**cmd**

**进程/端口**

taskkill /im chrome.exe /f其中/f表示强制终止，/im表示后边是进程名称，例如“taskmgr.exe"

taskkill /f /pid 3352 其中，/pid表示后边为进程号pid

winsock下，如果两个进程都设置setsockopt(sClient, SOL\_SOCKET, SO\_REUSEADDR, (char \*)&on, sizeof(on))则可bind同一个端口，第一个设置，第二个不设置或第二个设置、第一个不设置都不可以，第二个bind的时候会失败

start chrome.exe

start powershell.exe

netstat –ano

netstat –ano | findstr 6666

查看端口占用否：netstat -aon | findstr 1044 查看1044端口被占用情况

-a 显示所有连接和侦听端口。

-n 以数字形式显示地址和端口号

-o 显示拥有的与每个连接关联的进程 ID

**增删查改文件/目录：**

切换到E盘：e:

创建新文件 cd .> hello.cpp

cd.表示改变当前目录为当前目录，即等于没改变；而且此命令不会有输出

cls清屏

type查看文件内容，相当于linux的cat

复制文件copy hello.cpp world.cpp

删除文件del hello.cpp

move a.txt ../b.txt重命名/移动

5s内重启命令：shutdown -r -t 5

mkdir a\b\c支持递归创建目录，递归即指父级目录不存在时自动创建

（linux中递归创建目录：递归创建目录：mkdir -p a/b/c）

要删除目录： rmdir /s/q log

其中：

/s 是代表删除所有子目录跟其中的档案。

/q 是不要它在删除档案或目录时，不再问我 Yes or No 的动作。

**共享资源**

查看本机能访问所有共享资源：net use

断开与所有共享资源的连接：net use \* /del

连接共享机器：net use \\192.165.56.82 /user:flymna

**tar压缩**

tar –help查看帮助

压缩文件夹tar cvf uprade.tar upgrade

解压tar包tar xvf uprade.tar

XCOPY upgrade %cd% /S /E /Y

把upgrade目录下面的所有文件/子文件/文件夹/子文件夹全部复制到当前目录，包括空的文件夹，在复制过程中不作覆盖方式提示

%cd%：当前目录

/S： 复制目录和子目录，除了空的。

/E： 复制目录和子目录，包括空的。与 /S /E 相同

/Y： 禁止提示直接覆盖现存目标

**vscode：**

**ctrl+p搜索文件**

**注释选中的内容：alt+shift+a**

**vs中(Ctrl+逗号)，根据文件名打开文件**

**Bash：**

Linux 中的 shell 有很多类型。Linux 操作系统缺省的 shell 是Bourne Again shell，它是 Bourne shell 的扩展，简称 Bash。

#!/bin/bash

# This is a very simple example

echo Hello World

Linux系统将 "#!" 及该字串后面的信息解析为脚本的解释器

如果没指定，那么就用当前系统默认的shell(大多数linux默认是bash)，如果指定了解释器，那么就将该脚本交给指定的解释器

在示例中 第一行的 "#!" 及后面的 "/bin/bash" 就表明由安装路径/bin目录下的bash程序来解释执行

在 BASH 程序中从“#”号（注意：后面紧接着是“!”号的除外）开始的一行被注释

执行，打开Git bash，输入$ bash hello 或 $ sh hello

**变量：**

#!/bin/bash   
# give the initialize value to STR  
STR="Hello World"   
echo $STR

* 变量赋值时，'='左右两边都不能有空格；
* BASH 中的语句结尾不需要分号（";"）；
* 除了在变量赋值和在FOR循环语句头中，BASH 中的变量使用必须在变量前加"$"符号，如果第三行改为 "echo STR"，则==>output: STR
* 更为标准的变量引用方式是 ${STR} 这样的，$STR只不过是对 ${STR} 的一种简化
* **BASH 中没有浮点运算**，因此也就没有浮点类型的变量可用

整数运算

#!/bin/bash

x=1999

let "x = $x + 1" #该语句双引号内的空格取消也可以

echo $x #2000

x="olympic'"$x

echo $x #olympic'2000

关于整数变量计算，有如下几种：" + - \* / % "，他们的意思和字面意思相同。整数运算一般通过 let 和 expr 这两个指令来实现，如对变量 x 加 1 可以写作：let "x = $x + 1" 或者 x=`expr $x + 1`

**变量的作用域：**

在 BASH 程序中如果一个变量被使用了，那么直到该程序的结尾，该变量都一直有效

要想使用一个局部变量，则在变量首次被赋初值时加上 local 关键字

#!/bin/bash

HELLO=hi

function hello

{

local HELLO=WORlD

year=2018

local month=7

echo $HELLO

}

echo $HELLO #hi

hello #WORLD

echo $HELLO #hi

echo $year #2018

echo $month #空

$HELLO 的值在执行函数 hello 时并没有被改变。也就是说局部变量 $HELLO 的影响只存在于函数那个程序块中

**整数和字符串比较：**

在比较操作上，整数变量和字符串变量各不相同，详见下表：

|  |  |  |
| --- | --- | --- |
| 对应的操作 | 整数操作 | 字符串操作 |
| 相同equal to | -eq | = |
| 不同not equal to | -ne | != |
| 大于greater than | -gt | > |
| 小于less than | -lt | < |
| 大于或等于 greater equal | -ge |  |
| 小于或等于less equal | -le |  |
| 为空 |  | -z |
| 不为空 |  | ! -z |

比较字符串 a 和 b 是否相等就写作：if [ $a = $b ]

判断字符串 a为空就写作： if [ -z $a ]

判断字符串不为空：if [ -n $a ]

判断整数变量 a 是否大于 b 就写作：if [ $a -gt $b ]

#!/bin/bash

a="hello"

b="hello"

if [ $a=$b ]

then

echo 'are equal' #are equal

fi

c=10

d=$c

if [ ! $c -eq $d ]

then

echo 'c!=d'

else

echo 'c==d' #c==d

fi

判断字符串是否为空：

* str1=""
* if [ -z $str1 ] #一定不能忘记$，否则永远是非空字符串
* then
* echo '空字符串' #空字符串
* fi
* str2="-"
* if [ ! -z $str2 ]
* then
* echo '非空字符串' #非空字符串
* fi

**Bash操作文件：**

|  |  |
| --- | --- |
| 运算符 | 含义（ 满足下面要求时返回 TRUE ） |
| -e file | 文件 file 已经存在 |
| -f file | 文件 file 是普通文件 |
| -d file | 文件 file 是一个目录 |
| -r file | 文件 file 对当前用户可以读取 |
| -w file | 文件 file 对当前用户可以写入 |
| -x file | 文件 file 对当前用户可以执行 |
| -g file | 文件 file 的 GID 标志被设置 |
| -s file | 文件 file 大小不为零 |
| -u file | 文件 file 的 UID 标志被设置 |
| -O file | 文件 file 是属于当前用户的 |
| -G file | 文件 file 的组 ID 和当前用户相同 |
| file1 -nt file2 | 文件 file1 比 file2 更新 |
| file1 -ot file2 | 文件 file1 比 file2 更老 |

注意：上表中的 file 及 file1、file2 都是指某个文件或目录的路径

if [ -x /root ] 可以用于判断 /root 目录是否可以被当前用户进入



**字符串单/双引号及不加引号的区别：**

单引号：

　　可以说是所见即所得：即将单引号内的内容原样输出，或者描述为单引号里面看见的是什么就会输出什么。

双引号：

　　把双引号内的内容输出出来；如果内容中有命令，变量等，会先把变量，命令解析出结果，然后再输出最终内容来。

无引号：

如果内容中有命令，变量等，会先把变量，命令解析结果，然后在输出最终内容来；

如果字符串中带有空格等特殊字符，则不能完整的输出，需要改加双引号，一般连续的字符串，数字，路径等可以用，不过最好用双引号替代之

#!/bin/bash

hello='hi'

echo '$hello world' #$hello world

echo "$hello world" #hi world

echo "hello world" #hi world

**if语句：**

if [ …… ]，注意，中括号里的内容与左右括号必须有空格

if [ expression ]  
then  
statements  
fi

或者

if [ expression ]  
then  
statements  
else  
statements  
fi

或者

if [ expression ]  
then  
statements  
elif [ expression ]  
then   
statements  
else  
statements   
fi

**for循环：**

Bash的for循环类似Python

在for循环语句头中，变量不需要$

in后边是需要遍历的列表

do/done 对包含了循环体，相当于 C++ 语言中的一对大括号

#!/bin/bash

for day in Sun Mon Tue Wed Thu Fri Sat

do

echo $day

done

**函数：**

格式：unction my\_funcname

{

code block

}

或者

my\_funcname()

{

code block

}

BASH 中函数参数的定义并不需要在函数定义处就制定，而只需要在函数体中用 BASH 的保留变量 $1 $2 ... 来引用就可以了；BASH 的返回值可以用 return 语句来指定返回一个特定的整数，如果没有 return 语句显式的返回一个返回值，则返回值就是该函数最后一条语句执行的结果（一般为 0，如果执行失败返回错误码）。函数的返回值在调用该函数的程序体中通过 $? 保留字来获得

$1表示第一个参数，$2表示第二个参数...

$10 不能获取第十个参数，获取第十个参数需要${10}。当n>=10时，需要使用${n}来获取参数

$# 传递到脚本的参数个数

$\* 以一个单字符串显示所有向脚本传递的参数

函数不需要定义在程序开头

#!/bin/bash

square()

{

let "res=$1\*$1"

return $res

}

multip()

{

let "res=$1\*$2"

return $res

}

square 3

echo "result is $? " #result is 9

multip 3 5 6

echo "result is $? " #result is 15

set指令能设置所使用shell的执行方式

set –e：

DIR="$( cd "$(dirname "$0")" ; pwd -P )"

解释一下。

$0 类似于*python*中的*sys.argv[0]*等。 $0指的是*Shell*本身的文件名。类似的有如果运行脚本的时候带参数，那么$1 就是第一个参数，依此类推。   
dirname 用于取指定路径所在的目录 ，如 *dirname /home/ikidou* 结果为 */home*。   
$ 返回该命令的结果   
pwd -P 如果目录是链接时，格式：*pwd -P* 显示出实际路径，而非使用连接（*link*）路径。

**shell 命令：**

**系统信息**

cat etc/centos-release 输出CentOS Linux release 7.8.2003 (Core)

cat /proc/version

输出Linux version 3.10.0-1127.10.1.el7.x86\_64 (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red Hat 4.8.5-39) (GCC) ) #1 SMP Wed Jun 3 14:28:03 UTC 2020

**登录用户w命令**

**w命令查看登录用户信息**

* 用户名称
* 用户的机器名称或tty号
* 远程主机地址
* 用户登录系统的时间
* 空闲时间（作用不大）
* 附加到tty（终端）的进程所用的时间（JCPU时间）
* 当前进程所用时间（PCPU时间）
* 用户当前正在使用的命令



