

linux 命令

1. 七层模型介绍

- (1) 物理层：物理层负责最后将信息编码成电流脉冲或其它信号用于网上传输；
- (2) 数据链路层：
- (3) 网络层：网络层负责在源和终点之间建立连接。可以理解为，此处需要确定计算机的位置，怎么确定？IPv4，IPv6！
- (4) 传输层：传输层向高层提供可靠的端到端的网络数据流服务。可以理解为：每一个应用程序都会在网卡注册一个端口号，该层就是端口与端口的通信！
- (5) 会话层：会话层建立、管理和终止表示层与实体之间的通信会话；
- (6) 表示层：表示层提供多种功能用于应用层数据编码和转化，以确保以一个系统应用层发送的信息可以被另一个系统应用层识别。可以理解为：解决不同系统之间的通信，eg：Linux 下的 QQ 和 Windows 下的 QQ 可以通信；
- (7) 应用层：OSI 的应用层协议包括文件的传输、访问及管理协议(FTAM)，以及文件虚拟终端协议(VIP)和公用管理系统信息(CMIP)等。

2. TCP/IP 协议族常用协议

应用层：TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet 等等

传输层：TCP，UDP

网络层：IP，ICMP，OSPF，EIGRP，IGMP

链路层：SLIP，CSLIP，PPP，MTU

TCP 和 UDP 是传输层协议，而 IP 是网络层协议。

3. 负载均衡一般分为两种，一种是基于 DNS，另一种基于 IP 报文。

- 利用 DNS 实现负载均衡，就是在 DNS 服务器配置多个 A 记录，不同的 DNS 请求会解析到不同的 IP 地址。大型网站一般使用 DNS 作为第一级负载均衡。缺点是 DNS 生效时间略长，扩展性差。
- 基于 IP 的负载均衡，早期比较有代表性并且被大量使用的就是 LVS 了。原理是 LVS 在 Linux 内核态获取到 IP 报文后，根据特定的负载均衡算法将 IP 报文转发到整个集群的某台服务器中去。缺点是 LVS 的性能依赖 Linux 内核的网络性能，但 Linux 内核的网络路径过长导致了大量开销，使得 LVS 单机性能较低。

4. Nginx

如图所示：master process 是其主线程，worker process 顾名思义，“我是具体干活的啦”，笔者截图的进程还是比较少的，因为这是自己的服务器，没有太多服务，实际业务中有可能出现几十个 worker process，还是挺壮观的。

```

[root@ideadata ~]# ps -ef|grep nginx
root      392    371    0 20:45 pts/1      00:00:00 grep nginx
root      25789   1    0 Mar28 ?        00:00:00 nginx: master process ./nginx
root      25790 25789   0 Mar28 ?        00:00:13 nginx: worker process

```

Nginx 推荐的配置是，一个 worker process 对应一个 CPU 内核，确保硬件资源的有效利用。

每个 worker process 都是多线程且独立运行，负责获取新连接并进行处理。进程之间通过共享内存进行通信。

5. 负载均衡主要有硬件与软件两种实现方式，主流负载均衡解决方案中，硬件厂商以 F5 为代表，软件主要为 NGINX 与 LVS。
6. F5 是负载均衡的一种硬件解决方案，优点负载能力强，与系统无关，适用于一大堆设备、大访问量、简单应用。但成本高，无法有效掌握服务器及应用状态。

nginx 是一种负载均衡软件，性价比高，但负载能力受服务器性能影响。

7. Keepalived 是一种高可用解决方案，而 nginx 是负载均衡解决方案，nginx+Keepalived 可以实现高可用的负载均衡方案。其他同理。
8. heartbeat 主要是通过心跳检测，来达到一个高可用性的效果。
9. 常见的高可用解决方案：Heartbeat、Keepalived。常见的负载均衡解决方案：Haproxy、nginx、LVS。
10. 在 Linux 世界，进程不能直接访问硬件设备，当进程需要访问硬件设备(比如读取磁盘文件，接收网络数据等等)时，必须由用户态模式切换至内核态模式，通过系统调用访问硬件设备。过多的系统调用，造成频繁的内核/用户切换，造成性能下降。
11. shell 下的 strace 命令可以跟踪到一个进程产生的系统调用，包括参数、返回值、执行消耗的时间。

```

$strace cat /dev/null
execve("/bin/cat", ["cat", "/dev/null"], [/* 22 vars */]) = 0
brk(0)                                = 0xab1000
access("/etc/ld.so.nohwcap", F_OK)    = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0x7f29379a7000
access("/etc/ld.so.preload", R_OK)    = -1 ENOENT (No such file or directory)
...

```

每一行都是一条系统调用，等号左边是系统调用的函数名及其参数，右边是该调用的返回值。strace 显示这些调用的参数并返回符号形式的值。strace 从内核接收信息，而且不需要以任何特殊的方式来构建内核。

参数：

- (1) -f：跟踪由 fork 调用所产生的子进程。
- (2) -F：尝试跟踪 vfork 调用。在 -f 时，vfork 不被跟踪。

```
strace -f -F -o ~/straceout.txt myserver
```

(3) -o filename: 将 strace 的输出写入文件 filename。

(4) -T: 显示每一调用所耗的时间。

(5) -t: 在输出中的每一行前加上时间信息，秒级

(6) -tt: 在输出中的每一行前加上时间信息，微秒级。

(7) -ttt: 精确到微妙，而且时间表示为 unix 时间戳

(8) -e trace=process: 只跟踪有关进程控制的系统调用。

(9) -p pid: 跟踪指定的进程 pid

```
strace -o output.txt -T -tt -e trace=all -p 28979
```

跟踪 28979 进程的所有系统调用（-e trace=all），并统计系统调用的花费时间，以及开始时间（并以可视化的时分秒格式显示），最后将记录结果存在 output.txt 文件里面。

(10) -s: 用于指定 trace 结果的每一行输出的字符串的长度。

12. Netstat 命令用于显示各种网络相关信息，如连接的协议、本地地址、客户端地址、接口状态、应用信息等等等。

(1) -a: 列出所有当前的连接。

(2) -t: 列出 TCP 协议的连接。

(3) -u: 选项列出 UDP 协议的连接。

(4) -n: 禁用反向域名解析，加快查询速度。

(5) -l: 只列出在 listen（监听）中的连接。任何网络服务的后台进程都会打开一个端口，用于监听接入的请求。这些正在监听的套接字也和连接的套接字一样，也能被 netstat 列出来。

(6) -p: 获取进程名、进程号以及用户 ID。使用 -p 选项时，netstat 必须运行在 root 权限之下，不然它就不能得到运行在 root 权限下的进程名，而很多服务包括 http 和 ftp 都运行在 root 权限之下。

(7) -s: 打印出网络统计数据，包括某个协议下的收发包数量。如果想只打印出 TCP 或 UDP 协议的统计数据，只要加上对应的选项（-t 和 -u）即可。

(8) -r: 打印内核路由信息。打印出来的信息与 route 命令输出的信息一样。我们也可以使用 -n 选项禁止域名解析。

(9) -i: 打印网络接口信息，也就是 eth0 等网卡信息。

(10) -c: 每隔一个固定时间，执行 netstat 命令。

(11) -g: 输出 IPv4 和 IPv6 的多播组信息。

```
[root@MyServer ~]# netstat -tup
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45808    TIME_WAIT   -
tcp      0      40 107.150.12.152.static.q:ssh 112.96.109.78:8089     ESTABLISHED 1341/sshd
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45809    TIME_WAIT   -
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45815    TIME_WAIT   -
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45813    TIME_WAIT   -
tcp      0      0 107.150.12.152.static.q:ssh 218.92.1.158:30617     ESTABLISHED 2008/sshd
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45810    TIME_WAIT   -
tcp      0      0 107.150.12.152.static:23222 lax28s10-in-f206.1e10:https ESTABLISHED 1320/python
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45811    TIME_WAIT   -
tcp      0      0 107.150.12.152.static.q:ssh 218.92.0.163:19907     ESTABLISHED 2003/sshd
tcp      0      0 107.150.12.152.static:49355 74.125.20.188:hqvroom  ESTABLISHED 1320/python
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45814    TIME_WAIT   -
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45812    ESTABLISHED 1320/python
tcp      0      0 107.150.12.152.static:58003 112.96.109.78:45807    ESTABLISHED 1320/python
```

13.使用 Netstat 时需要注意接口状态，接口状态分为 linsten、ESTABLISHED 和 TIME_WAIT 等几种状态。

Perf

13.Perf 是用来进行软件性能分析的工具。它不但可以分析指定应用程序的性能问题，也可以用来分析内核的性能问题，当然也可以同时分析应用代码和内核，从而全面理解应用程序中的性能瓶颈。

14.安装 perf 非常简单，只要您有 2.6.31 以上的内核源代码，那么进入 tools/perf 目录然后敲入下面两个命令即可：

```
make
make install
```

15.Perf 是一个包含 22 种子工具的工具集，以下是最常用的 5 种：

- (1) perf-list
- (2) perf-stat
- (3) perf-top
- (4) perf-record
- (5) perf-report

16.Perf-list 用来查看 perf 所支持的性能事件，这些事件可以将它们划分为三类：

- (1) Hardware Event 是由 PMU（即电源管理单元）硬件产生的事件，比如 cache 命中，当您需要了解程序对硬件特性的使用情况时，便需要对这些事件进行采样；
- (2) Software Event 是内核软件产生的事件，比如进程切换，tick 数等；
- (3) Tracepoint event 是内核中的静态 tracepoint 所触发的事件，这些 tracepoint 用来判断程序运行期间内核的行为细节，比如 slab 分配器的分配次数等。

17.有些程序慢是因为计算量太大，其多数时间都应该在使用 CPU 进行计算，这叫做 CPU bound 型；有些程序慢是因为过多的 IO，这种时候其 CPU 利用率应该不高，

这叫做 IO bound 型；对于 CPU bound 程序的调优和 IO bound 的调优是不同的。

18. Perf stat 通过概括精简的方式提供被调试程序运行的整体情况和汇总数据。

```
$ perf stat ./t1
Performance counter stats for './t1':

262.738415 task-clock-msecs # 0.991 CPUs
2 context-switches # 0.000 M/sec
1 CPU-migrations # 0.000 M/sec
81 page-faults # 0.000 M/sec
9478851 cycles # 36.077 M/sec (scaled from 98.24%)
6771 instructions # 0.001 IPC (scaled from 98.99%)
111114049 branches # 422.908 M/sec (scaled from 99.37%)
8495 branch-misses # 0.008 % (scaled from 95.91%)
12152161 cache-references # 46.252 M/sec (scaled from 96.16%)
7245338 cache-misses # 27.576 M/sec (scaled from 95.49%)

0.265238069 seconds time elapsed
```

上面告诉我们，程序 t1 是一个 CPU bound 型，因为 task-clock-msecs 接近 1。缺省情况下，除了 task-clock-msecs 之外，perf stat 还给出了其他几个最常用的统计信息：

- (1) Task-clock-msecs: CPU 利用率，该值高，说明程序的多数时间花费在 CPU 计算上而非 IO。
- (2) Context-switches: 进程切换次数，记录了程序运行过程中发生了多少次进程切换，频繁的进程切换是应该避免的。
- (3) CPU-migrations: 表示进程 t1 运行过程中发生了多少次 CPU 迁移，即被调度器从一个 CPU 转移到另外一个 CPU 上运行。
- (4) Cycles: 处理器时钟，一条机器指令可能需要多个 cycles
- (5) Instructions: 机器指令数目。
- (6) IPC: 是 Instructions/Cycles 的比值，该值越大越好，说明程序充分利用了处理器的特性。
- (7) Cache-references: cache 命中的次数
- (8) Cache-misses: cache 失效的次数。

19. Perf top 用于实时显示当前系统的性能统计信息。该命令主要用来观察整个系统当前的状态，比如可以通过查看该命令的输出查看当前系统最耗时的内核函数或某个用户进程。类似 top 命令。通过添加 -e 选项，您可以列出造成其他事件的 TopN 个进程 / 函数。比如 -e cache-miss，用来看看谁造成的 cache miss 最多。

20. perf record 记录单个函数级别的统计信息，并使用 perf report 来显示统计结果。

```
perf record -e cpu-clock ./t1
```

perf report

结果如下图所示：

```
# Samples: 592670
#
# Overhead Command Shared Object Symbol
# .....
#
99.93% t1 ./t1 [.] longa
0.02% perf [kernel] [k] read_tsc
0.01% t1 [kernel] [k] spin_unlock_irqrestore
0.00% t1 [kernel] [k] kmem_cache_free
0.00% t1 [kernel] [k] file_free_rcu
0.00% t1 [kernel] [k] finish_task_switch
0.00% t1 [kernel] [k] rcu_process_callbacks
0.00% t1 [kernel] [k] add_timer_randomness
0.00% t1 [kernel] [k] do_softirq
0.00% perf [kernel] [k] spin_unlock_irqrestore
0.00% t1 [kernel] [k] check_for_new_grace_period
0.00% t1 [kernel] [k] scsi_softirq_done [scsi_mod]
0.00% perf [kernel] [k] ext3_get_blocks_handle [ext3]
0.00% t1 ./t1 [.] fool
0.00% perf [kernel] [k] kmem_cache_alloc
0.00% perf [kernel] [k] do_get_write_access [jbd]
0.00% t1 [kernel] [k] rcu_process_gp_end
0.00% t1 [kernel] [k] blk_update_bidi_request
```

使用 perf 的 -g 选项便可以得到需要的信息：

```
# Samples: 1022/0
#
# Overhead Command Shared Object Symbol
# .....
#
99.68% t1 ./t1 [.] longa
|
|--91.04%-- fool
|         foo
|         main
|         _libc_start_main
|         0x80482f1
|
|--8.96%-- foo2
|         foo
|         main
|         _libc_start_main
|         0x80482f1
|
--0.00%-- [...]

0.11% perf [kernel] [k] read_tsc
|
|--79.31%-- do_gettimeofday
```

-a 选项表示对所有 CPU 采样，如果只需针对特定的 CPU，可以使用”-C”项。

dstat

21.dstat 是 Linux 终端下全能系统监控工具。输入 dstat 命令，输出如下：

---total-cpu-usage---						-dsk/total-		-net/total-		---paging---		---system---	
usr	sys	idl	wai	hiq	siq	read	writ	recv	send	in	out	int	csw
1	0	97	1	0	0	5655B	475k	0	0	0	0	483	509
0	1	96	0	0	3	0	136k	774B	1380B	0	0	337	202
0	0	100	0	0	0	0	416k	6086B	11k	0	0	208	180
0	0	99	0	0	0	0	136k	1622B	1190B	0	0	178	196
1	0	99	0	0	0	0	0	204B	428B	0	0	243	197
0	0	100	0	0	0	0	0	3985B	3530B	0	0	173	182
0	0	93	6	0	0	0	1960k	204B	428B	0	0	457	897
0	0	98	1	0	0	0	1248k	5906B	15k	0	0	250	188
0	0	100	0	0	0	0	120k	204B	428B	0	0	174	184
0	0	100	0	0	0	0	120k	2040B	1952B	0	0	242	220
0	0	100	0	0	0	0	0	204B	428B	0	0	173	173
0	1	96	1	0	2	0	400k	3119B	1952B	0	0	447	257
0	0	100	0	0	0	0	168k	140B	428B	0	0	181	185
0	0	100	0	0	0	0	128k	6254B	13k	0	0	178	202
1	0	99	0	0	0	0	160k	140B	428B	0	0	301	240
0	0	100	0	0	0	0	0	5057B	3530B	0	0	200	206

默认输出显示的信息：

- (1) CPU 状态：CPU 的使用率。这项报告更有趣的部分是显示了用户，系统和空闲部分，这更好地分析了 CPU 当前的使用状况。如果你看到"wait"一栏中，CPU 的状态是一个高使用率值，那说明系统存在一些其它问题。当 CPU 的状态处在"waits"时，那是因为它正在等待 I/O 设备（例如内存，磁盘或者网络）的响应而且还没有收到。
- (2) 磁盘统计：磁盘的读写操作，这一栏显示磁盘的读、写总数。
- (3) 网络统计：网络设备发送和接受的数据，这一栏显示的网络收、发数据总数。
- (4) 分页统计：系统的分页活动。分页指的是一种内存管理技术用于查找系统场景，一个较大的分页表明系统正在使用大量的交换空间，或者说内存非常分散，大多数情况下你都希望看到 page in（换入）和 page out（换出）的值是 0 0。
- (5) 系统统计：这一项显示的是中断（int）和上下文切换（csw）。这项统计仅在有比较基线时才有意义。这一栏中较高的统计值通常表示大量的进程造成拥塞，需要对 CPU 进行关注。

22.默认情况下，dstat 每秒都会刷新数据。如果想退出 dstat，你可以按"CTRL-C"键。

23.dstat 可以通过传递 2 个参数运行来控制报告间隔和报告数量。例如，如果你想要 dstat 输出默认监控、报表输出的时间间隔为 3 秒钟，并且报表中输出 10 个结果，你可以运行如下命令：

```
dstat 3 10
```

24.在 dstat 命令中有很多参数可选，你可以通过 man dstat 命令查看，大多数常用的参数有这些：

- (1) -l：显示负载统计量
- (2) -m：显示内存使用率
- (3) -r：显示 I/O 统计

- (4) -s : 显示交换分区使用情况
- (5) -t : 将当前时间显示在第一行
- (6) -fs : 显示文件系统统计数据
- (7) -nocolor : 不显示颜色
- (8) -socket : 显示网络统计数据
- (9) -tcp : 显示常用的 TCP 统计
- (10) -udp : 显示监听的 UDP 接口及其当前用量的一些动态数据

25.当然不止这些用法，dstat 附带了一些插件很大程度地扩展了它的功能。你可以通过查看/usr/share/dstat 目录来查看它们的一些使用方法，常用的有这些：

- (1) --disk-util : 显示某一时间磁盘的忙碌状况
- (2) --freespace : 显示当前磁盘空间使用率
- (3) --proc-count : 显示正在运行的程序数量
- (4) --top-bio : 指出块 I/O 最大的进程
- (5) --top-cpu : 图形化显示 CPU 占用最大的进程
- (6) --top-io : 显示正常 I/O 最大的进程
- (7) --top-mem : 显示占用最多内存的进程

