# Cmake介绍

## ****1.1语法特性****

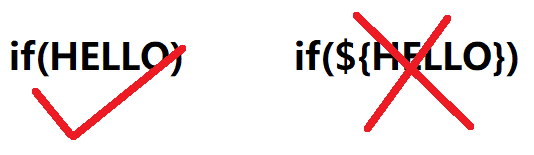
### **基本语法格式：指令(参数 1 参数 2…)**

* 参数使用**括弧**括起
* 参数之间使用**空格**或**分号**分开

### **指令是大小写无关的，参数和变量是大小写相关的**

|  |  |
| --- | --- |
| **1**  **2**  **3** | **set(HELLO hello.cpp) add\_executable(hello main.cpp hello.cpp)**  **ADD\_EXECUTABLE(hello main.cpp ${HELLO})** |

### **变量使用${}方式取值，但是在if控制语句中是直接使用变量名**

****

## ****1.2 重要指令****

* **cmake\_minimum\_required** **- 指定CMake的最小版本要求**

#CMake最小版本要求为2.8.3  
cmake\_minimum\_required(VERSION 2.8.3)

语法：**cmake\_minimum\_required(VERSION versionNumber [FATAL\_ERROR])**

* **project** **- 定义工程名称，并可指定工程支持的语言**

#指定工程名为HELLOWORLD  
project(HELLOWORLD)

语法：**project(projectname [CXX] [C] [Java])**

* **set** **- 显式的定义变量**

#定义SRC变量，其值为main.cpp hello.cpp  
set(SRC sayhello.cpp hello.cpp)

语法：**set(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])**

* **include\_directories - 向工程添加多个特定的头文件搜索路径**  --->相当于指定g++编译器的-I参数

#将/usr/include/myincludefolder 和 ./include 添加到头文件搜索路径  
include\_directories(/usr/include/myincludefolder ./include)

语法：**include\_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 …)**

* **link\_directories** **- 向工程添加多个特定的库文件搜索路径**  --->相当于指定g++编译器的-L参数

#将/usr/lib/mylibfolder 和 ./lib 添加到库文件搜索路径  
link\_directories(/usr/lib/mylibfolder ./lib)

语法：link\_directories(dir1 dir2 …)

* **add\_library** **- 生成库文件**

#通过变量 SRC 生成 libhello.so 共享库  
add\_library(hello SHARED ${SRC})

语法：**add\_library(libname [SHARED|STATIC|MODULE] [EXCLUDE\_FROM\_ALL] source1 source2 … sourceN)**

## ****1.3 常用变量****

**CMAKE\_C\_FLAGS   gcc编译选项**

**CMAKE\_CXX\_FLAGS  g++编译选项**

#在CMAKE\_CXX\_FLAGS编译选项后追加-std=c++11  
set( CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -std=c++11")

**CMAKE\_BUILD\_TYPE  编译类型(Debug, Release)**

#设定编译类型为debug，调试时需要选择debug  
set(CMAKE\_BUILD\_TYPE Debug)   
#设定编译类型为release，发布时需要选择release  
set(CMAKE\_BUILD\_TYPE Release)

**CMAKE\_BINARY\_DIR**

**PROJECT\_BINARY\_DIR**

**<projectname>\_\_BINARY\_DIR**

* 这三个变量指代的内容是一致的。
* 如果是 in source build，指的就是工程顶层目录。
* 如果是 out-of-source 编译,指的是工程编译发生的目录。
* PROJECT\_BINARY\_DIR 跟其他指令稍有区别，不过现在，你可以理解为他们是一致的。
* **CMAKE\_SOURCE\_DIR**

**PROJECT\_SOURCE\_DIR**  
**<projectname>\_\_SOURCE\_DIR**

* 1. 这三个变量指代的内容是一致的,不论采用何种编译方式,都是工程顶层目录。
  2. 也就是在 in source build时,他跟 CMAKE\_BINARY\_DIR 等变量一致。
  3. PROJECT\_SOURCE\_DIR 跟其他指令稍有区别,现在,你可以理解为他们是一致的。
* **CMAKE\_C\_COMPILER：指定C编译器**
* **CMAKE\_CXX\_COMPILER：指定C++编译器**
* **EXECUTABLE\_OUTPUT\_PATH：可执行文件输出的存放路径**
* **LIBRARY\_OUTPUT\_PATH：库文件输出的存放路径**

