000参考000

2025年5月12日 15:28

---------- Ep 0 前期准备 ----------

包括快捷键参考，QT安装，Cmake 教程，QT 中的一些外观 etc.

》》》》QT 的安装:

[参考视频](https://www.bilibili.com/video/BV1uV4y1X72Q/?share_source=copy_web&vd_source=ca2feff7d155a2579964dfa2c3173769)

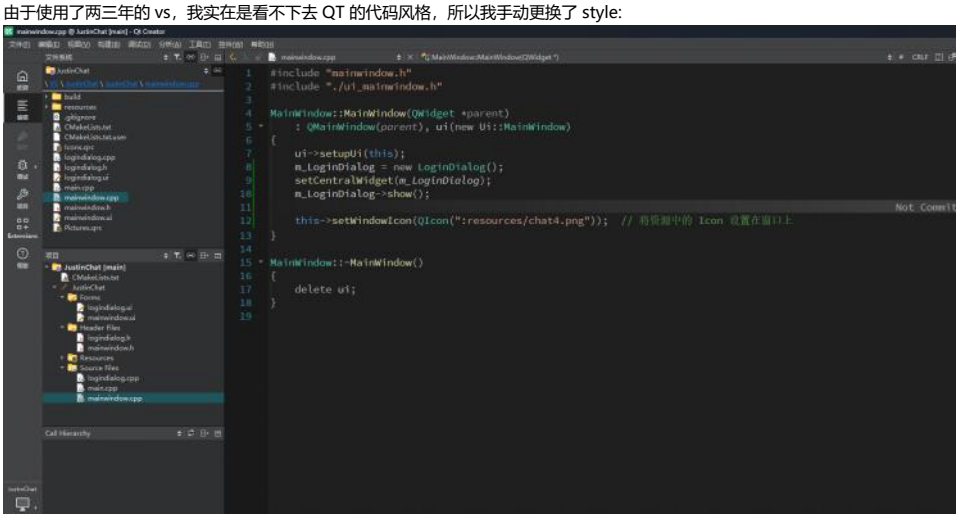
》》》》QT 编辑器的快捷键（长期更新）

Ctrl + R	运行项目
F2	快捷跳转至光标选中对象的声明或定义
Ctrl + B	构建项目（包括编译和链接过程）
Ctrl + Shift + B	清理项目
Ctrl + Space	智能感知（但与 Windows 系统冲突，需要改建）

----- 一些插件以及快捷键更改、日志库的引用、cmake的一些操作明细、Premake 脚本引入、宏的设置 ----------

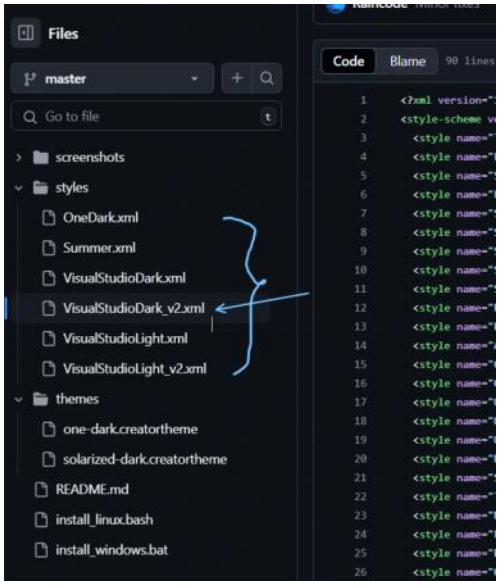
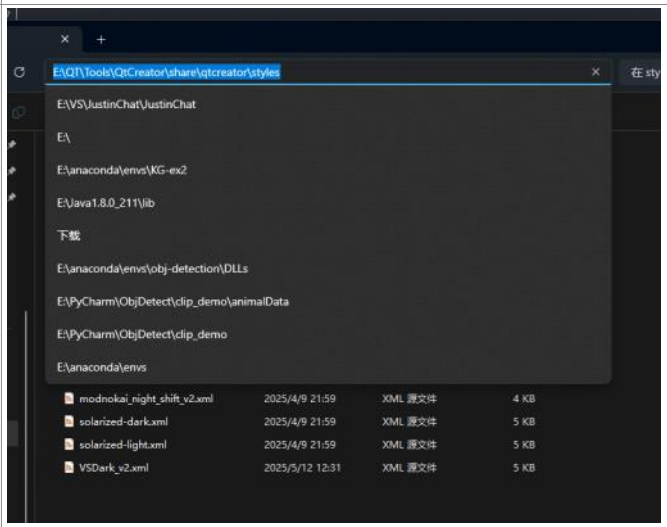
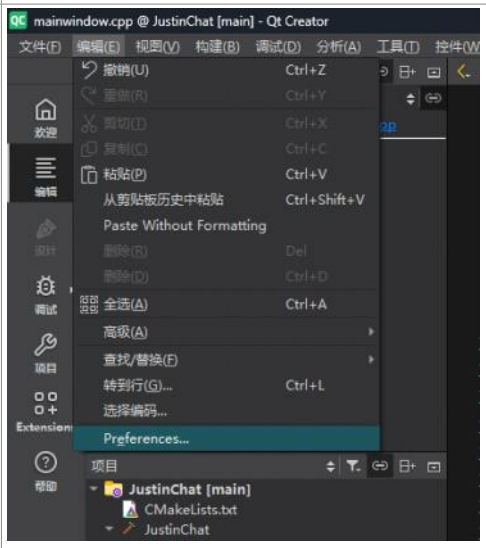
》》》》如何将 QT Creator 中的主题更换为 Visual Studio 风格的?

