

## 第一节课习题

### 2. 熟悉 Linux

#### 2.1 请描述 apt-get 安装软件的整体步骤，说明 Ubuntu 是如何管理软件依赖关系和软件版本的

1) APT (Advanced Package Tool, 高级软件包工具) 是一个强大的包管理系统。

安装软件包: `sudo apt-get install packagename`

删除软件包:

`sudo apt-get remove packagename`

2) APT 自动处理依赖关系并在系统软件包执行其他操作以便安装所要的软件包[1]

获取新的软件包列表:

`sudo apt-get update`

升级有可用更新的系统:

`sudo apt-get upgrade`

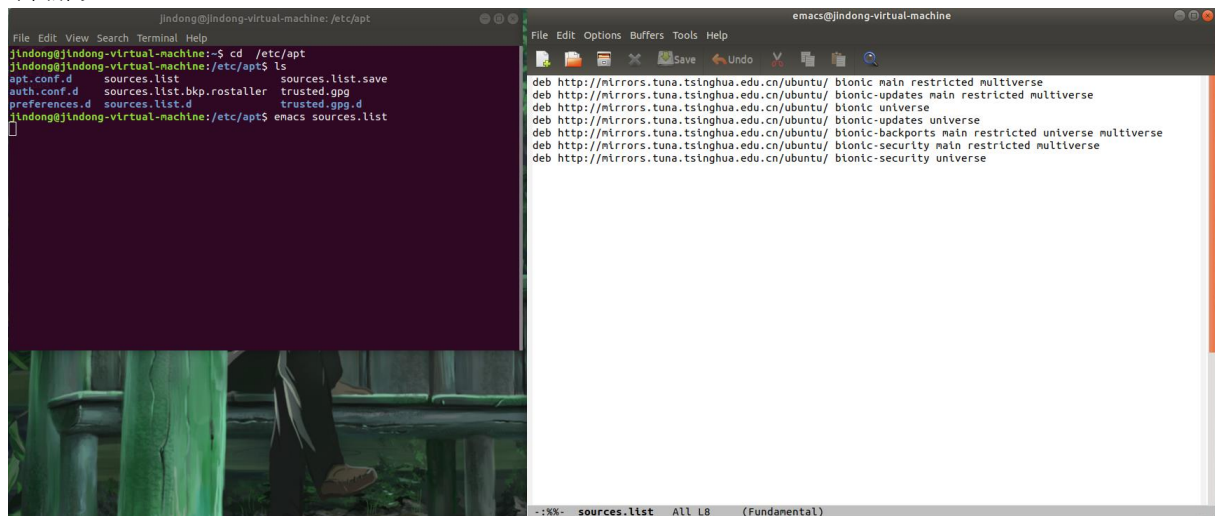
列出更多命令和选项:

`apt-get help`

#### 2.2 什么是软件源? 如何更换系统自带的软件源? 如何安装来自第三方软件源中的软件?

1) 软件源指的是通过一定的方式组织, 集中放置软件包的地方。很多的应用软件都可以在软件源当中找到。软件源可以是网络服务器, 是光盘, 甚至是硬盘上的一个目录。作为 debian 系的 ubuntu, 继承了 debian 的 deb 和 apt 系统, 只要设定好软件源, 就能很方便的利用指令 `sudo apt-get install packagename` 来安装软件了[2]。

2) 为了解决系统自带软件源访问速度慢的问题, 我们会选择更换系统自带的官方软件源。软件源 `sources.list` 文件存放在 `/etc/apt/` 目录下面, 文件的里面为软件源的路径, 我们可以使用任何一种文本编辑工具打开它。而这文件的格式为 `deb [web/ftp 地址] [发行版名字] [main/contrib/non-free]`。我们国内有许多学校和公司的镜像网站都对 Ubuntu 的官方软件源进行了镜像备份, 并且定期地进行同步更新, 而国内访问这些镜像网站的速度都要比直接访问 Ubuntu 官方镜像源要快地多, 所以我们可以将 Ubuntu 官方软件源的网址替换成国内的 Ubuntu 软件源镜像网站的网址[3]。例如我自己电脑上使用的就是清华大学开源镜像网站上的软件源, 如下图所示:



3) 如果我们要的软件在软件源中没有, 那么可以通过添加一些软件较为齐全的源或第三方软件源来解决, 也就是放入更多的软件到软件库中。只要把我们在网上找到的源地址加在 `sources.list` 最后一行就行了, 然后保存。回到终端下, 更新一下软件列表, 输入: `sudo apt-get update`。[2]

#### 2.3 除了 apt-get 以外, 还有什么方式在系统中安装所需软件? 除了 Ubuntu 以外, 其他发行版使用什么软件管理工具? 请至少各列举两种。

1) 除了 apt-get 以外, 也可以写选择使用[1]