# C++的Cmake编译

CMake目录结构：项目主目录存在一个CMakeLists.txt文件

**两种方式设置编译规则**：

（1）包含源文件的子文件夹包含CMakeLists.txt文件，主目录的CMakeLists.txt通过add\_subdirectory添加子目录即可；

（2）包含源文件的子文件夹未包含CMakeLists.txt文件，子目录编译规则体现在主目录的CMakeLists.txt中；

下载示例代码：

$git clone https://github.com/fhqddm/linux\_cmake\_tutorial.git

## ****2.1 编译流程****

**在 linux 平台下使用 CMake 构建C/C++工程的流程如下:**

* 手动编写 CmakeLists.txt。
* 执行命令 cmake PATH生成 Makefile ( PATH 是顶层CMakeLists.txt 所在的目录 )。
* 执行命令make 进行编译。

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | . # 表示当前目录  ./ # 表示当前目录  .. # 表示上级目录  ../ # 表示上级目录 |

### 2.1.1 **内部构建**： 不推荐

内部构建会在同级目录下产生一大堆中间文件，这些中间文件并不是我们最终所需要的，和工程源文件放在一起会显得杂乱无章。

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | # 在当前目录下，编译本目录的CMakeLists.txt，生成Makefile和其他文件  cmake .  # 执行make命令，生成target  make |

### 2.1.2 外部构建：

将编译输出文件与源文件放到不同目录中

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | # 1. 在当前目录下，创建build文件夹  mkdir build  # 2. 进入到build文件夹  cd build  # 3. 编译上级目录的CMakeLists.txt，生成Makefile和其他文件  cmake ..  # 4. 执行make命令，生成target  make |

## ****2.2 单个源文件编译（demo1）****

**（1）创建test1.cpp如下**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | #include <iostream>  using namespace std;  int main()  {  cout << "aaabbbccc" << endl;  } |

**（2）创建CmakeLists.txt如下**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | CMAKE\_MINIMUM\_REQUIRED(VERSION 3.10)  PROJECT(demo1)  ADD\_EXECUTABLE(demo1 test1.cpp) |

**（3）创建build文件夹，在里面执行cmake编译生成MakeFile，再进行make编译,生成demo1可执行文件**

## ****2.3多个源文件编译（demo2）****

**（1）创建mymath.hpp如下**

#ifndef MYMATH\_H

#define MYMATH\_H

double power(double,double);

#endif

**（2）创建mymath.cpp如下**

#include "./mymath.hpp"

double power(double base,double exponent)

 {

      int res=base;

      if(base==1)

      {

      return 1;

      }

      for(int i=1;i<exponent;i++)

      {

          res=res\*base;

      }

      return res;

}

}

**（3）创建main.cpp如下**

#include <iostream>

#include "./mymath.hpp"

using namespace std;

int main(int argc,char \*argv[])

{

    if(argc<3)

    {

        cout<<"Usage:"<<argv[0]<<endl;

        return 1;

    }

    double base =atof(argv[1]);

    int exponent=atoi(argv[2]);

    cout<<"result:"<<power(base,exponent)<<endl;

    return 0;

}

**（4）创建CmakeLists.txt如下**

CMAKE\_MINIMUM\_REQUIRED(VERSION 3.10)

PROJECT(demo2)

#把当前目录下的所有源码文件名赋给变量DIR\_HELLO\_SRCS

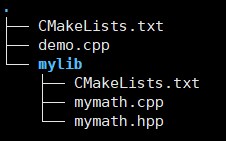
AUX\_SOURCE\_DIRECTORY(./ DIR\_SRCS)

ADD\_EXECUTABLE(demo2  ${DIR\_SRCS})

**（5）创建build文件夹，在里面执行cmake编译生成MakeFile，再进行make编译,生成demo2可执行文件**

## ****2.4 多文件多目录编译（demo3）****

**（1）搭建如下目录结构**



**（2）源文件./mylib/mymath.cpp 与./mylib/mymath.hpp与2.3中的相同，创建./mylib/CMakeLists.txt如下：**

aux\_source\_directory(. DIR\_LIB\_SRCS)

#生成动态库或静态库

add\_library(Mylib STATIC ${DIR\_LIB\_SRCS})

**（3）创建demo.cpp如下：**

#include <iostream>

#include "./mylib/mymath.hpp"

using namespace std;

int main(int argc,char \*argv[])

{

    if(argc<3)

    {

        cout<<"Usage:"<<argv[0]<<endl;

        return 1;

    }

    double base =atof(argv[1]);

    int exponent=atoi(argv[2]);

    cout<<"result:"<<power(base,exponent)<<endl;

    return 0;

