

## 点击 IOT物联网小镇

作者：道哥，10+年嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。

关注下方公众号，回复【书籍】，获取 Linux、嵌入式领域经典书籍；回复【PDF】，获取所有原创文章(PDF 格式)。

### 目录

#### 示例代码

mylib

myapp

#### Linux 下构建过程

cmake 配置

make 编译

测试、执行

#### Windows 下构建过程

第一步: cmake 配置

第二步: 编译

第三步: 执行

### 别人的经验，我们的阶梯！

大家好，我是道哥，今天我为大伙儿解说的技术知识点是：【使用 cmake 来构建跨平台的动态库和应用程序】。

在很久之前，曾经在B站上传过几个小视频，介绍了在Windows和Linux这两个平台下，如何通过cmake和make这两个构建工具，来编译、链接动态库、静态库以及可执行程序。

视频中的示例代码是提前写好的，因此重点就放在构建(Build)环节了。主要是介绍了动态库与动态库之间、应用程序与动态库之间的引用等等。

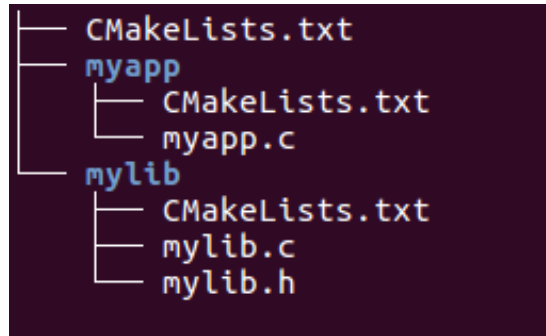
对动态库、静态库比较熟悉的小伙伴，应该很容易就能理解其中的内容。但是对 C 语言不熟悉的朋友，看起来还是有一点点障碍。

这篇文章，主要是把视频中的示例代码进行简化，只使用一个动态库和一个可执行文件，使用cmake构建工具，演示在 Windows 和 Linux 这两个平台下的构建过程。

本文的内容基本基础，算是使用 cmake 来构建跨平台程序的入门教程吧！

## 示例代码

首先看一下测试代码的全貌：



1. mylib: 只有一个源文件，编译输出一个动态库;
2. myapp: 也只有一个源文件，链接 mylib 动态库，编译输出一个可执行程序;

## mylib

在mylib目录中，一共有3个文件：[mylib.h](#), [mylib.c](#) 以及 [CMakeLists.txt](#)，内容分别如下：

```
// mylib/mylib.h w文件

#ifndef _MY_LIB_
#define _MY_LIB_

#ifdef MY_LINUX
#define MYLIB_API extern
#else
#ifdef MYLIB_EXPORT
#define MYLIB_API __declspec(dllexport)
#else
#define MYLIB_API __declspec(dllimport)
#endif
#endif

MYLIB_API int my_add(int num1, int num2);
MYLIB_API int my_sub(int num1, int num2);

#endif // _MY_LIB_
```

以上这个代码，主要是用在Windows系统的[动态导出库](#)，在Linux系统中，是不需要的。

具体来说：在Windows系统中，当编译动态库的时候，[打开](#)(定义)宏 MYLIB\_EXPORT，下面这个宏生效：

```
#define MYLIB_API __declspec(dllexport)
```

这样的话，两个函数 my\_add 和 my\_sub 的符号才可能被[导出到 mylib.lib 文件中](#)。

当这个动态库被应用程序(myapp)使用的时候，myapp.c在 [include](#) mylib.h 的时，[关闭](#)宏 MYLIB\_EXPORT，此时下面这个宏就生效：

```
#define MYLIB_API __declspec(dllimport)
```

为了简化宏定义的复杂度，这里就不考虑静态库了。

看完了头文件，再来看看源文件mylib.c:

```
// mylib/mylib.c 文件

#include "mylib.h"

int my_add(int num1, int num2)
{
    return (num1 + num2);
}

int my_sub(int num1, int num2)
{
    return (num1 - num2);
}
```

最后再来看一下mylib/CMakeLists.txt文件:

```
// mylib/CMakeLists.txt 文件

CMAKE_MINIMUM_REQUIRED(VERSION 3.5)
PROJECT(mylib VERSION 1.0.0)

# 自定义宏，代码中可以使用
ADD_DEFINITIONS(-DMYLIB_EXPORT)

# 头文件
INCLUDE_DIRECTORIES(./)

# 源文件
FILE(GLOB MYLIB_SRCS "*.c")

# 编译目标
ADD_LIBRARY(${PROJECT_NAME} SHARED ${MYLIB_SRCS})
```