-----添加/删除应用程序

-----Ubuntu Software 等简单的图形化安装程序

2) 有许多不同类型的 Linux 软件包文件。它们多数与特定 Linux 发行版的软件管理器相关联。如 Debian 软件包文件 (.deb 文件)、Redhat 软件包管理器文件 (.rpm 文件) 和 Tarballs (.tar 文件)。

例如我们可以使用 `sudo dpkg -i package_file.deb` 来安装 .deb 文件。例如我们可以使用程序 `alien` 将 .rpm 文件转化为 .deb 文件(在终端使用管理权限运行以下命令 `sudo alien package_file.rpm`)再安装即可。[1]

## 2.4 环境变量 PATH 是什么?有什么用途? LD\_LIBRARY\_PATH 是什么?指令 ldconfig 有什么用途?

1) 环境变量 (environment variables) 一般是指在操作系统中用来指定操作系统运行环境的一些参数。环境变量 PATH 规定系统中的可执行文件的位置。只要是处于这些位置中的可执行文件, 执行的时候就不需要指定路径, 直接执行即可。[4]

我们既可以临时设定 PATH, 例如 `PATH=$PATH:/home/catkin_ws`

也可以永久设定 PATH, 将语句 `PATH=$PATH:/home/catkin_ws` 添加到文件 `~/.bashrc` 的末尾。

2) LD\_LIBRARY\_PATH 是 Linux 环境变量名, 该环境变量主要用于指定查找共享库 (动态链接库) 时除了默认路径之外的其他路径。当执行函数动态链接 .so 时, 如果此文件不在缺省目录下 '/lib' and '/usr/lib'. 那么就需要指定环境变量 LD\_LIBRARY\_PATH。假如需要在已有的环境变量上添加新的路径名, 则采用如下方式: `LD_LIBRARY_PATH=NEWDIRS: $LD_LIBRARY_PATH`。(newdirs 是新的路径串)

在 linux 下可以用 export 命令来设置这个值, 比如在 linux 终端下输入: `export LD_LIBRARY_PATH=/opt/au1200_rm/build_tools/bin:$LD_LIBRARY_PATH`;然后再输入: `export` 即会显示是否设置正确

export 方式在重启后失效, 为了永久设定 LD\_LIBRARY\_PATH 也可以打开 /etc/environment 编辑 LD\_LIBRARY\_PATH 变量, 同时编辑 /etc/X11/Xsession.options 文件, 将 use-ssh-agent 更改为 no-use-ssh-agent, 这样设置的环境变量 LD\_LIBRARY\_PATH 可以通用。[5]

3) ldconfig 是动态链接库管理命令。为了让动态链接库为系统所共享, 还需运行动态链接库的管理命令 --ldconfig。此执行程序存放在 /sbin 目录下。ldconfig 通常在系统启动时运行, 而当用户安装了一个新的动态链接库时, 就需要手工运行这个命令。

更详细地说: ldconfig 命令的用途, 主要是在默认搜寻目录 (/lib 和 /usr/lib) 以及动态库配置文件 /etc/ld.so.conf 内所列的目录下, 搜索出可共享的动态链接库 (格式如前介绍, lib\*.so\*), 进而创建出动态装入程序 (ld.so) 所需的连接和缓存文件。缓存文件默认为 /etc/ld.so.cache, 此文件保存已排好序的动态链接库名字列表。[6]

简单地说: 往 /lib 和 /usr/lib 里面加东西后或者修改 /etc/ld.so.conf 后, 一定要调一下 ldconfig, 不然这个 library 会找不到。

总结来说: 设置共享库的路径主要以下几种方法[5]

- 将共享库放在目录 /lib 或者 /usr/lib 下, 并使用 ldconfig 使得添加即时生效
- 将动态库目录添加到文件 /etc/ld.so.conf, 并使用 ldconfig 使得添加即时生效
- /etc/environment 下编辑环境变量 LD\_LIBRARY\_PATH, 同时编辑 /etc/X11/Xsession.options 文件, 将 use-ssh-agent 更改为 no-use-ssh-agent 这样设置的环境变量 LD\_LIBRARY\_PATH 可以通用

## 2.5 Linux 文件权限有哪几种? 如何修改一个文件的权限?

1) Linux 文件权限主要有以下几种[7]

