Linux

cat命令

- 显示文件内容，如果没有文件或文件为`-`则读取标准输入。
- 将多个文件的内容进行连接并打印到标准输出。
- 显示文件内容中的不可见字符（控制字符、换行符、制表符等）。

```
cat file1 从第一个字节开始正向查看文件的内容
tac file1 从最后一行开始反向查看一个文件的内容
cat ./1.log ./2.log ./3.log                # 合并显示多个文件
cat -n file1 标示文件的行数
cat filename | head -n 3000 | tail -n +1000 显示1000行到3000行
cat filename | tail -n +3000 | head -n 1000 从第3000行开始，显示1000(即显示3000~3999行)
```

chmod

用来变更文件或目录的权限

- `u`符号代表当前用户。
- `g`符号代表和当前用户在同一个组的用户，以下简称组用户。
- `o`符号代表其他用户。
- `a`符号代表所有用户。
- `r`符号代表读权限以及八进制数`4`。
- `w`符号代表写权限以及八进制数`2`。
- `x`符号代表执行权限以及八进制数`1`。
- `X`符号代表如果目标文件是可执行文件或目录，可给其设置可执行权限。
- `s`符号代表设置权限suid和sgid，使用权限组合`u+s`设定文件的用户的ID位，`g+s`设置组用户ID位。
- `t`符号代表只有目录或文件的所有者才可以删除目录下的文件。
- `+`符号代表添加目标用户相应的权限。
- `-`符号代表删除目标用户相应的权限。
- `=`符号代表添加目标用户相应的权限，删除未提到的权限。

```
# 当前用户具有所有权限，组用户有读写权限，其他用户只有读权限。
chmod u=rwx, g=rw, o=r ./test.log
# 等价的八进制数表示:
chmod 764 ./test.log
```

chown

用来变更文件或目录的拥有者或所属群组

```
chown -R liu /usr/meng
# 将目录/usr/meng及其下面的所有文件、子目录的文件主改成 liu:
# -R表示递归处理
```

top

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。

常用命令如下：

1. top默认：每隔5秒显式所有进程的资源占用情况
2. P：按%CPU使用率排行
3. M：按%MEM排行
4. T： 根据时间/累计时间进行排序。
5. -d 秒数：每搁几秒显示所有进程的占用资源

scp

用于不同linux主机之间复制文件的

```
scp  local_file    remote_username@remote_ip:remote_file
```

find

找文件名

```
//带有正则表达式的
find . -name '[A-Z]*.txt' -print
```

```
# 当前目录搜索所有文件，文件内容 包含 “140.206.111.111” 的内容
find .  -name "*" | xargs grep "140.206.111.111"
```

sar

参考

`sar`（System Activity Reporter 系统活动情况报告）是目前 Linux 上最为全面的系统性能分析工具之一，可以从多方面对系统的活动进行报告，包括：文件的读写情况、系统调用的使用情况、磁盘 I/O、CPU 效率、内存使用状况、进程活动及 IPC 有关的活动等。我们可以使用`sar`命令来获得整个系统性能的报告。这有助于我们定位系统性能的瓶颈，并且有助于我们找出这些烦人的性能问题的解决方法。

- CPU 利用率

```
sar -u [ <时间间隔> [ <次数> ] ]

sar -u 1 3
Linux 2.6.32-642.13.1.el6.x86_64 (upfor106) 2018年04月25日 _x86_64_ (1 CPU)
10时33分08秒     CPU     %user     %nice   %system   %iowait    %steal     %idle
10时33分09秒     all      0.00      0.00      0.00      0.00      0.00    100.00
10时33分10秒     all      0.99      0.00      0.99      0.00      0.00     98.02
10时33分11秒     all      0.00      0.00      0.00      0.00      0.00    100.00
平均时间:        all      0.33      0.00      0.33      0.00      0.00     99.33

CPU：all 表示统计信息为所有 CPU 的平均值。
%user：显示在用户级别(application)运行使用 CPU 总时间的百分比
%nice：显示在用户级别，用于nice操作，所占用 CPU 总时间的百分比
%system：在核心级别(kernel)运行所使用 CPU 总时间的百分比
%iowait：显示用于等待I/O操作占用 CPU 总时间的百分比
%steal：管理程序(hypervisor)为另一个虚拟进程提供服务而等待虚拟 CPU 的百分比
%idle：显示 CPU 空闲时间占用 CPU 总时间的百分比
    1. 若 %iowait 的值过高，表示硬盘存在I/O瓶颈
    2. 若 %idle 的值高但系统响应慢时，有可能是 CPU 等待分配内存，此时应加大内存容量
    3. 若 %idle 的值持续低于1，则系统的 CPU 处理能力相对较低，表明系统中最需要解决的资源是 CP
```

- 内存利用率

```
sar -r [ <时间间隔> [ <次数> ] ]

sar -r 1 3
Linux 2.6.32-696.13.2.el6.x86_64 (upfor163) 2018年04月25日 _x86_64_ (2 CPU)
10时53分00秒 kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit
10时53分01秒   2027760   2028732     50.01    145492   1243820   1163900     28.69
10时53分02秒   2027620   2028872     50.02    145492   1243820   1163896     28.69
10时53分03秒   2028100   2028392     50.00    145492   1243820   1163900     28.69
平均时间:      2027827   2028665     50.01    145492   1243820   1163899     28.69

kbmemfree：这个值和 free 命令中的 free 值基本一致，所以它不包括 buffer 和 cache 的空间
kbmemused：这个值和 free 命令中的 used 值基本一致,所以它包括 buffer 和 cache 的空间
%memused：这个值是 kbmemused 和内存总量(不包括 swap)的一个百分比
kbbuffers 和 kbcached：这两个值就是 free 命令中的 buffer 和 cache
kbcommit：保证当前系统所需要的内存，即为了确保不溢出而需要的内存(RAM + swap)
%commit：这个值是 kbcommit 与内存总量(包括 swap)的一个百分比
```

- I/O 和传输速率信息状况

```
sar -b [ <时间间隔> [ <次数> ] ]
sar -b 1 3

Linux 2.6.32-696.13.2.el6.x86_64 (upfor163) 2018年04月25日 _x86_64_ (2 CPU)

10时58分15秒       tps      rtps      wtps   bread/s   bwrtn/s
10时58分16秒      7.00      0.00      7.00      0.00     64.00
10时58分17秒      4.04      0.00      4.04      0.00     80.81
10时58分18秒      0.00      0.00      0.00      0.00      0.00
平均时间:         3.67      0.00      3.67      0.00     48.00



tps：每秒钟物理设备的 I/O 传输总量
rtps：每秒钟从物理设备读入的数据总量
wtps：每秒钟向物理设备写入的数据总量
bread/s：每秒钟从物理设备读入的数据量，单位为：块/s
bwrtn/s：每秒钟向物理设备写入的数据量，单位为：块/s
```

- 网络统计信息

```
sar -n <关键词> [ <时间间隔> [ <次数> ] ]

sar -n DEV 1 5
平均时间:      IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst
平均时间:         lo      2.21      2.21      0.18      0.18      0.00      0.00      0.
平均时间:       eth0      4.62      3.82      0.37      1.90      0.00      0.00      0.
```

tar

首先要弄清两个概念：打包和压缩。打包是指将一大堆文件或目录变成一个总的文件；压缩则是将一个大的文件通过一些压缩算法变成一个小文件。这源于Linux中很多压缩程序只能针对一个文件进行压缩，这样当你想要压缩一大堆文件时，你得先将这一大堆文件先打成一个包（tar命令），然后再用压缩程序进行压缩（gzip bzip2命令）。

所以tar就是将一大堆文件打包成一个文件，然后再压缩。

```
tar -zcvf /tmp/bin-backup.tar.gz /home/vivek/bin/
将 /home/vivek/bin/ 目录打包，并使用 gzip 算法压缩。保存为 /tmp/bin-backup.tar.gz 文件。
```

df

用来检查linux服务器的文件系统的磁盘空间占用情况。可以利用该命令来获取硬盘被占用了多少空间，目前还剩下多少空间等信息。默认是KB为单位：使用`-h`选项以KB以上的单位来显示，可读性高：

```
[root@LinServ-1 ~]# df -h
文件系统              容量  已用 可用 已用% 挂载点
/dev/sda2             140G   27G  106G  21% /
/dev/sda1             996M   61M  884M   7% /boot
tmpfs                1009M     0 1009M   0% /dev/shm
/dev/sdb1             2.7T  209G  2.4T   8% /data1
```

free

可以显示当前系统未使用的和已使用的内存数目，还可以显示被内核使用的内存缓冲区。。在Linux系统监控的工具中，free命令是最经常使用的命令之一。

free -m(以MB为单位显示内存使用情况) -k(以KB为单位显示内存使用情况) -g(以GB为单位显示内存使用情况)

```
free -m
             total       used       free     shared    buffers     cached
Mem:          2016       1973         42          0        163       1497
-/+ buffers/cache:        312       1703
Swap:         4094          0       4094
```

netstat

Netstat 命令用于显示各种网络相关信息，如网络连接，路由表，实际的网络连接以及每一个网络接口设备的状态信息。Netstat用于显示与IP、TCP、UDP和ICMP协议相关的统计数据，一般用于检验本机各端口的网络连接情况。

- **直接使用netstat**

```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 2 210.34.6.89:telnet 210.34.6.96:2873 ESTABLISHED
tcp 296 0 210.34.6.89:1165 210.34.6.84:netbios-ssn ESTABLISHED
tcp 0 0 localhost.localdom:9001 localhost.localdom:1162 ESTABLISHED
tcp 0 0 localhost.localdom:1162 localhost.localdom:9001 ESTABLISHED
tcp 0 80 210.34.6.89:1161 210.34.6.10:netbios-ssn CLOSE

Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags Type State I-Node Path
unix 1 [ ] STREAM CONNECTED 16178 @000000dd
unix 1 [ ] STREAM CONNECTED 16176 @000000dc
unix 9 [ ] DGRAM 5292 /dev/log
unix 1 [ ] STREAM CONNECTED 16182 @000000df
```

输出结果可以分为两个部分：

i. Active Internet connections，称为有源TCP连接，其中"Recv-Q"和"Send-Q"指%0A的是接收队列和发送队列。这些数字一般都应该是0。如果不是则表示软件包正在队列中堆积。这种情况只能在非常少的情况见到。

ii. Active UNIX domain sockets，称为有源Unix域套接口(和网络套接字一样，但是只能用于本机通信，性能可以提高一倍)。
Proto显示连接使用的协议,RefCnt表示连接到本套接口上的进程号,Types显示套接口的类型,State显示套接口当前的状态,Path表示连接到套接口的其它进程使用的路径名。

- **列出所有端口 (包括监听和未监听的)**

```
netstat -a     #列出所有端口
netstat -at    #列出所有tcp端口
netstat -au    #列出所有udp端口
```

- **列出所有处于监听状态的 Sockets**

```
netstat -l        #只显示监听端口
netstat -lt       #只列出所有监听 tcp 端口
netstat -lu       #只列出所有监听 udp 端口
netstat -lx       #只列出所有监听 UNIX 端口
```

traceroute

追踪网络数据包的路由途径，预设数据包大小是40Bytes。

```
traceroute www.58.com -m(设置跳数)
traceroute to www.58.com (211.151.111.30), 30 hops max, 40 byte packets
 1  unknown (192.168.2.1)  3.453 ms  3.801 ms  3.937 ms
 2  221.6.45.33 (221.6.45.33)  7.768 ms  7.816 ms  7.840 ms
 3  221.6.0.233 (221.6.0.233)  13.784 ms  13.827 ms 221.6.9.81 (221.6.9.81)  9.758 ms
 4  221.6.2.169 (221.6.2.169)  11.777 ms 122.96.66.13 (122.96.66.13)  34.952 ms 221.6.2.53 (221.6.
 5  219.158.96.149 (219.158.96.149)  39.167 ms  39.210 ms  39.238 ms
 6  123.126.0.194 (123.126.0.194)  37.270 ms 123.126.0.66 (123.126.0.66)  37.163 ms  37.441 ms
 7  124.65.57.26 (124.65.57.26)  42.787 ms  42.799 ms  42.809 ms
 8  61.148.146.210 (61.148.146.210)  30.176 ms 61.148.154.98 (61.148.154.98)  32.613 ms  32.675 ms
 9  202.106.42.102 (202.106.42.102)  44.563 ms  44.600 ms  44.627 ms
10  210.77.139.150 (210.77.139.150)  53.302 ms  53.233 ms  53.032 ms
11  211.151.104.6 (211.151.104.6)  39.585 ms  39.502 ms  39.598 ms
12  211.151.111.30 (211.151.111.30)  35.161 ms  35.938 ms  36.005 ms
```

route

显示并设置Linux内核中的网络路由表，route命令设置的路由主要是静态路由。要实现两个不同的子网之间的通信，需要一台连接两个网络的路由器，或者同时位于两个网络的网关来实现。

可以查看（route），增加（route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0），删除（route del -net 224.0.0.0 netmask 240.0.0.0）

```
[root@localhost ~]# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
112.124.12.0    *               255.255.252.0   U     0      0        0 eth1
10.160.0.0      *               255.255.240.0   U     0      0        0 eth0
192.168.0.0     10.160.15.247   255.255.0.0     UG    0      0        0 eth0
172.16.0.0      10.160.15.247   255.240.0.0     UG    0      0        0 eth0
10.0.0.0        10.160.15.247   255.0.0.0       UG    0      0        0 eth0
default         112.124.15.247  0.0.0.0         UG    0      0        0 eth1

route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0    #增加一条到达224.0.0.0的路由。
route add -net 224.0.0.0 netmask 240.0.0.0 reject     #增加一条屏蔽的路由，目的地址为224.x.x.x将被拒绝
route del -net 224.0.0.0 netmask 240.0.0.0
route del -net 224.0.0.0 netmask 240.0.0.0 reject
route del default gw 192.168.120.240
route add default gw 192.168.120.240
```

ip

linux的ip命令和ifconfig类似，但前者功能更强大，并旨在取代后者。使用ip命令，只需一个命令，你就能很轻松地执行一些网络管理任务。ifconfig是net-tools中已被废弃使用的一个命令，许多年前就已经没有维护了。iproute2套件里提供了许多增强功能的命令，ip命令即是其中之一。

```
ip link show                    # 显示网络接口信息
ip link set eth0 up             # 开启网卡
ip link set eth0 down           # 关闭网卡
ip link set eth0 promisc on     # 开启网卡的混合模式
ip link set eth0 promisc offi   # 关闭网卡的混个模式
ip link set eth0 txqueuelen 1200 # 设置网卡队列长度
ip link set eth0 mtu 1400       # 设置网卡最大传输单元
ip addr show     # 显示网卡IP信息
ip addr add 192.168.0.1/24 dev eth0 # 设置eth0网卡IP地址192.168.0.1
ip addr del 192.168.0.1/24 dev eth0 # 删除eth0网卡IP地址
```

awk

awk 是一种脚本编程语言，用于在linux/unix下对文本和数据进行处理。数据可以来自标准输入(stdin)、一个或多个文件，或其它命令的输出。AWK命令是由三个人的名字首字母命名的--Aho、Weinberger和Kernighan。他们来自AT&T贝尔实验室，也在Unix Shell脚本中贡献了许多其他命令行提示。

