初探Makefile

概述

Makefile是gnu系列的一个C语言编译工具，作用是在没有IDE的情况下集成多文件编译命令，并能表示各文件的依赖关系。各大著名IDE大多也采用Makefile作为编译工具链，比如Dev。Jetbrains的Clion则更加强大，采用了cmake，但底层仍然是Makefile。

一言以蔽之，Makefile的主要作用是：

1、集成多个编译命令；

2、表示文件的依赖关系；

3、根据依赖关系中各文件的时间戳来决定重新编译哪些文件，节省不必要的重复编译过程。

Makefile文件的名称就是Makefile。当一个目录下有这个文件时，就可以使用make命令进行编译了。make命令适用于Linux，在Windows下，应该用`mingw32-make`命令。

Makefile的基本结构如下：

```
# variables defination，变量定义
object1: dependency1 dependency2 ...
    cmd1 # cmd是编译命令
    cmd2 # 命令要用tab键（制表符）开头，而不是四个空格
    ...
# object1 叫做目标，dependency1 叫做依赖
object2: dependency3 dependency4 ...
    cmd3
    cmd4
    ...
```

1、目标

目标是编译任务的名称。基本格式为：

```
main.exe: # 目标名称
    gcc main.c -o main.exe
```

也就是目标名+冒号。目标名的作用是给编译任务指定命令参数名，比如要编译main.exe时，可以输入命令

```
make main.exe # 也可以只输入make
```

Makefile还可以指定多个目标，比如

```
main.exe:
    gcc main.o fact.o -o main.exe

main.o:
    gcc -c main.c -o main.o

fact.o:
    gcc -c fact.c -o fact.o
```

指定了三个目标，make命令通过命令行参数可选择执行哪一个编译任务。比如输入`make main.o`就编译main.o。

Makefile还规定，如果缺省命令行参数，只输入一个make，则编译第一个目标。所以要编译main.exe，只需要输入make即可。

为了更好地表示依赖关系，一般来说，**目标名称用编译生成的文件名命名**。大家在写Makefile的时候务必遵守这个规则。

但也有例外的，比如下面这个：

```
clear:
    rm -f *.o
    rm main.exe
```

clear就不是文件名。主要是因为，它不和其它任何任务有依赖关系，并且它不是生成文件的任务，而是删除文件的任务，当然就不能用生成文件命表示。因此必须另外命名。清理命令常用clear、clean等表示。根据Makefile的代码风格约定，它一般写在最后。**这种不是文件名的目标，叫做伪目标或类目标。**

了解了目标的概念，我们还可以个性化定制自己需要的目标，比如一个冗长的运行命令行，可以写成

```
run:
    main.exe -t abc -h def < in.txt > out.txt
```

这样，每次运行的时候就可以输入`make run`而不必打那一长串恶心的命令了。

2、命令

Makefile再强大，也离不开基本命令，因此Makefile必须用操作系统支持的编译命令来撰写。

每一个目标下的每一条命令必须用制表符（tab键）开头（写Makefile的时候一定要注意，不要写成四个空格！切记切记！），以区别于目标。说白了就是，不能和目标对齐，否则make程序会误认为命令也是目标。

3、依赖关系

依赖关系是Makefile中最强大的功能了。它可以根据依赖关系图（依赖关系是一个有向无环图）来决定哪些文件需要重新编译，从而节省不必要的重新编译。

一般来说，我们只需要表示以下依赖关系：

1、可执行文件依赖于各个目标文件

2、目标文件依赖于相应的源文件以及源文件所包含的头文件。如果头文件有嵌套包含的关系，则一并考虑。

先看一个例子：

```c
// main.c
#include <stdio.h>
#include "fibo.h"
#include "fact.h"

int main() {
    int n;
    scanf("%d", &n);
    printf("%d %d", fibo(n), fact(n));
    return 0;
}
```

```c
// fibo.c
int fibo(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    return fibo(n - 1) + fibo(n - 2);
}
```

```c
// fact.c
int fact(int n) {
    if (n <= 0) return 1;
    return n * fact(n - 1);
}
```

main.c包含`fibo.h fact.h`。因此，main.exe依赖于`fact.o fibo.o main.o`，main.o依赖于`main.c fibo.h fact.h`。Makefile应该如下：

```
# 冒号后面要有空格，依赖文件名用空格隔开
main.exe: fact.o fibo.o main.o
    gcc fact.o fibo.o main.o -o main.exe

main.o: main.c fibo.h fact.h
    gcc -c main.c -o main.o

fact.o: fact.c
    gcc -c fact.c -o fact.o

fibo.o: fibo.c
    gcc -c fibo.c -o fibo.o
```

