**项目总体结构**

# **Hazel —— 引擎核心代码，编译为静态库，链接到Hazel-Editor**

* Core，共用的一些核心功能，重要内容包括Application（应用程序单例），Log（日志系统），Layer和LayerStack（层和层栈，管理事件传递和游戏更新），Window（窗口抽象类）。
* Event，事件系统。
* ImGui，定义UI层的行为。
* Math，数学工具。
* Renderer，渲染器（重要），涉及到图形API的封装，渲染合批等。
* Scene，场景系统，涉及到实体、组件、序列化等。
* Scripting，脚本系统，与Mono配合。
* Utils，提供平台无关的API。
* Platform，平台层，为每种图形API和每个平台提供具体实现，目前只支持OpenGL和Windows系统。
* vendor，引擎核心代码用到的第三方库，大部分是直接使用源码，与项目一起编译，编译为静态库后链接到Hazel静态库。

# **Hazel-Editor —— 引擎编辑器，编译为可执行程序。**

* SandboxProject，C#项目，内部有用户自定义的脚本，这些脚本会被编译为动态库，编辑器在运行游戏时会加载这些脚本中的内容，执行用户的自定义逻辑。
* src，编辑器界面的代码以及编辑器层EditorLayer，启动时会加入LayerStack，执行编辑器的更新逻辑和UI渲染。HazelEditorApp继承Application，创建唯一的应用程序单例。

# **Hazel-ScriptCore —— 引擎脚本系统，调用引擎的C++代码，并提供C#API，供Editor层的脚本使用。**

C#项目，类似于Unity为用户提供的C#接口，内部有Component，Input，Vector等攻击，这些脚本会被编译为动态库，随程序运行时加载。SandboxProject会引用这个项目，使得用户自定义的逻辑可以使用核心脚本中的API，类似于在Unity中编写自定义脚本需要Using UnityEngine。

**功能梳理**

# **程序入口（EntryPoint）与测试沙盒（Sandbox）**

引擎相关代码写在Hazel项目中，测试代码写在Sandbox项目中（Sandbox项目最后弃用，使用Hazel-Editor项目充当该角色），Sandbox项目为启动项目，相当于引擎的前端。程序入口点（EntryPoint.h，内含main函数）为一个头文件，位于Hazel项目中，Sandbox项目引用这个头文件，这样做的用意是：

**1. 核心代码的复用**

Hazel项目被设计成一个独立的库，它包含所有的核心逻辑。Sandbox 作为一个测试或示例项目，它可以直接链接 Hazel库，Hazel代码可以被多个不同的项目复用，而无需在每个项目中重复编写或复制核心功能。

**2. 分离逻辑和测试/运行环境**

* Hazel项目专注于核心功能的开发，不涉及具体的应用场景，例如 UI、输入等。
* Sandbox 主要用于测试 Hazel的功能， 这样可以保证 Hazel代码的纯净，减少对外部环境的依赖，使其更易于维护和移植。

**3. 程序入口点作为头文件**

由于 Sandbox 只是 Hazel的一个测试环境，它可能并不真正控制程序的入口逻辑，而是希望 Hazel定义应用的启动方式。这样，Sandbox 只需要关心 Hazel提供的接口。

**4. 便于不同的前端环境（如引擎编辑器）调用**

* 如果 Hazel作为一个游戏引擎核心，可能会有多个不同的前端：
  + Sandbox：用于调试和运行单个游戏实例
  + Editor：用于编辑器模式
* 通过这种 EntryPoint.h 方式，不同的前端可以共享 Hazel的启动逻辑，而又能够以不同的方式初始化自己的环境。

