视频会议服务器

项目解读

项目代码结构

源码文件

main.cpp	主程序入口代码
error.cpp	错误信息反馈模块
msg.h	消息定义
netheader.h	消息类型和图片格式定义
unp.h	自定义头文件
unpthread.h	自定义线程
unpthread.cpp	自定义线程
wrapunix.cpp	unix 接口扩展（API 函数封装）
net.cpp	网络模块
room.cpp	房间模块
userdeal.cpp	用户处理模块

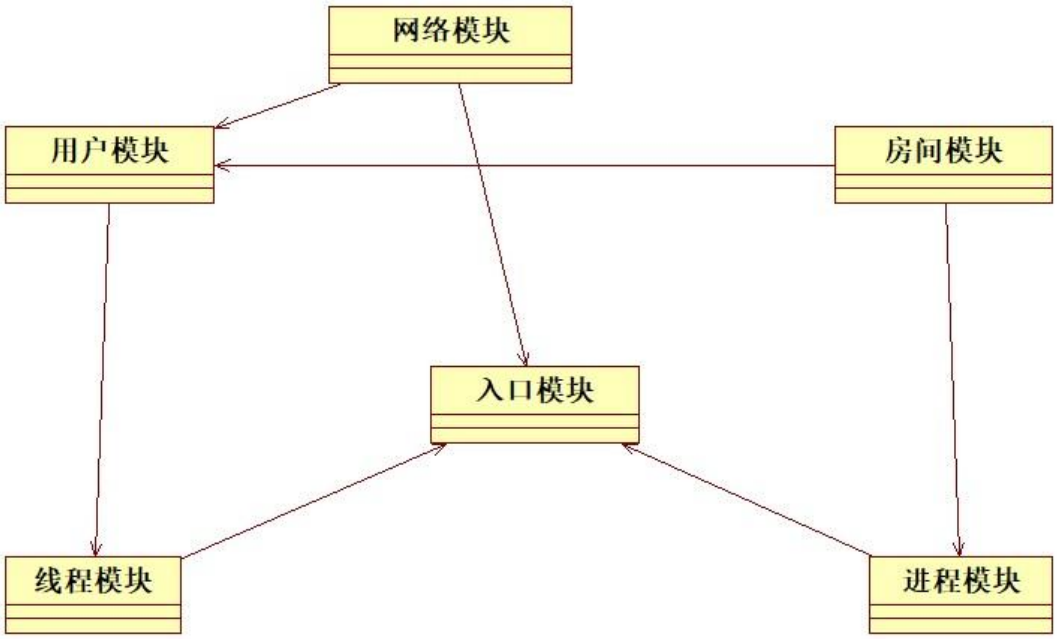
资源文件

Makefile	make 编译文件

项目架构

本项目整体属于典型的 CS 架构（客户端——服务器架构）中的客户端。
服务器端采用主从多进程架构。
进程数量可以依据实际服务器的硬件性能和需求指定进程和线程的数量上限。
整个采用一种自适应的结构，依据产品需求，通过启动参数随时调整。

项目模块关系



项目模块功能

入口模块	初始化其他模块，并且孵化线程和进程
线程模块	处理用户信息
进程模块	创建和维护进程，加载房间模块函数
网络模块	提供网络收发功能
用户模块	维护用户消息处理
房间模块	提供房间处理

项目流程

应用初始化

函数 main→
信号处理注册→
网络模块初始化→
房间模块初始化→
进程初始化→
线程初始化→
网络事件无尽循环

网络数据处理

发生读事件→
转发到子进程→
子进程读取消息数据→
处理数据：
如需应答加入发送队列→
    发送线程提取应答数据→
    应答数据发送
如需转发数据加入发送队列→
    发送线程提取转发数据→
    转发数据发送

项目技术栈

多进程编程

这里的每个房间采用独立进程来处理，一方面保证了各个房间数据的私密性，一方面也加强了各个房间的稳定性。
由于进程天然具有内存隔离的特性，所以各个房间的数据不会意外串访。
另外进程的独立性也使得某个房间崩溃的时候，不会让其他房间立刻也一起崩溃。
如果采用多线程来做，一个线程的崩溃，可能导致整个进程的崩溃，进而使得其他房间也失去服务。

多线程编程

为了提高服务器对网络数据处理的效率，采用了多线程的模式来处理网络数据。
另外将数据的收发和数据的处理也进行了分离。
接收数据的线程并不会处理数据，而是转发给其他线程或者进程进行处理。
这样能够快速进入下一个数据的接收过程。
避免因为长时间得不到处理而导致客户端失去响应。
一般网络编程的接收过程，往往会伴随着处理过程。
可是处理过程的时间损耗无法估计。
一旦损耗过大，就会反过来影响网络的接收过程。
所以这里进行了分离，接收是接收的线程，处理则是另外的线程。
接收效率大大提高，客户端的请求数据吞吐能力也显著增强。

