

cmake(五)Cmake构建静态库和动态库

原创

wzj_110

于 2021-04-16 07:59:59 发布


阅读量1.3k

收藏 3

点赞数 5

分类专栏: cmake DSL语言

版权

cmake DSL语言 专栏收录该内容

38 篇文章

已订阅

一 基本语法

CMake中要生成静态库或动态库，可以使用 `add_library` 指令（函数），该指令使用方法：

```
add_library(lib_name [SHARED|STATIC|MODULE]
                  [EXCLUDE_FROM_ALL] src1 src2 ... srcN)
```

参数说明：

- `lib_name`：指定生成库的名称
- `SHARED`：生成动态库 dynamic
- `STATIC`：生成静态库 static
- `MODULE`：使用 `dyld`（the dynamic link editor，苹果的动态链接器）的系统有效，如果不支持 `dyld`，则为 `SHARED` 默认不会自动构建
- `EXCLUDE_FROM_ALL`：指定此参数，则该库默认情况下不被构建，当有其它组件用到该库或手动构建时，才被构建
- `src1 src2 ... srcN`：用于构建库用到的源文件 显示声明使用的时候才构建

二 构建动态库

目的： 构建一个'库'供'他人'使用

① 新建目录演示

```
kiosk@k8s build $ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile
kiosk@k8s build $ rm -fr *
kiosk@k8s build $ cd ../../
kiosk@k8s CmakeProjects $ ls
HelloCmake
kiosk@k8s CmakeProjects $ mkdir HelloLibrary
kiosk@k8s CmakeProjects $ cd HelloLibrary/
kiosk@k8s HelloLibrary $
```

两个目录并行的

操作目录

https://blog.csdn.net/wzj_110

② 添加头文件

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 1) 我们一般在'.h后缀'的'头文件'里面只放入'函数声明'、'宏定义'、'函数原型'
- h -->'header'
- 2) 而'具体的实现'在'.cpp文件'里面
- 3) 在'编译(compile)'的时候,编译器会'自动加载'和'.h'匹配的'.cpp'文件

#ifndef的用法

1) 命名规则

核心： '独一无二'的,不要冲突

```

23 #ifndef _MATH_H
24 #define _MATH_H 1
25
26 #include <features.h>
27
28 _BEGIN_DECLS
"/usr/include/math.h" 491 lines --5%-- 28,1

```

2) 如何编写头文件

```

/*
 * 1) 防止一个源文件两次包含同一个头文件
 *
 * 2) 而不是防止两个源文件包含同一个头文件
 */

/*
 * #ifndef <标识>
 *

```

如何编写自己的C语言头文件

一些初学C语言的人，不知道头文件（*.h文件）原来还可以自己写的。只知道调用系统库函数时，要使用#include语句将某些头文件包含进去。其实，头文件跟C文件一样，是可以自己写的。头文件是一种文本文件，使用文本编辑器将代码编写好之后，以扩展名.h保存就行了。头文件中一般放一些重复使用的代码，例如函数声明，变量声明，常数定义，宏的定义等等。当使用#include语句将头文件引用时，相当于将头文件中所有内容，复制到#include处。为了避免因为重复引用而导致的编译错误，头文件常具有：

```

#ifndef _DELAY_H_
#define _DELAY_H_
//代码部分
#endif

```

的格式。

独一无二的

其中，DELAY为一个唯一的标号，命名规则跟变量的命名规则一样。常根据它所在的头文件名来命名，例如，如果头文件的文件名叫做stc15w.h，那么可以这样使用：

```

#ifndef _STC15W_H_
#define _STC15W_H_
//代码部分
#endif

```

这样写的意思就是，如果没有定义 __STC15W_H__，则定义 __STC15W_H__，并编译下面的代码部分，直到遇到#endif。这样，当重复引用时，由于 __STC15W_H__ 已经被定义，则下面的代码部分就不会被编译了，这样就避免了重复定义。另外，使用#include时，使用引号"与尖括号<>的意思是不一样的。使用引号"时，首先搜索工程文件所在目录，然后再搜索编译器头文件所在目录；而使用尖括号<>时，刚好是相反的顺序。

