

linux 从入门到放弃

蔚雷

September 20, 2018

Contents

1	自动化安装系统	3
1.1	kickstart 安装部署步骤	4
1.1.1	创建 ks.cfg 文件	5
1.2	Cobbler	6
2	docker 基础到 kubernetes 集群	9
2.1	docker 基础	9
2.2	dockerfile 使用总结	9
2.2.1	copy 和 add 的区别	9
2.2.2	ENTRYPOINT 和 CMD	9
2.3	docker 使用	11
2.4	docker compose	12
2.5	参考链接	12
2.6	kubernetes 组件及基础概念	12
2.6.1	kube-master	12
2.6.2	kube-node	13
2.6.3	pod	13
2.6.4	Replication Controller	13
2.6.5	replica set	13
2.6.6	Service	14
2.7	kubect1 高级用法	14
2.8	定义 pod	15
2.8.1	限制容器使用资源	15
2.8.2	pod 生命周期与重启策略	15
2.8.3	创建 pod 拉取私有库镜像	16
2.9	kubeconfig 配置	17
2.10	生产环境中使用 kubernetes	19
2.10.1	openshift	19
2.11	kubernetes addon	19
2.11.1	Discovering Services	19
2.12	pod 的持久化	19
3	自动化管理工具	21
3.1	ansible	21
3.2	salt	21
3.2.1	salt 介绍	21
3.2.2	salt 中 grains 与 pillar	21
3.2.3	远程执行	23
3.2.4	jinja	23
3.2.5	状态模块 state	23
3.2.6	salt 实践	25

Preface

运维工作四五年，所做之事已无乐趣，了无生趣，总结运维之事，安装系统，部署服务，改参数，调配置，部署代码，使用工具大致都相仿，什么 `shell`, `kickstart`, `cobbler`, `ansible`, `salt`, `nginx`, `apache`, `php`, `tomcat`, `keepalived`, `mysql`, `git`, `svn` `jenkins`. 等之类软件，用什么都不能说精，也不能说不会，昏昏沉沉一年过去，不闻其道，不见其神，只会其术，可谓一塌糊涂，这样下去无非是浪费时间，不如借此时机，把所学之事整理成成文，

Chapter 1

自动化安装系统

Redhat 系主要有两种方式安装系统 Kickstart 和 Cobbler。Kickstart 是一种无人值守的安装方式。它的工作原理是在安装过程中记录人工干预填写的各种参数，并生成一个名为 `ks.cfg` 的文件。如果在自动安装过程中出现要填写参数的情况，安装程序首先会去查找 `ks.cfg` 文件，如果找到合适的参数，就采用所找到的参数；如果没有找到合适的参数，便会弹出对话框让安装者手工填写。所以，如果 `ks.cfg` 文件涵盖了安装过程中所有需要填写的参数，那么安装者完全可以只告诉安装程序从何处下载 `ks.cfg` 文件，然后就去忙自己的事情。等安装完毕，安装程序会根据 `ks.cfg` 中的设置重启/关闭系统，并结束安装。

Cobbler 集中和简化了通过网络安装操作系统需要使用到的 DHCP、TFTP 和 DNS 服务的配置。Cobbler 不仅有一个命令行界面，还提供了一个 Web 界面，大大降低了使用者的入门水平。Cobbler 内置了一个轻量级配置管理系统，但它也支持和其它配置管理系统集成，如 Puppet，暂时不支持 SaltStack。

在这之前需要了解几个概念，PXE(pre-boot execution environment) 预启动执行环境，通过网络接口启动计算机，不依赖本地存储设备或本地已安装的操作系统，它的工作模式是 client/server 工作模式，PXE 客户端会调用网际协议 IP，用户数据报协议 (UDP)，动态主机设定 (DHCP)，小型文件传输协议 (TFTP) 等网络协议。PXE 工作过程

DHCP (Dynamic Host Configuration Protocol, 动态主机配置协议) 通常被应用在大型的局域网络环境中，主要作用是集中的管理、分配 IP 地址，使网络环境中的主机动态的获得 IP 地址、网关地址、DNS 服务器地址等信息，并能够提升地址的使用率。端口号 67，安装系统时开启，安装后关闭

TFTP (Trivial File Transfer Protocol, 简单文件传输协议) 是 TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。端口号为 69。

1.1

1. PXE 客户端 (需要安装系统的机器) 通过 PXE BOOTROM (自启动芯片) 会以 UDP 发送一个广播，向本网络中的 DHCP 服务器索取 IP
2. DHCP 服务端收到客户端的请求，验证是否来自合法的 PXE client 的请求，验证通过它将给客户端一个响应其中包含为客户端分配的 IP 地址，PXELINUX 启动程序 (TFTP) 位置以及配置文件所在位置
3. 客户端收到服务端的回应后会再请求传送启动所需文件：pxelinux.0, pxelinux.cfg/default, vmlinuz, initrd.img
4. 服务端通过 TFTP 通讯协议从 Boot server 下载启动安装程序所必需的文件，然后根据该文件中定义的引导顺序，启动 linux 安装程序的引导内核
5. 客户端通过 pxelinux.cfg/default 文件成功引导 linux 安装内核后，安装程序必须确定通过什么安装介质来安装 linux，如果通过网络安装 (nfs, ftp, http, ..)，便会初始化网络，并定位安装源位置。此时会读取 default 文件中指定的自动应答文件 ks.cfg 所在位置，根据该位置请求下载该文件
6. 从服务端下载完 ks.cfg 文件后，通过该文件找到 os server，并按照文件的配置请求下载安装过程需要的软件包。os server 和客户端建立连接后，将开始传输软件包，客户端将开始安装操作系统。安装完成后将重新引导计算机。

这里有个问题, 在第 2 步和第 5 步初始化 2 次网络了, 这是由于 PXE 获取的是安装用的内核以及安装程序等, 而安装程序要获取的是安装系统所需的二进制包以及配置文件。因此 PXE 模块和安装程序是相对独立的, PXE 的网络配置并不能传递给安装程序, 从而进行两次获取 IP 地址过程, 但 IP 地址在 DHCP 的租期内是一样的。

1.1 kickstart 安装部署步骤

服务端环境环境: CentOS release 6.7 (Final) ip 10.0.0.151 ,selinux, 防火墙关闭, 首先安装 DHCP,TFTP, HTTP 服务

```

1 $ yum install dhcp tftp-server httpd -y
2 $ cat /etc/dhcp/dhcpd.conf
3 subnet 10.0.0.0 netmask 255.255.255.0 {
4     range 10.0.0.100 10.0.0.200; # 可分配起始IP-结束IP
5     option subnet-mask 255.255.255.0; #netmask
6     default-lease-time 21600; #默认租用期限
7     max-lease-time 43200; #最大IP租用期限
8     next-server 10.0.0.151; #TFTP服务器IP
9     filename "/pxelinux.0"; # TFTP根目录下pxelinux.0文件位置
10 }
11 #如果多块网卡可以指定网卡,如果是一块就不需要修改
12 $ cat /etc/sysconfig/dhcpd
13 DHCPDARGS=eth0
14 $ cat /etc/xinetd.d/tftp
15 service tftp
16 {
17     socket_type = dgram
18     protocol = udp
19     wait = yes
20     user = root
21     server = /usr/sbin/in.tftpd
22     server_args = -s /var/lib/tftpboot #请注意这个地方以后会用到
23     disable = no #仅修改这里就可以
24     per_source = 11
25     cps = 100 2
26     flags = IPv4
27 }
28 $ sed -i "27i ServerName 127.0.0.1:80" /etc/httpd/conf/httpd.conf
29 $ mount /dev/sr0 /var/www/html/CentOS-6.7/
30 $ /etc/init.d/dhcpd start
31 $ /etc/init.d/xinetd start
32 $ /etc/init.d/httpd start

```

好吧让我们来看看效果如果出现下面画面证明配置成功啦

??

