

# makefile学习

---

## 1. GCC默认头文件搜索路径

---

```
echo | gcc -v -x c -E -
```

## 2. 编译过程

---

四个阶段：预处理，编译，汇编，链接

### 2.1 预处理阶段

预处理器 (cpp) hello.c --> hello.i

展开头文件，将头文件的内容复制到#include 的地方

宏替换

删除注释或将注释变为空行

进行条件编译的逻辑 (#ifndef #else #endif)

### 2.2 编译

编译器 (ccl) hello.i -> hello.s (汇编语言)

逐行检查语法错误

### 2.3 汇编阶段

汇编器 (as) hello.s --> hello.o 机器语言 (二进制代码)

用文本编辑器打开hello.o文件，将是乱码

### 2.4 链接阶段

连接器 (ld) hello.o printf.o --> hello (可执行程序)

将多个编译好的目标文件 (.o文件) 以及库文件链接在一起，形成一个可执行程序。

## 3. C语言的编译

---

### 3.1 c语言相关后缀

.a 静态库

.c 源文件

.h 头文件

.i 预处理文件

.o 目标文件

.s 汇编文件

.so 动态库, 共享库, 运行时库

```
// hello.c
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("hello world\n");

    return 0;
}
```

## 3.2 预处理阶段 (hello.c --> hello.i)

```
gcc -E hello.c      // 不会生成 .i 文件
gcc -E hello.c -o hello.i  // 生成 hello.i 文件
```

- -E 选项告诉编译器只进行预处理操作
- -o 选项把预处理的结果输出到指定文件

## 3.3 编译阶段 (hello.i --> hello.s)

```
gcc -S hello.i
gcc -S hello.i -o hello.s
// 也可以直接编译 hello.c 文件
```

- -S 选项告诉编译器, 进行预处理和编译生成汇编语言操作

## 3.4 汇编阶段 (hello.s --> hello.o)

```
gcc -c hello.s  // 也可以直接编译 hello.c 文件, gcc -c hello.c
gcc -c hello.s -o hello.o
gcc -c hello1.c hello2.c hello3.c  // 编译多个.c文件
```

## 3.5 链接阶段 (hello.o --> hello)

```
gcc hello.o      // 生成了 a.out 文件
gcc hello.o -o hello
```

## 3.6 执行程序

```
./hello
```

## 3.8 一步执行

```
gcc hello.c -o helloworld
./helloworld
```

## 4. C语言 .a 静态库的编译与链接

```
//TODO: add.c

int add(int a,int b)
{
    return a+b;
}
```

```
//TODO: minus.c

int minus(int a,int b)
{
    return a-b;
}
```

```
//TODO: main.c

#include <stdio.h>

int add(int a,int b);
int minus(int a,int b);

int main(int argc,char* argv[])
{
    int a=10;
    int b=5;
    int c;
    printf("a+b=%d\n",add(a,b));
    printf("a-b=%d\n",minus(a,b));

    return 0;
}
```

### 4.1 编译成 .o 文件

```
gcc -c [.c] -o [自定义文件名]
gcc -c [.c] [.c] ...    // 编译多个文件
```

```
gcc -c add.c minus.c    // 生成 add.o minus.o
```

## 4.2 编译静态库

```
ar -r [lib自定义库名.a] [.o] [.o] ... // 静态库名一般以lib开头
```

```
ar -r liboperation.a add.o minus.o // 生成 liboperation.a 静态库
```

## 4.3 链接成可执行文件

```
gcc [.c] [.a] -o [自定义输出文件名]
```

```
gcc [.c] -o [自定义输出文件名] -l[库名] -L[库所在路径]
```

```
gcc main.c liboperation.a -o exec // 链接目标文件和库文件，生成了可执行程序 exec
```

```
gcc main.o add.o minus.o -o exec1 // 链接目标文件，生成了可执行程序 exec1
```

