

Bash 常用命令

2023年2月23日 16:12

POSIX: 可移植操作系统接口 (Portable Operating System Interface of UNIX, 缩写为 POSIX) , 发布者为电气与电子工程师协会 (Institute of Electrical and Electronics Engineers) , 简称IEEE。POSIX是IEEE为要在各种UNIX操作系统上运行的软件而定义的一系列API标准的总称, 其正式称呼为IEEE 1003, 而国际标准名称为ISO/IEC 9945。

基础命令

Ctrl + r, 进入历史命令的搜索功能模式

history, 查看所有的历史命令

tty, 查看当前终端

输入 echo \$\$ 可以查看当前终端的进程pid

&表示这个程序在后台运行 # ./hello &

nohup command & # ctrl+d 或者 关闭窗口 进程仍然会在后台执行

./表示当前目录 如果不写./, Linux 会到系统路径下查找文件

所谓系统路径, 就是环境变量指定的路径, 我们可以通过修改环境变量添加自己的路径, 或者删除某个路径。很多时候, 一条 Linux 命令对应一个可执行程序, 如果执行命令时没有指明路径, 那么就会到系统路径下查找对应的程序。

| 管道 (pipeline) 连接上个指令的标准输出, 做为下个指令的标准输入。

通过 && 和 || 可以控制多条命令的执行情况 &&要求前一条命令执行成功才执行后面的语句 ||则是前一条失败才执行后面语句

查询CPU信息: cat /proc/cpuinfo

查看CPU的核的个数: cat /proc/cpuinfo | grep processor | wc -l

查看内存信息: cat /proc/meminfo

显示内存page大小 (以KByte为单位: pagesize

ifconfig, 查看内网 IP 等信息 (常用)

date, 查看系统时间 (常用)

date -s20080103, 设置日期 (常用)

date -s18:24, 设置时间

bashrc与profile都用于保存用户的环境信息, bashrc用于交互式non-login shell, 而profile用于交互式login shell。

/etc/profile, /etc/bashrc 是系统全局环境变量设定

~/.profile, ~/.bashrc用户目录下的私有环境变量设定

unset \$JAVA_HOME, 删除指定的环境变量

~/.profile与 ~/.bashrc的区别:

这两者都具有个性化定制功能

~/.profile可以设定本用户专有的路径, 环境变量, 等, 它只能登入的时候执行一次

~/.bashrc也是某用户专有设定文档, 可以设定路径, 命令别名, 每次shell script的执行都会使用它一次

我们可以在这些环境变量中设置自己经常进入的文件路径, 以及命令的快捷方式

Eg.

通过修改.bashrc 可以为命令设置别名 (该文件就在主文件夹底下 隐藏显式)

Eg. alias ls='ls -lrt' alias l='ls -al | more'

启动帐号后自动执行的是 文件为 .profile, 然后通过这个文件可设置自己的环境变量

Eg.安装的软件路径一般需要加入到path中:

PATH=\$APPDIR:/opt/app/soft/bin:\$PATH:/usr/local/bin:\$TUXDIR/bin:\$ORACLE_HOME/bin;export PATH

使用ls -l可查看文件的属性字段, 文件属性字段总共有10个字母组成, 第一个字母表示文件类型, 如果这个字母是一个减号"-", 则说明该文件是一个普通文件。字母“d”表示该文件是一个目录, 字母“d”,是directory(目录)的缩写。后面的9个字母为该文件的权限标识, 3个为一组, 分别表示文件所

属用户、用户所在组、其它用户的读写和执行权限；

Eg. -rwxrw-r-- colin king 725 2013-11-12 15:37 /home/colin/a

表示这个文件对文件拥有者colin这个用户可读写、可执行；对colin所在的组（king）可读可写；对其它用户只可读

• 帮助命令

whatis command	简要说明command的作用（显示command所处的man分类页面）
man command	查询命令command的说明文档
which command	查看程序的binary文件所在路径
whereis command	查看程序的搜索路径（用于当系统中安装了同一软件的多个版本时，不确定使用的是哪个版本时）

• 文件/目录管理

创建: mkdir

删除: rm

删除非空目录: rm -rf file目录

删除日志 rm *log (等价: \$find ./ -name "*log" -exec rm {} ;)

移动: mv

重命名: mv/rename

touch 文件名, 创建一个空白文件

复制: cp (复制目录: cp -r)

切换到上一个工作目录: cd -或者cd ..

切换到home目录: cd or cd ~

显示当前路径: pwd

显示当前目录下的文件 ls ls -a 显示隐藏文件

按时间排序, 以列表的方式显示目录项 ls -lrt

查看两个文件间的差别: diff file1 file2

在文件内查找指定的字符串: egrep 执行效果与"grep-E"相似

Eg. egrep Linux * #查找当前目录下包含字符串“Linux”的文件

改变文件的拥有者 chown

改变文件读、写、执行等属性

创建符号链接/硬链接: ln 加上命令-s为软链接 软连接等价于起别名 硬链接等价于复制一个新文件

cat (英文全拼: concatenate) 命令用于连接文件并打印到标准输出设备上。

-n 或 --number: 由 1 开始对所有输出的行数编号。

-b 或 --number-nonblank: 和 -n 相似, 只不过对于空白行不编号

> 可以进行输出重定向 cat默认输出到标准输出流

与cat结合: cat -n textfile1 > textfile2 // 把 textfile1 的文档内容加上行号后输入 textfile2 这个文档里

file命令用于辨识文件类型。 file [-bcLrvz][-f <名称文件>][-m <魔法数字文件>...][文件或目录...]

-b 列出辨识结果时, 不显示文件名称。

-c 详细显示指令执行过程, 便于排错或分析程序执行的情形。

-f<名称文件> 指定名称文件, 其内容有一个或多个文件名称时, 让file依序辨识这些文件, 格式为每列一个文件名称。

-L 直接显示符号连接所指向的文件的类别。

-m<魔法数字文件> 指定魔法数字文件。

-v 显示版本信息。

-z 尝试去解读压缩文件的内容。

[文件或目录...]为 要确定类型的文件列表, 多个文件之间使用空格分开, 可以使用shell通配符匹配多个文件。

ar命令用于建立或修改备存文件, 或是从备存文件中抽取文件。 ar可让您集合许多文件, 成为单一的备存文件。在备存文件中, 所有成员文件皆保有原来的属性与权限。

d 删除备存文件中的成员文件。

m 变更成员文件在备存文件中的次序。

p 显示备存文件中的成员文件内容。

q 将文件附加在备存文件末端。

r 将文件插入备存文件中。

t 显示备存文件中所包含的文件。

x 自备存文件中取出成员文件。

a<成员文件> 将文件插入备存文件中指定的成员文件之后。

- b<成员文件> 将文件插入备存文件中指定的成员文件之前。
- c 建立备存文件。
- i<成员文件> 将文件插入备存文件中指定的成员文件之前。
- o 保留备存文件中文件的日期。
- s 若备存文件中包含了对象模式，可利用此参数建立备存文件的符号表。
- S 不产生符号表。
- v 程序执行时显示详细的信息。

Eg. ar rv one.bak a.c b.c //打包 a.c b.c文件

• 文本处理

文件查找: find

Eg. 查找txt和pdf文件: find . \(-name \"*.txt\" -o -name \"*.pdf\" \) -print

按类型搜索 -type f 文件 /l 符号链接 /d 目录

按时间搜索

-atime 访问时间 (单位是天, 分钟单位则是-amin, 以下类似)

-mtime 修改时间 (内容被修改)

-ctime 变化时间 (元数据或权限变化)

按大小搜索 -size

找到后的操作: 删除 -delete 打印 -print 执行动作 -exec+命令

文本搜索: grep

-o 只输出匹配的文本行 VS -v 只输出没有匹配的文本行

-c 统计文件中包含文本的次数

-n 打印匹配的行号

-i 搜索时忽略大小写

-l 只打印文件名

xargs 能够将输入数据转化为特定命令的命令行参数; 是给命令传递参数的一个过滤器, 也是组合多个命令的一个工具。

之所以能用到这个命令, 关键是由于很多命令不支持|管道来传递参数, 而日常工作中有这个必要

-d 定义定界符 (默认为空格 多行的定界符为 n)

-n 指定输出为多行

-I {} 指定替换字符串, 这个字符串在xargs扩展时会被替换掉, 用于待执行的命令需要多个参数时

-0: 指定0为输入定界符

Eg. find . type f -name "*.swp" | xargs rm

对文件内容进行排序: sort

-n 按数字进行排序 VS -d 按字典序进行排序

-r 逆序排序

-k N 指定按第N列排序

uniq 命令用于检查及删除文本文件中重复出现的行列, 一般与 sort 命令结合使用。

-c或--count 在每列旁边显示该行重复出现的次数。

-d或--repeated 仅显示重复出现的行列。

-u或--unique 仅显示出一次的行列。

-w<字符位置>或--check-chars=<字符位置> 指定要比较的字符。

tr 命令用于转换或删除文件中的字符。tr 指令从标准输入设备读取数据, 经过字符串转译后, 将结果输出到标准输出设备
tr [OPTION] SET1 [SET2]

-d, --delete: 删除指令字符

-c, --complement: 反选设定字符。也就是符合 SET1 的部份不做处理, 不符合的剩余部份才进行转换

-s, --squeeze-repeats: 缩减连续重复的字符成指定的单个字符 常用于压缩多余的空格 cat file | tr -s ''

`cut`命令用于显示每行从开头算起 `num1` 到 `num2` 的文字。`cut` 命令从文件的每一行剪切字节、字符和字段并将这些字节、字符和字段写至标准输出。

- c : 以字符为单位进行分割。
- d : 自定义分隔符，默认为制表符。
- f : 与-d一起使用，指定显示哪个区域。

`wc` 统计行和字符的工具

- l // 统计行数
- w // 统计单词数
- c // 统计字符数

• 磁盘管理

查看磁盘空间利用大小: `df -h`

-h: human缩写，以易读的方式显示结果（即带单位：比如M/G，如果不加这个参数，显示的数字以B为单位）

查看目录所占空间大小:`du -sh`

- s 递归整个目录的大小
- h 人性化显示

打/解包: `tar`

`tar -cvf etc.tar /etc` #仅打包，不压缩！ 压缩用`zip`

`tar -xvf demo.tar`

- x 表示进行解包
- c :表示进行打包, 即将所有文件放到一个文件夹中
- v :显示打包进度
- f :使用档案文件
- z 解压gz文件
- j 解压bz2文件
- J 解压xz文件

