

# linux 从入门到放弃

蔚雷

November 28, 1028



# Contents

<b>I</b>	<b>linux 基础知识</b>	<b>3</b>
<b>1</b>	<b>自动化安装系统</b>	<b>5</b>
1.1	kickstart 安装部署步骤 . . . . .	6
1.1.1	创建 ks.cfg 文件 . . . . .	7
1.2	Cobbler . . . . .	8
<b>2</b>	<b>系统基础</b>	<b>11</b>
2.1	centos7 安装 . . . . .	11
2.1.1	桥接 . . . . .	11
2.2	磁盘分区与挂载 . . . . .	12
2.2.1	磁盘简介 . . . . .	12
2.2.2	磁盘分区 . . . . .	12
2.2.3	格式化与挂载 . . . . .	13
2.3	文件描述符及通配符 . . . . .	16
2.3.1	通配符 . . . . .	16
2.4	文件权限 . . . . .	16
2.5	shell . . . . .	17
2.5.1	shell 定义变量以及调用变量 . . . . .	17
2.5.2	在 shell 中的特殊变量名 . . . . .	17
2.5.3	变量赋值与转换 . . . . .	17
2.5.4	shell 里的运算 . . . . .	18
2.5.5	shell 里处理字符 . . . . .	18
2.5.6	shell 中的数组 . . . . .	18
2.5.7	shell 中的 if 判断语法 . . . . .	18
2.5.8	基本方法 . . . . .	19
<b>3</b>	<b>资源迁移</b>	<b>21</b>
3.1	磁盘网络复制 . . . . .	21
3.2	磁盘扩容 . . . . .	21
<b>4</b>	<b>网络基础知识</b>	<b>23</b>
<b>5</b>	<b>vim 语法总结</b>	<b>25</b>
<b>6</b>	<b>keepalive</b>	<b>27</b>
<b>II</b>	<b>存储与数据</b>	<b>29</b>
<b>7</b>	<b>redis 实战</b>	<b>31</b>
7.1	常见经验 . . . . .	31
7.2	redis cluster . . . . .	32

<b>III 虚拟化</b>	<b>33</b>
<b>8 虚拟化与 kvm</b>	<b>35</b>
8.1 虚拟化	35
<b>9 docker 基础到 kubernetes 集群</b>	<b>37</b>
9.1 docker 基础	37
9.2 dockerfile 使用总结	37
9.2.1 copy 和 add 的区别	37
9.2.2 ENTRYPOINT 和 CMD	37
9.3 docker 使用	39
9.4 docker compose	40
9.5 参考链接	40
9.6 kubernetes 组件及基础概念	40
9.6.1 kube-master	40
9.6.2 kube-node	41
9.6.3 pod	41
9.6.4 Replication Controller	41
9.6.5 replica set	41
9.6.6 Service	42
9.7 kubectl 高级用法	42
9.8 定义 pod	43
9.8.1 限制容器使用资源	43
9.8.2 pod 生命周期与重启策略	43
9.8.3 创建 pod 拉取私有库镜像	44
9.9 kubeconfig 配置	45
9.10 生产环境中使用 kubernetes	47
9.10.1 openshift	47
9.11 kubernetes addon	47
9.11.1 Discovering Services	47
9.12 pod 的持久化	47
<b>IV 持续集成与自动化</b>	<b>49</b>
<b>10 自动化工具</b>	<b>51</b>
10.1 ansible	51
10.2 salt	51
10.2.1 salt 介绍	51
10.2.2 salt 中 grains 与 pillar	52
10.2.3 远程执行	53
10.2.4 jinja	53
10.2.5 状态模块 state	54
10.2.6 salt 实践	55
<b>V 监控与数据展示</b>	<b>57</b>
<b>11 elk 简单介绍</b>	<b>59</b>
11.1 elasticsearch	59
11.2 logstash	59
11.2.1 输入 input	59
11.2.2 过滤 filter	59
11.2.3 输出 output	61

11.3 kibana . . . . . 61



# 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，





## **Part I**

# **linux 基础知识**



# 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：

```

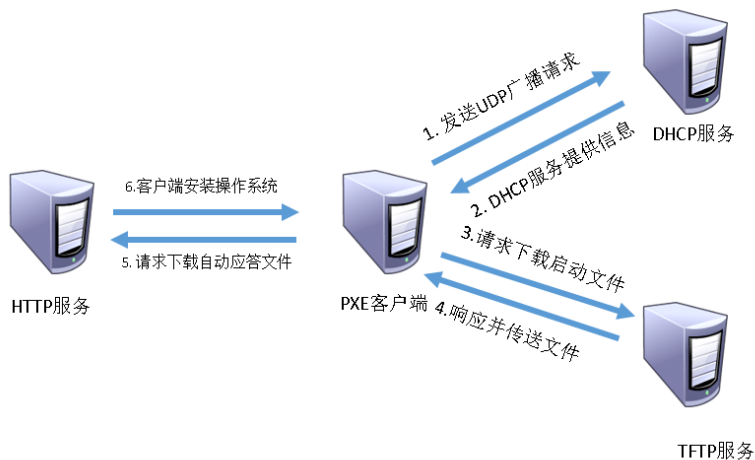
1 /tftpboot/pxelinux.cfg/01-88-99-aa-bb-cc-dd
2 /tftpboot/pxelinux.cfg/C0A801C3
3 /tftpboot/pxelinux.cfg/C0A801C
4 /tftpboot/pxelinux.cfg/C0A801
5 /tftpboot/pxelinux.cfg/C0A80
6 /tftpboot/pxelinux.cfg/C0A8
7 /tftpboot/pxelinux.cfg/C0A
8 /tftpboot/pxelinux.cfg/C0
9 /tftpboot/pxelinux.cfg/C
10 /tftpboot/pxelinux.cfg/default

```

## 1.2 Cobbler

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



Figure 1.1: pxe step



在第一部门中先了解 linux 基础操作知识，他们包括系统知识，网络知识，

Figure 1.2: pxe step



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





## Chapter 2

# 系统基础

### 2.1 centos7 安装

CentOS 7 网卡名不以 eth0 开始原因，是由于 systemd 和 udev 引入了一种新的网络设备命名方式-一致网络设备命名（CONSISTENT NETWORK DEVICE NAMING）。可以根据固件、拓扑、位置信息来设置固定名字，带来的好处是命名自动化，名字完全可预测，在硬件坏了以后更换也不会影响设备的命名，这样可以让硬件的更换无缝化。带来的不利是新的设备名称比传统的名称难以阅读。比如心得名称是 enp5s0。

想要改为像 centos6 一样以 eth0 开始的名称有两种方法，在出现安装新系统的时候按下 tab 键，在 kernel 启动选项中增加 net.ifnames=0 biosdevname=0

安装好后需要先做一些优化，fir



Figure 2.1: Asynchronous I/O model

当然，如果在安装时忘记操作了也可以在启动后操作。在 /etc/sysconfig/grub 下相应位置加上这两个参数，然后再改网卡名既可

```
1 GRUB_CMDLINE_LINUX="rd.lvm.lv=vg0/swap vconsole.keymap=us crashkernel=auto vconsole.font=
  latarcyrheb-sun16 net.ifnames=0 biosdevname=0 rd.lvm.lv=vg0/usr rhgb quiet"
2
3 grub2-mkconfig -o /boot/grub2/grub.cfg
```

安装完系统后一般需要关闭 selinux, NetworkManager, firewallld, 需要安装 net-tools, lsof, tcpdump, epel,

#### 2.1.1 桥接

一、网卡桥接设置：

- 网卡配置文件: 详见 ifcfg-enp8s0
- 网桥配置文件: 详见 ifcfg-br0

二、网卡绑定设置:

- 网卡配置文件 01: ifcfg-enp6s0f0
- 网卡配置文件 02: ifcfg-enp6s0f1
- 网桥配置文件: ifcfg-bond0

3. 在 bond0 基础上增加多个桥接网卡详情参见 ifcfg-bond0.300, ifcfg-virbr300; ifcfg-bond0.3960, ifcfg-virbr3960

## 2.2 磁盘分区与挂载

### 2.2.1 磁盘简介

用于存储数据的物理设备便叫磁盘, 磁盘接接口的不同可以分为: IDE, SATA, SCSI, SAS.

