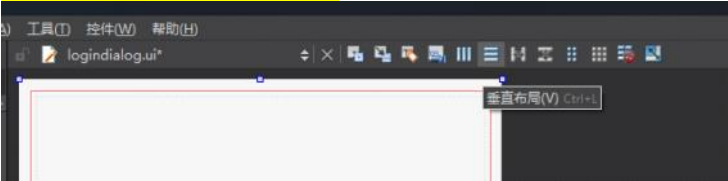


----- 登录界面 -----

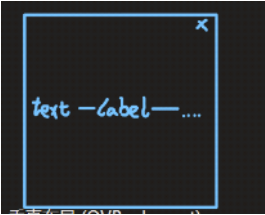
》》》什么是QT中的垂直布局和水平布局？



Qt中的水平布局（QHBoxLayout）和垂直布局（QVBoxLayout）是控制部件（如按钮、文本框、标签等）在界面中排放方式的布局管理器。

1. 水平布局 (QHBoxLayout)

- 在水平布局中，界面上的部件会沿着水平方向（从左到右）依次排列。
- 每个控件会水平摆放在一个行内，从左到右的顺序，控件之间的间隔可以通过布局管理器进行设置。



2. 垂直布局 (QVBoxLayout)

- 在垂直布局中，界面上的部件会沿着垂直方向（从上到下）依次排列。
- 每个控件会垂直摆放在一列中，从上到下的顺序，控件之间的间隔同样可以通过布局管理器进行设置。



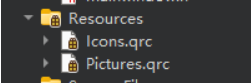
总结：

- 水平布局将部件按水平（从左到右）的顺序排列。
- 垂直布局将部件按垂直（从上到下）的顺序排列。

》》》QT中的资源文件：.qrc 可以在同目录下存在多个吗？比如resources之下有：Icons.qrc, Textures.qrc多个.qrc 文件

可以。

- 每个 .qrc 文件是一个资源集合，它可以包含多个文件，比如图片、图标、样式表等。
- 多个 .qrc 文件可以分门别类，例如你可以将图标资源放在 Icons.qrc 文件中，将纹理资源放在 Textures.qrc 文件中，这样可以更好地组织资源。

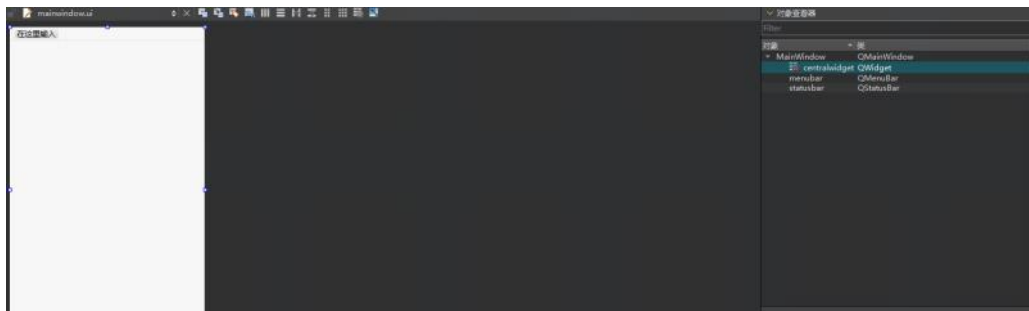


》》》Align：对齐

》》》setCentralWidget()

setCentralWidget 是 Qt 框架中 QMainWindow 类的一个成员函数，它用于设置主窗口的中央部件。
具体来说，setCentralWidget 允许你指定一个部件（比如一个小部件或一个布局）作为主窗口的中心区域，一般用来指定主窗口的内容。

比如：主窗口为空，但有几个组件，其中之一为：centralwidget，我们通过 SetCentralWidget() 函数将自定义的窗口指定于主窗口（main window），然后使用 show() 函数展示。



maniWindow.cpp

```
4 MainWindow::MainWindow(QWidget *parent)
5     : QMainWindow(parent), ui(new Ui::MainWindow)
6     {
7         ui->setupUi(this);
8         m_LoginDialog = new LoginDialog();
9         setCentralWidget(m_LoginDialog);
10        m_LoginDialog->show();
11
12        this->setWindowIcon(QIcon(":/resources/chat4.png")); // 将资源中的 Icon 设置在窗口上
13    }
14
```

》》》》 什么是 QT 中的 signals 和 slots ?

signals

signals 关键字是 Qt 中特有的，用于声明信号。

信号是 Qt 的对象间通信机制之一，signals 通常与槽（slots）配合使用。信号用于在对象之间发送通知，信号和槽的连接由 Qt 的 `QObject::connect()` 函数处理。

例如，当某个事件发生时，发出一个信号，其他对象可以响应该信号。

```
class MyClass : public QObject {
    Q_OBJECT // 必须包含此宏才能使用信号和槽

public:
    MyClass() : QObject() {}

signals:
    void valueChanged(int newValue); // 信号声明

public slots:
    void onValueChanged(int newValue) {
        qDebug() << "Value changed to" << newValue;
    }
};
```

在这个示例中，valueChanged 是一个信号，表示值已经变化。信号的发送通常通过 emit 关键字进行，如：

```
emit valueChanged(42);
```

slots

slots 也是在类中定义的，不同于 signals 的是，slots 可以放在类的 public、protected 或 private 部分，具体取决于你想如何控制访问权限。

public slots:	如果槽是公共的，那么外部代码可以直接调用这些槽函数。这在很多情况下是需要的，因为槽函数可能是信号的响应函数。
private slots:	如果槽是私有的，那么它们只能在类内部被调用，外部代码无法直接调用。通常用于仅在内部响应某些信号的场景。
protected slots:	保护槽可以在类的派生类中访问。

```
class MyClass : public QObject {
    Q_OBJECT

public:
    MyClass() {}

signals:
    void valueChanged(int newValue);

public slots:
    void onValueChanged(int newValue) {
        qDebug() << "Value changed to" << newValue;
    }

private slots:
    void internalSlot() {
        // 仅在内部调用的槽
    }
};
```

Signals 的作用域（是否存在修饰符？）

在 Qt 中，signals 不支持像 slots 那样使用 public 或 private 进行前向修饰，因为信号的作用是：让类的外部对象能够触发它们，并与某个对象进行通信。因此 signals 在类中默认是 public 的，如果信号被声明为私有或受保护的，外部对象就无法连接到它，这违背了信号与槽机制的设计目的。

```
class MyClass : public QObject {
    Q_OBJECT

public:
    MyClass() {}

signals: // 这里的信号默认是 public 的，不需要显式指定
    void valueChanged(int newValue); // 信号是 public 的，外部可以连接和触发

private:
    void privateMethod() {}
};
```

在这个示例中，valueChanged 信号默认是 public 的，你不需要显式地使用 public signals。Qt 也不允许将其声明为 private。

slots 和 signals 的区别：

signals:	通常是 public 的，用于与外部对象通信。信号不会有访问限制，因此能够被外部对象通过 connect() 连接和触发。
slots:	可以是 public、protected 或 private，这取决于设计者是否希望在外部分访问，或只在类内部调用。

》》》注意：两个不同的 LoginDialog

```
// 该 LoginDialog 属于 UI 命名空间下，继承于 Ui_LoginDialog
namespace Ui {
    class LoginDialog;
}

// 该 LoginDialog 属于全局范围下，继承于 QDialog
class LoginDialog : public QDialog // LoginDialog 继承于 QDialog，所以有一些默认的成员函数可以使用（比如 show()）
{
    Q_OBJECT

public:
    explicit LoginDialog(QWidget *parent = nullptr);
    ~LoginDialog();

private:
    Ui::LoginDialog *ui; // 注意：此处使用的类型是 Ui::LoginDialog，而非 ::LoginDialog

signals:
    void switchRegister();
}
```

》》》qss 文件编写格式文档

(参考网址)[<https://doc.qt.io/qt-6/stylesheet-syntax.html>]

.qss 文件是 Qt Style Sheets 文件，用于在 Qt 应用程序中定义界面元素的样式和外观，类似于网页中的 CSS（Cascading Style Sheets）。Qt Style Sheets 允许开发者控制 Qt 小部件（例如按钮、标签、文本框等）在多种情况下的外观、颜色、字体等属性。

》》qss 中的 # 是什么意思？

在 CSS 或 QSS 中，# 是一个选择符，表示选择具有特定 ID 的元素。

```
eg.
QDialog#LoginDialog {
    background-color: lightblue;
}
```

在 QSS 中使用 # 来选择这个 objectName 为 LoginDialog 的 QDialog，并手动将其背景颜色设置为蓝色。

》》》关于：QLatin1String 类

QLatin1String 不是一个函数，而是一个 Qt 类。它用于表示 Latin-1 编码（ISO 8859-1）的一种优化形式，通常用于字符串的存储和传递。

QLatin1String 类的作用：QLatin1String 主要用于处理 Latin-1 编码 的字符串，它通常比使用普通的 QString 更高效，特别是在处理大量 Latin-1 编码的数据时，因为 QLatin1String 使用的是一个只读的字节数组。

```
if(qss.open(QFile::ReadOnly))
{
    qDebug("Qss open success.");
    QString str = QLatin1String(qss.readAll());
    a.setStyleSheet(str);
    qss.close();
}
```

-----内存修复&qss-----

》》》树形管理机制：

Qt 有一个父子窗口的管理机制，这种机制有助于统筹窗口的生命周期，并对内存进行管理。
具体来说，Qt 的 父子关系 是基于 对象树形结构 来组织的，所有窗口控件（包括对话框、窗口等）都可以设计成这种结构（我猜想这可能是通过父类与子类之间的继承，来进行设计）。

树形结构及其特性

1. 树形结构：	Qt 中的对象（尤其是界面元素）是通过父子关系组织成一个树形结构的。 每个控件（比如 QWidget）都可以指定一个父控件（比如 MainWindow），它将成为该控件的父节点。如果某个控件没有父控件，它就是一个顶级控件（根控件）。 例如： <ul style="list-style-type: none">• MainWindow 是根控件，它没有父控件。• LoginDialog 是 MainWindow 的子控件，它的父控件是 MainWindow。 这个树形结构的组织方式类似于一个 父节点和子节点 之间的关系，所有控件都会被包含在这个树中。
2. 内存管理和生命周期：	当父控件被销毁时，所有子控件会自动被销毁，确保没有内存泄漏。这个机制是通过 QObject 的析构函数实现的。 例如： <ul style="list-style-type: none">• 当 MainWindow 被销毁时，LoginDialog 和 RegisterDialog 作为它的子控件会自动销毁。• 你不需要显式地删除 LoginDialog 或 RegisterDialog，Qt 会自动处理。
3. 可视化管理：	通过父子关系，Qt 可以控制控件的显示和隐藏。如果父控件被隐藏，所有子控件也会自动隐藏。如果父控件显示，子控件也会显示。 例如，MainWindow 隐藏时，LoginDialog 和 RegisterDialog 会自动隐藏，无需单独操作。
4. 信号和槽机制：	父子控件之间还可以通过 Qt 的信号和槽机制 进行通信。子控件可以发射信号，父控件可以连接这些信号，并作出响应。

》》》关于一些代码的调用流程，这是我的理解

```
m_LoginDialog->setWindowFlags(Qt::CustomizeWindowHint);
```

当我们使用 m_LoginDialog 的成员函数时，以下为该函数的使用思路
Tips：（鼠标单击某一个类名或者函数，当光标位于字段之上时，使用 F2 快捷键，可以快速转到定义文件）

首先LoginDialog 是我们在 MainWindow.h 中定义的成员	<pre>private: Ui::MainWindow *ui; LoginDialog* m_LoginDialog; RegisterDialog* m_RegisterDialog;</pre>
这个成员属于 LoginDialog 类，而 LoginDialog 属于父类 ::Qdialog	<pre>class LoginDialog : public QDialog { Q_OBJECT</pre>
::Qdialog 又属于 ::Qwidget 类	<pre>class Q_WIDGETS_EXPORT QDialog : public QWidget { Q_OBJECT friend class QPushButton;</pre>
因此 m_LoginDialog 成员是 ::Qwidget 类的派生，可以使用该类的成员函数，比如 setWindowFlags()	<pre>571 QList<QAction> actions() const, 572 #endif 573 574 QWidget *parentWidget() const; 575 576 void setWindowFlags(Qt::WindowFlags type); 577 inline Qt::WindowFlags windowFlags() const; 578 void setWindowFlag(Qt::WindowType, bool on = true); 579 void overrideWindowFlags(Qt::WindowFlags type); 580 581 inline Qt::WindowType windowType() const; 582</pre>

<p>当我们使用 <code>m_LoginDialog</code> 调用这个函数时，需要填入参数类型：<code>Qt::WindowFlags</code></p>	<div data-bbox="638 129 1422 188"><pre>m_LoginDialog->setWindowFlags(Qt::CustomizeWindowHint);</pre></div> <div data-bbox="638 215 1275 257"><pre>void setWindowFlags(Qt::WindowFlags type);</pre></div>
<p>打开定义我们发现这样的宏定义： (这表明：<code>Q_DECLARE_FLAGS</code> 宏定义了一个位标志类型，它允许你将多个 <code>WindowType</code> 枚举值组合成一个 <code>WindowFlags</code> 类型的变量。)</p>	<div data-bbox="638 288 1508 427"><pre>Q_DECLARE_FLAGS(WindowFlags, WindowType) Q_DECLARE_OPERATORS_FOR_FLAGS(WindowFlags, WindowType) enum WindowType { ... }; Type: class QFlags<Qt::WindowType></pre></div>
<p>因此我们在使用该函数时，需要在参数中填写诸如： <code>WindowType1 WindowType2 WindowType3</code> 这样的组合。</p> <p>然而 <code>WindowType</code> 的定义就在这个宏定义 (<code>Q_DECLARE_FLAGS</code>) 的上方。</p>	<p>在 <code>Qt</code> 这个命名空间下(<code>qnamespcae.h</code>)，有很多枚举类。其中之一是 <code>WindowType</code>：</p> <div data-bbox="638 490 1278 1158"><pre>enum WindowType { Widget = 0x00000000, Window = 0x00000001, Dialog = 0x00000002 Window, Sheet = 0x00000004 Window, Drawer = Sheet Dialog, Popup = 0x00000008 Window, Tool = Popup Dialog, ToolTip = Popup Sheet, SplashScreen = ToolTip Dialog, Desktop = 0x00000010 Window, SubWindow = 0x00000012, ForeignWindow = 0x00000020 Window, CoverWindow = 0x00000040 Window, WindowType_Mask = 0x000000ff, MSWindowsFixedSizeDialogHint = 0x00000100, MSWindowsOwnDC = 0x00000200, BypassWindowManagerHint = 0x00000400, X11BypassWindowManagerHint = BypassWindowManagerHint, FramelessWindowHint = 0x00000800, WindowTitleHint = 0x00001000, WindowSystemMenuHint = 0x00002000, WindowMinimizeButtonHint = 0x00004000, WindowMaximizeButtonHint = 0x00008000, WindowMinMaxButtonsHint = WindowMinimizeButtonHint WindowMaximizeButtonHint, WindowContextHelpButtonHint = 0x00010000, WindowShadeButtonHint = 0x00020000, WindowStaysOnTopHint = 0x00040000, WindowTransparentForInput = 0x00080000, WindowOverridesSystemGestures = 0x00100000, WindowDoesNotAcceptFocus = 0x00200000, MaximizeUsingFullscreenGeometryHint = 0x00400000, CustomizeWindowHint = 0x02000000, WindowStaysOnBottomHint = 0x04000000, WindowCloseButtonHint = 0x08000000, MacWindowToolBarButtonHint = 0x10000000, BypassGraphicsProxyWidget = 0x20000000, NoDropShadowWindowHint = 0x40000000, WindowFullscreenButtonHint = 0x80000000 };</pre></div>
<p>因此最终，我们可以在 <code>MainWindow.cpp</code> 中这样调用函数：</p>	<div data-bbox="638 1191 1576 1272"><pre>m_LoginDialog->setWindowFlags(Qt::CustomizeWindowHint Qt::FramelessWindowHint);</pre></div> <div data-bbox="638 1274 1576 1386"><p>1. <code>Qt::CustomizeWindowHint</code>： 这个标志允许用户修改窗口的默认外观，特别是窗口的边框和标题栏等。</p><p>2. <code>Qt::FramelessWindowHint</code>： 这个标志使窗口成为无边框窗口，意味着窗口没有默认的窗口边框、标题栏、最小化、最大化和关闭按钮。</p></div>

