更多嵌入式 Linux 学习资料，请关注：一口 Linux 回复关键字:1024

0. 前言

粉丝留言，想知道如何使用 Makefile 给多个文件和多级目录建立一个工程，必须安排！

关于 Makefile 的入门参考文章，可以先看这篇文章：

《Makefile 入门教程》

为了让大家有个更加直观的感受，一口君将之前写的一个小项目，本篇在该项目基础上进行修改。

该项目详细设计和代码，见下文：

《从 0 写一个《电话号码管理系统》的 C 入门项目【适合初学者】》

一、文件

好了，开始吧!

我们将该项目的所有功能函数放到以该函数名命名的 c 文件，同时放到对应名称的子目录中。

比如函数 allfree()，存放到 allfree/allfree.c 中

最终目录结构如下图所示：

```
peng@ubuntu:/mnt/hgfs/code/phone$ tree .
.
├── allfree
│   ├── allfree.c
│   └── Makefile
├── create
│   ├── create.c
│   └── Makefile
```

```
├── delete
│   ├── delete.c
│   └── Makefile
├── display
│   ├── display.c
│   └── Makefile
├── include
│   ├── Makefile
│   └── phone.h
├── init
│   ├── init.c
│   └── Makefile
├── login
│   ├── login.c
│   └── Makefile
├── main
│   ├── main.c
│   └── Makefile
├── Makefile
├── menu
│   ├── Makefile
│   └── menu.c
├── scripts
│   └── Makefile
└── search
    ├── Makefile
    └── search.c

11 directories, 22 files
```

直接看下编译结果吧：

```
peng@ubuntu:/mnt/hgfs/code/phone$ make
make[1]: Entering directory '/mnt/hgfs/code/phone/allfree'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/allfree'
make[1]: Entering directory '/mnt/hgfs/code/phone/create'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/create'
make[1]: Entering directory '/mnt/hgfs/code/phone/delete'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/delete'
make[1]: Entering directory '/mnt/hgfs/code/phone/display'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/display'
make[1]: Entering directory '/mnt/hgfs/code/phone/init'
```

```
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/init'
make[1]: Entering directory '/mnt/hgfs/code/phone/login'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/login'
make[1]: Entering directory '/mnt/hgfs/code/phone/menu'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/menu'
make[1]: Entering directory '/mnt/hgfs/code/phone/search'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/search'
make[1]: Entering directory '/mnt/hgfs/code/phone/main'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/mnt/hgfs/code/phone/main'
gcc -Wall -O3 -
o phone allfree/*.o create/*.o delete/*.o display/*.o init/*.o login/*.o menu/*.o s
earch/*.o main/*.o -lpthread
phone make done!
```

运行结果如下：

```
peng@ubuntu:/mnt/hgfs/code/phone$ ./phone
+=====================================================================+
|                                                                     |
|                欢迎来到一口Linux的通讯录                            |
|                                                                     |
|                        温馨提示：初始密码(yikoupeng)                |
|                        关注公众号：一口Linux                        |
+=====================================================================+
please input your password:
```

二、Makefile 常用基础知识点

[0] 符号'@' '$' '$$' '-' '-n '的说明

1. '@'
   通常 makefile 会将其执行的命令行在执行前输出到屏幕上。如果将'@'添加到命令行前，这个命令将不被 make 回显出来。例如：

```
@echo  --compiling module----;  // 屏幕输出  --compiling module----
echo  --compiling module----;  // 没有@ 屏幕输出 echo  --compiling module----
```

2. ' - '
   通常删除，创建文件如果碰到文件不存在或者已经创建，那么希望忽略掉这个错误，继续执行，就可以在命令前面添加 -，

```
-rm dir;
-mkdir aaadir;
```

3. ' $ '美元符号$，主要扩展打开 makefile 中定义的变量
4. ' $$ '$$ 符号主要扩展打开 makefile 中定义的 shell 变量

[1] wildcard

说明: 列出当前目录下所有符合模式“ PATTERN”格式的文件名,并且以空格分开。“ PATTERN”使用 shell 可识别的通配符，包括“ ?”（单字符）、“ *”（多字符）等。示例：

```
$(wildcard *.c)
```

返回值为当前目录下所有.c 源文件列表。

[2] patsubst

说明：把字串“ x.c.c bar.c”中以.c 结尾的单词替换成以.o 结尾的字符。示例：

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

函数的返回结果 是

```
x.c.o bar.o
```

[3] notdir

说明：去除文件名中的路径信息 示例：

```
SRC = ( notdir ./src/a.c )
```

去除文件 a . c 的路径信息 ， 使用 (notdir ./src/a.c) 去除文件 a.c 的路径信息，使用 (notdir./src/a.c)去除文件 a.c 的路径信息，使用(SRC)得到的是不带路径的文件名称，即 a.c。