• 进程管理

任何进程都与文件关联；我们会用到`lsof`工具（list opened files），作用是列举系统中已经被打开的文件。在linux环境中，任何事物都是文件，设备是文件，目录是文件，甚至sockets也是文件。用好`lsof`命令，对日常的linux管理非常有帮助。

查询正在运行的进程信息: `ps`

- A/-e 列出所有的进程
- w 显示加宽可以显示较多的资讯
- a: 显示终端上的所有进程，包括其他用户的进程
- u: 显示进程的详细信息
- x: 显示没有控制终端的进程
- j: 列出与作业控制相关的信息

查找指定进程: `ps -ef | grep 进程关键字`

`ps auxw --sort=-%cpu` 查看CPU使用率最高的进程

`--sort=-%cpu` 代表降序 直接 `--sort=%cpu` 表示升序

`ppid` 父进程id `pid`进程id `pgid`进程组id 输入参数`j`时可以看到这些信息

显示进程信息，并实时更新: `top`

在`top`界面输入字符命令后显示相应的进程状态

P: 根据CPU使用百分比大小进行排序。

M: 根据驻留内存大小进行排序。

i: 使`top`不显示任何闲置或者僵死进程。

m: 显示或隐藏内存状态信息

s: 设置刷新时间间隔

<Space>: 立即刷新

q: 退出`top`命令

T: 按MITE+排行

K: kill进程

查看端口占用的进程状态: lsof #直接加文件名 lsof abc.txt: 显示开启文件abc.txt的进程

-i: 查看端口占用 #lsof -i:8080 查看8080端口占用

-c 显示该进程现在打开的文件 # -c -p 1234 列出进程号为1234的进程所打开的文件

-u *username*: 查看用户*username*的进程所打开的文件

+d : 查询指定目录下被进程开启的文件 (使用+D 递归目录)

结束进程: kill #直接kill pid也可以杀死进程 加-9是彻底杀死

kill 本质上是将指定的信息送至程序。预设的信息为 SIGTERM(15), 可将指定程序终止。若仍无法终止该程序, 可使用 SIGKILL(9) 信息尝试强制删除程序。程序或工作的编号可利用 ps 指令或 jobs 指令查看。

-1 (HUP): 重新加载进程。

-9 (KILL): 杀死一个进程。

-15 (TERM): 正常停止一个进程。

-l <信息编号> 若不加<信息编号>选项, 则 -l 参数会列出全部的信息名称。

-s <信息名称或编号> 指定要送出的信息。#ps -s 9 pid

• 性能监控

查看CPU使用率: sar -u n m (n 为监控频率、m为监控次数)

查看内存使用状况: sar -r n m

当系统中sar不可用时, 可以使用以下工具替代: vmstat

• 网络工具

• ifconfig查看ip

Linux netstat 命令用于显示网络状态。利用 netstat 指令可让你得知整个 Linux 系统的网络情况。

-a或--all 显示所有连线中的Socket。

-c或--continuous 持续列出网络状态。

-e或--extend 显示网络其他相关信息。

-p或--programs 显示正在使用Socket的程序识别码和程序名称。

-r或--route 显示Routing Table。

-t或--tcp 显示TCP传输协议的连线状况。

-u或--udp 显示UDP传输协议的连线状况。

-l或--listening 显示监控中的服务器的Socket。

-n或--numeric 直接使用IP地址, 而不通过域名服务器。

Eg.	
列出所有 tcp 端口:	netstat -at
使用netstat工具查询端口:	netstat -antp grep 6379

查询7902端口现在运行什么程序:

```
#分为两步
#第一步, 查询使用该端口的进程的PID;
$ lsof -i:7902
COMMAND PID USER FD TYPE DEVICE SIZE NODE NAME
WSL    30294 tuapp  4u IPv4 447684086      TCP 10.6.50.37:tnos-dp (LISTEN)

#查到30294
#使用ps工具查询进程详情:
$ ps -fe | grep 30294
tdev5 30294 26160 0 Sep10 ? 01:10:50 tdesl -k 43476
root   22781 22698 0 00:54 pts/20 00:00:00 grep 11554
```

查看路由状态: route -n

探测前往地址IP的路由路径: traceroute /P

DNS查询, 寻找域名domain对应的IP: host domain

ssh登陆远程服务器host, ID为用户名: ssh ID@host

ftp/sftp文件传输: sftp ID@host

ftp登陆后, 可以使用下面的命令进一步操作:

get filename # 下载文件

put filename # 上传文件

ls # 列出host上当前路径的所有文件

```
cd # 在host上更改当前路径  
lls # 列出本地主机上当前路径的所有文件  
lcd # 在本地主机更改当前路径
```

网络复制:

```
将本地localpath指向的文件上传到远程主机的path路径: scp localpath ID@host:path  
以ssh协议, 遍历下载path路径下的整个文件系统, 到本地的localpath: scp -r ID@site:path localpath
```

• 用户管理

添加用户: useradd -m *username*

为用户添加/修改密码: passwd *username*

删除用户: userdel -r *username* #不带选项使用 userdel, 只会删除用户。用户的home目录将仍会在/home目录下。要完全的删除用户信息, 使用-r选项

切换到用户B: su *userB*

查看用户当前组: groups

变更用户组: usermod -G groupNname *username* # 一个用户可以属于多个组 -G是将用户加入到组, -g将用户加入到新的组, 并从原有的组中除去

使用chmod命令可以更改文件的读写权限, 更改读写权限有两种方法, 一种是字母方式, 一种是数字方式

字母方式: chmod 【-R】 userMark(+|-) PermissionsMark -R代表递归

+|-代表增加或删除权限

userMark取值: u: 用户 g: 组 o: 其它用户 a: 所有用户

PermissionsMark取值: r: 读 w: 写 x: 执行

chmod a+x main 对所有用户给文件main增加可执行权限

chmod g+w blogs 对组用户给文件blogs增加可写权限

数字方式:

数字方式直接设置所有权限, 相比字母方式, 更加简洁方便;

使用三位八进制数字的形式来表示权限, 第一位指定属主的权限, 第二位指定组权限, 第三位指定其他用户的权限, 每位通过4(读)、2(写)、1(执行)三种数值的和来确定权限。如6(4+2)代表有读写权, 7(4+2+1)有读、写和执行的权限。

chmod 740 main 将main的用户权限设置为rwxr----

更改文件或目录的拥有者: chown *username dirOrFile* #使用-R选项递归更改该目下所有文件的拥有者

vim基础

2023年2月28日 15:31

移动

- j, 下
- k, 上
- h, 左
- l, 右
- v, 按v之后按方向键可以选中你要选中的文字
- gg, 跳到第1行
- G, 跳到最后一行
- 16G或:16, 跳到第16行
- \$, 到本行行尾
- 0, 到本行行头
- ^ → 到本行的第一个非blank字符
- g_ → 到本行最后一个不是blank字符的位置。
- %, 匹配括号移动, 包括(), {}, []. (需要把光标先移到括号上)
- *, 匹配光标当前所在的单词, 移动光标到下一个 匹配单词
- #, 匹配光标当前所在的单词, 移动光标到上一个 匹配单词
- /pattern → 搜索 pattern 的字符串 (如果搜索出多个匹配, 可按n键到下一个)
- fa → 到下一个为a的字符处, 你也可以fs到下一个为s的字符。
- 3fa → 在当前行查找第三个出现的a。
- t, → 到逗号前的第一个字符。逗号可以变成其它字符。
- F和T → 和f和t一样, 只不过是相反方向。

插入

- i, 在当前行首插入
- A, 在当前行尾插入
- i, 在当前字符的左边插入
- a, 在当前字符的右边插入
- o, 在当前行下面插入一个新行
- O, 在当前行上面插入一个新行

删除

- x, 删除光标后的1个字符
- 2x, 删除光标后的2个字符
- X, 删除光标前的1个字符
- 2X, 删除光标前的2个字符

- dd, 删除当前行
- cc, 删除当前行后进入 insert 模式
- dw, 删除当前光标下的单词/空格
- d\$, 删除光标至 行尾 所有字符
- dG, 删除光标至 文件尾 所有字符
- 3dd, 从当前光标开始, 删掉 3 行
- dt" → 删除所有的内容, 直到遇到双引号—— "
- echo > aa.txt, 从 bash 角度清空文件内容, 这个比较高效

复制

- y, 复制光标所选字符
- yw, 复制光标后单词
- yy, 复制当前行
- 4yy, 复制当前行及下面 4 行
- y\$, 复制光标位置至 行尾 的内容
- y^, 复制光标位置至 行首 的内容

粘贴

- p, 将粘贴板中内容复制到 光标之后
- P, 将粘贴板中内容复制到 光标之前
- :%s/YouMeek/Judasn/g, 把文件中所有 YouMeek 替换为: Judasn
- :%s/YouMeek/Judasn/, 把文件中所有行中第一个 YouMeek 替换为: Judasn
- :s/YouMeek/Judasn/, 把光标当前行第一个 YouMeek 替换为 Judasn
- :s/YouMeek/Judasn/g, 把光标当前行所有 YouMeek 替换为 Judasn
- :s#YouMeek/#Judasn#, 除了使用斜杠作为分隔符之外, 还可以使用 # 作为分隔符, 此时中间出现的 / 不会作为分隔符, 该命令表示: 把光标当前行第一个 YouMeek/ 替换为 Judasn/
- :10,31s/YouMeek/Judasn/g, 把第 10 行到 31 行之间所有 YouMeek 替换为 Judasn

其他

- guu, 把当前行的字母全部转换成 小写
- gUU, 把当前行的字母全部转换成 大写
- g~~, 把当前行的字母是大写的转换成小写, 是小写的转换成大写
- u, 撤销
- Ctrl+r 反撤销
- Vim 提供了 12 个剪贴板, 分别是: 0,1,2,3,4,5,6,7,8,9,a," , 默认采用的是 " , 也就是 双引号。复制到某个剪切板的命令: "7y, 表示使用 7 号剪切板。黏贴某个剪切板 内容: "7p, 表示使用 7 号剪切板内容进行黏贴