》》》》Unpolish 和 Polish 函数

对于 `QWidget` 对象，我们使用其成员函数获取 `Widget` 的 `Style`，`style()` 函数返回 `QStyle` 类型的值。因此我们可以调用 `QStyle` 的成员函数 `Polish/Unpolish()`。

```
// GUI style setting
QStyle *style() const;
void setStyle(QStyle *);
// Widget types and states
```

关于这两个函数的解释：

<code>unpolish()</code> ：	<div data-bbox="225 1805 544 1830"><pre>void QStyle::unpolish(QWidget *widget);</pre></div> <div data-bbox="213 1861 963 1944"><ul style="list-style-type: none">• 作用：将指定控件从样式中移除，停止对控件的样式处理。也可以理解为“撤销”该控件的样式应用。• 参数：一个 <code>QWidget</code> 指针，表示要从样式中撤销的控件。• 用法：当你需要清除控件的样式状态，重新调整控件外观时，使用这个函数。</div>
<code>polish()</code> ：	<div data-bbox="225 1957 528 1982"><pre>void QStyle::polish(QWidget *widget);</pre></div> <div data-bbox="213 2013 963 2125"><ul style="list-style-type: none">• 作用：将控件重新应用样式，确保控件根据当前的样式进行渲染。• 参数：一个 <code>QWidget</code> 指针，表示要重新应用样式的控件。• 用法：通常在控件的外观或样式发生了变化时（比如样式设置更改、控件属性修改等）调用此方法，使得控件能够重新渲染。</div>

》》》setProperty() 函数

setProperty() 是 Qt 中 QObject 类的一个成员函数，它用于为对象设置自定义的属性。
这是 Qt 提供的一种机制，使用该函数，可以动态地为任何继承自 QObject 的对象（包括控件和其他类）添加或修改属性。

函数签名：	bool QObject::setProperty(const char *name, const QVariant &value);
参数：	<div>◦ name: 自定义的属性的名称，类型为 const char*, 即该属性的字符串标识符。</div> <div>◦ value: 要设置的属性值，类型为 QVariant。（QVariant 是 Qt 中用于封装各种类型数据的类，它使得你可以将任何类型的数据存储在一个对象中。）</div> <div>例如：widget->setProperty("customProperty", 42); // 设置一个名为 "customProperty" 的自定义属性，值为 42</div>
返回值：	返回 true 表示属性设置成功，返回 false 表示设置失败。
作用：	setProperty() 允许你为 QObject 的派生类（如 QWidget）动态添加或修改属性。 你可以使用这种机制随时存储和检索与对象相关的数据，而不必定义额外的成员变量，并且这些属性可以通过对象的名字在代码中动态设置和获取。这对于需要在运行时，通过字符串动态访问属性的场景非常有用。

例一：

```
QWidget *widget = new QWidget;
widget->setProperty("customProperty", 42);                    // 设置一个名为 "customProperty" 的自定义属性，值为 42

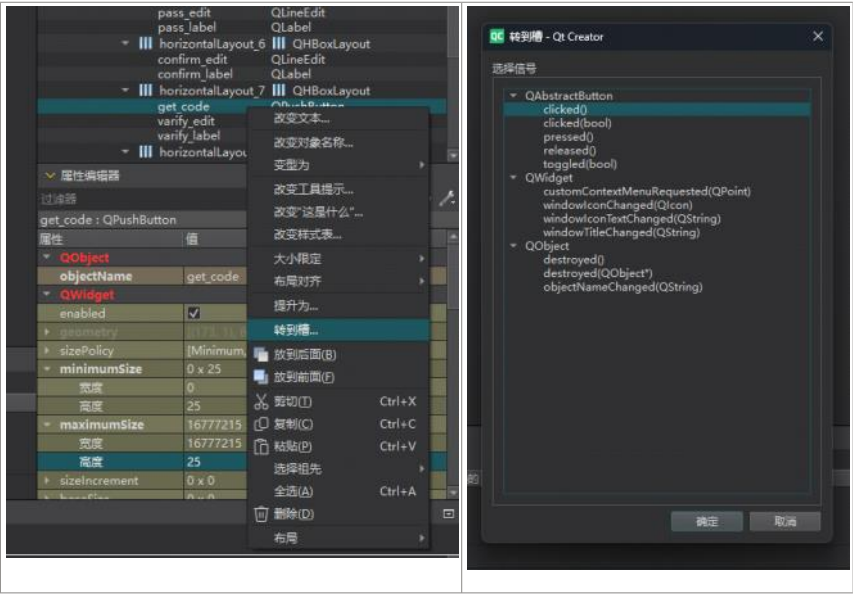
QVariant value = widget->property("customProperty");        // 获取属性 "customProperty" 的值
```

例二：

```
QWidget *widget = new QWidget;
widget->setProperty("customColor", QColor(255, 0, 0));        // 设置一个自定义颜色属性

// 在样式表中使用这个自定义属性
widget->setStyleSheet("background-color: qproperty(customColor);");
```

》》》手动为一个按钮编写一个单击的触发事件 和 直接在QT Creator中通过“转到槽”->选择Click()事件 有什么不同？



两者在呈现效果上并没有什么不同。
只不过手动编写 signals 和 slots 没有 QT 的自动化功能快捷。但手动编写事件触发函数可以带来更加灵活的方式。

》》》什么是正则表达式？正则表达式的规则是什么？

》》》正则表达式

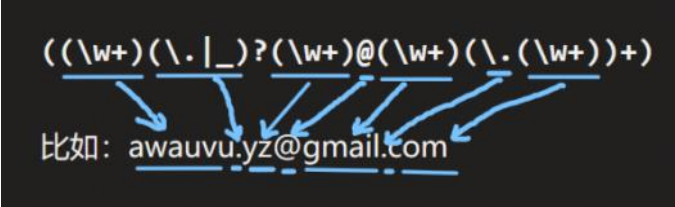
定义：

正则表达式（Regular Expression，简称 regex 或 regexp）是一种用于匹配字符串的模式。它由一些字符组成，利用这些字符可以定义复杂的匹配规则。并可以用来检查、查找、替换或操作字符串中的文本。

正则表达式的规则表：

普通字符：	字母、数字和标点符号等普通字符代表它们自己，如 a、1、# 等。							
特殊字符：	.:	匹配除换行符以外的任何单个字符。						
	^:	匹配输入字符串的开始位置。						
	\$:	匹配输入字符串的结束位置。						
	*:	匹配前一个字符零次或多次。						
	+:	匹配前一个字符一次或多次。						
	?:	匹配前一个字符零次或一次。						
	[]:	字符集，匹配括号内的任何字符，例如 [abc] 匹配 a、b 或 c。						
	:	表示“或”操作符，例如 a b 匹配 a 或 b。						
注意： 1 -> "." 是一个有意义的特殊字符，如果需要匹配".", 则需要对齐进行转移。 2 -> "**", "+", "?" 指的是匹配前一个字符一次或者多次，而不是前面所有字符一次或多次。 (比如((\w+)(_ .)?), 这里的 "?" 仅作用于(_ .)，即仅作用于"." "_" 二者的匹配。)								
转义字符：	\: 用来转义特殊字符，使其失去特殊含义，或者用于表示一些特殊字符，如： <table><tr><td>\d:</td><td>匹配一个数字，等价于 [0-9]。</td></tr><tr><td>\w:</td><td>匹配一个字母、数字或下划线，等价于 [A-Za-z0-9_]。</td></tr><tr><td>\s:</td><td>匹配一个空白字符（包括空格、制表符、换行符等）。</td></tr></table>		\d:	匹配一个数字，等价于 [0-9]。	\w:	匹配一个字母、数字或下划线，等价于 [A-Za-z0-9_]。	\s:	匹配一个空白字符（包括空格、制表符、换行符等）。
\d:	匹配一个数字，等价于 [0-9]。							
\w:	匹配一个字母、数字或下划线，等价于 [A-Za-z0-9_]。							
\s:	匹配一个空白字符（包括空格、制表符、换行符等）。							
量词：	{n}:	匹配前一个字符恰好出现 n 次，例如 a{3} 匹配 aaa。						
	{n,}:	匹配前一个字符至少出现 n 次，例如 a{2,} 匹配 aa、aaa、aaaa 等。						
	{n,m}:	匹配前一个字符出现 n 到 m 次，例如 a{2,4} 匹配 aa、aaa 或 aaaa。						
分组与捕获：	(): 用于分组，可以将多个字符组合成一个单元，进行整体匹配。分组还可以用于捕获匹配的内容，例如 (abc) 匹配 abc，并且可以获取匹配到的字符串。							

比如用于匹配邮箱的正则表达式：((\w+)(\.|_)?(\w+)@(\w+)(\.|_)(\w+))+)



》》对于末尾的理解

```
((\w+)(\.|_)?(\w+)@(\w+)(\.|_)(\w+))+)
```

这里的 “+” 指的是对 (\w+) 需要匹配多次，比如对于这个邮箱：user@mail.example.co.uk，我们需要对 (\w+) 匹配多次，因为我们会多次获取 example.co.uk

》》》QRegularExpression 是什么类型

QRegularExpression

QRegularExpression 是一个类，用于表示正则表达式对象。

(它是 Qt 5.0 引入的，用于替代早期版本中的 QRegExp 类，提供了更加现代的正则表达式支持，并且符合 ECMAScript 标准 (JavaScript 的正则表达式语法) 。

》》》match 和 hasMatch 函数

match() 函数	
函数原型：	QRegularExpressionMatch QRegularExpression::match(const QString &str, int offset = 0) const;
功能：	match() 用于检查字符串 str 是否与正则表达式模式匹配。它返回一个 QRegularExpressionMatch 对象，包含匹配结果的详细信息。
参数：	◦ str: 要匹配的字符串。 ◦ offset: 指定从哪个位置开始匹配（默认为 0）。
返回值：	返回一个 QRegularExpressionMatch 对象，如果匹配成功，则 QRegularExpressionMatch 对象包含匹配的细节，否则返回一个无效的匹配对象。
示例：	<pre>QRegularExpression re("\\d+"); // 匹配一个或多个数字 QRegularExpressionMatch match = re.match("1234"); if (match.hasMatch()) { qDebug() << "Matched!" << match.captured(0); // 输出匹配的内容 }</pre>

hasMatch() 函数	
函数原型:	bool QRegularExpression::hasMatch() const;
功能:	hasMatch() 用于检查正则表达式是否与输入的字符串匹配。它返回一个布尔值 true 或 false，指示是否有匹配。
返回值:	如果匹配成功，返回 true；否则返回 false。
示例:	<pre>QRegularExpression re("\\d+"); // 匹配一个或多个数字 QRegularExpressionMatch match = re.match("abc123"); if (match.hasMatch()) { qDebug() << "Match found!"; } else { qDebug() << "No match."; }</pre>

》》》》QT 中的 tr

```
#ifndef QT_NO_TRANSLATION
// full set of tr functions
# define QT_TR_FUNCTIONS \
    static inline QString tr(const char *s, const char *c = nullptr, int n = -1) \
    { return staticMetaObject.tr(s, c, n); }
#else
// inherit the ones from QObject
# define QT_TR_FUNCTIONS
#endif
```

Tr 是一个函数，当QT_TR_FUNCTIONS这个宏被调用之后，tr得以被定义。

tr的功能是:	tr用于实现字符串的国际化（i18n）。它将字符串标记为需要翻译的文本，并将其与应用程序的翻译文件进行关联。
用例:	<p>tr 通常用于 Qt 中的类（特别是 QObject 的子类）的方法和构造函数中。例如：</p> <pre>QString translatedText = tr("Hello, World!");</pre> <p>在这个例子中，"Hello, World!" 会被标记为一个待翻译的字符串。</p> <p>在应用程序运行时，如果存在相应语言的翻译文件（如 app_zh_CN.ts），这个字符串会被翻译成相应语言。</p>

》》》》一些误解:

这个是宏函数:	#define MAX(a, b) ((a) > (b) ? (a) : (b))
这个是宏:	# define QT_TR_FUNCTIONS static inline QString tr(const char *s, const char *c = nullptr, int n = -1) { return staticMetaObject.tr(s, c, n); }
	这个宏的作用是在调用 QT_TR_FUNCTIONS 时，会将 QT_TR_FUNCTIONS 与其之后的函数定义进行文本替换，从而在文件中定义一个函数。

----- Ep2 HTTP 管理者 -----

》》》》关于经典单例类和单例类的变体

经典单例:	<pre>class Singleton { private: static Singleton* instance; Singleton() {} // 私有构造函数 public: // 获取单例实例 static Singleton* GetInstance() { //..... return instance; } // 禁止拷贝构造和赋值操作，确保单例唯一性 Singleton(const Singleton&) = delete; Singleton& operator=(const Singleton&) = delete; }; // 初始化静态成员 Singleton* Singleton::instance = nullptr;</pre>
单例变体:	<pre>Template <typename T> class Singleton { private: static std::shared_ptr<T> instance; static std::mutex mtx; // 用于线程安全 Singleton() {} // 私有构造函数</pre>


```
public:
    // 获取单例实例，此处可以采用双重检查锁定
    static std::shared_ptr<T> GetInstance() {
        // .....
        return instance;
    }

    // 禁止拷贝构造和赋值操作，确保单例唯一性
    Singleton(const Singleton<T>&) = delete; // 取消复制功能（复制函数）
    Singleton<T>& operator= (const Singleton<T>&) = delete; // 禁止对 Singleton 进行赋值操作
};

// 初始化静态成员
std::shared_ptr<T> Singleton::instance = nullptr;
std::mutex Singleton::mtx;
```

Instance 可以是 Singleton 以外的类型，这取决于该单例类的使用方式。

- 》》 Singleton<T>& operator= (const Singleton<T>&) = delete;
- 》》 Singleton& operator= (const Singleton<T>& st) = delete;
- 》》 这两者有什么区别吗？

Singleton<T>& operator= (const Singleton<T>&) = delete;	是针对模板类的，它针对模板类的所有特化都禁止了赋值操作符。
Singleton& operator= (const Singleton<T>& st) = delete;	是针对特定类型的 Singleton 类的，它在这里表示禁止 Singleton 类型（非模板）对象之间的赋值。

但是在实际结果的呈现上，没有差别。

》》》 std::once_flag 和 std::call_once

std::once_flag:	std::once_flag 是 C++ 标准库中的同步工具，可以确保在多线程环境中某段代码（比如初始化函数） <u>只会执行一次</u> 。 std::once_flag 是一个标志对象，标记某个操作是否已经执行过，通常用于某个共享资源的初始化。
用法:	std::once_flag 通常与 std::call_once() 一起使用。你声明一个 std::once_flag 变量，然后在多线程代码中使用 std::call_once() 来确保某个代码块只执行一次。
代码示例:	std::once_flag flag;

std::call_once():	std::call_once() 是一个函数，确保传入的某个函数或代码块在多线程环境下只执行一次。 无论多少线程调用它，所传入的函数或代码块只会被执行一次，其它线程会等到第一次执行完成。
用法:	std::call_once() 接受一个 std::once_flag 对象和一个可调用对象作为参数。 即便是多个线程同时调用该函数，所传入的代码（或函数）也只会被执行一次，后续的线程会跳过这一执行。
示例:	<pre>#include <iostream> #include <thread> #include <mutex> std::once_flag flag; void initialize() { std::cout << "Initializing... Only once." << std::endl; } void thread_func() { std::call_once(flag, initialize); // 确保initialize()只执行一次 } int main() { std::thread t1(thread_func); std::thread t2(thread_func); std::thread t3(thread_func); t1.join(); t2.join(); t3.join(); return 0; }</pre>

工作原理:	<ul style="list-style-type: none">• std::once_flag 是一个特殊的对象，用来标记某个操作是否已经执行过。它通常是全局或静态的，在第一次调用时会被初始化。• std::call_once() 会检查 std::once_flag 是否已经标记过，如果没有，它就会执行传入的函数；如果已经执行过，则其他线程会跳过这个执行。这样确保了函数只执行一次。
-------	---

用处:

线程安全的初始化:	std::once_flag 和 std::call_once() 常用于保证某些初始化操作只执行一次，特别是在多线程环境下。例如，线程安全地初始化单例模式、全局资源或库加载等。
避免重复工作:	当多个线程尝试同时执行相同的初始化时，使用 std::call_once() 可以确保只执行一次初始化，避免了重复工作或多次初始化带来的潜在问题。
避免线程冲突:	std::call_once() 和 std::once_flag 可以确保代码只被执行一次，避免了多个线程同时执行某些敏感操作的竞态条件。

》》》std::share_ptr<T>(new T)和 make_ptr<T>()的区别?