- IDE 的英文全称为“Integrated Drive Electronics”, 即“电子集成驱动器”,
- SCSI 的英文全称为“Small Computer System Interface”
- SATA (Serial ATA) 又叫串口硬盘, PC 机硬盘的主流趋势。
- SAS(Serial Attached SCSI) 即串行连接 SCSI, 是新一代的 SCSI 技术, 此接口的设计是为了改善存储系统的效能、可用性和扩充性, 并且提供与 SATA 硬盘的兼容性

磁盘内部由多个盘片, 机械手臂, 磁头, 主轴马达组成。在读取数据时主轴马达驱动盘片转动, 机械手臂可伸展来让磁头 (head) 读取数据, 在盘片上存储数据, 所以磁盘的容量便要看盘片的质量。

**磁道 (Track)** : 在每个盘片上由不同半径组成的同心圆叫做磁道,

] 扇区 (Sector)] : 每个磁道中被分隔成的最小的俱单位便是扇区每个扇区为 512bytes

**柱面 (Cylinder)** : 由多个盘片相同磁道所组成的圆柱面便是柱面

磁盘读取与写入数据时会按柱面来写入, 只有柱面写完 (读完) 后才会切换磁道。磁盘容量计算方式:  $\text{Head} * \text{cylinder} * \text{Secor} * 512\text{bytes}$

每个磁盘的第一扇区非常重要, 因为在该扇区存放着两个重要信息 1. 主引导分区 (Master Boot Record, MBR) 有 446bytes 主要用于安装引导加载程序 2. 分区表 (Partition table) 记录整块磁盘分区状态, 有 64bytes

### 2.2.2 磁盘分区

由于分区表仅有 64bytes 所以只能记录 4 组记录区, 每组记录区记录了该区段的启始与结束的柱面号码。所以每个磁盘只能分四个主 (Primary) 或扩展分区 (Extended)。而每个磁盘只允许有一个扩展分区, 且扩展分区不能被格式化后存放数据, 需要在扩展分区之上划分逻辑分区 (Logical)。linux 设备中的文件名 1-4 给主或者扩展分区预留, 逻辑分区从 5 开始。

在 linux 下给磁盘分区的命令有 ‘fdisk’ 适合小于 2T 的磁盘分区, ‘parted’ 擅长于大于 2t 的磁盘分区。分区的实质便是修改分区表

### fdisk 对磁盘分区

使用命令 ‘fdisk -cu /dev/sdb’ 来进行对 sdb 分区，用命 -l 来查看分区分区时的命令有

- d delete a partition 删除一个分区
- n add a new partition 新增一个分区
- p print the partition table 把分区打印出来
- q quit without saving changes 不保存退出
- w write table to disk and exit 保存分区并退出

这里需要注意在交互式分区过程中输错之后需要用 **\*\*Ctrl + u\*\*** 来撤消。交互式实在太费事，这对于批量分区来说太费事可以用下面命令来一键搞定

```
1 echo -e "\n\n1\n\n10G\n\n\n2\n\n20G\n\nw" | fdisk /dev/sdb
```

上一便仅是创建两个主分区，第一个分区给 10G，第二个分区给 20G，创建其它分区也类似

### parted 对磁盘分区

parted 的操作都是实时的，也就是说你执行了一个分区的命令，他就实实在在地分区了，而不是像 fdisk 那样，需要执行 w 命令写入所做的修改，所以进行 parted 的测试千万注意不能在生产环境中！> 传统的 MBR(Master Boot Record) 分区方式，有一个局限：无法支持超过 2TB 的硬盘的分区（或单个分区超过 2TB）如果大于 2T 就要使用用 GPT(Globally Unique Identifier Partition Table Format) 分区概念。

非交互式分区方式

```
1 parted /dev/sdb mklabel gpt yes
2 parted /dev/sdb mkpart primary ext4 0 100 Ignore
3 parted /dev/sdb mkpart primary linux--swap 101 8192 Ignore
4 parted /dev/sdb mkpart logical ext4 8193 100GB Ignore
5 parted /dev/sdb mkpart logical ext4 101GB 3000GB Ignore
6 parted /dev/sdb quit
```

### 2.2.3 格式化与挂载

磁盘分区好后，必须先格式化后才能挂载

```
1 $ mkfs.ext4 /dev/sdb1
2 $ mkfs.ext4 /dev/sdb2
3
4 $ tune2fs -c -1 /dev/sdb1
5 tune2fs 1.41.12 (17-May-2010)
6 Setting maximal mount count to -1
7 #格式化后便可以挂载了
8 $ mount /dev/sdb1 /mnt
9 $ mount |grep --color=auto "/dev/sdb1"
10 /dev/sdb1 on /mnt type ext4 (rw)
```

在这里手动挂载后，系统重启后还需要再手动挂载一次，因为这里需要修改文件/etc/fstab 这个文件以达到开机自动挂载

磁盘被手动挂载之后都必须把挂载信息写入/etc/fstab 这个文件中，否则下次开机启动时仍然需要重新挂载。系统开机时会主动读取/etc/fstab 这个文件中的内容，根据文件里面的配置挂载磁盘。这样我们只需要将磁盘的挂载信息写入这个文件中我们就不需要每次开机启动之后手动进行挂载了。挂载的限制

- 根目录是必须挂载的，而且一定要先于其他 mount point 被挂载。因为 mount 是所有目录的跟目录，其他木有都是由根目录 / 衍生出来的。
- 挂载点必须是已经存在的目录。
- 挂载点的指定可以任意，但必须遵守必要的系统目录架构原则
- 所有挂载点在同一时间只能被挂载一次
- 所有分区在同一时间只能挂在一次
- 若进行卸载，必须将工作目录退出挂载点（及其子目录）之外。

下面我们看看/etc/fstab 文件，这是我的 linux 环境中/etc/fstab 文件中的内容

```

1 $ cat /etc/fstab
2 #
3 # /etc/fstab
4 # Created by anaconda on Wed Oct 28 23:23:38 2015
5 #
6 # Accessible filesystems, by reference, are maintained under '/dev/disk'
7 # See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
8 #
9 UUID=faba0886-9c24-430c-8ce5-f7980c283bbd / ext4 defaults 1 1
10 UUID=a2ea9c91-9424-4d8e-b15f-946ef8413877 /boot ext4 defaults 1 2
11 UUID=f11549e2-cd8a-4ec5-92ca-e8a83a16c87e swap swap defaults 0 0
12 tmpfs /dev/shm tmpfs defaults 0 0
13 devpts /dev/pts devpts gid=5,mode=620 0 0
14 sysfs /sys sysfs defaults 0 0
15 proc /proc proc defaults 0 0
16 /dev/sdb1 /mnt ext4 defaults 0 0

