



大象无形

虚幻引擎程序设计浅析

罗丁力 张三

电子工业出版社

Broadview
www.broadview.com.cn

大象无形

虚幻引擎程序设计浅析

罗丁力 张三



中国工信出版集团

电子工业出版社

大象无形

虛幻引擎程序设计浅析

罗丁力
张三著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

目 录

[内容简介](#)

[前言](#)

[第一部分 虚幻引擎C++编程](#)

[第1章 开发之前——五个最常见基类](#)

[1.1 简述](#)

[1.2 本立道生：虚幻引擎的UObject和Actor](#)

[1.2.1 UObject类](#)

[1.2.2 Actor类](#)

[1.3 灵魂与肉体：Pawn、Character和Controller](#)

[1.3.1 Pawn](#)

[1.3.2 Character](#)

[1.3.3 Controller](#)

[第2章 需求到实现](#)

[2.1 分析需求](#)

[2.2 转化需求为设计](#)

[第3章 创建自己的C++类](#)

[3.1 使用Unreal Editor创建C++类](#)

[3.2 手工创建C++类](#)

[3.3 虚幻引擎类命名规则](#)

[第4章 对象](#)

[4.1 类对象的产生](#)

[4.2 类对象的获取](#)

[4.3 类对象的销毁](#)

[第5章 从C++到蓝图](#)

[5.1 UPROPERTY宏](#)

[5.2 UFUNCTION宏](#)

[第6章 游戏性框架概述](#)

[6.1 行为树：概念与原理](#)

[6.1.1 为什么选择行为树](#)

[6.1.2 行为树原理](#)

[6.2 虚幻引擎网络架构](#)

[6.2.1 同步](#)

[6.2.2 广义的客户端-服务端模型](#)

[第7章 引擎系统相关类](#)

[7.1 在虚幻引擎4中使用正则表达式](#)

[7.2 FPaths类的使用](#)

[7.3 XML与JSON](#)

[7.4 文件读写与访问](#)

[7.5 GConf类的使用](#)

[7.5.1 写配置](#)

[7.5.2 读配置](#)

[7.6 UE_LOG](#)

[7.6.1 简介](#)

[7.6.2 查看Log](#)

[7.6.3 使用Log](#)

[7.6.4 自定义Category](#)

[7.7 字符串处理](#)

[7.8 编译器相关技巧](#)

[7.8.1 “废弃”函数的标记](#)

[7.8.2 编译器指令实现跨平台](#)

[7.9 Images](#)

第二部分 虚幻引擎浅析

第8章 模块机制

8.1 模块简介

8.2 创建自己的模块

8.2.1 快速完成模块创建

8.2.2 创建模块文件夹结构

8.2.3 创建模块构建文件

8.2.4 创建模块头文件与定义文件

8.2.5 创建模块预编译头文件

8.2.6 引入模块

8.3 虚幻引擎初始化模块加载顺序

8.4 道常无名： UBT和UHT简介

8.4.1 UBT

8.4.2 UHT

第9章 重要核心系统简介

9.1 内存分配

9.1.1 Windows操作系统下的内存分配方案

9.1.2 IntelTBB内存分配器

9.2 引擎初始化过程

9.3 并行与并发

9.3.1 从实验开始

9.3.2 线程

9.3.3 TaskGraph系统

9.3.4 Std::Threa

9.3.5 线程同步

9.3.6 多进程

第10章 对象模型

10.1 UObject对象

10.1.1 来源

10.1.2 重生：序列化

10.1.3 释放与消亡

10.1.4 垃圾回收

10.2 Actor对象

10.2.1 来源

10.2.2 加载

10.2.3 释放与消亡

第11章 虚幻引擎的渲染系统

11.1 渲染线程

11.1.1 渲染线程的启动

11.1.2 渲染线程的运行

11.2 渲染架构

11.2.1 延迟渲染

11.2.2 延迟渲染在PostProcess中的运用

11.3 渲染过程

11.3.1 延迟渲染到最终结果

11.3.2 渲染着色器数据提供

11.4 场景代理SceneProxy

11.4.1 逻辑的世界与渲染的世界

11.4.2 渲染代理的创建

11.4.3 渲染代理的更新

11.4.4 实战：创建新的渲染代理

11.4.5 进阶：创建静态渲染代理

11.4.6 静态网格物体渲染代理排序

11.5 Shader

- [11.5.1 测试工程](#)
- [11.5.2 定义Shader](#)
- [11.5.3 定义Shader对应的C++类](#)
- [11.5.4 我们做了什么](#)
- [11.6 材质](#)
- [11.6.1 概述](#)
- [11.6.2 材质相关C++类关系](#)
- [11.6.3 编译](#)
- [11.6.4 ShaderMap产生](#)
- [第12章 Slate界面系统](#)
- [12.1 Slate的两次排布](#)
- [12.2 Slate的更新](#)
- [12.3 Slate的渲染](#)
- [第13章 蓝图](#)
- [13.1 蓝图架构简述](#)
- [13.2 前端： 蓝图存储与编辑](#)
- [13.2.1 Schema](#)
- [13.2.2 编辑器](#)
- [13.3 后端： 蓝图的编译](#)
- [13.4 蓝图虚拟机](#)
- [13.4.1 便笺纸与白领的故事](#)
- [13.4.2 虚幻引擎的实现](#)
- [13.4.3 C++函数注册到蓝图](#)
- [13.5 蓝图系统小结](#)
- [第三部分 扩展虚幻引擎](#)
- [第14章 引擎独立应用程序](#)
- [14.1 简介](#)

[14.2 如何开始](#)

[14.3 BlankProgram](#)

[14.4 走得更远](#)

[14.4.1 预先准备](#)

[14.4.2 增加模块引用](#)

[14.4.3 添加头文件引用](#)

[14.4.4 修改Main函数为WinMain](#)

[14.4.5 添加LOCTEXT_NAMESPACE定义](#)

[14.4.6 添加SlateStandaloneApplication](#)

[14.4.7 链接CoreUObject](#)

[14.4.8 添加一个Window](#)

[14.4.9 最终代码](#)

[14.5 剥离引擎独立应用程序](#)

[第15章 插件开发](#)

[15.1 简介](#)

[15.2 开始之前](#)

[15.3 创建插件](#)

[15.3.1 引擎插件与项目插件](#)

[15.3.2 插件结构](#)

[15.3.3 模块入口](#)

[15.4 基于Slate的界面](#)

[15.4.1 Slate简介](#)

[15.4.2 Slate基础概念](#)

[15.4.3 最基础的界面](#)

[15.4.4 SNew与SAssignNew](#)

[15.4.5 Slate控件的三种类型](#)

[15.4.6 创建自定义控件](#)

[15.4.7 布局控件](#)

[15.4.8 控件参数与属性](#)

[15.4.9 Delegate](#)

[15.4.10 自定义皮肤](#)

[15.4.11 图标字体](#)

[15.4.12 组件继承](#)

[15.4.13 动态控制Slot](#)

[15.4.14 自定义容器布局](#)

[15.5 UMG扩展](#)

[15.6 蓝图扩展](#)

[15.6.1 蓝图函数库扩展](#)

[15.6.2 异步节点](#)

[15.7 第三方库引用](#)

[15.7.1 lib静态链接库的使用](#)

[15.7.2 dll动态链接库的使用](#)

[第16章 自定义资源和编辑器](#)

[16.1 简易版自定义资源类型](#)

[16.2 自定义资源类型](#)

[16.2.1 切分两个模块](#)

[16.2.2 创建资源类](#)

[16.2.3 在Editor模块中创建工厂类](#)

[16.2.4 引入Editor模块](#)

[16.3 自定义资源编辑器](#)

[16.3.1 资源操作类](#)

[16.3.2 资源编辑器类](#)

[16.3.3 增加3D预览窗口](#)

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

大象无形：虚幻引擎程序设计浅析 / 罗丁力，张三著. —北京：电子工业出版社，2017.5

ISBN 978-7-121-31349-3

I . ①大... II . ①罗...②张... III. ①游戏程序—程序设计 IV.
①TP317.6

中国版本图书馆CIP数据核字（2017）第076264号

策划编辑：符隆美

责任编辑：徐津平

印 刷：三河市良远印务有限公司

装 订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：787×980 1/16 印张：19.75 字数：380千字

版 次：2017年5月第1版

印 次：2017年5月第1次印刷

定 价：65.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlbs@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819 faq@phei.com.cn。

内容简介

本书以两位作者本人在使用虚幻引擎过程中的实际经历为参考，内容包括三大部分：虚幻引擎C++编程、虚幻引擎浅析和扩展虚幻引擎。本书提供了不同于官方文档内容的虚幻引擎相关细节和有效实践。有助于读者一窥虚幻引擎本身设计的精妙之处，并能学习到定制虚幻引擎所需的基础知识，实现对其按需定制。

本书适合初步了解虚幻引擎编程，并希望学习虚幻引擎架构或者希望定制和扩展虚幻引擎的人士。

前言

建德若偷，质真若渝。大方无隅，大器晚成。

大音希声，大象无形。夫唯道善贷且成。

——老子，《道德经》

虚幻引擎作为业界一流的次时代引擎，开发了无数成功的作品。在短暂的计算机图形学发展历史上，虚幻引擎历经四代，成为游戏引擎界举足轻重的成员之一。

但是虚幻引擎庞大而复杂的设计，阻碍了许多人学习的步伐。尽管有蓝图系统作为图形化编程，降低了虚幻引擎的上手难度，但是当开发者们走入虚幻引擎的C++范畴，依然会感觉到无从下手。

因此，我决定和我的同事一起来撰写本书。希望能够借助我们微薄之力，帮你理解庞大的虚幻引擎是如何工作的。笔者对本书内容的期望是，这是一本笔者在学习虚幻引擎时希望能够获得的书。

同时也请明白，虚幻引擎的代码量为五百万行。本书篇幅不足以分析整个虚幻引擎的所有模块，也无法精确地向读者展示每段代码的意义。相反地，本书立足于：展示引擎基本结构，即尽可能告诉读者“它是这么跑起来的”，对于希望精确研究每一段代码过程的读者，本书会告知你如何寻找到对应的代码。

本书主标题为“大象无形”，《道德经》中有言：大器晚成，大音希

声，大象无形。本书取“伟大的设计对于使用者来说似乎感觉不到存在”和“优秀的系统设计让开发者不需要过多了解原理即能使用”这样的含义。对于虚幻引擎而言，本书中介绍的很多知识，对于普通开发者来说似乎是“没有感觉到存在”的东西，例如引擎的渲染系统，普通开发者几乎只需要简单地完成导入和摆放就能使用，并不需要实际了解渲染系统的工作原理。能够达到这样的效果，恰恰说明了虚幻引擎设计的优秀：能够让开发者不需要了解系统的机制，就能够快速使用其来完成自己的需求，此即“无形”。然而这样优秀的设计是如何完成的？如何扩展这样的设计来让开发者完成自己独特的需求？这是本书希望探讨的内容。

本书由两位作者共同编撰而成，其中罗丁力先生完成了第一部分（除《引擎系统相关类》章节）和第二部分，以及第三部分中《引擎独立应用程序》《自定义资源和编辑器》章节，张三先生完成了第二部分中《引擎系统相关类》章节与第三部分中《插件开发》章节。

笔者才疏学浅，撰写本书仅仅为个人一家之言。欢迎每一位读者对本书提出建议和指正，也欢迎更多的人去撰写虚幻引擎相关的书籍，共同为虚幻引擎的推广、运用做出努力。你可以发送邮件到 three@sanwu.org。

感谢Unreal Engine，陪伴我度过了最美好的青春。

阅读之前

你好，欢迎你阅读本书。在这里我希望能向你讲述一些关于阅读本

书的约定。首先，这不是一本“虚幻引擎入门宝典”或是“虚幻引擎从入门到精通”。本书的作者们希望把视角集中到那些市面上的教程没有涉及的领域，所以我们不会教你：

1. 如何下载引擎
2. 如何安装引擎和Visual Studio
3. 如何更新引擎
4. 如何申请虚幻引擎账户

我们假定你已经掌握这些知识。并且我们也不会教你：

1. C++语法
2. C语言语法

我们认为你在使用虚幻引擎的C++语言进行编程之前，已经掌握了C++的基础语法，包括函数、变量、类、指针与模板。当然，我们会向你解释虚幻引擎中的独有的C++成分，包括C++11标准的一些内容。

如果你已经做好了准备，欢迎开始你的阅读之旅。本书分为以下三个部分：

虚幻引擎C++编程 这个部分简单介绍虚幻引擎的C++编程方式，你可以通过这个部分回顾、整理你从官方文档学习到的有关使用虚幻引擎 进行编程的知识，并给出了一部分官方文档尚未介绍但可以被使用的库、API与技巧。

虚幻引擎浅析 这个部分将会引导读者去研究虚幻引擎源码，并给出笔者认为在深入使用虚幻引擎进行游戏开发的过程中，可能会需要具备的引擎架构、模块如何工作的知识。换句话说，这个部分

介绍虚幻引擎是如何工作的。

扩展虚幻引擎 这个部分则是通过介绍虚幻引擎的插件编写，将第二部分的知识运用起来，让读者不至于觉得这是“屠龙之技”，虽有思辨的乐趣，却没有用武之地。进而赋予读者定制虚幻引擎 以符合自己游戏实际情况的能力。笔者认为这是专业游戏开发者所需要具备的技能。

在每一小节开头，笔者会提供一个常常被问及的问题，然后根据这个问题来阐述接下来的内容，就像这样：

问题

我该如何学习虚幻引擎？

读者可以在阅读完每一个小节后，回顾小节开头的问题，以检验自己是否已经理解了本节的内容。

笔者在这里衷心地祝愿你找到你希望学习的知识，祝你一切顺利！

鸣谢

本书在撰写过程中受到了大量同行、朋友及亲人的帮助，有许多同行无私地贡献了自己的想法、意见及自己宝贵的经验，在此对他们表示真挚的感谢：

非常感谢Net Fly和秦春林先生对本书的支持，他们不仅帮助笔者联系了本书的出版社，也非常认真地审阅本书的稿件，并给出了中肯有效的意见，没有他们的帮助，本书不可能出版。

非常感谢傅建钊先生对本书的帮助，提出了大量有效的意见，并组织了相当多的业内人士共同讨论本书的主题，他的知乎专栏《Inside UE4》对虚幻引擎的剖析同样非常精彩，建议读者可以参考。

同时，也有不少同行针对书中许多主题给出了自己独到的见解，并被整理到书中。LSFW先生给笔者多次反复讲解渲染框架设计，贡献出了自己对渲染系统的研究成果；黄河水先生、Dest1ny先生撰写了大量博客来分析虚幻引擎的底层架构，给笔者启发颇多；王德立先生帮助本书绘制了插图。还有许许多多同行，在此恕无法一一举名。

感谢三巫社区和Epic Games对本书的出版过程的支持与帮助。

最后，作者之一罗丁力希望感谢Black Rock Shooter，感谢她在撰写本书的过程中，对其鼓励与陪伴。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- 提交勘误： 您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

- 与作者交流： 在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31349>



第一部分

虚幻引擎C++编程

本文假定你已经了解C++的语法，包括变量、函数、类与指针。

我们不会在这里教你如何书写函数声明，如何调用一个函数等。本文并不需要你了解虚幻引擎的C++编程，因为我们会讲述到。但是，不同于官方教程的简单明了，我们将会向你介绍更多关于“为什么这样设计”的内容。如果你希望在几分钟之内上手虚幻引擎的C++编程，可能你更应该直接阅读虚幻引擎的C++Quick Start Guide。

如果你发现，在阅读了官方的文档之后，你依然感到无所适从，不知道如何下手的话，我想接下来的章节能够让你对虚幻引擎的开发，有更加全面的认识。

本部分希望用一组相互之间比较独立的短文，来向你陈述虚幻引擎C++编程的一些技巧，并帮助你准备后文需要的一些知识。

第1章 开发之前——五个最常见基类

1.1 简述

问题

如何最快上手虚幻引擎的C++编程？

许多人都询问过这样的问题。学习虚幻引擎的编程技术有许多的道路，但是笔者认为，抓住最核心的五个类，提纲挈领地学习，能够更好地理解。这五个类就是：

UObject Actor Pawn Controller Character

1.2 本立道生：虚幻引擎的**UObject**和**Actor**

1.2.1 **UObject**类

问题

什么时候该继承自**UObject**类？什么时候应该声明一个纯C++类？

任何一个C++程序员都知道，不同于Java，C++的类可以没有父类。那么，什么样的类对象应该继承自**UObject**类？

从语义上看，**UObject**类表示这是一个“对象”，但是这并不能说服一个C++程序员。毕竟，任何一个纯C++类都能实例化对象。事实上，我们应该这样思考。一个类继承自**UObject**类，应该是它需要**UObject**类提供的功能。什么样的功能让你选择继承自**UObject**类？

从虚幻引擎官方文档我们可以得知，**UObject**类提供了以下功能：

1. Garbage collection垃圾收集
2. Reference updating引用自动更新
3. Reflectio反射
4. Serialization序列化
5. Automatic updating of default property changes自动检测默认变量的更改
6. Automatic property initialization自动变量初始化
7. Automatic editor integration和虚幻引擎编辑器的自动交互
8. Type information available at runtime运行时类型识别
9. Network replication网络复制

我将会详细讲述这些功能中重点功能的含义。

垃圾收集

C++的内存管理是由程序员完成的。因此对象管理一直是一个很棘手的问题。往往一个对象可能会引用多个其他对象，同一个对象也可能可能会被多个对象引用。那么，当你不需要用到当前对象A的时候，该不该释放该对象所在的内存区域？

任何人都会犹豫：

释放 一旦有别的对象引用当前对象，释放后就会产生野指针。
当另一个对象来访问时，会看到空空如也甚至是其他对象的内存区域。

不释放 有可能我已经是最后一个引用这个对象的人了，一旦我丢弃这个指针，这个对象就不会再有人知道，这片内存区域永远无法被回收。

对此虚幻引擎提供了如下两个解决方案：

1. 继承自UObject类，同时指向UObject类实例对象的指针成员变量，使用UPROPERTY宏进行标记。虚幻引擎的UObject架构会自动地被**UPROPERTY**标记的变量 考虑到垃圾回收系统中，自动地进行对象的生命周期管理。
2. 采用智能指针。请注意，只有非**UObject**类型，才能够使用智能指针进行自动内存释放。关于智能指针的讨论，请看本书后面的章节。

反射

反射并不是图形学意义上的“反射”。而是指一种语言的机制。这样的机制在C#、Java中已经存在，但是C++并没有。我以一种通俗易懂的解释来描述反射，如果你需要反射的详细解释，请阅读搜索引擎中对反射的解释。

如果你是一名C++程序员，那么请你思考一个问题：

我该如何在运行时获取一个类？有哪些成员变量、成员函数？我该如何获取这些成员变量的名字？

很难对吗？C++本身没有提供这样一套机制。尽管你可以用各种方式来手动实现。

虚幻引擎实现了这样一套机制。如果你好奇反射是怎样实现的，可以阅读本书第二部分的内容。

序列化

当你希望把一个类的对象保存到磁盘，同时在下次运行时完好无损地加载，那么你同样需要继承自UObject类。

但是需要澄清的是，你可以通过给自己的纯C++类手动实现序列化所需要的函数，来让这个类支持序列化功能。这并不是UObject类独有的。

与虚幻引擎编辑器的交互

还记得虚幻引擎编辑器的Editor面板吗？你希望你的类的变量能够被Editor简单地编辑吗？那么你需要继承自这个类。

运行时类型识别

请注意，虚幻引擎打开了/G-编译器参数。意味着你无法使用C++标准的RTTI机制：dynamic_cast。如果你希望使用，请继承自UObject类，然后使用Cast<>函数来完成。

这是因为虚幻引擎实现了一套自己的、更高效的运行时类型识别的方案。

网络复制

当你在进行网络游戏开发（c/s架构）时，你一定希望能够自动地处理变量的同步。

而继承自UObject类，其被宏标记的变量能够自动地完成网络复制的功能。从服务器端复制对应的变量到客户端。

综上所述，当你需要这些功能的时候，你的这个类应该继承自UObject类。

请注意：UObject类会在引擎加载阶段，创建一个Default Object默认对象。这意味着：

1. 构造函数并不是在游戏运行的时候调用，同时即便你只有一个 UObject对象存在于场景中，构造函数依然会被调用两次。
2. 构造函数被调用的时候，UWorld不一定存在。GetWorld()返回值有可能为空！

1.2.2 Actor类

问题

什么时候该继承自**Actor**类？

Actor类是游戏中一切实体Actor的基类。这样的解释严格来说是错误的，笔者不能使用Actor来解释Actor。那么我们应该更加明晰一些。同样地，我们采用之前的分析方式。Actor类提供了什么功能，让我们选择继承自它？

有朋友会回答，Actor类在场景中拥有一个位置坐标和旋转量。

请注意，这是也是错误的。

Actor类拥有这样的能力：它能够被挂载组件。

组件并不是Actor。如果你观察，会发现所有组件的类的开头是U而不是A。虚幻引擎中，Component的含义与Unity引擎中的Component具有极大的区别。如果你从Unity引擎转移而来，我有必要向你澄清这一点：

虚幻引擎中，一个场景实体对应一个类。在Unity中，一个对象可以挂载多个脚本组件。每个脚本组件是一个单独的类。因而从某种意义上说，有许多Unity程序员认为这相当于一个场景实体可以看作是多个类。

虚幻引擎中，Component的含义被大大削弱。它只是组件，不能越俎代庖。Untiy引擎中组件的大多数功能将会交给对应的、继承自Actor类的子类来实现。

而坐标与旋转量，只是一个Scene Component组件。如果这个Actor不需要一个固定位置（例如你的某个Manager），你甚至可以不给Actor挂载Scene Component组件。

- 你希望让Actor被渲染？给一个静态网格组件。
- 你希望Actor有骨骼动画？给一个骨架网格物体组件。
- 你希望你的Actor能够移动？通常来说你可以直接在你的Actor类中书写代码来实现。当然，你也可以附加一个Movement组件以专门处理移动。

所以，需要挂载组件的时候，你才应该继承自Actor类。也就是说，我刚刚描述的，你的Manager，也许只需要一个纯C++类就够了（当然，你需要序列化之类的功能，那就是另一回事了）。

1.3 灵魂与肉体： Pawn、 Character 和 Controller

1.3.1 Pawn

在国际象棋里面， Pawn代表的是如图1-1所示：



图1-1 Pawn, 国际象棋棋子

没错，就是兵或卒。那么这个类为何这样命名？因为这个命名我认为是极为形象的。其十分生动地体现了Pawn类的特性。

如果你研究了Pawn类的源码，你会发现Pawn类提供了被“操作”的特性。它能够被一个Controller操纵。这个Controller可以是玩家，当然也可以是AI（人工智能）。这就像是一个棋手，操作着这个棋子。

这就是Pawn类，一个被操纵的兵或卒，一个一旦脱离棋手就无法自主行动的、悲哀的肉体。

1.3.2 Character

Character类代表一个角色，它继承自Pawn类。那么，什么时候该继承自Character类，什么时候该继承自Pawn类呢？这个问题的答案，我们必须从Character类的定义中寻找——它提供了什么样的功能？

Character类提供了一个特殊的组件，Character Movement。这个组件提供了一个基础的、基于胶囊体的角色移动功能。包括移动和跳跃，以及如果你需要，还能扩展出更多，例如蹲伏和爬行。

如果你的Pawn类十分简单，或者不需要这样的移动逻辑（比如外星人飞船），那么你可以不继承自这个类。请不要有负罪感：

1. 不是虚幻引擎中的每一个类，你都得继承一遍。
2. 在Unreal Engine 3中，没有Character类，只有Pawn类。

当然，现在很多游戏中的角色（无论是人类，还是某些两足行走的怪物），都能够适用于Character类的逻辑。

1.3.3 Controller

本节的标题是：灵魂与肉体。作为一名无神论者，我只是采用这样的比喻。但是似乎Epic的开发人员选择了棋手与棋子的比喻。相比之下，我的比喻还是太过肤浅了些。

Controller是漂浮在Pawn/Character之上的灵魂。它操纵着Pawn和Character的行为。Controller可以是AI， AIController类， 你可以在这个类中使用虚幻引擎优秀的行为树/EQS环境查询系统。同样也可以是玩家， Player Controller类。你可以在这个类中绑定输入， 然后转化为对Pawn的指令。

我希望阐述的是，为何虚幻引擎采用这样的设计。Epic给出的理由非常简单：“不同的怪物也许会共享同样的Controller，从而获得类似的行为”。其实， Controller抽象掉了“怪物行为”，也就是扮演了有神论者眼中“灵魂”的角色。

既然是灵魂，那么肉体就不唯一，因此灵魂可以通过 Possess/Unpossess来控制一个肉体，或者从一个肉体上离开。

肉体拥有的只是简单的前进、转向、跳跃、开火等函数。而 Controller则是能调用这些函数。从某种意义上来说，MVC中的 Controller与虚幻引擎这套系统有着某种类似。虚幻引擎的Controller对应着MVC的Controller， Pawn就是Model，而Pawn挂载的动态网格组件（骨架网格或者静态网格），对应着MVC的View。虽然这种比喻不是非常恰当，但是能方便理解。

第2章

需求到实现

问题

我有一个庞大的游戏创意，但这是我第一次制作游戏，我该设计哪些类？

2.1 分析需求

在前面的内容中，笔者介绍了虚幻引擎中重要的几个类的含义。并告诉了你如何继承。但是，如何从具体的需求中，分析出类的设计和架构，并最终导出一个完整的解决方案呢？这其实是更多人希望知道的。那么，我们该从什么地方开始呢？答案是，从需求开始。

考虑到阅读本书的有些朋友没有经过系统的软件工程训练（这不是你的错，有可能你是一个热爱游戏，热爱虚幻引擎的开发者），因此，我有必要简单介绍需求这个概念。

什么是需求？客户想要什么，就是需求。

那么，从游戏开发的角度而言，需求就是你身为游戏设计师，分析出的需要实现的东西。可能最开始只是模糊不清的字句，比如“我希望我的主角能够在楼宇之间不断跳跃”，或是“我希望我的主角能够手持一

个手电筒不断探索”。不管怎样，你的游戏应该有一个完整的、成文本的设计书。这个设计书描述了你这个游戏的大体设计。你可以从网络上找到各种各样的游戏设计书模板，但是你最终的目的是要向一个陌生人，用你的设计书阐述你的游戏设计，让他明白你想做一个什么样的游戏。接下来，你需要将你的设计转化为需求点。你应该按照分类来排列这些需求。如果你希望你的开发过程更加清晰可控，你应该采用专业的软件工程建模工具来绘制你的需求分析图。在UML（统一建模语言）中，你可以用“用例图”来表达你的每一个需求。

或者，通过有意义的短句：

“玩家可以通过按下空格键来跳跃”。

“玩家通过鼠标滚轮来切换武器”。

从而让你的设计变成一个一个的开发单位。

我知道，如果你是一个游戏行业的从业人士，你会认为我的这段描述是多么的粗略，甚至显得外行。但是，你应该意识到，我是在向一些非从业人士，那些依靠热血的独立开发者们，讲解软件工程的一些知识。

如果你是一名热血的开发者，正在使用QQ群，仅仅凭借聊天来继续你的项目，我希望你能够认真地看待我所提出的建议。另外，你还需要版本控制系统，以及起码的一个项目管理系统。如果你希望知道按照怎样的步骤来开发你的游戏，我建议你可以先从敏捷开发模型中选择一个，在你的项目中运用。软件工程是一门建立在无数开发者血泪之上的科学，请谨慎地对待这门科学给出的建议。

2.2 转化需求为设计

如果你接受过软件工程训练，那么你会不假思索地描述，在需求分析之后，应该是概要设计、详细设计和编码，之后会有测试等等。当然，你也会回忆起各种敏捷开发模型，例如XP、测试驱动开发、Scrum，等等。但我不会在这里讲述软件工程，而是把讨论聚集在如何实现一个具体需求设计上。这就好像，我不是在讨论绘画时的握笔、排线，而是在讨论在哪个位置下笔，在哪个位置排线。假设我们拿到的是这样的需求：

“玩家手中会持有一把武器，按下鼠标左键时，武器会射出子弹”。

从这句话中，我们能够找到这样的几个重要名词：玩家、武器和子弹。

我们意识到，这几个名词都可以作为类。也许有些类虚幻引擎已经提供给我们了，如玩家APlayerController类。那么，我们意识到，我们需要给武器和子弹各创建一个类。现在问题是，武器类该继承自什么？让我们回顾前面的章节。

首先，武器类有坐标吗？有的。这该是一个Actor的子类。

武器类是一种兵吗？不是，武器类不该是Pawn的子类。

恭喜你，你已经确定了武器类在整个游戏的类树中的位置。同样，你也能够确定子弹类在类树中的位置。它应该继承自Actor类，同时带有一个Projectile Movement组件。进一步你能够分析出，类与类之间的持有、通信关系：

1. 玩家类对象持有 武器类对象。
2. 武器类对象产生 子弹对象。
3. 玩家的输入会调用 武器类对象的函数，以发射子弹。

你已经能够想象出这几个类与函数的设计了。在下个章节中，笔者将会阐述如何把这些设计转化为实实在在的C++代码。

第3章 创建自己的C++类

问题

我想好我有哪些类了，现在我该怎么创建它们的代码呢？

3.1 使用Unreal Editor创建C++类

使用Unreal Editor创建C++类

你可以按照以下的步骤，使用Unreal Editor的C++类向导来创建你的C++类。在内容浏览器的C++类文件夹中单击鼠标右键，在弹出菜单中选择新建C++类，如图3-1所示。



图3-1 新建C++类第一步

在弹出的菜单中选择你的父类，如图3-2所示。

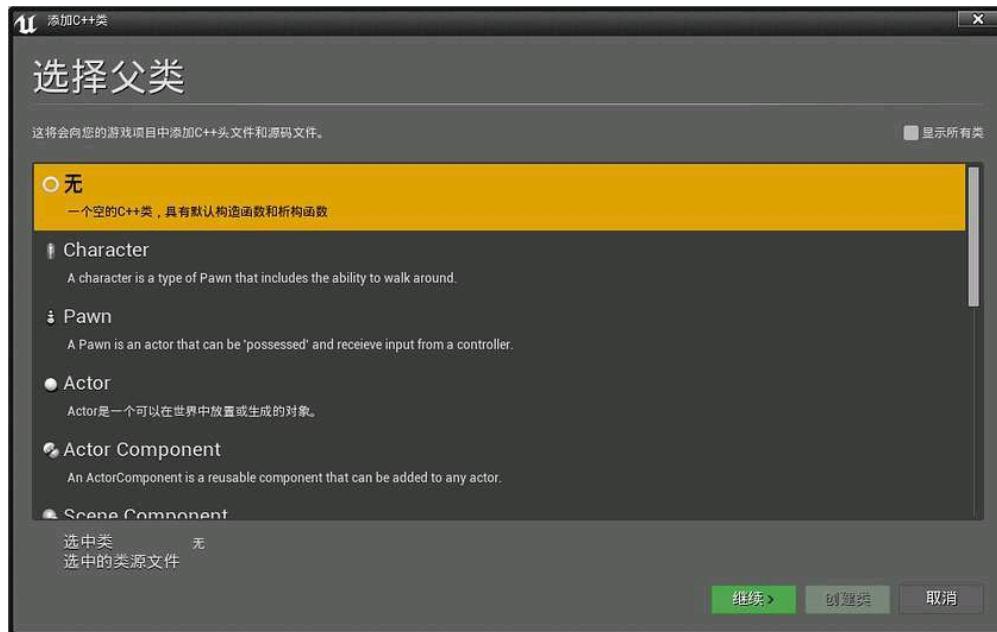


图3-2 新建C++类第二步

如果你找不到你的父类，请勾选“显示所有类”，如图3-3所示。

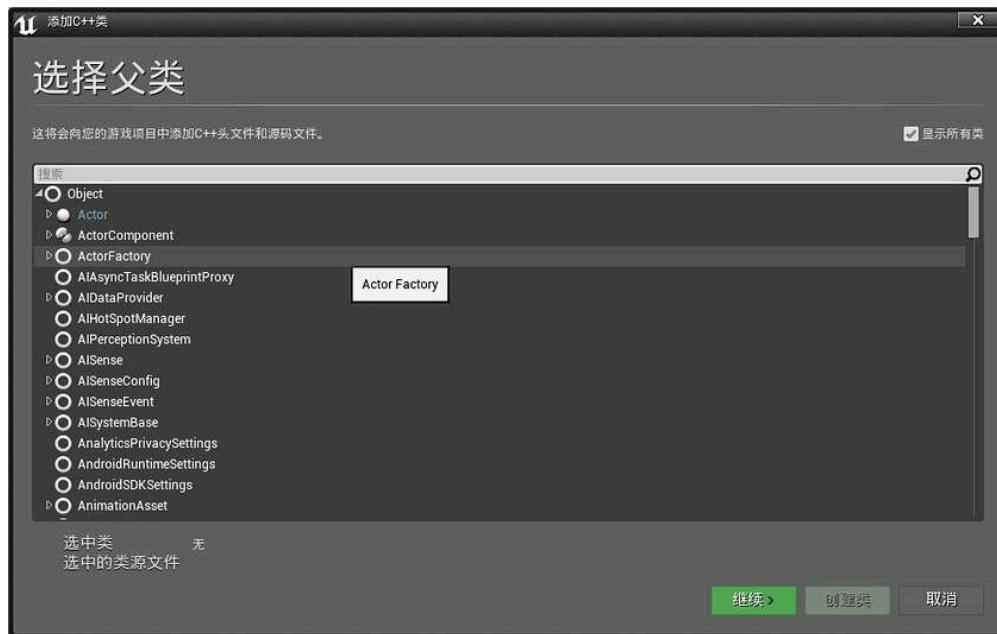


图3-3 新建C++类第三步

选中合适的父类后点击继续，填写你的类型名与路径，如图3-4所示。

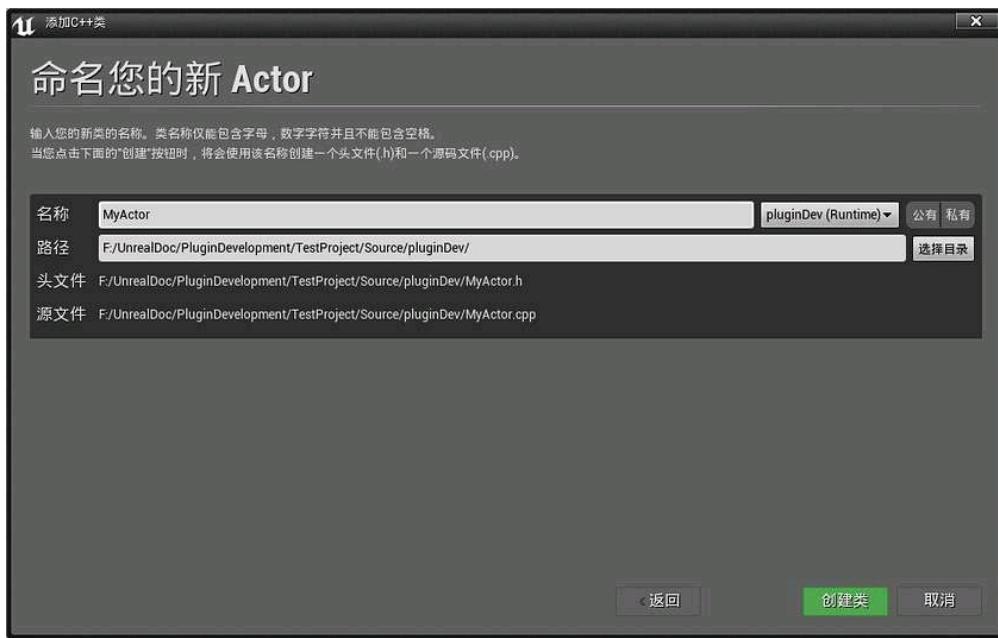


图3-4 新建C++类第四步

点击创建类后，虚幻引擎会自动打开Visual Studio。并且产生出两个模板文件（.h与.cpp），然后会自动编译，并且加载到引擎中。接下来你要做的与你在C++教程中学习到的一致。调用各种函数，完成你想要的功能吧。

3.2 手工创建C++类

如果你出于某种原因，希望自己手动创建C++类。你需要完成以下的步骤：

在工程目录的Source文件夹下，找到和你游戏名称一致的文件夹。根据不同人创建的工程结构不同，你可能会发现下面两种文件结构：

1. public文件夹，private文件夹，.build.cs文件。
2. 一堆.cpp和.h文件，.build.cs文件。

第一种文件结构是标准的虚幻引擎模块文件结构。

1. 创建你的.h和.cpp文件，如果你是第一种文件结构，.h文件放在public文件夹内，.cpp文件放置在private文件夹内。
2. 在.h中声明你的类：如果你的类继承自UObject，你的类名上方需要加入UCLASS()宏。同时，你需要在类体的第一行添加GENERATED_UCLASS_BODY()宏，或者GENERATED_BODY()宏。前者需要手动实现一个带有const FObjectInitializer&参数的构造函数。后者需要手动实现一个无参数构造函数。注意笔者说的是“实现”而非声明。
3. 在你的.cpp文件中，包含当前模块的PCH文件。一般是模块名+private PCH.h。如果是游戏模块，有可能包含的是游戏工程名.h。
4. 编译。

3.3 虚幻引擎类命名规则

终于我们要讨论到了虚幻引擎类的命名规则了。如果你仔细阅读，你会发现，笔者在前文中几乎没有用“Object”而是用的UObject作为类的名字。那么之前的U代表什么？这是按照虚幻引擎的命名规则，添加的命名前缀。常用的前缀如下：

- F 纯C++类
- U 继承自UObject，但不继承自Actor
- A 继承自Actor
- S Slate控件相关类
- H HitResult相关类

虚幻引擎头文件工具Unreal Header Tool会在编译前检查你的类命名。如果类的命名出现错误，那么它会提出警告并终止编译。在后文的描述中，笔者会按照以下规则：

1. 如果笔者是在阐述理论和逻辑，例如“你需要一个Weapon武器类”，这个时候笔者不会加上前缀。这是为了行文以及阅读的流畅。
2. 如果笔者是在描述具体的代码，例如“请阅读UStaticMeshComponent类的代码”，或者“调用AActor实例的GetActorLocation函数”，此时为了与实际代码匹配，笔者会加上类的前缀。

第4章 对象

问题

我也声明好了需要的类，那么：

我该如何实例化对象？

我该如何在世界中产生我声明的**Actor**类？

我该如何调用这些对象身上的函数？

4.1 类对象的产生

在标准C++中，一个类产生一个对象，被称为“实例化”。实例化对象的方法是通过new关键字。

而在虚幻引擎中，这一个问题变得略微复杂。对于某些类型，我们不得不通过调用某些函数来产生出对象。具体而言：

1. 如果你的类是一个纯C++类型（F开头），你可以通过new来产生对象。
2. 如果你的类继承自UObject但不继承自Actor，你需要通过NewObject函数来产生出对象。

3. 如果你的类继承自AActor，你需要通过SpawnActor函数来产生出对象。

New Object函数定义如下：

```
template<class T>
T* NewObject(
    UObject* Outer = (UObject*)GetTransientPackage(),
    UClass* Class = T::StaticClass(),
    FName Name = NAME_None,
    EObjectFlags Flags = RF_NoFlags,
    UObject* Template = nullptr,
    bool bCopyTransientsFromClassDefaults = false,
    FObjectInstancingGraph* InInstanceGraph = nullptr
)
```

事实上你可以简单地这样调用它：

```
NewObject<T>()
```

这会返回一个指向你的类的指针，此时这个对象被分配在临时包中。下一次加载会被清除。如果你的类继承自Actor，你需要通过UWorld对象（可以通过GetWorld()获得）的SpawnActor函数来产生出对象。函数定义如下（有多个，这里只列出一个）：

```
template< class T >
```

```
T* SpawnActor(  
    FVector const& Location,  
    FRotator const& Rotation,  
    const FActorSpawnParameters& SpawnParameters = FActorSpawnParamet  
    ()  
)
```

你可以这样简单地调用它：

```
GetWorld( )->SpawnActor<AYourActorClass>( )
```

极为特殊的，如果你需要产生出一个Slate类——如果你有这样的需求，要么你已经在进行很深的开发，要么就是你的教程的版本过老，依然在使用Slate来开发游戏的界面控件，你需要使用SNew函数。我无法给出SNew函数的原型。关于Slate的详细讨论，请阅读后文中Slate的章节。

4.2 类对象的获取

获取一个类对象的唯一方法，就是通过某种方式传递到这个对象的指针或引用。

但是有一个特殊的情况，也是大家经常询问到的：如何获取一个场景中，某种Actor的所有实例？答案是，借助Actor迭代器：
TActorIterator。示例代码如下：

```
for(TActorIterator <AActor> Iterator(GetWorld());Iterator;++Iterator)
{
    ...//do something
}
```

其中TActorIterator的泛型参数不一定是Actor，可以是你需要查找的其他类型。你可以通过

```
*Iterator
```

来获取指向实际对象的指针。或者，你可以直接通过

```
Iterator->YourFunction()
```

来调用你需要的成员函数。

4.3 类对象的销毁

如今，一个类走到了其生命的尽头。我们希望销毁它，从而获得其所占用的内存空间。我们应该采用什么样的方式？我并不认为这是一个简单的命题。事实上我会分几类来加以阐述：

纯C++类

如果你的纯C++类是在函数体中创建，而且不是通过new来分配内存，例如：

```
void YourFunction( )
{
    FYourClass YourObject=FYourClass();
    ....//Do something.
}
```

此时这个类的对象会在函数调用结束后，随着函数栈空间的释放，一起释放掉。不需要你手动干涉。

如果你的纯C++类是使用new来分配内存，而且你直接传递类的指针。那么你需要意识到：除非你手动删除，否则这一块内存将永远不会被释放。如果你忘记了，这将产生内存泄漏。

如果你的纯C++类使用new来分配内存，同时你使用智能指针TSharedPtr/TSharedRef来进行管理，那么你的类对象将不需要也不应该被你手动释放。智能指针会使用引用计数来完成自动的内存释放。你可以使用MakeShareable函数来转化普通指针为智能指针：

```
TSharedPtr<YourClass> YourClassPtr=MakeShareable(new YourClass())
```

笔者强烈建议，在你没有充分的把握之前，不要使用手动new/delete方案。你可以使用智能指针。

UObject类

UObject类的情况略有不同。事实上你无法使用智能指针来管理**UObject**对象。

前文已经提到，UObject采用自动垃圾回收机制。当一个类的成员变量包含指向UObject的对象，同时又带有UPROPERTY宏定义，那么这个成员变量将会触发引用计数机制。

垃圾回收器会定期从根节点Root开始检查，当一个UObject没有被别的任何UObject引用，就会被垃圾回收。你可以通过AddToRoot函数来让一个UObject一直不被回收。

Actor类

Actor类对象可以通过调用Destory函数来请求销毁，这样的销毁意味着将当前Actor从所属的世界中“摧毁”。但是对象对应内存的回收依然由系统决定。

第5章

从C++到蓝图

问题

虚幻引擎的蓝图真是太好用了，我该如何让蓝图能够调用我的C++类中的函数呢？

5.1 UPROPERTY宏

当你需要将一个UObject类的子类的成员变量注册到蓝图中时，你只需要借助UPROPERTY宏即可完成。

```
UPROPERTY( ... )
```

你可以传递更多参数来控制UPROPERTY宏的行为，通常而言，如果你要注册一个变量到蓝图中，你可以这样书写：

```
UPROPERTY(BlueprintReadWrite,VisibleAnywhere,Category="Object")
```

关于能够在UPROPERTY中使用的参数，请阅读官方文档的这一章节。

5.2 UFUNCTION宏

你也可以通过UFUNCTION宏来注册函数到蓝图中。下面是一个注册的案例：

```
UFUNCTION(BlueprintCallable,Category="Test")
```

其中BlueprintCallable是一个很重要的参数，表示这个函数可以被蓝图调用。可选的还有： BlueprintImplementEventBlueprintNativeEvent。前者表示，这个成员函数由其蓝图的子类实现，你不应该尝试在C++中给出函数的实现，这会导致链接错误。后者表示，这个成员函数提供一个“C++的默认实现”，同时也可以被蓝图重载。你需要提供一个“函数名_Implement”为名字的函数实现，放置于.cpp中。

第6章

游戏性框架概述

6.1 行为树：概念与原理

6.1.1 为什么选择行为树

在虚幻引擎3的时代，AI框架选择的是状态机来实现。甚至为了支持状态机编程，虚幻引擎的脚本语言Unreal Script专门增加了几个状态相关的关键字，足以看出虚幻引擎官方对状态机系统的重视。但是这个系统在虚幻引擎4中被行为树系统代替，状态机只在动画蓝图中保留（当然这并不妨碍你在任何地方书写状态机，毕竟用C++实现一个状态机模式并不复杂）。

那么，是什么促使Epic做出这样的决定呢？简而言之，同样的AI模式，用状态机会涉及大量的跳转，但是用行为树就相对来说更加简化。同时由于行为树的“退行”特点，也就是“逐个尝试，不行就换”的思路，更加接近人类的思维方式，因此当你熟悉了行为树的框架之后，能够更加快速地撰写AI相关的代码。

6.1.2 行为树原理

在对虚幻引擎的行为树进行介绍之前，我认为应该先介绍“行为树”。“行为树”是一种通用的AI框架或者说模式，其并不依附于特定的

引擎存在，并且虚幻引擎的行为树也与标准的行为树模式存在一定的差异。

现在关于行为树的讨论并不多，因此我将会简单介绍行为树的一些内容。请看如图6-1所示的行为树案例1： Selector。

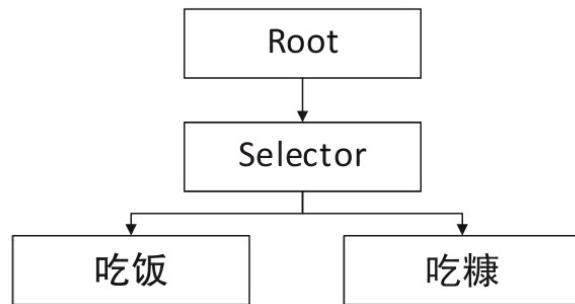


图6-1 行为树案例1： Selector

我们会发现，这是一个由节点、连接线构成的行为树。行为树包含三种类型的节点：

- 流程控制：包含Selector选择器和Sequence顺序执行器（关于平行执行parallel节点，暂时不做分析）。
- 装饰器：对子树的返回结果进行处理的节点。
- 执行节点：执行节点必然是叶子节点，执行具体的任务，并在任务执行一段时间后，根据任务执行成功与否，返回true或者false。

让我们看一个具体的例子，如图6-2所示：

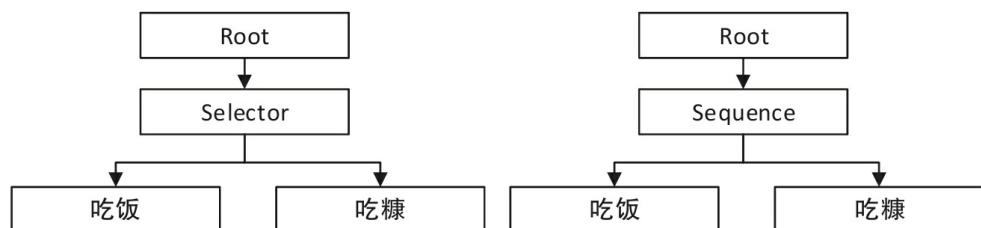


图6-2 行为树案例

除去根节点Root，Selector就是一个流程控制节点。Selector节点会从左到右逐个执行下面的子树，如果有一个子树返回true，它就会返回true，只有所有的子树均返回false，它才会返回false。这就类似于日常生活中“几个方案都试一试”的概念。

反映到如图6-2所示的行为树案例，对应左侧的图片。这个行为树实际上讲述了一段艰难的故事：在荒年，如果吃不起饭的时候，就只能选择吃糠了。

假如流程节点被换为了Sequence，那么Sequence节点就会按顺序执行自己的子树，只有当前子树返回true，才会去执行下一个子树，直到全部执行完毕，才会向上一级返回true。任何一个子树返回了false，它就会停止执行，返回false。类似于日常生活中“依次执行”的概念。把一个已有的任务分为几个步骤，然后逐个去执行，任何一个步骤无法完成，都意味着任务失败。也就是说，这一次的行为树表达了这样的概念：荒年好不容易收了点大米，先做成饭，然后慢慢吃，每天吃点饭之后，就开始吃糠，直到吃饱。

仔细归纳一下，我们会发现这两个行为树其实包含了许多信息：

Selector版本

终止性 只要能通过吃饭把自己吃饱，绝不吃糠，表达了一种“今朝有酒今朝醉”式的享乐主义理念。换句话说，只要“吃饭”节点返回成功，今天就算过去了，直接向Root返回成功。

优先级 即使是Selector选择器，依然具有优先级。一旦饿了优先找饭吃，而不是找糠。先吃好的，吃饱了再说。只有当“吃饭”节点返回失败的情况下，才开始尝试优先级更低的节点。

只要有一个成功便是成功，全部失败才失败 只要找到一点能吃的东西，吃饱了都算数。只有饭也没有，糠也没了，什么吃的都找不到了，才向Root汇报失败——没办法，真的没东西吃了。

Sequence版本

顺序性 这可能是一家精打细算的主人，即使有饭吃，也得一边吃点饭，一边吃糠。这是为了细水长流，以后每天都有点饭吃。也就是说，当“吃饭”返回成功的时候，需要继续执行接下来的节点。

任何一个步骤失败都失败，全部步骤做完才成功 当一点饭都没有的时候，主人家陷入了绝望的境地，也不去尝试吃糠了，直接向Root节点汇报“没办法了，真的失败了”。

标准的装饰器节点，是对子树返回的结果进行处理，再向上一级进行返回的。例如Force Success节点，就是强制让子树返回true，不管子树真正返回的是什么。如上文例子，假如在Root上加了一个Force Success装饰器节点，那就像是古代有些鱼肉百姓的官员，一方面对下面报上来的饥荒情况心知肚明，另一方面又不停向上级返回“成功”。

日常生活中的很多行为，都可以被这样的方式总结出来。而行为树对行为进行分析的关键在于，一定要从宏观到微观。先切分大的步骤，再逐步细化。以上班为例：

上班主要分为三个步骤：准备阶段，交通阶段，上楼阶段。

这三个步骤是按顺序执行的，不能分割。任何一个步骤出问题，都会导致上班不成功，比如你准备阶段不成功（“我再睡五分钟”，没想到醒过来已经是中午了），或者交通阶段不成功（被车撞飞导致无法继续