前提:	<p>单例类的大致结构:</p> <pre> template <typename T> class Singleton { protected: Singleton() = default; Singleton(const Singleton<T>&) = delete; Singleton& operator=(const Singleton<T>& st) = delete; protected: static std::shared_ptr<T> s_Instance; public: static std::shared_ptr<T> GetInstance() { s_Instance = shared_ptr<T>(new T); return s_Instance; }; ~Singleton() { std::cout << "this is singleton destruct" << std::endl; } }; template <typename T> std::shared_ptr<T> Singleton<T>::s_Instance = nullptr; </pre> <p>假设我们需要一个单例对象，那么我们可以使用静态函数直接获得，像这样 GetInstance<HttpManager>(); 注意: Singleton<HttpManager> HttpMgr; 是错误的，单例类不可以创建对象，因为他的构造函数是私有的。</p>
问题:	<p>单例类的 Instance 对象一般为静态变量，并在类外声明为空。用户/程序需要使用 Instance 时，我们会使用 GetInstance() 函数，该函数会初始化 Instance 变量。</p> <p>如果在初始化 s_instance 时，使用 make_shared<T>() 来进行初始化操作，则有这样一个问题： 在我们设定的前提下，当我们使用 GetInstance<T>() 函数时，函数其实是 GetInstance<HttpManager>()，但由于 HttpManager 是单例类，因此 GetInstance<HttpManager>() 函数无法实现 make_shared 这个方法，因为 HttpManager 是单例类，他的构造函数是 private，而 make_shared 因此无法访问到 HttpManager 的构造函数。</p>
解决方式:	<p>因此，我们必须在初始化 s_Instance 时，事先获得 T 类（即单例的 HttpManager 类）的访问权限。</p> <p>所以我们选择在 GetInstance<T>() 函数中使用 std::share_ptr<T>(new T) 来初始化 s_Instance。虽然 std::share_ptr<T>(new T) 也会遭遇访问权限的问题，但如果我们可以提早在 T 类（即单例的 HttpManager 类）的定义中将 class Singleton<HttpMgr> 声明为友元，就可以在使用 std::share_ptr<T>(new T) 时获得 T 的访问权限，进而使用 T（即单例的 HttpManager 类）所隐藏的构造函数，避免 make_shared 会遇到的问题。</p>
提醒: 为什么是友元?	<p>虽然 HttpManager 类继承自父类 Singleton，但是父类不可以访问子类的 private 类型成员。 然而友元可以访问某一类的 private 类型成员。</p>

》》图示:

前提:	<p>单例类的大致结构:</p> <pre> template <typename T> class Singleton { protected: Singleton() = default; Singleton(const Singleton<T>&) = delete; Singleton& operator=(const Singleton<T>& st) = delete; protected: static std::shared_ptr<T> s_Instance; public: static std::shared_ptr<T> GetInstance() { s_Instance = shared_ptr<T>(new T); return s_Instance; }; ~Singleton() { std::cout << "this is singleton destruct" << std::endl; } }; template <typename T> std::shared_ptr<T> Singleton<T>::s_Instance = nullptr; </pre> <p>假设我们需要一个单例对象，那么我们可以使用静态函数直接获得，像这样 GetInstance<HttpManager>(); 注意: Singleton<HttpManager> HttpMgr; 是错误的，单例类不可以创建对象，因为他的构造函数是私有的。</p>
问题:	<p>单例类的 Instance 对象一般为静态变量，并在类外声明为空。用户/程序需要使用 Instance 时，我们会使用 GetInstance() 函数，该函数会初始化 Instance 变量。</p> <p>如果在初始化 s_instance 时，使用 make_shared<T>() 来进行初始化操作，则有这样一个问题： 在我们设定的前提下，当我们使用 GetInstance<T>() 函数时，函数其实是 GetInstance<HttpManager>()，但由于 HttpManager 是单例类，因此 GetInstance<HttpManager>() 函数无法实现 make_shared 这个方法，因为 HttpManager 是单例类，他的构造函数是 private，而 make_shared 因此无法访问到 HttpManager 的构造函数。</p>

》》》如何在 使用 cmake 的 QT 项目中，添加外部库？（模块）

找到 `find_package`（这个函数出现了两次）， 还有 `target_link_libraries`

```
12
13 find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets Network)
14 find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets Network)
15
4
5 target_link_libraries(JustinChat PRIVATE Qt${QT_VERSION_MAJOR}::Widgets Qt${QT_VERSION_MAJOR}::Network)
6
```

他们分别进行以下操作：（详细可以自行查阅 cmake 文档）

- 1 加入组件
- 2 根据 QT 版本加载对应组件的模块
- 3 链接模块

》》》CRTP? CRTP? CRTP?

Eg. Class `HttpMgr` : public Qobject, public Singleton<`HttpMgr`> ,在 CRTP 技术的加持下，`HttpMgr` 可以继承他自己。

CRTP (Curiously Recurring Template Pattern)	是C++中一种常见的编程技巧，它通过模板继承实现了类与类之间的递归继承模式。简单来说，CRTP是一种让类自己作为模板参数的技巧。			
CRTP的基本思想	在CRTP中，一个类（通常是派生类）会作为模板参数被传递给基类。基类通常是一个模板类，模板参数就是派生类本身。通过这种方式，基类可以访问派生类的成员或方法。			
CRTP的例子	<p>假设我们有一个类 <code>Derived</code>，我们希望通过一个基类 <code>Base</code> 来给它添加一些功能。我们可以通过CRTP实现：</p> <table border="1"><tr><td><p>定义（模板类）：</p><pre>template <typename T> class Base { public: void interface() { // 调用派生类的方法 static_cast<T*>(this)->implementation(); } // 基类自己的一些方法 void base_method() { std::cout << "Base class method\n"; } };</pre></td><td><p>若接受派生类作为模板参数，则代码会是这种形式：</p><pre>template <typename Derived> class Base { public: void interface() { // 调用派生类的方法 static_cast<Derived*>(this)->implementation(); } // 基类自己的一些方法 void base_method() { std::cout << "Base class method\n"; } };</pre></td></tr></table> <pre>// 派生类继承自 Base 并传递自身作为模板参数 class Derived : public Base<Derived> { public: void implementation() { std::cout << "Derived class implementation\n"; } }; int main() { Derived d; d.interface(); // 调用派生类实现的方法 d.base_method(); // 调用基类方法 return 0; }</pre>		<p>定义（模板类）：</p> <pre>template <typename T> class Base { public: void interface() { // 调用派生类的方法 static_cast<T*>(this)->implementation(); } // 基类自己的一些方法 void base_method() { std::cout << "Base class method\n"; } };</pre>	<p>若接受派生类作为模板参数，则代码会是这种形式：</p> <pre>template <typename Derived> class Base { public: void interface() { // 调用派生类的方法 static_cast<Derived*>(this)->implementation(); } // 基类自己的一些方法 void base_method() { std::cout << "Base class method\n"; } };</pre>
<p>定义（模板类）：</p> <pre>template <typename T> class Base { public: void interface() { // 调用派生类的方法 static_cast<T*>(this)->implementation(); } // 基类自己的一些方法 void base_method() { std::cout << "Base class method\n"; } };</pre>	<p>若接受派生类作为模板参数，则代码会是这种形式：</p> <pre>template <typename Derived> class Base { public: void interface() { // 调用派生类的方法 static_cast<Derived*>(this)->implementation(); } // 基类自己的一些方法 void base_method() { std::cout << "Base class method\n"; } };</pre>			

还挺绕的，不是么。

》》》QNetworkAccessManager 类

<QNetworkAccessManager> 是 Qt 库中的一个头文件，它定义了 QNetworkAccessManager 类。

该类是 Qt 网络模块（QtNetwork）的一部分，提供了管理网络请求（例如 HTTP 请求）和响应的功能。主要用于处理应用程序与网络进行交互的任务，如发起 HTTP 请求、发送表单数据、上传文件、下载数据等。它提供了对不同类型的网络请求（如 GET、POST、PUT 等）的支持，并处理与远程服务器之间的通信。

作用：

主要功能：

发起网络请求:	QNetworkAccessManager 可以发起各种类型的网络请求, 包括 GET、POST、PUT、DELETE 等。你可以使用它来访问 Web 服务、下载文件等。
处理请求的响应:	通过与 QNetworkReply 对象结合, QNetworkAccessManager 可以接收请求的响应, 并处理返回的数据 (例如 JSON、HTML、XML 等格式)。
支持异步操作:	它支持异步请求, 即在发起请求时不会阻塞程序的其他部分, 允许你继续执行其他操作。请求完成时, 会触发信号, 你可以使用这些信号来处理响应或错误。

常见方法:

get():	发起一个 GET 请求, 通常用于获取数据。
post():	发起一个 POST 请求, 通常用于发送数据到服务器。
put():	发起一个 PUT 请求, 通常用于更新数据。
deleteResource():	发起一个 DELETE 请求, 用于删除服务器上的资源。

事件与信号:

finished():	当网络请求完成时, 这个信号会被发射, 可以用来处理返回的数据或错误信息。
errorOccurred():	如果网络请求出错, 这个信号会被触发, 允许你处理错误。

》》》》关于 QT 中的信号和槽机制的启用。QT 中 signals slots 函数的要求。QT 中的 connect() 的解释。

》》》》111111

》》》》Q_OBJECT 的定义及其作用

```
153  /* qmake ignore Q_OBJECT */
154  #define Q_OBJECT \
155  public: \
156      QT_WARNING_PUSH \
157      Q_OBJECT_NO_OVERRIDE_WARNING \
158      static const QMetaObject staticMetaObject; \
159      virtual const QMetaObject *metaObject() const; \
160      virtual void *qt_metacast(const char *); \
161      virtual int qt_metacall(QMetaObject::Call, int, void **); \
162      QT_TR_FUNCTIONS \
163  private: \
164      Q_OBJECT_NO_ATTRIBUTES_WARNING \
165      Q_DECL_HIDDEN_STATIC_METACALL static void qt_static_metacall(QObject *, QMetaObject::Call, int, void **); \
166      QT_WARNING_POP \
167      struct QPrivateSignal {}; \
168      QT_ANNOTATE_CLASS(qt_qobject, "")
169
```

- 作用
- 支持信号与槽的机制。
 - 支持运行时元信息、反射。
 - 支持动态属性和事件处理。
 - 允许 MOC 自动生成代码, 支持自动化的信号和槽连接。

解释

1. 信号和槽机制是否可用:

- 定义了 Q_OBJECT:**
你可以使用 Qt 的信号和槽机制。 (信号可以被发射, 槽可以被调用, 且可以通过 QObject::connect() 方法将信号和槽连接在一起。) 信号和槽机制支持跨线程调用。 (Qt 会自动处理线程间的信号和槽连接, 确保在正确的线程中调用槽。)
- 没有定义 Q_OBJECT:**
信号和槽机制无法使用。即使你在类中定义了 signals 和 public slots, Qt 也不会为它们生成相关的代码。 (这意味着你无法通过 connect() 来连接信号和槽, 或者使用 emit 来发射信号。) 信号和槽无法跨线程使用, 或者可能无法在不同的线程中正常工作。

2. MOC 生成的代码:

- 定义了 Q_OBJECT:**
Qt 的元对象编译器 (MOC) 会自动为你生成与信号、槽和元信息相关的代码, 并使其能够正确地参与 Qt 的信号和槽机制。 在运行时, Qt 可以动态地查询类的元信息和处理信号槽的连接。 (例如, 信号的实现和槽的连接都由 MOC 生成, metaObject() 方法可以用于动态查询类的元信息。)
- 没有定义 Q_OBJECT:**
MOC 不会生成这些代码, 你的类不会有与信号和槽相关的支持, 也无法查询元信息。 (你无法在运行时通过反射访问类的信号、槽或属性, 也不能使用 Qt 提供的信号和槽连接机制。)

3. 动态属性和反射功能:


- 定义了 Q_OBJECT:**
你可以使用 Qt 的动态属性系统, 比如通过 setProperty() 和 property() 来设置和获取对象的属性。此外, 可以使用 QMetaObject 来查询类的元信息, 如类名、信号、槽等。
- 没有定义 Q_OBJECT:**
你的类将没有这些功能, 因为 MOC 生成的代码包含了必要的元信息和属性处理机制。

没有 Q_OBJECT 时的后果:

如果你在类中没有定义 Q_OBJECT, 并尝试使用信号和槽, 程序将编译失败。你可能看到类似以下的错误:

“signal is not a member of class” 或 “no matching function for call to connect” 这些错误会提示你在类中没有启用信号和槽的功能。

》》》》！！！！》》将一个类定义为: 继承自 QObject 类 和 在类中声明宏定义 Q_OBJECT, 这两个步骤必须搭配使用吗? 是否可以单独使用? 如果可以单独使用, 两者有什么不同?

图示：	
是否必须搭配使用？	是的，这两者必须搭配使用，缺一不可。如果只使用其中之一，类的信号与槽机制或其他QT功能将无法正常工作。
两者的功能：	继承 <code>QObject</code> 类 -> 让类成为 Qt 对象，具备 Qt 的基础功能。 声明 <code>Q_OBJECT</code> 宏 -> 启用元对象系统，使得类能够支持信号与槽、反射机制等高级特性。

》》》》》QT 中的信号函数一般在什么时候发出信号？怎样发出信号？槽函数通过怎样的手段得知信号已经被发出？并接受该信号？

signal	
如何发送信号：	在 Qt 中，信号函数本身并不直接发出信号，信号的发出是通过调用 <code>emit</code> 关键字实现的。
何时发送信号：	我们必须在合适的地方使用 <code>emit</code> 来发出信号。（它通常是在特定条件满足时，由用户选择使用 <code>emit()</code> 发出信号。）
示例：	<pre>void MyClass::someFunction() { if(...) { // emit mySignal(); // 发出信号 } }</pre>

slot	
如何接受信号：	你需要通过 <code>QObject::connect()</code> 函数将信号与槽关联起来，连接后，信号触发时会调用对应的槽函数。
示例：	<pre>MyClass obj; connect(&obj, &MyClass::someFunction, &someObject, &SomeClass::someSlot);</pre>

》》》》2222222222

》》》》QT 中槽函数和信号函数的参数需要匹配，以下是其规则：

1. 参数类型匹配	信号和槽的参数类型必须匹配，即信号发射时传递的参数类型必须与槽函数中定义参数类型一致。 (括引用传递：如果信号发出的是某种类型的引用（如 <code>QString&</code> ），那么槽函数也必须接受同样类型的引用。)
2. 参数个数匹配	信号和槽的参数个数必须匹配。即信号定义参数数量和槽函数定义参数数量必须相等。
3. 默认参数	如果槽函数的参数有默认值，信号发射时可以不传递这些参数，但如果槽函数没有默认值，则必须在连接时传递对应的参数。
4. 如果信号发出的参数与槽的接收参数不同：	Qt 会尝试进行类型转换。例如，如果信号发出的参数类型是 <code>int</code> ，但槽函数的参数是 <code>double</code> ，Qt 会自动进行转换，但这仅在 Qt 支持的类型转换之间有效。 (如果类型不兼容，Qt 会报错，且信号与槽无法连接。)