它在命令行中使用，但更多是作为脚本来使用。awk有很多内建的功能，比如数组、函数等，这是它和C语言的相同之处，灵活性是awk最大的优势。

AWK命令的基本语法

```
awk [options] [program] [file]
```

awk工作流程是这样的：读入有\n换行符分割的一条记录，然后将记录按指定的域分隔符划分域，填充域，`$0`则表示所有域,`$1`表示第一个域,`$n`表示第n个域。默认域分隔符是"空白键" 或 "[tab]键"，举个例子：

```
[root@www ~]# last -n 5 <==仅取出前五行
root     pts/1    192.168.1.100  Tue Feb 10 11:21   still logged in
root     pts/1    192.168.1.100  Tue Feb 10 00:46 - 02:28  (01:41)
root     pts/1    192.168.1.100  Mon Feb  9 11:41 - 18:30  (06:48)
dmtsai   pts/1    192.168.1.100  Mon Feb  9 11:41 - 11:41  (00:00)
root     tty1                    Fri Sep  5 14:09 - 14:10  (00:01)
```

如果只是显示最近登录的5个帐号

```
#last -n 5 | awk  '{print $1}'
root
root
root
dmtsai
root
```

所以`$1`表示登录用户，`$3`表示登录用户ip,以此类推。

比如下列代码：

```
awk '{printf "第一列：%s 第二列：%s\n",$1,$2}' #在控制台输入两个信息，然后输出
melon water
第一列：melon 第二列：water
```

查看当前进程

```
ps -ef | grep java  # 显示出所有关于java的进程
```

杀死进程

```
kill -9 PID
```

`kill -9` 代表的信号是 `SIGKILL`，表示进程被终止，需要立即退出；表示强制杀死该进程，这个信号不能被捕获也不能被忽略。

查看进程的线程

找到相关进程，然后记录pid

```
pstree -p 5346(进程ID)
top -H 所有线程
top -H -p <pid>
```

如果要计算数量，可以用 `wc -l`，-l表示要计算行数

内核相关命令

uname

```
uname -r  # 查看内核版本
uname -a  #查看系统信息，内核名称，主机名，内核版本，处理器架构，处理器位数啥的
Linux VM-16-12-centos 3.10.0-1160.45.1.el7.x86_64 #1 SMP Wed Oct 13 17:20:51 UTC 2021 x86_64 x86_6
cat /proc/version #跟uname -a一样
```

查看CPU信息

```
cat /proc/cpuinfo
```

依次打印每一个核数的相关信息

查看系统位数

```
getconf LONG_BIT
```

查看Linux版本

```
cat /etc/os-release

NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"
```

查看内核版本

```
cat /proc/version

Linux version 3.10.0-1160.45.1.el7.x86_64 (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5

$arch即可查看linux的内核版本
```

Centos如何查看日志

```
/var/log/message 系统启动后的信息和错误日志，是Red Hat Linux中最常用的日志之一
/var/log/secure 与安全相关的日志信息
/var/log/maillog 与邮件相关的日志信息
/var/log/cron 与定时任务相关的日志信息
/var/log/spooler 与UUCP和news设备相关的日志信息
/var/log/boot.log 守护进程启动和停止相关的日志消息
/var/log/wtmp 该日志文件永久记录每个用户登录、注销及系统的启动、停机的事件
```

1、tail

```
tail  -n  10   test.log   查询日志尾部最后10行的日志;
```

2、head

```
head -n  10  test.log   查询日志文件中的头10行日志;
head -n -10  test.log   查询日志文件除了最后10行的其他所有日志;
```

3、cat

直接查看全部

4、sed

这个命令可以查找日志文件特定的一段 ,可以按照行号和时间范围查询

```
sed -n '5,10p' filename 这样你就可以只查看文件的第5行到第10行。
sed -n '/2014-12-17 16:17:20/,/2014-12-17 16:17:36/p'  test.log
```

Linux各个目录作用

目录	说明
/bin	存放二进制可执行文件(ls,cat,mkdir等)，常用命令一般都在这里。
/home	存放所有用户文件的根目录，是用户主目录的基点，比如用户user的主目录就是/home/us
/usr	用于存放系统应用程序，比较重要的目录 /usr/local 本地系统管理员软件安装目录（安装系统级的应用）。这是最庞大的目录，要用到的应用 众多的应用程序 /usr/sbin 超级用户的一些管理程序 /usr/doc linux文档 /usr/include linux下 常用的动态链接库和软件包的配置文件 /usr/man 帮助文档 /usr/src 源代码，linux内核的源 本地增加的库
/opt	额外安装的可选应用程序包所放置的位置。一般情况下，我们可以把tomcat等都安装到这
/proc	虚拟文件系统目录，是系统内存的映射。可直接访问这个目录来获取系统信息。
/root	超级用户（系统管理员）的主目录（特权阶级o）
/sbin	存放二进制可执行文件，只有root才能访问。这里存放的是系统管理员使用的系统级别的管
/dev	用于存放设备文件。
/mnt	系统管理员安装临时文件系统的安装点，系统提供这个目录是让用户临时挂载其他的文件
/boot	存放用于系统引导时使用的各种文件
/lib	存放跟文件系统中的程序运行所需要的共享库及内核模块。共享库又叫动态链接共享库，
/tmp	用于存放各种临时文件，是公用的临时文件存储点。
/var	用于存放运行时需要改变数据的文件，也是某些大文件的溢出区，比方说各种服务的日志
/lost+found	这个目录平时是空的，系统非正常关机而留下“无家可归”的文件（windows下叫什么.chk）

Python

Python PEP-8编码规范

一、代码编排

1. 缩进。4个空格的缩进（编辑器都可以完成此功能）
2. 每行最大长度79，换行可以使用反斜杠，最好使用圆括号。换行点要在操作符的后边敲回车。
3. 类和top-level函数定义之间空两行；
   类中的方法定义之间空一行；
   函数内逻辑无关段落之间空一行；
   其他地方尽量不要再空行。

二、import

1. 不要在一句import中多个库，比如import os, sys不推荐
2. 如果采用from XX import XX引用库，可以省略‘module.’

三、空格

1. 逗号、冒号、分号前不要加空格。
2. 操作符左右各加一个空格，不要为了对齐增加空格。
3. 函数的左括号前不要加空格。如Func(1)。
4. 序列的左括号前不要加空格。如list[2]。
5. 函数默认参数使用的赋值符左右省略空格。
6. 不要将多句语句写在同一行，尽管使用‘；’允许。
7. if/for/while语句中，即使执行语句只有一句，也必须另起一行。

四、命名

1. 类的属性若与关键字名字冲突，后缀一下划线，尽量不要使用缩略等其他方式。
2. 函数命名使用全部小写的方式，可以使用下划线。
3. 常量命名使用全部大写的方式，可以使用下划线。
4. 类的命名使用CapWords的方式，模块内部使用的类采用_CapWords的方式。

5. 类的属性有3种作用域public、non-public和subclass API，可以理解成C++中的public、private、protected，non-public属性前，前缀一条下划线。

内存管理

引用计数、垃圾回收、内存池机制三大块

一、变量与对象

1、变量，通过变量指针引用对象。变量指针指向具体对象的内存空间，取对象的值。

2、对象，类型已知，每个对象都包含一个头部信息（头部信息：类型标识符和引用计数器）

变量名没有类型，类型属于对象（因为变量引用对象，所以类型随对象），变量引用什么类型的对象，变量就是什么类型的。

id()是python的内置函数，用于返回对象的身份，即对象的内存地址。

通过is进行引用所指判断，is是用来判断两个引用所指的对象是否相同。

二、引用计数

引用计数：在Python中，每个对象都有指向该对象的引用总数

普通引用

当使用某个引用作为参数，传递给getrefcount()时，参数实际上创建了一个临时的引用。

容器对象

容器对象中包含的并不是元素对象本身，是指向各个元素对象的引用。

引用计数增加

1. 对象被创建
2. 引用对象被创建
3. 作为容器对象的一个元素

引用计数减少

1. 对象的别名被显式的销毁, `del m`
2. 对象的一个别名被赋值给其他对象
3. 对象从一个容器对象中移除，或容器对象本身被销毁
4. 一个本地引用离开了它的作用域，比如上面的foo(x)函数结束时，x指向的对象引用减1。

三、垃圾回收

当Python中的对象越来越多，占据越来越大的内存，启动垃圾回收(garbage collection)，将没用的对象清除。

原理：

当Python的某个对象的引用计数降为0时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾。比如某个新建对象，被分配给某个引用，对象的引用计数变为1。如果引用被删除，对象的引用计数为0，那么该对象就可以被垃圾回收。

注意：

1. 垃圾回收时，Python不能进行其它的任务，频繁的垃圾回收将大大降低Python的工作效率；
2. Python只会在特定条件下，自动启动垃圾回收（垃圾对象少就没必要回收）
3. 当Python运行时，会记录其中分配对象(object allocation)和取消分配对象(object deallocation)的次数。当两者的差值高于某个阈值时，垃圾回收才会启动。

分代回收

Python将所有的对象分为0，1，2三代；所有的新建对象都是0代对象；当某一代对象经历过垃圾回收，依然存活，就被归入下一代对象。

四、内存池

Python中有分为大内存和小内存：（256K为界限分大小内存）

1. 大内存使用malloc进行分配
2. 小内存使用内存池进行分配
3. Python的内存池(金字塔)如下：

第3层：最上层，用户对Python对象的直接操作
第1层和第2层：内存池，有Python的接口函数PyMem_Malloc实现-----若请求分配的内存在1~256字节之间就使用内存池管理系统进行分配，调用malloc函数分配内存，但是每次只会分配一块大小为256K的大块内存，不会调用free函数释放内存，将该内存块留在内存池中以便下次使用。
第0层：大内存-----若请求分配的内存大于256K，malloc函数分配内存，free函数释放内存。
第-1，-2层：操作系统进行操作

Python中的数据结构

一、序列

列表、元祖、字符串

可以“抽象”出序列的一些公共通用方法（不是你想像中的CRUD），这些操作包括：索引（indexing）、分片（sliceing）、加（adding）、乘（multiplying）以及检查某个元素是否属于序列的成员。除此之外，还有计算序列长度、最大最小元素等内置函数。

索引：就是根据index找到对应的值

分片：就是访问一定范围内的元素

是否属于序列的成员

```
str1='Hello'
print 'h' in str1
print 'H' in str1
num1=[1,2]
print 1 in num1
```

长度、最大最小值：获取容器的长度

列表

列表是可变的数组，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

元祖

元组与列表一样，也是一种序列，唯一不同的是元组不能被修改

a、逗号分隔一些值，元组自动创建完成；

b、元组大部分时候是通过圆括号括起来的；

c、空元组可以用没有包含内容的圆括号来表示；

字符串

一串字符组成的序列，都一样的

二、哈希表映射

就是散列表

字典的键可以是数字、字符串或者是元组，键必须唯一。在Python中，数字、字符串和元组都被设计成不可变类型，而常见的列表以及集合（set）都是可变的，所以列表和集合不能作为字典的键。键可以为任何不可变类型，这正是Python中的字典最强大的地方。

序号	函数及描述
1	dict.clear() 删除字典内所有元素
2	dict.copy() 返回一个字典的浅复制
3	dict.fromkeys() 创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值
4	dict.get(key, default=None) 返回指定键的值，如果键不在字典中返回 default 设置的默认值
5	key in dict 如果键在字典dict里返回true，否则返回false
6	dict.items() 以列表返回一个视图对象
7	dict.keys() 返回一个视图对象
8	dict.setdefault(key, default=None) 和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default

序号	函数及描述
9	dict.update(dict2) 把字典dict2的键/值对更新到dict里
10	dict.values() 返回一个视图对象
11	[pop(key,default]) 删除字典 key（键）所对应的值，返回被删除的值。
12	popitem() 返回并删除字典中的最后一对键和值。

三、集合

集合就是由序列（或者其他可迭代的对象）构建的，使用大括号 **{ }** 或者 **set()** 函数创建集合

add()	为集合添加元素
clear()	移除集合中的所有元素
copy()	拷贝一个集合
difference()	返回多个集合的差集
difference_update()	移除集合中的元素，该元素在指定的集合也存在。
discard()	删除集合中指定的元素
intersection()	返回集合的交集
intersection_update()	返回集合的交集。
isdisjoint()	判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 F
issubset()	判断指定集合是否为该方法参数集合的子集。
issuperset()	判断该方法的参数集合是否为指定集合的子集
pop()	随机移除元素
remove()	移除指定元素
symmetric_difference()	返回两个集合中不重复的元素集合。
symmetric_difference_update()	移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定

add()	为集合添加元素
union()	返回两个集合的并集
update()	给集合添加元素

解释//、%、**运算符？

//(Floor Division)-这是一个除法运算符，它返回除法的整数部分。

%(模数)-返回除法的余数。

`**`(幂)-它对运算符执行指数计算。`a ** b`表示a的b次方。

Python中的单引号和双引号有什么区别？

在Python中使用单引号(' ')或双引号(" ")是没有区别的，都可以用来表示一个字符串。

这两种通用的表达方式，除了可以简化程序员的开发，避免出错之外，还有一种好处，就是单引号可以减少转义字符的使用，使程序看起来更简洁，更清晰。

Python中append，insert和extend的区别?

