1.关于前端和bash

2024年5月25日 14:12

》》》》写在前面（一些指令）

>>>ping 域名	可以用于测试与目标网站（域名）之间的网络连接

>>>$ git difftool --dir-diff	用于使用绑定的difftool(BeyondCompare)进行远程仓库和本地仓库的差异代码查看 （保存于 C:\Users\"username"\AppData\Local\Temp\git-difftool.）
>>>git clone --branch <version_number> <URL>	克隆对应版本的库

>>>git checkout <version_number>	1.克隆最新版本之后，进行本地库文件的版本更换（通常 version_number 在 tags 中） 2.或者说是用来切换分支的时候使用，<version_number>括号里面就可以从 tags 换为 branches

>>>git status	查看当前分支的状态（暂存区中是否有未提交、工作树状态）
>>>git submodule add <URL> <path_you_want_save>	添加子模块
>>>git pull origin <branch_name>	对本地库的一个分支进行更新并将更改合并到本地

>>>
>>>
>>>
>>>
>>>
>>>
>>>

》》》》！！！！！！！！！！！！！！！！！！！！！

don't konw why but since i use VScode terminal to run the git command,the error which always appered in cmd is disapper.(like fatal:.....)
不知道为什么，但是由于我使用 VScode 终端运行 git 命令（从github desktop选项"Open in Visual Studio Code"进入的），（库存在目录下打开的）
cmd中使用Git命令时总是出现的错误消失了。（如 fatal:.....）

》》》》git bash第一次使用需要绑定用户信息

>>>git config --global user.name"your name"
>>>git config --global user.name"your email"

》》》》CMD（命令提示符）和 PowerShell 的区别。

功能和语法： PowerShell 是更现代化和功能更强大的命令行工具，提供了许多强大的功能，如管道操作、脚本编写、自定义函数、异常处理等。
与此相比，CMD 较为简单，功能相对有限，其语法和命令选项也受限制。

跨平台支持： PowerShell 是跨平台的，可以在 Windows、Linux 和 macOS 等多个操作系统上运行。
而 CMD 只能在 Windows 系统中使用。

脚本支持： PowerShell 是一种强大的脚本编程语言，并且与 .NET 框架深度集成，可以直接调用和操作 .NET 类库。因此可以用于编写复杂的自动化脚本和管理任务。
CMD 也支持一些简单的脚本编写，但脚本功能和灵活性相对较弱。

命令和别名： PowerShell 提供了大量的命令和模块，涵盖了广泛的系统管理和任务自动化需求。此外，PowerShell 还支持别名和函数。
CMD 的命令和功能相对较少，没有别名和函数的概念。

》》》》IP 地址、域名和网址、URI、主机名、端口

IP 地址（Internet Protocol Address）：	是一组数字，用于唯一标识计算机网络中的设备。IP 地址通常是由网络服务提供商（ISP）分配的，表示你的计算机在互联网上的位置。IPv4 地址由 4 个十进制数构成，每个数之间用点号分隔，如 192.168.1.1；而 IPv6 地址则更长，包含冒号和字母。

域名（Domain Name）：	是一个易于记忆的英文名称，用于代替 IP 地址来访问互联网上的服务器和网站。域名可以由多个部分组成，每个部分之间用点号分隔，例如 google.com 或 baidu.com。域名需要通过域名解析系统将其映射到相应的 IP 地址上，才能实现访问。

网址（URL，Uniform Resource Locator）：	是一个指向互联网上资源的地址，包括协议、域名、路径和查询参数等组成部分。网址通常用于浏览器中输入以访问网页或其他资源。例如，https://www.google.com/ 是一个网址，其中 https 是协议，www.google.com 是域名，后面的 / 是资源路径。

URI（Uniform Resource Identifier）：	用于标识互联网上的资源，包括 URL 和 URN（Uniform Resource Name）。与 URL 不同，URN 仅用于标识资源名称，而不包含访问该资源所需的协议和位置信息。

主机名（Hostname）：	是域名中的一部分，表示互联网上的服务器或设备的名称。主机名通常位于域名的最左侧部分，例如 www.google.com 中的 www。

端口（Port）：	用于标识计算机上运行的网络服务。每个网络服务都有一个唯一的端口号，例如 HTTP 服务的默认端口为 80，HTTPS 服务的默认端口为 443。在 URL 中，可以使用冒号和端口号将请求发送到正确的端口，例如 https://www.google.com:443/。

2.概念与操作

2024年5月24日 15:38

》》》》写在前面

win + x	打开快速访问菜单（包含一系列常用的系统管理工具和设置选项）
win + i	打开设置
Ctrl+shift+a	在文件资源管理器中打开当前文件
ctrl +j （alt + → ）	（在VS中）可以强制弹出当前对应的补全内容，这样就不用每次都删了重写来显示补全的信息了，很常用的一个小功能
Ctrl + Tab：	（VS中）按住Ctrl键并连续按下Tab键，可以在打开的文件之间进行切换。
Alt + D：	浏览器窗口中的地址栏会被选中并高亮显示，这使您可以立即开始输入网址或搜索内容
Win + R，输入Notepad：	打开记事本
.LOG：	为记事本使用时间戳
Shift + Ctrl + N：	新建文件夹
..\ ：	表示上一级文件
##	是预处理操作符，用于将宏参数连接起来。（EventType::##type 就会被展开为 EventType::Mouse）
#	是字符串化操作符，将宏参数转换为字符串。return #type; 如果 type 是 Mouse，那么 #type 就会被展开为 "Mouse"。
ctrl + n:	创建一个新的文件资源管理器界面（在文件资源管理器界面）
ctrl + L:	选中当前行（ VS2019 中好像被设置为剪切：参考官网）
Ctrl + Esc:	打开菜单（相当与单击 win ）
Ctrl + E:	和 Alt + D 类似（Ctrl + E 适合在进行搜索的时候快速定位到搜索框，而 Alt + D 更适合在直接编辑网址的时候快速定位到地址栏。）
Win + A:	打开快速设置页面
Ctrl + Shift + Space:	（在VS中）打开函数参数的提示框
VS2019中打开 minimap:	工具->选项->文本编辑器->C++->使用垂直滚动条的缩略图模式 (https://learn.microsoft.com/zh-cn/visualstudio/ide/how-to-track-your-code-by-customizing-the-scrollbar?view=vs-2022)
Win+Shift+N:	打开快速笔记 OneNote，或者在 Win+R 中输入 onenote 指令。

》》》》预定义的宏

参考网站：（ https://learn.microsoft.com/zh-cn/cpp/preprocessor/predefined-macros?view=msvc-170 ）

以下列举一些常用的：

标准预定义宏

`__DATE__`：	当前源文件的编译日期。 日期是 Mmm dd yyyy 格式的恒定长度字符串文本。 月份名 Mmm 与 C 运行时库 (CRT) asctime 函数生成的缩写月份名相同。 如果值小于 10，则日期 dd 的第一个字符为空格。 任何情况下都会定义此宏。
`__FILE__`：	当前源文件的名称。 `__FILE__` 展开为字符型字符串文本。 要确保显示文件的完整路径，请使用 （诊断中源代码文件的完整路径）/FC。 任何情况下都会定义此宏。
`__LINE__`：	定义为当前源文件中的整数行号。 可使用 `#line` 指令来更改此宏的值。 `__LINE__` 值的整型类型因上下文而异。 任何情况下都会定义此宏。
`__TIME__`：	预处理翻译单元的翻译时间。 时间是 hh:mm:ss 格式的字符型字符串文本，与 CRT asctime 函数返回的时间相同。 任何情况下都会定义此宏。

Microsoft 专用预定义宏

`__COUNTER__`：	展开为从 `0` 开始的整数文本。 每次在源文件或源文件的 `included` 头中使用时，此值都会递增 `1`。 当使用预编译头时，`__COUNTER__` 会记住其状态。 任何情况下都会定义此宏。
`_CPPUNWIND`：	如果设置了一个或多个 /GX（启用异常处理）、/clr（公共语言运行时编译）或/EH（异常处理模型）编译器选项，则定义为 1。 其他情况下则不定义。
`_DEBUG`：	当设置了 /LDd、/MDd 或 /MTd 编译器选项时，定义为 1。 其他情况下则不定义。
`_DLL`：	当设置了 /MD 或 /MDd（多线程 DLL）编译器选项时，定义为 1。 其他情况下则不定义。
`__FUNCDNAME__`：	定义为包含封闭函数修饰名的字符串文本。 此宏仅在函数中定义。 如果使用 /EP 或 /P 编译器选项，则不会展开 `__FUNCDNAME__` 宏。

》》》》“x64(active) 平台”和“x64 平台”的区别：

x64(active) 平台：在Visual Studio中，"x64(active)" 是指针对 64 位处理器架构的项目配置。选择这个配置意味着项目将会被编译成针对 64 位处理器的可执行文件，可以充分利用 64 位处理器的性能和内存寻址能力。

x64 平台：同样也是指针对 64 位处理器架构的项目配置。在较早的版本的Visual Studio中，可能会看到简单的 "x64" 选项，它与 "x64(active)" 实质上是相同的，都代表了针对 64 位处理器的项目配置。

在新版的Visual Studio中，可能会看到 "x64(active)" 选项，这是为了更清晰地表示当前项目配置是针对 64 位处理器的。因此，在这种情况下，“x64(active) 平台”和“x64 平台”其实是指同一个概念，只是一种是较早版本的表示方式，另一种是较新版本的表示方式。

》》》》宏（条件判断的实现逻辑）

宏不会自动定义。如果在 属性页 -> C++ -> 预处理器 中填入一个宏XXX，
这意味着在编译代码时会自动在预处理阶段为XXX这个宏定义一个值。

这样不用手动编写一个宏，可以直接使用#ifdef语句进行条件判断。

```
<---- Eg.

#define 宏名称 值或代码

#ifdef 标识符
    // 如果标识符已经被定义，则编译这部分代码
#else
    // 如果标识符没有被定义，则编译这部分代码
#endif

```

》》》》什么是API，OpenGL是API吗？GLFW是API吗？

1.API（Application Programming Interface，应用程序编程接口）是一组定义了软件组件如何相互交互的规范。
它定义了一组规范、协议和工具，允许不同的软件组件（如库、框架、操作系统等）之间进行通信和交互。
2.OpenGL 是一个图形渲染 API，它定义了一系列函数和数据结构，用于执行各种图形操作。
OpenGL 提供了一个标准化的接口，使得开发人员可以在不同的平台上编写图形应用程序，而无需关心底层硬件的细节。
因此，OpenGL 是一个 API。
3.GLFW（Graphics Library Framework）是一个用于创建窗口和处理用户输入的库，它并不是一个 API，而是一个库（Library）。
GLFW 提供了一系列函数辅助开发者创建基于 OpenGL 的图形应用程序，它使用 OpenGL 的 API 来实现其功能。

》》》》什么是打包?

封装（Packaging）可以理解为打包的一种表达方式，在软件开发中常用来指代将相关文件和资源封装到一个独立的包中以便于分发和使用。
（打包和封装在大部分情况下可以视作同义词。）

打包（Packaging）指的是将软件或应用程序的源代码、依赖项和其他必要的资源组合在一起，以便在其他环境中进行部署或分发。
它可以将一个或多个文件或目录打包成一个单独的可执行文件、库文件、安装程序、容器镜像等形式。

打包通常包括以下内容：

源代码：包括软件的原始代码文件，用于编译和构建可执行文件或库。
依赖项：软件所依赖的库文件、框架、工具或其他第三方组件。这些依赖项可能需要事先安装或打包到同一个包中，以确保软件在目标环境中能够正常运行。
资源文件：例如配置文件、模板、静态文件、图像、文档等，这些文件通常是软件运行所需的辅助资源。

```
eg:
class Calculator {
public:
    int func( int a ){
        return XXX::complex_function( a );
    }
};

int main(){
    Calulator calc;
    std::cout << "result is " << calc.func( X );
}

```

》》》》共享指针（智能指针）

特点：
1.引用计数：共享指针使用引用计数来追踪有多少个指针共同拥有一个对象。并自动管理对象的生命周期。

case 1：	每当创建一个共享指针指向对象时，引用计数加一；
case 2：	当销毁一个共享指针或者将其重新分配给其他对象时，引用计数减一。
case 3：	当引用计数为零时，表示没有指针指向对象，对象会被自动销毁。

2.多个指针共享所有权：共享指针允许多个指针同时拥有对同一个对象的所有权，这使得多个部分可以共享对象的状态和数据。

eg:
使用 std::shared_ptr 创建两个共享指针 ptr1 和 ptr2，它们都指向同一个 MyClass 对象。
通过共享指针，我们可以同时使用 ptr1 和 ptr2 访问和管理对象。

》》》》静态函数

静态函数是属于类而不是对象的函数，它们没有隐式的 this 指针，并且可以直接通过类名进行调用。
（静态函数适用于执行与类相关但不依赖于类的实例的操作，共享资源或封装辅助函数的场景。）
特点：
1.在一个静态成员函数中，只能访问静态声明的成员变量。
2.对于静态成员变量，你必须在某个地方进行定义。（并且定义处可以选择是否进行初始化）
这是因为静态成员变量属于类而不属于类的对象，所以需要在类外进行定义。
如果你没有提供定义，编译器将无法找到静态成员变量的实际存储位置，从而导致链接错误。

```
eg:
class log.h:
    class NUT_API Log
    {
    private:
        static std::shared_ptr<spdlog::logger> s_CoreLogger;
    public:
        inline static std::shared_ptr<spdlog::logger>& GetCoreLogger
    };

log.cpp:
std::shared_ptr <spdlog::logger> Log::s_CoreLogger;
```

》》》》::namespace::func();
"::Nut" 中的 "::" 表示全局命名空间，即根命名空间。表示访问全局命名空间中的 Nut 命名空间。

```
eg:
namespace Nut {
    void foo() {}
}

int main() {
    ::Nut::foo();       // 调用全局命名空间中的 Nut 命名空间下的 foo 函
    return 0;
}
```

》》》》双下划线 "__" ----> 预定义的宏
定义： 双下划线 "" 表示这是一个预定义的宏，由编译器或标准库定义。
目的： 一些预定义的宏都包含双下划线 "__"，例如 __cplusplus、LINE、FILE 等等。这样设计的目的是为了避免与用户自定义的标识符冲突，并且提供一些方便的功能。

》》》》(...) 和 __VA_ARGS__配对使用

1.(...) 是可变参数模板的语法，表示宏函数可以接受任意数量的参数。
2.VA_ARGS 是一个预定义的宏， 在 C++ 中用于表示可变参数列表。它将被展开成实际传入的可变参数列表。

一般情况下，在宏定义中使用 (...) 来接受可变数量的参数，在宏展开时使用 VA_ARGS 来引用这些参数。
下面是一个示例来说明 (...) 和 VA_ARGS 的配对使用：

```
#define PRINT_VALUES(format, ...) \
    printf(format, __VA_ARGS__);

int main() {
    PRINT_VALUES("%d %s\n", 10, "Hello");  // 输出：10 Hello
    return 0;
}

```

在这个例子中，PRINT_VALUES 宏使用了可变参数模板 (...) 来接受可变数量的参数，然后使用 VA_ARGS 来引用这些参数。
在宏展开时，VA_ARGS 将被实际传入的可变参数替换。

》》》》什么是bat文件
概念： BAT 文件是批处理文件的一种，它是一种包含一系列命令的文本文件，用于在 Windows 操作系统中批量执行命令。批处理文件使用扩展名为 .bat 或 .cmd。
用处： BAT 文件可以包含命令行命令、控制流语句（如条件判断和循环）、变量定义和其他批处理脚本语言的特性。
在BAT文件中输入call commands_in_cmd可以实现启动此BAT文件时，在当前路径的命令行中执行该命令。

》》》》什么是premake，什么是Lua
概念： Premake 使用 Lua 作为其脚本语言。
Lua 是一种轻量级、高效、可嵌入的脚本语言，非常适合用于配置文件和脚本语言的编写。

详见官方文档：（https://premake.github.io/docs/）

eg: 以下是 Premake 中 Lua 的一些基本语法和使用方法：

```
//1.注释，使用双连字符 "--" 进行单行注释，或者使用长注释形式 "--[[ ...
    --  单行注释
    --[[
        这是一个
        多行注释
    ]]


//2.变量：无需声明变量类型，直接赋值即可。
    name = "John"
    age = 25

//3.表（Table）：类似于字典或哈希表的数据结构，可以存储键值对。
    person = {
      name = "John",
      age = 25,
      gender = "male"
    }

//4.函数：使用 function 关键字定义函数。
    function greet(name)
      print("Hello, " .. name .. "!")
    end

    greet("John") -- 输出：Hello, John!

//5.控制流：支持条件语句（if-else）和循环语句（for、while）。
    if condition then
      -- 条件为真时执行的代码
    else
      -- 条件为假时执行的代码
    end

    for i = 1, 5 do
      print(i)
    end

    while condition do
      -- the 循环体
    end

//6.Premake API：Premake 提供了一组 API 来定义项目的属性、构建规则和配
-- 定义项目
project "MyProject"
  kind "ConsoleApp"
  language "C++"
  files { "src/*.cpp", "include/*.h" }
  includedirs { "include" }
  links { "LibraryA", "LibraryB" }

-- 定义构建规则
filter "system:windows"
  defines { "WINDOWS" }

filter "system:linux"
```

》》》》友元类和派生类的区别：
1.友元类是一种权限控制机制， 用于让一个类可以访问另一个类的私有和受保护成员，而不是继承关系。
2.派生类是一种继承关系， 用于创建一个新的类并从另一个类继承属性和方法，可以对继承的成员进行扩展和修改。{重写（override）}
派生类可以访问基类中的公有和受保护的成员，但不能直接访问基类中的私有成员。

》》》》预编译头，看视频吧
（【【72】【Cherno C++】【中字】C++的预编译头文件】
https://www.bilibili.com/video/BV1eu411f736/?share_source=copy_web&vd_source=ca2feff7d155a2579964dfa2c3173769）

```
50
51  -- 定义构建规则
52  filter "system:windows"
53    defines { "WINDOWS" }
54
55  filter "system:linux"
56    defines { "LINUX" }
57
58  //7. ".." 是字符串连接操作符。
59  local str1 = "Hello"
60  local str2 = "World"
61  local result = str1 .. " " .. str2  -- 将两个字符串连接在一起，中间加
62  print(result)  -- 输出: Hello World
```

》》》》premake脚本中的 include

作用：：通常情况下，在 Premake 脚本中使用 include 语句的主要目的是为了包含并执行指定目录下的 Premake 脚本文件。

解释：：

当你在外部的一个 Premake 脚本中写下 include "Nut/vendor/GLFW" 时，这一句实际上是告诉 Premake 构建系统去包含并执行 Nut/vendor/GLFW 目录下的 premake5.lua 文件。这里的含义是引入这个目录下的 Premake 脚本，并将其中定义的项目配置等内容合并到当前的 Premake 脚本中。

虽然 GLFW 目录下可能包含很多文件，但在构建系统中，通常会有一个用于配置和管理该库的专门的构建脚本（比如 premake5.lua）。通过在外部 Premake 脚本中使用 include "Nut/vendor/GLFW"，你实际上是在告诉 Premake 去处理 GLFW 这个目录作为一个整体，而不需要手动列出其中的所有文件。

实际逻辑：：在 Premake 中的 include 语句并不是简单的文本替换或复制粘贴操作。当执行 include 语句时，Premake 会实际上去加载并执行指定目录下的 Premake 文件，并将其中定义的项目配置、编译选项等内容合并到当前的 Premake 上下文中。

》》》》占位符格式化输出信息

eg.