**CPU硬件信息：**

[root@localhost zcj]$ lscpu

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 112

On-line CPU(s) list: 0-111

Thread(s) per core: 2

Core(s) per socket: 28

座： 2

座即Socket，cpu的插槽，即几个物理cpu

Core per Socket：每一个插槽对应的物理CPU上有多少个核

Thread per Core：每个核上有多少个线程

看个图：（几核几线程就是指有多少个“Core per Socket”多少个“Thread per Core”,当后者比前者多时，

说明启用了超线程技术）

或者：

[root@localhost zcj]$ cat /proc/cpuinfo | grep "physical id" | sort -u

physical id : 0

physical id : 1

两个物理cpu

[root@localhost zcj]$ cat /proc/cpuinfo | grep "cpu cores" | sort -u

cpu cores : 28

每个cpu28个物理核心

[root@localhost zcj]$ cat /proc/cpuinfo | grep "siblings" | sort -u

siblings : 56

每个cpu56个逻辑核，因此一个物理核心对应两个逻辑核

**top查看cpu进程信息：**

在top运行时，可以通过按“H”键将线程查看模式切换为开或关

在top运行时，可以通过按数字“1”键查看每个逻辑cpu的使用率

输出某个特定进程<pid>内运行的线程状况：top -Hp <pid>

top命令的显示信息：



第 1 行：



第 2~3 行为进程和CPU的信息：



IRQ: IRQ全称为Interrupt Request，即是“中断请求”的意思

统计信息区域的下方显示了各个进程的详细信息：

|  |  |  |
| --- | --- | --- |
| 序号 | 列名 | 含义 |
| a | PID | 进程id |
| b | PPID | 父进程id |
| e | USER | 进程所有者的用户名 |
| h | PR | 优先级 |
| i | NI | nice值。负值表示高优先级，正值表示低优先级 |
| k | %CPU | 上次更新到现在的CPU时间占用百分比 |
| m | TIME+ | 进程使用的CPU时间总计，单位1/100秒 |
| o | VIRT | 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES |
| q | RES | 进程使用的、未被换出的物理内存大小，单位kb。RES=CODE+DATA |
| t | SHR | 共享内存大小，单位kb |
| w | S | 进程状态。 **D**=不可中断的睡眠状态 **R**=运行 **S**=睡眠 **T**=跟踪/停止 **Z**=僵尸进程 |
| x | COMMAND | 命令名/命令行 |

**Linux线程占核分析：**

找cpu核对应的线程对应的代码：

1、top -Hp <进程号>

2、按下F

3、通过上下找到P这个项，按空格，使之显示\*被选中

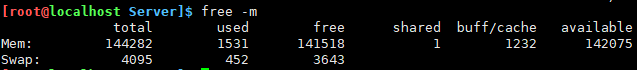
4、按ESC

5、右边的P显示的是线程运行的cpu核序号

6、gdb attatch 进程号

7、看线程堆栈，确定哪段代码是性能瓶颈

**内存总使用情况free -m**



主要看第一行Mem 总共 144282 M , 使用了1531 M , 剩余空闲 141518 M

**进程/端口**

查看端口占用：netstat -ntulp | grep 123

-t显示TCP传输协议的连线状况

-u显示UDP传输协议的连线状况

-l显示listening的服务器的Socket

-p显示正在使用Socket的程序识别码和程序名称

-n直接显示ip、port的数值而不是域名、服务

-a：display all sockets (default: connected)



ps命令是Process Status的缩写

ps -ef | head -10 //显示10行

ps -ef | grep rafias查看进程

后台跑任务：nohup ./test.sh &

终止进程：

杀前台进程：Ctrl+C、Ctrl+Z，再ps -ef | grep Media查看，有可能变为僵尸进程，杀掉

杀后台进程：执行 kill -9 +进程号

杀僵尸进程：ps -ef | grep defunct 后面尖括号里是defunct的都是僵尸进程。 在查看僵尸进程时，第三列就是父进程的pid，想要结束僵尸进程，只需kill掉父进程即可

kill -2 pid # 就是ctrl + c

kill -15 pid #通知进程"安全、干净的退出"，程序接到信号之后，退出前一般会进行一些"准备工作"，如资源释放、临时文件清理等，此时如果遇到阻塞或者其他问题导致无法成功，则应用程序可以选择忽略该终止信号。

kill -9 pid #强制终止进程，不能被阻塞或者忽略，可能有副作用

杀所有进程：killall -9 MediaTransform 其中，MediaTransform为进程名字

**服务器开放端口5656：**

iptables -L INPUT --line-numbers

-L INPUT表示罗列INPUT链中所有规则

--line-numbers表示在罗列的各条记录前加上序号1、2、3……

iptables -I INPUT 10 -p tcp --dport 5657 -j ACCEPT

-I INPUT 10表示向INPUT链插入一条规则，插入位置为10

-p tcp表示要插入的规则针对tcp协议

--dport 5656–dport：指定目标端口（destination port 目的端口）

-j ACCEPT指定动作类型为ACCEPT

**/proc/PID查看进程信息：**

Linux在启动一个进程时，系统会在/proc下创建一个以PID命名的文件夹，在该文件夹下会有进程信息，因此ll /proc/17898可查看进程17898的信息

* cwd符号链接的是进程运行目录；
* exe符号链接的是执行程序的绝对路径；
* cmdline就是程序运行时输入的命令行命令；
* environ记录了进程运行时的环境变量；

**ntp服务：**

启动ntp服务：service ntpd start

查看ntp的端口： netstat -nlptu | grep ntp 可以看到123端口

查看ntp服务状态：service ntpd status

停止ntp服务：service ntpd stop

**ping、curl、查看ip、域名、抓包**

查看域名对应的ip：

host partnerdemo.io.zixi.com

nslookup partnerdemo.io.zixi.com

ping partnerdemo.io.zixi.com

拉取网页内容：

curl partnerdemo.io.zixi.com，有时候浏览器可以打开，但禁止ping，这时候可以curl命令判断能并不能访问

**查看符号表、依赖库：**

nm -D input.apf查看符号表

ldd input.apf查看依赖库

**tcpdump抓包：**

tcpdump -i eno1 port 8888 -nn

指定目的ip：

cpdump -i eno1 "dst 192.165.153.183" -nn

指定源ip：

tcpdump -i eno1 "src 192.165.153.186" -nn

指定源ip源端口：

tcpdump -i eno1 "src 192.165.153.186 and src port 49134" -nn

指定源ip目的端口：

tcpdump -i eno1 "src 192.165.153.186 and dst port 8888" -nn

指定源ip、源或目的端口：

tcpdump -i eno1 "src 192.165.153.186 and port 49134" -nn

从所有网卡中捕获所有数据包：

tcpdump -i any

指定网卡抓包：

ifconfig查看网卡信息，发现有enp24s0f0、enp24s0f1两个网卡

tcpdump -i enp24s0f0 port 14153 -nn -w a.pcap

捕获本机发往本机的包：

tcpdump -i lo

要获取指定IP的数据包，不管是作为源地址还是目的地址，使用下面命令：

tcpdump host 192.168.1.100

tcpdump -i any port 2077 -nn

-nn参数以数值形式显示ip、port而不是解析为域名服务等

抓包保存：

tcpdump -i any port 9999 -nn -w cap.pcap

**增删查改文件/目录：**

cd

返回上一级 cd ..

cd / 返回到linux系统根目录

cd – 返回之前的目录

cd 返回到/home/zhangchangjian

递归创建目录：mkdir -p a/b/c

pwd

复制一个文件，并重命名：cp First.py cop.py

rm命令后边的参数：

-r 就是向下递归，不管有多少级目录，一并删除

-f 就是直接强行删除，不作任何提示的意思

（cmd的taskkill /f /im a.exe命令中，/f也表示强制）

rm a.txt：如果有a.txt则删除，没有则报错返回错误信息

rm –f a.txt：如果有a.txt则删除，没有则什么都不做不返回错误信息

mv src dst重命名文件

删除一个文件夹：rm -r saved

移动一个文件夹：mv saved ../thu0719

如果thu0719目录已存在，则将名为saved的文件夹移动至thu0719文件夹下；

如果thu0719目录不存在，则将名为saved的文件夹移动至../文件夹下，并重新命名为thu0719

查看目录树形结构：tree .git 其中.git为目录名

动态查看文件的最后100行：tail -100f sys.log

动态监测某个命令的执行结果，如监测.git目录的变化，则先cd到该目录，然后执行

watch -n 1 -d find .

其中-n 1每隔1秒更新一次

-d 高亮显示差异

find .是要监测的命令，表示递归查找当前目录下的所有文件和子目录

readlink命令：readlink -f  file\_path：获取一个文件或目录的绝对路径



**删除顽固文件：**

chattr -i -a /usr/bin/xrig

rm -rf /usr/bin/xrig

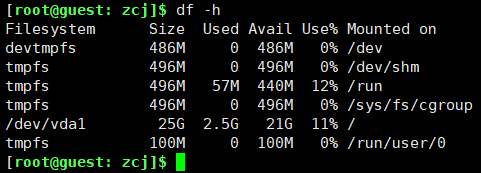
**字符串处理：**

cut -d '.' -f 3 命令：按符号.分割，取分割后的第3个片段



echo "192.165.56.97" | cut -d '.' -f 3 //输出56

**磁盘信息：**



**PS1变量设置：**

export PS1="\[\e[31;1m\][\u@\[\e[32;1m\]\h \[\e[34;1m\]\W]$\[\e[0m\] "



**secureCRT中：**

* clear命令、这个命令将会刷新屏幕，本质上只是让终端显示页向后翻了一页，如果向上滚动屏幕还可以看到之前的操作信息。
* 点Edit菜单，然后再选"Clear Screen and Scrollback"，真正清屏，而且把缓冲也清了。
* 在命令行窗口选中即复制; 在命令行窗口中右键即粘贴

**查找统计文件：**

find -name "\*.cpp"

递归查找当前目录下所有以”.cpp”结尾的文件和目录

如果要指定目录，则find /home/ –name "\*.cpp"

如果不递归，仅查找当前目录，则find -maxdepth 1 -name "\*.cpp"，maxdepth指定深度

find \* //递归地列出当前目录下所有文件及目录

（ls –l | grep "^-"| wc -l）查看某个文件夹下文件的个数。  
（ls –lR | grep "^-"| wc -l）查看某个文件夹下文件的个数，包括子文件夹下的文件个数。  
（ls –l | grep "^d"| wc -l）查看某个文件夹下文件夹的个数。  
（ls –lR | grep "^d"| wc -l）查看某个文件夹下文件夹的个数，包括子文件夹下的文件夹个数。  
（ls –l | wc -l）查看文件夹下所有的文件和文件夹。也就是统计ls -l命令所输出的行数

wc [选项] [文件]

选项有：

-c 统计字节数

-l 统计行数

-m 统计字符数，不能和-c同时使用

-L 打印最长行的长度

-w 统计字数

统计文件行数：wc -l blur\_data\_ldmk.txt

wc myfile.txt 默认的不带参数是会显示3个属性，分别为行数、单词数、字节数

type 后能够使用的类型有

f 常规文件

d 目录

l 连接

c 字符设备文件 character devices

b 块设备文件 block devices

比如要查找系统中所有以 .conf 结尾的文件

find / -type f -name "\*.conf"

统计指定后缀名的文件总个数命令：   
find . -name "\*.cpp" | wc -l

第一个‘|’是竖线，第二个’l’是line的首字母

**zip、tar压缩解压：**

zip压缩命令：

zip blur.zip blur/\*

zip -r blur.zip blur/ 当文件夹下文件很多时采用

压缩zip -q -r Server.zip Server/ 递归压缩整个目录

解压unzip Server.zip

-q:不显示指令执行过程

把一个文件abc.txt和一个目录dir1压缩成为yasuo.zip：

＃ zip -r yasuo.zip abc.txt dir1

有一个很大的压缩文件large.zip，不想解压缩，只想看看它里面有什么：

# unzip -v large.zip

下载了一个压缩文件large.zip，想验证一下这个压缩文件是否下载完全了

# unzip -t large.zip

用-v选项发现music.zip压缩文件里面有很多目录和子目录，并且子目录中其实都是歌曲mp3文件，我想把这些文件都下载到第一级目录，而不是一层一层建目录：

# unzip -j music.zip

tar -zxvf xxx.tar.gz 解压

只查看目录结构(不解开压缩文件)：tar -tvf  sns.tar.gz | grep ^d

tar文件有5个文件，但仅仅想解压其中一个文件或目录：tar -xvf data.tar /opt/oracle/product/10.2.0/a.txt

**rar压缩解压：**

在Linux下原本是不支持rar文件的，需要安装linux下的rarlinux

压缩文件夹时一定要带-r参数（压缩单个文件不用）否则不能对文件夹里的子目录作用

添加文件夹到压缩包，不存在压缩包则创建：rar a –r 压缩包名 文件夹名





解压到当前目录：unrar x \*.rar ./

解压到特定目录：unrar x \*.rar path

解压缩时，如果有同名文件，则会询问是否覆盖

**shell历史记录**

列出目前最近的5条记录： history 5

[root@guest: zcj]$ history 5

366 wc a.cpp

367 ll | grep "^-" | wc -l

368 ll

369 ll -R

370 history 5

立刻将目前的资料写入histfile当中：history -w

默认情况下，会将历史纪录写入~/.bash\_history当中！

在练习linux 的时候同时开好几个bash 介面，这些bash 的身份都是root 。这样会有~/.bash\_history 的写入问题吗？因为这些bash 在同时以root 的身份登入， 因此所有的bash 都有自己的1000 笔记录在记忆体中。因为等到登出时才会更新记录， 最后登出的那个bash 才会是最后写入的资料。如此一来其他bash 的指令操作就不会被记录下来了 (其实有被记录，只是被后来的最后一个bash 所覆盖更新了)