查看全部内存都有谁在占用：

```
dstat -g -l -m -s --top-mem
```

输出一个 csv 格式的文件用于以后，可以通过下面的命令：

```
# dstat -output /tmp/sampleoutput.csv -cdn
```

iostat

1. iostat 主要用于监控系统设备的 IO 负载情况，iostat 首次运行时显示自系统启动开始的各项统计信息，之后运行 iostat 将显示自上次运行该命令以后的统计信息。用户可以通过指定统计的次数和时间来获得所需的统计信息。iostat 也有一个弱点，就是它不能对某个进程进行深入分析，仅对系统的整体情况进行分析。

```
iostat -d -k 2
```

参数-d 表示，显示设备（磁盘）使用状态；-k 某些使用 block 为单位的列强制使用 Kilobytes 为单位；2 表示，数据显示每隔 2 秒刷新一次。输出结果如下：

```
[root@lan ~]# iostat -d -k 2
```

```
Linux 2.6.32-642.13.1.el6.x86_64 (lan) 10/02/17 _x86_64_ (1 CPU)
```

Device:	tps	kB_read/s	kB_wrtn/s	kB_read	kB_wrtn
sda	2.06	36.75	8.95	414428	100882

Device:	tps	kB_read/s	kB_wrtn/s	kB_read	kB_wrtn
sda	0.00	0.00	0.00	0	0

数据意义:

- (1) tps: 该设备每秒的传输次数。"一次传输"意思是"一次 I/O 请求"。多个逻辑请求可能会被合并为"一次 I/O 请求"。"一次传输"请求的大小是未知的。
- (2) kB_read/s: 每秒从设备 (drive expressed) 读取的数据量;
- (3) kB_wrtn/s: 每秒向设备 (drive expressed) 写入的数据量;
- (4) kB_read: 读取的总数据量;
- (5) kB_wrtn: 写入的总数量数据量; 这些单位都为 Kilobytes。

2. 指定监控的设备名称为 sda:

```
iostat -d sda 2
```

3. 参数:

- (1) -c: 仅显示 CPU 使用情况;
- (2) -d: 仅显示指定设备的设备利用率;
- (3) -k: 显示状态以千字节每秒为单位, 而不使用块每秒;
- (4) -m: 显示状态以兆字节每秒为单位;
- (5) -p: 仅显示块设备和所有被使用的其他分区的状态;
- (6) -t: 显示每个报告产生时的时间;
- (7) -V: 显示版本号并退出;
- (8) -x: 显示扩展状态。

用 `iostat -x /dev/sda1` 来观看磁盘 I/O 的详细情况:

```
iostat -x /dev/sda1
```

4. iostat 还有一个比较常用的选项-x, 该选项将用于显示和 io 相关的扩展数据。

5. iostat 还可以用-c 参数来获取 cpu 部分状态值:

```
iostat -c 1 10
avg-cpu: %user %nice %sys %iowait %idle
1.98 0.00 0.35 11.45 86.22
avg-cpu: %user %nice %sys %iowait %idle
1.62 0.00 0.25 34.46 63.67
```

6. 指定 -X 选项时, iostat 命令会生成 XML 文件。

7. 示例：

- (1) 要以两秒为时间间隔为逻辑名是 disk1 的磁盘显示一个连续的磁盘报告，请输入以下命令：

```
iostat -d disk1 2
```

- (2) 要以两秒为时间间隔为逻辑名是 disk1 的磁盘显示六个报告，请输入以下命令：

```
iostat disk1 2 6
```

- (3) 要以两秒为时间间隔为所有磁盘显示六个报告，请输入以下命令：

```
iostat -d 2 6
```

- (4) 要以两秒为时间间隔为名为 disk1、disk2 和 disk3 的磁盘显示六个报告，请输入以下命令：

```
iostat disk1 disk2 disk3 2 6
```

- (5) 要打印自引导以来的系统吞吐量报告，请输入以下命令：

```
iostat -s
```

Top

1. top 命令是 Linux 下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于 Windows 的任务管理器。
2. 使用技巧：
 - (1) 在 top 基本视图中，按键盘数字“1”，可监控每个逻辑 CPU 的状况
 - (2) 敲击键盘“b” 打开/关闭 top 进程加亮效果
 - (3) 敲击键盘“x” 打开/关闭排序列的加亮效果，通过” shift + >” 或” shift + <” 可以向右或左改变排序列

Sar

1. sar 工具将对系统当前的状态进行取样，然后通过计算数据和比例来表达系统的当前运行状态。它的特点是可以连续对系统取样，获得大量的取样数据。
2. sar 是查看操作系统报告指标的各种工具中，最为普遍和方便的；它有两种用法；
 - (1) 追溯过去的统计数据
 - (2) 周期性的查看当前数据

3. sar -u : 默认情况下显示的 cpu 使用率等信息。

```
root@MyVPS:/var/log/sysstat# sar 1 3
Linux 2.6.35-22-generic-pae (MyVPS)      12/28/2013      _i686_ (1 CPU)

08:20:08 AM    CPU      %user      %nice      %system      %iowait      %steal      %idle
08:20:09 AM    all        0.00        0.00         0.00         0.00         0.00       100.00
08:20:10 AM    all        0.00        0.00         0.00         0.00         0.00       100.00
```

可以看到这台机器使用了虚拟化技术，有相应的时间消耗； 各列的指标分别是：

- (1) %user 用户模式下消耗的 CPU 时间的比例；
- (2) %nice 通过 nice 改变了进程调度优先级的进程，在用户模式下消耗的 CPU 时间的比例
- (3) %system 系统模式下消耗的 CPU 时间的比例；
- (4) %iowait CPU 等待磁盘 I/O 导致空闲状态消耗的时间比例；
- (5) %steal 利用 Xen 等操作系统虚拟化技术，等待其它虚拟 CPU 计算占用的时间比例；
- (6) %idle CPU 空闲时间比例；

4. sar -q: 查看平均负载

5. sar 命令常用格式

```
sar [options] [-A] [-o file] t [n]
```

其中：

- (1) t 为采样间隔，n 为采样次数，默认值是 1；
- (2) -o file 表示将命令结果以二进制格式存放在文件中，file 是文件名。
- (3) options 为命令行选项，sar 命令常用选项如下：

6. 参数：

- (1) -A: 所有报告的总和
- (2) -u: 输出 CPU 使用情况的统计信息
- (3) -v: 输出 inode、文件和其他内核表的统计信息
- (4) -d: 输出每一个块设备的活动信息
- (5) -r: 输出内存和交换空间的统计信息
- (6) -b: 显示 I/O 和传送速率的统计信息
- (7) -a: 文件读写情况
- (8) -c: 输出进程统计信息，每秒创建的进程数
- (9) -R: 输出内存页面的统计信息
- (10) -y: 终端设备活动情况
- (11) -w: 输出系统交换活动信息

7. 示例：

- (1) 每 10 秒采样一次，连续采样 3 次，观察 CPU 的使用情况，并将采样结果以二进制形式存入当前目录下的文件 test 中，需键入如下命令：

```
sar -u -o test 10 3
```

- (2) 每 10 秒采样一次，连续采样 3 次，观察核心表的状态，需键入如下命令：

```
sar -v 10 3
```

- (3) 每 10 秒采样一次，连续采样 3 次，监控内存分页：

```
sar -r 10 3
```

- (4) 每 10 秒采样一次，连续采样 3 次，报告缓冲区的使用情况，需键入如下命令：

```
sar -b 10 3
```

8. 注意：命令相关栏目中的%user 中的百分号表示列数值为百分比，user 表示用户模式下相关的百分比。依此类推。

Vi

1. 为了做区分，将 vim 的模式分为以下几种：

- 命令模式：默认模式，打开 vim 这进入该模式。
- 编辑模式：使用 a、i、o、s 进入的模式。
- 尾行模式：使用冒号进入的模式。

2. vim 的配置文件是 vimrc，分系统配置和用户配置两种，通过在 vim 的正常模式下使用 version 来查看配置文件的位置：

```
:version
```

结果如下：

```
.....  
系统 vimrc 文件: "$VIM/vimrc"  
用户 vimrc 文件: "$HOME/.vimrc"  
第二用户 vimrc 文件: "~/.vim/vimrc"  
用户 exrc 文件: "$HOME/.exrc"  
defaults file: "$VIMRUNTIME/defaults.vim"  
$VIM 预设值: "/usr/share/vim"  
.....
```

默认的用户配置文件为空，需要自己添加。

3. "set nu": 显示行号。

- 这类"set xxx"的配置大多数都是在配置文件 vimrc 中使用的，同时也可以正常模式下使用。在正常模式下使用时可以加感叹号，例如"set xxx!"表示如果当前配置没显示行号，则显示行号，如果已经显示行号，则取消显示行号。

4. "set cul": 突出显示当前行。

"set cuc": 突出当前列。

5. vim 中有几个与编码有关的选项：

- encoding: Vim 内部使用的字符编码方式，包括 Vim 的缓冲区、菜单文本、消息文本等。
- fileencoding: Vim 中当前编辑的文件的字符编码方式，不管是否新文件，Vim 保存文件时也会将文件保存为这种字符编码方式。
- fileencodings: Vim 自动探测 fileencoding 的顺序列表，启动时会按照它所列出的字符编码方式逐一探测即将打开的文件的字符编码方式，并且将 fileencoding 设置为最终探测到的字符编码方式。因此最好将 Unicode 编码方式放到这个列表的最前面，将拉丁语系编码方式 latin1 放到最后面。
- termencoding: Vim 所工作的终端的字符编码方式。如果 vim 所在的 term 与 vim 编码相同，则无需设置。

'设置编码'

```
set fileencodings=utf-8,ucs-bom,gb18030,gbk,gb2312,cp936
set termencoding=utf-8
set encoding=utf-8
```

支持中文不乱码。

6. 括号匹配是指当你输入一个括号时，系统自动为你匹配对应的括号，例如当你输入右大括号"}"时，系统自动为你高亮大括号的左边部分。使用下列设置括号匹配：

set showmatch

实际上这个功能很多 ide 都有。

7. 设置缩进：

- tabstop: 设置 tab 缩进的距离。
- shiftwidth: 设置自动缩进。
- autoindent: 继承前一种缩进方式。在这种缩进形式中，新增加的行和前一使用相同的缩进形式。

'设置 Tab 长度为 4 空格'

```
set tabstop=4
```

'设置自动缩进长度为 4 空格'

```
set shiftwidth=4
```

'继承前一行的缩进方式，适用于多行注释'
set autoindent

8. Vim 编辑器里默认是不启用鼠标的，也就是说不管你鼠标点击哪个位置，光标都不会移动。通过以下设置就可以启动鼠标：

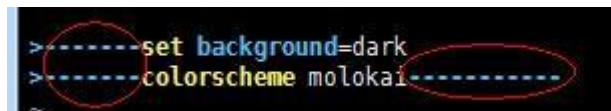
```
set mouse=a  
set selection=exclusive  
set selectmode=mouse,key
```

9. 在 Vim 中通过鼠标右键粘贴时会在行首多出许多缩进和空格，通过 set paste 可以在插入模式下粘贴内容时不会有任何格式变形、胡乱缩进等问题。

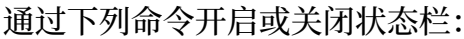
```
set paste
```

10. Vim 编辑器中默认不显示文件中的 tab 和空格符，通过 listchars 可以获得以下的显示效果，方便定位输入错误。

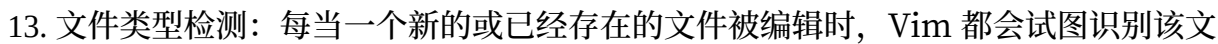
```
set listchars=tab:>-,trail:-
```



11. 状态栏是指下面的白色那栏：



12. 通过"set ruler"来显示光标所在的位置，也就是下面右下角"1,6"这个效果：



件的类型。使用"filetype plugin indent on"来打开文件类型检测功能。实际上该选项包括 filetype on、filetype plugin on 和 filetype indent on 三条基本命令：

- filetype on：打开文件类型检测功能。
- filetype plugin on：打开加载文件类型插件功能。
- filetype indent on：指定不同类型的文件定义不同的缩进格式。

14. 如果 Vim 没有正确检测到文件的类型，则需要通过命令 :set filetype 来手工指定文件类型。

➤ ":xxx"这种命令实际上就是在 vim 下输入冒号进入的那个模式。

15. 要让.vimrc 变更内容生效，一般的做法是先保存 .vimrc 再重启 vim，增加如下设置，可以实现保存 .vimrc 时自动重启加载：

'让 vimrc 配置变更立即生效'

```
autocmd BufWritePost $MYVIMRC source $MYVIMRC
```

16. 在命令模式下按以下按键可进入编辑模式，执行插入操作：

- (1) i：从光标当前所在位置的【前】一个字符处开始插入。
- (2) a：从光标当前所在位置的【后】一个字符处开始插入。
- (3) I：从光标当前所在行的【行首】处开始插入。
- (4) A：从光标当前所在行的【行尾】处开始插入。
- (5) 小写 o：从光标当前所在行的【下一行】处开始插入。
- (6) 大写 O：从光标当前所在行的【上一行】处开始插入。

17. 在命令模式下按以下按键可执行删除操作：

- (1) x：删除光标位置的【一个】字符。
- (2) dd：删除当前光标所在【行】。
- (3) d0：删除从光标所在位置到当前【行首】的内容。
- (4) d\$：删除从光标所在位置到当前【行尾】的内容。
- (5) cw 和 cW：删除从光标所在位置到当前【单词结束】部分的内容并进入插入模式。
- (6) cb 和 cB：删除从光标所在位置到当前【单词开始】部分的内容并进入插入模式。
- (7) dw 和 dW：删除从光标所在位置到当前【单词结束】部分的内容但不进入插入模式。
- (8) db 和 dB：删除从光标所在位置到当前【单词开始】部分的内容但不进入插入模式。

➤ 注意以下几点：

- Vim 的命令中，0 表示行首，\$ 表示行尾。
- w、b 命令用于光标移动。

- cW、cB、dW、dB 命令操作的单词是以空白字符（空格、Tab）分隔的字符。

18. d): 删除当前【句子】从光标位置开始到【句末】的内容。

d(: 删除当前【句子】从光标位置开始到【句首】的内容。

d}: 删除当前【段落】从光标位置开始到【段末】的内容。

d{: 删除当前【段落】从光标位置开始到【段首】的内容。

➤ Vim 命令中用 (和) 表示句子, { 和 } 表示段落。

19. 在命令模式下按以下按键可执行复制、粘贴操作:

- yw: 复制从光标所在位置到当前【单词结束】部分的内容。
- yy: 复制光标所在【行】的所有字符 (包含换行符)。
- 小写 p: 将最后一次删除或复制操作的文本内容粘贴到光标所在字符之【后】。
- 大写 P: 将最后一次删除或复制操作的文本内容粘贴到光标所在字符之【前】。

20. 在命令模式下按以下按键后, 再输入字符可替换原始文件中的内容:

- r: 替换光标当前所在字符一次。
- R: 一直替换光标所在字符, 直到按下[ESC]键为止。

21. 删除、复制操作的操作单位可以加操作次数, 操作对象的范围为: 操作次数 * 操作单位。例如: d3w 命令表示删除三个单词, 10dd 命令表示从光标所在行开始删除后面的十行。

22. 在命令模式下可执行撤销操作:

- u: 撤销最近的一次操作。
- <Ctrl> + r: 恢复最近的一次撤销操作。

23. 在尾行模式下执行命令 ":w a.txt" 可将 vim 当前打开的文件另存为新文件 a.txt。

24. 命令 "<Ctrl> + g" 可显示当前编辑文件名及行数, 可以在不退出 Vim 的情况下了解当前编辑文件的信息。

25. 在执行 Vim 光标移动命令时, 首先要分清楚是采用哪种操作单位: 一个字符、一个句子、一个段落、一行、一屏、一页。例如, 5h 命令表示左移 5 个字符, 8w 命令右移 8 个单词。

26. 在 Vim 移动光标命令中, h、j、k 和 l 分别代表上下左右方向的移动。例如, 5h 命令表示左移 5 个字符, 8l 命令右移 8 个字符。

27. 行级移动

- 0: 移动光标到当前行行首
- \$: 移动光标到当前行行尾
- ^: 移动光标到当前行的第一个非空字符
- nG: 移动光标到当前文件的第 n 行

- :n: 移动光标到当前文件的第 n 行 (同上)

28. 文件首尾移动

- gg: 移动光标到当前文件的第一行
- GG: 移动光标到当前文件的最后一行

29. 单词级移动:

- w 或 W: 移动到下一单词的开头。
- b 或 B: 移动到上一单词的开头。
- e 或 E: 移动到光标所在单词的末尾。

30. 段落级移动:

- }: 移动光标到当前段落的末尾。
- {: 移动到光标到当前段落的开头。

31. 屏幕级移动:

- H: 移动光标到屏幕的第一行。
- M: 移动光标到屏幕的中间一行。
- L: 移动光标到屏幕的最后一行。

32. 翻页:

- Ctrl + f 向前滚动一页。
- Ctrl + b 向后滚动一页。
- Ctrl + u 向前滚动半页。
- Ctrl + d 向后滚动半页。

33. 很多命令都可以和 vim 光标移动命令连动, 基本模式为:

开始位置:命令:结束位置

例如, Vim 命令 0y\$ 拆开分别表示: 0 移动光标到当前行首; y 复制; \$ 当前行尾。所以, 命令 0y\$ 意味着复制光标当前所在行所有内容; Vim 命令 ye, 表示从当前位置拷贝到当前所在单词的最后一个字符。

