

Linux项目自动化构建工具-make/Makefile

- 会不会写makefile，从一个侧面说明了一个人是否具备完成大型工程的能力
- 一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作
- makefile带来的好处就是——“自动化编译”，一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。
- make是一个命令工具，是一个解释makefile中指令的命令工具，一般来说，大多数的IDE都有这个命令，比如：Delphi的make，Visual C++的nmake，Linux下GNU的make。可见，makefile都成为了一种在工程方面的编译方法。
- make是一条命令，makefile是一个文件，两个搭配使用，完成项目自动化构建。



The screenshot shows a code editor with three files open. The first file, test7.c, contains the main function. The second file, test7.h, contains the function declaration for show(). The third file, buffers, contains the header file for test7.h. Red annotations are present: '主函数' (main function) for test7.c, '函数声明' (function declaration) for test7.h, and '头文件' (header file) for buffers.

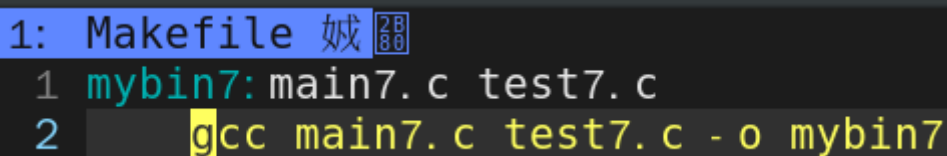
编译：

```
gcc main7.c test7.c -o mybin7
```

缺陷：当源文件有成百上千个时，gcc就很繁琐

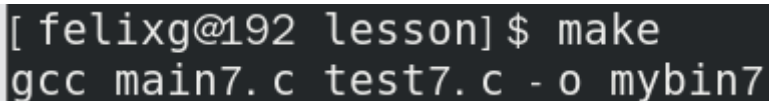
Makefile文件：

```
touch Makefile
```



The screenshot shows a terminal window with the contents of the Makefile. The first line is '1: Makefile 斌' and the second line is '1 mybin7: main7.c test7.c'. The third line is '2 gcc main7.c test7.c -o mybin7'.

make -- 编译



The screenshot shows a terminal window with the command 'make' being executed. The output is 'gcc main7.c test7.c -o mybin7'.

在Makefile中写入：(固定的套路)

删除文件就可以直接：make clean

Makefile:

编写Makefile，本质上是在编写依赖关系和依赖方法！！

1.依赖关系

- 上面的文件 `hello` ,它依赖 `hell.o`
- `hello.o` , 它依赖 `hello.s` `hello.s` , 它依赖 `hello.i`
- `hello.i` , 它依赖 `hello.c`

2.依赖方法:

- `gcc hello.* -option hello.*` ,就是与之对应的依赖关系

.PHONY: 可以理解为makefile的“关键字” , .PHONY: clean (伪目标) : 总是被执行的!

原理

make是如何工作的,在默认的方式下,也就是我们只输入make命令。那么,

1. make会在当前目录下找名字叫“Makefile”或“makefile”的文件。
2. 如果找到,它会找文件中的第一个目标文件(target),在上面的例子中,他会找到“hello”这个文件,并把这个文件作为最终的目标文件。
3. 如果hello文件不存在,或是hello所依赖的后面的hello.o文件的文件修改时间要比hello这个文件新(可以用touch 测试),那么,他就会执行后面所定义的命令来生成hello这个文件。
4. 如果hello所依赖的hello.o文件不存在,那么make会在当前文件中找目标为hello.o文件的依赖性,如果找到则再根据那一个规则生成hello.o文件。(这有点像一个堆栈的过程)
5. 当然,你的C文件和H文件是存在的啦,于是make会生成 hello.o 文件,然后再用 hello.o 文件声明make的终极任务,也就是执行文件hello了。
6. 这就是整个make的依赖性,make会一层又一层地去找文件的依赖关系,直到最终编译出第一个目标文件。
7. 在找寻的过程中,如果出现错误,比如最后被依赖的文件找不到,那么make就会直接退出,并报错,而对于所定义的命令的错误,或是编译不成功,make根本不理。
8. make只管文件的依赖性,即,如果在我找了依赖关系之后,冒号后面的文件还是不在,那么对不起,我就不工作啦。

实际的工作过程:

最终版本:

- `^`: 依赖文件列表
 - `@`: 目标文件
 - `.c`: 当前目录下的所有的.c文件展开
 - `.o`: 对应的.c形成的.o
- `<: .c`所代表的源文件,一个一个的拿出来,用gcc进行编译,形成同名的.o

