Module02-03
Linux 开发环境：make 和 makefile

Linux 开发环境 - make 和 makefile

- vim
- 使用 gcc/g++
- → make 和 makefile
- 使用 gdb
- CVS
- Eclipse CDT

make 和 makefile

这节的课程中，我们将通过一个 Makefile 的演变逐渐掌握如下知识点：

- make 命令简介
- makefile 规则
- 简单的 Makefile
- 使用变量
- 常用的自动变量
- 使用 %.o:%.c 样式
- .PHONY 与伪目标
- Makefile 模块化
- 修饰 Makefile

make 和 makefile - make 命令

- make： 一个项目管理的工具
  - 常用的执行方式：
    - 1： make
    - 2： make target
      - 上述两种方式默认会在当前目录下查找名为： makefile 或 Makefile
      - 第二种形式， make 特定的目标
    - 3： make -f file
    - 4： make -f file target
      - 同形式 1、 2，不过是指定名为的 file 的 makefile 文件
  - 示例：

```
$ make install
$ make -f sub1.mk              #make 指定的文件
```

- makefile 规则
  - makefile 的内容：
    - makefile 由一系列的规则（ rules ）组成，而规则可以简单的认为是由 目标、依赖，命令组成，形式如下：

```
# 规则
target ... : prerequisites ...
    command                # 命令之前必须以一个 tab 开头
    ...

# 例如:
hello : hello.o    # 目标 hello : 依赖 hello.o
    g++ -o"hello" hello.o   # 命令
```

  - 只有目标文件不存在或依赖文件比目标文件新的情况下，才需要执行命令

make 和 makefile - 简单的 makefile

- 简单的 makefile（目录结构）

```
./make_proj/
|--- bin/
|     |--- make_test          # 可执行文件
|--- build/                   # 目录结构与 src 相同，用于放置 .o 文件
|     |--- Makefile           #  makefile 文件
|     |--- main.o
|     |--- sub1/
|     |     |--- *.o
|     +--- sub2/
|           |--- *.o
|--- inc/                     # 头文件（.h 文件）目录
|     |--- *.h
|--- src/                     # 实现文件（.cpp 文件）目录
|     |--- main.cpp
|     |--- sub1/
|     |     |--- *.cpp
|     +--- sub2/
|           |--- *.cpp
```

make 和 makefile - 简单的 makefile

- 简单的 makefile（内容）：

```
#Makefile v01
all : ../bin/make_test

../bin/make_test : ./main.o ./sub1/dummy1.o \
        ./sub1/dummy2.o ./sub2/dummy3.o \
        ./sub2/dummy4.o
    g++ -o ../bin/make_test ./main.o ./sub1/dummy1.o \
        ./sub1/dummy2.o ./sub2/dummy3.o \
        ./sub2/dummy4.o

./main.o : ../src/main.cpp
    g++ -g -Wall -c ../src/main.cpp -I ../inc
......    # dummy1~dummy3步骤略
./sub2/dummy4.o : ../src/sub2/dummy4.cpp
    g++ -g -Wall -o ./sub2/dummy4.o -c \
        ../src/sub2/dummy4.cpp -I ../inc

clean:
    rm -rf ./*.o ./sub1/* ./sub2/* ../bin/*
```

make 和 makefile - 使用变量

- 使用变量：
  - 使用变量让 makefile 更容易管理
  - Makefile 变量的定义规则与 bash 变量定义类似（注意两者的区别）：

    ```
    variable = value 或 variable := value
    ```

  - 变量值可以通过追加而改变，如：

    ```
    OBJS += ../src/sub2/dummy3.o \
            ../src/sub2/dummy4.o
    ```

    上面语句表示，变量 OBJS（可能）在别处定义并已赋值，这里仅追加一些内容

- 使用变量（改造过的 makefile ）：

```
#Makefile v02
RM := rm -rf

OBJS = ./main.o ./sub1/dummy1.o \
       ./sub1/dummy2.o ./sub2/dummy3.o \
       ./sub2/dummy4.o

all : ../bin/make_test

../bin/make_test : ./main.o $(OBJS)
    g++ -o ../bin/make_test $(OBJS)

......    # main.o dummy1.o~dummy3.o步骤略
./sub2/dummy4.o : ../src/sub2/dummy4.cpp
    g++ -g -Wall -o ./sub2/dummy4.o -c \
        ../src/sub2/dummy4.cpp -I ../inc

clean:
    $(RM) $(OBJS) ../bin/*
```