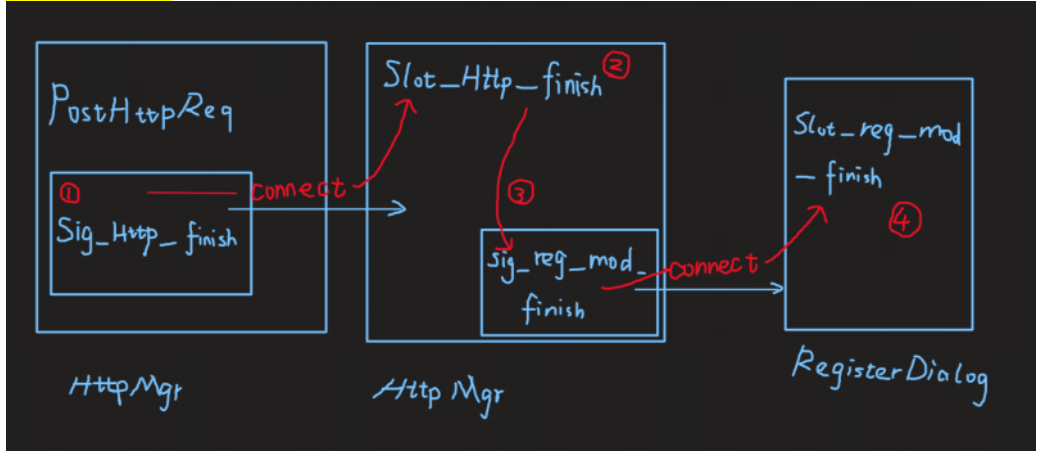
》》》》3333333333

》》》》`QObject::connect` 函数的参数，示意，重载，使用方法？

1. <code>QObject::connect</code> 函数的基本语法：	函数签名： <code>QObject::connect(sender, signal, receiver, slot);</code> <ul style="list-style-type: none">• <code>sender</code>: 发射信号的对象（通常是一个继承自 <code>QObject</code> 的类的实例）。• <code>signal</code>: 信号的名称。需要使用 Qt 的信号和槽机制进行声明，通常是 <code>SIGNAL()</code> 宏包裹的信号名称。• <code>receiver</code>: 接收信号的对象（通常是一个继承自 <code>QObject</code> 的类的实例）。• <code>slot</code>: 槽的名称，通常是 <code>SLOT()</code> 宏包裹的槽函数名称。						
2. <code>QObject::connect</code> 函数的重载： Qt 中的 <code>QObject::connect</code> 有多个重载版本，主要根据信号和槽的参数传递方式有所不同。以下是一些常见的重载版本：	<table><tr><td>2.1 基本版本（旧版信号和槽机制）：</td><td>这种连接方式是 Qt 经典的信号和槽机制（老版本）。 函数签名： <code>QObject::connect(sender, SIGNAL(signal_name(ArgType)), receiver, SLOT(slot_name(ArgType)));</code> 函数参数：<ul style="list-style-type: none">• <code>sender</code>: 信号发送方 (<code>QObject</code>)。• <code>signal_name</code>: 信号名称，必须是 <code>SIGNAL()</code> 宏的形式。• <code>receiver</code>: 信号接收方 (<code>QObject</code>)。• <code>slot_name</code>: 槽函数名称，必须是 <code>SLOT()</code> 宏的形式。</td></tr><tr><td>2.2 新版版本（基于函数指针的连接）：</td><td>Qt 5 引入了基于函数指针的新版本信号和槽连接，这种方式比传统的宏方式更安全，类型检查更严格，避免了运行时错误。 在新版本中，信号和槽的连接是类型安全的，编译时会检查信号和槽参数是否匹配。 函数签名： <code>QObject::connect(sender, &Sender::signal_name, receiver, &Receiver::slot_name);</code> 函数参数：<ul style="list-style-type: none">• <code>sender</code>: 信号发送方 (<code>QObject</code>)。• <code>signal_name</code>: 信号名称，是成员函数指针。• <code>receiver</code>: 信号接收方 (<code>QObject</code>)。• <code>slot_name</code>: 槽函数名称，是成员函数指针。</td></tr><tr><td>2.3 重载版本（支持 lambda 函数）：</td><td>这种方式允许使用 lambda 函数来作为槽，具有更高的灵活性和简洁性。 函数签名： <code>QObject::connect(sender, &Sender::signal_name, [=](ArgType arg){ // lambda 表达式中处理信号 });</code> 函数参数：<ul style="list-style-type: none">1. <code>sender</code>: 指向信号发送者对象的指针。它是发出信号的 <code>QObject</code> 类的实例。2. <code>&Sender::signal_name</code> (信号)</td></tr></table>	2.1 基本版本（旧版信号和槽机制）：	这种连接方式是 Qt 经典的信号和槽机制（老版本）。 函数签名： <code>QObject::connect(sender, SIGNAL(signal_name(ArgType)), receiver, SLOT(slot_name(ArgType)));</code> 函数参数： <ul style="list-style-type: none">• <code>sender</code>: 信号发送方 (<code>QObject</code>)。• <code>signal_name</code>: 信号名称，必须是 <code>SIGNAL()</code> 宏的形式。• <code>receiver</code>: 信号接收方 (<code>QObject</code>)。• <code>slot_name</code>: 槽函数名称，必须是 <code>SLOT()</code> 宏的形式。	2.2 新版版本（基于函数指针的连接）：	Qt 5 引入了基于函数指针的新版本信号和槽连接，这种方式比传统的宏方式更安全，类型检查更严格，避免了运行时错误。 在新版本中，信号和槽的连接是类型安全的，编译时会检查信号和槽参数是否匹配。 函数签名： <code>QObject::connect(sender, &Sender::signal_name, receiver, &Receiver::slot_name);</code> 函数参数： <ul style="list-style-type: none">• <code>sender</code>: 信号发送方 (<code>QObject</code>)。• <code>signal_name</code>: 信号名称，是成员函数指针。• <code>receiver</code>: 信号接收方 (<code>QObject</code>)。• <code>slot_name</code>: 槽函数名称，是成员函数指针。	2.3 重载版本（支持 lambda 函数）：	这种方式允许使用 lambda 函数来作为槽，具有更高的灵活性和简洁性。 函数签名： <code>QObject::connect(sender, &Sender::signal_name, [=](ArgType arg){ // lambda 表达式中处理信号 });</code> 函数参数： <ul style="list-style-type: none">1. <code>sender</code>: 指向信号发送者对象的指针。它是发出信号的 <code>QObject</code> 类的实例。2. <code>&Sender::signal_name</code> (信号)
2.1 基本版本（旧版信号和槽机制）：	这种连接方式是 Qt 经典的信号和槽机制（老版本）。 函数签名： <code>QObject::connect(sender, SIGNAL(signal_name(ArgType)), receiver, SLOT(slot_name(ArgType)));</code> 函数参数： <ul style="list-style-type: none">• <code>sender</code>: 信号发送方 (<code>QObject</code>)。• <code>signal_name</code>: 信号名称，必须是 <code>SIGNAL()</code> 宏的形式。• <code>receiver</code>: 信号接收方 (<code>QObject</code>)。• <code>slot_name</code>: 槽函数名称，必须是 <code>SLOT()</code> 宏的形式。						
2.2 新版版本（基于函数指针的连接）：	Qt 5 引入了基于函数指针的新版本信号和槽连接，这种方式比传统的宏方式更安全，类型检查更严格，避免了运行时错误。 在新版本中，信号和槽的连接是类型安全的，编译时会检查信号和槽参数是否匹配。 函数签名： <code>QObject::connect(sender, &Sender::signal_name, receiver, &Receiver::slot_name);</code> 函数参数： <ul style="list-style-type: none">• <code>sender</code>: 信号发送方 (<code>QObject</code>)。• <code>signal_name</code>: 信号名称，是成员函数指针。• <code>receiver</code>: 信号接收方 (<code>QObject</code>)。• <code>slot_name</code>: 槽函数名称，是成员函数指针。						
2.3 重载版本（支持 lambda 函数）：	这种方式允许使用 lambda 函数来作为槽，具有更高的灵活性和简洁性。 函数签名： <code>QObject::connect(sender, &Sender::signal_name, [=](ArgType arg){ // lambda 表达式中处理信号 });</code> 函数参数： <ul style="list-style-type: none">1. <code>sender</code>: 指向信号发送者对象的指针。它是发出信号的 <code>QObject</code> 类的实例。2. <code>&Sender::signal_name</code> (信号)						

	<p>这是发送者类（Sender）中定义的信号。信号是发送者对象通过某些事件或状态变化发出的通知，通常是在 Sender 类内部通过 signals 关键字定义的。例如，可能是 clicked、valueChanged 等信号。</p> <p>3. [=](ArgType arg) (lambda 表达式)：作为槽函数来处理信号。当信号发出时，lambda 表达式会被调用。</p>
2.4 连接类型（传递方式）：	Qt 还提供了几种 connect 的重载形式来指定信号和槽的调用方式：
函数签名：	QObject::connect(sender, &Sender::signal_name, receiver, &Receiver::slot_name, Qt::ConnectionType type);
函数参数：	<ul style="list-style-type: none">Qt::AutoConnection（默认）：根据信号和槽的执行线程自动选择连接方式。Qt::DirectConnection：直接在发出信号的线程中调用槽。Qt::QueuedConnection：将槽调用放入接收者线程的事件队列中。Qt::BlockingQueuedConnection：与 QueuedConnection 类似，但会阻塞直到槽函数执行完成。

设计流程图示



PostHttpRequest 函数参数定义的规则：（为什么参数是这样定义的？为了实现什么操作而定义？）

我们先实现PostHttpRequest请求的函数，也就是发送http的post请求。发送请求要用到请求的url，请求的数据(json或者protobuf序列化)，以及请求的id，以及哪个模块发出的请求mod，那么一个请求接口应该是这样的

```
void PostHttpRequest(QUrl url, QJsonObject json, ReqId req_id, Modules mod);
```

为什么称 Url 为路由？

URL（Uniform Resource Locator，统一资源定位符）被称为路由（route）是因为它在 Web 应用中承担了将请求映射到特定资源或功能的作用。简单来说，URL 本质上是 Web 应用中的“路由路径”，它定义了如何通过请求路径（URL）找到对应的页面、API 或其他资源。

为什么 URL 可以是路由

请求路径的映射功能：	<p>在 Web 开发中，URL 起到了路由的作用，它指引了客户端请求向服务器发送请求后，服务器应如何处理该请求。</p> <p>URL 通常包含路径、查询参数等信息，这些信息帮助 Web 应用确定用户请求的具体资源或服务。通过这个路径，服务器或框架会根据路由规则决定应该调用哪个处理程序（如控制器或方法）。</p> <p>例如，URL https://example.com/products/123 通常表示用户请求的是 products 路径下的某个产品，服务器会根据这个路径找到相关的资源（如数据库中的商品信息）。</p>
URL 结构：	<p>URL 的结构中，路径部分通常就是路由的体现。例如：</p> <ul style="list-style-type: none">o /home：指向网站的首页。o /profile/123：指向用户 ID 为 123 的个人资料页面。o /api/v1/users：指向 API 的用户数据资源。 <p>每个路径都会映射到 Web 应用中的一个具体的处理程序或控制器方法，即所谓的“路由处理”。</p>

》》》HTTP 请求的格式:

定义:	HTTP 请求的格式由 请求行 (Request Line)、请求头 (Headers)、空行 (CRLF) 和请求体 (Body) 四部分组成。
基本格式:	<请求行> <请求头> <空行> // 通过回车换行 (CRLF, 即 \r\n) 分隔头和体 <请求体> // 可选 (如 POST 请求携带数据)
示例 (GET 请求) 通常用于从服务器中获取数据	GET /api/user?id=123 HTTP/1.1 Host: example.com User-Agent: Mozilla/5.0 Accept: application/json Connection: keep-alive
示例 (POST请求) 通常用于向服务器发送数据	POST /api/login HTTP/1.1 Host: example.com Content-Type: application/json Content-Length: 45 { "username": "alice", "password": "123456" }

细则:

(1) 请求行 (Request Line)

格式:	<Method> <Request-URI> <HTTP-Version>								
字段说明:	<table><tr><td>字段</td><td>说明</td></tr><tr><td>Method</td><td>请求方法 (如 GET, POST, PUT, DELETE, HEAD)。</td></tr><tr><td>Request-URI</td><td>请求的资源路径 (如 /index.html 或带参数的 /search?q=hello)。</td></tr><tr><td>HTTP-Version</td><td>HTTP 协议版本 (如 HTTP/1.1 或 HTTP/2)。</td></tr></table>	字段	说明	Method	请求方法 (如 GET, POST, PUT, DELETE, HEAD)。	Request-URI	请求的资源路径 (如 /index.html 或带参数的 /search?q=hello)。	HTTP-Version	HTTP 协议版本 (如 HTTP/1.1 或 HTTP/2)。
字段	说明								
Method	请求方法 (如 GET, POST, PUT, DELETE, HEAD)。								
Request-URI	请求的资源路径 (如 /index.html 或带参数的 /search?q=hello)。								
HTTP-Version	HTTP 协议版本 (如 HTTP/1.1 或 HTTP/2)。								
示例:	GET /index.html HTTP/1.1								

(2) 请求头 (Headers)

格式:	<Header-Name>: <Header-Value> (每行一个键值对)																		
常见请求头:	<table><tr><td>请求头</td><td>作用</td></tr><tr><td>Host</td><td>目标服务器域名 (HTTP/1.1 必需字段)。</td></tr><tr><td>User-Agent</td><td>客户端标识 (如浏览器类型、操作系统)。</td></tr><tr><td>Accept</td><td>声明客户端可接收的响应数据类型 (如 text/html, application/json)。</td></tr><tr><td>Content-Type</td><td>请求体的数据类型 (如 application/json, application/x-www-form-urlencoded)。</td></tr><tr><td>Content-Length</td><td>请求体的字节数 (POST/PUT 必需)。</td></tr><tr><td>Authorization</td><td>身份验证凭证 (如 Bearer <token>)。</td></tr><tr><td>Cookie</td><td>客户端携带的 Cookie 信息。</td></tr><tr><td>Connection</td><td>控制连接行为 (如 keep-alive 保持长连接)。</td></tr></table>	请求头	作用	Host	目标服务器域名 (HTTP/1.1 必需字段)。	User-Agent	客户端标识 (如浏览器类型、操作系统)。	Accept	声明客户端可接收的响应数据类型 (如 text/html, application/json)。	Content-Type	请求体的数据类型 (如 application/json, application/x-www-form-urlencoded)。	Content-Length	请求体的字节数 (POST/PUT 必需)。	Authorization	身份验证凭证 (如 Bearer <token>)。	Cookie	客户端携带的 Cookie 信息。	Connection	控制连接行为 (如 keep-alive 保持长连接)。
请求头	作用																		
Host	目标服务器域名 (HTTP/1.1 必需字段)。																		
User-Agent	客户端标识 (如浏览器类型、操作系统)。																		
Accept	声明客户端可接收的响应数据类型 (如 text/html, application/json)。																		
Content-Type	请求体的数据类型 (如 application/json, application/x-www-form-urlencoded)。																		
Content-Length	请求体的字节数 (POST/PUT 必需)。																		
Authorization	身份验证凭证 (如 Bearer <token>)。																		
Cookie	客户端携带的 Cookie 信息。																		
Connection	控制连接行为 (如 keep-alive 保持长连接)。																		
示例:	Host: example.com User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Accept: /*/* Content-Type: application/json																		

(3) 空行

作用:	分隔请求头和请求体, 通过 \r\n (CRLF) 表示。
示例:	Headers... \r\n Body...