NUT_CORE_INFO("Creating window: {0} ({1} * {2})", props.Title, props.Width, props.Height); 就是在使用占位符来格式化输出信息。

{0}，{1}，{2}分别代表后面的三个参数信息

》》》》》》》对于一个数据 double pos，使用 (float)pos 和 float(pos) 这两种方式的类型转换有什么不同

1.(float)pos	是一种 C 风格的类型转换方式。这种方式在 C++ 中仍然有效，但不够安全，因为它可以进行任意类型的转换，包括隐式转换和强制转换，可能会导致潜在的错误。
2.float(pos)	是一种 C++ 风格的类型转换方式，称为函数风格的强制类型转换（functional cast）。这种方式在 C++ 中更为推荐，因为它提供了更明确的类型转换操作，同时在某些情况下还能提供更好的类型安全性。（会有警告但不会影响正常运行）

》》》》函数指针、std::bind 返回的对象、std::function 返回的对象和 lambda 表达式都可以用来表示函数，但它们之间有一些区别

1.函数指针（Function Pointers）：

概念：	函数指针是指向函数的指针，可直接调用目标函数。 使用函数指针可以直接传递函数地址，或者调用函数。
适用情况：	适用于简单的函数调用，无需状态或额外参数。

2.std::bind：

概念：	std::bind 可以用来将成员函数、自由函数、函数对象等绑定到一个函数对象上，并延迟执行。 可以绑定函数、成员函数、lambda 表达式等，以及提供额外参数。
适用情况：	通常用于创建包装了函数调用和参数的可调用对象。适用于需要延迟执行函数调用或传递额外参数的情况。

3.std::function：

概念：	std::function 是一个通用的可调用对象包装器，可以容纳任意可调用实体（函数指针、函数对象、成员函数指针、 lambda 等）。可以像函数指针一样调用 std::function 对象。
适用情况：	通常用于需要在运行时确定要调用的函数的情况，或者需要存储不同类型的可调用对象。

4.Lambda 表达式：

概念：	Lambda 表达式是一种用于定义匿名函数的语法，可以捕获外部变量。 可以直接在需要函数的地方定义并使用，比较灵活。
适用情况：	适用于需要编写临时的、较短小的函数，以及需要捕获外部作用域变量的情况。

使用情况：

1.使用函数指针	当您只需要简单地传递函数地址或进行函数调用时。
2.使用 std::bind	当您需要绑定函数及其参数，并且可能需要延迟执行函数。
3.使用 std::function	当您需要一个通用的可调用对象容器，并且希望能够存储不同类型的可调用实体。
4.使用 lambda 表达式	当您需要编写临时的、较短小的函数，以及需要捕获外部作用域变量的情况。

》》》》封装的演示：

```
1   #include <iostream>
2   #include <stdexcept>
3
4   // 原始函数，模拟需要进行权限验证的功能
5   void sensitiveOperation( auto args )
6   {
7       std::cout << "Sensitive operation performed" << std
8       //Or some codes for operate
9   }
10
11  // 用户类，用于存储用户权限信息
12  class User {
13  public:
14      bool hasPermission(const std::string& permission) {
15          // 检查用户是否具有特定权限
16          // 这里简化为直接返回 true
17          return true;
18      }
19  };
20
21  // 权限验证包装器
22  template <typename Func>
23  auto permissionWrapper(Func func, const User& user)
24  {
25      return [ func,  &user ]( auto args ) {
```

》》》》封装和包装

1.概念：封装（Encapsulation）	指的是将数据和行为（方法）捆绑在一个单元中，并对外部隐藏对象的内部状态（实现细节），只通过公共接口来访问和操作对象。
目的：	封装提供了一种保护对象状态的机制，可以有效地控制对象的访问权限。更容易维护。
2.概念：包装（Wrapper）	用于将一个对象或函数封装在另一个对象或函数中，以提供额外的功能或修改原始对象或函数的行为，同时保持原始接口不变。
目的：	包装器通常用于添加额外的功能、修改行为或者适应不同的需求，而不会改变原始对象或函数的核心逻辑。

区别：

封装更侧重于“隐藏对象的内部细节和提供公共接口”，以实现数据和行为的封装；

包装则更侧重于在不改变接口的情况下，“为对象或函数添加额外的功能或修改其行为”。

演示： ---->

》》》》关于emplace 和 push

emplace & push

emplace_back & push_back

在用法和语义上都是一样的，之不过push会额外复制一次用来传输

》》》》关于emplace 和 push

emplace & push

emplace_back & push_back

在用法和语义上都是一样的，之不过push会额外复制一次用来传输

》》》》emplace 和 emplace_back

emplack_back( )	只接受一个参数，就是传入的元素，因为会直接插入到vector的末尾
emplace( )	接受两个参数，一个是插入的位置，第二个是传入的元素。

但是emplace有两个要点：

1.插入的元素的位置在填入的位置之前。

2.插入之后，会返回一个值，该值为新插入的元素的位置。

》》》》vector.end( ) 是什么？std::find( ) 的返回值在什么时候与其相同？

1.m_Layers.end( ) 返回的是指向容器中最后一个元素之后的位置（即尾后迭代器），而不是指向空的指针。

2.当std::find( ) 没有找到所查证的元素时。就会返回vector.end( ) 相同的值。所以

```
auto it = std::find(m_Layers.begin(), m_Layers.end(), layer);
    if (it != m_Layers.end()) {
        XXXXXX
    }
```

多可以用来在vector中从头到尾的查询一个值，并做出判断。

```
// 权限验证包装器
template <typename Func>
auto permissionWrapper(Func func, const User& user)
{
    return [ func,  &user ]( auto args ) {
        if (user.hasPermission("execute")) {
            // 具有权限，调用原始函数
            func( args );
        } else {
            // 没有权限，抛出异常或返回错误信息
            throw std::runtime_error("Permission denied
        }
    };
}

int main()
{
    // 创建一个用户
    User currentUser;

    // 使用权限验证包装器
    auto wrappedFunc = permissionWrapper(sensitiveOpera

    // 调用被包装的函数
    try {  wrappedFunc( //args or none );  }
    catch (const std::exception& e) {  std::cerr << "Er

    return 0;
}
```

》》》》函数声明的最后const是什么意思？

在这个成员函数不允许修改成员变量的值

》》》》序列化是啥？

序列化是指将数据结构或对象转换为一种特定格式，以便于存储或传输，同时也可以在需要时将其恢复为原始的数据结构或对象的过程。

通常涉及以下几个方面：

将数据结构转换为字节流：将数据结构中的各个字段按照一定规则编码成字节流，以便于存储或传输。

包含足够的信息：序列化的数据应该包含足够的信息，以便在反序列化时能够准确地重新构建原始的数据结构或对象。

》》》》.hpp和.h文件的区别

目的：

为了更清晰地表明文件是用于 C++ 的代码还是传统的 C 代码。

.h 文件通常用于传统的 C/C++ 头文件

而 .hpp 文件则有时被用于 C++ 头文件，特别是在涉及到 C++ 的特性或标准库时。

》》》》什么是 constexpr？

概念：

constexpr是C++11引入的一个关键字，用于声明可以在编译时求值的常量表达式。它可以用于变量、函数、构造函数等的声明中。

变量：

通过在变量声明前加上constexpr关键字，可以将其定义为常量表达式。

必须满足以下条件：

初始化表达式在编译时就能计算出结果；

变量类型必须是字面类型（literal type），即不能包含用户自定义类型的成员函数，且所有非静态成员必须也是字面类型。

```
eg.
    constexpr int k = 10;
    constexpr double pi = 3.1415926;
```

函数：

通过在函数声明前加上constexpr关键字，可以将其定义为常量表达式。

函数必须满足以下条件：

函数体中只能包含一行返回语句；

返回值必须是字面类型；

所有参数必须是字面类型。

```
eg.
    constexpr int factorial(int n) {
        return (n == 0) ? 1 : n * factorial(n - 1);
    }
    constexpr int result = factorial(5); // 编译时就能计算出结果
```

构造函数：

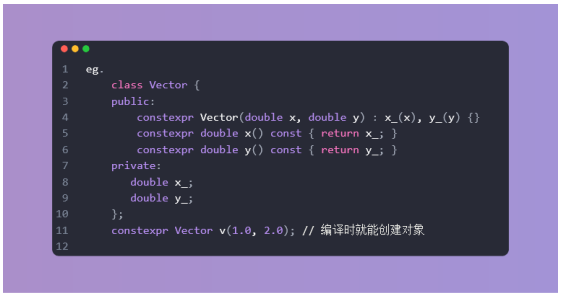
通过在构造函数声明前加上constexpr关键字，可以将其定义为常量表达式构造函数。

类必须满足以下条件：

类中的所有成员变量和成员函数都是字面类型；

构造函数只能有一个参数列表；

构造函数中只能使用一行赋值语句对成员变量进行初始化。



```
eg.
    class Vector {
    public:
        constexpr Vector(double x, double y) : x_(x), y_(y) {}
        constexpr double x() const { return x_; }
        constexpr double y() const { return y_; }
    private:
       double x_;
       double y_;
    };
    constexpr Vector v(1.0, 2.0); // 编译时就能创建对象
```

作用：

在使用constexpr时，最重要的是确保定义的常量表达式可以在编译时求值。

》》》》

int func();

int func() const;

const int& func();

const int& func() const; 的区别。

- **1.int func();**

非常量成员函数.

1.可以修改对象的成员变量

2.返回成员变量的拷贝

- **2.int func() const;**

成员函数

1.不会修改对象的成员变量

2.但是它返回的是成员变量的拷贝，而不是引用.(对返回值的修改不会影响到原始对象的成员变量。

- **3.const int& func();**

可以是一个全局函数，静态函数，或者是一个类的成员函数

1.在对应情况下，表示可以修改静态变量、全局变量或成员变量

2.返回一个对静态变量或者全局变量的常量引用

- **4.const int& func() const;**

成员函数

1.不会修改对象的成员变量

2.返回一个对成员变量的常量引用

GameEngine1

2024年5月25日 15:04

----------项目设置----------

》》》》项目设置：

1.建立一个github库

2.设置一个VS解决方案和项目

3.设置相应的配置

4.链接

1.配置：

设置引擎为一个库文件（dll），在外部将库文件链接到外部的应用项目（exe）

（静态库的形式类似于将一大堆库链接到游戏中）

（动态库的形式类似于将一大堆外部库先链接到dll文件中，再将这个dll文件链接到游戏中，这样我们的游戏只会依赖于这一个dll文件）

1.删除了适应平台（x86）

2.改变配置类型（exe -> dll）

3.更改输出目录和中间目录

2.新建一个项目并且配置其支持平台，输出和中间目录（和引擎相同）

3.设置启动项目：

右键Sandbox并选择该选项(.vs文件会保存我们在vs中做出的配置调整，但我们要为其他平台启动的人做一些调整0

4.调整sln文件中的启动项

在文本编辑器（可以是vscode）中打开解决方案文件.sln，调整前几句为

```
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "Sandbox", "Sandbox\Sandbox.vcxproj", "{28573136-9FAB-4D60-8F24-3DF8BCC0422B}"
EndProject
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "Nut", "Nut\Nut.vcxproj", "{7FB1D529-C182-497A-A2B0-633BC7A48C81}"
EndProject
```

5.链接

将启动项目和引擎库链接，右键sandbox -> add（添加） -> reference（引用） -> 勾选你的引擎文件

----------引擎入口点----------

》》》》什么是入口点

引擎入口点（Engine Entry Point）通常指的是一个程序的起始执行位置，也可以被称为主函数（Main Function）。程序从这里开始执行，并按照预定的流程继续执行。

eg:: 例如，在C语言中，引擎入口点通常被命名为main函数，它是程序的起始位置。在C++中，引擎入口点可以是全局的main函数，也可以是类的静态成员函数。

》》》》什么是沙盒(sandbox)

sandbox（沙盒）是一种安全机制，用于限制程序的访问权限和行为范围。它创建了一个受限的执行环境，将程序隔离在其中，以防止恶意代码或不安全的操作对系统造成损害。

》》》》虚析构函数?

当一个类的析构函数被声明为虚析构函数时，这意味着该类将成为多态类型，并且可以安全地通过基类指针删除派生类对象。

1.安全地销毁派生类对象：当使用基类指针指向派生类对象时，如果基类的析构函数不是虚函数，在使用delete删除指针时只会调用基类的析构函数。这可能导致派生类中的资源泄漏，因为派生类的析构函数未被调用。
通过将析构函数声明为虚函数，可以确保在删除指向派生类对象的基类指针时，会先调用派生类的析构函数，然后再调用基类的析构函数，从而正确释放派生类所占用的资源。

2.支持多态行为：在使用基类指针指向派生类对象并调用虚函数时，会根据对象的实际类型来调用相应的函数。

》》》》实现思路

通过应用程序是否执行任务（比如是否在windows平台，是否加载了一个dll文件...）来进行条件判断，也就是捕获了这些事件，然后利用这个条件运行某段代码。

》》》》类的继承

在 C++ 中，有三种继承方式：公有继承（public inheritance）、私有继承（private inheritance）和受保护继承（protected inheritance）。
它们的区别在于派生类对基类成员的访问权限。

公有继承（public inheritance）：

语法：使用 public 关键字进行声明，例如 class 派生类 : public 基类 {}。

基类的公有成员在派生类中仍然是公有的。

基类的保护成员在派生类中仍然是保护的。

基类的私有成员在派生类中不可访问。

私有继承（private inheritance）：

语法：使用 private 关键字进行声明，例如 class 派生类 : private 基类 {}。

基类的公有成员在派生类中变为私有的。

基类的保护成员在派生类中变为私有的。

基类的私有成员在派生类中不可访问。

受保护继承（protected inheritance）：

语法：使用 protected 关键字进行声明，例如 class 派生类 : protected 基类 {}。

基类的公有成员在派生类中变为受保护的。

基类的保护成员在派生类中仍然是受保护的。

基类的私有成员在派生类中不可访问。

选择继承方式应根据具体的设计需求和情况来决定。
通常情况下，公有继承是最常用的继承方式，因为它能够使派生类获得基类的接口和功能，并且符合面向对象编程的封装性和多态性原则。
私有继承和受保护继承在特定场景下有其用途，比如实现继承实现细节封装或限制派生类对基类接口的访问。

----------引擎日志----------

》》》》

思路：使用C#风格的库spdlog，将其创建为子模版。将spdlog接口打包，方便使用。
打包后将其设计为宏函数，方便使用。也方便在发行时候通过 #ifdef 来控制一系列宏函数打印的日志不用在发行版本使用。

》》》》子模版？

git submodule add 命令会在主仓库中创建一个指向子模块仓库的链接，并将子模块仓库克隆到指定的目录下。
这个链接存储在主仓库的 .gitmodules 文件中，以便记录和管理子模块的相关信息。

通过将外部依赖库作为子模块添加到主仓库中，你可以保持主仓库和子模块仓库的独立性。
这意味着主仓库和子模块仓库可以分别进行版本控制和更改，而不会相互干扰。
当你在不同的项目中使用相同的外部依赖库时，你只需要在这些项目中添加子模块的链接，而不必重复复制和维护这些外部依赖库的副本。

----------引擎脚本(use Premake API)----------

》》》》思路：

使用premake内置的API接口编写premake.lua脚本文件，使其自动构建特定于平台的项目文件，并自动化的完成Dll文件的复制--替换操作。
同时通过bat文件自动化输入命令启动premake5.exe文件的这个操作。

----------事件系统设计----------

》》》》设计：

四个事件文件
-->AppEvent
Event.h -->KeyEvent
-->MouseEvent

》》》》代码解析：

1.EventType::##type 和 #type 是什么？

EventType::##type 中的 ## 是预处理操作符，用于将宏参数 type 与 EventType:: 连接起来。例如，如果 type 是 Mouse，那么 EventType::##type 就会被展开为 EventType::Mous e。
return #type; 中的 # 是字符串化操作符，将宏参数转换为字符串。如果 type 是 Mouse，那么 #type 就会被展开为 "Mouse"。

2.宏定义 EVENT_CLASS_TYPE(type) 的逻辑是什么？

EVENT_CLASS_TYPE 宏定义了三个函数：GetStaticType、GetEventType 和 GetName。其中 GetStaticType 返回事件的静态类型，即将 type 参数与 EventType:: 连接。GetEvent Type 实际上调用了 GetStaticType 函数。GetName 返回事件对象的名称，即将 type 参数转换为字符串。

3.枚举类型 EventCategory 和位运算的使用是怎样的？

在代码中，使用 #define BIT(x) (1 << x) 定义掩码常量，表示对应位置为 1。而 EventCategory 枚举类型定义了五种事件类别。通过位运算 & 将 category 和该事件对象所属的类别进行比较，判断该事件对象是否包含在指定事件类别中。

```
enum EventCategory
{
    None = 0,
    Mouse = 0b00000001,    // 表示鼠标事件的掩码常量
    Keyboard = 0b00000010, // 表示键盘事件的掩码常量
    Window = 0b00000100     // 表示窗口事件的掩码常量
};
```

假设 GetCategoryFlags() 返回的是 Mouse 类别的掩码常量 0b00000001，
而 category 是另一个掩码常量，例如 Keyboard 类别的掩码常量 0b00000010。
那么当二者进行按位与运算时，结果如下所示：

```
0b00000001  (GetCategoryFlags() 的值，Mouse 类别的掩码常量)
&
0b00000010  (category 的值，Keyboard 类别的掩码常量)
----------
00000000   (结果为 0，表示不属于 Keyboard 类别)
```

因此，结果是一个新的值，其比特位是根据两个操作数的相应比特位进行按位与操作得到的。
在这个例子中，结果为 0，表示不属于键盘事件类别。

4.m_Event.GetEventType() == T::GetStaticType() 的作用是什么？

这段代码是在 EventDispatcher 类中定义的模板函数 Dispatch 中，用于根据事件类型分发事件处理函数。该模板函数可以接受一个函数对象 func，该对象的参数类型为 T&。在函数体内，判断传入的事件处理函数类型是否与当前事件对象的类型匹配。

5.m_Event.m_Handled = func(*(T*)&m_Event) 的作用是什么？

这段代码在 Dispatch 模板函数中，将事件对象转换为指定类型 T 后，调用传入的处理函数 func 来处理事件，并将处理结果存储在 m_Event.m_Handled 中，标记事件是否被处理。*(T*)&m_Event 表示强制将 m_Event 转换为 T 类型的引用，并将其作为参数调用函数对象 func。

6.template<typename T> using EventFn = std::function<bool(T&)>; 和 std::function 是什么？

EventFn 是一个别名模板，定义了一个函数对象类型 std::function<bool(T&)>，表示接受一个参数类型为 T&，返回类型为 bool 的函数对象。std::function 是一个通用的函数封装类，用于封装可调用对象，如函数指针、成员函数指针、Lambda 表达式等。

》》》》enum 和 enum class 的区别
enum class 中的成员在使用时候有类名这个作用域的限制，enum 则没有

》》》》什么是事件分发器
概念：事件分发器（Event Dispatcher）是一种设计模式，用于处理和分发事件（Event）的机制。
包括以下几个要点：
接收事件：事件分发器需要能够接收系统中产生的各种事件，如按键输入、鼠标点击、网络消息等。
分发事件：根据事件的类型和属性，事件分发器将事件分发给注册的事件处理函数或对象。
事件处理：事件处理函数负责对接收到的事件做出相应的处理，可能包括更新系统状态、触发其他操作等。

eg.一个简单的事件分发器的例子是一个图形界面应用程序，当用户点击按钮时，按钮控件会生成一个点击事件，事件分发器接收到该事件后，会将事件分发给注册的按钮点击事件处理函数，
从而执行按钮点击后的相应操作，比如显示弹窗、切换界面等。

》》》》std::to_string() 和 std::stringstream ss 的 ss.str()
std::to_string 函数只接受基本数据类型（例如 int、float 等）作为参数，并将其转换为 std::string 类型的字符串。
因此，直接将 std::stringstream 对象作为参数传递给 std::to_string 函数是不可行的，编译器会报错。

》》》》WindowResizeEvent WRE(1280, 720);
NUT_TRACE(WRE);
为什么能将WRE作为字符型的类型的参数传入NUT_TRACE这个宏中，并让其中的 trace( ) 函数接受WindowResizeEvent类中ToString函数的结果并输出日志?

回答：
在很多日志库中，当你将一个自定义类型的对象传递给日志输出函数时，
它们会通过调用该类型的特定方法（通常称为 ToString() 或类似的方法）来获取对象的字符串表示形式，然后将其输出到日志中。

在你的代码中，NUT_TRACE 宏展开后会调用 logger 对象的 trace 函数，并将传入的参数作为日志消息。
而在 trace 函数内部，由于传入的参数是一个 WindowResizeEvent 对象，
因此会调用 WindowResizeEvent 类中的 ToString() 方法来获取该对象的字符串表示形式。

所以，当你传递 WRE 对象给 NUT_TRACE 宏时，实际上是调用了 WRE.ToString() 方法，该方法返回一个描述 WindowResizeEvent 对象内容的字符串。
然后，这个字符串将被传递给 logger 对象的 trace 函数，并最终输出到日志中。

这种做法的好处是，可以灵活地将自定义类型的对象转换为字符串，并将其记录在日志中。

》》》》子类的构造函数中是否应该调用父类的构造函数?

在 C++ 中，如果子类构造函数没有显式调用父类构造函数，则会自动调用父类的默认构造函数（如果存在）。
以确保从父类继承而来的部分能够正确初始化，保证整个对象的完整性和正确性。

如果父类没有无参的默认构造函数，而只有带参数的构造函数，则子类必须通过初始化列表显式调用父类的构造函数来初始化从父类继承而来的部分。

eg.
```
class Base {
public:
  Base(int value) {
    std::cout << "Base constructor with value: " << value << std::endl;
  }
};

class Derived : public Base {
public:
  // 派生类构造函数没有显式调用基类构造函数
  Derived(int value) {
    // 派生类构造函数体
  }
};
```
在这个例子中，基类 Base 定义了带参数的构造函数 Base(int value)，而派生类 Derived 的构造函数没有显式调用基类构造函数。

接下来，如果我们尝试使用派生类 Derived 来创建对象：
Derived d(5);
派生类构造函数没有显式调用基类构造函数，编译器会自动尝试调用基类的默认构造函数。但是这个基类 Base 并没有默认构造函数，
因此编译器会报错指出找不到默认构造函数来初始化基类的部分。

为解决问题，可以通过初始化列表显式调用基类构造函数来初始化从基类继承来的部分：
```
class Derived : public Base {
public:
  Derived(int value) : Base(value) {
    // 派生类构造函数体
  }
};
```
修正后，我们在派生类的构造函数初始化列表中显式调用了基类的构造函数，并传递了合适的参数来初始化基类的部分。
这样就能够正确地初始化从基类继承而来的部分，避免了编译错误。

----------预编译头文件----------

》》》》理解

在premake中做出的项目设置实际上等同于在VS可视化界面上的设置

pch.h：（Use/Yc）

pch.cpp：（Create/Yc）

----------窗口和GLFW----------

》》》》fork（分支/派生）和submodule（子模块）

Fork= 就像是你复制了一个完整的项目到你自己的账号下，你可以在这个复制的项目上做任何修改而不影响原始项目。

你可以把这个复制的项目当作你自己的项目来管理。

Submodule= 就像是在一个项目中引入了另一个项目，但它们是独立的。主项目知道子项目的存在并能够与之交互，但它们是分开管理的。

子模块通常用于将一个项目作为另一个项目的一部分来使用。

简而言之，Fork 是复制整个项目到你自己的账号下，而 Submodule 是在一个项目中引入另一个项目作为子项目。

cherno的做法是：

1. 在 GitHub 上 Fork 了 glfw 库，获得自己的独立副本。
2. 向库中上传自己的premake文件。
2. 将这个 Fork 的 glfw 库作为子模块引入到自己的项目中，以便在项目中依赖和使用 glfw 库。

》》》》在查证过程中，发现cherno在当时使用的是3.3发布版本的一个开发分支。（https://github.com/TheCherno/glfw/tree/53c8c72c676ca97c10aedfe3d0eb4271c5b23dba）

位于（https://github.com/glfw/glfw/tree/53c8c72c676ca97c10aedfe3d0eb4271c5b23dba）

我选择先Fork最新的glfw，如果有其他情况以后再修正。

》》编译问题参考：

(http://t.csdnimg.cn/hQN5j)

It will help you a lot, believe me.

