16/12/19

**Linux(Bash)**

Bash file should start with:

#!/bin/bash

\*/bin/bash can be replaced by the which bash you use

Bash file not necessarily have to have a .sh extension, it’s visually more clear that a file with .sh is a bash file.

Environment variables in scripts:

Start with “$”, use “env” to check environment variable

Exp: $HOME in bash file

Don’t overwrite existing variables

Environment variables are should be uppercase

Exp:

FIRSTNAME = “Liang” (no space around equal sign)

Export FIRSTNAME

**[Export]** is a built-in command of the Bash shell. It is used to mark variables and functions to be passed to child processes. Basically, a variable will be included in child process environments without affecting other environments.

<https://www.journaldev.com/28251/export-command-linux>

In simple terms, environment variables are set when you open a new shell session. at any time if you change any of the variable values, the shell has no way of picking that change. The export command, on the other hand, provides the ability to update the current shell session about the change you made to the exported variable. You don’t have to wait until new shell session to use the value of the variable you changed.

<https://www.geeksforgeeks.org/export-command-in-linux-with-examples/>

**export FIRSTNAME=”Liang” (** “” stores normal value**)**

**export TODAYDATE=`date`** (``back ticks store the results of using command **date,** \*run the same command again to update date value)

COMMENTS in bash scripts

Start with # sign

Exp:

#COMMENTS

**Docker**

In docker-compose file

“Build .” build from DockerFile in current location

Swarm Mode

A server clustering service

To check swarm is active or not: **docker swarm | grep Swarm**

By default, swarm is inactive, to active swarm: **docker swarm init** (create a manager node)

Create a new service:

**docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]**

Exp:

docker service create alpine ping 8.8.8.8 (creates 1 alpine replica)

ID NAME MODE REPLICAS IMAGE PORTS

mjfnz592fd8g awesome\_bohr replicated 1/1 alpine:latest

**Usage: docker update [OPTIONS] CONTAINER [CONTAINER...]**

Update configuration of one or more containers

**Usage: docker service update [OPTIONS] SERVICE**

Update a service

When one of the containers failed, Swarm will automatically start a new container.

Steps:

1. docker swarm init --advertise-addr 192.168.0.48(current manager node IP) on manager node
2. docker swarm join --token SWMTKN-1-0v9wupgq9ni1m0t9y83b8g0xo9vz3g7ntoune0w6ldo5bcf93k-4zdxl7xqxmlb87xbk7nydawb9 192.168.0.48:2377 (Join swarm as a worker) on worker node
3. docker node ls(check node list)
4. Change a worker to Manager node:
   1. docker node update –role manager xxx(node name)
   2. docker swarm join-token manager(manage node) to get command for changing worker node to manager node(worker node)
5. docker service create --replicas 3 alpine ping 8.8.8.8 (create a servicewith 3 replicas on manager node)

\* docker node

\* docker swarm

\* docker service

what’s the relation between node, swarm and service?

A **node** is an instance of the Docker engine participating in the swarm.

A **task** is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container.

A **service** is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm. When you create a service, you specify which container image to use and which commands to execute inside running containers. In the replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state.

<https://docs.docker.com/engine/swarm/key-concepts/>

swarm mode that enables you to create a cluster of one or more Docker Engines called a swarm. A swarm consists of one or more nodes.

<https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>

What’s the difference between a work node and a manager node?

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 also run services as worker nodes, but you can configure them to run manager tasks exclusively and be manager-only nodes.

17/12/19

**Linux(Bash)**

Command Substitution, two methods

1. static data:

USERFILES=`find /home –user adminuser`

Echo “All Files Owned by adminuser:$USERFILES”

1. dynamic data:

shopt –s expand\_aliases

alias TODAY=”date”

D=TODAY

Echo “With Alias, TODAY is: $D”

**Shopt –s expand\_aliases** (expand alias in sub shell, alias only exist in sub shall -> more secure?)

Exit Status

**Echo $? (**check last command’s exit status, command success returns 0, if return not 0, means failed/error**)**

**Set –e (**when a bash file occurs an error, stop the process, exit the shell, this is to prevent the wrong script overwrite important values/scripts**)**

Arithmetic Operations (add, subtract, multiply, divide …)

**Expr expression**

Exp:

**Expr 2 + 2 (put space around “+” sign)**

Use “\\*” times to avoid “\*” wildcard, same happens to “()”, it should be “\( 2 + 2 \) / 4” result is 1

Global and Local Environment Variables

**Env and printenv (**Check global environment variables**)**

**Set (**Check Local environment variables, only specific to current login user**)**

**Docker**

Overlay Multi-Host Networking

**--driver overlay**

For container-to-container traffic inside a single Swarm

Exp:

docker network create --driver overlay mydrupal (create an overlay network called mydrupal)

docker service create --name drupal --network mydrupal -p 80:80 drupal

docker service create --name psql --network mydrupal -e POSTGRES\_PASSWORD=mypass postgres

(drupal and postgres can talk to each other)

Routing Mesh

Routes ingress(incoming) packets for a Service to proper Task, spands all nods in Swarm. It uses IPVS from Linux Kernel. Acts like a Load Balance across all tasks.

