C/C++编译工具

aihe

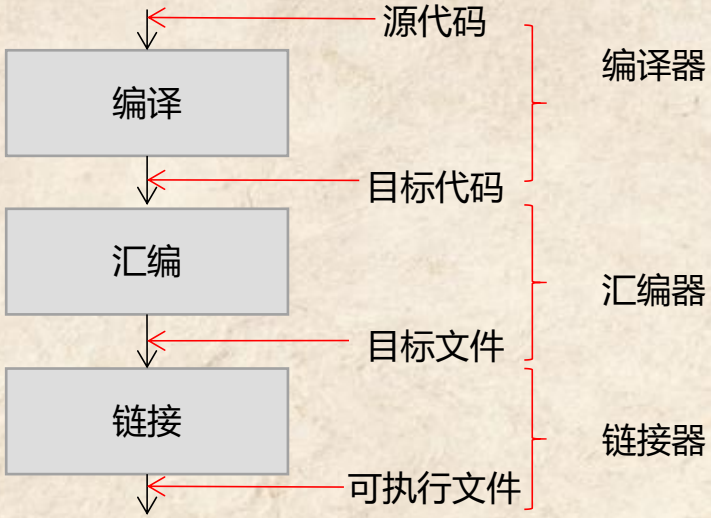
2018年1月

目录
CONTENTS

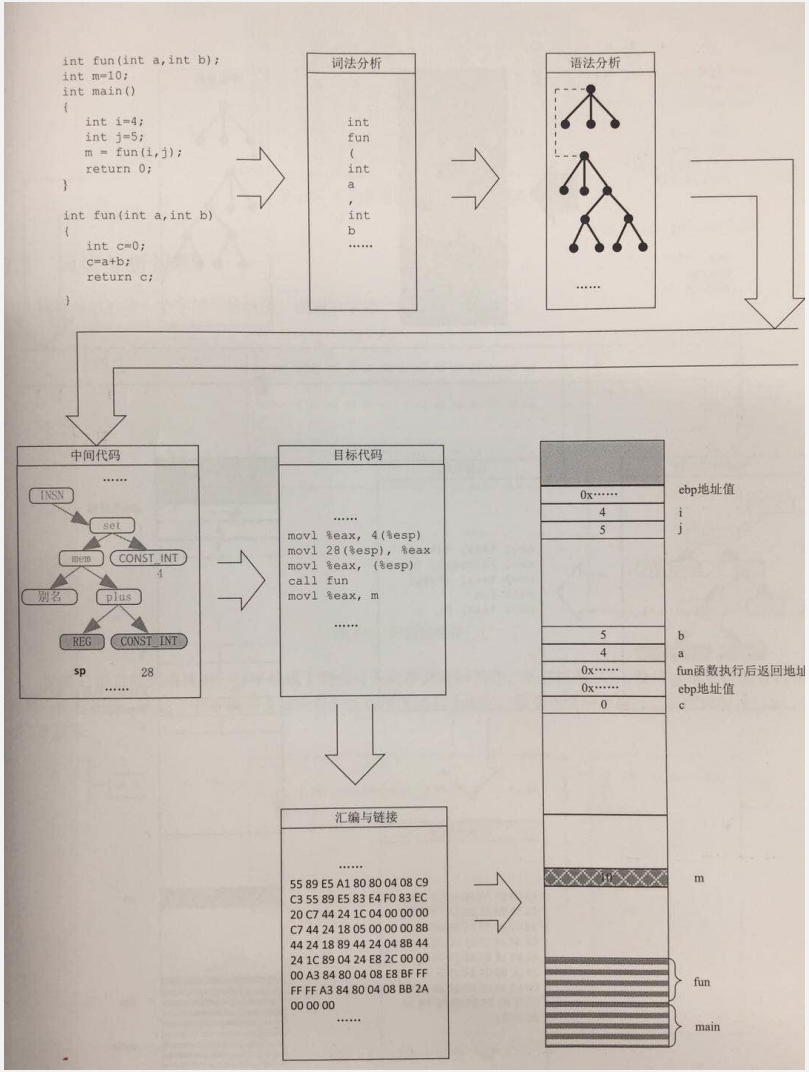
问题：c/c++编译过程