append：在列表末尾添加新元素。

insert：在列表的特定位置添加元素。

extend：通过添加新列表来扩展列表。

区分Python中的remove，del和pop？

remove：将删除列表中的第一个匹配值，它以值作为参数。

del：使用索引删除元素，它不返回任何值。

pop：将删除列表中顶部的元素，并返回列表的顶部元素。

如何更改列表的数据类型？

要将列表的数据类型进行更改，可以使用tuple()或者set()。

```
lst = [1,2,3,4,2]# 更改为集合set(lst)    ## {1,2,3,4}# 更改为元组tuple(lst)  ## (1,2,3,4,2)
```

Python是否有main函数？

是的，它有的。只要我们运行Python脚本，它就会自动执行。

你对Python类中的self有什么了解？

self表示类的实例。

通过使用self关键字，我们可以在Python中访问类的属性和方法。

在类的函数当中，必须使用self，因为类中没有用于声明变量的显式语法。

`_init_`在Python中有什么用？

`__init__`是Python类中的保留方法。

它被称为构造函数，每当执行代码时都会自动调用它，它主要用于初始化类的所有变量。

Python中的装饰器是什么？

装饰器的作用有两个：扩展功能和扩展权限认证

扩展功能

```
def 孙悟空():
  print('吃桃子')
孙悟空()
# 输出:吃桃子


def 炼丹炉(func): # func就是‘孙悟空’这个函数
  def 变身(*args, **kwargs): #*args, **kwargs就是‘孙悟空’的参数列表，这里的‘孙悟空’函数没有传参数，我们
      print('有火眼金睛了') # 加特效，增加新功能，比如孙悟空的进了炼丹炉后，有了火眼金睛技能
      return func(*args, **kwargs) #保留原来的功能，原来孙悟空的技能，如吃桃子
  return 变身 # 炼丹成功，更强大的，有了火眼金睛技能的孙悟空出世

@炼丹炉
def 孙悟空():
  print('吃桃子')

孙悟空()
# 输出: 有火眼金睛了 吃桃子
```

权限认证

```
def play():
  print('开始播放动画片 《喜洋洋和灰太狼》')

play()
# 输出:开始播放动画片 《喜洋洋和灰太狼》

userAge = 40
def canYou(func):
  def decorator(*args, **kwargs):
      if userAge > 1 and userAge < 10:
          return func(*args, **kwargs)
      print('你的年龄不符合要求，不能看')
  return decorator

@canYou
def play():
  print('开始播放动画片 《喜洋洋和灰太狼》')

play()
# 输出:你的年龄不符合要求，不能看 你可以修改上面的 userAge 为9 试试
```

使用正则表达式

导入

`import re`

findsearch

`search(pattern, string, flags=0)`

- 扫描整个string并返回匹配pattern的结果(None或对象)
- 有匹配的字符串的话返回一个对象(包含符合匹配条件的第一个字符串),否则返回None

```python
import re
#导入正则库

content = 'Hello 1234567 Hello 666'
#要匹配的文本
res = 'Hello\s'
#正则字符串

result = re.search(res, content)
if result is not None:
    print(result.group())
    #输出匹配得到的字符串 'hello'(返回的得是第一个'hello')

print(result.span())
#输出输出匹配的范围(匹配到的字符串在原字符串中的位置的范围)

res1 = 'Hello\s(\d)(\d+)'
result = re.search(res1, content)
print(result.group(1))
#group(1)表示匹配到的第一个组(即正则字符串中的第一个括号)的内容
print(result.group(2))
```

findall

`findall(pattern, string, flags=0)`

- 扫描整个context并返回匹配res的结果(None或列表)
- 有匹配的字符串的话返回一个列表(符合匹配条件的每个子字符串作为它的一个元素),否则返回None

```
import re

res = 'Hello\s'
results = re.findall(res, content)
if results is not None:
   print(results)
   #输出: ['hello','hello']

res1 = 'Hello\s(\d)(\d+)'
results = re.findall(res1, content)
if result is not None:
    print(results)
    # 当正则字符串中出现括号时,所得到列表的每个元素是元组
    # 每个元组的元素都是依次匹配到的括号内的表达式的结果
    #输出: [('1','1234567'),('6','666')]
```

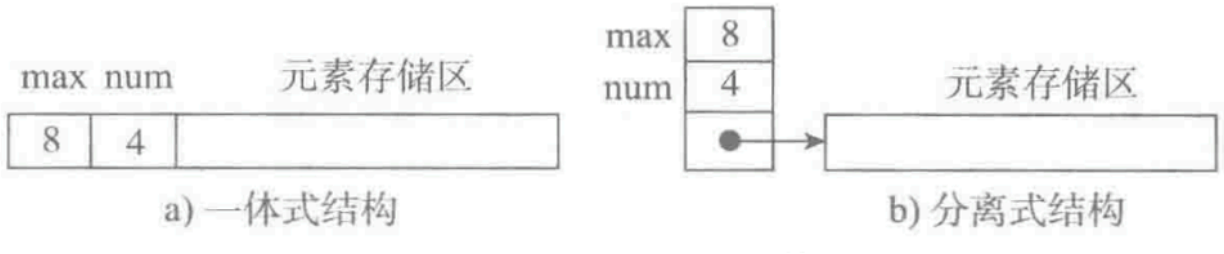
Python的列表(List)的底层实现原理

参考链接

1个字节有8位，内存寻址的最小单位就是字节。假设我们有一个int类型的值，它从0x10开始，一个int占据4个字节，则其结束于0x13。相同元素存储在一起非常有优势，因为相同元素他们的字节数是一样的，去元素的时候首元素地址+偏移量就可以找到对应索引的位置。

但是当不同的元素要挤在一个集合里时，用偏移量来定位就靠不住了，毕竟各自大小都不同。但不要忘了，内存地址本身的大小是固定的。假设集合里有12, 1.24, 'ab' 三种不同的元素，它们的位置各不连续，分散在不同的地方。就申请一块3个元素大小的连续内存区域，里面每个元素都分别指向集合内的元素。

顺序表在内存中有两种结构



存储表信息的单元与元素存储区以连续的方式安排在一块存储区里，两部分数据的整体形成一个完整的顺序表对象。一体式结构整体性强，易于管理。但是由于数据元素存储区域是表对象的一部分，顺序表创建后，元素存储区就固定了。分离式结构中表对象里只保存与整个表有关的信息（即容量和元素个数），实际数据元素存放在另一个独立的元素存储区里，通过链接与基本表对象关联。一旦表需要扩充，对于一体式结构来说，就要重新申请一块更大的空内存区域，将所有元素放入其中，再清空旧的内存区域。对于分离式结构来说，则需要将链接地址更新一下，顺序表对象是不变的。

list有以下几个特点：

1. 元素有位置下标，以索引就可以直接取到元素 --> 连续的存储空间，以偏移量计算取得元素，不必遍历所有元素
2. 元素无论如何改变，表对象不变，也就是其id不变 --> 分离式结构，表头和元素内容分开储存，这样在更改list时，表对象始终是同一个，只是其指向的地址不同
3. 元素可以是任意类型 --> 既要要求是连续存储，又可以存储不同类型的数据，那么其用的就是元素外置的方式，存储的只是地址的引用
4. 可以任意添加新元素 --> 要能不断地添加新元素，其使用了动态扩充的策略

从实现上来讲，在python中创建空ist时，会申请一个8个元素大小的内存区域。以后如果满了，就扩容4倍，且当元素总数达到50000时，再扩容就改为2倍。

Shell

awk命令

awk用于在linux/unix下对文本和数据进行处理。它处理的数据可以来自标准输入(stdin)、一个或多个文件，或其它命令的输出等。它支持用户自定义函数和动态正则表达式等先进功能，是linux/unix下的一个强大编程工具。它在命令行中使用，但更多是作为脚本来使用。

awk工作流程是这样的：读入有\n换行符分割的一条记录，然后将记录按指定的域分隔符划分域，填充域，`$0`则表示所有域,`$1`表示第一个域,`$n`表示第n个域。默认域分隔符是"空白键" 或 "[tab]键"，举个例子：

```
[root@www ~]# last -n 5 <==仅取出前五行
root     pts/1    192.168.1.100  Tue Feb 10 11:21   still logged in
root     pts/1    192.168.1.100  Tue Feb 10 00:46 - 02:28  (01:41)
root     pts/1    192.168.1.100  Mon Feb  9 11:41 - 18:30  (06:48)
dmtsai   pts/1    192.168.1.100  Mon Feb  9 11:41 - 11:41  (00:00)
root     tty1                    Fri Sep  5 14:09 - 14:10  (00:01)
```

如果只是显示最近登录的5个帐号

```
#last -n 5 | awk  '{print $1}'
root
root
root
dmtsai
root
```

所以`$1`表示登录用户，`$3`表示登录用户ip,以此类推。

比如下列代码：

```
awk '{printf "第一列: %s 第二列: %s\n",$1,$2}' #在控制台输入两个信息，然后输出
melon water
第一列: melon 第二列: water
```

1

一些语法

1. `$n`表示接收的命令参数，`$1`表示第一个参数。。。
2. x表示空
3. export会设置环境变量
4. |（管道运算符），例如command 1 | command 2

表示把第一个命令的command1执行的结果座位command2的输入传给command2

5. ipcalc -p ip地址 掩码，输出PREFIX=
   这个命令是显示给定的掩码或ip地址的前缀
6. shell中的print是ksh的内置命令，print不能使用%s,%d等格式转换符，但是可以自动换行
   而printf可以使用格式转换符但是不能自动换行。
7. -a 与
   -o 或
   ！非
8. -eq //equal 等于
   -ne //no equal 不等于
   -gt //great than 大于
   -lt // low than 小于
   ge // great and equal 大于等于，注意没有"-"
   le //low and equal 小于等于，注意没有“-”
9. shell 中利用 -n 来判定字符串非空
   例子：

```
if [ -n str1 ]
//当串的长度大于0时为真(串非空)
```

11. shell中的特殊变量

`$0`, `$#`, `$*`, `$@`, `$?`, `$$`和命令行参数

变量	含义
`$0`	当前脚本的文件名
`$n`	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是$1，第二个参数是$2。
`$#`	传递给脚本或函数的参数个数。
`$*`	传递给脚本或函数的所有参数。
`$@`	传递给脚本或函数的所有参数。被双引号(" ")包含时，与 $* 稍有不同，下面将会讲到。
`$?`	上个命令的退出状态，或函数的返回值。
`$$`	当前Shell进程本身的PID即进程ID。就是正在运行的脚本的进程ID
`$!`	shell最后运行的后台Process的PID，可调用上一个进程的进程ID
`$var`	使用一个定义过的变量var
`${var}`	变量，界定范围
`$()`	与``(反引号)类似，里面执行完再返回值，所有shell通用
`$[]`	可进行算术运算和逻辑运算，不支持浮点和字符串
`$(()) `	可进行算术运算和逻辑运算，不支持浮点和字符串。里面的变量可以省略$

12. Shell 脚本中的exit状态解释

如果退出状态为0，则命令执行成功。`exit 0`

如果命令失败，则退出状态为非零。`exit 1`

13. shit命令

shit一般和循环一起使用。对于命令行参数，其个数必须是确定的，或者当 Shell 程序不知道其个数时，可以把所有参

```shell
until [ $# -eq 0 ]
do
echo "第一个参数为: $1 参数个数为: $#"
shift
done
```

执行以上程序x_shift.sh:

$./x_shift.sh 1 2 3 4，结果如下：

第一个参数为：1 参数个数为：4

第一个参数为：2 参数个数为：3

第一个参数为：3 参数个数为：2

第一个参数为：4 参数个数为：1

14. tee命令

在输出信息的同时把信息记录到文件中

`ls | tee ls.txt` 将会在终端上显示ls命令的执行结果，并把执行结果输出到ls.txt文件中，将会覆盖原文件的内容

`ls | tee -a ls.txt` 保留ls.txt文件中原来的内容，并把ls命令的执行结果追加到ls.txt文件的最后，不覆盖原来

15. WC命令

wc命令是用来统计文件的行数、字节数等。其用法如下：

> wc [-lcmw] [file...]
>
> -l 统计行（line）数
>
> -w 统计字（word）数
>
> -c 统计字节（character）数

正则表达式 {#正则表达式 }

正则表达式	描述	
`^`	指定了匹配正则表达式的文本必须起始于字符串的首部	`^tux`能够
`$`	指定了匹配正则表达式的文本必须结束于目标字符串的尾部	`tux$`能够
`A`字符	正则表达式必须匹配该字符	
`.`	匹配任意一个字符	`Hack.`能够匹配`Hackl`
`[]`	匹配中括号内的任意一个字符。中括号内可以是一个字符组或字符范围	`coo[kl]`能够匹配`cook

| `[^]` | 匹配不在中括号内的任意一个字符。中括号内可以是一个字符组或字符范围 | `9[^01]`能够匹配`92`和

例子

能够匹配任意单词的正则表达式：

```shell
( +[a-zA-Z]+ +)
```

开头的`+`表示需要匹配一个或多个空格。字符组`[a-zA-Z]`用于匹配所有的大小写字母。随后的`+`表示至少要匹配一个字母，多者不限。最后的`+`表示需要匹配一个或多个空格来终结单词。

grep常用用法

```
[root@www ~]# grep [-acinv] [--color=auto] '搜寻字符串' filename
选项与参数:
-a ：将 binary 文件以 text 文件的方式搜寻数据
-c ：计算找到 '搜寻字符串' 的次数
-i ：忽略大小写的不同，所以大小写视为相同
-n ：顺便输出行号
-v ：反向选择，亦即显示出没有 '搜寻字符串' 内容的那一行！
--color=auto ：可以将找到的关键词部分加上颜色的显示喔！
```

GIT

如何将本地项目上传到git？

```
git pull origin master(或者分支名)                    //push前最好切换到主分支使用pull来拉取下最新资源。避
git init                //初始化版本库
git add .                //添加到缓存区中
git commit -m 'first commit'        //提交到版本库，并注释
git push -u origin master(或者分支名)        //第一次推送时候要写
git push origin master(或者分支名)     //如果不是第一次，则直接使用该命令即可推送修改
```

git常用命令

```
//新建分支
git branch 分支名
//切换分支
git checkout 分支名
//删除本地分支
git branch -d 分支名
//强制删除本地分支
git branch -D 分支名
//删除远程分支
git push origin --delete 分支名

//查看本地所有分支
git branch
//查看远程所有分支
git branch -r
//查看远程和本地所有分支--一般用这个
git branch -a

//重命名本地分支
git branch -m <oldbranch> <newbranch>
```

push和pull

push：本地分支合并到远程分支

pull：将远程分支合并到本地分支

git fetch&git pull详解

参考链接和评论

`git fetch`的意思是将远程主机的最新内容拉到本地，用户再检查无误后再决定是否合并到工作本地分支中

`git pull`是将远程主机中的最新内容拉取下来后直接合并，即：git pull = git fetch+git merge，这样可能会产生冲突，需要手动解决。

git冲突原因和解决冲突

产生原因：多个开发者同时使用或者操作git中的同一个文件，最后在依次提交commit和推送push的时候，第一个操作的是可以正常提交的，而之后的开发者想要执行pull和fetch操作的时候，就会报冲突异常conflict。

