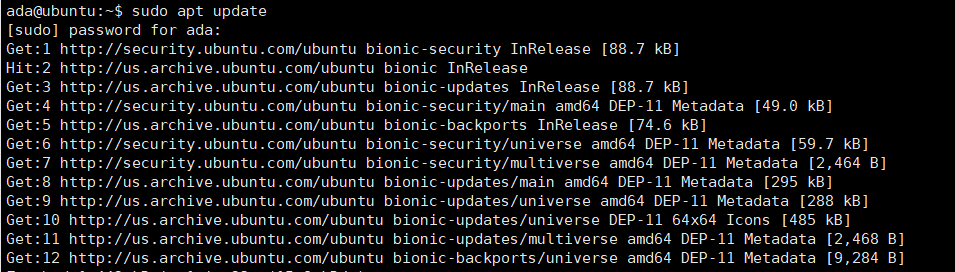
# 易栈服务端

## 项目环境的搭建

### 配置系统环境

默认的Ubuntu是使用官方的源



更新比较慢，非常卡，而且容易失败

修改更新源到清华源

<https://mirrors.tuna.tsinghua.edu.cn/>





远程连接到Ubuntu系统

输入sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak

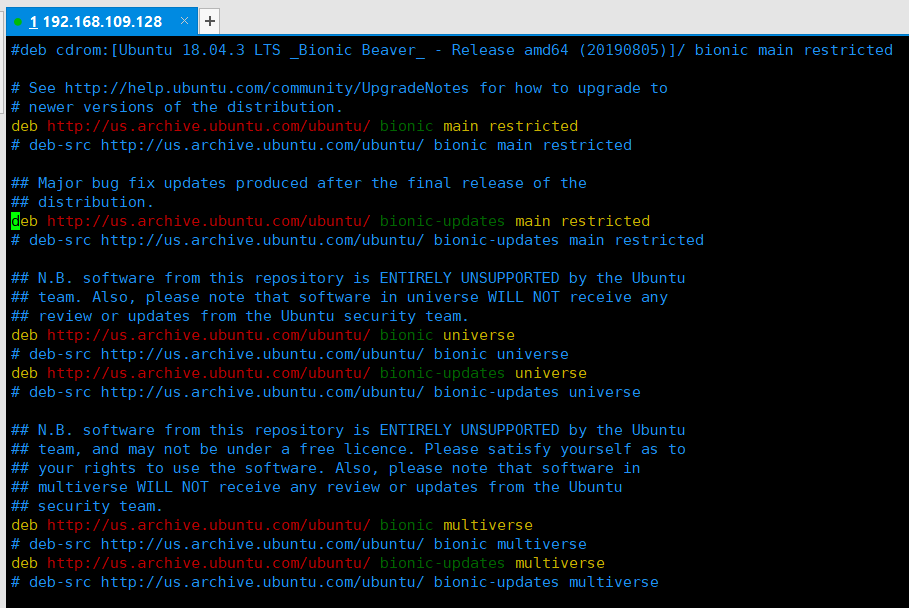
进行源的备份

输入sudo vim /etc/apt/sources.list

准备编辑源，如下图

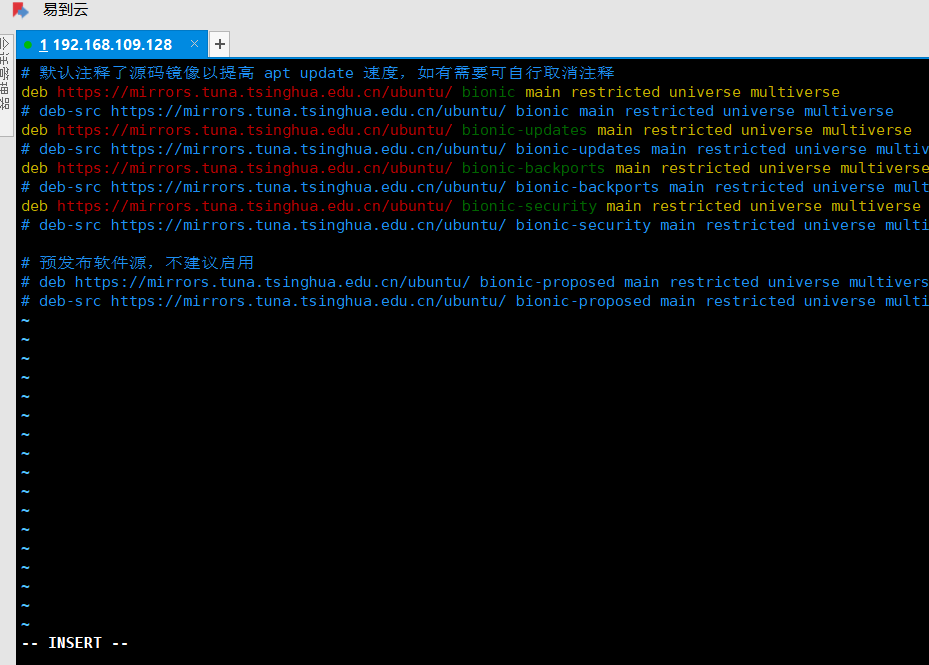


