### **服务器代码（server.cpp）**

#include "kcp\_server.h"

#include <memory>

#include <mutex>

#include <sstream>

#include <unordered\_set>

// 基于KcpServer封装业务class ChatServer : public KcpServer {

public:

// 使用基类的构造函数，方便创建ChatServer对象时传入必要参数

using KcpServer::KcpServer;

// 接收客户端发送过来的消息

// session 是客户端会话的智能指针，用于标识客户端

// msg 是客户端发送的消息内容

virtual void HandleMessage(const KcpSession::ptr &session,

const std::string &msg) override {

// 打印接收到的消息，方便调试查看

TRACE(msg);

// 创建一个字符串流，用于构造要广播给所有客户端的消息

std::stringstream ss;

// 拼接消息，包含客户端的地址和客户端发送的原始消息

ss << "user from [" << session->GetAddrToString() << "] : " << msg;

// 打印拼接好的消息

TRACE(ss.str());

// 调用通知函数，将拼接好的消息广播给所有客户端

Notify(ss.str());

}

// 处理新客户端连接的函数

// session 是新连接客户端的会话智能指针

void HandleConnection(const KcpSession::ptr &session) {

{

// 创建一个互斥锁，确保在多线程环境下对用户集合的操作是线程安全的

// 因为可能有多个客户端同时连接，需要避免数据竞争

Lock lock(mtx\_);

// 将新连接的客户端会话添加到用户集合中

users\_.insert(session);

}

// 创建一个字符串流，用于构造欢迎新用户的消息

std::stringstream ss;

ss << "user from [" << session->GetAddrToString()

<< "] join the ChatRoom!";

// 调用通知函数，将欢迎消息广播给所有客户端

Notify(ss.str());

}

// 处理客户端断开连接的函数

// session 是断开连接客户端的会话智能指针

void HandleClose(const KcpSession::ptr &session) {

// 打印客户端断开连接的信息，方便调试查看

TRACE("close ", session->GetAddrToString());

{

// 创建一个互斥锁，确保在多线程环境下对用户集合的操作是线程安全的

// 因为可能有多个客户端同时断开连接，需要避免数据竞争

Lock lock(mtx\_);

// 从用户集合中移除断开连接的客户端会话

users\_.erase(session);

}

// 创建一个字符串流，用于构造用户离开的消息

std::stringstream ss;

ss << "user from [" << session->GetAddrToString()

<< "] left the ChatRoom!";

// 调用通知函数，将用户离开的消息广播给所有客户端

Notify(ss.str());

}

// 通知所有客户端的函数

// str 是要广播的消息内容

void Notify(const std::string &str) {

// 创建一个互斥锁，确保在多线程环境下对用户集合的遍历和消息发送操作是线程安全的

// 因为可能在遍历过程中有客户端连接或断开，需要避免数据竞争

Lock lock(mtx\_);

// 遍历用户集合中的所有客户端会话

for (auto &user : users\_)

// 向每个客户端会话发送广播消息

user->Send(str.data(), str.size());

}

public:

// 定义锁类型，使用 std::unique\_lock 来管理互斥锁

using Lock = std::unique\_lock<std::mutex>;

private:

// 存储所有客户端会话的集合，使用 std::unordered\_set 方便快速查找和插入

std::unordered\_set<KcpSession::ptr> users\_;

// 互斥锁，用于保护对用户集合的并发访问

std::mutex mtx\_;};

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

// 检查命令行参数是否正确，需要传入服务端监听的 IP 地址和端口号

if (argc != 3) {

// 打印错误提示信息

TRACE("args error please input [ip] [port]");

// 程序异常退出

exit(-1);

}

// 获取命令行传入的服务端监听的 IP 地址

std::string ip = argv[1];

// 获取命令行传入的服务端监听的端口号，并转换为无符号 16 位整数

uint16\_t port = atoi(argv[2]);

// 创建 Kcp 选项对象，用于配置 KCP 连接的参数

KcpOpt opt;

// 设置会话标识，这里设置为 0

opt.conv = 0;

// 设置为服务端模式

opt.is\_server = true;