1. 两个分支中修改了同一个文件（不管什么地方）
2. 两个分支中修改了同一个文件的名称

冲突消除的方式：

1. git pull命令。拉取远程分支上的代码并合并到本地分支，目的是消除冲突；
2. git stash命令。把工作区的修改提交到栈区，目的是保存工作区的修改；

git撤销commit但是未git push的情况（如在 Git 恢复先前的提交？）

```
//找到上次git commit的id
git log
//执行撤销操作，同时将代码恢复到该commit_id之前的代码提交状态
git reset --hard  commit_id
//执行撤销但是保留更改
git reset  commit_id
```

git rebase和git merge

合并前有两个分支：

```
A <- B <- C    [master]
^
 \
  D <- E       [branch]
```

在 git merge master 之后：

```
A <- B <- C
^         ^
 \         \
  D <- E <- F
```

在 git rebase master 之后：

```
A <- B <- C <- D <- E
```

rebase变基会破坏分支

GDB

gdb使用流程

- 启动gdb -g表示调试
- list查看代码，默认查看10行
- run(或者写成r)，没有断点就运行到结束，有断点就运行到断点处
- start：程序从`main`函数的起始位置停下，开始逐步调试。
- break(b)，设置断点
- info break：查看断点信息
- delete：删除断点
- continue、step、next命令
    i. continue：继续执行程序，直到遇到下一个断点或者结束
    ii. next：单步执行，遇到函数时会跳过函数，不进入函数体内部
    iii. step：单步执行程序，但遇到函数会进入到函数内部
- until：结束一个循环体循环
- print：显示变量或者表达式的值

普通调试

- step和next的区别？

next会直接执行到下一句 ,step会进入函数体内部执行

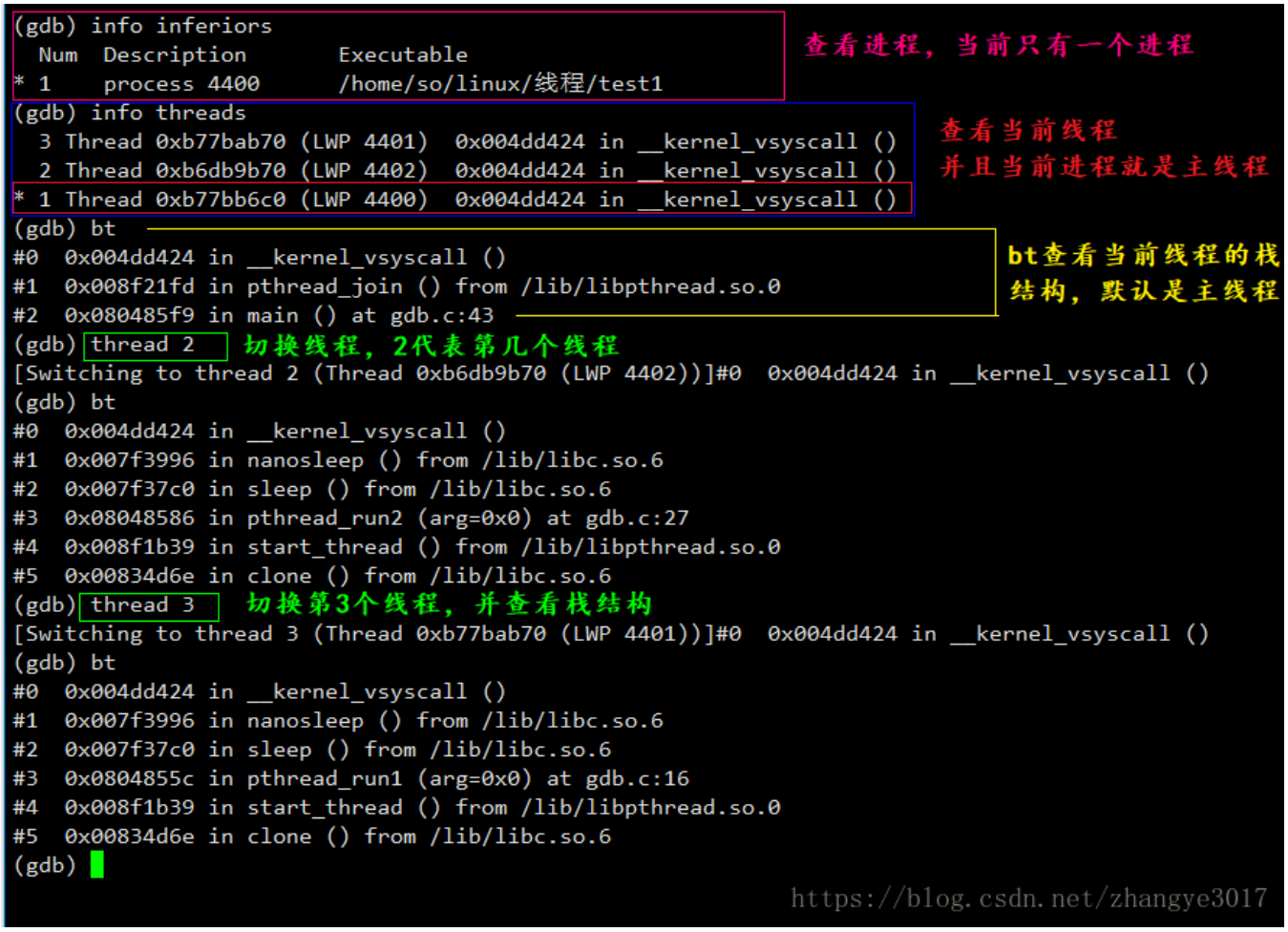
- list

查看源码

- set
  设置变量值
- backtrace
  查看函数调用的栈帧关系
- framework
  切换函数栈帧
- info

查看函数内部局部变量

多线程调试



- info threads
  查看所有线程，默认分支是主线程
- thread id
  切换线程
- bt
  查看每个线程的栈帧然后设置断点
- thread apply （n或all） 命令
  使用thread apply来让一个或是多个线程执行指定的命令。例如让所有的线程打印调用栈信息。
- set scheduler-locking on
  锁定只有当前的线程能够执行

GDB原理

gcc常用参数

- -v/--version：查看gcc的版本
- -I：编译的时候指定头文件路径，不然头文件找不到
- -c：将汇编文件转换成二进制文件，得到.o文件
- -g：gdb调试的时候需要加
- -D：编译的时候指定一个宏（调试代码的时候需要使用例如printf函数，但是这种函数太多了对程序性能有影响，因此如果没有宏，则#ifdefine的内容不起作用）
- -wall：添加警告信息
- -On：-O是优化代码，n是优化级别：1，2，3

CMake

> CMake介绍：CMake是一个跨平台的编译工具。能够输出各种各样的Makefile文件

cmake_minimum_required 命令

min 指定了编译当前的cmake需要的最old的版本号，如果cmake版本号小于指定的版本号时，运行cmake 就报错。

set命令

设置一个变量的值。

message 命令

用于记录log消息的，常用的有：

- FATAL_ERROR： cmake errer, 遇到该错误，cmake 直接停止处理。
- WARNING： 警告信息， 但是cmake会继续执行下去的。

- STATUS： 用于记录一些cmake 运行过程中的有用的信息。

project 命令

使用语法为： `project(项目名字, [VERSION 版本号] [DESCRIPTION 描述字符串] )`

add _executable

普通的可执行文件

add_library

用于添加一个库文件的target, 包含静态库、动态库等

target_include_directories

该命令用于指令要包含的路径名。 当使用 gcc 编译器编译目标时，就会使用`-I`选项指定该命令添加的路径， 使用`-I directory/include`.

add_subdirectory 命令

用于增加一个构建的子目录。 注意执行的数据流： cmake 命令会执行完子目录中的CmakeLists.txt文件之后，再执行后面的命令。

valgrind

协程

嵌入式

掌握常用的系统总线接口及典型外设开发，如USB、SPI、I2C、CAN、SDIO等，有基本的硬件功底，能看懂基本的原理图和PCB。

交叉编译

交叉编译指的是，在一种计算机环境中运行的编译程序，能编译出在另外一种环境下运行的代码。实际上包含两个概念：体系结构（Architecture）、操作系统（OperatingSystem）。同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。

程序开发一般有两种情形，第一种是在一种设备上开发，编译生成的程序在同类设备上运行，如我们电脑里的office等，这种叫本地编译。第二种则是在一种设备上进行编辑、编译(宿主机)，而生成的执行程序却在另一种设备上执行(目标机)，即开发环境和运行环境不一样，如单片机、嵌入式系统程序，这就是交叉编译。

为什么要在宿主机上进行开发，而不直接在目标机上进行？

最主要的原因是：目标机资源太有限了(出于成本考虑，其硬件资源一般都只能满足特定功能需求)，难以支撑开发环境的运行需求：如CPU占用，内存开销，硬盘占用等.

云原生

什么是云原生？

云原生是一种**构建和运行应用程序的方法**，是一套技术体系和方法论。云原生（CloudNative）

是一个组合词，Cloud+Native。Cloud表示应用程序位于云中，而不是传统的数据中心；Native表示应用程序从设计之初即考虑到云的环境，原生为云而设计，**在云上以最佳姿势运行**，充分利用和发挥云平台的弹性+分布式优势。云原生是一种**构建和运行应用程序的方法**，是一套技术体系和方法论。云原生（CloudNative）是一个组合词，Cloud+Native。Cloud表示应用程序位于云中，而不是传统的数据中心；Native表示应用程序从设计之初即考虑到云的环境，原生为云而设计，**在云上以最佳姿势运行**，充分利用和发挥云平台的弹性+分布式优势。

```
Error: connect ETIMEDOUT 93.46.8.90:443
```

总而言之，符合云原生架构的应用程序应该是：采用开源堆栈（K8S+Docker）进行容器化，基于微服务架构提高灵活性和可维护性，借助敏捷方法、DevOps支持持续迭代和运维自动化，利用云平台设施实现弹性伸缩、动态调度、优化资源利用率。

Docker

背景

我们写代码会接触开发环境，测试环境和生产环境。

代码开发环境下测试没问题后会打包发给测试人员，测试人员会在测试环境下跑，如果没问题，会将打包文件发送的生产环境部署。

但是会出现问题：叫做水土不服，可能在开发环境没问题，但是测试环境是可以的。

所以直接干脆把环境+代码（容器）直接发给测试，就会规避了软件跨环境迁移的问题。

概念

- Docker是一个开源的应用容器引擎，诞生于2013年初，基于Go语言实现，dotCloud公司出品（后改名Docker Inc）
- Docker可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的Linux机器上
- 容器是完全使用沙箱机制，相互隔离
- 容器性能开销极低

- docker是一种容器技术，解决了软件跨环境迁移的这么一个问题

架构

- 镜像（Image）：Docker镜像（Image），就相当于是一个root文件系统。比如官方镜像 ubuntu:16.04就包含了完整的一套Ubuntu16.04最小系统的root文件系统
- 容器（Container）：镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和对象一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等
- 仓库（Repository）：仓库可以看成一个代码控制中心，用来保存镜像

Docker服务命令

Docker 服务相关命令

- 启动docker 服务：
    - `systemctl start docker`
- 停止docker 服务：
    - `systemctl stop docker`
- 重启docker 服务：
    - `systemctl restart docker`
- 查看docker 服务状态：
    - `systemctl status docker`
- 设置开机启动docker：
    - `systemctl enable docker`

2.2 Docker 镜像相关命令

- 查看镜像：查看本地所有的镜像
    - ```
      docker images
      docker images -q #查看所有镜像的id
      ```
- 搜索镜像：从网络中查找需要的镜像
    - ```
      docker search 镜像名称
      ```
- 拉取镜像：从Docker 仓库下载镜像到本地，镜像名称格式为`名称:版本号`，如果版本号不指定则是最新的版本。如果不知道镜像版本，可以去docker hub 搜索对应镜像查看
    - ```
      docker pull 镜像名称
      ```
- 删除镜像：删除本地镜像
    - ```
      docker rmi 镜像id/名称号:版本号 #删除指定本地镜像
      docker rmi 'docker images -q' #删除所有本地镜像
      ```

2.3 Docker 容器相关命令

- 查看容器
    - ```
      docker ps #查看正在运行的容器
      docker ps -a #查看所有容器
      ```
- 创建并启动容器
    - ```
      docker run 参数 版本:版本号 </bin/bash>#默认为/bin/bash
      ```
    - 参数说明：
        - `-i`：保持容器运行。通常与`-t`同时使用。加入`it`这两个参数后，容器创建后自动进入容器中，退出容器后，容器自动关闭
        - `-t`：为容器重新分配一个伪输入终端，通常与`-i`同时使用

        - `-d`：以守护（后台）模式运行容器。创建一个容器在后台运行，需要使用`docker exec`进入容器`docker exec -it c2 /bin/bash`。退出后，容器不会关闭
        - `-it`创建的容器一般称为交互式容器；`-id`创建的容器一般称为守护式容器
        - `--name`：为创建的容器命名
- 进入容器
    - ```
      docker exec 参数 #退出容器，容器不会关闭
      ```
- 停止容器
    - ```
      docker stop 容器名称
      ```
- 启动容器
    - ```
      docker start 容器名称
      ```
- 删除容器：如果容器是运行状态则删除失败，需要停止容器参能删除
    - ```
      docker rm 容器名称
      ```
- 查看容器信息
    - ```
      docker inspect 容器名称
      ```

Docker 容器的数据卷

数据卷概念及作用

思考：

- Docker 容器删除后，在容器中产生的数据还在吗？
- Docker 容器和外部继器可以交换文件吗？
- 容器之间想要进行数据交互？

数据卷
      ```

- 数据卷是宿主机中的一个目录或文件，跟虚拟机挂载很像
- 当容器目录和数据卷目录绑定后，对方的修改会立即同步
- 一个数据卷可以被多个容器同时挂载
- 一个容器也可以被挂载多个数据卷

