

Makefile(上)

主要内容:

(http://v.youku.com/v_show/id_XMzg5OTI1MjAw.html?from=y1.7-2)

- 1.make 和 Makefile 的介绍
- 2.Makefile 基本规则
- 3.简单的 Makefile 编写
- 4.Make 自动化变量
- 5.Makefile 编译多个可执行文件

(一) 首先来看一下什么是 make 工具?

利用 make 工具可以自动完成编译工作。这些工作包括: 如果仅修改某几个源文件, 则只重新编译这几个源文件; 如果头文件被更改了, 则重新编译所有包含该头文件的源文件。利用这种自动编译可大大的简化开发工作, 避免不必要的重新编译。(PS: (1) 如果一个工程包含多个.c 文件, 常规的做法是对每个.c 文件进行编译。而且我们需要敲入好几条命令, 而且我们可能都忘记了那些文件做了修改, 这样我们就需要将所有的文件都编译一遍。相反, 如果我们使用 make 工具的话, 我们只需要编写一个脚本, 这个脚本就是 makefile, 将一些编译规则放到 makefile 里面, 我们只需要调一调 make 工具就可以完成整个工程的自动编译。这样的话, 大大的简化了开发工作。并且可以避免不必要的重新编译。因为它只编译更改过的文件。(2) 编译是将多个.c 文件生成.o 文件, 将.o 文件链接成可执行文件。)

(二) 接下来我们看一下 makefile。

Make 工具通过一个称为 Makefile 的文件来完成自动化的编译工作。Makefile 文件描述了整个工程的编译和连接的规则。

(三) Makefile 的基本规则。

①TARGETDEPENDENCIES ...
COMMAND ...

(这就是 Makefile 的基本规则, **TARGET** 是目标, 可以有一个文件, 也可以有多个文件, 通常情况下我们只有一个文件。**DEPENDENCIES** 为依赖列表, 可以有一个或多个文件组成, 当然也可以有零个文件, 也就是没有依赖文件。**COMMAND** 这是命令)

②目标 (**TARGET**) 程序产生的文件, 如可执行文件和目标文件, 也可以是其他一些中间过程要生成的文件, 比如说.s 文件等等; 目标也可以是要执行的动作, 如 clean, 也称为伪目标, 不是我们要生成的文

件。

③**依赖列表 (DEPENDENCIES)**, 依赖列表如果发生了改变, 就会通过前面①的命令生成目标。依赖列表是目标依赖的文件列表, 那么一个目标通常情况下都会依赖多个文件。

④**命令 (COMMAND)** 是 make 执行的动作 (命令是 shell 命令或者可在 shell 下执行的程序), 注意每个命令行的起始字符必须为 TAB 字符。

⑤ 如果 **DEPENDENCIES** 中的一个或者多个文件更新的话, **COMMAND** 命令就会执行, 命令执行的结果就是生成 **TARGET** 这就是 **Makefile** 最核心的内容。

(四) 最简单的 Makefile 的例子

```
main:main.o add.o sub.o
    gcc main.o add.o sub.o -o main
main.o:main.c add.h sub.h
    gcc -c main.c -o main.o
add.o:add.c add.h
    gcc -c add.c -o add.o
sub.o:sub.c sub.h
    gcc -c sub.c -o sub.o
clean:
    rm -f main main.o add.o sub.o
```

这个工程有三个模块 (**main.c,add.c,sub.c**) :

A terminal window screenshot showing a user navigating to a directory and creating files. The terminal output is as follows:

```
whd@whd-Lenovo:~/test$ cd
whd@whd-Lenovo:~$ ls
fftw-3.3.3  libpepflashplayer.so  PureGaussianBlur  Test_time.c  图片
GaussianBlur  libpng-1.5.4          README            Test_time.c~  文档
GDBNOTES     manifest.json          sources.list       公共的        下载
git          MSR_original          test              模板          音乐
LGPL         PureGaussian          Test_time         视频          桌面
whd@whd-Lenovo:~$ cd GDBNOTES/
whd@whd-Lenovo:~/GDBNOTES$ ls
gdb调试入门.wps
whd@whd-Lenovo:~/GDBNOTES$ touch main.c add.c sub.c add.h sub.h
whd@whd-Lenovo:~/GDBNOTES$ ls
add.c  add.h  gdb调试入门.wps  main.c  sub.c  sub.h
```

