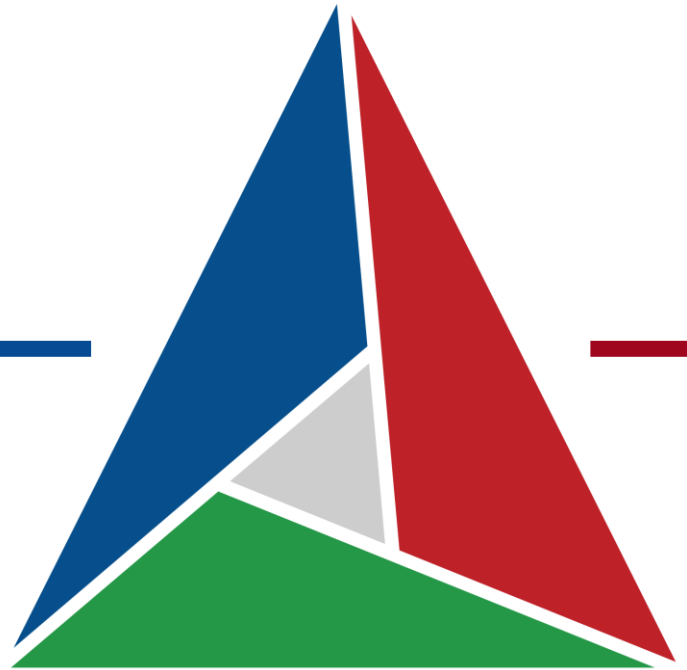
CMake的应用与实践



利凌云志

2019.11.17

内容概要

1. **认识CMake及应用**

   应用,与Gcc, Makefile, Autotools比较的优势?

2. **CMake语句的主体框架**

   cmake问题分析思路,主体的结构,语法的构成,及基本模块功能

3. **CMake的常用指令及变量**

   基本常用指令 (安装,测试,调试等),常用的Cmake语法变量含义

4. **CMake的实践应用**

   从简单的CMake文件说起---->生成链接库(静态&动态)---->

   如何引用链接库(内部&引用)---->更简单的组织Cmake的编译方式

1.认识CMake及应用

1. 认识CMake及应用

➢ CMake 是什么?

- 全称**Cross Platform Make**，起初为了跨平台需求，而后不断完善并广泛使用
- 一款优秀的**工程构建工具**（KDE开发者使用近10年autotools，在KDE4改用CMake）

➢ 特点及优势:

- 开放源代码，具有BSD许可
- 跨平台，支持Linux, Mac和Windows等不同操作系统
- 编译语言简单，易用，简化编译构建过程和编译过程
- 编程高效(比autotools快40%)，可扩展(ros中catkin，ament，colcon都是基于cmake构建）

1. 认识CMake及应用

➢ CMake 与其他编译工具的对比

◘ GCC

- 由GNU开发的编程语言译器，C/C++，Java等语言的开发；
- 当项目简单，可以gcc/g++编译目标和项目；
- 但比较复杂时，只用gcc组织编译架构变得极其困难；

◘ Makefile

- Makefile是有条理的gcc编译命令的文件，利用make工具来执行Makefile文件的编译指令；
- 当程序简单时，可以手写Makefile；
- 当程序复杂时，一般利用CMake和autotools来自动生成Makefile；

1. 认识CMake及应用

➢ CMake 与其他编译工具的对比

◘ Autotools

- autotools是一个工具集,具有灵活性较大,对用户角度使用较为友好(cmake生成文件权限较多)；
- 开发步骤太多，配置繁琐 [autoscan + autocconf + automake]；
- 通常编译的./configure文件，大多通过由autotools构建的，最终生成Makefile和config.h文件；

◘ CMake

- Cmake类似Make工具功能，用来“读取”并执行CMakeLists.txt文件的语句，最终生成Makefile文件；
- Cmake语言开发相对简单，易于理解；
- 目前很多项目正在抛弃Autotools，qmake等，转而采用cmake；

2.CMake的主体框架

2. CMake语法的主体框架

- 谈到Cmake我们能想到哪些问题？？？

  - 如何去组织一个项目的编译框架
  - 最终输出目标有哪些（可执行程序，动态库，静态库等）
  - 如何配置输出目标文件的指定编译参数（需要哪些编译参数及环境，需要哪些源文件）
  - 如何为指定的输出目标链接参数（怎么配置内外部依赖的pkg及lib，怎么链接外部库）

2. CMake语法的主体框架


```
command(arg1 arg2 ...)                                  # 运行命令
set(var_name var_value)                                 # 定义变量,或者给已经存在的变量赋值
command(arg1 ${var_name})                               # 使用变量

#********************************** 工程配置部分 **********************************#
cmake_minimum_required (VERSION num)                    # CMake最低版本号要求
project (cur_project_name)                              # 项目信息
set(CMAKE_CXX_FLAGS "XXX" )                            # 设定编译器版本，如-std=c++11
set(CMAKE_BUILD_TYPE "XXX")                             # 设定编译模式，如Debug/Release

#********************************** 依赖执行部分 **********************************#
find_package(std_lib_name VERSION REQUIRED)             # 引入外部依赖
add_library(<name> [lib_type] source1)                  # 生成库类型(动态,静态)
include_directories(${std_lib_name_INCLUEDE_DIRS})      # 指定include路径，放在add_executable前面
add_executable(cur_project_name XXX.cpp)                # 指定生成目标
target_link_libraries(${std_lib_name_LIBRARIES})        # 指定libraries路径，放在add_executable后面

#********************************** 其他辅助部分 **********************************#
function(function_name arg)                             # 定义一个函数
add_subdirectory(dir)                                   # 添加一个子目录
AUX_SOURCE_DIRECTORY(. SRC_LIST)                        # 查找当前目录所有文件，并保存到SRC_LIST变量中
FOREACH(one_dir ${SRC_LIST})
    MESSAGE(${one_dir})                                 # 使用message进行打印
ENDFOREACH(onedir)
```