前往公司），结果都是上班不成功，你今天的工资肯定没了。

那么我们要继续细分，以细分交通阶段为例。对于通常的城市上班族来说，基本上能选择的交通工具，按照优先级排列如下：

1. 地铁：因为地铁基本上通行时间固定，而且地铁很少挤不上去，所以会成为上班族首选的交通工具。
2. 开车：可能由于特殊原因，今天地铁整修，或者是被水淹了之类，总之地铁选择不了的情况下，就会考虑开车上班，虽然堵了一点，但是至少能到公司。
3. 走路：结果没想到大家都想开车上班，于是马路被堵得彻底走不动了。这时候就只能选择走路去公司了。

因此，交通阶段是一个Selector，是从多个加权方案（有优先级）中逐个尝试的。由此大家会发现，Selector与普通的随机选择还是有区别的，我们能够通过顺序来定义“从一般到特殊”的AI行为。这是行为树很强大的一个地方。

同样的，上楼阶段也有可能出现选择电梯或者楼梯的选择，这里不再赘述。最终应该是如图6-3所示的行为树案例3：上班。

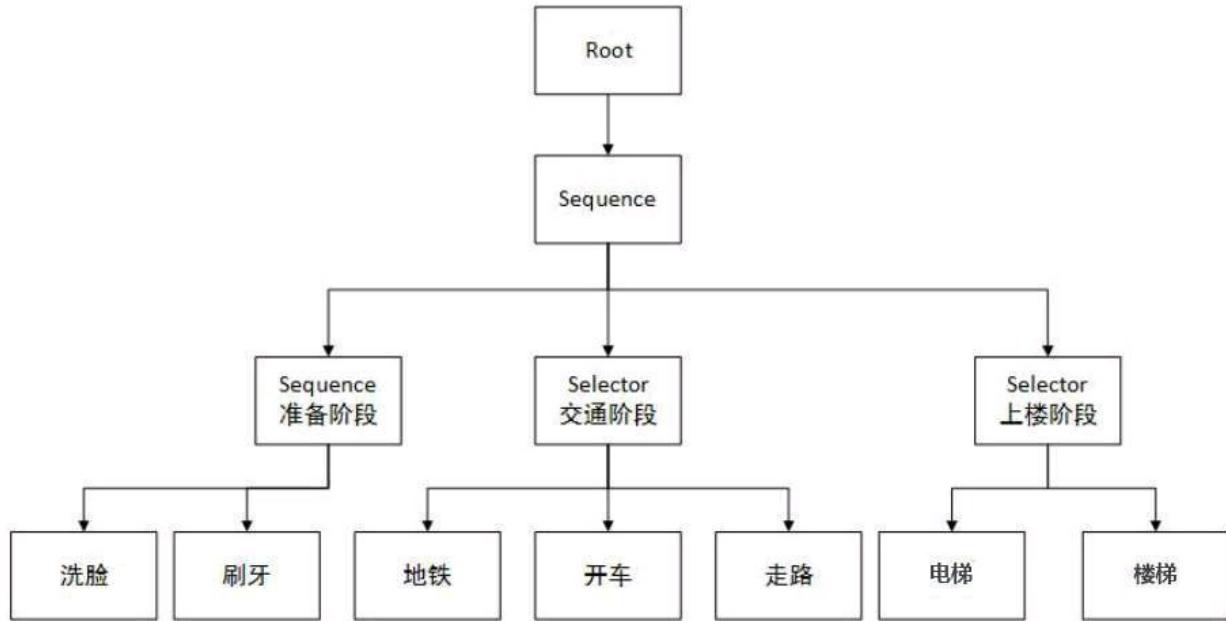


图6-3 行为树案例3:上班

通过这个行为树，我们会发现，在某个状况下，这个AI会按照这样
的方式执行：

1. 首先进入准备阶段：
 - a. 刷牙。
 - b. 洗脸。
2. 准备阶段完成，开始准备去上班：
 - a. 看看地铁能不能坐：
 - i. 走到地铁站，发现地铁因为修理被关闭。
 - ii. 返回false（回到家中）。
 - b. 选择开车方案：
 - i. 开车出门。
 - ii. 返回true，顺利到达上班地点。
 - c. 交通阶段完成，开始上楼：
 - i. 尝试电梯，发现电梯坏掉了，返回false。

ii. 尝试走楼梯，终于到达了办公室。

这可能是一个很倒霉的AI，不过它做出了一套让我们都能够认为很自然的行为。而不是因为地铁封闭，就傻傻地站在地铁口等待。这就是行为树系统的威力。

6.2 虚幻引擎网络架构

6.2.1 同步

从第一个联机游戏开始，同步就成为了一个重点的研究对象。随着需要同步的人数不断变多，联机同步架构的设计也在不断地变动。

最早的联机同步按照点对点网络的思路进行设计。也就意味着，假如开了一个房间，有4个玩家加入，那么玩家1输入每一个指令与消息，都会发往其他3个人，从而让4个人的画面得以一致。大致类似于，如果你按了一下W键，那么这个前进的指令就会发送到所有你连接的人的电脑上。然后你所控制的那个人物都会向前移动一点点。

在某些早期的联机游戏中，为了保证所有人的同步，甚至采用过更极端一些的方法，如强制所有人更新频率一致。

点对点同步带来的弊端相当得多，毕竟这是一个网状结构。例如：

1. 由于点对点同步带来的传输消耗，因此网络传输压力会很大。
2. 由于点对点同步不存在“权威”性，因此当其中一个人作弊时，会影响所有的客户端。且很难判定“作弊”。

因此，经典的服务器-客户端架构模型产生了。

一台（在当年）具有较高性能的主机被选出来，作为中心服务器。所有的“游戏性相关指令”都会被发往中心服务器进行处理，随后中心服务器会把世界的状态同步到各个客户端。

于是之前的网状结构变为了星型结构，且出现了权威服务器的概念。也就意味着，作弊变得更加困难。无法通过直接传递坐标（只能传递游戏性相关指令）给服务端，就算是本地客户端，坐标也来自于服务端同步过来的信息。

也就是说，我们能把游戏框架切分为两个部分：一部分是“指令”，是对游戏世界造成影响的代码请求，比如“人物前移3米”“人物挥刀碰到了怪物”；另一部分是“状态”，是游戏世界的各种数值状态，比如“当前人物生命值”“当前怪物生命值”。客户端只能向服务端发送“指令”，服务端根据指令处理后，改变游戏世界的状态，并将状态同步给每一个客户端。

这是相当优秀的一个设计，对同步考虑了非常多。因此，被作为了现在绝大多数网络同步模型的基本思路。不过这是不够的，因为有另一个非常基本的问题，那就是延迟。而为了解决延迟问题，虚幻引擎3提出了广义的客户端-服务端模型的概念。

6.2.2 广义的客户端-服务端模型

对于这个模型，其实虚幻引擎官方在Unreal Development Kit的文档UDN中，有一句非常精彩的表述：

客户端是对服务端的拙劣模仿

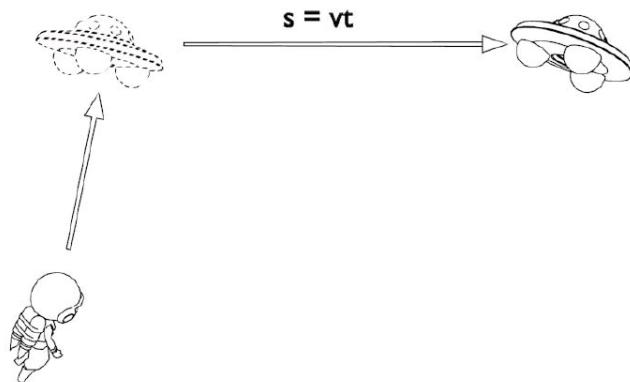
这句话的意思是说，客户端自己也同样运行着一个世界，并不断预测服务端的行为。从而不断更新当前世界，以最大程度地接近服务端的世界。譬如说，在虚幻引擎3的时代，服务端会不断同步当前对象的位置和速度到客户端，由于广泛存在着网络延时，因此当这个状态信息到达客户端时，实际上这个状态已经过时了。那么客户端怎么办呢？

让我们把思路扭转一下，也就是说，客户端不再试图去“同步”服务端，而是去“模仿”服务端。这就是说，我们承认“延迟”客观存在，只要我们的客户端模仿得别太差劲，那么玩家是可以接受这样的效果的。客户端可以根据同步数据发送时的当前对象的位置与速度，加上数据发送的时间，猜测出当前对象在服务端的可能位置。并且通过修正当前世界（比如调整当前对象的速度方向，指向新的位置），去模仿服务端位置。如果服务端的位置和客户端差距太大，就强行闪现修正。

而拙劣二字，就是在强调，服务端的世界是绝对正确的，而客户端则是不断试图猜测服务端当前时间的状态。

打个比方，如图6-4所示：假设一个飞行员在追踪一个UFO（不明飞行物），飞行员看不到那个UFO的位置，只能从基地给他的报告中获得UFO的位置与速度向量。如何才能尽可能靠近UFO呢？因为飞行员如果直接向基地给出的报告位置飞行，那么由于飞到那个位置需要一定时间，等飞行员飞到，UFO已经不在那儿了。飞行员可以选择根据自己飞行的时间、UFO当前位置、UFO的速度，猜测出一个位置，那个位置是当**UFO**保持当前速度方向、大小不变的情况下，自己的飞机最终一定会在那里和**UFO**汇合，然后向那个位置飞行。然后不断根据基地的信息修正，于是就能尽可能保证靠拢UFO的行动位置。

(a) 直接向服务器给出的目标地点追踪
到达后发现已经离开



(b) 借助目标当前速度和目标位置进行预测
向可能汇合的位置移动

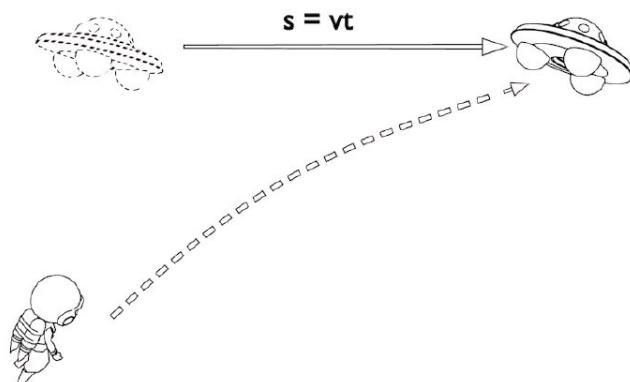


图6-4 网络同步的比喻：飞行员与UFO，作者王德立，已取得授权

再次反思我们刚才讨论的例子，就会发现，客户端的体验相对来说是非常流畅的。只要网络延迟不太大，那么客户端的对象就不会发生瞬移。这也就解释了有些采用同样模型的游戏中存在的“Ping神”现象。就是说当某个人延迟在阈值上下波动时，一会儿客户端会去用速度调整的方式修正位置（在地上滑来滑去），一会儿客户端又会直接把这个人的位置强行同步（滑了一会儿一下子又传送到原地）。

第7章

引擎系统相关类

问题

官方文档介绍的内容不多，我不知道我要的功能引擎有没有提供，怎么办？

本章希望向读者介绍一些笔者在开发过程中积累的虚幻引擎自带的功能。有些功能隐藏于代码中，没有出现在官方文档，因此尽可能介绍给读者，避免读者重复“造轮子”。

7.1 在虚幻引擎4中使用正则表达式

正则表达式，又称正规表示法、常规表示法。

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符，以及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。

在虚幻引擎4使用正则表达式，首先，我们要添加头文件。

```
#include "Regex.h"
```

需要注意的是，此头文件是放在Core模块里的。一般我们不需要额外在Build.cs里添加了。

然后，我们需要写表达式的内容。

如： FRegexPattern TestPattern(TEXT("\^d{5,12}"));

其中TEXT里的内容就是正则表达式的具体内容。

接下来，我们要用到另一个类： FRegexMatcher。

FRegexMatcher主要作用是驱动正则表达式的运行，因为 FRegexPattern只是一个表达式，其不具备任何作用。除了一个构造函数，其没有其他函数了。

FRegexMatcher提供多个函数，如返回查找字符的起始位置、结束位置、设置查找区间等。

但是我们最常用的还是FindNext()这个函数。它会返回一个bool值，表示是否查找到匹配表示式的内容。

代码示例：

```
FString TextStr("ABCDEFGHIJKLMN");
FRegexPattern TestPattern(TEXT("C.+H"));
FRegexMatcher TestMatcher(TestPattern, TextStr);
if (TestMatcher.FindNext()) {
    UE_LOG(MyLog, Warning, TEXT("找到匹配内容 %d -%d"),
    TestMatcher.GetMatchBeginning(),
```

```
TestMatcher.GetMatchEnding());//输出 找到匹配内容 2-8  
}
```

7.2 FPaths类的使用

在Core模块中，虚幻引擎4提供了一个用于路径相关处理的类——FPaths。在FPaths中，主要添加3类“工具”性质的API。

1. 具体路径类，如：FPaths::GameDir()可以获取到游戏根目录。
2. 工具类，如：FPaths::FileExists()用于判断一个文件是否存在。
3. 路径转换类，如：FPaths::ConvertRelativePathToFull()用于将相对路径转换为绝对路径。

由于FPaths类中的函数众多，这里不一一列举。

7.3 XML与JSON

XML

在Xml Parser模块中，提供了两个类解析XML数据，分别是FastXML与FXMLFile。相对而言，用FXMLFile更方便一些。所以本例使用FXMLFile。

首先，创建一个XML文件。示例内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<note name="Ami" age="100">
    <to>George</to>
    <from>John</from>
    <heading>Reminder </heading>
    <body>Don't forget the meeting!</body>
    <list>
        <line>Hello</line>
        <line>World</line>
        <line>PPPPP</line>
    </list>
</note>
```

解析XML代码如下（需要引入相应头文件）：

```
FString xmlFilePath =
FPaths::GamePluginsDir() / TEXT("SimpleWindow/Resources/Test.xml")
FXmlNode* xml = new FXmlNode();
xml->LoadFile(xmlFilePath);
FXmlNode* RootNode = xml->GetRootNode();
FString from_content =
RootNode->FindChildNode("from")->GetContent();
UE_LOG(LogSimpleApp , Warning, TEXT("from=%s"), *from_content);
FString note_name = RootNode->GetAttribute("name");
UE_LOG(LogSimpleApp , Warning, TEXT("note @name= %s"), *note_name
TArray<FXmlNode*> list_node
```

```
= RootNode->FindChildNode("list")->GetChildrenNodes();

for (FXmlNode* node : list_node)
{
    UE_LOG(LogSimpleApp, Warning, TEXT("list :%s "), *(node->
        GetContent()));
}
```

JSON

JSON解析需要用到JSON模块以及Include"Json.h"。使用示例：

```
FString JsonStr = "[{\\"author\\":\\"Tim\\"},{\\"age\\":\\"100\\"}]]";
TArray<TSharedPtr<FJsonValue>> JsonParsed;
TSharedRef< TJsonReader<TCHAR> > JsonReader = TJsonReaderFactory<
bool BFlag = FJsonSerializer::Deserialize(JsonReader, JsonParsed)
{
    UE_LOG(LogSimpleApp, Warning, TEXT("解析Json成功"));
    FString FStringAuthor = JsonParsed[0]->AsObject()->GetStringF
    UE_LOG(LogSimpleApp, Warning, TEXT("author = %s"), *FStringAut
}
```

7.4 文件读写与访问

虚幻引擎提供了与平台无关的文件读写与访问接口，即 FPlatformFileManager。考虑到许多读者有读写自己文件的需求，故花一定篇幅重点阐述文件读写。

该类定义于头文件PlatformFilemanager.h中，所属模块为Core，一般虚幻引擎会默认包含本模块，如果没有，请手动在当前模块的.build.cs 中包含。

通过以下调用：

```
FPlatformFileManager::Get()->GetPlatformFile();
```

能够获得一个IPlatformFile类型的引用。这个接口即提供了通用的文件访问接口，读者可以自行查询。

虚幻引擎之所以这么麻烦，是为了提供文件的跨平台性。如果纯粹提供静态函数或者全局函数，会增加代码的复杂程度。而且 FPlatformFileManager作为全局管理单例，能够更有效地管理文件读写操作。

以下文件获得函数调用的具体参数信息：

\Engine\Source\Runtime\Core\Public\GenericPlatform\GenericF

虚幻引擎提供的比较常用的函数如下：

拷贝函数 提供文件目录和文件的拷贝操作：

CopyDirectoryTree 递归拷贝某个目录；

CopyFile 拷贝当前文件。

创建函数 提供创建文件和目录的操作，目录创建成功或者目录已经存在都会返回真：

CreateDirectory 创建目录；

CreateDirectoryTree 创建一个目录树，即给定一个路径字符串，如果对应路径的父目录不存在，也会被创建出来。

删除函数 删除指定目录或文件，成功删除返回真，否则失败：

DeleteDirectory 删除指定目录；

DeleteDirectoryRecursively 递归删除指定目录；

DeleteFile 删除指定文件。

移动函数 只有一个函数MoveFile，在移动文件时用。

属性函数 提供对文件、目录的属性访问操作：

DirectoryExists 检查目录是否存在；

FileExists 检查文件是否存在；

GetStateData 获得文件状态信息，返回FFileStatData类型对象，这个对象其实包含了足够的状态信息，接下来的一系列函数也只是提供了快捷访问的接口。具体包含信息如表7-1所示：

表7-1 GetStateData包含信息

属性名	类型	含义
AccessTime	FDateTime	上次访问时间。如果无效则返回 FDateTime::MinValue
bIsDirectory	bool	是否是路径
bIsReadOnly	bool	是否是只读
CreationTime	FDateTime	返回文件的创建时间
FileSize	int64	返回文件大小, 单位为 byte, 如果文件大小未知返回 -1
ModificationTime	FDateTime	返回文件的上次修改时间, 如果无效返回 FdateTime::MinValue

GetAccessTimeStamp 获得当前文件上一次访问的时间;

SetTimeStamp 设置文件的修改时间;

FileSize 获得文件大小, 如果文件不存在返回-1;

IsReadOnly 文件是否只读。

遍历函数 该类函数都需要传入一个FDirectoryVisitor或 FDirectoryStatVisitor对象作为参数。你可以创造一个类继承自该类, 然后重写Visit函数。每当遍历到一个文件或者目录时, 遍历函数会调用Visitor对象的Visit函数以通知执行自定义的逻辑:

IterateDirectory 遍历某个目录;

IterateDirectoryRecursively 递归遍历某个目录;

IterateDirectoryStat 遍历文件目录状态, Visit函数参数为状态对象而非路径字符串;

IterateDirectoryStatRecursively 同上，递归遍历。

读写函数 最底层的读写函数往往返回IFileHandle类型的句柄，这个句柄提供了直接读写二进制的功能。如果非绝对必要，可以考虑更高层的API，下文有讲述：

OpenRead 打开一个文件用于读取，返回IFileHandle类型的句柄用于读取；

OpenWrite 打开一个文件用于写入，返回IFileHandle类型的句柄。

同时，针对一些极其普遍的需求，虚幻引擎提供了一套更简单的方式用于读写文件内容，即FFileHelper类，位于CoreMisc头文件中，提供以下静态函数：

LoadFileToArray 直接将路径指定的文件读取到一个TArray<uint8>类型的二进制数组中；

LoadFileToString 直接将路径指定的文本文件读取到一个FString类型的字符串中。请注意，字符串有长度限制，不要试图读取超大文本文件；

SaveArrayToFile 保存一个二进制数组到文件中；

SaveStringToFile 保存一个字符串到指定文件中；

CreateBitmap 在硬盘中创建一个BMP文件；

LoadANSITextFileToStrings 读取一个ANSI编码的文本文件到一

个字符串数组中，每行对应一个FString类型的对象。

7.5 GConfi类的使用

在我们平时的开发过程中，会经常有读写配置文件需求，需要虚幻引擎4提供底层的文件读写功能，而且C++原生也提供操作系统级别的文件读写。但读文件、写文件及解析内容等操作，还是不够方便。这时候我们就可以用虚幻引擎4提供的专门读写配置文件的类——GConfi。

GConfi类提供了非常方便的API让我们可以快速读写配置。而且GConfi是属于Core模块的类，一般也不需要进行添加模块引用。接下来我们分别看一下具体使用方法。

7.5.1 写配置

```
GConfig->SetString(  
TEXT("MySection"),  
TEXT("Name"),  
TEXT("李白"),  
FPaths::GameDir() / "MyConfig.ini");
```

以上即为一个简单的写配置示例。只需要调用GConfi的SetString函数就可以了。这个函数是保存一个字符串类型的变量到配置文件里。此函数共4个参数，第一个参数是指定Section，即一个区块（可以理解为一个分类）；第二个参数是指定此配置的Key；相当于此配置的具体名

字；第三个参数为具体的值；最后一个参数是指定配置文件的路径，如果文件不存在，会自动创建此文件。

除了SetString外，GConfig针对各种类型的数据都有相应的函数，如SetInt、SetBool、SetFloat等。具体的使用方法与SetString基本一样。

7.5.2 读配置

```
FString Result;
GConfig->GetString(
TEXT("MySection"),
TEXT("Name"),
Result,
FPaths::GameDir() / "MyConfig.ini");
```

与写配置不同，读配置则采用Get系列的函数。而且第三个参数从具体的值变为一个变量的引用。

使用GConfig来读写文件操作，变得非常简单。但需要注意的一点是写文件的时候，通常在运行过程中，进行写入配置操作值并不会马上写入到文件。如果你需要马上生效，则可以调用GConfig->Flush()函数。

7.6 UE_LOG

7.6.1 简介

Log作为开发中经常用到的功能，可以在任何需要的情况下记录程序运行的情况。Log通常有以下几个作用。

1. 记录程序运行过程，函数调用过程。
2. 记录程序运行过程中，参与运算的数据信息。
3. 将程序运行中的错误信息反馈给开发团队。

7.6.2 查看Log

Game模式

在Game（打包）模式下，记录Log需要在启动参数后加-Log。

编辑器模式

在编辑器下，需要打开Log窗口（Window->DeveloperTools->OutputLog）。

7.6.3 使用Log

```
UE_LOG(LogMy, Warning, TEXT("Hello World"));
UE_LOG(LogMy, Warning, TEXT("Show a String %s"), *FString("Hello"));
UE_LOG(LogMy, Warning, TEXT("Show a Int %d"), 100);
```

UE_LOG宏输出Log，第一个参数为Log的分类（需要预先定义）。第二个参数为类型，有**Log**、**Warning**、**Error** 三种类型。这三种类型

区别是颜色不同，Log为灰色，Warning为黄色，Error为红色。具体的输出内容为TEXT，可以根据需要自行构造。几种常用的符号如下：

1. %s 字符串 (FString)
2. %d 整型数据 (int32)
3. %f 浮点形 (float)

7.6.4 自定义Category

虚幻引擎4提供了多种自定义Category的宏。读者可自行参考LogMactos.h文件。这里介绍一种相对简单的自定义宏的方法。

```
DEFINE_LOG_CATEGORY_STATIC(LogMyCategory, Warning, All);
```

在使用DEFINE_LOG_CATEGORY_STATIC自定义Log分类的时候，我们可以将此宏放在你需要输出Log的源文件顶部。为了更方便地使用，可以将它放到PCH文件里，或者模块的头文件里（原则上是将Log分类定义放在被多数源文件include的文件里）。

7.7 字符串处理

在虚幻引擎中，“文字”类型其实是一组类型： FName， FText和 FString。这三种类型可以互相转换。当然还有TCHAR类型，只不过TCHAR不是虚幻引擎定义的字符串类。虚幻引擎把字符串进行细分，为的是加快访问速度，以及提供本地化支持。

FName

简单而言，FName是无法被修改的字符串，大小写不敏感。从语义上讲，名字也应该是唯一的。不管同样的字符串出现了多少次，在字符串表里只被存储一次。而借助这个哈希表，从字符串到FName的转换，以及根据Key查询FName会变得非常快速。

FText

FText表示一个“被显示的字符串”。所有你希望“显示”的字符串都应该是FText。因为FText提供了内置的本地化支持，也通过一张查找表来支持运行时本地化。FText不提供任何的更改操作，对于被显示的字符串来说，“修改”是一个非常不安全的操作。

FString

FString是唯一提供修改操作的字符串类。同时也意味着FString的消耗要高于FName和FText。

事实上，一般我们都使用FString来传递。尽管如此，Slate控件的文字参数往往是FText。这是为了强制要求本地化。

7.8 编译器相关技巧

7.8.1 “废弃”函数的标记

在虚幻引擎中，有“废弃函数”的概念。准备废弃或者更改一个函数的时候，虚幻引擎不会立即废弃（否则影响范围太大），而是在编译期

给出了一个警告。警告有点像这样： function Please update your code to the new API before upgrading to the next release, otherwise your project will no longer compile。需要注意的是，如果你使用编译器宏message，会在任何时候都会输出消息。只要你的这段代码被编译，就会输出。而虚幻引擎的废弃函数则是在“被调用”的时候才会输出。

虚幻引擎对不同平台提供了不同的宏定义，对GCC使用`_attribute_`关键字，对Visual Studio使用`_declspec`关键字。然后调用`deprecated`关键字来输出。还有许许多多的编译器关键字可以用来实现诸多效果。

7.8.2 编译器指令实现跨平台

虚幻引擎被设计为一个跨平台的引擎。但是虚幻引擎所有平台公用一套源代码。那么如何才能完成对不同平台的兼容呢？

至少在Windows平台下的程序入口点，即WinMain函数，在Linux下是没有的。

那么这该如何是好？一种传统的办法是通过多态来解决。即存在一个基类，定义了抽象的接口，独立于操作系统存在。然后每个操作系统对应版本继承自这个基类，然后做出自己的实现。运行时根据当前操作系统来进行切换。

虚幻引擎部分采用了这种方案，但是对于一些情形，例如Main函数，采用多态跨平台就显得很困难了。于是虚幻引擎采用了编译期跨平台的方案，即：

准备多个平台的实现，通过宏定义来切换不同的平台。例如虚幻引擎有一个通用的类：FPlatformMisc，包含了大量平台相关的工具函数。也有一系列的平台相关实现，如Linux下是FLinuxPlatformMisc，随后通过一个typedef来完成，即：
typedef FLinuxPlatformMisc
FPlatformMisc;。于是就完成了编译期跨平台的过程。

那么有读者就要提出问题了，假如Windows定义一下，Linux定义一下，那不就全乱了吗？

说得对，所以实际上，.build.cs文件就会对此做出判断，根据编译平台的类型是Win64还是Linux，会包含不同的文件夹，而不是一股脑儿全部包含，就不会出现冲突。

7.9 Images

问题

我希望能够转换图片的类型！我希望能直接从硬盘导入图片作为贴图！应该怎么办呢？

虚幻引擎提供了ImagerWrapper作为所有图片类型的抽象层。其思想设计还是非常精妙的。我们可以这样来看待所有的图片格式类型：

1. 图片文件自身的数据是压缩后的数据，称为CompressedData。
2. 图片文件对应的真正的RGBA数据，是没有压缩的，且与格式无关的（不考虑压缩带来的损失）的数据，称为RawData。

3. 同样图片保存为不同的格式， RawData不变（不考虑压缩）， CompressedData会随着图片格式不同而不同。
4. 因此，所有图片的格式都可以被抽象为一个CompressedData和 RawData的组合，这就是ImageWrapper模块设计的思路。

那么这样的设计思路有什么用处呢？让我们看几个案例：

读取**JPG**图片

1. 从文件中读取为TArray的二进制数据；
2. 用SetCompressData填充为压缩数据；
3. 使用GetRawData即可获取RGB数据。

转换**PNG**图片到**JPG**

1. 从文件中读取为TArray的二进制数据；
2. 用SetCompressData填充为压缩数据；
3. 使用GetRawData即可获取RGB数据；
4. 将RGB数据填充到JPG类型的ImageWrapper中；
5. 使用GetCompressData，即可获得压缩后的JPG数据；
6. 使用FFileHelper写入到文件中。

我在这里给出一个转换的代码案例：

```
bool FSystemToolsPublic::CovertPNG2JPG(const FString& SourceName,
                                         check(SourceName.EndsWith(TEXT(".png")))&&TargetName.EndsWith(
                                         TEXT(".jpg")));
    IImageWrapperModule & ImageWrapperModule = FModuleManager::
LoadModuleChecked< IImageWrapperModule >(FName("ImageWrapper"));
```

```
IImageWrapperPtr SourceImageWrapper= ImageWrapperModule.  
CreateImageWrapper(EImageFormat::PNG);  
  
IImageWrapperPtr TargetImageWrapper= ImageWrapperModule.  
CreateImageWrapper(EImageFormat::JPEG);  
  
TArray<uint8> SourceImageData;  
TArray<uint8> TargetImageData;  
int32 Width, Height;  
const TArray <uint8>* UncompressedRGBA = nullptr;  
if (!FPlatformFileManager::Get().GetPlatformFile().FileExists  
(*SourceName))  
{  
    ///文件不存在  
    return false;  
}  
if (!FFFileHelper::LoadFileToArray(SourceImageData, * SourceName))  
{  
    ///文件读取失败  
    return false;  
}  
if (SourceImageWrapper.IsValid() && SourceImageWrapper ->  
SetCompressed(SourceImageData.GetData(), SourceImageData.  
Num()))  
{  
    if (SourceImageWrapper ->GetRaw(ERGBFormat::RGBA, 8,  
    UncompressedRGBA))  
{  
        Height=SourceImageWrapper ->GetHeight();
```

```
Width = SourceImageWrapper ->GetWidth();

if (TargetImageWrapper ->SetRaw(
    UncompressedRGBA ->GetData(),
    UncompressedRGBA ->Num(), Width, Height,
    ERGBFormat::RGBA, 8))

{

    TargetImageData = TargetImageWrapper
->GetCompressed();

    if (!FFFileManagerGeneric::Get().
        DirectoryExists(*TargetName))

    {

        FFFileManagerGeneric::Get().
        MakeDirectory(*FPaths::GetPath(
            TargetName), true);

    }

    return FFileHelper::SaveArrayToFile(
        TargetImageData , *TargetName);

}

}

return false;
}
```

基于同样的思路，我们可以用这样的方式来读取硬盘中的贴图：

1. 从硬盘中读取压缩过的图片文件到二进制数组，获得图片压缩后数据；

2. 将压缩后的数据借助ImageWrapper的GetRaw转换为原始RGB数据；
3. 填充原始的RGB数据到UTexture的数据中。

读者可以参考这一段来自虚幻引擎官方wiki的Rama先生的代码，笔者加上了注释：

```
UTexture2D * FSystemToolsPublic::LoadTexture2DFromBytesAndExtensi
{
    const FString& ImagePath ,
    uint8* InCompressedData ,
    int32 InCompressedSize ,
    int32 & OutWidth,
    int32 & OutHeight
)
{
    UTexture2D * Texture = nullptr;
    IImageWrapperPtr ImageWrapper = GetImageWrapperByExtention(Im
    if (ImageWrapper.IsValid() && ImageWrapper ->SetCompressed(
        InCompressedData , InCompressedSize))//读取压缩后的
    {
        const TArray <uint8>* UncompressedRGBA = nullptr;
        if (ImageWrapper ->GetRaw(ERGBFormat::RGBA, 8,
            UncompressedRGBA))//获取原始图片数据
        {
            Texture = UTexture2D::CreateTransient(
                ImageWrapper ->GetWidth(), ImageWrapper ->
                GetHeight(), PF_R8G8B8A8);
        }
    }
}
```

```
    if (Texture != nullptr)
    {
        //通过内存复制，填充原始RGB数据到贴图的数据中
        OutWidth = ImageWrapper ->GetWidth();
        OutHeight = ImageWrapper ->GetHeight()
        ;
        void * TextureData = Texture->
            PlatformData ->Mips[0].BulkData.
            Lock(LOCK_READ_WRITE);
        FMemory::Memcpy(TextureData,
            UncompressedRGBA ->GetData(),
            UncompressedRGBA ->Num());
        Texture->PlatformData ->Mips[0].
            BulkData.Unlock();
        Texture->UpdateResource();
    }
}
return Texture;
}

UTexture2D * FSystemToolsPublic::LoadTexture2DFromFilePath(
    FString & ImagePath,
    int32 & OutWidth,
    int32 & OutHeight
)
{
    //文件是否存在
```

```
if (!FPlatformFileManager::Get().GetPlatformFile().FileExists(*ImagePath))
{
    return nullptr;
}

//读取文件资源
TArray<uint8> CompressedData;
if (!FFFileHelper::LoadFileToArray(CompressedData, *ImagePath))
{
    return nullptr;
}

return LoadTexture2DFromBytesAndExtension(ImagePath,
    CompressedData.GetData(), CompressedData.Num(), OutWidth,
    OutHeight);
}
```

第二部分 虚幻引擎浅析

考虑到不少人对虚幻引擎存在一种奇妙的恐惧，尤其是认为虚幻引擎是一大堆无法理解的、如魔法与巫术混合的代码的集合体，被充满玄学的编译器编译而产生的一个巨大无比的机械。因此，笔者希望在本部分最开头向大家回答一个简单的问题：虚幻引擎有Main函数吗？

答案是，有。虚幻引擎本质而言也是一个C++程序。对于Windows平台而言，你可以在Launcher模块下的LaunchWindows.cpp中找到你熟悉的WinMain函数。

第8章 模块机制

8.1 模块简介

问题

虚幻引擎为什么需要引入模块机制？

任何一名C++程序员，一定被C++的项目配置所困扰过。你需要添加头文件目录、lib目录等。还需要对Debug模式、Release模式下不同的包含做出控制。这是相当复杂的一个问题。更不用说对于虚幻引擎来说，需要的编译模式远远不止这么几种。在编辑器模式下，有些类会被引入，在最终发布的游戏中，这些类又完全不再需要，如何处理这样的问题呢？

在虚幻引擎3的时代，使用MakeFile来模拟模块，而到了虚幻引擎4的时代，为了彻底解决这样的问题，虚幻引擎借助UnrealBuildTool引入了模块机制[\[1\]](#)。

仔细观察虚幻引擎的源码目录，会发现其按照四大部分：Runtime、Development、Editor、Plugin来进行规划，而每个部分内部，则包含了一个一个的小文件夹。每个文件夹即对应一个模块。

一个模块文件夹中应该包含这些内容。

- Public文件夹
- Private文件夹
- .build.cs文件

模块的划分是一门艺术。你需要知道的是，不同模块之间的互相调用并不是很方便。只有通过XXXX_API宏暴露的类和成员函数才能够被其他模块访问。因此，模块系统也让你从一开始就对自己的类进行精心的设计，以免出现复杂的类与类依赖。

8.2 创建自己的模块

问题

我能不能创建自己的模块呢？

通过切分模块，能够有效地切分自己代码的结构和框架，另外，区分编辑器模块和运行时模块也是必须要做的事情。

创建一个新的模块分为如下几步：

1. 创建模块文件夹结构；
2. 创建模块构建文件.build.cs；
3. 创建模块头文件与实现文件；
4. 创建模块预编译头文件PrivatePCH.h；

5. 创建模块的C++声明和定义。

8.2.1 快速完成模块创建

如果你只需要快速创建一个模块，不考虑模块的加载和卸载行为，也不考虑模块接口的公开和隐藏，则可以这样快速创建一个模块。

在你C++工程的Source文件夹下，创建一个新的模块文件夹，然后创建如下的文件结构：

```
/模块文件夹
  └── 模块名.Build.cs
  └── 模块名.h
  └── 模块名.cpp
```

各文件对应内容如下：

```
using UnrealBuildTool;
//类的名称与模块名称一致
public class pluginDev : ModuleRules
{
    public pluginDev(TargetInfo Target)
    {
        PublicDependencyModuleNames.AddRange(new string[] { "Core" });
        PrivateDependencyModuleNames.AddRange(new string[] { });
    }
}
```

模块名.h(pluginDev.h)

```
#pragma once  
#include "Engine.h"
```

模块名.cpp(pluginDev.cpp)

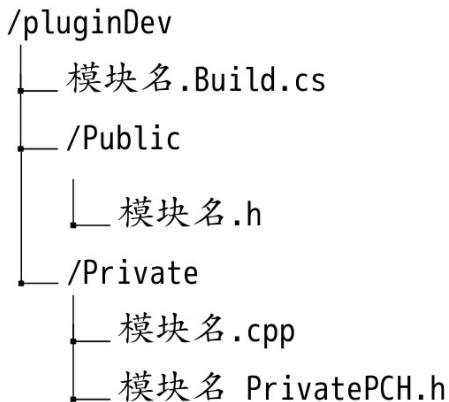
```
#include "pluginDev.h" //包含你刚刚创建的头文件  
IMPLEMENT_PRIMARY_GAME_MODULE( FDefaultGameModuleImpl , pluginDev
```

在这种情况下，你的“模块名.h”文件会被作为你的预编译头文件。当前模块其他类的.cpp文件，需要包含这个预编译头文件，否则将无法通过编译。

8.2.2 创建模块文件夹结构

对于一个标准的模块而言，前文的创建方式是不够的。

一个模块应当包含以下文件结构：



在你C++工程的Source文件夹下，创建一个新的模块文件夹，命名随意，此处以pluginDev为例。并在文件夹内添加Public和Private两个文件夹，形成如上所示的文件夹结构。

8.2.3 创建模块构建文件

一个模块需要用一个.Build.cs文件来告知UBT如何配置自己的编译和构建环境。因此你需要在上文的模块文件夹下，创建 pluginDev.Build.cs。并添加以下内容到文件中：

```
using UnrealBuildTool;
//类的名称与模块名称一致
public class pluginDev : ModuleRules
{
    public pluginDev(TargetInfo Target)
    {
        PublicDependencyModuleNames.AddRange(new string[] { "Core" });
        PrivateDependencyModuleNames.AddRange(new string[] { });
    }
}
```

```
    }  
}
```

8.2.4 创建模块头文件与定义文件

模块允许在C++层提供一个类，并实现StartupModule与ShutdownModule函数，从而自定义模块加载与卸载过程中的行为。创建对应的模块名.h和模块名.cpp，并填充内容如下。

完整版本的创建方式如下：

模块名.h(pluginDev.h)

```
#pragma once  
  
#include "ModuleManager.h"  
  
class FPluginDevModule : public IModuleInterface  
{  
  
public:  
    /** IModuleInterface implementation */  
    virtual void StartupModule() override;  
    virtual void ShutdownModule() override;  
};
```

模块名.cpp(pluginDev.cpp)

```
#include "PluginDevPrivatePCH.h"
```

```
//包含接下来要定义的模块预编译头文件名

void FPluginDevModule::StartupModule()
{
    //...这里书写模块启动时需要执行的内容
}

void FPluginDevModule::ShutdownModule()
{
    //...这里书写模块卸载时需要执行的内容
}

IMPLEMENT_MODULE(FPluginDevModule, pluginDev)
```

8.2.5 创建模块预编译头文件

预编译头文件能够加速代码的编译，因此当前模块公用的头文件可以放置于这个头文件中。

该头文件的标准命名方式为：“模块名PrivatePCH.h”，放置于Private文件夹中。内容可以为空。

当前模块所有的.cpp文件，都需要包含预编译头文件。

8.2.6 引入模块

对于游戏模块而言，引入当前模块的方式是在游戏工程目录下的Source文件夹中，找到工程名.Target.cs文件，打开后修改以下函数：

工程名.Target.cs

```
public override void SetupBinaries(
    TargetInfo Target,
    ref List<UEBuildBinaryConfiguration>
        OutBuildBinaryConfigurations ,
    ref List<string> OutExtraModuleNames
)
{
    OutExtraModuleNames.AddRange( new string[] { "pluginD" }
}
```

对于插件模块而言，方法是修改当前插件的.uplugin文件，该文件大概内容如下：

插件名.uplugin

```
{
    "FileVersion": 3,
    "Version": 1,
    "VersionName": "1.0",
    "FriendlyName": "插件名",
    "Description": "插件描述",
    "Category": "Other",
    "CreatedBy": "",
    "CreatedByURL": ""}
```

```
"DocsURL": "",  
"MarketplaceURL": "",  
"SupportURL": "",  
"Modules": [ //在这里引入模块  
 {  
     "Name": "插件模块名称",  
     "Type": "Editor", //模块加载类型  
     "LoadingPhase": "PostEngineInit" //模块加载时机  
 },  
,  
 "EnabledByDefault": true,  
 "CanContainContent": true,  
 "IsBetaVersion": false,  
 "Installed": false  
}
```

在Modules数组中添加Json对象，定义引入的模块。格式可以参考上文给出的案例。

8.3 虚幻引擎初始化模块加载顺序

本节希望给有需要的读者一个参考性质的模块加载顺序。实际上虚幻引擎模块加载分为两大部分，一部分是以硬编码形式硬性规定，另一部分则是松散加载。前半部分的加载顺序有自己的依赖原则。

注意，根据你引擎发布的版本不同

(Editor/Development/Shipping) , 模块加载也不相同。有些模块不会被加载。但是总体上说，模块的加载遵循这样的顺序：

1. 首先加载的是Platform File Module, 因为虚幻引擎要读取文件。
2. 接下来加载的是核心模块：
(FEngineLoop::PreInit → LoadCoreModules)。
3. 加载CoreUObject。
4. 然后在初始化引擎之前加载模块：
FEngineLoop::LoadPreInitModules。
5. 加载Engine。
6. 加载Renderer。
7. 加载AnimGraphRuntime。

根据你的平台不同，会加载平台相关的模块
(FPlatformMisc::LoadPreInitModules) , 在Windows平台下加载的是：

1. D3D11RHI (开启bForceD3D12后会加载D3D12)
2. OpenGLDrv
3. SlateRHIRenderer
4. Landscape
5. ShaderCore
6. TextureCompressor
7. Start Up Modules:FEngineLoop::LoadStartupCoreModules
8. Core (很有趣，虚幻引擎的核心Core模块加载的时机并不是在最初)
9. Networking

然后是平台相关模块，Windows平台下是：

1. XAudio2
2. HeadMountedDisplay
3. SourceCodeAccess
4. Messaging
5. SessionServices
6. EditorStyle
7. Slate
8. UMG
9. MessageLog
10. CollisionAnalyzer
11. FunctionalTesting
12. BehaviorTreeEditor
13. GameplayTasksEditor
14. GameplayAbilitiesEditor
15. EnvironmentQueryEditor
16. OnlineBlueprintSupport
17. IntroTutorials
18. Blutility

接下来，会根据启用的插件，加载对应的模块。然后是：

1. TaskGraph
2. ProfilerService

至此，虚幻引擎初始化完成所需的模块加载。

之后的Module加载，是根据情况来完成的。也就是说Unreal Editor自己来控制其需要加载什么样的程序。发布后的游戏因为没有携带

Unreal Editor，因此自然不需要加载这些模块。而笔者认为，对模块加载顺序的研究，主要的部分还是集中在引擎加载初期。后期的模块大多针对具体功能。

当然，有很多时候，我们需要处理模块依赖，这个时候其他模块的加载顺序也是有一定的研究价值的。

如果你好奇Editor模块的加载顺序，Editor模块加载是在UEditorEngine的Init中，被一个数组控制：

1. Documentation
2. WorkspaceMenuStructure
3. MainFrame
4. GammaUI
5. OutputLog
6. SourceControl
7. TextureCompressor
8. MeshUtilities
9. MovieSceneTools
10. ModuleUI
11. Toolbox
12. ClassViewer
13. ContentBrowser
14. AssetTools
15. GraphEditor
16. KismetCompiler
17. Kismet
18. Persona

- 19. LevelEditor
- 20. MainFrame
- 21. PropertyEditor
- 22. EditorStyle
- 23. PackagesDialog
- 24. AssetRegistry
- 25. DetailCustomizations
- 26. ComponentVisualizers
- 27. Layers
- 28. AutomationWindow
- 29. AutomationController
- 30. DeviceManager
- 31. ProfilerClient
- 32. SessionFrontend
- 33. ProjectLauncher
- 34. SettingsEditor
- 35. EditorSettingsViewer
- 36. ProjectSettingsViewer
- 37. Blutility
- 38. OnlineBlueprintSupport
- 39. XmlParser
- 40. UserFeedback
- 41. GameplayTagsEditor
- 42. UndoHistory
- 43. DeviceProfileEditor
- 44. SourceCodeAccess
- 45. BehaviorTreeEditor

- 46. HardwareTargetin
- 47. LocalizationDashboard
- 48. ReferenceViewer
- 49. TreeMap
- 50. SizeMap
- 51. MergeActors

以上内容是为了方便读者在引入模块的时候确定“哪些模块一定加载”。读者也可以根据这里的模块名称来大致推测自己需要的功能可能位于哪个模块。例如根据XmlParser模块名字，就能推测这里放置了XML解析需要的API。

8.4 道常无名： **UBT**和**UHT**简介

问题

模块机制很厉害，但是模块是用什么样的方式配置、编译、启动和运行呢？大牛们常常提到的“**UBT**”和“**UHT**”又是什么呢？

8.4.1 **UBT**

UBT概览

如果你下载了一份虚幻引擎的源代码，你能够在UBT（Unreal Build

Tool) 项目的Unreal Build Tool.cs文件中找到Main函数。

大致而言，UBT的工作分为三个阶段：

收集阶段 收集信息。UBT收集环境变量、虚幻引擎源代码目录、虚幻引擎目录、工程目录等一系列的信息。

参数解析阶段 UBT解析传入的命令行参数，确定自己需要生成的目标类型。

实际生成阶段 UBT根据环境和参数，开始生成makefile，确定C++的各种目录的位置。最终开始构建整个项目。此时编译工作交给标准C++编译器。

同时UBT也负责监视是否需要热加载。并且调用UHT收集各个模块的信息。

请注意，UBT被设计为一个跨平台的构建工具，因此针对不同的平台有相对应的类来进行处理。UBT生成的makefile会被对应编译器平台的ProjectFileGenerator用于生成解决方案，而不是一开始就针对某个平台的解决方案来确定如何生成。

再谈**WinMain**

在最开头我们谈到了WinMain函数在哪，解除了大家许多心理的戒备。但是其实接下来的问题更加严峻：WinMain函数最终怎么会被链接到.exe文件？

事实上，笔者也专门为此探究了一阵。如果你对此没什么兴趣，直接跳过本部分就行。

首先如果大家熟悉C++，就会知道，在Windows平台最终的.exe文件中，必须包含Main函数（C/C++控制台程序）或者WinMain函数（Windows）。否则是找不到应用程序的入口点的。而虚幻引擎的模块，最终大多被编译为DLL动态链接库文件。

如果你看过之前的模块加载（通过FModuleManager::LoadModule），你会发现这里有一大堆眼熟的模块DLL（Slate、SlateCore等）。

那么问题来了，为什么我们的Launcher模块没有被编译为DLL？是什么控制的呢？

答案是，在.target.cs文件控制。

我们现在的虚幻引擎4，被UE4Editor.Target.cs文件所控制编译，如果你是用的编译版引擎，你是看不到完整的编译控制的，所以你会困惑一些。其实，在.Target.cs文件中，如下所示：

```
public override void SetupBinaries(  
    TargetInfo Target,  
    ref List<OutBuildBinaryConfigurations> OutBuildBinaryConfigurations,  
    ref List<OutExtraModuleNames> OutExtraModuleNames  
)  
{  
    OutExtraModuleNames.Add("UE4Game");  
}
```

```
}
```

就完成了对二进制文件的设置工作。

在引擎源代码的UEBuildEditor.cs的SetupBinaries中，你会看到，你的BuildRules类在设置好自己的二进制数据输出后，OutBuildBinaryConfiguration数组会被添加一个特殊的成员：

Launcher模块会被设置为UEBuildBinaryTypes.Executable添加进来。作为.exe文件的包含模块。

此时.exe文件就会包含Launcher模块中平台对应的Main函数。对于Windows平台而言，WinMain函数会被链接进.exe文件。

当你双击虚幻引擎的Editor.exe后，WinMain函数被调用。虚幻引擎开始在你的系统上运行起来。

8.4.2 UHT

问题

前文有提到，**UHT**配合实现了反射机制，那**UHT**是如何完成的呢？

UHT（**Unreal Header Tool**）一个引擎独立应用程序

笔者不知道该如何形容这样的程序，只是勉强给出了一个我个人的命名。

引擎独立应用程序是指，这种应用程序依赖引擎。比如依赖UBT以配置编译环境，从而能跨平台编译，依赖引擎的某些模块。

但是这样的应用程序又不是一个引擎，也不是一个游戏。它最终输出为一个.exe文件。但是又不需要引擎完全启动，甚至不需要Renderer模块，以至于有可能只是一个命令行工具——比如UHT。

通过虚幻引擎源代码，你会发现UHT拥有自己的.target.cs和.build.cs文件。

在其.target.cs文件中，它把自己设置为exe的输出模块。

在.build.cs文件中，它指出了自己的依赖模块：

- Core
- CoreUObjects
- Json
- Projects

然后又包含了Launch模块的public/private文件夹，以便让自己能够调用GEngine->PreInit函数。

最终，UHT会被编译成一个.exe文件，通过命令行参数调用。

很奇妙不是吗？一个能够用来编译引擎的程序，居然依赖引擎本身。在本书的第三部分，将会教读者自己制作一个这样的程序。

UHT大致工作流程

UHT的Main函数在UnrealHeadToolMain.cpp文件中。这个文件提供了Main函数，并且也通过IMPLEMENT_APPLICATION宏声明了这是个独立应用程序。具体执行的内容如下：

1. 调用GEngineLoop->PreInit函数，初始化Log、文件系统等基础系统。从而允许借助UE_LOG宏输出log信息。
2. 调用UnrealHeaderTool_Main函数，执行真正的工作内容。
3. 调用FEngineLoop::AppExit退出。

那么，UnrealHeaderTool_Main函数到底干了什么？

UHT到底干了什么

首先，UBT会通过命令行参数告诉UHT，游戏模块对应的定义文件在哪。这个文件是一个.manifest文件。这是个由UBT生成的文件。如果你用记事本之类的应用打开，你会发现这是个Json字符串。

笔者把自己的一个工程中的这个Json字符串复制到了在线Json格式化工具，然后截了个图。这样你能更加直观地感受这个Json字符串装了些什么。

```

{
    "IsRuntimeGenerated": true,
    "PrivateHeaders": "F:\UE4\Source\\Runtime\\Core\\Public\\CoreObject.h",
    "GeneratedCPPfilenameBase": "F:\UE4\Source\\Runtime\\Core\\Generated\\"
}

```

The manifest file contains several entries for different modules, each defining its type (e.g., Runtime, Editor), source and output directories, and various compiler flags and include paths. For example, one entry for a 'TableComponent' module specifies 'ModuleType': 'Runtime', 'BaseDirectory': 'F:\UE4\Source\\Runtime\\TableComponent', 'IncludeBase': 'F:\UE4\Source\\Runtime\\TableComponent\\Classes', 'OutputDirectory': 'F:\UE4\Source\\Runtime\\TableComponent\\Generated\\', and 'PublicHeaders': 'F:\UE4\Source\\Runtime\\TableComponent\\Generated\\TableComponent.h'. Another entry for a 'CustomMeshComponent' module specifies 'ModuleType': 'Runtime', 'BaseDirectory': 'F:\UE4\Source\\Runtime\\CustomMeshComponent', 'IncludeBase': 'F:\UE4\Source\\Runtime\\CustomMeshComponent\\Classes', 'OutputDirectory': 'F:\UE4\Source\\Runtime\\CustomMeshComponent\\Generated\\', and 'PublicHeaders': 'F:\UE4\Source\\Runtime\\CustomMeshComponent\\Generated\\CustomMeshComponent.h'. The file also includes sections for 'PrivateHeaders' and 'GeneratedCPPfilenameBase'.

图8-1 .manifest文件内容

你会发现这个字符串包含了所有Module的编译相关的信息。包括各种路径，以及预编译头文件位置等。

然后UHT开始了自己的三遍编译： Public Classes Headers、 Public Headers、 Private Headers。第一遍其实是为了兼容虚幻引擎3时代的代码。虚幻引擎3时代的Classes文件夹会向Unreal Script暴露这些类。接下来的两遍则是解析头文件定义，并生成C++代码以完成需要的功能。生成的代码会和现有代码一起联合编译以产生结果。

虚幻引擎反射机制与超生游击队的故事

问题

UHT生成的是代码么？代码如何完成“类有哪些成员变量、成员函数”这样的信息的注册呢？

经过UHT的三遍编译，最终生成的是.generated.cpp和.generated.h这

两种文件。而我们理解UHT工作内容，也得根据这两个文件来理解。

让我们先来思考一个问题。众所周知，虚幻引擎的一个UClass，可以理解为包含UPROPERTY和UFUNCTION的一个数据结构：

UClass

- UPROPERTY

- ...

- UPROPERTY

- UFUNCTION

- ...

- UFUNCTION

随后虚幻引擎需要做的其实是，把C++类（也就是真正的“成员变量”和“成员函数”）与UClass中的对应的“Property数据类型”和“Function数据类型”绑定起来。

打个比方，UClass其实只是一张表，或者说类似户口本的东西。上面记录的是指向真实的“家庭”的指针，可能还包含一些额外的信息。假设我们拿到的是张家人的户口本（当然这里是假设的户口本，随便写了点信息），上面写着：

张大：
男性
1967年生
张二：
女性
1968年生

张三:

男性

1969年生

此时我们知道，这三个条目记录的是，张大、张二和张三的“额外信息”，或者说元数据metadata，但是到时候你找人，比如说你要拿着这个户口本找张大、张二和张三。你还得根据这个户口本条目找到对应的那个人，也就得通过一个指针。

现在问题是，虚幻引擎该怎么处理这个户口本机制呢？一种方式是，一开始编译时就把户口本都填好，放在一个文件里面。要找某家人的时候，就读取出来，开始查。但是大家都知道，每一次编译，函数的地址可能会发生变化，而每一次运行，函数映射到内存中的位置也可能不同。直接存储地址值是不行的。就比方说张家人每次人口普查之前都搬家（因为超生了，超生游击队），所以如果直接存储上次普查得到的地址，那么时过境迁，根据这个地址去找张家，实际找到的可能是李家，甚至这个地址都拆迁了。

所以虚幻引擎想了个办法。UHT不是存储的“每家人的户口本”，而是把“进行户口调查的过程”存储了下来。比方说，它存储了每个家庭有哪些人需要登记信息。然后在运行之初，借助某个特殊的技巧，在Main函数调用之前，逐个敲门让每家人进行登记。虽然张家人要搬家，没关系，跑得了和尚跑不了庙，我把这个城里的每个家庭找一遍，你总归还是要被我找到的。

当然，有些热心的读者要说了，万一张家又生了个娃（张四），岂不是记录不到了吗？虚幻引擎的UHT就是扮演着那个计生办的角色。你

的C++类，如果改动了头文件，肯定得编译吧？意味着你生娃总得给娃办一个户口吧，否则那个娃以后是黑户，书都读不了。所以当你给娃办一个户口（用UPROPERTY标记或者用UFUNCTION标记），编译的时候就被UHT看到了，它就把如何获取具体的信息（如何根据户口本找到张四）的步骤写在了.generated.cpp里面。

加载信息

仅仅记录获取信息的过程还不够，还需要在加载的时候调用记录好的过程来进行实际的记录。前文说道这是通过在Main函数调用前执行记录过程的技巧，实质上来说，方法是通过一个简单的C++机制：静态全局变量的初始化先于Main函数执行[\[2\]](#)。

在生成好的.generated.cpp文件中，会看到一个宏：

```
IMPLEMENT_CLASS
```

这个宏展开后实质上完成了两个内容：

1. 声明了一个具有独一无二名字的UClassCompiledInDefer<当前类名>静态全局变量实例。
2. 实现了当前类的GetPrivateStaticClass函数。

我们重点观察前一个步骤。这个类的构造函数调用了UClassCompiledInDefer函数。即：

1. 这个静态全局变量会在Main函数之前初始化。
2. 初始化就会调用该变量的构造函数。
3. 构造函数会调用UClassCompiledInDefer函数。
4. UClassCompiledInDefer会添加ClassInfo变量到延迟注册的数组中。

绕了这么大一圈，其实是希望能够在Main函数执行前，先执行UClassCompiledInDefer函数。这个函数带有“Defer”，是因为实际注册操作是后来执行的，但是在Main函数之前必须得“先摇个号”^[1]。这样虚幻引擎才知道有哪个类是需要延迟加载。借助这个技巧，虚幻引擎能够在启动前就给全部的类发个号，等时机成熟再一个一个加载信息。

[\[1\]](#) 据说在C++17中也会引入语言层面上的模块机制。当然这是后话，期待C++17标准的正式推出。

[\[2\]](#) 严格来说，无副作用的静态全局变量的初始化可以延迟到使用时，但是这不影响我们的分析。

第9章 重要核心系统简介

本章介绍了虚幻引擎中的一部分重要核心系统，这些系统与后文的讨论有关，故详细介绍。实际上虚幻引擎的Core模块包含大量的内容，例如TArray定义等，在此不做过多赘述。

9.1 内存分配

9.1.1 Windows操作系统下的内存分配方案

在Windows操作系统下，虚幻引擎是通过宏来控制，并在几个内存分配器中选择的。对应的代码如下：

```
FMalloc* FWindowsPlatformMemory::BaseAllocator()
{
    #if ENABLE_WIN_ALLOC_TRACKING
        _CrtSetAllocHook(WindowsAllocHook);
    #endif // ENABLE_WIN_ALLOC_TRACKING
    #if FORCE_ANSI_ALLOCATOR
        return new FMallocAnsi();
    #elif (WITH_EDITORONLY_DATA || IS_PROGRAM) &&
        TBB_ALLOCATOR_ALLOWED
```

```
    return new FMallocTBB();

#else

    return new FMallocBinned((uint32)(GetConstants().