}

**（4）创建CMakeLists.txt如下：**

CMAKE\_MINIMUM\_REQUIRED(VERSION 3.10)

PROJECT(demo3)

# ADD\_SUBDIRECTORY(src\_dir [binary\_dir] [EXCLUDE\_FROM\_ALL])

# 向当前工程添加存放源文件的子目录,并可以指定中间二进制和目标二进制的存放位置

# EXCLUDE\_FROM\_ALL含义：将这个目录从编译过程中排除

ADD\_SUBDIRECTORY(./mylib)

AUX\_SOURCE\_DIRECTORY(./ DIR\_SRCS)

ADD\_EXECUTABLE(demo3  ${DIR\_SRCS})

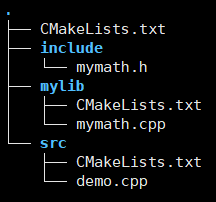
#为target添加需要链接的共享库

TARGET\_LINK\_LIBRARIES(demo3 Mylib)

**（5）创建build文件夹，在里面执行cmake编译生成MakeFile，再进行make编译,生成demo3可执行文件**

## ****2.5 多目录多文件标准工程（demo4）****

**（1）搭建如下目录结构**



**（2）创建./include/mymath.h**

#ifndef MYMATH\_H

#define MYMATH\_H

double power(double,double);

#endif

**（3）创建./mylib/mymath.cpp**

#include "../include/mymath.h"

double power(double base,double exponent)

 {

      int res=base;

      if(base==1)

      {

      return 1;

      }

      for(int i=1;i<exponent;i++)

      {

          res=res\*base;

      }

      return res;

}

**（4）创建./mylib/CMakeLists.txt**

AUX\_SOURCE\_DIRECTORY(. DIR\_LIB\_SRCS)

#使用SET命令重新定义LIBRARY\_OUTPUT\_PATH变量来指定最终的二进制文件的位置

SET(LIBRARY\_OUTPUT\_PATH ${PROJECT\_BINARY\_DIR}/lib)

ADD\_LIBRARY(Mylib STATIC ${DIR\_LIB\_SRCS})

**（5）创建./src/demo.cpp**

#include <iostream>

#include "../include/mymath.h"

using namespace std;

int main(int argc,char \*argv[])

{

    if(argc<3)

    {

        cout<<"Usage:"<<argv[0]<<endl;

        return 1;

    }

    double base =atof(argv[1]);

    int exponent=atoi(argv[2]);

    cout<<"result:"<<power(base,exponent)<<endl;

    return 0;

}

**（6）创建./src/CMakeLists.txt**

# INCLUDE\_DIRECTORIES([AFTER | BEFORE] [SYSTEM] dir1 dir2 … )

# 向工程添加多个特定的头文件搜索路径,路径之间用空格分隔,如果路径包含空格,可以使用双引号将它括起来,默认的行为为追加到当前头文件搜索路径的后面。有如下两种方式可以控制搜索路径添加的位置：

# CMAKE\_INCLUDE\_DIRECTORIES\_BEFORE,通过SET这个cmake变量为on,可以将添加的头文件搜索路径放在已有路径的前面

# 通过AFTER或BEFORE参数,也可以控制是追加还是置前

INCLUDE\_DIRECTORIES(${PROJECT\_SOURCE\_DIR}/mylib)

#使用SET命令重新定义EXECUTABLE\_OUTPUT\_PATH变量来指定最终的二进制文件的位置

SET(EXECUTABLE\_OUTPUT\_PATH ${PROJECT\_BINARY\_DIR}/bin)

#发现一个目录下所有的源代码文件并将列表存储在一个变量中

AUX\_SOURCE\_DIRECTORY(./ DIR\_SRCS)

ADD\_EXECUTABLE(demo4 ${DIR\_SRCS})

TARGET\_LINK\_LIBRARIES(demo4 Mylib)

**（7）创建./CMakeLists.txt**

CMAKE\_MINIMUM\_REQUIRED(VERSION 3.10)

PROJECT(demo3)

INCLUDE\_DIRECTORIES(./include)

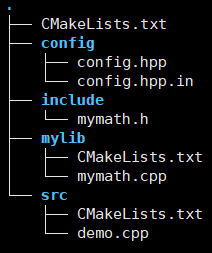
ADD\_SUBDIRECTORY(./mylib)

ADD\_SUBDIRECTORY(./src)