由于使用了两三年的 vs，我实在是看不下去 QT 的代码风格，所以我手动更换了 style:

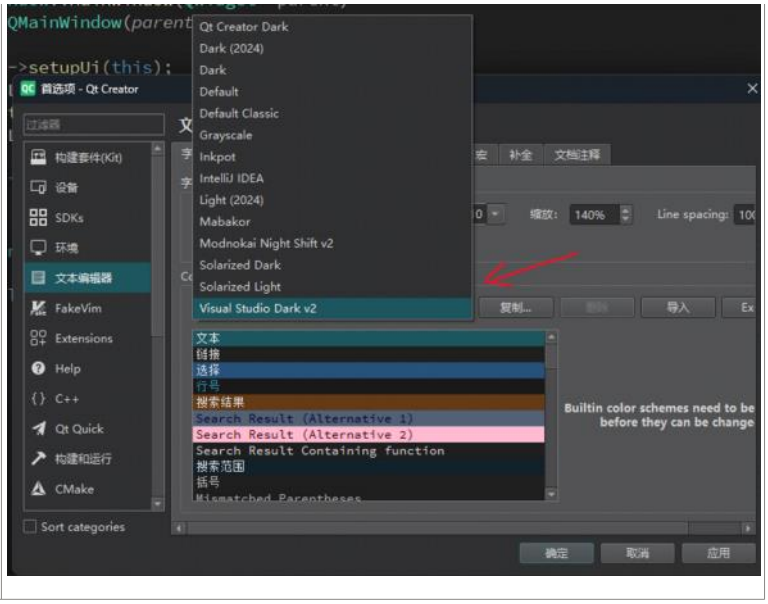


步骤:

首先，这里有一个仓库，存放了 Vs Dark 风格的 .xml 文件，（ https://github.com/Raincode/QtCreator-Color-Schemes ）	

在styles路径下寻找一个自己喜欢的风格	
将下载好的 .xml 文件放置在这个路径之下：QT下载目录\Tools\QTCreator\share\qtcreator\styles	
放置完成之后打开 preferences:	

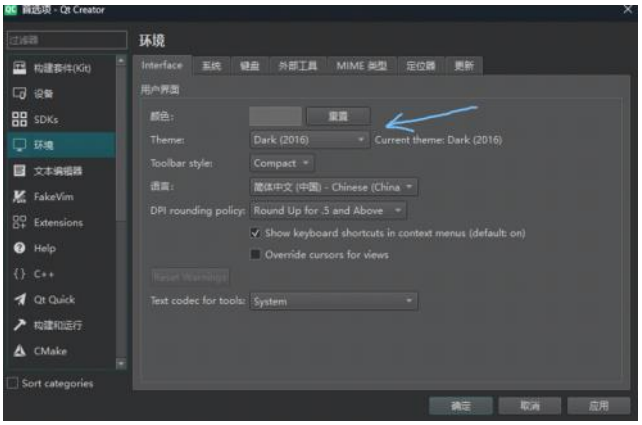
选择文本编辑器，然后选择你导入的风格：



》》如果你需要更改主题：Themes

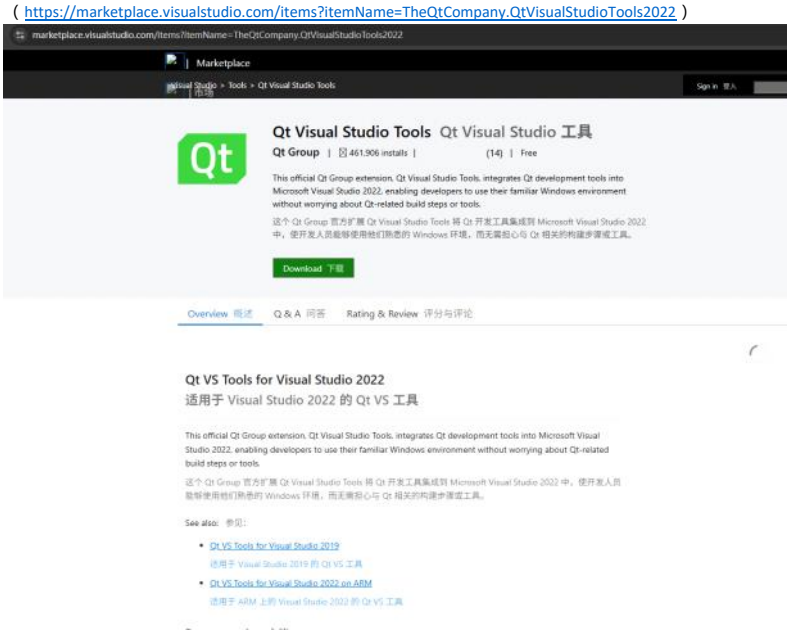
（和上述方法一致，将 themes 文件夹下的文件放置在：QT下载目录\Tools\QTCreator\share\qtcreator\themes 注意是：**themes !!!** ）