34. Vim 编辑器.命令可以用于重复执行命令。举例来说, 删除一个单词, 可以使用命令 dw, 接着, 我们可以使用命令 5.再连续删除 5 个单词, 这就是 Vim 中.点命令的重复功能。

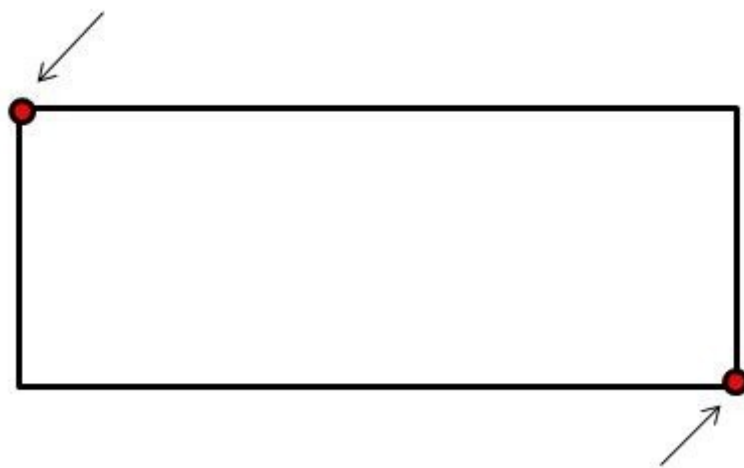
35. Vim 可视化模式下可以选择一块编辑区域, 然后对选中的文件内容执行插入、删除、替换、改变大小写等操作。在 Vim 命令模式下, 输入 v 或者 V 或者 Ctrl + v 都可进入可视化模式。


```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
1 root:x:0:0:root:/root:/bin/bash
2 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
3 bin:x:2:2:bin:/bin:/usr/sbin/nologin
4 sys:x:3:3:sys:/dev:/usr/sbin/nologin
5 sync:x:4:65534:sync:/bin:/bin/sync
6 games:x:5:60:games:/usr/games:/usr/sbin/nologin
7 man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
8 lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
9 mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
10 news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
11 uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
12 proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
13 www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
14 backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
15 list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
16 irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
17 gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
18 nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
19 systemd-network:x:100:102:systemd Network Management,,,:/run/systemd/netif:/usr/sbin/nologin
20 systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd/resolve:/usr/sbin/nologin
- 可视 -- 13,5 顶端
```

36. 按 `v` 或者 `V` 或者 `Ctrl + v` 都可进入可视化模式，这三个 Vim 可视化模式的主要区别在于：

- `v` 字符选择模式：选中光标经过的所有字符。
- `V` 行选择模式：选中光标经过的所有行。
- `<Ctrl> + v` 块选择模式：选中一整个矩形框表示的所有文本。

37. 默认情况下，可视化模式控制右下角端点的位置，当需要调整左上角端点的位置时，可以使用 `o` 键在左上角和右下角之间进行切换。



38. 使用可视化的块选择模式可以很方便地操作多行文本内容，比如一次性注释多行文本，

可以如下操作：

- (1) Ctrl + v 进入块选择模式，并且向上或向下移动光标选择多行。
- (2) 移动光标到行的起始位置。
- (3) 然后按大写的 I 进入行首插入模式，插入注释符。
- (4) 按 Esc 回到命令模式。

39. 在 Vim 命令模式下，输入 / 或 ? 符号就进入了搜索模式，/ 用于正向往下搜索，? 用于反向往上搜索。

40. 在搜索模式下可以对 Vim 打开的整个文本内容进行搜索，当按下 n 时可以继续正向查找下一个相匹配的目前单词。N 的作用与 n 相反，是往上反向搜索目标单词。

41. 在 Vim 命令模式下，如果要搜索当前光标下的单词，除了可以使用 / 和 ? 外，还可以使用特殊命令 * 和 #。假设光标当前所有单词为 the，那么可以在当前光标位置执行命令 *。

42. 为了在 Vim 中高亮显示被搜索的字符，可以使用命令 set hlsearch。

43. 默认 Vim 搜索命令是大小写敏感的，使用命令 :set ignorecase 会使得 Vim 搜索变得不区分大小写。

➤ 不管 ignorecase 选项的值是什么，都可以在搜索命令中使用 \c 来强制使得当前搜索模式不区分大小写，而命令 \C 则会强制当前搜索模式大小写敏感。因此，/the\c 既会查找 the，也会查找到 The。

44. 如果打开了 ignorecase 选项，那么你也应该设置 ":set smartcase"。当 ignorecase 和 smartcase 选项均打开时，只要你的搜索模式中包含大写字母，那 Vim 会认为你当前的搜索是区分大小写的，如果搜索模式中不包含任何大写字母，Vim 则会认为搜索应该不区分大小写。这是个比较“智能的”推测你搜索意图的机制。

例如，在打开上述两个选项的条件下，/The 只会查找到 The，而 /the 既会查找 the，也会查找到 The 等。

45. vim 支持直接使用正则表达式进行搜索：

```
:/^test.*$
```

46. Vim 替换命令的基本语法是：

```
[range]s/源字符串/目标字符串/[option]
```

- range：表示搜索范围，默认表示当前行。range 字段值 1,10 表示从第 1 到第 10 行，% 表示整个文件，相当于 (1,\$)。
- s：substitute 的简写，表示替换
- option：表示操作类型，默认只对第一个匹配的字符进行替换。g 表示全局替换，c 表示操作时需要确认(confirm)，i 表示不区分大小写(ignorecase)。

这些选项可以组合使用。

```
:1,$s/Vim/vim/gc
```

用 vim 来替换 Vim。

47. 当在替换命令中使用 c 选项时，会出现下面这样的确认停下：

```
=====
=  Welcome to the VIM Tutor - Version 1.7  =
=====

vim is a very powerful editor that has many commands, too many to
explain in a tutor such as this. This tutor is designed to describe
enough of the commands that you will be able to easily use Vim as
an all-purpose editor.

The approximate time required to complete the tutor is 25-30 minutes,
depending upon how much time is spent with experimentation.

ATTENTION:
The commands in the lessons will modify the text. Make a copy of this
file to practise on (if you started "vimtutor" this is already a copy).

replace with vim (y/n/a/q/l/^E/^Y)?
```

- y: 确认执行这个替换。
- n: 取消这个替换。
- a: 执行所有替换且不再询问。
- q: 退出而不做任何改动。
- l: 替换完当前匹配点后退出(last)。
- Ctrl + E: 向上翻滚一行。
- Ctrl + Y: 向下翻滚一行。

48. 在插入模式下，Vim 可以不借助任何插件实现自动补全功能：

- 单词补全：Ctrl + n。
- 行补全：在 Vim 插入模式下输入已经存在行的第一个单词，再按 Ctrl + x、Ctrl + l 命令，就会列出该整行出来实现 Vim 行自动补全。

49. 在自动补全中，可以使用字典来自定义补全功能。字典是一个文件，里面是需要补全的单词列表。既可以是一个单词一行，也可以是空格隔开的单词列表。

50. 若要实现基于单词表的 Vim 自动补齐，需要设置以下步骤：

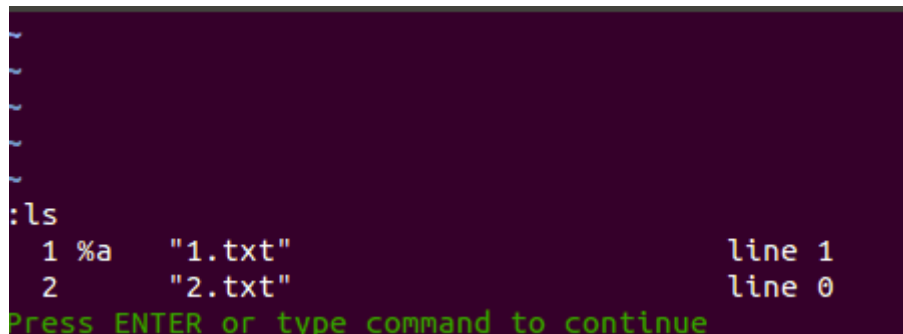
- (1) 在 ~/.vimrc 配置文件中加入代码：set dictionary=~/.dict.txt
dictionary+=~/.dict.txt。其中 dict.txt 是单词表文件。
- (2) 打开 Vim，在插入模式下输入 Ctrl + x 后再输入 Ctrl + k，就能看到 dict.txt 文件中定义的单词。
- (3) 若想直接通过 Ctrl + n 命令就显示其中的列表，再配置 .vimrc 文件，加入 set
complete=k complete+=k

51. Vim 打开文件进行编辑时其实编辑的是文件在内存中的映像，即 Vim 中的缓冲区。

52. Vim 支持同时在多个缓冲区进行操作，每打开一个文件，Vim 就会创建一个对应缓冲区。vim 直接列举文件的方式打开多个文件。

```
$ vim 1.txt 2.txt
```

53. `:ls` 和 `:buffers` 命令会列出所有被载入到内存中的缓冲区的列表，`%` 符号指明哪个缓冲区在当前窗口中：



```
:ls
 1 %a    "1.txt"                line 1
 2       "2.txt"                line 0
Press ENTER or type command to continue
```

54. 使用 Vim 缓冲区列表：

- `:bnext` 和 `:bprev` 命令正向或反向遍历列表。
- 命令 `Ctrl + ^` 可以在当前文件和轮换文件间快速切换。
- `:buffer n` 命令直接凭编号 `n` 跳转到该缓冲区。

55. 可以用 `:bdelete` 命令删除 Vim 缓冲区。

- `:bdelete n1 n2 n3`：删除特定缓冲区。
- `:n,m bdelete`：删除一个编号区间段的缓冲。

56. Vim 操作范围的表示方法为：`.` 表示当前行，`$` 表示结束，`%` 表示所有行，`+` 表示指定行下面的数行，`-` 表示指定行上面的数行，可灵活表示当前 Vim 打开文件的任意范围。

- `:1,$d`：所有行。
- `:$d`：最后一行。
- `:. ,5d`：当前行至第 5 行。
- `:. ,5d`：当前行至第 5 行。
- `:1,+3d`：第一行至当前行再加 3 行。
- `:. ,-3d`：当前行及向上的 3 行。

57. 除了 `"` 外，`;` 也可以分割行号用于表示 Vim 操作范围。区别在于，`"a,b` 的 `b` 是以当前行作为参考，而 `"a;b` 的 `b` 是以 `a` 行作为参考例如，假设当前光标所在为第 5 行，`:1,+1d` 命令会删除第 1 行至第 6 (`5+1`) 行，而 `:1;+1d` 命令则会删除第 1 行和第 2 (`1+1`) 行。

58. 如果想设置多个寻找条件，只需要在条件前再加上 /:

```
:/foo//bar//quux/d
```

首先在当前行之后寻找第一个包含“foo”字符的那一行，然后在找到的这一行之后寻找第一个包含“bar”字符的那一行，然后再在找到的这一行之后寻找第一个包含“quux”的那一行。注意第二个开始是两个//。

59. Vim 常用的折叠形式主要有 indent 和 syntax，只需要 Vim 配置文件 ~/.vimrc 中增加以下配置：

```
' 基于缩进进行代码折叠
set foldmethod=indent
' 启动 Vim 时关闭折叠
set nofoldenable
```

Vim 打开文件后，重复使用操作命令 za 可打开或关闭当前折叠；zM 用于关闭所有折叠，zR 则用来打开所有折叠。

60. Vim 分屏是指在同一个 Vim 窗口中同时显示多个文件的内容。

61. 使用 Vim 打开文件时，可以通过参数 -On 或 -on 来启动分屏。n 代表整数，表示将整个屏幕分成 n 部分。

62. Vim 尾行模式下执行命令 ":vsplit filename" 或缩写形式 ":vsp filename" 可实现 Vim 垂直方向分割屏幕，且打开新的文件 filename。

63. Vim 尾行模式下执行命令 ":split filename" 或缩写形式 ":sp filename" 可实现在 Vim 水平方向分割屏幕，且打开新的文件 filename。

64. Vim 命令行模式下执行下列命令可将当前打开的文件进行垂直分割：Ctrl+w v。

➤ 这里的 Ctrl+w v 这样的命令的意思是先按 Ctrl+w，再按 v 键。依此类推。

65. Vim 命令行模式下执行下列命令可将当前打开的文件进行水平方向分割：Ctrl+w s。

66. 在 Vim 命令行模式下，通过下列方式在当前的分割屏幕中按顺时针方向切换屏幕：Ctrl+w w。

67. 按指定方向切换屏幕：

- 命令 Ctrl+w h 用于把光标移到左边的屏幕中。
- 命令 Ctrl+w l 用于把光标移到右边的屏幕中。
- 命令 Ctrl+w j 用于把光标移到下边的屏幕中。
- 命令 Ctrl+w k 用于把光标移到上边的屏幕中。

68. 设置分屏大小

- 命令 Ctrl+w = 表示设置所有的分屏幕都有相同的高度
- 命令 Ctrl+w + 用于增加当前屏幕的高度
- 命令 Ctrl+w - 用于减少当前屏幕的高度

69. 命令 Ctrl+w c 用于关闭当前操作的 Vim 分屏幕。

70. Vim 中的 c 命令表示修改(change)，用于删除并进入插入模式，后面可以接一个移动范围，表示修改光标点到移动终点之间的内容并进入编辑模式。例如，Vim 命令 ciw 表示删除当前光标所在单词并进入插入模式。

71. Vim 命令 c\$ 表示删除光标当前位置到本行结尾处的内容并进入 Vim 插入模式。

72. Vim 文本对象选择范围：

- (1) iw：当前单词 (inside word)
- (2) aw：当前单词及一个空格 (around word)
- (3) iW：当前字串 (inside WORD)
- (4) aW：当前字串及一个空格 (around WORD)
- (5) is：当前句子 (inside sentence)
- (6) as：当前句子及一个空格 (around sentence)
- (7) ip：当前段落 (inside paragraph)
- (8) ap：当前段落及一个空行 (around paragraph)

例如，删除当前单词为 diw，替换当前单词为 ciw。

➤ a 和 i 相比，多了个空格或空行。

73. Vim 的删除、复制与粘贴命令均需要使用 Vim 寄存器。

74. Vim 提供了一组以 26 个英文字母命名的有名寄存器。用小写字母引用有名寄存器会覆盖该寄存器的原有内容，而用大写字母引用则会将新内容追加到该寄存器的原有内容之后。也就 qA 和 qa。

➤ 寄存器应该就相当是缓存。

75. Vim 中的宏可以理解为把指定的一系列操作命令录制并缓存到某个 Vim 寄存器中，然后在需要的时候将缓存的一系列指令进行回放，达到重复操作的目的，其作用类似于 vim 重复操作命令。

76. 在 Vim 命令模式下，按下 q 寄存器名后开始进入宏录制状态。寄存器名是指 a、b、c 等 26 个英文字母命名的有名寄存器，或 0-9 等 10 个数字寄存器。

在 vim 宏录制状态下，按 q 结束宏录制。

77. 假设录制的宏记录在寄存器 a，可以使用 Vim 命令 @a 执行这个宏，也可以加上执行次数 10@a 执行 10 次宏。

78. 可以使用以下方式修改 vim 寄存器 a 中的宏：

- 先按 G 跳转到文件末尾，然后按 o 新加一行后直接用 ESC 退出到命令模式。
- 使用命令 "ap 将寄存器 a 中的命令宏粘贴到当前位置，然后编辑这一行的宏内容，编辑结束后用 O 回到行首。

- 使用命令 "ayy 将当前行的内容复制到寄存器 a 中，达到修改寄存器 a 宏内容的目的。

79. 可以使用 linux 命令行的 let 命令给寄存器赋值达到保存特定宏内容到指定寄存器的目的。例如，let @a="0iSystem.out.println(^[A];^[", 将上述的操作命令保存到寄存器 a，然后可以用 @a 命令使用刚刚制作的宏。

80. global 命令用于在所有匹配行上运行可执行的 Ex 命令，缩写形式为 ":g"。

- 出于历史原因，在 Vim 命令行模式中执行的命令又被称做 Ex 命令。Ex 命令都需要先输入 : 后按回车才能执行，按 / 调出查找提示符或用 <Ctrl-r>= 访问表达式寄存器时，命令行模式也会被激活，但这两个并不是 Ex 命令。

81. Vim global 命令形式为：

```
:[range]g[lobal][!]/{pattern}/[cmd]
```

- range :表示操作范围，Vim:global 命令的默认作用范围是整个文件 (用 % 表示)。
- !: 表示反转 :global 命令的行为，将在没有匹配到指定模式的行上执行命令。
- pattern: 指定 :global 命令匹配模式，若将该域留空，Vim 会自动使用最近一次的查找模式。
- cmd: 除 :global 命令之外的任何 Ex 命令，Vim 缺省使用 :print 命令 (缩写 :p)。

```
:g/pattern/m$
```

将所有匹配的行移动到文件的末尾。

82. Vim 中有两个比较基础的概念：操作符和动作。一般来说，操作符用于删除或修改文本，动作 (用 {motion} 表示) 是指移动光标的命令或动作。

83. Vim 自带很多快捷键，再加上各类插件的快捷键，大量快捷键出现在单层空间中难免引起冲突，为缓解该问题，引入了前缀键。

84. Vim 中的前缀键类似于 C++ 中的命名空间，可以理解为是某个 Vim 命令开始执行的标识。默认的 Vim 前缀键是 \，在 Vim 命令模式下，可以使用命令：

```
:let mapleader=","
```

设置 Vim 的 Leader 键为 ","。

85. 使用 Vim 命令 ":map" 可以将键盘上的某个按键与 Vim 的命令映射起来，完成 Vim 快捷键的绑定。例如 ":map a b" 在 map 生效的情况下，按下 a 就等同于按下了 b。

86. 在 map 命令前加上前缀可以组合成几种不同的命令，表示在不同的 Vim 模式下生效。

- n: 在普通模式下生效。
- i: 在插入模式下生效。
- v: 在可视化模式下生效。
- c: 在命令模式下生效。
- o: 在命令等待时生效，比如输入 d 之后会等待输入下一个字符，可能是 d 或者数字。
- un: 删除键的映射。

- **nore**: 非递归, 意思是将 a 映射为 b, b 映射为 c, 输入 a 的时候不会被映射为 c, 而只会映射为 b。

" 在插入模式下非递归映射(为)<Esc>i"

```
inoremap ( )<Esc>i
inoremap [ ]<Esc>i
inoremap { }<Esc>i
inoremap " ""<Esc>i
```

这样输入(、[、{和“的时候都会自动补全, 并且把光标移到括号的内部。

87. 如果想在一系列连续行上执行一条普通模式命令, 可以用 Vim 的":normal"命令, 缩写形式":norm"。此命令在与 . 命令 或 vim 宏结合使用时, 只需花费很少的努力就能完成大量重复性任务。

88. Vim normal 命令的使用形式为:

```
:{range}norm[!] {commands}
```

表示在 {range} 指定的范围内的每行执行若干 普通模式命令 {commands}。

{commands} 不能以空格开始, 除非在空格前面加个计数 1。

例如, vim 命令":normal ggdd"会将光标移动到文件的第一行并删除它。(dd)。

89. normal 命令中的可选参数 "!"用于指示 vim 在当前命令中不使用任何 vim 映射; 如果没有显式使用 ! 选项, 即便是执行一个非递归映射 (noremap) 命令, 它的参数仍有可能被重新映射。

例如, 假设已经设置了 vim 映射 :nnoremap G dd, 则在 vim 普通模式按下 G 将执行命令 dd, 即会删除一整行; 此时, 若在 vim 命令行模式下执行命令 :normal G 同样将删除当前行而不会跳转到当前文件的末行。