**（8）创建build文件夹，在里面执行cmake编译生成MakeFile，再进行make编译,生成demo4可执行文件**

## ****2.6 自定义编译（demo5）****

**（1）搭建如下目录结构**



**（2）创建./config/config.hpp**

/\* #undef USE\_MYMATH \*/

**（3）创建./config/config.hpp.in**

#cmakedefine USE\_MYMATH

**（4）头文件./include/mymath.h、源文件./mylib/mymath.cpp与2.5相同**

**（5）创建./mylib/CMakeLists.txt**

AUX\_SOURCE\_DIRECTORY(. DIR\_LIB\_SRCS)

SET(LIBRARY\_OUTPUT\_PATH ${PROJECT\_BINARY\_DIR}/lib)

ADD\_LIBRARY(Mylib STATIC ${DIR\_LIB\_SRCS})

**（6）创建./src/CMakeLists.txt**

SET(EXECUTABLE\_OUTPUT\_PATH ${PROJECT\_BINARY\_DIR}/bin)

CONFIGURE\_FILE(

        "${PROJECT\_SOURCE\_DIR}/config/config.hpp.in"

        "${PROJECT\_SOURCE\_DIR}/config/config.hpp"

        )

OPTION(USE\_MYMATH

        OFF

        )

IF(USE\_MYMATH)

INCLUDE\_DIRECTORIES(${PROJECT\_SOURCE\_DIR}/mylib)

ENDIF(USE\_MYMATH)

AUX\_SOURCE\_DIRECTORY(./ DIR\_SRCS)

ADD\_EXECUTABLE(demo5 ${DIR\_SRCS})

TARGET\_LINK\_LIBRARIES(demo5 Mylib)

**（7）创建./CMakeLists.txt**

AUX\_SOURCE\_DIRECTORY(. DIR\_LIB\_SRCS)

SET(LIBRARY\_OUTPUT\_PATH ${PROJECT\_BINARY\_DIR}/lib)

ADD\_LIBRARY(Mylib STATIC ${DIR\_LIB\_SRCS})

**（8）创建build文件夹，在里面执行cmake编译生成MakeFile，再进行make编译,生成demo5可执行文件**

## ****2.7 引用opencv的程序编译（demo6）****

**（1）创建test.cpp如下**

#include <opencv2/opencv.hpp>

#include <iostream>

using namespace std;

using namespace cv;

int main()

{

    Mat \*image = new Mat();

    \*image = imread("../test.png");

    cout << image->cols << endl;

    cout << image->rows << endl;

    imwrite("../write.png",\*image);

    delete image;

}

**（2）创建CMakeLists.txt**

#指定cmake版本

cmake\_minimum\_required(VERSION 3.0)

#填写你的项目名字

project(testcv)

#定位opencv包，不需要改

find\_package( OpenCV REQUIRED )

 #设置opencv引用路径，不需要改

include\_directories( ${OpenCV\_INCLUDE\_DIRS} )

 #生成可执行文件，改称你自己的cpp源文件名和要生成的可执行文件名

add\_executable(testcv test.cpp)

