

----- Ep5 -----

》》》什么是 boost 中的 Beast 库? 有什么作用?

HTTP 协议支持:	◦ Boost.Beast 提供了对 HTTP 请求和响应的构建、解析和传输的支持。它支持常见的 HTTP 方法 (如 GET、POST、PUT、DELETE) 以及 HTTP 标头、消息体等内容的处理。 ◦ 它其中包含的方法能够用于构建 HTTP 客户端和服务端, 并且支持同步和异步 I/O 操作。
WebSocket 协议支持:	◦ Boost.Beast 还提供了对 WebSocket 协议的支持, 可以用来创建 WebSocket 客户端和服务端。 WebSocket 是一种全双工通信协议, 广泛用于实时应用 (例如聊天室、实时通知系统等)。 ◦ 它可以帮助开发者轻松处理 WebSocket 握手、消息发送和接收等操作。
与 Boost.Asio 的集成:	Boost.Beast 构建在 Boost.Asio 库之上, 后者是一个用于异步 I/O 操作的高性能库。 通过这种集成, Boost.Beast 可以轻松处理基于 I/O 的任务, 如接收和发送数据, 适用于高效的网络服务和客户端应用。
高效的处理机制:	Boost.Beast 还使用了现代 C++ 特性 (如模板、智能指针、移动语义等), 这能够提供高效的内存管理和性能表现, 适合需要高性能和低延迟的网络应用。

》》》什么是全双工? 半双工? 单工? (关于设备能否进行传播和接收、传播或接收能否同时发生的问题)

1. 全双工 (Full-Duplex)	全双工通信指的是在同一时间内, 通信双方可以同时进行双向数据传输。也就是说, 双方可以在同一时间既发送又接收数据。
特点:	◦ 同时发送和接收数据。 ◦ 双向通信不互相干扰。 ◦ 需要独立的信号通道来实现同时的发送和接收。
例子:	• 电话通信: 在电话通话过程中, 双方可以同时说话和听到对方的声音。 • 现代计算机网络: 例如以太网, 在全双工模式下, 数据可以同时发送和接收。 • 无线通信 (如 Wi-Fi): 现代的无线设备也支持全双工通信。

2. 半双工 (Half-Duplex)	半双工通信指的是通信双方在同一时间内只能单向传输数据。在一个时间段内, 数据只能在一个方向上传输, 另一方只能接收数据, 无法同时发送。 要实现双向通信, 设备需要切换方向。
特点:	◦ 在任意时刻只能发送或接收数据, 不能同时进行。 ◦ 数据流是单向的, 传输方向可以改变, 但不能同时改变。
例子:	◦ 对讲机: 对讲机是典型的半双工通信设备, 一个用户按下按钮讲话时, 另一方只能接收, 直到用户松开按钮才能接收或发送。 ◦ 无线电通信: 在许多无线电系统中, 广播和接收这两个操作只能是交替进行的。

3. 单工 (Simplex)	单工通信是一种通信模式, 在这种模式下, 数据只能沿着一个方向传输, 即只能从发送方到接收方, 不允许反向传输。 可以将其视为全双工和半双工的“极端”情况。
特点:	◦ 数据只能单向传输, 接收方无法发送任何反馈。 ◦ 通常只用于一些简单的数据传输场景。
例子:	◦ 电视广播: 电视台只会将信号广播到所有观众, 观众无法将信号发送回广播站。 ◦ 收音机: 收音机只能接收广播电台的信号, 无法反向传输数据。

》》》using tcp = boost::asio::ip::tcp; 和 namespace tcp = boost::asio::ip::tcp; 有什么不同? 可以代替使用吗?

特性	namespace tcp = boost::asio::ip::tcp;	using tcp = boost::asio::ip::tcp;
作用	为整个命名空间创建别名	为类型或类创建别名
用法	用于简化命名空间成员的访问	用于简化类型名的使用
访问方式	tcp:: 替代 boost::asio::ip::tcp::	tcp:: 替代 boost::asio::ip::tcp::
适用范围	适用于命名空间 (可以有多个成员)	适用于具体的类型 (如类、结构体、函数指针、模板等)

我们可以打开文件进行查阅, 可以看到 tcp 明显是一个类型, 可以查阅到相关定义:

》》》 async_accept 函数参数中填入的 lambda 表达式，其中 return; 如果被调用，会导致什么？

```
m_Acceptor.async_accept(m_Socket, [this](boost::beast::error_code ec)
{
    try
    {
        // 如果当前连接出错，则放弃该连接，并且继续监听新连接
        if(ec)
        {
            this->Start();
            return;
        }

        // 如果连接正常，则使用 HpptConnection 处理连接
        std::make_shared<HttpConnection>();
        this->Start();

        catch(std::exception& ex)
        {
            JC_CORE_ERROR("Exception is {}", ex.what());
            this->Start();
        }
    }
};
```

在这个代码段中，return 语句只会导致 当前回调函数 的终止，而 不会导致整个程序终止。
这是因为你在 Lambda 回调函数内部使用了 return，它只是控制 Lambda 函数的流，结束当前的回调执行。

》》》 shared_from_this 的作用

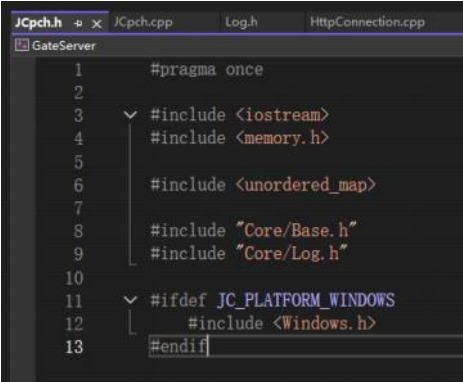
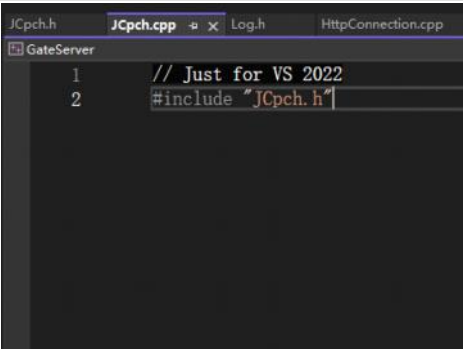
在异步回调中，你需要确保 CServer 对象在回调函数执行期间是有效的。如果你直接使用 this 指针，而此时 CServer 对象已经在别的地方被销毁或超出作用域，那么会导致 this 指针指向无效的内存地址，造成悬挂指针问题。

使用 shared_from_this() 通过智能指针来管理对象生命周期，这样即使 Start() 函数返回并进入异步回调，CServer 对象也会因为 shared_ptr 的引用计数而保持有效。这样就避免了对象被提前销毁的问题。

》》》 const.h 设置为预编译头文件

我没有选择将 const.h 仅仅作为一个全局的头文件来使用，我想将其作为一个预编译头文件：Precompile header，所以我这样操作：

在 VS 2022 中创建两个文件：

.h	 <pre>1 #pragma once 2 3 #include <iostream> 4 #include <memory.h> 5 6 #include <unordered_map> 7 8 #include "Core/Base.h" 9 #include "Core/Log.h" 10 11 #ifdef JC_PLATFORM_WINDOWS 12 #include <Windows.h> 13 #endif</pre>
.cpp	 <pre>1 // Just for VS 2022 2 #include "Jcpch.h"</pre>

然后在 premake 脚本中做声明：

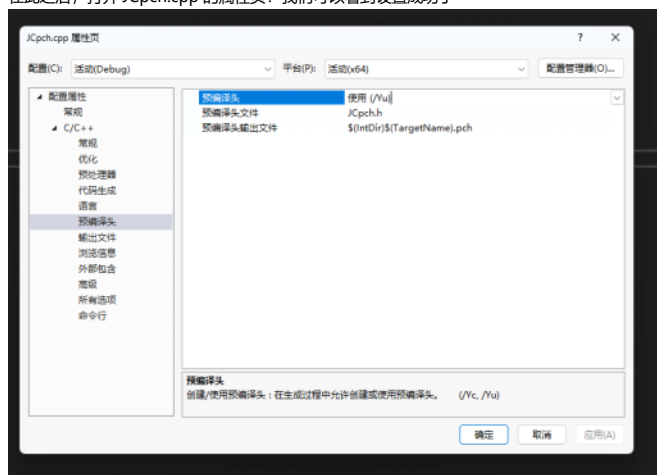
```
16 project "GateServer"
17     location "GateServer"
18     kind "ConsoleApp"
19     language "C++"
20     --cpddialect "C++17"           --C++标准（编译时）
21
22     targetdir ("%bin/%{cfg.buildcfg}-%{cfg.system}-%{cfg.architecture}%{prj.name}")
23     objdir ("%bin-int/%{cfg.buildcfg}-%{cfg.system}-%{cfg.architecture}%{prj.name}")
24
25     pchheader "JCpch.h"
26     pchsource "%{prj.name}/src/JCpch.cpp"
27
```