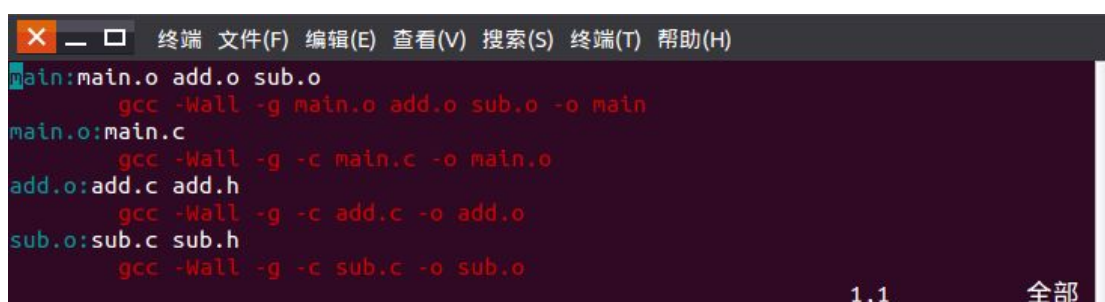
Touch 命令可以创建不存在的文件, touch 命令参数可更改文档或目录的日期时间, 包括存取时间和更改时间。

一个工程要有一个 main 函数, 别写 main()函数如下, 输入 vi main,c 命令, 输入 i, 进入编辑模式, 填写如下的内容:

```
int main(void)
{
    return 0;
}
```

Ctrl + C 组合键保存文件，**: wq**退出编辑模式。


接下来编写一个最简单的 Makefile 文件。



```
main:main.o add.o sub.o
    gcc -Wall -g main.o add.o sub.o -o main
main.o:main.c
    gcc -Wall -g -c main.c -o main.o
add.o:add.c add.h
    gcc -Wall -g -c add.c -o add.o
sub.o:sub.c sub.h
    gcc -Wall -g -c sub.c -o sub.o
```

注意：gcc 前面的空格是 TAB 键的作用。

编辑完 Makefile 文件之后,用 make 命令编译一下,可以看到我们并没有别写很多有效的代码，但是它已经可以生成 main 可执行文件。而且我们只需要敲入一条命令—make 命令即可完成。这非常的方便。为什么我们能够完成这样一件事情呢？它的执行顺序是什么样的呢？：



```
whd@whd-Lenovo:~/GDBNOTES$ vi Makefile
whd@whd-Lenovo:~/GDBNOTES$ ls
add.c  add.h  gdb调试入门.wps  main.c  Makefile  sub.c  sub.h
whd@whd-Lenovo:~/GDBNOTES$ make
gcc -Wall -g -c main.c -o main.o
gcc -Wall -g -c add.c -o add.o
gcc -Wall -g -c sub.c -o sub.o
gcc -Wall -g main.o add.o sub.o -o main
whd@whd-Lenovo:~/GDBNOTES$ ls
add.c  add.o  main  main.o  sub.c  sub.o
add.h  gdb调试入门.wps  main.c  Makefile  sub.h
```

我们看一下，我们敲入了 make 命令，它其实执行了这么多条语句：

```
whd@whd-Lenovo:~/GDBNOTES$ make
gcc -Wall -g -c main.c -o main.o
gcc -Wall -g -c add.c -o add.o
gcc -Wall -g -c sub.c -o sub.o
gcc -Wall -g main.o add.o sub.o -o main
```

首先看一下 Makefile 的内容：

(1) 敲入 make 命令之后，首先生成 main 这个目标，它发现第一个目标需要依赖于三个.o 文件，而这三个.o 文件并没有存在，那么它就会去先生成这些.o 文件，先生成 main.c 文件，用：

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

来生成，用下面的命令来生成 add.o

```
gcc -Wall -g -c add.c -o add.o
```

用下面的命令来生成 sub.o

```
gcc -Wall -g -c sub.c -o sub.o
```

我们可以看一下 make 命令之后显示的代码执行顺序，可以看到也是按照 main.o add.o sub.o 的顺序生成的，和分析的是一致的。三个生成完毕后，就可以组合起来生成 main 了。这就是 gcc 的编译的一个过程。

如果编译完之后，如果文件都没有改动的话，这时敲入 make 命令，这时是不会编译的，这是因为文件都没有变化。

```
whd@whd-Lenovo:~/GDBNOTES$ make
make: 'main' is up to date.
whd@whd-Lenovo:~/GDBNOTES$
```

那么我们更改了文件呢？比如用 `touch sub.c` 命令强制修改文件的修改时间。

```
whd@whd-Lenovo:~/GDBNOTES$ touch sub.c
whd@whd-Lenovo:~/GDBNOTES$ make
gcc -Wall -g -c sub.c -o sub.o
gcc -Wall -g main.o add.o sub.o -o main
whd@whd-Lenovo:~/GDBNOTES$ ls
add.c  add.o          main    main.o    sub.c  sub.o
add.h  gdb调试入门.wps main.c  Makefile  sub.h
whd@whd-Lenovo:~/GDBNOTES$
```

