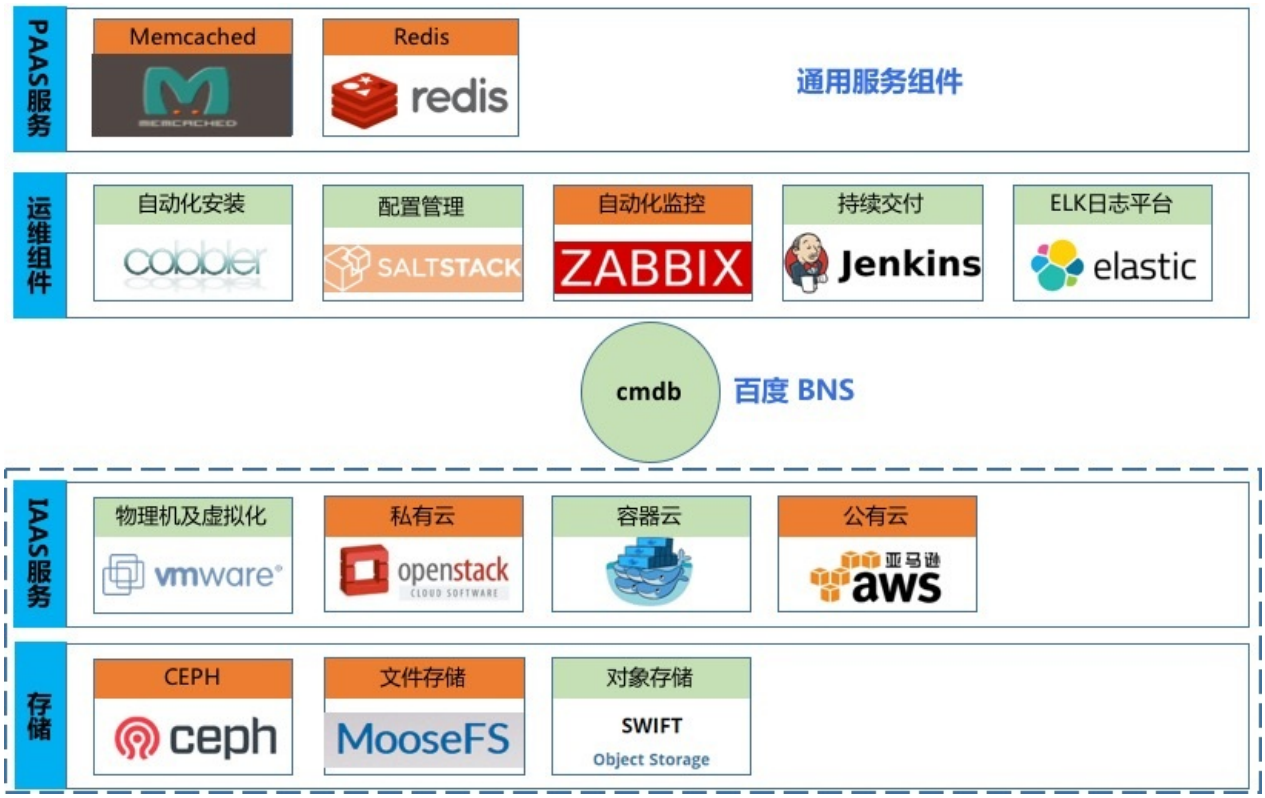
Table of Contents

运维实践指南



阅读本书

- 网上阅读(gitbooks)----gitbooks 网络被墙的厉害，可以通过下面链接在 github 上阅读
- 网上阅读(github)
- 下载本书(pdf)

相关内容

- 文档规范
- wiki
- 相关程序下载
- 点击进行反馈

参加步骤

- 在 GitHub 上 `fork` 到自己的仓库，然后 `clone` 到本地，并设置用户信息。

```
$ git clone https://github.com/meetbill/op_practice_book.git
$ cd op_practice_book
$ git config user.name "yourname"
$ git config user.email "your email"
```

- 修改代码后提交，并推送到自己的仓库。

```
$ #do some change on the content
$ git commit -am "Fix issue #1: change helo to hello"
$ git push
```

- 在 GitHub 网站上提交 pull request。
- 定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/meetbill/op_practice_book.git
$ git fetch upstream
$ git checkout master
$ git rebase upstream/master
$ git push -f origin master
```

小额捐款

如果觉得 `op_practice_book` 对您有帮助，可以请笔者喝杯咖啡（支付宝）

运维实践指南

此笔记中记录着学习点滴，希望可以帮到更多的人。

如果这其中有些是你正困惑的地方，那么此笔记也许能帮到你，如果有好的建议，戳这 进行提交下建议

Linux 篇

- Linux 基础
- Linux 工具
- Linux 安全
- Linux 优化
- 脚本编程 (shell)
- 常见服务架设
- 常用问题处理

Linux 基础

安装

CentOS 6.x

安装准备

```
1. 下载安装镜像文件
http://www.centos.org   ->downloads->mirrors
http://mirrors.aliyun.com/centos/6.8/isos/x86_64/
http://mirrors.aliyun.com/centos/6.8/isos/i386/
主要下载 Centos-6.8-x86_64-bin-DVD1.iso 和 Centos-6.8-x86_64-bin-DVD2.iso
```

安装 CentOS6.8

选择系统引导方式

```
选择 install or upgrade an existing system
```

检查安装光盘介质

```
选择：skip
```

选择安装过程语言

```
选择：english
```

选择键盘布局

```
选择：U.S.English
```

选择合适的物理设备

```
选择：basic storage devices
```

初始化硬盘提示

```
选择：yes ,discard and data
```

初始化主机名以及网络配置

```
（1）. 为系统设置主机名   主机名为：meetbill
（2）. 配置网卡及连接网络（可选）
```

系统时钟及时区

```
选择：Asia/Shanghai
取消：system clock uses UTC
然后：next
```

设置 ***root*** 口令

磁盘分区类型选择与磁盘分区配置过程

(1) 选择系统安装磁盘空间类型

```
选择：create custom layout
```

(2) 进入 'create custom layout'分区界面

```
可以 create （创建）,update（修改） ,delete（删除）等操作。
```

(3) 按企业生产标准定制磁盘分区

```
选择：standard partition
1）. 创建引导分区，/boot 分区
mount point:/boot
file system type:ext4
size:200

2）. 创建 swap 交换分区
mount point :<not applicable>
file system type:swap
size:1024 （物理内存的 1-2 倍）
addtion size options : fixed size
force to be a primary partition

3). 创建 ( / ) 根分区
mount point :/
file system type : ext4
size : 剩余
addtion size options : fill to maximum allowable size  （根分区是最后一个分区，所以把剩余的空间都分配给根分区）
force to be a primary partition

4). 格式化警告
选择： format
```

系统安装包的选择与配置

(1) 启动引导设备的配置 系统默认使用 GRUB 作为启动加载器，引导程序默认在 MBR 下：

```
```
install boot loader on /dev/sda ->change device
选择 master boot record -/dev/sda
[选择的是操作系统所在的那个设备，如 /dev/sda]

Boot Loader operation system list
列表中选择的是操作系统根目录 / 所在的分区，如 CentOS /dev/sda4

```
```

(2) 系统安装类型选择及自定义额外包组 系统默认是 desktop ，但是这里选择 minimal。 自定义安装包选择：customsize now base system : base 然后：next

开始安装 ***->***安装完成 ***->reboot***

系统安装后的配置

更新系统，打补丁到最新

```
修改更新 yum 源：
cp  /etc /yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.ori
wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirros.163.com/.help/CentOS 6-Base-16
3.repo
ll /etc/pki/rpm-gpg/
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY*
yum update -y

ps: 一般在首次安装时执行 yum update -y , 如果是在实际生产环境中，切记使用，以免导致异常。
```

安装额外的软件包

```
yum install tree telnet dos2unix sysstat lrzsz nc nmap -y
yum grouplist    #查看包组列表
yum frouplist "development Tools"
```

Bash 基础特性

命令历史

(1) 使用命令：history

(2) 环境变量：

```
    a) HISTSIZE：命令历史缓冲区中记录的条数，默认为 1000；

    b) HISTFILE：记录当前登录用户在 logout 时历史命令存放文件；

    c) HISTFILESIZE：命令历史文件记录历史的条数，默认为 1000；
```

(3) 操作命令历史：

```
    a) history –d OFFSET 删除指定行的命令历史；

    b) history –c 清空命令历史缓冲区中的命令；

    c) history # 显示历史中最近的#条命令；

    d) history –a 手动追加当前会话缓冲区中的命令至历史文件中；
```

(4) 调用历史中的命令：

```
a) !# : 重复执行第#条命令；

b) !! : 重复执行上一条（最近一条命令；)

c) !string : 重复执行最近一次以指定字符串开头的命令；

d) 调用上一条命令的最后一个参数：

     i. !$

     ii. ESC, .
```

(5) 控制命令历史的记录方式：

```
环境变量：HISTCONTROL

三个值：

    ignoredups：忽略重复的命令；所谓重复，一定是连续且完全相同，包括选项和参数；

    ignorespace：忽略所有以空白开头的命令，不记录；

    ignoreboth：忽略上述两项，既忽略重复的命令，也忽略空白开头的命令；
```

- 修改环境变量的方式：

```
export 变量名 =“VALUE”

或： VARNAME=“VALUE” export VARNAME
```

命令补全

内部命令：直接通过 shell 补全； 外部命令：bash 根据 PATH 环境变量定义的路径，自左而右地在每个路径搜寻以给定命令命名的文件，第一次找到即为要执行的命令；

- Note: 在第一次通过 PATH 搜寻到命令后，会将其存入 hash 缓存中，下次使用不再搜寻 PATH，从 hash 中查找；

```
[root@sslinux ~]# hash
hits command
 1 /usr/sbin/ifconfig
 1 /usr/bin/vim
 1 /usr/bin/ls
```

Tab 键补全： 若用户给出的字符在命令搜索路径中有且仅有一条命令与之相匹配，则 Tab 键直接补全；

若用户输入的字符在命令搜索路径中有多条命令与之相匹配，则再次 Tab 键可以将这些命令列出；

路径补全

以用户输入的字符串作为路径开头，并在其指定路径的上级目录下搜索以指定字符串开头的文件名；

```
如果唯一，则直接补全；

否则，再次 Tab，列出所有符合条件的路径及文件；
```

命令行展开

1）~ ：展开为用户的主目录；

```
[root@sslinux log]# pwd
/var/log
[root@sslinux log]# cd ~
[root@sslinux ~]# pwd
/root
```

2）~USERNAME ： 展开为指定用户的主目录；

```
[root@sslinux ~]# pwd
/root
[root@sslinux ~]# cd ~sslinux
[root@sslinux sslinux]# pwd
/home/sslinux
```

3）{} ： 可承载一个以逗号分隔的列表，并将其展开为多个路径；

```
[root@localhost test]# ls
[root@localhost test]# mkdir -pv ./tmp/{a,b}/shell
mkdir: created directory `./tmp'
mkdir: created directory `./tmp/a'
mkdir: created directory `./tmp/a/shell'
mkdir: created directory `./tmp/b'
mkdir: created directory `./tmp/b/shell'
[root@localhost test]# mkdir -pv ./tmp/{tom,johnson}/hi
[root@localhost test]# tree .

└── tmp
 ├── a
 │ └── shell
 ├── b
 │ └── shell
 ├── johnson
 │ └── hi
 └── tom
 └── hi
9 directories, 0 files
```

命令的执行状态结果

表示命令是否成功执行；

bash 使用特殊变量 $? 保存最近一条命令的执行状态结果；

- 环境变量 $? 的取值：

  0 ： 成功；

  1-255：失败，1,127,255 为系统保留；

- 程序执行有两类结果：

  程序的返回值；程序自身执行的输出结果；

  程序的执行状态结果；$?

```
[root@localhost test]# ls /etc/sysconfig/

[root@localhost test]# echo $?

0    #程序的执行状态结果；执行成功；

[root@localhost test]# ls /etc/sysconfig/NNNN

ls: cannot access /etc/sysconfig/NNNN: No such file or directory    #程序自身的执行结果；

[root@localhost test]# echo $?

2    #执行失败；
```

命令别名

- 通过 alias 命令实现：

a、alias ： 显示当前 shell 进程所有可用的命令别名；

b、定义别名，格式为： alias NAME='VALUE'

```
定义别名 NAME，其执行相当于执行命令 VALUE，VALUE 中可包含命令、选项以及参数；仅当前会话有效，不建议使用；
```

c、通过修改配置文件定义命令别名：

```
当前用户：~/.bashrc
全局用户：/etc/bashrc
```

- Bash 进程重新读取配置文件：

  ```
  source /path/to/config_file

  . /path/to/config_file
  ```

- 撤销别名： unalias

  ```
  unalias [-a] name [name...]
  ```

- Note:

  对于定义了别名的命令，要使用原命令，可通过、COMMAND 的方式使用；

- Example:

```
[root@sslinux sslinux]# alias
alias cp='cp -i'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
[root@sslinux sslinux]# grep alias /root/.bashrc
### User specific aliases and functions
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
```

通配符 glob

Bash 中用于文件名"通配"

- 通配符： *,?,[]

  1) * 任意长度的任意字符；

  ```
  a * b
  ```
  ```
  [root@sslinux sslinux]# ls -ld /etc/au*
  drwxr-x---. 3 root root 41 Sep 3 22:05 /etc/audisp
  drwxr-x---. 3 root root 79 Sep 3 22:09 /etc/audit
  ```

2) ? 任意单个字符；

```
            a?b
```

```
[root@sslinux sslinux]# ls -ld /etc/*d?t
drwxr-x---. 3 root root 79 Sep 3 22:09 /etc/audit
```

3) [] 匹配指定范围内的任意单个字符；

```
    [0-9]    [a-z]    不区分大小写；
    [admin]     可以是区间形式的，也可以是离散形式的；
```

```
[root@sslinux sslinux]# ls -ld /etc/[ab]*
drwxr-xr-x. 2 root root 4096 Sep 3 22:05 /etc/alternatives
drwxr-xr-x. 2 root root 33 Sep 3 22:04 /etc/avahi
drwxr-xr-x. 2 root root 33 Sep 3 22:04 /etc/bash_completion.d
-rw-r--r--. 1 root root 2835 Oct 29 2014 /etc/bashrc
drwxr-xr-x. 2 root root 6 Mar 6 2015 /etc/binfmt.d
```

4) 匹配指定范围以外的任意单个字符；

```
        [^0-9] : 单个非数字的任意字符；
```

- 专用字符结合：（表示一类字符中的单个）

[:digit:] 任意单个数字，相当于 [0-9];

[:lower:] 任意单个小写字母；

[:upper:] 任意单个大写字母；

[:alpha:] 任意单个大小写字母；

[:alnum:] 任意单个数字或字母；

[:space:] 任意空白字符；

[:punct:] 任意单个特殊字符；

- Note：

在使用 [] 应用专用字符集合时，外层也需要嵌套 []。

Example：

```
# ls -d /etc/*[[:digit:]]*[[:lower:]]
```

bash 快捷键

编辑命令

- Ctrl + a ：移到命令行首
- Ctrl + e ：移到命令行尾

- Ctrl + f ：按字符前移（右向）
- Ctrl + b ：按字符后移（左向）
- Alt + f ：按单词前移（右向）
- Alt + b ：按单词后移（左向）
- Ctrl + xx：在命令行首和光标之间移动
- Ctrl + u ：从光标处删除至命令行首
- Ctrl + k ：从光标处删除至命令行尾
- Ctrl + w ：从光标处删除至字首
- Alt + d ：从光标处删除至字尾
- Ctrl + d ：删除光标处的字符
- Ctrl + h ：删除光标前的字符
- Ctrl + y ：粘贴至光标后
- Alt + c ：从光标处更改为首字母大写的单词
- Alt + u ：从光标处更改为全部大写的单词
- Alt + l ：从光标处更改为全部小写的单词
- Ctrl + t ：交换光标处和之前的字符
- Alt + t ：交换光标处和之前的单词
- Alt + Backspace：与 Ctrl + w 相同类似，分隔符有些差别

重新执行命令

- Ctrl + r：逆向搜索命令历史
- Ctrl + g：从历史搜索模式退出
- Ctrl + p：历史中的上一条命令
- Ctrl + n：历史中的下一条命令
- Alt + .：使用上一条命令的最后一个参数

控制命令

- Ctrl + l：清屏
- Ctrl + o：执行当前命令，并选择上一条命令
- Ctrl + s：阻止屏幕输出
- Ctrl + q：允许屏幕输出
- Ctrl + c：终止命令
- Ctrl + z：挂起命令

Bang (!) 命令

- !!：执行上一条命令
- !blah：执行最近的以 blah 开头的命令，如 !ls

- !blah:p：仅打印输出，而不执行
- !$：上一条命令的最后一个参数，与 Alt + . 相同
- !$:p：打印输出 !$ 的内容
- !*：上一条命令的所有参数
- !:*p*：打印输出 *!* 的内容
- ^blah：删除上一条命令中的 blah
- ^blah^foo：将上一条命令中的 blah 替换为 foo
- ^blah^foo^：将上一条命令中所有的 blah 都替换为 foo

友情提示

```
以上介绍的大多数 Bash 快捷键仅当在 emacs 编辑模式时有效，

若你将 Bash 配置为 vi 编辑模式，那将遵循 vi 的按键绑定。

Bash 默认为 emacs 编辑模式。

如果你的 Bash 不在 emacs 编辑模式，可通过 set-o emacs 设置。

^S、^Q、^C、^Z 是由终端设备处理的，可用 stty 命令设置。
```

bash 的 io 重定向及管道

打开的文件都有一个 fd：file descriptor（文件描述符）

```
标准输入：keyboard，0

标准输出：monitor，1

标准错误输出：monitor，2
```

I/O 重定向

- 输出重定向：

  COMMAND > NEW_POS 覆盖重定向，目标文件中的原有内容会被清除；

  COMMAND >> NEW_POS 追加重定向，新内容会被追加到目标文件尾部；

- Note：

  为了在输出重定向时防止覆盖原有文件，建议使用以下设置：

  set –C ： 禁止将内容覆盖输出 (>) 至已有文件中，追加输出不受影响；

此时，若确定要将重定向的内容覆盖原有文件，可使用 >| 强制覆盖；

- Example:

```
[root@localhost test1]# echo "It's dangerous" > ./result.txt #输出到文件；
[root@localhost test1]# cat result.txt
It's dangerous
[root@localhost test1]# set –C #禁止将内容覆盖输出到已有文件；
[root@localhost test1]# echo "It's very dangerous" > ./result.txt
-bash: ./result.txt: cannot overwrite existing file #提示不能覆盖已存在文件；
[root@localhost test1]# echo "It's very dangerous" >| ./result.txt #强制覆盖
[root@localhost test1]#
[root@localhost test1]# set +C #取消禁止覆盖输出到已有文件；
[root@localhost test1]# echo "It's very dangerous" > ./result.txt
[root@localhost test1]#
```

- 错误输出：

  2>：覆盖重定向错误输出数据流；

  2>> ：追加重定向错误输出数据流；

```
[root@localhost test1]# lss -l /etc/ 2> ./error.txt
[root@localhost test1]# cat error.txt
-bash: lss: command not found
[root@localhost test1]# cat /etc/passwd.error 2>> ./error.txt
[root@localhost test1]# cat error.txt
-bash: lss: command not found
cat: /etc/passwd.error: No such file or directory
```

将标准输出和标准错误输出各自重定向至不同位置：

```
COMMAND > /path/to/file.out 2> /path/to/error.out
```

- Example:

```
# cat /etc/passwd > ./file.out 2> ./error.out
```

- 合并输出：

  合并标准输出和错误输出为同一个数据流进行重定向；(PS:重定向命令是倒序操作的，如 > file 2>&1 是先执行 2>&1 然后执行 > file)

```
&> 合并覆盖重定向；

&>> 合并追加重定向；
```

格式为：

```
COMMAND > /path/to/file.out 2> &1

COMMAND >> /path/to/file.out 2>> &1
```

Example:

```
[root@localhost test1]# ls -l /etc/ > ./file.out 2>&1
[root@localhost test1]# ls -l /etc/ &> file.out
[root@localhost test1]# ls -l /etcc/ &> file.out
[root@localhost test1]# cat file.out
ls: cannot access /etcc/: No such file or directory
```

- 输入重定向：**<**

```
HERE Documentation：<<

# cat << EOF

# cat > /path/to/somefile << EOF
```

Example: 输入重定向，输入完成后显示内容到标准输出上；

```
[root@localhost test1]# cat << EOF
> my name is kalaguiyin.
> I'm a tibetan.
> I come from Sichuan Provence.
> EOF
my name is kalaguiyin.
I'm a tibetan.
I come from Sichuan Provence.
```

Example：从标准输入读取输入并重定向到文件。

```
[root@localhost test1]# cat > hello.txt << EOF
> this is a test file.
> 中华人民共和国。
> EOF
[root@localhost test1]# cat hello.txt
this is a test file.
中华人民共和国。
```

- 管道：

COMMAND1 | COMMAND2 | COMMAND3 | …..

作用：前一个命令的执行结果将作为后一个命令执行的参数；

Note:

> 最后一个命令会在当前 shell 进程的子 shell 进程中执行；

```
[root@sslinux]# cat /etc/passwd | sort -t: -k3 -n | cut -d: -f1
root
bin
daemon
adm
lp
polkitd
sslinux
```

Linux 常用命令

系统

系统信息

命令	说明
# arch	显示机器的处理器架构
# cal 2016	显示 2016 年的日历表
# cat /proc/cpuinfo	查看 CPU 信息
# cat /proc/interrupts	显示中断
# cat /proc/meminfo	校验内存使用
# cat /proc/swaps	显示哪些 swap 被使用
# cat /proc/version	显示内核版本
# cat /proc/net/dev	显示网络适配器及统计
# cat /proc/mounts	显示已加载的文件系统
# clock -w	将时间修改保存到 BIOS
# date	显示系统日期
# date 072308302016.00	设置日期和时间 - 月日时分年、. 秒
# dmidecode -q	显示硬件系统部件 - (SMBIOS / DMI)
# hdparm -i /dev/hda	罗列一个磁盘的架构特性
# hdparm -tT /dev/sda	在磁盘上执行测试性读取操作
# lspci -tv	罗列 PCI 设备
# lsusb -tv	显示 USB 设备
# uname -m	显示机器的处理器架构
# uname -r	显示正在使用的内核版本

关机

命令	说明
# init 0	关闭系统
# logout	注销
# reboot	重启
# shutdown -h now	关闭系统
# shutdown -h 16:30 &	按预定时间关闭系统
# shutdown -c	取消按预定时间关闭系统
# shutdown -r now	重启

监视和调试

命令	说明
# free -m	以兆为单位罗列 RAM 状态
# kill -9 process_id	强行关闭进程并结束它
# kill -1 process_id	强制一个进程重载其配置
# last reboot	显示重启历史
# lsmod	罗列装载的内核模块
# lsof -p process_id	罗列一个由进程打开的文件列表
# lsof /home/user1	罗列所给系统路径中所打开的文件的列表
# ps -eafw	罗列 linux 任务
# ps -e -o pid,args --forest	以分级的方式罗列 linux 任务
# pstree	以树状图显示程序
# smartctl -A /dev/hda	通过启用 SMART 监控硬盘设备的可靠性
# smartctl -i /dev/hda	检查一个硬盘设备的 SMART 是否启用
# strace -c ls >/dev/null	罗列系统 calls made 并用一个进程接收
# strace -f -e open ls >/dev/null	罗列库调用
# tail /var/log/dmesg	显示内核引导过程中的内部事件
# tail /var/log/messages	显示系统事件
# top	罗列使用 CPU 资源最多的 linux 任务
# watch -n1 'cat /proc/interrupts'	罗列实时中断

公钥私钥

命令	说明
# ssh-keygen -t rsa -C "邮箱地址"	产生公钥私钥对
# ssh-copy-id -i ~/.ssh/id_rsa.pub root@192.168.0.2	将本地机器的公钥复制到远程机器的 root 用户的 authorized_keys 文件中
# ssh-keygen -p -f ~/.ssh/id_rsa	添加或修改 SSH-key 的私钥密码
# ssh-keygen -y -f ~/.ssh/id_rsa > id_rsa.pub	从私钥中生成公钥

其他

命令	说明
# alias hh='history'	为命令 history\（历史、) 设置一个别名
# gpg -c file1	用 GNU Privacy Guard 加密一个文件
# gpg file1.gpg	用 GNU Privacy Guard 解密一个文件
# ldd /usr/bin/ssh	显示 ssh 程序所依赖的共享库
# man ping	罗列在线手册页（例如 ping 命令）
# mkbootdisk --device /dev/fd0 `uname -r`	创建一个引导软盘
# wget -r www.example.com	下载一个完整的 web 站点
# wget -c www.example.com/file.iso	以支持断点续传的方式下载一个文件
# echo 'wget -c www.example.com/files.iso' | at 09:00	在任何给定的时间开始一次下载
# whatis ...keyword	罗列该程序功能的说明
# who -a	显示谁正登录在线，并打印出：系统最后引导的时间，关机进程，系统登录进程以及由 init 启动的进程，当前运行级和最后一次系统时钟的变化

资源

磁盘空间

命令	说明
# df -h	显示已经挂载的分区列表
# du -sh dir1	估算目录 'dir1' 已经使用的磁盘空间
# du -sk * | sort -rn	以容量大小为依据依次显示文件和目录的大小
# ls -lSr | more	以尺寸大小排列文件和目录
# rpm -q -a --qf '%10{SIZE}t% {NAME}n' | sort -k1,1n	以大小为依据依次显示已安装的 rpm 包所使用的空间 (centos, redhat, fedora 类系统、)

文件及文本处理

文件和目录

命令	说明
# cd /home	进入 '/home' 目录
# cd ..	返回上一级目录
# cd ../..	返回上两级目录
# cd	进入个人的主目录
# cd ~user1	进入个人的主目录
# cd -	返回上次所在的目录
# cp file1 file2	复制一个文件
# cp dir/* .	复制一个目录下的所有文件到当前工作目录
# cp -a /tmp/dir1 .	复制一个目录到当前工作目录
# cp -a dir1 dir2	复制一个目录
# cp file file1	将 file 复制为 file1
# iconv -l	列出已知的编码
# iconv -f fromEncoding -t toEncoding inputFile > outputFile	改变字符的编码
# find . -maxdepth 1 -name *.jpg -print -exec convert	batch resize files in the current directory and send them to a thumbnails directory (requires convert from Imagemagick)
# ln -s file1 lnk1	创建一个指向文件或目录的软链接
# ln file1 lnk1	创建一个指向文件或目录的物理链接
# ls	查看目录中的文件
# ls -F	查看目录中的文件
# ls -l	显示文件和目录的详细资料
# ls -a	显示隐藏文件
# ls *[0-9]*	显示包含数字的文件名和目录名
# lstree	显示文件和目录由根目录开始的树形结构
# mkdir dir1	创建一个叫做 'dir1' 的目录
# mkdir dir1 dir2	同时创建两个目录
# mkdir -p /tmp/dir1/dir2	创建一个目录树
# mv dir1 new_dir	重命名 / 移动 一个目录
# pwd	显示工作路径
# rm -f file1	删除一个叫做 'file1' 的文件

# rm -rf dir1	删除一个叫做 'dir1' 的目录并同时删除其内容
# rm -rf dir1 dir2	同时删除两个目录及它们的内容
# rmdir dir1	删除一个叫做 'dir1' 的目录
# touch -t 1607230000 file1	修改一个文件或目录的时间戳 - (YYMMDDhhmm)
# tree	显示文件和目录由根目录开始的树形结构

文件搜索

命令	说明
# find / -name file1	从 '/' 开始进入根文件系统搜索文件和目录
# find / -user user1	搜索属于用户 'user1' 的文件和目录
# find /home/user1 -name *.bin	在目录 '/ home/user1' 中搜索带有'.bin' 结尾的文件
# find /usr/bin -type f -atime +100	搜索在过去 100 天内未被使用过的执行文件
# find /usr/bin -type f -mtime -10	搜索在 10 天内被创建或者修改过的文件
# find / -name *.rpm -exec chmod 755 '{}' \;	搜索以 '.rpm' 结尾的文件并定义其权限
# find / -xdev -name *.rpm	搜索以 '.rpm' 结尾的文件，忽略光驱、捷盘等可移动设备
# locate *.ps	寻找以 '.ps' 结尾的文件 - 先运行 'updatedb' 命令
# whereis halt	显示一个二进制文件、源码或 man 的位置
# which halt	显示一个二进制文件或可执行文件的完整路径

文件的权限

命令	说明
# chgrp group1 file1	改变文件的群组
# chmod ugo+rwx directory1	设置目录的所有人、(u)、群组、(g) 以及其他人、(o) 以读、(r)、写、(w) 和执行、(x) 的权限
# chmod go-rwx directory1	删除群组、(g) 与其他人、(o) 对目录的读写执行权限
# chmod u+s /bin/file1	设置一个二进制文件的 SUID 位 - 运行该文件的用户也被赋予和所有者同样的权限
# chmod u-s /bin/file1	禁用一个二进制文件的 SUID 位
# chmod g+s /home/public	设置一个目录的 SGID 位 - 类似 SUID，不过这是针对目录的
# chmod g-s /home/public	禁用一个目录的 SGID 位
# chmod o+t /home/public	设置一个文件的 STIKY 位 - 只允许合法所有人删除文件
# chmod o-t /home/public	禁用一个目录的 STIKY 位
# chown user1 file1	改变一个文件的所有人属性
# chown -R user1 directory1	改变一个目录的所有人属性并同时改变改目录下所有文件的属性
# chown user1:group1 file1	改变一个文件的所有人和群组属性
# find / -perm -u+s	罗列一个系统中所有使用了 SUID 控制的文件
# ls -lh	显示权限
# ls /tmp | pr -T5 -W$COLUMNS	将终端划分成 5 栏显示

文件的特殊属性

命令	说明
# chattr +a file1	只允许以追加方式读写文件
# chattr +c file1	允许这个文件能被内核自动压缩 / 解压
# chattr +d file1	在进行文件系统备份时，dump 程序将忽略这个文件
# chattr +i file1	设置成不可变的文件，不能被删除、修改、重命名或者链接
# chattr +s file1	允许一个文件被安全地删除
# chattr +S file1	一旦应用程序对这个文件执行了写操作，使系统立刻把修改的结果写到磁盘
# chattr +u file1	若文件被删除，系统会允许你在以后恢复这个被删除的文件
# lsattr	显示特殊的属性

查看文件内容

命令	说明
# cat file1	从第一个字节开始正向查看文件的内容
# head -2 file1	查看一个文件的前两行
# less file1	类似于 'more' 命令，但是它允许在文件中和正向操作一样的反向操作
# more file1	查看一个长文件的内容
# tac file1	从最后一行开始反向查看一个文件的内容
# tail -2 file1	查看一个文件的最后两行
# tail -f /var/log/messages	实时查看被添加到一个文件中的内容

文本处理

命令	说明
# cat example.txt | awk 'NR%2==1'	删除 example.txt 文件中的所有偶数行
# echo a b c | awk '{print $1}'	查看一行第一栏
# echo a b c | awk '{print	查看一行的第一和第三栏

$1,$3}'	查看一行的第一和第三栏
# cat -n file1	标示文件的行数
# comm -1 file1 file2	比较两个文件的内容只删除 'file1' 所包含的内容
# comm -2 file1 file2	比较两个文件的内容只删除 'file2' 所包含的内容
# comm -3 file1 file2	比较两个文件的内容只删除两个文件共有的部分
# diff file1 file2	找出两个文件内容的不同处
# grep Aug /var/log/messages	在文件 '/var/log/messages'中查找关键词"Aug"
# grep ^Aug /var/log/messages	在文件 '/var/log/messages'中查找以"Aug"开始的词汇
# grep [0-9] /var/log/messages	选择 '/var/log/messages' 文件中所有包含数字的行
# grep Aug -R /var/log/*	在目录 '/var/log' 及随后的目录中搜索字符串"Aug"
# paste file1 file2	合并两个文件或两栏的内容
# paste -d '+' file1 file2	合并两个文件或两栏的内容，中间用"+"区分
# sdiff file1 file2	以对比的方式显示两个文件的不同
# sed 's/string1/string2/g' example.txt	将 example.txt 文件中的 "string1" 替换成 "string2"
# sed '/^$/d' example.txt	从 example.txt 文件中删除所有空白行
# sed '/ *|#/d; /^$/d' example.txt	去除文件 example.txt 中的注释与空行
# sed -e '1d' exampe.txt	从文件 example.txt 中排除第一行
# sed -n '/string1/p'	查看只包含词汇 "string1"的行
# sed -e 's/ *$//' example.txt	删除每一行最后的空白字符
# sed -e 's/string1//g' example.txt	从文档中只删除词汇 "string1" 并保留剩余全部
# sed -n '1,5p' example.txt	显示文件 1 至 5 行的内容
# sed -n '5p;5q' example.txt	显示 example.txt 文件的第 5 行内容
# sed -e 's/00*/0/g' example.txt	用单个零替换多个零
# sort file1 file2	排序两个文件的内容
# sort file1 file2 | uniq	取出两个文件的并集、（重复的行只保留一份、)
# sort file1 file2 | uniq -u	删除交集，留下其他的行
# sort file1 file2 | uniq -d	取出两个文件的交集、（只留下同时存在于两个文件中的文件、)
# echo 'word' | tr '[:lower:]' '[:upper:]'	合并上下单元格内容

字符设置和文件格式

命令	说明
# dos2unix filedos.txt fileunix.txt	将一个文本文件的格式从 MSDOS 转换成 UNIX
# recode ..HTML < page.txt > page.html	将一个文本文件转换成 html
# recode -l | more	显示所有允许的转换格式
# unix2dos fileunix.txt filedos.txt	将一个文本文件的格式从 UNIX 转换成 MSDOS

挂载

挂载一个文件系统

命令	说明
# fuser -km /mnt/hda2	当设备繁忙时强制卸载
# mount /dev/hda2 /mnt/hda2	挂载一个叫做 hda2 的盘 - 确保目录 '/mnt/hda2' 已经存在
# mount /dev/fd0 /mnt/floppy	挂载一个软盘
# mount /dev/cdrom /mnt/cdrom	挂载一个 cdrom 或 dvdrom
# mount /dev/hdc /mnt/cdrecorder	挂载一个 cdrw 或 dvdrom
# mount /dev/hdb /mnt/cdrecorder	挂载一个 cdrw 或 dvdrom
# mount -o loop file.iso /mnt/cdrom	挂载一个文件或 ISO 镜像文件
# mount -t vfat /dev/hda5 /mnt/hda5	挂载一个 Windows FAT32 文件系统
# mount /dev/sda1 /mnt/usbdisk	挂载一个 U 盘或闪存设备
# mount -t smbfs -o username=user,password=pass //WinClient/share /mnt/share	挂载一个 windows 网络共享
# umount /dev/hda2	卸载一个叫做 hda2 的盘 - 先从挂载点 '/mnt/hda2' 退出
# umount -n /mnt/hda2	运行卸载操作而不写入 /etc/mtab 文件 - 当文件为只读或当磁盘写满时非常有用

光盘

命令	说明
# cd-paranoia -B	从一个 CD 光盘转录音轨到 wav 文件中
# cd-paranoia --	从一个 CD 光盘转录音轨到 wav 文件中（参数、-3）
# cdrecord -v gracetime=2 dev=/dev/cdrom -eject blank=fast -force	清空一个可复写的光盘内容
# cdrecord -v dev=/dev/cdrom cd.iso	刻录一个 ISO 镜像文件
# gzip -dc cd_iso.gz | cdrecord dev=/dev/cdrom -	刻录一个压缩了的 ISO 镜像文件
# cdrecord --scanbus	扫描总线以识别 scsi 通道
# dd if=/dev/hdc | md5sum	校验一个设备的 md5sum 编码，例如一张 CD
# mkisofs /dev/cdrom > cd.iso	在磁盘上创建一个光盘的 iso 镜像文件
# mkisofs /dev/cdrom | gzip > cd_iso.gz	在磁盘上创建一个压缩了的光盘 iso 镜像文件
# mkisofs -J -allow-leading-dots -R -V	创建一个目录的 iso 镜像文件
# mount -o loop cd.iso /mnt/iso	挂载一个 ISO 镜像文件

用户管理

用户和群组

命令	说明
# chage -E 2016-12-31 user1	设置用户口令的失效期限
# groupadd [group]	创建一个新用户组
# groupdel [group]	删除一个用户组
# groupmod -n moon sun	重命名一个用户组
# grpck	检查 '/etc/passwd' 的文件格式和语法修正以及存在的群组
# newgrp - [group]	登陆进一个新的群组以改变新创建文件的预设群组
# passwd	修改口令
# passwd user1	修改一个用户的口令 \ (只允许 root 执行、)
# pwck	检查 '/etc/passwd' 的文件格式和语法修正以及存在的用户
# useradd -c "User Linux" -g admin -d /home/user1 -s /bin/bash user1	创建一个属于 "admin" 用户组的用户
# useradd user1	创建一个新用户
# userdel -r user1	删除一个用户 ('-r' 排除主目录、)
# usermod -c "User FTP" -g system -d /ftp/user1 -s /bin/nologin user1	修改用户属性

包管理

打包和压缩文件

命令	说明
# bunzip2 file1.bz2	解压一个叫做 'file1.bz2'的文件
# bzip2 file1	压缩一个叫做 'file1' 的文件
# gunzip file1.gz	解压一个叫做 'file1.gz'的文件
# gzip file1	压缩一个叫做 'file1'的文件
# gzip -9 file1	最大程度压缩
# rar a file1.rar test_file	创建一个叫做 'file1.rar' 的包
# rar a file1.rar file1 file2 dir1	同时压缩 'file1', 'file2' 以及目录 'dir1'
# rar x file1.rar	解压 rar 包
# tar -cvf archive.tar file1	创建一个非压缩的 tarball
# tar -cvf archive.tar file1 file2 dir1	创建一个包含了 'file1', 'file2' 以及 'dir1'的档案文件
# tar -tf archive.tar	显示一个包中的内容
# tar -xvf archive.tar	释放一个包
# tar -xvf archive.tar -C /tmp	将压缩包释放到 /tmp 目录下
# tar -cvfj archive.tar.bz2 dir1	创建一个 bzip2 格式的压缩包
# tar -xvfj archive.tar.bz2	解压一个 bzip2 格式的压缩包
# tar -cvfz archive.tar.gz dir1	创建一个 gzip 格式的压缩包
# tar -xvfz archive.tar.gz	解压一个 gzip 格式的压缩包
# unrar x file1.rar	解压 rar 包
# unzip file1.zip	解压一个 zip 格式压缩包
# zip file1.zip file1	创建一个 zip 格式的压缩包
# zip -r file1.zip file1 file2 dir1	将几个文件和目录同时压缩成一个 zip 格式的压缩包

RPM 包 (Fedora,RedHat and alike)

命令	说明
# rpm -ivh [package.rpm]	安装一个 rpm 包
# rpm -ivh --nodeeps [package.rpm]	安装一个 rpm 包而忽略依赖关系警告
# rpm -U [package.rpm]	更新一个 rpm 包但不改变其配置文件
# rpm -F [package.rpm]	更新一个确定已经安装的 rpm 包
# rpm -e [package]	删除一个 rpm 包

# rpm -qa	显示系统中所有已经安装的 rpm 包
# rpm -qa | grep httpd	显示所有名称中包含 "httpd" 字样的 rpm 包
# rpm -qi [package]	获取一个已安装包的特殊信息
# rpm -qg "System Environment/Daemons"	显示一个组件的 rpm 包
# rpm -ql [package]	显示一个已经安装的 rpm 包提供的文件列表
# rpm -qc [package]	显示一个已经安装的 rpm 包提供的配置文件列表
# rpm -q [package] --whatrequires	显示与一个 rpm 包存在依赖关系的列表
# rpm -q [package] --whatprovides	显示一个 rpm 包所占的体积
# rpm -q [package] --scripts	显示在安装 / 删除期间所执行的脚本 l
# rpm -q [package] --changelog	显示一个 rpm 包的修改历史
# rpm -qf /etc/httpd/conf/httpd.conf	确认所给的文件由哪个 rpm 包所提供
# rpm -qp [package.rpm] -l	显示由一个尚未安装的 rpm 包提供的文件列表
# rpm --import /media/cdrom/RPM-GPG-KEY	导入公钥数字证书
# rpm --checksig [package.rpm]	确认一个 rpm 包的完整性
# rpm -qa gpg-pubkey	确认已安装的所有 rpm 包的完整性
# rpm -V [package]	检查文件尺寸、 许可、类型、所有者、群组、MD5 检查以及最后修改时间
# rpm -Va	检查系统中所有已安装的 rpm 包、- 小心使用
# rpm -Vp [package.rpm]	确认一个 rpm 包还未安装
# rpm -ivh /usr/src/redhat/RPMS/`arch`/[package.rpm]	从一个 rpm 源码安装一个构建好的包
# rpm2cpio [package.rpm] | cpio --extract --make-directories *bin*	从一个 rpm 包运行可执行文件
# rpmbuild --rebuild [package.src.rpm]	从一个 rpm 源码构建一个 rpm 包

YUM 软件工具 (Fedora,RedHat and alike)

命令	说明
# yum -y install [package]	下载并安装一个 rpm 包
# yum localinstall [package.rpm]	将安装一个 rpm 包，使用你自己的软件仓库为你解决所有依赖关系
# yum -y update	更新当前系统中所有安装的 rpm 包
# yum update [package]	更新一个 rpm 包
# yum remove [package]	删除一个 rpm 包
# yum list	列出当前系统中安装的所有包
# yum repolist	显示可用的仓库
# yum search [package]	在 rpm 仓库中搜寻软件包
# yum clean [package]	清理 rpm 缓存删除下载的包
# yum clean headers	删除所有头文件
# yum clean all	删除所有缓存的包和头文件

备份

命令	说明
# find /var/log -name '*.log' | tar cv --files-from=- | bzip2 > log.tar.bz2	查找所有以 '.log' 结尾的文件并做成一个 bzip 包
# find /home/user1 -name '*.txt' | xargs cp -av --target-directory=/home/backup/ --parents	从一个目录查找并复制所有以 '.txt' 结尾的文件到另一个目录
# dd bs=1M if=/dev/hda | gzip | ssh user@ip_addr 'dd of=hda.gz'	通过 ssh 在远程主机上执行一次备份本地磁盘的操作
# dd if=/dev/sda of=/tmp/file1	备份磁盘内容到一个文件
# dd if=/dev/hda of=/dev/fd0 bs=512 count=1	做一个将 MBR (Master Boot Record) 内容复制到软盘的动作
# dd if=/dev/fd0 of=/dev/hda bs=512 count=1	从已经保存到软盘的备份中恢复 MBR 内容
# dump -0aj -f /tmp/home0.bak /home	制作一个 '/home' 目录的完整备份
# dump -1aj -f /tmp/home0.bak /home	制作一个 '/home' 目录的交互式备份
# restore -if /tmp/home0.bak	还原一个交互式备份
# rsync -rogpav --delete /home /tmp	同步两边的目录
# rsync -rogpav -e ssh --delete /home ip_address:/tmp	通过 SSH 通道 rsync
# rsync -az -e ssh --delete ip_addr:/home/public /home/local	通过 ssh 和压缩将一个远程目录同步到本地目录
# rsync -az -e ssh --delete /home/local ip_addr:/home/public	通过 ssh 和压缩将本地目录同步到远程目录
# tar -Puf backup.tar /home/user	执行一次对 '/home/user' 目录的交互式备份操作
# (cd /tmp/local/ && tar c .) | ssh -C user@ip_addr 'cd /home/share/ && tar x -p'	通过 ssh 在远程目录中复制一个目录内容
# (tar c /home) | ssh -C user@ip_addr 'cd /home/backup-home && tar x -p'	通过 ssh 在远程目录中复制一个本地目录
# tar cf - . | (cd /tmp/backup ; tar xf -)	本地将一个目录复制到另一个地方，保留原有权限及链接

磁盘和分区

文件系统分析

命令	说明
# badblocks -v /dev/hda1	检查磁盘 hda1 上的坏磁块
# dosfsck /dev/hda1	修复 / 检查 hda1 磁盘上 dos 文件系统的完整性
# e2fsck /dev/hda1	修复 / 检查 hda1 磁盘上 ext2 文件系统的完整性
# e2fsck -j /dev/hda1	修复 / 检查 hda1 磁盘上 ext3 文件系统的完整性
# fsck /dev/hda1	修复 / 检查 hda1 磁盘上 linux 文件系统的完整性
# fsck.ext2 /dev/hda1	修复 / 检查 hda1 磁盘上 ext2 文件系统的完整性
# fsck.ext3 /dev/hda1	修复 / 检查 hda1 磁盘上 ext3 文件系统的完整性
# fsck.vfat /dev/hda1	修复 / 检查 hda1 磁盘上 fat 文件系统的完整性
# fsck.msdos /dev/hda1	修复 / 检查 hda1 磁盘上 dos 文件系统的完整性

初始化一个文件系统

命令	说明
# fdformat -n /dev/fd0	格式化一个软盘
# mke2fs /dev/hda1	在 hda1 分区创建一个 linux ext2 的文件系统
# mke2fs -j /dev/hda1	在 hda1 分区创建一个 linux ext3\（日志型、) 的文件系统
# mkfs /dev/hda1	在 hda1 分区创建一个文件系统
# mkfs -t vfat 32 -F /dev/hda1	创建一个 FAT32 文件系统
# mkswap /dev/hda3	创建一个 swap 文件系统

SWAP 文件系统

命令	说明
# mkswap /dev/hda3	创建一个 swap 文件系统
# swapon /dev/hda3	启用一个新的 swap 文件系统
# swapon /dev/hda2 /dev/hdb3	启用两个 swap 分区

网络

网络 (LAN / WiFi)

命令	说明
# dhclient eth0	以 dhcp 模式启用 'eth0' 网络设备
# ethtool eth0	显示网卡 'eth0' 的流量统计
# host www.example.com	查找主机名以解析名称与 IP 地址及镜像
# hostname	显示主机名
# ifconfig eth0	显示一个以太网卡的配置
# ifconfig eth0 192.168.1.1 netmask 255.255.255.0	控制 IP 地址
# ifconfig eth0 promisc	设置 'eth0' 成混杂模式以嗅探数据包 (sniffing)
# ifdown eth0	禁用一个 'eth0' 网络设备
# ifup eth0	启用一个 'eth0' 网络设备
# ip link show	显示所有网络设备的连接状态
# iwconfig eth1	显示一个无线网卡的配置
# iwlist scan	显示无线网络
# mii-tool eth0	显示 'eth0'的连接状态
# netstat -tup	显示所有启用的网络连接和它们的 PID
# netstat -tupl	显示系统中所有监听的网络服务和它们的 PID
# netstat -rn	显示路由表，类似于“route -n”命令
# nslookup www.example.com	查找主机名以解析名称与 IP 地址及镜像
# route -n	显示路由表
# route add -net 0/0 gw IP_Gateway	控制预设网关
# route add -net 192.168.0.0 netmask 255.255.0.0 gw 192.168.1.1	控制通向网络 '192.168.0.0/16' 的静态路由
# route del 0/0 gw IP_gateway	删除静态路由
# echo "1" > /proc/sys/net/ipv4/ip_forward	激活 IP 转发
# tcpdump tcp port 80	显示所有 HTTP 回环
# whois www.example.com	在 Whois 数据库中查找

route 设置

基本使用

添加到主机的路由（就是一个 IP 一个 IP 添加）

```
route add -host 146.148.149.202 dev eno16777984
route add -host 146.148.149.202 gw 146.148.149.193
```

添加到网络的路由（批量）

```
route add -net 146.148.149.0 netmask 255.255.255.0 dev eno16777984
route add -net 146.148.149.0 netmask 255.255.255.0 gw 146.148.149.193
```

简洁写法

```
route add -net 146.148.150.0/24 dev eno16777984
route add -net 146.148.150.0/24 gw 146.148.150.193
```

添加默认网关

```
route add default gw 146.148.149.193
```

删除主机路由：

```
route del -host 146.148.149.202 dev eno16777984
```

删除网络路由：

```
route del -net 146.148.149.0 netmask 255.255.255.0
route del -net 146.148.150.0/24
```

删除默认路由

```
route del default gw 146.148.149.193
```

在 **linux** 下设置永久路由的方法

服务器启动时自动设置路由，第一想到的可能时 `rc.local`

按照 linux 启动的顺序，rc.local 里面的内容是在 linux 所有服务都启动完毕，最后才被执行的，也就是说，这里面的内容是在 NFS 之后才被执行的，那也就是说在 NFS 启动的时候，服务器上的静态路由是没有被添加的，所以 NFS 挂载不能成功。

/etc/sysconfig/static-routes

```
any net 192.168.3.0/24 gw 192.168.3.254
any net 10.250.228.128 netmask 255.255.255.192 gw 10.250.228.129
```

使用 static-routes 的方法是最好的。无论重启系统和 service network restart 都会生效

static-routes 文件又是什么呢，这个是 network 脚本 (/etc/init.d/network) 调用的，大致的程序如下

```
if [ -f /etc/sysconfig/static-routes  ]; then
    grep "^any" /etc/sysconfig/static-routes | while read ignore args ; do
        /sbin/route add -$args
    done
fi
```

从这段脚本可以看到，这个就是添加静态路由的方法，static-routes 的写法是

any net 192.168.0.0/16 gw 网关 ip

Microsoft windows 网络 (samba)

命令	说明
# mount -t smbfs -o username=user,password=pass //WinClient/share /mnt/share	挂载一个 windows 网络共享
# nbtscan ip_addr	netbios 名解析
# nmblookup -A ip_addr	netbios 名解析
# smbclient -L ip_addr/hostname	显示一台 windows 主机的远程共享
# smbget -Rr smb://ip_addr/share	像 wget 一样能够通过 smb 从一台 windows 主机上下载文件

IPTABLES (firewall)

命令	说明
# iptables -t filter -L	显示过滤表的所有链路
# iptables -t nat -L	显示 nat 表的所有链路
# iptables -t filter -F	以过滤表为依据清理所有规则
# iptables -t nat -F	以 nat 表为依据清理所有规则
# iptables -t filter -X	删除所有由用户创建的链路
# iptables -t filter -A INPUT -p tcp --dport telnet -j ACCEPT	允许 telnet 接入
# iptables -t filter -A OUTPUT -p tcp --dport http -j DROP	阻止 HTTP 连出
# iptables -t filter -A FORWARD -p tcp --dport pop3 -j ACCEPT	允许转发链路上的 POP3 连接
# iptables -t filter -A INPUT -j LOG --log-prefix	记录所有链路中被查封的包
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE	设置一个 PAT \（端口地址转换、) 在 eth0 掩盖发出包
# iptables -t nat -A PREROUTING -d 192.168.0.1 -p tcp -m tcp --dport 22 -j DNAT --to-destination 10.0.0.2:22	将发往一个主机地址的包转向到其他主机

Linux 简单管理

ssh

ssh 简介及基本操作

简介

SSH，全名 secure shell，其目的是用来从终端与远程机器交互，SSH 设计之处初，遵循了如下原则：

- 机器之间通讯的内容必须经过加密。
- 加密过程中，通过 public key 加密，private 解密。

密钥

SSH 通讯的双方各自持有一个公钥私钥对，公钥对对方是可见的，私钥仅持有者可见，你可以通过"ssh-keygen"生成自己的公私钥，默认情况下，公私钥的存放路径如下：

- 公钥：$HOME/.ssh/id_rsa.pub
- 私钥：$HOME/.ssh/id_rsa

基于口令的安全验证通讯原理

建立通信通道的步骤如下：

```
  1. 远程主机将公钥发给用户 ---- 远程主机收到用户的登录请求，把自己的公钥发给客户端，客户端检查这个 public key 是否在自己的 $HOME/.ssh/known_hosts 中，如果没有，客户端会提示是否要把这个 public key 加入到 known_hosts 中。
  2. 用户使用公钥加密密码 ------ 用户使用这个公钥，将登录密码加密后，发送回来
  3. 远程主机使用私钥加密 ------ 远程主机用自己的私钥，解密登录密码，如果密码正确，就同意用户登录。
  4. 客户端把 PUBLIC KEY(client), 发送给服务器。
  5. 至此，到此为止双方彼此拥有了对方的公钥，开始双向加密解密。
```

PS：当网络中有另一台冒牌服务器冒充远程主机时，客户端的连接请求被服务器 B 拦截，服务器 B 将自己的公钥发送给客户端，客户端就会将密码加密后发送给冒牌服务器，冒牌服务器就可以拿自己的私钥获取到密码，然后为所欲为。因此当第一次链接远程主机时，在上述步骤中，会提示您当前远程主机的"公钥指纹"，以确认远程主机是否是正版的远程主机，如果选择继续后就可以输入密码进行登录了，当远程的主机接受以后，该台服务器的公钥就会保存到 ~/.ssh/known_hosts 文件中。

StrictHostKeyChecking 和 UserKnownHostsFile

> - 如何让连接新主机时，不进行公钥确认？
>   - SSH 客户端的 StrictHostKeyChecking 配置指令，可以实现当第一次连接服务器时，自动接受新的公钥。
>   - [例子:] ssh -o StrictHostKeyChecking=no 192.168.0.110
> - 如何防止远程主机公钥改变导致 SSH 连接失败
>   - 将本地的 known_hosts 指向不同的文件，就不会造成公钥冲突导致的中断了,提示信息由公钥改变中断警告，变成了首次连接的提示。结合自动接收新的公钥
>   - [例子:] ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null 192.168.0.110

基于密匙的安全验证通讯原理

这种方式你需要在当前用户家目录下为自己创建一对密匙，并把公匙放在需要登录的服务器上。当你要连接到服务器上时，客户端就会向服务器请求使用密匙进行安全验证。服务器收到请求之后，会在该服务器上你所请求登录的用户的家目录下寻找你的公匙，然后与你发送过来的公匙进行比较。如果两个密匙一致，服务器就用该公匙加密"质询"并把它发送给客户端。客户端收到"质询"之后用自己的私匙解密再把它发送给服务器。与第一种级别相比，第二种级别不需要在网络上传送口令。

PS：简单来说，就是将客户端的公钥放到服务器上，那么客户端就可以免密码登录服务器了，那么客户端的公钥应该放到服务器上哪个地方呢？默认为你要登录的用户的家目录下的 .ssh 目录下的 authorized_keys 文件中（即：~/.ssh/authorized_keys）。 我们的目标是：用户已经在主机 A 上登录为 a 用户，现在需要以不输入密码的方式以用户 b 登录到主机 B 上。

可以把密钥理解成一把钥匙，公钥理解成这把钥匙对应的锁头， 把锁头（公钥）放到想要控制的 server 上，锁住 server, 只有拥有钥匙（密钥）的人， 才能打开锁头，进入 server 并控制 而对于拥有这把钥匙的人，必需得知道钥匙本身的密码，才能使用这把钥匙（除非这把钥匙没设置密码）, 这样就可以防止钥匙被了配了（私钥被人复制）

当然，这种例子只是方便理解罢了， 拥有 root 密码的人当然是不会被锁住的，而且不一定只有一把锁（公钥）, 但如果任何一把锁，被人用其对应的钥匙（私钥）打开了，server 就可以被那个人控制了 所以说，只要你曾经知道 server 的 root 密码，并将有 root 身份的公钥放到上面， 就可以用这个公钥对应的私钥"打开" server, 再以 root 的身分登录，即使现在 root 密码已经更改！

步骤如下：

1. 以用户 a 登录到主机 A 上，生成一对公私钥。
2. 把主机 A 的公钥复制到主机 B 的 authorized_keys 中，可能需要输入 b@B 的密码。

   ssh-copy-id -i ~/.ssh/id_dsa.pub b@B

3. 在 a@A 上以免密码方式登录到 b@B

   ```
   ssh b@B
   ```

tips:

假如 B 机器修改端口后，将主机 A 上的公钥复制到 B 机的操作方法是（下面方法中双引号是必须的）：

ssh-copy-id "-p 端口号 b@B"

SSH 端口转发

SSH 还同时提供了一个非常有用的功能，这就是端口转发。它能够将其他 TCP 端口的网络数据通过 SSH 链接来转发，并且自动提供了相应的加密及解密服务。这一过程有时也被叫做“隧道”(tunneling)，这是因为 SSH 为其他 TCP 链接提供了一个安全的通道来进行传输而得名。

SSH 端口转发自然需要 SSH 连接，而 SSH 连接是有方向的，从 SSH Client 到 SSH Server 。 而我们所要访问的应用也是有方向的，应用连接的方向也是从应用的 Client 端连接到应用的 Server 端。 比如需要我们要访问 Internet 上的 Web 站点时，Http 应用的方向就是从我们

自己这台主机 (Client) 到远处的 Web Server。

> - SSH 连接和应用的连接这两个连接的方向一致，那我们就说它是本地转发。
> - SSH 连接和应用的连接这两个连接的方向不同，那我们就说它是远程转发。

SSH 正向连接

正向连接就是 client 连上 server，然后把 server 能访问的机器地址和端口（当然也包括 server 自己）镜像到 client 的端口上。

```
何时使用本地 Tunnel？

> * 比如说你在本地访问不了某个网络服务（如 www.google.com)，而有一台机器（如：xx.xx.xx.xx) 可以，那么你就可以通过这台机器的 ssh 服务来转发
```

使用方法

```
ssh -L <local port>:<remote host>:<remote port> <SSH hostname>
ssh -L [客户端 IP 或省略]:[客户端端口]:[服务器能访问的 IP]:[服务器能访问的 IP 的端口] [登陆服务器的用户名 @服务器 IP] -p [服务器 ssh 服务端口（默认 22）]
```

ssh -L 1433:target_server:1433 user@ssh_host

windows 下使用本地转发 ***xshell***

```
(1)ssh 远程连接到 Linux
(2) 打开代理设置面板，点击：view -> Tunneling Pane, 在弹出的窗口选择 Forwarding Rules
(3) 在空白处右键：add。
在弹出的 Forwarding Rule，
Type 选择"Local(Outgoing)";
Source Host 使用默认的 localhost; Listen Port 添上端口 8888;
Destination Host 使用默认的 localhost；Destination Port 添上 80;

Destination Host 设置为 localhost 为要访问的机器，可以设置为登陆后的机器可以访问到的 IP
```

SSH 反向连接

反向连接就是 client 连上 server，然后把 client 能访问的机器地址和端口（也包括 client 自己）镜像到 server 的端口上。

```
何时使用反向连接？

比如当你下班回家后就访问不了公司内网的机器了，遇到这种情况可以事先在公司内网的机器上执行远程 Tunnel，连上一台公司外网的机器，等你下班回家后就可以通过公司外网的机器去访问公司内网的机器了。
```

使用方法

```
ssh -R <remote port>:<local host>:<local port> <SSH hostname>
ssh -R [服务器 IP 或省略]:[服务器端口]:[客户端能访问的 IP]:[客户端能访问的 IP 的端口] [登陆服务器的用户名 @服务器 IP] -p [服务器 ssh 服务端口（默认 22）]
```

外网机器 **A** 要控制 内网机器 **B**

A 主机：外网，ip：122.122.122.122，sshd 端口：2222（默认是 22)

B 主机：内网，sshd 端口：2222（默认是 22)

无论是外网主机 A，还是内网主机 B 都需要跑 ssh daemon

首先在内网机器 ***B*** 上执行

```
    ssh -NfR 1234:localhost:2222 user1@122.122.122.122 -p 2222
```

这句话的意思是将 A 主机的 1234 端口和 B 主机的 2222 端口绑定，相当于远程端口映射 (Remote Port Forwarding)。

外网机器 ***A*** 会 ***listen*** 本地 ***1234*** 端口

```
---- 外网机器 A sshd 会 listen 本地 1234 端口
    #netstat -tanp | grep sshd
    #Proto Recv-Q Send-Q   Local Address   Foreign Address   State       PID/Program n
ame
    #tcp     0      0      127.0.0.1:1234    0.0.0.0:*       LISTEN      4234/sshd

---- 在外网机器 A 登录内网机器 B（非 root 用户的话，直接 user@localhost 即可）
    #ssh user@localhost -p1234
```

SSH 反向连接自动重连

上面的反向连接（Reverse Connection）不稳定，可能随时断开，需要内网主机 B 再次向外网 A 发起连接，这时需要个"朋友"帮你在内网 B 主机执行这条命令。可以使用 Autossh 或者 while 进行循环。

(1) 在 B 机器上将 B 机器公钥放到外网机器 A 上（实现 B 机器自动登录到 A 机器）

(2) 用 Autossh 或者 while 循环 保持 ssh 反向隧道一直连接，CentOS 需要使用 epel 源下载

在 CentOS6 和 CentOS7 都可以执行下面的命令安装 epel 仓库

while

编写脚本写入如下内容

```
#!/bin/bash
# 远程机器的 IP 和端口
remote_ip=122.122.122.122
remote_port=2222

while [[ 1==1  ]]
do
    ssh  -o ServerAliveInterval=15 -o ServerAliveCountMax=3 -N -R:1234:localhost:22 -p
 ${remote_port} root@${remote_ip}
    sleep 3
done
```

执行脚本后，即可以通过登陆 122.122.122.122 机器访问本地 1234 端口进行访问此机器

注;可以将此脚本放在后台中运行，并加到系统自启动程序中

autossh

```
#yum -y install epel-release
```

安装号 autossh 后使用如下方法进行反向连接

```
#autossh -M 5678 -NfR 1234:localhost:2222 user1@122.122.122.122 -p2222
```

比之前的命令添加的一个 -M 5678 参数，负责通过 5678 端口监视连接状态，连接有问题时就会自动重连

windows 下 xshell 使用

- 私钥，在 Xshell 里也叫用户密钥
- 公钥，在 Xshell 里也叫主机密钥

  利用 xshell 密钥管理服务器远程登录，

  (1)Xshell 自带有用户密钥生成向导：点击菜单栏的工具 ->新建用户密钥生成向导 (2) 添加公钥 (Pubic Key) 到远程 Linux 服务器

用户管理

Linux 踢出其他正在 SSH 登陆用户

查看系统在线用户

```
[root@Linux ~]#w
 20:40:18 up 1 day, 23 min,  4 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
root     tty1     -                Fri09   10:10m  0.34s  0.34s -bash
root     pts/2    192.168.31.124   10:30    4:39m  0.99s  0.99s -bash
root     pts/3    192.168.31.124   19:55    0.00s  0.07s  0.00s w
root     pts/4    192.168.31.124   19:55    4:52   0.16s  0.16s -bash
```

查看当前自己占用终端

```
[root@Linux ~]# who am i
root     pts/4        2016-10-30 19:55 (192.168.31.124)
```

用 ***pkill*** 命令剔除对方

```
[root@Linux ~]# pkill -kill -t pts/2
[root@Linux ~]# w
 20:44:15 up 1 day, 27 min,  3 users,  load average: 0.01, 0.03, 0.01
 USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
 root     tty1     -                Fri09   10:14m  0.34s  0.34s -bash
 root     pts/3    192.168.31.124   19:55   51.00s  1.43s  1.35s vim base.md
 root     pts/4    192.168.31.124   19:55    0.00s  0.21s  0.00s w
```

如果最后查看还是没有干掉，建议加上 -9 强制杀死。 [root@Linux ~]# pkill -9 -t pts/2

使用脚本创建有 **sudo** 权限的用户

```
cat >./create_user.sh <<-'EOF'
#!/bin/bash
arg="ceshi"
if id ${arg}
then
    echo "the username is exsit!"
else
    useradd $arg
    echo "ceshi_password" | passwd --stdin $arg
    echo "${arg} ALL=(ALL) NOPASSWD:ALL" > /etc/sudoers.d/${arg}
fi
EOF
```

以上脚本会创建用户 `ceshi` 同时用户的密码为 `ceshi_password` ，并且此用户有 sudo 权限

无交互式修改用户密码

```
echo "123456" | passwd --stdin root
```

网卡 bond

Linux 多网卡绑定

网卡绑定 mode 共有七种 (0~6) bond0、bond1、bond2、bond3、bond4、bond5、bond6

常用的有三种

> - mode=0：平衡负载模式，有自动备援，但需要”Switch”支援及设定。
> - mode=1：自动备援模式，其中一条线若断线，其他线路将会自动备援。
> - mode=6：平衡负载模式，有自动备援，不必”Switch”支援及设定。

需要说明的是如果想做成 mode 0 的负载均衡，仅仅设置这里 options bond0 miimon=100 mode=0 是不够的，与网卡相连的交换机必须做特殊配置（这两个端口应该采取聚合方式），因为做 bonding 的这两块网卡是使用同一个 MAC 地址。从原理分析一下（bond 运行在 mode 0 下）：

mode 0 下 bond 所绑定的网卡的 IP 都被修改成相同的 mac 地址，如果这些网卡都被接在同一个交换机，那么交换机的 arp 表里这个 mac 地址对应的端口就有多 个，那么交换机接受到发往这个 mac 地址的包应该往哪个端口转发呢？正常情况下 mac 地址是全球唯一的，一个 mac 地址对应多个端口肯定使交换机迷惑了。所以 mode0 下的 bond 如果连接到交换机，交换机这几个端口应该采取聚合方式（cisco 称为 ethernetchannel，foundry 称为 portgroup），因为交换机做了聚合后，聚合下的几个端口也被捆绑成一个 mac 地址。我们的解 决办法是，两个网卡接入不同的交换机即可。

mode6 模式下无需配置交换机，因为做 bonding 的这两块网卡是使用不同的 MAC 地址。

其他设置

时区及时间

时区就是时间区域，主要是为了克服时间上的混乱，统一各地时间。地球上共有 24 个时区，东西各 12 个时区（东 12 与西 12 合二为一）。

UTC 和 GMT

时区通常写成 `+0800`，有时也写成 `GMT +0800`，其实这两个是相同的概念。

GMT 是格林尼治标准时间（Greenwich Mean Time）。

UTC 是协调世界时间（Universal Time Coordinated），又叫世界标准时间，其实就是 `0000` 时区的时间。

Linux 下调整时区及更新时间

修改系统时间

```
$ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
```

修改 /etc/sysconfig/clock 文件，修改为：

```
ZONE="Asia/Shanghai"
UTC=false
ARC=false
```

校对时间

```
$ntpdate cn.ntp.org.cn
```

设置硬件时间和系统时间一致并校准

```
$/sbin/hwclock --systohc
```

定时同步时间设置

凌晨 5 点定时同步时间

```
echo "0 5 * * *  /usr/sbin/ntpdate cn.ntp.org.cn" >> /var/spool/cron/root
或者
echo "0 5 * * *  /usr/sbin/ntpdate 133.100.11.8" >> /var/spool/cron/root
```

登录提示信息

修改登录前的提示信息

(1) 系统级别的设置方法 **/etc/issue**

使用此方法时远程 ssh 连接的时候并不会显示

```
在登录系统输入用户名之前，可以看到上方有 WELCOME..... 之类的信息，这里会显示 LINUX 发行版本名称，内核版本号，日期，机器信息等等信息，

首先打开 /etc/issue 文件，可以看到里面是这样一段"Welcome to <LINUX 发行版本名称》-kernel 后接各项参数、"

参数的各项说明：
\r 显示 KERNEL 内核版本号；
\l 显示虚拟控制台号；
\d 显示当前日期；
\n 显示主机名；
\m 显示机器类型，即 CPU 架构，如 i386 等；

可以显示所有必要的信息：

Welcome to <LINUX 发行版本名称》-kernel \r (\l) \d \n \m.
```

修改登录成功后的信息

motd(message of the day)

修改登录成功后的提示信息在此文件中添加内容即可：/etc/motd

如：

```
//////////////////////////////////////
系统初始化配置提示
xxxx

应用联系人：xxxx 联系方式：xxxx
//////////////////////////////////////
```

CentOS 7 vs CentOS 6 的不同

运行相关

桌面系统

```
[CentOS6] GNOME 2.x
[CentOS7] GNOME 3.x（GNOME Shell）
```

文件系统

```
[CentOS6] ext4
[CentOS7] xfs
```

内核版本

```
[CentOS6] 2.6.x-x
[CentOS7] 3.10.x-x
```

启动加载器

```
[CentOS6] GRUB Legacy (+efibootmgr)
[CentOS7] GRUB2
```

防火墙

```
[CentOS6] iptables
[CentOS7] firewalld
```

默认数据库

```
[CentOS6] MySQL
[CentOS7] MariaDB
```

文件结构

```
[CentOS6] /bin, /sbin, /lib, and /lib64 在 / 下
[CentOS7] /bin, /sbin, /lib, and /lib64 移到 /usr 下
```

主机名

```
[CentOS6] /etc/sysconfig/network
[CentOS7] /etc/hostname
```

时间同步

```
[CentOS6]
$ ntp
$ ntpq -p

[CentOS7]
$ chrony
$ chronyc sources
```

修改时区

```
[CentOS6]
$ vim /etc/sysconfig/clock
   ZONE="Asia/Shanghai"
   UTC=fales
$ sudo ln -s /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

[CentOS7]
$ timedatectl set-timezone Asia/Shanghai
$ timedatectl status
```

修改语言

```
[CentOS6]
$ vim /etc/sysconfig/i18n
   LANG="en_US.utf8"
$ /etc/sysconfig/i18n
$ locale

[CentOS7]
$ localectl set-locale LANG=en_US.utf8
$ localectl status
```

重启关闭

```
1) 关闭
[CentOS6]
$ shutdown -h now

[CentOS7]
$ poweroff
$ systemctl poweroff

2) 重启
[CentOS6]
$ reboot
$ shutdown -r now

[CentOS7]
$ reboot
$ systemctl reboot

3) 单用户模式
[CentOS6]
$ init S

[CentOS7]
$ systemctl rescue

4) 启动模式
[CentOS6]
[GUICUI]
$ vim /etc/inittab
  id:3:initdefault:
[CUIGUI]
$ startx

[CentOS7]
[GUICUI]
$ systemctl isolate multi-user.target
[CUIGUI]
$systemctl isolate graphical.target
默认
$ systemctl set-default graphical.target

[CentOS6]
$ chkconfig service_name on/off

[CentOS7]
$ systemctl enable service_name
$ systemctl disable service_name
```

服务一览

```
[CentOS6]
$ chkconfig --list

[CentOS7]
$ systemctl list-unit-files
$ systemctl --type service
```

强制停止

```
[CentOS6]
$ kill -9 <PID>

[CentOS7]
$ systemctl kill --signal=9 sshd
```

网络

网络信息

```
[CentOS6]
$ netstat
$ netstat -I
$ netstat -n

[CentOS7]
$ ip -s l
$ ss
```

IP 地址 **MAC** 地址

```
[CentOS6]
$ ifconfig -a

[CentOS7]
$ ip address show
```

路由

```
[CentOS6]
$ route -n
$ route -A inet6 -n

[CentOS7]
$ ip route show
$ ip -6 route show
```

Linux 工具篇

1 编程相关

1.1 vim IDE 工具

- VIM 一键 IDE

1.2 Git 分布式管理系统

1.2.1 Git 基础

环境配置

- `git config --global user.name your_name`：设置你的用户名，提交会显示
- `git config --global user.email your_email`：设置你的邮箱
- `git config core.quotepath false`：解决中文文件名显示为数字问题

基本操作

- `git init`：初始化一个 git 仓库
- `git add <filename>`：添加一个文件到 git 仓库中
- `git commit -m "commit message"`：提交到本地
- `git push [remote-name] [branch-name]`：把本地的提交记录推送到远端分支
- `git pull`：更新仓库 `git pull` = `git fetch` + `git merge`
- `git checkout -- <file>`：还原未暂存 (staged) 的文件
- `git reset HEAD <file>...`：取消暂存，那么还原一个暂存文件，应该是先 `reset` 后 `checkout`
- `git stash`：隐藏本地提交记录，恢复的时候 `git stash pop`。这样可以在本地和远程有冲突的情况下，更新其他文件

分支

- `git branch <branch-name>`：基于当前 commit 新建一个分支，但是不切换到新分支
- `git branch -r`：查看远程的所有分支（常用）
- `git checkout -b <branch-name>`：新建并切换分支

- `git checkout <branch-name>`：切换分支(常用)
- `git branch -d <branch-name>`：删除分支
- `git push origin <branch-name>`：推送本地分支
- `git checkout -b <local-branch-name> origin/<origin-branch-name>`：基于某个远程分支新建一个分支开发
- `git checkout --track origin/<origin-branch-name>`：跟踪远程分支（创建跟踪远程分支，Git 在 `git push` 的时候不需要指定 `origin` 和 `branch-name` ，其实当我们 `clone` 一个 repo 到本地的时候，`master` 分支就是 origin/master 的跟踪分支，所以提交的时候直接 `git push`)。
- `git push origin :<origin-branch-name>`：删除远程分支

实践 --- 主分支 Master/ 开发分支 Develop

```
主分支只用来分布重大版本，日常开发应该在另一条分支上完成。我们把开发用的分支，叫做 Develop。

# Git 创建 Develop 分支
git checkout -b develop master

# 将 Develop 分支发布到 Master 分支

# 切换到 Master 分支
git checkout master

# 对 Develop 分支进行合并
git merge --no-ff develop
上一条命令的 --no-ff 参数是什么意思。默认情况下，Git 执行"快进式合并"（fast-farward merge），会直接将 Master 分支指向 Develop 分支。

# 删除本地分支
git branch -d develop
```

标签

- `git tag -a <tagname> -m <message>`：创建一个标签（用 -a 指定标签名，-m 指定说明文字） 如 `git tag -a v1.0 -m "version 1.0 released mitaka"`
- `git tag`：显示已有的标签
- `git show tagname`：显示某个标签的详细信息
- `git push origin v1.0` push 到远端仓库 如 `git push -u ${PWD##*/} master v1.0`
- `git checkout -b <tag-name>`：基于某个 tag 创建一个新的分支

Git shortcuts/aliases

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
```

1.2.2 知识点

基本命令让你快速的上手使用 Git，知识点能让你更好的理解 Git。

文件的几种状态

- untracked: 未被跟踪的，没有纳入 Git 版本控制，使用 `git add <filename>` 纳入版本控制
- unmodified: 未修改的，已经纳入版本控制，但是没有修改过的文件
- modified: 对纳入版本控制的文件做了修改，git 将标记为 modified
- staged: 暂存的文件，简单理解：暂存文件就是 add 之后，commit 之前的文件状态

理解这几种文件状态对于理解 Git 是非常关键的（至少可以看懂一些错误提示了）。

快照和差异

详细可看：Pro Git: Git 基础 中有讲到 直接记录快照，而非差异比较，这里只讲我个人的理解。

Git 关心的是文件数据整体的变化，其他版本管理系统（以 svn 为例）关心的某个具体文件的差异。这个差异是好理解的，也就是两个版本具体文件的不同点，比如某一行的某个字符发生了改变。

Git 不保存文件提交前后的差异，不变的文件不会发生任何改变，对于变化的文件，前后两次提交则保存两个文件。举个例子：

SVN:

1. 新建 3 个文件 a, b, c，做第一次提交 -> `version1 : file_a file_b file_c`
2. 修改文件 b， 做第二次提交（真正提交的是 修改后的文件 b 和修改前的 `file_b` 的 diff) -> `version2: diff_b_2_1`
3. 当我要 checkout version2 的时候，实际上得到的是 `file_a file_b+diff_b_2_1 file_c`

Git:

1. 新建 3 个文件 a, b, c，做第一次提交 -> `version1 : file_a file_b file_c`
2. 修改文件 b （得到 `file_b1`), 做第二次提交 -> `version2: file_a file_b1 file_c`
3. 当我要用 version2 的时候，实际上得到的是 `file_a file_b1 file_c`

上面的 `file_a file_b1 file_c` 就是 version2 的 快照。

Git 数据结构

Git 的核心数是很简单的，就是一个链表（或者一棵树更准确一些？无所谓了），一旦你理解了它的基本数据结构，再去看 Git，相信你有不同的感受。继续用上面的例子（所有的物理文件都对应一个 SHA-1 的值）

当我们做第一次提交时，数据结构是这样的：

```
sha1_2_file_map:
    28415f07ca9281d0ed86cdc766629fb4ea35ea38 => file_a
    ed5cfa40b80da97b56698466d03ab126c5eec5a9 => file_b
    1b5ca12a6cf11a9b89dbeee2e5431a1a98ea5e39 => file_c

commit_26b985d269d3a617af4064489199c3e0d4791bb5:
    base_info:
        Auther: "JerryZhang(chinajiezhang@gmail.com)"
        Date: "Tue Jul 15 19:19:22 2014 +0800"
        commit_content: "第一次提交"
    file_list:
        [1]: 28415f07ca9281d0ed86cdc766629fb4ea35ea38
        [2]: ed5cfa40b80da97b56698466d03ab126c5eec5a9
        [3]: 1b5ca12a6cf11a9b89dbeee2e5431a1a98ea5e39
        pre_commit: null
    next_commit: null
```

当修改了 `file_b`，再提交一次时，数据结构应该是这样的：

```
sha1_2_file_map:
    28415f07ca9281d0ed86cdc766629fb4ea35ea38 => file_a
    ed5cfa40b80da97b56698466d03ab126c5eec5a9 => file_b
    1b5ca12a6cf11a9b89dbeee2e5431a1a98ea5e39 => file_c
    39015ba6f80eb9e7fdad3602ef2b1af0521eba89 => file_b1

commit_26b985d269d3a617af4064489199c3e0d4791bb5:
    base_info:
        Auther: "JerryZhang(chinajiezhang@gmail.com)"
        Date: "Tue Jul 15 19:19:22 2014 +0800"
        commit_content: "第一次提交"
    file_list:
        [1]: 28415f07ca9281d0ed86cdc766629fb4ea35ea38
        [2]: ed5cfa40b80da97b56698466d03ab126c5eec5a9
        [3]: 1b5ca12a6cf11a9b89dbeee2e5431a1a98ea5e39
    pre_commit: commit_a08a57561b5c30b9c0bf33829349e14fad1f5cff
    next_commit: null

commit_a08a57561b5c30b9c0bf33829349e14fad1f5cff:
    base_info:
        Auther: "JerryZhang(chinajiezhang@gmail.com)"
        Date: "Tue Jul 15 22:19:22 2014 +0800"
        commit_content: "更新文件 b"
    file_list:
        [1]: 28415f07ca9281d0ed86cdc766629fb4ea35ea38
        [2]: 39015ba6f80eb9e7fdad3602ef2b1af0521eba89
        [3]: 1b5ca12a6cf11a9b89dbeee2e5431a1a98ea5e39
    pre_commit: null
    next_commit: commit_26b985d269d3a617af4064489199c3e0d4791bb5
```