配置支持 PXE 的启动程序 syslinux syslinux 是一个功能强大的引导加载程序, 而且兼容各种介质。SYSLINUX 是一个小型的 Linux 操作系统, 它的目的是简化首次安装 Linux 的时间, 并建立维护或其它特殊用途的启动盘。如果没有找到 pxelinux.0 这个文件, 可以安装一下。

```

1 $ yum -y install syslinux
2 # 复制启动菜单程序文件
3 $ cp -a /var/www/html/CentOS-6.7/isolinux/* /var/lib/tftpboot/
4 $ cp /usr/share/syslinux/pxelinux.0 /var/lib/tftpboot/
5 $ ls /var/lib/tftpboot/

```



```

6 boot.cat grub.conf isolinux.bin memtest pxelinux.cfg TRANS.TBL vmlinuz
7 boot.msg initrd.img isolinux.cfg pxelinux.0 splash.jpg vesamenu.c32
8 # 新建一个pxelinux.cfg目录，存放客户端的配置文件
9 $ mkdir -p /var/lib/tftpboot/pxelinux.cfg
10 cp /var/www/html/CentOS-6.7/isolinux/isolinux.cfg /var/lib/tftpboot/pxelinux.cfg/default

```

1.1.1 创建 ks.cfg 文件

kickstart 是为了避免安装操作系统的过程中的交互操作，只要定义好一个 kickstar 自动应答配置文件 ks.cfg，并让安装程序知道该配置文件的位置，就可以在安装中读取配置文件来安装系统。

生成 kickstaart 配置文件的三种方法：

每安装好一台 Centos 机器，Centos 安装程序都会创建一个 kickstart 配置文件，记录你的真实安装配置。如果你希望实现和某系统类似的安装，可以基于该系统的 kickstart 配置文件来生成你自己的 kickstart 配置文件。（生成的文件名字叫 anaconda-ks.cfg 位于 /root/anaconda-ks.cfg）

Centos 提供了一个图形化的 kickstart 配置工具。在任何一个安装好的 Linux 系统上运行该工具，就可以很容易地创建你自己的 kickstart 配置文件。kickstart 配置工具命令为 redhat-config-kickstart (RHEL3) 或 system-config-kickstart (RHEL4, RHEL5)。网上有很多用 CentOS 桌面版生成 ks 文件的文章，如果有现成的系统就没什么可说。但没有现成的，也没有必要去用桌面版，命令行也很简单。

阅读 kickstart 配置文件的手册。用任何一个文本编辑器都可以创建你自己的 kickstart 配置文件。

ks.cfg 文件组成大致分为 3 段

命令段：键盘类型，语言，安装方式等系统的配置，有必选项和可选项，如果缺少某项必选项，安装时会中断并提示用户选择此项的选项

* 软件包段

语法基本可以写成

```

1 %packages
2 @groupname: 指定安装的包组
3 package_name: 指定安装的包
4 -package_name: 指定不安装的包
5 * 脚本段(可选)
6
7 %pre:安装系统前执行的命令或脚本(由于只依赖于启动镜像，支持的命令很少)
8 %post:安装系统后执行的命令或脚本(基本支持所有命令)

```

首先要使用 grub-crypt 生成一个密码用于 root 密码，编写 ks 配置文件放到 /var/www/html/ks_config/CentOS-6.7-ks.cfg，

优化脚本，也需要放在下面。/var/www/html/ks_config/optimization.sh

编辑 default 配置文件

```

1 vim /var/lib/tftpboot/pxelinux.cfg/default
2 default ks
3 prompt 0
4 label ks
5 kernel vmlinuz
6 append initrd=initrd.img ks=http://10.0.0.151/ks_config/CentOS-6.7-ks.cfg # 告诉安装程序ks.cfg文件
   在哪里
7 # append initrd=initrd.img ks=http://10.0.0.151/ks_config/CentOS-6.7-ks.cfg ksdevice=eth0
8 # 多了一个参数是为了指定网卡（用于多块多卡的时候）

```

知识扩展 PXE 配置文件 default 由于多个客户端可以从一个 PXE 服务器引导，PXE 引导映像使用了一个复杂的配置文件搜索方式来查找针对客户机的配置文件。如果客户机的网卡的 MAC 地址为 8F:3H:AA:6B:CC:5D，对应的 IP 地址为 10.0.0.195，那么客户机首先尝试以 MAC 地址为文件

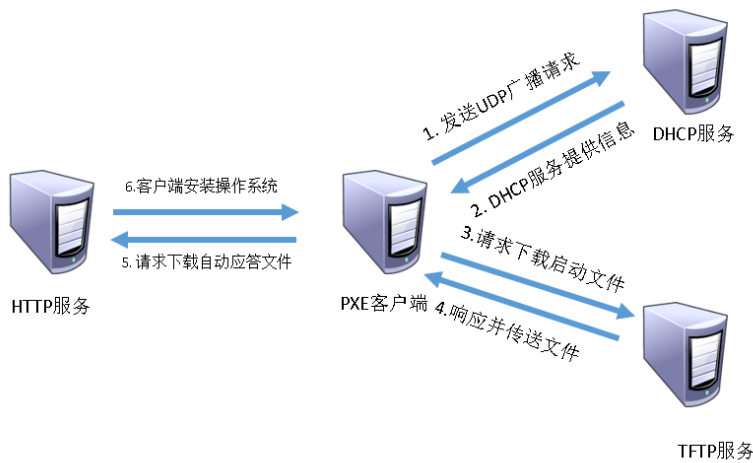
名匹配的配置文件，如果不存在就以 IP 地址来查找。根据上述环境针对这台主机要查找的以一个配置文件就是 /tftpboot/pxelinux.cfg/01-8F:3H:AA:6B:CC:5D。如果该文件不存在，就会根据 IP 地址来查找配置文件了，这个算法更复杂些，PXE 映像查找会根据 IP 地址 16 进制命名的客户机配置文件。例如:10.0.0.195 对应的 16 进制的形式为 C0A801C3。（可以通过 syslinux 软件包提供的 gethostip 命令将 10 进制的 IP 转换为 16 进制）如果 C0A801C3 文件不存在，就尝试查找 C0A801C 文件，如果 C0A801C 也不存在，那么就尝试 C0A801 文件，依次类推，直到查找 C 文件，如果 C 也不存在的话，那么最后尝试 default 文件。总体来说，pxelinux 搜索的文件的顺序是：

```
1 /tftpboot/pxelinux.cfg/01-88-99-aa-bb-cc-dd
2 /tftpboot/pxelinux.cfg/C0A801C3
3 /tftpboot/pxelinux.cfg/C0A801C
4 /tftpboot/pxelinux.cfg/C0A801
5 /tftpboot/pxelinux.cfg/C0A80
6 /tftpboot/pxelinux.cfg/C0A8
7 /tftpboot/pxelinux.cfg/C0A
8 /tftpboot/pxelinux.cfg/C0
9 /tftpboot/pxelinux.cfg/C
10 /tftpboot/pxelinux.cfg/default
```