下载好文件之后，于“环境”中选择刚刚下载的主题即可。



》》》》在 VS 2022 中使用 QT（使用该插件）

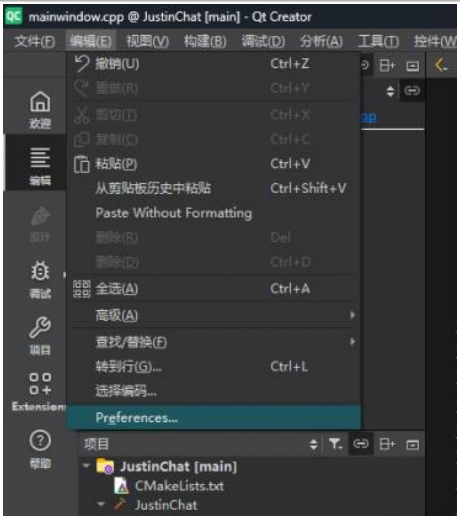
（ https://marketplace.visualstudio.com/items?itemName=TheQtCompany.QtVisualStudioTools2022 ）



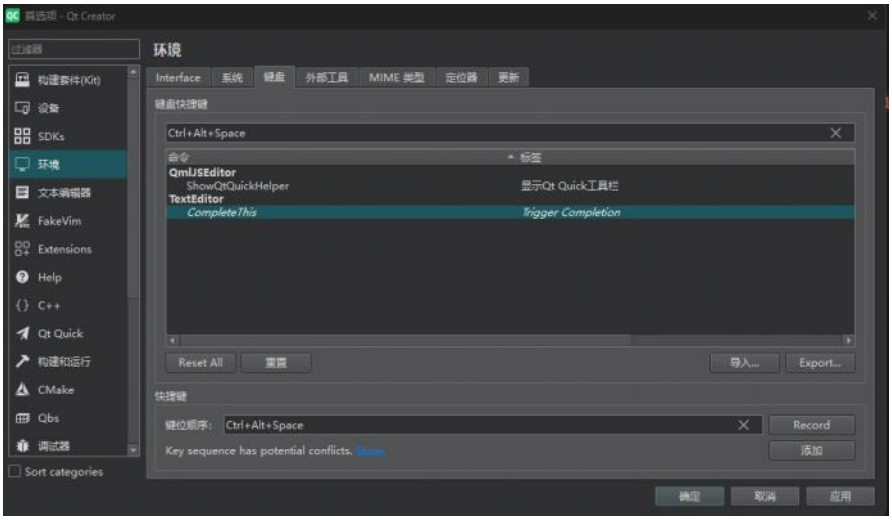
如何为 VS 添加插件，还请自行搜索。

》》》》QT 中的快捷键修改 :)

第一：智能感知

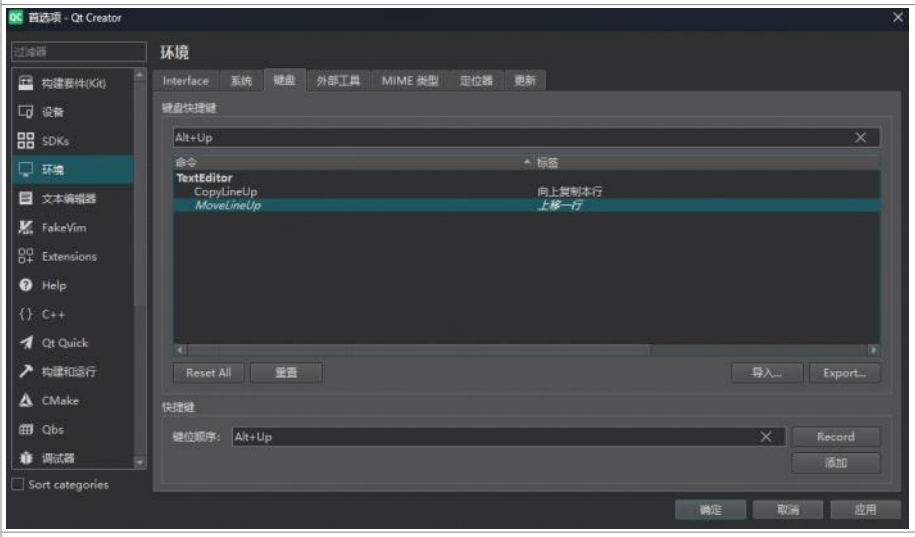


选择环境 -> 键盘 -> 搜索 CompleteThis ， 由于智能感知的默认快捷键是 Ctrl+Space,这于 windows 系统上的输入法切换起了冲突，所以我将其更改为 Ctrl+Alt+Space（来自后期：不好意思记错了，VS中的函数参数快捷提示是：Ctrl+Shift+Space）（这和VS中的函数参数提示快捷键相同。由于 Ctrl+J 被QT Creator 中的其他功能占用了，所以我选择了这个。）