文件操作

- :e <path/to/file> → 打开一个文件

- :saveas <path/to/file> → 另存为 <path/to/file>
- :w → 存盘
- :x, ZZ 或 :wq → 保存并退出 (:x 表示仅在需要时保存, ZZ不需要输入冒号并回车)
- :bn 和 :bp → 你可以同时打开很多文件, 使用这两个命令来切换下一个或上一个文件。 (使用:n也可以到下一个文件)

块操作:

- Ctrl+v 然后移动光标选择块区域 进行操作以后按esc使得操作对于整个块区域生效
- J → 把所有的行连接起来 (变成一行)
- < 或 > → 左右缩进
- = → 自动给缩进

GCC

2023年2月28日 15:32

GCC (英文全拼: GNU Compiler Collection) 是 GNU 工具链的主要组成部分，是一套以 GPL 和 LGPL 许可证发布的程序语言编译器自由软件，由 Richard Stallman 于 1985 年开始开发。是 Linux 下使用最广泛的 C/C++ 编译器是 GCC，大多数的 Linux 发行版本都默认安装，不管是开发人员还是初学者，一般都将 GCC 作为 Linux 下首选的编译工具。

gcc (GNU C Compiler) 是GCC中的c编译器，而g++ (GNU C++ Compiler) 是GCC中的c++编译器。

gcc和g++两者都可以编译c和cpp文件，但存在差异。gcc在编译cpp时语法按照c来编译但默认不能链接到c++的库（gcc默认链接c库，g++默认链接c++库）。g++编译.c和.cpp文件都统一按cpp的语法规则来编译。所以一般编译c用gcc，编译c++用g++

直接使用

gcc main.c # 在 gcc 命令后面紧跟源文件名 不管源文件的名字是什么，GCC 生成的可执行文件的默认名字始终是a.out。

不像 Windows，Linux 不以文件后缀来区分可执行文件，Linux 下的可执行文件后缀理论上可以是任意的，这里的.out只是用来表明它是 GCC 的输出文件

-o选项 可以自定义文件名 # **gcc main.c -o out/main**

上面是通过gcc命令一次性完成编译和链接的整个过程，这样最方便，大家在学习C语言的过程中一般都这么做。实际上，gcc命令也可以将编译和链接分开，每次只完成一项任务。

对语法错误的检查是在编译阶段进行的。

1) 预处理 (Preprocessing)

-E选项 进行宏替换 将源文件执行预处理操作 也即生成*.i文件, gcc编译器将对#开头的指令进行解析 # **gcc -E Demo.c -o Demo.i**

可用记事本打开

预处理时会将所有注释都去掉，所以反编译回来的代码是没有注释的！

使用指令gcc -E Demo.c 而使用-o不指定输出的文件名时内容将会直接输出到DOS框中，而不会产生文件

2) 编译 (Compiling)

-S选项 生成汇编代码 将 *.i 文件中源码转化为汇编代码 *.s 文件 用记事本打开会发现c源码已经被编译器转化为汇编代码 # **gcc -S Demo.i -o Demo.s**

如果使用指令 **gcc -S Demo.i** 即不指定输出文件名，默认也将会在当前目录下产生文件Demo.s

也可以直接由 .c 文件生成 .s 汇编文件 # **gcc -S hello.c**

3) 汇编 (Assembling)

-c选项 生成机器可识别代码 将源文件编译成目标文件 # **gcc -c main.c**

对于微软编译器（内嵌在 Visual C++ 或者 Visual Studio 中），目标文件的后缀为.obj；对于 GCC 编译器，目标文件的后缀为.o。

如果希望自定义目标文件的名字，那么仍然可以使用-o选项 **gcc -c main.c -o a.o**

也可以由 .i 或 .s 文件生成目标文件 .o # **gcc -c hello.i -o hello.o gcc -c hello.s -o hello.o**

4) 链接 (Linking)

在gcc命令后面紧跟目标文件的名字，就可以将目标文件链接成为可执行文件

在gcc命令后面紧跟源文件名字或者目标文件名字都是可以的，gcc命令能够自动识别到底是源文件还是目标文件：如果是源文件，那么要经过编译和链接两个步骤才能生成可执行文件；如果是目标文件，只需要链接就可以了。

gcc main.o 就将 main.o 链接为 默认的 a.out

使用-o选项仍然能够自定义可执行文件的名字 # **gcc main.o -o main.out**

-I : 指定头文件的包含路径。

-L : 指定链接库的包含路径。

库是写好的现有的，成熟的，可以复用的代码。现实中每个程序都要依赖很多基础的底层库，不可能每个人的代码都从零开始，因此库的存在意义非同寻常。**本质上来说库是一种可执行代码的二进制形式**，可以被操作系统载入内存执行。库有两种：静态库 (.a、.lib) 和动态库 (.so、.dll)。所谓静态、动态是指链接

使用静态库

之所以称为【静态库】，是因为在链接阶段，会将汇编生成的目标文件.o与引用到的库一起链接打包到可执行文件中。因此对应的链接方式称为静态链接。其实一个静态库可以简单看成是一组目标文件 (.o/.obj文件) 的集合，即很多目标文件经过压缩打包后形成的一个文件。

Linux下使用ar工具、Windows下vs使用lib.exe，将目标文件压缩到一起，并且对其进行编号和索引，以便于查找和检索。

静态库特点总结：

- 静态库对函数库的链接是放在编译时期完成的。
- 程序在运行时与函数库再无瓜葛，移植方便。
- 浪费空间和资源，因为所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。

将 .c 文件编译成静态库 .a 文件

先生成 .o 文件 再由 .o 文件生成 .a 文件

```
gcc -c foo.c      # 生成 foo.o 目标文件  
ar rcs libfoo.a foo.o # 生成 libfoo.a 静态库
```

-static：使用静态链接。

编译 hello.c 并链接静态库 libfoo.a (加上 -static 选项)

```
gcc hello.c -static libfoo.a -o hello
```

也可以使用 -L 指定库的搜索路径，并使用 -l 指定库名

```
gcc hello.c -static -L . -l foo -o hello # -L . 代表搜索当前目录 -l foo 指定库名为 foo
```

使用共享库

为什么需要动态库，其实也是静态库的问题导致。**空间浪费**是静态库的一个问题。另一个问题是静态库对程序的更新、部署和发布页会带来麻烦。如果静态库liba.lib更新了，所以使用它的应用程序都需要**重新编译**、发布给用户（对于玩家来说，可能是一个很小的改动，却导致整个程序重新下载，**全量更新**）。

动态库在程序编译时**并不会被连接到目标代码中，而是在程序运行时才被载入**。不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例，规避了空间浪费问题。动态库在程序运行时才被载入，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，**增量更新**。

-shared：创建共享库/动态库。

由于动态库可以被多个进程共享加载，所以需要使用 **-fPIC** 选项生成位置无关的代码

```
gcc foo.c -shared -fPIC -o libfoo.so
```

编译 hello.c 并链接共享库 libfoo.so

```
gcc hello.c libfoo.so -o hello
```

也可以使用 -L 和 -l 选项指定库的路径和名称

```
gcc hello.c -L . -l foo -o hello
```

但是此时直接运行 hello 程序仍然失败 原因是找不到 libfoo.so 共享库

这是因为 libfoo.so 并不在 Linux 系统的默认搜索目录中，解决办法是我们主动告诉系统，libfoo.so 共享库在哪里。

方式一：设置环境变量 **LD_LIBRARY_PATH**

```
export LD_LIBRARY_PATH=$(pwd)
```

方式二：使用 rpath 将共享库位置嵌入到程序

```
gcc hello.c -L . -lfoo -Wl,-rpath= `pwd` -o hello
```

rpath 即 run path，是一种可以将共享库位置嵌入程序中的方法，从而不用依赖于默认位置和环境变量。这里在链接时使用 -Wl,-rpath=/path/to/yours 选项，-Wl 会发送以逗号分隔的选项到链接器，注意逗号分隔符后面没有空格。

方式三：将 libfoo.so 共享库添加到系统路径

```
sudo cp libfoo.so /usr/lib/
```

通过命令 ldd 可以列出程序的动态依赖

```
wjc@wjc-virtual-machine:~$ ldd hello  
linux-vdso.so.1 (0x00007ffd2eaf1000)  
libfoo.so => /home/wjc/libfoo.so (0x00007fb66627a000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb665e89000)  
/lib64/ld-linux-x86-64.so.2 (0x00007fb66667e000)
```

其他选项

-D：在程序编译的时候，指定一个宏

-B <directory> : 将 <directory> 添加到编译器的搜索路径。
-save-temp : 不用删除中间文件。
-save-temp=<arg> : 不用删除指定的中间文件。
-time : 为每个子流程的执行计时。
-Xassembler <arg> : 将 <arg> 传递给汇编器 (assembler) 。
-Xpreprocessor <arg> : 将 <arg> 传递给预处理器 (preprocessor) 。
-Xlinker <arg> : 将 <arg> 传递给链接器 (linker) 。
-g : 生成调试信息。GNU 调试器可利用该信息, 否则不能使用GDB进行调试。
-O0/1/2/3 : 编译器的优化选项的4个级别, -O0表示没有优化,-O1为缺省值, -O3优化级别最高。
-Wall 生成所有警告信息。
-MM 输出依赖关系 (不包含标准库文件)

gcc编译多个文件

编译这三个文件, 可以一次编译:

```
gcc hello.c main.c -o main 生成可执行文件main
```

也可以独立编译:

```
gcc -Wall -c main.c -o main.o  
gcc -Wall -c hello.c -o hello.o  
gcc -Wall main.o hello.o -o main
```

独立编译的好处是, 当其中某个模块发送改变时, 只需要编译该模块就行, 不必重新编译所有文件, 这样可以节省编译时间。

GDB

2023年3月1日 14:30

GDB (GNU Debugger) 是UNIX及UNIX-like下的强大调试工具，可以调试ada, c, c++, asm, minimal, d, fortran, objective-c, go, java,pascal等语言。