1.2 Cobbler

yum install dhcp tftp-server xinetd httpd cobbler cobbler-web pykickstart -y



Figure 1.1: pxe step



kickstart https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/installation_guide/s1-kickstart2-options
<https://github.com/webdevops/Dockerfile.git>

Figure 1.2: pxe step



Name	Last modified	Size	Description
 Parent Directory		-	
 CentOS_BuildTag	05-Aug-2015 05:25	14	
 EFI/	05-Aug-2015 05:40	-	
 EULA	27-Nov-2013 17:36	212	
 GPL	27-Nov-2013 17:36	18K	
 Packages/	05-Aug-2015 05:48	-	
 RELEASE-NOTES-en-US.html	25-Jul-2015 20:37	1.3K	
 RPM-GPG-KEY-CentOS-6	27-Nov-2013 17:36	1.7K	
 RPM-GPG-KEY-CentOS-Debug-6	27-Nov-2013 17:36	1.7K	
 RPM-GPG-KEY-CentOS-Security-6	27-Nov-2013 17:36	1.7K	
 RPM-GPG-KEY-CentOS-Testing-6	27-Nov-2013 17:36	1.7K	
 TRANS.TBL	05-Aug-2015 05:50	3.3K	
 images/	05-Aug-2015 05:50	-	
 isolinux/	05-Aug-2015 05:40	-	
 repodata/	05-Aug-2015 05:50	-	

Chapter 2

docker 基础到 kubernetes 集群

2.1 docker 基础

在 gpu-docker 使用 <https://github.com/NVIDIA/nvidia-docker>

2.2 dockerfile 使用总结

使用 dockerfile 心得体会

2.2.1 copy 和 add 的区别

COPY <src> <dest>

add 和 **copy** 都会把 **src** 下面的复制到镜像中。例如 **COPY data /tmp** 便会把 **data** 下面的文件 **copy** 到 **/tmp** 下，如果想把整个目录都复制过去那么就必须写成 **COPY data /tmp/data** 这里 **copy** 和 **add** 是一样的

虽然他们两功能非常像，在官方文档中的 best practices for writing dockerfile 时还是推荐使用 **copy**

Although ADD and COPY are functionally similar, generally speaking, COPY is preferred. That's because it's more transparent than ADD. COPY only supports the basic copying of local files into the container, while ADD has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for ADD is local tar file auto-extraction into the image,

Because image size matters, using ADD to fetch packages from remote URLs is strongly discouraged; you should use curl or wget instead. That way you can delete the files you no longer need after they've been extracted and you won't have to add another layer in your image.

2.2.2 ENTRYPOINT 和 CMD

docker 并不会快照运行的进程，所以通过 **RUN** 命令运行的命令仅在 **docker build** 阶段的时候运行如果需要在容器启动的时候运行服务需要使用 **ENTRYPOINT** 和 **CMD** 来指定，并且这两命令都是放在 **dockerfile** 的最后

并且 **docker** 需要让进程一直处于 **running** 状态（前台，类似 **tail -F**），也就是说不能运行在后台模式，不然 **docker** 会 **exit**，并不会运行除非特殊需求之外，一般一个容器只运行一个服务，也有时候需要运行多个服务，这时候可以有两种方法来解决，一是把两个服务写到同一个 **shell** 里，然后运行，另一种便是使用 **supervisord**, **supervisord** 看起来是比较重的。

shell 示例

supervisord 示例

```
1 FROM ubuntu:latest
2 RUN apt-get update && apt-get install -y supervisor
3 RUN mkdir -p /var/log/supervisor
4 COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
5 COPY my_first_process my_first_process
```

```
6 COPY my_second_process my_second_process
7 CMD ["/usr/bin/supervisord"]
```

ENTRYPOINT 会把 docker run IMAGE 之外的所有参数都传给 ENTRYPOINT 执行的命令中。CMD 则是完全覆盖当 ENTRYPOINT 和 CMD 同时存在的时候 CMD 会做为参数传给 ENTRYPOINT。在 docker run 的时候如果有参数转进来，可以理解为覆盖 CMD 然后把它做为参数传给 ENTRYPOINT。例如

```
1 [root@ns1 test]# cat Dockerfile
2 FROM registry.gsadow.com:5043/centos
3 MAINTAINER from www.gsadow.com by sandow <j.k.yulei@gmail.com>
4
5 #ENTRYPOINT ["echo","entrypoint"]
6 CMD ["echo","cmd"]
7 [root@ns1 test]# docker build -t test .
8 Sending build context to Docker daemon 8.192kB
9 Step 1/3 : FROM registry.gsadow.com:5043/centos
10 ----> 36540f359ca3
11 Step 2/3 : MAINTAINER from www.gsadow.com by sandow <j.k.yulei@gmail.com>
12 ----> Using cache
13 ----> c123904bd244
14 Step 3/3 : CMD echo cmd
15 ----> Running in ea01464d732a
16 ----> 3bcd60c6bda
17 Removing intermediate container ea01464d732a
18 Successfully built 3bcd60c6bda
19 Successfully tagged test:latest
20 [root@ns1 test]# docker run --rm test
21 cmd
22 [root@ns1 test]# docker run --rm test aaa
23 caused "exec: \"aaa\": executable file not found in $PATH"
24 [root@ns1 test]# docker run --rm test echo aaa
25 aaa
26 [root@ns1 test]# cat Dockerfile
27 FROM registry.gsadow.com:5043/centos
28 MAINTAINER from www.gsadow.com by sandow <j.k.yulei@gmail.com>
29
30 ENTRYPOINT ["echo","entrypoint"]
31 CMD ["echo","cmd"]
32 [root@ns1 test]# docker build -t test .
33 [root@ns1 test]# docker run --rm test
34 entrypoint echo cmd
35 [root@ns1 test]# docker run --rm test echo 3
36 entrypoint echo 3
37 [root@ns1 test]# docker run --rm test cmd
38 entrypoint cmd
```

Both CMD and ENTRYPOINT instructions define what command gets executed when running a container. There are few rules that describe their co-operation.

- Dockerfile should specify at least one of CMD or ENTRYPOINT commands.
- ENTRYPOINT should be defined when using the container as an executable.
- CMD should be used as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container.

- CMD will be overridden when running the container with alternative arguments.