关于cmake的语法就不多说了，这里只用到了其中很少的一部分。

注意其中的一点: `ADD_DEFINITIONS(-DMYLIB_EXPORT)`，因为这个CMakeLists.txt是用来编译动态库的，因此在Windows平台下，每一个导出符号的前面需要加上 `__declspec(dllexport)`，因此需要[打开宏定义: MYLIB\\_EXPORT](#)。

## myapp

应用程序的代码就更简单了，只有两个文件: [myapp.c](#) 和 [CMakeLists.txt](#)，内容如下:

```
// myapp/myapp.c 文件

#include <stdio.h>
#include <stdlib.h>

#include "mylib.h"
```

```

int main(int argc, char *argv[])
{
    int ret1, ret2;
    int a = 5;
    int b = 2;

    ret1 = my_add(a, b);
    ret2 = my_sub(a, b);
    printf("ret1 = %d \n", ret1);
    printf("ret2 = %d \n", ret2);
    getchar();
    return 0;
}

```

HelloWorld级别的代码，不需要多解释！CMakeLists.txt内容如下：

```

// myapp/CMakeLists.txt 文件

CMAKE_MINIMUM_REQUIRED(VERSION 3.5)
PROJECT(myapp VERSION 1.0.0)

# 头文件路径
INCLUDE_DIRECTORIES(./include)

# 库文件路径
LINK_DIRECTORIES(./lib)

# 源文件
FILE(GLOB MYAPP_SRCS "*.c")

# 编译目标
ADD_EXECUTABLE(${PROJECT_NAME} ${MYAPP_SRCS})

# 依赖的动态库
TARGET_LINK_LIBRARIES(${PROJECT_NAME} mylib)

```

最后一行 `TARGET_LINK_LIBRARIES(${PROJECT_NAME} mylib)` 说明要链接mylib这个动态库。

那么到[哪个目录](#)下去查找相应的头文件和库文件呢？

通过这两行来指定查找目录：

```

# 头文件路径
INCLUDE_DIRECTORIES(./include)

# 库文件路径
LINK_DIRECTORIES(./lib)

```

这两个目录暂时还不存在，待会编译的时候我们再手动创建。

可以让 mylib 在编译时的输出文件，自动拷贝到指定的目录。但是为了不把问题复杂化，某些操作步骤通过手动操作来完成，这样也能更清楚的理解其中的链接过程。

最后就剩下最外层的CMakeLists.txt文件了：

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.5)
PROJECT(cmake_demo VERSION 1.0.0)

SET(CMAKE_C_STANDARD 99)

# 自定义宏，代码中可以使用
if (CMAKE_HOST_UNIX)
    ADD_DEFINITIONS(-DMY_LINUX)
else ()
    ADD_DEFINITIONS(-DMY_WINDOWS)
endif()

ADD_SUBDIRECTORY(mylib)
ADD_SUBDIRECTORY(myapp)
```

它所做的主要工作就是：根据不同的平台，定义相应的宏，并且添加了mylib和myapp这两个子文件夹。

## Linux 下构建过程

### cmake 配置

为了不污染源文件目录，在最外层目录下新建build目录，然后执行cmake指令：

```
$ cd ~/tmp/cmake_demo/
$ mkdir build
$ cd build/
$ ls
$ cmake ..
```

此时，在build目录下，产生如下文件：

```
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile  myapp  mylib
```

### make 编译

我们可以分别进入mylib和myapp目录，执行make指令来单独编译，也可以直接在build目录下编译所有的目标。

现在就直接在build目录下编译所有目标：

```
$ cd ~/tmp/cmake_demo/build
$ make
Scanning dependencies of target mylib
```

```
[ 25%] Building C object mylib/CMakeFiles/mylib.dir/mylib.c.o
[ 50%] Linking C shared library libmylib.so
[ 50%] Built target mylib
Scanning dependencies of target myapp
[ 75%] Building C object myapp/CMakeFiles/myapp.dir/myapp.c.o
~/tmp/cmake_demo/myapp/myapp.c:4:19: fatal error: mylib.h: 没有那个文件或目录
#include "mylib.h"
      ^
compilation terminated.
myapp/CMakeFiles/myapp.dir/build.make:62: recipe for target
'myapp/CMakeFiles/myapp.dir/myapp.c.o' failed
make[2]: *** [myapp/CMakeFiles/myapp.dir/myapp.c.o] Error 1
CMakeFiles/Makefile2:140: recipe for target 'myapp/CMakeFiles/myapp.dir/all' failed
make[1]: *** [myapp/CMakeFiles/myapp.dir/all] Error 2
Makefile:83: recipe for target 'all' failed
make: *** [all] Error 2
```

从提示信息中看出：已经编译生成了 `./mylib/libmylib.so` 文件，但是在编译可执行程序 `myapp` 时遇到了错误：找不到 `mylib.h` 文件！

在刚才介绍 `myapp/CMakeLists.txt` 文件时说到：应用程序查找头文件的目录是 `myapp/include`，查找库文件的目录是 `myapp/lib`。

但是这2个目录以及相应的头文件、库文件都不存在！

因此我们需要手动创建，并且把头文件 `mylib.h` 和库文件 `libmylib.so` 拷贝进去，操作过程如下：