```

可以看到 fstab 里一共有六列。

第一列 Device **\*\*Device\*\*** 是磁盘设备文件或者该设备的 Label 或者 UUID Label 就是分区的标签，在最初安装系统是填写的挂载点就是标签的名字。可以通过查看一个分区的 superblock 中的信息找到 UUID 和 Label name。例如我们要查看/dev/sda1 这个设备的 uuid 和 label name 使用设备名称 (/dev/sda) 来挂载分区时是被固定死的，一旦磁盘的插槽顺序发生了变化，就会出现名称不对应的问题。因为这个名称是会改变的。不过使用 label 挂载就不用担心插槽顺序方面的问题。不过要随时注意你的 Label name。至于 UUID，每个分区被格式化以后都会有一个 UUID 作为唯一的标识号。使用 uuid 挂载的话就不用担心会发生错乱的问题了。

```

1 $ dumpe2fs -h /dev/sda1
2 dumpe2fs 1.35 (28-Feb-2004)
3 Filesystem volume name: /boot #这个就是Label name
4 Last mounted on:
5 Filesystem UUID: 3b10fe13-def4-41b6-baae-9b4ef3b3616c #UUID
6 Filesystem magic number: 0xEF53
7 Filesystem revision #: 1 (dynamic)
8 Filesystem features: has_journal ext_attr resize_inode dir_index filetype needs_recovery sparse_super
9 Default mount options: (none)
10 Filesystem state: clean
11 #简单点的方式我们可以通过下面这个命令来查看
12 $ blkid /dev/sda1
13 /dev/sda1: LABEL="/boot" UUID="3b10fe13-def4-41b6-baae-9b4ef3b3616c" SEC_TYPE="ext3"
    TYPE="ext2"

```

第二列：Mount point 设备的挂载点，就是你要挂载到哪个目录下。

第三列：filesystem 磁盘文件格式，包括 ext2、ext3、ext4、reiserfs、nfs、vfat 等。生产场景中如果是大量小文件业务首选 reiserfs。而 ext4 适合视频下载，流媒体，数据库，小文件业务。

ReiserFS 是一个基于 B 状树的文件系统，拥有非常好的总体性能，特别是对于大量小文件。ReiserFS 拥有良好的伸缩性并具有日志功能。但该文件系统不再受到积极开发，不支持 SELinux，基本上已被 Reiser4 取代。ReiserFS 文件系统多年来一直用作一些发行版（包括 SUSE）的默认文件系统，但现在用得少了。

XFS 文件系统拥有日志功能，包含一些健壮的特性，并针对可伸缩性进行了优化。XFS 在 RAM 中强制缓存中转数据，因此如果使用 XFS，建议采用不间断电源供应。淘宝的数据库在使用此文件系统。

第四列：parameters 文件系统的参数

**Async/sync** 设置是否为同步方式运行，默认为 async

**auto/noauto** 当挂载 mount -a 的命令时，此文件系统是否被主动挂载。默认为 auto

**rw/ro** 是否以只读或者读写模式挂载

**exec/noexec** 限制此文件系统内是否能够进行“执行”的操作

**user/nouser** 是否允许用户使用 mount 命令挂载

**suid/nosuid** 是否允许 SUID 的存在

**Usrquota** 启动文件系统支持磁盘配额模式

**Grpquota** 启动文件系统对群组磁盘配额模式的支持

**Defaults** 同时具有 rw,suid,dev,exec,auto,nouser,async 等默认参数的设置

第五列：能否被 dump 备份命令作用，dump 是一个用来作为备份的命令。通常这个参数的值为 0 或者 1，0 代表不要做 dump 备份，1 代表要每天进行 dump 的操作，2 代表不定日期的进行 dump 操作。

第六列是否检验扇区开机的过程中，系统默认会以 fsck 检验我们系统是否为完整（clean）。0 不要检验，1 最早检验（一般根目录会选择），2 1 级别检验完成之后进行检验。

以上会用到的命令会另一篇文章专门介绍下面仅罗列一些相关的命令

格式：mkfs, tune2fs, dumpe2fs

挂载：mount umount /etc/fstab

磁盘检查，df, fsck, e2fsck

调整文件大小 resize2fs

分区：fdisk parted, partprobe, dd

给 swap 增加容量

```
1 dd if=/dev/zero of=/tmp/swap bs=1M count=128
2 mkswap /tmp/swap
3 swapon /tmp/swap
```

机械磁盘读写磁盘数据的原理小结：1. 磁盘是按照柱面为单位读写数据的，既先读取同一个盘面的某一个磁道，读完之后如果数据没有读完，磁头也不会切换到其他的磁道，而是选择切换磁头，读取下一个盘面相同半径的磁道，直到所有盘面的相同半径的磁道读取完成之后，如果数据还没有读写成，才会切换其他不同半径的磁道，这个切换磁道的过程称为寻道。2. 不同磁头间的切换是电子切换，而不同磁道间的切换需要磁头做径向运动，这个径向运动需要不进行电机调节，这个运行是机械的切换。

第一个硬盘第二个硬盘第三个硬盘，使用硬件 RAID，LVM 等工成一个或者多个虚拟磁盘，在系统中以块设备名体现/dev/sda

/dev/sdb 等，在进行使用之前需要进行格式化（创建虚拟文件系统，不同系统使用的文件系统不一样，xfs,ext3,ext4），每一个分区都有各自的 inode 与 block. 以供系统使用。

英文单词，Head 磁头，Sector 扇区，Track 磁道，Cylinder 柱面，Units 单元块，Block 数据块，Inode 索引节点 buffer: 一般用于写操作，写缓冲

Raid0 是条带化，把多个磁盘合起来组成一个大磁盘支持 1 块到多块盘，容量是所有磁盘之和，读写速度最快，没有冗余 Raid1 只支持偶数盘，镜像盘。读写性能一般，成本高 Raid5 奇偶校验盘，最少三块，可以块一块盘，写入恨不能不高 Raid10，先做 Raid1 然后再做 Raid0，保存备份以及数据量，最少 4 块盘，读写性能快，成本高。

## 2.3 文件描述符及通配符

### 2.3.1 通配符

普通命令都可以用的特殊符号，不同的通配符有不同的意义，现在简单介绍在 linux 中不同的通配符的不同意义。

简单举例

```
1  mkdir /etc/{bbc, blog}
2  echo {a..z}
3
4  a=1
5  printf "%a\n" #输出是1
6  printf '%a\n' #输出结果是 $a
7  echo date #输出结果是当时时间长格式
```

i link 硬链接 i\_count 进程

删除文件需要看所在目录是否有写权限，没有写权限时是无法删除目录下面的文件

## 2.4 文件权限

chmod 只有文件的属主或 root 才能来改变文件权限。

创建目录默认 755 创建文件默认 644

umask 修改默认权限通过八进制的数值来定义用户创建文件或目录的默认权限

sed -n '65.69p' /etc/bashrc

666-umask 若 umask 部分位为基数，那么在结果的相应位置加一

777-umask

umask 对应数值表示的是禁止的权限，

特殊权限位(用户权限位)

以下内容不重要：

suid s(x) S 4 sgid s(x) S 2 sticky t(x) T 1 沾滞位只能用 ROOT 来删除或创建。被创建的目录，任何用户可以在该目录下可以创建文件目录，但不能查看其它用户的内容，(/tmp)

