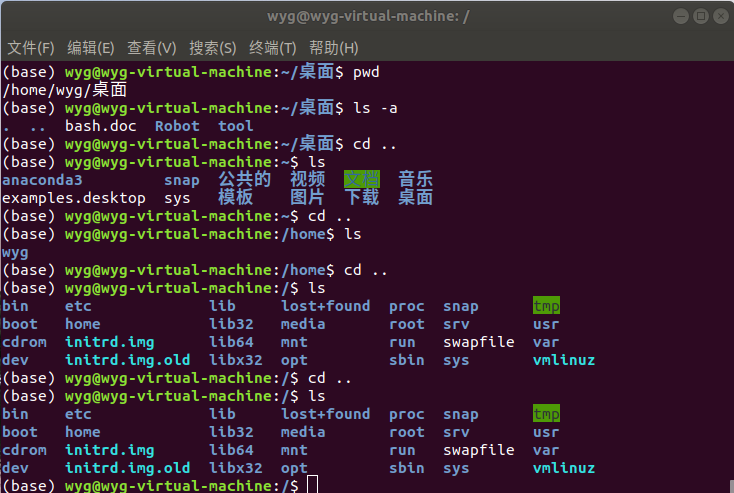
homework1

❖ 安装Ubuntu14.04或以上版本的linux系统并熟悉基本操作；

**（一）查看系统目录结构**

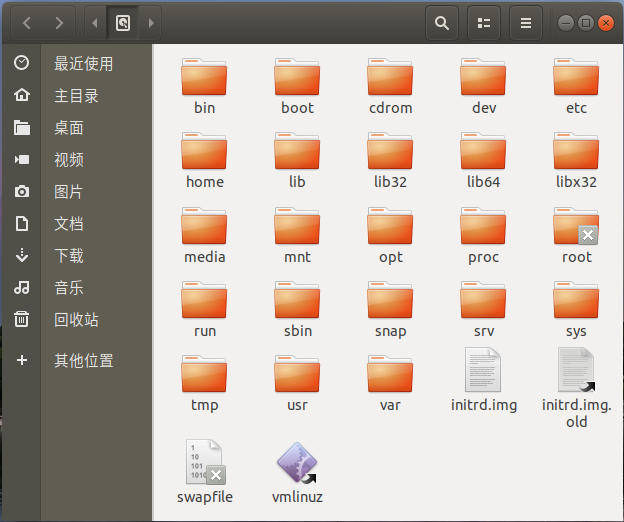
1）查看桌面在系统目录的位置，并观察目录结构



2）查看所有位置的存储设备



3）Linux的根目录



（1）/bin 用户二进制文件：包含二进制可执行文件，系统所有用户可执行文件都在这个文件夹里，例如：ls，cp，ping等。

（2）/sbin 系统二进制文件：包含二进制可执行文件，但只能由系统管理员运行，对系统进行维护。

（3）/etc 配置文件：包含所有程序配置文件，也包含了用于启动/停止单个程序的启动和关闭shell脚本。

（4）/dev 设备文件：包含终端所有设备，USB或连接到系统的任何设备。

（5）/proc 进程信息：包含系统进程的相关信息。

（6）/var 变量文件：可以找到内容可能增长的文件。

（7）/tmp 临时文件：包含系统和用户创建的临时文件。系统重启时，目录清空。

（8）/usr 用户程序

包含二进制文件、库文件、文档和二级程序的源代码。

/usr/bin中包含用户程序的二进制文件。如果你在/bin中找不到用户二进制文件，到/usr/bin目录看看。例如：at、awk、cc、less、scp。

/usr/sbin中包含系统管理员的二进制文件。如果你在/sbin中找不到系统二进制文件，到/usr/sbin目录看看。例如：atd、cron、sshd、useradd、userdel。

/usr/lib中包含了/usr/bin和/usr/sbin用到的库。

/usr/local中包含了从源安装的用户程序。例如，当你从源安装Apache，它会在/usr/local/apache2中。

（9）/home HOME目录：所有用户用来存档他们的个人档案。

（10）/boot 引导加载程序文件：包含引导加载程序相关的文件。

内核的initrd、vmlinux、grub文件位于/boot下。

（11）/lib 系统库：包含支持位于/bin和/sbin下的二进制文件的库文件.

库文件名为 ld或lib.so.\*

（12）/opt 可选的附加应用程序：opt代表opitional；

包含从个别厂商的附加应用程序。

附加应用程序应该安装在/opt/或者/opt/的子目录下。

（13）/mnt 挂载目录：临时安装目录，系统管理员可以挂载文件系统。

（14）/media 可移动媒体设备：用于挂载可移动设备的临时目录。

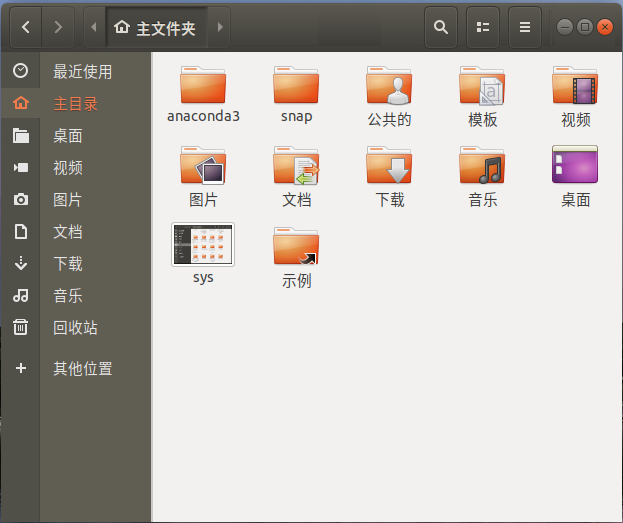
举例来说，挂载CD-ROM的/media/cdrom，挂载软盘驱动器的/media/floppy;