QT中代码上移一行的快捷键是 Ctrl+Shift+Up，VS 中是 Alt+Up

（下移同理：QT中为Ctrl+Shift+Up，VS为 Alt+Down ）



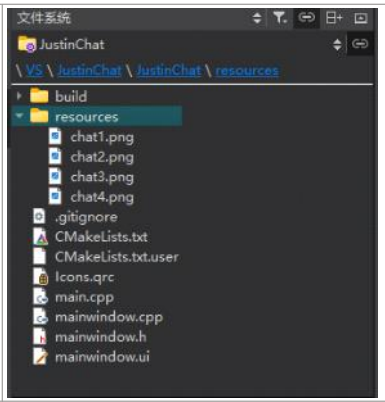
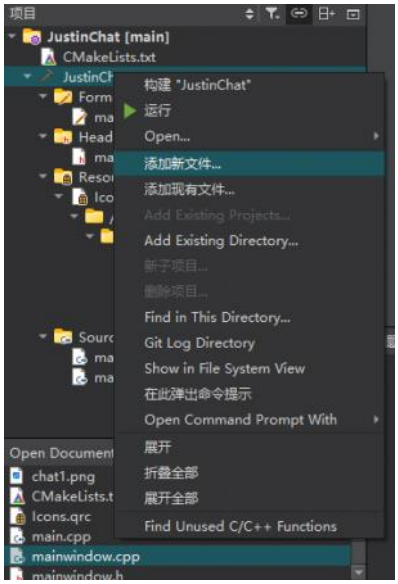
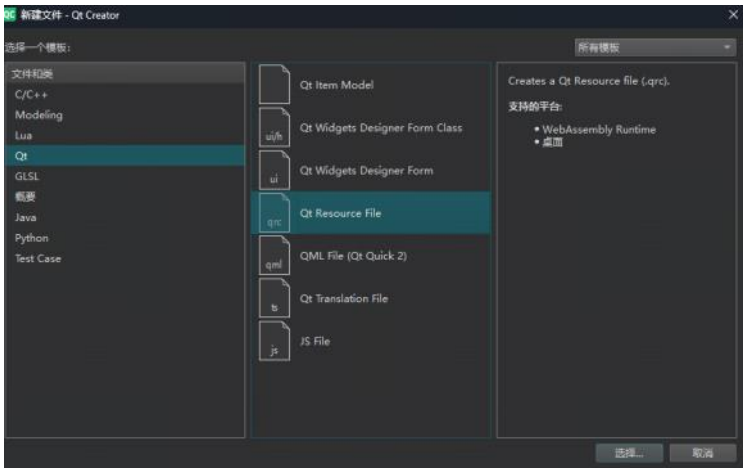
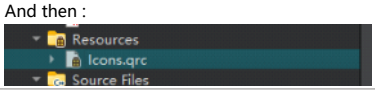
QT 中的向下复制是 Ctrl+Alt+Down, VS 中是 Ctrl+D

QT 中的转到行是 Ctrl + L , VS 中是 Ctrl + G

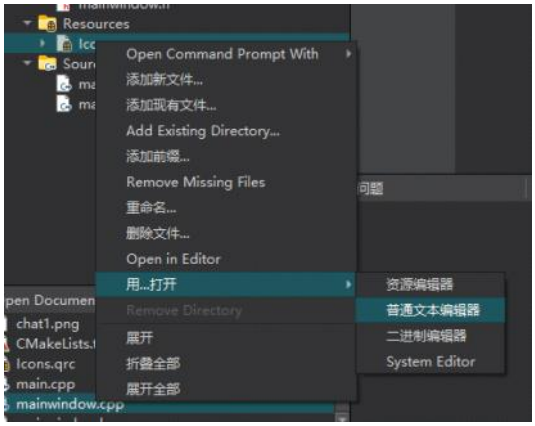
》》》》关于 Cmake 的知识（教程）：

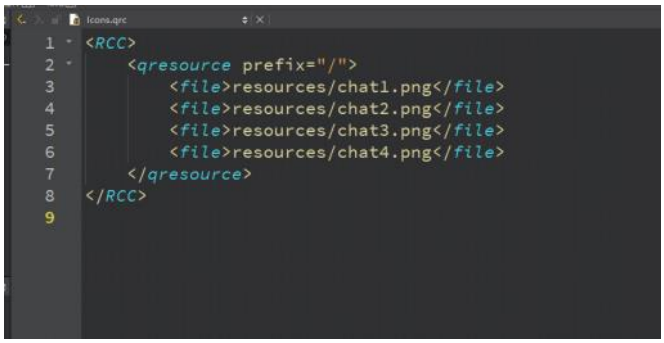
（英文）	https://cliutils.gitlab.io/modern-cmake/README.html
（中文）	https://modern-cmake-cn.github.io/Modern-CMake-zh_CN/

》》》》如何在不使用 cmake 或者 gmake 的情况下为主窗口更换 Icon ?