数据卷的作用

- 容器数据持久化
- 外部计价器和容器间接通信
- 容器之间数据交换

配置数据卷

- 创建启动容器时，使用`-v`参数 设置数据卷
    - `docker run ... -v 宿主机目录(文件):容器内目录(文件) ...`
- 注意事项：
    i. 目录必须是绝对路径
    ii. 如果目录不存在，会自动创建
    iii. 可以挂载多个数据卷

多容器数据交换

多个容器挂载到同一个数据卷

image这个c3就是一个数据卷容器

Docker 应用部署

MySQL部署

需求

- 在Docker 容器中部署MySQL，并通过外部MySQL 客户端操作MySQL Server

实现

1. 搜索mysql镜像

2. 拉取mysql镜像
3. 创建容器
4. 操作容器中的mysql

问题及解决方案

- 容器内的网络服务和外部机器不能直接通信，解决方法如下：
  外部机器和宿主机可以直接通信，同时宿主机和容器可以直接通信。当容器中的网络服务需要被外部机器访问时，可以将容器中提供服务的端口映射到宿主机的端口上。外部机器访问宿主机的端口，从而间接访问容器的服务
- 这种操作称为：**端口映射**

4.1.4 部署MySQL

1. 搜索mysql镜像

```
docker search mysql
```

2. 拉取mysql镜像

```
docker pull mysql:5.6
```

3. 创建容器，设置端口映射、目录映射

```
# 在/root目录下创建mysql目录用于存储mysql数据信息
mkdir ~/mysql
cd ~/mysql
```

```
docker run -id \
-p 3307:3306 \
--name=c_mysql \
-v $PWD/conf:/etc/mysql/conf.d \
-v $PWD/logs:/logs \
-v $PWD/data:/var/lib/mysql \
-e MYSQL_ROOT_PASSWORD=123456 \
mysql:5.6
```

   - 参数说明
     - `-p 3307:3306`：将容器的3306端口映射到宿主机的3307端

口
    - `--v $PWD/conf:/etc/mysql/conf.d`：将主机当前目录下的`conf/my.cnf`挂载到容器`/etc/mysql/my.cnf`配置目录
    - `-v $PWD/logs:/logs`：将主机当前目录下的`logs`目录挂载到容器的`/logs`目录日志
    - `-v $PWD/data:/var/lib/mysql`：将主机当前目录下的`data`目录挂载到容器的`/var/lib/mysql`数据目录
    - `-e MYSQL_ROOT_PASSWORD=123456`：初始化root 用户密码
4. 使用外部机器访问MySQL
   宿主机的ip，映射的端口号就行

Redis部署

1. 搜索Redis 镜像

```
docker search redis
```

2. 拉取Redis 镜像

```
docker pull redis:5.0
```

3. 创建容器，设置端口映射、目录映射

```
docker run -id --name=c_redis -p 6379:6379 redis:5.0
```

4. 使用外部机器连接redis

```
redis-cli.exe -h 192.168.187.129 -p 6379
```

Dockerfile

Docker 镜像原理

思考：

- Docker 镜像的本质是什么？
    - 是一个分层的文件系统

- Docker 中一个CentOS 镜像为什么只有200MB，而一个CentOS 操作系统的iso文件要几个G？
    - CentOS的iso镜像文件包含bootfs和rootfs，而Docker的CentOS镜像复用操作系统的bootfs，只有rootfs和其他镜像层
- Docker 中一个Tomcat 镜像为什么有500MB，而一个Tomcat 安装包只有70多MB？
    - 由于Docker中镜像是分层的，tomcat虽然只有70多MB，但他需要依赖于父镜像和基础镜像，所以整个对外暴露的tomcat镜像大小500多MB

操作系统组成部分：

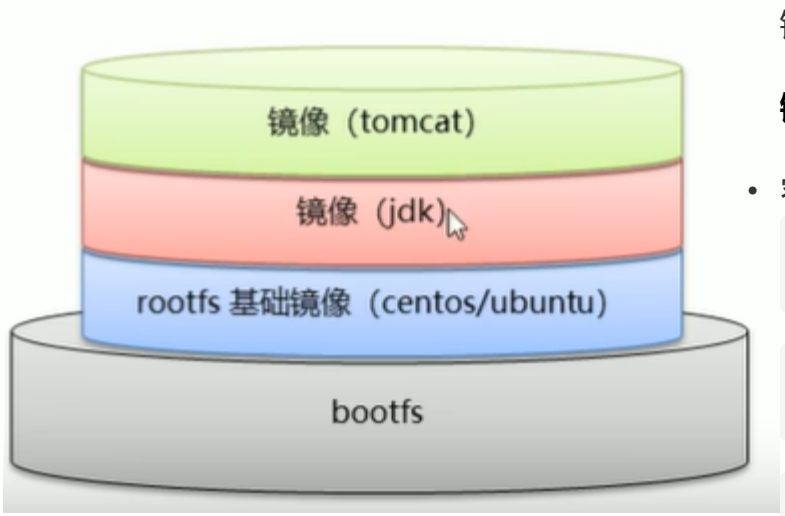
- 进程调度子系统
- 进程通信子系统
- 内存管理子系统
- 设备管理子系统
- **文件管理子系统**
- 网络通信子系统
- 作业控制子系统

Linux文件系统由bootfs 和rootfs 两部分组成

- bootfs：包含bootloader（引导加载程序）和kernel（内核）
- rootfs：root文件系统，包含的就是典型的Linux 系统中的/dev、/proc、/bin等标准目录和文件
- 不同的Linux 发行版，bootfs 基本一样，而rootfs 不同，如ubuntu，CentOS等

Docker 镜像原理：

- Docker 镜像是由特殊的文件系统叠加而成
- 最低端是bootfs，docker会使用宿主机的bootfs
- 第二层是root文件系统rootfs，称为base iamge，然后再往上可以叠加其他的镜像文件，如下
- 统一文件系统（Union File System）技术能够将不同的层整合成一个文件系统，为这些层提供了一个统一的视角，这样就隐藏了多层的存在，在用户的角度看来，只存在一个文件系统
- 一个镜像可以放在另一个镜像的上面。位于下面的镜像称为父镜像，最底部的镜像称为基础镜像
- 当从一个镜像启动容器时，Docker会在最顶层加载一个读写文件系统作为容器(想改



镜像的时候)

镜像制作：

- 容器转为镜像

```
docker commit 容器id 镜像名称:版本号
```

```
docker save -o 压缩文件名称 镜像名称
```

```
docker load -i 压缩文件名称
```

- 问题：挂载的文件不会出现在别的主机上（挂载的数据卷不会出现）
  也就是说在容器中添加的文件可以被打包放到别的主机，但是挂载的文件不可以。
  引出来了dockerfile概念

Dockerfile 概念及作用

Dockerfile 概念

- Dockerfile 是一个文本文件
- 包含了一条条的指令
- 每一条指令构建一层，基于基础镜像，最终构建出一个新的镜像
- 对于开发人员，可以为开发团队提供一个完全一致的开发环境
- 对于测试人员，可以直接拿开发时所构建的镜像或者通过Dockerfile 文件构建一个新的镜像开始工作了
- 对于运维人员，在部署时，可以实现应用的无缝移植

Dockerfile 关键字

关键字	作用	备注
FROM	指定父镜像	指定dockerfile基于哪个images构建
MAINTAINER	作者信息	用来标明这个dockerfile 谁写的
LABEL	标签	用来指明dockerfile 的标签，可以使用Label代替Mair

关键字	作用	备注
RUN	执行命令	执行一段命令 默认是`/bin/sh` 格式：`RUN command` 或
CMD	容器启动命令	提供启动容器时候的默认命令和ENTRYPOINT配合使
ENTRYPOINT	入口	一般在制作一些执行就关闭的容器中会使用
COPY	复制文件	build 的时候复制文件到image中
ADD	添加文件	build 的时候添加文件到iamge 中，不仅仅局限于当前
ENV	环境变量	指定build 时候的环境变量 可以在启动容器的时候 通
ARG	构建参数	构建参数 只在构建的时候使用参时 如果有ENV 那么
VOLUME	定义外部可以挂载的数据卷	指定build 的image 那些目录可以启动的时候挂载到文
EXPOSE	暴露端口	定义容器运行的时候监听的端口 启动容器的使用`-p`
WORKDIR	工作目录	指定容器内部的工作目录 如果没有创建则自动创建 如 的路径的相对路径
USER	指定执行用户	指定build 或者启动的时候 用户 在RUN CMD ENTRY
HEALTHCHECK	健康检查	指定监测当前容器的健康测试的命令 基本上没有 因为
ONBUILD	触发器	当存在ONBUILD 关键字的镜像作为基础镜像的时候 用处也不怎么大
STOPSIGNAL	发送信息量到宿主机	该STOPSIGNAL指令设置将发送到容器的系统调用信
SHELL	指定执行脚本的shell	指定RUN CMD ENTRYPOINT 执行命令的时候 使用

5.4 案例

5.4.1 案例一

需求：

自定义CentOS7镜像。要求：

1. 默认登录路径为`/usr`

2. 可以使用vim

实现步骤：

1. 定义父镜像：`FROM centos:7`
2. 定义作者信息：`MAINTAINER crisp077 <www.crisp077.xyz>`
3. 执行安装vim命令：`RUN yum install -y vim`
4. 定义默认的工作目录：`WORKDIR /usr`
5. 定义容器启动执行的命令：`CMD /bin/bash`

创建使用dockerfile的镜像：

```
docker build -f ./centos_docker -t crisp_centos:1 .
```

5.4.2 案例二

需求：

定义dockerfile，发布springboot 项目

实现步骤：

1. 定义父镜像：`FROM java:8`
2. 定义作者信息：`MAINTAINER crisp077 <www.crisp077.xyz>`
3. 将jar包添加到容器：`ADD springboot.jar app.jar`
4. 定义容器启动执行的命令：`CMD java -jar app.jar`
5. 通过dockerfile 构建镜像：`docker build -f dockerfile文件路径 -t 镜像名称:版本`

Docker 服务编排

服务编排的概念

微服务架构的应用系统中一般包含若干个微服务，每个微服务都会部署多个实例，如果每个微服务都要手动启动，维护工作量会很大

- 要从Dockerfile build image 或者去 dockerhub 拉取image
- 要创建多个container

- 要管理这些container（启动停止删除）

服务编排：

按照一定的业务规则批量管理容器

Dockers Compose 概述

Docker Compose 是一个编排多容器分布式部署的工具，提供命令集管理器化应用的完整开发期，包括服务构建，启动和停止。使用步骤：

1. 利用 Dockerfile 定义运行环境镜像
2. 使用 docker-compose.yml 定义组成应用的各服务
3. 运行 docker-compose up 启动应用

安装Docker Compose

```
# Compose 目前已经完全支持Linux、MAC OS、Windows，在安装Compose之前，需要先安装Docker。下面以编译好的二
curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-compose-'uname -s'-'unam
# 设置文件可执行权限 {#设置文件可执行权限 }
chmod +x /usr/local/bin/docker-compose
# 查看版本信息 {#查看版本信息 }
docker-compose -version
```

卸载Docker Compose

```
# 二进制包方式安装的，删除二进制文件即可 {#二进制包方式安装的删除二进制文件即可 }
rm /usr/local/bin/docker-compose
```

Docker容器虚拟化 与 传统虚拟机比较

容器就是将软件打包成标准化单元，以用于开发、交付和部署

- 容器镜像是轻量级的、可执行的独立软件包，包含软件运行所需要的所有内容：代码、运行时环境、系统工具、系统库和设置
- 容器化软件在任何环境中都能够始终如一地运行
- 容器赋予了软件独立性，使其免受外在环境差异的影响，从而有助于减少团队间在相同基础设施上运行不同软件时的冲突

相同：

- 容器和虚拟机具有相似的资源隔离和分配优势

不同：

- 容器虚拟化的是操作系统，虚拟机虚拟化的时硬件
- 传统的虚拟机可以运行不同的操作系统，容器只能运行同一类型的操作系统

特性	容器	虚拟机
启动	秒级	分钟级
性能	接近原生	弱于
系统支持两	单机支持上千个容器	一般几十个

如何在不同操作系统部署Docker环境

Docker刚推出的时候只支持Ubuntu，后来才一点点开始对其他平台的支持。所以在Ubuntu平台上部署Docker平台还是挺简单的。

Kubernetes

背景

Google 从 2000 年初就开始使用容器（Linux 容器）系统，Google 开发出来的第一个统一的容器管理系统在内部称之为 “Borg”，用来管理长时间运行的生产服务和批处理服务。由于 Borg 的规模、功能的广泛性和超高的稳定性，一直到现在 Borg 在 Google 内部依然是主要的容器管理系统。

Google 的第二套容器管理系统叫做 Omega，作为 Borg 的延伸，它的出现是出于提升 Borg 生态系统软件工程的愿望。Omega 应用到了很多在 Borg 内已经被认证的成功的模式，但是是从头开始来搭建以期更为一致的构架。由于越来越多的应用被开发并运行在 Borg 上，Google 开发了一个广泛的工具和服务的生态系统。它被应用到了很多在 Borg 内已经被认证的成功的模式，但是是从头开始来搭建以期更为一致的构架。这些系统提供了配置和更新 job 的机制，能够

预测资源需求，动态地对在运行中的程序推送配置文件、服务发现、负载均衡、自动扩容、机器生命周期管理、额度管理等。许多 Omega 的创新（包括多个调度器）都被收录进了 Borg。

Google 的第三套容器管理系统就是我们所熟知的 Kubernetes，它是针对在 Google 外部的对 Linux 容器感兴趣的开发者以及 Google 在公有云底层商业增长的考虑而研发的。和 Borg、Omega 完全是谷歌内部系统相比，Kubernetes 是开源的。像 Omega 一样，Kubernetes 在其核心有一个被分享的持久存储，有组件来检测相关 object 的变化。跟 Omega 不同的是，Omega 把存储直接暴露给信任的控制平面的组件，而在 Kubernete 中，提供了完全由特定领域更高层面的版本控制、认证、语义、策略的 REST API 接口，以服务更多的用户。更重要的是，Kubernetes 是由一群底层开发能力更强的开发者开发的，他们主要的设计目标是用更容易的方法去部署和管理复杂的分布式系统，同时仍能从容器提升的效率中受益。

2014 年 Kubernetes 正式开源，2015 年被作为初创项目贡献给了云原生计算基金会（CNCF），从此开启了 Kubernetes 及云原生化的大潮。