读取 (r) : 允许查看文件内容, 显示目录列表

写入 (w) : 允许修改文件内容, 允许在目录中新建、删除、移动文件或者子目录

可执行 (x) : 允许运行程序, 切换目录

无权限 (-) : 没有权限

2) 我们可以使用 chmod 指令来修改文件的权限, 通用格式为 `chmod [ugoa] [+|=] [rwx] 档案或目录` (见下图[8]), 其中 u 表示该档案的拥有者 (属主 owner), g 表示与该档案的拥有者属于同一个群体者 (属组 group), o 表示其他以外的人 (other), a 表示所有用户 (包括以上三种 all)。+ 表示增加权限、- 表示取消权限、= 表示唯一设定权限。r 表示可读取, w 表示可写入, x 表示可执行。

chmod	u	+(加入)	r	档案或目录
	g	-(除去)	w	
	o	=(设定)	x	
	a			

例如我查看我自己创建的 test.sh 文件的权限 (利用指令 `ls -l test.sh`), 发现它无法被所有用户执行, 于是使用 `chmod a+x test.sh` 设为所有人皆可执行, 见下图:

```
jindong@jindong-virtual-machine: ~/SLAM/Chap1
File Edit View Search Terminal Help
jindong@jindong-virtual-machine:~/SLAM/Chap1$ vim test.sh
jindong@jindong-virtual-machine:~/SLAM/Chap1$ ls -l test.sh
-rw-rw-r-- 1 jindong jindong 0 Sep 20 17:02 test.sh
jindong@jindong-virtual-machine:~/SLAM/Chap1$ chmod a+x test.sh
jindong@jindong-virtual-machine:~/SLAM/Chap1$ ls -l test.sh
-rwxrwxr-x 1 jindong jindong 0 Sep 20 17:02 test.sh
jindong@jindong-virtual-machine:~/SLAM/Chap1$
```

## 2.6 Linux 用户和用户组是什么概念? 用户组的权限是什么意思? 有哪些常见的用户组?

- 1) 用户是能够获取系统资源的权限的集合。在 linux 中的每个用户必须属于一个组, 不能独立于组外。  
用户组是具有相同特征用户的逻辑集合。有时我们需要让多个用户有相同的权限, 比如查看修改某个文件的权限, 一种方法就是对多个用户进行访问授权, 如果有 10 个用户的话, 就要授权 10 次。显然这种方法不太合理。另一种方法就是建立一个组, 让这 10 个用户放在同一个组中, 同时授权这个组有查看, 修改这个文件的权限。这就是用户组, Linux 将用户分组是 Linux 对用户进行管理以及访问控制权限的一种手段, 通过定义用户组, 很大程度上简化了管理工作。
- 2) 用户组的权限就是指该组中所有用户都拥有的权限, 比如对某个文件的读取权限等。
- 3) 常见的用户组有管理员组(超级用户)和普通组(只拥有部分权限)

## 2.7 常见的 Linux 下 C++编译器有哪几种? 在你的机器上, 默认用的是哪一种?它能够支持 C++的哪个标准?

- 1) 常见的 Linux 下 C++编译器包括 GCC、Clang、Visual Studio 以及 Eclipse。[9]
- 2) Ubuntu 18.04 默认使用 GCC 编译器, 通过指令 `gcc -v` 可知我电脑上当前 GCC 版本为 `gcc version 7.5.0`。(见下图)

```
jindong@jindong-virtual-machine: ~  
File Edit View Search Terminal Help  
jindong@jindong-virtual-machine:~$ gcc -v  
Using built-in specs.  
COLLECT_GCC=gcc  
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper  
OFFLOAD_TARGET_NAMES=avx512-none  
OFFLOAD_TARGET_DEFAULT=1  
Target: x86_64-linux-gnu  
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.5.0-3ubuntu1-18.04' --with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdc++-debug --enable-libstdc++-time=yes --with-default-libstdc++-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multilib --disable-werror --with-arch=32=i686 --with-tune=generic --enable-offload-targets=avx512-none --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu  
Thread model: posix  
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1-18.04)
```

- 3) 使用指令 `man gcc` 可在 `-std` 看到它支持 C++11 和 C++14 标准。(见下图)

```
jindong@jindong-virtual-machine: ~  
File Edit View Search Terminal Help  
The 1998 ISO C++ standard plus the 2003 technical corrigendum and some additional defect reports. Same as -ansi for C++ code.  
  
gnu++98  
gnu++03  
GNU dialect of -std=c++98.  
  
c++11  
c++0x  
The 2011 ISO C++ standard plus amendments. The name c++0x is deprecated.  
  
gnu++11  
gnu++0x  
GNU dialect of -std=c++11. The name gnu++0x is deprecated.  
  
c++14  
c++1y  
The 2014 ISO C++ standard plus amendments. The name c++1y is deprecated.  
  
gnu++14  
gnu++1y
```

## 3. SLAM 综述文献阅读

### 3.1 SLAM 会在哪些场合中用到? 至少列举三个方向。

SLAM 可被广泛应用于增强现实, 机器人对未知环境三维结构的实时重建和对自身的实时定位[12], 道路车辆自动驾驶导航, 高风险环境的营救任务, 航空航天陆路海洋等环境下的探险任务, 视觉监控系统, 医药领域等[10]。