赏金：10฿

1 编译过程



1 编译过程



ps：推荐书籍，《编译系统透视：图解编译原理》

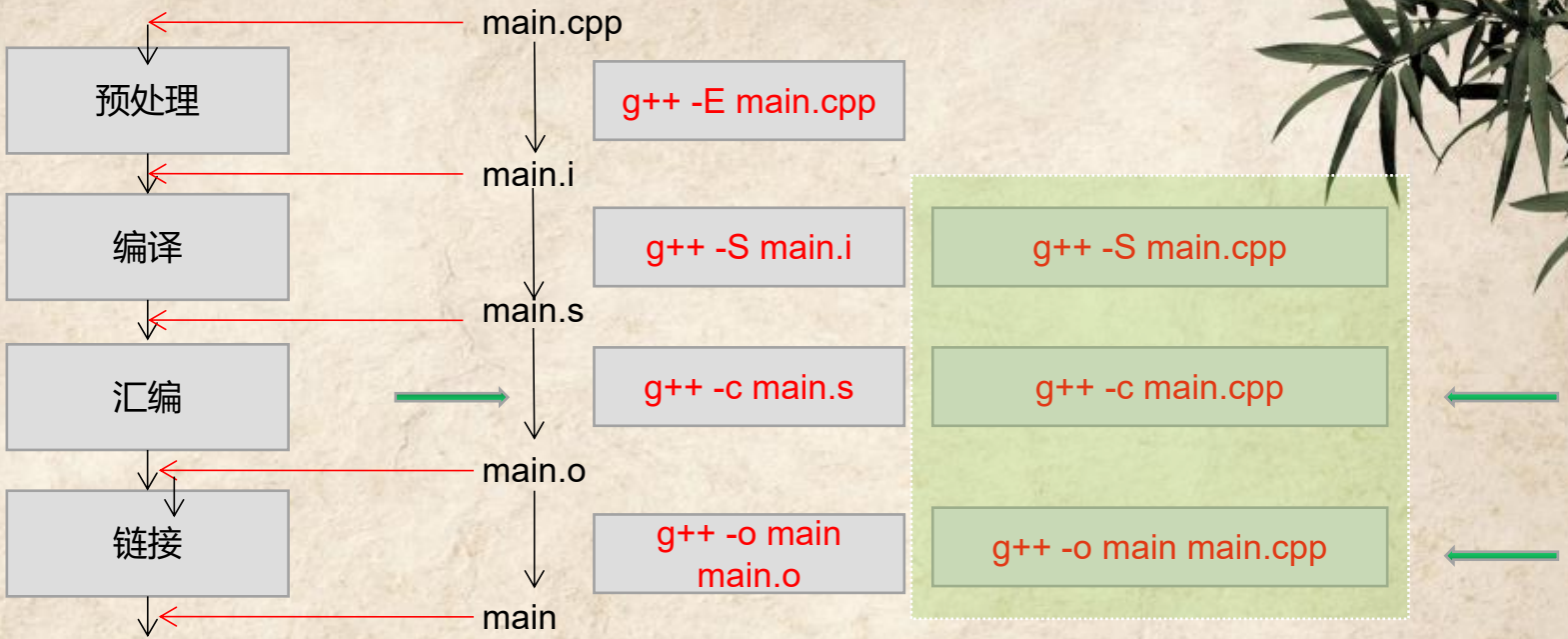
问题：除了gcc(g++)外，c/c++编译工具还有?