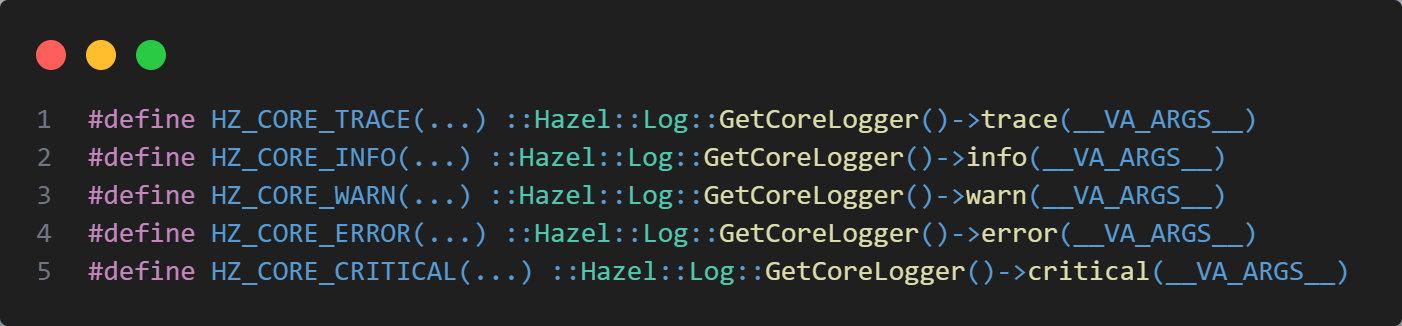
# **日志系统**

* 简单封装了spdlog库，提供CoreLogger和ClientLogger，Hazel项目使用CoreLogger，打印的日志会有HAZEL前缀，测试项目使用ClientLogger，打印的日志会有APP前缀。
* 提供宏简化使用，共有5个等级，从上到下重要性递增。

细节：

1、宏变长参数。…表示宏有任意数量的参数，\_\_VA\_ARGS\_\_会被替换为宏调用时传入的参数。

2、Hazel前的双冒号。表示在全局命名空间下寻找，防止错误。例如，在APP命名空间内有一个Hazel类，如果Hazel前没有双冒号，那么使用该宏时，就会找到APP命名空间下的Hazel类，导致错误。