》》》》lua中的语法

1.IncludeDir = {} 是创建了一个空的 Lua 表（table），用来存储不同模块或库的包含目录。

2.而 IncludeDir["GLFW"] 则是使用了 Lua 中的表索引操作，将键为 "GLFW" 的元素设置为 "Hazel/vendor/GLFW/include"。

3.%{IncludeDir.GLFW}表示要获取表 IncludeDir 中键为 "GLFW" 的元素值

》》》》glfwInit（）

通常情况下，glfwInit() 函数会返回一个整数值来指示初始化是否成功。

》》》》glfwSetWindowUserPointer( )

作用：将一个指向自定义数据的指针与 GLFW 窗口相关联

解释：

通过调用 glfwSetWindowUserPointer(m_Window, &m_Data); 函数，你将自定义数据 m_Data 与 GLFW 窗口 m_Window 相关联。这样做的目的通常是为了在程序中可以方便地访问和操作与该窗口相关的自定义数据。例如，当你需要在 GLFW 窗口回调函数中访问特定窗口的自定义数据时，可以使用 glfwGetWindowUserPointer(m_Window) 来获取该数据指针。

》》》》glfwSetWindowUserPointer 和 glfwGetWindowUserPointer的关系和用法

void glfwSetWindowUserPointer(GLFWwindow* window, void* pointer)

参数：

window：用于设置用户指针数据的窗口对象。

pointer： 想要关联的自定义指针数据（通常是一个结构体指针或其他数据类型的指针。）

功能：将用户自定义的指针数据与特定窗口对象关联起来。方便后续取出使用。

void* glfwGetWindowUserPointer(GLFWwindow* window)

参数：

window：想要获取用户指针数据的窗口对象。

返回值：

与窗口对象关联的，用户指明的 自定义指针数据（即上面关联进来的那个数据或结构体）。

注意：

返回值是一个void *，可以指向任何数据。所以在使用时也许需要你将返回值强制类型转换并赋值给其他变量。

功能：从特定窗口对象中获取之前通过 glfwSetWindowUserPointer 设置的用户自定义指针数据。

eg.

```
// 在初始化窗口时将自定义数据与窗口对象关联
MyData data;
glfwSetWindowUserPointer(window, &data);

// 在需要时从窗口对象中获取自定义数据
MyData* userData = static_cast<MyData*>(glfwGetWindowUserPointer(window));
if (userData) {   // 使用 userData 中的数据    }
```

----------窗口事件----------

》》》》一些涉及到的知识点:

lambda:

(https://www.bilibili.com/video/BV1mw41187Ac/?p=12&vd_source=64ca0934a8f5ef66a21e8d0bddd35f63)

std::placeholders:: 1:

是 C++ 标准库中定义的占位符，用于表示函数对象中的第一个参数。用于等待下次使用时在此占位符位置上填入的值。

（这里的placeholders::1的1好像只是标明占位符的，无其他意义，比如同时使用了两个占位符那第二个占位符就是placeholders::2,

其中数字与其使用时的位置和方法没有关系，仅仅代表占位符的标号）

std::bind:

std::bind 在实际使用中有多种用途。

1.延迟调用和参数绑定

2.改变函数签名

3.成员函数绑定

4.函数适配器

```
eg.
----std::bind 延迟调用一个函数，

#include <functional>
#include <iostream>

void delayedFunction(int a, int b) {
    std::cout << "Delayed function called with arguments: " << a <<
}

int main() {
   auto delayedFunc = std::bind(delayedFunction, 10, 20);
   // 延迟执行 delayedFunction，参数被预先绑定为 10 和 20
   // ...
   // 在需要的时候调用 delayedFunc
   delayedFunc();
   return 0;
}

----改变函数的签名，包括修改函数的参数类型或个数。
    #include <functional>
    #include <iostream>
    void originalFunction(int a, int b) {
        std::cout << "Original function called with arguments: " <<
    }

void modifiedFunction(double x, double y) {
    std::cout << "Modified function called with arguments: " << x <
}
int main() {
  // 使用 std::bind 将 modifiedFunction 的签名修改为接受两个 double 类
 auto modifiedFunc = std::bind(modifiedFunction, std::placeholders:
  // 在需要的时候调用 modifiedFunc，并传入合适类型的参数
  modifiedFunc(3.14, 2.71);      // 输出：Modified function called
  return 0;
}

----绑定类的成员函数，并指定对象实例作为第一个参数。
#include <functional>
#include <iostream>

class MyClass {
public:
    void memberFunction(int value) {
        std::cout << "Member function called with value: " << value
    }
};

int main() {
    MyClass obj;
    auto memberFunc = std::bind(&MyClass::memberFunction, &obj, std
    // 绑定 MyClass 的成员函数 memberFunction，并将 obj 作为对象实例
    // ...
    // 在需要的时候调用 memberFunc
    memberFunc(42);

   return 0;
}


----使用函数适配器进行参数绑定：
    #include <functional>
    #include <iostream>

void printSum(int a, int b) {
  std::cout << "Sum: " << a + b << std::endl;
}
int main() {
   auto sumFunc = std::bind(printSum, std::placeholders::_1, 5);
   // 将第二个参数绑定为 5，等待传入第一个参数
   // ...

  // 在需要的时候调用 sumFunc
   sumFunc(10); // 输出 Sum: 15

   return 0;
}
```

----------接下来我以发问的方式来查证疑惑（这都是我在学习时产生的疑惑）----------

》》》问题：Application.cpp中的语句m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent));在干什么？

在WindowsWindow.h中，有 inline void SetEventCallback(const EventCallbackFn& callback) override { m_Data.EventCallback = callback; } 这样的定义。
所以 SetEventCallback 这个函数需要接受一个 EventCallbackFn 类型的函数，也就是 void XXX(Event& e) 这样的函数。
而std::bind恰好能返回一组函数指针或者说一个函数对象，通过这个函数对象，我们可以用传入的 OnEvent 这个函数初始化 m_Data.EventCallback
（注意：在将成员函数作为函数对象传递时，需要绑定其对象，确保能通过对象正确的访问到这个成员函数）

而 BIND_EVENT_FN(OnEvent) 就像是对 OnEvent 做了一些暂缓的设置，以便之后处理
（我们后面会谈到）

》》关于函数指针：
https://www.bilibili.com/video/BV1254y1h7Ha/?vd_source=64ca0934a8f5ef66a21e8d0bddd35f63

》》》问题：占位符呢？
虽然有占位符的设计，但是m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent));这个OnEvent却没有填入参数
即使在绑定时没有显式地填入参数，但通过占位符的机制，函数对象仍然能够正确地接收事件参数并传递给 OnEvent 函数。
（注意是 std::placeholders::_1 而不是 std::placeholders::1， 有下划线）

理解：
通过使用占位符，函数对象会暂时（注意：暂时）保留一个位置用于接收后续传入的参数，并在调用时将这些参数正确地传递给被绑定的成员函数。

》》》m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent)); 是什么意思？
BIND_EVENT_FN(OnEvent)，代表了什么意思？
随后的data.EventCallback(event); 和以上有什么联系，为什么这样使用？

问题一：
首先我们在前面提到，m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent)); 其实是 std::bind( ) 返回了一个函数对象作为 SetEventCallback 的参数，
这用来初始化 data 中的一个元素 EventCallback 。

问题二：那么 BIND_EVENT_FN(OnEvent) 呢？

解释：在定义中我们看到 #define BIND_EVENT_FN(x) std::bind(&Application::x, this, std::placeholders::_1)
意思是为成员函数绑定了对象，并将其作为函数对象传递，这就是前两个参数的意义，
第三个参数：std::placeholders::_1 ， 指出了 OnEvent 的参数暂时被占位了，可以先不填入参数，以便之后处理。

问题三：
之后处理，实际上就是指之后的 data.EventCallback(event); 要进行的处理
通过 Data 类型的对象 data，我们调用出来了刚才初始化进 data 的那个函数：OnEvent。
（我们在之后通过 glfwSetWindowUserPointer 和 glfwGetWindowUserPointer 获取了 m_Data,并将其复制到一个名为data的引用上：WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window); ）
调用出来的 OnEvent( ) 就相当于 data.EventCallback( )，然而 OnEvent 在定义上是需要参数的，所以
data.EventCallback(event) == OnEvent(event) ,这个 event ，就是我们用占位符延缓的参数（这个参数被标明会在后续使用）

在使用 Event 对象作为 OnEvent 的参数填入之后，event这个参数参与到 OnEvent 函数体内的操作中去，完成我们定义的操作。

（在回调函数中我们这样使用：
```
    WindowResizeEvent event(width, height);
    data.EventCallback(event);
)
```

》》》》关于data的使用理解：

```
glfwSetWindowSizeCallback(m_Window, [](GLFWwindow* window, int width, int height)
{
    WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);
    data.Width = width;
    data.Height = height;

    WindowResizeEvent event(width, height);
    data.EventCallback(event);
});
```

逻辑：WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window); 的作用是从 GLFW 窗口中获取用户指针，并将其转换为 WindowData 类型的引用，从而可以访问窗口相关的数据。
（如果在之前的代码中将 m_Data 设置为窗口的用户指针（通过 glfwSetWindowUserPointer 函数），那么在这个回调函数中获取到的 data 就是之前声明的 m_Data。）

概念：通过 glfwSetWindowUserPointer 来将自定义数据与 GLFW 窗口关联起来，然后在回调函数中使用这些数据。
（这里的 WindowData& data 是对用户指针指向的 WindowData 结构体的 引用（注意：引用），因此对 data 的操作实际上是对窗口关联的数据进行修改或访问。）

深入理解：
（是否反复声明data对象？）
1.每次调用 glfwSetWindowSizeCallback 或 glfwSetWindowCloseCallback 时会重新获取窗口关联的 WindowData 数据，所以不是每次都重新声明 data，
而是获取同一个窗口关联的数据，传入并刷新。
（是否在更新同一个data对象中的值？）
2.是的，多次调用 glfwSetWindowSizeCallback 或 glfwSetWindowCloseCallback 绑定了不同的事件处理逻辑，但是它们都共享同一个 data，每次调用回调时 data 中的值会被修改，因为它们都是指向同一个 WindowData 数据结构的引用。
（是否使用的是私有变量m_Data？）
3.是的，通过 glfwSetWindowUserPointer(m_Window, &m_Data) 将 m_Data 绑定到 GLFW 窗口对象 m_Window 上。而在 glfwSetWindowSizeCallback 的 回调函数中，通过 glfwGetWindowUserPointer(window) 获取绑定在窗口上的 m_Data 结构体的指针，并将其转换为 WindowData& data 引用。
因此，回调函数中的 data 是直接引用并操作了 m_Data 结构体，而不是新声明的结构体。

》》》》为什么在 glfwSetWindowSizeCallback 中要执行 data.EventCallback(event) 这样的操作?

流程：

当窗口大小变化（或是触发某一操作对应的回调函数）时，GLFW 提供的回调函数 glfwSetWindowSizeCallback 会被自动触发，
然后根据我们对该回调函数的定义（定义包含在我们填入的lambda表达式或者函数指针中），在 glfwSetWindowSizeCallback 被自动调用时，
我们会执行到创建相应的事件对象 Event，然后调用之前在 Data 的EventCallback中存入的函数（ EventCallback 所指向的函数 OnEvent），
并将 event 作为参数传递给这个函数。
而这个函数我们是在 m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent)); 这里初始化给Data 的。

效果/目的：

为了确保在特定事件发生时能够调用已经设置好的事件回调函数

》》》》OnEvent为什么要被这样设置？为甚么在每一个回调函数之后都要写一次?

对每一个回调函数实际上都有调用 data.EventCallback(event) ；这用来调用存入data结构体的OnEvent函数，然而在OnEvent中，你可以传入任何事件，
但是只有当事件为WindowClose时候，有对应的处理方式：

dispatcher.Dispatch<WindowCloseEvent>(BIND_EVENT_FN(OnWindowClose)); 这用来实现对窗口关闭时要执行的操作，

但是 HZ_CORE_TRACE("{0}", e); 则是每个回调函数在 data.EventCallback(event); 时都会调用到的语句，是每一个回调函数都能触发的记录的操作。

》》》但是 OnEvent 仅仅只是关于 WindowClose 有对应的操作设计，为什么在 Cherno 的视频中其他的回调函数依旧能正常运行并相应?

1.

```
glfwSetWindowSizeCallback(m_Window, [](GLFWwindow* window, int width, int height)
{
    WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);
    data.Width = width;
    data.Height = height;

    WindowResizeEvent event(width, height);
    data.EventCallback(event);
});
```

中关键的是

```
data.Width = width;
data.Height = height;
```

data是对M_Data引用，因此Width和Height发生更改时，实际上在glfwCreateWindow这里，窗口就会因为参数的变化而让窗口变化

m_Window = glfwCreateWindow((int)props.Width, (int)props.Height, m_Data.Title.c_str(), nullptr, nullptr);

但关于为什么这里Cherno设置是 props.Width 而不是 m_Data.Width，我不很理解

2.

```
glfwSetWindowCloseCallback(m_Window, [](GLFWwindow* window)
{
    WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);
    WindowCloseEvent event;
    data.EventCallback(event);
});
```

当然像我们上面所说的，通过 data.EventCallback(event); 进入 OnEvent 中时，

成功满足了 dispatcher.Dispatch<WindowCloseEvent>(BIND_EVENT_FN(OnWindowClose));

故成功关闭

3.

- glfwSetKeyCallback(m_Window, [](GLFWwindow* window, int key, int scancode, int action, int mods)
- glfwSetScrollCallback(m_Window, [](GLFWwindow* window, double xOffset, double yOffset)
- glfwSetCursorPosCallback(m_Window, [](GLFWwindow* window, double xPos, double yPos)

这几个函数，现在并没有去设置按下时应该触发事件，这并不影响当前的操作，

我猜之后可能会在OnEvent中继续续写需要对应执行的一些函数，让事件分发器继续起作用。但现在并没有写。

关于 HZ_CORE_TRACE 能够响应对应键位按下的日志追踪，是因为诸如

- KeyPressedEvent event(key, 0);
- MouseButtonPressedEvent event(button);
- MouseScrolledEvent event((float)xOffset, (float)yOffset);
- MouseMovedEvent event((float)xPos, (float)yPos);

都是从程序自动反复调用回调函数时，从其中的参数中获取了数据，声明了对应的 EventCallback 对象（通过构造函数传入数据）

然后使用 data 的 EventCallback 中存入的 OnEvent，这里有 HZ_CORE_TRACE

OnEvent 接受了这个 Event 对象，然后通过 OnString 成功的获取数据并且打印出来了

（关于为什么会调用了OnString，请看 3game engine 中的这个问题描述：

》》》》WindowResizeEvent WRE(1280, 720);

NUT_TRACE(WRE);

为什么能将WRE作为字符型的类型的参数传入NUT_TRACE这个宏中，并让其中的 trace( ) 函数接受WindowResizeEvent类中ToString函数的结果并输出日志？）

》》》》回调函数的定义结构：理解

```
glfwSetWindowSizeCallback(m_Window,
    [](GLFWwindow* window, int width, int height)
    {
        WindowData& data = *(WindowData*) glfwGetWindowUserPointer(window);
        data.Width = width;
        data.Height = height;

        WindowResizeEvent event(width, height);
        data.EventCallback(event);
    }
);                                  //lambda表达式作为第二个参数
```

使用位置：通常在实际使用中，会在主函数（包括渲染循环）之前，初始化GLFW窗口之后进行回调函数的定义。

接受参数：一般回调函数接受两个参数，1.窗口对象，2.一个函数。

逻辑流程：在特定的事件比如窗口大小发生变化时候，回调函数会自动的监测到操作，并且获取数据。

（具体的说，应该是 GLFW 负责传递相应的数据给回调函数，然后回调函数再将这些数据传递给用户定义的处理函数。）

随后便会自动的去调用用户自己传入的函数，因为回调函数本身并不会进行任何操作，这些都需要用户自己定义。

提示：这个函数可以是在某处定义的，然后传入这个函数指针，或者也可以是一个lambda表达式（在作为参数的时候，可以就地定义的函数）

》》》》对于一个数据 double pos，使用 (float)pos 和 float(pos) 这两种方式的类型转换有什么不同

1.(float)pos 是一种 C 风格的类型转换方式。这种方式在 C++ 中仍然有效，但不够安全，因为它可以进行任意类型的转换，包括隐式转换和强制转换，可能会导致潜在的错误。
2.float(pos) 是一种 C++ 风格的类型转换方式，称为函数风格的强制类型转换（functional cast）。这种方式在 C++ 中更为推荐，因为它提供了更明确的类型转换操作，同时在某些情况下还能提供更好的类型安全性。
（会有警告但不会影响正常运行）

》》》》关于事件分发器Dispatcher的语法分析：
1. using EventFn = std::function<bool(T&)>;
声明了 EventFn 作为一个类型，这个类型代表一个接受 T 类型参数并返回 Bool 类型的函数，所以在我们后续使用时候，填入的 WindowCloseEvent

2.
```
EventDispatcher(Event& event)
             :m_Event(event)  {}
```

首先初始化一个对象

3,
```
template<typename T>
  bool Dispatch(EventFn<T> func) {                              // (这里将会在未来使用中填入一个函数指针)
      if (m_Event.GetEventType() == T::GetStaticType()) {       //!!! 静态函数在使用时需要使用类名或类型名来调用 (T::)
          m_Event.m_Handled = func(*(T*)&m_Event);              //*(T*) 表示：用 * 解引用 (T*) 所声明的T类型指针，实现强制类型转换
          return true;  }
      return false;
  }
```

后续我们会这样使用：

```
Event& e;         //作为  OnEvent  的一个参数传入的
EventDispatcher dispatcher(e);
dispatcher.Dispatch<WindowCloseEvent>(BIND_EVENT_FN(OnWindowClose));
```

思路：
首先用 e 初始化了一个对象，然后通过 dispatcher 的成员函数 Dispatch 将一个接收 T 类型的返回 bool 类的函数作为对象填入
（这个函数是 OnWindowClose ，只不过用 std::bind 将其作为函数对象传入，因为我们将延迟一些操作，所以不使用 std::functional）

然后，我们标明了 template<typename T> 为 WindowCloseEvent ，于是我们进行判断：
如果通过 OnEvent 传入的 Event e 与我们标明的 typename T 一致，
则说明这个事件与我们想要允许调用 BIND_EVENT_FN(OnWindowClose) 的事件是一致的
这就是：if (m_Event.GetEventType() == T::GetStaticType()) 的作用。

进入条件判断语句内部之后，m_Handled 被赋予了我们填入的函数的返回值，否则返回 false，即不允许该事件类型执行此函数

》》那么那一句是他执行我们定义的函数的语句呢？

首先我们 dispatcher.Dispatch<WindowCloseEvent>(BIND_EVENT_FN(OnWindowClose));
将函数传入，但是通过 std::bind 我们使用一个占位符延缓了参数填入的时间，在 Dispatch 中，
在 if (m_Event.GetEventType() == T::GetStaticType()) 的条件下使用了 m_Event.m_Handled = func(*(T*)&m_Event);
func(*(T*)&m_Event) 便是对填入的自定义函数的调用，同时将 m_Event 强制转换为允许的类型，在此处调用函数

》那为什么在赋值的过程中还可以进行函数中的操作呢？？
流程：
1.当调用一个函数时，函数体内的语句会被按顺序执行。
2.函数可以有返回语句或者没有返回语句。如果没有返回语句，则函数会自动返回一个默认值（对于 bool 类型，未显式返回的函数会返回 false）。
3.当函数执行完所有语句后，会将返回值传递给调用者。

----------layers----------

》》》》LayerStack.cpp中的语句大致意思：
LayerStack::LayerStack()：
　　初始化了 m_LayerInsert，它是一个迭代器，用于指示下一个被插入图层的位置。初始时，它指向 m_Layers 的开始位置。
LayerStack::~LayerStack()：
　　释放所有图层的内存。它遍历 m_Layers 并使用 delete 删除每个图层的指针，确保不会发生内存泄漏。

LayerStack::PushLayer(Layer* layer)：
　　将一个 “普通图层” 推入图层列表的前半部分。使用 m_Layers.emplace 在 m_LayerInsert 位置之前插入新图层，然后更新 m_LayerInsert 指向新的位置。
LayerStack::PushOverlay(Layer* overlay)：
　　将一个 “覆盖图层” 推入图层列表的后半部分。使用 m_Layers.emplace_back 在 m_Layers 的末尾添加新图层。

LayerStack::PopLayer(Layer* layer) 和
LayerStack::PopOverlay(Layer* overlay)：
　　从图层列表中删除指定的图层。它们使用 std::find 在 m_Layers 中找到要删除的图层，并使用 erase 函数从列表中移除。在删除图层后，m_LayerInsert 更新为指向正确的位置。