当提交完第二次的时候，执行 `git log` ，实际上就是从 `commit_a08a57561b5c30b9c0bf33829349e14fad1f5cff` 开始遍历然后打印 `base_info` 而已。

实际的 git 实际肯定要比上面的结构 (（的信息）的）要复杂的多，但是它的核心思想应该是就是，每一次提交就是一个新的结点。通过这个结点，我可以找到所有的快照文件。再思考一下，什么是分支？什么是 Tags，其实他们可能只是某次提交的引用而已（一个 `tag_head_node` 指向了某一次提交的 node)。再思考怎么回退一个版本呢？指针偏移！依次类推，上面的基本命令都可以得到一个合理的解释。

理解 **git fetch** 和 **git pull** 的差异

上面我们说过 `git pull` 等价于 `git fetch` 和 `git merge` 两条命令。当我们 `clone` 一个 repo 到本地时，就有了本地分支和远端分支的概念（假定我们只有一个主分支），本地分支是 `master` ，远端分支是 `origin/master` 。通过上面我们对 Git 数据结构的理解， `master` 和 `origin/master` 可以想成是指向最新 commit 结点的两个指针。刚 `clone` 下来的 repo， `master` 和 `origin/master` 指针指向同一个结点，我们在本地提交一次， `origin` 结点就更新一次，此时 `master` 和 `orgin/master` 就不再相同了。很有可能别人已经 commit 改 repo 很多次了，并且进行了提交。那么我们的本地的 `origin/master` 就不再是远程服务器上的最新的位置了。 `git fetch` 干的就是从服务器上同步服务器上最新的 `origin/master` 和一些服务器上新的记录 / 文件到本地。而 `git merge` 就是合并操作了（解决文件冲突）。 `git push` 是把本地的 `origin/master` 和 `master` 指向相同的位置，并且推送到远程的服务器。

1.2.3 其他操作

解决 GitHub commit 次数过多.git 文件过大

完全重建版本库

```
# rm -rf .git
# git init
# git add .
# git commit -a -m "[Update] 合并之前所有 commit"
# git remote add origin <your_github_repo_url>
# git push -f -u origin master
```

HTTP request failed

使用 git clone 失败

```
[root@localhost ~]# git clone https://github.com/meetbill/Vim.git
Initialized empty Git repository in /root/Vim/.git/
error:  while accessing https://github.com/meetbill/Vim.git/info/refs

fatal: HTTP request failed
```

解决方法

```
#git config --global http.sslVerify false
```

设置 Wiki 只能自己编写，其他人员只读

在项目中的设置中勾选 `Restrict editing to collaborators only`

修改最后一次 commit 操作

有时候我们提交完了才发现漏掉了几个文件没有加，或者提交信息写错了。想要撤消刚才的提交操作，可以使用 --amend 选项重新提交：

```
$ git commit -a -m 'initial commit'
$ git add forgotten_file
$ git commit --amend -m 'new commit'
```

1.2.4 Git tips

(1) 命令简化

```
cd git_repo（替换为项目名字）
git remote add ${PWD##*/} git@github.com:meetbill/${PWD##*/}.git
git push -u ${PWD##*/} master
```

(2) 提升 git 使用体验

> - git 命令自动补全
> - 命令行显示 git 信息

2 运维相关

2.1 sed

2.2 awk

AWK 是贝尔实验室 1977 年搞出来的文本处理工具。

之所以叫 AWK 是因为其取了三位创始人 Alfred Aho，Peter Weinberger, 和 Brian Kernighan 的 Family Name 的首字符

2.2.1 基础知识

分隔符

默认情况下， awk 使用空格当作分隔符。分割后的字符串可以使用 $1, $2 等访问。

上面提到过，我们可以使用 -F 来指定分隔符。 fs 如果是一个字符，可以直接跟在 -F 后面，比如使用冒号当作分隔符就是 -F: . 如果分隔符比较复杂，就需要使用正则表达式来表示这个分隔符了。 正则表达式需要使用引号引起来。 比如使用‘ab’ 当作分隔符，就是 -F 'ab' 了。 使用 a 或 b 作为分隔符，就是 -F '[ab]' 了。 关于正则表达式这里不多说了。

内建变量

```
$0     当前记录（这个变量中存放着整个行的内容）
$1~$n     当前记录的第 n 个字段，字段间由 FS 分隔
FS     输入字段分隔符 默认是空格或 Tab
NF     当前记录中的字段个数，就是有多少列
NR     已经读出的记录数，就是行号，从 1 开始，如果有多个文件话，这个值也是不断累加中。
FNR     当前记录数，与 NR 不同的是，这个值会是各个文件自己的行号
RS     输入的记录分隔符， 默认为换行符
OFS     输出字段分隔符， 默认也是空格
ORS     输出的记录分隔符，默认为换行符
FILENAME     当前输入文件的名字
```

转义

一般字符在双引号之内就可以直接原样输出了。 但是有部分转义字符，需要使用反斜杠转义才能正常输出。

```
\\   A literal backslash.
\a   The “alert” character; usually the ASCII BEL character.
\b   backspace.
\f   form-feed.
\n   newline.
\r   carriage return.
\t   horizontal tab.
\v   vertical tab.
\xhex digits
\c   The literal character c.
```

模式

```
~ 表示模式开始，与 == 相比不是精确比较
/ / 中是模式
! 模式取反
```

单引号

当需要输出单引号时，直接转义发现会报错。 由于 awk 脚本并不是直接执行，而是会先进行预处理，所以需要两次转义。 awk 支持递归引号。单引号内可以输出转义的单引号，双引号内可以输出转义的双引号。

比如需要输出单引号，则需要下面这样：

```
> awk 'BEGIN{print "\""}'
"
>  awk 'BEGIN{print "'\''"}'
'
```

当然，更简单的方式是使用十六进制来输出。

```
awk 'BEGIN{print "\x27"}'
```

2.2.2 脚本

```
BEGIN{ 这里面放的是执行前的语句 }
END {这里面放的是处理完所有的行后要执行的语句 }
{这里面放的是处理每一行时要执行的语句}
```

2.2.3 运算与编程

awk 是弱类型语言，变量可以是串，也可以是数字，这依赖于实际情况。所有的数字都是浮点型。

例如

```
//9
echo 5 4 | awk '{ print $1 + $2 }'

//54
echo 5 4 | awk '{ print $1 $2 }'

//"5 4"
echo 5 4 | awk '{ print $1, $2 }'

0-1-2-3-4-5-6
echo 6 | awk '{ for (i=0; i<=$0; i++){ printf (i==0?i:"-"i); }printf "\n";}'
```

Example

假设我们有一个日期 2014/03/27, 我们想处理为 2014-03-27. 我们可以使用下面的代码实现。

```
echo "2014/03/27" | awk -F/  '{print $1"-"$2"-"$3}'
```

假设 处理的日期都在 date 文件里。 我们可以导入文件来操作

文件名 date

```
2014/03/27
2014/03/28
2014/03/29
```

命令

```
awk -F/ '{printf "%s-%s-%s\n",$1,$2,$3}'  date
```

输出

```
2014-03-27
2014-03-28
2014-03-29
```

统计

```
awk '{sum+=$5} END {print sum}'
```

2.2.4 AWK 中输出外部变量

通过双引号内加个单引号将外部变量进行输出

```
wang="hello"
echo meetbill | awk '{print "'$wang' " $1}'
```

2.2.5 AWK if

必须用在{}中，且比较内容用 () 扩起来

```
awk -F: '{if($1~/mail/) print $1}'    /etc/passwd                              // 简写
awk -F: '{if($1~/mail/) {print $1}}'  /etc/passwd                              // 全写
awk -F: '{if($1~/mail/) {print $1} else {print $2}}' /etc/passwd               //if...els
e...
```

- 条件表达式
  - `== != > >=`
- 逻辑运算符
  - `&& ||` 如：查看使用了 CPU 0 核的进程

```
# ps -eF，其中 PSR 就是 (processor that process is currently assigned to.)
 或者 ps -eo pid,command,args,psr
ps -eF |awk '{if($7==0) print $0}'
```

2.3 排查 java CPU 性能问题

show-busy-java-threads.sh

```
curl -o show-busy-Java-threads.sh https://raw.githubusercontent.com/meetbill/op_practi
ce_code/master/Linux/op/show-busy-java-threads.sh
```

用于快速排查 `Java` 的 `CPU` 性能问题 (`top us` 值过高），自动查出运行的 `Java` 进程中消耗 `CPU` 多的线程，并打印出其线程栈，从而确定导致性能问题的方法调用。

PS，如何操作可以参见 @bluedavy 的《分布式 Java 应用》的【5.1.1 cpu 消耗分析】一节，说得很详细：

1. `top` 命令找出有问题 `Java` 进程及线程 `id` ：
   i. 开启线程显示模式
   ii. 按 `CPU` 使用率排序
   iii. 记下 `Java` 进程 `id` 及其 `CPU` 高的线程 `id`
2. 用进程 `id` 作为参数， `jstack` 有问题的 `Java` 进程
3. 手动转换线程 `id` 成十六进制（可以用 `printf %x 1234` ）
4. 查找十六进制的线程 `id` （可以用 `grep` ）

5. 查看对应的线程栈

查问题时，会要多次这样操作以确定问题，上面过程太繁琐太慢了。

2.3.1 用法

```
show-busy-java-threads.sh
# 从 所有的 Java 进程中找出最消耗 CPU 的线程（缺省 5 个），打印出其线程栈。

show-busy-java-threads.sh -c 《要显示的线程栈数》

show-busy-java-threads.sh -c 《要显示的线程栈数》 -p 《指定的 Java Process>

###############################
# 注意：
###############################
# 如果 Java 进程的用户 与 执行脚本的当前用户 不同，则 jstack 不了这个 Java 进程。
# 为了能切换到 Java 进程的用户，需要加 sudo 来执行，即可以解决：
sudo show-busy-java-threads.sh
```

2.3.2 示例

```
$ show-busy-java-threads.sh
[1] Busy(57.0%) thread(23355/0x5b3b) stack of java process(23269) under user(admin):
"pool-1-thread-1" prio=10 tid=0x000000005b5c5000 nid=0x5b3b runnable [0x000000004062c0
00]
   java.lang.Thread.State: RUNNABLE
    at java.text.DateFormat.format(DateFormat.java:316)
    at com.xxx.foo.services.common.DateFormatUtil.format(DateFormatUtil.java:41)
    at com.xxx.foo.shared.monitor.schedule.AppMonitorDataAvgScheduler.run(AppMonitorDa
taAvgScheduler.java:127)
    at com.xxx.foo.services.common.utils.AliTimer$2.run(AliTimer.java:128)
    at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:
886)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
    at java.lang.Thread.run(Thread.java:662)

[2] Busy(26.1%) thread(24018/0x5dd2) stack of java process(23269) under user(admin):
"pool-1-thread-2" prio=10 tid=0x000000005a968800 nid=0x5dd2 runnable [0x00000000420e90
00]
   java.lang.Thread.State: RUNNABLE
    at java.util.Arrays.copyOf(Arrays.java:2882)
    at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:100)
    at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:572)
    at java.lang.StringBuffer.append(StringBuffer.java:320)
    - locked <0x00000007908d0030> (a java.lang.StringBuffer)
    at java.text.SimpleDateFormat.format(SimpleDateFormat.java:890)
    at java.text.SimpleDateFormat.format(SimpleDateFormat.java:869)
    at java.text.DateFormat.format(DateFormat.java:316)
    at com.xxx.foo.services.common.DateFormatUtil.format(DateFormatUtil.java:41)
    at com.xxx.foo.shared.monitor.schedule.AppMonitorDataAvgScheduler.run(AppMonitorDa
taAvgScheduler.java:126)
    at com.xxx.foo.services.common.utils.AliTimer$2.run(AliTimer.java:128)
    at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:
886)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
...
```

上面的线程栈可以看出，`CPU` 消耗最高的 2 个线程都在执行 `java.text.DateFormat.format`，业务代码对应的方法是 `shared.monitor.schedule.AppMonitorDataAvgScheduler.run`。可以基本确定：

- `AppMonitorDataAvgScheduler.run` 调用 `DateFormat.format` 次数比较频繁。
- `DateFormat.format` 比较慢。（这个可以由 `DateFormat.format` 的实现确定。）

多个执行几次 `show-busy-java-threads.sh`，如果上面情况高概率出现，则可以确定上面的判定。# 因为调用越少代码执行越快，则出现在线程栈的概率就越低。

分析 `shared.monitor.schedule.AppMonitorDataAvgScheduler.run` 实现逻辑和调用方式，以优化实现解决问题。

- oldratlee

- silentforce 改进此脚本，增加对环境变量 `JAVA_HOME` 的判断。 #15
- liuyangc3
  - 优化性能，通过 `read -a` 简化反复的 `awk` 操作 #51
  - 发现并解决 `jstack` 非当前用户 `Java` 进程的问题 #50

3 系统相关

3.1 screen

现在很多时候我们的开发环境都已经部署到云端了，直接通过 SSH 来登录到云端服务器进行开发测试以及运行各种命令，一旦网络中断，通过 SSH 运行的命令也会退出，这个发让人发疯的。

好在有 screen 命令，它可以解决这些问题。我使用 screen 命令已经有三年多的时间了，感觉还不错。

3.1.1 screen 使用

新建一个 **Screen Session**

```
$ screen -S screen_session_name
```

将当前 **Screen Session** 放到后台

```
$ CTRL + A + D
```

唤起一个 **Screen Session**

```
$ screen -r screen_session_name
```

分享一个 **Screen Session**

```
$ screen -x screen_session_name
```

通常你想和别人分享你在终端里的操作时可以用此命令。

终止一个 **Screen Session**

```
$ exit
$ CTRL + D
```

查看一个 **screen** 里的输出

当你进入一个 screen 时你只能看到一屏内容，如果想看之前的内容可以如下：

```
$ Ctrl + a ESC
```

以上意思是进入 Copy mode，拷贝模式，然后你就可以像操作 VIM 一样查看 screen session 里的内容了。

可以 Page Up 也可以 Page Down。

3.1.2 开启 **screen** 状态栏

```
#curl -o screen.sh https://raw.githubusercontent.com/meetbill/op_practice_code/master/
Linux/tools/screen.sh
#sh screen.sh
```

4 网络相关

4.1 curl

4.1.1 HTTP 请求

GET 和 POST 是 HTTP 请求的两种基本方法，最直观的区别就是 GET 把参数包含在 URL 中，POST 通过 request body 传递参数。

```
    在万维网世界，TCP 就像汽车，我们用 TCP 来运输数据，它很可靠，
从来不会发生丢件少件的现象。但是如果路上跑的全是看起来一模一样
的汽车，那这个世界看起来是一团混乱，送急件的汽车可能被前面满载
货物的汽车拦堵在路上，整个交通系统一定会瘫痪。为了避免这种情况
发生，交通规则 HTTP 诞生了。HTTP 给汽车运输设定了好几个服务类别，
有 GET, POST, PUT, DELETE 等等，HTTP 规定，当执行 GET 请求的时候，
要给汽车贴上 GET 的标签（设置 method 为 GET)，而且要求把传送的数
据放在车顶上 (url) 以方便记录。如果是 POST 请求，就要在车上贴上
POST 的标签，并把货物放在车厢里。

    当然，你也可以在 GET 的时候往车厢内偷偷藏点货物，但是这是很不
光彩；也可以在 POST 的时候在车顶上也放一些数据。
```

4.1.2 curl 基础

在介绍前，我需要先做两点说明：

1. 下面的例子中会使用 httpbin.org ，httpbin 提供客户端测试 http 请求的服务，非常好用，具体可以查看他的网站。
2. 大部分没有使用缩写形式的参数，例如我使用 `--request` 而不是 `-X` ，这是为了好记忆。

下面开始简单介绍几个命令：

get

> - curl protocol://address:port/url?args

直接以个 GET 方式请求一个 url，输出返回内容：

```
curl httpbin.org
```

返回

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv='content-type' value='text/html;charset=utf8'>
  <meta name='generator' value='Ronn/v0.7.3 (http://github.com/rtomayko/ronn/tree/0.7.
3)'>
  <title>httpbin(1): HTTP Client Testing Service</title>
  <style type='text/css' media='all'>
  /* style: man */
  body#manpage {margin:0}
  .mp {max-width:100ex;padding:0 9ex 1ex 4ex}
  .mp p,.mp pre,.mp ul,.mp ol,.mp dl {margin:0 0 20px 0}
  .mp h2 {margin:10px 0 0 0}
......
```

post

> - curl --data "args" "protocol://address:port/url"
>   - -d/--data HTTP POST 方式传送数据 * --data-ascii 以 ascii 的方式 post 数据
>   - --data-binary 以二进制的方式 post 数据

使用 `--request` 指定请求类型， `--data` 指定数据，例如：

```
curl httpbin.org/post --request POST --data "name=keenwon&website=http://keenwon.com"
```

返回：

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "name": "tomshine",
    "website": "http://tomshine.xyz"
  },
  "headers": {
    "Accept": "*/*",
    "Content-Length": "41",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.43.0"
  },
  "json": null,
  "origin": "121.35.209.62",
  "url": "http://httpbin.org/post"
}
```

这个返回值是 httpbin 输出的，可以清晰的看出我们发送了什么数据，非常实用。

form 表单提交

form 表单提交使用 `--form`，使用 `@` 指定本地文件，例如我们提交一个表单，有字段 name 和文件 f：

```
curl httpbin.org/post --form "name=tomshine" --form "f=@/Users/tomshine/test.txt"
```

返回：

```
{
  "args": {},
  "data": "",
  "files": {
    "f": "Hello Curl!\n"
  },
  "form": {
    "name": "tomshine"
  },
  "headers": {
    "Accept": "*/*",
    "Content-Length": "296",
    "Content-Type": "multipart/form-data; boundary=------------------------3bd3dc24dca
6daf2",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.43.0"
  },
  "json": null,
  "origin": "112.95.153.98",
  "url": "http://httpbin.org/post"
}
```


4.1.3 curl 深入

显示头信息

使用 `--include` 在输出中包含头信息，使用 `--head` 只返回头信息，例如：

```
curl httpbin.org/post --include --request POST --data "name=tomshine"
```

返回：

```
HTTP/1.1 200 OK
Server: nginx
Date: Sun, 18 Sep 2016 01:23:28 GMT
Content-Type: application/json
Content-Length: 363
Connection: keep-alive
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "name": "tomshine"
  },
  "headers": {
    "Accept": "*/*",
    "Content-Length": "13",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.43.0"
  },
  "json": null,
  "origin": "121.35.209.62",
  "url": "http://httpbin.org/post"
}
```

再例如，只显示头信息的话：

```
curl httpbin.org --head
```

返回：

```
HTTP/1.1 200 OK
Server: nginx
Date: Sun, 18 Sep 2016 01:24:29 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 12150
Connection: keep-alive
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
```

详细显示通信过程

使用 `--verbose` 显示通信过程，例如：

```
curl httpbin.org/post --verbose --request POST --data "name=tomshine"
```

返回：

```
*   Trying 54.175.219.8...
* Connected to httpbin.org (54.175.219.8) port 80 (#0)
> POST /post HTTP/1.1
> Host: httpbin.org
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 13
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 13 out of 13 bytes
< HTTP/1.1 200 OK
< Server: nginx
< Date: Sun, 18 Sep 2016 01:25:03 GMT
< Content-Type: application/json
< Content-Length: 363
< Connection: keep-alive
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Credentials: true
<
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "name": "tomshine"
  },
  "headers": {
    "Accept": "*/*",
    "Content-Length": "13",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.43.0"
  },
  "json": null,
  "origin": "121.35.209.62",
  "url": "http://httpbin.org/post"
}
* Connection #0 to host httpbin.org left intact
```

设置头信息

使用 `--header` 设置头信息，`httpbin.org/headers` 会显示请求的头信息，我们测试下：

```
curl httpbin.org/headers --header "a:1"
```

返回：

```
{
  "headers": {
    "A": "1",
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.43.0"
  }
}
```

同样的，可以使用 `--header` 设置 `User-Agent` 等。

Referer 字段

设置 `Referer` 字段很简单，使用 `--referer` ，例如：

```
curl httpbin.org/headers --referer http://tomshine.xyz
```

返回：

```
{
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "Referer": "http://tomshine.xyz",
    "User-Agent": "curl/7.43.0"
  }
}
```

包含 cookie

使用 `--cookie` 来设置请求的 cookie，例如：

```
curl httpbin.org/headers --cookie "name=tomshine;website=http://tomshine.xyz"
```

返回：

```
{
  "headers": {
    "Accept": "*/*",
    "Cookie": "name=tomshine;website=http://tomshine.xyz",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.43.0"
  }
}
```

自动跳转

使用 `--location` 参数会跟随链接的跳转，例如：

```
curl httpbin.org/redirect/1 --location
```

httpbin.org/redirect/1 会 302 跳转，所以返回：

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.43.0"
  },
  "origin": "121.35.209.62",
  "url": "http://httpbin.org/get"
}
```

http 认证

当页面需要认证时，可以使用 `--user` ，例如：

```
curl httpbin.org/basic-auth/tomshine/123456 --user tomshine:123456
```

返回：

```
{
  "authenticated": true,
  "user": "tomshine"
}
```

4.2 tcpdump

4.2.1 tcp 三次握手和四次挥手

三次握手

A 主动打开连接，B 被动打开连接

```
(1) 第一次握手：建立连接时，客户端 A 发送 SYN 包 (SYN=x) 到服务器 B，并进入 SYN_SEND 状态，等待服务器 B 确认。
(2) 第二次握手：服务器 B 收到 SYN 包，必须确认客户 A 的 SYN(ACK=x+1)，同时自己也发送一个 SYN 包 (SYN=y)，即 SYN+ACK 包，此时服务器 B 进入 SYN_RECV 状态。
(3) 第三次握手：客户端 A 收到服务器 B 的 SYN＋ACK 包，向服务器 B 发送确认包 ACK(ACK=y+1)，此包发送完毕，客户端 A 和服务器 B 进入 ESTABLISHED 状态，完成三次握手。
完成三次握手，客户端与服务器开始传送数据。
```

为什么 A 还要发送一次确认呢？

主要是为了防止已失效的连接请求报文段突然有传送到 B，因而产生错误。

正常情况

A 发出连接请求，但是因为连接请求报文丢失为未收到确认。于是 A 在重传一次连接请求，后来收到了确认，建立了连接。数据传输完毕后，就释放了连接。A 共发送两个连接请求报文段，其中第一个丢失第二个到达了 B。

异常情况 A 发出的第一个连接请求报文段并没有丢失，而是在某些网络节点长时间滞留，导致延误到连接释放之后才到达了 B，本来这是一个早已经失效的报文段，但是 B 收到此失效的连接请求报文段之后，误以为是 A 又发出一次新的连接请求，于是就向 A 发送确认报段，同意建立连接。假如不采用三次握手，那么只要 B 发出确认，新的连接就建立了。由于现在 A 并没有发出建立连接的请求，因此不会理睬 B 的确认，也不会向 B 发送数据，但 B 却以为新的运输连接已经建立了，并且一直等待 A 发来数据。B 的资源就这样白白的浪费了， 采用三次握手的办法可以防止上述现象的发生。例如刚才的情况，A 不会向 B 的确认发出确认。B 由于接收不到确认，就知道 A 并没有要求建立连接。

使用 tcpdump 来验证 TCP 的三次握手

使用 ssh localhost 来连接主机，使用使用 tcpdump 来验证 TCP 的三次握手

```
[root@localhost apue]# tcpdump -i lo tcp -S
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 65535 bytes
1 15:08:03.039511 IP6 localhost.44910 > localhost.ssh: Flags [S], seq 3120401438, win
32752, options [mss 16376,sackOK,TS val 1319756 ecr 0,nop,wscale 7], length 0
2 15:08:03.039546 IP6 localhost.ssh > localhost.44910: Flags [S.], seq 404185237, ack
3120401439, win 32728, options [mss 16376,sackOK,TS val 1319756 ecr 1319756,nop,wscale
 7], length 0
3 15:08:03.039576 IP6 localhost.44910 > localhost.ssh: Flags [.], ack 404185238, win 2
56, options [nop,nop,TS val 1319756 ecr 1319756], length 0
4 15:08:03.064809 IP6 localhost.ssh > localhost.44910: Flags [P.], seq 404185238:40418
5259, ack 3120401439, win 256, options [nop,nop,TS val 1319781 ecr 1319756], length 21
15:08:03.064944 IP6 localhost.44910 > localhost.ssh: Flags [.], ack 404185259, win 256
, options [nop,nop,TS val 1319781 ecr 1319781], length 0
```

第一行就是第一次握手，客户端向服务器发送 SYN 标志 (Flags [S])，其中 seq = 3120401438； 第二行就是第二次握手，服务器向客户端进行 SYN+ACK 响应 (Flags [S.]), 其中 seq 404185237, ack 3120401439（是客户端的 seq+1 的值） 第三行就是第三次握手，客户端向服务器进行 ACK 响应 (Flags [.]), 其中 ack 404185238（就是服务器的 seq+1 的值）。

四次挥手

通信传输结束后，通信的双方都可以释放连接，现在 A 和 B 都处于 ESTABLISHED 状态。

由于 TCP 连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动，一个 TCP 连接在收到一个 FIN 后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

(1) 客户端 A 发送一个 FIN 包，用来关闭客户 A 到服务器 B 的数据传送。序列号 seq = u，它等于前面已传送过的数据的最后一个字节的序号加 1. 这时 A 进入 FIN-WAIT-1（终止等待 1) 状态，等待 B 的确认。

(2) 服务器 B 收到这个 FIN，它发回一个 ACK，确认序号是 ack = u +1。和 SYN 一样，一个 FIN 将占用一个序号。这个报文段自己的序号是 v，等于 B 前面已经传送过的数据的最后一个字节的序号加 1。然后 B 进程进入 CLOSE-WAIT（关闭等待）状态。TCP 服务器进程这时应通知高层应用进程，因而从 A 到 B 的这个方向的连接就释放了，这时的 TCP 连接处于半关闭 (half-close) 状态， 即 A 已经没有数据要发送了，但是 B 若发送数据，A 仍要接收，也就是说从 B 到 A 这个方向的连接并没有关闭，这个状态可能会持续一些时间。A 收到来到 B 的确认后就进入 FIN-WAIT-2（终止等待 2) 状态，等待 B 发出的连接释放报文段。

(3) 若 B 已经没有要向 A 发送的数据，其应用进程就会通知 TCP 释放连接，这时 B 发出的连接释放报文段必须使 FIN = 1。现假定 B 的序号为 w（在半关闭状态 B 可能要发送一些数据）。B 还必须重复上次发送过的确认号 ack = u +1。这时 B 就进入了 LAST-ACK（最后确认）状态，等待 A 的确认。

(4)A 在收到 B 的连接释放报文段后，必须对此发出确认。在确认报文段中把 ACK 置 1，确认号 ack = w + 1，而自己的序号是 seq = u + 1（根据 TCP 标准，前面发送过的 FIN 报文段要消耗一个序号），然后进入到 TIME-WAIT（时间等待）状态。

此时的 ***TCP*** 还没有完全的释放掉。必须经过时间等待计时器 ***(TIME-WAIT timer)*** 设置的时间 ***2MSL*** 后，***A*** 才进入到 ***CLOSED*** 状态。

> MSL 叫做最长报文段寿命，它是任何报文段被丢弃前在网络内的最长时间。

2MSL 也就是这个时间的两倍，RFC 建议设置为 2 分钟，但是 2 分钟可能太长了，因此 TCP 允许不同的实现使用更小的 MSL 值。

因此从 A 进入到 TIME-WAIT 状态后，要经过 4 分钟才能进入 CLOSED 状态，此案开始建立下一个新的连接。当 A 撤销相应的传输控制块 TCP 后，就结束了 TCP 连接。

使用 tcpdump 来验证 TCP 的四次挥手

退出 ssh 连接的主机，使用使用 tcpdump 来验证 TCP 的四次挥手

```
15:14:58.836149 IP6 localhost.44911 > localhost.ssh: Flags [P.], seq 1823848744:182384
8808, ack 3857143125, win 305, options [nop,nop,TS val 1735551 ecr 1735551], length 64
15:14:58.836201 IP6 localhost.44911 > localhost.ssh: Flags [F.], seq 1823848808, ack 3
857143125, win 305, options [nop,nop,TS val 1735551 ecr 1735551], length 0
15:14:58.837850 IP6 localhost.ssh > localhost.44911: Flags [.], ack 1823848809, win 31
8, options [nop,nop,TS val 1735554 ecr 1735551], length 0
15:14:58.842130 IP6 localhost.ssh > localhost.44911: Flags [F.], seq 3857143125, ack 1
823848809, win 318, options [nop,nop,TS val 1735559 ecr 1735551], length 0
15:14:58.842150 IP6 localhost.44911 > localhost.ssh: Flags [.], ack 3857143126, win 30
5, options [nop,nop,TS val 1735559 ecr 1735559], length 0
```

seq start:end 意思是初始序列号：结束序列号，***end = start + length,*** 但是接受包的结束序号应该是 ***end - 1***。

比如 start = 1，length = 3，那么真正的结束序号是 1+3-1 = 3, 因为开始序号也算在内的！

seq 1823848744:1823848808 意思是初始序列号：结束序列号，其实后面那个就是前面那个加上包长度 length = 64，实际上结束序列号是没有使用的，真正使用的序号是开始序号到结束序号 -1，也就是 1823848807

第一次挥手中客户端发送 FIN 即 [F.] seq u = 1823848808，也就是上次发送的数据的最后一个字节的序号加 1

第二次挥手中服务器发送 ACK 即 [.] ack 1823848809 = u + 1

第三次挥手中服务器发送 FIN 即 [F.] seq v = 3857143125， ack 1823848809 = u + 1

第四次挥手中客户端发送 ACK 即 [.] ack 3857143126 = v + 1

1. 默认情况下（不改变 socket 选项），当你调用 close 函数时，如果发送缓冲中还有数据，TCP 会继续把数据发送完。
2. 发送了 `FIN 只是表示这端不能继续发送数据（应用层不能再调用 send 发送）` ，但是还可以接收数据。
3. 应用层如何知道对方关闭？

在最简单的阻塞模型中，当调用 recv 时，如果返回 0，则表示对方关闭，

在这个时候通常的做法就是也调用 close，那么 TCP 层就发送 FIN，继续完成四次握手；

如果不调用 ***close***，那么对方就会处于 ***FIN_WAIT_2*** 状态，而本端则会处于 ***CLOSE_WAIT*** 状态；

1. 在很多时候，TCP 连接的断开都会由 TCP 层自动进行，例如你 CTRL+C 终止你的程序，TCP 连接依然会正常关闭。

4.2.2 tcpdump 使用

针对特定网口抓包 **(-i** 选项）

```
tcpdump -i eth0
```

指定抓包端口

```
tcpdump -i eth0 port 22
```

抓取特定目标 **ip** 和端口的包

```
tcpdump -i eth0 dst 10.70.121.92 and port 22
```

增加抓包时间戳 **(-tttt** 选项）

```
tcpdump -n -tttt -i eth0
```

4.3 nc

nc 检测端口更方便，同时批量进行检测端口的话是非常好的工具

4.3.1 语法

nc [-hlnruz][-g《网关...>][-G《指向器数目》][-i《延迟秒数》][-o《输出文件》][-p《通信端口》][-s《来源位址》][-v...][-w《超时秒数》][主机名称]『通信端口...]

```
参数说明：
-g《网关》 设置路由器跃程通信网关，最丢哦可设置 8 个。
-G《指向器数目》 设置来源路由指向器，其数值为 4 的倍数。
-h 在线帮助。
-i《延迟秒数》 设置时间间隔，以便传送信息及扫描通信端口。
-l 使用监听模式，管控传入的资料。
-n 直接使用 IP 地址，而不通过域名服务器。
-o《输出文件》 指定文件名称，把往来传输的数据以 16 进制字码倾倒成该文件保存。
-p《通信端口》 设置本地主机使用的通信端口。
-r 乱数指定本地与远端主机的通信端口。
-s《来源位址》 设置本地主机送出数据包的 IP 地址。
-u 使用 UDP 传输协议。
-v 显示指令执行过程。
-w《超时秒数》 设置等待连线的时间。
-z 使用 0 输入 / 输出模式，只在扫描通信端口时使用。

```

4.3.2 TCP 端口扫描

```
# nc -v -z -w2 10.20.144.145 1-100
Connection to 10.20.144.145 22 port [tcp/ssh] succeeded!
nc: connect to 10.20.144.145 port 23 (tcp) failed: Connection refused
nc: connect to 10.20.144.145 port 24 (tcp) failed: Connection refused
nc: connect to 10.20.144.145 port 25 (tcp) failed: Connection refused
...
Connection to 10.20.144.145 80 port [tcp/http] succeeded!
...
扫描 10.20.144.145 的端口 范围是 1-100
```
不加 -v 时仅输出 succeeded 的结果

4.3.3 扫描 UDP 端口

```
# nc -u -z -w2 10.20.144.145 1-1000 // 扫描 10.20.144.145 的端口 范围是 1-1000
扫描指定端口
```

5 日志相关操作

5.1 截取某段时间内的日志

```
sed -n '/2018-03-06 15:25:00/,/2018-03-06 15:30:00/p' access.log >25-30.log
```

5.2 处理日志文件中上下关联的两行

```
awk 'pattern { action  };pattern { action  };'
```

凡是被 {} 包裹的，就是 action, 凡是没有被{}包裹的，就是 pattern,

文件 d.txt 如下内容

```
ggg 1
portals: 192.168.5.41:3260
werew 2
portals: 192.168.5.43:3260
```

如何把文件 d.txt 内容变为如下内容

```
ggg 192.168.5.41:3260
werew 192.168.5.43:3260
```

方法

```
awk '/port/{print a" "$2}{a=$1}' d.txt
```

处理第一行的时候，以 port 开头吗？很明显，不以 port 开头，所以那个 pattern 不匹配，action 不执行。但执行了后面的 a=$1

处理第二行的时候，以 port 开头，打印出来 a 和本行 $2，再处理就是个循环过程。

总之，编写模式匹配时候，匹配的模式为第二行中的内容

Linux 安全

1 禁止 ping

禁止系统响应任何从外部 / 内部来的 ping 请求攻击者一般首先通过 ping 命令检测此主机或者 IP 是否处于活动状态 ，如果能够 ping 通某个主机或者 IP，那么攻击者就认为此系统处于活动状态，继而进行攻击或破坏。如果没有人能 ping 通机器并收到响应，那么就可以大大增强服务器的安全性，linux 下可以执行如下设置，禁止 ping 请求：

```
[root@localhost ~]#echo "1"> /proc/sys/net/ipv4/icmp_echo_ignore_all
```

默认情况下"icmp_echo_ignore_all"的值为"0"，表示响应 ping 操作。

可以加上面的一行命令到 /etc/rc.d/rc.local 文件中，以使每次系统重启后自动运行

2 禁止密码登陆

3 ssh 防暴力破解及提高 ssh 安全

4 运维操作审计

添加运维操作审计工具

5 双因子认证

海上生明月，天涯共此时！

5.1 安装及配置篇

5.1.1 环境

server：CentOS 6.5/CentOS 7.3

5.1.2 查看系统时间

使用外网机器时，创建的时候有可能不是北京时间

```
[root@centos ~]#date
Sun Aug 14 23:18:41 EDT 2011
[root@centos ~]# rm -rf /etc/localtime
[root@centos ~]# ln -s /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
```

5.1.3 安装 google authenticator

安装 EPEL 源并安装 google_authenticator

```
#yum -y install epel-release
#yum -y install google-authenticator
```

5.1.4 为 SSH 服务器用 Google 认证器

配置 **/etc/pam.d/sshd**

```
# CentOS 6.5 在"auth       include      password-auth"行前添加如下内容
# CentOS 7 在"auth       substack     password-auth"行前添加如下内容
auth       required pam_google_authenticator.so
```

即先 google 方式认证再 linux 密码认证

修改 **SSH** 服务配置 **/etc/ssh/sshd_config**

ChallengeResponseAuthentication no->yes

```
sed -i 's#^ChallengeResponseAuthentication no#ChallengeResponseAuthentication yes#' /e
tc/ssh/sshd_config
```

重启 **SSH** 服务

```
# CentOS6
#service sshd restart

# CentOS7
# systemctl restart sshd
```

关掉 **selinux**

```
# setenforce 0
# 修改 /etc/selinux/config 文件 将 SELINUX=enforcing 改为 SELINUX=disabled
```

5.1.5 生成验证密钥

在 Linux 主机上登陆需要认证的用户运行 Google 认证器（我这是使用 root 用户演示的）

```
$google-authenticator
```

直接一路输入 yes 即可，询问内容如下，想了解的可以看下

```
Do you want me to update your "~/.google_authenticator" file (y/n):y
应急码的保存路径

Do you want to disallow multiple uses of the same authentication token?
This restricts you to one login about every 30s,
but it increases your chances to notice or even prevent man-in-the-middle attacks (y/n
)
是否禁止一个口令多用，自然也是答 y

By default, tokens are good for 30 seconds and in order to compensate for possible tim
e-skew between the client and the server,
we allow an extra token before and after the current time. If you experience problems
with poor time synchronization, you can increase the window from its default size of 1
:30min to about 4min. Do you want to do so (y/n)
问是否打开时间容错以防止客户端与服务器时间相差太大导致认证失败。
这个可以根据实际情况来。如果一些 Android 平板电脑不怎么连网的，可以答 y 以防止时间错误导致认证失败。

If the computer that you are logging into isn't hardened against brute-force login att
empts,
you can enable rate-limiting for the authentication module.
By default, this limits attackers to no more than 3 login attempts every 30s.Do you wa
nt to enable rate-limiting (y/n)
选择是否打开尝试次数限制（防止暴力攻击），自然答 y
```

这里需要记住的是

```
$cat ~/.google_authenticator        手机密钥和应急码保存路径
密钥
Your emergency scratch codes are: 一些生成的 5 个应急码，每个应急码只能使用一次
```

5.2 使用

5.2.1 在安卓设备上运行 Google 认证器

安装 ***google*** 身份验证器

我的方法是在 UC 中搜索的 google 身份验证器进行的安装

输入密钥

选择"Enter provided key"选项，使用键盘输入账户名称和验证密钥

5.2.2 终端使用二次身份验证登陆

windows xshell

打开 xshell（其他终端类似），选择登陆主机的属性。设置登陆方法为 Keyboard Interactive

登陆时输入用户名后，接着输入手机设备上的数字，然后输入密码

linux

linux 下直接输入

```
#ssh 用户名 @IP
```

连接比较慢时可以修改本机的客户端配置文件 ssh_config，注意，不是 sshd_config

GSSAPIAuthentication yes -->no

```
#sed -i 's#GSSAPIAuthentication yes#GSSAPIAuthentication no#' /etc/ssh/ssh_config
```

5.3 常见问题及注意点

5.3.1 登陆失败

如果 SELinux 是打开状态，则会登陆失败，日志 /var/log/secret 中会有如下日志

```
Jan  3 23:42:50 hostname sshd(pam_google_authenticator)[1654]: Failed to update secret
 file "/home/username/.google_authenticator"
Jan  3 23:42:50 hostname  sshd[1652]: error: PAM: Cannot make/remove an entry for the
specified session for username from 192.168.0.5
```

5.3.2 是否可以不同的用户使用不用密钥

可以，只需要在不同的用户执行 `google-authenticator` 即可

5.3.3 是否可以使用 ssh 密钥直接登陆

可以，根据以上方法操作，只限制密码登陆时需要二次认证

5.4 原理

基于时间的一次性密码（Time-based One-time Password，简称 TOTP），只需要在手机上安装密码生成应用程序，就可以生成一个随着时间变化的一次性密码，用于帐户验证，而且这个应用程序不需要连接网络即可工作。仔细看了看这个方案的实现原理，发现挺有意思的。

5.4.1 前世今生

HOTP

Google 的两步验证算法源自另一种名为 HMAC-Based One-Time Password 的算法，简称 HOTP。HOTP 的工作原理如下：

客户端和服务器事先协商好一个密钥 K，用于一次性密码的生成过程，此密钥不被任何第三方所知道。此外，客户端和服务器各有一个计数器 C，并且事先将计数值同步。 进行验证时，客户端对密钥和计数器的组合 (K,C) 使用 HMAC（Hash-based Message Authentication Code）算法计算一次性密码，公式如下：

> HOTP(K,C) = Truncate(HMAC-SHA-1(K,C))

上面采用了 HMAC-SHA-1，当然也可以使用 HMAC-MD5 等。HMAC 算法得出的值位数比较多，不方便用户输入，因此需要截断（Truncate）成为一组不太长十进制数（例如 6 位）。计算完成之后客户端计数器 C 计数值加 1。用户将这一组十进制数输入并且提交之后，服务器端同样的计算，并且与用户提交的数值比较，如果相同，则验证通过，服务器端将计数值 C 增加 1。如果不相同，则验证失败。

这里的一个比较有趣的问题是，如果验证失败或者客户端不小心多进行了一次生成密码操作，那么服务器和客户端之间的计数器 C 将不再同步，因此需要有一个重新同步（Resynchronization）的机制。

TOTP

介绍完了 HOTP，Time-based One-time Password（TOTP）也就容易理解了。TOTP 将 HOTP 中的计数器 C 用当前时间 T 来替代，于是就得到了随着时间变化的一次性密码。非常有趣吧！

一句话概括就是

> 海上升明月，天涯共此时！

5.4.2 TOTP 中的特殊问题

时间 ***T*** 的选取 ***(30*** 秒作为时间片）

首先，时间 T 的值怎么选取？因为时间每时每刻都在变化，如果选择一个变化太快的 T（例如从某一时间点开始的秒数），那么用户来不及输入密码。如果选择一个变化太慢的 T（例如从某一时间点开始的小时数），那么第三方攻击者就有充足的时间去尝试所有可能的一次性密码（试想 6 位数字的一次性密码仅仅有 10^6 种组合），降低了密码的安全性。除此之外，变化太慢的 T 还会导致另一个问题。如果用户需要在短时间内两次登录账户，由于密码是一次性的不可重用，用户必须等到下一个一次性密码被生成时才能登录，这意味着最多需要等待 59 分 59 秒！这显然不可接受。综合以上考虑，

Google 选择了 30 秒作为时间片，T 的数值为从 Unix epoch（1970 年 1 月 1 日 00:00:00）来经历的 30 秒的个数。

网络延时处理

第二个问题是，由于网络延时，用户输入延迟等因素，可能当服务器端接收到一次性密码时，T 的数值已经改变，这样就会导致服务器计算的一次性密码值与用户输入的不同，验证失败。解决这个问题个一个方法是，服务器计算当前时间片以及前面的 n 个时间片内的一次性密码值，只要其中有一个与用户输入的密码相同，则验证通过。当然，n 不能太大，否则会降低安全性。 事实上，这个方法还有一个另外的功能。我们知道如果客户端和服务器的时钟有偏差，会造成与上面类似的问题，也就是客户端生成的密码和服务端生成的密码不一致。但是，如果服务器通过计算前 n 个时间片的密码并且成功验证之后，服务器就知道了客户端的时钟偏差。因此，下一次验证时，服务器就可以直接将偏差考虑在内进行计算，而不需要进行 n 次计算。

以上就是 Google 两步验证的工作原理，推荐大家使用，这确实是保护帐户安全的利器。

6 iptables 命令

6.1 iptables 是什么

iptables 是与 Linux 内核集成的 IP 信息包过滤系统，该系统有利于在 Linux 系统上更好地控制 IP 信息包过滤和防火墙配置。

6.2 iptables 示例

6.2.1 filter 表 INPUT 链

怎么处理发往本机的包。

```
# iptables {-A|-D|-I} INPUT rule-specification
# iptables -A INPUT -s 10.1.2.11 -p tcp --dport 80 -j DROP
# iptables -A INPUT -s 10.1.2.11 -p tcp --dport 80 -j REJECT --reject-with tcp-reset
# iptables -A INPUT -s 10.1.2.11 -p tcp --dport 80 -j ACCEPT
```

以上表示将从源地址 10.1.2.11 访问本机 80 端口的包丢弃（以 tcp-reset 方式拒绝和接受）。

> - -s 表示源地址（--src,--source），其后面可以是一个 IP 地址（10.1.2.11）、一个 IP 网段（10.0.0.0/8）、几个 IP 或网段（192.168.1.11/32,10.0.0.0/8，添加完成之后其实是两条规则）。
> - -d 表示目的地址（--dst,--destination），其后面可以是一个 IP 地址（10.1.2.11）、一个 IP 网段（10.0.0.0/8）、几个 IP 或网段（10.1.2.11/32,10.1.3.0/24，添加完成之后其实是两条规则）。
> - -p 表示协议类型（--protocol），后面可以是 tcp, udp, udplite, icmp, esp, ah, sctp, all，其中 all 表示所有的协议。
> - --sport 表示源端口（--source-port），后面可以是一个端口（80）、一系列端口（80:90，从 80 到 90 之间的所有端口），一般在 OUTPUT 链使用。
> - --dport 表示目的端口（--destination-port），后面可以是一个端口（80）、一系列端口（80:90，从 80 到 90 之间的所有端口）。
> - -j 表示 iptables 规则的目标（--jump），即一个符合目标的数据包来了之后怎么去处理它。常用的有 ACCEPT, DROP, REJECT, REDIRECT, LOG, DNAT, SNAT。
>   - (“就好象骗子给你打电话，ACCEPT 是接收，drop 就是直接拒收，reject 的话，相当于你还给骗子回个电话。”)

```
# iptables -A INPUT -p tcp --dport 80 -j DROP
# iptables -A INPUT -p tcp --dport 80:90 -j DROP
# iptables -A INPUT -m multiport -p tcp --dports 80,8080 -j DROP
```

以上表示将所有访问本机 80 端口（80 和 90 之间的所有端口，80 和 8080 端口）的包丢弃。

> - -m 匹配更多规则（--match），可以指定更多的 iptables 匹配扩展。可以是 tcp, udp, multiport, cpu, time, ttl 等，即你可以指定一个或多个端口，或者本机的一个 CPU 核心，或者某个时间段内的包。

6.2.2 filter 表 OUTPUT 链

怎么处理本机向外发的包。

```
# iptables -A OUTPUT -p tcp --sport 80 -j DROP
```

以上这条规则意思是不允许访问本机 80 端口的包出去。即你可以向本机 80 端口发送请求包，但是本机回应给你的包会被该条规则丢弃。

INPUT 链与 OUTPUT 链用法一样，但是表示的意思不同。

6.2.3 filter 表的 FORWARD 链

For packets being routed through the box（不知道怎么解释）。

其用法与 INPUT 链和 OUTPUT 链类似。

6.3 nat 表

nat 表有三条链，分别是 PREROUTING, OUTPUT, POSTROUTING。

6.3.1 nat 表 PREROUTING 链

修改发往本机的包。

```
# iptables -t nat -A PREROUTING -p tcp -d 202.102.152.23 --dport 80 -j DNAT --to-desti
nation 10.67.15.23:8080
# iptables -t nat -A PREROUTING -p tcp -d 202.102.152.23 -j DNAT --to-destination 10.6
7.15.23
```

以上这两条规则的意思是将发往 IP 地址 202.102.152.23 和端口 80 的包的目的地址修改为 10.67.15.23，目的端口修改为 8080。将发往 202.102.152.23 的其他非 80 端口的包目的地址修改为 10.67.15.23。第二条规则中的 -p tcp 是可选的，也可以指定其他协议。

其实类似这样的规则一般在路由器上做，路由器上有个公网 IP（202.102.152.23），其中有个用户的内网 IP（10.67.15.23）想提供外网的 web 服务，而路由器又不想将公网 IP 地址绑定到用户机器上，因此就出来了以上的蛋疼规则。

6.3.2 nat 表 POSTROUTING 链

修改本机向外发的包。

```
# iptables -t nat -A POSTROUTING -p tcp -s 10.67.15.23 --sport 8080 -j SNAT --to-sourc
e 202.102.152.23:80
# iptables -t nat -A POSTROUTING -p tcp -s 10.67.15.23 -j SNAT --to-source 202.102.152
.23
```

以上两条规则的意思是将从 IP 地址 10.67.15.23 和端口 8080 发出的包的源地址修改为 202.102.152.23，源端口修改为 80。将从 10.67.15.23 发出的非 80 端口的包的源地址修改为 202.102.152.23。

这两条正好与以上两条 PREROUTING 共同完成了内网用户想提供外网服务的功能。

其中的 --to-destination 和 --to-source 都可以缩写成 --to，在 DNAT 和 SNAT 中会分别被理解成 --to-destination 和 --to-source。

注：之所以将内网服务的端口和外网服务的端口写的不一致是因为二者其实真的可以不一致。另外，是否将 PREROUTNG 中的 -d 改为域名就可以使用一个公网 IP 为不同用户提供服务了呢？这个需要哥哥我稍后验证。

6.3.3 nat 表做 HA 的实例

有两台服务器和三个 IP 地址，分别是 10.1.2.21, 10.1.2.22, 10.1.5.11。假设他们提供的是相同的 WEB 服务，现在想让他们做 HA，而 10.1.5.11 是他们的 VIP。

- 10.1.2.21 这台的 NAT 规则如下：

```
# iptables -t nat -A PREROUTING -p tcp -d 10.1.2.11 --dport 80 -j DNAT --to-destinatio
n 10.1.2.21:80
# iptables -t nat -A POSTROUTING -p tcp -s 10.1.2.21 --sport 80 -j SNAT --to-source 10
.1.2.11:80
```

- 10.1.2.22 这台的 NAT 规则如下：

```
# iptables -t nat -A PREROUTING -p tcp -d 10.1.2.11 --dport 80 -j DNAT --to-destinatio
n 10.1.2.22:80
# iptables -t nat -A POSTROUTING -p tcp -s 10.1.2.22 --sport 80 -j SNAT --to-source 10
.1.2.11:80
```

默认可以认为 VIP 在 10.1.2.21 上挂着，那么当这台机器发生故障不能提供服务时，我们可以及时将 VIP 挂到 10.1.2.22 上，这样就可以保证服务不中断了。当然我们可以写一个简单的 SHELL 脚本来完成 VIP 的检测及挂载，方法非常简单。

注：LVS 的实现中貌似有这么一项，还没有深入去研究 LVS。

6.3.4 nat 表为虚拟机做内外网联通

宿主机内网 IP 是 10.67.15.183(eth1)，外网 IP 是 202.102.152.183(eth0)，内网网关是 10.67.15.1，其上面的虚拟机 IP 是 10.67.15.250(eth1)。

目前虚拟机只能连接内网，其路由信息如下：

```
# ip r s
10.67.15.0/24 dev eth1  proto kernel  scope link  src 10.67.15.250
169.254.0.0/16 dev eth1  scope link  metric 1003
192.168.0.0/16 via 10.67.15.1 dev eth1
172.16.0.0/12 via 10.67.15.1 dev eth1
10.0.0.0/8 via 10.67.15.1 dev eth1
default via 10.67.15.1 dev eth1
```

若要以 NAT 方式实现该虚拟机即能连接公网又能连接内网，则该虚拟机路由需要改成以下：

```
# ip r s
10.67.15.0/24 dev eth1  proto kernel  scope link  src 10.67.15.250
169.254.0.0/16 dev eth1  scope link  metric 1003
192.168.0.0/16 via 10.67.15.1 dev eth1
172.16.0.0/12 via 10.67.15.1 dev eth1
10.0.0.0/8 via 10.67.15.1 dev eth1
default via 10.67.15.183 dev eth1
```

虚拟机连接内网的网关地址也可以写成宿主机内网 IP 地址。

宿主机上面添加如下 NAT 规则：

```
# iptables -t nat -A POSTROUTING -s 10.67.15.250/32 -d 10.0.0.0/8 -j SNAT --to-source
10.67.15.250
# iptables -t nat -A POSTROUTING -s 10.67.15.250/32 -d 172.16.0.0/12 -j SNAT --to-sour
ce 10.67.15.250
# iptables -t nat -A POSTROUTING -s 10.67.15.250/32 -d 192.168.0.0/16 -j SNAT --to-sou
rce 10.67.15.250
# iptables -t nat -A POSTROUTING -s 10.67.15.250/32 -j SNAT --to-source 202.102.152.18
3
```

以上四条规则的意思是将从源地址 10.67.15.250 发往内网机器上的数据包的源地址改为 10.67.15.250。将从源地址 10.67.15.250 发往公网机器上的数据包的源地址修改为 202.102.152.183。

6.4 iptables 管理命令

6.4.1 查看 iptables 规则

```
# iptables -nL
# iptables -n -L
# iptables --numeric --list
# iptables -S
# iptables --list-rules
# iptables -t nat -nL
# iptables-save
```

> - -n 代表 --numeric，意思是 IP 和端口都以数字形式打印出来。否则会将 127.0.0.1:80 输出成 localhost:http。端口与服务的对应关系可以在 /etc/services 中查看。
> - -L 代表 --list，列出 iptables 规则，默认列出 filter 链中的规则，可以用 -t 来指定列出哪个表中的规则。
> - -t 代表 --tables，指定一个表。
> - -S 代表 --list-rules，以原命令格式列出规则。

iptables-save 命令是以原命令格式列出所有规则，可以 -t 指定某个表。

6.4.2 清除 iptables 规则

```
# iptables -F
# iptables --flush
# iptables -F OUTPUT
# iptables -t nat -F
# iptables -t nat -F PREROUTING
```

> - -F 代表 --flush，清除规则，其后面可以跟着链名，默认是将指定表里所有的链规则都清除。
>   - （警告：如果已经配置过默认规则为 deny 的环境，即 iptables -P INPUT DROP ，直接命令行执行 iptables -F 将使系统的所有网络访问中断，此坑已踩过）

6.4.3 保存 iptables 规则

```
# /etc/init.d/iptables save
```

该命令会将 iptables 规则保存到 /etc/sysconfig/iptables 文件里面，如果 iptable 有开机启动的话，开机时会自动将这些规则添加到机器上。

6.5 常用操作

iptables 命令中的很多选项前面都可以加"!"，意思是“非”。如"! -s 10.0.0.0/8"表示除这个网段以外的源地址，"! --dport 80"表示除 80 以外的其他端口。

6.5.1 使用 ip6tables 禁用 ipv6

目前 ipv6 不禁用会存在安全隐患，那我们就可以通过 ip6tables 禁用 ipv6，我们只要在 ip6tables 的 filter 表上的出入口以及转发做限定就行了。

```
[root@meetbill ~]# vim /etc/sysconfig/ip6tables
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]

# 添加这 3 条规则

-A INPUT -j REJECT --reject-with icmp6-adm-prohibited
-A FORWARD -j REJECT --reject-with icmp6-adm-prohibited
-A OUTPUT -j REJECT --reject-with icmp6-adm-prohibited
COMMIT
[root@meetbill ~]# /etc/init.d/ip6tables restart
```

可以通过 ifconfig 查看 ipv6 的地址

```
[root@meetbill ~]# ping6 -I eth0 fe80::20c:29ff:febc:8aab
```

6.5.2 配置 iptables 允许部分端口同行，其他全部阻止

将下列内容放到脚本中，然后执行脚本即可，此脚本可以重复执行

```
#!/bin/bash
iptables -F /* 清除所有规则 */
iptables -A INPUT -p tcp --dport 22 -j ACCEPT /*允许包从 22 端口进入*/
iptables -A OUTPUT -p tcp --sport 22 -m state --state ESTABLISHED -j ACCEPT /*允许从 22
 端口进入的包返回*/
iptables -A INPUT -s 127.0.0.1 -d 127.0.0.1 -j ACCEPT /*允许本机访问本机*/
iptables -A OUTPUT -s 127.0.0.1 -d 127.0.0.1 -j ACCEPT
iptables -A INPUT -p tcp -s 0/0 --dport 80 -j ACCEPT /*允许所有 IP 访问 80 端口*/
iptables -A OUTPUT -p tcp --sport 80 -m state --state ESTABLISHED -j ACCEPT
iptables -P INPUT DROP
iptables -P FORWARD DROP
iptables -P OUTPUT DROP
#iptables-save > /etc/sysconfig/iptables /*保存配置*/
iptables -L /* 显示 iptables 列表 */
```

（警告：如果已经配置过默认规则为 deny 的环境，即 iptables -P INPUT DROP ，直接命令行执行 iptables -F 将使系统的所有网络访问中断，此坑已踩过）

如何清除配置尼（-P 为默认规则）

```
#!/bin/bash
iptables -P INPUT ACCEPT
iptables -P FORWARD ACCEPT
iptables -P OUTPUT ACCEPT
iptables -F
#iptables-save > /etc/sysconfig/iptables
iptables -L
```

Linux 优化

说明

系统优化是一项复杂、繁琐、长期的工作，优化前需要监测、采集、测试、评估，优化后也需要测试、采集、评估、监测，而且是一个长期和持续的过程，不是说现在又花了、测试了，以后就可以一劳永逸，而不是说书本上的优化就适合眼下正在运行的系统，不同的系统、不同的硬件、不用的应用优化的重点也不同、优化的方法也不同、优化的参数也不同。

性能监测是系统优化过程中重要的一环，如果没有监测、不清楚性能瓶颈在哪里，怎么优化呢？所以 `找到性能瓶颈` 是性能监测的目的，也是系统优化的关键。

系统由若干子系统构成，通常修改一个子系统有可能影响到另外一个子系统，甚至会导致整个系统不稳定、崩溃。

所以说优化、监测、测试通常是连在一起的，而且是一个循环而且长期的过程，通常监测的子系统有以下这些：

- CPU
- Memory
- IO
- Network

这些子系统互相依赖，了解这些子系统的特性，监测这些子系统的系能参数以及及时发现可能会出现的瓶颈对系统优化很有帮助

应用类型

不同的系统用途也不同，要找到性能瓶颈需要知道系统跑的是什么应用、有些什么特点，比如 web server 对系统的要求肯定和 file server 不一样，所以分清不同系统的应用类型很重要。

通常应用可以分为两种类型：

- IO 相关
  - IO 相关的应用通常用来出来大量的数据，需要大量内存和存储，频繁 IO 操作读写数据
  - 而对 CPU 的要求则较少，大部分时间 CPU 都在等待硬盘，比如，数据库服务器、文件服务器等
- CPU 相关
  - CPU 相关的应用需要使用大量 CPU
  - 比如高并发的 web/mail 服务器、图像 / 视频处理、科学计算等都可视作 CPU 相关的应用

监测工具

我们只需要简单的工具就可以对 Linux 的性能进行监控，以下常用的工具：

工具	简介
top	查看进程活动状态以及一些系统状况
vmstat	查看系统状态、硬件和系统信息等
iostat	查看 CPU 负载、硬盘状况
sar	综合工具，查看系统状况
mpstat	查看多处理器状况
netstat	查看网络状况
iptraf	实时网络状态监测
tcpdump	抓取网络数据包，详细分析
tcptrace	网络包分析工具
netperf	网络带宽工具
dstat	综合了 vmstat、iostat、ifstat、netstat 等多个信息

本系列将按照 CPU、内存、磁盘 IO、网络这几个方面分别介绍

Linux 性能监测 CPU 篇

CPU 的占用主要取决于什么样的资源在 CPU 上面运行，比如拷贝一个文件通常占用较少的 CPU，因为大部分工作是由 DMA（Direct Memory Access）完成，只是在完成拷贝以后给一个中断让 CPU 知道拷贝已经完成；科学计算通常占用较多的 CPU，大部分计算工作都需要在 CPU 上完成，内存、硬盘等子系统只是做暂时的数据存储工作。

要想监测和理解 CPU 的性能需要知道一些的操作系统基本知识，比如：中断、进程调度、进程上下文切换、可运行队列等。

用一个例子来简单介绍一下这些概念和他们的关系，CPU 很无辜，是个任劳任怨的打工仔，每时每刻都有工作在做（进程、线程）并且自己有一张工作清单（可运行队列），由老板（进程调度）来决定他该干什么，他需要和老板沟通以便得到老板的想法并及时调整自己的工作（上下文切换），部分工作做完以后还需要及时向老板汇报（中断），所以打工仔（CPU）除了做自己该做的工作之外，还有大量时间和精力花在沟通和汇报上。

CPU 也是一种硬件资源，和任何其他设备一样也需要驱动和管理程序才能使用，我们可以把内核的进程调度看作是 CPU 的管理程序，用来管理和分配 CPU 资源，合理安排进程抢占 CPU，并决定哪个进程该使用 CPU、哪个进程该等待。

操作系统内核里的进程调度主要用来调度两类资源：进程（或线程）和中断，进程调度给不同的资源分配了不同的优先级，优先级最高的是硬件中断，其次是内核（系统）进程，最后是用户进程。

每个 CPU 都维护这一个可运行队列，用来存放那些可运行的线程。线程要么在睡眠状态（blocked 正在等待 IO）、要么在可运行状态，如果 CPU 当前负载太高而新的请求不断，就会出现进程调度暂时应付不过来的情况，这个时候就不得不把线程暂时放到可运行队列中。

本文是讨论的性能监测，上面淡了一堆都没提到性能，那么这些概念和性能监测有什么关系呢？关系重大！如果你是老板，你如何检查打工仔的效率（性能）呢？我们一般会通过以下这些信息来判断打工仔是否偷懒：

- 打工仔接受和完成多少任务并向老板汇报了（中断）
- 打工仔和老板沟通、写上每项工作的工作进度（上下文切换）
- 打工仔的工作列表是不是都有排满（可运行队列）
- 打工仔工作效率如何，是不是在偷懒（CPU 利用率）

现在把打工仔换成 CPU，我们可以通过查看这些重要参数：中断、上下文切换、可运行队列、**CPU** 利用率来检测 CPU 的性能。

底线

Linux 性能监测：介绍提到了性能监测需要知道底线，那么监测 CPU 性能的底线是什么呢？

通常我们期望我们的系统能达到以下目标：

- **CPU** 利用率，如果 CPU 用 100% 的利用率，那么应该达到这样一个平衡：65%-70% User Time，30%-35% System Time，0%-5% Idle Time
- 上下文切换，上下文切换应该和 CPU 利用率联系起来看，如果能保持上面的 CPU 利用率平衡，大量的上下文切换是可以接受的
- 可运行队列，每个可运行队列不应该由超过 1-3 个线程（每处理器），比如：双处理器系统的可运行队列里不应该超过 6 个线程

vmstat 命令

vmstat 是个查看系统整体性能的小工具，小巧，即使在很 heavy 的情况下也允许良好，并且可以用时间间隔采集得到连续的性能数据。

参数介绍：

- r，可运行队列的线程数，这些线程都是可运行状态，只不过 CPU 暂时不可用
- b，被 blocked 的进程数，正在等待 IO 请求
- in，被处理过的中断数
- cs，系统上正在做上下文切换的数目
- us，用户占用 CPU 的百分比
- sys，内核和中断占用 CPU 的百分比
- wa，所有可运行的线程被 blocked 以后都在等待 IO，这时候 CPU 空闲的百分比
- id，CPU 完全空闲的百分比

举两个现实中的例子来分析一下

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 4  0    140 2915476 341288 3951700  0    0     0     0 1057  523 19 81  0  0  0
 4  0    140 2915724 341296 3951700  0    0     0     0 1048  546 19 81  0  0  0
 4  0    140 2915848 341296 3951700  0    0     0     0 1044  514 18 82  0  0  0
 4  0    140 2915848 341296 3951700  0    0     0    24 1044  564 20 80  0  0  0
 4  0    140 2915848 341296 3951700  0    0     0     0 1060  546 18 82  0  0  0
```

从上面的数据可以看出几点：

1. interrupts(in) 非常高，context switch(cs) 比较低，说明这个 CPU 一直在不停的请求资源
2. user time(us) 一直保持在 80% 以上，而且上下文切换较低 (cs)，说明某个进程可能一直霸占着 CPU
3. run queue(r) 刚好在 4 个

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
14  0    140 2904316 341912 3952308  0    0     0   460 1106 9593 36 64  1  0  0
17  0    140 2903492 341912 3951780  0    0     0     0 1037 9614 35 65  1  0  0
20  0    140 2902016 341912 3952000  0    0     0     0 1046 9739 35 64  1  0  0
17  0    140 2903904 341912 3951888  0    0     0    76 1044 9879 37 63  0  0  0
16  0    140 2904580 341912 3952108  0    0     0     0 1055 9808 34 65  1  0  0
```

从上面的数据可以看出几点：

1. context switch(cs) 比 interrupts(in) 要高的多，说明内核不得不来回切换进程
2. 进一步观察发现 system time(sy) 很高而 user time(us) 很低，而且加上高频度的上下文切换 (cs)，说明正在运行的应用程序调用了大量的系统调用
3. run queue(r) 在 14 个线程以上，按照这个而是机器的硬件配置 (4 核），应该保持在 12 以内

mpstat 命令

mpstat 和 vmstat 类似，不同的是 mpstat 可以输出多个处理器的数据，下面的输出显示 CPU1 和 CPU2 基本上没有派上用场，系统有足够的能力处理更多的任务

```
$ mpstat -P ALL 1
Linux 2.6.18-164.el5 (vpsee) 11/13/2009

02:24:33 PM  CPU   %user   %nice    %sys %iowait    %irq   %soft  %steal   %idle    in
tr/s
02:24:34 PM  all    5.26    0.00    4.01   25.06    0.00    0.00    0.00   65.66   144
6.00
02:24:34 PM    0    7.00    0.00    8.00    0.00    0.00    0.00    0.00   85.00   100
1.00
02:24:34 PM    1   13.00    0.00    8.00    0.00    0.00    0.00    0.00   79.00    44
4.00
02:24:34 PM    2    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00
0.00
02:24:34 PM    3    0.99    0.00    0.99    0.00    0.00    0.00    0.00   98.02
0.00
```

ps 命令

如何查看某个程序、进程占用了多少 CPU 资源呢？下面是 java 在一台 Linux 服务器上的运行情况，当前只有 2 个 java 进程

```
$ while :; do ps -eo pid,ni,pri,pcpu,psr,comm | grep 'java'; sleep 1; done
 PID  NI PRI %CPU PSR COMMAND
 7252   0  24  3.2   3 java
 9846   0  24  8.8   0 java
 7252   0  24  3.2   2 java
 9846   0  24  8.8   0 java
 7252   0  24  3.2   2 java
```

Linux 性能监测 内存篇

这里讲到的 `内存` 包括物理内存和虚拟内存。虚拟内存 (Virtual Memory) 把计算机的内存空间扩展到硬盘，物理内存 (RAM) 和硬盘的一部分空间 (SWAP) 组合在一起作为虚拟内存为计算机提供了一个连续的虚拟内存空间，好处是我们拥有的内存 `变多了` ，可以运行更多、更大的程序，坏处是把部分硬盘当内存用，整体性能受到影响，硬盘读写速度要比内存慢几个数量级，并且 RAM 和 SWAP 之间的交换增加了系统的负担。

在操作系统里，虚拟内存被分为页，在 x86 系统上每个页大小是 4KB。Linux 内核读写虚拟内存是以“页”为单位操作的，把内存转移到硬盘交换空间 (SWAP) 和从交换空间读取内存的时候都是按页来读写的。

内存和 SWAP 的这种交互过程称为页面交换 (Paging)，值得注意的是 paging 和 swapping 是两个完全不同的概念，国内很多参考书把这两个概念混为一谈，swapping 也翻译为交换，在操作系统里是指把某程序完全交换到硬盘以腾出内存给新程序使用，和 paging 只交换程序的

部分（页面）是两个不同的概念。春吹的 swapping 在现代操作系统中已经很难看到了，因为把整个程序交换到硬盘的办法既耗时又费力而且没必要，现代操作系统基本都是 paging 或者 paging/swapping 混合，swapping 最初是在 Unix system V 上实现的。

虚拟内存管理是 Linux 内核里面最复杂的部分，要弄懂这部分内容可能需要一本书的讲解。这里只介绍和性能监测有关的两个内核进程：kswapd 和 pdflush。

kswapd daemon用来检查 pages_high 和 pages_low，如果可用内存少于 pages_low，kswapd 就开始扫描并试图释放 32 个页面，并且重复扫描释放的过程知道可用内存大于 pages_high 为止。扫描的时候检查 3 件事：

- 如果页面没有修改，把页放到可用内存列表里
- 如果页面被文件系统修改，把页面内容写到磁盘上
- 如果页面被修改了，但不是被文件系统修改的，把页面写到交换空间

pdflush daemon用来同步文件相关的内存页面，把内存页面及时同步到硬盘上。比如打开一个文件，文件被导入到内存里，对文件修改并保存后，内核并不马上保存文件到硬盘，由 pdfush 决定什么时候把相应页面写到硬盘，这由一个内核参数 vm.dirty_background_ratio 来控制，比如下面的参数显示脏页面（dirty pages）达到所有内存页面 10% 的时候开始写入硬盘。

```
# /sbin/sysctl -n vm.dirty_background_ratio
10
```

vmstat 命令

继续 vmstat 一些参数的介绍，上一篇 Linux 性能监测：CPU 介绍了 vmstat 的部分参数，这里介绍另外一部分。以下数据来自一个 256MB RAM，512MB SWAP 的 Xen VPS：

```
# vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  3 252696   2432    268   7148 3604 2368  3608  2372  288  288  0  0 21 78  1
 0  2 253484   2216    228   7104 5368 2976  5372  3036  930  519  0  0  0 100  0
 0  1 259252   2616    128   6148 19784 18712 19784 18712 3821 1853  0  1  3 95  1
 1  2 260008   2188    144   6824 11824 2584 12664  2584 1347 1174 14  0  0 86  0
 2  1 262140   2964    128   5852 24912 17304 24952 17304 4737 2341 86 10  0  0  4
```

memory

- swpd，已使用的 SWAP 控件大小，KB 为单位
- free，可用的物理内存大小，KB 为单位
- buff，物理内存用来缓存读写操作的 buffer 大小，KB 为单位
- cache，物理内存用来缓存进程地址空间的 cache 大小，KB 为单位

swap

- si，数据从 SWAP 读取到 RAM（swap in）的大小，KB 为单位
- so，数据从 RAM 写到 SWAP（swap in）的大小，KB 为单位

io

- bi，磁盘块从文件系统或 SWAP 读取到 RAM（blocks in）的大小，block 为单位
- bo，磁盘块从 RAM 写到文件系统或 SWAP（blocks out）的大小，block 为单位

上面是一个频繁读写交换区的例子，可以观察到以下几点：

- 物理可用内存 free 基本没有显著变化，swapd 逐步增加，说明最小可用的内存使用保持在 256MB X 10% = 2.56MB 左右，当脏数据达到 10% 的时候 (vm.dirty_background_ratio = 10) 就开始大量使用 swap
- buff 稳步减少说明系统知道内存不够用了，kwapd 正在从 buff 那里借用部分内存
- kswapd 持续把脏数据写到 swap 交换区 (so)，并且从 swapd 主键增加看出确实如此。根据上面将的 kswapd 扫描时检查的三件事，如果页面被修改了，但不是被文件系统修改的，把页面写到 swap，所以这里 swapd 持续增加

Linux 性能监测 磁盘 IO 篇

磁盘通常是计算机最慢的子系统，也是最容易出现性能瓶颈的地方，因为磁盘离 CPU 最远而且 CPU 访问磁盘涉及到机械操作，比如转轴、寻轨等，访问硬盘和访问内存之间的速度差别是以数量级来计算的，就像 1 天和 1 分钟的差别一样，要监测 IO 性能，有必要了解一下基本原理和 Linux 是如何处理硬盘和内存之间的 IO 的。

内存页

Memory 介绍中提到了内存和硬盘之间的 IO 是以页为单位来进行的，在 Linux 系统上 1 页的大小为 4K。可以用下面命令查看系统默认的页面大小：

```
$getconf  PAGESIZE
...
4096
...
```

缺页中断

Linux 利用虚拟内存极大的扩展了程序地址空间，是的原来物理内存不能容下的程序也可以通过内存和硬盘之间的不断交换（把暂时不用的内存页交换到硬盘，把需要的内存页从硬盘读到内存）来赢得更多的内存，看起来就像物理内存被扩大一样。

事实上这个过程对程序是完全透明的，程序完全不用理会自己哪一部分、什么时候被交换到内存，一切都在内核的虚拟内存管理来完成。

当程序启动的时候，Linux 内核首先检查 CPU 的缓存和物理内存，如果数据已经在内存里就忽略，如果数据不再内存里就引起一个缺页中断（**Page Fault**），然后从硬盘读取缺页，并把缺页缓存到物理内存中。

缺页中断可分为主缺页中断（Major Page Fault）和次缺页中断（Minor Page Fault），要从磁盘读取数据而产生的中断是主缺页中断；数据已经读到内存并被缓存起来，从内存缓存区中而不是直接从硬盘中读取数据而产生的中断是次缺页中断。

上面的内存缓存区起到了预读硬盘的作用，内核现在物理内存里寻找缺页，没有的话产生次缺页中断从内存缓存中找，如果还没有发现的话就从硬盘读取。很显然，把多于的内存拿出来做成内存缓存区有助于提高访问速度。

这里还有一个命中率的问题，运气好的话如果每次缺页都能从内存缓存区读取的话将会极大提升性能。要提升命中率的一个简单的方法就是增大内存缓存区面积，缓存区越大预存的页面就越多，命中率也会越多。

下面的 time 命令可以用来查看某程序第一次启动的时候产生了多少主缺页中断和次缺页中断：

```
$ /usr/bin/time -v date
...
Major (requiring I/O) page faults: 1
Minor (reclaiming a frame) page faults: 260
...
```

File Buffer Cache

从上面的内存缓存区（也叫文件缓存区 File Buffer Cache) 读取页比从硬盘读取页要快的多，所以 Linux 内核希望能尽可能产生次却也中断（从文件缓存区读），并且能尽可能避免主缺页中断（从硬盘读），这样随着次缺页中断的增多，文件缓存区也逐步增大，直到系统只有少量可用物理内存的时候 Linux 才开始释放不用的页。

我们运行 Linux 一段时间后会发现虽然系统上运行的程序不多，但是可用内存总是很少，这样给大家造成了 Linux 对内存管理很低效的假象，事实上 Linux 把哪些暂时不用的物理内存高效的利用起来做预存（内存缓存区）呢。下面打印的是一台 Sun 服务器上的物理内存和文件缓存区的情况：

```
$ cat /proc/meminfo
MemTotal:      8182776 kB
MemFree:       3053808 kB
Buffers:        342704 kB
Cached:        3972748 kB
```

这台服务器总共有 8GB 物理内存（MemTotal），3GB 左右可用内存（MemFree），343MB 左右用来做磁盘缓存（Buffers），4GB 左右用来做文件缓存区（Cached），可见 Linux 真的用了很多物理内存做 Cache，而且这个缓存区还可以不断增长。

页面类型

Linux 中内存页面有三种类型：

- Read Pages，只读页（或代码页），那些通过主缺页中断从硬盘读取的页面，包括不能修改的静态文件、可执行文件、库文件等。当内核需要它们的时候把它们读到内存中，当内存不足的时候，内核就释放它们到空闲列表，当程序再次需要它们的时候需要通过缺页中断再次读到内存
- Dirty Pages，脏页，指那些在内存中被修改过的数据页，比如文本文件等。这些文件有 pdflush 负责同步到硬盘，内存不足的时候由 kswapd 和 pdflush 把数据写回硬盘并释放内存
- Anonymous Pages，匿名页，那些属于某个进程但是又和任何文件无关联，不能被同步到硬盘上，内存不足的时候有 kswapd 负责将它们写到交换分区并释放内存

IO's Per Seconds(OIPS)

每次磁盘 IO 请求都需要一定的时间，和访问内存比起来这个等待时间简直难以忍受。

在一台 2001 年典型 1GHz PC 上，磁盘随机访问一个 word 需要 8000000 nanosec = 8 millisec，顺序访问一个 word 需要 200nanosec；而从内存访问一个 word 只需要 10 nanoses。（数据来自：Teach Yourself Programming in Ten Years）这个硬盘可以提供 125 次 IOPS（1000 ms / 8 ms）。

IOPS：每秒 IO 的次数。

顺序 IO 和随机 IO

IO 分为顺序 IO 和随机 IO 两种，性能监测前需要弄清楚系统偏向顺序 IO 的应用还是随机 IO 的应用。

随机 IO 是指同时顺序请求大量数据，比如数据库执行大量的查询、流媒体服务等，顺序 IO 可以同时很快的移动大量数据。可以这样来评估 IOPS 的性能，用每秒读写 IO 字节数除以每秒读写 IOPS 数，rkB/s 除以 r/s，wkB/s 除以 w/s。下面显示的是连续两秒的 IO 情况，可见

每次 IO 写的数据是增加的（45060.00 / 99.00 = 455.15 KB per IO，54272.00 / 112.00 = 484.57 KB per IO）。

相对随机 IO 而言，顺序 IO 更应该重视每次 IO 的吞吐能力（KB per IO）：

```
$ iostat -kx 1
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.00    0.00    2.50   25.25    0.00   72.25

Device:  rrqm/s   wrqm/s   r/s   w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await  svct
m  %util
sdb       24.00 19995.00 29.00 99.00  4228.00 45060.00   770.12    45.01  539.65   7.8
0  99.80

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.00    0.00    1.00   30.67    0.00   68.33

Device:  rrqm/s   wrqm/s   r/s   w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await  svct
m  %util
sdb        3.00 12235.00  3.00 112.00   768.00 54272.00   957.22   144.85  576.44   8.
70 100.10
```

随机 IO 是指随机请求数据，其 IO 速度不依赖于数据的大小和排序，依赖于磁盘的每秒能 IO 的次数，比如 Web 服务、Mial 服务等每次请求的数据都很小，随机 IO 每次同时会有更多的请求数产生，所以磁盘的每秒能 IO 多少次是关键

```
$ iostat -kx 1
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.75    0.00    0.75    0.25    0.00   97.26

Device:  rrqm/s   wrqm/s   r/s   w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await  svct
m  %util
sdb        0.00    52.00  0.00 57.00     0.00   436.00    15.30     0.03    0.54   0.2
3   1.30

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.75    0.00    0.75    0.25    0.00   97.24