我们根据 Makefile 的规则，当文件 sub.c 更新的时候，它会重新生成 sub.o,因为 sub.o 依赖于 sub.c 和 sub.h,只要其中一个文件发生了变化，就要重新编译生成 sub.o。那么 sub.o 发生了变化，那么就说明 main.o 需要重新编译生成。

```
whd@whd-Lenovo:~/GDBNOTES$ touch sub.c
whd@whd-Lenovo:~/GDBNOTES$ make
gcc -Wall -g -c sub.c -o sub.o
gcc -Wall -g main.o add.o sub.o -o main
```

可以看一下文件的执行过程，果然时这样的：由于 sub.c 的改变需要重新生成 sub.o,由于 sub.o 的版本要新于目标 main 的版本，所以需要重新编译生成 main。并不是 Makefile 中的所有代码都由重新执行了一遍。这就是 Makefile 的基本规则，它是通过时间来进行编译的，一旦依赖列表中的文件的修改时间比目标文件更新，那么就会按照命令重新生成目标。

接下来我们可以看一下：

```
clean:
    rm -f main main.o add.o sub.o
```

clean 是伪目标，伪目标并不是我们真正要生成的生成的文件。伪目标并没有依赖列表，只是用来执行后面的操作。它只是用来删除文件：


```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
whd@whd-Lenovo:~/GDBNOTES$ vim Makefile
whd@whd-Lenovo:~/GDBNOTES$ make clean
rm -f main main.o add.o sub.o
whd@whd-Lenovo:~/GDBNOTES$ ls
add.c add.h gdb调试入门.wps main.c Makefile sub.c sub.h
whd@whd-Lenovo:~/GDBNOTES$
```

将伪目标代码添加到 Makefile 文件中，然后执行 **make clean** 命令就可以很方便的将文件进行删除。

另外我们如果只想编译生成某个目标，如 main.o 文件（也就是说只执行 Makefile 中的某个命令），我们只需要执行：

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
whd@whd-Lenovo:~/GDBNOTES$ make main.o
gcc -Wall -g -c main.c -o main.o
whd@whd-Lenovo:~/GDBNOTES$ make add.o
gcc -Wall -g -c add.c -o add.o
whd@whd-Lenovo:~/GDBNOTES$ make sub.o
gcc -Wall -g -c sub.c -o sub.o
whd@whd-Lenovo:~/GDBNOTES$ make main
gcc -Wall -g main.o add.o sub.o -o main
whd@whd-Lenovo:~/GDBNOTES$
```

默认的，终端中只输入 **make** 命令是生成第一个目标。那么伪目标通常情况下，我们会使用这样一个特殊的目标来表示，

.PHONY:clean

我们先用#把该行代码注释掉，然后在该目录下新建一个名为 **clean** 文件（通过 **touch clean** 建立）。运行 **make clean** 命令，会提醒如下提示：

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
whd@whd-Lenovo:~/GDBNOTES$ vi Makefile
whd@whd-Lenovo:~/GDBNOTES$ make clean
make: 'clean' is up to date.
whd@whd-Lenovo:~/GDBNOTES$
```

因为我们没有显式指定 **clean** 为一个伪目标，加上前面的语句，就可以将 **clean** 设置为一个伪目标，运行的时候，就能执行删除操作：

```
make: 'clean' is up to date.
whd@whd-Lenovo:~/GDBNOTES$ vi Makefile
whd@whd-Lenovo:~/GDBNOTES$ make clean
rm -f main main.o add.o sub.o
whd@whd-Lenovo:~/GDBNOTES$
```

然后把 clean 文件删掉。

```
whd@whd-Lenovo:~/GDBNOTES$ vi Makefile
whd@whd-Lenovo:~/GDBNOTES$ make clean
rm -f main main.o add.o sub.o
whd@whd-Lenovo:~/GDBNOTES$ rm clean
whd@whd-Lenovo:~/GDBNOTES$
```

进一步用过观察 Makefile 里面的信息，我们发现好几处都是公用相同的信息，那么我们能不能用变量来定义这些重复的内容，然后再表示呢？接下来我们将讲一些 Makefile 自定义的一些变量，以及 Makefile 内部的一些自动化变量。首先我们来看一下：

Makefile 的自动化变量：

选项名	作用
<code>\$@</code>	规则的目标文件名
<code>\$<</code>	规则的第一依赖文件
<code>\$^</code>	规则的所有依赖文件列表

