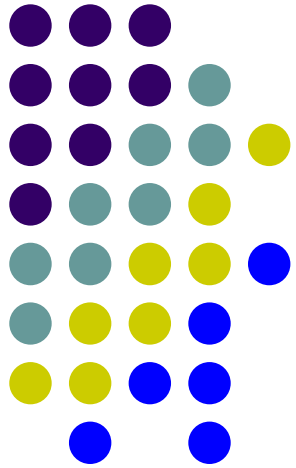
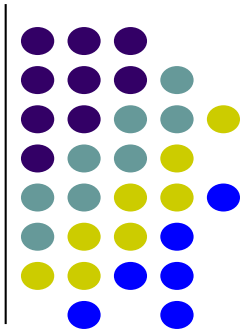
第8章

Linux下的C编程

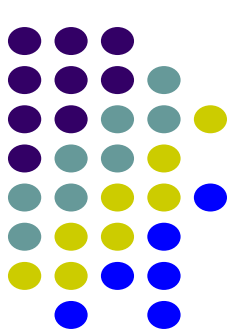


本章内容



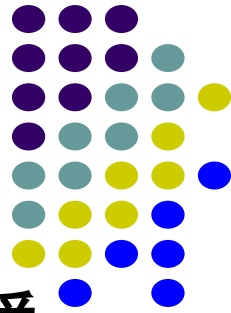
- 文本编辑器
- gcc编译
- gdb调试
- 使用make

Linux编程



- 大多数Linux软件是经过自由软件基金会（Free Software Foundation）提供的GNU公开认证授权的，因而通常被称作GNU软件。GNU软件免费提供给用户使用，并被证明是非常可靠和高效的。许多流行的Linux实用程序如C编译器、shell和编辑器都是GNU软件应用程序。
- Linux发行版中包含很多文本编辑器及软件开发工具，很多是基于C和C++等开发的。
- Linux下的程序设计与其他环境中的程序设计一样，主要涉及编辑器、编译链接器、调试器及项目管理工具。

Linux编程

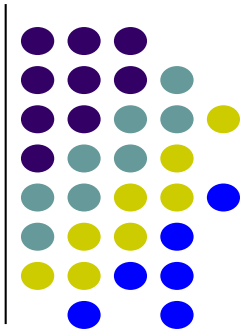


- **编辑器**——Linux中最常用的编辑器有Vi（Vim）和Emacs，它们功能强大，使用方便，广受编程爱好者的喜爱。
- **编译链接器**——在Linux中，最常用的编译器是Gcc编译器。
- **调试器**——Gdb是绝大多数Linux 开发人员所使用的调试器，它可以方便地设置断点、单步跟踪等，足以满足开发人员的需要。
- **项目管理器**——Linux中的项目管理器“make”有些类似于Windows中Visual C++里的“工程”，它是一种控制编译或者重复编译软件的工具，另外，它还能自动管理软件编译的内容、方式和时机，使程序员能够把精力集中在代码的编写上而不是在源代码的组织上。

8.1 文本编辑器vi

- Linux中有许多文本编辑工具，但由于vi编辑器是最常用、最标准的编辑器。某些Linux版本中不提供其他编辑程序，但必须都提供vi这一基本的编辑程序。对于所有版本的Linux系统，vi编辑器是完全相同的。
- vi是visual interface的简称，vi是Linux系统的第一个全屏幕交互式编辑程序，从诞生至今，一直得到广大用户的青睐。利用它可以建立、修改文本文件。在编辑文本过程中，可以执行输出、删除、查找、替换及块操作等众多文本操作。

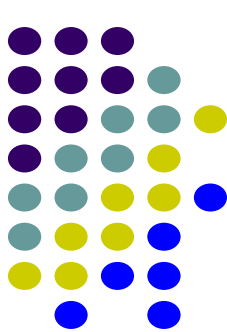
vi的三种工作模式



- Vi编辑器的工作模式包括三种：
  - 命令行模式——编辑器的普通操作，如移动光标、查找/替换等。
  - 文本插入模式——在编辑器内输入文本信息。
  - 末行模式——执行一些特殊命令
- Vi编辑器的三种模式可相互切换。

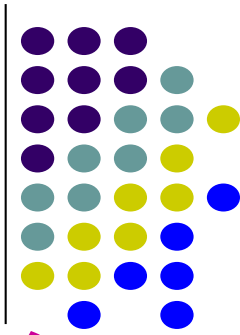
注意：改变vi的工作模式意味着改变键盘上按键的功能。

命令行模式



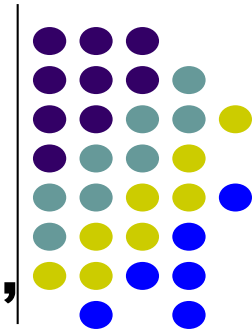
- 启动vi后首先就是处于命令行模式。在任何模式下，键入“Esc”键都会自动进入命令模式。此时，所有从键盘的键入都被作为命令来接收。用户可以输入各种合法的vi命令，用于管理自己的文档。此时，键盘上的任何字符都被当做编辑命令来解释。注意的是，所输入的命令并不在屏幕上显示出来。若输入的字符不是vi的合法命令，vi会响铃提示用户。
- vi编辑器的命令模式下所有的命令都是单字符的命令。

文本输入模式

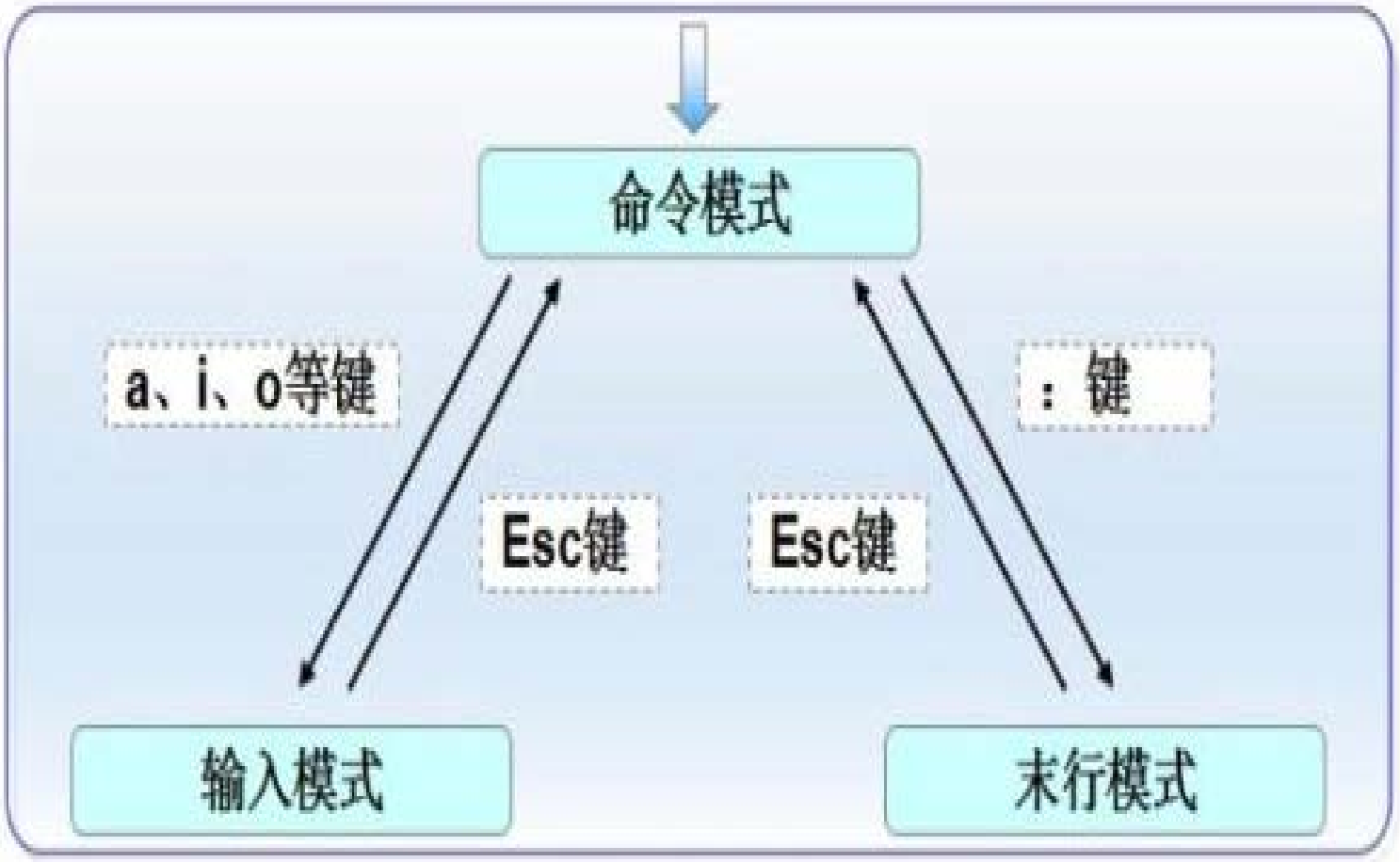


- 在命令模式下输入插入命令i、附加命令a、打开命令o、修改命令c、取代命令r或替换命令s都可以进入文本输入模式，也称编辑模式或插入模式。
- 一旦进入文本输入模式，用户输入的任何字符都被vi当作文件内容保存起来，并在屏幕上显示。键入“Esc”键，vi编辑器将自动回到命令模式，此时所有的按键又称为命令键。

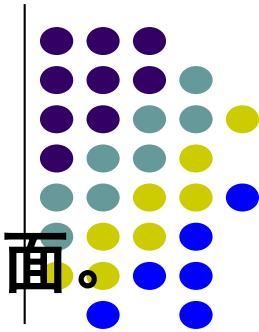
末行模式



- 在命令模式下，按“:”键即可进入末行模式。（此时，在屏幕的底行将出现一个编辑行，而光标将出现该行的行首。）
- 在末行模式下，可以在该行上键入字符串形式的末行命令。（例如：文件存盘退出，进入末行模式，最后键入字符“wq”并回车即可实现编辑程序文件的保存，同时退出vi编辑器。）
- 末行模式只是暂时性的模式，键入“Enter”键执行完末行命令后将自动退出该模式，并自动进入命令模式或退出vi编辑器。


