makefile 学习笔记

1 概述

1.1 Makefile 与 cmake 的区别

CMake 是一个项目构建工具，并且是跨平台的。关于项目构建我们所熟知的还有Makefile（通过 make 命令进行项目的构建），大多是IDE软件都集成了make，比如：VS 的 nmake、linux 下的 GNU make、Qt 的 qmake等，如果自己动手写 makefile，会发现，makefile 通常依赖于当前的编译平台，而且编写 makefile 的工作量比较大，解决依赖关系时也容易出错。

而 CMake 恰好能解决上述问题， 其允许开发者指定整个工程的编译流程，在根据编译平台，`自动生成本地化的Makefile和工程文件`，最后用户只需`make`编译即可，所以可以把CMake看成一款自动生成 Makefile的工具，其编译流程如下图：

简单点说就是，cmake 生成 makefile 文件

1.2 程序的编译与链接

1. 编译
   - 无论是 C、C++、还是 pas，首先要把源文件编译成中间代码文件，在 Windows 下也就是 `.obj 文件`，UNIX 下是 `.o 文件`，即 Object File，这个动作叫做编译（compile）。

    - 编译时，编译器需要的是语法的正确，函数与变量的声明的正确
2. 链接
    - 把大量的 Object File 合成执行文件。
    - 链接通常是你需要告诉编译器头文件的所在位置（头文件中应该只是声明，而定义应该放在C/C++文件中），只要所有的语法正确，编译器就可以编译出中间目标文件。
3. 库文件
   在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链接时需要明显地指出中间目标文件名，这对于编译很不方便，所以，我们要给中间目标文件打个包，在 Windows 下这种包叫“库文件”（Library File），也就是 .lib 文件，在 UNIX下，是 Archive File，也就是 .a 文件。

上述三个步骤总结如下：

总结一下，源文件首先会生成中间目标文件，再由中间目标文件生成执行文件。在编译时，编译器只检测程序语法，和函数、变量是否被声明。如果函数未被声明，编译器会给出一个警告，但可以生成Object File。而在链接程序时，链接器会在所有的 Object File 中找寻函数的实现，如果找不到，那到就会报链接错误码（Linker Error），在 VC 下，这种错误一般是：Link 2001 错误，意思说是说，链接器未能找到函数的实现。你需要指定函数的Object File.

1.3 Makefile 的基本规则

一个 Makefile 文件由一系列规则组成，每个规则定义了一个或多个目标文件的构建过程。每个规则由以下几个部分组成：

1. 目标（Target）：目标是规则的输出文件，也是规则的名称。它通常是一个可执行文件、库文件或中间文件。目标文件可以是 Makefile 中的一个或多个文件。
2. 依赖（Dependencies）：依赖是用于构建目标文件的输入文件或其他目标文件。依赖可以是源代码文件、头文件、库文件或其他规则定义的目标文件。如果依赖文件的修改时间较新，或者目标文件不存在，就会触发重新构建目标文件的动作。
3. 命令（Commands）：命令是构建目标文件的步骤。它们是 Makefile 中的一行或多行命令，用于编译、链接、复制文件等操作。**命令必须以一个 Tab 键开头，表示在 Makefile 中的命令行(shell命令)**。

下面是一个示例 Makefile 规则的基本结构：

```
target: dependencies
    command1
    command2
    ...
```

其中，`target` 是目标文件的名称，`dependencies` 是构建目标文件所需的依赖文件。`command1`、`command2` 等是构建目标文件的命令。

Makefile 中可以有多个规则，每个规则独占一行。Make 工具会按照规则的顺序执行，构建目标文件和满足依赖关系。

例如，下面是一个简单的 Makefile，用于构建一个名为 `hello` 的可执行文件：

```
hello: main.o utils.o
    gcc main.o utils.o -o hello

main.o: main.c utils.h
    gcc -c main.c

utils.o: utils.c utils.h
    gcc -c utils.c
```

在这个示例中，规则 `hello` 定义了目标文件 `hello` 的构建过程，它依赖于 `main.o` 和 `utils.o`。如果 `main.o` 或 `utils.o` 的修改时间较新，或者 `hello` 文件不存在，Make 工具将执行对应的命令来构建目标文件 `hello`。