参考

什么是？

是一种可自动实施 Linux 容器操作的开源平台。它可以帮助用户省去应用容器化过程的许多手动部署和扩展操作。也就是说，您可以将运行 Linux 容器的多组主机聚集在一起，由 Kubernetes 帮助您轻松高效地管理这些集群。

Kubernetes 是 Google 基于 Borg 开源的容器编排调度引擎，作为 CNCF（Cloud Native Computing Foundation）最重要的组件之一，它的目标不仅仅是一个编排系统，而是提供一个规范，可以让你来描述集群的架构，定义服务的最终状态，Kubernetes 可以帮你将系统自动得达到和维持在这个状态。

更直白的说，Kubernetes 用户可以通过编写一个 YAML 或者 json 格式的配置文件，也可以通过工具 / 代码生成或直接请求 Kubernetes API 创建应用，该配置文件中包含了用户想要应用程序保持的状态，不论整个 Kubernetes 集群中的个别主机发生什么问题，都不会影响应用程序的状态，你还可以通过改变该配置文件或请求 Kubernetes API 来改变应用程序的状态。

Kubernetes 通过声明式配置，真正让开发人员能够理解应用的状态，并通过同一份配置可以立马启动一个一模一样的环境，大大提高了应用开发和部署的效率，其中 Kubernetes 设计的多种资源类型可以帮助我们定义应用的运行状态，并使用资源配置来细粒度的明确限制应用的资源使用。

微服务

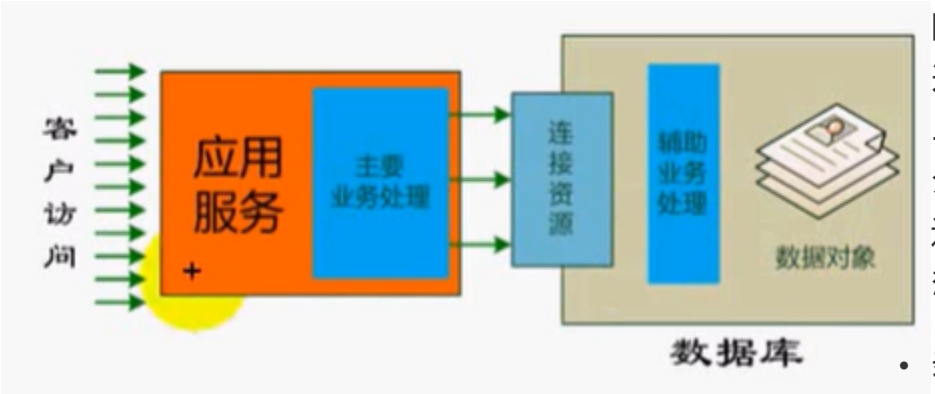
设计目标

- 高性能，针对大量的并发请求做出快速的做出响应
- 高可用，服务器7*24小时不间断的工作，故障转移自动完成不需要认为干预(failover)
- 伸缩性，良好的框架，分层处理，可以分布式部署。如果A与B不在同一个物理机上，则不能使用本地进程通信，比如共享内存，管道这类技术，应该使用TCP这样跨服务端的进程间通信

简单到复杂的服务器架构

首先，任何网络系统都可以抽象成C/S结构，比如我们常说的B/S模式，是WEB兴起后的一种C/S模式架构

然后最简单的情况是只有一台应用服务器（WEB服务器＋数据库服务器），这一台应用服务器做了三件事，或者说包含了三个模块，分别是IO处理单元，业务逻辑处理单元，存储处理单元。这种基本个人写个小项目的话都是在一台电脑上集成了这三个模块。



随着业务的增大，并发请求逐渐增多，我们就可以把这三个模块都拆开，每个模块分别用很多台及其去处理，这也是大型服务器项目的思想。

- 剥离数据库

  首先是把存储模块给剥离出去，因为随着用户访问量的增加，我们数据越来越多，肯定需要一个专门的服务器来存储数据。

  接着，因为我们的业务代码时时刻刻要去访问数据库，并且随着流量增大并发的增多，导致服务器请求失败或者响应时间大大增加。比如数据库只能够支持10个并发连接，但是我们服务器承接了1000个并发请求，会有990个请求失败。同时假设数

据库1秒内可以处理1000个请求，应用服务器有10000个并发，延迟可能会达到10秒。所以，数据库就成为了性能的瓶颈。

这个时候我们需要在应用服务器和数据库服务器之间加一个中间层，有的叫做DAL(DataBase Access Layer)，中间层可以是一台单独的服务器来完成。DAL里面呢对于现在的我来说就是可以用阻塞队列来维护连接请求，然后创建一个连接池，采用连接池+任务队列的方式来访问数据库，有了连接池我们就不需要每次有请求都必须重新连接数据库，因为涉及到TCP通信等等，效率肯定会有损失。

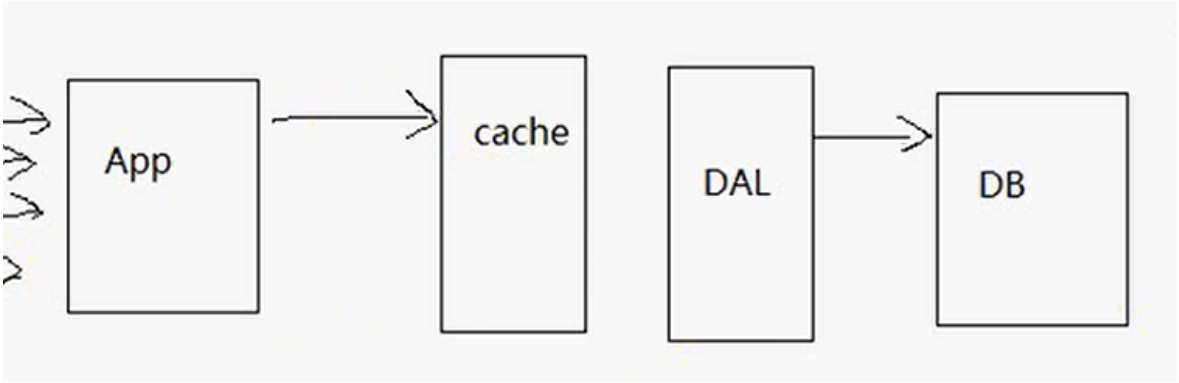
但是这样也不能完全解决，因为比如10万个并发，上面的架构可能只能解决5万并发，那么剩下一半就需要等待，就会出现延迟。

- 增加一个缓存

  缓存的思想在计算机中是普遍使用的，比如CPU的三级缓存这些。

  使用缓存会一个问题，即缓存的更新或缓存的同步问题（缓存一致性，这样是一道面试题，在数据库中有写）。

  服务器架构如下：



  一般我们业务都会用专业的缓存数据库， nosql当做缓存来使用

- 数据库太大了，更加细致分，读和写

  一般来说数据库都是读多写少，当有大量的并发请求的时候，我们就把数据库继续拆分，主从机制

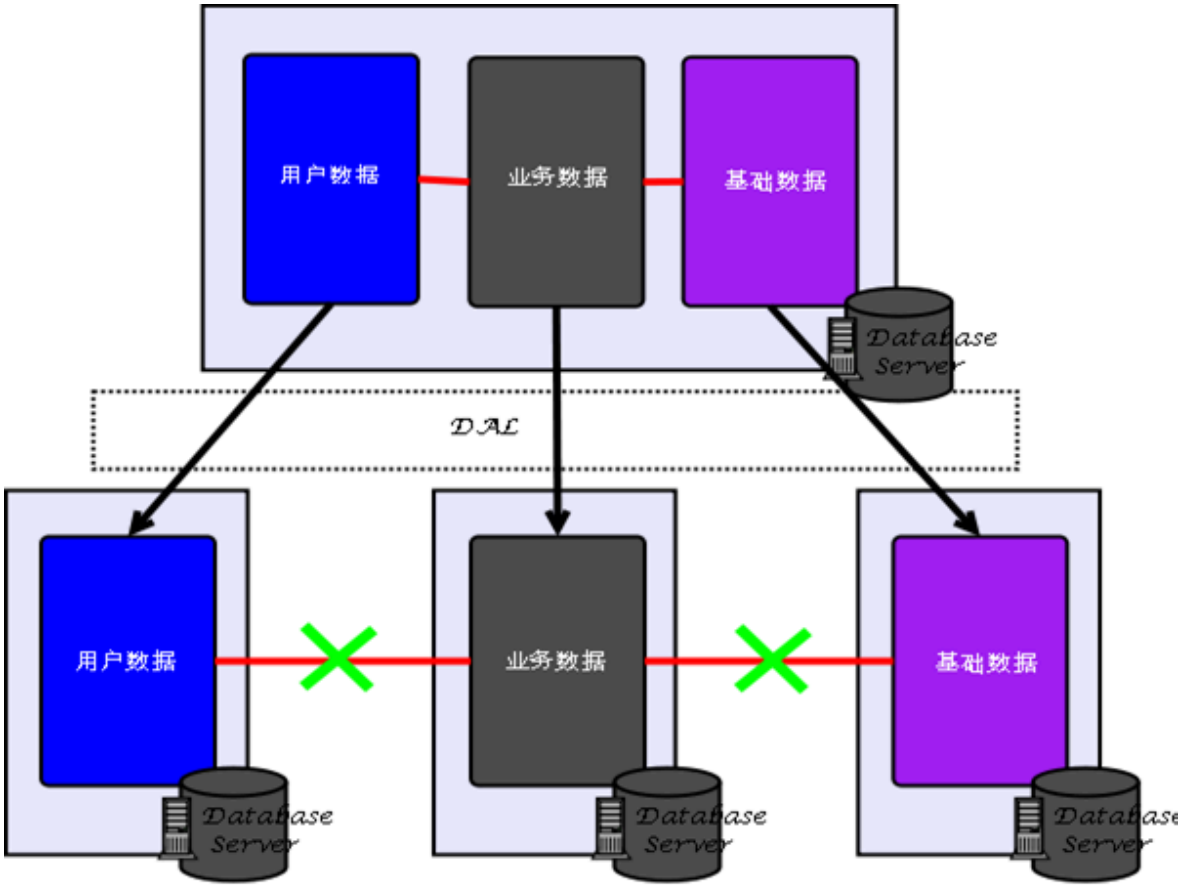
  master and slave， master库主要用来写，slave库主要用来读

  上面的结构对逻辑层更难，因为从数据库主要是为了读而存在的，即读的话去从数据库，写的话写入主数据库，当主数据库更新的时候会通知从数据库（同步机制 又叫做replication机制）