我们还需要为包含库目录添加 src，以便编译器在该文件夹中直接匹配 pch.h。（而不是 src/JCpch.h）

```
24
25     pchheader "JCpch.h"
26     pchsource "%{prj.name}/src/JCpch.cpp"
27
28     -- 宏定义
29     defines
30     {
31
32     }
33
34     -- 添加源文件
35     files
36     {
37         "%{prj.name}/src/**/*.h",
38         "%{prj.name}/src/**/*.cpp"
39     }
40
41     -- 添加包含目录
42     includedirs
43     {
44         "%{prj.name}/src",
45         "%{prj.name}/vendor/Boost1_88_0/include",
46         "%{prj.name}/vendor/Jsoncpp_1_9_6/include",
47         "%{prj.name}/vendor/Spdlog_1_15_3/include"
48     }
49
```

接下来运行 .bat 文件，并重载项目。

在此之后，打开 JCpch.cpp 的属性页：我们可以看到设置成功了



我们补全一下预编译头文件，Now it's finished

```
#pragma once

#include <iostream>
#include <memory.h>
#include <unordered_map>

#include <boost/beast/http.hpp>
#include <boost/beast.hpp>
#include <boost/asio.hpp>

#include "Core/Base.h"
#include "Core/Log.h"

namespace beast = boost::beast;
namespace http = boost::beast::http;
namespace net = boost::asio;
using tcp = boost::asio::ip::tcp;

#ifdef JC_PLATFORM_WINDOWS
#include <Windows.h>
#endif
```

在运行之前，请确保项目的每一个.cpp在最开始都包含了 JCpch.h 文件，否则会报错。

关于预编译头文件，正常的使用逻辑是包含在 .cpp 文件中，不需要包含在 .h 文件中，如果你在 .cpp 文件中正确地包含了预编译头文件，即使头文件中没有包含它，编译时也可以识别文件中的相关的声明。

示例：即使没有在 Cserver.h 中包含预编译头文件，照样能够识别。

```

#pragma once

class CServer : public std::enable_shared_from_this<CServer>
{
public:
    CServer(boost::asio::io_context& context, unsigned short& port);

    void Start(); // 用于监听链接, 并对新链接进行处理
private:
    boost::asio::io_context& m_Context;
    boost::asio::ip::tcp::acceptor m_Acceptor;
    boost::asio::ip::tcp::socket m_Socket;
};

```

```

#include "Jcpch.h"
#include "CServer.h"
#include "HttpConnection.h"

CServer::CServer(boost::asio::io_context& context, unsigned short& port)
    : m_Context(context), m_Acceptor(context, boost::asio::ip::tcp::endpoint(tcp::v4(), port)), m_Socket(context)
{
}

void CServer::Start()
{
    auto self = shared_from_this();
    m_Acceptor.async_accept(m_Socket, [self](boost::beast::error_code ec)
    {
        try
        {
            // 如果当前连接出错, 则放弃该连接, 并且继续监听新连接
            if(ec)
            {
                self->Start();
                return;
            }

            // 如果连接正常, 则使用 HttpConnection 处理连接
            std::make_shared<HttpConnection>(self->m_Socket)->Start();
            self->Start();
        }
        catch(std::exception& ex)
        {
            JC_CORE_ERROR("Exception is {}, ex.what()", ex.what());
            self->Start();
        }
    });
}

```

》》》一个疑问:

```

11 void CServer::Start()
12 {
13     auto self = shared_from_this();
14     m_Acceptor.async_accept(m_Socket, [self](boost::beast::error_code ec)
15     {
16         try
17         {
18             // 如果当前连接出错, 则放弃该连接, 并且继续监听新连接
19             if(ec)
20             {
21                 self->Start();
22                 return;
23             }
24
25             // 如果连接正常, 则使用 HttpConnection 处理连接
26             std::make_shared<HttpConnection>(self->m_Socket)->Start();
27             self->Start();
28         }
29         catch(std::exception& ex)
30         {
31             JC_CORE_ERROR("Exception is {}, ex.what()", ex.what());
32             self->Start();
33         }
34     });
35 }
36
37

```

这个函数难道会一直进行监听操作, 直至程序被终止吗? 为什么? (因为他进行的是监听操作吗?)

》》》什么是左值, 什么是右值?

左值 (Lvalue) :

特点:	<ul style="list-style-type: none"> 可以被赋值。 在程序中有明确的内存地址。即 --> 左值是可寻址的 (例如, 变量名就是一个左值)。
定义:	左值是指可以出现在赋值符号 "=" 左边的表达式。简单来说, 左值是有持久位置的对象, 可以通过其地址进行访问。

示例:	<code>int x = 10; // x 是左值</code> <code>x = 20; // x 作为左值出现在赋值左边</code>
-----	--

右值 (Rvalue) :

定义:	右值是指不能出现在赋值符号 “=” 左边的表达式，通常是临时对象或即将销毁的对象。
特点:	<ul style="list-style-type: none">右值是没有持久地址的。右值通常表示临时值，不能对其进行修改（不能取地址）。右值通常是表达式的结果，如常量或函数返回的临时值。
示例:	<code>int x = 10;</code> <code>int y = x + 5; // x + 5 是右值</code>

左值引用:

左值引用:	<code>int&</code> 是左值引用类型，表示对一个左值的引用。
示例:	<code>int a = 5;</code> <code>int& b = a; // b 是 a 的左值引用</code>

右值引用:

右值引用:	<code>int&&</code> 是右值引用类型，表示对一个右值的引用。右值引用允许我们移动资源而不是复制它们。
示例:	<code>int&& c = 10; // c 是右值引用，绑定到右值 10</code>

》》》std::move ?



std::move 可以将左值转为右值吗？

是的，std::move 可以将左值转为右值。

std::move 本身并不会移动任何数据，它的作用只是将一个左值转换为右值引用，从而允许将这个对象的资源转移（移动）给另一个对象。这表明我们想要“移动”这个对象的资源，而不是拷贝它。

示例:	<code>int x = 10;</code> <code>int&& y = std::move(x); // std::move 将左值 x 转为右值引用 y</code>
-----	--

为什么需要 std::move？

在 C++ 中，移动语义允许资源（如动态内存或文件句柄）从一个对象转移到另一个对象，而不是进行昂贵的拷贝操作。std::move 是实现这一机制的工具，告诉编译器这个对象不再需要，且其资源可以安全地转移。

示例:	<code>HttpConnection::HttpConnection(boost::asio::ip::tcp::socket socket)</code> <code>: m_Socket(std::move(socket)) // 使用 std::move, 将 socket 的资源转移到 m_Socket</code> <code>{</code> <code>}</code> <code>boost::asio::ip::tcp::socket</code> 是一个非拷贝类型（禁止拷贝构造和拷贝赋值），也就是说它不允许被拷贝。 如果你尝试直接将 <code>socket</code> 赋值给 <code>m_Socket</code> 而不使用 <code>std::move</code> ，编译器会报错，因为它无法进行拷贝。 通过 std::move 转移资源： <code>std::move(socket)</code> 将 <code>socket</code> 转换成右值引用，允许编译器将 <code>socket</code> 的内部资源（如缓冲区、网络连接等）转移给 <code>m_Socket</code> ，而不是进行拷贝。 这样可以避免不必要的资源复制，并且能够正确地初始化 <code>m_Socket</code> 。
-----	--

》》》复制与移动

复制 (Copying) :