Step1:在项目根目录下创建一个 resources 文件夹，用于存放资源	
Step2:为项目创建一个 .qrc 文件	  And then : 

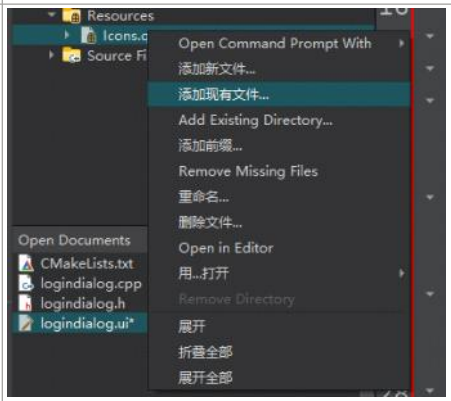
Step3（方法1）：用文本编辑器打开，并在 .qrc 中填入信息





```
<RCC>
    <qresource prefix="/">
        <file>resources/chat1.png</file>
        <file>resources/chat2.png</file>
        <file>resources/chat3.png</file>
        <file>resources/chat4.png</file>
    </qresource>
</RCC>
```

Step3（方法二）通过快捷键<添加现有文件>来直接将外部文件添加到 .qrc 文件中
（想要添加的文件可能位于项目之外，为了增强项目的可移植性，可以选择将文件存放在QT项目之内，比如 Step1 的操作）



Step4:最后在主窗口中添加语句：

```
#include "mainwindow.h"
#include "./ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    this->setWindowIcon(QIcon(":resources/chat3.png"));  // 将资源中的 Icon 设置在窗口上
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

》》》》 qss 文件编写格式文档
(参考网址)[https://doc.qt.io/qt-6/stylesheet-syntax.html]

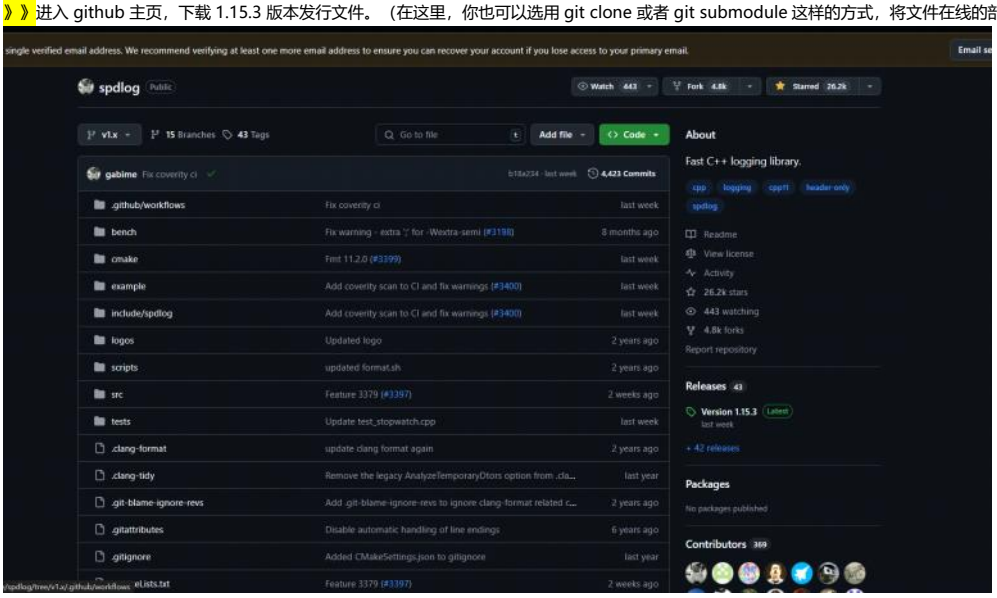
.qss 文件是 Qt Style Sheets 文件，用于在 Qt 应用程序中定义界面元素的样式和外观，类似于网页中的 CSS（Cascading Style Sheets）。

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。

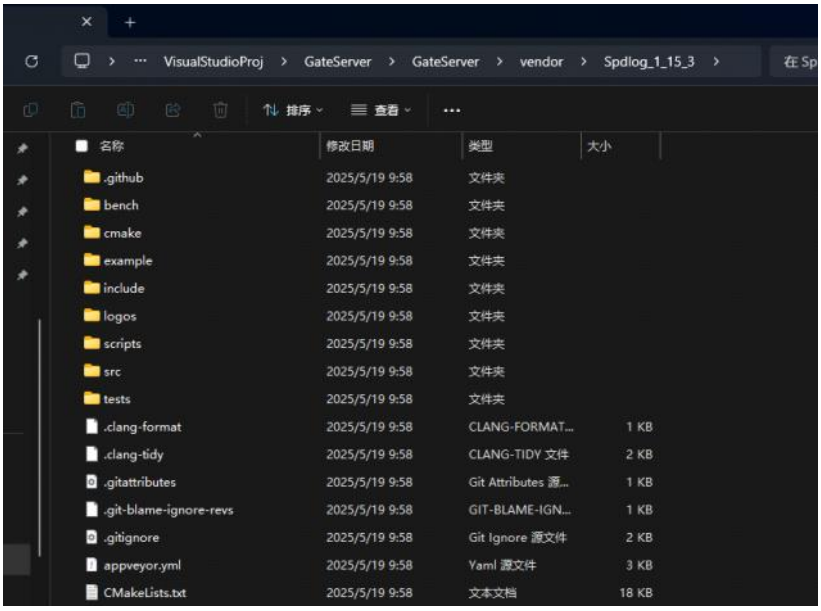
》》》》spdlog 日志库

接下来我们会经常编码，为了及时记录错误，我想引入日志库 spdlog。

》》进入 github 主页，下载 1.15.3 版本发行文件。（在这里，你也可以选用 git clone 或者 git submodule 这样的方式，将文件在线的部署到项目文件中。不过我选择下载发行文件）



下载完成后，将其解压到 GateServer\Vendor 新建的 Spdlog 目录下



》》更新 premake 文件，并运行 .bat 文件