(4) 请求体 (Body)

适用场景:	POST、PUT、PATCH 等方法需要发送数据时使用。									
数据格式:	由 Content-Type 在请求头中指定。 <table><tr><td>Content-Type</td><td>数据格式示例</td></tr><tr><td>application/json</td><td>{ "name": "Alice", "age": 25 }</td></tr><tr><td>application/x-www-form-urlencoded</td><td>username=alice&password=123</td></tr><tr><td>multipart/form-data</td><td>文件上传 (边界分隔不同部分)。</td></tr></table>		Content-Type	数据格式示例	application/json	{ "name": "Alice", "age": 25 }	application/x-www-form-urlencoded	username=alice&password=123	multipart/form-data	文件上传 (边界分隔不同部分)。
Content-Type	数据格式示例									
application/json	{ "name": "Alice", "age": 25 }									
application/x-www-form-urlencoded	username=alice&password=123									
multipart/form-data	文件上传 (边界分隔不同部分)。									
示例:	POST /submit-form HTTP/1.1 Content-Type: application/x-www-form-urlencoded Content-Length: 23 username=alice&age=30									

》》》一个 QNetworkRequest 类型的对象是否必须要事先设置 Url ?

```
QNetworkRequest request(url);
```

是的, 网络请求的核心就是需要知道请求去哪里, QNetworkRequest 的 URL 是必须要设置的, 因为它指明了请求的目标地址。

如果没有设置 URL，发起请求时会缺少目标地址，导致请求无法发送（HTTP 请求（如 GET、POST）需要知道请求的目的地和路径）。

》》是否也必须设置其他信息？

对于其他成员，并不是必须设置的，它们是可选的，取决于你需要执行的具体需求。

常见：	
1.请求头	请求头是可选的，但在许多情况下是需要的，尤其是当你需要指定请求的内容类型（如 Content-Type）或身份验证信息（如 Authorization）时。你可以通过 setHeader() 方法设置请求头。如果不设置，默认情况下会使用系统的默认请求头。
2. 请求方法（HTTP Method）	默认情况下，QNetworkRequest 使用的是 GET 方法。你不需要显式设置 HTTP 方法。除非你想发送 POST、PUT 等其他类型的请求，则需要使用 QNetworkRequest::setAttribute() 或其他方式来修改请求的行为。
3. 请求体/请求数据（Body）	请求的数据体（即请求的 payload）通常在 POST 或 PUT 请求中是必需的，但对于 GET 请求通常不需要。你可以使用 QNetworkAccessManager 来发送数据。对于 POST 请求，数据通常通过 QByteArray 传递。
不常见：	
4. Cookies	QNetworkRequest 允许你设置 cookies，这对于会话管理等操作非常有用。但 cookies 也是可选的，除非你的请求需要特定的 cookies 来维持会话状态。
5. 超时设置	默认情况下，QNetworkRequest 没有设置超时，你可以通过网络管理器（QNetworkAccessManager）来设置请求超时，但这也是可选的。如果不设置，默认情况下 Qt 的网络库会根据网络状态自动管理超时。
6. 缓存控制	你可以设置缓存控制策略，这通常用于 HTTP 请求中的缓存相关头部，但这也是可选的。如果不手动设置缓存控制，默认情况下 Qt 会自动处理缓存。

》》》从 QJsonObject 类型 -> QJsonDocument()处理之后的 QJsonDocument 类型 -> toJson()处理之后

》》》jsonObj 这个变量究竟是怎样变化的？

```
application/json");
QJsonDocument(jsonObj).toJson());
```

1 QJsonObject 类型	<p>从 QJsonObject 的构造函数中可以看到：</p> <p>当 jsonObj 属于 QJsonObject 这个类时，jsonObj 这个变量其实被存放在 pair 类型中。而 Pair 是一种数据结构，是一种键值对。</p> <pre>QJsonObject(std::initializer_list<QPair<QString, QJsonValue> > args);</pre> <p>对于 QJsonObject 类型的对象我们可以这样处理：（并以此从 QJsonObject 类型对象中，得到可使用的变量）</p> <pre>// 访问 QJsonObject 中的内容 QString name = jsonObj["name"].toString(); // "Alice" int age = jsonObj["age"].toInt(); // 30 QDebug() << "Name:" << name; QDebug() << "Age:" << age;</pre>
2 QJsonDocument 类型	经过 QJsonDocument 封装之后，jsonObj 依旧是一个内存中的数据结构。（通常是一些对象、字符串、数组的组合，包括 Qobject、QJsonArray）
3 toJson()	<p>经过 toJson 函数转换之后，数据变为 QByteArray 类型。</p> <p>toJson() 函数的定义：</p> <pre>#if !defined(QT_JSON_READONLY) defined(Q_CLANG_QDOC) QByteArray toJson(JsonFormat format = Indented) const; #endif</pre>

》》》》reply 是什么格式，记录了什么数据？使用 readAll() 可以返回什么样的数据？

<pre>//发送请求并获取回执、处理响应(发送请求时，使用 POST 类型请求) QNetworkReply* reply = m_Manager.post(request, data);</pre>	reply 得到的数据是服务器处理 http 信号之后，返回的原始数据。但是这些信息存放在 QNetworkReply 对象中，于是我们需要使用一个方法读出有效信息。
<pre>// 无错误则读取请求 QString res = reply->readAll();</pre>	使用 readAll() 函数处理之后，我们得到有效信息的二进制字节流(QByteArray)。比如 (0xFF, 0xD8, 0xFF, 0xE0, 0x00, 0x10, 0x4A, 0x46, ...)
<pre>// 无错误则读取请求 QString res = reply->readAll();</pre>	最后我们将其结果转化为 QString，转化为 res（呈现为人类可读的文本类型）

》》》 QByteArray::number() 有什么作用?

<pre>application/json"}; data.length(); instance-1</pre>	<pre>application/json"); , QByteArray::number(data.length());};</pre>
data.length() 返回的是 data 的长度，但格式为整数。	QByteArray 的 number 方法会将这个整数转换为字符串格式。
<pre>QByteArray data = "hello"; data.length(); // 5</pre>	<pre>QByteArray data = "hello"; QByteArray::number(data.length()); // "5"</pre>

》》》注意 这里引用的 finished() 定义是一个函数，不过 connect 要求填入的参数是函数指针，所以写做 &QNetworkReply::finished

```
QObject::connect(reply, &QNetworkReply::finished, [reply, reqID, mod, this]()
{
    if(reply->error() != QNetworkReply::NoError)
    {

        emit this->SigHttpFinish("", reqID, mod, ErrorCode::ERR_NETWORK);
    }
});
```

》》》 deleteLater() 是什么函数?

作用:	deleteLater() 是 Qt 中 QObject 类的一个成员函数。它用于 标记一个对象将在事件循环的某个时刻被删除。简而言之，它并不会立即删除对象，而是把删除操作安排到下一次事件循环中执行。
原因:	在 Qt 中，不建议直接在函数中调用 delete 删除对象，因为这可能会导致在对象的事件和信号槽机制运行时删除对象，从而出现访问空指针的风险。使用 deleteLater() 则可以确保删除操作在适当的时机发生，避免在当前执行的代码中立即删除对象。

》》》 enable_shared_from_this 什么类? 在哪里定义?

定义:	enable_shared_from_this 是一个 C++ 标准库 中的模板类，它位于 <memory> 头文件中。 <div><pre>template <class _Ty> class enable_shared_from_this { // provide member functions that create shared_ptr to this public: using _Esft_type = enable_shared_from_this; _NODISCARD shared_ptr<_Ty> shared_from_this() { return shared_ptr<_Ty>(_Wptr); } };</pre></div>
作用:	enable_shared_from_this 是一个辅助类模板，用于使得一个类的对象能够从 shared_ptr 获取指向自己的 shared_ptr 实例。 <u>通常，只有对象是由一个 shared_ptr 管理时，你才可以对其使用 enable_shared_from_this。</u>
思考:	<u>你会想，我们明明可以使用 this，而 this 指针也能获得指向该对象的指针，为什么需要使用 shared from this 来获取呢？</u> 答案是： 假设你的对象由智能指针管理（特指 shared_ptr，因为共享指针 shared_ptr 会自动计数，如果该对象被引用时，引用计数会自动 +1），当我们想要使用对象时，如果不小心在之前将该智能指针类型的对象引用清零（即没有代码调用/引用该对象时，智能指针中的引用计数会递减，直到归零，如果系统检测到计数为0，则会结束该智能指针的生命周期），那么在之后的使用中，这个指针类型的对象便是无效的。 为了避免这种情况发生，我们在调用之前提前使用 auto self = shared_from_this(); 这段代码，不仅仅获取指针（该指针存储的内容和 this 指针完全相同），同时还为智能指针添加一个引用计数，这确保我们在使用对象指针的过程中，操作始终有效。
详细描述:	1.类模板定义: enable_shared_from_this 是一个模板类，接受一个类型参数 T，通常是你想要使其能够获取 shared_ptr 的类。 例如，假设有一个类 MyClass，你希望它能从 shared_ptr<MyClass> 中获取指向它自己的 shared_ptr，则 MyClass 类可以继承自 enable_shared_from_this<MyClass>。 2.实现方式: enable_shared_from_this 提供了一个成员函数 shared_from_this()，允许从当前对象中获取一个指向自己的 shared_ptr。 <u>3.但是，shared_from_this() 只能在对象已经由一个 shared_ptr 管理时调用，否则会抛出异常 (std::bad_weak_ptr)。</u>
示例:	<pre>#include <iostream> #include <memory> class MyClass : public std::enable_shared_from_this<MyClass> { public: void print() {</pre>

```
        std::cout << "Hello, I am MyClass!" << std::endl;
    }

    void example() {
        // 获取指向当前对象的shared_ptr
        std::shared_ptr<MyClass> ptr = shared_from_this();
        ptr->print(); // 使用shared_ptr调用成员函数
    }
};

int main() {
    std::shared_ptr<MyClass> ptr = std::make_shared<MyClass>();
    ptr->example(); // 正常调用 (ptr 会调用 MyClass的成员函数, 该成员函数使用了 shared_from_this() 实现其详细操作)
    return 0;
}
```

》》》在 lambda 表达式中, 调用一个成员函数需要通过指针 -> 来调用, 而其他的成员函数调用该成员函数时, 直接使用就好, 不需要这样操作。为什么?

在成员函数中, this 是自动传递的指针 (这里的 this 是隐式的, 指向当前对象), 编译器会自动知道你是在调用当前对象的方法 (即 SigRegModeFinish), 因此不需要额外显式使用 -> 来访问其他成员函数。

而 lambda 表达式需要显式传入对象指针 (this 或者所谓的 self), this 是类的实例指针, 使用 -> 操作符才能访问其成员函数。

```
void HttpMgr::SlotHttpFinish(QString res, ReqID reqID, Module mod, ErrorCode errcode)
{
    if(mod == Module::REGISTER_MOD)
        emit SigRegModeFinish(res, reqID, errcode);
}

QObject::connect(reply, &QNetworkReply::finished, [reply, reqID, mod, self]()
{
    // 处理错误情况, 并通过 SigHttpFinish 发送 error code
    if(reply->error() != QNetworkReply::NoError)
    {
        qDebug() << reply->errorString();

        emit self->SigHttpFinish("", reqID, mod, ErrorCode::ERR_NETWORK);
        reply->deleteLater(); // 为程序的安全性, 不立即删除。
    }

    return;
}
```

》》》QT 中的 explicit 关键字

在 Qt 中, explicit 关键字与 C++ 中的作用是一样的: 用于修饰构造函数, 防止构造函数被用作隐式类型转换的目标。

```
class MyClass {
public:
    explicit MyClass(int value) {
        // 构造函数
    }
};

void someFunction(MyClass obj) {
    // 一些代码
}

int main() {
    someFunction(10); // 编译错误, 因为构造函数是 explicit, 禁止隐式转换
    return 0;
}
```

》》》toUtf8() 是什么函数, 有什么作用, 进行什么操作?

释义: toUtf8 是 Qt 中 QString 类的一个成员函数, 它将 QString 对象中的文本转换为 UTF-8 编码格式的字节数组 (QByteArray)。

函数定义

```
#if !defined(Q_CLANG_QDOC)
[[nodiscard]] QByteArray toLatin1() const &
{ return toLatin1_helper(*this); }
[[nodiscard]] QByteArray toLatin1() &&
{ return toLatin1_helper_inplace(*this); }
[[nodiscard]] QByteArray toUtf8() const &
{ return toUtf8_helper(*this); }
[[nodiscard]] QByteArray toUtf8() &&
{ return toUtf8_helper(*this); }
[[nodiscard]] QByteArray toLocal8Bit() const &
{ return toLocal8Bit_helper(isNull() ? nullptr : constData(), size()); }
[[nodiscard]] QByteArray toLocal8Bit() &&
{ return toLocal8Bit_helper(isNull() ? nullptr : constData(), size()); }
#else
```

》》》什么是网络编程中的穿透？什么是回包？

穿透（NAT 穿透）	穿透指的是在网络通信中，如何使得位于不同网络（尤其是受限网络，如 NAT 或防火墙后）的两台设备能够互相通信的过程。由于 NAT 会修改网络数据包中的源 IP 地址和端口，导致设备与外部世界的直接连接被阻断，所以在这种情况下需要一种方式来“穿透” NAT，使得位于局域网内的设备能够与外部设备建立连接。
常见的 NAT 穿透方法有：	1.STUN (Session Traversal Utilities for NAT): 通过让客户端向公共服务器发送请求，并利用服务器响应中的信息来进行 NAT 穿透。 2.TURN (Traversal Using Relays around NAT): 当 STUN 无法工作时，可以通过中继服务器转发通信数据，从而实现穿透。 3.UPnP (Universal Plug and Play): 设备可以通过 UPnP 协议请求路由器进行端口映射，从而实现穿透。

穿透技术的核心目标是让被 NAT 或防火墙隔离的设备之间能够进行直接的通信。

回包（Response Packet）

回包	回包指的是在网络通信中，一个设备收到请求之后，按照请求的内容或者协议的要求，向请求设备发送的响应数据包。回包通常包含请求的结果、确认信息、数据等。
在客户端与服务器的通信中，客户端发送请求包，服务器处理请求并通过回包返回结果。例如：	• 在 HTTP 请求中，客户端发送请求包，服务器处理后通过回包返回网页数据。 • 在 TCP 连接建立时，客户端发起连接请求，服务器响应回包以确认连接。

回包的内容会根据请求的不同而变化，它是通信过程中重要的一部分，通常带有状态信息、数据或错误信息。

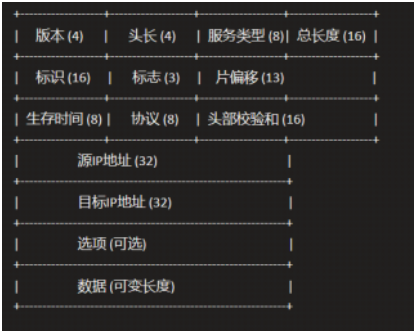

》》》HTTP，UDP，TCP等等协议的区别？包括应用层面和形式上的不同：包括协议的格式等等，这些格式的设置导致了什么，有什么好处？

1. 协议层级与功能定位

根据 TCP/IP 四层模型，这些协议分布在不同的层级，承担不同的职责：

层级	协议	核心功能
应用层	HTTP	定义应用程序间的数据交互规则（如网页请求/响应格式）。
传输层	TCP / UDP	提供端到端的数据传输服务（可靠性、流量控制、复用等）。
网络层	IP	实现主机到主机的数据包路由和寻址（基于IP地址）。
链路层	以太网 / Wi-Fi	负责物理介质上的数据传输（如MAC地址寻址、帧封装）。

2. 协议格式对比

(1) IP 协议（网络层）		(2) TCP 协议（传输层）	
格式：		格式：	
关键字段：	<ul style="list-style-type: none">• 协议字段：标识上层协议（如 6 表示 TCP，17 表示 UDP）。• 生存时间 (TTL)：防止数据包无限循环，每经过一个路由器减 1，归零则丢弃。	关键字段：	<ul style="list-style-type: none">• 序列号/确认号：实现可靠传输（通过确认机制和重传）。• 控制位：如 SYN（建立连接）、ACK（确认）、FIN（终止连接）。• 窗口大小：流量控制，避免发送方淹没接收方。

