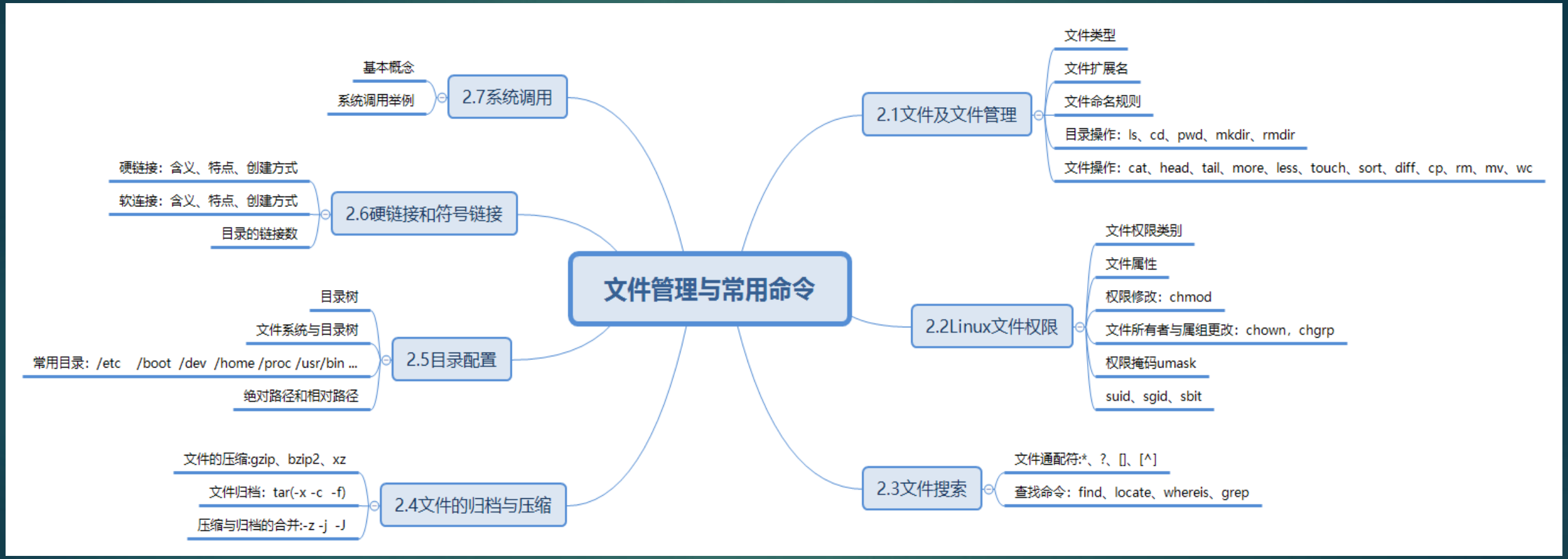

文件管理与常用命令
2.1文件及文件管理
文件类型
文件扩展名
文件命名规则
目录操作：ls、cd、pwd、mkdir、rmdir
文件操作：cat、head、tail、more、less、touch、sort、diff、cp、rm、mv、wc
2.2Linux文件权限
文件权限类别
文件属性
权限修改：chmod
文件所有者与属组更改：chown，chgrp
权限掩码umask
suid、sgid、sbit
2.3文件搜索
文件通配符:*、?、[]、[^]
查找命令：find、locate、whereis、grep
2.4文件的归档与压缩
文件的压缩:gzip、bzip2、xz
文件归档：tar(-x -c -f)
压缩与归档的合并:-z -j -J
2.5目录配置
目录树
文件系统与目录树
常用目录：/etc /boot /dev /home /proc /usr/bin ...
绝对路径和相对路径
2.6硬链接和符号链接
硬链接：含义、特点、创建方式
软连接：含义、特点、创建方式
目录的链接数
2.7系统调用
基本概念
系统调用举例

第3章 vi编辑器及Linux C开发工具

第3章 vi编辑器及Linux C开发工具

- 3.1 vi编辑器的使用
- 3.2 编译器gcc的使用
- 3.3 静态与动态链接库
- 3.4 分布式版本控制系统git
- 3.4 调试器GDB的使用
- 3.5 工程管理器make的使用

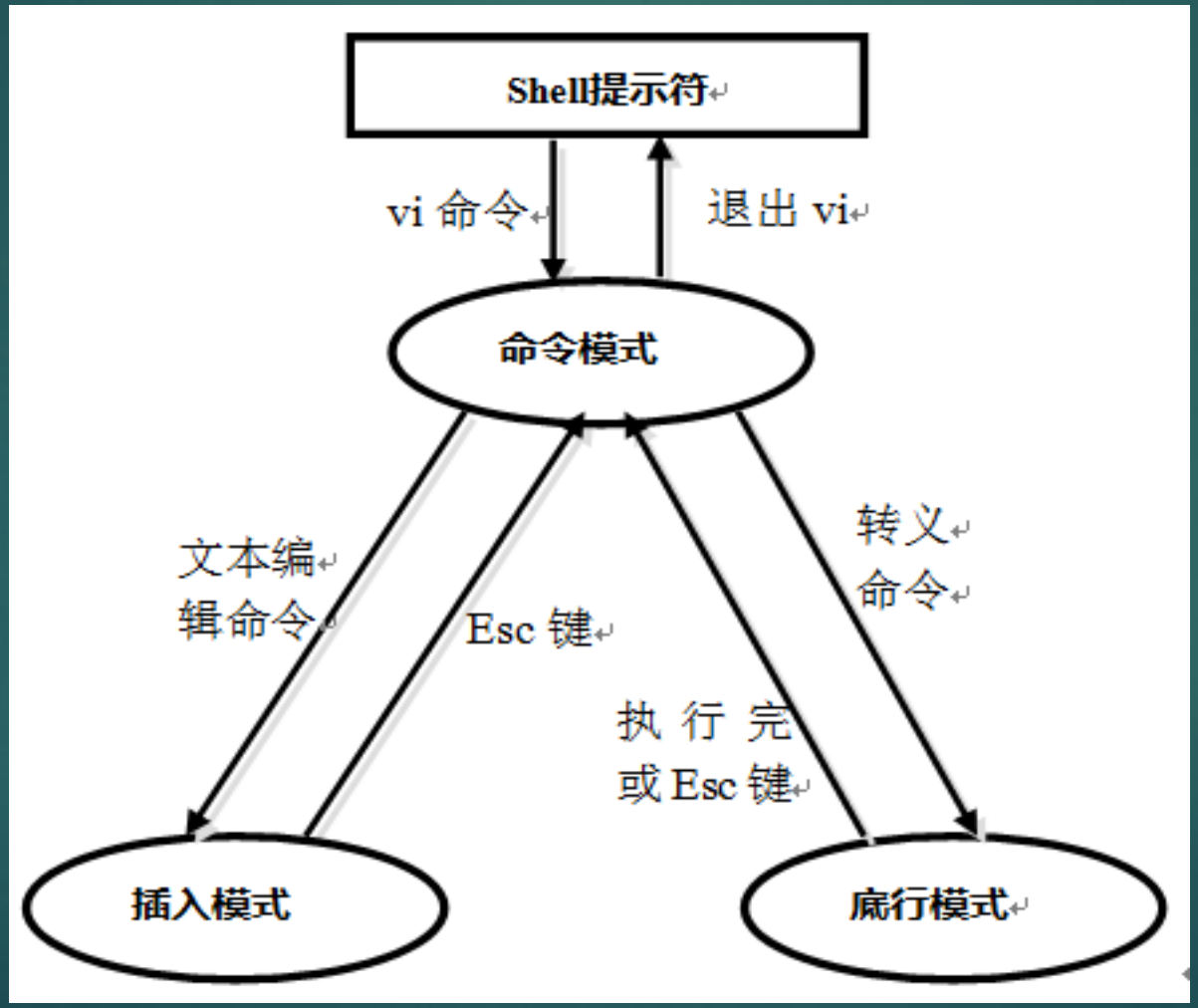
3.1 vi编辑器的使用

Vi编辑器介绍

- vi编辑器是Linux和Unix上最基本的文本编辑器，可以执行输出、删除、查找、替换、块操作等众多文本操作，工作在字符模式下。由于不需要图形界面，vi是效率很高的文本编辑器。尽管在Linux上也有很多图形界面的编辑器可用，但vi在系统和服务器管理中的功能是那些图形编辑器所无法比拟的。
- vim是vi的升级版本，能够用不同颜色或底行显示一些特殊的信息，支持多种程序的语法检测。
- 优势：
  1. 所有unix like系统都内建vi编辑器，而其它编辑器则不一定会存在
  2. 很多软件的编辑接口都会自动呼叫vi编辑器（visudo、crontab等）
  3. vim具有程序编辑能力，可以主动以字体颜色辨别语法的正确性
  4. 程序简单，所以编辑速度很快

Vi编辑器工作模式

- vim有3种基本工作模式：命令模式、插入模式和底行模式。



Vi编辑器工作模式

- 1．命令模式

键入“vi 文件名”就可启动vi。“命令模式”是进入vi编辑器的初始模式。

从键盘上输入的任何字符都被作为编辑命令来解释，例如，输入“a”表示附加命令，输入“i”表示插入命令，输入“dd”表示删除光标所在的行等。如果输入的字符不是vi合法的命令，则系统会发出“报警声”。

在“命令模式”下，输入的vi文本编辑命令并不在屏幕上显示。

Vi编辑器工作模式

- 命令模式下常用操作1

命令	功能
n<space>	n表示数字，按下数字n后再按空格，光标会向右移动n个字符。
n<enter>	N为数字，光标乡下移动n行
0或Home键	**移动到这一行的最前面字符处**
$或End键	**移动到这一行的最后面的字符处**
H	光标移动到这个屏幕的最上方那一行的第一个字符
M	光标移动到这个屏幕的中央那一行的第一个字符
L	光标移动到这个屏幕的最下方那一行的第一个字符
G	**移动到该文件的最后一行**
nG	**n为数字，移动到该文件的第n行。例如20G**
gg	移动到当前文件的最后一行，相当于1G