        PageSize&MAX_uint32), (uint64)MAX_uint32+1);

#endif

}
```

也就是说，Windows平台提供了标准的Malloc（ANSI）、Intel TBB内存分配器，以及Binned内存分配器三个方案。

9.1.2 IntelTBB内存分配器

如果读者对IntelTBB内存分配器感兴趣，可以参考这本书：《Intel Threading Building Blocks编程指南》。

采用TBB内存分配，主要是出于以下的原因：

1. 虚幻引擎工作在多个线程，而标准内存分配为了避免出现内存分配BUG，是强制同一时间只有一个线程分配内存。导致内存分配的速度大幅度降低。
2. 缓存命中问题。CPU中存在高速缓存，而同一个缓存，一次只能被一个线程访问。如果出现比较特殊的情况，如两个变量靠在一起：

```
int a;
int b;
```

然后线程1访问a，线程2访问b。理论上此时可以并行。但是由于在加载a的时候，缓存把b的内存空间也加载进去，导致线程2在访问

的时候，还需要重新加载缓存。这带来相当大的CPU周期浪费，被称为“假共享”。

《游戏引擎架构》一书中，对内存分配方案做出了相当多的描述，其重点提到的也是两个方面：

1. 通过内存池降低malloc消耗。
2. 通过对齐降低缓存命中失败消耗。而虚幻引擎的目的相同，方法不同。

Intel TBB提供了scalable_allocator:不在同一个内存池中分配内存，解决由于多线程竞争带来的无谓消耗；cache_aligned_allocator:通过缓存对齐，避免假共享。

这一方案的代价是内存消耗量增加。对应其他平台的内存分配方案在这里不再做过多介绍。有兴趣的读者可以自行分析。

在虚幻引擎中，主要使用的还是scalable_allocator。为方便读者参考，将FAllocTBB::Malloc的代码摘录如下：

```
void* FAllocTBB::Malloc( SIZE_T Size, uint32 Alignment )
{
    IncrementTotalMallocCalls();
    MEM_TIME(MemTime -= FPlatformTime::Seconds());
    void* NewPtr = NULL;
    if( Alignment != DEFAULT_ALIGNMENT )
    {
        Alignment = FMath::Max(Size >= 16 ? (uint32)16 : (uint32)
```

```
        Alignment);

    NewPtr = scalable_aligned_malloc( Size, Alignment );

}

else

{

    NewPtr = scalable_malloc( Size );

}

if( !NewPtr && Size )

{

    OutOfMemory(Size, Alignment);

}

#endif UE_BUILD_DEBUG || UE_BUILD_DEVELOPMENT

else if (Size)

{

    FMemory::Memset(NewPtr, DEBUG_FILL_NEW , Size);

}

#endif

MEM_TIME(MemTime += FPlatformTime::Seconds());

return NewPtr;

}
```

9.2 引擎初始化过程

问题

虚幻引擎是怎么运行起来的？

引擎初始化简介

虚幻引擎初始化分为两个过程，即预初始化PreInit和Init。其具体实现由FEngineLoop这个类来提供。而在不同平台上，入口函数不同（如Windows平台下是WinMain，Linux平台下是Main），不同的入口函数最后会调用同样的FEngineLoop中的函数，实现跨平台。

预初始化

PreInit是预初始化过程，和初始化过程最显著的区别在于PreInit带有参数CmdLine，也就是说，能够获得传入的命令行字符串。

这个过程主要是根据传入的命令行字符串来完成一系列的设置状态的工作，大致来说，能够分为这样几个设置的内容：

1. 设置路径：当前程序路径，当前工作目录路径，游戏的工程路径。
2. 设置标准输出：设置GLog系统输出的设备，是输出到命令行还是何处。

并且也初始化了一部分系统，包括：

- 初始化游戏主线程GameThread，其实只是把当前线程设置为主线程。

- 初始化随机数系统（用过C语言随机数库的都知道，随机数是需要初始化的，否则同样的种子会产生出虽然随机但是一模一样的随机序列）。
- 初始化TaskGraph任务系统，并按照当前平台的核心数量来设置TaskGraph的工作线程数量。同时也会启动一个专门的线程池，生成一堆线程，用于在需要的时候使用。也就是说虚幻引擎的线程数量是远多于核心数量的。在我的电脑上，初始化的线程数量是：Task线程3个，线程池线程8个。

预初始化过程也会判断引擎的启动模式，是以游戏模式启动，还是以服务器模式启动。

在完成这些之后，会调用LoadCoreModules。目前所谓的CoreModules指的就是CoreUObject。具体为何CoreUObject需要这样额外的启动，会在分析UObject的时候进行分析。

随后，所有的PreInitModules会被启动起来。这些强大的模块是：引擎模块、渲染模块、动画蓝图、Slate渲染模块、Slate核心模块、贴图压缩模块和地形模块。

当这些模块加载完毕后，AppInit函数会被调用，进入引擎正式的初始化阶段[\[1\]](#)。

初始化

此时虚幻引擎进入初始化流程。所有被加载到内存的模块，如果有PostEngineInit函数的，都会被调用从而初始化。这一过程是借助

IProjectManager完成的。

由于Init的过程被分摊到了每个模块中，因此初始化的过程显得格外简洁。

主循环

虚幻引擎的主循环代码，如果为了符合虚幻引擎的源码引用规范（30行以内），那么可以被表述为以下这样：

```
while( !GIsRequestingExit )  
{  
    EngineTick();  
}
```

所以说虚幻引擎也是按照标准的引擎架构来书写的，至少游戏主线程是存在一个专门的引擎循环的，也就是EngineTick。注意，虚幻引擎的渲染线程是独立更新的，不在我们主循环分析的内容中，可以看后文虚幻引擎渲染架构的分析。

引擎的Tick按照以下的顺序来更新引擎中的各个状态：

- 更新控制台变量。这些控制台变量可以使用控制台直接设置。
- 请求渲染线程更新当前帧率文字。
- 更新当前应用程序的时间，也就是App::DeltaTime。
- 更新内存分配器的状态。

- 请求渲染线程刷新当前的一些底层绘制资源。
- 等待Slate程序的输入状态捕获完成。
- 更新GEngine，调用GEngine->Tick。
- 假如现在有个视频正在播放，需要等待视频播放完。之所以在GEngine之后等待，是因为GEngine会调用用户的代码，此时用户有可能会请求播放一个视频。
- 更新SlateApplication。
- 更新RHI。
- 收集下一帧需要清理的UObject对象。

有一些Tick的内容没有写在这里，具体希望了解有哪些内容，可以阅读LaunchEngineLoop中的内容。

那么现在大家最关心的内容其实集中在了，在GEngine->Tick中到底做了什么。其实这个内容也并不是非常复杂。只不过引擎考虑了相当多的情况，所以有许许多多的If判断，从而阻碍大家去理清。

从最正常的角度来说，GEngine->Tick最重要的任务是更新当前World。无论是编辑器中正在编辑的那个World，还是说游戏模式下只有一个的那个World。此时所有World中持有的Actor都会被得到更新。

请注意，笔者并没有说UObject会得到更新，事实上有许多UObject根本没有Tick功能。

那么其他的Tick内容则是从时间分片的角度来看待很多任务。这个的意思是说，很多任务可能无法在一次Tick中完成，否则会卡死游戏主线程，带来极其不好的游戏体验，因此会分在多次Tick函数中完成。当前Tick需要完成什么任务，则是在Tick中根据诸多的状态来决定的。例

如正在加载地图，则每次都Tick一下，加载一点点；正在动态载入地图（StreamLevel），或是正在进行HotReload，这些都会被分在不同的Tick中，以一次载入一点点的方式完成。

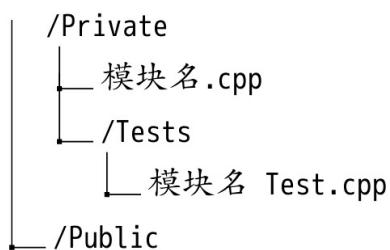
9.3 并行与并发

虚幻引擎的系统一开始就被设计为并行化，故对虚幻引擎并行、并发系统的研究非常重要，有助于理解接下来对渲染等过程的描述。同时，并行系统也是开发者手中的利器。

9.3.1 从实验开始

任何知识的学习都来源于实践，因此本节的最开头笔者将会讲述如何快速配置一个实验环境，来极快地撰写测试代码。

你需要的是利用虚幻引擎的Automation System，也就是自动化测试系统。具体的方式其实很简单。你需要在你的模块文件夹下建立一个Private文件夹，然后放入一个Test文件夹。在这个文件夹中，放置一个以Private文件夹中已有cpp的名字+Test.cpp结尾的文件。比如以下这样：



然后打开你的“模块名Test.cpp”，加入以下内容：

```
#include "当前模块对应的PrivatePCH.h"
#include "..."/包含其他需要的头文件
DEFINE_LOG_CATEGORY_STATIC(TestLog, Log, All);
IMPLEMENT_SIMPLE_AUTOMATION_TEST(FMultiThreadTest, "MyTest.Public
    .MultiThreadTest", EAutomationTestFlags::EditorContext | 
    EAutomationTestFlags::EngineFilter)
bool FMultiThreadTest::RunTest(const FString& Parameters)
{
    UE_LOG(TestLog, Log, TEXT("Hello"));
    return true;
}
```

这些代码看上去特别混乱，不管怎么说，先复制进去，然后笔者再解释好了。

随后正常编译并打开编辑器，请打开Frontend，中文版翻译为“会话窗口”，如图9-1所示。

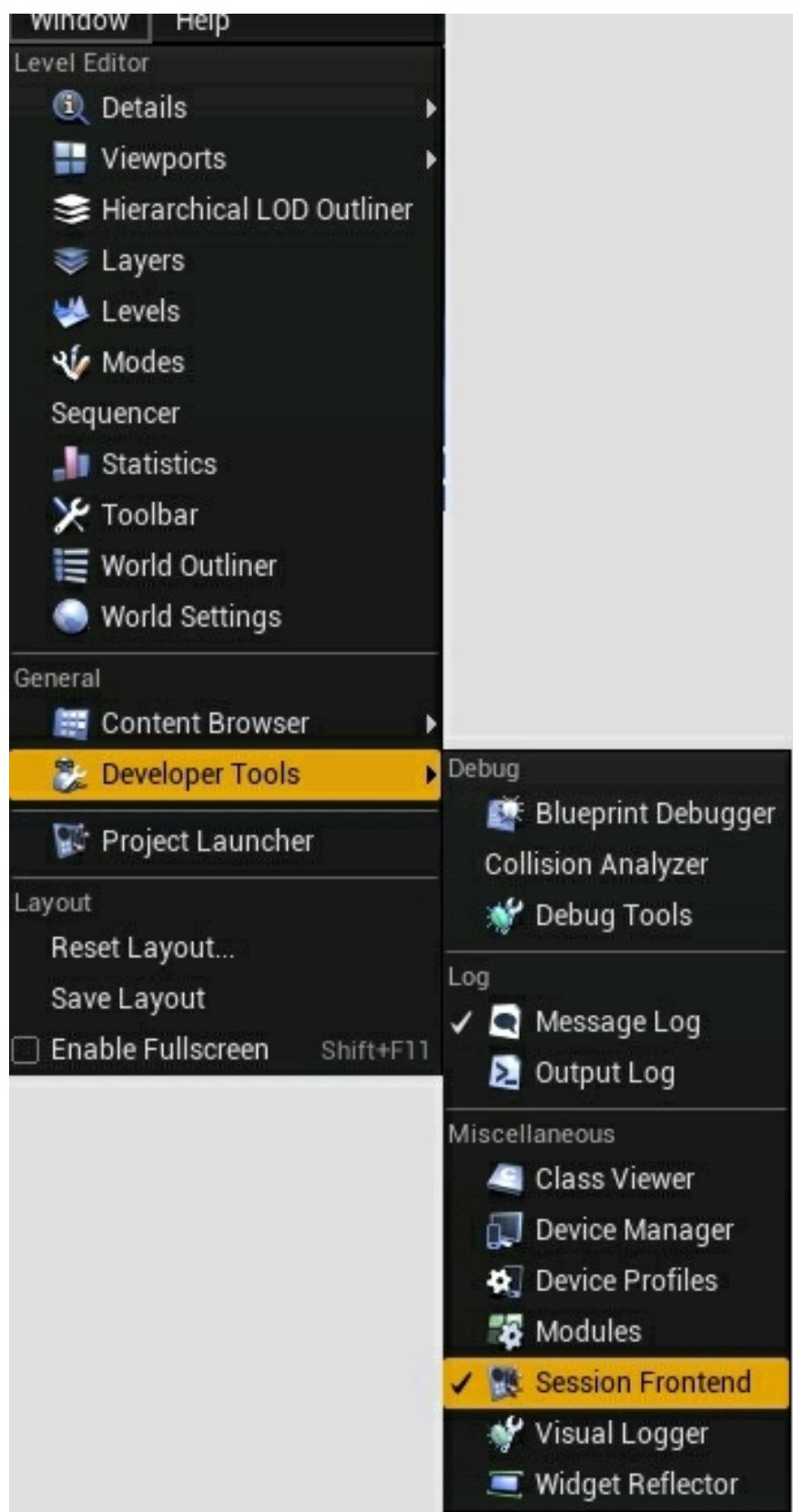


图9-1 打开会话窗口

然后进入以下页面，如图9-2所示。

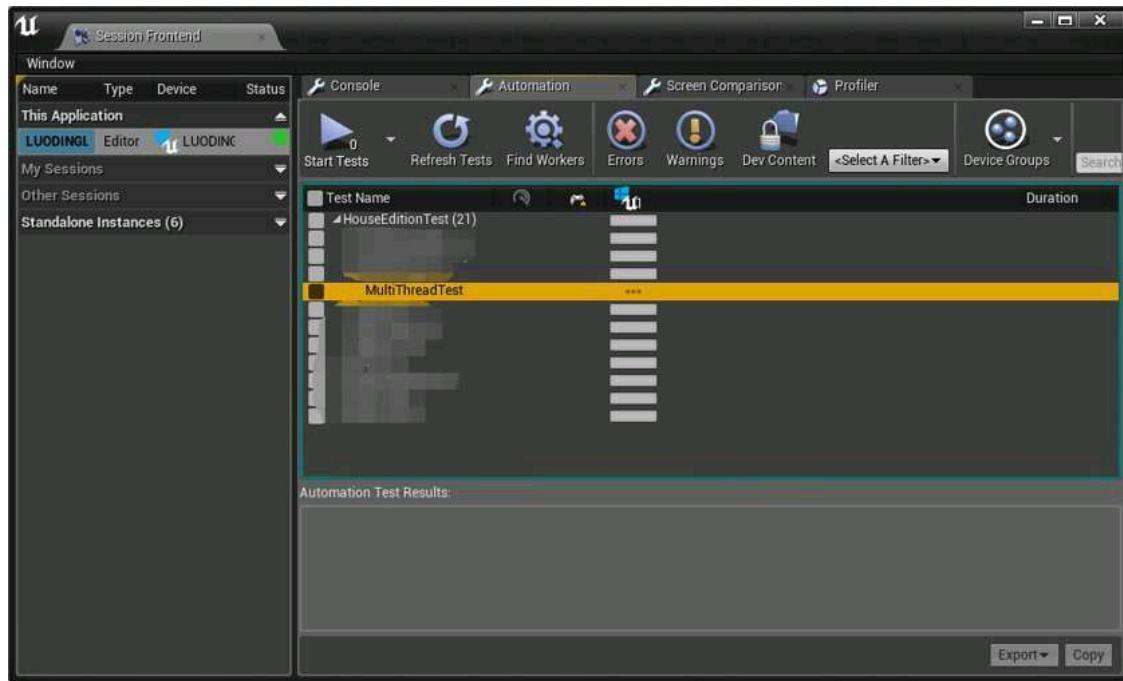


图9-2 会话窗口界面

勾选前面的白框然后点击Start Tests。你会在Message Output窗口看到这样一行输出，如图9-3所示。

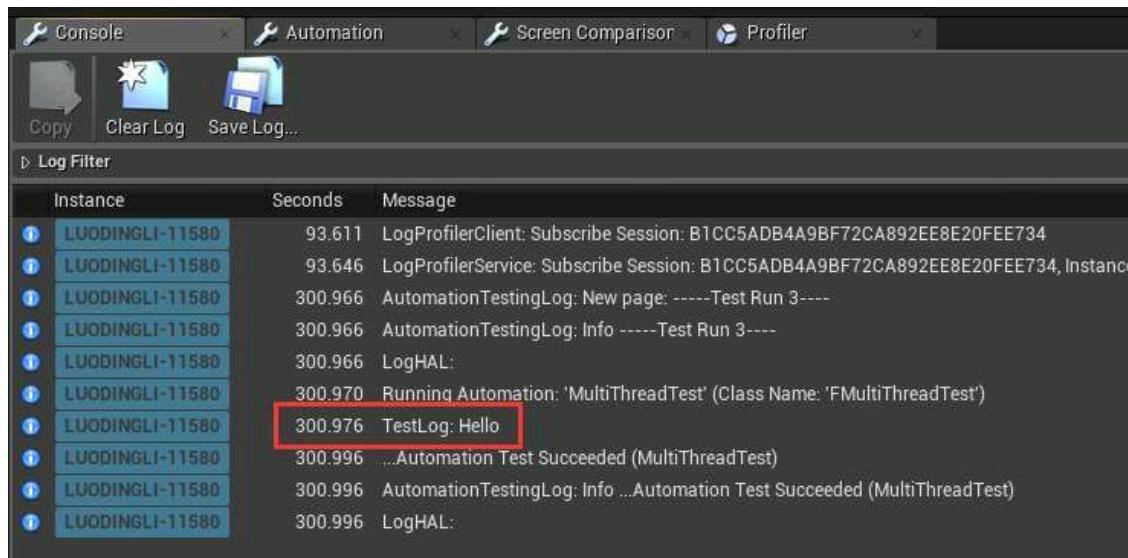


图9-3 能够正常看到输出的Log

恭喜，以后你就可以用这样的方式快速测试代码了。

9.3.2 线程

从实例开始

在虚幻引擎中，最接近于我们传统认为的“线程”这个概念的，就是FRunnable对象。FRunnable也有自己的子类，可以实现更多的功能。不过如果只是为了演示多线程的话，继承自FRunnable就足够了。那么首先请先感受一下用法，实例代码如下：

```
class FRunnableTestThread :public FRunnable {
public:
    FRunnableTestThread(int16 _Index) :Index(_Index) {}
    virtual bool Init() override
    {
        UE_LOG(TestLog, Log, TEXT("Thread %d Init"), Index);
        return true;
    }
    virtual uint32 Run() override
    {
        UE_LOG(TestLog, Log, TEXT("Thread %d Run:1"), Index);
        Sleep(10.0f);
    }
}
```

```
        UE_LOG(TestLog, Log, TEXT("Thread %d Run:2"), Index);
        Sleep(10.0f);
        UE_LOG(TestLog, Log, TEXT("Thread %d Run:3"), Index);
        Sleep(10.0f);
    return 0;
}

virtual void Exit() override
{
    UE_LOG(TestLog, Log, TEXT("Thread %d Exit"), Index);
}

private:
    int16 Index;
};

IMPLEMENT_SIMPLE_AUTOMATION_TEST(FFRunnableTest , "MyTest.Public"
    RunnableTest", EAutomationTestFlags::EditorContext | 
    EAutomationTestFlags::EngineFilter)

bool FFRunnableTest::RunTest(const FString& Parameters)
{
    FRunnableThread::Create(new FRunnableTestThread(0), TEXT("Test Thread 0"));
    FRunnableThread::Create(new FRunnableTestThread(1), TEXT("Test Thread 1"));
    FRunnableThread::Create(new FRunnableTestThread(2), TEXT("Test Thread 2"));
    return true;
}
```

如果你成功运行的话，结果应该是这样，如图9-4所示。

```
59.072 LogInfo:
59.079 Running Automation: 'RunnableTest' (Class Name: 'FFRunnableTest')
59.115 TestLog: Thread 0 Init
59.115 TestLog: Thread 0 Run:1
59.115 TestLog: Thread 0 Run:2
59.115 TestLog: Thread 1 Init
59.115 TestLog: Thread 1 Run:1
59.115 TestLog: Thread 0 Run:3
59.115 TestLog: Thread 1 Run:2
59.115 TestLog: Thread 2 Init
59.115 TestLog: Thread 2 Run:1
59.115 TestLog: Thread 1 Run:3
59.115 TestLog: Thread 0 Exit
59.115 TestLog: Thread 2 Run:2
59.128 TestLog: Thread 1 Exit
59.128 TestLog: Thread 2 Run:3
59.129 ...Automation Test Succeeded (RunnableTest)
```

图9-4 多个线程按照异步的方式打印内容

可以看到，所有的线程异步启动，并且非同步执行。这说明我们确实创建了三个线程，并运行它们。

如果你实际创建线程，并且在Run函数中间下断点，你能够在Visual Studio的线程调试窗口看到你所创建的线程名称，如图9-5所示。

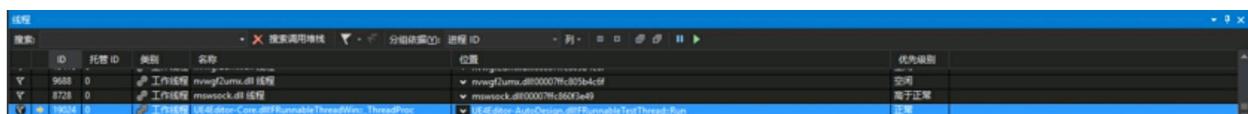


图9-5 刚刚创建的线程

分析

那么下面就可以来看，我们这段代码究竟完成了什么。

首先我们声明了一个继承自FRunnable的类，并实现了三个函数：Init、Run和Exit。Init函数内放置初始化代码，Run函数放置运行的代码，Exit函数用于在线程退出的时候做清理工作。这些就是创建一个线程所需要完成的。事实上，你需要做的其实非常少（当然无法达到C++11那种极为简单的地步）。

那么在生成一个线程对象之后，接下来需要的是“启动这个线程”。方法就是借助FRunnableThread的“Create”方法。第一个参数传入一个FRunnable对象，第二个参数则是线程的名字。

9.3.3 TaskGraph系统

虚幻引擎的另一个多线程系统则更加的现代，是基于Task思想，对线程进行复用从而实现的系统。频繁创建和销毁线程的代价是很大的，但是很多时候我们创建的线程都是临时的，不会伴随我们的软件的整个生命周期。而为了达到并行化的目的，我们可以把需要执行的“指令”和“数据”封成一个包，然后交给Task Graph，当Task Graph对应的线程有空闲，就会取出其中的Task执行。

感谢虚幻引擎论坛的Rama先生，在虚幻引擎早期就对Task Graph系统给出了相当清晰的描述。

从实践开始

同样地，让我们先来看一段示例性的代码，并看看运行结果：

```
class FShowcaseTask {

public:

    FShowcaseTask(int16 _Index) :Index(_Index) {}

    static const TCHAR* GetTaskName()

    {

        return TEXT("FShowcaseTask");

    }

    FORCEINLINE static TStatId GetStatId()

    {

        RETURN_QUICK_DECLARE_CYCLE_STAT(FShowcaseTask ,

            STATGROUP_TaskGraphTasks);

    }

    static ENamedThreads::Type GetDesiredThread()

    {

        return ENamedThreads::AnyThread;

    }

    static ESubsequentsMode::Type GetSubsequentsMode()

    {

        return ESubsequentsMode::TrackSubsequents;

    }

    void DoTask(ENamedThreads::Type CurrentThread , const FGraphE
& MyCompletionGraphEvent)

    {

        UE_LOG(TestLog, Log, TEXT("Thread %d Run:1"), Index);

        Sleep(10.0f);

        UE_LOG(TestLog, Log, TEXT("Thread %d Run:2"), Index);

        Sleep(10.0f);

    }

}
```

```

        UE_LOG(TestLog, Log, TEXT("Thread %d Run:3"), Index);

    Sleep(10.0f);

}

private:

    int16 Index;

};

IMPLEMENT_SIMPLE_AUTOMATION_TEST(FMyTaskGraphTest , "MyTest.Publi
    .TaskGraphTest", EAutomationTestFlags::EditorContext |

    EAutomationTestFlags::EngineFilter)

bool FMyTaskGraphTest::RunTest(const FString& Parameters)
{
    TGraphTask<FShowcaseTask >::CreateTask(NULL, ENamedThreads::
    TGraphTask<FShowcaseTask >::CreateTask(NULL, ENamedThreads::
    TGraphTask<FShowcaseTask >::CreateTask(NULL, ENamedThreads::
    return true;
}

```

如果一切没有问题，你会得到这样的结果，如图9-6所示。

```

① LUODINGLI-14372 28.910 Running Automation: 'TaskGraphTest' (Class Name: 'FMyTaskGraphTest')
① LUODINGLI-14372 28.910 TestLog: Thread 1 Run:1
① LUODINGLI-14372 28.910 TestLog: Thread 0 Run:1
① LUODINGLI-14372 28.948 TestLog: Thread 2 Run:1
① LUODINGLI-14372 28.948 TestLog: Thread 2 Run:2
① LUODINGLI-14372 28.948 TestLog: Thread 0 Run:2
① LUODINGLI-14372 28.948 TestLog: Thread 1 Run:2
① LUODINGLI-14372 28.948 TestLog: Thread 2 Run:3
① LUODINGLI-14372 28.948 TestLog: Thread 1 Run:3
① LUODINGLI-14372 28.948 TestLog: Thread 0 Run:3
① LUODINGLI-14372 28.952 ...Automation Test Succeeded (TaskGraphTest)

```

图9-6 Caption

分析：首先需要明白的是，Task Graph由于采用的是模板匹配，因此并不需要每个Task继承自一个指定的类，而是只要具有指定的几个函数，就能够让模板编译通过。这些函数就是：

- GetTaskName:静态函数，返回当前Task的名字。
- GetStatId:静态函数，返回当前Task的ID记录类型，你可以借助RETURN_QUICK_DECLARE_CYCLE_STAT宏快速定义一个并返回。
- GetDesiredThread:可以指定这个Task是在哪个线程执行，关于可选项有哪些，可以查看ENamedThreads::Type。
- GetSubsequentsMode:这是Task Graph用来进行依赖检查的前置标记，可选包括Track Subsequents和Fire And Forget两个选项。前者表示这个Task有可能是某个其他Task的前置条件，所以Task Graph系统会反复检查这个Task有没有执行完成。后者表示一旦开始就不用去管了，直到执行完毕。也就不能作为其他Task的前置条件。
- DoTask:最重要的函数，里面是这个Task的执行代码。

而启动一个Task的方式是这样的：

```
TGraphTask::CreateTask(NULL, ENamedThreads::GameThread).  
ConstructAndDispatchWhenReady(0);
```

其实需要注意的是，这是一个连续函数调用：

CreateTask的结果被调用了ConstructAndDispatchWhenReady。而给Task传递的参数是在这个时候给出的。这里的0就是构造函数里面的ID。这实质上是一种“延迟构造”的设计，因为当前的Task有可能依赖于

其他的Task，因此构造函数的调用会延迟到很久之后。

如果你下断点查看的话，能够发现，执行代码没有运行在一个单独的线程之中，而是运行在已有的线程TaskGraphThread2中，如图9-7所示。

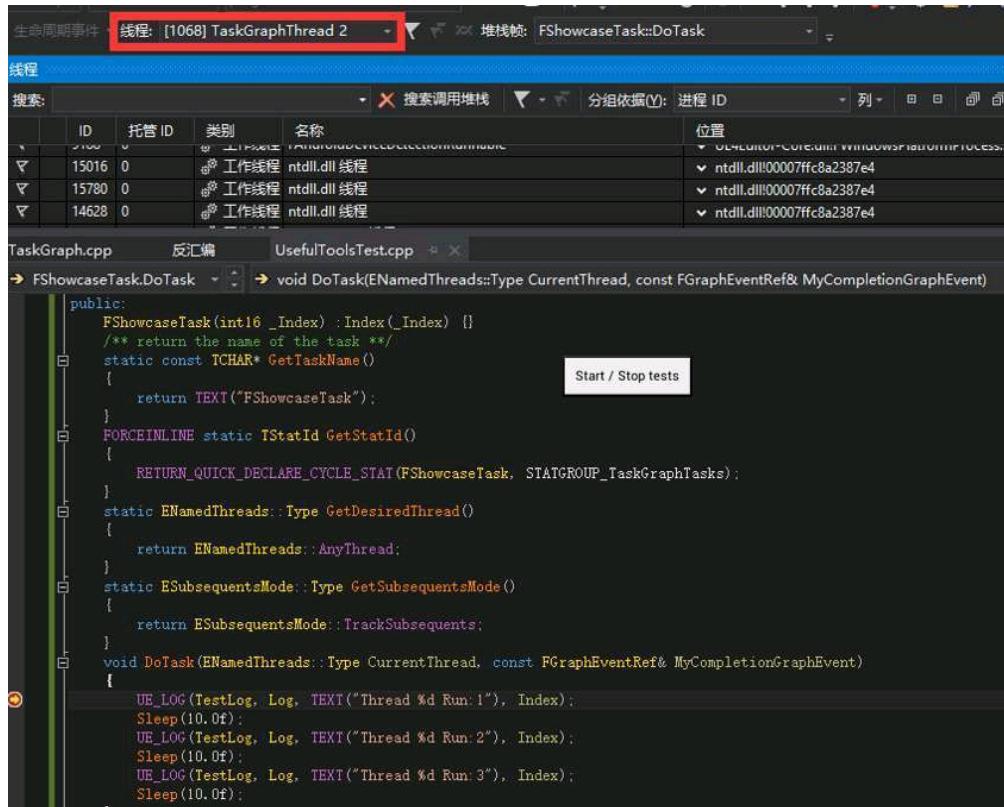


图9-7 代码在TaskGraphThread2中执行，而不是在新的线程中执行

9.3.4 Std::Threa

作为C++11的特性之一，笔者其实是相当喜欢Std::Thread的。而且写出来的代码也是最为简单、凝练且带有一种C++特有的气息。

从实践开始

这是目前最简单的多线程代码，为了运行，请在开头包含：

```
IMPLEMENT_SIMPLE_AUTOMATION_TEST(FSTDThreadTest, "MyTest.PublicTe
    STDThreadTest", EAutomationTestFlags::EditorContext | 
    EAutomationTestFlags::EngineFilter)

bool FSTDThreadTest::RunTest(const FString& Parameters)
{
    std::function<void(int16)> TaskFunction=
    [=](int16 Index) {
        UE_LOG(TestLog, Log, TEXT("Thread %d Run:1"), Index);
        Sleep(10.0f);

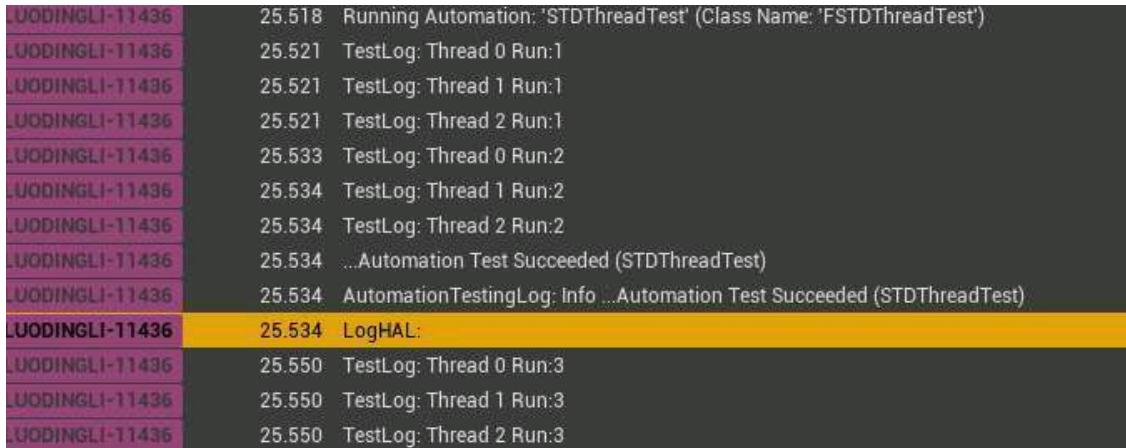
        UE_LOG(TestLog, Log, TEXT("Thread %d Run:2"), Index);
        Sleep(10.0f);

        UE_LOG(TestLog, Log, TEXT("Thread %d Run:3"), Index);
        Sleep(10.0f);

    };
    new std::thread(TaskFunction, 0);
    new std::thread(TaskFunction, 1);
    new std::thread(TaskFunction, 2);
    return true;
}
```

最终的输出结果如图9-8所示。

可以看到输出的结果也是没有问题的。



The screenshot shows a terminal window with several lines of log output. The log entries are color-coded: purple for file names, black for line numbers, and green for log messages. The log messages indicate the execution of a test automation script across three threads, with successful completion and logging of the HAL.

LUODINGLI-11436	25.518	Running Automation: 'STDThreadTest' (Class Name: 'FSTDThreadTest')
LUODINGLI-11436	25.521	TestLog: Thread 0 Run:1
LUODINGLI-11436	25.521	TestLog: Thread 1 Run:1
LUODINGLI-11436	25.521	TestLog: Thread 2 Run:1
LUODINGLI-11436	25.533	TestLog: Thread 0 Run:2
LUODINGLI-11436	25.534	TestLog: Thread 1 Run:2
LUODINGLI-11436	25.534	TestLog: Thread 2 Run:2
LUODINGLI-11436	25.534	...Automation Test Succeeded (STDThreadTest)
LUODINGLI-11436	25.534	AutomationTestingLog: Info ...Automation Test Succeeded (STDThreadTest)
LUODINGLI-11436	25.534	LogHAL:
LUODINGLI-11436	25.550	TestLog: Thread 0 Run:3
LUODINGLI-11436	25.550	TestLog: Thread 1 Run:3
LUODINGLI-11436	25.550	TestLog: Thread 2 Run:3

图9-8 使用标准库提供的多线程函数

分析

我们可以把这段代码分为两段，前半段是定义“指令”，实质上是通过Lambda表达式，定义了一个“可执行对象”。这就是functional这个头文件的用处。其统一了普通的函数，通过重载“()”操作符实现的“函数对象”，以及由Lambda表达式定义的匿名函数。

后半段则是定义Thread，其直接New出Thread对象，然后传递前面规定的指令，以及后边的参数。标准库的线程对象一旦被实例化，则立刻就会开始执行。

9.3.5 线程同步

我们已经能够创建线程来并发执行任务，那么接下来必须要对我们创建的线程进行一定程度上的控制，否则我们的线程就会像脱缰的野马

一样狂飙：我们不知道它们什么时候已经完成了任务。因此引入线程的同步非常有必要。虚幻引擎吸纳了大量的线程同步思想，操作系统与C++11标准都有所吸纳。总体而言，虚幻引擎的线程同步系统是分层级的。

最底层的包括：FCriticalSection临界区，这是实现线程同步的基石。其有一对Lock、Unlock操作原语，当一个线程将一个临界区Lock的时候，其他线程试图Lock就会阻塞，直到这个线程Unlock或者超时为止。而虚幻引擎的Mutex实现也是依赖于临界区的。这也是最符合传统多线程编程思想的多线程同步方式。但是代价就是有可能某一个线程需要轮询临界区，而且很多功能非常不方便实现（例如一个从子线程获取返回值，都必须要写一个忙等死循环，或者写成一个在Tick中不断查询的代码）。

而虚幻引擎也提供了更先进的线程同步方式，包括：

1. 投递Task到另一个线程的TaskGraph中，这样另一个线程就会去执行这个Task，给人的感觉就像是“一个线程通知了另一个线程执行某一段代码”，很符合人们的思维直觉。
2. 使用TFuture、TPromise。这两个应该是来自于C++11标准和Boost标准库的思想，C++11也提供了std::future和std::promise。

这个思想对于笔者来说也是非常新鲜的。关于这两者的介绍，笔者将会在后文中详细叙说，这里只给出一个形象的例子：一个年轻的小伙子A，一个他的女神B。

小伙子这会儿正一无所有，但是他有志向。于是他向他的女神“承诺（Promise）”自己将会为女神买一辆宝马车。

于是女神调用了他的承诺的GetFuture函数，获得了一个“未来”值，然后和他在一起相处了起来。而女神也相信他，愿意给他一段时间。

经过了一段时间之后，假如这个小伙子足够努力，于是他真的买了一辆宝马车，通过Promise对象的SetValue函数把宝马车放到了Promise对象中，而有一天，当他的女神忽然想起，自己需要出门撑撑门面（或者什么其他理由都行），女神想起了她男朋友的承诺。于是她调用了“未来”的Get()来获取当年承诺的内容。果然她真的得到了一辆宝马车。于是她开心地出门了。

请回想一下之前的那个“获取子线程的计算结果”的案例，会发现这样一对泛型是非常方便的。尤其是你可以借助IsValid之类的函数查看你的宝马车是否到账的情况下，这样的编程模型几乎是大大简化了传统异步程序的设计。

在这一套高级同步系统的基础上，还有一个更简练的方案，就是虚幻引擎提供的Async模板。其提供了一段示例代码让笔者非常惊艳：

```
auto Result = Async() EAsyncExecution::Thread, {
    return 123;
}
```

返回值为一个TFuture对象。需要注意的是，Async一定会在另一个非GameThread线程执行请求的任务，因此不能在任务中对UObject相关的类实例进行修改，否则会出现问题。

9.3.6 多进程

虚幻引擎同样提供了对进程的封装。在Core模块的GenericPlatformProcess中，可以看到FGenericPlatformProcess类的定义。其提供的CreateProc静态函数能够根据提供的URL启动一个进程，返回FProcHandle类型的进程句柄，并通过管道的形式进行数据交换。

[1] AppInit并不是一个很短的函数，其有一系列初始化过程，但这并不是非常核心，所以不再加以笔墨讨论。

第10章 对象模型

问题

我想知道虚幻引擎对象的生命周期！

第一部分介绍虚幻引擎C++编程的章节，初步介绍了虚幻引擎的对象系统。本章则深入到细节中，具体分析虚幻引擎的多个对象模型，详细介绍对象的整个生命周期。这里涉及的对象主要为：UObject对象、组件对象和Actor对象。

10.1 UObject对象

前情提要

在第一部分《对象》章节中，我们介绍了UObject对象的产生与销毁：

产生：UObject对象借助NewObject<T>函数产生，无法通过标准New操作符产生。

销毁：UObject对象由虚幻引擎垃圾回收系统管理，不需要手

动请求销毁。

10.1.1 来源

前文已经有所讲述，UObject对象的初始化不能直接采用New操作符完成。而是必须要借助NewObject模板函数完成。读者可参考UObject 初始化过程示意图，便可初步理解这个问题——因为对于UObject而言，对象的初始化被分为了两个阶段：内存分配阶段和对象构造阶段，如图10-1所示。

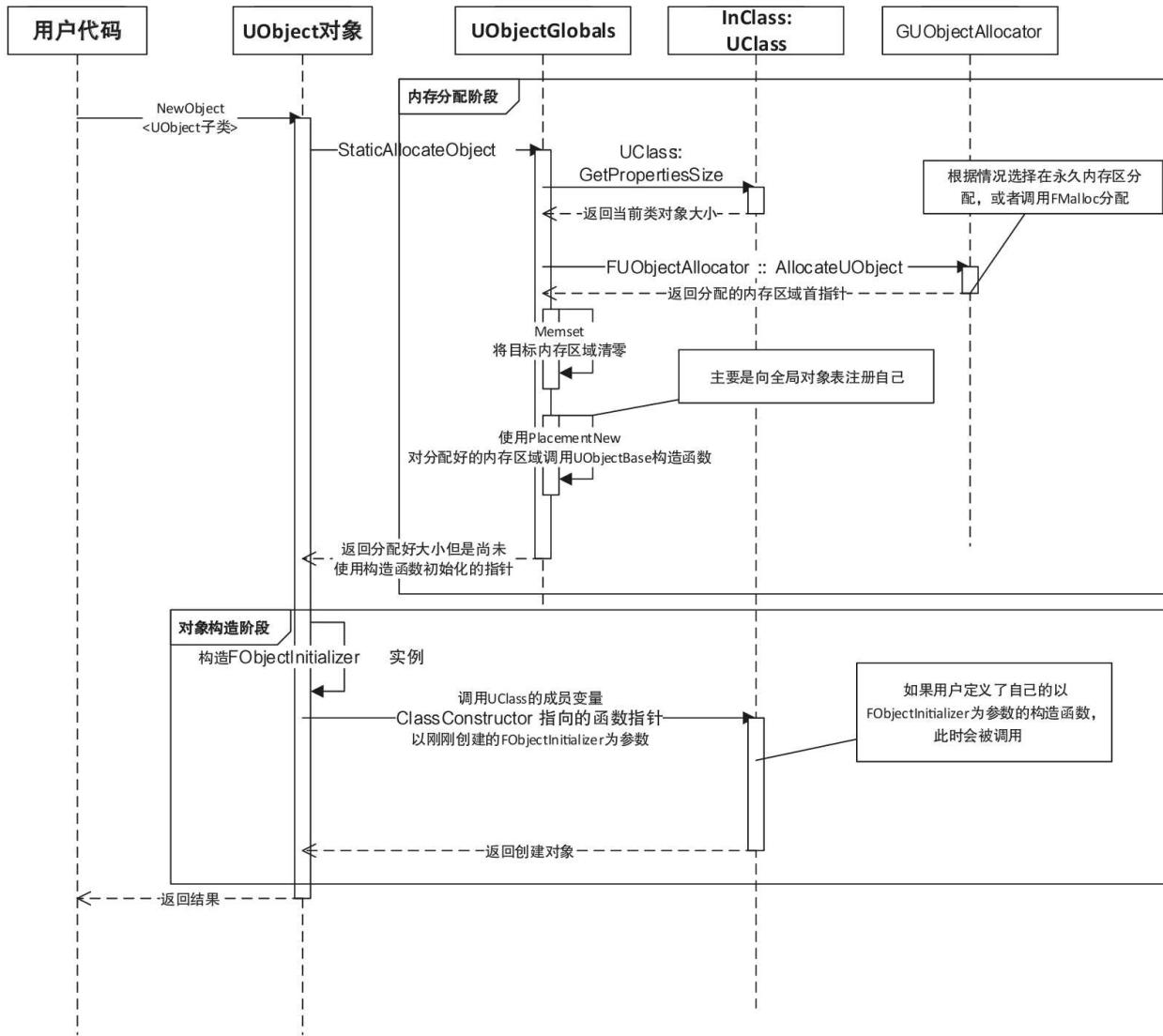


图10-1 UObject初始化

内存分配阶段 这个阶段虚幻引擎干涉了标准的内存分配过程。示意图只展示了“主流”情况，实际上有大量的判断来确保内存对齐、类默认对象不会重复创建等。但是总体来说，其获取当前 UObject 对象对应的 UClass 类的信息，根据类成员变量的总大小，加上内存对齐，然后在内存中分配一块合适的区域存放。除此之外也调用了 UObjectBase 原地进行构造。如果读者不大熟悉 PlacementNew，则可以理解为“不分配内存，直接假设当前内存指

向的区域为一个已经分配好并与当前对象等大的内存区域，然后调用构造函数”。这一过程相对来说比较好理解。

对象构造阶段 这个阶段则显得麻烦一些。有读者会提问为何不直接使用PlacementNew完成内存构造，偏偏要多此一举，用UClass的构造函数指针ClassConstructor完成对象构造？请见下文。

是的，为何需要一个专门的、以FObjectInitializer为参数的函数指针完成对象构造？在解答之前，请允许笔者先解释这个指针是在何处赋值的：通常来说，该指针在反射生成的.**generated.h** 中完成注册赋值。如果开发者自己实现了一个以FObjectInitializer为参数的构造函数，则直接将该函数指针指向一个静态函数**_DefaultConstructor**，定义如下：