// 设置保活心跳包的超时时间，单位为毫秒，这里设置为 5000 毫秒（即 5 秒）

opt.keep\_alive\_timeout = 5000;

// 创建 ChatServer 对象，传入 Kcp 选项、IP 地址和端口号

ChatServer server(opt, ip, port);

// 启动服务端，开始监听客户端连接和接收消息

server.Run();

return 0;}

### **客户端代码（client.cpp）**

#include "kcp\_client.h"

#include "trace.h"

#include <thread>

// 基于KcpClient封装业务class ChatClient : KcpClient {

public:

// 使用基类的构造函数，方便创建ChatClient对象时传入必要参数

using KcpClient::KcpClient;

// 启动客户端的函数

void Start() {

// 创建一个新线程，用于处理客户端的运行逻辑

std::thread t([this]() {

while (true) {

// 线程休眠 10 微秒，避免 CPU 占用过高

usleep(10);

// 调用基类的 Run 函数，处理客户端的消息接收和发送等逻辑

if (!Run()) {

// 如果 Run 函数返回 false，表示出现错误

// 打印错误信息

TRACE("error ouccur");

// 跳出循环，结束线程

break;

}

}

});

while (true) {

// 定义一个字符串变量，用于存储用户输入的消息

std::string msg;

// 从标准输入读取一行用户输入的消息

std::getline(std::cin, msg);

// 调用基类的 Send 函数，将用户输入的消息发送给服务端

Send(msg.data(), msg.length());

}

// 等待处理客户端运行逻辑的线程退出

t.join();

}

// 处理服务端发送过来的消息的函数

// msg 是服务端发送的消息内容

virtual void HandleMessage(const std::string &msg) override {

// 将服务端发送的消息打印到控制台

std::cout << msg << std::endl;

}

// 处理客户端断开连接的函数

virtual void HandleClose() override {

// 打印客户端断开连接的信息

std::cout << "close kcp connection!" << std::endl;

// 程序异常退出

exit(-1);

}};

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

// 检查命令行参数是否正确，需要传入服务端的 IP 地址和端口号

if (argc != 3) {

// 打印错误提示信息

TRACE("args error please input [ip] [port]");

// 程序异常退出

exit(-1);

}

// 获取命令行传入的服务端的 IP 地址

std::string ip = argv[1];

// 获取命令行传入的服务端的端口号，并转换为无符号 16 位整数

uint16\_t port = atoi(argv[2]);

// 初始化随机数种子，确保每次运行程序时生成的随机数不同

srand(time(NULL));

// 创建 Kcp 选项对象，用于配置 KCP 连接的参数

KcpOpt opt;

// 生成一个随机的会话标识，用于唯一标识客户端会话

opt.conv = rand() \* rand();

// 设置为客户端模式

opt.is\_server = false;

// 设置保活心跳包的间隔时间，单位为毫秒，这里设置为 1000 毫秒（即 1 秒）

opt.keep\_alive\_timeout = 1000;

// 打印生成的会话标识，方便调试查看

TRACE("conv = ", opt.conv);

// 创建 ChatClient 对象，传入服务端的 IP 地址、端口号和 Kcp 选项

ChatClient client(ip, port, opt);

// 启动客户端

client.Start();

return 0;}

**CMAKE**

CMAKE\_MINIMUM\_REQUIRED(VERSION 2.6)

project(kcp LANGUAGES C)

include(CTest)

include(GNUInstallDirs)

SET(CMAKE\_BUILD\_TYPE "Debug")

SET(CMAKE\_CXX\_FLAGS\_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g2 -ggdb -std=c++11")

SET(CMAKE\_CXX\_FLAGS\_RELEASE "$ENV{CXXFLAGS} -O0 -Wall -std=c++11")

add\_library(kcp STATIC ikcp.c)

install(FILES ikcp.h DESTINATION ${CMAKE\_INSTALL\_INCLUDEDIR})

install(TARGETS kcp

    EXPORT kcp-targets

    ARCHIVE DESTINATION ${CMAKE\_INSTALL\_LIBDIR}

    INCLUDES DESTINATION ${CMAKE\_INSTALL\_INCLUDEDIR}

)

install(EXPORT kcp-targets

    FILE kcp-config.cmake

    NAMESPACE kcp::

