> 这是作为新手的鄙人的新手教程，主要是与大一升大二的同学们分享交流。若有纰漏还请指正。
>
> 2016.8.19/15:48更新：有两个Makefile不应该使用通配符 `%` ，已修改

# Makefile简介

Makefile是一个文件名为 `Makefile` 的文本文件，用于定义项目的编译规则，以便于整个项目的编译。

（创建方法： `touch Makefile` 编辑方法： `gedit Makefile` `vim Makefile` ...）

如果不使用Makefile，可能我们就需要跟之前一样手打一大串编译命令来编译代码——大一时便是深有体会。 `g++ main.cpp support1.cpp support2.cpp support3.cpp ...` 这样的命令每次都要打一遍，如果是在平时的题目倒还好，如果遇到有数十个cpp和hpp的项目那就不好玩了，而且这样编译，有些依赖关系不清楚也是一个麻烦。

Makefile中就可以定义好各个文件的依赖关系，在之后再需要编译时，只需要执行 `make` 命令就可以自动编译了。

在一次 `make` 之后，**一般** 会生成很多 **目标文件(*.o)** 和一个可执行文件，当这些文件和源代码都没有被修改时，再次执行 `make` 会提示 `make: 'bin/your_program' is up to date.` ，而当你只修改了一个源代码文件再执行 `make` 时，它也不会重复编译已经最新的文件，而只编译依赖了你的源代码的文件，这对提高编译效率是非常重要的。

# 编译过程

关于编译过程，其实应该在C/C++的课程中已经讲过的……然而像鄙人，就算听过也还是弄不清楚，所以还是需要再参考一些资料。本文就不再赘述，仅提供一篇博客(←这是超链接)吧。

# Makefile初级教程

**规则** 的基本语法：

```
target ... : prerequisites ...
    command
    ...
    ...
```

- **target** 是下面的命令的 **目标** ，即下面命令是为了target而生的。这个 **目标** 可以是*.o文件，也可以是可执行文件

- **prerequisites** 则是生成该目标所 **依赖** 的文件，如果找不到依赖的文件，下面的命令就不会执行且会中断`make`
- **command** 就是生成目标文件的命令，一般就是编译命令了，如`g++ main.cpp`等等(**注意**：命令前面必须有 `Tab` (‘\t’) **真正** 的`Tab`，这样make才会认为它是指令)

直接看一个例子会更加直观。

假设有下列三个文件和下述的目录结构

```
// main.cpp
#include <iostream>
#include <string>
#include "support1.hpp"

int main() {
  std::string s = "Testing";
  support1::PrintItself(s);
  return 0;
}
```

```
// support1.cpp
#include "support1.hpp"
void support1::PrintItself(std::string s) {
  std::cout << s << std::endl;
}
```

```
// support.hpp
#include <iostream>
namespace support1 {
  void PrintItself(std::string);
}
```

```
# 目录结构
user@computer:~/learnmake$ tree
.
├── main.cpp
├── Makefile
├── support1.cpp
└── support1.hpp
```

main.cpp中include了support1，我们的 **目的** 是编译出一个可执行文件`main`

毫无疑问可以使用`g++ main.cpp support1.cpp -o main`这样的命令，于是就写成了下面的Makefile

```
# Makefile version 1
main: main.cpp support1.cpp support1.hpp
  g++ main.cpp support1.cpp -o main
```

- main就是上文说到的target
- 生成它依赖于所有的代码文件(main.cpp support1.cpp support1.hpp)
- 生成它的命令是`g++ main.cpp support1.cpp -o main`

这个Makefile简单易懂，但是`不行，这不程序员`。这样的关系还是不太清晰。我们一般让各个cpp文件生成.o文件，再将它们链接起来。也就是需要下面的命令

```
g++ support1.cpp -c              # 生成support1.o
g++ main.cpp -c                  # 生成main.o
g++ main.o support1.o -o main    # 链接生成可执行文件main
```

那就按这个改吧

```
# Makefile version 2
main: main.o support1.o
    g++ main.o support1.o -o main

support1.o: support1.hpp support1.cpp
    g++ -c support1.cpp

main.o: main.cpp
    g++ -c main.cpp
```

