UE4 RHI与原生OpenGL的层层调用

简介

在通用游戏引擎架构中，我们能看到引擎在底层架构中有一个平台独立层，旨在让游戏引擎能在多个平台上实现通用功能，并且保证包装接口在所有硬件平台上均为一致。在本文中我们将自底向上地关注UE4是如何为了达成同样的目的专门对DX11、DX12和OpenGL等渲染平台抽象出相同的接口，以此方便引擎渲染模块对渲染的操作。UE4将这个抽象接口称为*Render Hardware Interface (RHI)*。因此我们不会在引擎实现图像渲染功能时直接看到对渲染接口的调用。

目录

UE4编码基础

UE4引擎中的模块其实就是一系列源文件，可能是C++模块或者是C#模块。其中虚幻四的模块主要由自定义的C#脚本——“*.Build.cs" 来定义。由此我们可以得知，创建一个C++模块的规则就是，在文件夹中添加一个模块结构并用”.Build.cs“来管理模块之间的依赖关系等，其典型结构如下：

```
using UnrealBuildTool;
using System.Collections.Generic;

public class MyModule : ModuleRules
{
    public MyModule(ReadOnlyTargetRules Target) : base(Target)
    {
        // Settings go here
        // Dependencies
        PrivateDependencyModuleNames.Add("HTML5JS");
        PublicDependencyMOduleNames.AddRange(
            new string[] {
                "DirectXMath",
            };
        )
    }
}
```

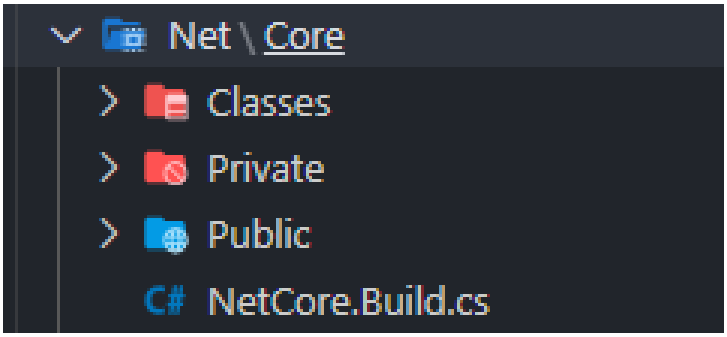
UBT能够根据Target的定义来构建多个不同的目标类型：Game，Client，Server，Editor和Program。Program主要指构建在虚幻引擎上的独立实用程序类似插件等。其中每一个".Target.cs" 主要定义当前Program的特性，其典型结构如下：

```
using UnrealBuildTool;
using System.Collections.Generic;

public class MyTarget : TargetRules
{
    public MyTarget(TargetInfo Target) : base(Target)
    {
        // Settings go here
    }
}
```

根据以上概念，我们不难发现UBT总是被优先编译出来的原因。

模块结构



当前模块Net中很明显有三个子文件夹，分别是*Classes*, *Private*和*Public*。

- **Class文件夹**算是历史遗留问题，因为在旧版本中所有UObject和其衍生类的声名都是在Classes文件夹中。但现在这个限制已经不存在了。
- **Public文件夹**的作用就是当该模块被其他模块依赖时，该模块的Public文件夹路径会被其他模块自动添加为“include path”。
- **Private文件夹**与Public文件夹相对，存放不需要暴露的实现细节如源文件和预编译头。
- ***.Build.cs文件**储存配置参数用于指导UBT处理模块。

> 并不是Public下放头文件，Private下放实现文件，而是Public放需要暴露给别人的头文件,其他不需要暴露给其他人的头文件和实现文件放到Private目录下。
> Private文件夹并没什么特殊性，只是习惯性地被拿来与Public成对使用而已，并非强制的，你使用Pipixia放私有文件都可以。
> 游戏模块（非插件模块），一般来说也不会被其他模块依赖，因为游戏模块基本就是所有其他现有模块的“使用者”，所以你会看到网上某些游戏模块的源码结构根本不遵守Public/Private分离的惯例，对于游戏模块来说，这是可取的，但不推荐，因为你的游戏模块可能会有一个对应的编辑器模块，而这个编辑器模块一般是要依赖你的游戏模块的。