细节

假设我们有两个文件名一样的头文件stc15w.h，但内容却是不一样的。一个保存在编译器指定的头文件目录下，我们把它叫做文件I；另一个则保存在当前工程的目录下，我们把它叫做文件II。如果我们使用的是#include<>则我们引用到的是文件I。如果我们使用的是#include "stc15w.h"，则我们引用的将是文件II。

3) 最终效果

```

8  * #ifndef <标识>
9  *
10 * 含义：if not defined sth
11 *
12 * 标识的命名规则：一般是头文件名全大写，前后加下划线，并把文件名中的"."也变成下划线
13 *
14 * main.h --> _MAIN_H
15 *
16 * */
17 #ifndef _HELLOLIBRARY_H
18 #define _HELLOLIBRARY_H
19
20 #include <iostream>
21 void hello_library();
22
23 #endif
24
HelloLibrary.h" [Modified] 24 lines --58%--

```

代码部分

声明一个函数

名字不太规范，一般都是小写

③ 头文件的具体实现

头文件对应的源文件，实现头文件里声明的函数，简单的输出字符串

'注意': 'hello_lirary'函数重命名为'hello_library'

```

1 #include <stdlib.h>
2 #include "HelloLibrary.h"
3
4 void hello_lirary() {
5     std::cout << "Hello SHared Library! -- 共享库" << std::endl;
6 }

```

注意：“双引号”和“<>”的区别

先在当前的工程下找,没有则从系统头文件库找

要和.h文件中函数的声明一致

同文件名不同后缀

"HelloLibrary.cpp" [Modified][New file] 6 lines --100%-- 6,1 All

④ 主程序

目的：调用‘动态生成库中定义’的hello_lirary函数

```

1 #include "HelloLibrary.h"
2
3 int main() {
4     /*先include进来,然后调用动态库的函数*/
5     hello_lirary();
6     return EXIT_SUCCESS;
7 }

```

主程序

"Main.cpp" 7 lines --14%-- 1,1 All

⑤ 编辑CMakeLists.txt文件

‘注意’：add_library()函数中‘HelloLirary.cpp’应为‘HelloLibrary.cpp’ -->少了一个字符‘b’

```

1 # 1) cmake最低版本
2 cmake_minimum_required(VERSION 3.8)
3 # 2) 项目的名称
4 project(HelloLibrary)
5 # 3) 基于HelloLibrary.cpp生成SHARED动态库
6 add_library(hello_library SHARED HelloLibrary.cpp)
7 # 4) 当前项目源码增加到系统头文件的路径中,方便后续查找 -->统一用<>方式
8 include_directories(${PROJECT_SOURCE_DIR})
9 # 5) 基于Main.cpp生成最终的二进制可执行文件hello_main
10 add_executable(hello_main Main.cpp)
11 # 6) 指定目标文件(生成的二进制所依赖的库(前面生成的共享库))
12 target_link_libraries(hello_main hello_library)

```

生成的共享库为下面做准备

"CMakeLists.txt" 12L, 587C

cmake_minimum_required指令，
用于**检测CMake的版本**，指定**最小版本至少为3.8**

思考：如果cmake版本过低呢？

add_library指令，指定库名称为hello_library，由于指明生成SHARED库（动态库），实际生成的库文件名为libhello_library.so，用于生成该库的源文件为HelloLibrary.cpp

https://blog.csdn.net/wzj_110

include_directories指令，用于增加包含头文件的路径，这里是把项目源目录增加到包含头文件的路径中

类似于把某个环境变量加入到PATH中

https://blog.csdn.net/wzj_110

add_executable指令，用于指定生成可执行文件，生成的可执行文件名为hello_main，用到的源文件为Main.cpp

https://blog.csdn.net/wzj_110

target_link_libraries指令，指定某个目标的生成所依赖的库，这里要生成的可执行文件hello_main要依赖hello_library这个库，也就是add_library指令生成的那个库

https://blog.csdn.net/wzj_110

```
kiosk@k8s HelloLibrary $ ls
CMakeLists.txt  HelloLibrary.cpp  HelloLibrary.h  Main.cpp
kiosk@k8s HelloLibrary $
```