“回车”就是CHAR(13)，即\r；“换行”就是CHAR(10)，即\n

Unix系统里，每行结尾只有"<换行>"，即"\n"；Windows系统里面，每行结尾是"<回车><换行>"，即"\r\n"；Mac系统里，每行结尾是"<回车>"。一个直接后果是，Unix/Mac系统下的文件在Windows里打开的话，所有文字会变成一行；而Windows里的文件在Unix/Mac下打开的话，在每行的结尾可能会多出一个^M符号

**文件传输sz/rz、scp：**

在服务器上使用rz上传本地的文件到服务器时，出现一坨乱码，并报如下错误：

“Zmodem transfer canceled by remote side ”

加上-be参数即可解决问题：

-b 以二进制方式传输（binary）。

-e 对控制字符转义（escape），这可以保证文件传输正确。

scp下载文件到本地：

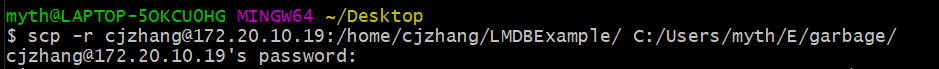
本地为Windows系统，则打开Git Bash，输入指令



cjzhang@172.20.10.19 表示使用cjzhang用户登录远程服务器172.20.20.19

输入完命令后，回车确定，会提示输入用户cjzhang的密码

scp下载文件夹到本地：



本地地址在前，服务器地址在后，则是上传



启用压缩，传输文件夹：scp –r –C 源 目的

**vim编辑器：**

查找字符串GetPlatform，底行命令模式输入：/GetPlatform，回车确定

再按n（小写）查看下一个匹配

按N(大写）查看上一个匹配

所有的Unix Like系统都会内建vi文本编辑器，其他的文书编辑器则不一定会存在。Vim是从 vi 发展出来的一个文本编辑器，目前使用比较多



命令模式按i进入输入模式，按ESC退出，按:在底行可以输入命令

不保存退出：切换到底线命令模式下，输入q!命令

使用vim时，按了Ctrl + s后，vim“卡住”。退出这种状态，按下Ctrl + q

正常保存并退出：在底线命令模式下，输入wq命令。

只读文件wq!会报错'readonly' option is set (add ! to override)，则输入:wq!强制修改或者在启动vim时直接加上sudo，即sodu vi First.py

命令模式下光标移动

上下左右箭头可用

翻半页:ctrl+d向下翻半页 ctlr+u向上翻半页（d=down u=up）

数字+G：移动到第几行

0：移动到行首(^：移动到行首第一个非空字符)

$：移动到行末

u 撤销上一步的操作

Ctrl+r 恢复上一步被撤销的操作

:set nu //显示行号

:set nonu //取消显示行号

:%d //清空文件内容：

yy复制光标所在的那一行

p将复制的数据，粘贴在光标的下一行

H：光标移动到这个屏幕的最上方那一行的第一个字符,H你就把它记成是header

M：光标移动到这个屏幕的中央那一行的第一个字符,M你就把它记成middle的缩写

L：光标移动到这个屏幕的最下方那一行的第一个字符,L你就把它记成last的缩写

向下移动一行，可用<enter>；向下移动那个多行：数字+<enter>

向后移动一个字符，可用<space>；向后移动多个字符：数字+<space>

x ：每按一次，删除光标所在位置的后面一个字符

#x ：删除光标所在位置的后面#个字符，例如， 6x 表示删除光标所在位置的后面6个字符

dd删除光标所在的那一整行

yy复制光标所在的那一行

数字+dd：删除光标所在行向下 n 行，例如 20dd 则是删除 20 行

数字+yy：复制光标所在行向下 n 行，例如 20yy 则是复制 20 行

dG：删除光标所在行到最后一行的所有数据

yG：复制光标所在行到最后一行的所有数据

p将复制的数据，粘贴在光标的下一行；P上一行

d1G：删除光标所在行到第一行的所有数据

y1G：复制光标所在行到第一行的所有数据

vim的删除，即剪切

* $ screen #这样就可以新建窗口，进入到一个窗口中，但是这样窗口就没有名字，无法区分他们
* $ screen -S name #这样新建一个名字为name的窗口，并入到该窗口中   
  例如：screen -S count 新建了一个叫count的窗口并进入
* $ screen command #这样新建一个窗口并在窗口中执行command，同样没有名字   
  例如：screen python ./a.py 新建并执行a.py程序

screen窗口下，ctrl a+d，把 screen session detach 掉后，会回到还没进 screen 时的状态，实现任务与当前窗口分离，让服务器在后台运行程序

想恢复窗口，screen -ls -> 显示所有的 screen sessions

screen -r [keyword] -> 选择一个screen session 恢复对话



杀死一个会话窗口kill -9 threadnum

窗口被杀死后，再用screen -ls 可以看到该窗口后面的(???dead)字样，说明窗口死了，但是仍在占用空间。这时需要清除窗口

$ screen -wipe #自动清除死去的窗口



vi分屏：

使用大写的O参数来垂直分屏。

vim -On file1 file2 ...

使用小写的o参数来水平分屏。

vim -on file1 file2 ...

注释: n是数字，表示分成几个屏

要在各个屏间切换，只需要先按一下Ctrl+W，然后上下左右

关闭一个窗口，ctrl+w c

上下分割，并打开一个新的文件:sp filename

左右分割，并打开一个新的文件:vsp filenam

向右移动：Ctrl+W L

向左移动：Ctrl+W H

vim全局替换：

语法为 :[addr]s/源字符串/目的字符串/[option]

1,7 指從第一行至第七行，1,$ 指從第一行至最後一行，也就是整篇文章，也可以 % 代表

全局替换命令为：:%s/源字符串/目的字符串/g

如，21到42行的cout替换为print

:21,42s/cout/print/g

文件中所有的good替换为bad

:%s/good/bad/g

如果要替换的字符串中有/，则用#分割四部分

每行的行首都添加一个字符串：%s/^/要插入的字符串

每行的行尾都添加一个字符串：%s/$/要插入的字符串

例如，每行末尾添加空格再添加”label”： :%s/$/ label/g

0：跳转到行首

^：跳转到行首

$：跳转到行末

vim多行注释和取消注释

1.多行注释：

* 进入命令行模式，按ctrl + v进入 visual block纵向编辑模式
* 按上、下、左、右选中目标
* 按大写字母I，再插入注释符，例如#
* 按esc键就会全部注释了

2.取消多行注释：

* 进入命令行模式，按ctrl + v进入纵向编辑模式 visual block模式，
* 按上、下、左、右选中目标
* 按d键就可全部取消注释

列模式下，可选中某几列的内容



纵向编辑模式选中某几列后，r进入修改模式，d删除所选，大写字母I进入插入模式

底行命令模式输入“：32,65d”,回车键，32-65行就被删除了

底行命令模式输入“：32,65y”,回车键，32-65行就被复制了

鼠标左键双击一个单词，则复制之

在终端Vim中粘贴代码时，发现插入的代码会有多余的缩进，而且会逐行累加。原因是终端把粘贴的文本存入键盘缓存（Keyboard Buffer）中，Vim则把这些内容作为用户的键盘输入来处理。导致在遇到换行符的时候，如果Vim开启了自动缩进，就会默认的把上一行缩进插入到下一行的开头，最终使代码变乱

在~/.vimrc中添加set pastetoggle=<F9>

以后在插入模式下，只要按F9键就可以切换自动缩进

**stderr和stdout重定向：**

标准输入(stdin)默认为键盘输入；

标准输出(stdout)默认为屏幕输出；

标准错误输出(stderr)默认为屏幕输出（上面的std表示standard）。

标准输入的文件描述符为0，标准输出的文件描述符为1，标准错误输出的文件描述符为2

">"是标准输出重定向的代表符号， ">>" 则表示不清除原来的而追加输出

标准错误输出重定向2>，2>>则表示追加

标准错误输出重定向到标准输出：2>&1

常用命令：

* find /home -name lost\* 2>err.log，标准错误输出（如因为访问权限限制）被存入err.log文件中
* find /home -name lost\* > err.log，标准输出被存入err.log文件中
* find /home -name lost\* >err.log 2>&1，标准错误输出和标准输出都被存入到文件中。（首先将标准错误输出重定向到标准输出中，再将标准输出重定向到文件）
* cat Example.txt |tee out.txt #将cat的标准输出即输出到屏幕，又保存到文件
* ll dir 2>&1 | tee out.txt错误输出和标准输出都是既显示于屏幕又重定向到文件。

合并用cat file1 file2 > file\_merged

**nvidia-smi查看GPU：**



Fan：风扇转速，从0到100%之间变动，这个速度是计算机期望的风扇转速，实际情况下如果风扇堵转，可能打不到显示的转速

Perf：是性能状态，从P0到P12，P0表示最大性能，P12表示状态最小性能。

Pwr：是能耗

Persistence-M：是持续模式的状态，持续模式虽然耗能大，但是在新的GPU应用启动时，花费的时间更少，这里显示的是off的状态

Memory Usage是显存使用率

GPU-Util：浮动的GPU利用率

**权限和umask：**

chmod 751 file   ：给file的属主分配读、写、执行(7，111)的权限，给file的所在组分配读、执行(5，101)的权限，给其他用户分配执行(1，001)的权限

umask 0022：对文件则是rw- rw- rw-减去--- -w- -w-，最终新建的文件的权限是-rw-r--r--

对目录则是rwxrwxrwx减去--- -w- -w-，最终新建的目录的权限是drwxr-xr-x

umask 0123，对目录是rwxrwxrwx减去--x -w- -wx，最终新建的目录的权限是drw-r-xr--

**变数：**

type ll：查询指令ll是别名、还是内建指令、还是外部指令

alias：显示所有别名指令

alias lm='ls -al' ：设置别名

unalias [ls](https://www.baidu.com/s?wd=ls&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)删除一个别名

反斜线(\)续行



查看当前用户的shell：echo $SHELL

引用一个变量且用“：-”设定缺省值时，如果变量不存在或者值为空，则显示缺省值且不改变变量自身

echo ${name:-'xyLi'}，当变数name不存在或者设定name=，则打印xyLi

创建变数：



清除时，unset myname

追加内容：



${}变量替换

$()命令替换

env指令：查阅当前shell 环境中的环境变量

set指令：查看当前shell 环境中的变量以及环境变量

LD\_LIBRARY\_PATH=./ 设置变量LD\_LIBRARY\_PATH为./，可以用unset LD\_LIBRARY\_PATH清除

export LD\_LIBRARY\_PATH=./

设置环境变量LD\_LIBRARY\_PATH为./ 可以用unset LD\_LIBRARY\_PATH清除

在当前shell下，用bash命令去启用另一个新的shell ，新的那个shell就是子程序

子程序仅会继承父程序的环境变量，不会继承父程序的自订变数

declare –x name将已存在的自订变数变为环境变数

declare +x name将已存在的环境变数变为自定变数

父程序的自订变数是无法在子程序内使用的。但是通过export将变数变成环境变数后，就能够在子程序底下应用，在子程序下用env命令可以看到定义的环境变数

在子程序下添加一个自订变数，再转为环境变数，则该子程序的子程序用env可以看到新添的环境变数，但在该子程序的父程序下用env看不到新添的环境变数，即子程序中定义的环境变数不能被父程序看到