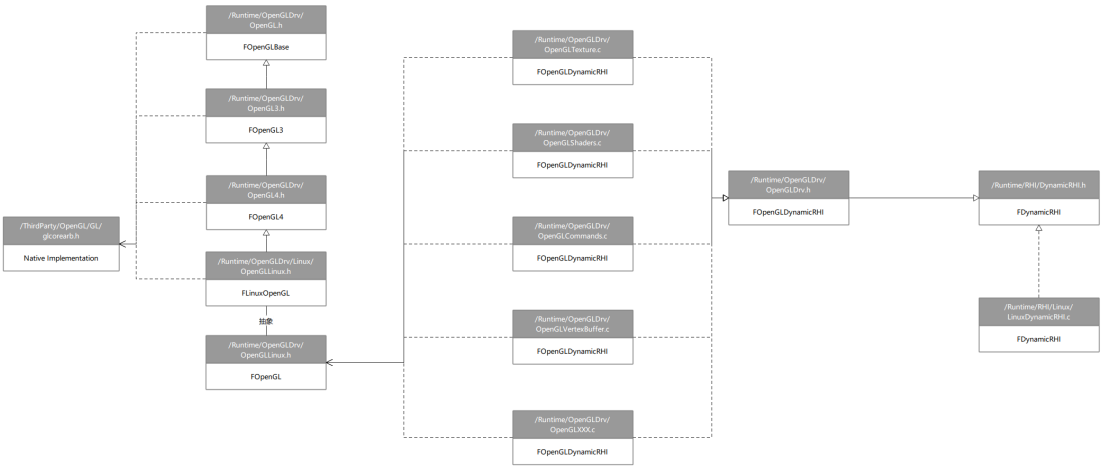
命名规范

接下来我们要准备看一些源码，那就需要了解一下引擎中变量或者函数的命名规范。引擎中的变量主要由第一个字母的前缀来表征它的用途，当前主要分为六类：

- 派生自Actor类——前缀为A，如AController
- 派生自UObject类——前缀为U，UContextReader
- 派生自Swidget（Slate UI）类——前缀为S，
- Enums的前缀为E
- Template前缀为T，如TArray，TChar
- 其余类前缀均为F

> Reference

依赖关系图

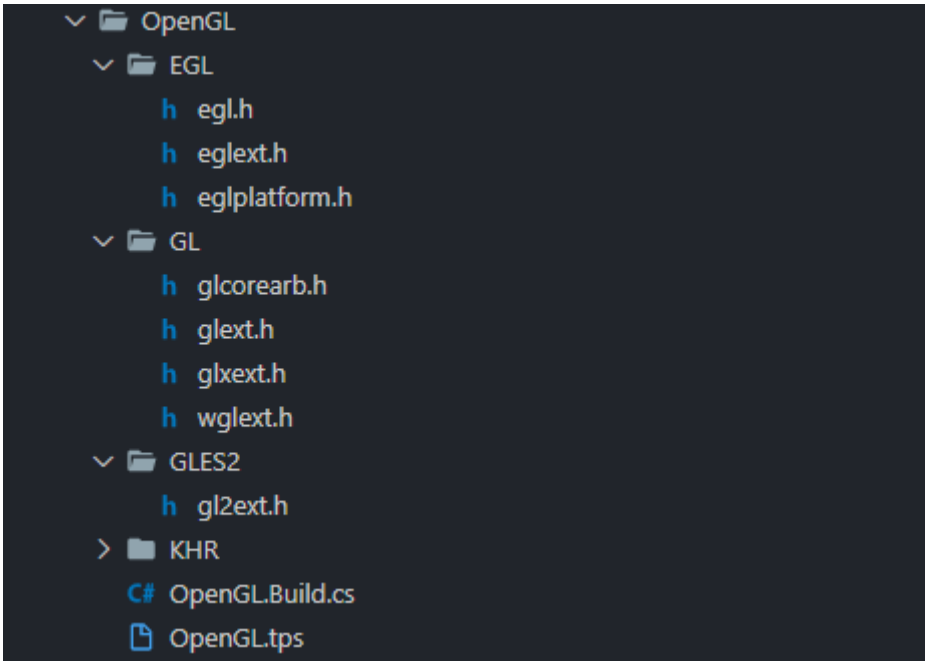


引擎的四层封装

引擎为了让不同的渲染API能在各自支持的平台上进行渲染的工作，从渲染的标准API实现开始向上层层封装了四层。以OpenGL为例：分别将OpenGL的原生实现翻译并针对引擎做适当的调整（即有的功能可以实现，有的可能不可以）；进行平台特异化，对于Windows平台，引入Win独有的<wgl.h>，用枚举当做容器，用指针来传递；将翻译好的接口封装起来当做OpenGL的引擎层面的业务接口；以及最终用抽象类来当做各种不同渲染RHI的入口。