Vi编辑器工作模式

- 命令模式下常用操作2

命令	功能
x,X	**在当前行中，x为向后删除一个字符（相当于[del]按键)，X为向前删除一个字（BackSpace）**
nx	n为数字，连续向后删除n 个字符。例如：10x，连续删光标后10个字符
dd	**删除游标所在的那一整行**
ndd	**n为数字。删除光标所在的向下n行，例如20dd则是删除20行**
d1G	删除光标所在到第一行的所有数据
dG	删除光标所在到最后一行的所有数据
d$	删除光标所在处，到该行的最后一个字符
d0	删除光标所在处，到该行的最前面一个字符

Vi编辑器工作模式

- 命令模式下常用操作3

命令	功能
yy	**复制光标所在的那行**
nyy	**n为数字。复制光标所在的向下n行，例如20yy 则是复制20行**
y1G	复制光标所在行到第一行的所有数据
yG	复制光标所在行到最后一行的所有数据
y0	复制光标所在的那个字符到该行行首的所有数据
y$	复制光标所在的那个字符到该行行首的所有数据
p,P	**p为将已复制的数据在光标下一行贴上，P 则为贴在游标上一行!**
J	将光标所在行与下一行的数据结合成同一行
u	**复原前一个动作。**

Vi编辑器工作模式

- 2．插入模式

“插入模式”是vi编辑器最简单的模式，在该模式下没有繁琐的命令，用户从键盘输入得任何有效字符都被认为是正在编辑的文件内容，并显示在vi编辑器中。只有在插入模式下，才可以进行文本的输入操作。

从“插入模式”切换至“命令模式”，可以通过按Esc键完成。

Vi编辑器工作模式

- 命令模式进入插入模式命令

命令	功能
i，I	**进入插入模式(Insert mode): i为从目前光标所在处插入， I 为在目前所在行的第一个非空格符处开始插入。**
a,A	**进入插入模式(Insert mode):a为从目前光标所在的下一个字符处开始插入，A为从光标所在行的最后一个字符处开始插入。**
o,O	**进入插入模式(Insert mode):**这是英文字母o的大小写。o为「目前光标所在的下一行处插入新的一行』;0为在目前光标所在处的上一行插入新的一行!
r,R	进入取代模式(Replace mode): r只会取代光标所在的那一个字符一次;R会一直取代光标所在的文字,直到按下ESC为止;

Vi编辑器工作模式

- 3．底行模式

“底行模式”是指可以在编辑器最底部的一行输入控制操作命令，主要用来进行一些文字编辑的辅助功能，比如字符串搜索、替代、保存文件，退出vi等。

在“命令模式”下，通过输入冒号（：）、问号（？）或斜杠（/），可以切换至“底行模式”。

Vi编辑器工作模式

- 底行模式常用指令

命令	功能
:w	**将编辑的数据写入硬盘文件中(常用)**
:w!	**若文件属性为『只读』时，强制写入该文件。不过，到底能不能写入，还是跟你对该文件的权限有关.**
:q	**离开vi(常用)**
:q!	**若曾修改过文件，又不想储存，使用!为强制离开不储存文件。**
:wq	**储存后离开，若为:wq!则为强制储存后离开（常用)**
:ZZ	**若文件没有更动，则不储存离开，若文件已经被更动过，则储存后离开!**
:w filename	将编辑的数据储存成另一个文件（另存为）
:r filename	在编辑的数据中，读入另一个文件的数据。将「filename」这个文件内容加到游标所在行后面
:n1,n2 w filename	将n1到n2的内容储存成fi lename这个文件。

Vi编辑器工作模式

- 底行模式常用指令

命令	功能
/word	**向光标之下寻找一个名称为 word 的字符串。例如要在文件内搜寻vbird这个字符串，就输入/vbird即可!（常用)**
?word	**向光标之上寻找一个字符串名称为 word 的字符串。**
:n1 , n2s/word1/word2/g	**为word2 !举例来说，在100到200 行之间搜寻 vbird 并取代为 VBIRD则: 100,200s /vbird/VBIRD/gl。(常用)**
:1,$s/word1/word2/g	**从第一行到最后一行寻找wordl字符串，并将该字符串取代为word2 ! (常用)**
:1,$s/word1/word2/gc	**从第一行到最后一行寻找wordl字符串，并将该字符串取代为word2 !且在取代前显示提示字符给用户确认。**
set nu	显示行号，设定之后，会在每一行的前缀显示该行的行号
set nonu	与set nu相反，为取消行号!

Vi编辑器工作模式


排序:(例: 1-5行排序:光标放在1行上,然后!5G,然后sort)
字体设定:(set guifont=Raize\ Bold\ 13)
键绑定:(如: map h <Insert>;map <F2> h{ } 按下F2后进入Insert模式并打印{ })
Esc 命令模式
~ 转换大小写
` 跳跃到标记处
! 外部过滤器
@ 执行寄存器
# 反向查找
$ 行末
% 括号匹配
^ 行首
& :s//~/
*
(句首
) 下一句首
_
- 前一行行首
+ 次行行首
= 自动格式化
Q 切换至ex模式
q
W 下一单词
w 下一单词
E 单词尾
e 单词尾
R 替换模式
r 替换字符
T
t
Y 复制行
y 复制
U
u 撤销
I 到行首插入
i 插入模式
O 分段(前)
o 分段(后)
P 粘贴(前)
p 粘贴(后)
{ 段首
[
} 段尾
]
A 在行末附加
a 附加
S 删除行并插入
s 删除字符并插入
D 删除至行末
d 删除
F 行内字符反向查找
f 行内字符查找
G 文件尾/行号
g 附加命令
H 画面顶
h
J 合并两行
j
K
k
L 画面底
l
: ex命令
; 重复t/T/f/F
"
' 跳转到位置标记
| 行首/列
\
Z 退出
z 附加命令
X 退格
x 删除字符
C 修改至行末
c 修改
V visual行模式
v visual模式
B
b 前一单词
N 查找上一除
n 查找下一除
M
m 设定标记
< 反缩进
, 反向t/T/f/F
> 缩进
. 重复指令
? 向前查找
/ 向后查找
动作 移动光标或欲定义操作的范围
指令 直接执行的指令
红色指令进入编辑模式
操作 后接用以表示操作范围的指令
extra 特殊功能需额外输入
主要ex指令: :w (保存)
:q (退出)
:e (打开文件) <Tab>
:%s/x/y/g (以y全文替换x)
:h (帮助)
其它重要指令:
Ctrl-R: 重做
Ctrl-F/-B: 向前(下)翻页/向后(上)翻页
Ctrl-E/-Y: 向前(下)一列/向后(上)一列
Ctrl-V: 切换visual模式
(1) 在复制/粘贴/删除 指令前使用 "x (x=a...z)
使用指令的寄存器(如: "ay$ 复制该行目前位置到行尾的内容到寄存器'a')
(2) 命令前添加数字
重复指定次数的操作
(如: 2p, d2w, 5i, d4j)
(3) 重复光标所在字符处指定的查找
(dd = 删除本行, >>= 行首缩进)
(4) ZZ 保存并退出
(5) zt: 移动游标所在行至畫面頂端,
zb: 底端, zz: 中央
(6) gg: 文件首;ddp/P: 交换上下两行
daw: 删除单词;全选: ggVG
w,e,b指令
b(小写): quux((foo, bar, baz);
B(大写): quux(foo, bar, baz);
(数字)n + >> :行共同缩进(如:2>>表示这行到下一行共同缩进)

vi的暂存文件

vi编辑文件时，会在编辑文件所在的文件夹下生成.filename.swap。 记录了对文件的操作。若vi编辑器非正常退出、掉线，则可用该文件进行救援。

下次编辑该文件将会出现若干提示：

Oopen Read-only：以只读方式打开，可以多人同时查看

Edit anyway：正常方式打开文件，但不会载入暂存文件，可能出现两个使用者互相改变对方的文件问题。

Recover：加载暂存文件，将之前未存储内容找回。但要手动删除swap文件

Delete it：确定暂存文件误用可以直接删除。

Quit：离开vim

Abort:和quit类似，回到命令行

Vi编辑器扩展功能

1. 区块选择

v：字符选择，会将光标经过的地方反白选择

V：行选择，会将光标经过的行反白选择

Ctr+v：区块选择，可以用长方形的方式选择文件内容

Vi编辑器扩展功能

2. 多文本编辑

vi打开多个文件，:n 编辑下一个，：N编辑上一个，：files列出打开的所有文件

命令	功能
方向键	不同的方向键会使光标向相应位置移动一个字符
:args	显示多文件信息
:next或:n	向后切换文件
:prev或:N	向前切换文件
:first	定位首文件
:last	定位尾文件
Ctrl + ^	快速切换到编辑器中切换前的文件
:files	列出目前vi编辑器打开的所有文件
: r filename	将filename文件内容读入当前打开的文件光标所处的位置

Vi编辑器扩展功能

3. 多窗口功能

: sp {filename}　　ctrl+w+（上/下）

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:6:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
/etc/passwd [+][只读]
#DEFINE         compressor      gzip -c7
#-----------------------------------------------------------
# Misc definitions:  same as program definitions above.
#
#DEFINE         whatis_grep_flags               -i
#DEFINE         apropos_grep_flags              -iEw
#DEFINE         apropos_regex_grep_flags        -iE
#-----------------------------------------------------------
man_db.conf
```

Vi编辑器常用设置

- 使用vi编辑器会在家目录下生成~/.viminfo文件，记录访问位置、搜索记录等。

```
[liu@lesson tmp]$ more ~/.viminfo
# This viminfo file was generated by Vim 7.4.
# You may edit it if you're careful!