CMake的工程文件 源文件：生成库 头文件：声明函数 生成二进制可执行文件

实验：编写这四个文件

⑥ 采用外部编译

采用外部编译，新建一个build目录

执行cmake ..，CMakeLists.txt文件位于..目录，即上一层目录，生成Makefile

```

kiosk@k8s HelloLibrary $ mkdir build
kiosk@k8s HelloLibrary $ cd build/
kiosk@k8s build $ cmake ..
CMake Error at CMakeLists.txt:2 (cmake_minimum_required):
  CMake 3.8 or higher is required. You are running version 2.8.12.2

第一次报错原因: cmake默认采用2.8版本,版本太低

-- Configuring incomplete, errors occurred!
kiosk@k8s build $ cmake3 ..
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is GNU 4.8.5
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc - works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ - works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
CMake Error at CMakeLists.txt:6 (add_library):
  Cannot find source file:
    HelloLirary.cpp

实际文件: HelloLibrary.cpp 文件名写错了

Tried extensions .c .C .c++ .cc .cpp .cxx .cu .m .M .mm .h .hh .h++ .hm
.hpp .hxx .in .txx

CMake Error at CMakeLists.txt:6 (add_library):
  No SOURCES given to target: hello_library

CMake Generate step failed. Build files cannot be regenerated correctly.

```

生成cmake的'工程文件'CMakeLists.txt文件

⑦ 修改CmakeLists错误

cmake3 ../CmakeLists --> make Makefile('编译') -->生成'这里是动态库'和'二进制可执行文件'

```

kiosk@k8s build $ cmake3 ..
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is GNU 4.8.5
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc - works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ - works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /var/ftp/pub/pub/cmake/test/CmakeProjects/HelloLibrary/build
kiosk@k8s build $ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile

```

有了Makefile，执行make命令，生成库和可执行程序

collect2: error: ld returned 1 exit status 解决方案

(.text+0x13): undefined reference to

- 1 备注: 报错的'原因'--> 'HelloLibrary.cpp'中函数的定义有问题
- 2
- 3 本来应该是'hello_library()',实际'错误'的定义为'hello_lirary()'


```
kiosk@k8s build $ make 报错
Scanning dependencies of target hello_library
[ 25%] Building CXX object CMakeFiles/hello_library.dir/HelloLibrary.cpp.o
[ 50%] Linking CXX shared library libhello_library.so
[ 50%] Built target hello_library
Scanning dependencies of target hello_main
[ 75%] Building CXX object CMakeFiles/hello_main.dir/Main.cpp.o
[100%] Linking CXX executable hello_main
CMakeFiles/hello_main.dir/Main.cpp.o: In function `main':
Main.cpp:(.text+0x5): undefined reference to `hello_library()'
collect2: error: ld returned 1 exit status
make[2]: *** [hello_main] Error 1
make[1]: *** [CMakeFiles/hello_main.dir/all] Error 2
make: *** [all] Error 2
```

https://blog.csdn.net/wzj_110

⑧ 修改HelloLibrary.cpp错误

```
kiosk@k8s HelloLibrary $ cd build/
kiosk@k8s build $ ls
kiosk@k8s build $ cmake3 注意：所处的位置
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is GNU 4.8.5
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc - works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ - works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /var/ftp/pub/pub/cmake/test/CmakeProjects/HelloLibrary/build
kiosk@k8s build $ ls
CMakeCache.txt CMakeFiles cmake_install.cmake Makefile
kiosk@k8s build $ make
Scanning dependencies of target hello_library
[ 25%] Building CXX object CMakeFiles/hello_library.dir/HelloLibrary.cpp.o
[ 50%] Linking CXX shared library libhello_library.so
[ 50%] Built target hello_library
Scanning dependencies of target hello_main
[ 75%] Building CXX object CMakeFiles/hello_main.dir/Main.cpp.o
[100%] Linking CXX executable hello_main
[100%] Built target hello_main
kiosk@k8s build $ ls
CMakeCache.txt CMakeFiles cmake_install.cmake
kiosk@k8s build $
```