OpenGL原生实现

第三方OpenGL模块



原生Khronos Group的OpenGL的实现是放在引擎的第三方文件夹中（虽然说是第三方库，但是引擎常常对里面的库进行魔改，具体可见SDL2），引入方法是脚本语言C#进行Markup来实现类似宏的预编译方法，

```
17  THIRD_PARTY_INCLUDES_START
18      #include <GL/glcorearb.h>
19      #include <GL/glext.h>
20  THIRD_PARTY_INCLUDES_END
```

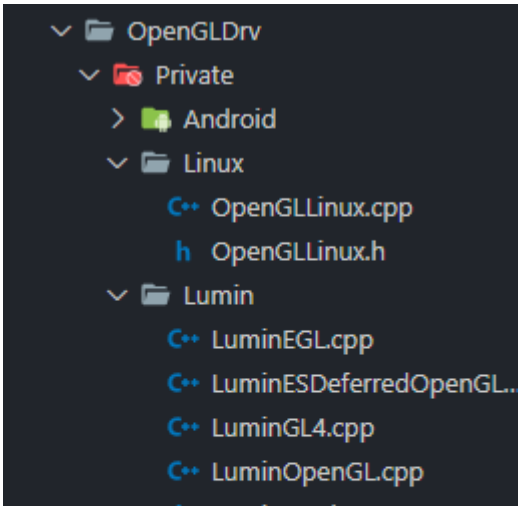
ThirdParty/OpenGL/GL/glcorearb.h & glext.h

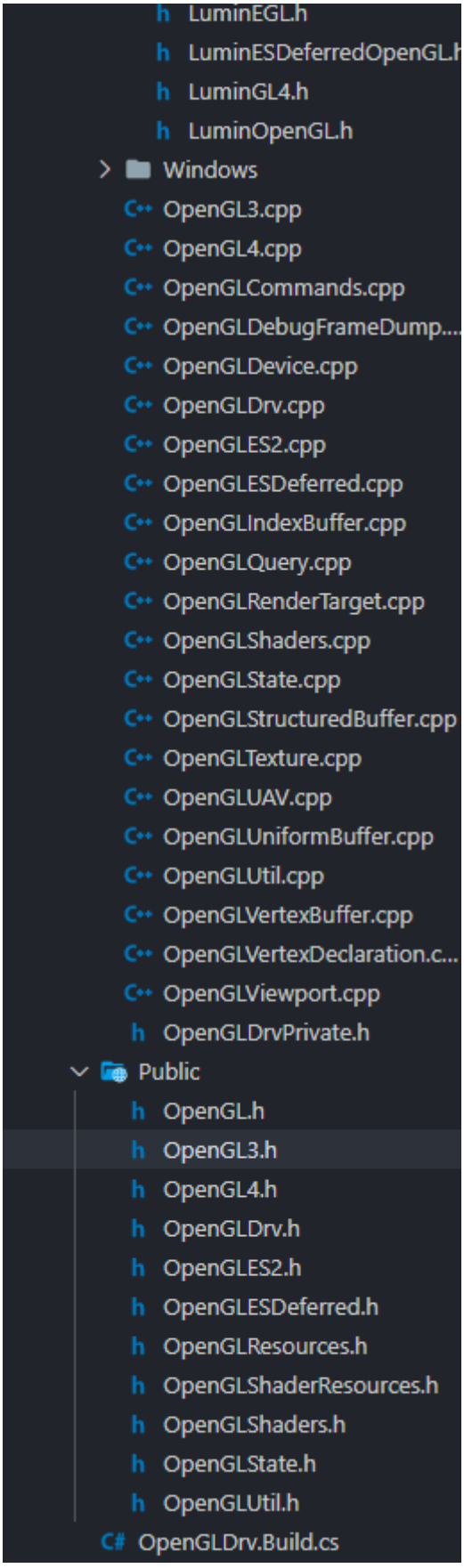
这些头文件是OpenGL的标准原生实现接口，主要由引擎引入并进行进一层的封装。不止是OpenGL，Vulkan、DX9、DX11和DX12的原生实现也分别可以第三方文件夹中找到。但是Metal的原生实现并没有找到

