目录

- 目录
  - 网络编程的基本流程
    - 终端节点的创建
      - 客户端之终端节点的创建
      - 服务端之终端节点的创建
    - 创建socket
      - 服务端与客户端之创建socket
      - 服务端之接收链接请求的socket（对应accept）
    - 绑定acceptor
    - 可以把端口绑定和acceptor的传建合起来写
    - 链接指定的端点
    - 服务器接收来连接
      - 创建acceptor的几种方法
    - 关于buffer
      - 字符串的buffer
      - 数组的buffer
      - 流式的buffer
      - 容器的buffer
      - 指向缓冲区的指针的buffer
  - 同步读写
    - 同步写 write_some
    - 同步写 send
    - 同步写 write
    - 同步读 read_some
    - 同步读 receive
    - 同步读 read
    - 读取直到指定字符
  - 同步读写的客户端与服务端
    - 客户端的设计
    - 服务端设计
      - session函数
      - server函数
      - 完整服务端代码
    - 同步读写的优劣
  - asio异步读写操作及注意事项
    - 异步写操作
    - 异步读操作
    - 总结 完整代码
      - Session.h头文件
      - Session.cpp
  - asio官方案例存在的隐患
    - Session类
    - Server类

        - Server实现
    - 隐患
    - 总结
    - 完整代码
        - 头文件
        - 实现
- 使用伪闭包实现连接的安全回收
    - 智能指针管理Session
    - Session的uuid
    - 隐患1
    - 如何实现伪闭包
    - 完整的代码
        - 实现1
        - 头文件1
- 封装服务器发送队列
    - 数据节点设计
    - 封装发送接口
- 修改读回调
    - 该节总结
    - 封装后完整代码
        - msgNode
        - CSession
        - CServer
- 处理网络粘包问题
    - 什么是粘包
    - 粘包原因
    - 处理粘包
    - 完善消息节点
    - CSession类完善
    - 完善接收逻辑
    - 服务端完整代码
        - CSession头文件
        - CSession 实现
        - CServer头文件
        - CServer实现
        - MsgNode
    - 客户端修改
    - 粘包测试
    - 目前服务端通信流程图
    - io_context
- 字节序处理和发送队列控制
    - 字节序问题
    - 如何区分本机字节序
    - 服务器使用网络字节序
    - 消息队列控制
- protobuf配置和使用

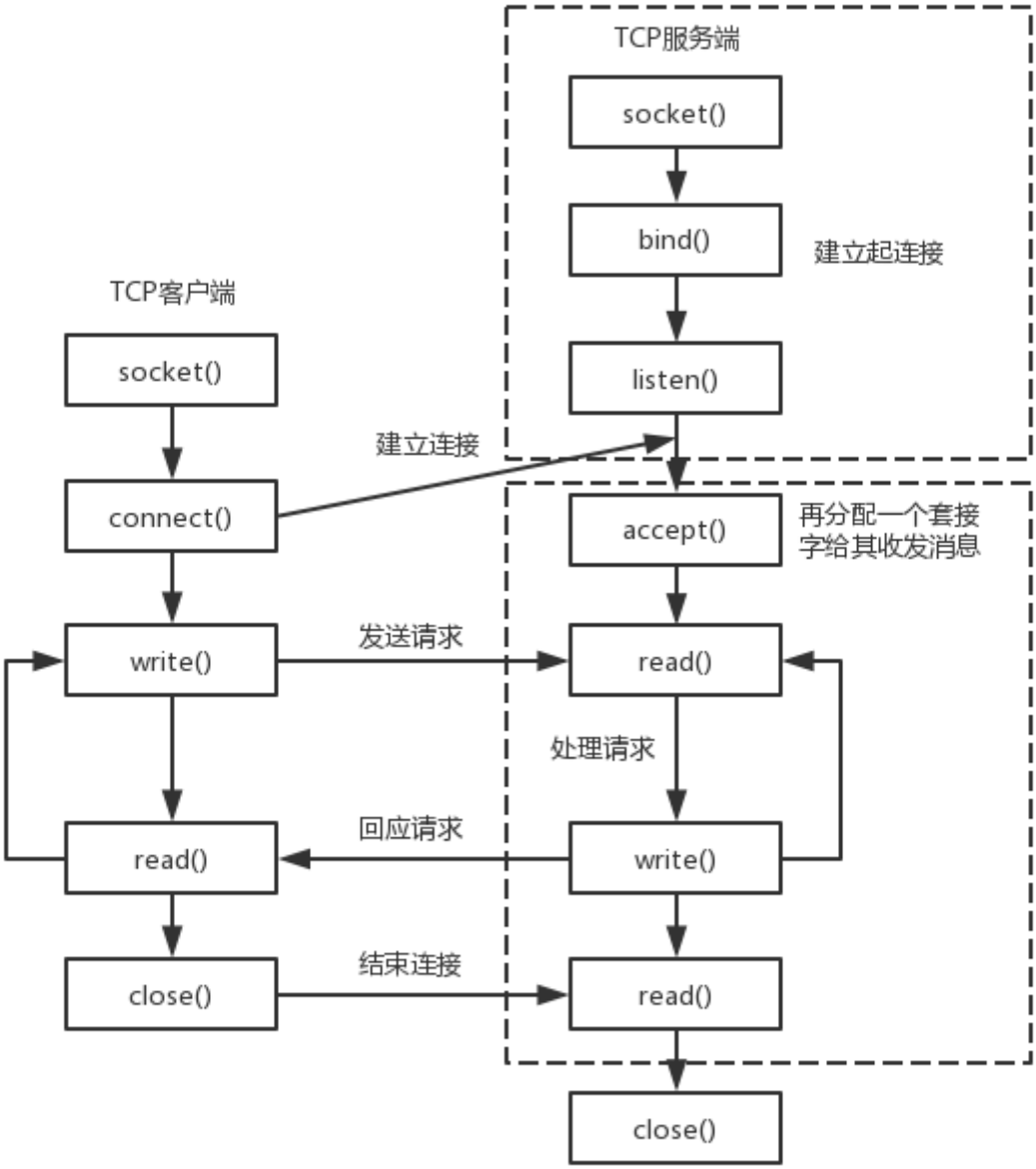
        - portobuf简介
        - 生成pb文件
        - 在网络编程中的应用
    - jsoncpp的使用与配置
        - 简介
        - 配置参考我的csdn收藏
        - 测试
        - 网络编程中的应用
        - 新版JSON库 nlohmann/json
    - asio粘包处理的简单方式
        - 简单方式
        - 获取头部数据
        - 获取消息体
    - 服务器逻辑层设计和消息完善
        - 简介1
        - 服务器架构设计
        - 消息头完善
            - 我们将上述结构定义在MsgNode.h中
            - 实现MsgNode
        - Session类改写
            - 完整的代码2
                - Session.h
                - Session.cpp
    - 单例模式实现逻辑层设计
        - 单例模板类
        - LogicSystem单例类
            - LogicSystem 完整代码
                - 头文件
                - 实现
    - 服务器优雅退出
        - 退出方式1：开辟线程，让服务器运行在线程中并接受退出信号退出
        - 退出方式2：使用asio底层异步等待函数
        - 总结
    - asio多线程模型IOServicePool
        - 简介
        - 单线程和多线程对比
        - IOServicePool实现
            - IOServicePool的声明:
            - 实现
        - 优雅退出
    - asio多线程模式IOThreadPool
        - 结构图
        - 先实现IOThreadPool
            - IOThreadPool 头文件
            - IOThreadPool 实现
                - iocp的流程是这样的

        - epoll流程是这样的
    - 隐患
    - 利用strand改进
    - CSession代码
    - 性能对比
    - 取舍
- boost::asio协程实现并发服务器
    - 简介
    - 协程案例
    - 完整并发服务器
        - AsilOServicePool
            - AsioIOServicePool头文件
            - AsioIOServicePool实现
        - cosnt.h
        - CServer
            - CServer头文件
            - CServer实现
        - CSession
            - CSession.h
            - CSession实现
        - LogicSystem
            - LogicSystem头文件
            - LogicSystem实现
        - MsgNode
            - MsgNode头文件
            - MsgNode 实现
- 使用asio实现http服务器
    - 简介
    - Http包头信息
        - HTTP请求头
        - HTTP响应头
- 使用beast网络库实现http服务器
    - 简介
    - 连接类
    - 完整代码
- beast网络库实现websocket服务器
    - 简介
    - 构造websocket
    - 开发的websocket代码
        - Connection.h
        - Connection.cpp
        - ConnectionMgr.h
        - ConnectionMgr.cpp
        - WebSocketServer.h
        - WebSocketServer.cpp
        - main.cpp

- 总结

网络编程的基本流程

- 网络编程的基本流程对于服务端是这样的
- 服务端 1）socket——创建socket对象。 2）bind——绑定本机ip+port。 3）listen——监听来电，若在监听到来电，则建立起连接 4）accept——再创建一个socket对象给其收发消息。原因是现实中服务端都是面对多个客户端，那么为了区分各个客户端，则每个客户端都需再分配一个socket对象进行收发消息。 5）read、write——就是收发消息了。
- 对于客户端是这样的
- 客户端 1）socket——创建socket对象。 2）connect——根据服务端ip+port，发起连接请求。 3）write、read——建立连接后，就可发收消息了。
- 图示如下



- 了解了解 Reactor模式以及proactor模式

终端节点的创建

- 所谓终端节点就是用来通信的端对端的节点，可以通过ip地址和端口构造，其它节点可以连接这个终端节点做通信。

客户端之终端节点的创建

- 如果我们是客户端，我们可以通过对端的ip和端口构造一个endpoint，用这个endpoint和其通信。

```
#include "endpoint.h"
#include <boost/asio.hpp>
#include <iostream>
```

```
int client_end_point() {
 std::string raw_ip_address = "127.4.8.1";
 unsigned short port_num = 8888;
    // 用来进行错误处理
 boost::system::error_code ec;
    // from_string函数是用来字符串IP地址形式转换为网络地址的表示形式
 boost::asio::ip::address ip_address =
boost::asio::ip::address::from_string(raw_ip_address, ec);
 if (ec.value() != 0) {
  std::cout << "Failed to parse the IP address. Error code = " << ec.value() <<
".Message is" << ec.message();
  return ec.value();
 }
 boost::asio::ip::tcp::endpoint ep(ip_address, port_num);
}
```

服务端之终端节点的创建

```
int server_end_point() {
 unsigned short port_num = 8888;
 // 表示可以接收任意IPV4地址
 boost::asio::ip::address ip_address = boost::asio::ip::address_v4::any();
 boost::asio::ip::tcp::endpoint ep(ip_address, port_num);
 return 0;
}
```

创建socket

- 创建socket分为4步，创建上下文iocontext，选择协议，生成socket，打开socket。

服务端与客户端之创建socket

```
int create_tcp_socket() {
 // 创建上下文io_context (旧版本为io_service)
 boost::asio::io_context ioc;
 // 选择协议
 boost::asio::ip::tcp protocol = boost::asio::ip::tcp::v4();
 // 创建socket,ioc来管理套接字的创建和销毁
 boost::asio::ip::tcp::socket socket(ioc);
 // 用来处理错误信息
 boost::system::error_code ec;
 // 打开socket
 socket.open(protocol, ec);
 if (ec.value() != 0) {
  // 打开失败
  std::cout
   << "Failed to open the socket! Error code = "
   << ec.value() << ". Message: " << ec.message();
```

```
  return ec.value();
 }
 return 0;
}
```

服务端之接收链接请求的socket（对应accept）

```
int create_acceptor_socket() {
 //创建上下文io_context
 boost::asio::io_context ioc;
 // 选择协议
 boost::asio::ip::tcp protocol = boost::asio::ip::tcp::v4();
 // 创建socket
 boost::asio::ip::tcp::acceptor acceptor(ioc);
 // 用来处理错误信息
 boost::system::error_code ec;
 // 打开socket(打开监听状态)
 acceptor.open(protocol, ec);

 if (ec.value() != 0) {
  // 打开失败
  std::cout
   << "Failed to open the socket! Error code = "
   << ec.value() << ". Message: " << ec.message();
  return ec.value();
 }
 return 0;
}
```

绑定acceptor

- 对于acceptor类型的socket，服务器要将其绑定到指定的断点,所有连接这个端点的连接都可以被接收到。

```
int bind_acceptor_socket() {
 unsigned short port_num = 8888;
 boost::asio::ip::tcp::endpoint ep(boost::asio::ip::address_v4::any(),
  port_num);
 boost::asio::io_context ioc;
 boost::asio::ip::tcp::acceptor acceptor(ioc, ep.protocol());
 boost::system::error_code ec;
    //将端点绑定道acceptor这个socket上，与这个端点链接的端点都能被接受到
 acceptor.bind(ep, ec);
 if (ec.value() != 0) {
  //打开失败
  std::cout << "Failed to bind the acceptor socket."
   << "Error code = " << ec.value() << ". Message: "
   << ec.message();
```

```
    return ec.value();
   }
   return 0;
  }
```

可以把端口绑定和acceptor的传建合起来写

```
// 新版本的写法,默认绑定了8888的端口
boost::asio::ip::tcp::acceptor acceptor(ioc,boost::asio::ip::tcp::v4(),8888);
```

链接指定的端点

- 作为客户端可以连接服务器指定的端点进行连接

```
//对应客户端发送链接请求
int connect_to_end() {
 std::string raw_ip_address = "192.168.168.124";
 unsigned short port_num = 8888;
 try {
  boost::asio::ip::tcp::endpoint
ep(boost::asio::ip::address::from_string(raw_ip_address),
   port_num);
  boost::asio::io_context ioc;
  boost::asio::ip::tcp::socket sock(ioc, ep.protocol());
  sock.connect(ep);
 }
 catch (boost::system::system_error& e) {
  std::cout << "Error occured! Error code = " << e.code()
   << ". Message: " << e.what();
  return e.code().value();
 }
 return 0;
}
```

服务器接收来连接

- 当有客户端连接时，服务器需要接收连接

```
int accept_new_connection() {
 //指定连接队列的大小(用来存取未被及时处理的链接)
 const int BACKLOG_SIZE = 30;
 //端口
 unsigned short port_num = 8888;
 // 创建端点
 boost::asio::ip::tcp::endpoint ep(boost::asio::ip::address_v4::any(),
  port_num);
 boost::asio::io_context ioc;
```

```
 try {
        /*新版连接方式
        //boost::asio::ip::tcp::acceptor acceptor(io_context,
boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), port_num));
        //这个可写可不写
        acceptor.listen();
        */
  // 创建指定类型的连接器
  boost::asio::ip::tcp::acceptor acceptor(ioc, ep.protocol());
  //绑定端点
  acceptor.bind(ep);
  //监听事件
  acceptor.listen(BACKLOG_SIZE);
  // 创建一个活跃的sock 用来连接
  boost::asio::ip::tcp::socket sock(ioc);
  // 连接
  acceptor.accept(sock);
 }
 catch (boost::system::system_error& e) {
  std::cout << "Error occured! Error code = " << e.code()
   << ". Message: " << e.what();
  return e.code().value();
 }
}
```

- 在早期版本的 Boost.Asio 中，在使用 ip::tcp::acceptor 对象之前，通常需要调用 bind 函数将 acceptor 绑定到特定的地址和端口上，并调用 listen 函数开始监听连接请求。
- 但在较新的版本中，Boost.Asio 已经做了一些改进，使得在创建 ip::tcp::acceptor 对象时可以通过构造函数直接指定绑定地址和端口，并且开始监听连接请求。这样可以简化代码，并且提供了更方便的接口。

创建acceptor的几种方法

当使用 Boost.Asio 创建 TCP 服务器时，通常有以下几种方法来创建 acceptor 对象：

使用端点对象创建 acceptor：

这是最常见的方法。你首先创建一个 TCP 端点对象（boost::asio::ip::tcp::endpoint），指定要监听的特定地址和端口。然后，使用这个端点对象来创建 acceptor 对象。

```
boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::address_v4::any(), 8888);
boost::asio::io_context io_context;
boost::asio::ip::tcp::acceptor acceptor(io_context, endpoint);
```

使用协议对象创建 acceptor：

- 你也可以直接使用 TCP 协议对象（boost::asio::ip::tcp::v4() 或 boost::asio::ip::tcp::v6()）来创建 acceptor 对象。在这种情况下，acceptor 将监听服务器上的所有网络接口。

```
boost::asio::io_context io_context;
boost::asio::ip::tcp::acceptor acceptor(io_context, boost::asio::ip::tcp::v4(),
8888);
```

延迟绑定端口：

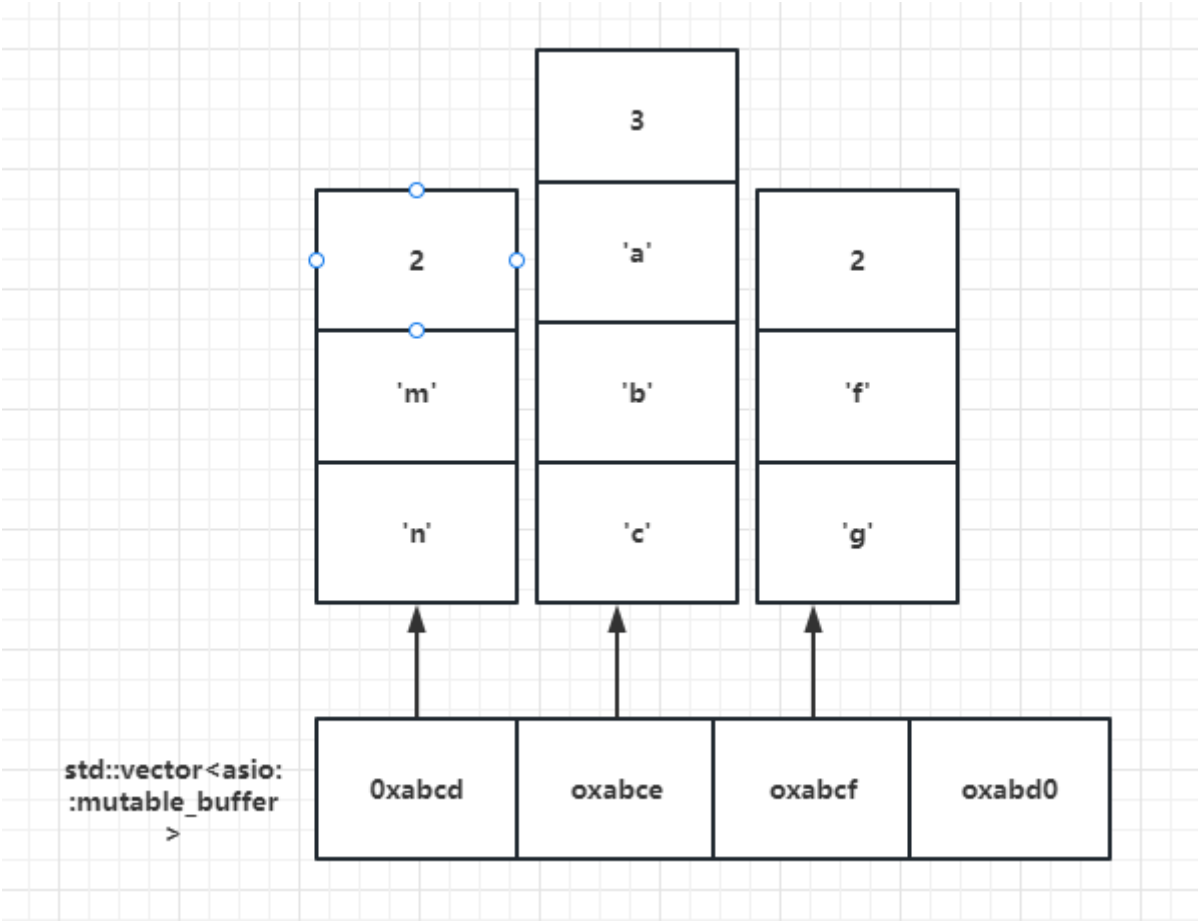
- 有时你可能希望在创建 acceptor 对象后，稍后再将其绑定到指定的地址和端口上。这样的话，在创建 acceptor 时不需要传递端点对象或协议对象。

```
boost::asio::io_context io_context;
boost::asio::ip::tcp::acceptor acceptor(io_context);
// ...
boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::address_v4::any(), 8888);
acceptor.bind(endpoint);
```

- 总的来说，创建 acceptor 对象的方法取决于你的需求。如果你想监听特定的地址和端口，则使用端点对象或协议对象；如果你希望在稍后再绑定端口，则延迟绑定端口。

关于buffer

- 任何网络库都有提供buffer的数据结构，所谓buffer就是接收和发送数据时缓存数据的结构。
- boost::asio提供了**asio::mutable_buffer** 和 **asio::const_buffer**这两个结构，他们是一段连续的空间，首字节存储了后续数据的长度。
- asio::mutable_buffer用于写服务，asio::const_buffer用于读服务。但是这两个结构都没有被asio的api直接使用。
- 对于api的buffer参数，asio提出了MutableBufferSequence和ConstBufferSequence概念，他们是由多个asio::mutable_buffer和asio::const_buffer组成的。也就是说boost::asio为了节省空间，将一部分连续的空间组合起来，作为参数交给api使用。
- 我们可以理解为MutableBufferSequence的数据结构为**std::vector**asio::mutable_buffer 结构如下



- 每个vector存储的都是mutable_buffer的地址，每个mutable_buffer的第一个字节表示数据的长度，后面跟着数据内容。
- 这么复杂的结构交给用户使用并不合适，所以asio提出了buffer()函数，该函数接收多种形式的字节流，该函数返回asio::mutable_buffers_1 o或者asio::const_buffers_1结构的对象。
- 如果传递给buffer()的参数是一个只读类型，则函数返回asio::const_buffers_1 类型对象。
- 如果传递给buffer()的参数是一个可写类型，则返回asio::mutable_buffers_1 类型对象。
- **asio::const_buffers_1**和**asio::mutable_buffers_1**是asio::mutable_buffer和asio::const_buffer的适配器，提供了符合MutableBufferSequence和ConstBufferSequence概念的接口，所以他们可以作为boost::asio的api函数的参数使用。
- 简单概括一下，我们可以用buffer()函数生成我们要用的缓存存储数据。 比如boost的发送接口send要求的参数为ConstBufferSequence类型

```
template<typename ConstBufferSequence>
std::size_t send(const ConstBufferSequence & buffers);
```

- 我们需要手动转换

```
// 模拟构造const_buffer的结构,这样写非常的麻烦
void use_const_buffer() {
 std::string buf = "hello world";
 boost::asio::const_buffer asio_buf(buf.c_str(), buf.length());
 std::vector<boost::asio::const_buffer> buffers_sequence;
 buffers_sequence.push_back(asio_buf);
}
```

字符串的buffer

- 最终buffers_sequence就是可以传递给发送接口send的类型。但是这太复杂了，可以直接用buffer函数转化为send需要的参数类型

```
// 他会自动模拟上述操作
void use_buffer_str() {
 boost::asio::const_buffers_1 output_buffer = boost::asio::buffer("hello world");
}
```

- output_buf可以直接传递给该send接口。我们也可以将数组转化为send接受的类型

数组的buffer

```
void use_buffer_array() {
 const size_t BUF_SIZE_BYTES = 20;
 std::unique_ptr<char[]> buf(new char[BUF_SIZE_BYTES]);
 auto input_buf = boost::asio::buffer(static_cast<void*>(buf.get()),
BUF_SIZE_BYTES);
}
```

- 其中可以不强制转换为void*

流式的buffer

- 对于流式操作，我们可以用streambuf，将输入输出流和streambuf绑定，可以实现流式输入和输出。

```
void use_stream_buffer() {
    boost::asio::streambuf buf;
    std::ostream output(&buf);
    // Writing the message to the stream-based buffer.
    output << "Message1\nMessage2";
    // Now we want to read all data from a streambuf
    // until '\n' delimiter.
    // Instantiate an input stream which uses our
    // stream buffer.
    std::istream input(&buf);
    // We'll read data into this string.
    std::string message1;
    std::getline(input, message1);
    // Now message1 string contains 'Message1'.
}
```

容器的buffer

```
std::vector<char> data = { /* 初始化数据 */ };
boost::asio::const_buffer buffer = boost::asio::buffer(data);
```

指向缓冲区的指针的buffer

```
char data[1024];
boost::asio::mutable_buffer buffer = boost::asio::buffer(data, sizeof(data));
```

同步读写

同步写 write_some

- boost::asio提供了几种同步写的api，write_some可以每次向指定的空间写入固定的字节数，如果写缓冲区满了，就只写一部分，返回写入的字节数。

```
void write_to_socket(boost::asio::ip::tcp::socket& sock) {
 std::string buf = "Hello World!";
 std::size_t total_bytes_written = 0;
 //循环发送
    //write_some返回每次写入的字节数
    //total_bytes_written是已经发送的字节数。
    //每次发送buf.length()- total_bytes_written)字节数据
 while (total_bytes_written != buf.length()) {
  total_bytes_written += sock.write_some(boost::asio::buffer(
   buf.c_str() + total_bytes_written, buf.length() - total_bytes_written));
 }
```

```
void send_data_by_write_some(){
 try {
  std::string raw_ip_address = "192.168.168.123";
  unsigned short port_nums = 8888;
  boost::asio::ip::tcp::endpoint
ep(boost::asio::ip::address::from_string(raw_ip_address), port_nums);
  boost::asio::io_context ioc;
  boost::asio::ip::tcp::socket sock(ioc, ep.protocol());
  sock.connect(ep);
  write_to_socket(sock);
 }
 catch (boost::system::system_error& e) {
  std::cout << "Error occured! Error code = " << e.code()
   << ". Message: " << e.what();
 }
}
```

同步写 send

- write_some使用起来比较麻烦，需要多次调用，asio提供了send函数。send函数会一次性将buffer中的内容发送给对端，如果有部分字节因为发送缓冲区满无法发送，则阻塞等待，直到发送缓冲区可用，则继续发送完成。

```
void send_data_by_send() {
 try {
  std::string raw_ip_address = "192.168.168.123";
  unsigned short port_nums = 8888;
  boost::asio::ip::tcp::endpoint
ep(boost::asio::ip::address::from_string(raw_ip_address), port_nums);
  boost::asio::io_context ioc;
  boost::asio::ip::tcp::socket sock(ioc, ep.protocol());
  sock.connect(ep);
  std::string buf = "hello world";
  // send表示发完你要发的所有数据为止才返回
  //三种放回值 <0 表示系统级错误 =0 对端关闭
  // >0 必为发送的长度
  int send_length = sock.send(boost::asio::buffer(buf.c_str(), buf.length()));

 }
 catch (boost::system::system_error& e) {
  std::cout << "Error occured! Error code = " << e.code()
   << ". Message: " << e.what();
 }
}
```

同步写 write

- 类似send方法，asio还提供了一个write函数，可以一次性将所有数据发送给对端，如果发送缓冲区满了则阻塞，直到发送缓冲区可用，将数据发送完成。

```
void send_data_by_write() {
 try {
  std::string raw_ip_address = "192.168.168.123";
  unsigned short port_nums = 8888;
  boost::asio::ip::tcp::endpoint
ep(boost::asio::ip::address::from_string(raw_ip_address), port_nums);
  boost::asio::io_context ioc;
  boost::asio::ip::tcp::socket sock(ioc, ep.protocol());
  sock.connect(ep);
  std::string buf = "hello world";
  // send表示发完你要发的所有数据为止才返回
  //三种放回值 <0 表示系统级错误 =0 对端关闭
  // >0 必为发送的长度
  int send_length = boost::asio::write(sock, boost::asio::buffer(buf.c_str(),
buf.length()));
 }
 catch (boost::system::system_error& e) {
```

```
  std::cout << "Error occured! Error code = " << e.code()
   << ". Message: " << e.what();
 }
}
```

同步读 read_some

- 同步读和同步写类似，提供了读取指定字节数的接口read_some

```
std::string read_from_socket(boost::asio::ip::tcp::socket& sock) {
 const unsigned short MESSAGE_SIZE = 7;
 char buf[MESSAGE_SIZE];
 std::size_t total_bytes_read = 0;
 while (total_bytes_read != MESSAGE_SIZE) {
  total_bytes_read += sock.read_some(boost::asio::buffer(buf + total_bytes_read,
MESSAGE_SIZE - total_bytes_read));
 }
 return std::string(buf,total_bytes_read);
}
```

```
void read_data_by_read_some() {
 std::string raw_ip_address = "192.168.168.132";
 unsigned short port_num = 888;
 try {
  boost::asio::ip::tcp::endpoint
ep(boost::asio::ip::address::from_string(raw_ip_address), port_num);
  boost::asio::io_context ioc;
  boost::asio::ip::tcp::socket sock(ioc, ep.protocol());
  sock.connect(ep);
  read_from_socket(sock);
 }
 catch (boost::system::system_error& e) {
  std::cout << "Error occured! Error code = " << e.code()
   << ". Message: " << e.what();
  //return e.code().value();
 }
}
```

同步读 receive

- 可以一次性同步接收对方发送的数据

```
int read_data_by_receive() {
    std::string raw_ip_address = "127.0.0.1";
    unsigned short port_num = 3333;
    try {
        boost::asio::ip::tcp::endpoint
```

```
            ep(boost::asio::ip::address::from_string(raw_ip_address),
                port_num);
        boost::asio::io_service ios;
        boost::asio::ip::tcp::socket sock(ios, ep.protocol());
        sock.connect(ep);
        const unsigned char BUFF_SIZE = 7;
         char buffer_receive[BUFF_SIZE];
        int receive_length =  sock.receive(boost::asio::buffer(buffer_receive,
BUFF_SIZE));
        if (receive_length <= 0) {
            cout << "receive failed" << endl;
        }
    }
    catch (boost::system::system_error& e) {
        std::cout << "Error occured! Error code = " << e.code()
            << ". Message: " << e.what();
        return e.code().value();
    }
    return 0;
}
```

同步读 read

- 可以一次性同步读取对方发送的数据

```
using namespace boost;
int read_data_by_read() {
    std::string raw_ip_address = "127.0.0.1";
    unsigned short port_num = 3333;
    try {
        asio::ip::tcp::endpoint
            ep(asio::ip::address::from_string(raw_ip_address),
                port_num);
        asio::io_service ios;
        asio::ip::tcp::socket sock(ios, ep.protocol());
        sock.connect(ep);
        const unsigned char BUFF_SIZE = 7;
        char buffer_receive[BUFF_SIZE];
        int receive_length = asio::read(sock, asio::buffer(buffer_receive,
BUFF_SIZE));
        if (receive_length <= 0) {
            cout << "receive failed" << endl;
        }
    }
    catch (system::system_error& e) {
        std::cout << "Error occured! Error code = " << e.code()
            << ". Message: " << e.what();
        return e.code().value();
    }
    return 0;
 }
```

读取直到指定字符

- 我们可以一直读取，直到读取指定字符结束

```
std::string  read_data_by_until(asio::ip::tcp::socket& sock) {
    asio::streambuf buf;
    // Synchronously read data from the socket until
    // '\n' symbol is encountered.
    asio::read_until(sock, buf, '\n');
    std::string message;
    // Because buffer 'buf' may contain some other data
    // after '\n' symbol, we have to parse the buffer and
    // extract only symbols before the delimiter.
    std::istream input_stream(&buf);
    std::getline(input_stream, message);
    return message;
 }
```

同步读写的客户端与服务端

- 前面我们介绍了boost::asio同步读写的api函数，现在将前面的api串联起来，做一个能跑起来的客户端和服务器。
- 客户端和服务器采用阻塞的同步读写方式完成通信

客户端的设计

- 客户端设计基本思路是根据服务器对端的ip和端口创建一个endpoint，然后创建socket连接这个endpoint，之后就可以用同步读写的方式发送和接收数据了。

```
#include <boost/asio.hpp>
#include <iostream>
using namespace boost::asio::ip;
constexpr int MAX_LENGTH = 1024;
int main()
{
    try {
        //创建上下文
        boost::asio::io_context ioc;
        //构造终端端点
        tcp::endpoint ep(address::from_string("127.0.0.1"), 8888);
        //创建socket
        tcp::socket sock(ioc, ep.protocol());
        //声明错误
        boost::system::error_code error = boost::asio::error::host_not_found;
        //建立连接
        sock.connect(ep, error);
        if (error) {
            std::cout << "connect failed, code is"
```

```
                << error.value() << "message is " << error.message() << std::endl;
            return 0;
        }
        std::cout << "Enter message: ";
        char request[MAX_LENGTH];
        std::cin.getline(request, MAX_LENGTH);
        size_t request_length = std::strlen(request);
        boost::asio::write(sock, boost::asio::buffer(request, request_length));

        char reply[MAX_LENGTH];
        size_t reply_length = boost::asio::read(sock, boost::asio::buffer(reply,
request_length));
        std::cout << "reply is " << std::endl;
        std::cout.write(reply, reply_length);
        std::cout << std::endl;

    }
    catch (std::exception& e) {
        std::cerr << "Exception" << e.what() << std::endl;
    }
}
```

服务端设计

session函数

- 创建session函数，该函数为服务器处理客户端请求，每当我们获取客户端连接后就调用该函数。在session函数里里进行echo方式的读写，所谓echo就是应答式的处理

```
void session(socket_ptr sock) {
 try {
  while (true) {
   char data[max_length];
   memset(data, '\0', max_length);
   boost::system::error_code error;
   //size_t length = boost::asio::read(sock, boost::asio::buffer(data,
max_length), error);
   size_t length = sock->read_some(boost::asio::buffer(data, max_length), error);
   if (error == boost::asio::error::eof) {
    std::cout << "connection closed by peer" << std::endl;
    break;
   }
   else if (error) {
    throw boost::system::system_error(error);
   }
   std::cout << "receive from " << sock->remote_endpoint().address().to_string()
<< std::endl;
   std::cout << "receive message is " << data << std::endl;
   //回传给对方
   boost::asio::write(*sock, boost::asio::buffer(data, length));
  }
```

```
 }
 catch (std::exception& e) {
  std::cerr << "Exception in thread: " << e.what() << std::endl;
 }
}
```

server函数

- server函数根据服务器ip和端口创建服务器acceptor用来接收数据，用socket接收新的连接，然后为这个socket创建session。

```
void server(boost::asio::io_context& io_context, unsigned short port) {
 // boost::asio::ip::address_v4::any()第一个参数与这个效果一样
 tcp::endpoint ep(tcp::v4(), port);
 tcp::acceptor a(io_context, ep);
 while (true) {
  socket_ptr sock(new tcp::socket(io_context));
  a.accept(*sock);
  auto t = std::make_shared<std::thread>(session, sock);
  thread_set.insert(t);
 }
}
```