定义:	复制是将一个对象的内容拷贝到另一个对象中。通常，复制会创建对象的一个副本，并且源对象和目标对象各自拥有自己的资源。
过程:	当进行复制时，原始对象的内容（如内存、数据等）会被逐个拷贝到新的内存位置。 复制操作会导致开销，特别是当对象包含动态分配的资源（如指针、文件句柄等）时。
示例:	<code>std::vector<int> vec1 = {1, 2, 3};</code> <code>std::vector<int> vec2 = vec1; // 复制, vec2 拷贝了 vec1 的所有元素</code>

移动 (Moving / 资源转移) :

定义:	移动是将一个对象的资源（如内存、指针等）从一个对象转移到另一个对象，而不是复制这些资源。移动后的原对象通常不再拥有这些资源，移动操作不会对这些资源进行复制，而是直接“转移”它们的所有权。
具体操作:	在移动操作中，原对象持有某些资源（例如动态分配的内存、文件句柄等），而目标对象需要接管这些资源。移动操作通常通过直接传递资源的地址或指针来实现这一转移。简而言之，原对象的指针（内存地址）会直接赋给目标对象。 例如，如果一个对象内有一个指向动态分配内存的指针，移动操作会将该指针直接转移到新对象，而原对象的指针会被置为 <code>nullptr</code> 或清空，从而避免资源的重复释放。
过程:	移动操作的关键是资源的所有权转移，原对象会被标记为“空”或“无效”状态，而目标对象获得资源的所有权。移动通常比复制更高效，因为它避免了不必要的资源分配和复制。
示例:	<code>std::vector<int> vec1 = {1, 2, 3};</code>


```
std::vector<int> vec2 = std::move(vec1); // 移动, vec2 获得 vec1 的资源, vec1 变为空状态
```

移动和复制的关系：

复制：	拷贝资源，两个对象各自拥有独立的资源副本，可能需要额外的内存分配。
移动：	转移资源的所有权，一个对象不再拥有资源，另一个对象获得所有权，通常不需要额外的内存分配或数据拷贝。

举例：（移动和复制的关系：）

用我的话来讲，复制就是：	我有一套房子，如果你想使用我的房子做一些事情，则需要你自己去买一套一模一样的房子，并可由你自己进行装潢。
而移动则是：	我有一套房子，如果你想使用我的房子做一些事情，我直接把房产证给你，在拥有者这一栏划掉我的名字，并写上你的名字，转移所有权，然后房间任你处置。

```
》》》注意：这里使用初始化列表对 buffer 进行初始化，使用的是 "{}" 而不是 "[]"。

#pragma once

class HttpConnection : public std::enable_shared_from_this<HttpConnection>
{
public:
    HttpConnection(boost::asio::ip::tcp::socket socket);

    void Start();
private:
    void HandleReq();
    void WriteResponse();
    void CheckDeadline();
private:
    boost::asio::ip::tcp::socket m_Socket;

    boost::beast::flat_buffer m_Buffer{ 8192 };

    () namespace boost {
        uest<http::dynamic_body> m_Request;
        onse<http::dynamic_body> m_Response;
        er m_Decline(m_Socket.get_executor(), std::chrono::seconds(60));
    };
};
```

》》》boost::beast::http::request<http::dynamic_body> m_Request;
》》》boost::beast::http::response<http::dynamic_body> m_Response;
》》》的作用？

1. boost::beast::http::request<http::dynamic_body> m_Request	<ul style="list-style-type: none">• boost::beast::http::request<http::dynamic_body> 是 Boost.Beast 提供的用于表示 HTTP 请求的模板类。• http::dynamic_body 表示请求的 请求体，其内容可以是任意格式的数据，dynamic_body 允许处理大小可变的请求体（如 JSON、HTML、文件上传等）。它的主要作用是处理请求的正文内容，例如通过 POST 或 PUT 请求发送的数据。
作用：	<ul style="list-style-type: none">• m Request 是一个 HTTP 请求对象，它存储了来自客户端的 HTTP 请求信息，代表一个客户端发送到服务器的 HTTP 请求。包括请求方法（GET、POST、PUT 等）、头部信息和请求体。• 它可用于发送请求、获取响应、或者在服务器端接收并处理客户端的请求。

2. boost::beast::http::response<http::dynamic_body> m_Response	<ul style="list-style-type: none">• boost::beast::http::response<http::dynamic_body> 是 Boost.Beast 提供的用于表示 HTTP 响应的模板类。• http::dynamic_body 同样表示响应的 响应体。这使得响应体可以处理大小可变的数据（如文本、图片、视频、JSON 等）。
作用：	<ul style="list-style-type: none">• m Response 是一个 HTTP 响应对象。它存储了服务器返回给客户端的 HTTP 响应信息，代表服务器向客户端发送的 HTTP 响应。这个响应包含了状态码、响应头以及响应体。• 它可以用来在服务器端创建并发送响应，或者在客户端接收并处理响应。

具体应用场景

在服务器端：

1. 接收 HTTP 请求：服务器通过 m_Request 来接收客户端发送的请求，提取出请求头、请求方法和请求体等内容。
2. 处理请求并生成响应：服务器根据接收到的请求内容，通过 m_Response 构造一个适当的 HTTP 响应（例如返回数据、状态码等），将响应返回给客户端。

》》》什么是 Boost::ignore_unused？有什么作用？

boost::ignore_unused 是 Boost 库中的一个工具，用于标记未使用的变量，以消除编译器的警告信息。

作用：
在 C++ 编程中，如果你声明了一个变量，但在代码中并未使用它，编译器通常会发出一个警告。这种警告通常是为了提醒开发者：声明了一个变量但是没有使用它，可能是代码中存在逻辑错误或者冗余。有时候，出于某些需求（例如接口的通用性或者模板编程），你可能需要在某些情况下声明变量，但又不一定会使用它。这时候，boost::ignore_unused 可以用来避免这些警告。

示例：

<p>在 <code>HttpConnection</code> 的成员函数 <code>start</code> 进行设计的时候，我们发现这个函数 <code>boost::ignore_unused</code></p> <pre>void HttpConnection::Start() { auto self = shared_from_this(); http::async_read(_socket, _buffer, _request, [self](beast::error_code ec, std::size_t bytes_transferred) { try { if (ec) { std::cout << "http read err is " << ec.what() << std::endl; return; } //处理读到的数据 boost::ignore_unused(bytes_transferred); self->HandleReq(); self->CheckDeadline(); } catch (std::exception& exp) { std::cout << "exception is " << exp.what() << std::endl; } } };</pre>	<p>如果你不打算在回调中使用这个值，就没有必要将它传递给回调函数，直接去掉它是没有问题的。你删除后的代码逻辑如下：</p> <pre>void HttpConnection::Start() { auto self = shared_from_this(); http::async_read(_socket, _buffer, _request, [self](beast::error_code ec) { try { if (ec) { std::cout << "http read err is " << ec.what() << std::endl; return; } //处理读到的数据 self->HandleReq(); self->CheckDeadline(); } catch (std::exception& exp) { std::cout << "exception is " << exp.what() << std::endl; } });</pre>
---	---

》》》关于 `http::async_read` 数据流通的一些解释

```
void HttpConnection::Start()
{
    auto self = shared_from_this();
    boost::beast::http::async_read(m_Socket, m_Buffer, m_Request,
        [self](boost::beast::error_code ec, std::size_t bytesTransferred)
        {
            try
```

请求数据的流程

1. 客户端发送请求：客户端通过 TCP 连接向服务器发送 HTTP 请求（如 GET /index.html HTTP/1.1）。
2. 接收数据： `async_read` 异步地从 `_socket` 中读取数据并存入 `_buffer` 缓冲区。
3. 解析数据：在读取完成后， `Boost.Beast` 自动解析 `_buffer` 中的数据，并将解析结果存储到 `_request` 中。
4. 调用回调函数：一旦数据成功读取并解析，回调函数中的代码会被调用， `self->HandleReq()` 会进一步处理请求。

所以我们大部分数据是从 `socket` 中传递过来的，而 `socket` 由 `CServer` 初始化

```
CServer::CServer(boost::asio::io_context& context, unsigned short& port)
: m_Context(context), m_Acceptor(context, boost::asio::ip::tcp::endpoint(tcp::v4(), port)), m_Socket(context)
{
    // ...
}
```

》》》 `boost::beast::http::request` 的成员函数 `target()` 是什么作用？

request.target() 是 `Boost.Beast` 库中 `http::request` 类的一个成员函数。它用于获取 HTTP 请求的目标（即请求的路径和查询字符串部分）。