```
$ cd ~/tmp/cmake_demo/myapp/
$ mkdir include lib
$ cp ~/tmp/cmake_demo/mylib/mylib.h ./include/
$ cp ~/tmp/cmake_demo/build/mylib/libmylib.so ./lib/
```

注意：刚才编译生成的库文件 `libmylib.so` 是在 `build` 目录下。

准备好头文件和库文件之后，再次编译一下：

```
$ cd ~/tmp/cmake_demo/build/
$ make
[ 50%] Built target mylib
[ 75%] Building C object myapp/CMakeFiles/myapp.dir/myapp.c.o
[100%] Linking C executable myapp
[100%] Built target myapp
```

此时，就在 `build/myapp` 目录下生成可执行文件 `myapp` 了。

## 测试、执行

```
$ cd ~/tmp/cmake_demo/build/myapp
$ ./myapp
ret1 = 7
ret2 = 3
```

完美！

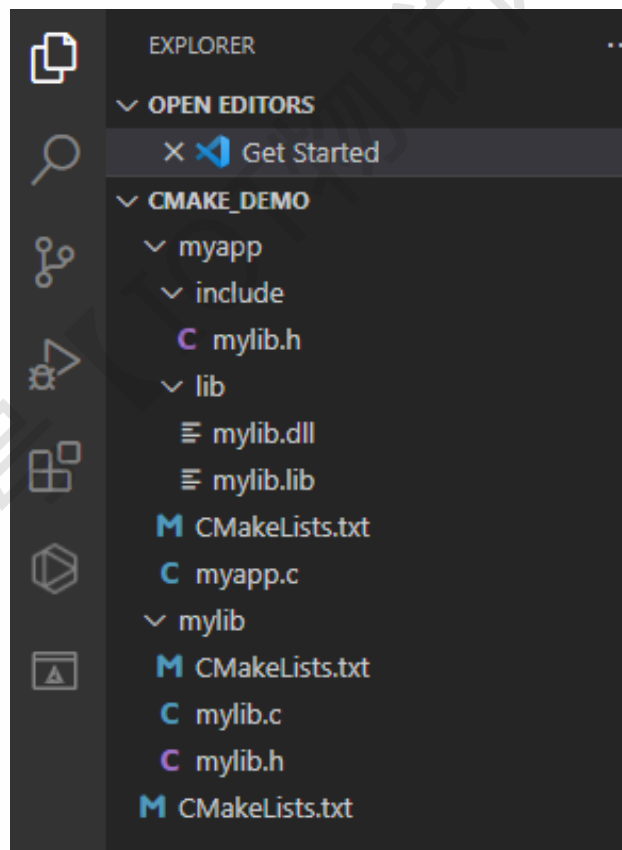
由于我们是在build目录下编译的，编译过程中所有的输出和中间文件，都放在build目录下，一点都没有污染源文件。

## Windows 下构建过程

把Linux系统中的build文件夹删除，然后把测试代码压缩，复制到Windows系统中继续测试。

在Windows下编译，一般就很少使用命令行了，大部分都使用VS或者VSCode来编译。

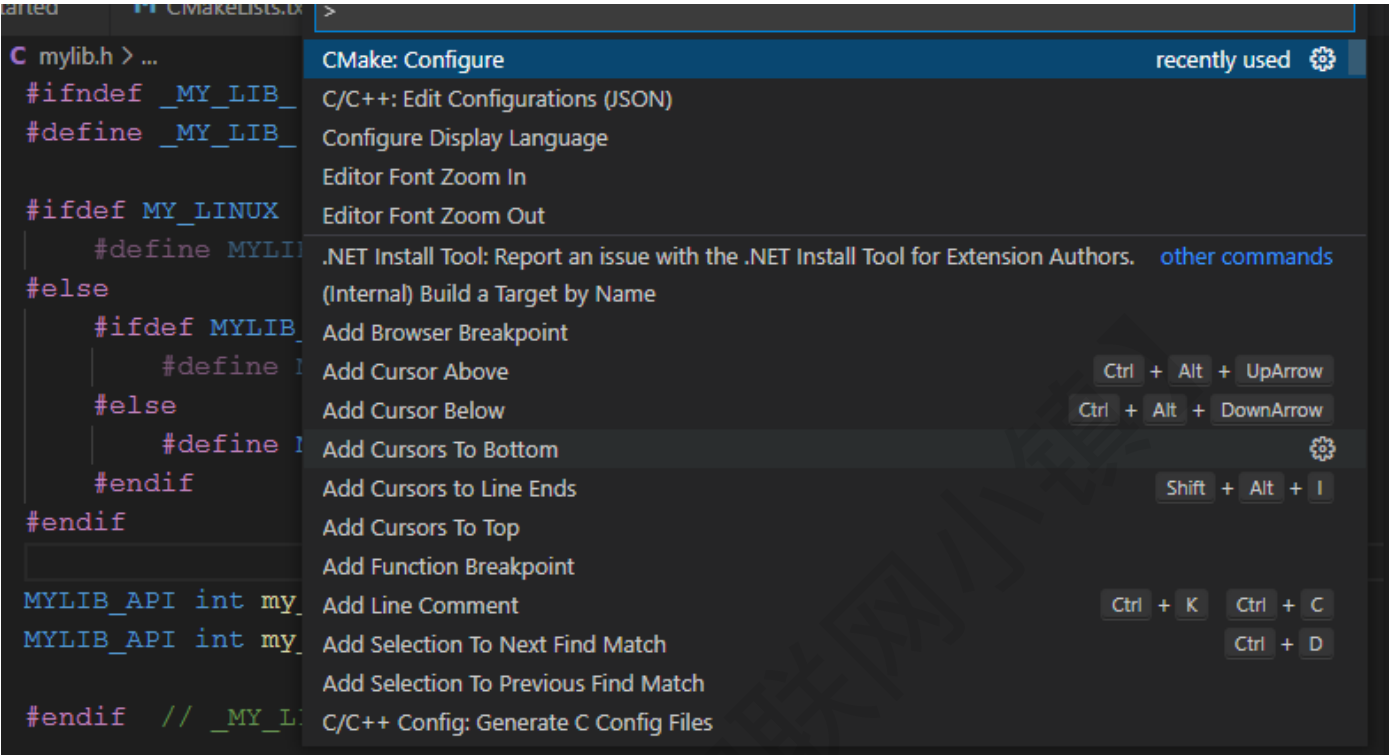
打开VSCode，然后打开测试代码文件夹 `cmake_demo`:



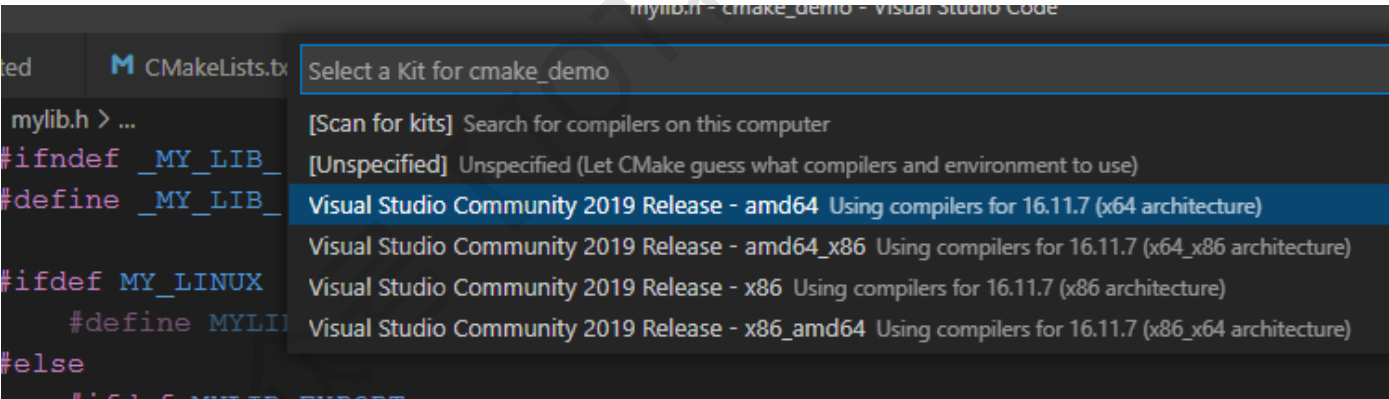
因为需要使用cmake工具来构建，所以需要在VSCode安装 `cmake` 插件。

# 第一步: cmake 配置

按下键盘ctrl + shift + p, 在命令窗口中选择Cmake: Configure, 如果没看到这个选项, 就手动输入前面的几个字符, 然后就可以智能匹配到:



在第一次 Configure 的时候, 会弹出下面的选项, 来选择编译器:



我们这里选择 64 位的 amd64。

配置的结果输出在最下面窗口中的output标签中, 如下所示:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[cmake] -- The CXX compiler identification is MSVC 19.29.30137.0
[cmake] -- Detecting C compiler ABI info
[cmake] -- Detecting C compiler ABI info - done
[cmake] -- Check for working C compiler: d:/Program Files (x86)/Microsoft
[cmake] -- Detecting C compile features
[cmake] -- Detecting C compile features - done
[cmake] -- Detecting CXX compiler ABI info
[cmake] -- Detecting CXX compiler ABI info - done
[cmake] -- Check for working CXX compiler: d:/Program Files (x86)/Microsof
[cmake] -- Detecting CXX compile features
[cmake] -- Detecting CXX compile features - done
[cmake] -- Configuring done
[cmake] -- Generating done
[cmake] -- Build files have been written to: F:/tmp/cmake_demo/build
```