### 3.2 SLAM 中定位与建图是什么关系? 为什么在定位的同时需要建图?

最开始的时候, 定位和建图是被分开研究的, 后来发现定位与建图是互相依赖, 相辅相成的。为了使得机器人能够精确地被定位, 一张精确地地图是必须的。反过来, 为了构建一张精确地地图, 精确的定位也是必须的。[10]在环境未知的情况下, 机器人需要一边定位来确定周围的额环境, 一边需要根据周围的环境来估计自身的位置, 因此这个问题被称为同时定位与建图。

### 3.3 SLAM 发展历史如何? 我们可以将它划分为几个阶段?

我们可以将 SLAM 发展划分为三个阶段

#### 第一阶段: 初步发展阶段

1985–1990: Chatila and Laumond 和 Smith et al. 分别在 1985 年和 1990 年提出了同时定位和建图技术。随后该问题被命名为 SLAM (simultaneous localization and mapping) 问题。

2006: Durrant 和 Bailey 等撰写了关于 SLAM 的教程将 SLAM 问题分为了两个部分。后又实现使用 laser range-finder 传感器, 利用概率方法(probabilistic approach)建立了 2D 地图。

2008: Thrun 和 Leonard 介绍了 SLAM 问题, 可以基于三种方式解决, 分别是基于扩展卡尔曼滤波器, 基于图优化和基于粒子滤波器的方法, 对解决 SLAM 问题的方法有了一个比较明确的划分。但是以上方法并未关注只使用视觉传感器的 SLAM 方案(vision-only SLAM)。

2009: Kragic 和 Vincze 发表了关于计算机视觉应用于机器人领域的论文, 着重考虑了视觉 SLAM 问题。

## 第二阶段: 应用多传感器融合阶段

2002-2009: 期间出现了基于多种不同传感器的 SLAM 方案, 例如基于声纳 (Tardós et al. 2002; Ribas et al. 2008), 基于激光测距 (Nüchter et al. 2007; Thrun et al. 2006), 基于摄像头 (Se et al. 2005; Lemaire et al. 2007; Davison 2003; Bogdan et al. 2009), 基于 GPS (Thrun et al. 2005a)。基于这些传感器的 SLAM 方案都有各自的不同的噪声问题和有限的测量范围, 受限于测量环境。后来尝试将增量式传感器 (encoders, accelerometers 和 gyroscopes) 结合到 SLAM 方案中, 提出了多传感器融合 (fusion of information) 的 SLAM 方案 (Castellanos et al. 2001; Majumder et al. 2005; Nützi et al. 2010), 来提高定位视图的准确性和鲁棒性。

## 第三阶段: 主要基于视觉(vision-only)的阶段

由于多传感器的 SLAM 方案增加了成本, 传感器的重量, 以及对载体的能量需求, 所以提出一个只基于视觉摄像头的方案是很重要的, 另一方面视觉 SLAM 可以提供环境信息丰富程度是其他传感器方案所不能比的, 视觉信息可以帮助我们很好的甄别环境里面的人和物[10]。很遗憾的是视觉 SLAM 方案可能会出现很多问题, 比如由于摄像头像素问题, 光照强度和物体的快速移动等问题。

在过去 10 年, 发表了很多基于视觉(vision-only)的 SLAM 方案:

2002-2003: Seet al. 2002 和 Olson et al. 2003 发表基于双目摄像头的视觉 SLAM 方案, 这是视觉 SLAM 领域的第一个方案。

但是由于双目或三目摄像头的成本问题, 我们通常偏向于使用成对的单目摄像头, 当然我们不得不考虑软硬件的同步, 摄像头内置和外置参数调整等多个问题。期间针对摄像头参数校准问题提出了很多不同的参数校准方法。[10]

2007: Klein 等提出基于关键帧 BA 的单目 V-SLAM, 称为 PTAM (Parallel Tracking and Mapping) 创新性的实现了跟踪与建图过程的并行化。目前市面上很多 V-SLAM 系统都是基于 PTAM 的算法框架改进而来。[12]

2011: Newcombe 等提出的 DTAM, 是基于直接跟踪的 V-SLAM。直接跟踪法(Direct Tracking)不依赖于特征点的提取和匹配, 而是直接通过比较像素颜色来求解相机运动, 因此通常在特征缺失、图像模糊等情况下有更好的鲁棒性。[12]