命令模式
a、i、o等键
：键
Esc键
Esc键
输入模式
末行模式

vi的运行与退出



- 使用vi进行编辑工作的第一步是进入编辑模式界面。

启动vi时可附带选项参数，以适应某些特殊的应用场合。

基本格式：vi［选项］文件名列表

常用选项说明：

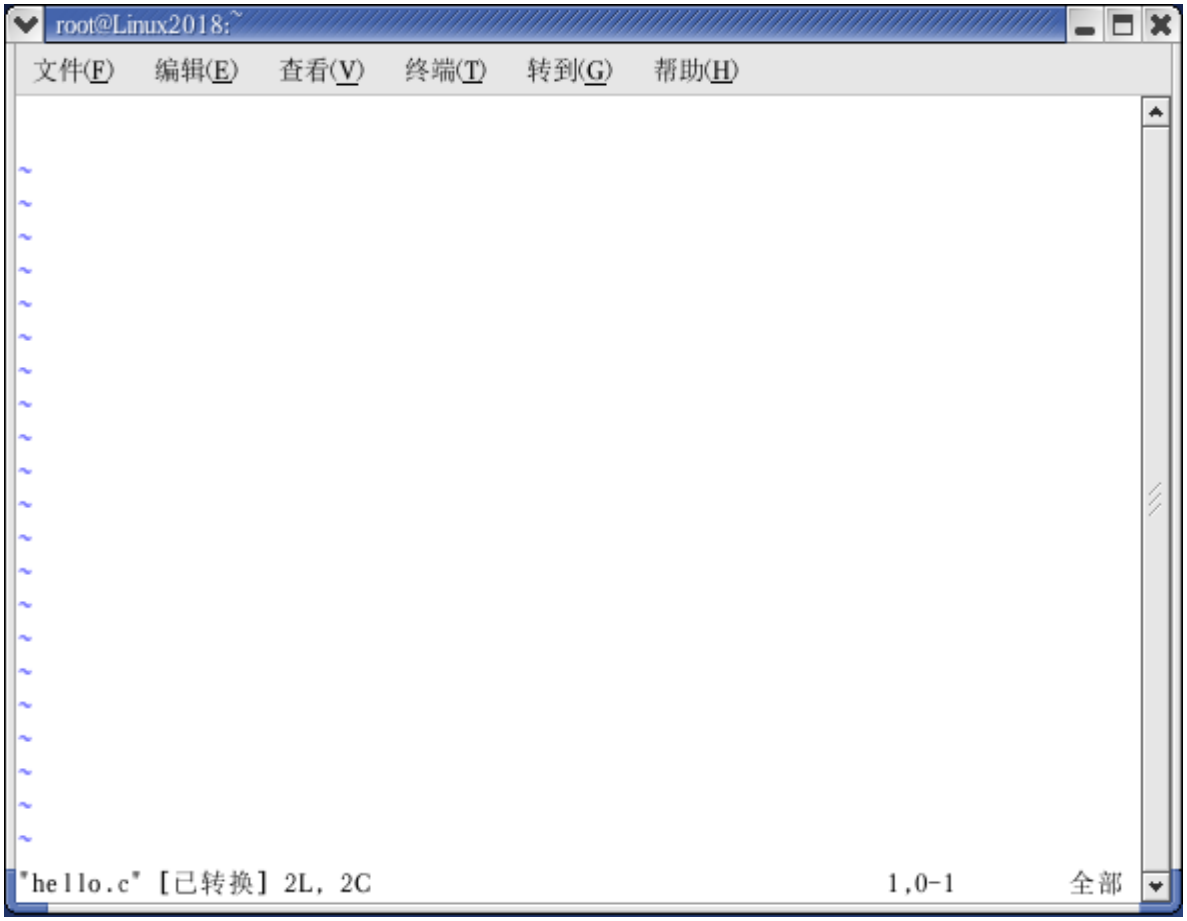
-r　用于恢复系统突然崩溃时正在编辑的文件；

-R　用于以只读方式打开文件；

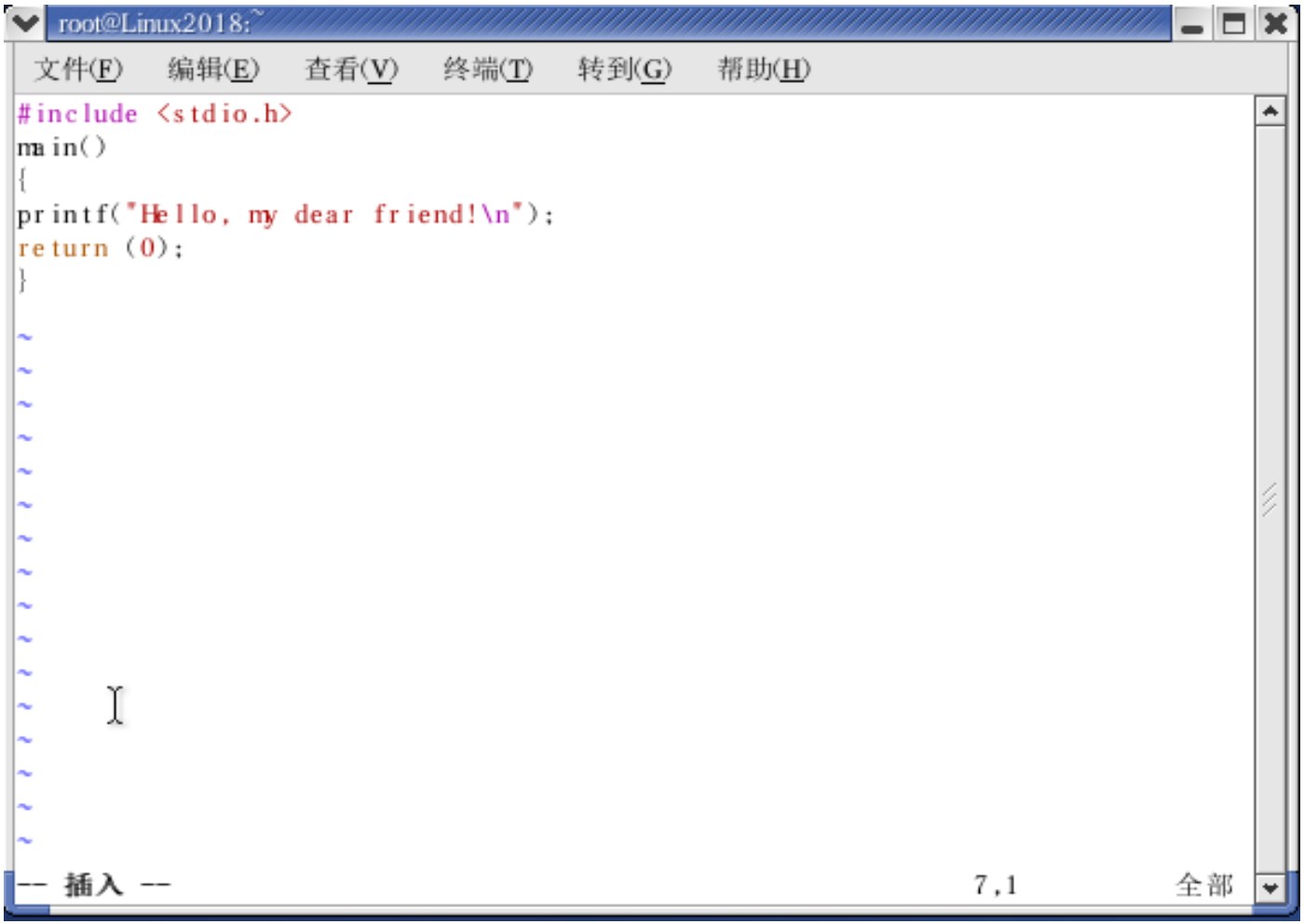
+n　用来指明进入vi编辑器后，屏幕显示到文件的第n行为止，如果不指定n，则显示到末行。

例如：在shell命令行下，键入 vi hello.c

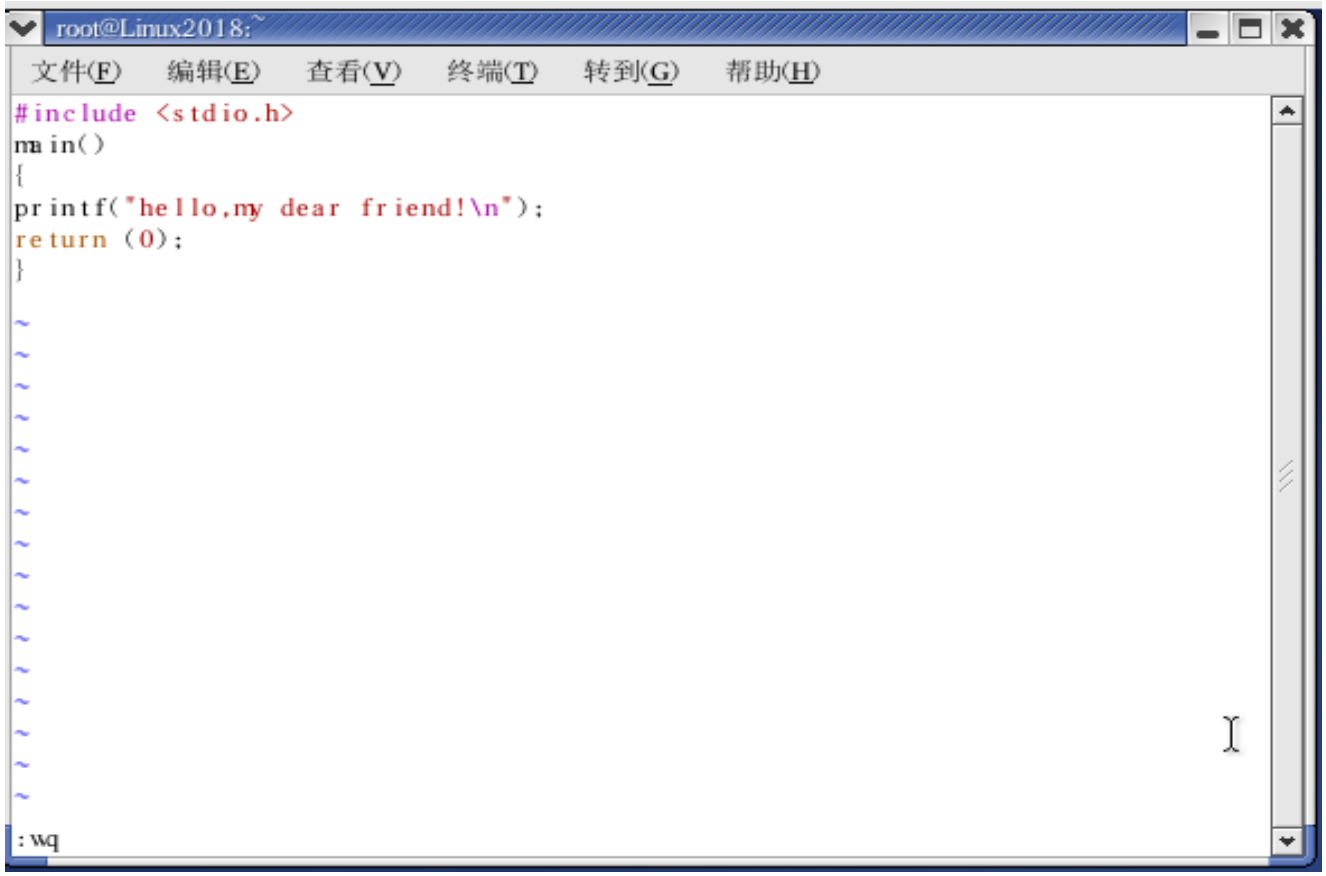
［root@ Linux　root］# vi hello.c



如果当前目录下有hello.c文件，则打开该文件，如果当前目录下无该文件，则新建hello.c文件。此时进入的是命令模式，光标位于屏幕的顶端。

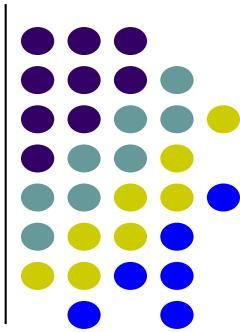


在命令行模式下键入i，可切换Vi编辑器到插入模式，在屏幕底部显示有“插入”表示插入模式，在该模式下可以录入程序源代码。



在插入模式中，按下“Esc”建，则当前模式转入到命令模式，再键入“:”键，则从命令模式转入到末行模式，并在底行显示“:”，在末行模式下键入字符串命令“wq”（存盘退出），界面如图所示，未键入回车时，末行命令不被执行，键入回车可发现编辑器退出。

末行命令



(1) 列出行号

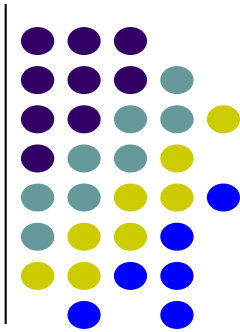
：se nu

输入“se nu”后，会在文件中的每一行前面列出行号。

(2) 跳到文件中的某一行

：#，“#”号表示一个数字，在冒号后输入一个数字，再按回车键就会跳到该行了，例如输入数字15，再回车，就会跳到文章的第15行。

末行命令

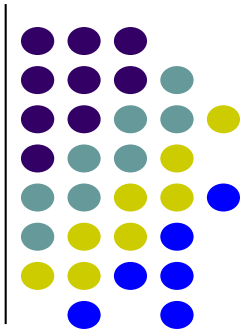


(3) 查找字符

：/关键字，先按“/”，再输入您想寻找的字符，如果第一次找的关键字不是您想要的，可以一直按“n”会往后寻找到您要的关键字为止。

：?关键字，先按“?”键，再输入您想寻找的字符，如果第一次找的关键字不是您想要的，可以一直按“n”会往前寻找到您要的关键字为止。

末行命令



(4) 保存文件

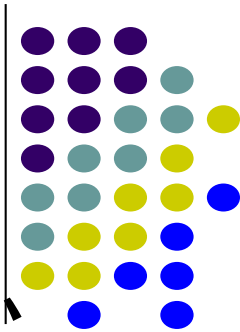
：w，在冒号后面输入字母w就可以将文件保存起来。

：w filename 将编辑的文本保存到名为filename的文件中。

(5) 离开vi

：q，在冒号后面输入字母q就是退出vi，如果无法离开vi，可以在q后跟一个“!”强制离开vi。一般建议离开时，搭配wq一起使用，这样在退出的时候是先保存文件再退出。

vi基本编辑命令

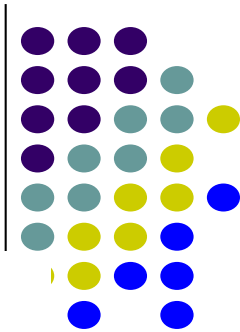


Vi编辑器的文本编辑命令包括移动光标、滚屏、删改、复制、粘贴等操作。

1．基本的光标移动

- 在vi末行模式下，只能在末行输入末行命令，不能移动光标；
- 在vi文本插入模式下，可以使用键盘上的4个方向键移动光标，其他字符键或组合键只能在光标所在位置插入一个新的字符；
- 在vi的命令模式下，也可以使用一些特殊键或组合键来整屏移动文本。