具体来说：	<ul style="list-style-type: none">• <code>http::request</code> 类表示一个 HTTP 请求，其中包含了 HTTP 请求的各种信息，如请求方法、目标、请求头等。• <code>target()</code> 函数返回的是 HTTP 请求的目标路径，也就是 URL 中的路径部分。它不包括协议（如 <code>http://</code>）和主机名（如 <code>example.com</code>），但是包括路径和查询字符串（如果有的话）。
返回内容：	<ul style="list-style-type: none">• 类型： <code>target()</code> 返回一个 <code>boost::beast::string_view</code>，这实际上是一个轻量级的、只读的字符串视图，指向 HTTP 请求中的路径部分。• 值： 它返回的值通常是类似 <code>/index.html</code>、<code>/api/v1/resource?param=value</code> 这样的字符串，代表 HTTP 请求的目标资源。
示例：	<p>假设客户端发起了以下 HTTP 请求：</p> <pre>GET /path/to/resource?query=1 HTTP/1.1 Host: example.com ...</pre> <p>在这个例子中， <code>request.target()</code> 将返回 <code>/path/to/resource?query=1</code>。</p>
重要性？	在处理 HTTP 请求时，通常需要获取请求的目标路径来进行路由或资源定位。例如，Web 服务器根据请求的目标路径来决定返回哪个资源或执行哪种操作。所以 <code>target()</code> 获取的值挺重要。


```

    >>>> response.result
    >>>> response.set
    >>>> beast::ostream

```

1. _response.result(http::status::not_found);

这个函数设置 HTTP 响应的状态码，指示请求的结果。
result 是 Boost.Beast 中 http::response 类的成员函数。HTTP 响应的状态码告诉客户端请求的处理结果。状态码可以分为不同的类别，如：

1xx（信息性状态码）：	表示请求已经被接受，处理正在进行。
2xx（成功状态码）：	表示请求成功，如 200 OK。
3xx（重定向状态码）：	表示需要客户端进一步操作，例如 301 Moved Permanently。
4xx（客户端错误状态码）：	表示客户端请求有误，404 Not Found 就属于这一类，表示服务器无法找到请求的资源。
5xx（服务器错误状态码）：	表示服务器内部错误，如 500 Internal Server Error。

当你使用 _response.result(http::status::not_found) 时，你告诉服务器要返回一个 404 Not Found 状态码，意味着请求的资源没有在服务器上找到。

2. _response.set(http::field::content_type, "text/plain");

这个函数设置 HTTP 响应头部字段。HTTP 响应头包含了关于响应的元信息，如内容类型、长度、服务器信息等。通过 set() 函数，你可以在 HTTP 响应中添加或修改某个字段的值。

在这个例子中，_response.set(http::field::content_type, "text/plain"); 设置了响应头中的 Content-Type 字段，告诉客户端服务器返回的是纯文本格式的数据。
Content-Type 是响应头中最常用的字段之一，它告诉客户端如何解释响应体的内容。常见的 Content-Type 类型包括：

text/html：	表示响应内容是 HTML 文档。
application/json：	表示响应内容是 JSON 格式。
image/png：	表示响应内容是 PNG 图像。
text/plain：	表示响应内容是纯文本。

text/plain 格式意味着响应体中的内容没有任何格式化，它是普通的文本数据。对于 404 Not Found 响应，通常返回纯文本说明请求资源没有找到。

3. beast::ostream(_response.body()) << "url not found\r\n";

这个函数通过输出流将数据写入 HTTP 响应体（body）。beast::ostream 是 Boost.Beast 提供的一个输出流类，它用于向 HTTP 响应的 body 中写入数据。
当客户端请求一个不存在的 URL 时，服务器通常返回一个包含错误信息的响应体。在这个例子中，"url not found\r\n" 被写入到响应体中。这会直接告诉客户端，所请求的资源没有找到。

beast::ostream 是一个与标准 C++ 输出流 (std::ostream) 类似的接口，它支持将数据流写入缓冲区。写入内容时，\r\n 是常见的换行符，它是 HTTP 响应中文本内容的标准格式。
除了简单的文本数据，你还可以将 JSON、HTML 或其他格式的内容写入响应体。beast::ostream 提供了简单而灵活的方式来处理 HTTP 响应的内容。

>>>> 处理 HTTP 请求时，必须进行的操作包括：
 >>>> 处理 HTTP 请求时，必须进行的操作包括：

1. 设置 HTTP 响应状态码。
2. 设置响应头部（如 Content-Type、Content-Length 等）。
3. 生成响应体（实际返回的内容）。
4. 处理错误和异常，返回适当的错误码和信息。
5. （视情况）处理 CORS。
6. （视情况）设置 Cookie。
7. 设置缓存控制。
8. （视情况）确保安全性，如使用 HTTPS 和设置相关的安全头部。
9. 返回正确格式的内容。

>>>> 什么是短连接，什么是长连接？

1. 短连接（Short Connection）

特点	<ul style="list-style-type: none"> • 每次通信都建立新连接，完成后立即关闭。 • 适用于 请求-响应模式（如 HTTP/1.0）。 • 每次请求都需要 三次握手（建立连接）和四次挥手（关闭连接）。
工作流程	1. 客户端 发起 TCP 连接（三次握手）。 2. 客户端 发送请求，服务器 返回响应。 3. 连接立即关闭（四次挥手）。 4. 下次请求时，重新建立连接。
优点	<ul style="list-style-type: none"> • 实现简单，服务器不需要维护连接状态。 • 适合低频请求（如网页浏览、传统 API 调用）。
缺点	<ul style="list-style-type: none"> • 频繁建立/关闭连接，性能开销大（三次握手、四次挥手耗时）。

	• 高并发时服务器压力大（每个请求都要新建连接）。
典型应用	• HTTP/1.0（默认短连接）。 • 简单的 REST API 请求。

2. 长连接（Long Connection）

特点	• 一次连接，多次通信，完成后不会立即关闭。 • 适用于 持续交互（如 WebSocket、数据库连接）。 • 减少握手开销，提高性能。
工作流程	1. 客户端 发起 TCP 连接（三次握手）。 2. 客户端 和 服务器 可以 多次交换数据。 3. 连接保持，直到超时或主动关闭。 4. 下次请求 复用同一个连接。
优点	• 减少 TCP 握手/挥手开销，提高效率。 • 适合高频请求（如实时通信、游戏、数据库访问）。 • 降低服务器负载（减少连接数）。
缺点	• 服务器需要维护连接状态（可能占用更多内存）。 • 需要心跳机制（防止连接被误杀）。
典型应用	• HTTP/1.1（Keep-Alive）（默认复用连接）。 • WebSocket（全双工长连接）。 • MySQL/Redis 数据库连接池。 • 实时通信（如聊天、直播）。

》》》》 POST 请求和 GET 请求

GET GET GET

GET请求：	GET请求是用来从服务器获取数据的请求。它是一个无副作用的请求，即不会修改服务器上的资源。 GET请求通常用于请求数据或者获取某些信息，并且请求参数通常会通过URL传递。
操作：	• 获取资源：通常用于请求页面内容、查询数据库中的数据、获取图片等静态资源。 • 传递参数：GET请求将请求的参数附加在URL后面，通常以?开头，多个参数之间用&连接。例如：https://example.com/api?name=John&id=123.
GET请求的特点：	1. 参数通过URL传递，请求体为空。 2. 请求内容可以被缓存。 3. 浏览器可以书签GET请求，也就是说，GET请求的URL可以被保存并稍后再次访问。 4. 请求参数有限制：由于GET请求的参数是附加在URL后的，所以URL长度有限制，通常为2048个字符左右。 5. 无副作用：GET请求通常是只读取数据，不修改服务器上的任何资源。
客户端发起GET请求后，服务器一般的处理：	• 服务器接收到GET请求后，会解析URL中附带的参数。 • 然后根据请求的URL和参数，查找相应的资源或执行对应的查询操作。 • 最终，服务器将资源返回给客户端（如网页、图片、JSON数据等）。
处理流程简要：	1. 客户端发送GET请求。 2. 服务器解析请求并获取资源。 3. 服务器返回请求的资源或响应数据。