对于C程序来说，需要在编译时加上-g参数，保留调试信息，否则不能使用GDB进行调试。

直接使用 gdb + 文件名

如果没有调试信息，会提示no debugging symbols found。

如果是下面的提示：Reading symbols from helloWorld...done. 则可以进行调试。

接着进入交互界面 输入命令进行调试

run 参数 # 如果运行的程序需要参数 则在run后面直接跟参数即可
或者**start** 默认在最开始有个临时断点

```
#include<stdio.h>
int main(int argc,char *argv[])
{
    if(1 >= argc)
    {
        printf("usage:hello name\n");
        return 0;
    }
    printf("Hello World %s!\n",argv[1]);
    return 0 ;
}
```

```
$ gdb hello
(gdb)run 编程珠玑
Starting program: /home/shouwang/workspaces/c/hello 编程珠玑
Hello World 编程珠玑!
[Inferior 1 (process 20084) exited normally]
(gdb)
```

或者使用**set args**，然后再用run启动

set args 用于传递main函数参数

show args 可以查看参数

当程序运行的过程中**异常终止或崩溃**，操作系统会将程序当时的内存状态记录下来，保存在一个**core文件**中，这种行为就叫做Core Dump（中文有的翻译成“核心转储”）。我们可以认为 core dump 是“内存快照”，但实际上，除了内存信息之外，还有些关键的程序运行状态也会同时 dump 下来，例如寄存器信息（包括程序指针、栈指针等）、内存管理信息、其他处理器和操作系统状态和信息。core dump 对于编程人员诊断和调试程序是非常有帮助的，因为对于有些程序错误是很难重现的，例如指针异常，而 core dump 文件可以再现程序出错时的情景。

调试core文件

当程序core dump时，可能会产生core文件，它能够很大程序帮助我们定位问题。但前提是系统没有限制core文件的产生。可以使用命令**ulimit -c**查看是否允许生成

结果为0则代表不能生成core文件 可以通过以下命令解除限制 两种方式可选其一。第一种无限制，第二种指定最大产生文件的大小。

ulimit -c unlimited #表示不限制core文件大小

ulimit -c 10 #设置最大大小，单位为块，一块默认为512字节

使用core文件：

(gdb) core-file core 就可以查看生成的core文件

调试已运行程序

首先使用 **ps** 命令找到进程id 接着用 **attach + id** 启动调试

```
ps -ef|grep 进程名
$ gdb
```

(gdb) attach 20829

断点设置

Gdb模式下可以使用**info break**查看已设置断点
或者简写为 **i b**

根据行号设置断点 **b/break 9 #break** 可简写为**b** 或者 **b test.c: 9** 程序运行到第9行的时候会断住

根据函数名设置断点 **b printNum # printNum** 为函数名 程序在调用到printNum函数的时候会断住

根据条件设置断点

假设程序某处发生崩溃，而崩溃的原因怀疑是某个地方出现了非期望的值，那么你就可以在这里断点观察，当出现该非法值时，程序断住。这个时候我们可以借助gdb来设置条件断点

break test.c:23 if b==0 # 当在b等于0时，程序将会在第23行断住。

或者**condition**有着类似的作用，假设上面的断点号为1，那么：

condition 1 b==0 # 会使得b等于0时，产生断点1

根据规则设置断点

对所有以printNum开头的函数都设置断点，可以使用下面的方式：

rbreak printNum* # 所有以printNum开头的函数都设置了断点

设置临时断点 **tbreak**

tbreak test.c: 10 # 在第10行设置临时断点

跳过多次设置断点 **ignore**

假如有某个地方，我们知道可能出错，但是前面30次都没有问题，虽然在该处设置了断点，但是想跳过前面30次：

ignore 1 30 # 1为断点号 30是跳过次数

根据表达式值变化产生断点 **watch** # **watch a** 当变量a的值变化时 中断并打印a的值

rwatch 和 **awatch** 同样可以设置观察点

前者是当变量值被读时断住，后者是被读或者被改写时断住。

禁用或启动断点

disable #禁用所有断点
disable bnum #禁用标号为bnum的断点
enable #启用所有断点
enable bnum #启用标号为bnum的断点
enable delete bnum #启动标号为bnum的断点，并且在此之后删除该断点

断点清除

clear #删除当前行所有breakpoints
clear function #删除函数名为function处的断点
clear filename: function #删除文件filename中函数function处的断点
clear lineNumber #删除行号为lineNumber处的断点
clear filename: lineNumber #删除文件filename中行号为lineNumber处的断点
delete #删除所有breakpoints,watchpoints和catchpoints
delete bnum #删除断点号为bnum的断点

变量查看

普通变量查看

使用**print** (可简写为**p**) 打印变量内容 **# p a**

有时候，多个函数或者多个文件会有同一个变量名，这个时候可以在前面加上函数名或者文件名来区分

p 'main'::b

打印指针指向内容

使用解引用方式打印 **# p *d**

仅仅使用*只能打印第一个值，如果要打印多个值(如数组)，后面跟上@**并加上要打印的长度。**

p *d@10

\$ 可表示上一个变量

假设此时有一个链表linkNode，它有next成员代表下一个节点，则可使用下面方式不断打印链表内容：

```
(gdb) p *linkNode  
(这里显示linkNode节点内容)  
(gdb) p *$.next  
(这里显示linkNode节点下一个节点的内容)
```

使用 **set** 可以定义一个变量

如果想要查看前面数组的内容，可以定义下标变量后，将下标一个一个累加

```
(gdb) set $index=0  
(gdb) p b[$index++]  
$11 = 1  
(gdb) p b[$index++]  
$12 = 2  
(gdb) p b[$index++]  
$13 = 3
```

按照特定格式打印变量

对于简单的数据，print默认的打印方式已经足够了，它会根据变量类型的格式打印出来，但是有时候这还不够，我们需要更多的格式控制。

/x 按十六进制格式显示变量。
/d 按十进制格式显示变量。
/u 按十六进制格式显示无符号整型。
/o 按八进制格式显示变量。
/t 按二进制格式显示变量。
/a 按十六进制格式显示变量。
/c 按字符格式显示变量。
/f 按浮点数格式显示变量。

还是以辅助程序来说明，正常方式打印字符数组c：

```
1 (gdb) p c  
2 $18 = "hello,shouwang"
```

但是如果我们要查看它的十六进制格式打印呢？

```
1 (gdb) p/x c  
2 $19 = {0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x73, 0x68, 0x6f, 0x75, 0x77, 0x61,  
3 0x6e, 0x67, 0x0}  
4 (gdb)
```

但是这种方式仍然无法查看浮点数的二进制格式，因为直接打印它首先会被转换成整型查看内存内容

查看内存内容

examine(简写为x) 可以用来查看内存地址中的值。语法如下：

x/[n][f][u] addr

- n 表示要显示的内存单元数，默认值为1。
- f 表示要打印的格式，前面已经提到了格式控制字符
- u 要打印的单元长度 单元长度常见有如下：b 字节 h 半字，即双字节 w 字，即四字节 g 八字节
- addr 内存地址

```
(gdb) x/4tb &e  
0x7fffffffdbd4: 00000000 00000000 00001000 01000001  
(gdb)
```

此处 n=4, f = t, u = b 代表打印四个字节大小的e变量所处地址的内存单元中的内容 以二进制形式

自动显示变量内容

假设我们希望程序断住时，就显示某个变量的值，可以使用**display**命令。 # **display e**
那么每次程序断住时，就会打印e的值。要查看哪些变量被设置了display，可以使用：**info display** 命令
如果想要清除可以使用 **delete display [num]** # num为前面变量前的编号,不带num时清除所有。

查看寄存器内容 # **info registers**

单步调试

单步执行

next命令（可简写为n） 用于在程序断住后，继续执行下一条语句，假设已经启动调试，并在第12行停住，如果要继续执行，则使用n执行下一条语句，如果后面跟上数字num，则表示执行该命令num次，就达到继续执行n行的效果了：

单步进入

step命令（可简写为s），它可以单步跟踪到函数内部，但前提是该函数有调试信息并且有源码信息
如果没有函数调用，s的作用与n的作用并无差别，仅仅是继续执行下一行。它后面也可以跟数字，表明要执行的次数。

继续执行到下一个断点

continue命令（可简写为c）或者fg，它会继续执行程序，直到再次遇到断点处

继续运行到指定位置

假如我们在25行停住了，现在想要运行到29行停住，就可以使用**until命令（可简写为u）** 它利用的是临时断点。

跳过执行

skip可以在step时跳过一些不想关注的函数或者某个文件的代码

```
skip function add #step时跳过add函数
info skip #查看step情况
```

skip也后面也可以跟文件：

```
skip file gdbStep.c
```

这样gdbStep.c中的函数都不会进入。

- **skip delete [num]** 删除skip
- **skip enable [num]** 使能skip
- **skip disable [num]** 去使能skip

查看源码或对源码进行编辑

查看源码内容

list（可简写为l），它可以将源码列出来

直接输入l可从第一行开始显示源码，继续输入l，可列出后面的源码。后面也可以跟上 + 或者 -，分别表示要列出上一次列出源码的后部分或者前面部分。 # l - l +

l 后面可以跟行号，表明要列出某行附近的源码 # l 9

列出指定函数附近的源码 l后面跟函数名即可

设置源码一次列出行数

默认是10行 可以通过**set listsize**属性来设置

```
1 (gdb) set listsize 20
2 (gdb) show listsize
3 Number of source lines gdb will list by default is 20.
```

设置为0或者unlimited，这样设置之后，列出就没有限制了，但源码如果较长，查看将会不便。

列出指定行之间的源码

```
list first,last
```

启始行和结束行号之间用逗号隔开。两者之一也可以省略，例如： **list 3,**

省略结束行的时候，它列出从开始行开始，到指定大小行结束，而省略开始行的时候，到结束行结束，列出设置的大小行，例如默认设置为10行，则到结束行为止，总共列出10行。

列出指定文件的源码

I location 其中location可以是 文件名加行号 或 函数名
I test.c: 1 # 查看指定文件指定行
I test.c: printNum1 # 指定文件指定函数
I test.c: 1, test.c: 3 # 指定文件指定行之间