```
static void __DefaultConstructor(const FObjectInitializer  
{ new((EInternal*)X.GetObj())TClass(X); }
```

也就是说，笔者所言的“构造函数指针”是不妥的。实际上，你也无法直接获得一个类的构造函数指针。这个指针指向的是一个静态函数。这个函数的工作就是，获取当前FObjectInitializer对象持有的、指向刚刚构造出来的UObject的指针，然后对其调用PlacementNew，传入FObjectInitializer作为参数调用构造函数，完成构造。如果开发者只是实现了一个无参数的构造函数，则直接调用无参数构造函数，完成构造。

此时读者肯定觉得，虚幻引擎设计者“智商掉线”，绕了半天，最后还是走回了原来的路子，有何意义？这是因为虚幻引擎希望获得ClassConstructor供复用。也就是说虚幻引擎希望能够获得某个类的构造函数，就像一个点金手一样，划出一片内存，然后点一下就会模塑出一

个类的对象。甚至在使用这个点金手的时候，不需要知道这个点金手所属的到底是哪个类。如果使用PlacementNew方案就地构造，就必然需要传入类型参数作为模板参数，这无法满足虚幻引擎的需求。

总而言之，虚幻引擎采用了两步构造的机制，先分配内存，做出一些额外处理后再模塑对象。这也是为何不能直接给构造函数传递参数的原因：因为调用的__DefaultConstructor只能采用一个参数。这个函数是预先定义好的，故无法什么参数都传递。

10.1.2 重生：序列化

序列化是指将一个对象变为更易保存的形式，写入到持久存储中。反序列化则反之，从持久存储中读取数据，然后还原原先的对象。通俗来说就是我们说的保存读取的过程。UObject的序列化和反序列化都对应函数Serialize。因为正向写入和反向读取需要按照同样的方式进行。

需要注意的是，序列化/反序列化的过程是一个“步骤”，而不同于某些语言或者引擎的实现是一个完整的对象初始化过程。意思就是，即使是通过反序列化的方式从持久存储中读出一个对象，也是需要先实例化对象，然后才反序列化，而非通过一段数据，直接就能通过反序列化获得对象。

所以我们主要分析反序列化，这是因为当理解对象如何读取出来之后，写入就显得很简单了。实例化一个对象之后，传递一个FArchive参数调用反序列化函数，接下来具体过程如下：

1. 通过GetClass函数获取当前的类信息，通过GetOuter函数获取Outer。这个Outer实际上指定了当前UObject会被作为哪一个对象的

子对象进行序列化。

2. 判断当前等待序列化的对象的类UClass的信息是否被载入，没有的话：
 - a. 预载入当前的类信息；
 - b. 预载入当前类的默认对象CDO的信息。
3. 载入名字。
4. 载入Outer。
5. 载入当前对象的类信息，保存于ObjClass对象中。
6. 载入对象的所有脚本成员变量信息。这一步必须在类信息加载后，否则无法根据类信息获得有哪些脚本成员变量需要加载。对应函数为SerializeScriptProperties。
 - a. 调用FArchive::MarkScriptSerializationStart函数，标记脚本数据序列化开始；
 - b. 调用ObjClass对象的SerializeTaggedProperties，载入脚本定义的成员变量；
 - c. 调用MarkScriptSerializationEnd标记脚本数据序列化结束。

这是反序列化的大致框架。在逐个解释之前，请允许笔者先解释以下两个概念：

切豆腐理论 对于硬盘上保存的数据来说，其本身不具备“意义”，其含义取决于我们如何解释 这一段数据。我们每次反序列化一个C++基本对象，例如一个float浮点数，我们是从豆腐最开头切下一块和浮点数长度一样大的豆腐，然后把这块豆腐解释 为一个浮点数。那读者会问，你怎么知道这块豆腐就是浮点数？不是布尔值？或者是某个字符串？答案是，你按照什么样的顺序把豆腐排起来的，并按照同样顺序切，那就能保证每次切出来的都是正确的。

我放了三块豆腐：豆腐1（float）、豆腐2（bool）、豆腐3（double）。然后把它们按顺序排好靠紧，于是豆腐之间的缝隙就没了。我可以把这三块豆腐看成一块完整的大豆腐，放冰箱里冻起来。下次要吃的时候，我该怎么把它们切成原样？很简单，我知道豆腐1、豆腐2、豆腐3的长度，所以我在1号长度处切一刀，1号+2号长度处切一刀。妥了！三块豆腐就分开了。

递归序列化/反序列化 一个类的成员变量会有以下类型：C++基本类型、自定义类型的对象、自定义类型的指针。关于指针问题，我们将会在后文分析。这里主要分析前两者。第一个，对于C++基本类型对象，我们可以用切豆腐理论序列化。那么自定义类型怎么办？毕竟这个类型是我自己定义的，我有些变量不想序列化（比如那些无关紧要的、可以由其他变量计算得来的变量）怎么解决？答案是，如果需要序列化自定义类型，就调用自定义类型的序列化函数。由该类型自行决定。于是这就转化为了一棵像树一样的序列化过程。沿用前文的切豆腐理论。当我们需要向豆腐列表增加一块由别人定义的豆腐的时候，我们遵循谁定义谁负责的原则，让别人把豆腐给我们。切豆腐的时候，由于我们只知道接下来要切的豆腐的类型，却不知道具体应该怎么切，我们就把整块豆腐都交给那个人，让他自己处理。切完了再还给我们。

理解这两个概念之后，我们就能开始分析虚幻引擎的反序列化过程。这其实是一个两步过程，其中第一步可选：

1. 从另一块豆腐中加载对象所属的类信息，一旦加载完成就像获得了一张当前豆腐的统计表，这块豆腐都有哪几节，每节对应类型是什么，都在这张表里。
2. 根据统计表，开始切豆腐。此时我们已经知道每块豆腐切割位置和

获取顺序了，还原成员变量简直如同探囊取物。

同时，虚幻引擎也对序列化后的大小进行了优化。我们不妨思考前文所述的切豆腐理论，如果我们完成序列化整个类，那么对于继承深度较深的子类，势必要序列化父类的全部数据。那么每个子类对象都必须占用较大的空间。有没有办法优化？我们会发现，其实子类对象很多数据是共同的，它们都是来自同样父类的默认值。这些数据只需要序列化一份就可以了。换句话说：

虚幻引擎序列化每个继承自UClass的类的默认值（即序列化CDO），然后序列化对象与类默认对象的差异。这样就节约了大量的子类对象序列化后的存储空间。

接下来需要解决的问题是，自定义类型的序列化。即，对于对象指针，如何进行序列化？直接序列化对象指针是完全不正确的：因为下一次加载时，内存地址不一定是一模一样的。那么将子对象完全序列化到自己体内呢？同样不可取，因为当两个对象同时引用同一个对象时，序列化会出现两份一模一样的对象。对于虚幻引擎来说，解决这个问题的任务交给了序列化类，而不是UObject类本身。典型的案例可以参考虚幻引擎UPackage的存储过程。

UPackage存储过程简介

这一段内容受到了黄河水 [\[1\]](#) 先生在群中讨论的启发，并取得了他的授权从而能够将他的想法收纳入本书中，对此笔者非常感谢。

UPackage序列化和反序列化过程中涉及的主要概念如下：

UPackage 显然涉及，不再赘述。

UObject 这里主要指UPackage中等待被序列化的对象。

UClass 等待被序列化对象对应的类。

UProperty 被UClass持有，存储这个类的属性（成员变量）信息。

FLinkerLoad 继承自FArchive，负责作为“豆腐”处理序列化和反序列化的过程。

CDO (Class Default Object) 类默认对象。一个类会在内存中放置一个默认对象，包含默认值。

UPackage需要序列化的对象其实是自身的元数据信息，以及包内的一系列对象，如图10-2所示。这些对象有可能互相引用，有可能引用其他包的对象，此时UPackage面临前文所述的问题。

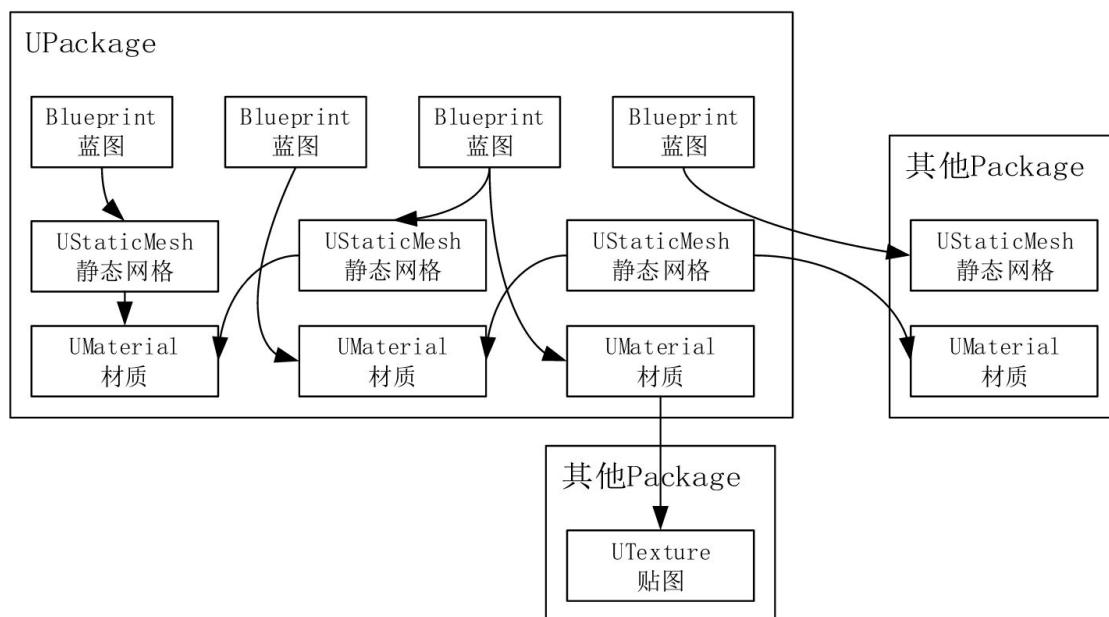


图10-2 UPackage内部对象的复杂引用关系：包含内部对象之间的互相引用，也包含其他包中的对象的引用

如何将这样的复杂引用过程转化为更适合存储和读取的形式？虚幻引擎设计了这样的机制来完成，如图10-3所示：

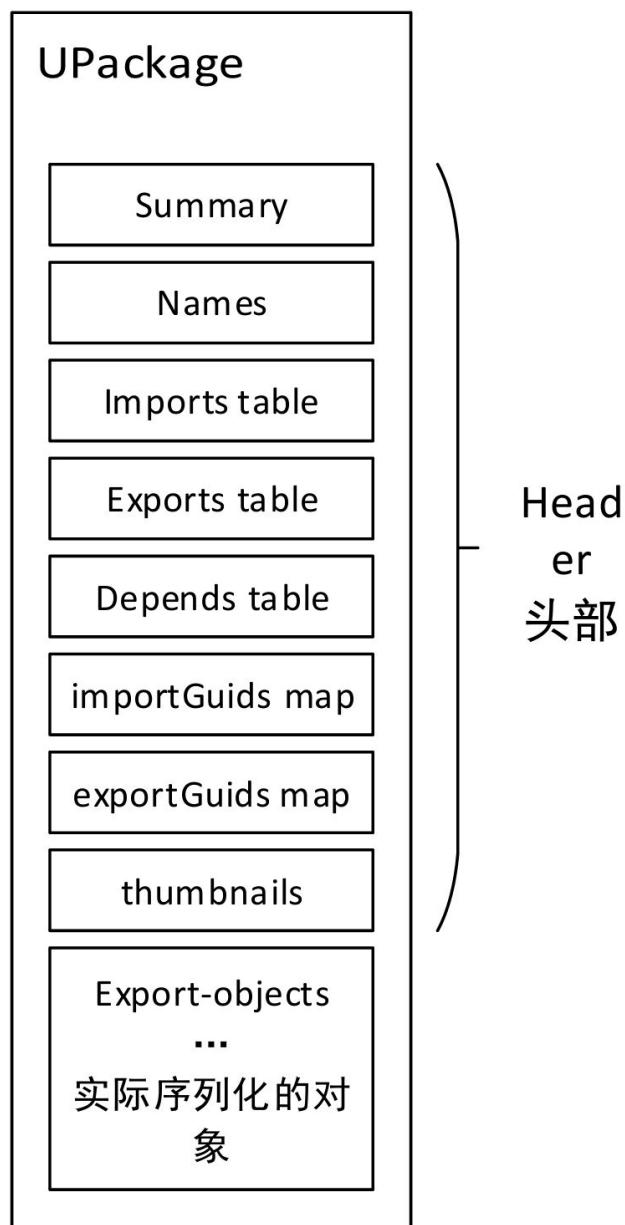
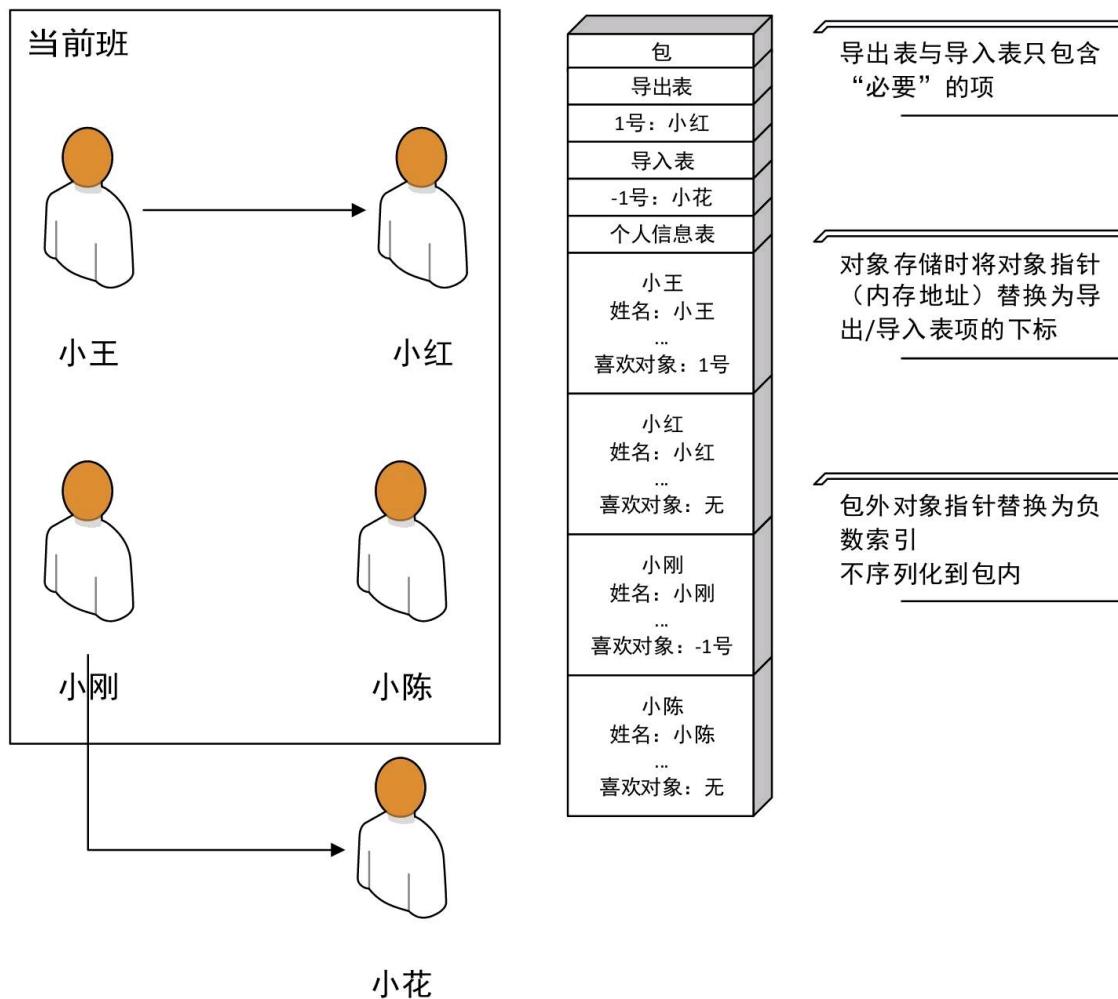


图10-3 UPackage的结构示意图，可以看到头部的导入表和导出表

为了方便描述，举一个班级的例子。这个班级如大家高中读书的时候一样，充满了早恋的味道（对象之间互相引用，还引用了其他班的对象，甚至还有多角恋），而且老师为了解决同桌早恋问题，经常让全班换座位（每次加载内存地址都会改变）。此时有个新来的学生——小明，希望能够记录全班谈恋爱对象配对名单，如图10-4所示：

序列化



1. 在包最前方有两张表，导出表Export Table和导入表Import Table，

前者可以理解为本班人员名单，后者可以理解为隔壁班人员名单。

2. 当序列化当前包内一个对象的时候，遇到一个UObject指针怎么办？此时肯定不能直接序列化指针的值。这类似于小明在记录小王喜欢的对象小红的时候，不能直接记录小王喜欢的人的座位，否则第二天座位一变动，就出事了。此时小明灵机一动：
 - a. 拿出导出表，往里面加了一项：1号-小红。
 - b. 修改“小王喜欢的对象”字段，将其从一个座位编号，变成了导出表里面的一个项的编号：1号。
 - c. 查看谁是小红真正的男朋友（NewObject的时候指定的Outer），到时候由他负责真正序列化小红。
 - d. 继续存储其他有关的信息：如果遇到普通数据（小王的名字，FName；小王的年龄，Int8），就直接序列化，如果遇到UObject，就重复第二和第三步。
3. 这时候小明发现小刚喜欢的对象居然是隔壁班的小花，小明无奈，只能再找出一张表：导入表，然后加了一项-1：小花（导入表项为负，导出表项为正，方便区分）。总不能把隔壁班的人也给序列化到本班的数据里面吧。然后把这项的编号替换到小刚喜欢的人的座位里。
4. 全部记录完毕，把两张表都保存起来，对象本身的数据则逐个排放到表后面，存放起来。

反序列化

需要注意的是，虚幻引擎的序列化、反序列化系统追求尽可能的高效，如图10-5所示，意味着不需要存储的数据绝不存储，其实质上是先

实例化对应类的对象，然后还原原始对象数据的过程。因此小明的故事变得恐怖了起来。

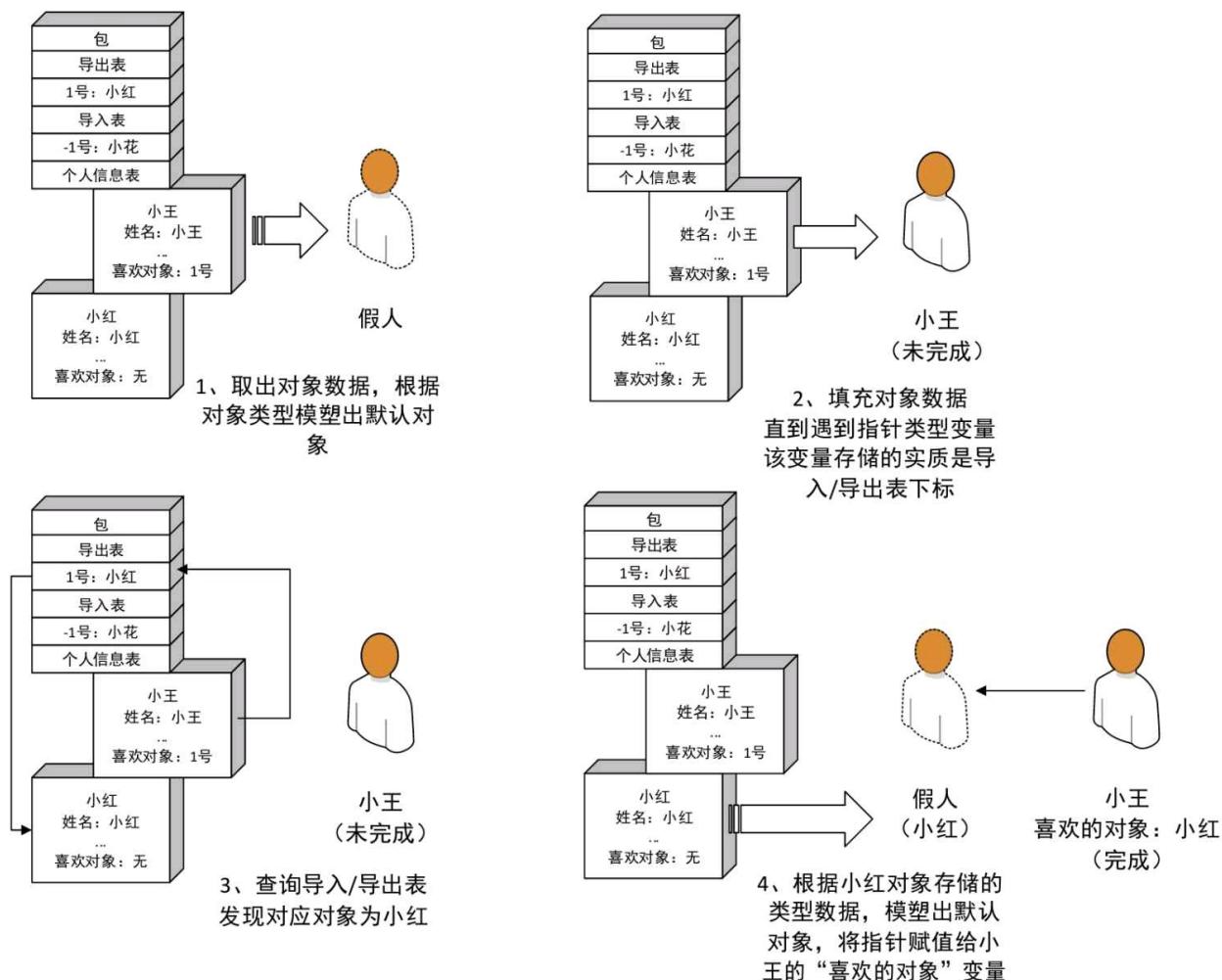


图10-5 反序列化：先模塑类默认对象，再填充保存的数据，遇到指针变量的情况就会查表

1. 小明大清早来到学校，走进教室一看，教室里空空如也，一个学生都没有（内存此时完全为空，没有任何对象信息）。完蛋了，老师马上要来上课了！想不到全班同学都翘课了！小明灵机一动，自己包里还有全班的数据呢！
2. 首先把表格全序列化出来，有了表格，其他都好说。
3. 从后面抽出一个对象，看看对象是什么类型，根据这个类型，把对象先模塑出来：

- a. 如果 UClass 类型数据还没载入，先把 UClass 载入了，并把 CDO 给读取了。
 - b. 根据 UClass 信息，模塑一个对象——说通俗点，小明在全班第一排第一桌直接给造了一个假人出来，这个假人现在还有什么特征都没有。
4. 根据这个对象的类信息，读取等大的数据，并根据类信息中包含的成员变量信息，判断这个成员变量的类型，执行以下步骤：
- a. 假如是基础对象（名字， FName；年龄， Int8），就直接把这个对象给反序列化。此时这个假人拥有了名字和年龄。可能还有杂七杂八的属性，比如身高、体重、瞳距之类的。
 - b. 此时不可避免地遇到了 UObject 类型的成员变量。糟糕，想不到这个小子居然还搞早恋！怎么办，这会儿全班还缺人呢。小明此时一拍脑袋，想起来做记录的时候，这里存储的是两张表的表项。小明赶紧看了看这个成员变量是正还是负：
为正 查 Export Table 导出表，看看这个对象有没有被序列化，如果有，就把对应对象的指针替换，否则就先造个假人丢在那里，等待此人的 Outer 负责实际序列化。相当于小明看了看，这表项有没有对应班上某个座位。没有的话，随便造个假人丢在某个空位置上，把座位号填到表项里，这样其他人如果把这个假人当对象，就会直接指定到对应的位置。
为负 查 ImportTable 导入表，看看对应的 Package 有没有载入内存，没载入，就通知校长把隔壁班给喊来上课；如果已经载入了，小明就跑到隔壁班，大吼一声：“小花！小花坐哪儿？”然后把获得的结果填入到表项里面。
这就相当于小明趴在正在还原的这个同学耳朵边上，指着班上某个正坐在座位上的假人说：你记住，坐在那儿的就是你喜欢的对象！或者：你记住，隔壁班第三排第二个就是你喜欢的对象！

欢的对象！

5. 经过这一波折腾，全班会经历这样一个过程：

- a. 首先班上会逐渐出现原来的同学和一些假人（已经被NewObject模型出来，但是还没有根据反序列化信息恢复成原始对象的对象）。
- b. 随后假人会逐渐被还原为原始对象。即随着读取的信息越来越多，根据反序列化后的信息还原成和原始对象一致的对象越来越多。
- c. 最后全班所有人都会被恢复为和原始对象一致的人。也许小王同学喜欢的那个人的座位号变了（反序列化后指针的值被修正），但是新的座位号上坐着的人，是和他当年喜欢的小红一模一样的人。

6. 最后当老师跨进教室，只会说：啊，今天你们换过座位啦！然后旁若无人地上课。

通俗地来说就是这样的过程。从这个过程中，我们能获取到一些非常有趣的信息和经验：

序列化必要的、差异性的数据 不必要的引用不需要被序列化和反序列化。因此如果你的成员变量没有被UPROPERTY标记，其不会被序列化。如果你的这个成员变量值与默认值一致，也不会占用空间进行序列化。

先模型对象，再还原数据 这个过程笔者多次重点阐述，就是为了强调虚幻引擎的这个设计。先把对象通过NewObject模型，然后还原差异性的数据。且被模型出的对象会作为其他对象修正指针指向的基础。正如前文所言，小明不会因为小王的对象小红还没被序列化就束手无策，没被序列化就直接实例化一个假人丢在那，大不

了以后读取到小红的数据时，把那个假人的信息改成和小红一样就好。

对象具有“所属”关系 由NewObject指定的Outer负责序列化和反序列化。

鸭子理论 叫起来像鸭子，看起来像鸭子，动起来像鸭子，那就是鸭子。说话像小红，看起来像小红，做事情像小红，那就是小红。也就是说，如果一个对象的所有成员变量与原始对象一致（指针的值可以不同，但指向的对象要一致），则该对象就是原始对象。

10.1.3 释放与消亡

销毁过程

UObject对象无法被手动释放，只能被手动请求“ConditionalBeginDestroy”来完成销毁。如函数名称所示，具体是否销毁、何时销毁，取决于虚幻引擎，而非取决于请求者。实质上来说BeginDestroy函数只是设置了当前UObject的RF_BeginDestroyed为真，然后通过SetLinker函数将当前对象从linker导出表中清除。

在时机成熟时，由FinishDestory函数完成UObject销毁操作。其首先销毁非C++的成员变量，对应函数DestroyNonNativeProperties。其核心是一个for循环，获取当前UClass类的析构函数链表，调用每个析构函数的DestroyValue_InContainer函数，以完成自身的销毁。

```
for (UProperty* P = GetClass()->DestructorLink; P; P = P->
    DestructorLinkNext)
{
    P->DestroyValue_InContainer(this);
}
```

触发销毁

前文已经叙述，客户代码只能请求销毁UObject而不能实际触发整个销毁过程。那么，什么样的情况会真正触发销毁呢？典型的情况是垃圾回收器来触发（StaticConstructUObject也会，但是不算典型情况）。

垃圾回收器实际上执行两个步骤：析构和回收。前者负责调用析构函数，通知对象进行析构操作。后者则负责回收当前UObject占用的内存。这一段代码可以在bageCollection.cpp中的IncrementalPurgeGarbage函数找到，如图10-6所示。

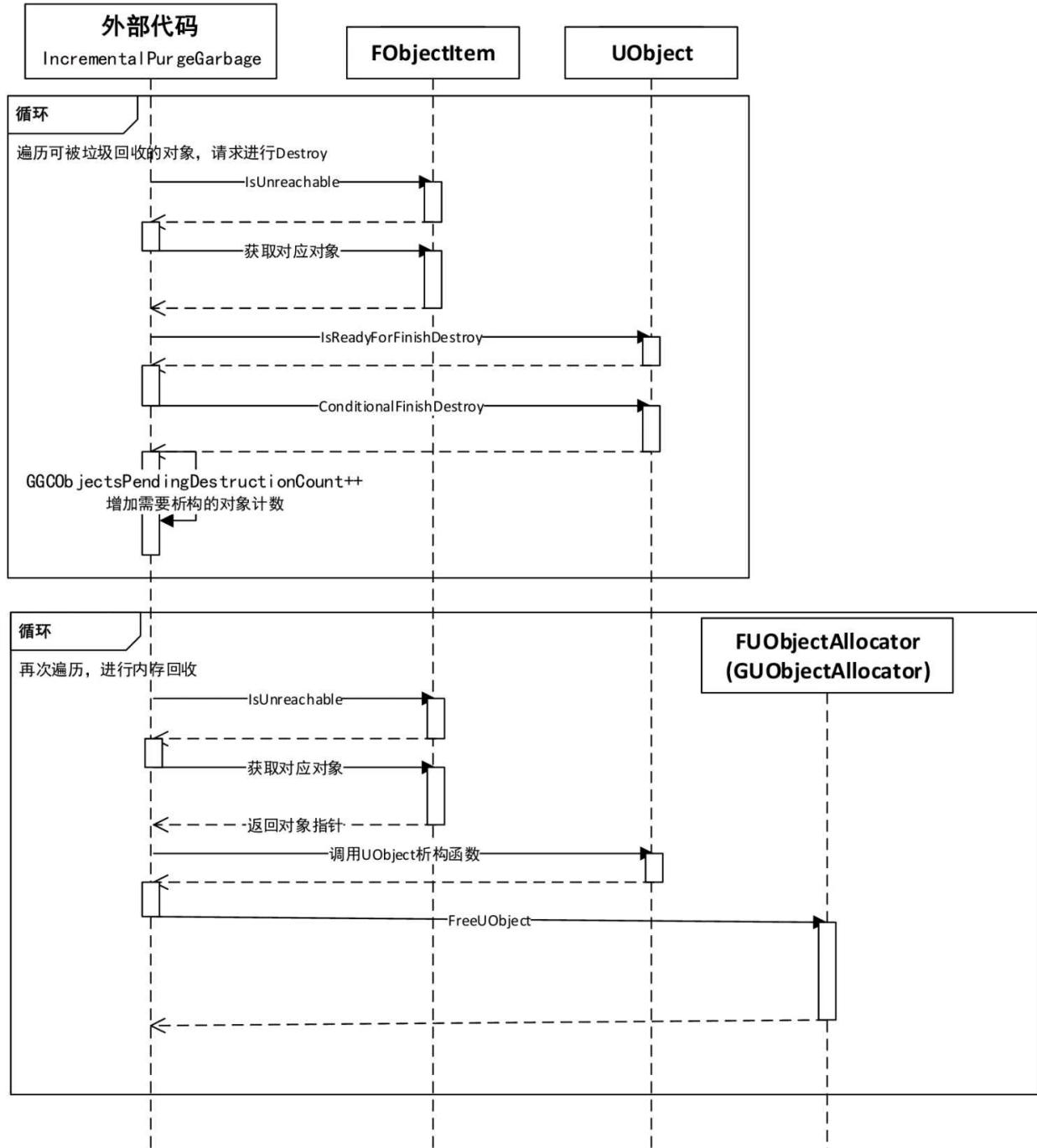


图10-6 UObject垃圾回收：分为两个遍历阶段，前阶段为通知各UClass父类进行析构阶段，后阶段为C++析构和内存回收阶段

10.1.4 垃圾回收

本质上说，垃圾回收（GC）应当作为UObject对象释放与消亡内容的一个子部分来进行阐述。但是由于垃圾回收部分涉及内容相当多，不适合作为一个小节来进行阐述，因此下面单独列出来对虚幻引擎的垃圾回收过程进行简单阐述。如果读者希望深入探索虚幻引擎的垃圾回收具体过程，并研究垃圾回收的源码实现，可以参考风恋残雪先生此文 [\[2\]](#)。

垃圾回收算法简介

关于垃圾回收算法的研究持续了很多年，也有许多书籍是集大成之作，如果读者第一次接触垃圾回收算法，推荐阅读《垃圾回收算法与实现》一书。

垃圾回收算法简单来说是解决这样一个问题：在一个合租房内公共区域里面的物品，每个人原则上会去收拾自己的垃圾（即原则上父对象负责回收子对象释放时占用的内存空间）。但是在丢掉某个东西的时候，不知道别人有没有在用，贸然丢弃就会引发问题。例如我认为这个属于我的电磁炉已经不需要了，我准备买个新的，于是随手丢进了垃圾堆。过了一会儿舍友从房间里面出来，端着锅准备煮面，发现电磁炉没了（在别的对象持有当前对象引用的情况下释放对象，导致野指针）。为了避免这个情况，合租房的每个人只好采取一个保守的策略：大家都不丢垃圾，于是屋子里面堆满了垃圾。

此时就必须采用一系列的方式来解决这样的问题，比较典型的算法有以下这些：

引用计数算法 给每个东西上面挂一个数字牌。我要用的时候就加1，不用了就减1。一旦最后一个不用这个东西的人发现，减1之后为0，于是知道这个东西没人需要用了，直接丢进垃圾堆。

优势 引用计数不需要暂停，是逐渐完成的。将垃圾回收的过程分配到运行的过程中。而不是一下子要求暂停所有人的生活，然后开始大扫除。

劣势

指针操作开销 每次使用都必须要调整牌子，频繁使用的物品翻来覆去地修改牌子数值是很大的一笔开销。

环形引用 假如是互相引用的对象，例如锅与锅盖配套，锅和锅盖互相引用，导致双方的牌子都是1。但是实际上大家都准备把锅和锅盖一起换掉，却总误认为有人还需要锅和锅盖。

标记-清扫算法 标记-清扫算法是追踪式GC的一种。追踪式引用计数算法会寻找整个对象引用网络，来寻找不需要垃圾回收的对象，这与引用计数式的“只关注单个对象”的思路刚好相反。首先假定所有东西都是垃圾，然后从每个舍友（在垃圾回收算法中称为根）开始搜寻，即让每个舍友指认哪些东西是他需要的，最终剩下的没有被指认过的东西，那就是大家都不需要的垃圾，因此就可以直接回收。

优势 没有环形引用问题，即使锅盖和锅互相引用，只要大家都不用这俩，都会被作为垃圾丢掉。

劣势

暂停 必须要有人大喊一声“大家先别动，我们先搞一次大扫除！”然后才能开始这个算法，算法结束之后，大家才能继续做手头的事情。这会导致系统具有明显延迟。

内存碎片 如果完全按照刚才所述的情况实现，即只丢掉垃圾而不整理，就会导致可用空间越来越细碎，最终导致大型对象无法被分配。

针对标准算法存在的问题，必然有诸多改进方案。虚幻引擎的智能指针系统采用的是引用计数算法，使用弱指针方案来（部分）解决环形引用问题[\[3\]](#)。

在讨论虚幻引擎的算法及其选择的原因之前，笔者先介绍一下GC的分类方式，以及虚幻引擎选择的理由，如表10-1所示。

表10-1 垃圾回收算法简单分类

分类	项目	描述
引用计数/追踪式 GC	引用计数	通过额外的计数来对单个对象的引用次数进行计算，当引用计数为零时回收对象
	追踪式	扫描系统对象引用网络，寻找被引用的对象，留下的对象即为垃圾
保守/精确	保守式	可以不使用额外信息、不修改现有框架。根据一些特性推断出可能为指针的内存区域，根据这些指针判断对应对象已被引用，从而释放那些“绝对不可能被引用”的对象。不求全部回收，但求不错删
	精确式	需要额外信息来进行辅助识别指针字段，但是能够精确认别每一个被引用的对象
搬迁式/非搬迁式	搬迁式	对象在 GC 的过程中在内存中的位置会进行移动
	非搬迁式	对象在 GC 的过程中在内存中的位置不会进行移动
实时/非实时	实时	实时 GC 不需要中断用户程序进行
	非实时	非实时 GC 会要求暂停来进行垃圾回收，在垃圾回收期间用户程序不能运行
渐进/非渐进	渐进	逐步完成搜索与回收
	非渐进	一口气完成搜索和回收操作

UObject的标记清扫算法

如今虚幻引擎开始选择自己的垃圾回收算法。抛开智能指针系统暂且不谈，笔者揣测虚幻引擎根据以下思路选择了自己的方案：

1. UClass包含了类的成员变量信息，类的成员变量也包含了“是否是指向对象的指针”的信息，因此具备选择精确式GC的客观条件。也就是复用反射系统，来完成对每一个被引用的对象的定位。所以虚幻引擎选择精确式**GC**。
2. 虚幻引擎已经使用智能指针系统管理非UObject对象。此时由于有

反射系统提供每一个对象互相引用的信息（想象一张庞大的UML实例图，虚幻引擎通过UObject、参考对应UClass，就能还原出这张庞大的实例图），从而能够实现对象引用网络。故采用追踪式GC。

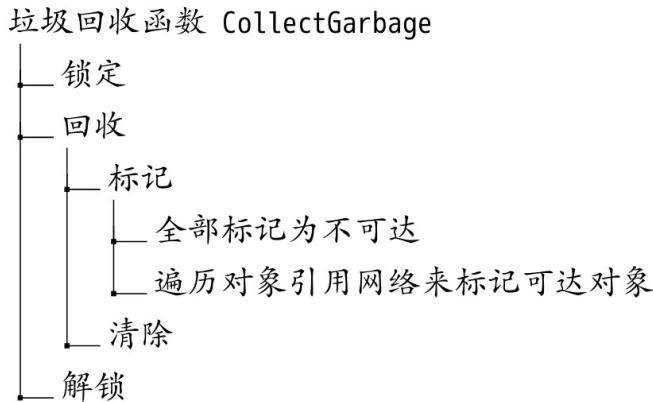
3. 虚幻引擎在回收的过程中没有搬迁对象，应当是考虑到对象搬迁过程中修正指针的庞大成本。
4. 虚幻引擎选择了一个非实时但是渐进式的垃圾回收算法。将垃圾回收的过程分步、并行化，以削弱选择追踪式GC带来的停等消耗。

虚幻引擎的垃圾回收在根源上是一个标记-清扫算法，笔者用一幅调用图来简单整理其思路，我们会逐步补全和细化这张图：



借助GGarbageCollectionGuardCritical.GCLock/GCUnlock函数，使得在垃圾回收期间，其他线程的任何UObject的操作都不会工作，从而避免出现一边回收一边操作导致的各种问题。

回收过程对应函数为FRealtimeGC::PerformReachabilityAnalysis，其可以看作两个步骤：标记和清除。标记过程为：全部标记为不可达，然后遍历对象并引用网络来标记可达对象。清除过程也很简单，直接检查标记，对没有被标记可达的对象调用ConditionalBeginDestroy函数来请求删除。



虚幻引擎使用了大量的技术来加速标记与回收过程，包括多线程和基于簇的方式，此处采用从简单到复杂的方式来阐述。

全部标记为不可达的算法很简单，虚幻引擎有个 `MarkObjectsAsUnreachable` 函数就是用来标记不可达的。借助 `FRawObjectIterator` 遍历所有的 Object，然后设置标记为 `Unreachable` 即可。

但是接下来这个步骤就比较麻烦，即遍历对象引用网络来标记可达对象。细分来说有两个问题：

1. 从哪里开始遍历？
2. 如何根据当前对象确定下一个遍历对象？

对于第一个问题来说，在之前标记不可达算法后，就把所有的“必然可达”对象都收集到了数组中。这些必然可达对象刚开始大部分是那些被添加到“根”的对象 [\[4\]](#)。这些对象是一定不会被垃圾回收的，相当于前面举例的“舍友”。在收集的过程中，可达的对象还不断会被添加到这个数组中。

接下来，会遍历这些必然可达对象，然后搜索这些必然可达对象能

够到达的其他对象，从而标记所有可达对象。这些必然可达对象就好像是渔网最上面的一条浮子，浮子下面的就是一大堆的互相引用的对象。

众所周知，网络，或者更准确一些，有向图的遍历，是非常复杂的过程。传统方案是采用深度优先遍历或者广度优先遍历。虚幻引擎面临的问题是去选择一套能够满足并行化需求的对象引用的遍历方案。

虚幻引擎采用的方案是：将二维的结构一维化来方便遍历。

这需要一个前端的配合，有点像是一个小型虚拟机的思路。每个 UClass 会生成一套以 FGCReferenceInfo 为成员的流 FGCReferenceTokenStream。这个流是紧凑排列的、类引用结构的信息。每一个 FGCReferenceInfo 结构会被存储为一个 uint32 的数据，这个微型结构包含以下字段：

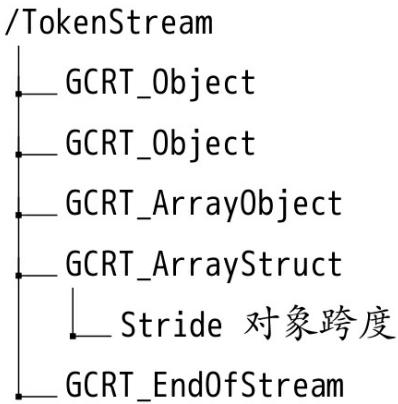
ReturnCount 返回深度计数，**8位** 用于统计当前需要返回的深度。如果是数组的最后一个元素，值为1；如果是数组中一个结构体中的数组的最后一个元素，值为2。以此类推。

Type 类型，**4位** 当前 Info 的类型。

Offse 偏移，**20位** 当前成员在对象中的内存偏移量。

加起来刚好32位，可以看作是虚拟机的一个字节码。

对每个 UClass，生成的是这样的一根像糖葫芦一样的 FGCReferenceInfo 数组，里面存储的其实是当前类对其他对象的引用结构。这个引用结构的一个例子如：



前端：这棵树的产生是通过遍历当前UClass的所有UProperty，对每个UProperty调用EmitReferenceInfo函数，产生一个一个的Token。对于每种类型的UProperty，由对应的UProperty子类（例如UObjectProperty）实现EmitReferenceInfo来产生对应Token。

后端：这棵树最后依然是被线性存储。形成一个一维线性结构，交给TFastReferenceCollector进行处理。这个Collector通过不断遍历这个线性结构，调用FGCReferenceProcessor来处理每一个引用。

FGCReferenceProcessor是具体设置每一个对象标记的类。其HandleObjectReference函数具体处理对象引用，首先会将当前引用目标对象的可达性设置为可达，然后放入前文所叙述的必然可达数组中。

这个设计的核心目的是为了加速：即由于线性结构遍历是非常容易进行并行化的，所以可以在合适的时机将垃圾回收任务并行化。遍历Token的时候，会根据实际情况，将前面提到的必然可达数组拆分为几个段落，对每个段落使用并行化的方式进行加速遍历。

而最终的清扫算法比标记算法的设计思路简单。虚幻引擎采用了增量清扫的算法，对应函数为IncrementalPurgeGarbage函数。清扫算法的过程在前文UObject的触发销毁章节进行了讲解。所谓的增量清扫，是

指虚幻引擎会考虑时间限制等，一段一段进行销毁的触发。需要注意的是，由于可能会在两次清扫时间之间产生新的UObject，故在每次进入清扫时，需要检查GObjCurrentPurgeObjectIndexNeedsReset，并根据实际情况，重新生成对象迭代器，避免迭代器失效。



基于簇的垃圾回收

虚幻引擎采用了基于簇的垃圾回收算法，用于加速cook后对象的回收。能够作为簇根的为UMaterial和UParticleSystem。当垃圾回收阶段检查到一个簇根不可达，整个簇都会被回收，加速回收操作的速度，节省了再去处理簇的子对象的时间。

10.2 Actor对象

前情提要

在第4章《对象》中，介绍了Actor对象的产生与销毁：

产生 Actor对象借助UWorld::SpawnActor<T>函数产生，需要获取一个UWorld*指针，才能产生Actor对象。

销毁 Actor对象可以借助Destory函数进行销毁，但是最终依然由虚幻引擎垃圾回收系统完成内存回收。

在虚幻引擎官方网站，你能很轻松地找到这样一幅图像 [\[5\]](#)，如图 10-7 所示。这幅图解释了虚幻引擎中最基础的对象 Actor 对象的生命周期。然而官方网站对 Actor 生命周期的介绍只是寥寥几笔带过，并没有详细深入。因此，下面将会详细探讨 Actor 对象的整体生命周期。

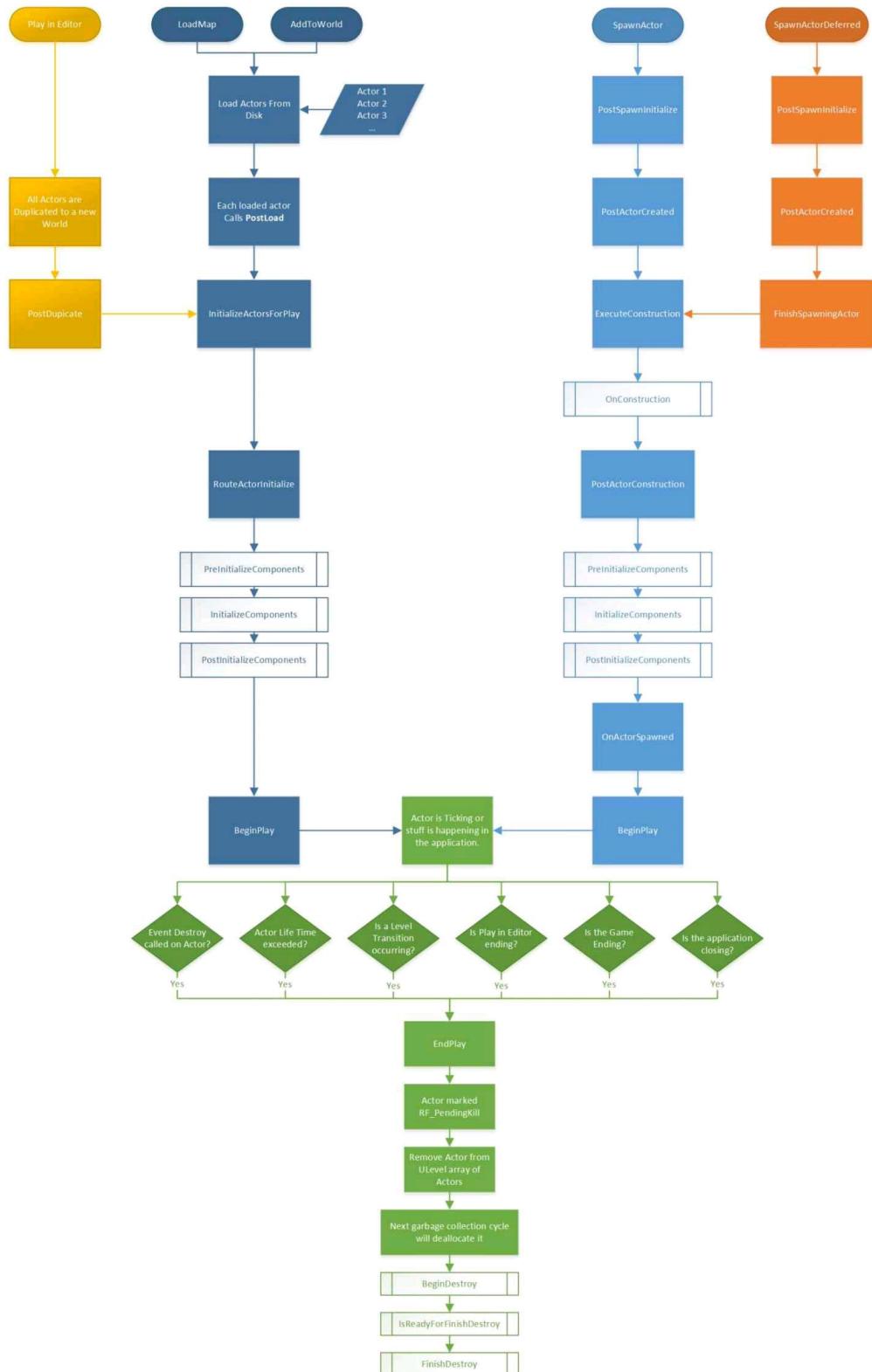


图10-7 Actor对象的生命周期，来自虚幻引擎官方网站文档

10.2.1 来源

对比UObject对象，Actor对象的来源是通过SpawnActor函数生成。最简单的一种函数原型为：

```
template< class T >
T* SpawnActor(
    const FActorSpawnParameters& SpawnParameters =
        FActorSpawnParameters())
{
    return CastChecked<T>(SpawnActor(T::StaticClass(),
        NULL, NULL, SpawnParameters), ECastCheckedType::
        NullAllowed);
}
```

SpawnActor只能够从UWorld调用，Actor也只会从属于World（演员必将依附所处的世界而存在）。换而言之，你无法直接通过NewObject来生成出一个Actor，但是AActor作为继承自UObject的类，其必然经历NewObject的构造过程，这个过程是在UWorld::SpawnActor中调用。在NewObject前后，虚幻引擎均进行了一系列的操作，从而保证Actor的正确构造。

构造**Actor**之前的处理 主要是三个内容：检查、获取Actor的模板Template，根据ESpawnActorCollisionHandlingMethod确定是否生成。

检查 借助HasAnyClassFlags函数，通过Flag检查：当前类是

否被废弃，当前类是否是抽象类，当前类是否真的继承自Actor，传递的Actor的模板是否和当前Actor类型对应，是否在构造函数中创建Actor等。

模板 此模板不是C++的那个模板，这个模板是指Actor的模型基础。虚幻引擎允许原型式模型Actor对象，即从一个已有对象中模型出一个新的模一样的对象，而非从类中模型。如果没有提供模板，当前Actor所属类的CDO将会作为模板。

碰撞处理 使用蓝图生成Actor的读者都应该知道，其需要提供一个ESpawnActorCollisionHandlingMethod枚举值，用来指定当生成的Actor位置存在碰撞（即俗称“挤到了”）的情况应该如何处理。这里就会根据提供的值进行处理：

SpawnParameters.bNoFail 特殊标记值，该标记放于SpawnParameters中，如果设置为真，则忽略所有的碰撞处理，强制生成。

DontSpawnIfColliding 在指定位置检测是否有几何碰撞，如果有就放弃生成对象。

其他的处理方式，留到生成Actor后进行。

通过**NewObject**模型**Actor** 对应的代码为：