赏金：10฿

2.1 gcc/g++编译过程


main.cpp
源代码
预处理
main.i
源代码
编译器
编译
main.s
目标代码
汇编
汇编器
main.o
目标文件
链接
链接器
main
可执行文件

2.1 gcc/g++编译过程



ps: AT&T汇编语法简介 https://blog.csdn.net/p8x8x8/article/details/6531472

2.1 gcc/g++编译过程

- 静态库封装命令：

```
g++ -c a.cpp b.cpp c.cpp
ar cr libmylib.a a.o b.o c.o
```

- 动态库封装命令：

```
g++ a.cpp b.cpp c.cpp -fPIC -shared -o libmylib.so
```

- 链接-已有库（优先动态，其次静态）命令：

```
g++ main.cpp -L. -lmylib -o main
```

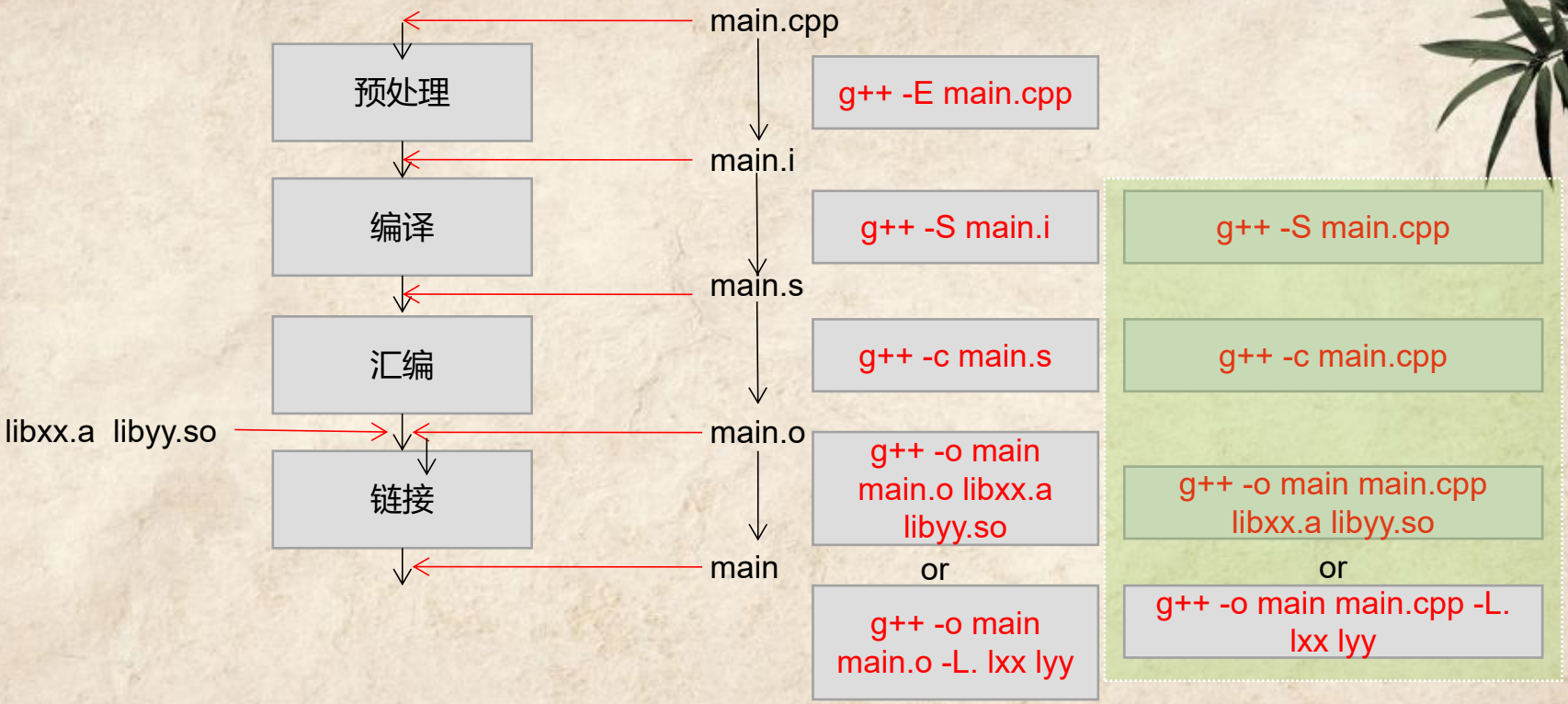
或

```
g++ main.cpp libmylib.so -o main / g++ main.cpp libmylib.a -o main
```