更换源码目录

例如，你编译好的程序文件，放到了另外一台机器上进行调试，或者你的源码文件全都移动到了另外一个目录，怎么办呢？

可以使用**dir**命令指定源码路径 # **dir ./temp**

也可以使用**set substitute-path from [] to []**将原来的路径替换为新的路径

借助 **readelf (bash)**命令可以知道原来的源码路径

readelf main -p .debug_str

编辑源码

为了避免已经启动了调试之后，需要编辑源码，又不想退出，可以直接在gdb模式下编辑源码，它默认使用的编辑器是**/bin/ex**，但是你的机器上可能没有这个编辑器，或者你想使用自己熟悉的编辑器，那么可以通过下面的方式在**bash**下进行设置：

```
EDITOR=/usr/bin/vim  
export EDITOR
```

使用 **edit location** 可以在调试模式下编辑源码

```
edit 3 #编辑第三行  
edit printNum #编辑printNum函数  
edit test.c: 5 #编辑test.c第五行
```

启动时，带上**tui**(Text User Interface)参数，会有意想不到的效果，它会将调试在多个文本窗口呈现：

gdb main -tui

The screenshot shows the GDB interface in TUI mode. On the left, there is a code editor window titled "demo.c" containing the following C code:

```
demo.c  
1  #include<stdio.h>  
2  #include<stdlib.h>  
3  #include"test.h"  
4  
5  int main(){  
6      int a = N;  
7      int b = 2;  
8      int c = 0;  
9  
10     c = a + b;  
11     printf("%d\n",c);  
12  
13     CODE
```

On the right, the GDB command history is displayed:

```
exec No process In:  
(gdb) l test.c  
Function "test.c" not defined.  
(gdb) 4 in /home/wjc/demo.c  
(gdb) l test.c:1  
(gdb) l demo.c:1  
(gdb) b 7  
Breakpoint 1 at 0x788: file demo.c, line 7.  
(gdb) █
```

GIT

2023年3月2日 15:22

Git是一个开源的分布式版本控制系统，可以有效、高速地处理从很小到非常大的项目版本管理

版本控制最主要的功能就是**追踪文件的变更**。它将什么时候、什么人更改了文件的什么内容等信息忠实地记录下来。每一次文件的改变，文件的版本号都将增加。除了记录版本变更外，版本控制的另一个重要功能是**并行开发**。软件开发往往是多人协同作业，版本控制可以有效地解决版本的同步以及不同开发者之间的开发通信问题，提高协同开发的效率。主要功能：

- 检入检出控制：同步控制的实质是版本的检入检出控制。检入就是把软件配置项从用户的工作环境存入到软件配置库的过程，检出就是把软件配置项从软件配置库中取出的过程。检入是检出的逆过程。同步控制可用来确保由不同的人并发执行的修改不会产生混乱。
- 分支与合并：版本分支（以一个已有分支的特定版本为起点，但是独立发展的版本序列）的人工方法就是从主版本——称为主干上拷贝一份，并做上标记。在实行了版本控制后，版本的分支也是一份拷贝，这时的拷贝过程和标记动作由版本控制系统完成。版本合并（来自不同分支的两个版本合并为其中一个分支的新版本）有两种途径，一是将版本A的内容附加到版本B中；另一种是合并版本A和版本B的内容，形成新的版本C。
- 历史记录：版本的历史记录有助于对软件配置项进行审核，有助于追踪问题的来源。历史记录包括版本号、版本修改时间、版本修改者、版本修改描述等最基本的内容，还可以有其他一些辅助性内容，比如版本的文件大小和读写属性。如果我们开发中的新版本发现不适合用户的体验，这时候就可以找到历史记录的响应版本号，回退到历史记录版本！

配置GIT

首先需要确定版本控制人信息 以追踪变更者

@提交信息版本操作者信息：

```
git config --global user.name "Ziph" # 【用户名】  
git config --global user.email "mylifes1110@163.com" # 【邮箱】
```

@查看信息：

```
git config -l
```

仓库

版本库又名仓库，英文名repository，可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

- 工作区：由我们使用命令git init初始化的一个本地文件夹，而初始化后的这个文件夹就被称为工作区
- 暂存区：由我们使用命令git add .把文件添加到暂存区，而被添加的位置则是工作区中.git目录中的index文件，所以这也叫做索引
- 版本库：工作区中会有一个隐藏的.git文件夹，这个不算是工作区，而是Git的版本库，版本库中记录了我们提交过的所有版本，也就是说版本库管理着我们的所有内容
- 分支：版本库中包含若干分支，提交后的文件就会存储在分支中，而开始的时候版本库中只会有一个主分支master

在使用git来管理本地仓库时，每次对工作区中的内容做的任何变化都需要add（添加）和commit（提交）操作来同步git版本库，只有做了添加和提交操作git才能管理我们的工作区。

创建版本库：git init, 先创建或找到一个文件夹，打开命令窗口切换到该路径下，输入git init执行初始化本地工作区，在该工作区内会初始化生成一个.git目录，而该目录就是版本库，它保存着仓库的所有信息.所有有关仓库的指令 如关联远程仓库 克隆等 都必须在已经被git init的文件夹下进行

添加文件: `git add`, 先在工作区中放入一个文件, 然后在命令行窗口中执行`git add 文件名`即可向暂存区中添加一个文件。类似的, `git add 文件名1 文件名2 ...` 即可向暂存区中添加多个文件。`git add 文件夹名` 即可向暂存区中添加该文件夹以及文件夹内的所有内容。使用`git add .`命令来添加工作区中的所有内容到暂存区。

提交文件: `git commit`, 使用`git commit 文件名1 文件名2 -m "本次提交的描述信息"`, 注意提交的描述信息是为了记录本次提交而方便查找, 所以尽量能明确反映本次提交。`git commit . -m "本次提交的描述信息"` 命令来提交所有工作区的文件文件。使用`git commit -a -m "本次添加并提交的描述信息"`命令来自动添加和提交所有文件。

修补提交: 提交后发现有问题, 比如注释忘记修改, 比如提交描述不够详细等等。可以执行`git commit --amend -m "描述信息"`来再次提交替换上次提交

选项 -m 让 Git 将 本次提交的描述信息 记录到项目的历史记录中。

关于向git提交后的文件, 删除和修改我们只需要重新提交即可。也就是说, 我们挪动或删除了工作区中的文件或更改了工作区中的目录结构, 都需要重新向git添加和提交你所变动的文件。

文件状态: 使用`git status` 命令查看文件的状态 是否已提交 是否添加追踪等等

```
root@wjc-virtual-machine:/home/wjc/gittest# git status  
位于分支 master
```

尚无提交

未跟踪的文件:

(使用 "`git add <文件>...`" 以包含要提交的内容)

```
test.c
```

提交为空, 但是存在尚未跟踪的文件 (使用 "`git add`" 建立跟踪)

在Git中, 分支是项目的版本。

- 1: 表示位于分支master上。
- 2: Git指出了项目中未被跟踪的文件, 因为我们还没有告诉它要跟踪哪些文件。
- 3: 被告知没有将任何东西添加到当前提交中, 但我们可能需要将未跟踪的文件加入到仓库中。

- **细节**: 可以使用`git diff` 命令来比对工作区内文件的变动状态
- **比对**: 使用`git diff 文件名` 命令来比对工作区和暂存区 (若暂存区没有则比对分支)
- **比对工作区与分支的最新结果**: 使用`git diff HEAD -- 文件名` 命令来比对工作区和分支的最新结果
- **比对暂存区与分支的最新结果**: 使用`git diff --staged 文件名` 命令来比对暂存区与分支的最新结果
- **注意**: `git diff HEAD -- 文件名`命令--与文件名之间必须要有空格, 不要写错!

日志操作

每次提交, git都会随着提交的变动来记录版本变化, 所以在工作区中的所有操作都会留下日志。

- **查看所有提交日志**: 使用`git log` 命令来显示从最早的提交点到当前提交点的所有日志
- **查看执行条数的提交日志**: 使用`git log -数量` 命令来显示最近指定数量条的提交日志
- **简洁日志显示**: 使用`git log --oneline` 命令来显示比较简洁的提交日志
- **图形化显示分支走向**: 使用`git log --oneline --graph` 命令来图形化显示分支走向

`git log` 命令显示的提交ID是很长的字符串, 而使用`git log --oneline`命令来简洁显示的提交ID是一个7位的字符串
我们使用ID的时候并不需要使用很长的ID来操作, 而一般使用前7位

版本回退

每次修改文件并添加和提交。git都会记录一个版本, 如果有需要可以回退到之前的数据版本状态

- **回退上一个版本**: 使用`git reset --hard HEAD~` 命令来回退到上一个版本
- **回退上上个版本**: 使用`git reset --hard HEAD~~` 命令来回退到上上个版本
- **回退到上某数量个版本**: 使用`git reset --hard HEAD~数量` 命令来回退到上某数量个版本
- **回退到某次提交时的版本**: 使用`git reset --hard commitID` 命令来回退到某次提交时的版本

回退版本并不会删除任何版本, 所以版本之间可以来回切换

撤销修改

工作区撤销：执行 `git checkout -- 文件名` 命令可以撤销到最近一次 `git add` 或 `git commit` 的状态

工作区撤销内部流程：你执行了工作区撤销命令，如果暂存区有此文件，则将暂存区中的文件恢复到工作区中；如果暂存区没有此文件，则将分支中的文件恢复到工作区中

暂存区撤销：先执行 `git reset HEAD 文件名` 命令将该文件移除暂存区，后执行 `git checkout -- 文件名` 命令回退到上一个版本

暂存区撤销场景：如果在工作区中修改了文件并发送到了暂存区中，但文件中有需要撤销的内容

分支

每一个被git管理的仓库都会有一个默认的主分支（master分支）。分支中接收git commit提交的内容，为一个一个不断向前发展的提交点。每个提交点都保存了一个版本。每个分支，都有一个指针，指针默认指向最近一次提交的版本。工作区中的内容，初始状态，就是指针所指向的版本状态。git在保存每个版本时（对应提交点），并不是复制所有文件，没有改动的文件只是记录一个链接。

实际开发中master分支尽量只存放稳定的代码提交，保证master分支稳定，有效。因为这样保证了我们的开发进度不会受到影响。在需要编写新功能时，新建一个开发用的分支，所有改动都在该分支上提交，在功能完整实现后，将该分支的最终内容合并到master分支。这样，既保证开发中有多个版本可以控制，又保证master分支是稳定，有效的。

