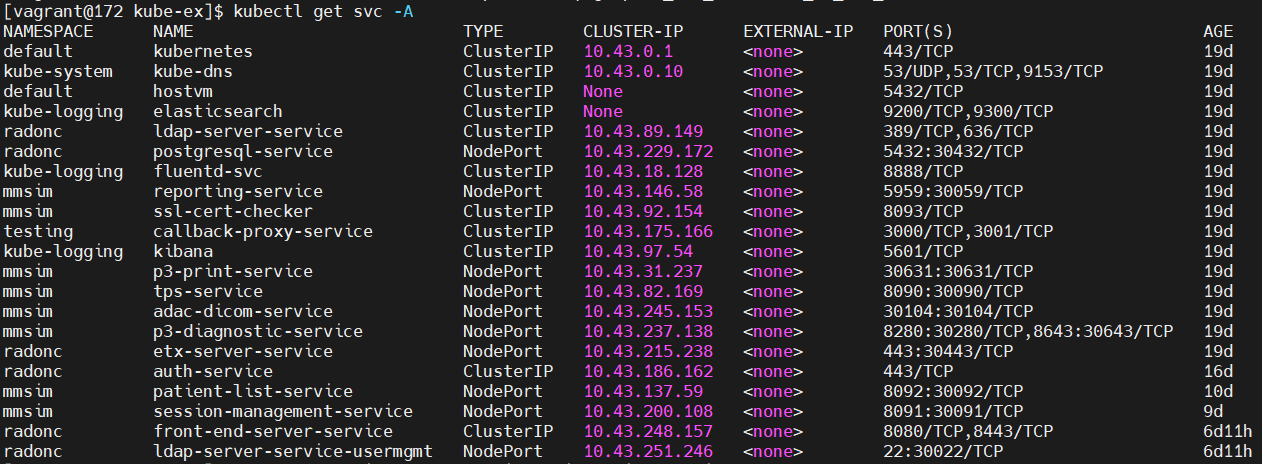
port: 443

targetPort: 8443

selector: # 具有app=kubia标签的pod都属于该服务

app: kubia

$ kubectl get svc -A # 列出服务



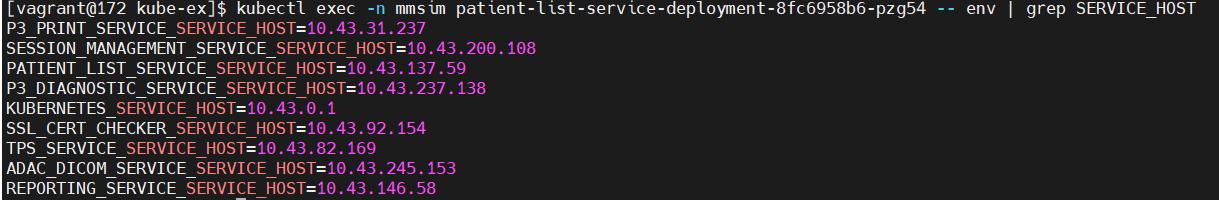
CLUSTER-IP是集群IP地址，只能在集群内部被访问。如何在内部集群测试服务？

$ kubectl exec <pod-id> -- curl -s http://<ClusterIP> # 在运行的容器中执行命令

如何发现服务？

# 通过环境变量来访问service

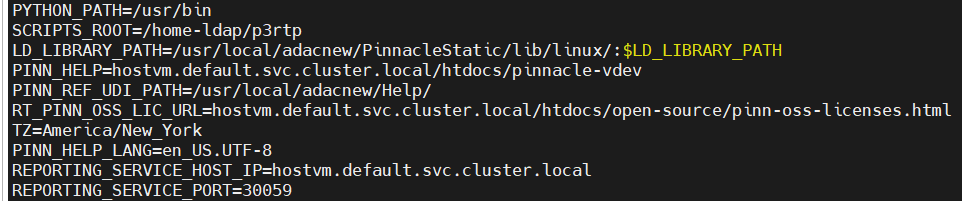
$ kubectl exec <pod> -- env | grep SERVICE\_HOST



# 通过FQDN（全限定域名）访问

每个服务从内部DNS服务器中获得一个DNS条目，客户端的pod在知道服务名称的情况下可以通过FQDN来访问

$ docker exec <tps-system-id> printenv

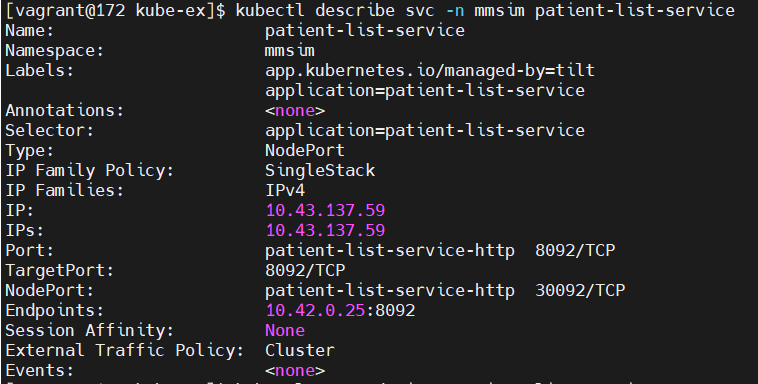


其中REPORTING\_SERVICE\_HOST\_IP是用FQDN表示

$ kubectl exec <pod-id> -- curl -s <http://hostvm.default.svc.cluster.local>

$ kubectl exec <pod-id> -- ping hostvm.default.svc.cluster.local # ping不成功，是因为服务的集群IP是一个虚拟IP,只有在与服务端口结合才有意义

$ kubectl describe svc patient-list-service -n mmsim



如何暴露服务给外部客户端？

**# 通过NodePort类型的服务**

kubia-svc-nodeport.yaml

apiVersion: v1

kind: Service

metadata:

name: kubia-nodeport

spec:

type: NodePort # 为NodePort设置服务类型

ports:

* port: 80 # 服务集群IP的端口号

targetPort: 8080 # 背后pod的目标端口号

nodePort: 30123 # 通过集群节点的30123端口可以访问该服务

selector:

app: kubia

现在客户端可以通过任何节点上的30123端口访问到你的pod

$ curl http://<node-ip>:30123

**# 通过负载均衡器暴露服务**

设置NodePort服务类型的节点若发生故障时，客户端无法再访问该服务，这就是为什么将负载均衡器放在节点前面以确保发送的请求传播到所有健康节点，并且从不将它们发送到当时处于脱机状态的节点的原因。

kubia-svc-loadbalancer.yaml

apiVersion: v1

kind: Service

metadata:

name: kubia-loadbalancer

spec:

type: LoadBalancer # 该服务从kubernets集群的基础架构获取负载平衡器

ports:

* port: 80
* targetPort: 8080

selector:

app: kubia

$ kubectl get svc -A

这时候你能看到EXTERNAL-IP 即负载均衡器的IP地址，因为你可以通过该IP地址访问服务。

$ curl http://<EXTERNAL-IP>

Web: http://<EXTERNAL-IP>

会发现curl 访问服务会碰到不同的pod, 但浏览器访问服务不会碰到不同的pod。这是因为浏览器使用keep-alive连接，并通过单个连接发送所有请求，而curl每次都会打开一个新连接。

# 通过Ingress暴露服务

每个LoadBalancer服务都需要自己的负载均衡器以及独有的公有IP地址，而Ingress只需要一个公网IP就能为许多服务提供访问。

kubia-ingress.yaml

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: kubia

spec:

rules:

* host: kubia.example.com

http:

paths:

- path: /kubia

backend:

serviceName: kubia

servicePort: 80

- path: /foo

backend:

serviceName: foo

servicePort: 80

$ kubectl get ingresses // 获取Ingress IP地址

一旦知道IP地址，通过配置DNS服务器将kubia.example.com解析为此IP地址，或者在/etc/hosts文件中添加：<Ingress IP> kubia.example.com

$ curl <http://kubia.example.com/kubia>

$ curl <http://kubia.example.com/foo>

卷：将磁盘挂载到容器

Pod类似逻辑主机，在逻辑主机中运行的进程共享诸如CPU、RAM、网络接口等资源，但pod中的每个容器都有自己独立的文件系统，因为文件来自容器镜像。

新容器并不会识别前一个容器写入文件系统内的任何内容，即使新启动的容器运行在同一个pod中。

Linux允许在文件树中的任意位置挂载文件系统，通过将相同的卷挂载到两个容器中，它们可以对相同的文件进行操作。卷被绑定到pod的lifecycle中，只有在pod存在时才会存在，但取决于卷的类型，即使在pod和卷消失之后，卷的文件也可能保持原样。

卷类型：

**emptyDir:** 用于存储临时数据的简单空目录。卷从一个空目录开始，运行在pod内的应用程序可以写入它需要的任何文件。因为卷的生存周期与pod的生存周期相关联，所以当删除pod时，卷的内容就会丢失。

Fortune-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: fortune

spec:

containers:

* image: luksa/fortune

name: html-generator

volumeMounts: // 卷html挂载在容器的/var/htdocs中

* name: html

mountPath: /var/htdocs

* image: nginx:alpine

name: web-server

volumeMounts: // 卷html挂载在容器的/usr/share/nginx/html中

* name: html

mountPath: /usr/share/nginx/html

readOnly: true

ports:

* containerPort: 80

protocol: TCP

volumes: // 名为html的emptyDir卷

* name: html

emptyDir: {}

medium: Memory // 默认是磁盘，这里设置为内存，即文件存储在内存

gitRepo: 通过检出Git仓库的内容来初始化的卷。同emptyDir卷一样，与pod同生命周期。

hostPath: 用于将目录从工作节点的文件系统挂载到pod中。因为卷的内容存储在特定节点的文件系统中，所以当数据库pod被重新安排在另一个节点时，会找不到数据。hostPath卷主要用来访问节点的日志文件、kubeconfig or CA 证书。hostPath卷通常用于尝试单节点集群中的持久化存储。

**nfs: 挂载到pod中的NFS共享卷。用于持久化存储。**

mongodb-pod-nfs.yaml

apiVersion: v1

kind: Pod

metadata:

name: mongodb

spec:

containers:

* image: mongo

name: mongodb

volumeMounts: // 卷html挂载在容器的/data/db中

* name: mongodb-data

mountPath: /data/db

ports:

* containerPort: 27017

protocol: TCP

volumes: // 名为mongodb-data的NFS卷，只需指定NFS服务器和共享路径

* name: mongodb-data

nfs:

server: 1.2.3.4

path: /some/path

$ kubectl create -f mongodb-pod-nfs.yaml

$ kubectl exec -it mongodb mongo

>use mystore

>db.foo.insert({name: ‘foo’})

>db.foo.find()

**持久卷和持久卷声明：** 从底层存储技术解耦pod。理想情况下，在kubernets上部署应用程序的开发人员不需要知道底层使用哪种存储技术，不需要了解使用哪些类型物理服务器来运行pod,与基础设施相关的交互是集群管理员独有的控制领域。当开发人员需要一定数量的持久化存储来进行应用时，可以向Kubernetes请求，就像在创建pod时可以请求CPU、内存和其他资源一样。

1. 创建持久卷PV (PersistentVolume): 集群管理员设置底层存储，指定卷大小和所支持的访问模式，然后通过kubernetes API服务器创建持久卷并注册。

mongodb-pv.yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: mongodb-pv

…

1. 创建持久卷声明PVC (PersistentVolumeClaim): 用户指定所需要的最低容量要求和访问模式，然后通过kubernetes API服务器创建持久卷声明，Kubernetes将找到可匹配的持久卷并将其绑定到持久卷声明。

mongodb-pvc.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mongodb-pvc

…

1. 在pod使用持久卷声明

mongodb-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: mongodb

spec:

containers:

* image: mongo

name: mongodb

volumeMounts: // 卷html挂载在容器的/data/db中

* name: mongodb-data

mountPath: /data/db

ports:

* containerPort: 27017

protocol: TCP

volumes:

* name: mongodb-data

persistentVolumeClaim:

claimName: mongodb-pvc

动态持久卷：仅创建PVC, 使用默认的存储类，从而将持久化存储附加到一个容器中

mongodb-pvc-dp-nostorageclass.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mongodb-pvc2

spec:

resources:

requests:

storage: 100Mi

accessModes:

* ReadWriteOnce

configMap、secret、downwardAPI: 不是用于存储数据，而是用于将kubernetes元数据公开给运行在pod中的应用程序

## 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**:

docker\_build + k8s\_yaml # a Docker image to build + Kubernetes YAML to apply

local\_resource() # a command to run locally.



Tilt’s control flow:

1. execute the Tiltfile in its entirety, and create resource definitions.
2. Whenever the Tiltfile changes, re-execute it and update the internal resource definitions.
3. the engine executes the resources and deployed Kubernetes objects to your cluster.

If it’s a local resource, run the command locally.

If Dockerfile and YAML, build image and deploy it to the cluster

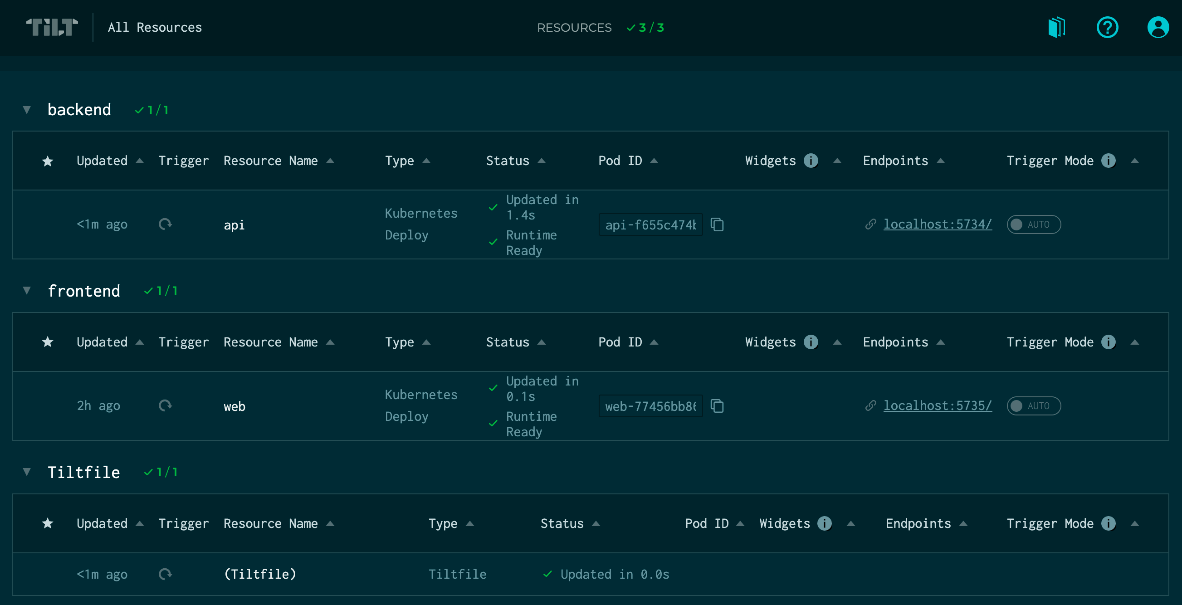
1. Lastly, resources get updated by triggering event (e.g., definition changing, a relevant file changing, or the user manually triggering the resource)

once you’ve run tilt up, you can focus on your code and let Tilt continuously react to your changes. Tilt won’t re-deploy any services that are already running and up-to-date!