[4] 包含头文件路径

使用-I+头文件路径的方式可以指定编译器的头文件的路径 示例：

```
INCLUDES = -I./inc
$(CC) -c $(INCLUDES) $(SRC)
```

[5] addsuffix

函数名称：加后缀函数—addsuffix。语法：

```
$(addsuffix SUFFIX,NAMES…)
```

函数功能：为“NAMES…”中的每一个文件名添加后缀“SUFFIX”。参数“NAMES…”为空格分割的文件名序列，将“SUFFIX”追加到此序列的每一个文件名 的末尾。返回值：以单空格分割的添加了后缀“SUFFIX”的文件名序列。函数说明：示例：

```
$(addsuffix .c,foo bar)
```

返回值为

```
foo.c bar.c
```

[6] 包含另外一个文件：include

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

```
include file.dep
```

即把 file.dep 文件在当前 Makefile 文件中展开，亦即把 file.dep 文件的内容包含进当前 Makefile 文件

在 include 前面可以有一些空字符，但是绝不能是[Tab]键开始。

[7] foreach

foreach 函数和别的函数非常的不一样。因为这个函数是用来做循环用的 语法是：

```
$(foreach <var>,<list>,<text> )
```

这个函数的意思是，把参数中的单词逐一取出放到参数*所指定的变量中，然后再执行所包含的表达式。*

每一次会返回一个字符串，循环过程中，的所返回的每个字符串会以空格分隔，最后当整个循环结束时，所返回的每个字符串所组成的整个字符串（以空格分隔）将会是 foreach 函数的返回值。

所以，*最好是一个变量名，可以是一个表达式，而中一般会使用这个参数来依次枚举中的单词。*

举例:

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

上面的例子中，`$(name)`中的单词会被挨个取出，并存到变量“n”中，“`$(n).o`”每次根据“`$(n)`”计算出一个值，这些值以空格分隔，最后作为 foreach 函数的返回，所以，`$(files)`的值是“a.o b.o c.o d.o”。

注意，foreach 中的*参数是一个临时的局部变量，foreach 函数执行完后，参数的变量将不在作用，其作用域只在 foreach 函数当中。*

[8] call

“ call”函数是唯一一个可以创建定制化参数函数的引用函数。使用这个函数可以实现对用户自己定义函数引用。我们可以将一个变量定义为一个复杂的表达式，用“ call”函数根据不同的参数对它进行展开来获得不同的结果。

函数语法：

```
$(call variable,param1,param2,...)
```

函数功能：在执行时，将它的参数“ param”依次赋值给临时变量“ `$(1)`”、“ **`$(2)`**” call 函数对参数的数目没有限制，也可以没有参数值，没有参数值的“ call”没有任何实际存在的意义。执行时变量“ variable”被展开为在函数上下文有效的临时变量，变量定义中的“ `$(1)`”作为第一个参数，并将函数参数值中的第一个参数

赋值给它；变量中的“$(2)”一样被赋值为函数的第二个参数值；依此类推（变量**$(0)**代表变量“variable”本身）。之后对变量“variable”表达式的计算值。

返回值：参数值“param”依次替换“$(1)”、“$(2)”..... 之后变量“variable”定义的表达式的计算值。

函数说明：

1. 函数中“variable”是一个变量名，而不是变量引用。因此，通常“call”函数中的“variable”中不包含“$”（当然，除非此变量名是一个计算的变量名）。
2. 当变量“variable”是一个 make 内嵌的函数名时（如“if”、“foreach”、“strip”等），对“param”参数的使用需要注意，因为不合适或者不正确的参数将会导致函数的返回值难以预料。
3. 函数中多个“param”之间使用逗号分割。
4. 变量“variable”在定义时不能定义为直接展开式！只能定义为递归展开式。