主体框架:

- 工程配置部分

工程名、编译调试模式、编译系统语言

- 依赖执行部分

工程包，头文件、依赖库等

- 其他辅助部分(非必需)

参数打印、遍历目录等

- 判断控制部分(非必需)

条件判断、函数定义、条件执行等

2. CMake语法的主体框架

```
#********************************** 判断控制部分 **********************************#
if(expression)                    # 不区分大小写，并使用"#"来进行注释
    COMMAND1(ARGS)
ELSE(expression)
    COMMAND2(ARGS)
ENDIF(expression)

# expression
IF(var)                           # 不是空，0，N，NO，OFF，FALSE，NOTFOUND 或 <var>_NOTFOUND时，为真
IF(NOT var)                       # 与上述条件相反。
IF(var1 AND var2)                 # 当两个变量都为真是为真。
IF(var1 OR var2)                  # 当两个变量其中一个为真时为真。
IF(COMMAND cmd)                   # 当给定的cmd确实是命令并可以调用是为真
IF(EXISTS dir)                    # 目录名存在
IF(EXISTS file)                   # 文件名存在
IF(IS_DIRECTORY dirname)          # 当dirname是目录
IF(file1 IS_NEWER_THAN file2)     # 当file1比file2新,为真
IF(variable MATCHES regex)        # 符合正则

# 循环
WHILE(condition)
    COMMAND1(ARGS)
    // ...
ENDWHILE(condition)
```

Note:

1.对于${X}，X表示变量名称，${X}表示变量值，if语句除外。

2.导入的功能函数，如SET大小写均可，没有特殊限制

3.本文使用vscode，并下载CMAKE插件可显示相应指令的详细说明【ubuntu系统】

3.CMake的常用指令及变量

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ✓ PROJECT
- ✓ ADD_EXECUTABLE
- ✓ ADD_SUBDIRECTORY
- ✓ INCLUDE _DIRECTORIES
- ✓ LINK_DIRECTORIES
- ✓ TARGET_LINK_LIBRARIES
- ✓ ADD_LIBRARY
- ✓ AUX_SOURCE_DIRECTORY
- ✓ FOREACH
- ✓ MESSAGE
- ✓ IF ELSE ENDIF
- ✓ WHILE ENDWHILE
- ✓ FIND_PACKAGE
- ✓ SET

3. CMake的常用指令及变量

➢ CMake基本常用指令

- **ADD_DEFINITIONS**
- OPTION
- ADD_CUSTOM_TARGET
- ADD_DEPENDENCIES
- INSTALL
- TARGET_INCLUDE_DIRECTORIES
- SET_TARGET_PROPERTIES
- ENABLE_TESTING/ADD_TEST

➢ ADD_DEFINITIONS

- 为源文件的编译添加由-D引入的宏定义。
- 命令格式为 : add_definitions(-DFOO -DBAR ...)
- ✓ 例如：add_definitions(-DWIN32)

```
cmake -D CMAKE_BUILD_TYPE=RELEASE \
-D CMAKE_INSTALL_PREFIX=/usr/local \
-D WITH_CUDA=ON \
-D ENABLE_FAST_MATH=1 \
-D CUDA_FAST_MATH=1 \
-D WITH_CUBLAS=1 \
-DINSTALL_C_EXAMPLES=OFF \
-D INSTALL_PYTHON_EXAMPLES=ON \
```

OpenCV CMake
编译选项

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- **OPTION**
- ADD_CUSTOM_TARGET
- ADD_DEPENDENCIES
- INSTALL
- TARGET_INCLUDE_DIRECTORIES
- SET_TARGET_PROPERTIES
- ENABLE_TESTING/ADD_TEST

➢ OPTION

- 提供用户可以选择的选项。
- 命令格式为：

option(<variable> "description [initial value])

```
option (
        USE_MYMATH
        "Use tutorial provided math implementation"
        ON
       )
```

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- OPTION
- **ADD_CUSTOM_TARGET**
- ADD_DEPENDENCIES
- INSTALL
- TARGET_INCLUDE_DIRECTORIES
- SET_TARGET_PROPERTIES
- ENABLE_TESTING/ADD_TEST

➢ ADD_CUSTOM_COMMAND/TARGET

- [COMMAND] : 为工程添加一条自定义的构建规则。
- [TARGET] : 用于给指定名称的目标执行指定的命令，该目标没有输出文件，并始终被构建。

```
# 伪代码：为了说明生成自定义的命令
add_custom_command(TARGET ${CV_ADVANCE_NAME}
      PRE_BUILD
      COMMAND "伪代码 find_package std_msgs"
      )
# 为了说明导入生成自定义构建的命令
add_custom_target(CV_ADVANCE) ALL
      DEPENDS ${CV_ADVANCE_NAME} # 依赖add_custom_command输出的package包
      COMMENT "ros package std_msgs"
    )
```

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- OPTION
- ADD_CUSTOM_TARGET
- **ADD_DEPENDENCIES**
- INSTALL
- TARGET_INCLUDE_DIRECTORIES
- SET_TARGET_PROPERTIES
- ENABLE_TESTING/ADD_TEST

➢ ADD_DEPENDENCIES

- 用于解决链接时依赖的问题，用target_link_libraries可以搞定吗？？？

```
# 添加执行文件
add_executable(cur_project_name XXX.cpp)
# 将库文件链接链接到当前执行文件
target_link_libraries(cur_project_name ${std_lib_name_LIBRARIES})
```

- 当定义的target依赖的另一个target，确保在源码编译本target之前，其他的target已经被构建，使用该语句。
- **举个栗子吧** 😳

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- OPTION
- ADD_CUSTOM_TARGET
- **ADD_DEPENDENCIES**
- INSTALL
- TARGET_INCLUDE_DIRECTORIES
- SET_TARGET_PROPERTIES
- ENABLE_TESTING/ADD_TEST