- **创建分支：**使用 `git branch 分支名` 命令创建分支，会与当前分支保持同样的数据状态，即新分支和当前分支指向同一个提交点
- **切换分支：**使用 `git checkout 分支名` 命令切换分支，切换分支后工作区中显示当前分支内容（切换分支实际上是切换了分支的指针，让指针指向了所要切换到分支）
- **查看当前分支：**使用 `git branch` 命令来查看当前分支
- **查看当前分支详细信息：**使用 `git branch -vv` 命令查看分支详细信息，分支信息则是所跟踪的远程分支信息以及是否领先远程分支等等
- **合并分支：**如果新分支编写完成后，先使用 `git branch master` 命令切换到master分支，再使用 `git merge 新分支名` 命令将新分支合并到master分支。此次合并就是将master的指针移到了新分支的位置，等价于快速合并
- **查看当前合并分支：**分支合并后可以使用 `git branch --merged` 命令查看被当前分支合并了的分支
- **删除分支：**将分支合并后，如果新分支不再继续使用，可以先使用 `git branch --merged` 命令查看合并分支以确认我们即将删除的分支的确是无用分支后，再使用 `git branch -d 分支名` 命令删除需要删除的无用分支。

解决分支冲突

两个分支对同一个文件做了改动，所以在合并时git会无法确定保留哪个分支上的数据就会产生冲突

终止合并分支：当出现分支冲突时可以使用 `git merge --abort` 命令来终止合并分支

避免因为空白导致冲突：在合并分支时，如果有空白内容有可能会出现分支冲突现象，所以此时可以使用 `git merge 分支名 -Xignore-all-space` 命令来避免因为空白而导致的分支冲突

远程仓库（Github）

git本地仓库和GitHub或码云之间传输，建议设置SSH key，避免在传输中反复输入密码

设置SSH key：执行 `ssh-keygen -t rsa -C 邮箱` 命令后的每一步都按Enter键确定就好，知道命令执行结束（-C 后面的内容随意写就行，这只是作为title而已）

如果key创建在root下就必须以root用户才能传输 创建在哪个用户文件夹底下就哪个用户能传。

命令执行完毕后，会在你电脑的C:\Users\主机名\.ssh目录下生成密钥文件。**id_rsa**是私钥，不能泄露出去。**id_rsa.pub**是公钥，可以放心地告诉任何人。

随后注册登录GitHub，在账户设置中选择**SSH Keys**，在Title中随意填写内容，在Key中填写id_rsa.pub文件中的所有内容在GitHub中添加好自己的公钥，这样和Git服务器通信时（clone, push, pull）git服务器就可以识别出你的身份了！

- **关联远程仓库**: 关联远程仓库只需要执行 `git remote add 关联别名 仓库地址` 命令即可 (注意: 别名是可以自己取名设置的, 但是不要忘记就好, 因为后续push的时候会用到) (仓库地址在code下ssh里 形如`git@github.com:wjcsw/test.git`)
- **上传到GitHub远程仓库**: 执行 `git push 关联别名 master` 命令将文件上传到GitHub服务器的主master分支
上传到GitHub远程仓库后, 我们就可以正常的在GitHub查看所上传的文件。设置一次关联后, 我们在本地仓库上传到GitHub远程仓库都需要 `add -> commit -> push`
- **查看关联的所有远程仓库**: 执行`git remote -v`命令查看关联的所有远程仓库
- **查看关联后远程仓库分支和本地仓库分支的对应关系**: 执行`git remote show` 关联别名命令查看
- **删除关联**: 执行 `git remote remove 关联别名` 命令删除关联
- **重命名关联别名**: 执行`git remote rename 原关联别名 新关联别名`命令重命名关联别名
- **上传到GitHub远程仓库**: 执行`git push 关联别名 master`命令来将本地仓库的文件上传到GitHub远程仓库显示 (注意: 我们是可以指定上传的分支的!)
- **本地存在分支上传GitHub分支**: 执行`git push 关联别名 本地仓库分支:GitHub仓库分支`命令会将本地仓库存在分支上传到GitHub分支
- **拉取远程仓库分支**: 执行`git fetch 关联别名 master`命令来拉取master分支下的内容
- **手动合并本地库分支**: 执行`git merge 关联别名/master`命令来手动合并本地库分支下的内容
- **比较拉取内容中的分支和本地分支中的不同**: 首先执行`git checkout` 分支命令来切换到想要比较并拉取的分支, 再执行`git diff` 关联别名/分支命令来比较拉取的内容中的分支和本地分支的不同

下载操作等价于拉取远程的新内容, 并合并到当前分支的操作

下载远程内容: 可以执行`git pull 关联别名 master`命令来完成对远程仓库主分支内容的下载操作, 该操作省略了本地仓库分支 (当前分支), 默认的将远程仓库master主分支上的内容下载到了本地仓库的master主分支

下载远程内容的完整写法: `git pull 关联别名 远程仓库分支:本地仓库分支 (当前分支)`

克隆操作将GitHub远程仓库的所有内容下载到本地, 该方式自动搭建了本地与GitHub远程仓库的关联

clone操作1: 执行命令`git clone SSH地址`将远程仓库clone到本地, 已设置key, 不用命令

clone操作2: 执行命令`git clone HTTPS地址`将远程仓库clone到本地, 该方式需要输入GitHub口令

标签常用命令

```
1 #创建标签
2
3 ##对当前版本建立标签
4 git tag tagname
5
6 ##对历史版本建立标签
7 git tag tagname commit_id
8
9 ##commit_id 添加说明
10 git tag -a tagname -m "描述..."
11
12 ##查看所有标签
13 git tag
14
15 ##查看某个标签具体信息
16 git show tagname
17
18 #删除标签
19
20 ##删除本地标签
21 git tag -d tagname
22
23 #推送标签
24
25 ##推送本地的某个标签到远程
26 git push origin tagname
27
28 ##一次性推送所有分支
29 git push origin -tags
30
31 #删除远程标签
32
33 ##先删除本地
34 git tag -d tagname
35
36 ##从远程删除
37 git push origin :refs/tags/tagname
```

Makefile

2023年3月3日 15:18

如果程序包含很多个源文件，用gcc命令逐个去编译时，就发现很容易混乱而且工作量大，所以出现了make工具

make工具可以看成是一个智能的批处理工具，它本身并没有编译和链接的功能，而是用类似于批处理的方式——通过调用makefile文件中用户指定的命令来进行编译和链接的。

makefile关系到了整个工程的编译规则。一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令。makefile命令中就包含了调用gcc（也可以是别的编译器）去编译某个源文件的命令。但是当工程非常大的时候，手写makefile也是非常麻烦的，如果换了个平台makefile又要重新修改，这时候就出现了Cmake这个工具。

cmake就可以更加简单的生成makefile文件给make用。当然cmake还可以跨平台生成对应平台能用的makefile，我们就不用再自己去修改了。cmake根据一个叫CMakeLists.txt文件（学名：组态档）去生成makefile。

总的来说就是由cmake来跨平台生成makefile，再由make/ninja等编译工具根据makefile的规范进行编译

Makefile的规则：

```
target ... : prerequisites ...
    command
...
...
```

targets是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。

command是命令行，如果其不与“target:prerequisites”在一行，那么，必须以[Tab键]开头，如果和prerequisites在一行，那么可以用分号做为分隔。

prerequisites也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“过时的”，被认为是需要重生成的。

如果命令太长，可以使用反斜框（'/'）作为换行符。make对一行上有多少个字符没有限制。规则告诉make两件事，文件的依赖关系和如何生成目标文件。
一般来说，make会以UNIX的标准Shell，也就是/bin/sh来执行命令。

这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在command中。说白一点就是说，prerequisites中如果有任何一个以上的文件比target文件要新的话，command所定义的命令就会被执行。这就是Makefile的规则。也就是Makefile中最核心的内容。

我们可以在一个makefile中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，清除中间文件等等。定义一个目标文件其冒号后什么也没有，那么，make就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在make命令后明显得指出这个label的名字。

```
demo.out:demo.o test.o
    gcc demo.o test.o -o demo.out

demo.o:demo.c test.h
    gcc -c demo.c -o demo.o

test.o:test.c test.h
    gcc -c test.c -o test.o

clean:
    rm demo.out demo.o test.o
```

其中clean为清除中间文件的命令 调用时使用make clean调用

在默认的方式下，也就是我们只输入make命令。那么，

1、make会在当前目录下找名字叫“Makefile”或“makefile”的文件。要指定特定的Makefile，你可以使用make的“-f”和“--file”参数，如：make -f Make.Linux或make --file Make.AIX。

2、如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“demo.out”这个文件，并把这个文件作为最终的目标文件。

3、如果demo.out文件不存在，或是demo.out所依赖的后面的.o文件的文件修改时间要比edit这个文件新，那么，他就会执行后面所定义的命令来生成edit这个文件。

4、如果demo.out所依赖的.o文件也存在，那么make会在当前文件中找目标为.o文件的依赖性，如果找到则再根据那一个规则生成.o文件。（这有点像一个堆栈的过程）

5、当然，你的C文件和H文件是存在的啦，于是make会生成.o文件，然后再用.o文件生命make的终极任务，也就是执行文件demo.out了。

这就是整个make的依赖性，make会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么make就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make根本不理。

Makefile里主要包含了五个东西：**变量定义、显式规则、隐晦规则、文件指示和注释**。

1、变量的定义。Makefile中的变量其实就是C/C++中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。例如上面我们需要编辑的文件有两个，那么我们可以将其定义为 **objects = test.o demo.o** 以帮助我们在更多文件时减少重复使用。使用变量时 **\$(@)**。

可以让通配符在变量中展开 使用关键字 **wildcard objects := \$(wildcard *.o)** objects的值是所有[.o]的文件名的集合

2、显式规则。**显式规则说明了，如何生成一个或多个的目标文件**。这是由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。刚才写的疑似shell脚本的Makefile全部都是显示规则。