这就表明cmake配置成功，正确的执行了每一个文件夹下的 CMakeLists.txt 文件。

这个时候，来看一下资源管理器中有啥变化：自动生成了 build 目录，其中的文件如下：

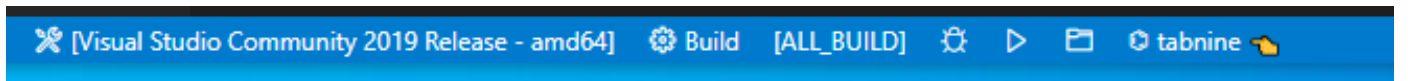
|                            |                  |                     |       |
|----------------------------|------------------|---------------------|-------|
| .cmake                     | 2021/12/12 19:50 | 文件夹                 |       |
| CMakeFiles                 | 2021/12/12 19:50 | 文件夹                 |       |
| myapp                      | 2021/12/12 19:50 | 文件夹                 |       |
| mylib                      | 2021/12/12 19:50 | 文件夹                 |       |
| ALL_BUILD.vcxproj          | 2021/12/12 19:50 | VC++ Project        | 72 KB |
| ALL_BUILD.vcxproj.filters  | 2021/12/12 19:50 | VC++ Project Fil... | 1 KB  |
| cmake_demo.sln             | 2021/12/12 19:50 | Microsoft Visual... | 5 KB  |
| cmake_install.cmake        | 2021/12/12 19:50 | CMake 源文件           | 2 KB  |
| CMakeCache.txt             | 2021/12/12 19:50 | 文本文档                | 15 KB |
| ZERO_CHECK.vcxproj         | 2021/12/12 19:50 | VC++ Project        | 72 KB |
| ZERO_CHECK.vcxproj.filters | 2021/12/12 19:50 | VC++ Project Fil... | 1 KB  |