链接静态库命令（-static强制所有的库都使用静态库版本）：

```
g++ main.cpp -L. -lmylib -static -o main
```

2.1 gcc/g++编译过程



ps: AT&T汇编语法简介 https://blog.csdn.net/p8x8x8/article/details/6531472

2.1 gcc/g++编译过程

小试牛刀：编译项目，并输出图案。

- 项目来源：
https://github.com/aiainui/CompileTest.git
CompileTest/DrawingAll

- 要求：
1）生成.o文件
2）将Drawing.cpp封装成动态库
3）链接1）和2）生成可执行文件main

提示1-库路径设置（运行前）：
export CPLUS_INCLUDE_PATH=CPLUS_INCLUDE_PATH:头文件路径
export LIBRARY_PATH=$LIBRARY_PATH:库文件路径

提示2-库路径设置（链接时）：
-Wl,-rpath=库文件路径

赏金：20฿

2.1 gcc/g++编译过程

小试牛刀：编译项目，并输出图案。(答案)

- 项目来源：

https://github.com/aiainui/CompileTest.git
CompileTest/DrawingAll

- 要求：

1）将ProgressBar.cpp生成.o文件
2）将Drawing.cpp封装成动态库
3）链接1）和2）生成可执行文件main

```
g++ -c src/ProgressBar.cpp -o src/ProgressBar.o

g++ src/Drawing.cpp -fPIC -shared -o lib/libDrawing.so -Iinclude

g++ -o main -L./lib -I./include -lDrawing src/ProgressBar.o test/main.cpp -Wl,-rpath=./lib/
```

问题：运行时，动态库找不到的解决办法有哪些?

ref:
#运行时动态库路径设置
https://www.cnblogs.com/homejim/p/8004883.html
#1 动态库放在如下路径
/lib或/lib64
/usr/lib或/usr/lib64
#2 设置链接路径
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
《your_lib_path》
#3 修改配置文件/etc/ld.so.conf
/etc/ld.so.cache中缓存了动态库路径
#4 链接时加如下命令
-Wl,-rpath=《my_thirdparty_lib_path》
#5 软连接
#6 copy库到当前目录

赏金：10฿

2.2 makefile编写

❖ 书写规则

target ... : prerequisites ...

command

...

```
1 main:a.o b.o main.o
2     g++ -o main main.o a.o b.o
3 a.o:a.cpp
4     g++ -o a.o -c a.cpp
5 b.o:b.cpp
6     g++ -o b.o -c b.cpp
7 main.o:main.cpp
8     g++ -o main.o -c main.cpp
9 clean:
10    rm *.o main
```

2.2 makefile编写

- 变量

声明：GXX = g++、GXX := g++
取值：$(GXX)、${GXX}、$GXX

自动变量	含义
$@	目标集合
$%	当目标是函数库文件时，表示其中的目标文件名
$<	第一个依赖目标. 如果依赖目标是多个，逐个表示依赖目标
$?	比目标新的依赖目标的集合
$^	所有依赖目标的集合，会去除重复的依赖目标
$+	所有依赖目标的集合，不会去除重复的依赖目标

2.2 makefile编写

- 变量

```
main:a.o b.o main.o
    g++ -o main main.o a.o b.o
a.o:a.cpp
    g++ -o a.o -c a.cpp
b.o:b.cpp
    g++ -o b.o -c b.cpp
main.o:main.cpp
    g++ -o main.o -c main.cpp
clean:
    rm *.o main
```

```
GXX = g++

TARGET = main

OBJECT = \
        a.o \
        b.o \
        main.o

main:$(OBJECT)
    $(GXX) -o $@ $^
%.o:%.cpp
    $(GXX) -c $< -o $@
clean:
    rm $(OBJECT) $(TARGET)
```

- ...&逻辑表达式&函数(命令包)...