* Source code file changes → sync to running container
* Dependency changes (e.g. package.json) → sync to running container and then run code in the container (e.g. npm install)
* Build spec changes (e.g. Dockerfile) → re-build container image + re-deploy
* Deployment spec changes (e.g. app.yaml) → reconcile deployment state (e.g. kubectl apply -f ...)
* Tiltfile changes → re-evaluate and create new resources, modify existing, and delete obsolete as needed

you want to know what error it caused, whether it's an error at build-time, deploy-time, or run-time. Each resource has a line in the UI that can be expanded and investigated.

**Resource Overview**



**TriggerMode: Auto** (Tilt’s default); that is, updates are automatically triggered whenever Tilt detects a change to a relevant file.

**TriggerMode: Manual**. Tilt will still monitor file changes associated with your resources, but instead of automatically rebuilding and/or deploying every time a relevant file changes, Tilt will simply indicate in the UI that files have changed, and give you a button that you can use to kick off the update.

$ kubectl describe pods <Pod ID> --namespace=<namespace>

Click <Resource Name> -> Resource Details

All Levels: Build Only // filter log for build

All Levels: Runtime Only // filter log for runtime

Same to Errors: Build Only or Runtime Only

Keyword/Regex Filter

**Resource Update**

Whenever you run tilt up or change a source file, Tilt might:

* Compile code locally on your machine (e.g. make)
* Build a container image (docker build …)
* Pushing the built image to local registry (docker push ...)
* Deploy the updated image to the cluster (kubectl apply -f …)

When you tilt up, if your services are already running and haven’t changed, Tilt won’t unnecessarily re-deploy them!

## Tilt CLI

$ tilt // help

$ tilt version

$ tilt up // run all services

$ tilt up consumer // run services in consumer

// after tilt up

$ tilt get uiresources // list all resources

$ kubectl describe pods <Pod ID> --namespace=<namespace>

$ tilt get filewatches

$ tilt describe filewatches

$ tilt logs <resource>

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

Tiltfile is a program, you can configure it with familiar constructs like loops, functions, arrays, etc

Any relative paths in your Tiltfile are evaluated relative to the location of the Tiltfile.

**// build image**

#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.

run('pip install -r /app/requirements.txt', trigger= './api/requirements.txt')]

)

when we change our project’s dependencies in ./api/requirements.txt, the updated version of the file will first be synced to the container. Then, because it matches the run step’s trigger condition, the command will be run in the container to install new/updated dependencies.

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

**custom\_build**(

'frontend',

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

['./frontend'],

)

//**local** runs a command, and returns its stdout as text

// **read\_file** to record accesses. Tilt reexecute the Tiltfile when called files change

def microservice\_yaml(name): // 定义函数to generate yaml text

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

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

// **deploy** **to Kubernetes** via k8s\_yaml

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

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

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

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

// **Configuring Kubernetes Resources**

**// same parameter for k8s\_resource, local\_resource and dc\_resource**

**k8s\_resource**( // for Kubernetes workloads

workload='frontend', # must match pod name of k8s object in k8s\_yaml

port\_forwards='9000:8000', # localhost:container

resource\_deps=['cluster-setup'], # Startup Order

labels=["database", "script"], # Grouping services in web UI

)

**// for local jobs and servers, execute an arbitrary command on your local filesystem**

**local\_resource**('yarn', cmd='yarn install', deps=['package.json'])

local\_resource('local-server',

cmd='go build ./cmd/myserver',

serve\_cmd='./myserver --port=8001',

deps=['cmd/myserver'],

allow\_parallel=True,

auto\_init=False, trigger\_mode=TRIGGER\_MODE\_MANUAL, # default is auto

readiness\_probe=probe(period\_secs=15,

http\_get=http\_get\_action(port=8001, path="/health"))

)

`serve\_cmd` starts a process that runs a server (long-running process)

`deps` called by `cmd`, should be added for tilt watch.

`probe` for local resources determine whether the serve\_cmd is considered ready. this ensures that Tilt waits for the probe to be successful before starting any dependent resources for the first time. Probes can be an HTTP GET request (http\_get\_action), a TCP socket connection (tcp\_socket\_action),

**// for docker compose services**

**dc\_resource**("my-resouce", auto\_init=False)

**// Startup Order**

// Create a K8s resource from existing objects,

// Kubernetes Object Selectors **$NAME: $KIND: $NAMESPACE** (e.g.: redis:deployment:default).

k8s\_resource(

objects=['my-ns:namespace', 'kafka:crd', 'some-ingress:ingress'],

new\_name='cluster-setup',

)

k8s\_resource('frontend',

resource\_deps=['cluster-setup']

)

If you run tilt up frontend to run only the frontend resource, that also implicitly brings up all of frontend’s transitive dependencies.

**How to determine resource ready?** By default, a resource is “ready” when:

* For k8s\_resource: the pod is running and Kubernetes considers all of its containers ready. A job is considered ready when it has completed.
* For dc\_resource: the container is started (NB: Tilt doesn’t currently observe docker-compose health checks).
* For local\_resource: the command has succeeded at least once.

We can customize the definition of readiness in local\_resource():

**// Tiltfile Config**

config settings coming from command-line args and tilt\_config.json

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

config.define\_string\_list("foo") $ tilt up -- --foo bar --foo baz

config.define\_string("foo") $ tilt invocation: tilt up -- --foo bar

config.define\_bool("foo") $ tilt invocation: tilt up -- --foo or tilt up -- --foo=False

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')

**// Many Tiltfiles and Many Repos**

* One main Tiltfile that subincludes a subdirectory Tiltfile for each service.
* Individual service Tiltfiles that include a library of common functions.
* Conditional loading of sets of services based on what you have checked out.
* Loading some services for editing and some services for running read-only.
* Loading prebuilt services from other repos.
* Checking out git repos or git submodules from the Tiltfile.

// By default, image build and deploys can run 3 at a time

**update\_settings**(max\_parallel\_updates=10)

// Loading Library Functions for services

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

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

// Loading sub-Tiltfiles

**include**('./frontend/Tiltfile')

include('./backend/Tiltfile')

// Loading Services Conditionally

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

if USE\_OAUTH2:

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

load("./extensions/git\_resource/Tiltfile", "git\_checkout")

git\_url = "ssh://tfsemea1.ta.philips.com:22/tfs/TPC\_Region22/RadOnc/\_git/{0}#integration"

tilt\_repo = "api-lib"

repo\_path = os.path.abspath("../" + tilt\_repo)

**git\_checkout**(git\_url.format(tilt\_repo), repo\_path) // git checkout from remote to local

base\_tiltfile\_path = "{0}/develop/tilt/Tiltfile"

tiltfile\_path = base\_tiltfile\_path.format(repo\_path)

include(tiltfile\_path)

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**

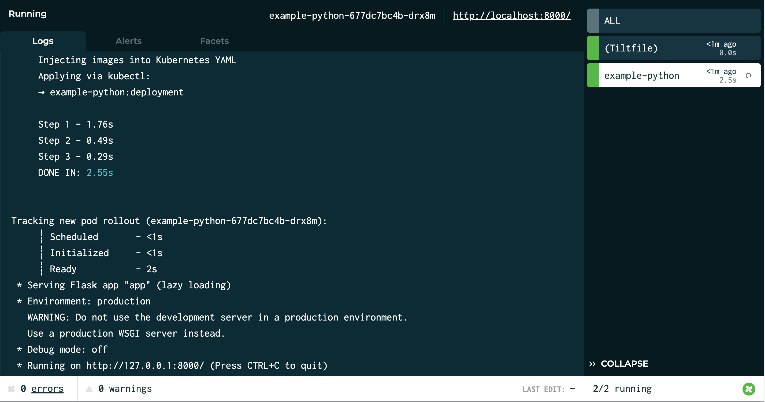
docker\_build('example-python-image', '.') // build image with name and current Dir

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.

Key concepts

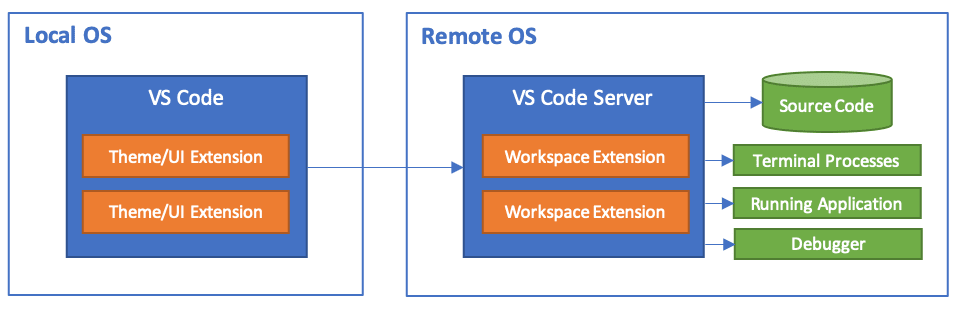
On Resources

A “resource” is a bundle of work managed by Tilt. For example: a Docker image to build + a Kubernetes YAML to apply.

Runtime Status

# 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

# CI (continuous integration) CD (Continuous Delivery)

## 理论知识

《Continuous Integration –Improving Software Quality and Reducing Risk》

《Continuous Delivery – Reliable Software Release through Build, Test and Deployment Automation》

优化软件开发过程：迭代增量式开发

CI场景步骤：

1. 开发者向版本控制库(git)提交代码。
2. CI server每过几分钟轮询版本控制库中的变更。若检测到变更，CI Server就会从库中取得最新的代码副本，执行构建脚本(Jenkins)，该脚本对构件进行集成。

构建脚本：编译源代码，集成数据库，执行测试，执行审查和部署软件

1. CI服务器向指定的项目成员发出电子邮件，提供构建结果的反馈信息。

构建脚本的逻辑过程：清理(生成的目录) -> 编译源代码 -> 集成数据库 -> 执行测试 -> 执行审查 -> 部署软件

**版本控制库：**目的是通过一个受控的访问库来管理源代码和其他软件资产（如文档）的变更。要确保构建软件需要的所有东西都放在版本控制库中。

**数据库集成：**将数据库源代码即数据库定义语言（DDL）脚本，数据库操作语言(DML)脚本，存储过程定义、分区等。存放在系统中，像其他源代码一样。例如：当项目成员（开发人员或DBA）修改了数据库脚本并提交到版本控制系统时，集成源代码的构建脚本将重建数据库和数据，作为集成构建过程的一部分。数据库脚本：删除并创建数据库和表，加入存储过程和触发器，插入测试数据

**执行测试：静态测试，Copy and Paste测试，单元测试(如**Junit, NUnit or xUnit框架)，**覆盖率测试。还有组件测试、负载/性能测试、安全测试等**。覆盖率测试可以指出哪些源代码没有对应的测试。

**自动化的代码审查：**如静态和动态分析，通过强制遵守规则来增加软件的品质。

**持续部署：**让您在任何时候都可以拿出能工作的、可部署的软件。

CI系统的主要目标是生成一组包含最新代码变更的软件工作，提供给测试环境进行测试，要做的事情包括从版本控制库中签出源代码文件、进行构建、成功执行所有测试和审查、为发布版做上标签、准备好部署文件。

CI如何与其他开发实践配合？

* 开发者测试：开发者编写测试时通常使用某种xUnit框架
* 坚持编码标准：

通过运行自动化的静态分析工具，根据已建立的编码标准审查源代码，报告标准的遵守情况。如果有违反编码标准的代码，就不让构建通过。

通过自动化的审查工具来判断是否遵守了项目的架构标准，比如利用JDepend or NDepend依赖分析工具来生成架构遵守情况的报告，来了解分层架构中数据层是否访问业务层。

* 重构和小发行版本：通过重构将重复代码放到一个方法或一个组件中。
* 共同拥有代码：避免知识孤岛。

CI实践：

* 经常提交代码: 分解任务/工作项，每一项都能在数小时内完成，进行小的变更，每天至少至少向版本控制库提交一次代码
* 不要提交无法集成的构建：
* 立即修复无法集成的构建
* 编写自动化的开发者测试
* 必须通过所有测试和审查：避免注释掉失败测试。
* 执行私有构建：先执行私有构建，即从版本控制库中取出变更，与本地变更一起执行本地构建
* 避免签出无法构建的代码

Doxygen：自动化代码文档工具，作为CI系统的一部分，自动生成源代码的文档，并创建软件的UML图模型。

集中放置软件资产：源代码、构建脚本、配置文件、工具（如应用服务器、数据库服务器和静态分析工具）、测试代码、数据库脚本/文件等所有软件资源都放在版本控制库中（如Git and Artifactory），从而减少错误的依赖关系，在换到不同的计算机上时，减少构建失败的机会

创建一致的目录结构：会使构建软件更容易

针对所有环境构建：

* 在版本控制库中维护不同的配置文件（如local.properties, integration.properties, qa.properties, production.properties）
* 将构建脚本参数化

构建类型：

* 私有构建，通过执行私有构建，您将您所做的变更与版本控制库中最新的代码进行集成
* 集成构建，向版本控制库的主线（如head or trunk）提交的变更，可以采用分阶段构建（轻量级构建少于10分钟，包括编译和单元测试，重量级构建执行较慢的测试如组件测试、系统测试和性能测试等）
* 发布构建