    DESTINATION ${CMAKE\_INSTALL\_LIBDIR}/cmake/kcp

)

if (BUILD\_TESTING)

    enable\_language(CXX)

    add\_executable(client client.c ikcp.c delay.c)

    target\_link\_libraries(client pthread)

    add\_executable(server server.c ikcp.c)

    add\_executable(test\_kcp test.cpp ikcp.c)

    add\_executable(chat\_server chat\_server.cc kcp\_server.cc kcp\_session.cc udp\_socket.cc ikcp.c)

    target\_link\_libraries(chat\_server pthread)

    add\_executable(chat\_client chat\_client.cc kcp\_client.cc kcp\_session.cc udp\_socket.cc ikcp.c)

    target\_link\_libraries(chat\_client pthread)

    if(MSVC AND NOT (MSVC\_VERSION LESS 1900))

        target\_compile\_options(kcp\_test PRIVATE /utf-8)

    endif()

endif ()

### **编译和运行步骤**

#### **编译步骤**

bash

# 进入项目的 day07 目录cd kcp/day07

# 创建一个 build 目录用于存放编译生成的文件mkdir buildcd build

# 使用 CMake 生成 Makefile

cmake ..

# 编译项目make

#### **运行步骤**

* ****服务端运行****：

./chat\_server 0.0.0.0 10001

这里 0.0.0.0 表示监听所有可用的网络接口，10001 是服务端监听的端口号。

* ****客户端运行****：

./chat\_client 127.0.0.1 10001

这里 127.0.0.1 是服务端的 IP 地址，10001 是服务端监听的端口号，你可以根据实际情况修改服务端的 IP 地址。

### **可重写的接口**

#### **服务器端代码（server.cpp）**

在服务器端代码中，基于KcpServer类封装了SimpleServer类，以下是可以重写的接口：

1. **HandleMessage**方法****
   1. ****作用****：当服务器接收到客户端发送的消息时，会调用此方法进行处理。你可以根据具体业务需求，对收到的消息进行不同的处理，例如消息解析、消息转发等。
   2. ****代码示例****：

virtual void HandleMessage(const KcpSession::ptr &session, const std::string &msg) override {

std::cout << "Received from " << session->GetAddrToString() << ": " << msg << std::endl;

std::string response = "Server received: " + msg;

session->Send(response.data(), response.size());}

1. **HandleConnection**方法****
   1. ****作用****：当有新的客户端连接到服务器时，会调用此方法。你可以在此方法中进行一些连接建立后的初始化操作，例如记录连接信息、发送欢迎消息等。
   2. ****代码示例****：

void HandleConnection(const KcpSession::ptr &session) {

std::cout << "New connection from " << session->GetAddrToString() << std::endl;}

1. **HandleClose**方法****
   1. ****作用****：当客户端与服务器的连接关闭时，会调用此方法。你可以在此方法中进行一些连接关闭后的清理操作，例如释放资源、记录日志等。
   2. ****代码示例****：

void HandleClose(const KcpSession::ptr &session) {

std::cout << "Connection closed from " << session->GetAddrToString() << std::endl;}

#### **客户端代码（client.cpp）**

在客户端代码中，基于KcpClient类封装了SimpleClient类，以下是可以重写的接口：

1. **HandleMessage**方法****
   1. ****作用****：当客户端接收到服务器发送的消息时，会调用此方法进行处理。你可以根据具体业务需求，对收到的消息进行不同的处理，例如消息显示、消息解析等。
   2. ****代码示例****：

virtual void HandleMessage(const std::string &msg) override {

std::cout << "Received from server: " << msg << std::endl;}

1. **HandleClose**方法****
   1. ****作用****：当客户端与服务器的连接关闭时，会调用此方法。你可以在此方法中进行一些连接关闭后的清理操作，例如释放资源、记录日志等。
   2. ****代码示例****：

virtual void HandleClose() override {

std::cout << "close kcp connection!" << std::endl;}