当执行make命令时，解析程序首先检查终极目标main.exe的依赖，发现它依赖于`fact.o fibo.o main.o`，于是递归地检查它们三个目标，比如检查到main.o，发现它依赖于`main.c fibo.h fact.h`，接下来检查这三个文件，以此类推，直到检查到无依赖的目标（可以称为叶子节点）为止。在回溯阶段就会执行相应的编译命令。这就是Makefile的基本流程。

而到了如果检查到叶子节点为最新，则在回溯阶段不执行编译命令。这就减少了重复编译。这也是为什么有时候执行make会出现`nothing to do`的提示。

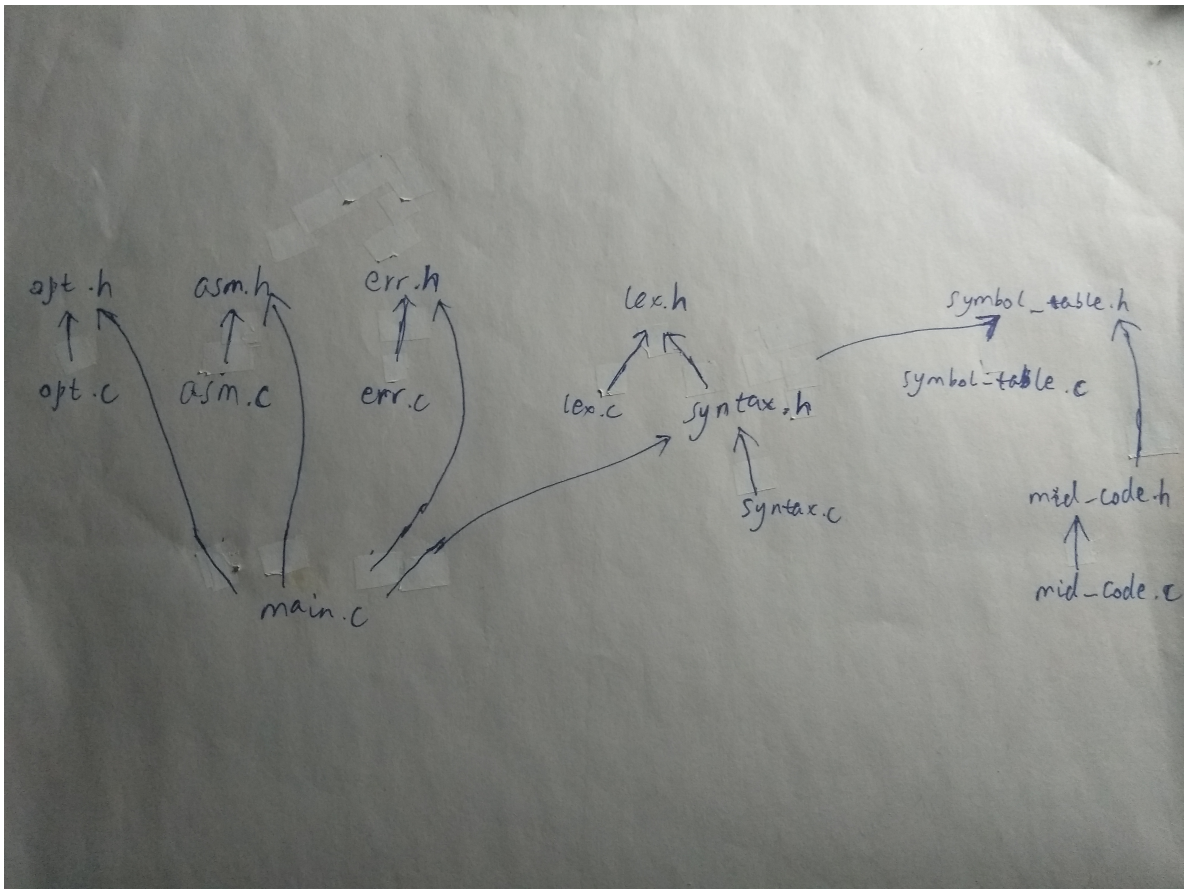
我们来用另一个复杂的例子加深理解。比如下面这个目录树，#号后面标识了包含关系，也就是说这个源文件包含了后面注释的头文件：

```
.
|-- asm.c # asm.h
|-- asm.h
|-- err.c # err.h
|-- err.h
|-- lex.c # lex.h
|-- lex.h
|-- main.c # asm.h err.h syntax.h opt.h
|-- mid_code.c # mid_code.h
|-- mid_code.h # symbol_table.h
```

```
|-- opt.c # opt.h
|-- opt.h
|-- symbol_table.c # symbol_table.h
|-- symbol_table.h
|-- syntax.c # syntax.h
+-- syntax.h # symbol_table.h lex.h
```

我们来分析一下，首先，每个目标文件`.o`都依赖于相应的c文件。同时由于.c文件的包含关系，相应的目标文件还依赖于被包含的头文件。而最终的可执行文件又依赖于每一个目标文件。