直接指令下达(不论是绝对路径/相对路径还是${PATH}内)，或者是利用bash (或sh)来下达脚本时，该script都会使用一个新的bash环境来执行脚本内的指令

source对script 的执行是在当前bash环境下

$为bash子程序号码，bash命令打开子程序，exit命令退出当前子程序



=设定变量后，export myname可以将自定义变数转为环境变数，即添加到环境变量，unset可以从当前的环境变量中删除之

PS1是命令提示字元，其中符号的含义：

* \d ：可显示出『星期月日』的日期格式，如："Mon Feb 2"
* \H ：完整的主机名称。举例来说，鸟哥的练习机为『study.centos.vbird』
* \h ：仅取主机名称在第一个小数点之前的名字，如鸟哥主机则为『study』后面省略
* \t ：显示时间，为24 小时格式的『HH:MM:SS』
* \T ：显示时间，为12 小时格式的『HH:MM:SS』
* \A ：显示时间，为24 小时格式的『HH:MM』
* \@ ：显示时间，为12 小时格式的『am/pm』样式
* \u ：目前使用者的帐号名称，如『dmtsai』；
* \v ：BASH 的版本资讯，如鸟哥的测试主机版本为4.2.46(1)-release，仅取『4.2』显示
* \w ：完整的工作目录名称，由根目录写起的目录名称。但家目录会以~ 取代；
* \W ：利用basename 函数取得工作目录名称，所以仅会列出最后一个目录名。
* \# ：下达的第几个指令。\$ ：提示字元，如果是root 时，提示字元为# ，否则就是$ 啰～



问号也是一个特殊的变数，上一个执行的指令所回传的值，一般来说，如果成功的执行该指令，则会回传一个0值，如果执行过程发生错误，就会回传错误代码(一个非0值)



read从标准输入读入一行

read -p "Please keyin your name: " name，从键盘输入获取参数传入变量name



变数类型预设为字符串，数值只有整数



**du查看文件夹大小：**

Linux 查看某个目录下文件的大小，使用du命令，缺省地，du命令会递归地打出每个子目录的大小

du –s:当前目录的总大小，不递归

du –s study/ 只显示目录study/的大小，不递归



du：递归显示当前目录及其下子文件夹的大小

du study/ ：递归显示目录study/及其下子文件夹的大小



du --max-depth=1：递归显示当前目录及其下子目录大小，但截至一级子目录，即显示当前目录及当前目录下的一级子目录，不显示文件大小

du --max-depth=1 mnistExample/：显示mnistExample目录及其下子目录大小，但截至一级子目录



–max-depth=<目录层数> 超过指定层数的目录后，予以忽略

-h：以便于阅读的方式展示

**conda包管理命令：**

conda安装包：conda install protobuf

conda查看matplotlib包的版本号conda list matplotlib

conda list查看已安装的所有包及其版本

conda install numpy=1.12.1

conda update numpy=1.93

查一下我们想要的软件包是否可供conda安装：conda search beautifulsoup4

对于使用conda install不可用的软件包，要从Anaconda.org下载到当前环境，我们将通过键入我们想要的包的完整的URL来指定Anaconda.org作为“通道”。在浏览器中，转到<http://anaconda.org>。我们正在寻找一个名为“bottleneck”的包，在左上角名为“Search Anaconda Cloud”的框中，输入“bottleneck”，然后单击“Search”按钮。在Anaconda.org上有十多个bottleneck可用，但我们想要的最多下载量的副本。因此，你可以通过点击“下载”标题按下载数量进行排序。通过单击软件包名称选择下载量最多的版本。这将带您到Anaconda.org详细信息页面，其中显示用于下载的确切命令：

conda install --channel https://conda.anaconda.org/pandas bottleneck



下载某个版本时：右键，复制链接地址，然后wget 链接地址，如wget <https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/Anaconda2-2.4.0-Linux-x86_64.sh>

bash可以限制使用者的某些系统资源，包括可以开启的档案数量，可以使用的CPU时间，可以使用的记忆体总量

ulimit –a：列出你目前身份(假设为一般帐号)的所有限制资料数值

ldd能够显示可执行模块的dependency，需要的共享库

cd /data/zhangchangjian/caffe/build/lib

ldd libcaffe.so.1.0.0 | grep opencv

**~/.bashrc和~/.bash\_profile和/etc/bashrc：**

source filepath

使当前shell读入路径为filepath的shell文件并依次执行文件中的所有语句

source filename其实只是简单地读取脚本里面的语句依次在当前shell里面执行，没有建立新的子shell。那么脚本里面所有新建、改变变量的语句都会保存在当前shell里面

sh filepath

sh filepath 会重新建立一个子shell，在子shell中执行脚本里面的语句，该子shell继承父shell的环境变量，但子shell是新建的，其改变的变量不会被带回父shell，除非使用export。

修改~/.bashrc后source ~/.bashrc不一定生效，最好重新断掉与服务器的连接重连

shell的自订变数、环境变数

用户环境变数

系统环境变数

~/.bash\_profile和~/.bashrc:用户登陆时，执行~.bash\_profile，该文件开头调用~/.bashrc，~/.bashrc开头调用/etc/bashrc；~.bash\_profile和~/.bashrc除了开头的调用语句，后边给当前shell添加环境变数。







用 ls -lh 来查看某个文件的属性，可以看到有类似-rwxrwxrwx，值得注意的是第一个符号是 - ，这样的文件在Linux中就是普通文件 if [ –f 文件名 ]，判断是否为普通文件

caffe编译：先make，等CXX .build\_release/src/caffe/proto/caffe.pb.cc后再make –j。如果直接make –j，则由于多线程，可能某个线程用到caffe.pb.cc的时候，caffe.pb.cc还未生成，因而报错

make -j4  (-j4表示使用4个线程编译，可以大大加快编译速度）

运行Caffe程序时，必须在Caffe的根目录下，不然会出错。原因是Caffe开发团队在有的运行文件中的路径是从其根目录开始的

**查看安装的protoc位置及版本：**

命令 whereis protoc 可以查看哪些路径下安装了protoc

命令which protoc 可以查看默认选用protoc的路径

命令 protoc --version 可以查看当前protoc版本

which命令的作用是，在PATH变量指定的路径中，搜索某个系统命令的位置，并且返回第一个搜索结果，PATH中有很多路径，从前到后逐个搜索。也就是说，使用which命令，就可以看到某个系统命令是否存在，以及执行的到底是哪一个位置的命令

[cjzhang@gpu019 caffe ]$which protoc：

/usr/local/bin/protoc **#所以敲protoc指令时，执行的即该目录下的protoc**

[cjzhang@gpu019 caffe ]$whereis protoc：

protoc: /usr/bin/protoc /usr/local/bin/protoc /usr/local/protobuf/bin/protoc /home/cjzhang/software/anaconda2/bin/protoc

PATH=/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/usr/local/protobuf/bin:/home/cjzhang/software/anaconda2/bin

/usr/local/bin/protoc –version：libprotoc 3.5.0

/usr/bin/protoc –version：libprotoc 2.5.0

/usr/local/protobuf/bin/protoc –version：libprotoc 3.6.0

**指令搜索、万用子元、;、&&、||、|：**

如果一个指令(例如ls)被下达时，到底是哪一个ls被拿来运作，依照顺序：

1. 以相对/绝对路径执行指令，例如『 /bin/ls 』或『 ./ls 』；
2. 由alias 找到该指令来执行；
3. 由bash 内建的(builtin) 指令来执行；
4. 透过$PATH 这个变数的顺序搜寻到的第一个指令来执行



万用子元：

|  |  |
| --- | --- |
| 符号 | 意义 |
| \* | 代表『 0 个到无穷多个』任意字元 |
| ? | 代表『一定有一个』任意字元 |
| [ ] | 同样代表『一定有一个在括号内』的字元(非任意字元)。例如[abcd] 代表『一定有一个字元， 可能是a, b, c, d 这四个任何一个』 |
| [ - ] | 若有减号在中括号内时，代表『在编码顺序内的所有字元』。例如[0-9] 代表 0 到9 之间的所有数字，因为数字的语系编码是连续的！ |
| [^ ] | 若中括号内的第一个字元为指数符号(^) ，那表示『反向选择』，例如[^abc] 代表一定有一个字元，只要是非a, b, c 的其他字元就接受的意思。 |

找出/etc/底下以cron为开头的档名 ：ll -d /etc/cron\* <==加上-d是为了仅显示目录而已

找出/etc/底下档名含有数字的档名 ：ll -d /etc/\*[0-9]\* <==记得中括号左右两边均需\*

cmd1 && cmd2：

* 若cmd1执行完毕且正确执行($?=0)，则开始执行cmd2。
* 若cmd1执行完毕且为错误($?≠0)，则cmd2不执行

cmd1 || cmd2：

* 若cmd1执行完毕且正确执行($?=0)，则cmd2不执行
* 若cmd1执行完毕且为错误($?≠0)，则开始执行cmd2

例：去best目录，没有则建立：cd best || mkdir best;cd best



* 管线命令 | 仅会处理standard output，对于standard error output 会予以忽略



cut类似Python的split()

**正规表示法与万用字元是完全不一样的东西！**』因为『万用字元(wildcard)代表的是bash操作介面的一个功能』，但正规表示法则是一种字串处理的表示方式

**SVN：**

trunk作为项目的主分支，只维护可发布的稳定代码。开发不在trunk上进行，只有要发布的时候才把代码合并到trunk。

root

|-- trunk

|-- branches

| |-- v1.0-dev

| |-- v1.0-stage

| |-- v2.1-dev

| |-- v2.1-stage

| |-- v2.1.1-stage

| `-- ...

`-- tags

|-- v1.0.0

|-- v1.1.0

|-- v1.1.1

|-- v2.0.0

`-- ...

trunk 主干是代码稳定的。每次上线时都是从 trunk 上创建一个对应版本号的 tag 分支。

dev上开发，提交至svn，dev上继续开发，提交至svn…… commit

将dev上的所有提交合并至stage分支

对stage分支上的功能进行测试

stage分支上功能无误后，合并至trunk主干



**Git：**

**Git的四个区域：**



Workspace：工作区

Index / Stage：暂存区

Repository：仓库区（或本地仓库）

Remote：远程仓库



工作区即电脑里能看到的目录。

隐藏目录.git，是Git的版本库。

Git的版本库里有stage（或者叫index）的暂存区、本地仓库HEAD

git add a.txt将工作区中的文件添加到暂存区

git commit将暂存区中所有修改一次性提交到分支，注意执行git commit后并不会清空暂存区



git的分支和HEAD：

git中的分支，其实本质上仅仅是个指向 commit 对象的可变指针。

HEAD是个指针，指向当前活动分支

**查看状态git status：**

#查看指定文件状态

git status [filename]

#查看所有文件状态

git status

# 添加指定文件到暂存区

$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录

$ git add [dir]

# 添加当前目录的所有文件到暂存区

$ git add .