看来，流程与Linux系统中都是一样的，只不过这里是VSCode主动帮我们做了一些事情。

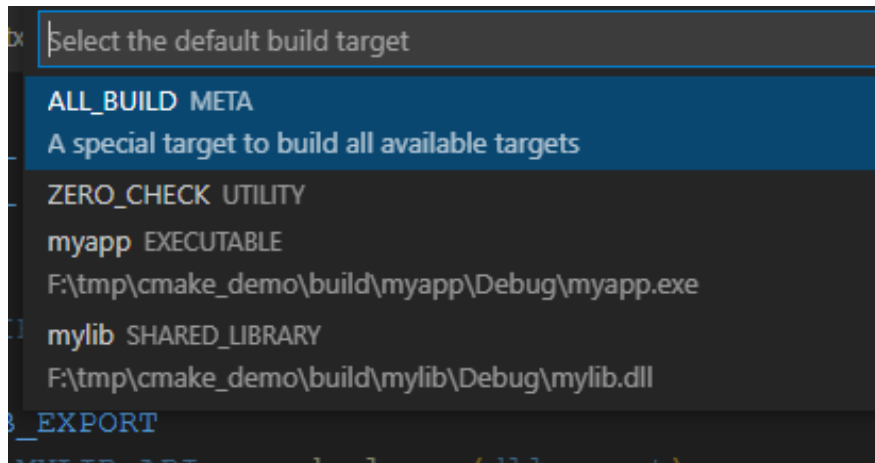
## 第二步: 编译

配置之后，下一步就是编译了。

按下 `shift + F7`，或者单击VSCode底部的 Build 图标：



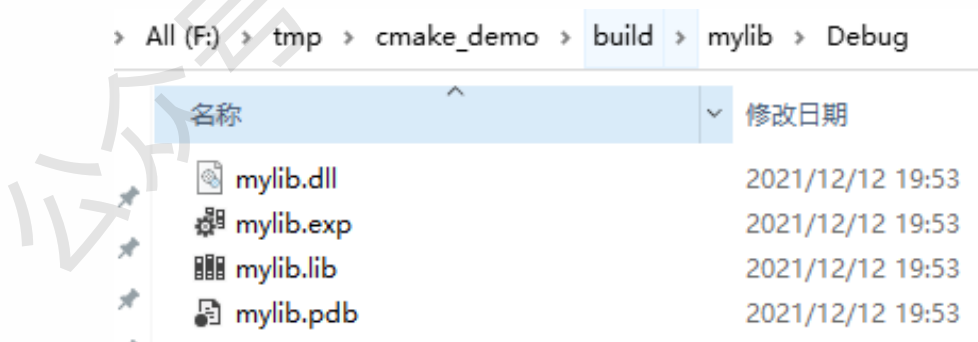
弹出编译目标列表：



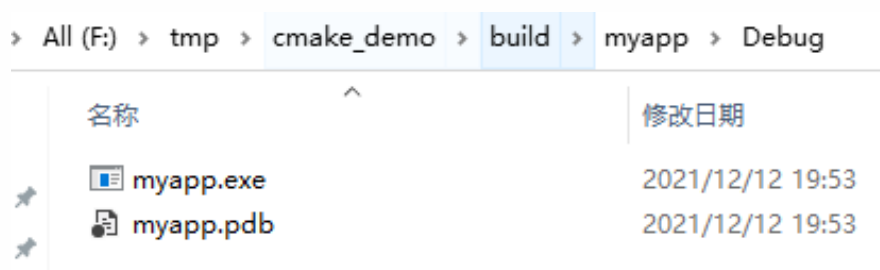
这里选择ALL\_BUILD，也就是编译所有的目标：mylib 和 myapp，输出如下：

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[proc] Executing command: "d:\Program Files (x86)\Microsoft Visual Studio\2019
[build] 用于 .NET Framework 的 Microsoft (R) 生成引擎版本 16.11.2+f32259642
[build] 版权所有 (C) Microsoft Corporation。保留所有权利。
[build]
[build] Checking Build System
[build] Building Custom Rule F:/tmp/cmake_demo/mylib/CMakeLists.txt
[build] mylib.c
[build] 正在创建库 F:/tmp/cmake_demo/build/mylib/Debug/mylib.lib 和对象 F:/t
[build] mylib.vcxproj -> F:\tmp\cmake_demo\build\mylib\Debug\mylib.dll
[build] Building Custom Rule F:/tmp/cmake_demo/myapp/CMakeLists.txt
[build] myapp.c
[build] myapp.vcxproj -> F:\tmp\cmake_demo\build\myapp\Debug\myapp.exe
[build] Building Custom Rule F:/tmp/cmake_demo/CMakeLists.txt
[build] Build finished with exit code 0
```

来看一下编译的输出文件：



mylib.dll 就是编译得到的动态链接库，mylib.lib是导入符号。



myapp.exe 是编译得到的可执行程序。

### 第三步: 执行

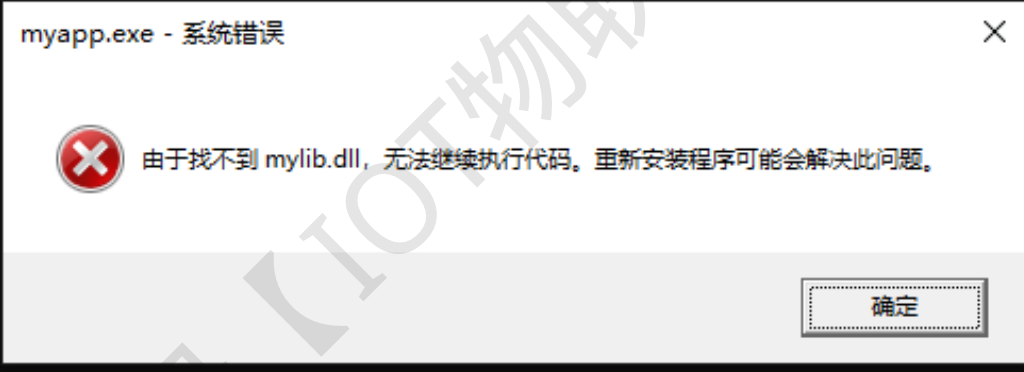
我们先在命令行窗口中执行一下myapp.exe:

```
C:\Windows\System32\cmd.exe - myapp.exe
Microsoft Windows [版本 10.0.19042.1348]
(c) Microsoft Corporation。保留所有权利。

F:\tmp\cmake_demo\build\myapp\Debug>dir
驱动器 F 中的卷是 A11
卷的序列号是 0849-FFB1

F:\tmp\cmake_demo\build\myapp\Debug 的目录
2021/12/12  19:53    <DIR>          .
2021/12/12  19:53    <DIR>          ..
2021/12/12  19:53                   53,248 myapp.exe
2021/12/12  19:53                   798,720 myapp.pdb
                2 个文件             851,968 字节
                2 个目录  55,927,631,872 可用字节

F:\tmp\cmake_demo\build\myapp\Debug>myapp.exe
```



提示错误: 找不到动态链接库!

手动把mylib.dll拷贝到myuapp.exe同一个目录下, 然后再执行一次 myapp.exe:

```
F:\tmp\cmake_demo\build\myapp\Debug>dir
驱动器 F 中的卷是 A11
卷的序列号是 0849-FFB1

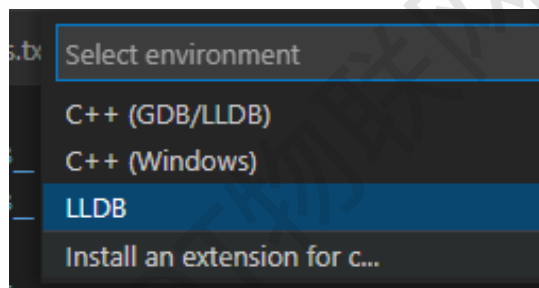
F:\tmp\cmake_demo\build\myapp\Debug 的目录
2021/12/12  19:59    <DIR>          .
2021/12/12  19:59    <DIR>          ..
2021/12/12  19:53             53,248 myapp.exe
2021/12/12  19:53            798,720 myapp.pdb
2021/12/12  19:53            50,176 mylib.dll
                3 个文件             902,144 字节
                2 个目录  55,927,578,624 可用字节

F:\tmp\cmake_demo\build\myapp\Debug>myapp.exe
ret1 = 7
ret2 = 3
```

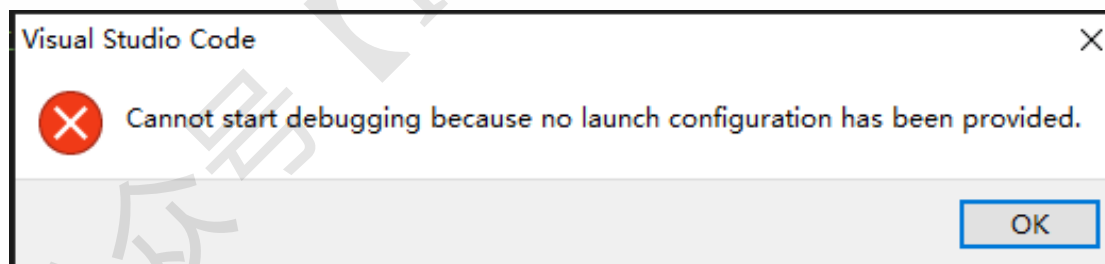
完美！

但是，既然已经用VSCode来编译了，那就继续在VSCode中进行代码调试吧。

按下调试快捷键F5，第一次会弹出调试器选择项：



选择LLDB，然后弹出错误对话框：



因为我们没有提供相应的配置文件来告诉VSCode调试哪一个可执行程序。

单击[OK]之后，VSCode会自动为我们生成 `.vscode/launcher.json` 文件，内容如下：

```

1  {
2      // Use IntelliSense to learn about possible attributes.
3      // Hover to view descriptions of existing attributes.
4      // For more information, visit: https://go.microsoft.com
5      "version": "0.2.0",
6      "configurations": [
7          {
8              "type": "lldb",
9              "request": "launch",
10             "name": "Debug",
11             "program": "${workspaceFolder}/<your program>",
12             "args": [],
13             "cwd": "${workspaceFolder}"
14         }
15     ]
16 }

```

把其中的program项目，改成可执行程序的全路径：