Device:  rrqm/s   wrqm/s   r/s   w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await  svct
m  %util
sdb        0.00    56.44  0.00 66.34     0.00   491.09    14.81     0.04    0.54   0.1
9   1.29
```

按照上面的公式得出：436.00 / 57.00 = 7.65KB per IO， 491.09 / 66.34 = 7.40 KB per IO ，与顺序 IO 比较发现，随机 IO 的 KB per IO 小到可以忽略不计，可见对于随机 IO 而言重要的是每秒能 IOPS 的次数，而不是每次 IO 的吞吐能力（KB per IO）

SWAP

当系统没有足够物理内存来应付所有请求的时候就会用到 swap 设备，swap 设备可以是一个文件，也可以是磁盘分区。

不过要小心的是，使用 swap 的代价非常大。如果系统没有物理内存可用，就会频繁 swapping，如果 swap 设备和程序正在访问的数据在同一个文件系统上，那会碰到严重的 IO 问题，最终导致整个系统迟缓，甚至崩溃。

swap 设备和内存之间的 swapping 状况是判断 Linux 系统性能的重要参考，我们已经有很多工具可以用来监测 swap 和 swapping 的情况，比如：top、cat/proc/meminfo、vmstat 等：

```
$ cat /proc/meminfo
MemTotal:      8182776 kB
MemFree:       2125476 kB
Buffers:        347952 kB
Cached:        4892024 kB
SwapCached:        112 kB
...
SwapTotal:     4096564 kB
SwapFree:      4096424 kB
...
```

关掉 swap

```
(1) 将 /etc/fstab 文件中所有设置为 swap 的设备关闭
[root@meetbill ~]# swapoff -a

(2) 设置开机不启动 swap
将 /etc/fstab 中 swap 行注释掉
```

Linux 性能监测 网络篇

网络的监测是所有 Linux 子系统里面最复杂的，有太多的因素在里面，比如：延迟、阻塞、冲突、丢包等，更糟的是与 Linux 主机相连的路由器、交换机、无线信号都会影响到整体网络并且很难判断是因为 Linux 网络子系统的问题还是别的设备的问题，增加了监测和判断的复杂度。

现在我们使用的所有网卡都称为自适应网卡，意思是说能根据网络上的不同网络设备导致的不同网络速度和工作模式进行自动调整。我们可以通过 ethtool 共苦；来查看网卡的配置和工作模式：

```
# /sbin/ethtool eth0
Settings for eth0:
Supported ports: [ TP ]
Supported link modes:   10baseT/Half 10baseT/Full
                        100baseT/Half 100baseT/Full
                        1000baseT/Half 1000baseT/Full
Supports auto-negotiation: Yes
Advertised link modes:  10baseT/Half 10baseT/Full
                        100baseT/Half 100baseT/Full
                        1000baseT/Half 1000baseT/Full
Advertised auto-negotiation: Yes
Speed: 100Mb/s
Duplex: Full
Port: Twisted Pair
PHYAD: 1
Transceiver: internal
Auto-negotiation: on
Supports Wake-on: g
Wake-on: g
Current message level: 0x000000ff (255)
Link detected: yes
```

上面给出的例子说明网卡有 10baseT，100baseT 和 1000baseT 三种选择，目前正在自适应为 100baseT（Speed：100MB/s）。可以通过 ethtool 工具强制网卡工作在 1000basseT 下：

```
# /sbin/ethtool -s eth0 speed 1000 duplex full autoneg off
iptraf
```

两台主机之间有网线（或无线）、路由器、交换机等设备，测试两台主机之间的网络性能的一个办法就是在这两个系统之间互发数据并统计结果，看看吞吐量、延迟、速率如何。

iptraf 就是一个很好的查看本机网络吞吐量的好工具，支持文字图形界面，很直观。下面图片显示在 100mbps 速率的网络下这个 Linux 系统的发送传输率有点慢，Outgoing rates 只有 66mbps：

```
# iptraf -d eth0
```

netperf

netperf 运行在 client/server 模式下，比 iptraf 能更多样化的测试终端的吞吐量。先在服务器端启动 netserver：

```
# netserver
Starting netserver at port 12865
Starting netserver at hostname 0.0.0.0 port 12865 and family AF_UNSPEC
```

然后在客户端测试服务器，执行一次持续 10 秒的 TCP 测试：

```
# netperf -H 172.16.38.36 -l 10
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 172.16.38.36 (172.16.38.36) p
ort 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    10.32      93.68
```

从上面输出可以看出，网络吞吐量在 94mbps 左右，对于 100mbps 的网络来说这个性能算的上很不错。

上面的测试是在服务器和客户端位于同一个局域网，并且局域网是有线网的情况，你也可以试试不同结构、不同速率的网络，比如：网络之间中间多个路由器、客户端在 wi-fi、VPN 等情况。

netperf 还可以通过建立一个 TCP 连接并顺序地发送数据包来测试每秒有多少 TCP 请求和响应。下面的输出显示在 TCP requests 使用 2K 大小，responses 使用 32K 的情况下处理速率为每秒 243：

```
# netperf -t TCP_RR -H 172.16.38.36 -l 10 -- -r 2048,32768
TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 172.16.38.36 (172.1
6.38.36) port 0 AF_INET
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  2048     32768   10.00     243.03
16384  87380
```

同时可以使用 netperf 持续发送数据包，通过 atop 查看网卡流量查看网络状态是否良好

iperf

iperf 和 netperf 运行方式类似，也是 server/client 模式，现在服务器端启动 iperf：

```
# iperf -s -D
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
Running Iperf Server as a daemon
The Iperf daemon process ID : 5695
```

然后在客户端对服务器进行测试，客户端线连接到服务器端（172.16.38.36），并在 30 秒内每隔 5 秒对服务器和客户端之间的网络进行一次带宽测试和采样：

```
# iperf -c 172.16.38.36 -t 30 -i 5
------------------------------------------------------------
Client connecting to 172.16.38.36, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 172.16.39.100 port 49515 connected with 172.16.38.36 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 5.0 sec  58.8 MBytes  98.6 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3]  5.0-10.0 sec  55.0 MBytes  92.3 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 10.0-15.0 sec  55.1 MBytes  92.4 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 15.0-20.0 sec  55.9 MBytes  93.8 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 20.0-25.0 sec  55.4 MBytes  92.9 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 25.0-30.0 sec  55.3 MBytes  92.8 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec    335 MBytes  93.7 Mbits/sec
```

tcpdump 和 tcptrace

tcpdump 和 tcptrace 提供了一种更细致的分析方法，先用 tcpdump 按要求捕获数据包把结果输出到某一文件，然后再用 tcptrace 分析其文件格式。这个工具组合可以提供一些难以用其他工具发现的信息：

```
# /usr/sbin/tcpdump -w network.dmp
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
511942 packets captured
511942 packets received by filter
0 packets dropped by kernel

# tcptrace network.dmp
1 arg remaining, starting with 'network.dmp'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

511677 packets seen, 511487 TCP packets traced
elapsed wallclock time: 0:00:00.510291, 1002714 pkts/sec analyzed
trace file elapsed time: 0:02:35.836372
TCP connection info:
  1: zaber:54581 - boulder:111 (a2b)                  6>    5<  (complete)
  2: zaber:833 - boulder:32774 (c2d)                  6>    5<  (complete)
  3: zaber:pcanywherestat - 172.16.39.5:53086 (e2f)   2>    3<
  4: zaber:716 - boulder:2049 (g2h)                 347>  257<
  5: 172.16.39.100:58029 - zaber:12865 (i2j)          7>    5<  (complete)
  6: 172.16.39.100:47592 - zaber:36814 (k2l)        255380> 255378<  (reset)
  7: breakpoint:45510 - zaber:7012 (m2n)              9>    5<  (complete)
  8: zaber:35813 - boulder:111 (o2p)                  6>    5<  (complete)
  9: zaber:837 - boulder:32774 (q2r)                  6>    5<  (complete)
 10: breakpoint:45511 - zaber:7012 (s2t)              9>    5<  (complete)
 11: zaber:59362 - boulder:111 (u2v)                  6>    5<  (complete)
 12: zaber:841 - boulder:32774 (w2x)                  6>    5<  (complete)
 13: breakpoint:45512 - zaber:7012 (y2z)              9>    5<  (complete)
```

tcptrace 功能很强大，还可以通过过滤和布尔表达式来找出有问题的连接，比如，找出转播大于 100segments 的连接：

```
# tcptrace -f'rexmit_segs>100' network.dmp
```

如果发现连接 #10 有问题，可以查看关于这个连接的其他信息：

```
# tcptrace -o10 network.dmp
```

下面的命令使用 tcptrace 的 slice 模式，程序自动在当前目录创建了一个 slice.dat 文件，这个文件包含了每隔 15 秒的转播信息：

```
# tcptrace -xslice network.dmp

# cat slice.dat
date                segs    bytes  rexsegs rexbytes      new   active
--------------- -------- -------- -------- -------- -------- --------
16:58:50.244708    85055  4513418        0        0        6        6
16:59:05.244708   110921  5882896        0        0        0        2
16:59:20.244708   126107  6697827        0        0        1        3
16:59:35.244708   151719  8043597        0        0        0        2
16:59:50.244708    37296  1980557        0        0        0        3
17:00:05.244708       67     8828        0        0        2        3
17:00:20.244708      149    22053        0        0        1        2
17:00:35.244708       30     4080        0        0        0        1
17:00:50.244708       39     5688        0        0        0        1
17:01:05.244708       67     8828        0        0        2        3
17:01:11.081080       37     4121        0        0        1        3
```

ulimit 关于系统连接数的优化

linux 默认值 open files 和 max user processes 为 1024

#ulimit -n

1024

#ulimit –u

1024

问题描述： 说明 server 只允许同时打开 1024 个文件，处理 1024 个用户进程

使用 ulimit -a 可以查看当前系统的所有限制值，使用 ulimit -n 可以查看当前的最大打开文件数。

新装的 linux 默认只有 1024 ，当作负载较大的服务器时，很容易遇到 error: too many open files 。因此，需要将其改大。

解决方法：

使用 ulimit –n 65535 可即时修改，但重启后就无效了。（注 ulimit -SHn 65535 等效 ulimit -n 65535 ，-S 指 soft ，-H 指 hard)

修改方式

有如下三种修改方式：

1. 在 /etc/rc.local 中增加一行 ulimit -SHn 65535

2. 在 /etc/profile 中增加一行 ulimit -SHn 65535
3. 在 /etc/security/limits.conf 最后增加：

```
* soft nofile 65535
* hard nofile 65535
* soft nproc 65535
* hard nproc 65535
```

具体使用哪种，在 CentOS 中使用第 1 种方式无效果，使用第 3 种方式有效果，而在 Debian 中使用第 2 种有效果

ulimit -n

65535

ulimit -u

65535

备注：ulimit 命令本身就有分软硬设置，加 -H 就是硬，加 -S 就是软默认显示的是软限制

soft 限制指的是当前系统生效的设置值。 hard 限制值可以被普通用户降低。但是不能增加。 soft 限制不能设置的比 hard 限制更高。 只有 root 用户才能够增加 hard 限制值。

Shell 基础及实例

1 shell 编程环境

1.1 编程基础知识

1.1.1 程序编程风格

> - 过程式：以指令为中心，数据服务于指令；
> - 对象式：以数据为中心，指令服务于数据；

shell 程序，提供了编程能力，解释执行，shell 就是一解释器；

1.1.2 程序的执行方式

过程式编程的三种结构；

> - 顺序执行
> - 循环执行
> - 选择执行

1.1.3 shell 脚本

首行特定格式： `#!/bin/bash`

1.1.4 运行脚本的两种方式

> - a、 给予执行权限，通过具体的文件路径指定文件执行；
> - b、 直接运行解释器，将脚本作为解释器程序的参数运行；

Example

```
[root@localhost test1]# vim test.sh
[root@localhost test1]# bash test.sh
hello,girl
[root@localhost test1]# chmod +x test.sh
[root@localhost test1]# ./test.sh
hello,girl
```

1.2 Bash 编程

> - bash 是弱类型编程，变量默认为字符型；
> - 把所有要存储的数据统统当做字符进行存储；
> - 变量不需要事先声明，可以在调用时直接赋值使用，参与运算会自动进行隐式类型转换；
> - 不支持浮点数；

1.3 逻辑运算

- 与：&& 同为 1 则为 1，否则为 0；
- 或：|| 同为 0 则为 0，否则为 1；
- 非：取反，!0 为 1，!1 为 0；
- 短路与运算：双目运算符前面的结果为 0，则结果一定为 0，后面的不执行；
- 短路或运算：双目运算符前面的结果为 1，则结果一定为 1，后面的不执行；

2 bash 变量类型

变量类型决定了变量的数据存储格式、存储空间大小以及变量能参与的运算种类；

2.1 强弱类型语言的区别

- 强类型：定义变量时必须执行类型、参与运算必须符合类型要求；调用未声明变量会产生错误；
- 弱类型：无需指定类型，默认均为字符型；参与运算会自动进行隐式类型转换；变量无需事先定义即可直接调用；

2.2 Bash 中的变量

根据变量的生效范围等标准划分：

- 本地变量：生效范围为当前 shell 进程；对当前 shell 之外的其他 shell 进程，包括当前 shell 的子 shell 进程均无效；
- 环境变量：生效范围为当前 shell 进程及其子进程；
- 局部变量：生效范围为当前 shell 进程中某代码片断（通常指函数）
- 位置变量：`$1, $2, ...` 来表示，用于让脚本在脚本代码中调用通过命令行传递给它的参数；
- 特殊变量：`$?, $0, $*, $@, $#`
  - `$$`：代表所在命令的 PID（不常用）
  - `$!`：代表最后执行的后台命令的 PID（不常用）
  - `$?`：上一条命令的执行状态结果
  - `$0`：命令本身
  - `$*`：传递给脚本的所有参数，以一对双引号给出参数列表
  - `$@`：传递给脚本的所有参数，将各个参数分别加双引号返回
  - `$#`：传递给脚本的参数的个数；

2.2.1 本地变量

变量赋值：name='VALUE'

```
a) 在赋值时，VALUE 可以使用以下引用：

【1】可以是直接字符串；name="username"
【2】变量引用：name=“$username”
【3】命令引用：name=`COMMAND`,name=$(COMMAND)
```

变量引用：`${name}`，花括号可省略：`$name`

引号引用：

> - `" "`：弱引用，其中的变量引用会被替换为变量值；
> - `' '`：强引用，其中的变量不会被替换为变量值，而保持原字符串；

查看所有已定义的变量：#set

销毁变量：# unset name

2.2.2 环境变量

```
变量声明、赋值：
   export name=VALUE
   declare -x name=VALUE
变量引用：
   $name
   ${name}
显示所有环境变量：
   export
   env
   printenv
销毁环境变量：
   unset name
```

Bash 中内建的环境变量：

> - PATH，SHELL，UID，HISTSIZE, HOME, PWD, OLD, HISTFILE, PS1

2.2.3 只读变量

相当于常量，变量值不可变，不能再进行赋值运算；

声明只读变量的格式：

```
readonly name
declare –r name
```

2.2.4 位置变量

在脚本代码中调用通过命令行传递给脚本的参数；

```
$1,$2,…. : 对应调用第 1、第 2 等参数；
$0: 命令本身；
$*: 传递给脚本的所有参数；
$@: 传递给脚本的所有参数；
$#: 传递给脚本的参数的个数；
```

2.3 变量特殊用法

2.3.1 将多行结果赋值给变量

将多行结果赋值给变量时使用变量时需要注意

如：ls_data=$(ls -l)

> - 在 Mac 上直接输出 echo ${ls_data} 即正常结果
> - 在 CentOS 上操作时
>   - echo ${ls_data} 时为不换行内容，整体只有一行
>   - echo "${ls_data}" 时内容输出正常

比如要获取 redis 的 info 信息时，不加双引号输出时，输出的内容仅仅为多行结果的最后一行，加双引号输出时可以正常输出

2.3.2 去除行结果后的特殊字符

如要去掉 `^M`

```
echo ${variable}|tr -d '\r'
```

3 bash 的配置文件

> - 全局配置：
>   - /etc/profile
>   - /etc/profile.d/*.sh
>   - /etc/bashrc
> - 用户配置：
>   - ~/.bash_profile
>   - ~/.bashrc

4 bash 中的算术运算符

`+，-，*，/，%，**`

实现算术运算的方式：

```
(1) let var= 算术表达式
(2) var=$『算术表达式』
(3) var=$(（算术表达式）)
(4) var=$(expr arg1 arg2 arg3...)
```

乘法符号在有些场景中需要转义；

- bash 内建随机数生成器：$RANDOM
- 增强型赋值： +=，-=，*=，/=，%=

  let varOPERvalue：

  ```
  例如：letcount+=1
  ```

- 自增，自减：

  let var+=1

  ```
  let var++
  ```

  let var-=1

  ```
  let var--
  ```

5 条件测试

判断某需求是否满足，需要有测试机制来实现；

- Note：专用的测试表达式需要由测试命令辅助完成测试过程；
- 测试命令：

  test EXPRESSION

[EXPRESSION]

[[EXPRESSION]]

```
Note: EXPRESSION 前后必须有空白字符，否则报错；
```

Bash 的测试类型

- 数值测试：

  -gt : 是否大于； >

  -ge：是否大于等于； >=

  -eq：是否等于 ==

  -ne：是否不能于 !=

  -lt ：是否小于 <

  -le ：是否小于等于； <=

- 字符串测试：

  - `==` : 是否等于；
  - `>` : 是否大于；
  - `<` : 是否小于；
  - `!=` : 是否不等于；
  - `=~` : 左侧字符串是否能够被右侧的 PATTERN 所匹配；

- Note：此表达式一般用于 [[ ]] 中；

  -z “STRING” : 测试字符串是否为空，空则为真，不空则为假；

  -n “STRING” ：测试字符串是否不空，不空则为真，空则为假；

Note：在字符串比较时用到的操作数都应该使用引号；

文件测试

- (a) 存在性测试：

  -a FILE

  -e FILE：文件存在性测试，存在为真，否则为假；

- (b) 存在性及类别测试

  -b FILE ：是否存在且为块设备文件；

  -c FILE ：是否存在且为字符设备文件；

-d FILE ：是否存在且为目录文件；

-f FILE ：是否存在且为普通文件；

-h FILE 或 –L FILE：是否存在且为符号链接文件；

-p FILE：是否存在且为命名管道文件；

-S FILE ：是否存在且为套接字文件；

- (c) 文件权限测试：

  -r FILE：是否存在且可读

  -w FILE: 是否存在且可写

  -x FILE: 是否存在且可执行

- (d) 文件特殊权限测试：

  -g FILE：是否存在且拥有 sgid 权限；

  -u FILE：是否存在且拥有 suid 权限；

  -k FILE：是否存在且拥有 sticky 权限；

- (e) 文件大小测试：

  -s FILE：是否存在且非空；

- (f) 文件是否打开：

  -t fd：fd 表示文件描述符是否已经打开且与某终端相关

  -N FILE：文件自从上一次被读取之后是否被修改过；

  -O FILE：当前有效用户是否为文件属主；

  -G FILE：当前有效用户是否为文件属组；

- (g) 双目测试：

  FILE1 –ef FILE2 ： FILE1 与 FILE2 是否指向同一个设备上的相同 inode；

  FILE1 –nt FILE2 ：FILE 是否新于 FILE2；

  FILE1 –ot FILE2 ：FILE1 是否旧于 FILE2；

- (h) 组合测试条件：

  完成逻辑运算：

  第一种方式：

```
        COMMAND1 && COMMAND2

        COMMAND1 || COMMAND2

        !COMMAND

    eg：[ -e FILE ] && [ -r FILE ] 文件是否存在且是否有读权限；
```

```
第二种方式：

        EXPRESSION1 –a EXPRESSION2

        EXPRESSION1 –o EXPRESSION2

        !EXPRESSION

必须使用测试命令进行；
```

- Example

```
# [ -z "$hostName" -o "$hostName"=="localhost.localdomain" ] && hostname hostname_
w
# -z 判断 hostName 是否为空，-o 表示或者，即：hostName 为空或者值为 localhot.localdomain
 的时候，使用 hostname 命令修改主机名；
```

- Example

```
[root@bill ~]# [ -f /bin/cat -a -x /bin/cat ] && cat /etc/fstab
#判断文件 /bin/cat 是否存在且是否有可执行权限，&& 是短路与，如果前面执行结果为真则使用 cat 命令#查
看文件；
#
# /etc/fstab
# Created by anaconda on Fri Jul 3 03:08:29 2015
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/centos-root / xfs defaults 0 0
UUID=3d04d82b-c52b-4184-8d64-1826db6e2eac /boot xfs defaults 0 0
/dev/mapper/centos-home /home xfs defaults 0 0
/dev/mapper/centos-swap swap swap defaults 0 0
```

- 快速查找选项定义的方法：

  man bash --> /^[[:space:]]*-f

```
# [ -f /bin/cat -a -x /bin/cat ] && cat /etc/fstab
#如果文件存在且有执行权限，就用它查看文件内容；
```

6 bash 脚本编程之用户交互

- read 命令：

  read [option]… [name….]

  ```
  -p “prompt” 提示符；

  -t TIMEOUT 用户输入超时时间；
  ```

- 检测脚本中的语法错误：

  ```
  bash –n /path/to/some_script
  ```

- 调试执行，查看执行流程：

  ```
  bash –x /path/to/some_script
  ```

- Example

```
#!/bin/bash
#
#Description: Test read command's grammer.
read -t 50 -p "Enter a disk special file:" diskfile #将用户输入的内容赋值给 diskfile 变量
[ -z "$diskfile" ] && echo "Fool" && exit 1
#判断 diskfile 的值是否为空，如果为空则输出，并退出；
if fdisk -l | grep "^Disk $diskfile" &> /dev/null;then
 fdisk -l $diskfile
else
 echo "Wrong disk special file."
 exit 2
fi
```

7 流程控制

if 语句

```
if 语句：
     CONDITION：
     bash 命令：
```

用命令的执行状态结果：

```
 成功：true，即执行状态结果值为 0 时；

 失败：false，即执行状态结果为 1-255 时；
```

成功或失败的定义：取决于用到的命令；

- 单分支 if：

  ```
  if CONDITION；then
     if-true（条件为真时的执行语句集合）
  fi
  ```

- Example

  ```
  #!/bin/bash
  #if 单分支语句测试；
  if [ $UID -eq 0 ];then
     echo "It's amdinistrator."
  fi
  ```

- 双分支 if：

  ```
  if CONDITION;then
    if-true
  else
    if-false
  fi
  ```

- Example

```
#!/bin/bash
#
#if 双分支语句测试；
if [ $UID -eq 0 ];then
 echo "It's administrator."
else
 echo "It's Comman User."
fi
```

- 多分支 if：

```
if CONDITION1；then
   if-true
elif CONDITION2;then
   if-true
elif CONDITION3;then
   if-true
….
else
   all-false
fi
```

逐条件进行判断，第一次遇为“眞”条件时，执行其分支，而后结束；

- Exmaple: 用户键入文件路径，脚本来判断文件类型；

```
#!/bin/bash
#
#if 语句多分支语句；
read -t 20 -p "Enter a file path:" filename

if [ -z "$filename" ];then #判断变量是否为空；
 echo "Usage:Enter a file path."
 exit 2
fi

if [ ! -e $filename ];then #判断用户输入的文件是否存在；
 echo "No such file."
 exit 3
fi

if [ -f $filename ];then #判断是否为普通文件；
 echo "A common file."
elif [ -d $filename ];then #判断是否为目录；
 echo "A directory"
elif [ -L $filename ];then #判断是否为链接文件；
 echo "A symbolic file."
else
 echo "Other type."
fi
```

for 循环

for 循环基础

循环体：要执行的代码，可能要执行 n 遍；

循环需具备进入循环的条件和退出循环的条件；

- for 循环：

```
for 变量名 in 列表；do
   循环体
done
```

- 执行机制：

  依次将列表中的元素赋值给“变量”；每次赋值后即执行一次循环体；直到列表中的元素耗尽，循环结束；

Example 添加 **10** 个用户，用户名为：**user1-user10**：密码同用户名

```
#!/bin/bash
#
#for 循环，使用列表；
#添加 10 个用户，user1-user10

if [ ! $UID -eq 0 ];then #判断执行脚本的是否为 root 用户，若不是则直接退出；
    echo "Only root can use this script."
    exit 1
fi

for i in {1..10};do
    if id user$i &> /dev/null;then    #判断用户是否已经存在；
        echo "user$i exists"
    else
        useradd user$i
        if [ $? -eq 0 ];then   #判断前一条命令是否执行成功；
            echo "user$i" | passwd --stdin user$i &> /dev/null
            #passwd 命令从标准输入获得命令，即管道前命令的执行结果；
        fi
    fi
done
```

- 列表生成方式：

  (1) 直接给出列表 for i in {bill johnson rechard}

  (2) 整数列表

  ```
  (a) {start..end}

  (b) $(seq [start [step]] end)
  ```

  (3) 返回列表的命令

```
$(COMMAND) --> 如：$(ls /var)
```

(4) glob

```
/etc/rc.d/rc3.d/K*
```

(5) 变量引用

```
$@ , $*  -->所有向脚本传递的参数；
```

Example 判断某路径下所有文件的类型

```
#!/bin/bash
#
#for 循环使用命令返回列表；

for file in $(ls /var);do #使用命令生成列表；
 if [ -f /var/$file ];then
     echo "Common file."
 elif [ -L /var/$file ];then
     echo "Symbolic file."
 elif [ -d /var/$file ];then
     echo "Directory."
 else
     echo "Other type"
 fi
done
```

Example 使用 **for** 循环统计关于 **tcp** 端口监听状态

```
#!/bin/bash
#
#使用 for 循环过滤 netstat 命令中关于 tcp 的信息；
declare -i estab=0
declare -i listen=0
declare -i other=0

for state in $( netstat -tan | grep "^tcp\>" | awk '{print $NF}');do

 if [ "$state" == 'ESTABLISHED' ];then
     let estab++
 elif [ "$state" == 'LISTEN' ];then
     let listen++
 else
     let other++
 fi
done

echo "ESTABLISHED:$estab"
echo "LISTEN:$listen"
echo "Unknow:$other"
```

for 循环的特殊格式

```
    for ((控制变量初始化；条件判断表达式；控制变量的修正表达式))；do
        循环体
    done
```

此种格式和 C 语言等的格式是一样一样的，只是多了一对括号；

控制变量初始化：仅在运行到循环代码段时执行一次；

控制变量的修正表达式：每轮循环结束会先进行控制变量修正运算，而后再在条件判断；

Example 求 100 以内所有正整数之和

```
#!/bin/bash
#
#for 循环，类似 C 语言格式，求 100 以内正整数之和；

declare -i sum=0

for ((i=1;i<=100;i++));do
    let sum+=$i
done

echo "Sum:$sum."
```

while 循环

while 基础

语法：

```
while CONDITION；do
    循环体
done
```

```
CONDITION：循环控制条件；进入循环之前，先做一次判断；
每一次循环之后会再次做判断；条件为“true”，则执行一次循环；
直到条件测试状态为“false”终止循环；

因此：CONDTION 一般应该有循环控制变量；
而此变量的值会在循环体不断地被修正，直到最终条件为 false，结束循环。
```

Example 用 while 求 100 以内所有正整数之和

```
#!/bin/bash
#使用 while 求 100 以内正整数之和；
declare -i sum=0
declare -i i=1
while [ $i -le 100 ];do
 let sum+=$i
 let i++
done
echo $i
echo "Summary:$sum."
```

Example 用 while 添加 10 个用户

```
#!/bin/bash
#
#使用 while 循环添加 10 个用户
declare -i i=1
declare -i users=0

while [ $i -le 10 ];do
 if ! id user$i &> /dev/null;then
     useradd user$i
     echo "Add user: user$i"
     let users++
 fi
 let i++
done
echo "Add $users users."
```

Example 利用 **RANDOM** 生成 **10** 个随机数字，输出这 **10** 个数字，并显示其中的最大者和最小者

```
#!/bin/bash
#
#利用 RANDOM 生成 10 个随机数，输出，并求最大值和最小值；
declare -i max=0
declare -i min=0
declare -i i=1

while [ $i -le 9 ];do
    rand=$RANDOM
    echo $rand

    if [ $i -eq 1 ];then
        max=$rand
        min=$rand
    fi

    if [ $rand -gt $max ];then
        max=$rand
    fi
    if [ $rand -lt $min ];then
        min=$rand
    fi
    let i++
done

echo "MAX:$max."
echo "MIN:$min."
```

创建死循环

```
while true;do
    循环体
done
```

```
until false；do
    循环体
done
```

Example 每隔 **3** 秒钟到系统上获取已经登录的用户信息；如果用户输入的用户名登录了，则记录于日志中，并退出

```
#!/bin/bash
#
#用 while 造成死循环，在系统上每隔 3 秒判断一次用户输入的用户名是否登录；
read -p "Enter a user name:" username

while true;do
    if who | grep "^$username" &> /dev/null;then
        break
    fi
    sleep 3
done

echo "$username logggen on." >> /tmp/user.log
```

while 循环遍历文件的每一行

```
    while read line;do
        循环体
    done < /PATH/FROM/SOMEFILE
```

依次读取 /PATH/FROM/SOMEFILE 文件中的每一行，且将该行赋值给变量 line；

另一种也很常见的用法【推荐】：

```
command | while read line
do
    ...
done
```

Example 依次读取 **/etc/passwd** 文件中的每一行，找出其 **ID** 号为偶数的所有用户，显示其用户名、**ID** 号及默认 **shell**

```
#!/bin/bash
#while 循环的特殊用法 读取指定文件的每一行并赋值给变量

while read line;do
    if [ $[`echo $line | cut -d: -f3` % 2] -eq 0 ];then
        echo -e -n "username:`echo $line | cut -d: -f1`\t"
        echo -e -n "uid: `echo $line | cut -d: -f3`\t"
        echo "SHELL:`echo $line | cut -d: -f7`"
    fi
done < /etc/passwd
```

while 与 for 的区别

行读取

> - while 循环
>   - 以行读取文件
> - for 循环
>   - 以空格和回车符分割读取文件，也就是碰到空格和回车，都会执行循环体，所以需要以行读取的话，就要把文件行中的空格转换成其他字符。

ssh 命令操作

> - for 循环
>   - for line in $(cat $file) 在循环体中进行 ssh 命令操作可以依次执行
> - while 循环
>   - 循环体内有 ssh、scp、sshpass 的时候会执行一次循环就退出的情况，解决该问题方法有如下两种
>   - a、使用 `ssh -n "command"` ；
>   - b、将 while 循环内加入 null 重定向，如 `ssh "cmd" < /dev/null` 将 ssh 的输入重定向输入。

case 语句

```
case 变量引用 in
PAT1)
        分支 1
        ;
PAT2）
        分支 2
        ;
….
*)
        默认分支
        ;
esac
```

case 支持 glob 风格的通配符：

```
    *: 任意长度任意字符；
    ?: 任意单个字符；
    []：指定范围内的任意单个字符；
    a|b: a 或 b
```

Example 使用 **case** 语句改写前一个练习

```
#!/bin/bash
#
cat << EOF
cpu) show cpu information;
mem) show memory information;
disk) show disk information;
quit) quit
============================
EOF
read -p "Enter a option: " option
while [ "$option" != 'cpu' -a "$option" != 'mem' -a "$option" != 'disk' -a "$option" !
= 'quit' ]; do
    read -p "Wrong option, Enter again: " option
done

case "$option" in
cpu)
    lscpu
    ;;
mem)
    cat /proc/meminfo
    ;;
disk)
    fdisk -l
    ;;
*)
    echo "Quit..."
    exit 0
    ;;
esac
```

8 函数

函数基础

```
函数的作用：
过程式编程：为实现代码重用
      模块化编程
      结构化编程；
```

- 语法一：

```
function f_name {
  …函数体…..
}
```

- 语法二：

```
f_name() {
  …函数….
}
```

- 函数调用：函数只有被调用才会执行：

  调用：给定函数名

```
  函数名出现的地方，会被自动替换为函数代码；
```

- 函数的生命周期：被调用时创建，返回时终止；

  return 命令返回自定义状态结果；

```
  0：成功

  1-255：失败
```

Example 通过函数，创建 **10** 个用户

```
#!/bin/bash
#
#通过调用函数添加 10 个用户

function adduser {
    if id $username &> /dev/null;then
        echo "$username exists."
        return 1
    else
        useradd $username
        [ $? -eq 0 ] && echo "Add $username finished." && return 0
    fi
}

for i in {1..10};do
    username=myuser$i
    adduser
done
```

Example 编写一个服务启动关闭脚本

```
#!/bin/bash
# chkconfig: - 88 12
# description: test service script
```

```
prog=$(basename $0)
lockfile=/var/lock/subsys/$prog
start() {
    if [ -e $lockfile ];then
        echo "$prog is already running."
        return 0
    else
        touch $lockfile
        [ $? -eq 0 ] && echo "Starting $prog finished."
    fi
}
stop() {
    if [ -e $lockfile ];then
        rm -f $lockfile && echo "Stop $prog ok."
    else
        echo "$prog is stopped yet."
    fi
}
status() {
    if [ -e $lockfile ];then
        echo "$prog is running."
    else
        echo "$prog is stopped."
    fi
}
usage() {
    echo "Usage:$prog {start | stop | restart | status}"
}
if [ $# -lt 1 ];then
    usage
    exit 1
fi

case $1 in
start)
    start
    ;;
stop)
    stop
    ;;
restart)
    stop
    start
    ;;
status)
    status
    ;;
*)
    usage
esac
```

函数返回值

函数的执行结果返回值：

(1)　使用 echo 或 print 命令进行输出；

(2)　函数体中调用命令的执行结果；

函数的退出状态码：

(1)　默认取决于函数体中执行的最后一条命令的退出状态码；

(2)　自定义退出状态码；

使用 return 关键字；

函数可以接受参数：

传递参数给函数：调用函数时，在函数名后面以空白分隔给定参数列表即可；

例如：“testfunc arg1 arg2 …”

在函数体当中，可使用 $1,$2,…. 调用这些参数；还可以使用 $@,$*,$#等特殊变量；

```
#!/bin/bash
#
#使用带参数的函数添加 10 个用户

function adduser {            #一个可接受参数的函数
    if [ $# -lt 1 ];then
        return 2   # 2: no arguments
    fi

    if id $1 &> /dev/null;then
        echo "$1 exists."
        return 1;
    else
        useradd $1
        [ $? -eq 0 ] && echo "Add $1 finished." && return 0
    fi
}

while true;do        #死循环，直到输入的值为 quit 是退出；
    read -t 20 -p "Please input your username(quit to cancel):" username
    if [ $username == "quit" ];then
        echo "Quit."
        exit 0
    else
        adduser $username
    fi
done
```

- 变量作用域：

  本地变量：当前 shell 进程，为了执行脚本会启动专用的 shell 进程；因此，本地变量的作用范围是当前 shell 脚本程序文件；

  局部变量：函数的生命周期：函数结束时变量被自动销毁；

  > 如果函数中有局部变量，其名称同本地变量无关；

在函数中定义局部变量的方法：

```
    local NAME=VALUE
```

函数递归：函数直接或间接调用自身；

Example 求 N 的阶乘

```
N！=N（N-1）(N-2)….1
    N(N-1)!=N(N-1)(N-2)!
```

```
#!/bin/bash
#
#利用函数递归求 N 的阶乘

fact() {
    if [ $1 -eq 0 -o $1 -eq 1 ];then
        echo 1
    else
        echo $[$1*$(fact $[$1-1])]
    fi
}
fact 5
```

9 数组

数组

> - 变量：存储单个元素的内存空间；
> - 数组：存储多个元素的连续的内存空间；
>   - 索引：编号从 0 开始，属于数值索引；
>   - 注意：索引页可支持使用自定义的格式，而不仅仅是数值格式；bash 的数组支持稀疏格式；

定义

在 Shell 中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的一般形式为：

```
array_name=(value1 ... valuen)
```

例如：

```
array_name=(value0 value1 value2 value3)
```

或者

```
array_name=(
value0
value1
value2
value3
)
```

还可以单独定义数组的各个分量：

```
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2
```

Example 生成 **10** 个随机数保存于数组中，并找出其最大值和最小值

```
#!/bin/bash
#
#生成 10 个随机数保存于数组中；

declare -a rand
declare -i max=0
declare -i min=0

for i in {0..9};do
    rand[$i]=$RANDOM
    if [ $i -eq 1 ];then
        max=${rand[$i]}
        min=${rand[$i]}
    fi
    echo ${rand[$i]}
    [ ${rand[$i]} -gt $max ] && max=${rand[$i]}
    [ ${rand[$i]} -lt $min ] && min=${rand[$i]}
done

echo "Max:$max"
echo "Min:$min"
```

Example 定义一个数组，数组中的元素是 **/var/log** 目录下所有以**.log** 结尾的文件；要统计其下标为偶数的文件中的行数之和

```
#!/bin/bash
#
#定义一个数组，数组中的元素是 /var/log 目录下所有以.log 结尾的文件；
#要统计其下标为偶数的文件中的行数之和；

declare -a files
files=(/var/log/*.log)
declare -i lines=0

for i in $(seq 0 $[${#files[*]}-1]);do
    if [ $[$i%2] -eq 0 ];then
        let lines+=$(wc -l ${files[$i]} | cut -d' ' -f1)
    fi
done

echo "Lines:$lines."
```

引用数组中的元素

- 所有元素：${ARRAY[@]} , ${ARRAY[*]}
- 数组切片：${ARRAY[@]:offset:number}

  ```
  offset: 要跳过的元素个数；

  number：要取出的元素个数，取偏移量之后的所有元素：${ARRAY[@]:offset};
  ```

- 向数组中追加元素：ARRAY[${ARRAY[*]}]
- 删除数组中的某元素：unset ARRAY[INDEX]
- 关联数组： declare –A ARRAY_NAME ARRAY_NAME=([index_name1]=’val1’ [index_name2]=’val2’ ….)

```
[root@bill scripts]# declare -a array
[root@bill scripts]# #声明一个数组，不是必要的
[root@bill scripts]# array=(0 1 2)
[root@bill scripts]# array=([0]=0 [1]=1 [2]=v2)
[root@bill scripts]# array[0]=5
[root@bill scripts]# echo $array
5
```

```
[root@bill scripts]# #以空白作为分隔符拆分字符串为数组
[root@bill scripts]# str="1 2 3"
[root@bill scripts]# array=($str)
[root@bill scripts]# echo $array
1
```

```
[root@bill scripts]# #使用其他分隔符拆分字符串为数组，需指定 IFS
[root@bill scripts]# IFS=: array=($PATH)
[root@bill scripts]# echo $array
/usr/local/sbin
```

- 引用数组元素： $array ${array} ${array[0]} #第 0 个元素

  ${array[n]} #第 n 个元素（n 从 0 开始计算）

- 引用整个数组： ${array[*]} ${array[@]} 这两种方式等同，会把数组展开。

  ${array[*]} 表示把数组拼接在一起的整个字符串，如果作为参数传递，会把整个字符串作为一个参数。

  ${array[@]} 如果作为参数传递，表示把数组中每个元素作为一个参数，数组有多少个元素，就会展开成多少个参数。

- 计算数组元素长度；

  ${#array[*]}

  ${#array[@]}

  ```
  不是 ${#array}，

  因为它等同于 ${#array[0]}
  ```

- 遍历数组： for i in "${array[@]}";do

  ```
  echo $i;
  ```

  done

10 bash 的字符串处理工具

字符串切片

- ${var:offset:number}

取字符串最右侧的几个字符：${var: -lengh}

Note：冒号后面必须有一空白字符；

基于模式取子串

- ${var#*word}：其中 word 可以是指定的任意字符；

  功能：自左而右，查找 var 变量所存储的字符串中，第一次出现的 word, 删除字符串开头至第一次出现 word 字符之间的所有字符；

- ${var##*word}：其中 word 可以是指定的任意字符；

  功能：自左而右，查找 var 变量所存储的字符串中出现的 word，删除字符串开头至最后一次由 word 指定的字符之间的所有内容；

bash 基于模式取子串

```
[root@bill scripts]# file="/var/log/messages"
[root@bill scripts]# echo ${file#*/}
var/log/messages
[root@bill scripts]# echo ${file##*/}
messages
```

- ${var%word*}：其中 word 可以是指定的任意字符；

  功能：自右而左，查找 var 变量所存储的字符串中，第一次出现的 word, 删除字符串最后一个字符向左至第一次出现 word 字符之间的所有字符；

- ${var%%word*}：其中 word 可以是指定的任意字符；

  功能：自右而左，查找 var 变量所存储的字符串中出现的 word,，只不过删除字符串最右侧的字符向左至最后一次出现 word 字符之间的所有字符；

  ```
  [root@bill scripts]# file="/var/log/messages" #从右至左，匹配'/ '
  [root@bill scripts]# echo ${file%/*}
  /var/log
  [root@bill scripts]# echo ${file%%/*}     # 双 % 匹配并删除后为空；
  [root@bill scripts]#
  ```

Exampleurl=http://www.google.com:80, 分别取出协议和端口

```
[root@bill scripts]# url=http://www.google.com:80
[root@bill scripts]# echo ${url##*:}    #取端口号
80
[root@bill scripts]# echo ${url%%:*}   #取协议
http
[root@bill scripts]#
```

查找替换

```
${var/pattern/substi}：查找 var 所表示的字符串中，第一次被 pattern 所匹配到的字符串，以 substi 替换之；

${var//pattern/substi}: 查找 var 所表示的字符串中，所有能被 pattern 所匹配到的字符串，以 substi 替换之；

${var/#pattern/substi}：查找 var 所表示的字符串中，行首被 pattern 所匹配到的字符串，以 substi 替换之；

${var/%pattern/substi}：查找 var 所表示的字符串中，行尾被 pattern 所匹配到的字符串，以 substi 替换之；
```

```
[root@bill scripts]# var=$(head -n 1 /etc/passwd)
[root@bill scripts]# echo $var
root x 0 0 root /root /bin/bash
[root@bill scripts]# echo ${var/root/ROOT}  #替换第一次匹配到的 root 为 ROOT
ROOT x 0 0 root /root /bin/bash
[root@bill scripts]# echo ${var//root/ROOT}  #替换所有匹配到的 root 为 ROOT
ROOT x 0 0 ROOT /ROOT /bin/bash
```

```
[root@bill scripts]# useradd bash -s /bin/bash
[root@bill scripts]# cat /etc/passwd | grep "^bash.*bash$"
bash:x:1029:1029::/home/bash:/bin/bash
[root@bill scripts]# line=$(cat /etc/passwd | grep "^bash.*bash$")
[root@bill scripts]# echo $line
bash x 1029 1029  /home/bash /bin/bash
[root@bill scripts]# echo ${line/#bash/BASH}    #替换行首的 bash 为 BASH
BASH x 1029 1029  /home/bash /bin/bash
[root@bill scripts]# echo ${line/%bash/BASH}    #替换行尾的 bash 为 BASH
bash x 1029 1029  /home/bash /bin/BASH
```

查找并删除

```
${var/pattern}：查找 var 所表示的字符串中，删除第一次被 pattern 所匹配到的字符串

${var//pattern}：查找 var 所表示的字符串中，删除所有被 pattern 所匹配到的字符串；

${var/#pattern}：查找 var 所表示的字符串中，删除行首被 pattern 所匹配到的字符串；

${var/%pattern}：查找 var 所表示的字符串中，删除行尾被 pattern 所匹配到的字符串；
```

- Example ``` [root@bill scripts]# line=$(tail -n 1 /etc/passwd) [root@bill scripts]# echo $line bash x 1029 1029 /home/bash /bin/bash [root@bill scripts]# echo ${line/bash} #查找并删除第一次匹配到的 bash x 1029 1029 /home/bash /bin/bash [root@bill scripts]# echo ${line//bash} #查找并删除所有匹配到的 bash x 1029 1029 /home/ /bin/ [root@bill scripts]# echo ${line/#bash} #查找并删除匹配到的行首的 bash x 1029 1029 /home/bash /bin/bash [root@bill scripts]# echo ${line/%bash} #查找并删除匹配到的行尾 bash bash x 1029 1029 /home/bash /bin/

```
### 字符大小写转换
    ${var^^}：把 var 中的所有小写字母转换为大写；

    ${var,,}：把 var 中的所有大写字母转换为小写；

- Example
```

[root@bill scripts]# line=$(tail -n 1 /etc/fstab) #将文件最后一行的值赋值给变量 [root@bill scripts]# echo ${line^^} #全部转换为大写后输出 /DEV/MAPPER/CENTOS-SWAP SWAP SWAP DEFAULTS 0 0 [root@bill scripts]# line= `echo ${line^^}` #将转换为大写后的值在赋值给变量； [root@bill scripts]# echo $line #确认目前变量的值全为大写字母； /DEV/MAPPER/CENTOS-SWAP SWAP SWAP DEFAULTS 0 0 [root@bill scripts]# echo ${line,,} #全部转换为大写后输出； /dev/mapper/centos-swap swap swap defaults 0 0

```
### 变量赋值

    ${var:-value}：如果 var 为空或未设置，那么返回 value；否则，则返回 var 的值；

    ${var:=value}：如果 var 为空或未设置，那么返回 value，并将 value 赋值给 var；否则，则返回 v
ar 的值；

- Example
```

[root@bill scripts]# echo $test #变量值为空；

[root@bill scripts]# echo ${test:-helloworld} # :- 仅返回设定值，不修改； helloworld [root@bill scripts]# echo $test #变量的值依然为空；

[root@bill scripts]# echo ${test:=helloworld} # := 返回设定值，并赋值给变量； helloworld
[root@bill scripts]# echo $test # 变量值已修改； helloworld

```
## 11 Bash 命令自动补全

### 11.1 内置补全命令

Bash 内置有两个补全命令，分别是 compgen 和 complete。compgen 命令根据不同的参数，生成匹配单词的候选补全列表，例如：
```

$ compgen -W 'hi hello how world' h hi hello how

```
compgen 最常用的选项是 -W，通过 -W 参数指定空格分隔的单词列表。h 即我们在命令行当前键入的单词，执行完后会输出候选的匹配列表，这里是以 h 开头的所有单词。

complete 命令的参数有点类似 compgen，不过它的作用是说明命令如何进行补全，例如同样使用 -W 参数指定候选的单词列表：
```

$ complete -W 'word1 word2 word3 hello' foo $ foo w $ foo word word1 word2 word3

```
我们还可以通过 -F 参数指定一个补全函数：
```

$ complete -F _foo foo

```
现在键入 foo 命令后，会调用_foo 函数来生成补全的列表，完成补全的功能，这一点正是补全脚本实现的关键所在，我们会在后面介绍。

补全相关的内置变量

除了上面的两个补全命令外，Bash 还有几个内置的变量用来辅助补全功能，这里主要介绍其中三个：

> * COMP_WORDS: 类型为数组，存放当前命令行中输入的所有单词；
> * COMP_CWORD: 类型为整数，当前光标下输入的单词位于 COMP_WORDS 数组中的索引；
> * COMPREPLY: 类型为数组，候选的补全结果；
> * COMP_WORDBREAKS: 类型为字符串，表示单词之间的分隔符；
> * COMP_LINE: 类型为字符串，表示当前的命令行输入；

例如我们定义这样一个补全函数_foo：
```

$ function _foo() { echo -e "\n" declare -p COMP_WORDS declare -p COMP_CWORD declare -p COMP_LINE declare -p COMP_WORDBREAKS } $ complete -F _foo foo

```
假设我们在命令行下输入以下内容，再按下 Tab 键补全：
```

$ foo b

declare -a COMP_WORDS='([0]="foo" [1]="b")' declare -- COMP_CWORD="1" declare -- COMP_LINE="foo b" declare -- COMP_WORDBREAKS=" \"'><=;|&(:"

对着上面的结果，我想应该比较容易理解这几个变量。当然正如我们之前据说，Bash-completion 包并非是必须的，补全功能是 Bash 自带的。

11.2 编写脚本

补全脚本分成两个部分：编写一个补全函数和使用 complete 命令应用补全函数。后者的难度几乎忽略不计，重点在如何写好补全函数。难点在，似乎网上很少与此相关的文档，但是事实上，Bash-completion 自带的补全脚本是最好的起点，可以挑几个简单的改改基本上就可以使用了。

一般补全函数（假设这里依然为_foo) 都会定义以下两个变量：

local cur prev

其中 cur 表示当前光标下的单词，而 prev 则对应上一个单词：

cur="${COMP_WORDS[COMP_CWORD]}" prev="${COMP_WORDS[COMP_CWORD-1]}"

11.2.1 支持主选项

初始化相应的变量后，我们需要定义补全行为，即输入什么的情况下补全什么内容，例如当输入 - 开头的选项的时候，我们将所有的选项作为候选的补全结果：

local opts="-h --help -f --file -o --output"

if [[${cur} == -*]] ; then COMPREPLY=($(compgen -W "${opts}" -- ${cur})) return 0 fi

不过再给 COMPREPLY 赋值之前，最好将它重置清空，避免被其它补全函数干扰。

现在完整的补全函数是这样的：

function _foo() { local cur prev opts

```
COMPREPLY=()

cur="${COMP_WORDS[COMP_CWORD]}"
prev="${COMP_WORDS[COMP_CWORD-1]}"
opts="-h --help -f --file -o --output"

if [[ ${cur} == -* ]] ; then
    COMPREPLY=( $(compgen -W "${opts}" -- ${cur}) )
    return 0
fi
```

}

现在在命令行下就可以对 foo 命令进行参数补全了：

$ complete -F _foo foo $ foo - -f --file -h --help -o --output

11.2.2 支持子选项
当然，似乎我们这里的例子没有用到 prev 变量。用好 prev 变量可以让补全的结果更加完整，例如当输入 --file 之后，我们希望补全特殊的文件（假设以.sh 结尾的文件）：

```
case "${prev}" in
    -f|--file)
        COMPREPLY=( $(compgen -o filenames -W "`ls *.sh`" -- ${cur}) )
        ;;
esac
```

现在再执行 foo 命令，--file 参数的值也可以补全了：

$ foo --file a.sh b.sh c.sh

11.2.3 安装补全脚本

如果安装了 Bash-completion 包，可以将补全脚本放在 /etc/bash_completion.d 目录下，或者放到~/.bash_completion 文件中。
如果没有安装 Bash-completion 包，可以把补全脚本放到~/.bashrc 或者其它能被 shell 加载的初始化文件中。
12 常用实例
12.1 推荐添加内容

set -u # 确保变量都被初始化 set -e # 确保捕获所有非 0 状态 set -o pipefail # 结合 -e 可以捕获管道后的异常状态

```
### 12.2 脚本的配置文件
-    (1) 定义文本文件，每行定义"name=value"
-    (2) 在脚本中 source 此文件即可
```

[root@bill scripts]# touch /tmp/config.test #创建配置文件； [root@bill scripts]# echo "name=bill" >> /tmp/config.test #在配置文件中定义变量； [root@bill scripts]# vim script_configureFile.sh #编写脚本，导入配置文件；如内容所示； [root@bill scripts]# bash script_configureFile.sh #脚本执行结果； bill [root@bill scripts]# cat script_configureFile.sh

!/bin/bash

source /tmp/config.test #导入配置文件，脚本自身并未定义变量；

echo $name #引用的是配置文件中的变量 name

```
### 12.3 ssh 登录相关

> * 可以使用 sshpass 进行直接传入密码
> * 一定要加 -o StrictHostKeyChecking=no 否则如果是第一次访问对应的机器，会执行无效
```

ssh_option="-o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -o NumberOfPasswordPrompts=0 -o ConnectTimeout=3 -o ConnectionAttempts=3" export WSSH="./tools/sshpass -p ${PASSWD} ssh ${ssh_option}" export WSCP="./tools/sshpass -p ${PASSWD} scp ${ssh_option}"

```
> * StrictHostKeyChecking=no          自动信任主机并添加到known_hosts文件
> * UserKnownHostsFile=/dev/null     跳过检查 ~/.ssh/known_hosts 中的公钥操作，比如因为远端机器重装导致无法登录问题
> * NumberOfPasswordPrompts=0        规避没有信任关系挂死的问题，当对应的机器需要输入密码时，会直接返回异常（异常返回码为 255），而不是阻塞在输入密码页面
> * ConnectTimeout=3                  连接超时时间，3秒
> * ConnectionAttempts=3              连接失败后重试次数，3次
---------------------
作者：DemonHunter211
来源：CSDN
原文：https://blog.csdn.net/kwame211/article/details/79076513
版权声明：本文为博主原创文章，转载请附上博文链接！
### 12.4 ping 文件列表中所有主机
```

!/bin/bash

file=$1

[[-z $file]] && exit 0 cat $file| while read line;do ping=`ping -c 1 $line|grep loss|awk '{print $6}'|awk -F "%" '{print $1}'` if [$ping -eq 100];then echo ping $i fail else echo ping $i ok fi done

```
### 12.5 shell 模板变量替换

#### 应用场景

双引号是 json 的标准，如果一个服务的配置文件是 json 的，使用特定的配置（模板）生成对应的配置文件时。
要是使用 shell，这样也可以做到：

#### 使用方式

模板文件
```

{ "test_ip" : ${test_ip}, "test_port" : ${test_port}, }

```
脚本
```

test_host="127.0.0.1" test_port=9099

content=$(cat ./config_tpl) content_new=$(eval "cat <<EOF $content EOF")

echo $content_new

```
## 13 日常使用库
```

f_yellow='\e[00;33m' f_red='\e[00;31m' f_green='\e[00;32m' f_reset='\e[00;0m'

function p_warn { echo -e "${f_yellow}[WRN]${f_reset} ${1}" }

function p_err { echo -e "${f_red}[ERR]${f_reset} ${1}" }

function p_ok { echo -e "${f_green}[OK]${f_reset} ${1}" }

ROOT_PATH= `S=\readlink "$0"`; [-z "$S"] && S=$0; dirname $S` cd ${ROOT_PATH} ```

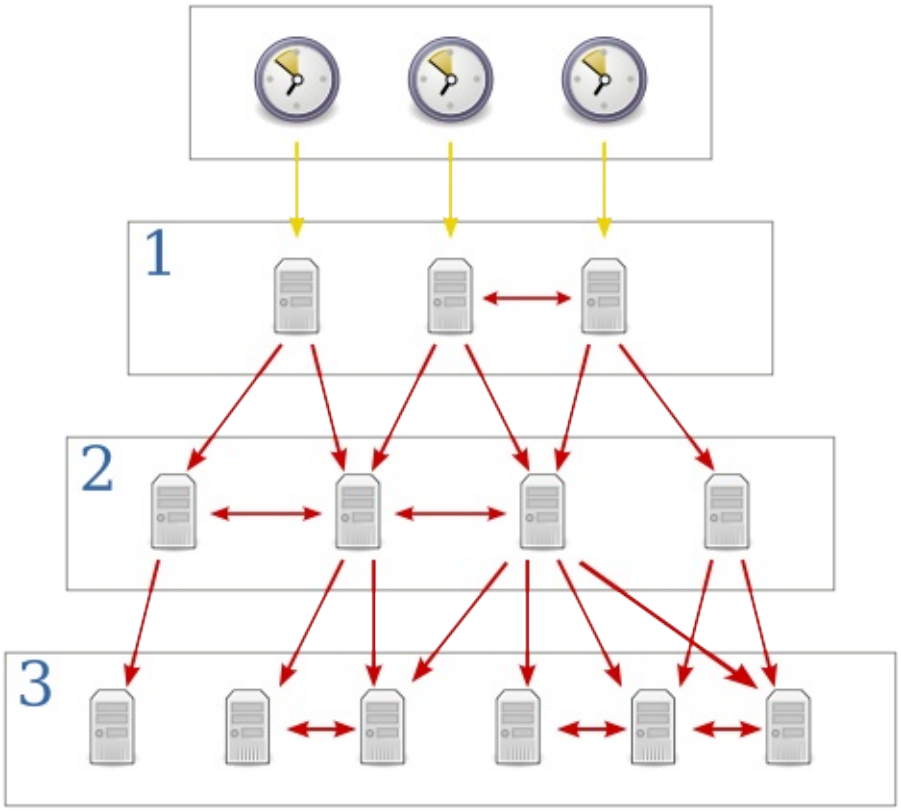
常见服务架设

NTP

简介

Network Time Protocol-NTP 是用来使计算机时间同步化的一种协议，它可以使计算机对其服务器或时钟源（如石英钟，GPS 等等）做同步化，它可以提供高精准度的时间校正（LAN 上与标准间差小于 1 毫秒，WAN 上几十毫秒），且可使用加密确认的方式来防止恶毒的协议攻击。默认使用 `UDP 123 端口`

NTP 提供准确时间，首先需要一个准确的 UTC 时间来源，NTP 获得 UTC 的时间来源可以从原子钟、天文台、卫星，也可从 Internet 上获取。时间服务器按照 NTP 服务器的等级传播，根据离外部 UTC 源的远近将所有服务器归入不用的层 (Stratum) 中。Stratum-1 在顶层由外部 UTC 接入，stratum-1 的时间服务器为整个系统的基础，Stratum 的总数限制在 15 以内。下图为 NTP 层次图：



ntpd

NTP Server 安装配置

关于 NTP 服务器的安装，根据不同版本安装方法也不同。REDHAT 系统则可以使用 yum 安装，Ubuntu 系列可以使用 `apt-get` 安装，这里不做具体的介绍，主要详细介绍配置文件的信息。

对于 CentOS 过滤注释和空行后，NTP 配置文件内容如下

```
# grep -vE '^#|^$' /etc/ntp.conf
driftfile /var/lib/ntp/drift

# 默认对所有 client 拒绝所有的操作
restrict default kod nomodify notrap nopeer noquery
restrict -6 default kod nomodify notrap nopeer noquery

# 允许本机地址的一切操作
restrict 127.0.0.1
restrict -6 ::1

# 允许其他机器连接
restrict default kod nomodify

server 0.centos.pool.ntp.org
server 1.centos.pool.ntp.org
server 2.centos.pool.ntp.org
includefile /etc/ntp/crypto/pw
keys /etc/ntp/keys
```

配置选项说明

- `driftfile` 选项，用来保存系统时钟频率偏差。ntpd 程序使用它来自动地补偿时钟的自然漂移，从而使时钟即使在切断了外来时源的情况下，仍能保持相当的准确度。`无需更改`
- `restrict` 语法为：restrict IP mask 掩码 参数
  - IP 规定了允许或不允许访问的地址（此处若为 default，即为 0.0.0.0 所有 ip），配合掩码可以对某一网段进行限制。
    - `ignore`：关闭所有 NTP 服务
    - `nomodiy`：客户端不能修改服务端的时间，但可以作为客户端的校正服务器
    - `notrust`：拒绝没有通过认证的客户端
    - `kod`：kod 技术科阻止 "Kiss of Death" 包（一种 DOS 攻击）对服务器的破坏
    - `nopeer`：不与其它同一层的 NTP 服务器进行同步
    - `noquery`：不提供时间查询，即用户端不能使用 ntpq，ntpc 等命令来查询 ntp 服务器
    - `notrap`：不提供trap远端事件登陆的功能
- `server [IP|FQDN|prefer]` 指该服务器上层 NTP Server，使用 prefer 的优先级最高，没有

使用 prefer 则按照配置文件顺序由高到低，默认情况下至少 15min 和上层 NTP 服务器进行时间校对

- `fudge`：可以指定本地 NTP Server 层，如 `fudge 127.0.0.1 stratum 9`
- `broadcast 网段 子网掩码`：指定 NTP 进行时间广播的网段，如 `broadcast 192.168.1.255`
- `logfile`：可以指定 NTP Server 日志文件

bill 提醒

```
restrict 用于权限控制，server 用于设定上级时间服务器
主要是这两个参数
```

几个与 NTP 相关的配置文件：`/usr/share/zoneinfo/` 、`/etc/sysconfig/clock` 、`/etc/localtime`

- `/usr/share/zoneinfo/`：存放时区文件目录
- `/etc/sysconfig/clock`：指定当前系统时区信息
- `/etc/localtime`：相应的时区文件

如果需要修改当前时区，则可以从 /usr/share/zoneinfo/ 目录拷贝相应时区文件覆盖 /etc/localtime 并修改 /etc/sysconfig/clock 即可

```
cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
sed -i 's:ZONE=.*:ZONE="Asia/Shanghai":g' /etc/sysconfig/clock
```

相关命令

`ntpstat` 查看同步状态

```
# ntpstat
synchronised to NTP server (192.168.0.18) at stratum 4
   time correct to within 88 ms      # 表面时间校正 88ms
   polling server every 1024 s        # 每隔 1024s 更新一次
```

`ntpq` 列出上层状态

```
# ntpq -np
   remote refid  st t when poll reach   delay   offset  jitter
==============================================================
* NTPD(IP)  IP   3  u  101 1024  377   14.268    0.998   0.143
```

输出说明：

- `remote`：NTP Server

- `refid`：参考的上层 ntp 地址
- `st`：层次
- `when`：上次更新时间距离现在时常
- `poll`：下次更新时间
- `reach`：更新次数
- `delay`：延迟
- `offset`：时间补偿结果
- `jitter`：与 BIOS 硬件时间差异

`ntpdate` 同步当前时间：`ntpdate NTP 服务器地址`

chrony

使用安装命令安装 chrony 包即可

chrony server

配置文件: /etc/chrony.conf

对于 chrony server 来说，主要配置两项，上游的 ntp 服务器和对下游的权限

> - 上游的 ntp 服务器
>   - 有固定的 ntp 服务器或者可连互联网
>   - 配置 `server 0.centos.pool.ntp.org iburst` 即可
>   - 无外网环境使用本地的时间进行往下游同步
>   - 配置 `local stratum 10`
> - 对下游的权限
>   - `allow 10.0.0.0/24` 对 10.0.0 网段开放
>   - `allow 0/0` 对所有 IP 开放

bill 提醒

```
(1) 无外网环境时，如果没有设置 local stratum 0，下游服务器显示的状态是不可达状态
(2) 不加 allow 记录时，默认拒绝所有连接
(3) chrony 端口为 udp 123
```

启动并设置开机自启

```
# systemctl enable chronyd.service
# systemctl start chronyd.service
```

chrony client

对于 client 来说，只需要配置上游的服务器

配置文件: /etc/chrony.conf

添加 `server 上游服务器IP/主机名 iburst` 即可

启动并设置开机自启

```
# systemctl enable chronyd.service
# systemctl start chronyd.service
```

查看同步状态

```
#chronyc sources -v
```

上面命令会输出上游服务器的连接状态

> - `* 正常`
> - `? 不可达`

Cron

Cron 基础

什么是 cron, crond, crontab

> **cron** is the general name for the service that runs scheduled actions. **crond** is the name of the daemon that runs in the background and reads **crontab** files.

简单理解：cron 是服务，crond 是守护进程， crontab 的 crond 的配置文件。

crontab 选项

- `crontab -e` : Edit your crontab file, or create one if it doesn't already exist. # 推荐使用命令新增计划任务 -- 语法检查
- `crontab -l` : Display your crontab file.
- `crontab -r` : Remove your crontab file. # 慎用
- `crontab -u user` : Used in conjunction with other options, this option allows you to modify or view the crontab file of user. When available, only administrators can use this option.

crontab 格式

```
minute(s) hour(s) day(s) month(s) weekday(s) command(s)
```

```
# Use the hash sign to prefix a comment
# +————- minute (0 – 59)
# |  +————- hour (0 – 23)
# |  |  +——- day of month (1 – 31)
# |  |  |  +—- month (1 – 12)
# |  |  |  |  +–- day of week (0 – 7) (Sunday=0 or 7)
# |  |  |  |  |
# *  *  *  *  *  command to be executed
```

使用举例

使用命令 `crontab -e` 编辑 crontab 文件。

(1) 在每天的 7 点同步服务器时间

```
0 7 * * * ntpdate 192.168.1.112
```

(2) 每两个小时执行一次

```
0 */2 * * * echo "2 minutes later" >> /tmp/output.txt
```

(3) 每周五早上十点写周报

```
0 10 * * * 5 /home/jerryzhang/update_weekly.py
```

(4) 每天 6, 12, 18 点执行一次命令

```
0 6,12,18 * * *  /bin/echo hello
```

(5) 每天 13, 14, 15, 16, 17 点执行一次命令

```
0 13-17 * * *  /bin/echo hello
```

注：

- 程序执行完毕，系统会给对应用户发送邮件，显示该程序执行内容，如果不想收到，可

以重定向内容 `> /dev/null 2>&1`

- 如果执行语句中有 `%` 号，需要使用反斜杠 '\' 转义

rsync

rsync 基本介绍

`rsync` 是类 unix 系统下的数据镜像备份工具，从软件的命名上就可以看出来了—— remote sync。它的特性如下：

- 1、可以镜像保存整个目录树和文件系统
- 2、可以很容易做到保持原来文件的权限、时间、软硬链接等等
- 3、无须特殊权限即可安装
- 4、优化的流程，文件传输效率高
- 5、可以使用 rsh、ssh 等方式来传输文件，当然也可以通过直接的 socket 连接
- 6、支持匿名传输

在使用 rsync 进行远程同步时，可以使用两种方式：远程 **Shell** 方式（用户验证由 ssh 负责）和 **C/S** 方式（即客户连接远程 rsync 服务器，用户验证由 rsync 服务器负责）。

无论本地同步目录还是远程同步数据，首次运行时将会把全部文件拷贝一次，以后再运行时将只拷贝有变化的文件（对于新文件）或文件的变化部分（对于原有文件）。

rsync 工作场景

> - 两台服务器之间数据同步。
> - 把所有客户服务器数据同步到备份服务器，生产场景集群架构服务器备份方案。
> - rsync 结合 inotify 的功能做实时的数据同步。

使用方法

rsync 可以使用 ssh 和 C/S 方式进行传输文件，以下使用 ssh 方式

```
rsync [OPTION]... SRC [SRC]... [USER@]HOST:DEST # 执行“推”操作
or   rsync [OPTION]... [USER@]HOST:SRC [DEST]   # 执行“拉”操作
```

rsync 选项

```
Usage: rsync [OPTION]... SRC [SRC]... DEST
  or   rsync [OPTION]... SRC [SRC]... [USER@]HOST:DEST
  or   rsync [OPTION]... SRC [SRC]... [USER@]HOST::DEST
  or   rsync [OPTION]... SRC [SRC]... rsync://[USER@]HOST[:PORT]/DEST
  or   rsync [OPTION]... [USER@]HOST:SRC [DEST]
  or   rsync [OPTION]... [USER@]HOST::SRC [DEST]
  or   rsync [OPTION]... rsync://[USER@]HOST[:PORT]/SRC [DEST]
The ':' usages connect via remote shell, while '::' & 'rsync://' usages connect
to an rsync daemon, and require SRC or DEST to start with a module name.
```

注：在指定复制源时，路径是否有最后的 “/” 有不同的含义，例如：

- /data ：表示将整个 /data 目录复制到目标目录
- /data/ ：表示将 /data/ 目录中的所有内容复制到目标目录

常用选项

- `-v` : Verbose (try -vv for more detailed information) # 详细模式显示
- `-e` "ssh options" : specify the ssh as remote shell # 指定 ssh 作为远程 shell
- `-a` : archive mode # 归档模式，表示以递归方式传输文件，并保持所有文件属性，等于 -rlptgoD
  - `-r` (--recursive) : 目录递归
  - `-l` (--links) ：保留软链接
  - `-p` (--perms) ：保留文件权限
  - `-t` (--times) ：保留文件时间信息
  - `-g` (--group) ：保留属组信息
  - `-o` (--owner) ：保留文件属主信息
  - `-D` (--devices) ：保留设备文件信息
- `-z` ：压缩文件
- `-h` ：以可读方式输出
- `-H` ：复制硬链接
- `-X` ：保留扩展属性
- `-A` ：保留 ACL 属性
- `-n` ：只测试输出而不正真执行命令，推荐使用，特别防止 `--delete` 误删除！
- `--stats` ：输出文件传输的状态
- `--progress` ：输出文件传输的进度
- `--exclude=PATTERN` ：指定排除一个不需要传输的文件匹配模式
- `--exclude-from=FILE` ：从 FILE 中读取排除规则
- `--include=PATTERN` ：指定需要传输的文件匹配模式
- `--include-from=FILE` ：从 FILE 中读取包含规则
- `--numeric-ids` ：不映射 uid/gid 到 user/group 的名字
- `-S, --sparse` ：对稀疏文件进行特殊处理以节省 DST 的空间（有空洞文件时使用）

- `--delete`：删除 DST 中 SRC 没有的文件，也就是所谓的镜像 [mirror] 备份
- `-P` 等同于 `--partial` 保留那些因故没有完全传输的文件，以是加快随后的再次传输

一些命令

常用命令

```
#rsync -avzP --delete [SRC] [DEST]
```

注：日常传输时参数记不清楚时，只需要加 `-a` 参数即可，如果有稀疏文件，则添加 `-S` 选项可以提升传输性能。

```
[tips]
稀疏文件（Sparse File）

在 UNIX 文件操作中，文件位移量可以大于文件的当前长度，在这种情况下，对该文件的下一次写将延长该文件，并在文件中构成一个空洞。位于文件中但没有写过的字节都被设为 0。

稀疏文件与其他普通文件基本相同，区别在于文件中的部分数据是全 0，且这部分数据不占用磁盘空间。
下面是稀疏文件的创建与查看方法
[root@Linux ceshi]# dd if=/dev/zero of=sparse-file bs=1 count=1 seek=1024k
[root@Linux ceshi]#  ls -l sparse-file
-rw-r--r-- 1 root root 1048577 6 月  19 10:20 sparse-file
[root@Linux ceshi]# du -sh sparse-file
4.0K    sparse-file
[root@Linux ceshi]# cat sparse-file  >> meetbill_file
[root@Linux ceshi]# du -sh meetbill_file
1.1M    meetbill_file
[root@Linux ceshi]# ll
总用量 1032
-rw-r--r-- 1 root root 1048577 6 月  19 10:21 meetbill_file
-rw-r--r-- 1 root root 1048577 6 月  19 10:20 sparse-file
[root@Linux ceshi]# ll -h
总用量 1.1M
-rw-r--r-- 1 root root 1.1M 6 月  19 10:21 meetbill_file
-rw-r--r-- 1 root root 1.1M 6 月  19 10:20 sparse-file
```

ssh 端口非默认 22 同步

使用 ssh 方式传输时如果连接服务器 ssh 端口非标准，则需要通过 `-e` 选项指定：

```
#rsync -avzP --delete  -e "ssh -p 22222" [USER@]HOST:SRC [DEST]
```

ssh 自动接受公钥和修改 known_hosts 文件

```
#rsync -a -e "ssh -oUserKnownHostsFile=/dev/null -oStrictHostKeyChecking=no" [USER@]HO
ST:SRC [DEST]
```

inotify+rsync 实现实时文件同步

存储数据异地灾备

需求背景

服务器文件需要实时同步，即使是轮询，也存在同步延迟，inotify 的出现让真正的实时成为了现实 我们可以用 inotify 去监控文件系统的事件变化，一旦有我们期望的事件发生，就使用 rsync 进行冗余同步

架构

用途	IP
服务端 A	192.168.199.101
服务器 B（备份服务器）	192.168.199.102

```
   +--------+          +--------------------+
   |服务器 A |--------->|服务器 B（备份服务器）|
   +--------+          +--------------------+

  inotify+rsync             rsync
```

脚本内容

所有配置只需要在服务器 A 上配置即可

(1) 安装 `inotify-tools` (yum -y install inotify-tools)

(2) 配置服务器 A 使用秘钥登录服务器 B

(3) 在服务器 A 上编写脚本，主要配置服务器 B 的机器 IP，登录用户，以及服务器器 A 的存储目录和存储数据异地灾备目录

将此文件保存到 /opt/inotify_rsync.sh

```
    #!/bin/bash
    host=192.168.199.102
    user=root
    # 服务器存储目录
    src='/tmp/src1/'
    # 存储数据异地灾备目录
    dest='/tmp/dest1'

    inotifywait -mrq -e modify,attrib,moved_to,moved_from,move,move_self,create,delete
,delete_self --timefmt='%d/%m/%y %H:%M' --format='%T %w%f %e' $src | while read chgeFi
le
    do
        rsync -avPz --delete $src $user@$host:$dest &>>./rsync.log
    done
```

下载脚本

```
#curl -o inotify_rsync.sh https://raw.githubusercontent.com/meetbill/op_practice_code/
master/Linux/service/inotify_rsync.sh
```

(3) 启动异地灾备程序

```
    #nohup /bin/bash /opt/inotify_rsync.sh &  // 后台不挂断地运行命令
    #echo "nohup /bin/bash /opt/inotify_rsync.sh &" >> /etc/rc.local // 设置 linux 服务
器启动自动启动 nohup
```

原理

1. 使用 inotifywait 监控文件系统时间变化
2. while 通过管道符接受内容，传给 read 命令
3. read 读取到内容，则执行 rsync 程序

常见问题

对大磁盘进行 **inotify** 监听时出错

```
Failed to watch /mnt/;upper limit on inotify watches reached!
Please increase the amount of inotify watches allowed per user via `/proc/sys/fs/inoti
fy/max_user_watches'.`
```

cat 一下这个文件，默认值是 8192，echo 8192000 > /proc/sys/fs/inotify/max_user_watches 即可~

telnet-server

安装使用

```
#curl -o telnet-server.tar.gz https://raw.githubusercontent.com/meetbill/op_practice_code/master/Linux/service/telnet-server.tar.gz
#tar -zxvf telnet-server.tar.gz
#cd telnet-server*
#sh start.sh
```

执行程序后有三项，执行第一项可以进行安装并启动 telnet-server，第二项会关闭 telnet-server 并将开机自动启动关闭

测试

需要测试 telnet 是否成功开启

```
#telnet localhost
```

输入用户名密码能登录成功。同时需要测试下其他机器远程 telnet 是否成功，如果不成功，那么很有可能是防火墙的问题

```
#iptables -I INPUT -p tcp --dport 23 -jACCEPT
#service iptables save
#service iptables restart
```

ftp

ftp 简介

ftp 工作会启动两个通道：控制通道 ， 数据通道。在 ftp 协议中，控制连接均是由客户端发起的，而数据连接有两种模式：port 模式（主动模式）和 pasv 模式（被动模式）

- **PORT** 模式： 在客户端需要接收数据时，ftp_client （大于 1024 的随机端口） —> PORT 命令 —> ftp_server (21) 发送 PORT 命令，这个 PORT 命令包含了客户端是用什么端口来接收数据（大于 1024 的随机端口），在传送数据时， ftp_server 将通过自己的 TCP 20 端口和 PORT 中包含的端口建立新的连接来传送数据。

- **PASV** 模式： 传送数据时，ftp_client —> PASV 命令 —> ftp_server(21) 发送 PASV 命令时，ftp_server 自动打开一个 1024--5000 之间的随机端口并且通知 ftp_client 在这个端口上传送数据，然后客户端向指定的端口发出请求连接，建立一条数据链路进行数据传输。

如果想对访问 FTP 的帐户给予更多的权限，可以用本地帐户来实现。但是，本地帐户默认情况下是可以登陆 Linux 系统的，这样对 Linux 系统来说是一个安全隐患。那么怎么能在灵活的赋予 FTP 用户权限的前提下，保证 FTP 服务器乃至整个 Linux 系统的安全呢？使用虚拟用户就是一种解决办法

安装包

> - vsftpd
> - db4*

安装配置

```
[root@meetbill ~]#curl -o ftptool.sh https://raw.githubusercontent.com/meetbill/op_practice_code/master/Linux/service/ftptool.sh
[root@meetbill ~]#chmod +x ftptool.sh
[root@meetbill ~]#./ftptool.sh install_server
[root@meetbill ~]#./ftptool.sh add_user
[root@meetbill ~]#./ftptool.sh start
```

常用问题处理

1 系统配置

1.1 Yum 安装安装包时提示证书过期

yum 安装安装包时提示"Peer's Certificate has expired"

https 的证书是有开始时间和失效时间的。因此本地时间要在这个证书的有效时间内。不过最好的方式，还是能够把时间进行同步。

```
# ntpdate pool.ntp.org
```

1.2 系统日志中的时间不准确

重启下 rsyslog 服务

```
/etc/init.d/rsyslog restart
```

1.3 Linux 系统日志出现 hung_task_timeout_secs 和 blocked for more than 120 seconds

问题现象

Linux 系统出现系统没有响应。 在 /var/log/message 日志中出现大量的类似如下错误信息：

```
echo 0 > /proc/sys/kernel/hung_task_timeout_secs disables this message.
blocked for more than 120 seconds
```

同时看监控时发现，服务器异常期间磁盘 io 比较高，cpu load 比较高

问题原因

默认情况下， Linux 会最多使用 40% 的可用内存作为文件系统缓存。当超过这个阈值后，文件系统会把将缓存中的内存全部写入磁盘， 导致后续的 IO 请求都是同步的。 将缓存写入磁盘时，有一个默认 120 秒的超时时间。 出现上面的问题的原因是 IO 子系统的处理速度不够快，不能在 120 秒将缓存中的数据全部写入磁盘。 IO 系统响应缓慢，导致越来越多的请求堆积，最终系统内存全部被占用，导致系统失去响应。

处理方法

根据应用程序情况，对 vm.dirty_ratio，vm.dirty_background_ratio 两个参数进行调优设置。例如，推荐如下设置：

```
# sysctl -w vm.dirty_ratio=10
# sysctl -w vm.dirty_background_ratio=5
# sysctl -p
```

如果系统永久生效，修改 /etc/sysctl.conf 文件。加入如下两行：

```
#vi /etc/sysctl.conf
vm.dirty_background_ratio = 5
vm.dirty_ratio = 10
```

重启系统生效。

内核参数解释

> - vm.dirty_background_ratio: 这个参数指定了当文件系统缓存脏页数量达到系统内存百分之多少时（如 5%）就会触发 pdflush/flush/kdmflush 等后台回写进程运行，将一定缓存的脏页异步地刷入外存；
> - vm.dirty_ratio: 而这个参数则指定了当文件系统缓存脏页数量达到系统内存百分之多少时（如 10%），系统不得不开始处理缓存脏页（因为此时脏页数量已经比较多，为了避免数据丢失需要将一定脏页刷入外存）；在此过程中很多应用进程可能会因为系统转而处理文件 IO 而阻塞。

一般情况下，dirty_ratio 的触发条件不会达到，因为每次会先达到 vm.dirty_background_ratio 的条件，然后触发 flush 进程进行异步的回写操作，但是这一过程中应用进程仍然可以进行写操作，如果应用进程写入的量大于 flush 进程刷出的量，就会达到 vm.dirty_ratio 这个参数所设定的坎，此时操作系统会转入同步地处理脏页的过程，阻塞应用进程。

2 磁盘

2.1 lvm 变为 inactive 状态

lvscan 查看 lvm 状态