这里就可以再说说 **依赖关系** 怎么体现了。按顺序来读：

1. 需要得到目标`main`，先看看它所依赖的文件在不在——并没有
2. 于是先生成所依赖的文件`main.o`和`support1.o`
3. `main.o`依赖`main.cpp`，在目录中寻找，有则执行命令，无则报错
4. `support1.o`同上
5. 依赖文件已经都存在了，生成目标`out`
6. 结束

这样，一个简单的Makefile就完成了，然而它还有很多可以改进的地方

先看下面这一个Makefile

```
# Makefile version 3
CC := g++
FLAGS := -std=c++11 -w

main: support1.o main.o
    $(CC) $(FLAGS) main.o support1.o -o $@

support1.o: support1.hpp support1.cpp
    $(CC) $(FLAGS) -c support1.cpp

main.o: main.cpp
    $(CC) $(FLAGS) -c main.cpp

clean:
    @rm -f *.o
    @rm -f *.gch
    @rm -f main
```

突然出现的这些语法，下面我们逐个说说

- 变量(如`CC := g++`、`$(CC)`)
  - 在Makefile中可以使用"<name> := <content>"声明变量，并在此后使用$(<name>)可以使用，类似C/C++中的宏，它是字符串的直接替换。如`$(CC) $(FLAGS) -c support1.cpp`会变成`g++ -std=c++11 -w -c support1.cpp`。
  - 前缀是`$`符号，后面带个括号`()`的都是变量，如`$@`也是变量，称为 **自动化变量** ，它是目标文件的名字。如目标为`main`的这一段的命令会变成`g++ -std=c++ -w main.o support1.o -o main`

- 伪目标(没有依赖文件的目标`clean:`)
  - `clean`一段中，target后面没有依赖文件，这称为 **伪目标** ，作用是写了之后就可以使用`make clean`来执行它的命令集了
  - 这里的命令集可以使用许多shell命令，但似乎并不是所有都能用
  - 命令前面的`@`表示 **不显示这条命令** ，和Windows下的.bat类似(下面分别是不使用`@`和使用`@`的对比)

```
在Makefile中加入
foo:
  echo "Testing"
# 同样地，这是伪目标，可以使用make foo来执行其命令集

user@computer:~/learnmake$ make foo
echo "Testing"
Testing

Makefile中改为
foo:
  @echo "Testing"

user@computer:~/learnmake$ make foo
Testing
```

# 复杂一点的情况

有时候一个项目的目录并没有如此简单，例如：

```
.
├── bin
├── build
├── include
├── lib
|   ├── mysql
|   └── mysql++
└── src
```

- bin是编译生成的可执行文件
- build是编译的中间文件如*.o文件
- include是各种.h或.hpp文件
- lib是一些必要的库文件
- src是.cpp文件

这个时候，事情并不简单。因为现在的代码“身首异处”，所以要把它们的路径告诉编译器才行，我们使用 **相对路径** 来达到效果。

`.`表示当前目录，`..`表示上一级目录，用类似这样的相对路径来找到需要的文件，当然`./`经常可以省略……(于是本文并没有者两个的使用)

以下面的目录结构为目标(假设原本不存在`bin`目录和`build`目录)：

```
.
├── bin
│   └── main
├── build
│   ├── main.o
│   └── support1.o
├── include
│   └── support1.hpp
├── Makefile
└── src
    ├── main.cpp
    └── support1.cpp
```

因为最后的目标`main`在`bin`目录中，所以target也应该是`bin/main`，而其依赖的两个文件也要写清楚目录了`build/main.o` `build/support1.o`。

而在编译前，应该确认`bin`和`build`都是存在的目录，因此需要在编译前多一条命令`mkdir -p bin`。因为要指明文件目录，于是编译命令变成`g++ -std=c++11 -w build/main.o build/support1.o -o bin/main`

那么编译生成support1.o的时候命令就是`g++ -std=c++11 -w src/support1.cpp -c -o build/support1.o`