项目清理

- 工程是需要被清理的
- 像clean这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要make执行。即命令——“make clean”，以此来清除所有的目标文件，以便重编译。
- 但是一般我们这种clean的目标文件，我们将它设置为伪目标,用 .PHONY 修饰,伪目标的特性是，总是被执行的。
- 可以将我们的 hello 目标文件声明成伪目标，测试一下。

Linux调试器-gdb使用

调试：

1.发现问题

2.定位问题 (gdb)

3.分析问题

4.解决问题

1.背景

gdb：默认是以release方式进行发布的！不可以被调试的

release：是一般软件进行开发，并交付给用户的模式

debug：生成的软件内部，包含了调试信息

debug生成的程序，体积上一定大于release

Linux默认是release

如何改为debug？

在makefile文件加入-g选项：

此时就可以使用gdb进行调试了

```
[gaofan@192 Add]$ gdb add_d
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-100.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
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/felixg/code/lesson/Add/add_d... done.
(gdb)
```

2. 开始使用

gdb binFile 退出： ctrl + d 或 quit 调试命令：

• list / 1 行号:

显示binFile源代码，接着上次的位置往下列，每次列10行。 [重要]

```
(gdb) l 1
1      #include "add.h"
2
3
4      int Add(int num)
5      {
6          int i=0;
7          int sum=0;
8          for(; i<=num; i++)
9          {
10
```

```
(gdb) l 9
4      int Add(int num)
5      {
6          int i=0;
7          int sum=0;
8          for(; i<=num; i++)
9          {
10
11              sum+=i;
12          }
13      return sum;
```

```
(gdb) l
14      }
15      int main()
16      {
17          printf("process begin running...! \n");
18          ;
19          int result=Add(100);
20          printf("result:%d\n", result);
21          printf("process end running...! \n");
22          return 0;
23      }
```

• list / 1 函数名: 列出某个函数的源代码。

- break(b) 行号:

在某一行设置断点 [重要]

- break 函数名: 在某个函数开头设置断点

```
(gdb) b 17
Breakpoint 1 at 0x4005b3: file add.c, line 17.
(gdb)
```

- run(或r):
- r或run: 运行程序。 [重要] VS中的F5

从开始连续而非单步执行程序, 运行到断点处; 如果没有断点, r直接运行程序到结束; 也可以从头开始

```
(gdb) r
Starting program: /home/felixg/code/lesson/Add/add_d

Breakpoint 1, main () at add.c:17
17      printf("process begin running...!\n");
Missing separate debuginfos, use: debuginfo-install glibc-2.17-196.el7.x86_64
(gdb)
```

不中断点全部运行

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/felixg/code/lesson/Add/add_d
process begin running...!
result: 5050
process end running...!
[Inferior 1 (process 5348) exited normally]
```

- n 或 next:

单条执行。 [重要] 相当于VS中F10

```
17      printf("process begin running...!\n");
Missing separate debuginfos, use: debuginfo-install glibc-2.17-196.el7.x86_64
(gdb) n
process begin running...!
18      int result=Add(100);
```

- s或step:

进入函数调用 [重要] VS中的F11

- display 变量名:

跟踪查看一个变量, 每次停下来都显示它的值 VS中的监视

也可以查看地址:

- undisplay:

取消对先前设置的那些变量的跟踪

- until X行号:

跳至X行（会自动跳过空行）

- finish:

执行到当前函数返回，然后挺下来等待命令

- info(或i) breakpoints:

- info break : 查看断点信息。

查看当前设置了哪些断点

```
(gdb) info b
Num      Type           Disp Enb Address            What
1        breakpoint      keep y   0x00000000004005b3 in main at add.c:17
          breakpoint already hit 1 time
(gdb)
```

- delete breakpoints n:

删除序号为n（而不是行号）的断点 [重要]

```
(gdb) info b
Num      Type           Disp Enb Address            What
1        breakpoint      keep y   0x00000000004005b3 in main at add.c:17
          breakpoint already hit 1 time
(gdb) d 17
No breakpoint number 17.
(gdb) d 1
(gdb) info b
No breakpoints or watchpoints.
(gdb)
```

- delete breakpoints: 删除所有断点
- disable breakpoints: 禁用断点
- enable breakpoints: 启用断点
- print(p): 打印表达式的值，通过表达式可以修改变量的值或者调用函数
- p 变量: 打印变量值。 [重要]
- set var: 修改变量的值

- continue(或c):

从当前位置开始连续而非单步执行程序 [重要] 从一个断点跳转至另一个断点VS中的F5

- breaktrace(或bt): 查看各级函数调用及参数

- `info (i) locals`: 查看当前栈帧局部变量的值

- `quit`:

退出gdb