》》》》关于“普通图层”和“覆盖图层”的理解
在CS:GO游戏中
普通图层：一般是游戏场景和玩家角色，它们包含了游戏世界的内容以及玩家的交互。
覆盖图层：一般是设置菜单、商店界面等UI，它们会覆盖在游戏场景上方，用于显示各种菜单、选项、提示等用户界面元素。

》》》》层的传入顺序

层的设置：

层这个数组整体分为两部分：前半部分为普通图层，后半部分为覆盖图层。

就绪后从头到尾开始绘制。（layer3, layer2, layer1, overlay1, overlay2, overlay3）

层的就绪：

上述的函数：

```
void LayerStack::PushLayer(Layer* layer) {
    m_LayerInsert = m_Layers.emplace(m_LayerInsert, layer);
    //此操作不仅将 layer 插入到指定位置，还会将返回的迭代器赋值给 m_LayerInsert，
    以便下次插入时插入在这个位置，并将此前这个位置的元素后移一位。 }

void LayerStack::PushOverlay(Layer* overlay) {
    m_Layers.emplace_back(overlay); }
```

解释与结果：

调用 PushLayer 推入普通图层到数组前半部分时，新图层会被插入到 m_LayerInsert 迭代器所指向的位置之前。因此，如果 m_LayerInsert 最初指向 layer1，那么插入 layer1 应该是这样的顺序：layer3, layer2, layer1。

调用 PushOverlay 函数将覆盖图层推入图层列表的后半部分时，新图层会被添加到列表的末尾。因此，使用overlay1, overlay2, overlay3 分别表示推入的覆盖图层应该是这样的顺序：overlay1, overlay2, overlay3。

》》》运行流程（以CSGO举例的话）

1.图层将会是

（layer3, layer2, layer1, overlay1, overlay2, overlay3）

其中普通图层先后由 layer1：游戏UI、layer2：游戏角色、layer3：游戏背景

覆盖图层先后由 overlayer1：菜单、overlayer2：设置、overlayer3：设置中的一个选项

层栈结构：

overlay3　某一设置选项图层

overlay2　设置图层

overlay1　菜单图层

layer1 游戏中的UI（击杀敌人时显示的图标）

layer2 游戏角色

layer3 游戏背景

2.事件将会是：

从最后的图层开始处理。（要对最顶层的界面进行交互，退出后才能对低一层的界面操作，否则不符合直觉和视觉。）

所以以一直在进行交互的这个图层永远是最顶层的图层。

》》拓展：

（即便是在一个图层上新生成一个图层，也是由此前作为最顶层的图层做处理，由他生成一个图层，

然后该屈居较低一层，使新图层作为最顶层，以便进行操作）

》》事件反向处理的原因：

1.符合人的操作习惯：先处理最上层的图层可以更快地响应用户的操作。

2.符合人的视觉习惯：保持游戏画面的逻辑性，避免混乱和不连贯的情况出现。

》》》》层栈结构的理解

层栈结构：

overlay3　某一设置选项图层

overlay2　设置图层

overlay1　菜单图层

layer1 游戏中的UI（击杀敌人时显示的图标）

layer2 游戏角色

layer3 游戏背景

问题发现与分析：

1.层栈只是一个理想上的栈结构，实际上Cherno只设置了一个 vector 来存储图层，所以这个图层结构是

layer3, layer2, layer1, overlay1, overlay2, overlay3 躺地上的。

！！！！

2.因为在 pushlayer 函数中我发现emplace函数会将元素插入到当前位置之前，

所以我一直在考虑传入普通图层的时候是否要特定顺序，比如先传入较高层次的图层？

但是覆盖图层却是先传入较低层次的图层（这和两个push函数传入元素的方向有关系）

2.另外我注意到Cherno所说，“我们将覆盖图层放在列表最后，我们总是希望他最后渲染”，所以我猜测在这里，Cherno决定的渲染顺序是

直接从栈底（vector头部）开始向上（vector尾部）处理

最终总结：

1.事件处理：从栈顶到栈底（从vector尾部到头部）

2.渲染处理：

第一种（当普通图层先传入较高层次，覆盖图层先传入较低层次情况下）

overlay3　某一设置选项图层

overlay2　设置图层

overlay1　菜单图层　↑

layer1 游戏中的UI（击杀敌人时显示的图标）　↓

layer2 游戏角色

layer3 游戏背景

渲染方向：从栈底到栈顶（从vector头部到尾部）

第二种（当普通图层和覆盖图层都是先传入较低层次时）

overlay3　某一设置选项图层

overlay2　设置图层

overlay1　菜单图层　↑

layer1 游戏背景 ↓
layer2 游戏角色
layer3 游戏中的UI（击杀敌人时显示的图标）
渲染方向：普通图层部分从顶到底，之后覆盖图层从底到顶

不知道我的分析是否正确，但以此看来两种方法各有裨益。
还是要根据Cherno后续的操作来分析

》》》》emplace函数实际情况
1.循环通过 emplace 向 vector 中传入元素时，会将新元素放在先前元素之前，并且返回一个指向最新的元素的指针

layer3 layer2 layer1
↑
指针

2.越晚传入的元素地址越大

layer3 layer2 layer1
指针地址：3 2 1

》》》》pop函数中Insert--的作用？？？？？？？？，判断条件可以用来删除除栈顶元素以外的其他元素吗
在我反复观看代码，并且理解层栈结构之后，我认为：
前提：
在推入三个layer之后，Insert是在Layer3这里指着的。
1.递减的设计思考：
虽然 Insert-- 确实会将指针指向下一个元素，但是这完全建立在删除的元素一定是最晚传入（栈顶元素）的基础上。
如果删除栈顶元素， Insert 由栈顶被移到下一个元素上，并且这下一个元素接替栈顶的位置。
否则 Insert-- 在删除其他元素时是完全没有其他意义的。
2.判断条件的思考：
2.那既然已经这样设计了，那Insert--就只能在删除固定的、明确的、栈顶元素情况下使用了
所以，这前面的判断条件便仅仅为了确保所要删除的元素存在 vector 中
并不存在用 std::find 去寻找除栈顶以外的元素并将其删除的思路了。

结论：
在层栈中，PopLayer函数用于删除普通图层这一部分的栈顶，
1.Insert--的作用是将指向头部的指针位置移向下一个充当栈顶的元素这里
2.判断条件式确保删除操作的安全性，避免删除其他数据。并无其他作用。

对于PopOverlay来讲，并没有提供 Insert 这一指针（实际上是迭代器类型，可以这样理解）
所以只需传入 overlay 的栈顶元素，直接调用erase即可。

》》》》erase 函数的参数及其用法
1.erase(iterator pos)：
删除指定位置的元素。pos是一个指向待删除元素的迭代器。
2.erase(iterator first, iterator last)：
删除指定范围内的元素。first和last是表示范围的迭代器，删除的元素包括first指向的元素，但不包括last指向的元素。

》》》》layer stack中的
std::vector<Layer*>::iterator begin() { return m_Layers.begin(); }
std::vector<Layer*>::iterator end() { return m_Layers.end(); }
为什么需要这两个函数?

1.首先要了解基于范围的 for 循环这个概念：
概念：基于范围的 for 循环也称 for each 循环，是一种简化遍历容器元素的语法。
使用要则：允许使用者直接遍历容器中的元素，而不必使用迭代器和循环索引。
2.现在了解这两个函数的作用;
因为在 for each 循环中，会使用容器确定整个 for each 循环的范围，这就要求使用的容器具有 begin() 和 end() 成员函数来返回迭代器。
以便正确遍历容器中的元素。

》》》》一个错误分析
```
for (auto it = m_LayerStack.end(); it != m_LayerStack.begin(); )
{
    (*--it)->OnEvent(e);
    if (e.Handled)
        break;
}
```

这里的 (*--it) 可以写为 *(--it) 吗?
理论上来讲，这二者并没有区别，但是在实际使用中可能会由于优先级的问题发生问题。
所以要么 (*--it)->OnEvent(e); 要么 (*(--it))->OnEvent(e);

》》》》sandbox 中的函数和 项目Nut (Hazel) 有什么关系，这些函数是怎样能够影响到 application 中的函数的?
他们是怎样传递的?

1.在代码中，Sandbox 类的构造函数
```
Sandbox() {
    PushLayer(new ExampleLayer());
```

}

通过 PushLayer ( ) 创建 ExampleLayer 图层对象并将其添加到 LayerStack 中，
（而且 PushLayer 函数是 Nut 项目中 LayerStack 类中的一个函数）

2.在项目 Sandbox 中，ExampleLayer 这个类继承自 Nut::Layer，并且重写了 OnUpdate 和 OnEvent 函数。
这意味着当你创建 ExampleLayer 对象并将其添加到 LayerStack 中后，这些重写的函数会在你对 ExampleLayer 对象进行操作时被"对应的"调用
（因为在 ExampleLayer 中，这几个函数被重写了，并且作为 ExampleLayer 这个类的成员函数）

比如在 application.cpp 中，(*--iter)->OnEvent(e); 就是自动辨别 iter 的类型，然后自动的使用了这个类下的成员函数 OnEvent

- ---

所以，在 Application 类的 Run 函数中，你遍历 m_LayerStack 并调用其每个图层的 OnUpdate 函数。
但是由于 ExampleLayer 是 Layer 的子类，所以当你对刚才传入的 LayerStack 的 ExampleLayer 类型的对象进行操作时，ExampleLayer 中重写的 OnUpdate 函数会被调用。

- ---

同样，在 Application 类的 OnEvent 函数中，你也遍历 m_LayerStack 并调用每个图层的 OnEvent 函数。
由于 ExampleLayer 也重写了 OnEvent 函数，所以在这里也会调用 ExampleLayer 中重写的 OnEvent 函数。

----------OpenGL & Glad----------

》》》》int status = gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);是在干嘛?
作用：
使用 glfwGetProcAddress 获取当前环境下的 OpenGL 函数的地址，并通过 gladLoadGLLoader 将这些函数指针加载到程序中，从而使得程序可以调用 OpenGL 提供的各种函数进行图形渲染等操作。

之前都在使用glfw，经此之后可以使用gl的函数，以此获取用来进行图形渲染的函数。

》》》》一个不同：
Cherno在程序中为了避免 glad.h 和 glfw3.h 包含两个 gl.h（好像是这个问题？）导致的错误
选择加入一个宏定义以确保 gl.h 不会被包含两次。

但是我记得在之前学习图形渲染的时候，有一个方法也行，就是先包含 glad.h：
#include <glad/glad.h>
#include <GLFW/glfw3.h>
于是我在WindowsWindow.h中按照这样的方式同时包含两个文件，并在 WindowsWindow.cpp 中删除 glad.h。这样也是可以正常运行的。
参考：（https://learnopengl-cn.github.io/01%20Getting%20started/03%20Hello%20Window/）

----------ImGui----------

》》》》因为真的很想复刻Cherno的操作（而且对于最新版本的imgui我也不是很理解其文件构架）
所以我决定将Cherno当时使用的imgui版本作为一个库，然后使用。
（http://t.csdnimg.cn/SRD0V）
这是我的方法，如果需要可以查看

》》》》什么是单例模式?
概念：
单例模式是一种设计模式，用于确保类只有一个实例，并提供一个全局访问点来访问该实例。

在以下情况下，可以考虑使用单例模式：
1.全局访问点：
当需要在整个应用程序中共享某个对象实例时，单例模式可以提供一个全局访问点，使得任何地方都可以方便地获取到这个实例。
2.资源共享：
在需要共享资源的情况下，比如数据库连接池、日志文件等，可以使用单例模式确保资源的唯一性和合理的管理。
3.控制实例个数：
有些情况下，系统中某个类只能有一个实例，比如线程池、缓存、配置文件等，这时可以使用单例模式来限制实例个数。
4.节省内存：
有些对象占用大量内存，频繁创建销毁会带来性能问题，使用单例模式可以避免重复创建实例，节省内存空间。
5.全局状态管理：
在需要维护全局状态的场景下，比如全局配置信息、用户登录信息等，单例模式可以提供一个统一的状态访问接口。

》》》》单例模式的讲解：
（https://www.bilibili.com/video/BV1bR4y177Hp/?spm_id_from=333.999.0.0&vd_source=64ca0934a8f5ef66a21e8d0bddd35f63）

为什么在代码这里需要使用单例模式呢?
因为我们在游戏引擎中只会需要一个Applicaiton，所以我们使用了单例模式。

----------ImGui事件----------

》》》》鼠标事件函数为什么最后一句是return false;事件消费是什么?
为什么 ImGuiLayer::OnMouseButtonPressedEvent 最后需要返回 false ?

当处理事件时，返回 false 通常表示事件没有被“消费”，即并未完全处理。在这种情况下，我们可能希望其他地方也有机会继续处理这个事件。如果函数返回 true，则表示事件已经被处理了，不需要进一步传播。
因此，在这段代码中，返回 false 是为了让其他地方也有机会处理鼠标按键按下事件。

举例：
eg1:
有时候，我们会选择长按某一按键达成某种操作。这就要求我们在一帧结束之后，继续询问并处理该事件。
否则想达成长按按键时，却会因为事件被提前“消费”而中断操作。
eg2:
在打开的商城页面中，在点下商品之后，此图层并不会消失，而是等待另一个按钮“购买”被点击后该图层才会消失。
1.点击商品时，整体事件并没有处理完
2.为了购买按钮的事件能够触发，我们将前一个事件标记为未处理完成，将其进一步传播
3,直到购买按钮被触发，整个事件完成

》》》》glViewport的参数

概念： 设置视口（Viewport），用来指定 OpenGL 渲染的目标区域在帧缓冲区中的位置和大小。
原型： void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
参数：
x：指定视口的左下角 X 坐标。
y：指定视口的左下角 Y 坐标。
width：指定视口的宽度。
height：指定视口的高度。

》》》》io.KeyCtrl = io.KeysDown[GLFW_KEY_RIGHT_CONTROL] || io.KeysDown[GLFW_KEY_LEFT_CONTROL];
的逻辑是什么？

如果任一 Ctrl 键被按下，则 io.KeysDown[GLFW_KEY_？？_CONTROL] 的值为 true，否则为 false。
左右 Ctrl 键的按下状态进行逻辑或运算，最终将结果赋值给 io.KeyCtrl ，表示用户是否按下了任意一个 Ctrl 键。

》》》》关于WindowsWindow.cpp中的回调函数和ImGuiLayer.cpp中的OnKeyTypedEvent的关系
为什么这两个函数有所联系？？？

```
glfwSetCharCallback( m_Window, [](GLFWwindow* window, unsigned int keycode)
    {
      WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);

      KeyTypedEvent event(keycode);
      data.EventCallback(event);
    }
);

bool ImGuiLayer::OnKeyTypedEvent(KeyTypedEvent& e)
{
    ImGuiIO& io = ImGui::GetIO();
    int keycode = e.GetKeyCode();
    if (keycode > 0 && keycode < 0x10000)
        io.AddInputCharacter((unsigned short)keycode);

    return false;
}
```

因为在回调函数glfeSetCharCallback中有通过回调函数获取的keycode构建KeyTypedEvent对象的过程 -->即 KeyTypedEvent event(keycode);
此时我们就获取了键盘上输入的关键字，而且存入了这个事件对象中，然后在之后的ImGuiLayer上的demo窗口中，我们使用int keycode = e.GetKeyCode();获取了Keycode，
所以相当于我们从回调函数中获取了keycode这个数据，然后在demo窗口中我们就能够使用它。

----------Github & Repo----------
》》》》

----------Pull requests----------

》》》》startproject "Sandbox"，为什么 Error: unable to set startproject in project scope, should be workspace
startproject 命令应该在工作区的作用域内. 要放置在 Premake 脚本的顶层.

```
eg:
workspace "Work_space"
    configurations { "Debug", "Release" }
    ......

    startproject "Sandbox"  -- 将启动项目设置为 "Sandbox"

project "Sandbox"
    kind "ConsoleApp"
    language "C++"
    ......
```

----------polling input轮询输入----------

》》》》独立窗口
概念：
一个在操作系统中独立存在的、可以单独打开、关闭和移动的窗口。
特点：
可以被拖动到屏幕上的任何位置
可以被最小化、最大化和关闭
可以独立于其他窗口存在

》》》》静态类

概念：

一个类

1.只包含静态成员（静态属性、静态方法等）

2.不能被实例化

3.也不能继承其他类

用途：

作为工具类：类中包含一组静态方法，二这些方法能够提供一些通用的功能（计算、日期、操作...）

在使用这些方法时，我们不需要实例化该类就能使用其中的静态成员函数。

全局访问点：静态类中的静态成员可以被全局访问，以供整个程序使用这些方法或者属性。

常量集合： 静态类可以用于存储常量，在需要时可以通过类名直接进行访问

举例：

单例模式就是一种静态类的例子

》》》》在GetMousePosImpl中，返回了{ sth_about_std::pair<> }

而在GetMouseX中，为何使用auto [x,y] 来接受参数?

原因：

这其中涉及到了结构化绑定。

什么是结构化绑定?

概念:

结构化绑定（Structured Binding）是 C++17 中引入的一个特性，

它允许将一个复合类型（如 pair、tuple、数组等）的成员解构为单独的变量。

语法：

auto [var1, var2, ...] = expression;

参数：

var1, var2, ... 是要绑定的变量名，用逗号分隔。

expression（表达式） 是返回复合类型的表达式，可以是函数返回值或其他包含多个值的表达式。

作用：

这样可以方便地从复合类型中提取各个成员，并将它们赋值给单独的变量。

要求/规范：

（只能对以下类型的对象使用）：

标准库中的 tuple 类型：可以将 tuple 的各个元素解构为单独的变量。

标准库中的 pair 类型：可以将 pair 的两个成员解构为单独的变量。

数组：可以将数组的各个元素解构为单独的变量。

eg.

在 auto [x, y] = GetMousePosImpl(); 中，GetMousePosImpl() 返回一个 std::pair<float, float> 类型的对象

而通过结构化绑定 auto [x, y]，这个对象被解构为两个单独的变量 x 和 y，可以被单独使用。

----------Keycodes & MouseButtonCodes（键盘和鼠标代码）----------

》》》》

没什么要记的

----------数学库----------

》》》》glm/glm/gtc 和 glm/glm/ext 的区别

注意到 github 上最新的例子引用的头文件是来自 ext 中的头文件，而glm 9.9.0 版本示例使用了 gtc 中的文件，故发问。

gtc：

包含一组便利性函数和工具，用于扩展 GLM 的功能（矩阵变换、投影等）提供许多常用的数学运算和变换函数，方便开发者快速实现各种数学操作。

ext:

包含了 GLM 的扩展功能（一些实验性质的函数或者功能更为专业化的内容）可能还在开发中或者不太稳定。

GameEngine2

2024年5月25日 16:31

----------ImGui docking & viewport 停靠和视口----------

》》》》我所做的：

选择要使用的版本：
Cherno 使用的应该是1.67或者1.68版本中正在开发的、还未合并至主分支的 docking 中的代码。
做法：
所以我从 ImGui v1.68 版本拉取了 docking 分支，将其作为个人库的一个 docking 分支（之前将 v1.66b 拉取到了个人库中）
然后在 docking 分支中上传对应的 premake5.lua 文件，以备使用。

报错：
对了，在子模块根目录下一开始 git checkout docking 时候，会报错（error: pathspec 'docking' did not match any file(s) known to git）
此时 git branch 查看分支状况时候发现只有一个分支，这表明子模块还未更新
解决方法：
需要运行 git pull 拉取一下，然后可以查看到 docking 分支（我的电脑上是这样的，虽然git pull 之后提示 Already up to date.）

》》注意：
由于我之前并没有定义 GLFW_INCLUDE_NONE,也没有像 Cherno 一样将 glad.h 和 GLFW.h 按照一定顺序包含
所以会导致一些错误，现已修复。 可以参考：（http://t.csdnimg.cn/ogOD3）

》》》》关于库，分支的问题：

》》》一般情况下，一个库的不同分支中的文件一样吗?
答：
一个库的不同分支中的文件可能会有一些差异。
不同分支可能会有针对不同功能需求的变化，比如新添加的特性或优化，这导致文件的更改。
或者说一个分支会是另一个开发路线，进行下一个版本的更新，这都将导致文件的不同。

》》》这两个分支一般会是什么关系，是两个相互没有什么联系的文件区域吗?
答：
分支之间可以相互独立，也可以有一定程度的关联。
一般来讲，一个库的不同分支之间通常是有一定联系的，他们可能代表着同一库的不同版本、不同特性或不同目标的开发路径。

》》》两个分支可以被单独的下载或者使用吗?
答：
分支通常可以单独下载或使用，具体取决于代码管理工具（如Git）的支持和库的发布方式。

》》》不同分支的文件管理的状态?
不同的分支中的文件可以分开单独管理。
可以在不同的分支中对同一个仓库中的文件进行不同的修改，而不会相互影响。

》》》tags 和 branches 的区别
tags:
　　标签通常用于标识特定的版本或提交，一旦创建就不会随着新的提交而改变

branches:
　　分支用于代表不同的代码开发路径，可以持续地进行提交和修改

》》》》git clone 克隆的代码来自哪个分支?
如果没有指定特定的分支或标签，git clone 命令会默认克隆源库的主分支（通常是 master 或 main 分支）。

》》》》关于 ImGui，docking 分支从哪个版本开始正式投入使用?