```
    -- 添加包含目录
    includedirs
    {
        "%{prj.name}/vendor/Boost1_88_0/include",
        "%{prj.name}/vendor/Jsoncpp_1_9_6/include",
        "%{prj.name}/vendor/Spdlog_1_15_3/include"
    }

    -- 添加库目录
```

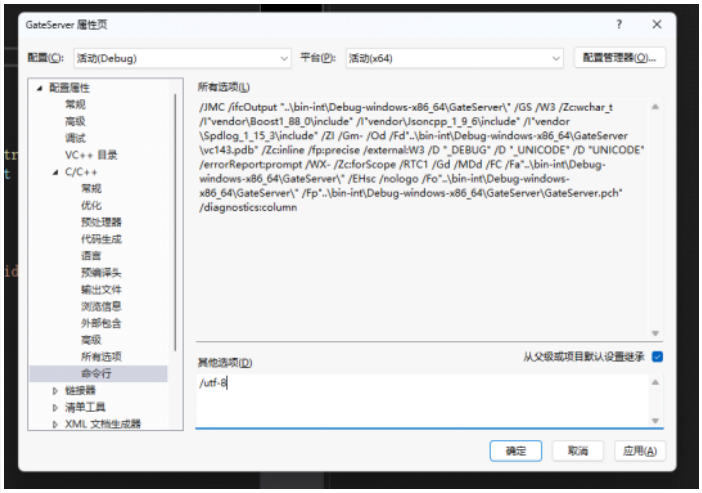
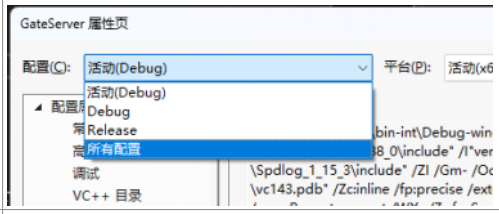
》》打开 GateServer，尝试编译，但是出现了错误，为什么呢。

```
files
{
    "%{prj.name}/**.h",
    "%{prj.name}/**.cpp"
}
```

这是因为我们之前在premake 脚本中跟踪源文件时，包含了 GateServer 名下的所有 .h 和 .cpp 文件，这导致我们不可避免的将 vendor 中的 .h 和 .cpp 文件也包含了进来。
如果这些项目以外的文件参与了编译，则会导致项目无法正常运行，所以我们只需要包含项目的文件即可。

```
-- 添加源文件
files
{
    "%{prj.name}/*.h",
    "%{prj.name}/*.cpp"
}
```

》》尝试运行，发现只剩下一个错误：

错误分析：	该错误来自 fmt 库（spdlog 的依赖），要求项目使用 UTF-8 编码 编译。Visual Studio 默认使用系统本地编码（如 GBK），未启用 /utf-8 选项。
错误解决方案 1	Visual Studio 设置： 打开项目属性 → C/C++ → 命令行 → 在 其他选项 中添加：/utf-8  记得调整一下配置（如果你需要对所有配置都进行调整的话：） 
错误解决方案 2	通过 premake 脚本进行设置