- 常用的自动变量
  - 自动变量类似于 bash 的内置变量，常用的如下列表：

变量	说明
$@	指代规则中所有目标文件，如：main.o : main.cpp ($@ 指代 main.o)
$%	当目标文件是静态库（.a 文件）时，指代目标文件的成员。例如，如果一个目标是 "foo.a(bar.o)"，那么，"$%" 就是 "bar.o"，"$@" 就是 "foo.a"。如果目标不是静态库文件，那么，其值为空
$<	指代第一个依赖文件名
$?	指代所有比目标文件新的依赖文件
$^	指代所有依赖文件名，中间由空格分隔
$+	类似 "$^"，也是指代所有依赖文件。但改该变量不去除重复的依赖文件。

  - 使用变量（包括自动变量）的安全方式是将变量名放入 ( ) 中：

    如：$< → $(<)

make 和 makefile - 常用的自动变量

- 常用的自动变量（继续改造 makefile ）：

```
#Makefile v03
RM := rm -rf
INC := ../inc
OBJS = ./main.o ./sub1/dummy1.o \
       ./sub1/dummy2.o ./sub2/dummy3.o \
       ./sub2/dummy4.o

all : ../bin/make_test

../bin/make_test : $(OBJS)
        g++ -o $(@) $(^)

./main.o : ../src/main.cpp
        g++ -g -Wall -o $(@) -c $(<) -I$(INC)
......
./sub2/dummy4.o : ../src/sub2/dummy4.cpp
        g++ -g -Wall -o $(@) -c $(<) -I$(INC)
clean:
        $(RM) $(OBJS) ../bin/*
```

make 和 Makefile - 使用 %.o:%.c 样式

- 使用 %.o:%.c 样式
  - % 的通配规则简单描述为：置换文件名中除前缀和后缀之外的主干部分
  - %.o:%.c 样式示例说明：
    - 如当前目录（执行 make 命令的目录）下有： a.cpp, b.cpp，则

```
%.o:%.cpp
    command ...

# 上面一条规则等价于下面几条规则:
a.o : a.cpp
    command ...
b.o : b.cpp
    command ...
```

make 和 makefile - 使用 %.o:%.c 样式

- 使用 %.o:%.c 样式（继续改造 makefile ）：

```
#Makefile v04
RM := rm -rf
INC := ../inc
OBJS = ./main.o ./sub1/dummy1.o \
       ./sub1/dummy2.o ./sub2/dummy3.o \
       ./sub2/dummy4.o

all : ../bin/make_test

../bin/make_test : $(OBJS)
    g++ -o $(@) $(^)
./%.o : ../src/%.cpp
    g++ -g -Wall -o $(@) -c $(<) -I$(INC)
./sub1/%.o : ../src/sub1/%.cpp
    g++ -g -Wall -o $(@) -c $(<) -I$(INC)
./sub2/%.o : ../src/sub2/%.cpp
    g++ -g -Wall -o $(@) -c $(<) -I$(INC)

clean:
    $(RM) $(OBJS) ../bin/*
```

make 和 makefile - 伪目标

- 伪目标是什么
  - 如之前 Makefile 中的 clean 就是一个伪目标：

```
clean:
    $(RM) $(OBJS) ../bin/*
```

这个目标并没有依赖文件，而 rm 命令也不会生成一个名为 clean 的文件。当 make clean 时， rm 命令总是会被执行，因为 clean 这个文件不存在，按规则就得执行 clean 目标下的命令。