```
AActor* const Actor = NewObject <AActor >(  
LevelToSpawnIn ,  
Class ,  
NewActorName ,
```

```
SpawnParameters.ObjectFlags ,  
Template);
```

请注意，第一个参数为Outer，即负责序列化和反序列化的对象。

从这里可以看出Actor由当前关卡（如果没有手动指定的话）负责存储信息。这里就会调用以const FObjectInitializer&为参数的构造函数，如果读者书写了很多Actor的代码，应该能够回忆起，正是在这个函数里面，我们通过CreateDefaultSubobject等函数构造出了默认的组件。相应的，组件的构造函数也在此时被调用。

构造**Actor**后的处理 可以理解为两个步骤：注册Actor和初始化Actor。阅读这一段时可以参考Actor生命周期图中SpawnActor那一列，会更有顺序感。

1. 添加Actor到当前关卡的Actor列表中。
2. 调用**PostSpawnInitialize** 函数。
 - a. 获取Actor根组件以计算Actor的位置。
 - b. 调用**DispatchOnComponentsCreated** 函数，从而调用所有之前通过CreateDefaultSubobject函数构造出来的组件的OnComponentCreated函数。
 - c. 通过RegisterAllComponents注册所有的Actor组件。实质上调用组件的RegisterComponent函数。所以，如果动态生成和添加组件，务必自己手动调用该函数以注册。
 - d. 设置当前Actor的Owner [\[6\]](#)。
 - e. 调用**PostActorCreated** 函数，通知当前Actor已经生成完成。在Actor中，这个函数只是一个空函数，开发者可以根据自己的需求在这里进行处理。

f. 调用**FinishSpawning** 函数：

- i. 调用**ExecuteConstruction**函数。如果当前Actor继承自蓝图Actor类，此时会拉出所有的蓝图父类串成一串，然后从上往下调用蓝图构造函数（此时顺便也会把Timeline组件生成出来）。最后调用用户在蓝图定义的构造函数脚本。也就是说，在蓝图中书写的构造函数实际上不是在构造阶段被调用！
- ii. 调用虚函数**OnConstruction**，通知C++层，当前脚本构造已经完成。开发者可以重载该函数，介入这一过程。

g. 调用**PostActorConstruction** 函数，该函数主要处理了组件的初始化过程。蓝图也有可能创建自己的组件，因此直到这时才获得了所有Actor已经被创建但是还没有初始化的组件，组件的初始化需要放到这里进行。

- i. 调用**PreInitializeComponents** 函数。对于Actor来说，这里只处理一件事情：如果当前Actor需要自动获取输入，则试图获取当前PlayerController，然后启用输入；如果当前PlayerController不存在，则调用当前Level的**RegisterActorForAutoReceiveInput**以启用输入。该函数可以被重载以实现自己的逻辑。
- ii. 调用**InitializeComponents**，实质是遍历当前所有的组件，调用其**InitializeComponent**函数以通知初始化。这也是一个通知函数，开发者可以重载该函数以实现自己的初始化。父类只是简单设置**bHasBeenInitialized**为真。
- iii. 根据前文提到的碰撞处理方式，对当前Actor的位置进行处理。例如开发者希望能够在移动Actor位置后生成，则调整位置来让Actor能够顺利产生出来，不至于卡住。如果是产生不出来就摧毁的类型，那就摧毁掉Actor，然后

输出一个Log提示。

- iv. 调用**PostInitializeComponents**。对于Actor类来说，这里只是设置bActorInitialized为真，顺便向寻路系统注册自己，然后调用UpdateAllReplicatedComponents函数。这个函数同样可以重载以实现自己的逻辑。
- v. 调用著名的**BeginPlay** 函数。这是这个Actor的第一声“啼哭”，即第一次Tick。接下来这个Actor将会进入自己的Tick循环。

对于Actor的创建来说，最大的问题在于处理Actor所属组件的创建。不仅仅C++层会有默认的组件，蓝图层也会有组件产生。因此，对组件的初始化会延后到所有组件被注册到世界之后进行。需要注意的是，Actor默认的BeginPlay函数会通知蓝图层的BeginPlay，换句话说，如果你重载后首先调用Super::BeginPlay，会发现蓝图BeginPlay会首先调用，然后才是自己C++层的BeginPlay，顺序正好相反。

10.2.2 加载

Actor作为UObject的子类，其加载过程前部分依然遵循UObject的加载过程。Actor只会调用自己的PostLoad函数来做一些处理，不同的子类有不同处理方式，对于Actor基类来说，该函数主要实现：如果当前Actor存在Owner，添加自己到Owner的Children数组中。由于Children数组虽然是UProperty，但是被标记为transient，不会参与到序列化中。

请注意，如果是在编辑器中，到这一步，序列化已经完成，直到进入PIE模式才会继续进行Actor初始化过程。读者可能希望了解游戏世界的初始化过程，其大致过程如下：

UWorld::InitializeActorsForPlay 该函数依序调用以下函数：

UWorld::UpdateWorldComponents 添加属于世界（但是不属于任何Actor）的组件，例如绘制线条用的 PersistentLineBatcher等。同时如果有其他的子关卡，这里开始更新子关卡的组件。

GEngine->SpawnServerActors 根据服务端的Actor信息，在本地生成对应Actor。

AGameMode::InitGame 调用当前游戏模式的初始化函数。

ULevel::RouteActorInitialize 遍历所有关卡，对Actor进行初始化。请注意，此时Actor的组件都是已经生成完毕了的，序列化的时候已经存储了足够的数据。此时需要进行的仅仅是通知初始化。故该函数遍历关卡中的所有Actor，执行顺序如下：

AActor::PreInitializeComponents

AActor::InitializeComponents

AActor::PostInitializeComponents

AActor::BeginPlay

ULevel::SortActorList 对所有关卡（包括子关卡）的Actor对象进行排序，准确的说应当是整理。这里是针对网络设计的，将Actor按照是否需要网络同步，放入对应数组进行整理。

UNavigationSystem::OnInitializeActors 通知寻路系统开始初始化。

UAISystem::InitializeActorsForPlay 通知AI系统初始化。

正是由于UObject序列化机制非常强大，因此Actor能够比较完整地保存当前的状态。当加载后，只需要进行有限的初始化工作，就能够还原Actor的工作状态。同时，这一段流程也描述了，如果希望在加载和创建**Actor**的时候均执行某段函数，应当如何重载函数完成。简而言之：

1. 希望在Actor生成的时候触发，但是不希望在Actor加载的时候触发，可以重载PostActorCreated这样的函数，具体时机可以查看前面的函数调用序列，主要取决于是否需要假定组件均已经初始化。
2. 希望在Actor加载的时候触发，可以重载PostLoad函数。
3. 希望均触发，可以按需求考虑重载BeginPlay函数。

10.2.3 释放与消亡

在第一部分已经阐述，Actor的释放和销毁通过手动请求Destroy实现。调用一个Actor的Destroy函数销毁Actor的过程如下：

UWorld::DestroyActor Actor的Destroy函数只是一个辅助函数，实际上是请求当前世界来摧毁自己。该函数完成：

Actor::Destroyed 调用该函数通知本Actor已经被摧毁。虽然这个Actor尚未被垃圾回收，但是在这一步发送通知，意味着在此之后不能再访问本Actor。这里也会在网络上通知本Actor

被摧毁。

取消绑定子 Actor 虽然父级Actor被删除了，但是子Actor不应当被同样删除，因此需要从父级Actor上取消绑定。

Actor::SetOwner(NULL) 将Actor的Owner设置为空，不再参与服务器同步。

UWorld::RemoveActor 从当前世界的Actor列表中移除当前Actor。

Actor::UnregisterAllComponents 通知所有的组件，从世界中取消注册。

Actor::MarkPendingKill 设置PendingKill这个标记，表示当前UObject需要被删除。

Actor::MarkPackageDirty 设置最外侧的Package为Dirty。

Actor::MarkComponentsAsPendingKill 对当前Actor的所有子组件添加PendingKill。

Actor::RegisterAllActorTickFunctions 取消注册所有的ActorTick函数。

触发垃圾回收 当下一次垃圾回收过程开始的时候，会检测到这个Actor已经没有任何引用（已经从ULevel的ActorList中被取下），且被设置为PendingKill。于是进入到UObject的垃圾回收流程。

由此可见，Actor的释放过程主要有两个重要任务：

1. 通知服务器在服务端和各客户端之间删除当前Actor。
 2. 从世界Actor列表等取消注册自己，从而完全删除当前Actor和世界的关联，从而满足垃圾回收的要求。
-

[1] 博客地址<http://blog.csdn.net/huangzhipeng>。

[2] 虚幻引擎4垃圾回收，风恋残雪：
http://www.cnblogs.com/ghl_carmack/p/6112118.html。

[3] 程序员往往忘记去判断哪些是弱指针、哪些是强指针，从而导致内存释放的问题。如果你编写Slate应用程序，一定要注意这个问题。

[4] 可以通过AddToRoot函数手动将UObject添加到根以避免被错误垃圾回收。

[5] 对应链接：
<https://docs.unrealengine.com/latest/CHN/Programming/UnrealArch>

[6] Owner与Outer不同。在网络同步中，Owner用于指定当前Actor的主人，其中一个用处是：如果当前Actor的Owner是本地玩家，则不会从服务器端同步状态到本地，而是将本地状态向服务端同步。

第11章

虚幻引擎的渲染系统

问题

虚幻引擎这么厉害的画面，是通过什么样的过程产生的呢？

从哪里开始渲染，从何处取回渲染结果？

虚幻引擎的渲染系统是一套非常庞大的系统，跨越多个线程，由大量的类辅助完成。因此本章的讲解将会采用“从大笔泼墨的结构性描述，到工笔细描的细节性描述”这样的思路。

11.1 渲染线程

对虚幻引擎有一定研究的读者一定曾听说“游戏线程（逻辑线程）”与“渲染线程”的说法。而笔者认为，对渲染线程的理解可以从这一句话入手：“渲染线程是游戏线程的奴隶”。对应的含义是以下两点：

外包团队 渲染线程是一个外包团队，其实质上并不知道自己真正执行了什么。游戏线程不停地向渲染线程对应的任务池派送任务，渲染线程只是埋头去执行。

追本溯源 对渲染线程执行逻辑的分析离不开对游戏线程渲染相关代码的分析。游戏线程是木偶师，渲染线程是木偶，只看木偶不能窥得全貌，必须要看木偶师的想法，才能理解整个系统。

对于渲染线程的设计，两种方案是：

1. 渲染线程具有独立的逻辑系统。
2. 渲染线程只是简单地作为工作线程不断执行游戏线程赋予的工作。

大多数的引擎都选择了工作线程的方案。例如IdTech引擎采用了将渲染指令打包成一个一个的数据包，发送到渲染线程进行执行的方式，被称为渲染前端-后端方案。渲染后端线程依然是一个简单工作线程，只是随着历史进步，虚幻引擎借助C++的语言特性给出了语法上更优美的方案。游戏线程方案能够大幅度简化线程同步的逻辑，渲染线程如果频繁阻塞同步，对整体效率会有极为不利的影响。

11.1.1 渲染线程的启动

虚幻引擎的Slate系统本身借助虚幻引擎RHI渲染，但是我们暂时抛开这一点，关注场景本身的渲染，不引入过多的讨论内容。

虚幻引擎在FEngineLoop::PreInit中对渲染线程进行初始化。具体的位置是在StartRenderingThread函数里面。此时虚幻引擎主窗口甚至尚未被绘制出来。

渲染线程的启动位于StartRenderingThread全局函数中。大致来说，这个函数执行了以下内容：

1. 创建渲染线程类实例。
2. 通过FRunnableThread::Create函数创建渲染线程。
3. 等待渲染线程准备好从自己的TaskGraph取出任务并执行。
4. 注册渲染线程。
5. 创建渲染线程心跳更新线程。

11.1.2 渲染线程的运行

渲染线程的主要执行内容在全局函数RenderingThreadMain中，实质上来说，渲染线程只是一个机械性地执行任务包的“奴隶”，如图11-1所示。游戏线程会借助EQUEUE_Render_COMMAND系列宏，向渲染线程的TaskMap中添加渲染任务。渲染线程则不断提取这些任务去执行。

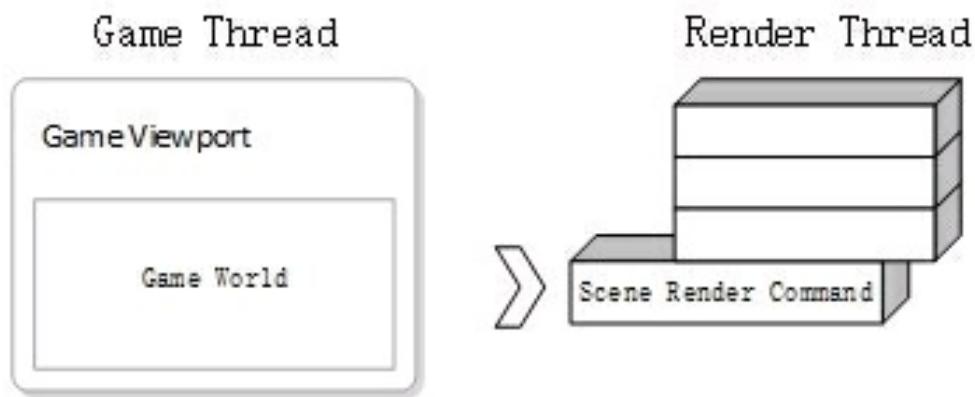


图11-1 游戏线程与渲染线程交互

另外，需要注意的一点是，渲染线程也并非直接向GPU发送指令，而是将渲染命令添加到RHICommandList，也就是RHI命令列表中。由RHI线程不断取出指令，向GPU发送，并阻塞等待结果。此时RHI线程虽然阻塞，但是渲染线程依然正常工作，可以继续处理向RHI命令列表填充指令，如图11-2所示。从而增加CPU时间的利用率，避免渲染线程

凭空等待GPU端的处理。

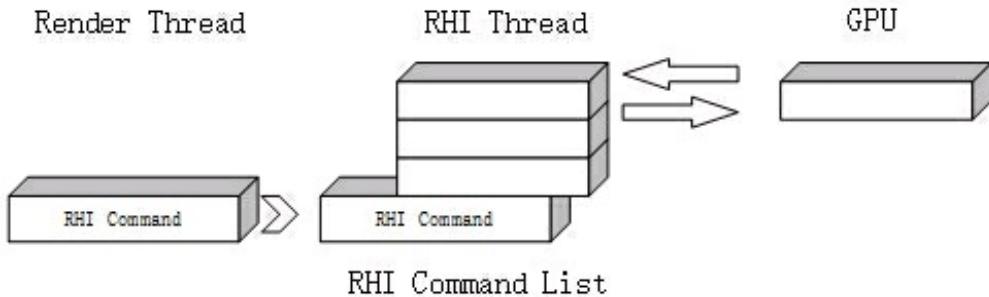


图11-2 渲染线程与RHI线程

11.2 渲染架构

11.2.1 延迟渲染

延迟渲染，英文名为Deferred Rendering。虚幻引擎对于场景中所有不透明物体的渲染方式，就是延迟渲染。而对于透明物体的渲染方式，则是前向渲染（Forward Rendering）。

所谓延迟渲染，是为了解决场景中由于物体、光照数量带来的计算复杂度问题。假如场景中有5个物体和5个光源，每个物体都单独计算光照，那么计算量将是 $5 \times 5 = 25$ 次光照计算。随着光源数量的提升，光照计算的计算量会呈几何级数上升。这样的代价是非常高昂的。而延迟渲染则是将“光照渲染”延迟进行。每次渲染，将会把物体的BaseColor（基础颜色）、粗糙度、表面法线、像素深度等分别渲染成图片，然后根据这几张图，再逐个计算光照。带来的好处是，无论场景中有多少个物体，经过光照准备阶段之后，都只是变成了几张贴图，计算光照的时候，根据每个像素的深度，能够计算出像素在世界空间位置，然后根据表面法

线、粗糙度等参数，带入公式进行计算。于是之前 5×5 的光照渲染计算量，被直线降低为 $5+5$ 的计算量。 $5+5$ 的来历是：

5个物体→多张贴图：需要5次渲染。

5个光源→逐光源计算光照结果：需要5次渲染。

随着光源数量的增多，延迟渲染节省的计算量会越来越多。

但是延迟光照的代价是，半透明的物体无法被渲染。因此在虚幻引擎里面，是先进行延迟光照计算不透明物体，然后借助深度排序，计算半透明物体的。因此我们可以看到玻璃折射材质，假如背后是不透明物体，效果还是相当有趣的，但是当其背后是半透明物体，就容易由于深度排序出现各种问题，这也是需要开发者进行注意的。

11.2.2 延迟渲染在**PostProcess**中的运用

刚刚介绍了虚幻引擎的延迟渲染架构，那么这样的知识是否有用处呢？我们都强调，理论联系实践，那么我们怎么理论联系实践呢？

首先，由于虚幻引擎4采用的延迟渲染架构，导致我们无法像虚幻引擎3那样，获取到光源的向量LightVector——因为材质在被计算的时候，光照还没有发生。这也限制了我们在材质设计上的发挥，比如以前我们制作Tone Shader动画风格的材质，是可以在材质中完成的，现在我们已经无法这样做了。

那么我们的思路就必须跟随虚幻引擎4的延迟渲染架构进行转变，很多设计就得在Post Process中完成。我们可以在Post Process场景处理

中，获取到之前渲染出来的缓冲区，如Scene Color、Rough、World Normal等。这些东西的运用，就看大家仁者见仁智者见智了。

譬如可以通过对深度缓冲区运用边缘检测算法，从而检测出模型的外边缘，然后通过计算当前像素周围几个像素的World Normal与当前像素的World Normal的夹角变化值，从而检测出模型的内边缘。组合外边缘和内边缘，就能完成比较接近手绘的描边效果。

当然，这个方案的局限性在于，外边缘的粗细不容易被美术调整。关于描边方案的讨论，推荐大家查看《罪恶装备XXrd》的开发者访谈，其中有对其设计描边、动画渲染方案的思路的详细描述。

这里只是举例，对于Post Process中延迟渲染信息的利用，肯定有许许多多的方案。本人才疏学浅，只能举出这样的案例，如果希望在这个方面有更深层次的运用，可以参考虚幻引擎官方商城相关的案例。

11.3 渲染过程

上文主要阐述了虚幻引擎整体架构的设计。并且给出了一些名词的定义。但是上文没有阐述一个问题：屏幕上的画面究竟是如何呈现的？也就是说，我们在屏幕上看到了一棵“树”，我们知道这棵树的结果——花花绿绿的像素，也知道这棵树的来源——导入的fbx文件、创建的材质及定义的Static Mesh Component静态网格物体组件，但是它们之间的过程是怎样的呢？如图11-3所示。

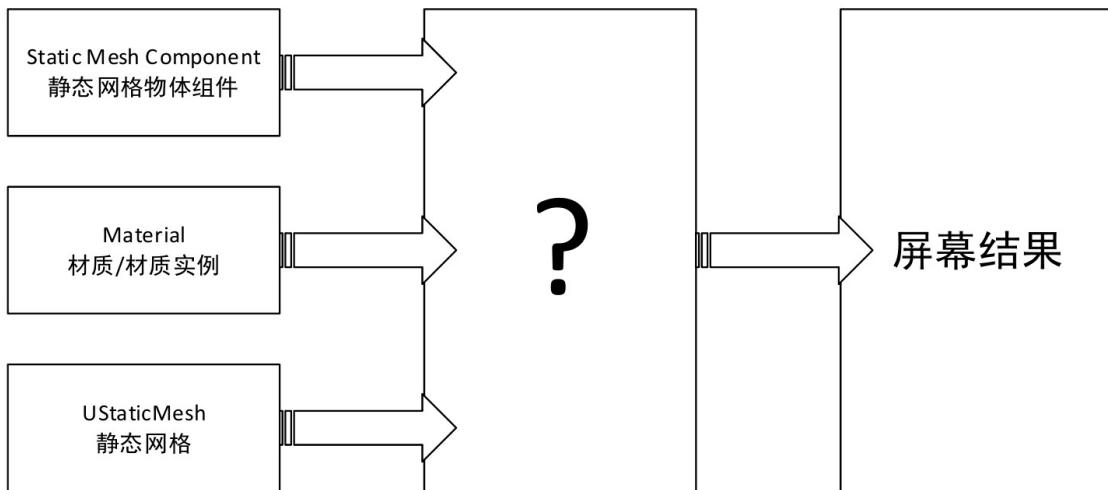


图11-3 中间的过程是什么

前文有述，虚幻引擎的渲染过程是一个异步的、多线程配合的过程。而为了方便描述和读者理解，这里不再考虑RHI线程的RHI绘制指令列表这一部分的异步性，也就是假定当一个渲染指令被填充到RHI绘制指令列表后，该指令直接被完成。我们把异步的分析主要集中在游戏线程和渲染线程之间的通信上。

同时，我们这一次的分析将会采用倒推的方式，从结果向来源逆推。笔者也尽量给出自己的分析过程，并提示对应源码的位置，让读者也能理解笔者分析的思路，而不是直接看到笔者的结果，相对来说更容易接受。

11.3.1 延迟渲染到最终结果

前文有对延迟渲染原理的描述。故这里将直接开始对虚幻引擎延迟渲染过程的步骤进行分析，如图11-4所示。这部分主要分析的内容是FDeferedSceneRenderer::Render函数。

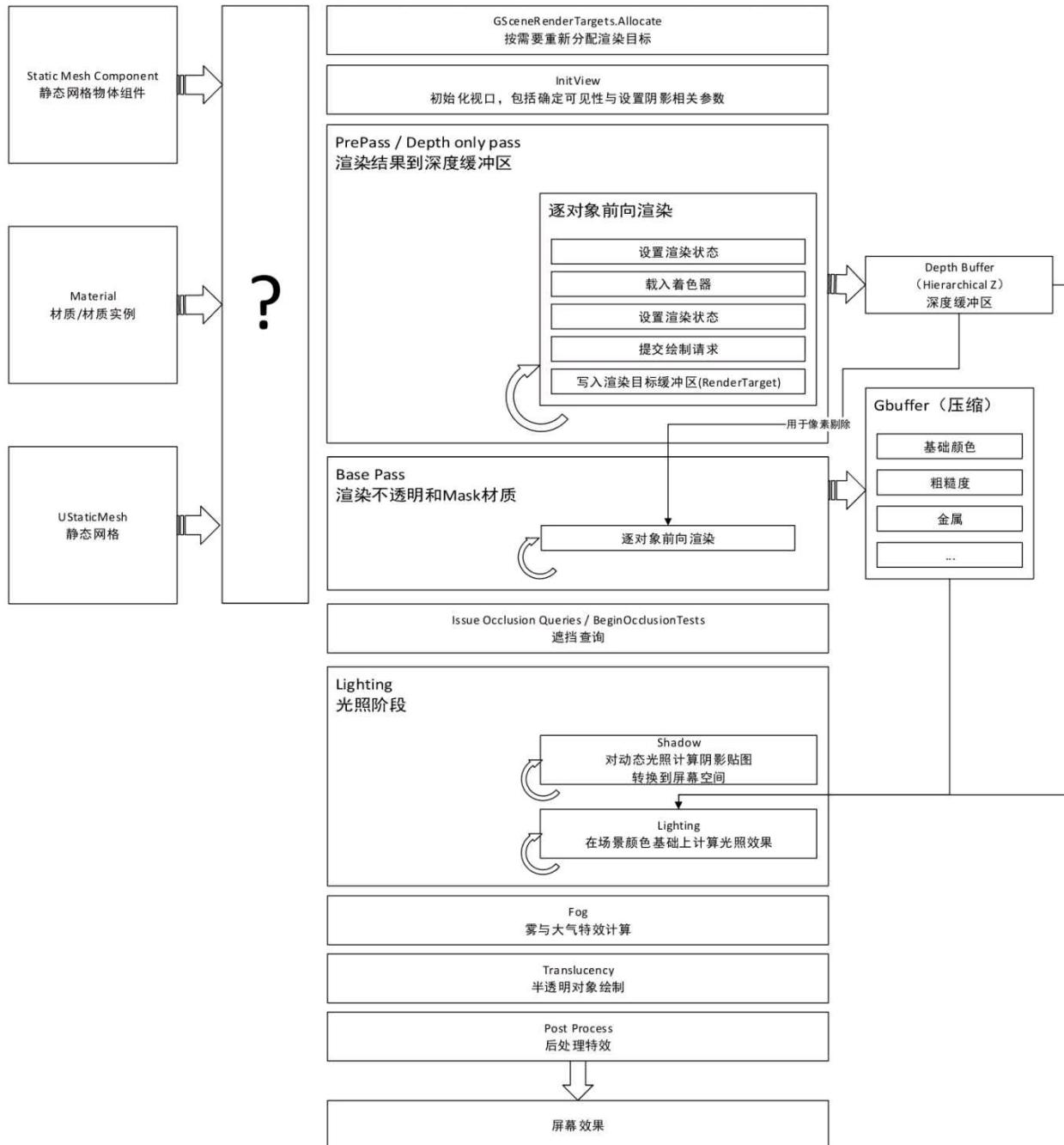


图11-4 延迟渲染流程到最终结果

准备

在进行分析之前，我们势必要做出一些假定，从而确保我们的讨论能够继续。现在我们已经知晓了摄像机的角度，并且完成了场景中对象的收集，它们已经进行了归并，而且转化为了很容易获取顶点缓冲区、索引缓冲区，以及着色器信息的某种中间格式。至于这个归并和转化过程，我们将会在下一节分析。此时我们不妨丢掉实体的概念，想象你手里的只有一层“皮囊”，那就是由顶点和索引规定的三角形面片组合，以及对应的着色器信息。每个对象都对应这么一个皮囊。这就是接下来分析的数据基础。

初始化视口

在重新分配完渲染目标之后，我们要做的第一件事情是什么？不能一开始就着手绘制——因为此时我们拿到的是整个场景的皮囊。全部画完不是不可以，但是大量无用的计算会花在完全处在屏幕之外的皮囊上。注意观察那幅流程图11-4，即使是延迟渲染，也有几个Pass是使用前向渲染管线，逐个皮囊地绘制到缓冲区中的。

因此，此时有必要进行可见性剔除。需要注意的是，该函数命名为InitViews，这是为了照顾不止一个视口的情况，例如编辑器的三视图。此时可以一口气对多个视图进行渲染，而不是逐视图地渲染，能够节约很多过程。如果一口气渲染多个视图，又只根据一个视图进行剔除，那么其他视图中这个对象会根本显示不出来。

剔除过程分为三步：预设置可见性，可见性计算，完成可见性计算。

预设置可见性 对应函数为PreVisibilityFrameSetup。该函数主要执行以下内容：

1. 根据当前画质设置，设置TemporalAA的采样方式，同时确定采样位置。这个采样位置会用于微调接下来的矩阵。如果读者对这个微调不大理解，可以参考Epic关于TemporalAA采样的PPT。简单来说：

TemporalAA采样希望更精确地对一个像素覆盖的区域进行求解。一种思路是用更高的分辨率进行渲染，这样单个像素对应的区域实际上渲染了多个像素，然后将多个像素的值进行混合得到单一像素的值。这是一种“暴力”提高精确度的方法，代价也很高昂。另一种思路则更有技巧性，即每一帧渲染的时候，让这个像素覆盖的位置进行微弱偏移，然后混合前面几帧的渲染结果。

打个比方，假如现在有一排12个机器人狙击手向正前方射击，它们只会向正前方开火。此时处在两个狙击手中间的敌人会被“漏掉”从而不被击中。我们希望能够打中更多的敌人，一种方式是在每两个狙击手中间再放一个狙击手，这是“暴力”提高精度。另一种方式是让狙击手每次射击时，都往左右随机稍微偏一点，这样就能在不提升狙击手数量的情况下，击中更多的敌人。放在我们这里，就是让当前像素能够获得更多的信息。

2. 设置视口矩阵，包括视口投影矩阵和转换矩阵。

可见性计算 对应的函数为ComputeViewVisibility。该函数主要执行以下内容：

1. 初始化视口的一系列用于可视化检测的缓冲区。这些缓冲区实际上是一系列的位数组。用0和1代表是否可见。所以遵照习惯，翻译为位图，但是请将这个位图与bmp位图区分开来。这个对应的是

BitMap，也就是用于查询的逐位的数组。

2. 排除特殊情况，大多数情况下视口都使用平截头体剔除。对应的是模板函数FrustumCull。该函数内部使用了ParallelFor函数来进行并行化的异步剔除。所谓平截头体剔除就是，摄像机视口的远近平面构成了一个类似梯台的体积，在该体积内的对象会被保留，在该体积外的对象的可视化检测缓冲区对应的比特位就会被设置为0，表示不会被看见。
3. 对于编辑器来说，还有一个特殊的步骤。此时会对过小的线框直接剔除掉，以加速整体性能。如果是在线框模式下，此时所有的非线框也会被剔除掉。
4. 在非线框模式下，此时会对处于视口范围内，但是被别的对象遮挡的对象进行一次剔除。这个过程用的是上一帧计算遮挡结果。这个数据来源，请见下文中对HZB的介绍。
5. 在此之后，根据所有的可见性位图，设置每个需要渲染的对象的可见性状况，即Hiddenflag。
6. 接下来会给每个对象返回自己是否可见的机会，这是开发者可以自行干涉的部分。下一节会讲述具体如何操作。
7. 最后一步是获取所有动态对象的渲染信息，对应函数是每个RenderProxy的GetDynamicMeshElements函数。下一节会详细讲述RenderProxy，以及静态和动态的对象在渲染过程中的区别。这里请读者先作为一个步骤记忆下来，可以把RenderProxy理解为前文的比喻“皮囊”，网格物体组件对应的皮囊是网格物体的RenderProxy，持有顶点和索引信息，材质对应的皮囊是MaterialRenderProxy，持有需要的着色器信息。在阅读完后文关于RenderProxy讲解的时候可以再次回头来看这个过程。

完成可见性计算 对应的函数为PostVisibilityFrameSetup。

1. 对半透明的对象进行排序。半透明对象的渲染由于涉及互相遮挡，则必须按照从后往前的顺序来渲染，才能确保渲染结果的正确性。因此，需要此时完成排序。
2. 对每个光照确定当前光照可见的对象列表，这里也是使用平截头体剔除。
3. 初始化雾与大气的常量值。

在这个阶段也完成对阴影的计算。包括对覆盖整个世界的阴影、对固定光照的级联阴影贴图和对逐对象的阴影贴图的计算。

需要注意的是，虚幻引擎的剔除方式是借助ParallelFor的线性剔除，而不是借助八叉树等的树状剔除。对应的大致代码为：

```
template<bool UseCustomCulling , bool bAlsoUseSphereTest >
static int32 FrustumCull(const FScene* Scene, FViewInfo& View)
{
    ParallelFor(NumTasks,
    [&NumCulledPrimitives , Scene, &View,
    MaxDrawDistanceScale](int32 TaskIndex)
    {
        //...
        for (int32 WordIndex = TaskWordOffset;
            WordIndex < TaskWordOffset +
            FrustumCullNumWordsPerTask && WordIndex *
            NumBitsPerDWORD < BitArrayNumInner;
            WordIndex++)
    {

```

```
//...
for (int32 BitSubIndex = 0;
     BitSubIndex < NumBitsPerDWORD &&
     WordIndex * NumBitsPerDWORD +
     BitSubIndex < BitArrayNumInner;
     BitSubIndex++, Mask <= 1)
{
    //...
}
//...
}
},
);
return NumCulledPrimitives.GetValue();
}
```

寒霜引擎的开发者在PPT中也有论述，并行化的线性结构剔除在性能上会优于基于树的剔除。如果读者有兴趣深究此问题，可以在网上搜索《Culling Battlefield》，有详细的性能剖析报告。在虚幻引擎的该段代码的注释也给出了性能优势的剖析报告，如果读者拥有UDN的账号，也可以进入研究一番。

PrePass预处理阶段

PrePass的主要目的是为了降低Base Pass的渲染工作量。通过渲染一

次深度信息，如果某个像素点的深度检测失败，即不符合当前深度值，那么这个像素将不会进行工作量最大的像素渲染器计算。打个比方，这就像我们在制作浮雕的时候，首先在表面上雕刻出浮雕的外形，这时表面上每一个点都是距离人眼最近的点，它们是一定能够被看见的——这样一来，对浮雕上的每个点进行上色，都只会对真的能够看见的点进行绘制，而不需要绘制那些在浮雕“里面”的点。

PrePass过程是可选的。在笔者的Windows平台下的主机上，这个步骤经常被略过执行。这取决于NeedsPrePass函数的返回结果。其判断以下几个条件：

1. 不是基于分块的GPU，即TiledGPU，例如ARM移动端平台很多就是。
2. 渲染器的EarlyZPassMode参数不为DDM_None，或GEarlyZPassMovable不为0。

两者必须同时满足，才会进行PrePass计算，否则是不会进行的。

此时参与渲染的包括不透明对象和Mask对象。这些对象中，静态网格物体构成了一套列表，即Scene的DepthDrawList与MaskedDepthDrawList。在列表绘制完毕之后才会收集动态对象，再次绘制。

从渲染过程示意图中可以看到，对象的渲染始终按照设置渲染状态、载入着色器、设置渲染参数、提交渲染请求、写入渲染目标缓冲区的步骤进行。这个过程也是传统的图形API逐个绘制的基本思路。有些步骤可以归并，例如设置渲染状态，由于并非每次都需要修改，因此部分状态的设置被提前到最开头统一设置。这里只是给出一个通用的步骤

过程，方便读者去理解。实际上虚幻引擎的渲染过程考虑了诸多问题，包括能否并行化、是否打开针对VR进行优化的InstancedStereo技术等，对于这些条件，虚幻引擎都有对应的渲染路径设计。我们的步骤分析依旧遵循抓大放小的思路，以主要路径为介绍重点，略去大量的特殊情况判断。

总体来说，预处理阶段的工作过程如下：

设置渲染状态 这个步骤对应函数SetupPrePassView，目的在于关闭颜色写入，打开深度测试与深度写入。PrePass这个步骤不需要计算颜色，只需要计算每个不透明物体的像素的深度。

渲染三个绘制列表 这三个绘制列表是由静态模型组成的，通过可见性位图控制是否可见。渲染顺序依次为：

1. 只绘制深度的列表PositionOnlyDepthDrawList，这个列表里的对象只在深度渲染过程中起作用。
2. 深度绘制列表DepthDrawList，这是最主要的不透明物体渲染列表。
3. 带蒙版的深度绘制列表MaskedDepthDrawList，蒙版即对应材质系统中的Mask类型材质。

绘制动态的预处理阶段对象 这个阶段会通过ShouldUseAsOccluder函数询问Render Proxy是否被当作一个遮挡物体，同时也会配合其他情况（是否为可移动等），决定是否需要在这个阶段绘制。

大体的过程就是如此，不过我们对一个核心的问题一笔带过：绘制过程。绘制过程的步骤实质上是三个：设置绘制状态并载入着色器，设置着色器参数，提交渲染请求。至于最终的写入渲染目标，是在步骤之

前就已经通过RHI的SetRenderTarget设置好了。GPU会默认地将渲染结果写入到当前设置的渲染目标中，不需要我们人为进行干涉。由于多个Pass中绘制过程是类似的，典型案例是TStaticMeshDrawList::DrawVisible函数，故下面我们将基于这个函数重点分析这三个步骤。

DrawVisible绘制可见对象

前文有述，绘制可见对象的基础是可见对象列表。尤其是对于静态网格物体而言，是以TStaticMeshDrawList为单位进行成批绘制的。在绘制之前，每个绘制列表已经进行了排序，尽可能公用同样的绘制状态。具体如何排序，这里暂时略去不讲，在后文会详细描述。此时读者只需要知道，每个列表都公用以下着色器状态：

1. 顶点描述Vertex Declaration
2. 顶点着色器Vertex Shader
3. 壳着色器Hull Shader
4. 域着色器Domain Shader
5. 像素着色器Pixel Shader
6. 几何着色器Geometry Shader

换句话说，你可以把一个着色器状态看作是顶点描述和整个渲染管线用到的着色器对象的集合。而同一个列表中，这些东西都是一致的，区别只是在于渲染过程具体参数不同，如顶点缓冲区不同、索引缓冲区不同。

载入公共着色器信息 这是逐列表完成的，对应的函数为
SetBoundShaderState和SetSharedState。

1. SetBoundShaderState载入了需要的着色器。
2. SetSharedState则因不同类型而异。对比较有代表性的TBasePass而言，是设置顶点着色器和像素着色器的参数。如果是透明物体，即透明度设置为Additive、Translucent、Modulate，会在此时再次设置混合模式。

逐元素渲染 需要注意的是，此时的元素并非是一比一对应，而是经过组合的，即并非Element，而是BatchElement。主要包含两个子步骤：

1. 对每个DrawingPolicy调用SetMeshRenderState函数，设置渲染状态。包括调用每个着色器的SetMesh函数，以设置与当前Mesh相关的参数，如变换矩阵等。
2. 调用当前Batch Element的DrawMesh函数，实际完成绘制。需要注意的是，顶点缓冲区和索引缓冲区是一开始就已经上传到显存准备好了的，这时候只需要调用RHICmdList的DrawIndexedPrimitive函数，指定好顶点缓冲区和索引缓冲区的位置，就可以了。这两个位置是包含在当前Element持有的信息里面的。

BasePass

BasePass是一个极为重要的阶段，这个阶段完成的是通过逐对象的绘制，将每个对象和光照相关的信息都写入到缓冲区中。其中逐对象渲

染的过程与前文对DrawVisible的分析过程一致，那么核心问题就是：

1. 逐对象绘制后的结果如何写入到缓冲区中？
2. 缓冲区中的内容如何管理？

理论上这里还应该有个问题：如何还原缓冲区中的参数，以进行光照计算？而这个问题将会留到下文对光照过程的分析中进行描述。

如果你对前文中PrePass阶段的绘制过程还有印象，那么你会发现，BasePass的过程和PrePass的过程非常接近——它们都按照设置渲染视口、渲染静态数据及渲染动态数据三个步骤完成。

设置渲染视口 这一次渲染视口的状态设置与PrePass略有差异。

1. 如果PrePass阶段已经写入深度，则深度写入被关闭，直接使用已经写入的深度结果进行深度检测。
2. 通过

```
RHICmdList.SetBlendState(TStaticBlendStateWriteMask<CW_RGBA,  
                           CW_RGBA, CW_RGBA, CW_RGBA>::GetRHI());
```

打开了前4个渲染目标的RGBA写入。TStaticBlendStateWriteMask这个模板是用模板参数定义渲染目标是否可写入，最高支持8个渲染目标，这里只打开了前4个。

3. 设置了视口区域大小。这个大小会因为是否开启InstancedStereoPass而有所变化。

渲染静态数据 与PrePass有一定区别的是，如果之前已经进行了深度渲染，那么会首先渲染Masked蒙版对象，然后渲染普通的不透明对象。否则就会反过来，先渲染不透明对象，再渲染蒙版对象。

渲染动态数据 区别不大。

需要注意的是，BasePass阶段采用了MRT(Multi-Render Target)多渲染目标技术，从而允许Shader在渲染过程中向多个渲染目标进行输出。有读者就会问，那这几个渲染目标从哪里来的？答案是由当前请求渲染的视口（Viewport）分配的，并不归属SceneRenderer分配。对应的是FSceneViewport::BeginRenderFrame函数。

解决了“从哪里来”的问题，还有两个问题要解决：如何写入？向何处去？看来我们的分析方式带有了点古希腊哲学的味道。

如何向多个渲染目标写入？这个问题的答案并不在C++代码中，而是在Shader着色器代码中。如果你打开Engine/Shader/BasePassPixelShader.usf文件，就会看到著名的Main函数，定义了用到的多个输出（不考虑特殊情况）：SV_Target0输出了颜色数据，SV_Target1-3则作为GBuffer输出目标。整个着色器的代码非常长，这里只是大概描述过程：通过GetMaterialXXX系列函数，获取材质的各个参数，比如BaseColor基本颜色、Metallic金属等。然后填充到GBuffer结构体中，最后通过EncodeGBuffer函数，把GBuffer结构体压缩、编码后，输出到前面定义的几个OutGBuffer渲染目标中。

至于向何处去，由于这几个渲染目标会一直存在于当前的渲染上下文中，所以在下个阶段可以直接使用。换句话说，类似于炒菜的时候，最开始下单的那个服务员会给一个盘子，负责每个步骤的人会从上一个步骤的人那接过盘子，在自己处理过后，继续传递给下个人。这几个渲染目标会放在盘子里，不断向下传递。这就是BasePass的功能。

RenderOcclusion渲染遮挡

虚幻引擎的遮挡计算，实质上是在PrePass中直接进行基于并行队列的硬件遮挡查询。除非在r.HZBOcclusion这个控制台变量被设置为1的情况下，或者有些特效需要的情况下，才会开启HierarchicalZ-BufferOcclusion Cullin用作遮挡查询。

这个技术听上去十分厉害，但是全平台默认关闭，由于不清楚虚幻引擎在新的版本中是否会进一步完善和启用这个系统，故这里大致解释一下工作流程。如果读者对这个技术具体实现感兴趣，可以以 Hierarchical Z-Buffer Occlusion Cullin为关键词搜索，能够获得大量有关实现的论文。

总体来说，这个步骤是为了尽可能剔除处于屏幕内但是被其他对象遮挡的对象。之前我们有描述，在视口初始化阶段，剔除了处于视锥体之外的对象。但是依然有大量对象处于视锥体内，却被其他对象遮挡。比如一座山背面的一大堆石头，这些石头能够正常通过我们的视锥体遮挡测试，却并不需要渲染。

因此，HZB渲染遮挡技术被用于解决这个问题，通常的HZB步骤如下：

1. 预先准备屏幕的深度缓冲区，这个缓冲区将会作为深度测试的基础数据。因此，这个步骤必须在PrePass之后，如果没有PrePass，则必须在BasePass之后。
2. 逐层创建缓冲区的Mipmap级联贴图。层级越高，贴图分辨率越低，对应的区域越大。请想象成打马赛克，马赛克越大，每个马赛

克块遮盖的区域就越大。而每个马赛克的值对应这个区域“最远”元素到屏幕的距离（深度最大值）。

3. 计算所有需要进行测试的对象的包围球半径，根据这个半径，选择对应的深度缓冲区层级进行深度测试，判断是否被遮挡。这个的用意在于，如果对象较大，我们可以直接用更高层、更大块的马赛克进行测试——这个对象的深度若比这块马赛克对应的距离还远，那么该对象一定被遮挡，因为马赛克对应的是这一片区域中可见元素的最远距离。

需要注意的是，OpenGL平台下不会进行这个测试。这个步骤中的第二步可以使用像素着色器多次绘制完成级联贴图层级，第三步则可以使用计算着色器ComputeShader，或者使用顶点着色器进行计算，将结果写入到一个渲染目标中。从而借助GPU的高度并行化来加速这个遮挡剔除过程。

这个步骤写出的结果会被用于下一帧计算，而不是在本帧。

光照渲染

本节对应的函数是RenderLights。光照渲染与阴影渲染是分离的，阴影渲染是在视口初始化阶段完成，处于整个渲染流水线中非常靠前的位置。大体上的步骤如下：

1. 收集可见光源。除了对可见性标记的判断外，剔除的方法主要是视锥体裁剪。这一阶段会对每个光源构建FLightSceneInfo结构，然后通过ShouldRenderLights对光源是否需要渲染进行计算。这里不需

要再进行一次视锥体裁剪来判断可见性，直接利用最开始视口初始化阶段保存的VisibleLightInfos信息，以当前Id作为索引查询即可获得结果。

2. 对收集好的光源进行排序。将不需要投射阴影、无光照函数（LightFunction）的光源排在前面，从而避免切换渲染目标。
3. 接下来的阶段因具体情况而异。如果是存在基于图块的光照（TiledDeferredLighting），则通过RenderTiledDeferredLighting对光照进行计算。如果是PC平台，大部分光照依然使用RenderLight函数进行光照计算。
4. 接下来会进行判断，如果当前平台支持着色器模型5（Shader Model 5），则会计算反射阴影贴图与LPV信息。

略去较为复杂的部分不谈，核心光照渲染部分集中于RenderLight函数，每个光源都会调用该函数，其遍历所有视口，计算光照强度，并叠加到屏幕颜色上。该函数执行步骤如下：

1. 设置混合模式为叠加。
2. 判断光源类型：

平行光源 平行光源情况相对简单，对于虚幻引擎来说，平行光源只是一个没有范围概念，只有方向概念的光源。

- a. 载入延迟渲染光照对应的顶点和像素着色器（对应为TDeferredLightVS和TDeferredLightPS）。
- b. 设置光照相关参数。
- c. 绘制一个覆盖全屏幕的矩形，调用着色器。

非平行光源 此时比较复杂，如果摄像机在光源范围内，会出现问题。打个比方，如果点光源对应的几何体把摄像机扣住了，这个时候会出现渲染与预想效果不符的情况：球体一部分在摄像机

后面，直接被剔除，没有渲染；球体的另一部分虽然在摄像机正面，但是有可能会被别的对象挡住。挡住的部分就无法进行光照计算。所以此时需要特殊步骤，如图11-5所示。

- a. 判断摄像机是否在光照几何体范围内。
- b. 如果是，关闭深度测试，从而避免背面被遮盖部分不进行光照渲染。
- c. 否则，打开深度测试以加速渲染。
- d. 载入着色器。
- e. 设置光照相关参数。
- f. 根据是点光源还是聚光灯，绘制一个对应的几何体，从而排除几何体外对象的渲染，加速光照计算。

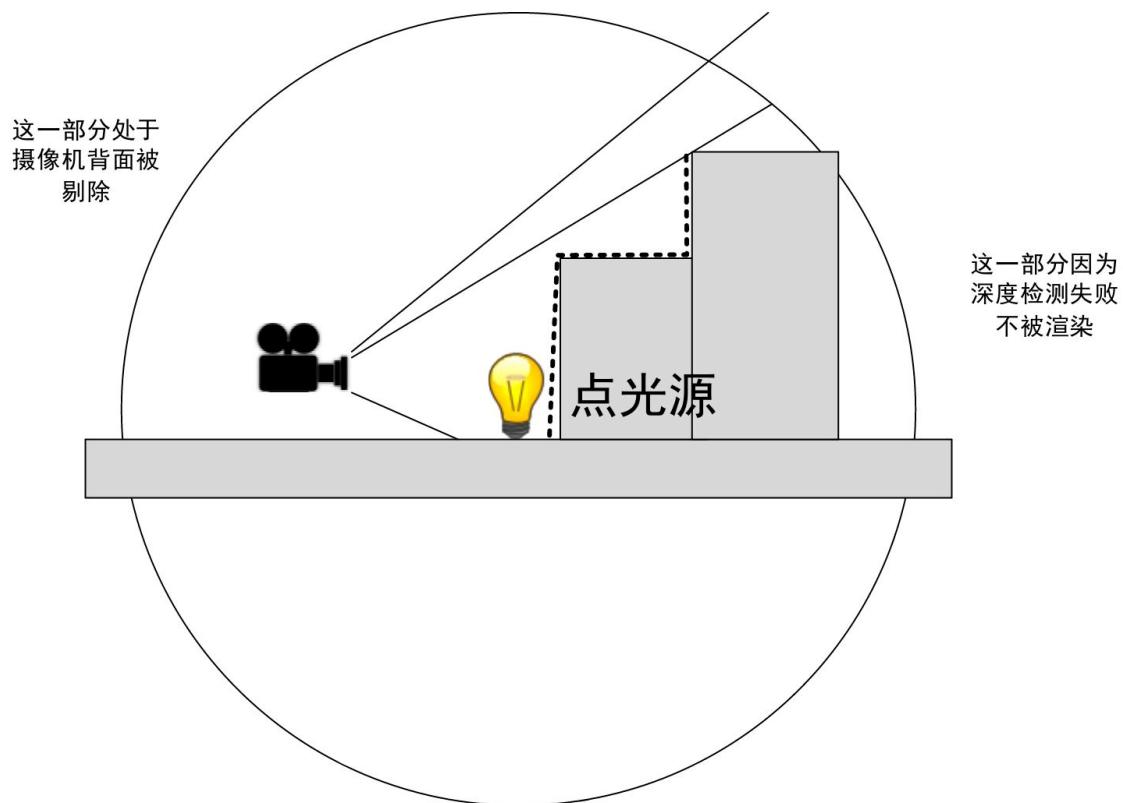


图11-5 点光源渲染：若不关闭深度检测，虚线部分光照不会计算

至此，对整个渲染过程中相对较重要、有共性的部分，进行了粗略的介绍，对于大气、透明物体、后处理过程，恕笔者不再着重叙述，一方面为了避免读者阅读过于枯燥，另一方面，读者可以根据前文分析与代码对应，理解虚幻引擎渲染设计基本规律，然后自行去剖析这些部分。笔者在介绍过程中，尽可能给出步骤对应的函数，如果读者对某一个过程有兴趣，可以在虚幻引擎源代码中查询对应函数，了解更多的细节。

11.3.2 渲染着色器数据提供

ShaderMap

关于ShaderMap应该被翻译成什么，笔者相当纠结，最终决定保留英文原文。Shader是着色器无误，但是Map这个概念很难描述，翻译为“着色器图”很容易被误解为是某个图像，实际上这里的Map可以理解为一个表格。

在讲述这个概念前，我们首先分析一下，虚幻引擎的着色器数量。如果你曾经重新编译着色器，会发现虚幻引擎的着色器数量高得令人恐怖，动辄是几千个着色器编译，看得让人心头发毛。而且这几千个着色器数量远远超过了材质的数量——我们发现了一个惊人的结论：单个材质会编译出多个着色器！

那么着色器按照什么样的方式被归类？方式如下：

1. 如果当前着色器类型继承自FMaterialShader，则对每个材质类型编

译出一组对应渲染管线的着色器，一般是顶点着色器、像素着色器组合。例如FLightFunctionVS/PS会对每个材质编译出一份实例。

2. 如果当前着色器类型继承自FMeshMaterialShader，则对每个材质类型的每个顶点工厂类型编译出一组顶点着色器和像素着色器。实际上，前文介绍的从延迟渲染到最终结果的过程中，每个Pass基本都有对应的顶点着色器和像素着色器，例如BasePass对应的是TBasePassVS/PS这一对着色器组合。

这里提到了顶点工厂，简而言之，顶点工厂负责抽象顶点数据以供后面的着色器获取，从而让着色器能够忽略由于顶点类型造成的差异，比如普通的静态网格物体和使用GPU进行蒙皮的物体，两者顶点数据不同，但是通过顶点工厂进行抽象后，提供统一的数据获取接口，供后面的着色器调用。

虚幻引擎的“材质”实质上是提供了一系列的函数接口，调用这些函数接口就能获得对应材质参数。单个材质不会产生一个完整的、带有具体执行过程的顶点着色器及像素着色器。读者不妨自行在材质编辑器中打开“HLSL代码”选项，能够看到材质转换完成的代码。

因此，对于继承自FMeshMaterialShader的着色器类型，读者可以这样理解：

1. 着色器不存在“动态链接”的概念，不可能出现动态切换某一部分函数的情况。换句话说，只要一部分函数的实现方式改变，最终会产生不一样的着色器原始代码，进而会编译出一个不一样的着色器对象。
2. 由于继承自FMeshMaterialShader的着色器类型需要适配不同的顶点工厂类型（同一份材质既可以被用于静态网格物体，也可以被用于

骨架网格物体），也需要适配不同的材质类型（每个材质由于节点不同，获取某个材质参数的表达式也就不一样），对于顶点工厂+材质类型的组合，只要有一项不同，就必须产生新的一份着色器代码用于编译。

因此，我们可以从两个维度来看待这个被虚幻引擎官方文档称为“稀疏矩阵”的着色器方案：

- a. 从渲染阶段（PrePass、BasePass等）角度来看，每个渲染阶段对应的着色器（如TBasePassVS/PS）都需要对每个顶点工厂、每个材质类型产生出一组着色器，放置在对应材质类型的着色器缓存中。
- b. 从材质类型角度来看，每个材质根据自身参与的渲染阶段不同，需要对每个顶点工厂类型、每个自己参与的渲染阶段，产生出一组着色器，放置在自己的着色器缓存中。此时被称为ShaderMap。

这就形成一套“三维”的着色器矩阵——你可以想象一个三维魔方，长度为每个材质类型，宽度为每个渲染阶段，高度为每个顶点工厂类型，那么这个魔方的每一个方格都对应了一组着色器组合，当然由于材质不一定参与全部阶段，这个魔方有很多的空缺。

这里再做一些额外的解释：实质上虚幻引擎还会根据光照贴图类型做出不同的处理，笔者认为再加入这个影响，读者理解起来更加困难，而且读者在理解三个参量的情况下，理解不同光照贴图类型带来的不同处理，会更加容易。因此，下文中以三参量模型来进行解释。

着色器数据选择

前文描述了巨大的着色器魔方。那么，如何根据当前阶段、当前材质类型、当前顶点工厂类型，从这个魔方中获得需要的着色器组合呢？

以一个静态网格物体渲染为例，对着色器数据选择的过程大体描述如下。请注意，以下描述的流程位于渲染线程中，对应F开头的类，而不是U开头的类。关于这两者区别的详细描述，可以参考下文关于场景代理的介绍。

1. 渲染线程遍历当前场景，添加静态网格到渲染列表。
2. 当一个静态网格FStaticMesh被添加到渲染列表时，其会根据自己参与的渲染阶段，通过F<渲染阶段（如BasePass)>DrawingPolicyFactory的AddStaticMesh函数开始选取当前渲染阶段对应的着色器。
3. 通过当前静态网格的MaterialRenderProxy材质渲染代理成员变量的GetMaterial函数，获得对应的FMaterial指针，该指针将会作为材质筛选的依据。随后调用Process<渲染阶段>Mesh，例如在BasePass调用的是ProcessBasePassMesh。
4. 该函数的最后一个参数会传入一个结构体，描述了一组渲染动作，其对应的Process函数会根据传入的光照贴图代理类型来做出不同的渲染方式，在此暂不分析光照贴图相关内容。其调用了AddMesh函数，在参数中创建了最重要的DrawingPolicy绘制代理。
5. 这个代理的类型即对应渲染阶段，传入参数包括顶点工厂指针和材质代理指针。至此，前文描述中的三个决定因素：渲染阶段、顶点工厂类型、材质类型均齐备，调用Get<渲染阶段>Shaders函数：
 - a. 该函数调用当前传入的材质类型的GetShader模板函数，从材质对象中抽取出对应的着色器，赋值给对应阶段的着色器变量，例如BasePass阶段。顶点工厂为VertexFactoryType时，抽

取渲染管线着色器的案例代码如下（LightMapPolicyType为光
照贴图类型，这里暂不讨论）：