从 ImGui 版本 1.80 开始，docking 功能被添加到主分支（master branch）中
因此在 1.80 版本及之后的版本，在主分支上便可以找到对 docking 功能的支持。

》》》从 1.80 开始，docking 功能已经被添加到主分支，为什么在主分支之外仍然存在一个 docking 分支呢?
因为开发团队为了保持代码的整洁和稳定性，在主分支之外继续维护一个用于开发和测试新功能的分支。

》》》如何理解被添加到主分支?
首先要知道，不同分支一般存放的代码有什么区别?
主分支：
　　在开源项目中，通常会有一个主要的代码库，其中包含了当前版本的稳定代码以及最新的功能开发。
　　这个主要的代码库就是主分支（master branch）或者叫主线。
其他分支：
　　其他分支一般用于不同的目的，比如开发新功能、修复 bug、实验性质的功能等。

添加到主分支的意思?
一个项目通常包含多个分支，docking 恰恰就是用来开发和测试 docking（停靠）功能的。
当 docking 功能开发完成并被认为稳定时，开发者就将其合并到主分支中，成为主要代码库的一部分。

这就是添加到主分支的意思。

》》》》origin 在 Git 中的意思

origin 是默认的远程仓库名称，通常指向你从中克隆或者拉取代码的远程仓库。
我们一般使用 origin 来表示默认的远程仓库，就而不必每次都指定完整的远程仓库名称。

例如：
git checkout -b docking origin/docking。
从 ImGui 的源仓库克隆到本地，并且将其命名为 origin，同时你在个人的远程仓库也叫 ImGui，那么 origin/docking 表示从名为 origin 的远程仓库中获取 docking 分支的引用。

》》》》子模块切换分支的方法

1. 进入子模块目录，找到想要切换分支的子模块目录，然后进入它的根目录切换到想要的分支：
git checkout branch_name　　//这会将子模块切换到名为 branch_name 的分支。

2. 返回到子模版目录：
cd ..

3.1 提交主项目的变更：
git add path/to/submodule

3.2 git commit -m "Switch submodule to branch_name"
（可选：如果你只是想在本地切换子模块的分支，而不需要将这个更改记录在主项目的提交历史中，那么提交主项目的变更就不是必须的。）

之后如果需要将父仓库推送到远程仓库，使用命令 git push 进行推送

》》》》Git 指令中 fetch 和 pull 的区别
fetch:
　　fetch 命令会从远程仓库下载新的提交和分支，但不会自动合并任何下载的更改到你当前的工作分支上，即不会修改你的工作目录中的文件
pull:
　　pull 命令实际上是执行了 fetch 命令，然后立即将远程分支的更改合并到当前分支中
　　（即 git fetch 和 git merge 命令的组合）

》》》》和 void ImGuiLayer::Begin( ) , void ImGuiLayer::End( ) ,void ImGuiLayer::OnImGuiRender( ) 的关系?
实际上参考 main.cpp 中的例子可以知道，m_ImGuiLayer 的 begin和end 分别对应

```
    // Start the Dear ImGui frame
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();
    ImGui::NewFrame();
```

和

```
// Update and Render additional Platform Windows
// (Platform functions may change the current OpenGL context,
// so we save/restore it to make it easier to paste this code elsewhere.
// For this specific demo app we could also call glfwMakeContextCurrent(window) directly)
 if (io.ConfigFlags & ImGuiConfigFlags_ViewportsEnable)
 {
     GLFWwindow* backup_current_context = glfwGetCurrentContext();
     ImGui::UpdatePlatformWindows();
     ImGui::RenderPlatformWindowsDefault();
     glfwMakeContextCurrent(backup_current_context);
 }
```

Begin 是在真正的渲染前所做准备的工作（创建新帧）。
End 是对渲染内容进行的渲染呈现。

所以 OnImGuiRender ( )函数中对应main.cpp，应该实现的是

```
// 1. Show the big demo window
        if (show_demo_window)
           ImGui::ShowDemoWindow(&show_demo_window);

// 2. Show a simple window that we create ourselves. We use a Begin/End pair to create a named window.
    {
        static float f = 0.0f;
        static int counter = 0;

        ImGui::Begin("Hello, world!");                          // Create a window called "Hello, world!" and append into it.

        ImGui::Text("This is some useful text.");               // Display some text (you can use a format strings too)
    ........
    ........
        ImGui::End();
    }

    // 3. Show another simple window.
    if (show_another_window)
    {....省略......}
```

这便是真正要渲染的内容，其中正使用了 ImGui::Begin( ) ,ImGui::End( )
所以void ImGuiLayer::Begin( ) , void ImGuiLayer::End( ) ,void ImGuiLayer::OnImGuiRender( ) 的关系一目了然

》》》》关于 Cherno 所说的 OnUpdate 和 OnImGuiRender 的区别怎么理解

Application.cpp中：

```
        for (Layer* layer : m_LayerStack)
            layer->OnUpdate();                  //执行逻辑更新

        m_ImGuiLayer->Begin();
        for (Layer* layer : m_LayerStack)
            layer->OnImGuiRender();             // 进行渲染操作（执行渲染更新）
        m_ImGuiLayer->End();
```

OnUpdate:

OnUpdate() 函数被用来执行逻辑更新。通常情况下，OnUpdate() 函数用于更新应用程序的逻辑状态。
例如更新对象的位置、处理输入事件、执行物理模拟等等。这个过程在每一帧都会执行，以确保实时修正逻辑操作内容。

在每一帧的渲染过程中，首先进行逻辑更新，然后才进行渲染操作

OnImGuiRender:

得到正确的逻辑状态，进行内容的最新渲染结果。

》》》》Cherno 提出的问题
为什么在 Sandbox 中的运行语句会导致问题？

ImGui 被设置成静态库 .lib
Nut(Hazel) 是 .dll
Sandbox 是 .exe

现在 .dll（Hazel） 接受 .lib（ImGui） 中的函数，然后.exe（Sandbox）能够使用存放在 .dll 中的函数
虽然 .lib 中有所有的函数名称，但是 .dll 是动态加载的，如果一个函数在之后不被使用，.dll就不会执行那个函数。（.dll有能力删除.lib中未使用的内容）
正因为 .exe 使用的函数是 .dll 中没有加载过的（.dll 中没有包含的函数名称），所以在 .exe（Sandbox） 中使用这些函数就会导致程序崩溃。

----------渲染简介和渲染架构----------

讲了一些基础，有点晦涩，介于之前看过 OpenGL 教程，也就马马虎虎看完了。
涉及的要点可以自己搜索，我建议还是将后面几集做完了回头看。

----------渲染和维护----------

》》》》Cherno在程序的属性页进行了修改，虽然我照做了，但并没有成功
于是我在 ImGui 的 premake 文件中做了修改：

```
filter "system:windows"
    systemversion "latest"
    cppdialect "C++17"
    staticruntime "On"

    defines
    {
        "IMGUI_API=__declspec(dllexport)"
    }
```

然后成功实现了。（这个方法是我在youtube某一个视频下方找到的）

----------静态库和无警告----------

》》》》关于使用静态库的好坏，截取了一条评论作为参考。

Static libraries for these kinds of projects are totally fine! The reason is that the user is not gonna run more than 2 -4 applications at the same time, actually, 90% of the users will probably run only one application at a time. Dlls make sense when creating OS -level libraries that are being used by hundreds of processes at the same time, like a windowing API (win32) for example, or Xorg (Linux). Then yes, it would be a waste of memor y, but in the application layer usually, users run one application at a time.

熟肉：
此类项目的静态库完全没问题！ 原因是用户不会同时运行超过 2-4 个应用程序，实际上，90% 的用户可能一次只会运行一个应用程序。 当创建同时被数百个进程使用的操作系统级库时，Dll 很有意义，例如窗口 API (win32) 或 Xorg (Linux)。 是的，这会浪费内存，但在应用程序层中，用户通常一次运行一个应用程序。

----------渲染上下文----------

》》》》什么是句柄
概念： 句柄（Handle）是用于标识资源或对象的抽象概念。
形式： 通常是一个数值或者引用，用来表示系统所管理的资源，例如内存块、文件、图形界面元素等。
作用： 提供对某些资源的访问和操作方式，从而不需要直接访问资源本身。

举例：

文件句柄（C语言实现）：
FILE *fileHandle;
fileHandle = fopen("example.txt", "r"); // fopen() 打开文件将返回一个文件指针，可以视作文件句柄

在程序中打开一个文件时，系统会返回一个文件句柄，用于标识该文件。通过 fileHandle 获取句柄，可以用来对文件进行读取或写入操作。

》》》》为什么要抽象上下文?
为了拓展程序，增加其普适性。将上下文抽象，我们可以采用不同的API来进行上下文的设置。
比如使用OpenGL、Vulcan、DirectX 等。

----------首个三角形----------

》》》》涉及到很多OpenGL中的函数和知识，
可以看OpenGL的参考文档，
当然也可以看cherno的教程。
或者Learn OpenGL官网。

我看完了Cherno 的教程，LearnOpenGL学了一半，所以这里没啥要记的。
我可以把学习Cherno 时记得笔记放在笔记文件夹中，格式会有点乱，可以参考一下。
（为啥格式会这么乱啊，特地去修改了一次，如果格式任然乱也有可能因为缩放导致的，尝试缩放看看）

》》》》后面很多东西其实都是Cherno在他教程里面所教的，一定去看看他的视频。
基础知识和类的抽象什么的，稍有不同。Cherno的思路也很不错，赞。

----------OpenGL着色器----------

》》》》std::make_ptr<> 和 reset 的概念与区别
区别：
std::make_ptr 创建智能指针的全局函数
reset 则是智能指针对象的成员函数

- ***std::make_ptr<>***

概念：
是一个模板函数，用于创建智能指针，并将其初始化为指定类型的对象。
参数：
要创建的对象的构造函数中的参数。
返回值：
返回一个指向新分配的对象的智能指针。

- ***.reset()***

概念：
智能指针类的成员函数，用于重新分配该智能指针所拥有的资源。
参数：
一个指针（指向新对象的指针）或者为空。
注意：
如果智能指针之前拥有资源，该资源会被释放。
如果不传递参数给 reset，则智能指针将被重置为空。

》》》》想起来一个东西，关于上一节的顶点属性的两个参数 stride（步幅） & offset（偏移量）不同情况下的理解

参数概念：
步幅（stride）指的是相邻顶点数据在数组中的字节间隔
偏移量（offset）指的是每个顶点属性在数组中的起始位置于数组本身开始位置之间的字节偏移量。

1.如果顶点是在结构体中放着的，像这样：

```
struct vertices {
    float position [3] ();
    float color[3] ();
}
```

实际的结构就会是：（位置和颜色分开存储，但每个属性的数据块在其各自区域内紧密排列）

```
{                       //紧密排列指的是空间内存放的都是顶点数据，没有其他东西
    x1, y1, z1,
    x2, y2, z2,
    x3, y3, z3,
    r1, g1, b1,
    r2, g2, b2,
    r3, g3, b3
}
```

stride：是相邻顶点数据在数组中的字节间隔
结果：
position 的 stride 就是 3 * sizeof(float)， color 的 stride 会是 4 * sizeof(float)
1.（能不能像下一个排列那样写成 7 * sizeof(float)： 不能，因为位置和颜色分开储存了）

2.（能不能都写成 0： 可以，因为每个属性的数据块是分开储存的，OpenGL可以不用知道 stride 有多大，直接顺着读取下一个相同属性的分量）

offset： 是指一个顶点属性开始时的位置的偏移量。

结果：

position 的 offset 就会是 0， color 的 offset 就会是9个浮点类型（9 * 4 = 36 byte），offset 会根据顶点属性的不同而变化，

每一个都不同，越向后某一个属性的 offset 就越大（累加）。

2.如果顶点所有属性全都放在一个数组中，像这样：

```
float vertices[] = {
      // 位置                 // 颜色
      0.5f, -0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   // 右下
     -0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   // 左下
      0.0f,  0.5f, 0.0f,   0.0f, 0.0f, 1.0f    // 顶部
};
```

实际结构会是：

```
{
      x1, y1, z1, r1, g1, b1,
      x2, y2, z2, r2, g2, b2,
      x3, y3, z3, r3, g3, b3
};
```

stride： 指的是指的是相邻顶点数据在数组中的字节间隔

结果：

这里的一个完整的顶点应该是（x1, y1, z1, r1, g1, b1），position和color的stride 会是6个浮点类型（6 * 4 = 24 byte）

（能不能都写成 0： 不能，这里的顶点不是分开储存的，OpenGL将无法正确地从一个顶点的属性跳到下一个顶点的相同属性）

offset： 是指一个顶点属性开始时的位置的偏移量。

结果：

position 的 offset 就会是 0， color 的 offset 就会是3个浮点类型（3 * 4 = 12 byte），offset 会根据顶点属性的不同而变化，

每一个都不同，越向后某一个属性的 offset 就越大（累加）。

通过相同的 stride，得知一个完整的正确的顶点是什么样的。

通过不同的 offset 我们可以去合适的地方访问到正确的顶点数据。

这个问题曾让我犯难，希望这种解释足够明了。

----------渲染接口抽象----------

》》》》只包含了父类的头文件，是否可以使用子类中的函数

分情况：

1.使用的函数是父类中的虚函数，在子类中重写。

可以使用，因为父类的头文件包含子类虚函数的声明。

2.使用的函数是仅子类中有的。

不可以使用，编译器无法识别和访问子类的其他函数。

》》》》缓冲区抽象之后的初始化方式为什么是这样？

为什么在Application中不使用 Buffer 的子类 OpenGLBuffer 的构造函数来初始化顶点缓冲区对象

m_VertexBuffer.reset(OpenGLVertexBuffer(...));

甚至是不直接 OpenGLVertexBuffer VB(...) 这样来初始化对象呢？

反而是使用了 Create 这个静态函数

m_VertexBuffer.reset(VertexBuffer::Create(vertices, sizeof(vertices))); 来初始化对象。

这个问题在看了视频5/7时候令人不解，但是到后面发现。

Create 函数会根据实际情况（你选择的API）选择一个接口，在对应接口下，Create 会根据实际选择对应的文件，这其中包含该接口规范下编写的构造函数，能过做到对应情况下调用正确的接口函数。

所以这样使用的原因是，需要通过这个函数来选择接口，条件满足时，会自动选择恰当的构造函数。

》》》》glGenBuffers 和 glCreateBuffers 的区别

glGenBuffers：

概念：

glGenBuffers 函数是 OpenGL 旧版本函数，用于一次性生成一个或多个未命名的缓冲区对象的 -> 标识符。

不同：

生成的标识符并不会自动与任何缓冲区对象关联，仅仅是标识缓冲区对象的唯一名称。

glCreateBuffers：

概念：

glCreateBuffers 是 OpenGL 4.5 引入的函数，用于一次性创建一个或多个缓冲区 -> 对象，并返回对应的标识符。

不同：

glCreateBuffers 会自动将标识符和对象相关联，并将生成的缓冲区对象绑定到 GL_ARRAY_BUFFER 目标上（如果创建的是顶点缓冲区）。

总结：

在使用上，glCreateBuffers 更加方便，因为它不仅仅创建了缓冲区对象的标识符，还自动将其绑定到了适当的目标上。
（他简化了代码，减少了 GenBuffers 和 BindBuffer 的组合操作，提高可读和易用性。）
而 glGenBuffers 则需要在生成标识符后，再通过 glBindBuffer 将其绑定到目标上，多了一步操作。

理解：
但是从实际的使用情况上来看（Cherno 在代码中虽然使用了 Create ，但还是因为没有 Bind 而报错），二者应该是可以相互替换的。

》》》》关于类的静态成员变量的初始化，注意！！！
前提：
静态变量需要被初始化。

正确示范：
```
enum class RendererAPI
    {
        none = 0, OpenGL
    };

    class Renderer
    {
        static inline RendererAPI SetAPI(RendererAPI api) { s_API = api; }
        static inline RendererAPI GetAPI() { return s_API; }
    private:
        static RendererAPI s_API;
    };

    RendererAPI Renderer::s_API = RendererAPI::OpenGL;
    // 静态成员变量需要在类外被初始化
```

错误示范1（有警告，很明显错了）
```
enum class RendererAPI
    {
        none = 0, OpenGL
    };

    class Renderer
    {
        static inline RendererAPI SetAPI(RendererAPI api) { s_API = api; }
        static inline RendererAPI GetAPI() { return s_API; }
    private:
        static RendererAPI s_API = RendererAPI::OpenGL;
        // 静态变量不能在声明的时候进行定义
    };
```

错误示范2（无警告，但依旧错）
```
enum class RendererAPI
    {
        none = 0, OpenGL
    };

    class Renderer
    {
        static inline RendererAPI SetAPI(RendererAPI api) { s_API = api; }
        static inline RendererAPI GetAPI() { return s_API; }
    private:
        static RendererAPI s_API;
        Renderer::s_API = RendererAPI::OpenGL;
        // 实际上这是一个静态成员变量的声明和一个成员变量的定义，这实际上是两个不同的变量
    };
```

结论：
静态成员变量的初始化只能在类的外部进行，不能在类内进行。
静态成员变量的初始化只能在类的外部进行，不能在类内进行。
静态成员变量的初始化只能在类的外部进行，不能在类内进行。

》》》》#if 和 #ifdef 的区别
#if 和 #ifdef 的作用类似，但是语义不同。
#if 用来检查条件表达式的真假
#ifdef 用来检查标识符是否已经被定义

----------顶点缓冲区布局----------

》》》》enum class A : uint8_t 其中 " : uint8_t " 是什么意思？
这表示枚举类中每个枚举常量所分配的枚举值在内存中以 uint8_t 的形式存储，所以可以限制枚举值的范围。
枚举值指的就是代表每一个枚举变量/常量的数字。（默认情况下从0到后累加）

什么是 uint8_t ?
C++标准库中定义的无符号 8 位整数类型，其取值范围是 0 到 255。

》》》》什么是初始化列表 Initialized_list()
概念：
是一种用于初始化对象的机制，它允许在对象创建时提供一个包含初始值的列表。
用途：
通常用于初始化数组、容器、类的成员等。

eg:
```
// 使用初始化列表初始化数组
  int arr[] = {1, 2, 3, 4, 5};

// 使用初始化列表初始化 std::vector
  std::vector<int> vec = {6, 7, 8, 9, 10};
```
》》》》在设计完一切后，我发现一个问题，关于 stride 和 offset 的正确性。

记得上面我们说过的两种形式下的顶点结构所对应的不同的 stride 和 offset吗

1.

（位置和颜色分开存储，但每个属性的数据块在其各自区域内紧密排列）

```
{ x1, y1, z1,
 x2, y2, z2,
 x3, y3, z3,
 r1, g1, b1,
 r2, g2, b2,
 r3, g3, b3 }
```

2.

```
{ x1, y1, z1, r1, g1, b1,
  x2, y2, z2, r2, g2, b2,
  x3, y3, z3, r3, g3, b3 }
```

这两种情况下，position 和 color 分别对应 stride 和 offset 需要区别处理。

然而在我们在代码设计的过程中，这个函数计算出来的结果

```
void CalcOffsetAndStride(){
    m_Stride = 0;
    uint32_t offset = 0;
    for (auto& element : m_Elements){
        element.Offset = offset;
        offset += element.Size;
        m_Stride += element.Size;}
}
```

对应的应该是

```
{ x1, y1, z1, r1, g1, b1,
  x2, y2, z2, r2, g2, b2,
  x3, y3, z3, r3, g3, b3 };
```

时，所对应的两个 stride 和 两个 offset.

但是我发现在 Application 中进行声明的时候，BufferLayout layout = {{ShaderDataType::Float3, "a_Position"}, {ShaderDataType::Float4, "a_Color" }};

看起来像是

```
{ x1, y1, z1,
 x2, y2, z2,
 x3, y3, z3,
 r1, g1, b1,
 r2, g2, b2,
 r3, g3, b3 }
```

这种形式呢，于是我思考，为什么Cherno在设计的时候，照常使用并得到正常结果呢。

其实问题在于，我将

BufferLayout layout = {{ShaderDataType::Float3, "a_Position"}, {ShaderDataType::Float4, "a_Color"}}; 完全理解成了一段对顶点结构定义的语句。

实际上这只是一段为了方便计算 stride 和 offset ，也方便阅读代码设计的语句。

查看定义，BufferLayout(std::initializer_list<LayoutElement> elements)，我们填入的初始化表属于LayoutElement规格。

而查看

```
struct LayoutElement {
    std::string Name;
    ShaderDataType Type;
    uint32_t Size;
    uint32_t Offset;
    LayoutElement(ShaderDataType type, const std::string& name)
        :Type(type), Name(name), Size( ShaderDataTypeSize(type) ), Offset(0){ }
};
```

这里也没有任何对于顶点数据，或者顶点结构的定义。

因为我们（Cherno）在设计时并不是通过 BufferLayout layout = {{ShaderDataType::Float3, "a_Position"}, {ShaderDataType::Float4, "a_Color"}}; 来确定顶点的，我们会先行定义顶点（以这种方式）

```
float vertices[3 * 6] = {
  -0.5f, -0.5f, 0.0f, r1, g1, b1,
   0.5f, -0.5f, 0.0f, r2, g2, b2,
   0.0f,  0.5f, 0.0f r3, g3, b3
};
```

所以这样的设计是完全没有问题的。

//事实上，只要我沉住气看个十来分钟，就能看到Cherno使用

//float vertices[3 * 6] = {

// -0.5f, -0.5f, 0.0f, r1, g1, b1,

//   0.5f, -0.5f, 0.0f, r2, g2, b2,

//   0.0f, 0.5f, 0.0f r3, g3, b3

//};

//这样的语句了，我为什么不沉下心来看看呢?