POST POST POST

POST请求：	POST请求用于向服务器发送数据，通常用于提交表单数据或上传文件等操作。 与GET不同，POST请求会将请求数据包含在请求体中，而不是URL中。
操作：	• 提交数据：例如，提交用户的表单数据、登录请求、注册信息等。 • 修改服务器资源：POST请求会对服务器上的资源进行操作，可能会修改数据、存储数据、上传文件等。
POST请求的特点：	1. 参数通过请求体传递，不会显示在URL中，传输更为安全。 2. 请求内容不会被缓存。 3. 无长度限制：POST请求的数据量没有像GET请求那样的URL长度限制，适用于大数据的传输。 4. 可能引起副作用：POST请求一般用于向服务器提交数据，可能会修改服务器上的资源（例如，创建新记录、更新数据等）。
客户端发起POST请求后，服务器一般的处理：	• 服务器接收到POST请求后，会解析请求体中的数据（例如，表单数据、JSON数据等）。 • 根据数据内容，服务器可能会修改数据库或执行其他操作。 • 服务器根据操作的结果，返回一个响应（例如，操作成功的确认信息、错误消息或跳转指令等）。
处理流程简要：	1. 客户端发送POST请求，携带数据。 2. 服务器解析请求体中的数据，并执行相应的操作（如数据库插入、更新等）。 3. 服务器返回处理结果（如成功、失败、错误消息等）。

》》》》 既然有请求，那么服务器处理了请求之后，会做出响应

》》》》 HTTP 响应的格式：

HTTP响应是服务器向客户端（如浏览器）发送的消息，表示请求的处理结果。它通常包含以下内容：

- 1. 状态行：指示请求的处理结果和状态码。
- 2. 响应头：包含有关响应的信息，比如内容类型、缓存控制等。

3. 响应体：包含实际的响应数据，如HTML页面、图片、JSON、XML等。

响应的组成部分

状态行（Status Line）	<ul style="list-style-type: none">包含HTTP版本、状态码和状态消息。例如：HTTP/1.1 200 OK<ul style="list-style-type: none">HTTP/1.1：表示使用的HTTP协议版本。200：表示请求成功（状态码）。OK：状态码的描述信息。
响应头（Response Headers）	<ul style="list-style-type: none">含有描述响应内容的元信息。例如：<ul style="list-style-type: none">Content-Type: text/html 表示响应体是HTML内容。Content-Length: 1234 表示响应体的大小是1234字节。Cache-Control: no-cache 指示客户端不要缓存响应。Set-Cookie: sessionId=abc123 用于设置浏览器的cookie。
响应体（Response Body）	<ul style="list-style-type: none">响应体包含了服务器返回的具体数据内容，如网页HTML、图片、视频、JSON数据等。对于GET请求，响应体通常是请求的资源（如网页、图片、JSON数据等）。对于POST请求，响应体可能是服务器操作结果的反馈，或者确认消息等。

常见的 HTTP 响应格式

<table><tr><td>HTML</td><td>如果客户端请求网页（如GET /index.html），响应体通常是HTML格式的页面。</td></tr><tr><td>示例：</td><td>HTTP/1.1 200 OK Content-Type: text/html; charset=UTF-8 Content-Length: 215 <html> <head> <title>Example</title> </head> <body> <h1>Welcome to the site!</h1> </body> </html></td></tr></table>	HTML	如果客户端请求网页（如GET /index.html），响应体通常是HTML格式的页面。	示例：	HTTP/1.1 200 OK Content-Type: text/html; charset=UTF-8 Content-Length: 215 <html> <head> <title>Example</title> </head> <body> <h1>Welcome to the site!</h1> </body> </html>	<table><tr><td>JSON</td><td>常用于API响应，特别是AJAX请求或RESTful API接口，响应体通常是JSON格式的数据。</td></tr><tr><td>示例：</td><td>HTTP/1.1 200 OK Content-Type: application/json Content-Length: 52 { "message": "Form submitted successfully", "status": "OK" }</td></tr></table>	JSON	常用于API响应，特别是AJAX请求或RESTful API接口，响应体通常是JSON格式的数据。	示例：	HTTP/1.1 200 OK Content-Type: application/json Content-Length: 52 { "message": "Form submitted successfully", "status": "OK" }
HTML	如果客户端请求网页（如GET /index.html），响应体通常是HTML格式的页面。								
示例：	HTTP/1.1 200 OK Content-Type: text/html; charset=UTF-8 Content-Length: 215 <html> <head> <title>Example</title> </head> <body> <h1>Welcome to the site!</h1> </body> </html>								
JSON	常用于API响应，特别是AJAX请求或RESTful API接口，响应体通常是JSON格式的数据。								
示例：	HTTP/1.1 200 OK Content-Type: application/json Content-Length: 52 { "message": "Form submitted successfully", "status": "OK" }								
<table><tr><td>XML</td><td>在某些情况下，特别是SOAP Web服务中，响应体可能是XML格式。</td></tr><tr><td>示例：</td><td>HTTP/1.1 200 OK Content-Type: application/xml Content-Length: 92 <response> <status>success</status> <message>Data processed successfully</message> </response></td></tr></table>	XML	在某些情况下，特别是SOAP Web服务中，响应体可能是XML格式。	示例：	HTTP/1.1 200 OK Content-Type: application/xml Content-Length: 92 <response> <status>success</status> <message>Data processed successfully</message> </response>	<table><tr><td>图片/二进制文件</td><td>如果请求的是图片或其他二进制文件，响应体会包含这些文件的数据。</td></tr><tr><td>示例：</td><td>HTTP/1.1 200 OK Content-Type: image/jpeg Content-Length: 2048 [二进制图像数据]</td></tr></table>	图片/二进制文件	如果请求的是图片或其他二进制文件，响应体会包含这些文件的数据。	示例：	HTTP/1.1 200 OK Content-Type: image/jpeg Content-Length: 2048 [二进制图像数据]
XML	在某些情况下，特别是SOAP Web服务中，响应体可能是XML格式。								
示例：	HTTP/1.1 200 OK Content-Type: application/xml Content-Length: 92 <response> <status>success</status> <message>Data processed successfully</message> </response>								
图片/二进制文件	如果请求的是图片或其他二进制文件，响应体会包含这些文件的数据。								
示例：	HTTP/1.1 200 OK Content-Type: image/jpeg Content-Length: 2048 [二进制图像数据]								
<table><tr><td>纯文本</td><td>有时响应体可能是纯文本内容。</td></tr><tr><td>示例：</td><td>HTTP/1.1 200 OK Content-Type: text/plain Content-Length: 15 Hello, world!</td></tr></table>	纯文本	有时响应体可能是纯文本内容。	示例：	HTTP/1.1 200 OK Content-Type: text/plain Content-Length: 15 Hello, world!					
纯文本	有时响应体可能是纯文本内容。								
示例：	HTTP/1.1 200 OK Content-Type: text/plain Content-Length: 15 Hello, world!								

响应示例

GET请求的响应：	请求：GET /index.html HTTP/1.1
响应：	HTTP/1.1 200 OK Content-Type: text/html; charset=UTF-8 Content-Length: 215 <html> <head> <title>Welcome</title> </head> <body> <h1>Hello, world!</h1> </body> </html>
POST请求的响应：	请求：POST /submit-form HTTP/1.1（包含表单数据）
响应：	HTTP/1.1 200 OK Content-Type: application/json Content-Length: 52 { "message": "Form submitted successfully", "status": "OK" }

》》》其他响应

1. FTP (File Transfer Protocol) 响应

响应格式:	FTP响应通常包含状态码和附带信息，通常是三位数字的状态码，后跟一条信息。
用途:	用于文件传输。
示例:	<ul style="list-style-type: none">200 OK: 表示命令成功执行。550 Requested action not taken: 表示无法访问请求的文件。
完整示例:	220 FTP server ready. 230 User logged in, proceed. 550 File not found.