比如说：

```
PHONY:clean
main:main.o add.o sub.o
    gcc -Wall -g main.o add.o sub.o -o main
main.o:main.c
    gcc -Wall -g -c main.c -o main.o
add.o:add.c add.h
    gcc -Wall -g -c add.c -o add.o
sub.o:sub.c sub.h
    gcc -Wall -g -c sub.c -o sub.o
clean:
    rm -f main main.o add.o sub.o
```

如果上面的第二行是一条规则的话，那么 main 就是规则的目标文件

名,main.o 就是规则第一个依赖文件,该行的 main.o add.o sub.o 文件就是规则的依赖列表,下面的以此类推。接下来我们实用一下他们:首先拷贝一份 Makefile,加上一些变量, OBJECTS 是定义一些变量,

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
.PHONY:clean
OBJECTS=main.o add.o sub.o
main:$(OBJECTS)
    gcc -Wall -g $^ -o $@
main.o:main.c
    gcc -Wall -g -c $< -o $@
add.o:add.c add.h
    gcc -Wall -g -c $< -o $@
sub.o:sub.c sub.h
    gcc -Wall -g -c $< -o $@
clean:
    rm -f main $(OBJECTS)

"Makefile" 12L, 237C 1,1 全部
```

这样的话,我们的 Makefile 就会更加简洁,更加专业一些。我们通过自动化的变量\$@/\$</\$^和自定义的变量 OBJECTS 来简化 Makefile 的编写。测试使可以正常运行的:

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
whd@whd-Lenovo:~/GDBNOTES$ make
gcc -Wall -g -c main.c -o main.o
gcc -Wall -g -c add.c -o add.o
gcc -Wall -g -c sub.c -o sub.o
gcc -Wall -g main.o add.o sub.o -o main
whd@whd-Lenovo:~/GDBNOTES$
```

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
gcc -Wall -g -c add.c -o add.o
gcc -Wall -g -c sub.c -o sub.o
gcc -Wall -g main.o add.o sub.o -o main
whd@whd-Lenovo:~/GDBNOTES$ make clean
rm -f main main.o add.o sub.o
whd@whd-Lenovo:~/GDBNOTES$
```

通常我们将该文件明名为大写的 Makefile,当然小写的也是可以的。输入命令 make 系统就会去寻找大写的或者小写的 Makefile。那么如果 Makefile 是其他文件呢? 也是可以的。

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
whd@whd-Lenovo:~/GDBNOTES$ make clean -f Makefile.1
rm -f main main.o add.o sub.o
whd@whd-Lenovo:~/GDBNOTES$
```


这说明用 Makefile.1 中的规则来执行。为了更清楚的看一下是执行该文件中的规则，clean 中添加新的一行代码：echo “begin delete ...”

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
.PHONY:clean
main:main.o add.o sub.o
    gcc -Wall -g main.o add.o sub.o -o main
main.o:main.c
    gcc -Wall -g -c main.c -o main.o
add.o:add.c add.h
    gcc -Wall -g -c add.c -o add.o
sub.o:sub.c sub.h
    gcc -Wall -g -c sub.c -o sub.o
clean:
    echo "begin delete ..."
    rm -f main main.o add.o sub.o
-- 插入 --                                     11,15 全部
```

然后执行删除的操作：

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
whd@whd-Lenovo:~/GDBNOTES$ vi Makefile.1
whd@whd-Lenovo:~/GDBNOTES$ ls
add.c  add.o      main      main.o      Makefile.1      sub.c  sub.o
add.h  gdb调试入门.wps  main.c    Makefile    makefile (上) .wps  sub.h
whd@whd-Lenovo:~/GDBNOTES$ make clean -f Makefile.1
echo "begin delete..."
begin delete...
rm -f main main.o add.o sub.o
whd@whd-Lenovo:~/GDBNOTES$ ls
add.c  gdb调试入门.wps  Makefile    makefile (上) .wps  sub.h
add.h  main.c            Makefile.1  sub.c
whd@whd-Lenovo:~/GDBNOTES$
```

将 echo 前面加上@,那么这条命令就不会输出，但是 echo 命令中的字符串还是会输出的：

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
.PHONY:clean
main:main.o add.o sub.o
    gcc -Wall -g main.o add.o sub.o -o main
main.o:main.c
    gcc -Wall -g -c main.c -o main.o
add.o:add.c add.h
    gcc -Wall -g -c add.c -o add.o
sub.o:sub.c sub.h
    gcc -Wall -g -c sub.c -o sub.o
clean:
    @echo "begin delete..."
    rm -f main main.o add.o sub.o
1,1 全部

终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
whd@whd-Lenovo:~/GDBNOTES$ make clean -f Makefile.1
begin delete...
rm -f main main.o add.o sub.o
whd@whd-Lenovo:~/GDBNOTES$ vi Makefile.1
whd@whd-Lenovo:~/GDBNOTES$
```