```
[root@DB01 log]# lvscan
ACTIVE       '/dev/OraBack/backupone' [7.00 TB] inherit
ACTIVE       '/dev/OraBack/backuptwo' [7.00 TB] inherit
ACTIVE       '/dev/OraBack/backupthree' [1.00 TB] inherit
inactive     '/dev/OraBack【vg 名字】/orcl' [3.00 TB] inherit
```

激活 VG

```
[root@DB01 log]# vgchange -ay OraBack
4 logical volume(s) in volume group "OraBack" now active
```

lvscan 查看 lvm 状态

```
[root@DB01 log]# lvscan
ACTIVE            '/dev/OraBack/backupone' [7.00 TB] inherit
ACTIVE            '/dev/OraBack/backuptwo' [7.00 TB] inherit
ACTIVE            '/dev/OraBack/backupthree' [1.00 TB] inherit
ACTIVE            '/dev/OraBack/orcl' [3.00 TB] inherit
```

挂载

mount -a

数据库篇

- MySQL
- MongoDB
- Redis

Mysql

前言

MySQL 引擎

一种存储数据的技术。

并发控制

当多个连接对记录进行修改时保证数据的一致性和完整性。 可以理解为同步与互斥，原理和操作系统的那部分知识一致。

事务处理

这也是数据库区别于文件系统的一点。保证数据的完整性。

外键

保证数据的一致性。

索引

对数据表中一列或多列值进行排序的一种结构。是进行快速定位的方法。

对比

- MyISAM: 适用于事务处理不多时；
- InnoDB: 适用于事务处理比较多，且需要外键支持时；（锁颗粒为行锁，相对效率低）
- Memory：将所有数据保存在 RAM 中，在需要快速查找引用和其他类似数据的环境下，可提供极快的访问。

修改存储引擎，使用 `SHOW CREATE TABLE table_name;` 查看存储引擎类型。

- 在配置文件中 `-default-storage-engine = engine_name`
- 创建数据表时 `CREATE TABLE tbl_name() ENGINE = engine_name;`

特点	MyISAM	BDB	Memory	InnoDB	Archive
存储限制	没有	没有	有	64TB	没有
事务安全	-	支持	-	支持	-
锁机制	表锁	页锁	表锁	行锁	行锁
B 树索引	支持	支持	支持	支持	-
哈希索引	-	-	支持	支持	-
全文索引	支持	-	-	-	-
集群索引	-	-	-	支持	-
数据缓存	-	-	支持	支持	-
索引缓存	支持	-	支持	支持	-
数据可压缩	支持	-	-	-	支持
空间使用	低	低	N/A	高	非常低
内存使用	低	低	中等	高	低
批量插入的速度	高	高	高	低	非常高
支持外键	-	-	-	支持	-

查看下是否支持 InnoDB

通过命令行进入 mysql 或者是通过 phpmyadmin 进行操作，执行如下命令：

```
SHOW variables like 'have_%';
显示结果中会有如下 3 种可能的结果：
have_innodb YES
have_innodb NO
have_innodb DISABLED
这 3 种结果分别对应：
已经开启 InnoDB 引擎
未安装 InnoDB 引擎
未启用 InnoDB 引擎
```

针对第二种未安装，只需要安装即可；针对第三种未启用，则打开 MySQL 配置文件，找到 skip-innodb 项，将其改成#skip-innodb，之后重启 mysql 服务即可。

安装完 MySQL 后必须调整的 10 项配置

写在开始前

即使是经验老道的人也会犯错，会引起很多麻烦。所以在盲目的运用这些推荐之前，请记住下面的内容：

- 一次只改变一个设置！这是测试改变是否有益的唯一方法。
- 大多数配置能在运行时使用 SET GLOBAL 改变。这是非常便捷的方法它能使你在出问题后快速撤销变更。但是，要永久生效你需要在配置文件里做出改动。
- 一个变更即使重启了 MySQL 也没起作用？请确定你使用了正确的配置文件。请确定你把配置放在了正确的区域内（所有这篇文章提到的配置都属于 [mysqld])
- 服务器在改动一个配置后启不来了：请确定你使用了正确的单位。例如，`innodb_buffer_pool_size` 的单位是 MB 而 `max_connection` 是没有单位的。
- 不要在一个配置文件里出现重复的配置项。如果你想追踪改动，请使用版本控制。
- 不要用天真的计算方法，例如"现在我的服务器的内存是之前的 2 倍，所以我得把所有数值都改成之前的 2 倍"。

基本配置

你需要经常察看以下 3 个配置项。不然，可能很快就会出问题。

`innodb_buffer_pool_size`：这是你安装完 InnoDB 后第一个应该设置的选项。缓冲池是数据和索引缓存的地方：这个值越大越好，这能保证你在大多数的读取操作时使用的是内存而不是硬盘。典型的值是 5-6GB(8GB 内存），20-25GB(32GB 内存），100-120GB(128GB 内存）。

`innodb_log_file_size`：这是 redo 日志的大小。redo 日志被用于确保写操作快速而可靠并且在崩溃时恢复。一直到 MySQL 5.1，它都难于调整，因为一方面你想让它更大来提高性能，另一方面你想让它更小来使得崩溃后更快恢复。幸运的是从 MySQL 5.5 之后，崩溃恢复的性能的到了很大提升，这样你就可以同时拥有较高的写入性能和崩溃恢复性能了。一直到 MySQL 5.5，redo 日志的总尺寸被限定在 4GB（默认可以有 2 个 log 文件）。这在 MySQL 5.6 里被提高。

一开始就把 `innodb_log_file_size` 设置成 512M（这样有 1GB 的 redo 日志）会使你有充裕的写操作空间。如果你知道你的应用程序需要频繁的写入数据并且你使用的时 MySQL 5.6，你可以一开始就把它这是成 4G。

`max_connections`：如果你经常看到'Too many connections'错误，是因为 `max_connections` 的值太低了。这非常常见因为应用程序没有正确的关闭数据库连接，你需要比默认的 151 连接数更大的值。`max_connection` 值被设高了（例如 1000 或更高）之后一个主要缺陷是当服务器运行 1000 个或更高的活动事务时会变的没有响应。在应用程序里使用连接池或者在 MySQL 里使用 进程池 有助于解决这一问题。

InnoDB 配置

从 MySQL 5.5 版本开始，InnoDB 就是默认的存储引擎并且它比任何其他存储引擎的使用都要多得多。那也是为什么它需要小心配置的原因。

`innodb_file_per_table`：这项设置告知 InnoDB 是否需要将所有表的数据和索引存放在共享表空间里（`innodb_file_per_table` = OFF）或者为每张表的数据单独放在一个.ibd 文件（`innodb_file_per_table` = ON）。每张表一个文件允许你在 drop、truncate 或者 rebuild 表时回收磁盘空间。这对于一些高级特性也是有必要的，比如数据压缩。但是它不会带来任何性能收益。你不想让每张表一个文件的主要场景是：有非常多的表（比如 10k+）。

MySQL 5.6 中，这个属性默认值是 ON，因此大部分情况下你什么都不需要做。对于之前的版本你必须在加载数据之前将这个属性设置为 ON，因为它只对新创建的表有影响。

`innodb_flush_log_at_trx_commit`：默认值为 1，表示 InnoDB 完全支持 ACID 特性。当你的主要关注点是数据安全的时候这个值是最合适的，比如在一个主节点上。但是对于磁盘（读写）速度较慢的系统，它会带来很巨大的开销，因为每次将改变 flush 到 redo 日志都需要额外的 fsyncs。将它的值设置为 2 会导致不太可靠（unreliable）因为提交的事务仅仅每秒才 flush 一次到 redo 日志，但对于一些场景是可以接受的，比如对于主节点的备份节点这个值是可以接受的。如果值为 0 速度就更快了，但在系统崩溃时可能丢失一些数据：只适用于备份节点。

`innodb_flush_method`：这项配置决定了数据和日志写入硬盘的方式。一般来说，如果你有硬件 RAID 控制器，并且其独立缓存采用 write-back 机制，并有着电池断电保护，那么应该设置配置为 O_DIRECT；否则，大多数情况下应将其设为 fdatasync（默认值）。sysbench 是一个可以帮助你决定这个选项的好工具。

`innodb_log_buffer_size`：这项配置决定了为尚未执行的事务分配的缓存。其默认值（1MB）一般来说已经够用了，但是如果你的事务中包含有二进制大对象或者大文本字段的话，这点缓存很快就会被填满并触发额外的 I/O 操作。看看 `Innodb_log_waits` 状态变量，如果它不是 0，增加 `innodb_log_buffer_size`。

其他设置

`query_cache_size`：query cache（查询缓存）是一个众所周知的瓶颈，甚至在并发并不多的时候也是如此。 最佳选项是将其从一开始就停用，设置 `query_cache_size` = 0（现在 MySQL 5.6 的默认值）并利用其他方法加速查询：优化索引、增加拷贝分散负载或者启用额外的缓存（比如 memcache 或 redis）。如果你已经为你的应用启用了 query cache 并且还没有发现任何问题，query cache 可能对你有用。这是如果你想停用它，那就得小心了。

`log_bin`：如果你想让数据库服务器充当主节点的备份节点，那么开启二进制日志是必须的。如果这么做了之后，还别忘了设置`server_id`为一个唯一的值。就算只有一个服务器，如果你想做基于时间点的数据恢复，这（开启二进制日志）也是很有用的：从你最近的备份中恢复（全量备份），并应用二进制日志中的修改（增量备份）。二进制日志一旦创建就将永久保存。所以如果你不想让磁盘空间耗尽，你可以用 PURGE BINARY LOGS 来清除旧文件，或者设置`expire_logs_days`来指定过多少天日志将被自动清除。

记录二进制日志不是没有开销的，所以如果你在一个非主节点的复制节点上不需要它的话，那么建议关闭这个选项。

`skip_name_resolve`：当客户端连接数据库服务器时，服务器会进行主机名解析，并且当 DNS 很慢时，建立连接也会很慢。因此建议在启动服务器时关闭`skip_name_resolve`选项而不进行 DNS 查找。唯一的局限是之后`GRANT`语句中只能使用 IP 地址了，因此在添加这项设置到一个已有系统中必须格外小心。

总结

当然还有其他的设置可以起作用，取决于你的负载或硬件：在慢内存和快磁盘、高并发和写密集型负载情况下，你将需要特殊的调整。然而这里的目标是使得你可以快速地获得一个稳健的 MySQL 配置，而不用花费太多时间在调整一些无关紧要的 MySQL 设置或读文档找出哪些设置对你来说很重要上。

mysql 日志

MySQL 服务器上一共有六种日志：错误日志，查询日志，慢查询日志，二进制日志，事务日志，中继日志

慢日志 (5.1.73)

慢查询日志，顾名思义就是记录执行比较慢查询的日志

配置

两种方法

修改配置（永久生效）

修改配置文件：`/etc/my.cnf`

```
[mysqld]
slow_query_log = 1
slow_query_log_file = /tmp/mysql_slow_queries.log
long_query_time = 10
```

配置后重启 mysql

修改变量（重启后失效）

```
mysql> set global slow_query_log=1;
```

5.1.73 默认慢查询日志路径为：/var/run/mysqld/mysqld-slow.log

查看变量

慢查询打开状态 **(slow_query_log)** 和日志位置 **(slow_query_log_file)**

```
mysql> SHOW VARIABLES LIKE 'slow%';
```

```
+---------------------------+-----------------------------+
| Variable_name             | Value                       |
+---------------------------+-----------------------------+
| slow_launch_time          | 2                           |
| slow_query_log            | ON                          |
| slow_query_log_file       | /tmp/mysql_slow_queries.log|
+---------------------------+-----------------------------+
3 rows in set (0.01 sec)
```

慢查询时间

```
mysql> SHOW VARIABLES LIKE 'long%';
```

```
+---------------------------+-----------+
| Variable_name             | Value     |
+---------------------------+-----------+
| long_query_time           | 10.000000 |
+---------------------------+-----------+
```

测试

登陆 mysql

```
mysql>select sleep(11);
```

查看日志

tail -f /tmp/mysql_slow_queries.log

日志查询

列出记录次数最多的 10 个 sql 语句

```
mysqldumpslow -s c -t 10 /tmp/mysql_slow_queries.log
```

得到返回记录最多的 10 个 SQL。

```
mysqldumpslow -s r -t 10 /tmp/mysql_slow_queries.log
```

清理 MySQL binlog

概要

作为 master 的 Mysql 运行久了以后会在根目录中产生大量的 binlog 日志，如果不及时清理，会占用大量的磁盘空间，也会对数据库的正常运行带来隐患

> 之所以要开启 binlog 是因为，mysql 的主备复制是建立在 master 产生 binlog 的基础上

相关基本参数

--log-bin[=base_name]

Item	Format
Command-Line Format	--log-bin
Option-File Format	log-bin
System Variable Name	log_bin
Variable Scope	Global
Dynamic Variable	No
Permitted Values Type	file name

> Enable binary logging. The server logs all statements that change data to the binary log, which is used for backup and replication.The option value, if given, is the basename for the log sequence. The server creates binary log files in sequence by adding a numeric suffix to the basename. It is recommended that you specify a basename (see Section C.5.8, “Known Issues in MySQL”, for the reason). Otherwise, MySQL uses host_name-bin as the basename.

这是手册中关于 binlog 作用的两点描述

> - For replication, the binary log on a master replication server provides a record of the data changes to be sent to slave servers. The master server sends the events contained in its binary log to its slaves, which execute those events to make the same data changes that were made on the master.
> - Certain data recovery operations require use of the binary log. After a backup has been restored, the events in the binary log that were recorded after the backup was made are re-executed. These events bring databases up to date from the point of the backup

max_binlog_size

Item	Format
Command-Line Format	--max_binlog_size=#
Option-File Format	max_binlog_size
System Variable Name	max_binlog_size
Variable Scope	Global
Dynamic Variable	Yes
Permitted Values Type	numeric
Default	1073741824
Range	4096 .. 1073741824

> If a write to the binary log causes the current log file size to exceed the value of this variable, the server rotates the binary logs (closes the current file and opens the next one). The minimum value is 4096bytes. The maximum and default value is 1GB.
>
> **Note:** If max_relay_log_size is 0, the value of max_binlog_size applies to relay logs as well.

--log-bin-index[=file_name]

Item	Format
Command-Line Format	--log-bin-index=name
Option-File Format	log-bin-index
System Variable Name	log_bin
Variable Scope	Global
Dynamic Variable	No
Permitted Values Type	file name

> The index file for binary log file names. If you omit the file name,and if you did not specify one with --log-bin, MySQL uses host_name-bin.index as the file name.

expire_logs_days

Item	Format
Command-Line Format	--expire_logs_days=#
Option-File Format	expire_logs_days
System Variable Name	expire_logs_days
Variable Scope	Global
Dynamic Variable	Yes
Permitted Values Type	numeric
Default	0
Range	0 .. 99

> The number of days for automatic binary log file removal. The default is 0, which means “no automatic removal.” Possible removals happen at startup and when the binary log is flushed. Log flushing occurs as indicated in Section 5.2, “MySQL Server Logs”.

清理方法

手动清理

使用 ***PURGE BINARY LOGS*** 进行清理

```
mysql> help purge
Name: 'PURGE BINARY LOGS'
Description:
Syntax:
PURGE { BINARY | MASTER } LOGS
    { TO 'log_name' | BEFORE datetime_expr }

The binary log is a set of files that contain information about data
modifications made by the MySQL server. The log consists of a set of
binary log files, plus an index file (see
http://dev.mysql.com/doc/refman/5.1/en/binary-log.html).

The PURGE BINARY LOGS statement deletes all the binary log files listed
in the log index file prior to the specified log file name or date.
BINARY and MASTER are synonyms. Deleted log files also are removed from
the list recorded in the index file, so that the given log file becomes
the first in the list.

This statement has no effect if the server was not started with the
--log-bin option to enable binary logging.

URL: http://dev.mysql.com/doc/refman/5.1/en/purge-binary-logs.html

Examples:
PURGE BINARY LOGS TO 'mysql-bin.010';
PURGE BINARY LOGS BEFORE '2008-04-02 22:46:26';

mysql>
```

当 slave 正在复制时，这条命令也是安全的，如果尝试删除一个正在被读的日志文件，这个语句将什么事情也不做。

但是如果一个 slave 没有连接上 master，而在 master 上删除了它还没读取的日志文件，一旦 salve 连接上 master，slave 将出错，无法正常复制。

尽量遵循下面的流程，以确保安全删除日志文件：

- 1. 在各个 slave 服务器上，使用**SHOW SLAVE STATUS**检查哪一个日志文件正在被读取
- 1. 使用**SHOW BINARY LOGS**在 master 上获取一份日志文件列表
- 1. 根据所有的 slaves 服务器，决定最早的那个日志文件是哪一个，这个就是目标文件，如果所有的 slaves 都更新完所有操作，这个日志文件就是列表中的最后一个
- 1. 对所有即将删除的日志文件进行备份（这不是必要步骤，但是建议这么做）

- 1. 使用**Purge**清理掉到目标日志的所有日志文件

> **Note:** 当**.index**中列出的文件在系统中已经被各种原因移除（比如使用**rm**手动删除了）的情况下使用 **PURGE BINARY LOGS TO** 或 **PURGE BINARY LOGS BEFORE**会报错，解决办法是手动编辑**.index**文件，确保里面列出的文件在系统中真实存在，然后再次使用**PURGE BINARY LOGS**

检查当前系统中的日志文件

```
mysql> show binary logs;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000001 |       125 |
| mysql-bin.000002 |       125 |
| mysql-bin.000003 |       125 |
| mysql-bin.000004 |       125 |
| mysql-bin.000005 |       125 |
| mysql-bin.000006 |       125 |
| mysql-bin.000007 |       125 |
| mysql-bin.000008 |       125 |
| mysql-bin.000009 |       125 |
| mysql-bin.000010 |       125 |
| mysql-bin.000011 |       125 |
| mysql-bin.000012 |       125 |
| mysql-bin.000013 |       125 |
| mysql-bin.000014 |       125 |
| mysql-bin.000015 |       125 |
| mysql-bin.000016 |       106 |
+------------------+-----------+
16 rows in set (0.00 sec)

mysql> \! ls -l /var/lib/mysql
total 20648
-rw-rw----. 1 mysql mysql 10485760 Mar 18 17:32 ibdata1
-rw-rw----. 1 mysql mysql  5242880 Mar 18 17:32 ib_logfile0
-rw-rw----. 1 mysql mysql  5242880 Mar 18 17:32 ib_logfile1
-rw-r-----. 1 mysql root     27513 Mar 19 00:27 localhost.localdomain.err
-rw-r-----. 1 mysql root     36471 Apr  1 17:07 m2.err
-rw-rw----. 1 mysql mysql        5 Apr  1 17:07 m2.pid
-rw-rw----. 1 mysql mysql      125 Mar 31 22:27 m2-relay-bin.000001
-rw-rw----. 1 mysql mysql      106 Apr  1 17:07 m2-relay-bin.000002
-rw-rw----. 1 mysql mysql       44 Apr  1 17:07 m2-relay-bin.index
-rw-rw----. 1 mysql mysql       68 Apr  1 17:07 master.info
drwx------. 2 mysql mysql     4096 Mar 31 19:28 mysql
-rw-rw----. 1 mysql mysql      125 Mar 18 23:42 mysql-bin.000001
-rw-rw----. 1 mysql mysql      125 Mar 19 00:27 mysql-bin.000002
-rw-rw----. 1 mysql mysql      125 Mar 19 11:37 mysql-bin.000003
-rw-rw----. 1 mysql mysql      125 Mar 19 15:03 mysql-bin.000004
-rw-rw----. 1 mysql mysql      125 Mar 19 15:29 mysql-bin.000005
```

```
-rw-rw----. 1 mysql mysql      125 Mar 19 15:56 mysql-bin.000006
-rw-rw----. 1 mysql mysql      125 Mar 19 16:45 mysql-bin.000007
-rw-rw----. 1 mysql mysql      125 Mar 19 17:27 mysql-bin.000008
-rw-rw----. 1 mysql mysql      125 Mar 19 17:56 mysql-bin.000009
-rw-rw----. 1 mysql mysql      125 Mar 19 19:06 mysql-bin.000010
-rw-rw----. 1 mysql mysql      125 Mar 19 20:11 mysql-bin.000011
-rw-rw----. 1 mysql mysql      125 Mar 20 02:33 mysql-bin.000012
-rw-rw----. 1 mysql mysql      125 Mar 28 02:36 mysql-bin.000013
-rw-rw----. 1 mysql mysql      125 Mar 28 06:08 mysql-bin.000014
-rw-rw----. 1 mysql mysql      125 Mar 31 22:27 mysql-bin.000015
-rw-rw----. 1 mysql mysql      106 Apr  1 17:07 mysql-bin.000016
-rw-rw----. 1 mysql mysql      304 Apr  1 17:07 mysql-bin.index
srwxrwxrwx. 1 mysql mysql        0 Apr  1 17:07 mysql.sock
-rw-rw----. 1 mysql mysql       60 Apr  1 17:07 relay-log.info
drwx------. 2 mysql mysql     4096 Mar 18 17:32 test
mysql>
```

删除掉**mysql-bin.000004**之前的日志

```
mysql> purge binary logs to 'mysql-bin.000004';
Query OK, 0 rows affected (0.00 sec)

mysql> show binary logs;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000004 |       125 |
| mysql-bin.000005 |       125 |
| mysql-bin.000006 |       125 |
| mysql-bin.000007 |       125 |
| mysql-bin.000008 |       125 |
| mysql-bin.000009 |       125 |
| mysql-bin.000010 |       125 |
| mysql-bin.000011 |       125 |
| mysql-bin.000012 |       125 |
| mysql-bin.000013 |       125 |
| mysql-bin.000014 |       125 |
| mysql-bin.000015 |       125 |
| mysql-bin.000016 |       106 |
+------------------+-----------+
13 rows in set (0.00 sec)

mysql> \! ls -l /var/lib/mysql
total 20636
-rw-rw----. 1 mysql mysql 10485760 Mar 18 17:32 ibdata1
-rw-rw----. 1 mysql mysql  5242880 Mar 18 17:32 ib_logfile0
-rw-rw----. 1 mysql mysql  5242880 Mar 18 17:32 ib_logfile1
-rw-r-----. 1 mysql root     27513 Mar 19 00:27 localhost.localdomain.err
-rw-r-----. 1 mysql root     36471 Apr  1 17:07 m2.err
-rw-rw----. 1 mysql mysql        5 Apr  1 17:07 m2.pid
-rw-rw----. 1 mysql mysql      125 Mar 31 22:27 m2-relay-bin.000001
```

```
-rw-rw----. 1 mysql mysql      106 Apr  1 17:07 m2-relay-bin.000002
-rw-rw----. 1 mysql mysql       44 Apr  1 17:07 m2-relay-bin.index
-rw-rw----. 1 mysql mysql       68 Apr  1 17:07 master.info
drwx------. 2 mysql mysql     4096 Mar 31 19:28 mysql
-rw-rw----. 1 mysql mysql      125 Mar 19 15:03 mysql-bin.000004
-rw-rw----. 1 mysql mysql      125 Mar 19 15:29 mysql-bin.000005
-rw-rw----. 1 mysql mysql      125 Mar 19 15:56 mysql-bin.000006
-rw-rw----. 1 mysql mysql      125 Mar 19 16:45 mysql-bin.000007
-rw-rw----. 1 mysql mysql      125 Mar 19 17:27 mysql-bin.000008
-rw-rw----. 1 mysql mysql      125 Mar 19 17:56 mysql-bin.000009
-rw-rw----. 1 mysql mysql      125 Mar 19 19:06 mysql-bin.000010
-rw-rw----. 1 mysql mysql      125 Mar 19 20:11 mysql-bin.000011
-rw-rw----. 1 mysql mysql      125 Mar 20 02:33 mysql-bin.000012
-rw-rw----. 1 mysql mysql      125 Mar 28 02:36 mysql-bin.000013
-rw-rw----. 1 mysql mysql      125 Mar 28 06:08 mysql-bin.000014
-rw-rw----. 1 mysql mysql      125 Mar 31 22:27 mysql-bin.000015
-rw-rw----. 1 mysql mysql      106 Apr  1 17:07 mysql-bin.000016
-rw-rw----. 1 mysql mysql      247 Apr  1 19:47 mysql-bin.index
srwxrwxrwx. 1 mysql mysql        0 Apr  1 17:07 mysql.sock
-rw-rw----. 1 mysql mysql       60 Apr  1 17:07 relay-log.info
drwx------. 2 mysql mysql     4096 Mar 18 17:32 test
mysql>
```

查看 binlog 事件

```
mysql> show binlog events\G
*************************** 1. row ***************************
   Log_name: mysql-bin.000004
        Pos: 4
 Event_type: Format_desc
  Server_id: 2
End_log_pos: 106
       Info: Server ver: 5.1.73-14.12-log, Binlog ver: 4
*************************** 2. row ***************************
   Log_name: mysql-bin.000004
        Pos: 106
 Event_type: Stop
  Server_id: 2
End_log_pos: 125
       Info:
2 rows in set (0.03 sec)

mysql>
```

根据时间来清理

```
mysql> purge master logs before '2015-03-19 15:56:00'
Query OK, 0 rows affected (0.01 sec)

mysql> show binary logs;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000006 |       125 |
| mysql-bin.000007 |       125 |
| mysql-bin.000008 |       125 |
| mysql-bin.000009 |       125 |
| mysql-bin.000010 |       125 |
| mysql-bin.000011 |       125 |
| mysql-bin.000012 |       125 |
| mysql-bin.000013 |       125 |
| mysql-bin.000014 |       125 |
| mysql-bin.000015 |       125 |
| mysql-bin.000016 |       106 |
+------------------+-----------+
11 rows in set (0.00 sec)

mysql>
```

清理 5 天之前的日志

```
mysql> select now();
+---------------------+
| now()               |
+---------------------+
| 2015-04-02 13:52:13 |
+---------------------+
1 row in set (0.00 sec)

mysql> select date_sub(now(),interval 5 day);
+--------------------------------+
| date_sub(now(),interval 5 day) |
+--------------------------------+
| 2015-03-28 13:52:15            |
+--------------------------------+
1 row in set (0.00 sec)

mysql> purge master logs before date_sub(now(),interval 5 day);
Query OK, 0 rows affected (0.01 sec)

mysql> show binary logs;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000015 |       125 |
| mysql-bin.000016 |       125 |
| mysql-bin.000017 |       106 |
+------------------+-----------+
3 rows in set (0.00 sec)

mysql> \! ls -l *mysql-bin*
-rw-rw----. 1 mysql mysql 125 Mar 31 22:27 mysql-bin.000015
-rw-rw----. 1 mysql mysql 125 Apr  1 22:56 mysql-bin.000016
-rw-rw----. 1 mysql mysql 106 Apr  2 10:35 mysql-bin.000017
-rw-rw----. 1 mysql mysql  57 Apr  2 13:52 mysql-bin.index
mysql> \! cat mysql-bin.index
./mysql-bin.000015
./mysql-bin.000016
./mysql-bin.000017
mysql>
```

使用 ***RESET MASTER*** 进行清理

RESET MASTER会删除 index 文件中所有的 binary log，重新设置 binary log 为空，创建新的 binary log，这条语句适合在第一次 master 运行启动后，不太适合在生产环境中已经运行了好久的情况下。

> **Note: RESET MASTER和PURGE BINARY LOGS有以下两点不同**
>
> - 1.**RESET MASTER**会移除掉 index 文件中的所有日志，然后只留下一个空的以**.000001**结尾的日志文件，但是**PURGE BINARY LOGS**不会重设后缀
> - 2.**RESET MASTER**不是设计来用在有任何 slave 正在运行的情况下，但**PURGE BINARY LOGS**是设计来在此种情况下使用的，当 slave 的复制正在运行时使用**PURGE BINARY LOGS**也是安全的

RESET MASTER在第一次设置 master 和 slave 的情况下非常有用，可以按照下面的步骤进行检查：

- 1. 运行 master 和 slave, 开启复制
- 1. 在 master 中进行一系列的插入测试
- 1. 在 slave 中检查更新有无按预期同步
- 1. 在 salve 上确认复制可以正常运行后，**RESET SLAVE** 然后 **STOP SLAVE** ，然后确保所有不想要的数据在 slave 上已经清理
- 1. 在 master 上执行**RESET MASTER**清除掉测试数据
- 1. 在检查所有的不想要的测试数据和日志已经清理掉后，可以在 slave 上重新开启复制

自动清理

可以使用**expire_logs_days**系统变量来设定日志过期时间，自动删除过期日志，如果环境中有复制，注意要设定合适的值，这个值要大于最坏情况下 slave 可能落后于 master 的天数。

> 自动清理操作会发生在系统开启和日志刷新的时候

设定 5 天为日志过期时间

```
mysql> show variables like "%expire%";
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| expire_logs_days | 0     |
+------------------+-------+
1 row in set (0.00 sec)

mysql> set global expire_logs_days=5 ;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like "%expire%";
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| expire_logs_days | 5     |
+------------------+-------+
1 row in set (0.00 sec)

mysql>
```

mysql 管理

用户管理

创建用户

CREATE USER 'username'@'host' IDENTIFIED BY 'password';

用户创建之后只能看到 `information_schema` 数据库，使用 `show grants;` 看到自己的权限为 `GRANT USAGE ON *.* TO 'user'@'localhost'` ;

为用户授权

使用***GRANT*** 命令为用户授予数据库操作的权力，比如增删改查等。

```
> GRANT ALL PRIVILEGES ON *.* TO 'myuser'@'%' IDENTIFIED BY 'mypassword' WITH GRANT OP
TION;
> FLUSH   PRIVILEGES;
```

其中 `%` 表示对所有主机开放，只对本机开放时可以写 `127.0.0.1`

修改用户密码

SET PASSWORD（推荐）,MySQL5.7.6 及以后：

```
SET PASSWORD [FOR user] = password_option
password_option: {
    PASSWORD('auth_string') # 弃用
  | 'auth_string'
}
# 修改自己的密码
SET PASSWORD = '123456';
# 修改一用户的密码
SET PASSWORD FOR 'user'@'34.23.44.32' = '123456';
```

ALTER（推荐）

```
ALTER USER 'user'@'localhost' IDENTIFIED BY 'auth_string';
```

忘记 root 密码

```
# 正确关闭 MySQL 服务
$ mysqld stop
# 开启不验证登录
$ mysqld_safe --skip-grant-tables &
# 用上述其他方法修改密码
# 退出重启 MySQL

# 方法二：
1. 在 /etc/mysql/my.cnf 或者 /etc/my.cnf 中 [mysqld] 加入 `skip-grant-tales` 可以免除密码登陆。
2. `service mysqld start`启动 mysql 后，直接在终端输入`mysql`可以免密码直接登陆。
3. > flush privileges;
4. set password for root@localhost = 'new-pass';

在 MySQL 5.7 之后，Windows 使用解压缩安装 MySQL, 需要依次执行如下方便初始化：
mysqld --initialize-insecure # 删除其他所有用户，无密码初始化 MySQL root 用户，如果没有 insecure, 则会创建随机密码
mysqld -install # 安装服务
之后再不输入密码登录 root 用户，再设置密码。
```

mysqladmin（不安全）, 在终端输入：

```
$ mysqladmin -u username -p password '123456'
```

UPDATE user 表

```
UPDATE user SET password=password('123456') WHERE user='name' AND host='12.44.33.22';
FLUSH privileges;
```

mysql 主从

清除主从关系

一般情况下清除主从关系只需要做以下操作即可

- 登陆 slave 数据库后执行 `stop slave`
- 登陆 slave 数据库后执行 `RESET SLAVE`
- 重启 slave 的 mysql 服务

阅读下方内容了解更多清除主从关系的相关操作

RESET MASTER

删除所有 index file 中记录的所有 binlog 文件，将日志索引文件清空，创建一个新的日志文件，这个命令通常仅仅用于第一次用于搭建主从关系的时的主库，

注意

reset master 不同于 purge binary log 的两处地方

> - reset master 将删除日志索引文件中记录的所有 binlog 文件，创建一个新的日志文件 起始值从 000001 开始，然而 purge binary log 命令并不会修改记录 binlog 的顺序的数值
> - reset master 不能用于有任何 slave 正在运行的主从关系的主库。因为在 slave 运行时刻 reset master 命令不被支持，reset master 将 master 的 binlog 从 000001 开始记录，slave 记录的 master log 则是 reset master 时主库的最新的 binlog, 从库会报错无法找的指定的 binlog 文件。

RESET SLAVE

```
使用 reset slave 之前必须使用 stop slave 命令将复制进程停止。

reset slave 将使 slave 忘记主从复制关系的位置信息。该语句将被用于干净的启动，它删除 master.info 文件和 relay-log.info 文件以及所有的 relay log 文件并重新启用一个新的 relaylog 文件。

在 5.1.73（后面的版本貌似也是） 的版本中 reset slave 并不会清理存储于内存中的复制信息比如  master host, master port, master user, or master password, 也就是说如果没有使用 change master 命令做重新定向，执行 start slave 还是会指向旧的 master 上面。当从库执行 reset slave 之后，将 mysqld 重启后复制参数将被重置。

注 所有的 relay log 将被删除不管他们是否被 SQL thread 进程完全应用（这种情况发生于备库延迟以及在备库执行了 stop slave 命令）, 存储复制链接信息的 master.info 文件将被立即清除，如果 SQL thread 正在复制临时表的过程中，执行了 stop slave ，并且执行了 reset slave，这些被复制的临时表将被删除。
```

RESET SLAVE ALL(5.6 后支持）

在 5.6.3 版本以及以后 使用使用 RESET SLAVE ALL 来完全的清理复制连接参数信息。

mysql 工具

- mysql 日常定时备份

mysql 运维

MySQL 忘记密码

现象

```
#mysql -u root -p
#就会出现：ERROR 1045 (28000): Access denied for user ''@'localhost' (using password: NO
)
```

解决方法

```
l 关闭 mysql
# service mysqld stop

2. 屏蔽权限
# mysqld_safe --skip-grant-table
屏幕出现：
161028 22:30:08 mysqld_safe Logging to '/var/log/mysqld.log'.
161028 22:30:08 mysqld_safe Starting mysqld daemon with databases from /var/lib/mysql'

3. 新开起一个终端输入
#mysql -u root
mysql> UPDATE user SET Password=PASSWORD('newpassword') where USER='root';
mysql> FLUSH PRIVILEGES;// 记得要这句话，否则如果关闭先前的终端，又会出现原来的错误
mysql> \q
#/etc/init.d/mysqld restart
```

mysql.sock 位置错误

日常迁移完数据库的存储路径后，client 登陆失败

解决方法

修改 /etc/my.cnf 配置文件

```
[client]
socket = /tmp/mysql.sock
```

Mongodb

Mongodb 备份

Mongodb 用的是可以热备份的 mongodump 和对应恢复的 mongorestore, 在 linux 下面使用 shell 脚本写的定时备份，代码如下

1. 定时备份

```
    #!/bin/bash
    sourcepath='/usr'/bin   #mongodump 命令所在路径
    targetpath='/var/lib/mongo/mongobak'   #备份存放位置
    nowtime=$(date +%Y%m%d)

    start()
    {
      ${sourcepath}/mongodump -u username -p password -d dbname --host 127.0.0.1 --por
t 27017 --out ${targetpath}/${nowtime}
    }
    execute()
    {
      start
      if [ $? -eq 0 ]
      then
        echo "back successfully!"
      else
        echo "back failure!"
      fi
    }

    if [ ! -d "${targetpath}/${nowtime}/" ]
    then
     mkdir ${targetpath}/${nowtime}
    fi
    execute
    echo "============== back end ${nowtime} =============="
```

1. 定时清除，保留 7 天的纪录

```
#!/bin/bash
targetpath='/var/lib/mongo/mongobak'
nowtime=$(date -d '-7 days' "+%Y%m%d")
if [ -d "${targetpath}/${nowtime}/" ]
then
  rm -rf "${targetpath}/${nowtime}/"
  echo "=======${targetpath}/${nowtime}/=== 删除完毕 =="
fi
echo "===$nowtime ==="
```

1. 服务器的时间要同步，同步的方法

```
微软公司授时主机（美国）      time.windows.com
台警大授时中心（台湾）         asia.pool.ntp.org
中科院授时中心（西安）         210.72.145.44
网通授时中心（北京）            219.158.14.130
调用同步：  ntpdate asia.pool.ntp.org
```

1. 设置上面脚本权限和定时任务

   权限：chmod +x filename 定时任务：crontab -e

```
10 04 * * * /shell/mongobak.sh 1>/var/log/crontab_mongo_back.log &
10 02 * * * /shell/mongobakdelete.sh 1>/var/log/crontab_mongo_delete.log &
```

每天凌晨 4 点 10 开始进行备份，2 点 10 分删除旧的备份

1 Redis

1.1 持久化

1.1.1 AOF 重写机制

AOF 重写触发条件

AOF 重写可以由用户通过调用 BGREWRITEAOF 手动触发。服务器在 AOF 功能开启的情况下，会维持以下三个变量：

> - 记录当前 AOF 文件大小的变量 aof_current_size。
> - 记录最后一次 AOF 重写之后，AOF 文件大小的变量 aof_rewrite_base_size。
> - 增长百分比变量 aof_rewrite_perc。

每次当 serverCron（服务器周期性操作函数，在 src/redis.c 中）函数执行时，它会检查以下条件是否全部满足，如果全部满足的话，就触发自动的 AOF 重写操作：

> - 没有 BGSAVE 命令（RDB 持久化）/AOF 持久化在执行；
> - 没有 BGREWRITEAOF 在进行；
> - auto-aof-rewrite-percentage 参数不为 0
> - 当前 AOF 文件大小要大于 server.aof_rewrite_min_size（默认为 1MB）
> - 当前 AOF 文件大小和最后一次重写后的大小之间的比率等于或者等于指定的增长百分比（在配置文件设置了 auto-aof-rewrite-percentage 参数，不设置默认为 100%）

如果前面四个条件都满足，并且当前 AOF 文件大小比最后一次 AOF 重写时的大小要大于指定的百分比，那么触发自动 AOF 重写。

源码如下：

```
 /* Trigger an AOF rewrite if needed */
        // 触发 BGREWRITEAOF
         if (server.rdb_child_pid == -1 &&
             server.aof_child_pid == -1 &&
             server.aof_rewrite_perc &&
             // AOF 文件的当前大小大于执行 BGREWRITEAOF 所需的最小大小
             server.aof_current_size > server.aof_rewrite_min_size)
         {
            // 上一次完成 AOF 写入之后，AOF 文件的大小
            long long base = server.aof_rewrite_base_size ?
                            server.aof_rewrite_base_size : 1;

            // AOF 文件当前的体积相对于 base 的体积的百分比
            long long growth = (server.aof_current_size*100/base) - 100;

            // 如果增长体积的百分比超过了 growth ，那么执行 BGREWRITEAOF
            if (growth >= server.aof_rewrite_perc) {
                redisLog(REDIS_NOTICE,"Starting automatic rewriting of AOF on %lld%% g
rowth",growth);
                // 执行 BGREWRITEAOF
                rewriteAppendOnlyFileBackground();
            }
         }
```

1.2 主从同步

主从同步相关参数

> - repl-backlog-size: 增量重传 buf
> - repl-timeout: 主动超时
> - client-output-buffer-limit（和写入量有关）
>   - 这个参数分为 3 部分，第二部分涉及 slave
>   - slave 部分默认值：256M 64M 60 秒
>   - output-buffer 缓冲区里放的是主库待同步给从库的操作数据。
>   - 如果 output-buffer>256M 则从节点需要重新全同步，如果 256>output-buffer>64 且持续时间 60 秒，则从节点需要重新全同步。

主从同步

分别启动 master 和 slave 后，会自动启动同步 slave 出现如下类似日志，则同步已完成：

```
[4611] 24 Aug 19:11:46.843 * MASTER <-> SLAVE sync started
[4611] 24 Aug 19:11:46.844 * Non blocking connect for SYNC fired the event.
[4611] 24 Aug 19:11:46.844 * Master replied to PING, replication can continue...
[4611] 24 Aug 19:11:46.844 * Partial resynchronization not possible (no cached master)
[4611] 24 Aug 19:11:46.844 * Full resync from master: 0629e2e6e79c13c21ff38b638b600918
3140939a:1
[4611] 24 Aug 19:13:55.662 * MASTER <-> SLAVE sync: receiving 5774276835 bytes from ma
ster
[4611] 24 Aug 19:14:45.578 * MASTER <-> SLAVE sync: Flushing old data
[4611] 24 Aug 19:16:57.509 * MASTER <-> SLAVE sync: Loading DB in memory
[4611] 24 Aug 19:19:44.191 * MASTER <-> SLAVE sync: Finished with success
```

repl-timeout

若 slave 日志出现如下行：

```
# Timeout receiving bulk data from MASTER... If the problem persists try to set the
 'repl-timeout' parameter in redis.conf to a larger value.
```

调整 slave 的 redis.conf 参数：

```
repl-timeout 60  # 将数值设得更大
如：config set repl-timeout 600
```

写入量太大超出 output-buffer

若 slave 日志出现如下行：

```
# I/O error reading bulk count from MASTER: Resource temporarily unavailable
# I/O error trying to sync with MASTER: connection lost
```

调整 master 分配给 slave client buffer：

```
client-output-buffer-limit slave 256mb 64mb 60
# 256mb 是一个硬性限制，当 output-buffer 的大小大于 256mb 之后就会断开连接
# 64mb 60 是一个软限制，当 output-buffer 的大小大于 64mb 并且超过了 60 秒的时候就会断开连接
# 或者全部设为 0，取消限制。

如：config set client-output-buffer-limit "slave 0 0 0"
```

repl-backlog-size 太小导致失败

当 master-slave 复制连接断开，server 端会释放连接相关的数据结构。replication buffer 中的数据也就丢失，当断开的 slave 重新连接上 master 的时候，slave 将会发送 psync 命令（包含复制的偏移量 offset），请求 partial resync。如果请求的 offset 不存在，那么执行全量的 sync 操作，相当于重新建立主从复制。

```
Unable to partial resync with slave $slaveip:6379 for lack of backlog (Slave request w
as: 5974421660).
```

调整 repl-backlog-size 大小

1.3 Redis bug

1.3.1 AOF 句柄泄露 bug

表现

日志中提示

```
* Residual parent diff successfully flushed to the rewritten AOF (329.83 MB)
* Background AOF rewrite finished successfully
* Starting automatic rewriting of AOF on 100% growth
# Can't rewrite append only file in background: fork: Cannot allocate memory
* Starting automatic rewriting of AOF on 100% growth
# Can't rewrite append only file in background: fork: Cannot allocate memory
* Starting automatic rewriting of AOF on 100% growth
# Can't rewrite append only file in background: fork: Cannot allocate memory
* Starting automatic rewriting of AOF on 100% growth
# Error opening /setting AOF rewrite IPC pipes: Numerical result out of range
* Starting automatic rewriting of AOF on 100% growth
# Error opening /setting AOF rewrite IPC pipes: Numerical result out of range
# Error registering fd event for the new client: Numerical result out of range (fd=101
28)
# Error registering fd event for the new client: Numerical result out of range (fd=101
28)
```

使用 lsof 命令检查 fd 数，发现当时进程打开的 fd 数已经达到 10128 个，而其中大部分基本都是 pipe. 在 Redis 中，pipe 主要用于父子进程间通信，如 AOF 重写、基于 socket 的 RDB 持久化等场景。

分析

fd 限制

首先，我们定位到 client 连接报错的主要调用链为 networking.c/acceptCommonHandler => networking.c/createClient => ae.c/aeCreateFileEvent：

```
static void acceptCommonHandler(int fd, int flags, char *ip) {
    client *c;
    if ((c = createClient(fd)) == NULL) {
        serverLog(LL_WARNING,
            "Error registering fd event for the new client: %s (fd=%d)",
            strerror(errno),fd);
        close(fd); /* May be already closed, just ignore errors */
        return;
    }
    //……
}
int aeCreateFileEvent(aeEventLoop *eventLoop, int fd, int mask,
        aeFileProc *proc, void *clientData)
{
    if (fd >= eventLoop->setsize) {
        errno = ERANGE;
        return AE_ERR;
    }
//……
}
```

而 eventLoop->setsize 则是在 server.c/initServer 中被初始化和设置的，大小为 maxclient+128 个。而我们 maxclient 采用 Redis 默认配置 10000 个，所以当 fd=10128 时就出错了。

```
server.el = aeCreateEventLoop(server.maxclients+CONFIG_FDSET_INCR);
```

aof 重写子进程启动失败为何不关闭 pipe

aof 重写过程由 server.c/serverCron 定时时间事件处理函数触发，调用 aof.c/rewriteAppendOnlyFileBackground 启动 aof 重写子进程。在 rewriteAppendOnlyFileBackground 方法中我们注意到如果 fork 失败，过程就直接退出了。

```
int rewriteAppendOnlyFileBackground(void) {
    //……
    if (aofCreatePipes() != C_OK) return C_ERR; // 创建 pipe
    //……
    if ((childpid = fork()) == 0) {
        /* Child */
        //……
    } else {
        /* Parent */
        // 子进程启动出错处理
        if (childpid == -1) {
            serverLog(LL_WARNING,
                "Can't rewrite append only file in background: fork: %s",
                strerror(errno)); // 最初内存不足正是这里打出的错误 log
            return C_ERR;
        }
    //……
    }
}
```

而关闭 pipe 的方法，是在 server.c/serverCron => aof.c/backgroundRewriteDoneHandler 中发现 AOF 重写子进程退出后被调用：

```
//……
    /* Check if a background saving or AOF rewrite in progress terminated. */
    if (server.rdb_child_pid != -1 || server.aof_child_pid != -1 ||
        ldbPendingChildren())
    {
        //……
        // 任意子进程退出时执行
        if ((pid = wait3(&statloc,WNOHANG,NULL)) != 0) {
            //……
            if (pid == -1) {
                serverLog(……);
            } else if (pid == server.rdb_child_pid) {
                backgroundSaveDoneHandler(exitcode,bysignal);
            } else if (pid == server.aof_child_pid) { // 发现是 aof 重写子进程完成
                backgroundRewriteDoneHandler(exitcode,bysignal); // 执行后续工作，包括关闭
 pipe
            }
            //……
        }
    }
//……
```

由此可见，如果 aof 重写子进程没有启动，则 pipe 将不会被关闭。而下次尝试启动 aof 重写时，又会调用 aof.c/aofCreatePipes 创建新的 pipe。

解决

> - 2015 年就被两次在社区上报（参考 https://github.com/antirez/redis/issues/2857
> - 2016 年有开发者提交代码修复此问题，直至 2017 年 2 月相关修复才被合入主干（参考 https://github.com/antirez/redis/pull/3408）
> - 这只长寿的 bug 在 3.2.9 版本已修复

1.3.2 在 AOF 文件 rewrite 期间如果设置 config set appendonly no，会导致 redis 进程一直死循环不间断触发 rewrite AOF

此 BUG 在 4.0.7 版本修复 (2018.1 月）

https://github.com/antirez/redis/commit/a18e4c964e9248008e0fba7efc1cad9ba9b8b1c3

根因

redis 在 AOF rewrite 期间设置了 appendonly no，会 kill 子进程，设置 server.aof_fd = -1，但是并未更新 server.aof_rewrite_base_size。

在 serverCron 中触发 AOF rewrite 时未判断当前 appendonly 是否为 yes，只判断了 server.aof_current_size 和 server.aof_rewrite_base_size 增长是否超过阈值

AOF rewrite 重写完成后发现 server.aof_fd=-1 也未更新 server.aof_rewrite_base_size，导致 serverCron 中一直触发 AOF rewrite。

1.3.3 redis slots 迁移的时候，永不过期的 key 因为 ttl>0 而过期，导致迁移丢失数据

详细见博客 https://blog.csdn.net/doc_sgl/article/details/53825892

对应 PR: https://github.com/antirez/redis/pull/3673/files

在 4.0rc2 版本中进行修复

根因

所有丢失 key 的 ttl 因为没有处理而使用了前一个 key 的 ttl！

问题出在下面代码的 for 循环，对于不过期的 key,ttl 应该是 0，但是如果前面有过期的 key,ttl>0. 那么在下一个处理不过期 key 时，expireat=-1，不会进入 if，ttl 还是使用前一个 ttl，导致一个永不过期的 key 因为 ttl>0 而过期。

```
/* MIGRATE host port key dbid timeout [COPY | REPLACE]
 *
 * On in the multiple keys form:
 *
 * MIGRATE host port "" dbid timeout [COPY | REPLACE] KEYS key1 key2 ... keyN */
void migrateCommand(client *c) {
    long long ttl, expireat;
    ttl = 0;
    ...

    /* Create RESTORE payload and generate the protocol to call the command. */
    /*
        问题出在这个 for 循环，对于不过期的 key,ttl 应该是 0，但是如果前面有过期的 key,ttl>0. 在处理不过期 key 时，expireat=-1，导致 ttl 还是使用前一个 ttl.
        导致一个永不过期的 key 因为 ttl>0 而过期。
    */
    for (j = 0; j < num_keys; j++) {
        /
        expireat = getExpire(c->db,kv[j]);
        if (expireat != -1) {
            ttl = expireat-mstime();
            if (ttl < 1) ttl = 1;
        }
        serverAssertWithInfo(c,NULL,rioWriteBulkCount(&cmd,'*',replace ? 5 : 4));
        if (server.cluster_enabled)
            serverAssertWithInfo(c,NULL,
                rioWriteBulkString(&cmd,"RESTORE-ASKING",14));
        else
            serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,"RESTORE",7));
        serverAssertWithInfo(c,NULL,sdsEncodedObject(kv[j]));
        serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,kv[j]->ptr,
                sdslen(kv[j]->ptr)));
        serverAssertWithInfo(c,NULL,rioWriteBulkLongLong(&cmd,ttl));

        /* Emit the payload argument, that is the serialized object using
         * the DUMP format. */
        createDumpPayload(&payload,ov[j]);
        serverAssertWithInfo(c,NULL,
            rioWriteBulkString(&cmd,payload.io.buffer.ptr,
                               sdslen(payload.io.buffer.ptr)));
        sdsfree(payload.io.buffer.ptr);

        /* Add the REPLACE option to the RESTORE command if it was specified
         * as a MIGRATE option. */
        if (replace)
            serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,"REPLACE",7));
    }
```

1.4 redis 日志

1.4.1 日常日志

```
DB 0: 1 keys (0 volatile) in 4 slots HT
```

> - Redis 中的 DB 是相互独立存在的，所以可以出现重复的 key。好处一是，对小型项目可以做如下设置： 1 号 DB 做开发，2 号 DB 做测试等等。
>   - Redis Cluster 方案只允许使用 0 号数据库
> - 0 volatile: 目前 0 号 DB 中没有 volatile key，volatile key 的意思是 过特定的时间就被 REDIS 自动删除，在做缓存时有用。
> - 4 slots HT: 目前 0 号 DB 的 hash table 只有 4 个 slots(buckets)
>   - //todo

2 Redis twemproxy 集群

> - Nutcracker，又称 Twemproxy（读音："two-em-proxy"）是支持 memcached 和 redis 协议的快速、轻量级代理；
> - 它的建立旨在减少后端缓存服务器上的连接数量；
> - 再结合管道技术（pipelining*）、及分片技术可以横向扩展分布式缓存架构；
>   - Redis pipelining（流式批处理、管道技术）：将一系列请求连续发送到 Server 端，不必每次等待 Server 端的返回，而 Server 端会将请求放进一个有序的管道中，在执行完成后，会一次性将结果返回（解决 Client 端和 Server 端的网络延迟造成的请求延迟）

2.1 Twemproxy 特性

twemproxy 的特性：

- 支持失败节点自动删除
  - 可以设置重新连接该节点的时间
  - 可以设置连接多少次之后删除该节点
- 支持设置 HashTag
  - 通过 HashTag 可以自己设定将两个 key 哈希到同一个实例上去
- 减少与 redis 的直接连接数
  - 保持与 redis 的长连接
  - 减少了客户端直接与服务器连接的连接数量
- 自动分片到后端多个 redis 实例上
  - 多种 hash 算法：md5、crc16、crc32 、crc32a、fnv1_64、fnv1a_64、fnv1_32、fnv1a_32、hsieh、murmur、jenkins
- 多种分片算法：ketama（一致性 hash 算法的一种实现）、modula、random
  - 可以设置后端实例的权重
- 避免单点问题
  - 可以平行部署多个代理层，通过 HAProxy 做负载均衡，将 redis 的读写分散到多个 twemproxy 上。
- 支持状态监控
  - 可设置状态监控 ip 和端口，访问 ip 和端口可以得到一个 json 格式的状态信息串
  - 可设置监控信息刷新间隔时间
- 使用 pipelining 处理请求和响应
  - 连接复用，内存复用
  - 将多个连接请求，组成 reids pipelining 统一向 redis 请求
- 并不是支持所有 redis 命令
  - 不支持 redis 的事务操作
  - 使用 SIDFF, SDIFFSTORE, SINTER, SINTERSTORE, SMOVE, SUNION and SUNIONSTORE 命令需要保证 key 都在同一个分片上。

2.2 环境说明

```
4 台 redis 服务器
10.10.10.4:6379   - 1
10.10.10.5:6379   - 2
```

2.2 安装依赖

安装 autoconf centos 7 yum 安装既可， autoconf 版本必须 2.64 以上版本

```
yum -y install autoconf
```

2.3 安装 Twemproxy

```
git clone https://github.com/twitter/twemproxy.git
autoreconf -fvi          #生成 configure 文件
./configure --prefix=/opt/local/twemproxy/ --enable-debug=log
make && make install
mkdir -p /opt/local/twemproxy/{run,conf,logs}
ln -s /opt/local/twemproxy/sbin/nutcracker /usr/bin/
```

2.4 配置 Twemproxy

cd /opt/local/twemproxy/conf/

vi nutcracker.yml #编辑配置文件

```
meetbill:

  listen: 10.10.10.4:6380                          #监听端口
  hash: fnv1a_64                                   #key 值 hash 算法，默认 fnv1a_64
  distribution: ketama                             #分布算法
  #ketama 一致性 hash 算法；modula 非常简单，就是根据 key 值的 hash 值取模；random 随机分布
  auto_eject_hosts: true                           #摘除后端故障节点
  redis: true                                      #是否是 redis 缓存，默认是 false
  timeout: 400                                     #代理与后端超时时间，毫秒
  server_retry_timeout: 200000                     #摘除故障节点后重新连接的时间，毫秒
  server_failure_limit: 1                          #故障多少次摘除
  servers:
   - 10.10.10.4:6379:1 server1
   - 10.10.10.5:6379:1 server2
```

检查配置文件是否正确

```
nutcracker -t -c /opt/local/twemproxy/conf/nutcracker.yml
```

2.5 启动 Twemproxy

2.5.1 启动命令详解

```
Usage: nutcracker [-?hVdDt] [-v verbosity level] [-o output file]
[-c conf file] [-s stats port] [-a stats addr]
[-i stats interval] [-p pid file] [-m mbuf size]
参数    释义
-h, –help   查看帮助文档，显示命令选项
-V, –version    查看 nutcracker 版本
-t, –test-conf  测试配置脚本的正确性
-d, –daemonize  以守护进程运行
-D, –describe-stats 打印状态描述
-v, –verbosity=N    设置日志级别 (default: 5, min: 0, max: 11)
-o, –output=S   设置日志输出路径，默认为标准错误输出 (default: stderr)
-c, –conf-file=S    指定配置文件路径 (default: conf/nutcracker.yml)
-s, –stats-port=N   设置状态监控端口，默认 22222 (default: 22222)
-a, –stats-addr=S   设置状态监控 IP，默认 0.0.0.0 (default: 0.0.0.0)
-i, –stats-interval=N   设置状态聚合间隔 (default: 30000 msec)
-p, –pid-file=S 指定进程 pid 文件路径，默认关闭 (default: off)
-m, –mbuf-size=N    设置 mbuf 块大小，以 bytes 单位 (default: 16384 bytes)
```

2.5.2 启动

```
nutcracker -d -c /opt/local/twemproxy/conf/nutcracker.yml -p /opt/local/twemproxy/run/
redisproxy.pid -o /opt/local/twemproxy/logs/redisproxy.log
```

2.6 查看状态

2.6.1 状态参数

```
nutcracker --describe-stats
This is nutcracker-0.2.4

pool stats:
  client_eof          "# eof on client connections"
  client_err          "# errors on client connections"
  client_connections  "# active client connections"
  server_ejects       "# times backend server was ejected"
  forward_error       "# times we encountered a forwarding error"
  fragments           "# fragments created from a multi-vector request"

server stats:
  server_eof          "# eof on server connections"
  server_err          "# errors on server connections"
  server_timedout     "# timeouts on server connections"
  server_connections  "# active server connections"
  requests            "# requests"
  request_bytes       "total request bytes"
  responses           "# respones"
  response_bytes      "total response bytes"
  in_queue            "# requests in incoming queue"
  in_queue_bytes      "current request bytes in incoming queue"
  out_queue           "# requests in outgoing queue"
  out_queue_bytes     "current request bytes in outgoing queue"
```

2.6.2 状态实例

```
#curl  -s http://127.0.0.1:22222|python -mjson.tool
{
    "meetbill": {                    # 配置名称
        "client_connections": 0,     # 当前活跃的客户端连接数
        "client_eof": 0,
        "client_err": 2,             # 客户端连接错误次数
        "forward_error": 0,          # 转发错误次数
        "fragments": 0,
        "server_ejects": 0           # 后端服务被踢出次数
        "server1": {
            "in_queue": 0,
            "in_queue_bytes": 0,
            "out_queue": 0,
            "out_queue_bytes": 0,
            "request_bytes": 0,      # 已请求字节数
            "requests": 0,           # 已请求次数
            "response_bytes": 0,     # 已相应字节数
            "responses": 0,          # 已响应次数
            "server_connections": 0, # 当前活跃的服务端连接数
            "server_eof": 0,
            "server_err": 0,         # 服务端连接错误次数
            "server_timedout": 0     # 因连接超时的服务端错误次数
        },
        "server2": {
            "in_queue": 0,
            "in_queue_bytes": 0,
            "out_queue": 0,
            "out_queue_bytes": 0,
            "request_bytes": 0,
            "requests": 0,
            "response_bytes": 0,
            "responses": 0,
            "server_connections": 0,
            "server_eof": 0,
            "server_err": 0,
            "server_timedout": 0
        },
    },
    "service": "nutcracker",
    "source": "meetbill",    # 主机名
    "timestamp": 1520780415, # 当前时间戳
    "uptime": 3160,          # 服务已经启动的时间（单位：秒
    "version": "0.2.4"
}
```

2.6.3 使用 Python 获取 Twemproxy 状态

使用 curl 获取 Twemproxy 状态时，如果后端的 redis 或者 memcache 过多，将会导致获取状态内容失败，可以进行如下解决方法

```
def fetch_stats(ip, port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))
    raw = ""
    while True:
        data = s.recv(1024)
        if len(data) == 0:
            break
        raw += data
    s.close()
    stats = json.loads(raw)
    return stats
```

3 redis cluster

3.1 cluster 命令

- 集群 (cluster)
  - cluster info 打印集群的信息
  - cluster nodes 列出集群当前已知的所有节点 (node)，以及这些节点的相关信息
- 节点 (node)
  - cluster meet 将 ip 和 port 所指定的节点添加到集群当中，让它成为集群的一份子
  - cluster forget 从集群中移除 node_id 指定的节点
  - cluster replicate 将当前节点设置为 node_id 指定的节点的从节点
  - cluster saveconfig 将节点的配置文件保存到硬盘里面
  - cluster slaves 列出该 slave 节点的 master 节点
  - cluster set-config-epoch 强制设置 configEpoch
- 槽 (slot)
  - cluster addslots [slot ...] 将一个或多个槽 (slot) 指派 (assign) 给当前节点
  - cluster delslots [slot ...] 移除一个或多个槽对当前节点的指派
  - cluster flushslots 移除指派给当前节点的所有槽，让当前节点变成一个没有指派任何槽的节点
  - cluster setslot node 将槽 slot 指派给 node_id 指定的节点，如果槽已经指派给另一个节点，那么先让另一个节点删除该槽，然后再进行指派
  - cluster setslot migrating 将本节点的槽 slot 迁移到 node_id 指定的节点中
  - cluster setslot importing 从 node_id 指定的节点中导入槽 slot 到本节点
  - cluster setslot stable 取消对槽 slot 的导入 (import) 或者迁移 (migrate)
- 键 (key)
  - cluster keyslot 计算键 key 应该被放置在哪个槽上
  - cluster countkeysinslot 返回槽 slot 目前包含的键值对数量
  - cluster getkeysinslot 返回 count 个 slot 槽中的键
- 其它
  - cluster myid 返回节点的 ID
  - cluster slots 返回节点负责的 slot
  - cluster reset 重置集群，慎用

3.2 redis cluster 配置

```
cluster-enabled yes
```

如果配置 yes 则开启集群功能，此 redis 实例作为集群的一个节点，否则，它是一个普通的单一的 redis 实例。

```
cluster-config-file nodes-6379.conf
```

虽然此配置的名字叫"集群配置文件"，但是此配置文件不能人工编辑，它是集群节点自动维护的文件，主要用于记录集群中有哪些节点、他们的状态以及一些持久化参数等，方便在重启时恢复这些状态。通常是在收到请求之后这个文件就会被更新。

```
cluster-node-timeout 15000
```

这是集群中的节点能够失联的最大时间，超过这个时间，该节点就会被认为故障。如果主节点超过这个时间还是不可达，则用它的从节点将启动故障迁移，升级成主节点。注意，任何一个节点在这个时间之内如果还是没有连上大部分的主节点，则此节点将停止接收任何请求。一般设置为 15 秒即可。

```
cluster-slave-validity-factor 10
```

如果设置成 0，则无论从节点与主节点失联多久，从节点都会尝试升级成主节点。如果设置成正数，则 cluster-node-timeout 乘以 cluster-slave-validity-factor 得到的时间，是从节点与主节点失联后，此从节点数据有效的最长时间，超过这个时间，从节点不会启动故障迁移。假设 cluster-node-timeout=5，cluster-slave-validity-factor=10，则如果从节点跟主节点失联超过 50 秒，此从节点不能成为主节点。注意，如果此参数配置为非 0，将可能出现由于某主节点失联却没有从节点能顶上的情况，从而导致集群不能正常工作，在这种情况下，只有等到原来的主节点重新回归到集群，集群才恢复运作。

```
cluster-migration-barrier 1
```

主节点需要的最小从节点数，只有达到这个数，主节点失败时，它从节点才会进行迁移。更详细介绍可以看本教程后面关于副本迁移到部分。

```
cluster-require-full-coverage yes
```

在部分 key 所在的节点不可用时，如果此参数设置为"yes"（默认值）, 则整个集群停止接受操作；如果此参数设置为”no”，则集群依然为可达节点上的 key 提供读操作。

3.3 redis cluster 状态

127.0.0.1:8001> cluster info

- cluster_state:ok
  - 如果当前 redis 发现有 failed 的 slots，默认为把自己 cluster_state 从 ok 个性为 fail, 写入命令会失败。如果设置 cluster-require-full-coverage 为 no, 则无此限制。
- cluster_slots_assigned:16384 #已分配的槽
- cluster_slots_ok:16384 #槽的状态是 ok 的数目
- cluster_slots_pfail:0 #可能失效的槽的数目
- cluster_slots_fail:0 #已经失效的槽的数目
- cluster_known_nodes:6 #集群中节点个数
- cluster_size:3 #集群中设置的分片个数
- cluster_current_epoch:15 #集群中的 currentEpoch 总是一致的，currentEpoch 越高，代表节点的配置或者操作越新，集群中最大的那个 node epoch
- cluster_my_epoch:12 #当前节点的 config epoch，每个主节点都不同，一直递增，其表示某节点最后一次变成主节点或获取新 slot 所有权的逻辑时间。
- cluster_stats_messages_sent:270782059
- cluster_stats_messages_received:270732696

```
127.0.0.1:8001> cluster nodes
25e8c9379c3db621da6ff8152684dc95dbe2e163 192.168.64.102:8002 master - 0 1490696025496
15 connected 5461-10922
d777a98ff16901dffca53e509b78b65dd1394ce2 192.168.64.156:8001 slave 0b1f3dd6e53ba76b866
4294af2b7f492dbf914ec 0 1490696027498 12 connected
8e082ea9fe9d4c4fcca4fbe75ba3b77512b695ef 192.168.64.108:8000 master - 0 1490696025997
14 connected 0-5460
0b1f3dd6e53ba76b8664294af2b7f492dbf914ec 192.168.64.170:8001 myself,master - 0 0 12 co
nnected 10923-16383
eb8adb8c0c5715525997bdb3c2d5345e688d943f 192.168.64.101:8002 slave 25e8c9379c3db621da6
ff8152684dc95dbe2e163 0 1490696027498 15 connected
4000155a787ddab1e7f12584dabeab48a617fc46 192.168.67.54:8000 slave 8e082ea9fe9d4c4fcca4
fbe75ba3b77512b695ef 0 1490696026497 14 connected
```

- 节点 ID：例如 25e8c9379c3db621da6ff8152684dc95dbe2e163
- ip:port：节点的 ip 地址和端口号，例如 192.168.64.102:8002
- flags：节点的角色 (master,slave,myself) 以及状态 (pfail,fail)
- 如果节点是一个从节点的话，那么跟在 flags 之后的将是主节点的节点 ID，例如 192.168.64.156:8001 主节点的 ID 就是 0b1f3dd6e53ba76b8664294af2b7f492dbf914ec
- 集群最近一次向节点发送 ping 命令之后，过了多长时间还没接到回复
- 节点最近一次返回 pong 回复的时间
- 节点的配置纪元 (config epoch)
- 本节点的网络连接情况
- 节点目前包含的槽，例如 192.168.64.102:8002 目前包含的槽为 5461-10922

4 原理说明

4.1 一致性 hash

4.1.1 传统的取模方式

例如 10 条数据，3 个节点，如果按照取模的方式，那就是

> - node a: 0,3,6,9
> - node b: 1,4,7
> - node c: 2,5,8

当增加一个节点的时候，数据分布就变更为

> - node a:0,4,8
> - node b:1,5,9
> - node c: 2,6
> - node d: 3,7

总结：数据 3,4,5,6,7,8,9 在增加节点的时候，都需要做搬迁，成本太高

4.1.2 一致性哈希方式

最关键的区别就是，对节点和数据，都做一次哈希运算，然后比较节点和数据的哈希值，数据取和节点最相近的节点做为存放节点。这样就保证当节点增加或者减少的时候，影响的数据最少。还是拿刚刚的例子，（用简单的字符串的 ascii 码做哈希 key）：

十条数据，算出各自的哈希值

> - 0：192
> - 1：196
> - 2：200
> - 3：204
> - 4：208
> - 5：212
> - 6：216
> - 7：220
> - 8：224
> - 9：228

有三个节点，算出各自的哈希值

> - node a: 203
> - node g: 209
> - node z: 228

这个时候比较两者的哈希值，如果大于 228，就归到前面的 203，相当于整个哈希值就是一个环，对应的映射结果：

> - node a: 0,1,2
> - node g: 3,4
> - node z: 5,6,7,8,9

这个时候加入 node n, 就可以算出 node n 的哈希值：

> - node n: 216

这个时候对应的数据就会做迁移：

> - node a: 0,1,2
> - node g: 3,4
> - node n: 5,6
> - node z: 7,8,9

这个时候只有 5 和 6 需要做迁移

4.1.3 虚拟节点

另外，这个时候如果只算出三个哈希值，那再跟数据的哈希值比较的时候，很容易分得不均衡，因此就引入了虚拟节点的概念，通过把三个节点加上 ID 后缀等方式，每个节点算出 n 个哈希值，均匀的放在哈希环上，这样对于数据算出的哈希值，能够比较散列的分布（详见下面代码中的 replica）

通过这种算法做数据分布，在增减节点的时候，可以大大减少数据的迁移规模。

4.2 redis 过期数据存储方式以及删除方式

当你通过 expire 或者 pexpire 命令，给某个键设置了过期时间，那么它在服务器是怎么存储的呢？到达过期时间后，又是怎么删除的呢？

4.2.1 存储方式

比如：

```
redis> EXPIRE book 5
(integer) 1
```

首先我们知道，redis 维护了一个存储了所有的设置的 key->value 的字典。但是其实不止一个字典的。

redis 有一个包含过期事件的字典

每当有设置过期事件的 key 后，redis 会用当前的事件，加上过期的时间段，得到过期的标准时间，存储在 expires 字典中。

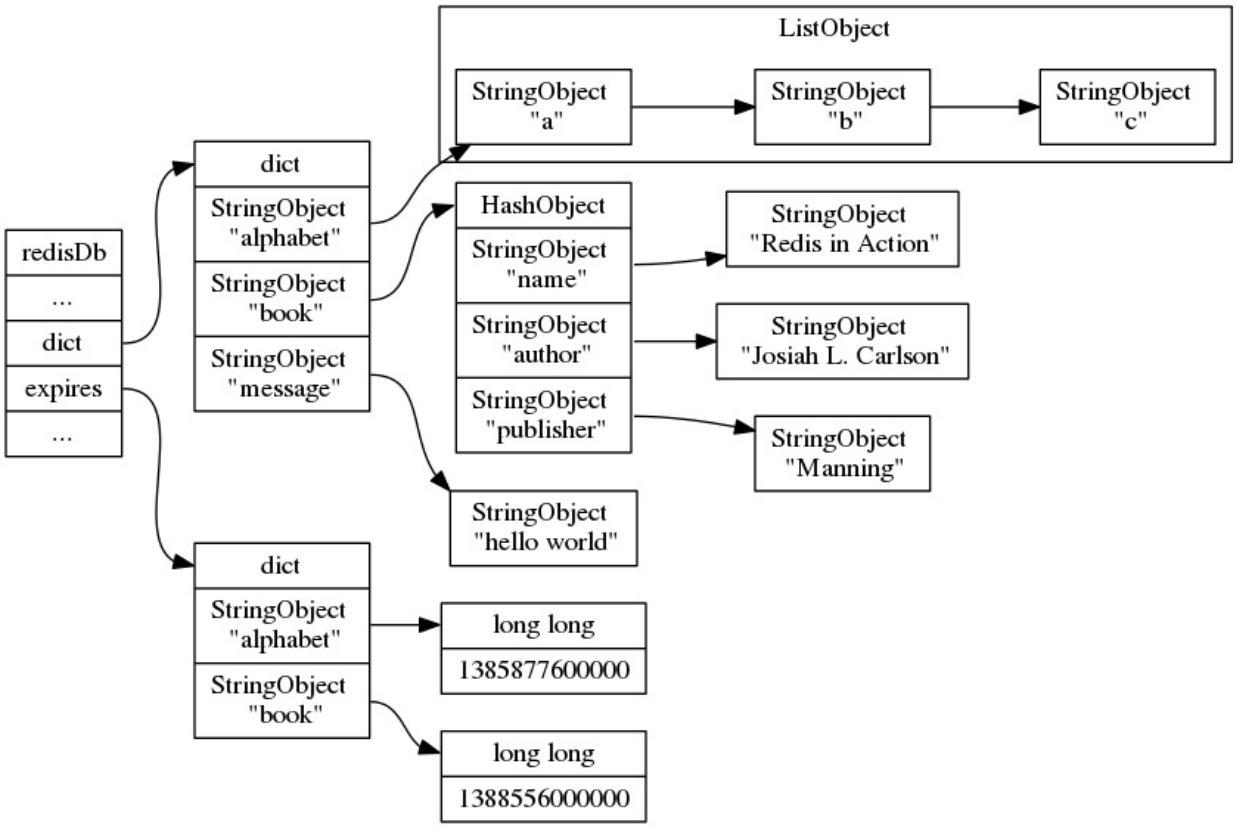


图 9-12　带有过期字典的数据库例子

从上图可以看出来，比如你给 book 设置过期事件，那么 expires 字典的 key 也为 book，值是当前的时间 +5s 后的 unix time。

4.2.2 删除方式

如果一个键已经过期了，那么 redis 的如果删除它呢？redis 采用了 2 种删除方式；

惰性删除

惰性删除的原理是：放任键过期不管，但是每次从键空间获取键的时候，如果该键存在，再去 expires 字典判断这个键是不是过期。如果过期则返回空，并删除该键。过程如下：

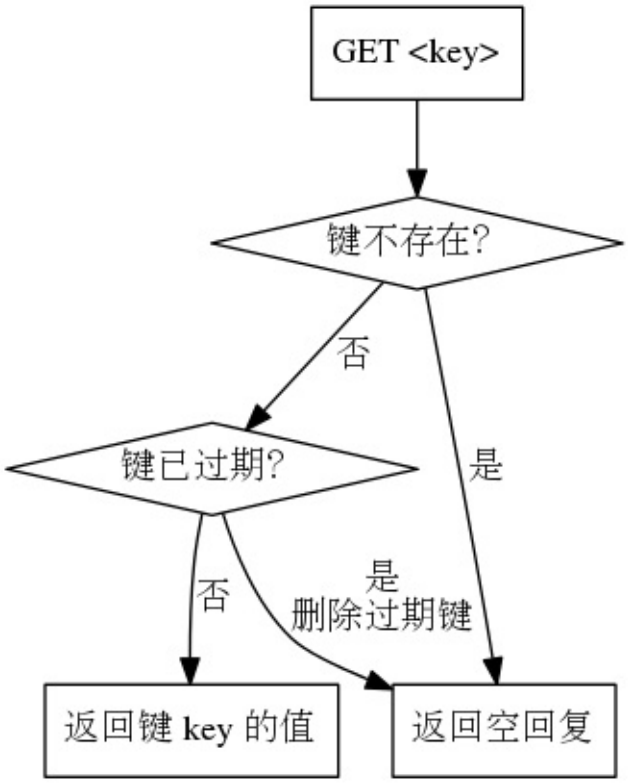


图 9-16　GET 命令的执行过程

- 优点：惰性删除对 cpu 是友好的。保证在键必须删除的时候才会消耗 cpu
- 缺点：惰性删除对内存特别不友好。虽然键过期，但是没有使用则一直存在内存中。

定期删除

redis 架构中的时间事件，每隔一段时间后，在规定的时间内，会主动去检测 expires 字典中包含的 key 进行检测，发现过期的则删除。在 redis 的源码 redis.c/activeExpireCycle 函数中。 下面分别是这个函数的源码与伪代码：

```
void  activeExpireCycle(int type) {
    // 静态变量，用来累积函数连续执行时的数据
    static  unsigned  int current_db =  0; /* Last DB tested. */
    static  int timelimit_exit =  0; /* Time limit hit in previous call? */
    static  long  long last_fast_cycle =  0; /* When last fast cycle ran. */

    unsigned  int j, iteration =  0;
    // 默认每次处理的数据库数量
    unsigned  int dbs_per_call = REDIS_DBCRON_DBS_PER_CALL;
    // 函数开始的时间
    long  long start =  ustime(), timelimit;

    // 快速模式
    if (type == ACTIVE_EXPIRE_CYCLE_FAST) {
        // 如果上次函数没有触发 timelimit_exit ，那么不执行处理
        if (!timelimit_exit) return;
        // 如果距离上次执行未够一定时间，那么不执行处理
```

```
        if (start < last_fast_cycle + ACTIVE_EXPIRE_CYCLE_FAST_DURATION*2) return;
        // 运行到这里，说明执行快速处理，记录当前时间
        last_fast_cycle = start;
    }

    /*
    * 一般情况下，函数只处理 REDIS_DBCRON_DBS_PER_CALL 个数据库，
    * 除非：
    * 当前数据库的数量小于 REDIS_DBCRON_DBS_PER_CALL
    * 如果上次处理遇到了时间上限，那么这次需要对所有数据库进行扫描，
    * 这可以避免过多的过期键占用空间
    */
    if (dbs_per_call > server.dbnum  || timelimit_exit)
    dbs_per_call = server.dbnum;

    // 函数处理的微秒时间上限
    // ACTIVE_EXPIRE_CYCLE_SLOW_TIME_PERC 默认为 25 ，也即是 25 % 的 CPU 时间
    timelimit =  1000000*ACTIVE_EXPIRE_CYCLE_SLOW_TIME_PERC/server.hz/100;
    timelimit_exit =  0;
    if (timelimit <=  0) timelimit =  1;

    // 如果是运行在快速模式之下
    // 那么最多只能运行 FAST_DURATION 微秒
    // 默认值为 1000 （微秒）
    if (type == ACTIVE_EXPIRE_CYCLE_FAST)
    timelimit = ACTIVE_EXPIRE_CYCLE_FAST_DURATION; /* in microseconds. */

    // 遍历数据库
    for (j =  0; j < dbs_per_call; j++) {
        int expired;
        // 指向要处理的数据库
        redisDb *db = server.db+(current_db % server.dbnum);

        // 为 DB 计数器加一，如果进入 do 循环之后因为超时而跳出
        // 那么下次会直接从下个 DB 开始处理
        current_db++;

        do {
            unsigned  long num, slots;
            long  long now, ttl_sum;
            int ttl_samples;

            // 获取数据库中带过期时间的键的数量
            // 如果该数量为 0 ，直接跳过这个数据库
            if ((num =  dictSize(db->expires)) ==  0) {
                db->avg_ttl  =  0;
                break;
            }
            // 获取数据库中键值对的数量
            slots =  dictSlots(db->expires);
            // 当前时间
            now =  mstime();
```

```
            // 这个数据库的使用率低于 1% ，扫描起来太费力了（大部分都会 MISS）
            // 跳过，等待字典收缩程序运行
            if (num && slots > DICT_HT_INITIAL_SIZE &&
            (num*100/slots <  1)) break;

            // 已处理过期键计数器
            expired =  0;
            // 键的总 TTL 计数器
            ttl_sum =  0;
            // 总共处理的键计数器
            ttl_samples =  0;

            // 每次最多只能检查 LOOKUPS_PER_LOOP 个键
            if (num > ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP)
            num = ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP;

            // 开始遍历数据库
            while (num--) {
                dictEntry *de;
                long  long ttl;

                // 从 expires 中随机取出一个带过期时间的键
                if ((de =  dictGetRandomKey(db->expires)) ==  NULL) break;
                // 计算 TTL
                ttl =  dictGetSignedIntegerVal(de)-now;
                // 如果键已经过期，那么删除它，并将 expired 计数器增一
                if (activeExpireCycleTryExpire(db,de,now)) expired++;
                if (ttl <  0) ttl =  0;
                // 累积键的 TTL
                ttl_sum += ttl;
                // 累积处理键的个数
                ttl_samples++;
            }