90. 虽然用":normal "命令可以执行任意的普通模式命令, 但当它和 Vim 的重复命令结合在一起时最为强大。当需要重复的操作比较简单时, 可以配合使用":normal ." 命令, 而如果需要执行的重复操作比较复杂时, 可以使用":normal @q"命令来发挥 vim 宏强大的操作记录功能。

```
:normal A;
```

文件每行结尾加分号。

91. Vim 的位置标记可以实现在文档中的快速跳转。可以通过命令 mark 或缩写形式 m 手动设置位置标记, Vim 也会自动记录某些自身感兴趣的位置点, 辅助实现某些快速跳转功能。
92. Vim 允许在打开的文件中放置自定义的标记。命令 ma 表示用 a 标记当前的光标位置。
93. 可以在文本中使用 26 个标记。小写标记值在每个缓冲区局部可见, 而大写标记则全局可见。
94. Vim 的位置标记可以实现在文档中的快速跳转。可以通过命令 mark 或缩写形式 m 手动设置位置标记, Vim 也会自动记录某些自身感兴趣的位置点, 辅助实现某些快

速跳转功能。例如使用命令"ma"来记录位置点 a，然后使用'a 来跳转。

跳转标记之后，还可以使用两个单引号跳回原来的位置。

95. 利用:marks 命令，可以列出所有标记。

96. 使用":delmarks a b c"命令，可以删除某个或多个标记；而":delmarks!"命令，则会删除所有标记。

97. ~/.viminfo 文件由 Vim 系统自动生成，用于记录和保存一些 Vim 的操作记录和状态信息，便于重启 Vim 进程后能恢复之前的各种历史操作行为。使用":wviminfo file_name"命令，可以手动创建一个 viminfo 文件。

98. Vim 的":copy"命令的缩写形式":co"或":t"，可以把一行或多行从文档的某个位置复制到另一个位置，而":move"命令则可以把一行或多行移到文档的其他地方。

99. 整行拷贝可以在 Vim 普通模式下用 y 命令解决，但有一个缺点就是必须把光标移到要拷贝的行上才能执行该操作。

100. copy 命令的格式为：

```
:[range]copy {address}
```

[range] 表示要复制的行范围，{address} 表示复制的目标位置，这两个参数都可以缺省，用于表示 Vim 光标所在当前行。下面是一些例子：

- :3,5t.: 把第 3 行到第 5 行的内容复制到当前行下方
- :t5: 把当前行复制到第 5 行下方
- :t.: 复制当前行到当前行下方 (等价于普通模式下的 yyp)
- :t\$: 把当前行复制到文本结尾
- :<,>t0: 把高亮选中的行复制到文件开头

101. Vim 行移动命令 :move 格式为：

```
[range]move{address}
```

例如，在 Vim 命令行模式下执行命令 :<,>m\$ 可以把当前高亮选中的所有行移动到文件末尾处，而 :8,10m2 可以把当前打开文件的第 8~10 行内容移动到第 2 行下方。

102. Vim 标签页和浏览器的标签页差不多，是可以容纳一系列 Vim 窗口的容器。

```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
+ {root} lan
26 dnsmasq:x:108:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
27 rtkit:x:109:114:RealtimeKit,,,:/proc:/usr/sbin/nologin
28 cups-pk-helper:x:110:116:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
29 speech-dispatcher:x:111:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
30 whoopsie:x:112:117:/:/nonexistent:/bin/false
31 kernoops:x:113:65534:Kernel Oops Tracking Daemon,,,:/usr/sbin/nologin
32 saned:x:114:119:/:/var/lib/saned:/usr/sbin/nologin
33 pulse:x:115:120:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin
34 avahi:x:116:122:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin
35 colord:x:117:123:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
36 hplip:x:118:7:HPLIP system user,,,:/var/run/hplip:/bin/false
37 geoclue:x:119:124:/:/var/lib/geoclue:/usr/sbin/nologin
38 gnome-initial-setup:x:120:65534:/:/run/gnome-initial-setup:/bin/false
39 gdm:x:121:125:Gnome Display Manager:/var/lib/gdm3:/bin/false
40 lan:x:1000:1000:lan,,,:/home/lan:/bin/bash
41 mysql:x:122:127:MySQL Server,,,:/nonexistent:/bin/false
42 irooo^[
43 t
44 rootrootroot:x:0:0:root:/root:/bin/bash
```

43.1

底端

菜单栏下面的那两个 root 和 lan 就是两个标签页，灰色背景的标签页面为非当前标签页，所以上面这个的当前标签页为 root。

103. Vim 命令行模式下使用命令 :tabedit 文件名 或 tabe 文件名 可以在新标签页中打开指定的文件。也可以使用 :tabnew 文件名 在新标签页中编辑新的文件。
104. Vim 命令行命令 :tab split 会在新的 Vim 标签页中打开当前活动缓冲区对应的文件。
105. Vim 默认最多只能打开 10 个标签页，可以用 set tabpagemax=20 设置最大标签页上限为 20。
106. Vim 普通模式下使用命令 gt 或命令行命令 :tabnext (缩写形式 :tabn) 可以移动到下一个标签页。相反地，gT 或 :tabprevious (缩写形式 :tabp) 可以移动到上一个标签页。
107. 快速移动到第一个标签页，可以使用 :tabfirst 或 :tabr 命令，而命令 :tablast 可用来快速移动到最后一个标签页。
108. Vim 命令行命令 :tabs 可以列出所有的标签页和它们包含的窗口，当前窗口用 > 表示，修改过的缓冲区用 + 表示。
109. Vim 命令行命令 :tabclose (缩写形式为 :tabc) 可以关闭当前标签页及其中的所有窗口，而命令 :tabonly (缩写形式为 :tabo) 将关闭所有其他标签页，只保留当前活动标签页。
110. Vim 命令行命令 :tabmove n (缩写形式为 :tabm) 用于将当前标签页移到第 n 个标签页之后。如果没有为 :tabm 命令指定参数 n，那么会将当前标签页移动到最后位置。例如，命令 :tabm 1 将把当前标签页移动到第 2 的位置。

➤ 注: 标签页编号是从 0 开始计数的。

111. Vim 命令行命令 `:tabdo` 可以同时多个标签页中执行命令。假如当前一共打开了多个标签页, 若想把这些文件中的 `food` 都替换成 `drink`, 可以使用 Vim 命令:

```
:tabdo %s/food/drink/g
```

一次完成对所有文件的替换操作, 而不用针对每个文件重复操作。

112. 默认情况下, 只有用户新建了标签页才会在窗口上方显示标签栏, 如果希望总是显示标签栏, 那么可以用 `set showtabline=2` 进行设置。更多内容, 可以参考 Vim 入门级配置 和 使用 Powerline 插件强化 Vim 状态栏。

如果希望完全不显示标签栏, 可以使用 `set showtabline=0` 进行设置。

113. Vim 参数列表记录了在启动时作为参数传递给 Vim 的文件列表, 在 Vim 命令行模式下执行 `:args` 命令可以打印出当前参数列表的内容。可以用 `:next` 及 `:prev` 命令遍历参数列表中的所有文件。

114. Vim 命令行命令 `autocmd` 用于指示 Vim 监听某一类事件, 一旦该事件发生, Vim 将执行指定的命令。在文件读写, 缓冲区或窗口进出, 甚至 Vim 退出等事件发生时, 都可以指定要自动执行的命令。`autocmd` 语句的这种检测机制可用于设置用户感兴趣的事件发生时自动执行某些操作。

```
autocmd BufNewFile *.cpp exec ":call SetTitle()"
func SetTitle()
    call setline(1,"/*")
    call append(line("."), " * Copyright (C) ".strftime("%Y")." All rights reserved.")
    call append(line(".")+1, " *")
    call append(line(".")+2, " * FileName    : ".expand("%:t"))
    call append(line(".")+3, " * Author      : vim.ink")
    call append(line(".")+4, " * Email       : admin@vim.ink")
    call append(line(".")+5, " * Date        : ".strftime("%Y 年%m 月%d 日"))
    call append(line(".")+6, " * Description : ")
    call append(line(".")+7, " */")
endfunc
"自动将光标定位到末尾"
autocmd BufNewFile * normal G
```

将上面的 Vim 配置添加到 Vim 配置文件 `~/.vimrc` 中, 实现新建后缀名为 `.cpp` 时自动在新文件中添加文件作者注释信息并自动将光标定位到文件末尾。

115. 在 Vim 的命令行模式中, 在命令前加一个 `!` 前缀就可以调用外部的 shell 程序。例如, 如果想在 Vim 内部查看当前目录下的所有文件, 可以在 Vim 命令行模式下运行命令 `!:ls`

➤ `!:ls` 和 `:ls` 是两个不同的 Vim 命令, 前者是在 Vim 中调用 shell 命令 `ls`, 用于列出目录下的所有文件, 后者调用的是 Vim 内置命令 `ls`, 用来显示当前缓冲区列表的内容。

116. 上面介绍的 `!{cmd}` 这种语法适用于执行一次性命令, 如果想在 shell 中连续

121. 设置完 `makeprg` 选项后，可以直接在 Vim 中执行命令 `:make` 来进行源码编译。
122. `make` 之后的每一条出错信息，Vim 都会在 `quickfix` 列表中为其创建一项记录。我们可以上下浏览这些记录项，让 Vim 跳转到产生错误信息所在的源文件行上。输入 `:cc` 命令让 Vim 再次显示此信息。也可以使用 `:cw` 命令打开一个 `quickfix` 窗口，把所有的出错信息显示出来。
123. Vim 的 `:make` 命令不仅限于调用外部的 `make` 程序，也可以调用任何安装在操作系统上的编译器。
124. 快速浏览 `quickfix` 列表的命令。
- (1) `":copen` 或 `:cw`：如果有错误，打开 `quickfix` 窗口。
 - (2) `":cc N"`：跳转到第 `n` 项，显示详细内容。
 - (3) `":cclose"`：关闭 `quickfix` 窗口。
 - (4) `":cnext` 或 `:cn`：跳转到下一项。
 - (5) `":cprevious` 或 `:cprev` 或 `:cp`：跳转到上一项。
 - (6) `":cfirst"`：跳转到第一项。
 - (7) `":clast"`：跳转到最后一项。
 - (8) `":cnfile"`：跳转到下一个文件中的第一项。
 - (9) `":cpfile"`：跳转到上一个文件中的最后一项。
125. Vim 会区分实际行与屏幕行：
- 实际行：vim 中实际显示的行号。
 - 屏幕行：如果一行内容太长而分为两行，这第二行就是屏幕行。
126. 使用 Vim 命令 `gj` 和 `gk` 按屏幕行向下、向上移动光标。
127. Vim 会把文件中的文件名当成一个超链接，设置了正确的 `path` 选项后，在 Vim 普通模式下可以用 `gf` 命令跳转到当前光标下文件名对应的文件。
128. `path` 选项定义了一个目录列表，在使用 `gf`、`find`、以及 `CTRL-W f` 等 vim 命令时，如果使用的是相对路径，那么就会在 `path` 选项定义的目录列表中查找相应的文件。`path` 选项以逗号分隔各目录名。

```
:set path+=./include,,
```

129. 每次用 `gf` 命令时，Vim 都会在跳转列表中增添一条记录。所以在新打开的 `event.h` 文件中，可以用 `<Ctrl-o>` 命令返回到上一次所在的 `event.c` 文件中。
130. `<Ctrl-o>` 命令用于后退到之前的文件中，而 `<Ctrl-i>` 命令则是前进到之前打开过的文件中。
131. Vim 内置拼写检查器，使用命令 `:set spell` 可以对当前文件中所有未在字典中

出现过的单词进行标记并高亮显示。

132. Vim 的拼写检查器开启后将使用英语字典作为缺省的拼写字典进行单词比较, 可以通过配置 `spelllang` 选项更改缺省设置。

可以使用 `zg` 命令将当前光标下的单词添加到 Vim 的拼写字典中; 使用 `zw` 命令将光标所在处的单词标记为拼写错误, 即将该单词从拼写文件中删除); 此外, Vim 专门提供了一条撤销命令 `zug`, 用于撤销对光标下单词所执行的 `zg` 或 `zw`。

`spellfile` 选项用于指定由 `zg` 和 `zw` 命令添加、删除的单词所保存的文件路径。如以下配置所示, 可以同时指定多个拼写文件, 维护多份单词列表。

```
setlocal spelllang=en_us
setlocal spellfile=~/.vim/spell/en.utf-8.add
setlocal spellfile+=~/vim.ink/vimtutor/jargon.utf-8.add
```

133. 下表总结了在普通模式下操作 Vim 拼写检查器的基本命令。

- `]s`: 跳到下一处拼写错误。
- `[s`: 跳到上一处拼写错误。
- `z=`: 为当前单词提供更正建议。
- `zg`: 把当前单词添加到拼写文件中。
- `zw`: 把当前单词从拼写文件中删除。
- `zug`: 撤销针对当前单词的 `zg` 或 `zw` 命令。

134. vim 支持会话保存和恢复的功能: 当重新打开 IDE 时, 软件会自动恢复到上次退出时的环境, 包括恢复窗口布局、所打开的文件列表等。

135. Vim 会话具体保存哪些信息由 `sessionoptions` 选项决定, 可通过 `:set sessionoptions`。

```
:set sessionoptions=blank,buffers,curdir,folds,help,options,tabpages,winsize
```

136. 在 Vim 命令行模式下使用命令 `:mksession [file_name]` 可用来创建一个会话文件, 创建的会话文件保存在当前目录下, 如果省略文件名则会创建一个名为 `Session.vim` 的会话文件。

➤ 会话文件保存的内容就是窗口布局、所打开的文件列表等。

137. 如果 `session` 文件所在的整个文件都被移动过, 则 `session` 文件会失效。要解决这个问题, 需要在 `sessionoptions` 去掉 `curdir` 并增加 `sesdir`。设置此选项后, `session` 文件中保存的是文件的相对路径, 而不是绝对路径。这样每次载入 `session` 文件时, 此文件所在的目录就被设为 Vim 的当前工作目录。

138. 使用 `:mksession` 命令创建完会话文件后, 可以在 Vim 命令行模式下使用 `:source file_name` 来导入指定的会话文件。

139. 使用 `:grep` 命令，可以在不退出 Vim 的情况下调用 linux 的 `grep` 命令，实现在多个文件中查找某个模式。

```
:grep vim.ink *
```

在当前目录的所有文件中查找单词 `vim.ink`。

- 虽然只调用了 `:grep vim.ink *`，但 Vim 会自动加入 `-n` 参数指定 `grep` 命令在输出结果中加入行号信息。

140. `vim` 是 `vi` 的扩展，与 `vi` 并不完全兼容性：

- `set compatible`：关闭所有扩展的功能，尽量模拟 `vi` 的行为。但这样就不应用 `vim` 的很多强大功能，所以一般没有什么特殊需要的话（比如执行很老的 `vi` 脚本），都不使用这种模式。
- `set nocompatible`：关闭兼容模式。一般在配置文件第一行使用。

141. `Vundle` 是一个 Vim 插件管理器。

142. `runtimepath` 是 `vim` 的环境变量，用于 `vim` 查找 `scripts`、`syntax files`、`plugins` 等的路径。

```
set rtp+=~/vim/bundle/DrawIt/
```

`rtp` 就是设置 `runtimepath`。

143. 使用 `Vundle` 时，在 `.vimrc` 文件的 `call vundle#begin()` 和 `call vundle#end()` 之间使用 `Plugin` 来添加插件：

```
set nu
set nocompatible
filetype off

set rtp+=~/vim/bundle/Vundle.vim
call vundle#begin()
Plugin 'VundleVim/Vundle.vim'
call vundle#end()
```

```
filetype plugin indent on
```

- 注意，`source` 配置文件时会出错，安装插件不需要调用 `source` 命令。

144. `Plugin` 命令支持不同地址的插件：

- Github 上的插件的格式为：

```
Plugin '用户名/插件名'
```

其 github 地址通常为 `github.com/用户名/插件名`。例如

```
call vundle#begin()
Plugin 'VundleVim/Vundle.vim'
```

```
Plugin 'scrooloose/nerdtree'  
call vundle#end()
```

- 来自 vim scripts 的插件:

```
Plugin '插件名称'
```

例如:

```
Plugin 'L9'
```

- 由 Git 支持但不在 github 上的插件仓库:

```
Plugin 'git clone 后面的地址'
```

例如:

```
Plugin 'git://git.wincent.com/command-t.git'
```

- 本地的 Git 仓库:

```
Plugin 'file:///+本地插件仓库绝对路径'
```

例如:

```
Plugin 'file:///home/gmarik/path/to/plugin'
```

- 插件在仓库的子目录中, 也就是作为子插件存在的插件:

```
Plugin 'rstacruz/sparkup', {'rtp': 'vim/'}
```

插件在 sparkup/vim 目录下。注意要正确设置 runtimepath 目录。

145. 如果已经安装过这个插件, Vundle 可利用以下方式避免命名冲突:

```
Plugin 'ascenator/L9', {'name': 'newL9'}
```

将 L9 重命名为 newL9。

146. Vundle 常用的命令

- ":PluginList": 列出所有已配置的插件。
- ":PluginInstall": 安装插件。要注意的是, 这里的安装插件的原理是从 vimrc 中读取 Plugin 配置内容, 找到想安装的插件, 然后从 github 上下载并安装。
- ":PluginUpdate": 更新插件。
- ":PluginSearch foo": 搜索 foo, 追加 `!` 清除本地缓存。
- ":PluginClean": 清除未使用插件, 需要确认; 追加 `!` 自动批准移除未使用插件

147. nerdtree 是 vim 的文件管理插件, 通过 git 命令安装:

```
git clone https://github.com/scrooloose/nerdtree.git ~/.vim/bundle/nerdtree
```

注意，不能通过网页下载安装，通过网页下载安装的无法运行。

148. nerdtree 命令：

- (1) ":NERDTree": 打开一个新的 NERD 树。树的根取决于参数给出。有 3 种情况：如果没有给出参数，则为当前目录将会被使用。如果给出了一个目录，那么将使用该目录。如果是书签给出名称，将使用相应的目录。

```
:NERDTree /home/marty/vim7/src
```