# Value of 'encoding' when this file was written
*encoding=utf-8


# hlsearch on (H) or off (h):
~H
# Last Search Pattern:
~MSle0~/set

# Command Line History (newest to oldest):
:q
:!ls -al
:/set
```

Vi编辑器常用设置

- 而对于不同的发型版，vi的默认设置不同，比如搜索到的词是否高亮、是否默认缩进等等。不建议修改， 可以修改~/.vimrc 文件（自建）

使用set all 可以查看所有环境的设置

命令	功能
:set autoindent	缩进每一行，使之与前一行相同
:set noautoindent	取消缩进
:set number	在编辑文件时显示行号
:set nonumber	不显示行号
:set ruler	在屏幕底端显示光标所在的行、列位置
:set noruler	不显示光标所在的行、列位置

Vi编辑器常用设置

- 配置文件/etc/vimrc

```
if v:lang =~ "utf8$" || v:lang =~ "UTF-8$"
   set fileencodings=ucs-bom,utf-8,latin1
endif

set nocompatible        " Use Vim defaults (much better!)
set bs=indent,eol,start         " allow backspacing over everything in insert
 mode
"set ai                 " always set autoindenting on
"set backup             " keep a backup file
set viminfo='20,\"50    " read/write a .viminfo file, don't store more
                        " than 50 lines of registers
set history=51          " keep 50 lines of command line history
set ruler               " show the cursor position all the time