Two ways:

1. container-to-container in an overlay network (uses VIP)
2. External traffic incoming to published ports (all nodes listen)

Demo commands:

Swarm:

docker swarm init --advertise-addr 192.168.0.8 ( two nodes joined as worker nodes)

Networks:

docker network create -d overlay backend

docker network create -d overlay frontend

Services:

docker service create --name vote -p 80:80 --network frontend --replicas 2 bretfisher/examplevotingapp\_vote

docker service create --name redis --network frontend redis:3.2

docker service create --name db --network backend --mount type=volume,source=db-data,target=/var/lib/postgresql/data postgres:9.4

docker service create --name worker --network frontend --network backend bretfisher/examplevotingapp\_worker:java

docker service create --name result --network backend -p 5001:80 bretfisher/examplevotingapp\_result

Stacks

Production grade compose

Stacks accept Compose files as their declarative definition for services, networks and volumes

Stack manages all those objects for users, including overlay network per stack.

**Docker stack deploy**

Exp: Docker stack deploy –c xxxx.yml xxx

(Can’t do build just like in docker-compose.yml file, this is because building shouldn’t happen on production Swarm)

on local machine Compose ignores deploy, Swarm ignores build

Stack reads docker-compose file without needing docker-compose CLI

Once use a configuration file to manage all the configurations, only change the conguration file.

To update a current stack, change the configuration file(.yml), then

Docker stack deploy –c xxxx.yml xxx (same as the first time deploy stack, the system will recognise the changes in the file)

Secrets Storage

Secure solution for storing secrets in Swarm

What is a Secret?

1. Usernames and passwords
2. TLS certificates and keys
3. SSH keys

Secret only in Swarm, but local docker-compose can use file-based secrets, but not secure (testing purpose?)

Secrets are first stored in Swarm, then assigned to a Service(s)

Only stored on disk on Manager nodes(encrypted) as key-value pair, can have alias

Exp:

/run/secrets/<secret-name> or /run/secrets/<secret-alias>

Two ways to create secrets:

1. Have a .txt file with secret value, run command **docker secret create xxx(secret variable) xx.txt**
2. **Echo “xxxx”(secret value) | docker secret create xxx(secret variable) –**

both will restore secrets locally, which is not recommended

Can also have secret files already and associated in stack(docker-compose.yml) file.

18/12/19

**Linux(Bash)**

Special Characters - Quotes and Escapes

Back slash **\** escapes a single character

Exp:

Echo “\$HOME” gets result-> $HOME (if no \, gets the current user home directory)

Single quote **‘’** escapes more than one character

Exp:

Echo ‘$HOME $PWD’ gets result -> $HOME $PWD

Double quotes **“”** escape first character

/dev/null

anything write to /dev/null will be ignored

echo “abc” >>/dev/null

no return value

Read Statement

**Read**

Exp:

In a shell script

Echo “Entre your first name:”

Read FIRSTNAME

Echo “Your first name is $FIRSTNAME”

Shell Expansion

Echo sh{ot,ort,oot} -> shot short shoot

Echo st{il,al}l -> still stall

Echo ~ -> /root (~ represents user home directory)

Echo ~+ (current working directory, same with PWD)

Echo ~- (Old PWD)

Echo “${!HO\*}” (what does **$** and **!** mean in here?)

The ‘**$**’ character introduces parameter expansion, command substitution, or arithmetic expansion.

${!prefix\*}

${!prefix@}

Expands to the names of variables whose names begin with prefix, separated by the first character of the IFS special variable. When ‘@’ is used and the expansion appears within double quotes, each variable name expands to a separate word.