The table below shows what command is executed for different ENTRYPOINT / CMD combinations:

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT ["exec_entry", "p1_entry"]
No CMD	error, not allowed	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD ["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

postgresl 官方例中 ENTRYPOINT 是这个样子的

```

1 #!/bin/bash
2 set -e
3
4 if [ "$1" = 'postgres' ]; then
5     chown -R postgres "$PGDATA"
6
7     if [ -z "$(ls -A "$PGDATA")" ]; then
8         gosu postgres initdb
9     fi
10
11     exec gosu postgres "$@"
12 fi
13
14 exec "$@"

```

exec

http://xstarcd.github.io/wiki/shell/exec_redirect.html

su,sudo 经常会需要 TTY 和信号转发行为，它们在设置和使用上比较。gosu，便是在特定的用户下运行特定的程序然后退出管道。<https://github.com/tianon/gosu>

2.3 docker 使用

一般启动方式，以 gitlab 为例

```

1 sudo docker run --detach \
2     --hostname gitlab.example.com \
3     --publish 443:443 --publish 80:80 --publish 22:22 \
4     --name gitlab \
5     --restart always \
6     --volume /srv/gitlab/config:/etc/gitlab \
7     --volume /srv/gitlab/logs:/var/log/gitlab \
8     --volume /srv/gitlab/data:/var/opt/gitlab \
9     gitlab/gitlab-ce:latest

```

2.4 docker compose

```
1 web:
2   image: 'gitlab/gitlab-ce:latest'
3   restart: always
4   hostname: 'gitlab.example.com'
5   environment:
6     GITLAB_OMNIBUS_CONFIG: |
7       external_url 'http://gitlab.example.com:9090'
8       gitlab_rails['gitlab_shell_ssh_port'] = 2224
9   ports:
10    - '9090:9090'
11    - '2224:22'
12   volumes:
13    - '/srv/gitlab/config:/etc/gitlab'
14    - '/srv/gitlab/logs:/var/log/gitlab'
15    - '/srv/gitlab/data:/var/opt/gitlab'
```

docker-compose up -d

2.5 参考链接

[dockerfile_best-practices](#) [what-is-the-difference-between-cmd-and-entrpoint-in-a-dockerfile](#) [what-is-the-difference-between-the-copy-and-add-commands-in-a-dockerfile](#) [ulti-service_container](#)

2.6 kubernetes 组件及基础概念

[kubernetes 基础介绍视频链接](#)

2.6.1 kube-master

- Kubernetes is highly api centered. The Kubernetes API server validates and configures data for the api objects which include pods, services, replicationcontrollers, and others. The API Server services REST operations and provides the frontend to the cluster's shared state through which all other components interact.
- The Kubernetes scheduler is a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity. The scheduler needs to take into account individual and collective resource requirements, quality of service requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, deadlines, and so on. Workload-specific requirements will be exposed through the API as necessary.
- The Kubernetes controller manager is a daemon that embeds the core control loops shipped with Kubernetes. In applications of robotics and automation, a control loop is a non-terminating loop that regulates the state of the system. In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. Examples of controllers that ship with Kubernetes today are the replication controller, endpoints controller, namespace controller, and serviceaccounts controller

The Kubernetes network proxy runs on each node. This reflects services as defined in the Kubernetes API on each node and can do simple TCP,UDP stream forwarding or round robin TCP,UDP forwarding across a set of backends. Service cluster ips and ports are currently found through Docker-links-compatible environment variables specifying ports opened by the service proxy. There is an optional addon that provides cluster DNS for these cluster IPs. The user must create a service with the apiserver API to configure

the proxy. The kubelet is the primary “node agent” that runs on each node. The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn’t manage containers which were not created by Kubernetes.

Other than from an PodSpec from the apiserver, there are three ways that a container manifest can be provided to the Kubelet. File: Path passed as a flag on the command line. Files under this path will be monitored periodically for updates. The monitoring period is 20s by default and is configurable via a flag. HTTP endpoint: HTTP endpoint passed as a parameter on the command line. This endpoint is checked every 20 seconds (also configurable with a flag). HTTP server: The kubelet can also listen for HTTP and respond to a simple API (underspec’d currently) to submit a new manifest.

fluentd is the component which is basically responsible for managing the logs and talking to the central logging mechanism

2.6.2 kube-node

kops 安装, 升级, 管理工具
flanneld

```
1 /usr/lib/sysctl.d/00-system.conf
2 net.bridge.bridge-nf-call-iptables=1
3 net.bridge.bridge-nf-call-ip6tables=1
4 sysctl -w net.ipv4.ip_forward=1
5 sysctl -w net.bridge.bridge-nf-call-ip6tables=1
6 sysctl -w net.bridge.bridge-nf-call-iptables=1
```

0down vote What parameters did you provide for kubeadm? If you want to use flannel as the pod network, specify `--pod-network-cidr 10.244.0.0/16` if you’re using the daemonset manifest below. However, please note that this is not required for any other networks besides Flannel Execute these commands on every node:

2.6.3 pod

pod is a group of one or more containers that are always co-located and co-scheduled that share the context, containers in a pod share the same IP address, ports, hostname, and storage. modeled like a virtual machine: each container represents one process tightly coupled with other containers in the same pod

pod are scheduled in nodes, fundamental unit of deployment in kubernetes.

use cases for pod

2.6.4 Replication Controller

Ensures that a pod or homogeneous set of pods are always up and available Always maintains desired number of pods if there are excess pod, they get killed new pods are launched when they fail, get deleted, or terminated

creating a replication controller with a count of 1 ensures that a pod is always available RC and Pods are associated through labels

2.6.5 replica set

replica set is the advancement to replication controller replica sets are the next generation Replication controller ensures specified numbers of pods are always running Pods are replaced by Replica Sets when a failure occurs labels and selectors are used for associating pods with replica set usually combined with pods when defining the deployment

如果定义的 pod, 删除或者终止后不会重建, 定义成 RC 后, 删除或者终结后还会重建一个
`kubectl scale rc web --replicas=20` 也可以这样来定制 RC

separate statefull containers and stateless containers because stateful containers have very stringent requirements for example i might want to have an SSD based storage that is mounted as a volume

2.6.6 Service

a Service is an abstraction of a logical set of Pods defined by a policy it acts as the intermediary for pods to talk to each other selectors are used for accesing all the pods that match a specific lable service is an object in kubernetes - similar to pods and RCs each Service exposes one of more ports and targetPorts: port will expose to its consumers the targetPorts is how it is going to route the traffic to the destination pods The targetPort is mapped to the port exposed by matching Pods Kubernetes Services Support TCP and UDP portocols

```
kubectl create -f pod.yml -f rc.yml kubectl create -f svc.yml
```

```
kubectl apply -f svc.yml
```

red, green, blue 可以分别定义开发，测试，正式环境，然后通过 svc 提供服务，只需要改变 select 便可以轻松切到某个环境

2.7 kubectl 高级用法

kubectl 来获取特殊的值

```

1
2 获取node name
3  kubectl get nodes -o jsonpath='{range.items[*].metadata}{.name} {end}{'
4
5 获取node ip
6  kubectl get nodes -o jsonpath='{range .items[*].status.addresses[?(@.type=="ExternalIP")]}{.address} {
   end}{'
7
8 一： get 过滤及格式化输出
9  kubectl get pods --all-namespaces -o jsonpath='{..image}'
10 kubectl get pods --all-namespaces -o jsonpath='{.items[*].spec.containers[*].image}'
11
12  * .items[*]: for each returned value
13  * .spec: get the spec
14  * .containers[*]: for each container
15  * .image: get the image
16 上面的一般都不会格式化输出，需要使用range来结合使用
17 kubectl get pods --all-namespaces -o=jsonpath='{range .items[*]}{"\n"}{.metadata.name}{"\t"}{
   range .spec.containers[*]}{.image}{"", " "}{end}{end}{'
18
19 Kubectl run my-web --image=nginx --port=80
20 Kubectl expose deployment my-web --target-port=80 --type=NodePort
21
22 Kubectl get svc my-web -o to-templates='{{{(index .spec.ports 0).nodePort}}}'
23
24 切到另一个集群
25 kubectl config view
26 kubectl config user-context xxxx
27 kubectl config use-context xxxx