如何减少构建时间？可以通过分阶段构建来减少一次构建所花的时间，或者增量式构建（很少改动的DLL或共享对象库作为独立的CI项目，通过项目间的依赖关系在您的项目中引用）

构建触发机制：用户驱动（手工发起），定期执行，轮询变更，事件驱动

持续数据库集成

数据库代码(DDL,DML,配置文件等)在本质上与系统的其他源代码没有差别，对数据库源代码的改动可以触发集成构建，就像其他源代码发生变更一样。

将数据库作为一种类型的源代码，通过版本控制库进行管理

利用版本控制库共享数据库资产，包括：

* 删除、创建表和视图的DDL,包括约束条件和触发器。
* 存储过程和函数
* 实体关系图
* 针对不同环境的测试数据
* 具体的数据库配置

如果DBA不再删除表、重建表、创建测试环境、分配访问权限，那么他们可以改进数据库的性能，改进SQL的性能，数据规范化以及其他增值活动

持续测试

单元测试验证软件系统中所有小元素的行为，这些小元素通常都是一个类。单元测试的关键在于没有外部的依赖关系。真正的单元测试应该在少于1s的时间内完成。

组件测试或子系统测试验证系统的各个部分，可能需要安装安装整个系统或某些外部依赖关系，如数据库、文件系统或网络终端等。

系统测试运行整个软件系统，因此需要完整地安装系统。

功能测试是按照更像客户使用系统的方式来测试系统。功能测试正如其名，它从客户的视角来测试应用程序，这意味着测试将模仿客户的行为，这些测试也被称为验收测试。

开发者测试的一项实践是将单元测试和源代码分别放在两个目录下。将测试用例限制为一个断言。

持续审查

利用工具进行自动化的代码审查能解决80%的问题，让人来处理另外20%的重要问题。

审查与测试的区别：测试是动态的，它执行软件，目的是测试软件的功能。审查则基于一组预先定义的规则分析代码，审查者（或者静态或动态的分析工具）是由确定的标准指导指导的，团队应该坚持这些标准（通常是一些编码或设计方面的测试指标）。

利用IDE来执行本地的自动化审查是很值得推荐的。

在一些软件中大量出现的内部注释就是复杂度的一个信号。降低复杂度最有效的方法就是提取方法技术，将复杂度分解到一些更小，更可管理更可测试的方法中去。

静态代码分析工具：Java平台下的PMD, .NET平台下的FxCop and NDepend, Python的pyLint

行覆盖率：代码行被执行过

路径覆盖率：分支覆盖率

持续部署

* 为库中的资产打上标签
* 得到干净的环境，避免不当的假定
* 直接从版本控制库中取出软件资产，生成构建版并为它提供标签，然后目标机器上安装安装。
* 在生产环境的克隆环境中成功地执行所有级别的测试
* 创建构建反馈报告
* 如果需要，您可以利用版本控制库中的标签回滚构建版。

标签使我们能够追踪一组文件的历史，而不仅仅是单个文件的历史

在构建软件时，你能够自动实现每一层的删除和安装：

1)操作系统

2) 操作系统的配置如网络连接、用户和防火墙

3) 软件所需的服务器组件（如应用服务器、数据库服务器和消息服务器等）

4) 服务器配置

5) 第三方工具（如Web框架、对象-关系映射工具等

6）定制的软件如为用户编写的软件

构建版标签，通过两步完成：1）版本控制库中的代码需要一个标签；2）对代码的构建动作需要一个标签。版本控制库标签说明一组文件（通常是未编译的）是相关的。构建标签说明构建的二进制输出是唯一的。

项目经理PM决定资源的分配（人、硬件和补给）、时间和成本

架构师/技术负责人关注整个系统，品质测量的结果，例如那些静态和动态分析工具（审查工具）给出的结果，确保遵守编码标准和架构标准。

## Jenkins : automation server (build, test and deploy)

<https://www.jenkins.io/>

Jenkins run as a standalone application in its own process.

Commit Triggers Pipeline

Agent Selected based on Configured Labels

Agent Runs Build

* **Install Jenkins Server**

**Dockerfile**

FROM jenkins/jenkins:2.414.3-jdk17

USER root

RUN apt-get update && apt-get install -y lsb-release

RUN curl -fsSLo /usr/share/keyrings/docker-archive-keyring.asc \

https://download.docker.com/linux/debian/gpg

RUN echo "deb [arch=$(dpkg --print-architecture) \

signed-by=/usr/share/keyrings/docker-archive-keyring.asc] \

https://download.docker.com/linux/debian \

$(lsb\_release -cs) stable" > /etc/apt/sources.list.d/docker.list

RUN apt-get update && apt-get install -y docker-ce-cli

USER jenkins

RUN jenkins-plugin-cli --plugins "blueocean docker-workflow"

$ docker network create jenkins // Create a bridge network in Docker

$ docker build -t myjenkins-blueocean:2.414.3-1 .

$ docker run \

--name jenkins-blueocean \

--restart=on-failure \ // Always restart the container if it stops

--detach \ // Runs the current container in the background

--network jenkins \

--env DOCKER\_HOST=tcp://docker:2376 \

--env DOCKER\_CERT\_PATH=/certs/client \

--env DOCKER\_TLS\_VERIFY=1 \

--publish 8080:8080 \

--publish 50000:50000 \

--volume jenkins-data:/var/jenkins\_home \

--volume jenkins-docker-certs:/certs/client:ro \

myjenkins-blueocean:2.414.3-1

$ docker ps // check Jenkins server is running

$ docker exec -it jenkins-blueocean bash // access container

$ docker logs <container-id>

// get Administrator Password

$ docker exec <container-id> cat /var/Jenkins\_home/secrets/initialAdminPassword

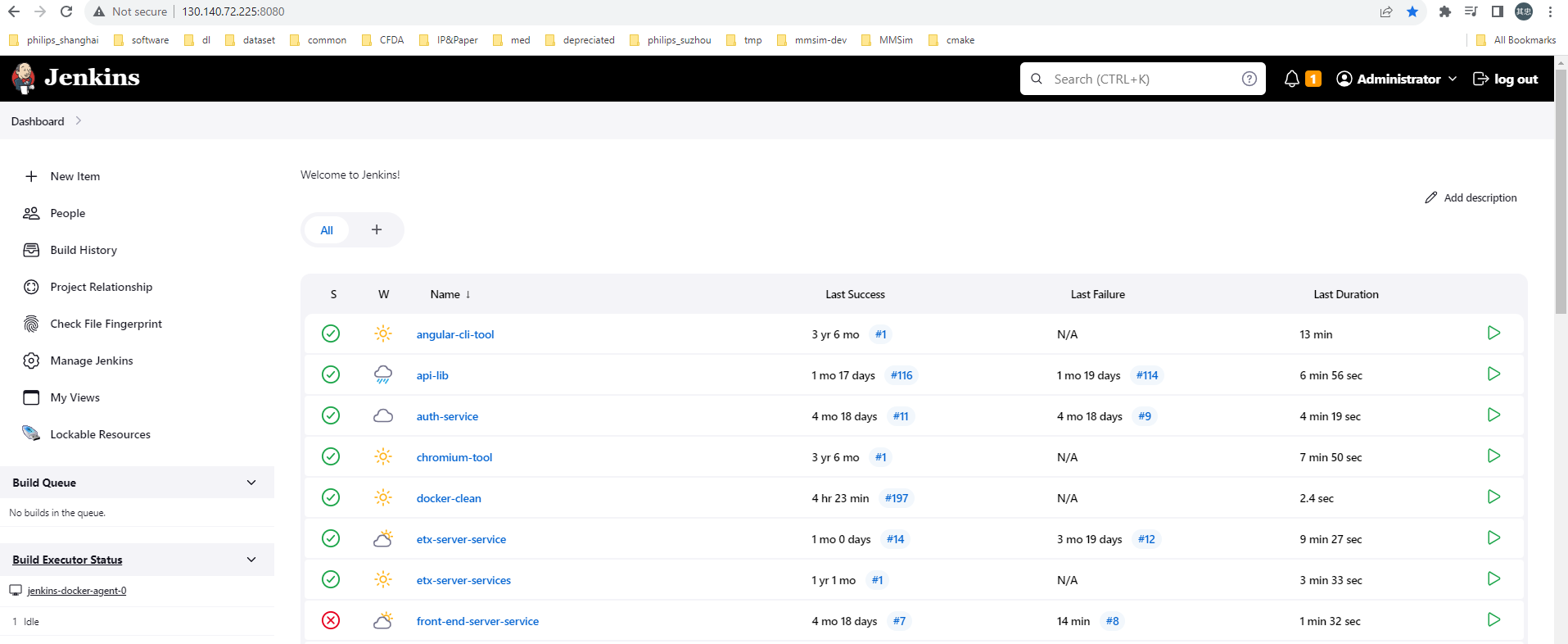
* **using Jenkins via Browser**

Browser: <http://127.0.0.1:8080>

Click `Install suggested plugins` // Customize Jenkins with plugins

Create the first administrator user

Click `Start using Jenkins`

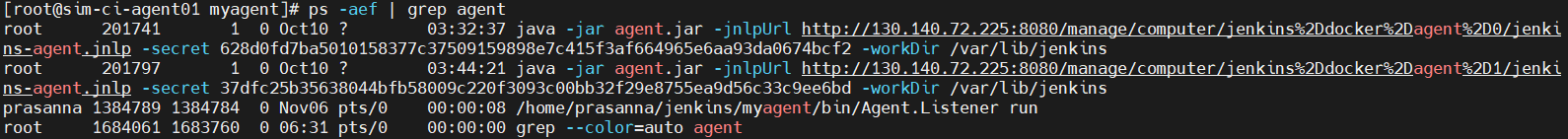


Dashboard -> Manage Jenkins -> Manage Nodes and Clouds: install Docker

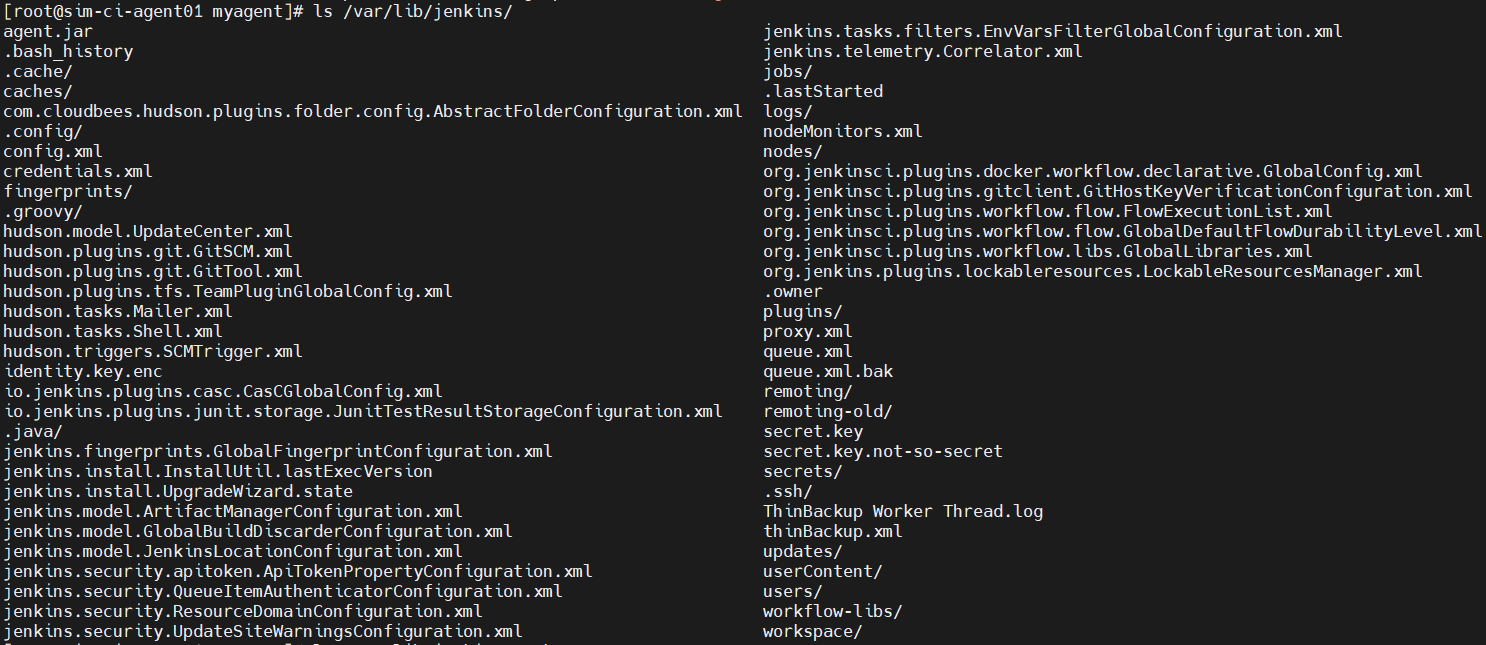
Dashboard -> Manage Jenkins -> Plugin Manager: Installed plugins

* **view work directory after Master is setup**

[root@sim-ci-agent01 myagent]# ps -aef | grep agent



# ls /var/lib/Jenkins/



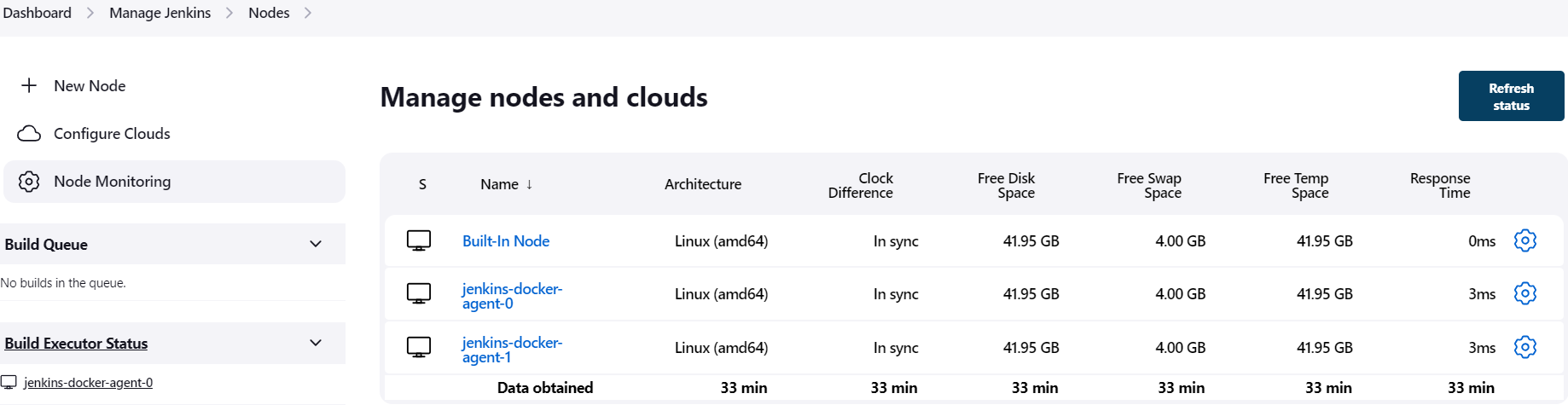
# ls /var/lib/Jenkins/workspace/

A screen shot of a computer program

Description automatically generated

* **Create Agents**

Dashboard -> Manage Jenkins -> Manage Nodes and Clouds:



// Dashboard -> New Item -> FreeStyle Project: python\_project

Configure -> General: …

->Source Code Management: <repository URL> like https://github.com/...