下面内容全部删除（输入dd删除一行，按住不放，不停的删除一行）



然后输入a进入插入模式

把之前的内容复制后，可以直接粘贴过来

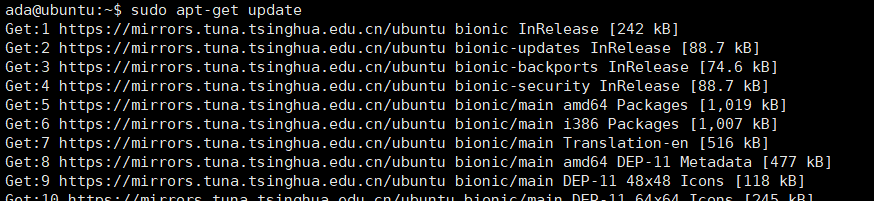


然后按一下ESC

再输入:wq回车，即可保存退出



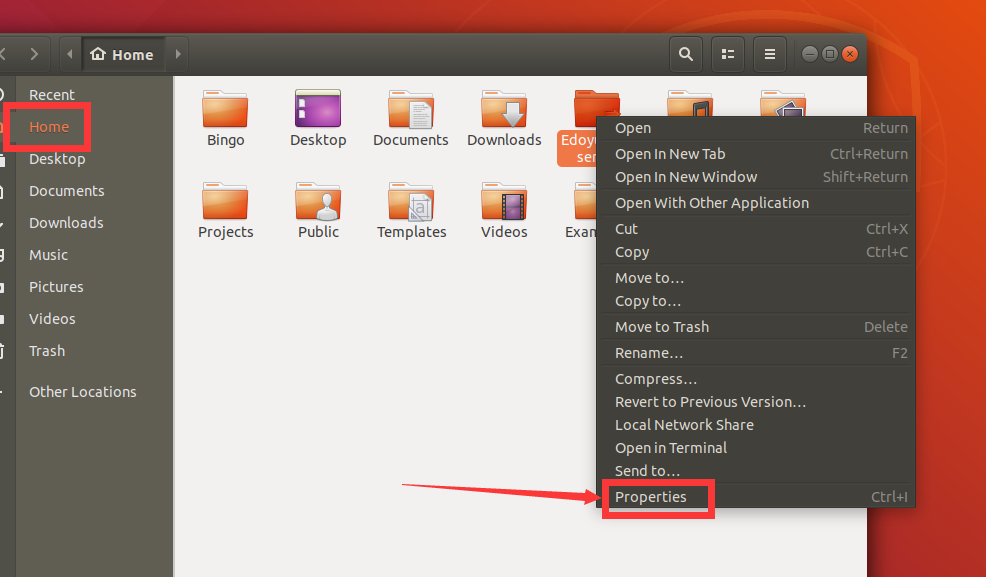
再次更新的时候



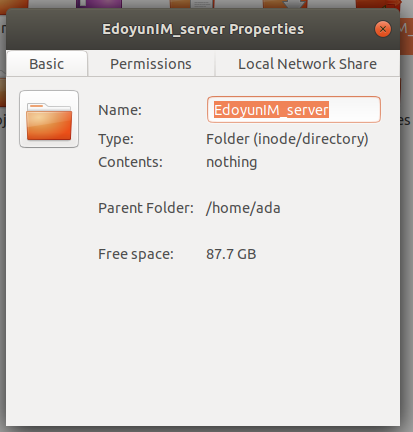
可以看到已经是清华的源了

### 局域网共享文件夹

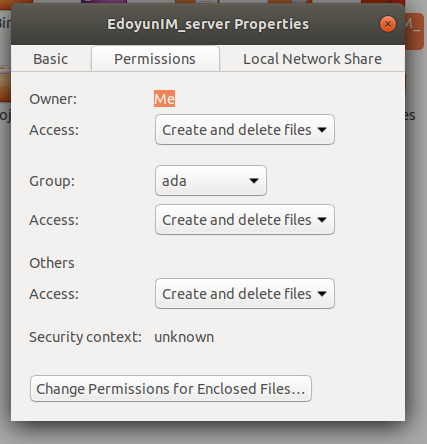
在home下新建一个文件夹EdoyunIM\_server，右键单击选择属性（Properties）