据此可画出包含关系树：



每个A→B表示A依赖于B。这是一个有向无环图，我们对每个.c文件进行DFS，可达的所有结点就是它的依赖。比如，`syntax.c`共有`syntax.h lex.h symbol_table.h`三个依赖。

包含关系如此复杂，为了保持Makefile的简洁性，我们应该做保守处理，也就是认为每个.c源文件依赖于所有的头文件。这样虽说损失了一些性能，但编译出来的结果绝对正确。

假设我们的编译还需要加上`-Wall -O2`选项，表示输出尽可能多的警告，并进行`O2`优化。编译命令就是`gcc -Wall -O2`

所以，我们的Makefile就是这样：

```
main.exe: main.o lex.o err.o syntax.o \
          symbol_table.o mid_code.o asm.o opt.o
    gcc -Wall -O2 main.o lex.o err.o syntax.o \
        symbol_table.o mid_code.o asm.o opt.o -o main.exe

main.o: main.c lex.h err.h syntax.h symbol_table.h \
        mid_code.h asm.h opt.h
    gcc -Wall -O2 -c main.c -o main.o

lex.o: lex.c lex.h err.h syntax.h symbol_table.h \
       mid_code.h asm.h opt.h
    gcc -Wall -O2 -c lex.c -o lex.o

err.o: err.c lex.h err.h syntax.h symbol_table.h \
       mid_code.h asm.h opt.h
    gcc -Wall -O2 -c err.c -o err.o

syntax.o: syntax.c lex.h err.h syntax.h symbol_table.h \
          mid_code.h asm.h opt.h
    gcc -Wall -O2 -c syntax.c -o syntax.o

symbol_table.o: symbol_table.c lex.h err.h syntax.h \
                symbol_table.h mid_code.h asm.h opt.h
    gcc -Wall -O2 -c symbol_table.c -o symbol_table.o

mid_code.o: mid_code.c lex.h err.h syntax.h \
            symbol_table.h mid_code.h asm.h opt.h
    gcc -Wall -O2 -c mid_code.c -o mid_code.o

asm.o: asm.c lex.h err.h syntax.h symbol_table.h \
       mid_code.h asm.h opt.h
    gcc -Wall -O2 -c asm.c -o asm.o

opt.o: opt.c lex.h err.h syntax.h symbol_table.h \
       mid_code.h asm.h opt.h
    gcc -Wall -O2 -c opt.c -o opt.o

clean:
    rm -f main.exe
    rm -f *.o
```

4、宏

刚才那个复杂的例子大家都看到了，是不是写起来超级恶心？大量的重复冗余堆在一起，不仅看起来想吐，而且写起来容易出错。如果我们的Makefile都是这个样子，那还不如直接打命令呢。

这个世界是懒人创造的，Makefile的创始人早就考虑过这个问题。所以引入了一种叫做宏的东西，就类似于C语言里面的`#define`。它的功能和C语言宏定义一毛一样，就是简单的文本替换。宏定义的格式如下：

```
vname1 := .... # 注意，:=的左右两边一定要有空格！
vname2 := ....
```

引用宏定义的方法是：

```
$(vname1)
$(vname2)
```

所以，上面那个例子，可以很简单的写成这样：

```
objects := main.o lex.o err.o syntax.o \
           symbol_table.o mid_code.o asm.o opt.o
cc := g++ -Wall -O2
srcs := main.c lex.c err.c syntax.c symbol_table.c \
        mid_code.c asm.c opt.c
headers := lex.h err.h syntax.h symbol_table.h \
           mid_code.h asm.h opt.h

main.exe: $(objects)
    $(cc) $(objects) -o main.exe

main.o: $(srcs) $(headers)
    $(cc) -c main.c -o main.o

lex.o: lex.c $(headers)
    $(cc) -c lex.c -o lex.o

err.o: err.c $(headers)
    $(cc) -c err.c -o err.o

syntax.o: syntax.c $(headers)
    $(cc) -c syntax.c -o syntax.o

symbol_table.o: symbol_table.c $(headers)
    $(cc) -c symbol_table.c -o symbol_table.o

mid_code.o: mid_code.c $(headers)
```

```
    $(cc) -c mid_code.c -o mid_code.o

asm.o: asm.c $(headers)
    $(cc) -c asm.c -o asm.o

opt.o: opt.c $(headers)
    $(cc) -c opt.c -o opt.o

clean:
    rm -f main.exe
    rm -f *.o
```

怎么样？是不是感觉一下子清爽了很多？

如果你认真领会了以上内容，那么你对Makefile算是真正入门了。

Makefile还有一些更高级的用法，包括宏定义中的`:=`和`=`的区别，以及多个Makefile相互包含等。有兴趣的同学可以进一步了解相关内容。