- (2) ":NERDTreeToggle": 如果 NERD 树已经打开，则重新打开。如果 NERD 树没打开，则打开 NERD 树。
- (3) ":NERDTreeFocus": 如果 NERD 树当前不可见，则打开（或重新打开）；否则，光标移动到已经打开的 NERD 树。
- (4) ":NERDTreeMirror": 在当前选项卡中共享另一个选项卡中的现有 NERD 树。
- (5) ":NERDTreeClose": 关闭此选项卡中的 NERD 树。
- (6) ":NERDTreeCWD": 将 NERDTree 根目录更改为当前工作目录。如果不此选项卡存在 NERDTree，将打开一个新选项卡。

149. nerdtree 切换工作台：

- (1) ctrl + w + h: 光标 focus 左侧树形目录。
- (2) ctrl + w + l: 光标 focus 右侧文件显示窗口。
- (3) ctrl + w + w: 光标自动在左右侧窗口切换。
- (4) ctrl + w + r: 移动当前窗口的布局位置。

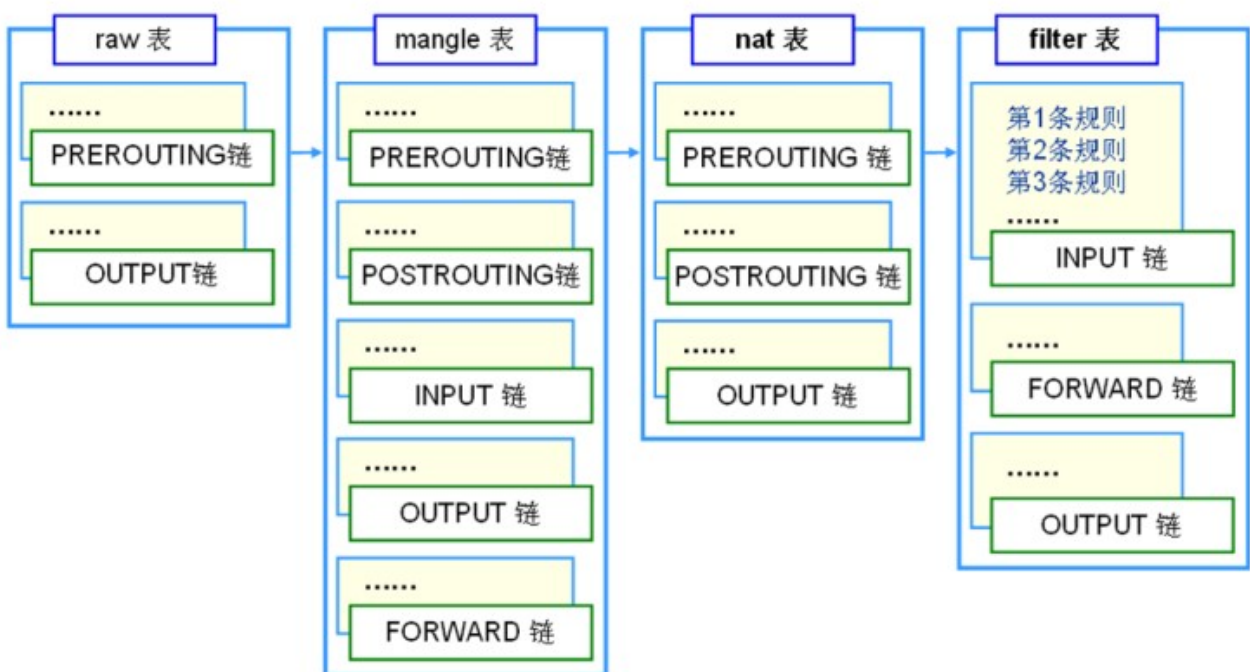
150. nedrdtree 相关命令：

- (1) o: 在已有窗口中打开文件、目录或书签，并跳到该窗口。
- (2) go: 在已有窗口中打开文件、目录或书签，但不跳到该窗口。
- (3) t: 在新 Tab 中打开选中文件/书签，并跳到新 Tab。
- (4) T: 在新 Tab 中打开选中文件/书签，但不跳到新 Tab。
- (5) i: 在当前窗口切分出一个新窗口，打开选中文件，并跳到该窗口。其实就是将右边的窗口一分为二。
- (6) gi: 在当前窗口切分出一个新窗口，打开选中文件，并跳到该窗口。
- (7) !: 执行当前文件。
- (8) O: 递归打开选中 结点下的所有目录。
- (9) m: 文件操作：复制、删除、移动等

151.

Iptables

1. netfilter/iptables 包过滤系统被称为单个实体，但它实际上由两个组件 netfilter 和 iptables 组成。
 - (1) netfilter 组件也称为内核空间，是内核的一部分，由一些信息包过滤表组成，这些表包含内核用来控制信息包过滤处理的规则集。
 - (2) iptables 组件是一种工具，也称为用户空间（userspace），它使插入、修改和删除信息包过滤表中的规则变得容易。
2. 规则其实就是网络管理员预定义的条件。当数据包与规则匹配时，iptables 就根据规则所定义的方法来处理这些数据包，如放行（accept）、拒绝（reject）和丢弃（drop）等。配置防火墙的主要工作就是添加、修改和删除这些规则。
3. 链（chains）是数据包传播的路径，每一条链其实就是众多规则中的一个检查清单，每一条链中可以有一条或数条规则。当一个数据包到达一个链时，iptables 就会从链中第一条规则开始检查，看该数据包是否满足规则所定义的条件。如果满足，系统就会根据该条规则所定义的方法处理该数据包；否则 iptables 将继续检查下一条规则，如果该数据包不符合链中任一条规则，iptables 就会根据该链预先定义的默认策略来处理数据包。（链、数据包传播路径）



4. 表提供特定的功能，iptables 内置了 4 个表，即 raw 表、mangle 表、nat 表和 filter 表，分别用于实现包过滤，网络地址转换和包重构的功能。

(1) RAW 表

优先级最高，可以对收到的数据包在连接跟踪前进行处理。一旦用户使用了 RAW 表，在某个链上，RAW 表处理完后，将不再做地址转换和数据包的链接跟踪处理了。

RAW 表可以应用在那些不需要做 nat 的情况下，以提高性能。如大量访问的 web 服务器，可以让 80 端口不再让 iptables 做数据包的链接跟踪处理，以提高用户的访问速度。

(2) mangle 表

主要用于对指定数据包进行更改。

(3) nat 表

主要用于网络地址转换 NAT，该表可以实现一对一、一对多、多对多等 NAT 工作，iptables 就是使用该表实现共享上网的。

(4) filter 表

主要用于过滤数据包，该表根据系统管理员预定义的一组规则过滤符合条件的数据包。Filter 表是默认的表。

5. 规则表之间的优先顺序：

Raw—>mangle—>nat—>filter

6. iptables 按照封包的处理流程的先后顺序可以分为 5 个链：

(1) PREROUTING：在数据包进入路由表之前处理。

(2) INPUT：对通过路由表后目的地为本机的封包进行处理。

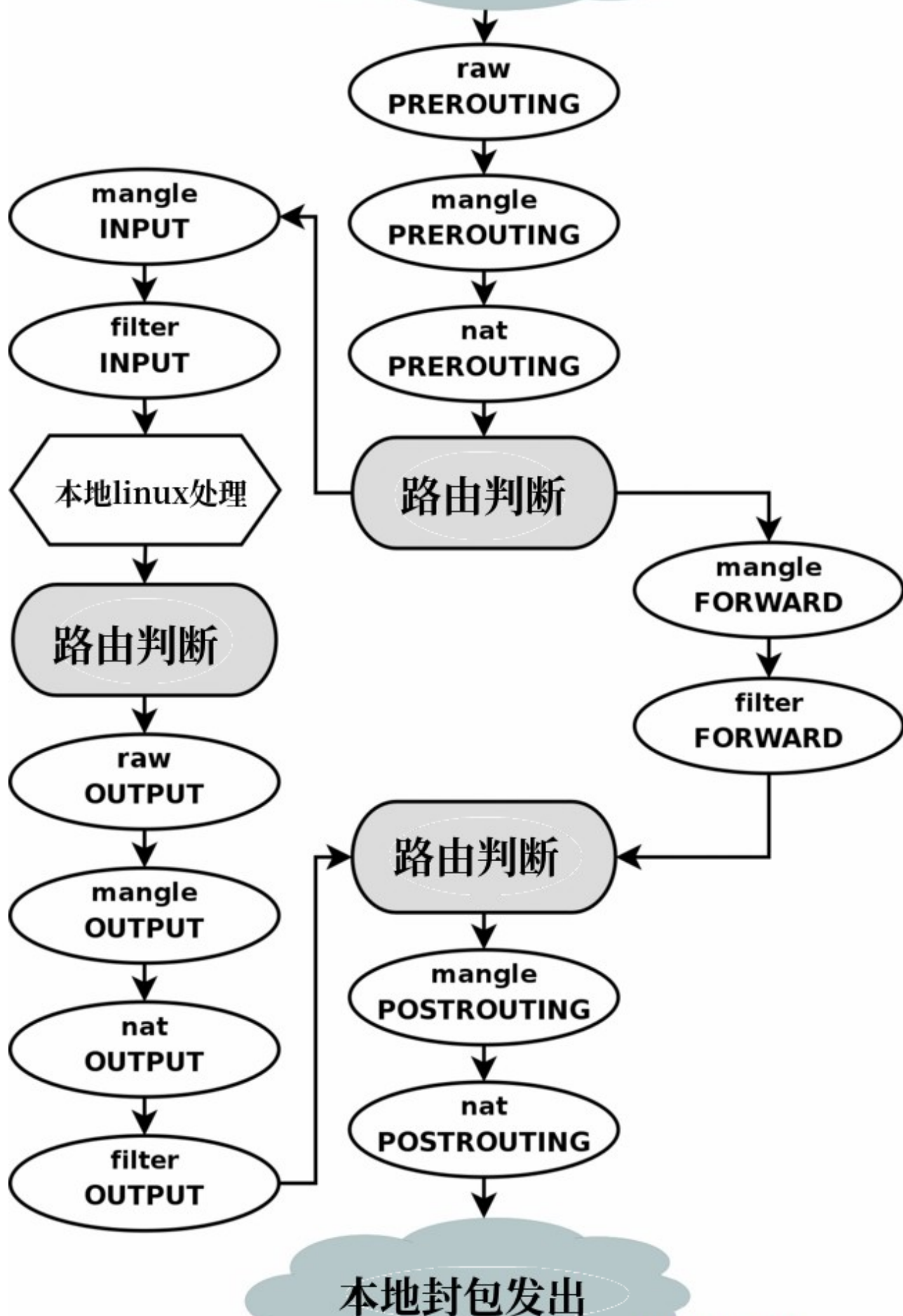
(3) FORWARDING：对通过路由表但目的地不为本机的封包进行处理。

(4) OUTPUT：封包由本机产生，向外转发。

(5) POSTROUTING：封包发送到网卡接口之前进行处理。

7. iptables 四表五链的处理流程图如下：

网络封包传入



8. iptables 的命令格式较为复杂，一般的格式如下：（命令、匹配、操作）

```
iptables [-t 表] 命令 匹配 操作
```

说明：

(1) -t 表

表选项用于指定命令应用于哪个 iptables 内置表。

(2) 命令 （命令大写）

命令选项用于指定 iptables 的执行方式，包括插入规则，删除规则和添加规则，如下所示。

- a) -P: 定义默认策略
- b) -L: 查看 iptables 规则列表
- c) -A: 在规则列表的最后增加 1 条规则
- d) -I: 在指定的位置插入 1 条规则
- e) -D: 从规则列表中删除 1 条规则
- f) -R: 替换规则列表中的某条规则
- g) -F: 删除表中所有规则
- h) -Z: 将表中数据包计数器和流量计数器归零
- i) -X: 删除自定义链
- j) -v: 与-L 他命令一起使用显示更多更详细的信息

(3) 匹配规则

匹配选项指定数据包与规则匹配所具有的特征，包括源地址，目的地址，传输协议和端口号。如下所示：

- a) -i: 指定数据包从哪个网络接口进入。
- b) -o: 指定数据包从哪个网络接口输出。
- c) -p: 指定数据包匹配的协议，如 TCP、UDP 和 ICMP 等。
- d) -s: 指定数据包匹配的源地址。
- e) --sport: 指定数据包匹配的源端口号。
- f) -d: 指定数据包的目的地址。
- g) --dport: 指定数据包匹配的端口号。

9. iptables 的-m 参数用于指定使用的扩展模块，例如 status 模块、multiport 模块等。

```
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
```

允许进入的数据包只能是刚刚我发出去的数据包的回应。

10. iptables 状态类似 TCP 握手包的状态，它有四种状态：

- NEW：表示新连接。
- ESTABLISHED：类似 TCP 的 ESTABLISHED，当数据开始传输时进入该状态。
- RELATED：当一个连接和某个已处于 ESTABLISHED 状态的连接有关系时，就被认为是 RELATED 的了。换句话说，一个连接要想是 RELATED 的，首先要有一个 ESTABLISHED 的连接。这个 ESTABLISHED 连接再产生一个主连接之外的连接，这个新的连接就是 RELATED 的了。
- INVALID：说明数据包不能被识别属于哪个连接或没有任何状态。有几个原因可以产生这种情况，比如，内存溢出，收到不知属于哪个连接的 ICMP 错误信息。一般丢弃这个状态的任何东西，因为防火墙认为这是不安全的東西。

```
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
```

接受 RELATED 状态和 ESTABLISHED 状态的请求。

11.iptables 规则的动作：

- (1) REJECT：拦阻该数据包，并返回数据包通知对方。使用--reject-with 选项可以设置提示信息，告诉对方被拒绝的原因，默认为端口不可达。

```
iptables -A INPUT -p TCP --dport 22 -j REJECT --reject-with icmp-port-unreachable
```

- (2) DROP：丢弃数据包不予处理，进行完此处理动作后，将不再比对其它规则，直接中断过滤程序。
- (3) REDIRECT：将封包重新导向到另一个端口，进行完此处理动作后，将会继续比对其它规则。这个功能可以用来实作透明代理或用来保护 web 服务器。例如：

```
iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-ports 8081
```

- (4) SNAT：改写封包来源 IP 为某特定 IP 或 IP 范围，可以指定 port 对应的范围，进行完此处理动作后，将直接跳往下一个规则链。
- (5) MASQUERADE：改写封包来源 IP 为防火墙的 IP，可以指定 port 对应的范围，进行完此处理动作后，直接跳往下一个规则链（mangle:postrouting）。与 SNAT 略有不同，当进行 IP 伪装时，不需指定要伪装成哪个 IP，IP 会从网卡直接读取，当使用拨接连线时，IP 通常是由 ISP 公司的 DHCP 服务器指派的，这个时候 MASQUERADE 特别有用。范例如下：

```
iptables -t nat -A POSTROUTING -p TCP -j MASQUERADE --to-ports 21000-31000
```

- (6) LOG：将数据包相关信息纪录在 /var/log 中，详细位置请查阅 /etc/syslog.conf 配置文件，进行完此处理动作后，将会继续比对其它规则。例如：（LOG、记录在/var/log 中）

```
iptables -A INPUT -p tcp -j LOG --log-prefix "input packet"
```

- (7) DNAT: 改写数据包目的地 IP 为某特定 IP 或 IP 范围, 可以指定 port 对应的范围, 进行完此处理动作后, 将会直接跳往下一个规则链 (filter:input 或 filter:forward)。范例如下: (DNAT、改写目的 IP)

```
iptables -t nat -A PREROUTING -p tcp -d 15.45.23.67 --dport 80 -j DNAT --to-destination 192.168.10.1-192.168.10.10:80-100
```

- (8) MIRROR: 镜像数据包, 也就是将来源 IP 与目的地 IP 对调后, 将数据包返回, 进行完此处理动作后, 将会中断过滤程序。(MIRROR、对调 IP 后返回)
- (9) QUEUE: 中断过滤程序, 将封包放入队列, 交给其它程序处理。透过自行开发的处理程序, 可以进行其它应用, 例如: 计算联机费用.....等。(QUEUE、放入队列)
- (10) RETURN: 结束在目前规则链中的过滤程序, 返回主规则链继续过滤, 如果把自订规则看成是一个子程序, 那么这个动作, 就相当于提早结束子程序并返回到主程序中。(RETURN、返回)
- (11) MARK: 将封包标上某个代号, 以便提供作为后续过滤的条件判断依据, 进行完此处理动作后, 将会继续比对其它规则。范例如下: (MARK、封包标上代号)

```
iptables -t mangle -A PREROUTING -p tcp --dport 22 -j MARK --set-mark 22
```

12. 当 iptables 不指定端口时, 表示该规则适用于除规则表中已经被设置了的端口之外的所有端口:

```
iptables -A INPUT -j REJECT --reject-with icmp-host-prohibited
```

拒绝所有规则表中的端口之外的所有端口。

- 不指定端口的设置要放在规则表最后, 否则会出问题。

13. 使用 iptables 程序建立的规则只会保存在内存中, 通常我们在修改了 iptables 的规则重启 iptables 后, 之前修改的规则又消失了。对于 RHEL 和 CentOS 系统可以使用 service iptables save 将当前内存中的规则保存到 /etc/sysconfig/iptables 文件中

```
[root@lampbo ~]# service iptables save
```

14. iptables save 命令保存的规则在系统重启之后, 需要执行 iptables restore 来恢复原有规则。

```
# iptables-restore < /etc/sysconfig/iptables
```

15. /etc/sysconfig/iptables 文件是 iptables 保存规则的文件, 里面定义了系统本身的默认规则 and 用户保存的规则:

```
[root@MyServer shadowsocks]# cat /etc/sysconfig/iptables
# Generated by iptables-save v1.4.7 on Wed Mar 6 17:06:22 2019
*filter
:INPUT ACCEPT [285:23141]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [447:37606]
-A INPUT -p tcp -m tcp --dport 58001 -j ACCEPT
-A INPUT -p udp -m udp --dport 58001 -j ACCEPT
-A INPUT -p udp -m udp --dport 58002 -j ACCEPT
-A INPUT -p tcp -m tcp --dport 58002 -j ACCEPT
```

16. 使用 iptables-save 命令可以查询到保存的规则：

```
[root@MyServer shadowsocks]# iptables-save
# Generated by iptables-save v1.4.7 on Wed Mar 6 19:41:25 2019
*filter
:INPUT ACCEPT [12981:3848617]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [11490:3900290]
COMMIT
# Completed on Wed Mar 6 19:41:25 2019
```

17. 修改/etc/sysconfig/iptables-config 将里面的 IPTABLES_SAVE_ON_STOP="no", 这一句的"no"改为"yes"这样每次服务在停止之前会自动将现有的规则保存在 /etc/sysconfig/iptables 这个文件中。