规则 `main.o` 和 `utils.o` 分别定义了 `main.o` 和 `utils.o` 的构建过程，它们分别依赖于对应的源代码文件和头文件。如果源代码文件或头文件的修改时间较新，或者目标文件不存在，Make 工具将执行对应的命令来构建目标文件 `main.o` 或 `utils.o`。

这只是 Makefile 中规则的基本概念和结构。Makefile 还支持变量、条件语句、循环等高级特性，以提供更复杂的构建过程。你可以根据具体的需求使用适当的规则和功能来编写 Makefile 文件。

1.4 Makefile 中使用变量

在 Makefile 中，可以使用变量来存储和引用值。变量可以包含文件名、编译器选项、目录路径等常用的值。使用变量可以使 Makefile 更加灵活和可维护，方便在需要时进行修改。

变量定义及其使用方法：

```
#定义改一个变量
object = main.o test.o

edit: $(object)
    gcc -o edit $(object)

%.o: %.c
    gcc -c $< -o $@
```

下面是一个示例，演示了如何在 Makefile 中使用变量：

```
CC = gcc
CFLAGS = -Wall -Werror
SRC = main.c utils.c
OBJ = $(SRC:.c=.o)
TARGET = hello

$(TARGET): $(OBJ)
    $(CC) $(OBJ) -o $(TARGET)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

在这个示例中，我们定义了几个变量：

- `CC` 存储了编译器的名称，这里是 `gcc`。
- `CFLAGS` 存储了编译器的选项，这里是 `-Wall -Werror`，表示开启所有警告并将警告视为错误。
- `SRC` 存储了源代码文件的名称，这里是 `main.c utils.c`。
- `OBJ` 使用了变量替换功能，将 `SRC` 中的 `.c` 扩展名替换为 `.o` 扩展名。
- `TARGET` 存储了目标文件的名称，这里是 `hello`。

在规则中，我们使用这些变量来构建目标文件和执行命令。例如，在规则 `$(TARGET): $(OBJ)` 中，我们使用 `$(CC)` 和 `$(OBJ)` 分别引用了编译器和目标文件的变量值。同样，在规则 `%.o: %.c` 中，我们使用了 `$<` 和 `$@`，它们分别表示当前依赖文件和目标文件的变量值。

最后，我们还定义了一个 `clean` 规则，用于清理生成的目标文件和可执行文件。在这个规则中，我们使用了 `$(OBJ)` 和 `$(TARGET)` 来引用变量值，以便删除相应的文件。

通过使用变量，我们可以方便地修改编译器、选项、源文件以及目标文件的名称，而不必在整个 Makefile 中逐个修改。这提高了 Makefile 的可维护性和灵活性。

对上述的 %.o: %.c的说明

在 Makefile 中，规则 `%.o: %.c` 表示将所有以 `.o` 结尾的目标文件依赖于同名但以 `.c` 结尾的源文件。这个规则可以用来编译每个源文件并生成对应的目标文件。

在这个规则中，`$<` 和 `$@` 是自动化变量，它们分别表示当前规则中的依赖文件和目标文件的变量值。

- `$<` 表示依赖文件，即源文件的变量值。在这个示例中，`%.c` 表达式匹配到的源文件会被赋值给 `$<`，然后在命令中使用。
- `$@` 表示目标文件的变量值，即规则的目标文件。在这个示例中，`%.o` 表达式匹配到的目标文件会被赋值给 `$@`，然后在命令中使用。

这样，当 Make 工具执行这个规则时，它会将 `$<` 替换成当前匹配到的源文件的名称，将 `$@` 替换成当前匹配到的目标文件的名称。这样，我们可以通过引用这些自动化变量来构建编译命令，以确保每个源文件都会被正确编译为对应的目标文件。

1.5 makefile 自动推导

只要 make 看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，如果 make 找到一个 whatever.o，那么 whatever.c，就会是 whatever.o 的依赖文件。并且 `cc -c whatever.c` 也会被推导出来

例如下面的例子：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
    cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
    rm edit $(objects)
```