网络编程

我没有使用 epoll 而是使用了 select 做为服务器的网络框架。
这里主要有以下的考量：
1. Epoll 模型更为复杂，开发和维护成本更高。
2. 涉及视频传输的服务器，其瓶颈往往是带宽而不是其他硬件资源。也就是并发量往往不会太大，一般在 200 左右。因为一个用户 500k 的带宽，200 个用户意味着 100M 的带宽。这个带宽成本已经非常高了。
3. 视频会议中一个房间的人数并不会太夸张，往往是几十到上百。更大规模的视频会议，需要的是直播而不是视频会议。因为超过一百规模的会议，不可能所有用户同时进行网络视频，这个是带宽无法承受的。而且这样的会议也会有主讲人。这种场景下，直播会更方便。我的目标是做一个小巧的视频会议系统，而不是大型的直播系统。
综上所述，我选择 select 而非 epoll。
我认为前者更适合我的这个项目。

视频会议服务器

简单的视频会议服务器核心功能其实是提供房间服务。
比如房间的创建和销毁。
用户加入和离开房间。
用户在房间中获得的消息转发服务等等。
视频数据在服务器也只是一个普通的数据包，只是尺寸可能大一点点而已。
所以这里服务器采用了 Linux 系统，这种系统内存占用较小的操作系统。
这样可以腾出更多资源来开启房间进程，提供更高的服务上限，最大可能的提高硬件资源的利用率。

项目问题

如何实现进程池？其作用是什么？

在 Linux 中使用 fork 就可以很容易的创建进程，但是管理这些进程则较为麻烦。
我首先要解决的问题是进程间的通信。
我这里选用了 sendmsg 和 recvmsg 来完成通信。
这种模式的好处在以下几个方面：
一，其具有私密性。使用本地套接字通信，也是可以的。但是其他进程也可以通过此本地 ip 和端口访问这个通信通道。
二，其具有易用性。相较于管道和套接字，这种模式下，接收者可以直接拿到包长度。
基于这些方面的考量，我选择了此模式来完成通信。
完成了通信，那么进程池也就好实现了。
首先是完成进程的通信模块。

利用这个模块就可以控制其他进程了。
然后还是利用这个进程，还可以向其他进程收发数据。
这样本进程就可以控制和传输数据。
然后统一编号，就可以批量处理这些进程，形成进程池操作了。
有了进程池，我们就可以将一些业务丢给这些进程池去完成，把主进程从业务处理的繁杂耗时任务中解放出来，做一些更重要的调度和控制工作。

如何实现线程池？其作用是什么？

线程池的实现要比进程池更加容易，因为线程池中的各个线程之间，内存是互通的。
这样线程控制和数据传输就更为容易。
线程池一般用于执行同质化的多个任务。
比如 select 的事件处理。
处理过程其实是同质化的（同质化：具有同样的性质，或者功能可以并列。比如都是网络处理。同质化不意味着完全一样：同样是网络处理，A 线程是用户登录消息，B 线程可能是用户文字消息）。
其作用是尽可能的提高硬件的利用效率。
包括提高网络利用效率、处理器利用效率等等。
比如单个任务可能只会使用到处理器 5%的性能，而这些任务的触发时机又很随机。
那么创建一个 20 线程的线程池，来做任务处理。
高峰的时候就可以同时处理 20 个任务，将处理器的性能利用到 100%。

为什么网络通信使用 select 而不是 epoll？

我没有使用 epoll 而是使用了 select 做为服务器的网络框架。
这里主要有以下的考量：
4. Epoll 模型更为复杂，开发和维护成本更高。
5. 涉及视频传输的服务器，其瓶颈往往是带宽而不是其他硬件资源。也就是并发量往往不会太大，一般在 200 左右。因为一个用户 500k 的带宽，200 个用户意味着 100M 的带宽。这个带宽成本已经非常高了。
6. 视频会议中一个房间的人数并不会太夸张，往往是几十到上百。更大规模的视频会议，需要的是直播而不是视频会议。因为超过一百规模的会议，不可能所有用户同时进行网络视频，这个是带宽无法承受的。而且这样的会议也会有主讲人。这种场景下，直播会更方便。我的目标是做一个小巧的视频会议系统，而不是大型的直播系统。
综上所述，我选择 select 而非 epoll。
我认为前者更适合我的这个项目。

进程间通信有哪些方法？你的项目中选用了哪一种？为什么？

我这里选用了 sendmsg 和 recvmsg 来完成进程间通信。
这种模式的好处在以下几个方面：
一，其具有私密性。使用本地套接字通信，也是可以的。但是其他进程也可以通过此本地 ip 和端口访问这个通信通道。
二，其具有易用性。相较于管道和套接字，这种模式下，接收者可以直接拿到包长度。