（15）/srv 服务数据：srv代表服务。

包含服务器特定服务相关的数据。

例如，/srv/cvs包含cvs相关的数据。

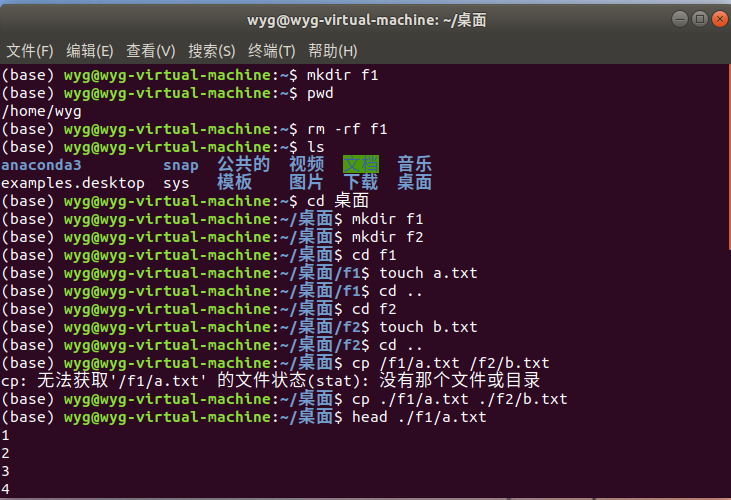
4）主文件夹，位于根目录的/home/username下：所有用户用来存档他们的个人档案，每个用户对应一个主文件夹。