◘ 栗子一：利用CMake构建添加依赖（伪代码）

```
# 添加了一条自定义构建的规则
add_custom_target(CV_ADVANCE DEPENDS ${CV_ADVANCE_NAME})
# 为项目添加了一个可执行文件
add_executable(${PORJECT_NAME} ${SRC_LIST})
# 链接了一个标准的库文件
target_link_libraries(${PORJECT_NAME} ${std_lib_name_LIBRARIES})
# 为项目链接一个依赖文件，项目程序依赖CV_ADVANCE
add_dependencies(${PORJECT_NAME} CV_ADVANCE)
```

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- OPTION
- ADD_CUSTOM_TARGET
- **ADD_DEPENDENCIES**
- INSTALL
- TARGET_INCLUDE_DIRECTORIES
- SET_TARGET_PROPERTIES
- ENABLE_TESTING/ADD_TEST

◘ 栗子二：ROS1.0在该指令上的应用（工程实例）

```
# 常规语法
cmake_minimum_required(VERSION 2.8.3)
project(HelloCV)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs genmsg)
# ros生成msg的语法
generate_messages(DEPENDENCIES std_msgs)
add_message_files(FILES HelloCvMsg.msg)
# 声明是catkin包
catkin_package()
# 创建工程实例
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(greet src/greet.cpp)
target_link_libraries(greet ${catkin_LIBRARIES})
# 添加greet依赖，依赖std_msgs 的变量
add_dependencies(greet HelloCV_generate_messages_cpp)
```

```
# HelloCvMsg.msg文件下的定义格式
std_msgs/String greeter
```

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- OPTION
- ADD_CUSTOM_TARGET
- ADD_DEPENDENCIES
- **INSTALL**
- TARGET_INCLUDE_DIRECTORIES
- SET_TARGET_PROPERTIES
- ENABLE_TESTING/ADD_TEST

➢ INSTALL

- 用于定义安装规则，安装的内容可以包括目标二进制、动态库、静态库以及文件、目录、脚本等。
- 常用的如OpenCV一般情况下安装到系统目录，即/usr/lib, /usr/bin和/usr/include [Ubuntu系统]

```
INSTALL(
        TARGETS myrun mylib mystaticlib
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION libstatic
        )
```

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- OPTION
- ADD_CUSTOM_TARGET
- ADD_DEPENDENCIES
- INSTALL
- **TARGET_INCLUDE_DIRECTORIES**
- SET_TARGET_PROPERTIES
- ENABLE_TESTING/ADD_TEST

➢ TARGET_INCLUDE_DIRECTORIES

- 设置include文件查找的目录，具体包含头文件应用形式，安装位置等。
- 命令格式为：

  target_include_directories(<target>[SYSTEM][BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items])

- interface|public|private 的作用范围？？？

```
# 外部链接时include的位置,如果不指明外部依赖无法找到对应的include库
target_include_directories(hello_cv_2_add_static_lib PUBLIC
                           $<BUILD_INTERFACE:${CMAKE_SOURCE_DIR}/include>     #源文件安装位置
                           $<INSTALL_INTERFACE:include>)                      #安装文件夹位置
```

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- OPTION
- ADD_CUSTOM_TARGET
- ADD_DEPENDENCIES
- INSTALL
- TARGET_INCLUDE_DIRECTORIES
- **SET_TARGET_PROPERTIES**
- ENABLE_TESTING/ADD_TEST

➢ SET_TARGET_PROPERTIES

- 设置目标的一些属性来改变它们构建的方式。
- 命令格式为：

```
set_target_properties(target1 target2 ...
                      PROPERTIES prop1 value1
                      prop2 value2 ...)
```

```
set_target_properties(exampleCv
PROPERTIES
ARCHIVE_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/lib"
LIBRARY_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/lib"
)
```

3. CMake的常用指令及变量

➢ CMake基本常用指令

- ADD_DEFINITIONS
- OPTION
- ADD_CUSTOM_TARGET
- ADD_DEPENDENCIES
- INSTALL
- TARGET_INCLUDE_DIRECTORIES
- SET_TARGET_PROPERTIES
- **ENABLE_TESTING/ADD_TEST**

➢ ENABLE_TESTING/ADD_TEST

- [enable_testing] :用来控制Makefile是否构建test目标。
- [add_test] : 一般需要和enable_testing()配合使用
- 命令格式为：

  ADD_TEST(testname Exename arg1 arg2 ...)
- 生成makefile后可用make test执行测试

```
ADD_TEST(mytest ${PROJECT_BINARY_DIR}/bin/main)
ENABLE_TESTING()
```

3. CMake的常用指令及变量

- **CMake基本常用变量**

变量名	具体含义
CMAKE_INSTALL_PREFIX	构建install的路径
$ENV{HOME}	HOME环境下的目录路径
PROJECT_NAME	工程名变量
<PKG>_INCLUDE_DIR	导入包头文件全路径
<PKG>_LIBRARIES	导入库文件的全路径
PROJECT_SOURCE_DIR	构建工程的全路径
CMAKE_VERSION	Cmake的版本号
CMAKE_SOURCE_DIR	源码树的顶层路径

4.实践:从简单CMake说起

4. 实践：从简单的CMake说起

实践一：生成一个简单的Hello_CV工程实例

4. 实践：从简单的CMake说起

- 实践一：生成一个简单的Hello_CV工程实例

hello_cv_1.h

```
#ifndef HELLO_CV_1_H_
#define HELLO_CV_1_H_
#include <string>
#include <iostream>

namespace hello_cv_1
{
class HelloCv_1
{
    public:
      HelloCv_1();
      ~HelloCv_1();
      std::string getMsgsContent(std::string say_something);
    private:
      std::string say_something;
};
}
#endif
```

main.cpp

```
#include "hello_cv_1.h"
#include <string>
#include <iostream>

int main()
{
    hello_cv_1::HelloCv_1 helloCv_1_;

    std::cout << helloCv_1_.getMsgsContent("Hello_OpenCV") << std::endl;
    return 0;
}
```