//否则我绝对不会提出这么蠢的问题。 :-|

》》》》关于为什么在 BufferLayout 类型中使用 m_Stride 记录步幅，而不在LayoutElement(BufferElement)类型中和 offset 一样，将 stride 作为结构体的成员，用来记录步幅呢，反而在 Layou tElement(BufferElement) 结构体中，使用 offset 这个成员来记录偏移量。

因为 offset 对于每一个属性的顶点都不一样，需要区别起来，这样在 Calc 函数中进行 offset 的计算得到的不同 offset 会存放在其对应属性的结构体中

但是对于 stride，有一个规律，就是在

```
{ x1, y1, z1, r1, g1, b1,
  x2, y2, z2, r2, g2, b2,
  x3, y3, z3, r3, g3, b3 }
```

情况下，一旦结构是这样，且数据已经确定，每一个顶点属性函数 glVertexAttribPointer() 中 stride 参数这里填入的数据都会是一样的。

对于这个例子，位置 x,y,z 和 颜色 r,g,b 在顶点属性函数的 stride 参数这里填入的都是同样的 stride : sizeof(float) * 6。所以对于这种结构的顶点（仅对于第二种情况的这种排列方式来讲），offset 需要多个，但是 stride 只需要一个，故可以放在成员变量中这样定义。

》》》》m_Elements 是 std::vector<LayoutElement> 类型的，对于函数

BufferLayout(std::initializer_list<LayoutElement> elements)

:m_Elements(elements) 来说，为什么初始化表类型的参数在传入后，可以正确的被单独存放在 std::vector 的不同位置中?

理论上来看的话：初始化列表语法允许在创建对象时使用花括号 {} 来传递 “一组”值，并且编译器会将每个花括号内的值作为单独的初始化列表处理。
当你传递多个初始化列表用于初始化 BufferLayout 类型的对象时：
BufferLayout layout（{element1},{element2},{element3},{element4}）
每个初始化列表都会被视为一个单独的参数，在构造函数中会被分别处理。

这样，每个初始化列表中的元素都会被分别存储到 std::vector<LayoutElement> 中的不同位置，而不会被合并到同一个位置。

》》》》
int func();
int func() const;
const int& func();
const int& func() const; 的区别。

- **1.int func();**

非常量成员函数.
1.可以修改对象的成员变量
2.返回成员变量的拷贝

- **2.int func() const;**

成员函数
1.不会修改对象的成员变量
2.但是它返回的是成员变量的拷贝，而不是引用.(对返回值的修改不会影响到原始对象的成员变量。

- **3.const int& func();**

可以是一个全局函数，静态函数，或者是一个类的成员函数
1.在对应情况下，表示可以修改静态变量、全局变量或成员变量
2.返回一个对静态变量或者全局变量的常量引用

- **4.const int& func() const;**

成员函数
1.不会修改对象的成员变量
2.返回一个对成员变量的常量引用

》》为什么有的是成员函数有的不是，怎么辨别?
通常末尾有 const 关键字表明这个成员函数不会修改对象的成员变量，所以能够明确这种声明方式的函数通常会是一个成员函数。

》》const 在参数位置 void func(const char& str) 是什么意思?
1.函数不会修改这个参数的值
2.传递给函数的参数是一个对 char 的非常量引用
》这和 void func(char str) const 的区别是，前者说明不能修改类外变量，后者说明不能修改的是成员变量。

》》》》问题
1. 在 buffer.h 定义的时候，记得将 class BufferLayout 放在 class VertexBuffer 之前，
因为在 VertexBuffer 中 使用了 BufferLayout 类型声明变量，因为使用的东西需要是之前定义过的，所以要把 BuffeLayout 这个类的定义放在前面

2. 我在实现element.GLType 的时候，我没有想 Cherno 一样将 GetTypeToGLType() 这个函数放在 Application.cpp 中，而是在 Buffer.h中定义了。
这里就需要注意一个问题，由于没有选择在 Application 中将其定义为一个函数，而是在 Buffer.h 中这样定义

```
GLenum GetTypeToGLType() const {
    switch (Type) {
    case ShaderDataType::Float:   return GL_FLOAT;
    case ShaderDataType::Int:          return GL_INT;
    case ShaderDataType::Mat3:    return GL_FLOAT;
    }
    NUT_CORE_ASSERT(false, "Unknown ShaderDataType !");
    return 0;
}
```

为此我特地在 Buffer.h 中添加了 <glad/glad.h> 的声明，正是这一句代码，会导致 Sandbox 产生不能打开 glad/glad.h 文件的报错。
所以我在 premake 文件的 sandbox 项目中的 includedirs 添加了 "%{IncludeDir.Glad}"，这解决了问题。

》》》》注意
在将初始化列表作为参数填入 BufferLayout 的 layout 中时，要确保 vertices 中有对应属性的顶点。
如果只更新布局而不同步更新顶点的话，会导致无渲染结果。

----------VertexArray 顶点数组----------

》》》》发现个问题（多次包含头文件导致的重复定义错误。）

在使用单例类的时候，我直接在头文件中将类中的静态成员变量放在类之外定义了。
这会导致一个问题，如果在使用多个文件，且这些文件中都包含了这个类，这会导致静态成员变量的重定义。（会在链接时报错，fatal error LNK1169: 找到一个或多个多重定义的符号
error LNK2005: "private: static enum Nut::RendererAPI Nut::Renderer::s_API" 已经在 Nut.lib(Buffer.obj) 中定义；）

届时，将此定义置于 对应的 .cpp 文件中即可解决。。（或者使用 inline 关键字来定义静态成员变量。）

----------渲染流和提交----------

这一集就是抽象了代码，使结构明了，不用再显式的调用 gl 函数去渲染物体了

》》》》variable = new classname;
variable = new classname(); 的区别。

一般情况下，
1.variable = new classname;
指隐式的调用默认的构造函数（无参），但是如果未定义默认的构造函数（或者函数不可见），则不会通过编译
2.variable = new classname();
指显式的调用默认的构造函数（无参），如果没有默认的构造函数，编译器则会自动生成一个。

但是：在此处（Cherno 在文件 RendererCommand.cpp 中定义的）classname（OpenGLRendererAPI）是 RendererAPI 的子类。

所以在使用 new classname 的时候，会优先调用父类的构造函数。虽然父类文件中没有定义构造函数，但是该类是单例类，此时编译器会为父类生成一个默认的无参构造函数。
此时variable = new classname;
variable = new classname(); 没有差别。

》》》》Renderer、RendererAPI、OpenGLRendererAPI、RendererCommand 几个文件之间的关系

1.定义函数（RendererAPI、OpenGLRendererAPI）
RendererAPI 定义了几个虚函数，这几个虚函数将在对应的接口文件中被定义实现，这里我们选择定义在 OpenGL 接口中。
在 OpenGLRendererAPI 中，我们定义了几个虚函数的实现，这将完成我们在渲染循环中将要进行的渲染操作。

2.将函数打包在对应接口下（RendererCommand）
通过上面两个文件，我们只定义了操作，但是没有设计根据情况自动调用对应接口的操作类，所以我们定义了 RendererCommand
在 .cpp 中我们能够 RendererAPI* RenderCommand::s_RendererAPI = new OpenGLRendererAPI; 调用对应接口处，
然后在 .h 中设置内联函数，使用 RendererCommand 类中的 s_RendererAPI 来调用对应接口中的函数。
eg.
s_RendererAPI->SetClearColor(color);

3.按需调用函数（Renderer）
最后，仅仅在 Renderer 文件中对 RendererCommand 中打包好的函数进行调用即可，此时要调用的函数已经是对应接口下的（高度抽象的）函数了
之后，我们只需要将想调用的函数放在 Renderer 中，然后通过 Renderer 来访问就好了。

》》》》发现一个问题：为啥在application.cpp中
RendererCommand::Clear();
RendererCommand::SetClearColor({ 0.1f, 0.1f, 0.1f, 1 });
Renderer::BeginScene();

m_SquareShader->Bind();
Renderer::Submit(m_SquareVA);

m_Shader->Bind();
Renderer::Submit(m_VertexArray);

Renderer::EndScene();
调换
m_SquareShader->Bind();
Renderer::Submit(m_SquareVA);
和
m_Shader->Bind();
Renderer::Submit(m_VertexArray);
的顺序。

一开始是三角形覆盖在方形之上，但为啥转换过来之后却只绘制出来个方形，没看到三角形嘞？？？

对了这是因为渲染顺序的原因，后渲染的物体会覆盖在先渲染的平面图形之上，这是正常的。

GameEngine3

2024年5月25日 17:03

----------相机&原理----------

》》》》查看这两篇说明，一个是坐标系统，一个是摄像机
（https://learnopengl-cn.github.io/01%20Getting%20started/08%20Coordinate%20Systems/）
（https://learnopengl-cn.github.io/01%20Getting%20started/09%20Camera/）
来自 learn OpenGL，很不错的网站，通俗易懂。

完全阐明了这一集的知识点。

----------正交相机----------

》》》》原理：

最开始处于局部空间，在进行坐标变换和摄像机设置的时候，需要进行以下操作。

局部空间 * 模型矩阵 -> 世界空间	模型矩阵（类型：glm::translate）
世界空间 * 观察矩阵 -> 观察空间	模型矩阵（类型：glm::rotate）
观察空间 * 投影矩阵 -> 裁剪空间	模型矩阵（类型：glm::ortho/glm::prespective）
裁剪空间 + 坐标变换 -> 屏幕空间	自定义的操作（类型：glm::translate/glm::scale/glm::rotate）

一般会为坐标乘以 Projection * View * Model。
（比如着色器中会：gl_Position = projection * view * model * vec4(aPos, 1.0f); ）

》》》》inverse 函数在RecalcMatrix() 中的作用
首先，在 Cherno 的代码中，Cherno 将模型矩阵和观察矩阵设置后，放在 transform 中，然后赋值给 m_ViewMatrix.
其实，在代码中 m_ViewMatrix 实际上指代的应该是 m_ModelViewMatrix.

问题：
这时候讲讲为什么要使用 inverse 对 transform 进行转换，然后赋值给 m_ViewMatrix.

因为观察矩阵（View Matrix）通常需要描述观察者相对于世界空间的位置和方向，而不是物体相对于观察者的位置和方向。
也就是说该矩阵是用来作用于摄像机 camera 的，而不是作用于空间中的物体。

如果不进行逆操作，会将变换应用到物体上，那么结果将描述物体相对于 camera 的位置和方向，而不是 camera 相对于世界空间的位置和方向。

》》》》在着色器中，对坐标进行转换时进行乘法的顺序。
gl_Position = projection * view * model * a_Pos;矩阵有先后顺序，而且需要在坐标之前先进行运算。

》》》》glm::value_ptr 的概念与作用
概念：获取 GLM 类型（如矩阵、向量等）内部数据的指针
作用：将 GLM 类型转换为指向C++内部数据的指针，以便将数据作为参数传递给 OpenGL 函数或其他需要 >指针参数<的函数

eg.
```
glm::mat4 matrix = glm::mat4(1.0f);                                    // 创建一个4x4的单位矩阵
glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(matrix));     // 将矩阵传递给OpenGL
```

----------Moving to sandbox （含摄像机移动）----------

》》》》关于垂直同步（V-Sync）的理解

》》V-Sync 用于解决图像撕裂的问题，什么是图像撕裂？为什么会出现图像撕裂？
一般情况下 GPU 的渲染速度会比屏幕的刷新率快，当 GPU 的渲染速度高于显示器的刷新率时，GPU 在显示器完成一次完整的刷新之前已经渲染了新的图像，而显示器可能还在显示上一帧的部分内容。
这导致在显示图像的某个位置上出现了不连续的线条，即图像撕裂。

》》
理解：
通过垂直同步对 GPU 和显示器的垂直刷新率进行同步，限制GPU输出的帧率，使其与显示器的刷新率保持一致。
（在 OpenGL 中，开启垂直同步后就是调用 glfwSwapInterval(0) 产生这样的效果）
这样，GPU只会在显示器完成一次完整的垂直刷新周期后才输出下一帧图像，从而避免了图像撕裂现象的发生。

缺点：
因为GPU必须等待显示器完成垂直刷新周期后才能输出新的帧，所以不能充分利用显卡性能，可能导致输入延迟和帧率下降。
在 CS::GO 中，我会选择将其关闭 XD

----------TimeStep----------

》》》》关于 Conversion Operators。

》》》》 operator float() { return Variable } 是啥?

概念:

转换运算符/转换操作符（Conversion Operators）是一种特殊的成员函数，

它们允许用户定义自定义类型之间的隐式或显式类型转换(允许你将 Timestep 对象隐式地转换为 float 之类的数据类型).

语法:

operator target_type();

参考视频:

来自 Cherno 的（https://www.youtube.com/watch?v=OK0G4cmeX-I&t=368s)

eg.

```
operator float() const { return m_Time; }      // This is for a varible which typed by "Timestep" can be use with arithmetic operator like + - * /
float GetSeconds() const { return m_Time; }
float GetMilliseconds() const { return m_Time * 1000.0f; }
private:
	float m_Time;
```

----------Transform（变换）----------

》》》》关于 SetPositoin 和 变换 的区别

SetPosition 和 SetRotation 是对摄像机的变换。这一集中我们需要对物体进行变换。

----------Texture（纹理）----------

》》》》Cherno 对于纹理系统设计的一些展望。

没有什么知识点感觉。

关于其中的一些细节Cherno 会慢慢实现的，到时候我再就 PBR 之类的知识点进行学习。

----------着色器抽象和统一变量----------

》》》》参考文献:

来自 Learn opengl：（https://learnopengl-cn.github.io/01%20Getting%20started/05%20Shaders/）

包含此集全部内容。（除了 ImGUI UI 的实现）

》》》》在类的成员函数之后调用 = default 是什么意思?

1.这个语句只可以出现在以下特殊成员函数之后:

默认构造函数
拷贝构造函数
移动构造函数
析构函数
拷贝赋值运算符
移动赋值运算符

2.意为指示编译器对其生成默认的实现，比如默认的构造函数。

》》》》关于 Cherno 将 Submit 的 Shader 类型的参数转换为 OpenGLShader 类型，这是否会报错?

问题概述:

在 C++ 中，你可以

1.将指向 派生类对象的指针/引用 转换为 指向基类对象的指针/引用,

2.将指向 基类对象的指针/引用 转换为 指向派生类对象的指针/引用。

但值得注意的是:

第一种情况称之为：向上转型（upcasting），无需进行显式转换

第二种情况称之为：向下转型（downcasting），需要进行显式类型转换。（正如 Cherno 所做的那样 std::d_p_cast<XX>(XX) ）

需要注意:

1.向上转型:

1.1在转换后如果只访问基类 Shader 中的成员，因为之前派生类对象 OpenGLshader 包含了基类 Shader 对象的所有成员，所以这没有什么差别，很安全。

1.2但是如果你希望转换后还能访问到派生类 OpenGLShader 中特有的成员，就需要用到虚函数这样动态加载的方法。

2.向下转型:

2.1向下转型是有风险的，因为只有当基类 Shader 指针或引用确实指向一个派生类 OpenGLShader 对象时，向下转型才是安全的。

（我们对于 Submit 函数 shader 传入的参数就是 OpenGLShader 类型的，所以安全）

2.2对于原来的参数， Shader shader，我们对其进行向下转换后，对于 OpenGLShader 类型的参数只能调用该派生类中的成员函数。

那么现在关于问题的答案是：不会。

//Cherno 传入了对应的派生类型的参数，而且处理的是向下转型, 使用的是派生类中的成员函数。

》》》》关于 std::dynamic_pointer_cast

1.初始指针：直接使用类型名称声明指针类型 eg: int*
优点是不会使用额外的性能用来维护
缺点是不安全。

2.智能指针：std::dynamic_pointer_cast
优点是提供更高的类型安全性，比如可以自动管理内存、处理异常
缺点是使用额外的性能来维护程序正确性。（其实在 >小项目< 中没多少）

概念：
std::dynamic_pointer_cast 是 C++ 标准库中的一个模板函数，用于在运行时进行动态类型转换。

----------指针(Ref & Scope)----------

》》》》关于智能指针，Cherno有一期视频介绍了。
check that out if you haven't already. （https://www.bilibili.com/video/BV1hv411W7kX/?spm_id_from=333.999.0.0&vd_source=64ca0934a8f5ef66a21e8d0bddd35f63）

》》》》什么是强引用？什么是弱引用？二者有什么作用？
声明：
在 C++ 中，通常没有与其他编程语言中的强引用和弱引用直接对应的概念。但是，可以使用普通指针和智能指针来模拟类似的引用行为。

概念（Java 中）：

- 强引用：
  强引用是一种对对象的正常引用，它会使对象的引用计数加一。
  只要存在强引用指向一个对象，该对象就不会被垃圾回收系统回收，即使内存紧张也不会被释放，

除非所有指向该对象的强引用都被解除（即没有任何对象持有对该对象的强引用），对象才会被销毁和释放内存。

- 弱引用
  弱引用是一种非强制性的引用，它不会增加对象的引用计数。
  弱引用允许对象被垃圾回收系统回收，即使还有弱引用指向该对象，只要没有强引用指向它，对象就可能被回收。

对应用例（C++）：

- 强引用：

在 C++ 中，使用普通指针或者智能指针来引用对象时，通常就会是一种强引用。
1.普通指针不具备自动管理内存的能力，需要手动释放资源
2.智能指针（如 std::shared_ptr 内含计数系统）同时也可以自动管理内存生命周期，当没有任何智能指针指向对象时，会自动释放资源。

- 弱引用：

在 C++ 中，可以使用 std::weak_ptr 来模拟弱引用的行为。
std::weak_ptr 是 std::shared_ptr 的伴随类，它允许观察 std::shared_ptr 指向的对象，但不会增加对象的引用计数。
因此，当所有 std::shared_ptr 指向对象的引用都释放时，即使存在 std::weak_ptr，对象也会被释放。

参考Java中的概念理解。

作用：
1.强引用可以确保对象在需要时不会被提前释放。
2.弱引用可以防止对象复用，同时避免内存泄漏。

》》》》关于unique_ptr 的理解。

unique_ptr 可以看做是强引用。

相较于普通指针：
std::unique_ptr 是一种独占所有权的智能指针，与普通的指针不同，它不允许多个指针同时指向同一块内存区域。
相较于共享指针 shared_ptr：
std::unique_ptr 不会增加引用计数，而是在其生命周期结束时自动释放所管理的对象。更加轻量。

》》》》什么是“原子”？
概念：
“原子”（atomic）是一个术语，用来描述不可分割、不被中断的操作或动作。
当我们说某个操作是原子的，意思是这个操作在执行过程中不会被其他操作打断，无论是因为多线程竞争还是其他原因。

常见的原子操作：
原子读写：确保读取或写入数据时不会受到其他线程的干扰。
原子增量/减量：确保对数值进行加减操作时是不可分割的。
原子比较和交换（CAS）：一种常用的原子操作，用于在多线程环境中实现锁定机制，确保在执行特定条件下的数据交换是原子的。

作用：
原子操作在并发和多线程环境中起关键作用，因为它确保了数据的一致性和完整性。在这些环境中，多个线程可能会同时尝试修改同一个数据。这时，如果没有原子操作，可能会发生竞态条件（race condition），导致数据错误或不可预测的行为。

使用：
C++：可以使用 <atomic> 头文件提供的 std::atomic 类型来确保原子性。

》》》》什么是“原子增量”？ 什么是“原子减量”？

原子增量（Atomic Increment）：
原子增量是指对共享变量进行加一操作，并且这个操作是原子性的，即不会被中断或者被其他线程干扰。
（在多线程环境中，当多个线程同时尝试对同一个共享变量进行增量操作时，如果不使用原子操作，可能会出现竞态条件（race condition）导致结果不确定或者出错。）
原子增量操作保证了在任何时刻只有一个线程能够对变量进行递增操作，从而确保了结果的正确性。

原子减量（Atomic Decrement）：
原子减量与原子增量相反，是对共享变量进行减一操作，并且保证操作的原子性。
类似于原子增量，原子减量操作也是为了避免竞态条件而设计的，确保在多线程环境下对共享变量进行减少操作时的正确性。

----------Texture（纹理）----------

》》》》部分知识参考（https://learnopengl-cn.github.io/01%20Getting%20started/06%20Textures/）
也可以去看Cherno OpenGL系列的视频。

》》》》glTextureStorage2D和 glTextureSubImage2D分别是什么用法，这两个函数有什么作用？这和glTexImage2D有什么不同？

- glTexImage2D：
  glTexImage2D 用于指定一个二维纹理图像的数据。
  这个函数会为纹理对象分配内存并设置其格式、宽度、高度以及初始数据。

当调用 glTexImage2D 时，如果纹理对象已经存在，则会重新分配内存并覆盖原有数据。

- glTextureStorage2D：
  glTextureStorage2D 用于分配存储纹理对象的内存空间，但不会填充具体的纹理数据。
  这个函数通常在创建纹理对象时使用，用于指定纹理的维度、级别数量和内部格式，但不会填充实际的像素数据。

与 glTexImage2D 不同，glTextureStorage2D 不会重新分配内存或更改纹理对象的大小，只会分配足够的内存空间。

- glTextureSubImage2D：
  glTextureSubImage2D 用于更新已分配内存的纹理对象的部分或全部像素数据。
  这个函数可以用于更新纹理对象的某个区域，而不影响其他区域。

它通常用于动态更新纹理数据，比如视频纹理、实时渲染等场景。

》》也就是说glTexImage2D就是glTextureStorage2D和 glTextureSubImage2D组合使用的效果吗？
差不多是的。因为：
1.glTexImage2D 提供了一个简单的接口，用于分配内存并设置初始纹理数据。它相当于一次性完成了内存分配和初始化数据的工作。

2.glTextureStorage2D 专门用于分配内存空间，但不填充具体数据。它通常用于创建纹理对象并指定其格式、大小和级别数量。
glTextureSubImage2D 则用于在已分配内存的纹理对象上更新数据。它允许你在不重新分配内存的情况下，动态更新纹理的像素数据。

所以可以通过组合 glTextureStorage2D 和 glTextureSubImage2D 来实现类似 glTexImage2D 的功能

》》》》glActiveTexture(GL_TEXTURE0); + glBindTexture(GL_TEXTURE_2D, texture1);
和 glBindTextureUint的区别。

- glActiveTexture(GL_TEXTURE0);
- glBindTexture(GL_TEXTURE_2D, texture1);

这是传统的绑定纹理的方式，适用于 OpenGL 2.0 及更高版本，
首先，使用 glActiveTexture() 设置活动的纹理单元，然后通过 glBindTexture() 将特定的纹理对象绑定到该纹理单元。
适合用于绑定和切换不同的纹理单元，以便在着色器中使用。

- glBindTextureUnit(unit, texture);

这个函数在 OpenGL 4.5 及更高版本中可用。
这个方法直接将纹理对象绑定到特定的纹理单元，而无需先激活纹理单元。

区别：
glBindTextureUnit() 更直接、更高效，因为它不需要显式调用 glActiveTexture()。有助于减少代码复杂性并提高性能。
因为可以直接操作纹理单元，而无需通过传统的激活-绑定流程。在 4.5 以上版本，可以用来替代 Active + Bind;

》》》》stb 库的支持
QUICK NOTES:
Primarily of interest to game developers and other people who can
avoid problematic images and only need the trivial interface

JPEG baseline & progressive (12 bpc/arithmetic not supported, same as stock IJG lib)
PNG 1/2/4/8/16-bit-per-channel

TGA (not sure what subset, if a subset)
BMP non-1bpp, non-RLE
PSD (composited view only, no extra channels, 8/16 bit-per-channel)

GIF (*comp always reports as 4-channel)
HDR (radiance rgbE format)
PIC (Softimage PIC)
PNM (PPM and PGM binary only)

Animated GIF still needs a proper API, but here's one way to do it:
http://gist.github.com/urraka/685d9a6340b26b830d49

- decode from memory or through FILE (define STBI_NO_STDIO to remove code)
- decode from arbitrary I/O callbacks
- SIMD acceleration on x86/x64 (SSE2) and ARM (NEON)

Full documentation under "DOCUMENTATION" below.