接下来，引擎将对OpenGL原生实现进行一系列的封装直到在引擎渲染层面，接口可以被统一起来。

OpenGLDrv模块

模块的文件结构大致如下所示：

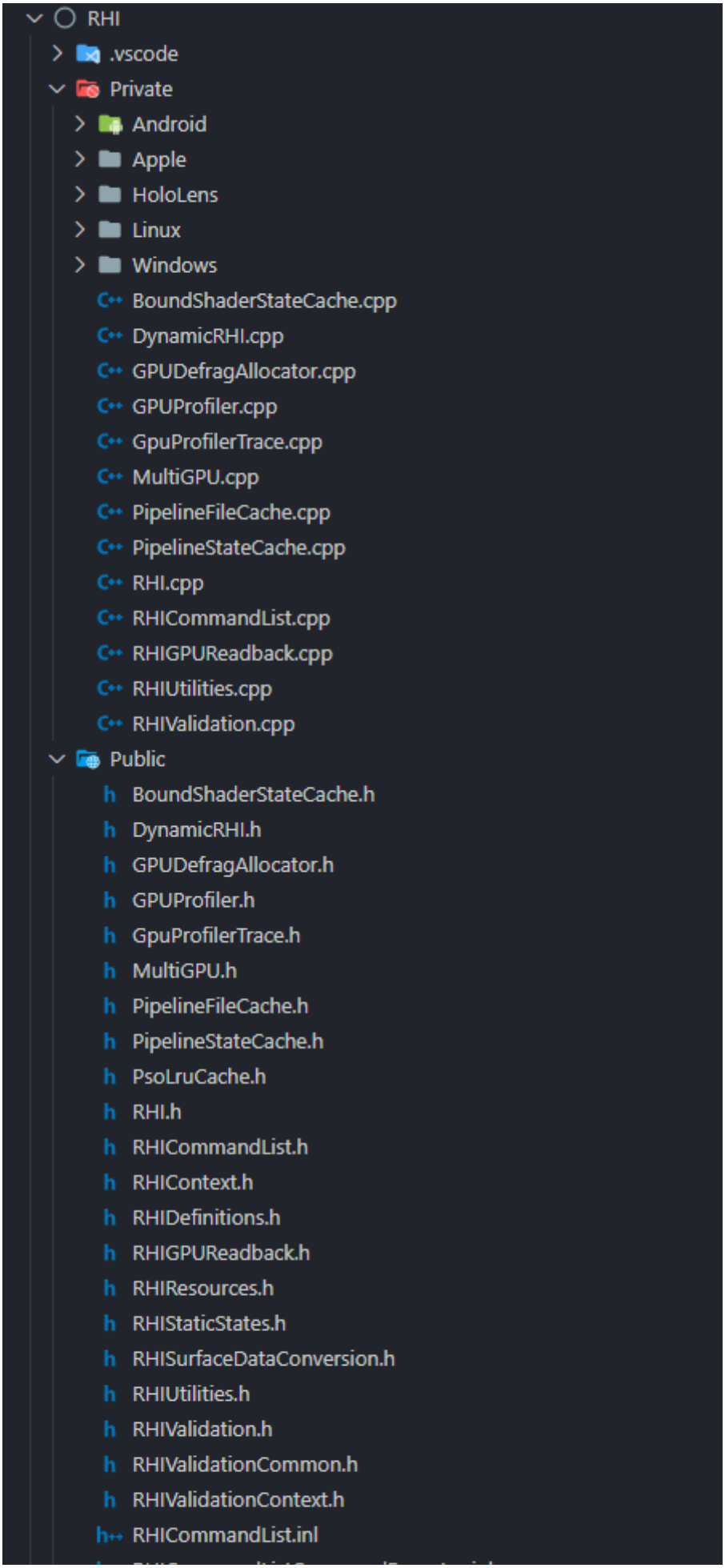


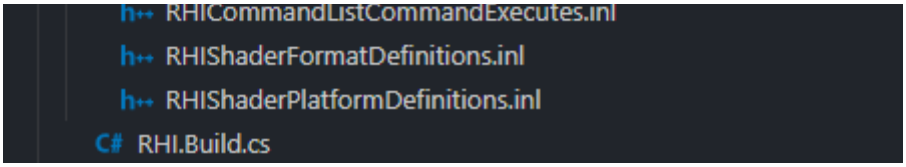


- 在前三层封装中我们将看到引擎是如何将原生OpenGL与OpenGLRHI联系起来的
- 第一层封装发生在Public文件夹下的OpenGL.h & OpenGL3.h & OpenGL4.h
- 第二层封装发生在Private/OS文件夹下的OpenGLOS_NAME.h
- 第三层封装发生在Public文件夹下的OpenGLDrv.h

RHI模块

模块的文件架构大致如下所示：





- 在最后一层封装中我们将引擎所使用的RHI是如何实现的
- 最后一层封装发生在Public文件下的DynamicRHI.h中

第一层封装

Runtime/OpenGLDrv/OpenGL.h

首先OpenGL.h定义了基类FOpenGLBase，其中包含了大部分OpenGL中的一些不常见的接口以及参数以及支持的功能，并对其进行最基本的封装（将原本GL的函数名修改一下）。

1. 变量：支持的RGB格式

```
141     static FORCEINLINE bool SupportsBGRA8888RenderTarget()              { return true; }
142     static FORCEINLINE bool SupportsSRGB()                              { return true; }
143     static FORCEINLINE bool SupportsRGBA8()                             { return true; }
144     static FORCEINLINE bool SupportsDXT()                               { return true; }
145     static FORCEINLINE bool SupportsPVRTC()                             { return false; }
146     static FORCEINLINE bool SupportsATITC()                             { return false; }
147     static FORCEINLINE bool SupportsASTC()                              { return bSupportsASTC; }
```

2. 变量：不支持Computer Shader

```
164     static FORCEINLINE bool SupportsComputeShaders()                    { return false; }
165     static FORCEINLINE bool SupportsTextureView()                       { return false; }
166     static FORCEINLINE bool SupportsSeamlessCubeMap()                   { return false; }
167     static FORCEINLINE bool SupportsDrawIndirect()                      { return false; }
```

3. 函数：OpenGL的原生函数

```
387     static FORCEINLINE void BufferSubData(GLenum Target, GLintptr Offset, GLsizeiptr
        Size, const GLvoid* Data)  { glBufferSubData(Target, Offset, Size, Data); }