	<ul style="list-style-type: none">• 片偏移：支持大数据包分片传输，由接收端重组。		
设计优势：	<ul style="list-style-type: none">• 分片重组：允许大数据包适应不同网络的最大传输单元（MTU）。• 全局寻址：通过 IP 地址实现跨网络的端到端路由。	设计优势：	<ul style="list-style-type: none">• 可靠性：通过三次握手、数据确认、超时重传确保数据完整有序。• 拥塞控制：动态调整发送速率，避免网络拥塞（如慢启动、拥塞避免）。 <ul style="list-style-type: none">• 传输层协议：TCP 是传输层协议，用于确保在两个端点之间可靠的数据传输。• 面向连接：TCP 是面向连接的协议，通信前需要先建立连接（三次握手），通信结束后需要关闭连接（四次挥手）。• 可靠性：TCP 保证数据的可靠性，包括数据的顺序、完整性和无丢失传输。使用了流量控制、拥塞控制和重传机制。
(3) UDP 协议（传输层）		(4) HTTP 协议（应用层）	
格式：		格式（HTTP/1.1 请求示例）：	<pre>GET /index.html HTTP/1.1 Host: www.example.com User-Agent: Mozilla/5.0 Accept: text/html (请求体, GET 通常无内容)</pre>
关键字段：	<ul style="list-style-type: none">• 无序列号/确认号：不保证数据可靠到达。• 校验和可选：部分场景可关闭以提升性能。	关键特点：	<ul style="list-style-type: none">• 文本协议：人类可读，但 HTTP/2 后改为二进制帧以提高效率。• 无状态：每次请求独立，依赖 Cookie/Session 维持状态。• 方法语义：GET（获取资源）、POST（提交数据）、PUT（更新资源）等。
设计优势：	<ul style="list-style-type: none">• 低延迟：无连接、无握手，适合实时应用（如视频通话、游戏）。• 轻量级：头部开销小（仅 8 字节），传输效率高。	设计优势：	<ul style="list-style-type: none">• 灵活性：支持多种内容类型（JSON、HTML、图片等）。• 可扩展性：通过头部字段（如 Content-Type、Authorization）增强功能。
1	<ul style="list-style-type: none">• 传输层协议：UDP 是一个面向数据报的协议，属于传输层协议。• 无连接：UDP 是无连接的，不需要建立连接即可发送数据。• 不可靠：UDP 不保证数据包的送达、顺序和完整性，数据包可能丢失、乱序或重复。	1	<ul style="list-style-type: none">• 应用层协议：HTTP 属于应用层协议，是用于客户端（通常是浏览器）和服务端之间交换数据的协议。• 无连接：HTTP 协议是无连接的，客户端和服务端在请求响应过程中不保持连接，每次请求都需要建立新的连接。• 请求-响应模式：HTTP 通常采用请求-响应模式，客户端发送请求，服务端返回响应。• 面向文本的协议：HTTP 消息通常是基于文本的，包括请求头、请求体（可选），响应头、响应体（可选）。

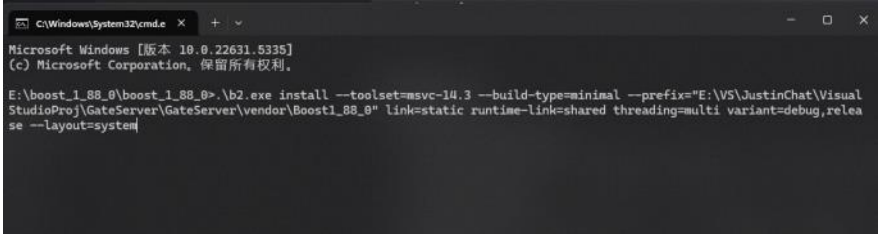
3. 应用场景对比

协议	典型应用场景	选择原因
HTTP	网页浏览（HTTPS）、REST API、文件下载	标准化、易调试、兼容性强。
TCP	文件传输（FTP）、电子邮件（SMTP）、远程登录（SSH）	需要可靠传输和有序交付。
UDP	实时音视频（Zoom、VoIP）、DNS查询、在线游戏（低延迟）	容忍少量丢包，追求传输速度。
IP	所有基于 IP 的网络通信（如 TCP/UDP/ICMP）	提供全局寻址和路由，是互联网

----- Ep 4 -----

》》》下载 boost 并创建控制台项目，相信大家没有什么问题。

.\b2.exe install --toolset=msvc-14.3 --build-type=minimal --prefix="E:\VS\JustinChat\VisualStudioProj\GateServer\GateServer\vendor\Boost1_88_0" link=static runtime-link=shared threading=multi variant=debug,release --layout=system



关键参数说明

1. --build-type=minimal

仅构建当前指定的变体（如 debug 和 release），避免生成多余的动态库。

2. --layout=system

将所有库文件直接输出到 lib 目录，不生成 static 或 shared 子目录。

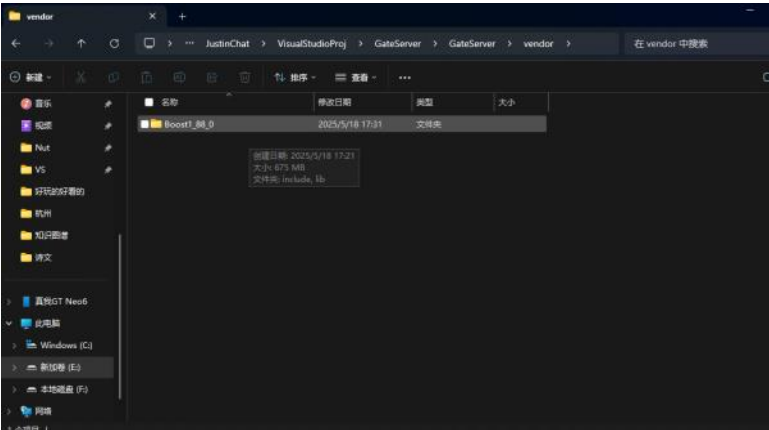
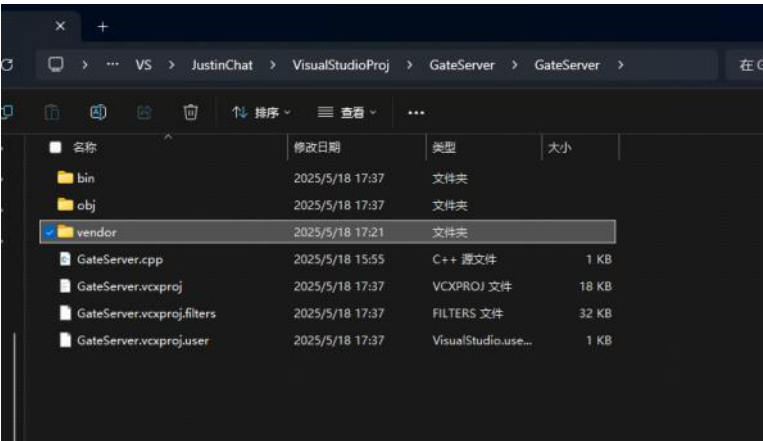
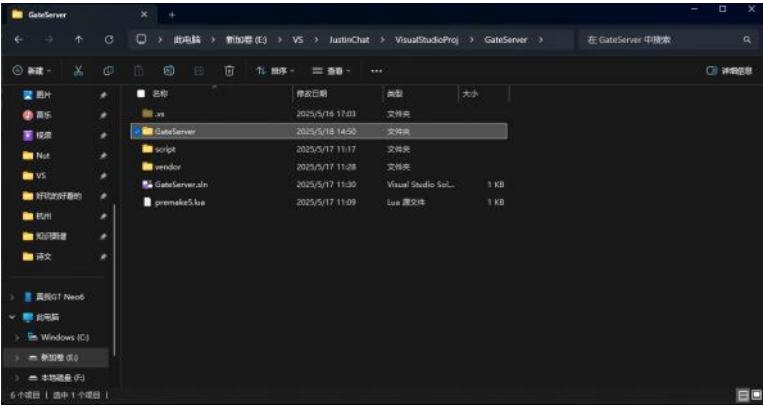
3. 显式指定变体

通过 variant=debug,release 明确构建调试和发布版本，无需依赖 --build-type=complete。

4. 避免冲突参数

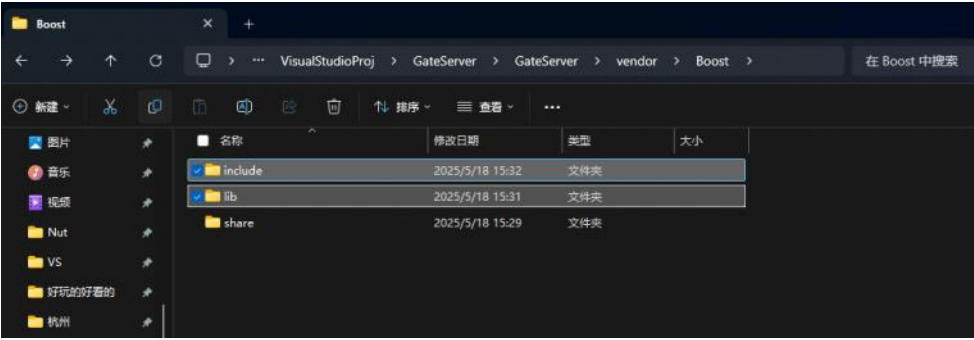
移除了 debug relese 冗余参数（已通过 variant=debug,release 覆盖）。

b2.exe 运行完毕后, include 和 lib 位于项目目录的 GateServer 路径下:



》》注意:

GateServer\vendor\Boost 中存放了自动生成的 include 和 lib (但不知为何, 也生成了 share, 这个文件可能导致错误, 我将其删除了)



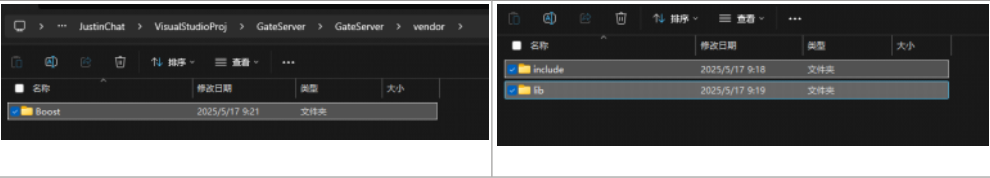
但我精简了路径，删除了一个多余的文件夹，因而 include 之后便是 boost 这个目录，boost 中存放 accumulators、algorithm 等等文件夹

```
E:\VS\JustinChat\VisualStudioProj\GateServer\GateServer\vendor\Boost\include\boost
```

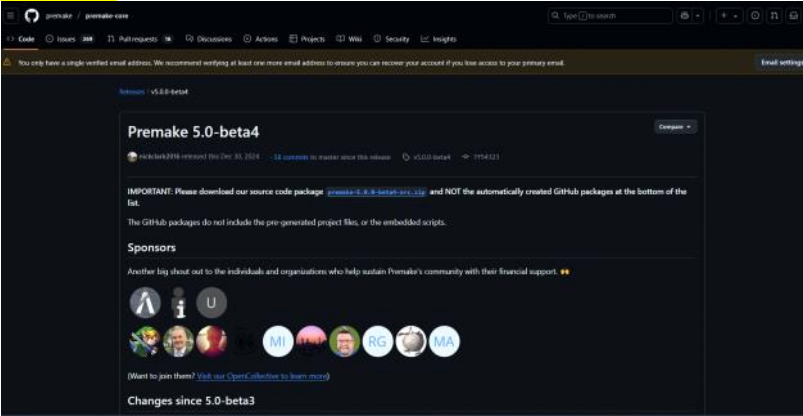
》》》但是我不想使用 VS 属性管理器，我要玩一点有趣的东西，我想使用 premake 脚本完成这一操作。

》》000000000000我们在 GateServer\GateServer 下创建一个 vendor。这里的 vendor 表示存储项目的依赖项。

在这个 vendor 中，我们存放一下 boost 的文件（创建一个 Boost 文件夹，存放刚才下载的 ????? 和 lib 目录 <---- 以 1.88.0 版本为例）



》》11111111在 github 上下载 premake （premake 从 bete-1 开始支持 VS 2022 编辑器）



翻到下面选择 windows 系统支持文件：



》》22222222解压 premake.zip 文件（在项目根目录下创建 vendor 文件夹，并在 vendor 中创建 premake\bin，然后放置解压好的文件）

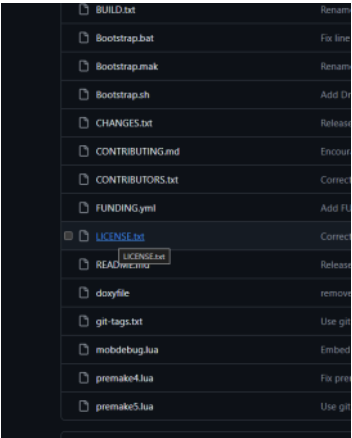
根目录下创建的 vendor 一般用于存放一些外部文件。

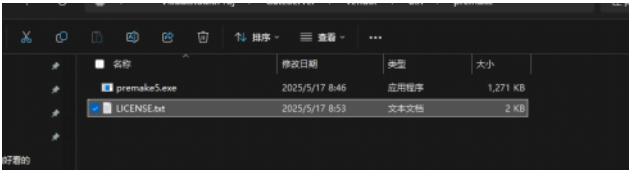
名称	修改日期	类型	大小
.vs	2025/5/16 17:03	文件夹	
GateServer	2025/5/16 17:17	文件夹	
vendor	2025/5/17 8:47	文件夹	
GateServer.sln	2025/5/16 17:03	Visual Studio Sol...	2 KB

```
E:\VS\JustinChat\VisualStudioProj\GateServer\vendor\premake\bin
```

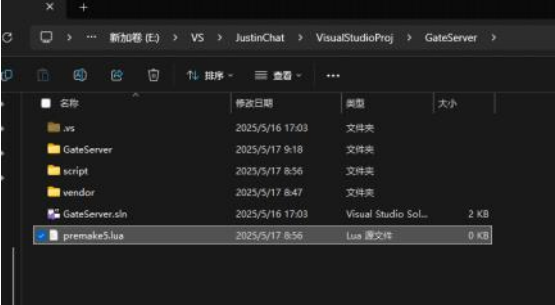
名称	修改日期	类型	大小
premake5.exe	2025/5/17 8:46	应用程序	1,271 KB


不要忘了放置 premake 的许可协议文件：



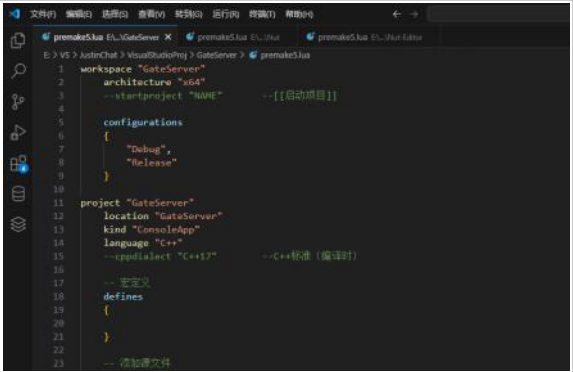


》》333333333 创建 premake5.lua 脚本文件（同样在根目录下）





进行编写：



》》444444444 编写 premake5.lua 文件

首先 .lua 是一种变成语言，不过不需要额外学习，因为在使用 premake 的过程中我们可以大概领会其语法。
对于 preamake 文件的编写规则，可以大致看一些。在使用中慢慢学习。[参考文档]（<https://premake.github.io/docs/>）

一个基本的 premake 文件示例：（对于 GateServer 这个项目的 premake，具体参考我文件中的代码）

```
workspace "MyProject"
    configurations { "Debug", "Release" }

project "MyApp"
    kind "ConsoleApp"
    language "C++"

    -- 添加包含目录
    includedirs { "path/to/include" }

    -- 添加库目录
    libdirs { "path/to/lib" }

    -- 添加源文件
    files { "src/**/*.cpp" }

    -- 在 Debug 和 Release 配置下链接库
    links { "mylib" }

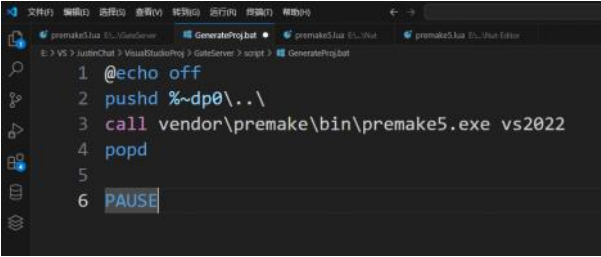
    -- 选择编译配置
    filter "configurations:Debug"
        symbols "On"

    filter "configurations:Release"
        optimize "On"
```