授权方法 chmod (4000|2000|1000) /bin/rm chmod (u|g|o)+(s|t)

seuid 权限位当二进制命令执行修改的是命令而非文件仅对二进制命令才有作用。suid 权限仅在程序命令执行过程中有效。suid 是比较危险的功能，

sgid 是针对用户组权限位的对文件来说，sgid 的功能如下

sgid 仅对二进制的命令程序有效二进制命令或程序需要有可执行权限执行命令在任意用户可以获得该命令程序执行期间所属组的权限。

sgid 针对目录创建一个目录，要求在其目录下创建的文件或目录的 group 继承该目录组 chmod 2755 /home/admins/

设置 seuid chmod 4755 /bin/rm

find / -perm 4755 -type f

更改文件属主与组 chown chgrp

```
chown owner:group directory|file chown :group directory|file chown owner directory|file
groupadd test -g 501
```

## 2.5 shell

可执行文件开头第一行一般我们会指定用什么解释器来执行该文件比如 shell 脚本的文件开头一般会加 `#!/bin/sh`

### 2.5.1 shell 定义变量以及调用变量

运行 shell 时会遇到三种变量 1. 局部变量, 在脚本或命令中定义, 仅在当前 shell 实例中有效, 其他 shell 启动的程序不能访问局部变量。2. 环境变量, 所有的程序, 包括 shell 启动的程序, 都能访问环境变量, 有些程序需要环境变量来保证其正常运行。必要的时候 shell 脚本也可以定义环境变量。3. shell 变量, 是由 shell 程序设置的特殊变量。shell 变量中有一部分是环境变量, 有一部分是局部变量, 这些变量保证了 shell 的正常运行

定义变量时, 变量名开始必须以 `[a-zA-Z]` 开始, 中间不可以有空格或标点符号 (可以用“\_”), 变量名不可以使用 `bash` 的关键字。调用变量, 只需要在变量名前加“\$”便可以了, 考虑到解释器识别边界的问题, 一般我们会在变量名外加大括号来确定变量名删除变量可以用 `'unset'` 来取消变量的定义。

现在我们便创建一个 `test.sh` 文件并且给它执行权限。可以做测试大括号 (花括号) 是为了让解释器识别变量名的边界, 如果不加的话变量名就成了 `$myAgeyears` 这个变量名为空, 输出来的便只有 `Today, I'm old` 这样与期望值并不一样。

### 2.5.2 在 shell 中的特殊变量名

变量	含义
<code>\$0</code>	当前脚本的文件名
<code>\$n</code>	传递给脚本或函数的参数。 <code>n</code> 是一个数字, 表示第几个参数, 例 <code>\$1</code> 。如果超过 10 便需要写成 <code>\$10</code>
<code>\$#</code>	传递给脚本或函数的参数个数。
<code>\$*</code>	传递给脚本或函数的所有参数。
<code>@</code>	传递给脚本或函数的所有参数。被双引号 ("" ) 包含时, 与 <code>\$*</code> 稍有不同, 下面将会讲到。
<code>?</code>	上个命令的退出状态, 或函数的返回值。
<code>\$\$</code>	当前 Shell 进程 ID。对于 Shell 脚本, 就是这些脚本所在的进程 ID。

现在我们接着在 `test.sh` 里第 17-22 行内容, 执行 `./test.sh hello world` 后会打印的内容为

```
1 ./tesh.sh
2 hello
3 world
4 hello world
5 hello world
6 2
```

### 2.5.3 变量赋值与转换

形式	说明
<code>\$var</code>	变量本来的值
<code>\$var:-word</code>	如果变量 <code>var</code> 为空或已被删除 (unset), 那么返回 <code>word</code> , 但不改变 <code>var</code> 的值。
<code>\$var:=word</code>	如果变量 <code>var</code> 为空或已被删除 (unset), 那么返回 <code>word</code> , 并将 <code>var</code> 的值设置为 <code>word</code> 。
<code>\$var:?message</code>	如果变量 <code>var</code> 为空或已被删除 (unset), 那么将消息 <code>message</code> 送到标准错误输出, 可以写为 <code>&gt;&amp;2</code>
<code>\$var:+word</code>	如果变量 <code>var</code> 被定义, 那么返回 <code>word</code> , 但不改变 <code>var</code> 的值。

### 2.5.4 shell 里的运算

在原生 `bash` 中不支持简单的数学运算，但是可以通过其他命令来实现，例如 `awk` 和 `expr`，`expr` 最常用。在使用 `expr` 时的格式为 ‘`expr 1 + 2`’ 关系运算

运算符	说明	举例
<code>-eq</code>	检测两个数是否相等, 相等返回 <code>true</code>	<code>[ \$a -eq \$b ]</code> 返回 <code>true</code>
<code>-ne</code>	检测两个数是否相等, 不相等返回 <code>true</code>	<code>[ \$a -ne \$b ]</code> 返回 <code>true</code>
<code>-gt</code>	检测左边的数是否大于右边的, 如果是, 则返回 <code>true</code>	<code>[ \$a -gt \$b ]</code> 返回 <code>false</code>
<code>-lt</code>	检测左边的数是否小于右边的, 如果是, 则返回 <code>true</code>	<code>[ \$a -lt \$b ]</code> 返回 <code>true</code>
<code>-ge</code>	检测左边的数是否大等于右边的, 如果是, 则返回 <code>true</code>	<code>[ \$a -ge \$b ]</code> 返回 <code>false</code>
<code>-le</code>	检测左边的数是否小于等于右边的, 如果是, 则返回 <code>true</code>	<code>[ \$a -le \$b ]</code> 返回 <code>true</code> 。

#### 逻辑运算

运算符	说明	举例
<code>!</code>	非运算, 表达式为 <code>true</code> 则返回 <code>false</code>	<code>[ false ]</code> 返回 <code>true</code>
<code>-o</code>	或运算, 有一个表达式为 <code>true</code> 则返回 <code>true</code>	<code>[ \$a -lt 20 -o \$b -gt 100 ]</code> 返回 <code>true</code>
<code>-a</code>	与运算, 两个表达式都为 <code>true</code> 才返回 <code>true</code>	<code>[ \$a -lt 20 -a \$b -gt 100 ]</code> 返回 <code>false</code>

#### 字符串运算符

运算符	说明	举例
<code>=</code>	检测两个字符串是否相等, 相等返回 <code>true</code>	<code>[ \$a = \$b ]</code> 返回 <code>false</code> 。
<code>!=</code>	检测两个字符串是否相等, 不相等返回 <code>true</code>	<code>[ \$a != \$b ]</code> 返回 <code>true</code>
<code>-z</code>	检测字符串长度是否为 0, 为 0 返回 <code>true</code>	<code>[ -z \$a ]</code> 返回 <code>false</code>
<code>-n</code>	检测字符串长度是否为 0, 不为 0 返回 <code>true</code>	<code>[ -n \$a ]</code> 返回 <code>true</code>
<code>str</code>	检测字符串是否为空, 不为空返回 <code>true</code>	<code>[ str \$a ]</code> 返回 <code>true</code>