熟肉：快速注释：
主要是游戏开发者和其他有能力的人感兴趣
避免有问题的图像，只需要简单的界面

JPEG 基线和渐进（不支持 12 bpc/算术，与库存 IJG 库相同）
PNG 每通道 1/2/4/8/16 位

TGA（不确定是哪个子集，如果是子集）
BMP 非 1bpp、非 RLE
PSD（仅合成视图，无额外通道，每通道 8/16 位）

GIF（*comp 始终报告为 4 通道）
HDR（亮度rgbE格式）
PIC（Softimage PIC）
PNM（仅限 PPM 和 PGM 二进制）

动画 GIF 仍然需要适当的 API，但这是一种方法：
http://gist.github.com/urraka/685d9a6340b26b830d49

- 从内存或通过 FILE 解码（定义 STBI_NO_STDIO 以删除代码）
- 从任意 I/O 回调中解码
- x86/x64 (SSE2) 和 ARM (NEON) 上的 SIMD 加速

完整文档位于下面的“文档”下。

----------blending（混合）----------

》》》》函数使用自行查阅文档或资料，Cherno 系列视频或 Learn OpenGL文档。

》》》》关于为什么使用多层抽象来实现 Renderer:: 的方法?
为了高度抽象，以供多接口使用。
》》RenderCommand 的作用?
1.可以直接静态的调用一些命令，而不是实现某接口的方法。
2.方便多线程的使用。

----------Shader Asset Abstract（着色器资产抽象）----------

》》》》 bat语法参考（浏览器打开）
https://winddoing.github.io/post/29269ff3.html?highlight=bat

》》》》@echo off 和 IF %ERRORLEVEL% NEQ 0 (PAUSE) 在批处理文件中的意思?

@echo off
关闭批处理文件的命令回显功能。
命令回显是指在批处理文件执行时显示每个命令在命令提示符窗口中的执行结果。
作用：这使输出简洁，执行高效。

```
IF %ERRORLEVEL% NEQ 0 (
  PAUSE
)
```

1. %ERRORLEVEL% 代表上一个命令的退出代码（或错误代码）
Windows上，程序和命令执行完成后会返回一个退出代码。返回代码0通常表示成功，非零表示错误或异常。
2. IF %ERRORLEVEL% NEQ 0
如果 %ERRORLEVEL% 的值不等于0（即先前的命令发生了错误）
3. PAUSE
暂停批处理文件的执行，直到用户按下任意键。

作用：如果执行某个命令后发生错误，批处理文件就会暂停并等待用户响应。否则直接运行完毕。

》》》》在 gitignore 中标明 bin 和 bin/ 有什么区别?

bin：表示会忽略所有包含 bin 的文件或目录
bin/：表示会忽略名为 bin 的文件目录（文件夹）

》》》》https://github.com/TheCherno/Hazel/pull/107
》》》》在这个 pull request 中，两人讨论一个问题，即 OpenGLBuffer 文件中，OpenGLIndexBuffer

OpenGLIndexBuffer 函数定义里面（注意是：IndexBuffer 中）
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, count * sizeof(uint32_t), indices, GL_STATIC_DRAW);
是否应该被写为：
glBindBuffer(GL_ARRAY_BUFFER, m_RendererID);
glBufferData(GL_ARRAY_BUFFER, count * sizeof(uint32_t), indices, GL_STATIC_DRAW);

@LovelySanta 说，在 4.5 中，vertex buffer 和 index buffer 只在 Bind( )
void OpenGLIndexBuffer::Bind() const {
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererID); }
这里有区别，需要分别写做：GL_ARRAY_BUFFER 和 GL_ELEMENT_ARRAY_BUFFER,
在 OpenGLIndexBuffer 和 OpenGLVertexBuffer 中，则都在 glBufferBind 和 Data 中填入 GL_ARRAY_BUFFER.

文献附上：
4.5（翻阅第60页）https://registry.khronos.org/OpenGL/specs/gl/glspec45.core.pdf
4.6（翻阅第60页）https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf

结论：（绑定缓冲区时指明即可，创建缓冲区时不用区别开来）
在OpenGL 4.5核心配置中，只需将缓冲区绑定到GL_ARRAY_BUFFER，而不需要在创建索引缓冲区时绑定到GL_ELEMENT_ARRAY_BUFFER。
唯一需要将 indexbuffer 的 slot 设置为 GL_ELEMENT_ARRAY_BUFFER 的情况是在绑定时，以指定缓冲区包含索引。

》》》》premake 中的 flags：
flags:
用于设置编译器选项或标志的一个属性. "MultiProcessorCompile"
是一个 MSVC 编译器的选项，用于启用多处理器编译。

作用：
当这个标志被设置时，编译器会尝试使用计算机上的多个处理器来并行编译源文件，以加快编译速度。

》》》》读取文件的函数设计理解：

》》关于 ifstream 类对象的理解：

对象的概念：
std::ifstream 对象 (比如 in) 是一个文件流，它代表了一个与文件相关联的输入流。
对象持有：
该对象持有一系列操作文件的方法和属性，你可以通过这些方法和属性来读取文件内容、控制文件读取位置、判断文件是否打开等等。
一般来讲，in 本身不包含文件中的数据，他只是持有一些方法，对文件操作，在某些情况下（比如将文件读取出来）作用于你自己声明的 string 类型变量上。

》》函数设计：
1.std::ifstream in(filepath, std::ios::in, std::ios::binary); --> 打开文件以备后续读取
从 filepath 中访问文件

1.1 以输入（读取）模式打开文件（std::ios::in），
1.2 并以二进制模式打开文件（std::ios::binary），后续将文件作为原始二进制数据读取。

2.if (in)
如果成功读取，in 会被隐式转换为 true，if (in) 表示文件若打开 ...

3.in.seekg(0, std::ios::end);
读取文件之前设置文件读取位置，通过 seekg 让这行代码将文件读取位置设置为：读取位置为末尾，且不设置方向。

3.1 这里的参数 0 表示相对于某个参考点的 offset 偏移量，（用来确定相对于参考点的读取位置）
(正数表示向文件末尾方向移动多少个字节，负数表示向文件开头方向移动多少个字节，零表示不移动，origin设置的参考点位置已经满足我们的用例)
3.2 std::ios::end 则就是偏移量的参考点，表示相对于文件末尾的位置。（用来确定读取的位置，offset 设置在参考点之前或之后多少个字节的位置）
（可以是 std::ios::beg（文件开头）、std::ios::cur（当前读取位置）、std::ios::end（文件末尾）中的一个，
由于只有三个预定义的参数而没有其他参数可填，所以我们在必要时需要使用 offset 来确定一个比较准确的位置）。
eg.
// seekg(10, std::ios::beg) //将读取位置移动到文件起始位置后第10个字节处。
// seekg(-5, std::ios::end) //读取位置移动到文件末尾之前的5个字节处。

所以seekg 操作实际上是在移动文件流的读取位置指针，我们显式的设置这个指针的位置。

4.result.resize(in.tellg());
这行代码将字符串 result 的大小进行调整，以便容纳整个文件。
4.1 tellg 函数用于获取当前的读取位置，返回一个整数值。此时获取了文件末尾位置，0表示不用前后移动，就返回当前位置，
所以我们返回了这个末尾位置来为 result 确定大小。

5.in.seekg(0, std::ios::beg);
这行代码将文件读取位置重新设置到文件的开头，以便从文件开头开始读取数据。
5.1 目的：设置读取位置为开始，以便接下来的 read 函数作用时从正确的位置开始读取文件。

6.in.read(&result[0], result.size());
开始读取数据。
&result[0] 是 result 的起始指针，表示将文件读到哪个缓冲区中去。

result.size() 是要读多大的文件（读多少字节数的文件），result.size() 恰好是一整个文件的大小。

7.in.close();
关闭文件流。

》》我们通过 seekg 确定了文件目前的指针位置，也就是读取时开始的位置，那读取方向由谁确定?
问题分析：
seekg 函数用于确定读取位置，而具体的读取方向则由后续的读取操作来确定。
设置方式：
读取方向是由后续的读取操作（比如 getline、read 等），通过这些函数我们设置读取方向。

》》》》关于 unordered_map 的一些使用方式。（无序映射）
使用前需要包含头文件 <unordered_map>

1.初始化

```
std::unordered_map<KeyType, ValueType> Map
或者使用初始化列表
std::unordered_map<KeyType, ValueType> Map = {
    {key1, value1},
    {key2, value2}
};
```

2.插入元素

```
Map[key] = value;  或者使用 insert 函数   Map.insert({key, value});
```

3.访问元素

```
Map[key]  也可以使用一些成员函数，进行判断：
    Count（对应某键的值的数量）
    Map.count(key)

Find（查找元素）
auto it = Map.find(key)
```

find: 返回一个迭代器，指向查找到的元素，如果未找到，则返回指向容器末尾的迭代器，即 unordered_map::end()。

4.删除元素

```
Map.erase(key);
```

5.可以用于遍历

```
for (const auto& pair : Map) { }
```

或者使用迭代器遍历

```
for (auto it = Map.begin(); it != Map.end(); ++it) { }
```

》》》》在 Windows 系统中，为什么使用 " \r\n " 来代表新行（另起一行）
\r，\n：代表回车符和换行符。
在文本操控中，\r 一般用于控制光标在文本输出中返回到当前行的起始位置，而不换行。
故在 Windows 操作系统中，使用 "\r\n" 表示文本中的一行结束，并开始新的一行。

》》》》关于 find_first_of 和 find_first_not_of 函数
1. find_first_of
原型：

```
size_t find_first_of(const string& str, size_t pos = 0) const noexcept;
```

参数：

```
str 参数是要搜索的字符集合，pos 参数是搜索的起始位置，默认为 0。
```

返回值：

```
如果找到匹配的字符，则返回该字符在字符串中的位置索引；如果没有找到匹配的字符，则返回 string::npos。
```

释义：

```
在字符串中查找第一个与指定字符相匹配的字符，并返回其位置。
```

2.find_first_not_of
原型：

```
size_t find_first_not_of(const string& str, size_t pos = 0) const noexcept;
```

参数：

```
str 参数是要搜索的字符集合，pos 参数是搜索的起始位置，默认为 0。
```

返回值：

```
如果找到第一个不匹配的字符，则返回其在字符串中的位置索引；如果没有找到不匹配的字符，则返回 string::npos。
```

释义：

```
在字符串中查找第一个与指定字符不匹配的字符，并返回其位置。
```

eg.
"a bc \n abc aabcds \r\n"
find_first_of ("\r\n"); 返回 \r\n 在文本中的索引。
↓
"a bc \n abc aabcds \r\n"

"adb ccc\n bacccab\r\n"
find_first_not_of ("\r\n"); 返回此句中第一个不是 \r\n 的字符在文本中的索引（位置）

↓
"adb ccc\n bacccab\r\n"

》》》》关于 PreProcess 函数的理解：

1.提前处理：

```
const char* typeToken = "#type";                //确定类型助记符的格式
size_t typeTokenLength = strlen(typeToken);     //确定助记符的长度
size_t pos = source.find(typeToken, 0);         //从头开始查找助记符的位置并返回
while (pos != std::string::npos)    {}          //如果找到了助记符（开始时逐行都进行判断，如果找到则进入循环。但由于循环中会对 pos 进行刷新，可以在
                                    //退出当前循环之后，通过 pos = source.find(typeToken, nextLinePos); 跳过多余的语句快速进行判断）
```

2.循环中的设计：

```
size_t eol = source.find_first_of("\r\n", pos);            //使用 find_first_of 得到助记符这一行的末尾位置
HZ_CORE_ASSERT(..., "Syntax error");                       //断言
size_t begin = pos + typeTokenLength + 1;                  //计算助记符 #type 之后的字符应该出现的位置，并储存
std::string type = source.substr(begin, eol - begin);      //复制 #type 之后的类型标识 “vertex/fragment.pixel”
HZ_CORE_ASSERT(..., "Invalid shader type specified");      //断言

size_t nextLinePos = source.find_first_not_of("\r\n", eol); //使用 find_first_not_of 得到助记符下一行的字符串开头的位置并储存
pos = source.find(typeToken, nextLinePos);                 //将 pos 更新至从新行开始后遇到的下一个助记符 #type 的位置
shaderSources[...] = source.substr(nextLinePos, pos - (nextLinePos == std::string::npos ? source.size() - 1 : nextLinePos));

//如果找不到新行，即该类型着色器没有放置代码，且位于文件末尾。则直接从助记符的开始复制直到文件末尾。（下一个提问会分析这一设计）
//如果找到新行即该着色器是有实际代码的，则直接从实际代码的开始复制到下一个助记符之前。
```

3.为什么会在复制一个着色器代码时使用 source.size 这样一个完整的大小？万一后面还有着色器呢？

前提：

实际上，我们使用 size_t nextLinePos = source.find_first_not_of("\r\n", eol);来判断一个助记符 #type vertex 这一行之后有没有字符，然后将其复制。
无论是有字符还是空行，我们都返回位置。（有新行，且新行上有实际代码，返回位置。有新行，但新行没有实际代码，也返回位置。）
至于编译结果，我们交给编译器去判断，空与不空导致的结果会是某种情况下所需要的。

分析：

但是注意看 substr 的第二个参数 pos - (nextLinePos == std::string::npos ... ， 这里的 std::string::npos 指的是完全没有任何东西，包括 '\n' '\r'等。
也就是说，如果满足此条件，则表明 #type fragment 后面已经是文件末尾，后面没有任何东西了。

```
eg.
#type vertex
vertex shader code

#type fragment
```

注意，像是：

```
eg.
#type vertex
vertex shader code

#type fragment \n
```

提示：

这样不满足 nextLinePos == std::string::npos 条件，因为其之后虽然没有实际的代码，但在 find 函数的眼中，有 '\n', '\r' 的存在，会将 pos - (nextLinePos == std::string::npos ? source.size() - 1 : nextLinePos) 判别为 pos - nextLinePos 。
（尽管这样没有什么问题，有时候着色器在调试过程中确乎不会存放代码。

总结：

但注意这个 nextLinePos == std::string::npos 条件的满足条件，所以我们才会大胆的使用 source.size 这整个读出的字符串大小来计算位置。
因为此时 #type vertex 已经位于文件末尾了）

4.一些特殊情况呢，会怎么处理?

4.1（第一种）

如果处理的字符串是第一种情况

```
“#type vertex
adfdjkl
asgdas”
```

而不是第二种情况

```
“#type vertex
adfdjkl
asgdas

#type fragment”
```

结果：

则在前二种情况的第一次循环中，pos 会在 substr 处理之前通过 find() 进行更新，在第二种情况下 pos 是一个正常位置，会正确计算。
但在第一种情况的第一次循环中，pos 会因为找不到第二个 #type 而在更新后返回 std::string::npos, 这在计算时会出现 std::string::npos - nextLinePos。

4.2（第二种）

如果处理的字符串仅仅是
"#type vertex"

结果：

则在第一次循环中，处理 #type vertex 时，nextLinePos 会在 substr 运行之前被赋值为 std::string::npos ,pos 也会是 npos，
然后在 source.substr(nextLinePos, pos - (nextLinePos == std::string::npos ? source.size() - 1 : nextLinePos));
会返回 pos - (source.size() - 1) -> pos - source.size() + 1 -> std::string::npos - source.size() + 1.

思考：

根据查证，std::string::npos 是一个常量，代表了 std::string 类型中无效的位置。当 npos 与任何数值相减时，结果仍然是 npos，因为它代表的是一个无效的位置。
这表明上述情况会导致 substr () 返回无效字符串吗？这也许是一个 bug? 这果真是一个bug吗?

4.3 问题分析

首先我们另举一个例子来看：

```
“#type vertex
dasfgda
agdsa

#type fragment
sadfsga
asdgas”
```

问题分析：

我们得知在第一次while循环中没有什么异常，但在第二次 while 循环中：

size_t nextLinePos = source.find_first_not_of("\r\n", eol); 正常运行且返回有效数值，

但是会因为 #type fragment 之后没有出现类似的 #type XXX 而导致 pos = source.find(typeToken, nextLinePos); 返回 std::string::npos

问题解决：

4.3.1.此时循环是否会提前中断?

此时，尽管循环的条件是 (pos != std::string::npos) ,但我们没有使用条件判断手动退出，所以并不会提前退出。

而是会在更新 pos 之后，继续进行之后的语句，最后在下一次循环轮询时结束循环。

4.3.2.substr 的返回值是否无效?

并不是无效的，虽然

shaderSources[ShaderTypeFromString(type)] = source.substr(nextLinePos, pos - (nextLinePos == std::string::npos ? source.size() - 1 : nextLinePos));

使用 pos 来确定代码的结束位置，且 pos 在分析后应该是 std::string::npos,

但即使 pos 是 std::string::npos，std::string::substr 也会将结束位置自动设置为字符串的末尾。

因此即使没有下一个 #type 标记，代码仍然能够正确地截取到最后一个着色器的代码。

4.3.3.为什么substr ( ) 可以返回一个有效的值?