**git diff比较差异：**

* 工作区和暂存区的差异：$ git diff names.txt
* 工作区文件和本地仓库文件的差异： $ git diff HEAD names.txt
* 暂存区和本地仓库的差异： $ git diff –-cached names.txt

注：如果想查看所有差异，而不是针对某个文件的差异，则去掉上边三条指令最后的文件名字段

$ cat names.txt

cat: names.txt: No such file or directory

$ echo zcj >names.txt #新建文件并写入zcj

$ git add names.txt #添加到暂存区

$ echo lxy >names.txt #写入lxy覆盖原内容

$ git diff names.txt

diff --git a/names.txt b/names.txt

index 744853d..c16ba47 100644

--- a/names.txt # ---a表示后边字段为修改之前的文件

+++ b/names.txt # +++b表示修改后的文件

@@ -1 +1 @@

-zcj #-

+lxy #+

$ git diff HEAD names.txt

diff --git a/names.txt b/names.txt

new file mode 100644

index 0000000..c16ba47

--- /dev/null #之前仓库中无此文件

+++ b/names.txt #工作区新增文件

@@ -0,0 +1 @@

+lxy #新增文件的内容

**git checkout**

$ git checkout filename

#用暂存区中filename文件来覆盖工作区中的filename文件。

#如果暂存区无filename文件，则命令报错

#如果存在同名分支，则git无法区分是要切换分支还是从暂存区恢复文件，此时会报错，应该添加两个连字符以示区分，即git checkout -- filename

$ git checkout branch

#切换到branch分支，则HEAD指针指向branch分支的最近一次commit

$ git checkout branch filename

#维持HEAD的指向不变。用branch所指向的提交中filename替换暂存区和工

#作区中相应的文件。注意会将暂存区和工作区中的filename文件直接覆盖。

$ git checkout commit\_id filename

#表示恢复文件到某次提交

例如：

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ cat f.txt

cat: f.txt: No such file or directory

$ echo flowers >f.txt

$ git commit f.txt # git commit filename提交暂存区文件到仓库区

error: pathspec 'f.txt' did not match any file(s) known to git.

$ git add f.txt

$ git commit f.txt

$ echo games >> f.txt

$ git add f.txt #此时工作区与暂存区f.txt文件无差别，工作区与仓库f.txt文件有差别

$ echo books >> f.txt #此时工作区、暂存区、仓库各不相同

$ git diff f.txt

diff --git a/f.txt b/f.txt

index 40dcfe9..fd30aa9 100644

--- a/f.txt

+++ b/f.txt

@@ -1,2 +1,3 @@

flowers

games

+books

$ git diff HEAD f.txt

diff --git a/f.txt b/f.txt

index ed0efa5..fd30aa9 100644

--- a/f.txt

+++ b/f.txt

@@ -1 +1,3 @@

flowers

+games

+books

$ git checkout f.txt #用暂存区覆盖工作区

$ cat f.txt

flowers

games

$ git diff f.txt #工作区与暂存区无差别了

$ git checkout master f.txt #由仓库恢复，覆盖工作区暂存区

$ cat f.txt

flowers

$ git diff f.txt ; git diff HEAD f.txt #无输出

**git checkout [commit号]**

用某次commit覆盖工作区暂存区，但HEAD指针不变。一般用途：git log查看以前的某次提交，git checkout [commit号]切换到该次提交，git branch name创建新的分支并切换到该新分支，这时候HEAD指针就指向了新分支，而新分支就是之前的那个commit

**git commit：**

# 提交暂存区到仓库区

$ git commit -m [message]

# 提交暂存区的指定文件到仓库区

$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次commit之后的变化，直接到仓库区，跳过add步骤，但对新添加的文件无效

$ git commit -a

**git push**

git push的一般形式为 git push <远程主机名> <本地分支名>:<远程分支名>

git push origin master: master ，即是将本地的master分支（第一个master参数）推送到远程仓库origin的master分支（第二个master参数）

git push origin master ：本地master分支推送到远程主机origin

git push -u origin master：第一次推送master分支时，加上了-u参数，把本地的master分支内容推送到远程主机origin，还会把本地的master分支和远程分支关联起来，在以后的推送或者拉取时就可以简化命令

远程仓库名是在创建的时候指定的：



查看远程仓库地址：



**.gitignore忽略文件**

有些时候我们不想把某些文件纳入版本控制中，比如数据库文件，临时文件，设计文件等在主目录下建立".gitignore"文件，此文件有如下规则：

1. 忽略文件中的空行或以井号（#）开始的行将会被忽略。
2. 可以使用Linux通配符。例如：星号（\*）代表任意多个字符，问号（？）代表一个字符，方括号（[abc]）代表可选字符范围，大括号（{string1,string2,...}）代表可选的字符串等。
3. 如果名称的最前面有一个感叹号（!），表示例外规则，将不被忽略。

\*.a # 所有以 '.a' 为后缀的文件都屏蔽掉

core.\* # 仓库中所有以 'core.' 开头的文件都屏蔽

tags # 仓库中所有名为 tags 的文件都屏蔽

/tags # 只屏蔽仓库根目录下的 tags文件，其他目录中的不屏蔽

readme.md # 屏蔽仓库中所有名为 readme.md 的文件

!/readme.md # 在上一条屏蔽规则的条件下，不屏蔽仓库根目录下的 readme.md 文件

tools/\* # 屏蔽目录 tools及其下所有文件、子目录，等价于tools，等价于tools/

#想忽略某个文件夹，但又不想忽略这个文件夹下的某个子目录

nums/\*

!nums/1/

#想忽略某个文件夹，但又不想忽略这个文件夹下的某个文件

good/\*

!good/\*txt

**.gitignore被track的文件**

在初始化git仓库的时候没有创建.gitignore文件来过滤不必要提交的文件, 后来却发现某些文件不需要提交, 但是这些文件已经被提交了, 这时候创建.gitignore文件忽略这些文件时, 发现ignore的规则对那些已经被track的文件无效。因此需要

$ git rm –-cache -r CartoonMaker/Debug #需要删除暂存区或分支上的文件, 但本地又需要使用, 只是不希望这个文件被版本控制。这里不想CartoonMaker/Debug目录下的所有文件被版本控制

rm 'CartoonMaker/Debug/CartoonMaker.log'

rm 'CartoonMaker/Debug/CartoonMaker.obj'

rm 'CartoonMaker/Debug/CartoonMaker.pch'

$ git commit

$ git push origin

**提示有大文件，不能push**

$ git filter-branch --force --prune-empty --index-filter 'git rm -rf --cached --ignore-unmatch VideoPlayerMFC/\*' --tag-name-filter cat -- --all

$ git push --force --all

**clone：**

克隆远程仓库到本地：先进入/e/study目录，执行下面命令之后，克隆到该目录下

myth@DESKTOP-UN1P95E MINGW64 /e/study

$ git clone https://github.com/leixiaohua1020/simplest\_ffmpeg\_audio\_player.git

**git reset撤销commit**

在Git中commit错误的文件，但还没有push到远程仓库。如何不影响本地的源码，撤销那一次commit?

1. git log查看commit历史记录，由上到下显示，最新的提交（想撤掉的错误提交）、次新的提交（想恢复到的提交状态）……
2. 记录由上至下出现的第二个commit\_id（2a9512cfdeda3c7869072a269ae815a4fb8a8ab8）
3. git reset --soft commit\_id （如果使用git reset --hard commit\_id，则本地源码也会变为上一个版本的内容）

即：git reset --soft d1a65e9ac9a7c4396206f0072b7fbc9138a26c1f

git reset –hard：本地仓库、暂存区、工作区都回退到之前版本的内容

git reset –mixed：本地仓库、暂存区回退到之前版本的内容，工作区不变

git reset –soft：本地仓库回退到之前版本的内容，暂存区和工作区不变。

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/Learning-Notes (master)

$ git log

commit e71ba2757c962bbc8b04d6812685dd76901c169d (HEAD -> master)

Author: mythsaber <lolwowexcalibur@gmail.com>

Date: Thu Jul 18 22:42:45 2019 +0800

routinly commit

commit 2a9512cfdeda3c7869072a269ae815a4fb8a8ab8 (origin/master, origin/HEAD)

Author: mythsaber <lolwowexcalibur@gmail.com>

Date: Mon Jul 8 10:35:06 2019 +0800

routinly commit

commit 24700811197d8f3940e5c1d192cfd069f9a0951b

Author: mythsaber <lolwowexcalibur@gmail.com>

Date: Sun Jun 30 21:19:47 2019 +0800

routinely commit

$ git reset --soft 2a9512cfdeda3c7869072a269ae815a4fb8a8ab8

HEAD is now at 2a9512c routinly commit

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/Learning-Notes (master)

$ git log

commit 2a9512cfdeda3c7869072a269ae815a4fb8a8ab8 (HEAD -> master, origin/master, origin/HEAD)

Author: mythsaber <lolwowexcalibur@gmail.com>

Date: Mon Jul 8 10:35:06 2019 +0800

routinly commit

commit 24700811197d8f3940e5c1d192cfd069f9a0951b

Author: mythsaber <lolwowexcalibur@gmail.com>

Date: Sun Jun 30 21:19:47 2019 +0800

routinely commit

* 修改工作区文件后，无论是否add到暂存区，只要没有commit，则可以git reset --hard HEAD用本地仓库所有内容覆盖工作区和暂存区

git reset迁出仓库内容

*git* *reset* *HEAD* -- <*filename*> *拉取最近一次提交到版本库的文件到暂存区* *改操作不影响工作区*

如果修改了工作区的内容，且add到了暂存区，要想恢复为与仓库的一致，则先git reset HEAD <filename>，使得暂存区与仓库一致，再git checkout -- <filename>用暂存区的内容覆盖工作区

git rm

git rm不涉及对本地仓库的更改

* 情景一：工作区和暂存区中文件辛弃疾.txt无差异，执行git rm xinqiji.txt，则同时从工作区、暂存区删除了该文件，之后可以直接git commit –m "delete a file"提交暂存区至本地仓库
* 情景二：add文件bad.txt至暂存区后，又在工作区修改了该文件，此时

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git rm bad.txt

error: the following file has staged content different from both the

file and the HEAD:

bad.txt

(use -f to force removal)

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git rm -f bad.txt //此时，工作区与暂存区无差别，都没有bad.txt文件

* 情景三：

git rm --cache 文件名仅仅删除暂存区的文件，不会影响工作区的文件，与git add 文件名恰好完全相反

git rm –cache –r 目录名

执行后输入git status命令，git会告知有一个未跟踪的文件TestFile.txt。

**git revert：**

考虑这个例子，我们提交了 6 个版本，其中 3-4 包含了错误的代码需要被回滚掉。 同时希望不影响到后续的 5-6。

\* 982d4f6 (HEAD -> master) version 6

\* 54cc9dc version 5

\* 551c408 version 4, harttle screwed it up again

\* 7e345c9 version 3, harttle screwed it up

\* f7742cd version 2

\* 6c4db3f version 1

这种情况在团队协作的开发中会很常见：可能不小心合入了错误的代码，合入一段时间后才发现存在问题，但存在后续提交，使得直接reset回滚不太现实。

思路是产生一个新的 Commit 撤销之前的错误提交。

使用 git revert <commit> 可以撤销指定的提交， 要撤销一串提交可以用 <commit1>..<commit2> 语法。 注意这是一个前开后闭区间，即不包括 commit1，但包括 commit2。

git revert --no-commit f7742cd..551c408

git commit -a -m 'This reverts commit 7e345c9 and 551c408'

其中 f7742cd 是 version 2，551c408 是 version 4，这样被移除的是 version 3 和 version 4。 注意 revert 命令会对每个撤销的 commit 进行一次提交，--no-commit 后可以最后一起手动提交。

此时 Git 记录是这样的：

\* 8fef80a (HEAD -> master) This reverts commit 7e345c9 and 551c408

\* 982d4f6 version 6

\* 54cc9dc version 5

\* 551c408 version 4, harttle screwed it up again

\* 7e345c9 version 3, harttle screwed it up

\* f7742cd version 2

\* 6c4db3f version 1

现在的 HEAD（8fef80a）就是我们想要的版本，把它 Push 到远程即可。

**Git编码问题**

Windows自带的记事本保存为UTF-8时，带BOM，push到github，网页显示有乱码。因此保存为不带BOM的UTF-8格式较好

文件名最好使用英文，否则某些命令执行后的输出信息里如果引用文件名，则不能正常中文显示，如

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: "\350\213\217\344\270\234\345\235\241.txt" //本来文件名为中文

**重命名工作区某个已commit文件**

git mv -f 辛弃疾.txt xinqiji.txt

git commit -a -m "rename to xinqiji.txt"

**分支显示、创建、切换**

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git branch luck //创建新分支

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git branch //显示当前所有分支，前边带星号的为当前分支

luck

\* master

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git checkout luck //切换到luck分支

Switched to branch 'luck'

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (luck)

$ git branch

\* luck

master

**分支合并**

以下合并luck分支到master分支

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (luck)

$ git checkout master //首先切换到master分支

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git merge luck //然后将luck分支merge到当前分支（当前分支即luck分支）

Updating 5be397a..dabe8ae

Fast-forward

xinqiji.txt | 4 +++-

1 file changed, 3 insertions(+), 1 deletion(-)

一般在其它分支如dev修改了代码，想合并到主干分支，最佳做法是：

1. 切换到dev分支
2. merge主干分支到dev分支，有冲突则解决冲突
3. 切换到主干分支，merge dev分支到主干分支

如果两个分支矛盾

master分支，新增文件，初始文件内容“竹杖芒鞋轻胜马”，commit

新建bad分支，在bad分支中向文件新增内容 “乱石穿空，惊涛拍岸，卷起千堆雪”，commit

回到master分支，在文件中新增语句“一蓑烟雨任平生”，commit

执行git merge bad，提示矛盾

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git merge bad

Auto-merging sudongpo.txt

CONFLICT (content): Merge conflict in sudongpo.txt

Automatic merge failed; fix conflicts and then commit the result.