388     static FORCEINLINE void DeleteBuffers(GLsizei Number, const GLuint* Buffers)
        { glDeleteBuffers(Number, Buffers); }
389     static FORCEINLINE void DeleteTextures(GLsizei Number, const GLuint* Textures)
        { glDeleteTextures(Number, Textures); }
390     static FORCEINLINE void Flush()
        { glFlush(); }
391     static FORCEINLINE GLuint CreateShader(GLenum Type)
        { return glCreateShader(Type); }
392     static FORCEINLINE GLuint CreateProgram()
        { return glCreateProgram(); }
```

接下来为了适配OpenGL3.X以及4.X，引擎分别抽象出了两个头文件，OpenGL3.h以及OpenGL4.h。

Runtime/OpenGLDrv/OpenGL3.h

OpenGL3.h中定义了FOpenGLBase的子类——FOpenGL3类，并在其中补充了GL3.X新增的函数接口，如`glBindFragDataLocation()`

```
214     static FORCEINLINE void BindFragDataLocation(GLuint Program, GLuint Color, const
        GLchar* Name)
215     {
216         glBindFragDataLocation(Program, Color, Name);
217     }
```

Runtime/OpenGLDrv/OpenGL4.h

相对应的，OpenGL4.h中定义了FOpenGL3的子类——FOpenGL4类，并在其中补充了GL4.X新增的函数接口，如`glBindVertexBuffer() GL_ARB_vertex_attrib_binding`

```
70      static FORCEINLINE void BindVertexBuffer(GLuint BindingIndex, GLuint Buffer,
        GLintptr Offset, GLsizei Stride)
71      {
72          glBindVertexBuffer(BindingIndex, Buffer, Offset, Stride);
73      }
```

```
15  struct FOpenGL4 : public FOpenGL3
16  {
17      static FORCEINLINE bool SupportsComputeShaders()                { return
        bSupportsComputeShaders; }
18      static FORCEINLINE bool SupportsDrawIndirect()                  { return
        true; }
19      static FORCEINLINE bool SupportsVertexAttribBinding()           { return
        bSupportsVertexAttribBinding; }
20      static FORCEINLINE bool SupportsTextureView()                   { return
        bSupportsTextureView; }
```