  - .PHONY 标签：
    - 作用是显式指定某个目标为伪目标，如 .PHONY : clean ，表明不论是否有 clean 这个文件， make clean 中 make 都会将 clean 当作伪目标。

make 和 makefile - 伪目标

- 为 Makefile 添加 .PHONY 标签

```
#Makefile v05
RM := rm -rf
INC := ../inc
OBJS = ./main.o ./sub1/dummy1.o \
       ./sub1/dummy2.o ./sub2/dummy3.o \
       ./sub2/dummy4.o

all : ../bin/make_test

../bin/make_test : $(OBJS)
    g++ -o $(@) $(^)
./%.o : ../src/%.cpp
    g++ -g -Wall -o $(@) -c $(<) -I$(INC)
./sub1/%.o : ../src/sub1/%.cpp
    g++ -g -Wall -o $(@) -c $(<) -I$(INC)
./sub2/%.o : ../src/sub2/%.cpp
    g++ -g -Wall -o $(@) -c $(<) -I$(INC)
.PHONY : clean
clean:
    $(RM) $(OBJS) ../bin/*
```

- Makefile 模块化
  - 从这次的项目目录结构看，我们可以按源文件的目录结构将项目划分成 3 个部分： ./src ， ./src/sub1 ， ./src/sub2
  - 根据上述划分，我们可以将 Makefile 分为 4 个文件：
    - ./build/Makefile： 主 makefile ，负责完成最后的工作，如链接，产生最终目标文件
    - ./build/sub.mk： 负责生成 ./src 下的目标文件 (.o 文件)
    - ./build/sub1/sub.mk： 负责生成 ./src/sub1 下的目标文件
    - ./build/sub2/sub.mk： 负责生成 ./src/sub2 下的目标文件

- Makefile 模块化 - 改进后的 Makefile

```
#Makefile v06
RM := rm -rf

include ./sub.mk
include ./sub1/sub.mk
include ./sub2/sub.mk

all : ../bin/make_test

../bin/make_test : $(OBJS)
    g++ -o $(@) $(^)

.PHONY : clean
clean:
    $(RM) $(OBJS) ../bin/*
```

- Makefile 模块化 - 划分的子 makefile

```
#./sub.mk
OBJS += \
./main.o

./%.o: ../src/%.cpp
    gcc -g -Wall -c -o $(@) $(<) -I"../inc"
```

```
#./sub1/sub.mk
OBJS += \
./sub1/dummy1.o \
./sub1/dummy2.o

./sub1/%.o: ../src/sub1/%.cpp
    gcc -g -Wall -c -o $(@) $(<) -I"../inc"
```

./sub2/sub.mk 同 ./sub1/sub.mk，略

- Makefile 模块化 - 改进后的 Makefile

```
#Makefile v07
RM := rm -rf

-include ./sub.mk        #1
-include ./sub1/sub.mk       #1
-include ./sub2/sub.mk       #1

all : ../bin/make_test

../bin/make_test : $(OBJS)
    @echo 'Building target: $@'             #2
    @echo 'Invoking: GCC C++ Linker'        #2
    g++ -o $(@) $(^)
    @echo 'Finished building target: $@'    #2
    @echo ' '       #2

.PHONY : clean
clean:
    -$(RM) $(OBJS) ../bin/*       #1
```

- Makefile 模块化 - 改进后的 Makefile

```
#./sub1/sub.mk
OBJS += \
./sub1/dummy1.o \
./sub1/dummy2.o

./sub1/%.o: ../src/sub1/%.cpp
    @echo 'Building file: $<'                    #2
    @echo 'Invoking: GCC C++ Compiler'           #2
    gcc -g -Wall -c -o $(@) $(<) -I"../inc"
    @echo 'Finished building: $<'                #2
    @echo ' '              #2
```

#1：在指令前加 - ，表示即使这条指令执行失败，也不影响 make 继续往下执行；

#2：通过 echo 命令打印一些信息。

在命令前添加 @ ，表示不打印这条命令本身，而是打印命令的输出，如：

echo 'test' 输出两行：echo 'test' 和 test

@echo 'test' 只输出一行：test

make 和 Makefile - 小结

- 通过这次课程的学习和实践，我们循序渐进的完成且不断改进一个项目的 makefile
- 我们完全可以运用这次课程所讲的内容编写一个合格的 makefile 文件
- 当然， Makefile 的规则远远不止于我们这次课囊括的内容，更全面的学习，强烈建议大家花些时间和精力阅读 The GNU Make Manual：

  http://www.gnu.org/software/make/manual/
- 建议大家了解一下 GNU 项目中的 autoconf 、 automake 等 autotools ，非常强大的 makefile 配置以及生成工具

  http://www.gnu.org/software/autoconf/manual/