<https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html>

Echo and sign value at one time

Echo “${FIRSTNAME:=LIANG}”

return LIANG

Types of Variables (Implicit vs. Explicit Definition)

Declare

**Declare –i xxx=xxx** (implicitly declare a variable as an integer)

**Declare +i xxx** (to unset a implicitly signed variable type, for integer)

**Declare –r xxx=xxx** or **readyonly xxx=xxx** (declare a read-only variable)

Arrays

Declare an array:

ArrayName=(value1 value2 value3 …)

No “,” between values like in programming languages

Use index to see a value

Echo ${ArrayName[0]} (echo first value, just like in all programming languages)

Echo ${ArrayName[\*]} (echo all values)

<https://www.geeksforgeeks.org/array-basics-shell-scripting-set-1/>

**Docker**

App lifecycle with docker-compose

If docker-compose.override.yml and docker-compose.yml in same file, system will pick docker-compose.yml first then use docker-compose.override.yml to override docker-compose.yml file.

**docker stack deploy --compose-file docker-compose.yml vossibility** (a single docker compose file)

**docker stack deploy --compose-file docker-compose.yml -c docker-compose.prod.yml vossibility** (a base docker compose file + environment-specific override compose file)

<https://docs.docker.com/engine/reference/commandline/stack_deploy/>

Service Updates

**Docker service update –image xxapp:1.0.1 <servicename>** (updates a service with an newer image)

**Docker service update –en-add NODE\_ENV=production –publish-rm 8080** (Adding env variable and remove a port in one command)

**Docker service scale web=8 api=6** (Change number of replicas of two services)

**Docker stack deploy –c xxx.yml <stackname>** (Update stack using newer yml file)

<https://docs.docker.com/engine/reference/commandline/service_update/>

**Docker service update –force xx** (Balancing service, force service update)

**Docker-Kubernetes**

Swarm vs Kubernetes

Swarm: easier to deploy/manage

Kubernetes: more features and flexibility

**Advantages of Swarm:**

1. Comes with Docker, single vendor container platform
2. Easiest orchestrator to deploy/manage yourself
3. Follows 80/20 rule, 20% of features for 80% of use cases
4. Runs anywhere Docker does: local, Cloud, datacentre; ARM, Windows, 32bit
5. Secure by default
6. Easier to troubleshoot

**Advantages of Kubernetes:**

1. Clouds will deploy/manage K8S for you
2. Infrastructure vendors are making their own distributions
3. Widest adoption and community
4. Flexible: Covers widest set of use cases
5. “K8S first” vendor support

K8S basic terms:

K8S: the whole orchestration system

Kubectl: CLI to configure K8S and manage apps

Node: Single server in the K8S cluster (Similar to Worker)

A Pod always runs on a Node. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by the Master. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster.

<https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/>

Kubelet: K8S agent running on nodes

Control Plane: Set of containers (Master, similar to Manager in Swarm) that manage the cluster, includes API server, scheduler, controller manager, etcd and more

Pod: one or more containers running together on one Node. Containers are always in pods

A Pod is a Kubernetes abstraction that represents a group of one or more application containers (such as Docker or rkt), and some shared resources for those containers.

<https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/>

Controller: for creating/updating pods and other objects. Types of controllers: deployment, replicaSet, StatefulSet, DaemoSet, Job, CronJob …

Services: network endpoint to connect to pod

Namespace: Filtered group of objects in cluster

Container(s) -> Pod -> Node (Worker) -> Control Plane (Master)

Two ways to deploy Pods(containers) in K8S: commands or YAML

Use command:

Kubectl version (Check is K8S is working, should have two versions)

Kubectl run my-nginx –image nginx (run a pod of the nginx web server)

Kubectl get pods (list the pods)

Kubectl get all (see all objects)

When a pod is created, a deployment controller is also created, then replicaSet controller is also created. Deployment controller manages replicaSet controller, replicaSet manages pods.

**$ deployment NAME --image=image [--dry-run]**

Kubectl delete deployment my-nginx (similar to remove(rm) in docker, it deletes deployment controller, replicaSet controller and dops)

Scaling ReplicaSets

Kubectl scale deploy/my-apache --replicas 2

Kubectl scale deployment my-apache --replicas 2

Above two commands are same, in K8S, deploy=deployment=deployments

Kubectl get pods

Kubectl logs deployment/my-apache --follow --tail 1 (get container logs, return last line only)

Kubectl describe pod/my-apache-xx-xxx (use selector, based on the label containers shared. Get a bunch of details about an object, including events, labels are given when cubectl run xxx, similar to inspect in docker/swarm, in the end of output, can see all the event details)

Kubectl get pods –w (like watch command)

Kubectl delete pod/my-apache-xxx-xxx (delete a pod)