都是按照功能来命名，文件夹名称语言随系统语言同步变化

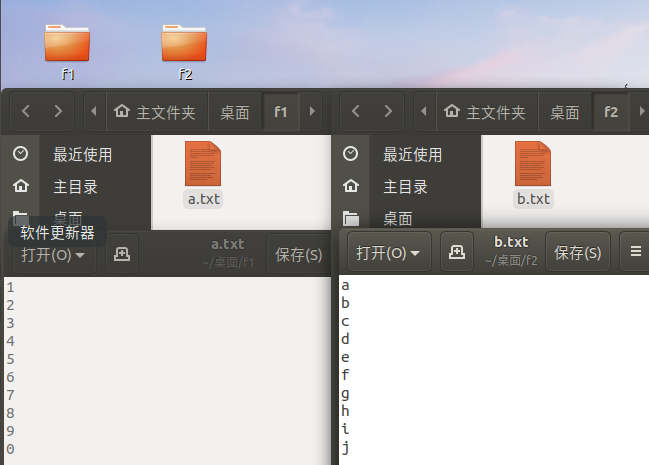
**（二）Linux基本操作**

1）增删查改文件或文件夹

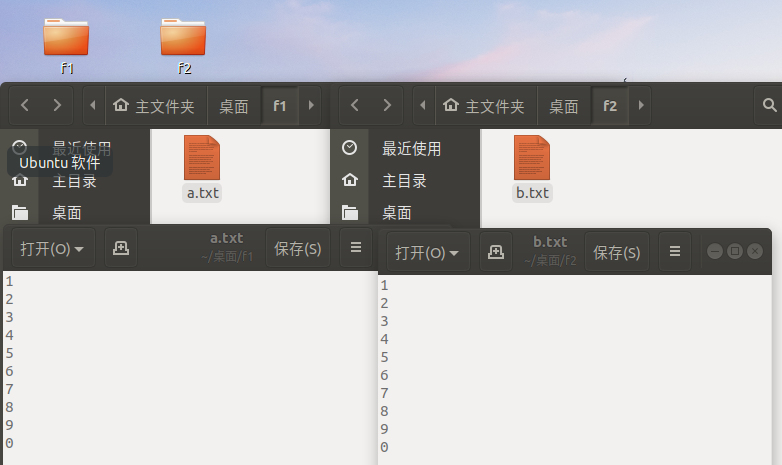


2）练习：将f1文件夹中的a.txt的内容复制到f2文件夹中的b.txt中

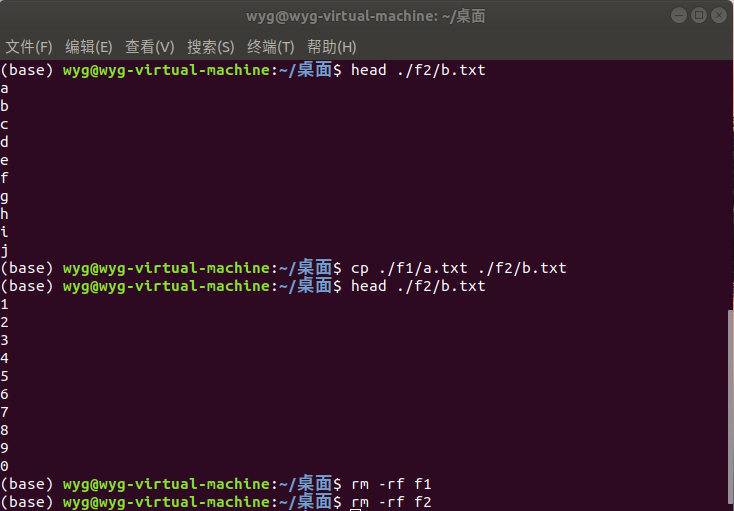
复制前的文件内容



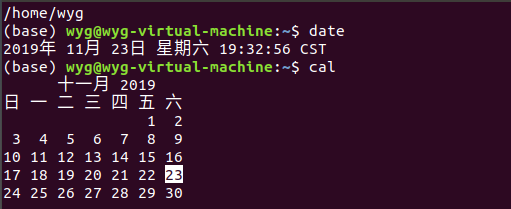
复制后的文件内容



查看文件的前十行



3）查看日期与时间



**（三）Vim编辑器**

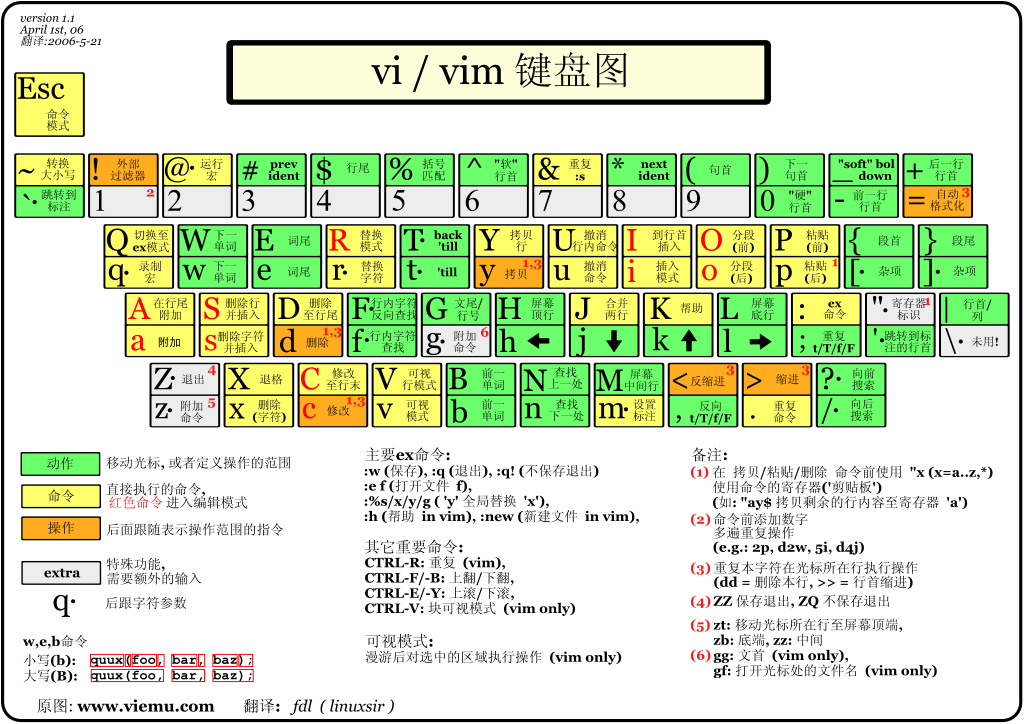
1）删除vi编辑器，安装vim编辑器



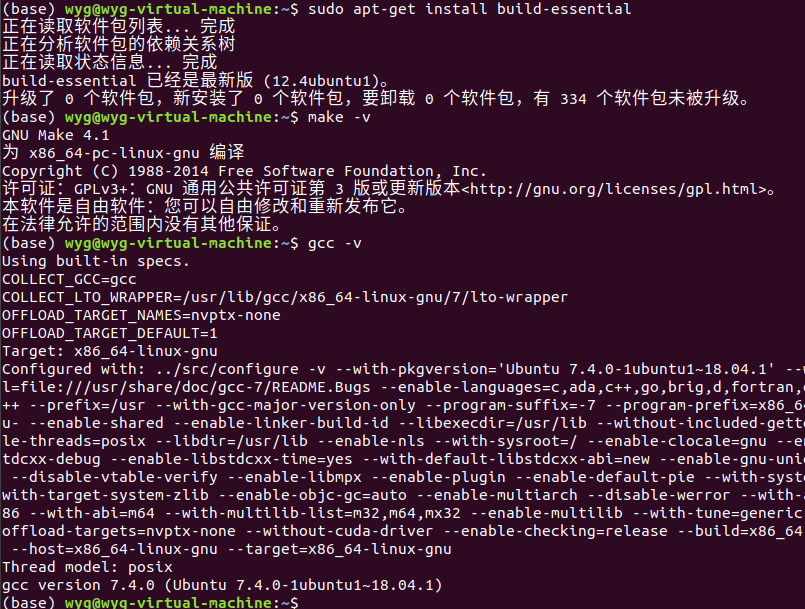
2）进入vim编辑器教程



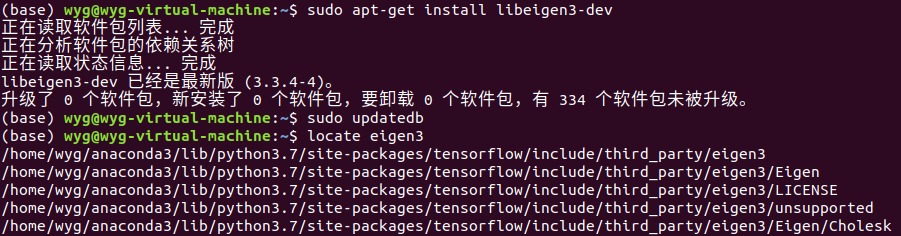
进去了解了解了集本操作，就是移动光标有点不习惯，感觉鼠标点击会快一些



**（四）安装gcc&g++**



**（五）安装 Eigen**



❖ 书写⼀个由 cmake 组织的 C++ ⼯程，并书写CMakeLists.txt，要求如下：

**1. 参考示例代码，编译 include/hello.h 和 src/hello.c 构成libhello.so 库。其中hello.c 中提供⼀个函数 sayHello()，调⽤此函数时往屏幕输出⼀⾏“Hello ”**

**2. ⽂件 useHello.c 中含有⼀个 main 函数，它可以编译成⼀个可执⾏⽂件，名为“sayhello”。**

**3. 默认⽤ Release 模式编译这个⼯程。**

**4. 支持使⽤命令 sudo make install，该命令将 hello.h 放**

**⾄/usr/local/include/下，将 libhello.so 放 ⾄/usr/local/lib/下**

1)准备程序文件

由题意文件目录结构如下：

.