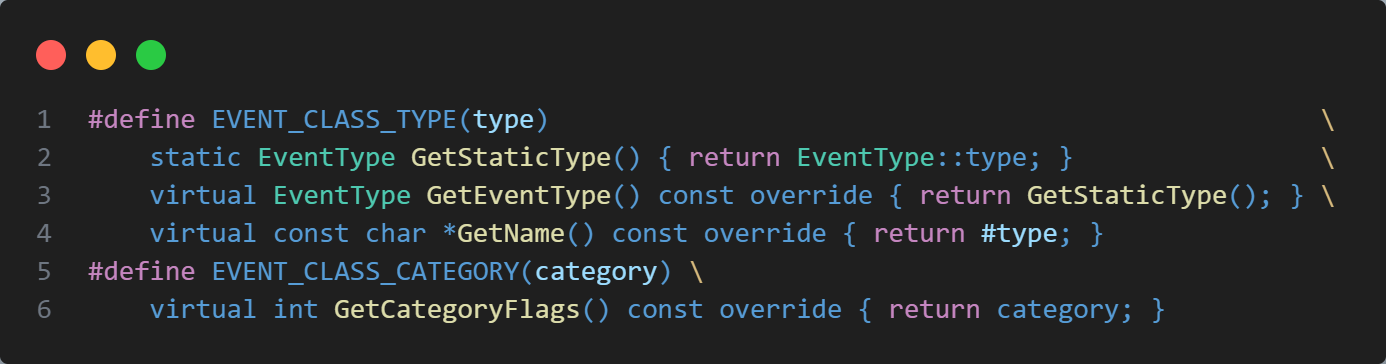
# **事件和窗口**

## **事件系统**

* 事件抽象类Event

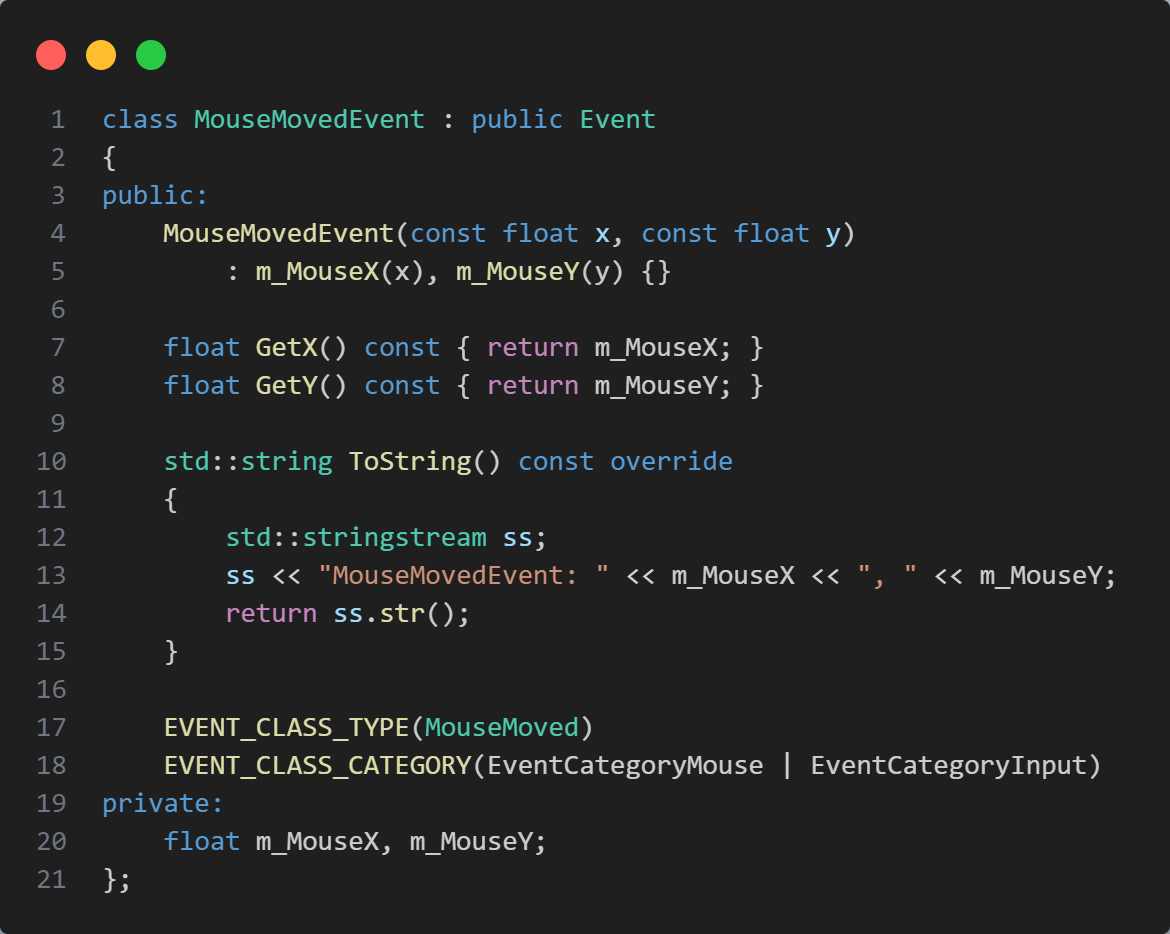
提供访问事件Type和事件Category的接口，其中事件Type代表具体的类型，例如鼠标按下，键盘抬起等；事件Category是事件的类别，例如键盘事件、输入事件，采用位运算管理。例如按下键盘，既是键盘事件，又是输入事件，可表示为EventCategoryKeyboard | EventCategoryInput。

事件抽象类中的每个接口都要重写，使用宏避免大量重复的代码。#type表示将传入的宏参数转化为const char\*。事件的派生类只需使用这两个宏就可以定义自身的具体类型和事件类别。