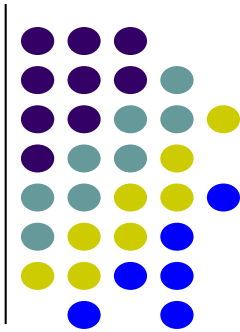
光标移动操作



光标移动命令如下：

↑	光标上移一行；
←	光标左移一个字符；
→	光标右移一个字符；
↓	光标下移一行；
0（数字0）	光标移至行首，等价于Home键；
$	光标移至行尾，等价于End键；

滚屏命令



ctrl+b　　屏幕往“后”翻动一页，等价于PageDown键；

ctrl+f　　屏幕往“前”翻动一页，等价于PageUp键；

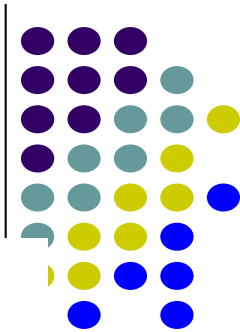
- 可以在这两个命令之前加上一个数字n，则屏幕向前或向后翻滚n页。

ctrl+u　　屏幕往“后”翻动半页；

ctrl+d　　屏幕往“前”翻动半页；

- 可以在这两个命令之前加上一个数字n，则屏幕向前或向后翻滚n行。

状态命令

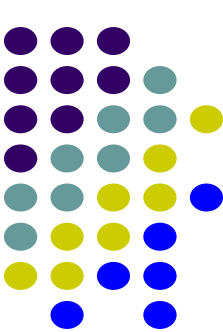


- 查看Vi状态行上的状态信息，包括正在编辑的文件名、是否修改过、当前行号、文件的行数以及光标之前的行占整个文件的百分比。使用“ctrl+g”。

vi命令模式下的基本命令

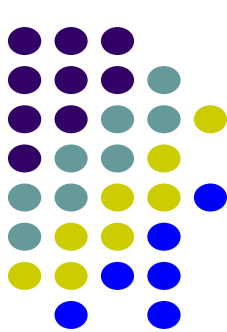
命令	作用	命令	作用
i	切换到插入模式，此时可以从光标当前位置插入文本；	dG	删除从光标至文件尾的内容；
a	切换到插入模式，此时可以在光标所在字符后插入文本；	d1G	删除从光标至文件首的内容；
o	切换到插入模式，在光标所在位置下插入新行来输入文本；	yy	复制光标所在行；
I	切换到插入模式，此时光标处于行首；	nyy	从光标所在行开始，向下复制n行，类似于Windows环境下，选中n行后，再按下的“Ctrl+C”；
A	切换到插入模式，此时光标处于行尾；	p	将缓冲区内的信息粘贴到光标所在位置（与yy或nyy搭配），类似于Windows环境下，选中n行后，再按下的“Ctrl+V”；
O	切换到插入模式，在光标所在位置上插入新行来输入文本；	r	修改光标所在位置的字符；
x	删除光标所在位置的一个字符；	R	修改字符，直到按下“Esc”键为止；
X	删除光标前一个字符；	/string	在光标之后查找一个名为string的字符串（string可以是任意字符串）；
dd	将光标当前所在行剪切到剪贴板；	?string	在光标之前查找一个名为string的字符串（string可以是任意字符串）；
ndd	从光标所在行开始，向下剪切n行到剪贴板，类似于Windows环境下，选中n行后，再按下的“Ctrl+X”；	u	取消上一次的操作，相当于undo。

Vi插入模式下的基本命令



- vi插入模式的基本命令只有一个，也就是键入“Esc”键退到命令模式。
- 键盘上的功能键与Windows环境下类似，如方向键、“PageUp”与“PageDown”键、“Insert”与“Delete”键、“Home”与“End”键等。
- 键入字符键或数字键相当于在光标位置新插入相应的文本。

8.2 GCC编译器

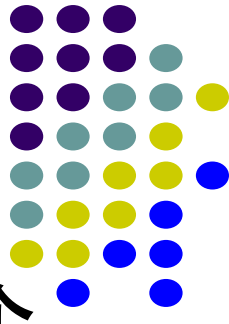


- GCC（GNU Compiler Collection）是GNU推出的功能强性能优越的编译器。
- GCC是可以在多种平台上编译出可执行程序的编译器集合，集成C、C++、Objective C、Java等多种语言编译器。
- 在Linux系统中，编译器通过程序的扩展名可分辨出编写原始程序代码所用的语言，由于不同程序所需要执行编译的步骤是不同的。因此GCC可根据不同的扩展名对它们进行分别处理。

GCC所支持的扩展名文件

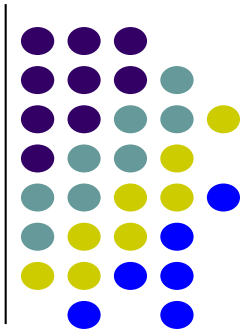
扩 展 名	文 件 类 型	后续编译流程
.c	C语言源代码文件	预处理、编译、汇编、链接
.C/.cc/.cxx	C++源代码文件	预处理、编译、汇编、链接
.m	Objective-C源代码文件	预处理、编译、汇编、链接
.i	已经预处理过的C源代码文件	编译、汇编、链接
.ii	已经预处理过的C++源代码文件	编译、
.s	汇编语言源代码文件	汇编、链接
.S	经过预编译的汇编语言源代码文件	汇编、链接
.a	由目标文件构成的档案库文件	链接
.o	编译后的目标文件	链接
.h	程序所包含的头文件	

GCC编译过程



- 使用GCC编译程序时,编译过程可以被细分为四个阶段:C/C++编译的控制
  - 预处理：对源代码文件中的文件包含(include)、预编译语句(如宏定义define等)进行分析。
  - 编译：就是把C/C++代码“翻译”成汇编代码。
  - 汇编：将第二步输出的汇编代码翻译成符合一定格式的机器代码，生成以.s为后缀的目标文件。
  - 链接：将上步生成的目标文件和系统库的目标文件和库文件链接起来，最终生成了可以在特定平台运行的可执行文件。

GCC语法格式



使用GCC编译器的基本语法格式如下:

gcc［选项］准备编译的文件［选项］［目标文件］

常用选项:

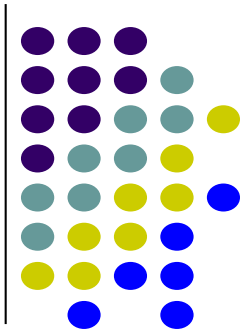
-E　　预处理，预处理之后的代码将送往标准输出

-S　　编译为汇编代码

-c　　编译为目标文件，不连接库

说明：如果目标文件缺省，gcc编译出来的程序后缀是一个名为a.out的可执行文件。

GCC编译



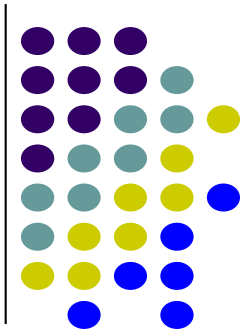
举例说明gcc的最基本语法，首先创建一个HelloWorld.c的文件

Vi HelloWorld.c

在vi中输入如下内容，并保存。

```
#include <stdio.h>
int main()
{
   printf( “Hello World!\n");
   return (0) ;
}
```

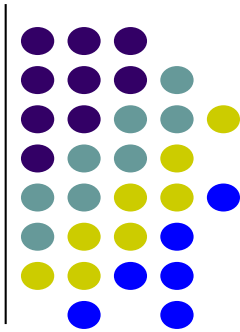
GCC编译



然后编译运行

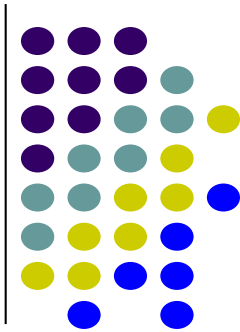
```
[root@linux root]# gcc HelloWorld.c
[root@linux root]# ./a.out
   Hello World!
[root@linux root]#
```

- 这样，gcc 编译器会生成一个名为a.out的可执行文件，然后执行./a.out就可以看到程序的输出结果了。 在终端上显示：Hello World!
- 在执行a.out文件时，在文件前面添加./，这是让Shell在当前目录下去寻找需要运行的可执行文件。



下面来具体的分析这四个阶段:
首先创建一个hello.c的文件
Vi hello.c
在vi中输入如下内容，并保存。

```
#include <stdio.h>
int main()
{
   printf("Hello, my dear friend!\n");
   return (0) ;
}
```



1．预处理阶段

执行如下命令进行hello.c程序的预处理:

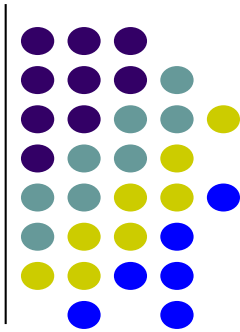
```
[root@linux root]# gcc -E hello.c -o hello.i
```

该命令使用选项“-E”指定只进行预处理，“hello.c”是源程序文件，选项“-o”指定生成目标文件，“hello.i”是预处理过程生成的目标文件。因此，本次编译到预处理结束后就停止编译过程，但要把源代码中的stdio.h编译进来。

使用如下命令查看预处理过程生成的目标文件hello.i。

```
[root@localhost root]# vi hello.i
```

下图给出了观察到的hello.i文件末尾部分的截图，可以看出预处理过后，hello.i文件有839行，因为预处理过程把头文件stdio.h编译进来了。



```
817
818
819
820
821
822 extern char *ctermid (char *__s) ;
823 # 655 "/usr/include/stdio.h" 3
824 extern void flockfile (FILE *__stream) ;
825
826
827
828 extern int ftrylockfile (FILE *__stream) ;
829
830
831 extern void funlockfile (FILE *__stream) ;
832 # 679 "/usr/include/stdio.h" 3
833
834 # 2 "hello.c" 2
835 main()
836 {
837 printf("Hello, my dear friend!\n");
838 return (0);
839 }
:se nu                                                    839,1         底端
```

2. 编译阶段

执行如下命令对预处理文件hello.i进行编译。

```
[root@linux root]# gcc -S hello.i -o hello.S
```

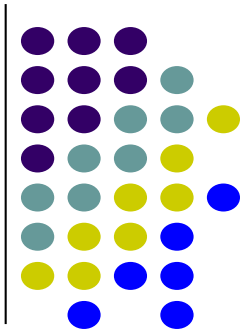
该命令使用选项“-S”指定只进行到编译阶段，“hello.i”是进行编译的源文件，选项“-o”指定生成目标文件，“hello.S”是编译生成的目标文件名。因此，本次编译到编译阶段结束后就停止编译过程，不进入汇编阶段。

使用如下命令查看编译过程生成的目标文件hello.S。

```
[root@linux root]# vi hello.S
```

下图给出了观察到的hello.S文件的部分截图，可以看出编译阶段过后，GCC已经将hello.i文件转化为汇编文件。

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello, my dear friend!\n"
        .text
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        leave
        ret
.Lfe1:
        .size   main,.Lfe1-main
"hello.S" [已转换] 24L, 392C                                  1,2-9         顶端
```



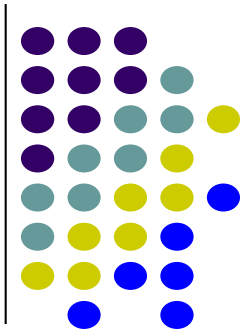
3．汇编阶段

执行如下命令对文件hello.S进行汇编。

```
[root@linux root]# gcc -c hello.S -o hello.o
```

该命令使用选项“-c”指定只进行到汇编阶段结束为止，“hello.S”是进行汇编的源文件，选项“-o”指定生成目标文件，“hello.o”是编译生成的目标文件名。

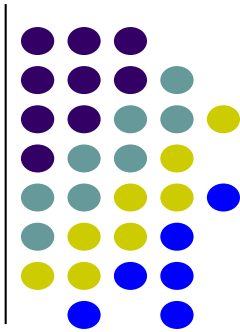
因此，本次编译到汇编阶段结束后就停止编译过程，不进入链接阶段。hello.o为二进制目标代码文件。



4. 链接阶段

函数库一般分为静态库和动态库两种。

- 静态库是指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了。其后缀名一般为“.a”。
- 动态库与之相反，在编译链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时运行链接文件加载库，这样可以节省系统的开销。动态库一般后缀名为“.so”，如前面所述的libc.so.6就是动态库。gcc在编译时默认使用动态库。



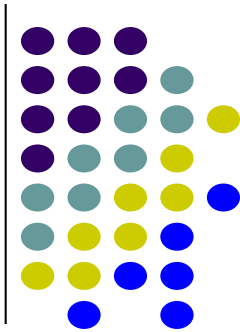
完成链接之后，gcc就可生成可执行文件，命令如下:

```
[root@linux root]# gcc hello.o -o hello
```

该命令gcc之后无选项参数，表示对指定的源文件进行编译，直到输出执行文件，示例中的源文件为hello.o，输出的执行文件hello。

运行该可执行文件,出现正确的结果如下。

```
[root@linux root]#./hello
Hello,my dear friend!
```

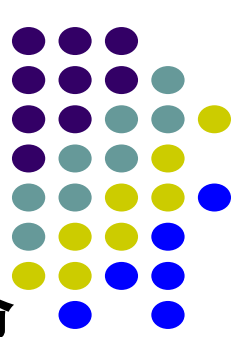


同时编译多个文件

- 在采用模块化的设计思想进行软件开发时，通常整个程序是由多个源文件组成的，相应的也就形成了多个编译单元，使用gcc能够很好地管理这些编译单元。
- 假设有一个由foo1.c和foo2.c两个源文件组成的程序，为了对它们进行编译，并最终生成可执行文件foo，可以使用下面这条命令：

```
[root@linux root]# gcc foo1.c foo2.c -o foo
```

GCC编译选项介绍

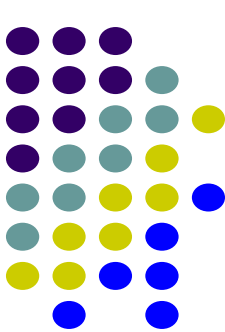


- gcc有超过100个的编译选项，具体的可以使用命令man gcc查看。
- gcc的常用选项:

“-E”、“-o”、“-S”、“-c”选项在前一小节中已经介绍了其使用方法。下面介绍另外几个常用的选项：

-g：产生调试工具(GNU的gdb)所必要的符号信息，要想对编译出的程序进行调试，就加入这个选项。

GCC编译选项



-I dir选项

-I dir选项可以在头文件的搜索路径列表中添加dir目录。

Linux中的头文件的默认位置是“/usr/include/”目录，因此，当用户希望添加放置在其他位置的头文件时，就可以通过“-I dir”选项来指定，这样。

例如：在“/root/work/gcc/”目录下有两个文件hello1.c和my.h，而hello1.c 刚好添加了my.h头文件。

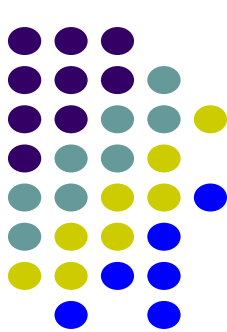
【清单1】

```
#include <my.h>
int main()
{
   printf("Hello!!\n");
   return 0;
}
```

【清单2】

```
#include<stdio.h>
```

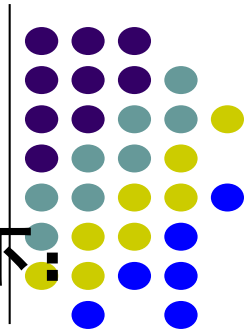
GCC编译选项



这样，就可在gcc命令行中加入“-I”选项，使得编译过程在搜索头文件时包含指定的路径信息，gcc就能够执行出正确结果。

```
# gcc hello1.c -I /root/work/gcc/ -o hello1
# ./hello1
    Hello!!
```

GCC编译选项

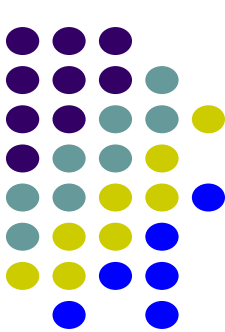


如果更改hello1.c的头文件引用方式，程序清单3如下：

```
#include  “my.h”
int main()
{
    printf("Hello!!\n");
    return 0;
}
```

可不在gcc命令行中加入“-I dir”选项，因为在include语句中，< >表示在标准路径中搜索，“”表示在本目录中搜索。

GCC编译选项



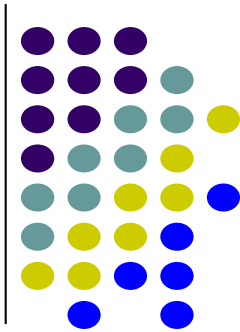
-L dir选项

选项“-L dir”的功能与“-I dir”类似，能够在库文件的搜索路径列表中添加dir目录。例如有程序hello2.c需要用到目录“/root/work/gcc/lib/”下的一个动态库libxch.so，则只需键入如下命令即可：

```
#gcc hello2.c -L /root/work/gcc/lib/ hello2
```

注意：“-I dir”和“-L dir”都只是指定了路径，而没有指定文件，因此不能在路径中包含文件名。

GCC编译选项

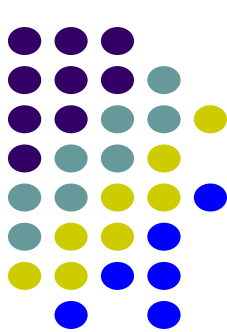


-static：静态链接库文件

例如：gcc -static hello.c -o hello

当使用静态库时，连接器找出程序所需的函数，然后将它们拷贝到可执行文件，一旦连接成功，静态程序库也就不再需要了。对动态库而言，就不是这样，动态库会在执行程序内留下一个标记‘指明当程序执行时，首先必须载入这个库。动态库节省空间。（可以比较静态链接与动态链接可执行文件的大小）

GCC编译选项



出错检查与警告提示选项

gcc包含完整的出错检查和警告提示功能，它们可以帮助Linux程序员写出更加专业和优美的代码。

选项	作用
-ansi	支持符合ANSI标准的C程序;
-pedantic	允许发出ANSI C标准所列的全部警告信息;
-pedantic-error	允许发出ANSI C标准所列的全部错误信息;
-w	关闭所有告警;
-Wall	允许发出Gcc提供的所有有用的告警信息;

使用示例：

C程序文件illcode.c，程序如清单4：

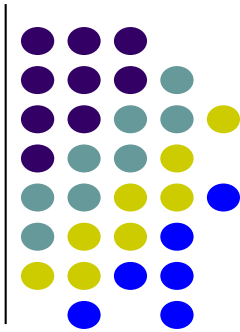
【清单4】

```
#include<stdio.h>
void main()
{
    long long var = 1;
    printf("This is not a standard C code!\n");
}
```

这段代码存在的问题：

- main函数的返回值被声明为void，但实际上应该是int；
- 使用了GNU语法扩展，即使用long long来声明64位整数，不符合ANSI/ISO C语言标准；

GCC编译选项



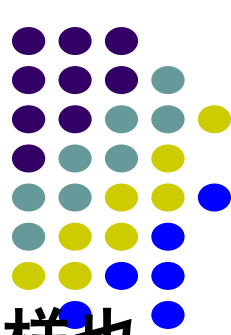
-ansi选项

该选项强制GCC生成标准语法所要求的告警信息，尽管这还并不能保证所有没有警告的程序都是符合ANSI C标准的。运行结果如下所示：

```
[root@linux root]# gcc -ansi illcode.c -o illcode
illcode.c:in function ‘main’:
illcode.c:3:warning: return type of ‘main’ is not ‘int’
```

从运行结果看，该选项并没有发现“long long”这个无效数据类型的错误，只发现了程序第3行的main函数返回类型错误。

GCC编译选项



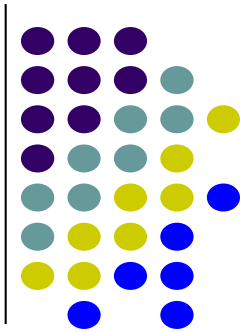
-pedantic选项

该选项允许发出ANSI C标准所列的全部警告信息，同样也保证所有没有警告的程序都是符合ANSI C标准的。其运行结果如下所示：

```
[root@linux root]# gcc -pedantic illcode.c -o illcode
illcode.c:in function ‘main’:
illcode.c:3:warning: return type of ‘main’ is not ‘int’
illcode.c:4:warning:ISO C90 does not support ‘long long’
```

从运行结果看，使用该选项查看出了“long long”这个无效数据类型的错误。

GCC编译选项



-Wall选项

该选项允许发出gcc能够提供的所有有用的告警信息，例如：

```
[root@linux root]# gcc -Wall illcode.c -o illcode
illcode.c:3:warning: return type of ‘main’ is not ‘int’
illcode.c:in function ‘main’:
illcode.c:4: warning: unused variablr ‘var’
```

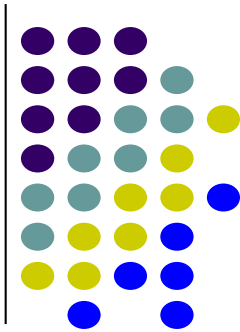
使用“-Wall”选项找出了未使用的变量var，但它并没有找出无效数据类型的错误。

GCC编译选项

优化选项

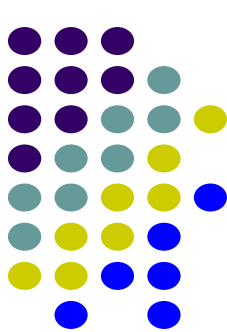
- gcc可以对代码进行优化，它通过编译选项“-On”来控制优化代码的生成，其中n是一个代表优化级别的整数。对于不同版本的gcc来讲，n的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从0变化到3。
- 不同的优化级别对应不同的优化处理工作。如使用优化选项“-O”主要表示没有优化。“-O1”为默认级别的优化。使用优化选项“-O2”除了完成所有“-O1”级别的优化之外，同时还要进行一些额外的调整工作，如处理器指令调度等。选项“-O3”则还包括循环展开和其他一些与处理器特性相关的优化工作。

例如：源程序文件名为optimize.c，程序代码如清单5:



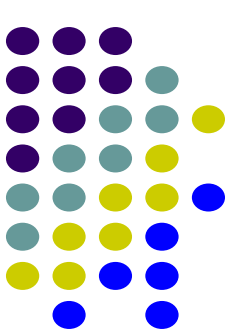
```
#include <stdio.h>
int main(void)
{
   double c;
   double result;
   double temp;
   for(c=0;c<2000.0*2000.0*2000.0/20.0+2020;c+=(5-1)/4)
   {
      temp = c/1979;
      result = c;
   }
   printf("Result is %lf\n", result);
   return(0);
}
```

首先不加任何优化选项进行编译，并借助Linux提供的time命令统计出该程序在运行时所需要的时间。然后使用优化选项来对代码进行优化处理，再次测试一下运行时间，两次运行的时间对比情况如图所示。



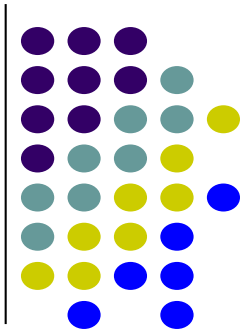
```
[root@Linux2018 root]# vi optimize.c
[root@Linux2018 root]# gcc -Wall optimize.c -o optimize
[root@Linux2018 root]# time ./optimize
Result is 202019.000000

real    0m0.003s
user    0m0.000s
sys     0m0.000s
[root@Linux2018 root]# gcc -Wall -O  optimize.c -o optimize
[root@Linux2018 root]# time ./optimize
Result is 202019.000000

real    0m0.001s
user    0m0.000s
sys     0m0.000s
[root@Linux2018 root]#
```

8.3 GDB调试器



- 一个功能强大的调试器不仅为程序员提供了跟踪程序的手段，还能帮助程序员找到解决问题的方法。GDB是GNU开源组织发布的一个强大的UNIX下的程序调试工具。
- 对于Linux程序员来讲，GDB（GNU Debugger）通过与GCC的配合使用，为基于Linux的软件开发提供了一个完善的调试环境。
- 默认情况下，GCC在编译时不会将调试符号插入到生成的二进制代码中，因为这样会增加可执行文件的大小。如果需要在编译时生成调试符号信息，可以使用GCC的-g选项。

GDB调试器



- GDB具有如下几个主要的功能：

（1）查看文件；

（2）设置断点；

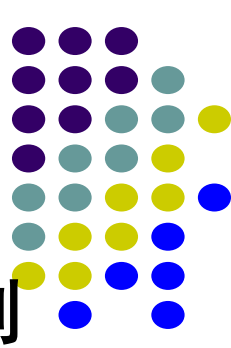
（3）查看断点情况；

（4）监视程序中变量的值；

（5）单步运行；

（6）恢复程序运行；

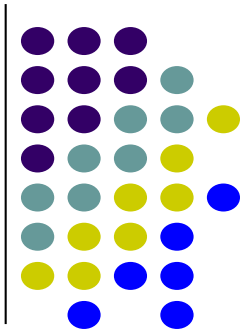
GDB基本命令



- GDB支持很多命令，这些命令从简单的文件装入到检查所调试的堆栈的内容，可以实现不同的功能。

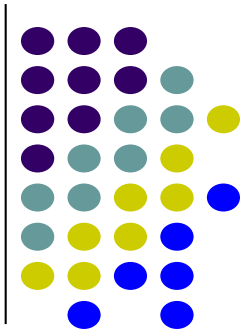
file	装入想要调试的可执行文件
kill	异常终止在gdb 控制下运行的程序
list	列出正在执行的程序的源代码的一部分
next	执行一行源代码但不进入函数内部
run	执行当前被调试的程序
quit	退出gdb
watch	监视一个变量的值而不管它何时被改变
Break	设置一个断点，使程序执行到这里时被挂起

GDB基本命令（续）



make	不退出gdb的情况下重新产生可执行文件
clear	删除刚才停止处的断点
info	显示与该程序有关的各种信息
print	显示变量或表达式的值
pwd	显示当前工作目录
pype	显示一个数据结构的内容
set variable	给变量赋值
signal	将一个信号发送到正在运行的进程
step	执行一行源代码而且进入函数内部
until	结束当前循环
whatis	显示变量或函数类型
info breakpoint	列出当前所设置的所有观察点
enable breakpoint	禁止断点
disable breakpoint	启用断点

GDB使用流程

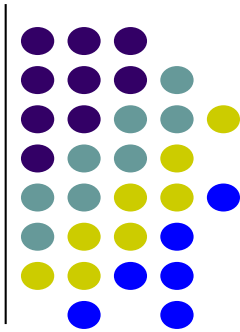


举例：程序文件名为test.c，程序代码如清单6,通过对该程序的调试示例熟悉GDB的使用流程。

【清单6】