# 5. C语言 .so 动态库的编译与链接

## 5.1 编译二进制.o文件

```
gcc -c -fpic [.c/.cpp] [.c/.cpp] ...
```

```
gcc -c -fpic add.c minus.c // 生成了 add.o minus.o
```

## 5.2 编译动态库文件

```
gcc -shared [.o] [.o] ... -o [lib自定义库名.so]
```

```
gcc -shared add.o minus.o -o liboperation.so
```

# 直接生成 .so

```
gcc -fpic -shared [.c] [.c] [...] -o lib[库名].so
```

**libname.so.x.y.z**

lib --> 固定代表共享库

name --> 共享库名称

so --> 固定后缀

x --> 主版本号

y --> 次版本号

z --> 发行版本号

## 5.3 链接动态库生成可执行程序

```
gcc [.c/.cpp] -o [自定义可执行文件名] -l[库名] -L[库路径] -w1, -rpath=[库路径]
```

```
gcc main.c -o main -loperation -L/home/book/Desktop/cprogram/Shared_Library  
// 库名是 operation,不是 liboperation
```

## 5.4 出现的问题(动态库没有加载到内存中)

```
./main
```

**./main: error while loading shared libraries: liboperation.so: cannot open shared object file: No such file or directory**

加载动态库失败了：liboperation，不能打开动态库文件。

**静态库：**GCC进行链接时，会把静态库中的代码打包到可执行程序中

**动态库：**运行时才加载的库，GCC进行链接时，动态库的代码不会被打包到可执行程序中

使用命令 `ldd xxx` 检查动态库的依赖关系

由于动态库没有被加载到内存中，导致 `./main` 运行失败

### 解决动态库加载失败问题

#### (1) 临时修改

```
export  
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/book/Desktop/cprogram/Shared_Library
```

#### (2) 当前用户永久修改

```
# 修改当前用户的环境变量文件（~/ .bashrc）（永久设置，针对当前用户）  
vim ~/.bashrc  
# 在文件末尾添加  
export  
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/book/Desktop/cprogram/Shared_Library  
# 让环境变量立即有效  
source ~/.bashrc
```

#### (3) 所有用户永久修改

```
# 修改 /etc/profile，即修改系统环境变量（永久设置，针对所有用户）  
sudo vim /etc/profile  
# 在文件末尾添加  
export  
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/book/Desktop/cprogram/Shared_Library  
# 让环境变量立即生效  
source /etc/profile
```

## 6. C++的编译

**.a 静态库文件**

**.c/.c++/.cc/.cp/.cpp/.cxx 源文件**

**.h 头文件**

**.ii 预处理文件**

**.o 目标文件**

**.s 汇编代码**

**.so 动态库文件**

c++编译时将c的gcc换为g++

## 7. Makefile基本格式，规则，伪目标

### 7.1 基本格式

```
targets: prerequisites          # targets依赖于prerequisites
[tab键]command
```

- **target** : 目标文件，可以是Object file，也可以是执行文件，还可以是一个标签（Label）。
- **prerequisites** :要生成那个targets所需要的文件或目标。
- **command** : 是make需要执行的命令

```
debug:
    @echo hello
# @ 将命令隐藏起来。 make debug 时不会显示echo hello，只会显示hello7.2
```

### 7.2 Makefile规则

- make 会在当前目录下找到一个名字叫 `Makefile` 或 `makefile` 的文件
- 如果找到，它会找文件中第一个目标文件（target），并把这个文件作为最终的目标文件。
- 如果target文件不存在，或是target文件依赖的.o文件（prerequisites）的文件修改时间要比target这个文件要新，就会执行后面所定义的命令command 来生成target文件
- 如果target依赖的.o文件（prerequisites）也存在，make会在当前文件中找到target为.o文件的依赖性，如果找到，再根据哪个规则生成.o文件

### 7.3 伪目标

“伪目标”不是一个文件，只是一个标签。我们要显示地指明这个“目标”才能让其生效。

“伪目标”的取名不能和文件名重名，否则就不会执行这个命令

为了避免和文件重名的这种情况，可以使用一个特殊的标记 `.PHONY` 来显示地指明一个目标是“伪目标”，向make说明，不管是否有这个同名的文件，这个目标就是“伪目标”。

```
.PHONY: clean
```