2.2 makefile编写

- 链接库，比如：lib/libtest.so

```
GXX = g++

TARGET = main

OBJECT = \
         a.o \
         b.o \
         main.o

main:$(OBJECT)
    $(GXX) -o $@ $^
%.o:%.cpp
    $(GXX) -c $< -o $@
clean:
    rm $(OBJECT) $(TARGET)
```

```
INCLUDEDIR = -I.

LIBDIR = \
         -L. -ltest

GXX = g++

CPPFFLAGS = \
            -Wl,-rpath=.


TARGET = main

OBJECT = \
         a.o \
         b.o \
         main.o

main:$(OBJECT)
    $(GXX) -o $@ $^ $(LIBDIR) $(INCLUDEDIR) $(CPPFFLAGS)
%.o:%.cpp
    $(GXX) -c $< -o $@
clean:
    rm $(OBJECT) $(TARGET)
```

2.2 makefile编写

- [asr]其他例子1，

```
1
2  INCLUDEDIR = -I.
3
4  LIBDIR = \
5          -Lsrc -ldecoder \
6          -Ldnn -lmmtScore \
7          -Lapm -lbdAPMAPI \
8          -Lsrc -lBV322PCM \
9          -Lpki -lasr_pki_linuxPC64 \
10         -Lnet -lasr_net_fsnlm \
11         -lcrypto -lpthread -lm -lstdc++ \
12         -Ldnn/mkl_lib \
13         -Wl,--start-group -lmkl_sequential -lmkl_core -lmkl_intel_lp64 -Wl,--end-group -lirc \
14         -Llib2-64/ullib/lib -lullib
15
16 define mkObjDir
17     @ test -d $(1) || mkdir -p $(1)
18 endef
19
20 GCC = g++
21 CPPFLAGS = -Wall -Winline -pipe -ffast-math -D_LINUX_64_
22
23 OBJDIR = obj
24
25 TARGET1 = asr_decoder_online
26
27 OBJ1 = $(addprefix $(OBJDIR)/, src/client_server.o)
28
29 all: $(TARGET1)
30     rm -rf output
31     mkdir -p output
32     mv ${TARGET1}  output/
33
34 $(TARGET1) : $(OBJ1)
35     $(GCC) -DDEBUG -g -o $@ $^ $(LIBDIR) $(INCLUDEDIR)
36
37 $(OBJDIR)/%.o : %.cpp
38     @ test -d $(OBJDIR) || mkdir -p $(OBJDIR)
39     $(call mkObjDir,$(dir $@))
40     $(GCC) -DDEBUG -g $(CPPFLAGS) -c $< -o $@ $(INCLUDEDIR)
41
42 clean:
43     rm -rf ./obj
44     rm -rf ./output
```

2.2 makefile编写

- [cv]其他例子2

```
LIBS = -lpthread -lm -lstdc++ -Llib -lopencv_core  -lopencv_highgui \
       -lopencv_imgcodecs  -lopencv_imgproc -lcurl -lssl -lcrypto \
       -lgraph -lrecongnition -lcaffe -lz

CPPFLAGS = -Wall -Winline -pipe -ffast-math -D_LINUX_64_
LDFLAGS = -I. -I./include/  -I./opencv/include/  -L/data/www/ocr/ocr-souti/souti_root/bin/lib/
LDFLAGS+= -Wl,-rpath=/data/www/ocr/ocr-souti/souti_root/bin/lib/

define mkObjDir
    @ test -d $(1) || mkdir -p $(1)
endef

define mkGitInfo
    @echo `git log | head -3` > output/version.txt
    @echo "------" >> output/version.txt
    @echo `git diff` >> output/version.txt
endef

GCC = g++ -std=c++11  -O2

OBJDIR = obj

TARGET1 = ocr_souti
TARGET2 = ocr_souti_debug

COREOBJ = \
        src/LIST.o \
        src/Chinese.o \
        src/PreProcess.o \
        src/MemPool.o \
        src/Tools.o \
        src/ImageCache.o \
        src/Pack.o \
        src/Decoder.o

COREOBJ_DEBUG = $(patsubst %.o,%_debug.o,$(COREOBJ))

OBJ1 = $(addprefix $(OBJDIR)/, $(COREOBJ) src/client_server.o)
OBJ2 = $(addprefix $(OBJDIR)/, $(COREOBJ_DEBUG) src/client_server_debug.o)
```

```

all: $(TARGET1) $(TARGET2)
    rm  -rf ../server
    mkdir -p ../server
    mv ${TARGET1}  ../server/
    mv ${TARGET2}  ../server/
    rm -rf ./obj

$(TARGET1) : $(OBJ1)
    $(GCC) -o $@ $^ $(LIBS) $(LDFLAGS)
$(TARGET2) : $(OBJ2)
    $(GCC) -g -o $@ $^ $(LIBS) $(LDFLAGS)

$(OBJDIR)/%.o : %.cpp
    @ test -d $(OBJDIR) || mkdir -p $(OBJDIR)
    $(call mkObjDir,$(dir $@))
    $(GCC) -O2 $(CPPFLAGS) -c $< -o $@ $(LDFLAGS)

$(OBJDIR)/%_debug.o : %.cpp
    @ test -d $(OBJDIR) || mkdir -p $(OBJDIR)
    $(call mkObjDir,$(dir $@))
    $(GCC) $(CPPFLAGS) -g -c $< -o $@ $(LDFLAGS)

clean:
    rm -rf ./obj
    rm -rf ../server
```