第二层封装

引擎为了更好地引入如OpenGL这种跨平台支持的渲染接口，它根据不同的平台抽象出来不同的类并进行封装。

Runtime/OpenGLDrv/Linux/OpenGLLinux.h(.cpp)

首先引入了OpenGL库的原生实现，

```
17  THIRD_PARTY_INCLUDES_START
18      #include <GL/glcorearb.h>
19      #include <GL/glext.h>
20  THIRD_PARTY_INCLUDES_END
```

接着将所有的gl接口用枚举存起来，并用指针来获取，

```
23  #define ENUM_GL_ENTRYPOINTS(EnumMacro) \
24      EnumMacro(PFNGLBINDTEXTUREPROC,glBindTexture) \
25      EnumMacro(PFNGLBLENDFUNCPROC,glBlendFunc) \
26      EnumMacro(PFNGLCOLORMASKPROC,glColorMask) \
27      EnumMacro(PFNGLCOPYTEXIMAGE1DPROC,glCopyTexImage1D) \
28      EnumMacro(PFNGLCOPYTEXIMAGE2DPROC,glCopyTexImage2D) \
29      EnumMacro(PFNGLCOPYTEXSUBIMAGE1DPROC,glCopyTexSubImage1D) \
30      EnumMacro(PFNGLCOPYTEXSUBIMAGE2DPROC,glCopyTexSubImage2D) \
31      EnumMacro(PFNGLCULLFACEPROC,glCullFace) \
32      EnumMacro(PFNGLDELETETEXTURESPROC,glDeleteTextures) \
33      EnumMacro(PFNGLDEPTHFUNCPROC,glDepthFunc) \
34      EnumMacro(PFNGLDEPTHMASKPROC,glDepthMask) \
35      EnumMacro(PFNGLDEPTHRANGEPROC,glDepthRange) \
36      EnumMacro(PFNGLDISABLEPROC,glDisable) \
37      EnumMacro(PFNGLDRAWARRAYSPROC,glDrawArrays) \
38      EnumMacro(PFNGLDRAWBUFFERPROC,glDrawBuffer) \
39      EnumMacro(PFNGLDRAWELEMENTSPROC,glDrawElements) \
40      EnumMacro(PFNGLENABLEPROC,glEnable) \
41      EnumMacro(PFNGLFINISHPROC,glFinish) \
```

```
360 /** List of all OpenGL entry points. */
361 #define ENUM_GL_ENTRYPOINTS_ALL(EnumMacro) \
362     ENUM_GL_ENTRYPOINTS(EnumMacro) \
363     ENUM_GL_ENTRYPOINTS_OPTIONAL(EnumMacro)
364
365 /** Declare all GL functions. */
366 #define DECLARE_GL_ENTRYPOINTS(Type,Func) extern Type OPENGLDRV_API Func;
367
368 // We need to make pointer names different from GL functions otherwise we may end up
    getting
369 // addresses of those symbols when looking for extensions.
370     namespace GLFuncPointers
371     {
372         ENUM_GL_ENTRYPOINTS_ALL(DECLARE_GL_ENTRYPOINTS);
373     };
374
375     #undef DECLARE_GL_ENTRYPOINTS
376
377 // this using is needed since the rest of code uses plain GL names
378 using namespace GLFuncPointers;
```

注册进SDL2内，让OpenGL与SDL2协同渲染，

```
862         if (bOpenGLSupported)
863         {
864             // Initialize all entry points required by Unreal.
865             #define GET_GL_ENTRYPOINTS(Type,Func) GLFuncPointers::Func =
                reinterpret_cast<Type>(SDL_GL_GetProcAddress(#Func));
866             ENUM_GL_ENTRYPOINTS(GET_GL_ENTRYPOINTS);
867             ENUM_GL_ENTRYPOINTS_OPTIONAL(GET_GL_ENTRYPOINTS);
868             #undef GET_GL_ENTRYPOINTS
```