```
workspace "GateServer"
    architecture "x64"
    --startproject "NAME"        --[[启动项目]]

    configurations
    {
        "Debug",
        "Release"
    }

    -- 针对 MSVC 工具链启用 /utf-8
    filter "toolset:msc*"
        buildoptions { "/utf-8" }
```

filter "toolset:msc*"	msc* 匹配所有 MSVC 工具链（如 msvc、msvc-llvm）。 这能够确保 /utf-8 仅在 Windows + MSVC 环境下生效，避免影响其他平台（如 Linux/GCC）。
buildoptions 的作用：	直接向编译器命令行添加选项，等效于手动在 Visual Studio 中添加 /utf-8。

需要注意的是 configuration 需要放置在 filter "toolset:msc*" 之前。如果放在filter "toolset:msc*" 之下的话，会导致configuration 受到 filter 的影响。

configurations 在 workspace 的全局作用域中直接定义，避免被 filter 块覆盖。

》》调整过后，可以运行。接下来我们来编写代码。(我从之前一个项目中拿来了和谐代码。关于代码的设计可以自行查阅，不过没有这个必要，大概看一边，下次有用到的时候，从之前的项目中粘贴一下就好了)

需要注意的是，如果你想将其放置在自己的项目中，需要更改命名空间、输出文件的命名、宏定义的命名（ JChat、JustinChat.log、JC_CORE_WRAN）等等

.cpp

```
#include "Log.h"

#include <spdlog/sinks/stdout_color_sinks.h>
#include <spdlog/sinks/basic_file_sink.h>

namespace JChat {

    std::shared_ptr<spdlog::logger> Log::s_CoreLogger;//静态成员变量必须要进行定义（定义处可以选择是否进行初始化）

    void Log::Init()
    {
        //LogSink（日志目标）：表示日志消息的输出方式或存储位置，并将其存储在智能指针 shared_ptr 中
        std::vector<spdlog::sink_ptr> logSinks;
        // 存储一个将日志消息输出到控制台，并带有颜色的日志目标。
        logSinks.emplace_back(std::make_shared<spdlog::sinks::stdout_color_sink_mt>());
        // 指定一个日志目标，输出到 "Nut.log" ，并在每次启动时清空文件（true 表示清空，false 表示追加写入）。
        logSinks.emplace_back(std::make_shared<spdlog::sinks::basic_file_sink_mt>("JustinChat.log",
true));

        //样式设计
        logSinks[0]->set_pattern("%^[%T] %n: %v%$");
        logSinks[1]->set_pattern("[%T] [%l] %n: %v");

        //创建日志记录器
        // Logger with range on sinks: logger(std::string name, It begin, It end) 传入所有需要使用的日志目标
        s_CoreLogger = std::make_shared<spdlog::logger>("JstChat", begin(logSinks), end(logSinks));
        // Register the given logger with the given name(使用给定的名称注册给定的记录器)
        spdlog::register_logger(s_CoreLogger);
        // Core_Logger 记录 Trace 及以上级别的信息
        s_CoreLogger->set_level(spdlog::level::trace);
        s_CoreLogger->flush_on(spdlog::level::trace);

    }

}
```

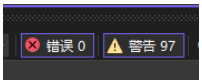
.h

```h
#pragma once

#include <spdlog/spdlog.h>
#include <spdlog/fmt/ostr.h>

namespace JChat {

    class Log
    {
    private:
        static std::shared_ptr<spdlog::logger> s_CoreLogger;//静态成员函数只能访问静态成员变量
    public:
        static void Init();
        inline static std::shared_ptr<spdlog::logger>& GetCoreLogger() { return s_CoreLogger; }
    };

}

//core log macros
#define JC_CORE_TRACE(...)\
    ::JChat::Log::GetCoreLogger()->trace(__VA_ARGS__);  //Before the JC needs "::": 表示在全局中调用这个GetCoreLogger

#define JC_CORE_INFO(...)\
    ::JChat::Log::GetCoreLogger()->info(__VA_ARGS__);   //(...)表示宏函数可以接受任意个参数

#define JC_CORE_WARN(...)\
    ::JChat::Log::GetCoreLogger()->warn(__VA_ARGS__);   //__VA_ARGS__是一个预定义的宏（前后的双下划线表示这是一个预定义的），可以用来动态的接收的未知个参数。

#define JC_CORE_ERROR(...)\
    ::JChat::Log::GetCoreLogger()->error(__VA_ARGS__);

#define JC_CORE_CRITICAL(...)\
    ::JChat::Log::GetCoreLogger()->critical(__VA_ARGS__);
```

》》运行结果：

```cpp
int main()
{
    JChat::Log::Init();
    JC_CORE_WARN("Hello world")

    Json::Value root;
```

```
[11:38:57] JstChat: Hello world
request is {
        "data" : "hello world",
        "id" : 1001
}

msg id is 1001 msg is "hello world"