```

2.8 定义 pod

2.8.1 限制容器使用资源

创建包含一个容器的 Pod，这个容器申请 100M 的内存，并且内存限制设置为 200M 放在 container 下面

jkjl [Survey2014]

```

1 resources:
2   limits:
3     memory: "200Mi"
4   requests:
5     memory: "100Mi"
```

2.8.2 pod 生命周期与重启策略

A pod (as in a pod of whales or pea pod) is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers. A pod's contents are always co-located and co-scheduled, and run in a shared context. A pod models an application-specific "logical host" - it contains one or more application containers which are relatively tightly coupled—in a pre-container world, they would have executed on the same physical or virtual machine.

Termination-of-pods

The kubelet can optionally perform and react to two kinds of probes on running Containers:

The kubelet uses liveness probes to know when to restart a Container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

The kubelet uses readiness probes to know when a Container is ready to start accepting traffic. A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

livenessProbe Indicates whether the Container is running. If the liveness probe fails, the kubelet kills the Container, and the Container is subjected to its restart policy. If a Container does not provide a liveness probe, the default state is Success.

readinessProbe Indicates whether the Container is ready to service requests. If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the Pod. The default state of readiness before the initial delay is Failure. If a Container does not provide a readiness probe, the default state is Success

Probes have a number of fields that you can use to more precisely control the behavior of liveness and readiness checks:

initialDelaySeconds Number of seconds after the container has started before liveness or readiness probes are initiated.

periodSeconds How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.

timeoutSeconds Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.

successThreshold Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness. Minimum value is 1.

failureThreshold When a Pod starts and the probe fails, Kubernetes will try failureThreshold times before giving up. Giving up in case of liveness probe means restarting the Pod. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

HTTP probes have additional fields that can be set on httpGet:

host Host name to connect to, defaults to the pod IP. You probably want to set “Host” in httpHeaders instead.

scheme Scheme to use for connecting to the host (HTTP or HTTPS). Defaults to HTTP.

path Path to access on the HTTP server.

httpHeaders Custom headers to set in the request. HTTP allows repeated headers.

port Name or number of the port to access on the container. Number must be in the range 1 to 65535. For an HTTP probe, the kubelet sends an HTTP request to the specified path and port to perform the check. The kubelet sends the probe to the pod’s IP address, unless the address is overridden by the optional hostfield in httpGet. If scheme field is set to HTTPS, the kubelet sends an HTTPS request skipping the certificate verification. In most scenarios, you do not want to set the host field. Here’s one scenario where you would set it. Suppose the Container listens on 127.0.0.1 and the Pod’s hostNetwork field is true. Then host, under httpGet, should be set to 127.0.0.1. If your pod relies on virtual hosts, which is probably the more common case, you should not use host, but rather set the Host header in httpHeaders.

configure-liveness-readiness-probes

A Probe is a diagnostic performed periodically by the kubelet on a Container. To perform a diagnostic, the kubelet calls a Handler implemented by the Container. There are three types of handlers:

ExecAction Executes a specified command inside the Container. The diagnostic is considered successful if the command exits with a status code of 0.

TCPSocketAction Performs a TCP check against the Container’s IP address on a specified port. The diagnostic is considered successful if the port is open.

HTTPGetAction Performs an HTTP Get request against the Container’s IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

Each probe has one of three results: Success: The Container passed the diagnostic. Failure: The Container failed the diagnostic. Unknown: The diagnostic failed, so no action should be taken.

A PodSpec has a restartPolicy field with possible values Always, OnFailure, and Never. The default value is Always. restartPolicy applies to all Containers in the Pod. restartPolicy only refers to restarts of the Containers by the kubelet on the same node. Failed Containers that are restarted by the kubelet are restarted with an exponential back-off delay (10s, 20s, 40s ...) capped at five minutes, and is reset after ten minutes of successful execution. As discussed in the Pods document, once bound to a node, a Pod will never be rebound to another node.

pod-lifecycle

Kubernetes supports the postStart and preStop events. Kubernetes sends the postStart event immediately after a Container is started, and it sends the preStop event immediately before the Container is terminated

<https://kubernetes.io/docs/tasks/configure-pod-container/attach-handler-lifecycle-event/> <https://kubernetes.io/docs/concepts/containers/lifecycle-hooks/>

2.8.3 创建 pod 拉取私有库镜像

拉取镜像由 imagePullPolicy 来控制拉取策略，他有三个值 Always IfNotPresent Never

当使用私有库拉镜像的时候需要创建 secret

```
kubectl create secret docker-registry myregistrykey --docker-server=DOCKER_REGISTRY_SERVER
--docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_
EMAIL secret "myregistrykey" created.
```

使用 `yaml`，来创建 `secret` 这时候有一点麻烦，需要先 `docker login` 登录私钥库，可以看到在家目录多出来个 `.docker` 隐藏目录，下面有 `config.json` 文件，这时需要使用 `base64` 加密，在定义 `secret` 时需要把加密后的值连续的赋予给 `data[".dockerconfigjson"]`

```

1 docker login -u admin -p Harbor123 10.10.39.226
2 [root@mobius_004 ~]# cd .docker/
3 [root@mobius_004 .docker]# ls
4 config.json
5 [root@mobius_004 .docker]# cat config.json
6 {
7     "auths": {
8         "10.10.39.226": {
9             "auth": "YWRTaW46SGFyYm9yMTIzNDU="
10        }
11    },
12    "HttpHeaders": {
13        "User-Agent": "Docker-Client/18.01.0-ce (linux)"
14    }
15 }
16 [root@mobius_004 .docker]# base64EncodeData=$(base64 -w 0 config.json)
17 [root@mobius_004 .docker]# echo $base64EncodeData|base64 --decode
18 apiVersion: v1
19 kind: Secret
20 metadata:
21   name: myregistrykey
22   namespace: awesomeapps
23 data:
24   .dockerconfigjson: $base64EncodeData
25
26 type: kubernetes.io/dockerconfigjson
27
28 然后在创建pod的时候指定secrets来拉镜像
29
30 apiVersion: v1
31 kind: Pod
32 metadata:
33   name: foo
34   namespace: awesomeapps
35 spec:
36   containers:
37   - name: foo
38     image: janedoe/awesomeapp:v1
39   imagePullSecrets:
40   - name: myregistrykey

```

<https://kubernetes.io/docs/concepts/containers/images/>

2.9 kubeconfig 配置

`kubeconfig` 在 `kubectl`, `kubelet`, `kube-proxy`, `bootstrap` 都会用到，配置 `kubeconfig` 可以用三种方式通过命令方式

```

1 kubectl config set-cluster kubernetes \
2   --certificate-authority=/etc/kubernetes/ssl/ca.pem \