->Build Triggers:

->Build Environment:

->Build Steps: Execute shell

->Post-build Actions:

Execute shell:

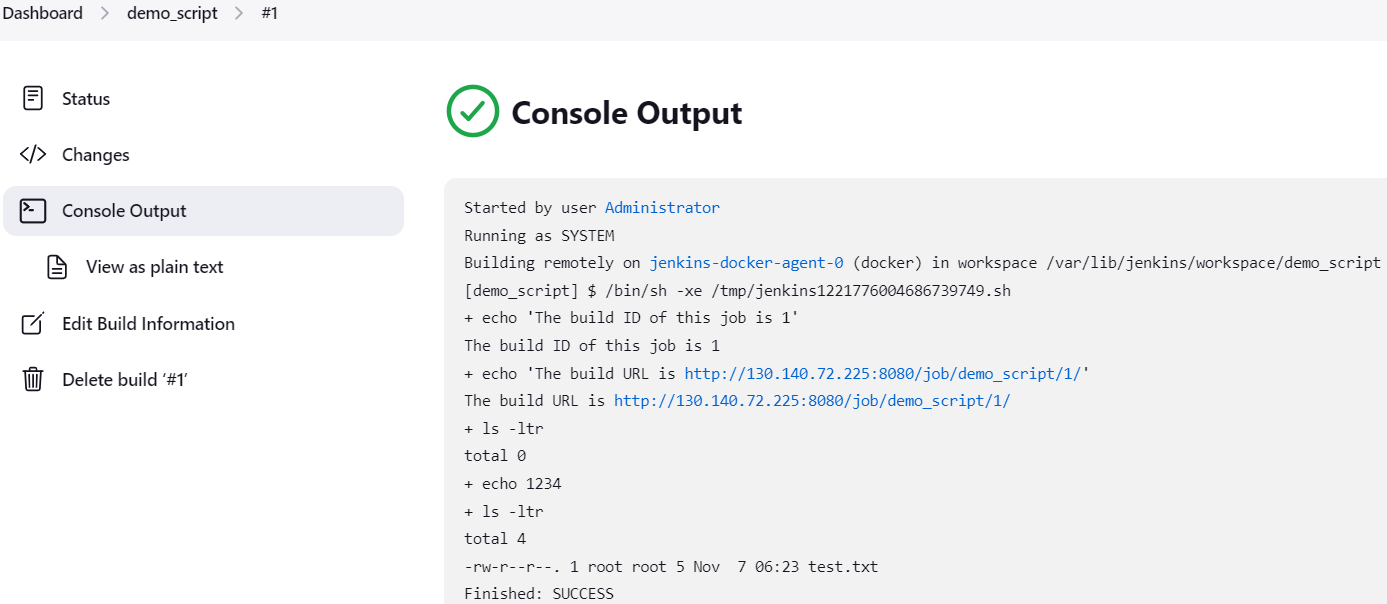
echo "The build ID of this job is ${BUILD\_ID}"

echo "The build URL is ${BUILD\_URL}"

echo "1234" > test.txt // workspace/python\_project/test.txt

python3 helloworld.py

click `Build Now`



A screenshot of a computer

Description automatically generated

// Dashboard -> New Item -> Pipeline:

Configure ->General:

->Build Triggers:

->Advanced Project Options:

->Pipeline: 1) Pipeline script, 2) Pipeline script from SCM

use Groovy Syntax and use Stages to break down Components of builds

Pipeline script:

pipeline {

agent { // select agent to let job to run on it

node {

label 'docker'

}

}

stages {

stage('Clone') {

steps {

git branch: 'master'

url: 'http://github.com/…/\*.git'

}

}

stage('Build') {

steps {

sh '''

docker build -t myawesomeapp:${BUILD\_NUMBER}

'''

}

}

stage('Test') {

steps {

sh '''

docker run -t myawesomeapp:${BUILD\_NUMBER}

curl localhost:5000

'''

}

}

stage('Package') {

steps {

sh '''

docker push -t qzlin/myawesomeapp:${BUILD\_NUMBER}

'''

}

}

stage('Deploy') {

steps {

}

}

}

}

Pipeline script from SCM:

SCM: Git

Repository URL: ssh://tfsemea1.ta.philips.com:22/tfs/TPC\_Region22/RadOnc/\_git/tps-system

Credentials: jenkinstfsgit(clone tfs git repo from Jenkins)

Branches to build: ${BRANCH}

Script Path: develop/jenkins/Jenkinsfile

Example:

tps-system/develop/Jenkins/Jenkinsfile

pipeline {

parameters { // 定义输入参数, ${params.xxxx}

string(name: 'BRANCH', defaultValue: 'integration', description: 'Branch to checkout')

booleanParam(name: 'RT\_PUBLISH', defaultValue: false, description: 'Trigger publishing …')

}

agent { // 定义pipeline执行节点

label ‘docker’ // label指定运行节点, docker控制目标节点的docker运行相关内容

}

environment { // 定义环境变量，从而Script代码中引用${env.xxxx}

RT\_COMPONENT\_NAME = 'patient-list-service'

RT\_REPO\_ID = "${params.RT\_REPO\_ID}"

}

stages {

stage(‘Cleanup’) {

steps { // 执行步骤via script, 基于groovy语法定制的DSL语言

${env. RT\_COMPONENT\_NAME}

pwd, readFile, writeFile,

echo “show …”

if (fileExists('src/main/resources/index.html')) {

} else {}

try {

} catch (exc) {

} finally {}

sh ‘’’ // On Linux, BSD, and Mac OS (Unix-like) systems

echo “execute multi shell commands”

./deploy.sh

‘’’

bat ‘’’

‘’’

dir('bin') { deleteDir() } //Recursively delete current dir from the workspace

sh 'git clean -xdff --exclude=.venv --exclude=.tools'

}

}

stage(‘Build Image’) {

steps {

script { //execute a block of scripted pipeline in declarative pip

// build image and push image to artifactory,

// ‘rt-credentials-artifactory-api-key’ is Credentials created in Jenkins

docker.withRegistry(‘<artifactory>’, ‘rt-credentials-artifactory-api-key’) {

docker.build(“”, “--file develop/jenkins/Dockerfile”)

}

}

}

}

stage(‘Bootstrap Python’) {

agent {

docker {

image "jenkins\_local/patient-list-service:"

reuseNode true

args '-u root'

}

}

}

stage(‘Parallel Stages) {

parallel {

stage(‘xxx’){}

stage(‘xxx’){}

}

}

}

post { // stage运行结束时的操作

always { //无论pipeline运行完成状态如何都会运行

}

}

}

## Example: mmsim

patient-list-service/

develop/jenkins/

Dockerfile

Jenkinsfile

**develop/jenkins/Dockerfile**

FROM radonc-docker-virtual.artifactory-ehv.ta.philips.com/radonc/tools/base-cicd-centos7:1.0.22123.62213

WORKDIR /tmp

curl --noproxy '\*' -O " https://artifactory-ehv.ta.philips.com/artifactory/radonc-generic-virtual/components/python/python-3.6.8-rt-centos7-1.0.0.0.tgz"

tar xzf python-3.6.8-rt-centos7-1.0.0.0.tgz --absolute-names

ENV PATH "/opt/rt/python-3.6.8/bin:${PATH}"

**develop/jenkins/Jenkinsfile**

${WORKSPACE}=/var/lib/jenkins/workspace/patient-list-service

Running on jenkins-docker-agent-1 in ${WORKSPACE}

**// stage('Cleanup')**

clean ${WORKSPACE}/bin

clean ${WORKSPACE}/build

${WORKSPACE}$ git clean -xdff --exclude=.venv --exclude=.tools

Removing .docker/

Removing .mypy\_cache/

Removing \_\_pycache\_\_/

Removing poetry.lock

Removing pyproject.toml

**// stage('Build Image')**

docker build -t jenkins\_local/patient-list-service:<id>

…

--file develop/jenkins/Dockerfile .

// job run inside docker container

docker run -t -d -u 0:0 -u root

-w ${WORKSPACE}

-v ${WORKSPACE}: ${WORKSPACE}

jenkins\_local/patient-list-service:<id>

**// stage('Bootstrap Python'):** installing python packages

python3.6m rt.py --no-hotstart poetry run python -V

**// stage('Copy-Paste-Detector Tests')**

\$RT\_PMD\_PATH/bin/run.sh cpd --language python --minimum-tokens 50 --files ./src

**// stage('Dynamic Tests')**

python3 rt.py dynamic-tests --run-all --xml-output build/reports/dynamic-tests.xml --verification-output-dir ${WORKSPACE}/build/verification/

**// stage('Static Tests')**

python3 rt.py static-tests --test-report True

**// stage('Build Coverity')**

cov-build --dir build/coverity\_intdir

--no-command

--fs-capture-search ${WORKSPACE}

2>&1

| tee -a build/log/build-coverity-logs.txt

cov-analyze --dir build/coverity\_intdir

--strip-path ${WORKSPACE}

2>&1

| tee -a build/log/build-cov-analyze-logs.txt

cov-commit-defects --dir build/coverity\_intdir

--host ${RT\_COVERITY\_HOST}

--https-port ${RT\_COVERITY\_HOST\_PORT}

--user ${RT\_COVERITY\_KEY\_USR} --password ${RT\_COVERITY\_KEY\_PSW}

--stream "${RT\_PROJECT}"

2>&1

| tee -a build/log/build-cov-commit-defects-logs.txt

// stage('TICS build')

**How to run CICD in tilt dev?**

[vagrant@172 patient-list-service]$ develop/tilt/run-builder-container.sh

[tilt@172 host-data]$ /opt/tics/codecheckers/linux/chk/pmd/bin/run.sh cpd --language python --minimum-tokens 50 --files ./src

[tilt@172 host-data]$ python3 rt.py static-tests

[tilt@172 host-data]$ python3 rt.py dynamic-tests

## PMD Source Code Analyzer Project

CPD : Copy Paste Detector 检测重复代码

Supported languages: [apex, cpp, cs, ecmascript, fortran, go, groovy, java, jsp, kotlin, matlab, objectivec, perl, php, plsql, python, ruby, scala, sql, swift, vf]

Download PMD zip and unzip it

C:\>pmd-bin-6.12.0\bin\cpd.bat --minimum-tokens 100 --files c:\jdk18\src\java # Windows

$ pmd-bin-6.12.0/bin/run.sh cpd --minimum-tokens 100 --files /path/to/java/code # \*nix

$ ./run.sh cpd --help

在检测重复代码时，程序会先将所有的代码文件读入内存，等待查重算法结束后将结果输出。整个过程中非常的消耗内存。

--encoding: Character encoding to use when processing files

--exclude: Files to be excluded from CPD check

--failOnViolation, -failOnViolation: By default CPD exits with status 4 if code duplications are found. Disable this option with '-failOnViolation false' to exit with 0 instead and just write the report. Default: true

--filelist: Path to a file containing a list of files to analyze.

--files: List of files and directories to process

--format: Report format. Default value is text

**--help, -h:** Print help text

--ignore-annotations: Ignore language annotations when comparing text, Default: false

--ignore-identifiers: Ignore constant and variable names when comparing text, Default: false

--ignore-literals: Ignore number values and string contents when comparing text, Default: false

--ignore-usings: Ignore using directives in C#, Default: false

**--language :** Sources code language. Default value is java

\* **--minimum-tokens:** minimum token length which should be reported as a duplicate. Default: 0

--no-skip-blocks: Do not skip code blocks marked with --skip-blocks-pattern (e.g. #if 0 until #endif)

Default: false

--non-recursive : Don't scan subdirectiories, Default: false

--skip-blocks-pattern: Pattern to find the blocks to skip. Start and End pattern separated by

|. Default is "#if 0|#endif".

Default: #if 0|#endif

--skip-duplicate-files: Ignore multiple copies of files of the same name and length in comparison, Default: false

--skip-lexical-errors: Skip files which can't be tokenized due to invalid characters instead of aborting CPD, Default: false

--uri: URI to process

Formats: [csv, csv\_with\_linecount\_per\_file, text, vs, xml]

# Source Code Management (SCM)

## Azure DevOps - Git

<https://learn.microsoft.com/en-us/azure/devops/repos/?view=azure-devops>

TFVC (Team Foundation Version Control) is a centralized version control system. You should use Git for version control in your projects and begin to move your existing TFVC projects to Git.

**Branch**: keep a history of commits and provide a way to isolate changes for a feature or a bug fix from your main branch

**Branch policies**: Isolate work in progress from the completed work in your main branch.

**Clone**: Cloning a repo downloads all commits and branches in the repo.

**Commit**: A commit is a group of changes saved to your local repository.

**Fork**: The new fork acts as if someone cloned the original repository and then pushed to a new, empty repository. use pull requests to push the changes back to the original repository.

e.g., fork open source project. Typically, only core contributors to your project should have direct commit rights to your original repo. Other collaborators should use a forking workflow to isolate their proposed changes until the core contributors have a chance to review their work.