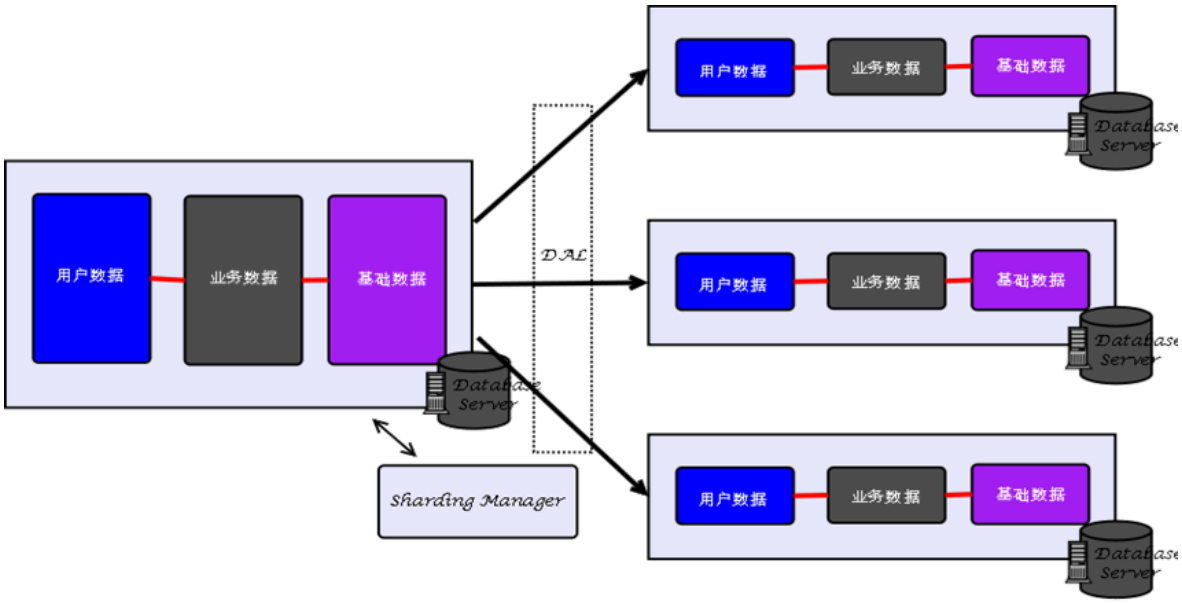
  把一个数据库变成了多个数据库服务器，实现了负载均衡，提高了并发能力

- 下一步，分库分表

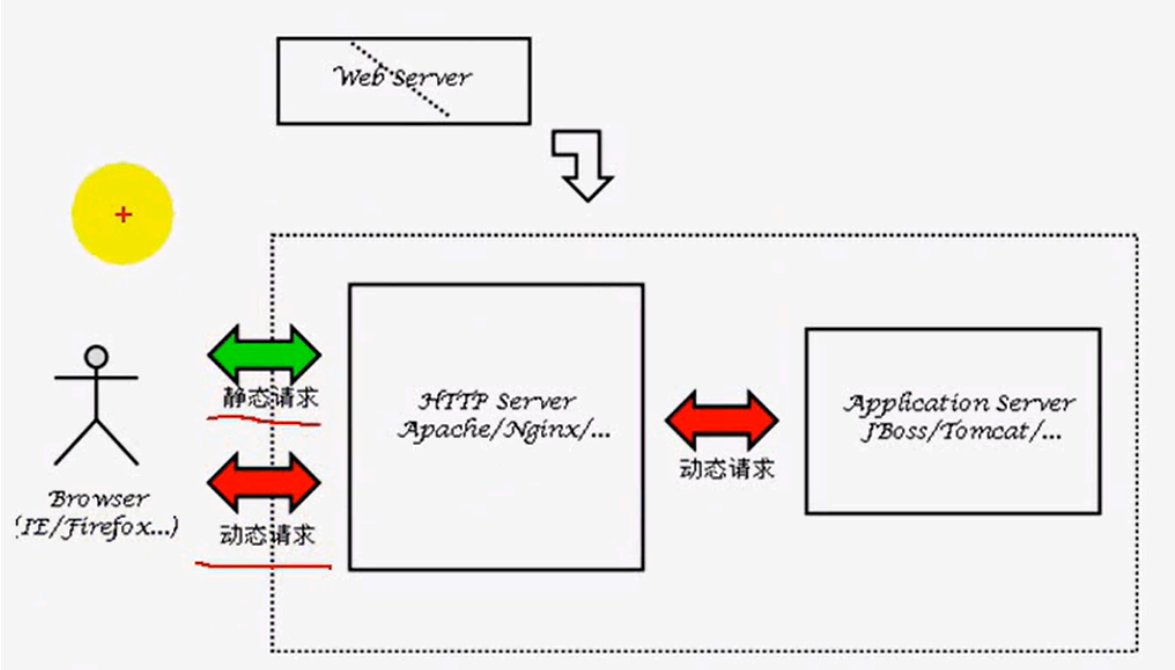
  分库/垂直分区



分表/水平分区

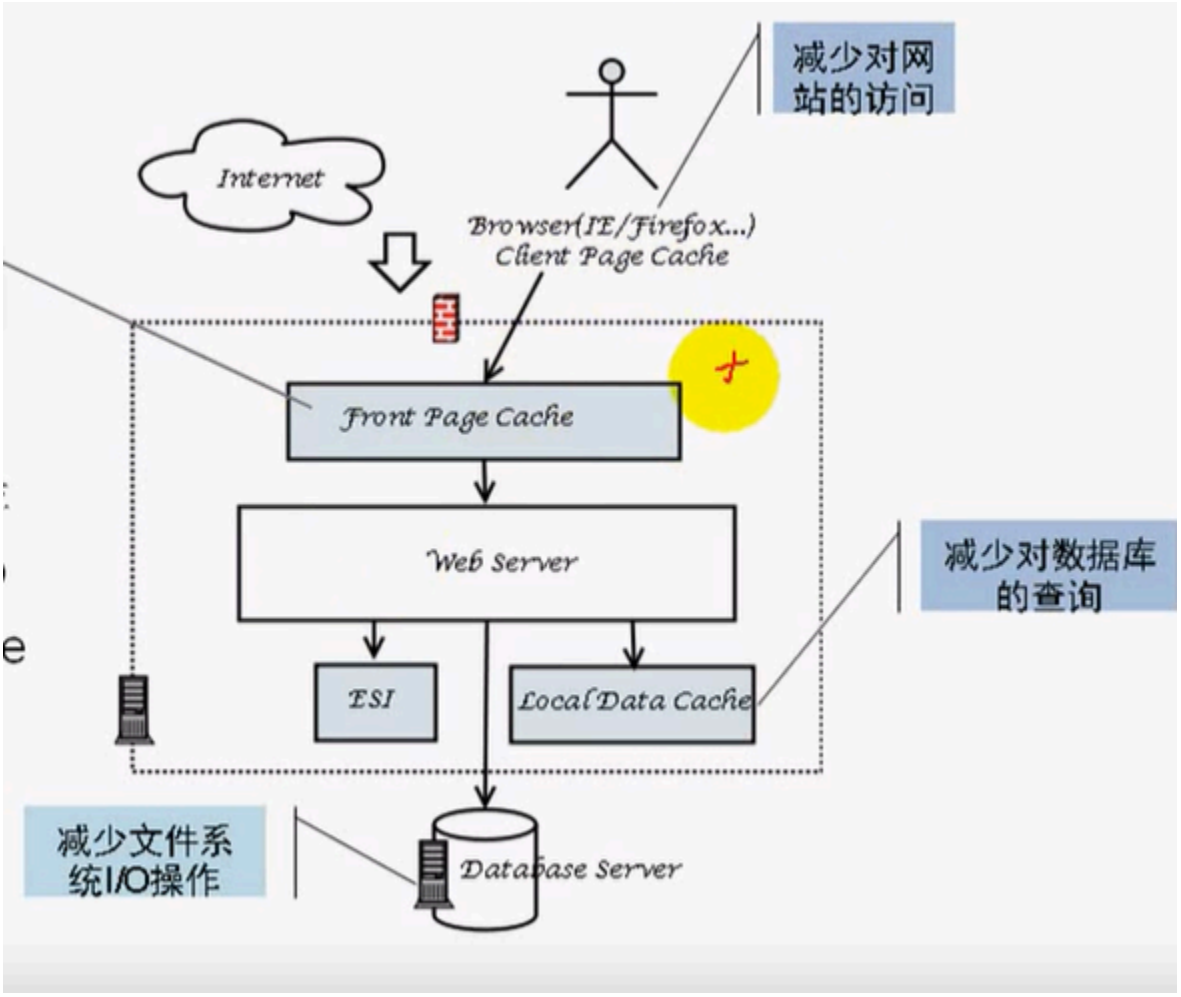


- 数据库没啥问题了，然后对业务处理拆分
  客户或者服务器访问浏览器，有动态请求和静态请求，如下图



静态请求就是html、js、cs这种，动态请求就是asp.net、JSP、PHP这种。所以自然而然，服务器也分为了HTTP server(Apache、Nginx)和Application server(Tomcat)。

- 客户端和服务器之间的缓存处理



i. 浏览器缓存
ii. 前端页面缓存squid
iii. 页面片段缓存ESI（Edge Side Includes）
iv. 本地数据缓存

- 为了更细分业务，可以增加一个任务服务器，在客户端和应用服务器之间
  i. 任务服务器用来分配任务
     用来监视应用服务器的负载，CPU，IO，内存换页等情况
     然后任务服务器将各个类型的任务分配给应用服务器
  ii. 应用服务器主动索取任务，这种会比较好
- 随着访问量的又一部加大，我们可以增加成web服务器集群，然后会出现负载均衡的问题
  负载均衡在整个服务器架构中分为前端负载均衡，应用服务器负载均衡，数据库负载均衡
  前端负载均衡（客户→Web服务器这个过程）
  i. DNS负载均衡
     在DNS服务器中，可以为多个不同的地址配置同一个名字，对于不同的

客户机访问同一个名字，得到不同的地址。比如百度的域名www.baidu,.com对应很多个地址

ii. 反向代理

> 关于正向代理和反向代理我放到计算机网络那一块

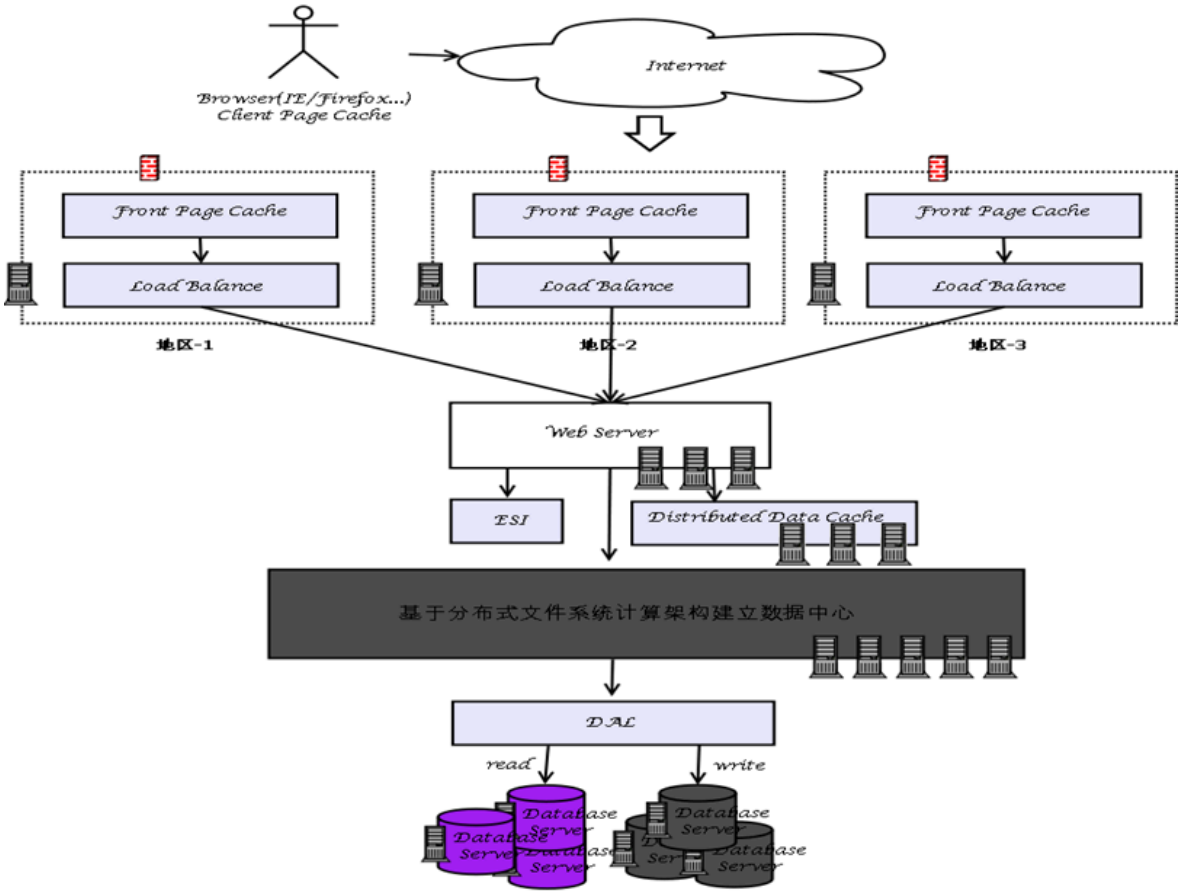
使用代理服务器将请求发给内部服务器，让代理服务器将请求均匀转发给多台内部web服务器之一，从而达到负载均衡的目的。标准代理方式是客户使用代理访问多个外部Web服务器，而这种代理方式是多个客户使用它访问多个内部Web服务器，因此也被称为反向代理模式。

iii. 基于NAT的负载均衡

外部客户端访问会进入到NAT服务器，然后NAT地址映射到不同的内部服务器

- 数据量再续加大的话

首先存储数据，要建立一个分布式文件系统数据中心，nosql的存储并发性都比较高然后分地区的去建立服务体系，如下：



> DFS分布式文件系统，如：Lustre\HDFS\GFS\TFS\FreeNas等
>
> Key-Value DB，也作为NoSQL解决方案，如:BigTable\Tair\Hbase\HyperTable

> 等
>
> Map/Reduce算法（计算框架），基本上现有NoSQL数据库中都支持此算法。
>
> 提供完整解决方案：
>
> Google(GFS|BigTable|Map/Reduce)

X86/ARM

重温CPU

中央处理单元（CPU）主要由运算器、控制器、寄存器三部分组成，从字面意思看运算器就是起着运算的作用，控制器就是负责发出CPU每条指令所需要的信息，寄存器就是保存运算或者指令的一些临时文件，这样可以保证更高的速度。CPU有着处理指令、执行操作、控制时间、处理数据四大作用

CISC和RISC

精简指令集和复杂指令集

最初的程序都是面向CPU直接用汇编来写程序，这一时期也非常的朴实无华，没有那么多花哨的概念，什么面向对象啦，什么设计模式啦，统统没有，总之这个时期的程序员写代码只需要看看ISA就可以了。

CISC(复杂指令集计算机）

CISC（complexinstruction set computer，复杂指令集计算机）。

直到1970s年代，这一时期编译器还非常差劲，大部分程序还是用汇编语言写的，纯手工编。这一时期的大部分程序都在直接用汇编语言编写，因此大家普遍认为指令集应该更加丰富一些，程序员常用的操作最好都有对应的特定指令，首先如果指令集很少，那么程序员表达一些认为会多写很多汇编指令，很不方。所以大家认为高级语言中的一些概念比如函数调用、循环控制、复杂的寻址模式、数据结构和数组的访问等都应该直接有对应的机器指令。其次由于冯诺依曼体系结构的影响，“**程序应该和数据一样都作为比特保存在计算机存储设备中**”，因此代码也要占空间，

早期计算机空间很宝贵，这么小的内存要想装入更多的程序就必须仔细的设计机器指令以节省程序占据的空间，就必须要求机器指令要完成尽可能多的任务。可以看到复杂指令集是这一时期必然的选择，该指令集就这样诞生了并开始成为主流。

对于指令集中的每一条机器指令都有一小段对应的程序，这些程序存储在CPU中，这些程序都是由更简单的指令组成，这些指令就是所谓的**微代码**，Microcode。对于含有微代码设计的CPU来说，CPU直接执行的并不是机器指令，而是微代码，微代码是CPU以及机器指令的中间层，机器指令相对于微代码来说是“更高级的语言”，机器指令对程序员来说可见，但微代码对程序员来说不可见，程序员无法直接使用微代码来控制CPU。由于微代码的设计思想是将复杂机器指令在CPU内部转为相对简单的机器指令，这一过程对编译器不可见，也就是说你没有办法通过编译器去影响CPU内部的微代码运行行为，因此如果微代码出现bug那么编译器是无能为力的，你没有办法通过编译器生成其它机器指令来修复问题而只能去修改微代码本身。此外他还发现有时一些复杂的机器指令执行起来要比等价的多个简单指令要慢。

因此一个大牛被委以重任来改善微代码设计，为此他还专门发表了论文，但他后来又推翻了自己想法，认为微代码设计的复杂性问题很难解决，有问题的是微代码这种设计本身。因此，有人开始反思，是不是还会有更好的设计。即，为什么不直接用一些简单的指令来替换掉那些复杂的指令呢？

阅读扩展：

复杂指令集

精简指令集

RISC（精简指令集计算机）

RISC（reduced instruction set computer，精简指令集计算机）是一种执行较少类型计算机指令的微处理器。这样一来，它能够以更快的速度执行操作。目前常见使用RISC的处理器包括DEC Alpha、ARC、ARM、MIPS、PowerPC、SPARC和SuperH等。

复杂指令集诞生的必然

基于对程序员方便编写汇编语言以及节省代码存储空间的需要，直接促成了复杂指令集的设计，因此我们可以看到复杂指令集是这一时期必然的选择，该指令集就这样诞生了并开始成为主流。就这样经过一段时间后，人们发现了新的问题，由于单条指令复杂，设计解码机器指令的硬件成为了一件非常麻烦的事情。