            // 为这个数据库更新平均 TTL 统计数据
            if (ttl_samples) {
                // 计算当前平均值
                long  long avg_ttl = ttl_sum/ttl_samples;
                // 如果这是第一次设置数据库平均 TTL ，那么进行初始化
                if (db->avg_ttl  ==  0) db->avg_ttl  = avg_ttl;
                /* Smooth the value averaging with the previous one. */
                // 取数据库的上次平均 TTL 和今次平均 TTL 的平均值
                db->avg_ttl  = (db->avg_ttl+avg_ttl)/2;
            }

            // 我们不能用太长时间处理过期键，
            // 所以这个函数执行一定时间之后就要返回

            // 更新遍历次数
            iteration++;

            // 每遍历 16 次执行一次
            if ((iteration &  0xf) ==  0  &&  /* check once every 16 iterations. */
```

```
            (ustime()-start) > timelimit)
            {
                // 如果遍历次数正好是 16 的倍数
                // 并且遍历的时间超过了 timelimit
                // 那么断开 timelimit_exit
                timelimit_exit =  1;
            }

            // 已经超时了，返回
            if (timelimit_exit) return;

            // 如果已删除的过期键占当前总数据库带过期时间的键数量的 25 %
            // 那么不再遍历
        } while (expired > ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP/4);
    }
}
```

伪代码是：

```
# 默认每次检测的数据库数量为 16
DEFAULT_DB_NUMBERS = 16
# 默认每次检测的键的数量最大为 20
DEFAULT_KEY_NUMBERS = 20
# 全局变量，记录当前检测的进度
current_db = 0
def activeExpireCycle():
    # 初始化要检测的数据库数量
    # 如果服务器的数据库数量小于 16，则以服务器的为准
    if server.dbnumbers < DEFAULT_DB_NUMBERS:
        db_numbers = server.dbnumbers
    else
        db_numbers = DEFAULT_DB_NUMBERS

    # 遍历每次数据库
    for i in range(db_numbers):
        # 如果 current_db 的值等于服务器的数量，代表已经遍历全，则重新开始
        if current_db = db_numbers:
            current_db = 0

        # 获取当前要处理的数据库
        redisDb = server.db[current_db]

        # 将数据库索引 +1，指向下一个数据库
        current_db++

        do
            # 检测数据库中的键
            for j in range(DEFAULT_KEY_NUMBERS):
                # 如果数据库中没有过期键则跳过这个库
                if redisDb.expires.size() == 0:break

                # 随机获取一个带有过期事件的键
                key_with_ttl = redisDb.expires.get_random_key()

                # 检测键是不是过期了，如果过期则删除
                if is_expired(key_with_ttl):
                    delete_key(key_with_ttl)
            # 已达到时间上限，则停止处理
            if reach_time_limit(): retrun
        while expired>ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP/4
```

对 activeExpireCycle 进行总结：

- redis 默认 1s 调用 10 次，这个是 redis 的配置中的 hz 选项。hz 默认是 10，代表 1s 调用 10 次，每 100ms 调用一次。
- hz 不能太大，太大的话，cpu 会花大量的时间消耗在判断过期的 key 上，对 cpu 不友好。但是如果你的 redis 过期数据过多，可以适当调大。
- hz 不能太小，因为太小的话，一旦过期的 key 太多可能会过滤不完。
- redis 执行定期删除函数，必须在一定时间内，超过该时间就 return。事件定义

为 `timelimit = 1000000*ACTIVE_EXPIRE_CYCLE_SLOW_TIME_PERC/server.hz/100` 可以看出该时间与 hz 成反比，hz 默认 10，timelimit 就为 25ms；hz 修改为 100，那么 timelimit 就为 2.5ms。

- 抽取 20 个数据进行判断删除为一个轮训，每经过 16 个轮训才会去判断一次时间是不是超时。
- 如果一个数据库，使用率低于 1%，则不去进行定期删除操作。
- 如果对一个数据库，这次删除操作，已经删除了 25% 的过期 key，那么就跳过这个库。

4.2.3 redis 主从删除过期 key 方式

当 redis 主从模型下，从服务器的删除过期 key 的动作是由主服务器控制的。

- 1、主服务器在惰性删除、客户端主动删除、定期删除一个 key 的时候，会向从服务器发送一个 del 的命令，告诉从服务器需要删除这个 key。

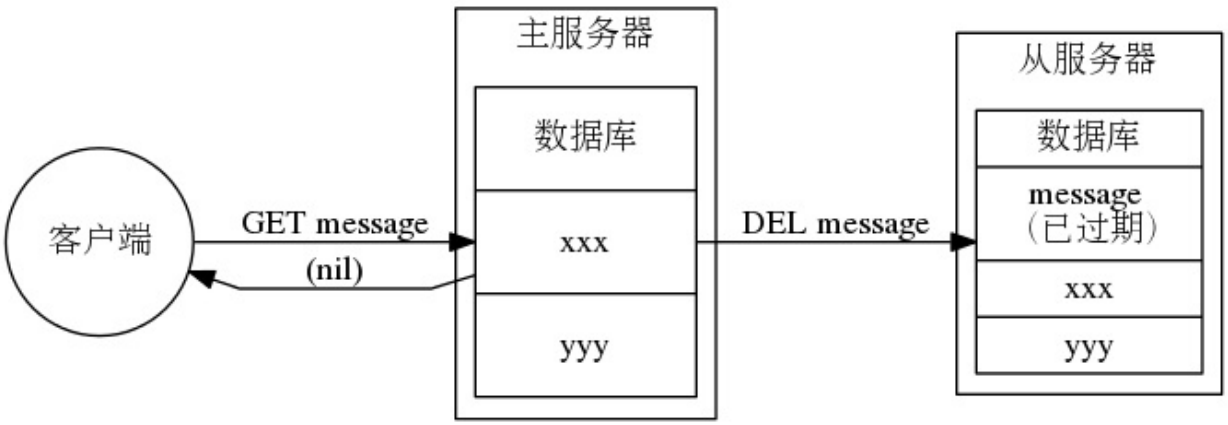


图 9-19　主从服务器删除过期键 （3）

- 2、从服务器在执行客户端读取 key 的时候，如果该 key 已经过期，也不会将该 key 删除，而是返回一个 null

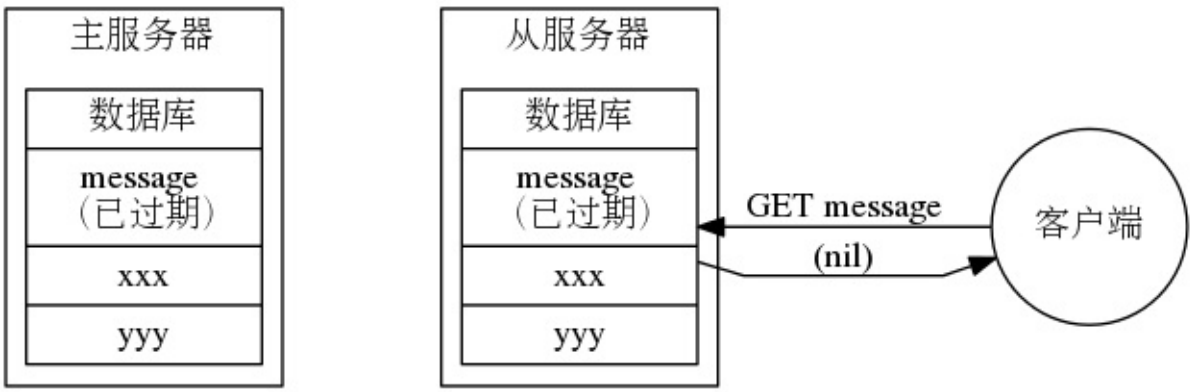


图 9-18　主从服务器删除过期键 （2）

- 3、从服务器只有在接收到主服务器的 del 命令才会将一个 key 进行删除。

4.2.4 总结

- 1、expires 字典的 key 指向数据库中的某个 key，而值记录了数据库中该 key 的过期时间，过期时间是一个以毫秒为单位的 unix 时间戳；
- 2、redis 使用惰性删除和定期删除两种策略来删除过期的 key；惰性删除只会在碰到过期 key 才会删除；定期删除则每隔一段时间主动查找并删除过期键；
- 3、当主服务器删除一个过期 key 后，会向所有的从服务器发送一条 del 命令，显式的删除过期 key；
- 4、从服务器即使发现过期 key 也不会自作主张删除它，而是等待主服务器发送 del 命令，这种统一、中心化的过期 key 删除策略可以保证主从服务器的数据一致性。

5 其他相关

5.1 内核参数 overcommit

它是 内存分配策略，可选值：0、1、2。

> - 0， 表示内核将检查是否有足够的可用内存供应用进程使用；如果有足够的可用内存，内存申请允许；否则，内存申请失败，并把错误返回给应用进程。
> - 1， 表示内核允许分配所有的物理内存，而不管当前的内存状态如何。
> - 2， 表示内核允许分配超过所有物理内存和交换空间总和的内存

什么是 Overcommit 和 OOM

Linux 对大部分申请内存的请求都回复"yes"，以便能跑更多更大的程序。因为申请内存后，并不会马上使用内存。这种技术叫做 Overcommit。当 linux 发现内存不足时，会发生 OOM killer(OOM=out-of-memory)。它会选择杀死一些进程（用户态进程，不是内核线程），以便释放内存。 当 oom-killer 发生时，linux 会选择杀死哪些进程？选择进程的函数是 oom_badness 函数（在 mm/oom_kill.c 中），该函数会计算每个进程的点数 (0~1000)。点数越高，这个进程越有可能被杀死。每个进程的点数跟 oom_score_adj 有关，而且 oom_score_adj 可以被设置 (-1000 最低，1000 最高）。

解决方法： 很简单，按提示的操作（将 vm.overcommit_memory 设为 1）即可：可以通过 `cat /proc/sys/vm/overcommit_memory` 和 `sysctl -a | grep overcommit` 查看 有三种方式修改内核参数，但要有 root 权限：

> - （1）编辑 /etc/sysctl.conf ，改 vm.overcommit_memory=1，然后 sysctl -p 使配置文件生效
> - （2）sysctl vm.overcommit_memory=1
> - （3）echo 1 > /proc/sys/vm/overcommit_memory

MemCache

1 MemCache 访问热点导致服务雪崩 case

1.1 背景

15 年，某产品线发生了一起严重的丢失用户流量的事故，就这个 case 来谈谈 MemCache 使用不当的问题。

他们的使用方法是这样的，在站点的主页上每次请求会首先请求一个单热点 key，value 大概在 250 KB 左右。

以千兆网卡的容量计算，热点机器网卡容量极限为 400 QPS。若请求失败（cache 失效、访问超时等），PHP 会根据业务逻辑，再请求大约 600 个 cache 数据。

然后重新构造首页的数据块。

在当天下午 14 ~ 15 点左右，用户流量有自然增长，超过了 400 QPS，于是那台热点的机器单机网卡打满，大批量的首页请求获取热点 cache 失败。

PHP 业务为了重新构造数据块，另外请求 600 个 key，导致所有的 cache 机器请求都上涨，网卡占用上涨。同时，由于请求 600 多次 cache 需要耗时过长，产生了很多的长耗时请求，这些请求占用 dbproxy 连接不释放，导致 DB 的连接数也打满。

此时，其他请求大量失败，因此对于 memcache 调用减少，但是还是有相当量的首页请求仍然在请求单热点，导致单热点网卡依然处于打满情况，其他机器网卡有所下降。此时产品线无法提供正常服务，处于挂站状态。

1.2 那我们如何防止这样的事故发生

1.2.1 明确使用场景，防止滥用

首先要确定一个需求是不是适合用 cache。大多数场景下，cache 里存储的都是几百个字节的小数据（如帖子列表、用户信息、图片元信息等），复杂的结构数据序列化之后一般也不会超过 2 KB。250 KB 的大数据块，如果是图片，应该塞到图片存储系统；如果是整个网页，那么展示时“实时渲染 VS 直接从 cache 取”这两种方案，还需商榷。

1.2.2 不人为制造访问热点

Memcache 的访问本身就具有一定的热点（比如某些书看的人多、一段时间内的热门话题等），在实际工程中，这些热点也是需要尽量被平均的。然而在这个 case 中，人为制造了热点，即，对同一个 key 的访问在每一个请求的关键路径上，这是一定需要避免的。

1.2.3 实例拆分

多个业务（如主页和文章列表等）使用同一个 Memcache 实例，某一个业务流量飙升（正常增长或隐藏的 bug 导致流量异常）就会导致其它所有的业务访问受影响，一挂挂全站。这时需要把比较重要的服务依赖的 cache 拆分成单独的实例，尽量减少互相影响，提升可用性。

web篇

- web 基础
- nginx
- django

web 基础

1 一次完整的 HTTP 请求所经历的 7 个步骤

HTTP 通信机制是在一次完整的 HTTP 通信过程中，Web 浏览器与 Web 服务器之间将完成下列 7 个步骤：

- (1) 建立 TCP 连接
  - 在 HTTP 工作开始之前，Web 浏览器首先要通过网络与 Web 服务器建立连接，该连接是通过 TCP 来完成的，该协议与 IP 协议共同构建 Internet，即著名的 TCP/IP 协议族，因此 Internet 又被称作是 TCP/IP 网络。HTTP 是比 TCP 更高层次的应用层协议，根据规则，只有低层协议建立之后才能，才能进行更层协议的连接，因此，首先要建立 TCP 连接，一般 TCP 连接的端口号是 80。
- (2) Web 浏览器向 Web 服务器发送请求命令
  - 一旦建立了 TCP 连接，Web 浏览器就会向 Web 服务器发送请求命令。例如：GET/sample/hello.jsp HTTP/1.1。
- (3) Web 浏览器发送请求头信息
  - 浏览器发送其请求命令之后，还要以头信息的形式向 Web 服务器发送一些别的信息，之后浏览器发送了一空白行来通知服务器，它已经结束了该头信息的发送。
- (4) Web 服务器应答
  - 客户机向服务器发出请求后，服务器会客户机回送应答， HTTP/1.1 200 OK ，应答的第一部分是协议的版本号和应答状态码。
- (5) Web 服务器发送应答头信息
  - 正如客户端会随同请求发送关于自身的信息一样，服务器也会随同应答向用户发送关于它自己的数据及被请求的文档。
- (6) Web 服务器向浏览器发送数据
  - Web 服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以 Content-Type 应答头信息所描述的格式发送用户所请求的实际数据。
- Web 服务器关闭 TCP 连接
  - 一般情况下，一旦 Web 服务器向浏览器发送了请求数据，它就要关闭 TCP 连接，然后如果浏览器或者服务器在其头信息加入了这行代码：
  - Connection:keep-alive
  - TCP 连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

2 HTTP 报文

HTTP 报文是面向文本的，报文中的每一个字段都是一些 ASCII 码串，各个字段的长度是不确定的。HTTP 有两类报文：请求报文和响应报文。

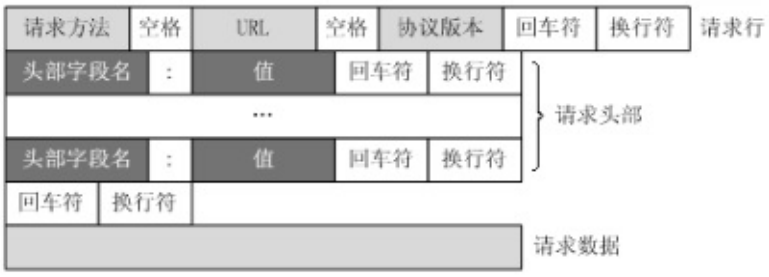
2.1 HTTP 请求报文解剖

HTTP 请求报文由 3 部分组成（请求行 + 请求头 + 请求体）：

下面是一个实际的请求报文：

> - (1) 是请求方法，GET 和 POST 是最常见的 HTTP 方法，除此以外还包括 DELETE、HEAD、OPTIONS、PUT、TRACE。不过，当前的大多数浏览器只支持 GET 和 POST，Spring 3.0 提供了一个 HiddenHttpMethodFilter，允许你通过“_method”的表单参数指定这些特殊的 HTTP 方法（实际上还是通过 POST 提交表单）。服务端配置了 HiddenHttpMethodFilter 后，Spring 会根据_method 参数指定的值模拟出相应的 HTTP 方法，这样，就可以使用这些 HTTP 方法对处理方法进行映射了。
> - (2) 为请求对应的 URL 地址，它和报文头的 Host 属性组成完整的请求 URL，
> - (3) 是协议名称及版本号。
> - (4) 是 HTTP 的报文头，报文头包含若干个属性，格式为“属性名：属性值”，服务端据此获取客户端的信息。
> - (5) 是报文体，它将一个页面表单中的组件值通过 param1=value1&param2=value2 的键值对形式编码成一个格式化串，它承载多个请求参数的数据。不但报文体可以传递请求参数，请求 URL 也可以通过类似于“/chapter15/user.html?param1=value1&param2=value2”的方式传递请求参数。

对照上面的请求报文，我们把它进一步分解，你可以看到一幅更详细的结构图：



HTTP 请求报文头属性

报文头属性是什么东西呢？我们不妨以一个小故事来说明吧。

> 快到中午了，张三丰不想去食堂吃饭，于是打电话叫外卖：老板，我要一份『鱼香肉丝』，要 12：30 之前给我送过来哦，我在江湖湖公司研发部，叫张三丰。

这里，你要『鱼香肉丝』相当于 HTTP 报文体，而“12：30 之前送过来”，你叫“张三丰”等信息就相当于 HTTP 的报文头。它们是一些附属信息，帮忙你和饭店老板顺利完成这次交易。

请求 HTTP 报文和响应 HTTP 报文都拥有若干个报文关属性，它们是为协助客户端及服务端交易的一些附属信息。

常见的 HTTP 请求报文头属性

Accept

请求报文可通过一个“Accept”报文头属性告诉服务端 客户端接受什么类型的响应。

如下报文头相当于告诉服务端，俺客户端能够接受的响应类型仅为纯文本数据啊，你丫别发其它什么图片啊，视频啊过来，那样我会歇菜的~~~：

```
Accept:text/plain
```

Accept 属性的值可以为一个或多个 MIME 类型的值，关于 MIME 类型，大家请参考：http://en.wikipedia.org/wiki/MIME_type

Cookie

客户端的 Cookie 就是通过这个报文头属性传给服务端的哦！如下所示：

```
Cookie: $Version=1; Skin=new;jsessionid=5F4771183629C9834F8382E23BE13C4C
```

服务端是怎么知道客户端的多个请求是隶属于一个 Session 呢？注意到后台的那个 jsessionid=5F4771183629C9834F8382E23BE13C4C 木有？原来就是通过 HTTP 请求报文头的 Cookie 属性的 jsessionid 的值关联起来的！（当然也可以通过重写 URL 的方式将会话 ID 附带在每个 URL 的后面哦）。

Referer

表示这个请求是从哪个 URL 过来的，假如你通过 google 搜索出一个商家的广告页面，你对这个广告页面感兴趣，鼠标一点发送一个请求报文到商家的网站，这个请求报文的 Referer 报文头属性值就是 http://www.google.com。

```
唐僧到了西天。
如来问：侬是不是从东土大唐来啊？
唐僧：厉害！你咋知道的！
如来：呵呵，我偷看了你的 Referer...
```

Cache-Control

对缓存进行控制，如一个请求希望响应返回的内容在客户端要被缓存一年，或不希望被缓存就可以通过这个报文头达到目的。

如以下设置，相当于让服务端将对应请求返回的响应内容不要在客户端缓存：

```
Cache-Control: no-cache
```

其它请求报文头属性

参见：http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

2.2 HTTP 响应报文解剖

响应报文结构

HTTP 的响应报文也由三部分组成（响应行 + 响应头 + 响应体）：

以下是一个实际的 HTTP 响应报文：

> - (1) 报文协议及版本；
> - (2) 状态码及状态描述；
> - (3) 响应报文头，也是由多个属性组成；
> - (4) 响应报文体，即我们真正要的“干货”。

响应状态码

和请求报文相比，响应报文多了一个“响应状态码”，它以“清晰明确”的语言告诉客户端本次请求的处理结果。

HTTP 的响应状态码由 5 段组成：

> - 1xx 消息，一般是告诉客户端，请求已经收到了，正在处理，别急...
> - 2xx 处理成功，一般表示：请求收悉、我明白你要的、请求已受理、已经处理完成等信息。
> - 3xx 重定向到其它地方。它让客户端再发起一个请求以完成整个处理。
> - 4xx 处理发生错误，责任在客户端，如客户端的请求一个不存在的资源，客户端未被授权，禁止访问等。
> - 5xx 处理发生错误，责任在服务端，如服务端抛出异常，路由出错，HTTP 版本不支持等。

以下是几个常见的状态码：

> - 200 OK

你最希望看到的，即处理成功！

> - 303 See Other

我把你 redirect 到其它的页面，目标的 URL 通过响应报文头的 Location 告诉你。

```
悟空：师傅给个桃吧，走了一天了
唐僧：我哪有桃啊！去王母娘娘那找吧
```

> - 304 Not Modified

告诉客户端，你请求的这个资源至你上次取得后，并没有更改，你直接用你本地的缓存吧，我很忙哦，你能不能少来烦我啊！

> - 404 Not Found

你最不希望看到的，即找不到页面。如你在 google 上找到一个页面，点击这个链接返回 404，表示这个页面已经被网站删除了，google 那边的记录只是美好的回忆。

> - 500 Internal Server Error

看到这个错误，你就应该查查服务端的日志了，肯定抛出了一堆异常，别睡了，起来改 BUG 去吧！

其它的状态码参见：http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

有些响应码，Web 应用服务器会自动给生成。你可以通过 HttpServletResponse 的 API 设置状态码：

常见的 HTTP 响应报文头属性

Cache-Control

响应输出到客户端后，服务端通过该报文头属告诉客户端如何控制响应内容的缓存。

下面，的设置让客户端对响应内容缓存 3600 秒，也即在 3600 秒内，如果客户再次访问该资源，直接从客户端的缓存中返回内容给客户，不要再从服务端获取（当然，这个功能是靠客户端实现的，服务端只是通过这个属性提示客户端“应该这么做”，做不做，还是决定于客户端，如果是自己宣称支持 HTTP 的客户端，则就应该这样实现）。

```
Cache-Control: max-age=3600
```

ETag

一个代表响应服务端资源（如页面）版本的报文头属性，如果某个服务端资源发生变化了，这个 ETag 就会相应发生变化。它是 Cache-Control 的有益补充，可以让客户端“更智能”地处理什么时候要从服务端取资源，什么时候可以直接从缓存中返回响应。

关于 ETag 的说明，你可以参见：http://en.wikipedia.org/wiki/HTTP_ETag。

Location

我们在 JSP 中让页面 Redirect 到一个某个 A 页面中，其实是让客户端再发一个请求到 A 页面，这个需要 Redirect 到的 A 页面的 URL，其实就是通过响应报文头的 Location 属性告知客户端的，如下的报文头属性，将使客户端 redirect 到 iteye 的首页中：

```
Location: http://www.iteye.com
```

Set-Cookie

服务端可以设置客户端的 Cookie，其原理就是通过这个响应报文头属性实现的：

```
Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1
```

其它 HTTP 响应报文头属性

更多其它的 HTTP 响应头报文，参见：http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

nginx

安装

安装依赖

安装 nginx 之前，确保系统已经安装 gcc、openssl-devel、pcre-devel 和 zlib-devel 软件库

- gcc 可以通过光盘直接选择安装
- openssl-devel、zlib-devel 可以通过光盘直接选择安装，https 时使用
- pcre-devel 安装 pcre 库是为了使 nginx 支持 HTTP Rewrite 模块

下载

nginx 下载

编译安装

通过上面的下载页下载最新的稳定版

```
#wget http://nginx.org/download/nginx-1.8.0.tar.gz
#tar xzvf nginx-1.8.0.tar.gz
#cd nginx-1.8.0
#./configure --prefix=/opt/X_nginx/nginx --with-http_ssl_module
#make && sudo make install
```

> - --prefix=/opt/X_nginx/nginx 安装目录
> - --with-http_ssl_module 添加 https 支持

编译时将 **ssl** 模块静态编译

```
./configure  --prefix=/opt/X_nginx/nginx \
             --with-openssl=../openssl-1.0.2l \
             --with-zlib=../zlib-1.2.11 \
             --with-pcre=../pcre-8.41 \
             --with-http_ssl_module
```

nginx 服务架构

模块化结构

> nginx 服务器的开发 `完全` 遵循模块化设计思想

模块化开发

1. 单一职责原则，一个模块只负责一个功能
2. 将程序分解，自顶向下，逐步求精
3. 高内聚，低耦合

nginx 的模块化结构

- 核心模块：nginx 最基本最核心的服务，如进程管理、权限控制、日志记录；
- 标准 HTTP 模块：nginx 服务器的标准 HTTP 功能；
- 可选 HTTP 模块：处理特殊的 HTTP 请求
- 邮件服务模块：邮件服务
- 第三方模块：作为扩展，完成特殊功能

nginx 的模块清单

- 核心模块
  - ngx_core
  - ngx_errlog
  - ngx_conf
  - ngx_events
  - ngx_event_core
  - ngx_epll
  - ngx_regex
- 标准 HTTP 模块
  - ngx_http
  - ngx_http_core #配置端口，URI 分析，服务器相应错误处理，别名控制 (alias) 等
  - ngx_http_log #自定义 access 日志
  - ngx_http_upstream #定义一组服务器，可以接受来自 proxy, Fastcgi,Memcache 的重定向；主要用作负载均衡
  - ngx_http_static
  - ngx_http_autoindex #自动生成目录列表
  - ngx_http_index #处理以 `/` 结尾的请求，如果没有找到 index 页，则看是否开启了 `random_index` ；如开启，则用之，否则用 autoindex
  - ngx_http_auth_basic #基于 http 的身份认证 (auth_basic)
  - ngx_http_access #基于 IP 地址的访问控制 (deny,allow)
  - ngx_http_limit_conn #限制来自客户端的连接的响应和处理速率
  - ngx_http_limit_req #限制来自客户端的请求的响应和处理速率
  - ngx_http_geo
  - ngx_http_map #创建任意的键值对变量
  - ngx_http_split_clients
  - ngx_http_referer #过滤 HTTP 头中 Referer 为空的对象
  - ngx_http_rewrite #通过正则表达式重定向请求
  - ngx_http_proxy
  - ngx_http_fastcgi #支持 fastcgi
  - ngx_http_uwsgi
  - ngx_http_scgi
  - ngx_http_memcached
  - ngx_http_empty_gif #从内存创建一个 1×1 的透明 gif 图片，可以快速调用
  - ngx_http_browser #解析 http 请求头部的 User-Agent 值
  - ngx_http_charset #指定网页编码
  - ngx_http_upstream_ip_hash
  - ngx_http_upstream_least_conn
  - ngx_http_upstream_keepalive
  - ngx_http_write_filter

    - ngx_http_header_filter
    - ngx_http_chunked_filter
    - ngx_http_range_header
    - ngx_http_gzip_filter
    - ngx_http_postpone_filter
    - ngx_http_ssi_filter
    - ngx_http_charset_filter
    - ngx_http_userid_filter
    - ngx_http_headers_filter #设置 http 响应头
    - ngx_http_copy_filter
    - ngx_http_range_body_filter
    - ngx_http_not_modified_filter
- 可选 HTTP 模块
    - ngx_http_addition #在响应请求的页面开始或者结尾添加文本信息
    - ngx_http_degradation #在低内存的情况下允许服务器返回 444 或者 204 错误
    - ngx_http_perl
    - ngx_http_flv #支持将 Flash 多媒体信息按照流文件传输，可以根据客户端指定的开始位置返回 Flash
    - ngx_http_geoip #支持解析基于 GeoIP 数据库的客户端请求
    - ngx_google_perftools
    - ngx_http_gzip #gzip 压缩请求的响应
    - ngx_http_gzip_static #搜索并使用预压缩的以.gz 为后缀的文件代替一般文件响应客户端请求
    - ngx_http_image_filter #支持改变 png，jpeg，gif 图片的尺寸和旋转方向
    - ngx_http_mp4 #支持.mp4,.m4v,.m4a 等多媒体信息按照流文件传输，常与 ngx_http_flv 一起使用
    - ngx_http_random_index #当收到 / 结尾的请求时，在指定目录下随机选择一个文件作为 index
    - ngx_http_secure_link #支持对请求链接的有效性检查
    - ngx_http_ssl #支持 https
    - ngx_http_stub_status
    - ngx_http_sub_module #使用指定的字符串替换响应中的信息
    - ngx_http_dav #支持 HTTP 和 WebDAV 协议中的 PUT/DELETE/MKCOL/COPY/MOVE 方法
    - ngx_http_xslt #将 XML 响应信息使用 XSLT 进行转换
- 邮件服务模块
    - ngx_mail_core
    - ngx_mail_pop3
    - ngx_mail_imap

    - ngx_mail_smtp
    - ngx_mail_auth_http
    - ngx_mail_proxy
    - ngx_mail_ssl
- 第三方模块
    - echo-nginx-module #支持在 nginx 配置文件中使用 echo/sleep/time/exec 等类 Shell 命令
    - memc-nginx-module
    - rds-json-nginx-module #使 nginx 支持 json 数据的处理
    - lua-nginx-module

nginx 的 web 请求处理机制

作为服务器软件，必须具备并行处理多个客户端的请求的能力， 工作方式主要以下 3 种：

- 多进程 (Apache)
    - 优点：设计和实现简单；子进程独立
    - 缺点：生成一个子进程要内存复制，在资源和时间上造成额外开销
- 多线程 (IIS)
    - 优点：开销小
    - 缺点：开发者自己要对内存进行管理；线程之间会相互影响
- 异步方式 (nginx)

经常说道异步非阻塞这个概念， 包含两层含义：

通信模式：

```
+ 同步：发送方发送完请求后，等待并接受对方的回应后，再发送下个请求
+ 异步：发送方发送完请求后，不必等待，直接发送下个请求
```

nginx 配置文件实例

```
#定义 nginx 运行的用户和用户组
user www www;

#nginx 进程数，建议设置为等于 CPU 总核心数。
worker_processes 8;

#nginx 默认没有开启利用多核 CPU, 通过增加 worker_cpu_affinity 配置参数来充分利用多核 CPU 以下是
 8 核的配置参数
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000 00100000 01000000 100
```

```
00000;

#全局错误日志定义类型，[ debug | info | notice | warn | error | crit ]
error_log /var/log/nginx/error.log info;

#进程文件
pid /var/run/nginx.pid;

#一个 nginx 进程打开的最多文件描述符数目，理论值应该是最多打开文件数（系统的值 ulimit -n）与 nginx 进程数相除，但是 nginx 分配请求并不均匀，所以建议与 ulimit -n 的值保持一致。
worker_rlimit_nofile 65535;

#工作模式与连接数上限
events
{
    #参考事件模型，use [ kqueue | rtsig | epoll | /dev/poll | select | poll ]; epoll 模型是 Linux 2.6 以上版本内核中的高性能网络 I/O 模型，如果跑在 FreeBSD 上面，就用 kqueue 模型。
    #epoll 是多路复用 IO(I/O Multiplexing) 中的一种方式，但是仅用于 linux2.6 以上内核，可以大大提高 nginx 的性能
    use epoll;

    ##########################################################################################
    #单个后台 worker process 进程的最大并发链接数
    #事件模块指令，定义 nginx 每个进程最大连接数，默认 1024。最大客户连接数由 worker_processes 和 worker_connections 决定
    #即 max_client=worker_processes*worker_connections, 在作为反向代理时：max_client=worker_processes*worker_connections / 4
    worker_connections 65535;
    ##########################################################################################
}

#设定 http 服务器
http {
    include mime.types; #文件扩展名与文件类型映射表
    default_type application/octet-stream; #默认文件类型
    #charset utf-8; #默认编码

    server_names_hash_bucket_size 128; #服务器名字的 hash 表大小
    client_header_buffer_size 32k; #上传文件大小限制
    large_client_header_buffers 4 64k; #设定请求缓
    client_max_body_size 8m; #设定请求缓
    sendfile on; #开启高效文件传输模式，sendfile 指令指定 nginx 是否调用 sendfile 函数来输出文件，对于普通应用设为 on，如果用来进行下载等应用磁盘 IO 重负载应用，可设置为 off，以平衡磁盘与网络 I/O 处理速度，降低系统的负载。注意：如果图片显示不正常把这个改成 off。
    autoindex on; #开启目录列表访问，合适下载服务器，默认关闭。
    tcp_nopush on; #防止网络阻塞
    tcp_nodelay on; #防止网络阻塞

    ##连接客户端超时时间各种参数设置##
    keepalive_timeout  120;          #单位是秒，客户端连接时时间，超时之后服务器端自动关闭该连接
 如果 nginx 守护进程在这个等待的时间里，一直没有收到浏览发过来 http 请求，则关闭这个 http 连接
    client_header_timeout 10;        #客户端请求头的超时时间
    client_body_timeout 10;          #客户端请求主体超时时间
```

```
    reset_timedout_connection on;     #告诉 nginx 关闭不响应的客户端连接。这将会释放那个客户端所占有的内存空间
    send_timeout 10;                  #客户端响应超时时间，在两次客户端读取操作之间。如果在这段时间内，客户端没有读取任何数据，nginx 就会关闭连接
    ##################################

    #FastCGI 相关参数是为了改善网站的性能：减少资源占用，提高访问速度。下面参数看字面意思都能理解。
    fastcgi_connect_timeout 300;
    fastcgi_send_timeout 300;
    fastcgi_read_timeout 300;
    fastcgi_buffer_size 64k;
    fastcgi_buffers 4 64k;
    fastcgi_busy_buffers_size 128k;
    fastcgi_temp_file_write_size 128k;

    ###作为代理缓存服务器设置#######
    ###先写到 temp 再移动到 cache
    #proxy_cache_path /var/tmp/nginx/proxy_cache levels=1:2 keys_zone=cache_one:512m inactive=10m max_size=64m;
    ###以上 proxy_temp 和 proxy_cache 需要在同一个分区中
    ###levels=1:2 表示缓存级别，表示缓存目录的第一级目录是 1 个字符，第二级目录是 2 个字符 keys_zone=cache_one:128m 缓存空间起名为 cache_one 大小为 512m
    ###max_size=64m 表示单个文件超过 128m 就不缓存了  inactive=10m 表示缓存的数据，10 分钟内没有被访问过就删除
    #########end####################

    #####对传输文件压缩############
    #gzip 模块设置
    gzip on; #开启 gzip 压缩输出
    gzip_min_length 1k; #最小压缩文件大小
    gzip_buffers 4 16k; #压缩缓冲区
    gzip_http_version 1.0; #压缩版本（默认 1.1，前端如果是 squid2.5 请使用 1.0）
    gzip_comp_level 2; #压缩等级，gzip 压缩比，1 为最小，处理最快；9 为压缩比最大，处理最慢，传输速度最快，也最消耗 CPU；
    gzip_types text/plain application/x-javascript text/css application/xml;
    #压缩类型，默认就已经包含 text/html，所以下面就不用再写了，写上去也不会有问题，但是会有一个 warn。
    gzip_vary on;
    ################################

    #limit_zone crawler $binary_remote_addr 10m; #开启限制 IP 连接数的时候需要使用

    upstream blog.ha97.com {
        #upstream 的负载均衡，weight 是权重，可以根据机器配置定义权重。weigth 参数表示权值，权值越高被分配到的几率越大。
        server 192.168.80.121:80 weight=3;
        server 192.168.80.122:80 weight=2;
        server 192.168.80.123:80 weight=3;
    }

    #虚拟主机的配置
    server {
        #监听端口
```

```
        listen 80;

        #############https##################
        #listen 443 ssl;
        #ssl_certificate /opt/https/xxxxxx.crt;
        #ssl_certificate_key /opt/https/xxxxxx.key;
        #ssl_protocols SSLv3 TLSv1;
        #ssl_ciphers HIGH:!ADH:!EXPORT57:RC4+RSA:+MEDIUM;
        #ssl_prefer_server_ciphers on;
        #ssl_session_cache shared:SSL:2m;
        #ssl_session_timeout 5m;
        ####################################end

        #域名可以有多个，用空格隔开
        server_name www.ha97.com ha97.com;
        index index.html index.htm index.php;
        root /data/www/ha97;
        location ~ .*.(php|php5)?$ {
            fastcgi_pass 127.0.0.1:9000;
            fastcgi_index index.php;
            include fastcgi.conf;
        }

        #图片缓存时间设置
        location ~ .*.(gif|jpg|jpeg|png|bmp|swf)$ {
            expires 10d;
        }

        #JS 和 CSS 缓存时间设置
        location ~ .*.(js|css)?$ {
            expires 1h;
        }

        #日志格式设定
        log_format access '$remote_addr - $remote_user [$time_local] "$request" ' '$st
atus $body_bytes_sent "$http_referer" ' '"$http_user_agent" $http_x_forwarded_for';

        #定义本虚拟主机的访问日志
        access_log /var/log/nginx/ha97access.log access;

        #对 "/" 启用反向代理
        location / {
            proxy_pass http://127.0.0.1:88;
            proxy_redirect off;
            proxy_set_header X-Real-IP $remote_addr;
            #后端的 Web 服务器可以通过 X-Forwarded-For 获取用户真实 IP
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            #以下是一些反向代理的配置，可选。
            proxy_set_header Host $host;
            client_max_body_size 10m; #允许客户端请求的最大单文件字节数
            client_body_buffer_size 128k; #缓冲区代理缓冲用户端请求的最大字节数，

            ##代理设置 以下设置是 nginx 和后端服务器之间通讯的设置##
```

```
            proxy_connect_timeout 90; #nginx 跟后端服务器连接超时时间（代理连接超时）
            proxy_send_timeout 90; #后端服务器数据回传时间（代理发送超时）
            proxy_read_timeout 90; #连接成功后，后端服务器响应时间（代理接收超时）
            proxy_buffering on;    #该指令开启从后端被代理服务器的响应内容缓冲 此参数开启后 pro
xy_buffers 和 proxy_busy_buffers_size 参数才会起作用
            proxy_buffer_size 4k;  #设置代理服务器（nginx）保存用户头信息的缓冲区大小
            proxy_buffers 4 32k;   #proxy_buffers 缓冲区，网页平均在 32k 以下的设置
            proxy_busy_buffers_size 64k; #高负荷下缓冲大小（proxy_buffers*2）
            proxy_max_temp_file_size 2048m; #默认 1024m, 该指令用于设置当网页内容大于 proxy
_buffers 时，临时文件大小的最大值。如果文件大于这个值，它将从 upstream 服务器同步地传递请求，而不是
缓冲到磁盘
            proxy_temp_file_write_size 512k; 这是当被代理服务器的响应过大时 nginx 一次性写入
临时文件的数据量。
            proxy_temp_path  /var/tmp/nginx/proxy_temp;    ##定义缓冲存储目录，之前必须要先
手动创建此目录
            proxy_headers_hash_max_size 51200;
            proxy_headers_hash_bucket_size 6400;
            ##############################################################
        }

        #设定查看 nginx 状态的地址
        location /nginxStatus {
            stub_status on;
            access_log on;
            auth_basic "nginxStatus";
            auth_basic_user_file conf/htpasswd;
            #htpasswd 文件的内容可以用 apache 提供的 htpasswd 工具来产生。
        }

        #本地动静分离反向代理配置
        #所有 jsp 的页面均交由 tomcat 或 resin 处理
        location ~ .(jsp|jspx|do)?$ {
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_pass http://127.0.0.1:8080;
        }

        #所有静态文件由 nginx 直接读取不经过 tomcat 或 resin
        location ~ .*.(htm|html|gif|jpg|jpeg|png|bmp|swf|ioc|rar|zip|txt|flv|mid|doc|p
pt|pdf|xls|mp3|wma)$
        { expires 15d; }

        location ~ .*.(js|css)?$
        { expires 1h; }
    }
}
```

nginx 服务器基础配置指令

nginx.conf 文件的结构

- Global: nginx 运行相关
- events: 与用户的网络连接相关
- http
  - http Global: 代理，缓存，日志，以及第三方模块的配置
  - server
    - server Global: 虚拟主机相关
    - location: 地址定向，数据缓存，应答控制，以及第三方模块的配置

> 所有的所有的所有的指令，都要以 ; 结尾

nginx 运行相关的 Global 部分

配置运行 nginx 服务器用户

user nobody nobody;

配置允许生成的 worker process 数

worker_processes auto; worker_processes 4;

> 这个数字，跟电脑 CPU 核数要保持一致

```
# grep ^proces /proc/cpuinfo
processor       : 0
processor       : 1
processor       : 2
processor       : 3
# grep ^proces /proc/cpuinfo | wc -l
4
```

配置 nginx 进程 PID 存放路径

pid logs/nginx.pid;

> 这里面保存的就是一个数字，nginx master 进程的进程号

配置错误日志的存放路径

error_log logs/error.log; error_log logs/error.log error;

配置文件的引入

include mime.types; include fastcgi_params; include ../../conf/*.conf;

与用户的网络连接相关的 events

设置网络连接的序列化

accept_mutex on;

> 对多个 nginx 进程接收连接进行序列化，防止多个进程对连接的争抢（惊群）

设置是否允许同时接收多个网络连接

multi_accept off;

事件驱动模型的选择

use select|poll|kqueue|epoll|rtsig|/dev/poll|eventport

> 这个重点，后面再看

配置最大连接数

worker_connections 512;

http

http Global 代理 - 缓存 - 日志 - 第三方模块配置

定义 MIME-Type

include mime.types; default_type application/octet-stream;

自定义服务日志

access_log logs/access.log main; access_log off;

配置允许 sendfile 方式传输文件

sendfile off;

sendfile on; sendfile_max_chunk 128k;

> nginx 每个 worker process 每次调用 sendfile() 传输的数据量的最大值

Refer:

- Linux kenel sendfile 如何提升性能
- nginx sendifle tcp_nopush tcp_nodelay 参数解释

配置连接超时时间

> 与用户建立连接后，nginx 可以保持这些连接一段时间，默认 75s 下面的 65s 可以被 Mozilla/Konqueror 识别，是发给用户端的头部信息 `Keep-Alive` 值

keepalive_timeout 75s 65s;

单连接请求数上限

> 和用户端建立连接后，用户通过此连接发送请求；这条指令用于设置请求的上限数

keepalive_requests 100;

server

配置网络监听

listen *:80* | :8000; # 监听所有的 80 和 8000 端口

listen 192.168.1.10:8000; listen 192.168.1.10; listen 8000; # 等同于 listen *:8000; listen 192.168.1.10 default_server backlog=511; # 该 ip 的连接请求默认由此虚拟主机处理；最多允许 1024 个网络连接同时处于挂起状态

基于名称的虚拟主机配置

server_name myserver.com www.myserver.com;

server_name *.myserver.com www.myserver.* myserver2.*; # 使用通配符

> 不允许的情况： server_name www.ab*d.com;* # ``只允许出现在 www 和 com 的位置

server_name ~^www\d+.myserver.com$; # 使用正则

> nginx 的配置中，可以用正则的地方，都以 `~` 开头
>
> from nginx~0.7.40 开始，server_name 中的正则支持 字符串捕获功能（capture）

server_name ~^www.(.+).com$; # 当请求通过 www.myserver.com 请求时， myserver 就被记录到 `$1` 中，在本 server 的上下文中就可以使用

如果一个名称 被多个虚拟主机的 server_name 匹配成功，那这个请求到底交给谁处理呢？看优先级：

1. 准确匹配到 server_name
2. 通配符在开始时匹配到 server_name
3. 通配符在结尾时匹配到 server_name
4. 正则表达式匹配 server_name
5. 先到先得

配置 **https** 证书

原理

https 是在 http 和 TCP 中间加上一层加密层

> - 浏览器向服务端发送消息时：本质上是浏览器（客户端）使用服务端的公钥来加密信息，服务端使用自己的私钥解密，
> - 浏览器从服务端获取消息是：服务端使用自己私钥加密，浏览器（客户端）使用服务端的公钥来解密信息

在这个过程中，需要保证服务端给浏览器的公钥不是假冒的。证明服务端公钥信息的机构是 CA（数字认证中心）

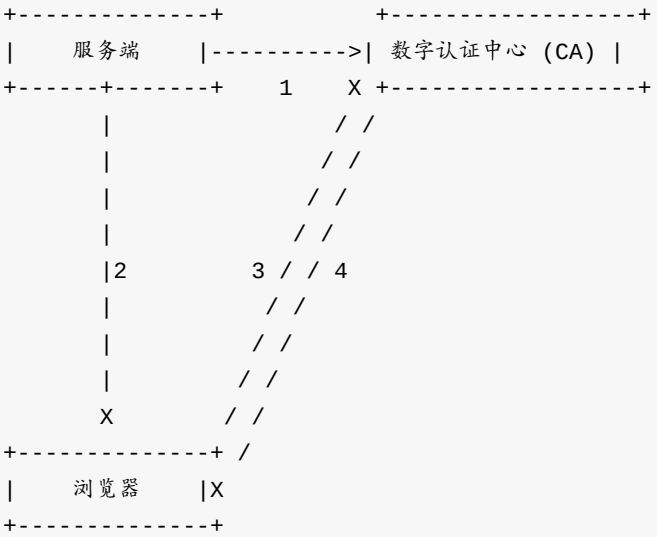
可以理解为：如果想证明一个人的身份是真的，就得证明这个人的身份证是真的

数字证书

```
数字证书相当于物理世界中的身份证，
在网络中传递信息的双方互相不能见面，利用数字证书可确认双方身份，而不是他人冒充的。
这个数字证书由信任的第三方，即认证中心使用自己的私钥对 A 的公钥加密，加密后文件就是网络上的身份证了，即数字证书
```

大致可以理解为如下

1．服务端将自己的公钥和其他信息（服务端数字证书），请求数字认证中心签名，数字认证中心使用自己的私钥在证书里加密（只有数字认证中心的公钥才能解开）
2．服务端将自己的证书（证书里面包括服务端的公钥）给浏览器
3．浏览器的“证书管理器”中有“受信任的根证书颁发机构”列表，客户端在接收到响应后，会在这个列表里查看是否存在解开该服务器数字证书的公钥。有两种错误情况：如果公钥在这个列表里，但是解码后的内容不匹配，说明证书被冒用；如果公钥不在这个列表里，说明这张证书不是受信任的机构所颁发，他的真实性无法确定
4．如果一切都没问题，浏览器就可以使用服务器的公钥对信息内容进行加密，然后与服务器交换信息（已加密）



只要证书（证书里有服务端的公钥）是可信的，公钥就是可信的。

证书格式

Linux 下的工具们通常使用 base64 编码的文本格式，相关常用后缀如下

- 证书
  - .crt
  - .pem
  - .cer(IIS 等一些平台下，则习惯用 cer 作为证书文件的扩展名，二进制证书）
- 私钥：.key
- 证书请求：.csr
- 其他
  - .keystore java 密钥库（包括证书和私钥）

制作证书

```
1. 生成服务器端的私钥 (key 文件)
$openssl genrsa  -out server.key 1024

2. 生成服务器端证书签名请求文件 (csr 文件);
$ openssl req -new -key server.key -out server.csr

...
Country Name:CN------------ 证书持有者所在国家
State or Province Name:BJ-- 证书持有者所在州或省份（可省略不填）
Locality Name:BJ----------- 证书持有者所在城市（可省略不填）
Organization Name:SC------- 证书持有者所属组织或公司
Organizational Unit Name:.- 证书持有者所属部门（可省略不填）
Common Name :ceshi.com----- 域名
Email Address:------------- 邮箱（可省略不填）

A challenge password:------ 直接回车
An optional company name:-- 直接回车


3. 生成证书文件 (crt 文件)
$ openssl x509 -req -days 1000 -in server.csr -signkey server.key -out server.crt
```

以上生成 server.crt server.key 文件即是用于 HTTPS 配置的证书和 key

如果想查看证书里面的内容，可以通过 $openssl x509 -in server.crt -text -noout 查看

配置 **nginx**

在 nginx 的 server 区域内添加如下

```
listen 443 ssl;
ssl_certificate /opt/https/server.crt;
ssl_certificate_key /opt/https/server.key;
ssl_protocols SSLv3 TLSv1;
ssl_ciphers HIGH:!ADH:!EXPORT57:RC4+RSA:+MEDIUM;
ssl_prefer_server_ciphers on;
ssl_session_cache shared:SSL:2m;
ssl_session_timeout 5m;
```

基于 **IP** 的虚拟主机配置

> 基于 IP 的虚拟主机，需要将网卡设置为同时能够监听多个 IP 地址

```
ifconfig
# 查看到本机 IP 地址为 192.168.1.30
ifconfig eth1:0 192.168.1.31 netmask 255.255.255.0 up
ifconfig eth1:1 192.168.1.32 netmask 255.255.255.0 up
ifconfig
# 这时就看到 eth1 增加来 2 个别名， eth1:0 eth1:1

# 如果需要机器重启后仍保持这两个虚拟的 IP
echo "ifconfig eth1:0 192.168.1.31 netmask 255.255.255.0 up" >> /etc/rc.local
echo "ifconfig eth1:0 192.168.1.32 netmask 255.255.255.0 up" >> /etc/rc.local
```

再来配置基于 IP 的虚拟主机

```
http {
    ...
    server {
     listen 80;
     server_name 192.168.1.31;
     ...
    }
    server {
     listen 80;
     server_name 192.168.1.32;
     ...
    }
}
```

配置 **location** 块

> location 块的配置，应该是最常用的了

location [= | ~ | ~* | ^~] uri {...}

这里内容分 2 块，匹配方式和 uri， 其中 uri 又分为 标准 uri 和正则 uri

先不考虑 那 4 种匹配方式

1. nginx 首先会再 server 块的多个 location 中搜索是否有 `标准 uri` 和请求字符串匹配， 如果有，记录匹配度最高的一个；
2. 然后，再用 location 块中的 `正则 uri` 和请求字符串匹配， 当第一个 `正则 uri` 匹配成功，即停止搜索， 并使用该 location 块处理请求；
3. 如果，所有的 `正则 uri` 都匹配失败，就使用刚记录下的匹配度最高的一个 `标准 uri` 处理请求
4. 如果都失败了，那就失败喽

再看 4 种匹配方式：

- `=`：用于 `标准 uri` 前，要求请求字符串与其严格匹配，成功则立即处理
- `^~`：用于 `标准 uri` 前，并要求一旦匹配到，立即处理，不再去匹配其他的那些个 `正则 uri`
- `~`：用于 `正则 uri` 前，表示 uri 包含正则表达式，并区分大小写
- `~*`：用于 `正则 uri` 前，表示 uri 包含正则表达式，不区分大小写

> `^~` 也是支持浏览器编码过的 URI 的匹配的哦，如 `/html/%20/data` 可以成功匹配 `/html/ /data`

[root] 配置请求的根目录

Web 服务器收到请求后，首先要在服务端指定的目录中寻找请求资源

```
root /var/www;
```

root 后跟的指定目录是上级目录

该上级目录下要含有和 location 后指定名称的同名目录才行，末尾“/”加不加无所谓

```
location /c/ {
      root /a/
}
```

访问站点 http://location/c 访问的就是 /a/c 目录下的站点信息。

[alias] 更改 location 的 URI

除了使用 root 指明处理请求的根目录，还可以使用 alias 改变 location 收到的 URI 的请求路径

```
location ~ ^/data/(.+\.(htm|html))$ {
    alias /locatinotest1/other/$1;
}
```

alias 后跟的指定目录是准确的，并且末尾必须加**“/”**，否则找不到文件

```
location /c/ {
      alias /a/
}
```

访问站点 http://location/c 访问的就是 /a/ 目录下的站点信息。

【注】一般情况下，在 location / 中配置 root，在 location /other 中配置 alias 是一个好习惯。

设置网站的默认首页

index 指令主要有 2 个作用：

- 对请求地址没有指明首页的，指定默认首页
- 对一个请求，根据请求内容而设置不同的首页，如下：

```
location ~ ^/data/(.+)/web/$ {
    index index.$1.html index.htm;
}
```

设置网站的错误页面

error_page 404 /404.html; error_page 403 /forbidden.html; error_page 404 =301 /404.html;

```
location /404.html {
    root /myserver/errorpages/;
}
```

基于 IP 配置 nginx 的访问权限

```
location / {
    deny 192.168.1.1;
    allow 192.168.1.0/24;
    allow 192.168.1.2/24;
    deny all;
}
```

> 从 192.168.1.0 的用户时可以访问的，因为解析到 allow 那一行之后就停止解析了

基于密码配置 nginx 的访问权限

auth_basic "please login"; auth_basic_user_file /etc/nginx/conf/pass_file;

> 这里的 file 必须使用绝对路径，使用相对路径无效

```
# /usr/local/apache2/bin/htpasswd -c -d pass_file user_name
# 回车输入密码，-c 表示生成文件，-d 是以 crypt 加密。

name1:password1
name2:password2:comment
```

> 经过 basic auth 认证之后没有过期时间，直到该页面关闭；如果需要更多的控制，可以使用 HttpAuthDigestModule http://wiki.nginx.org/HttpAuthDigestModule

应用

架设简单文件服务器

将 /data/public/ 目录下的文件通过 nginx 提供给外部访问

```
#mkdir /data/public/
#chmod 777 /data/public/
```

```
worker_processes 1;
error_log logs/error.log info;
events {
    use epoll;
}
http {
    server {
        # 监听 8080 端口
        listen 8080;
        location /share/ {
            # 打开自动列表功能，通常关闭
            autoindex on;
            # 将 /share/ 路径映射至 /data/public/，请保证 nginx 进程有权限访问 /data/public/
            alias /data/public/;
        }
    }
}
```

nginx 正向代理

- 正向代理指代理客户端访问服务器的一个中介服务器，代理的对象是客户端。正向代理就是代理服务器替客户端去访问目标服务器
- 反向代理指代理后端服务器响应客户端请求的一个中介服务器，代理的对象是服务器。

1. 配置

代理服务器配置

nginx.conf

```
server{
    resolver x.x.x.x;
#       resolver 8.8.8.8;
    listen 82;
    location / {
            proxy_pass http://$http_host$request_uri;
    }
    access_log  /data/httplogs/proxy-$host-aceess.log;
}
```

location 保持原样即可，根据自己的配置更改 listen port 和 dnf 即 resolver 验证： 在需要访问外网的机器上执行以下操作之一即可：

```
1. export http_proxy=http://yourproxyaddress：proxyport（建议）
2. vim ~/.bashrc
    export http_proxy=http://yourproxyaddress：proxyport
```

2 不足 nginx 不支持 CONNECT 方法，不像我们平时用的 GET 或者 POST，可以选用 apache 或 squid 作为代替方案。

nginx 服务器基础配置实例

```
user nginx nginx;

worker_processes 3;

error_log logs/error.log;
pid myweb/nginx.pid;

events {
    use epoll;
    worker_connections 1024;
}

http {
    include mime.types;
    default_type applicatioin/octet-stream;

    sendfile on;

    keepalive_timeout 65;
```

```
    log_format access.log '$remote_addr [$time_local] "$request" "$http_user_agent"';

    server {
        listen 8081;
        server_name myServer1;

        access_log myweb/server1/log/access.log;
        error_page 404 /404.html;

        location /server1/location1 {
            root myweb;
            index index.svr1-loc1.htm;
        }

        location /server1/location2 {
            root myweb;
            index index.svr1-loc2.htm;
        }
    }

    server {
        listen 8082;
        server_name 192.168.0.254;

        auth_basic "please Login:";
        auth_basic_user_file /opt/X_nginx/nginx/myweb/user_passwd;

        access_log myweb/server2/log/access.log;
        error_page 404 /404.html;

        location /server2/location1 {
            root myweb;
            index index.svr2-loc1.htm;
        }

        location /svr2/loc2 {
            alias myweb/server2/location2/;
            index index.svr2-loc2.htm;
        }

        location = /404.html {
            root myweb/;
            index 404.html;
        }
    }
}
```

```
#./sbin/nginx -c conf/nginx02.conf
nginx: [warn] the "user" directive makes sense only if the master process runs with su
per-user privileges, ignored in /opt/X_nginx/nginx/conf/nginx02.conf:1
.
├── 404.html
├── server1
│   ├── location1
│   │   └── index.svr1-loc1.htm
│   ├── location2
│   │   └── index.svr1-loc2.htm
│   └── log
│       └── access.log
└── server2
    ├── location1
    │   └── index.svr2-loc1.htm
    ├── location2
    │   └── index.svr2-loc2.htm
    └── log
        └── access.log

8 directories, 7 files
```

测试 **myServer1** 的访问

```
http://myserver1:8081/server1/location1/
this is server1/location1/index.svr1-loc1.htm

http://myserver1:8081/server1/location2/
this is server1/location1/index.svr1-loc2.htm
```

测试 **myServer2** 的访问

```
http://192.168.0.254:8082/server2/location1/
this is server2/location1/index.svr2-loc1.htm

http://192.168.0.254:8082/svr2/loc2/
this is server2/location1/index.svr2-loc2.htm

http://192.168.0.254:8082/server2/location2/
404 404 404 404
```

使用缓存

创建缓存目录

```
mkdir  /tmp/nginx_proxy_cache2
chmod 777 /tmp/nginx_proxy_cache2
```

修改配置文件

```
# http 区域下添加缓存区配置
proxy_cache_path /tmp/nginx_proxy_cache2 levels=1 keys_zone=cache_one:512m inactive=60
s max_size=1000m;

# server 区域下添加缓存配置
#缓存相应的文件（静态文件）
location ~ \.(gif|jpg|png|htm|html|css|js|flv|ico|swf)(.*) {
     proxy_pass http://IP: 端口；#如果没有缓存则通过 proxy_pass 转向请求
     proxy_redirect off;
     proxy_set_header Host $host;
     proxy_cache cache_one;
     proxy_cache_valid 200 302 1h;            #对不同的 HTTP 状态码设置不同的缓存时间，h 小
时，d 天数
     proxy_cache_valid 301 1d;
     proxy_cache_valid any 1m;
     expires 30d;
}
```

使用 **location** 反向代理到已有网站

```
location ~/bianque/(.*)$ {
        proxy_pass http://127.0.0.1:8888/$1/?$args;
    }
```

> - 加内置变量 $args 是保障 nginx 正则捕获 get 请求时不丢失，如果只是 post 请求，`$args` 是非必须的
> - `$1` 取自正则表达式部分()里的内容

其他

ngx_http_sub_module 替换响应中内容

- ngx_http_sub_module nginx 用来替换响应内容的一个模块（应用：有些程序中写死了端口，可以通过此工具将页面中的端口替换为其他端口）

配置 **http** 强制跳转 **https**

在 nginx 配置文件中的 server 区域添加如下内容

```
if ($scheme = 'http') {
    rewrite ^(.*)$ https://$host$uri;
}
```

django

env

```
python 2.6.6
django_1.4.22
```

【目录】

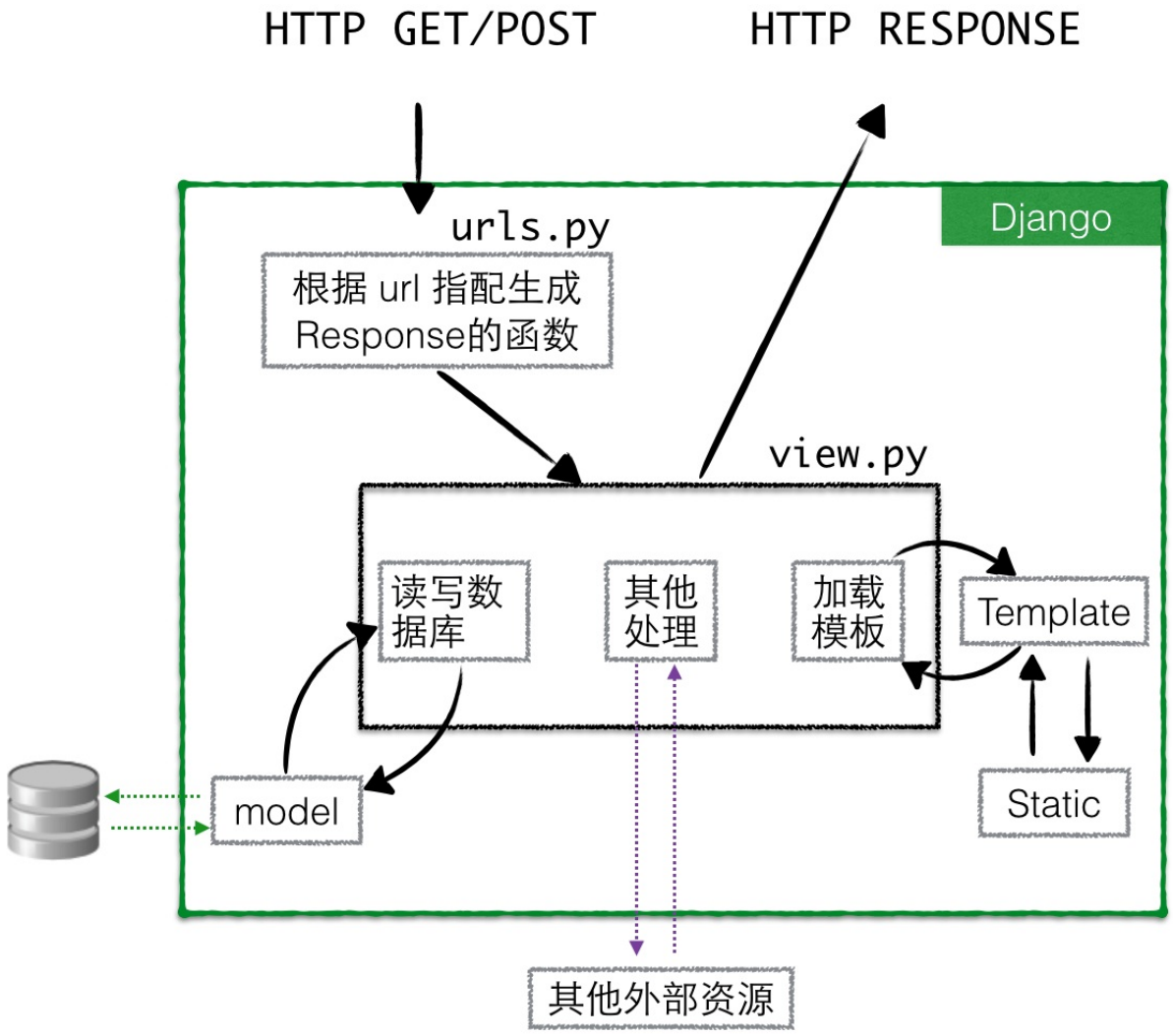
django 开始

django

Django 里是模型（Model）、模板 (Template) 和视图（Views），Django 也被称为 MTV 框架 。在 MTV 开发模式中：

- M 代表模型（Model），即数据存取层。 该层处理与数据相关的所有事务：如何存取、如何验证有效
- T 代表模板 (Template)，即表现层。 该层处理与表现相关的决定：如何在页面或其他类型文档中进行显示。
- V 代表视图（View），即业务逻辑层。 该层包含存取模型及调取恰当模板的相关逻辑。

可以把它看作模型与模板之间的桥梁。



我个人理解：可以把 Template 看作是含有变量的字符串，View 调用模板时，就是将变量传给 Template 的字符串，并将页面显示出来，具体如何显示不是咱们要关心的事，咱们只需要将变量传递给 template 即可

使用 bootstrap

django

settings

神奇的 Python 内部变量 **file** , 该变量被自动设置为代码所在的 Python 模块文件名。

```
import os.path
TEMPLATE_DIRS = (
    os.path.join(os.path.dirname(__file__), 'templates').replace('\\','/'),
)
```

or

```
import os.path
SITE_ROOT = os.path.abspath(os.path.dirname(__file__))

STATIC_ROOT = os.path.join(SITE_ROOT,'static')
STATICFILES_DIRS = (
    # Don't forget to use absolute paths, not relative paths.
    ("css", os.path.join(STATIC_ROOT,'css')),
    ("js", os.path.join(STATIC_ROOT,'js')),
    ("img", os.path.join(STATIC_ROOT, 'img')),
    ("font", os.path.join(STATIC_ROOT, 'font')),
    ("liger", os.path.join(STATIC_ROOT, 'liger')),
    ("bootstrap3", os.path.join(STATIC_ROOT, 'bootstrap3')),
)
TEMPLATE_DIRS = (
    os.path.join(SITE_ROOT,'templates'),
)
```

bootstrap

Bootstrap 的使用一般有两种方法。一种是引用在线的 Bootstrap 的样式，一种是将 Bootstrap 下载到本地进行引用。 使用本地的 Bootstrap

下载 Bootstrap 到本地进行解压，解压完成，你将得到一个 Bootstrap 目录，结构如下：

```
[root@Linux bootstrap-3.3.5-dist]# tree
.
├── css
│   ├── bootstrap.css
│   ├── bootstrap.css.map
│   ├── bootstrap.min.css
│   ├── bootstrap-theme.css
│   ├── bootstrap-theme.css.map
│   └── bootstrap-theme.min.css
├── fonts
│   ├── glyphicons-halflings-regular.eot
│   ├── glyphicons-halflings-regular.svg
│   ├── glyphicons-halflings-regular.ttf
│   ├── glyphicons-halflings-regular.woff
│   └── glyphicons-halflings-regular.woff2
└── js
    ├── bootstrap.js
    ├── bootstrap.min.js
    └── npm.js
```

本地调用如下：

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Hello Bootstrap</title>
        <!-- Bootstrap core CSS -->
        <link
            href="./bootstrap-3.3.5-dist/css/bootstrap.min.css" rel="stylesheet">
        <style type='text/css'>
            body {
                background-color: #CCC;
            }
        </style>
    </head>
    <body>
        <h1>hello Bootstrap<h1>
    </body>
</html>
```

django 登陆

flow chart

```
+---------+      +---------+       +------------+
| url.py  |----->| view.py |------>| templates  |
| (Login) |      | (Login) |       | login.html |
+---------+      +---------+       +------------+

在登陆页输入账号密码后，通过认证函数进行认证，如果认证通过跳转到主页，否则重新登陆
```

detailed

url.py

```
 url('^$','strap.view.LogIn'),
 url('^login/$','strap.view.LogIn'),         //login
 url('^index/$','strap.view.account_auth'),  //authentication
 url('^showDashboard/$','strap.view.show'),  //Go to the home page
```

view.py

```
  from django.http import HttpResponseRedirect, HttpResponse
  from django.shortcuts import render_to_response, render
  from django.contrib import auth
  import datetime
  def index(request):
      now = datetime.datetime.now()
      return render(request,'index.html')

  def LogIn(request):
      if request.user is not None:
          logout_view(request)
      return render(request,'login.html')

  def logout_view(request):
      user = request.user
      auth.logout(request)
      # Redirect to a success page.
      return HttpResponse("%s logged out!" % user)

  def account_auth(request):
      username = request.POST.get('username')
      password = request.POST.get('password')
      tri_user = auth.authenticate(username=username,password=password)
      if tri_user is not None:
      auth.login(request,tri_user)
      return HttpResponseRedirect('/showDashboard')
      else:
      return render_to_response('login.html',{'login_err':'Wrong username or password!
'})

  def show(request):
      return render(request,'index.html')
```

templates

```
  <!DOCTYPE html>
  <html lang="en">
    <head>
      <meta charset="utf-8">
      <meta http-equiv="X-UA-Compatible" content="IE=edge" />

      <title>Admin</title>
      <meta name="description" content="">
      <meta name="author" content="">

      <!-- http://davidbcalhoun.com/2010/viewport-metatag -->
      <meta name="HandheldFriendly" content="True">
      <meta name="MobileOptimized" content="320">
      <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-sc
ale=1.0, user-scalable=no">
```

```
      <!-- For Modern Browsers -->
      <link rel="shortcut icon" href="img/favicons/favicon.png">
      <!-- For everything else -->
      <link rel="shortcut icon" href="img/favicons/favicon.ico">
      <!-- For retina screens -->
      <link rel="apple-touch-icon-precomposed" sizes="114x114" href="img/favicons/appl
e-touch-icon-retina.png">
      <!-- For iPad 1-->
      <link rel="apple-touch-icon-precomposed" sizes="72x72" href="img/favicons/apple-
touch-icon-ipad.png">
      <!-- For iPhone 3G, iPod Touch and Android -->
      <link rel="apple-touch-icon-precomposed" href="img/favicons/apple-touch-icon.png
">

      <!-- iOS web-app metas -->
      <meta name="apple-mobile-web-app-capable" content="yes">
      <meta name="apple-mobile-web-app-status-bar-style" content="black">
      <!-- Add your custom home screen title: -->
      <meta name="apple-mobile-web-app-title" content="Jarvis">

      <!-- Startup image for web apps -->
      <link rel="apple-touch-startup-image" href="img/splash/ipad-landscape.png" media
="screen and (min-device-width: 481px) and (max-device-width: 1024px) and (orientation
:landscape)">
      <link rel="apple-touch-startup-image" href="img/splash/ipad-portrait.png" media=
"screen and (min-device-width: 481px) and (max-device-width: 1024px) and (orientation:
portrait)">
      <link rel="apple-touch-startup-image" href="img/splash/iphone.png" media="screen
 and (max-device-width: 320px)">

      <link type="text/css" rel="stylesheet" href="/static/css/login.css"></link>
      <link type="text/css" rel="stylesheet" href="/static/bootstrap3/css/bootstrap.cs
s"></link>
  </head>

  <body class="eternity-form scroll-animations-activated">
  <section data-panel="sixth" id="loginPage" class="colorBg6 colorBg dark active">
          <div class="container">
          <div class="login-form-section">
              <br>
              <br>
              <br>

              <div data-animation="bounceInLeft" class="forgot-password-section animat
ed bounceInLeft">
                  <div class="section-title">
                     <h3>Login</h3>
                  </div>

                  <div class="forgot-content">
                      <form method ="post" id="target" action="/index/">
                          <div class="textbox-wrap">
                              <div class="input-group">
```

```
                                <span class="input-group-addon "><i
                                    class="glyphicon glyphicon-user"></i></span> <in
put type="text"
                                    placeholder="Username" name="username" class="fo
rm-control" required="required">
                            </div>
                        </div>
                        <div class="textbox-wrap">
                            <div class="input-group">
                               <span class="input-group-addon "><i
                                    class="glyphicon glyphicon-lock"></i></span> <in
put type="password"
                                    placeholder="Password" name="password" class="fo
rm-control "
                                    required="required">
                            </div>
                        </div>
                        <div class="forget-form-action clearfix">
                           <button class="btn btn-success pull-right green-btn"
                                type="submit">
                                LogIn   <i class="glyphicon glyphicon-chevron-r
ight"></i>
                            </button>
                        </div>
                    </form>
                </div>
            </div>
          </div>
          </div>
      </section>
      <script type="text/javascript" src="/static/js/jquery-1.9.1.js"></script>
      <script type="text/javascript" src="/static/bootstrap3/js/bootstrap.min.js"></sc
ript>

      <script type="text/javascript">
          $(function() {
              $(".form-control").focus(function() {
                  $(this).closest(".textbox-wrap").addClass("focused");
              }).blur(function() {
                  $(this).closest(".textbox-wrap").removeClass("focused");
              });

              if('{{login_err}}') {
                  var html = '<div data-animation-delay=".2s" data-animation="fadeInRi
ghtBig" class="login-form-links link1 animated fadeInRightBig"
style="animation-delay: 0.2s;"><h4 class="blue">{{login_err}}</h4></div>'
                  $('div.login-form-section').append(html);
              }
          });
      </script>
  </body>
  </html>
```

django admin 密码重置

```
python manage.py shell


然后获取你的用户名，并且重设密码：
from django.contrib.auth.models import User
user = User.objects.get(username='admin')
user.set_password('new_password')
user.save()
```

这样就可以使用新密码进行登陆了

django 输出到固定日志

将 log 封装成一个单独的 app

```
[root@Linux mysite]# django-admin.py startapp log
[root@Linux mysite]# cd log
[root@Linux log]# ls
__init__.py  models.py  tests.py  views.py
```

编写 log 程序

```
curl -o BLog.py https://raw.githubusercontent.com/meetbill/MyPythonLib/master/log_utils/BLog/BLog.py
```

在本项目其他应用中的 view.py 中调用 BLog

```
from django.shortcuts import render,render_to_response

from log.BLog import Log
# 是否在终端显示
debug=True
# 日志文件
logpath = "/tmp/test.log"
# 设置日志文件为 5Mb 时进行轮转，并且最多只保留个日志
logger = Log(logpath,level="debug",is_console=debug, mbs=5, count=5)


def face(request):
    logstr="########"
    logger.error(logstr)
    logger.info(logstr)
    logger.warn(logstr)
    return render_to_response('register.html',{})
```

测试

当在 view.py 中设置了终端显示时

注：如果修改前 django 项目是运行的，那么当我们在程序中加入导入模块的程序时，需要重启下 django 应用，如果我们修改的程序不涉及导入模块部分，则不需要重启应用

django FAQ

django (1)Django 表单提交出现 CSRF verification failed. Request aborted

```
由于我们创建一个 POST 表单（它具有修改数据的作用），所以我们需要小心跨站点请求伪造。
谢天谢地，你不必太过担心，因为 Django 已经拥有一个用来防御它的非常容易使用的系统。
简而言之，所有针对内部 URL 的 POST 表单都应该使用{% csrf_token %}模板标签。
[templates-form]
{% csrf_token %}

[View]
return render_to_response('polls/detail.html', {'poll': p},
                  context_instance=RequestContext(request))
```

django 自用开发模板

pine

监控篇

> - zabbix
> - monit

zabbix

快速安装

> 物理机上安装 docker部署

zabbix 模板

zabbix_templates

zabbix 管理工具

zabbix_manager

目录

概述

Monit 是 Unix 系统中用于管理和监控进程、程序、文件、目录和文件系统的工具。使用 Monit 可以检测进程是否正常运行，如果异常可以自动重启服务以及报警，当然，也可以使用 Monit 检查文件和目录是否发生修改，例如时间戳、校验和以及文件大小的改变。

> - 官方网址：http://mmonit.com/monit/
> - 源代码包：http://mmonit.com/monit/dist/
> - 二进制包：http://mmonit.com/monit/dist/binary/

常用操作

Monit 默认的配置文件是 `~/.monitrc` ，如果没有该文件，则使用 `/etc/monitrc` 文件。在启动 Monit 的时候，可以指定使用的配置文件：

```
$ monit -c /var/monit/monitrc
```

在第一次启动 **monit** 的使用，可以使用如下命令测试配置文件（控制文件）是否正确

```
$ monit -t
$ Control file syntax OK
```

如果配置文件没有问题的话，就可以使用 `monit` 命令启动 **monit** 了。

```
$ monit
```

> 当启动 **monit** 的时候，可以使用命令行选项控制它的行为，命令行提供的选项优先于配置文件中的配置。

支持命令行的选项

下列是 **monit** 支持的选项

- **-c** 指定要使用的配置文件
- **-d n** 每隔 n 秒以守护进程的方式运行 monit 一次，在配置文件中使用 [`set daemon`] 进行配置
- **-g name** 设置用于 `start` , `stop` , `restart` , `monitor` , `unmonitor` 动作的组名
- **-l logfile** 指定日志文件，[`set logfile`]
- **-p pidfile** 在守护进程模式使用锁文件，配置文件中使用 [`set pidfile`] 指令
- **-s statefile** 将状态信息写入到该文件，[`set statefile`]
- **-I** 不要以后台模式运行（需要从 init 运行）
- **-i** 打印 monit 的唯一 ID
- **-r** 重置 monit 的唯一 ID
- **-t** 检查配置文件语法是否正确
- **-v** 详细模式，会输出针对信息
- **-vv** 非常详细的模式，会打印出现错误的堆栈信息
- **-H [filename]** 打印文件的 MD5 和 SHA1 哈希值 Print MD5 and SHA1，如果没有提供文件名，则为标准输入
- **-V** 打印版本号
- **-h** 打印帮助信息

命令行参数

当 **Monit** 以守护进程运行的时候，可以使用下列的参数连接它的守护进程（默认是 TCP 的 127.0.0.1:2812）使其执行请求的操作。

- `start all`

  启动配置文件中列出的所有的服务并且监控它们，如果使用 `-g` 选项提供了组选项，则只对该组有效。

- `start name`

  启动指定名称的服务并对其监控，服务名为配置文件中配置的服务条目名称。

- `stop all`

  与 `start all` 相对。

- `stop name`

与 `start name` 相对。

- `restart all`

  重启所有的 service

- `restart name`

  重启指定 service

- `monitor all`

  允许对配置文件中所有的服务进行监控

- `monitor name`

  允许对指定的 service 监控

- `unmonitor all`

  与 `monitor all` 相对

- `unmonitor name`

  与 `monitor name` 相对

- `status`

  打印每个服务的状态信息

- `reload`

  重新初始化 Monitor 守护进程，守护进程将会重载配置文件以及日志文件

- `quit`

  关闭所有 `monitor` 进程

- `validate`

  检查所有配置文件中的服务，当 Monitor 以守护进程运行的时候，这是默认的动作

- `procmatch regex`

  对符合指定模式的进程进行简单测试，该命令接受正则表达式作为参数，并且显示出符合该模式的所有进程。

配置文件

Monit 的配置文件叫做 `monitrc` 文件。默认为 `~/.monitrc` 文件，如果该文件不存在，则尝试 `/etc/monitrc` 文件，然后是 `@sysconfdir@/monitrc` ，最后是 `./monitrc` 文件。

> 这里所说的配置文件实际上就是控制文件（control file）

Monit 使用它自己的领域语言 (DSL) 进行配置，配置文件包含一系列的服务条目和全局配置项。

在语意上，配置文件包含以下三部分：

- 全局 set 指令

  该指令以 `set` 开始，后面跟着配置项

- 全局 include 指令

  该指令以 `include` 开头，后面是 glob 字符串，指定了要包含的配置文件位置

- 一个或多个服务条目指令

  每一个服务条目包含关键字 `check` ，后面跟着服务类型。每一条后面都需要跟着一个唯一的服务标识名称。monit 使用这个名称来引用服务以及与用户进行交互。

当前支持九种类型的 `check` 语句：

1. `CHECK PROCESS <unique name> <PIDFILE <path> | MATCHING <regex>>`

   这里的 `PIDFILE` 是进程的 PID 文件的绝对路径，如果 PID 文件不存在，如果定义了进程的 start 方法的话，会调用该方法。

   `MATCHING` 是可选的指定进程的方式，使用名称规则指定进程。

2. `CHECK FILE <unique name> PATH <path>`

   检查文件是否存在，如果指定的文件不存在，则调用 start 方法。

3. `CHECK FIFO <unique name> PATH <path>`
4. `CHECK FILESYSTEM <unique name> PATH <path>`
5. `CHECK DIRECTORY <unique name> PATH <path>`
6. `CHECK HOST <unique name> ADDRESS <host address>`
7. `CHECK SYSTEM <unique name>`
8. `CHECK PROGRAM <unique name> PATH <executable file> [TIMEOUT <number> SECONDS]`
9. `CHECK NETWORK <unique name> <ADDRESS <ipaddress> | INTERFACE <name>>`