- 创建线程调用session函数可以分配独立的线程用于socket的读写，保证acceptor不会因为socket的读写而阻塞。

完整服务端代码

```
#include <iostream>
#include <boost/asio.hpp>
#include <thread>
#include <set>

using namespace boost::asio::ip;
constexpr int max_length = 1024;
typedef std::shared_ptr<tcp::socket> socket_ptr;
std::set<std::shared_ptr<std::thread>> thread_set;

void session(socket_ptr sock) {
 try {
  while (true) {
   char data[max_length];
   memset(data, '\0', max_length);
   boost::system::error_code error;
   //size_t length = boost::asio::read(sock, boost::asio::buffer(data,
max_length), error);
   size_t length = sock->read_some(boost::asio::buffer(data, max_length), error);
   if (error == boost::asio::error::eof) {
```

```
      std::cout << "connection closed by peer" << std::endl;
      break;
     }
     else if (error) {
      throw boost::system::system_error(error);
     }
     std::cout << "receive from " << sock->remote_endpoint().address().to_string()
<< std::endl;
     std::cout << "receive message is " << data << std::endl;
     //回传给对方
     boost::asio::write(*sock, boost::asio::buffer(data, length));
    }
   }
   catch (std::exception& e) {
    std::cerr << "Exception in thread: " << e.what() << std::endl;
   }
}

void server(boost::asio::io_context& io_context, unsigned short port) {
   // boost::asio::ip::address_v4::any()第一个参数与这个效果一样
   tcp::endpoint ep(tcp::v4(), port);
   tcp::acceptor a(io_context, ep);
   while (true) {
    socket_ptr sock(new tcp::socket(io_context));
    a.accept(*sock);
    auto t = std::make_shared<std::thread>(session, sock);
    thread_set.insert(t);
   }
}
int main()
{
   try {
    boost::asio::io_context ioc;
    server(ioc, 8888);
    for (auto& td : thread_set) {
     if (td->joinable()) {
      td->join();
     }
    }
   }
   catch (std::exception& e) {

   }
}
```

同步读写的优劣

1. 同步读写的缺陷在于读写是阻塞的，如果客户端对端不发送数据服务器的read操作是阻塞的，这将导致服务器处于阻塞等待状态。
2. 可以通过开辟新的线程为新生成的连接处理读写，但是一个进程开辟的线程是有限的，约为2048个线程，在Linux环境可以通过unlimit增加一个进程开辟的线程数，但是线程过多也会导致切换消耗的时间片

较多。
3. 该服务器和客户端为应答式，实际场景为全双工通信模式，发送和接收要独立分开。
4. 该服务器和客户端未考虑粘包处理。

- 综上所述，是我们这个服务器和客户端存在的问题，为解决上述问题，我们在接下里的文章里做不断完善和改进，主要以异步读写改进上述方案。 当然同步读写的方式也有其优点，比如客户端连接数不多，而且服务器并发性不高的场景，可以使用同步读写的方式。使用同步读写能简化编码难度。

asio异步读写操作及注意事项

- 我们定义一个session类，这个session类表示服务器处理客户端连接的管理类

```
#pragma once
#include <thread>
#include <boost/asio.hpp>
#include <iostream>

class Session
{
public:
 Session(std::shared_ptr<boost::asio::ip::tcp::socket> socket);
 void Connect(const boost::asio::ip::tcp::endpoint& ep);
private:
 std::shared_ptr<boost::asio::ip::tcp::socket> sock;
};
```

```
#include "Session.h"

Session::Session(std::shared_ptr<boost::asio::ip::tcp::socket> socket):
sock(socket)
{
}

void Session::Connect(const boost::asio::ip::tcp::endpoint& ep)
{
 sock->connect(ep);
}
```

异步写操作

- 在写操作前，我们先封装一个Node结构，用来管理要发送和接收的数据，该结构包含数据域首地址，数据的总长度，以及已经处理的长度(已读的长度或者已写的长度)

```
class MsgNode {
public:
```

```
//用来构造写节点
 MsgNode(const char* msg, int total_len) : _total_len(total_len), _cur_len(0) {
  _msg = new char[total_len];
  memcpy(_msg, msg, total_len);
 }

//用来构造读节点
 MsgNode(int total_len) :_total_len(total_len), _cur_len(0) {
  _msg = new char[total_len];
 }

 ~MsgNode() {
  delete[] _msg;
 }

// 要发送的总长度
int _total_len;
// 当前已经发送了多少长度
int _cur_len;
// 指向数据的指针
char* _msg;
};
```

- 写了两个构造函数，两个参数的负责构造写节点，一个参数的负责构造读节点。
- 接下来为Session添加异步写操作和负责发送写数据的节点

```
class Session
{
public:
 Session(std::shared_ptr<boost::asio::ip::tcp::socket> socket);
 void Connect(const boost::asio::ip::tcp::endpoint& ep);
//第一个参数错误码，第二个参数当前准备发送的长度，第三个参数为占位符用来增加引用
 void WriteCallBackErr(const boost::system::error_code& ec, std::size_t
bytes_transferred,
  std::shared_ptr<MsgNode>);
 void WriteToSocketErr(const std::string buf);
private:
 std::shared_ptr<boost::asio::ip::tcp::socket> sock;
 std::shared_ptr<MsgNode> _send_node;
};
```

- WriteToSocketErr函数为我们封装的写操作，WriteCallBackErr为异步写操作回调的函数，为什么会有三个参数呢，
- 我们可以看一下asio源码

```
BOOST_ASIO_COMPLETION_TOKEN_FOR(void (boost::system::error_code,
        std::size_t)) WriteToken
          BOOST_ASIO_DEFAULT_COMPLETION_TOKEN_TYPE(executor_type)>
```

```
  BOOST_ASIO_INITFN_AUTO_RESULT_TYPE_PREFIX(WriteToken,
      void (boost::system::error_code, std::size_t))
  async_write_some(const ConstBufferSequence& buffers,
      BOOST_ASIO_MOVE_ARG(WriteToken)token
        BOOST_ASIO_DEFAULT_COMPLETION_TOKEN(executor_type))
```

- sync_write_some是异步写的函数，这个异步写函数有两个参数，第一个参数为ConstBufferSequence常引用类型的buffers，
- 第二个参数为WriteToken类型，而WriteToken在上面定义了，是一个函数对象类型，返回值为void，参数为error_code和size_t，
- 所以我们为了调用async_write_some函数也要传入一个符合WriteToken定义的函数，就是我们声明的WriteCallBackErr函数，
- 前两个参数为WriteToken规定的参数，第三个参数为MsgNode的智能指针，这样通过智能指针保证我们发送的Node生命周期延长。
- 我们看一下WriteToSocketErr函数的具体实现

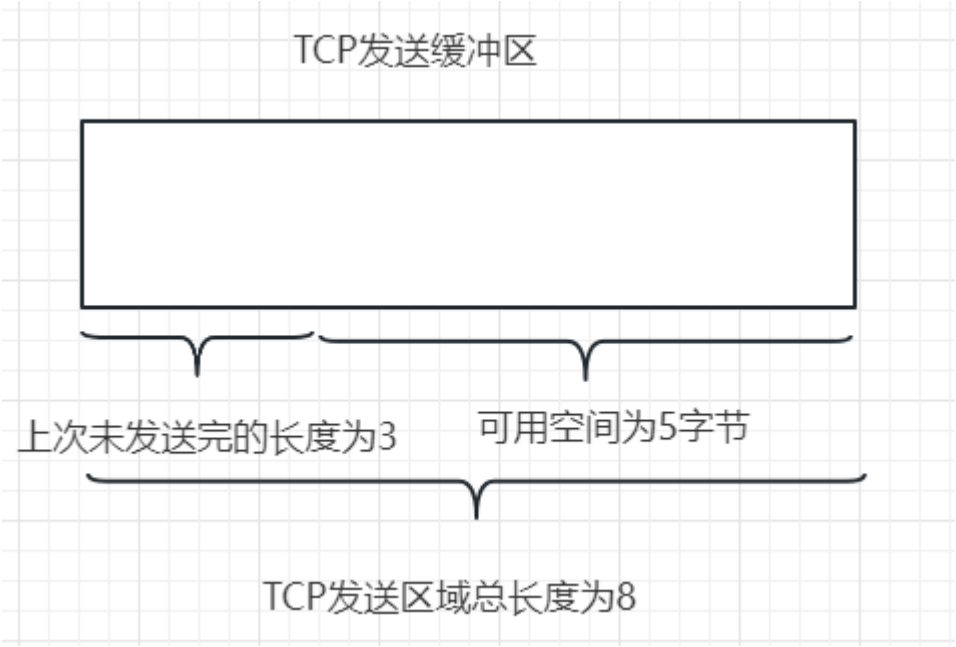
```
void Session::WriteToSocketErr(const std::string buf)
{
 //定义你要写的数据节点
 _send_node = std::make_shared<MsgNode>(buf.c_str(), buf.length());
 //通过socket官方异步写发送数据
 this->sock->async_read_some(boost::asio::buffer(_send_node->_msg, _send_node-
>_total_len),
  std::bind(&Session::WriteCallBackErr, this, std::placeholders::_1,
std::placeholders::_2,
   _send_node));
}
```

- 因为WriteCallBackErr函数为三个参数且为成员函数，而async_write_some需要的回调函数为两个参数，所以我们通过bind将三个参数转换为两个参数的普通函数。
- 我们看看回调函数的实现

```
void Session::WriteCallBackErr(const boost::system::error_code& ec, std::size_t
bytes_transferred, std::shared_ptr<MsgNode> msg_node)
{
 //当前要转发的数据加上已经转发的数据 与 发送总数据做比较
 if (bytes_transferred + msg_node->_cur_len < msg_node->_total_len) {
  //更新当前已发送的数据
  _send_node->_cur_len += bytes_transferred;
  // 递归调用继续发送未发送玩的数据
  this->sock->async_read_some(boost::asio::buffer(_send_node->_msg
   + _send_node->_cur_len, _send_node->_total_len - _send_node->_cur_len),
   std::bind(&Session::WriteCallBackErr, this, std::placeholders::_1,
```

```
std::placeholders::_2,
    _send_node));
 }
}
```

- 在WriteCallBackErr函数里判断如果已经发送的字节数没有达到要发送的总字节数，那么就更新节点已经发送的长度，然后计算剩余要发送的长度，如果有数据未发送完，再次调用async_write_some函数异步发送。
- **但是这个函数并不能投入实际应用，因为async_write_some回调函数返回已发送的字节数可能并不是全部长度。比如TCP发送缓存区总大小为8字节，但是有3字节未发送(上一次未发送完)，这样剩余空间为5字节**



- 此时我们调用async_write_some发送hello world!实际发送的长度就是为5，也就是只发送了hello，剩余world!通过我们的回调继续发送。
- 而实际开发的场景用户是不清楚底层tcp的多路复用调用情况的，用户想发送数据的时候就调用WriteToSocketErr,或者循环调用WriteToSocketErr，**很可能在一次没发送完数据还未调用回调函数时再次调用WriteToSocketErr**，因为boost::asio封装的是epoll和iocp等多路复用模型，当写事件就绪后就发数据，发送的数据按照async_write_some调用的顺序发送，所以回调函数内调用的async_write_some可能并没有被及时调用。
- 比如如下代码：

```
//用户发送数据
WriteToSocketErr("Hello World!");
//用户无感知下层调用情况又一次发送了数据
WriteToSocketErr("Hello World!");
```

- 那么很可能第一次只发送了Hello，后面的数据没发完，第二次发送了Hello World!之后又发送了World!
- 所以对端收到的数据很可能是"HelloHello World! World!"

- 那怎么解决这个问题呢，**我们可以通过队列保证应用层的发送顺序。**我们在Session中定义一个发送队列，然后重新定义正确的异步发送函数和回调处理

```
class Session
{
public:
 Session(std::shared_ptr<boost::asio::ip::tcp::socket> socket);
 void Connect(const boost::asio::ip::tcp::endpoint& ep);
 //第一个参数错误码，第二个参数当前准备发送的长度，第三个参数为占位符用来增加引用
 void WriteCallBackErr(const boost::system::error_code& ec, std::size_t
bytes_transferred,
  std::shared_ptr<MsgNode> msg_node);
 void WriteToSocketErr(const std::string buf);

 //正确的处理方式
 void WriteCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred);
 void WriteToSocket(const std::string buf);

private:
 //用来装要发送的数据
 std::queue<std::shared_ptr<MsgNode>> _send_queue;
 std::shared_ptr<boost::asio::ip::tcp::socket> sock;
 std::shared_ptr<MsgNode> _send_node;
 //标记是否有数据未发送完
 bool _send_pending;
};
```

- 定义了bool变量_send_pending，该变量为true表示一个节点还未发送完。
- _send_queue用来缓存要发送的消息节点，是一个队列。
- 我们实现异步发送功能

```
void Session::WriteCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred)
{
 if (ec.value() != 0) {
  std::cout << "Error , code is " << ec.value() << " . Message is " <<
ec.message();
  return;
 }

 //取出队首元素即当前未发送完数据
 auto& send_data = _send_queue.front();
 send_data->_cur_len += bytes_transferred;
 //数据未发送完， 则继续发送
 if (send_data->_cur_len < send_data->_total_len) {
  this->sock->async_write_some(boost::asio::buffer(send_data->_msg + send_data-
>_cur_len, send_data->_total_len - send_data->_cur_len),
```

```
    std::bind(&Session::WriteCallBack,
     this, std::placeholders::_1, std::placeholders::_2));
   return;
  }

  //如果发送完，则pop出队首元素
  _send_queue.pop();

  //如果队列为空，则说明所有数据都发送完,将pending设置为false
  if (_send_queue.empty()) {
   _send_pending = false;
  }

  //如果队列不是空，则继续将队首元素发送
  if (!_send_queue.empty()) {
   auto& send_data = _send_queue.front();
   this->sock->async_write_some(boost::asio::buffer(send_data->_msg + send_data-
>_cur_len, send_data->_total_len - send_data->_cur_len),
    std::bind(&Session::WriteCallBack,
     this, std::placeholders::_1, std::placeholders::_2));
  }
}

void Session::WriteToSocket(const std::string buf)
{
  //将数据放入队列中
  _send_queue.emplace(new MsgNode(buf.c_str(), buf.length()));
  //如果还有数据未发送完，直接返回
  if (_send_pending) {
   return;
  }

  //异步发送数据，因为异步所以不会一下发送完
  this->sock->async_write_some(boost::asio::buffer(buf),
std::bind(&Session::WriteCallBack, this, std::placeholders::_1,
std::placeholders::_2));
  _send_pending = true;
}
```

- async_write_some函数不能保证每次回调函数触发时发送的长度为要总长度，这样我们每次都要在回调函数判断发送数据是否完成，asio提供了一个更简单的发送函数**async_send**，这个函数在发送的长度**未达到我们要求的长度时就不会触发回调**，所以触发回调函数时要么时**发送出错了**要么是**发送完成**了,其**内部的实现原理**就是帮我们**不断的调用async_write_some直到完成发送**，所以**async_send不能和async_write_some混合使用**，我们基于async_send封装另外一个发送函数

```
//这个回调函数被调用并且没有异常肯定是发完了一个数据
void Session::WriteAllCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred) {
  if (ec.value() != 0) {
   std::cout << "Error occured! Error code = "
```

```
    << ec.value()
    << ". Message: " << ec.message();
   return;
  }
  //如果发送完，则pop出队首元素
  _send_queue.pop();
  //如果队列为空，则说明所有数据都发送完,将pending设置为false
  if (_send_queue.empty()) {
   _send_pending = false;
  }
  //如果队列不是空，则继续将队首元素发送
  if (!_send_queue.empty()) {
   auto& send_data = _send_queue.front();
         //send的时候由于使用的是async_send所以这可以不使用数据偏移，而直接发送buf，
boost::asio::buffer(buf)
   this->sock->async_send(boost::asio::buffer(send_data->_msg + send_data-
>_cur_len, send_data->_total_len - send_data->_cur_len),
    std::bind(&Session::WriteAllCallBack,
     this, std::placeholders::_1, std::placeholders::_2));
  }
 }

 //不能与async_write_some混合使用
 void Session::WriteAllToSocket(const std::string buf) {
  //插入发送队列
  _send_queue.emplace(new MsgNode(buf.c_str(), buf.length()));
  //pending状态说明上一次有未发送完的数据
  if (_send_pending) {
   return;
  }
  //异步发送数据，因为异步所以不会一下发送完
  this->sock->async_send(boost::asio::buffer(buf),
   std::bind(&Session::WriteAllCallBack, this,
    std::placeholders::_1, std::placeholders::_2));
  _send_pending = true;
 }
```

异步读操作

- 接下来介绍异步读操作，异步读操作和异步的写操作类似同样又async_read_some和async_receive函数，前者触发的回调函数获取的读数据的长度可能会小于要求读取的总长度，后者触发的回调函数读取的数据长度等于读取的总长度。
- 先基于async_read_some封装一个读取的函数ReadFromSocket，同样在Session类的声明中添加一些变量

```
 void Session::ReadFromSocket() {
  if (_recv_pending) {
   return;
  }
  //可以调用构造函数直接构造，但不可用已经构造好的智能指针赋值
  /*auto _recv_nodez = std::make_unique<MsgNode>(RECVSIZE);
```

```
 _recv_node = _recv_nodez;*/
 _recv_node = std::make_shared<MsgNode>(RECVSIZE);
    //发送要读的数据
 sock->async_read_some(boost::asio::buffer(_recv_node->_msg, _recv_node-
>_total_len), std::bind(&Session::ReadCallBack, this,
  std::placeholders::_1, std::placeholders::_2));
 _recv_pending = true;
}

void Session::ReadCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred) {
 _recv_node->_cur_len += bytes_transferred;
 //没读完继续读
 if (_recv_node->_cur_len < _recv_node->_total_len) {
        //要加上数据的偏移
  sock->async_read_some(boost::asio::buffer(_recv_node->_msg + _recv_node-
>_cur_len,
   _recv_node->_total_len - _recv_node->_cur_len),
std::bind(&Session::ReadCallBack, this,
    std::placeholders::_1, std::placeholders::_2));
  return;
 }
 //将数据投递到队列里交给逻辑线程处理，此处略去
 //如果读完了则将标记置为false
 _recv_pending = false;
 //指针置空
 _recv_node = nullptr;
}
```

- 我们基于async_receive再封装一个接收数据的函数

```
void Session::ReadAllFromSocket() {
 if (_recv_pending) {
  return;
 }
 //可以调用构造函数直接构造，但不可用已经构造好的智能指针赋值
 /*auto _recv_nodez = std::make_unique<MsgNode>(RECVSIZE);
 _recv_node = _recv_nodez;*/
 _recv_node = std::make_shared<MsgNode>(RECVSIZE);
 sock->async_receive(boost::asio::buffer(_recv_node->_msg, _recv_node-
>_total_len), std::bind(&Session::ReadAllCallBack, this,
  std::placeholders::_1, std::placeholders::_2));
 _recv_pending = true;
}
void Session::ReadAllCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred) {
 _recv_node->_cur_len += bytes_transferred;
 //将数据投递到队列里交给逻辑线程处理，此处略去
 //如果读完了则将标记置为false
 _recv_pending = false;
 //指针置空
```

```
 _recv_node = nullptr;
}
```

总结 完整代码

Session.h头文件

```
#pragma once
#include <thread>
#include <boost/asio.hpp>
#include <iostream>
#include <queue>
//最大报文接收大小
const int RECVSIZE = 1024;
class MsgNode {
public:
 MsgNode(const char* msg, int total_len) : _total_len(total_len), _cur_len(0) {
  _msg = new char[total_len];
  memcpy(_msg, msg, total_len);
 }

 MsgNode(int total_len) :_total_len(total_len), _cur_len(0) {
  _msg = new char[total_len];
 }

 ~MsgNode() {
  delete[] _msg;
 }

 // 要发送的总长度
 int _total_len;
 // 当前发送了多少长度
 int _cur_len;
 // 指向数据的指针
 char* _msg;
};

class Session
{
public:
 Session(std::shared_ptr<boost::asio::ip::tcp::socket> socket);
 void Connect(const boost::asio::ip::tcp::endpoint& ep);
 //第一个参数错误码，第二个参数当前准备发送的长度，第三个参数为占位符用来增加引用
 void WriteCallBackErr(const boost::system::error_code& ec, std::size_t
bytes_transferred,
  std::shared_ptr<MsgNode> msg_node);
 void WriteToSocketErr(const std::string buf);

 //正确的处理方式
 void WriteCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred);
```

```
 void WriteToSocket(const std::string buf);

 void WriteAllCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred);
 void WriteAllToSocket(const std::string buf);

 void ReadFromSocket();
 void ReadCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred);

 void ReadAllFromSocket();
 void ReadAllCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred);

private:
 //用来装要发送的数据
 std::queue<std::shared_ptr<MsgNode>> _send_queue;
 std::shared_ptr<boost::asio::ip::tcp::socket> sock;
 std::shared_ptr<MsgNode> _send_node;
 //标记是否有数据未发送完
 bool _send_pending;

 //读取节点
 std::shared_ptr<MsgNode> _recv_node;
 //标记数据是否接收完全
 bool _recv_pending;
};
```

Session.cpp

```
#include "Session.h"

Session::Session(std::shared_ptr<boost::asio::ip::tcp::socket> socket):
sock(socket), _send_pending(false),_recv_pending(false)
{
}

void Session::Connect(const boost::asio::ip::tcp::endpoint& ep)
{
 sock->connect(ep);
}

void Session::WriteCallBackErr(const boost::system::error_code& ec, std::size_t
bytes_transferred, std::shared_ptr<MsgNode> msg_node)
{
 //当前要转发的数据加上已经转发的数据 与 发送总数据做比较
 if (bytes_transferred + msg_node->_cur_len < msg_node->_total_len) {
  //更新当前已发送的数据
  _send_node->_cur_len += bytes_transferred;
  // 递归调用继续发送未发送玩的数据
```

```
  this->sock->async_read_some(boost::asio::buffer(_send_node->_msg
   + _send_node->_cur_len, _send_node->_total_len - _send_node->_cur_len),
   std::bind(&Session::WriteCallBackErr, this, std::placeholders::_1,
std::placeholders::_2,
    _send_node));
 }
}

void Session::WriteToSocketErr(const std::string buf)
{
 //定义你要写的数据节点
 _send_node = std::make_shared<MsgNode>(buf.c_str(), buf.length());
 //通过socket官方异步写发送数据
 this->sock->async_read_some(boost::asio::buffer(_send_node->_msg, _send_node-
>_total_len),
  std::bind(&Session::WriteCallBackErr, this, std::placeholders::_1,
std::placeholders::_2,
   _send_node));
}

void Session::WriteCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred)
{
 if (ec.value() != 0) {
  std::cout << "Error , code is " << ec.value() << " . Message is " <<
ec.message();
  return;
 }

 //取出队首元素即当前未发送完数据
 auto& send_data = _send_queue.front();
 send_data->_cur_len += bytes_transferred;
 //数据未发送完， 则继续发送
 if (send_data->_cur_len < send_data->_total_len) {
  this->sock->async_write_some(boost::asio::buffer(send_data->_msg + send_data-
>_cur_len, send_data->_total_len - send_data->_cur_len),
   std::bind(&Session::WriteCallBack,
    this, std::placeholders::_1, std::placeholders::_2));
  return;
 }

 //如果发送完，则pop出队首元素
 _send_queue.pop();

 //如果队列为空，则说明所有数据都发送完,将pending设置为false
 if (_send_queue.empty()) {
  _send_pending = false;
 }

 //如果队列不是空，则继续将队首元素发送
 if (!_send_queue.empty()) {
  auto& send_data = _send_queue.front();
  this->sock->async_write_some(boost::asio::buffer(send_data->_msg + send_data-
>_cur_len, send_data->_total_len - send_data->_cur_len),
```

```
    std::bind(&Session::WriteCallBack,
     this, std::placeholders::_1, std::placeholders::_2));
  }
 }

 void Session::WriteToSocket(const std::string buf)
 {
  //将数据放入队列中
  _send_queue.emplace(new MsgNode(buf.c_str(), buf.length()));
  //如果还有数据未发送完，直接返回
  if (_send_pending) {
   return;
  }

  //异步发送数据，因为异步所以不会一下发送完
  this->sock->async_write_some(boost::asio::buffer(buf),
 std::bind(&Session::WriteCallBack, this, std::placeholders::_1,
 std::placeholders::_2));
  _send_pending = true;
 }

 //这个回调函数被调用并且没有异常肯定是发完了一个数据
 void Session::WriteAllCallBack(const boost::system::error_code& ec, std::size_t
 bytes_transferred) {
  if (ec.value() != 0) {
   std::cout << "Error occured! Error code = "
    << ec.value()
    << ". Message: " << ec.message();
   return;
  }
  //如果发送完，则pop出队首元素
  _send_queue.pop();
  //如果队列为空，则说明所有数据都发送完,将pending设置为false
  if (_send_queue.empty()) {
   _send_pending = false;
  }
  //如果队列不是空，则继续将队首元素发送
  if (!_send_queue.empty()) {
   auto& send_data = _send_queue.front();
   this->sock->async_send(boost::asio::buffer(send_data->_msg + send_data-
 >_cur_len, send_data->_total_len - send_data->_cur_len),
    std::bind(&Session::WriteAllCallBack,
     this, std::placeholders::_1, std::placeholders::_2));
  }
 }

 //不能与async_write_some混合使用
 void Session::WriteAllToSocket(const std::string buf) {
  //插入发送队列
  _send_queue.emplace(new MsgNode(buf.c_str(), buf.length()));
  //pending状态说明上一次有未发送完的数据
  if (_send_pending) {
   return;
  }
```

```
 //异步发送数据，因为异步所以不会一下发送完
 this->sock->async_send(boost::asio::buffer(buf),
  std::bind(&Session::WriteAllCallBack, this,
   std::placeholders::_1, std::placeholders::_2));
 _send_pending = true;
}

void Session::ReadFromSocket() {
 if (_recv_pending) {
  return;
 }
 //可以调用构造函数直接构造，但不可用已经构造好的智能指针赋值
 /*auto _recv_nodez = std::make_unique<MsgNode>(RECVSIZE);
 _recv_node = _recv_nodez;*/
 _recv_node = std::make_shared<MsgNode>(RECVSIZE);
 sock->async_read_some(boost::asio::buffer(_recv_node->_msg, _recv_node-
>_total_len), std::bind(&Session::ReadCallBack, this,
  std::placeholders::_1, std::placeholders::_2));
 _recv_pending = true;
}

void Session::ReadCallBack(const boost::system::error_code& ec, std::size_t
bytes_transferred) {
 _recv_node->_cur_len += bytes_transferred;
 //没读完继续读
 if (_recv_node->_cur_len < _recv_node->_total_len) {
  sock->async_read_some(boost::asio::buffer(_recv_node->_msg + _recv_node-
>_cur_len,
   _recv_node->_total_len - _recv_node->_cur_len),
std::bind(&Session::ReadCallBack, this,
    std::placeholders::_1, std::placeholders::_2));
  return;
 }
 //将数据投递到队列里交给逻辑线程处理，此处略去
 //如果读完了则将标记置为false
 _recv_pending = false;
 //指针置空
 _recv_node = nullptr;
}

void Session::ReadAllFromSocket() {
 if (_recv_pending) {
  return;
 }
 //可以调用构造函数直接构造，但不可用已经构造好的智能指针赋值
 /*auto _recv_nodez = std::make_unique<MsgNode>(RECVSIZE);
 _recv_node = _recv_nodez;*/
 _recv_node = std::make_shared<MsgNode>(RECVSIZE);
 sock->async_receive(boost::asio::buffer(_recv_node->_msg, _recv_node-
>_total_len), std::bind(&Session::ReadAllCallBack, this,
  std::placeholders::_1, std::placeholders::_2));
 _recv_pending = true;
}
void Session::ReadAllCallBack(const boost::system::error_code& ec, std::size_t
```

```
bytes_transferred) {
 _recv_node->_cur_len += bytes_transferred;
 //将数据投递到队列里交给逻辑线程处理，此处略去
 //如果读完了则将标记置为false
 _recv_pending = false;
 //指针置空
 _recv_node = nullptr;
}
```

- 发送的时候推荐使用 **asyc_send** 而接受的时候 推荐使用 **asyc_rend_some 函数**

asio官方案例存在的隐患

Session类

- Session类主要是处理客户端消息收发的会话类，为了简单起见，我们不考虑粘包问题，也不考虑支持手动调用发送的接口，只以应答的方式发送和接收固定长度(1024字节长度)的数据。

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
class Session
{
public:
 Session(boost::asio::io_context& ioc) :sock(ioc) {

 }
 boost::asio::ip::tcp::socket& Socket() {
  return sock;
 }

 void Start();

private:
 void handle_read(const boost::system::error_code& ec,
  size_t bytes_transferred);
 void handle_write(const boost::system::error_code& ec);
 boost::asio::ip::tcp::socket sock;
 enum{max_length = 1024};
 char _data[max_length];
};
```

1. _data用来接收客户端传递的数据
2. socket为单独处理客户端读写的socket。
3. handle_read和handle_write分别为读回调函数和写回调函数。

- 接下来我们实现Session类

```
#include "Session.h"
#include <iostream>

void Session::Start() {
 memset(_data, 0, max_length);
 sock.async_read_some(boost::asio::buffer(_data, max_length),
  std::bind(&Session::handle_read, this, std::placeholders::_1,
std::placeholders::_2));
}

void Session::handle_read(const boost::system::error_code& error,
 size_t bytes_transferred) {
    if (!error) {
        std::cout << "server receive data is " << _data << std::endl;
        boost::asio::async_write(sock, boost::asio::buffer(_data,
bytes_transfered),
            std::bind(&Session::handle_write, this, std::placeholders::_1));
    }
    else {
        std::cout << "read error" << std::endl;
        delete this;
    }
}
void Session::handle_write(const boost::system::error_code& error) {
    if(!error){
    memset(_data, 0, max_length);
    sock.async_read_some(boost::asio::buffer(_data, max_length),
std::bind(&Session::handle_read,
        this, std::placeholders::_1, std::placeholders::_2));
     }
    else {
        delete this;
    }
}
```

- 在Start方法中我们调用异步读操作，监听对端发送的消息。当对端发送数据后，触发handle_read函数
- handle_read函数内将收到的数据发送给对端，当发送完成后触发handle_write回调函数。
- handle_write函数内又一次监听了读事件，如果对端有数据发送过来则触发handle_read，我们再将收到的数据发回去。从而达到应答式服务的效果。

Server类

- Server类为服务器接收连接的管理类

```
class Server {
public:
 Server(boost::asio::io_context& ioc, unsigned short port);
private:
 void start_accept();
 void handle_accept(Session* new_session, const boost::system::error_code& error);
```

```
 boost::asio::io_context& _ioc;
 boost::asio::ip::tcp::acceptor _acceptor;
};
```

Server实现

```
Server::Server(boost::asio::io_context& ioc, unsigned short port) : _ioc(ioc),
_acceptor(ioc, boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), port)) {
    std::cout << "Server start success, on port: " << port << std::endl;
    start_accept();
}

void Server::start_accept() {
    Session* new_session = new Session(_ioc);
    _acceptor.async_accept(new_session->Socket(),
        std::bind(&Server::handle_accept, this, new_session,
std::placeholders::_1));
}
void Server::handle_accept(Session* new_session, const boost::system::error_code&
error) {
    if (!error) {
        new_session->Start();
    }
    else {
        std::cout << "accept error" << std::endl;
        delete new_session;
    }
    // 处理完事件后继续监听连接
    start_accept();
}
```

隐患

- 该demo示例为仿照asio官网编写的，其中存在隐患，**就是当服务器即将发送数据前(调用async_write前)，此刻客户端中断**，服务器此时调用async_write会触发发送回调函数，判断ec为非0进而执行**delete this逻辑回收session**。但要注意的是客户端关闭后，在tcp层面会触发读就绪事件，服务器会触发读事件回调函数。在读事件回调函数中判断错误码ec为非0，**进而再次执行delete操作，从而造成二次析构，这是极度危险的。**

总结

- 本文介绍了异步的应答服务器设计，但是这种服务器并不会在实际生产中使用，主要有两个原因:

1. 因为该服务器的发送和接收以应答的方式交互，而并不能做到应用层想随意发送的目的，也就是未做到完全的收发分离(全双工逻辑)。
2. 该服务器未处理粘包，序列化，以及逻辑和收发线程解耦等问题。
3. 该服务器存在二次析构的风险。

- 对于官方案例，他考虑到了这个二次析构问题，所以它只会在读成功后进行写，这就保证了同一时期只有一个读取或者写(单工)，但是这依然是一个隐患。

完整代码

头文件

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
class Session
{
public:
 Session(boost::asio::io_context& ioc) :sock(ioc) {

 }
 boost::asio::ip::tcp::socket& Socket() {
  return sock;
 }

 void Start();

private:
 void handle_read(const boost::system::error_code& ec,
  size_t bytes_transferred);
 void handle_write(const boost::system::error_code& ec);
 boost::asio::ip::tcp::socket sock;
 enum{max_length = 1024};
 char _data[max_length];
};

class Server {
public:
 Server(boost::asio::io_context& ioc, unsigned short port);
private:
 void start_accept();
 void handle_accept(Session* new_session, const boost::system::error_code& error);
 boost::asio::io_context& _ioc;
 boost::asio::ip::tcp::acceptor _acceptor;
};
```

实现

```
#include "Session.h"
#include <iostream>

void Session::Start() {
 memset(_data, 0, max_length);
 sock.async_read_some(boost::asio::buffer(_data, max_length),
```

```
  std::bind(&Session::handle_read, this, std::placeholders::_1,
std::placeholders::_2));
}

void Session::handle_read(const boost::system::error_code& error,
 size_t bytes_transferred) {
    if (!error) {
        std::cout << "server receive data is " << _data << std::endl;
        boost::asio::async_write(sock, boost::asio::buffer(_data,
bytes_transferred),
            std::bind(&Session::handle_write, this, std::placeholders::_1));
    }
    else {
        std::cout << "read error" << std::endl;
        delete this;
    }
}
void Session::handle_write(const boost::system::error_code& error) {
    if(!error){
    memset(_data, 0, max_length);
    sock.async_read_some(boost::asio::buffer(_data, max_length),
std::bind(&Session::handle_read,
        this, std::placeholders::_1, std::placeholders::_2));
     }
    else {
        delete this;
    }
}

Server::Server(boost::asio::io_context& ioc, unsigned short port) : _ioc(ioc),
_acceptor(ioc, boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), port)) {
    std::cout << "Server start success, on port: " << port << std::endl;
    start_accept();
}

void Server::start_accept() {
    Session* new_session = new Session(_ioc);
    _acceptor.async_accept(new_session->Socket(),
        std::bind(&Server::handle_accept, this, new_session,
std::placeholders::_1));
}
void Server::handle_accept(Session* new_session, const boost::system::error_code&
error) {
    if (!error) {
        new_session->Start();
    }
    else {
        std::cout << "accept error" << std::endl;
        delete new_session;
    }
    // 处理完事件后继续监听连接
    start_accept();
}
```

使用伪闭包实现连接的安全回收

- 之前的异步服务器为echo模式，但其存在安全隐患，就是在极端情况下客户端关闭导致触发写和读回调函数，二者都进入错误处理逻辑，进而造成二次析构的问题。
- 下面我们介绍通过C11智能指针构造成一个伪闭包的状态延长session的生命周期。

智能指针管理Session

- 我们可以通过智能指针的方式管理Session类，将acceptor接收的链接保存在Session类型的智能指针里。由于智能指针会在引用计数为0时自动析构，所以为了防止其被自动回收，也方便Server管理Session，因为我们后期会做一些重连踢人等业务逻辑，我们在Server类中添加成员变量，该变量为一个map类型，key为Session的uid，value为该Session的智能指针。

```
class CServer
{
public:
    CServer(const boost::asio::io_context& io_context, short port);
    void ClearSession(std::string);
private:
    void HandleAccept(std::shared_ptr<CSession>, const boost::system::error_code&
error);
    void StartAccept();
    boost::asio::io_context& _io_context;
    short _port;
    boost::asio::ip::tcp::acceptor _acceptor;
    std::map<std::string, std::shared_ptr<CSession>> _sessions;
};
```