- 源文件源码

hello_cv_1.cpp

```
#include "hello_cv_1.h"
#include <string>
#include <iostream>

namespace hello_cv_1
{
HelloCv_1::HelloCv_1()
{
}
HelloCv_1::~HelloCv_1()
{
}

std::string HelloCv_1::getMsgsContent(std::string say_something)
{
    return say_something;
}
}
```

4. 实践：从简单的CMake说起

➢ 实践一：生成一个简单的Hello_CV工程实例

◘ 工程文件目录

- 采用外部编译
- ✓ 外部编译 && 内部编译
- 文件位置存放规则
- ✓ src && include && build

```
build
CMakeLists.txt
include
└── hello_cv_1.h
src
├── hello_cv_1.cpp
└── main.cpp
```

◘ CMakeLists.txt 文件

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_1)

add_compile_options(-std=c++11)

include_directories(include)
add_executable(hello_cv_1 src/main.cpp src/hello_cv_1.cpp)
```

```
mkdir build && cd build  # 建立build并进入到该文件夹
cmake ..                 # 生成makefile文件
make                     # 编译makefile，并生成bin文件
./hello_cv_1             # 执行文件后输出"Hello_OpenCV"
```

4. 实践：从简单的CMake说起

实践二：为Hello_CV实例添加一个库

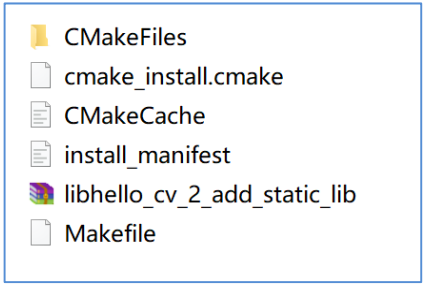
4. 实践：从简单的CMake说起

- 实践二：为Hello_CV实例添加一个库

- CMakeLists.txt 文件

```
cmake_minimum_required(VERSION 2.8.3)

project(hello_cv_2_add_static_lib)

add_compile_options(-std=c++11)

include_directories(include)

# add_executable(hello_cv_2 src/main.cpp src/hello_cv_1.cpp)

add_library(hello_cv_2_add_static_lib STATIC src/hello_cv_1.cpp)
```

```
STATIC    # 静态库，在链接其他目标时使用
SHARED    # 动态链接库，运行时加载
MODULE    # 运行时使用dlopen-系列的函数动态链接
```



4. 实践：从简单的CMake说起

实践三：为实例同时添加一个动/静态库

4. 实践：从简单的CMake说起

◘ 生成指定路径的动态库

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_3_add_shared_lib)

add_compile_options(-std=c++11)

# 设置库文件的输出目录,默认输出到执行目录空间下
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)    # 工程的根目录
include_directories(include)

# 静态库名为XXX.a 动态库名为XXX.so
add_library(hello_cv_3_lib SHARED src/hello_cv_1.cpp)
```

◘ 生成库的说明：

- 在cmakelist同级目录的lib文件夹下生成
- Windows中静态库.lib后缀,动态库为.dll
- linux中静态库.a后缀，动态库.so后缀

4. 实践：从简单的CMake说起

- 一点思考：同时构建静态和动态的同名库？？？

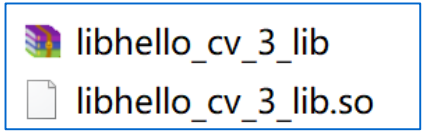
```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_3_add_shared_lib)

add_compile_options(-std=c++11)

# 设置库文件的输出目录,默认输出到执行目录空间下
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)    # 工程的根目录
include_directories(include)

# 静态库名为XXX.a 动态库名为XXX.so
add_library(hello_cv_3_lib SHARED src/hello_cv_1.cpp)

# 如果同时生成动静态库,可以再次使用add_library?
add_library(hello_cv_3_lib STATIC src/hello_cv_1.cpp)
```

```
CMake Error at CMakeLists.txt:14 (add_library):
  add_library cannot create target "hello_cv_3_lib" because another target
  with the same name already exists.  The existing target is a shared library
  created in source directory
  "/media/henry_pan/Data/slam_research_club/project/src/hello_cv_3_add_shared_li
b".
```

4. 实践：从简单的CMake说起

- 一点思考：同时构建静态和动态的同名库？？？

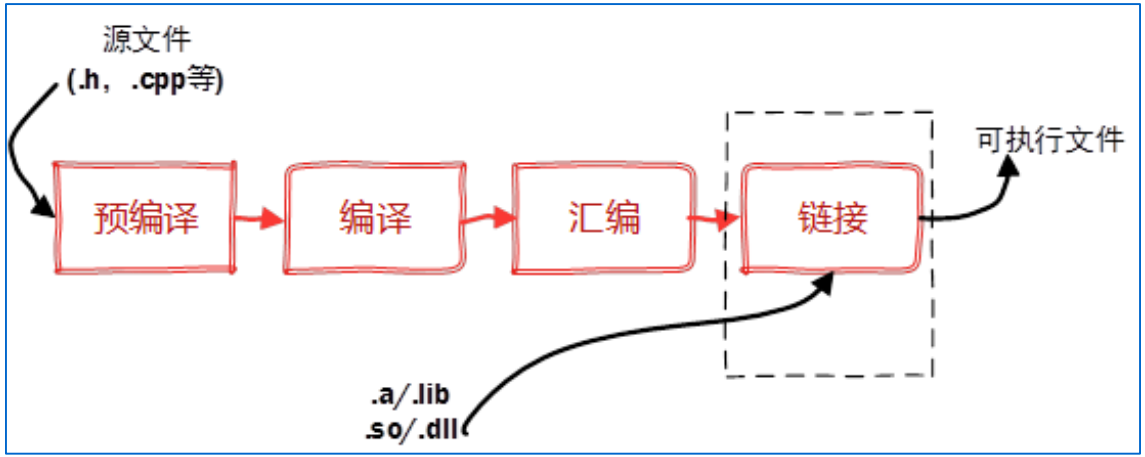
```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_3_add_shared_lib)

add_compile_options(-std=c++11)

# 设置库文件的输出目录,默认输出到执行目录空间下
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)   # 工程的根目录
include_directories(include)

# 静态库名为XXX.a 动态库名为XXX.so
add_library(hello_cv_3_lib SHARED src/hello_cv_1.cpp)

# 如果同时生成动静态库,可以再次使用add_library?
# add_library(hello_cv_3_lib STATIC src/hello_cv_1.cpp)

add_library(hello_cv_3_add_static_lib STATIC src/hello_cv_1.cpp)
# 设置文件名替换,注意和上一行的先后顺序
set_target_properties(hello_cv_3_add_static_lib PROPERTIES OUTPUT_NAME "hello_cv_3_lib")
```