函数示例：

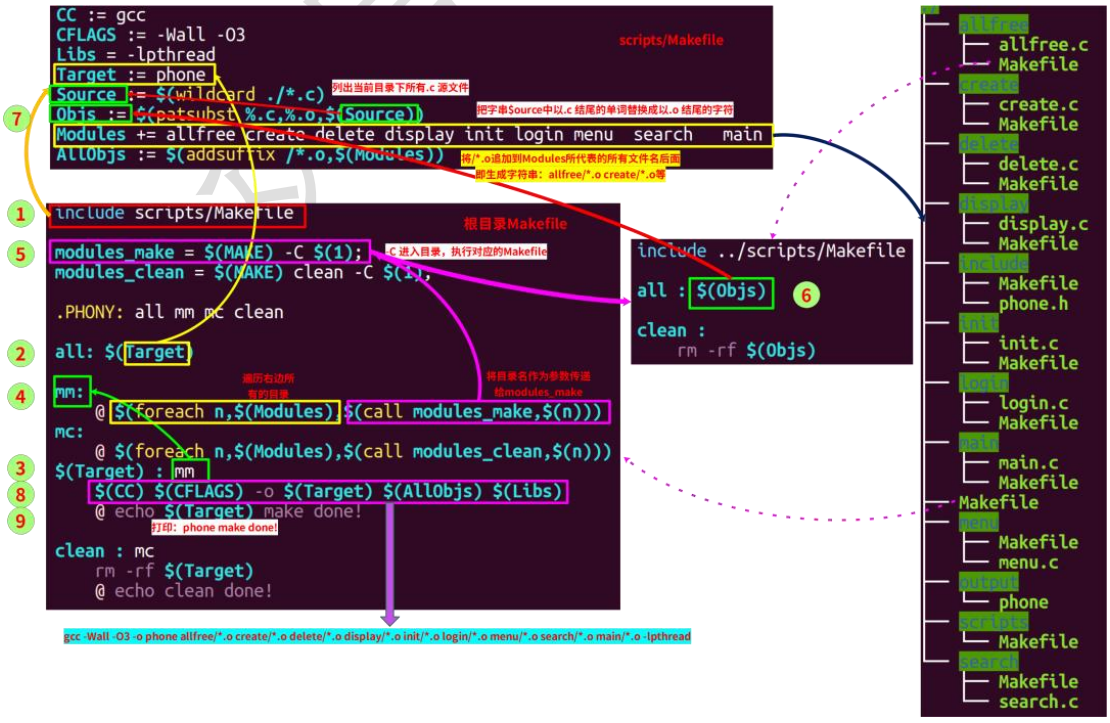
```
reverse = $(2)$(1)
foo = $(call reverse,a,b)
all:
 @echo "foo=$(foo)"
```

执行结果：

```
foo=ba
```

即 a 替代了替代了(2)

三、编译详细说明



我们在根目录下执行 make 命令后，详细步骤如下：

1. include scripts/Makefile ： 将文件替换到当前位置，
2. 使用默认的目标 all，该目标依赖于$(Target)$(Target) 在 scripts/Makefile 中定义了，即 phone
3. 而$(Target)依赖于 mm
4. mm 这个目标会执行

```
@ $(foreach n,$(Modules),$(call modules_make,$(n)))
```

Modules 是所有的目录名字集合， foreach 会遍历字符串$(Modules)中每个词语， 每个词语会赋值给 n， 同时执行语句：

```
call modules_make,$(n)
```

5. modules_make 被$(MAKE) -C $(1)所替代，

$(MAKE) 有默认的名字 make -C：进入子目录执行 make$(1) ：是步骤 4 中$(n)，即每一个目录名字

最终步骤 4 的语句就是进入到每一个目录下，执行每一个目录下的 Makefile

6. 进入某一个子目录下，执行 Makefile 默认目标是 all，依赖 Objs

```
Objs := $(patsubst %.c,%.o,$(Source))
```

patsubst 把字串$ource 中以.c 结尾的单词替换成以.o 结尾的字符 而

```
Source := $(wildcard ./*.c)
```

wildcard 会列举出当前目录下所有的.c 文件

所以第 6 步最终就是将子目录下的所有的.c 文件，编译生成对应文件名的.o 文件

8.

```
$(CC) $(CFLAGS) -o $(Target) $(AllObjs) $(Libs)
```

这几个变量都在文件 scripts/Makefile 中定义$(CC) ： 替换成 gcc，制定编译器 $(CFLAGS) ： 替换成-Wall -O3，即编译时的优化等级-o $(Target)：生成可执行程序 phone$(AllObjs) ：

```
AllObjs := $(addsuffix /*.o,$(Modules))
```

addsuffix 会将 /*.o 追加到$(Modules)中所有的词语后面，也就是我们之前在子目录下编译生成的所有的.o 文件$(Libs) ：替换为-lpthread，即所需要的动态库

大家可以根据这个步骤，来分析一下执行 make clean 时，执行步骤

完整的实例程序公众号后台回复：**电话号码管理**

《电话号码管理-makefile 版.rar》