2.2 makefile编写

小试牛刀：编译项目，并输出图案。

- 项目来源：
https://github.com/aiainui/CompileTest.git
CompileTest/DrawingAll
- 要求：

1）将ProgressBar.cpp生成.o文件
2）将Drawing.cpp封装成动态库
3）链接1）和2）生成可执行文件main

2.2 makefile编写

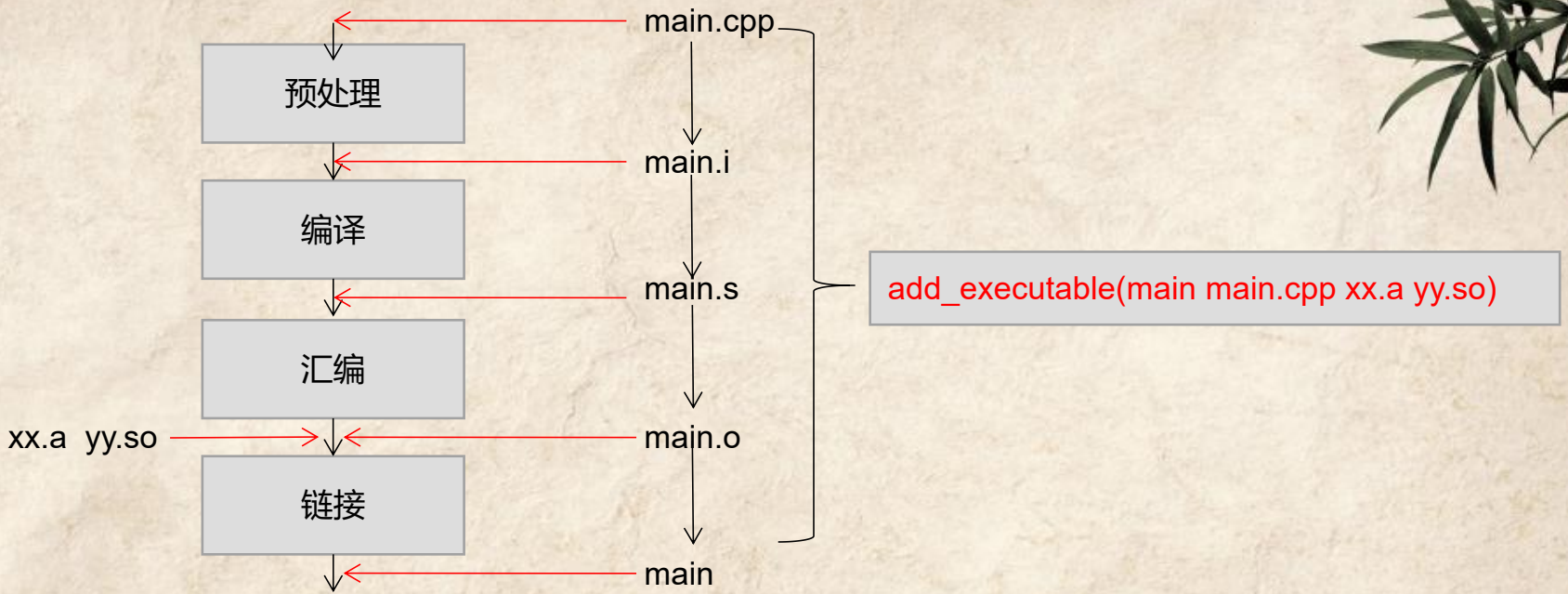
小试牛刀：编译项目，并输出图案。(答案)