├── build

├── CMakeLists.txt

├── include

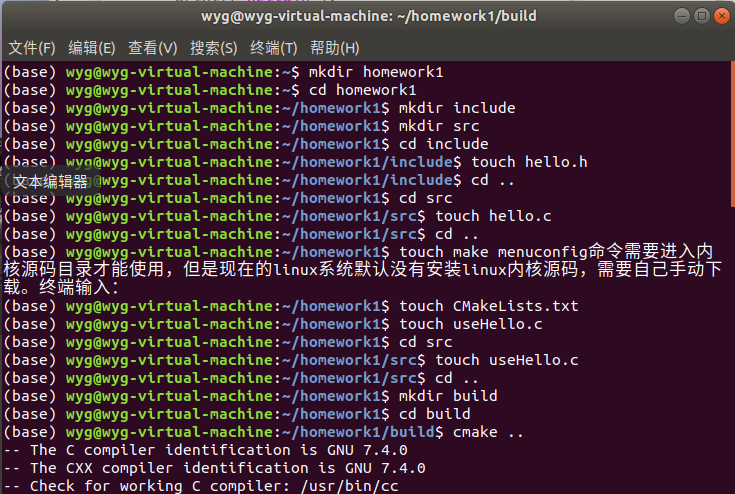
│ └── hello.h

└── src

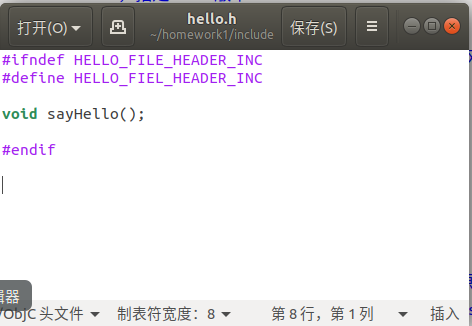
├── hello.c

└── useHello.c

生成目录结构



头文件hello.h，如下所示：



源文件hello.c，如下所示：



useHello.c主函数，如下所示：



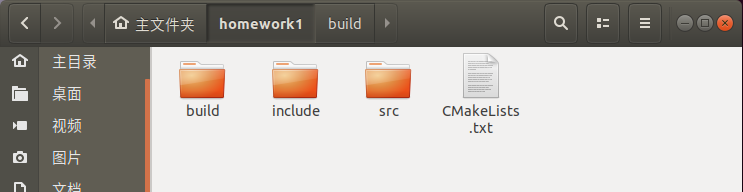
2) 编写CMakeLists.txt

接下来编写CMakeLists.txt文件，该文件放在和src，include的同级目录，实际方哪里都可以，只要里面编写的路径能够正确指向就好了。CMakeLists.txt文件，如下所示：

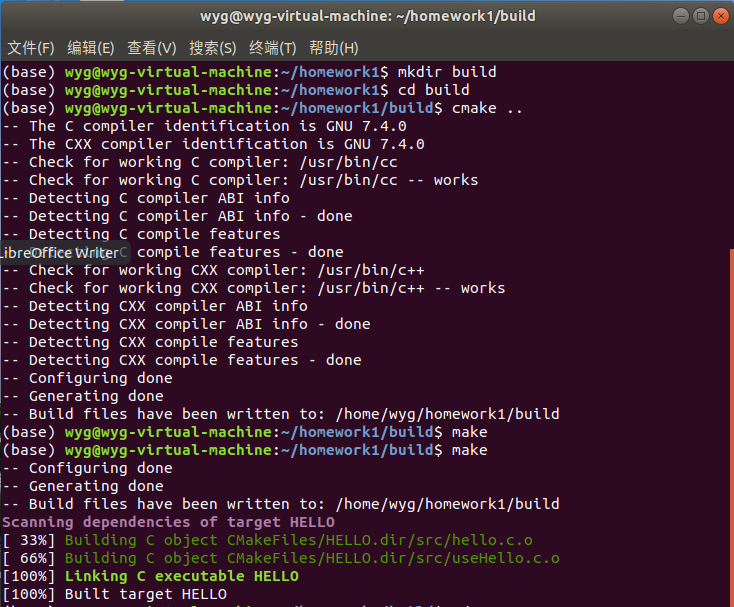


3) 编译和运行程序

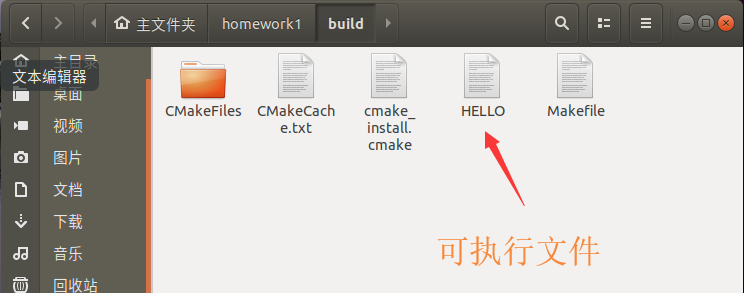
准备好了以上的所有材料，接下来，就可以编译了，由于编译中出现许多中间的文件，因此最好新建一个独立的目录build，在该目录下进行编译：