> 之所以要用这种引入方式，是因为SDL2官网中写着，*If the GL library is loaded at runtime with SDL_GL_LoadLibrary(), then all GL functions must be retrieved this way. Usually this is used to retrieve function pointers to OpenGL extensions.*

同时借助SDL2来帮助OpenGL管理语境Context，

```
111 /**
112  * Create a dummy window used to construct OpenGL contexts.
113  */
114 void Linux_PlatformCreateDummyGLWindow( FPlatformOpenGLContext *OutContext )
115 {
116     static bool bInitializedWindowClass = false;
117
118     // Create a dummy window.
119     SDL_HWindow DummyWindow = SDL_CreateWindow( NULL,
120                                                 0, 0, 1, 1,
121                                                 SDL_WINDOW_OPENGL |
                                                    SDL_WINDOW_BORDERLESS |
                                                    SDL_WINDOW_HIDDEN |
                                                    SDL_WINDOW_SKIP_TASKBAR );
122
123     if (DummyWindow == nullptr)
124     {
125         FString SdlError(UTF8_TO_TCHAR(SDL_GetError()));
126
127         FText ErrorMessage = FText::Format(NSLOCTEXT("Renderer",
            "LinuxCannotCreatePlatformWindow", "Cannot create OpenGL-enabled SDL window.
            SDL error: '{0}'."), FText::FromString(SdlError));
128         FPlatformMisc::MessageBoxExt(EAppMsgType::Ok,
129             *ErrorMessage.ToString(),
130             *(NSLOCTEXT("Renderer", "LinuxCannotCreatePlatformWindowTitle", "Cannot
                create SDL window.").ToString()));
131         FPlatformMisc::RequestExit(true);
132         // unreachable
133         return;
134     }
135     else
136     {
137         SDL_SetWindowTitle(DummyWindow, "UE4 Dummy GL window");
138     }
139
140     OutContext->hWnd                   = DummyWindow;
141     OutContext->bReleaseWindowOnDestroy = true;
142 }
```

> 之所以UE4要使用SDL2，是因为相比于GLFW，SDL2除了窗口库之外还提供了各种多媒体库，同时还有管理Context的能力。

最终，OpenGLLinux.h定义了上层会真正使用的派生类FLinuxOpenGL，实际上继承自FOpenGL4，保证了功能的连续性，同时还定义了一部分GL最新以及引擎刚加入的功能。最终的最终，FLinuxOpenGl会被翻译成FOpenGL，也就是后面OpenGL接口被引用的地方。

第三层封装

当OpenGL的接口可以被获取之后，它们就可以被组合起来创建一些新的功能或者是实现一些业务，如此生成的接口就是引擎的第三层封装。

Runtime/OpenGLDrv/OpenGLDrv.h

在OpenGLDrv.h中，引擎定义了FOpenGLDynamicRHI类派生自FDynamicRHI类（更通用化的RHI），其中包含了大量的上层功能性接口的虚函数，分别对应渲染管线的不同位置，例如，

```
/* 生成顶点缓冲 */
virtual FVertexBufferRHIRef RHICreateVertexBuffer(uint32 Size, uint32 InUsage, FRHIResourceCreateInfo& CreateInfo) final override;

/* 生成纹理 */
virtual FTexture2DRHIRef RHICreateTexture2D(uint32 SizeX, uint32 SizeY, uint8 Format, uint32 NumMips, uint32 NumSamples, uint32 Flags, FRHIResourceCreateInfo& CreateInfo) final override;
virtual FTexture2DRHIRef RHICreateTexture2D(uint32 SizeX, uint32 SizeY, uint8 Format, uint32 NumMips, uint32 NumSamples, uint32 Flags, FRHIResourceCreateInfo& CreateInfo) final override;

/* 生成对应的着色器 */
virtual FPixelShaderRHIRef RHICreatePixelShader(const TArray<uint8>& Code) final override;
virtual FVertexShaderRHIRef RHICreateVertexShader(const TArray<uint8>& Code) final override;
```