到底什么是静态库和动态库？？？

4. 实践：从简单的CMake说起

- 从程序的工作过程说起



静态库、动态库的区别来自【链接阶段】如何处理库，链接成可执行的程序。

4. 实践：从简单的CMake说起

- 静态库

➢ **定义：**

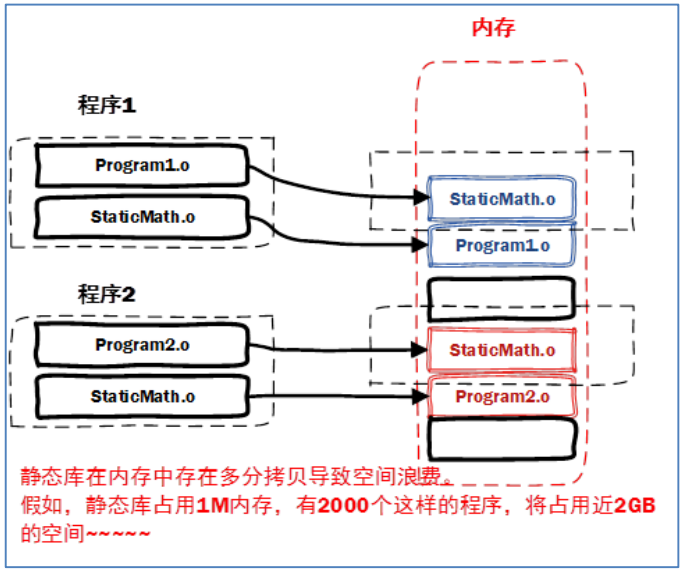
- **链接阶段**，库中目标文件所含的所有将被程序使用的函数的机器码，被copy到最终的可执行文件中。**因此对应的链接方式称为静态链接。**

➢ 特点：

- 静态库对函数库的链接是放在**编译时期**完成的；
- 程序在运行时与函数库再无瓜葛，**移植方便**；
- 运行**效率相对快**；
- **占用磁盘和内存空间**，因为所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。

4. 实践：从简单的CMake说起

- 静态库有局限性吗？？？



静态库链接方式

- 从静态库局限性来看：
- **空间浪费**是静态库的一个问题；
- 静态库对程序的更新、部署和发布页会带来麻烦；
- 如果静态库lib更新了，所以使用它的应用程序都需要重新编译、发布给用户；
- 对于玩家来说，可能是一个很小的改动，却导致整个程序重新下载，**全量更新**。

4. 实践：从简单的CMake说起

- 动态库

➢ 定义：

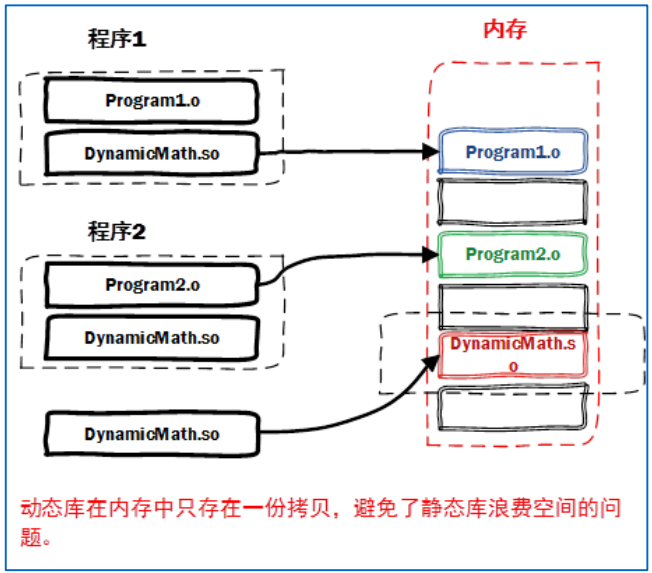
- 程序编译时并不会被连接到目标代码中，而是在**程序运行是才被载入**。

➢ 特点：

- 可执行文件只包含它需要的函数的**引用表**，而不是所有的函数代码；
- 只有在程序执行时，那些**需要的函数代码**才被拷贝到内存中。
- 动态库在**程序运行是才被载入**，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，**增量更新**。

4. 实践：从简单的CMake说起

◘ 动态库的特点



动态库链接方式

➢ **动态库的特点：**

- ◘ 对库函数的链接载入推迟到程序运行的时期；
- ◘ 实现进程之间的**资源共享**，也称为共享库；
- ◘ 一些程序升级变得简单，**增量更新**；
- ◘ 但依赖的模块也要存在，如果使用载入时动态链接，程序启动时发现DLL不存在，系统将终止程序并给出错误信息，**依赖性强**。

4. 实践：从简单的CMake说起

画芝士点啦

- 一句话总结：

- **静态库**是牺牲了空间效率，换取了**时间效率**；
- **共享库**是牺牲了时间效率，换取了**空间效率**；

- 工程应用

- 一般在比较固定参数的工程项目中，如底层驱动逻辑比较稳定，会考虑静态库的使用；
- 在需要多次调试优化版本中，比如PUBG（吃鸡）的游戏平衡体验更新，会考虑动态库的使用。