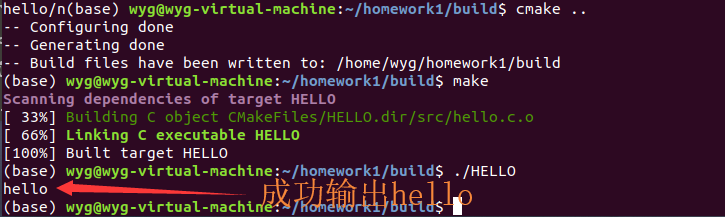
编译步骤如下所示：



注意在build的目录下生成了一个可执行的文件HELLO（忘记改成sayhello了）



运行获取结果如下：



4）最后执行make install

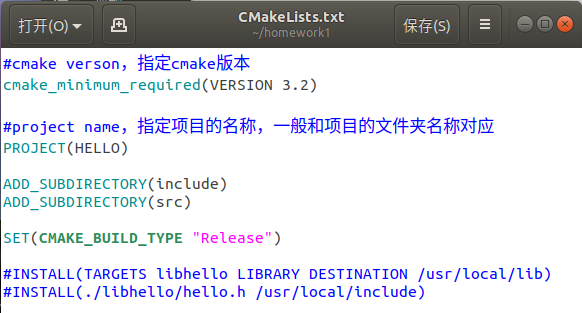
刚开始失败了，于是找教程，需要重写3个cmakelists.txt文件

在对应路径下新建文件



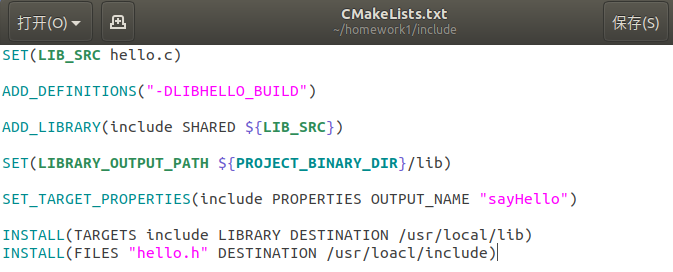
总目录下的cmakelists.txt文件

需要添加子目录操作



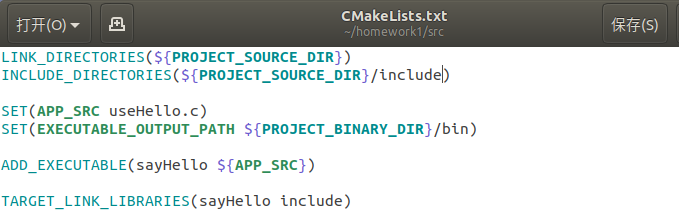
include目录下的cmakelists.txt文件

设置了一些依赖关系