18. iptables 命令-L 后面可以接链名, -t 后接表名:

```
[root@MyServer ~]# iptables -v -n -L OUTPUT
统计 OUTPUT 链的流量统计。
```

19. 查看防火墙当前状态:

```
[root@MyServer ~]# service iptables status
```

20. 使用--line 可以显示规则编号:

```
# iptables -v -n -L OUTPUT --line
Chain OUTPUT (policy ACCEPT 675 packets, 89886 bytes)
num  pkts bytes target    prot opt in     out     source          destination
1    0    0      tcp -- *      *    0.0.0.0/0      0.0.0.0/0      tcp dpt:50002
2    0    0      tcp -- *      *    0.0.0.0/0      0.0.0.0/0      tcp dpt:50001
```

21.使用-D 参数时后接链名，链名后接规则编号，表示删除该链下的指定规则：

```
# iptables -D OUTPUT 2
```

22.iptables 本身会对 INPUT 链等进行流量统计，但只统计总流量，而不会具体到统计某个端口的流量。通过类似下面这样的方式可以开启某个端口的流量统计：

```
# iptables -A INPUT -p tcp --dport 80
# iptables -A OUTPUT -p tcp --sport 80
```

开启 80 的输入流量统计和输出流量统计。

➤ 跟添加一条普通的规则相比，这个没有 ACCEPT、DROP 等规则的动作。

23.使用下列命令查看流量统计值：

```
# iptables -L -v -n
```

其中的参数-v 表示显示完整的信息，以便观察传输量统计的状况，一般不使用-v 时，只会显示如来源地址等简明的信息。

参数-n 设定 iptables 不要将 IP 反查为域名。

流量统计默认以 m 为单位，如果需要以数据包为单位，可以使用-x 参数。

```
# iptables -L -v -n -x
```

24.流量统计值在服务器或 iptables 重启后会丢失，所以稳妥的方法是使用 crontab 任务来定期将流量值写入文件中。

25.使用 iptables 统计流量时，要注意--dport、--sport、INPUT 链和 OUTPUT 链之间的关系。--dport 和 INPUT 结合使用，而--sport 和 OUTPUT 结合使用。而如果--dport 和 OUTPUT 结合使用时根本无法统计流量，因为这种搭配是错的。

26.常用命令示例：

(1) 命令 -A, --append

范例：iptables -A INPUT -p tcp --dport 80 -j ACCEPT

说明：新增规则到 INPUT 规则链中，规则时接到所有目的端口为 80 的数据包的流入连接，该规则将会成为规则链中的最后一条规则。

(2) 命令 -D, --delete

范例：iptables -D INPUT 1

说明：从 INPUT 规则链中删除 1 号规则。

(3) 命令 -R, --replace

范例：iptables -R INPUT 1 -s 192.168.0.1 -j DROP

说明：取代现行第一条规则，规则被取代后并不会改变顺序。

(4) 命令 -I, --insert

范例: `iptables -I INPUT 1 -p tcp --dport 80 -j ACCEPT`

说明: 在第一条规则前插入一条规则, 原本该位置上的规则将会往后移动一个顺位。

(5) 命令 -L, --list

范例: `iptables -L INPUT`

说明: 列出 INPUT 规则链中的所有规则。

(6) 命令 -F, --flush

范例: `iptables -F INPUT`

说明: 删除 INPUT 规则链中的所有规则。

(7) 命令 -Z, --zero

范例: `iptables -Z INPUT`

说明: 将 INPUT 链中的数据包计数器归零。它是计算同一数据包出现次数, 过滤阻断式攻击不可少的工具。

(8) 命令 -N, --new-chain

范例: `iptables -N denied`

说明: 定义新的规则链。

(9) 命令 -X, --delete-chain

范例: `iptables -X denied`

说明: 删除某个规则链。

(10) 命令 -P, --policy

范例: `iptables -P INPUT DROP`

说明: 定义默认的过滤策略。数据包没有找到符合的策略, 则根据此预设方式处理。

(11) 命令 -E, --rename-chain

范例: `iptables -E denied disallowed`

说明: 修改某自订规则链的名称。

27.

Docker 技术入门与实战

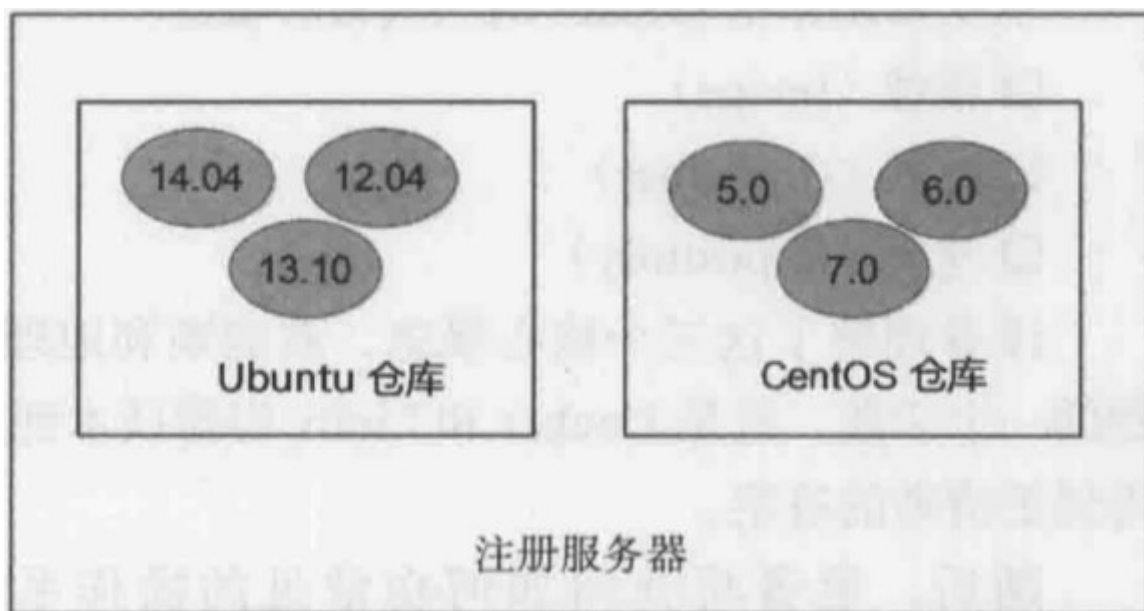
第 1 章 初识 Docker

1. Docker 引擎的基础是 linux 容器技术。

2. Docker 容器是镜像的一个运行实例，所不同的是，它带有额外的可写文件层。每个容器内运行一个应用，不同的容器相互隔离，容器之间也可以建立通信机制。容器的创建和停止都十分快速，对资源的需要也远远低于虚拟机。
3. 容器拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。
4. 容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用数据卷、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主(或网络存储)发生读写，其性能和稳定性更高。
5. 数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。
6. 可以通过 Docker 容器来打包应用，这样服务器迁移时只需要在服务器上启动需要的容器就可以，无需重新搭建 LAMP 之类的环境。
7. Docker 在开发和运维过程中，具有以下几个方面的优势：
 - (1) 更快的交付和部署。使用 Docker，开发人员可以使用镜像来快速构建一套标准的开发环境。
 - (2) 更高效的资源利用。Docker 是内核级的虚拟化，对资源的额外需要很低。
 - (3) 更轻松的迁移和扩展。Docker 可以在任何平台上运行。
 - (4) 更简单的更新管理。

第 2 章 Docker 的核心概念与安装

1. Docker 镜像类似于虚拟机镜像，可以将它理解为一个面向 Docker 引擎的只读模板，包含了文件系统。如只包含 Ubuntu 的镜像，称为 Ubuntu 镜像，而镜像也可以安装 Apache 应用程序，可以称为 Apache 镜像。
2. 镜像创建 Docker 容器的基础。
3. 容器是从镜像创建的应用运行实例，可以将其启动、开始、停止、删除，而这些容器都是相互隔离、相互不可见的。
4. 容器自身是只读的。容器从镜像启动的时候，Docker 会在镜像最上层创建一个可写层，镜像本身保存不变。
5. Docker 仓库类似于代码仓库，是 Docker 集中存放镜像文件的场所，使用不同的标签来区分镜像文件。
6. 注册服务器是存放仓库的地方，其上往往存放着多个仓库。



7. 根据所存储的镜像是否公开，Docker 可以分为公开仓库和私有仓库。最大的公开仓库是 Docker Hub，存放了数量庞大的镜像供用户下载，国内的公开仓库包括 Docker Pool 等。
8. 当用户创建了自己的镜像之后，就可以使用 `docker push` 命令将它上传到指定的公开或私有仓库中。这样用户下次在另外一台机器上使用该镜像时，只需要将其从仓库上 `pull` 下来就可以了。

```
docker push [options] name[:tag]
```

options 说明：

- `--disable-content-trust` :忽略镜像的校验，默认开启

例如，上传本地镜像 `myapache:v1` 到镜像仓库中：

```
docker push myapache:v1
```

第3章 镜像

1. 使用 `docker pull` 命令从网络上下载镜像。语法格式为：

```
Docker pull name[:tag]
```

不指定 tag，则默认选择 latest 标签。例如：

```
$ sudo docker pull ubuntu
```

也可以从指定服务器仓库中下载镜像：

```
$ sudo docker pull dl.dockerpool.com:5000/ubuntu
```

从镜像源 `dl.dockerpool.com` 中下载最新的 `ubuntu` 镜像。

2. `Docker images` 命令可以列出本地主机上已有的镜像。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04	5506de2b643b	1 weeks ago	197.8 MB
dl.dockerpool.com:5000/ubuntu	latest	5506de2b643b	1 weeks ago	197.8 MB

- PEPOSITORY: 来自哪个仓库。
- TAG: 镜像标签。
- IMAGE ID: 镜像 ID, 具有唯一性。
- CREATED: 创建时间。
- VIRTUAL SIZE: 镜像大小。

3. Docker tag 命令可以为本地镜像添加新的标签。例如:

```
root@runoob:~# docker tag ubuntu:15.10 runoob/ubuntu:v3
```

将镜像 ubuntu:15.10 标记为 runoob/ubuntu:v3 镜像。

4. 使用 docker inspect 命令可以获取该镜像的详细信息。该命令返回一个 JSON 格式的消息, 如果只要其中一项内容, 可以使用 -f 参数来指定, -f 后接一个 go 模板。

```
[root@localhost ~]# docker inspect docker.io/ubuntu
[
  {
    "Id": "sha256:ebcd9d4fca80e9e8afc525d8a38e7c56825dfb4a220ed77156f9fb13b14d4ab7",
    "RepoTags": [
      "docker.io/ubuntu:latest"
    ],
    "RepoDigests": [
      "docker.io/ubuntu@sha256:382452f82a8bbd34443b2c727650af46aced0f94a44463c62a9848133ecb1aa8"
    ],
    "Parent": "",
    "Comment": "",
    "Created": "2017-05-15T16:43:41.431635374Z",
    "Container": "5ce5a4b089acb343949a0eb22bc3a546a4a383c377cfd02dba707c3ba3e0c572",
    "ContainerConfig": {
      "Hostname": "b0f1518251f3",
      "Domainname": "",
      "User": "",
      "AttachStdin": false,
      "AttachStdout": false,
      "AttachStderr": false,
      "Tty": false,
      "OpenStdin": false,
      "StdinOnce": false,
      "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
      ],
      "Cmd": [
        "/bin/sh",
        "-c",
        "#(nop) "
      ]
    }
  ]
]
```

<http://blog.csdn.net/xlemonok>

例如:

```
sudo docker inspect -f '{{.Created}}' docker.io/ubuntu
```

显示镜像的创建时间。

5. Go 模板常用语法:

- (1) 系统变量 {{.}}: 点号表示当前对象及上下文, 和 Java、C++ 中的 this 类似。可以直接通过 {{.}} 获取当前对象。

```
sudo docker inspect -f '{{.Created}}' docker.io/ubuntu
```

获取当前模板的创建时间。

- (2) 如果返回结果也是一个 Struct 对象 (Json 中以花括号/大括号包含)，则可以直接通过点号级联调用，获取子对象的指定属性值。

```
sudo docker inspect -f '{{.ContainerConfig.User}}' docker.io/ubuntu
```

6. 使用 `docker search` 命令可以搜索远端仓库中共享的镜像，默认搜索 Docker Hub 官方仓库中的镜像。示例如下：

Docker search term

支持的参数：

- `--automated`：仅显示自动创建的镜像。
- `--no-trunc`：输出信息不截断显示。
- `-s`：`--stars` 的缩写，指定仅显示评价为指定星级以上的镜像。

例如，搜索带 `mysql` 关键字的镜像：

```
$ sudo docker search centos
```

默认是按星级评价进行排序。

```
$ sudo docker search centos
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED	
centos	The official build of CentOS.			465	[OK]
tianon/centos	CentOS 5 and 6, created using rinse instea...	28			
blalor/centos	Bare-bones base CentOS 6.5 image	6			[OK]

Tianon/centos 一行中的表示该镜像属于 tianon 用户的 centos 仓库里的镜像。

```
$ docker search --stars=3 --no-trunc busybox
```

显示名称包含“busybox”的映像，至少 3 颗星，描述不会在输出中截断。

7. 使用 `docker rmi` 命令可以删除镜像，命令格式为：

```
$ docker rmi image
```

image 可以为标签或 ID。例如，要删除

掉 `dl.dockerpool.com:5000/ubuntu:latest` 镜像，可以使用如下命令：

```
$ sudo docker rmi dl.dockerpool.com:5000/ubuntu:latest
```

当同一个镜像拥有多个标签时，`docker rmi` 命令只删除镜像多个标签中的指定标签，并不影响镜像文件。

8. 当镜像只剩下一个时，使用 `docker rmi` 命令会彻底删除该镜像。
9. 使用 `docker rmi` 命令后面跟镜像文件的 ID 或 ID 可区分的一部分前缀时，会先尝试删除所有指向该镜像的标签，再删除该镜像文件本身。
10. 当镜像有创建的容器存在时，镜像文件默认无法删除。如果想强行删除，可以使用 `-f` 参数：

```
$ sudo docker rmi -f ubuntu
```

不推荐使用-f，因为会造成一些问题。

11.删除容器正确的做法是，先删除依赖于该镜像的所有容器，再删除镜像。

12.创建镜像的方法有基于已存在的容器创建、基于本地归档文件导入和基于 Dockerfile 创建三种：

(1) 基于已存在的容器创建：该方法使用 docker commit 命令：

```
Docker commit [options] container[repository[:tag]]
```

主要选项：

- -a: --author 的缩写，即作者信息。
- -m: --message=的缩写，即提交信息。
- -p: --pause 的缩写，默认为 true，提交时暂停容器执行。

例如：

```
$ docker commit -a "runoob.com" -m "my apache" a404c6c174a2 mymysql:v1
```

将容器 a404c6c174a2 保存为镜像 mymysql:v1。顺利的话，命令返回新镜像的 ID 信息。

(2) 基于本地归档文件导入

(3) 基于 Dockerfile 创建。

13.使用 docker import 命令可以从归档文件中创建镜像，也可以从远程 url 中创建镜像。

(1) docker import 语法：

```
docker import [options] file|url|- [repository[:tag]]
```

options 说明：

- -c : 应用 docker 指令创建镜像；
- -m: 提交时的说明文字；

例如：

```
$ docker import my_ubuntu_v3.tar runoob/ubuntu:v4
```

从镜像归档文件 my_ubuntu_v3.tar 创建镜像，命名为 runoob/ubuntu:v4。

```
$ docker import http://example.com/exampleimage.tgz
```

如果需要从远程 url 中创建镜像，url 地址必须是完整的，也就是以 http 或 https 开头。

(2) docker import 使用破折号代表从标准输入中获取数据：

```
$ cat exampleimage.tgz | docker import - exampleimagelocal:new
```

14.使用 docker save 将指定镜像保存成 tar 归档文件。

```
docker save [options] image [image...]
```

options 说明:

- -o: 输出到的文件。

例如:

```
$ docker save -o my_ubuntu_v3.tar runoob/ubuntu:v3
```

从镜像 runoob/ubuntu:v3 中生成 my_ubuntu_v3.tar 文档。

15. docker load 命令用于从归档文件或标准输入中载入镜像到本地镜像库。例如:

```
$ sudo docker load --input ubuntu_14.04.tar
```

从归档文件中载入本地镜像库。

```
$ sudo docker load < ubuntu_14.04.tar
```

从标准输入中载入本地镜像库。

16. 用户在 DockerHub 网站注册后, 即可上传自制的镜像。例如用户 user 上传本地 test:latest 镜像, 可以先添加新的标签 user/test:latest, 然后用 docker push 命令上传镜像:

```
$ sudo docker tag test:latest user/test:latest
```

```
$ sudo docker push user/test:latest
```

需要输入网站登陆信息或进行注册。

第 4 章 容器

1. create 命令创建一个新的容器但不启动它:

```
$ docker create [options] image [command] [arg...]
```

例如, 使用 docker 镜像 nginx:latest 创建一个容器, 并将容器命名为 myrunoob:

```
$ docker create --name myrunoob nginx:latest
```

参数--name 用于为容器指定一个名称。

2. docker start: 启动一个或多个已经被停止的容器, 语法格式如下:

```
docker start [options] container [container...]
```

例如:

```
$ docker start myrunoob
```

启动已被停止的容器 myrunoob。

3. docker stop: 停止一个运行中的容器, 语法格式如下:

```
docker stop [options] container [container...]
```

-t 参数用于指定多少秒之后再停止窗口, 默认值是 10。

```
$ docker stop myrunoob
```

10 秒后停止运行中的容器 myrunoob。

4. docker restart: 重启容器, 语法格式如下:

```
docker restart [options] container [container...]
```

-t 参数用于指定多少秒后重启容器。

```
$ docker restart myrunoob
```

10 秒后重启容器 myrunoob

5. docker run : 创建一个新的容器并运行一个命令。该命令首先在指定的映像上创建一个可写容器层, 然后启动它。

```
docker run [options] image [command] [arg...]
```

options 说明:

- -a stdin: 指定标准输入输出内容类型, 可选 stdin/stdout/stderr 三项;
- -d: 后台运行容器, 并返回容器 id;
- -i: 以交互模式运行容器, 通常与 -t 同时使用;
- -t: 为容器重新分配一个伪输入终端, 通常与 -i 同时使用;
- -p: 将一个指定的主机端口映射到容器中。
- -P: 将一个随机端口映射到容器中。
- -v: 绑定一个数据卷。
- --name: 为容器指定一个名称;
- -w: 设置工作目录。

例如, 使用 docker 镜像 nginx:latest 以后台模式启动一个容器, 并将容器命名为 mynginx:

```
$ docker run --name mynginx -d nginx:latest
```

--name 参数用于指定容器名称, -d 参数表示后台运行容器, 并返回容器 ID。

又如, 使用镜像 nginx:latest 以交互模式启动一个容器, 在容器内执行/bin/bash 命令。

```
$ docker run -it nginx:latest /bin/bash
```

6. 在 Dockerfile 中, 两行 RUN 命令的执行环境根本不同, 是两个完全不同的容器。

```
RUN cd /app
```

```
RUN echo "hello" > world.txt
```

上面只有第一条语句切换了工作目录, 而第二条并没有切换。所以即使 app 目录下有 world.txt, 但 echo 命令的工作目录并不在 app 目录下, 所以可能会出错。

7. 下面的命令则启动一个 bash 终端, 允许用户进行交互。

```
$ sudo docker run -t -i ubuntu:14.04 /bin/bash
```

```
root@af8bae53bdd3:/#
```


在此之后，就可以使用各种 shell 命令对容器进行操作。

8. docker attach 用于将标准输入、标准输出和标准出错连接到正在运行中的容器，以便查看容器输出，或以交互的方式控制它。语法格式如下：

```
docker attach [options] container
```

要 attach 上去的容器必须正在运行，可以同时连接上同一个容器来共享屏幕。例如：

```
$ docker attach mynginx
```

9. docker ps 用于列出容器，语法格式如下：

```
docker ps [options]
```

例如，列出所有在运行的容器信息。

```
$ docker ps
```

10. docker exec 用于在运行的容器中执行命令，语法格式如下：

```
docker exec [options] container command [arg...]
```

options 说明：

- -d: 分离模式: 在后台运行
- -i: 即使没有附加也保持 STDIN 打开
- -t: 分配一个伪终端

例如，在容器 mynginx 中以交互模式执行容器内/root/runoob.sh 脚本：

```
$ docker exec -it mynginx /bin/sh /root/runoob.sh
```

11. docker rm 用于删除一个或多个容器，语法格式如下：

```
docker rm [options] container [container...]
```

options 说明：

- -f: 通过 SIGKILL 信号强制删除一个运行中的容器
- -l: 移除容器间的网络连接，而非容器本身
- -v: 删除与容器关联的卷

例如：

```
docker rm -f db01、db02
```

强制删除容器 db01、db02。

12. docker export 用于将文件系统作为一个 tar 归档文件导出。

```
docker export [options] container
```

options 说明：

- -o: 将输入内容写到文件。

例如：

```
$ docker export -o mysql-`date +%Y%m%d`.tar a404c6c174a2
~$ ls mysql-`date +%Y%m%d`.tar
mysql-20160711.tar
```

将 id 为 a404c6c174a2 的容器按日期保存为 tar 文件。

第 5 章 仓库

1. 对于仓库地址 dl.dockerpool.com/ubuntu 来说，dl.dockerpool.com 是注册服务地址，ubuntu 是仓库名。
2. 目前 Docker 官方维护了一个公共仓库 <https://hub.docker.com>。
3. docker login 命令用于登陆到一个 Docker 镜像仓库，如果未指定镜像仓库地址，默认为官方仓库 Docker Hub，语法如下：

```
docker login [options] [server]
```

options 说明：

- -u：登陆的用户名
- -p：登陆的密码

例如，登陆到 Docker Hub：

```
$ docker login -u 用户名 -p 密码
```

4. docker logout 用于退出一个 Docker 镜像仓库，如果未指定镜像仓库地址，默认为官方仓库 Docker Hub：

```
docker logout [options] [server]
```

options 说明：

- -u：登陆的用户名
- -p：登陆的密码

例如，退出 Docker Hub：

```
$ docker logout
```

5. 自动创建功能使用户通过 Docker Hub 指定一个跟踪的目标网站上的项目，一旦发现新提交，则自动执行创建。
6. 官方提供 registry 镜像来简单搭建一套本地私有仓库环境：

```
$ sudo docker run -d -p 5000:5000 registry
```

这将自动下载并启动一个 registry 容器。-p 参数表示将容器的 5000 端口映射到主机 5000 端口，即仓库端口为 5000。

第6章 数据管理

1. 容器中管理数据主要有两种方式：

- 数据卷
- 数据卷容器

2. 数据卷的使用类似于 linux 下对目录或文件进行 mount 操作。

3. docker run 命令的-v 参数绑定一个数据卷，多次使用-v 标记可以绑定多个数据卷。例如：

```
$ sudo docker run -d -p --name web -v /webapp training/webapp python app.py
```

-v 后面不接冒号时，docker 会选择一个主机目录下新建一个目录挂载到容器的/webapp 下，通常是在/var/lib/docker/volumes 目录下新建一个目录。

4. run 命令的-v 参数是用于挂载数据卷，如果后接 xxx:xxx 形式，表示将主机的文件或文件夹挂载到容器中。

5. Docker 挂载数据卷默认权限是读写，用户可以指定为只读：

```
$ sudo docker run -d -p --name web -v /src/webapp:/opt/webapp:ro training/webapp python app.py
```

加了:ro 之后，容器内挂载的数据卷就无法修改了。

6. Docker run 命令的-v 参数也可以从主机挂载单个文件到容器中作为数据卷：

```
$ sudo docker run --rm -it -v ~/.bash_history:/bash_history ubuntu /bin/bash
```

这个就可以记录在容器输入过的命令历史了。

7. 数据卷容器其实就是一个普通的容器，专门用来提供数据卷供其他容器挂载。例如，使用以下命令创建数据卷容器 dbdata，并在其中创建一个数据卷挂载到容器的/dbdata 下：

```
$ sudo docker run -it -v /dbdata --name dbdata ubuntu
```

8. 数据卷目录：指数据卷容器中的数据卷所在的目录。

```
$ sudo docker run -it -v /dbdata --name dbdata ubuntu
```

这里的数据卷目录就是 dbdata 目录。

9. --volumes-from 参数用于挂载数据卷容器，当主容器挂载数据卷容器时，主容器和数据卷容器都存在着相同的数据卷目录。例如创建 db1 和 db2 两个数据卷，并挂载 dbdata 数据卷容器：

```
$ sudo docker run -it --volumes-from dbdata --name db1 ubuntu
$ sudo docker run -it --volumes-from dbdata --name db2 ubuntu
```

此时，数据卷容器的/dbdata 目录挂载到容器 db1 和 db2 下/dbdata 目录。三个容器任何一方在 dbdata 目录下写入，其他容器都可以看到。

还可以从其他已经挂载了其他数据卷容器的容器来挂载数据卷：

```
$ sudo docker run -d --name db3 --volumes-from db1 training/postgres
```

容器 db3 挂载一个数据卷容器到 db1 中。

10. 数据卷容器是一个普通的 Docker 容器，可以不需要启动。数据卷容器只有一个数据卷，并不是数据卷的仓库。

11. 如果删除了挂载的容器，数据卷不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时显式使用 `docker rm -v` 命令来指定同时删除关联的容器。

12. 使用下面的命令来备份 dbdata 数据卷容器内的数据卷到本地主机当前目录：

```
$ sudo docker run --volumes-from dbdata -v $(pwd):/backup --name worker
ubuntu tar cvf /backup/backup.tar /dbdata
```

这个原理实际上就是使用一个主容器 A 来连接数据卷容器 B，并将数据卷容器中的数据打包，保存到 `/backup` 目录下。因为 A 将当前主机工作目录挂载到 A 的 `/backup` 目录中，所以打包出来存放到 `/backup` 目录中的 `tar` 文件实际上放到了主机的当前工作目录中。

13. 如果要将数据恢复到一个容器，可以按照下面的步骤操作：

(1) 首先创建一个带有数据卷的容器 dbdata2：

```
$ docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

这个数据卷用来存放解压的数据。

(2) 然后创建另一个新的容器，挂载 dbdata2 的容器，并使用 `untar` 解压备份文件到所挂载的容器卷中：

```
$ docker run --volumes-from dbdata2 -v $(pwd):/backup --name worker ubuntu
bash
$ cd /dbdata
$ tar xvf /backup/backup.tar
```

这里的关键是这个 `-v` 参数，表示将主机当前目录映射到 `backup` 目录。

第 7 章 网络基础配置

1. 容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 `-P` 或 `-p` 参数来指定端口映射。当使用 `-P` 标记时，Docker 会随机映射一个 49000~49900 的端口到内部容器开放的网络端口。

2. `docker ps` 的 `-l` 参数可以看到相应的端口映射。

```
$ sudo docker run -d -P training/webapp python app.py
```

```
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
bc533791f3f5	training/webapp:latest	python app.py	5 seconds ago	Up 2 seconds	

```
0.0.0.0:49155->5000/tcp nostalgic_morse
```

3. -p (小写的) 则可以指定要映射的端口, 并且, 在一个指定端口上只可以绑定一个容器。支持的格式有:

- ip:hostPort:containerPort。
- ip::containerPort。
- hostPort:containerPort。

4. 使用 hostPort:containerPort 格式本地的 5000 端口映射到容器的 5000 端口, 可以执行:

```
$ sudo docker run -d -p 5000:5000 training/webapp python app.py
```

5. 可以使用 ip:hostPort:containerPort 格式指定映射使用一个特定地址, 比如 localhost 地址 127.0.0.1:

```
$ sudo docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py
```

6. 使用 ip::containerPort 绑定 localhost 的任意端口到容器的 5000 端口, 本地主机会自动分配一个端口。

```
$ sudo docker run -d -p 127.0.0.1::5000 training/webapp python app.py
```

7. 使用 docker port 来列出指定的容器的端口映射, 语法格式如下:

```
docker port [options] container [private_port[/proto]]
```

示例如下:

```
$ docker port nostalgic_morse 5000  
127.0.0.1:49155.
```

8. 容器有自己的内部网络和 ip 地址, 使用 docker inspect 可以获取所有的变量。

9. -p 标记可以多次使用来绑定多个端口:

```
$ sudo docker run -d -p 5000:5000 -p 3000:80 training/webapp python app.py
```

10. 容器的连接系统是除了端口映射外, 另一种跟容器中应用交互的方式。该系统会在源和接收容器之间创建一个隧道, 接收容器可以看到源容器指定的信息。

11. 连接系统依据容器的名称来执行。因此, 首先需要自定义一个好记的容器命名, 容器的名称是唯一的。

12. 在执行 docker run 的时候如果添加 --rm 标记, 则容器在终止后会立刻删除。注意, --rm 和 -d 参数不能同时使用。

- 13.使用--link 参数可以让容器之间安全的进行交互，格式为--link name:alias，其中 name 是要链接的容器的名称，alias 是这个连接的别名。

```
$ sudo docker run -d -P --name web --link db:db training/webapp python app.py
```

此时，db 容器和 web 容器建立互联关系，web 容器将被允许访问 db 容器的信息。

- 14.使用 env 命令来查看 web 容器的环境变量：

```
$ sudo docker run --rm --name web2 --link db:db training/webapp env
...
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5000_TCP=tcp://172.17.0.5:5432
DB_PORT_5000_TCP_PROTO=tcp
DB_PORT_5000_TCP_PORT=5432
DB_PORT_5000_TCP_ADDR=172.17.0.5
...
```

其中 DB_ 开头的环境变量是供 web 容器连接 db 容器使用，前缀采用大写的连接别名。

- 15.使用--link 建立连接关系后，Docker 添加被连容器的 host 信息到主连容器的 /etc/hosts 的文件。下面是父容器 web 的 hosts 文件：

```
$ sudo docker run -t -i --rm --link db:db training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7 aed84ee21bde
...
172.17.0.5 db
```

这里有 2 个 hosts，第一个是 web 容器，web 容器用 id 作为他的主机名，第二个是 db 容器的 ip 和主机名。可以在 web 容器中安装 ping 命令来测试跟 db 容器的连通。

- 16.C/S 结构，即大家熟知的客户机和服务器结构。B/S 结构即浏览器和服务器结构，普通网站即是 B/S 架构。

第 8 章 使用 Dockerfile 创建镜像

1. 把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是 Dockerfile。
2. FROM 指令用于指定基础镜像。语法格式为：

```
FROM <image>
或
from <image>:<tag>
```

注意，FROM 必须是第一条指令。例如：

FROM ubuntu

3. 除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 scratch。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。对于 Linux 下静态编译的程序来说，并不需要有操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 FROM scratch 会让镜像体积更加小巧。
4. RUN 指令是用来执行命令行命令的。由于命令行的强大能力，RUN 指令在定制镜像时是最常用的指令之一。其格式有两种：
 - shell 格式：

RUN <命令>

例如：

```
RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
```

- exec 格式：

```
RUN ["可执行文件", "参数 1", "参数 2"]
```

这更像是函数调用中的格式。

5. Dockerfile 中每一个指令都会建立一层，RUN 也不例外。新建立一层，在其上执行这些命令，执行结束后，commit 这一层的修改，构成新的镜像。所以在使用 RUN 时要减少 RUN 语句的数量，如下所示：

FROM debian:jessie

```
RUN buildDeps='gcc libc6-dev make' \  
&& apt-get update \  
&& apt-get install -y $buildDeps \  
&& wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz" \  
&& mkdir -p /usr/src/redis \  
&& tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \  
&& make -C /usr/src/redis \  
&& make -C /usr/src/redis install \  
&& rm -rf /var/lib/apt/lists/* \  
&& rm redis.tar.gz \  
&& rm -r /usr/src/redis \  
&& apt-get purge -y --auto-remove $buildDeps  
而不是使用：
```

FROM debian:jessie

```
RUN apt-get update
```

```
RUN apt-get install -y gcc libc6-dev make
```

```
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz"
RUN mkdir -p /usr/src/redis
RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1
RUN make -C /usr/src/redis
RUN make -C /usr/src/redis install
```

6. docker build 命令使用 Dockerfile 创建镜像。

docker build [选项] <上下文路径>

- -t: 以"name: tag"的格式指定镜像的名字和标签。
- -f: 指定 Dockerfile, 注意这里指定的是目录, docker 会在该目录中查找 dockerfile 文件。

示例:

```
$ docker build -t nginx:v3 .
```

使用当前目录的 Dockerfile 创建镜像, v3 后面的点号表示上下文目录为当前目录。

```
$ docker build -t nginx:v3 -f ./config . # 将 Dockerfile 重命名为 config
```

指定 dockerfile 文件。

7. 上下文目录实际上就是和 Docker 引擎进行数据传送的目录, 这个目录是指定的, 通常和 Dockerfile 文件所在的目录是同一个目录。

8. docker build 还支持从 URL 构建, 比如可以直接从 Git repo 中构建:

```
$ docker build https://github.com/twang2218/gitlab-ce-zh.git#:8.14
```

这行命令指定了构建所需的 Git repo, 并且指定默认的 master 分支, 构建目录为 / 8.14/, 然后 Docker 就会自己去 git clone 这个项目、切换到指定分支、并进入到指定目录后开始构建。

9. 如果所给出的 URL 不是个 Git repo, 而是个 tar 压缩包, 那么 Docker 引擎会下载这个包, 并自动解压缩, 以其作为上下文, 开始构建。

10. docker 允许从标准输入中读取上下文压缩包进行构建, 从标准输入中读取数据构建时需要使用破折号-:

```
$ docker build - <context.tar.gz
```

如果发现标准输入的文件格式是 gzip、bzip2 以及 xz 的话, 将会使其为上下文压缩包, 直接将其展开, 将里面视为上下文, 并开始构建。

11. COPY 语句用于复制文件:

```
COPY <src> <dest>
```

COPY 指令将从构建上下文目录中<源路径>的文件/目录复制到新的一层的镜像内的目标路径位置。注意:

- 源路径是指上下文目录或其子目录, 而不能是其祖先目录。
- 目标路径可以是容器内的绝对路径, 也可以是相对于工作目录的相对路径。

- <源路径> 可以是多个，甚至可以是通配符，其通配符规则要满足 Go 的 filepath.Match 规则：

```
COPY hom* /mydir/
```

复制所有 hom 开头的文件到容器的 mydir 目录中。

```
COPY hom?.txt /mydir/
```

复制所有以 hom 开头、以.txt 结尾的文件到容器的 mydir 目录中。

12.ADD 语句也用于复制指定的文件到目标地址中，ADD 语句在 COPY 基础上增加了一些功能。

- ADD 语句会解压压缩文件并把它们添加到镜像中。
- ADD 语句支持从 url 拷贝文件到镜像中。

ADD 语句语法格式和 COPY 类似：

```
ADD <src> <dest>
```

一般情况下，所有的文件复制均使用 COPY 指令，仅在需要自动解压缩的场合使用 ADD。

```
ADD http://example.com/big.tar.xz /usr/src/things/
```

从指定的 url 中下载压缩包解压并复制到/usr/src/things 目录中。

13.CMD 指令用于指定启动容器时执行的语句，每个 Dockfile 只能有一条 CMD 语句。如果指定了多条语句，只有最后一条会被执行。CMD 格式有两种：

- shell 方式：

```
CMD <命令>
```

例如：

```
CMD top
```

- exec 格式：

```
CMD ["可执行文件", "参数 1", "参数 2"...]
```

例如：

```
CMD ["top"]
```

14.使用 shell 模式时，docker 会以 /bin/sh -c "task command" 的方式执行任务命令。也就是说容器中的 1 号进程不是任务进程而是 bash 进程。

15. 以 shell 方式来启动进程时，主进程是 shell，一执行完后 shell 就退出了，依赖于其上的服务也终止了。而以 exec 方式执行程序就不存在这个问题：

```
$ CMD ["nginx", "-g", "daemon off;"]
```

16.ENTRYPOINT 的格式和 RUN 指令格式一样，分为 exec 格式和 shell 格式。

17.ENTRYPOINT 的功能和 CMD 一样，都是在指定容器启动程序及参数。不同之处在于：

- (1) CMD 语句是镜像的默认命令，也就是说，如果在 docker run 时不指定执行命令，容器默认会执行镜像的 CMD 语句指定的指令。
- (2) ENTRYPOINT 是容器的入口。
- (3) 如果 dockerfile 中指定了 ENTRYPOINT 或 run 中指定了容器需要执行的命令，则 CMD 中指定的命令会被覆盖。