》》555555555555 编写完成之后，创建批处理文件（.bat 文件）

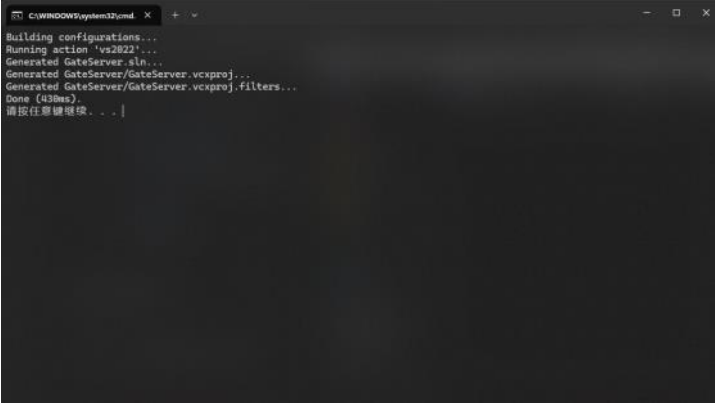
在刚刚创建过的 Script 文件夹中，我们存放一个 .bat 文件。在此之前有几点我要声明一下：

1	Premake5.lua 文件可以在 cmd 中直接通过指定命令运行。 创建的 .bat 文件的作用是：在 .bat 文件中保存自定义的功能，然后通过双击运行 .bat 文件，实现在 cmd 中运行命令的效果。
2	Premake5.lua 文件可以放在项目中的任何地方，但是文件路径会受影响，.bat 代码要按需更改。
3	.bat 文件也可以放在任何地方，我们只需要在指令中调整命令被调用的位置即可。（我选择放在 script 的文件夹中，主要是方便管理）



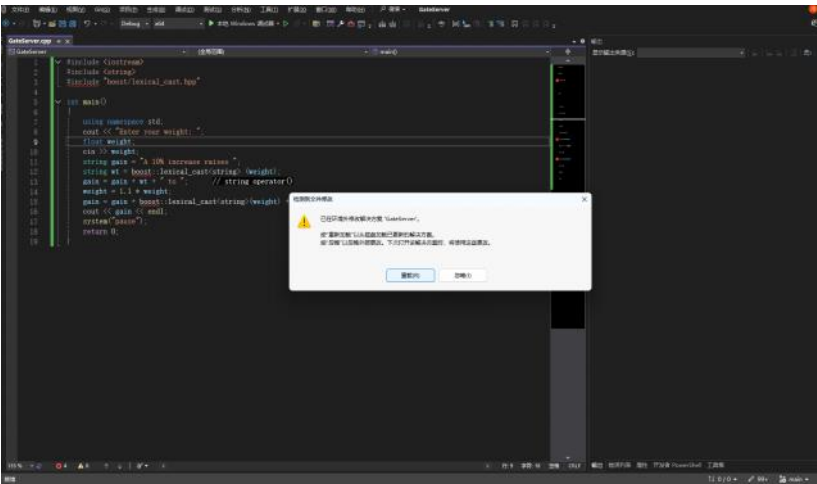
具体功能自行查阅。

》》6666666666 双击运行 .bat 文件

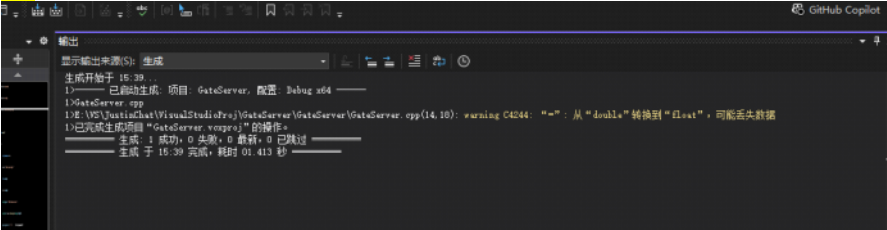


表明运行成功。

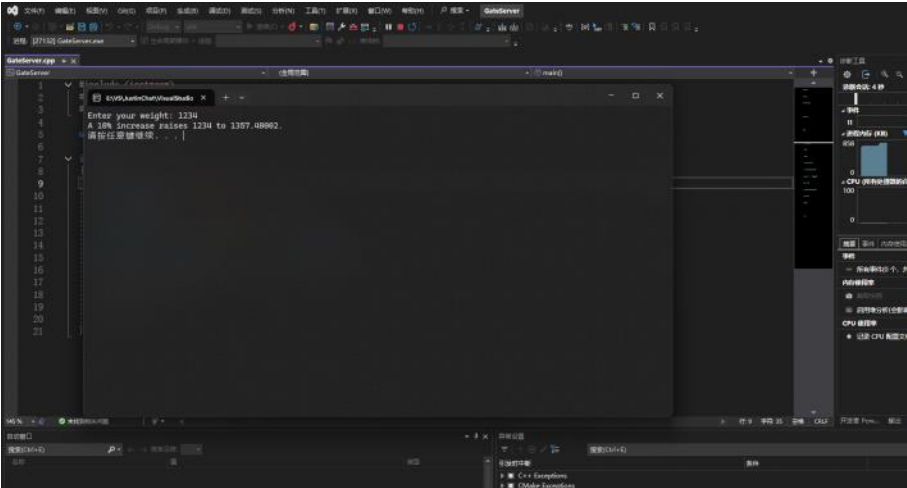
我们在 VS 中重载一下项目：



Ctrl + F7 可以看到，成功编译

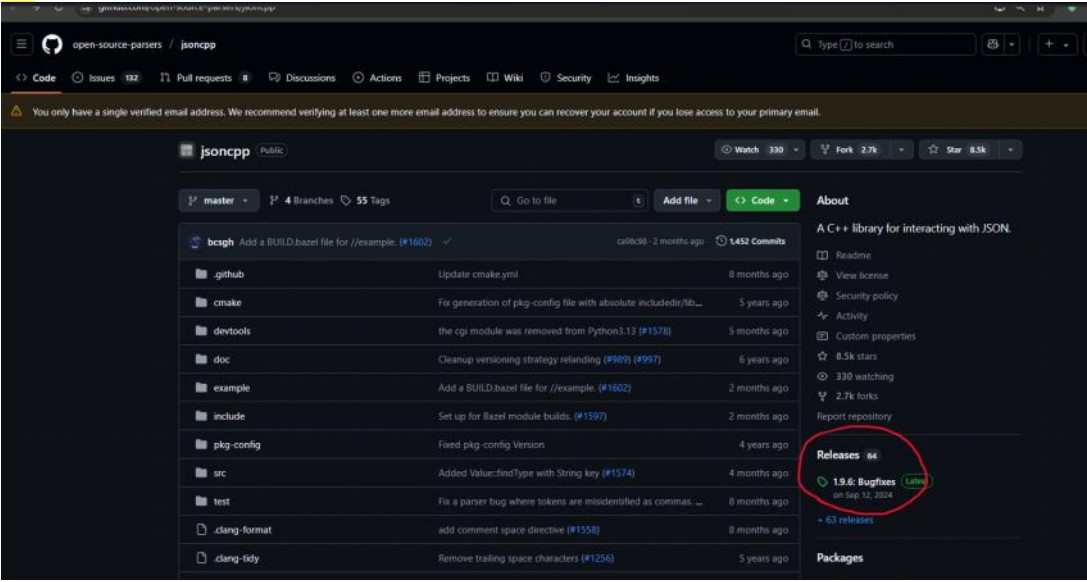


F5 运行



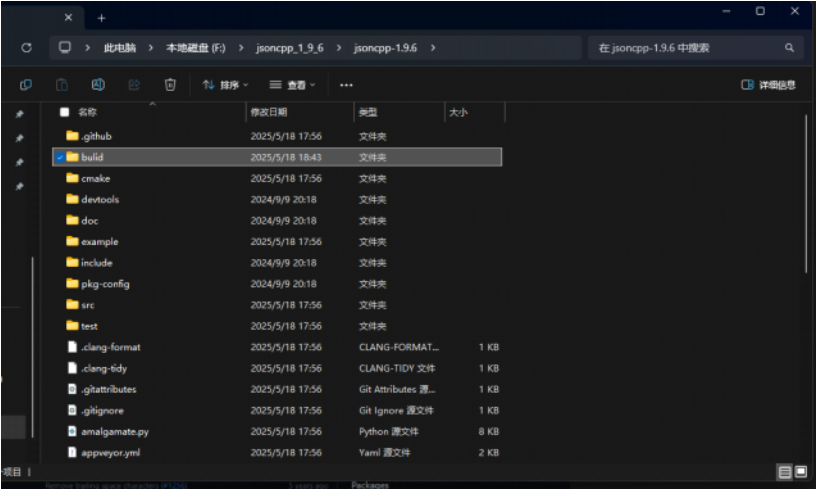
jsoncpp jsoncpp jsoncpp jsoncpp jsoncpp

下载 jsoncpp 压缩包



》》下载完成后，解压。我们可以得到一些文件。

为了避免误操作其他文件，我们在其中创建一个文件夹 bulid，用于构建。



》》使用命令进行处理

在 bulid 目录下，我们进入命令行，并运行：（以红色明文标注的命令为准，如果你准备在此之后选用生成的 .sln 进行操作，则可以运行第一个指令）

1111111

```
cmake .. -G "Visual Studio 17 2022" -A x64 -DCMAKE_INSTALL_PREFIX=../x64 -DBUILD_SHARED_LIBS=OFF -DJSONCPP_WITH_TESTS=OFF
```

生成支持 VS2022 多配置 (Release/Debug) 的解决方案：

```
bash
```

```
cmake .. ^
-G "Visual Studio 17 2022" -A x64 ^
-DCMAKE_INSTALL_PREFIX=../x64 ^
-DBUILD_SHARED_LIBS=OFF ^
-DJSONCPP_WITH_TESTS=OFF
```

o 关键参数说明：

- G "Visual Studio 17 2022" -A x64：指定生成 VS2022 的 64 位解决方案。
- DCMAKE_INSTALL_PREFIX=../x64：设置安装根目录为 build/x64。
- DBUILD_SHARED_LIBS=OFF：生成静态库（.lib）。
- DJSONCPP_WITH_TESTS=OFF：禁用测试用例以加快编译速度。

不需要在命令中额外指定Release 版本（使用 /MD） / Debug 版本（使用 /MDd）

因为 JSONCPP 的 CMake 脚本已内置对 MSVC 运行时库的支持，默认行为如下：
Release 模式：生成使用 /MD 的静态库（文件名 jsoncpp.lib）。
Debug 模式：生成使用 /MDd 的静态库（文件名 jsoncppd.lib，后缀 d 表示 Debug）。

222222222222

以上命令在最终处理完毕之后，会得到两个命名相同的 .lib 文件，但这可能会导致 Debug 和 release 生成的两种 .lib 因命名重复而覆盖。
如果想要提前区别这两种 .lib，可以在命令中特别声明。

命令如下：
cmake .. -G "Visual Studio 17 2022" -A x64 -DCMAKE_INSTALL_PREFIX=./x64 -DBUILD_SHARED_LIBS=OFF -DJSONCPP_WITH_TESTS=OFF -DCMAKE_DEBUG_POSTFIX="d"

若需强制指定运行时库类型（如确保使用 /MD 或 /MT），可在 CMake 命令中通过 CMAKE_CXX_FLAGS 参数控制：

```
bash
cmake .. ^
-G "Visual Studio 17 2022" -A x64 ^
-DCMAKE_INSTALL_PREFIX=./x64 ^
-DBUILD_SHARED_LIBS=OFF ^
-DJSONCPP_WITH_TESTS=OFF ^
-DCMAKE_CXX_FLAGS_RELEASE="/MD" ^
-DCMAKE_CXX_FLAGS_DEBUG="/MDd"
```

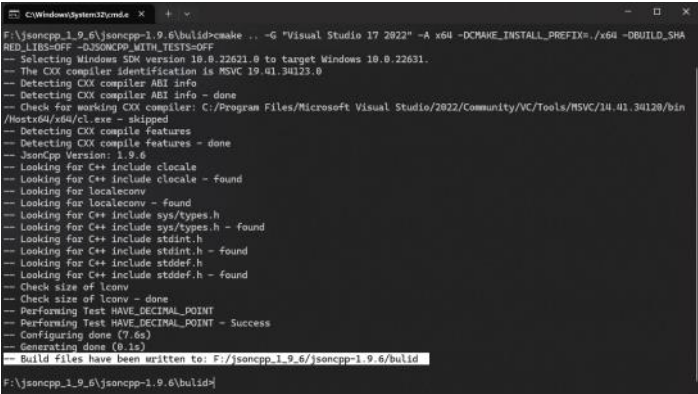
运行 CMake 时显式指定 按配置分离输出路径：

```
bash
cmake .. ^
-G "Visual Studio 17 2022" -A x64 ^
-DCMAKE_INSTALL_PREFIX=./x64 ^
-DBUILD_SHARED_LIBS=OFF ^
-DJSONCPP_WITH_TESTS=OFF ^
-DCMAKE_DEBUG_POSTFIX="d" # 强制 Debug 库添加 "d" 后缀
```

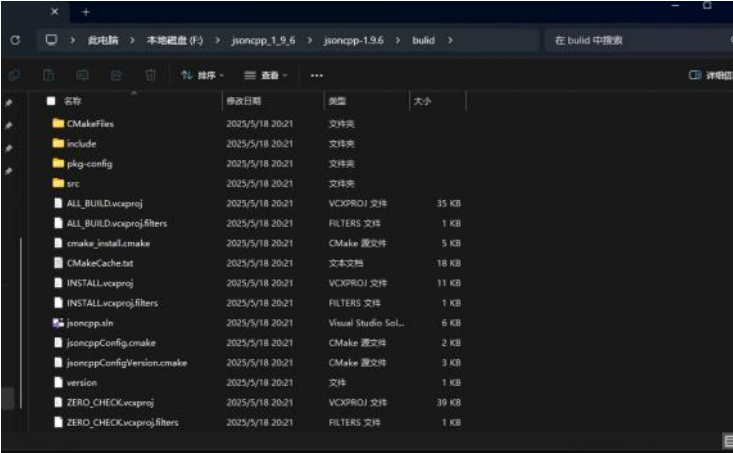
结果将会是：

<input checked="" type="checkbox"/>	jsoncpp.lib	2025/5/18 21:52	VisualStudio.lib....	1,167 KB
<input checked="" type="checkbox"/>	jsoncppd.lib	2025/5/18 21:53	VisualStudio.lib....	4,340 KB

处理后得到以下结果：



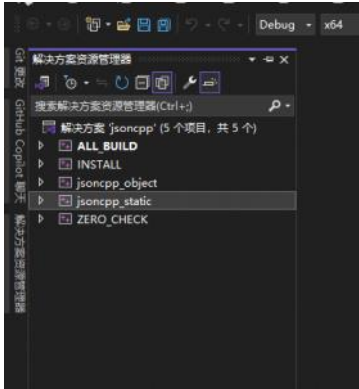
在 build 目录下，我们得到一些通过脚本自动生成的文件：



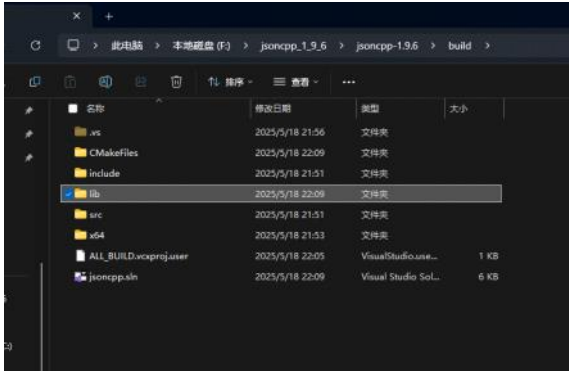
在这里，你可以使用 VS 运行 .sln，并得到对应文件。但我想继续使用 cmd，并通过命令完成这些任务。