```
"program": "F:/tmp/cmake_demo/build/myapp/Debug/myapp.exe"
```

然后再次按下F5键，这回终于可以正确执行了：

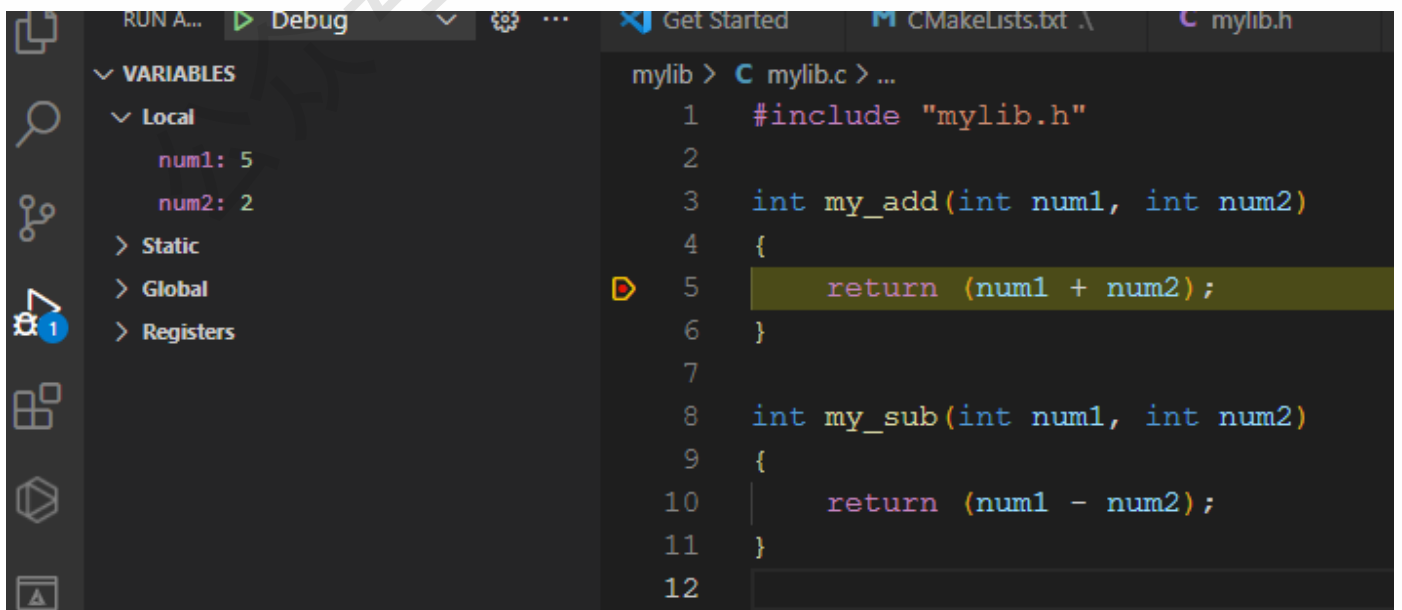
```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

ret1 = 7
ret2 = 3
PS F:\tmp\cmake_demo>

```

此时，就可以在mylib.c或者myapp.c中设置断点，然后进行单步调试程序了：



RUN A... **Debug** Get Started CMakeLists.txt \ C mylib.h

**VARIABLES**

- Local
  - argc: 1
  - > argv: {0x0000025f6cfff25e0}
  - ret1: 7
  - b: 2
  - a: 5
  - ret2: 3
- Static
- Global
- Registers

**WATCH**

```
myapp > C myapp.c > main(int, char * [])
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "mylib.h"
5
6  int main(int argc, char *argv[])
7  {
8      int ret1, ret2;
9      int a = 5;
10     int b = 2;
11
12     ret1 = my_add(a, b);
13     ret2 = my_sub(a, b);
14     printf("ret1 = %d \n", ret1);
15     printf("ret2 = %d \n", ret2);
16     getchar();
17     return 0;
18 }
```

-----End-----

## 推荐阅读

- 【1】 [《Linux 从头学》系列文章](#)
- 【2】 [C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻](#)
- 【3】 [原来gdb的底层调试原理这么简单](#)
- 【4】 [内联汇编很可怕吗？看完这篇文章，终结它！](#)

其他系列专辑：[精选文章](#)、[应用程序设计](#)、[物联网](#)、[C语言](#)。



微信搜一搜

Q IOT物联网小镇

星标公众号，第一时间看文章！

C/C++、物联网、嵌入式、Lua语言  
Linux 操作系统、应用程序开发设计



扫码关注公众号



道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。