转自[C++静态库与动态库(吴秦)] https://www.cnblogs.com/skynet/p/3372855.html

4. 实践：从简单的CMake说起

知识回顾：

- 我们前三个实例都干了些啥？？？
- 接下来我们还需要讲啥？？？

4. 实践：从简单的CMake说起

实践四：为实例添加一个内部的链接库

4. 实践：从简单的CMake说起

链接一个内部库的依赖

```
# 内部引入动态库的形式
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_4_target_link_inner_lib)

add_compile_options(-std=c++11)

# 设置库文件的输出目录,默认输出到执行目录空间下
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)   # 工程的根目录
set(LIBRARY_OUTPUT_NAME ${PROJECT_NAME})
include_directories(include)

# 静态库名为XXX.a 动态库名为XXX.so
add_library(hello_cv_4_lib SHARED src/hello_cv_1.cpp)

add_executable(${PROJECT_NAME} src/main.cpp)

target_link_libraries(${PROJECT_NAME} hello_cv_4_lib)
```

想一想？？？为什么要有内部链接库

```
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/lib)

add_library(${PROJECT_NAME} SHARED
src/System.cc
src/Tracking.cc
src/LocalMapping.cc
src/LoopClosing.cc
src/ORBextractor.cc
src/ORBmatcher.cc
src/FrameDrawer.cc
src/Converter.cc
src/MapPoint.cc
src/KeyFrame.cc
src/Map.cc
src/MapDrawer.cc
src/Optimizer.cc
src/PnPsolver.cc
src/Frame.cc
src/KeyFrameDatabase.cc
src/Sim3Solver.cc
src/Initializer.cc
src/Viewer.cc
)
```

多个程序生成一个动态库

4. 实践：从简单的CMake说起

实践五：为实例添加一个外部的链接库

4. 实践：从简单的CMake说起

◘ 如何导入一个外部链接库

➢ 链接库导入的是什么？

- 头文件
- 库文件

➢ 导入库文件的几点尝试

- 1. 利用绝对路径的方式导入
- 2. 利用Install的方式导入
- 3. 利用find_package的方式导入

4. 实践：从简单的CMake说起

- 利用绝对路径的方式导入

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_5_target_link_outer_lib)
add_compile_options(-std=c++11)

# 方法一:已知目标库的位置，直接将头文件和库文件导入到工程中
set(dpd_lib
    /home/henry_pan/ws_psl/slam_research_club/project/src/hello_cv_2_add_static_lib)
include_directories(${dpd_lib}/include)
add_executable(${PROJECT_NAME} src/main.cpp)
target_link_libraries(${PROJECT_NAME} ${dpd_lib}/lib/libhello_cv_2_add_static_lib.a)
```

- 方法分析：
- ✓ 通过导入绝对路径的方法，可以生成可执行文件；
- ✓ 依赖路径必须是绝对路径，可移植性和通用性不强；
- ✓ 导入方法太**笨**，有没有更有效的方式来管理库的依赖？？？

4. 实践：从简单的CMake说起

- 利用INSTALL的方式导入

库文件安装

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_7_install_lib)
add_compile_options(-std=c++11)
include_directories(include)
add_library(hello_cv_7_install_lib STATIC src/hello_cv_1.cpp)

#方法二：将头文件和库文件安装到/usr目录下
set(CMAKE_INSTALL_PREFIX /usr)
# 如不指定，默认安装路径会安装到/usr/local
message(${CMAKE_INSTALL_PREFIX})
install(FILES include/hello_cv_1.h DESTINATION include)
install(TARGETS hello_cv_7_install_lib
        # 将静态库安装到${CMAKE_INSTALL_PREFIX}/lib目录下
        ARCHIVE DESTINATION lib
        )
```

库文件导入

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_8_find_outer_lib)
add_compile_options(-std=c++11)

# 方法二:将头文件和库文件安装到/usr目录下（导入库）
include_directories(/usr/include)
add_executable(${PROJECT_NAME} src/main.cpp)
target_link_libraries(${PROJECT_NAME}
    /usr/lib/libhello_cv_2_add_static_lib.a
    )
```

- 方法分析：
- ✓ 对于库文件，头文件等一般安装放在/usr或者/usr/local中，导入比较方便；
- ✓ 有没有更通用的方式，来进行库文件的自动查找？？？

4. 实践：从简单的CMake说起

◘ 如何使用find_package----从原理说起

- 由于cmake**不提供搜索库**的方法，不会对库本身的环境变量进行设置（**需要手动配置**）；
- 一般称为**模块模式**和**配置模式**，顾名思义分为module和config；
- 指令按照**优先级顺序**在指定路径查找**Findxxx.cmake**和**xxxConfig.cmake**；
- cmake能够找到这两个文件中的**任何一个**，我们都能成功使用该库。

常用变量	常用指令	含义解释
<NAME>_FOUND	If(<NAME>_FOUND)	找到库的标志
<NAME>_INCLUDE_DIR <NAME>_INCLUDES	find_path(<NAME>_INCLUDES)	库的头文件路径
<NAME>_LIBRARY <NAME>_LIBS	find_library(<NAME>_LIBS)	库文件路径

4. 实践：从简单的CMake说起

◘ 利用FIND_PACKAGE方式-------【模块模式】

➢ FindXXX.cmake文件定义

第一页（共两页）

```
find_path(hello_cv_7_install_lib_INCLUDE_DIR
  NAMES hello_cv_1.h
  # <必须绝对路径,推荐安装在"/usr/share/cmake/Modules/"目录下">
  PATHS "/media/henry_pan/Data/slam_research_club/project/src/hello_cv_7_install_lib/install/include/")
find_library(hello_cv_7_install_lib_LIBRARY
  # names也可以指定具体文件名libhello_cv_7_install_lib.a
  NAMES hello_cv_7_install_lib      # 思考题:同时生成.a和.so,那么lib链接谁 ???
  PATHS "/media/henry_pan/Data/slam_research_club/project/src/hello_cv_7_install_lib/install/lib/")