- 通过Server中的_sessions这个map管理链接，可以增加Session智能指针的引用计数，只有当Session从这个map中移除后，Session才会被释放。
- 所以在接收连接的逻辑里将Session放入map

```
void CServer::StartAccept() {
    std::shared_ptr<CSession> new_session = std::make_shared<CSession>(_ioc,
this);
    _acceptor.async_accept(new_session->GetSocket(),
std::bind(&CServer::HandleAccept, this, new_session, std::placeholders::_1));
}
void CServer::HandleAccept(std::shared_ptr<CSession> new_session, const
boost::system::error_code& error) {
    if (!error) {
        new_session->Start();
        //将session加入map
        _sessions.insert(std::make_pair(new_session->GetUuid(), new_session));
    }
    else {
        std::cout << "session accept failed, error is " << error.what() <<
```

```
std::endl;
    }
    StartAccept();
}
```

- StartAccept函数中虽然new_session是一个局部变量，但是我们通过bind操作，将new_session作为数值传递给bind函数，而bind函数返回的函数对象内部引用了该new_session所以引用计数增加1，这样保证了new_session不会被释放。
- 在HandleAccept函数里调用session的start函数监听对端收发数据，并将session放入map中，保证session不被自动释放。
- 此外，需要封装一个释放函数，将session从map中移除，当其引用计数为0则自动释放

```
void CServer::ClearSession(std::string& uuid) {
    _sessions.erase(uuid);
}
```

Session的uuid

- 关于session的uuid可以通过boost提供的生成唯一id的函数获得，当然你也可以自己实现雪花算法。

```
CSession(boost::asio::io_context& ioc, CServer* server) :sock(ioc),_server(server)
{
 boost::uuids::uuid a_uuid = boost::uuids::random_generator()();
 _uuid = boost::uuids::to_string(a_uuid);
}
```

- 另外我们修改Session中读写回调函数关于错误的处理，当读写出错的时候清除连接

```
void CSession::Start() {
    memset(_data, 0, max_length);
    // 错误用法，因为会生成两个shared_ptr来管理同一块内存，他们两的引用计数可能不同
  /*  sock.async_read_some(boost::asio::buffer(_data, max_length),
        std::bind(&CSession::handle_read, this, std::placeholders::_1,
std::placeholders::_2, std::shared_ptr<CSession>(this)));*/

    //正确用法，使用 std::shared_from_this()需要继承
std::enable_shared_from_this<CSession> 模板类
    //shared_from_this()内部实现是返回一个智能指针，和存在的智能指针去同步
    sock.async_read_some(boost::asio::buffer(_data, max_length),
        std::bind(&CSession::handle_read, this, std::placeholders::_1,
std::placeholders::_2, shared_from_this()));
}

void CSession::handle_read(const boost::system::error_code& error,
    size_t bytes_transferred, std::shared_ptr<CSession> _self_shared) {
```

```
    if (!error) {
        std::cout << "server receive data is " << _data << std::endl;
        boost::asio::async_write(sock, boost::asio::buffer(_data,
bytes_transferred),
            std::bind(&CSession::handle_write, this, std::placeholders::_1,
_self_shared));
    }
    else {
        std::cout << "read error" << std::endl;
        _server->ClearSession(_uuid);
    }
}
void CSession::handle_write(const boost::system::error_code& error,
std::shared_ptr<CSession> _self_shared) {
    if (!error) {
        memset(_data, 0, max_length);
        sock.async_read_some(boost::asio::buffer(_data, max_length),
std::bind(&CSession::handle_read,
            this, std::placeholders::_1, std::placeholders::_2, _self_shared));
    }
    else {
        std::cout << "write error" << std::endl;
        _server->ClearSession(_uuid);
    }
}
```

隐患1

- 正常情况下上述服务器运行不会出现问题，但是当我们像上次一样模拟，在服务器要发送数据前打个断点，此时关闭客户端，在服务器就会先触发写回调函数的错误处理，再触发读回调函数的错误处理，这样session就会两次从map中移除，因为map中key唯一，所以第二次map判断没有session的key就不做移除操作了。
- 但是这么做还是会有崩溃问题，因为第一次在session写回调函数中移除session，session的引用计数就为0了，调用了session的析构函数，这样在触发session读回调函数时此时session的内存已经被回收了自然会出现崩溃的问题。解决这个问题可以利用智能指针引用计数和bind的特性，实现一个伪闭包的机制延长session的生命周期。

如何实现伪闭包

思路：

1. 利用智能指针被复制或使用引用计数加一的原理保证内存不被回收
2. bind操作可以将值绑定在一个函数对象上生成新的函数对象，如果将智能指针作为参数绑定给函数对象，那么智能指针就以值的方式被新函数对象使用，那么智能指针的生命周期将和新生成的函数对象一致，从而达到延长生命的效果。

- 以HandleWrite举例,在**bind时传递_self_shared指针增加其引用计数**，这样_self_shared的生命周期就和async_write的第二个参数(也就是asio要求的回调函数对象)生命周期一致了。
- 除此之外，我们也要在第一次绑定读写回调函数的时候传入智能指针的值,但是要注意传入的方式，不能用两个智能指针管理同一块内存，如下用法是错误的。

- shared_ptr(this)生成的新智能指针和this之前绑定的智能指针并不共享引用计数，所以要通过shared_from_this()函数返回智能指针，该智能指针和其他管理这块内存的智能指针共享引用计数。
- shared_from_this()函数并不是session的成员函数，要使用这个函数需要继承std::enable_shared_from_this

完整的代码

实现1

```
#include "Session.h"
#include <iostream>
#include <boost/asio.hpp>

void CServer::StartAccept() {
    std::shared_ptr<CSession> new_session = std::make_shared<CSession>(_ioc,
this);
    _acceptor.async_accept(new_session->GetSocket(),
std::bind(&CServer::HandleAccept, this, new_session, std::placeholders::_1));
}
void CServer::HandleAccept(std::shared_ptr<CSession> new_session, const
boost::system::error_code& error) {
    if (!error) {
        new_session->Start();
        //将session加入map
        _sessions.insert(std::make_pair(new_session->GetUuid(), new_session));
    }
    else {
        std::cout << "session accept failed, error is " << error.what() <<
std::endl;
    }
    StartAccept();
}

void CServer::ClearSession(std::string& uuid) {
    _sessions.erase(uuid);
}

CServer::CServer(boost::asio::io_context& ioc, unsigned short port) : _ioc(ioc),
_acceptor(ioc, boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), port)) {
    std::cout << "Server start success, on port: " << port << std::endl;
    StartAccept();
}

void CSession::Start() {
    memset(_data, 0, max_length);
    // 错误用法，因为会生成两个shared_ptr来管理同一块内存，他们两的引用计数可能不同
  /*  sock.async_read_some(boost::asio::buffer(_data, max_length),
        std::bind(&CSession::handle_read, this, std::placeholders::_1,
std::placeholders::_2, std::shared_ptr<CSession>(this)));*/

    //正确用法，使用 std::shared_from_this()需要继承
std::enable_shared_from_this<CSession> 模板类
```

```
    //shared_from_this()内部实现是返回一个智能指针，和存在的智能指针去同步
    sock.async_read_some(boost::asio::buffer(_data, max_length),
        std::bind(&CSession::handle_read, this, std::placeholders::_1,
std::placeholders::_2, shared_from_this()));
}

void CSession::handle_read(const boost::system::error_code& error,
    size_t bytes_transferred, std::shared_ptr<CSession> _self_shared) {
    if (!error) {
        std::cout << "server receive data is " << _data << std::endl;
        boost::asio::async_write(sock, boost::asio::buffer(_data,
bytes_transferred),
            std::bind(&CSession::handle_write, this, std::placeholders::_1,
_self_shared));
    }
    else {
        std::cout << "read error" << std::endl;
        _server->ClearSession(_uuid);
    }
}
void CSession::handle_write(const boost::system::error_code& error,
std::shared_ptr<CSession> _self_shared) {
    if (!error) {
        memset(_data, 0, max_length);
        sock.async_read_some(boost::asio::buffer(_data, max_length),
std::bind(&CSession::handle_read,
            this, std::placeholders::_1, std::placeholders::_2, _self_shared));
    }
    else {
        std::cout << "write error" << std::endl;
        _server->ClearSession(_uuid);
    }
}
```

头文件1

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <map>

class CServer;
class CSession: public std::enable_shared_from_this<CSession>
{
public:
 CSession(boost::asio::io_context& ioc, CServer* server)
:sock(ioc),_server(server) {
  boost::uuids::uuid a_uuid = boost::uuids::random_generator()();
  _uuid = boost::uuids::to_string(a_uuid);
```

```
 }
 boost::asio::ip::tcp::socket& GetSocket() {
  return sock;
 }

 void Start();

 std::string& GetUuid() {
  return _uuid;
 }

private:
 void handle_read(const boost::system::error_code& ec,
  size_t bytes_transferred, std::shared_ptr<CSession> _self_shared);
 void handle_write(const boost::system::error_code& ec, std::shared_ptr<CSession>
_self_shared);
 boost::asio::ip::tcp::socket sock;
 enum { max_length = 1024 };
 char _data[max_length];

 // 存储唯一标识
 std::string _uuid;

 // 指向CServer的指针
 CServer* _server;
};

class CServer
{
public:
    CServer(boost::asio::io_context& ioc, unsigned short port);
    void ClearSession(std::string& uuid);
private:
    void HandleAccept(std::shared_ptr<CSession>, const boost::system::error_code&
error);
    void StartAccept();
 boost::asio::io_context& _ioc;
    unsigned short port;
    boost::asio::ip::tcp::acceptor _acceptor;
    std::map<std::string, std::shared_ptr<CSession>> _sessions;
};
```

封装服务器发送队列

- **封装发送队列来保证发送数据的有序性**

数据节点设计

```
#pragma once
#include <memory>
class MsgNode
```

```
{
public:
 //用来构造发送节点
 MsgNode(char* msg, int max_len):_total_len(max_len), _cur_len(0) {
  _data = new char[max_len];
  memcpy(_data, msg, max_len);
 }

 ~MsgNode() {
  delete[] _data;
 }
private:
 //已发送的数据
 int _cur_len;
 //总共需要发送的数据
 int _total_len;
 //存取数据的数组
 char* _data;
};
```

1. _cur_len表示数据当前已处理的长度(已经发送的数据或者已经接收的数据长度)，因为一个数据包存在未发送完或者未接收完的情况。
2. _max_len表示数据的总长度。
3. _data表示数据域，已接收或者已发送的数据都放在此空间内。

封装发送接口

- 首先在CSession类里新增一个队列存储要发送的数据，因为我们不能保证每次调用发送接口的时候上一次数据已经发送完，就要把要发送的数据放入队列中，通过回调函数不断地发送。而且我们不能保证发送的接口和回调函数的接口在一个线程，所以要增加一个锁保证发送队列安全性。
- 我们新增一个发送接口Send

```
// 发送接口
void Send(char* msg, int max_length);
private:
    // 发送队列
std::queue<std::shared_ptr<MsgNode>> _send_que;

//保证多线程的安全性
std::mutex _send_lock;
```

- 实现发送接口

```
void CSession::Send(char* msg, int max_length) {
    //pending 为false表示发送缓冲区是空的
    bool pending = false;
    std::lock_guard<std::mutex> lock(_send_lock);
```

```
    if (!_send_que.empty()) {
        pending = true;
    }
    _send_que.push(std::make_shared<MsgNode>(msg, max_length));
    if (pending) return;
    boost::asio::async_write(sock, boost::asio::buffer(msg, max_length),
        std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
shared_from_this()));
}
```

- 发送接口里判断发送队列是否为空，如果不为空说明有数据未发送完，需要将数据放入队列，然后返回。如果发送队列为空，则说明当前没有未发送完的数据，将要发送的数据放入队列并调用async_write函数发送数据。

- 回调函数的实现

```
void CSession::HandleWrite(const boost::system::error_code& error,
std::shared_ptr<CSession> _self_shared) {
    if (!error) {
        std::lock_guard<std::mutex> lock(_send_lock);
        //因为使用的是async_write，数据一定被发送玩所以可以直接出队
        _send_que.pop();
        if (!_send_que.empty()) {
            auto& msgnode = _send_que.front();
            boost::asio::async_write(sock, boost::asio::buffer(msgnode->_data,
msgnode->_total_len),
                std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
_self_shared));
        }
    }
    else {
        std::cout << "handle write failed, error is " << error.what() <<
std::endl;
        _server->ClearSession(_uuid);
    }
}
```

- 判断发送队列是否为空，为空则发送完，否则不断取出队列数据调用async_write发送，直到队列为空。

修改读回调

- 因为我们要一直监听对端发送的数据，所以要在每次收到数据后继续绑定监听事件

```
void CSession::HandleRead(const boost::system::error_code& error,
    size_t bytes_transferred, std::shared_ptr<CSession> _self_shared) {
    if (!error) {
        std::cout << "server receive data is " << _data << std::endl;
        Send(_data, bytes_transferred);
        memset(_data, 0, max_length);
```

```
        sock.async_read_some(boost::asio::buffer(_data, max_length),
            std::bind(&CSession::HandleRead, this, std::placeholders::_1,
std::placeholders::_2, _self_shared));
    }
    else {
        std::cout << "read error" << std::endl;
        _server->ClearSession(_uuid);
    }
}
```

该节总结

- 虽然实现了全双工，但是未处理粘包问题

封装后完整代码

msgNode

```
#pragma once
#include <memory>
class MsgNode
{
public:
 //用来构造发送节点
 MsgNode(char* msg, int max_len):_total_len(max_len), _cur_len(0) {
  _data = new char[max_len];
  memcpy(_data, msg, max_len);
 }

 ~MsgNode() {
  delete[] _data;
 }

 //已发送的数据
 int _cur_len;
 //总共需要发送的数据
 int _total_len;
 //存取数据的数组
 char* _data;
};
```

CSession

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include <boost/uuid/uuid_generators.hpp>
```

```
#include <boost/uuid/uuid_io.hpp>
#include <map>
#include <queue>
#include <mutex>
#include "MsgNode.h"
#include "CServer.h"

class CServer;
class CSession: public std::enable_shared_from_this<CSession>
{
public:
 CSession(boost::asio::io_context& ioc, CServer* server)
:sock(ioc),_server(server) {
  boost::uuids::uuid a_uuid = boost::uuids::random_generator()();
  _uuid = boost::uuids::to_string(a_uuid);
 }
 boost::asio::ip::tcp::socket& GetSocket() {
  return sock;
 }

 void Start();

 std::string& GetUuid() {
  return _uuid;
 }

 // 发送接口
 void Send(char* msg, int max_length);

private:
 void HandleRead(const boost::system::error_code& ec,
  size_t bytes_transferred, std::shared_ptr<CSession> _self_shared);
 void HandleWrite(const boost::system::error_code& ec, std::shared_ptr<CSession>
_self_shared);
 boost::asio::ip::tcp::socket sock;
 enum { max_length = 1024 };
 char _data[max_length];

 // 存储唯一标识
 std::string _uuid;

 // 指向CServer的指针
 CServer* _server;

 // 发送队列
 std::queue<std::shared_ptr<MsgNode>> _send_que;

 //保证多线程的安全性
 std::mutex _send_lock;
};
```

```
#include "CSession.h"
#include <iostream>
#include <boost/asio.hpp>
#include <thread>

void CSession::Send(char* msg, int max_length) {
    //pending 为false表示发送缓冲区是空的
    bool pending = false;
    std::lock_guard<std::mutex> lock(_send_lock);
    if (!_send_que.empty()) {
        pending = true;
    }
    _send_que.push(std::make_shared<MsgNode>(msg, max_length));
    if (pending) return;
    boost::asio::async_write(sock, boost::asio::buffer(msg, max_length),
        std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
shared_from_this()));
}

void CSession::Start() {
    memset(_data, 0, max_length);
    // 错误用法，因为会生成两个shared_ptr来管理同一块内存，他们两的引用计数可能不同
  /*  sock.async_read_some(boost::asio::buffer(_data, max_length),
        std::bind(&CSession::handle_read, this, std::placeholders::_1,
std::placeholders::_2, std::shared_ptr<CSession>(this)));*/

    //正确用法，使用 std::shared_from_this()需要继承
std::enable_shared_from_this<CSession> 模板类
    //shared_from_this()内部实现是返回一个智能指针，和存在的智能指针去同步
    sock.async_read_some(boost::asio::buffer(_data, max_length),
        std::bind(&CSession::HandleRead, this, std::placeholders::_1,
std::placeholders::_2, shared_from_this()));
}

void CSession::HandleRead(const boost::system::error_code& error,
    size_t bytes_transferred, std::shared_ptr<CSession> _self_shared) {
    if (!error) {
        std::cout << "server receive data is " << _data << std::endl;
        Send(_data, bytes_transferred);
        memset(_data, 0, max_length);
        sock.async_read_some(boost::asio::buffer(_data, max_length),
            std::bind(&CSession::HandleRead, this, std::placeholders::_1,
std::placeholders::_2, _self_shared));
    }
    else {
        std::cout << "read error" << std::endl;
        _server->ClearSession(_uuid);
    }
}
void CSession::HandleWrite(const boost::system::error_code& error,
std::shared_ptr<CSession> _self_shared) {
    if (!error) {
        std::lock_guard<std::mutex> lock(_send_lock);
```

```
        //因为使用的是async_write，数据一定被发送玩所以可以直接出队
        _send_que.pop();
        if (!_send_que.empty()) {
            auto& msgnode = _send_que.front();
            boost::asio::async_write(sock, boost::asio::buffer(msgnode->_data,
msgnode->_total_len),
                std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
_self_shared));
        }
    }
    else {
        std::cout << "handle write failed, error is " << error.what() <<
std::endl;
        _server->ClearSession(_uuid);
    }
}
```

CServer

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include "CSession.h"
class CSession;
class CServer
{
public:
    CServer(boost::asio::io_context& ioc, unsigned short port);
    void ClearSession(std::string& uuid);
private:
    void HandleAccept(std::shared_ptr<CSession>, const boost::system::error_code&
error);
    void StartAccept();
    boost::asio::io_context& _ioc;
    unsigned short port;
    boost::asio::ip::tcp::acceptor _acceptor;
    std::map<std::string, std::shared_ptr<CSession>> _sessions;
};
```

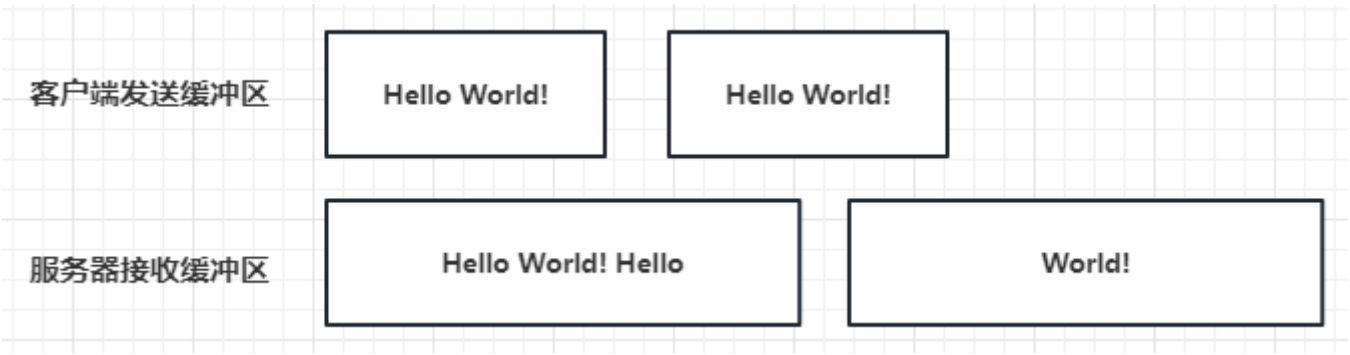
```
#include "CServer.h"

void CServer::StartAccept() {
    std::shared_ptr<CSession> new_session = std::make_shared<CSession>(_ioc,
this);
    _acceptor.async_accept(new_session->GetSocket(),
std::bind(&CServer::HandleAccept, this, new_session, std::placeholders::_1));
}
void CServer::HandleAccept(std::shared_ptr<CSession> new_session, const
```

```
boost::system::error_code& error) {
    if (!error) {
        new_session->Start();
        //将session加入map
        _sessions.insert(std::make_pair(new_session->GetUuid(), new_session));
    }
    else {
        std::cout << "session accept failed, error is " << error.what() <<
std::endl;
    }
    StartAccept();
}

void CServer::ClearSession(std::string& uuid) {
    _sessions.erase(uuid);
}

CServer::CServer(boost::asio::io_context& ioc, unsigned short port) : _ioc(ioc),
_acceptor(ioc, boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), port)) {
    std::cout << "Server start success, on port: " << port << std::endl;
    StartAccept();
}
```

处理网络粘包问题

什么是粘包

- 粘包问题是服务器收发数据常遇到的一个现象，当客户端发送多个数据包给服务器时，服务器底层的tcp接收缓冲区收到的数据为粘连在一起的，如下图所示：

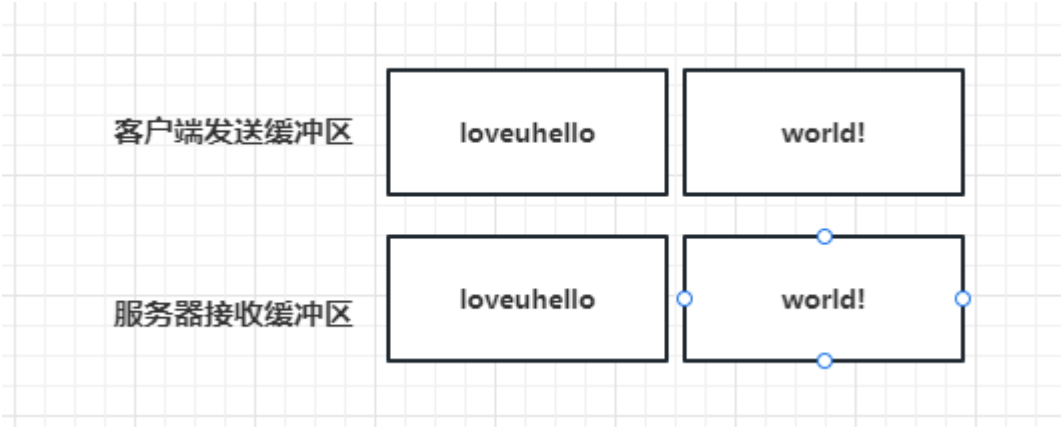


- 当客户端发送两个Hello World！给服务器，服务器TCP接收缓冲区接收了两次，一次是Hello World!Hello, 第二次是World！。

粘包原因

- 因为TCP底层通信是面向字节流的，TCP只保证发送数据的准确性和顺序性，字节流以字节为单位，客户端每次发送N个字节给服务端，N取决于当前客户端的发送缓冲区是否有数据，比如发送缓冲区总大小为10个字节，当前有5个字节数据(上次要发送的数据比如'loveu')未发送完，那么此时只有5个字节空闲空间，我们调用发送接口发送hello world！其实就是只能发送Hello给服务器，那么服务器一次性读取到的

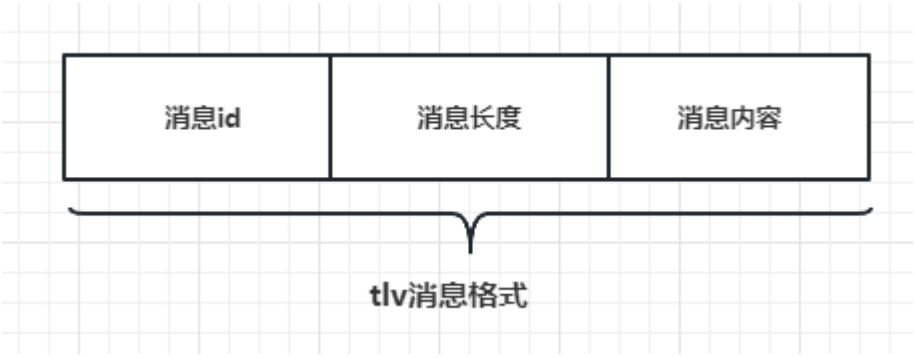
数据就很可能是loveuhello。而剩余的world！只能留给下一次发送，下一次服务器接收到的就是world！如下图：



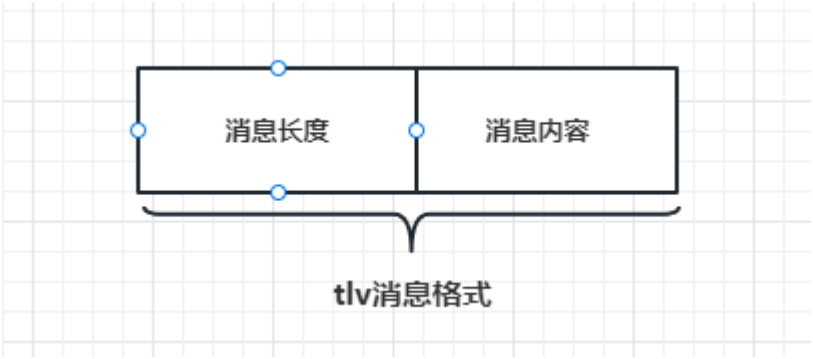
- 这是最好理解的粘包问题的产生原因。还有一些其他的原因比如
    1. 客户端的发送频率远高于服务器的接收频率，就会导致数据在服务器的tcp接收缓冲区滞留形成粘连，比如客户端1s内连续发送了两个hello world！,服务器过了2s才接收数据，那一次性读出两个hello world！。
    2. tcp底层的安全和效率机制不允许字节数特别少的小包发送频率过高，tcp会在底层累计数据长度到一定大小才一起发送，比如连续发送1字节的数据要累计到多个字节才发送，可以了解下tcp底层的Nagle算法。
    3. 再就是我们提到的最简单的情况，发送端缓冲区有上次未发送完的数据或者接收端的缓冲区里有未取出的数据导致数据粘连。

处理粘包

- 处理粘包的方式主要采用应用层定义收发包格式的方式，这个过程俗称切包处理，常用的协议被称为tlv协议(消息id+消息长度+消息内容)，如下图：



- 为保证大家容易理解，我们先简化发送的格式，格式变为消息长度+消息内容的方式，之后再完善为tlv格式。
- 简化后的结构如下图：



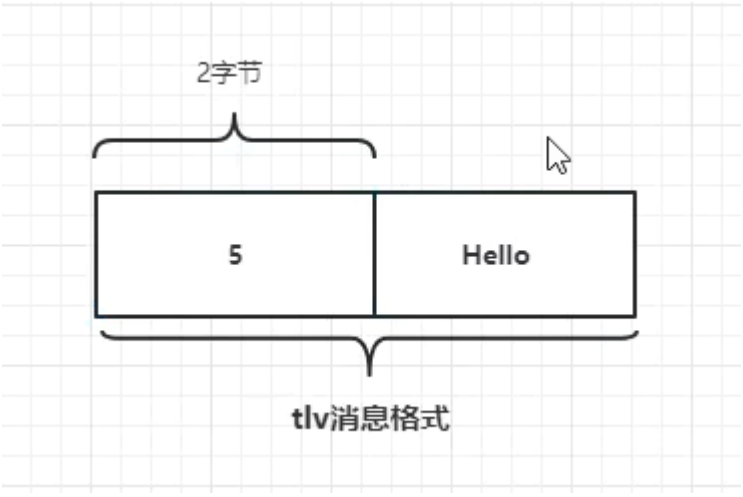
完善消息节点

```
#pragma once
#include <memory>
#include <boost/asio.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <boost/uuid/uuid_generators.hpp>

constexpr int MAX_LENGTH = 1024 * 2;
constexpr int HEAD_LENGTH = 2;

class MsgNode
{
public:
    //用来构造发送节点
    MsgNode(char* msg, short max_len):_total_len(max_len + HEAD_LENGTN),
_cur_len(0) {
        // 要多留一个空间存/0
        _data = new char[_total_len + 1];
        // 将前两个字节赋值为长度
        memcpy(_data, &max_len, HEAD_LENGTN);
        //偏移两个字节存数据长度
        memcpy(_data + HEAD_LENGTN, msg, max_len);
        _data[_total_len] = '\0';
    }

    MsgNode(short max_len):_total_len(max_len), _cur_len(0) {
        _data = new char[_total_len + 1]();
    }

    ~MsgNode() {
        delete[] _data;
    }

    void Clear() {
        ::memset(_data, 0, _total_len);
        _cur_len = 0;
    }

    //已发送的数据
    int _cur_len;
    //总共需要发送的数据
```

```
    int _total_len;
    //存取数据的数组
    char* _data;
};
```

1. 两个参数的构造函数做了完善，之前的构造函数通过消息首地址和长度构造节点数据，现在需要在构造节点的同时把长度信息也写入节点,该构造函数主要用来发送数据时构造发送信息的节点。
2. 一个参数的构造函数为较上次新增的，主要根据消息的长度构造消息节点，该构造函数主要是接收对端数据时构造接收节点调用的。
3. 新增一个Clear函数清除消息节点的数据，主要是避免多次构造节点造成开销。

- 数据这么传输



CSession类完善

- 为能够对收到的数据切包处理，需要定义一个消息接收节点，一个bool类型的变量表示头部是否解析完成，以及将处理好的头部先缓存起来的结构。

```
//收到的消息结构
std::shared_ptr<MsgNode> _recv_msg_node;
// 表示头部结构是否接收完
bool _b_head_parse;
//收到的头部结构
std::shared_ptr<MsgNode> _recv_head_node;
```

- _recv_msg_node用来存储接受的消息体信息
- _recv_head_node用来存储接收的头部信息
- _b_head_parse表示是否处理完头部信息
- 同时我们新增一个HEAD_LENGTH变量表示数据包头部的大小，修改原消息最大长度为1024*2

```
constexpr int MAX_LENGTH = 1024 * 2;
constexpr int HEAD_LENGTN = 2;
```

- 修改完后的头文件

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <map>
#include <queue>
#include <mutex>
#include "MsgNode.h"
#include "CServer.h"


class CServer;
class CSession: public std::enable_shared_from_this<CSession>
{
public:
    CSession(boost::asio::io_context& ioc, CServer* server);

    ~CSession();

    boost::asio::ip::tcp::socket& GetSocket() {
        return sock;
    }

    std::string& GetUuid() {
        return _uuid;
    }

    void Start();

    void Close();

    std::shared_ptr<CSession> SharedSelf();

    // 发送接口
    void Send(char* msg, int max_length);

private:
    void HandleRead(const boost::system::error_code& ec,
        size_t bytes_transferred, std::shared_ptr<CSession> _self_shared);
    void HandleWrite(const boost::system::error_code& ec,
std::shared_ptr<CSession> _self_shared);
    boost::asio::ip::tcp::socket sock;

    //这个为socket接收数据时候先存入的
    char _data[MAX_LENGTH];
```

```
    // 存储唯一标识
    std::string _uuid;

    // 指向CServer的指针
    CServer* _server;

    // 发送队列
    std::queue<std::shared_ptr<MsgNode>> _send_que;

    //保证多线程的安全性
    std::mutex _send_lock;

    //收到的消息结构，也就是完整的数据
    std::shared_ptr<MsgNode> _recv_msg_node;
    // 表示头部结构是否接收完
    bool _b_head_parse;
    //收到的头部结构，也就是完整的数据长度
    std::shared_ptr<MsgNode> _recv_head_node;

    // 用于标记是否关闭
    bool _b_close;
};
```

完善接收逻辑

- 需要修改HandleRead函数

```
//第二个参数为你接收到的需要读的字节数，第三个参数增加引用计数，防止未处理为就被析构以及二次析构问题
void CSession::HandleRead(const boost::system::error_code& error, size_t bytes_transferred, std::shared_ptr<CSession> shared_self) {
    if (!error) {
        //已经存入数组的字符数
        int copy_len = 0;
        // 需要读的字节数大于0
        while (bytes_transferred > 0) {
            // 头部结构未被处理完
            if (!_b_head_parse) {
                //收到的数据不足头部大小（待接收的数据字节数加上当前接收完的字节数小于头部结构（两字节））
                if (bytes_transferred + _recv_head_node->_cur_len < HEAD_LENGTH) {
                    //拷贝接收到的头部大小
                    memcpy(_recv_head_node->_data + _recv_head_node->_cur_len, _data + copy_len, bytes_transferred);
                    //更新已经接收到的字节数
                    _recv_head_node->_cur_len += bytes_transferred;
                    //将接收数组清空
                    ::memset(_data, 0, MAX_LENGTH);
                    //继续监听读事件
                    sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
```