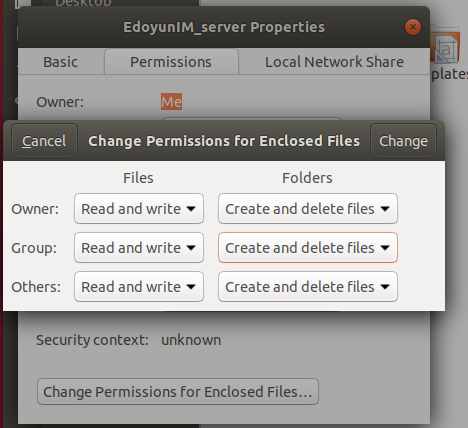


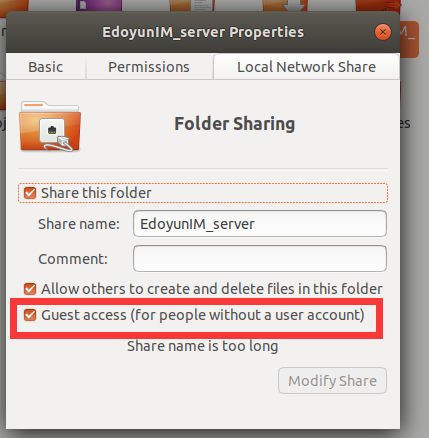
可以看到下面的内容



分别设置Permissions（权限）和Local Network Share



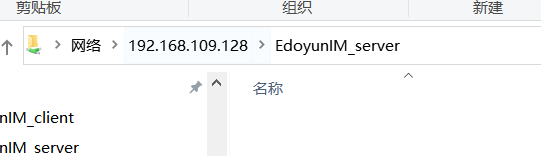




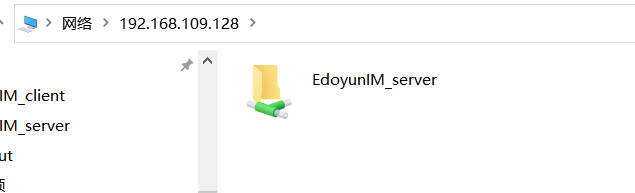
一定要开启客户访问，否则就需要创建ada组的新账号，使用密码访问

可能会需要安装一些服务，选择安装即可

在Windows下打开试一试



然后切换到这个目录

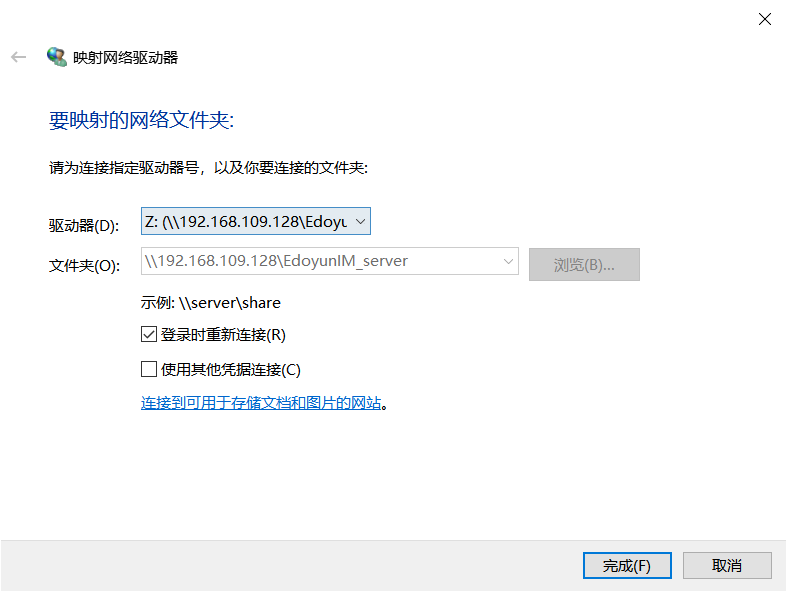


右键单击文件夹

找到 映射网络驱动器



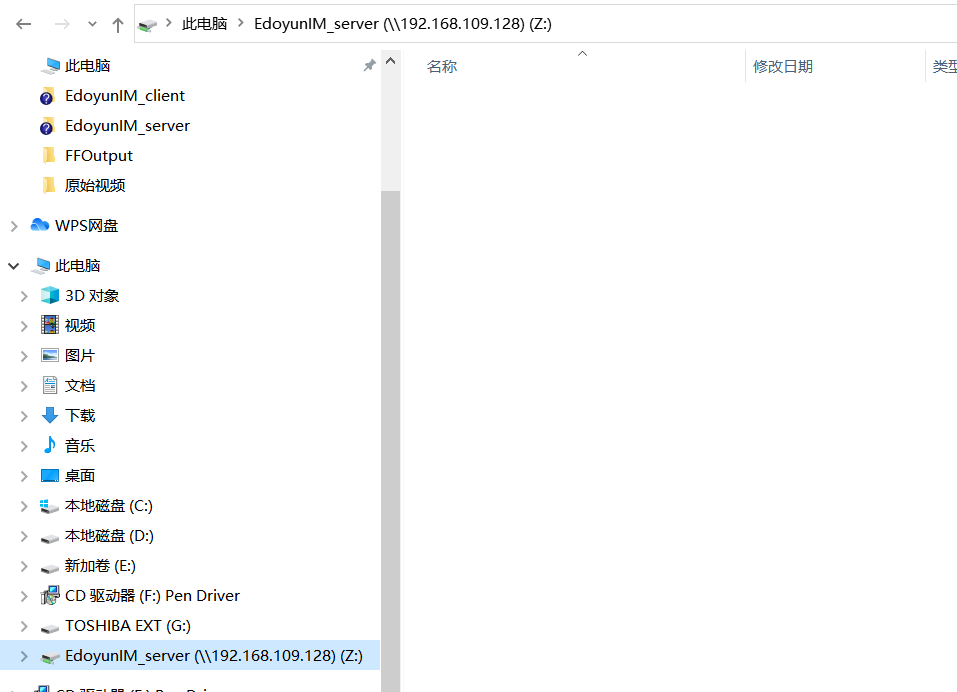
选择Z或者其他你喜欢的驱动器



然后在我的电脑即可看到这个驱动器



打开后，即可看到一个空的文件夹



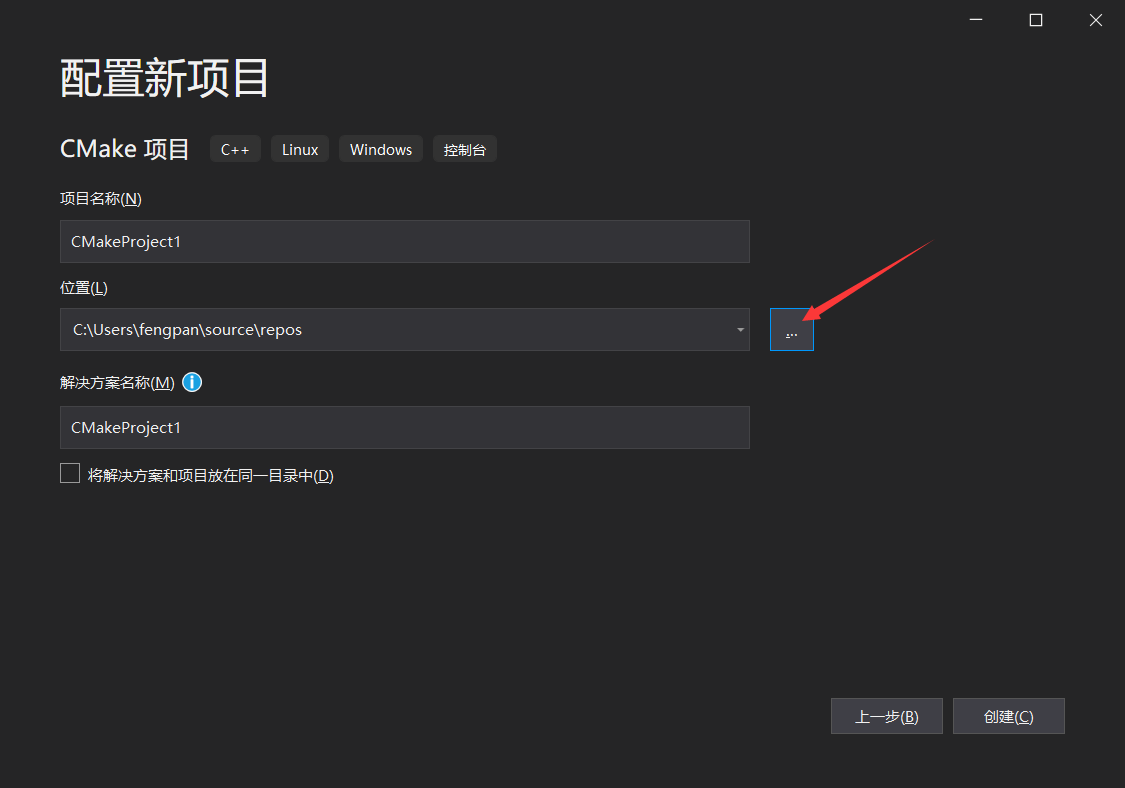
后续的项目，我们就在这里创建

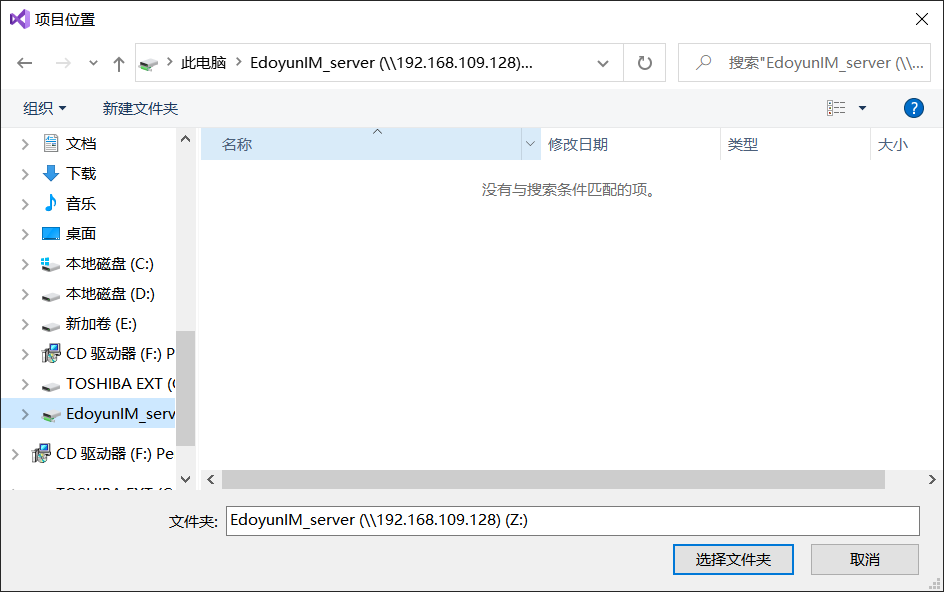
在Windows编辑，在Linux里面编译

### 建立cmake项目



选择项目位置到网络共享文件夹





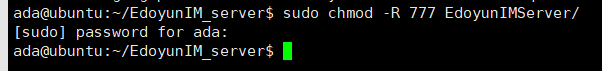