```
#include <stdio.h>
void sum(int m);
int main()
{
   int i,n=0;
   sum(50);
   for(i=1; i<=50; i++)
   {
      n += i;
   }
printf("The sum of 1-50 is %d \n", n );
   return(0);
}
void sum(int m)
{
   int i,n=0;
   for(i=1; i<=m;i++) n += i;
   printf("The sum of 1-m is %d\n", n);
}
```

GDB调试器



首先使用gcc对test.c进行编译。

注意：gdb进行调试的是可执行文件，而不是如“.c”的源代码，因此，需要先通过gcc编译生成可执行文件才能用gdb进行调试。编译时一定要加上选项“-g”，这样编译出的可执行代码中才包含调试信息，否则gdb无法载入该可执行文件。

```
[root@linux root]# gcc -g test.c -o test
```

这段程序没有错误，但调试完全正确的程序可以更加了解GDB的使用流程。

进入GDB调试环境

```
[root@linux root]# gdb test
GNU gdb Red Hat Linux (6.3.0.0-1.132.EL5rh)
Copyright 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb)
```

可以看出，在GDB的启动画面中指出了GDB的版本号、使用的库文件等信息，最后进入“（gdb）”命令行调试界面。

1．查看程序源文件

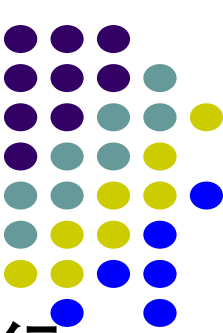
在GDB调试器中，查看源程序文件的调试命令是“list”，使用示例如下：

```
(gdb) list
1 #include <stdio.h>
2 void sum(int m);
3 int main()
4 {
5           int i,n=0;
6           sum(50);
7           for(i=1; i<=50; i++)
8           {
9                       n += i;
10          }
```

```
(gdb) list
11          printf("The sum of 1～50  is %d \n", n );
12          return（0）;
13 }
14 void sum(int m)
15 {
16          int i,n=0;
17          for(i=1; i<=m;i++) n += i;
18          printf("The sum of  1～m is = %d\n", n);
19 }
 （gdb)
```

说明：每次执行list命令只显示10行源程序文件，如果程序超过10行，则可继续使用list命令接着显示程序后面的代码。）

进入GDB调试环境



2. 设置断点

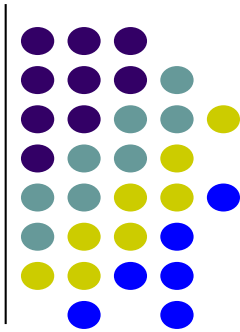
设置断点是调试程序中的一个重要手段，它可以使程序执行到一定位置时暂停运行。程序员在该位置处可以方便地查看变量的值、堆栈情况等，从而找出代码的错误所在。

在gdb中设置断点非常简单，设置命令为“break n”（或简写成‘b’）来设置断点,n表示断点所在的行号，具体使用示例如下：

```
(gdb) break 6
Breakpoint 1 at 0x8048384: file test.c, line 6.
```

注意：在gdb中利用行号设置断点是指代码运行到对应行之前将其停止，如上例中，代码运行到第5行之后暂停（并没有运行第6 行）。

GDB调试器



3. 查看断点设置情况

在设置完断点之后，可以键入“info b”来查看设置断点情况。

```
(gdb) info b
Num   Type        Disp   Enb   Address        What
1   breakpoint    keep   y    0x0804833f    in main at test.c:6
```

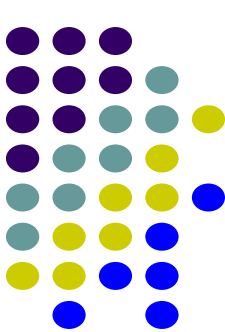
4. 运行程序

接下来就可运行程序了，gdb默认从首行开始运行程序，运行程序的命令为“run”，若想从程序中指定行开始运行，可在run后面加上行号。

```
(gdb) run
Starting program: /root/myprog/test

Breakpoint 1, main () at test.c:6
6 sum(50);
```

注意：程序运行到断点处就停止了。

GDB调试器

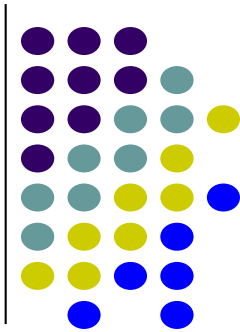


5. 查看变量值

在程序停止运行之后，程序员所要做的工作是查看断点处的相关变量值。在gdb中可查看变量的命令为“print 变量名”，使用示例如下：

```
(gdb) print n
$1 = 0
(gdb) print i
$2 = 1073828704
```

分析：为什么变量“i”的值为如此奇怪的一个数字呢？而变量“n”的值却是正常的？



6. 单步运行

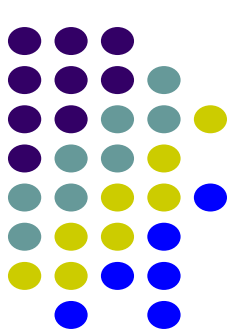
单步运行命令：“next”或“step”，

“next”或“step”的区别：若有函数调用的时候，“step”会进入该函数而“next”不会进入该函数。

使用示例如下：

```
(gdb)next
The sum of 1-m is 1275
7 for(i=1; i<=50; i++)

(gdb) step
sum (m=50) at test.c:16
16 int i,n=0;
```

可见，使用“next”后，程序显示函数sum的运行结果并向下执行，而使用“step”后则进入到sum函数之中单步运行。



7. 恢复程序运行

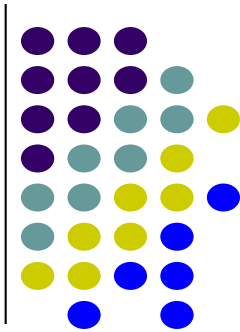
程序运行中断后，可以使用命令“c”（continue）恢复程序的正常运行了。这时，它会把剩余还未执行的程序执行完，并显示剩余程序中的执行结果。以下是之前使用“next”命令恢复后的执行结果：

```
(gdb) c 50
Continuing.
The sum of 1-50 is :1275
Program exited normally.
```

可以看出，程序在运行完后正常退出，之后处于“停止状态”。

输入quit或者按下Ctrl-d退出调试状态。

GDB调试器



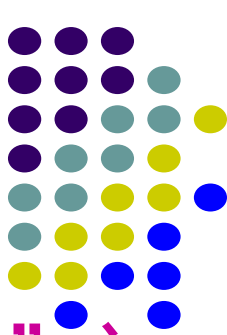
- GDB中设置断点有多种方式，如

  按行设置断点；

  按函数设置断点；

  按条件设置断点。

- 按行设置断点的方法前文已介绍，下面结合前一节的代码，具体介绍后两种设置断点的方法。



函数断点：

GDB中按函数设置断点只需把函数名放在命令“break”之后，

如下所示：

```
(gdb) break sum
Breakpoint 1 at 0x80483d1: file test.c, line 16.
(gdb) info b
Num Type       Disp  Enb  Address      What
1 breakpoint   keep  y    0x0804838a   in sum at test.c:16
```

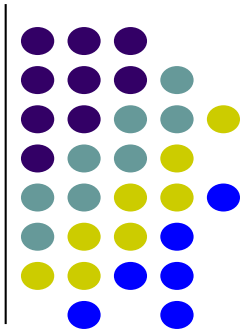
条件断点：

gdb中设置条件断点的格式为：

break行数和函数名if表达式。具体实例如下所示：

```
(gdb) break 8 if i= =10
Breakpoint 1 at 0x804835b: file test.c, line 8.
(gdb) info b
Num    Type    Disp  Enb     Address  What
1 breakpoint keep   y   0x0804835b in main at test.c:8
stop only if i = = 10
(gdb) run
Starting program: /root/myprog/test
The sum of 1-m is 1275
Breakpoint 1, main () at test.c:9
9 n += i;
(gdb) print i
$1 = 10
```

GDB调试器

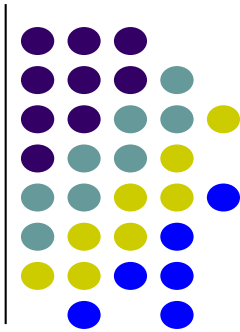


- GDB中修改运行参数相关命令

gdb还可以修改运行时的参数，并使该变量按照用户当前输入的值继续运行。

设置方法为：在单步执行的过程中，键入命令:

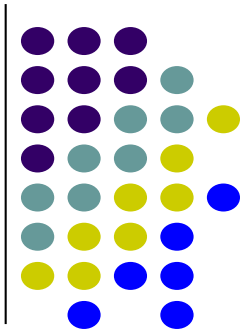
“set 变量＝设定值”。这样，在此之后，程序就会按照该设定的值运行。结合上一节的代码将n的初始值设为4，示例如下所示：



```
(gdb) break 7
Breakpoint 5 at 0x8048391: file test.c, line 7.
(gdb) run
Starting program: /root/myprog/test
The sum of 1-m is 1275
Breakpoint 5, main () at test.c:7
7 for(i=1; i<=50; i++)
(Gdb) set n=4
(Gdb) c
Continuingc
The sum of 1-50 is 1279
Program exited normally.
```

可以看到，最后的运行结果确实比之前的值大。

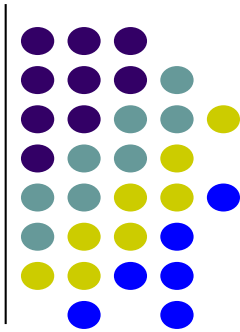
GDB调试器



查看所设断点情况：

```
（gdb） info breakpoint
Num    Type      Disp    Enb      Address          What
1 breakpoint   keep    y    0x0804838a     in sum at test.c:16
2 breakpoint   keep    y    0x0804833f     in main at test.c:6
(gdb) disable breakpoint 1
Num    Type      Disp    Enb      Address          What
1 breakpoint   keep    n    0x0804838a     in sum at test.c:16
2 breakpoint   keep    y    0x0804833f     in main at test.c:6
(gdb) ensable breakpoint 1
Num    Type      Disp    Enb      Address          What
1 breakpoint   keep    y    0x0804838a     in sum at test.c:16
2 breakpoint   keep    y    0x0804833f     in main at test.c:6
```

GDB调试

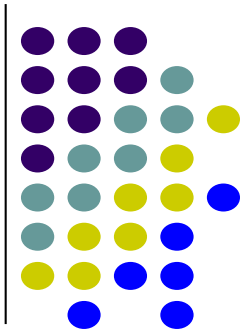


有了以上的感性认识，下面再举例用gdb调试问题代码：

程序文件名为greeting.c，程序代码如清单7

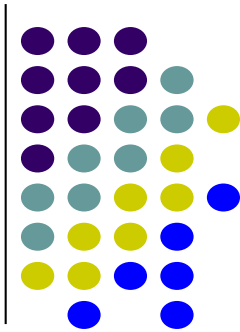
```
#include <stdio.h>
void my_print (char *string);
void my_print2 (char *string);
main( )
{
  char my_string[ ] = "hello there";
  my_print (my_string);
  my_print2 (my_string);
}
void  my_print (char *string)
{
  printf ("The string is %s\n", string);
}
```

GDB调试