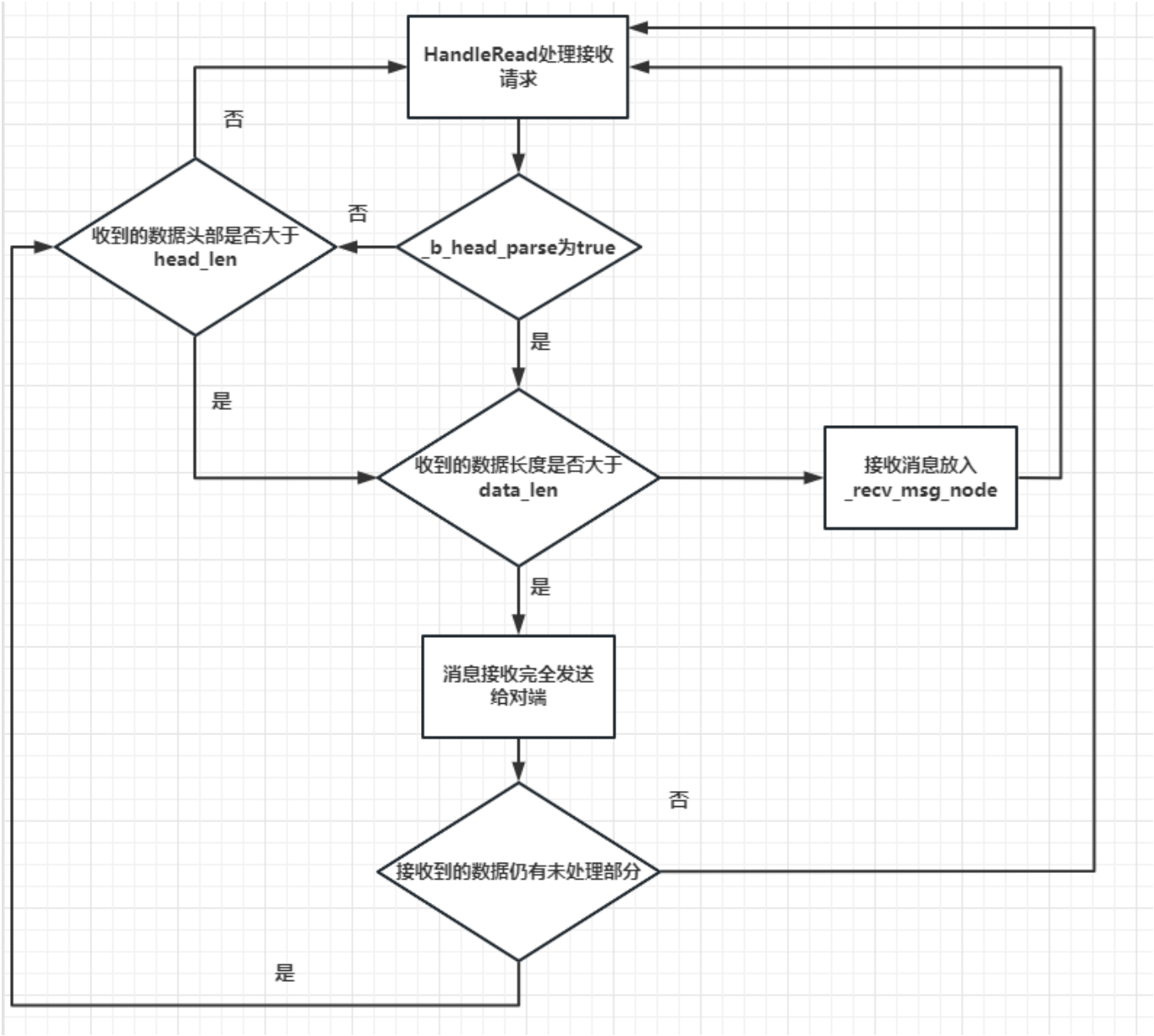
```
                        std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                    return;
                }
                //收到的数据比头部多
                //头部剩余未复制的长度，剩余可能为0 可能>0，不可能<0
                int head_remain = HEAD_LENGTH - _recv_head_node->_cur_len;
                //将头部补充完整
                memcpy(_recv_head_node->_data + _recv_head_node->_cur_len, _data +
copy_len, head_remain);
                //更新已处理的data长度和剩余未处理的长度
                copy_len += head_remain;
                bytes_transferred -= head_remain;
                //获取头部数据
                short data_len = 0;
                //把两字节的数据拷贝出来，得到实际数据的长度
                memcpy(&data_len, _recv_head_node->_data, HEAD_LENGTH);

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

                //头部长度非法
                if (data_len > MAX_LENGTH) {
                    std::cout << "invalid data length is " << data_len <<
std::endl;
                    //断开连接
                    _server->ClearSession(_uuid);
                    return;
                }
                //头部长度合法的话继续执行，接收一个short，构造一个接收节点
                _recv_msg_node = std::make_shared<MsgNode>(data_len);
                //消息的长度小于头部规定的长度，说明数据未收全，则先将部分消息放到接收节
点里
                if (bytes_transferred < data_len) {
                    //先缓存数据道接收节点里
                    memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data
+ copy_len, bytes_transferred);
                    _recv_msg_node->_cur_len += bytes_transferred;
                    ::memset(_data, 0, MAX_LENGTH);
                    //继续监听读事件
                    sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                        std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                    //头部处理完成
                    _b_head_parse = true;
                    return;
                }
                //消息的长度大于等于头部规定的长度，说明数据粘包了，并且由于进入此逻辑
时，头部节点刚被处理完，剩下的一定是接收的数据且能一次性读完
                memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data +
copy_len, data_len);

                _recv_msg_node->_cur_len += data_len;
                copy_len += data_len;
                bytes_transferred -= data_len;
```

```
                _recv_msg_node->_data[_recv_msg_node->_total_len] = '\0';
                std::cout << "receive data is " << _recv_msg_node->_data <<
std::endl;
                //此处可以调用Send发送测试
                Send(_recv_msg_node->_data, _recv_msg_node->_total_len);

                //继续轮询剩余未处理数据，也就是处理粘包的数据
                _b_head_parse = false;
                //清空接收节点
                _recv_head_node->Clear();
                //如果待接收的数据字节数小于等于0，也就是处理完了接收到的所有数据
                if (bytes_transferred <= 0) {
                    //清空接收数据的数组
                    ::memset(_data, 0, MAX_LENGTH);
                    sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                        std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                    return;
                }
                //继续循环
                continue;
            }

            //已经处理完头部，处理上次未接受完的消息数据
            //得到剩余未处理的数据总字节数
            int remain_msg = _recv_msg_node->_total_len - _recv_msg_node-
>_cur_len;
            //待接收的数据小于剩余数据总量，bytes_trabsferred就是表示_data里数据剩余未
读取的数据
            if (bytes_transferred < remain_msg) {
                //继续装数据
                memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data +
copy_len, bytes_transferred);
                //更新当前已经缓冲的数据量
                _recv_msg_node->_cur_len += bytes_transferred;
                //因为此时已经处理完_data里的数据了 所以可以直接清零，继续处理
                ::memset(_data, 0, MAX_LENGTH);
                sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                    std::bind(&CSession::HandleRead, this, std::placeholders::_1,
std::placeholders::_2, shared_self));
                return;
            }

            //待接收的数据大于等于剩余数据总量的话，也就是说此次处理后，仍有剩余数据未被
缓存，出现了粘包问题
            memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data +
copy_len, remain_msg);
            _recv_msg_node->_cur_len += remain_msg;
            bytes_transferred -= remain_msg;
            //更新_data里被读取的位置
            copy_len += remain_msg;
            //数据已经接收完整，此时节点里存的是完整数据可以发送
            _recv_msg_node->_data[_recv_msg_node->_total_len] = '\0';
            std::cout << "receive data is " << _recv_msg_node->_data << std::endl;
```

```
            //此处可以调用Send发送测试
            Send(_recv_msg_node->_data, _recv_msg_node->_total_len);
            //继续轮询剩余未处理数据
            _b_head_parse = false;
            _recv_head_node->Clear();
            if (bytes_transferred <= 0) {
                ::memset(_data, 0, MAX_LENGTH);
                sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                    std::bind(&CSession::HandleRead, this, std::placeholders::_1,
std::placeholders::_2, shared_self));
                return;
            }
            continue;
        }
    }
    else {
        std::cout << "handle read failed, error is " << error.what() << std::endl;
        //Close();
        _server->ClearSession(_uuid);
    }
}
```

1. copy_len记录的是已经处理过数据的长度，因为存在一次接收多个包的情况，所以copy_len用来做已经处理的数据长度的。
2. 首先判断_b_head_parse是否为false，如果为false则说明头部未处理，先判断接收的数据是否小于头部，如果小于头部大小则将接收到的数据放入_recv_head_node节点保存，然后继续调用读取函数监听对端发送数据。否则进入步骤3.
3. 如果收到的数据比头部多，可能是多个逻辑包，所以要做切包处理。根据之前保留在_recv_head_node的长度，计算出剩余未取出的头部长度，然后取出剩余的头部长度保存在_recv_head_node节点，然后通过memcpy方式从节点拷贝出数据写入short类型的data_len里，进而获取消息的长度。接下来继续处理包体，也就是消息体，判断接收到的数据未处理部分的长度和总共要接收的数据长度大小，如果小于总共要接受的长度，说明消息体没接收完，则将未处理部分先写入_recv_msg_node里，并且继续监听读事件。否则说明消息体接收完全，进入步骤4
4. 将消息体数据接收到_recv_msg_node中，接受完全后返回给对端。当然存在多个逻辑包粘连，此时要判断bytes_transferred是否小于等于0，如果是说明只有一个逻辑包，我们处理完了，继续监听读事件，就直接返回即可。否则说明有多个数据包粘连，就继续执行上述操作。
5. 因为存在_b_head_parse为true，也就是包头接收并处理完的情况，但是包体未接受完，再次触发HandleRead，此时要继续处理上次未接受完的消息体，大体逻辑和3，4一样。

- 以上就是处理粘包的过程，我们绘制流程图更明了一些



服务端完整代码

CSession头文件

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <map>
#include <queue>
#include <mutex>
#include "MsgNode.h"
#include "CServer.h"


class CServer;
class CSession: public std::enable_shared_from_this<CSession>
{
```

```
public:
    CSession(boost::asio::io_context& ioc, CServer* server);

    ~CSession();

    boost::asio::ip::tcp::socket& GetSocket() {
        return sock;
    }

    std::string& GetUuid() {
        return _uuid;
    }

    void Start();

    void Close();

    std::shared_ptr<CSession> SharedSelf();

    // 发送接口
    void Send(char* msg, int max_length);

private:
    void HandleRead(const boost::system::error_code& ec,
        size_t bytes_transferred, std::shared_ptr<CSession> _self_shared);
    void HandleWrite(const boost::system::error_code& ec,
std::shared_ptr<CSession> _self_shared);
    boost::asio::ip::tcp::socket sock;

    //这个为socket接收数据时候先存入的
    char _data[MAX_LENGTH];

    // 存储唯一标识
    std::string _uuid;

    // 指向CServer的指针
    CServer* _server;

    // 发送队列
    std::queue<std::shared_ptr<MsgNode>> _send_que;

    //保证多线程的安全性
    std::mutex _send_lock;

    //收到的消息结构，也就是完整的数据
    std::shared_ptr<MsgNode> _recv_msg_node;
    // 表示头部结构是否接收完
    bool _b_head_parse;
    //收到的头部结构，也就是完整的数据长度
    std::shared_ptr<MsgNode> _recv_head_node;

    // 用于标记是否关闭
    bool _b_close;
};
```

CSession 实现

```
#include "CSession.h"
#include <iostream>
#include <boost/asio.hpp>
#include <thread>

CSession::CSession(boost::asio::io_context& io_context, CServer* server) :
    sock(io_context), _server(server), _b_close(false), _b_head_parse(false) {
    boost::uuids::uuid  a_uuid = boost::uuids::random_generator()();
    _uuid = boost::uuids::to_string(a_uuid);
    _recv_head_node = std::make_shared<MsgNode>(HEAD_LENGTH);
}

CSession::~CSession() {
    std::cout << "~CSession destruct " << std::endl;
}

void CSession::Send(char* msg, int max_length) {
    //pending 为false表示发送缓冲区是空的
    bool pending = false;
    std::lock_guard<std::mutex> lock(_send_lock);
    if (!_send_que.empty()) {
        pending = true;
    }
    _send_que.push(std::make_shared<MsgNode>(msg, max_length));
    if (pending) return;
    auto& msgnode = _send_que.front();
    boost::asio::async_write(sock, boost::asio::buffer(msgnode->_data, msgnode-
>_total_len),
        std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
shared_from_this()));
}

void CSession::Start() {
    memset(_data, 0, MAX_LENGTH);
    // 错误用法，因为会生成两个shared_ptr来管理同一块内存，他们两的引用计数可能不同
  /*  sock.async_read_some(boost::asio::buffer(_data, max_length),
        std::bind(&CSession::handle_read, this, std::placeholders::_1,
std::placeholders::_2, std::shared_ptr<CSession>(this)));*/

    //正确用法，使用 std::shared_from_this()需要继承
std::enable_shared_from_this<CSession> 模板类
    //shared_from_this()内部实现是返回一个智能指针，和存在的智能指针去同步
    sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
        std::bind(&CSession::HandleRead, this, std::placeholders::_1,
std::placeholders::_2, shared_from_this()));
}

void CSession::Close() {
```

```
    //关闭套接字
    sock.close();
    _b_close = true;
}

std::shared_ptr<CSession> CSession::SharedSelf() {
    return shared_from_this();
}

//第二个参数为你接收到的需要读的字节数，第三个参数增加引用计数，防止未处理为就被析构以及二次析构问题
void CSession::HandleRead(const boost::system::error_code& error, size_t bytes_transferred, std::shared_ptr<CSession> shared_self) {
    if (!error) {
        //已经存入数组的字符数
        int copy_len = 0;
        // 需要读的字节数大于0
        while (bytes_transferred > 0) {
            // 头部结构未被处理完
            if (!_b_head_parse) {
                //收到的数据不足头部大小（待接收的数据字节数加上当前接收完的字节数小于头部结构（两字节））
                if (bytes_transferred + _recv_head_node->_cur_len < HEAD_LENGTH) {
                    //拷贝接收到的头部大小
                    memcpy(_recv_head_node->_data + _recv_head_node->_cur_len, _data + copy_len, bytes_transferred);
                    //更新已经接收到的字节数
                    _recv_head_node->_cur_len += bytes_transferred;
                    //将接收数组清空
                    ::memset(_data, 0, MAX_LENGTH);
                    //继续监听读事件
                    sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                        std::bind(&CSession::HandleRead, this, std::placeholders::_1, std::placeholders::_2, shared_self));
                    return;
                }
                //收到的数据比头部多
                //头部剩余未复制的长度，剩余可能为0 可能>0，不可能<0
                int head_remain = HEAD_LENGTH - _recv_head_node->_cur_len;
                //将头部补充完整
                memcpy(_recv_head_node->_data + _recv_head_node->_cur_len, _data + copy_len, head_remain);
                //更新已处理的data长度和剩余未处理的长度
                copy_len += head_remain;
                bytes_transferred -= head_remain;
                //获取头部数据
                short data_len = 0;
                //把两字节的数据拷贝出来，得到实际数据的长度
                memcpy(&data_len, _recv_head_node->_data, HEAD_LENGTH);

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

                //头部长度非法
                if (data_len > MAX_LENGTH) {
```

```
                    std::cout << "invalid data length is " << data_len <<
std::endl;
                    //断开连接
                    _server->ClearSession(_uuid);
                    return;
                }
                //头部长度合法的话继续执行，接收一个short，构造一个接收节点
                _recv_msg_node = std::make_shared<MsgNode>(data_len);
                //消息的长度小于头部规定的长度，说明数据未收全，则先将部分消息放到接收节
点里
                if (bytes_transferred < data_len) {
                    //先缓存数据道接收节点里
                    memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data
+ copy_len, bytes_transferred);
                    _recv_msg_node->_cur_len += bytes_transferred;
                    ::memset(_data, 0, MAX_LENGTH);
                    //继续监听读事件
                    sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                        std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                    //头部处理完成
                    _b_head_parse = true;
                    return;
                }
                //消息的长度大于等于头部规定的长度，说明数据粘包了，并且由于进入此逻辑
时，头部节点刚被处理完，剩下的一定是接收的数据且能一次性读完
                memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data +
copy_len, data_len);

                _recv_msg_node->_cur_len += data_len;
                copy_len += data_len;
                bytes_transferred -= data_len;
                _recv_msg_node->_data[_recv_msg_node->_total_len] = '\0';
                std::cout << "receive data is " << _recv_msg_node->_data <<
std::endl;
                //此处可以调用Send发送测试
                Send(_recv_msg_node->_data, _recv_msg_node->_total_len);

                //继续轮询剩余未处理数据，也就是处理粘包的数据
                _b_head_parse = false;
                //清空接收节点
                _recv_head_node->Clear();
                //如果待接收的数据字节数小于等于0，也就是处理完了接收到的所有数据
                if (bytes_transferred <= 0) {
                    //清空接收数据的数组
                    ::memset(_data, 0, MAX_LENGTH);
                    sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                        std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                    return;
                }
                //继续循环
                continue;
            }
```

```
            //已经处理完头部，处理上次未接受完的消息数据
            //得到剩余未处理的数据总字节数
            int remain_msg = _recv_msg_node->_total_len - _recv_msg_node-
>_cur_len;
            //待接收的数据小于剩余数据总量，bytes_trabsferred就是表示_data里数据剩余未
读取的数据
            if (bytes_transferred < remain_msg) {
                //继续装数据
                memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data +
copy_len, bytes_transferred);
                //更新当前已经缓冲的数据量
                _recv_msg_node->_cur_len += bytes_transferred;
                //因为此时已经处理完_data里的数据了 所以可以直接清零，继续处理
                ::memset(_data, 0, MAX_LENGTH);
                sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                    std::bind(&CSession::HandleRead, this, std::placeholders::_1,
std::placeholders::_2, shared_self));
                return;
            }

            //待接收的数据大于等于剩余数据总量的话，也就是说此次处理后，仍有剩余数据未被
缓存，出现了粘包问题
            memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data +
copy_len, remain_msg);
            _recv_msg_node->_cur_len += remain_msg;
            bytes_transferred -= remain_msg;
            //更新_data里被读取的位置
            copy_len += remain_msg;
            //数据已经接收完整，此时节点里存的是完整数据可以发送
            _recv_msg_node->_data[_recv_msg_node->_total_len] = '\0';
            std::cout << "receive data is " << _recv_msg_node->_data << std::endl;
            //此处可以调用Send发送测试
            Send(_recv_msg_node->_data, _recv_msg_node->_total_len);
            //继续轮询剩余未处理数据
            _b_head_parse = false;
            _recv_head_node->Clear();
            if (bytes_transferred <= 0) {
                ::memset(_data, 0, MAX_LENGTH);
                sock.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
                    std::bind(&CSession::HandleRead, this, std::placeholders::_1,
std::placeholders::_2, shared_self));
                return;
            }
            continue;
        }
    }
    else {
        std::cout << "handle read failed, error is " << error.what() << std::endl;
        //Close();
        _server->ClearSession(_uuid);
    }
}
void CSession::HandleWrite(const boost::system::error_code& error,
```

```
std::shared_ptr<CSession> _self_shared) {
    if (!error) {
        std::lock_guard<std::mutex> lock(_send_lock);
        std::cout << "send data " << _send_que.front()->_data + HEAD_LENGTH <<
std::endl;
        //因为使用的是async_write，数据一定被发送玩所以可以直接出队
        _send_que.pop();
        if (!_send_que.empty()) {
            auto& msgnode = _send_que.front();
            for (int i = 0; i < msgnode->_total_len; i++) std::cout << msgnode-
>_data[i + 2];
            boost::asio::async_write(sock, boost::asio::buffer(msgnode->_data,
msgnode->_total_len),
                std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
_self_shared));
        }
    }
    else {
        std::cout << "handle write failed, error is " << error.what() <<
std::endl;
        Close();
        _server->ClearSession(_uuid);
    }
}
```

CServer头文件

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include "CSession.h"
class CSession;
class CServer
{
public:
    CServer(boost::asio::io_context& ioc, unsigned short port);
    void ClearSession(std::string& uuid);
private:
    void HandleAccept(std::shared_ptr<CSession>, const boost::system::error_code&
error);
    void StartAccept();
    boost::asio::io_context& _ioc;
    unsigned short port;
    boost::asio::ip::tcp::acceptor _acceptor;
    std::map<std::string, std::shared_ptr<CSession>> _sessions;

};
```

CServer实现

```
#include "CServer.h"

void CServer::StartAccept() {
    std::shared_ptr<CSession> new_session = std::make_shared<CSession>(_ioc,
this);
    _acceptor.async_accept(new_session->GetSocket(),
std::bind(&CServer::HandleAccept, this, new_session, std::placeholders::_1));
}
void CServer::HandleAccept(std::shared_ptr<CSession> new_session, const
boost::system::error_code& error) {
    if (!error) {
        new_session->Start();
        //将session加入map
        _sessions.insert(std::make_pair(new_session->GetUuid(), new_session));
    }
    else {
        std::cout << "session accept failed, error is " << error.what() <<
std::endl;
    }
    StartAccept();
}

void CServer::ClearSession(std::string& uuid) {
    _sessions.erase(uuid);
}

CServer::CServer(boost::asio::io_context& ioc, unsigned short port) : _ioc(ioc),
_acceptor(ioc, boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), port)) {
    std::cout << "Server start success, on port: " << port << std::endl;
    StartAccept();
}
```

MsgNode

```
#pragma once
#include <memory>
#include <boost/asio.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <boost/uuid/uuid_generators.hpp>

constexpr int MAX_LENGTH = 1024 * 2;
constexpr int HEAD_LENGTH = 2;

class MsgNode
{
public:
    //用来构造发送节点
    MsgNode(char* msg, short max_len):_total_len(max_len + HEAD_LENGTH),
_cur_len(0) {
```

```
        // 要多留一个空间存/0
        _data = new char[_total_len + 1]();
        // 将前两个字节赋值为长度
        memcpy(_data, &max_len, HEAD_LENGTH);
        //偏移两个字节存数据长度
        memcpy(_data + HEAD_LENGTH, msg, max_len);
        _data[_total_len] = '\0';
    }

    //用来接收事件
    MsgNode(short max_len):_total_len(max_len), _cur_len(0) {
        _data = new char[_total_len + 1]();
    }

    ~MsgNode() {
        delete[] _data;
    }

    void Clear() {
        ::memset(_data, 0, _total_len);
        _cur_len = 0;
    }

    //已发送的数据
    short _cur_len;
    //总共需要发送的数据
    short _total_len;
    //存取数据的数组
    char* _data;
};
```

客户端修改

- 客户端的发送也要遵循先发送数据2个字节的数据长度，再发送数据消息的结构。 接收时也是先接收两个字节数据获取数据长度，再根据长度接收消息。

```
#include <iostream>
#include <boost/asio.hpp>
using namespace std;
using namespace boost::asio::ip;
const int MAX_LENGTH = 1024*2;
const int HEAD_LENGTH = 2;
int main()
{
    try {
        //创建上下文服务
        boost::asio::io_context   ioc;
        //构造endpoint
        tcp::endpoint  remote_ep(address::from_string("127.0.0.1"), 8888);
        tcp::socket  sock(ioc);
        boost::system::error_code   error = boost::asio::error::host_not_found; ;
```

```
        sock.connect(remote_ep, error);
        if (error) {
            cout << "connect failed, code is " << error.value() << " error msg is
" << error.message();
            return 0;
        }

        std::cout << "Enter message: ";
        char request[MAX_LENGTH];
        std::cin.getline(request, MAX_LENGTH);
        size_t request_length = strlen(request);
        char send_data[MAX_LENGTH] = { 0 };
        memcpy(send_data, &request_length, 2);
        memcpy(send_data + 2, request, request_length);
        boost::asio::write(sock, boost::asio::buffer(send_data,
request_length+2));

        char reply_head[HEAD_LENGTH];
        size_t reply_length =
boost::asio::read(sock,boost::asio::buffer(reply_head, HEAD_LENGTH));
        short msglen = 0;
        memcpy(&msglen, reply_head, HEAD_LENGTH);
        char msg[MAX_LENGTH] = { 0 };
        size_t  msg_length = boost::asio::read(sock,boost::asio::buffer(msg,
msglen));

        std::cout << "Reply is: ";
        std::cout.write(msg, msglen) << endl;
        std::cout << "Reply len is " << msglen;
        std::cout << "\n";
    }
    catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

粘包测试

- 为了测试粘包，需要制造粘包产生的现象，可以让客户端发送的频率高一些，服务器接收的频率低一些，这样造成前后端收发数据不一致导致多个数据包在服务器tcp缓冲区滞留产生粘包现象。
- 测试粘包之前，在服务器的CSession类里添加打印二进制数据的函数，便于查看缓冲区的数据

```
//头文件#include <iomanip>
void CSession::PrintRecvData(char* data, int length) {
    std::stringstream ss;
    std::string result = "0x";
    for (int i = 0; i < length; i++) {
        std::string hexstr;
        ss << std::hex << std::setw(2) << std::setfill('0') << int(data[i]) <<
```

```
std::endl;
        ss >> hexstr;
        result += hexstr;
    }
    std::cout << "receive raw data is : " << result << std::endl;
}
```

- 然后将这个函数放到HandleRead里，每次收到数据就调用这个函数打印接收到的最原始的数据，然后睡眠2秒再进行收发操作，用来延迟接收对端数据制造粘包，之后的逻辑不变

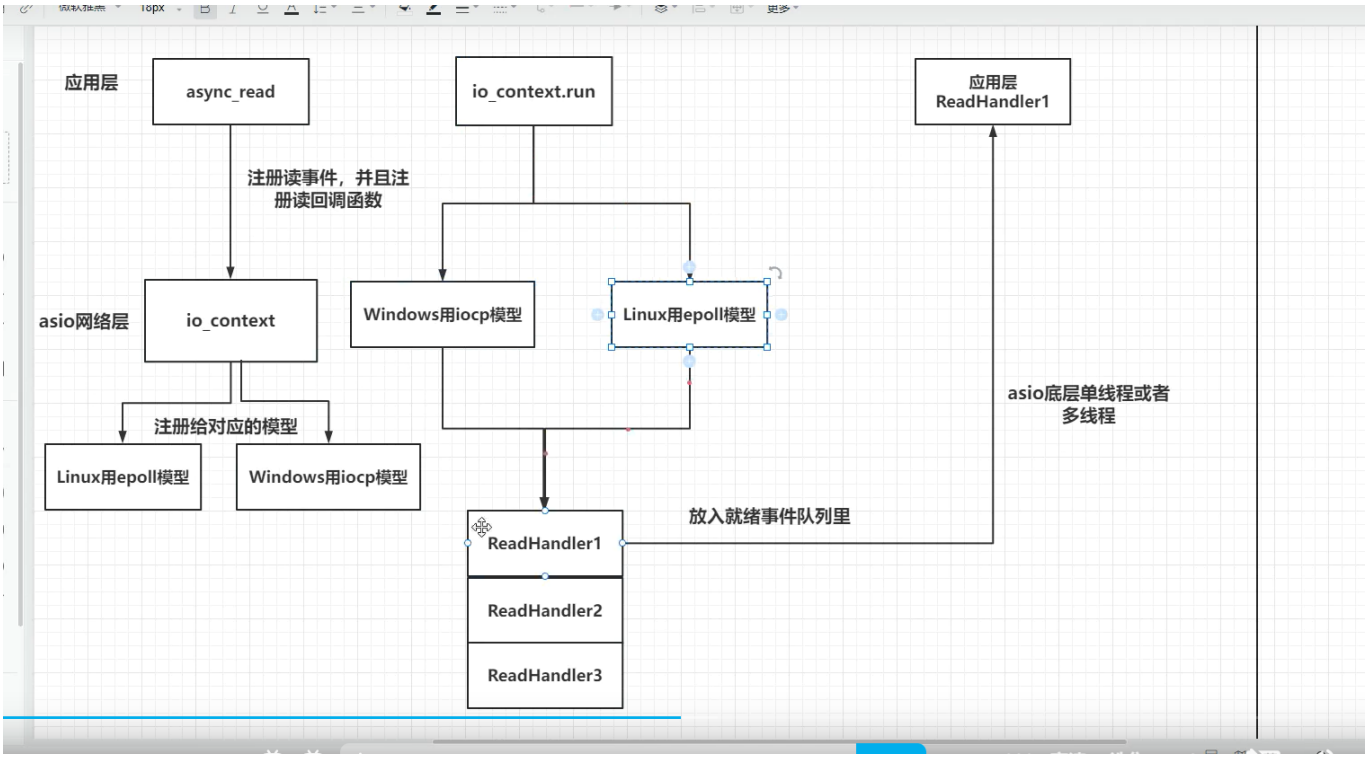
```
void CSession::HandleRead(const boost::system::error_code& error, size_t
bytes_transferred, std::shared_ptr<CSession> shared_self){
    if (!error) {
        PrintRecvData(_data, bytes_transferred);
        std::chrono::milliseconds dura(2000);
        std::this_thread::sleep_for(dura);
    }
}
```

- 修改客户端逻辑实现手收发分离

```
#include <iostream>
#include <boost/asio.hpp>
using namespace std;
using namespace boost::asio::ip;
const int MAX_LENGTH = 1024*2;
const int HEAD_LENGTH = 2;
int main()
{
    try {
        //创建上下文服务
        boost::asio::io_context   ioc;
        //构造endpoint
        tcp::endpoint  remote_ep(address::from_string("127.0.0.1"), 10086);
        tcp::socket  sock(ioc);
        boost::system::error_code   error = boost::asio::error::host_not_found; ;
        sock.connect(remote_ep, error);
        if (error) {
            cout << "connect failed, code is " << error.value() << " error msg is
" << error.message();
            return 0;
        }
        //发送数据线程
        thread send_thread([&sock] {
            for (;;) {
                //防止一直占用CPU，使用休眠
                this_thread::sleep_for(std::chrono::milliseconds(2));
                const char* request = "hello world!";
                size_t request_length = strlen(request);
```

```
                char send_data[MAX_LENGTH] = { 0 };
                memcpy(send_data, &request_length, 2);
                memcpy(send_data + 2, request, request_length);
                boost::asio::write(sock, boost::asio::buffer(send_data,
request_length + 2));
            }
            });
        //接收数据线程
        thread recv_thread([&sock] {
            for (;;) {
                this_thread::sleep_for(std::chrono::milliseconds(2));
                cout << "begin to receive..." << endl;
                char reply_head[HEAD_LENGTH];
                size_t reply_length = boost::asio::read(sock,
boost::asio::buffer(reply_head, HEAD_LENGTH));
                short msglen = 0;
                memcpy(&msglen, reply_head, HEAD_LENGTH);
                char msg[MAX_LENGTH] = { 0 };
                size_t  msg_length = boost::asio::read(sock,
boost::asio::buffer(msg, msglen));
                std::cout << "Reply is: ";
                std::cout.write(msg, msglen) << endl;
                std::cout << "Reply len is " << msglen;
                std::cout << "\n";
            }
            });
        send_thread.join();
        recv_thread.join();
    }
    catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

目前服务端通信流程图



io_context

- 里面维护了一个队列，会把注册的回调函数以及读写函数写入队列中

字节序处理和发送队列控制

字节序问题

- 在计算机网络中，由于不同的计算机使用的 CPU 架构和字节顺序可能不同，因此在传输数据时需要对数据的字节序进行统一，以保证数据能够正常传输和解析。这就是网络字节序的作用。
- 具体来说，计算机内部存储数据的方式有两种：大端序（Big-Endian）和小端序（Little-Endian）。在大端序中，高位字节存储在低地址处，而低位字节存储在高地址处；在小端序中，高位字节存储在高地址处，而低位字节存储在低地址处。
- 在网络通信过程中，通常使用的是大端序。这是因为早期的网络硬件大多采用了 Motorola 处理器，而 Motorola 处理器使用的是大端序。此外，大多数网络协议规定了网络字节序必须为大端序。
- 因此，在进行网络编程时，需要将主机字节序转换为网络字节序，也就是将数据从本地字节序转换为大端序。可以使用诸如 htonl、htons、ntohl 和 ntohs 等函数来实现字节序转换操作。
- 综上所述，网络字节序的主要作用是统一不同计算机间的数据表示方式，以保证数据在网络中的正确传输和解析。

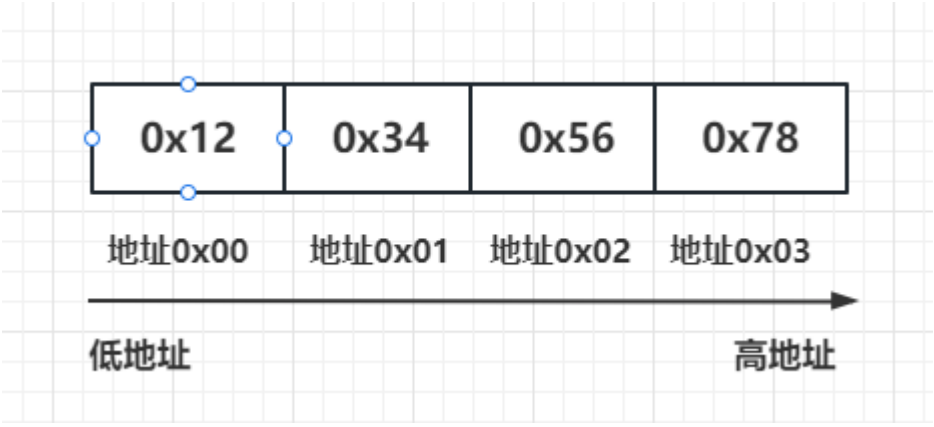
如何区分本机字节序

- 如何区分本机字节序，可以通过判断低地址存储的数据是否为低字节数据，如果是则为小端，否则为大端，下面写一段代码讲述这个逻辑

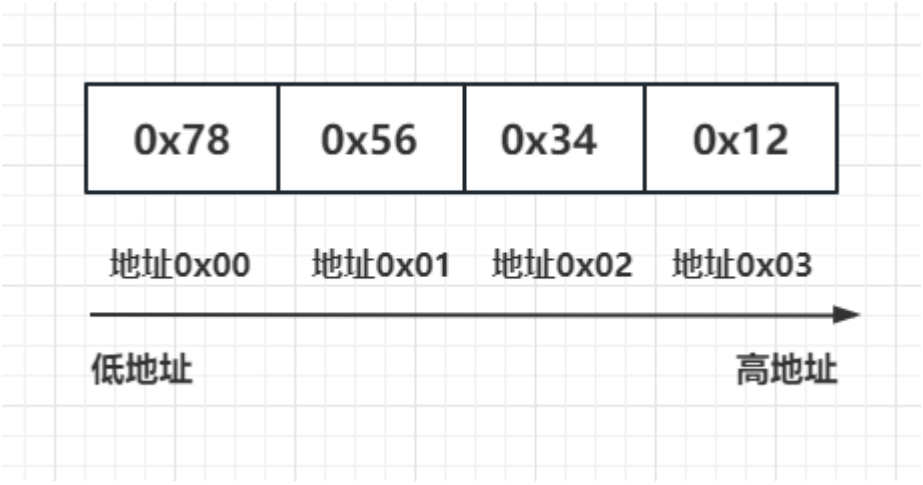
```
#include <iostream>
using namespace std;
```

```
// 判断当前系统的字节序是大端序还是小端序
bool is_big_endian() {
    int num = 1;
    if (*(char*)&num == 1) {
        // 当前系统为小端序
        return false;
    } else {
        // 当前系统为大端序
        return true;
    }
}
int main() {
    int num = 0x12345678;
    char* p = (char*)&num;
    cout << "原始数据: " << hex << num << endl;
    if (is_big_endian()) {
        cout << "当前系统为大端序" << endl;
        cout << "字节序为: ";
        for (int i = 0; i < sizeof(num); i++) {
            cout << hex << (int)*(p + i) << " ";
        }
        cout << endl;
    } else {
        cout << "当前系统为小端序" << endl;
        cout << "字节序为: ";
        for (int i = sizeof(num) - 1; i >= 0; i--) {
            cout << hex << (int)*(p + i) << " ";
        }
        cout << endl;
    }
    return 0;
}
```

- 在上述代码中，使用了一个 is_big_endian() 函数来判断当前系统的字节序是否为大端序。该函数通过创建一个整型变量 num，并将其最低位设置为 1，然后通过指针强制转换成字符指针，判断第一个字节是否为 1 来判断当前系统的字节序。
- 在 main 函数中，定义了一个整型变量 num，并将其初始化为 0x12345678。接着，使用 char* 类型的指针 p 来指向 num 的地址。然后，通过判断当前系统的字节序来输出 num 的字节序。
- 如果当前系统为大端序，则按照原始顺序输出各个字节；如果当前系统为小端序，则需要逆序输出各个字节。
- 大端模式



- 小端模式



服务器使用网络字节序

- 为保证字节序一致性，网络传输使用网络字节序，也就是大端模式。 在 boost::asio 库中，可以使用 boost::asio::detail::socket_ops::host_to_network_long() 和 boost::asio::detail::socket_ops::host_to_network_short() 函数将主机字节序转换为网络字节序。具体方法如下：

```
#include <boost/asio.hpp>
#include <iostream>
int main()
{
    uint32_t host_long_value = 0x12345678;
    uint16_t host_short_value = 0x5678;
    uint32_t network_long_value =
boost::asio::detail::socket_ops::host_to_network_long(host_long_value);
    uint16_t network_short_value =
boost::asio::detail::socket_ops::host_to_network_short(host_short_value);
    std::cout << "Host long value: 0x" << std::hex << host_long_value <<
std::endl;
    std::cout << "Network long value: 0x" << std::hex << network_long_value <<
std::endl;
    std::cout << "Host short value: 0x" << std::hex << host_short_value <<
std::endl;
    std::cout << "Network short value: 0x" << std::hex << network_short_value <<
```

```
std::endl;
    return 0;
}
```

- 上述代码中，使用了 boost::asio::detail::socket_ops::host_to_network_long() 和 boost::asio::detail::socket_ops::host_to_network_short() 函数将主机字节序转换为网络字节序。
- host_to_network_long() 函数将一个 32 位无符号整数从主机字节序转换为网络字节序，返回转换后的结果。host_to_network_short() 函数将一个 16 位无符号整数从主机字节序转换为网络字节序，返回转换后的结果。
- 在上述代码中，分别将 32 位和 16 位的主机字节序数值转换为网络字节序，并输出转换结果。需要注意的是，在使用这些函数时，应该确保输入参数和返回结果都是无符号整数类型，否则可能会出现错误。
- 同样的道理，我们只需要在服务器发送数据时，将数据长度转化为网络字节序，在接收数据时，将长度转为本机字节序。
- 在服务器的HandleRead函数里，添加对data_len的转换，将网络字节转为本地字节序

```
            //获取头部数据
            short data_len = 0;
            //把两字节的数据拷贝出来，得到实际数据的长度
            memcpy(&data_len, _recv_head_node->_data, HEAD_LENGTH);
            data_len =
boost::asio::detail::socket_ops::network_to_host_short(data_len);
            std::cout << "data_len is " << data_len << std::endl;
```