IF (hello_cv_7_install_lib_INCLUDE_DIR AND hello_cv_7_install_lib_LIBRARY)
    SET(hello_cv_7_install_lib_FOUND TRUE)
ENDIF (hello_cv_7_install_lib_INCLUDE_DIR AND hello_cv_7_install_lib_LIBRARY)
```

- 小贴士：通过**打印${XXX_INCLUDE_DIR}和${XXX_LIBRARY}的变量来debug**是否输出正确；

4. 实践：从简单的CMake说起

利用FIND_PACKAGE方式-------【模块模式】

- FindXXX.cmake文件定义

```
IF (hello_cv_7_install_lib_FOUND)
    # 指定了QUIET选项,如果没有找到包配置文件，CMake将会生成一个warnning
    IF (NOT hello_cv_7_install_lib_FIND_QUIETLY)
        MESSAGE(STATUS "Found hello_cv_7_install_lib:
                    ${hello_cv_7_install_lib_LIBRARY}")
    ENDIF (NOT hello_cv_7_install_lib_FIND_QUIETLY)
ELSE (hello_cv_7_install_lib_FOUND)
    # 指定了REQUIRED选项,如果没有找到包配置文件，CMake将会停止编译产生错误
    IF (hello_cv_7_install_lib_FIND_REQUIRED)
        MESSAGE(FATAL_ERROR "Could not find hello_cv_7_install_lib library")
    ENDIF (hello_cv_7_install_lib_FIND_REQUIRED)
ENDIF (hello_cv_7_install_lib_FOUND)
```

4. 实践：从简单的CMake说起

◘ 利用FIND_PACKAGE方式-------【模块模式】

➢ 库文件的生成

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_7_install_lib)
add_compile_options(-std=c++11)
include_directories(include)
add_library(hello_cv_7_install_lib STATIC src/hello_cv_1.cpp)

#方法三：利用find_package,自定义FindXXX.cmake生成库
set(CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake)
set(CMAKE_INSTALL_PREFIX ${PROJECT_SOURCE_DIR}/install)
message(${CMAKE_MODULE_PATH})
message(${CMAKE_INSTALL_PREFIX})
install(FILES cmake/Findhello_cv_7_install_lib.cmake DESTINATION cmake)
install(FILES include/hello_cv_1.h DESTINATION include)
install(TARGETS hello_cv_7_install_lib
        # 将静态库安装到${CMAKE_INSTALL_PREFIX}/lib目录下
        ARCHIVE DESTINATION lib
        )
```

◘ 方法分析：

✓ 设置指定的cmake文件生成目录

✓ 设置install的生成目录，统一管理三个文件；

✓ 外部导入文件只需知道cmake路径即可；

4. 实践：从简单的CMake说起

利用FIND_PACKAGE方式-------【模块模式】

➢ 库文件的导入

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_8_find_outer_lib)
add_compile_options(-std=c++11)
# 方法三:利用find_package查找
set(CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/../hello_cv_7_install_lib/install/cmake/)
find_package(hello_cv_7_install_lib REQUIRED)
if(hello_cv_7_install_lib_FOUND)
    include_directories(${hello_cv_7_install_lib_INCLUDE_DIR})
    add_executable(${PROJECT_NAME} src/main.cpp)
    target_link_libraries(${PROJECT_NAME}
                          ${hello_cv_7_install_lib_LIBRARY})
else(hello_cv_7_install_lib_FOUND)
    message(FATAL_ERROR "hello_cv_7_install_lib library not found")
endif(hello_cv_7_install_lib_FOUND)
```

- 思考：什么时候可以用相对路径，什么时候可以用绝对路径？？？

Note：

✓ 由于本实例设置了自定义路径，需要指定cmake_module_path；

✓ 默认下find_package会/usr/share/cmake/module等路径下各个包目录中查找Find*.cmake文件

- 回顾：模块模式的三个步骤？？？如果不想手动写FindXXX.cmake？？？

4. 实践：从简单的CMake说起

利用FIND_PACKAGE方式-------【配置模式】

- 库文件的生成

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_2_add_static_lib)
add_compile_options(-std=c++11)
include_directories(include)
add_library(hello_cv_2_add_static_lib STATIC src/hello_cv_1.cpp)
# 设置install安装目录,类似于cmake -DCMAKE_INSTALL_PREFIX指令
set(CMAKE_INSTALL_PREFIX ${PROJECT_SOURCE_DIR}/install)

# 方法三:设置自定义的install路径
target_include_directories(hello_cv_2_add_static_lib PUBLIC                    # 设置库的属性
                           $<BUILD_INTERFACE:${CMAKE_SOURCE_DIR}/include>      # 源文件安装位置
                           $<INSTALL_INTERFACE:include>)                       # 安装文件夹位置
```

4. 实践：从简单的CMake说起

利用FIND_PACKAGE方式-------【配置模式】

- 库文件的生成

```
# 设置目标include的头文件的属性
set_target_properties(hello_cv_2_add_static_lib
                      PROPERTIES
                      PUBLIC_HEADER "include/hello_cv_1.h")

install(TARGETS hello_cv_2_add_static_lib
        EXPORT hello_cv_2_add_static_lib-targets   # 供外部库使用的声明
        PUBLIC_HEADER DESTINATION include
        ARCHIVE DESTINATION lib         # 将静态库安装到${CMAKE_INSTALL_PREFIX}/lib目录下
        )
install(EXPORT hello_cv_2_add_static_lib-targets
        NAMESPACE hello_cv_2_add_static_lib::
        # 外部要使用find_package一定要使用,命名必须XX-congfig.cmake
        FILE hello_cv_2_add_static_lib-config.cmake
        DESTINATION lib/cmake/hello_cv_2_add_static_lib)
```