```
void my_print2 (char *string)
{
  char  *string2;
  int  size, i;
  size = strlen (string);
  string2 = (char *) malloc (size + 1);

 for (i = 0; i < size; i++)
       string2[size - i] = string[i];
  string2[size + 1] = '\0';
  printf ("The string printed backward is %s\n", string2);
}
```

GDB调试



该程序执行时，显示如下结果：

The string is hello there

The string printed backward is

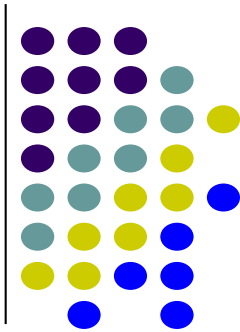
输出的第一行是正确的，但第二行打印出来的信息有误，所设想的输出应该是：

The string printed backward is ereht olleh

下面用gdb对程序进行调试

```
# gdb greeting
(gdb) file greeting
(gdb) run
```

GDB调试



Starting program:/root/greeting

The string is hello there

The string printed backward is

Program exited with code 040

可以看出，函数my_printf2没有正常工作，为了找出bug所在，可以在my_printf2函数的for语句后面设置一个断点，先使用list命令列出源代码。

根据列出的源程序，可以看出要设置断点的地方在第22行。键入设置断点命令：

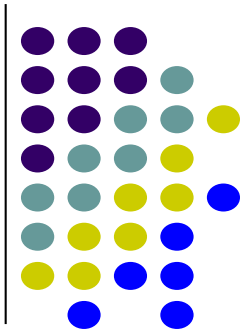
（gdb) break 22

gdb将做出如下显示：

Breakpoint 1 at ox804842e:file greeting.c line 22

(gdb)

GDB调试



再使用run命令，输出如下结果：

Starting program:/root/greeting

The string is hello there

Breakpoint 1,my_print2(string=oxbfffee00“hello there”) at greeting.c :22

22 string2[size-i]=string[i];

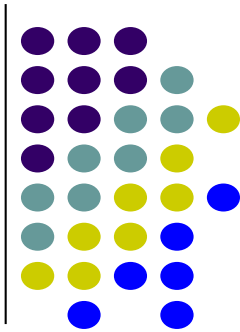
通过设置对string2[size-i]变量的值的观察点，来检查错误产生的原因。输入命令及输出如下：

（gdb) watch string2[size-i]

Watchpoint 2: string2[size-i]

观察点设好后，下面用next命令来分步执行for循环：

GDB调试



（gdb) next

经过第一次循环后，gdb显示string2[size-i]的值是“h”,gdb输出如下提示：

Watchpoint 2，string2[size-i]

Old value = 0 ‘\0’

New value= 104 ‘h’

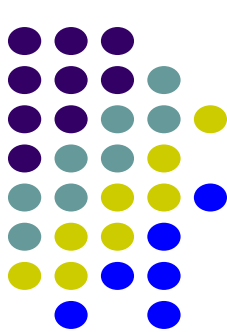
my_print2(string=oxbfffee00“hello there”) at greeting.c :21

21　　for(i = 0; i < size; i++)

(gdb)

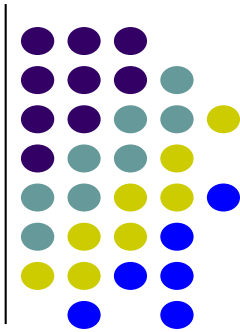
该值是正确的，后来的数次循环结果都是正确的。当 i = 10时，表达式string2[size-i]的值等于“e”,size-i的值等于1，最后一个字符已经复制到新字符串了。

GDB调试



- 再继续循环执行下去，将会看到已经没有值分配给string2[0]了。而它是新串的第一个字符。因为malloc函数在分配内存时把该字符串初始化为空（null)字符。所以string2的第一个字符是空字符，这就是为什么在打印string2时没有输出的原因。
- 修正错误，只需要把代码里写入string2的第一个字符的偏移量改为size-1，而不是size。因为string2的大小为12，但起始偏移量是0，串内的字符从偏移量0到偏移量10，偏移量11为空字符保留，从而引起了字符错位，导致无法获得正确的输出结果。

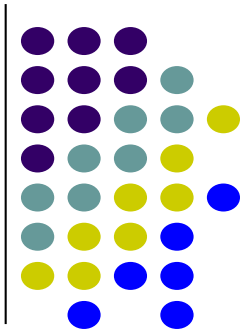
GDB调试



修正后的代码：

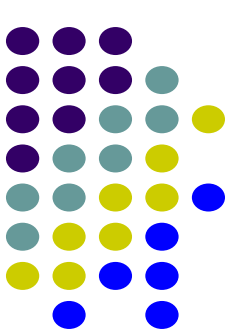
```
#include <stdio.h>
void my_print (char *string);
void my_print2 (char *string);
main( )
{
  char my_string[ ] = "hello there";
  my_print (my_string);
  my_print2 (my_string);
}
void my_print (char *string)
{
  printf ("The string is %s\n", string);
}
```

GDB调试



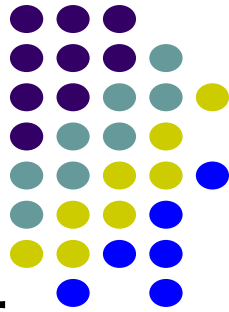
```
void my_print2 (char *string)
{
  char *string2;
  int size, size2, i;
  size = strlen (string);
  size2 = size-1;
  string2 = (char *) malloc (size + 1);
 for (i = 0; i < size; i++)
       string2[siz2 - i] = string[i];
  string2[size] = '\0';
  printf ("The string printed backward is %s\n", string2);
}
```

8.4 make工程管理器



- 对于繁多的源文件，如果每次都要用gcc命令进行编译的话，那么程序员来说是一件很困难的事情。而make工具则可以自动完成编译工作，并且可以只对程序员在上次编译后修改过的部分进行编译。从而减少编译的工作量。
- make工程管理器就是一个“自动编译管理器”，这里的“自动”是指它能够将大型的开发项目分解为多个更为易于管理的模块，简洁明快地理顺各个源文件之间纷繁复杂的相对关系。因此，有效地利用make和make file工具可以大大提高项目开发的效率。

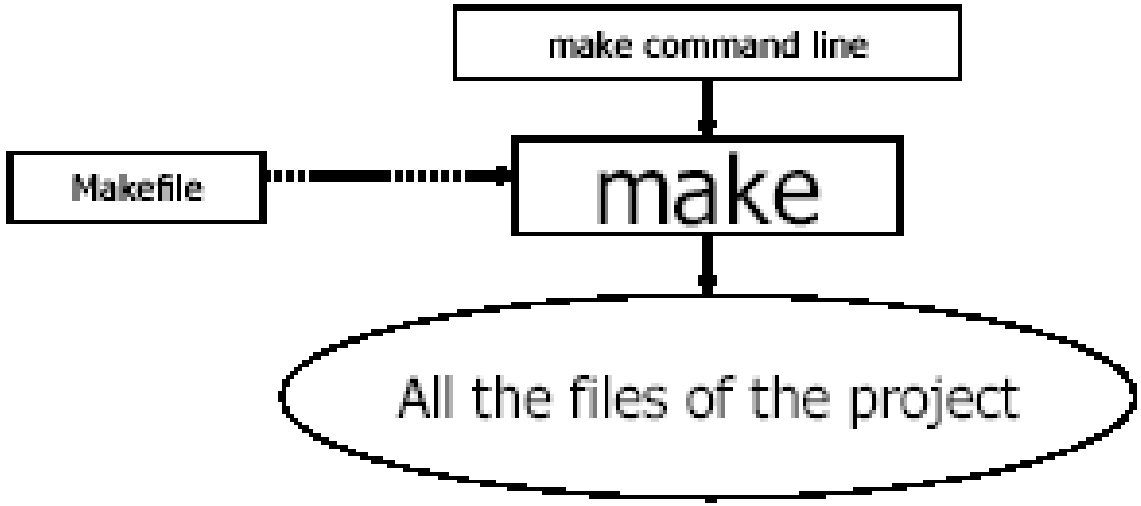
Makefile文件



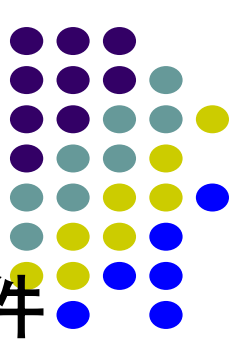
- make工具最主要最基本的功能是通过makefile文件来描述源程序之间的相互关系并自动维护编译工作。
- makefile是一个文本形式的数据库文件，其中包含一些规则，它告诉make编译哪些文件、怎样编译以及在什么条件下去编译。
- 换句话说，这些规则主要是描述哪些文件（称为target目标文件，不要和编译时产生的目标文件相混淆）是从哪些别的文件（称为dependency依赖文件）中产生的，以及用什么命令（command）来执行这个过程。

Makefile文件

- 既然make的一切行为都依据makefile文件，那么编写好makefile至关重要。Makefile说明了组成程序的各模块间的相互关系及更新模块时必须进行的动作。
- make按照这些说明自动地维护这些模块。在源文件无更改的情况下，make什么也不做。



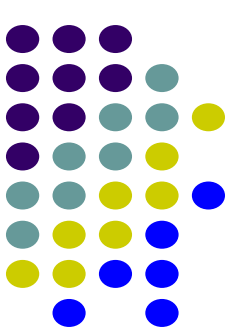
Makefile



例1，有3个头文件：a.h、b.h、c.h，3个C语言源文件：main.c、2.c、3.c，具体情况如下：

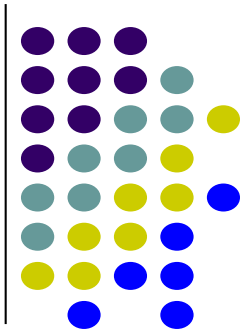
```
/******main.c*****/
#include “a.h”
…
/******2.c*****/
#include “a.h”
#include “b.h”
…
/******3.c*****/
#include “b.h”
#include “c.h”
…
```

Makefile



- 如果程序员只修改文件c.h，则源文件main.c和2.c无需修改，3.c需要重新编译。但如果修改的是文件b.h，而程序员忘记重新编译源文件2.c，则最终程序就可能无法正常工作。
- 可见手动处理这些问题很容易出错，那么自动的解决办法，就是写一个makefile文件和源代码放在同一个目录下，make工具会依据makefile文件中说明的源文件之间的依赖关系和构建规则，在必要时重新编译所有受改动影响的源文件。

Makefile

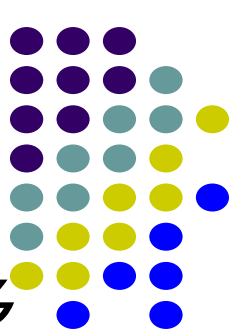


- 依赖关系定义最终应用程序中的每个文件与源文件之间的关系。
- 规则的写法是：先写目标名称，然后紧跟一个冒号，接着是空格或制表符，最后是用空格或制表符隔开的文件列表。如上例中，可以把依赖关系表示为：

```
myapp: main.o  2.o 3.o
main.o: main.c a.h
2.o:      2.c a.h b.h
3.o:      3.c b.h c.h
```

- 这组依赖关系形成层次结构，显示了源文件之间的关系。

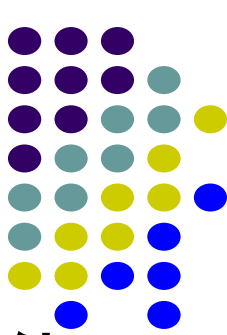
创建规则



- 上例2.c修改后，2.o用什么命令来创建。makefile有许多默认的规则，用户也可以指定创建规则，大多数规则都包含一个简单命令。如上例，可以创建一个makefile，命名为Makefile:

```
myapp: main.o  2.o 3.o
    gcc -o  myapp main.o 2.o 3.o
main.o: main.c a.h
    gcc -c  main.c
2.o:   2.c a.h b.h
    gcc  -c  2.c
3.o:    3.c b.h c.h
    gcc  -c  3.c
```

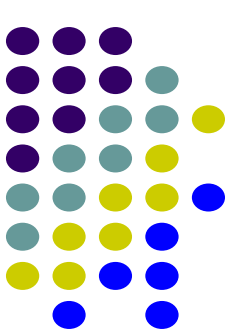
Makefile规则



- 前面的示例给了我们一个感兴认识，我们要写一个Makefile来告诉make命令如何编译和链接这几个文件。

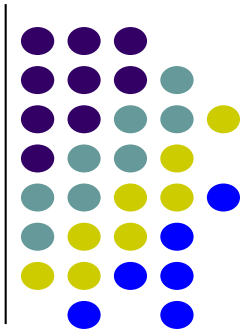
  这个Makefil需要包含如下规则：
  - 如果这个工程没有编译过，那么我们的所有C文件都要编译并被链接。
  - 如果这个工程的某几个C文件被修改，那么我们只编译被修改的C文件，并链接目标程序。
  - 如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的C文件，并链接目标程序。
- 只要Makefile写得够好，所有这一切只用一个make命令就可以完成，make命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

Make命令



- make是一个命令工具，是一个解释makefile文件中指令的命令工具，
- make的常用选项：
  - -f　　文件名　指定需要执行的makefile文件
  - -d　　Debug模式，输出有关文件和检测时间的详细信息
  - -n　　不执行命令，仅输出需要执行的命令
  - -p　　显示makefile中预设变量和隐含规则
  - -l dir　当包含其他makefile文件时，利用该选项指定搜索目录

Makefile的命名



- Makefile的默认文件名为GNUmakefile、makefile或Makefile，多数Linux程序员习惯使用第三种。
- 在默认情况下，make会执行当前目录下的Makefile文件。若当前目录下找不到相关的Makefile文件，则会出现错误：

  make: *** No targets specified and no makefile found. Stop.
- Makefile文件的命名可以为“Makefile"或"makefile"。
- 如果使用非标准命名的makefile，必须用命令参数“-f <name>” 或 “--file <name>”告诉make 读入name 作为makefile文件。

格式：　　make -f　youfile

　　或　　make --file youfile

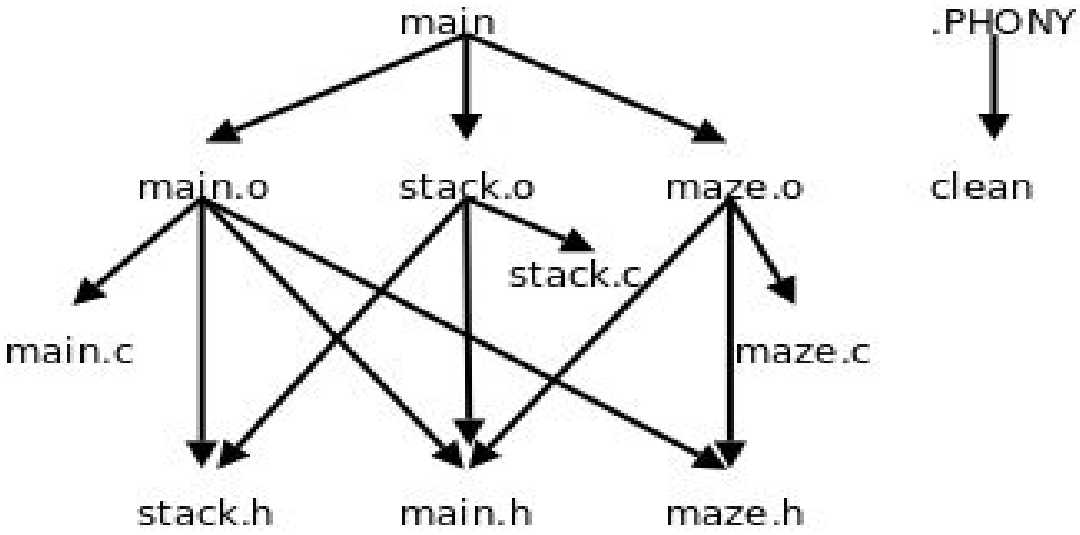
make的执行方式

- make处理makefile的过程，分为以下两个阶段：

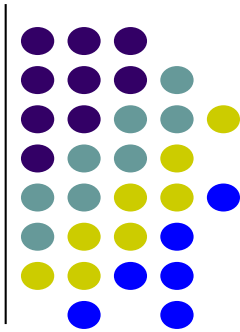
1. 从前到后读取所有规则，建立起一个完整的依赖关系图

2. 把缺省目标或者命令行指定的目标作为最终目标，

根据依赖关系图选择适当的规则执行。（缺省目标： makefile里第一个规则的目标。当命令make后面不指定目标时，make把makefile的缺省目标作为最终目标。例如直接执行make）



make的执行方式

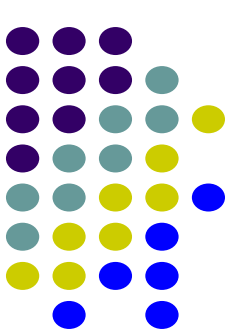


- make如何根据规则，生成最终目标？

1. 首先判断目标的依赖文件的是否都存在。若有依赖文件不存在，寻找以该文件为目标的规则，以生成该文件。
2. 若目标的依赖文件都存在，判断目标文件是否存在。若目标文件不存在，则执行规则里的命令。若目标文件存在，判断目标文件的修改时间是否比所有的依赖文件都新，若是，说明目标文件是最新的，不执行命令。否则执行规则里的命令。
3. make确保在依赖关系链上的目标都比依赖文件新。

- 这就是make的基本工作原理，也是最核心的规则。

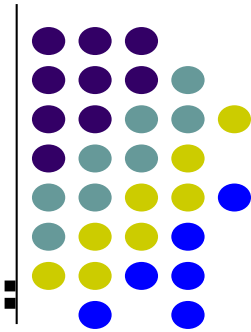
Makefile的组成



- Makefile里主要包含了五个部分的内容：

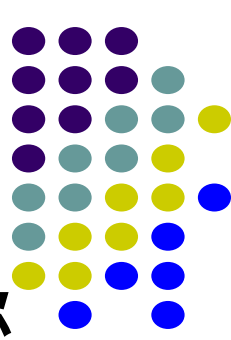
  显式规则、隐式规则、变量定义、文件指示、注释
- 显式规则：显式规则说明了如何生成一个或多个目标文件。这是由Makefile 的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。
- 隐式规则：由于make有自动推导的功能，所以隐式的规则可以让程序员比较简略地书写Makefile，这是由make所支持,例如，makefile发现.o文件，程序就会自动去找.c文件，并编译成.o文件。
- 变量定义:就是使用一个字符串代表一段文本串。在Makefile中可定义一系列的变量，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。

Makefile的组成



- 文件指示：文件指示与编译器的伪指令类似，包含：
  - 在一个Makefile中引用另一个Makefile，就像C语言中的include一样。被包含的文件会原模原样的放在当前文件的包含位置。
  - 决定是否忽略Makefile中的一部分，就像C语言中的预编译#if一样。
  - 定义一个变量；
- 注释： Makefile注释使用“#”，“#”字符后的内容都被作为是注释内容处理，直到行末。若Makefile需要用到“#”，则需要做转义“\#”。

伪目标

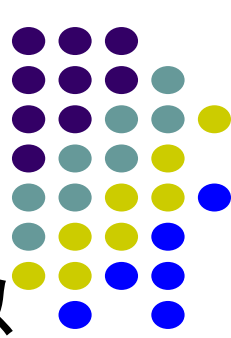


- Makefile中把没有任何依赖只有执行动作的目标称为“伪目标”(phony targets)
- 比如：clean。

```
clean :
    rm  *.o *.a
```

make默认把第一个目标认作default目标，clean一般都是放在文件的最后

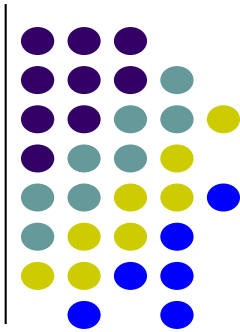
Makefile的内容



- Makefile文件主要含有一系列的规则，每条规则包含以下内容：
  - target：目标体。即make最终需要创建的对象。另外，目标也可以是一个make执行的动作名称。
  - dependency：依赖体。依赖体通常是编译目标体要依赖的一个或多个文件。
  - command：命令。为了从指定的依赖体创建出目标体所需执行的shell命令。一个规则可以有多个命令行，每一条命令占一行。

注意：每一个命令的第一个字符必须是制表符[Tab]，如果使用空格会产生问题，make会在执行过程中显示Missing Separator（缺少分隔符）并停止。

Makefile文件格式

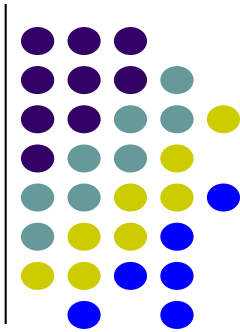


- 一个简单的Makefile文件的语法格式如下：

```
target… : dependency   [dependency […] ]
                COMMAND
                COMMAND
                […]
```

- target目标可能为如下三种情况：
  1. 执行文件
  2. 目标文件
  3. 伪目标

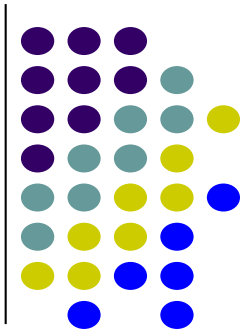
Makefile文件



例2，有以下的Makefile文件：

```
# 一个简单的Makefile的例子（以#开头的为注释行）
test：  prog.o code.o
        gcc prog.o code.o –o test
prog.o：prog.c prog.h code.h
        gcc  –c prog.c –o prog.o
code.o：code.c code.h
        gcc –c code.c –o code.o
clean：
        rm –f  *.o
```

Makefile文件

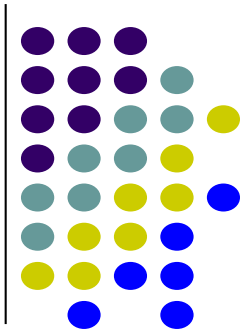


- 上面的Makefile文件中共定义了四个目标：

  test、prog.o、code.o、clean

- 目标从每行的最左边开始写，后面跟一个冒号（：），如果有与这个目标有依赖性的其他目标或文件，把它们列在冒号后面，并以空格隔开。
- 然后另起一行开始写实现这个目标的一组命令。在Makefile中，可使用续行号（\）将一个单独的命令行延续成几行。但要注意在续行号（\）后面不能跟任何字符（包括空格和键）。

Makefile文件



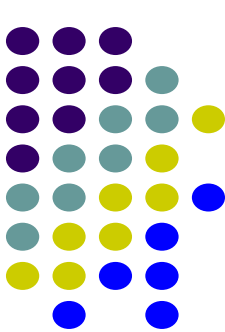
- 调用make命令可输入：

  # make target

  target是Makefile文件中定义的目标之一，如果省略target，make就将生成Makefile文件中定义的第一个目标。对于上面Makefile的例子，单独的一个“make”命令等价于：# make test

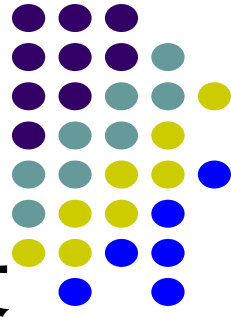
- 因为test是Makefile文件中定义的第一个目标，所以make首先将其读入，然后从第一行开始执行，把第一个目标test作为它的最终目标，后面的目标的更新都会影响到test的更新。第一条规则说明只要文件test的时间戳比文件prog.o或code.o中的任何一个旧，下一行的编译命令将会被执行。

Makefile文件



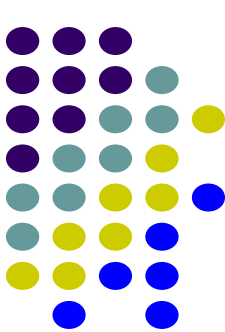
- 但是，在检查文件prog.o和code.o的时间戳之前，make会在下面的行中寻找以prog.o和code.o为目标的规则，在第三行中找到了关于prog.o的规则，该文件的依赖文件是prog.c、prog.h和code.h。
- 同样，make会在后面的规则行中继续查找这些依赖文件的规则，如果找不到，则开始检查这些依赖文件的时间戳，如果这些文件中任何一个的时间戳比prog.o的新，make将执行“gcc –c prog.c –o prog.o”命令，更新prog.o文件。

Makefile文件



- 同样方法，接下来对code.o做类似的检查，依赖文件是code.c和code.h。当make执行完所有这些套嵌的规则后，make将处理最顶层的test规则。如果关于prog.o和code.o的两个规则中的任何一个被执行，至少其中一个.o目标文件就会比test新，那么就要执行test规则中的命令，因此make去执行gcc命令将prog.o和code.o连接成目标文件test。
- make做的工作就是从下到上依照规则链执行目标文件的时间戳比依赖文件时间戳旧的规则，直到最顶层的规则。
- 在上面Makefile的例子中，还定义了一个目标clean，它是Makefile中常用的一种专用目标（伪目标），即删除所有的目标模块。

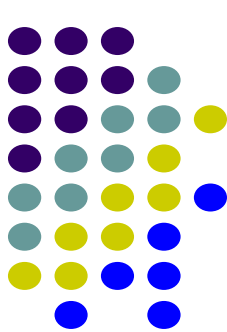
Makefile中的变量



- Makefile里的变量就像一个环境变量。事实上，环境变量在make中也被解释成make的变量。这些变量区分大小写，一般使用大写字母。通常可以从任何地方引用定义的变量，变量的主要作用如下：

  （1）保存文件名列表。在前例中，作为依赖文 件的一些目标文件名出现在可执行文件的规则中，而在这个规则的命令行里同样包含这些文件并传递给gcc做为命令参数。如果使用一个变量来保存所有的目标文件名，则可以方便地加入新的目标文件而且不易出错。

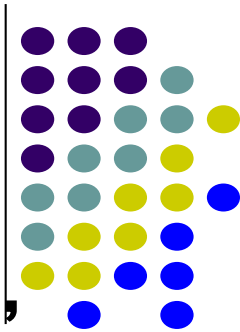
Makefile中的变量



（2）保存可执行命令名，如编译器。在不同的Linux系统中存在很多相似的编译器系统，这些编译器系统会有细微的差别，如果项目被用在一个非gcc的系统里，则必须将所有出现编译器名的地方改成用新的编译器名，如果使用一个变量来代替编译器名，那么只需要改变该变量的值。其他所有地方的命令名就都改变了。

（3）保存编译器的参数。在很多源代码编译时，gcc需要很长的参数选项，有时，所有的编译命令使用一组相同的选项，如果把这组选项定义为一个变量，把这个变量放在所有引用编译器的地方。当要改变选项的时候，只需改变一次这个变量的内容即可。

定义变量

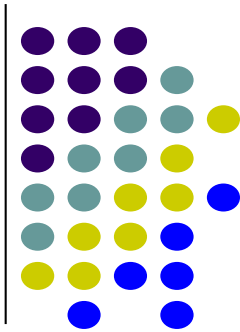


- Makefile中的变量是用一个文本串在Makefile中来定义，这个文本串就是变量的值。
- 定义变量：在一行的开始写下变量名，后面跟一个“=”号，等号右面设定这个变量的值。下面是定义变量的语法：

  VARNAME=string
- 引用时，把变量用括号括起来，并在前面加上$符号，就可以引用变量的值：

  ${VARNAME}
- make解释规则时，VARNAME在等式右端展开为定义它的字符串。变量一般都在Makefile的头部定义。如果变量的值发生变化，就只需要在一个地方修改，从而简化了Makefile的维护。

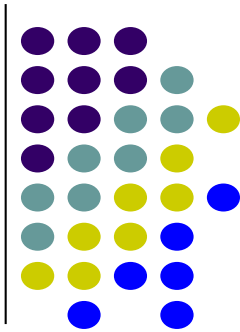
定义变量



- 现在利用变量把前面例2的Makefile重写一遍：