- 在服务器的发送数据时会构造消息节点，构造消息节点时，将发送长度由本地字节序转化为网络字节序

```
//用来构造发送节点
MsgNode(char* msg, short max_len):_total_len(max_len + HEAD_LENGTH), _cur_len(0) {
    // 要多留一个空间存/0
    _data = new char[_total_len + 1]();
    //转为网络字节序
    int max_len_network =
boost::asio::detail::socket_ops::host_to_network_short(max_len);
    // 将前两个字节赋值为长度
    memcpy(_data, &max_len_network, HEAD_LENGTH);
    //偏移两个字节存数据长度
    memcpy(_data + HEAD_LENGTH, msg, max_len);
    _data[_total_len] = '\0';
}
```

消息队列控制

- 发送时我们会将发送的消息放入队列里以保证发送的时序性，每个session都有一个发送队列，因为有的时候发送的频率过高会导致队列增大，所以要对队列的大小做限制，当队列大于指定数量的长度时，就

丢弃要发送的数据包，以保证消息的快速收发。

```
void CSession::Send(char* msg, int max_length) {
    std::lock_guard<std::mutex> lock(_send_lock);
    int send_que_size = _send_que.size();
    if (send_que_size > MAX_SENDQUE) {
        cout << "session: " << _uuid << " send que fulled, size is " <<
MAX_SENDQUE << endl;
        return;
    }
    _send_que.push(make_shared<MsgNode>(msg, max_length));
    if (send_que_size>0) {
        return;
    }
    auto& msgnode = _send_que.front();
    boost::asio::async_write(_socket, boost::asio::buffer(msgnode->_data, msgnode-
>_total_len),
        std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
SharedSelf()));
}
```

protobuf配置和使用

portobuf简介

- Protocol Buffers（简称 Protobuf）是一种轻便高效的序列化数据结构的协议，由 Google 开发。它可以用于将结构化数据序列化到二进制格式，并广泛用于数据存储、通信协议、配置文件等领域。
- 我们的逻辑是有类等抽象数据构成的，而tcp是面向字节流的，我们需要将类结构序列化为字符串来传输。

生成pb文件

- 要想使用protobuf的序列化功能，需要生成pb文件，pb文件包含了我们要序列化的类信息。我们先创建一个msg.proto，该文件用来定义我们要发送的类信息

```
syntax = "proto3";
//一个类
message Book
{
   string name = 1;
   int32 pages = 2;
   float price = 3;
}
```

- 这个文件定义了一个名为Book的消息类型，包含三个字段：name、pages和price。其中每个字段都有一个数字标识符，用于标识该字段在二进制流中的位置。
- 我们使用protoc.exe 基于msg.proto生成我们要用的C++类

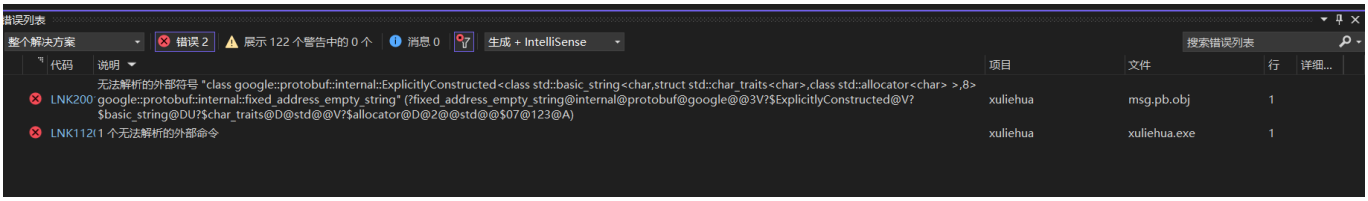
- 在proto所在文件夹执行如下命令：

```
protoc --cpp_out=. ./msg.proto
```

- —cpp_out= 表示指定要生成的pb文件所在的位置 ./msg.proto 表示msg.proto所在的位置，因为我们是在msg.proto所在文件夹中执行的protoc命令,所以是当前路径即可。
- 执行后，会看到当前目录生成了msg.pb.h和msg.pb.cc两个文件，这两个文件就是我们要用到的头文件和cpp文件。
- 我们将这两个文件添加到项目里，然后在主函数中包含msg.pb.h，做如下测试

```
#include <iostream>
#include "msg.pb.h"
int main()
{
    //客户端做的事情
    Book book;
    book.set_name("CPP programing");
    book.set_pages(100);
    book.set_price(200);
    std::string bookstr;
    //序列化写到字符串里
    book.SerializeToString(&bookstr);
    std::cout << "serialize str is " << bookstr << std::endl;

    //假设以下是服务端做的事情
    Book book2;
    //从字符串中反序列化
    book2.ParseFromString(bookstr);
    std::cout << "book2 name is " << book2.name() << " price is "
        << book2.price() << " pages is " << book2.pages() << std::endl;
    getchar();
}
```

- 测试发现报错了



- 解决办法

```
PROTOBUF_USE_DLLS
```

- 运行后又发现无法找到dll文件

- 解决办法
- 将缺失的dll文件放在.exe的同级目录下

在网络编程中的应用

- 先为服务器定义一个用来通信的proto,根据你设计发送的数据来定

```
syntax = "proto3";
message MsgData
{
   int32  id = 1;
   string data = 2;
}
```

- id代表消息id，data代表消息内容
- 我们用protoc生成对应的pb.h和pb.cc文件,方法见上
- 将proto,pb.cc,pb.h三个文件复制到我们之前的服务器项目里并且配置。
- 我们修改服务器接收数据和发送数据的逻辑
- 当服务器收到数据后，完成切包处理后，将信息反序列化为具体要使用的结构,打印相关的信息，然后再发送给客户端
- 服务端

```
  MsgData msgdata;
    std::string receive_data;
    //反序列化
    msgdata.ParseFromString(std::string(_recv_msg_node->_data, _recv_msg_node-
>_total_len));
    std::cout << "recevie msg id  is " << msgdata.id() << " msg data is " <<
msgdata.data() << endl;
    std::string return_str = "server has received msg, msg data is " +
msgdata.data();
    MsgData msgreturn;
    msgreturn.set_id(msgdata.id());
    msgreturn.set_data(return_str);
    //序列化
    msgreturn.SerializeToString(&return_str);
    Send(return_str);
```

- 客户端
- 同样的道理，客户端在发送的时候也利用protobuf进行消息的序列化，然后发给服务器

```
    MsgData msgdata;
    msgdata.set_id(1001);
    msgdata.set_data("hello world");
    std::string request;
    msgdata.SerializeToString(&request);
```

jsoncpp的使用与配置

简介

- jsoncpp 是一个 C++ JSON 库，它提供了将 JSON 数据解析为 C++ 对象、将 C++ 对象序列化为 JSON 数据的功能。它支持所有主流操作系统（包括 Windows、Linux、Mac OS X 等），并且可以与常见编译器（包括 Visual Studio、GCC 等）兼容。
- jsoncpp 库是以源代码的形式发布的，因此使用者需要自己构建和链接库文件。该库文件不依赖于第三方库，只需包含头文件即可使用。
- jsoncpp 库的特点包括：

1. 轻量级：JSON 解析器和序列化器都非常快速，不会占用太多的 CPU 和内存资源；
2. 易于使用：提供简单的 API，易于理解和使用；
3. 可靠性高：经过广泛测试，已被许多企业和开发者用于生产环境中；
4. 开源免费：遵循 MIT 许可证发布，使用和修改均免费。

- 总之，jsoncpp 是一款优秀的 C++ JSON 库，它可以帮助你轻松地处理 JSON 数据，为你的项目带来便利和高效, 一般在前后端交互中用的多

配置参考我的csdn收藏

测试

```
#include <iostream>
#include <json/json.h>
#include <json/value.h>
#include <json/reader.h>
int main()
{
    Json::Value root;
    root["id"] = 1001;
    root["data"] = "hello world";
    std::string request = root.toStyledString();
    std::cout << "request is " << request << std::endl;
    Json::Value root2;
    Json::Reader reader;
    reader.parse(request, root2);
    std::cout << "msg id is " << root2["id"] << " msg is " << root2["data"] <<
```

```
std::endl;
}
```

网络编程中的应用

- 在客户端发送数据时对发送的数据进行序列化

```
  Json::Value root;
    root["id"] = 1001;
    root["data"] = "hello world";
    std::string request = root.toStyledString();
    size_t request_length = request.length();
    char send_data[MAX_LENGTH] = { 0 };
    //转为网络字节序
    int request_host_length =
boost::asio::detail::socket_ops::host_to_network_short(request_length);
    memcpy(send_data, &request_host_length, 2);
    memcpy(send_data + 2, request.c_str(), request_length);
    boost::asio::write(sock, boost::asio::buffer(send_data, request_length + 2));
```

- 我们可以在服务器收到数据时进行json反序列化

```
Json::Reader reader;
    Json::Value root;
    reader.parse(std::string(_recv_msg_node->_data, _recv_msg_node->_total_len),
root);
    std::cout << "recevie msg id  is " << root["id"].asInt() << " msg data is "
    << root["data"].asString() << endl;
```

新版JSON库 nlohmann/json

asio粘包处理的简单方式

简单方式

- 之前我们介绍了通过async_read_some函数监听读事件，并且绑定了读事件的回调函数HandleRead

```
_socket.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, SharedSelf()));
```

- async_read_some 这个函数的特点是只要对端发数据，服务器接收到数据，即使没有收全对端发送的数据也会触发HandleRead函数，所以我们会在HandleRead回调函数里判断接收的字节数，接收的数据可能不满足头部长度，可能大于头部长度但小于消息体的长度，可能大于消息体的长度，还可能大于多个消息体的长度，所以要切包等，这些逻辑写起来很复杂，所以我们可以通过读取指定字节数，直到读完这些字节才触发回调函数，那么可以采用async_read函数，这个函数指定读取指定字节数，只有完全读完才会触发回调函数。

获取头部数据

- 我们可以读取指定的头部长度，大小为HEAD_LENGTH字节数，只有读完HEAD_LENGTH字节才触发HandleReadHead函数

```
void CSession::Start(){
    _recv_head_node->Clear();
    boost::asio::async_read(_socket, boost::asio::buffer(_recv_head_node->_data,
HEAD_LENGTH), std::bind(&CSession::HandleReadHead, this,
        std::placeholders::_1, std::placeholders::_2, SharedSelf()));
}
```

- 这样我们可以直接在HandleReadHead函数内处理头部信息

```
void CSession::HandleReadHead(const boost::system::error_code& error, size_t
bytes_transferred, std::shared_ptr<CSession> shared_self) {
    if (!error) {
        if (bytes_transferred < HEAD_LENGTH) {
            cout << "read head lenth error";
            Close();
            _server->ClearSession(_uuid);
            return;
        }
        //头部接收完，解析头部
        short data_len = 0;
        memcpy(&data_len, _recv_head_node->_data, HEAD_LENGTH);
        cout << "data_len is " << data_len << endl;
        //此处省略字节序转换
        // ...
        //头部长度非法
        if (data_len > MAX_LENGTH) {
            std::cout << "invalid data length is " << data_len << endl;
            _server->ClearSession(_uuid);
            return;
        }
        _recv_msg_node= make_shared<MsgNode>(data_len);
        boost::asio::async_read(_socket, boost::asio::buffer(_recv_msg_node-
```

```
>_data, _recv_msg_node->_total_len),
            std::bind(&CSession::HandleReadMsg, this,
            std::placeholders::_1, std::placeholders::_2, SharedSelf()));
    }
    else {
        std::cout << "handle read failed, error is " << error.what() << endl;
        Close();
        _server->ClearSession(_uuid);
    }
}
```

- 接下来根据头部内存储的消息体长度，获取指定长度的消息体数据，所以再次调用async_read，指定读取_recv_msg_node->_total_len长度，然后触发HandleReadMsg函数

获取消息体

- HandleReadMsg函数内解析消息体，解析完成后打印收到的消息，接下来继续监听读事件，监听读取指定头部大小字节，触发HandleReadHead函数， 然后再在HandleReadHead内继续监听读事件，获取消息体长度数据后触发HandleReadMsg函数，从而达到循环监听的目的。

```
void CSession::HandleReadMsg(const boost::system::error_code& error, size_t
bytes_transferred,
    std::shared_ptr<CSession> shared_self) {
    if (!error) {
        PrintRecvData(_data, bytes_transferred);
        std::chrono::milliseconds dura(2000);
        std::this_thread::sleep_for(dura);
        _recv_msg_node->_data[_recv_msg_node->_total_len] = '\0';
        cout << "receive data is " << _recv_msg_node->_data << endl;
        Send(_recv_msg_node->_data, _recv_msg_node->_total_len);
        //再次接收头部数据
        _recv_head_node->Clear();
        boost::asio::async_read(_socket, boost::asio::buffer(_recv_head_node-
>_data, HEAD_LENGTH),
            std::bind(&CSession::HandleReadHead, this, std::placeholders::_1,
std::placeholders::_2,
                SharedSelf()));
    }
    else {
        cout << "handle read msg failed,  error is " << error.what() << endl;
        Close();
        _server->ClearSession(_uuid);
    }
}
```

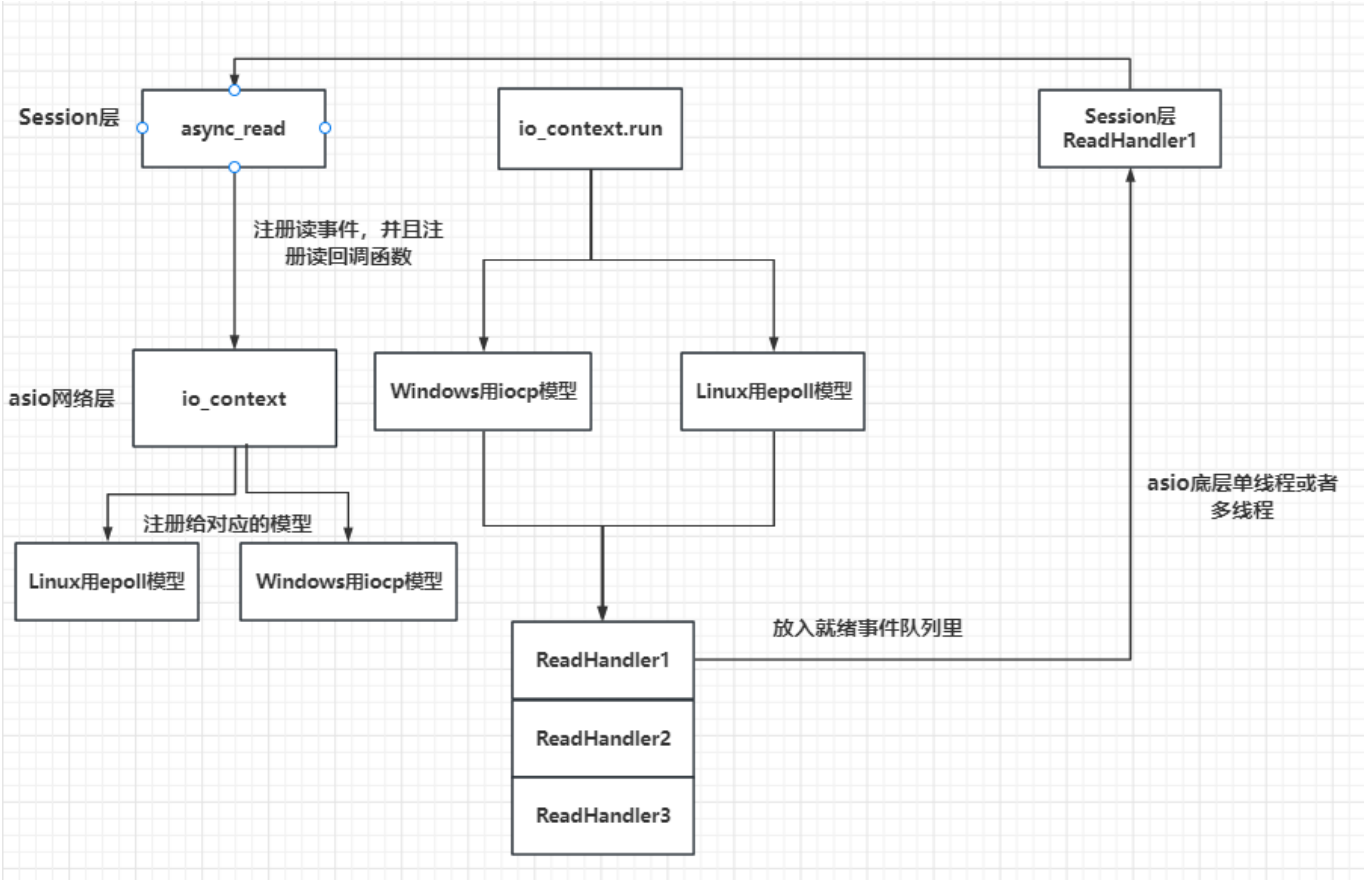
服务器逻辑层设计和消息完善

简介1

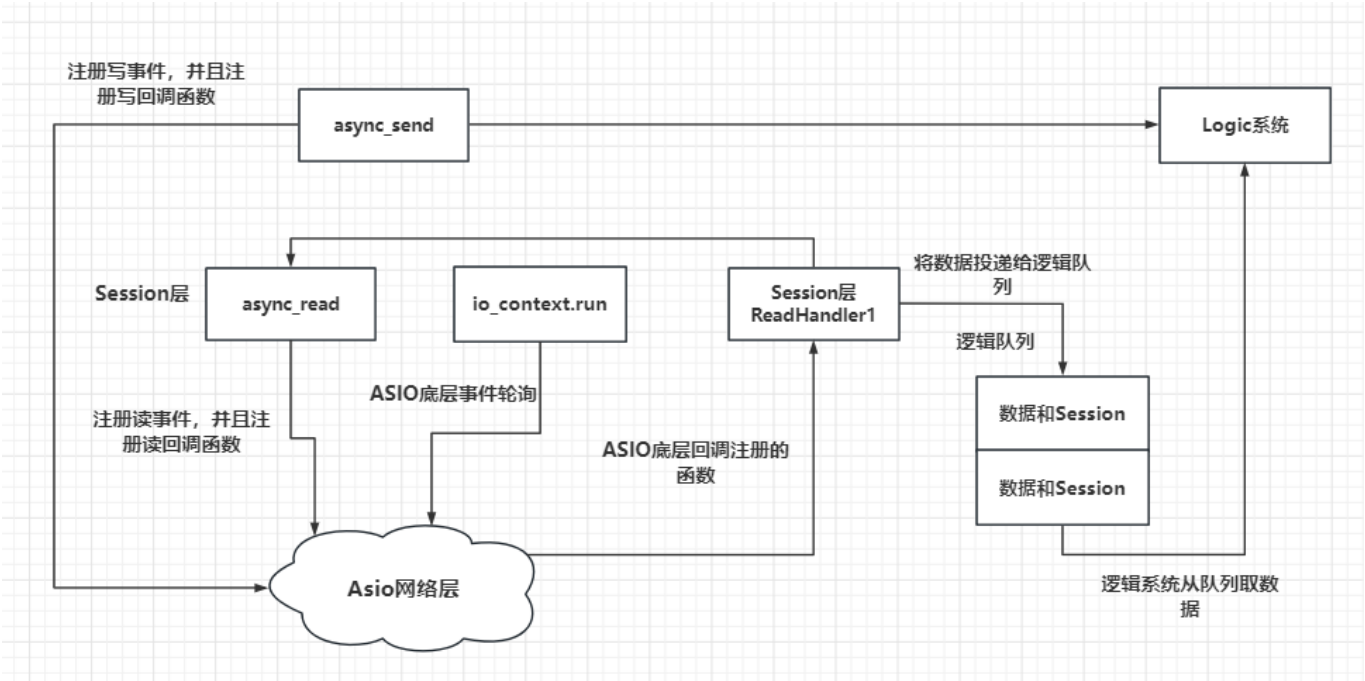
- 本文概述基于boost::asio实现的服务器逻辑层结构，并且完善之前设计的消息结构。因为为了简化粘包处理，我们简化了发送数据的结构,这次我们给出完整的消息设计，以及服务器架构设计。

服务器架构设计

- 之前我们设计了Session(会话层)，并且给大家讲述了Asio底层的通信过程，如下图

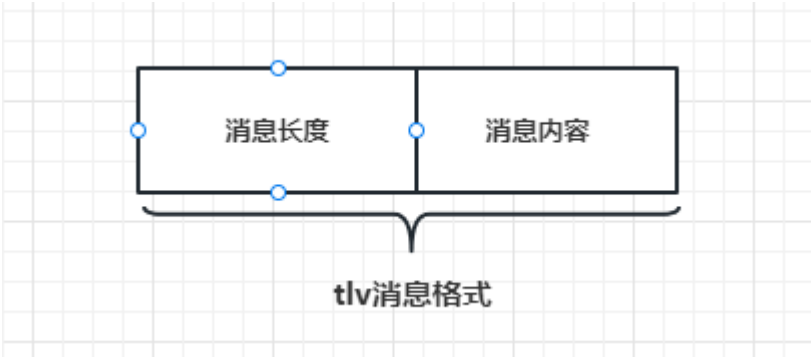


- 我们接下来要设计的服务器结构是这样的

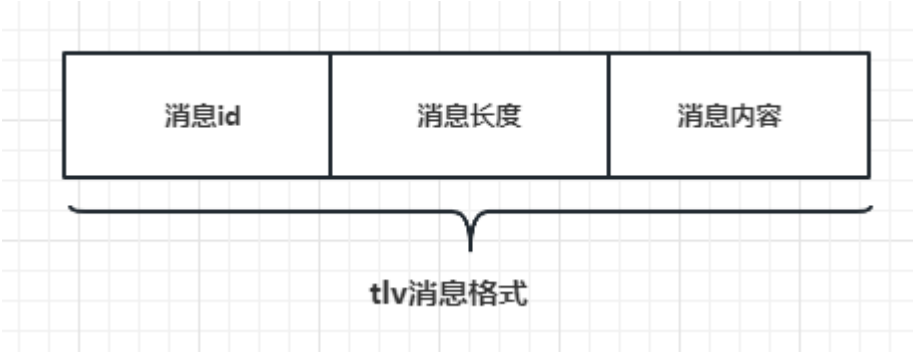


消息头完善

- 我们之前的消息头仅包含数据域的长度，但是要进行逻辑处理，就需要传递一个id字段表示要处理的消息id，当然可以不在包头传id字段，将id序列化到消息体也是可以的，但是我们为了便于处理也便于回调逻辑层对应的函数，最好是将id写入包头。
- 之前我们设计的消息结构是这样的



- 现在改为这样



- 为了减少耦合和歧义，我们重新设计消息节点。

    1. MsgNode表示消息节点的基类，头部的消息用这个结构存储。
    2. RecvNode表示接收消息的节点。
    3. SendNode表示发送消息的节点。

我们将上述结构定义在MsgNode.h中

```
#pragma once
#include <string>
#include <iostream>
#include <boost/asio.hpp>
#include "const.h"

class MsgNode
{
public:
    MsgNode(short max_len) :_total_len(max_len), _cur_len(0) {
        _data = new char[_total_len + 1];
        _data[_total_len] = '\0';
    }

    ~MsgNode() {
        std::cout << "destruct MsgNode" << std::endl;
```

```
    }

    void Clear() {
        memset(_data, 0, _total_len);
        _cur_len = 0;
    }

    short _cur_len;
    short _total_len;
    char* _data;
};

class RecvNode :public MsgNode {
public:
    RecvNode(short max_len, short msg_id);
private:
    short _msg_id;
};

class SendNode : public MsgNode {
public:
    SendNode(const char* msg, short max_len, short msg_id);
private:
    short _msg_id;
};
```

实现MsgNode

```
#include "MsgNode.h"

RecvNode::RecvNode(short max_len, short msg_id):MsgNode(max_len),_msg_id(msg_id)
{
}

SendNode::SendNode(const char* msg, short max_len, short msg_id):MsgNode(max_len +
HEAD_TOTAL_LENGTH),_msg_id(msg_id)
{
    //先发送id,转成网络字节序，网络字节序为大端
    short msg_id_network =
boost::asio::detail::socket_ops::host_to_network_short(msg_id);
    memcpy(_data, &msg_id_network, HEAD_ID_LEN);
    //在发送长度,转成网络字节序
    short msg_len_network =
boost::asio::detail::socket_ops::host_to_network_short(max_len);
    memcpy(_data + HEAD_ID_LEN, &msg_len_network, HEAD_DATA_LEN);
    //最后发送实际消息,我们传的是protobuf或json序列化的字节流所以不用转字节序
    memcpy(_data + HEAD_TOTAL_LENGTH, msg, max_len);
}
```

- SendNode发送节点构造时，先将id转为网络字节序，然后写入_data数据域。
- 然后将要发送数据的长度转为大端字节序，写入_data数据域，注意要偏移HEAD_ID_LEN长度。
- 最后将要发送的数据msg写入_data数据域，注意要偏移HEAD_ID_LEN+HEAD_DATA_LEN

Session类改写

完整的代码2

Session.h

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include <queue>
#include <mutex>
#include <boost/uuid/uuid_io.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include "MsgNode.h"
#include "const.h"
//避免循环依赖问题使用前置声明
class CServer;
class CSession: public std::enable_shared_from_this<CSession>
{
public:
    CSession(boost::asio::io_context& ioc, CServer* server);
    ~CSession();
    boost::asio::ip::tcp::socket& GetSocket();
    std::string& GetUuid();
    void Start();
    void Send(char* msg, short max_len, short msgid);
    void Send(std::string msg, short msgid);
    void Close();
    std::shared_ptr<CSession> SharedSelf();
private:
    void HandleRead(const boost::system::error_code& error, size_t
bytes_transferred, std::shared_ptr<CSession> shared_self);
    void HandleWrite(const boost::system::error_code& error,
std::shared_ptr<CSession> shared_self);
    boost::asio::ip::tcp::socket _socket;
    std::string _uuid;
    char _data[MAX_LENGTH];
    CServer* _server;
    //标志是否关闭
    bool _b_close;
    //用于保证发送信息的时序性
    std::queue<std::shared_ptr<SendNode>> _send_que;
    //对队列加锁
    std::mutex _send_lock;
    // 收到的消息结构
    std::shared_ptr<RecvNode> _recv_msg_node;
    //判断头部节点是否构造完成
    bool _b_head_parse;
```

```
    //收到的头部结构,包括消息id与消息体长度
    std::shared_ptr<MsgNode> _recv_head_node;
};
```

Session.cpp

```
#include "CSession.h"
#include "CServer.h"
#include <iostream>
#include <sstream>
#include <json/json.h>
#include <json/value.h>
#include <json/reader.h>

CSession::CSession(boost::asio::io_context& ioc, CServer* server):
    _socket(ioc), _server(server), _b_close(false), _b_head_parse(false)
{
    //创建一个没有重复的id
    boost::uuids::uuid a_uuid = boost::uuids::random_generator()();
    _uuid = boost::uuids::to_string(a_uuid);
    //不管是收到的信息还是发出的信息都有头部节点，所以放在构造函数中
    _recv_head_node = std::make_shared<MsgNode>(HEAD_TOTAL_LENGTH);
}

CSession::~CSession()
{
    std::cout << "~CSession destruct" << std::endl;
}

boost::asio::ip::tcp::socket& CSession::GetSocket()
{
    // TODO: 在此处插入 return 语句
    return _socket;
}

std::string& CSession::GetUuid()
{
    // TODO: 在此处插入 return 语句
    return _uuid;
}

//对接收数据清零，并且开始异步非阻塞读取
void CSession::Start()
{
    memset(_data, 0, MAX_LENGTH);
    _socket.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
std::bind(&CSession::HandleRead, this,
        std::placeholders::_1, std::placeholders::_2, SharedSelf()));
}
```

```
//发送C++风格的string信息
void CSession::Send(std::string msg, short msgid)
{
    //由于需要对队列操作，所以要加锁
    std::lock_guard<std::mutex> lock(_send_lock);
    //队列的长度
    int send_que_size = _send_que.size();
    //限制队列长度
    if (send_que_size > MAX_SENDQUE) {
        std::cout << "session: " << _uuid << " send que fulled, size is " <<
MAX_SENDQUE << std::endl;
        return;
    }
    //把数据插入队列，保证时序性
    _send_que.push(std::make_shared<SendNode>(msg.c_str(), msg.length(), msgid));

    //对于多个套接字，只允许当队列长度为1的时候发送数据，因为我们是在插入之前计算的队列长
度，所以是判断大于0
    if (send_que_size > 0) {
        return;
    }

    //从队列取出数据，写入发送缓冲区
    auto& msgnode = _send_que.front();
    boost::asio::async_write(_socket, boost::asio::buffer(msgnode->_data, msgnode-
>_total_len),
        std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
SharedSelf()));
}

void CSession::Send(char* msg, short max_length, short msgid) {

    std::lock_guard<std::mutex> lock(_send_lock);
    int send_que_size = _send_que.size();
    if (send_que_size > MAX_SENDQUE) {
        std::cout << "session: " << _uuid << " send que fulled, size is " <<
MAX_SENDQUE << std::endl;
        return;
    }

    _send_que.push(std::make_shared<SendNode>(msg, max_length, msgid));

    if (send_que_size > 0) {
        return;
    }

    auto& msgnode = _send_que.front();
    boost::asio::async_write(_socket, boost::asio::buffer(msgnode->_data, msgnode-
>_total_len),
        std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
SharedSelf()));
}
```

```
void CSession::Close() {
    _socket.close();
    _b_close = true;
}

//用来让指针同步引用，而不会导致指向一个内存的两个智能指针引用计数不一致问题
std::shared_ptr<CSession>CSession::SharedSelf() {
    return shared_from_this();
}

void CSession::HandleWrite(const boost::system::error_code& error,
std::shared_ptr<CSession> shared_self) {
    //增加异常处理
    try {
        if (!error) {
            std::lock_guard<std::mutex> lock(_send_lock);
            //cout << "send data " << _send_que.front()->_data+HEAD_LENGTH <<
endl;
            _send_que.pop();

            //如果队列不为空，继续发送数据
            if (!_send_que.empty()) {
                auto& msgnode = _send_que.front();
                boost::asio::async_write(_socket, boost::asio::buffer(msgnode-
>_data, msgnode->_total_len),
                    std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
shared_self));
            }
        }
        else {
            std::cout << "handle write failed, error is " << error.what() <<
std::endl;
            Close();
            _server->ClearSession(_uuid);
        }
    }
    catch (std::exception& e) {
        std::cerr << "Exception code : " << e.what() << std::endl;
    }

}

void CSession::HandleRead(const boost::system::error_code& error, size_t
bytes_transferred, std::shared_ptr<CSession> shared_self) {
    try {
        if (!error) {
            //已经移动的字符数
            int copy_len = 0;
            while (bytes_transferred > 0) {
                if (!_b_head_parse) {
                    //收到的数据不足头部大小
                    if (bytes_transferred + _recv_head_node->_cur_len <
HEAD_TOTAL_LENGTH) {
                        memcpy(_recv_head_node->_data + _recv_head_node->_cur_len,
```

```
_data + copy_len, bytes_transferred);
                        _recv_head_node->_cur_len += bytes_transferred;
                        ::memset(_data, 0, MAX_LENGTH);
                        _socket.async_read_some(boost::asio::buffer(_data,
MAX_LENGTH),
                            std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                        return;
                    }
                    //收到的数据比头部多
                    //处理头部剩余未复制的长度
                    int head_remain = HEAD_TOTAL_LENGTH - _recv_head_node-
>_cur_len;
                    memcpy(_recv_head_node->_data + _recv_head_node->_cur_len,
_data + copy_len, head_remain);
                    //更新已处理的data长度和剩余未处理的长度
                    copy_len += head_remain;
                    bytes_transferred -= head_remain;
                    //获取头部MSGID数据
                    short msg_id = 0;
                    memcpy(&msg_id, _recv_head_node->_data, HEAD_ID_LEN);
                    //网络字节序转化为本地字节序
                    msg_id =
boost::asio::detail::socket_ops::network_to_host_short(msg_id);
                    std::cout << "msg_id is " << msg_id << std::endl;
                    //id非法
                    if (msg_id > MAX_LENGTH) {
                        std::cout << "invalid msg_id is " << msg_id << std::endl;
                        _server->ClearSession(_uuid);
                        return;
                    }
                    short msg_len = 0;
                    memcpy(&msg_len, _recv_head_node->_data + HEAD_ID_LEN,
HEAD_DATA_LEN);
                    //网络字节序转化为本地字节序
                    msg_len =
boost::asio::detail::socket_ops::network_to_host_short(msg_len);
                    std::cout << "msg_len is " << msg_len << std::endl;
                    //id非法
                    if (msg_len > MAX_LENGTH) {
                        std::cout << "invalid data length is " << msg_len <<
std::endl;
                        _server->ClearSession(_uuid);
                        return;
                    }

                    _recv_msg_node = std::make_shared<RecvNode>(msg_len, msg_id);

                    //消息的长度小于头部规定的长度，说明数据未收全，则先将部分消息放到接
收节点里
                    if (bytes_transferred < msg_len) {
                        memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len,
_data + copy_len, bytes_transferred);
                        _recv_msg_node->_cur_len += bytes_transferred;
```

```
                        ::memset(_data, 0, MAX_LENGTH);
                        _socket.async_read_some(boost::asio::buffer(_data,
MAX_LENGTH),
                            std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                        //头部处理完成
                        _b_head_parse = true;
                        return;
                    }

                    //消息的长度大于等于头部规定的长度，说明数据收全，解析消息体
                    memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data
+ copy_len, msg_len);
                    _recv_msg_node->_cur_len += msg_len;
                    copy_len += msg_len;
                    bytes_transferred -= msg_len;
                    _recv_msg_node->_data[_recv_msg_node->_total_len] = '\0';
                    //cout << "receive data is " << _recv_msg_node->_data << endl;

                    //此处可以调用Send发送测试
                    Json::Reader reader;
                    Json::Value root;

                    //解析成json对象
                    reader.parse(std::string(_recv_msg_node->_data,
_recv_msg_node->_total_len), root);

                    // 将json格式转换为不同格式
                    std::cout << "recevie msg id is " << root["id"].asInt() << "
msg data is "
                        << root["data"].asString() << std::endl;

                    root["data"] = "server has received msg, msg data is " +
root["data"].asString();

                    //反序列化成字符串
                    std::string return_str = root.toStyledString();

                    //发送接受的消息
                    Send(return_str, root["id"].asInt());

                    //继续轮询剩余未处理数据
                    _b_head_parse = false;
                    _recv_head_node->Clear();
                    if (bytes_transferred <= 0) {
                        ::memset(_data, 0, MAX_LENGTH);
                        _socket.async_read_some(boost::asio::buffer(_data,
MAX_LENGTH),
                            std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                        return;
                    }
                    continue;
                }
```