原型：

std::string substr(size_t pos = 0, size_t count = npos) const;

定义标明：

1.如果省略 count 参数（默认 count = npos），则函数默认截取从 pos 开始直到字符串的末尾。

2.如果填满参数，则当 pos 参数大于等于字符串的长度时，substr 会返回一个空字符串。

（人话：从整个字符串的末尾甚至更靠后的地方开始复制，这后面肯定没有字符了）

3.而如果 pos 参数小于字符串的长度但 count 参数过大导致超出字符串末尾时，substr 会自动将 count 调整为使得截取不超出字符串末尾的最大值。

（人话：开始位置正确，但是要复制的长度被设置的超出了字符串本身的大小，则截取字符串全部）

结论：

因此，当 count 参数被我们计算得到的值为 std::string::npos 时，substr ( ) 实际上会默认截取从 pos 开始直到字符串的末尾。

这就是为什么即使在没有下一个 #type 标记的情况下，代码仍然能够正确地截取到最后一个着色器的代码。

》》》》关于对一个指针类型变量使用 reset() 和 直接传递一个返回对应指针类型函数 这两种方法的不同。

eg.

m_TextureShader.reset(Nut::Shader::Create(textureVertexSrc, textureFragSrc));

m_TextureShader = Nut::Shader::Create(textureVertexSrc, textureFragSrc);二者的区别。

区别：

reset() 会在重新赋值之前释放原有的资源，从而避免内存泄漏。是一个安全的成员函数。

而直接使用 Create() 返回的指针（将指针赋值给一个改变量）并不会自动的进行资源释放，需要自己定义资源的正确释放。

》》》》发现 Cherno 的一个小错误

代码：

在PreProcess 函数中，Cherno 这样定义

```
while (pos != std::string::npos)
    {
        size_t begin = pos + tokenLength + 1;
        size_t end_of_line = source.find_first_of("\r\n", pos);
        NUT_CORE_ASSERT( end_of_line != std::string::npos, "Syntax error");
```

问题：

但是正常使用的时候，我发现 end_of_line 应该是一个有效的正确的值，却一直触发断言。

(end_of_line != std::string::npos) 返回的是一个 bool 量，其值确实为 true, 触发断言的原因是格式的错误。

改正：

正确形式应该是

NUT_CORE_ASSERT((end_of_line != std::string::npos), "Syntax error");

而不是

NUT_CORE_ASSERT(end_of_line != std::string::npos, "Syntax error");

原因：

如果不加这个括号，宏会被展开为

```
{
    if ( ! end_of_line != std::string::npos )
    { /* ... */ }
}
```

因为 ! 的优先级高于 !=，所以会变成if ((! end_of_line) != std::string::npos),这样一来即使 end_of_line 是 true（有效值），

!(end_of_line) 也会是 false，然后 false != std::string::npos 会是 true，这会导致断言的触发，而实际上代码并没有问题。

修改后则不会出现这种 end_of_line 被修改了的情况。

----------Shader Library（着色器库）----------

》》》》关于 Cherno 提到的问题

std::vector<GLenum> glShaders(shaderSources.size());　　//逻辑上是 resize

会导致初始化动态数组的时候提前新增两个空白的位置，而之后的 glShaders.push_back(shader); 才会推入正确的数据，而且位置位于前两个空白之后，

解决方式：
std::vector<GLuint> glShaders;
glShaders.reserve(shaderSources.size());　　//逻辑上是 reserve
这样可以为数组在内存中先分配确定的内存（先预留两个位置），当然也可以在两个位置之后继续动态拓展。
但并不用空白数据初始化这两个位置的内存，而是交给后面push_back进来的数据填充。

》》》》在临摹代码之前，我先将一些亟待解决的代码细节优化做了。以下是一些笔记。

》》(const void*)element.Offset 和 (const void*)(intptr_t)element.Offset的区别
(const void*)(intptr_t)element.Offset：
这个表达式先将 element.Offset 转换为 intptr_t 类型，然后将其强制转换为 const void* 类型。
其中intptr_t 是一种整数类型，足够大以容纳指针的位模式。（这样确保了数据传递和使用的正确性）

》》src/Nut/Input.h 中的更改是什么？为什么？
为什么：
首先，这个类是一个用于判断输入的抽象的基类，包含了一些序虚函数，作为接口被使用。
并且，这个类也被声明为单例类，意为这不能被实例化。（就算被实例化，这个对象也不包含任何数据，只包含能使用的接口）

是什么：
所以我们定义并禁用了两个函数：复制构造函数和赋值运算符
eg.
```
Input(const Input&) = delete;
Input& operator=(const Input&) = delete;
```
1.
= delete 关键字表明这两个函数是被删除的，也就是说被禁止使用。
2.
Input(const Input&) = delete;
复制构造函数在对象被复制时调用，用于将一个对象复制到另一个对象中。
通过将其定义为 = delete，编译器会禁止复制构造函数的使用。即禁止任何尝试复制该类对象的行为，从而确保该类对象不会被意外地复制。
3.
Input& operator=(const Input&) = delete;
赋值运算符用于将一个对象赋值给另一个对象。
通过将其定义为 = delete，编译器会禁止赋值运算符的使用。即禁止对该类对象进行赋值操作，从而确保对象不会被错误地赋值。

GameEngine4

2023年12月3日 21:48

----------Shader Library（着色器库）----------

》》》》终于开始着色器库的设计，来看看着色器库是用来干嘛的。
着色器库的设置是静态的，用于自动加载着色器内容、诊断着色器。
高度抽象的设置可以为我们自动化识别接口，使在程序中的调用更加简洁，同时也隐藏了一些细节（自动化处理，不用手动显示设置）

》》》》以下是一些设计中的理解：

》》在 ShaderLibary 中，使用之前定义的指针 Ref 和 Scope，不需要包含头文件即可使用，为什么。
原理：
命名空间的作用域在整个程序中都有效，只要命名空间被正确声明和定义，其成员在程序的任何地方都可见。

不使用命名空间的话，直接在代码中使用 Nut::Ref 也是可行的，

》》关于Cherno对于从文件名截取着色器名称的手法，在 C++ 版本的逐步发展中，还有其他更便捷的方法。
原版：

```
// Get Shader's name though filepath name
// maybe :        1.assets/textures/shader.glsl        2./shader       3.shader.glsl    4.shader

size_t lastSlash = filepath.find_last_of('/\\');
lastSlash = (lastSlash == std::string::npos ? 0 : lastSlash + 1);
size_t lastDot = filepath.rfind('.');
lastDot = (lastDot == std::string::npos ? filepath.size() : lastDot);

size_t count = lastDot - lastSlash;
m_Name = filepath.substr(0, count);
```

新版：（C++ 17）

```
#include <filesystem>

std::filesystem::path path = filepath;
m_Name = path.stem().string(); // Returns the file's name stripped of the extension.
```

截取自评论：
@This new feature is very useful for use in Asset Managers as well which I'm currently implementing in my project, it support s native file path directory seeking.
@And it is bulletproof. His code isn't. If the path is "../testures/texture", the count will be negative, because the last do t is before the last slash. And this is a valid path.

----------How to build 2D engine----------

没什么要记的，后面也都会涉及。这一集就是一个大概思路。

----------Camera Controllers----------

》》》》大致思路
将实际控制通过 Camera Controller 来调用，而不是像以前一样通过 Camera 类来直接的修改和更新 Camera 的值。
现在的 Camera 可以认为是一个深层次的、存放了一些计算方法的库，controller 会调用这些方法，而用户只需要使用 Controller 来进行操纵。

》》》》std::max( )?
std::max 是 C++ 标准库中的一个函数模板，用于比较两个值，并返回其中较大的值。它定义在 <algorithm> 头文件中。

》》》》claudiaIDE 2019
一个可以更改 VS2019 背景的插件，这就是我昨天没有更选代码的原因。因为我去折腾壁纸了。

----------Resizing----------

》》》》minimized 这个变量的意义？
举个例子：
比如你在打一个游戏（英雄联盟），现在你打开了购买装备的界面，然后你又做了切换应用的操作。
此时游戏会被最小化到后台，购买界面也会随游戏被最小化。也不可以在脱离游戏界面时被操作。
只有你重新进入游戏界面，继续运行游戏时，这个购买界面才可以被响应（进行购买、关闭等操作）

》》》》关于函数设计时的思考。
application 中的 OnWindowResize 是为了让视口填充整个窗口。

OrthoGraphicCameraController 中的 OnWindowResized 是为了让视口中的物体在窗口大小调整时候比例自适应，而且呈现正确的效果。

但是函数

```
bool OrthoGraphicCameraController::OnWindowResized(WindowResizeEvent e)
{
    m_AspectRatio = (float)e.GetWidth() / (float)e.GetHeight();            //设置回调的宽高比
    m_Camera.SetProjectionMatrix(-m_AspectRatio * m_ZoomLevel, m_AspectRatio * m_ZoomLevel, -m_ZoomLevel, m_ZoomLevel);
    return false;
}
```

是这样设计的，

所以 m_AspectRatio 会随 width（分子） 的大小调整正确变化，但是与 Height（分母）的大小变化刚好相反。

而且由于 m_AspectRatio 是 width 除以 height ， 所以当使用鼠标在窗口对角进行操作时（即对 Width 和 Height 同时进行改变），渲染的物体比例不变。

这是一个待处理的瑕疵。

---------- maintenance ----------

》》》》pushd 和 popd 的使用

pushd：	将当前目录入栈，并切换到指定的目录。
popd：	从栈中弹出最近保存的目录，并切换到该目录。

eg.

::使用 pushd 命令切换目录并将当前目录推入栈中：

pushd <目标目录>

::使用 popd 命令从栈中弹出最近保存的目录并切换到该目录：

popd

---------- preparing for 2D rendering ----------

》》》》没什么要记的，但是我顺手同步跟新一下错误修复的 submit ，并做笔记。

》》》》gitignore 规范

.gitignore 文件的规范是通过简单的文本格式来定义忽略规则。每个规则占用一行，用于指定要忽略的文件、文件夹或者特定模式。

文件匹配：

使用斜杠（/）	表示路径分隔符。
使用星号（*）	表示匹配任意数量的字符（除了路径分隔符）。
使用双星号（**）	表示匹配任意数量的字符（包括路径分隔符）。
使用问号（?）	表示匹配一个任意字符。
使用感叹号（!）	表示不忽略的文件或文件夹。

注释：

使用井号（#）	开头的行表示注释，这些行可能会被Git忽略，一般另起一行来写。

注释可以写在规则的上方，用于对规则进行解释或提供其他相关信息。

eg.

```
# 忽略所有的编译输出文件
*.o
*.exe
*.dll

# 忽略bin文件夹及其下的内容
/bin/

# 忽略.vscode文件夹
.vscode/

# 不忽略lib文件夹下的example.txt文件
!lib/example.txt
```

》》*bin\和**bin\之间的区别：

***bin**：表示匹配当前目录下的任意一级子目录中的bin文件夹。例如，src/bin/、lib/bin/等。

****bin**：表示匹配当前目录及其所有子目录中的bin文件夹。例如，src/bin/、src/utils/bin/、lib/bin/等。

》》\bin\和bin\在.gitignore文件中的区别：

****\bin**：这个规则表示匹配当前目录及其所有子目录中的bin文件夹。例如，src/bin/、src/utils/bin/、lib/bin/等都会被匹配到。

****bin**：这个规则表示匹配当前目录及其直接子目录中的名为bin的文件夹。例如，src/bin/、lib/bin/会被匹配到，但是src/utils/bin/不会被匹配到。

》》vs\和.vs\的区别：

从 '\' 看出这都表示忽略文件夹（只不过名称不一样，'.'开头的一般是隐藏文件夹）

》》》》什么是 constexpr ?

constexpr是C++11引入的一个关键字，用于声明可以在编译时求值的常量表达式。它可以用于变量、函数、构造函数等的声明中。

》》constexpr中所有参数必须是字面类型。那

```
template<typename T>
using Scope = std::unique_ptr<T>;
template<typename T, typename ... Args>
constexpr Scope<T> CreateScope(Args&& ... args)
{
    return std::make_unique<T>(std::forward<Args>(args)...);
}
```

这个函数声明为什么是正确的呢?

因为：C++17对constexpr函数进行了一些扩展，放宽了函数参数的限制。

如果函数的参数满足以下条件之一，那么即使函数本身不是字面量函数，也可以声明为constexpr函数：

1.参数类型是字面类型；

2.参数是指向字面类型的指针或引用；

3.参数是数组类型，并且其元素类型是字面类型。

具体分析：

在CreateScope中，虽然模板参数Args可以包含任意类型的参数，但在调用std::make_unique<T>时，参数是通过完美转发传递的

因此参数类型和值在编译期是可以确定的，可以满足constexpr函数的要求。

》》》》constexpr （常量表达式）的好处。

constexpr的好处：提高程序的性能、安全性和可维护性，尤其适用于需要在编译期确定结果的函数。

》》》》template<typename T, typename ... Args> 模版中的 typename ... Args是什么?

》》》》CreateScope(Args&& ... args)中的参数是什么?

typename ... Args 是一个模板参数包，意味着 Args 是一个包含零个或多个模板参数的集合。（可以是任意数量的其他类型。）

Args&& ... args 中的 Args&& 是一种右值引用折叠语法。表示将模板参数包 Args 中的每个参数都作为右值引用传递给函数。

其中：

Args	是一个模板参数包。
&&	表示右值引用，表示参数 args 是右值引用类型。
...	则表示参数包展开，即将参数包中的每个参数都单独地传递给函数。
args	是函数的参数名。在函数中，它代表了接受模板参数包 Args 中传递进来的参数。

》》》》在函数的返回值中，我发现了 std::forward 这是什么东西？怎样使用?

概念：	**s**td::forward 是一个 C++ 标准库中的函数模板，用于实现完美转发（Perfect Forwarding）。
作用：	std::forward 允许在函数模板中保持 被传入的参数的类别（左值或右值）和常量性。 通过将参数以原始的值类别（左值或右值类型）传递，来避免不必要的拷贝和转换，提高程序的性能和效率。 // 后面会涉及到一些概念，先将笔记看下去 ~

参数与返回值:	它接受一个参数，并返回相同类型的参数，并且根据参数的值类别（左值或右值）不同，它会将参数转发为左值引用或右值引用。 （以此确保参数值属性的正确传递）

使用：

在例子 std::forward<Args>(args)... 中： <Args>指定了传递给 std::forward 函数的参数类型，(args)...指定了传入的每个参数。

结果：

整个表达式的作用是将模板参数包 args 中的每个参数都通过 std::forward 转发给其他函数，并保持其原始的值类别。

这样就实现了完美转发的效果，使得参数在传递过程中保持了原始的左值或右值特性。

》》什么是完美转发？什么是左值引用？什么是右值引用?

--- 0. 值的类别（左值、右值）

左值（Lvalue）

概念：	有名称 / 在内存中有确定位置的表达式或对象。
意义：	可以被引用、修改和取地址。

eg.	int x = 5; 中的x就是一个左值。

右值（Rvalue）

概念：	不具有名称 / 在内存中没有确定位置的临时表达式或对象。
意义：	不能被引用，只能被移动或复制。
eg.	int y = 3 + 2; 中的3 + 2就是一个右值。

--- 1.引用的两种类型（左值引用、右值引用）。这两种引用基本是对左右值用法的拓展，其本旨与左右值相像。

左值引用：

概念：	使用 & 符号声明的引用类型。
意义：	表示对一个命名对象的引用，该对象可以被修改。
eg.	int x = 10; int& y = x;

右值引用：

概念：	使用 && 符号声明的引用类型。
意义：	表示对一个临时对象或将要销毁的对象的引用，该对象不能被修改。
eg.	int&& z = 20;

--- 2.
完美转发：完美转发是一种技术，通常涉及使用 std::forward 函数模板来将参数转发为左值引用或右值引用。
在将各种类型的参数传入 std::forward 的过程中，保留原始的值的左右值特性，避免不必要的拷贝和转换，提高程序的性能和效率。

》》》》如果在 return 中不使用 std::forward 呢，有什么结果?

return std::make_unique<T>(std::forward<Args>(args)...);
return std::make_unique<T>(args...);
虽然后者没有错误，但是会丧失参数原本的语义。这会带来性能上的损失，尤其是在处理大型对象时。

》》为什么说会导致性能的损失呢?
一般来说，在传入具有左右值属性的参数时，编译器会根据左右引用的不同属性选择动态转移或复制(拷贝)来传递。也就是移动语义和复制语义。
（通常，右值引用可以触发移动语义，左值引用可以触发复制语义）

如果不使用 std::forward 来保证左右值属性的正确传递，将会导致参数传递时丧失了对应的语义。这意味着即使传递的是右值，也会进行复制构造，而不是移动构造。
然而复制构造需要分配额外的时间和内存来进行深度复制操作，这会消耗相对较大的内存储备，降低性能表现。

》》》》什么是移动语义和复制语义？为什么右值触发移动语义，左值触发复制语义?

在C++中，值的语义（复制或移动）和值的类别（左值和右值）是密切相关的。

复制语义：

概念：	在将一个对象传递给另一个对象时，会创建该对象的一个全新的、独立的拷贝，两者之间没有关联。
实现方式：	通过拷贝构造函数来实现。

移动语义：

概念：	它允许在不复制内存的情况下将对象从一个位置（例如临时对象）转移到另一个位置。它适用于临时对象或者不再需要的对象。
不同之处：	移动语义将资源的所有权从一个对象转移到另一个对象，而不是创建资源的拷贝。
实现方式：	通过移动构造函数和移动赋值运算符来实现。

开销：

复制语义（大）	分配额外的内存和时间来进行深度复制，这会消耗大量的内存，尤其是对于包含大量数据成员的对象（包含大量数据）。
移动语义（小）	显著提高程序的性能和效率（特别是动态分配资源时）。

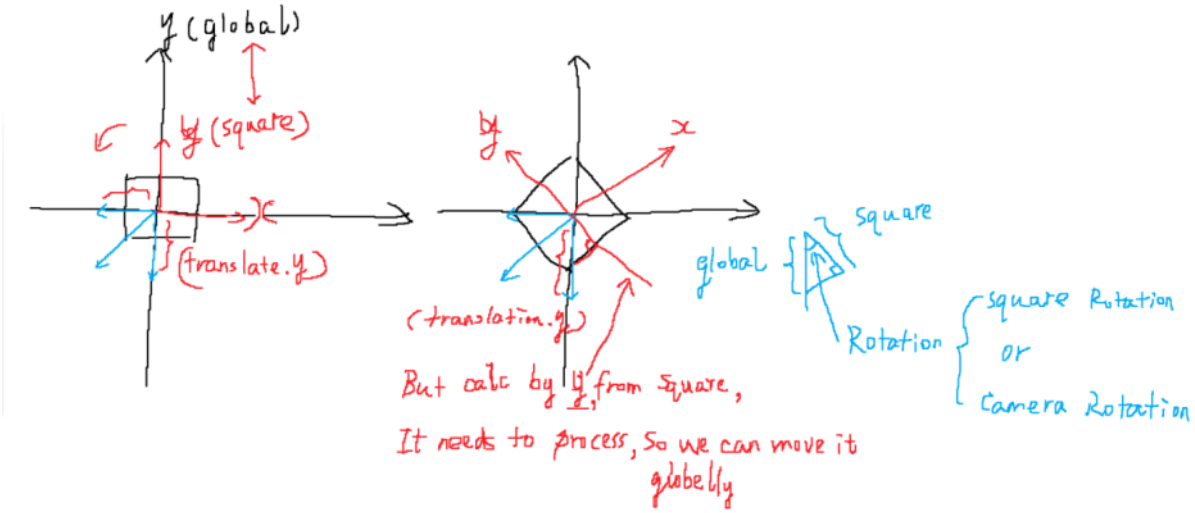
右值引用触发移动语义：

原因：	右值引用通常绑定到临时对象或将要销毁的对象上，这些对象不再需要原来的值，因此可以使用移动语义安全地将其资源移动到新的对象中。
意义：	这可以在处理大型对象或需要频繁传递所有权的情况下提高程序性能。

左值引用触发复制语义：

原因：	左值引用通常绑定到具名对象上，这些对象仍然需要保持原来的值。因此，对左值引用进行操作时会触发复制语义，即将原来对象的值复制到新的对象中。
意义：	这样对象的值也不会被意外地修改。

关于通过物体在物体（局部）的坐标轴上的偏移量 计算全局坐标偏移量的理解。



》》》》**在宏定义中，#ifdef _WIN32 这个条件能够在32位操作平台上被自动触发，为什么?**

_WIN32	是一个预定义的宏，通常由Windows的编译器在编译过程中自动定义。

这个宏用来标识当前代码是否在Windows平台上编译运行的一个条件宏，由编译器根据编译环境自动设置。

》》》》**在 .gitmodules 文件中，branch = XXX 这个指令的操作是什么意思？有什么作用?**

作用：	指定每一个子模块在项目中使用的分支名称。
意义：	对于每个子模块，都可以指定使用的特定分支，这样可以确保每个子模块都使用了正确的代码版本。 可以保证主项目和子模块之间的关联始终指向相应的稳定版本或者需要的特性分支。

#define EVENT_CLASS_TYPE(type) static EventType GetStaticType() { return EventType::##type; }\
和
#define EVENT_CLASS_TYPE(type) static EventType GetStaticType() { return EventType::type; }\
这两段代码在使用上有没有什么不同?

虽说这两个宏定义都是在填入 type 的时候定义一个函数 GetStaticType() 并返回 EventType,但是不同之处在于 ##
1.有 ## 时，填入的 type 会被拼接在 EventType 之后，并在预处理阶段动态的合成，结果会作为一个临时生成的类型，并返回。
2.无 ## 时，填入的 type 会根据填入的 type 返回一个事件类型，但此时填入的 type 如果生成的结果是没有被设置过的一种 type，会导致返回报错，因为 EventType::type 并不一定存在。
而不是像上面的那样返回一个临时类型，而无论正误。

》》》》模板中的参数包是什么，怎样使用？什么是参数包展开？ 参数包和完美转发的关系是什么?

- **参数包（Parameter Pack）** 是C++11及以后版本中模板元编程中的一个特性，它允许你定义一个可以接受任意数量模板参数的模板类或模板函数。
  参数包用...来表示，它可以与类型参数（typename... Args）或值参数（auto... args 或 Args... args）一起使用。
- **参数包展开（Parameter Pack Expansion）** 是对参数包中每个元素进行操作的语法，它使用...来指示编译器对参数包中的每个元素执行的操作。

  通常与模板函数和模板类中的参数包一起使用。
- **操作规则**：
- ***值参数包***：
  值参数包用于模板函数，允许函数接受任意数量的值作为参数。例如：

  template<typename... Args>

  void foo(Args... args) {

  // 在这里可以对 args 进行操作

```
}
```

args 是一个值参数包，它可以接受任意数量和类型的值作为函数参数。

- ***参数包展开***:
  参数包展开通常与递归模板或函数一起使用，以便对参数包中的每个元素执行操作。在函数模板中，可以使用递归终止函数和递归函数来展开参数包。例如，下面的函数使用递归模板来打印参数包中的所有值：

```
template<typename First, typename... Rest>
void print(First value, Rest... rest) {
    std::cout << value << std::endl;
    Function(rest...); // 参数包展开
}
```

在这个例子中，print 函数首先处理参数包中的第一个值，然后调用Function来处理剩余的值（通过参数包展开）。

- ***std::forward和完美转发***:
  当处理值参数包时，经常需要保持原始参数的左值或右值性（lvalue or rvalue）。std::forward 函数模板用于在模板函数中完美转发参数，它接受一个参数类型和参数本身，并返回该参数的正确引用类型（左值引用或右值引用）。

```
template<typename... Args>
void wrapper(Args&&... args) {
    foo(std::forward<Args>(args)...); // 完美转发参数包
}
```

在这个例子中，wrapper 函数接受任意数量和类型的参数，并使用 std::forward 将它们完美转发给 foo 函数。

<TargetConditionals.h>文件是什么？有什么作用?
苹果提供的 SDK 中的 usr/include/TargetConditionals.h 文件，会自动配置编译器编译的代码所需要使用的微处理器指令集、运行系统以及运行时环境。

<TargetConditionals.h> 适用于所有的编译器，但是它只能被运行于 Mac OS X 系统上的编译器所识别。