2. SMTP (Simple Mail Transfer Protocol) 响应

用途:	用于邮件发送。
响应格式:	SMTP响应通常是三位数字代码，并附带一个简短的描述信息。
常见状态码:	<ul style="list-style-type: none">220: 服务就绪。250: 请求完成。550: 拒绝邮件发送。
示例:	220 smtp.example.com ESMTP Exim 4.94.2 Wed, 20 May 2025 10:23:45 +0000 250 OK 550 Requested action not taken: mailbox unavailable

3. IMAP (Internet Message Access Protocol) 响应

用途:	用于电子邮件的接收与管理。
响应格式:	IMAP的响应格式也是以状态码为主，后跟响应信息，通常使用一组标签或标识符。
示例:	<ul style="list-style-type: none">OK: 表示命令成功执行。NO: 表示命令失败。BAD: 表示请求无效。
示例:	* 3 FETCH (FLAGS (\Seen) UID 1234) OK FETCH completed. * BYE IMAP4rev1 server terminating connection

4. DNS (Domain Name System) 响应

用途:	用于域名解析。
响应格式:	DNS响应通常由DNS服务器返回，它包含查询的结果，如IP地址或其他资源记录。
常见响应:	<ul style="list-style-type: none">查询类型是A记录时，返回IP地址。查询类型是MX记录时，返回邮件交换服务器的域名。
示例:	;; ANSWER SECTION: example.com. 3600 IN A 93.184.216.34

》》》常用的 async 异步函数

1. async_read

定义:	async_read 是异步读取数据的函数，它从套接字或流中读取数据，直到读取到指定的字节数或遇到分隔符（例如，换行符、文件结束符等）。它不会阻塞程序的执行，而是通过回调机制将读取结果返回。
用法:	<ul style="list-style-type: none">你可以在一个异步任务中调用 async_read 来读取网络数据。它接受三个参数: socket（套接字对象）、buffer（存储数据的缓冲区）、以及一个回调函数（或者是一个接受完成参数的lambda函数），在读取完成后会调用回调函数。
何时使用:	当你需要异步读取数据并且不想让主线程阻塞时使用，特别是在处理高并发连接时。
示例:	<pre>boost::asio::async_read(socket, buffer, [](boost::system::error_code ec, std::size_t length) { if (!ec) { std::cout << "Read " << length << " bytes." << std::endl; } });</pre>

2. async_write

定义:	async_write 是异步写入数据的函数，它将数据写入到套接字或流中，并且不会阻塞线程，写入操作完成后会触发回调函数。
用法:	<ul style="list-style-type: none">与 async_read 类似，async_write 也是非阻塞的，它接受套接字、数据缓冲区以及一个回调函数作为参数。回调函数会在数据写入完成后被调用，可以在回调函数中处理错误或者进行后续操作。
何时使用:	当你需要异步地将数据发送到远程服务器，尤其是在高负载、高并发场景中，不希望阻塞主线程。
示例:	<pre>boost::asio::async_write(socket, boost::asio::buffer(data), [](boost::system::error_code ec, std::size_t length) { if (!ec) {</pre>

```
std::cout << "Wrote " << length << " bytes." << std::endl;
}
});
```

3. `async_accept`

定义：	<code>async_accept</code> 用于在服务器端接受连接。当有客户端尝试连接时，它会异步地接受连接并触发回调函数。
用法：	你通常会在服务器端使用 <code>async_accept</code> 来接受来自客户端的连接，并且该函数不会阻塞程序的执行。连接成功后，回调函数将被调用，之后可以在回调中进行后续的读写操作。
何时使用：	适用于服务器端需要处理多个客户端连接，且希望非阻塞地接受连接请求时。
示例：	<pre>acceptor.async_accept(socket, [](boost::system::error_code ec) { if (!ec) { std::cout << "Connection accepted." << std::endl; } });</pre>

4.1 `async_connect`

定义：	<code>async_connect</code> 用于异步地连接到远程端点（如服务器）。
用法：	它发起连接请求，并在连接成功后调用回调函数。
何时使用：	当你需要在客户端应用程序中异步连接到远程服务器时使用。
示例：	<pre>boost::asio::async_connect(socket, endpoints, [](boost::system::error_code ec, tcp::endpoint endpoint) { if (!ec) { std::cout << "Connected to " << endpoint << std::endl; } });</pre>

4.2 `async_read_until`

定义：	<code>async_read_until</code> 用于异步地从流中读取数据，直到遇到特定的分隔符（如换行符、某个字符或模式）。
用法：	它可以读取直到指定的分隔符或字节数，非常适用于协议设计中需要按行、按分隔符读取数据的场景。
何时使用：	当数据中存在明确的分隔符时，使用此函数非常方便。
示例：	<pre>boost::asio::async_read_until(socket, buffer, '\n', [](boost::system::error_code ec, std::size_t length) { if (!ec) { std::cout << "Read until newline: " << length << " bytes." << std::endl; } });</pre>

4.3 `async_wait`

定义：	<code>async_wait</code> 用于等待某个定时器的超时事件。它可以使程序在等待一段时间后执行回调。
用法：	通常用于定时任务或超时检测，指定时间过去后调用回调函数。
何时使用：	适用于需要设置延时或超时功能时。
示例：	<pre>boost::asio::steady_timer timer(io_context, boost::asio::chrono::seconds(5)); timer.async_wait([](boost::system::error_code ec) { if (!ec) { std::cout << "Timer expired!" << std::endl; } });</pre>

5. `async_shutdown`

定义：	<code>async_shutdown</code> 用于异步关闭一个套接字，关闭连接时不阻塞。
用法：	通常用于关闭TCP连接时，确保在关闭时不会阻塞其他操作。
何时使用：	当你需要优雅地关闭网络连接且不希望阻塞时使用。
示例：	<pre>boost::asio::async_shutdown(socket, [](boost::system::error_code ec) { if (!ec) { std::cout << "Socket closed." << std::endl; } });</pre>

》》》我终于明白为什么需要使用 `shared_from_this` 了（请看实例：`m_Socket` 是一个已经被定义的变量）

这样的代码有一种危险： 你想直接在 Lambda 中捕获 <code>m_Socket</code> 和 <code>m_Deadline</code> ，但这两个变量是 <code>HttpConnection</code> 类的成员变量（也就是你当前对象的成员）。	<pre>auto self = shared_from_this(); boost::beast::http::async_write(XXX, XXX, [m_Socket, m_Deadline](boost::beast::error_code ec, std::size_t size) { { m_Socket.shutdown(XXX, ec); } });</pre>
---	--

如果你直接捕获这些成员变量，它们会在 Lambda 执行时被销毁，导致你在异步操作过程中访问到悬空对象（悬空引用），即这些对象已经不再存在或者被销毁了。	
解决方法是使用 shared_from_this(): 它返回一个 shared_ptr，确保当前对象在每一次的 Lambda 回调中仍然有效，不会被销毁。	<pre>auto self = shared_from_this(); boost::beast::http::async_write(XXX, XXX, [self](boost::beast::error_code ec, std::size_t size) { self->m_Socket.shutdown(XXX, ec); });</pre>