2015: Mur-Artal 等提出并开源的 ORB-SLAM 是目前性能最好的单目 V-SLAM 系统之一。相较于 PTAM 的双线程, 它采用了并行跟踪, 局部建图, 闭环检测三线程[11]。ORB-SLAM 选用了 ORB 特征, 基于 ORB 描述量的特征匹配和重定位, 加入了循环回路的检测和闭合机制, 以消除误差累积, 通过方位图(Pose Graph)优化来闭合回路。[12]

## 3.4 从什么时候开始 SLAM 区分为前端和后端? 为什么我们要把 SLAM 区分为前端和后端?

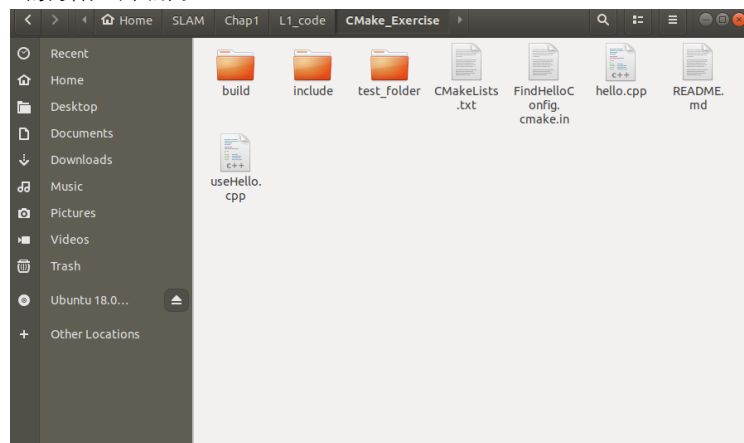
在 Klein 提出的 PTAM 中首次有了前端与后端的概念: 前端为跟踪线程, 根据获取的图像数据实时估计位姿; 后端引入了关键帧机制, 实现对地图的非线性优化。[11]

## 3.5 列举三篇在 SLAM 领域的经典文献。

最经典的综述性文献是 DurrantWhyte 等[3-4]于 2006 年撰写的关于 SLAM 的教程。然后就视觉 SLAM 领域而言, 比较经典的是 2007 年 Klein 等提出基于关键帧 BA 的单目 V-SLAM 方案的论文, 以及 2015 年 Mur-Artal 等提出并开源的 ORB-SLAM 方案的论文。[12]

## 4. CMake 练习

#文件夹 CMake\_Exercise 的内容如下图所示



可按照以下步骤输入命令行对文件夹 CMake\_Exercise 中程序进行验证

#步骤 1. 首先编译文件夹 CMake\_Exercise 中的 CMakeLists.txt

```
cd ~/../CMake_Exercise
```

```
mkdir build
```

```
cd build
```