对比cmake3和make文件的差异

so结尾的文件表示动态库

二进制可执行文件

make多出来的两个文件

https://blog.csdn.net/wzj_110

细节： '动态库'会在前面'自动'加上'lib'的前缀'

⑨ 测试二进制可执行文件

```
kiosk@k8s build $ ./hello_main
Hello SHared Library! -- 共享库
kiosk@k8s build $
```

三 构建静态库

① 保持环境的干净

```
kiosk@k8s build $ ls
CMakeCache.txt cmake_install.cmake libhello_library.so
CMakeFiles hello_main Makefile
kiosk@k8s build $ rm -fr *
kiosk@k8s build $ ls
kiosk@k8s build $
```

② 修改CMakeLists.txt文件

- 1 | 目的： 生成'静态'库,而不是'动态'库
- 2 |
- 3 | kiosk@k8s build \$ vim ../CMakeLists.txt

修改CMakeLists.txt项目文件， add_library指令中，把SHARED 改为STATIC

https://blog.csdn.net/wzj_110

```
1 # 1) cmake最低版本
2 cmake_minimum_required(VERSION 3.8)
3 # 2) 项目的名称
4 project(HelloLibrary)
5 # 3) 基于HelloLibrary.cpp生成静态库
6 add_library(hello_library STATIC HelloLibrary.cpp)
7 # 4) 当前项目源码增加到系统头文件的路径中,方便后续查找 -->统一用<>方式
8 include_directories(${PROJECT_SOURCE_DIR})
9 # 5) 基于Main.cpp生成最终的二进制可执行文件hello_main
10 add_executable(hello_main Main.cpp)
11 # 6) 指定目标文件(生成的二进制)所依赖的库(前面生成的共享库)
12 target_link_libraries(hello_main hello_library)
```

③ 修改头文件的实现

```
1 备注： 函数的实现'象征性'的修改下
2
3 vim ../HelloLibrary.cpp
```

```
1 #include "HelloLibrary.h"
2
3 void hello_library() {
4     std::cout << "Hello STATIC Library! -- 静态库" << std::endl;
5 }
```

④ 测试

再来跑一次，cmake ..，make，这次生成的库
文件名为libhello_library.a

```
kiosk@k8s build $ cmake3 ..
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is GNU 4.8.5
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc - works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ - works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /var/ftp/pub/pub/cmake/test/CmakeProjects/HelloLibrary/build
kiosk@k8s build $ make
Scanning dependencies of target CMakeFiles/hello_library
[ 25%] Building CXX object CMakeFiles/hello_library.dir/HelloLibrary.cpp.o
[ 50%] Linking CXX static library libhello_library.a
[ 50%] Built target hello_library
Scanning dependencies of target hello_main
[ 75%] Building CXX object CMakeFiles/hello_main.dir/Main.cpp.o
[100%] Linking CXX executable hello_main
[100%] Built target hello_main
kiosk@k8s build $ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  hello_main  libhello_library.a  Makefile
kiosk@k8s build $ ./hello_main
Hello STATIC Library! -- 静态库
```

四 静态库和动态库的区别

静态库在程序编译时会被连接到目标代码中，程序运行时不再需要静态库；而动态库在程序编译时，不会放到连接的目标代码中，而是在程序运行时被载入，因此在程序运行时还需要动态库的存在。

静态库和动态库的区别



显示推荐内容