**Pull request**: let your team review code and give feedback on changes before you merge it into the main branch.

**Fetch**: downloads remote commits that aren't in your local branch, but doesn't merge them.

**Pull**: updates the code in your local repository with the changes from other members of your team that are in the remote repository.

**Push**: When you push, Git uploads the saved commits in your checked branch to the remote repository. If the branch exists on the remote repository, Git takes the commits and adds them to that branch on the remote repository. If that branch doesn't exist, Git creates a new branch with the same commits as your local branch.

**Sync**: performs a Pull then a Push.

Azure DevOps

1. **create organization and project.**

e.g., https://tfsemea1.ta.philips.com/tfs/**TPC\_Region22/RadOnc**

1. **Install Git command-line tools for Windows or Linux and configure SSH authentication.**

Personal access tokens: The credential manager creates the token in Azure DevOps and saves it locally for use with the Git command line or other client.

SSH key authentication: 1) create public and private key pair on your computer, 2) just add a public key to Azure DevOps and clone your repos using SSH.

1. **clone the repo to your computer.**

* If You don't have any code yet, first Create a new Git repo in your project.
* If the code is in another Git repo, import it into a new or existing empty Git repo.
* If the code is on your local computer and not yet in version control, either create a new Git repo in your project or add your code to an existing repository.

Branch Policies enforce your team's code quality and change management standards.

Integration branch -> branch policies: min reviewers, linked work items, …

<https://learn.microsoft.com/en-us/azure/devops/repos/git/branch-policies?view=azure-devops&tabs=browser>

$ git clone <repo>

$ cd <repo>

1. **Work in a branch**

If you're working with a previously cloned repository, ensure that you've checked out the right branch and that it's up to date before you create your new branch. the default branch was almost always named master/main/trunk/development.

$ git checkout main // ensure that you've checked out the right branch

$ git pull origin main // ensure that it's up to date

$ git checkout -b features/cool-feature // create and switch to a branch

$ git branch features/cool-feature // create a branch

$ git checkout features/cool-feature // switch to a new branch

Edit the source code.

Git tracks changes to all tracked files, Git separates tracked files into three categories: 1) Unmodified files - The files you haven't changed since your last commit. 2) Modified files - The files you've changed since your last commit but haven't staged for the next commit. 3) Staged files - The files you've changed since your last commit and staged for the next commit.

A screen shot of a computer screen

Description automatically generated

Commits are created in your local Git repo. Each commit doesn't have to be perfect, and it might take several commits to accomplish an intended change.

$ git add .

$ git commit -m “…”

$ git push origin features/cool-feature // Push your branch to the remote repository

// 回滚至某一commit版本

$ git log -10 //显示近10次提交

$ git reset --soft <commit id> //取消近几次提交，回到<commit id>提交之前git add之后

$ git restore --staged . // 取消git add操作（将暂存区文件退回）

$ git restore . // discard changes到 git add 之前

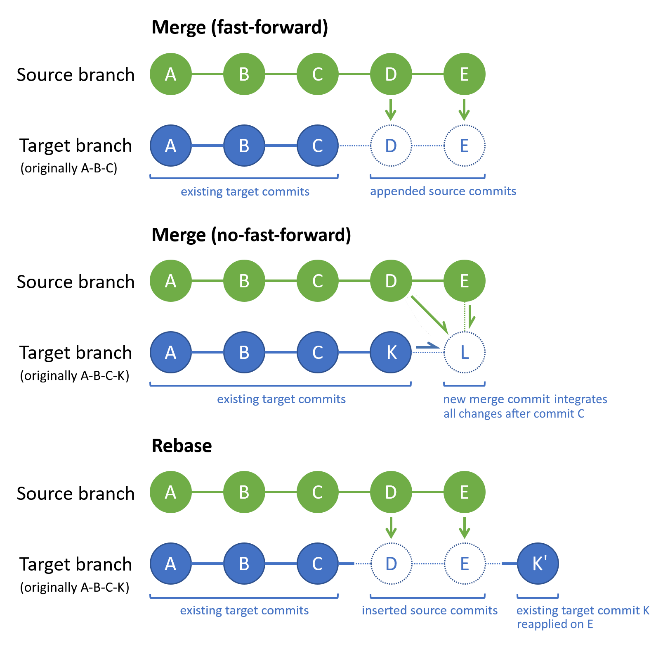
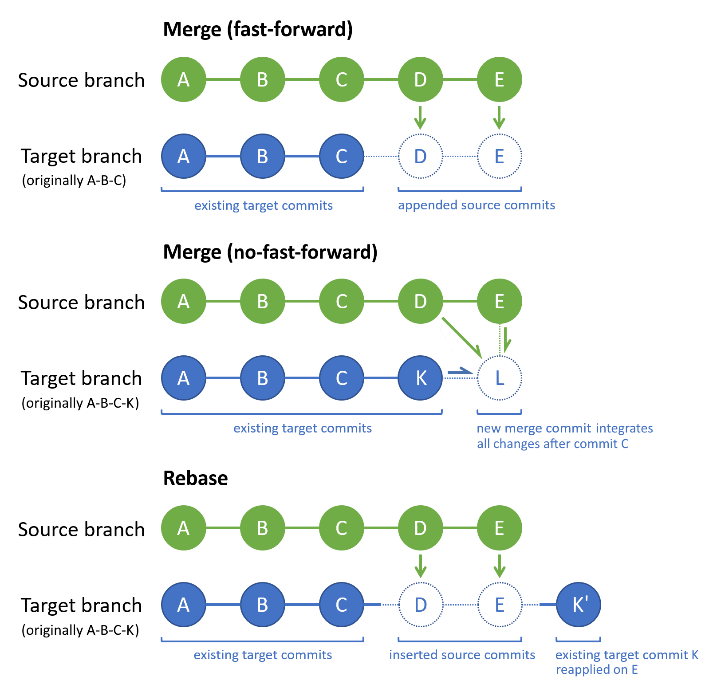
// 若之前不小心提交含大文件的commit, 尽管回滚到提交之前的版本，但git是按提交差异化存储，git push会把大文件的变动记录，如何去除大文件的变动记录？可以squash last X commits together in Git

$ git rebase -i HEAD~3 // squash last 3 commits,

HEAD是当前分支的最新版本，HEAD~3指倒数第三个版本

1. **Update feature branch with latest integration**

<https://learn.microsoft.com/en-us/azure/devops/repos/git/merging?view=azure-devops&tabs=visual-studio-2022>

A diagram of a branch

Description automatically generated

the default merge type will be a fast-forward merge. Git merge only modifies the target branch—the source branch remains unchanged.

While you're working on your feature branch, it's a good idea to periodically switch to your main branch and pull new commits to keep it current with the remote main branch.

**Git rebase only modifies the target branch—the source branch remains unchanged**. When you encounter one or more merge conflicts, you must resolve them to complete the rebase. If you're the only person working on your feature or bugfix branch, consider using Git rebase

$ git checkout main

$ git pull // latest integration

$ git checkout features/cool-feature

$ git rebase main //feature branch based on latest integration

If conflicts, resolve the conflicts with VS code. Then

$ git add .

$ git commit -m “solve rebase main”

$ git rebase --continue

1. **Merge your changes with a pull request.**

Pull requests combine the review and merge of your code into a single collaborative process. After you’re done fixing a bug or new feature in a branch, create a new pull request. Add the members of the team to the pull request so they can review and vote on your changes.

Provide enough detail in the description to quickly bring reviewers up to speed with your changes.

Repos > Files -> You updated features/cool-feature just now -- Create a pull request: //dialog

features/cool-feature into main

Title, Description, add Reviewers, link Work Items

review the files included in the pull request at the bottom of the New Pull Request window.

->create -> complete: // dialog

Title: Merged PR …,

Description: xxxx

Select `Delete features/cool-feature after merging` -> complete merge

1. **View history and clean up**

The code in your main branch should pass tests, build cleanly, and always be current.

Repos > Files > History > click one commit, to view your changes

$ git checkout main // switch to main branch

$ git pull origin main // pull down merged main branch

$ git branch -d features/cool-feature // delete local copy of branch

1. **get branch from remote**

$ git branch -r // 显示远程仓库分支

$ git fetch // 更新本地仓库（从远程仓库）

$ git switch features/cool-feature // 本地切换到远程仓库分支

**.gitignore**

Not every file in your project should be tracked by Git. E.g., binaries, logs, tracing output or diagnostic data from your builds and tests.

Each line in a .gitignore file specifies a file search pattern relative to the .gitignore file path.

you can add a .gitignore file to any project folder to let Git know which files to ignore within that folder and its subfolders at any nested depth.

$ git update-index --skip-worktree <file path> // temporarily stop tracking a local repo file

$ git update-index --no-skip-worktree <file path> // resume tracking

To permanently remove a file from the Git snapshot so that Git no longer tracks it, but without deleting it from the filesystem.

$ git rm --cached <file path>

$ git commit <some message>

**Name and Email**

When you commit to your local repo, Git includes your name and email address as part of the commit.

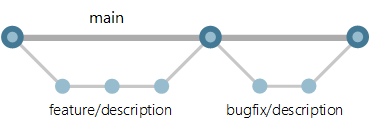
When you set up Git for the first time on a new machine, you may create a commit using Git's "best effort" information. Git will prompt you to set an explicit name and email address, and the new settings will be used for subsequent commits.

$ git config --global user.name "Frances Totten" // change name and email

$ git config --global user.email "frances\_t@fabrikam.com"

Use feature branches for your work

Develop your features and fix bugs in feature branches based off your main branch



Some suggestions for naming your feature branches:

* users/username/description
* users/username/workitem
* bugfix/description
* feature/feature-name
* feature/feature-area/feature-name
* hotfix/description

## Git command

Books:

《GitHub 入门》

《Git版本控制管理》

《精通Git》

版本控制是一套系统，该系统按时间顺序记录某一个或一系列文件的变更，让你可以查看其以前的特定版本。**DVCS客户端**并非仅仅是检出文件的最新快照，而是对代码仓库进行完整的镜像。

开发以及维护开发出来的代码，方便读取代码的历史版本、记录所有的修改。

分布式开发：允许并行开发，各人可以在自己的版本库中独立且同时地开发，而不需要与一个中心版本库时刻同步（因为这样会造成开发瓶颈）

CVS给予每个开发人员对于自己的私有版本写的权限，因此，不同开发人员的改动可以自动合并，除非两个开发人员尝试修改同一行。**如果出现修改同一行的情况，那这一行将会作为“冲突”被标记出来，由开发人员手动去解决。**

Git版本库只是一个简单的数据库，其中包含所有用来维护与管理项目的修订版本和历史信息。Git在每个版本库里维护一组配置值。版本库的用户和email地址。Git对每个网站、每个用户和每个版本库的配置和设置信息都进行管理和检查。在版本库中，Git维护两个主要的数据结构：**对象库(boject store)和索引(index)**.所有这些版本库数据存放在工作目录根目录下一个名为.git的隐藏子目录中。

对象库在复制操作的时候能进行有效复制，这也是用来支持完全分布式VCS的一种技术。对象库是Git版本库实现的心脏。它包含你的原始数据文件和所有日志消息、作者信息、日期，以及其化用来重建项目任意版本或分支的信息。

索引是暂时的信息，对版本库来说是私有的，并且可以在需要的时候按需求进行创建和修改。

在互联网上，文件或者任意大小的blob都可以通过仅比较它们的SHA1标识符来判断是否相同

**Git仅根据文件内容来计算每一个文件的散列码**，如果文件有相同的SHA1值，它们的内容就是相同的，然后将这个blob对象放到对象库里，并以SHA1值作为索引。如果这些文件中的一个发生了变化，Git会为它计算一个新SHA1值，识别出它现在是一个不同的blob对象，然后把这个新的blob加到对象库里。原来的blob在对象库里保持不变，为没有变化的文件所使用。

Git把文件名视为一段区别于文件内容的数据。Git仅仅记录每个路径名，并且确保能通过它的内容精确地重建文件和目录，这些是由散列值来索引的

Git使用打包文件(pack file)的更有效的存储机制。要创建一个打包文件，Git首先定位内容非常相似的全部文件，然后为它们之一存储整个内容。之后计算相似文件之间的差异并且**只存储差异**。

SHA1：安全散列加密算法，3b1..., 160位，20个字节 = 40个字节的十六进制来显示

散列函数：在数学意义上是一个真正的函数，对于一个给定的输入，它总产生相同的输出。这样的散列函数有时也称为摘要，用来强调它就像散列对象的摘要一样。应用：提供了一种有效的方法来比较两个对象，甚至是两个非常大而复杂的数据结构，而且并不需要完全传输。

* 工作区：指工作目录和能看到的文件（受版本控制和不受版本控制）
* 暂存区：工作区的新文件和已经修改的受版本控制的文件，通过git add file添加到暂存区（内部机制：git add file生成对应的object对象，放在.git/objects目录下，状态变成了staged。
* 版本库：通过git commit，引用.git/objects，将提交文件状态变成committed (也是unmodified状态)，从而修改会提交到版本库中。版本库记录每一次提交，追溯每一次修改内容。

文件状态：

* 不受版本控制的untracked状态
* 受版本控制并且已修改的modified状态
* 受版本控制已修改并提交到暂存区的staged状态
* 从暂存区已经提交到本地仓库的committed状态
* 提交到本地仓库未修改或者从远程仓库克隆下来的unmodified状态

1. Git配置

Git是作为一套简单的、独特的、独立的命令提供的，并按照“UNIX"工具包的哲学来开发的：打造小的、可互操作的工具。

GNU风格：git commit -m "..." 等价于 git commit --message="..."