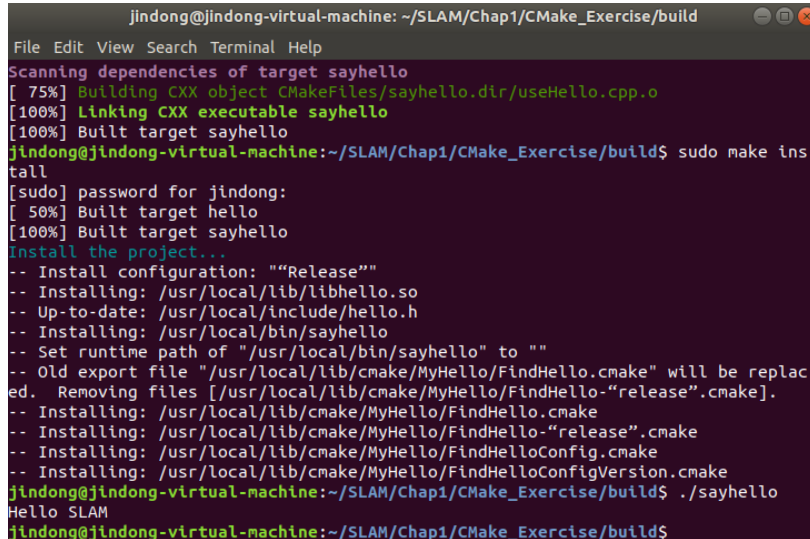
```
cmake ..
```

```
make
```

# 文件夹 CMake\_Exercise/build 中生成动态库 libhello.so 和 可执行文件 sayhello

```
./useHello
```

# 执行可执行文件 sayhello => 成功 (见下图)



```
jindong@jindong-virtual-machine: ~/SLAM/Chap1/CMake_Exercise/build
File Edit View Search Terminal Help
Scanning dependencies of target sayhello
[ 75%] Building CXX object CMakeFiles/sayhello.dir/useHello.cpp.o
[100%] Linking CXX executable sayhello
[100%] Built target sayhello
jindong@jindong-virtual-machine:~/SLAM/Chap1/CMake_Exercise/build$ sudo make install
[sudo] password for jindong:
[ 50%] Built target hello
[100%] Built target sayhello
Install the project...
-- Install configuration: "Release"
-- Installing: /usr/local/lib/libhello.so
-- Up-to-date: /usr/local/include/hello.h
-- Installing: /usr/local/bin/sayhello
-- Set runtime path of "/usr/local/bin/sayhello" to ""
-- Old export file "/usr/local/lib/cmake/MyHello/FindHello.cmake" will be replaced. Removing files [/usr/local/lib/cmake/MyHello/FindHello-"release".cmake].
-- Installing: /usr/local/lib/cmake/MyHello/FindHello.cmake
-- Installing: /usr/local/lib/cmake/MyHello/FindHello-"release".cmake
-- Installing: /usr/local/lib/cmake/MyHello/FindHelloConfig.cmake
-- Installing: /usr/local/lib/cmake/MyHello/FindHelloConfigVersion.cmake
jindong@jindong-virtual-machine:~/SLAM/Chap1/CMake_Exercise/build$ ./sayhello
Hello SLAM
jindong@jindong-virtual-machine:~/SLAM/Chap1/CMake_Exercise/build$
```

# 步骤 2: 安装

```
sudo make install
```

# 安装成功, 并生成 FindHello.cmake 文件 => 可查看目录/usr/local/include 和 /usr/local/lib

# 步骤 3: 测试是否可以通过 find\_package 定位到这个库 MyHello::hello

# 文件夹 test\_folder 中的内容用来测试 find\_package 能否可成功定位到库 MyHello::hello

```
cd ~/../CMake_Exercise/test_folder
```

```
mkdir build
```

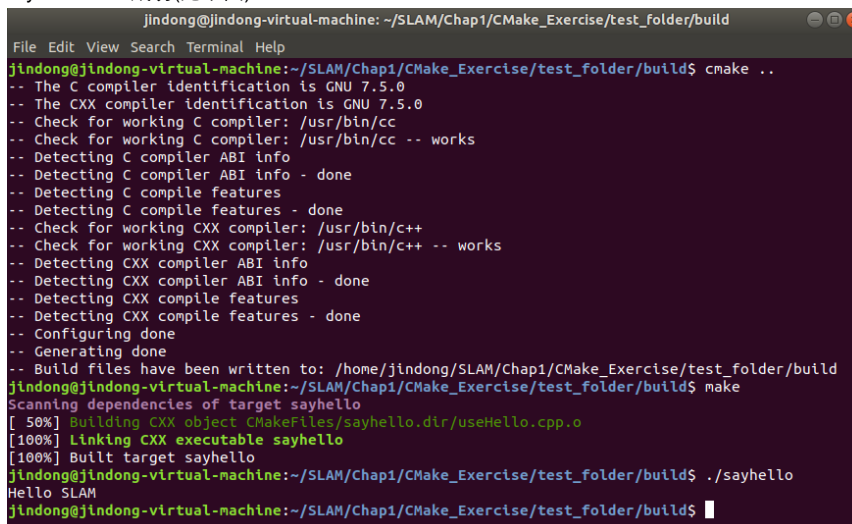
```
cd build
```

```
cmake ..
```

```
make
```

```
./sayhello
```

# 执行可执行文件 sayhello => 成功(见下图)



```
jindong@jindong-virtual-machine: ~/SLAM/Chap1/CMake_Exercise/test_folder/build
File Edit View Search Terminal Help
jindong@jindong-virtual-machine:~/SLAM/Chap1/CMake_Exercise/test_folder/build$ cmake ..
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- 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: /home/jindong/SLAM/Chap1/CMake_Exercise/test_folder/build
jindong@jindong-virtual-machine:~/SLAM/Chap1/CMake_Exercise/test_folder/build$ make
Scanning dependencies of target sayhello
[ 50%] Building CXX object CMakeFiles/sayhello.dir/useHello.cpp.o
[100%] Linking CXX executable sayhello
[100%] Built target sayhello
jindong@jindong-virtual-machine:~/SLAM/Chap1/CMake_Exercise/test_folder/build$ ./sayhello
Hello SLAM
jindong@jindong-virtual-machine:~/SLAM/Chap1/CMake_Exercise/test_folder/build$
```

# 证明其他用户可以通过 find\_package 找到我的库



## 5. gflags, glog, gtest 的使用

1)

**gtest 安装步骤:**

```
$ sudo apt-get install libgtest-dev
$ cd /usr/src/gtest
$ sudo mkdir build
$ cd build
$ sudo cmake ..
$ sudo make
$ sudo cp libgtest*.a /usr/local/lib
```

**gflags 安装步骤:**

```
$ git clone https://github.com/gflags/gflags.git
$ cd gflags
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install
```