- 项目来源：
https://github.com/aiainui/CompileTest.git
CompileTest/DrawingAll
- 要求：

1）将ProgressBar.cpp生成.o文件
2）将Drawing.cpp封装成动态库
3）链接1）和2）生成可执行文件main

```
LIBS = -L./lib -lDrawing

INCLUDEDIR = -I./include/
LDFLAGS = -Wl,-rpath=./lib/

OBJDIR = src
TESTDIR = test
BINDIR = bin
LIBDIR = lib


TARGET = $(addprefix $(BINDIR)/, main)

COREOBJ = \
        ProgressBar.o
OBJ1 = $(addprefix $(OBJDIR)/, $(COREOBJ))

all: $(TARGET)
    @echo "--end-makefile--"

$(TARGET) : $(OBJ1) $(TESTDIR)/main.cpp
    mkdir -p ./bin
    $(CXX) $^ -o $@ $(LIBS) $(INCLUDEDIR) $(LDFLAGS)

$(OBJDIR)/%.o:%.cpp
    $(CXX) $^ -o $@

clean:
    rm -rf ./bin
    rm -rf ./src/*.o
```

3. cmake


main.cpp
预处理
main.i
编译
main.s
汇编
xx.a yy.so
main.o
链接
main
add_executable(main main.cpp xx.a yy.so)

3. cmake

- 静态库封装：

```
AUX_SOURCE_DIRECTORY(. SRC_CPP)
add_library(mylib ${SRC_CPP})
```

- 动态库封装命令：

```
AUX_SOURCE_DIRECTORY(. SRC_CPP)
add_library(Test SHARED  ${SRC_CPP})
```

- 链接-已有库（优先动态，其次静态）命令：

```
link_directories("${PROJECT_SOURCE_DIR}/.")
target_link_libraries(main mylib)
or
link_libraries("${PROJECT_SOURCE_DIR}/libmylib.so")
```

3. cmake

画“心形图案”的makefile写法

- 链接动态库

```
→  MultiDirMultiCpp1 git:(master) ✗
.
├── build
├── CMakeLists.txt
├── include
│   └── Drawing.h
├── lib
│   └── libDrawing.so
├── src
└── test
    └── main.cpp
```

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
project (Demo3)
include_directories(${PROJECT_SOURCE_DIR}/include)
aux_source_directory(test MAIN_SRC)
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
link_libraries("${PROJECT_SOURCE_DIR}/lib/libDrawing.so")
add_executable(main ${MAIN_SRC})
```

4. NLP组-编译工具

✓ 编译

```
cmake_build
├── AddIncludePath.py
├── build_run.sh
└── CMakeLists.txt
```

../../cmake_build/build_run.sh

4. NLP组-编译工具

- ✓ 工程文件结构1