分别在OpenGLVertexBuffer.cpp, OpenGLTexture.cpp和OpenGLShaders.cpp中进行实现。

Runtime/OpenGLDrv/OpenGLVertexBuffer.cpp

现在GL的接口可以通过引用上面提到过的FOpenGL的调用。OpenGLVertexBuffer.cpp主要实现了RHI级别用OpenGL创建一个VBO的接口。

```
void* GetAllocation( void* Target, uint32 Size, uint32 Offset, uint32 Alignment = 16)
{
    check(Alignment < MaxAlignment);
    check(Offset < MaxOffset);
    check(FMath::IsPowerOfTwo(Alignment));

    uintptr_t AlignmentSubOne = Alignment - 1;

    if (FOpenGL::SupportsBufferStorage() && OpenGLConsoleVariables::bUseStagingBuffer)
    {
        if (PoolVB == 0)
        {
            FOpenGL::GenBuffers(1, &PoolVB);
            glBindBuffer(GL_COPY_READ_BUFFER, PoolVB);
            FOpenGL::BufferStorage(GL_COPY_READ_BUFFER, PerFrameMax * 4, NULL, GL_MAP_WRITE_BIT | GL_MAP_PERSISTENT_BIT | GL_MAP_COHERENT_BIT);
            PoolPointer = (uint8*)FOpenGL::MapBufferRange(GL_COPY_READ_BUFFER, 0, PerFrameMax * 4, FOpenGL::EResourceLockMode::RLM_WriteOnlyPersistent);

            FreeSpace = PerFrameMax * 4;

            check(PoolPointer);
        }
        check (PoolVB);
```

Runtime/OpenGLDrv/OpenGLTextures.cpp

OpenGLTextures.cpp主要实现了RHI级别用OpenGL创建绑定纹理。

Runtime/OpenGLDrv/OpenGLShaders.cpp

OpenGLShaders.cpp主要实现了RHI级别用OpenGL创建着色器、注入代码等操作。

还有其他部分cpp文件联合实现了OpenGLDrv中的虚函数

最后一层封装

Runtime/RHI/DynamicRHI.h

在DynamicRHI.h中，引擎定义了业务接口的最终形态FDynamicRHI类，它是FOpenGLDynamicRHI的基类或父类。另一个尤为重要的是FDefaultRHIRenderQueryPool类。

Runtime/RHI/Linux/LinuxDynamicRHI.cpp

与OpenGLDrv.h类似，DynamicRHI.h也考虑到了用户在不同平台下会用不同API的现象。因此，DynamicRHI.h还被抽象出不同的平台实现`FDynamicRHI* PlatformCreateDynamicRHI()`。LinuxDynamicRHI.h主要实现了在Linux平台下，引擎所能使用的渲染RHI。

1. 首先会获取命令行的参数，得知用户强制使用的渲染库，如-opengl3，-vulkan，-d3d11，-d3d12等
2. 根据引擎内的配置文件BaseEngine.ini来获取支持的着色器格式并存入一个数组中

```
[/Script/WindowsTargetPlatform.WindowsTargetSettings]
+TargetedRHIs=PCD3D_SM5
MinimumOSVersion=MSOS_Vista

[/Script/WindowsMixedRealityRuntimeSettings.WindowsMixedRealityRuntimeSettings]
RemoteHoloLensIP=
MaxBitrate=4000

[/Script/LinuxTargetPlatform.LinuxTargetSettings]
; When neither -vulkan nor -opengl4 are passed on the commandline, the engine will default to the first targeted RHI.
+TargetedRHIs=SF_VULKAN_SM5

; OpenGL4 is deprecated, you can comment this back in to add it to your targeted RHI list
+TargetedRHIs=GLSL_430


[/Script/MacTargetPlatform.MacTargetSettings]
MaxShaderLanguageVersion=3
+TargetedRHIs=SF_METAL_SM5
UseFastIntrinsics=False
ForceFloats=False
EnableMathOptimisations=True
```

3. 遍历数组
    1. `if 不强制使用OpenGL && Vulkan成功 && 配置文件配置Vulkan`

       创建DynamicRHIModule指针指向VulkanRHI模块
    2. `if 不强制使用Vulkan && OpenGL成功 && 配置文件配置GLSL`

       创建DynamicRHIModule指针指向OpenGLDrv模块

> DynamicRHIModule单纯是实现Dynamic RHI的一个接口，里面主要实现IsSupported——定义当前平台是否支持选择的RHI以及CreateRHI——创建FDynamicRHI

4. 由DynamicRHIModule结合FeatureLevel（UE4自定义Shader Model）创建具体渲染库的FDynamicRHI