```
                //已经处理完头部，处理上次未接受完的消息数据
                //接收的数据仍不足剩余未处理的
                int remain_msg = _recv_msg_node->_total_len - _recv_msg_node-
>_cur_len;
                if (bytes_transferred < remain_msg) {
                    memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data
+ copy_len, bytes_transferred);
                    _recv_msg_node->_cur_len += bytes_transferred;
                    ::memset(_data, 0, MAX_LENGTH);
                    _socket.async_read_some(boost::asio::buffer(_data,
MAX_LENGTH),
                        std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                    return;
                }
                memcpy(_recv_msg_node->_data + _recv_msg_node->_cur_len, _data +
copy_len, remain_msg);
                _recv_msg_node->_cur_len += remain_msg;
                bytes_transferred -= remain_msg;
                copy_len += remain_msg;
                _recv_msg_node->_data[_recv_msg_node->_total_len] = '\0';
                //cout << "receive data is " << _recv_msg_node->_data << endl;
                    //此处可以调用Send发送测试
                Json::Reader reader;
                Json::Value root;
                reader.parse(std::string(_recv_msg_node->_data, _recv_msg_node-
>_total_len), root);
                std::cout << "recevie msg id  is " << root["id"].asInt() << " msg
data is "
                    << root["data"].asString() << std::endl;
                root["data"] = "server has received msg, msg data is " +
root["data"].asString();
                std::string return_str = root.toStyledString();
                Send(return_str, root["id"].asInt());
                //继续轮询剩余未处理数据
                _b_head_parse = false;
                _recv_head_node->Clear();
                if (bytes_transferred <= 0) {
                    ::memset(_data, 0, MAX_LENGTH);
                    _socket.async_read_some(boost::asio::buffer(_data,
MAX_LENGTH),
                        std::bind(&CSession::HandleRead, this,
std::placeholders::_1, std::placeholders::_2, shared_self));
                    return;
                }
                continue;
            }
        }
        else {
            std::cout << "handle read failed, error is " << error.what() <<
std::endl;
            Close();
            _server->ClearSession(_uuid);
```

```
        }
    }
    catch (std::exception& e) {
        std::cout << "Exception code is " << e.what() << std::endl;

    }
}
```

单例模式实现逻辑层设计

单例模板类

- 接下来我们实现一个单例模板类，因为服务器的逻辑处理需要单例模式，后期可能还会有一些模块的设计也需要单例模式，所以先实现一个单例模板类，然后其他想实现单例类只需要继承这个模板类即可。

```
#pragma once
#include <iostream>
#include <mutex>
#include <memory>

template<typename T>
class Singleton {
public:
    static std::shared_ptr<T> GetInstance() {
        //只有第一次调用的时候会初始化，再次调用就不会初始化
        //其声明周期与进程生命周期相同
        static std::once_flag s_flag;
        std::call_once(s_flag, [&]() {
            _instance = shared_ptr<T>(new T);
            });
        return _instance;
    }

    void PrintAddress() {
        std::cout << _instance->get() << std::endl;
    }
private:
    Singleton() = default;
    Singleton(Singleton<T>& st) = delete;
    Singleton& operator=(const Singleton<T>& st) = delete;

    static std::shared_ptr<T> _instance;
protected:
    ~Singleton() {
        std::cout << "this is singleton destruct " << std::endl;
    }
};

// 因为是模板类，所以初始化的时候不能放在.cpp里，得放在.h里
// 并且又由于是模板类的静态成员，所以如果省略了 std::shared_ptr<T>，编译器将无法识别 _instance 的类型，并且无法进行类型推断。因此，在定义静态成员变量时，必须明确指明其类型。
```

```
template<class T>
std::shared_ptr<T> Singleton<T>::_instance = nullptr;
```

- 其中的细节

    1. 因为是模板类，所以初始化的时候不能放在.cpp里，得放在.h里，并且又由于是模板类的静态成员，所以如果省略了 std::shared_ptr，编译器将无法识别 _instance 的类型，并且无法进行类型推断。因此，在定义静态成员变量时，必须明确指明其类型。
    2. 得保证线程安全，可以使用C++11的once_flag 与 call_once，std::call_once 函数接受一个 std::once_flag 对象和一个函数作为参数，它会确保这个函数只被调用一次，实现原理是用加锁和一个标志位来实现,其逻辑与下属类似：

    ```
    static int* instance
    bool flag = false;
    if(flag) return instance;
    std::mutex mtx;
    std::lock_guard<std::mutex> lock(mtx);
    if(flag) return instance;
    instance = new int(2);
    flag = true;
    ```

- 单例模式模板类将无参构造，拷贝构造，拷贝赋值都设定为protected属性，其他的类无法访问，其实也可以设置为私有属性。析构函数设置为公有的，其实设置为私有的更合理一点。 Singleton有一个static类型的属性_instance, 它是我们实际要开辟类型的智能指针类型。 s_flag是函数GetInstance内的局部静态变量，该变量在函数GetInstance第一次调用时被初始化。以后无论调用多少次GetInstance s_flag都不会被重复初始化，而且s_flag存在静态区，会随着进程结束而自动释放。 call_once只会调用一次，而且是线程安全的， 其内部的原理就是调用该函数时加锁，然后设置s_flag内部的标记，设置为已经初始化，执行lambda表达式逻辑初始化智能指针，然后解锁。第二次调用GetInstance 内部还会调用call_once, 只是call_once判断s_flag已经被初始化了就不执行初始化智能指针的操作了。

LogicSystem单例类

- 我们实现逻辑系统的单例类，继承自Singleton，这样LogicSystem的构造函数和拷贝构造函数就都变为私有的了，因为基类的构造函数和拷贝构造函数都是私有的。另外LogicSystem也用了基类的成员_instance和GetInstance函数。从而达到单例效果。

```
#pragma once
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <map>
#include <functional>
#include <json/json.h>
#include <json/value.h>
#include <json/reader.h>
#include "Singleton.h"
```

```
#include "const.h"
#include "CSession.h"

typedef std::function<void(std::shared_ptr<CSession> session, const short& msg_id,
const std::string& msg_data)> FunCallBack;
class LogicSystem :public Singleton<LogicSystem> {
    //由于在Singleton的GetInstacne中调用了LogicSystem的私有构造函数
    //所以得声明友元，让对方能够调用
    friend class Singleton<LogicSystem>;
public:
    //设置为公有是为了让单例类能够正常析构
    ~LogicSystem();
    //将逻辑节点投递到消息队列里
    void PostMsgToQue(std::shared_ptr<LogicNode> msg);


private:
    LogicSystem();

    //注册功能
    void RegisterCallBacks();
    void HelloWordCallBacks(std::shared_ptr<CSession> new_session, const short&
msg_id, const std::string& msg_data);
    void DealMsg();

    //用来存储逻辑节点
    std::queue<std::shared_ptr<LogicNode>> _msg_que;

    //需要加锁
    std::mutex mtx;
    //由于有消息队列，所以需要阻塞线程，所以需要
    //条件变量来通知被阻塞的线程
    std::condition_variable _consume;

    //还需要工作线程来消费逻辑消息
    std::thread _worker_thread;

    //使用这个来标记逻辑停止
    bool _b_stop;

    //用来存储回调函数，也就是注册进来
    std::map<short, FunCallBack> _fun_callbacks;
};
```

1. FunCallBack为要注册的回调函数类型，其参数为绘画类智能指针，消息id，以及消息内容
2. _msg_que为逻辑队列
3. _mutex 为保证逻辑队列安全的互斥量
4. _consume表示消费者条件变量，用来控制当逻辑队列为空时保证线程暂时挂起等待，不要干扰其他线程。
5. _fun_callbacks表示回调函数的map，根据id查找对应的逻辑处理函数。
6. _worker_thread表示工作线程，用来从逻辑队列中取数据并执行回调函数。

7. _b_stop表示收到外部的停止信号，逻辑类要中止工作线程并优雅退出。

- LogicNode定义在CSession.h中

```
class LogicNode {
    friend class LogicSystem;
public:
    LogicNode(std::shared_ptr<CSession> session, std::shared_ptr<RecvNode>
recvnode);
private:
    std::shared_ptr<CSession> _session;
    std::shared_ptr<RecvNode> _recvnode;
};
```

- 其包含了会话类的智能指针，主要是为了实现伪闭包，防止session被释放。 其次包含了接收消息的节点类的智能指针。
- 实现如下:

```
LogicNode::LogicNode(std::shared_ptr<CSession> session, std::shared_ptr<RecvNode>
recvnode): _session(session), _recvnode(recvnode)
{

}
```

- LogicSystem的构造函数如下

```
LogicSystem::LogicSystem(): _b_stop(false) {
    //用于绑定回调函数
    RegisterCallBacks();
    //启动工作线程
    _worker_thread = std::thread(&LogicSystem::DealMsg, this);

}
```

- 构造函数中将停止信息初始化为false，注册消息处理函数并且启动了一个工作线程，工作线程执行DealMsg逻辑。 注册消息处理函数的逻辑如下:

```
void LogicSystem::RegisterCallBacks()
{
    _fun_callbacks[MSG_HELLO_WORD] = std::bind(&LogicSystem::HelloWordCallBacks,
this,
        std::placeholders::_1, std::placeholders::_2, std::placeholders::_3);
}
```

- MSG_HELLO_WORD定义在const.h中

```
enum MSG_IDS {
    MSG_HELLO_WORD = 1001
};
```

- MSG_HELLO_WORD表示消息id，HelloWordCallBack为对应的回调处理函数

```
void LogicSystem::HelloWordCallBacks(std::shared_ptr<CSession> session, const
short& msg_id, const std::string& msg_data)
{
    Json::Reader reader;
    Json::Value root;
    //将msg_data 内容系列化到root里
    reader.parse(msg_data, root);
    std::cout << "receive msg id is" << root["id"].asInt() << "msg data is "
        << root["data"].asString() << std::endl;
    root["data"] = "server has receive msg, msg data is " +
root["data"].asString();

    std::string return_str = root.toStyledString();
    //发送信息
    session->Send(return_str, root["id"].asInt());
}
```

- 在HelloWordCallBack里我们根据消息id和收到的消息，做了相应的处理并且回应给客户端。

- 工作线程的处理函数DealMsg逻辑

```
void LogicSystem::DealMsg()
{
    for (;;) {
        //配合条件变量使用，以及加锁
        std::unique_lock<std::mutex> unique_lk(mtx);
        //判断队列为空则用条件变量阻塞等待，并释放锁
        while (_msg_que.empty() && !_b_stop) {
            _consume.wait(unique_lk);
        }
        /* 也可以这么写
            _consume.wait(unique_lk, [this](){
                return !_msg_que.empty() && _b_stop
            })
        */

        //判断是否为关闭状态，把所有逻辑执行完后则退出循环
        if (_b_stop) {
            while (!_msg_que.empty()) {
                auto msg_node = _msg_que.front();
```

```
                std::cout << "recv_msg id  is " << msg_node->_recvnode->_msg_id <<
std::endl;
                auto call_back_iter = _fun_callbacks.find(msg_node->_recvnode-
>_msg_id);
                if (call_back_iter == _fun_callbacks.end()) {
                    _msg_que.pop();
                    continue;
                }
                call_back_iter->second(msg_node->_session, msg_node->_recvnode-
>_msg_id,
                    std::string(msg_node->_recvnode->_data, msg_node->_recvnode-
>_cur_len));
                _msg_que.pop();
            }
            break;
        }

        //如果没有停服，且说明队列中有数据
        auto msg_node = _msg_que.front();
        std::cout << "recv_msg id  is " << msg_node->_recvnode->_msg_id <<
std::endl;
        auto call_back_iter = _fun_callbacks.find(msg_node->_recvnode->_msg_id);
        if (call_back_iter == _fun_callbacks.end()) {
            _msg_que.pop();
            continue;
        }
        //因为second为FunCallBack其参数为三个，所以需要三个函数
        call_back_iter->second(msg_node->_session, msg_node->_recvnode->_msg_id,
            std::string(msg_node->_recvnode->_data, msg_node->_recvnode-
>_cur_len));
        _msg_que.pop();
    }
}
```

1. DealMsg逻辑中初始化了一个unique_lock，主要是用来控制队列安全，并且配合条件变量可以随时解锁。lock_guard不具备解锁功能，所以此处用unique_lock。
2. 我们判断队列为空，并且不是停止状态，就挂起线程。否则继续执行之后的逻辑，如果_b_stop为true，说明处于停服状态，则将队列中未处理的消息全部处理完然后退出循环。如果_b_stop未false，则说明没有停服，是consumer发送的激活信号激活了线程，则继续取队列中的数据处理。

- LogicSystem的析构函数需要等待工作线程处理完再退出，但是工作线程可能处于挂起状态，所以要发送一个激活信号唤醒工作线程。并且将_b_stop标记设置为true。

```
LogicSystem::~LogicSystem(){
    _b_stop = true;
    _consume.notify_one();
    _worker_thread.join();
}
```

- 因为网络层收到消息后我们需要将消息投递给逻辑队列进行处理，那么LogicSystem就要封装一个投递函数

```
void LogicSystem::PostMsgToQue(std::shared_ptr<LogicNode> msg)
{
    std::unique_lock<std::mutex> unique_lk(mtx);
    _msg_que.push(msg);
    //由0变为1则发送通知信号,通知被阻塞的线程
    if (_msg_que.size() == 1) {
        unique_lk.unlock();
        _consume.notify_one();
    }
}
```

- 在Session收到数据时这样调用

```
LogicSystem::GetInstance()->PostMsgToQue(make_shared<LogicNode>
(shared_from_this(), _recv_msg_node));
```

LogicSystem 完整代码

头文件

```
#pragma once
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <map>
#include <functional>
#include <json/json.h>
#include <json/value.h>
#include <json/reader.h>
#include "Singleton.h"
#include "const.h"
#include "CSession.h"

typedef std::function<void(std::shared_ptr<CSession> session, const short& msg_id,
const std::string& msg_data)> FunCallBack;
class LogicSystem :public Singleton<LogicSystem> {
    //由于在Singleton的GetInstacne中调用了LogicSystem的私有构造函数
    //所以得声明友元，让对方能够调用
    friend class Singleton<LogicSystem>;
public:
    //设置为公有是为了让单例类能够正常析构
    ~LogicSystem();
    //将逻辑节点投递到消息队列里
```

```
    void PostMsgToQue(std::shared_ptr<LogicNode> msg);


private:
    LogicSystem();

    //注册功能,是注册函数的统一接口
    void RegisterCallBacks();
    //具体的注册的函数
    void HelloWordCallBacks(std::shared_ptr<CSession> new_session, const short&
msg_id, const std::string& msg_data);
    void DealMsg();

    //用来存储逻辑节点
    std::queue<std::shared_ptr<LogicNode>> _msg_que;

    //需要加锁
    std::mutex mtx;
    //由于有消息队列，所以需要阻塞线程，所以需要
    //条件变量来通知被阻塞的线程
    std::condition_variable _consume;

    //还需要工作线程来消费逻辑消息
    std::thread _worker_thread;

    //使用这个来标记逻辑停止
    bool _b_stop;

    //用来存储回调函数，也就是注册进来
    std::map<short, FunCallBack> _fun_callbacks;
};
```

实现

```
#include "LogicSystem.h"

LogicSystem::~LogicSystem()
{
    _b_stop = true;
    // 可能还有被挂起的线程
    _consume.notify_one();

    _worker_thread.join();
}

void LogicSystem::PostMsgToQue(std::shared_ptr<LogicNode> msg)
{
    std::unique_lock<std::mutex> unique_lk(mtx);
    _msg_que.push(msg);
    //由0变为1则发送通知信号,通知被阻塞的线程
```

```
    if (_msg_que.size() == 1) {
        unique_lk.unlock();
        _consume.notify_one();
    }
}

void LogicSystem::RegisterCallBacks()
{
    _fun_callbacks[MSG_HELLO_WORD] = std::bind(&LogicSystem::HelloWordCallBacks,
this,
        std::placeholders::_1, std::placeholders::_2, std::placeholders::_3);
}

void LogicSystem::HelloWordCallBacks(std::shared_ptr<CSession> session, const
short& msg_id, const std::string& msg_data)
{
    Json::Reader reader;
    Json::Value root;
    //将msg_data 内容系列化到root里
    reader.parse(msg_data, root);
    std::cout << "receive msg id is" << root["id"].asInt() << "msg data is "
        << root["data"].asString() << std::endl;
    root["data"] = "server has receive msg, msg data is " +
root["data"].asString();

    std::string return_str = root.toStyledString();
    //发送信息
    session->Send(return_str, root["id"].asInt());
}

void LogicSystem::DealMsg()
{
    for (;;) {
        //配合条件变量使用，以及加锁
        std::unique_lock<std::mutex> unique_lk(mtx);
        //判断队列为空则用条件变量阻塞等待，并释放锁
        while (_msg_que.empty() && !_b_stop) {
            _consume.wait(unique_lk);
        }
        /* 也可以这么写
            _consume.wait(unique_lk, [this](){
                return !_msg_que.empty() && _b_stop
            })
        */

        //判断是否为关闭状态，把所有逻辑执行完后则退出循环
        if (_b_stop) {
            while (!_msg_que.empty()) {
                auto msg_node = _msg_que.front();
                std::cout << "recv_msg id  is " << msg_node->_recvnode->_msg_id <<
std::endl;
                auto call_back_iter = _fun_callbacks.find(msg_node->_recvnode-
>_msg_id);
                if (call_back_iter == _fun_callbacks.end()) {
```

```
                    _msg_que.pop();
                    continue;
                }
                call_back_iter->second(msg_node->_session, msg_node->_recvnode->_msg_id,
                    std::string(msg_node->_recvnode->_data, msg_node->_recvnode->_cur_len));
                _msg_que.pop();
            }
            break;
        }

        //如果没有停服，且说明队列中有数据
        auto msg_node = _msg_que.front();
        std::cout << "recv_msg id  is " << msg_node->_recvnode->_msg_id << std::endl;
        auto call_back_iter = _fun_callbacks.find(msg_node->_recvnode->_msg_id);
        if (call_back_iter == _fun_callbacks.end()) {
            _msg_que.pop();
            continue;
        }
        //因为second为FunCallBack其参数为三个，所以需要三个函数
        call_back_iter->second(msg_node->_session, msg_node->_recvnode->_msg_id,
            std::string(msg_node->_recvnode->_data, msg_node->_recvnode->_cur_len));
        _msg_que.pop();
    }
}

LogicSystem::LogicSystem(): _b_stop(false) {
    //用于绑定回调函数
    RegisterCallBacks();
    //启动工作线程
    _worker_thread = std::thread(&LogicSystem::DealMsg, this);

}
```

服务器优雅退出

退出方式1：开辟线程，让服务器运行在线程中并接受退出信号退出

```
#include "CServer.h"
#include <csignal>
#include <thread>
#include <mutex>
#include <condition_variable>
using namespace std;
bool bstop = false;
std::condition_variable cond_quit;
std::mutex mutex_quit;
```

```
void signal_handler(int sig) {
    if (sig == SIGINT || sig == SIGTERM) {
        //访问共享数据接得加锁
        std::unique_lock<std::mutex> lock(mutex_quit);
        bstop = true;
        cond_quit.notify_one();
    }
}

int main()
{
    try {
        boost::asio::io_context ioc;
        std::thread network_thread([&ioc]() {
            CServer server(ioc, 8888);
            ioc.run();
            });

        signal(SIGINT, signal_handler);
        signal(SIGTERM, signal_handler);

        std::unique_lock<std::mutex> lock(mutex_quit);
        //阻塞的时候自动解锁
        cond_quit.wait(lock, []() { {
                return bstop;
            }});

        ioc.stop();
        network_thread.join();
    }
    catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

退出方式2：使用asio底层异步等待函数

```
#include "CServer.h"
#include <csignal>
#include <thread>
#include <mutex>
#include <condition_variable>
using namespace std;

int main()
{
    try {
        boost::asio::io_context ioc;
        boost::asio::signal_set signals(ioc, SIGINT, SIGTERM);
```

```
        //参数列表有两个参数表示接收到的信号
        signals.async_wait([&ioc](auto, auto) {
            ioc.stop();
            });
        CServer server(ioc, 8888);
        ioc.run();
    }
    catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

总结

- 两种方式都对服务器进行了优雅的退出

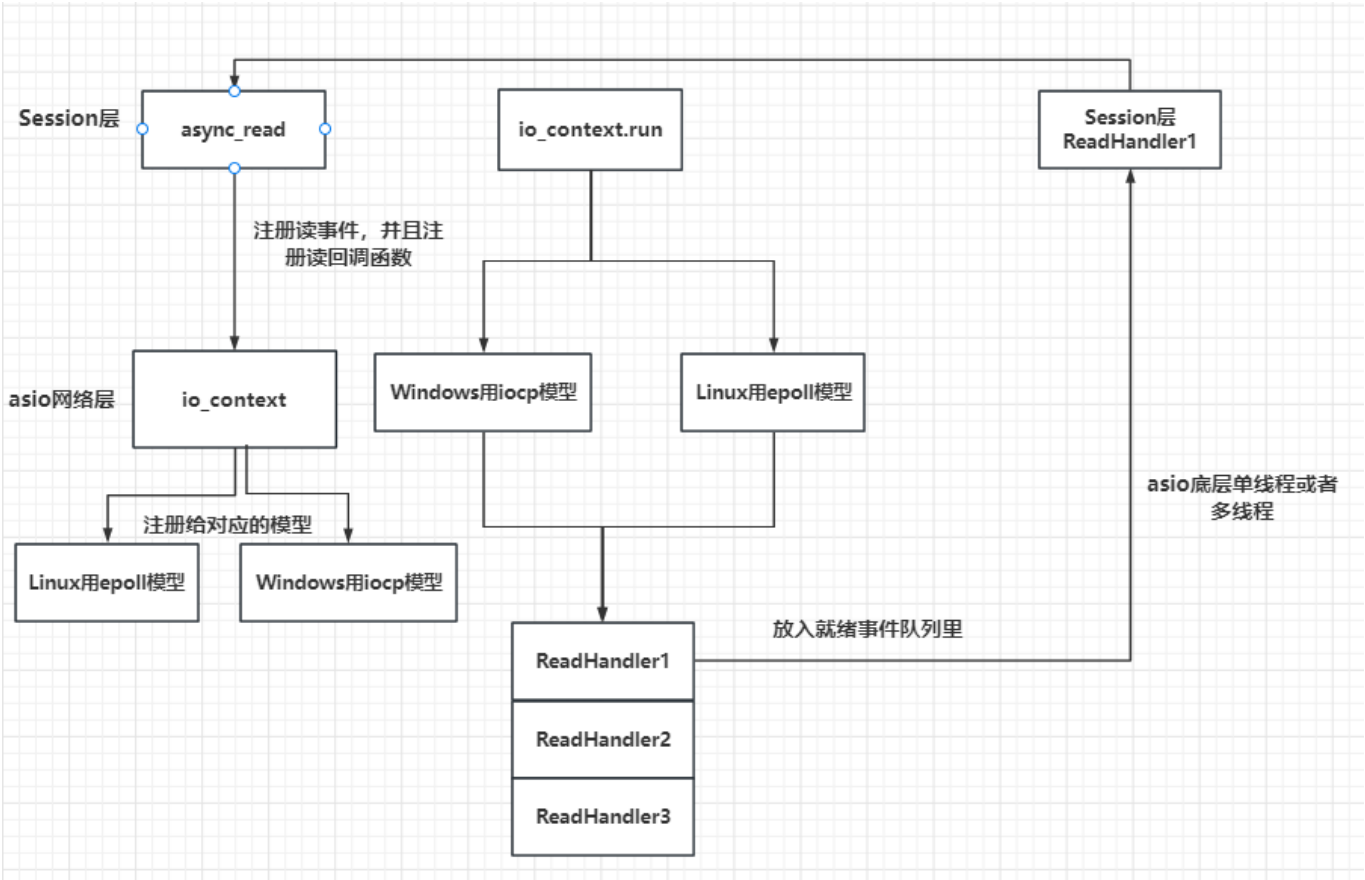
asio多线程模型IOServicePool

简介

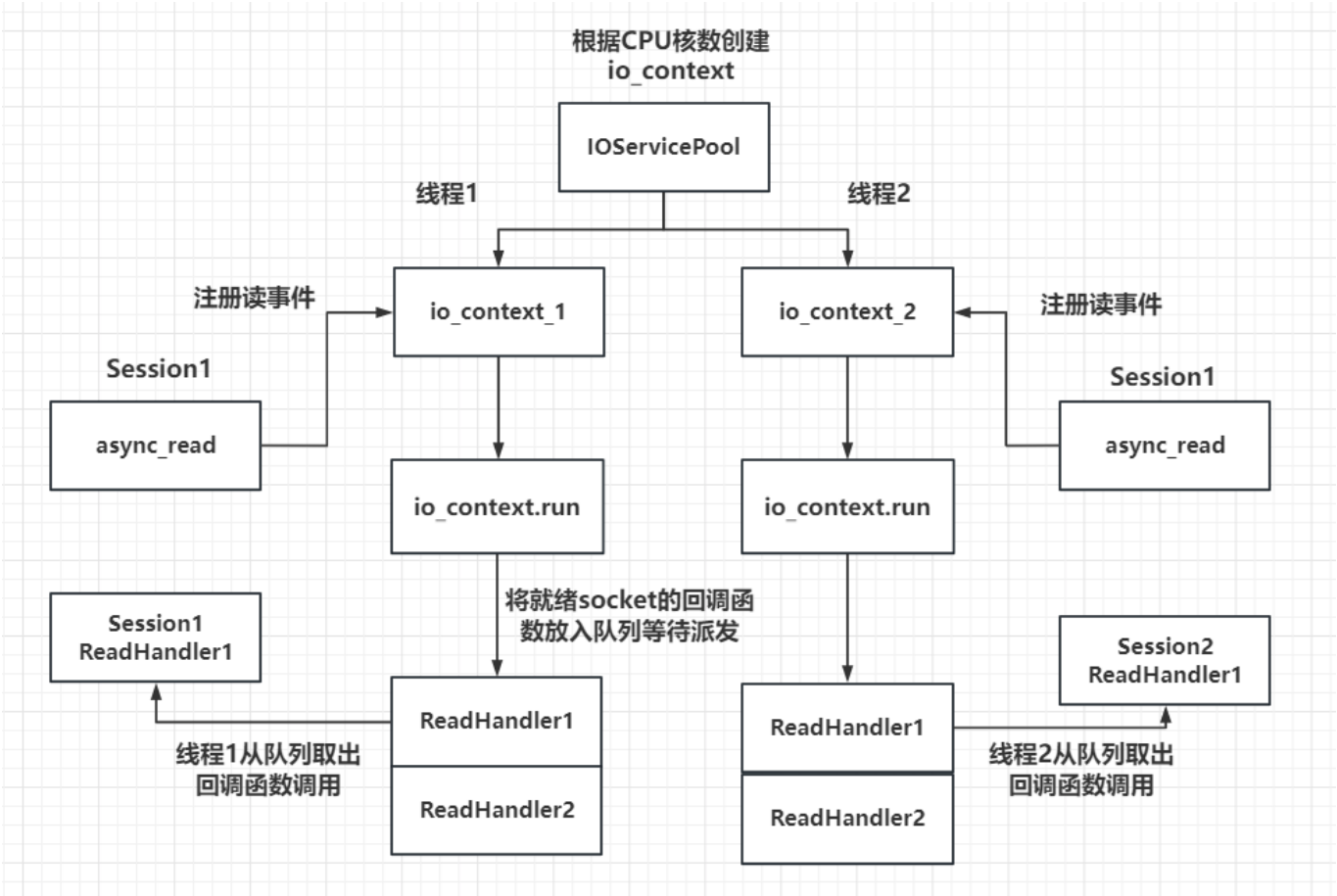
- 前面的设计，我们对asio的使用都是单线程模式，为了提升网络io并发处理的效率，这一次我们设计多线程模式下asio的使用方式。总体来说asio有两个多线程模型，第一个是启动多个线程，每个线程管理一个iocontext。第二种是只启动一个iocontext，被多个线程共享，后面的文章会对比两个模式的区别，**这里先介绍第一种模式，多个线程，每个线程管理独立的iocontext服务。**

单线程和多线程对比

- 之前的单线程模式图如下



- 我们设计的IOServicePool类型的多线程模型如下：



- IOServicePool多线程模式特点

1. 每一个io_context跑在不同的线程里，所以同一个socket会被注册在同一个io_context里，它的回调函数也会被单独的一个线程回调，那么对于同一个socket，他的回调函数每次触发都是在同一个线程里，就不会有线程安全问题，网络io层面上的并发是线程安全的。
2. 但是对于不同的socket，回调函数的触发可能是同一个线程(两个socket被分配到同一个io_context)，也可能不是同一个线程(两个socket被分配到不同的io_context里)。所以如果两个socket对应的上层逻辑处理，如果有交互或者访问共享区，会存在线程安全问题。比如socket1代表玩家1，socket2代表玩家2，玩家1和玩家2在逻辑层存在交互，比如两个玩家都在做工会任务，他们属于同一个工会，工会积分的增加就是共享区的数据，需要保证线程安全。可以通过加锁或者逻辑队列的方式解决安全问题，我们目前采取了后者。
3. 多线程相比单线程，极大的提高了并发能力，因为单线程仅有一个io_context服务用来监听读写事件，就绪后回调函数在一个线程里串行调用, 如果一个回调函数的调用时间较长肯定会影响后续的函数调用，毕竟是穿行调用。而采用多线程方式，可以在一定程度上减少前一个逻辑调用影响下一个调用的情况，比如两个socket被部署到不同的iocontext上，但是当两个socket部署到同一个iocontext上时仍然存在调用时间影响的问题。不过我们已经通过逻辑队列的方式将网络线程和逻辑线程解耦合了，不会出现前一个调用时间影响下一个回调触发的问题。

IOServicePool实现

- 在使用拷贝构造的时候参数列表的&是必须的，原因：**这不仅仅只是为了减少一次构造成本，更重要是为了避免递归构造**

```
Pool(const Pool x);

Pool p1 = p2;
```

- 以上就会有循环构造的情况，p2传给参数列表的时候又要调用拷贝构造 Pool x = p2, 之后又要进行拷贝构造，如此循环
- 同时一定要判断拷贝的是不是自己这种情况
- IOServicePool本质上是一个线程池，基本功能就是根据构造函数传入的数量创建n个线程和iocontext，然后每个线程跑一个iocontext，这样就可以并发处理不同iocontext读写事件了。

IOServicePool的声明:

```
#pragma once
#include "CSession.h"
#include <boost/asio.hpp>
#include <map>
#include <mutex>
class CServer
{
public:
    CServer(boost::asio::io_context& ioc, unsigned short port);
    ~CServer();
    void ClearSession(std::string& uuid);
private:
```

```
    //处理连接的回调函数
    void HandleAccept(std::shared_ptr<CSession> session, const
boost::system::error_code& error);
    //开始建立连接的函数
    void StartAccept();
    boost::asio::ip::tcp::acceptor _acceptor;
    boost::asio::io_context& _ioc;
    unsigned short _port;
    //将连接加入map，增加引用计数，防止未执行完逻辑就被析构
    std::map<std::string, std::shared_ptr<CSession>> _sessions;
    std::mutex mtx;
};
```

1. _ioServices是一个IOService的vector变量，用来存储初始化的多个IOService。
2. WorkPtr是boost::asio::io_context::work类型的unique指针。 在实际使用中，我们通常会将一些异步操作提交给io_context进行处理，然后该操作会被异步执行，而不会立即返回结果。如果没有其他任务需要执行，那么io_context就会停止工作，导致所有正在进行的异步操作都被取消。这时，我们需要使用boost::asio::io_context::work对象来防止io_context停止工作。

- boost::asio::io_context::work的作用是持有一个指向io_context的引用，并通过创建一个“工作”项来保证io_context不会停止工作，直到work对象被销毁或者调用reset()方法为止。当所有异步操作完成后，程序可以使用work.reset()方法来释放io_context，从而让其正常退出。

3. _threads是一个线程vector,管理我们开辟的所有线程。
4. _nextIOService是一个轮询索引，我们用最简单的轮询算法为每个新创建的连接分配io_context.
5. 因为IOServicePool不允许被copy构造，所以我们将其拷贝构造和拷贝复制函数置为delete

实现

```
#include "IOServicePool.h"

IOServicePool::~IOServicePool()
{
    std::cout << "destruct IOServicePool" << std::endl;
}

void IOServicePool::Stop()
{
    for (auto& work : _works) {
        //因为仅仅执行work.reset并不能让iocontext从run的状态中退出
        //当iocontext已经绑定了读或写的监听事件后，还需要手动stop该服务。
        //停止与work相关的上下文
        work->get_io_context().stop();
        //重新设成空
        work.reset();
    }

    for (auto& td : _threads) {
        if (td.joinable()) {
```

```
            td.join();
        }
    }
}

boost::asio::io_context& IOServicePool::GetIOService()
{
    auto& service =  _ioService[_nextIOService++];
    if (_nextIOService == _ioService.size()) {
        _nextIOService = 0;
    }
    return service;
}

IOServicePool::IOServicePool(std::size_t size) : _works(size),
_ioService(size),_nextIOService(0)
{
    //初始化works
    for (std::size_t i = 0; i < size; i++) {
        _works[i] = std::unique_ptr<Work>(new Work(_ioService[i]));
    }

    //遍历多个ioservice，创建多个线程，每个线程内部启动ioservice
    for (std::size_t i = 0; i < _ioService.size(); i++) {
        _threads.emplace_back([this, i]() {
            _ioService[i].run();
        });
    }
}
```

优雅退出

- IOServicePool多线程服务器退出时，需要捕获退出信号如SIGINT,SIGTERM等，将退出信号和一个iocontext绑定，当收到退出信号时，我们将IOServicePool停止，并且停止iocontext即可。