4. 实践：从简单的CMake说起

利用FIND_PACKAGE方式-------【配置模式】

- 库文件的导入

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_5_target_link_outer_lib)
add_compile_options(-std=c++11)
# 方法三:利用find_package查找(配置模式)
set(CMAKE_PREFIX_PATH ${PROJECT_SOURCE_DIR}/../hello_cv_2_add_static_lib/install)
find_package(hello_cv_2_add_static_lib REQUIRED)
if(hello_cv_2_add_static_lib_FOUND)
    add_executable(${PROJECT_NAME} src/main.cpp)
    target_link_libraries(${PROJECT_NAME}
                          hello_cv_2_add_static_lib::hello_cv_2_add_static_lib)
else(hello_cv_2_add_static_lib_FOUND)
    message(FATAL_ERROR "hello_cv_2_add_static_lib library not found")
endif(hello_cv_2_add_static_lib_FOUND)
```

- 注意：两种模式下设置的环境变量的区别

4. 实践：从简单的CMake说起

知识回顾：

- 我们后两个实例都干了些啥？？？
- 基本上Cmake的基本语法就已经介绍完毕
- 完事了嘛？别急，最后一个彩蛋...

4. 实践：从简单的CMake说起

- 更简便的cmake模块构建工具？？？
  - 回顾实例四，使用自定义find_package不是自定义构建模块，就是配置config文件（麻烦）
  - 介绍一种基于cmake语法的上层编译工具**—catkin工具**
- catkin工具基本简介
  - 安装以ubuntu16.04为例(apt安装)

```
sudo apt-cache search cmake    #搜索可用的cmake安装包
sudo apt-get install cmake     #也可以源码安装
sudo apt-cache search catkin
sudo apt-get install python-catkin-pkg python-catkin-tools
```

4. 实践：从简单的CMake说起

◘ catkin工具基本简介

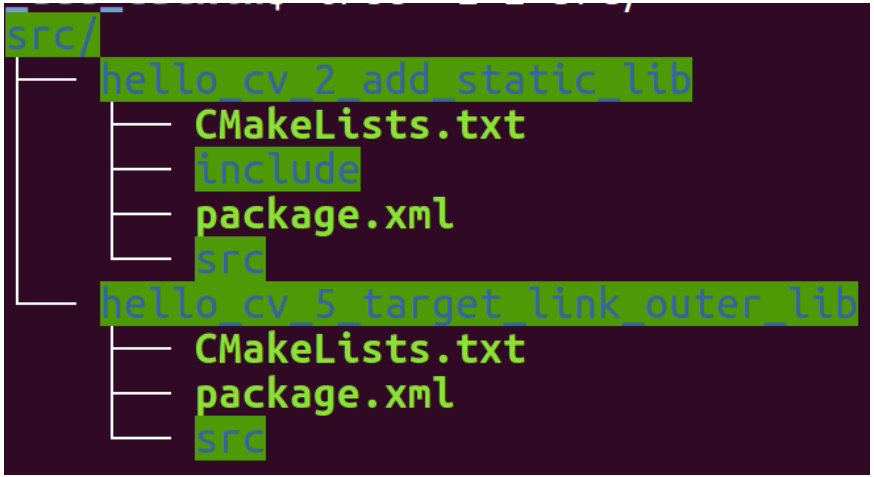
- ➢ 如何管理pkg和包之间的相互关系

A. Pkg的目录结构解释（pro2库，pro5导入库）

B. 利用指令`catkin create --pkg XXX`

C. 自动生成CMakeLists和package文件

D. 两个文件的作用解释



4. 实践：从简单的CMake说起

- 利用catkin管理项目实例
  - pro2生成库

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_2_add_static_lib)
add_compile_options(-std=c++11)
find_package(catkin REQUIRED)
catkin_package(
               INCLUDE_DIRS include
               LIBRARIES hello_cv_2_add_static_lib
               #  DEPENDS system_lib # 如果依赖其他库添加
              )
include_directories(include ${catkin_INCLUDE_DIRS})
add_library(${PROJECT_NAME} STATIC src/hello_cv_1.cpp )
```

```
<?xml version="1.0"?>
<package format="2">
  <name>hello_cv_2_add_static_lib</name>
  <version>0.0.0</version>
  <description>The hello_cv_2_add_static_lib package</description>

  <maintainer email="henry_pan@todo.todo">henry_pan</maintainer>

  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <export>

  </export>
</package>
```

4. 实践：从简单的CMake说起

- 利用catkin管理项目实例
  - pro5导入库

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_cv_5_target_link_outer_lib)
add_compile_options(-std=c++11)
find_package(catkin REQUIRED
             hello_cv_2_add_static_lib)
include_directories(include ${catkin_INCLUDE_DIRS} )
add_executable(${PROJECT_NAME} src/main.cpp)
target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES} )
```

```
<?xml version="1.0"?>
<package format="2">
  <name>hello_cv_5_target_link_outer_lib</name>
  <version>0.0.0</version>
  <description>The hello_cv_5_target_link_outer_lib package</description>

  <maintainer email="henry_pan@todo.todo">henry_pan</maintainer>

  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <depend>hello_cv_2_add_static_lib</depend>

  <export>

  </export>
</package>
```

4. 实践：从简单的CMake说起

利用catkin管理项目实例

- 如何使用catkin

```
cd hello_cv_6_add_catkin    # 进入ubuntu的文件目录下
catkin build                # 构建完毕后生成三个文件夹
```

- 生成build，devel，和log文件夹

  - 对应lib生成在/devel下
  - 对应bin生成在/build下

- Catkin工具工程实例

  - ROS1.0[机器人操作系统]是基于catkin工具进行管理，更以实现pkg的管理与依赖。

什么是「 SLAM研习社」？

- 计算机视觉life读者自发组织，专注于SLAM的开源学习组织
- 宗旨：SLAM技术从基础到应用，懂原理会应用，完全搞透彻，不留死角
- 角色：主讲、嘉宾、联络、宣传、策划
- 报名：simiter@126.com
- 报名主讲/嘉宾不需要你是大牛，只要熟悉SLAM某个知识点即可
- SLAM研习社详细介绍、直播预告、视频回放链接
- https://github.com/electech6/LearnSLAM/blob/master/README.md

在线问答

感谢各位聆听

Thanks for Listening