Kuberctl delete deploy/xxxx

kubectl Cheat Sheet

<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

kubectl Commands

[https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands - -strong-getting-started-strong-](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#-strong-getting-started-strong-)

19/12/19

**Linux(Bash)**

Passing Variables to Scripts at the Command Line

In the script:

Using **$1 $2** …to refer to the variables that passed at command line

Exp:

**$ abc.sh “Liang” “Chen”** (command line)

**echo “First name is $1 second name is $2”** (in script) -> return First name is Liang second name is Chen

If/Then/Else

To check if a file exist: -a

Exp:

If [ -a “mytxt.txt” ]; then

**If [ condition ];**

**Then**

**#commands to be run if true**

**elif [ condition ];**

**then**

**#commands to be run if true**

**else**

**#commands to be run if true**

**fi**

For loop

**For arg in [List]**

**Do**

**#commands**

**Done**

**For ((expression1; expression2; expression3)) do**

**#commands**

**Done**

Exp:

For ((i=1;i<=100;i++)) do

Echo $i

Done

While loop

**While [ condition ] do**

**#commands**

**Done**

Case statement (same as switch in C#)

**Case “$VAR” in**

**Pattern\_1 )**

**#commands**

**;; (;;means break)**

**Pattern\_2 )**

**#commands**

**;;**

**\* )**

**#default statement**

**;;**

**esac**

**Docker-Kubernetes**

**Kubectl expose** (creates a service for existing pods)

A service is a stable address for pod(s)

Use service to connect pod(s)

CoreDNS allows us to resolve services by name

Service types:

ClusterIP, NodePort, LoadBalancer, ExternName

ClusterIP (Default):

Single, internal virtual IP allocated

Only reachable from within cluster (nodes and pods)

Pods can reach service on apps port number

NodePort:

High port allocated on each node

Port is open on every node’s IP

Anyone can connect (if they can reach node)

ClusterIP and NodePort are always available in K8S

LoadBalancer:

Mostly used in Cloud

Controls a LB endpoint external to the cluster

Only available when infrastructure provider gives you a LB (AWS ELB…)

Creates NodePorts + ClusterIP services, tells LB to send to NodePort, only for traffic coming in from external sources

ExternalName:

Adds CNAME DNS record to CoreDNS only

Not used for Pods, but for giving pods a DNS name to use for something outside K8S

Create ClusterIP & NodePort Service

**kubectl create deployment httpenv --image=bretfisher/httpenv**

**kubectl expose deployment/httpenv --port 8888** (open port 8888 for deployment httpenv)

**kubectl expose deployment --port 8888 --name httpenv-np --type NodePort httpenv** (NodePort, if no –type -> default is ClusterIP)

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/httpenv ClusterIP 10.96.30.81 <none> 8888/TCP 4m44s

service/httpenv-np NodePort 10.102.76.156 <none> 8888:31351/TCP 27s

httpenv-np’s port: 8888:31351 (8888 is inside the cluster, 31351 is on node expose to outside, range between 30000-32767)

Create LoadBalancer

**kubectl expose deployment --port 8888 --name httpenv-lb --type LoadBalancer httpenv**

DNS in K8S

**kubectl get namespaces**

NAME STATUS AGE

default Active 22h

docker Active 22h

kube-node-lease Active 22h

kube-public Active 22h

kube-system Active 22h

Generators

Commands use helper templates called “generators”

**kubectl create deployment test --image nginx --dry-run -o yaml (**output the result of the command to a yaml file without making any actual change.)

apiVersion: apps/v1

kind: Deployment

metadata:

creationTimestamp: null

labels:

app: test

name: test

spec:

replicas: 1

selector:

matchLabels:

app: test

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

app: test

spec:

containers:

- image: nginx

name: nginx

resources: {}

status: {}

what’s the difference between K8S client version and server version?

I have upgraded my client version to 1.17.0, but my server version is still 1.14.8.

Follow link below, only upgraded client version. (already posted my question on Udemy, waiting for answer) <https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-macos>

Bret replied:

Client: the cli binary you are running commands on

Server: the API service running on the node

How you control their versions depends on how you installed them and where they are running.