在大部分情况下，`gcc` 和 `cc` 是可以互换使用的，它们代表同一个编译器.`gcc` 是 GNU Compiler Collection（GNU 编译器集合）的缩写，而 `cc` 是 C Compiler（C 编译器）的缩写。

在大多数的 Unix-like 系统中，`cc` 是一个符号链接（symbolic link），指向系统默认的 C 编译器，通常是 `gcc`。这样做是为了方便用户使用 `cc` 命令来编译 C 代码，而无需关心具体使用的是哪个编译器。

因此，当你在 Makefile 中使用 `gcc` 或 `cc` 来编译 C 代码时，效果是相同的，它们都会调用系统默认的 C 编译器进行编译。具体系统中的默认编译器可能会有所不同，可以通过在命令行中执行 `cc --version` 或 `gcc --version` 来查看默认的 C 编译器版本信息。

`.PHONY` 表示，clean 是个伪目标文件。

1.6 清空目标文件的规则

每个 Makefile 中都应该写一个清空目标文件（.o 和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。

一般的风格都是：

```
clean:
    rm edit $(objects)
```

更为稳健的做法是：

```
.PHONY : clean
clean :
    -rm edit $(objects)
```

前面说过，`.PHONY 意思表示 clean 是一个“伪目标”`，。而在 `rm 命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事`。当然，clean 的规则不要放在文件的开头，不然，这就会变成 make 的默认目标，相信谁也不愿意这样。不成文的规矩是——“clean 从来都是放在文件的最后”。

1.7 注释

Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释是用“#”字符，这个就像 C/C++中的“//”一样。如果你要在你的 Makefile 中使用“#”字符，可以用反斜框进行转义，如：`\#`。

还值得一提的是，在 `Makefile 中的命令，必须要以[Tab]键开始`.

1.8 Makefile 文件名

默认的情况下，make 命令会在当前目录下按顺序找寻文件名为“GNUmakefile”、“makefile”、“Makefile”的文件，找到了解释这个文件。**在这三个文件名中，最好使用“Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。**

最好不要用“GNUmakefile”，这个文件是 GNU 的 make 识别的。有另外一些 make 只对全小写的“makefile”文件名敏感，但是基本上来说，大多数的 make 都支持“makefile”和“Makefile”这两种默认文件名。

也可以使用别的文件名来书写Makefile， 比 如 ： “Make.Linux”，
“Make.Solaris”，“Make.AIX”等，如果要指定特定的 Makefile，你可以使用 make 的 `“-f”和“--file” 参数`，如：make -f Make.Linux 或 make --file Make.AIX。

1.9 引用其他的 Makefile

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。**

include 的语法是include filename 可以是当前操作系统 Shell 的文件模式（可以保含路径和通配符） **在include前面可以有一些空字符，但是绝不能是[Tab]键开始。**include 和可以用一个或多个空格隔开。举个例子，你有这样几个 Makefile：a.mk、b.mk、c.mk，还有一个文件叫 foo.make，以及一个变量 $(bar)，其包含了 e.mk 和 f.mk，那么，下面的语句：

```
include foo.make *.mk $(bar)
# 等价于:
include foo.make a.mk b.mk c.mk e.mk f.mk
```

make 命令开始时，会把找寻 include 所指出的其它 Makefile，并把其内容安置在当前的位置上。就好像 C/C++的#include 指令一样。如果文件都没有指定绝对路径或是相对路径的话，make 会在当前目录下首先寻找，如果当前目录下没有找到，那么，make 还会在下面的几个目录下找：

1、如果 make 执行时，有 `-I` 或 `--include-dir` 参数，那么 make 就会在这个参数所指定的目录下去寻找。
2、如果目录/include（一般是：/usr/local/bin 或/usr/include）存在的话，make 也会去找。如果有文件没有找到的话，make 会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成 makefile 的读取，make 会再重试这些没有找到，或是不能读取的文件，如果还是不行，make 才会出现一条致命信息。**如果你想让 make不理那些无法读取的文件，而继续执行，你可以在 include 前加一个减号 `-`。**