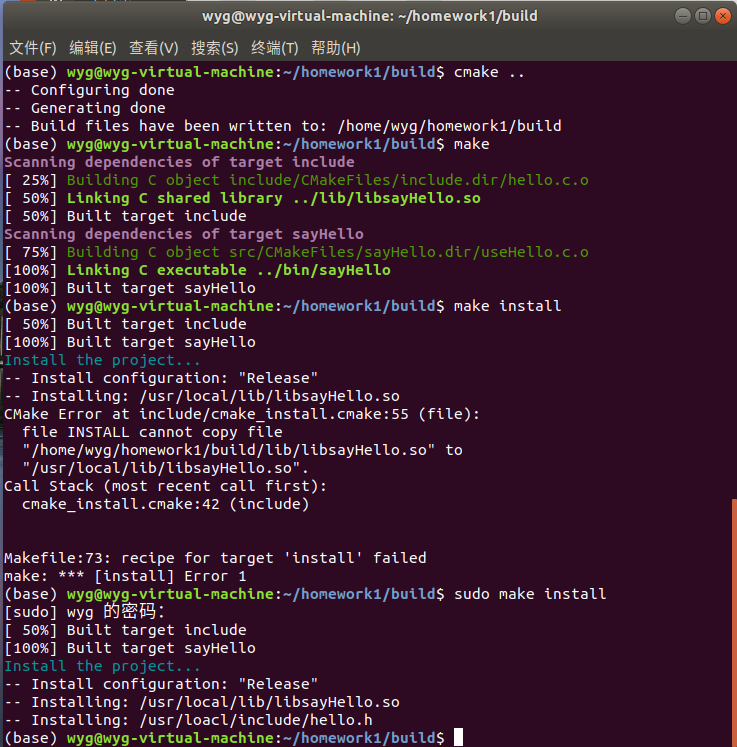


src目录下的cmakelists.txt文件

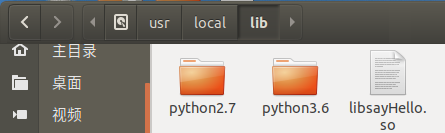
也设置了一些依赖关系



编译并添加到库中

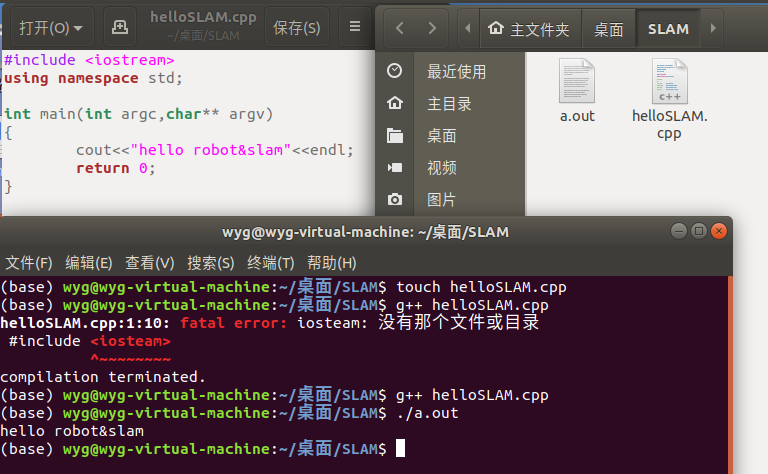


完成安装

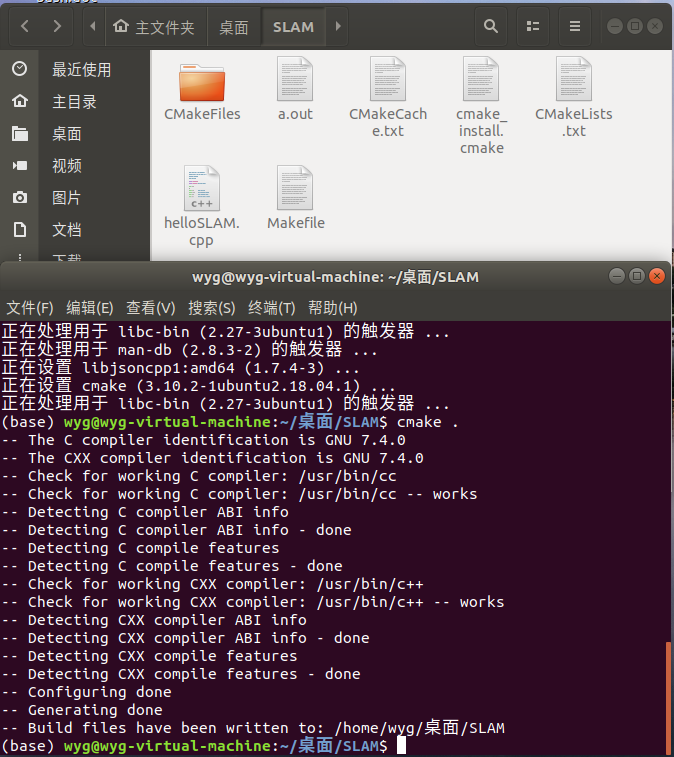


**附.书上示例代码练习**

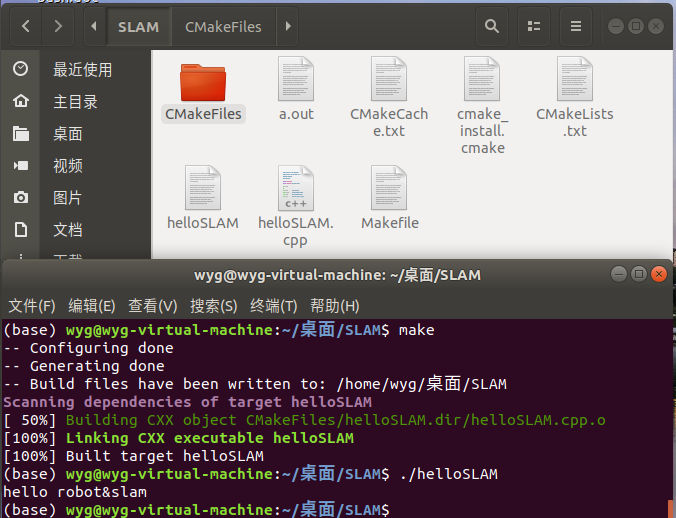
尝试用g++单文件编译，并执行



使用cmake编译



生成了可执行文件



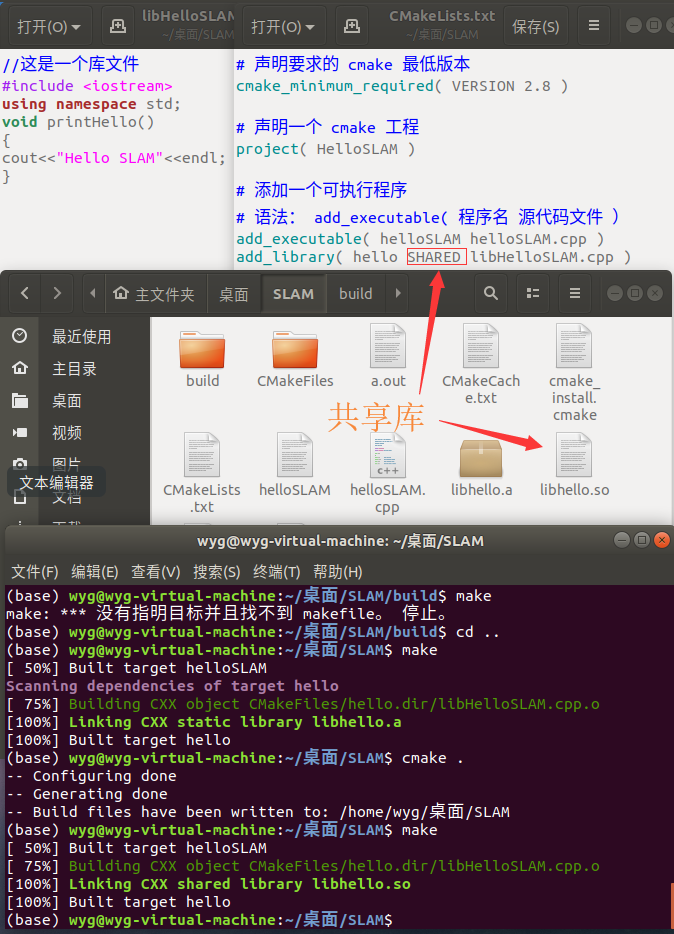
CMakeLists.txt的写法



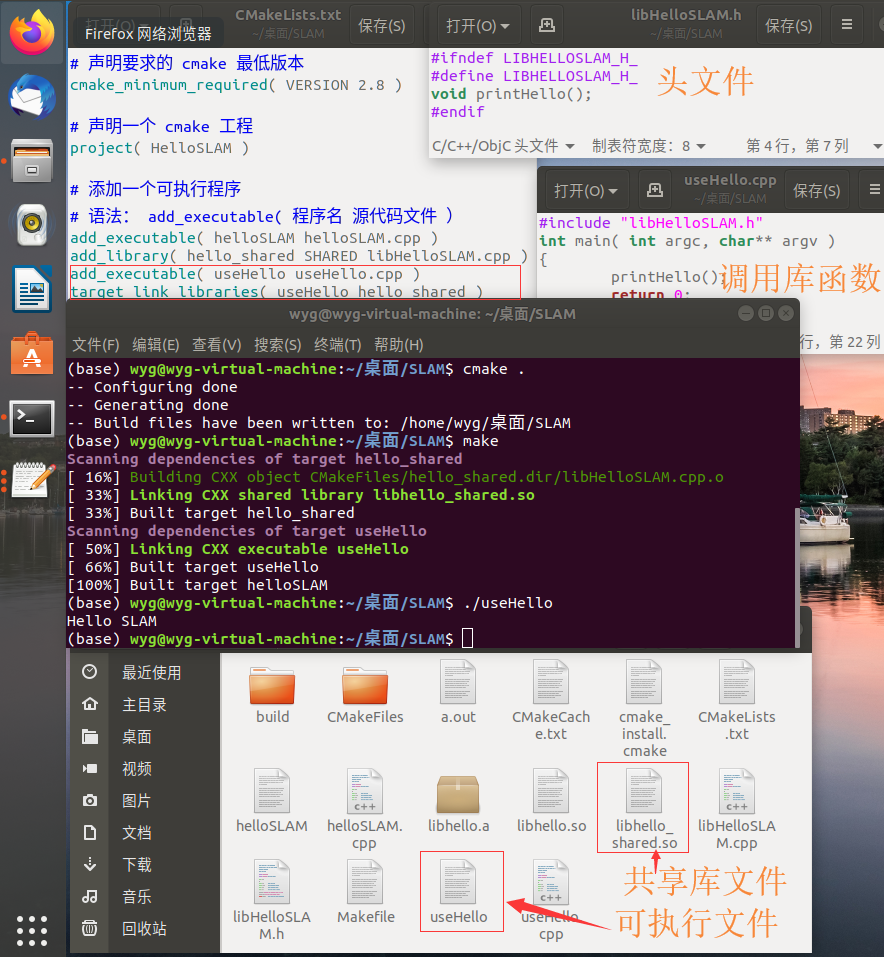