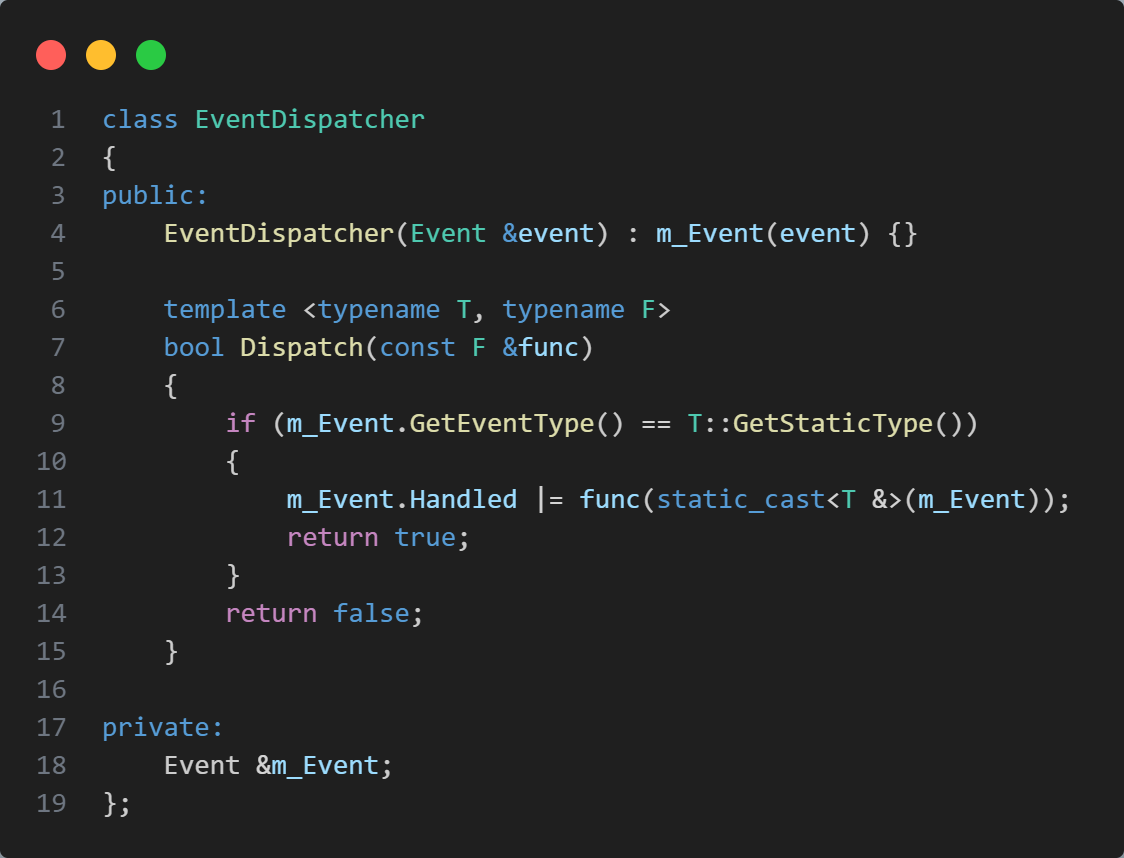
* 事件具体类

分为鼠标事件、键盘事件、应用程序事件，它们都继承事件抽象类Event，并扩展自己的逻辑，下面是鼠标移动事件的例子。

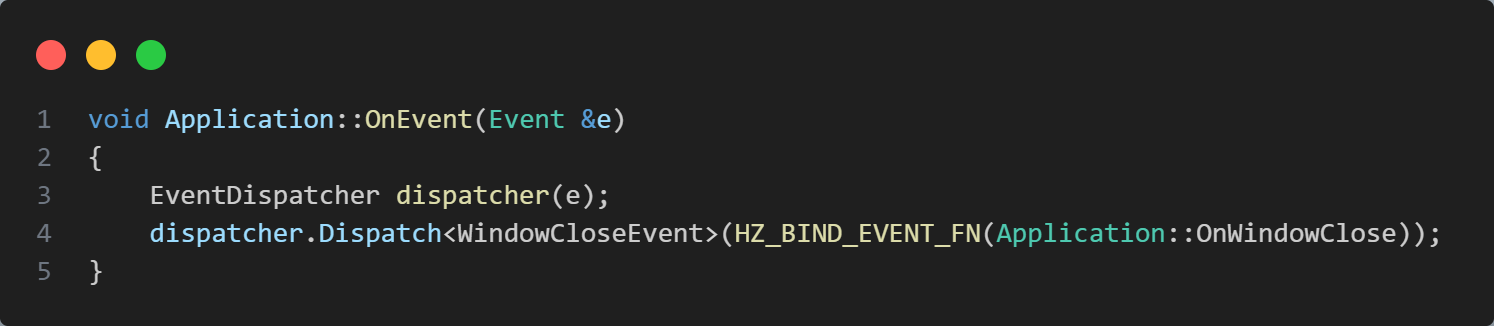


* 事件转发器EventDispatcher

在用到事件系统的地方使用事件对象初始化EventDispatcher对象，根据事件类型是否与转发器的模版参数匹配来决定是否处理该事件。Dispatch函数接受一个可调用对象作为事件处理回调函数，返回true代表该事件被捕获。这个传入的可调用对象必须返回一个bool值，如果可调用对象返回true，Handled会置为true，意思是希望将事件的传播阻断在本层；如果可调用对象返回false，意思是事件可以继续传播。例如我们不希望UI层的事件传递到场景层，那么UI层的事件处理回调函数就要返回true，接下来可通过Handled决定是否让事件继续传递。



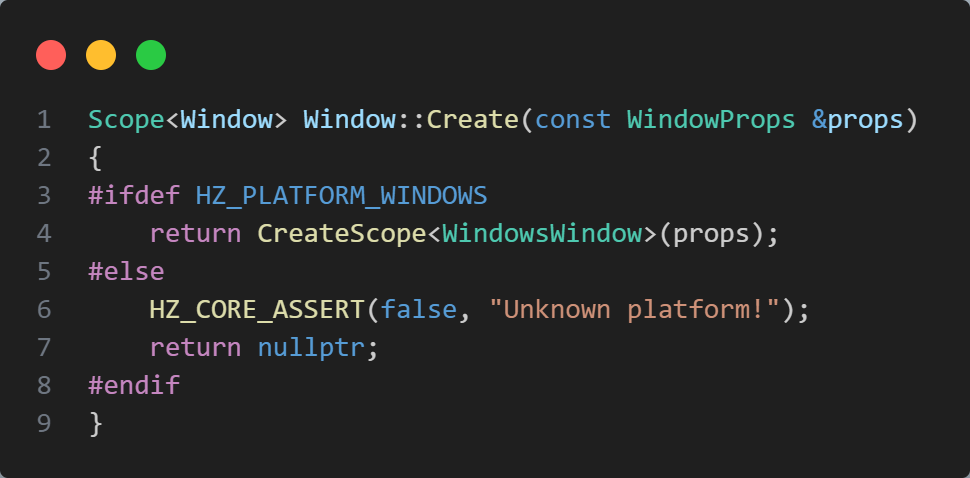
使用示例：EventDispatcher对象希望捕获一个WindowCloseEvent事件，事件发生时回调Application::OnWindowClose函数。事件发生时，Dispatch函数内部会判断该事件是不是WindowCloseEvent事件，如果是，则捕获成功，回调Application::OnWindowClose函数。



## **窗口系统**

* 抽象窗口

定义了一系列平台无关的API，例如获取窗口宽高，设置我们自定义的回调函数。回调函数形式是参数为Event对象，返回值为void。在平台层可以继承该类，并定义具体的窗口，例如Windows系统使用Windows系统原生窗口，苹果系统使用苹果系统原生窗口。本项目暂时只支持Windows系统，并使用GLFW作为窗口。在创建窗口时，使用宏进行选择性编译，创建对应平台的窗口，如下所示：



* 具体窗口（平台特定窗口）

暂时只支持Windows系统，具体窗口为GLFW。内部有一个WindowData结构体，含有窗口的标题、宽高、回调函数。在Init函数中初始化glfw窗口，并使用glfwSetWindowUserPointer(m\_Window, &m\_Data)定义窗口用户指针，方便接下来glfw窗口的原生回调函数的注册。注册回调函数时，只需使用glfwGetWindowUserPointer取出WindowData结构体指针，并调用我们自定义的回调函数。注册glfw回调的示例如下，当发生窗口Resize的事件时，glfw会响应Lambda表达式内的内容，取出WindowData结构体指针，重新设置窗口的宽高，并产生一个WindowResizeEvent事件，以该事件作为参数，调用我们自定义的回调函数。