#### 文件测试运算符

操作符	说明	举例
<code>-b file</code>	检测文件是否是 ** 块设备 ** 文件, 如果是, 则返回 <code>true</code>	<code>[ -b \$file ]</code> 返回 <code>true</code>
<code>-c file</code>	检测文件是否是 ** 字符设备 ** 文件, 如果是, 则返回 <code>true</code>	<code>[ -c \$file ]</code> 返回 <code>true</code>
<code>-d file</code>	检测文件是否是 ** 目录 ** , 如果是, 则返回 <code>true</code>	<code>[ -d \$file ]</code> 返回 <code>true</code>
<code>-f file</code>	检测文件是否是 ** 普通文件 ** (既不是目录, 也不是设备文件), 如果是, 则返回 <code>true</code>	<code>[ -f \$file ]</code> 返回 <code>true</code>
<code>-g file</code>	检测文件是否 ** 设置了 SGID 位 ** , 如果是, 则返回 <code>true</code>	<code>[ -g \$file ]</code> 返回 <code>true</code>
<code>-k file</code>	检测文件是否设置了 ** 粘着位 (Sticky Bit) ** , 如果是, 则返回 <code>true</code>	<code>[ -k \$file ]</code> 返回 <code>true</code>
<code>-p file</code>	检测文件是否是具名 ** 管道 ** , 如果是, 则返回 <code>true</code>	<code>[ -p \$file ]</code> 返回 <code>true</code>
<code>-u file</code>	检测文件是否设置了 ** SUID 位 ** , 如果是, 则返回 <code>true</code>	<code>[ -u \$file ]</code> 返回 <code>true</code>
<code>-r file</code>	检测文件是否 ** 可读 ** , 如果是, 则返回 <code>true</code>	<code>[ -r \$file ]</code> 返回 <code>true</code>
<code>-w file</code>	检测文件是否 ** 可写 ** , 如果是, 则返回 <code>true</code>	<code>[ -w \$file ]</code> 返回 <code>true</code>
<code>-x file</code>	检测文件是否 ** 可执行 ** , 如果是, 则返回 <code>true</code>	<code>[ -x \$file ]</code> 返回 <code>true</code>
<code>-s file</code>	检测文件是否 ** 为空 ** (文件大小是否大于 0), 不为空返回 <code>true</code>	<code>[ -s \$file ]</code> 返回 <code>true</code>
<code>-e file</code>	检测文件 (包括目录) ** 是否存在 ** , 如果是, 则返回 <code>true</code>	<code>[ -e \$file ]</code> 返回 <code>true</code>

### 2.5.5 shell 里处理字符串

先定义一个变量: `string="sadow is a gentleman"`

我们可以计算字符串长度 `echo ${#string}`,

或者实现切片 `${string:1:4}` 幸运的是这里 `index` 都是以 0 开头。也可以查找字符 `expr index "$string" sadow`

### 2.5.6 shell 中的数组

在 `shell` 中只可以建立一维数组, 并且 `index` 从 0 开始, 创建数组用小括号。类似这样 `array_name=(value0 value1 value2 value3)`

读取数组中某个值时可以用 `$array_name[index]` 读取所有值可以用 “\*” 或者 “@” 计算长度仅需要 `array_name` 前加 “#” 与之前一样

### 2.5.7 shell 中的 if 判断语法



```

1  if [ expression 1 ]
2  then
3      Statement(s) to be executed if expression 1 is true
4  elif [ expression 2 ]
5  then
6      Statement(s) to be executed if expression 2 is true
7  elif [ expression 3 ]
8  then
9      Statement(s) to be executed if expression 3 is true
10 else
11     Statement(s) to be executed if no expression is true
12 fi

```

### 2.5.8 基本方法

**case** 与 excel 里的 **case** 类似，取值先匹配每一个模式，模式匹配后，刚执行匹配模式相应命令，而不会继续其他模式。如果无一匹配模式，使用星号“\*”来捕获该值，再执行后面的命令。**case** 的值后面必须为‘关键字 **in**’，每一模式必须以左括号结束，取值可以为变量或常数。匹配发现聚会符合某一模式后，其间所有命令开始执行直至遇到‘;;’，结束。

```

1  case var in
2  pattern1)
3      command1
4      command2
5      command3
6      ;;
7  pattern2)
8      command1
9      command2
10     command3
11     ;;
12 *)
13     command1
14     command2
15     command3
16     ;;
17 esac

```

#### for 语法

```

1  for 变量 in 列表
2  do
3      command1
4      command2
5      ...
6      commandN
7  done

```

#### while 语法

```

1  while command
2  do
3      Statement(s) to be executed if command is true
4  done

```

## until 语法

```
1 until command
2 do
3     Statement(s) to be executed until command is true
4 done
```

函数语法，**function** 可有可无，不过做为一个合格的编程人员，有必要加上的。这才是规范。

```
1 function function_name () {
2     list of commands
3     [ return value ]
4 }
```

向函数内传递文件和上面一样 ‘function\_name p1 p2 p3’ 然后在函数内部用 \$n 调用

# Chapter 3

## 资源迁移

### 3.1 磁盘网络复制

dd 是一个非常强大的命令，可以直接复制块文件，经常用于磁盘复制，其复制速度比一般 copy,mv 快的多。曾经使用此命令恢复引导分区。

dd 复制整个磁盘 dd if=/dev/sda of=/dev/sdb. 如果要通过网络把本地磁盘复制到另一端，可以这样 dd if=/dev/sda |ssh root@servername.net "dd of=/dev/sdb", 但是由于 ssh 是基于加密传输，传输速度会相当慢。但如果使用 netcat，基于 tcp sockets 传输那传输数据。下面是有关基于 ssh, nc 压缩与不压缩数据的测试。传输的为 10G 的分区。

	Time Elapsed (Sec)	Speed (MB/s)
Over SSH	1787.4	6.1
Over Netcat (no compression)	1622.4	6.6
Over Netcat (bzip compression)	889.3	12.1
Over Netcat (16M block size + bzip)	490.0	21.9
	Time Savings (Seconds)	Percentage Savings
Over SSH	-	0%
vs Netcat (no compression)	165.0	9%
vs Netcat (bzip compression)	898.1	50%
vs Netcat (16M block size + bzip)	1297.3	73%

在服务端开始 nc 服务 nc -l 19000|bzip2 -d|dd bs=16M of=/dev/sdb 在客户端开始向服务端传输数据 dd bs=16M if=/dev/sda|bzip2 -c|nc serverB.example.net 19000

备份系统 sudo rsync -aXv / --exclude="/dev/\*","/proc/\*","/sys/\*","/tmp/\*","/run/\*","/mnt/\*","/media/\*","/lost+found" root@servername.net:/ dd if=/dev/sda bs=1 count=2048|ssh root@servername.net "dd bs=1 of=/dev/sda"

### 3.2 磁盘扩容

有时候在针对磁盘分区的时候会故意留下一部分空白分区，以供后来不同用途进行再分区挂载。但是当根分区磁盘不够用时，需要把空白分区容量部分分配给根，鉴于此可以使用下面方法扩展根分区。

操作步骤以 CentOS 6.5 64bit 50GB 系统盘为例，root 分区在最末尾分区 (e.g: /dev/xvda1: swap,/dev/xvda2: root) 的扩容场景。

执行以下命令，查询当前弹性云服务器的分区情况。

```
1 [root@sluo-ecs-5e7d ~]# parted -l /dev/xvda
2 Model: Xen Virtual Block Device (xvd)
3 Disk /dev/xvda: 53.7GB
4 Sector size (logical/physical): 512B/512B
5 Partition Table: msdos
6
7 Number Start End Size Type File system Flags
8 1 1049kB 4296MB 4295MB primary linux-swaps(v1)
```

```

9 2 4296MB 42.9GB 38.7GB primary ext4 boot
10 [root@sluo-ecs-5e7d ~]# blkid
11
12 /dev/xvda1: UUID="25ec3bdb-ba24-4561-bcdc-802edf42b85f" TYPE="swap"
13 /dev/xvda2: UUID="1a1ce4de-e56a-4e1f-864d-31b7d9dfb547" TYPE="ext4"