生成静态库：



生成共享库：

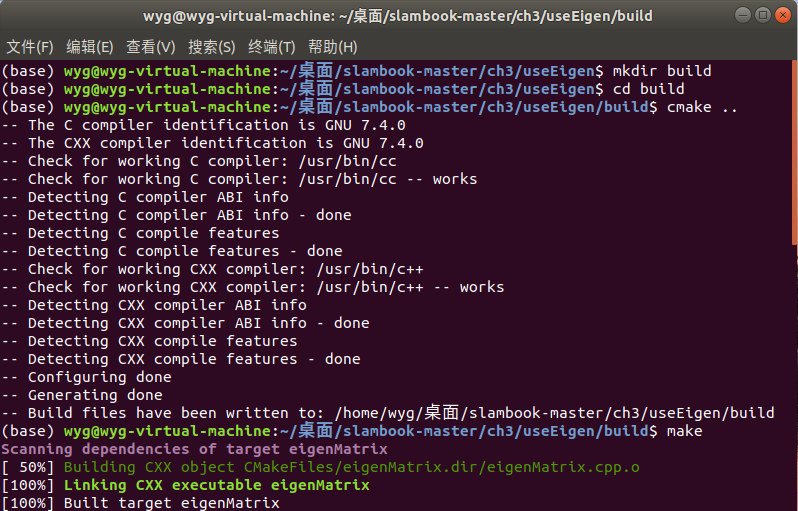


从编译到执行：

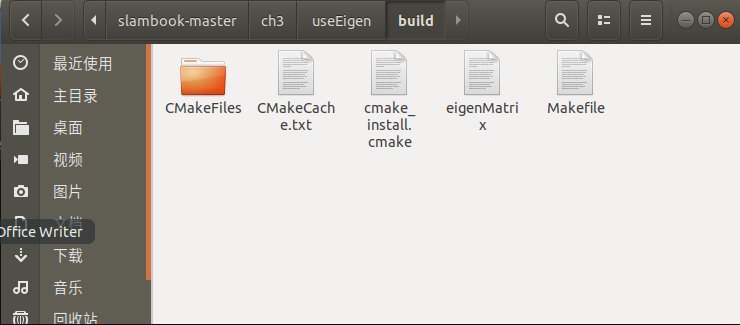


❖ 运行参考书《视觉SLAM十四讲》P45-47的eigenMatrix.cpp程序并熟悉代码；

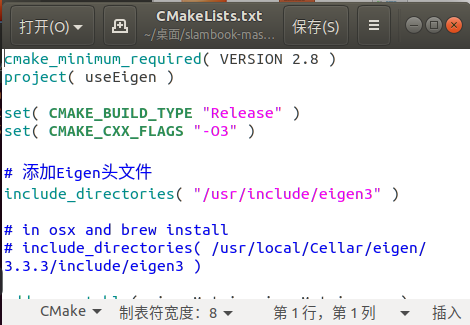
1）编译



创建了build文件，避免编译后的文件太乱

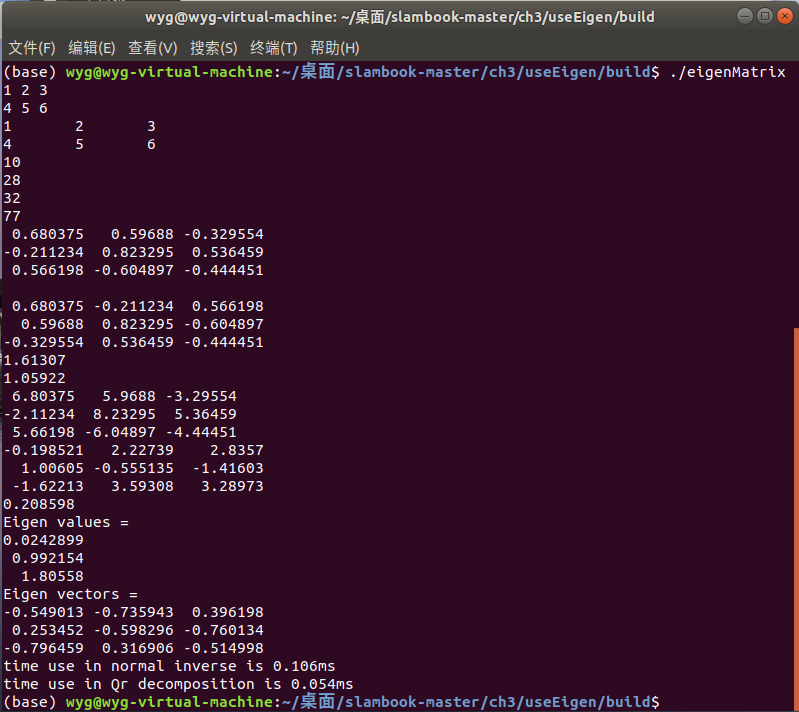


查看CMakeLists.txt包含了eigen3线性代数库



2）运行

查看输出结果



查看eigenMatrix.cpp程序复习

#include <iostream>

using namespace std;