<https://www.udemy.com/course/docker-mastery/learn/lecture/15126234#questions>

Imperative VS Declarative

Imperative: Focus on how a program operates (do things step by step)

Declarative: Focus on what a program should accomplish (system take care of the process)

Imperative command in K8S:

Kubectl run, kuberctl create deployment, kubectl update

Declarative:

Kubectl apply –f my-resources.yaml

Automation, need to understand YAML keys and values

Three management approaches:

1. Imperative commands: run, expose, scale, edit, create deployment…

Best for dev/learning/personal projects

Easy to learn, hardest to manager over time

1. Imperative objects (middle ground): create –f files.yml, replace –f file.yml, delete…

Good for prod of small environments, single file per command

Store your changes in git-based yaml files

Hard to automation

1. Declarative objects: apply –f file.yml, dir\, diff…

Best for prod, easier to automate

Harder to understand and predict changes

Watched a video talks about Cloud Computing in the Year 2020

<https://www.youtube.com/watch?v=1pBuwKwaHp0>

In the video, it states that serverless will play a big role in 2020, I’m agreed with that, therefore, need to start doing AWS lambda studying.

Holiday Docker and Kubernetes Q&A: DevOps and Docker Show (Ep 66) By Bret Fisher Docker and DevOps

<https://www.youtube.com/watch?v=Oj6wDZKbLoU>

20/12/19

**Linux(Bash)**

Reading Files

**echo "Entre a file name to read:"**

**read FILE**

**while read -r SUPERHERO; do**

**echo "Superhero Name: $SUPERHERO"**

**done < "$FILE"**

(read a file, then output its content with prefixed text, read –r read from a file)

File Descriptors and Handles

**echo "Entre a filename to read:"**

**read FILE**

**exec 5<>$FILE**

**while read -r SUPERHERO; do**

**echo "Superhero name: $SUPERHER"**

**done <&5**

**echo "THis file Read On: `date`" >&5**

**exec 5>&-**

(use file descriptor to represents a file, has to be equal or greater to 3, < read, > write, in this case 5 represents the file, in the end, exec 5>&- to close the file)

IFS and Delimiting

**echo "Enter filename to parse: "**

**read FILE**

**echo "Enter the Delimiter: "**

**read DELIM**

**IFS="$DELIM"**

**while read -r CPU MEMORY DISK; do**

**echo "CPU: $CPU"**

**echo "Memory: $MEMORY"**

**echo "Disk: $DISK"**

**done <"$FILE"**

(IFS represents delimiter, by default its “ “ space)

Traps and Signals

**trap 'echo " - Please press Q to exit.."' SIGINT SIGTERM SIGTSTP**

**clear**

**while [ "$CHOICE" != "Q" ] && [ "$CHOICE" != "q" ]; do**

**echo "MAIN MENU"**

**echo "1) ONE"**

**echo "2) TWO"**

**echo "3) THREE"**

**echo "Q) Quite"**

**echo CHOICE**

**clear**

**done**

**Docker-Kubernetes**

**Kubectl apply**

**Kubectl apply -f filename.yaml** (create/update resources in a file)

**Kubectl apply -f myyaml/** (create/update a whole directory of yaml)

**Kubectl apply -f https://xxx.xxx/xxx.yml** (create/update from a URL)

**Curl -L** [**https://xxx.xxx/xxx**](https://xxx.xxx/xxx)(look at the yaml file first)

Configuration YAML

YAML or JSON?

YAML is more human readable, in the background, the system converts YAML to JSON.

Each file contains one or more manifests, each manifest describes an API object (deployment, job, secret), each manifest needs four parts (four root key: values in the file)

apiVersion: get the API versions the cluster supports

**kubectl api-version**

kind: get a list of resources the cluster supports

**kubectl api-resources**

metadata: only name is required

spec: where all the action is at

exp YAML file:

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.17.3

ports:

- containerPort: 80

**kubectl explain services –recursive** (display all the keys each kind supports)

**kubectl explain services.spec** (ServiceSpec describes the attributes that a user creates on a service)

**kubectl explain services.spec** (type determines how the Service is exposed. Defaults to ClusterIP. Valid options are ExternalName, ClusterIP, NodePort, and LoadBalancer)

kubectl apply –f app.yml

vim app.yml (change app.yml)

kubectl diff –f app.yml (use diff to see the changes)

Labels:

Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object.

<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>

Labels goes under metadata in YAML

Simple list of key:value for identifying your resource later by selecting, grouping, or filtering for it

Common examples include tier:frontend, app:api, env:prod, customer:xxx.co

Filter exp:

Kubectl get pods –l app=nginx (only get pods with nginx label)

Kubectl applu –f xx.yaml –l app=nginx ( only apply certain parts of the yaml file)

Label Selectors

Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).

Via a label selector, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

The API currently supports two types of selectors: equality-based and set-based. A label selector can be made of multiple requirements which are comma-separated. In the case of multiple requirements, all must be satisfied so the comma separator acts as a logical AND (&&) operator.

<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>