日志

Monit 将会使用日志文件记录运行状态以及错误消息，在配置文件中使用 `set logfile` 指令指定日志配置。

如果希望使用自己的日志文件，使用下列指令：

```
set logfile /var/log/monit.log
```

如果要使用系统的 syslog 记录日志，使用下列指令：

```
set logfile syslog
```

如果不想开启日志功能，只需要注释掉该指令即可。

日志文件的格式为：

```
[date] priority : message
```

例如：

```
[CET Jan  5 18:49:29] info : 'mymachine' Monit started
```

守护进程模式

```
set daemon n (n 的单位是秒)
```

指定 Monit 在后台轮询检查进程运行状态的时间。你可以使用命令行参数 `-d` 选项指定这个时间，当然，建议在配置文件中进行设置。

Monit 应该总是以后台的守护进程模式运行，如果你不指定该选项或者是命令行的 `-d` 选项，则只会在运行 Monit 的时候对它监控的文件或者进程检查一次然后退出。

Init 支持

配置 `set init` 可以防止 monit 将自身转化为守护进程模式，它可以让前台进程运行。这需要从 init 运行 monit，另一种方式是使用 crontab 定时任务运行，当然这样的话你需要在运行前使用 `monit -t` 检查一下控制文件是否存在语法错误。

要配置 monit 从 init 运行，可以在 monit 的配置文件中使用 `set init` 指令或者命令行中使用 `-I` 选项，以下是需要在 `/etc/inittab` 文件中增加的配置。

```
# Run Monit in standard run-levels
mo:2345:respawn:/usr/local/bin/monit -Ic /etc/monitrc
```

> inittab 文件格式：`id:runlevels:action:process` 该行配置是为 Monit 指定了 id 为 mo，运行级别 2-5 有效，`respawn` 指明了无论进程是否已经运行，都对进程 restart

在修改完 init 配置文件后，可以使用如下命令测试 `/etc/inittab` 文件并运行 Monit:

```
telinit q
```

> `telinit q` 用于重载守护进程的配置，等价于 `systemctl daemon-reload`

对于没有 `telinit` 的系统，执行如下命令：

```
kill -1 1
```

如果 Monit 已经系统启动的时候运行对服务进行监控，在某些情况下，可能会出现竞争。也就是说如果一个服务启动的比较慢，Monit 会假设该服务没有运行并且可能会尝试启动该服务和报警，但是事实上该服务正在启动中或者已经在启动队列里了。

包含文件

```
include globstring
```

例如 `include /etc/monit.d/*.cfg` 会将 `/etc/monit.d/` 目录下所有的 `.cfg` 文件包含到配置文件中。

管理工具

monit_manager

原文：monit 官方文档 Version

存储篇

- 磁盘及 RAID
- DAS/SAN/NAS
- gfs
- Glusterfs
- ceph
- moosefs

磁盘及 RAID

磁盘管理

查看磁盘信息

lsscsi

```
--classic|-c      alternate output similar to 'cat /proc/scsi/scsi'
--device|-d       show device node's major + minor numbers
--generic|-g      show scsi generic device name
--help|-h         this usage information
--hosts|-H        lists scsi hosts rather than scsi devices
--kname|-k        show kernel name instead of device node name
--list|-L         additional information output one
                  attribute=value per line
--long|-l         additional information output
--protection|-p   show data integrity (protection) information
--sysfsroot=PATH|-y PATH    set sysfs mount point to PATH (def: /sys)
--transport|-t    transport information for target or, if '--hosts'
                  given, for initiator
--verbose|-v      output path names where data is found
--version|-V      output version string and exit
```

查看磁盘运行状态

```
lsscsi -l
[0:0:0:0]    disk    ATA      ST2000LM003 HN-M 0007  -
  state=running queue_depth=64 scsi_level=6 type=0 device_blocked=0 timeout=0
[0:0:1:0]    disk    ATA      ST2000LM003 HN-M 0007  -
  state=running queue_depth=64 scsi_level=6 type=0 device_blocked=0 timeout=0
[0:0:2:0]    disk    ATA      ST2000LM003 HN-M 0007  /dev/sda
  state=running queue_depth=64 scsi_level=6 type=0 device_blocked=0 timeout=30
[0:0:3:0]    disk    ATA      ST2000LM003 HN-M 0007  /dev/sdb
  state=running queue_depth=64 scsi_level=6 type=0 device_blocked=0 timeout=30
[0:0:4:0]    disk    ATA      ST2000LM003 HN-M 0007  /dev/sdc
  state=running queue_depth=64 scsi_level=6 type=0 device_blocked=0 timeout=30
[0:0:5:0]    disk    ATA      ST2000LM003 HN-M 0007  /dev/sdd
  state=running queue_depth=64 scsi_level=6 type=0 device_blocked=0 timeout=30
[0:0:6:0]    disk    ATA      ST2000LM003 HN-M 0007  /dev/sde
  state=running queue_depth=64 scsi_level=6 type=0 device_blocked=0 timeout=30
[0:0:7:0]    disk    ATA      ST2000LM003 HN-M 0006  /dev/sdf
  state=running queue_depth=64 scsi_level=6 type=0 device_blocked=0 timeout=30
[0:1:0:0]    disk    LSILOGIC Logical Volume   3000  /dev/sdg
  state=running queue_depth=64 scsi_level=3 type=0 device_blocked=0 timeout=30
```

smartctl

smartctl 可以查看磁盘的 SN，WWN 等信息。还有是否有磁盘坏道的信息

```
$ smartctl -a -f brief /dev/sdb

# 如果磁盘位于 RAID 下面，比如 megaraid，可以使用如下命令
# smartctl -a -f brief -d megaraid,1 /dev/sdb

smartctl 5.43 2012-06-30 r3573 [x86_64-linux-3.12.21-1.el6.x86_64] (local build)
Copyright (C) 2002-12 by Bruce Allen, http://smartmontools.sourceforge.net
```

```
=== START OF INFORMATION SECTION ===
Device Model:     ST2000LM003 HN-M201RAD
Serial Number:    S34RJ9EG109476
LU WWN Device Id: 5 0004cf 20eeefc42
Firmware Version: 2BC10007
User Capacity:    2,000,398,934,016 bytes [2.00 TB]
Sector Sizes:     512 bytes logical, 4096 bytes physical
Device is:        Not in smartctl database [for details use: -P showall]
ATA Version is:   8
ATA Standard is:  ATA-8-ACS revision 6
Local Time is:    Mon Jun 22 07:48:24 2015 UTC
SMART support is: Available - device has SMART capability.
SMART support is: Enabled

Vendor Specific SMART Attributes with Thresholds:
ID# ATTRIBUTE_NAME          FLAGS    VALUE WORST THRESH FAIL RAW_VALUE
  1 Raw_Read_Error_Rate     POSR-K   91   91   051    -    11787
  2 Throughput_Performance  -OS--K   252   252   000    -    0
  3 Spin_Up_Time            PO---K   086   086   025    -    4319
  4 Start_Stop_Count        -O--CK   100   100   000    -    16
  5 Reallocated_Sector_Ct   PO--CK   252   252   010    -    0
  7 Seek_Error_Rate         -OSR-K   252   252   051    -    0
  8 Seek_Time_Performance   --S--K   252   252   015    -    0
  9 Power_On_Hours          -O--CK   100   100   000    -    2277
 10 Spin_Retry_Count        -O--CK   252   252   051    -    0
 12 Power_Cycle_Count       -O--CK   100   100   000    -    34
191 G-Sense_Error_Rate      -O---K   252   252   000    -    0
192 Power-Off_Retract_Count -O---K   252   252   000    -    0
194 Temperature_Celsius     -O----   064   064   000    -    22 (Min/Max 18/28)
195 Hardware_ECC_Recovered  -O-RCK   100   100   000    -    0
196 Reallocated_Event_Count -O--CK   252   252   000    -    0
197 Current_Pending_Sector  -O--CK   100   100   000    -    11
198 Offline_Uncorrectable   ----CK   252   252   000    -    0
199 UDMA_CRC_Error_Count    -OS-CK   200   200   000    -    0
200 Multi_Zone_Error_Rate   -O-R-K   100   100   000    -    3
223 Load_Retry_Count        -O--CK   252   252   000    -    0
225 Load_Cycle_Count        -O--CK   090   090   000    -    108289
                            ||||||_ K auto-keep
                            |||||__ C event count
                            ||||___ R error rate
                            |||____ S speed/performance
                            ||_____ O updated online
                            |______ P prefailure warning

SMART Error Log Version: 1
No Errors Logged
```

解释下各属性的含义

ID# ATTRIBUTE_NAME FLAG VALUE WORST THRESH TYPE UPDATED WHEN_FAILED RAW_VALUE

> - **ID** 属性 ID，1~255
> - **ATTRIBUTE_NAME** 属性名
> - **FLAG** 表示这个属性携带的标记。使用 -f brief 可以打印
> - **VALUE** Normalized value, 取值范围 1 到 254. 越低表示越差。越高表示越好。(with 1 representing the worst case and 254 representing the best)。注意 wiki 上说的是 1 到 253. 这个值是硬盘厂商根据 RAW_VALUE 转换来的，smartmontools 工具不负责转换工作。
> - **WORST** 表示 SMART 开启以来的，所有 Normalized values 的最低值。(which represents the lowest recorded normalized value.)
> - **THRESH** 阈值，当 Normalized value 小于等于 THRESH 值时，表示这项指标已经 failed 了。注意这里提到，如果这个属性是 pre-failure 的，那么这项如果出现 Normalized value<=THRESH, 那么磁盘将马上 failed 掉
> - **TYPE** 这里存在两种 TYPE 类型，Pre-failed 和 Old_age.
>   1. Pre-failed 类型的 Normalized value 可以用来预先知道磁盘是否要坏了。例如 Normalized value 接近 THRESH 时，就赶紧换硬盘吧。
>   2. Old_age 类型的 Normalized value 是指正常的使用损耗值，当 Normalized value 接近 THRESH 时，也需要注意，但是比 Pre-failed 要好一点。
> - **UPDATED** 这个字段表示这个属性的值在什么情况下会被更新。一种是通常的操作和离线测试都更新 (Always), 另一种是只在离线测试的情况下更新 (Offline).
> - **WHEN_FAILED** 这字段表示当前这个属性的状态 : failing_now(normalized_value <= THRESH), 或者 in_the_past(WORST <= THRESH), 或者 - , 正常 (normalized_value 以及 wrost >= THRESH).
> - **RAW_VALUE** 表示这个属性的未转换前的 RAW 值，可能是计数，也可能是温度，也可能是其他的。 注意 RAW_VALUE 转换成 Normalized value 是由厂商的 firmware 提供的，smartmontools 不提供转换。

注意有个 FLAG 是 KEEP, 如果不带这个 FLAG 的属性，值将不会 KEEP 在磁盘中，可能出现 WORST 值被刷新的情况，例如这里的 ID=1 的值，已经 89 了，重新执行又变成 91 了，但是 WORST 的值并不是历史以来的最低 89。遇到这种情况的解决办法是找个地方存储这些值的历史值。

因此监控磁盘的重点在哪里呢？严重情况从上到下：

- 1. 最严重的情况 WHEN_FAILED = FAILING_NOW 并且 TYPE=Pre-failed, 表示现在这个属性已经出问题了。并且硬盘也已经 failed 了。
- 1. 次严重的情况 WHEN_FAILED = in_the_past 并且 TYPE=Pre-failed, 表示这个属性曾经出问题了。但是现在是正常的。
- 1. WHEN_FAILED = FAILING_NOW 并且 TYPE=Old_age, 表示现在这个属性已经出问题了。但是硬盘可能还没有 failed.
- 1. WHEN_FAILED = in_the_past 并且 TYPE=Old_age, 表示现在这个属性曾经出问题了。但是现在是正常的。

为了避免这 4 种情况的发生。

- 1. 对于 UPDATE=Offline 的属性，应该让 smartd 定期进行测试 (smartd 还可以发邮件）. 或者 crontab 进行测试。
- 1. 应该时刻关注磁盘的 Normalized value 以及 WORST 的值是否接近 THRESH 的值了。当有值要接近 THRESH 了，提前更换硬盘。
- 1. 温度，有些磁盘对温度比较敏感，例如 PCI-E SSD 硬盘。如果温度过高可能就挂了。这里读取 RAW_VALUE 就比较可靠了。

各个属性的含义

- **read error rate** 错误读取率：记录读取数据错误次数（累计），非 0 值表示硬盘已经或者可能即将发生坏道
- **throughput performance** 磁盘吞吐量：平均吞吐性能（一般在进行了人工 Offline S.M.A.R.T. 测试以后才会有值。）；
- **spinup time** 主轴电机到达要求转速时间（毫秒 / 秒）；
- **start/stop count** 电机启动 / 停止次数（可以当作开机 / 关机次数，或者休眠后恢复，均增加一次计数。全新的硬盘应该小于 10）；
- **reallocated sectors count** 重分配扇区计数：硬盘生产过程中，有一部分扇区是保留的。当一些普通扇区读 / 写 / 验证错误，则重新映射到保留扇区，挂起该异常扇区，并增加计数。随着计数增加，io 性能骤降。如果数值不为 0，就需要密切关注硬盘健康状况；如果持续攀升，则硬盘已经损坏；如果重分配扇区数超过保留扇区数，将不可修复；
- **seek error rate** 寻道错误率：磁头定位错误一次，则技术增加一次。如果持续攀升，则可能是机械部分即将发生故障；
- **seek timer performance** 寻道时间：寻道所需要的时间，越短则读取数据越快，但是如果时间增加，则可能机械部分即将发生故障；
- **power-on time** 累计通电时间：指硬盘通电时间累计值。（单位：天 / 时 / 分 / 秒。休眠 / 挂起不计入？新购入的硬盘应小于 100hrs）；
- **spinup retry count** 电机启动失败计数：电机启动到指定转速失败的累计数值。如果失败，则可能是动力系统产生故障；
- **power cycle count** 电源开关计数：每次加电增加一次计数，新硬盘应小于 10 次；
- **g-sensor error rate** 坠落计数：异常加速度（例如坠落，抛掷）计数——磁头会立即回到 landing zone，并增加一次计数；
- **power-off retract count** 异常断电次数：磁头在断电前没有完全回到 landing zone 的次数，每次异常断电则增加一次计数；
- **load/unload cycle count** 磁头归位次数：指工作时，磁头每次回归 landing zone 的次数。（ps：流言说某个 linux 系统——不点名，在使用电池时候，会不断强制磁头归为，而磁头归位次数最大值约为 600k 次，所以认为 linux 会损坏硬盘，实际上不是这样的）；
- **temperature** 温度：没嘛好说的，硬盘温度而已，理论上比工作环境高不了几度。（sudo hddtemp /dev/sda）
- **reallocetion event count** 重映射扇区操作次数：上边的重映射扇区还记得吧？这个就是操作次数，成功的，失败的都计数。成功好说，也许硬盘有救，失败了，也许硬盘就要报废了；
- **current pending sector count** 待映射扇区数：出现异常的扇区数量，待被映射的扇区数量。 如果该异常扇区之后成功读写，则计数会减小，扇区也不会重新映射。读错误不会重新映射，只有写错误才会重新映射；
- **uncorrectable sector count** 不可修复扇区数：所有读 / 写错误计数，非 0 就证明有坏道，硬盘报废；

对于 SSD 硬盘，需要关注的几个参数

SSD 磨损数据分析： SLC 的 SSD 可以擦除 10 万次，MLC 的 SSD 可以擦除 1 万次

Media Wearout Indicator

定义：表示 SSD 上 NAND 的擦写次数的程度，初始值为 100，随着擦写次数的增加，开始线性递减，递减速度按照擦写次数从 0 到最大的比例。一旦这个值降低到 1，就不再降了，同时表示 SSD 上面已经有 NAND 的擦写次数到达了最大次数。这个时候建议需要备份数据，以及更换 SSD。

解释：直接反映了 SSD 的磨损程度，100 为初始值，0 为需要更换，有点类似游戏中的血点。

结果：磨损 1 点

Re-allocated Sector Count

定义：出厂后产生的坏块个数，如果有坏块，从 1 开始增加，每 4 个坏块增加 1

解释：坏块的数量间接反映了 SSD 盘的健康状态。

结果：基本上都为 0

Host Writes Count

定义：主机系统对 SSD 的累计写入量，每写入 65536 个扇区 raw value 增加 1

解释：SSD 的累计写入量，写入量越大，SSD 磨损情况越严重。每个扇区大小为 512bytes，65536 个扇区为 32MB

结果：单块盘 40T

Timed Workload Media Wear

定义：表示在一定时间内，盘片的磨损比例，比 Media Wearout Indicator 更精确。

解释：可以在测试前清零，然后测试某段时间内的磨损数据，这个值的 1 点相当于 Media Wearout Indicator 的 1/100，测试时间必须大于 60 分钟。另外两个相关的参数：Timed Workload Timer 表示单次测试时间，Timed Workload Host Read/Write Ratio 表示读写比例。

Available_Reservd_Space

SSD 上剩余的保留空间，初始值为 100，表示 100%，阀值为 10，递减到 10 表示保留空间已经不能再减少

MegaCli

查看 media error, other error

LSIUtil

查看磁盘的物理位置，error 检测

lsblk

磁盘扩展

Linux 下 XFS 扩展

XFS 是一个开源的（GPL）日志文件系统，最初由硅谷图形（SGI）开发，现在大多数的 Linux 发行版都支持。事实上，XFS 已被最新的 CentOS/RHEL 7 采用，成为其默认的文件系统。在其众多的特性中，包含了“在线调整大小”这一特性，使得现存的 XFS 文件系统在已经挂载的情况下可以进行扩展。

扩展前

```
[root@meetbill ~]# xfs_info /mnt/
meta-data=/dev/sdb    isize=512    agcount=4, agsize=196608 blks
         =            sectsz=512   attr=2, projid32bit=1
         =            crc=1        finobt=0 spinodes=0
data     =            bsize=4096   blocks=786432, imaxpct=25
         =            sunit=0      swidth=0 blks
naming   =version 2   bsize=4096   ascii-ci=0 ftype=1
log      =internal    bsize=4096   blocks=2560, version=2
         =            sectsz=512   sunit=0 blks, lazy-count=1
realtime =none        extsz=4096   blocks=0, rtextents=0


[root@meetbill ~]# df -h
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/cl-root   17G  8.4G  8.7G  49% /
devtmpfs             902M     0  902M   0% /dev
tmpfs                912M     0  912M   0% /dev/shm
tmpfs                912M  8.7M  904M   1% /run
tmpfs                912M     0  912M   0% /sys/fs/cgroup
/dev/sda1           1014M  141M  874M  14% /boot
tmpfs                183M     0  183M   0% /run/user/0
/dev/sdb             3.0G   33M  3.0G   2% /mnt
```

将磁盘 (/dev/sdb) 进行扩展后，扩展磁盘的方式比如虚拟机对虚拟磁盘进行扩展或 isics 对存储进行扩展，磁盘扩展后，我们还需要对文件系统进行扩展 (/mnt)

我们用到的是 `xfs_growfs` 命令

```
[root@meetbill ~]# xfs_growfs /mnt/
meta-data=/dev/sdb    isize=512    agcount=4, agsize=196608 blks
         =            sectsz=512   attr=2, projid32bit=1
         =            crc=1        finobt=0 spinodes=0
data     =            bsize=4096   blocks=786432, imaxpct=25
         =            sunit=0      swidth=0 blks
naming   =version 2   bsize=4096   ascii-ci=0 ftype=1
log      =internal    bsize=4096   blocks=2560, version=2
         =            sectsz=512   sunit=0 blks, lazy-count=1
realtime =none        extsz=4096   blocks=0, rtextents=0
data blocks changed from 786432 to 1310720
```

大功告成，如果 `xfs_growfs` 不加任何参数，则会对指定挂载目录自动扩展 XFS 文件系统到最大的可用大小。`-D` 参数可以设置为指定大小

测试硬盘

dd 测试硬盘读写速度

如何正确使用 dd 工具测试磁盘的 I/O 速度

一般情况下，我们都是使用 dd 命令创建一个大文件来测试磁盘的读写速度。我们分析一下 dd 命令是如何工作的。

```
dd if=/dev/zero of=/xiaohan/test.iso bs=1024M count=1
```

测试显示的速度是 dd 命令将数据写入到内存缓冲区中的速度

```
dd if=/dev/zero of=/xiaohan/test.iso bs=1024M count=1;sync
```

测试显示的跟上一种情况是一样的，两个命令是先后执行的，当 sync 开始执行的时候，dd 命令已经将速度信息打印到了屏幕上，仍然无法显示从内存写硬盘时的真正速度。

```
dd if=/dev/zero of=/xiaohan/test.iso bs=1024M count=1 conv=fdatasync
```

这种情况加入这个参数后，dd 命令执行到最后会真正执行一次“同步 (sync)”操作，所以这时候你得到的是读取这 128M 数据到内存并写入到磁盘上所需的时间，这样算出来的时间才是比较符合实际的。

```
dd if=/dev/zero of=/xiaohan/test.iso bs=1024M count=1 oflag=dsync
```

这种情况下，dd 在执行时每次都会进行同步写入操作。也就是说，这条命令每次读取 1M 后就要先把这 1M 写入磁盘，然后再读取下面这 1M，一共重复 128 次。这可能是最慢的一种方式，基本上没有用到写缓存 (write cache)。

总结：

```
建议使用测试写速度的方式为：
dd if=/dev/zero of=/xiaohan/test.iso bs=1024M count=1 conv=fdatasync

建议使用测试读速度的方式为：
dd if=/xiaohan/test.iso of=/dev/zero bs=1024M count=1 iflag=direct
*注：要正确测试磁盘读写能力，建议测试文件的大小要远远大于内存的容量！！！
```

RAID

RAID, Redundant Arrays of Inexpensive Disks, 容错式廉价磁盘阵列.RAID 的基本原理是把多个便宜的小磁盘组合到一起，成为一个磁盘组，使性能达到或超过一个容量巨大、价格昂贵的磁盘。目前 RAID 技术大致分为两种：基于硬件的 RAID 技术和基于软件的 RAID 技术。其中在 Linux 下通过自带的软件就能实现 RAID 功能，这样便可省去购买昂贵的硬件 RAID 控制器和附件就能极大地增强磁盘的 IO 性能和可靠性。由于是用软件去实现的 RAID 功能，所以它配置灵活、管理方便。同时使用软件 RAID，还可以实现将几个物理磁盘合并成一个更大的虚拟设备，从而达到性能改进和数据冗余的目的。当然基于硬件的 RAID 解决方案比基于软件 RAID 技术在使用性能和服务性能上稍胜一筹，具体表现在检测和修复多位错误的能力、错误磁盘自动检测和阵列重建等方面。

RAID 动画演示

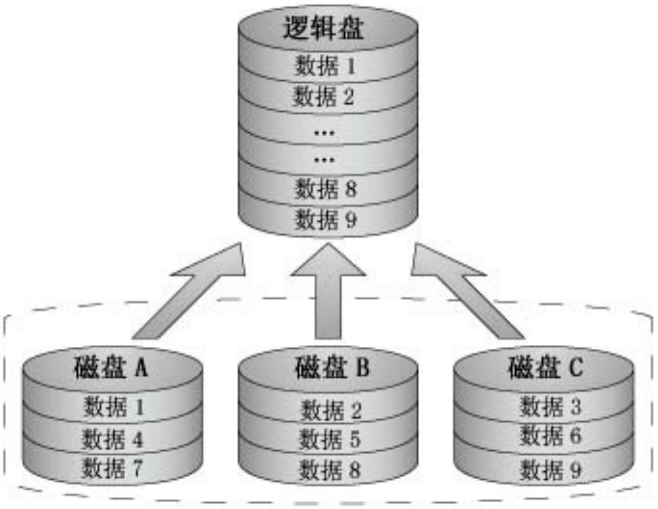
RAID 动画演示下载

RAID 分类

常用的 RAID RAID0/RAID1/RAID5/RAID10

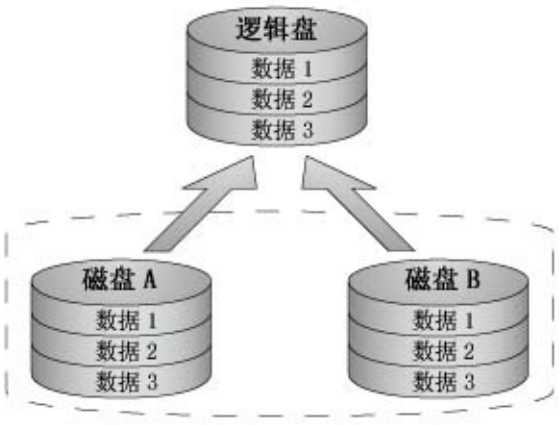
级别	优点	缺点
RAID 0	存取速度最快	没有容错
RAID 1	完全容错	成本高
RAID 5	多任务可容错	写入时有 overhead
RAID 0+1/RAID 10	速度快，完全容错，成本高	

RAID 0(striped)



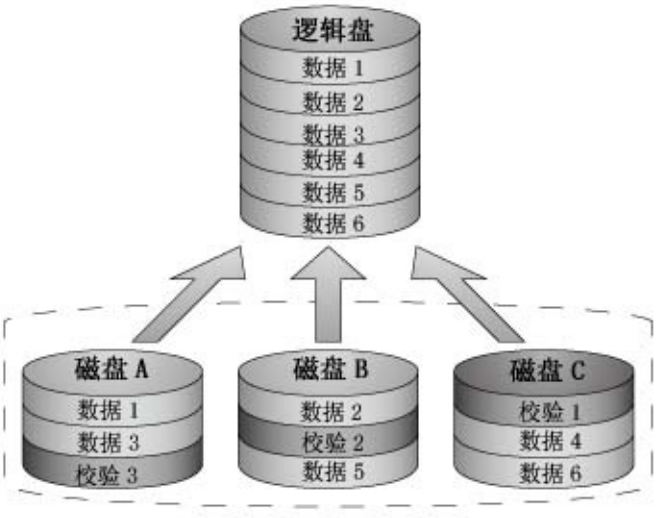
Striped 模式，把连续的数据分散懂啊多个磁盘上存取。速度快，但是没有冗余。

RAID 1(mirroring)



RAID 1 可以用于两个或 2xN 个磁盘，并使用 0 块或更多的备用磁盘，每次写数据时会同时写入镜像盘。这种阵列可靠性很高，但其有效容量减小到总容量的一半，同时这些磁盘的大小应该相等，否则总容量只具有最小磁盘的大小。

RAID 5 分布式奇偶校验的独立磁盘结构



RAID0+1/1+0

raid0 over raid1

MegaCli 管理工具

Megacli_tui

RAID 日常运维

查看 RAID 信息

```
dmesg | grep -i raid
```

更换硬盘时需要注意地方

```
(1) 更换损坏硬盘前，必须查看阵列的当前状态，保证除损坏的硬盘外，其他硬盘处于正常的 ONLINE 状态
(2) 更换硬盘必须及时
(3) 更换的新硬盘必须是完好的
(4) 在阵列数据重建完成之前，不能插拔任何硬盘
```

RAID5 单块硬盘故障恢复方法

单个硬盘失效，我们通过热插拔拔下来再插上去。如热插拔没用在进入 RAID 配置界面，将该硬盘进行 ForceOnLine 操作。还可以通过更换其它硬盘插槽，切记不要打乱磁盘顺序。如果上面操作不能解决问题，尝试将该硬盘格式化后插入，然后使用 ReBuild 操作。在这过程中可能会遇到不能格式化现象，这是因为硬盘物理错误严重，应该更换硬盘后重建数据来解决问题。

DAS/SAN/NAS

目前常见的三种存储结构

> - DAS：直连存储
> - SAN：存储区域网
> - NAS：网络附属存储

DAS

DAS : Application --> File system --> Disk Storage

DAS：直连式存储依赖服务器主机操作系统进行数据的 IO 读写和存储维护管理，数据备份和恢复要求占用服务器主机资源（包括 CPU、系统 IO 等）

SAN

SAN : Application --> File system --> Networking --> Disk Storage

IPSAN 与 FCSAN

NAS

NAS : Application --> Networking --> File system --> Disk Storage

NAS，网络附加存储，中心词"存储"，是的，它是一个存储设备。

NAS 是一个设备。CIFS/NFS 是一种协议。可以在 NAS 上启用 CIFS/NFS 协议，这样，用户就能使用 CIFS/NFS 协议进行访问了。

一句话，**CIFS** 用于 **UNIX** 和 **windows** 间共享，而 **NFS** 用于 **UNIX** 和 **UNIX** 之间共享

nfs(UNIX 和 UNIX 之间共享协议）

NFS 的基本原则是“容许不同的客户端及服务端通过一组 RPC 分享相同的文件系统”，它是独立于操作系统，容许不同硬件及操作系统的系统共同进行文件的分享。

NFS 搭建

> NFS 服务端部署环境准备

服务器系统	角色	IP
CentOS6.6 x86_64	NFS 服务端（NFS-SERVER）	192.168.1.21
CentOS6.6 x86_64	NFS 客户端（NFS-CLIENT1）	192.168.1.22

服务器版本：6.x

> NFS 软件列表

`NFS` 可以被视为一个 `RPC` 程序，在启动任何一个 `RPC` 程序之前，需要做好端口的对应映射作用，这个映射工作就是由 `rpcbind` 服务来完成的，因此在提供 `NFS` 之前必须先启动 `rpcbind` 服务

首先准备以下软件包

- nfs-utils（NFS 服务主程序，包括 rpc.nfsd、rpc.mountd 两个 deamons 和相关文档说明及执行命令文件等）
- rpcbind

> 安装 NFS 软件包

三台机器都需要安装 `NFS` 软件包，showmount 命令在 `NFS` 包中，客户端 `NFS` 服务不配置，不启动

安装 **NFS** 软件包

```
[root@nfs-server ~]$ yum install nfs-utils rpcbind -y
```

NFS 服务器配置

- `NFS` 的常用目录

目录路径	目录说明
/etc/exports	NFS 服务的主要配置文件
/usr/sbin/exportfs	NFS 服务的管理命令
/usr/sbin/showmount	客户端的查看命令
/var/lib/nfs/etab	记录 NFS 分享出来的目录的完整权限设定值
/var/lib/nfs/rtab	记录连接的客户端信息

- `NFS` 服务端的权限设置，`/etc/exports` 文件配置格式中小括号中的参数

参数名称 (*为重要参数)	参数用途
rw*	Read-write，表示可读写权限
ro	Read-only，表示只读权限
sync*	请求或写入数据时，数据同步写入到 NF SServer 中，（优点：数据安全不会丢，缺点：性能较差）
async*	请求或写入数据时，先返回请求，再将数据写入到 NFSServer 中，异步写入数据
no_root_squash	访问 NFS Server 共享目录的用户如果是 root 的话，它对共享目录具有 root 权限
not_squash	访问 NFS Server 共享目录的用户如果是 root 的话，则它的权限，将被压缩成匿名用户
all_squash*	不管访问 NFS Server 共享目录的身份如何，它的权限都被压缩成一个匿名用户，同事它的 UID、GID 都会变成 nfsnobody 账号身份
anonuid*	匿名用户 ID
anongid*	匿名组 ID
insecure	允许客户端从大于 1024 的 TCP/IP 端口连 NFS 服务器
secure	限制客户端只能从小于 1024 的 TCP/IP 端口连接 NFS 服务器（默认设置）
wdelay	检查是否有相关的写操作，如果有则将这些写操作一起执行，这样可提高效率（默认设置）
no_wdelay	若有写操作则立即执行（应与 sync 配置）
subtree_check	若输出目录是一个子目录，则 NFS 服务器将检查其父目录的权限（默认设置）
no_subtree_check	即使输出目录是一个子目录，NFS 服务器也不检查其父目录的权限，这样做可提高效率

启动 **NFS** 服务端

```
```
启动 rpcbind 状态
[root@nfs-server ~]# /etc/init.d/rpcbind start
Starting rpcbind: [OK]

查看 rpcbind 状态
[root@nfs-server ~]# /etc/init.d/rpcbind status
rpcbind (pid 1826) is running...

查看 rpcbind 默认端口 111
[root@nfs-server ~]# lsof -i :111
```
```

```
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
rpcbind 1826  rpc  6u IPv4 12657  0t0  UDP *:sunrpc
rpcbind 1826  rpc  8u IPv4 12660  0t0  TCP *:sunrpc (LISTEN)
rpcbind 1826  rpc  9u IPv6 12662  0t0  UDP *:sunrpc
rpcbind 1826  rpc 11u IPv6 12665  0t0  TCP *:sunrpc (LISTEN)

# 查看 rpcbind 服务端口
[root@nfs-server ~]# netstat -lntup|grep rpcbind
tcp  0 0 0.0.0.0:111  0.0.0.0:*        LISTEN   1826/rpcbind
tcp  0 0 :::111       :::*             LISTEN   1826/rpcbind
udp  0 0 0.0.0.0:729  0.0.0.0:*                 1826/rpcbind
udp  0 0 0.0.0.0:111  0.0.0.0:*                 1826/rpcbind
udp  0 0 :::729       :::*                      1826/rpcbind
udp  0 0 :::111       :::*                      1826/rpcbind

# 查看 rpcbind 开机是否自启动
[root@nfs-server ~]# chkconfig --list rpcbind
rpcbind    0:off   1:off   2:on   3:on  4:on   5:on    6:off

# 查看 nfs 端口信息（没有发现）
[root@nfs-server ~]# rpcinfo -p localhost
   program vers proto   port  service
    100000    4   tcp    111  portmapper
    100000    3   tcp    111  portmapper
    100000    2   tcp    111  portmapper
    100000    4   udp    111  portmapper
    100000    3   udp    111  portmapper
    100000    2   udp    111  portmapper

# 启动 NFS 服务
[root@nfs-server ~]# /etc/init.d/nfs start
Starting NFS services:                                [  OK  ]
Starting NFS quotas:                                  [  OK  ]
Starting NFS mountd:                                  [  OK  ]
Starting NFS daemon:                                  [  OK  ]
正在启动 RPC idmapd：                                [确定]

# 设置 nfs 开机自启动
[root@nfs-server ~]# chkconfig nfs on

# 查看 nfs 开机是否启动（已打开）

如何确定`rpcbind`服务一定在`NFS`服务之前启动？？？

# 无须调整，默认 rpcbind 开机顺序为 13，nfs 为 30
[root@nfs-server ~]# cat /etc/init.d/rpcbind|grep 'chkconfig'
# chkconfig: 2345 13 87（开机启动顺序 13）

[root@nfs-server ~]# cat /etc/init.d/nfs|grep 'chkconfig'
# chkconfig: - 30 60（开机启动顺序 30）
```

配置 **NFS** 服务端

NFS 配置文件为 /etc/exports

配置格式

```
```
/etc/exports 配置文件格式
NFS 共享的目录 NFS 客户端地址 (arg1，arg2...)
NFS 共享的目录 NFS 客户端地址 1(arg1，arg2...) 客户端地址 2(arg1,arg2...)
```
```

如

```
```
 /home/share 192.168.102.15(rw,sync) *(ro)
```
```

配置实例

```
```

创建共享目录
mkdir /data

NFS 配置文件添加共享目录相关信息
cat >>/etc/exports<< EOF
########nfs sync dir by zhangjie at 20150909########
/data *(rw,sync,all_squash)
EOF

NFS 平滑生效
/etc/init.d/nfs reload

查看共享记录
[root@nfs-server ~]# showmount -e localhost
Export list for localhost:
/data *

本机挂载测试
[root@nfs-server ~]# mount -t nfs 192.168.1.21:/data /mnt

查看是否已经挂载成功
[root@nfs-server ~]# df -h
Filesystem   Size Used Avail Use% Mounted on
/dev/sda3   18G 1.6G   15G 10% /
tmpfs   491M   0 491M   0% /dev/shm
/dev/sda1   190M   61M 120M 34% /boot
192.168.0.1:/data   18G 1.6G   15G 10% /mnt

配置例子
/ceshi_test *(rw,sync,no_root_squash,nohide,no_root_squash,no_subtree_check,sync)
```
```

配置 Linux NFS 客户端

```
```
启动 rpcbind 服务
[root@lamp01 ~]# /etc/init.d/rpcbind start
Starting rpcbind: [OK]

测试是否可以连接 NFS 服务器
[root@client ~]# showmount -e 192.168.1.21
Export list for 192.168.1.21:
/data *

挂载客户端 NFS 服务
[root@lamp01 ~]# mount -t nfs 192.168.1.21:/data /mnt

查看是否挂载成功
```
```

```
[root@lamp01 ~]# df -h
Filesystem          Size  Used Avail Use% Mounted on
/dev/sda3            18G  1.6G   15G  10% /
tmpfs               491M     0  491M   0% /dev/shm
/dev/sda1           190M   61M  120M  34% /boot
192.168.1.21:/data   18G  1.6G   15G  10% /mnt

# 查看 NFS 服务器完整参数配置（仔细看默认添加了很多参数，这里的 anonuid 用户、anongid 组）
[root@nfs-server /]# cat /var/lib/nfs/etab
/data   *(rw,sync,wdelay,hide,nocrossmnt,secure,root_squash,no_all_squash,no_subtree_c
heck,secure_locks,acl,anonuid=65534,anongid=65534,sec=sys,rw,root_squash,no_all_squash
)

# 查看用户组为 65534 的用户（nfsnobody 用户）
[root@nfs-server /]# grep '65534' /etc/passwd
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin

# 更改目录所属用户、所属组
[root@nfs-server /]# chown -R nfsnobody.nfsnobody /data/

# 查看目录所属用户、所属组
[root@nfs-server /]# ls -ld /data/
drwxr-xr-x 2 nfsnobody nfsnobody 4096 9 月   8 07:16 /data/

> NFS 系统安全挂载

一般`NFS`服务器共享的只是普通的静态数据（图片、附件、视频等等），不需要执行 suid、exec 等权限，挂载的这个文件系统，只能作为存取至用，无法执行程序，对于客户端来讲增加了安全性，（如：很多木马篡改站点文件都是由上传入口上传的程序到存储目录，然后执行的）注意：非性能的参数越多，速度可能越慢

# 安全挂载参数（nosuid、noexec、nodev）
mount -t nfs nosuid,noexec,nodev,rw 192.168.1.21:/data /mnt

# 禁止更新目录及文件时间戳挂载（noatime、nodiratime）
mount -t nfs noatime,nodiratime 192.168.1.21:/data /mnt

# 安全加优化的挂载方式（nosuid、noexec、nodev、noatime、nodiratime、intr、rsize、wsize）
mount -t nfs -o nosuid,noexec,nodev,noatime,nodiratime,intr,rsize=131072,wsize=131072
192.168.1.21:/data /mnt

# 默认挂载方式（无）
mount -t nfs 192.168.24.7:/data /mnt
```

配置 **Windows NFS** 客户端

```
```
启动 windos NFS 客户端服务：
1. 打开控制面板 ->程序 ->打开或关闭 windows 功能 ->NFS 客户端
勾选 NFS 客户端，即开启 windows NFS 客户端服务。

2.win+R->cmd
mount 192.168.1.21:/data X:
成功挂载，打开我的点脑，你即可在你网络位置看到 X: 盘了

X: 你挂载的网络文件盘 -- 注意，可能会与你的其他盘冲突，你可以随意更改

3. 取消挂载
直接在 我的电脑 里面鼠标点击取消映射网络驱动器 X:
或者：win+R->cmd
输入：umount X:
(umount -a 取消所有网络驱动器)
```
```

常见问题

rpcbind 安装失败

yum 安装时提示如下

```
error: %pre(rpcbind-0.2.0-12.el6.x86_64) scriptlet failed, exit status 6
Error in PREIN scriptlet in rpm package rpcbind-0.2.0-12.el6.x86_64
error:   install: %pre scriptlet failed (2), skipping rpcbind-0.2.0-12.el6
Verifying  : rpcbind-0.2.0-12.el6.x86_64                      1/1
Failed:
    rpcbind.x86_64 0:0.2.0-12.el6
```

因为通过 chattr +i 把 /etc/passwd /etc/group /etc/shadow /etc/gshadow 锁定了。

chattr -i 解锁后，问题解决。

nfs 客户端挂载失败

现象：客户端 mount 失败，同时 rpcbind 服务是停止的，怎么都启不来 mount 时提示如下：

```
mount.nfs: rpc.statd is not running but is required for remote locking.
mount.nfs: Either use '-o nolock' to keep locks local, or start statd.
mount.nfs: an incorrect mount option was specified
```

调试后发现，rpcbind 要求在 /etc/sysconfig/network 文件里写一个 NETWORKING=yes 才行，加上配置后，即可启动 rpcbind 服务

nfs 客户端无法 chown

nfs 常规配置后，客户端可以创建，删除，chmod；但无法修改属主和属组；

解决方法：

挂载时，加上 vers=3 即可，例：

```
#mount -t nfs -o vers=3 server:/share /mnt
```

CIFS(UNIX 和 windows 间共享协议）

在 Linux 上连接 windows 上 NAS 设备时，需要 cifs-utils 支持

```
#yum -y install cifs-utils
```

给挂载共享文件夹指定 owner 和 group

在服务器部署的时候需要把文件夹设置在 windows 的共享文件上。在使用 mount 命令挂载到 linux 上后。文件路径和文件都是可以访问，但是不能写入，导致系统在上传文件的时候提示“权限不够，没有写权限”。用"ls-l"查看挂载文件的权限设置是 drwxr-xr-x, 很明显没有写权限。想当然使用 chmod 来更改文件夹权限，结果提示权限不够。root 和当前用户都不能正常修改权限。

可以添加两个参数即可达到我们所要的效果：

```
#mount -t cifs -o username="***",password="***",gid=***,uid=**** //WindowsHost/sharefo
lder  /home/xxx/shared
```

gid 和 uid，可以使用 id username 来获得

给 mount 共享文件夹所在组的写权限

```
#mount -t cifs -o username="Administrator",password="PasswordForWindows",uid=test_user
,gid=test_user,dir_mode=0777 //192.168.1.2/test /mnt/
```

永久挂载 Windows 共享

```
#mount -t cifs -o username="***",password="***",gid=500,uid=500 //WindowsHost/sharefol
der  /home/xxx/shared
```

如上挂载时，可写入 fstab 文件

```
//WindowsHost/sharefolder /home/xxx/shared cifs username=***,password=***,uid=500,gid=
500 0 0
```

遗憾的是，此命令具有明显的安全问题，因为您必须在 /etc/fstab 条目中公开密码，而文件 /etc/fstab 通常可供系统上的每个用户读取。要解决此问题，可使用 credentials 挂载选项将用户名和密码放在指定的文本文件中。例如：

```
//WindowsHost/sharefolder /home/xxx/shared cifs credentials=/etc/cred.ceshi,ui500,gid=
500 0 0
```

一个 credentials 文件的格式如下所示：

```
username=***
password=MYPASSWORD
```

然后可使用以下命令，使 /etc/cred.ceshi 文件仅可供 root 用户（必须以其身份执行 mount 命令的用户）读取：

```
#chmod 600 /etc/cred.ceshi
```

GFS

```
* [分布式文件系统的要求](#分布式文件系统的要求)
* [GFS 基于的假设](#gfs-基于的假设)
```


GFS 作为一个分布式的文件系统，除了要满足一般的文件系统的需求之外，还根据一些特殊的应用场景（原文反复提到的 `application workloads and technological environment`），来完成整个系统的设计。

分布式文件系统的要求

一般的分布式文件系统需要满足以下四个要求：

- Performance：高性能，较低的响应时间，较高的吞吐量
- Scalability: 易于扩展，可以简单地通过增加机器来增大容量
- Reliability: 可靠性，系统尽量不出错误
- Availability: 可用性，系统尽量保持可用

（注：关于 reliability 和 availability 的区别，请参考 这篇）

GFS 基于的假设

基于对实际应用场景的研究，GFS 对它的使用场景做出了如下假设：

1. GFS 运行在成千上万台便宜的机器上，这意味着节点的故障会经常发生。 必须有一定的容错的机制来应对这些故障。
2. 系统要存储的文件通常都比较大，每个文件大约 100MB 或者更大， GB 级别的文件也很常见。必须能够有效地处理这样的大文件， 基于这样的大文件进行系统优化。
3. workloads 的读操作主要有两种：
    - 大规模的流式读取，通常一次读取数百 KB 的数据， 更常见的是一次读取 1MB 甚至更多的数据。 来自同一个 client 的连续操作通常是读取同一个文件中连续的一个区域。
    - 小规模的随机读取，通常是在文件某个随机的位置读取 几个 KB 数据。 对于性能敏感的应用通常把一批随机读任务进行排序然后按照顺序批量读取， 这样能够避免在通过一个文件来回移动位置。（后面我们将看到， 这样能够减少获取 metadata 的次数，也就减少了和 master 的交互）
1. workloads 的写操作主要由大规模的，顺序的 append 操作构成。 一个文件一旦写好之后，就很少进行改动。因此随机的写操作是很少的， 所以 GFS 主要针对于 append 进行优化。
2. 系统必须有合理的机制来处理多个 client 并发写同一个文件的情况。 文件经常被用于生产者 - 消费者队列，需要高效地处理多个 client 的竞争。 正是基于这种特殊的应用场景，GFS 实现了一个无锁并发 append。
3. 利用高带宽比低延迟更加重要。基于这个假设， 可以把读写的任务分布到各个节点， 尽量保证每个节点的负载均衡， 尽管这样会造成一些请求的延迟。

架构

下面我们来具体看一下 GFS 的整个架构。

可以看到 GFS 由三个不同的部分组成，分别是 `master` ， `client` , `chunkserver` 。

> - `master` 负责管理整个系统（包括管理 metadata，垃圾回收等），一个系统只有一个 `master` 。
> - `chunkserver` 负责保存数据，一个系统有多个 `chunkserver` 。
> - `client` 负责接受应用程序的请求，通过请求 `master` 和 `chunkserver` 来完成读写等操作。

由于系统只有一个 `master` ， `client` 对 `master` 请求只涉及 metadata，数据的交互直接与 `chunkserver` 进行，这样减小了 `master` 的压力。

一个文件由多个 chunk 组成，一个 chunk 会在多个 `chunkserver` 上存在多个 replica。对于新建文件，目录等操作，只是更改了 metadata，只需要和 `master` 交互就可以了。注意，与 linux 的文件系统不同，目录不再以一个 inode 的形式保存，也就是它不会作为 data 被保存在 `chunkserver`。如果要读写文件的文件的内容，就需要 `chunkserver` 的参与，`client` 根据需要操作文件的偏移量转化为相应的 `chunk index`，向 `master` 发出请求，`master` 根据文件名和 `chunk index`，得到一个全局的 `chunk handle`，一个 chunk 由唯一的一个 `chunk handle` 所标识，`master` 返回这个 `chunk handle` 以及拥有这个 chunk 的 `chunkserver` 的位置。（不止一个，一个 chunk 有多个 replica，分布在不同的 `chunkserver`。必要的时候，`master` 可能会新建 chunk，并在 `chunkserver` 准备好了这个 chunk 的 replica 之后，才返回）`client` 拿到 `chunk handle` 和 `chunkserver` 列表之后，先把这个信息用文件名和 `chunk index` 作为 key 缓存起来，然后对相应的 `chunkserver` 发出数据的读写请求。这只是一个大概的流程，对于具体的操作过程，下面会做分析。

Chunk 大小

Chunk 的大小是一个值得考虑的问题。在 GFS 中，chunk 的大小是 64MB。这比普通文件系统的 block 大小要大很多。在 `chunkserver` 上，一个 chunk 的 replica 保存成一个文件，这样，它只占用它所需要的空间，防止空间的浪费。

Chunk 拥有较大的大小由如下几个好处：

- 它减少了 `client` 和 `master` 交互的次数。
- 减少了网络的开销，由于一个客户端可能对同一个 chunk 进行操作，这样可以与 `chunkserver` 维护一个长 TCP 连接。
- chunk 数目少了，metadata 的大小也就小了，这样节省了 `master` 的内存。

大的 chunk size 也会带来一个问题，一个小文件可能就只占用一个 chunk，那么如果多个 `client` 同时操作这个文件的话，就会变成操作同一个 chunk，保存这个 chunk 的 `chunkserver` 就会称为一个 hotspot。这样的问题对于小的 chunk 并不存在，因为如果是小的 chunk 的话，一个文件拥有多个 chunk，操作同一个文件被分布到多个 `chunkserver`. 虽然在实践中，可以通过错开应用的启动的时间来减小同时操作一个文件的可能性。

Metadata

GFS 的 `master` 保存三种 metadata：

1. 文件和 chunk 的 namespace（命名空间）-- 整个文件系统的目录结构以及 chunk 基本信息
2. 文件到 chunk 的映射
3. 每一个 chunk 的具体位置

metadata 保存在内存中，可以很快地获取。 前面两种 metadata 会通过 operation log 来持久化。 第 3 种信息不用持久化，因为在 `master` 启动时， 它会问 `chunkserver` 要 chunk 的位置信息。 而且 chunk 的位置也会不断的变化，比如新的 `chunkserver` 加入。 这些新的位置信息会通过日常的 `HeartBeat` 消息由 `chunkserver` 传给 `master` 。

将 metadata 保存在内存中能够保证在 `master` 的日常处理中很快的获取 metadata， 为了保证系统的正常运行， `master` 必须定时地做一些维护工作，比如清除被删除的 chunk， 转移或备份 chunk 等，这些操作都需要获取 metadata。 metadata 保存在内存中有一个不好的地方就是能保存的 metadata 受限于 `master` 的内存， 不过足够大的 chunk size 和使用前缀压缩，能够保证 metadata 占用很少的空间。

对 metadata 进行修改时，使用锁来控制并发。需要注意的是，对于目录， 获取锁的方式和 linux 的文件系统有点不太一样。在目录下新建文件， 只获取对这个目录的读锁，而对目录进行 snapshot，却对这个目录获取一个写锁。 同时，如果涉及到某个文件，那么要获取所有它的所有上层目录的读锁。 这样的锁有一个好的地方是可以在通过一个目录下同时新建两个文件而不会冲突， 因为它们都是获得对这个目录的读锁。

Operation Log

Operation log 用于持久化存储前两种 metadata，这样 `master` 启动时， 能够根据 operation log 恢复 metadata。同时，可以通过 operation log 知道 metadata 修改的顺序， 对于重现并发操作非常有帮助。因此，必须可靠地存储 operation log， 只有当 operation log 已经存储好之后才向 `client` 返回。 而且，operation log 不仅仅只保存在 `master` 的本地，而且在远程的机器上有备份， 这样，即使 `master` 出现故障，也可以使用其他的机器做为 `master` 。

从 operation log 恢复状态是一个比较耗时的过程，因此，使用 checkpoint 来减小 operation log 的大小。 每次恢复时，从 checkpoint 开始恢复，只处理 checkpoint 只有的 operation log。 在做 checkpoint 时，新开一个线程进行 checkpoint，原来的线程继续处理 metadata 的修改请求， 此时把 operation log 保存在另外一个文件里。

容错机制

Master 容错

通过操作日志加 checkpoint 的方式进行，并且有一台称为 "Shadow Master" 的实时热备。

> - GFS Master 的修改操作总是先记录操作日志，然后修改内存。
> - Master 会定期将内存中的数据以 checkpoint 文件的形式转存到磁盘中
> - 实时热备，所有元数据修改操作都发送到实时热备才算成功。

一致性模型

关于一致性，先看几个定义，对于一个 file region，存在以下几个状态：

- consistent。如果任何 replica, 包含的都是同样的 data。
- defined。defined 一定是 consistent，而且能够看到一次修改造成的结果。
- undefined。undefined 一定是 consistent，是多个修改混合在一块。举个例子， 修改 a 想给文件添加 A1,A2，修改 b 想给文件添加 B1,B2，如果最后的结果是 A1,A2,B1,B2，那么就是 defined，如果是 A1,B1,A2,B2，就是 undefined。
- inconsitent。对于不同的 replica，包含的是不同的 data。

在 GFS 中，不同的修改可能会出现不同的状态。对于文件的 append 操作（是 GFS 中的主要写操作）， 通过放松一定的一致性，更好地支持并发，在下面的具体操作时再讲述具体的过程。

Lease 机制

`master` 通过 lease 机制把控制权交给 `chunkserver` ，当写一个 chunk 时， `master` 指定一个包含这个 chunk 的 replica 的 `chunkserver` 作为 `primary replica` ， 由它来控制对这个 chunk 的写操作。一个 lease 的过期时间是 60 秒，如果写操作没有完成， `primary replica` 可以延长这个 lease。 `primary replica` 通过一个序列号控制对这个 chunk 的写的顺序， 这样能够保证所有的 replica 都是按同样的顺序执行同样的操作，也就保证了一致性。

版本号

对于每一个 chunk 的修改，chunk 都会赋予一个新的版本号。 这样，如果有的 replica 没有被正常的修改（比如修改的时候当前的 `chunkserver` 挂了）, 那么这个 replica 就被 `stale replica` ，当 `client` 请求一个 chuck 时， `stale replica` 会被 `master` 忽略， 在 `master` 的定时管理过程中，会把 `stale replica` 删除。

负载均衡

为了尽量保证所有 `chunkserver` 都承受差不多的负载， `master` 通过以下机制来完成：

- 首先，在新建一个 chunk 或者是复制一个 chunk 的 replica 时， 尽量保证负载均衡。
- 当一个 chunk 的 replica 数量低于某个值时，尝试给这个 chuck 复制 replica
- 扫描整个系统的分布情况，如果不够平衡，则通过移动一些 replica 来达到负责均衡的目的。

注意， `master` 不仅考虑了 `chunkserver` 的负载均衡，也考虑了机架的负载均衡。

基本操作

Read

Read 操作其实已经在上面的 Figure 1 中描述得很明白了，有如下几个过程：

1. `client` 根据 chunk size 的大小，把 `(filename,byte offset)` 转化为 `(filename,chunk index)`，发送 `(filename,chunk index)` 给 `master`
2. `master` 返回 `(chunk handle, 所有正常 replica 的位置)`，`client` 以 `(filename,chunk index)` 作为 key 缓存这个信息
3. `client` 发 `(chunk handle,byte range)` 给其中一个 `chunkserver`，通常是最近的一个。
4. `chunkserver` 返回 chunk data

Overwrite

直接假设 `client` 已经知道了要写的 chunk，如 Figure 2，具体过程如下：

1. `client` 向 `master` 询问拥有这个 chunk 的 lease 的 `primary replica`，如果当前没有 `primary replica`， `master` 把 lease 给其中的 replica
2. `master` 把 `primary replica` 的位置和其他的拥有这个 chunk 的 replica 的 `chunkserver`（`secondary replica`）的位置返回，`client` 缓存这个信息。
3. `client` 把数据以流水线的方式发送到所有的 replica，流水线是一种最高效利用的带宽的方法，每一个 replica 把数据用 LRU buffer 保存起来，并向 `client` 发送接受到的信息。
4. `client` 向 `primary replica` 发送 write 请求，`primary replica` 根据请求的顺序赋予一个序列号
5. `primary replica` 根据序列号修改 replica 和请求其他的 `secondary replica` 修改 replica，这个统一的序列号保证了所有的 replica 都是按照统一的顺序来执行修改操作。
6. 当所有的 `secondary replica` 修改完成之后，返回修改完成的信号给 `primary replica`
7. `primary replica` 向 `client` 返回修改完成的信号，如果有任何的 `secondary replica` 修改失败，信息也会被发给 `client`，`client` 然后重新尝试修改，重新执行步骤 3-7。

如果一个修改很大或者到了 chuck 的边界，那么 client 会把它分成两个写操作，这样就有可能发生在两个写操作之间有其他的写操作，所以这时会出现 undefined 的情况。

Record Append

Record Append 的过程相对于 Overwrite 的不同在于它的错误处理不同，当写操作没有成功时，`client` 会尝试再次操作，由于它不知道 offset，所以只能再次 append，这就会导致在一些 replica 有重复的记录，而且不同的 replica 拥有不同的数据。

为了应对这种情况的发生，应用程序必须通过一定的校验手段来确保数据的正确性，如果对于生产者 - 消费者队列，消费者可以通过唯一的 id 过滤掉重复的记录。

Snapshot

Snapshot 是对文件或者一个目录的“快照”操作，快速地复制一个文件或者目录。 GFS 使用*Copy-on-Write*实现 snapshot，首先 `master` revoke 所有相关 chunk 的 lease， 这样所有的修改文件的操作都需要和 `master` 联系， 然后复制相关的 metadata，复制的文件跟原来的文件指向同样的 chunck， 但是 chuck 的 reference count 大于 1。

当有 `client` 需要写某个相关的 chunck C 时， `master` 会发现它的 reference count 大于 1， `master` 推迟回复给 `client` ，先新建一个 `chunk handle` C'， 然后让所有拥有 C 的 replica 的 `chunkserver` 在本地新建一个同样的 C‘的 replica， 然后赋予 C’的一个 replica 一个 lease，把 C'返回给 `client` 用于修改。

Delete

当 `client` 请求删除文件时，GFS 并不立即回收这个文件的空间。 也就是说，文件相关的 metadata 还在， 文件相关的 chunk 也没有从 `chunkserver` 上删除。 GFS 只是简单的把文件删除的 operation log 记下， 然后把文件重新命名为一个 hidden name， 里面包含了它的删除时间。 在 `master` 的日常维护工作时， 它会把删除时间删除时间超过 3 天的文件从 metadata 中删除， 同时删除相应 chunk 的 metadata， 这样这些 chunk 就变成了 orphan chunk， 它们会在 `chunkserver` 和 `master` 进行 `Heartbeat` 交互时从 `chunkserver` 删除。

这样推迟删除（原文叫垃圾回收）的好处有：

- 对于分布式系统而言，要确保一个动作正确执行是很难的， 所以如果当场要删除一个 chunk 的所有 replica 需要复杂的验错，重试。 如果采用这种推迟删除的方法，只要 metadata 被正确的处理，最后的 replica 就一定会被删除， 非常简单
- 把这些删除操作放在 `master` 的日常处理中，可以使用批处理这些操作， 平摊下来的开销就小了
- 可以防止意外删除的可能，类似于回收站

这样推迟删除的不好在于浪费空间，如果空间吃紧的话， `client` 可以强制删除， 或者指定某些目录下面的文件直接删除。

GlusterFS

说明

简介

```
GlusterFS 是 Scale-Out 存储解决方案 Gluster 的核心，它是一个开源的分布式文件系统， 具有强大的横向扩展能力，通过扩展能够支持数 PB 存储容量和处理数千客户端。

GlusterFS 借助 TCP/IP 或 InfiniBand RDMA 网络将物理分布的存储资源聚集在一起，使用单一全局命名空间来管理数据。

GlusterFS 基于可堆叠的用户空间设计，可为各种不同的数据负载提供优异的性能。

GlusterFS 支持运行在任何 IP 网络上的标准应用程序的标准客户端，用户可以在全局统一的命名空间中使用 NFS/CIFS 等标准协议来访问应用数据。

GlusterFS 使得用户可摆脱原有的独立、高成本的封闭存储系统，能够利用普通廉价的存储设备来部署可集中管理、横向扩展、虚拟化的存储池，存储容量可扩展至 TB/PB 级。

目前 GlusterFS 已被 Red Hat 收购，它的官网是：https://www.gluster.org/
```

GlusterFS 在企业中的应用场景

理论和实践上分析，GlusterFS 目前主要适用于大文件存储场景，对于小文件尤其是海量小文件，存储效率和访问性能都表现不佳。建议存放文件大小大于 1MB

GlusterFS 安装

环境说明

> - CentOS 7
> - GlusterFS

3 台机器安装 GlusterFS 组成一个集群。

```
服务器：
10.1.0.11
10.1.0.12
10.1.0.13

配置 hosts

10.1.0.11 manager
10.1.0.12 node-1
10.1.0.13 node-2

client:
10.1.0.10 client
```

安装 GlusterFS

CentOS 安装 GlusterFS 非常的简单

在三个节点都安装 GlusterFS

```
# 安装 GlusterFS yum 源文件
#yum install centos-release-gluster

# 安装 GlusterFS 软件
yum install -y glusterfs glusterfs-server glusterfs-fuse glusterfs-rdma
```

启动 GlusterFS

```
[root@manager ~]#systemctl start glusterd.service
[root@manager ~]#systemctl enable glusterd.service
[root@manager ~]#systemctl status glusterd.service
● glusterd.service - GlusterFS, a clustered file-system server
  Loaded: loaded (/usr/lib/systemd/system/glusterd.service; disabled; vendor preset: d
isabled)
  Active: active (running) since 一 2017-02-27 17:55:56 CST; 11s ago
 Process: 5476 ExecStart=/usr/sbin/glusterd -p /var/run/glusterd.pid --log-level $LOG_
LEVEL $GLUSTERD_OPTIONS (code=exited, status=0/SUCCESS)
Main PID: 5477 (glusterd)
 Memory: 15.0M
 CGroup: /system.slice/glusterd.service
         └─5477 /usr/sbin/glusterd -p /var/run/glusterd.pid --log-level INFO

2 月 27 17:55:56 meetbill systemd[1]: Starting GlusterFS, a clustered file-system serve
r...
2 月 27 17:55:56 meetbill systemd[1]: Started GlusterFS, a clustered file-system server
.
```

配置 GlusterFS

在 manager 节点上配置，将 节点 加入到 集群中。

```
[root@manager ~]#gluster peer probe manager
peer probe: success. Probe on localhost not needed

[root@manager ~]#gluster peer probe node-1
peer probe: success.

[root@manager ~]#gluster peer probe node-2
peer probe: success.
```

查看集群状态：

```
[root@manager ~]#gluster peer status
Number of Peers: 2

Hostname: node-1
Uuid: 41573e8b-eb00-4802-84f0-f923a2c7be79
State: Peer in Cluster (Connected)

Hostname: node-2
Uuid: da068e0b-eada-4a50-94ff-623f630986d7
State: Peer in Cluster (Connected)
```

创建数据存储目录：

```
[root@manager ~]#mkdir -p /opt/gluster/data
[root@node-1 ~]# mkdir -p /opt/gluster/data
[root@node-2 ~]# mkdir -p /opt/gluster/data
```

查看 volume 状态：

```
[root@manager ~]#gluster volume info
No volumes present
```

volume 模式说明

一、 默认模式，既 DHT, 也叫 分布卷：将文件已 hash 算法随机分布到 一台服务器节点中存储。 gluster volume create test-volume server1:/exp1 server2:/exp2

二、 复制模式，既 AFR, 创建 volume 时带 replica x 数量：将文件复制到 replica x 个节点中。 gluster volume create test-volume replica 2 transport tcp server1:/exp1 server2:/exp2

三、 条带模式，既 Striped, 创建 volume 时带 stripe x 数量： 将文件切割成数据块，分别存储到 stripe x 个节点中 (类似 raid 0)。 gluster volume create test-volume stripe 2 transport tcp server1:/exp1 server2:/exp2

四、 分布式条带模式（组合型），最少需要 4 台服务器才能创建。 创建 volume 时 stripe 2 server = 4 个节点： 是 DHT 与 Striped 的组合型。 gluster volume create test-volume stripe 2 transport tcp server1:/exp1 server2:/exp2 server3:/exp3 server4:/exp4

五、 分布式复制模式（组合型）, 最少需要 4 台服务器才能创建。 创建 volume 时 replica 2 server = 4 个节点：是 DHT 与 AFR 的组合型。 gluster volume create test-volume replica 2 transport tcp server1:/exp1 server2:/exp2 server3:/exp3 server4:/exp4

六、 条带复制卷模式（组合型）, 最少需要 4 台服务器才能创建。 创建 volume 时 stripe 2 replica 2 server = 4 个节点： 是 Striped 与 AFR 的组合型。 gluster volume create test-volume stripe 2 replica 2 transport tcp server1:/exp1 server2:/exp2 server3:/exp3 server4:/exp4

七、 三种模式混合，至少需要 8 台 服务器才能创建。 stripe 2 replica 2 , 每 4 个节点 组成一个 组。 gluster volume create test-volume stripe 2 replica 2 transport tcp server1:/exp1 server2:/exp2 server3:/exp3 server4:/exp4 server5:/exp5 server6:/exp6 server7:/exp7 server8:/exp8

创建 GlusterFS 磁盘

```
[root@manager ~]#gluster volume create models replica 3 manager:/opt/gluster/data node
-1:/opt/gluster/data node-2:/opt/gluster/data force
volume create: models: success: please start the volume to access data
```

再查看 volume 状态：

```
[root@manager ~]#gluster volume info

Volume Name: models
Type: Replicate
Volume ID: e539ff3b-2278-4f3f-a594-1f101eabbf1e
Status: Created
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: manager:/opt/gluster/data
Brick2: node-1:/opt/gluster/data
Brick3: node-2:/opt/gluster/data
Options Reconfigured:
performance.readdir-ahead: on
```

启动 models

```
[root@manager ~]#gluster volume start models
volume start: models: success
```

GlusterFS 客户端

部署 GlusterFS 客户端并 mount GlusterFS 文件系统

```
[root@client ~]#yum install -y glusterfs glusterfs-fuse
[root@client ~]#mkdir -p /opt/gfsmnt
[root@client ~]#mount -t glusterfs manager:models /opt/gfsmnt/
```

```
[root@node-94 ~]#df -h
文件系统 容量 已用 可用 已用 % 挂载点
/dev/mapper/vg001-root 98G 1.2G 97G 2% /
devtmpfs 32G 0 32G 0% /dev
tmpfs 32G 0 32G 0% /dev/shm
tmpfs 32G 130M 32G 1% /run
tmpfs 32G 0 32G 0% /sys/fs/cgroup
/dev/mapper/vg001-opt 441G 71G 370G 17% /opt
/dev/sda2 497M 153M 344M 31% /boot
tmpfs 6.3G 0 6.3G 0% /run/user/0
manager:models 441G 18G 424G 4% /opt/gfsmnt
```

维护

1. 查看 GlusterFS 中所有的 volume:

```
[root@manager ~]#gluster volume list
```

1. 删除 GlusterFS 磁盘：

```
[root@manager ~]#gluster volume stop models   #停止名字为 models 的磁盘
[root@manager ~]#gluster volume delete models #删除名字为 models 的磁盘
```

注： 删除 磁盘 以后，必须删除 磁盘 (/opt/gluster/data) 中的 （ .glusterfs/ .trashcan/ ）目录。 否则创建新 volume 相同的 磁盘 会出现文件 不分布，或者 类型 错乱 的问题。

1. 卸载某个节点 GlusterFS 磁盘

```
[root@manager ~]#gluster peer detach node-2
```

1. 设置访问限制，按照每个 volume 来限制

```
[root@manager ~]#gluster volume set models auth.allow 10.6.0.*,10.7.0.*
```

1. 添加 GlusterFS 节点：

```
[root@manager ~]#gluster peer probe node-3
[root@manager ~]#gluster volume add-brick models node-3:/opt/gluster/data
```

注：如果是复制卷或者条带卷，则每次添加的 Brick 数必须是 replica 或者 stripe 的整数倍

1. 配置卷

```
[root@manager ~]# gluster volume set
```

1. 缩容 volume:

先将数据迁移到其它可用的 Brick，迁移结束后才将该 Brick 移除：

```
[root@manager ~]#gluster volume remove-brick models node-2:/opt/gluster/data node-3:/o
pt/gluster/data start
```

在执行了 start 之后，可以使用 status 命令查看移除进度：

```
[root@manager ~]#gluster volume remove-brick models node-2:/opt/gluster/data node-3:/o
pt/gluster/data status
```

不进行数据迁移，直接删除该 Brick：

```
[root@manager ~]#gluster volume remove-brick models node-2:/opt/gluster/data node-3:/o
pt/gluster/data commit
```

注意，如果是复制卷或者条带卷，则每次移除的 Brick 数必须是 replica 或者 stripe 的整数倍。

扩容：

```
gluster volume add-brick models node-2:/opt/gluster/data
```

1. 修复命令：

```
[root@manager ~]#gluster volume replace-brick models node-2:/opt/gluster/data node-3:/
opt/gluster/data commit -force
```

1. 迁移 volume:

```
[root@manager ~]#gluster volume replace-brick models node-2:/opt/gluster/data node-3:/
opt/gluster/data start
```

pause 为暂停迁移

```
[root@manager ~]#gluster volume replace-brick models node-2:/opt/gluster/data node-3:/
opt/gluster/data pause
```

abort 为终止迁移

```
[root@manager ~]#gluster volume replace-brick models node-2:/opt/gluster/data node-3:/
opt/gluster/data abort
```

status 查看迁移状态

```
[root@manager ~]#gluster volume replace-brick models node-2:/opt/gluster/data node-3:/
opt/gluster/data status
```

迁移结束后使用 commit 来生效

```
[root@manager ~]#gluster volume replace-brick models node-2:/opt/gluster/data node-3:/
opt/gluster/data commit
```

1. 均衡 volume:

```
[root@manager ~]#gluster volume models lay-outstart
[root@manager ~]#gluster volume models start
[root@manager ~]#gluster volume models startforce
[root@manager ~]#gluster volume models status
[root@manager ~]#gluster volume models stop
```

Ceph

Ceph

Ceph 提供了对象、块和文件存储功能。

Ceph 对象存储

- REST 风格的接口
- 与 S3 和 Swift 兼容的 API
- S3 风格的子域
- 统一的 S3/Swift 命名空间
- 用户管理
- 利用率跟踪
- 条带化对象
- 云解决方案集成
- 多站点部署
- 灾难恢复

Ceph 块设备

- 瘦接口支持
- 映像尺寸最大 16EB
- 条带化可定制
- 内存缓存
- 快照
- 写时复制克隆
- 支持内核级驱动
- 支持 KVM 和 libvirt
- 可作为云解决方案的后端
- 增量备份

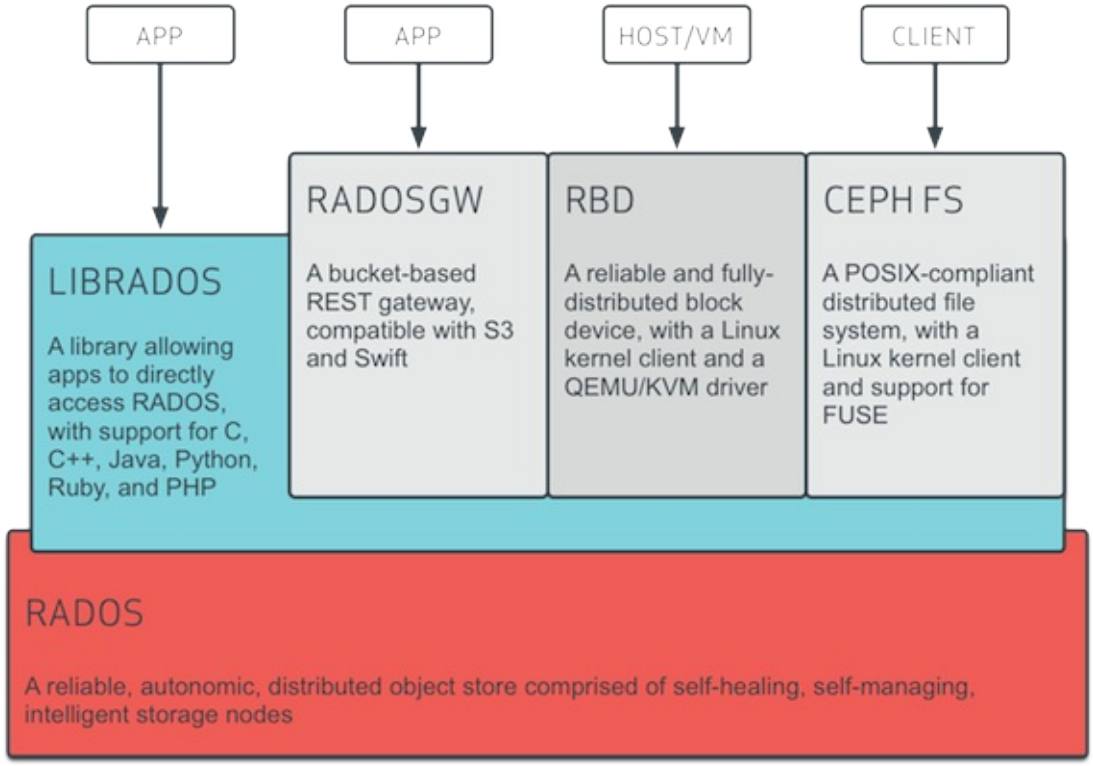
Ceph 文件系统

- 与 POSIX 兼容的语义
- 元数据独立于数据
- 动态重均衡
- 子目录快照
- 可配置的条带化
- 有内核驱动支持
- 有用户空间驱动支持
- 可作为 NFS/CIFS 部署
- 可用于 Hadoop（取代 HDFS)

Ceph 架构

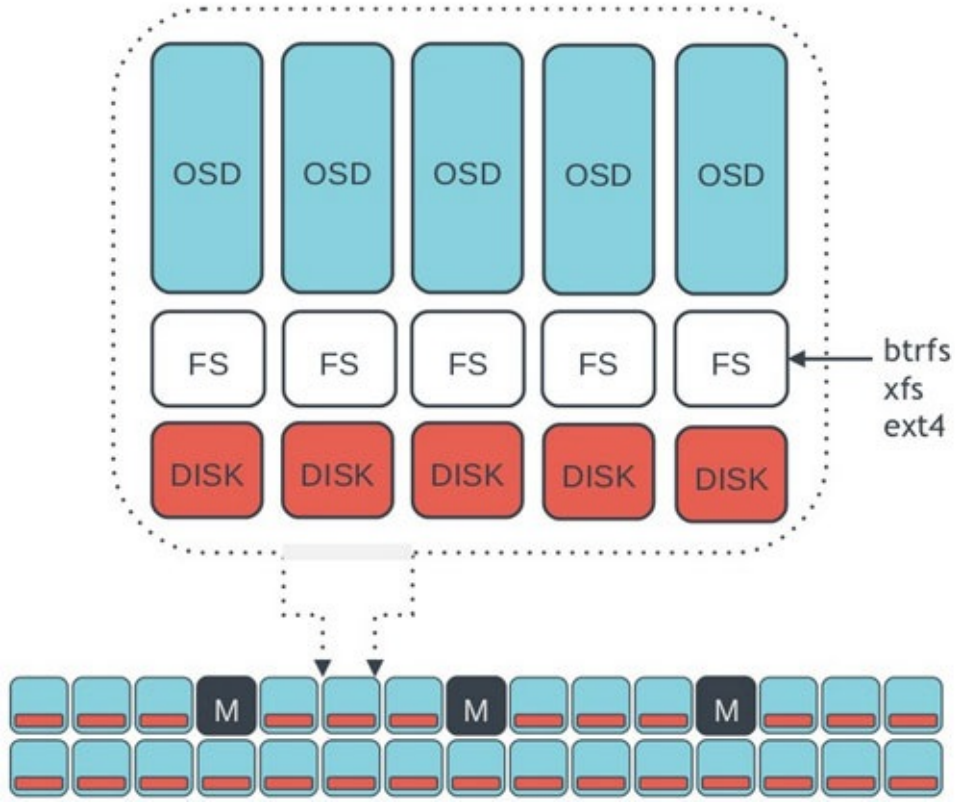
Rados 核心组件，提供高可靠、高可扩展、高性能的分布式对象存储架构，利用本地文件系统存储对象。

Client RBD Radosgw Librados Cephfs



Ceph 组件

最简的 Ceph 存储集群至少要一个监视器和两个 OSD 守护进程，只有运营 Ceph 文件系统时元数据服务器才是必需的。**OSD**（对象存储守护进程） 存储数据，处理数据复制、恢复、回填、重均衡，并向监视器提供邻居的心跳信息。



Monitor 维护着各种集群状态图，包括监视器图、OSD 图、归置组 (PG) 图和 CRUSH 图。
MDS（元数据服务器） 存储元数据。缓存和同步元数据，管理名字空间。

硬件配置

CPU

元数据服务器对 CPU 敏感，CPU 性能尽可能高。 OSD 需要一定的处理能力，CPU 性能要求较高。

内存

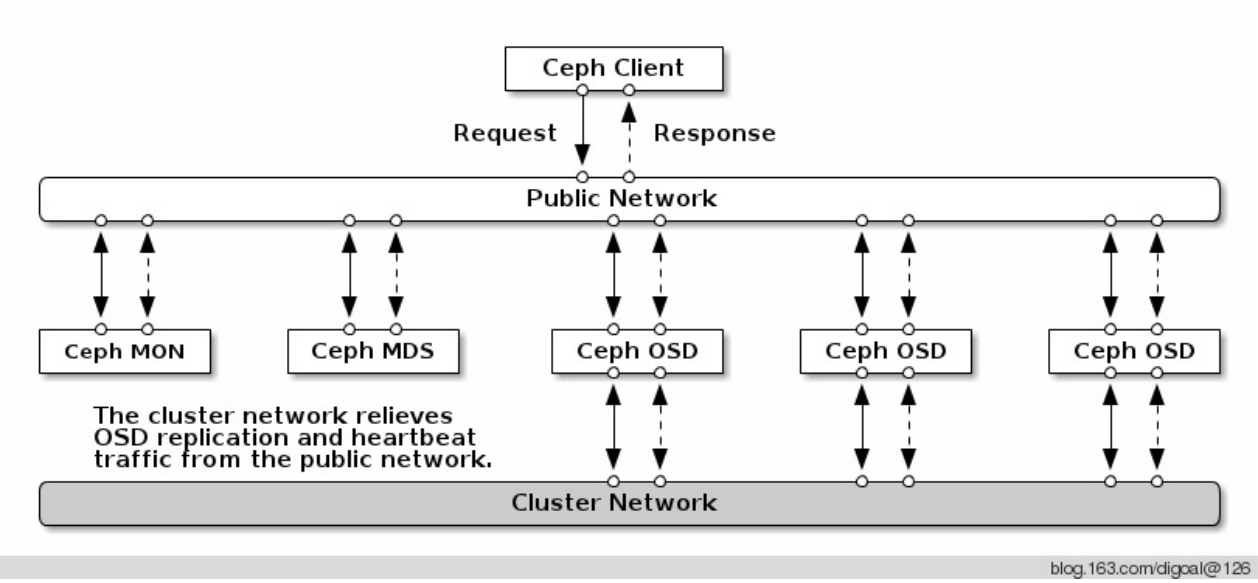
元数据服务器和监视器对内存要求较高。 OSD 对内存要求较低。 恢复期间，占用内存较大。