" Only do this part when compiled with support for autocommands
if has("autocmd")
```

Vi编辑器使用练习

1. 在tmp目录下建立vitest目录并进入
2. 将/etc/man_db.conf复制到该目录下
3. Vi打开改文件
4. 设定显示行号
5. 移动到底50行，向后移动20个字符
6. 移动到第一行，并向下搜学gizp字符串
7. 将29到41行之间的小写man改为大写MAN
8. 修改完后，撤回刚才的所有操作
9. 复制第66行到71行的内容到最后一行
10. 113到128行中去掉注释符号“#”
11. 文件另存为man.test.config
12. 保存文件并离开

- vi 总结

Linux底下的配置文件多为文本文件，故使用 vim 即可进行设定编辑;

vim可视为程序编辑器，可用以编辑shell script，配置文件等，避免打错字;

vi为所有unix like 的操作系统都会存在的编辑器，且执行速度快速;

vi有三种模式，命令模式可变换到插入与底行模式，但插入模式与底行模式不能互换;

常用的按键有i,[Esc],:wq等;

vi的画面大略可分为两部份，(1)上半部的本文与(2)最后一行的状态+底行模式;

数字是有意义的，用来说明重复进行几次动作的意思，如5yy为复制5行之意;

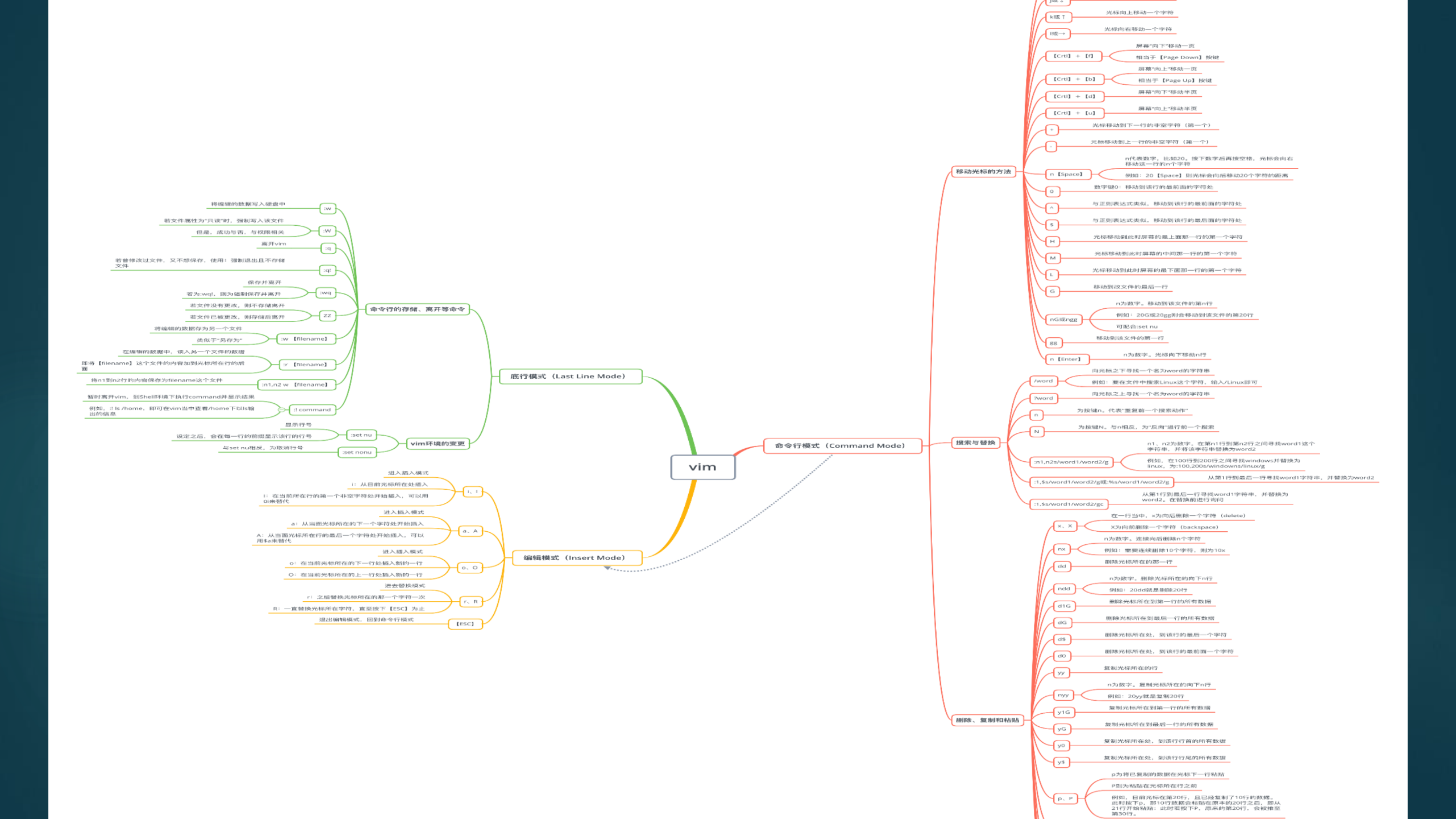
光标的移动中，大写的G经常使用，尤其是1G,G移动到文章的头/尾功能!vi的取代功能也很好! :n1,n2s/old/new/g ;

进入编辑模式几乎只要记住: i, o,R三个按钮即可!尤其是新增一行的o与取代的R

vim会主动的建立 swap 暂存档，所以不要随意断线!如果在文章内有对齐的区块，可以使用[ctrl]-v进行复制/贴上/删除的行为。

使用:sp功能可以分区窗口

vim 的环境设定可以写入在~/.vimrc文件中;


vim
底行模式（Last Line Mode）
命令行的存储、离开等命令
:w
将编辑的数据写入硬盘中
:W
若文件属性为"只读"时，强制写入该文件
但是，成功与否，与权限相关
:q
离开vim
:q!
若曾修改过文件，又不想保存，使用！强制退出且不存储文件
:wq
保存并离开
ZZ
若文件没有更改，则不存储离开
若文件已被更改，则存储后离开
:w【filename】
将编辑的数据存为另一个文件
类似于"另存为"
:r【filename】
在编辑的数据中，读入另一个文件的数据
即将【filename】这个文件的内容加到光标所在行的后面
:n1,n2 w【filename】
将n1到n2行的内容保存为filename这个文件
:! command
暂时离开vim，到Shell环境下执行command并显示结果
例如，:! ls /home，即可在vim当中查看/home下以ls输出的信息
vim环境的变更
:set nu
显示行号
设定之后，会在每一行的前缀显示该行的行号
:set nonu
与set nu相反，为取消行号
编辑模式（Insert Mode）
i、I
进入插入模式
i：从目前光标所在处插入
I：在当前所在行的第一个非空字符处开始插入，可以用0i来替代
a、A
进入插入模式
a：从当面光标所在的下一个字符处开始插入
A：从当面光标所在行的最后一个字符处开始插入，可以用$a来替代
o、O
进入插入模式
o：在当前光标所在的下一行处插入新的一行
O：在当前光标所在的上一行处插入新的一行
r、R
进去替换模式
r：之后替换光标所在的那一个字符一次
R：一直替换光标所在字符，直至按下【ESC】为止
【ESC】
退出编辑模式，回到命令行模式
命令行模式（Command Mode）
移动光标的方法
j或↓
光标向下移动一个字符
k或↑
光标向上移动一个字符
l或→
光标向右移动一个字符
【Ctrl】+【f】
屏幕"向下"移动一页
相当于【Page Down】按键
【Ctrl】+【b】
屏幕"向上"移动一页
相当于【Page Up】按键
【Ctrl】+【d】
屏幕"向下"移动半页
【Ctrl】+【u】
屏幕"向上"移动半页
+
光标移动到下一行的非空字符（第一个）
-
光标移动到上一行的非空字符（第一个）
n【Space】
n代表数字，比如20，按下数字后再按空格，光标会向右移动这一行的n个字符
例如：20【Space】则光标会向后移动20个字符的距离
0
数字键0：移动到该行的最前面的字符处
^
与正则表达式类似，移动到该行的最前面的字符处
$
与正则表达式类似，移动到该行的最后面的字符处
H
光标移动到此时屏幕的最上面那一行的第一个字符
M
光标移动到此时屏幕的中间那一行的第一个字符
L
光标移动到此时屏幕的最下面那一行的第一个字符
G
移动到改文件的最后一行
nG或ngg
n为数字。移动到该文件的第n行
例如：20G或20gg则会移动到该文件的第20行
可配合:set nu
gg
移动到该文件的第一行
n【Enter】
n为数字。光标向下移动n行
搜索与替换
/word
向光标之下寻找一个名为word的字符串
例如：要在文件中搜索Linux这个字符，输入/Linux即可
?word
向光标之上寻找一个名为word的字符串
n
为按键n，代表"重复前一个搜索动作"
N
为按键N，与n相反，为"反向"进行前一个搜索
:n1,n2s/word1/word2/g
n1、n2为数字，在第n1行到第n2行之间寻找word1这个字符串，并将该字符串替换为word2
例如，在100行到200行之间寻找windows并替换为linux，为:100,200s/windows/linux/g
:1,$s/word1/word2/g或:%s/word1/word2/g
从第1行到最后一行寻找word1字符串，并替换为word2
:1,$s/word1/word2/gc
从第1行到最后一行寻找word1字符串，并替换为word2，在替换前进行询问
删除、复制和粘贴
x、X
在一行当中，x为向后删除一个字符（delete）
X为向前删除一个字符（backspace）
nx
n为数字，连续向后删除n个字符
例如：需要连续删除10个字符，则为10x
dd
删除光标所在的那一行
ndd
n为数字，删除光标所在的向下n行
例如：20dd就是删除20行
d1G
删除光标所在到第一行的所有数据
dG
删除光标所在到最后一行的所有数据
d$
删除光标所在处，到该行的最后一个字符
d0
删除光标所在处，到该行的最前面一个字符
yy
复制光标所在的行
nyy
n为数字。复制光标所在的向下n行
例如：20yy就是复制20行
y1G
复制光标所在到第一行的所有数据
yG
复制光标所在到最后一行的所有数据
y0
复制光标所在处，到该行行首的所有数据
y$
复制光标所在处，到该行行尾的所有数据
p、P
p为将已复制的数据在光标下一行粘贴
P则为粘贴在光标所在行之前
例如，目前光标在第20行，且已经复制了10行的数据。此时按下p，那10行数据会粘贴在原本的20行之后，即从21行开始粘贴；此时若按下P，原来的第20行，会被推至第30行。

3.2 编译工具gcc

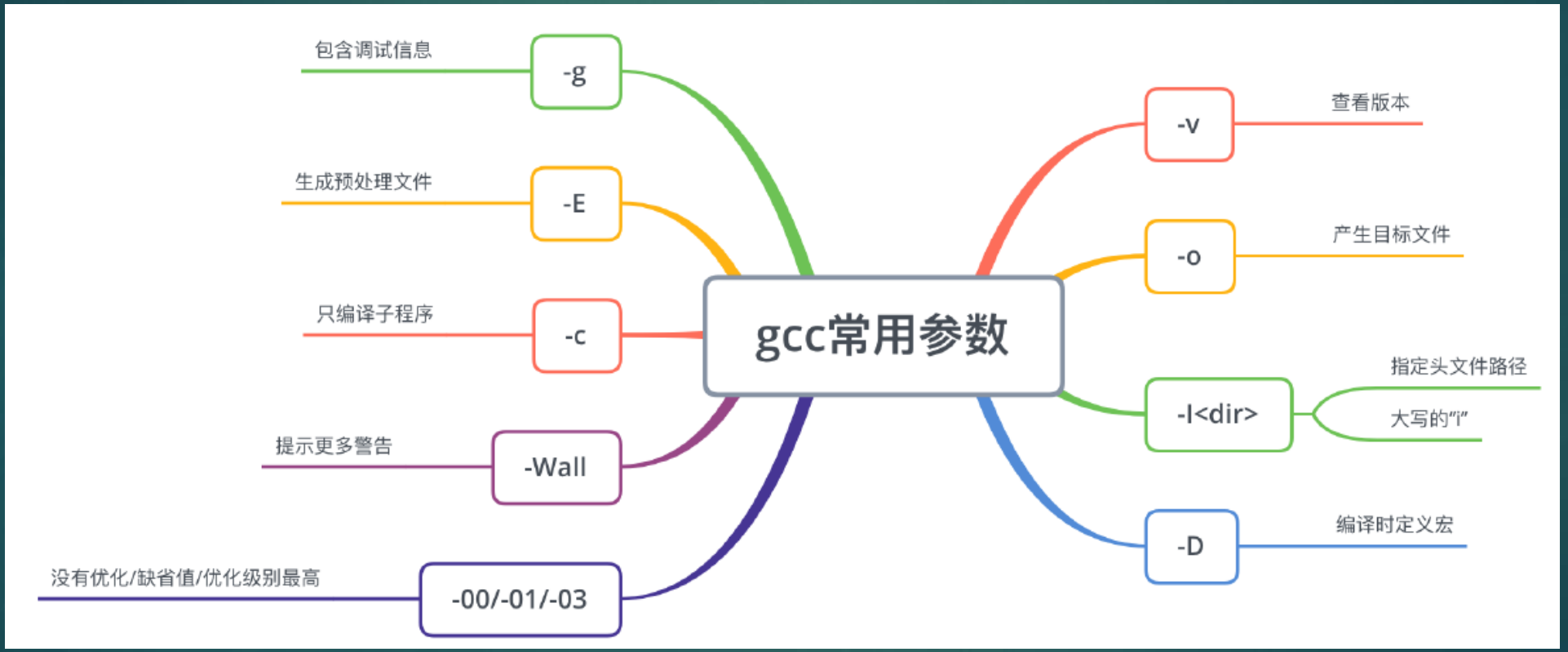
GCC概述

- GCC全称GNU Compiler Collection，GNU编译套件。
- GCC是由GNU开发的编程语言编译器，包括C、Cpp、Objective-C、Fortran、Java、Ada、Golang。

GCC基本语法

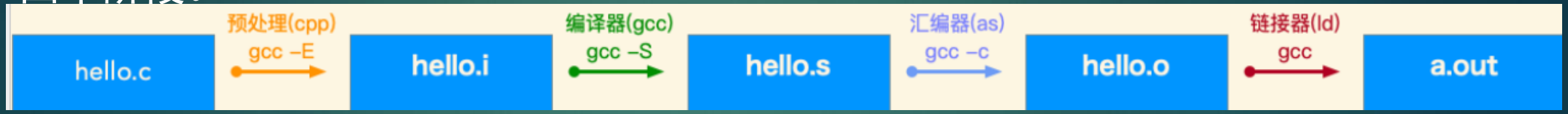
- 基本语法

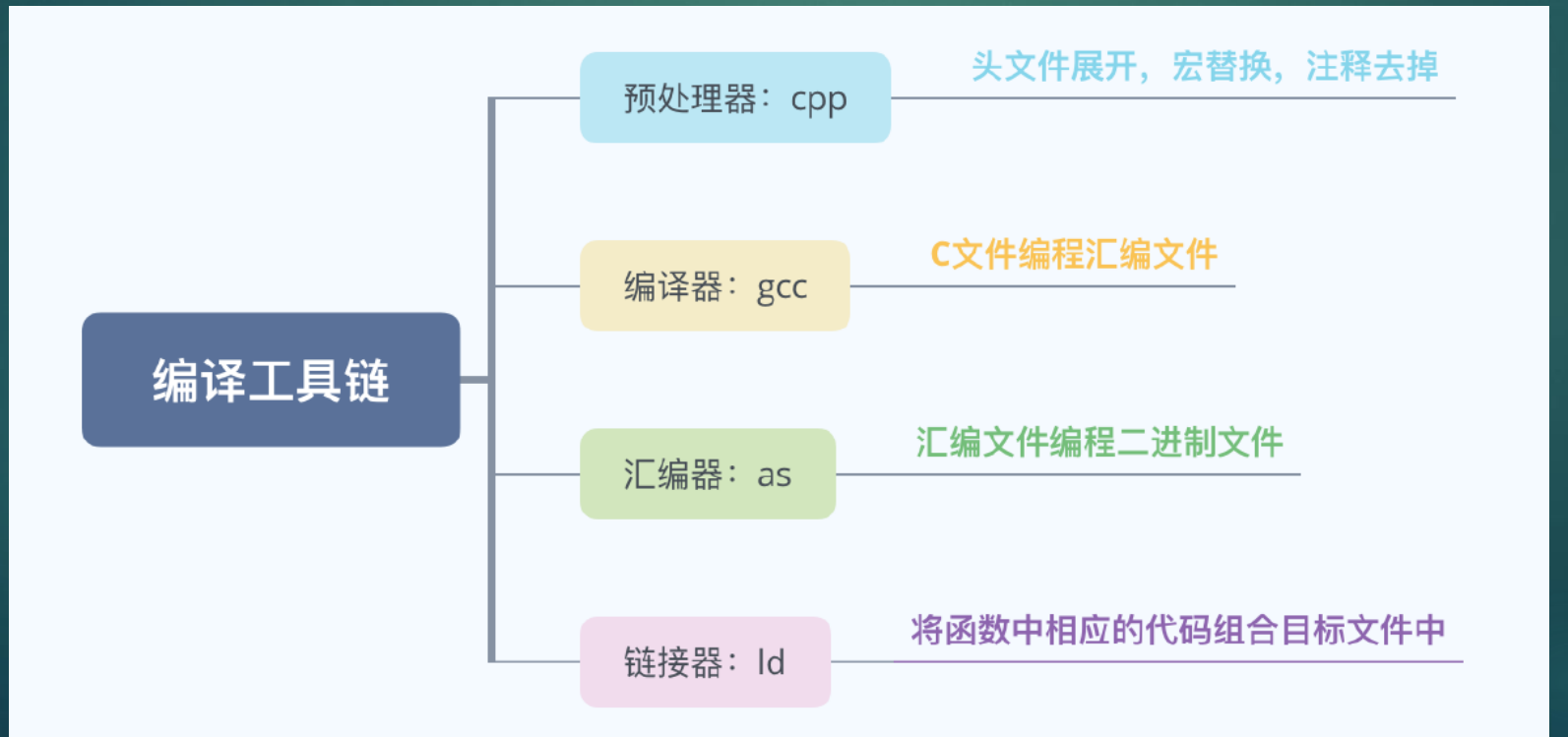
```
gcc [option | filename]
```



GCC编译流程

使用gcc进行的编译过程是一个相对复杂的过程，可分为预处理（Pre-Processing）、编译（Compiling）、汇编（Assembling）和链接（Linking）四个阶段。





预处理阶段

输入：C语言的源文件

输出：生成*.i得中间文件

功能：处理文件中的#ifdef，#include和#define等预处理命令。

格式：gcc -E -o［目标文件］［编译文件］

或 gcc -E［编译文件］ -o ［目标文件］

选项“-E”可以使编译器在预处理结束时就停止编译

选项“-o”是指定GCC输出的结果。

编译文件一般以.c为后缀名，目标文件以.i为后缀名

预处理阶段

编辑test.c文件，内容如下