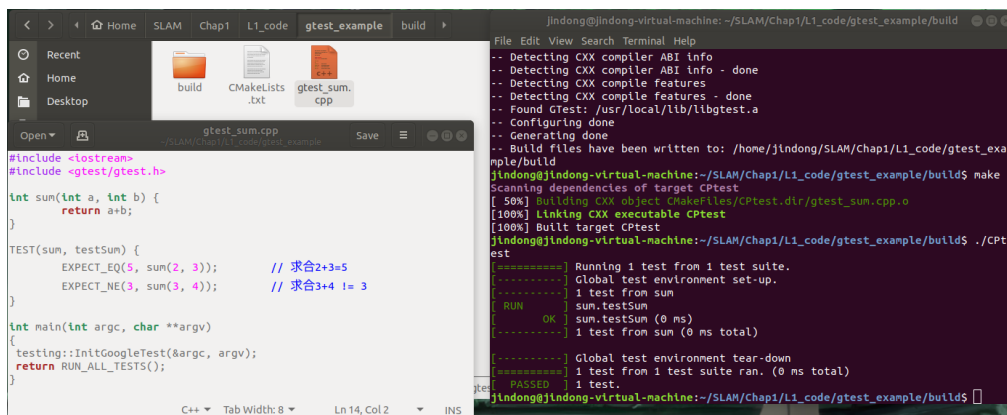
**glog 安装步骤: (glog 安装失败!!!)**

```
$ git clone https://github.com/google/glog
$ sudo apt-get install autoconf automake libtool
$ cd glog
$ mkdir build
$ cd build
$ cmake ..
$ make -j 24
$ sudo make install
```

2) **glog 安装失败, 故未能完成第 2 小题**

3) 书写了一个 gflags 测试程序, 在文件夹 useHello\_with\_gflags 内

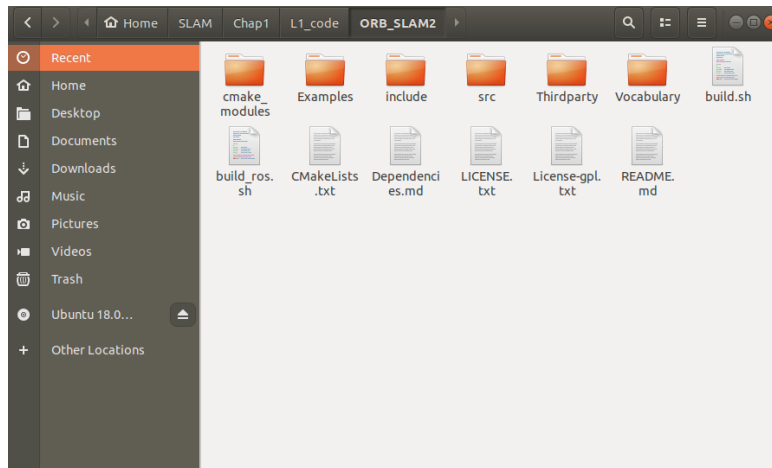
4) 书写了一个 gtest 单元测试程序, 在文件夹 gtest\_example 内。编译结束后运行可执行文件 Cptest, 结果如下图所示, 发现 gtest 工作正常。



```
File Edit View Search Terminal Help
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features - done
-- Found GTest: /usr/local/lib/libgtest.a
-- Configuring done
-- Generating done
-- Build files have been written to: /home/jindong/SLAM/Chap1/L1_code/gtest_example/build
jindong@jindong-virtual-machine:~/SLAM/Chap1/L1_code/gtest_example/build$ make
Scanning dependencies of target Cptest
[ 50%] Building CXX object CMakeFiles/Cptest.dir/gtest_sum.cpp.o
[100%] Linking CXX executable Cptest
jindong@jindong-virtual-machine:~/SLAM/Chap1/L1_code/gtest_example/build$ ./Cptest
===== Running 1 test from 1 test suite.
----- Global test environment set-up.
----- 1 test from sum
[ RUN      ] sum.testSum
[ OK       ] sum.testSum (0 ms)
----- 1 test from sum (0 ms total)
----- Global test environment tear-down
===== 1 test from 1 test suite ran. (0 ms total)
[ PASSED  ] 1 test.
jindong@jindong-virtual-machine:~/SLAM/Chap1/L1_code/gtest_example/build$
```

## 6. 理解 ORB\_SLAM2 框架

1) 下载完成截图如下



2)

(a)

新生成的库文件共 1 个，为动态链接库  $\${PROJECT\_NAME}$  -> libORB\_SLAM2.so

生成的可执行文件共 6 个，分别为 rgbd\_tum, stereo\_kitti, stereo\_euroc, mono\_tum, mono\_kitti, mono\_euroc