```
VertexShader = Material.GetShader<TBasePassVS<  
    LightMapPolicyType, false>>(VertexFactoryType);  
PixelShader = Material.GetShader<TBasePassPS<  
    LightMapPolicyType, false>>(VertexFactoryType);
```

该类对应BasePassRendering.h头文件中的GetBasePassShaders函
数。

- b. 材质（FMaterial）的GetShader函数则首先以当前的顶点工厂类
型的id为索引，通过GetMeshShaderMap函数从
OrderedMeshShaderMaps成员变量中查询到对应顶点工厂类型的
MeshShaderMap。
- c. 随后，调用当前的MeshShaderMap的GetShader函数，以当前着
色器类型为参数，查询到实际对应的着色器。

总结一下：实质上获取一组着色器组合需要的三个变量：渲染阶
段、顶点工厂类型、材质类型。

渲染阶段 直接用代码区分，以模板形式完成定义，例如通过：

```
TBasePassPS<TUniformLightMapPolicy<Policy>, true>
```

定义。最终会产生出一个FMeshMaterialShaderType*类型的变量，
用于上文提到的MeshShaderMap->GetShader参数作为查询。

材质类型 获取于当前网格物体的Material参数。实质上拉取材质

就是从需要绘制的网格物体获取对应材质，再直接调用材质对应的GetShader函数完成。

顶点工厂类型 获取于当前网格物体的VertexFactory参数。传给上文提到的材质对应的GetShader作为参数。

11.4 场景代理**SceneProxy**

前文集中讲解渲染的整体过程，相当于以时间为维度来理解渲染过程。本节则更倾向于以对象为维度理解渲染过程。游戏中的对象如何最终变化为更适合的渲染数据，然后完成上文提到的渲染过程呢？

11.4.1 逻辑的世界与渲染的世界

虚幻引擎的框架设计的一个基本思路是，逻辑与渲染分离。即存在一个逻辑上的游戏世界，包含各种场景中的Actor，也包含Actor的组件。同时，也存在一个用于渲染的世界，这个世界中包含了呈现逻辑世界所需要的信息。渲染的世界如同布景一般，其只会呈现必要的内容——在当前摄像机范围内的，可以被渲染的内容。

这有一点类似MVC的思想，Model就是我们的逻辑世界，其包含的是逻辑信息。例如一个Static Mesh Actor，会包含位置、旋转、缩放，以及其对应的静态网格物体的引用。但是无论是Static Mesh Actor，还是这个Actor持有的Static Mesh Component组件，均不会去处理“渲染”相关的代码，而是提供一个用于渲染的“影子”，这个影子来完成具体的顶点构造、三角形填充等任务。那么，为何虚幻引擎采用这样的设计？让

我们先进行一番简单的思考，如果是我们来设计一个简单的、带有渲染功能的引擎，一种最简单的方式就是：设计一个Actor基类，带有两个虚函数Tick和Render。其中Tick负责每帧的逻辑更新，Render负责在渲染的时候绘制自身。

但是假如在多线程的情况下，这种设计存在一个明显的问题：有可能**Tick**函数执行到一半时，**Render**函数就被调用了，其最终渲染结果不固定。

即假如当前对象包含两个组件A和B，渲染A组件的时候，Tick函数尚未调用，然后Tick函数被调用，整个对象位置移动了，然后渲染B组件，此时B组件会在新的位置。呈现的效果就是：A、B组件被撕裂开了。

解决这样的问题，一种条件反射式的想法就是：在Tick函数和Render函数前后加锁，同一时间如果正在Tick，Render函数就会被阻塞。反之亦然。

这诚然是一种理想化的解决方案，但是实际生产中绝对不可能采用这种方案。这会导致Tick和Render被频繁阻塞。多线程的每一次阻塞都是有成本的。对于Tick这种高频调用的函数，任何一点时间成本，都会积累为一个巨大的延时。为了避免这样的情况，虚幻引擎设计了一种牺牲精确程度换取效率提升的思路，姑且把Tick函数所在的线程称为游戏线程，负责渲染的函数称为渲染线程，则：

1. 游戏线程遍历所有的逻辑对象，对于每个逻辑对象中可以显示的组件：
 - a. 创建一个SceneProxy场景代理，完成场景代理初始化。

- b. SceneProxy会根据对象位置等，更新自己用于渲染的信息。
2. 在合适的时机收集SceneProxy对象，并创建渲染指令，添加到渲染线程的渲染指令列表末尾。
3. 渲染线程不断从渲染指令列表中取出渲染指令进行渲染。

SceneProxy的世界，其实是逻辑世界的“残影”，具有一定的延迟。同时，SceneProxy的世界，也是逻辑世界的“具象化”，因为其把逻辑世界中的逻辑信息剥离（去除了具体是玩家还是怪物，是树还是花这样的信息），转化为能够被渲染的三角形、渲染参数等信息。而值得一提的是，这个世界的更新是依赖于当前View的，也就是观察视角。根据观察视角，有些逻辑对象将不会创建自己的渲染内容，同样地，根据在屏幕上的大小，逻辑对象可以在诸多LOD中进行选择，以降低渲染的压力。

而最重要的一点是，这样的设计去除了对锁的依赖。渲染线程只会参考SceneProxy进行渲染，而不会关注原始的对象的情况。两者不会产生前文中提到的，由于线程异步带来的同步问题。

11.4.2 渲染代理的创建

任何一个可以被渲染的组件，都需要创建一个对应的渲染代理。因此在UPrimitiveComponent类中，提供了如下虚函数：

```
virtual FPrimitiveSceneProxy* CreateSceneProxy()
```

这个函数将会在组件注册到世界中的时候被调用。而不是每帧都会调用。

在自己创建的组件类中重载此函数，即可返回自己的SceneProxy。而此时你拥有极大的自由度：你可以引入和提交Shader，你可以手动提交顶点缓冲区和索引缓冲区，你可以手动请求绘制等。这种自由度甚至高过了ProcedureMeshComponent，毕竟你可以引入自己的Shader。

11.4.3 渲染代理的更新

渲染代理的更新分为两个部分进行：一个是静态的部分，通过重载DrawStaticElements函数完成；另一个是动态的部分，通过重载GetDynamicMeshElements完成。其中静态部分的更新更加节省资源，动态部分则赋予了更高的自由度。

静态部分更新遵循以下的规则：

1. DrawStaticElements函数只会在场景收集的时候被调用，以收集静态物体。其通常无法在接下来的渲染过程中得到更新。
2. 当静态对象被移动、旋转或者缩放的时候，OnTransformChanged函数会被调用。同时DrawStaticElements函数会被重新调用以更新。

而相对应的，动态部分的更新则频繁的多：

1. GetDynamicMeshElements在每次场景开始渲染的时候就会由FDeferredShadingSceneRenderer通过FSceneRenderer调用。
2. GetDynamicMeshElements会根据当前视角角度，提交需要的Element。

不管是静态部分的更新，还是动态部分的更新，最终都不是直接对对象进行绘制，而是将Mesh信息输入到传入的FMeshElementCollector

中，最后会由渲染器统一进行渲染。而不是调用这两个函数进行渲染。

11.4.4 实战：创建新的渲染代理

如今我们已经有了充分的理论储备。那么接下来我们应该通过一个实战来检验所学到的知识。首先你需要创建一个继承自 UPrimitiveComponent的组件类用于测试。具体的创建过程，笔者就不再赘述，我这里创建了一个叫作UTestCustomComponent的组件。头文件 UTestCustomComponent.h代码如下：

```
#pragma once
#include "Engine.h"
#include "TestCustomComponent.generated.h"
UCLASS()
class UTestCustomComponent : public UPrimitiveComponent
{
GENERATED_BODY()
};
```

接下来我们就可以重载之前阐述过的CreateSceneProxy函数。同时我们也要顺便创建一个自己的SceneProxy类，用于返回。笔者在这里这样书写：

```
class FTestCustomComponentSceneProxy : public FPrimitiveSceneProxy
{
public:
```

```
FTestCustomComponentSceneProxy(UPrimitiveComponent* Component
)
: FPrimitiveSceneProxy(Component){}
virtual uint32 GetMemoryFootprint(void) const override
{
    return(sizeof(*this) + GetAllocatedSize());
}
virtual void GetDynamicMeshElements(const TArray<const
    FSceneView*>& Views, const FSceneViewFamily& ViewFamily,
    uint32 VisibilityMap , class FMeshElementCollector&
    Collector) const override
{
    FBox TestDynamicBox = FBox(FVector(-100.0f), FVector
        (100.0f));
    DrawWireBox(
        Collector.GetPDI(0),
        GetLocalToWorld(),
        TestDynamicBox ,
        FLinearColor::Red,
        ESceneDepthPriorityGroup::SDPG_Foreground ,
        10.0f);
}
virtual FPrimitiveViewRelevance GetViewRelevance(const
    FSceneView* View) const override
{
    FPrimitiveViewRelevance Result;
    Result.bDrawRelevance = IsShown(View);
```

```
    Result.bDynamicRelevance = true;
    Result.bShadowRelevance = IsShadowCast(View);
    Result.bEditorPrimitiveRelevance =
        UseEditorCompositing(View);
    return Result;
}
};
```

这是一个相当简单的SceneProxy类。尽管如此，依然包含了相当多的代码。总体来说，共有三个函数被创建了出来：

1. GetMemoryFootprint函数，这是用来跟踪分配内存的。
2. GetDynamicMeshElement函数，每次收集动态的元素都会被调用，这里笔者通过DrawBox来提交绘制一个立方体。
3. GetViewRelevance函数，这个函数根据传入View，判断当前对象是否可见。

最终，需要让我们新建的组件返回这个SceneProxy，方法就是重载CreateSceneProxy函数，代码如下：

```
FPrimitiveSceneProxy* UTestCustomComponent::CreateSceneProxy()
{
    return new FTestCustomComponentSceneProxy(this);
}
```

最终的头文件和.cpp文件如下：

```
#pragma once

#include "Engine.h"
#include "TestCustomComponent.generated.h"
UCLASS(meta = (BlueprintSpawnableComponent))
class UTestCustomComponent : public UPrimitiveComponent
{
    GENERATED_BODY()

public:
    virtual FPrimitiveSceneProxy* CreateSceneProxy() override;
};

#include "CustommadePrivatePCH.h"
#include "TestCustomComponent.h"

class FTestCustomComponentSceneProxy : public FPrimitiveSceneProxy
{
public:
    FTestCustomComponentSceneProxy(UPrimitiveComponent* Component
        :FPrimitiveSceneProxy(Component){}
    virtual uint32 GetMemoryFootprint(void) const override
    {
        return(sizeof(*this) + GetAllocatedSize());
    }
    virtual void GetDynamicMeshElements(const TArray<const FSceneView>
        Views, const FSceneViewFamily& ViewFamily, uint32 VisibilityM
        class FMeshElementCollector& Collector) const override
    {
        FBox TestDynamicBox = FBox(FVector(-100.0f), FVector(100.0f));
    }
};
```

```
    DrawWireBox(
```

```
        Collector.GetPDI(0),
```

```
        GetLocalToWorld(),
```

```
        TestDynamicBox ,
```

```
        FLinearColor::Red,
```

```
        ESceneDepthPriorityGroup::SDPG_Foreground ,
```

```
        10.0f);
```

```
}
```

```
virtual FPrimitiveViewRelevance GetViewRelevance(const FSceneView
```

```
        View) const override
```

```
{
```

```
    FPrimitiveViewRelevance Result;
```

```
    Result.bDrawRelevance = IsShown(View);
```

```
    Result.bDynamicRelevance = true;
```

```
    Result.bShadowRelevance = IsShadowCast(View);
```

```
    Result.bEditorPrimitiveRelevance = UseEditorCompositing(\
```

```
        ;
```

```
    return Result;
```

```
}
```

```
};
```

```
FPrimitiveSceneProxy* UTestCustomComponent::CreateSceneProxy()
```

```
{
```

```
    return new FTestCustomComponentSceneProxy(this);
```

```
}
```

如果你给一个Actor添加我们新创建的组件，然后运行游戏进行测

试，你会发现出现了一个红色的线框盒子。

回顾一下我们的代码，可以发现，现在我们已经能够借助DrawWiredBox系列函数，来提交绘制请求。类似地，我们也可以请求绘制线条。同时，也可以借助DynamicMeshBuilder，来实现构建动态的网格，并请求渲染。

当然，你肯定会说，我们绕了如此大的一个圈子，以实现虚幻引擎提供的ProceduralMeshComponent组件早已实现的功能，是否有一些杀鸡用牛刀的嫌疑？其实，此处描述的内容是为了印证之前对虚幻引擎场景代理机制的介绍和说明，为了方便读者更好理解。借助场景代理，能够实现的功能远比此处讲述的ProceduralMeshComponent组件能够实现的要多很多。那么接下来，我们就要超越ProceduralMeshComponent了。

11.4.5 进阶：创建静态渲染代理

ProceduralMeshComponent在虚幻引擎4.11的时代，是没有办法进行静态构建的，也许你阅读到这一段的时候，虚幻引擎已经支持静态的程序化模型（虚幻引擎4.13的特性，笔者撰写时4.13还没有发布），不过在笔者撰写本节时是不可行的。而我们可以通过重载DrawStaticElements函数来实现返回静态的模型。这个模型拥有一部分可变性——你可以在编辑器中调整参数，例如修改位置和缩放，然后模型会更新（其RenderProxy渲染代理会被标记为Dirty，从而请求更新）。但是在运行时，其只会在组件初始化的时候返回一次模型信息，之后就不会再返回，能够节省大量的性能。对于那些希望在编辑阶段修改，在运行时固定的模型，这个机制尤为合适。

而同样地，为了完成这样的功能，我们需要做的步骤就会多很多。大体来说，我们需要完成的步骤有以下：

1. 创建顶点缓冲区类和索引缓冲区类，如果你学过DirectX或者OpenGL，你对这个过程应该会非常熟悉。
2. 创建顶点工厂，顶点工厂可以统一不同类型的顶点信息，转变成可以被统一Shader使用的顶点。这一点笔者会在接下来说明。
3. 在RenderProxy的构造函数中：
 - a. 填充顶点缓冲区和索引缓冲区类中的数据（请注意，此时还没有绑定缓冲区到显存，这些数据依然在内存中）。
 - b. 初始化顶点工厂。
 - c. 请求在渲染线程中，完成对顶点、索引缓冲区和顶点工厂的初始化。
4. 重载RenderProxy的DrawStaticElements函数，创建一个Mesh并请求绘制。

这里的步骤相当的多。如果你对DirectX或者OpenGL的渲染过程比较熟悉，那么理解起来可能更加简单。如果你不是非常熟悉，建议你先阅读DirectX或者OpenGL渲染流程的相关知识。在此笔者只做简单的介绍。

我们首先需要在我们的.build.cs控制文件中，引入三个新的模块 RenderCore，ShaderCore和RHI：

```
PrivateDependencyModuleNames.AddRange(new string[]{  
    "CoreUObject", "Engine", "Slate", "SlateCore", "InputCore",  
    "RenderCore", "ShaderCore", "RHI"
```

```
});
```

这是笔者在测试项目中使用的模块，你可以直接使用这行。

随后我们就可以继续在TestCustomComponent.cpp文件中，书写我们的代码了。首先需要定义我们自己的VertexBuffer和IndexBuffe。理论上说，甚至可以定义自己的顶点类型，这在虚幻引擎中当然是支持的。不过为了简化这部分的复杂度，可以使用虚幻引擎为我们预定义 FDynamicMeshVertex。顺便一提，如果希望自行定义顶点结构，Shader也需要酌情修改。

```
/** Vertex Buffer */

class FTestCustomComponentVertexBuffer : public FVertexBuffer
{
public:
    TArray<FDynamicMeshVertex> Vertices;
    virtual void InitRHI() override
    {
        FRHIResourceCreateInfo CreateInfo;
        void* VertexBufferData = nullptr;
        VertexBufferRHI = RHICreateAndLockVertexBuffer(
            Vertices.Num() * sizeof(FDynamicMeshVertex),
            BUF_Static, CreateInfo, VertexBufferData);
        FMemory::Memcpy(VertexBufferData, Vertices.GetData(),
            Vertices.Num() * sizeof(FDynamicMeshVertex));
        RHIUnlockVertexBuffer(VertexBufferRHI);
    }
}
```

```
};

/** Index Buffer */

class FTestCustomComponentIndexBuffer : public FIndexBuffer
{
public:
    TArray<int32> Indices;
    virtual void InitRHI() override
    {
        FRHISourceCreateInfo CreateInfo;
        IndexBufferRHI = RHICreateIndexBuffer(sizeof(int32),
            Indices.Num() * sizeof(int32), BUF_Static,
            CreateInfo);
        void* Buffer = RHILockIndexBuffer(IndexBufferRHI, 0,
            Indices.Num() * sizeof(int32), RLM_WriteOnly);
        FMemory::Memcpy(Buffer, Indices.GetData(), Indices.
            Num() * sizeof(int32));
        RHIFreeIndexBuffer(IndexBufferRHI);
    }
};
```

每个缓冲区都包含了一个数据数组——位于内存中，用于填充顶点和索引数据，以及一个InitRHI函数。通过重载该函数，就可以在显存中创建顶点和索引缓冲区。随后通过锁存、内存拷贝、取消锁定等操作，上传我们的顶点和索引数据到显存中。

在定义完毕顶点和索引缓冲区后，我们还需要定义一个相对来说比较特殊的结构，那就是FVertexFactory（顶点工厂）。顶点工厂这个概

念是虚幻引擎创造的，而不是DirectX或者OpenGL的概念。顶点工厂是为了抽象不同类型的顶点数据，转化为统一的三角面数据。举个例子，对于带有骨骼的模型来说，虚幻引擎提供了GPUSkinnedVertexFactory系列类，其根据原始的模型和骨骼数据在GPU端借助Shader完成蒙皮变形的操作，就是根据骨骼位置计算出最终的顶点。在这之后，骨骼模型与普通的静态网格物体已经没有什么太大的区别，可以执行同样的渲染管线。

在这个案例中，我们来实现一个最为简单的顶点工厂，其只是简单地定义了顶点数据的格式。代码如下：

```
/** Vertex Factory */
class FTestCustomComponentVertexFactory : public FLocalVertexFactory
{
public:
    FTestCustomComponentVertexFactory()
    {}

    /** Initialization */
    void Init(const FTestCustomComponentVertexBuffer*
        VertexBuffer)
    {
        if (IsInRenderingThread())
        {
            // Initialize the vertex factory's stream
            components.

            DataType NewData;
            NewData.PositionComponent =

```

```

STRUCTMEMBER_VERTEXSTREAMCOMPONENT(
    VertexBuffer , FDynamicMeshVertex , Position
    , VET_Float3);

NewData.TextureCoordinates.Add(
    FVertexStreamComponent(VertexBuffer ,
    STRUCT_OFFSET(FDynamicMeshVertex ,
    TextureCoordinate), sizeof(
    FDynamicMeshVertex), VET_Float2)
);

NewData.TangentBasisComponents[0] =
    STRUCTMEMBER_VERTEXSTREAMCOMPONENT(
        VertexBuffer , FDynamicMeshVertex , TangentX
        , VET_PackedNormal);

NewData.TangentBasisComponents[1] =
    STRUCTMEMBER_VERTEXSTREAMCOMPONENT(
        VertexBuffer , FDynamicMeshVertex , TangentZ
        , VET_PackedNormal);

SetData(NewData);
}

else
{
    ENQUEUE_UNIQUE_RENDER_COMMAND_TWOPARAMETER(
        InitTestCustomComponentVertexFactor ,
        FTestCustomComponentVertexFactory*,
        VertexFactory , this,
        const FTestCustomComponentVertexBuffer*,
        VertexBuffer , VertexBuffer ,

```

```
{  
    // Initialize the vertex factory's  
    // stream components.  
  
    DataType NewData;  
  
    NewData.PositionComponent =  
        STRUCTMEMBER_VERTEXSTREAMCOMPONENT  
        (VertexBuffer , FDynamicMeshVertex ,  
         Position, VET_Float3);  
  
    NewData.TextureCoordinates.Add(  
        FVertexStreamComponent(  
            VertexBuffer,  
            STRUCT_OFFSET(  
                FDynamicMeshVertex,  
                TextureCoordinate), sizeof  
                (FDynamicMeshVertex),  
                VET_Float2)  
    );  
  
    NewData.TangentBasisComponents[0] =  
        STRUCTMEMBER_VERTEXSTREAMCOMPONENT  
        (VertexBuffer, FDynamicMeshVertex,  
         TangentX, VET_PackedNormal);  
  
    NewData.TangentBasisComponents[1] =  
        STRUCTMEMBER_VERTEXSTREAMCOMPONENT  
        (VertexBuffer, FDynamicMeshVertex,  
         TangentZ, VET_PackedNormal);  
  
    VertexFactory->SetData(NewData);  
};
```

```
    }  
}  
};
```

相对来说，这一段代码的量会比较长，仔细观察会发现代码有一段重复，为何同样的代码要写两遍呢？那是因为，我们必须要确保 我们的顶点工厂初始化在渲染线程中执行。也就是说，我们会首先判定当前是不是渲染线程，如果是，就会直接执行初始化代码；否则就需要借助 ENQUEUE UNIQUE RENDER COMMAND XXXPARAMETER 系列宏去请求在渲染线程中执行当前代码。如果对这个机制不大明白，可以再回顾一下之前阐述多线程渲染架构的内容。

那么这一段初始化代码的具体含义是什么呢？我们都知道，虚幻引擎的渲染架构实际上是 DirectX 和 OpenGL 的一个统一抽象，也就是说，这段代码一定能在 DirectX 或者 OpenGL 中找到对应的概念操作。DirectX11 中有 Layer 的概念：对上传的每个顶点数据，需要手动指定其如何“切分”为具体的顶点属性。由于允许我们自己定义顶点数据，因此我们需要手动地把上传到显存中的顶点数据切分成具体的属性（或者说成员变量），然后传递给着色器。这个过程，实质上是制定每个属性到当前顶点数据开头的“偏移”与“长度”。这就像我们来到陌生的街上，有一排店面，需要知道这条街上店面的归属情况。为了达到这样的目的，由于店主的店面都连接在一起，我们只需要收集每个店主第一家店面的门牌号，以及店面的数量，就可以把整条街的店面切分为每个店主自己所属的段。

同样地，这一段代码也是一种切分的过程。其实际上是将 VertexBuffer 中的数据切分之后，指定给对应的 DataType 中对应的字段。

这个DataType已经定义好了一系列的数据，包括位置、贴图UV、法线。由于我们继承于LocalVertexFactory，因此我们只需要指定局部空间的位置，从本地空间到世界空间的转换将会由其自动完成。笔者会在接下来对调用的讲解中，进一步讲解这个转换过程。

为了把我们刚刚定义的顶点缓冲区、索引缓冲区和顶点工厂引入到SceneProxy中，我们需要增加以下三个成员变量：

```
FTestCustomComponentIndexBuffer     IndexBuffer;
FTestCustomComponentVertexBuffer   VertexBuffer;
FTestCustomComponentVertexFactory VertexFactory;
```

而为了方便测试，也需要修改我们的构造函数，并增加一些测试性顶点来方便显示。新的构造函数如下：

```
FTestCustomComponentSceneProxy(UPrimitiveComponent* Component)
: FPrimitiveSceneProxy(Component)
{
    const float BoxSize = 100.0f;
    //填充顶点
    VertexBuffer.Vertices.Add(FVector(0.0f));
    VertexBuffer.Vertices.Add(FVector(BoxSize, 0.0f, 0.0f));
    VertexBuffer.Vertices.Add(FVector(0.0f, BoxSize, 0.0f));
    VertexBuffer.Vertices.Add(FVector(0.0f, 0.0f, BoxSize));
    //填充索引
    IndexBuffer.Indices.Add(0);
```

```
IndexBuffer.Indices.Add(1);
IndexBuffer.Indices.Add(2);

IndexBuffer.Indices.Add(0);
IndexBuffer.Indices.Add(2);
IndexBuffer.Indices.Add(3);

IndexBuffer.Indices.Add(0);
IndexBuffer.Indices.Add(3);
IndexBuffer.Indices.Add(1);

IndexBuffer.Indices.Add(3);
IndexBuffer.Indices.Add(2);
IndexBuffer.Indices.Add(1);
//初始化
VertexFactory.Init(&VertexBuffer);
BeginInitResource(&IndexBuffer);
BeginInitResource(&VertexBuffer);
BeginInitResource(&VertexFactory);
}
```

这里增加了4个顶点与12个索引数据，最终构造了一个三棱锥。你也可以自己修改为一个立方体或者球体。在把这些数据填充到缓冲区之后，我们需要用顶点缓冲区初始化顶点工厂。这个初始化是一个类似“预初始化”的过程。真正完成显卡关联的数据初始化，是使用BeginInitResource函数完成的。BeginInitResource函数内部也是借助

EQUEUE_RENDER_COMMAND系列宏向渲染线程请求初始化。务必注意的是，顶点工厂初始化有两次：一次是CPU端数据的初始化，随后在渲染线程还有一次资源初始化。

相对应地，我们申请了资源，就务必需要手动释放，这很类似COM资源的申请与释放。我们可以在析构函数中完成这个过程，如下：

```
virtual ~FTestCustomComponentSceneProxy()
{
    VertexBuffer.ReleaseResource();
    IndexBuffer.ReleaseResource();
    VertexFactory.ReleaseResource();
}
```

同样地，我们也需要在GetViewRelevance函数中简单地说明当前网格可见：

```
virtual FPrimitiveViewRelevance GetViewRelevance(const FSceneView
                                                 View) const override
{
    FPrimitiveViewRelevance Result;
    Result.bStaticRelevance = true;
    Result.bDrawRelevance = true;
    Result.bDynamicRelevance = true;
    return Result;
```

```
}
```

到这里终于可以开始完成最终的渲染绘制调用了。我们撰写了这么多的代码，终于到了展现努力结果的时候了。让我们完成DrawStaticElement函数吧。

```
virtual void DrawStaticElements(FStaticPrimitiveDrawInterface* PD)
{
    FMeshBatch Mesh;
    FMeshBatchElement& BatchElement = Mesh.Elements[0];
    BatchElement.IndexBuffer = &IndexBuffer;
    Mesh.bWireframe = false;
    Mesh.VertexFactory = &VertexFactory;
    Mesh.MaterialRenderProxy = UMaterial::GetDefaultMaterial(
        MD_Surface)->GetRenderProxy(false);
    BatchElement.PrimitiveUniformBuffer =
        CreatePrimitiveUniformBufferImmediate(GetLocalToWorld(),
        GetBounds(), GetLocalBounds(), true, UseEditorDepthTest());
    ;
    BatchElement.FirstIndex = 0;
    BatchElement.NumPrimitives = IndexBuffer.Indices.Num() / 3;
    BatchElement.MinVertexIndex = 0;
    BatchElement.MaxVertexIndex = VertexBuffer.Vertices.Num() -
        1;
    Mesh.ReverseCulling = IsLocalToWorldDeterminantNegative();
```

```
    Mesh.Type = PT_TriangleList;
    Mesh.DepthPriorityGroup = SDPG_Foreground;
    Mesh.bCanApplyViewModeOverrides = false;
    Mesh.bDisableBackfaceCulling = false;
    PDI->DrawMesh(Mesh, 1.0f);
}
```

这里需要介绍两个概念：MeshBatch和BatchElement。MeshBatch是一系列的网格的集合，而BatchElement则是单个的网格元素。这似乎还是很难理解，那么，一个简单的例子是，MeshBatch存储了整个顶点缓冲区（即指定了当前的顶点工厂），而每个单独的BatchElement都有自己的索引缓冲区。当然，它们也有更多的区别，具体的内容繁多，在此不详细讲解。

在代码中，有一个比较重要的代码是对PrimitiveUniformBuffer的填充，这是通过CreatePrimitiveUniformBufferImmediat函数完成的。其中就指定了从本地空间到世界空间的变换矩阵。

在填充了全部的数据之后，通过PDI的DrawMesh函数，渲染整个静态网格。

最终的代码如下：

```
#include "CustommadePrivatePCH.h"
#include "TestCustomComponent.h"
#include "DynamicMeshBuilder.h"
/** Vertex Buffer */
```

```
class FTestCustomComponentVertexBuffer : public FVertexBuffer
{
public:
    TArray<FDynamicMeshVertex> Vertices;

    virtual void InitRHI() override
    {
        FRHISourceCreateInfoCreateInfo;
        void* VertexBufferData = nullptr;
        VertexBufferRHI = RHICreateAndLockVertexBuffer(Vertices.Num(),
            sizeof(FDynamicMeshVertex), BUF_Static, CreateInfo,
            VertexBufferData);
        FMemory::Memcpy(VertexBufferData, Vertices.GetData(), Vertices.Num() *
            sizeof(FDynamicMeshVertex));
        RHILockVertexBuffer(VertexBufferRHI);
    }
};

/** Index Buffer */
class FTestCustomComponentIndexBuffer : public FIndexBuffer
{
public:
    TArray<int32> Indices;

    virtual void InitRHI() override
    {
        FRHISourceCreateInfoCreateInfo;
        IndexBufferRHI = RHICreateIndexBuffer(sizeof(int32), Indices.Num());
    }
};
```

```
        sizeof(int32), BUF_Static, CreateInfo);

    void* Buffer = RHILockIndexBuffer(IndexBufferRHI , 0, Indices.Num(),
        sizeof(int32), RLM_WriteOnly);

    FMemory::Memcpy(Buffer, Indices.GetData(), Indices.Num()
        * sizeof(int32));

    RHILockIndexBuffer(IndexBufferRHI);

}

};

/** Vertex Factory */

class FTestCustomComponentVertexFactory : public FLocalVertexFactory
{
public:
    FTestCustomComponentVertexFactory()
    {}

    /** Initialization */
    void Init(const FTestCustomComponentVertexBuffer* VertexBuffer)
    {
        if (IsInRenderingThread())
        {
            // Initialize the vertex factory's stream components.

            DataType NewData;

            NewData.PositionComponent = STRUCTMEMBER_VERTEXSTREAMCOMPONENT(
                VertexBuffer , FDynamicMeshVertex , Position, VET_Float3);

            NewData.TextureCoordinates.Add(
                FVertexStreamComponent(VertexBuffer , STRUCT_OFFSET(FDynamicMeshVertex,
                    TextureCoordinate), sizeof(FDynamicMeshVertex), VET_Float3));
        }
    }
};
```

```

NewData.TangentBasisComponents[0] =
    STRUCTMEMBER_VERTEXSTREAMCOMPONENT(VertexBuffer ,
FDynamicMeshVertex , TangentX, VET_PackedNormal);
NewData.TangentBasisComponents[1] =
    STRUCTMEMBER_VERTEXSTREAMCOMPONENT(VertexBuffer ,
FDynamicMeshVertex , TangentZ, VET_PackedNormal);
SetData(NewData);

}

else
{
    ENQUEUE_UNIQUE_RENDER_COMMAND_TWOPARAMETER(
        InitTestCustomComponentVertexFactor ,
        FTestCustomComponentVertexFactory*, VertexFactory , t
    const FTestCustomComponentVertexBuffer*, VertexBuffer
    {

        // Initialize the vertex factory's stream component
        DataType NewData;
        NewData.PositionComponent = STRUCTMEMBER_VERTEXSTREAMCOMPONENT(
            VertexBuffer , FDynamicMeshVertex , Position,
            NewData.TextureCoordinates.Add(
                FVertexStreamComponent(VertexBuffer , STRUCT_OFFSET(
                    , TextureCoordinate), sizeof(FDynamicMeshVertex),
                    );
        NewData.TangentBasisComponents[0] =
            STRUCTMEMBER_VERTEXSTREAMCOMPONENT(VertexBuffer ,
FDynamicMeshVertex , TangentX, VET_PackedNormal);
        NewData.TangentBasisComponents[1] =

```

```
STRUCTMEMBER_VERTEXSTREAMCOMPONENT(VertexBuff
    FDynamicMeshVertex , TangentZ, VET_PackedNorm
    VertexFactory ->SetData(NewData);

});

}

};

class FTestCustomComponentSceneProxy :public FPrimitiveSceneProxy
{
public:
    FTestCustomComponentIndexBuffer     IndexBuffer;
    FTestCustomComponentVertexBuffer    VertexBuffer;
    FTestCustomComponentVertexFactory   VertexFactory;

public:
    FTestCustomComponentSceneProxy(UPrimitiveComponent* Component
        :FPrimitiveSceneProxy(Component)
    {

        const float BoxSize = 100.0f;
        //填充顶点
        VertexBuffer.Vertices.Add(FVector(0.0f));
        VertexBuffer.Vertices.Add(FVector(BoxSize, 0.0f, 0.0f));
        VertexBuffer.Vertices.Add(FVector(0.0f, BoxSize, 0.0f));
        VertexBuffer.Vertices.Add(FVector(0.0f, 0.0f, BoxSize));
        //填充索引
        IndexBuffer.Indices.Add(0);
        IndexBuffer.Indices.Add(1);
        IndexBuffer.Indices.Add(2);
    }
}
```

```
    IndexBuffer.Indices.Add(0);
    IndexBuffer.Indices.Add(2);
    IndexBuffer.Indices.Add(3);

    IndexBuffer.Indices.Add(0);
    IndexBuffer.Indices.Add(3);
    IndexBuffer.Indices.Add(1);

    IndexBuffer.Indices.Add(3);
    IndexBuffer.Indices.Add(2);
    IndexBuffer.Indices.Add(1);
    //初始化
    VertexFactory.Init(&VertexBuffer);
    BeginInitResource(&IndexBuffer);
    BeginInitResource(&VertexBuffer);
    BeginInitResource(&VertexFactory);
}

virtual ~FTestCustomComponentSceneProxy()
{
    VertexBuffer.ReleaseResource();
    IndexBuffer.ReleaseResource();
    VertexFactory.ReleaseResource();
}

virtual uint32 GetMemoryFootprint(void) const override
{
    return(sizeof(*this) + GetAllocatedSize());
}
```

```
}

virtual void GetDynamicMeshElements(const TArray<const FSceneView>& Views, const FSceneViewFamily& ViewFamily, uint32 Visibility, class FMeshElementCollector& Collector) const override

{

    FBox TestDynamicBox = FBox(FVector(-100.0f), FVector(100.0f));

    DrawWireBox(
        Collector.GetPDI(0),
        GetLocalToWorld(),
        TestDynamicBox ,
        FLinearColor::Red,
        ESceneDepthPriorityGroup::SDPG_Foreground ,
        10.0f);

}

virtual FPrimitiveViewRelevance GetViewRelevance(const FSceneView& View) const override

{

    FPrimitiveViewRelevance Result;

    Result.bStaticRelevance = true;

    Result.bDrawRelevance = true;

    Result.bDynamicRelevance = true;

    return Result;

}

virtual void DrawStaticElements(FStaticPrimitiveDrawInterface& Interface) const override

{
```

```

        FMeshBatch Mesh;

        FMeshBatchElement& BatchElement = Mesh.Elements[0];

        BatchElement.IndexBuffer = &IndexBuffer;

        Mesh.bWireframe = false;

        Mesh.VertexFactory = &VertexFactory;

        Mesh.MaterialRenderProxy = UMaterial::GetDefaultMaterial(
            ->GetRenderProxy(false));

        BatchElement.PrimitiveUniformBuffer =
            CreatePrimitiveUniformBufferImmediate(GetLocalToWorld(
                ), GetLocalBounds(), true, UseEditorDepthTest()));

        BatchElement.FirstIndex = 0;

        BatchElement.NumPrimitives = IndexBuffer.Indices.Num() / 3;

        BatchElement.MinVertexIndex = 0;

        BatchElement.MaxVertexIndex = VertexBuffer.Vertices.Num();

        Mesh.ReverseCulling = IsLocalToWorldDeterminantNegative();

        Mesh.Type = PT_TriangleList;

        Mesh.DepthPriorityGroup = SDPG_Foreground;

        Mesh.bCanApplyViewModeOverrides = false;

        Mesh.bDisableBackfaceCulling = false;

        PDI->DrawMesh(Mesh, 1.0f);

    }

};

FPrimitiveSceneProxy* UTestCustomComponent::CreateSceneProxy()
{
    return new FTestCustomComponentSceneProxy(this);
}

```

给一个Actor添加你刚刚写的TestComponent组件。你会发现一个红色的线框立方体与一个灰色的三棱锥。外侧的是动态的模型，每一帧都会通过GetDynamicMeshElements获取；内侧的是静态的模型，只会在初始化的时候获取一次。



静态渲染代理

在笔者写作时虚幻引擎的最新版本（4.14版本）中，已经支持Procedural Mesh Component静态化。故建议读者在理解原理的基础上，尽可能使用虚幻引擎官方提供的接口。如果读者希望使用笔者提供的代码，请注意在承载渲染代理的组件类中，需要重载CalcBounds函数，返回当前组件的包围盒，否则静态渲染结果会被剔除，从而无法被看见。

11.4.6 静态网格物体渲染代理排序

静态网格物体元素列表如果直接遍历列表渲染，会反复设置状态，非常缓慢，如图11-6所示。如果按公用状态归并为链表，可以加快速度，如图11-7所示。

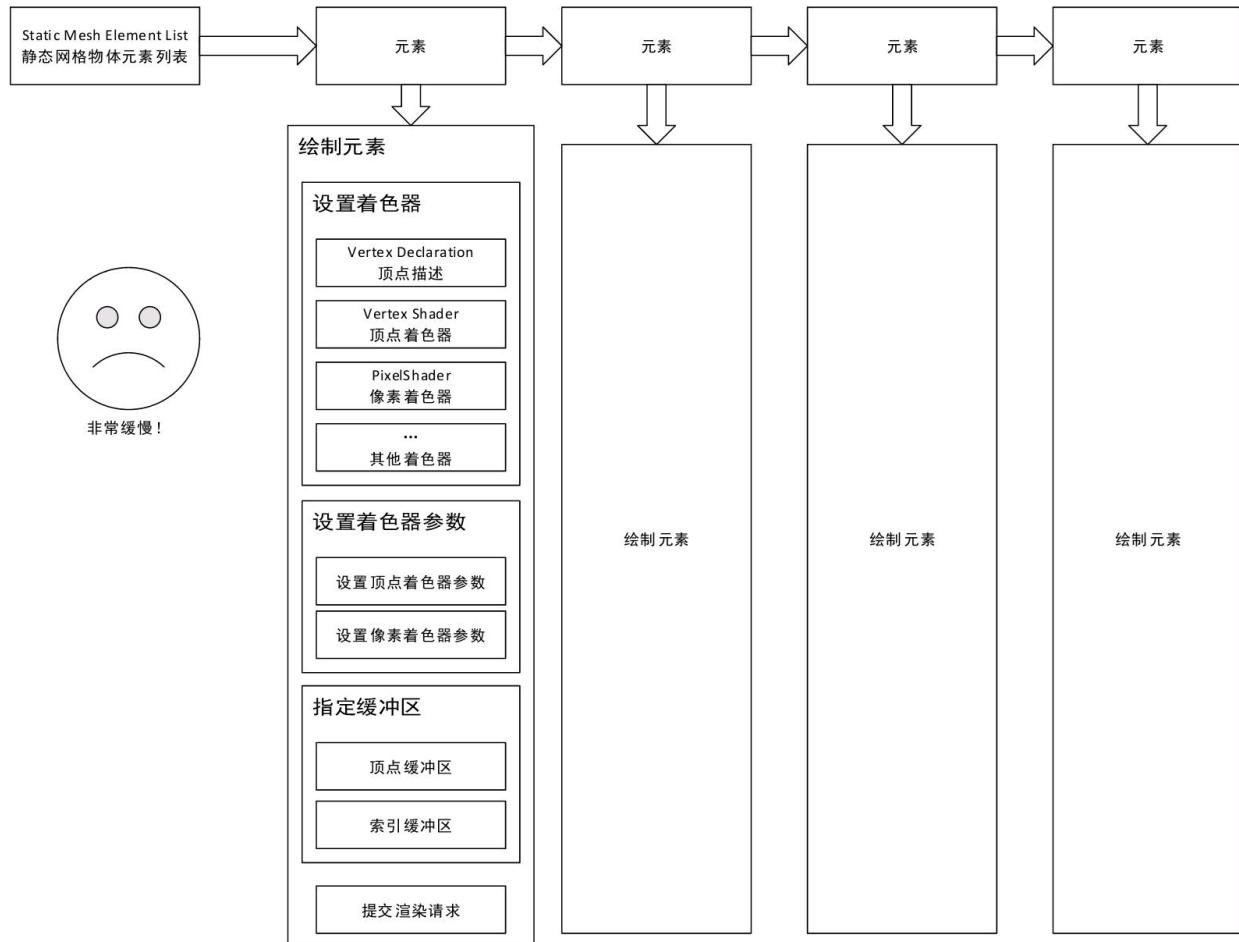


图11-6 直接遍历列表渲染：反复设置状态，非常缓慢

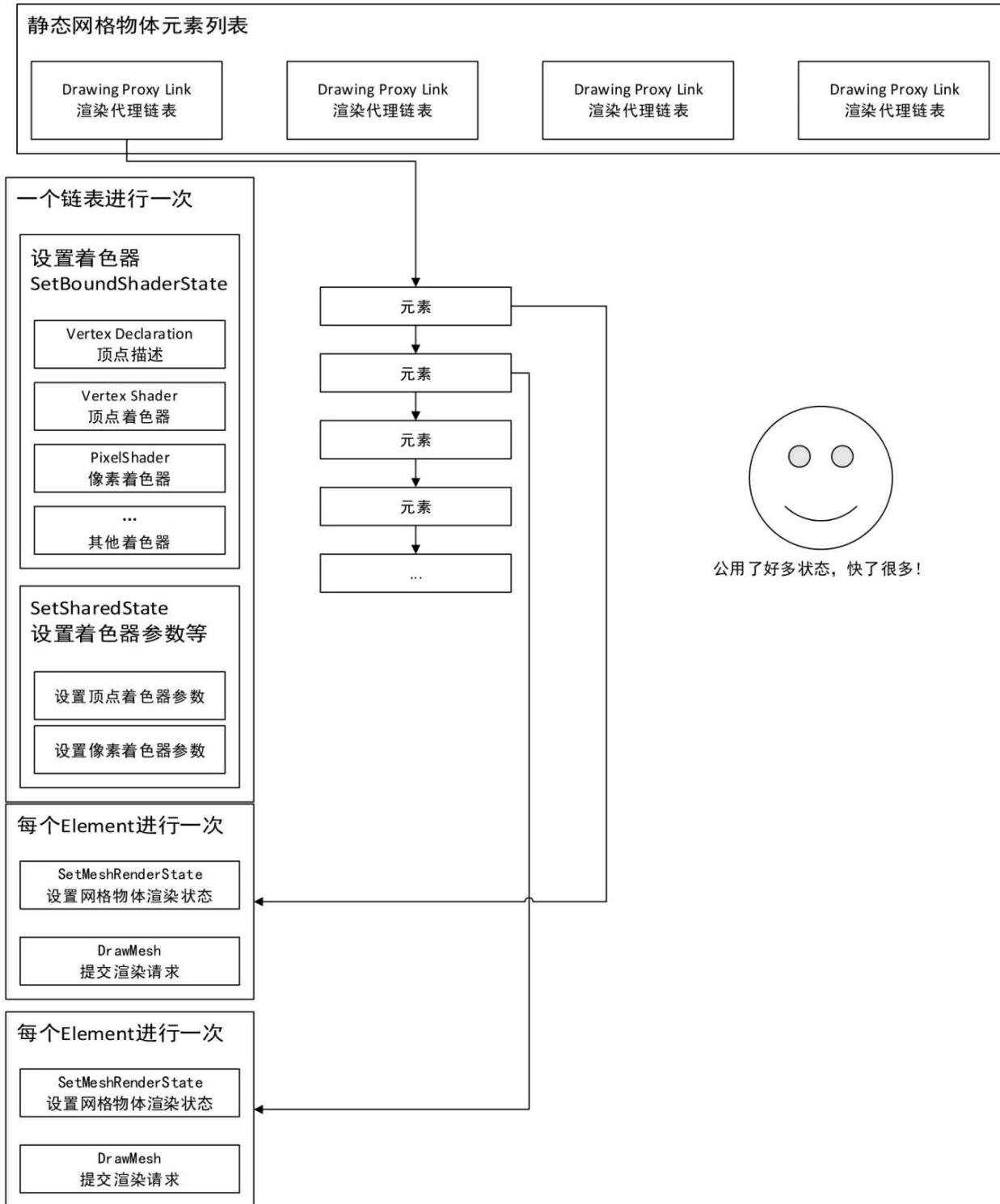


图11-7 按公用状态归并为链表，加快速度

DrawStaticElements函数只会在场景收集的时候被调用。这个收集

过程不仅仅是将静态渲染代理填充到一个列表里面完事，顶点缓冲区而是将这些渲染代理填充到多个渲染列表中，这些渲染列表以渲染状态分割，以便尽可能公用渲染状态。

当一个新的静态元素被收集的时候，会调用当前静态网格物体绘制列表的索引缓冲区AddMesh函数，其会查找当前TStaticMeshDrawList的DrawingPolicySet，查看是否有一致的Policy，如果遇到匹配的，就会挂接在对应的列表下，否则就会新建一个列表，然后加入。

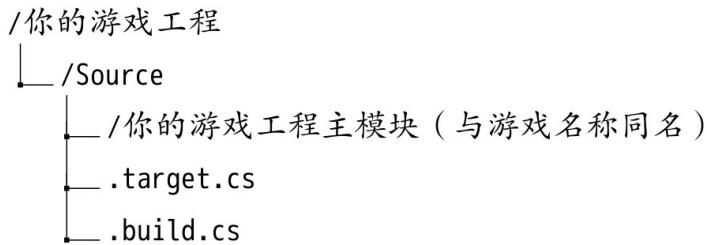
那么如何判断是否一致呢？答案在对应的Policy的Matches和Compare函数，每个类型都不一样，如果读者希望深入学习，可以亲自去看一下。

11.5 Shader

11.5.1 测试工程

由于Shader的载入是一个非常特殊的时机，因此，不能直接使用传统的工程文件新建流程。而是需要对你的游戏模块的加载时机进行设定。因此，请按照笔者说的方式来创建这个工程。

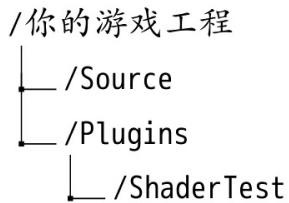
首先请创建一个标准的C++工程。这时候你的工程结构实质上应该是：



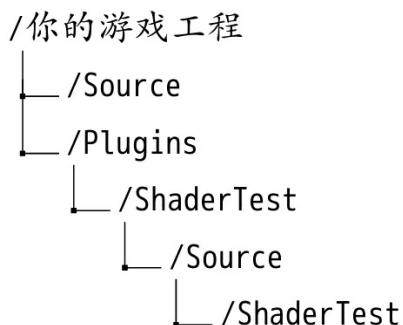
这时候你的游戏工程主模块的加载顺序是“Default”。这会导致我们的Shader无法加载。因此，我们必须要新建一个在PostConfigIni阶段加载的模块。

但是由于游戏模块的PostConfigIni模块是无效的（笔者测试的结果如此），所以我们必须把这个测试模块放在一个插件中。

请在工程目录下建立一个Plugins文件夹，然后在文件夹中添加一个ShaderTest文件夹。这个ShaderTest文件夹中将会放我们的插件代码。现在工程目录如下：



在ShaderTest文件夹中增加一个Source文件夹，然后在Source文件夹中添加一个ShaderTest文件夹，这个是我们插件的主模块。于是我们的文件夹结构会是：

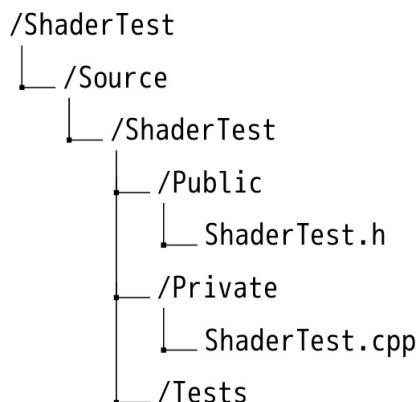


然后在模块的ShaderTest下放置一个ShaderTest.build.cs文件。这个文件将会定义这个模块的依赖模块等。具体内容就是：