```
#include <stdo.h>
int main( )
{ int a;
   scanf(“%d”,a);
   printf(“a=%d”,a);
 }
```

编译命令：gcc –E test.c –o test.i

- 提示错误：stdo.h：没有那个文件或目录
  scanf语句的错误不提示

编译阶段

输入：中间文件*.i

输出：汇编语言文件*.s

功能：此时检查语法错误。

格式：gcc -S -o [目标文件] [编译文件]

或 gcc -S [编译文件] -o [目标文件]

选项“-S”可以使编译器完成编译阶段就停止

选项“-o”是指定GCC输出的结果。

实例：gcc -S test.i -o test.s

提示：format ‘%d’ expects type ‘int *’ but argument 2 has type ‘int’

汇编阶段

汇编阶段

输入：汇编文件*.s

输出：二进制机器代码*.o

格式：gcc -c -o [目标文件] [编译文件]

或 gcc -c [编译文件] -o [目标文件]

选项“-c”可以使编译器完成汇编阶段就停止

选项“-o”是指定GCC输出的结果。

实例：gcc -c test.s -o test.o

链接阶段

链接阶段

输入：二进制机器代码文件*.o

输出：可执行的二进制代码文件

格式：gcc -o［目标文件］［编译文件］

或 gcc［编译文件］ -o ［目标文件］

实例：gcc test.o -o test

$./test //执行可执行程序

直接编译

gcc［编译文件］-o［目标文件］

```
gcc test.c -o test
```

多文件编译

格式1：多文件同时编译

```
gcc 1.c 2.c 3.c  -o test
$./test
```

格式2：每个文件分别进行编译，然后链接成可执行文件

```
gcc  -c 1.c  -o 1.o
gcc  -c 2.c  -o 2.o
gcc  -c 3.c  -o 3.o
gcc 1.o 2.o 3.o  -o test
$./test      //执行可执行程序
```

多文件编译实例

建立文件夹multi，建立三个文件other1.c，other2.c，maintest.c

- other1.c

```
#include <stdio.h>
void welcome()
{
 printf("Welcome to compile multiple files!\n");
}
```

- Other2.c

```
int add(int x, int y)
{  return x+y;}
int sub(int x, int y)
{  return x-y;}
```

多文件编译实例

建立文件夹multi，建立三个文件other1.c，other2.c，maintest.c

- test.c

```
#include <stdio.h>
void main()
{  int a=15,b=3,c;
   printf("test multiple file compile! ");
   welcome();
  c=add(a,b);
  printf("%d+%d=%d\n",a,b,c);
  c=sub(a,b);
  printf("%d-%d=%d\n",a,b,c);
}
```

多文件编译实例

多文件编译：gcc other1.c other2.c test.c -o test

运行：./test

```
[liu@thispc mutifile]$ gcc other1.c other2.c test.c -o test
[liu@thispc mutifile]$ ./test
test multiple file compile!15+3=18
15-3=12
```

为other1.c和other2.c增加头文件

- other1.h

```
#ifndef other1_h
#define other1_h
#include <stdio.h>
void welcome();
#endif
```

- other2.h

```
#ifndef other2_h
#define other2_h
#include <stdio.h>
int add(int, int);
int sub(int, int);
#endif
```

修改other1.c和other2.c

- other1.c

```
#include “other1.h”
void welcome()
{
 printf("Welcome to compile multiple files!\n");
}
```

- other2.c

```
#include “other2.h”
int add(int x, int y)
{  return x+y;}
int sub(int x, int y)
{  return x-y;}
```

修改test.c

- test.c

```
#include <stdio.h>
#include “other1.h”
#include “other2.h”
void main()
{  int a=15,b=3,c;
   printf(“test multiple file compile!”);
   welcome();
  c=add(a,b);
  printf("%d+%d=%d\n",a,b,c);
 c=sub(a,b);
  printf("%d-%d=%d\n",a,b,c);}
```

修改test.c

编译：gcc other1.c other2.c test.c -o test

执行：./test

[liu@thispc mutifile2]$ gcc other1.c other2.c test.c -o test

[liu@thispc mutifile2]$ ./test

test multiple file compile!15+3=18

15-3=12

尝试若将.h 拿到其它文件夹，会有什么问题？

编译器gcc默认在当前目录和/usr/include目录下查找相应的头文件，如果在这两个目录里找不到相应的头文件则报错。
上例中，头文件都放在了当前目录的子目录head里，因此找不到，报错。

3.3 静态与动态链接库

Linux库的创建与使用

库：事先已经编译好的代码，经过编译后可以直接调用的文件，本质上来说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。

系统提供的库的路径

/usr/lib

/usr/lib64

Linux库文件名的组成

前缀（lib）+库名+后缀（.a静态库；.so动态库）

libmm.a：库名为mm的静态库；

libnn.so：库名为nn的动态库。

静态库与动态库的区别

- 静态库的代码在编译时就拷贝到应用程序中，因此当有多个程序同时引用一个静态库函数时，内存中将会调用函数的多个副本。由于是完全拷贝，因此一旦连接成功，静态库就不再需要了，代码体积大。

- 动态库在程序内留下一个标记，指明当程序执行时，首先必须要载入这些库。在程序开始运行后调用库函数时才被载入，被调用函数在内存中只有一个副本，代码体积小。

静态库的创建

- 1. 在一个头文件中声明静态库所导出的函数
- 2. 在一个源文件中实现静态库所导出的函数
- 3. 编译源文件，生成目标文件（*. o）
- 4. 通过命令ar将目标文件加入到某个静态库中
- ar　rcs　静态库名　目标文件列表
- 5. 将静态库拷贝到系统默认的存放库文件的目录或指定目录下

静态库的创建实例

1、2. 使用前面讲解的other1.h, other1.c, other2.h, other2.c, test.c

3.编译生成目标代码

gcc –c other1.c –o other1.o

gcc –c other2.c –o other2.o

将目标文件加入到静态库中

ar rcs libother1.a other1.o

ar rcs libother2.a other2.o

静态库的使用

（1）将静态库拷贝到系统默认的存放库文件的目录

```
sudo cp libother1.a libother2.a /usr/lib/
```

-lname：指示编译器，在链接时，装载名为libname.a的函数库，该函数库位于系统定义的目录或由-L选项选项指定的目录下。例如，-lm表示链接名为libm.a的数学函数库。

```
gcc  -o test test.c  -lother1  -lother2
./test    //可以正常执行
```

静态库的使用

（2）指定库文件路径

-L <dir>：功能是用于指明库文件的路径

```
gcc -o test test.c -lother1 -lother2 -L./
```

-I <dir>:功能是指定头文件路径

例如：将other1.h，other2.h移动到head目录

需要： gcc -o test test.c -lother1 -lother2 -L. -Ihead

静态库的创建实例

（3）可以将多个*.o文件创建一个静态库

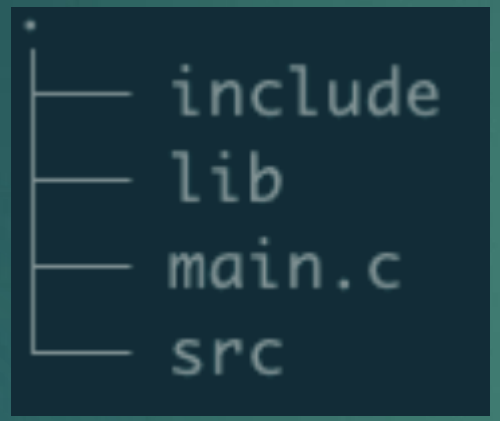
```
ar rcs libmylib.a other1.o other2.o

sudo cp libmylib.a /usr/lib/

gcc  -o maintest test.c  -lmylib
./maintest
```

静态库的推荐用法

- 以如下项目结构构建项目：



  - 调用方式：gcc -o test test.c -I include -Llib -lother1 -lother2
- 不要将生成的库文件，头文件放到/usr/lib,/usr/include, 而是放到/usr/local/lib, /usr/local/include
  - 调用方式：gcc –o test test.c –lother1 –lother2

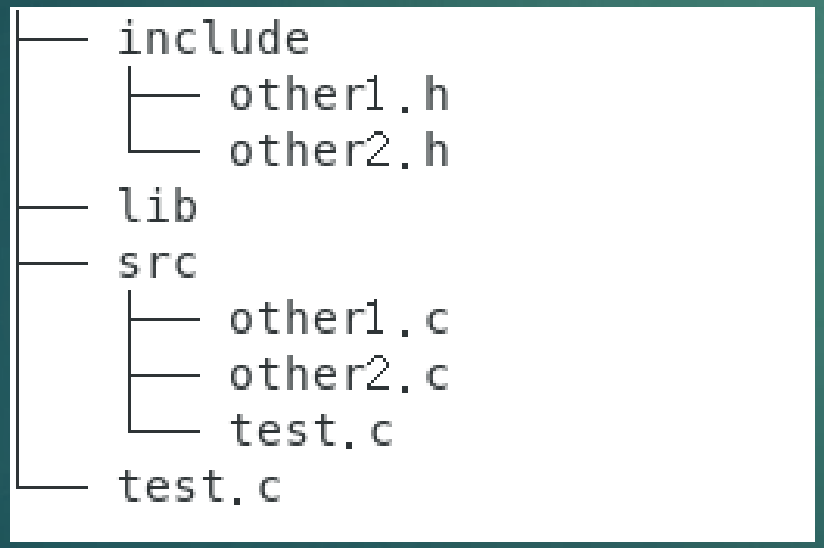
动态的创建步骤

1. 在头文件（.h）中声明动态库所导出的函数
2. 在源文件（.c）中实现动态库所导出的函数
3. 编译源文件，生成与位置无关的目标文件（.o）
4. 创建动态库

动态的创建步骤

1. 在头文件（.h）中声明动态库所导出的函数
2. 在源文件（.c）中实现动态库所导出的函数

以上节项目为例：



动态的创建步骤

3.编译源文件，生成与位置无关的目标文件（.o）

```
gcc -fPIC  *.c -I  ../include  -c
```

```
[liu@thispc src]$ gcc -fPIC *.c -I ../include  -c
[liu@thispc src]$ ls
other1.c  other1.o  other2.c  other2.o
```

动态的创建步骤

4. 创建动态库

gcc -shared -o libdy.so *.o

参数：

```
[liu@thispc src]$ gcc -shared -o libdy.so *.o
[liu@thispc src]$ ls
libdy.so  other1.c  other1.o  other2.c  other2.o
```

动态的使用

1. 方式1

gcc [源文件] -L [动态库路径] -l [动态库名] -I [头文件路径] -o [可执行文件]