```
OBJS=prog.o code.o
CC=gcc
test：${ OBJS }
        ${ CC } –o test ${ OBJS }
prog.o：prog.c prog.h code.h
        ${ CC } –c prog.c –o prog.o
code.o：code.c code.h
        ${ CC } –c code.c –o code.o
clean：
        rm –f *.o
```

特殊变量



- 常用的特殊变量有：

  $@　表示目标文件的完整名称。

  $<　表示规则中的第一个依赖文件。

  $?　表示规则中所有比目标新的依赖文件，组成一个列表，以空格分隔。

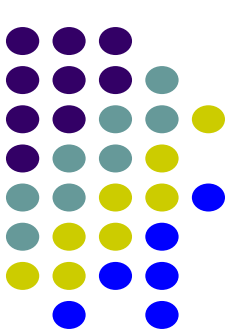
  $^　表示规则中的所有依赖文件，组成一个列表，以空格分隔。

- 例如：下面左边的规则，可以改写成右边的形式

```
file1: file1.o file2.o
        gcc  -o  file1  file1.o  file2.o
```

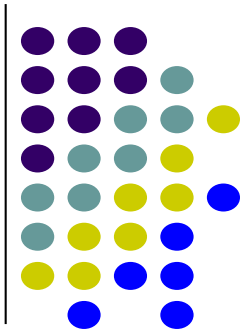
```
file1: file1.o file2.o
        gcc  -o  $@  $^
```

Makefile的隐含规则



- 在上面的例子中，几个产生目标文件的命令都是从“.c”的C语言源文件和相关文件通过编译产生“.o”目标文件，这也是一般的步骤。
- 实际上，make可以使工作更加自动化，make知道一些默认的动作，有一些称作隐含规则的内置的规则，这些规则告诉make当用户没有完整地给出某些命令的时候，应该怎样执行。
- 若目标是.o文件，那么他会自动的去寻找相应的.c文件， 并隐式的进行编译。例如，把生成prog.o和code.o的命令从规则中删除，make将会查找隐含规则，然后会找到并执行一个适当的命令。

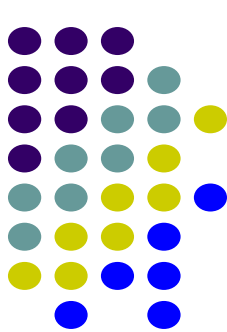
Makefile的隐含规则



- 在上面的例子中，利用隐含规则，可以简化为：

```
OBJS=prog.o code.o
CC=gcc
test：  ${ OBJS }
        ${ CC } –o $@ $^
prog.o：prog.c prog.h code.h
code.o：code.c code.h
clean：
        rm –f *.o
```

Makefile练习1



假设：/home/myprg/目录下有两个文件分别为hello.c和hello.h，文件hello.h在文件hello.c中被引用，要创建的目标文件为hello。

分析：

- 编译过程的target应该是

  hello

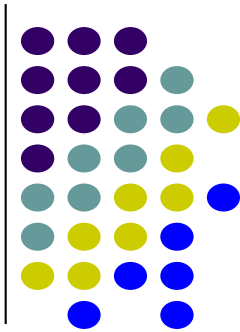
- 编译过程的dependency应该是文件

  hello.c和hello.h

- 编译过程从依赖体创建目标体所执行的COMMAND命令

  gcc -c hello.c -o hello

Makefile练习1



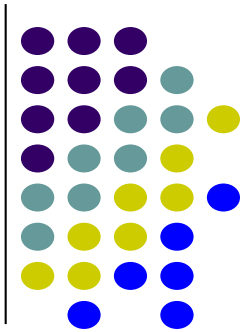
制作Makefile文件：

打开vi编辑器编写Makefile文件

[root@linux myprg]#vi Makefile

```
hello: hello.c hello.h
    gcc  hello.c -o hello
clean:
    rm  -fr hello.c hello.h
```

使用make命令

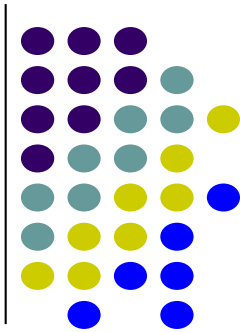


命令格式如下：

```
make [target]
```

执行make命令时，自动读入Makefile文件并执行对应target的COMMAND语句，并找到相应的依赖文件。示例如下：

```
[root@Linux myprg]#make hello
gcc hello.c -o hello
[root@Linux myprg]#ls
hello  hello.c  hello.h  Makefile
[root@Linux myprg]#make clean
rm -fr hello.c hello.h
[root@Linux myprg]#ls
hello  Makefile
```

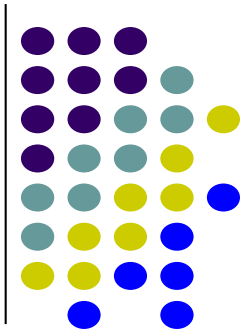


上例中

- make hello命令执行了目标体（hello）所对应的命令语句（gcc hello.c -o hello），并生成了该目标体（hello）。
- make clean命令则执行相应命令(rm -fr hello.c hello.h)删除了文件hello.c和hello.h。

执行make命令时自动检测相关文件的时间戳。如执行make hello命令，make首先检查目标体和依赖体之间的时间戳，如果hello.c或hello.h的时间戳比hello更新，则对应的COMMAND命令被执行，否则不被执行。

Makefile练习2



有三个源程序

- 程序fun1.c：

```
#include <stdio.h>
void fun1(){printf("this is fun1\n");}
```

- 程序fun2.c：

```
#include <stdio.h>
void fun2(){printf("this is fun2\n");}
```

- 程序main.c:

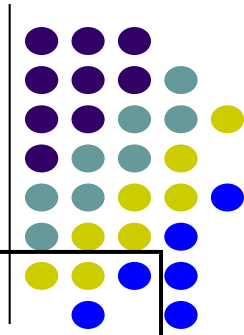
```
 #include <stdio.h>
void fun1();
void fun2();
int main()
{
    fun1();
    fun2();
    return 0;
}
```

如果使用gcc编译出应用程序test:

```
#gcc fun1.c fun2.c main.c -o test2
```

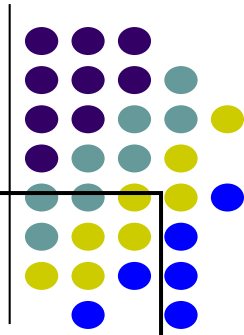
接下来，我们使用Makefile来写一个脚本。

Makefile练习2



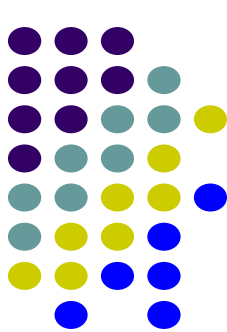
```
#sample makefile
test2: fun1.o fun2.o main.o
        gcc fun1.o fun2.o main.o -o  test2
fun1.o:fun1.c
        gcc -c fun1.c
fun2.o:fun2.c
        gcc -c fun2.c
main.o:main.c
        gcc -c main.c
```

Makefile定义变量



```
#sample makefile
CC=gcc
OBJS=fun1.o fun2.o main.o
EXEC=test2
all:$(OBJS)
        $(CC) $(OBJS) -o $(EXEC)
fun1.o:fun1.c
        $(CC) -c fun1.c
fun2.o:fun2.c
        $(CC) -c fun2.c
main.o:main.c
        $(CC) -c main.c
```

利用隐式规则



- 由于make有自动推导的功能，所以隐式的规则可以让程序员比较简略地书写Makefile。
- make在解释Makefile时，若目标是.o文件，那么他会自动的去寻找相应的.c文件， 并隐式的进行编译。
- 引用未定义的变量时，不会出错，但其值为空，即什么都没有。

Makefile隐式规则

```
#sample makefile
CC=gcc
OBJS=fun1.o fun2.o main.o
EXEC=test2

all:$(OBJS)
        $(CC) $(OBJS) -o $(EXEC)
fun1.o:
fun2.o:
main.o:
```

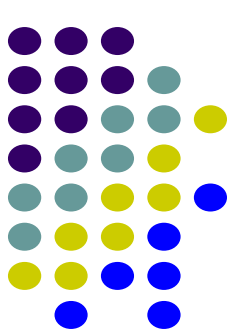
简化2： 使用隐式规则，目标文件为.o文件，make自动推导搜索.c文件，并编译。

Makefile隐式规则

```
#sample makefile
CC=gcc
OBJS=fun1.o fun2.o main.o
EXEC=test2


all:$(OBJS)
        $(CC) $(OBJS) -o $(EXEC)
```

简化3： 使用隐式规则，目标的依赖为三个.o文件，fun1.o, fun2.o, main.o， make自动推导，找到相应.c文件生成找到.o文件。

Makefile的变量替换



- 在Makefile中通常指定需要编译的.c文件，我们可以将多个.c文件名保存在变量SRCS中，再定义另一个变量OBJS使其保存.o文件名。
- 我们可以直接使用变量的替换方式，将SRCS中的“.c”字符串替换成“.o”，赋值OBJS变量，其格式为：

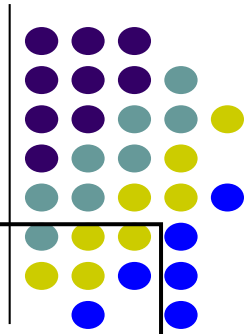
  foo = $(var:a=b)

  将var变量中的a替换成b，并返回给foo。
- 例：

  SRCS = fun1.c fun2.c main.c
  OBJS = $(SRCS：.c=.o)

  那么变量OBJS值为fun1.o fun2.o main.o

Makefile优化



```
#sample makefile
CC=gcc
SRCS=fun1.c fun2.c main.c

EXEC=test2

all:
        $(CC) $(SRCS) -o $(EXEC)
```