》》》socket 的成员函数 close() 和 shutdown() 有什么不同？

close():	<ul style="list-style-type: none">close() 用于 完全关闭套接字，即关闭网络连接。这是一个终止连接的操作，不再接受任何数据的发送或接收。当你调用 close() 时，套接字将被完全关闭，且不再能够进行任何网络操作。这通常用于连接已经完成，或者即将被完全销毁的情况。						
shutdown():	<ul style="list-style-type: none">shutdown() 用于 部分关闭套接字，通常分为三个方向：接收（boost::asio::ip::tcp::socket::shutdown_receive）、发送（boost::asio::ip::tcp::socket::shutdown_send），或两者都关闭（boost::asio::ip::tcp::socket::shutdown_both）。 <table><tr><td>shutdown_receive:</td><td>此操作关闭接收端口，意味着你不再期望接收更多的数据，但仍然允许向对方发送数据。这在一些协议或场景中非常有用，比如你想在客户端发送完所有数据后，确保不再接收来自服务器的数据，但服务器仍然可以接收客户端的数据。这种情况下，shutdown_receive() 可以防止进一步接收数据，但连接仍然可以继续发送数据。</td></tr><tr><td>shutdown_send:</td><td>此操作关闭发送端口，意味着你已经完成了所有数据的发送，但仍然希望接收来自对方的数据。这通常发生在你已发送完所有需要发送的数据，并希望等待对方的响应或最后的数据包。这对于处理 半关闭 状态的连接特别有用，比如在 TCP 协议中，客户端可以关闭发送数据流，但继续接收对方的数据，直到对方也关闭连接。</td></tr><tr><td>shutdown_both:</td><td>该操作同时关闭接收和发送端口，即彻底关闭连接。这通常是在双方完成通信后，确保连接不能再发送或接收数据的场景中使用。</td></tr></table> <ul style="list-style-type: none">调用 shutdown() 后，你可以阻止进一步的数据接收或发送，但连接本身仍然保持打开，允许完成一些收尾工作，如处理数据或接收更多的关闭信号。	shutdown_receive:	此操作关闭接收端口，意味着你不再期望接收更多的数据，但仍然允许向对方发送数据。这在一些协议或场景中非常有用，比如你想在客户端发送完所有数据后，确保不再接收来自服务器的数据，但服务器仍然可以接收客户端的数据。这种情况下，shutdown_receive() 可以防止进一步接收数据，但连接仍然可以继续发送数据。	shutdown_send:	此操作关闭发送端口，意味着你已经完成了所有数据的发送，但仍然希望接收来自对方的数据。这通常发生在你已发送完所有需要发送的数据，并希望等待对方的响应或最后的数据包。这对于处理 半关闭 状态的连接特别有用，比如在 TCP 协议中，客户端可以关闭发送数据流，但继续接收对方的数据，直到对方也关闭连接。	shutdown_both:	该操作同时关闭接收和发送端口，即彻底关闭连接。这通常是在双方完成通信后，确保连接不能再发送或接收数据的场景中使用。
shutdown_receive:	此操作关闭接收端口，意味着你不再期望接收更多的数据，但仍然允许向对方发送数据。这在一些协议或场景中非常有用，比如你想在客户端发送完所有数据后，确保不再接收来自服务器的数据，但服务器仍然可以接收客户端的数据。这种情况下，shutdown_receive() 可以防止进一步接收数据，但连接仍然可以继续发送数据。						
shutdown_send:	此操作关闭发送端口，意味着你已经完成了所有数据的发送，但仍然希望接收来自对方的数据。这通常发生在你已发送完所有需要发送的数据，并希望等待对方的响应或最后的数据包。这对于处理 半关闭 状态的连接特别有用，比如在 TCP 协议中，客户端可以关闭发送数据流，但继续接收对方的数据，直到对方也关闭连接。						
shutdown_both:	该操作同时关闭接收和发送端口，即彻底关闭连接。这通常是在双方完成通信后，确保连接不能再发送或接收数据的场景中使用。						

》》》友元的含义以及使用:

当你在一个类中指定一个类作为友元类时，意味着友元类的成员函数可以访问该类的私有和保护成员，尽管这些成员在类的外部通常是不可访问的。

<pre>class ClassA { private: int privateVar; public: ClassA() : privateVar(42) {} // 指定 ClassB 为友元类（表明 B 可以使用 A 的私有和保护成员） friend class ClassB; };</pre>	<pre>class ClassB { public: void accessClassA(ClassA& obj) { // 可以访问 ClassA 的私有成员 privateVar std::cout << "Private variable of ClassA: " << obj.privateVar << std::endl; } };</pre>
---	--

》》》int main 主函数中的逻辑是什么意思？

1. 初始化网络上下文 (io_context)	<pre>net::io_context ioc{ 1 }; • ioc 是一个 I/O 上下文，它是 Boost.Asio 中的一个核心对象，负责管理所有异步操作的调度。 • 1 表示线程数量，通常会使用多个线程来执行异步任务，但在这里它是初始化为 1。</pre>
2. 设置信号处理 (signal_set)	<pre>boost::asio::signal_set signals(ioc, SIGINT, SIGTERM); • signal_set 是一个对象，用于异步处理系统信号。在这里，它会监听 SIGINT（通常由 Ctrl+C 发送）和 SIGTERM（通常由系统发送来终止程序）信号。 • 这个设置是为了让程序在接收到终止信号时能够进行清理并退出。</pre>
3. 异步等待信号 (async_wait)	<pre>signals.async_wait([&ioc](const boost::system::error_code& error, int signal_number) { if (error) { return; } ioc.stop(); }); • async_wait 是异步操作，它会等待指定的信号（SIGINT 或 SIGTERM）到来。 • 一旦信号到来，回调函数会被触发。 • 回调函数出发后，如果没有错误（error 为 false），则调用 ioc.stop() 停止 io_context，它会停止处理进一步的异步操作并退出 ioc.run() 循环。</pre>
4. 启动服务器 (CServer)	<pre>std::make_shared<CServer>(ioc, port)->Start();</pre>

- `std::make_shared<CServer>(ioc, port)` 创建一个共享指针，指向 CServer 类的实例，CServer 类假定是某种网络服务器。
- 该服务器对象通过 ioc 和端口号 (8080) 初始化，并调用 `Start()` 启动服务。
- CServer 的 `Start` 方法可能包含一些异步网络操作，如接受连接或处理数据。

》》》信号

一些信号的定义：（可以看到 SIGINT 指的是 interrupt:打断，即 --> 用户主动中断；SIGTERM 指的是软件层面上的终止信号）

```

27 // Signal types
28 #define SIGINT 2 // interrupt
29 #define SIGILL 4 // illegal instruction - invalid function image
30 #define SIGFPE 8 // floating point exception
31 #define SIGSEGV 11 // segment violation
32 #define SIGTERM 15 // Software termination signal from kill
33 #define SIGBREAK 21 // Ctrl-Break sequence
34 #define SIGABRT 22 // abnormal termination triggered by abort call
35
36 #define SIGABRT_COMPAT 6 // SIGABRT compatible with other platforms, same as SIGABRT
37
38 // Signal action codes
39 #define SIG_DFL ((_crt_signal_t)0) // default signal action
40 #define SIG_IGN ((_crt_signal_t)1) // ignore signal
41 #define SIG_GET ((_crt_signal_t)2) // return current value
42 #define SIG_SEG ((_crt_signal_t)3) // signal gets error
43 #define SIG_ACK ((_crt_signal_t)4) // acknowledge
44
45 #ifdef _CORECRT_BUILD
46 // Internal use only! Not valid as an argument to signal().
47 #define SIG_DIE ((_crt_signal_t)5) // terminate process
48 #endif
49
50 // Signal error value (returned by signal call on error)
51 #define SIG_ERR ((_crt_signal_t)-1) // signal error value
52
53

```

》》》什么是 static_cast? 为什么需要使用 static_cast?

什么是 static cast ?

static_cast 是 C++ 中的一种类型转换操作符，用于在类型之间进行显式的转换。它在编译时进行类型检查，能够安全地将一种类型转换为另一种类型，只要这种转换是合法的。与其他类型转换操作符（如 `reinterpret_cast` 或 `dynamic_cast`）不同，`static_cast` 只适用于类型之间具有明确关系（如继承关系或基本类型转换）的转换。

为什么需要 static_cast<unsigned short>(8080)?

1. 类型转换：在 C++ 中，8080 是一个常量整数（int 类型）。然而，`unsigned short` 是一种较小的整数类型，通常为 16 位，表示范围从 0 到 65535（通常是 $2^{16} - 1$ ）。当我们把一个值赋给 `unsigned short` 类型的变量时，必须确保它符合该类型的范围。

```

unsigned short port = 8080;
boost::asio::signal_set s{
    signals.async_wait([](const boost::system::error_code& ec, int signalNumber)
    {
        // ...
    })
};

```

2. 显式转换：使用 `static_cast<unsigned short>(8080)` 将 8080 显式地转换为 `unsigned short` 类型。虽然 8080 在 `int` 类型范围内，但是为了确保类型的正确性和避免潜在的隐式转换错误，显式地使用 `static_cast` 可以帮助我们：
 - 明确指定我们想要的类型转换。
 - 使代码更具可读性和可维护性，特别是在类型转换可能影响程序行为时。

如果 8080 的值超过了 `unsigned short` 的上限（65535），赋值可能会导致数据丢失或溢出。使用 `static_cast` 可以明确指定转换类型，并且在编译时提醒你进行这种转换。

 - 即使 8080 在 `unsigned short` 的有效范围内，使用 `static_cast` 也是一种良好的编程实践，确保代码清晰、显式。

》》提示

```

unsigned short port = unsigned short(8080);

```

这样写效果也是一样的。

》》》ec.what() 和 ec.message() 的区别?

```

signals.async_wait([](const boost::system::error_code& ec, int signalNumber)
{
    if (ec)
        JC_CORE_ERROR("in {} ", ec.what(), __FILE__);
    return;
})