```

安装 `growpart` 工具。yum install cloud-utils-growpart. 工具 `growpart` 可能集成在 `cloud-utils-growpart/cloud-utils/cloud-initramfs-tools/cloud-init` 包里，可以直接执行命令 `yum install cloud-*` 确保 `growpart` 命令可用即可。

执行以下命令，使用工具 `growpart` 将第二分区的根分区进行扩容。

```

1 [root@sluo-ecs-5e7d ~]# growpart /dev/xvda 2
2 CHANGED: partition=2 start=8390656 old: size=75495424 end=83886080 new: size=96465599,end
   =104856255
3 # 执行以下命令，检查在线扩容是否成功。
4 [root@sluo-ecs-5e7d ~]# parted -l /dev/xvda
5 Model: Xen Virtual Block Device (xvd)
6 Disk /dev/xvda: 53.7GB
7 Sector size (logical/physical): 512B/512B
8 Partition Table: msdos
9
10 Number Start End Size Type File system Flags
11 1 1049kB 4296MB 4295MB primary linux-swaps(v1)
12 2 4296MB 53.7GB 49.4GB primary ext4 boot
13
14
15 [root@sluo-ecs-a611 ~]# resize2fs -f /dev/xvda2
16 resize2fs 1.42.9 (28-Dec-2013)
17 Filesystem at /dev/xvda2 is mounted on /; on-line resizing required
18 old_desc_blocks = 3, new_desc_blocks = 3

```

## Chapter 4

# 网络基础知识

一个 hub 就是一个冲突域，同一时间只能有一台 pc 在发送数据包，其他 pc 只能监听网络，直到网络空载 csma-cd

交换机：一个端口就是一个冲突域，广播域。

tcp 面向连接的网络协议，打电话 udp 无连接网络协议，不可靠传输



## Chapter 5

# vim 语法总结

批量删除，以批量注释命令模式下按 **ctrl + v** 移动光标选中需要修改的行，然后按 **x** 或者 **d** 删除该内容或者按 **I** 或 **A** 进入编辑模式然后增加内容然后按 **esc** 退出后便可以看到所选区域都有显示  
vim 下在编辑模式下也有自动补全功能便不是 **tab** 键，而是 **ctrl + n**, **ctrl + o** 跳到上一次修改的地方





## Chapter 6

# keepalive

LVS 是 Linux Virtual Server 的简称, 也就是 Linux 虚拟服务器, 是一个由章文嵩博士发起的自由软件项目, 它的官方站点是 [www.linuxvirtualserver.org](http://www.linuxvirtualserver.org)。现在 LVS 已经是 Linux 标准内核的一部分, 在 Linux2.4 内核以前, 使用 LVS 时必须重新编译内核以支持 LVS 功能模块, 但是从 Linux2.4 内核以后, 已经完全内置了 LVS 的各个功能模块, 无需给内核打任何补丁, 可以直接使用 LVS 提供的各种功能。

(1) LVS 是四层负载均衡, 也就是说建立在 OSI 模型的第四层——传输层之上, 传输层上有我们熟悉的 TCP/UDP, LVS 支持 TCP/UDP 的负载均衡。因为 LVS 是四层负载均衡, 因此它相对于其它高层负载均衡的解决办法, 比如 DNS 域名轮流解析、应用层负载的调度、客户端的调度等, 它的效率是非常高的。

(2) LVS 的转发主要通过修改 IP 地址 (NAT 模式, 分为源地址修改 SNAT 和目标地址修改 DNAT)、修改目标 MAC (DR 模式) 来实现。

□ NAT 模式: 网络地址转换



## **Part II**

# 存储与数据



## Chapter 7

# redis 实战

RemoteDictionary Server 是一个基于 Key-value 键值对的持久化数据库数据库存储系统，redis 和 memcached 缓存服务器很像，但是 redis 支持的数据存储类型更丰富，包括 String，list，set 和 zset 等。redis 可以持久化缓存，还会周期性的把更新的数据写入到磁盘以及把修改的操作记录追加到文件里记录下来，比 memcached 更有优势的是 redis 还支持 master-slave 主从同步。

redis 支持两种不同的方式对数据持久化 RDB,AOF。RDB 在特定时间对数据集做快照存储，其格式为二进制，AOF 是把每一个操作追加到文件中，当服务重启后会按顺序对命令重新执行一次重建数据。当两种持久化都开启时优先使用 AOF 文件恢复原始数据

### 7.1 常见经验

1.Master 写内存快照，save 命令调度 rdbSave 函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以 Master 最好不要写内存快照。

2.Master AOF 持久化，如果不重写 AOF 文件，这个持久化方式对性能的影响是最小的，但是 AOF 文件会不断增大，AOF 文件过大会影响 Master 重启的恢复速度。

3.Master 调用 BGREWRITEAOF 重写 AOF 文件，AOF 在重写的时候会占大量的 CPU 和内存资源，导致服务 load 过高，出现短暂服务暂停现象。4. Master 最好不要做任何持久化工作，包括内存快照和 AOF 日志文件，特别是不要启用内存快照做持久化。如果数据比较关键，某个 Slave 开启 AOF 备份数据，策略为每秒同步一次。

5. 为了主从复制的速度和连接的稳定性，Slave 和 Master 最好在同一个局域网内。尽量避免在压力较大的主库上增加从库

6. 为了 Master 的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master<-Slave1<-Slave2<-Slave3……，这样的结构也方便解决单点故障问题，实现 Slave 对 Master 的替换，也即，如果 Master 挂了，可以立马启用 Slave1 做 Master，其他不变。

下面是我的一个实际项目的情况，大概情况是这样的：一个 Master，4 个 Slave，没有 Sharding 机制，仅是读写分离，Master 负责写入操作和 AOF 日志备份，AOF 文件大概 5G，Slave 负责读操作，当 Master 调用 BGREWRITEAOF 时，Master 和 Slave 负载会突然陡增，Master 的写入请求基本上都不响应了，持续了大概 5 分钟，

上面的情况本来不会也不应该发生的，是因为以前 Master 的这个机器是 Slave，在上面有一个 shell 定时任务在每天的上午 10 点调用 BGREWRITEAOF 重写 AOF 文件，后来由于 Master 机器 down 了，就把备份的这个 Slave 切成 Master 了，但是这个定时任务忘记删除了，就导致了上面悲剧情况的发生，原因还是找了几天才找到的。将 no-appendfsync-on-rewrite 的配置设为 yes 可以缓解这个问题，设置为 yes 表示 rewrite 期间对新写操作不 fsync，暂时存在内存中，等 rewrite 完成后再写入。最好是不开启 Master 的 AOF 备份功能。Redis 主从复制的性能问题，第一次 Slave 向 Master 同步的实现是：Slave 向 Master 发出同步请求，Master 先 dump 出 rdb 文件，然后将 rdb 文件全量传输给 slave，然后 Master 把缓存的命令转发给 Slave，初次同步完成。第二次以及以后的同步实现是：Master 将变量的快照直接实时依次发送给各个 Slave。不管什么原因导致 Slave 和 Master 断开重连都会重复以上过程。Redis 的主从复制是建立在内存快照的持久化基础上，只要有 Slave 就一定会有内存快照发生。虽然 Redis 宣称主从复制无阻塞，但由于磁盘 io 的限制，如果 Master 快照文件比较大，那么 dump 会耗费比较长的时间，这个过程中 Master 可能无法响应请求，也就是说服务会中断，对于关键服务，这个后果也是很可怕的。