3、隐晦规则。由于我们的**make有自动推导的功能**，所以**隐晦的规则可以让我们比较粗糙地简略地书写Makefile，这是由make所支持的**。

4、文件指示。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。

5、注释。Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用“#”字符，这个就像C/C++中的“//”一样。如果你要在你的Makefile中使用“#”字符，可以用反斜杠进行转义，如：“/#”。

make很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的make会自动识别，并自己推导命令。只要make看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，并且命令 **gcc -c xx.c** 也会被推导出来，于是，我们的makefile再也不用写得上面这么复杂。

同时 也可以将多个目标文件写在一起共享同一个依赖

为了避免和文件重名的情况，我们可以使用一个特殊的标记”.PHONY”来显示地指明一个目标是“伪目标”，向make说明，不管是否有这个文件，这个目标就是“伪目标”。

”.PHONY”表示，clean是个伪目标文件

```
obj = demo.o test.o

demo.out: $(obj)
    gcc $(obj) -o demo.out

$(obj): test.h

.PHONY: clean
clean:
    rm demo.out $(obj)
```

rm命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。

.SUFFIXES: 用于配置“后缀规则”。后缀规则是.a.b形式的规则，例如您在.f.o规则中看到的规则。它们是一种告诉make的方法，只要您看到例如.f文件(源文件)，就可以按照该规则从其中创建.o文件(目标文件)。

.SUFFIXES: .c .o

.c.o:

gcc -c -o \$@ \$<

这等效于%形成模式规则：

%.o: %.c

gcc -c -o \$@ \$<

在Makefile使用**include**关键字可以把别的Makefile包含进来，这很像C语言的**#include**，被包含的文件会原模原样的放在当前文件的包含位置。include的语法是：

include <filename>

filename可以是当前操作系统Shell的文件模式（可以包含路径和通配符）

GNU的make工作时的执行步骤如下：

- 1、读入所有的Makefile。
- 2、读入被include的其它Makefile。
- 3、初始化文件中的变量。
- 4、推导隐晦规则，并分析所有规则。
- 5、为所有的目标文件创建依赖关系链。
- 6、根据依赖关系，决定哪些目标要重新生成。
- 7、执行生成命令。

文件搜索

可以通过**定义特殊变量“VPATH”**，make就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）

VPATH = src : ./header

另一个设置文件搜索路径的方法是使用make的“vpath”关键字（注意，它是全小写的），这不是变量，这是一个make的关键字，这和上面提到的那个VPATH变量很类似，但是它更为灵活。它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

- 1、**vpath <pattern> <directories>**
为符合模式<pattern>的文件指定搜索目录<directories>。
- 2、**vpath <pattern>**
清除符合模式<pattern>的文件的搜索目录。
- 3、**vpath**
清除所有已被设置好了的文件搜索目录。

vpath使用方法中的<pattern>需要包含“%”字符。“%”的意思是匹配零或若干字符，例如，“%.h”表示所有以“.h”结尾的文件。<pattern>指定了要搜索的文件集，而<directories>则指定了<pattern>的文件集的搜索的目录。

Eg. **vpath %.h ./headers**

该语句表示，要求make在“./headers”目录下搜索所有以“.h”结尾的文件。（如果某文件在当前目录没有找到的话）

静态模式

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活。语法：

```
<targets ...>: <target-pattern>: <prereq-patterns ...>
  <commands>
  ...
  ...
```

targets定义了一系列的目标文件，可以有通配符。是目标的一个集合。

target-pattern是指明了targets的模式，也就是的目标集模式，是从targets中取出的目标集。

prereq-patterns是目标的依赖模式，它对target-pattern形成的模式再进行一次依赖目标的定义。即对targets中取出的目标集更改%后内容。

Eg.

```
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c
  $(CC) -c $(CFLAGS) $< -o $@
```

上面的例子中，指明了我们的目标从\$object中获取，“%.o”表明要所有以“.o”结尾的目标，也就是“foo.o bar.o”，也就是变量\$object集合的模式，而依赖模式“%.c”则取模式“%.o”的“%”，也就是“foo bar”，并为其加下“.c”的后缀，于是，我们的依赖目标就是“foo.c bar.c”。而命令中的“\$<”和“\$@”则是自动化变量，“\$<”表示所有的依赖目标集（也就是“foo.c bar.c”），“\$@”表示目标集（也就是“foo.o bar.o”）。

书写命令

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被make显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

@echo 正在编译XXX模块.....

当make执行时，会输出“正在编译XXX模块.....”字串，但不会输出命令，如果没有“@”，那么，make将输出：

echo 正在编译XXX模块.....

正在编译XXX模块.....

make执行时，带入make参数“-n”或“--just-print”，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

如果要让上一条命令的结果应用在下一条命令时，你应该使用**分号**分隔这两条命令。比如你的第一条命令是cd命令，你希望第二条命令得在cd之后的基础上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。

每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会**终止执行当前规则**，这将有可能终止所有规则的执行。

但有些时候，命令的出错并不表示就是错误的。例如mkdir命令，我们一定需要建立一个目录，如果目录不存在，那么mkdir就成功执行，万事大吉，如果目录存在，那么就出错了。我们之所以使用mkdir的意思就是一定要有这样的一个目录，于是我们就不希望mkdir出错而终止规则的运行。为了做到这一点，忽略命令的出错，我们可以在Makefile的命令行前加一个减号“-”（在Tab键之后），标记为**不管命令出不出错都认为是成功的**。

嵌套执行make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的Makefile，这有利于让我们的Makefile变得更加地简洁

例如，我们有一个子目录叫subdir，这个目录下有个Makefile文件，来指明了这个目录下文件的编译规则。那么我们总控的Makefile可以这样书写：

```
subsystem:  
cd subdir && $(MAKE)
```

我们把这个Makefile叫做“总控Makefile”，总控Makefile的变量可以传递到下级的Makefile中（如果你显示的声明），但是不会覆盖下层的Makefile中所定义的变量，除非指定了“-e”参数。

如果你要传递变量到下级Makefile中，那么你可以使用这样的声明：

```
export <variable ...> 如果你要传递所有的变量，那么，只要一个export就行了。后面什么也不用跟。
```

如果你不想让某些变量传递到下级Makefile中，那么你可以这样声明：

```
unexport <variable ...>
```

定义命令包

如果Makefile中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以“define”开始，以“endef”结束，如：

```
define run-yacc  
yacc $(firstword $^)  
mv y.tab.c $@  
endef
```

这里，“run-yacc”是这个命令包的名字，其不要和Makefile中的变量重名。在“define”和“endef”中的两行就是命令序列。

变量值的替换。

我们可以替换变量中的共有的部分，其格式是“\$(var:a=b)”或是“\${var:a=b}”，其意思是，把变量“var”中所有以“a”字符串结尾的“a”替换成“b”字符串。这里的“结尾”意思是“空格”或是“结束符”。

```
foo := a.o b.o c.o  
bar := $(foo:.o=.c)
```

这个示例中，我们先定义了一个“\$(foo)”变量，而第二行的意思是把“\$(foo)”中所有以“.o”字符串结尾全部替换成“.c”，所以我们的“\$(bar)”的值就是“a.c.b.c.c.c”。

我们可以使用“+ =”操作符给变量追加值，如：

```
objects = main.o foo.o bar.o utils.o  
objects += another.o
```

我们的\$(objects)值变成：“main.o foo.o bar.o utils.o another.o”（another.o被追加进去了）

前面我们所讲的在Makefile中定义的变量都是“全局变量”，在整个文件，我们都可以访问这些变量。

同样可以为某个目标设置局部变量，它可以和“全局变量”同名，它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

```
<target ...> : <variable-assignment>
```

```
<target ...> : override <variable-assignment>
```

设置了这样一个变量，这个变量会作用到由这个目标所引发的所有规则中去

```
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”

模式变量

局部变量可以定义在某个目标上，模式变量的好处就是，我们可以给定一种“模式”，可以把变量定义在符合这种模式的所有目标上。可以以如下方式给所有以`.o`结尾的目标定义目标变量：

```
% .o : CFLAGS = -O
```

条件判断

使用条件判断，可以让make根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。

```
ifeq $(CC),gcc
libs=$(libs_for_gcc)
else
libs=$(normal_libs)
endif

foo: $(objects)
$(CC) -o foo $(objects) $(libs)

ifneq ifeq
ifdef ifndef 测试一个变量是否有值，其并不会把变量扩展到当前位置
```

使用函数

函数调用，很像变量的使用，也是以`$`来标识的，其语法如下：

```
$(<function> <arguments>)
<function>就是函数名，make支持的函数不多。<arguments>是函数的参数，参数间以逗号，“ 分隔，而函数名和参数之间以“空格”分隔。
```

```
comma:=,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
```

`$(comma)`的值是一个逗号。`$(space)`使用了`$(empty)`定义了一个空格，`$(foo)`的值是“`a b c`”，`$(bar)`的定义用，调用了函数“`subst`”，这是一个替换函数，这个函数有三个参数，第一个参数是被替换字符串，第二个参数是替换字符串，第三个参数是替换操作作用的字符串。这个函数也就是把`$(foo)`中的空格替换成逗号，所以`$(bar)`的值是“`a,b,c`”。

字符串函数

<code>\$subst <from>,<to>,<text></code>	把字符串 <code><text></code> 中的 <code><from></code> 字符串替换成 <code><to></code> 并返回被替换成后的字符串
<code>\$patsubst <pattern>,<replacement>,<text></code>	查找 <code><text></code> 中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式 <code><pattern></code> ，如果匹配的话，则以 <code><replacement></code> 替换。返回被替换成后的字符串
<code>\$strip <string></code>	去掉 <code><string></code> 字符串中开头和结尾的空字符。
<code>\$findstring <find>,<in></code>	在字符串 <code><in></code> 中查找 <code><find></code> 字符串。如果找到，那么返回 <code><find></code> ，否则返回空字符串。
<code>\$word <n>,<text></code>	取字符串 <code><text></code> 中第 <code><n></code> 个单词。（从一开始）
<code>\$sort <list></code>	给字符串 <code><list></code> 中的单词排序（升序）。

`$foreach <var>,<list>,<text>` 把参数`<list>`中的单词逐一取出放到参数`<var>`所指定的变量中，然后再执行`<text>`所包含的表达式。每一次`<text>`会返回一个字符串，循环过程中，`<text>`所返回的每个字符串会以空格分隔，最后当整个循环结束时，`<text>`所返回的每个字符串所组成的一个字符串（以空格分隔）将会是`foreach`函数的返回值。所以，`<var>`最好是一个变量名，`<list>`可以是一个表达式，而`<text>`中一般会使用`<var>`这个参数来依次枚举`<list>`中的单词。