修改项目名为EdoyunIMServer

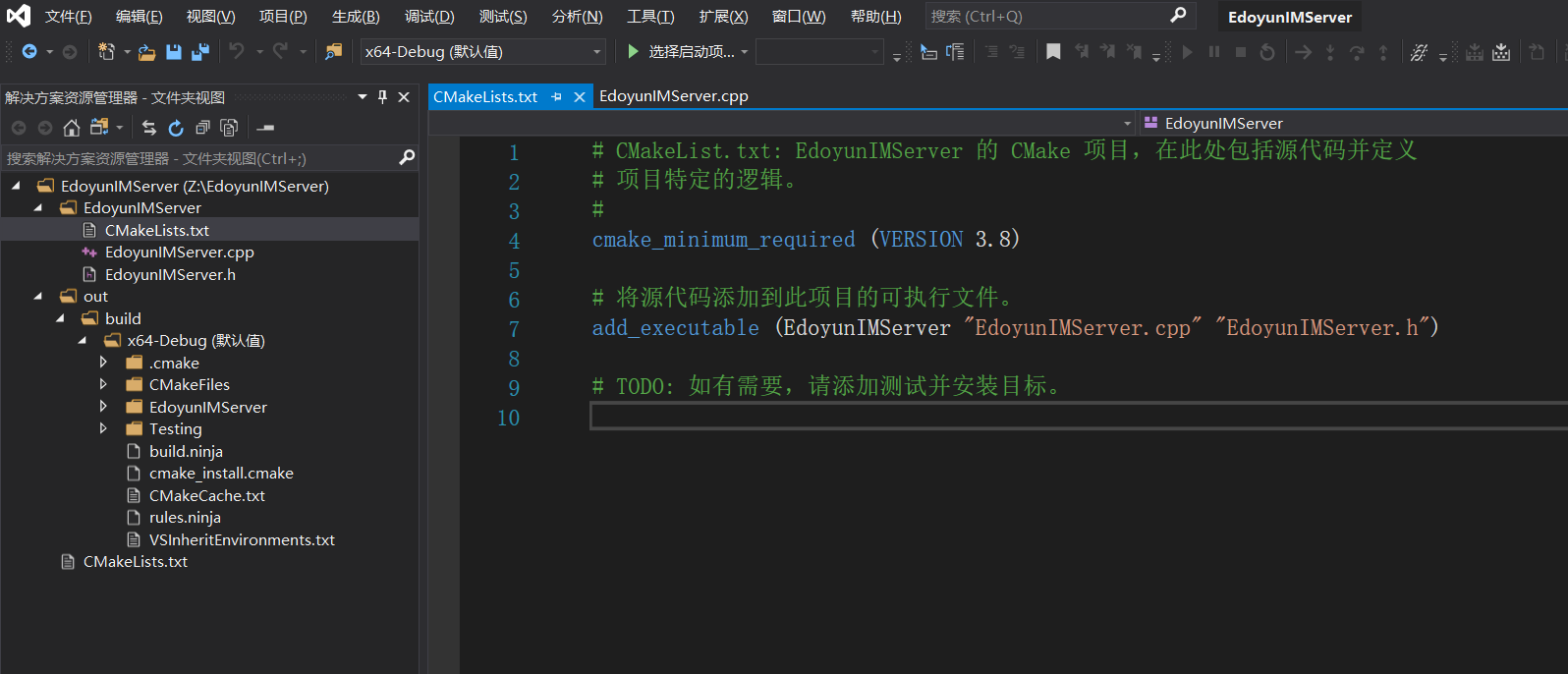


如果提出警告，说权限不足，无法写入或者修改一些文件

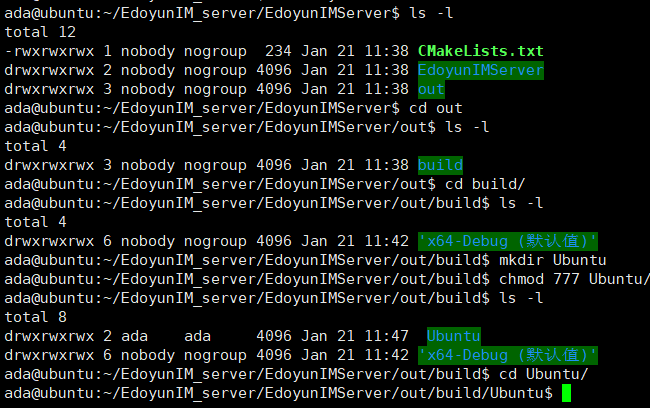
则可以在命令行输入如下命令：

sudo chmod -R 777 /home/ada/EdoyunIMServer/

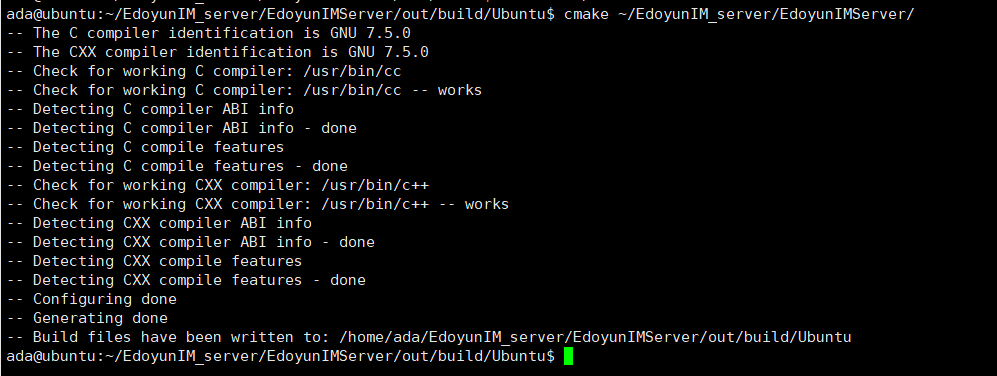




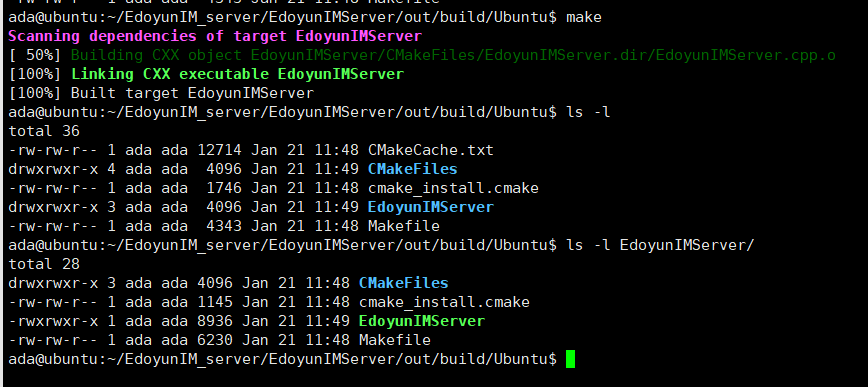
在out/build/建立一个Ubuntu文件夹



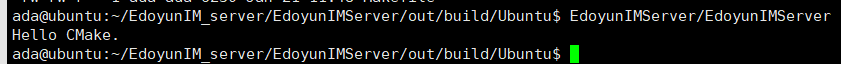
然后执行cmake



然后执行make



可以在EdoyunIMServer目录下面看到可执行程序



执行一下，即可看到项目内容

到此，项目基本的环境配置就完成了

## CMake入门

# CMakeList.txt: EdoyumIMServer 的 CMake 项目，在此处包括源代码并定义

# 项目特定的逻辑。

# //

cmake\_minimum\_required (VERSION 3.8)

project(EdoyunServer)

#这个宏，会影响编译参数

set(CMAKE\_CXX\_FLAGS "**${CMAKE\_CXX\_FLAGS}** -std=c++0x -g -Wall -O0 -Wno-unused-variable -pthread")

#设置库目录

link\_directories(# 针对64位的

**${PROJECT\_SOURCE\_DIR}**/lib

/usr/lib64/mysql/

)

find\_library(MYSQL\_LIB libmysqlclient.so /usr/lib64/mysql/)

if (NOT MYSQL\_LIB)#NOT 只能用大写

message(FATAL\_ERROR "not found mysqlclient.so at lib64")

endif(NOT MYSQL\_LIB)#必须进行匹配，否则会报错

set(chat\_srcs

"EdoyumIMServer.cpp"

"EdoyumIMServer.h")

# 将源代码添加到此项目的可执行文件。

add\_executable (imchatserver **${chat\_srcs}**)

#只是确认库存在或者设置库目录，是没有用的，需要链接进来才能起效果

target\_link\_libraries(imchatserver mysqlclient)

# 将源代码添加到此项目的可执行文件。

add\_executable (imfileserver **${chat\_srcs}**)

#只是确认库存在或者设置库目录，是没有用的，需要链接进来才能起效果

target\_link\_libraries(imfileserver)