单点故障问题，由于目前 Redis 的主从复制还不够成熟，所以存在明显的单点故障问题，这个目前只能自己做方案解决，如：主动复制，Proxy 实现 Slave 对 Master 的替换等，这个也是 Redis 作者目前比较优先的任务之一，作者的解决方案思路简单优雅

## 7.2 redis cluster

统计生产上比较大的 key `./redis-cli -bigkeys`, 不要对压力大的实例上运行命令，  
针对 Redis cluter 增加节点并重新分 slot 槽到新节点

```
1 yum -y install ruby rubygems
2 ~/redis-3.0.7/src/redis-trib.rb add-node 10.10.32.57:7000 10.10.32.27:7001
3 ~/redis-3.0.7/src/redis-trib.rb add-node --slave 10.10.32.57:7001 10.10.32.27:7000
4
5 ./src/redis-trib.rb reshard --timeout 1600 reshard --from redis_id1 redis_id2 --to redis_new--
6 slots 3000 10.10.32.27:7000
```

## **Part III**

# 虚拟化





## Chapter 8

# 虚拟化与 kvm

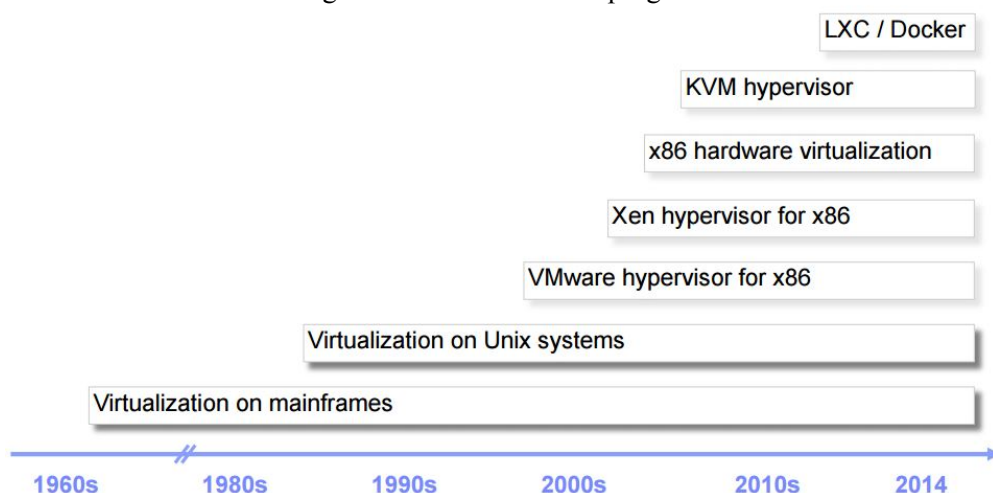
KVM 全称是基于内核的虚拟机 (Kernel-based Virtual Machine)，它是 Linux 的一个内核模块，该内核模块使得 Linux 变成了一个 Hypervisor：KVM 是基于虚拟化扩展 (Intel VT 或者 AMD-V) 的 X86 硬件的开源的 Linux 原生的全虚拟化解决方案。

KVM 中，虚拟机被实现为常规的 Linux 进程，由标准 Linux 调度程序进行调度；虚机的每个虚拟 CPU 被实现为一个常规的 Linux 线程。这使得 KVM 能够使用 Linux 内核的已有功能。但是，KVM 本身不执行任何硬件模拟，需要用户空间程序通过 /dev/kvm 接口设置一个客户机虚拟服务器的地址空间，向它提供模拟 I/O，并将它的视频显示映射回宿主的显示屏。目前这个应用程序是 QEMU。

### 8.1 虚拟化

X86 操作系统是设计在直接运行在裸硬件设备上的，因此它自动认为完全占有计算机硬件，X86 平台的指令集权限划分为特权模式：Ring0, Ring1, Ring2, Ring3。操作系统使用 Ring0 级别；应用程序使用 Ring3 级别。驱动程序使用 Ring1, Ring2 级别。应用程序不能做受控操作，如果需要做，比如要访问硬盘，写文件，那就要通过执行系统调用，执行系统调用的时候 CPU 的运行级别发生 3 到 0 的切换，并跳转到系统调用对应的内核代码位置执行，内核就为你完成设备访问，完成之后再从 0 返回 3，这个过程也称为用户态的内核态的切换。X86 平台在虚拟化方面的一个难点就是如何将虚拟机越级的指令使用进行隔离。

Figure 8.1: virtualization progress



软件虚拟化、基于二进制翻译的全虚拟化 (full Virtualization with Binary Translation)：客户操作系统运行在 Ring1，它运行特权指令时，会触发异常 (CPU 机制，没有权限的指令会触发异常) 然后 VMM (Virtual Machine Manager) 捕获这个异常，在异常里面做翻译，模拟，最后返回到客户操作系统内，客户操作系统认为自己特权指令工作正常，继续运行。但这个性能损耗大。典型厂商 VMware

```

1 yum install qemu-kvm qemu-img
2 yum install virt-install bridge-utils libvirt
3 systemctl start libvirtd
4
5 qemu-img create -f qcow2 centos-7.1.qcow2 50g
6 virt-install --name Centos-7.1-x86_64-base --virt-type kvm \
7   --memory 1024 --vcpus 1 \
8   --cdrom /data/iso/CentOS-7-x86_64-DVD-1503-01_2.iso \
9   --disk path=/base_image/centos-7.1.qcow2,bus=virtio \
10  --network network=br0,model=virtio \
11  --graphics vnc,listen=0.0.0.0 --noautoconsole

```

下面有一个坑，network=br0, 会去找 libvirt 里 network 里定义的网络，如果使用 bridge, 则需要把 network=br0 改为 bridge=br0 既可如果使用现有 qcow2 做磁盘，就不需要使用 cdrom, 直接使用 -boot hd 既可

创建 windows

```

1 virt-install --name zbddzx_ceshi-05 --ram 8192 --cpus 2 \
2   --cdrom=/Data/Base_images/2012R2.iso \
3   --disk path=/opt/zbddzx_ceshi-05.qcow2,bus=virtio \
4   --graphics vnc,listen=0.0.0.0 \
5   --network bridge=virbr300,model=virtio \
6   --noautoconsole

```

guestfish 里面包含很多非常有用的工具，比如 virt-copy-in 可以把宿主机的文件直接 copy 进主机 virt-copy-in /etc/selinux/config -a /Data/Base\_image/CentOS-6.6.qcow2 /etc/selinux/

安装 KVM 后都会发现网络接口里多了一个叫做 virbr0 的虚拟网络接口, 一般情况下，虚拟网络接口 virbr0 用作 nat，以允许虚拟机访问网络服务，但 nat 一般不用于生产环境。我们可以使用以下方法删除 virbr0

1、先使用 virsh net-list 查看所有的虚拟网络：virsh net-list 2、卸载与删除 virbr0 虚拟网络接口先关闭 virsh net-destroy default 再删除 virsh net-undefine default

从一个 xml 文件定义 default 网络, 执行如下命令: virsh net-define /var/lib/libvirt/network/default.xml