```
using UnrealBuildTool;
public class ShaderTest : ModuleRules
{
public ShaderTest(TargetInfo Target)
{
PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "RHI", "RenderCore", "ShaderCore" });
}
}
```

这里面定义了这个模块依赖的其他模块。这里我们主要依赖RHI, RenderCore和ShaderCore模块。

然后我们需要添加Public和Private文件夹。并添加ShaderTest模块的实现：在Public中放置ShaderTest.h，在Private中放置ShaderTest.cpp。也就是说，我们现在的文件目录结构变为了：



而ShaderTest.h和ShaderTest.cpp里面依然是一些常规的“模块实现”代码。笔者的代码是这样的：

ShaderTest.h

```
#include "Engine.h"
```

ShaderTest.cpp

```
#include "ShaderTest.h"  
IMPLEMENT_PRIMARY_GAME_MODULE( FDefaultGameModuleImpl , ShaderTest  
    ShaderTest" );
```

这些代码足够完成将一个模块定义的任务。最后还需要做一件事来引用我们的模块：用记事本编辑工程文件uproot，在“Modules”中增加对ShaderTest模块的引用：

```
{  
    "Name": "ShaderTest",  
    "Type": "Runtime",  
    "LoadingPhase": "PostConfigInit"  
}
```

记得加逗号。此时如果你右键单击uproot,生成VS工程文件，那么会看到引用了我们的新模块。

11.5.2 定义Shader

此时我们就需要随便定义一个Shader了。出于快速测试的考虑，笔者提供了一个快速的测试：

```
#include "Common.usf"

void MainVertexShader(
    float4 InPosition : ATTRIBUTE0,
    float2 InUV : ATTRIBUTE1,
    out float2 OutUV : TEXCOORD0,
    out float4 OutPosition : SV_POSITION
)
{
    OutPosition = InPosition;
    OutUV = InUV;
}

void MainPixelShader(
    in float2 uv : TEXCOORD0,
    out float4 OutColor : SV_Target0
)
{
    OutColor = PSConstants.StartColor;
}
```

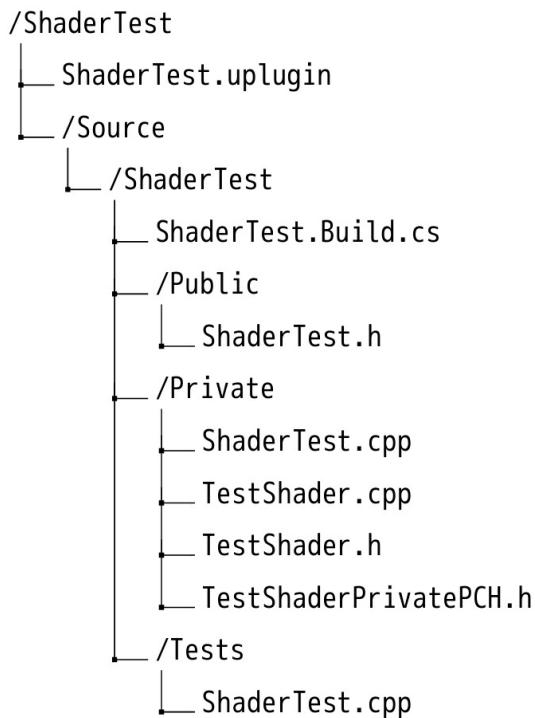
这个Shader的意思就是： StartColor提供的是多少，就会把贴图设置成和StartColor完全一致的颜色。然后把这个Shader放在虚幻引擎的

Shader目录下（以后你可以自己写代码，让模块载入的时候自动把Shader复制过去）。

11.5.3 定义**Shader**对应的C++类

此时我们已经拥有了一个Shader，但是需要让虚幻引擎能够识别、载入这个Shader，所以我们需要提供一个C++类来完成这样的工作。因此，我们需要在模块的Private目录下，添加一个.h、.cpp:TestShader.h和TestShader.cpp。我们还需要添加PrivatePCH.h这个预编译头文件，同时还需要在Private文件夹中添加Test文件夹。我们将会借助单元测试来快速测试我们的Shader。如果你对这个过程不太熟悉，可以阅读前文中对多线程编程描述里面的单元测试方式。

总而言之，目前的目录结构应该是这样：



那么接下来就是要填充TestShaderPrivatePCH.h和TestShader.h的内容了。笔者先给出它们的内容，然后再讲述这些内容的具体含义。

TestShaderPrivatePCH.h

```
#pragma once
#include "Engine.h"
#include "RHIStrictStates.h"
```

TestShader.h

```
#pragma once
#include "GlobalShader.h"
#include "UniformBuffer.h"
#include "RHICmdList.h"

//`定义我们自己的StartColor。需要和Shader匹配`
BEGIN_UNIFORM_BUFFER_STRUCT(FPixelShaderConstantParameters , )
DECLARE_UNIFORM_BUFFER_STRUCT_MEMBER(FVector4, StartColor)
END_UNIFORM_BUFFER_STRUCT(FPixelShaderConstantParameters)

typedef TUniformBufferRef <FPixelShaderConstantParameters >
    FPixelShaderConstantParametersRef;

//`定义我们自己的顶点结构`
struct FTextureVertex
{
    FVector4 Position;
    FVector2D UV;
```

```
};

//`添加我们自己的顶点结构描述`  
class FTextureVertexDeclaration : public FRenderResource  
{  
public:  
    FVertexDeclarationRHIF Ref VertexDeclarationRHI;  
    virtual void InitRHI() override  
    {  
        FVertexDeclarationElementList Elements;  
        uint32 Stride = sizeof(FTextureVertex);  
        Elements.Add(FVertexElement(0, STRUCT_OFFSET(FTextureVertex , Pos), VET_Float4, 0, Stride));  
        Elements.Add(FVertexElement(0, STRUCT_OFFSET(FTextureVertex , UV), VET_Float2, 1, Stride));  
        VertexDeclarationRHI = RHICreateVertexDeclaration(Elements);  
    }  
    virtual void ReleaseRHI() override  
    {  
        VertexDeclarationRHI.SafeRelease();  
    }  
};  
//`测试Shader的顶点着色器`  
class FTestShaderVertexShaderExample : public FGlobalShader  
{  
DECLARE_SHADER_TYPE(FTestShaderVertexShaderExample , Global);  
public:  
    static bool ShouldCache(EShaderPlatform Platform) { return true;
```

```
FTestShaderVertexShaderExample(const ShaderMetaType::
    CompiledShaderInitializerType& Initializer) :
FGlobalShader(Initializer)
{}

FTestShaderVertexShaderExample() {}

};

//`像素着色器`  

class FTestShaderPixelShaderDeclaration : public FGlobalShader
{
DECLARE_SHADER_TYPE(FTestShaderPixelShaderDeclaration , Global);
public:
FTestShaderPixelShaderDeclaration() {}
explicit FTestShaderPixelShaderDeclaration(const ShaderMetaType::
    CompiledShaderInitializerType& Initializer);

static bool ShouldCache(EShaderPlatform Platform) { return
    IsFeatureLevelSupported(Platform, ERHIFeatureLevel::SM5); }

virtual bool Serialize(FArchive& Ar) override
{
    bool bShaderHasOutdatedParams = FGlobalShader::Serialize(Ar);
    Ar << TextureParameter;
    return bShaderHasOutdatedParams;
}

//This function is required to let us bind our runtime surface to
//the shader using an SRV.
void SetSurfaces(FRHICmdList& RHICmdList,
    FShaderResourceViewRHIRef TextureParameterSRV);
//This function is required to bind our constant / uniform buffer
```

```
the shader.

void SetUniformBuffers(FRHICmdList& RHICmdList,
                      FPixelShaderConstantParameters& ConstantParameters);
//This is used to clean up the buffer binds after each invocation
//so they can be
//let them be changed and used elsewhere if needed.

void UnbindBuffers(FRHICmdList& RHICmdList);

private:
//This is how you declare resources that are going to be made
//available in the HLSL
FShaderResourceParameter TextureParameter;
};

TestShader.cpp:

#include "TestShaderPrivatePCH.h"
#include "ShaderParameterUtils.h"
#include "RHIStrictStates.h"
IMPLEMENT_UNIFORM_BUFFER_STRUCT(FPixelShaderConstantParameters ,
                                "PSConstants"))

FTestShaderPixelShaderDeclaration::FTestShaderPixelShaderDeclaration()
{
    const ShaderMetaType::CompiledShaderInitializerType& Initializer
        = FGlobalShader(Initializer)
    {
        //This call is what lets the shader system know that the surface
        //OutputSurface is going to be available in the shader. The second
        //parameter is the name it will be known by in the shader
        TextureParameter.Bind(Initializer.ParameterMap , TEXT(
            "TextureParameter")); //The text parameter here is the name of
        //parameter in the shader
    }
}
```

```
}

void FTestShaderPixelShaderDeclaration::SetUniformBuffers(
    FRHICmdList& RHICmdList, FPixelShaderConstantParameters&
    ConstantParameters)
{
    FPixelShaderConstantParametersRef ConstantParametersBuffer;
    ConstantParametersBuffer = FPixelShaderConstantParametersRef::
        CreateUniformBufferImmediate(ConstantParameters ,
        UniformBuffer_SingleDraw);

    SetUniformBufferParameter(RHICmdList, GetPixelShader(),
        GetUniformBufferParameter <FPixelShaderConstantParameters >()
        ConstantParametersBuffer);
}

void FTestShaderPixelShaderDeclaration::SetSurfaces(FRHICmdList
    RHICmdList, FShaderResourceViewRHIFRef TextureParameterSRV)
{
    FPixelShaderRHIParamRef PixelShaderRHI = GetPixelShader();
    if (TextureParameter.IsBound())//This actually sets the shader
        resource view to the texture parameter in the shader :)
    RHICmdList.SetShaderResourceViewParameter(PixelShaderRHI ,
        TextureParameter.GetBaseIndex(), TextureParameterSRV);
}

void FTestShaderPixelShaderDeclaration::UnbindBuffers(FRHICmdList
    & RHICmdList)
{
    FPixelShaderRHIParamRef PixelShaderRHI = GetPixelShader();
    if (TextureParameter.IsBound())

```

```
RHICmdList.SetShaderResourceViewParameter(PixelShaderRHI ,
    TextureParameter.GetBaseIndex(), FShaderResourceViewRHIParameter,
);
}

IMPLEMENT_SHADER_TYPE(, FTestShaderVertexShaderExample , TEXT("TestShader"), TEXT("MainVertexShader"), SF_Vertex);

IMPLEMENT_SHADER_TYPE(, FTestShaderPixelShaderDeclaration , TEXT("TestShader"), TEXT("MainPixelShader"), SF_Pixel);
```

在完成了这些之后，请在Tests文件夹中的单元测试ShaderTest.cpp中撰写一个单元测试。代码如下：

```
#include "../TestShaderPrivatePCH.h"
#include "../TestShader.h"
#include "AutomationTest.h"

DEFINE_LOG_CATEGORY_STATIC(TestLog, Log, All);

TGlobalResource <FTextureVertexDeclaration> GTextureVertexDeclaration;

void SaveScreenshot(FRHICommandListImmediate& RHICmdList,
    FTexture2DRHIFRef CurrentTexture);

void DoShaderTest() {
    FRHICommandListImmediate& RHICmdList = GRHICommandList.
        GetImmediateCommandList();

    FRHISourceCreateInfo info;
    ERHIFeatureLevel::Type FeatureLevel = ERHIFeatureLevel::SM5;
    //Create a temp Texture as Render Target
    FTexture2DRHIFRef TextureParameter = RHICreateTexture2D(128, 128,
```

```

    PF_R8G8B8A8, 1, 1, TexCreate_RenderTargetable , info);

//We Need a SRV to send to Shader

FShaderResourceViewRHIFRef TextureParameterSRV =
    RHICreateShaderResourceView(TextureParameter.GetReference(), ETextureFormat::R8G8B8A8,
SetRenderTarget(RHICmdList, TextureParameter , FTextureRHIFRef()));
RHICmdList.SetRasterizerState(TStaticRasterizerState <>::GetRHI());
RHICmdList.SetDepthStencilState(TStaticDepthStencilState <false,
    CF_Always >::GetRHI()));

static FGlobalBoundShaderState BoundShaderState;
//Get Vertex and Pixel Shader From Global Shader Map

TShaderMapRef <FTestShaderVertexShaderExample > VertexShader(
    GetGlobalShaderMap(FeatureLevel));
TShaderMapRef <FTestShaderPixelShaderDeclaration > PixelShader(
    GetGlobalShaderMap(FeatureLevel));

SetGlobalBoundShaderState(RHICmdList, FeatureLevel , BoundShaderState);
    GTextureVertexDeclaration.VertexDeclarationRHI , *VertexShader;
PixelShader);

PixelShader->SetSurfaces(RHICmdList, TextureParameterSRV);

FPixelShaderConstantParameters ConstantParameters;
//Set Parameters
ConstantParameters.StartColor = FVector4(0.5f, 0.5f, 0.0f, 1.0f);
PixelShader->SetUniformBuffers(RHICmdList, ConstantParameters);
//Build Vertices
FTextureVertex Vertices[4];
Vertices[0].Position = FVector4(-1.0f, 1.0f, 0, 1.0f);
Vertices[1].Position = FVector4(1.0f, 1.0f, 0, 1.0f);
Vertices[2].Position = FVector4(-1.0f, -1.0f, 0, 1.0f);

```

```

Vertices[3].Position = FVector4(1.0f, -1.0f, 0, 1.0f);

Vertices[0].UV = FVector2D(0, 0);
Vertices[1].UV = FVector2D(1, 0);
Vertices[2].UV = FVector2D(0, 1);
Vertices[3].UV = FVector2D(1, 1);

//Ask Video Card(RHI) to draw

DrawPrimitiveUP(RHICmdList, PT_TriangleStrip , 2, Vertices, sizeof
    Vertices[0]));

PixelShader->UnbindBuffers(RHICmdList);

//Save Result to Disk

SaveScreenshot(RHICmdList, TextureParameter);

}

void SaveScreenshot(FRHICmdListImmediate& RHICmdList,
    FTexture2DRHIFRef CurrentTexture)

{
check(IsInRenderingThread());

TArray<FColor> Bitmap;

FReadSurfaceDataFlags ReadDataFlags;

ReadDataFlags.SetLinearToGamma(false);
ReadDataFlags.SetOutputStencil(false);
ReadDataFlags.SetMip(0); //No mip supported ofc!

//This is pretty straight forward. Since we are using a standard
//format, we can use this convenience function instead of having
//lock rect.

RHICmdList.ReadSurfaceData(CurrentTexture , FIntRect(0, 0,
    CurrentTexture ->GetSizeX(), CurrentTexture ->GetSizeY()), Bi
    ReadDataFlags);

```

```
// if the format and texture type is supported

if (Bitmap.Num())
{
    // Create screenshot folder if not already present.

    IFileManager::Get().MakeDirectory(*FPaths::ScreenShotDir(), true)

    const FString ScreenFileName(FPaths::ScreenShotDir() / TEXT(
        "VisualizeTexture"));

    uint32 ExtendXWithMSAA = Bitmap.Num() / CurrentTexture ->GetSizeX();

    // Save the contents of the array to a bitmap file. (24bit only so
        // alpha channel is dropped)

    FFileHelper::CreateBitmap(*ScreenFileName , ExtendXWithMSAA ,
        CurrentTexture ->GetSizeY(), Bitmap.GetData());
}

else
{
}

IMPLEMENT_SIMPLE_AUTOMATION_TEST(FShaderTestBaseTest , "ShaderTestBaseTest",
    EAutomationTestFlags::EditorContext | EAutomationTestFlags::EngineFilter)

bool FShaderTestBaseTest::RunTest(const FString& Parameters)
{
    ENQUEUE_UNIQUE_RENDER_COMMAND(FTestShaderRunner ,
    {
        DoShaderTest();
    }
);
```

```
return true;  
}
```

最后编译整个工程。然后在自动化测试会话窗口中运行这个测试。

如果不出意外。你将会在工程目录\Saved\Screenshots\Windows下看到一个纯色的bmp文件。这表示我们的代码成功了。

11.5.4 我们做了什么

似乎有一种失落的感觉，不是吗？我们花费了如此巨大的成本，却只完成了一个非常简单的东西，那就是画了一个纯色正方形。这有什么意义呢？不要着急，请先让笔者解释一下，我们完成了什么：

- 我们撰写了一个Shader（不是Material）。
- 我们将这个Shader让虚幻引擎识别，并载入到了虚幻引擎的Shader列表中。
- 我们向渲染线程请求使用这个Shader，并绘制了一个正方形。
- 我们从渲染线程取回了结果。

我们之前是在虚幻引擎的帮助下完成对象的绘制，并在虚幻引擎的帮助下，将我们的材质转化为Shader来提供给虚幻引擎。如今我们能够完全脱离虚幻引擎的辅助来绘制对象了！

好吧，尽管我们完成了了不起的成就，但是那些代码如果读者一点也不明白，请允许笔者逐步地对这些代码内容进行讲解。事实上，我们的代码可以看作这样的几个层级：

Shader层 :我们的usf文件。

Shader包裹层 (**FTestShader**) : 我们的C++类，提供了与Shader进行交互的诸多功能，比如设置Shader的参数。

Shader操作层 :这些被笔者直接写在了单元测试中。实际上这些操作可以被封装为一个单独的类。

那么接下来我们就要分析我们的Shader包裹层。从TestShader.h中我们会发现，实际上我们的代码可以看作以下三段：

Shader变量声明与定义 这里的变量声明与定义与我们usf文件中的内容是匹配的。

顶点结构描述 在OpenGL和DirectX中，都允许你自己定义顶点包含的数据，而不是规定顶点具体包含哪些信息。在我们的这个案例里面，顶点只包含两个数据：顶点位置和UV坐标。实际上你可以定义更多的数据，例如顶点法线方向、顶点颜色，甚至你可以定义第二套UV。

Shader着色器描述 由于我们的着色器有两个部分：顶点着色器和像素着色器。因此，对应的代码也分为以下两个类：

顶点着色器 顶点着色器部分我们十分简单地用了一个默认定义，因为实际上我们也没有对顶点着色器内容进行任何多余的规定。

像素着色器 像素着色器部分则更加复杂一些。由于我们的像素着色器需要一个传入参数：StartColor。因此，我们需要申请一个通用的存储区，并将我们的参数绑定到存储区中。同时

我们也需要一个ShaderResourceView来允许Shader对贴图进行随机的读写访问。

对于Shader的操作层，首先最重要的一点就是，我们必须让绘制操作是在渲染线程中执行的。因为所有的渲染请求都是由渲染线程管理，如果游戏线程试图请求渲染绘制，将会导致断言失败。而且这也是不合理的（不利于并行化）。

所以我们需要借助一个宏：

`ENQUEUE_UNIQUE_RENDER_COMMAND` 系列

这个宏的实质是向渲染线程的TaskGraph中挂载一个Task。由于渲染线程的TaskGraph有做多线程停等机制，因此当一个新TaskGraph挂载的时候，如果渲染线程已经有任务，会在执行完任务之后执行这个新的Task。否则就会直接执行，从而实现了“游戏线程向渲染线程发送指令”这样的设计。我们获取了RHI命令列表，然后向命令列表中填充了以下一系列的命令：

1. 创建了一个临时的贴图。
2. 填充了一系列的命令，包括填充我们的Shader到显存。
3. 创建了一个正方形，然后用DrawPrimitiveUP去请求GPU绘制。
4. 取消绑定，把渲染结果写出到一个贴图上。

这就是我们这段代码的具体含义。接下来我们会继续在这样的基础上扩展，以制作更多有趣的Shader。

11.6 材质

在对材质系统进行介绍之前，笔者必须强调，本书立足于“程序”和“架构”的分析，因此对具体渲染方程的描述不会涉及过多。也就是说，只会介绍“代码是这样计算”，但是为何采用这样的算法、算法的优劣，如果要介绍的话就会是长长的推导论文。笔者并不是计算机图形学出身，不敢妄言这些公式优劣高低，因此很惭愧不能给大家剖析公式的优美之处。本书内容专注于虚幻引擎自身架构的设计，而不会出现大量的数学内容，可能对数学有些吃力的读者会感到轻松。

对于材质系统进行研究之前，我们首先需要整理一下材质系统本身是如何工作的。换句话说，在寻找问题的答案之前，首先需要明确问题是什么。向上，材质系统提供了一套节点编辑系统用于编辑当前材质，同时我们也知道，这个节点编辑系统必然会编译成具体的Shader代码以供调用。向下，材质系统最终会通过传递当前材质的RenderProxy，从而让网格的渲染代理能够知道如何渲染当前对象。

因此，我们对材质系统的研究，也将会按照这个向上和向下的思路来完成。

当然，需要注意的是，不同于蓝图系统的高度可扩展性和可定制性，虚幻引擎的渲染系统，尤其是材质系统，实际上是不支持随意定制的。许多人会提出反驳，认为Custom节点可以输入自己的HLSL代码，相当于提供了无限的自由度。但是这种想法是错误的，因为Custom实际上只产生了一个节点——你没有办法干涉整个材质的渲染过程。甚至连材质的Shader Model类型都是被写死在虚幻引擎代码中的。

也就是说，新增ShadingModel就必须改动源码。仅仅依靠插件无法达到增加新的着色模型的效果。

11.6.1 概述

编辑器

许多朋友对虚幻引擎的材质的最深印象就是那个节点编辑器。那么遵循从兴趣开始的原则，我们先从编辑器开始讲。如何找到是什么样的代码产生了编辑器呢？如果你并不清楚材质编辑器是一个什么样的Slate控件，你可以打开控件反射器，鼠标光标指向当前材质编辑器的窗口，就能看到控件类型。当然，如果看过前文关于蓝图的介绍，你会提出这应该是一个SGraphEditor。于是我们搜索“SNew(SGraphEditor)”，很快就会发现——产生材质编辑器的代码在MaterialEditor.cpp中：

```
return SNew(SGraphEditor)
    .AdditionalCommands(GraphEditorCommands)
    .IsEditable(true)
    .TitleBar(TitleBarWidget)
    .Appearance(AppearanceInfo)
    .GraphToEdit(Material->MaterialGraph)
```

材质编辑器本身就是一个SGraphEditor。大部分内容都不大需要解释，导致逻辑蓝图编辑器与材质蓝图编辑器的最大区别来源于传递给GraphToEdit这个Slate变量的参数不同。Material变量就是当前正在编辑的材质，只是传递给材质编辑器的材质蓝图不同于逻辑蓝图，导致蓝图的编辑、节点都完全不一样了。

材质蓝图

材质蓝图（MaterialGraph）依然是一个蓝图，因此UMaterialGraph类继承自UEdGraph类。比较特殊的部分如下：

Material 持有指向对应材质的引用。

UMaterialFunction 当前对应的材质函数。如果这是一个普通材质，那么这个变量为空。

RootNode 根节点。这个节点就是那个有一堆输入端口（BaseColor、粗糙度）的节点。如果是材质函数，这个变量为空。

MaterialInputs 材质输入。是一个FMaterialInputInfo类型的数组。如果是材质函数，这个变量不会被设置。

AddExpression(UMaterialExpression*Expression) 添加一个材质表达式到当前材质蓝图，返回新创建的蓝图节点指针。

LinkGraphNodesFromMaterial() 根据材质信息链接蓝图。

LinkMaterialExpressionsFromGraph() 根据蓝图信息链接材质。

IsInputActive(classUEdGraphPin*GraphPin) 判断当前给定的蓝图输入是否可用。

关于这些函数的进一步分析，会留到下文介绍Material之后再进行讲解，这里希望读者对这个结构先有一些印象。

UMaterial

UMaterial这个结构则相对来说更加复杂，如图11-8所示。我们不妨向上一层，首先查看UMaterialInterface这个接口定义。这个接口抽象了材质与材质实例之间的共有特性。即使这样，UMaterialInterface依然包含大量的接口定义和代码。分析引擎的方法是抓大放小，并非每一个函数、每一个变量的原理我们都需要了解的一清二楚，而是要抓住重点。所以我们不再列出一大堆函数的定义。对于继承自UMaterialInterface的类，最主要的两个功能是编译和提供着色器组合。前者被CompileProperty和CompilePropertyEx完成，后者是通过GetRenderProxy函数返回一个FMaterialRenderProxy结构实现。

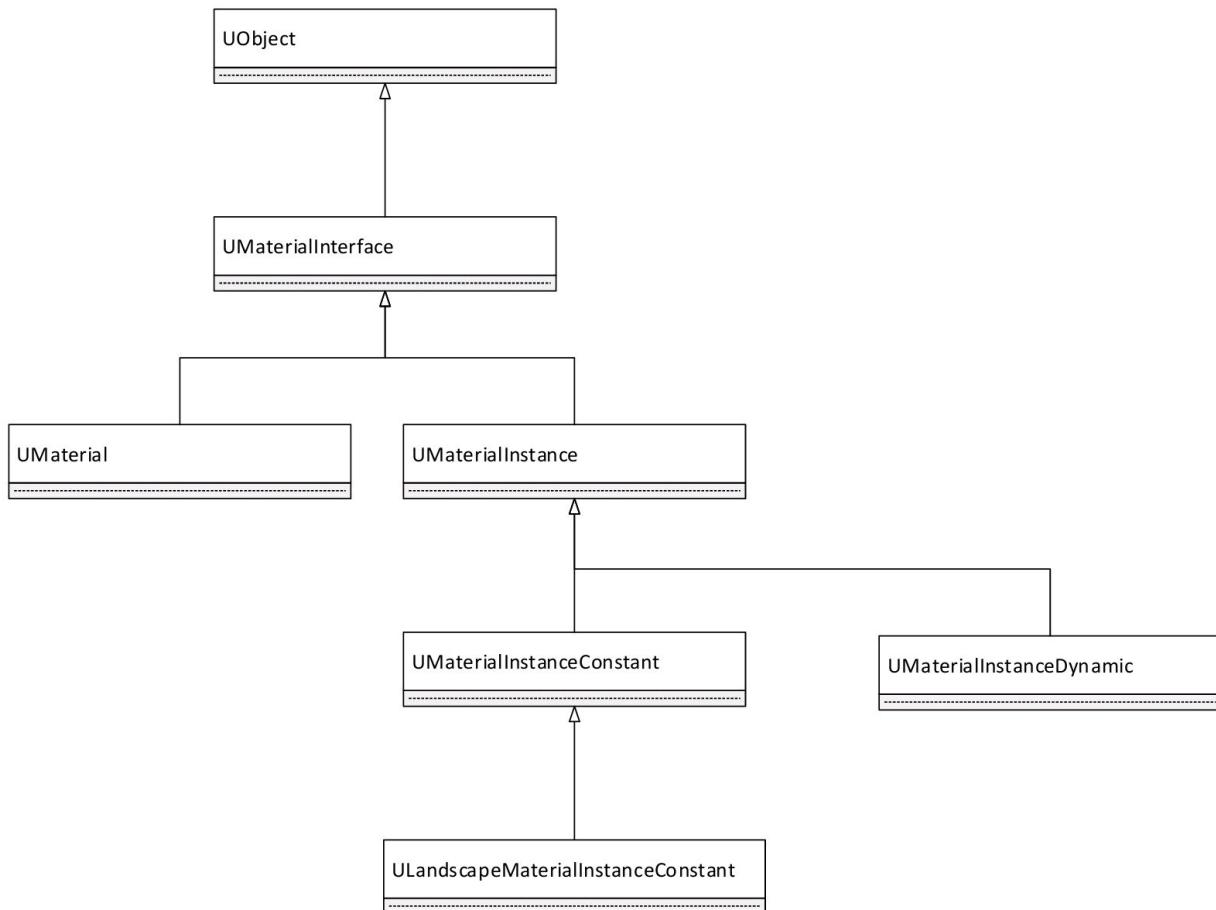


图11-8 UMaterial系列继承关系

11.6.2 材质相关C++类关系

现在摆在我们面前的是三个主要的类：UMaterial，FMaterial，FMaterialRenderProxy。首先我们必须理清这三个关系，如图11-9所示。

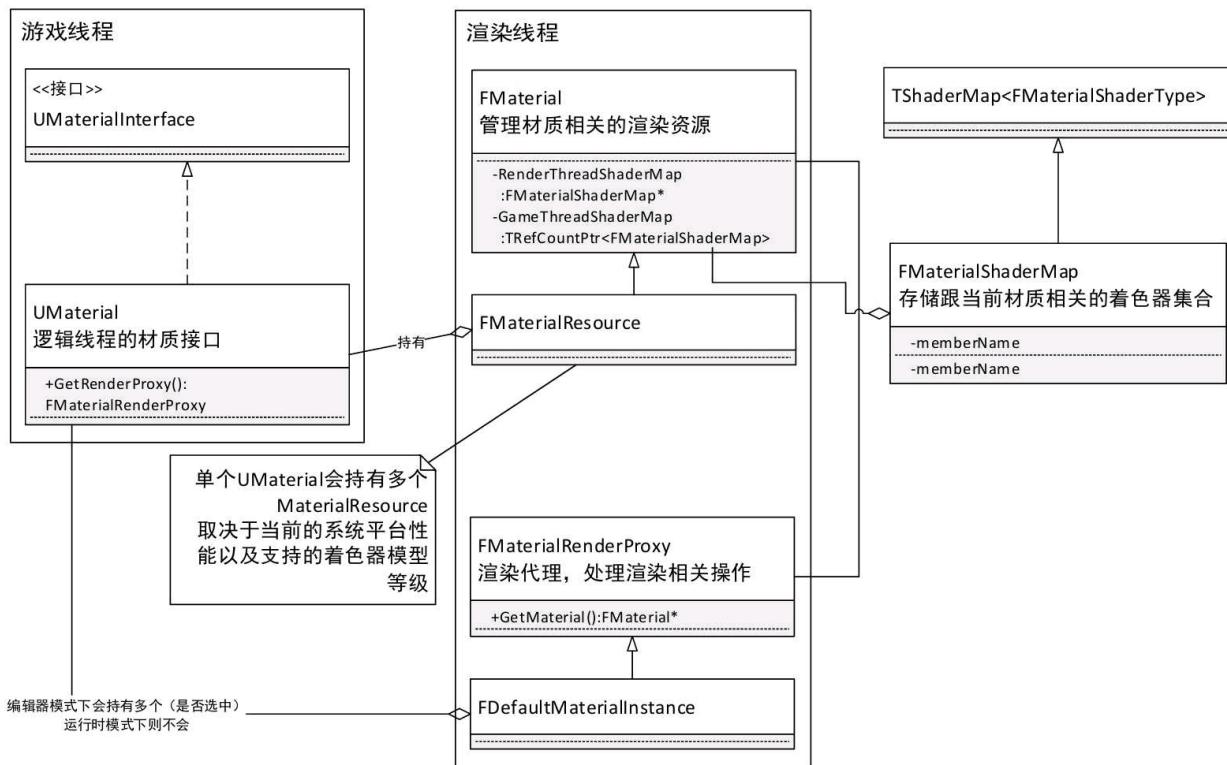


图11-9 UMaterial, FMaterial, FMaterialRenderProxy及相关类关系

按照前文叙述的判定规则：U开头代表游戏线程，F开头代表渲染线程。我们将UMaterialInstanceConstant这三个类分成了两族：用于逻辑线程使用的UMaterial，用于渲染使用的FMaterial和FMaterialRenderProxy。其对应用途如下：

UMaterial 持有逻辑线程相关的材质信息，并作为逻辑线程的访

间接口，管理其余两个渲染线程的类。

FMaterial 负责管理渲染资源的类。这里的渲染资源主要就指ShaderMap。其负责处理ShaderMap的编译等。

FMaterialRenderProxy 渲染代理，负责处理渲染相关操作的类。

这三个类，加上由FMaterial负责管理的FMaterialShaderMap类（及其子类）共同完成了材质这个数据结构的表达。

11.6.3 编译

当你点击材质编辑器的应用按钮或者保存按钮的时候，UMaterial的PostEditChangeProperty 函数就被触发。于是对当前材质的节点树进行编译，生成着色器缓存 CacheShaders。顺便一提，这个编译对应的函数是BeginCompileShaderMap，这个编译有可能失败，虚幻引擎只是“开始”编译，如果失败的话，还是会还原成原来的ShaderMap。关于命名为何带有Begin，在虚幻引擎中，大部分带有Begin的函数，其执行过程有一部分在另一个线程中，通常是在渲染线程中完成。如果你对这个内容比较陌生，可以再次回头看一下前文中关于并行与并发的描述。

而编译过程中，编译器根据依赖平台不同而不同，在笔者使用的Win64平台，是通过FHLSLMaterialTranslator完成的。其会根据UMaterial，调用对应的CompilePropertyEx逐属性地进行编译，最终的结果是一段HLSL代码。你可以从材质编辑器的窗口→HLSL代码中看到最终编译的结果。

从节点图转化为HLSL的过程其实并不是非常的复杂。你可以在引擎Engine\Shaders文件夹中找到一个叫MaterialTemplate.usf的文件。这个就是编译的“模板”。实际上转化过程就是把这个模板中的一些部分替换为具体表达式。甚至比PHP渲染静态页面的过程还要简单，其直接通过类似printf中的%s标记来完成替换。

已经产生的HLSL代码会被处理，得到ShaderMap结构。那么我们接下来需要分析的就是这个颇为独特的ShaderMap。

11.6.4 ShaderMap产生

在前文对渲染过程的描述中，已经介绍了位于渲染线程的渲染代码是如何使用ShaderMap提供的数据，从中提取出真正需要的着色器并提交绘制请求。以上回答的是“我是谁”和“我往哪里去”的问题。接下来我们将会分析“我从哪里来”的问题，即材质如何编译、产生ShaderMap并缓存起来。采用倒过来的介绍顺序的原因是，当读者首先阅读“ShaderMap产生”的介绍时，会对一大堆陌生的东西产生困惑：为什么要产生这个东西？这个东西有什么用？为何要产生多个版本的顶点和像素渲染器，而不是同一个版本？所以笔者采用倒叙方式进行介绍。

前文已述，当HLSL代码被生成后，就需要进入真正的着色器编译阶段。首先需要创建一个ShaderMap实例。为了方便叙述且与源码一致，我们称该实例为NewShaderMap。

材质节点图生成的HLSL代码只是一批函数，并不具备完整的着色器信息。这些代码会被嵌入到真正的着色器编译环境中，然后重新编译为最终的ShaderMap中的每一个着色器。

这个着色器编译环境对应的类是FShaderCompilerEnvironment。该类通过MaterialTranslator::GetMaterialEnvironment函数初始化实例，主要完成以下两个步骤：

1. 根据当前材质的各种属性，设置各种着色器宏定义。从而控制编译过程中的各种宏开关是否启用。比如如果当前材质的变量bUsesLightMapUVs被设置为真，则设置宏定义中的LIGHTMAP_UV_ACCESS为真。
2. 根据FHLSLMaterialTranslator在解析过程中得出的当前参数集合，添加参数定义到环境中。

初始化完成后，开始进行实际的编译工作：

1. 获取刚刚生成的HLSL代码（就是FString格式的字符串）。
2. 将刚刚生成的代码作为“Material.usf”文件，添加到编译环境中。
3. 调用NewShaderMap的Compile函数：
 - a. 调用FMaterial::SetupMaterialEnvironment函数，设置当前的编译环境。笔者尚不明白为何编译环境需要分为两个步骤进行设置。
 - i. 通过FShaderUniformBufferParameter::ModifyCompilationEnvironment把通用缓存的参数结构定义添加到编译环境中。这个定义虽然看上去是个文件（“UniformBuffers变量名.usf”），但其实是通过CreateUniformBufferShaderDeclaration根据你用到的参数生成的，比如你用的贴图，就会转化为Texture2D。
 - ii. 根据曲面细分设置一系列的宏。
 - iii. 根据混合模式（Opaque、Masked、Translucent、

- Additive、Modulate) 设置宏。
- iv. 根据接收贴花的类型设置宏定义。
 - v. 根据Unlit、DefaultLit等光照模式设置宏定义。
- b. 获取所有的顶点工厂类型 (FVertexFactoryType)，对每个顶点工厂类型执行以下操作：
- i. 创建一个FMeshMaterialShaderMap*指针MeshShaderMap。
 - ii. 查看当前顶点工厂类型对应的FMeshMaterialShaderMap是不是已经存在，已经存在就赋值给该指针，否则创建一个新的。
 - iii. 对当前MeshShaderMap，调用BeginCompile：
 - I. 遍历包含所有的FShaderType的列表，获取 FMeshMaterialShaderType*ShaderType。
 - II. 通过HasShaderType函数检查是否已经编译，如果没有编译，将一个编译任务加入到编译工作列表 SharedShaderJobs中。

根据当前的顶点工厂类型，添加工厂定义到编译环境中。

调用GlobalBeginCompileShader创建编译任务。设置 FShaderCompilerInput实例Input，包括编译目标、着色器格式、源文件名、入口名等。然后设置一大堆的编译参数，这里不再赘述，有兴趣的读者可以直接查看该函数。
 - iv. 使用异步着色器编译流水线执行这些着色器编译任务。

第12章

Slate界面系统

作为核心界面系统，Slate是一个跨平台的、硬件加速的图形界面框架，采用了C++的泛型编程来允许直接使用C++语言撰写界面。

12.1 Slate的两次排布

Slate是一个分辨率自适应的相对界面布局系统，为了能够完成这样的目的，Slate实质上采用了一个“两次”排布的思路。

1. 首先，递归计算每个控件的大小，父控件会根据子控件来计算自己的大小。
2. 然后，根据控件大小，具体计算出每个控件的绘制位置。

由于有部分控件是“大小可改变”的，因此必须先计算出“固定大小”，才可能实际排布这些控件。

12.2 Slate的更新

Slate系统的更新，实质上是与引擎内部更新分开的。FEngineLoop类会首先更新引擎，随后在确保FSlateApplication已经初始化后，调用FSlateApplication的Tick函数进行更新。

`FSlateApplication`代表了当前正在运行的Slate程序，F开头意味着这个类本身并非一个Slate组件，其只是行使“管理”的责任。

从实际行为上看，`FSlateApplication`也确实是负责绘制当前窗口，当自身的`Tick`函数被调用的时候，它会请求更新所有的Window，即调用`TickWindowAndChildren`函数。随后才会调用Draw系列的函数，去绘制所有的对象。

12.3 Slate的渲染

需要注意的是，Slate的渲染并非是一个“递归渲染”的过程，而是一个“先准备，再渲染”的过程。

所有的Slate对象将会首先准备需要渲染的内容，即`WindowElement`。然后这些内容会被交给`SlateRHIRenderer`负责，最终被绘制出来。

Slate同样是借助虚幻引擎的RHI硬件绘制接口进行绘制的。所以Slate的绘制实际上走的是标准的渲染流程：

1. 将控件对象转换为需要绘制的图形面片。
2. 通过PixelShader和VertexShader来使用GPU进行绘制。
3. 拿回绘图结果，显示在`SWindow`中。

值得强调的是，Slate的渲染是没有开启深度检测的。而且在`SlateVertexShader`中，每个渲染对象的Z轴均会被设置为0，这意味着：

1. 控件对象简单地按照绘制顺序堆叠，后一个控件将会直接叠加在前

一个控件之上。

2. 控件对象不会存在一个更新区域 的概念，这意味着即使被遮盖住的位置，依然会被绘制出来。

关于这一点，可能会有读者提出争议，但实际上根据RenderDoc抓取的GPU绘制结果，这一结论是正确的，如图12-1所示：

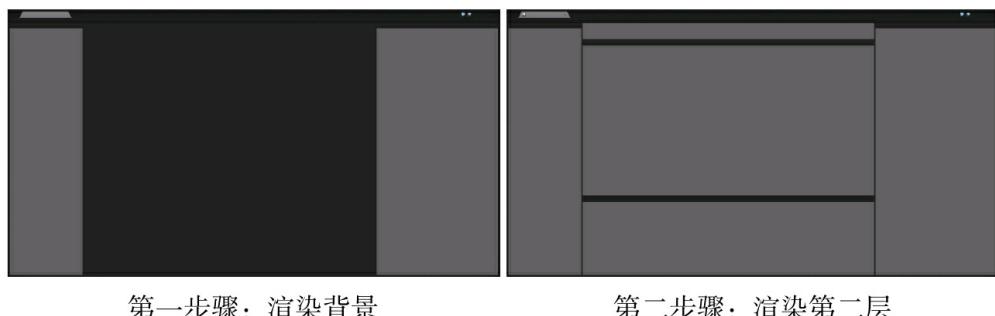


图12-1e Slate界面绘制案例

当然，如果简单地逐控件进行绘制，那么每个控件将会产生一个DrawCall渲染请求，这是一个非常低效的过程，这将导致复杂界面的渲染速度非常低下。

为了解决这个问题，虚幻引擎采用了一个称为ElementBatch（对象批量渲染）的概念。为了能够引出这样的概念，我们不妨进行这样一番推理：

1. 有些控件对象实际上是可以同时渲染的，只要它们互相不重叠就可以。
2. 重叠的对象必须按照先后次序渲染。
3. 实际上来说，只有SOverlay、SCanvas这样的控件，才会产生控件对象堆叠，类似SVerticalBox这样的控件，是根本不会产生堆叠的。

因此，如图12-2所示，对每个需要堆叠对象的层级编号增加1，就能够构成一个树状结构。其中如果是平级对象，如SVerticalBox，其子对象的层级编号相等。如果是SOVERLAY，其每一个子对象的层级编号，都是在其上一个子对象层级编号基础上加1。

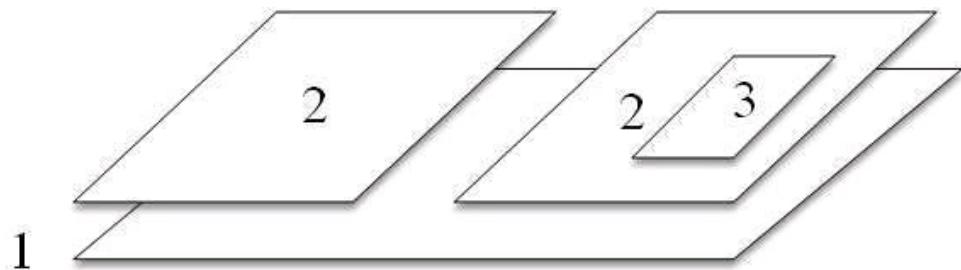


图12-2 Slate渲染层级示意

于是我们会发现，处于同层级编号的几何对象，都是不产生堆叠和遮挡的。所以可以将这些对象同时进行渲染。对于上图中所展示的案例而言，这样的批量渲染方案，能够将渲染请求次数从5降低到3。对于真正的控件来说，渲染量将会降低得更为明显，因此这是一个相当取巧的方案。

第13章 蓝图

问题

用了这么久的蓝图，它究竟是如何工作的呢？

C++函数是如何向蓝图暴露的呢？

蓝图虚拟机是如何运行的呢？

13.1 蓝图架构简述

虚幻引擎提供了蓝图系统作为可视化编程系统。蓝图系统从Kismet系统发展而来，同时整合了UnrealScript的诸多优势，从而成为了一个完整的面向对象的可视化编程系统。虽然对蓝图的争议从虚幻引擎4.0.1版本开始一直持续到4.13版本，但是我们不可否认，虚幻引擎的蓝图系统在高速迭代上具有极大的优势。小型蓝图的编译速度远远快过C++的编译速度。并且最新版本的虚幻引擎能够在发布的时候将蓝图编译为C++，从而提升蓝图在最终发布版本的执行效率，进而弥补蓝图在性能上的固有缺陷。因此分析，选择C++的理由可能只剩下C++本身更大范围的API，以及蓝图系统在内容较多时候不如C++直观的原因。

蓝图系统依然是一套依托于虚幻引擎现有UClass、UProperty、

UFunction框架，根植于虚幻引擎UnrealScript字节码编译器系统的一套可视化编程系统。这就意味着：

1. 蓝图最终编译结果依然会转化为 UClass、 UProperty、 UFunction 信息。其中指令代码将会存储于 UFunction 的信息中。
2. 蓝图本身可以看作是一种可视化版的 UnrealScript，是经过转化和经过语法解析生成的字节码。

13.2 前端：蓝图存储与编辑

需要注意的是，蓝图系统实际上是由三个部分组成的：蓝图编辑系统、蓝图本身、蓝图编译后的字节码。最终编译完成的蓝图字节码将不会包含蓝图本身的节点信息。这部分信息是在UEdGraph中存储的，这也是一种优化。

UEdGraph用于表示蓝图的数据结构。从整体来说，可以把其看作以下这个结构：

```
UEdGraph
| -Schema
| -Nodes
| -SubGraphes
```

13.2.1 Schema

Schema在英文中的意思为模式，笔者觉得理解为语法比较合适。其规定了当前蓝图能够产生什么样的节点等信息。具体而言，当定义了自

己的Schema之后，通过重载对应的函数即可实现语法的规定，部分相关函数的含义列表如下：

1. GetContextMenuActions 定义在当前蓝图编辑器中右键菜单的菜单项。通过向 FGraphContextMenuBuilder 引用中填充内容，实现对右键菜单的定制。
2. GetGraphContextActions 与上面介绍的 ContextMenuActions 不同之处在于，定义的是右键菜单和拖曳连线之后弹出的菜单公用的菜单项。
3. CanCreateConnection 该函数传入两个 UEdGraphPin，判断是否能够建立一条连线。需要注意的是，返回值并非一个布尔值。其通过构造一个 FPinConnectionResponse 作为返回。构造函数中需要传入两个参数，第一个参数为具体的连接结果，第二个参数是一个消息返回，包括：
CONNECT_RESPONSE_MAKE 可以连接，直接建立起一条连线。
CONNECT_RESPONSE_DISALLOW 不准建立连线。
CONNECT_RESPONSE_BREAK_OTHERS_AA (第一个参数) 对应的连接头的所有其他连接线都会被截断。
CONNECT_RESPONSE_BREAK_OTHERS_B B (第二个参数) 对应的连接头的所有其他连接线都会被截断。
CONNECT_RESPONSE_BREAK_OTHERS_AB 两个连接头对应的其他连接线都会被截断。
CONNECT_RESPONSE_MAKE_WITH_CONVERSION_NODE 建立起一条连接线，但会立即产生一个转换节点。
4. CanMergeNode 字面含义为能够合并节点，具体作用不明。
5. Try 系列函数包括 TryCreateConnection 这样的函数，其主要是在进行具体的连线之前进行一次先行检测，此时返回 false 作为失败，不会

有任何副作用，只是“尝试失败”。这个函数比CanCreateConnection简单，且无副作用。而CanCreateConnection的某些返回值是有副作用的。

6. Debug监视相关函数，包括DoesSupportPinWatching、IsPinBeingWatched、ClearPinWatch，主要用于设置节点值的监视。
7. CreateDefaultNodesForGraph为当前UEdGraph创建一个默认节点。
许多蓝图都有一个基础节点，例如状态机的入口节点。通过重载该函数可以实现。
8. Drop系列函数在从外部拖放一个资源对象进入蓝图时触发。包括拖放到蓝图中、拖放到节点上、拖放到Pin上。
9. CreateConnectionDrawingPolicy创建一个连接线绘制代理。可以通过自定义代理来修改连接线的样式

这里只介绍了部分相对来说常用的重载函数。如果希望查看具体有哪些语法可以重载，可以查看EdGraphSchema.h中的定义。根据笔者的经验，实现一套自己的状态机蓝图是完全没有问题的。

13.2.2 编辑器

蓝图的编辑器实质上是一个Slate控件，即SGraphEditor。可以这样理解，UEdGraph存储蓝图的“数据”资料。SGraphEditor通过解析数据，生成对应的“显示控件”，并处理交互。类似于MVC架构中的View部分。

通过在SNew实例化SGraphEditor的时候，传入一个UEdGraph*类型的参数GraphToEdit，即可指定当前编辑器正在编辑的蓝图。当然，也有更多的参数可以设置：

- IsEditable是否可以编辑。
- DisplayAsReadOnly是否以只读方式显示。
- IsEmpty是否为空。
- TitleBar编辑器顶部工具栏的控件。

以上只举出部分参数。具体可以参考GraphEditor.h中SGraphEditor的定义。

行为树编辑器是这样产生自己的GraphEditor的：