(b)

include: 所有头文件

src: 所有源文件

Examples: 4 个有功能包，分别为 Monocular, RGB-D, ROS, Stereo, 每个功能包都能生成一个可执行文件，实现特定的功能。

(c)

每一个可执行文件都链接到同样的一些动态库

$\${PROJECT\_NAME}$

$\${OpenCV\_LIBS}$

$\${EIGEN3\_LIBS}$

$\${Pangolin\_LIBRARIES}$

$\${PROJECT\_SOURCE\_DIR}/Thirdparty/DBow2/lib/libDBow2.so$

$\${PROJECT\_SOURCE\_DIR}/Thirdparty/g2o/lib/libg2o.so$

## 7. \*使用摄像头或视频运行 ORB-SLAM2

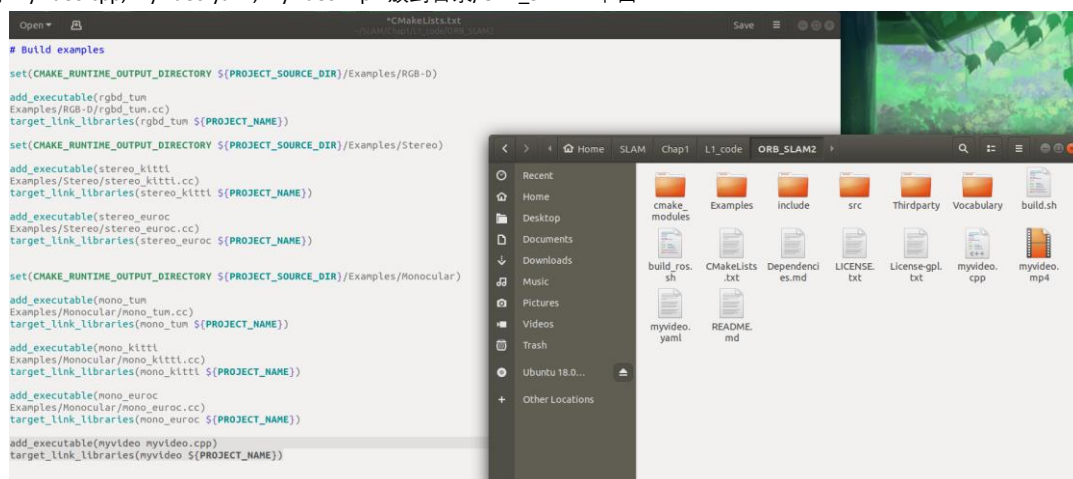
1) 编译失败!!!

2) 修改 CMakeLists.txt 如下图所示，在文件末尾加上：

add\_executable(myvideo myvideo.cpp)

target\_link\_libraries(myvideo  $\${PROJECT\_NAME}$ )

并且将 myvideo.cpp, myvideo.yaml, myvideo.mp4 放到目录/ORB\_SLAM2 下面



参考:

[1] Ubuntu 桌面入门指南:

<https://wiki.ubuntu.org.cn/Ubuntu%E6%A1%8C%E9%9D%A2%E5%85%A5%E9%97%A8%E6%8C%87%E5%8D%97#2.4.1..E2.80.83APT>

[2] 详解 Ubuntu 软件源:

<https://www.jianshu.com/p/0c8ae4597bc6>

[3] ubuntu 软件源的解释及配置:

<https://blog.csdn.net/nei504293736/article/details/7855054>

[4] Ubuntu 中关于环境变量 PATH 的作用

[https://blog.csdn.net/qg\\_44986938/article/details/106288159](https://blog.csdn.net/qg_44986938/article/details/106288159)

[5] Ubuntu 下的环境变量 LD\_LIBRARY\_PATH

<https://blog.csdn.net/lqhbupt/article/details/7875112>

[6] ldconfig 命令的作用

<https://blog.csdn.net/wooin/article/details/580679>

[7] 一文带你彻底搞懂 Linux 文件权限管理

<https://segmentfault.com/a/1190000039202476>

[8] 鸟哥的 Linux 私房菜基础学习篇(第三版) – page. 184

[9] 【Linux】除了 gcc，还有哪些常用的编译器

[https://blog.csdn.net/swag\\_wg/article/details/90315353](https://blog.csdn.net/swag_wg/article/details/90315353)

[10] Visual Simultaneous Localization and Mapping: A Survey; Jorge Fuentes-Pacheco, Jose Ruiz Ascencio, J. M. Rendon-Mancha

[11] 视觉 SLAM 的 VO 与后端图优化理论整理

<https://zhuanlan.zhihu.com/p/360225462>

[12] 基于单目视觉的同时定位与地图构建方法综述; 刘浩敏, 章国锋, 鲍虎军