基于这些方面的考量，我选择了此模式来完成通信。

用户加入房间的过程能简单描述一下吗？

服务器启动，我们会依据参数创建好房间
然后用户连接服务器。
然后用户选择创建房间，这样我们会将从房间池里面选择一个空房间发给用户。
如果没有，则创建失败。

然后其他用户上线。
这些用户填入房间号，我们就会将该用户 ID 分配给指定的房间。
接着向房间内的所有用户发出消息，通知用户加入。

经过上面的步骤之后，用户就加入了房间。

请描述一下在房间中传输视频信息和文字信息的过程

客户端在产生视频帧数据后，会封装成消息发给服务器。
服务器会找到该客户端所在的房间，如果找不到则丢弃，并返回错误提示。
如果找到，则将该数据转发给其他所有的用户。
其他用户拿到帧数据，解析成功后，再将帧画面显示在界面上。
文字信息也是同样的过程。

Linux 下信号是什么？你是如何处理的？

信号是操作系统响应某些条件而产生的一个事件。
信号的定义是在 signal.h 头文件里面的。
信号本质上是软件中断，是对中断在软件上的模拟，是异步通信，也是一种跨进程通信。

我在项目中主要处理了 SIGCHLD 和 SIGPIPE 信号
前者表示子进程结束，后者表示管道错误。
这些都是在多进程编程的时候需要处理的信号。
我采用 sigaction 函数来注册信号处理函数。
对于管道错误信号，我直接进行忽略处理。
而对于进程结束信号，我实现了 sig_chld 函数来进行处理。
在该函数中，我调用了 waitpid 函数来回收子进程资源。
同时设置了 WNOHANG 标志，来进行非阻塞处理。
这些就是我如何处理信号的。

如何自定义信号处理过程？

自定义信号处理过程分为两步：
第一步是注册信号处理函数。
这个需要使用到 sigaction 函数。
该函数的第一个参数指定要自己定义处理的信号值。第二个参数指定一个 sigaction 结构体。
该结构体包含了一个函数指针。我们可以把这个指针指向我们自定义的处理函数。
第二步就是实现这个处理函数。
这个函数有两种形式，一种是单参数形式，参数就是信号的值。
一种则复杂一点，还有一个 siginfo_t 参数和 void*指针，可以从里面解析出更多的值。
这种一般用于段错误、除零错误等特殊情况。
经过上面两步，我们就可以完整的实现自定义信号的处理过程。
本项目中，子进程结束信号就需要我们自己来处理。
在自定义处理中，我们加入了一些日志信息，可以很好的监视子进程的情况。
这个功能在多进程开发过程中非常重要，很多 bug 需要通过这样的方式来定义和发现。

Linux 线程间传输数据有哪些方法？你的项目选择的是哪一种？为什么？

线程间传输数据的方法比较多，比较常用的有直接内存访问，本地网络传输，管道，文件等等方法。
本项目我选择的直接内存访问。
这种方法的缺点是在多线程环境下，容易出现访问冲突。
为了解决这个问题，我使用了 Linux 的线程锁。
访问之前先上锁，然后再进行读写。完成读写操作后，再进行解锁。
这样多个线程同时访问该数据的时候，只会有一个线程能拿到访问权。
使用这种方式，主要是基于两点考虑。
第一是这种方式需要的内存冗余较小。数据基本无需反复复制或者拷贝。
建立一个全局可访问的队列，有什么数据直接放入队列即可。
用完直接从队列删除。
如果是网络或者管道模式，则需要建立发送或者接收缓冲区。

同一个数据，可能在传输过程中多次被复制或者缓冲。
第二是这种方式速度快成本低。
比如文件方式，其速度就远远比不上内存直接访问。
第三是这种方式简单易维护。
简单就意味着错误的概率小，容易上手和理解。
这样就容易维护。有问题，无论是分析问题，定位问题还是解决问题都要容易很多。
其他方式这方面则有很大的劣势。
综上，我选择了带互斥锁的直接内存访问的方式来进行线程间数据传输。

你的项目中遇到过哪些问题？你是如何解决的？

最大的问题是跨进程调试问题。
尤其是开发阶段，子进程会出现各种问题，然后突然闪退。
断点都不好下。
用 gdb 调试的时候，一开始也不知道在哪里下断点。
单步 gdb 操作起来非常费时费力。
而且不一定那一次就能触发 bug。
所以后面是结合日志分析的办法来定位问题的。
具体做法如下：
首先实现一个日志模块，利用宏来降低日志成本。
然后在相关流程路径上的各个函数入口和出口添加日志。
当发现子进程异常退出的时候，先分析代码，找到一个大致的范围。
比如日志到了哪个模块的哪个函数消失的。
然后将该函数的参数、返回值、局部相关的变量都打印出来。
并且添加日志，看看具体是到哪一行日志消失的。
然后拿着这些参数的值，结合代码一步步分析哪里可能出错。
必要的时候，还需要查看一些其他被调用函数的返回值或者参数的值。
经过上面的操作之后，问题都能能够定位出来，并且找到崩溃原因。
但是解决这些问题还需要再分析一下。
因为有的可能单纯是代码输入错误，有的可能是流程设计就有问题——一些特殊情况没考虑到。
完成之后，还要再复测一下，确保问题真正解决了。
最后去掉无关的日志，再次测试一下，进行验证。
其他的，我认为问题不大。因为整个项目难度并不算太大。