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”这个命令。

## 8. Makefile变量的定义，使用

变量在声明时需要赋初值。使用时，需要在变量名前加 \$ 符号，并用小括号 () 把变量括起来。

## 8.1 变量的定义

```
cpp := src/main.cpp
obj := objs/main.o
```

## 8.2 变量的引用

- 可以用 () 或 {}

```
cpp := ./main.cpp
obj := objs/main.o

${obj} : ${cpp}
    @g++ -c ${cpp} -o ${obj}
complie: ${obj}

debug:
    @echo ${cpp}
    @echo ${obj}

.PHONY: complie debug
```

## 8.3 预定义变量

- \$@ : 目标 (target) 的完整名称。
- \$< : 第一个依赖文件 (prerequisites) 的名称
- \$^ : 所有依赖文件 (prerequisites) , 以空格分开, 不包含重复的依赖文件

```
cpp := ./main.cpp
obj := objs/main.o

${obj} : ${cpp}
    @g++ -c $< -o $@
# g++ -c main.cpp -o objs/main.o

complie: ${obj}

clean:
    @rm -r objs

debug:
    @echo ${cpp}
    @echo ${obj}

.PHONY: clean complie debug
```

## 9. Makefile 几种等号，续行符

## 9.1 = (相当于赋给地址或指针)

- 简单的赋值运算
- 用于将右边的值分配给左边的变量
- 如果在后面的语句中重新定义了该变量，则使用新的值

```
HOST_ARCH = aarch64
TARGET = ${HOST_ARCH}

HOST_ARCH = amd64

debug:
    @echo ${TARGET}
    @echo ${HOST_ARCH}

.PHONY: debug
# TARGET = amd64
```

## 9.2 := (相当于直接赋值)

- 立即赋值运算符
- 用于在定义变量时立即求值
- 该值在定义后不再更改
- 即使在后面的语句中重新定义了该变量

```
HOST_ARCH := aarch64
TARGET := ${HOST_ARCH}

HOST_ARCH := amd64

debug:
    @echo ${TARGET}
    @echo ${HOST_ARCH}

.PHONY: debug
# TARGET = aarch64
```

## 9.3 ?=

- 默认赋值运算符
- 如果变量已经定义，则不进行任何操作
- 如果变量没有定义，则求值并分配

```
#HOST = aarch64
HOST ?= amd64

debug:
    echo ${HOST}

.PHONY: debug
```



## 9.4 累加 +=

```
include_paths := src

CXXFLAGS := -m64 -fPIC -g -O0 -std=c++11 -w -fopenmp

CXXFLAGS += ${include_paths}
debug:
    @echo ${CXXFLAGS}

.PHONY: debug
```

## 9.5 续行符 \

```
LDLIBS := cudart opencv_core \
        gomp nvinfer protobuf cudnn pthread \
        cublas nvcaffe_parser nvinfer_plugin
```

## 9.6 通配符 \* 与 %

- \* : shell命令中的通配符, 表示匹配任意字符串, 可以用在目录名或文件名
- % : make语法中的通配符, 表示匹配任意字符串, 并将匹配的字符串作为变量使用

# 10. Makefile的常用函数

函数调用, 很像变量的使用, 用 "\$" 来标识, 语法:

- `$(fn arguments)` 或 `${fn arguments}`  
# fn 函数名  
# arguments 函数参数, 参数间以逗号 "," 分隔, 函数名与参数之间用"空格"分隔

## 10.1 shell

```
$(shell <command> <arguments>)
```

- 名称: shell 命令函数 ——shell
- 功能: 调用 shell 命令 command
- 返回: 函数返回 shell 命令 command 的执行结果

```
# shell 指令, 在c++program目录下找到所有的.cpp文件
cpp_c++program := ${shell find ../c++program/ -name "*.cp"}

debug:
    @echo ${cpp_c++program}

.PHONY: debug

# shell指令, 获取计算机架构
HOST_ARCH := $(shell uname -m)
```

## 10.2 subst , patsubst

### 10.2.1 subst

```
$(subst <from>,<to>,<text>)
```

- 名称：字符串替换函数
- 功能：把字符串中的字符串替换为
- 返回：函数返回被替换过后的字符串

```
cpp_src := ${shell find ../c++program/ -name "*.cpp" }  
cpp_obj := ${subst ../c++program/,../c++program/objs/,${cpp_src}}  
cpp_obj := ${subst .cpp,.o,${cpp_obj}}
```