1、设置 virbr0 自动启动，执行如下命令：virsh net-start default

2、自启动 virsh net-autostart default

<https://github.com/webdevops/Dockerfile.git>

## Chapter 9

# docker 基础到 kubernetes 集群

### 9.1 docker 基础

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

### 9.2 dockerfile 使用总结

使用 dockerfile 心得体会

#### 9.2.1 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.

#### 9.2.2 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 的时候如果有参数传进来，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](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>

## 9.3 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

```

## 9.4 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`

## 9.5 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](#)

## 9.6 kubernetes 组件及基础概念

[kubernetes 基础介绍视频链接](#)

### 9.6.1 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

### 9.6.2 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:

### 9.6.3 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

### 9.6.4 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

### 9.6.5 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

### 9.6.6 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

## 9.7 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

```



## 9.8 定义 pod

### 9.8.1 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"
```

### 9.8.2 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/>

## 9.8.3 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/>

## 9.9 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

```

## 9.10 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>

### 9.10.1 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>

## 9.11 kubernetes addon

### 9.11.1 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>

## 9.12 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

## **Part IV**

# 持续集成与自动化





# Chapter 10

## 自动化工具

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。

### 10.1 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

我们可以使用 fetch 模块来收集配置文件, 在 play book 里不仅可以指定 vars 变量，还可以指定 vars 文件, var files

```
1 - hosts: myhosts
2   vars_files:
3     - default_step.yml
```

### 10.2 salt

#### 10.2.1 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。

### 10.2.2 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 `Minion` 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

### 10.2.3 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`

### 10.2.4 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 %}

```

### 10.2.5 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.sys 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,

### 10.2.6 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

## **Part V**

# 监控与数据展示





# Chapter 11

## elk 简单介绍

### 11.1 elasticsearch

elasticsearch 其功能主要用于存储与搜索 `curl -XGET 'http://localhost:9200/_nodes'`  
删除日志, `curl -XDELETE 127.0.0.1:9200/fudao_requestlog-2017.12.10`  
列出所有索引并显示状态与索引大小。 `curl -XGET 10.9.199.212:9200/_cat/indices`

### 11.2 logstash

logstash 是一个轻量, 开源的服务端数据处理管道, 能够收集来自各种来源的数据, 实时转换并将其发送到目标位置。其包含三个部分, `input,filter,output`, 下面分三部分一一介绍

#### 11.2.1 输入 input

采集各种样式、大小和来源的数据, 数据往往以各种各样的形式, 或分散或集中地存在于很多系统中。Logstash 支持各种输入选择, 可以在同一时间从众多常用来源捕捉事件, 支持的输入 `plugins`, 里面有各种详细介绍, 这里就简单介绍一个用 `redis` 做为 `input` 的例子

```
1  input {  
2    redis {  
3      data_type => "pattern_channel"  
4      key => "logstash-*"  
5      host => "192.168.0.2"  
6      port => 6379  
7      threads => 5  
8      type => "redis-test"  
9    }  
10 }
```

#### 11.2.2 过滤 filter

数据从源传输到存储库的过程中, Logstash 过滤器能够解析各个事件, 识别已命名的字段以构建结构, 并将它们转换成通用格式, 以便更轻松、更快速地分析和实现商业价值。利用 `grok` 从非结构化数据中派生了结构, 从 IP 地址破译出地理坐标, `filter plugins`

针对不同来源, 不同类型数据做不同处理及输出, 可以在 `input` 中增加 `type` 这个字段, 然后在 `filter,output` 的时候使用 `type` 这个字段来判断来源是什么, 怎么处理, 怎么输出。也可以使用 `input` 过来默认增加的字段来处理。

在 logstash 里字段都需要使用 `[fieldname]` 来进行处理, 字段分为 `top-level(agent, ip, request, ua, response)`, `nested filed(status, bytes, os)`。在指定 `nested filed` 的时候可以 `[ua][os]`

```
1  filter{
```

```

2      if[source]=~"ftp.log"{
3          grok{
4              match=>{
5                  "message"=>[
6                      "\[%{TIMESTAMP_ISO8601:timestamp}\] ALL AUDIT: User \[%{GREEDYDATA:
                          userId}\] \[%{GREEDYDATA:var}\] \[%{HOSTNAME:ip}\] \[%{
                          GREEDYDATA:event}\].",
7                      "\[%{TIMESTAMP_ISO8601:timestamp}\] ALL AUDIT: User \[%{GREEDYDATA:
                          userId}\] \[%{GREEDYDATA:event}\] \[%{GREEDYDATA:filename}\]."]
8                  ]
9              }
10
11
12          if[event]=~"retrieving file"{
13              add_tag=>["Download"]
14          }else if[event]=~"storing file"{
15              add_tag=>["Upload"]
16          }else if[event]=~"has logged in"{
17              add_tag=>["Login"]
18          }
19          add_tag=>["log_ftp"]
20      }
21
22  }
23

```

在处理数据的时候经常会用到 if 判断，判断的表达式有，==, !=, <, >, <=, >=, 正常 =, !, in, not in, and, or, nand, xor, ! 意思是否定的意思，多个表达试可以使用 () 括起来。

```

1
2  if EXPRESSION {
3      ...
4  } else if EXPRESSION {
5      ...
6  } else {
7      ...
8  }

```

使用 mutate 来移除一个字段

```

1  filter {
2      if [action] == "login" {
3          mutate { remove_field => "secret" }
4      }
5  }

```

使用 in 判断, 在 add tag 后, 可以使用 if "aaa" in [tags] 来判断 tags 里是否有 aaa

```

1  filter {
2      if [foo] in [foobar] {
3          mutate { add_tag => "field in field" }
4      }
5      if [foo] in "foo" {
6          mutate { add_tag => "field in string" }
7      }
8      if "hello" in [greeting] {

```

```

9     mutate { add_tag => "string in field" }
10   }
11   if [foo] in ["hello", "world", "foo"] {
12     mutate { add_tag => "field in list" }
13   }
14   if [missing] in [alsomissing] {
15     mutate { add_tag => "shouldnotexist" }
16   }
17   if !("foo" in ["hello", "world"]) {
18     mutate { add_tag => "shouldexist" }
19   }
20 }

```

### 11.2.3 输出 output

尽管 Elasticsearch 是我们的首选输出方向，能够为我们的搜索和分析带来无限可能，但它并非唯一选择，比如 `exec`, `file`, `hadoop` 等，详见 [output plugins](#)

在 `output`，也支持 `sprintf` format，类似像统计一个状态数量便可以用到这个。平进经常用到的类似时间格式了，`path => "/var/log/%type. %{+yyyy.MM.dd.HH}"` 详细请参见 [在配置中使用事件数据和字段](#)

```

1 output {
2   statsd {
3     increment => "apache. %{[response][status]}"
4   }
5 }

```

使用 `exec` 做为 `output`，运行自定义脚本，需要先安装插件 `bin/logstash-plugin install logstash-output-exec`

```

1 output {
2   if [type] == "abuse" {
3     exec {
4       command => "iptables -A INPUT -s %{clientip} -j DROP"
5     }
6   }
7 }

```

如果要在 `logstash` 使用 `geoip` 可以安装相应插件 `bin/logstash-plugin install logstash-filter-geoip` 并下载最新静态 IP 地址库到本地。<https://dev.maxmind.com/geoip/geoip2/geolite2/> 下载 GeoLite2 City maxmindDB 放到指定目录

## 11.3 kibana