```
gcc test.c -Llib -ldy -Iinclude -o test
./test
```

```
[liu@thispc mutifile5]$ ./test
./test: error while loading shared libraries: libdy.so: cannot open shared object file: No such file or directory
o.so.1 =>  (0x00007fff1071a000)
    libdy.so => not found
    libc.so.6 => /lib64/libc.so.6 (0x00007fe7f2626000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fe7f29f4000)
```

执行时动态获取找不到

动态的使用

2. 方式2

gcc［源文件］-I ［头文件路径］［库文件名（libxxx.so）］-o ［可执行文件］

```
gcc test.c -linclude ./lib/libdy.so -o test
./test
```

可以运行，但移动test后不能运行，为什么， 怎么解决？

动态的使用

解决找不到链接库的方法

1. 更新LD_LIBRARY_PATH

export LD_LIBRARY_PATH=[自定义动态库路径]

只起到临时作用

LD_LIBRARY_PATH: 指定查找动态库的路径（除了默认路径），该路径在默认路径前进行查找。

2. 配置环境变量

3. 将动态库的绝对路径写入/etc/ld.so.conf文件，后使用ldconfig -v命令更新

4. 直接将动态库拷贝至/usr/lib等系统目录下（可行但强烈不推荐）

动态的使用

将编译得到的二进制文件拷贝到别的主机，是否能够运行？

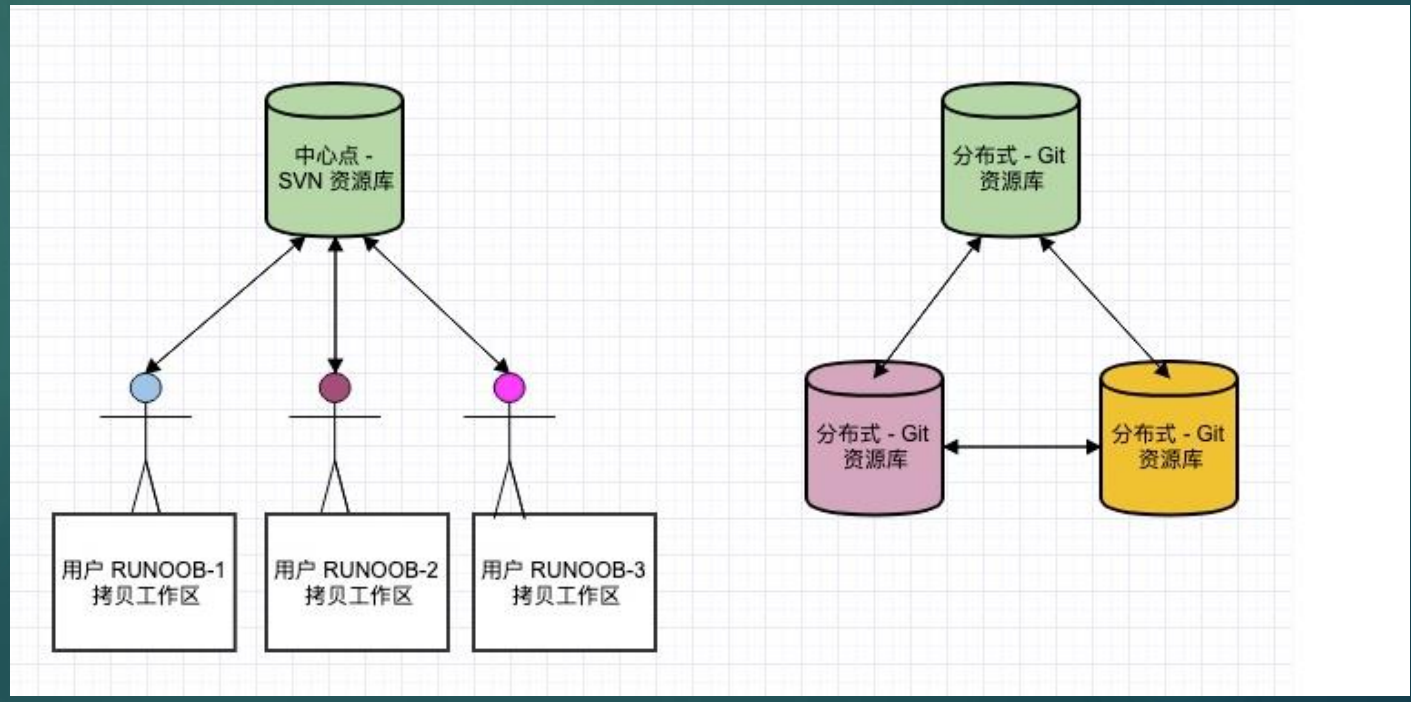
！动态不能，需要把.so也拷贝。

http://mooc1.chaoxing.com/nodedetailcontroller/visitnodedetail?courseId=214915308&knowledgeId=361436629

3.4 分布式版本控制系统git

git简介

- Git 是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。
- Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。
- Git 与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持。



git的安装

- 1.用git命令查看是否已安装

- 2.若未安装　sudo apt-get install git　(ubuntu)

  sudo　yum install git　(centos)

- 3 进行配置，指定使用git的账号和用户名

```
[root@testpc ~]# git config --global user.email "pliu16@mails.jlu.edu.cn"
[root@testpc ~]# git config --list
user.name=Coder-Peng
user.email=pliu16@mails.jlu.edu.cn
```

创建版本库

- 版本库又称仓库（repository），仓库中存放被git管理的文件，每个文件的修改、删除，git都能够跟踪，可以方便追踪历史。

- 方法：

```
[root@testpc ~]# mkdir repo_git
[root@testpc ~]# cd repo_git
[root@testpc repo_git]# git init
Initialized empty Git repository in /root/repo_git/.git/
```

创建成功，多了.git目录，用来跟踪管理版本库，不能删除

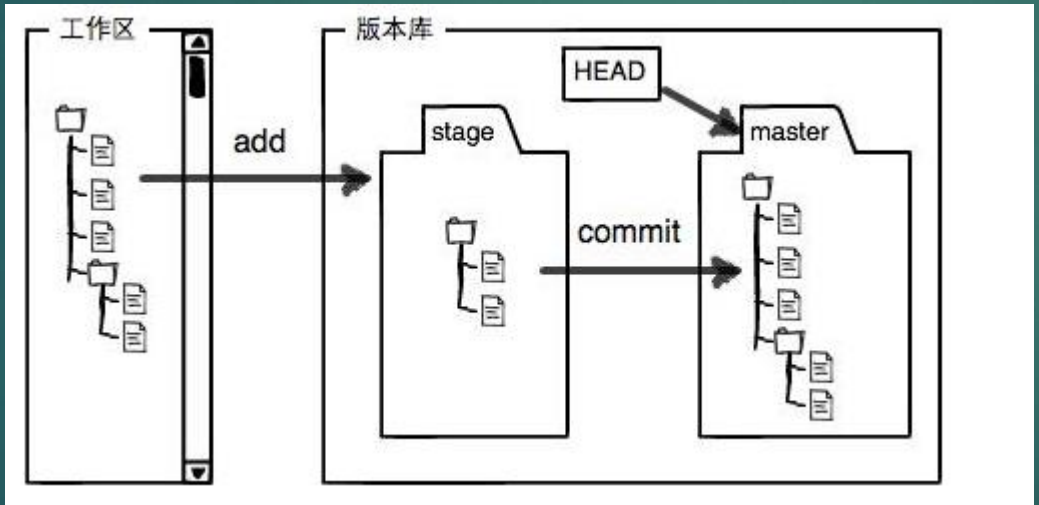
配置用户名、邮箱

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
```

工作区和版本库

工作区指工作目录，而工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

向版本库中添加文件

前提：在刚创建的repo_git目录（或子目录）下建立文件，如 “README.txt”，添加内容

步骤：

```
git add README.txt     # 没有输出
git commit -m “A description for the…”
git status     #查看状态
```

版本回退

前提： 后续开发需要修改之前的文件，如README.txt, 想退回之前的版本

1. git status # 查看状态
2. git diff # 用diff 格式显示不同
3. git add README.txt #

   git commit “add a word” # 提交修改版本
4. git log # 查看提交记录
5. 再次修改，提交

回退：

git reset --hard HEAD^ # 退回后新的不存在了，若git log还存在可用版本号回退

git reset --hard 版本号（来自git log）

git reflog # 记录每一次更改，可找到最新版本

撤销修改（删除）

1. git checkout -- filename，回到最近一次git commit或git add时的状态。

   若未放到暂存区，回到和版本库一样的状态

   放到暂存区，则回到添加到暂存区后的状态

2. Git reset HEAD filename：撤销暂存区中的内容（git add）

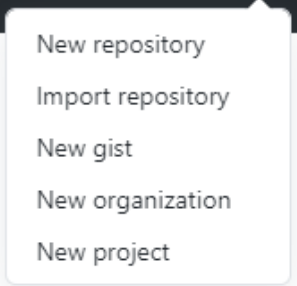
3. git rm filename：删除提交到版本库中的文件 （错删，则git checkout -- filename 从版本库恢复）

远程操作

前提：已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作

步骤：登录github，创建和项目同名repository

```
git remote add origin  git@github.com:Coder-Peng/repo_git.git
git push -u origin master   #第一次提交本地库
```



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?
Import a repository.

Owner * **Repository name** *

Coder-Peng / test_git

Great repository names are short and memorable. Need inspiration? How about **jubilant-octo-fiesta**?

Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

Add a README file
This is where you can write a long description for your project. Learn more.

Add .gitignore
Choose which files not to track from a list of templates. Learn more.

Choose a license
A license tells others what they can and can't do with your code. Learn more.

Create repository

添加配置

[root@testpc repo_git]# ssh-keygen -t rsa -C "pliu16@mails.jlu.edu.cn"

Generating public/private rsa key pair.