数据存储

建议用容量大于 1TB 的硬盘。 每个 OSD 守护进程占用一个驱动器。 分别在单独的硬盘运行操作系统、OSD 数据和 OSD 日志。 SSD 顺序读写性能很重要。 SSD 可用于存储 OSD 的日志。

Ceph 网络

建议每台服务器至少两个千兆网卡，分别用于公网（前端）和集群网络（后端）。集群网络用于处理有数据复制产生的额外负载，而且可用防止拒绝服务攻击。考虑部署万兆网络。



最低硬件推荐（小型生产集群及开发集群）

Ceph-osd

CPU:

- 1x 64-bit AMD-64
- 1x 32-bit ARM dual-core or better
- 1x i386 dual-core

RAM: ~1GB for 1TB of storage per daemon **Volume Storage:** 1x storage drive per daemon **Journal:** 1x SSD partition per daemon (optional) **Network:** 2x 1GB Ethernet NICs

Ceph-mon

CPU:

- 1x 64-bit AMD-64/i386
- 1x 32-bit ARM dual-core or better
- 1x i386 dual-core

RAM: 1GB per daemon **Disk Space:** 10GB per daemon **Network:** 2x 1GB Ethernet NICs

Ceph-mds

CPU:

> - 1x 64-bit AMD-64 quad-core
> - 1x 32-bit ARM quad-core
> - 1x i386 quad-core

RAM: 1GB minimum per daemon **Disk Space:** 1MB per daemon **Network:** 2x 1GB Ethernet NICs

生产集群

Dell PowerEdge R510

CPU： 2x 64-bit quad-core Xeon CPUs **RAM:** 16GB **Volume Storage:** 8x 2TB drives.1 OS,7 Storage **Client Network:** 2x 1GB Ethernet NICs **OSD Network:** 2x 1GB Ethernet NICs **Mgmt. Network:** 2x 1GB Ethernet NICs

Dell PowerEdge R515

CPU： 1x hex-core Opteron CPU **RAM:** 16GB **Volume Storage:** 12x 3TB drives.Storage **OS Storage:** 1x 500GB drive. Operating System. **Client Network:** 2x 1GB Ethernet NICs **OSD Network:** 2x 1GB Ethernet NICs **Mgmt. Network:** 2x 1GB Ethernet NICs

推荐操作系统

Ceph 依赖

Linux 内核

Ceph 内核态客户端：

> - v3.16.3 or later (rbd deadlock regression in v3.16.[0-2])
> - NOT v3.15.* (rbd deadlock regression)
> - V3.14.*
> - v3.6.6 or later in the v3.6 stable series
> - v3.4.20 or later in the v3.4 stable series

btrfs: v3.14 或更新

系统平台 (FIREFLY 0.80)

Distro	Release	Code Name	Kernel	Notes	Testing
Ubuntu	12.04	Precise Pangolin	linux-3.2.0	1,2	B,I,C
Ubuntu	14.04	Trusty Tahr	linux-3.13.0		B,I,C
Debian	6.0	Squeeze	linux-2.6.32	1,2,3	B
Debian	7.0	Wheezy	linux-3.2.0	1,2	B
CentOS	6	N/A	linux-2.6.32	1,2	B,I
RHEL	6		linux-2.6.32	1,2	B,I,C
RHEL	7		linux-3.10.0		B,I,C
Fedora	19.0	Schrodinger's Cat	linux-3.10.0		B
Fedora	20.0	Heisenbug	linux-3.14.0		B

Note:

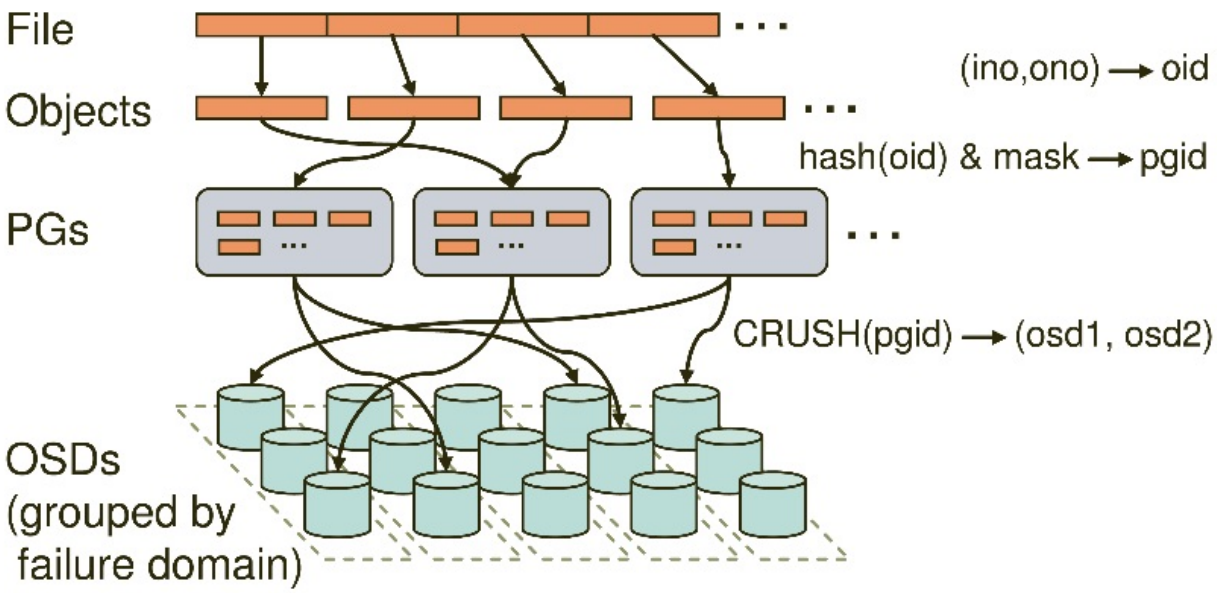
> - 1: 默认内核 btrfs 版本较老，不推荐用于 ceph-osd 存储节点；要升级到推荐的内核，或者改用 xfs,ext4
> - 2: 默认内核带的 Ceph 客户端较老，不推荐做内核空间客户端（内核 RBD 或 Ceph 文件系统），请升级到推荐内核。
> - 3: 默认内核或已安装的 glibc 版本若不支持 syncfs(2) 系统调用，同一台机器上使用 xfs 或 ext4 的 ceph-osd 守护进程性能不会如愿。

测试版：

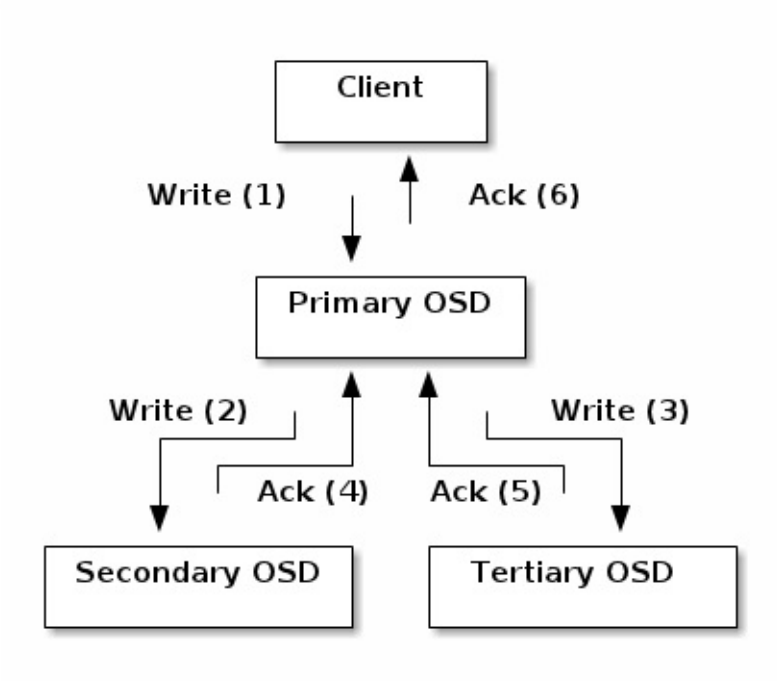
> - B: 我们持续地在这个平台上编译所有分支、做基本单元测试；也为这个平台构建可发布软件包。
> - I: 我们在这个平台上做基本的安装和功能测试。
> - C: 我们在这个平台上持续地做全面的功能、退化、压力测试，包括开发分支、预发布版本、正式发布版本。

数据流向

Data-->obj-->PG-->Pool-->OSD



数据复制



数据重分布

影响因素

OSD OSD weight OSD crush weight

Ceph 应用

RDB 为 Glance Cinder 提供镜像存储 提供 Qemu/KVM 驱动支持 支持 openstack 的虚拟机迁移

RGW 替换 swift 网盘

Cephfs 提供共享的文件系统存储 支持 openstack 的虚拟机迁移

ceph 安装及应用

- ceph 安装及应用

ceph 安装及应用部分目前更新在 `ceph 实践` 的 wiki 中

MooseFS

MooseFS 简介

本文介绍 `MooseFS` 架构原理，安装配置和使用方法。

`MooseFS` 是一种容错的分布式文件系统。它将数据分散到多个物理位置（服务器），在用户看来是一个统一的资源。 `MooseFS` 支持 `FUSE` 标准接口，能够无缝实现从本地文件的迁移。同时，`MooseFS` 提供比 `NFS` 更好的可运维性。

功能特性

对于标准的文件操作，`MooseFS` 表现与其它类 Unix 文件系统一致。

支持的通用文件系统特性有：

- 层次结构（目录数），是一种操作友好的文件系统。
- 兼容 POSIX 文件属性
- 支持特殊文件（块设备与字符设备，管道和套接字）
- 符号链接（软链接）和硬链接。
- 基于 IP 地址和（或）密码的访问权限控制

`MooseFS` 独特的特性有：

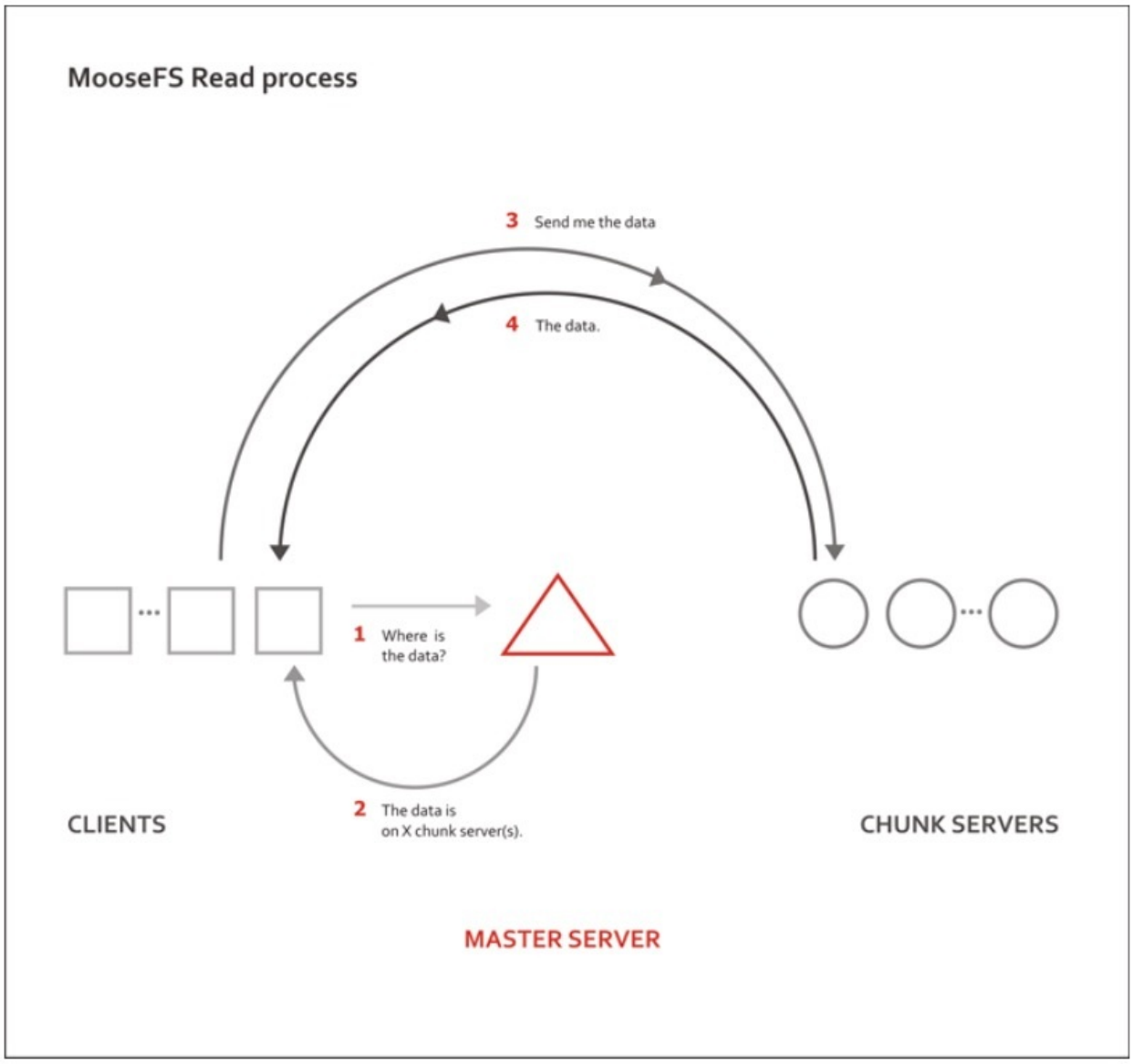
- 高可靠性（数据的多个副本存储在多个不同服务器上）
- 容量动态扩展。只要增加新的机器／磁盘
- 删除的文件保留一段时间（可配），像是文件系统的回收站。
- 即使文件在被读写，也可以持续做文件快照。

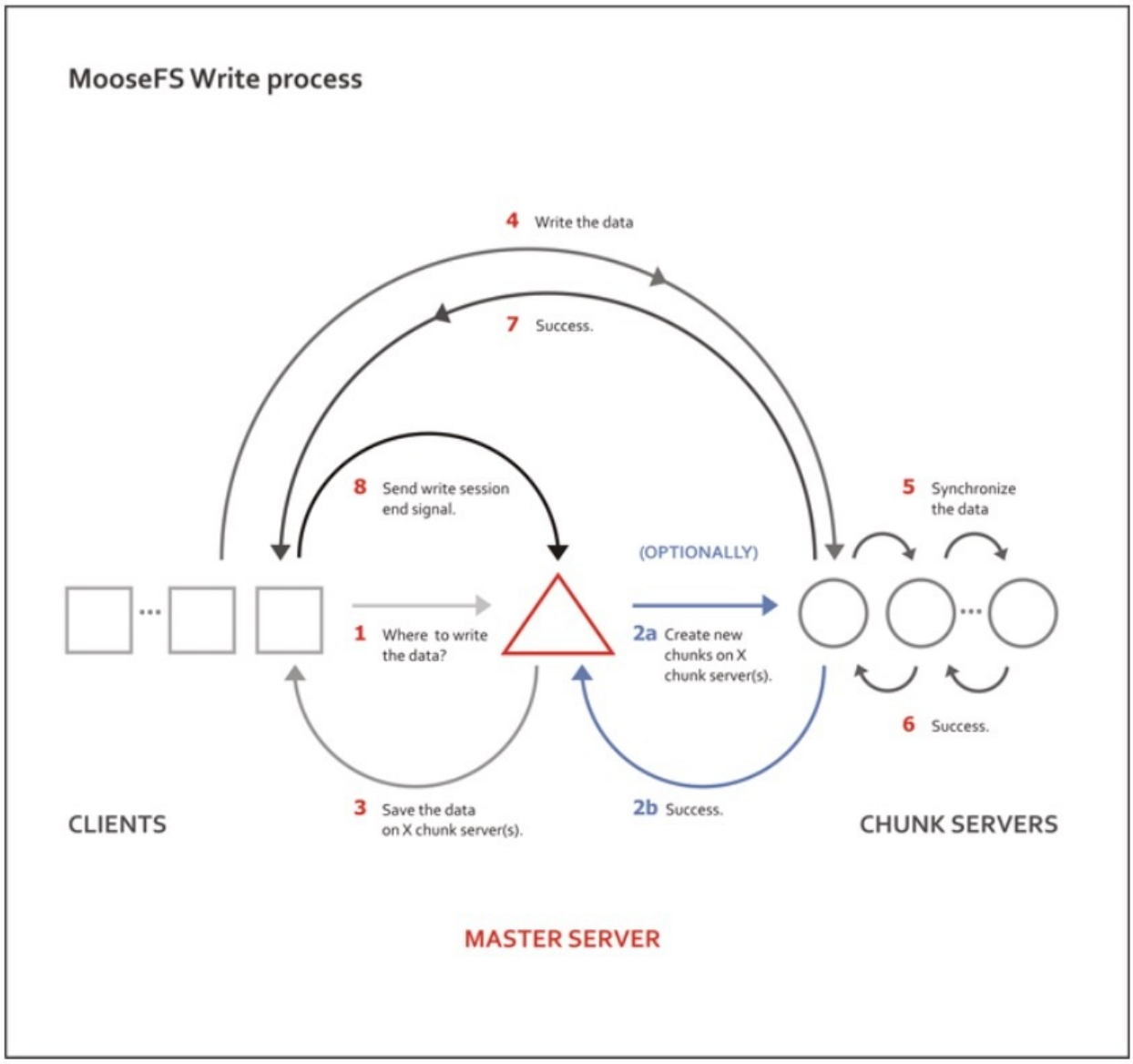
架构原理

MooseFS 包含 4 个组件

- 管理节点 master servers。支持单活。存储所有文件的元数据
- 数据节点 chunk servers 数量不限。存储文件数据，相互同步文件。
- 元数据备份服务器 metalogger server。数量不限。保存元数据变更日志，周期性的下载元数据文件。主节点失效时可以替代主节点。
- 客户端。挂载使用文件系统。

文件的读写流程可以根据以下图示来理解：


MooseFS Read process
3 Send me the data
4 The data.
1 Where is the data?
2 The data is on X chunk server(s).
CLIENTS
CHUNK SERVERS
MASTER SERVER



关于分片

文件数据分片（chunks）后保存，分片默认最大值为 64MiB。分片对应 chunkserver 中的文件。 分片数据是版本化的，如果文件执行更新后，某台机器还有旧版本的数据，则会删除该机器上的旧版本文件，并同步到改文件的最新版本。

容错／高可靠

将文件分发多份到多个服务器中存储，以实现高可靠。通过设置单个文件的 `goal` 来指定文件应该可以保留的副本数。 重要数据建议将 goal 设置大于 2；而将 goal 设为 1，则文件只在 1 台数据节点上保存

chunk 存储选择算法

如果自己设计一套 chunkserver 选择算法，我们要达到哪些目标呢？

> 1. 文件打散后尽量平均分布到各台 chunkserver 上
> 2. 各台 chunkserver 上的 chunk 数量尽可能的平均
> 3. 数据分发过程衡量系统负载，尽量把数据放在负载低的 chunkserver 上
> 4. 数据分发过程是否应该衡量各台 chunkserver 的可用空间？
> 5. 机架感应？

回到 MFS 使用过程中会有一个疑问.chunkserver 的选择是怎么选择的. 怎么才能保证数据保存占用空间平衡甚至平均？这就是数据分布算法. 也正是分布式文件系统的核心容. 所以在此，转来一篇关于 MFS 的 chunk 存储选择算法的文章。

还记得 matocsserventry 结构中的 carry 字段么，这个字段就是分布算法的核心. 每台 chunkserver 会有自己的 carry 值，在选择 chunkserver 会将每台 chunkserver 按照 carry 从大到小做快速排序，优先选择 carry 值大的 chunkserver 来使用。

在描述具体算法前，先介绍三个概念：

> - allcnt:mfs 中可用的 chunkserver 的个数
> - availcnt:mfs 中当前可以直接存储数据的 chunkserver 的个数
> - demand: 当前文件的副本数目

先说 allcnt, 可用的 chunkserver 要满足下面几个条件：

> 1. chunkserver 是活着的
> 2. chunkserver 的总空间大于 0
> 3. chunkserver 的可用空间（总空间 - 使用空间）大于 1G

availcnt 指的是 carry 值大于 1 的可用 chunkserver 的个数. 也就是在 allcnt 的约束条件上加一条 carry 值大于 1. 文件 1.txt 需要存储 2 个副本，但是 mfs 中仅仅有 1 台 chunkserver 可用，也就是 `demand>allcnt` 的时候，mfs 会自动减少文件的副本个数到 allcnt, 保证文件可以成功写入系统。

关于 carry 有下面几个规则：

> 1. 仅 carry 值大于 1 的 chunkserver 可以存储新数据
> 2. 每台 chunkserver 存储新数据后其 carry 会减 1
> 3. demand>availcnt 的时候，会递归的增加每台 chunkserver 的 carry 值，直到 `demand<=availcnt` 为止
> 4. 每台 chunkserver 每次 carry 值的增加量等于当前 chunkserver 总空间除以最大的 chunkserver 总空间

上面的规则比较复杂. 举个例子就更加清晰了。

```
chunkserver 1：totalspace:3.94G carry:0.463254
chunkserver 2：totalspace:7.87G carry:0.885674
```

文件 1.txt 大小 1k,mfs 默认一个 chunk 大小为 64M, 所以仅仅需要一个 chunk 就够了. 此时 availcnt=0,demand=1, 所以需要增加 carry 值

```
chunkserver 1：carry=0.463254 + (3.94/7.87) = 0.463254 + 0.500005 = 0.963259
chunkserver 2：carry=0.885674 + (7.87/7.87) = 0.885674 + 1.000000 = 1.885674
```

此时 availcnt=1,demand=1, 所以不需要增加 carry 值，对 chunkserver 按照 carry 从大到小排序结果为：`chunkserver 2 > chunkserver 1`, 文件 1.txt 的 chunk 会存储到 chunkserver 2 上，同时 chunkserver 2 的 carry 会减 1

如下：

```
chunkserver 1：carry=0.963259
chunkserver 2：carry=1.885674 – 1 = 0.885674
```

文件 2.txt 大小 1k,mfs 默认一个 chunk 大小为 64M, 所以仅仅需要一个 chunk 就够了. 此时 availcnt=0,demand=1. 所以需要增加 carry 值

```
chunkserver 1：carry=0.963259 + (3.94/7.87) = 0.963259 + 0.500005 = 1.463264
chunkserver 2：carry=0.885674 + (7.87/7.87) = 0.885674 + 1.000000 = 1.885674
```

此时 availcnt=2,demand=1, 所以不需要增加 carry 值，对 chunkserver 按照 carry 从大到小排序结果为：`chunkserver 2 > chunkserver 1`, 文件 2.txt 的 chunk 会存储到 chunkserver 2 上，同时 chunkserver 2 的 carry 会减 1

如下：

```
chunkserver 1：carry=1.463264
chunkserver 2：carry=1.885674 – 1 = 0.885674
```

文件 3.txt 大小 1k,mfs 默认一个 chunk 大小为 64M, 所以仅仅需要一个 chunk 就够了. 此时 availcnt=1,demand=1, 所以不需要增加 carry 值. 对 chunkserver 按照 carry 从大到小排序结果为：`chunkserver 1 > chunkserver 2`, 文件 3.txt 的 chunk 会存储到 chunkserver 1 上，同时 chunkserver 1 的 carry 会减 1

如下：

```
chunkserver 1：carry=1.463264 – 1 = 0.463264
chunkserver 2：carry=0.885674
```

因为两台 chunkserver 的总空间大小不一致，根据算法总空间大的那台 chunkserver 会存储更多的新数据。

记住：仅仅和 **chunkserver** 的总空间有关系和可用空间没有任何关系, 也就是说，当各台 chunkserver 总空间大小差不多的情况下，chunk 能更好的平均分布，否则 mfs 会更倾向于选择总空间大的机器来使用。

最后一个问题，当 mfs 刚刚启动的时候，carry 值是如果获得的？

答案：随机产生，通过 rndu32() 这个函数，随机产生一个小于 1, 大于等于 0 的数。

测试结果如下：

```
Nov 23 01:01:25 sunwg mfsmaster[13175]: 192.168.0.159,0.594834
Nov 23 01:01:25 sunwg mfsmaster[13175]: 192.168.0.160,0.000000
Nov 23 01:03:58 sunwg mfsmaster[13187]: 192.168.0.159,0.516242
Nov 23 01:03:58 sunwg mfsmaster[13187]: 192.168.0.160,0.826559
Nov 23 01:04:17 sunwg mfsmaster[13192]: 192.168.0.159,0.123765
Nov 23 01:04:17 sunwg mfsmaster[13192]: 192.168.0.160,0.389592
```

安装及使用

配置要求

管理节点 是系统的核心，需要使用稳定性高的硬件设备，如冗余电源，ECC 内存，RAID1/RAID5/RAID10。 根据文件数量的不同，也需要配置比较多的内存（一般来说，100 万个文件对应 300MiB 内存）。 硬盘容量需要考虑文件数量和文件操作数量（一般来说，20GiB 磁盘可以保存 2500 万文件的元数据，或者 50 小时的文件操作日志）。 管理节点如此重要，也需要根据情况做好安全设置。

元数据备份服务器 只需要和管理节点有同样多的内存和磁盘来存储数据即可。

数据节点 只需要保持足够的磁盘容量。

安装

在 CentOS 系统上安装。

首先配置使用软件仓库。

将 moosefs 仓库到 GPG KEY 加入本地软件包管理工具。

```
curl "http://ppa.moosefs.com/RPM-GPG-KEY-MooseFS" -o /etc/pki/rpm-gpg/RPM-GPG-KEY-Moos
eFS
```

增加 MooseFS3.0 仓库配置项。

```
curl "http://ppa.moosefs.com/MooseFS-3-el$(grep -o '[0-9]*' /etc/centos-release |head
-1).repo" -o /etc/yum.repos.d/MooseFS.repo
```

使用以下命令来安装软件包：

```
# For Master Server:
yum install moosefs-master moosefs-cli moosefs-cgi moosefs-cgiserv

# For Chunkservers:
yum install moosefs-chunkserver

# For Metaloggers:
yum install moosefs-metalogger

# For Clients:
yum install moosefs-client
```

启动文件系统

```
# To start process manually:
mfsmaster start
mfschunkserver start
# For sysv os family - EL5, EL6:
service moosefs-master start
service moosefs-chunkserver start
# For systemd os family - EL7:
systemctl start moosefs-master.service
systemctl start moosefs-chunkserver.service
```

系统参数

```
# for master server
sysctl vm.overcommit_memory=1
```

使用方法

挂载

```
$ sudo mfsmount -H <mfs-master> [-P 9421] [-S /] [-o rw|ro] /mnt/mfs
```

其中，`-H` / `-P` 代表 mfsmaster 的 IP 和端口；`-S` 挂载 MooseFS 中的路径；`-o rw` 或 `-o ro` 设置读写或只读模式；`/mnt/mfs` 为本地挂载路径。

设置冗余度

通过配置冗余度来保证出现失效时不丢失数据。 冗余度为 N 时，能够在不超过 N-1 个 chunkserver 同时出现失效时不丢失数据。

默认我们设置文件的冗余度为 2，即支持有 1 个 chunkserver 失败时不影响使用。

调整数据冗余度 (goal)

```
$ sudo mfssetgoal -r 2 /mnt/mfs
```

其中，`-r` 选项代表递归目录及子目录的文件。

可以通过 `mfsgetgoal` 来读取当前的冗余度

```
$ sudo mfsgetgoal /mnt/mfs
/mnt/mfs 2
```

可以通过 `mfscheckfile` 来读取特定文件的冗余度设置与生效情况

```
$ sudo mfscheckfile /mnt/mfs/testfile
/mnt/mfs/testfile:
2 copies: 1 chunks
```

建议：

- 最低设置为 2，保证不出现文件丢失；
- 一般情况下设置为 3，应该是足够安全的；
- 对于足够重要的数据，可以设置为 4 或者更高，但是不能超过 chunkserver 实例数量。

设置 **Trash time**

文件删除后会在 moosefs 的垃圾站中保留一段时间。 通过 `mfsgettrashtime` 能读取过期时间的设置。

使用快照

使用 MooseFS 的一个好处是可以支持文件或目录的快照。 我们知道 MooseFS 的分块都是版本化的，因此支持快照的方式保留一个文件的副本。 在文件被修改前，这个副本并不会占用额外的空间。

启动顺序与停止顺序

启动顺序

- 启动 mfsmaster
- 启动 mfschunkserver
- 启动 mfsmetalogger
- 在 client 节点执行 mfsmount

停止顺序

- 在所有 client 节点执行 umount
- mfschunkserver stop
- mfsmetalogger stop
- mfsmaster stop

数据恢复

当出现 master 节点出现问题时，可以通过 `mfsmetarestore` 来恢复元数据。

```
mfsmetarestore -a -d /storage/mfsmaster
```

如果 master 节点故障严重无法启动，可以利用 metalogger 节点的元数据备份来恢复。 首先在选定的节点上按照 mfsmaster，使用之前 master 节点的相同配置； 从备份设备或 metalogger 拷贝 `metadata.mfs.back` 文件到新 master 节点； 从 metalogger 拷贝失败前元数据最新的 changelog 文件（ `changelog.*.mfs` ）； 执行 `mfsmetarestore -a` 。

Automated Failover

生产环境使用 MooseFS 时，需要保证 master 节点的高可用。 使用 `ucarp` 是一种比较成熟的方案。

`ucarp` 类似于 `keepalived` ，通过主备服务器间的健康检查来发现集群状态，并执行相应操作。

日常问题及修复方法

Client 操作与修复

客户端强制 `kill -9` 杀掉 `mfsmount` 进程，需要先 `umount` , 然后再 `mount` , 否则会提示：

```
fuse: bad mount point `/mnt/test/': Transport endpoint is not connected
see: /data/jingbo.li/mfs/bin/mfsmount -h for help
```

关于修复

使用过程中遭遇 master 断电导致服务停止，可以使用 `mfsmetarestore -a` 修复才能启动，如果无法修复，使用 `metalogger` 上的备份日志进行恢复： `mfsmetarestore -m metadata.mfs.back -o metadata.mfs changelog_ml.*.mfs` , 但是此方法也不是万能的，但凡此过程 chunks 块出现问题，可以使用 `mfsfilerepair` 进行修复。

`mfsfilerepair` 主要是处理坏文件的（如写操作引起的 I/O 错误）使文件能够部分可读. 作用如下：在丢失块的情况下使用 0 对丢失文件进行填充；在块的版本号不匹配时设置快的版本号为 master 上已知的能在 chunkerservers 找到的最高版本号；

注意：

> 因为在第二种情况的内容不匹配，可能发生在块具有相同的版本，建议进行文件的拷贝（而不是进行不快照！), 并删除原始文件再进行文件的修复. 恢复后会有文件丢失或损坏。

Chunker 的空间

查看 MooseFS 文件的使用情况，请使用 `mfsdirinfo` 命令，相当于 `df` .

快照 snapshot

可以快照任何一个文件或目录，语法： `mfsmakesnapshot src dst` , 但是 src 和 dst 必须都属于 mfs 体系，即不能 mfs 体系中的文件快照到其他文件系统。

---

mfsappendchunks

追加 chunks 到一个文件，追加文件块到另一个文件. 如果目标文件不存在，则会创建一个空文件，然后继续将块进行追加。

回收站机制

其实 MFS 有类似 windows 的回收站这种机制，当文件不小心删除了，不用担心，去回收站去找. 随时可以恢复. 当然，我所说的随时随地恢复要看你回收站的数据保存多长时间了（默认 24 小时）.

- 首先挂载辅助系统

单独安装或挂载 **MFSMETA** 文件系统，它包含目录 /trash （包含仍然可以被还原的删除文件的信息）和 `/trash/undel` （用于获取文件）, 用一个 `-m` 或 `-o mfsmeta` 的选项，这样可以挂接一个辅助的文件系统 MFSMETA, 这么做的目的是对于意外的从 MooseFS 卷上删除文件或者是为了释放磁盘空间而移动的文件而又此文件又过去了垃圾文件存放期的恢复。

> 例如：

```
mfsmount -m /mnt/mfsmeta -H mfs1.com.org
或者
mfsmount -o mfsmeta -H mfs1.com.org /mnt/mfsmeta
```

需要注意的是，如果要挂载 mfsmeta, 一定要在 mfsmaster 的 mfsexports.cfg 文件中加入如下条目：* . rw

挂载后在 /mnt/mfsmeta 目录下分 reserved 和 trash 两个目录，trash 为已删除文件存放目录，删除时间根据 mfsgettrashtime 设置时间来自动删除。

- 设置文件或目录的删除时间

> 一个删除的文件能够存放在“ 垃圾箱”中的时间称为隔离时间，这个时间可以用 `mfsgettrashtime` 命令来查看： 默认时间为 86400, 即时间为 24 小时

```
[root@linux mnt]# mfsgettrashtime filename（某文件名）
filename: 86400
```

> 用 `mfssettrashtime` 命令来设置上面的这个有效时间，要注意的是，保存时间单位为秒。

```
[root@Linux mnt]# mfssettrashtime 60 filename
filename: 60
```

- 恢复删除的文件

把删除的文件移到 /trash/undel 下，就可以恢复此文件. 在 MFSMETA 的目录里，除了 `trash` 和 `trash/undel` 两个目录，还有第三个目录 `reserved` , 该目录内有已经删除的文件，但却被其他用户一直打开着。 在用户关闭了这些被打开的文件后， `reserved` 目录中的文件将被删除，文件的数据也将被立即删除. 此目录不能进行操作。

MFS 集群的维护

启动和停止 MFS 集群

启动

最安全的启动 MooseFS 集群（避免任何读或写的错误数据或类似的问题）的方式是按照以下命令步骤：

```
(1) 启动 mfsmaster 进程
(2) 启动所有的 mfschunkserver 进程
(3) 启动 mfsmetalogger 进程（如果配置了 mfsmetalogger）
(4) 当所有的 chunkservers 连接到 MooseFS master 后，任何数目的客户端可以利用 mfsmount 去挂接被 export 的文件系统。（可以通过检查 master 的日志或是 CGI 监视器来查看是否所有的 chunkserver 被连接）。
```

停止

```
(1) 在所有的客户端卸载 MooseFS 文件系统（用 umount 命令或者是其它等效的命令）
(2) 用 mfschunkserver stop 命令停止 chunkserver 进程
(3) 用 mfsmetalogger stop 命令停止 metalogger 进程
(4) 用 mfsmaster stop 命令停止 master 进程
```

MFS chunkservers 的维护

若每个文件的 goal（目标）都不小于 2，并且没有 under-goal 文件（这些可以用 mfsgetgoal -r 和 mfsdirinfo 命令来检查），那么一个单一的 chunkserver 在任何时刻都可能做停止或者是重新启动。以后每当需要做停止或者是重新启动另一个 chunkserver 的时候，要确定之前的 chunkserver 被连接，而且要没有 under-goal chunks。

MFS 元数据备份

通常元数据有两部分的数据：

> - 主要元数据文件 metadata.mfs，当 mfsmaster 运行的时候会被命名为 metadata.mfs.back
> - 元数据改变日志 changelog.*.mfs，存储了过去的 N 小时的文件改变（N 的数值是由 BACK_LOGS 参数设置的，参数的设置在 mfschunkserver.cfg 配置文件中）。 主要的元数据文件需要定期备份，备份的频率取决于取决于多少小时 changelogs 储存。 元数据 changelogs 实时的自动复制。

MFS Master 的恢复

一旦 mfsmaster 崩溃（例如因为主机或电源失败），需要最后一个元数据日志 changelog 并入主要的 metadata 中。这个操作时通过 mfsmetarestore 工具做的，最简单的方法是：

$/usr/local/mfs/bin/mfsmetarestore -a

如果 master 数据被存储在 MooseFS 编译指定地点外的路径，则要利用 -d 参数指定使用，如：

$/usr/local/mfs/bin/mfsmetarestore -a -d /opt/mfsmaster

从 MetaLogger 中恢复 Master

有些童鞋提到：如果 mfsmetarestore -a 无法修复，则使用 metalogger 也可能无法修复，暂时没遇到过这种情况，这里不暂不考虑。 找回 metadata.mfs.back 文件，可以从备份中找，也可以中 metalogger 主机中找（如果启动了 metalogger 服务），然后把 metadata.mfs.back 放入 data 目录，一般为{prefix}/var/mfs 从在 master 宕掉之前的任何运行 metalogger 服务的服务器上拷贝最后 metadata 文件，然后放入 mfsmaster 的数据目录。 利用 mfsmetarestore 命令合并元数据 changelogs，可以用自动恢复模式 mfsmetarestore –a，也可以利用非自动化恢复模式 $mfsmetarestore -m metadata.mfs.back -o metadata.mfs changelog_ml.*.mfs 或：强制使用 metadata.mfs.back 创建 metadata.mfs，可以启动 master，但丢失的数据暂无法确定。

物理机，云服务及虚拟化篇

- 物理机
- AWS
- 阿里云
- KVM
- Docker
- OpenStack
- K8s

物理机常见问题即处理方法

开机无法启动

提示无法找到系统盘

场景

现象

```
Reboot and select proper boot device or insert boot media in selected boot device and
press a key
```

可能原因

先排查硬件原因 ---> 再排查软件原因

硬件原因

> - RAID 卡（可以开机时查看 RAID 卡 能否识别到硬盘）
> - 硬盘（硬盘状态灯是否正常）

软件原因

> - 开机启动项顺序
> - RAID 卡设置的启动盘配置
> - 引导项丢失

重做引导项方法

> - 插入光盘进入救援模式（选择完英文等，选择 continue 进入救援模式）
> - chroot /mnt/sysimage
> - /sbin/grub-install /dev/sda(sda 是系统所在的设备）
> - 多系统加引导项的话可以添加到 /boot/grub/grub.conf
> - 重启服务器

AWS

0 AWS 产品

计算

- Amazon EC2 云中的虚拟服务器
- Auto Scaling
- Amazon VPC 隔离的云资源
- Elastic Load Balancing

联网

- Amazon VPC 隔离的云资源
- AWS Direct Connect AWS 的专用网络连接
- Amazon Route 53 可扩展的域名系统 (DNS)
- Elastic Load Balancing

存储和内容分发

- Amazon S3 可扩展的云存储
- Amazon Glacier 云中的低成本归档存储
- Amazon EBS EC2 块存储卷
- AWS Storage Gateway 将内部 IT 环境与云存储相集成
- Amazon CloudFront 全球内容分发网络 (CDN)
- AWS Import/Export 大容量数据传输

数据库

- Amazon RDS 适用于 MySQL、Oracle、SQL Server 和 PostgreSQL 的管理型关系数据库服务
- Amazon DynamoDB 快速、可预测、高可扩展的 NoSQL 数据存储
- Amazon Redshift 快速、功能强大的完全管理型 PB 级数据仓库服务
- Amazon ElastiCache 基于内存的缓存服务

分析

- Amazon Kinesis 实时数据流处理
- Amazon Redshift 快速、功能强大的完全管理型 PB 级数据仓库服务
- Amazon EMR 托管的 Hadoop 框架
- AWS Data Pipeline 适用于周期性数据驱动工作流的编排服务

应用程序服务

- Amazon CloudSearch 托管的搜索服务
- Amazon AppStream 低延迟应用程序流媒体传输
- Amazon SES 电子邮件发送服务
- Amazon SQS 消息队列服务
- Amazon SNS 推送通知服务
- Amazon SWF 用于协调应用程序组件的工作流服务

- Amazon FPS 基于 API 的付款服务
- Amazon Elastic Transcoder 易用型可扩展媒体转码

部署与管理

- AWS Elastic Beanstalk AWS 应用程序容器
- AWS OpsWorks DevOps 应用程序管理服务
- AWS CloudFormation AWS 资源创建模板
- AWS CloudTrail 用户活动与变更追踪
- Amazon CloudWatch 资源与应用程序监控
- AWS Identity and Access Management (IAM) 可配置的 AWS 访问控制
- AWS CloudHSM 有助于实现监管合规性的基于硬件的密钥存储
- AWS 管理控制台 基于 Web 的用户界面
- AWS 命令行界面 管理 AWS 服务的统一工具

移动服务

- Amazon Cognito 用户身份和数据同步
- Amazon Mobile Analytics 快速、安全的应用程序使用情况分析
- Amazon SNS 跨越多种平台发送通知、更新和促销信息
- AWS 移动软件开发工具包 快速轻松开发高质量移动应用程序

应用程序

- Amazon WorkSpaces - 云中的虚拟桌面
- Amazon Zocalo - 安全的企业文档存储和共享

1 使用 AWS S3

S3 是 Simple Storage Service 的缩写，是 AWS 提供的云存储服务，价格公道、服务稳定，因此被广泛应用在静态文件存储、内容备份、大数据分析领域。

基本概念

在使用 S3 前首先需要了解一些基本概念：

- 对象：即文件。
- Bucket：官方翻译为存储桶，是在网络存储服务中广泛使用的一个概念，通常用于区分文件所在区域，可以对比操作系统不同盘符来理解。
- AWS CLI：AWS 提供的控制台工具，aws-cli 是其在 PyPI 上注册的名字，shell 中的命令为 `aws`。
- AWS IAM：即 Identity and Access Management，是 AWS 的身份认证服务，用于对用

户身份及资源进行授权管理，需要配置号已授权的 AWS 认证信息才可以使用其服务。

- S3 资源：S3 资源以 `s3://bucket-name/` 开始。

基本操作

创建以及管理 Bucket

就像你首先需要有硬盘以及挂载盘符才能在本地操作文件一样，正式使用 S3 提供服务前你同样需要准备好 Bucket：

```
aws s3 mb s3://bucket-name
```

Bucket 名称必须唯一，并且应符合 DNS 标准：可以包含小写字母、数字、连字符（-）和点号（.），只能以字母或数字开头和结尾，连字符或点号后不能跟点号。

想要列出以创建的 Bucket 可以使用 ls 命令：

```
$ aws s3 ls
       CreationTime Bucket
       ------------ ------
2015-11-11 11:11:11 my-bucket
2015-11-11 11:11:11 my-bucket1
```

删除空 Bucket 可以使用 `aws s3 rb s3://bucket-name`，非空 Bucket 需要加上 `--force` 命令，这一点对于管理文件对象也是一样。

`mb` 、 `rb` 命令分别是 `MakeBucket` 和 `RemoveBucket` 的缩写。

操作文件

S3 的服务基于文件对象，提供了类似 Linux 环境下的文件访问操作：

```
aws s3 ls s3://bucket-name[/folder]/
aws s3 cp s3://bucket-name/file s3://bucket-name[/folder]/
aws s3 mv s3://bucket-name/file s3://bucket-name[/folder]/
aws s3 rm s3://bucket-name/file
aws s3 rm s3://bucket-name[/folder]/ --recursive
```

操作文件的方式基本上和 Linux 环境类似，`--recursive` 用于递归调用，类似 Linux 命令的 `-r` 参数。

同步本地文件、目录

AWS CLI 还提供了一个本地文件同步命令 `sync` ：

```
aws s3 sync local_dir s3://my-bucket/MyFolder
```

`sync` 会递归地将本地文件复制到 S3 Bucket 中，如存在重名文件则覆盖处理。如果需要像同步盘一样删除已不存在的文件可以加上 `--delete` 命令。

`sync` 还可以通过 `--exclude` 和 `--include` 来指定不同步的目录，以及不同步的目录中的例外。

权限控制

默认情况下，所有 Amazon S3 资源都是私有的，包括存储桶、对象和相关子资源（例如，lifecycle 配置和 website 配置）。只有资源拥有者，即创建该资源的 ***AWS*** 账户可以访问该资源。资源拥有者可以选择通过编写访问策略授予他人访问权限

Amazon S3 提供的访问策略

> - 基于资源的策略 ------- 存储桶策略和访问控制列表 (ACL – 注：每个存储桶和对象都有关联的 ACL)
> - 基于用户策略 ------ 使用 IAM 管理，IAM 用户必须拥有两种权限：一种权限来自其父账户，另一种权限来自要访问的资源的拥有者 AWS 账户。

实际应用

每天凌晨备份 **Postgres** 数据库

```
pg_dump -Fc --no-owner  bxzz_exercise > /tmp/tmp.pgdump && aws s3 cp tmp.pgdump s3://f
ilebackup/pg/$(date +%Y/%m/%d).pgdump
```

crontab 设置：

```
0 2 * * * pg_dump -Fc --no-owner  bxzz_exercise > /tmp/tmp.pgdump && aws s3 cp tmp.pgd
ump s3://filebackup/pg/$(date +\%Y/\%m/\%d).pgdump
```

将 **S3** 当作同步盘

检测到文件变动时触发：

```
aws s3 sync . s3://my-bucket/MyFolder --exclude '*.txt' --include 'MyFile*.txt' --excl
ude 'MyFile?.txt'
```

参考：AWS S3 官方中文文档

2 Amazon IAM（身份及访问管理）

IAM enables you to control who can do what in your AWS account.

-

存储桶策略示例

创建存储桶后需要创建个 IAM 用户和关联下权限

创建对某个存储桶有所有权限实例

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListAllMyBuckets"
            ],
            "Resource": "arn:aws-cn:s3:::*"
        },
        {
            "Effect": "Allow",
            "Action": "s3:*",
            "Resource": [
                "arn:aws-cn:s3:::backet-name/*"
             ]
        }
   ]
}
```

注：Amazon 资源名称 (ARN) 和 AWS 服务命名空间

arn:partition:service:region:account-id:resource

partition: 资源所处的分区。对于标准 AWS 区域，分区是 aws。如果资源位于其他分区，则分区是 aws-partitionname。例如，位于 中国（北京） 区域的资源的分区为 aws-cn。

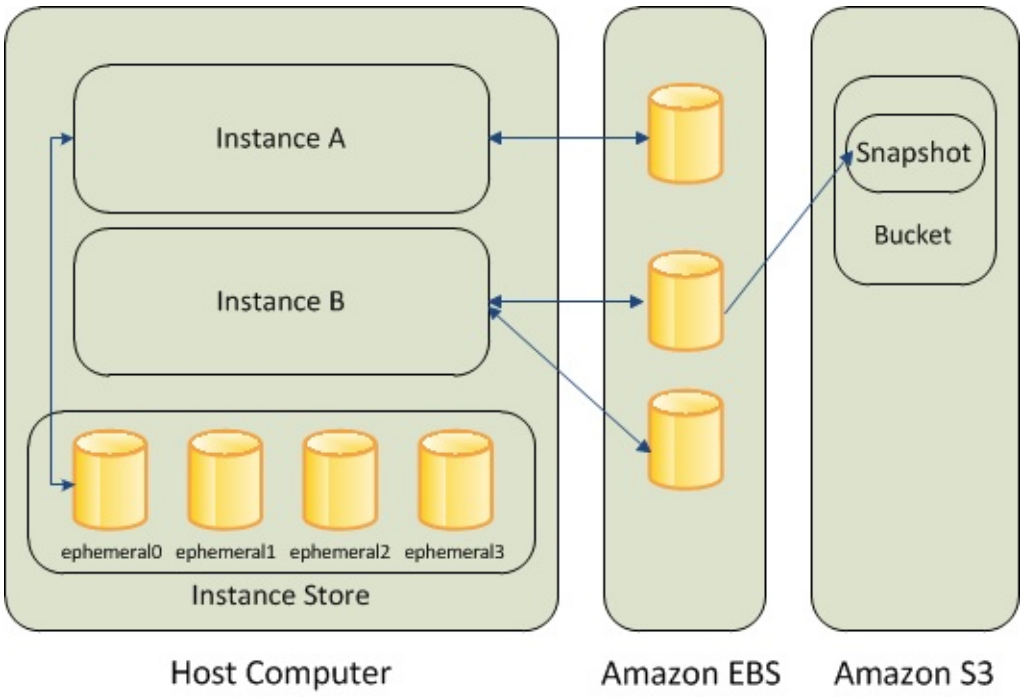
3 EC2

重启与停止以及终止之间的区别

性能	重启	停止 / 启动（仅限 **Amazon EBS** 支持的实例）	终止
主机	实例保持在同一主机上运行	实例在新主机上运行	无
私有和公有 IP 地址	这些地址保持不变	EC2-Classic：实例获得新的私有和公有 IP 地址 EC2-VPC：实例保留其私有 IP 地址。实例获取新的公有 IP 地址，除非它具有弹性 IP 地址 (EIP)（该地址在停止 / 启动过程中不更改）。	无
弹性 IP 地址 (EIP)	EIP 保持与实例关联	EC2-Classic：EIP 不再与实例关联，EC2-VPC：EIP 保持与实例关联	EIP 不再与实例关联
实例存储卷	数据保留	数据将擦除	数据将擦除
根设备卷	卷将保留	卷将保留	默认情况下将删除卷
记账功能	实例计费小时不更改	实例的状态一旦变为 stopping，就不再产生与该实例相关的费用。每次实例从 stopped 转换为 pending 时，我们都会启动新的实例计费小时	实例的状态一旦变为 shutting-down，就不再产生与该实例相关的费用

存储

> - Amazon EBS
> - Amazon EC2 实例存储卷（停止实例时，会删除数据，只有部分实例有）
> - Amazon S3



4 AWS VPC

VPC 中几个概念

VPC

- VPC 即 virtual private cloud，是个虚拟的局域网
- AWS 云中的一个私有的、隔离的部分
- 可自定义的虚拟网络拓扑

子网

VPC 是为了将你的所有服务与外界隔离开来，但是范围比较大，如果你的局域网内部还需要进一步的网络划分，那么需要设置子网。子网位于 VPC 内部。

路由器

这个页面上没有

路由表

路由表创建在 VPC 上，创建时需要选择一个对应的 VPC

在 VPC 内创建的所有路由表都会包含一条到达该 VPC 的路由项，而且不能删除。可以在此基础上再添加新路由项，如 Internet 网关。

主要功能是将消息从 VPC 内发到 VPC 外，不是子网间使用的

Internet 网关

如果要上网，Internet 网关是必须的，创建好后还要将其关联到路由表。点击做导航“路由表”，在右面的列表选中一项，在下方的路由选项卡中可以点击“编辑”添加 Internet 网关

安全组

安全组是入站规则与出站规则的集合。安全组同样是建立在 VPC 上的，创建时需要指定 VPC

VPC 的地区

- 区域
  - 相互隔离的地区区域
- 可用区 (AZ)
  - 数据中心

AWS 有 10 个区域、每个区域有多个可用区

VPC 规划

- 考虑将来的扩展
- VPC 可以从 /16 到 /28
- CIDR 不可修改
- 考虑将来是否需要与公司网络建立链接
- 重复的 IP 地址空间 = 未来的痛苦

5 AWS 客户端

AWS CLI

AWS CLI 是 AWS 提供的命令行工具，使用 Python 开发支持 Python 2.6.5 以上绝大多数 Python 版本。

安装

在 Unix/Linux 平台安装 AWS CLI 建议使用 pip：

```
pip install aws-cli
```

注意这里是"aws-cli"而不是"aws"

个人还推荐一个叫做 saws 的 aws-cli 封装包，提供了强大的命令补全功能：

```
pip install saws
```

配置

在使用 aws-cli 之前，你首先需要配置好个人身份信息以及偏好区域。配置个人身份信息前还需要注册 AWS ISM，并为自己的身份授予对应的权限。

AWS 的配置文件分 config 和 credentials，默认存储在 ~/.aws 目录中，格式如下：

```
# ~/.aws/credentials
[default]
aws_access_key_id=XXXXXXXXXXXXXXXXXXX
aws_secret_access_key=XXXXXXXXXXXXXXXXXXX
[dev]
aws_access_key_id=XXXXXXXXXXXXXXXXXXX
aws_secret_access_key=XXXXXXXXXXXXXXXXXXX
[s3]
aws_access_key_id=XXXXXXXXXXXXXXXXXXX
aws_secret_access_key=XXXXXXXXXXXXXXXXXXX
```

```
# ~/.aws/config
[default]
region=cn-north-1
output=json
[user1]
region=us-west-2
output=text
```

参数解释：

- `aws_access_key_id`：AWS 访问密钥。
- `aws_secret_access_key`：AWS 私有密钥。
- `aws_session_token`：AWS 会话令牌。只有在使用临时安全证书时才需要会话令牌。
- `region`：AWS 区域。
- `output`：输出格式（json、text 或 table）

环境变量

AWS CLI 支持以下变量：

- `AWS_ACCESS_KEY_ID`：AWS 访问密钥。
- `AWS_SECRET_ACCESS_KEY`：AWS 私有密钥。访问和私有密钥变量会覆盖证书和 config 文

件中存储的证书。

- `AWS_SESSION_TOKEN`：会话令牌。只有在使用临时安全证书时才需要会话令牌。
- `AWS_DEFAULT_REGION`：AWS 区域。如果设置，此变量会覆盖正在使用的配置文件的默认区域。
- `AWS_DEFAULT_PROFILE`：要使用的 CLI 配置文件的名称。可以是存储在证书或 config 文件中的配置文件的名称，也可以是 default，后者使用默认配置文件。
- `AWS_CONFIG_FILE`：CLI config 文件的路径。

命令行参数

参考：AWS 官方文档

使用

```
aws ec2 describe-instances --profile dev
aws ec2 describe-instances --profile default
aws s3api put-object --body /root/start.sh --bucket bucket-name --key "start.sh"  --pr
ofile s3
```

saws 工具

saws 是 aws-cli 封装包

S3

上传文件

前提：AWS 的配置中的访问密钥对 S3 的某 bucket-name 有权限

输入 saws 后输入

```
saws>aws s3api put-object --body /root/start.sh --bucket bucket-name --key "start.sh"
{
    "ETag": "\"2bdd5dd11b4273cfb0a807539325xxx\""
}
```

下载文件

前提：AWS 的配置中的访问密钥对 S3 的某 bucket-name 有权限

输入 saws 后输入

```
saws>aws s3api get-object --bucket bucket-name --key "start.sh" /root/start.sh2"
{
    "AcceptRanges": "bytes",
    "ContentType": "binary/octet-stream",
    "LastModified": "Thu, 13 Oct 2016 02:55:48 GMT",
    "ContentLength": 241,
    "ETag": "\"2bdd5dd11b4273cfb0a807539325xxx\"",
    "Metadata": {}
}
```

S3cmd

下载及配置

在 Linux 上 安装 s3 客户端

下载 s3cmd

下载后解压进入到目录中

连接 AWS S3 时可以通过 `s3cmd --config` 进行配置， `连接本地自有的存储可以配置如下`

在当前用户的家目录下创建 .s3cfg 文件、（也是 s3cmd 默认配置文件路径、），并填入以下内容：

```
[default]
access_key = XXXXXXXXXXXXXXXXXXX
secret_key = XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
host_base = 10.20.144.2
host_bucket = 10.20.144.2:80/%(bucket)
use_https = False
# 签名是 V2 的话设置为 True
signature_v2 = True
```

使用 S3cmd

配置完 S3cmd 就可以通过它来使用对象存储了。

对象存储中有两个非常重要的概念，bucket 和 object。object 对应需要存储的文件，而 bucket 作为 object 的存储空间。所以对象存储的操作主要涉及到的就是对 bucket 和 object 的操作。

1. 操作 bucket
    - 列举 **bucket**

```
# s3cmd ls
```

以列举当前的 bucket 为例：

```
# s3cmd ls
2015-08-12 03:56  s3://test-bucket_1
```

- 创建 **bucket**

```
# s3cmd mb s3://BUCKET
```

以创建名为 test_bucket_1 的 bucket 为例：

```
# s3cmd mb s3://test-bucket_1
Bucket 's3://test-bucket_1/' created
```

- 删除 **bucket**

```
# s3cmd rb s3://BUCKET
```

以删除我们刚创建的 test_bucket_1 为例：

```
# s3cmd rb s3://test-bucket_1
Bucket 's3://test-bucket_1/' removed
```

2. 操作 object

需要在 S3 中存储的文件在对象存储中被称为 object。为了说明上传 object 的过程，首先我们来创建一个用来上传的文件 test.txt （也就是一个 object), 并写入 samplecontent 作为文件的内容：

- 准备文件

```
# cat test.txt
samplecontent
```

- 创建 **bucket**

```
# s3cmd mb s3://test-bucket_2
Bucket 's3://test-bucket_2/' created
```

- 上传 **object**

```
s3cmd put FILE [FILE...] s3://BUCKET[/PREFIX]
```

以上传刚创建的 test.txt 文件到 test_bucket_2 bucket 为例：

```
# s3cmd put test.txt s3://test-bucket_2
WARNING: Module python-magic is not available. Guessing MIME types based on file e
xtensions.
test.txt -> s3://test-bucket_2/test.txt  [1 of 1]
14 of 14   100% in    0s    20.59 kB/s
14 of 14   100% in   90s     0.16 B/s  done
```

- 列出 **bucket** 中 **object**

```
# s3cmd ls [s3://BUCKET[/PREFIX]]
```

以列出当前 bucket 中的 object:

```
# s3cmd ls s3://test-bucket_2
2015-08-12 04:22        14   s3://test-bucket_2/test.txt
```

- 下载 **bucket** 中 **object**

```
# s3cmd get s3://BUCKET/OBJECT LOCAL_FILE
```

下载当前 bucket 中的文件 test.txt，并本地命名 localtest.txt：

```
# s3cmd get s3://test-bucket-2/test.txt  localtest.txt
s3://test-bucket-2/test.txt -> localtest.txt  [1 of 1]
s3://test-bucket-2/test.txt -> localtest.txt  [1 of 1]
348 of 348   100% in    0s    10.58 kB/s  done
```

- 删除 **bucket** 中 **object**

```
# s3cmd del s3://BUCKET/FILENAME
```

删除当前 bucket 中的 test.txt 对象：

```
# s3cmd del s3://test-bucket-2/test.txt
File s3://test-bucket-2/test.txt delete
```

- 拷贝 **bucket** 中 **object**

```
# s3cmd cp s3://BUCKET1/OBJECT1 s3://BUCKET2[/OBJECT2]
```

拷贝对象，从一个 bucket 到另一个 bucket：

```
# s3cmd cp s3://test-bucket-2/test1.txt s3://test-bucket-1/test1.txt
WARNING: Retrying failed request: /test1.txt ()
WARNING: Waiting 3 sec...
File s3://test-bucket-2/test1.txt copied to s3://test-bucket-1/test1.tx
```

- 获取 **object** 信息

```
# s3cmd info s3://BUCKET/OBJECT
```

获取当前 object 的信息：

```
# s3cmd info s3://test-bucket-2/test1.txt
s3://test-bucket-2/test1.txt (object):
File size: 348
Last mod:  Fri, 14 Aug 2015 02:02:37 GMT
MIME type: text/plain
MD5 sum:   4b49d7dd076b0b71e0eda307388fac57
SSE:       NONE
```

其余使用请参见 S3cmd usage: S3cmd 使用手册

阿里云

1 访问控制 (RAM)

1.1 创建 ECS 管理员

创建 **ECS** 管理员群组

点击访问控制 -->群组管理 -->新建群组

在创建好的群组上点击授权，选择 AliyunECSFullAccess（管理云服务器服务 (ECS) 的权限） 和 AliyunBSSOrderAccess（在费用中心 (BSS) 查看订单、支付订单及取消订单的权限）

创建用户

点击用户管理 -->新建用户...->加入 ECS 管理员群组

2 ECS

2.1 使用 API 控制 ECS

工具及简单使用说明

2.2 克隆实例

- 系统盘 ---- 通过创建自定义镜像的方式，创建一个自定义镜像，然后使用这个自定义镜像创建 ECS 即可。
- 数据盘 ---- 对已经配置完成的数据盘进行打快照，然后在购买或者升级页面，添加磁盘的地方点：“用快照创建磁盘”，选择你要的快照即可。

3 OSS

3.1 创建一个 OSS

3.1.1 Bucket

创建好后需要用到的是 Bucket 名字和 endpoint

3.1.2 访问策略（访问控制）

需要一个 AccessKey

使用 RAM 步骤

- (1) 自定义策略，使得策略只对新创建的 Bucket 有完全权限

```
{
  "Statement": [
      {
          "Action": "oss:*",
          "Effect": "Allow",
          "Resource": [
              "acs:oss:*:*:my-oss",
              "acs:oss:*:*:my-oss/*"
                    ]
              }
        ],
  "Version": "1"
}
```

- (2) 创建对应的群组，并授权对应的策略
- (3) 创建用户，并将此用户加入此群组中
- (4) 创建 AccessKey

3.2 OSSFS - 将 OSS 挂载到本地文件系统工具

3.2.1 简介

ossfs 能让您在 Linux/Mac OS X 系统中把 Aliyun OSS bucket 挂载到本地文件 系统中，您能够便捷的通过本地文件系统操作 OSS 上的对象，实现数据的共享。

3.2.2 功能

ossfs 基于 s3fs 构建，具有 s3fs 的全部功能。主要功能包括：

- 支持 POSIX 文件系统的大部分功能，包括文件读写，目录，链接操作，权限， uid/gid，以及扩展属性（extended attributes）
- 通过 OSS 的 multipart 功能上传大文件。
- MD5 校验保证数据完整性。

3.2.3 安装

预编译的安装包

我们为常见的 linux 发行版制作了安装包：

- Ubuntu-14.04
- CentOS-7.0/6.5/5.11

请从 版本发布页面 选择对应的安装包下载安装，建议选择最新版本。

- 对于 Ubuntu，安装命令为：

```
sudo apt-get update
sudo apt-get install gdebi-core
sudo gdebi your_ossfs_package
```

- 对于 CentOS6.5 及以上，安装命令为：

```
sudo yum localinstall your_ossfs_package
```

- 对于 CentOS5，安装命令为：

```
sudo yum localinstall your_ossfs_package --nogpgcheck
```

源码安装

如果没有找到对应的安装包，您也可以自行编译安装。编译前请先安装下列依赖库：

Ubuntu 14.04:

```
sudo apt-get install automake autotools-dev g++ git libcurl4-gnutls-dev \
                     libfuse-dev libssl-dev libxml2-dev make pkg-config
```

CentOS 7.0:

```
sudo yum install automake gcc-c++ git libcurl-devel libxml2-devel \
                 fuse-devel make openssl-devel
```

然后您可以在 github 上下载源码并编译安装：

```
git clone https://github.com/aliyun/ossfs.git
cd ossfs
./autogen.sh
./configure
make
sudo make install
```

3.2.4 运行

设置 Bucket name, access key/id 信息，将其存放在 /etc/passwd-ossfs 文件中， 注意这个文件的权限必须正确设置，建议设为 640。

```
echo my-bucket:my-access-key-id:my-access-key-secret > /etc/passwd-ossfs
chmod 640 /etc/passwd-ossfs
```

将 OSS Bucket mount 到指定目录

```
ossfs my-bucket my-mount-point -ourl=my-oss-endpoint
```

示例

将 `my-bucket` 这个 Bucket 挂载到 `/tmp/ossfs` 目录下，AccessKeyId 是 `faint` ，AccessKeySecret 是 `123` ，oss endpoint 是 `http://oss-cn-hangzhou.aliyuncs.com`

```
echo my-bucket:faint:123 > /etc/passwd-ossfs
chmod 640 /etc/passwd-ossfs
mkdir /tmp/ossfs
ossfs my-bucket /tmp/ossfs -ourl=http://oss-cn-hangzhou.aliyuncs.com
```

卸载 Bucket:

```
umount /tmp/ossfs # root user
fusermount -u /tmp/ossfs # non-root user
```

常用设置

- 使用 `ossfs --version` 来查看当前版本，使用 `ossfs -h` 来查看可用的参数
- 如果使用 ossfs 的机器是阿里云 ECS，可以使用内网域名来避免流量收费和 提高速度：

  ```
  ossfs my-bucket /tmp/ossfs -ourl=http://oss-cn-hangzhou-internal.aliyuncs.com
  ```

- 在 linux 系统中，updatedb 会定期地扫描文件系统，如果不想 ossfs 的挂载目录被扫描，可参考 FAQ 设置跳过挂载目录
- 如果你没有使用 eCryptFs 等需要 XATTR 的文件系统，可 以通过添加 `-o noxattr` 参数来提升性能
- ossfs 允许用户指定多组 bucket/access_key_id/access_key_secret 信息。当 有多组信息，写入 passwd-ossfs 的信息格式为：

  ```
  bucket1:access_key_id1:access_key_secret1
  bucket2:access_key_id2:access_key_secret2
  ```

- 生产环境中推荐使用 supervisor 来启动并监控 ossfs 进程，使 用方法见 FAQ

高级设置

- 可以添加 `-f -d` 参数来让 ossfs 运行在前台并输出 debug 日志
- 可以使用 `-o kernel_cache` 参数让 ossfs 能够利用文件系统的 page cache，如果你有多台机器挂载到同一个 bucket，并且要求强一致性，请不要使用此选项

3.2.5 常见问题处理

遇到错误不要慌：) 按如下步骤进行排查：

1. 如果有打印错误信息，尝试阅读并理解它
2. 查看 `/var/log/syslog` 或者 `/var/log/messages` 中有无相关信息

```
grep 's3fs' /var/log/syslog
grep 'ossfs' /var/log/syslog
```

3. 重新挂载 ossfs，打开 debug log：

```
ossfs ... -o dbglevel=debug -f -d > /tmp/fs.log 2>&1
```

然后重复你出错的操作，出错后将 `/tmp/fs.log` 保留，自己查看或者发给我

FAQ

3.2.6 局限性

ossfs 提供的功能和性能和本地文件系统相比，具有一些局限性。具体包括：

- 随机或者追加写文件会导致整个文件的重写。
- 元数据操作，例如 list directory，性能较差，因为需要远程访问 oss 服务器。
- 文件 / 文件夹的 rename 操作不是原子的。
- 多个客户端挂载同一个 oss bucket 时，依赖用户自行协调各个客户端的行为。例如避免多个客户端写同一个文件等等。
- 不支持 hard link。
- 不适合用在高并发读 / 写的场景，这样会让系统的 load 升高

3.2.6 相关链接

- ossfs wiki
- s3fs - 通过 fuse 接口，mount s3 bucket 到本地文件系统。

1 什么是 KVM

KVM 是指基于 Linux 内核的虚拟机（Kernel-based Virtual Machine）。2006 年 10 月，由以色列的 Qumranet 组织开发的一种新的“虚拟机”实现方案。2007 年 2 月发布的 Linux 2.6.20 内核第一次包含了 KVM 。增加 KVM 到 Linux 内核是 Linux 发展的一个重要里程碑，这也是第一个整合到 Linux 主线内核的虚拟化技术。

KVM 在标准的 Linux 内核中增加了虚拟技术，从而我们可以通过优化的内核来使用虚拟技术。在 KVM 模型中，每一个虚拟机都是一个由 Linux 调度程序管理的标准进程，你可以在用户空间启动客户机操作系统。一个普通的 Linux 进程有两种运行模式：内核和用户。KVM 增加了第三种模式：客户模式（有自己的内核和用户模式）。

一个典型的 KVM 安装包括以下部件：

> - 一个管理虚拟硬件的设备驱动，这个驱动通过一个字符设备 /dev/kvm 导出它的功能。通过 /dev/kvm 每一个客户机拥有其自身的地址空间，这个地址空间与内核的地址空间相分离或与任何一个正运行着的客户机相分离。
> - 一个模拟硬件的用户空间部件，它是一个稍微改动过的 QEMU 进程。从客户机操作系统执行 I/O 会拥有 QEMU。QEMU 是一个平台虚拟化方案，它允许整个 PC 环境（包括磁盘、显示卡（图形卡）、网络设备）的虚拟化。任何客户机操作系统所发出的 I/O 请求都被拦截，并被路由到用户模式用以被 QEMU 过程模拟仿真。

2 安装 KVM

2.1 系统要求

KVM 需要有 CPU 的支持 (Intel VT 或 AMD SVM)，在安装 KVM 之前检查一下 CPU 是否提供了虚拟技术的支持

- 基于 `Intel` 处理器的系统，运行 `grep vmx /proc/cpuinfo` 查找 CPU flags 是否包括 `vmx` 关键词
- 基于 `AMD` 处理器的系统，运行 `grep svm /proc/cpuinfo` 查找 CPU flags 是否包括 `svm` 关键词
- 检查 BIOS，确保 BIOS 里开启 `VT` 选项

注：

- 一些厂商禁止了机器 BIOS 中的 VT 选项 , 这种方式下 VT 不能被重新打开
- /proc/cpuinfo 仅从 Linux 2.6.15(Intel) 和 Linux 2.6.16(AMD) 开始显示虚拟化方面的信息。请使用 uname -r 命令查询您的内核版本。如有疑问，请联系硬件厂商

```
egrep "(vmx|svm)" /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_
cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm dts tpr_shadow v
nmi flexpriority ept vpid
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_
cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm dts tpr_shadow v
nmi flexpriority ept vpid
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_
cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm dts tpr_shadow v
nmi flexpriority ept vpid
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_
cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm dts tpr_shadow v
nmi flexpriority ept vpid
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_
cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm dts tpr_shadow v
nmi flexpriority ept vpid
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_
cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm dts tpr_shadow v
nmi flexpriority ept vpid
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_
cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm dts tpr_shadow v
nmi flexpriority ept vpid
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_
cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm dts tpr_shadow v
nmi flexpriority ept vpid
```

2.2 安装 KVM 软件

安装 KVM 模块、管理工具和 libvirt （一个创建虚拟机的工具）

```
yum install -y qemu-kvm libvirt virt-install virt-manager bridge-utils
/etc/init.d/libvirtd start
chkconfig libvirtd on
```

2.2.1 确保正确加载 KVM 模块

```
lsmod  | grep kvm
kvm_intel              54285  0
kvm                   333172  1 kvm_intel
```

2.2.2 检查 KVM 是否正确安装

```
virsh -c qemu:///system list
 Id    Name                           State
----------------------------------------------------
```

如果这里是错误信息，说明安装出现问题

2.3 配置网络

KVM 上网有两种配置，一种是 default，它支持主机和虚拟机的互访，同时也支持虚拟机访问互联网，但不支持外界访问虚拟机，另外一种是 bridge 方式，可以使虚拟机成为网络中具有独立 IP 的主机。

2.3.1 默认网络 virbro

默认的网络连接是 virbr0，它的配置文件在 /var/lib/libvirt/network 目录下，默认配置为

```
cat /var/lib/libvirt/network/default.xml

  default
  77094b31-b7eb-46ca-930e-e0be9715a5ce
```

2.3.2 桥接网络

配置桥接网卡，配置如下

```
more /etc/sysconfig/network-scripts/ifcfg-\*
:::::::::::::: 新建文件
/etc/sysconfig/network-scripts/ifcfg-br0
::::::::::::::
DEVICE=br0
ONBOOT=yes
TYPE=Bridge
BOOTPROTO=static
IPADDR=192.168.39.20
NETMASK=255.255.255.0
GATEWAY=192.168.39.1
DNS1=8.8.8.8
:::::::::::::: 物理网卡
/etc/sysconfig/network-scripts/ifcfg-em1
::::::::::::::
DEVICE=em1
TYPE=Ethernet
ONBOOT=yes
BOOTPROTO=static
BRIDGE=br0
::::::::::::::
```

2.4 配置 VNC

(1) 修改 **VNC** 服务端的配置文件

```
[root@LINUX ~]# vim /etc/libvirt/qemu.conf
vnc_listen = "0.0.0.0"   第十二行，把 vnc_listen 前面的#号去掉。
```

(2) 重启 **libvirtd** 和 **messagebus** 服务

```
[root@LINUX ~]# /etc/init.d/libvirtd restart
Stopping libvirtd daemon:                                  [  OK  ]
Starting libvirtd daemon: libvirtd: initialization failed  [FAILED]
解决办法：
[root@LINUX ~]# echo "export LC_ALL=en_US.UTF-8"  >>  /etc/profile
[root@LINUX ~]# source /etc/profile
[root@LINUX ~]# /etc/init.d/libvirtd restart
[root@LINUX ~]# /etc/init.d/messagebus restart
```

3 创建虚拟机

virt-manager 是基于 libvirt 的图像化虚拟机管理软件，操作类似 vmware，不做详细介绍。

- (1)Virt-manager 图形化模式安装
- (2)Virt-install 命令模式安装【本文使用此方式】
- (3)Virsh XML 模板安装

3.1 上传 ISO

```
[root@LINUX ~]# mkdir -p /home/iso
[root@LINUX ~]# mkdir -p /home/kvm
将 iso 拷贝到 /home/iso 目录
```

3.2 创建 KVM 虚拟机的磁盘文件

本例创建的磁盘文件为 10G，实际使用中应注意下 /home 的空间，可以设置为 100G

```
[root@LINUX ~]# cd /home/kvm/
[root@LINUX ~]# qemu-img create -f qcow2 -o preallocation=metadata kvm_mode.img 10G
```

3.3 启动虚拟机

3.3.1 启动虚拟机参数说明

virt-install 命令有许多选项，这些选项大体可分为下面几大类，同时对每类中的常用选项也做出简单说明。

一般选项：指定虚拟机的名称、内存大小、VCPU 个数及特性等；

> - -n NAME, --name=NAME：虚拟机名称，需全局惟一；
> - -r MEMORY, --ram=MEMORY：虚拟机内在大小，单位为 MB；
> - --vcpus=VCPUS[,maxvcpus=MAX][,sockets=#][,cores=#][,threads=#]：VCPU 个数及相关配置；
> - --cpu=CPU：CPU 模式及特性，如 coreduo 等；可以使用 qemu-kvm -cpu ? 来获取支持的 CPU 模式；

安装方法：指定安装方法、GuestOS 类型等；

- -c CDROM, --cdrom=CDROM：光盘安装介质；
- -l LOCATION, --location=LOCATION：安装源 URL，支持 FTP、HTTP 及 NFS 等，如 ftp://172.16.0.1/pub；
- --pxe：基于 PXE 完成安装；
- --livecd: 把光盘当作 LiveCD；
- --os-type=DISTRO_TYPE：操作系统类型，如 linux、unix 或 windows 等；
- --os-variant=DISTRO_VARIANT：某类型操作系统的变体，如 rhel5、fedora8 等；
- -x EXTRA, --extra-args=EXTRA：根据 --location 指定的方式安装 GuestOS 时，用于传递给内核的额外选项，例如指定 kickstart 文件的位置，--extra-args "ks=http://172.16.0.1/class.cfg"
- --boot=BOOTOPTS：指定安装过程完成后的配置选项，如指定引导设备次序、使用指定的而非安装的 kernel/initrd 来引导系统启动等 ；例如：
- --boot cdrom,hd,network：指定引导次序；
- --boot kernel=KERNEL,initrd=INITRD,kernel_args="console=/dev/ttyS0"：指定启动系统的内核及 initrd 文件；

存储配置：指定存储类型、位置及属性等；

- --disk=DISKOPTS：指定存储设备及其属性；格式为 --disk /some/storage/path,opt1=val1，opt2=val2 等；常用的选项有：
  - device：设备类型，如 cdrom、disk 或 floppy 等，默认为 disk；
  - bus：磁盘总结类型，其值可以为 ide、scsi、usb、virtio 或 xen；
  - perms：访问权限，如 rw、ro 或 sh（共享的可读写），默认为 rw；
  - size：新建磁盘映像的大小，单位为 GB；
  - cache：缓存模型，其值有 none、writethrouth（缓存读）及 writeback（缓存读写）；
  - format：磁盘映像格式，如 raw、qcow2、vmdk 等；
  - sparse：磁盘映像使用稀疏格式，即不立即分配指定大小的空间；
- --nodisks：不使用本地磁盘，在 LiveCD 模式中常用；

网络配置：指定网络接口的网络类型及接口属性如 MAC 地址、驱动模式等；

- -w NETWORK, --network=NETWORK,opt1=val1,opt2=val2：将虚拟机连入宿主机的网络中，其中 NETWORK 可以为：
  - bridge=BRIDGE：连接至名为"BRIDEG"的桥设备；
  - network=NAME：连接至名为"NAME"的网络；
  - 其它常用的选项还有：
    - model：GuestOS 中看到的网络设备型号，如 e1000、rtl8139 或 virtio 等；
    - mac：固定的 MAC 地址；省略此选项时将使用随机地址，但无论何种方式，对于 KVM 来说，其前三段必须为 52:54:00；
- --nonetworks：虚拟机不使用网络功能；

图形配置：定义虚拟机显示功能相关的配置，如 VNC 相关配置；

- --graphics TYPE,opt1=val1,opt2=val2：指定图形显示相关的配置，此选项不会配置任何显示硬件（如显卡），而是仅指定虚拟机启动后对其进行访问的接口；
  - TYPE：指定显示类型，可以为 vnc、sdl、spice 或 none 等，默认为 vnc；
  - port：TYPE 为 vnc 或 spice 时其监听的端口；
  - listen：TYPE 为 vnc 或 spice 时所监听的 IP 地址，默认为 127.0.0.1，可以通过修改 /etc/libvirt/qemu.conf 定义新的默认值；
  - password：TYPE 为 vnc 或 spice 时，为远程访问监听的服务进指定认证密码；
- --noautoconsole：禁止自动连接至虚拟机的控制台；

设备选项：指定文本控制台、声音设备、串行接口、并行接口、显示接口等；

- --serial=CHAROPTS：附加一个串行设备至当前虚拟机，根据设备类型的不同，可以使用不同的选项，格式为“--serial type,opt1=val1,opt2=val2,...”，例如：
- --serial pty：创建伪终端；
- --serial dev,path=HOSTPATH：附加主机设备至此虚拟机；
- --video=VIDEO：指定显卡设备模型，可用取值为 cirrus、vga、qxl 或 vmvga；

3.3.2 bridge 网络模式启动虚拟机

有独立 IP 时使用这种方式

```
[root@LINUX ~]# chmod -R 777 /etc/libvirt
[root@LINUX ~]# chmod -R 777 /home/kvm
[root@LINUX ~]#virt-install --name=kvm_test --ram 4096 --vcpus=4 \
       -f /home/kvm/kvm_mode.img --cdrom /home/iso/sucunOs_anydisk.iso \
       --graphics vnc,listen=0.0.0.0,port=7788, --network bridge=br0 \
       --force --autostart
```

3.3.3 nat 模式启动虚拟机

没有独立 IP 时使用这种方式