```
names := a b c d
files := $(foreach n,$(names),$(n).o)
```

`$(name)`中的单词会被挨个取出，并存到变量`n`中，“`$(n).o`”每次根据`$(n)`计算出一个值，这些值以空格分隔，最后作为`foreach`函数的返回，所以，`$(files)`的值是“`a.o b.o c.o d.o`”。注意，`foreach`中的`<var>`参数是一个临时的局部变量，`foreach`函数执行完后，参数`<var>`的变量将不在作用，其作用域只在`foreach`函数当中。

常用的预定义变量：

- `$*` 不包含扩展名的目标文件名称。
- `$+` 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件。
- `$<` 第一个依赖文件的名称。
- `$?` 所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚。
- `$@` 目前规则中所有的目标的集合。
- `$^` 所有的依赖文件，以空格分开，不包含重复的依赖文件。

\$% 如果目标是归档成员，则该变量表示目标的归档成员名称。

Cmake

2023年3月7日 13:10

cmake跨平台生成makefile，再由make/ninja等编译工具根据makefile的规范进行编译

Cmake根据 CMakeLists.txt 文件执行 **指令是大小写无关的**

一个最简单的Cmake项目形如

```
cmake_minimum_required(VERSION 3.15)

# set the project name
project(Tutorial)

# add the executable
add_executable(Tutorial tutorial.cpp)
```

cmake_minimum_required 指定使用 CMake 的最低版本号，**project** 指定项目名称，**add_executable** 用来生成可执行文件，需要**指定生成可执行文件的名称和相关源文件(如果有多个，那么就用空格隔开)**。

构建 **cmake + CMakeLists.txt文件存放的目录** 如在当前文件夹底下 就直接 **cmake .**
编译和链接 **cmake --build + 生成的文件希望存放的目录**

变量

```
set(SRC_LIST a.cpp b.cpp c.cpp) 通过set命令可以指定变量 类似于makefile的变量
使用时也类似 ${SRC_LIST}

list 命令 list(APPEND EXTRA_LIBS MathFunctions)
APPEND表示将元素MathFunctions追加到列表EXTRA_LIBS中
可以通过SET指令重新定义 EXECUTABLE_OUTPUT_PATH 和
LIBRARY_OUTPUT_PATH 变量来指定最终的目标二进制的位置
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
SET(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)
```

输出信息

```
MESSAGE([SEND_ERROR | STATUS | FATAL_ERROR] "message")
```

向终端输出用户定义的信息，包含三种类型：

```
SEND_ERROR #产生错误，生成过程被跳过。
STATUS #输出前缀为--d的信息。
FATAL_ERROR #立即终止所有的cmake过程。
```

添加搜索目录

```
target_include_directories(${PROJECT_NAME} PUBLIC
                           ${PROJECT_BINARY_DIR}
                           )
```

target_include_directories 命令可以将需要添加的头文件目录加入搜索路径

但是如果源文件很多，把所有源文件的名字都加进add_executable去将是一件烦人的工作。更省事的方法是使用 **aux_source_directory** 命令
aux_source_directory(<dir> <variable>)
该命令会查找指定目录下的所有源文件，然后将结果存进指定变量名

添加子目录

使用命令 **add_subdirectory** 指明本项目包含一个子目录，这样子目录下的 CMakeLists.txt 文件和源代码也会被处理。

ADD_SUBDIRECTORY(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
这个指令用于向当前工程添加存放源文件的子目，并可以指定中间二进制和目标二进制存放的位置。EXCLUDE_FROM_ALL参数的含义是将这个目录从编译过程中排除。

使用链接库

**ADD_LIBRARY(libname [SHARED|STATIC|MODULE][EXCLUDE_FROM_ALL]
source1 source2 ... sourceN)**

SHARED,动态库

STATIC,静态库

MODULE，在使用dyld的系统有效，如果不支持dyld，则被当做SHARED对待。

EXCLUDE_FROM_ALL参数的意思是这个不会被默认构建，除非有其他的组件依赖或者手工构建。

使用命令 **add_library** 将 src 目录中的源文件编译为静态链接库

```
aux_source_directory(. DIR_LIB_SRCS)
add_library(MathFunctions ${DIR_LIB_SRCS})
```

使用命令 **target_link_libraries** 指明可执行文件需要连接一个链接库

```
target_link_libraries(Demo MathFunctions)
```

配置头文件

configure_file 命令用于加入一个配置头文件

```
configure_file (
    "${PROJECT_SOURCE_DIR}/config.h.in"
    "${PROJECT_BINARY_DIR}/config.h"
)
```

加入一个配置头文件 config.h，这个文件由 CMake 从 config.h.in 生成，通过这样的机制，将可以通过预定义一些参数和变量来控制代码的生成。

自定义编译选项

option 命令可以添加一个选项，并且设置默认值

```
option (USE_MYMATH
        "Use provided math implementation" ON)
# 添加了一个 USE_MYMATH 选项，并且默认值为 ON
```

config.h.in 文件

使用 **#cmakedefine USE_MYMATH**

这样 CMake 会自动根据 CMakeLists 配置文件中的设置自动生成 config.h 文件

添加库的使用要求

INTERFACE是指消费者需要、但生产者不需要的那些东西

使用`INTERFACE`可以指定使用要求 使得任何使用该库的文件自动包含该路径

```
target_include_directories(MathFunctions
    INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
)
```

除了 `INTERFACE`，还有`PRIVATE` 和 `PUBLIC`。`INTERFACE`表示消费者需要生产者不需要，`PRIVATE`表示消费者不需要生产者需要，`PUBLIC` 表示消费者和生产者都需要。

如何安装

```
cmake -D CMAKE_INSTALL_PREFIX=/usr .
```

`INSTALL`指令包含了各种类型，我们需要一个个分开解释：

目标文件的安装：

```
INSTALL(TARGETS targets ...
    [[ARCHIVE|LIBRARY|RUNTIME]
     [DESTINATION <dir>]
     [PERMISSIONS permissions ...]
     [CONFIGURATIONS [Debug|Release|...]]
     [COMPONENT <component>]
     [OPTIONAL]
    ][...])
```

`TARGETS`后面跟的就是我们通过`ADD_EXECUTABLE`或者`ADD_LIBRARY`定义的目标文件，可能是可执行二进制、动态库、静态库。

目标类型：`ARCHIVE`特指静态库，`LIBRARY`特指动态库，`RUNTIME`特指可执行目标二进制。

`DESTINATION`定义了安装的路径，如果路径以/开头，那么指的是绝对路径，这时候`CMAKE_INSTALL_PREFIX`其实就无效了。如果你希望使用`CMAKE_INSTALL_PREFIX`来定义安装路径，就要写成相对路径，不要以/开头，那么安装后的路径就是
\${CMAKE_INSTALL_PREFIX}/<DESTINATION定义的路径>

```
INSTALL(TARGETS myrun mylib mystaticlib
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION libstatic
    )
```

二进制myrun安装到\${CMAKE_INSTALL_PREFIX}/bin目录

动态库lib mylib安装 \${CMAKE_INSTALL_PREFIX}/lib目录

静态库lib mystaticlib安装到 \${CMAKE_INSTALL_PREFIX} / libstatic目录。

为工程添加测试

添加测试同样很简单。CMake 提供了一个称为 CTest 的测试工具。我们要做的只是在项目根目录的 CMakeLists 文件中调用一系列的 `add_test` 命令。

```

# 启用测试
enable_testing()

# 测试程序是否成功运行
add_test (test_run Demo 5 2)

# 测试帮助信息是否可以正常提示
add_test (test_usage Demo)
set_tests_properties (test_usage
    PROPERTIES PASS_REGULAR_EXPRESSION "Usage: .* base exponent")

# 测试 5 的平方
add_test (test_5_2 Demo 5 2)

set_tests_properties (test_5_2
    PROPERTIES PASS_REGULAR_EXPRESSION "is 25")

# 测试 10 的 5 次方
add_test (test_10_5 Demo 10 5)

set_tests_properties (test_10_5
    PROPERTIES PASS_REGULAR_EXPRESSION "is 100000")

# 测试 2 的 10 次方
add_test (test_2_10 Demo 2 10)

set_tests_properties (test_2_10
    PROPERTIES PASS_REGULAR_EXPRESSION "is 1024")

```

test_run 用来测试程序是否成功运行并返回 0 值

PASS_REGULAR_EXPRESSION 用来测试输出是否包含后面跟着的字符串

可以定义一个宏来简化测试

```

# 定义一个宏，用来简化测试工作
macro (do_test arg1 arg2 result)
    add_test (test_${arg1}_${arg2} Demo ${arg1} ${arg2})
    set_tests_properties (test_${arg1}_${arg2}
        PROPERTIES PASS_REGULAR_EXPRESSION ${result})
endmacro (do_test)

# 使用该宏进行一系列的数据测试
do_test (5 2 "is 25")
do_test (10 5 "is 100000")
do_test (2 10 "is 1024")

```

让 CMake 支持 gdb 的设置也很容易，只需要指定 Debug 模式下开启 -g 选项：

```
set(CMAKE_BUILD_TYPE "Debug")
set(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")
set(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")
```

生成安装包

CPack，它同样也是由 CMake 提供的一个工具，专门用于打包。首先在顶层的 CMakeLists.txt 文件尾部添加下面几行：

```
# 构建一个 CPack 安装包
include (InstallRequiredSystemLibraries)
set (CPACK_RESOURCE_FILE_LICENSE
    "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
set (CPACK_PACKAGE_VERSION_MAJOR "${Demo_VERSION_MAJOR}")
set (CPACK_PACKAGE_VERSION_MINOR "${Demo_VERSION_MINOR}")
include (CPack)
```

在bash下调用

生成二进制安装包： cpack -C CPackConfig.cmake

生成源码安装包： cpack -C CPackSourceConfig.cmake