记事本打开此时的txt文件，内容为：



其中，<<<<<<<，=======，>>>>>>>为Git采用的用于标记出不同分支的内容的符号

用vscode打开该文件，vscode由于安装了合适的插件，打开后界面如下



在上方可选择采用哪个分支、采用所有分支等选项，方便

如点击Accept incoming change，则txt文件内容变为：



然后

git add sudongpo.txt

git commit -m "fix merge conflict" //这两句不能合并为一句，否则报错

如果之后不想用某个分支了，可以删除之，如删除luck分支，git branch -d luck

如果要丢弃一个没有被合并过的分支，可以通过git branch -D 分支名强行删除

合并前：



合并后：



这种合并模式称之为Fast forward模式

加上git merge --no-ff -m "merge with no-f" feature则强制关闭Fast forward模式

--no-ff 会让 Git 生成一个新的提交对象。为什么要这样？通常我们把 master 作为主分支，上面存放的都是比较稳定的代码，提交频率也很低，而 feature 是用来开发特性的，上面会存在许多零碎的提交，快进式合并会把feature的提交历史混入到master中，搅乱master的提交历史



**哪些分支需要推送？**

并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

* master分支是主分支，因此要时刻与远程同步；
* dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；
* bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；
* feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发

**git log：**

git log --graph --pretty=oneline --abbrev-commit

git log

git reflog

**stash与bug分支**

正在dev分支上开发，突然接到一个修复一个代号101的bug的任务

当前正在dev上进行的工作还没有提交：

$ git status

On branch dev

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: hello.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

并不是不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。

这时该怎么办？

Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

$ git stash

Saved working directory **and** index state WIP on dev: f52c633 add merge

现在，用git status查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug，修复完并提交后。恢复之前保存的工作现场：

* 一是用git stash apply恢复，但是恢复后，stash内容并不删除，需要用git stash drop来删除；
* 另一种方式是用git stash pop，恢复的同时把stash内容也删了

**多人协作：**

多人协作时，大家都会往master和dev分支上推送各自的修改。

一位伙伴也克隆了远程仓库，协助你开发

$ git clone git@github.com:michaelliao/learngit.git

此时，默认情况下，你的小伙伴本地仓库只有的master分支。可以用git branch命令看看：

$ git branch

\* maste

现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

$ git checkout -b dev origin/dev

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

$ git push origin dev

**To** github.com:michaelliao/learngit.git

! [rejected] dev -> dev (non-fast-forward)

error: failed **to** push **some** refs **to** 'git@github.com:michaelliao/learngit.git'

hint: Updates were rejected because the tip **of** your **current** branch **is** behind its remote counterpart.

你的小伙伴的最新提交和你试图推送的提交有冲突。解决办法很简单，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送：

$ git branch --set-upstream-to=origin/dev dev 指定本地dev分支与远程origin/dev分支的链接

$ git pull

Auto-merging env.txt

CONFLICT (add/add): Merge conflict **in** env.txt

Automatic merge failed; fix conflicts **and** **then** commit the result.

git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的[解决冲突](http://www.liaoxuefeng.com/wiki/896043488029600/900004111093344)完全一样。解决后，提交，再push，最终成功

git pull 默认等于 git fetch + git merge，拉取远端代码，并合并到本地

git fetch 命令默认等于git fetch origin

git merge 命令默认等于git merge origin/master

假设本来只有master、bad分支，如果伙伴创建了dev分支并push到了远端仓库，自己也想在dev分支开发：

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git pull

From https://github.com/mythsaber/GitTest

\* [new branch] dev -> origin/dev

Already up to date.

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git branch

bad

\* master

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git branch dev //在本地创建与远端同名的分支，此时该分支即有了远端dev分支的内容

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git branch

bad

dev

\* master

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (master)

$ git checkout dev

Switched to branch 'dev'

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (dev)

$ ll

total 6

-rw-r--r-- 1 myth 197121 16 6月 21 22:56 a.txt

-rw-r--r-- 1 myth 197121 14 6月 21 22:15 daynote.txt

-rw-r--r-- 1 myth 197121 15 6月 21 22:15 hello.txt

-rw-r--r-- 1 myth 197121 70 6月 21 22:15 sudongpo.txt

-rw-r--r-- 1 myth 197121 10 6月 21 22:15 word.txt

-rw-r--r-- 1 myth 197121 146 6月 21 22:15 xinqiji.txt

//对其中的a.txt做了修改

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (dev)

$ git commit -a -m "modify a.txt in dev branch"

[dev 773797e] modify a.txt in dev branch

1 file changed, 1 insertion(+), 1 deletion(-)

myth@DESKTOP-UN1P95E MINGW64 /e/GitHub/GitTest (dev)

$ git push origin dev

**commit object、tree object与blob object的关系：**





**makefile：**

1. 规则

规则包含两个部分：依赖关系（目标+依赖目标）和生成目标的方法

<target> : <prerequisites>

[tab] <commands>

target ：要生成的目标文件名称，简称为目标，

prerequisites：依赖的文件，也称为依赖目标

[tab]即tab键，必须存在

commands：任意的shell命令

功能：prerequisites中如果有一个以上的文件比target文件要新的话（或者target文件不存在），command所定义的命令就会被执行。

示例1：由main.cpp拷贝生成pow.cpp，则makefile如下，执行make

pow.cpp:main.cpp

cp main.cpp pow.cpp

示例2：编译链接，由main.cpp生成可执行文件

CC=g++

CFLAGS=-g -Wall

LIBS=-lm

OBJ=main.o

$(OBJ):main.cpp

$(CC) $(CFLAGS) -c main.cpp -o $(OBJ)

TARGET=main

$(TARGET):$(OBJ)

$(CC) $(CFLAS) -o $(TARGET) $(OBJ) $(LIBS)

.PHONY:clean

clean:

rm -f $(OBJ) $(TARGET)

将编译器定义为变量CC

将编译参数选项定义为变量CFLAGS，-g表示可调试

-w：关闭编译时的警告，不显示任何warning，默认行为

-Wall：编译时显示所有警告。

由于第一个目标是main.o，因此单单执行make的时候，只会生成main.o，如果再执行一次make，会提示main.o已经是最新的了

为了得到main，需要先make，再make main

为一次make即生成main，则调整顺序如下，即可：

TARGET=main

$(TARGET):$(OBJ)

$(CC) $(CFLAS) -o $(TARGET) $(OBJ) $(LIBS)

$(OBJ):main.cpp

$(CC) $(CFLAGS) -c main.cpp -o $(OBJ)

1. make和make clean

make命令等于make -f makefile

make clean命令等于make clean -f makefile

命令行输入 make 将默认执行第一个target（常为 all）下方的命令；

输入 make clean，则指定执行clean这个target下方的命令

1. [makefile中的all和.PHONY](https://www.cnblogs.com/tureno/articles/6210869.html)

请编写一个makefile同时编译、链接下面两个程序：

//main1.c：

#include <cstdio>

int main()

{

    printf("main1\n");

}

//main2.c：

#include <cstdio>

int main()

{

    printf("main2\n");

}

【分析】：这里需要生成两个可执行文件main1和main2（两个目标）。由于makefile只能有一个目标，所以可以构造一个没有规则的终极目标all，并以这两个可执行文件作为依赖。如下：

//makefile：

all:main1 main2

main1: main1.c

    @gcc main1.c -o main1

main2: main2.c

    @gcc main2.c -o main2

其中@表示禁止回显

很多时候我们在执行make时会产生许多过程文件，比如将上面的makefile改为：

//makefile：

all:main1 main2

main1: main1.c

    @gcc -o main1 main1.c

main2: main2.o

    @gcc -o main2 main2.o

main2.o: main2.c

    @gcc –o main2.o -c main2.c

这样就会生成一个我们不需要的过程文件main2.o。

如果希望将生成的过程文件删掉，再增加一个目标clean即可：

all:main1 main2 clean

main1: main1.c

    @gcc main1.c -o main1

main2: main2.o

    @gcc main2.o -o main2

main2.o: main2.c

    @gcc -c main2.c

clean:

    @rm -f main2.o

但如果目录有名为clean的文件，由于目标clean没有依赖，所以始终是最新的，所以命令rm -f main2.o永远不会被执行。为避免这个问题，可使用".PHONY"指明该目标

单词phony (即phoney)的意思是：伪造的，假的

在Makefile中，target默认是文件，而.PHONY后接target表示这是一个伪造的target, 而不是真实存在的文件，因此.PHONY后接的target的生成规则一定会被执行，不受磁盘真实文件的影响

all:main1 main2 clean

main1: main1.c

    @gcc main1.c -o main1

main2: main2.o

    @gcc main2.o -o main2

main2.o: main2.c

    @gcc -c main2.c

.PHONY: clean

clean:

@-rm -f main2.o

rm 命令前面加了一个小减号的意思就是这条命令如果执行出错，忽略之，继续make的执行，，而不是立即退出

chmod +x a.sh 给a.sh可执行权限

C++工程的makefile文件中，定义了包含目录、库目录、依赖库的名字，还会调用prebuild.bat、postbuild.bat，最后调用g++的编译命令生成exe或动态库静态库等

1. $(foreach <var>,<list>,<text>)

等价于C++的for(auto var: list) {text;}

把参数<list>中的单词逐一取出赋值给<var>变量，然后再执行<text>表达式。每一次<text>会返回一个字符串，当整个循环结束时，各个<text>调用所返回的字符串以空格分隔构成整个字符串，作为foreach函数的返回值。

1. $(wildcard PATTERN)

wildcard：通配符

返回已经存在的、使用空格分开的、匹配指定模式的所有文件的列表

如，获取三个目录下的所有.cpp文件：

SRC\_DIRS=../../../../src ../../../../src/encoder ../../../../src/common

SRC=$(foreach dir, $(SRC\_DIRS), $(wildcard $(dir)/\*.cpp))

SOURCE\_DIR = . #当前目录

SOURCE\_FILE = $(wildcard $(SOURCE\_DIR)/\*.cpp)

target:

@echo $(SOURCE\_FILE)

make后发现，输出的为当前目录下所有的 .cpp 文件：

./function1.cpp ./function2.cpp ./main.cpp

其中@echo前加@是为了避免命令回显，上文中make clean调用了rm -rf会在terminal中输出这行命令，如果在rm前加了@则不会输出了。

1. 变量赋值：

makefile中的变量类似于C++中的宏，会被展开

=  延时变量，只有被使用时才展开定义  
:=  立即变量，定义时的赋值立即有效  
?=  条件变量，当变量为空时才赋值  
+= 追加赋值，被使用时才展开定义，用空格分隔