```

ec.what()	通常用于提供简短的错误描述，适用于用于异常处理时输出错误信息。
ec.message()	提供更详细的错误描述，帮助理解具体的错误原因或背景。

》》》EXIT_FAILURE

意义：	EXIT_FAILURE 是一个宏常量，用于表示程序执行失败的退出状态。它通常在程序执行异常时返回，表示程序未正常完成，错误退出。而 EXIT_SUCCESS 与其相对，表示程序成功退出。
定义：	EXIT_FAILURE 在标准库头文件 <cstdlib> 中定义。 #define EXIT_FAILURE 1 <ul style="list-style-type: none">EXIT_FAILURE 通常被定义为 1（某些系统中可能是其他值），代表程序异常退出。EXIT_SUCCESS 是另一个常量，通常定义为 0，表示程序成功退出。

》》》现在一切准备就绪，我决定先把 GET 请求的处理函数注释掉，看看会怎样。

```
#include "Jpch.h"
#include "LogicSystem.h"
#include "HttpConnection.h"

// Constructor is private
LogicSystem::LogicSystem()
{
    // 这里存储了一个对于客户端 GET 请求的简单处理方法（答复一些简单的字符）
    // *RegGet("/get_test", [](std::shared_ptr<HttpConnection> connection)
    // {
    //     boost::beast::ostream(connection->m_Response.body()) << "Recevie get_test request :-)\n";
    // }, *);
}

LogicSystem::~LogicSystem()
{
}

void LogicSystem::RegGet(std::string url, std::function<void(std::shared_ptr<HttpConnection>>> func)
{
    m_GetHandlers.insert(std::make_pair(url, func));
}

bool LogicSystem::HandleGet(std::string path, std::shared_ptr<HttpConnection> connection)
{
    if(m_GetHandlers.find(path) == m_GetHandlers.end())
```

可以看到，这会导致请求没有正常处理。这是函数的定义：

```
void HttpConnection::HandleReq()
{
    m_Response.version(m_Request.version()); // 设置版本
    m_Response.keep_alive(false);           // 设置为短连接 (http)

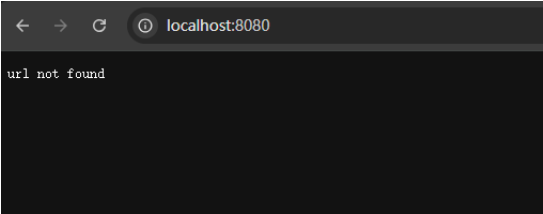
    // 处理 GET 请求
    if (m_Request.method() == boost::beast::http::verb::get)
    {
        // 如果检测到对应请求，则进行处理。
        bool success = LogicSystem::GetInstance()->HandleGet(m_Request.target(), shared_from_this());

        if (!success)
        {
            m_Response.result(http::status::not_found);
            m_Response.set(http::field::content_type, "text/plain");
            boost::beast::ostream(m_Response.body()) << "url not found \r\n";

            // 将 m_Reponse 的内容写入当前连接的 socket 中，并在函数中终止 socket 的发送端，表示发送完毕
            WriteResponse();

            return;
        }
    }
}
```

这是 localhost:8080



》》》当程序运行超过 1 分钟之后，如果我们刷新 localhost:8080 页面，会得到这样一行日志：
(程序启动时间为 16:04:36，期间我刷新了三次，第三次超过了 60s 时间限制)

```
E:\VS\JustinChat\VisualStudio x + -
[16:04:36] JstChat: Justin Chat GateServer is running
[16:04:41] JstChat: Can't find '/' in GetHandlers, FILE:E:\VS\JustinChat\VisualStudioProj\GateServer\GateServer\src\Logi
cSystem.cpp
[16:05:36] JstChat: Can't find '/' in GetHandlers, FILE:E:\VS\JustinChat\VisualStudioProj\GateServer\GateServer\src\Logi
cSystem.cpp
[16:05:44] JstChat: Http read error is end of stream [beast.http:1 at E:\VS\JustinChat\VisualStudioProj\GateServer\GateS
erver\vendor\boost_1_88_0\include\boost\beast\http\impl\read.hpp:241:21 in function 'void __cdecl boost::beast::http::det
ail::read_some_op<class boost::asio::basic_stream_socket<class boost::asio::ip::tcp, class boost::asio::any_io_executor>,
class boost::beast::basic_flat_buffer<class std::allocator<char>>, 1>::operator ()>class boost::asio::detail::composed_o
p<class boost::beast::http::detail::read_some_op<class boost::asio::basic_stream_socket<class boost::asio::ip::tcp, class
boost::asio::any_io_executor>, class boost::beast::basic_flat_buffer<class std::allocator<char>>, 1>, struct boost::asio:
:detail::composed_work<void __cdecl(class boost::asio::any_io_executor)>, class boost::asio::detail::composed_op<class bo
ost::beast::http::detail::read_op<class boost::asio::basic_stream_socket<class boost::asio::ip::tcp, class boost::asio::a
ny_io_executor>, class boost::beast::basic_flat_buffer<class std::allocator<char>>, 1>, struct boost::beast::http::basic_dy
namic_body<class boost::beast::basic_multi_buffer<class std::allocator<char>>>, class std::allocator<char>, class lambda
a_473b1bbaf63d9c1c8010fca62fa29d49>>, void __cdecl(class boost::system::error_code, unsigned __int64)>, void __cdecl(class
boost::system::error_code, unsigned __int64)>>(class boost::asio::detail::composed_op<class boost::beast::http::detail::
read_some_op<class boost::asio::basic_stream_socket<class boost::asio::ip::tcp, class boost::asio::any_io_executor>, class
boost::beast::basic_flat_buffer<class std::allocator<char>>, 1>, struct boost::beast::asio::detail::composed_work<void __cdecl(
class boost::asio::any_io_executor)>, class boost::asio::detail::composed_op<class boost::beast::http::detail::read_op<cl
ass boost::asio::basic_stream_socket<class boost::asio::ip::tcp, class boost::asio::any_io_executor>, class boost::beast::
basic_flat_buffer<class std::allocator<char>>, 1>, struct boost::beast::http::basic_dynamic_body<class boost::beast::basic
_multi_buffer<class std::allocator<char>>>, class std::allocator<char>, class lambda_473b1bbaf63d9c1c8010fca62fa29d49>
>>, void __cdecl(class boost::system::error_code, unsigned __int64)>, void __cdecl(class boost::system::error_code, unsigned
```

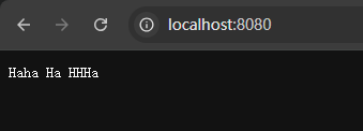
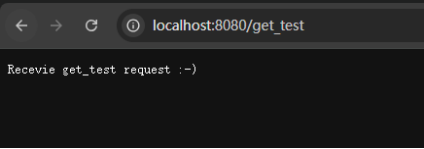
这行日志来自: `HttpConnection::Start()` 函数, `boost::beast::http::async_read` 表示在读取数据时检测到 连接被对方关闭 (例如客户端断开或请求不完整)。
这表明我们的超时检测奏效了。

))) 重新为 GetHandlers 添加不同功能

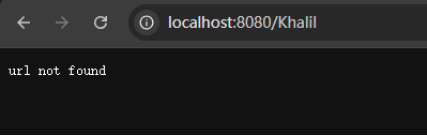
```
// Constructor is private
LogicSystem::LogicSystem()

// 这里存储对于客户端 GET 请求的简单处理方法 (答复一些简单的字符)
// "get_test" 表示客户发出请求的网址是 http://localhost:8080/get_test
RegGet("/get_test", [](std::shared_ptr<HttpConnection> connection)
{
    boost::beast::ostream(connection->m_Response.body()) << "Recevie get_test request :-)";
});

// "/" 表示客户发出请求的网址是 http://localhost:8080/
RegGet("/", [](std::shared_ptr<HttpConnection> connection)
{
    boost::beast::ostream(connection->m_Response.body()) << "Haha Ha 哈哈";
});
```



))) 如果访问未注册的目标, 则



并报错:

```
E:\VS\JustinChat\VisualStudio x + -
[16:05:01] JstChat: Justin Chat GateServer is running
[16:05:07] JstChat: Can't find '/Khalil' in GetHandlers, FILE:E:\VS\JustinChat\VisualStudioProj\GateServer\GateServer\src
\LogicSystem.cpp
```

----- Ep 6 Http post 请求 -----

))))