Enter file in which to save the key (/root/.ssh/id_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

将id_rsa.pub 复制到github中

远程克隆

远程有项目组其他人上传的项目，或开源项目，需要克隆到本地开发。

步骤：git clone git@github.com:Coder-Peng/repo_git.git

分支管理

分支用于将原版本做快照进行独立的开发，开发后进行合并。

每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支，目前只有一个分支叫做主分支master。

分支的创建：git checkout –b newbranch

#等价于 git branch newbranch git checkout newbranch

git switch –c newbranch　git switch newbranch

查看当前分支：git branch

在newbranch修改文件，提交

切换回master分支：git checkout master

合并分支：git merge newbranch

删除分支：git branch –d newbranch

冲突管理

前提：多个分支修改相同文件，合并时会产生冲突。

例如：创建新分支newbra

git checkout –b newbra

在该分支上修改文件，并提交。

切换回master分支，在master分支上修改并提交。

此时，合并分支newbra的更改将出现冲突

git merge newbra

git status 查看状态， 手动查看文件并解决冲突

在次提交解决冲突后的文件，删除分支newbra

3.5调试器GDB的使用

调试器GDB的介绍

- gdb是GNU开源组织发布的一个强大的Linux下的程序调试工具。
- gdb和其他调试器一样，可以在程序中设置断点、查看变量值，一步一步地跟踪程序的执行过程。
- 利用调试器的这些功能可以方便地找出程序中存在的非语法错误。

启动和退出gdb

- gdb调试对象：可执行程序，并且在gcc编译时必须加上-g选项。
- 实例：编写一个用于调试testgdb.c文件

启动和退出gdb

```
#include <stdio.h>
int get_sum(int n)
{
   int sum=0,i;
   for(i=0; i<n; i++)
      sum+=i;
   return sum;
 }

 int main( )
 {
   int i=100,result;
   result=get_sum(i);
   printf("1+2+...+%d=%d\n",i,result);
   return 0;
 }
```

编译并运行该程序

$ gcc testgdb.c –o test

$ ./test

输出：1+2+…+100=4950

[root@thispc gdb]# ./test

1+2+...+100=4950

程序有错误，正确结果是5050

gdb调试程序的编译

$ gcc –g testgdb.c –o test

要使用gdb进行调试的程序，在编译时需要添加-g 选项。

gdb的启用和退出

- 启动方式1：gdb 程序名

  例如：gdb test

  启动gdb后，首先
  (gdb)之后输入让

  若不想显示版权

- 启动方式2：首先

  gdb –q

  file test

- gdb的退出

  (gdb)quit（q）

```
[liu@bogon gdb]$ gdb test
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-119.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/l
This is free software: you are free to change and redistribu
There is NO WARRANTY, to the extent permitted by law.  Type
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/liu/repo_git/gdb/test...done.
(gdb) q
[liu@bogon gdb]$ aa
```

显示程序源代码list

- list

list: 输出从上次调用list命令开
次运行该命令则显示接下来的

list -（减号）：输出从上次调

list n：输出n行附近的10行代

list function：输出函数func

list 5，10：显示第5到第10行

```
(gdb) list
13        result=get_sum(i);
14        printf("1+2+...+%d=%d\n",i,result);
15        return 0;
16        }
(gdb) list -
3         {
4         int sum=0,i;
5         for(i=0;i<n;i++)
6            sum+=i;
7          return sum;
8         }
9
10        int main()
11        {
12        int i=100,result;
(gdb) list 5,10
5         for(i=0;i<n;i++)
6            sum+=i;
7          return sum;
8         }
9
10        int main()
```

搜索字符串-1

- forward/search：从当前行向后查找某个字符串的程序行，查找时不包括当前行，可以用list n,n,将当前行设置为n。

  forward 字符串或search 字符串

例：

```
(gdb) list 1,1
(gdb) search get_sum
结果：2 int get_sum(int n)

(gdb) list 3,3
(gdb) forward get_sum
结果：13 result=get_sum(i);
```

搜索字符串-2

- reverse-search 字符串：从当前行向前查找第一个匹配的字符串。
- 例：(gdb) list 2,2

  (gdb) reverse-search get_sum

  结果：Expression not found

  (gdb) list 4,4

  (gdb) reverse-search get_sum

  结果：2 int get_sum(int n)

执行程序

- 使用gdb test或file test 只是装入程序，程序并没有运行
- 运行

  (gdb) run

  结果：starting program: 文件路径

  1+2+…+100=4950

  program exited normally

设置和管理断点

- 1、以行号设置断点

  格式：break n

  功能：当程序运行到指定行时，会暂停执行，指定行的代码不执行。

  例：(gdb) break 14

  反馈：Breakpoint 1 at ox8048432: file test.c, line 15

  (gdb) run

结果：starting program: /home/lyj/test

Breakpoint 1， main( ) at test.c 14

14 printf(“1+2+…+%d=%d\n”,i,result);

设置和管理断点

- 2、以函数名设置断点

  格式：break 函数名

  例：(gdb) break get_sum

  反馈：Breakpoint 1 at ox8048432: file test.c, line 4

  (gdb) run

  注意：以函数名设置断点，会停在函数体的第一行。

设置和管理断点

```
 1 #include<stdio.h>
 2 int get_sum(int n)
 3 {
 4 int sum=0,i;
 5 for(i=0;i<n;i++)
 6     sum+=i;
 7  return sum;
 8 }
 9
10 int main()
11 {
12 int i=100,result;
13 result=get_sum(i);
14 printf("3+2+...+%d=%d\n",i,result);
15 return 0;
16 }
```

设置和管理断点

- 3、以条件表达式设置断点

  格式：break 行号或函数名 if 条件

  功能：程序在运行过程中，当某个条件满足时，程序在某行中断暂停执行

  例：(gdb) break 6 if i==99

含义：当程序执行到第6行时，判断条件i==99是否成立，若成立则暂停。

到达该断点时i、sum分别是多少？

```
(gdb) break 6 if i==99
Breakpoint 1 at 0x400544: file testgdb.c, line 6.
(gdb) run
Starting program: /home/liu/repo_git/gdb/test

Breakpoint 1, get_sum (n=100) at testgdb.c:6
6           sum+=i;
Missing separate debuginfos, use: debuginfo-insta
(gdb) print i
$1 = 99
(gdb) print sum
$2 = 4851
```

设置和管理断点

- 4、以条件表达式变化设置断点

  格式：watch 条件表达式

  功能：程序在运行过程中，当某个条件满足时，程序在某行中断暂停执行

  注意：watch必须在程序运行的过程中设置观察点，即run之后才能设置，并且要保证条件表达式中的变量已经使用过。

设置和管理断点

- 4、以条件表达式变化设置断点

  格式：watch 条件表达式

  例1：

  (gdb) break 12

  (gdb) run

  (gdb) watch sum==3

  No symbol “sum” in current context

  例2

  (gdb) break 4

  (gdb) run

  (gdb) watch sum==3

设置和管理断点

- 4、以条件表达式变化设置断点

  例3：观察sum变量的变化情况

```
(gdb) break 4
Breakpoint 1 at 0x400534: file testgdb.c, line 4.
(gdb) run
Starting program: /home/liu/repo_git/gdb/test

Breakpoint 1, get_sum (n=100) at testgdb.c:4
4       int sum=0,i;
Missing separate debuginfos, use: debuginfo-install gli
(gdb) watch sum
Hardware watchpoint 2: sum
(gdb) continue
Continuing.
Hardware watchpoint 2: sum

Old value = 0
New value = 1
get_sum (n=100) at testgdb.c:5
5       for(i=0;i<n;i++)
(gdb) continue
Continuing.
Hardware watchpoint 2: sum

Old value = 1
New value = 3
get_sum (n=100) at testgdb.c:5
5       for(i=0;i<n;i++)
```

设置和管理断点

- 5、查看当前设置的断点

  格式：info breakpoints

  例1：

  (gdb) break 7

  (gdb) break 15 if result==5050

  (gdb) info breakpoints

结果：

1 breakpoint keep y 0x080483fa in get_sum at test.c:7

2 breakpoint keep y 0x08939324 in main at test.c 15 stop only if result==5050

keep:生效一次后不失效

y：是否有效

设置和管理断点

- 6、使中断失效或有效

（1）失效：disable 断点编号

（2）有效：enable 断点编号

- 7、删除断点

（1） clear 行号：删除此行断点

（2） delete：删除程序中所有断点

（3）delete 断点编号：删除指定编号的断点，若一次要删除多个断点，各个断点编号以空格隔开。

设置和管理断点

- 6、使中断失效或有效
- 实例

(gdb) break 6

(gdb) break 7

(gdb) break 8 if sum==5050

(gdb) clear

```
(gdb) break 6
Breakpoint 9 at 0x400544: file testgdb.c, line 6.
(gdb) break 7
Breakpoint 10 at 0x400556: file testgdb.c, line 7.
(gdb) break 8 if sum==5050
Breakpoint 11 at 0x400559: file testgdb.c, line 8.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
9       breakpoint     keep y   0x0000000000400544 in get_sum at testgdb.c:6
10      breakpoint     keep y   0x0000000000400556 in get_sum at testgdb.c:7
11      breakpoint     keep y   0x0000000000400559 in get_sum at testgdb.c:8
        stop only if sum==5050
(gdb) del
Delete all breakpoints? (y or n) y
(gdb) info breakpoints
No breakpoints or watchpoints
```

查看和设置变量的值

- 当程序执行到中断点暂停执行时，往往要查看变量或表达式的值，借此了解程序的执行状态，进而发现问题所在
- 1、print命令

  功能：打印变量或表达式的值，还可以用来对某个变量进行赋值。

  print 变量或表达式：打印变量或表达式的值

  print 变量=值：对变量进行赋值

查看和设置变量的值

- 1、print命令

实例：

(gdb) break 6

(gdb) run

(gdb) print i<n

(gdb) print i

(gdb) print sum

(gdb) print i=200

(gdb)continue　继续

```
(gdb) break 6
Breakpoint 1 at 0x400544: file testgdb.c, line 6.
(gdb) run
Starting program: /home/liu/repo_git/gdb/test

Breakpoint 1, get_sum (n=100) at testgdb.c:6
6           sum+=i;
Missing separate debuginfos, use: debuginfo-instal
(gdb) print i<n
$1 = 1
(gdb) print i>n
$2 = 0
(gdb) print i
$3 = 0
(gdb) print sum
$4 = 0
```

```
(gdb) print i=200
$5 = 200
(gdb) continue
Continuing.
1+2+...+100=200
[Inferior 1 (process 34528) exited normally]
```