```

```

3  --embed-certs=true \
4  --server=${KUBE_APISERVER} \
5  --kubeconfig=kube-proxy.kubeconfig
6  # 设置客户端认证参数
7  kubectl config set-credentials kube-proxy \
8  --client-certificate=/etc/kubernetes/ssl/kube-proxy.pem \
9  --client-key=/etc/kubernetes/ssl/kube-proxy-key.pem \
10 --embed-certs=true \
11 --kubeconfig=kube-proxy.kubeconfig
12 # 设置上下文参数
13 kubectl config set-context default \
14 --cluster=kubernetes \
15 --user=kube-proxy \
16 --kubeconfig=kube-proxy.kubeconfig
17 # 设置默认上下文
18 kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig
19 mv kube-proxy.kubeconfig /etc/kubernetes/

```

直接编辑方式

.kube/config 这个配置文件可以指定文件或者指定 base64 值

```

1  apiVersion: v1
2  kind: Config
3  users:
4  - name: kubelet
5    user:
6      client-certificate-data: <base64-encoded-cert>
7      client-key-data: <base64-encoded-key>
8  clusters:
9  - name: local
10    cluster:
11      certificate-authority-data: <base64-encoded-ca-cert>
12  contexts:
13  - context:
14      cluster: local
15      user: kubelet
16    name: service-account-context
17  current-context: service-account-context

```

加密密钥可以使用 base64 /Users/yulei/Documents/ansible/roles/kubernetes/files/ssl/ca.pem 便可得到

To generate the base64 encoded client cert, you should be able to run something like `cat /var/run/kubernetes/kubelet_36kr.pem | base64`. If you don't have the CA certificate handy, you can replace the `certificate-authority-data: <base64-encoded-ca-cert>` with `insecure-skip-tls-verify: true`.

If you put this file at `/var/lib/kubelet/kubeconfig` it should get picked up automatically. Otherwise, you can use the `--kubeconfig` argument to specify a custom location.

或者在 config 里指定文件

```

apiVersion: v1 clusters: - cluster: certificate-authority: /etc/kubernetes/certs/ca.crt server: https://kubernetesmaster
name: default-cluster contexts: - context: cluster: default-cluster user: default-admin name: default-system
current-context: default-system kind: Config preferences: users: - name: default-admin user: client-
certificate: /etc/kubernetes/certs/server.crt client-key: /etc/kubernetes/certs/server.key

```

2.10 生产环境中使用 kubernetes

Kubernetes in prod

<https://techbeacon.com/one-year-using-kubernetes-production-lessons-learned>

<https://github.com/kelseyhightower/confd>

<https://www.graylog.org/>

<https://www.loggly.com/blog/top-5-docker-logging-methods-to-fit-your-container-deployment-strategy/>

<https://medium.com/readme-mic/kubernetes-1-year-in-production-f406bdb95c22>

<https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.9/>

<https://blog.dockbit.com/kubernetes-canary-deployments-for-mere-mortals-6696910a52b2>

<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

<https://www.loggly.com/resource/log-management-handbook-docker/>

<https://medium.com/readme-mic/kubernetes-1-year-in-production-f406bdb95c22>

<https://medium.com/readme-mic/kubernetes-1-year-in-production-f406bdb95c22>

2.10.1 openshift

<https://github.com/openshift/origin>

<http://www.linkedin.com/pulse/part-2-kubernetes-services-minikube-docker-james-denman>

学习集群

<https://www.katacoda.com/courses/kubernetes/playground>

2.11 kubernetes addon

2.11.1 Discovering Services

discovering services - dns the DNS server watches kubernetes API for new Services the DNS server creates a set of DNS records for each Services Services can be resolved by the name within the same namespace Pods in other namespaces can access the Service by adding the namespace to the DNS path my-service.my-namespace

Discovering Services - env vars

Service Types ClusterIP: service is reachable only from inside of the cluster NodePort Service is reachable through NodeIP:NodePort address LoadBalancer service is reachable through an external load balancer mapped to NodeIP:NodePort address

kubernetes create Docker link compatible environment variables in all pods containers can use the environment variable to talk to the service endpoint

<https://segmentfault.com/a/1190000002892825>

2.12 pod 的持久化

persistence in pods

pods are ephemeral and stateless

volumes bring persistence to pods

kubernetes volumes are similar to docker volumes, but managed differently

all containers in pod can access the volumes

volumes are associated with the lifecycle of pod

directories in the host are exposed as volumes

volumes may be based on a variety of storage backends

kubernetes have three method to persistence into your workload

1: basic volume will persistence to you pod with from limitations 2: rely on distributed storage like NFS 3: clear dispersing,

kubernetes volume types

```
1  -- hostbased
2      emptydir
3      hostpath
4  -- block storage
5      amazon EBS
6      GCE Persistent Disk
7      Azure Disk
8      VSphere Volume
9  -- Distributed file System
10     NFS
11     Ceph
12     Gluster
13     Amazon EFS
14  -- other
15     Flocker
16     iScsi
17     Git Repo
18
19
20
21 gcloud container clusters get-credentials jani-gke-demo asia-east1-a
22 gcloud compute disks create --size=10G --zone=asia-east1-a my-data-disk
23 gcloud compute disks delete --zone=asia-east1-a my-data-dis
```

Understanding Persistent Volume and Claims

PersistentVolume(PV) Networked storage in the cluster pre-provisioned by an administrator PersistentVolumeClaim (PVC) Storage resource requested by a user. StorageClass types of supported storage profiles offered by administrator

Chapter 3

自动化管理工具

服务器环境中要想保证其稳定运行, 必不可少的便是标准化, 自动化, 设想任何一个运维人员都是上去手动修改主机配信息, 一旦出故障, 如果此运维人员还在职, 且还记得修改过什么配置, 还可以恢复回来, 但恢复时长也相当长, 这对 IT 管理造成相当大的困难, 公司服务器标准化, 自动化势在必行。

3.1 ansible

Ansible 基于 python 研发的自动化运维工具, ansible 是无客户端也不需要服务端工具, 十分方便, 主要基于 openssl 所以安全性也比较高, 但是因为任务按队列依次执行, 所以并没有 saltstack 那样并发的快. 特别是维护上百台机器后会感觉到明显慢很多。

ansible core : ansible 自身核心模块 host inventory: 主机库, 定义可管控的主机列表 connection plugins: 连接插件, 一般默认基于 ssh 协议连接 modules: core modules (自带模块)、custom modules (自定义模块) playbooks : 剧本, 按照所设定编排的顺序执行完成安排任务
收集配置文件

```
ansible -i environments/fudao/hosts account -m fetch -a 'dest=/data/yulei/ansible-auto/roles/fudao/files/
src=/data/soft/nginx_80/conf/vhost/account.conf'
```

在 play book 里不仅可以指定 vars 变量, 还可以指定 vars 文件

var files

```
- hosts: myhosts vars_files: - default_step.yml
```

3.2 salt

3.2.1 salt 介绍

与 ansible 不同的是, salt 是一个 C/S 架构的软件, salt 管理端为 master, 客户端叫 minion, 通过 server 端下发指令, 客户端受指令的方式进行操作, saltstack 基于 zeromq 消息队列来管理成千上万台主机客户端, 传输指令执行相关操作。采用 RSA key 方式进行身份确认, 传输采用 AES 方式进行加密, 这使得它的安全性得到了保证。