#include <ctime>

// Eigen 部分

#include <Eigen/Core>

// 稠密矩阵的代数运算（逆，特征值等）

#include <Eigen/Dense>

#define MATRIX\_SIZE 50

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 本程序演示了 Eigen 基本类型的使用

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main( int argc, char\*\* argv )

{

    // Eigen 中所有向量和矩阵都是Eigen::Matrix，它是一个模板类。它的前三个参数为：数据类型，行，列

    // 声明一个2\*3的float矩阵

    Eigen::Matrix<float, 2, 3> matrix\_23;

    // 同时，Eigen 通过 typedef 提供了许多内置类型，不过底层仍是Eigen::Matrix

    // 例如 Vector3d 实质上是 Eigen::Matrix<double, 3, 1>，即三维向量

    Eigen::Vector3d v\_3d;

    // 这是一样的

    Eigen::Matrix<float,3,1> vd\_3d;

    // Matrix3d 实质上是 Eigen::Matrix<double, 3, 3>

    Eigen::Matrix3d matrix\_33 = Eigen::Matrix3d::Zero(); //初始化为零

    // 如果不确定矩阵大小，可以使用动态大小的矩阵

    Eigen::Matrix< double, Eigen::Dynamic, Eigen::Dynamic > matrix\_dynamic;

    // 更简单的

    Eigen::MatrixXd matrix\_x;

    // 这种类型还有很多，我们不一一列举

    // 下面是对Eigen阵的操作

    // 输入数据（初始化）

    matrix\_23 << 1, 2, 3, 4, 5, 6;

    // 输出

    cout << matrix\_23 << endl;

    // 用()访问矩阵中的元素

    for (int i=0; i<2; i++) {

        for (int j=0; j<3; j++)

            cout<<matrix\_23(i,j)<<"\t";

        cout<<endl;

    }

    // 矩阵和向量相乘（实际上仍是矩阵和矩阵）

    v\_3d << 3, 2, 1;

    vd\_3d << 4,5,6;

    // 但是在Eigen里你不能混合两种不同类型的矩阵，像这样是错的

    // Eigen::Matrix<double, 2, 1> result\_wrong\_type = matrix\_23 \* v\_3d;

    // 应该显式转换

    Eigen::Matrix<double, 2, 1> result = matrix\_23.cast<double>() \* v\_3d;

    cout << result << endl;

    Eigen::Matrix<float, 2, 1> result2 = matrix\_23 \* vd\_3d;

    cout << result2 << endl;

    // 同样你不能搞错矩阵的维度

    // 试着取消下面的注释，看看Eigen会报什么错

    // Eigen::Matrix<double, 2, 3> result\_wrong\_dimension = matrix\_23.cast<double>() \* v\_3d;

    // 一些矩阵运算

    // 四则运算就不演示了，直接用+-\*/即可。

    matrix\_33 = Eigen::Matrix3d::Random();      // 随机数矩阵

    cout << matrix\_33 << endl << endl;

    cout << matrix\_33.transpose() << endl;      // 转置

    cout << matrix\_33.sum() << endl;            // 各元素和

    cout << matrix\_33.trace() << endl;          // 迹

    cout << 10\*matrix\_33 << endl;               // 数乘

    cout << matrix\_33.inverse() << endl;        // 逆

    cout << matrix\_33.determinant() << endl;    // 行列式

    // 特征值

    // 实对称矩阵可以保证对角化成功

    Eigen::SelfAdjointEigenSolver<Eigen::Matrix3d> eigen\_solver ( matrix\_33.transpose()\*matrix\_33 );

    cout << "Eigen values = \n" << eigen\_solver.eigenvalues() << endl;

    cout << "Eigen vectors = \n" << eigen\_solver.eigenvectors() << endl;

    // 解方程

    // 我们求解 matrix\_NN \* x = v\_Nd 这个方程

    // N的大小在前边的宏里定义，它由随机数生成

    // 直接求逆自然是最直接的，但是求逆运算量大

    Eigen::Matrix< double, MATRIX\_SIZE, MATRIX\_SIZE > matrix\_NN;

    matrix\_NN = Eigen::MatrixXd::Random( MATRIX\_SIZE, MATRIX\_SIZE );

    Eigen::Matrix< double, MATRIX\_SIZE,  1> v\_Nd;

    v\_Nd = Eigen::MatrixXd::Random( MATRIX\_SIZE,1 );

    clock\_t time\_stt = clock(); // 计时

    // 直接求逆

    Eigen::Matrix<double,MATRIX\_SIZE,1> x = matrix\_NN.inverse()\*v\_Nd;

    cout <<"time use in normal inverse is " << 1000\* (clock() - time\_stt)/(double)CLOCKS\_PER\_SEC << "ms"<< endl;

    // 通常用矩阵分解来求，例如QR分解，速度会快很多

    time\_stt = clock();

    x = matrix\_NN.colPivHouseholderQr().solve(v\_Nd);

    cout <<"time use in Qr decomposition is " <<1000\*  (clock() - time\_stt)/(double)CLOCKS\_PER\_SEC <<"ms" << endl;

    return 0;

}