### 10.2.2 patsubst

```
$(patsubst <pattern>,<replacement>,<text>)
```

- 名称：模式字符串替换
- 功能：通配符%，表示任意长度的字符串，从text中取出pattren，替换为 replacement

```
src := ${shell find ../c++program/ -name "*.cpp"}  
obj := ${patsubst ../c++program/%.cpp,../c++program/objs/%.o,${src}}
```

## 10.3 foreach

```
$(foreach <var>,<list>,<text>)
```

- 名称：循环函数
- 功能：把字符串中的元素（路径）逐一取出来赋值给var，执行包含的表达式
- 返回：所返回的每个字符串所组成的整个字符串（以空格分隔）

```
include_path := /usr/include \  
               /usr/incude/opencv2/core  
  
include_path := $(foreach item,${include_path},-I${item})  
  
debug:  
    @echo ${include_path}  
  
.PHONY: debug
```

在Makefile中，‘-I’后面跟一个路径，用于告诉编译器在那些目录中查找头文件。在这个代码中，**\${item}**代表在foreach循环中的每个路径，**-I\${item}**实际为每个路径加上-I的前缀，这样编译器在编译时就会将这些路径作为头文件的搜索路径。

### m10.3.1 foreach 同等效果:

```
include_path := /usr/include \  
              /usr/include/opencv2/core  
  
I_flag := $(include_path:%=-I%)  
# % 是用通配符符号，用于匹配include_path中的每个元素  
# ==-I% 是替换的格式，表示将每个匹配到的元素前加上 '-I' 前缀
```

## 10.4 dir

```
$(dir <names...>)
```

- 名称：取目录的函数
- 功能：从文件名序列中取出目录部分。目录部分是指最后一个反斜杠"/"之前的部分。如果没有反斜杠，就返回 "./"
- 返回：返回文件名序列的目录部分

```
cpp_src := $(shell find . -name "*.cpp")  
cpp_obj := $(patsubst ./%.cpp,./objs/%.o,${cpp_src})  
  
./objs/%.o : ./%.cpp  
    @mkdir -p $(dir $@)  
    @g++ -c $^ -o $@  
  
complie: ${cpp_obj}  
  
debug:  
    @echo ${cpp_src}  
    @echo ${cpp_obj}  
  
.PHONY: debug complie
```

在Makefile规则中，`%`是Make语法中的通配符，而`*`是Shell中的通配符。在Makefile规则中，必须使用`%`来表示通配符，表示对目标文件和依赖文件的模式匹配。

## 10.5 notdir

```
$(notdir <names...>)
```

- 名称：提取文件名的函数
- 功能：从一个文件路径中提取文件名（去除了路径部分）
- 返回：返回文件名

```

ibs := ${notdir ${shell find /usr/lib -name "lib*"}}
#libs := ${notdir ${libs}}

debug:
    @echo ${libs}

.PHONY: debug

```

## 10.6 filter

```
$(filter pattern..., text)
```

- 功能: **filter**函数会遍历`text`中的每个元素, 然后根据`pattern`进行模式匹配。如果元素与任意一个`pattern`匹配, 则该元素会被包含在结果列表中, 否则会被忽略。(从一个列表中筛选出符合指定模式的元素)
- 返回: 返回一个新的列表
- **pattern**: 是一个模式, 可以包含通配符
- **text**: 是一个以空格或Tab字符分隔的列表

```

libs := ${notdir ${shell find /usr/lib -name "lib*"}}

a_lib := ${filter %.a,${libs}}
so_lib := ${filter %.so,${libs}}

debug:
    @echo ${so_lib}

.PHONY: debug

```

## 10.7 basename

```
${basename <names...>}
```

- 功能: 从一个文件路径中提取文件名的基本部分(删除路径名和扩展名部分),
- 返回: 返回文件名的基本部分
- **name**: 一个或多个文件路径(可以是变量、通配符表达式等)

```

# 删掉了 后缀名
libs := ${notdir ${shell find /usr/lib -name "lib*"}}

a_lib := ${basename ${filter %.a,${libs}}}
so_lib := ${basename ${filter %.so,${libs}}}

debug:
    @echo ${a_lib}

.PHONY: debug