在每个 minion 启动后便会自动生成 RSA 公密钥, 存入于/etc/salt/pki/minion, 中, 根据 minion 配置文件中 master 地址, 主动发送公钥给 master 等待 master 接收, master 接收后便可以批量管理主机。

3.2.2 salt 中 grains 与 pillar

Grains 是 saltstack 组件之一, 记录 saltstack Minion 的一些静态信息的组件, (CPU, 内存, 磁盘, 网络, 等) 可以通过 grains.items 查看某台 minion 的所有 Grain 信息, minion 的 grains 信息会在 minions 启动时汇报给 master, 在实际应用环境中我们需要根据自己的业务需求去定义 grains, 在每次修改完 grains 后需要同步更新 grains. salt '*' saltutil.sync_grains。了解更多关于 grains 函数使用命令查看 salt -E 'client*' sys.list_functions grains

自定义 grains 有三种方法: 第一种在/etc/salt/master 里直接配置,

```

1 grains:
2   roles:
3     - webserver
4     - memcache

```

第二种在另起一个文件/etc/salt/grains 在里面定义,

```

1 roles:
2   - webserver
3   - memcache

```

第三种使用 python 定义在 minion 配置文件中配置 grains 放到任何环境中 _grains 目录下

```

1 [root@linux-node1 /srv/salt/_grains]# cat my_grains.py
2 #!/usr/bin/env python
3 # -*- coding: utf-8 -*-
4
5 def my_grains():
6     # 初始化一个grains字典
7     grains = {}
8     grains['iaas'] = 'openstack'
9     grains['edu'] = 'sadow'
10    return grains
11 [root@linux-node1 /srv/salt/_grains]# cat roles.py
12 #!/usr/bin/env python
13 # -*- coding: utf-8 -*-
14 import os.path
15 def roles():
16     roles_file= "/etc/salt/roles"
17     roles_list= []
18     if os.path.isfile(roles_file):
19         roles_fd = open(roles_file, "r")
20         for eachroles in roles_fd:
21             roles_list.append(eachroles[:-1])
22     return {'roles': roles_list}
23 if __name__ == "__main__":
24     print roles()

```

三种方法优先级为从高到低依次为系统自带, grains 文件配置, master grains.

Pillar

数据管理中心 Pillar Pillar 也是 salt 组件之一, 叫数据管理中心, 或者说是配置管理中心。会经常配合 states 在大规模配置管理工作中使用它, pillar 在 saltstack 中主要的作用就是存储和定义配置管理中需要的一些数据, 比如软件版本号, 用户名, 密码, 配置等信息, 它的定义存储格式跟 grains 类似, 同样增加完 pillar 配置后需要刷新 salt '*' saltutil.refresh_pillar。查看 pillar salt '*' pillar.items master 端配置文件中指定了 pillar 的文件存放位置,

```

1 pillar_roots:
2   base:
3     - /srv/pillar

```

同状态模块一样, 里面需要有 top.sls 指定入口文件, 编写方式也一样。

grains 与 pillar 的区别名称存储位置数据类型数据采集更新方式应用 Grains Minion 端静态数据 Minion 启动时收集, 也可以使用 saltutil.sync_grans 进行刷新存储 Mnion 基本数据, 比如用于匹配

Minion, 自身数据可以用来做资产管理等。Pillar Master 端动态数据在 master 端定义, 指定给对应的 minion, 可以使用 `saltutil.refresh_pillar` 刷新存储 master 指定的数据, 只有指定的 Minion 可以看到。用于敏感数据保存

3.2.3 远程执行

有时候仅需要使用 salt 运行简单的命令, 可以使用 `cmd.run` 模块 **salt TARGET cmd.run 'w'** 目标端 target 指定主机名, 这里可以使用到正则匹配, grains 匹配, pillar 匹配, 主要组, 或者直接列出主机名, 这里的匹配在 `top.sls` 也可以同样适用

目标可以通过正则来匹配 minion id. 或者用 grains, pillar subnet/ip address, compound matching, node groups 来匹配

管理对象 target saltstack 系统中我们的管理对象叫作 target, 在 master 上我们可以采用不同的 target 去管理不同的 minion 在 target options 下可以分很多种匹配方式

分发文件 1. salt-cp 批量分发文件 □ salt-cp 语法格式为 `salt-cp '*' [options] SOURCE DEST`

3.2.4 jinja

在编写状态文件的时候经常会引用变量, grains, pillar, 这时候就需要使用 jinja

变量使用 Grains: `grains['fqdn_ip4']`

变量使用执行模块: `salt['network.hw_addr']('eth0')`

变量使用 Pillar: `pillar['apache']['PORT']`

jinja 模版来写 keepalived 的优写级

```
1 {% if grains['fqdn']== 'lb-node1.unixhot.com' %}
2 - ROUTEID: HAPROXY_MASTER
3 - STATEID: MASTER
4 - PRIORITYID: 101
5 {% elif grains['fqdn']== 'lb-node2.unixhot.com' %}
6 - ROUTEID: HAPROXY_BACKUP
7 - STATEID: BACKUP
8 - PRIORITYID: 100
9 {% endif %}
```

```
1 {% set motd = ['/etc/motd'] %}
2 {% if grains['os'] == 'Debian' %}
3   {% set motd = ['/etc/motd.tail', '/var/run/motd'] %}
4 {% endif %}
5
6 {% for motdfile in motd %}
7   {{ motdfile }}:
8     file.managed:
9       - source: salt://motd
10  {% endfor %}
```

3.2.5 状态模块 state

状态模块描述 minion 端的状态, 按照官网的说明, 往往最强大, 最有用的工程解决方案都是基于简单的原则, (Many of the most powerful and useful engineering solutions are founded on simple principles. Salt States strive to do just that: K.I.S.S. (Keep It Stupidly Simple))

salt state 的核心便是 sls 文件 (salt state file) sys 文件描述了那些系统应该是什么样子。sys 是以 yaml 为格式序列化存储数据, 所以其本质上就是字典, 列表, 数字, 举个例子

```
1 apache:
2   pkg.installed: []
```

```

3  service.running:
4      - enable: True
5      - require:
6          - pkg: apache

```

这个 sls 状态文件将会确保 `apache` 已经安装，并且已经在运行。第一行 `apache` 是这个数据集的 ID，全局惟一，一个 ID 下可以有多个模块，但是不能使用多次使用同一个模块。第二三行表示那些状态模块需要运行。基本模式是 `<state_module>.<function>`，`pkg.installed` 确定当前主机已经安装了指定软件，如果不指定 `pkgs` 则默认安装第一行 ID 名。第三行 `service.running` 表示确保软件已经在运行。如果不指定 `name`，默认以 ID 为软件名。最后两行 `require` 表示 `service.running` 需要依赖于 ID 为 `apache` 下的 `pkg` 模块运行完后才会执行。所以上面可以修改为

```

1  testpkg:
2      pkg.installed:
3          - pkgs:
4              - httpd
5      service.running:
6          - name: httpd
7          - enable: True
8          - require:
9              - pkg: testpkg