精简指令集主要体现在以下三个方面：

1. **指令本身的复杂度**。精简指令集的思想其实很简单，干嘛要去死磕复杂的指令，去掉复杂指令代之以一些简单的指令。有了简单指令CPU内部的微代码也不需要了，没有了微代码这层中间抽象，编译器生成的机器指令对CPU的控制力大大增强，有什么问题让写编译器的搞就好了，显然调试编译器这种软件要比调试CPU这种硬件要简单很多。注意，精简指令集思想不是说指令集中指令的数量变少，而是说一条指令背后代表的动作更简单了。
2. **编译器**。编译器对CPU的控制力更强。在复杂指令集下，CPU会对编译器隐藏机器指令的执行细节，就像微代码一样，编译器对此无能为力。而在精简指令集下CPU内部的操作细节暴露给编译器，编译器可以对其进行控制。
3. 存储。在复杂指令集下，一条机器指令可能涉及到从**内存**中取出数据、执行一些操作比如加和、然后再把执行结果写回到内存中，注意这是在一条机器指令下完成的。但在精简指令集下，只能操作寄存器中的数据，不可以直接操作内存中的数据，也就是说这些指令比如加法指令不会去访问内存。
   精简指令集下有专用的 load 和 store 两条机器指令来负责内存的读写，其它指令只能操作CPU内部的寄存器，这是和复杂指令集一个很鲜明的区别。

X86架构和ARM架构区别

序号	架构	特点
1	ARM	主要是面向移动、低功耗领域，因此在设计上更偏重节能、能效方面
2	X86	主要面向家用、商用领域，在性能和兼容性方面做得更好

一、性能

X86结构的电脑无论如何都比ARM结构的系统在性能方面要快得多、强得多。　但ARM的优势不在于性能强大而在于效率，ARM采用RISC流水线指令集，在完成综合性工作方面根本就处于劣势，而在一些任务相对固定的应用场合其优势就能发挥得淋漓尽致。

二、制造工艺

一般而言，制造工艺的纳米数越小，能量的使用效率越高。ARM处理器使用更低的制造工艺，拥有类似的温控效果。比

进入移动行业时，Intel依然使用和台式机同样的复杂指令集架构，试图将其硬塞入给移动设备使

用的体积较小的处理器中。但是Intel i7处理器平均发热率为45瓦。基于ARM的片上系统（其中包括图形处理器）的发热率最大瞬间峰值大约是3瓦，约为Intel i7处理器的1/15。

三、操作系统的兼容性

X86系统由微软及Intel构建的Wintel联盟一统天下，垄断了个人电脑操作系统近30年，形成巨大的用户群，也深深固化了众多用户的使用习惯，同时x86系统在硬件和软件开发方面已经形成统一的标准，几乎所有x86硬件平台都可以直接使用微软的视窗系统及现在流行的几乎所有工具软件

ARM系统几乎都采用Linux的操作系统，而且几乎所有的硬件系统都要单独构建自己的系统，与其他系统不能兼容，这也导致其应用软件不能方便移植，这一点一直严重制约了ARM系统的发展和应用。GOOGLE开发了开放式的Android系统后，统一了ARM结构电脑的操作系统，使新推出基于ARM结构的电脑系统有了统一的、开放式的、免费的操作系统，为ARM的发展提供了强大的支持和动力。

四、64位计算

Intel并没有开发64位版本的x86指令集。64位的指令集名为x86-64（有时简称为x64），实际上是AMD设计开发的。Intel想做64位计算，它知道如果从自己的32位x86架构进化出64位架构，新架构效率会很低，于是它搞了一个新64位处理器项目名为IA64（已被放弃）。

AMD知道自己造不出能与IA64兼容的处理器，于是它把x86扩展一下，加入了64位寻址和64位寄存器。最终出来的架构，就是 AMD64，

成为了64位版本的x86处理器的标准。

五、功耗

功耗和工艺制程相关。ARM的处理器不管是哪家主要是靠台积电等专业制造商生产的，而Intel是由自己的工厂制造的。

X86汇编指令集

寄存器结构

imgX86处理器中有8个32位的通用寄存器。由于历史的原因，EAX通常用于计算，ESP指示栈

指针(用于指示栈顶位置)，而EBP则是基址指针（用于指示子程序或函数调用的基址指针）。

数据表示

在X86汇编语言中用汇编指令.DATA声明静态数据区（类似于全局变量），数据以单字节DB、双字节DW,、或双字（4字节）DD的方式存放。还可以声明连续的数据和数组，声明数组时使用DUP关键字

Z整数	**DD 1, 2, 3**	**声明三个4比特的值，分别为123，3就可以使用Z的地址+8找到**
bytes	DB 10 DUP(?)	声明10个未初始化字节
arr	DD 100 DUP(0)	从位置arr开始声明100个4字节字，全部初始化为0
str	DB 'hello',0	声明从地址str开始的6个字节，初始化为hello和null（0）字节的ASCII字符值。

基本的操作指令（包括数据传送指令、逻辑计算指令、算数运算指令），以及函数的调用规则

mov指令将第二个操作数（可以是寄存器的内容、内存中的内容或值）复制到第一个操作数（寄存器或内存）。mov不能用于直接从内存复制到内存

push指令将操作数压入内存的栈中，栈是程序设计中一种非常重要的数据结构，其主要用于函数调用过程中，其中ESP只是栈顶。在压栈前，首先将ESP值减4（X86栈增长方向与内存地址编号增长方向相反），然后将操作数内容压入ESP指示的位置

pop指令与push指令相反，它执行的是出栈的工作。它首先将ESP指示的地址中的内容出栈，然后将ESP值加4.

add指令将两个操作数相加，且将相加后的结果保存到第一个操作数中。

sub指令指示第一个操作数减去第二个操作数，并将相减后的值保存在第一个操作数

inc, dec— Increment, Decrement， inc,dec分别表示将操作数自加1，自减1

and, or, xor— Bitwise logical and, or and exclusive or，逻辑与、逻辑或、逻辑异或操作指令，

用于操作数的位操作，操作结果放在第一个操作数中。

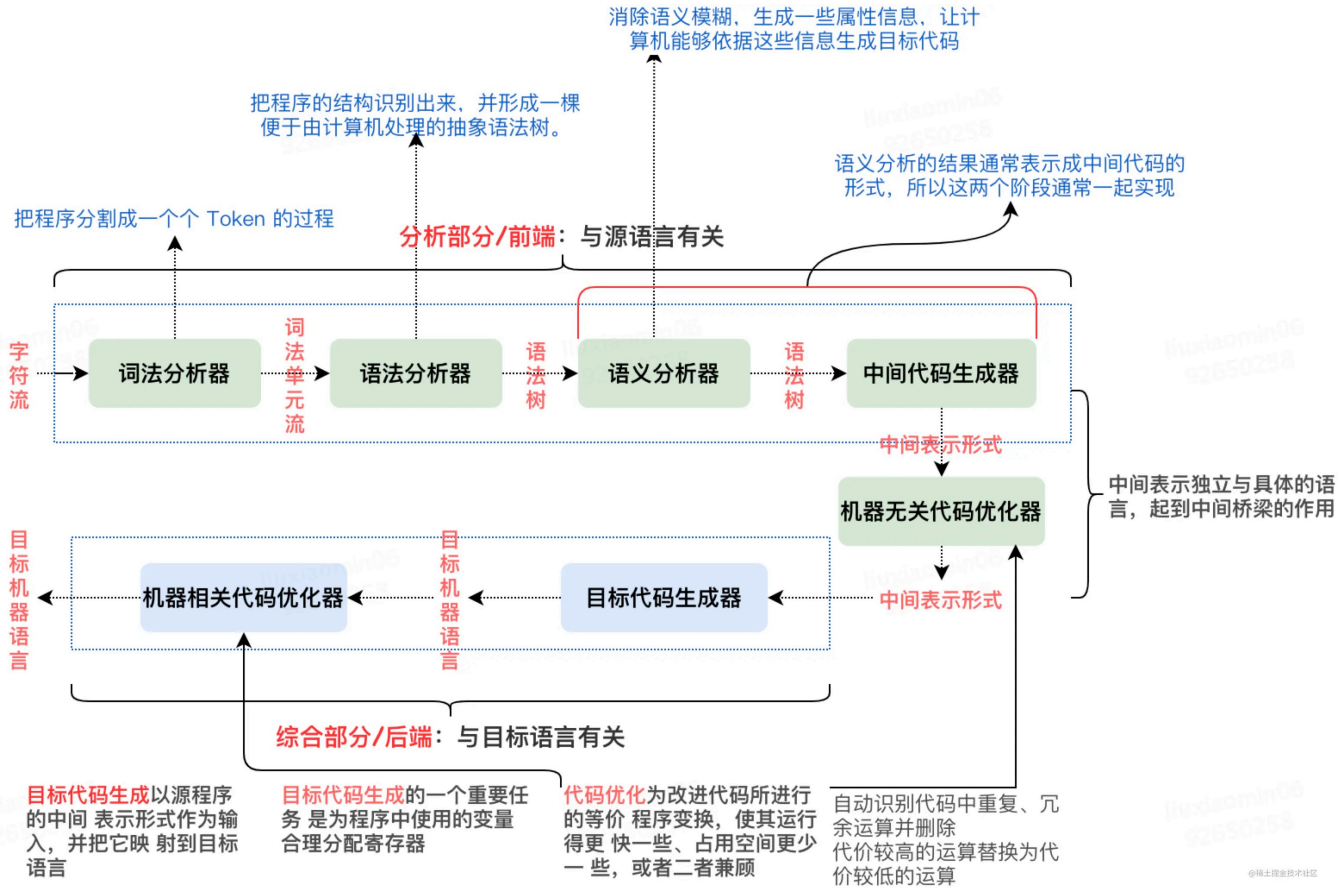
not— Bitwise Logical Not，位翻转指令，将操作数中的每一位翻转，即0->1, 1->0。

jmp — Jump，控制转移到label所指示的地址，（从label中取出执行执行）

编译原理

参考链接

编译器结构



词法分析

词法分析是编译的第一个阶段，从左到右逐行扫描源程序的字符，识别出各个单词(是高级语言中有是在意义的最小语法单元，由字符构成)，然后确定单词的类型。将识别的单词转换成统一的机内表示即词法单元 简称Token，其表示如下：

$token$:< 种别码，属性值 >，token有两个分量，种别码和属性

	单词类型	种别	种别码
1	关键字	program、if、else、then、...	一词一码
2	标识符	变量名、数组名、记录名、过程名、...	多词一码
3	常量	整型、浮点型、字符型、布尔型、...	一型一码
4	运算符	算术（+ - * / ++ --） 关系（> < == != >= <=） 逻辑（& | ~）	一词一码 或 一型一码
5	界限符	; ( ) = { } ...	一词一码

- **一词一码**：关键字是唯一的且事先可以确定，为每个关键字分配一个种别码
- **多词一码**：标识符统一作为一类单词分配同一个种别码，为了区分不同的标示符，用token的第二个分量“属性值”存放不同标示符具体的字面值
- **一型一码**：不同类型的常量他们的构成方式是不同的，例如，我们为每种类型的常量分配一个种别码，为了区分同一类型下的不同常量，也用token的第二个分量“属性值”存放每个常量具体的值 下面图中是一个词法分析后得到的token序列的例子

描述词法规则的有效工具是正规式和有限自动机。

正规式

正规式:用来确定单词是否和程序语言规范。

首先程序语言都有一定的词法规则，按照这些词法规则产生的单词符号都是一些特殊的字符串，
因此，可以形式化地描述词法规则，叫做正规式，对应的单词集合称为正规集

正规集是一个集合，而正规式是表示正规集的一种方法。不同正规式也可以表示同一个正规集，即正规式与正规集之间是多对一的关系。若正规式P和Q表示了同一个正规集，则称P和Q是等价的，记为P=Q

有限自动机

有限自动机：通过有限自动机进行单词和正规式比较

- NFA: Nondeterministic Finite Automaton不确定的有限自动机。

> - DFA: Deterministic Finite Automaton确定的有限自动机，DFA是NFA的一个特例

语法分析

语法分析器从词法分析器的输出中识别出各类短语，并构造语法分析树(parse tree),，语法分析树描述了句子的语法结构

语法分析的方法：

1. **推导**：最左推导、最右推导
2. **归约**：最右归约、最左归约,推导的逆过程就是归约

语法树什么的

语义分析

主要由两个任务

收集检查属性信息

比如变量属性，看你是简单变量还是符合变量（数组）；类型，看你是整型，字符型，布尔型还是指针；存储位置、存储长度；值；作用域；参数和返回信息什么的。

语义检查

1. 看看变量或过程未经声明就使用
2. 变量或过程名重复声明
3. 运算分量类型不匹配
4. 操作符与操作数之间的类型不匹配
    - 数组下标不是整数
    - 对非数组变量使用数组访问操作符
    - 对非过程名使用过程调用操作符
    - 过程调用的参数类型或数目不匹配
    - 函数返回类型有误

生成中间代码

通常和语义分析一起实现。对语法分析识别出的各类语法范畴，分析他的含义，进行初步翻译，产生介于源代码和目标代码质检的一种代码

代码优化

对前面生成的中间代码进行加工变换，以便在最后极端产生更为高效的目标代码，需要遵循等价变换的原则，优化的方面包括：公共子表达式的提取、合并已知量、删除无用语句、循环优化。

目标代码生成

把经过优化的中间代码转化成特定机器上的低级语言

出错处理

如果源程序有错误，编译程序应设法发现错误并报告给用户。由专门的出错处理程序来完成。错误类型：

- 语法错误：在词法分析和语法分析阶段检测出来
- 语义错误：一般在语义分析阶段检测
- 逻辑错误：不可检测，比如死循环，一般不处理因为没办法在编译阶段检测出来

GPU架构

参考链接

概述

GPU全称是**Graphics Processing Unit**，图形处理单元。它的功能最初与名字一致，是专门用于绘制图像和处理图元数据的特定芯片，后来渐渐加入了其它很多功能。GPU自从上世纪90年代出现雏形以来，经过20多年的发展，已经发展成不仅仅是渲染图形这么简单，还包含了数学

计算、物理模拟、AI运算等功能。

物理结构

由于纳米工艺的引入，GPU可以将数以亿记的晶体管和电子器件集成在一个小小的芯片内。从宏观物理结构上看，现代大多数桌面级GPU的大小跟数枚硬币同等大小，部分甚至比一枚硬币还小。

NVidia Tesla架构、NVidia Fermi架构、NVidia Maxwell架构

图形渲染原理

graphics pipeline

Vulkan

DirectX

OpenGL