> :-)骗你的

由于在support1.cpp中我们写了`#include "support1.hpp"`，按以前的经验，这样写是只能找到同一个文件夹下的文件的，所以需要加一个编译器的参数`-I./include`，让它去其他地方找头文件。这样编译命令就完成了，生成.o文件的也类似。那么Makefile就可以写出来了[1]

```
CC := g++
FLAGS := -std=c++11 -w
bin/main: build/support1.o build/main.o
    @mkdir -p bin
    $(CC) $(FLAGS) -I./include build/support1.o build/main.o -o $@

build/support1.o: src/support1.cpp
    @mkdir -p build
    $(CC) $(FLAGS) -I./include -c -o $@ src/support1.cpp

build/main.o: src/main.cpp
    @mkdir -p build
    $(CC) $(FLAGS) -I./include -c -o $@ src/main.cpp

clean:
    @rm -rf build
    @rm -rf bin
```

这样就好了吗？不，不可以。将来要是文件结构有了偏差，写这个Makefile的人是要负责的。如果目录不同了，那么改各个路径要每个都改，而最好的做法应该是只改一处就影响全局。所以应该下面这样会更好。

```
CC := g++
FLAGS := -std=c++11 -w
INC_DIR := include
SRC_DIR := src
BUILD_DIR := build
BIN_DIR := bin
INCLUDE := -I./$(INC_DIR)
$(BIN_DIR)/main: $(BUILD_DIR)/support1.o $(BUILD_DIR)/main.o
    @mkdir -p $(BIN_DIR)
    $(CC) $(FLAGS) $(INCLUDE) $(BUILD_DIR)/support1.o $(BUILD_DIR)/main.o
 -o $@

$(BUILD_DIR)/support1.o: $(SRC_DIR)/support1.cpp
    @mkdir -p $(BUILD_DIR)
    $(CC) $(FLAGS) $(INCLUDE) -c -o $@ $(SRC_DIR)/support1.cpp

$(BUILD_DIR)/main.o: $(SRC_DIR)/main.cpp
    @mkdir -p $(BUILD_DIR)
    $(CC) $(FLAGS) $(INCLUDE) -c -o $@ $(SRC_DIR)/main.cpp

clean:
    @rm -rf $(BUILD_DIR)
    @rm -rf $(BIN_DIR)
```

**对于不同的情况，只要一个一个目标、一个一个依赖来写就好了**

---

**防止背的锅越来越大的分割线……**

---

# 下面用的是通配符，并不适用于所有情况

# 还请各位谨慎使用

然而这个Makefile还可以更加简短并更加完善，如下面代码(出现的新东西，后面细说)

```
CC := g++
FLAGS := -std=c++11 -w
INC_DIR := include
SRC_DIR := src
BUILD_DIR := build
BIN_DIR := bin
INCLUDE := -I./$(INC_DIR)

$(BIN_DIR)/main: $(BUILD_DIR)/support1.o $(BUILD_DIR)/main.o
    @mkdir -p $(BIN_DIR)
    $(CC) $(FLAGS) $(INCLUDE) $^ -o $@

$(BUILD_DIR)/%.o: $(SRC_DIR)/%.cpp
    @mkdir -p $(BUILD_DIR)
    $(CC) $(FLAGS) $(INCLUDE) -c -o $@ $<

clean:
    @rm -rf $(BUILD_DIR)
    @rm -rf $(BIN_DIR)
```

- 两个自动化变量
  - `$^`依赖文件的集合，用空格分隔
  - `$<`第一个依赖文件
  - 其实在`%.o`部分，也可以换成`$^`
  - 还有上文已经提过的`$@`目标文件
- 通配符`%`
  - 作用是就是`*`的作用了……通配……

这里的好处就是当`src`目录下再多cpp文件时，生成%.o文件的这一部分不用更改，生成可执行文件那里只要加一个依赖就好了

实际上还有方法让生成可执行文件的规则也自动，但又会聊到更加深的内容了

---

或许这篇新手向的教程并不让你尽兴，因此如果你想了解更多的Makefile知识，可以参考下面两篇文章

跟我一起写Makefile(←这是超链接):写的非常详细，但少了自动化变量的解释

补充:自动化变量(←这是超链接)

---

1. 可能有好朋友注意到这里的support1.hpp去掉了，~~因为编译的时候就会找support1.hpp，找不到就会编译错误，所以它是否写在依赖规则里并不影响结果，也就可有可无了~~订正：在这

里只是为了达到使用下文通配符%就包含两个cpp的效果，实际上写这个也是必要的，例如：support1.hpp被修改后，再次使用make，如果不写，那么make就不会重新编译了 ↩