```
[root@LINUX ~]# chmod -R 777 /etc/libvirt
[root@LINUX ~]# chmod -R 777 /home/kvm
[root@LINUX ~]#virt-install --name=kvm_test --ram 4096 --vcpus=4 \
       -f /home/kvm/kvm_mode.img --cdrom /home/iso/sucunOs_anydisk.iso \
       --graphics vnc,listen=0.0.0.0,port=7788,--network network=default \
       --force --autostart
```

3.4 连接虚拟机

> - (1) 网上下载 VNC 客户端
> - (2) 用 VNC 客户端连接并安装虚拟机的操作系统（VNC 连上之后，跟安装 Linux CentOS 6.5 系统一样，重新装一次）

```
点击 continue 是如果出现闪退的情况，请修改 Option->Expert->ColorLevel 的值为 full
```

4 管理 KVM

4.1 管理 KVM 上的虚拟机

- virsh list #显示本地活动虚拟机
- virsh list --all #显示本地所有的虚拟机（活动的 + 不活动的）
- virsh start x #启动名字为 x 的非活动虚拟机
- virsh shutdown x #正常关闭虚拟机
- virsh dominfo x #显示虚拟机的基本信息
- virsh autostart x #将 x 虚拟机设置为自动启动

Docker

1 CentOS7 安装 Docker

1.1 准备

CentOS7 x86-64

查看版本

```
#uname -r
3.10.0-123.el7.x86_64
```

1.2 安装 Docker

1.2.1 本地源安装

CentOS 7.3 离线安装 Docker-ce(1703)

```
[root@meetbill ~]#curl -o docker_install.tar.gz https://raw.githubusercontent.com/meet
bill/op_practice_code/master/cloud/docker/docker_install.tar.gz
[root@meetbill ~]#tar -zxvf docker_install.tar.gz
[root@meetbill ~]#cd docker_install
[root@meetbill ~]#sh install.sh
[root@meetbill ~]#systemctl start docker
```

1.2.2 网络源安装

添加 **Docker** 版本仓库

```
cat >/etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg

[docker-ce-stable]
name=Docker CE Stable - $basearch
baseurl=https://download.docker.com/linux/centos/7/$basearch/stable
enabled=1
gpgcheck=1
gpgkey=https://download.docker.com/linux/centos/gpg
EOF
```

安装 **Docker**

docker 在 17 年 3 月份后，Docker 分成了企业版（EE）和社区版（CE），转向基于时间的 YY.MM 形式的版本控制方案，17.03 相当于 1.13.1 版本

```
#yum install docker-ce
```

安装旧版本 (1.12) 方法 `yum install docker-engine`

设置 **Docker** 开机自启动

```
#systemctl enable docker.service
```

启动 **Docker daemon**

```
#systemctl start docker
```

验证 **Docker** 安装是否成功

```
#docker run --rm hello-world
-------------------------------------------------- 以下是程序输出
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
   1. The Docker client contacted the Docker daemon.
   2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   3. The Docker daemon created a new container from that image which runs the
      executable that produces the output you are currently reading.
   4. The Docker daemon streamed that output to the Docker client, which sent it to yo
ur terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

创建 **Docker** 组

将 host 下的普通用户添加到 docker 组中后，可以不使用 sudo 即可执行 docker 程序（只是减少了每次使用 sudo 时输入密码的过程罢了，其实 docker 本身还是以 sudo 的权限在运行的。)

```
sudo usermod -aG docker your_username
```

其他配置

设置 ipv4 转发 (CentOS 上需要配置），实践中发现 Ubuntu 和 SUSE 上无需配置

查看

```
[root@meetbill ~]#sysctl net.ipv4.ip_forward
```

临时更改

```
[root@meetbill ~]#sysctl -w net.ipv4.ip_forward=1
```

永久更改

```
[root@meetbill ~]#echo "net.ipv4.ip_forward=1" >> /etc/sysctl.conf
[root@meetbill ~]#sysctl -p
[root@meetbill ~]#sysctl net.ipv4.ip_forward
```

1.3 卸载 Docker

1.3.1 列出安装的 Docker

```
yum list installed | grep docker
```

1.3.2 删除安装包

```
sudo yum -y remove docker-engine.x86_64
```

1.3.3 删除数据文件

```
rm -rf /var/lib/docker
```

2 Docker 基础

2.1 Docker 三大核心概念

- 镜像 Image 镜像就是一个只读的模板。比如，一个镜像可以包含一个完整的 CentOS 系统，并且安装了 zabbix 镜像可以用来创建 Docker 容器。 其他人制作好镜像，我们可以拿过来轻松的使用。这就是吸引我的特性。
- 容器 Container Docker 用容器来运行应用。容器是从镜像创建出来的实例（好有面向对象的感觉，类和对象），它可以被启动、开始、停止和删除。
- 仓库 Repository 个好理解了，就是放镜像的文件的场所。比如最大的公开仓库是 Docker Hub。

2.2 Docker 镜像使用

当运行容器时，使用的镜像如果在本地中不存在，docker 就会自动从 docker 镜像仓库中下载，默认是从 Docker Hub 公共镜像源下载。 下面我们来学习：

> - 管理和使用本地 Docker 主机镜像
> - 拖取公共镜像源中的镜像
> - 创建镜像

2.2.1 Docker tag

docker tag : 标记本地镜像，将其归入某一仓库。

语法

```
docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

实例

将镜像 Ubuntu:15.10 标记为 runoob/ubuntu:v3 镜像。

```
root@runoob:~# docker tag ubuntu:15.10 runoob/ubuntu:v3
root@runoob:~# docker images   runoob/ubuntu:v3
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
runoob/ubuntu       v3                  4e3b13c8a266        3 months ago        136.3
MB
```

2.2.2 导入导出镜像

导出 #docker save -o zabbix.tar meetbill/zabbix

导入 #docker load -i zabbix.tar

```
注意：导出镜像时使用 imagesid 导出后，如下，导入镜像时 REPOSITORY 和 TAG 会为 <none>（我个人认为是一个 imagesid 可对应多组 REPOSITORY 和 TAG 的原因）
#docker save -o zabbix.tar imagesid
```

2.3 Docker 网络

Docker 的网络模式大致可以分成四种类型，在安装完 Docker 之后，宿主机上会创建三个网络，分别是 bridge 网络，host 网络，none 网络，可以使用 docker network ls 命令查看。

bridge 方式（默认）、none 方式、host 方式、container 复用方式

1、Bridge 方式： --network=bridge

容器与 Host 网络是连通的： eth0 实际上是 veth pair 的一端，另一端（vethb689485）连在 docker0 网桥上 通过 Iptables 实现容器内访问外部网络

2、None 方式： --network=none

这样创建出来的容器完全没有网络，将网络创建的责任完全交给用户。可以实现更加灵活复杂的网络。 另外这种容器可以可以通过 link 容器实现通信。

3、Host 方式： --network=host

容器和主机公用网络资源，使用宿主机的 IP 和端口 这种方式是不安全的。如果在隔离良好的环境中（比如租户的虚拟机中）使用这种方式，问题不大。

4、Container 复用方式： --network=container:name or id

新创建的容器和已经存在的一个容器共享一个 IP 网络资源

2.4 私有仓库

2.4.1 环境准备

ip

```
role    ip
docker 仓库机    192.168.1.52
docker 客户机    192.168.1.136
```

2.4.2 搭建

(1) 搭建仓库 **registry**

```
docker pull regsity
```

基于私有仓库镜像运行容器

```
> docker run -d --name registry --restart always -p 5000:5000 -v  /data/registry:/var/
lib/registry registry
```

(2) 访问私有仓库

```
>curl -X GET http://192.168.1.52:5000/v2/_catalog
{"repositories":[]}   #私有仓库为空，没有提交新镜像到仓库中
```

(3) 为基础镜像打个标签

根据 images 建立 tag,xxxxxxx 为某镜像 id 或 name

docker tag xxxxxxx 192.168.1.52:5000/zabbix

(4) 改 **Docker** 配置文件制定私有仓库 **url**

> echo '{ "insecure-registries":["192.168.1.52:5000"] }' > /etc/docker/daemon.json
> systemctl restart docker

(5) 提交镜像到本地私有仓库中

docker push 192.168.1.52:5000/zabbix

(6) 查看私有仓库是否存在对应的镜像

root@localhost ~

> curl -X GET http://192.168.1.52:5000/v2/_catalog {"repositories":["zabbix"]} curl -X GET
> http://192.168.1.52:5000/v2/zabbix/tags/list {"name":"zabbix","tags":["latest"]}

2.4.3 在 docker 客户机验证

(1) 修改 **Docker** 配置文件

```
echo '{ "insecure-registries":["192.168.1.52:5000"] }' > /etc/docker/daemon.json
systemctl restart docker
```

(2) 从私有仓库中下载已有的镜像

```
docker pull 192.168.1.52:5000/centos
```

至此，私有仓库已 OK

3 Dockerfile 最佳实践

1、挑选合适的基础镜像

基础镜像尽量选最小的镜像

如果是对系统没有过深入学习的可使用比较成熟的基础镜像，如 `Ubuntu`，`CentOS` 等，因为基础镜像只需要下载一次即可共享，并不会造成太多的存储空间浪费。它的好处是这些镜像的生态比较完整，方便我们调试

2、优化 **apt-get/yum** 相关操作

将多个安装软件操作合并成一个，安装完成后使用 clean 清理一下

3、动静分离

经常变化的内容和基本不会变化的内容要分开，把不怎么变化的内容放在下层，创建出来不同基础镜像供上层使用。比如可以创建各种语言的基础镜像，
`python2.7`、`python3.5`、`go1.7`、`java7` 等等，这些镜像包含了最基本的语言库，每个组可以在上面继续构建应用级别的镜像。

4、最小原则：只安装必需的东西

很多人构建镜像时，会将可能用到的东西都打包到镜像中。必须要遏制这种想法，镜像中应该只包含必需的东西，任何可以有也可以没有的东西就不需要放在里面了。因为镜像的扩展很容易，而且运行容器的时候也很方便地对其进行修改。这样可以保证镜像尽可能的小，构建的时候尽可能的快，也保证未来的更快传输、更省网络资源。

5、使用更少的层

虽然看起来把不同的命令尽量分开来，写在多个命令中容易阅读和理解。但是这样会导致出现太多的镜像层，从而不好管理和分析镜像，而且镜像的层是有限的。尽量把内容相关的内容放到同一个层，使用换行符进行分割，这样可以进一步减小镜像大小，并且方便查看镜像历史。

6、减少每层的内容

尽管只安装必须的内容，在这个过程中也可能会产生额外的内容或者临时文件，我们要尽量让每层安装的东西保持最小。

```
- 比如使用 `--no-install-recommends` 参数告诉 `apt-get` 不要安装推荐的软件包
- 安装完软件包，清除 `/var/lib/apt/list/` 缓存
- 删除中间文件：比如下载的压缩包，或者是只用了一次的软件包
- 删除临时文件：如果命令产生了临时文件，也要及时删除
```

7、不要在 `Dockerfile` 中修改文件的权限

因为 `docker` 镜像是分层的，任何修改都会新增一个层，修改文件或者目录权限也是如此。
如果修改大文件或者目录的权限，会把这些文件复制一份，这样很容易导致镜像很大。
解决方案也很简单，要么在添加到 `Dockerfile` 之前就把文件的权限和用户设置好，要么在容器启动脚本 (**entrypoint**) 中做些修改。

8、合理使用 **ADD** 命令

> - DD 命令和 COPY 命令在很大程度上功能是一样的，但是 COPY 语义更加直接。但是唯一例外的是 ADD 命令自带解压功能，如果需要拷贝并解压一个文件到镜像中，我们可以使用 ADD 命令，除此之外，推荐使用 COPY。
> - 如果是使用 ADD 命令来获取网络资源，是不推荐的。网络资源应该使用 RUN wget 或者 curl 命令来获取。

总之，优先使用 COPY

Docker daemon 日志的位置，根据系统不同各不相同。

- Ubuntu - /var/log/upstart/docker.log
- CentOS - /var/log/daemon.log | grep docker
- Red Hat Enterprise Linux Server - /var/log/messages | grep docker

4 Docker 应用

4.1 MySQL

(1) 拉取镜像 这里我们拉取官方的镜像，标签为 5.6

meetbill@Linux:~$ docker pull mysql:5.6

等待下载完成后，我们就可以在本地镜像列表里查到 REPOSITORY 为 mysql, 标签为 5.6 的镜像。

(2) 使用 **mysql** 镜像

运行容器

```
meetbill@Linux:~$mkdir mysql;cd mysql
meetbill@Linux:~/mysql$ docker run -d \
--restart always \
-p 3306:3306 \
--name mymysql \
-v $PWD/data:/var/lib/mysql \
-e MYSQL_ROOT_PASSWORD=123456  mysql:5.6
```

命令说明：

> - -p 3306:3306：将容器的 3306 端口映射到主机的 3306 端口
> - -v $PWD/data:/var/lib/mysql：将主机当前目录下的 data 目录挂载到容器的 /mysql_data
> - -e MYSQL_ROOT_PASSWORD=123456：初始化 root 用户的密码

5 其他

5.1 CentOS 6.5 上安装 Docker

```
rpm -ivh http://dl.Fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
yum -y install docker-io
// 更新 device-mapper-libs
yum install device-mapper-*
/etc/init.d/docker start
```

常见错误

```
启动 docker 报错，错误 log：
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
WARN[0000] You are running linux kernel version 2.6.32-431.el6.x86_64, which might be
unstable running docker. Please upgrade your kernel to 3.10.0.

docker: relocation error: docker: symbol dm_task_get_info_with_deferred_remove, versio
n Base not defined in file libdevmapper.so.1.02 with link time reference

原因：是因为 libdevmapper 版本太旧，需要 update【yum install device-mapper-*】
```

5.2 Alpine Linux

Alpine Linux 打出的包非常小

Alpine Linux, 一个只有 5M 的 Docker 镜像

6 Docker 常见问题

6.1 Docker 容器故障致无法启动解决实例

docker zabbix-server 启动异常退出后，启动失败，解决的方法如下

查找启动文件

```
root@ubuntu:~#find / -name 'docker-zabbix'
/xxxx/subvolumes/2086357831.../bin/docker-zabbix
/xxxx/subvolumes/080fd911a6.../bin/docker-zabbix
/xxxx/subvolumes/87bb2f9818...-init/bin/docker-zabbix
/xxxx/87bb2f98185649304c505.../bin/docker-zabbix
```

修改配置文件进行调试（多输出一些信息进行判断和调试）

6.2 启动容器失败

提示如下

```
Error response from daemon: driver failed programming external connectivity on endpoin
t zabbix (f76e6128eb80f9b9b2a50bc8642d7d9d25dc491b58fcccadcc700943487960bd):  (iptable
s failed: iptables --wait -t nat -A DOCKER -p tcp -d 0/0 --dport 10080 -j DNAT --to-de
stination 172.17.0.11:80 ! -i docker0: iptables: No chain/target/match by that name.
 (exit status 1))
Error: failed to start containers: zabbix
```

解决方法

```
重启 Docker
#systemctl restart docker
```

6.3 CentOS7 上运行容器挂载卷没有写入权限

在 CentOS7 中运行容器，发现挂载的本地目录在容器中没有执行权限，原因是 CentOS7 中的安全模块 selinux 把权限禁掉了，至少有以下三种方式解决挂载的目录没有权限的问题：

1，在运行容器的时候，给容器加特权：

示例：docker run -i -t --privileged=true -v /home/docs:/src waterchestnut/nodejs:0.12.0

2，临时关闭 selinux：

示例：su -c "setenforce 0"

之后执行：docker run -i -t -v /home/docs:/src waterchestnut/nodejs:0.12.0

注意：之后要记得重新开启 selinux，命令：su -c "setenforce 1"

3，添加 selinux 规则，将要挂载的目录添加到白名单：

示例：chcon -Rt svirt_sandbox_file_t /home/docs

之后执行：docker run -i -t -v /home/docs:/src waterchestnut/nodejs:0.12.0

OpenStack

OpenStack 实践

K8s

Kubernetes 概述

简介

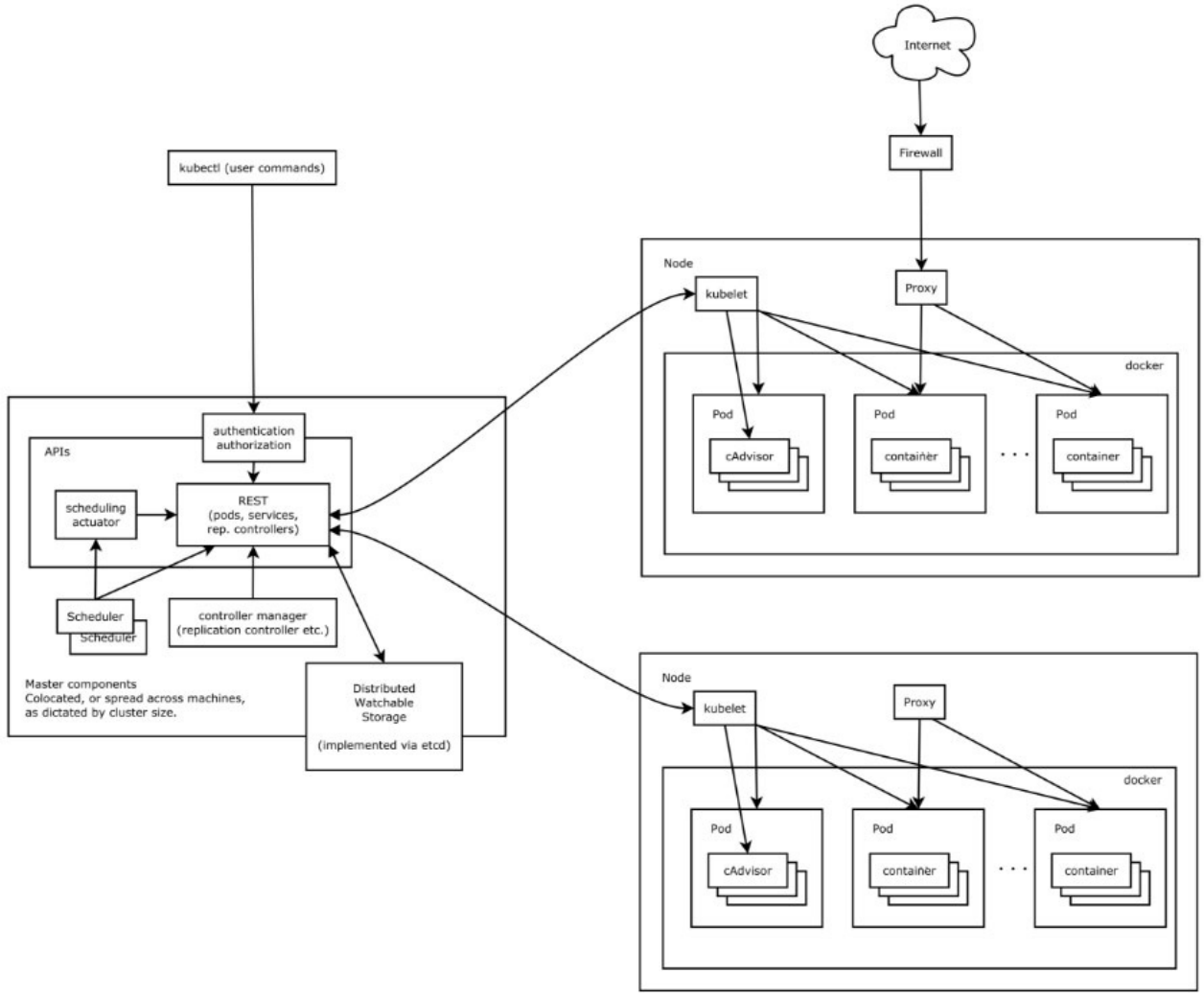
Kubernetes 是一个开源的，用于管理云平台中多个主机上的容器化的应用，Kubernetes 的目标是让部署容器化的应用简单并且高效（powerful）,Kubernetes 提供了应用部署，规划，更新，维护的一种机制。

特性

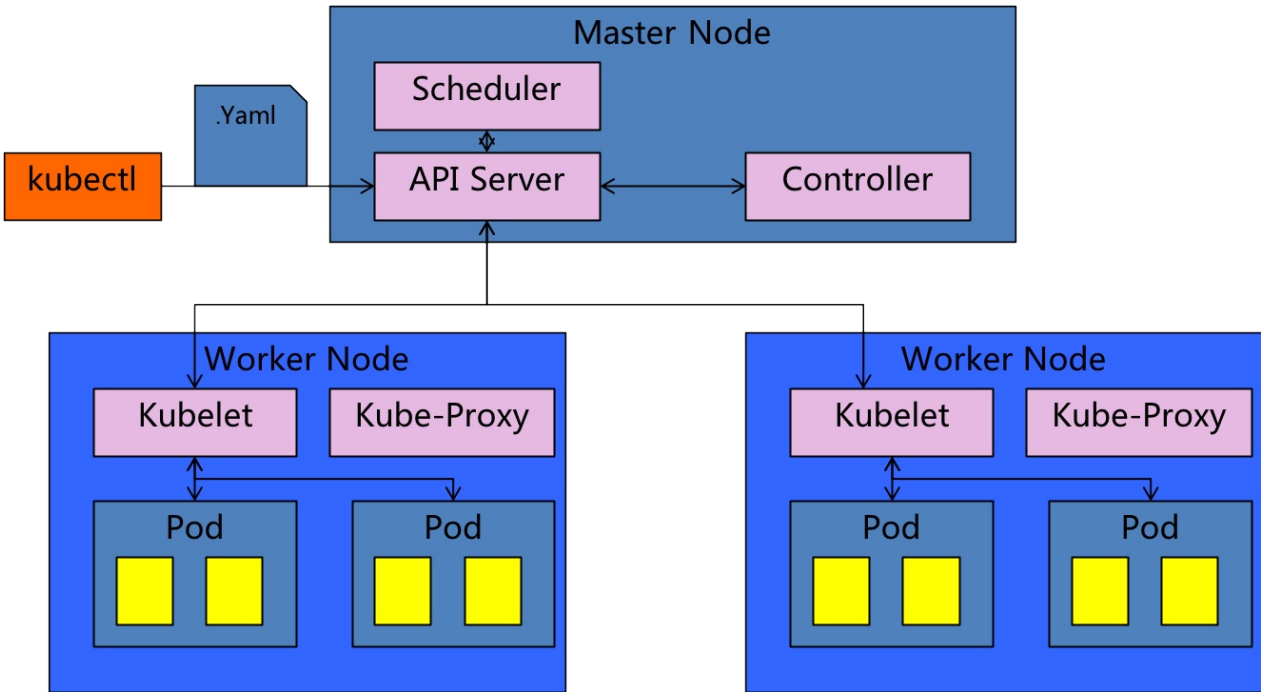
> - 服务发现和负载均衡
> - 自我修复 - 重新启动失败的容器
> - 横向缩放 - 使用简单的命令或 UI，或者根据 CPU 的使用情况自动调整应用程序副本数
> - 自动部署和回滚
> - 密钥和配置管理 - 部署和更新密钥和应用程序配置，不会重新编译镜像，不会暴露
> - 更多

Kubernetes 设计架构

Kubernetes 集群包含有节点代理 kubelet 和 Master 组件 (APIs, scheduler, etc)，一切都基于分布式的存储系统。下面这张图是 Kubernetes 的架构图。



k8s 全景简图



官网体验教程

https://kubernetes.io/docs/tutorials/kubernetes-basics/

官网通过如下 6 部分来实践如何玩转 k8s

> - 创建 k8s 集群
> - 部署应用
> - 内部访问应用
> - 外部访问应用
> - Scale 应用
> - 滚动更新

重要概念

> - Cluster : 计算，存储和网络资源的集合，K8s 利用这些资源运行各种基于容器的应用
>   - 比如在百度 CCE 上首先需要创建个集群（购买 N 台 BCC 实例时，实际后台是创建了 N+3 实例，其中 3 台用于创建 Master 节点）
> - Master : 主要负责调度，决定应用放在哪里运行
> - Node ：Node 的职责是运行容器应用。
>   - Node 由 Master 管理，Node 负责监控并汇报容器的状态
>   - 根据 Master 的要求管理容器的生命周期
> - Pod ：K8s 的最小工作单元。每个 Pod 包含一个或者多个容器。
>   - Pod 中的容器会作为一个整体被 Master 调度到一个 Node 上运行
> - Controller : K8s 通过 Controller 来管理 Pod，Controller 定义了 Pod 的部署特性：
>   - Deployment
>   - ReplicaSet
>   - DaemonSet
>   - StatefuleSet
>   - Job
> - Service：定义了外界访问一组 Pod 的方式
>   - ClusterIP
>   - NodePort
>   - LoadBalancer
> - Namespace
>   - 多个用于或者项目可以使用同一个 cluster ,然后使用不同的 Namespace
>   - --namespace=name名

集群应用篇

- 负载均衡
- LVS
- 高可用的 LVS 负载均衡集群

负载均衡

解决的问题

- 能够将大规模并发访问和数据流量分发到多台内部服务器上，减少用户的等待时间；
- 当有重负载的计算请求时，能够将请求分解成多个任务，并将这些任务分配到内部多个计算服务器上，收集处理内部计算服务器的处理结果，汇总结果并返回给用户；
- 负载均衡能够大大提高系统的处理能力、提高系统灵活性；
- 高可用：当某服务器出现故障时，不影响其它服务器和用户的运行和使用；
- 当后端某个服务器出现故障时，能够将该服务器从服务列表中删除，当服务器恢复时，再将该服务器加入到列表中；
- 可伸缩：能够不影响其它服务器和用户的情况下进行扩容；

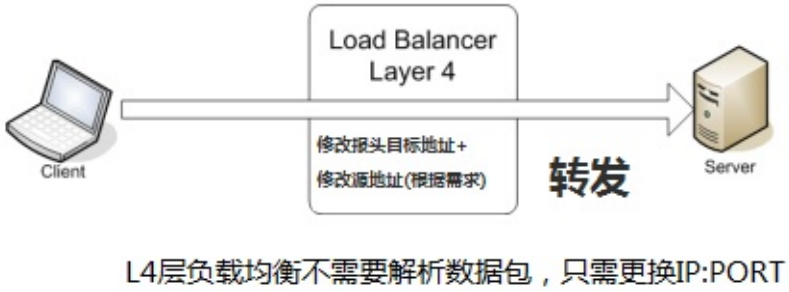
负载均衡的本质是数据包的转发，即如何将数据包转发到负载最小的服务器上去。最常用的硬件方案有 F5，软件方案有 LVS+Keepalived。

网络层次上的负载均衡

- 二层负载均衡，是通过一个虚拟的 MAC 地址接收请求，然后再分配到真实的 MAC 地址；
- 三层负载均衡，是通过一个虚拟的 IP 地址接收请求，然后分配到真实的 IP 地址；
- 四层负载均衡，是通过一个虚拟 IP+ 端口进行接收，然后分配到真实的服务器；
- 七层负载均衡，是通过一个虚拟的主机名或 URL 接收请求，然后分配到真实的服务器。可以根据 URL，浏览器类别，语言等，将请求发给不同的内部服务器。

四层负载均衡

- 首先会配置 frontend 的 IP:PORT 与 backend 的 IP:PORT 映射关系。当有客户端请求到来时，会根据映射关系，将请求转发到 backend 的服务器上去。
- 工作在 L4 层的负载均衡器，不需要对客户端的数据包内容进行解析。如 SYN 包到来时，负载均衡器只需要选择一个最佳的内部服务器，将 SYN 包中的 dst IP:PORT 替换为内部服务器的 IP:PORT，并直接转发给该内部服务器即可。对有些部署，可能还需要修改 source IP:PORT，这样负载均衡器可以收到内部服务器返回的包。

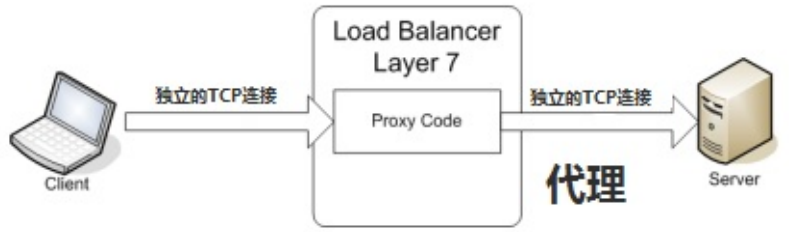


L4层负载均衡不需要解析数据包，只需更换IP:PORT

七层负载均衡

L7 层负载均衡，是应用层的负载均衡。

负载均衡器需要先和客户端建立连接 (TCP 三次握手），接收客户端发过来的报文，然后根据报文特定字段的内容，来选择内部服务器。L7 层负载均衡器，是一个代理服务器，需要与客户端和内部服务器间都建立连接。 一般来说，L7 层负载均衡的处理能力，低于 L4 层。



优点：

- 能够更好的拓展内部网络。如：能够将使用英语的和使用汉语的客户端请求，发送到不同的内部服务器。
- 能够将对图片的请求，发送到图片服务器，同时图片服务器可以加缓存。
- 能够提前过滤掉一些非法的请求和无用的数据包，而不用将这些请求发送到内部服务器，减轻内部服务器的压力。

缺点：

- 速度上，不如 L4 层快

负载均衡算法

- 轮循（Round Robin）：将每次请求，轮流的分配给内部服务器。当内部服务器的软硬件配置相当时，比较适合。
- 权重轮循（Weighted Round Robin）：根据内部服务器的配置不同，给每台服务器一个权重。如服务器 A 的权值被设计成 1，B 的权值是 3，C 的权值是 2，客户请求依次发给 [ABBBCC]。权重大的，分配到的任务就多。
- 随机（Random）：将请求随机分配给内部中的多个服务器。

- 权重随机（Weighted Random）：此种均衡算法类似于权重轮循算法，不过在处理请求分担时是个随机选择的过程。
- 响应速度（Response Time）：负载均衡器与内部服务器建立连接，并定时向内部服务器发 ping 包（或者其他包也行）。根据各服务器的响应速度，决定将用户请求发给哪台内部服务器。该均衡算法能够较好的反应内部服务器运行状况。
- 连接数（Connections）：有些内部服务器可能直接与客户端建立连接。这时可以询问内部服务器当前的用户连接数，并将新的用户请求发送给连接数最少的内部服务器。比较适合长连接。
- 处理能力（Processing Capacity）：将内部服务器的 CPU/ 内存 /IO/ 当前负载，根据一定的算法换成负载值，并定时上报给负载均衡器。负载均衡器每次将用户请求转发给当前 Load 值最低的内部服务器。
- DNS 轮询：DNS 也可以用来做负载均衡。网络上，客户端一般通过域名来找到服务器的 IP 地址，DNS 服务器在接收客户端查询时，按顺序将服务器的 IP 地址返回给客户端，来达到均衡的目的。比较适合全局负载均衡。
- Hash：将访问用户的 IP 地址进行 hash，根据 hash 的结果来决定将该用户定向到哪台后端服务器。

LVS

LVS 简介

LVS(Linux Virtual Server)，早已加入到 Linux 内核中。 LVS 使用的是 IP 负载均衡技术 (ipvs 模块实现）。LVS 安装在 DirectorServer(DS) 上，DS 根据配置信息虚拟出一个 ip(VIP)，同时根据配置信息，生成路由表，将用户的数据包按照一定的法则转发到后端 RealServer(RS) 上。

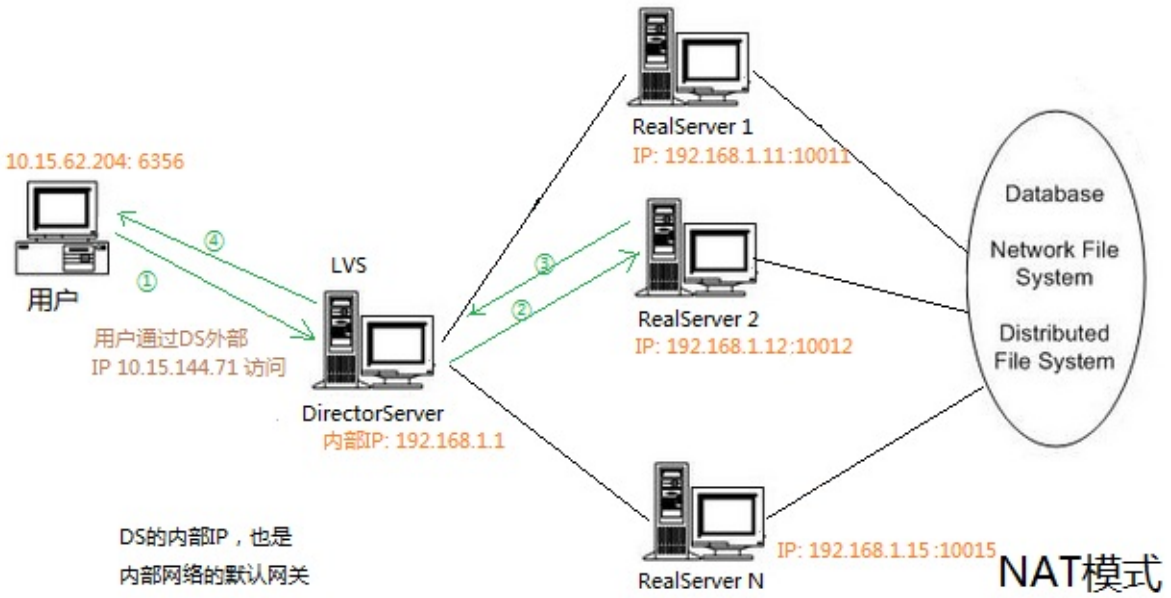
LVS 进行 L4 层转发，且在内核中，速度极快。与 Keepalived 相比，缺少对内部服务器的健康检查，且存在单点故障。 LVS 使用 ipvsadm 来配置 ipvs。

数据包的转发，有三种方法：NAT/TUN/DR。

模式	网络要求	是否需要 **VIP**	端口映射	**DS** 参与回包	**ARP** 隔离	效率
NAT	同一网段	不需要	支持	是	不需要	最慢
TUN	可在同一物理网络或不同物理网络	需要	-	否	-	中等
DR	同一物理网络	需要	不支持	否	需要	最高

数据包三种转发方式

NAT 网络地址翻译技术

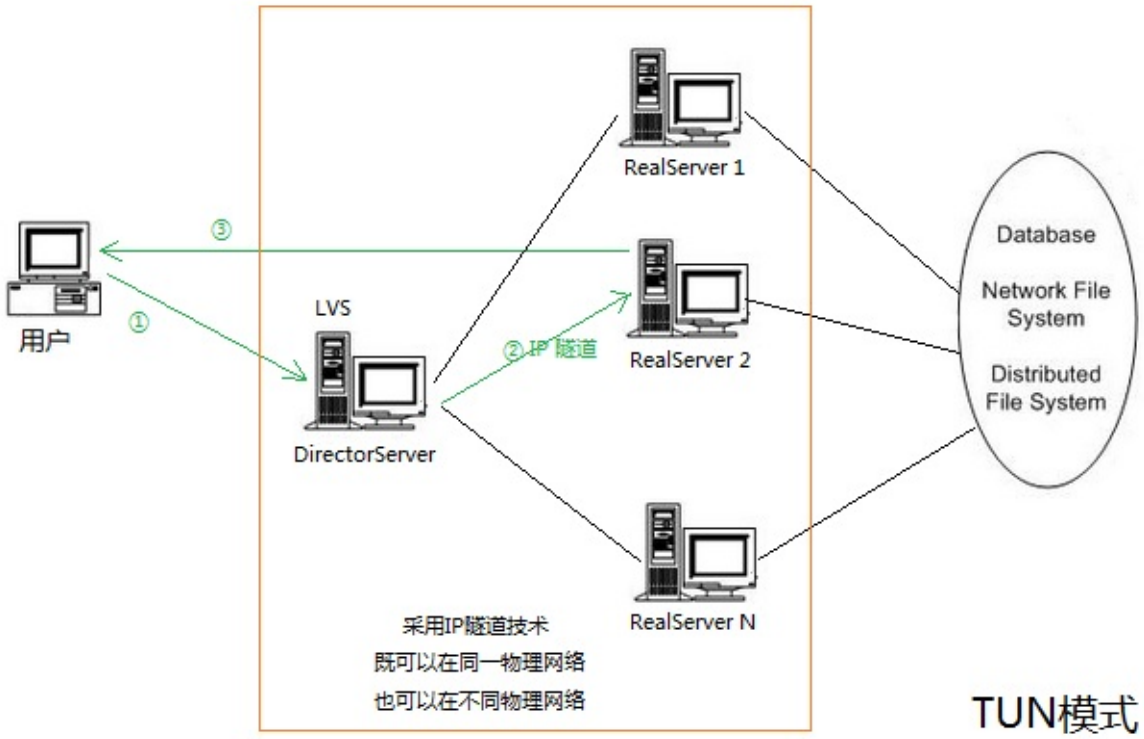


DS 和 RS，都在同一个网段，才能进行 NAT 模式转发。同时需要将 DS 的内部 IP 设置为内部网络的默认网关，RS 在回包时，直接发给内部网关（即 DS），由内部网关进行转发。 用户通过 DS 的外部 IP 地址进行访问。 NAT 模式下，是可以进行端口映射的。

整个数据包流程如下：

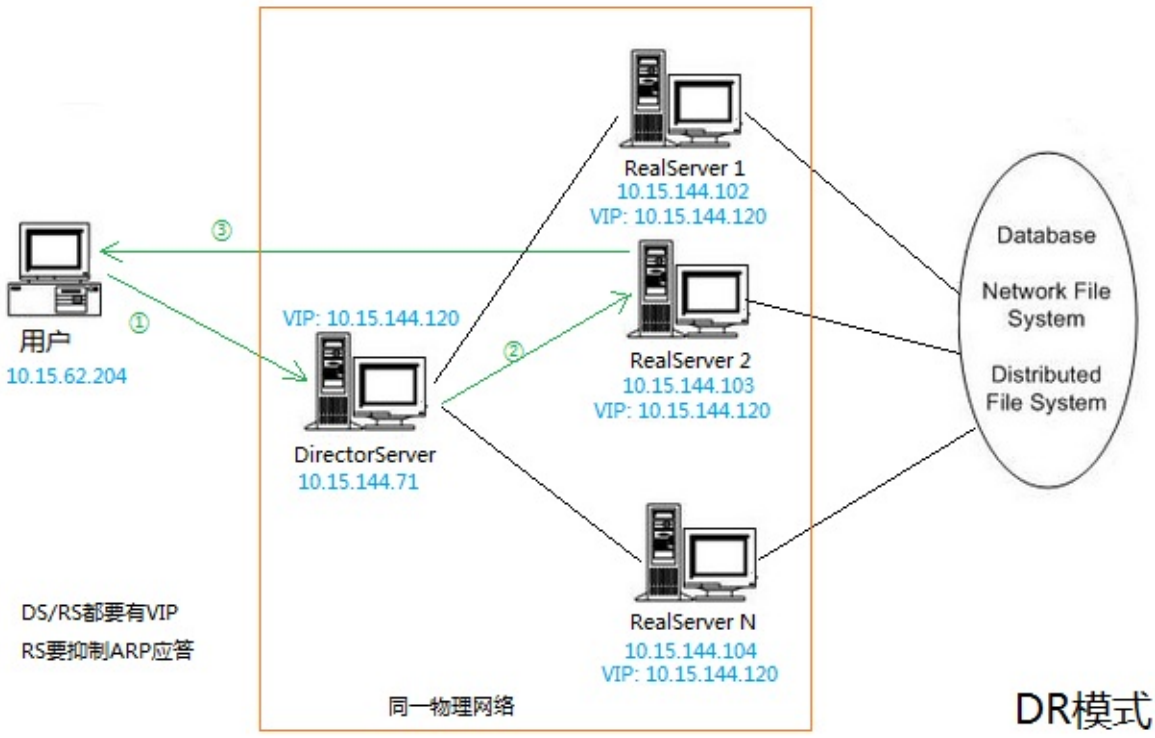
```
假设用户 IP 为 10.15.62.204，使用端口 6356；
用户 ->DS: src(10.15.62.204:6356) dst(10.15.144.71:80)
DS 的外部 IP 收到数据包后，内核进行转发
DS->RS2: src(10.15.62.204:6356) dst(192.168.1.12:10012)
RS2 收到数据包，处理完毕后，将回包通过内部网关发出去
RS2->DS: src(192.168.1.12:10012) dst(10.15.62.204:6356)
DS 收到会包后，由内核转发给用户
DS->用户：src(10.15.144.71:80) dst(10.15.62.204:6356)
```

TUN IP 隧道技术



调度器采用 IP 隧道技术，将用户的请求转发到 RS，RS 直接将响应发给用户。 TUN 模式下，RS 的回包，不需要经过 DS，而是直接发给客户端。

DR 直接转发



DS 通过改写请求报文的 MAC 地址，将请求发给 RS，RS 直接将响应发给用户。 **DR** 方式的效率最高，但要求 DS 和 RS 在同一物理网络。 DR 模式不支持端口映射；同时 RS 需要抑制关于 VIP 的 ARP 应答。

整个数据包处理过程：

```
假设客户端 IP 10.15.62.204，使用端口 6356
用户 ->DS: src(10.15.62.204:63565) dst(10.15.144.120:80)
DS 收到包后，通过负载均衡算法，选择 RS2，改写包的目的 MAC 地址，将数据包发给 RS2
DS->RS2: src(10.15.62.204:63565) dst(10.15.144.120:80)，目的 MAC 发生改变
RS2 处理完毕后，将回包直接发给客户端
RS2->用户：src(10.15.144.120:80) dst(10.15.62.204:63565)
```

从转发效率来讲，NAT 最差；TUN 多了 ip 隧道的处理，次之；DR 效率最高。

配置脚本

以 DR 模式为例

```
# 安装 LVS 机器（即 DirectorServer）脚本，lvs-DR.sh
# 设置 VIP，并设置转发规则

#!/bin/sh

VIP=10.15.144.120
RIP1=10.15.144.102
RIP2=10.15.144.103
RIP3=10.15.144.104

. /etc/rc.d/init.d/functions
case "$1" in
    start)
        echo " start LVS of Director Server"
        /sbin/ifconfig eth0:1 $VIP broadcast $VIP netmask 255.255.255.255 up # 添加虚拟
设备 eth0:1 和虚拟 IP
        /sbin/route add -host $VIP dev eth0:1
        echo "1" >/proc/sys/net/ipv4/ip_forward # 允许转发

        /sbin/ipvsadm -C   #Clear IPVS table
        #set LVS
        /sbin/ipvsadm -A -t $VIP:10087 -s wrr -p 60
        /sbin/ipvsadm -a -t $VIP:10087 -r $RIP1 -g -w 2 # -g 为 DR 模式， -w 为权重
        /sbin/ipvsadm -a -t $VIP:10087 -r $RIP2 -g -w 1
        /sbin/ipvsadm -a -t $VIP:10087 -r $RIP3 -g -w 1

        /sbin/ipvsadm  # 打印 ipvs 信息
    ;;
    stop)
        echo "close LVS Directorserver"
        echo "0" >/proc/sys/net/ipv4/ip_forward
        /sbin/ipvsadm -C
        /sbin/ifconfig eth0:1 down
    ;;
```

```
    *)
        echo "Usage: $0 {start|stop}"
        exit 1
esac

#------------------------------------------------------------------------
# 在 3 台 RealServer 上，配置 VIP，关闭对该 VIP 的 ARP 应答，所执行的脚本：rs-DR.sh
#!/bin/bash

VIP=10.15.144.120

. /etc/rc.d/init.d/functions
case "$1" in
    start)
        echo " Start LVS  of  Real Server"
        /sbin/ifconfig lo:0 $VIP netmask 255.255.255.255 broadcast $VIP up
        /sbin/route add -host $VIP dev lo:0
        # 忽略收到的 arp 广播
        echo "1" >/proc/sys/net/ipv4/conf/lo/arp_ignore
        # 封装数据包时，忽略源 ip(lvs 服务器 ip)，而是将 VIP 做为源 ip
        echo "2" >/proc/sys/net/ipv4/conf/lo/arp_announce
        echo "1" >/proc/sys/net/ipv4/conf/all/arp_ignore
        echo "2" >/proc/sys/net/ipv4/conf/all/arp_announce
        sysctl -p > /dev/null 2>&1
        ;;
    stop)
        /sbin/ifconfig lo:0 down
        echo "close LVS Director server"
        route del $VIP > /dev/null 2>&1
        echo "0" >/proc/sys/net/ipv4/conf/lo/arp_ignore
        echo "0" >/proc/sys/net/ipv4/conf/lo/arp_announce
        echo "0" >/proc/sys/net/ipv4/conf/all/arp_ignore
        echo "0" >/proc/sys/net/ipv4/conf/all/arp_announce
        ;;
    *)
        echo "Usage: $0 {start|stop}"
        exit 1
esac
```

在 DS 和 RS 上运行相应的脚本后，LVS 负载均衡系统就搭建完毕了。

ipvsadm

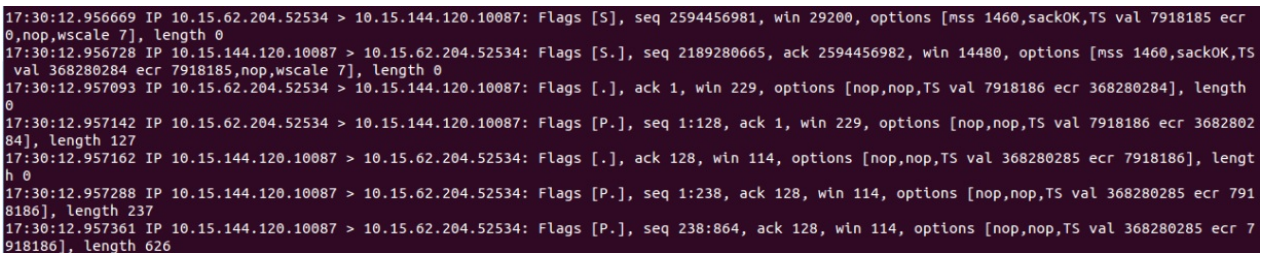
利用 ipvs 管理工具 ipvsadm，查看 DS 内部情况

```
[root@10.15.144.71 lvs]# ipvsadm
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port Forward Weight ActiveConn InActConn
TCP 10.15.144.120:10087 wrr persistent 2
  -> 10.15.144.102:10087 Route 2 0 4
  -> 10.15.144.103:10087 Route 1 0 0
  -> 10.15.144.104:10087 Route 1 0 0

# 在 10.15.62.204 上进行测试，常用命令有：
ll@ll-rw:~$ wget http://10.15.144.120:10087/index.html
ll@ll-rw:~$ ab -n 100 -c 10 http://10.15.144.120:10087/index.html

[root@10.15.144.71 lvs]# ipvsadm -L -c
IPVS connection entries
pro expire state       source               virtual              destination
TCP 00:22 FIN_WAIT 10.15.62.204:50937 10.15.144.120:10087 10.15.144.102:10087
TCP 00:04 FIN_WAIT 10.15.62.204:50930 10.15.144.120:10087 10.15.144.102:10087
TCP 01:50 FIN_WAIT 10.15.62.204:50959 10.15.144.120:10087 10.15.144.103:10087
TCP 00:50 NONE 10.15.62.204:0 10.15.144.120:10087 10.15.144.103:10087
TCP 00:22 NONE 10.15.62.204:0 10.15.144.120:65535 10.15.144.102:65535
TCP 00:06 FIN_WAIT 10.15.62.204:50931 10.15.144.120:10087 10.15.144.102:10087

# 关掉 RealServer 服务器中的服务，再次在 10.15.62.204 测试时，收到的错误信息
ll@ll-rw:~$ wget http://10.15.144.120:10087/index.html
--2014-11-18 16:47:40-- http://10.15.144.120:10087/index.html
Connecting to 10.15.144.120:10087... failed: No route to host.
```

当 DR 模式时，数据包是直接从 RS 返回到客户端的，所以在 RS 上也需要虚拟出设备和 IP（lo:0 10.15.14.120）。RS 直接利用该 IP 进行返回。 同时，在同一子网内，有多个 Server 都拥有 10.15.14.120 这个 IP。当其它机器进行 ARP 查询时 (who has ip 10.15.1.120)，只能够由 DS 进行响应，其他 Server 不能够响应。这也是 RS 需要使用 echo "0" >/proc/sys/net/ipv4/conf/lo/arp_ignore 的原因。

tcpdump 抓包分析

使用 tcpdump 在 DS 上抓包，可以看到从 User 来的数据包，直接转发给了 RS；RS 收到数据包后，也是直接回复给了 User，不需要再经过 DS 转发。下图是在 RS(10.15.14.102) 上抓包的截图：



问题分析

LVS 存在单点故障。当 LVS 服务器挂掉了，整个系统就完了。 LVS 不检测内部服务器的状态。当内部服务器挂掉时，仍然将请求发往该服务器。

LVS+Keepalived解决方案：

- 实现主备模式解决单点故障。
- 内部服务器有问题时，将其从可用服务器列表中删除；当其恢复时，再将其加入到可用服务器列表。

Keepalived 使用

1 Keepalived 介绍及安装

1.1 介绍

Keepalived 是一个基于 VRRP 协议来实现的 WEB 服务高可用方案，其功能类似于 [heartbeat], 可以利用其来避免单点故障。一个 WEB 服务至少会有 2 台服务器运行 Keepalived，一台为主服务器（MASTER），一台为备份服务器（BACKUP），但是对外表现为一个虚拟 IP，主服务器会发送特定的消息给备份服务器，当备份服务器收不到这个消息的时候，即主服务器宕机的时候，备份服务器就会接管虚拟 IP，继续提供服务，从而保证了高可用性。

```
     +---------VIP(192.168.0.3)----------+
    |                                    |
    |                                    |
server(MASTER) <----keepalived----> server(BACKUP)
(192.168.0.1)                       (192.168.0.2)
```

1.1.1 LVS 和 Keepalived 的关系

LVS 可以不依赖 Keepalived 而进行分发请求，但是想让负载调度器动态监控眞实服务器心跳需要写很复杂的代码。而 Keepalived 正是一个通过简单配置就能满足请求分发、心跳检测、集群管理的好工具

1.2 安装

编译安装：

```
$ wget http://www.keepalived.org/software/keepalived-1.2.2.tar.gz</a>
$ tar -zxvf keepalived-1.2.2.tar.gz
$ cd keepalived-1.2.2
$ ./configure --prefix=/usr/local/keepalived
$ make && make install
```

拷贝需要的文件：

```
$ cp /usr/local/keepalived/etc/rc.d/init.d/keepalived /etc/init.d/keepalived
$ cp /usr/local/keepalived/sbin/keepalived /usr/sbin/
$ cp /usr/local/keepalived/etc/sysconfig/keepalived /etc/sysconfig/
$ mkdir -p /etc/keepalived/
$ cp /usr/local/etc/keepalived/keepalived.conf /etc/keepalived/keepalived.conf
```

`/etc/keepalived/keepalived.conf` 是默认配置文件

1.3 使用

```
$ /etc/init.d/keepalived start | restart | stop
```

当启动了 Keepalived 之后，通过 `ifconfig` 是看不到 VIP 的，但是通过 `ip a` 命令是可以看到的。 当 MASTER 宕机，BACKUP 升级为 MASTER，这些 VRRP_Instance 状态的切换都可以在 `/var/log/message` 中进行记录。

2 Keepalived 配置相关

Keepalived 只有一个配置文件 /etc/keepalived/keepalived.conf，里面主要包括以下几个配置区域，分别是 global_defs、static_ipaddress、static_routes、vrrp_script、vrrp_instance 和 virtual_server。

2.1 global defs 区域

主要是配置故障发生时的通知对象以及机器标识

```
global_de_s {
    notification_email {
        a@abc.com
        b@abc.com
        ...
    }
    notification_email_from alert@abc.com
    smtp_server smtp.abc.com
    smtp_connect_timeout 30
    enable_traps
    router_id host163
}
```

- notification_email 故障发生时给谁发邮件通知。
- notification_email_from 通知邮件从哪个地址发出。
- smpt_server 通知邮件的 smtp 地址。
- smtp_connect_timeout 连接 smtp 服务器的超时时间。
- enable_traps 开启 SNMP 陷阱（[Simple Network Management Protocol][snmp]）。
- router_id 标识本节点的字条串，通常为 hostname，但不一定非得是 hostname。故障发生时，邮件通知会用到。

2.2 vrrp script 区域

用来做健康检查的，当时检查失败时会将 `vrrp_instance` 的 `priority` 减少相应的值。

```
vrrp_script chk_http_port {
    script "</dev/tcp/127.0.0.1/80"
    interval 1
    weight -10
}
```

以上意思是如果 `script` 中的指令执行失败，那么相应的 `vrrp_instance` 的优先级会减少 10 个点。

2.3 VRRPD 配置

在 [VRRP] 协议中，有两组重要的概念：

> - VRRP 路由器和虚拟路由器
> - 主控路由器和备份路由器

VRRP 路由器是指运行 VRRP 的路由器，是物理实体，虚拟路由器是指 VRRP 协议创建的，是逻辑概念。一组 VRRP 路由器协同工作，共同构成一台虚拟路由器。该虚拟路由器对外表现为一个具有唯一固定 IP 地址和 MAC 地址的逻辑路由器。处于同一个 VRRP 组中的路由器具有两种互斥的角色：

主控路由器和备份路由器，一个 VRRP 组中有且只有一台处于主控角色的路由器，可以有一个或者多个处于备份角色的路由器。VRRP 协议使用选择策略从路由器组中选出一台作为主控，负责 ARP 相应和转发 IP 数据包，组中的其它路由器作为备份的角色处于待命状态。当由于某种原因主控路由器发生故障时，备份路由器能在几秒钟的时延后升级为主路由器。由于此切换非常迅速而且不用改变 IP 地址和 MAC 地址，故对终端使用者系统是透明的。

VRRPD 配置包括两部分

> - VRRP 同步组 (synchroization group)
> - VRRP 实例 (VRRP Instance)

2.3.1 VRRP Sync Groups

vrrp_rsync_group 用来定义 vrrp_intance 组，使得这个组内成员动作一致。举个例子来说明一下其功能：

两个 vrrp_instance 同属于一个 vrrp_rsync_group，那么其中一个 vrrp_instance 发生故障切换时，另一个 vrrp_instance 也会跟着切换（即使这个 instance 没有发生故障）。

eg: 机器有两个网段，一个内网一个外网，每个网段开启一个 VRRP 实例，假设 VRRP 配置为检查内网，那么当外网出现问题时，VRRPD 认为自己仍然健康，那么不会发送 Master 和 Backup 的切换，从而导致了问题。Sync group 就是为了解决这个问题。可以将两个实例都放到一个 Sync group，这样，group 里面任何一个实例出现问题都会发生切换

```
vrrp_sync_group VG_1 {
    group {
        inside_network   # name of vrrp_instance (below)
        outside_network  # One for each moveable IP.
        ...
    }
    notify_master /path/to_master.sh
    notify_backup /path/to_backup.sh
    notify_fault "/path/fault.sh VG_1"
    notify /path/notify.sh
    smtp_alert
}
```

- notify_master/backup/fault 分别表示切换为主 / 备 / 出错时所执行的脚本。
- notify 表示任何一状态切换时都会调用该脚本，并且该脚本在以上三个脚本执行完成之后进行调用，Keepalived 会自动传递三个参数（$1 = "GROUP"|"INSTANCE"，$2 = name of group or instance，$3 = target state of transition(MASTER/BACKUP/FAULT)）。
- smtp_alert 表示是否开启邮件通知（用全局区域的邮件设置来发通知）。

2.3.2 VRRP 实例配置

vrrp_instance 用来定义对外提供服务的 VIP 区域及其相关属性。

```
vrrp_instance VI_1 {
    state MASTER
    interface eth0
    use_vmac <VMAC_INTERFACE>
    dont_track_primary
    track_interface {
        eth0
        eth1
    }
    mcast_src_ip <IPADDR>
    lvs_sync_daemon_interface eth1
    garp_master_delay 10
    virtual_router_id 1
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 12345678
    }
    virtual_ipaddress {
        10.210.214.253/24 brd 10.210.214.255 dev eth0
        192.168.1.11/24 brd 192.168.1.255 dev eth1
    }

    virtual_routes {
        172.16.0.0/12 via 10.210.214.1
        192.168.1.0/24 via 192.168.1.1 dev eth1
        default via 202.102.152.1
    }

    track_script {
        chk_http_port
    }

    nopreempt
    preempt_delay 300
    debug
    notify_master <STRING>|<QUOTED-STRING>
    notify_backup <STRING>|<QUOTED-STRING>
    notify_fault <STRING>|<QUOTED-STRING>
    notify <STRING>|<QUOTED-STRING>
    smtp_alert
}
```

- state 可以是 MASTER 或 BACKUP，不过当其他节点 Keepalived 启动时会将 priority 比较大的节点选举为 MASTER，因此该项其实没有实质用途。
- interface 节点固有 IP（非 VIP）的网卡，用来发 VRRP 包。
- use_vmac 是否使用 VRRP 的虚拟 MAC 地址。

- dont_track_primary 忽略 VRRP 网卡错误。（默认未设置）
- track_interface 监控以下网卡，如果任何一个不通就会切换到 FALT 状态。（可选项）
- mcast_src_ip 修改 vrrp 组播包的源地址，默认源地址为 master 的 IP。（由于是组播，因此即使修改了源地址，该 master 还是能收到回应的）
- lvs_sync_daemon_interface 绑定 lvs syncd 的网卡。
- garp_master_delay 当切为主状态后多久更新 ARP 缓存，默认 5 秒。
- virtual_router_id 取值在 0-255 之间，用来区分多个 instance 的 VRRP 组播。

注意： 同一网段中 virtual_router_id 的值不能重复，否则会出错，相关错误信息如下。

```
Keepalived_vrrp[27120]: ip address associated with VRID not present in received packet
 :
one or more VIP associated with VRID mismatch actual MASTER advert
bogus VRRP packet received on eth1 !!!
receive an invalid ip number count associated with VRID!
VRRP_Instance(xxx) ignoring received advertisment...
```

可以用这条命令来查看该网络中所存在的 vrid： `tcpdump -nn -i any net 224.0.0.0/8`

- priority 用来选举 master 的，要成为 master，那么这个选项的值 [最好高于其他机器 50 个点][priority_more_than_50]，该项 [取值范围][priority] 是 1-255（在此范围之外会被识别成默认值 100）。
- advert_int 发 VRRP 包的时间间隔，即多久进行一次 master 选举（可以认为是健康查检时间间隔）。
- authentication 认证区域，认证类型有 PASS 和 HA（IPSEC），推荐使用 PASS（密码只识别前 8 位）。
- virtual_ipaddress VIP，不解释了。
- virtual_routes 虚拟路由，当 IP 漂过来之后需要添加的路由信息。
- virtual_ipaddress_excluded 发送的 VRRP 包里不包含的 IP 地址，为减少回应 VRRP 包的个数。在网卡上绑定的 IP 地址比较多的时候用。
- nopreempt 允许一个 priority 比较低的节点作为 master，即使有 priority 更高的节点启动。

首先 nopreemt 必须在 state 为 BACKUP 的节点上才生效（因为是 BACKUP 节点决定是否来成为 MASTER 的），其次要实现类似于关闭 auto failback 的功能需要将所有节点的 state 都设置为 BACKUP，或者将 master 节点的 priority 设置的比 BACKUP 低。我个人推荐使用将所有节点的 state 都设置成 BACKUP 并且都加上 nopreempt 选项，这样就完成了关于

autofailback 功能，当想手动将某节点切换为 MASTER 时只需去掉该节点的 nopreempt 选项并且将 priority 改的比其他节点大，然后重新加载配置文件即可（等 MASTER 切过来之后再将配置文件改回去再 reload 一下）。

当使用 `track_script` 时可以不用加 `nopreempt` ，只需要加上 `preempt_delay 5` ，这里的间隔时间要大于 `vrrp_script` 中定义的时长。

- preempt_delay master 启动多久之后进行接管资源（VIP/Route 信息等），并提是没有 `nopreempt` 选项。

2.4 LVS 配置

virtual_server_group 一般在超大型的 LVS 中用到，一般 LVS 用不过这东西，因此不多说。

```
virtual_server IP Port {
    delay_loop <INT>
    lb_algo rr|wrr|lc|wlc|lblc|sh|dh
    lb_kind NAT|DR|TUN
    persistence_timeout <INT>
    persistence_granularity <NETMASK>
    protocol TCP
    ha_suspend
    virtualhost <STRING>
    alpha
    omega
    quorum <INT>
    hysteresis <INT>
    quorum_up <STRING>|<QUOTED-STRING>
    quorum_down <STRING>|<QUOTED-STRING>
    sorry_server <IPADDR> <PORT>
    real_server <IPADDR> <PORT> {
        weight <INT>
        inhibit_on_failure
        notify_up <STRING>|<QUOTED-STRING>
        notify_down <STRING>|<QUOTED-STRING>
        # HTTP_GET|SSL_GET|TCP_CHECK|SMTP_CHECK|MISC_CHECK
        HTTP_GET|SSL_GET {
            url {
                path <STRING>
                # Digest computed with genhash
                digest <STRING>
                status_code <INT>
            }
            connect_port <PORT>
            connect_timeout <INT>
            nb_get_retry <INT>
            delay_before_retry <INT>
        }
    }
}
```

- delay_loop 延迟轮询时间（单位秒）。
- lb_algo 后端调试算法（load balancing algorithm）。
- lb_kind LVS 调度类型 [NAT][nat]/[DR][dr]/[TUN][tun]。
- virtualhost 用来给 HTTP_GET 和 SSL_GET 配置请求 header 的。
- sorry_server 当所有 real server 宕掉时，sorry server 顶替。
- real_server 真正提供服务的服务器。
- weight 权重。

- notify_up/down 当 real server 宕掉或启动时执行的脚本。
- 健康检查的方式，N 多种方式。
- path 请求 real serserver 上的路径。
- digest/status_code 分别表示用 genhash 算出的结果和 http 状态码。
- connect_port 健康检查，如果端口通则认为服务器正常。
- connect_timeout,nb_get_retry,delay_before_retry 分别表示超时时长、重试次数，下次重试的时间延迟。

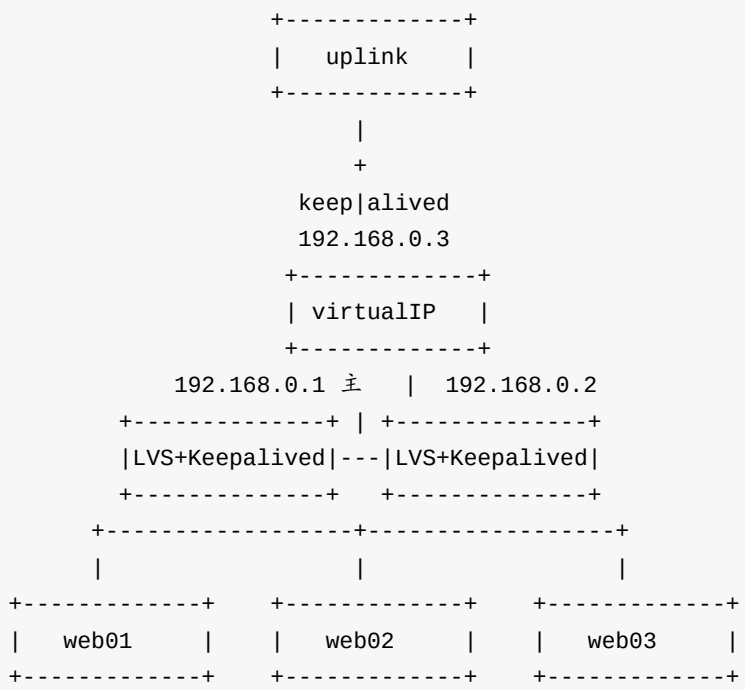
其他选项暂时不作说明。

3 Keepalived 工作原理

Keepalived 是以 VRRP 协议为实现基础的，VRRP 全称 Virtual Router Redundancy Protocol，即虚拟路由冗余协议。

虚拟路由冗余协议，可以认为是实现路由器高可用的协议，即将 N 台提供相同功能的路由器组成一个路由器组，这个组里面有一个 master 和多个 backup，master 上面有一个对外提供服务的 VIP（该路由器所在局域网内其他机器的默认路由为该 VIP），master 会发组播，当 backup 收不到 vrrp 包时就认为 master 宕掉了，这时就需要根据 VRRP 的优先级***vrrp_priority***来选举一个 backup 当 master***select_master***。这样的话就可以保证路由器的高可用了。

Keepalived 主要有三个模块，分别是 core、check 和 vrrp。core 模块为 Keepalived 的核心，负责主进程的启动、维护以及全局配置文件的加载和解析。check 负责健康检查，包括常见的各种检查方式。vrrp 模块是来实现 VRRP 协议的。



3.1 VRRP 工作流程

(1). 初始化

路由器启动时，如果路由器的优先级是 255（最高优先级，路由器拥有路由器地址）, 要发送 VRRP 通告信息，并发送广播 ARP 信息通告路由器 IP 地址 对应的 MAC 地址为路由虚拟 MAC, 设置通告信息定时器准备定时发送 VRRP 通告信息，转为 MASTER 状态；否则进入 BACKUP 状态，设置定时器检查 定时检查是否收到 MASTER 的通告信息。

(2).Master

```
设置定时通告定时器；

用 VRRP 虚拟 MAC 地址响应路由器 IP 地址的 ARP 请求；

转发目的 MAC 是 VRRP 虚拟 MAC 的数据包；

如果是虚拟路由器 IP 的拥有者，将接受目的地址是虚拟路由器 IP 的数据包，否则丢弃；

当收到 shutdown 的事件时删除定时通告定时器，发送优先权级为 0 的通告包，转初始化状态；

如果定时通告定时器超时时，发送 VRRP 通告信息；

收到 VRRP 通告信息时，如果优先权为 0, 发送 VRRP 通告信息；否则判断数据的优先级是否高于本机，或相等而且实际 IP 地址大于本地实际 IP, 设置定时通告定时器，复位主机超时定时器，转 BACKUP 状态；否则的话，丢弃该通告包；
```

(3).Backup

```
设置主机超时定时器；

不能响应针对虚拟路由器 IP 的 ARP 请求信息；

丢弃所有目的 MAC 地址是虚拟路由器 MAC 地址的数据包；

不接受目的是虚拟路由器 IP 的所有数据包；

当收到 shutdown 的事件时删除主机超时定时器，转初始化状态；

主机超时定时器超时的时候，发送 VRRP 通告信息，广播 ARP 地址信息，转 MASTER 状态；

收到 VRRP 通告信息时，如果优先权为 0, 表示进入 MASTER 选举；否则判断数据的优先级是否高于本机，如果高的话承认 MASTER 有效，复位主机超时定时器；否则的话，丢弃该通告包；
```

3.2 MASTER 和 BACKUP 节点的优先级如何调整？

首先，每个节点有一个初始优先级，由配置文件中的 priority 配置项指定，MASTER 节点的 priority 应比 BAKCUP 高。 运行过程中 Keepalived 根据 vrrp_script 的 weight 设定，增加或减小节点优先级。规则如下：

1. 当 weight > 0 时，vrrp_script script 脚本执行返回 0（成功）时优先级为 priority + weight, 否则为 priority. 当 BACKUP 发现自己的优先级大于 MASTER 通告的优先级时，进行主从切换。
2. 当 weight < 0 时，vrrp_script script 脚本执行返回非 0（失败）时优先级为 priority + weight, 否则为 priority. 当 BACKUP 发现自己的优先级大于 MASTER 通告的优先级时，进行主从切换。
3. 当两个节点的优先级相同时，以节点发送 VRRP 通告的 IP 作为比较对象，IP 较大者为 MASTER.

以上文中的配置为例：

```
HOST1: 192.168.0.1, priority=91, MASTER(default)
HOST2: 192.168.0.2, priority=90, BACKUP
VIP: 192.168.0.3 weight = 2
```

抓包命令：tcpdump -nn vrrp

3.3 ARP 查询处理

当内部主机通过 ARP 查询虚拟路由器 IP 地址对应的 MAC 地址时，MASTER 路由器回复的 MAC 地址为虚拟的 VRRP 的 MAC 地址，而不是实际网卡的 MAC 地址，这样在路由器切换时让内网机器觉察不到；而在路由器重新启动时，不能主动发送本机网卡的实际 MAC 地址。如果虚拟路由器开启的 ARP 代理 (proxy_arp) 功能，代理的 ARP 回应也回应 VRRP 虚拟 MAC 地址；

3.4 虚拟 IP 地址和 MAC 地址

VRRP 组（备份组）中的虚拟路由器对外表现为唯一的虚拟 MAC 地址，地址格式为 00-00-5E-00-01-[VRID], VRID 为 VRRP 组的编号，范围是 0~255.

3.5 Keepalived 进程

Keepalived 主要有三个模块，分别是 core、check 和 vrrp。

> - core 模块为 Keepalived 的核心，负责主进程的启动、维护以及全局配置文件的加载和解析。
> - check 负责健康检查，包括常见的各种检查方式。
> - vrrp 模块是来实现 VRRP 协议的。

3.6 Keepalived 健康检查方式

Keepalived 对后端 realserver 的健康检查方式主要有以下几种：

TCP_CHECK

工作在第 4 层，Keepalived 向后端服务器发起一个 tcp 连接请求，如果后端服务器没有响应或者超时，那么这个后端将从服务器中移除

HTTP_GET

工作在第 5 层，通过向指定的 URL 执行 http 请求，将得到的结果比对（经检验此种方法在多个实体服务器只能检测到第一个，故不可行）

SSL_GET

与 HTTP_GET 类似 ***MISC_CHECK***

用脚本来检测，脚本如果带有参数，需要将脚本和参数放入到双引号内。脚本的返回值需要为：

> - 0---------- 检测成功
> - 1---------- 检测失败，将从服务器池中移除
> - 2~255-------- 检测成功；如果有设置 misc_dynamic，权重自动调整为退出码 -2，如果退出码为 200，权重自动调整为 198=200-2

4 Keepalived 场景应用

4.1 Keepalived 主从切换

主从切换比较让人蛋疼，需要将 backup 配置文件的 priority 选项的值调整的比 master 高 50 个点，然后 reload 配置文件就可以切换了。当时你也可以将 master 的 Keepalived 停止，这样也可以进行主从切换。

4.2 Keepalived 仅做 HA 时的配置

请看该文档同级目录下的配置文件示例。

用 tcpdump 命令来捕获的结果如下：

```
17:20:07.919419 IP 192.168.1.1 > 224.0.0.18: VRRPv2, Advertisement, vrid 1, prio 200,
authtype simple, intvl 1s, length 20
```

5 其他配置

5.1 重定向 Keepalived 输出日志

(1) 修改 /etc/sysconfig/keepalived

把 KEEPALIVED_OPTIONS="-D" 修改为 KEEPALIVED_OPTIONS="-D -d -S 0"

其中 -S 指定 syslog 的 facility"

同时创建 /var/log/keepalived 目录

```
#mkdir /var/log/keepalived
```

(2) 在 /etc/rsyslog.conf 中添加

```
# keepalived -S 0
local0.*                    /var/log/keepalived/keepalived.log
```

(3) 重启 Rsyslog 和 Keepalived 服务

```
#/etc/init.d/rsyslog restart
#/etc/init.d/Keepalived restart
```

5.2 只用 VRRP 模块

假如不使用 LVS 的话，即无需加载 ip_vs 模块（注；不装 ipvsadm 的话，直接启动 Keepalived 的话，会因为没有 ip_vs 模块而一直在日志中输出错误日志）

修改 /etc/sysconfig/keepalived

把 KEEPALIVED_OPTIONS="-D" 修改为 KEEPALIVED_OPTIONS="-D -P"

6 常见问题

6.1 virtual_router_id 冲突

Keepalived 日志提示

```
[Time] [Hostname] Keepalived_vrrp: ip address associated with VRID not present in rece
ived packet : [VIP]
[Time] [Hostname] Keepalived_vrrp: one or more VIP associated with VRID mismatch actua
l MASTER advert
[Time] [Hostname] Keepalived_vrrp: bogus VRRP packet received on eth0 !!!
[Time] [Hostname] Keepalived_vrrp: VRRP_Instance(web_1) Dropping received VRRP packet.
..
```

解决方法

```
同一集群的 Keepalived 的主、备机的 virtual_router_id 必须相同，取值 0-255
但是同一内网中不应有相同 virtual_router_id 的集群
修改 virtual_router_id 就可以了
```

6.2 VIP 无法访问

6.2.1 VIP 被抢占

```
arping -I eth0 VIP
```

查看是否输出两个不同的 Mac 地址，如果是则地址被占用

6.2.2 网关的 ARP 缓存没有刷新

```
arping -I eth0 -c 5 -s VIP GATEWAY
```

ZooKeeper

1 ZooKeeper 安装

ZooKeeper 的安装模式分为三种，分别为：单机模式（stand-alone）、集群模式和集群伪分布模式。ZooKeeper 单机模式的安装相对比较简单，如果第一次接触 ZooKeeper 的话，建议安装 ZooKeeper 单机模式或者集群伪分布模式。

http://hadoop.apache.org/zookeeper/releases.html

2.1 单机模式

```
[meetbill@meetbill_dev01 /opt]$ tar xvfz  zookeeper-3.4.10.tar.gz
[meetbill@meetbill_dev01 /opt]$ ZOOKEEPER_HOME=/opt/zookeeper-3.4.10
```

配置：

```
# 添加一个 zoo.cfg 配置文件
[meetbill@meetbill_dev01 ~]$ cd $ZOOKEEPER_HOME/conf
[meetbill@meetbill_dev01 conf]$ cp zoo_sample.cfg zoo.cfg

# 修改配置文件（zoo.cfg）
dataDir=/home/hadoop/app/tmp/zk

# 启动 zk
[meetbill@meetbill_dev01 zookeeper]$ bin/zkServer.sh start
```

2.2 集群模式

为了获得可靠的 ZooKeeper 服务，用户应该在一个集群上部署 ZooKeeper 。只要集群上大多数的 ZooKeeper 服务启动了，那么总的 ZooKeeper 服务将是可用的。另外，最好使用奇数台机器。 如果 zookeeper 拥有 5 台机器，那么它就能处理 2 台机器的故障了。

之后的操作和单机模式的安装类似，我们同样需要对 JAVA 环境进行设置，下载最新的 ZooKeeper 稳定版本并配置相应的环境变量。不同之处在于每台机器上 conf/zoo.cfg 配置文件的参数设置，参考下面的配置：

```
tickTime=2000
dataDir=/var/zookeeper/
clientPort=2181
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

server.id=host:port:port. 指示了不同的 ZooKeeper 服务器的自身标识，作为集群的一部分的机器应该知道 ensemble 中的其它机器。用户可以从“ server.id=host:port:port. ”中读取相关的信息。 在服务器的 data （ dataDir 参数所指定的目录）目录下创建一个文件名为 myid 的文件，这个文件中仅含有一行的内容，指定的是自身的 id 值。比如，服务器“ 1 ”应该在 myid 文件中写入“ 1 ”。这个 id 值必须是 ensemble 中唯一的，且大小在 1 到 255 之间。这一行配置中，第一个端口（ port ）是从（ follower ）机器连接到主（ leader ）机器的端口，第二个端口是用来进行 leader 选举的端口。在这个例子中，每台机器使用三个端口，分别是： clientPort ， 2181 ； port ， 2888 ； port ， 3888 。

2 客户端

```
[meetbill@meetbill_dev01 ~]$ zkCli.sh
Connecting to localhost:2181
2018-07-21 01:11:38,350 [myid:] - INFO  [main:Environment@100] - Client
...
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper]
```

3 ZooKeeper 的简单操作

3.1 ls

使用 ls 命令来查看当前 ZooKeeper 中所包含的内容：

```
[zk: 10.77.20.23:2181(CONNECTED) 1] ls /
[zookeeper]
```

3.2 create

创建一个新的 znode ，使用 create /zk myData 。这个命令创建了一个新的 znode 节点“ zk ”以及与它关联的字符串：

```
[zk: 10.77.20.23:2181(CONNECTED) 2] create /zk myData
Created /zk
```

再次使用 ls 命令来查看现在 zookeeper 中所包含的内容：

```
[zk: 10.77.20.23:2181(CONNECTED) 3] ls /
[zk, zookeeper]
```

此时看到， zk 节点已经被创建。

3.3 get

我们运行 get 命令来确认第二步中所创建的 znode 是否包含我们所创建的字符串：

```
[zk: 10.77.20.23:2181(CONNECTED) 4] get /zk
myData
Zxid = 0x40000000c
time = Tue Jan 18 18:48:39 CST 2011
Zxid = 0x40000000c
mtime = Tue Jan 18 18:48:39 CST 2011
pZxid = 0x40000000c
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
```

3.4 set

我们通过 set 命令来对 zk 所关联的字符串进行设置：

```
[zk: 10.77.20.23:2181(CONNECTED) 5] set /zk shenlan211314
cZxid = 0x40000000c
ctime = Tue Jan 18 18:48:39 CST 2011
mZxid = 0x40000000d
mtime = Tue Jan 18 18:52:11 CST 2011
pZxid = 0x40000000c
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 13
numChildren = 0
```

3.5 delete

下面我们将刚才创建的 znode 删除：

```
[zk: 10.77.20.23:2181(CONNECTED) 6] delete /zk
```

最后再次使用 ls 命令查看 ZooKeeper 所包含的内容：

```
[zk: 10.77.20.23:2181(CONNECTED) 7] ls /
[zookeeper]
```

经过验证， zk 节点已经被删除。

其他篇

- windows下服务

windows服务

xp下建设VPN服务器

可以执行以下步骤完成VPN服务器部署

- 1.打开网络连接
- 2.点击网络任务重的"创建一个新的连接",
- 3.[新建连接向导]，下一步
  - [网络连接类型]--选择"设置高级连接"，点击下一步
  - [高级连接选项]--选择"接受传入的连接"，点击下一步
  - [传入的虚拟专用网(VPN)连接]--选择"允许虚拟专用连接(A)",点击下一步
  - [用户权限]--点击"添加"，添加用户后，勾选上此用户，点击下一步
  - [网络软件]--鼠标选中"Internet协议后(TCP/IP)",点击属性
    - 选择制定TCP/IP地址，如从 192.168.199.2 到 192.168.199.10 点击确定
  - 点击完成
- 4 在内网中设置VPN服务器后，如果想让外网的服务器进行连接到此VPN服务器，可以端口转发下1723和1701两个端口