```

```
# 删除lib前缀,用subst函数,用 空白 替换掉lib
libs := ${notdir ${shell find /usr/lib -name "lib*"}}

a_lib := ${subst lib,,${basename ${filter %.a,${libs}}}}
so_lib := ${subst lib,,${basename ${filter %.so,${libs}}}}

debug:
    @echo ${a_lib}

.PHONY: debug
```

## 10.8 filter-out

- 剔除不想要得字符串

```
obj := objs/add.o objs/minus.o objs/main.o
cpp_objs := ${filter-out objs/main.o,${obj}}
```

## 11.makefile编译过程

### 11.1 编译带头文件的程序

test0803/

```
├─ include
│   ├── add.hpp
│   └─ Minus.hpp
├─ makefile
├─ obj
│   ├── add.o
│   ├── main.o
│   └─ Miuns.o
├─ src
│   ├── add.cpp
│   ├── main.cpp
│   └─ Miuns.cpp
└─ workspace
    └─ exec
```

```
// add.hpp
#ifndef __ADD__
#define __ADD__

int add(int a,int b);

#endif
```

```
// Minus.hpp
#ifndef __MINUS__
#define __MINUS__

int Minus(int a,int b);

#endif
```

```
// add.cpp
#include "add.hpp"

int add(int a,int b)
{
    return a+b;
}
```

```
// Minus.cpp
#include "Minus.hpp"

int Minus(int a,int b)
{
    return a-b;
}
```

```
//main.cpp
#include "add.hpp"
#include "Minus.hpp"
#include <iostream>
using namespace std;

int main(void)
{
    int a=10;
    int b=5;
    int c = add(a,b);
    cout<<"a+b="<<c<<endl;
    c = Minus(a,b);
    cout<<"a-b="<<c<<endl;

    return 0;
}
```

```
# makefile
cpp_srcs := $(shell find src -name "*.cpp")
cpp_objs := ${patsubst src/%.cpp,obj/%.o,${cpp_srcs}}

include_paths := /home/book/Desktop/test0803/include

I_flags := ${include_paths:%=-I%}
# I_flag := ${foreach var,${include_paths},-I${var}}
```

```

compile_options := -m64 -O3 -w ${I_flags}

obj/%.o : src/%.cpp
    @mkdir -p $(dir $@)
    @g++ -c $^ -o $@ ${compile_options}

#链接
workspace/exec : ${cpp_objs}
    @mkdir -p ${dir $@}
    @g++ $^ -o $@

run : workspace/exec
    @./$<

clean:
    @rm -rf workspace/exec obj/%.o

debug:
    @echo ${cpp_srcs}
    @echo ${cpp_objs}
    @echo ${I_flags}
    @echo ${compile_options}

.PHONY: debug run clean

```

## 11.2 c/c++编译选项

### 11.2.1 编译选项

**-m64** : 指定为64位应用程序

**-std=** : 指定编译标准, eg: -std=c++11, -std=c++14

**-g** : 包含调试信息

**-w** : 不显示警告信息

**-o** : 优化等级, 通常使用: -O3

**-I** : 加在头文件路径前 (大写的i), **-Iinclude**(include为头文件目录), 指定头文件的搜索路径

**fpic** : (position-Independent Code) ,没有绝对地址, 全部使用相对地址, 代码可以被加载到内存的任意位置, 且可以正确的执行。**生成与位置无关的代码**。这正是共享库所要求的, 共享库被加载时, 在内存的位置不是固定的。

### 11.2.2 链接选项

**-l** : 加在库名前面

**-L** : 加在库路径前面

**-Wl,<选项>** : 将逗号分隔的<选项>传递给链接器

-rpath：运行的时候，去找的目录。运行的时候，要找 .so 文件，会从这个选项里指定的地方寻找

## 11.3 预处理 (main.cpp --> main.i)

```
// main.cpp
#include <iostream>
using namespace std;

int main(void)
{
    cout<<"Hello world\n";

    return 0;
}
```

```
cpp_srcs := ${shell find . -name "*.cpp"}
cpp_objs := ${patsubst %.cpp,%.i,${cpp_srcs}}

%.i : %.cpp
    @g++ -E $^ -o $@

compile : ${cpp_objs}

debug:
    @echo ${cpp_objs}

.PHONY: debug compile
```