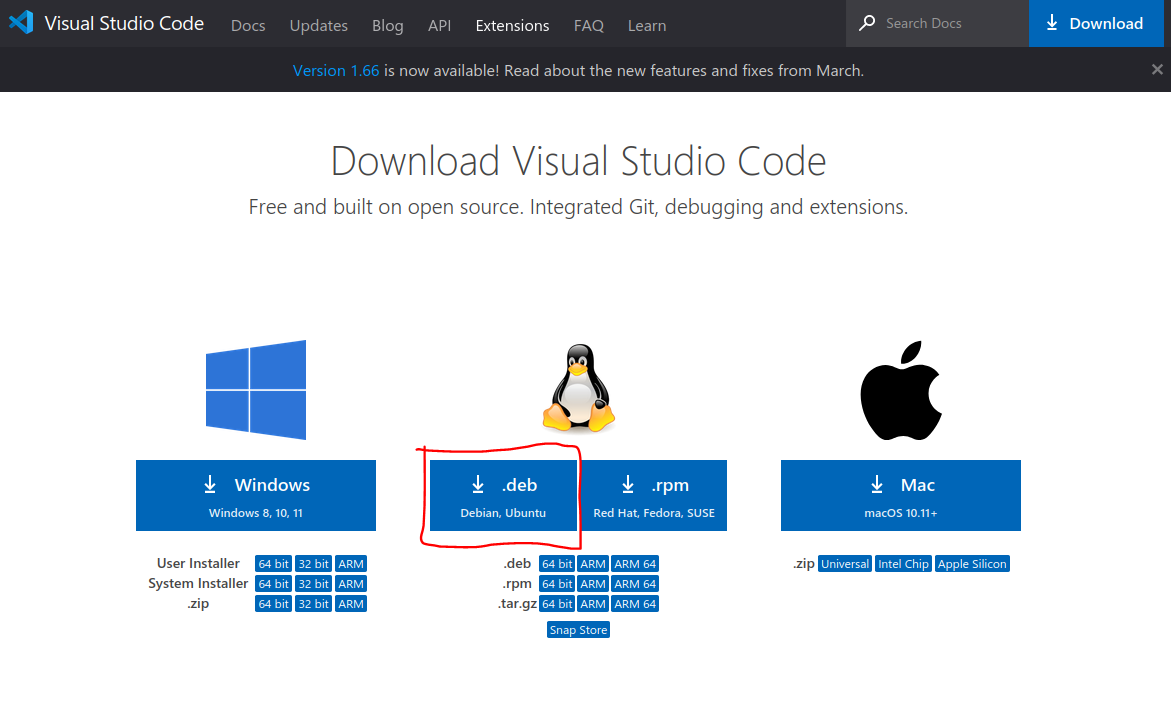
#设置链接库路径，前面改称你自己的项目名

target\_link\_libraries( testcv ${OpenCV\_LIBS} )

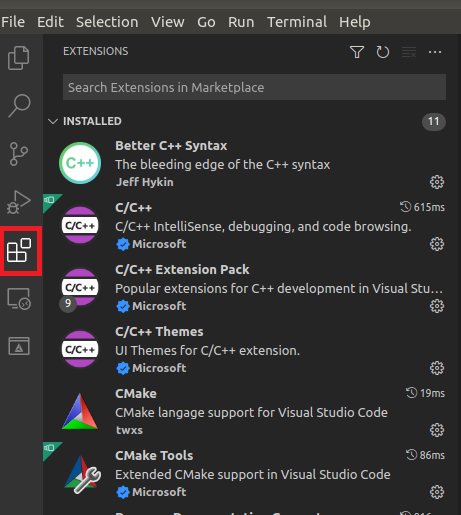
**（3）创建build文件夹，在里面执行cmake编译生成MakeFile，再进行make编译,生成**testcv**可执行文件**

# 使用VSCode在Linux中调试c++程序

## 3.1 下载安装Visual Studio Code



## 3.2 安装c++开发插件



## 3.3 使用VSCode调试程序****（demo7）****

**（1）创建与与2.7中的相同的源文件test.cpp**

**（2）创建CMakeLists.txt**

#指定cmake版本

cmake\_minimum\_required(VERSION 3.0)

#填写你的项目名字

project(testcv)

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall")

set(CMAKE\_BUILD\_TYPE Debug)

#定位opencv包，不需要改

find\_package( OpenCV REQUIRED )

 #设置opencv引用路径，不需要改

include\_directories( ${OpenCV\_INCLUDE\_DIRS} )

 #生成可执行文件，改称你自己的cpp源文件名和要生成的可执行文件名

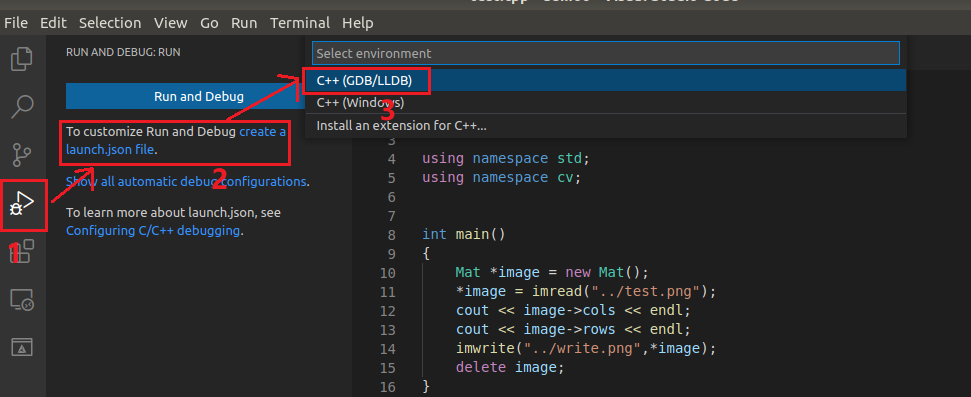
add\_executable(testcv test.cpp)