```
docker run -it ubuntu /bin/bash
```

这条命令中的/bin/bash 会把 ubuntu 镜像中的 CMD 指定的命令覆盖掉。

18.如果 run 指定的容器需要执行的命令后面有参数，那么后面的参数都会作为 entrypoint 的参数。如果 run 后面没有额外的参数，但是 cmd 有，那么 cmd 的全部内容会作为 entrypoint 的参数：

```
FROM centos
```

```
CMD ["p in cmd"]
```

```
ENTRYPOINT ["echo"]
```

这个 dockerfile 实际上执行的是"echo p in cmd"，所以输出"p in cmd"。

- 通常情况下混合使用 ENTRYPOINT 和 CMD，CMD 提供默认参数，而 ENTRYPOINT 提供命令。

19.ENV 指令用于设置环境变量：

```
ENV <key> <value>
```

```
ENV <key1>=<value1> <key2>=<value2>...
```

例如：

```
ENV TERM xterm
```

设置 TERM 的值为 xterm。

```
ENV abc=bye def=$abc
```

设置 abc 和 def 的值。

20.docker build 指令有一个参数--build-arg=[]允许设置创建镜像时使用的变量。例如：

```
$ docker build --build-arg user=what_user Dockerfile
```

这--build-arg 后面指定的 user 参数我们可以使用 ARG 语句来定义。

21.ARG 语句用于定义构建时需要的参数。语法格式如下：

```
ARG <参数名>[=<默认值>]
```

22. ARG 所设置的构建环境的环境变量，在将来容器运行时是不会存在这些环境变量的。但是不要因此就使用 ARG 保存密码之类的信息，因为 docker history 还是可以看到所有值的。

23. 为了防止运行时用户忘记将动态文件所保存目录挂载为卷，在 Dockerfile 中，我们可以事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

24. VOLUME 用于定义匿名卷，语法格式如下：

```
VOLUME ["<路径 1>", "<路径 2>"...]
```

```
VOLUME <路径>
```

示例如下：

```
VOLUME /data
```

25. EXPOSE 语句是声明运行时容器提供服务端口。语法格式为：

```
EXPOSE <端口 1> [<端口 2>...]
```

这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。

```
EXPOSE 8080
```

```
EXPOSE 22
```

```
EXPOSE 8009
```

```
EXPOSE 8005
```

```
EXPOSE 8443
```

26. 在 Dockerfile 中，每一行的执行环境都不同，例如，下面这条语句会出错：

```
RUN cd /app
```

```
RUN echo "hello" > world.txt
```

语句本意是/app/world.txt，结果是上下文目录下的 world.txt。

27. 在 shell 中使用 cd 来切换目录，而在 docker 中使用 cd 切换目录会出错。

WORKDIR 语句就是用于切换工作目录的，语法格式如下：

```
WORKDIR <工作目录路径>
```

例如：

```
WORKDIR /app
```

将当前工作目录切换到/app 目录下。

28. USER 用于指定当前用户，格式如下：

```
USER <用户名>
```

例如：

```
RUN groupadd -r redis && useradd -r -g redis redis
```

```
USER redis
```

```
RUN [ "redis-server" ]
```

29. HEALTHCHECK 用于判断容器是否正常运行，格式如下：

```
HEALTHCHECK [选项] CMD <命令>          #设置检查容器健康状况的命令
```

HEALTHCHECK 指令是告诉 Docker 应该如何进行判断容器的状态是否正常。

HEALTHCHECK 支持下列选项：

- --interval=<间隔>：两次健康检查的间隔，默认为 30 秒；
- --timeout=<时长>：健康检查命令运行超时时间，如果超过这个时间，本次健康检查就被视为失败，默认 30 秒；
- --retries=<次数>：当连续失败指定次数后，则将容器状态视为 unhealthy，默认 3 次。

示例如下：

```
FROM nginx
```

```
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
```

```
HEALTHCHECK --interval=5s --timeout=3s CMD curl -fs http://localhost/ || exit 1
```

每 5 秒检查 curl 一次容器，如果 curl 命令超过 3 秒没响应就视为失败，使用 exit 退出。

30. 如果基础镜像有健康检查指令，使用这行可以屏蔽掉其健康检查指令：

```
HEALTHCHECK NONE
```

31.

第 15 章 构建 Docker 容器集群

1. Docker 集群最核心的问题就是让不同的主机的 Docker 容器可以相互访问。

(1) 利用端口映射实现容器之间的快速互联。

(2) 使用自定义网桥连接跨主机容器。

(3) 使用 Ambassador 容器解决跨主机的容器互联。

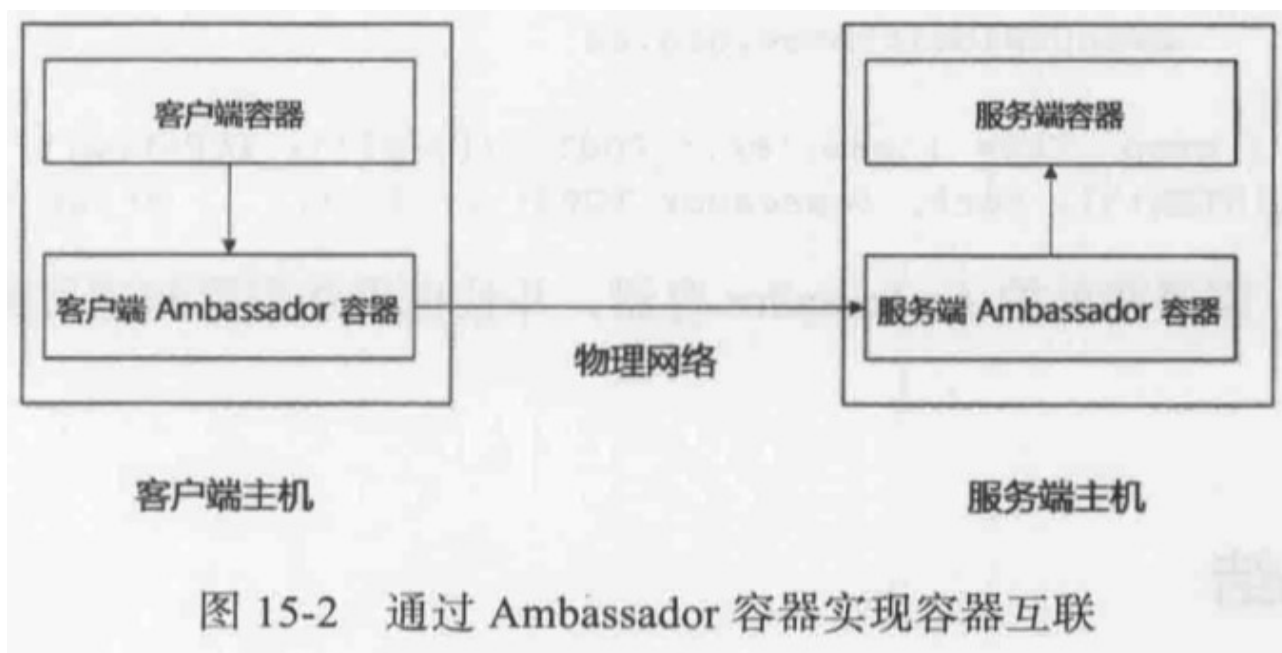
2. 桥接器，又称网桥，一种网络设备，负责网络桥接之用。桥接器将网络的多个网段在数据链路层连接起来（即桥接）。它可以有效地联接两个局域网，它根据 MAC 地址来转发帧，可以看作一个“低层的路由器”。远程网桥通过一个通常较慢的链路（如电话线）连接两个远程 LAN。

3. Docker 服务默认会创建一个 docker0 网桥，其上有一个 docker0 内部接口，它在内核层连通了其他的物理或虚拟网卡，这就将所有容器和本地主机都放到同一个物理网络。

4. linux 使用 `brctl show` 来查看网桥和端口连接信息。

```
$ sudo brctl show
bridge name    bridge id        STP enabled  interfaces
docker0        8000.3a1d7362b4ee  no          veth65f9
```

5. 每次创建一个新容器的时候，Docker 从可用的地址段中选择一个空闲的 IP 地址分配给容器的 `eth0` 端口。使用本地主机上 `docker0` 接口的 IP 作为所有容器的默认网关。
6. 除了默认的 `docker0` 网桥，用户也可以指定网桥来连接各个容器。在启动 Docker 服务的时候，使用 `-b BRIDGE` 或 `--bridge=BRIDGE` 来指定使用的网桥。
7. Ambassador 容器也是一种 Docker 容器，它在内部提供了转发服务。



8. `brctl` 命令用于管理网桥，其常见命令如下：

(1) `brctl addbr` 用于创建网桥：

```
$ sudo brctl addbr br1
创建网桥 br1。
```

(2) `brctl delbr` 用于删除网桥：

```
$ brctl delbr br1
```

(3) `brctl addif` 用于将端口加入网桥：

```
$ sudo brctl addif br1 eth0
```

(4) brctl delif 用于从网桥中删除端口：

```
$ sudo brctl delif br1 eth0
```

(5) brctl show 用于查询网桥信息：

```
$ sudo brctl show br1
```

9. ip link 用于设置 device 的相关设定，参数：

(1) show：仅显示出这个设备的相关内容，如果加上 -s 会显示更多统计数据；

```
ip [-s] link show
```

-s 表示显示更多统计数据。

(2) set：可以开始设置项目，device 指的是 eth0, eth1 等等设备代号；

```
ip link set [device] [动作与参数]
```

其中，动作与参数有以下选项：

- up|down：启动或关闭某个设备。
- address：如果设备可以更改 MAC，用这个参数修改。
- name：给予这个设备一个特殊的名字。
- mtu：设置最大传输单元。

例如，启动 eth0：

```
$ ip link set eth0 up
```

将 MTU 修改为 1000：

```
$ ip link set eth0 mtu 1000
```

10. ip address 用于设置与 IP 相关的内容，有两种。

(1) show：显示 IP 信息。

```
ip address show
```

(2) add 和 del：增加或删除相关内容。

```
ip address [add|del] [IP 参数] [dev 设备名] [相关参数]
```

相关参数：如下所示：

- broadcast：设定广播位址，如果设定值是 + 表示让系统自动计算；
- label：该设备的别名，例如 eth0:0；

示例，删除设备：

```
$ ip address del 192.168.50.50/24 dev eth0
```

11. ip route 用于设置路由的相关内容。

(1) show: 显示 IP 信息。

ip route show

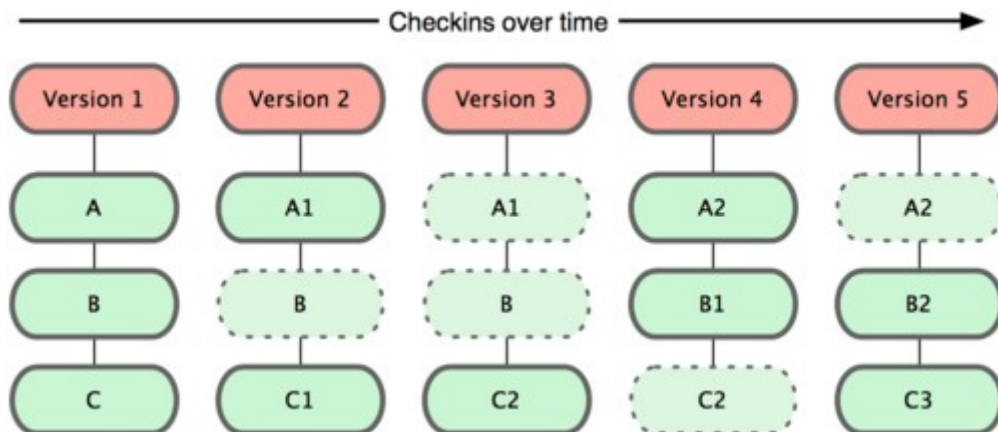
(2) add 和 del: 增加或删除相关内容。

ip route [add|del] [IP 或网域] [via gateway] [dev 设备]

- IP 或网域: 可使用 192.168.50.0/24 之类的网域或者是单纯的 IP ;
- via: 从那个 gateway 出去, 不一定需要;
- dev: 由那个设备连出去, 需要;
- mtu: 可以额外的设定 MTU 的数值;

Github

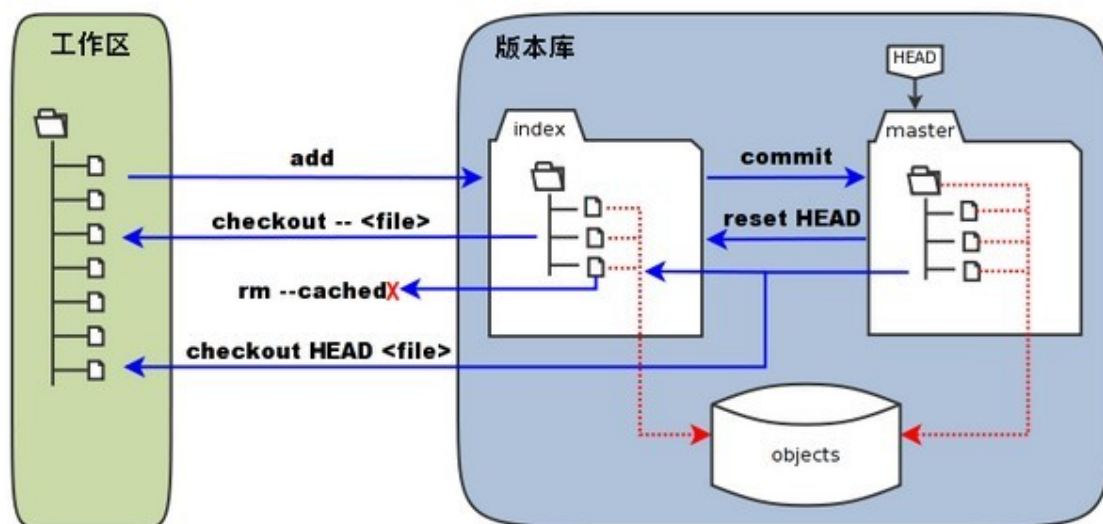
1. Git 是版本控制系统, 能保存一个项目不同的版本。
2. 其他的版本控制系统在每个版本中记录着各个文件的具体差异, 而 Git 使用快照技术来保存不同的版本, 一个版本实际上只是一个快照, 保存着文件的不同状态。例如:



一个项目最初有 A、B、C 三个物理文件, 版本 2 修改了文件 A 和文件 C, 而文件 B 不同, 则版本 2 的快照指向 A1、B 和 C1。版本 3 又修改了文件 C1, 则版本 3 的快照指向 A1、B 和 C2, 依此类推。

3. Git 有工作区、暂存区和版本库的概念:

- 工作区: 就是电脑里能看到的目录。
- 暂存区: 一般存放在 ".git 目录下" 下的 index 文件 (.git/index) 中, 所以我们把暂存区有时也叫作索引。
- 版本库: 工作区有一个隐藏目录 .git, 这个不算工作区, 而是 Git 的版本库。



- 版本库其实就是下面所说的仓库的一种，版本库是远程仓库，还有本地仓库，本地仓库是私有仓库。

4. Git 使用 `git init` 命令来初始化一个 Git 仓库，Git 的很多命令都需要在 Git 的仓库中运行。

```
$ git init
```

`git init` 后面可以接一个目录，表示将指定目录初始化为仓库：

```
$ git init newrepo
```

将 `newrepo` 目录初始化为本地仓库。

5. `git clone`：用于从现有 Git 仓库中拷贝项目，让自己能够查看该项目，或者进行修改。

```
$ git clone git://github.com/schacon/grit.git newpath
```

克隆指定项目到 `newpath` 中，其中 `newpath` 可以省略。如果省略 `newpath`，则新目录名字为 `grit`。

6. `git add` 命令用于将指定文件添加到暂存区。

```
$ git add readme.txt ant.txt
```

`git add` 命令支持将多个文件添加到暂存区，也支持通配符的方式添加到暂存区：

```
$ git add *.html
```

7. `git status` 命令用于显示工作目录和暂存区之间的更改状态。使用此命令能看到那些修改被暂存到了，哪些没有，哪些文件没有被 Git tracked 到。
8. `git diff` 命令用于显示工作区文件和暂存区文件之间的区别，`git status` 是显示哪些文件被修改了，而 `git diff` 则是显示文件的哪行被修改了。

- 尚未缓存的改动: `git diff`。
- 查看已缓存的改动: `git diff --cached`。
- 查看已缓存的与未缓存的所有改动: `git diff HEAD`。
- 显示摘要而非整个 diff: `git diff --stat`。

```
$ git diff --cached
```

9. `git commit`: 将缓存区内容添加到仓库中。
 - `-m`: 添加注释并提交。

```
$ git commit -m '第一次版本提交'
```

- `-a`: 提交那些

10. `git reset HEAD` 命令用于取消已缓存的内容, 一般用于撤消之前的一些操作, 如 `git add`、`git commit` 等。

```
$ git reset HEAD hello.php
```

取消 `hello.php` 的更改。

11. `git rm` 用于从工作区或暂存区删除文件。

- 使用 `git rm` 来删除文件时, 同时还会将这个删除操作记录下来, 提交后就可以删除版本库中的文件; 而使用 `rm` 来删除文件, 仅仅是删除了物理文件, 没有将其从 `git` 的记录中剔除, 提交后版本库中的记录不会被删除。

12. `git mv` 命令用于移动或重命名一个文件、目录、软连接。其实就相当于 `linux` 的 `mv` 命令。

```
$ git mv README README.md
```

13. 分支其实就相当于一份复制, 只是 `git` 的分支并非是一份真正的复制品, `git` 的分支使用快照技术实现。常用的分支有 `master` 分支和 `develop` 分支。

- `master` 分支: 主分支, 正在开发的所有分支都需要合并到该分支中。
- `develop` 分支: 用于日常开发的分支, 往往需要拆分成很多子分支。

14. `git checkout` 命令用于切换分支或恢复工作树文件。例如, 如果发现本地文件删错了, 可以使用这个文件来恢复。

```
$ git checkout hello.c
```

恢复 `hello.c` 文件。

15. `git branch` 命令用于列出、创建或删除分支。

- 不接任何参数为当前有哪些分支:

```
$ git branch
master
* wchar_support
```

有星号的表示为当前分支。后接一个参数-a 表示列出本地和远程分支。

- 后接一个名字为新建一个分支：

```
$ git branch dev2
```

新建分支 dev2,

16.git push 命令用于将本地分支的更新，推送到远程主机。

17.（等待补充）