查看和设置变量的值

- 2、whatis 命令

  功能：用于显示某个变量或表达式的数据类型。

  格式：whatis 变量或表达式

  实例

```
(gdb) break 7
(gdb) run
(gdb) whatis i
(gdb) type=int
(gdb) whatis sum+0.5
(gdb)type=double
```

查看和设置变量的值

- 3、set 命令
  - 功能：给变量赋值。
  - 格式：set variable 变量=值

控制程序的执行

- 当程序执行到指定的中断点，查看了变量或表达式的值后，可以让程序一直运行下去直到下一个断点或运行完为止。
- 1、continue

程序继续运行，直到下一个断点或运行完为止

- 2、kill

结束当前程序的调试

- 3、next和step

功能：一次一条执行该程序代码

区别：next把函数调用当作一条语句来执行

step跟踪进入函数，一次一条地执行函数内的代码

控制程序的执行

- 例1 next

```
(gdb) break 12
Breakpoint 1 at 0x400563: file testgdb.c, line 12.
(gdb) run
Starting program: /home/liu/repo_git/gdb/test

Breakpoint 1, main () at testgdb.c:12
12      int i=100,result;
Missing separate debuginfos, use: debuginfo-install
(gdb) print i
$1 = 0
(gdb) next
13      result=get_sum(i);
(gdb) print i
$2 = 100
(gdb) print result
$3 = 0
(gdb) next
14      printf("1+2+...+%d=%d\n",i,result);
(gdb) print result
$4 = 4950
(gdb) continue
Continuing.
1+2+...+100=4950
[Inferior 1 (process 36245) exited normally]
(gdb)
```

控制程序的执行

- 例2 step

```
(gdb) break 12
Breakpoint 1 at 0x400563: file testgdb.c, lin
(gdb) run
Starting program: /home/liu/repo_git/gdb/test

Breakpoint 1, main () at testgdb.c:12
12      int i=100,result;
Missing separate debuginfos, use: debuginfo-i
(gdb) print i
$1 = 0
(gdb) step
13      result=get_sum(i);
(gdb) step
get_sum (n=100) at testgdb.c:4
4       int sum=0,i;
(gdb) step
5       for(i=0;i<n;i++)
(gdb) step
6           sum+=i;
(gdb) continue
Continuing.
1+2+...+100=4950
[Inferior 1 (process 36266) exited normally]
(gdb)
```

帮助命令

- 为了降低用户使用 GDB 调试器的学习成本，GDB 提供了 help 命令，它可以帮用户打印出目标命令的功能和具体用法。
- 为了方便用户能够快速地从众多 GDB 命令中查找到目标命令，help 命令根据不同 GDB 命令的功能对它们做了分类，查看某类下的命令，使用help + 类名。

例如：help running

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
```

帮助命令

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

gdb命令及应用

命令	作用
file	后跟需要载入调试的可执行文件名称
kill	终止正在调试的程序
list	列出产生执行文件的源代码的一部分
next	执行一行源代码但不进入函数内部
step	执行一行源代码而且进入函数内部
run	执行当前被调试的程序
continue	继续执行
quit	终止 gdb
watch	使你能监视一个变量的值而不管它何时被改变
print	显示表达式的值
break	在代码里设置断点，这将使程序执行到这里时被挂起
shell	不离开 gdb 就执行 shell 命令

断点设置及查看

格式	作用
break <linenum>	在指定行号停住。
break <function>	在进入指定函数时停住。
break +offset（-offset）	在当前行号的前面或后面的offset行停住，offset为自然数。
break filename:function	在源文件filename的function函数的入口处停住。
break	无参数时表示在下一条指令处停止运行。
break ... if <condition>	在条件成立时断点生效，一般break后也需跟上行号。
info break [n]	查看第n个断点信息，break后无数字，列出所有断点信息。
disable break [n] [range]	使断点不起作用，n指定断点号，range指定范围
enable break[n] [range]	使断点起作用，n指定断点号，range指定范围
clear	清除所有已定义的断点

3.6 工程管理器make的使用

工程管理器make的使用

- make是最常用的构建工具，诞生于1977年，主要用于C语言项目。

- 背景：源代码可以通过编译后执行，但一个软件通常包含大量源码文件，且这些文件中往往存在着相互依赖关系，需要决定编译的顺序并指定链接文件，手动进行编译、链接将十分困难。
- Make工具最主要也是最基本的功能就是通过makefile文件来描述源程序之间的相互关系并自动维护编译工作。而makefile 文件需要按照某种语法进行编写，文件中需要说明如何编译各个源文件并连接生成可执行文件，并要求定义源文件之间的依赖关系。

工程管理器make的使用

- 优势：

可以使用Make工程管理器来提高效率，只需要编写makefile文件制定相应的编译和链接规则，然后在终端执行make命令就可以高效方便的完成最后的编译工作，且如果以后修改个别文件时，make会自动检查出哪些文件被修改过，从而只对这些文件再次进行编译，保证最终的可执行文件是由最新的模块构建的。

make概念

- make: 制作。比如，要做出文件a.txt，可以执行：

  $ make a.txt

- make本身不知道如何做出a.txt，需要告知make如何调用其它命令完成这个目标。

- 比如，假设文件a.txt依赖于b.txt和c.txt，是后面两个文件连接（cat命令）的结合。那么，make需要知道下面的规则：

  a.txt: b.txt c.txt　　// a.txt 依赖b.txt c.txt

  cat b.txt c.txt > a.txt　// 输出重定向到a.txt

此时，make a.txt这条命令的背后实际分为两步：

(1) 确认b.txt和c.txt必须已经存在；

(2) 使用cat命令将两个文件进行合并，输出为新的文件。

make概念

像这样的规则，需要写在一个名为Makefile的文件中，make命令依赖这个文件进行构建。

Makefile也可以写成makefile，或者用命令行参数指定为其它文件名:

```
$ make -f rules.txt
$ make --file=rules.txt
```

make只是一个根据指定的Shell命令进行构建的工具。

Makefile的文件格式

Makefile文件由一系列规则构成。每条规则的形式如下：

```
<target>: <prerequisites>    # 目标: 前置条件
[tab] <command>              # [必须有个table键] <命令>
```

“目标”是必须的；“前置条件”和“命令”是可选的，但两者必存其一。

明确：构建目标的前置条件是什么，以及如何构建。

Makefile的文件格式

1.target

- 一个目标（target）构成一个规则。目标通常是文件名，指明命令所要构建的对象，比如a.txt。
- 目标可以是一个文件名，也可以是多个文件名，之间用空格分隔。

- 除了文件名，目标还可以是某个操作的名字，这称之为“伪目标”（phony target）。 例如：

```
clean：

    rm *.o
```

上面代码的目标是clean，不是一个文件名，是一个操作的名字，属于“伪目标”，作用是删除文件。 $ make clean

Makefile的文件格式

1.target

- 有可能在当前路径下，刚好有一个文件叫做clean

为了避免这种情况，可以声明clean是“伪目标”：

```
.PHONY: clean
clean:
    rm *.o
```

- 声明clean是“伪目标”之后，make不会去检查是否存在一个叫做clean的文件，而是每次运行都执行对应的命令。

Makefile的文件格式

- Makefile有多个目标，如果make命令运行时没有指定目标，默认会执行Makefile文件的第一个目标。

Makefile的文件格式

2. prerequisites

- 前置条件通常是一组文件名，之间用空格分隔。它指定了“目标”是否重新构建的判断标准: 只要一个前置文件不存在，或者更新过，“目标”就需要重新构建。

```
result.txt: source.txt

    cp source.txt result.txt
```

- 构建result.txt的前置条件时source.txt。如果当前路径下，source.txt已经存在，那么make result.txt可以正常运行；否则必须再写一条规则，用以生成source.txt。

Makefile的文件格式

- 再写一条规则:

  source.txt:

  echo “this is the source” > source.txt

source.txt没有前置条件，意味着跟其它文件无关，只要source.txt不存在，每次调用make source.txt，都会生成source.txt。

Makefile的文件格式

- 连续执行两次make result.txt。

- 第一次会先创建source.txt，再创建result.txt。
- 第二次执行，make发现source.txt没有更新，就会不执行任何操作。

Makefile的文件格式

3. command

- 命令(command)表示如何更新目标文件，由一行或多行Shell命令组成；
- 是构建“目标”的具体指令；
- 结果通常是生成目标文件。
- 每行命令之前必须有一个tab键。如果想用其它键，可以用内置变量.RECIPEPREFIX声明。

  例：.RECIPEPREFIX=> ， 用.RECIPEPREFIX指定大于号(>)替代tab键。

Makefile的文件格式

3. command

- 每行命令都是在一个单独运行的Shell中执行的，这些Shell间没有继承关系。

var:

```
export foo=bar
echo “foo=[$$foo]”
```

结果：

Makefile的文件格式

解决方法：

(1) ;

(2) \　#　“;”后可以换行

(3) .ONESHELL

Make使用实例（构建）

一个简单的makefile文件

```
main: main.o module1.o module2.o
	gcc main.o module1.o module2.o –o main
main.o: main.c head1.h head2.h common_head.h
	gcc –c main.c
module1.o: module1.c head1.h
	gcc –c module1.c
module2.o: module2.c head2.h
	gcc –c module2.c
```

Make使用实例

(1)第一次执行

$make

执行后输出：

gcc –c main.c

gcc –c module1.c

gcc –c module2.c

gcc main.o module1.o module2.o –o main

Make使用实例

(2)部分修改后执行

假设修改了头文件head1.h

$make

执行后输出：

gcc –c main.c

gcc –c module1.c

gcc main.o module1.o module2.o –o main

案例（源码 安装gdb）

1.获取源码：wget http://ftp.gnu.org/gnu/gdb/gdb-9.2.tar.xz

2.解压： tar –Jxvf gdb-9.2.tar.xz

3.查看README文件，会告诉如何安装

4.执行.configure，检查相关依赖、权限，创建makefile文件

5.执行make，依据makefile文件进行编译，编译成可执行文件，但尚未安装到预定目录

6.执行make install，编译完成的文件安装到指定目录，完成安装