VAR\_A=abc

VAR\_B=$(VAR\_A)

VAR\_C:=$(VAR\_A)

VAR\_A=def

.PHONY: all

all:

echo $(VAR\_A) #输出为def

echo $(VAR\_B) #输出为def

echo $(VAR\_C) #输出为abc

Makefile中注释是用 # 字符，作用同C/C++中的 //s ，其后的本行字符都被注释。如果要在Makefile中使用 # 字符，可以用反斜杠进行转义，如： \#

VAR\_A:=$(VAR\_B)222

VAR\_B=$(VAR\_A)

#由于一开始VAR\_B没有值，所以VAR\_A的值为222，VAR\_B的值为222

VAR\_D=111

VAR\_D?=000

#VAR\_D的值仍为111

VAR\_E=a.txt

VAR\_F=b.txt

VAR\_F+=$(VAR\_E)

VAR\_E=b.txt

.PHONY: target

target:

echo $(VAR\_F) #输出b.txt b.txt

1. 定义多行变量：

define指示符后面跟变量名，而重起一行定义变量的值，定义是以endef 关键字结束。

变量的值可以包含函数、命令、文字，或是其它变量

//makefile内容

define twoline

@echo foo

@echo $(shell pwd)

endef

all:

$(twoline)

#运行结果：

foo

/home/cmake

1. 变量的值作为变量

“把变量的值再当成变量”这种技术，既可以可以用在操作符的左边，又可以用在操作符的右边

//示例1

first\_second = Hello

a = first

b = second

all = $($a\_$b)

#这里的 $a\_$b 组成了“first\_second”，于是， $(all) 的值就是“Hello”

//示例2

a\_objects := a.o b.o c.o

1\_objects := 1.o 2.o 3.o

sources := **$($(**a1**)**\_objects:.o=.c**)**

这个例子中，如果 $(a1) 的值是“a”的话，那么， $(sources) 的值就是“a.c b.c c.c”；如果 $(a1) 的值是“1”，那么 $(sources) 的值是“1.c 2.c 3.c”。

//示例3

dir := /home/c++/ipc

$(dir)\_srcs := $(wildcard $(dir)/\*.cpp)

all:

@echo $(dir)\_srcs

@echo $($(dir)\_srcs)

#这里定义了变量，名为/home/c++/ipc\_srcs，值为/home/c++/ipc/目录下所有cpp文件的路径名集合

第一行的echo $(dir)\_srcs打印变量名

第二行的@echo $($(dir)\_srcs)打印变量值

1. 变量替换：

（1）patsubst

pattern substitution，模式替换

$(patsubst 原模式，目标模式，文件列表)。

注意用%表示模式，如果某个文件模式不匹配则原样输出

可以方便地将.cpp 文件的后缀换成.o，如下示例

SOURCES = main.cpp function1.cpp function2.cpp

OBJS = $(patsubst %.cpp, %.o, $(SOURCES))

.PHONY: target

target:

@echo $(SOURCES)

@echo $(OBJS)

注意用%表示模式。输出的结果为：

main.cpp function1.cpp function2.cpp

main.o function1.o function2.o

（2）OBJ:=$(SRC:.cpp=.o)

将SRC变量中所有以.cpp结尾的字串替换成以.o结尾的子串，然后赋值给OBJ

这里的“结尾”意思是后边是“空格”或“结束符”

如果某个字串不匹配则原样输出

（3）静态模式

a := $(foo:%.o=%.c)：将变量foo中所有符合%.o模式的字串替换为%.c模式

x := $(bar:a.%=b.%)：将变量foo中所有符合a.%模式的字串替换为b.%模式

//示例

foo := a.o b.obj c.o d.o

a := $(foo:%.o=%.c)

b := $(foo:.o=.c)

bar :=a.c a.o log.txt a.exe

x := $(bar:a.%=b.%)

y := $(bar:a.=b.)

all:

@echo $(a)#结果为a.c b.obj c.c d.c

@echo $(b)#结果为a.c b.obj c.c d.c

@echo $(x)#结果为b.c b.o log.txt b.exe

@echo $(y)#结果为a.c a.o log.txt a.exe

1. 目标变量：

目标变量是一种局部变量，作用域为目标，而且全局

目标变量的作用范围只在这条规则以及连带规则中，在其作用范围内会覆盖全局的同名变量

语法格式：<target ...> : <variable-assignment>

<variable-assignment>是各种赋值表达式，如 = 、 := 、 += `` 或是 ``?=

prog : CFLAGS = -g

prog : prog.o foo.o bar.o

$(CC) $(CFLAGS) prog.o foo.o bar.o

prog.o : prog.c

$(CC) $(CFLAGS) prog.c

foo.o : foo.c

$(CC) $(CFLAGS) foo.c

bar.o : bar.c

$(CC) $(CFLAGS) bar.c

在这个示例中，不管全局的 $(CFLAGS) 的值是什么，在prog目标，以及其所引发的所有规则中（prog.o foo.o bar.o的规则）， $(CFLAGS) 的值都是 –g

给所有以.o结尾的目标定义目标变量：

%.o : CFLAGS = -O

1. make的命令行参数中设置变量

shell中执行：make dir=/home/

则makefile中可访问变量dir

即使在makefile中也定义了变量dir，也不会覆盖命令行参数的值

如果想在makefile中对dir赋值，override dir=/opt/

如果想在makefile中对dir追加，override dir += /usr/

//示例

[root@DESKTOP-UN1P95E cmake]$ cat makefile

day := thu

override year := 2020

override love += football

all:

@echo $(color)

@echo $(day)

@echo $(year)

@echo $(love)

[root@DESKTOP-UN1P95E cmake]$ make color=red day=fri year=2020 love=tennis

red //来自命令行参数

fri //不能覆盖命令行中的值

2020 //覆盖命令行中的值

tennis football ..对命令行中定义的值追加

[root@DESKTOP-UN1P95E cmake]$ make

//空

thu

2020

football

1. 嵌套makefile中变量的传递：

export传递变量到下级（类似父shell的环境变量可以传递到子shell）：

#主控makefile

DAY=08-07

export DAY

all:

cd enterdoor && make

echo $(DAY)

export DAY表示将当前makefile中的变量DAY传递给下级makefile

如果下级enterdoor/makfile也定义了变量DAY，则会覆盖主控makefile传递而来的变量

如果不想下级makefile覆盖主控makefile传递的变量，则执行主控makefile时加上-e参数

如果主控mkaefile没有export DAY，则不会传递参数给下级

不需要export也会传递的特殊变量：

有两个变量SHELL和MAKEFLAGS ，不管是否export，总会传递到下层Makefile。特别是 MAKEFLAGS 变量，其中包含了make的参数信息，如果我们执行“总控Makefile”时有make参数或是在上层 Makefile中定义了这个变量，那么 MAKEFLAGS 变量将会是这些参数，并会传递到下层Makefile中

但是make命令中的有几个参数并不往下传递，它们是 -C , -f , -h, -o 和 -W

//主控makefile

all:

cd enterdoor && make

@echo SHELL=$(SHELL)

@echo MAKEFLAGS=$(MAKEFLAGS)

//下级enterdoor/makefile

all:

@echo SHELL=$(SHELL)

@echo MAKEFLAGS=$(MAKEFLAGS)

//make主控结果：

cd enterdoor && make

make[1]: Entering directory '/home/cmake/enterdoor'

SHELL=/bin/sh

MAKEFLAGS=w

make[1]: Leaving directory '/home/cmake/enterdoor'

SHELL=/bin/sh

MAKEFLAGS=

1. 示例及$@、$^、$<

$@：表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，"$@"就是所有符合模式的一系列的文件集。

$^：所有的依赖的集合，以空格分隔。如果在依赖中有多个重复的，那个这个变量会去除重复的依赖，只保留一份。

$<：依赖中的第一个文件名。如果依赖是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。

将命令g++ -o hello main.cpp a.cpp b.cpp写到makefile中：

[root@guest: zcj]$ cat makefile

SRCS = $(wildcard ./\*.cpp)

OBJS := $(SRCS:.cpp=.o)

CC = g++

CFLAGS = -c -Wall

LFLAGS = -Wall

all: main.out

main.out: $(OBJS)

@echo $(SRCS) #运行时打印出./b.cpp ./a.cpp ./main.cpp

@echo $(OBJS) #运行时打印出./b.o ./a.o ./main.o

$(CC) $(LFLAGS) -o $@ $^ #其中$^表示所有的依赖的集合

main.o: main.cpp

$(CC) $(CFLAGS) $< -o $@ #其中$@表示目标，$<表示第一个依赖，也即main.cpp

a.o: a.cpp

$(CC) $(CFLAGS) $< -o $@

b.o: b.cpp

$(CC) $(CFLAGS) $< -o $@

clean:

rm -rf \*.o main.out

[root@guest: zcj]$

或者



1. 回显@和仅显示-n：

@置于命令前，禁止回显

make –n，则只显示命令，但不执行，方便调试看书写的命令是执行起来是什么样子

1. 多目标：

Makefile的规则中的目标可以是多个

一般多个目标的生成命令大体类似，因此可以将其合并起来，并借助自动化变量 $@ 实现对多个目标的遍历

[root@DESKTOP-UN1P95E cmake]$ cat makefile

a.cpp b.cpp: main.cpp

cp main.cpp $@

[root@DESKTOP-UN1P95E cmake]$ make -n

cp main.cpp a.cpp

[root@DESKTOP-UN1P95E cmake]$ make a.cpp -n

cp main.cpp a.cpp

[root@DESKTOP-UN1P95E cmake]$ make b.cpp -n

cp main.cpp b.cpp

可以看到，多目标命令会被展开，各目标分别对应一个规则

[root@DESKTOP-UN1P95E cmake]$ cat makefile

all:a.cpp b.cpp

a.cpp b.cpp: main.cpp

cp main.cpp $@

[root@DESKTOP-UN1P95E cmake]$ make -n

cp main.cpp a.cpp

cp main.cpp b.cpp

1. 多依赖：

多目标规则等价于多条单目标规则

但多依赖规则，并非等价于多个单依赖规则

[root@DESKTOP-UN1P95E cmake]$ cat makefile

all:a.o b.o

a.o b.o: a.cpp b.cpp

cp $< $@

[root@DESKTOP-UN1P95E cmake]$ make -n

cp a.cpp a.o

cp a.cpp b.o

[root@DESKTOP-UN1P95E cmake]$ cat makefile

all:a.o b.o

a.o b.o: a.cpp b.cpp

cp $^ $@

[root@DESKTOP-UN1P95E cmake]$ make -n

cp a.cpp b.cpp a.o

cp a.cpp b.cpp b.o

对多目标规则，如果要为多个目标指定不同的依赖，则需要借助静态模式：

[root@DESKTOP-UN1P95E cmake]$ cat makefile

all:a.o b.o

a.o b.o: %.o: %.cpp

cp $< $@

[root@DESKTOP-UN1P95E cmake]$ make -n

cp a.cpp a.o

cp b.cpp b.o

1. 多条命令前后关系：

all:

cd enterdoor && make

等价于make -C enterdoor。只有第一条命令执行成功，才会执行后边的命令（且是在第一条命令的基础上执行）

如果cd enterdoor失败，则终止；如果cd enterdoor成功，则继续，因此会执行enterdoor下的make

all:

cd enterdoor ; make

无论cd enterdoor是否成功，都会执行make。如果成功，则执行enterdoor下的make，如果失败，则执行当前目录下的make

all:

cd enterdoor

make

第二条命令并非在第一条命令的基础上执行。如果cd enterdoor失败，则终止，如果失败，则执行当前目录下的make

1. -w参数：

-w 或是 --print-directory 会在make的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级make目录是“/home/hchen/gnu/make”，如果我们使用 make -w 来执行，那么当进入该目录时，我们会看到:

make: Entering directory `/home/hchen/gnu/make'.