#设置链接库路径，前面改称你自己的项目名

target\_link\_libraries( testcv ${OpenCV\_LIBS} )

**（3）打开终端，创建build文件夹，在里面执行cmake编译生成MakeFile，再进行make编译,生成testcv可执行文件。**

**（4）在VSCode选择Deubg图标，再选择create a launch.json file，在环境中选择C++(GDB/LLDB)，自动生成launch.json文件。**

****

**（5）在根目录.vscode文件夹中找到launch.json，内容修改成如下。**

{

    // 使用 IntelliSense 了解相关属性。

    // 悬停以查看现有属性的描述。

    // 欲了解更多信息，请访问: https://go.microsoft.com/fwlink/?linkid=830387

    "version": "0.2.0",

    "configurations": [

     {

      "name": "(gdb) 启动",// 配置名称

      "type": "cppdbg",// 配置类型

      "request": "launch",// 请求配置类型,launch或者attach

      "program": "${workspaceFolder}/build/testcv",// 进行调试程序的路径，程序生成文件.out

      "args": [],// 传递给程序的命令行参数，一般为空

      "stopAtEntry": false,// 调试器是否在目标的入口点停止，

      "cwd": "${workspaceFolder}",// 项目目录

      "environment": [],

      "externalConsole": true,// 调试时是否显示控制台窗口，一般为true显示控制台

      "MIMode": "gdb",// 指定连接的调试器

      "setupCommands": [

       {

        "description": "为 gdb 启用整齐打印",

        "text": "-enable-pretty-printing",

        "ignoreFailures": true

       }

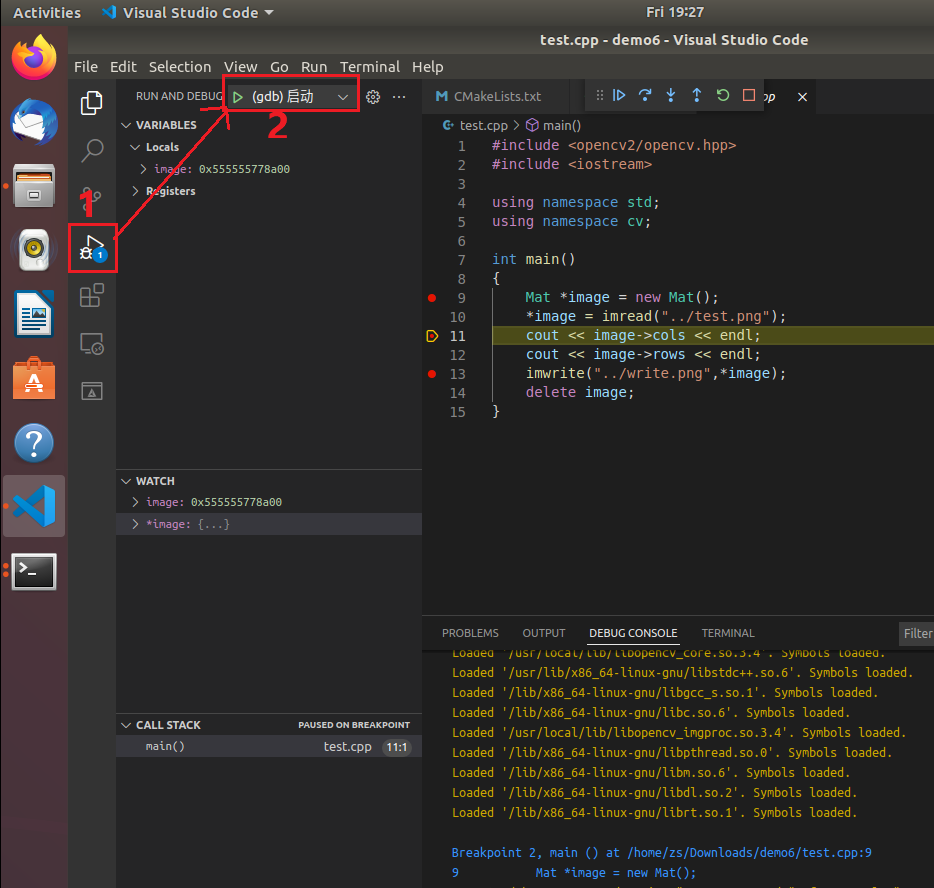
      ]

     }

    ]

   }

**（6）执行调试**

****

# 资源下载目录

<https://github.com/fhqddm/linux_cmake_tutorial>或

$git clone https://github.com/fhqddm/linux\_cmake\_tutorial.git