$ git //列出选项和最常用的子命令

$ git help –all

$ git <verb> --help

//git config获取和设置配置变量，每一级会覆盖上一级中的变量

$ git config --system … // 修改**所有用户的配置文件: /etc/gitconfig**

$ git config --global … // 修改**个人用户的配置文件: ~/.gitconfig**

$ cat .git/config // **当前仓库的配置文件.git/config**

//列出所有的配置参数，在当前仓库会列出三级变量

$ git config --list

// git每一次提交都会用到用户名和邮箱

$ git config --global user.name "QiZhong Lin" //配置git用户名和邮箱

$ git config --global user.email "qizhong.liin@philips.com"

$ git config --global https.proxy <http://185.46.212.34:10015> //设置代理

$ git config --global http.proxy <http://185.46.212.34:10015>

$ git config --global http.proxy 165.225.96.34:10015 // Or ENV HTTP\_PROXY

$git config --global core.editor emacs //修改编辑器

$ git config --unset --global user.email //移除设置

$ git config --global --unset-all http.proxy

$ find .git/objects //查找指定目录下的文件

1. 建立Git版本库

// Git created in remote repos

register username/password

new repository -> repository name -> create repository -> <https://github.com/qizhonglin/doc.git>

$ git clone https://github.com/qizhonglin/doc.git

$ cd doc

// Git created locally

$ mkdir doc && cd doc

doc$ git init //将目录转化为Git版本库

为了显示目录是一个Git版本库，git init命令创建了一个隐藏目录，在项目的顶层目录，名为.git, 而CVS和SVN则将修订版信息放在项目的每一个目录下的CVS和.svn子目录里，Git把所有修订信息都放在这唯一的顶层.git目录里。

// 添加文件进行版本控制

doc$ git add \*.c //添加到暂存区

doc$ git add LICENSE

doc $ git add . //把当前目录及子目录中的文件都添加到暂存区

doc$ git commit -m "initial project version" // 提交到版本库

//把当前目录及子目录中的文件都添加到暂存区，然后提交到版本库

doc$ git commit -a -m “update…” //等价git add . and git commit

**回退操作**

<https://blog.csdn.net/albertsh/article/details/104719370>

git checkout：检出

$ git checkout branch\_name切换分支操作，实际上是把指定分支在仓库中对应的所有文件检出来覆盖当前工作区，最终表现为切换分支

$ git checkout – file\_name当不指定commit id就是将暂存区的内容恢复到工作区，从而达到回退本地修改的作用

git revert: 归还

$ git revert 对某次提交（即修改）执行归还操作，其实就是反向再修改一次

举例：当我们在一个文件中添加一行内容，并提交到版本库后，产生一个commit id, 如果$ git revert <commit id>, 就相当于在工作区中的那个文件将刚在新加的一行内容删除掉，然后再进行一次提交

git restore: 还原工作区的文件

用来分担git checkout功能，将暂存区或版本库中的文件覆盖本地文件的修改，可以达到回退修改的目的。同时也可以使用版本库中的文件覆盖暂存区的文件，达到回退git add 命令的目的

git reset: 设置分支的头部指向

当进行一系列提交后，发现最近几次提交有问题，想从提交记录中删除

$ git reset <commit id> 表示当前分支回退到这个<commit id>对应的状态，之后的记录被删除，工作区的文件状态根据参数的不同恢复到不同的状态

$ git reset --soft <commit id> //被回退的修改放回在暂存区，可以再次提交

$ git reset --mixed <commit id> //（默认）放回在工作目录，可以先回到暂存区再提交

$ git reset --hard <commit id> // 被回退的修改直接舍弃

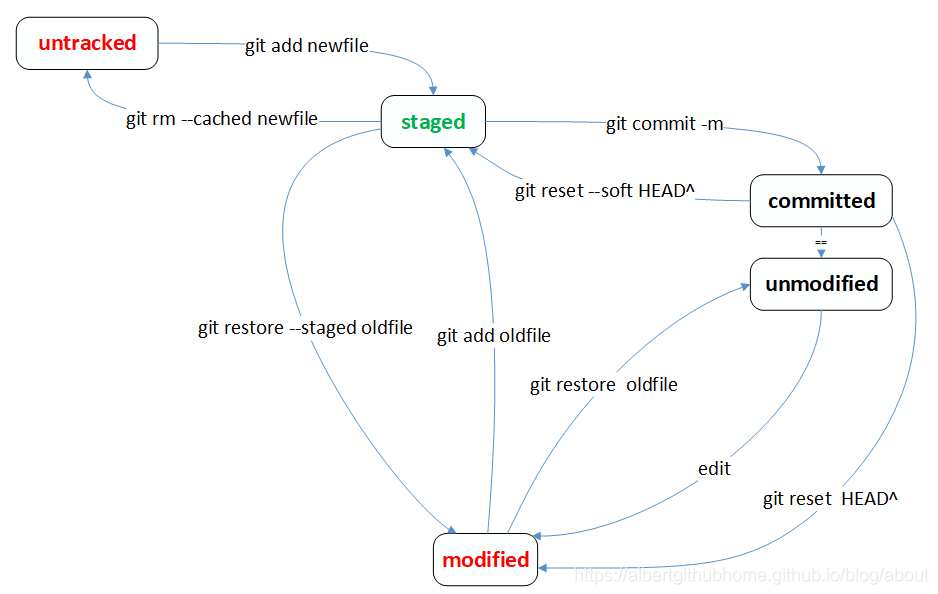
git rm: 删除

如果它和某些命令反着来就是一种回退

$ git add newfile\_name //将文件存在暂存区

$ git rm --cached newfile\_name //将文件从暂存区移除掉，但在工作区还在

$ git rm newfile\_name //从暂存区和工作区都删除，相当于



//还原00: 工作中未加到暂存区和版本库的文件，还原今天所做的修改

未添加到暂存区就没有被追踪，任何修改没办法回退

//还原01: 工作区中未加到暂存区和版本库的文件，执行了git add

$ git add new.txt

$ git restore --staged new.txt //灰色为等价方法

$ git rm --cached new.txt

$ git reset HEAD new.txt

//还原02: 版本库的文件，修改或删除后未执行git add

$ git restore README.md

$ git checkout -- README.md

$ git reset --hard HEAD

//还原03: 版本库的文件，修改或删除后执行了git add

$ git add README.md //使用git add之后，文件的改变就放到了暂存区

$ git restore --staged README.md //文件状态退回到<还原02>

$ git reset HEAD README.md

//还原04: 版本库的文件，修改或删除后执行了git add, git commit

$ git reset HEAD^ //取消最后一次提交记录,文件状态回到<还原02>

$ git reset --soft HEAD^ //取消最后一次提交记录,文件状态回到<还原03>

HEAD^表示最新版本前一版，即倒数第二版本，HEAD^^倒数第三版本, …

HEAD~1表示最新版本前一版，即倒数第二版本，HEAD~2倒数第三版本，…

git reset命令是一个重置HEAD的命令，要求版本库指向一个合法提交

//还原05: 版本库中的文件，修改回删除后执行了git add, git commit, git push

先按还原04处理，再将本地分支push到远程分支

$ git reset HEAD^

…

$ git push -f //git push不成功，因为本地回退，落后于远程

//还原06: 两次git commit之后产生两条日志，只还原第一次提交

$ git revert HEAD^

解决冲突后提交

$ git reset HEAD^^ //还原两次提交，修改工作区文件，再提交

// 删除文件和添加操作一样，删除操作也需要两步：

$ git rm poem.html // 表示你想要删除这个文件的意图并暂存这个变更

$ git commit -m "Remove a poem" // 在版本库里实现这个变更

// 把文件保留在工作目录，但从暂存区中移除该文件，即将文件保留在硬盘上，但不想让Git对其进行跟踪管理

$ git rm --cached poem.html

// 文件改名

$ git mv foo.html bar.html // 等价如下

$ mv foo.html bar.html

$ git rm foo.html

$ git add bar.html

$ git commit -m "Moved foo to bar"

// 撤销已暂存的文件

$ git add \* // 所有文件添加到暂存区

$ git status //会猜你接下来要做的事情

On branch master

Changes to be committed:

(usr "git reset HEAD <file>…" to unstage)

$ git reset HEAD foo.html // 将该文件移动暂存区

// 撤销对文件的修改(还未add到暂存区)，建议用储藏stash or 分支

edit foo.html

$ git checkout -- foo.html //放弃对文件的修改

git stash

保存当前工作目录的临时状态，包括暂存区和已修改但未暂存的文件。它会会将这些修改保存在一个临时区域（即“堆栈”）中，让你能够回到一个干净的工作目录，可以进行其他操作。等到你完成其他任务后，可以再回到之前的状态，继续之前的开发。

使用场景：

* 切换分支： 当你正在开发一个功能或修复一个bug，但需要切换到另一个分支来处理其他任务时，使用git stash可以将当前的修改保存起来。这样你可以切换到其他分支并开始另一个任务，而无需提交或放弃你当前的修改。
* 合并代码： 在进行代码合并操作之前，你可能需要切换到目标分支并更新代码。使用git stash可以保存当前分支的修改，然后切换到目标分支并执行更新操作。完成后，你可以切换回原分支，并使用git stash pop来恢复之前的修改
* 临时修复问题： 如果你遇到一个紧急的问题，需要快速切换到其他分支进行修复，但又不想丢失当前的修改，可以使用git stash将修改保存起来。然后你可以切换到修复分支，并在修复完成后再回到原分支恢复之前的修改。
* 多任务处理： 在开发过程中，你可能会同时处理多个任务或功能。当你想切换到另一个任务时，可以使用git stash将当前任务的修改保存起来，然后切换到另一个任务并开始工作。完成后，你可以回到之前的任务并使用git stash pop来恢复修改。

$ git stash list // 查看当前保存的所有stash列表

$ git stash save "message" //保存当前的工作目录状态到一个新的stash

$ git stash --all // 保存所有的修改，包括暂存区和未暂存的修改，以及未跟踪的文件。

$ git stash show [stash] // 查看某个特定stash的变更内容

$ git stash apply [stash] // 恢复stash （不会从stash列表中移除）

$ git stash pop [stash] // 恢复stash （会从stash列表中移除）

$ git stash drop [stash] // 从stash列表中移除

$ git stash clear // 删除所有的stash

$ git stash branch <branch\_name> [stash] // 基于stash创建新分支,从而干净环境开发

// 查看历史提交

$ git log //如果很长，按q退出

$ git log -p -2 //显示每次提交的差异，-2显示最近两次提交

$ git log --stat //显示每次提交中被更改的文件的统计信息

$ git log --pretty=oneline //每行提交一个提交

$ git log --since=2.weeks //最近两周的提交

$ git show 9da... //查看特定提交的详细信息, 若无提交码，只显示最近一次

// 远程仓库

$ git clone ssh://….

$ git remote -v //显示远程仓库

origin ssh://…/\_git/simulation-development-builder (fetch)

origin ssh://…/\_git/simulation-development-builder (push)

$ git remote add pb ssh://… //添加远程仓库,并命名为pb

$ git remote rename pb paul //重命名远程仓库

$ git remote rm paul //由于迁移服务器地址，需要删除某个远程仓库地址

$ git fetch // 更新本地仓库（从远程仓库）

$ git pull // 获取远程分支新数据，并尝试合并入当前本地分支

// 当在本地某个分支执行git push时，会推送到远程的哪个对应分支上去

// 如果别人和你都clone这个仓库，而他先推送，你后推送，那么你的这次推送会直接被拒绝，你必须先拉取别人的变量，将其整合到你的工作成果中，然后才推送

$ git push [remote-name] [branch-name]

1. branch

doc (master) $ git branch // list local branch

doc (master) $ git branch -r // list remote branch

doc (master) $ git branch -a // list local and remote branch

doc (master) $ git branch feature // create new branch

doc (master) $ git checkout feature // switch to new branch in local repo

doc (feature) $ git reflog // action log of current repository

CRUD files inside GitHub/doc

doc (feature) $ git status

$ git show-branch --more=10 //显示当前开发分支简洁的单行摘要

$ git diff

// check in, git push -u origin master

doc (feature) $ git push

# 合并分支

doc (feature) $ git checkout master # switch to master branch

doc (master) $ git merge feature // merge

如果branch feature and master 有冲突（即可能修改同一个文件），git无法自动合并，需要手动处理冲突，按i编辑提交信息，:wq!保存并退出页面

doc (master) $ git status # 查看冲突信息

查看造成冲突的文件，修改冲突部分，比如\*.txt

doc (master) $ git add \*.txt # 对修改后冲突文件

doc (master) $ git commit -m ‘fix conflict’ # 创建一个修改冲突的提交。

实际上master一般作为一个保护分支，是不允许被直接提交和推送的，这种情况，我们可以会去到对应的git管理页面（比如gitlab或者github），通过创建PR的方式，来创建合并请求，并在通过人工审核后进行合并。这样就少了需要在本地进行本地分支合并的过程。

# 分支回滚

GitHub/doc (master) $ git reset --hard fd0cbf0 // reset to selected version, fd0... is commit id

# 分支删除

doc (master) $ git branch -d feature # 提醒删除

doc (master) $ git branch -D feature # 强制删除

doc (master) $ git push origin --delete feature # 远端删除

// check out, git pull origin master

doc (master) $ git pull

1. 回退指定版本

doc (master) $ git tag -l #检查历史版本

doc (master) $ git checkout tags/v2.7.1 #回退2.7.1版本

…

GitHub/doc (master) $ git branch #查看本地版本 估计含最新版本

GitHub/doc (master) $ git checkout develop #选择最新版本

1. delete repository in Github

进入指定repository -> Settings -> delete repository

1. push existing Git project to Github

在Github先创建repository: https://github.com/qizhonglin/ionic-blank-ex0.git

$git remote add origin https://github.com/qizhonglin/ionic-blank-ex0.git

$git push origin master

1. Git pull 强制覆盖本地文件

git fetch --all

git reset --hard origin/master

1. ignore files, only submit selected files

# create file .gitignore

\_\_pycache\_\_/ ignore any directory called \_\_pycache\_\_/ in any subdirectory

build/

1. if the repository already exists, do the following:

git rm -r --cached .

git add .

git commit -m "..."

git push

Tool: GitHub Desktop

a. Install GitHub Desktop

username and password

b. Autenticate GitHub

if behind a proxy server,

git config --global http.proxy <http://ip:port>

Example:

**Add features to omnilearn**

1. git clone <https://github.com/philips-internal/omnilearn.git>

Note: set git proxy, but unset script proxy

$ git config --global http.proxy <http://185.46.212.34:10015>

$ git config --global https.proxy <https://185.46.212.34:10015>

User: [qizhonglin](mailto:qizhong.lin@philips.com)