## 11.4 编译(main.i --> main.s)

```
cpp_srcs := ${shell find . -name "*.cpp"}
# find 命令在当前目录及其子目录查找 '.cpp' 文件,但它返回的是相对路径,包含了当前目录的信息,返回的是 ./main.cpp
cpp_objs := ${patsubst ./%.cpp,obj/%.s,${cpp_srcs}}
cpp := ${shell find . -name "*.cpp" -printf "%P\n"}
# find命令的 -printf "%p\n" 选项来获取文件的相对路径,返回的是不包含当前目录信息的'.cpp'文件路径, 即 main.cpp
obj/%.s : %.cpp
    @mkdir -p ${dir $@}
    @g++ -S $^ -o $@

compile: ${cpp_objs}

debug:
    @echo ${cpp_objs}
    @echo ${cpp_srcs}
    @echo ${cpp}
```



```
.PHONY: compile
```

## 11.5 汇编(main.s-->main.o)

```
cpp_srcs := ${shell find . -name '*.cpp'}
cpp_objs := ${patsubst ./%.cpp,obj/%.o,${cpp_srcs}}

obj/%.o : %.cpp
    @mkdir -p ${dir $@}
    @g++ -c $^ -o $@

compile : ${cpp_objs}

debug :
    @echo ${cpp_objs}

.PHONY: compile debug
```

## 11.6 链接(生成可执行程序)

```
cpp_srcs := ${shell find . -name '*.cpp'}
cpp_objs := ${patsubst ./%.cpp,obj/%.o,${cpp_srcs}}

obj/%.o : %.cpp
    @mkdir -p ${dir $@}
    @g++ -c $^ -o $@

workspace/exec : ${cpp_objs}
    @mkdir -p ${dir $@}
    @g++ $^ -o $@

run : workspace/exec
    @./$<

debug :
    @echo ${cpp_objs}

clean:
    @rm -rf ./workspace ./obj

.PHONY: debug run clean
```

## 12. 静态库编译

StaticLibrary/

├── include

| ├── add.hpp

| └── Minus.hpp

└── lib

```

|   └─ libxxx.a
├─ makefile
├─ obj
|   └─ add.o
|   └─ main.o
|   └─ Miuns.o
├─ src
|   └─ add.cpp
|   └─ main.cpp
|   └─ Miuns.cpp
└─ workspace

```

## 12.1 编译过程

### 12.1.1 源文件[.c/.cpp] --> Object文件[.o]

```
g++ -c [.c/.cpp] [.c/.cpp] ... -o [.o] [.o] ... -I[.h/.hpp] -g
```

### 12.1.2 Object文件[.o] --> 静态库文件[libname.a]

```
ar -r [libname.a [.o] [.o] ...
```

### 12.1.3 main文件[.c/.cpp] --> Object文件[.o]

```
g++ -c main.cpp -o mian.o -I[.h/.hpp]
```

### 12.1.4 链接main的Object文件与静态库文件[libname.a]

```
g++ main.o -o [可执行文件] -l[name] -L[静态库路径]
# name为静态库名字
```

## 12.2 makefile

```

lib_srcs := ${filter-out src/main.cpp , ${shell find src -name "*.cpp"}}
lib_objs := ${patsubst src/%.cpp,obj/%.o,${lib_srcs}}

include_path := /home/book/Desktop/StaticLibrary/include

library_paths := /home/book/Desktop/StaticLibrary/lib

linking_libs := xxx

I_flag := ${foreach var,${include_path},-I${var}}

l_options := ${linking_libs:%=-l%}
L_options := ${library_paths:%=-L%}

compile_flags := -g -O3 -m64 ${I_flag} -std=c++11
linking_flags := ${l_options} ${L_options}

```

```

#-----编译静态库-----

obj/%.o : src/%.cpp
    @mkdir -p ${dir $@}
    @g++ -c $^ -o $@ ${compile_flags}

lib/libxxx.a : ${lib_objs}
    @mkdir -p ${dir $@}
    @ar -r $@ ${lib_objs}

static_lib : lib/libxxx.a

#-----链接静态库-----

obj/main.o : src/main.cpp
    @g++ -c $< -o $@ ${compile_flags}

workspace/exec : obj/main.o
    @mkdir -p ${dir $@}
    @g++ $< -o $@ ${linking_flags}

run: workspace/exec
    @./$^

debug :
    @echo ${lib_objs}
    @echo ${lib_srcs}

clean:
    rm -rf obj lib

.PHONY : debug static_lib run