# TODO: 如有需要，请添加测试并安装目标。

## 一个服务器的开始

### 后台运行

如何开启一个后台运行程序

* 参数处理

getopt函数

* 信号处理

signal函数

* 子进程

fork函数

* 后台运行

setsid 使得前台无法影响后台

重置stdin stdout stderr到/dev/null 使得后台无法影响前台

* 整体流程

1. 处理信号
2. 分析参数
3. 守护模式
4. 设置日志
5. 主程序
6. 结束退出

### Muduo（木铎）库介绍

木铎的含义：文事奋木铎，武事奋金铎



源码下载地址：

<https://github.com/chenshuo/muduo>

下载木铎库

git clone <https://github.com/chenshuo/muduo.git>

如果Ubuntu提示没有git命令

则使用

sudo apt install git

命令来安装

如果网速不好也可以通过

<https://github.com/chenshuo/muduo/archive/master.zip>

直接下载源码

或者将 <https://github.com/chenshuo/muduo.git>导入到码云上再下载

文档结构

.  
├── base  
│ ├── AsyncLogging.h\*  
│ ├── Atomic.h  
│ ├── BlockingQueue.h  
│ ├── BoundedBlockingQueue.h  
│ ├── BUILD.bazel  
│ ├── Condition.h  
│ ├── copyable.h  
│ ├── CountDownLatch.h  
│ ├── CurrentThread.h  
│ ├── Date.h  
│ ├── Exception.h  
│ ├── FileUtil.h  
│ ├── GzipFile.h  
│ ├── LogFile.h\*  
│ ├── Logging.h\*  
│ ├── LogStream.h\*  
│ ├── Mutex.h  
│ ├── noncopyable.h  
│ ├── ProcessInfo.h  
│ ├── Singleton.h  
│ ├── StringPiece.h  
│ ├── tests  
│ ├── Thread.h  
│ ├── ThreadLocal.h  
│ ├── ThreadLocalSingleton.h  
│ ├── ThreadPool.h  
│ ├── Timestamp.h  
│ ├── TimeZone.h  
│ ├── Types.h  
│ └── WeakCallback.h  
└── net  
 ├── Acceptor.h  
 ├── boilerplate.h  
 ├── Buffer.h  
 ├── BUILD.bazel  
 ├── Callbacks.h  
 ├── Channel.h  
 ├── CMakeLists.txt  
 ├── Connector.h  
 ├── Endian.h  
 ├── EventLoop.h  
 ├── EventLoopThread.h  
 ├── EventLoopThreadPool.h  
 ├── http  
 │ ├── BUILD.bazel  
 │ ├── CMakeLists.txt  
 │ ├── HttpContext.h  
 │ ├── HttpRequest.h  
 │ ├── HttpResponse.h  
 │ ├── HttpServer.h  
 │ └── tests  
 ├── InetAddress.h  
 ├── inspect  
 │ ├── Inspector.h  
 │ ├── PerformanceInspector.h  
 │ ├── ProcessInspector.h  
 │ ├── SystemInspector.h  
 │ └── tests  
 ├── poller  
 │ ├── EPollPoller.h  
 │ └── PollPoller.h  
 ├── Poller.h  
 ├── protobuf  
 │ ├── BufferStream.h  
 │ └── ProtobufCodecLite.h  
 ├── protorpc  
 │ ├── google-inl.h  
 │ ├── README  
 │ ├── RpcChannel.h  
 │ ├── RpcCodec.h  
 │ ├── rpc.proto  
 │ ├── RpcServer.h  
 │ └── rpcservice.proto  
 ├── Socket.h  
 ├── SocketsOps.h  
 ├── TcpClient.h  
 ├── TcpConnection.h  
 ├── TcpServer.h  
 ├── tests  
 ├── Timer.h  
 ├── TimerId.h  
 ├── TimerQueue.h  
 └── ZlibStream.h

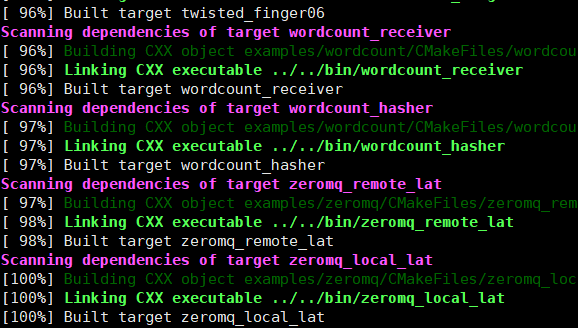
muduo库的编译：

安装boost库

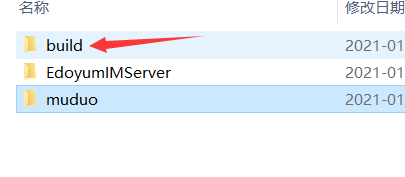
sudo apt install g++ cmake make libboost-dev