Password: ghp\_gOxpDVE1me3FVCK96zRmKWHdD9paHS1tNPkt

1. write issue in omnilearn platform

issue/#650\_GetSpatialDistributionPlots

1. create branch

omnilearn$ git branch -r # list remote branch

omnilearn$ git branch # list local branch

omnilearn$ git checkout develop # switched to branch “develop”

omnilearn$ git checkout -b feature/#650\_GetSpatialDistributionPlots

... # add code

omnilearn$ git add . / git commit -m “…”

omnilearn$ git push --set-upstream origin features/#650\_GetSpatialDistributionPlots

if not access to write, apply to Tanja or Alex

# 分支删除

omnilearn$ git branch -d feature/#650\_GetSpatialDistributionPlots # 提醒删除

omnilearn$ git branch -D feature/#650\_GetSpatialDistributionPlots # 强制删除

omnilearn$ git push origin --delete feature/#650\_GetSpatialDistributionPlots # 远端删除

1. create PR (pull request)

assign reviewer: Tanja

1. refine code and description according to reviewer’s comments until resolve conversion

同时开发两个项目omnilearn和Trauma

* 下载omnilearn and trauma
* Trauma引用指定版本omnilearn作为虚拟环境
* Trauma引用开发版本omnilearn作为虚拟环境 (一般假定开发版omnilearn已更新)

进入Trauma的虚拟环境，并且路径为开发版omnilearn,即如下：

**(Trauma的虚拟环境) qzlin@qzlin:/.../开发版omnilearn$** python setup.py install

#从而更新虚拟环境为开发版omniearn

* Trauma引用开发版本omnilearn的某指定版本作为虚拟环境

qzlin@qzlin:/.../开发版omnilearn$ git tag -l #查看历史版本

qzlin@qzlin:/.../开发版omnilearn$ git checkout tags/v2.7.1 #回退历史版本

**(Trauma的虚拟环境) qzlin@qzlin:/.../开发版omnilearn$** python setup.py install

#从而更新虚拟环境为历史omniearn

FAQ:

how to solve the slow access of github?

http://blog.csdn.net/wu\_\_di/article/details/50538916

access http://github.global.ssl.fastly.net.ipaddress.com/#ipinfo to get the fast IP for github.global.ssl.fastly.net, then search fast IP for github.com

$sudo gedit /etc/hosts

192.168.xx.xx github.com

185.31.17.xx github.global.ssl.fastly.net

then update DNS cache

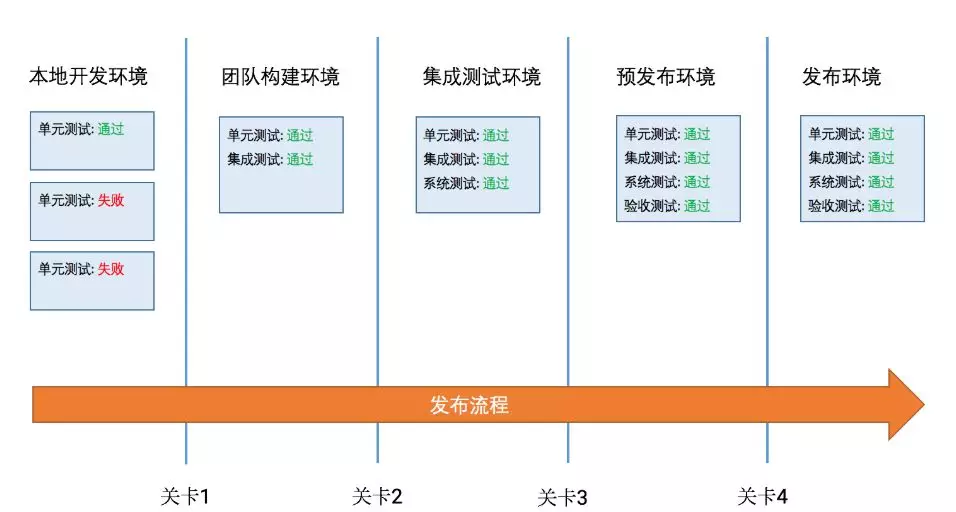
$ sudo /etc/init.d/nscd restart

# Binary & Artifact Management

## JFrog Artifactory

JFrog Artifactory 企业通用制品库，支持所有开发语言，是整个 DevOps 流水线中所有软件包、容器映像和 Helm 图表的单一数据源。被 Amazon、Google、Netflix、Uber、VMware、Facebook 和 Spotify 使用。

在软件交付中，质量可信、安全可信是评估版本可靠性的两个重要标准。在这个过程中，就像使用漏斗一样将构建通过层层筛选，从 int 仓库到 stage 仓库，最后到 release 仓库完成了制品的发布。通过 Artifactory 为制品管理的打造一个单一可信源，从而为软件的持续交付铺路。



如果把每个阶段执行的任务和结果都作为元数据记录在软件制品上，然后在各个交付阶段之间设置准入关卡，就可以通过系统自动化的方式解决各个阶段准入准出的合规性问题，从而降低大量的沟通成本，并且对于交付过程有很好的可追溯性，比如哪个阶段最耗时，哪个阶段经常容易出问题以及交付质量的变化趋势。

Artifactory 依赖解析：Artifactory 将构建任务、构建历史及依赖信息有条理地管理起来，方便架构师在微服务拆分时，对正反向依赖进行追踪

**JFrog Artifactory管理二进制文件和包依赖**

跟 CI 工具更好的集成，通过提供的丰富的 REST API，因此 GUI 页面上的任何操作都可以通过代码以编程方式完成，方便实现 CI/CD

**Artifactory仓库命名方法：<team>-<technology>-<maturity>-<locator>**

如team1-generic-dev-amec

dev 意味着 development（开发），该仓库对所有产品成员都具有读写权限，他们可以上传一些库或其他一些二进制文件。

int 表示 integration（集成），比如从 Jenkins 里成功构建的制品将首先放在这个存储库下，如果构建失败，它将不会被上传到 Artifactory。

stage 表示预发布仓库，通过 Unit Test/Smoke Test 的制品会被 Promote 这个仓库待进一步测试，比如手动测试。

release 通过测试的制品会被 Promote 到这个仓库下。

## Nexus Repository Manager

**Component** **(artifacts, packages, bundles, archives):** is a package of resources that your software application uses (e.g., a library or a framework). Examples of components in these repositories: 1) package managers like npm, nuget, gem, 2) build tools such as Maven, Gradle, rake, grunt, 3) IDE’s such as Eclipse, IntelliJ, Visual Studio.

**Asset**: is a single file associated with a component.

**Repositories (registry):** components are aggregated into collections called a repository in order to make consumption and usage easier. Examples of repositories available on the internet as a service include the following:

1) Central Repository (Maven Central): is the largest repository for Java-based component, managed by Sonatype.

2) NuGet Gallery,

3) RubyGems.org,

4) npmjs.org,

5) PyPI

6) Docker.

**Maven Repository Format**: When Maven downloads a component like a dependency or a plugin from a repository, it also downloads that component’s **POM** (Project Object Model) in XML which describes a project and lists a project’s dependencies — the binary software components.

# commons-lang with version 1.2.0 in group org.apache.commons

<repoURL>/org/apache/commons/commons-lang/1.2.0/commons-lang-1.2.0.jar

<repoURL>/org/apache/commons/commons-lang/1.2.0/commons-lang-1.2.0.pom

hosting your own **private repositories** for internal components has proven to be a very efficient methodology to exchange components during all phases of the software development life cycle

为什么需要安装**repository manager** (e.g., Neuxs) ?

* 一些无法从外部仓库下载的构件，如内部项目需要部署到私服上，以便其他依赖项目使用。
* 为了节省带宽和时间，在局域网架设私有仓库服务器，用其代理所有外部的远程仓库。

当本地项目需要下载构件时，先去私服请求，若没有，则再去远程仓库请求，从远程仓库下载构件后，把构件缓存在私服上。

repository formats use databases for storage and REST API interactions

via Nexus Repository Manager OSS to proxy and cache other repositories such as Red Hat, Oracle or the Apache Software foundation

Download nexus-3.15.2-01-unix.tar.gz

$ tar -xvzf nexus-3.15.2-01-unix.tar.gz /opt

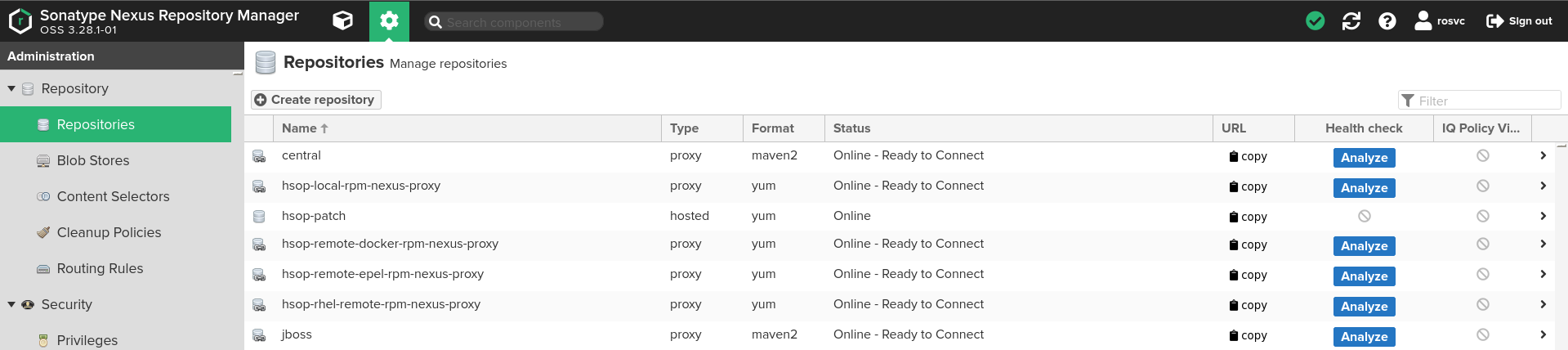
$ cd /opt/nexus/bin

$ ./nexus run # 启动Nexus服务

Started Sonatype Nexus OSS 3.15.2-01

In ETX of mmsim, open webbrowser:

Firefox: <https://repo-0:3443> rosvc/radonc@123



在Repositories仓库管理界面中有多种默认的仓库，也可以添加新的仓库。

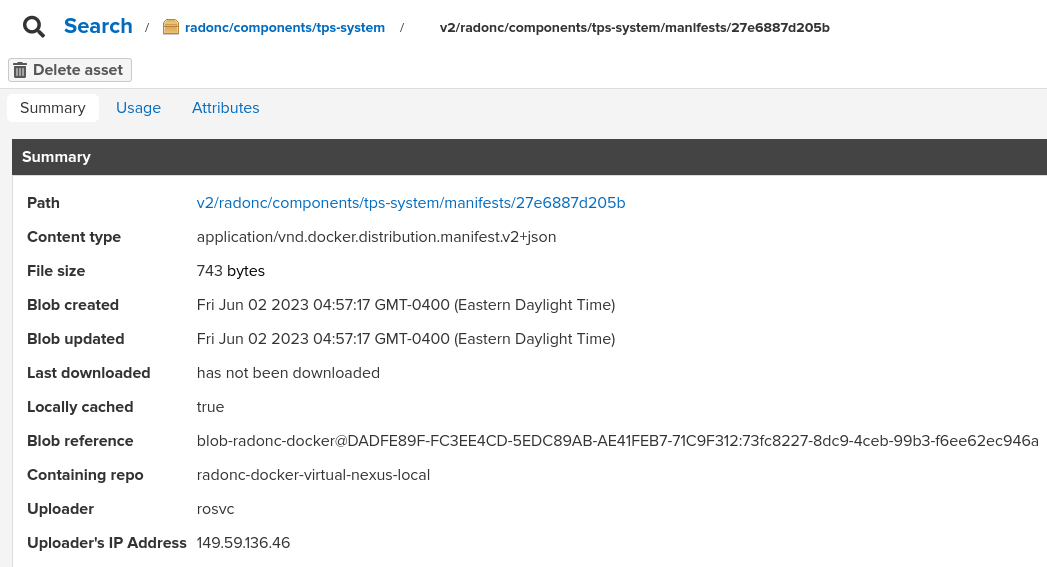
Type=proxy（如central, radonc-docker-virtual-nexus-proxy）：stores components in a remote repository. 表示代理远程仓库，当在下载组件时，如果代理仓库搜索不到，则会把请求转发到远程仓库。从远程仓库下载后缓存到代理仓库。

Type=hosted（如private-release, radonc-docker-virtual-nexus-local）：stores components within your Nexus Repository instance. 表示宿主仓库，主要用来部署团队内部使用的内部组件。

Type=group（如public, radonc-docker-virtual, radonc-pypi-virtual）: contains either or both hosted and proxy repositories. 一般包含多个proxy仓库和hosted仓库，在项目中一般引入group仓库就可以下载到proxy and hosted的包

Search: tps-system

点击每个包，可以查看相关信息（包括下载和属性信息）



# BDD（Behavior Driven Development行为驱动开发）

根据明确的预期行为（一组场景）来进行开发和测试。这种预期行为使用一种特定规范格式进行描述，旨在消除需求从客户，到产品经理，再到开发/测试时的信息失真问题。

行为驱动多用于UI层的测试。因此BDD框架的自动化一般结合Selenium使用。

BDD并不会定义软件怎么做，而是能做什么。最终通过python代码进行验证。

BDD允许不懂编程的项目成员用自然语言阐述系统的功能和使用场景，从而测试人员可以根据描述直接编写测试用例并进行自动化测试。

<https://behave.readthedocs.io/en/latest/>

install behave and selenium

pip install behave

pip install selenium

features

baidu.feature //Gherkin语言（自然语言）编写，对非技术人员比较友好

environment.py

steps

baidu\_steps.py

“.feature”文件有两个用途：文档和自动化测试。在“.feature”里编写测试场景

**baidu.feature**

Feature: 百度搜索 //功能名称

Scenario: 搜索关键词 //场景描述

Given 关键词 behave // put the system in a known state

When 打开百度页面 // take key actions the user (or external system) performs

And 输入关键词

And 点击百度一下按钮

Then 页面标题中应包含关键词 // observe outcomes.