```

要想运行状态文件需要在 `/etc/salt/master` 中开始 `file_roots` 配置

```

1  file_roots:
2      base:
3          - /srv/salt
4      dev:
5          - /srv/salt/dev/services
6          - /srv/salt/dev/states
7      prod:
8          - /srv/salt/prod/services
9          - /srv/salt/prod/states

```

`base`, `dev`, `prod` 表示环境，`salt` 默认会去 `base` 环境下去找状态文件，假设把上面内容保存到 `/srv/salt/apache/init.sls`，要运行单个 sls 文件可以使用命令 `salt '*' state.sls apache` 运行，这里的 `apache`, `salt` 会去 `base` 环境下找 `apache.sls`，如果没有，会继续找有没有目录 `apache`，并且下面有 `init.sls`，如果都没有则返回错误。如果要运行 `/srv/salt/apache/install.sls` 最后的状态文件变成 `apache.install` 既可

当 `apache` 目录下有多个 sls 时，可以使用 `include apache.xxx` 来引到当前文件中

但是如果环境比较多，不同的主机运行不同的状态文件，你又不想一次次敲命令，又乱又容易弄错怎么办，这时候就出现 `top.sls`，在整个 `salt` 状态文件里惟一，他定义了针对不同环境下不同主机运行不同的状态文件。默认放到 `base` 环境根目录下，也就是 `/srv/salt` 下。

```

1  base:
2      'os:Fedora':
3          - webserver
4          - match: grain
5  dev:
6      'dev-*':
7          - vim
8      'db*dev*':
9          - db
10 prod:
11     '10.10.200.0/24':
12         - match: ipcidr

```

13 | — deployments.qa.site1

最后使用命令 `salt '*' state.highstate`, 一次搞定。

这里仅列出简单两个模块两个方法的使用法, `salt` 有非常多的模块可以使用, 可以通过命令来获取所有模块以及模块中的方法及用途

查看 `state` 模块 `'Minion'` `sys.list_state_modules`

查看指定 `states(git)` 的所有 functions `salt 'Minion' sys.list_state_functions git`

查看指定 function 的用法 `salt 'Minion' sys.state_doc git.config`

针对管理对象操作 `module` 是我们日常使用 `saltstack` 最多的一个组件, 是用于管理对象操作的, 这也是 `saltstack` 通过 `push` 的方式管理的入口, 比如管理日常简单的执行命令, 查看包安装情况, 查看服务运行情况等都是通过 `module` 来实现的

为什么要加 `require`? 就是因为 `salt` 本身是并发的去处理任务, `service` 和 `pkg` 有可能现时运行, 这样有可能达不到预期结果, 所以加着 `require` 做前后依赖。除 `require` 外, 还有很多条件

`require`: 在执行这一步之前需要满足的东西都列在下面, 不满足就不执行, 可以依赖整个 `sls`, `'require': ['sls':"foo"]`, `require_in` 反过来被谁依赖

`watch` 里面任何一个状态变化变触发, 并不是所有的 `state` 都支持 `watch`, `service state` 能支持, `watch_in` 被谁监控

`unless` 执行下面内容, 如果结果为 `False` 才执行该 `state`, `"unless":["rpm -q vim-enhanced"],"ls /usr/bin/vim"]`, `onlyif` 结果返回为 `True` 才执行该 `state`

`onfail`, 另一个 `state` 执行失败后执行这个, `onchanges` 另一个 `state` 执行成功并且产生变化执行, `prereq` 被要求在 `xxx state` 之前执行, `use` 利用另一个 `state` 的参数, `listen/listen_in` 和 `watch/watch_in` 类似在所有 `state` 最后执行, `include` 组合多个 `state`, `extend` 对之前内容扩展,

3.2.6 salt 实践

环境准备, 使用两台主机做测试, 并且在每一台机子都做 `hosts` 解析

- master 端主机名 `master_101` ip `172.16.1.101`
- minion 端主机名 `client_102` ip `172.16.1.102`

```

1 #安装软件
2 yum install salt-master salt-minion -y
3
4 #master端
5 systemctl start salt-master
6 systemctl enable salt-master
7
8 # minion端
9 sed -i 's/#master: salt/master: 172.16.1.101/g' /etc/salt/minion
10 systemctl start salt-minion
11 systemctl enable salt-minion
12 # 接收 公钥
13 salt-key -a client_102 -y
14 Accepted Keys:
15 client_102
16 Denied Keys:
17 Unaccepted Keys:
18 Rejected Keys:
19 #测试连通性
20 salt 'client_102' test.ping

```

```

21 client_102:
22 True

```

使用 salt-api

```

1 useradd -M -s /sbin/nologin saltapi
2 passwd saltapi
3 [root@linux-node1 /etc/pki/tls/certs]# make testcert
4 cd /etc/pki/tls/private
5 openssl rsa -in localhost.key -out salt_nopass.key
6 systemctl start salt-api
7 pip install CherryPy==3.2.6
8 [root@linux-node1 ~]# vim /etc/salt/master
9 default_include: master.d/*.conf
10 rest_cherry:
11 host: 192.168.56.11
12 port: 8000
13 ssl_crt: /etc/pki/tls/certs/localhost.crt
14 ssl_key: /etc/pki/tls/private/salt_nopass.key
15 external_auth:
16 pam:
17 saltapi:
18 - '*'
19 - '@wheel'
20 - '@runner'
21 [root@linux-node1 ~]# curl -k https://192.168.56.11:8000/login \
22 -H 'Accept: application/x-yaml' \
23 -d username='saltapi' \
24 -d password='saltapi' \
25 -d eauth='pam'
26 return:
27 * eauth: pam □ expire: 1464663850.123221 □ perms: □
28   * *
29   * '@wheel'
30   * '@runner' □ start: 1464620650.123221 □ token: 785db9bc5e79dee828bfb1649bc49c59900e0ebf □
      user: saltapi
31 curl -k https://192.168.56.11:8000/minions/linux-node1.oldboyedu.com \
32 -H 'Accept: application/x-yaml' \
33 -H 'X-Auth-Token: 785db9bc5e79dee828bfb1649bc49c59900e0ebf' \
34 curl -k https://192.168.56.11:8000/ \
35 -H 'Accept: application/x-yaml' \
36 -H 'X-Auth-Token: 785db9bc5e79dee828bfb1649bc49c59900e0ebf' \
37 -d client='runner' \
38 -d fun='manage.status'
39 curl -k https://192.168.56.11:8000/ \
40 -H 'Accept: application/x-yaml' \
41 -H 'X-Auth-Token: 785db9bc5e79dee828bfb1649bc49c59900e0ebf' \
42 -d client='local' \
43 -d tgt='*' \
44 -d fun='test.ping'
45 https://github.com/binbin91/oms

```

Bibliography

- [1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The L^AT_EX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.
- [2] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891–921, 1905.
- [3] Knuth: Computers and Typesetting,
<http://www-cs-faculty.stanford.edu/~uno/abcde.html>
- [4] 丘迟: exec 重定向以及 io 重定向
http://xstarcd.github.io/wiki/shell/exec_redirect.html