```

## 13. 动态库编译

### 13.1 编译过程

#### 13.1.1 源文件[.c/.cpp] --> Object文件[.o]

```
g++ -c [.c/.cpp] [.c/.cpp]... -o [.o] [.o]... -I[.h/.hpp] -g -fpic
```

#### 13.1.2 Object文件[.o] --> 动态库文件[lib库名.so]

```
g++ -shared [.o] [.o]... -o [lib库名.so]
```

#### 13.1.3 main文件[.c/.cpp] --> Object文件[.o]

```
g++ -c [main.c/.cpp] -o main.o -I[.h/.hpp] -g
```

### 13.1.4 链接动态库文件

```
g++ main.o -o [可执行程序] -l[库名] -L[库路径] -wl,-rpath=[库路径]
```

## 13.2 makefile

### 13.2.1 更标准的makefile

```
cpp_srcs := ${shell find src -name '*.cpp'}
cpp_objs := ${patsubst src/%.cpp,obj/%.o,${cpp_srcs}}

so_objs := ${filter-out obj/main.o,${cpp_objs}}

include_paths := /home/book/Desktop/SharedLibrary/include

library_paths := /home/book/Desktop/SharedLibrary/lib

linking_libs := xxx

I_options := ${include_paths:%=-I%}
l_options := ${linking_libs:%=-l%}
L_options := ${library_paths:%=-L%}
r_options := ${library_paths:%=-wl,-rpath=%}

compile_options := -g -m64 -O3 -w -fpic ${I_options}
linking_options := ${l_options} ${L_options} ${r_options}

#-----编译cpp文件-----

obj/%.o : src/%.cpp
    @mkdir -p ${dir $@}
    @g++ -c $^ -o $@ ${compile_options}

compile : ${cpp_objs}

#-----编译动态库-----

lib/libxxx.so : ${so_objs}
    @mkdir -p ${dir $@}
    @g++ -shared $^ -o $@

dynamic : lib/libxxx.so

#-----链接动态库-----

workspace/exec : obj/main.o compile dynamic
    @mkdir -p ${dir $@}
    @g++ $< -o $@ ${linking_options}
```

```

run : workspace/exec
    @./$<

debug :
    @echo ${so_objs}

clean :
    @rm -rf lib obj workspace

.PHONY: debug compile dynamic clean run

```

### 13.2.2 不太标准的makefile

```

lib_srcs := ${filter-out src/main.cpp,${shell find src -name '*.cpp'}}

lib_objs := ${patsubst src/%.cpp,obj/%.o,${lib_srcs}}

include_paths := /home/book/Desktop/SharedLibrary/include

library_paths := /home/book/Desktop/SharedLibrary/lib

linking_libs := xxx

I_options := ${include_paths:%=-I%}

l_options := ${linking_libs:%=-l%}
L_options := ${library_paths:%=-L%}
r_options := ${library_paths:%=-Wl,-rpath=%}

compile_flags := -m64 -g -O3 ${I_options} -fpic
linking_flags := ${l_options} ${L_options} ${r_options}

#-----编译动态库-----

obj/%.o : src/%.cpp
    @mkdir -p ${dir $@}
    @g++ -c $^ -o $@ ${compile_flags}

lib/libxxx.so : ${lib_objs}
    @mkdir -p ${dir $@}
    @g++ -shared $^ -o $@

shared_lib : lib/libxxx.so

#-----链接动态库-----
obj/main.o : src/main.cpp
    @g++ -c $< -o $@ ${compile_flags}

```

```
workspace/exec : obj/main.o
    @mkdir -p ${dir $@}
    @g++ $< -o $@ ${linking_flags}

run: workspace/exec
# @LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${library_paths}
    @./$<

debug:
    @echo ${lib_objs}

clean:
    @rm -rf obj workspace lib

.PHONY: clean debug run shared_lib
```