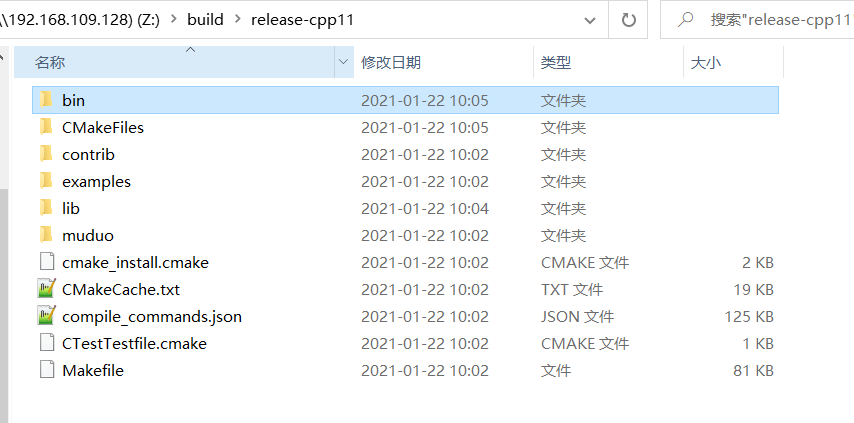
然后运行./build.sh



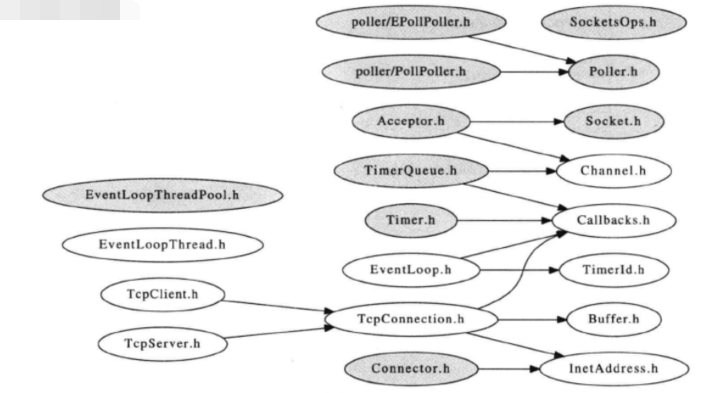
编译完成后，在build目录下



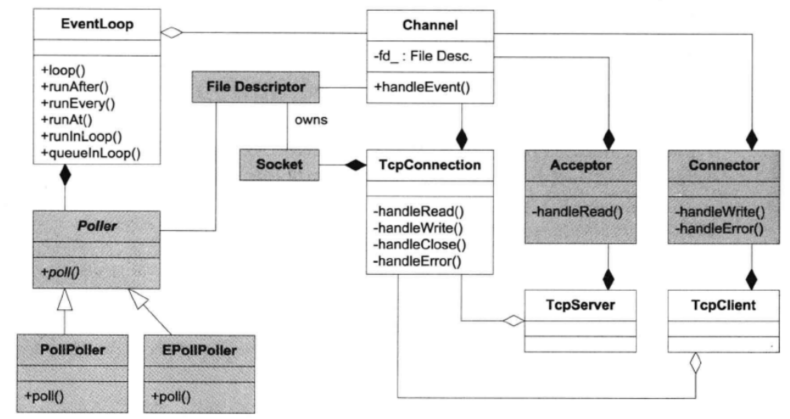
可以找到生成的文件



头文件的包含结构



简化的类图



整个模块采用reactor模式

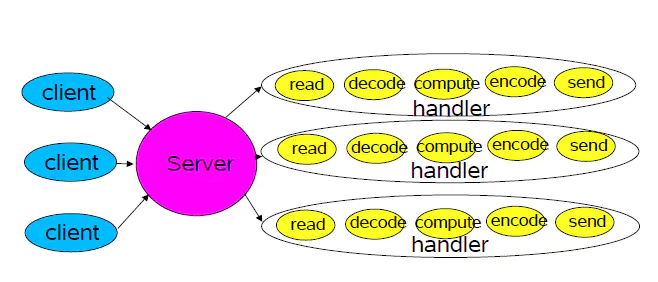
### Reactor模式介绍

起源

一般我们做服务器，常常是基于线程架构的

也就是thread-based architecture

这种模式可以用下图来描述：



这种方式适合初中级程序员

优点在于结构清晰，逻辑明了，可读性强

但是缺点也是非常明显的：

每个客户端的请求，都需要一个线程来处理

如果有一万个用户，就需要一万个线程来处理

有十万个用户，就需要开十万个线程

暂且不说十万个，单单一万个线程，仅仅是切换这些线程

基本CPU就废了

处了这个问题，还有另外一个问题，就是无法控制算力

比如一百个线程，可能某一刻，一百个线程都在闲着，CPU闲到发慌

也可能下一刻所有的线程都需要忙起来，CPU瞬间原地爆炸

而且这种情况完全不受服务器控制

也难以预测，完全由用户的行为来主导

基于上面这种方式的问题

我们有了**事件驱动架构**（event-driven architecture）

这种架构把要使用CPU的内容定义为一个事件

比如网络编程中的客户端接入、数据读取、数据发送、连接关闭

当有客户端接入的事件发生的时候

我们就安排线程来处理接入

处理完了，就将该事件移出线程，等待其他事件的发生

这种方式有很多好处

比如我们可以控制CPU的利用率

还是上面那个例子，一百个用户连上来

如果某一刻一百个人都在做请求

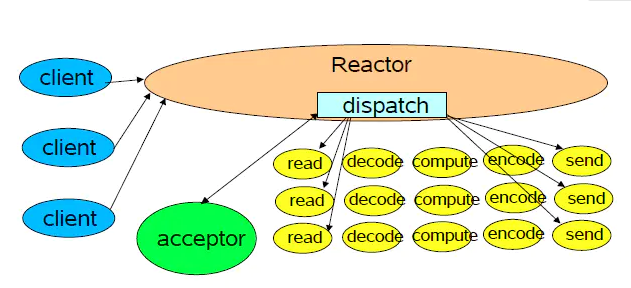
这个时候我们可以指定几个线程来处理这些请求

没有处理到的用户会在一个事件队列中等候处理

这样不会出现一百个用户同时被处理，但是每个都无法很好的处理

基于事件驱动架构的模式就是Reactor模式

这种模式的典型情况如下图



用户来了，reactor（反应堆）会将事件分发给接收、读取、解码、计算、编码、发送各个处理模块

这些模块没有单独的执行权限

全部靠reactor来分派线程，进而执行内容

木铎库就是采用这种方式来处理的

这种方式，除非没有任何事件发生

否则线程就一直都会干活

但是再多事件发生，也只会有指定数量的线程在干活

如果硬件性能差，则把数量减少一些

如果硬件性能好，则把线程数量增加一些

整个服务器的性能峰值就可以控制在手上了

在这种反应堆模式中，几个概念是需要强调的

* Handle（句柄）

具体的事件源，可以是文件描述符、网络套接字等等

* Synchronous Event Demultiplexer同步事件分离器

分离器一般是系统的接口

比如select、poll或者epoll函数

这些东西将程序的状态由事件触发状态切换到事件处理状态

比如select会阻塞，直到某个select关注的handle产生事件

* Event Handler事件处理器

这个元素里面一般包含一个回调函数

当handle上产生事件的时候

这个回调函数则可能被执行

* Concrete Event Handler具体的事件处理器

注意，这个一般是事件处理器的子类

会实现具体的回调，完成业务逻辑

* Initiation Dispatcher初始分发器

提供注册、删除与转发event handler的方法

当同步事件分离器（Synchronous Event Demultiplexer）发现某个handle上有事件的时候

就会通知初始分发器（本概念）来调用事件处理器来处理事件

* 工作流程
* 用户需要在初始分发器中注册具体的事件处理器

说明在什么事件发生的时候，调用本具体的事件处理器

注意，注册的时候，也要事件处理器绑定好对应的句柄

* 注册完成后，初始分发器会开启事件循环，然后使得同步事件分离器来等待事件的发生
* 当某个句柄上产生的事件为就绪的时候

同步事件分离器就会通知初始化分发器

* 初始化分发器会调用事件处理器的回调：

通过事件来定位对应的句柄和句柄回调方法

* 具体的事件处理器会调用器内部关联的回调方法来处理具体的事件

### 使用木铎库创建一个简单的服务器

#include "EdoyumIMServer.h"

#include <signal.h>

#include <fcntl.h>

#include "base/Logging.h"

#include "net/EventLoop.h"

#include "net/EventLoopThread.h"

#include "net/EventLoopThreadPool.h"

#include "net/TcpServer.h"

using namespace std;

void show\_help(const char\* cmd)

{

cout << "found error argument!\r\n";

cout << "Usage:" << std::endl;

cout << cmd << " [-d]" << std::endl;

cout << "\t-d run in daemon mode.\r\n";

}

void signal\_exit(int signum)

{

cout << "signal " << signum << " found, exit ...\r\n";

//TODO:退出的清除

switch (signum)

{