```
int main()
{
    try {
        auto pool = AsioIOServicePool::GetInstance();
        boost::asio::io_context  io_context;
        boost::asio::signal_set signals(io_context, SIGINT, SIGTERM);
        signals.async_wait([&io_context,pool](auto, auto) {
            io_context.stop();
            pool->Stop();
            });
        CServer s(io_context, 10086);
        io_context.run();
    }
    catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << endl;
```

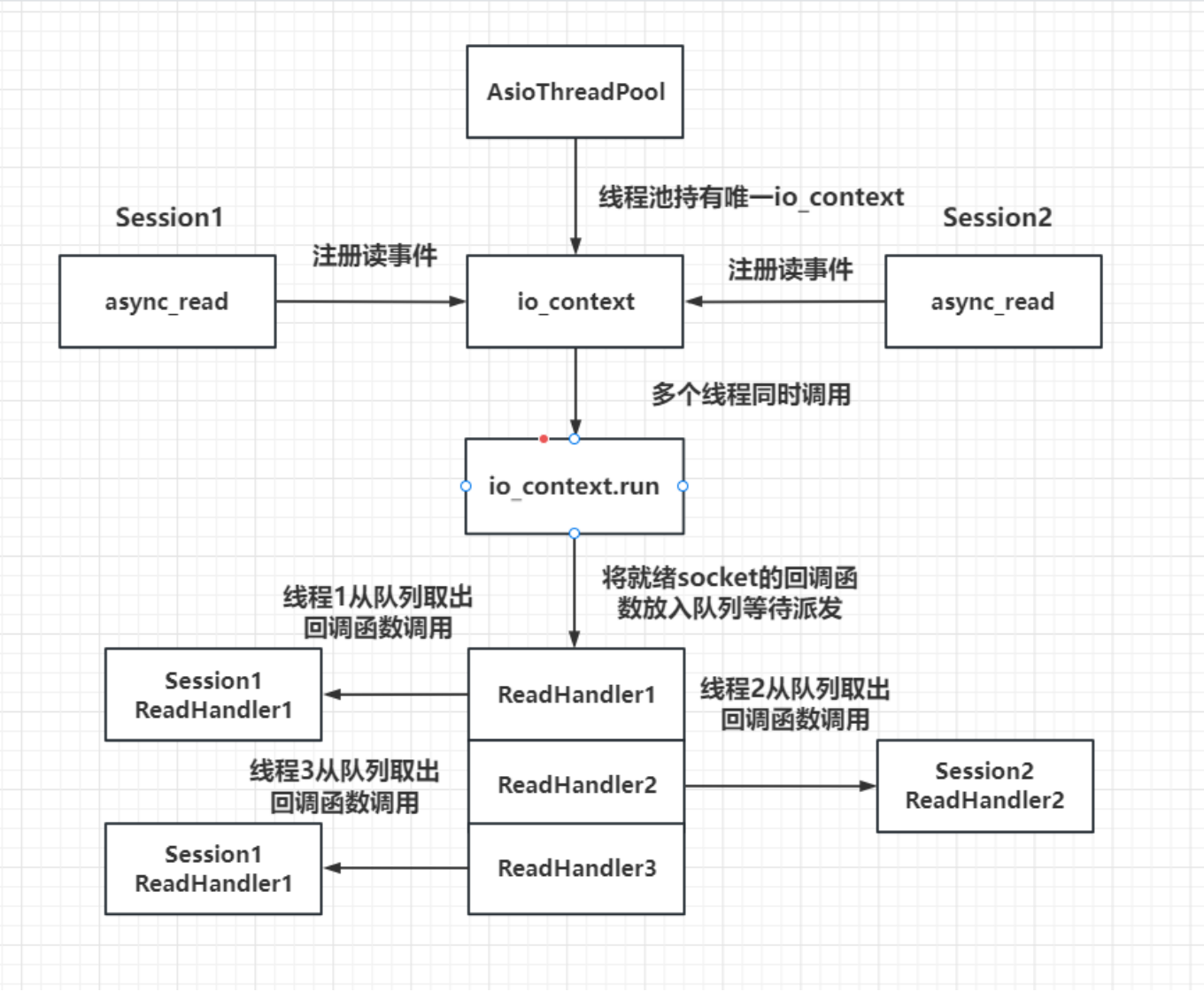
```
    }
}
```

asio多线程模式IOThreadPool

- 今天给大家介绍asio多线程模式的第二种，之前我们介绍了IOServicePool的方式，一个IOServicePool开启n个线程和n个iocontext，每个线程内独立运行iocontext, 各个iocontext监听各自绑定的socket是否就绪，如果就绪就在各自线程里触发回调函数。为避免线程安全问题，我们将网络数据封装为逻辑包投递给逻辑系统，逻辑系统有一个单独线程处理，这样将网络IO和逻辑处理解耦合，极大的提高了服务器IO层面的吞吐率。**这一次介绍的另一种多线程模式IOThreadPool，我们只初始化一个iocontext用来监听服务器的读写事件，包括新连接到来的监听也用这个iocontext。只是我们让iocontext.run在多个线程中调用，这样回调函数就会被不同的线程触发，从这个角度看回调函数被并发调用了。**

结构图

- 线程池模式的多线程模型调度结构图,如下



先实现IOThreadPool

IOThreadPool 头文件

```
#pragma once
#include "Singleton.h"
#include <boost/asio.hpp>
#include <vector>
class IOThreadPool :public Singleton<IOThreadPool>
{
public:
    using Work = boost::asio::io_context::work;
    friend class Singleton<IOThreadPool>;
    ~IOThreadPool();
    boost::asio::io_context& GetIOService();
    void Stop();
private:
    IOThreadPool(int threadNum = std::thread::hardware_concurrency());
    void Start();
    IOThreadPool(const IOThreadPool&) = delete;
    IOThreadPool& operator = (const IOThreadPool&) = delete;

    std::atomic_int thread_nums;

    boost::asio::io_context _service;

    //防止io_context.run的时候退出
    std::unique_ptr<Work> _work;

    std::vector<std::thread> pools;
};
```

- IOThreadPool继承了Singleton，实现了一个函数GetIOService获取iocontext

IOThreadPool 实现

```
#include "IOThreadPool.h"

IOThreadPool::~IOThreadPool()
{
    std::cout << "destruct IOThreadPool " << std::endl;
}

boost::asio::io_context& IOThreadPool::GetIOService()
{
    // TODO: 在此处插入 return 语句
    return _service;
}

void IOThreadPool::Stop()
{
    _work.reset();
    for (auto& td : pools) {
        td.join();
```

```
    }
}

void IOThreadPool::Start()
{
    for (int i = 0; i < thread_nums; i++) {
        pools.emplace_back([this]() {
            _service.run();
        });
    }
}

IOThreadPool::IOThreadPool(int threadNum):
    _work(new Work(_service))
{
    if (threadNum < 1) thread_nums = 1;
    else thread_nums = threadNum;
    Start();
}
```

- 构造函数中实现了一个线程池，线程池里每个线程都会运行_service.run函数，_service.run函数内部就是从iocp或者epoll获取就绪描述符和绑定的回调函数，进而调用回调函数，因为回调函数是在不同的线程里调用的，所以会存在不同的线程调用同一个socket的回调函数的情况。
- _service.run 内部在Linux环境下调用的是epoll_wait返回所有就绪的描述符列表，在windows上会循环调用GetQueuedCompletionStatus函数返回就绪的描述符，二者原理类似，进而通过描述符找到对应的注册的回调函数，然后调用回调函数。

iocp的流程是这样的

```
IOCP的使用主要分为以下几步：
1 创建完成端口(iocp)对象
2 创建一个或多个工作线程，在完成端口上执行并处理投递到完成端口上的I/O请求
3 Socket关联iocp对象，在Socket上投递网络事件
4 工作线程调用GetQueuedCompletionStatus函数获取完成通知封包，取得事件信息并进行处理
```

epoll流程是这样的

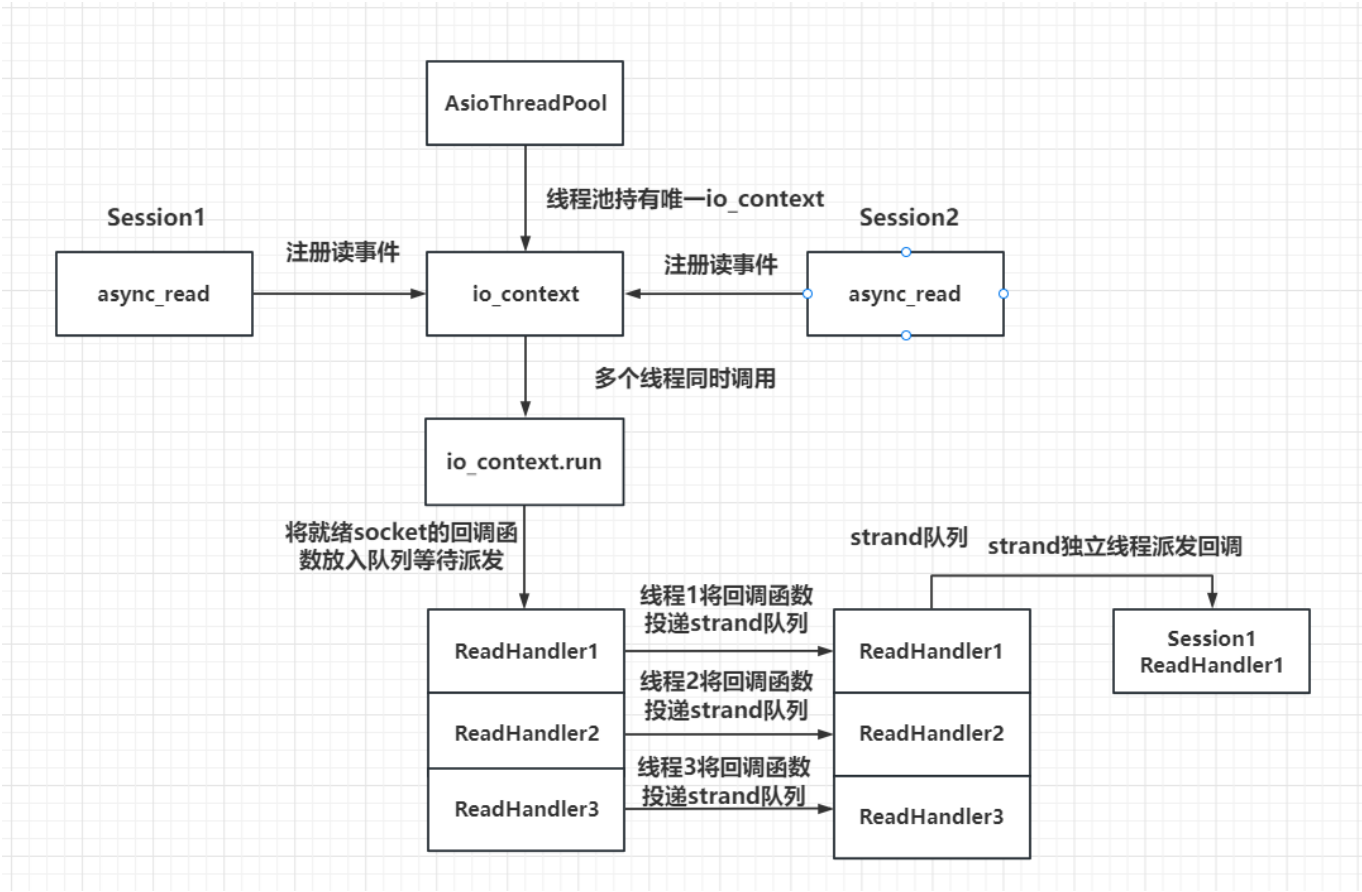
```
1 调用epoll_creat在内核中创建一张epoll表
2 开辟一片包含n个epoll_event大小的连续空间
3 将要监听的socket注册到epoll表里
4 调用epoll_wait，传入之前我们开辟的连续空间，epoll_wait返回就绪的epoll_event列表，
epoll会将就绪的socket信息写入我们之前开辟的连续空间
```

隐患

- IOThreadPool模式有一个隐患，同一个socket的就绪后，触发的回调函数可能在不同的线程里，比如第一次是在线程1，第二次是在线程3，如果这两次触发间隔时间不大，那么很可能出现不同线程并发访问数据的情况，比如在处理读事件时，第一次回调触发后我们从socket的接收缓冲区读数据出来，第二次回调触发,还是从socket的接收缓冲区读数据，就会造成两个线程同时从socket中读数据的情况，会造成数据混乱。

利用strand改进

- 对于多线程触发回调函数的情况，我们可以利用asio提供的串行类strand封装一下，这样就可以被串行调用了，其基本原理就是在线程各自调用函数时取消了直接调用的方式，而是利用一个strand类型的对象将要调用的函数投递到strand管理的队列中，再由一个统一的线程调用回调函数，调用是串行的，解决了线程并发带来的安全问题。



- 图中当socket就绪后并不是由多个线程调用每个socket注册的回调函数，而是将回调函数投递给strand管理的队列，再由strand统一调度派发。
- 为了让回调函数被派发到strand的队列，我们只需要在注册回调函数时加一层strand的包装即可。
- 在CSession类中添加一个成员变量

```
strand<io_context::executor_type> _strand;
```

- CSession构造函数

```
CSession::CSession(boost::asio::io_context& io_context, CServer* server):
    _socket(io_context), _server(server), _b_close(false),
    _b_head_parse(false), _strand(io_context.get_executor()){
    boost::uuids::uuid  a_uuid = boost::uuids::random_generator()();
    _uuid = boost::uuids::to_string(a_uuid);
    _recv_head_node = make_shared<MsgNode>(HEAD_TOTAL_LEN);
}
```

- 可以看到_strand的初始化是放在初始化列表里，利用io_context.get_executor()返回的执行器构造strand。
- 因为在asio中无论iocontext还是strand，底层都是通过executor调度的，我们将他理解为调度器就可以了，如果多个iocontext和strand的调度器是一个，那他们的消息派发统一由这个调度器执行。
- 我们利用iocontext的调度器构造strand，这样他们统一由一个调度器管理。在绑定回调函数的调度器时，我们选择strand绑定即可。
- 比如我们在Start函数里添加绑定，将回调函数的调用者绑定为_strand

```
void CSession::Start(){
    ::memset(_data, 0, MAX_LENGTH);
    _socket.async_read_some(boost::asio::buffer(_data, MAX_LENGTH),
        boost::asio::bind_executor(_strand, std::bind(&CSession::HandleRead, this,
            std::placeholders::_1, std::placeholders::_2, SharedSelf())));
}
```

- 同样的道理，在所有收发的地方，都将调度器绑定为_strand， 比如发送部分我们需要修改为如下

```
auto& msgnode = _send_que.front();
    boost::asio::async_write(_socket, boost::asio::buffer(msgnode->_data, msgnode-
>_total_len),
    boost::asio::bind_executor(_strand, std::bind(&CSession::HandleWrite, this,
std::placeholders::_1, SharedSelf()))
        );
```

- 修改main函数

```
#include "CServer.h"
#include <csignal>
#include <thread>
#include <mutex>
#include <condition_variable>
#include "IOThreadPool.h"
using namespace std;
bool bstop = false;
std::condition_variable cond_quit;
```

```
std::mutex mtx;
int main()
{
    try {
        auto pool = IOThreadPool::GetInstance();
        boost::asio::io_context ioc;
        boost::asio::signal_set signals(ioc, SIGINT, SIGTERM);
        //参数列表有两个参数表示接收到的信号,他的注册函数会在asio的独立线程里
        //所以共享变量需要加锁
        signals.async_wait([&ioc](auto, auto) {
            ioc.stop();
            pool->stop();
            //加锁与条件变量是因为在主线程里并没有调用
            //io_context.run()函数，而是在线程池里调用
            //线程池相当于主线程的子线程
            std::unique_lock<std::mutex> lock(mtx);
            bstop = true;
            cond_quit.notify_one();
            });
        //修改这里
        CServer server(pool->GetIOService(), 8888);
        {
            std::unique_lock<std::mutex> lock(mtx);
            while (!bstop) {
                //线程挂起，锁释放
                cond_quit.wait(lock);
            }
        }
    }
    catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

CSession代码

- 改动了将回调函数绑定到_strand的处理器里上

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include <queue>
#include <mutex>
#include <boost/uuid/uuid_io.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include "MsgNode.h"
#include "const.h"

//避免循环依赖问题使用前置声明
class CServer;
```

```
class LogicSystem;
class CSession: public std::enable_shared_from_this<CSession>
{
public:
    CSession(boost::asio::io_context& ioc, CServer* server);
    ~CSession();
    boost::asio::ip::tcp::socket& GetSocket();
    std::string& GetUuid();
    void Start();
    void Send(char* msg, short max_len, short msgid);
    void Send(std::string msg, short msgid);
    void Close();
    std::shared_ptr<CSession> SharedSelf();
private:
    void HandleRead(const boost::system::error_code& error, size_t
bytes_transferred, std::shared_ptr<CSession> shared_self);
    void HandleWrite(const boost::system::error_code& error,
std::shared_ptr<CSession> shared_self);
    boost::asio::ip::tcp::socket _socket;
    std::string _uuid;
    char _data[MAX_LENGTH];
    CServer* _server;
    //标志是否关闭
    bool _b_close;
    //用于保证发送信息的时序性
    std::queue<std::shared_ptr<SendNode>> _send_que;
    //对队列加锁
    std::mutex _send_lock;
    // 收到的消息结构
    std::shared_ptr<RecvNode> _recv_msg_node;
    //判断头部节点是否构造完成
    bool _b_head_parse;
    //收到的头部结构,包括消息id与消息体长度
    std::shared_ptr<MsgNode> _recv_head_node;

    //并行执行事件改为串行执行，使用strand
    //每个srand有个执行类型，让他为上下文的执行类型
    //类似我们封装的logic system
    boost::asio::strand<boost::asio::io_context::executor_type> _strand;
};

class LogicNode {
    friend class LogicSystem;
public:
    LogicNode(std::shared_ptr<CSession> session, std::shared_ptr<RecvNode>
recvnode);
private:
    std::shared_ptr<CSession> _session;
    std::shared_ptr<RecvNode> _recvnode;
};
```

性能对比

- 为了比较两种服务器多线程模式的性能，我们还是利用之前测试的客户端，客户端每隔10ms建立一个连接，总共建立100个连接，每个连接收发500次，总计10万个数据包，测试一下性能。
- 客户端测试代码如下

```
#include <iostream>
#include <boost/asio.hpp>
#include <thread>
#include <json/json.h>
#include <json/value.h>
#include <json/reader.h>
#include <chrono>
using namespace std;
using namespace boost::asio::ip;
const int MAX_LENGTH = 1024 * 2;
const int HEAD_LENGTH = 2;
const int HEAD_TOTAL = 4;
std::vector<thread> vec_threads;
int main()
{
    auto start = std::chrono::high_resolution_clock::now(); // 获取开始时间
    for (int i = 0; i < 100; i++) {
        vec_threads.emplace_back([]() {
            try {
                //创建上下文服务
                boost::asio::io_context   ioc;
                //构造endpoint
                tcp::endpoint  remote_ep(address::from_string("127.0.0.1"),
10086);
                tcp::socket  sock(ioc);
                boost::system::error_code   error =
boost::asio::error::host_not_found; ;
                sock.connect(remote_ep, error);
                if (error) {
                    cout << "connect failed, code is " << error.value() << " error
msg is " << error.message();
                    return 0;
                }
                int i = 0;
                while (i < 500) {
                    Json::Value root;
                    root["id"] = 1001;
                    root["data"] = "hello world";
                    std::string request = root.toStyledString();
                    size_t request_length = request.length();
                    char send_data[MAX_LENGTH] = { 0 };
                    int msgid = 1001;
                    int msgid_host =
boost::asio::detail::socket_ops::host_to_network_short(msgid);
                    memcpy(send_data, &msgid_host, 2);
```

```
                    //转为网络字节序
                    int request_host_length =
boost::asio::detail::socket_ops::host_to_network_short(request_length);
                    memcpy(send_data + 2, &request_host_length, 2);
                    memcpy(send_data + 4, request.c_str(), request_length);
                    boost::asio::write(sock, boost::asio::buffer(send_data,
request_length + 4));
                    cout << "begin to receive..." << endl;
                    char reply_head[HEAD_TOTAL];
                    size_t reply_length = boost::asio::read(sock,
boost::asio::buffer(reply_head, HEAD_TOTAL));
                    msgid = 0;
                    memcpy(&msgid, reply_head, HEAD_LENGTH);
                    short msglen = 0;
                    memcpy(&msglen, reply_head + 2, HEAD_LENGTH);
                    //转为本地字节序
                    msglen =
boost::asio::detail::socket_ops::network_to_host_short(msglen);
                    msgid =
boost::asio::detail::socket_ops::network_to_host_short(msgid);
                    char msg[MAX_LENGTH] = { 0 };
                    size_t  msg_length = boost::asio::read(sock,
boost::asio::buffer(msg, msglen));
                    Json::Reader reader;
                    reader.parse(std::string(msg, msg_length), root);
                    std::cout << "msg id is " << root["id"] << " msg is " <<
root["data"] << endl;
                    i++;
                }
            }
            catch (std::exception& e) {
                std::cerr << "Exception: " << e.what() << endl;
            }
            });
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    for (auto& t : vec_threads) {
        t.join();
    }
    // 执行一些需要计时的操作
    auto end = std::chrono::high_resolution_clock::now(); // 获取结束时间
    auto duration = std::chrono::duration_cast<std::chrono::seconds>(end - start);
// 计算时间差，单位为微秒
    std::cout << "Time spent: " << duration.count() << " seconds." << std::endl;
// 输
    getchar();
    return 0;
}
```

- 测试得出今天实现的多线程模式较之前的IOServicePool版本慢了7秒

取舍

- 实际的生产和开发中，我们尽可能利用C++特性，使用多核的优势，将iocontext分布在不同的线程中效率更可取一点（也就是第一种），但也要防止线程过多导致cpu切换带来的时间片开销，所以尽量让开辟的线程数小于或等于cpu的核数，从而利用多核优势。

boost::asio协程实现并发服务器

简介

- 之前介绍了asio服务器并发编程的几种模型，包括单线程，多线程IOServicePool，多线程IOThreadPool等，今天带着大家利用asio协程实现并发服务器。利用协程实现并发程序有两个好处
    1. 将回调函数改写为顺序调用，提高开发效率。
    2. 协程调度比线程调度更轻量化，因为协程是运行在用户空间的，线程切换需要在用户空间和内核空间切换。

协程案例

- asio官网提供了一个协程并发编程的案例，我们列举一下

```
#include <boost/asio/co_spawn.hpp>
#include <boost/asio/detached.hpp>
#include <boost/asio/io_context.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <boost/asio/signal_set.hpp>
#include <boost/asio/write.hpp>
#include <iostream>
#include <string>

//允许异步等待
using boost::asio::awaitable;
//启动协程需要
using boost::asio::co_spawn;
//启动协程的方式，让协程独立启动
using boost::asio::detached;
//让协程可以等待
using boost::asio::use_awaitable;
//返回协程当前执行的环境，将此可以作为调度器
namespace this_coro = boost::asio::this_coro;

awaitable<void> echo(boost::asio::ip::tcp::socket sock) {
    try {
        char data[1024];
        while (true) {
            size_t n = co_await sock.async_read_some(boost::asio::buffer(data),
use_awaitable);
            std::cout << "receive message is " << std::string(data, n) <<
std::endl;
            co_await boost::asio::async_write(sock, boost::asio::buffer(data, n),
use_awaitable);
            std::cout << "send message: " << std::string(data, n) << "successed"
<< std::endl;
        }
```

```
    }
    catch (std::exception& e) {
        std::cout << "echo Exception is " << e.what() << std::endl;
    }
}

//为了让协程能够使用，得需要加入关键字awaitable
awaitable<void> listener() {
    //co_await 是异步查询来获得调度器，如果没查到就挂起
    //执行主线程中的别的协程，知道能捕获到执行器再切回来
    auto executor = co_await this_coro::executor;
    boost::asio::ip::tcp::endpoint ep(boost::asio::ip::tcp::v4(), 8888);
    boost::asio::ip::tcp::acceptor acceptor(executor, ep);
    while (true) {
        std::cout << "等待客户端的连接" << std::endl;
        //加入co_await关键字就不需要传递回调函数了，挂起回释放协程使用权
        //use_awaitable让这个函数能够阻塞等待
        boost::asio::ip::tcp::socket sock = co_await
acceptor.async_accept(use_awaitable);
        co_spawn(executor, echo(std::move(sock)), detached);
    }

}
int main() {
    try {
        //参数是用来指定并发的级别，如果是1就表示只开一个工作线程
        //如果是0就是默认，如果大于os的实际，则根据os的实际来运行
        boost::asio::io_context ioc(1);
        boost::asio::signal_set signals(ioc, SIGINT, SIGTERM);
        signals.async_wait([&](auto, auto) {
            ioc.stop();
            });
        //协程创建函数，第一个参数为上下文，第二个参数为一个协程函数
        //第三个参数为执行策略，策略表示协程将在执行完成后自动销毁，而不会等待其父协程的
完成
        co_spawn(ioc, listener(), detached);
        ioc.run();
    }
    catch (std::exception& e) {
        std::cout << "Exception is" << e.what() << std::endl;
    }
    return 0;
}
```

1. 我们用awaitable声明了一个函数，那么这个函数就变为可等待的函数了，比如listener被添加awaitable之后，就可以被协程调用和等待了。
2. co_spawn表示启动一个协程，参数分别为调度器，执行的函数，以及启动方式, 比如我们启动了一个协程，deatched表示将协程对象分离出来，这种启动方式可以启动多个协程，他们都是独立的，如何调度取决于调度器，在用户的感知上更像是线程调度的模式，类似于并发运行，其实底层都是串行的。

```
co_spawn(io_context, listener(), detached);
```

我们启动了一个协程，执行listener中的逻辑，listener内部co_await 等待 acceptor接收连接，如果没有连接到来则挂起协程。执行之后的io_context.run()逻辑。所以协程实际上是在一个线程中串行调度的，只是感知上像是并发而已。 3. 当acceptor接收到连接后，继续调用co_spawn启动一个协程，用来执行echo逻辑。echo逻辑里也是通过co_wait的方式接收和发送数据的，如果对端不发数据，执行echo的协程就会挂起，另一个协程启动，继续接收新的连接。当没有连接到来，接收新连接的协程挂起，如果所有协程都挂起，则等待新的就绪事件(对端发数据，或者新连接)到来唤醒。

- **使用协程的时候有没有 co_await的区别**
    1. 不使用 co_await： 如果不使用 co_await，acceptor.async_accept 将会返回一个可等待对象，但不会在此处等待该对象的完成。相反，它将立即返回，继续执行后续的代码，而不管是否有连接请求到来。这可能导致后续的代码在没有获得有效的 tcp::socket 对象的情况下进行执行，从而产生错误或未定义的行为。
    2. 使用 co_await： 当使用 co_await 时，协程会在 acceptor.async_accept 返回的可等待对象完成之前挂起，并暂停当前协程的执行。这意味着协程会等待连接请求到来，并在收到请求后继续执行。在这种情况下，async_accept 返回的 tcp::socket 对象将被分配给变量 sock，以便后续与客户端进行通信。

完整并发服务器

- 由于服务器发送数据或者请求比较频繁，所以考虑设计为不用协程而用线程的方式，这样相比协程可以增加效率，而且发送可能会在其他线程
- 我们可以利用协程改进服务器编码流程，用一个iocontext管理绑定acceptor用来接收新的连接，再用一个iocontext或以IOServicePool的方式管理连接的收发操作，在每个连接的接收数据时改为启动一个协程，通过顺序的方式读取收到的数据

AsiIOServicePool

AsioIOServicePool头文件

```
#pragma once
#include <iostream>
#include <boost/asio.hpp>
#include <vector>
class AsioIOServicePool
{
public:
    using IOService = boost::asio::io_context;
    using Work = boost::asio::io_context::work;
    using WorkPtr = std::unique_ptr<Work>;
    ~AsioIOServicePool();
    AsioIOServicePool(const AsioIOServicePool&) = delete;
    AsioIOServicePool& operator=(const AsioIOServicePool&) = delete;
    static AsioIOServicePool& GetInstance() {
        static AsioIOServicePool ins;
        return ins;
    }
    void Stop();
    boost::asio::io_context& GetIOService();
```

```
private:
    AsioIOServicePool(const int nums = std::thread::hardware_concurrency());
    std::vector<IOService> _ioService;
    std::vector<WorkPtr> _works;
    std::vector<std::thread> _threads;

    //因为轮询，所以要记录下一个io_context的下标是多少
    std::size_t _nextIOService;
};
```

AsioIOServicePool实现

```
#include "AsioIOServicePool.h"

AsioIOServicePool::~AsioIOServicePool()
{
    std::cout << "AsioIOService Pool destruct " << std::endl;
}

void AsioIOServicePool::Stop()
{
    //将work置空，会让ioService自动析构
    for (auto& work:_works) {
        work.reset();
    }
    //等待线程完成任务
    for (auto& td : _threads) {
        if (td.joinable()) {
            td.join();
        }
    }
}

boost::asio::io_context& AsioIOServicePool::GetIOService()
{
    auto& service = _ioService[_nextIOService++];
    if (_nextIOService == _ioService.size()) {
        _nextIOService = 0;
    }
    return service;
}

AsioIOServicePool::AsioIOServicePool(const int nums):_ioService(nums),
_works(nums),_nextIOService(0)
{
    //初始化works指针
    for (std::size_t i = 0; i < nums; i++) {
        //右边为右值
        _works[i] = std::unique_ptr<Work>(new Work(_ioService[i]));
    }
```

```
    //遍历多个ioService 创建多个线程，每个线程内部启动一个ioService
    for (std::size_t i = 0; i < nums; i++) {
        _threads.emplace_back([this, i]() {
            _ioService[i].run();
            });
    }
}
```

cosnt.h

- 专门用来存放常量

```
#pragma once
constexpr int MAX_LENGTH = 1024 * 2;
constexpr int HEAD_TOTAL_LEN = 4;
constexpr int HEAD_ID_LEN = 2;
constexpr int HEAD_DATA_LEN = 2;
constexpr int MAX_RECVQUE = 10000;
constexpr int MAX_SENDQUE = 1000;

enum MSG_IDS {
    MSG_HELLO_WORLD = 1001
};
```

CServer

CServer头文件

```
#pragma once
#include <memory.h>
#include <iostream>
#include <map>
#include <mutex>
#include <condition_variable>
#include <boost/asio.hpp>
#include "CSession.h"
//负责监听客户端的链接，处理链接
class CServer
{
public:
    CServer(boost::asio::io_context& io_context, short port);
    ~CServer();
    void ClearSession(std::string& uuid);
private:
    void HandleAccpet(std::shared_ptr<CSession> session, const
boost::system::error_code& error);
    void StartAccept();
```

```
    boost::asio::io_context& _io_context;
    std::mutex mtx;
    short _port;
    boost::asio::ip::tcp::acceptor _acceptor;
    std::map<std::string, std::shared_ptr<CSession>> _sessions;
};
```

CServer实现

```
#include "CServer.h"
#include "AsioIOServicePool.h"

CServer::CServer(boost::asio::io_context& io_context, short port)
    :_io_context(io_context),_port(port),_acceptor(io_context,
boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), port))
{
    std::cout << "Server start success, on port: " << _port << std::endl;
    StartAccept();
}

CServer::~CServer()
{
    std::cout << "Server destruct listen on port : " << _port << std::endl;
}

void CServer::ClearSession(std::string& uuid)
{
    //加锁
    std::lock_guard<std::mutex> lock(mtx);
    if (_sessions.find(uuid) != _sessions.end()) _sessions.erase(uuid);
}

void CServer::HandleAccpet(std::shared_ptr<CSession> session, const
boost::system::error_code& error)
{
    if (!error) {
        //启动服务
        session->Start();
        //加锁
        std::lock_guard<std::mutex> lock(mtx);
        _sessions.insert(std::make_pair(session->GetUuid(), session));
    }
    else {
        std::cout << "session accept failed, error is " << error.what() <<
std::endl;
    }
    StartAccept();
}
```

```
void CServer::StartAccept()
{
    //取得上下文
    auto& io_context = AsioIOServicePool::GetInstance().GetIOService();
    //构造一个CSession的智能指针,也就是创建连接
    std::shared_ptr<CSession> new_session = std::make_shared<CSession>
(_io_context, this);
    //进行异步连接
    _acceptor.async_accept(new_session->GetSocket(),
        std::bind(&CServer::HandleAccpet, this, new_session,
std::placeholders::_1));
}
```

CSession

CSession.h

```
#pragma once
#include <memory>
#include <boost/asio.hpp>
#include <string>
#include <boost/asio/co_spawn.hpp>
#include <boost/asio/detached.hpp>
#include <queue>
#include <mutex>
#include "const.h"
#include "MsgNode.h"
class CServer;
//用来处理客户端与服务器之间通s信的作用
class CSession:public std::enable_shared_from_this<CSession>
{
public:
    CSession(boost::asio::io_context& io_context, CServer* server);
    ~CSession();
    boost::asio::ip::tcp::socket& GetSocket() {
        return _socket;
    }
    void Start();
    void Close();
    std::string& GetUuid();
    void Send(const char* msg, short max_length, short msg_id);
    void Send(std::string msg, short msg_id);
    void HandleWrite(const boost::system::error_code& error,
std::shared_ptr<CSession> shared_self);
private:
    boost::asio::io_context& _io_context;
    CServer* _server;
    boost::asio::ip::tcp::socket _socket;
    std::string _uuid;
    bool _b_close;
```

```
    std::mutex mtx;
    std::queue<std::shared_ptr<SendNode>> _send_que;
    std::shared_ptr<RecvNode> _recv_msg_node;
    std::shared_ptr<MsgNode> _recv_head_node;
};

class LogicNode {
    friend class LogicSystem;
public:
    LogicNode(std::shared_ptr<CSession> session, std::shared_ptr<RecvNode>
recvnode) :
        _session(session), _recvnode(recvnode)
    {

    }
private:
    std::shared_ptr<CSession> _session;
    std::shared_ptr<RecvNode> _recvnode;
};
```

CSession实现

```
#include "CSession.h"
#include "CServer.h"
#include <boost/uuid/uuid_io.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include "LogicSystem.h"
CSession::CSession(boost::asio::io_context& io_context, CServer* server):
_io_context(io_context), _server(server),_socket(io_context),_b_close(false)
{
    boost::uuids::uuid a_uuid = boost::uuids::random_generator()();
    _uuid = boost::uuids::to_string(a_uuid);
    //初始化头部节点
    _recv_head_node = std::make_shared<MsgNode>(HEAD_TOTAL_LEN);
}

CSession::~CSession()
{
    try {
        std::cout << "~CSession destruct" << std::endl;
        Close();
    }
    catch (std::exception& e) {
        std::cout << "exception is " << e.what() << std::endl;
    }
}

void CSession::Start()
{
```

```
    //为了防止智能指针被意外的释放
    auto shared_this = shared_from_this();
    //开启协程
    boost::asio::co_spawn(_io_context, [=, this]()->boost::asio::awaitable<void> {
        try {
            while (!_b_close)
            {
                //接收数据前先清空数组
                _recv_head_node->Clear();
                //开始接收数据
                size_t n = co_await boost::asio::async_read(_socket,
boost::asio::buffer(_recv_head_node->_data,
                    HEAD_TOTAL_LEN), boost::asio::use_awaitable);
                if (n == 0) {
                    std::cout << "receive peer closed" << std::endl;
                    Close();
                    _server->ClearSession(_uuid);
                    //协程的返回
                    co_return;
                }

                //获取头部MSGID数据
                short msg_id = 0;
                memcpy(&msg_id, _recv_head_node->_data, HEAD_ID_LEN);
                //将网络字节序转成本地字节序
                short msg_id_host =
boost::asio::detail::socket_ops::network_to_host_short(msg_id);
                std::cout << "msg_id is" << msg_id_host << std::endl;
                if (msg_id_host > MAX_LENGTH) {
                    std::cout << "invaild msg id is " << msg_id_host << std::endl;
                    //std::cout << "hello" << std::endl;
                    Close();
                    _server->ClearSession(_uuid);
                    co_return;
                }

                //获取数据长度
                short msg_len = 0;
                memcpy(&msg_len, _recv_head_node->_data + HEAD_ID_LEN,
HEAD_DATA_LEN);
                short msg_len_host =
boost::asio::detail::socket_ops::network_to_host_short(msg_len);
                std::cout << "msg len is" << msg_len_host << std::endl;
                if (msg_len_host > MAX_LENGTH) {
                    std::cout << "invaild msg len is " << msg_len_host <<
std::endl;
                    Close();
                    _server->ClearSession(_uuid);
                    co_return;
                }

                //实际数据
                _recv_msg_node = std::make_shared<RecvNode>(msg_len_host,
msg_id_host);
```

```
                //读出包体
                n = co_await boost::asio::async_read(_socket,
boost::asio::buffer(_recv_msg_node->_data,
                    _recv_msg_node->_total_len), boost::asio::use_awaitable);
                if (n == 0) {
                    std::cout << "receive peer closed" << std::endl;
                    Close();
                    _server->ClearSession(_uuid);
                    //协程的返回
                    co_return;
                }

                _recv_msg_node->_data[_recv_msg_node->_total_len] = '\0';
                std::cout << "recv data is" << _recv_msg_node->_data << std::endl;