E:\VS\JustinChat\VisualStudioProj\GateServer\bin\Debug-windows-x86_64\GateServer\GateServer.exe (进程 32716)已退出，代码为 0 (0x0)。
要在调试停止时自动关闭控制台，请启用"工具"->"选项"->"调试"->"调试停止时自动关闭控制台"。
按任意键关闭此窗口. . .
```



至于错误，在我的生涯中，我几乎没怎么关照过。问题不大。

》》》》修改文件构架（非必要，个人习惯而已）

这里的 bin 和 obj 看着实在太丑了，还有这个 GateServer.cpp，必须给他放在 src 里面去。

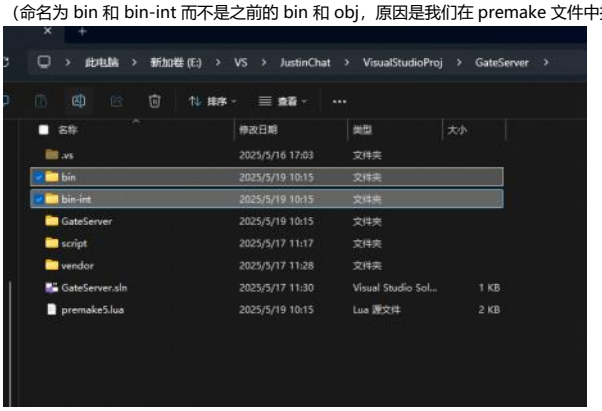
》》调整 tar 和 obj 输出目录

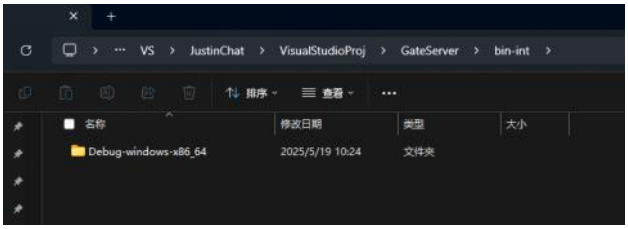
首先删除当前存在于文件系统中的 bin 和 obj 文件夹，并在 premake 脚本中更改 tar 和 obj 的输出目录（将其改为输出至项目根目录下）

```
project "GateServer"
    location "GateServer"
    kind "ConsoleApp"
    language "C++"
    --cppdialect "C++17"          --C++标准（编译时）

    targetdir ("bin/%{cfg.buildcfg}-%{cfg.system}-%{cfg.architecture}/%{prj.name}")
    objdir ("bin-int/%{cfg.buildcfg}-%{cfg.system}-%{cfg.architecture}/%{prj.name}")

    -- 宏定义
    defines
```

修改完成后运行一下 .bat 文件，然后重载并重新运行项目。

可以看到 targer 和 obj 已经被输出在根目录下了
（命名为 bin 和 bin-int 而不是之前的 bin 和 obj，原因是我们在 premake 文件中指定了他们输出于 bin 和 bin-int 中，而不是 bin 和 obj）





由于项目路径发生了改变， .gitignore 文件也要跟进一下。

```
/VisualStudioProj/GateServer/.vs
/VisualStudioProj/GateServer/bin
/VisualStudioProj/GateServer/bin-int
```

》》还有这个 .cpp ，我放在 src 文件夹中。

由于项目路径改变，Premake 脚本也要跟进一下

```
files
{
    "%{prj.name}/src/*.h",
    "%{prj.name}/src/*.cpp"
}
```

》》》添加断言的宏定义（个人喜好，非必要）
为了能够方便的为程序添加一个断点（输入一个条件，如果识别到条件错误，则自动引发断点），我对程序进行以下设置。

》》首先，让程序自动的在 Debug 时，定义一个名为 JC_DEBUG 的宏定义

```
72      -- 运行时库配置
73      filter "configurations:Debug"
74          defines { "JC_DEBUG" }
75          symbols "On"
76          runtime "Debug"  -- 对应 /MDd
77
78      filter "configurations:Release"
79          defines { "JC_RELEASE" }
80          optimize "On"
81          runtime "Release"  -- 对应 /MD
```

》》然后进行编程（内容放置在 Base.h 之下）

```
#ifdef JC_DEBUG
    #define JC_ENABLE_ASSERTS

    #ifdef JC_PLATFORM_WINDOWS
        #define JC_DEBUGBREAK() __debugbreak();
    #else
        #error "Platform doesn't support debugbreak yet! "
    #endif
#else
    #define JC_DEBUGBREAK()
#endif

#ifdef JC_ENABLE_ASSERTS
    #define JC_CORE_ASSERT(x, ...) \
            {if(!x){\
                JC_CORE_ERROR("Assertion Failed: {0}", __VA_ARGS__);\
                JC_DEBUGBREAK();}\
            }
#else
    #define JC_CORE_ASSERT(x, ...)
    #define JC_ASSERT(x, ...)
#endif
```

这个文件的大致作用是，识别程序是否运行在 window x64 下的 debug 模式下，如果是，则定义一个宏，内容是:

```
#define JC_CORE_ASSERT(x, ...) \
    {if(!x){\
        JC_CORE_ERROR("Assertion Failed: {0}", __VA_ARGS__);\
        JC_DEBUGBREAK();}\
    }
```

具体请查看 base.h 文件