case SIGINT:

case SIGKILL:

case SIGTERM:

case SIGILL:

case SIGSEGV:

case SIGTRAP:

case SIGABRT:

//TODO:

break;

default:

//TODO:

break;

}

exit(signum);

}

void daemon()

{

signal(SIGCHLD, SIG\_IGN);

int pid = fork();

if (pid < 0) {

cout << "fork call error,code is " << pid << " error code is " << errno << std::endl;

exit(-1);

}

if (pid > 0) {//主进程在此结束

exit(0);

}

//这里的代码只可能是子进程了

//这里可以避免父进程所在的会话结束时，把子进程带走

//不让前台影响后台

setsid();

//不让后台影响前台

int fd = open("/dev/null", O\_RDWR, 0);

cout << "invoke success!" << std::endl;

cout << "STDIN\_FILENO is " << STDIN\_FILENO << std::endl;

cout << "STDOUT\_FILENO is " << STDOUT\_FILENO << std::endl;

cout << "STDERR\_FILENO is " << STDERR\_FILENO << std::endl;

cout << "fd is " << fd << std::endl;

if (fd != -1) {

dup2(fd, STDIN\_FILENO);

dup2(fd, STDOUT\_FILENO);

dup2(fd, STDERR\_FILENO);

}

if (fd > STDERR\_FILENO)

close(fd);

}

void onConnection(const muduo::net::TcpConnectionPtr& conn)

{

cout << conn->name() << std::endl;

}

void onMessage(const muduo::net::TcpConnectionPtr& conn,

muduo::net::Buffer\* buf,

muduo::Timestamp time)

{

conn->send(buf);

conn->shutdown();

}

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

{

signal(SIGCHLD, SIG\_DFL);

signal(SIGPIPE, SIG\_IGN);//网络当中，管道操作

signal(SIGINT, signal\_exit);//中断错误

signal(SIGKILL, signal\_exit);

signal(SIGTERM, signal\_exit);//ctrl + c

signal(SIGILL, signal\_exit);//非法指令错误

signal(SIGSEGV, signal\_exit);//段错误

signal(SIGTRAP, signal\_exit);//ctrl + break

signal(SIGABRT, signal\_exit);//abort函数调用触发

cout << "imchatserver is invoking ..." << endl;

int ch = 0;

//sever -a server a

bool is\_daemon = false;

while ((ch = getopt(argc, argv, "d")) != -1) {

cout << "ch = " << ch << std::endl;

cout << "current " << optind << " value:" << argv[optind - 1] << std::endl;

switch (ch)

{

case 'd':

is\_daemon = true;

break;

default:

show\_help(argv[0]);

return -1;

}

}

if (is\_daemon) {

daemon();

}

muduo::net::EventLoop loop;

muduo::net::InetAddress addr(9527);

muduo::net::TcpServer server(&loop, addr, "echo server");

server.setConnectionCallback(onConnection);

server.setMessageCallback(onMessage);

server.start();

loop.loop();

return 0;

}

### 用木铎库实现一个单例服务器

预备知识：

几个关键字=default =delete explicit implicit

=default

=delete

explicit

implicit

decltype

PTHREAD\_ONCE\_INIT

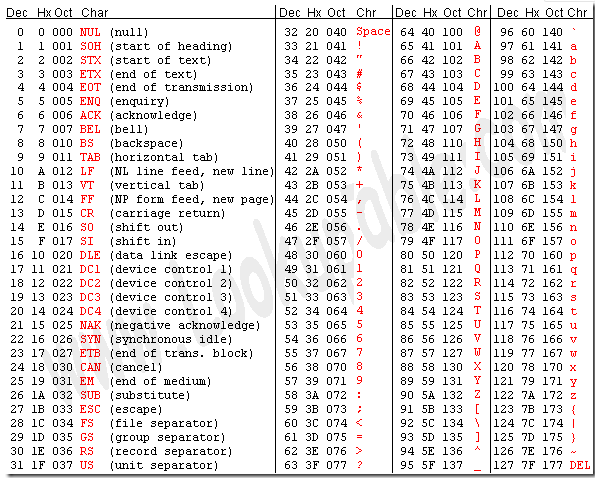
SFINAE Substitution failure is not an error替换失败不是错误

## Linux下数据库的设计与开发

### MySQL数据的基本接口与封装

内容见课程

附件：Ascii码表



## 业务的开发

内容见视频课程