而在完成下层make后离开目录时，我们会看到:

make: Leaving directory `/home/hchen/gnu/make'

1. 获取当前目录/用户/系统架构：

ifeq (0,${MAKELEVEL})

cur-dir := $(shell pwd)

whoami := $(shell whoami)

host-type := $(shell arch)

MAKE := ${MAKE} cur\_dir=$(cur-dir) host-type=${host-type} whoami=${whoami}

endif

all:

@echo ${MAKELEVEL}

@echo $(MAKE)

运行结果为：

[root@DESKTOP-UN1P95E cmake]$ make

0

make cur\_dir=/home/cmake host-type=x86\_64 whoami=root

ifeq, endif为条件表达式，判断变量MAKELEVEL值是否为0

makefile支持嵌套执行，变量MAKELEVEL记录调用层数，由0开始

1. 定义空格：

变量定义中，变量值的前导空格字符在变量引用和函数调用时被丢弃。变量值中的尾空格是不被忽略的

#定义一个空格

nullstring :=

space := $(nullstring) #end of the line

#上一行中注意#号前有一个空格

#也可以采用下面的方法定义一个空格

nullstring :=

space := $(nullstring) $(nullstring)#end of the line

# 上一行中注意两个$(nullstring)之间有一个空格

因为变量值中的尾空格不被忽略，所以变量定义和注释在同一行时，应该注意！

如有下面的例子：

dir := /foo/bar #directory to put the frobs in

变量dir的值是 "/foo/bar "（后面有4个空格），如果一个文件以它作为路径来表示 ，如“$(dir)/file” ，那么就错了

1. 条件语句：

CC=g++

all:

ifeq ($(CC),g++)

@echo g++

else

@echo not g++

endif

等价于：

CC=g++

ifeq ($(CC),g++)

exe=@echo g++

else

exe=@echo not g++

endif

all:

$(exe)

ifeq的参数中可以含make函数

//判断src中是否含有hello子串

src=goog hellostudyi

has=true

ifeq ($($(findstring hello, src)),)

has=false

endif

all:

@echo $(has)

#结果打印true，如果上文是$(findstring hellox, src)，则这里为false

函数$(findstring <find>,<in>)

功能：在字串 <in> 中查找 <find> 字串。

返回：如果找到，那么返回 <find> ，否则返回空字符串

ifneq与ifeq相对，表示不等于

ifdef判断变量是否定义（而非是否为空），与ifndef相对

示例一：

bar =

foo = $(bar)

ifdef foo

frobozz = yes

else

frobozz = no

endif

示例二：

foo =

ifdef foo

frobozz = yes

else

frobozz = no

endif

第一个例子中， $(frobozz) 值是 yes ，第二个则是 no

1. 其它：

引用其它makefile

如include foo.make

作用同C++的#include <foo.make>，被包含的文件会原模原样的放在当前文件的包含位置

make支持三个通配符： \* ， ? 和 ~

如果我们的文件名中有通配符，如： \* ，那么可以用转义字符 \ ，如 \\* 来表示真实的 \* 字符，而不是任意长度的字符串

通配符用在命令中：

clean:

rm -rf \*.o #删除所有名字以.o结尾的文件

通配符用在变量中：

objects = \*.o

上面这个例子，表示了通配符同样可以用在变量中。并不是说 \*.o 会展开，不！objects的值就是 \*.o 。Makefile中的变量其实就是C/C++中的宏。如果你要让通配符在变量中展开，也就是让objects的值是所有 .o 的文件名的集合，那么，你可以这样：

objects := $(wildcard \*.o)

**ssh免密登录：**

win10上以ssh连接服务器45.77.131.212

cmd中操作：

ssh 45.77.131.212：由于当前在win10系统上是myth用户，因此连接的也是45.77.131.212的myth账号

ssh [root@45.77.131.212](mailto:root@45.77.131.212)：指明要连接的是服务器的root账号

提示输入密码，输入后登陆成功，进入服务器的root用户根目录

logout，登出，回到cmd状态

再次连接，又要输入密码

免密登录：

cmd中输入ssh-keygen生成公钥id\_rsa、私钥id\_rsa.pub对



默认放置于 用户/.ssh目录下

将公钥的内容添加到服务器的用户根目录/.ssh/authorized\_keys文件的最后一行，没有文件则新建

[root@guest: ~]$ mkdir .ssh

[root@guest: ~]$ cd .ssh/

[root@guest: .ssh]$ echo 'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCvIoK1JbjodQCUJ7BssN126ytCBssNR2Y4yqvWShTITl/U4JdheUKezEcpTRSfwhbAXAwg/CB+7p0ICySGsEhnHL2I9F7oIjk/HylNPizx9Bym3Sbiv9PpbU0G+/xN0JYAgFcW40LCyCvFy4KjiyPPMsBOPaez3Nlz5TZx1iljuT9TCp4I1NGBXqMuoN0eLQnUFEpLwttAJFSLf8vDgojpk6g4HznhQVUFr2/oGU8RqSY/KeSukiVmD/fZjpQDRK9OM4o0UZzOTqd3FgQsyCSaq9YaAsAzwzHOAWEPrulds8jJEC7MTm82ksYpiYtyD0ckiUZvknqJQP8TFC/pDj3F myth@DESKTOP-UN1P95E' >> authorized\_keys

再次在cmd中尝试连接，发现直接登录上：



**Docker：**



镜像如nginx镜像、mysql镜像等

镜像运行（run指令）起来后，产生一个容器，容器像一个虚拟机一样，有linux的一整套目录，如bin、usr等

Docker利用容器 (Container) 独立运行的一个或一组应用，容器是用镜像创建的运行实例，它可以被开始、停止、删除，每个容器都是相互隔离的、保证安全的平台，可以把容器看做是一个简易版的Linux系统

pull nginx 镜像：docker pull nginx（等效于nginx:latest）

运行：docker run -【d】（后台运行不阻塞shell） 【-p 80:80】（指定容器端口映射，内部：外部） nginx

查看正在运行的容器：docker ps

删除容器：docker rm -f <container id(不用打全，前缀区分)>

进入容器：docker exec -it <container id(不用打全，前缀区分)> bash

commit镜像：docker commit <container id(不用打全，前缀区分)> <name>

查看镜像列表：docker images

使用运行刚才commit的镜像：docker run -d <name>

使用Dockerfile构建镜像：docker build -t <name> <存放Dockerfile的文件夹>

删除镜像：docker rmi <name>，删除前需要停止依赖次镜像的容器

保存镜像为tar：docker save <name> > <tar name>

从tar加载镜像：docker load < <tar name>

一些启动参数：

后台运行容器：-d

容器内外端口映射：-p 内部端口号:外部端口号

目录映射：-v 'dir name' : <dir>

指定映像版本：<name>:<ver>

**用户态、内核态：**

linux中，man syscalls命令可查看所有的系统调用



操作系统分为内核态、用户态：cpu当前在执行内核代码，则处于内核态；否则为用户态

cpu有各种特权级，分为特权模式、用户模式等。

操作系统内核态对应CPU的特权模式、用户态对应cpu用户模式

cpu不同特权级下，同一条指令可能产生不同的结果。比如cpu用户模式不能访问某个敏感的内存去破坏操作系统。

CPU中引入了一种称为MMU的单元，它能从内存中以指定格式加载配置，从而影响用户模式下访问内存的特性。在用户模式下，所有内存访问经过MMU，从而对内存的访问受到了保护；在特权模式下，内存访问绕过MMU，直接访问物理内存，从而获得完整的权限。

Q：cpu只是冷冰冰地执行一条一条的命令，怎么区分当前“访问内存”这一条指令是否应执行？

Q：cpu（硬件）提供了接口（即指令）供软件（操作系统）控制，接口包括：进入/离开某个特权级、配置特权级下cpu的行为。操作系统使用接口配置cpu在用户模式下不能访问这个内存，特权模式下能访问这个内存。如果当前指令要访问该内存，而cpu发现自己处于用户模式下，cpu就会报告一个异常，操作系统捕捉这个异常后，就知道程序违规了。

具体实现为:

*CPL是当前内存段的权限级别(Current Privilege Level)，是当前正在执行的代码所在的段的特权级，存在于cs寄存器的低两位。*

*DPL存储在GDT表中，规定访问该段的权限级别(Descriptor Privilege Level)，每个段的DPL固定。*

*当进程访问一个段时，需要特权级检查，一般要求DPL >= CPL（数值越大特权级越低）*

由用户态切换到内核态的几种途径：系统调用、中断、异常

1、系统调用：系统调用即操作系统提供的函数调用。如fopen是c库函数，open是系统调用，fopen调用了open；malloc是c库函数，调用了系统调用brk或mmap申请虚拟内存空间。

2、中断：操作系统开始操作后，到硬件操作完成前，cpu闲置，操作系统可使之执行其它任务。当硬件操作完成时，会触发中断，**这个中断使CPU进入特权模式**。这样就操作系统进入了内核态、cpu变为特权模式。操作系统根据中断号查IDT表执行相应的中断处理函数。

3、异常：如缺页异常、用户模式下收到一条不允许执行的指令会发送异常

Q：假设当前处于用户态，要调用系统函数，系统函数访问了敏感内存，那执行到“访问敏感内存”这一条指令是，CPL<DPL，而CPL<DPL时cpu无法访问内存，岂不是无法完成系统调用了？

A：**系统调用也是借助中断实现的**，是硬件提供的支持。对x86cpu，**中断指令int 0x80将使得CS的CPL改为0**，这是用户程序发起的调用内核代码的唯一方式。即fork, open, write等函数，其实现中包含一条int 0x80指令，而该条指令在内存中的DPL是3，同时将函数名对应的中断号、函数参数如write的const char\*参数写入寄存器，是典型的c代码内嵌汇编编程。

**操作系统**

MOV指令可以在CPU内或CPU和存储器之间传送字或字节，它传送的信息可以从寄存器到寄存器，立即数到寄存器，立即数到存储单元，从存储单元到寄存器，从寄存器到存储单元，从寄存器或存储单元到除CS外的段寄存器**(注意立即数不能直接送段寄存器)，**从段寄存器到寄存器或存储单元

例： MOV AL，‘E’

把立即数(字符E的ASC码)送到AL寄存器

程序加载到内存，ip指针指向程序，cpu取指执行

CS:IP 两个寄存器指示了CPU当前将要读取的指令的地址，其中CS为代码段寄存器，而IP为指令指针寄存器，在端中的偏移

1、对用户程序，操作系统加载可执行文件到内存，并设置CS:IP指向可执行文件的起始地址

2、加载操作系统，计算机开机时硬件自动设置CS=0xFFFF、IP=0x0000，而0xFFFF0即为BIOS映射区地址，开机时自动写入内存。BIOS控制cpu检查硬件，读磁盘第一个扇区（即0磁道0扇区，即操作系统的引导扇区）到0x7c00处，设置cs=0x07c0、ip=0x0000。磁盘第一个扇区存放的就是操作系统代码。

jmpi是段间跳转指令，用于x86实模式下，

      如：BOOTSEG = 0x0c70

            jmpi    4, #BOOTSEG

      假如当前段CS==00h，那么执行此指令后将跳转到段CS==0x0c70，当然段cs的值也变为0x0c70，接下来将执行指令0x0c70:0004处的指令。

CPU存在实模式和保护模式的操作模式。当计算机启动后，我们进入的是实模式，而实模式的内存寻址方式是通过：CS左移4位 + IP = 物理地址。这样只能具有1MB的寻址能力。

保护模式下，引入了数据结构GDT，即全局描述符表，GDT是一个数组，每一个元素就是一个描述符，每一个描述符共64位，包含了以下的这些信息：段基址、段长度、属性。保护模式下CS存放的是GDT的索引，查表得到段基址，再加偏移IP，得到内存地址。可寻址更多的内存空间。