顺便一提，如果你选择打开 vs 处理，这里的 jsoncpp_static 等同于视频中的 lib_json

(之后的步骤和视频中相同：先设置属性->Release 版本（使用 /MD） / Debug 版本（使用 /MDd），然后生成）

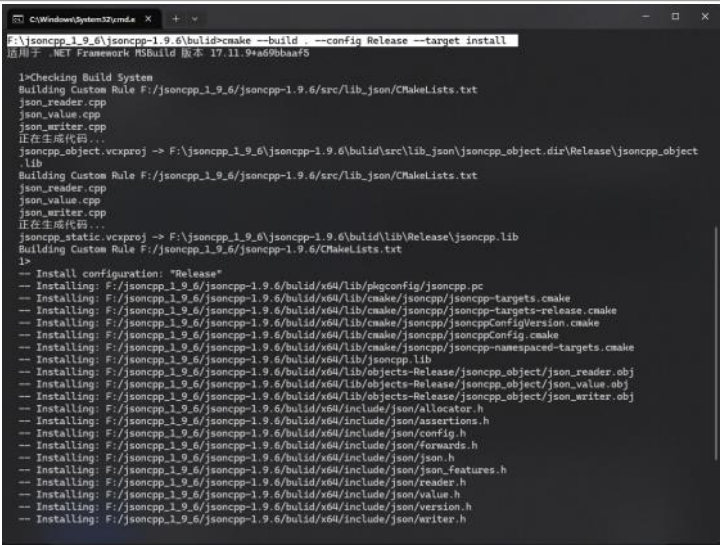


完成操作后，不是在 x64 文件夹中寻找 .lib 文件，而是在同层级的 lib 文件夹中寻找生成的两个 .lib 文件。



使用命令生成 .lib 文件。在命令行中输入以下两条指令：

Release 版本（使用 /MD）
cmake --build . --config Release --target install



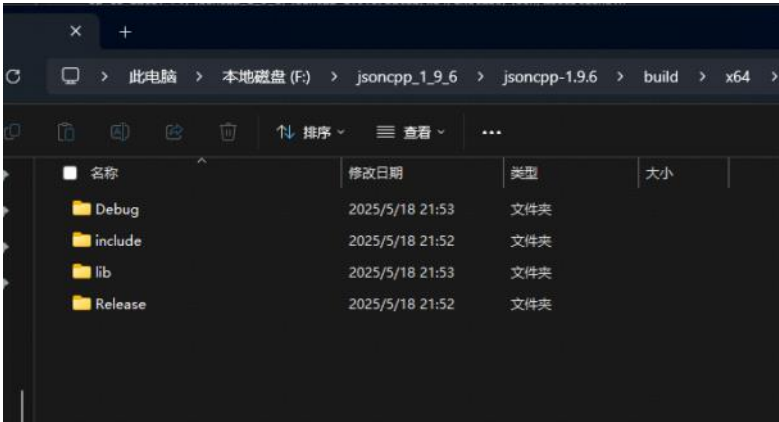
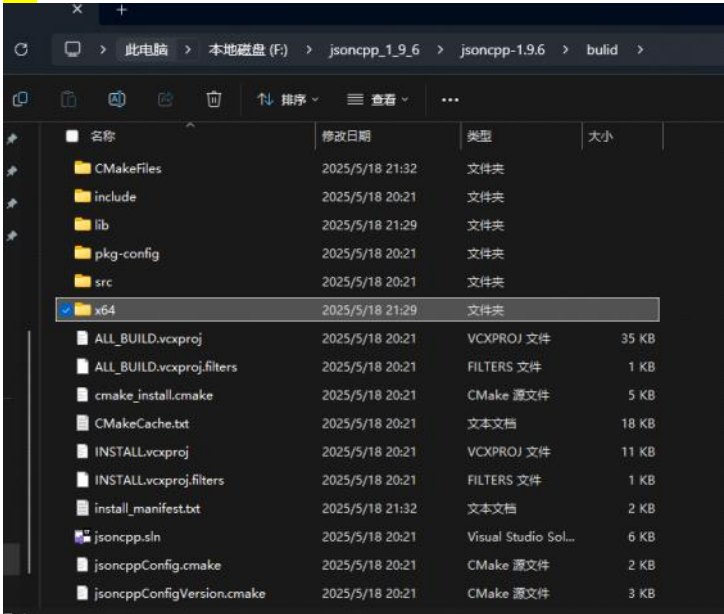
Debug 版本 (使用 /MDd)
cmake --build . --config Debug --target install

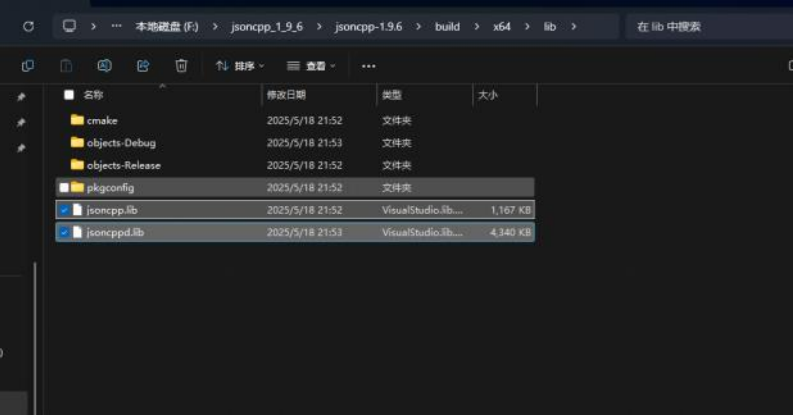
```

C:\Windows\system32\cmd.exe
-- Installing: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/writer.h
F:\jsoncpp_1.9.6\jsoncpp-1.9.6\build>cmake --build . --config Debug --target install
适用于 .NET Framework 的 MSBuild 版本 17.11.9*ad9bbaaf5

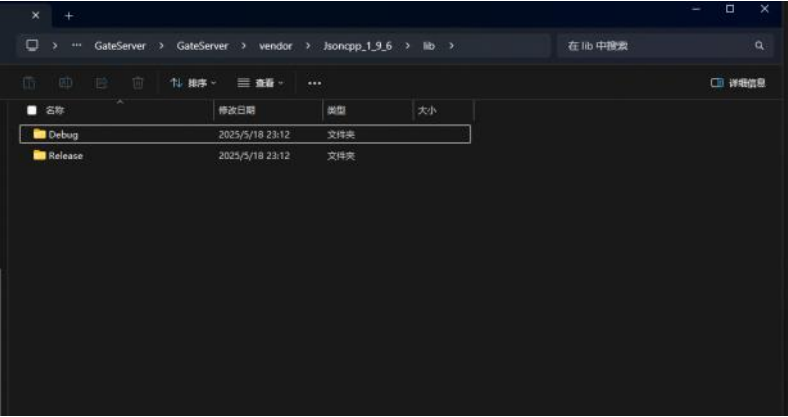
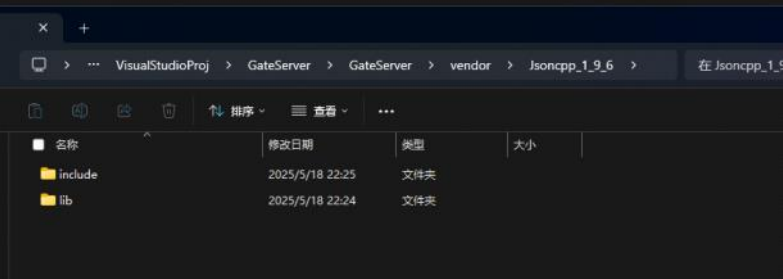
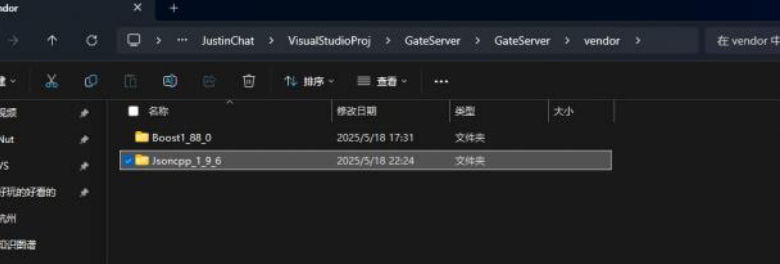
1>Checking Build System
Building Custom Rule F:/jsoncpp_1.9.6/jsoncpp-1.9.6/src/lib_json/CMakeLists.txt
json_reader.cpp
json_value.cpp
json_writer.cpp
正在生成代码...
jsoncpp_object.vcxproj -> F:\jsoncpp_1.9.6\jsoncpp-1.9.6\build\src\lib_json\jsoncpp_object.dir\Debug\jsoncpp_object.1
ib
Building Custom Rule F:/jsoncpp_1.9.6/jsoncpp-1.9.6/src/lib_json/CMakeLists.txt
json_reader.cpp
json_value.cpp
json_writer.cpp
正在生成代码...
jsoncpp_static.vcxproj -> F:\jsoncpp_1.9.6\jsoncpp-1.9.6\build\lib\Debug\jsoncpp.Lib
Building Custom Rule F:/jsoncpp_1.9.6/jsoncpp-1.9.6/CMakeLists.txt
1>
-- Install configuration: "Debug"
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/pkgconfig/jsoncpp.pc
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/cmake/jsoncpp/jsoncpp-targets.cmake
-- Installing: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/cmake/jsoncpp/jsoncpp-targets-debug.cmake
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/cmake/jsoncpp/jsoncppConfigVersion.cmake
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/cmake/jsoncpp/jsoncppConfig.cmake
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/cmake/jsoncpp/jsoncpp-namespaced-targets.cmake
-- Installing: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/jsoncpp.Lib
-- Installing: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/objects-Debug/jsoncpp_object/json_reader.obj
-- Installing: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/objects-Debug/jsoncpp_object/json_value.obj
-- Installing: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/lib/objects-Debug/jsoncpp_object/json_writer.obj
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/allocator.h
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/assertions.h
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/config.h
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/forwards.h
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/json.h
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/json_features.h
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/reader.h
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/value.h
-- Up-to-date: F:/jsoncpp_1.9.6/jsoncpp-1.9.6/build/x64/include/json/version.h
```

lib 文件被生成。我们看到在 build 中，新生成了一个文件夹 :x64。





》》将这些生成好的 .lib 放在项目的 vendor 文件夹中。（include 在生成 .lib 文件之前，我们已经放置过了，详情查阅视频）
打开 vendor 我们创建 Jsoncpp_1_9_6，然后在其中的include 和 lib 文件夹中存放对应文件。（lib文件夹中需要创建两个文件夹 debug 和 release）

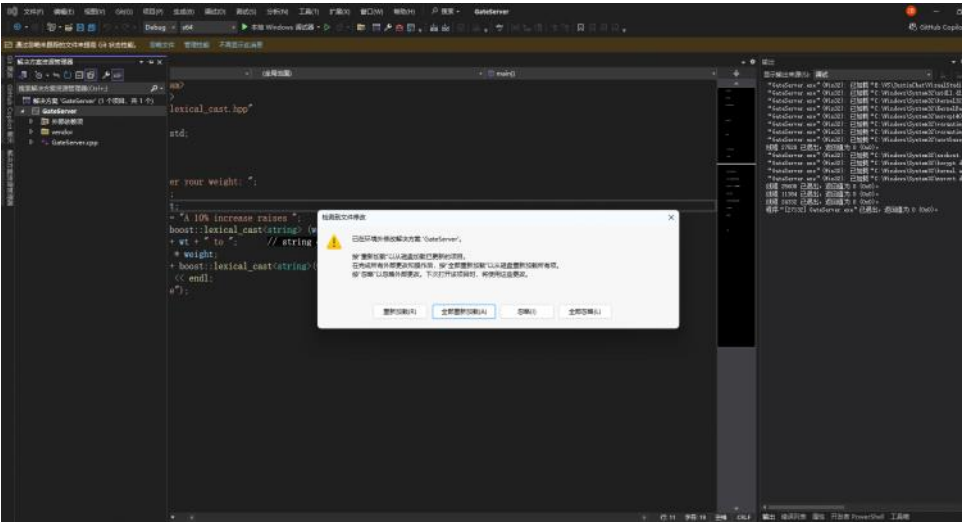


》》一个疑问：关于处理脚本时，额外生成的 include，是否需要包含？
jsoncpp的头文件（位于include/json目录下，不需要手动生成，解压后便存在）通常是平台无关的，不依赖于具体的编译配置。
无论是Release还是Debug版本，头文件的内容都是一样的。因此，在编译过程中，头文件不需要被重新生成或修改，只需要将它们复制到指定的安装目录中即可。

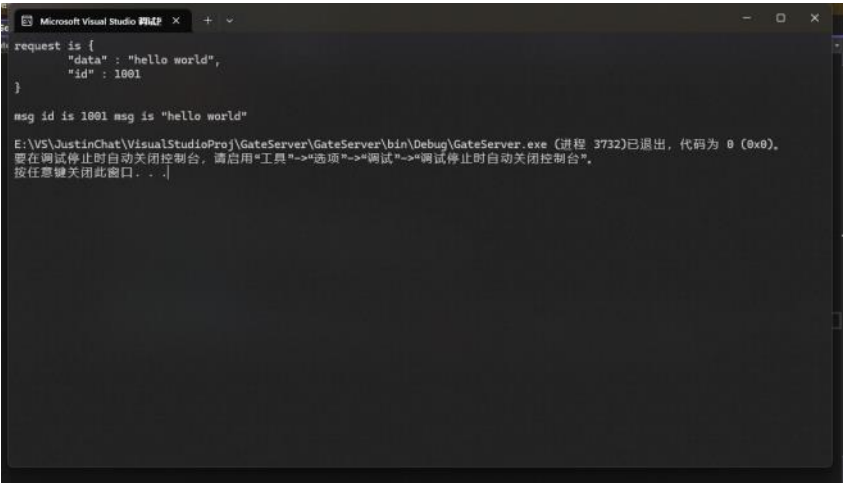
》》处理完所有文件之后，我们更新一下 premake 文件

```
32
33 -- 添加包含目录
34 includedirs
35 {
36     "%(prj.name)/vendor/Boost1_88_0/include",
37     "%(prj.name)/vendor/Jsoncpp_1_9_6/include"
38 }
39
40 -- 添加库目录
41 libdirs
42 {
43     "%(prj.name)/vendor/Boost1_88_0/lib"
44 }
45
46 -- 在 Debug 和 Release 配置同样的链接库
47 links
48 {
49 }
50
51 -- 对于 jsoncpp 需要动态设置库目录和链接库 (因为二者有 debug 和 release 不同环境下的运行库)
52 filter "configurations:Debug"
53     libdirs
54     {
55         "%(prj.name)/vendor/Jsoncpp_1_9_6/lib/Debug" -- Debug 库路径
56     }
57     links { "jsoncppd.lib" } -- Debug 库
58
59 filter "configurations:Release"
60     libdirs
61     {
62         "%(prj.name)/vendor/Jsoncpp_1_9_6/lib/Release" -- Release 库路径
63     }
64     links { "jsoncpp.lib" } -- Release 库
65
```

然后重新运行 .bat 文件。重新载入 GateServer。



F5成功运行:



》》》》为什么有的库只需要包含即可，有的库却需要包含并且链接？
这涉及到静态库和动态库的区别。

静态库 (.lib / .a)

特点:	在编译时被完整嵌入到可执行文件中。
链接行为:	必须显式链接到最终的可执行文件或动态库中。

如果未链接，会导致 未定义的符号错误（如 undefined reference）。

动态库（.dll / .so）

特点：	在运行时由操作系统动态加载。
链接行为：	<ul style="list-style-type: none">• 在编译时只需指定动态库的 导入库（如 Windows 的 .lib 或 Linux 的 .so），无需嵌入代码。• 如果未链接导入库，同样会导致编译错误。• 运行时需要确保动态库的路径在系统搜索路径中（如 Windows 的 PATH 或 Linux 的 LD_LIBRARY_PATH）。

》》》premake 脚本中的 files 一般对应项目源码文件，不会跟踪外部库的文件（比如 boost 的 include 文件 .hpp），但是有一种情况需要除外。

只有以下场景需要将第三方文件添加到 files 块中：

直接修改第三方库的源码：	如果你需要定制或调试第三方库的代码。
使用单文件库（如 STB 库）：	部分库（如 stb_image.h）需要在一个 .cpp 文件中显式包含实现。

```
-- 示例：单文件库需要显式包含实现
files {
    "vendor/stb/stb_image.h",
    "vendor/stb/stb_image.cpp" -- 包含一次实现
}
```