                //投递到逻辑系统的逻辑队列里，交给队列处理

LogicSystem::GetInstance().PostMsgToQue(std::make_shared<LogicNode>
(shared_from_this(), _recv_msg_node));
            }
        }
        catch (std::exception e) {
            std::cout << "Exception is " << e.what() << std::endl;
            Close();
            //关闭后从map中移除session
            _server->ClearSession(_uuid);
        }
        }, boost::asio::detached);
}

void CSession::Close()
{
    _b_close = true;
    //关闭socket
    _socket.close();
}

std::string& CSession::GetUuid()
{
    return _uuid;
    // TODO: 在此处插入 return 语句
}

void CSession::Send(const char* msg, short max_length, short msg_id)
{
    std::unique_lock<std::mutex> lock(mtx);
    int send_que_size = _send_que.size();
    if (send_que_size > MAX_SENDQUE) {
        std::cout << "session: " << _uuid << "send que fulled, size is "
            << MAX_SENDQUE << std::endl;
        return;
    }
    //往消息队列里插入数据
```

```
    _send_que.push(std::make_shared<SendNode>(msg, max_length, msg_id));
    //判断数据量是否大于0，大于0就不发送数据，只有当发送队列里长度为1的时候才发送数据
    //那么为什么他判断大于0呢，因为我们是先统计的数据量，再插入队列
    if (send_que_size > 0) {
        return;
    }

    std::shared_ptr<SendNode> msgnode = _send_que.front();
    lock.unlock();
    //传递shread_from_this可以增加引用，防止session意外的释放
    boost::asio::async_write(_socket, boost::asio::buffer(msgnode->_data, msgnode-
>_total_len),
        std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
shared_from_this()));
}

void CSession::Send(std::string msg, short msg_id)
{
    Send(msg.c_str(), msg.length(), msg_id);
}

//注意对于error_code 必须加入const，否则会在async_write处会报错
void CSession::HandleWrite(const boost::system::error_code& error,
std::shared_ptr<CSession> shared_self)
{
    try {
        if (!error) {
            std::unique_lock<std::mutex> lock(mtx);
            //调用这个回调函数的时候说明数据已经处理完了
            _send_que.pop();
            //如果队列不为空就要继续发送数据
            if (!_send_que.empty()) {
                auto& msgnode = _send_que.front();
                boost::asio::async_write(_socket, boost::asio::buffer(msgnode-
>_data, msgnode->_total_len),
                    std::bind(&CSession::HandleWrite, this, std::placeholders::_1,
shared_from_this()));
            }
        }
        else {
            std::cout << "handle write failed, error is" << error.what() <<
std::endl;
            Close();
            _server->ClearSession(_uuid);
        }
    }
    catch (std::exception& e) {
        std::cout << "exception is " << e.what() << std::endl;
        Close();
        _server->ClearSession(_uuid);
    }
}
```

LogicSystem

LogicSystem头文件

```
#pragma once
#include <iostream>
#include <thread>
#include <mutex>
#include <queue>
#include <condition_variable>
#include <map>
#include <functional>
#include <json/json.h>
#include <json/reader.h>
#include <json/value.h>
#include "const.h"
#include "CSession.h"
//因为是string 可以调用函数获取长度，所以不需要传长度
typedef std::function<void(std::shared_ptr<CSession> session, const short& msg_id,
const std::string& msg_data)> FunCallBack;
class LogicSystem
{
public:
    LogicSystem(const LogicSystem&) = delete;
    LogicSystem& operator=(const LogicSystem&) = delete;
    ~LogicSystem();
    static LogicSystem& GetInstance() {
        static LogicSystem ins;
        return ins;
    }
    void PostMsgToQue(std::shared_ptr<LogicNode> msg);
private:
    LogicSystem();
    void RegisterCallBacks();
    void HelloWorldCallBack(std::shared_ptr<CSession> session, const short&
msg_id, const std::string& msg_data);
    void DealMsg();
    std::thread _worker;
    std::mutex mtx;
    //与直接使用LogicNode的区别
    std::queue<std::shared_ptr<LogicNode>> _msg_que;
    std::condition_variable _consume;
    bool _b_stop;
    std::map<short, FunCallBack> _fun_callbacks;
};
```

LogicSystem实现

```
#include "LogicSystem.h"

LogicSystem::~LogicSystem()
{
    _b_stop = true;
    _consume.notify_one();
    _worker.join();
}

void LogicSystem::PostMsgToQue(std::shared_ptr<LogicNode> msg)
{
    std::unique_lock<std::mutex> unique_lk(mtx);
    _msg_que.push(msg);

    //由0变为1则发送通知信号
    if (_msg_que.size() == 1) {
        unique_lk.unlock();
        _consume.notify_one();
    }
}

LogicSystem::LogicSystem():_b_stop(false)
{
    RegisterCallBacks();
    _worker = std::thread(&LogicSystem::DealMsg, this);
}

void LogicSystem::RegisterCallBacks()
{
    //任务编号,将他存储起来
    _fun_callbacks[MSG_HELLO_WORLD] = std::bind(&LogicSystem::HelloWorldCallBack,
this,
        std::placeholders::_1, std::placeholders::_2, std::placeholders::_3);
}

void LogicSystem::HelloWorldCallBack(std::shared_ptr<CSession> session, const
short& msg_id, const std::string& msg_data)
{
    Json::Reader reader;
    Json::Value root;
    reader.parse(msg_data, root);
    std::cout << "recevie msg id is " << root["id"].asInt() << "receive msg data
is "
        << root["data"].asString() << std::endl;
    root["data"] = "server has received msg, msg data is " +
root["data"].asString();
    std::string return_str = root.toStyledString();
    session->Send(return_str, root["id"].asInt());
}

void LogicSystem::DealMsg()
{
    while (true) {
```

```
        std::unique_lock<std::mutex> lock(mtx);
        _consume.wait(lock, [this]() {
            return _b_stop || !_msg_que.empty();
            });
        if (_b_stop) {
            while (!_msg_que.empty()) {
                std::shared_ptr<LogicNode> msg_node = _msg_que.front();
                std::cout << "recv_msg id  is " << msg_node->_recvnode->_msg_id <<
std::endl;
                auto call_back_iter = _fun_callbacks.find(msg_node->_recvnode-
>_msg_id);
                //如果没有注册回调函数就直接出队
                if (call_back_iter == _fun_callbacks.end()) {
                    _msg_que.pop();
                    continue;
                }
                //否则调用回调函数
                call_back_iter->second(msg_node->_session, msg_node->_recvnode-
>_msg_id,
                    std::string(msg_node->_recvnode->_data, msg_node->_recvnode-
>_total_len));
                _msg_que.pop();
            }
            break;
        }
        //如果没有停服，且说明队列中有数据
        auto msg_node = _msg_que.front();
        std::cout << "recv_msg id  is " << msg_node->_recvnode->_msg_id <<
std::endl;
        auto call_back_iter = _fun_callbacks.find(msg_node->_recvnode->_msg_id);
        if (call_back_iter == _fun_callbacks.end()) {
            _msg_que.pop();
            continue;
        }
        call_back_iter->second(msg_node->_session, msg_node->_recvnode->_msg_id,
            std::string(msg_node->_recvnode->_data, msg_node->_recvnode-
>_total_len));
        _msg_que.pop();
    }
}
```

MsgNode

MsgNode头文件

```
#pragma once
#include <iostream>
#include "const.h"
#include <string>
#include <boost/asio.hpp>
```

```
class MsgNode
{
public:
    MsgNode(short max_len) :_total_len(max_len), _cur_len(0) {
        _data = new char[_total_len + 1];
        _data[_total_len] = '\0';
    }

    ~MsgNode() {
        std::cout << "destruct MsgNode" << std::endl;
        delete[] _data;
    }

    void Clear() {
        memset(_data, 0, _total_len);
        _cur_len = 0;
    }
public:
    int _cur_len;
    int _total_len;
    char* _data;
};

class RecvNode :public MsgNode {
public:
    RecvNode(short max_len, short msg_id);
public:
    short _msg_id;
};

class SendNode :public MsgNode {
public:
    SendNode(const char* msg, short max_len, short msg_id);
public:
    short _msg_id;
};
```

MsgNode 实现

```
#include "MsgNode.h"

RecvNode::RecvNode(short max_len, short msg_id):MsgNode(max_len), _msg_id(msg_id)
{
}

SendNode::SendNode(const char* msg, short max_len, short msg_id):MsgNode(max_len +
HEAD_TOTAL_LEN), _msg_id(msg_id)
{
    //先发送id,转为网络字节序
```

```
    short msg_id_net =
boost::asio::detail::socket_ops::host_to_network_short(msg_id);
    memcpy(_data, &msg_id_net, HEAD_ID_LEN);
    //再发送长度，转为网络字节序
    short msg_len_net =
boost::asio::detail::socket_ops::host_to_network_short(max_len);
    memcpy(_data + HEAD_ID_LEN, &msg_len_net, HEAD_DATA_LEN);
    //最后在发送数据
    memcpy(_data + HEAD_TOTAL_LEN, msg, max_len);
}
```

使用asio实现http服务器

简介

- 前文介绍了asio如何实现并发的长连接tcp服务器，今天介绍如何实现http服务器，在介绍实现http服务器之前，需要讲述下http报文头的格式，其实http报文头的格式就是为了避免我们之前提到的粘包现象，告诉服务器一个数据包的开始和结尾，并在包头里标识请求的类型如get或post等信息。

Http包头信息

- 一个标准的HTTP报文头通常由请求头和响应头两部分组成。

HTTP请求头

- HTTP请求头包括以下字段：

    1. Request-line：包含用于描述请求类型、要访问的资源以及所使用的HTTP版本的信息。
    2. Host：指定被请求资源的主机名或IP地址和端口号。
    3. Accept：指定客户端能够接收的媒体类型列表，用逗号分隔，例如 text/plain, text/html。
    4. User-Agent：客户端使用的浏览器类型和版本号，供服务器统计用户代理信息。
    5. Cookie：如果请求中包含cookie信息，则通过这个字段将cookie信息发送给Web服务器。
    6. Connection：表示是否需要持久连接（keep-alive）

- 比如下面就是一个实际应用

```
GET /index.html HTTP/1.1
Host: www.example.com
Accept: text/html, application/xhtml+xml, */*
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Cookie: sessionid=abcdefg1234567
Connection: keep-alive
```

- Request-line：指定使用GET方法请求/index.html资源，并使用HTTP/1.1协议版本。
- Host：指定被请求资源所在主机名或IP地址和端口号。

- Accept：客户端期望接收的媒体类型列表，本例中指定了text/html、application/xhtml+xml和任意类型的文件（/）。
- User-Agent：客户端浏览器类型和版本号。
- Cookie：客户端发送给服务器的cookie信息。
- Connection：客户端请求后是否需要保持长连接。

HTTP响应头

- HTTP响应头包括以下字段：

    1. Status-line：包含协议版本、状态码和状态消息。
    2. Content-Type：响应体的MIME类型。
    3. Content-Length：响应体的字节数。
    4. Set-Cookie：服务器向客户端发送cookie信息时使用该字段。
    5. Server：服务器类型和版本号。
    6. Connection：表示是否需要保持长连接（keep-alive）。

- 在实际的HTTP报文头中，还可以包含其他可选字段。
- 如下就是一个例子

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 1024
Set-Cookie: sessionid=abcdefg1234567; HttpOnly; Path=/
Server: Apache/2.2.32 (Unix) mod_ssl/2.2.32 OpenSSL/1.0.1e-fips mod_bwlimited/1.4
Connection: keep-alive
上述响应头包括了以下字段：
```

- Status-line：指定HTTP协议版本、状态码和状态消息。
- Content-Type：指定响应体的MIME类型及字符编码格式。
- Content-Length：指定响应体的字节数。
- Set-Cookie：服务器向客户端发送cookie信息时使用该字段。
- Server：服务器类型和版本号。
- Connection：服务器是否需要保持长连接。
- 源码请看https://gitee.com/secondtonone1/boostasio-learn

使用beast网络库实现http服务器

简介

- 前面的几篇文章已经介绍了如何使用asio搭建高并发的tcp服务器，以及http服务器。但是纯手写http服务器太麻烦了，有网络库beast已经帮我们实现了。这一期讲讲如何使用beast实现一个http服务器。

连接类

- 我们先实现http_server函数

```
void http_server(boost::asio::ip::tcp::acceptor& acceptor,
boost::asio::ip::tcp::socket& socket) {
    acceptor.async_accept(socket, [&](boost::system::error_code ec) {
        if (!ec) {
            //启动http的connection
            std::make_shared<http_connection>(std::move(socket))->Start();
        }
        //不管成功与失败都要继续监听请求
        http_server(acceptor, socket);
        });
}
```

- http_server中添加了异步接收连接的逻辑，当有新的连接到来时创建http_connection,然后启动服务，新连接监听对端数据。接下来http_server继续监听对端的新连接。

- 连接类http_connection里实现了start函数监听对端数据

```
void Start() {
    //读取请求
    read_request();
    //开始超时返回机制,对于短连接
    check_deadline();
}
```

- 处理读请求,将读到的数据存储再成员变量request_中，然后调用process_request处理请求

```
//实现读请求
void read_request() {
    //千万不要用make_shared
    auto self = shared_from_this();
    boost::beast::http::async_read(socket_, buffer_, request_,
        [self](boost::system::error_code ec, std::size_t bytes_transferred) {
            //因为用不到后面那个参数，所以忽略他
            //由于底层必须接收它，所以即使不用也要声明
            boost::ignore_unused(bytes_transferred);
            if (!ec) {
                self->process_request();
            }
        });
}
```

- check_deadline主要时用来检测超时，当超过一定时间后自动关闭连接，因为http请求时短链接

```
// 检测定时器
void check_deadline() {
    //构造伪闭包，防止被析构的时候，意外地被释放
    auto self = shared_from_this();
    //开始异步等待60s,传递的时候多传递一个self，这样做是为了
    //防止在60之内的等待中，http_connection先被释放掉
    //而导致this为空的崩溃
    deadline_.async_wait([self](boost::system::error_code ec) {
        if (!ec) {
            self->socket_.close();
        }
        });
}
```

- process_request函数中区分请求的类型，进行不同类型的处理如post还是get请求

```
void process_request() {
    //设置回应版本,设置为请求回来的
    response_.version(request_.version());
    //设置为短连接
    response_.keep_alive(false);
    //
    switch (request_.method()) {
    case boost::beast::http::verb::get:
        //设置为错误请求的状态码
        response_.result(boost::beast::http::status::ok);
        //设置响应头的MIME类型，也就是回应的数据类型
        response_.set(boost::beast::http::field::server, "Beast");
        //创建回应
        create_get_response();
            break;
    case boost::beast::http::verb::post:
        //设置为错误请求的状态码
        response_.result(boost::beast::http::status::ok);
        //设置响应头的MIME类型，也就是回应的数据类型
        response_.set(boost::beast::http::field::server, "Beast");
        //创建回应
        create_post_response();
        break;
    default:
        //设置为错误请求的状态码
        response_.result(boost::beast::http::status::bad_request);
        //设置响应头的MIME类型，也就是回应的数据类型
        response_.set(boost::beast::http::field::content_type, "text/plain");
        //在响应头的body写数据
        boost::beast::ostream(response_.body()) << "invaild request-method"
            << std::string(request_.method_string()) << "`";
        break;
    }
}
```

- create_response函数中解析了不同的路由处理get请求

```
void create_get_response() {
    //判断请求的路由,假设他要问请求了多少次
    if (request_.target() == "/count") {
        //设置返回的MIME类型为HTML
        response_.set(boost::beast::http::field::content_type, "text/html");
        boost::beast::ostream(response_.body())
            << "<html>\n"
            << "<head><title>Current time</title></head>\n"
            << "<body>\n"
            << "<h1>Current time</h1>\n"
            << "<p>The current time is "
            << my_program_state::request_count()
            << " seconds since the epoch.</p>\n"
            << "</body>\n"
            << "</html>\n";
    }
    //假设问的是时间
    else if (request_.target() == "/time") {
        //设置返回的MIME类型为HTML
        response_.set(boost::beast::http::field::content_type, "text/html");
        boost::beast::ostream(response_.body())
            << "<html>\n"
            << "<head><title>Current time</title></head>\n"
            << "<body>\n"
            << "<h1>Current time</h1>\n"
            << "<p>The current time is "
            << my_program_state::now()
            << " seconds since the epoch.</p>\n"
            << "</body>\n"
            << "</html>\n";
    }
    else {
        //设置状态为404，未找到页面
        response_.result(boost::beast::http::status::not_found);
        boost::beast::ostream(response_.body())
            << "File not found\r\n";
    }
}
```

- create_post_response处理了post请求中的一部分路由

```
    void create_post_response() {
        //判断路由是不是email
        if (request_.target() == "/email")
        {
            //取出body
            auto& body = this->request_.body();
            //得到body里的内容。将buffer转换为string
```

```
            auto body_str = boost::beast::buffers_to_string(body.data());
            std::cout << "receive body is " << body_str << std::endl;
            //设置为json
            this->response_.set(boost::beast::http::field::content_type,
"text/json");
            //这是回应给对方用的json
            Json::Value root;
            //用于解析请求
            Json::Reader reader;
            //表示原始的根
            Json::Value src_root;
            //解析到原始的根里
            bool parse_success = reader.parse(body_str, src_root);
            //如果解析错误
            if (!parse_success) {
                std::cout << "Failed to parse JSON data!" << std::endl;
                root["error"] = 1001;
                //将json序列化为字符串
                std::string jsonstr = root.toStyledString();
                //写道回应里的body里
                boost::beast::ostream(this->response_.body()) << jsonstr;
                return;
            }
            //如果解析正确
            auto email = src_root["email"].asString();
            std::cout << "email is " << email << std::endl;

            root["error"] = 0;
            root["email"] = src_root["email"];
            root["msg"] = "recevie email post success";
            //也就是以字符串的形式显示，其内部还是json格式
            std::string jsonstr = root.toStyledString();
            boost::beast::ostream(this->response_.body()) << jsonstr;
        }
        else
        {
            response_.result(boost::beast::http::status::not_found);
            response_.set(boost::beast::http::field::content_type, "text/plain");
            boost::beast::ostream(response_.body()) << "File not found\r\n";
        }
    }

};
```

- write_response发送请求

```
void write_response() {
    auto self = shared_from_this();
    //设置数据的长度类似tlv的长度
    response_.content_length(response_.body().size());
    //调用写函数
    boost::beast::http::async_write(socket_, response_,
```

```
        [self](boost::system::error_code ec, std::size_t bytes_transgerred) {
            //由于服务器是全双工的，我们关闭服务器的时候不能直接调用close
            //因为一个服务器会调用很多很多连接，如果服务器主动断开连接
            //会有很多客户端都需要等待四次挥手，所以我们为了避免这种情况
            //可以只关闭服务器的发送端
            self-
>socket_.shutdown(boost::asio::ip::tcp::socket::shutdown_send,ec);
            //将定时器取消掉
            self->deadline_.cancel();
        });
}
```

完整代码

```
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <boost/beast/version.hpp>
#include <boost/asio.hpp>
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <memory>
#include <string>
#include <iostream>
#include <string>
#include <json/json.h>
#include <json/value.h>
#include <json/reader.h>

namespace my_program_state {

    //统计别人请求的个数
    std::size_t request_count() {
        static std::size_t count = 0;
        return ++count;
    }

    //得到当前时间
    std::time_t now() {
        return std::time(0);
    }
}

//建立连接，实际上建立过程和tcp服务器类似
class http_connection :public std::enable_shared_from_this<http_connection> {
public:
    //不用引用是为了让每一个connection都独立管理自己的socket
    //所以为了避免构造 得使用移动构造函数
    http_connection(boost::asio::ip::tcp::socket socket) :
        socket_(std::move(socket))
    {
```

```
    }
    void Start() {
        //读取请求
        read_request();
        //开始超时返回机制,对于短连接
        check_deadline();
    }
private:
    //用于读取与发送的套接字
    boost::asio::ip::tcp::socket socket_;
    //接收缓冲区,指定最大为8K
    boost::beast::flat_buffer buffer_{ 8192 };
    //请求头,其中dynamic_body 表示接收任意类型请求可以是html可以是javaScript
    boost::beast::http::request< boost::beast::http::dynamic_body> request_;
    //响应头
    boost::beast::http::response<boost::beast::http::dynamic_body> response_;
    //构造一个计时器
    boost::asio::steady_timer deadline_{
        socket_.get_executor(),std::chrono::seconds(60) };

    //实现读请求
    void read_request() {
        //千万不要用make_shared
        auto self = shared_from_this();
        boost::beast::http::async_read(socket_, buffer_, request_,
            [self](boost::system::error_code ec, std::size_t bytes_transferred) {
                //因为用不到后面那个参数，所以忽略他
                //由于底层必须接收它，所以即使不用也要声明
                boost::ignore_unused(bytes_transferred);
                if (!ec) {
                    self->process_request();
                }
            });
    }
    // 检测定时器
    void check_deadline() {
        //构造伪闭包，防止被析构的时候，意外地被释放
        auto self = shared_from_this();
        //开始异步等待60s,传递的时候多传递一个self，这样做是为了
        //防止在60之内的等待中，http_connection先被释放掉
        //而导致this为空的崩溃
        deadline_.async_wait([self](boost::system::error_code ec) {
            if (!ec) {
                self->socket_.close();
            }
            });
    }
    void process_request() {
        //设置回应版本,设置为请求回来的
        response_.version(request_.version());
        //设置为短连接
        response_.keep_alive(false);
        //
```

```
        switch (request_.method()) {
        case boost::beast::http::verb::get:
            //设置为错误请求的状态码
            response_.result(boost::beast::http::status::ok);
            //设置响应头的MIME类型，也就是回应的数据类型
            response_.set(boost::beast::http::field::server, "Beast");
            //创建回应
            create_get_response();
                break;
        case boost::beast::http::verb::post:
            //设置为错误请求的状态码
            response_.result(boost::beast::http::status::ok);
            //设置响应头的MIME类型，也就是回应的数据类型
            response_.set(boost::beast::http::field::server, "Beast");
            //创建回应
            create_post_response();
            break;
        default:
            //设置为错误请求的状态码
            response_.result(boost::beast::http::status::bad_request);
            //设置响应头的MIME类型，也就是回应的数据类型
            response_.set(boost::beast::http::field::content_type, "text/plain");
            //在响应头的body写数据
            boost::beast::ostream(response_.body()) << "invaild request-method"
                << std::string(request_.method_string()) << "`";
            break;
        }
    }
    void create_get_response() {
        //判断请求的路由,假设他要问请求了多少次
        if (request_.target() == "/count") {
            //设置返回的MIME类型为HTML
            response_.set(boost::beast::http::field::content_type, "text/html");
            boost::beast::ostream(response_.body())
                << "<html>\n"
                << "<head><title>Current time</title></head>\n"
                << "<body>\n"
                << "<h1>Current time</h1>\n"
                << "<p>The current time is "
                << my_program_state::request_count()
                << " seconds since the epoch.</p>\n"
                << "</body>\n"
                << "</html>\n";
        }
        //假设问的是时间
        else if (request_.target() == "/time") {
            //设置返回的MIME类型为HTML
            response_.set(boost::beast::http::field::content_type, "text/html");
            boost::beast::ostream(response_.body())
                << "<html>\n"
                << "<head><title>Current time</title></head>\n"
                << "<body>\n"
                << "<h1>Current time</h1>\n"
                << "<p>The current time is "
```

```
                << my_program_state::now()
                << " seconds since the epoch.</p>\n"
                << "</body>\n"
                << "</html>\n";
        }
        else {
            //设置状态为404，未找到页面
            response_.result(boost::beast::http::status::not_found);
            boost::beast::ostream(response_.body())
                << "File not found\r\n";
        }
    }

    void write_response() {
        auto self = shared_from_this();
        //设置数据的长度类似tlv的长度
        response_.content_length(response_.body().size());
        //调用写函数
        boost::beast::http::async_write(socket_, response_,
            [self](boost::system::error_code ec, std::size_t bytes_transgerred) {
                //由于服务器是全双工的，我们关闭服务器的时候不能直接调用close
                //因为一个服务器会调用很多很多连接，如果服务器主动断开连接
                //会有很多客户端都需要等待四次挥手，所以我们为了避免这种情况
                //可以只关闭服务器的发送端
                self-
>socket_.shutdown(boost::asio::ip::tcp::socket::shutdown_send,ec);
                //将定时器取消掉
                self->deadline_.cancel();
            });
    }

    void create_post_response() {
        //判断路由是不是email
        if (request_.target() == "/email")
        {
            //取出body
            auto& body = this->request_.body();
            //得到body里的内容。将buffer转换为string
            auto body_str = boost::beast::buffers_to_string(body.data());
            std::cout << "receive body is " << body_str << std::endl;
            //设置为json
            this->response_.set(boost::beast::http::field::content_type,
"text/json");
            //这是回应给对方用的json
            Json::Value root;
            //用于解析请求
            Json::Reader reader;
            //表示原始的根
            Json::Value src_root;
            //解析到原始的根里
            bool parse_success = reader.parse(body_str, src_root);
            //如果解析错误
            if (!parse_success) {
                std::cout << "Failed to parse JSON data!" << std::endl;
```

```
                root["error"] = 1001;
                //将json序列化为字符串
                std::string jsonstr = root.toStyledString();
                //写道回应里的body里
                boost::beast::ostream(this->response_.body()) << jsonstr;
                return;
            }
            //如果解析正确
            auto email = src_root["email"].asString();
            std::cout << "email is " << email << std::endl;

            root["error"] = 0;
            root["email"] = src_root["email"];
            root["msg"] = "recevie email post success";
            //也就是以字符串的形式显示，其内部还是json格式
            std::string jsonstr = root.toStyledString();
            boost::beast::ostream(this->response_.body()) << jsonstr;
        }
        else
        {
            response_.result(boost::beast::http::status::not_found);
            response_.set(boost::beast::http::field::content_type, "text/plain");
            boost::beast::ostream(response_.body()) << "File not found\r\n";
        }
    }

};

void http_server(boost::asio::ip::tcp::acceptor& acceptor,
boost::asio::ip::tcp::socket& socket) {
    acceptor.async_accept(socket, [&](boost::system::error_code ec) {
        if (!ec) {
            //启动http的connection
            std::make_shared<http_connection>(std::move(socket))->Start();
        }
        //不管成功与失败都要继续监听请求
        http_server(acceptor, socket);
        });
}

int main() {
    try {
        //服务器的地址，最后不要写内网，现在为了测试写内网
        auto const address = boost::asio::ip::make_address("127.0.0.1");
        unsigned short port = static_cast<unsigned short>(8080);
        boost::asio::io_context io_context(1);
        boost::asio::ip::tcp::acceptor acceptor{ io_context, {address, port} };
        boost::asio::ip::tcp::socket socket(io_context);
        http_server(acceptor, socket);
        io_context.run();
    }
    catch (std::exception e) {
        std::cout << "Error: " << e.what() << std::endl;
        return EXIT_FAILURE;
```

```
    }
}
```

beast网络库实现websocket服务器

简介

- 使用beast网络库实现websocket服务器，一般来说websocket是一个长连接的协议，但是自动包含了解包处理，当我们在浏览器输入一个http请求时如果是以ws开头的如ws://127.0.0.1:9501就是请求本地9501端口的websocket服务器处理。而beast为我们提供了websocket的处理方案，我们可以在http服务器的基础上升级协议为websocket，处理部分websocket请求。如果服务器收到的是普通的http请求则按照http请求处理。我们可以从官方文档中按照示例逐步搭建websocket服务器。

构造websocket

- 原文档连接：https://llfc.club/category?catid=225RaiVNI8pFDD5L4m807g7ZwmF#!aid/2RIhdbut49eOVgjIVq9aj6nF7Rg

开发的websocket代码

Connection.h

```
#pragma once
#include <iostream>
#include <boost/beast.hpp>
#include <boost/asio.hpp>
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <queue>
#include <mutex>
#include <string>
#include <condition_variable>
#include <memory>

namespace net = boost::asio;
namespace beast = boost::beast;
class Connection:public std::enable_shared_from_this<Connection>
{
public:
    Connection(net::io_context& ioc);
    //启动连接
    void Start();
    //发送数据
    void Async_Send(std::string data);
    //用于升级为websocket
    void Async_Accept();
    //发送回调函数
    void SendCallBack(std::string msg);
    //得到底层socket
```

```
    net::ip::tcp::socket& GetSocket();
    //得到id
    std::string GetUuid();
private:
    //指向websocket的智能指针
    std::unique_ptr<beast::websocket::stream<beast::tcp_stream>> _ws_ptr;
    //保证时序性的队列
    std::queue<std::string> _send_que;
    net::io_context& _ioc;
    //websocket的缓冲区数据
    beast::flat_buffer _recv_buffer;
    std::mutex mtx;
    std::condition_variable cv;
    std::string _uuid;
};
```

Connection.cpp

```
#include "Connection.h"
#include "ConnectionMgr.h"
//使用strand来让处理变成串行
Connection::Connection(net::io_context& ioc):_ioc(ioc),
_ws_ptr(std::make_unique<beast::websocket::stream<beast::tcp_stream>>
(net::make_strand(ioc)))
{
    boost::uuids::random_generator grenerator;
    boost::uuids::uuid uuid = grenerator();

    //转换为string
    _uuid = boost::uuids::to_string(uuid);
}

void Connection::Start()
{
    //增加有引用计数
    auto self = shared_from_this();
    //先接受数据
    _ws_ptr->async_read(_recv_buffer, [self](boost::system::error_code ec,
std::size_t t) {
        try {fer.size());
            if (ec) {
                std::cout << "websocket async read error is " << ec.what() <<
std::endl;
                ConnectionMgr::GetInstance().Remove_Connection(self->GetUuid());
                return;
            }
            //用于创建文本消息数据帧，是用来发送的
            self->_ws_ptr->text(self->_ws_ptr->got_text());
            //将缓冲区数据转为string格式
```

```
            //.data是为了得到起始位置的指针
            std::string recv_data = boost::beast::buffers_to_string(self-
>_recv_buffer.data());
            //清空缓冲区，方便下次接收
            self->_recv_buffer.consume(self->_recv_buffer.size());
            std::cout << "Recevie data is " << recv_data << std::endl;
            //发送回去
            self->Async_Send(std::move(recv_data));
            //继续接收数据
            self->Start();
        }
        catch (std::exception& e) {
            std::cerr << "Exception is " << e.what() << std::endl;
        }
        });
}

void Connection::Async_Send(std::string data)
{
    try {
        {
            std::lock_guard<std::mutex> lock(mtx);
            int que_len = _send_que.size();
            _send_que.push(data);
            if (que_len > 0) return;
        }
        SendCallBack(std::move(data));
    }
    catch (std::exception& e) {
        std::cerr << "Async Send failed, Exception is " << e.what() << std::endl;
    }
}

void Connection::Async_Accept()
{
    auto self = shared_from_this();
    _ws_ptr->async_accept([self](boost::system::error_code err) {
        try {
            if (!err) {
                ConnectionMgr::GetInstance().Add_Connection(self);
                self->Start();
            }
            else {
                std::cout << "websocket accept failed, err is " << err.what() <<
std::endl;
            }
        }
        catch (std::exception& exp) {
            std::cout << "websocket async accept exception is " << exp.what();
        }
        });
}

void Connection::SendCallBack(std::string msg)
```

```
{
    auto self = shared_from_this();
    //发送数据给客户端
    _ws_ptr->async_write(boost::asio::buffer(msg.c_str(), msg.length()),
        [self](boost::system::error_code err, std::size_t  nsize) {
            try {
                if (err) {
                    std::cout << "async send err is " << err.what() << std::endl;
                    ConnectionMgr::GetInstance().Remove_Connection(self->_uuid);
                    return;
                }

                std::string send_msg;
                {
                    std::lock_guard<std::mutex> lck_gurad(self->mtx);
                    self->_send_que.pop();
                    //如果数据空了就返回
                    if (self->_send_que.empty()) {
                        return;
                    }
                    send_msg = self->_send_que.front();
                }

                self->SendCallBack(std::move(send_msg));
            }
            catch (std::exception& exp) {
                std::cout << "async send exception is " << exp.what() <<
std::endl;
                ConnectionMgr::GetInstance().Remove_Connection(self->_uuid);
            }
        });
}

net::ip::tcp::socket& Connection::GetSocket()
{
    auto& con_ptr = beast::get_lowest_layer(*_ws_ptr).socket();
    return con_ptr;
    // TODO: 在此处插入 return 语句
}

std::string Connection::GetUuid()
{
    return _uuid;
}
```

ConnectionMgr.h

```
#pragma once
#include <unordered_map>
#include "Connection.h"
```

```
class ConnectionMgr
{
public:
    static ConnectionMgr& GetInstance();
    void Remove_Connection(std::string uuid);
    void Add_Connection(std::shared_ptr<Connection> con);
private:
    ConnectionMgr();
    ConnectionMgr(const ConnectionMgr&) = delete;
    ConnectionMgr& operator=(const ConnectionMgr&) = delete;
    //存储连接
    std::unordered_map<std::string, std::shared_ptr<Connection>> _cons;
};
```

ConnectionMgr.cpp

```
#include "ConnectionMgr.h"

ConnectionMgr& ConnectionMgr::GetInstance()
{
    static ConnectionMgr ins;
    return ins;
}

void ConnectionMgr::Remove_Connection(std::string uuid)
{
    _cons.erase(uuid);
}

void ConnectionMgr::Add_Connection(std::shared_ptr<Connection> con)
{
    if (_cons.count(con->GetUuid()) == 1) return;
    _cons[con->GetUuid()] = con;
}

//这个必须写
ConnectionMgr::ConnectionMgr()
{

}
```

WebSocketServer.h

```
#pragma once
#include "ConnectionMgr.h"
```

```
namespace net = boost::asio;
namespace beast = boost::beast;
class WebSocketServer
{
public:
    WebSocketServer(net::io_context& ioc, unsigned short port);
    void StartAccept();
private:
    WebSocketServer(const WebSocketServer&) = delete;
    WebSocketServer& operator=(const WebSocketServer&) = delete;

    net::ip::tcp::acceptor _acceptor;
    net::io_context& _io_context;
};
```

WebSocketServer.cpp

```
#include "WebSocketServer.h"
#include "Connection.h"
void WebSocketServer::StartAccept()
{
    try {
        auto con_ptr = std::make_shared<Connection>(_io_context);
        //监听连接
        _acceptor.async_accept(con_ptr->GetSocket(), [this, con_ptr]
(boost::system::error_code ec) {
            if (ec) {
                std::cout << "accept failed, error is " << ec.what() << std::endl;
                return;
            }
            //升级http为websocket
            con_ptr->Async_Accept();
            //继续监听连接
            StartAccept();
            });
    }
    catch (std::exception& e) {
        std::cerr << "Exception is " << e.what() << std::endl;
    }
}

WebSocketServer::WebSocketServer(net::io_context& ioc, unsigned short
port):_io_context(ioc),
_acceptor(ioc, net::ip::tcp::endpoint(net::ip::tcp::v4(), port))
{
    std::cout << "Server start on port : " << port << std::endl;
}
```

main.cpp

```
#include "WebSocketServer.h"

int main()
{
    net::io_context ioc;
    WebSocketServer server(ioc, 8888);
    server.StartAccept();
    ioc.run();
}
```

总结

- 头文件预编译的时候不要出现父子的情况，也就是说beast.hpp 包含了 beast/core，两个一起被编译就会出现问题
- 一旦声明构造函数，就要显示写出构造函数，因为他不会提供默认构造函数