```
#############################################################################
# 目录结构：
#            当前目录(CMakeLists.txt)
#          /    |      |      \
#         /     |      |       \
#      build   src    test     dpd
#      /   \                /  |   |     \          \
#     /     \              /   |   |      \          \
#   bin     lib      dpd_inc*  | dpd_src* dpd_std* dpd_cmake_*_?
#                         dpd_lib*
#
#############################################################################
#
# src:      源代码目录
# build:    编译目录，在该目录下执行 cmake .. 及 make
# test:     测试文件目录，源码无关(当且仅当文件名以main开始时编成可执行文件)
# dpd:      存放各依赖链接
# dpd/dpd_inc*:      依赖库的头文件路径(不会递归添加)
# dpd/dpd_lib*:      依赖库的路径(会递归添加)
# dpd/dpd_src*:      依赖的源文件目录
# dpd/dpd_std*:      依赖的同此结构的文件目录
# dpd/dpd_rinc*:     依赖库的头文件路径(会递归添加)
# dpd/dpd_cmake_*_?:find_package *为关键字#为库名的.cmake, 指向的路径为CMAKE_PREFIX_PATH
#
# bulid/bin: 存放生成的可执行文件
# bulid/lib: 存放生成的库
#
# output:  install输出文件夹
#
#############################################################################
```

4. NLP组-编译工具

✓ 工程文件结构1

make.conf

```
project_name=net_comm
sys_lib_list=-lcrypto\\\ -lpthread\\\ -lm
# for opencv
#sys_lib_list=-lpthread\\\ -lm\\\ -ltiff\\\ -ljpeg\\\ -lz\\\ -lpng\\\ -ljasper\\\ -ldl
macro_definition=-DMY_LOG_LEVEL=0\\\ -DUSE_HOT_UPDATE
# idiom 1 qa 2 image_similar 3 ws_pos 4
#macro_definition=-DPRODUCT_SERVICE_NUMBER=2
#keep_make_tool=True
make_debug=True
#make_install_cpack=True
#install_dir=..\\/..\\/output
```

4. NLP组-编译工具

✓ 工程文件结构1，例：词语相似度计算

```
→  SimilarityCalculator git:(dyl) ✗
.
├── conf
│   └── log.conf
├── data
│   ├── category_num2str.new.txt
│   ├── category_num2str.txt
│   ├── category_standard.model
│   ├── category_standard.new.model
│   ├── category_str2num.txt
│   └── num2IC.txt
├── dpd
│   ├── dpd_inc_common -> ../../../common/tool/cpp
│   └── dpd_std_glog -> ../../../common/tool/cpp/glog
├── make.conf
├── out
├── src
│   ├── BaseIC.h
│   ├── BaseSemanticSimilarity.h
│   ├── ClassifyingDictionary.cpp
│   ├── ClassifyingDictionary.h
│   ├── ConceptTopologyIC.cpp
│   ├── ConceptTopologyIC.h
│   ├── InfoSemanticSimilarity.h
│   ├── LinInfoSemanticSimilarity.cpp
│   ├── LinInfoSemanticSimilarity.h
│   └── PathSemanticSimilarity.h
└── test
    ├── main_compare.cpp
    ├── main_create_num2IC.cpp
    ├── main_get_similar_word.cpp
    └── main_test_calsim.cpp
```

作业 使用“NLP组-编译工具”编译如下项目

小试牛刀：编译项目，并输出图案。

- 项目来源：
https://github.com/aiainui/CompileTest.git
CompileTest/DrawingAll
- 要求：

1）将ProgressBar.cpp生成.o文件
2）将Drawing.cpp封装成动态库
3）链接1）和2）生成可执行文件main

作业 使用“NLP组-编译工具”编译如下项目

小试牛刀：编译项目，并输出图案。（答案）

- 项目来源：
https://github.com/aiainui/CompileTest.git
CompileTest/DrawingAll

```
→  DrawAll
.
├── dpd
│   ├── dpd_inc_drawing -> ../include
│   └── dpd_lib_drawing -> ../lib
├── include
│   └── Drawing.h
├── lib
│   └── libDrawing.so
├── make.conf
├── src
│   └── ProgressBar.cpp
└── test
    └── main.cpp
```

```
→  DrawAll ../../cmake_build/build_run.sh
./build_run.sh

Set: project_name=drawing
Set: build_dir=build
Set: make_tool=cmake
Set: make_debug=True
→  DrawAll ./build/bin/main_drawing
I am drawing a heart...
```