// 对于操作前和操作后的前置条件和后置条件，是放在了“environment.py”文件中定义

**environment.py**

from selenium import webdriver

def before\_scenario(context, scenario):

context.driver = webdriver.Chrome(r"/usr/local/bin/chromedriver")

def after\_scenario(context, scenario):

context.driver.close()

**baidu\_steps.py**

from behave import given, when, then

from selenium import webdriver

from time import sleep

# context是Feature或Scenario的实例化，有参数的话，加上对应参数，可以用来传递信息。

@given('关键词 {keyword}') # 对应步骤 Given 关键词 behave， 参数放在{}中

def step\_impl(context, keyword):

context.keyword = keyword # 将参数绑定上下文对象，以便其他步骤使用

@when('打开百度页面')

def step\_impl(context):

context.driver = driver = webdriver.Chrome() # 同样绑定上下文对象

driver.implicitly\_wait(10)

driver.get('https://www.baidu.com')

@when('输入关键词')

def step\_impl(context):

context.driver.find\_element('id', 'kw').send\_keys(context.keyword)

@when('点击百度一下按钮')

def step\_impl(context):

context.driver.find\_element('id', 'su').click()

sleep(0.5)

@then('页面标题中应包含关键词')

def step\_impl(context):

assert context.keyword in context.driver.title

通过命名behave调用“.feature”文件，映射到py文件里的步骤下的函数，执行这些函数。

features$ behave // 测试

$ pip install allure-behave

Install allure命令行工具

// 结合allure生成报告, -o是指定allure报告数据输出目录

feature$ behave -f allure\_behave.formatter:AllureFormatter -o allure\_data

feature$ allure generate allure\_data -o allure\_html // 生成html报告.

# 自动化运维：Ansible

Ansible 是一个用 Python 开发的自动化运维工具，执行批量系统配置、程序部署、运行命令和滚动更新等任务。Ansible 本身是一个执行框架，真正执行运维任务的是它的诸多模块。

管理机(192.168.16.130)和多台客户机(192.168.16.135, 192.168.16.136), 在管理机上操作，客户机不需。

# 确保管理机可以访问客户机

[root@localhost ~] ping -c2 192.168.16.135; ping -c2 192.168.16.136

# 确保管理可以免密ssh登录客户机

[root@localhost ~] ssh-keygen #生成密钥

[root@localhost ~] ssh-copy-id root@192.168.16.135 #拷贝密钥至客户机

[root@localhost ~] ssh-copy-id root@192.168.16.136 #拷贝密钥至客户机

[root@localhost ~] yum -y install ansible 在主机上安装ansible

[root@localhost ~] ansible -h

[root@localhost ~] ansible-doc -l 列出所有模块

[root@localhost ~] ansible-doc <module> 查看指定模块的帮助用法

[root@localhost ~] ansible localhost -m ping 使用ping模块测试连通性

## ad-doc

# 在客户机上安装软件 (如tree, httpd)

[root@localhost ~] ansible 192.168.16.135 -m yum -a 'name=tree state=installed'

[root@localhost ~] ansible 192.168.16.135 -m yum -a 'name=httpd state=installed'

# 在客户机上查看软件是否已安装

[root@192.168.16.135 ~] rpm -qa tree

[root@192.168.16.135 ~] rpm -qa httpd

# 在客户机上卸载软件 (如tree, httpd)

[root@localhost ~] ansible 192.168.16.135 -m yum -a 'name=tree state=absent'

[root@localhost ~] ansible 192.168.16.135 -m yum -a 'name=httpd state=absent'

command默认模块 (依赖 Python 解析器) ，不支持shell变量和管道等 shell 相关的东西

[root@localhost ~] ansible 192.168.16.135 -m command -a 'chdir=/etc cat centos-release'

shell模块 (依赖 Python 解析器): 用shell执行命令

[root@localhost ~] ansible 192.168.16.135 -m shell -a "echo HOSTNAME"

script模块: 直接调用目标客户端的/bin/sh来执行ansible服务器上的脚本

[root@localhost ~] ansible 192.168.16.135 -m script -a /data/test.sh

raw、。

Copy模块: 从ansible服务器主控端复制文件到远程主机

[root@localhost ~] ansible 192.168.16.135 -m copy -a “src=/etc/ dest=/backup”

Fetch模块 : 从远程主机提取文件至ansible的主控端

[root@ansible ~] ansible all -m fetch -a 'src=/etc/redhat-release dest=/data/os'

File模块: 设置文件属性

[root@ansible ~] ansible all -m file -a 'path=/data/test.txt state=touch'

# 递归删除目录和删除文件

[root@ansible ~] ansible webservers -m file -a "dest=/path/to/c state=absent"

unarchive模块: 解包解压缩

[root@ansible ~] ansible all -m unarchive -a 'src=/tmp/foo.zip dest=/data copy=no mode=0777'

copy：默认为yes，将ansible主机上的压缩包传到远程主机后解压缩至特定目录

copy=no, 将远程主机上的某个压缩包解压缩到指定路径下

Archive模块：打包压缩

[root@ansible ~] ansible all -m archive -a 'path=/var/log/ dest=/data/log.tar.bz2 format=bz2 owner=wang mode=0600'

Hostname模块: 管理主机名

[root@ansible ~] ansible 192.168.16.135 -m hostname -a “name=websrv”

Cron模块: 计划任务

[root@ansible ~] ansible 10.0.0.8 -m cron -a 'hour=2 minute=30 weekday=1-5 name="backup mysql" job=/root/mysql\_backup.sh'

Yum模块: 管理软件包，只支持RHEL，CentOS，fedora，不支持Ubuntu其它版本

[root@ansible ~] ansible websrvs -m yum -a ‘name=httpd state=present’ #安装

[root@ansible ~] ansible websrvs -m yum -a ‘name=httpd state=absent’ #删除

Service模块: 管理服务

[root@ansible ~] ansible all -m service -a 'name=httpd state=started enabled=yes'

User模块：管理用户

[root@ansible ~] ansible all -m user -a 'name=user1 comment=“test user” uid=2048 home=/app/user1 group=root'

Group模块：管理组

[root@ansible ~] ansible websrvs -m group -a 'name=nginx gid=88 system=yes'

Lineinfile模块：相当于sed，可以修改文件内容

Replace模块: 类似于sed命令，主要也是基于正则进行匹配和替换

Setup模块: 收集主机系统信息，facts信息可以直接以变量使用。如果主机较多影响执行速度，可以使用gather\_facts: no来禁止 Ansible 收集 facts 信息

## Playbook

通过Ansible 提供的用于远程执行命令的 command、shell、raw、script 模块，我们可以

1. 将被管理节点的IP写入 Ansible 的“演员”/etc/ansbile/hosts/ inventory，
2. 将要执行的命令编排成 Ansible 的 playbook，
3. 然后运行这个 playbook 就可以达到远程配置的目的了

Ansible 里的 playbook 作用就跟拍电影的剧本一样，把要执行的命令、任务在 playbook 里编排好，然后 Ansible 执行这个 playbook

playbook #相当于总剧本

|----play #相当于场景剧本

|----tasks #相当于分镜头剧本

|----pre\_tasks # 执行环境的准备

**|----tasks # 执行命令清单**

|----post\_tasks #执行环境的清理

Ansible playbook YAML 语法:

--- #开头，非必需

- name: Network Getting Started First Playbook **# play名称**

connection: network\_cli #连接方式

gather\_facts: false #是否收集被管理节点的信息.

hosts: all #针对哪些被管理节点

**tasks: # 任务清单**

- name: Get config for VyOS devices #任务名称

**vyos\_facts: #模块**

gather\_subset: all #收集信息子集.

使用空格来分隔内容,用 ‘#’ 来写注释.

* **变量**

# 自定义变量和外部变量文件

vars:

http\_port: 80

vars\_files: # 使用外部的变量文件隔离敏感数据

- /vars/external\_vars.yml

# Ansible facts

Ansible facts主要用于获取远程系统的数据,从而可以在playbook中作为变量使用.

gather\_facts: yes # 收集被管理节点的信息.

$ ansible localhost -m setup

# 命令行传递变量

ansible-playbook release.yml --extra-vars "hosts=vipers user=starbuck"

ansible-playbook release.yml --extra-vars "@some\_file.json"

foo: "{{ variable }}/additional/string/literal" # 使用变量

单引号或双引号区别：双引号可以使用转义.

template: src=foo.cfg.j2 dest={{ remote\_install\_path }}/foo.cfg

{{ ansible\_eth0.ipv4.address }} # 嵌套的数据结构

{{ foo[0] }} # 访问数组元素

基于Jinja2使用变量，在模板中你自动会获取在主机范围之内的所有变量的访问权

变量优化级

1. extra vars (在命令行中使用-e) 优先级最高
2. 然后是在inventory中定义的连接变量(比如ansible\_ssh\_user)
3. 接着是大多数的其它变量(命令行转换,play中的变量,included的变量,role中的变量等)
4. 然后是在inventory定义的其它变量
5. 然后是由系统发现的facts
6. 然后是 "role默认变量", 这个是最默认的值,很容易丧失优先权

* 条件语句

vars:

epic: true

tasks:

- shell: echo "This certainly is epic!"

when: epic

when: not epic

when: foo is defined

when: bar is not defined

* 循环

- name: add several users

user: name={{ item }} state=present groups=wheel

with\_items:

- testuser1

- testuser2

等效

- name: add user testuser1

user: name=testuser1 state=present groups=wheel

- name: add user testuser2

user: name=testuser2 state=present groups=wheel

- name: Print phone records

debug: msg="User {{ item.key }} is {{ item.value.name }} ({{ item.value.telephone }})"

with\_dict: "{{users}}" # 循环哈希表

假定

users:

alice:

name: Alice Appleworth

telephone: 123-456-7890

bob:

name: Bob Bananarama

telephone: 987-654-3210

- copy: src={{ item }} dest=/etc/fooapp/ owner=root mode=600

with\_fileglob: # 循环文件列表

- /playbooks/files/fooapp/\*

- debug: msg="{{ item.0 }} and {{ item.1 }}"

with\_together: # 循环并行数据集

- "{{alpha}}"

- "{{numbers}}"

- file: dest=/var/stuff/{{ item }} state=directory

with\_sequence: start=4 end=16 stride=2 # 循环整数序列

- debug: msg={{ item }}

with\_random\_choice: # 循环随机序列

- "go through the door"

- "drink from the goblet"

- "press the red button"

- "do nothing"

# Do-Until循环

- action: shell /usr/bin/foo

register: result

until: result.stdout.find("all systems go") != -1

retries: 5

delay: 10

递归运行shell模块,直到模块结果中的stdout输出中包含”all systems go”字符串,或者该任务按照10秒的延迟重试超过5次

ansible playbook 默认第一个 task 是 Gathering Facts 收集各主机的 facts 信息，以方便我们在 paybook 中直接引用 facts 里的信息。

每个 task 的目标在于执行一个 module, 通常是带有特定的参数来执行。在参数中可以使用变量（variables）。 modules具有”幂等”性。

**notify and handlers**

当远端系统被人改动时,可以重放 playbooks 达到恢复的目的，也可以响应这种改动

# 当文件内容被改动时, 重启两个 services

- name: template configuration file

template: src=template.j2 dest=/etc/foo.conf

notify:

- restart memcached

- restart apache

handlers: #主要是用来重启服务或者触发系统重启

- name: restart memcached

service: name=memcached state=restarted

- name: restart apache

service: name=apache state=restarted

notify下列出的即是 handlers. Handlers 也是一些 task 的列表, 通过名字来引用。

**Roles**

Roles 基于一个已知的文件结构，去自动的加载某些 vars\_files，tasks 以及 handlers

site.yml

webservers.yml

fooservers.yml

roles/

common/

files/

templates/

tasks/

handlers/

vars/

defaults/

meta/

webservers/

files/

templates/

tasks/

handlers/

vars/

defaults/

meta/

如果roles/x/tasks/main.yml存在, 其中列出的 tasks 将被添加到 play 中

如果roles/x/handlers/main.yml存在, 其中列出的 handlers 将被添加到 play 中

如果roles/x/vars/main.yml存在, 其中列出的 variables 将被添加到 play 中

如果roles/x/meta/main.yml存在, 其中列出的 “角色依赖” 将被添加到 roles 列表中

所有 copy tasks 可以引用 roles/x/files/ 中的文件，不需要指明文件的路径。

所有 script tasks 可以引用 roles/x/files/ 中的脚本，不需要指明文件的路径。

所有 template tasks 可以引用 roles/x/templates/ 中的文件，不需要指明文件的路径。

所有 include tasks 可以引用 roles/x/tasks/ 中的文件，不需要指明文件的路径。

**roles/x/meta/main.yml**

---

dependencies: # 角色依赖，自动将其他 roles 拉取到现在使用的 role

- { role: common, some\_parameter: 3 }

- { role: apache, port: 80 }

- { role: postgres, dbname: blarg, other\_parameter: 12 }

**webservers.yaml**

---

- hosts: webservers

roles:

- common

- webservers

# 为 roles 设置触发条件, 从而条件子句用到 role 每一个 task 上

- { role: some\_role, when: "ansible\_os\_family == 'RedHat'" }

**site.yml**

---

- include: webservers.yml

- include: dbservers.yml

$ ansible-playbook site.yml -f 10

# Security Assessment

## CVSS: Common Vulnerability Scoring System通用漏洞评分系统

CVSS provides a way to capture the principal characteristics of a vulnerability and produce a numerical score (Scores range from 0 to 10, with 10 being the most severe) reflecting its severity. The numerical score can then be translated into a qualitative representation (such as low, medium, high, and critical) to help organization properly assess and prioritize their vulnerability management processes.

|  |  |
| --- | --- |
| The severity rating for the numerical values are mapped below | |
| None | 0.0 |
| Low | 0.1 - 3.9 |
| Medium | 4.0 - 6.9 |
| High | 7.0 - 8.9 |
| Critical | 9.0 - 10.0 |

The **Severity** and **CVSS vector** of each vulnerability is calculated using the CVSS V3 **Base Score Metrics** Calculator located [here](https://nvd.nist.gov/CVSS/v3-calculator). Vulnerabilities identified during security assessment are classified into standardized categories. Refer following table for more information:

|  |  |
| --- | --- |
| Categories for vulnerability classification | |
| Web application security assessment | OWASP Top Ten - 2017 |
| Mobile application security assessment | OWASP Top Ten - 2016 |
| IoT/Hardware security assessment | OWASP Top Ten – 2014 |