```
SNew(SGraphEditor)
    .AdditionalCommands(GraphEditorCommands)
    .IsEditable(this, &FBehaviorTreeEditor::InEditMode,
                bGraphIsEditable)
    .Appearance(this, &FBehaviorTreeEditor::GetGraphAppearance)
    .TitleBar(TitleBarWidget)
    .GraphToEdit(InGraph)
    .GraphEvents(InEvents);
```

之前已经提到过，UEdGraph并非一定需要编译：UEdGraph只是一种数据包。如果你有需要，你可以定义自己的蓝图系统，用于编辑自己的素材资源。例如著名的地下城建筑师插件就是通过自定义蓝图完成对地下城生成逻辑的编辑。

13.3 后端： 蓝图的编译

只有包含具体逻辑的蓝图才需要编译为蓝图字节码，有一些蓝图仅仅只需要自身数据就可以了。

在对这个流程进行分析之前，我们首先必须要明确，蓝图作为一种面向对象的语言，其三大数据的存储方式为：元数据 metadata（UClass）、属性数据、方法数据。也就是说，我们还得再次对UClass做进一步分析。

UClass继承自UStruct。需要注意的是，UStruct只包含成员变量的反射信息，其支持高速的构造和析构；而UClass则更加重量级，其需要对成员函数进行反射。在UClass中有成员变量FuncMap，其用于存储当前UClass包含的成员函数信息。换句话说，尽管从感觉上而言，我们调用的是对象 的某个成员函数，但是实际上，同一个类的所有对象共有同样的成员函数，因此成员函数的信息存储于类数据中，而非存储在每个对象中。

也就是说，类的元数据信息、类的成员函数属性信息、成员变量属性信息均存储于类UClass的数据中，而非类对象的数据中。每个对象只包含自己可能变化的部分，也就是对象自己的成员变量数据信息。

那么，蓝图编译完成产生的结果，应当是一个包含完整信息的UClass对象，而不是对应的类的对象。如果觉得这个很难理解，我们可以这样想：UClass就像是一套考试的试卷，其包含自己独有的信息，例如自己的成员变量信息，如姓名、班级、学号，甚至答案本身也是一种成员变量；其也有自己的成员函数信息，就像一道一道的题目。而每个学生的答案实际上只是一个数据包，是这个试卷产生的实例，包含自己独立的成员变量。但是其对应的函数都是一致的（每一道题都一样）。我们的蓝图UBlueprint编译完毕后，产生的是试卷，这就是UBlueprint和

UClass的关系。务必理清这个关系，才继续往下阅读！

根据虚幻引擎官方文档（由于该文档没有被翻译，笔者会翻译部分内容，并增加了笔者的个人理解）。一个蓝图会经历以下过程，最终产生出UClass：

类型清空 清空当前类的内容。每个蓝图生成的类，即 UBlueprintGeneratedClass，都会被复用，并非被删除后创建新的实例。对应函数为CleanAndSanitizeClass。

创建类成员变量 根据蓝图中的“NewVariables”数组与其他位置定义的类成员变量，创建UProperties。对应的函数为 CreateClassVariablesFromBlueprint。

创建成员函数列表 编译器执行这四个过程：处理当前的事件蓝图，处理当前的函数蓝图，对函数进行预编译，创建函数列表。

处理事件蓝图 调用CreateAndProcessUberGraph函数，将所有的事件蓝图复制到一张超大蓝图里面，此时每个节点都有机会去展开自己（例如宏节点）。同时每个事件蓝图都会创建一个 FKismetFunctionContext对象与之对应。

处理函数蓝图 普通的函数蓝图通过ProcessOneFunctionGraph函数进行处理。此时每个函数蓝图会被复制到一个暂时的蓝图里面，同时节点有机会被展开。同样地，每个函数蓝图都会有一个 FKismetFunctionContext与之对应。

预编译函数 PrecompileFunction函数对函数进行预编译。具体而言，其完成了这样的内容：

- “修剪”蓝图，只有连接的节点才会被保留，无用的节点会被删掉。
- 运行现在还剩下的节点句柄的RegisterNets函数。
- 填充函数的骨架：包括参数和局部变量的信息。但是里面还没有脚本代码。

组合和链接类 此时编译器已经获得了当前类的所有属性与函数的信息，可以开始组合和链接类了。包括填充变量链表、填充变量的大小、填充函数表等。这一步本质上产生了一个类的头文件以及一个类默认对象（CDO）。但是缺少类的最终标记以及元数据。

编译函数 请注意这一步还没产生实际的虚拟机码！这一步包括：

- 调用每个节点句柄的Compile函数，从而生成一个FKismetCompiledStatement对象。
- 调用AppendStatementForNode函数。

完成类编译 填充类的标记和元数据，并从父类中继承需要的标记和元数据。最终进行一系列的最终检测，确保类被正确编译。

后端产生最终代码 后端逐函数地转换节点状态集合为最终代码。如果使用FKismetCompilerVMBackend，则产生虚拟机字节码，使用FKismetCppBackend则产生C++代码。在笔者撰写时，CppBackend作为实验性功能，已经可以投入使用。

复制类默认值对象属性 借助一个特殊函数——CopyPropertiesForUnrelatedObjects，从而将老的类默认对象的值复制到新的对象中。因为这个转换是通过基于Tag的序列化完成的，因此只要名字没变，值就会被转换过来。而组件则会被重新实例

化，并被适当地修复。在整个过程中，生成的新类的默认对象CDO是权威参照。

重新实例化 由于新的类的大小可能会改变，参数也有可能增减，编译器需要对原来的那个类的所有对象进行重新实例化。首先借助TObjectIterator来找到正在编译的类的所有实例，生成一个新的类，然后通过CopyPropertiesForUnrelatedObjects将老实例的值更新到新的实例。

在这个过程中涉及的术语如下：

FKismetCompilerContext 完成实际编译工作的类。每次编译都会创建一个新的对象。这个类存储了正在被编译的蓝图和类的引用。

FKismetFunctionContext 包含编译一个单独函数的信息，持有对应蓝图（不一定是函数蓝图）的引用、属性的引用以及生成的UFunction的引用。

FNodeHandlingFunctor 单例类，也是辅助类。用于处理编译过程中的一类 节点的类。包含一系列的函数，用于注册连接线以及生成编译后的Statement信息。

FKismetCompiledStatement 一个编译器的独立工作单元。编译器把节点转换为一系列已经编译好的表达式，最终后端会将表达式转换为字节操作码。案例：变量赋值、Goto、Call。

FKismetTerm 蓝图中的一个端子（literal、const或者variable的引用）。每个数据链接点都对应一个这个东西！你可以在

`NodeHandlingFunctor`中创建你自己的端子，用来捕获变量或者传递结果。

官方文档关于具体编译过程的描述相对来说比较笼统，因此笔者会用一个更加具体的方式来描述编译的某些过程。但是官网的文档依然非常有价值的，笔者在分析这一段的时候，会大量引用文档中的过程和名词，如果你不记得某个名字的意义，请务必返回查询。

多编译器适配

对于每一个对当前蓝图进行编译的请求（通过调用`FKismet2CompilerModule`的`CompileBlueprint`函数），`FKismet2CompilerModule`会询问当前所有的`Compiler`，调用它们的`CanCompile`函数，询问是否可以编译当前蓝图。

换句话说，如果某个类实现了`IKismetCompilerInterface`接口，那么就可以通过以下代码：

```
IKismetCompilerInterface& KismetCompilerModule =
    FModuleManager::LoadModuleChecked<IKismetCompilerInterface>("KismetCompiler");
KismetCompilerModule.GetCompilers().Add(this);
```

来在Kismet编译器中注册编译器。虚幻引擎自己的系统UMG编辑器就是通过这样的方式注册自己为一个编译器的。

编译上下文

这里讨论的蓝图主要是我们平时使用的代码蓝图，也就是不讨论状态机、动画蓝图和UMG蓝图。如果发现KismetCompiler能够编译当前蓝图，虚幻引擎会创建一个Kismet编译器上下文，即FKismetCompilerContext结构。这个结构会负责编译当前的蓝图。官方文档描述的编译过程，其实可以在FKismetCompilerContext的Compile函数中找到对应。

存在多种类型的CompilerContext，持有编译“一段”内容所需要的信息。接下来就会讲解主要上下文之间的关系。

整理与归并

图13-1展示了蓝图编译过程中，从蓝图UBlueprint结构到编译上下文FKismetCompileContext的主要过程。其中非箭头线表示持有关系，箭头表示步骤。这一步主要是将蓝图的数据结构进行整理、简化，转化为线性的调用链表，方便接下来逐节点地编译。

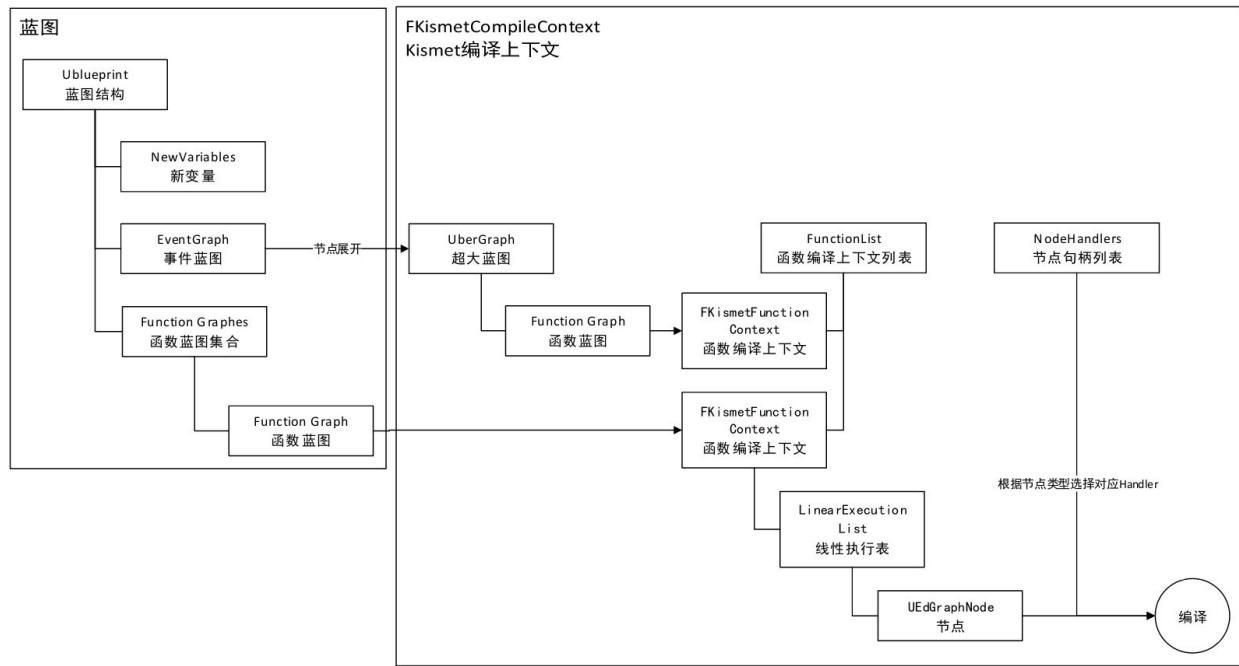


图13-1 前半部分：从蓝图到编译上下文

依前文所言，首先会对新变量进行处理，添加到成员变量数组中。这一段并不复杂，在此不做赘述，重点在接下来的过程。

事件蓝图展开 首先就是对事件蓝图的处理。事件蓝图将会被展开为一个超大的蓝图，虚幻引擎源码称之为UberGraph（这个Uber不是你手机上的那个APP）。在这个过程中会展开收缩的节点，然后整理为一个一个的函数蓝图。其中一个事件实质上是被看作一个单独函数处理。因此单个超大蓝图持有多个子函数蓝图。对每一个子函数蓝图，分配一个函数编译上下文。

函数蓝图整理 将每个蓝图的函数蓝图集合，拆分为一个一个的函数蓝图，然后转化为一个一个的函数编译上下文。

函数编译上下文列表收集 完成了从事件蓝图、函数蓝图到函数编译上下文的转换之后，这些上下文会被收入函数编译上下文列表，即FunctionList中。到这一步，已经完成了从“具体蓝图”到“统

一函数编译结构”的转换。如果你再次查看所配的示意图，会发现编译上下文列表与事件蓝图和函数蓝图集合的位置是对应的，有一种对称的感觉。这就是笔者希望读者意识到的这样一种对应关系。

从函数到具体节点 此时已经可以开始对函数进行编译。函数编译上下文通过对当前节点的整理，能够得到一个“线性执行表”。需要注意的是，蓝图函数确实是线性执行的。即使是借助宏实现的 Sequence序列执行，最终依然会被整理为线性执行表。这个表里面就对应着一个一个的函数节点。

节点编译 此时线性执行表中依然存在以UEdGraphNode表示的节点。这些节点将会被编译系统处理而转化为合适的编译信息。为了能够应对更多的节点类型，而不需要频繁变更编译器代码，虚幻引擎使用NodeHandlers系统，在编译每个节点的时候，寻找对应的 NodeHandler来完成编译。如果你创建了一个继承自UK2Node的类，你可以通过重载CreateNodeHandler以返回处理当前节点的节点句柄处理器。

节点处理

自此，编译过程开始进入逐节点的处理阶段。各路NodeHandler开始登台显神威。目前虚幻引擎比较主要的节点句柄包括：

FKCHandler_CallFunction 处理函数调用节点。

FKCHandler_EventEntry 处理事件入口节点。

FKCHandler_MathExpression 处理数学表达式。

FKCHandler_Passthru 处理返回值节点。

FKCHandler_VariableSet 处理变量设置值节点。

说来难以相信，但是当你展开所有节点之后，蓝图的主要节点都可以被归并到这些节点中去。

接下来，对于每个节点，虚幻引擎都会通过 AppendStatementForNode 给其挂上 FBlueprintCompiledStatement，如图 13-2 所示。如前文所言，FBlueprintCompiledStatement 是一个独立的编译结构。在笔者前文中，这个结构被标记为需要重构（原文为：“//@TODO:Too rigid / icky design”），因此如果你看到的版本与笔者看到的不同，请见谅。毕竟虚幻引擎开发者们更新引擎代码的速度远远超过了笔者写分析的速度。

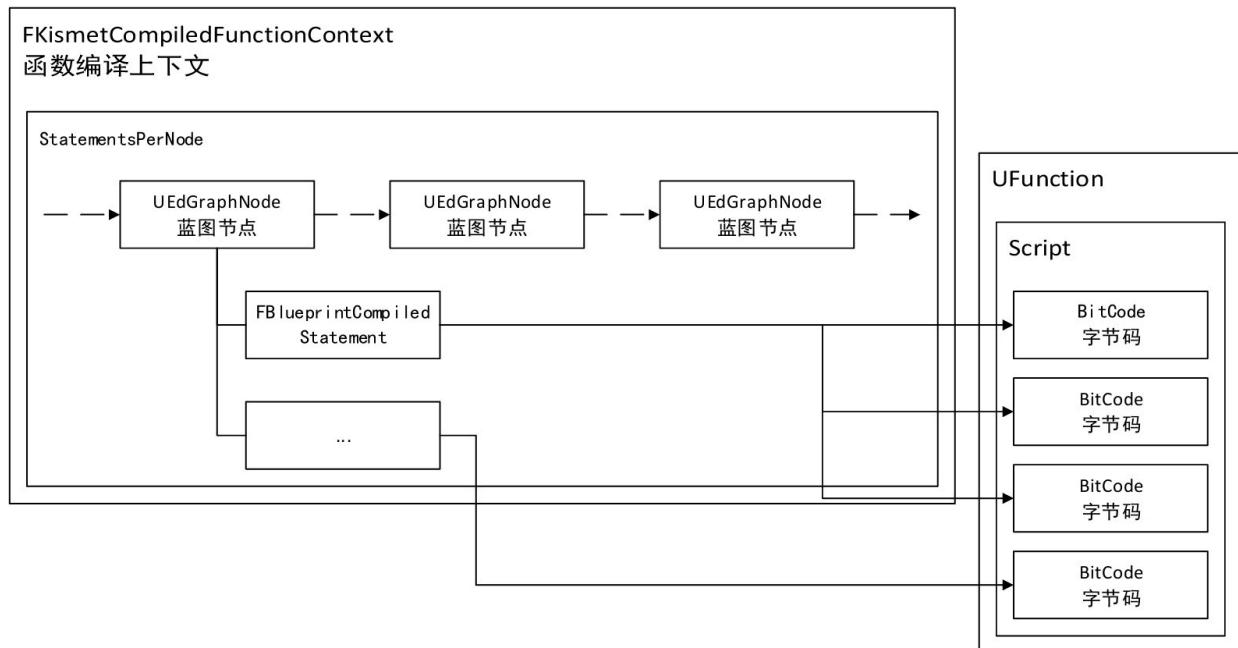


图13-2 后半部分：从函数上下文到最终脚本代码

而目前笔者看到的这个结构中，存储的信息包括：

EKismetCompiledStatementType 类型枚举。这个值非常重要。会在接下来的描述中分析。实质上这个值决定了接下来字段的启用与含义。可能这也是为何需要重构的原因。为了方便描述，在下面的成员变量介绍中对类型枚举的名字省略前缀“KCST”。

FunctionContext 函数上下文。

FunctionToCall 当前类型为CallFunction的时候启用，调用某个函数的时候该字段被赋值。

TargetLabel 当类型为Goto或者CallFunction的时候启用，指向跳转的目标Statement。

UbergraphCallIndex 当类型为CallFunction，且TargetLabel非空时启用，指向超大蓝图中的某个事件/函数的索引。

LHS 根据名称推测应为“左值目标”之类的含义，类型为FBPTerminal*，指向赋值语句的目标或者函数调用的结果。

RHS 同上，推测应为“右值目标”之类的含义，类型为FBPTerminal*，指向赋值语句的源数据，或者函数调用的参数列表。

bIsJumpTarget 这是否是一个跳转目标？

bIsInterfaceContext CallFunction启用，是否是一个接口上下文？

bIsParentContext CallFunction启用，是否是在父类中调用？

ExecContext WireTraceSite启用，即将被执行的Pin（具体意义笔者尚不清楚）。

Comment 注释。

实际上来说，FBlueprintCompiledStatement结构是一个“头”+“信息”的结构，如果你曾经学过汇编语言或者机器指令，那么你可以理解为：一个Statement就像是一个定长指令，包含开头的一个操作码与后面的操作参数。指令本身定长（即FBlueprintCompiledStatement结构大小固定），操作码不同，填充不同字段。

那么我们是时候来看EKismetCompiledStatementType这个枚举了。这个枚举指示了32种操作。下面笔者进行了部分翻译，为了方便阅读，笔者略去枚举前面的KCST前缀。

Nop 0号枚举，什么都没有。

CallFunction 函数调用指令，对应类C++代码的：

```
TargetObject->FunctionToCall(wiring)
```

Assignment 赋值指令。对应类C++代码的：

```
TargetObject->TargetProperty = [wiring]
```

CompileError 编译出错。

UnconditionalGoto 无条件跳转指令，就是C++的goto语句。跳转

到TargetLabel。对应类C++代码的：

```
goto TargetLabel
```

PushState 压栈。将TargetLabel对应的Statement压栈。对应类C++代码的：

```
FlowStack.Push(TargetLabel)
```

GotoIfNot 如果条件不成立则跳转，对应类C++代码的：

```
if (!TargetObject->TargetProperty)  
    goto TargetLabel
```

Return 返回指令。

EndOfThread 当前线程结束。对应类C++代码为：

```
if (FlowStack.Num())  
{  
    NextState = FlowStack.Pop;  
}  
else  
{  
    return;  
}
```

从这段代码可以看出，蓝图系统的所谓线程实际上是顺序执行的状态的集合。并行的多个执行序列实质上是通过压栈之后，逐个执行的。

Comment 注释。

ComputedGoto 计算后的Goto。这个指令对应的代码是：

```
NextState=LHS
```

笔者并不太明白为何这样对应。

EndOfThreadIfNo 如果条件不成立，则结束当前线程。对应代码为：

```
if (!TargetObject->TargetProperty)
{
    if (FlowStack.Num())
    {
        NextState = FlowStack.Pop();
    }
    else
    {
        return;
    }
}
```

CastObjToInterface 类型转换，将目标对象转换为指定的接口。

DynamicCast 类型转换，与上面这个不同的是，转换的目标是一个类。

ObjectToBool 对象转化为布尔值。实质上就是判断当前对象是不是None。

AddMulticastDelegate 增加一个多播代理。

ClearMulticastDelegate 清空目标多播代理。

BindDelegate 创建一个简单代理。

RemoveMulticastDelegate 移除一个多播代理。

CallDelegate 分发多播代理。

CreateArray 创建数组。

CastInterfaceToObj 通过指定接口获取对应对象。

GotoReturn 返回。

GotoReturnIfNot 如果条件不成立则返回。

SwitchValue 对应SwitchCase语句。

逐节点地创建了Statement之后，就开始了最终的后端编译。如果是使用UnrealScript的脚本字节码虚拟机，此时会逐函数地编译。首先创建一个FScriptBuilderBase，称为ScriptWriter，然后调用ScriptWriter的

`GenerateCodeForStatement`函数，对每个`Statement`生成字节码。构造`ScriptWriter`的时候会传入当前`UFunction`的`Script`变量的引用作为参数（该变量不在`UFunction`定义中，在`UStruct`定义中），在向`ScriptWriter`写入的时候，就自动地不断填充`UFunction`的`Script`脚本数组。

由于`Statement`作为中间环节，已经对语法进行了高度的整理以方便接下来的生成过程，因此这个`GenerateCodeForStatement`函数已经没有太多的语法分析之类的内容，蓝图字节码本身更类似于汇编式的执行方式，故而最终的字节码发射更类似于直接在`ScriptWriter`中填充字节码。具体字节码的定义太多，有兴趣的读者，可以自行打开。

在`Engine\Source\Runtime\CoreUObject\Public\UObject\Script.h`里面有完整的字节码值和含义对应。笔者不再过多分析的原因还包括，虚拟机字节码系统逐渐在被C++后端系统替代。说不定读者你拿到本书的时候，虚幻引擎已经放弃虚拟机字节码系统了。

一点后话：VM虚拟机调用

经过编译生成的`Script`，最终是如何被实际调用呢？如果读者对这个感兴趣，笔者可以简单解释一番，如果读者非常希望了解虚拟机的原理，可以查看本章最后部分关于虚拟机的介绍。编译完成的`Script`会被放在`UFunction`对应的`Script`数组中。同时当前`UFunction`的`FUNC_Native`标记不会被设置。因此在`UFunction`的`Bind`阶段（也就是将`UFunction`与所属类对接的时候），将会设置自身的`Func`指针指向`UObject`类的`ProcessInternal`函数，而非当前`UFunction`所属的`UClass`类中的C++函数表`NativeFunctionLookupTable`中对应的函数，如图13-3所示。

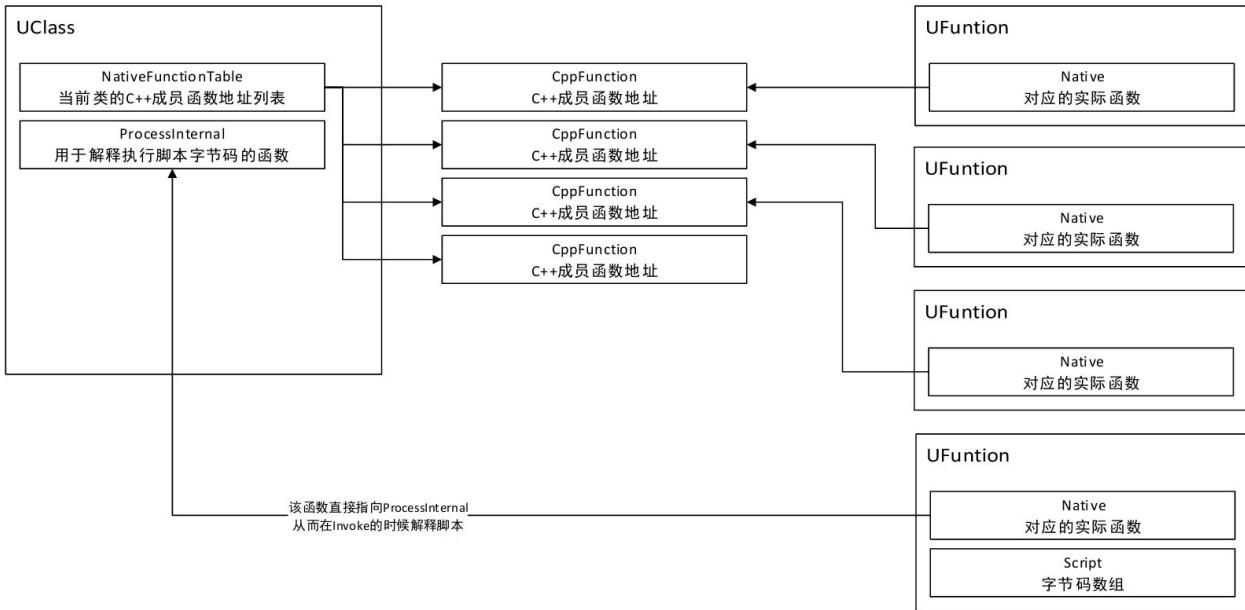


图13-3 UFunction和UClass的关系示意

在调用UFunction的Invoke函数时， UObject::ProcessInternal函数会被“当作”一个成员函数调用，去执行当前UFunction包含的脚本字节码。如果你不大熟悉如何调用一个UFunction，那么调用一个UFunction的写法如下：

```

UFunction* HandleChangeFunction = OwnerObject ->FindFunction(*
    SearchFunctionName);

if (HandleChangeFunction!=nullptr)
{
    FFrame Frame = FFrame(OwnerObject, HandleChangeFunction , NULL
    );
    HandleChangeFunction ->Invoke(OwnerObject, Frame, NULL);
}

```

此时会分配一个Frame作为调用的数据结构。如果此时UFunction是

一个脚本函数，而非一个Native函数，Frame会通过Step函数，一步一步解释执行脚本字节码。

那么有些读者可能执着地希望了解：字节码到实际机器码是如何完成的呢？

其实非常简单。有一个GNatives数组，里面是和VM操作码对应的执行函数。对于每一个操作码，以操作码为数组下标，取出来就是一个对应的执行函数，直接填充参数然后执行即可获取执行结果。这给人的感觉更像是一种解释器，而非一个最终编译为汇编机器码的执行前编译虚拟机。

借助IMPLEMENT_VM_FUNCTION宏就可以在GNatives中注册一个函数，你可以在ScriptCore中搜索这个宏，能找到每个字节码对应的具体执行函数定义。

小结

到这里笔者真的是舒了一口气，总算是将虚幻引擎的逻辑蓝图相关的内容向读者呈现完毕了。总而言之，这是一个涉及颇多方面的系统。从非常适合编辑的UEdGraph结构开始，逐步归并整理，以产生UClass结构。然后对逻辑相关的部分进行处理之后，不断向适合顺序执行的字节码结构靠拢，最终被Backend发射成为最终的字节码。

其实从实用主义的角度而言，知道如何向蓝图暴露函数、蓝图如何调用C++层的函数就已经能够使用蓝图完成大部分的开发工作了。进一步，如果有需要，希望用C++调用蓝图函数的知识也并不复杂。那么为

何非要了解蓝图的编译和工作方式呢？那是因为笔者希望读者能拥有扩展虚幻引擎本身的能力，针对项目的需求，扩展蓝图自身的节点，甚至创造自己的蓝图系统以应对特殊的需求（比如剧情分支树等）。另外，笔者也希望读者拥有对虚幻引擎自身工作机制的好奇心，以及对虚幻引擎本身进行研究的勇气。虚幻引擎的源码就在你的眼前，去了解它吧！不要惧怕什么！

13.4 蓝图虚拟机

如果读者曾经学习过Java，那么对“虚拟机”（VM）这样的概念应该并不陌生。虚幻引擎内置的，用于执行蓝图编译后的字节码的虚拟机继承自虚幻引擎3时代的UnrealScript虚拟机。

简单来说，虚幻引擎提供了一套基于字节码 和栈 的虚拟机。笔者试图简单地向读者解释一个与虚幻引擎设计接近的抽象虚拟机的工作原理，这是因为虚幻引擎本身的虚拟机有自身的优化、妥协与设计，这会让本来就复杂的系统变得更加复杂。

13.4.1 便笺纸与白领的故事

让我们先抛开复杂的虚幻引擎或者虚拟机的概念，来看一个笨拙的白领的故事。这个笨拙的白领是公司老板的孩子，没有任何工作经验，却被老板委以重任。老板想出了一个绝妙的方法来让他的孩子能够工作，如图13-4所示：

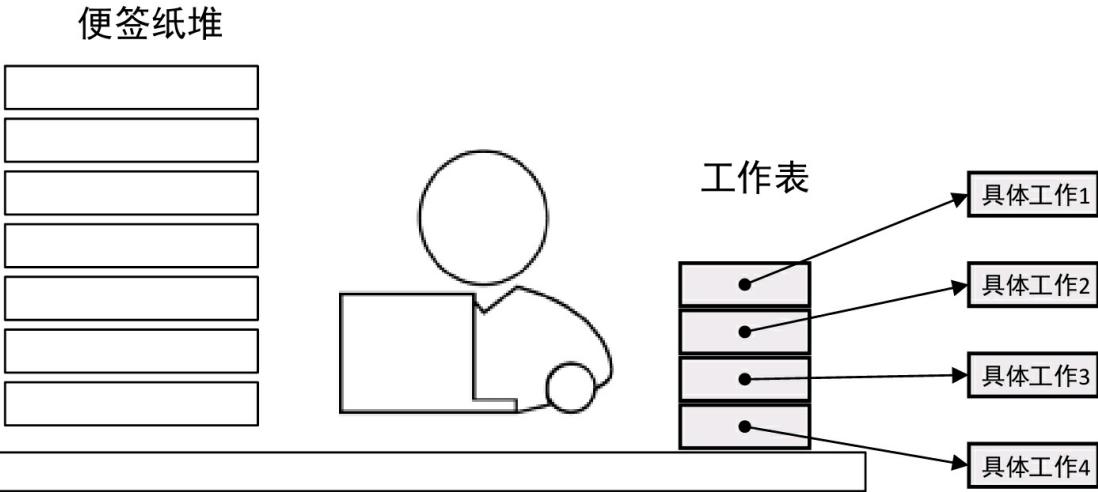


图13-4 白领虚拟机，右侧是工作表格，左侧是被细分成很小的步骤的便笺纸堆

1. 把他的工作分解为极其简单的小步骤，分别写在一张一张的便笺纸上，然后不停地放在桌子左边的一大堆便笺纸的最上边。
2. 在右侧放着一张简单的表格，这个表格从上到下编好了号，每一个格子都对应着“打钩”、“签字”、“把文件丢给隔壁桌”这样的简单指令。
3. 每个便笺纸上的小步骤都不用写文字，全都是一个编号。这个编号与表格中的某一个工作对应。
4. 老板给这个白领配了一大堆的助手，每个助手有一个编号，和右侧的工作表对应，且每个助手只负责一个任务，他们每天就等待这个白领喊自己的编号，一旦被喊，就走过去执行工作。
5. 这个白领每次从左边的便笺纸堆最下面抽出一张便笺纸，然后查找右侧的表格，找到了，就喊对应的助手过来执行任务。

接下来老板遇到了一定的困扰，假如是简单的指令，例如“打钩”，那么这个白领还能应付，但是如果需要提供“内容”（数据），例如“在纸上写字”这样的指令，就需要同时提供书写的內容，这可怎么办？这难不住智慧的老板，他发现可以这样解决问题：

规定：当遇到“在纸上写字”指令的时候，接下来的第一张便笺纸就是要写得文字！

于是当这个白领遇到“在纸上写字”的指令时，他会直接再抽出一张便笺纸，把这张便笺纸的内容理解（解释）为“字符串”这样的文本数据，然后递给那个可怜的助手，告诉他：给我写这上面的内容！——我们的白领还是一个文盲，他并不认识这张纸上的字到底是数字还是一段中文，如果我们的便笺纸堆顺序有误，他甚至会把这张纸上的字当作工作表中的某个索引去查询，毫不意外，他什么也找不到，于是就蠢蠢地愣在那里等待下班！或者，假如因为意外，下一张便笺纸并不是要写的字符串，而是其他的数字或者是乱七八糟的东西，忠实的助手还是会去抄写，只是写出的都是混乱的数据！

这可是一个重要的信息：便笺纸堆里面不仅有指令，还有数据！同样地，字节码中也装着数据！

但是这并没有解决问题，因为还有一个重复的指令：“在纸上写N遍字符串XXX”，这可需要两个参数：一个N、一个XXX字符串。怎么才能告诉助手这些信息呢？为了维护身为老板孩子的威严，除了喊编号，他可不能直接向助手说话——否则岂不是暴露了自己是文盲的秘密？这同样难不倒老板，老板立刻发布规定：

1. 在办公桌中央，还有一个用于交换便笺纸的临时便笺纸堆。
2. 每当遇到类似“在纸上写N遍字符串XXX”这样的指令时：
 - a. 白领从便笺纸堆底下抽出一张便笺纸A，放入临时便笺纸堆最上面。
 - b. 白领再从便笺纸堆底下抽出一张便笺纸B，放入临时便笺纸堆最上面。

c. 这个时候临时便笺纸堆情况如下：



d. 喊对应编号的助手过来。

3. 忠实的助手就走了过来，按照老板的规定，做这些事情：

- a. 拿出临时便笺纸堆最上面的一张（便笺纸B），解释为字符串XXX。
- b. 拿出临时便笺纸堆最上面的一张（便笺纸A），解释为书写的遍数N。
- c. 开始抄写。

4. 助手不准怀疑这些便笺纸的正确性，也不准窥探其他便笺纸是什么，只准拿走规定数量的便笺纸，做自己该做的事。

熟悉汇编语言的读者应该立刻就能明白，这是函数调用时通过压栈和出栈进行数据交换的方法。总而言之，笨拙的白领居然真的就这样干起了活儿，还做的有模有样，最苦的还是那个负责把任务拆分成一个一个小步骤的人（编译器）。

13.4.2 虚幻引擎的实现

上一个故事中的概念，实际上都映射一个真实虚拟机需要具备的工作，如表13-1所示：

表13-1 对应关系

故事中	真实虚拟机	虚幻引擎
笨拙的白领	虚拟机执行系统	FFrame
负责拆分指令的人	编译器	FKismetCompilerContext 等多个类
左手边的便笺纸堆	字节码数组	UStruct::Script
右手边的工作表	虚拟机字节码解释表	GNatives 数组
助手们	字节码执行函数	注册到 GNatives 中的每个函数
暂存便笺纸堆	虚拟机栈	无完全一致对应

前文已经提到，虚幻引擎的执行系统与这个故事描述的理想虚拟机存在一定的差异。对于真实的虚幻引擎来说，其执行系统被封装到 FFrame这个类中。这个类成员如下：

执行环境相关变量

Node 当前函数对应的蓝图节点。

Object 所属的对象，相当于C++执行中的this指针。

Code 编译完成的字节码数组，类型为uint8*，说明这是一个字节码数组。

Locals 局部变量，每一个UFunction会在这个区域分配和自己的局部变量长度相等的内存，然后在此构造UProperty类型的对象，作为局部变量使用。

辅助变量

MostRecentProperty 、 MostRecentPropertyAddress最近使用的变量及其地址。

FlowStack 执行流程栈。可以向内压入和弹出执行流程。这

是为了完成顺序执行多条流程这样的要求，实质上是每次压入多个执行流程的起始地址，然后每执行完毕一个流程，弹出一个地址，再执行下一个流程。

PreviousFrame 上一个帧。

OutParams 输出参数。

CurrentNativeFunction 当前正执行完毕的原生函数。

成员函数 最重要的成员函数即：获取字节码并执行的Step函数^[1]，和一系列的从字节码中读取数据的Read函数。

Step 每调用一次该函数，就会取出一个字节码，然后执行。

对应代码如下：

```
void FFrame::Step(UObject *Context , RESULT_DECL)
{
    int32 B = *Code++;
    (Context ->*GNatives[B])(*this , RESULT_PARAM);
}
```

可以看到，先通过Code指针获取当前字节码（顺便下移指针到下一个字节码），以此作为索引查找GNatives数组并获取对应的原生函数指针，然后调用。

Read系列函数 包括ReadInt、ReadFloat、ReadName、ReadObject、ReadWord、ReadProperty等，以ReadFloat为例，

删除对字节对齐进行判断的宏后，代码如下：

```
inline float FFrame::ReadFloat()
{
    float Result;
    Result = *(float*)Code;
    Code += sizeof(float);
    return Result;
}
```

可见这是直接将字节码数组的下一个float长度的数据读出，解释为float返回。印证了前文叙述的字节码不仅包含指令，还包含数据的说法。

由于虚幻引擎的蓝图系统存在“局部变量”的概念，因此暂存栈可以被更准确的局部变量绑定代替。

那么我们来看一个具体的实例：

准备

这会儿有一个Actor对象X，使用C++通过Actor::ProcessEvent函数调用X中的一个蓝图函数Func，开始进入到函数执行阶段。

执行

一个Frame对象会被创建出来，作为当前的执行帧，并控制字节码的执行。构造函数传入的比较重要的参数有以下两个：

UObject* InObject 所属的对象，包含执行所需要的成员变量数据。

UFunction* InNode 执行的函数，包含执行所需要的成员函数指令数据。

通过以下代码，Code成员变量将会被赋予函数所属的字节码数组的首地址指针：

```
Code(InNode->Script.GetData())
```

在本次执行的内容中，字节码数组总共有85个字节码元素。我们取出一小节来进行分析。

如何知道这些字节码对应什么函数呢？我们可以在运行时中断执行，然后查看GNatives数组，这个数组指明了每一个字节码对应的函数，需要注意的是，不能简单认为每一个字节码都对应函数，前文有描述，字节码是包含数据的。经过翻译，我们获得了这样的结果，如表13-2所示：

表13-2 字节码案例

字节码	函数名/数据
94	execTracepoint
90	execWireTracepoint
15	execLet
96	LocallyKnownProperty 在获取赋值目标失败的时候才会使用
250	
199	
109	
36	
2	
0	
0	
0	execLocalVariable
104	execCallMathFunction
128	数学函数地址
231	
244	
73	
36	
2	
0	
0	
...	...

我们重点关注15这个字节码。execLet是一个赋值指令，就像 $a=1+3$ 这样的表达式，那么其需要执行以下内容：

获得赋值目标 execLet调用Stack.Step函数来获取赋值目标，存入MostRecentPropertyAddress。我们这次遇到的是execLocalVariable指

令，表示将一个局部变量的值读取出来，具体来说，就是将当前Frame的Locals数组首地址转化为一个指针，复制给MostRecentPropertyAddress这个成员变量。

表达式求值并赋予 已经完成赋值目标的获得，接下来需要完成“值”的计算，故execLet再次调用Stack.Step函数，传入当前对象与赋值目标以完成真正的赋值。本次赋值函数对应了execMakeVector函数，也就是先组成一个FVector，再赋值给目标。

换句话说，Let指令执行的依然是类似 $a=1+3$ 的过程，先获得等号左边的地址，再将地址交给右边的表达式求值函数，用于让函数将计算完毕后的结果填充到对应地址。

13.4.3 C++函数注册到蓝图

当使用UFUNCTION宏定义了一个可以被虚拟机调用的函数时，UHT会自动帮助你生成一个看上去像这样的函数：

```
DECLARE_FUNCTION(exec[你的函数名])
{
    P_GET_TARRAY_REF(USceneComponent*,
    Z_Param_Out_AllComponet); //获取参数
    P_FINISH;
    P_NATIVE_BEGIN;
    this->GetAllSubCustomComponet(Z_Param_Out_AllComponet
    ); //实际调用区域
    P_NATIVE_END;
```

```
}
```

DECLARE_FUNCTION这个宏生成了一个“exec[你的函数名]”为名称的函数，参数分别为FFrame&与RESULT_DECL，请读者务必记住这里的定义，下文会用到。对应的宏定义为：

```
#define DECLARE_FUNCTION(func) void func( FFrame& Stack, RESULT_D  
)
```

剩下的宏主要作用就是从虚拟机中准备调用你的函数需要的参数环境等。随后在.generated.cpp中，这个函数会借助FNativeFunctionRegistrar::RegisterFunction函数，被注册到当前UClass类的NativeFunction列表中，并增加UFunction信息到UClass。

假如一个蓝图需要调用一个C++函数，编译器会设置字节码EX_FinalFunction，并将对应的UFunction指针同时放入到下一个字节码中，执行时就会调用：

```
CallFunction( Stack, RESULT_PARAM, (UFunction*)Stack.  
    ReadObject() );
```

从而将对应的exec函数所属的UFunction对象取出，CallFunction函数在检查到这是一个原生函数时，会直接调用UFunction::Invoke函数。在Invoke函数内部会调用当前UFunction函数的Func函数指针：

```
return (Obj->*Func)(Stack, RESULT_PARAM);
```

注意参数！这个函数就是上文中，UHT通过DECLARE_FUNCTION宏帮我们生成的exec函数！

简而言之，C++函数封装过程可以这样理解：

1. C++函数被UHT识别，生成包裹函数exec<函数名>。
2. exec<函数名>被FNativeFunctionRegistrar在运行前注册到UClass的函数列表中，创建UFunction信息。
3. 蓝图编译器在发现函数调用节点时，生成调用State，发现是原生函数，生成EX_FinalFunction字节码，并将对应的UFunction指针压栈。
4. 蓝图执行系统发现EX_FinalFunction字节码，于是读取下一段字节，解释为UFunction*。
5. 通过UFunction*的Invoke函数，调用exec<函数名>包裹函数。
6. 包裹函数拆包并准备参数列表，调用C++函数。

13.5 蓝图系统小结

蓝图一直是虚幻引擎富有争议的部分。无数的程序员诟病这个系统，认为极为不工程化、面向大型工程没有效率。同时Epic又极力推崇这个系统，致力于让蓝图能够完成更多的需求。也有许多人支持蓝图这个系统，认为蓝图在快速成形、高速迭代上具有自己的优势。

而实际而言，蓝图系统与虚幻引擎在运行时编译C++（HotReloadC++）系统，完成了传统的“脚本语言”需要完成的任务。运行时编译C++，允许游戏的底层逻辑能够更高效地被迭代验证，而蓝图

系统则承担了需要不断变化的关卡、特性逻辑部分。

最重要的一点是，蓝图系统极大降低了编译器校验的难度，其直接将类整理为了成员变量、成员函数、表达式这样的语义对象，从而降低了对语义进行上下文分析的难度。这从某种意义上说，也是对稳定性的提升。

笔者曾经使用UnrealScript系统，那时遇到过一个极难发现的BUG：当在UnrealScript的State状态块中，声明一个与类成员变量一模一样的变量时，编译器不会检查出错误。甚至连字节码虚拟机都没有检查出错误。直到对这个变量进行赋值时，整个虚拟机就崩溃了，没有给出任何有价值的报错信息。

蓝图通过自己的可视化语法来规避了这种问题。作为一个自行实现的脚本语言，Epic应当是有这方面的考虑。在笔者撰写本书时，蓝图系统已经能够通过编译为C++代码来降低运行时的性能消耗，因此蓝图最大的问题依然是工程上的，而非效率上的。本文只阐述蓝图的工作原理，而不对“蓝图是否好用”的问题作出一个定论性的回答。



蓝图字节码存储

前文有叙述，蓝图字节码在编译时，如果遇到对象指针，会直接存储指针值本身。这就涉及一个很严重的问题：序列化结束后，对象在内存中的位置会发生改变，原有的存储于字节码中的指针值会失效。虚幻引擎采用的方案是在存储之前再次遍历字节码数组，依次取出字节码过滤，遇到对象指针值时，替换为ImportTable或者ExportTable中的索引值。在反序列化时同样会重新遍历，并修正指针的值。

[1] 还有一系列的优化版的Step函数：StepExplicitProperty、
StepCompiledIn和StepCompiledInRef。

第三部分 扩展虚幻引擎

欢迎你，我的朋友！在阅读了这么多关于虚幻引擎的原理和机制的内容之后，终于有机会施展自己学到的知识，定制虚幻引擎来符合自己的需求。

第14章 引擎独立应用程序

问题

我该如何写出**UHT**这样，能够引用虚幻引擎的**API**，又不需要依赖引擎运行的程序呢？

14.1 简介

什么是引擎独立应用程序？让我们首先看一下虚幻引擎的Launcher运行器。不知道大家是否曾经好奇，虚幻引擎的Launcher运行器是什么写的呢？如果你曾经仔细观察过虚幻引擎的运行器的文件结构，你会惊讶地发现，这个文件结构非常类似于虚幻引擎本身的文件结构，而非虚幻引擎编译打包形成的游戏的文件结构。这意味着，虚幻引擎的运行器是一个微型虚幻引擎，而不是一个打包形成的游戏！

如何开发这样的东西呢？如果我们希望在一个小型的、不是游戏的应用程序中继续使用虚幻引擎的一部分API，那么我们就需要学习虚幻引擎运行器这样的技术。

14.2 如何开始

关于如何撰写这样的应用程序的介绍，虚幻引擎的官方文档语焉不详，几乎没有说明什么。但是虚幻引擎提供了几个案例性的程序，如BlankProgram和SlateViewer。我们可以从这几个应用程序着手进行分析和学习。

简而言之，开发这样的程序需要遵循以下步骤：

1. 建立文件结构。
2. 配置.Target.cs和.build.cs。
3. 撰写代码。
4. 剥离应用程序。

由于没有了虚幻引擎提供的打包机制，我们需要自己完成应用程序的剥离和发布。这个发布过程并不是一个简单地做加法的过程，而是一个做减法的过程。我们不是选择出哪些文件是我们需要的，并添加到程序中，而是选择出哪些文件是我们不需要的，从程序中删除。

那么，让我们首先把目光投向虚幻引擎源代码目录下的一个小小的文件夹，就是Source\Programs\BlankProgram。

14.3 BlankProgram

这个文件夹包含了以下的文件结构：

```
BlankProgram
|   BlankProgram.Build.cs
|   BlankProgram.Target.cs
|
└── Private
    BlankProgram.cpp
    BlankProgram.h
```

如何编译呢？你只需要打开源码版本的虚幻引擎的工程文件，然后运行解决方案窗口中的BlankProgram项目，你会看到弹出了一个命令行窗口，显示出了一行“HelloWorld”。

在对源码进行分析之前，我们必须先关注如何配置这个工程。怎么告诉UHT和UBT，我们需要编译出一个可以运行的.exe文件，而不是虚幻引擎的插件、模块或者游戏？

答案是：我们通过.Target.cs指定。如果你忘记了UBT的相关内容，可以翻一翻第二部分中，对UBT虚幻引擎构建工具的描述。由于虚幻引擎对源码引用的限制，笔者将会逐函数地分析.Target.cs的内容。你可以对照对应的文件，然后阅读笔者的讲解。比较重要的函数主要有SetupBinaries和SetupGlobalEnvironment。

```
public override void SetupBinaries(
    TargetInfo Target,
    ref List<UEBuildBinaryConfiguration>
        OutBuildBinaryConfigurations ,
    ref List<string> OutExtraModuleNames
)
```

```
{  
    OutBuildBinaryConfigurations.Add(  
        new UEBuildBinaryConfiguration( InType:  
            UEBuildBinaryType.Executable,  
            InModuleNames: new List<string>() { "  
                BlankProgram" } )  
    );  
}
```

在SetupBinaries中，我们指定了两样信息：

- 需要被引入的模块名，在这里叫BlankProgram，指向的是.build.cs中定义的BlankProgram模块。
- 需要指定引入模块的编译类型，这里编译的类型是UEBuildBinaryType.Executable。

第二个编译类型指定非常重要，意味着我们的模块将不再被编译为二进制数据模块，用于编译链接，而是编译为可执行的exe应用程序。换句话说，对SetupBinaries函数的重载，完成了对编译为exe应用程序的指定。

```
public override void SetupGlobalEnvironment(  
    TargetInfo Target,  
    ref LinkEnvironmentConfiguration  
        OutLinkEnvironmentConfiguration ,  
    ref CPPEnvironmentConfiguration  
        OutCPPEnvironmentConfiguration
```

```
)  
{  
    UEBuildConfiguration.bCompileLeanAndMeanUE = true;  
    BuildConfiguration.bUseMallocProfiler = false;  
    UEBuildConfiguration.bBuildEditor = false;  
    UEBuildConfiguration.bBuildWithEditorOnlyData = true;  
    UEBuildConfiguration.bCompileAgainstEngine = false;  
    UEBuildConfiguration.bCompileAgainstCoreUObject =  
        false;  
    OutLinkEnvironmentConfiguration.  
        bIsBuildingConsoleApplication = true;  
}
```

笔者删除了注释以符合虚幻引擎的源码引用协议。这里我们能够看到7个控制变量的设定，具体含义如下：

1. **bCompileLeanAndMeanUE**: 这个控制变量表示编译为“部分”虚幻引擎，也就是非完整的虚幻引擎。
2. **bUseMallocProfile**: 这个控制变量表示是否使用内存分配剖析器。BlankProgram关闭了内存分配剖析。虚幻引擎给出的理由是，UHT有可能会在虚幻引擎本身编译之前被编译（实际上经常就是这样）。如果先于引擎本身编译，同时又打开内存剖析，则UHT会出现异常工作。同样地，BlankProgram也是一样。
3. 接下来是两个BuildEditor控制。我们不需要一个编辑器，但是我们需要编辑器相关的数据。
4. **bCompiledAgainstEngine**和**bCompileAgainstCoreUObject**: 这两个标志控制了是直接静态链接还是动态链接的方式来与Engine模块和

CoreUObject模块链接。这里BlankApp采用动态链接的方式以降低体积。

5. bIsBuildingConsoleApplication: 是否编译为控制台命令程序。这实质是指定程序入口点。比如对于控制台应用程序，入口点将会是Main函数。

在此基础上，我们的HelloWorld程序就会显得比较简单，核心代码如下：

```
#include "BlankProgram.h"
#include "RequiredProgramMainCPPInclude.h"
DEFINE_LOG_CATEGORY_STATIC(LogBlankProgram, Log, All);
IMPLEMENT_APPLICATION(BlankProgram, "BlankProgram");
INT32_MAIN_INT32_ARGC_TCHAR_ARGV()
{
    GEngineLoop.PreInit(ArgC, ArgV);
    UE_LOG(LogBlankProgram, Display, TEXT("Hello World"));
    return 0;
}
```

整个代码中包含的内容不多，相对来说，Main函数中的内容理解起来比较简单，即预初始化引擎来完成日志系统初始化，从而支持UE_LOG宏，输出Hello World。为了配合UE_LOG宏，我们需要定义log category，所以需要在最前方书写DEFINE_LOG_CATEGORY_STATIC宏。

此时相对来说陌生的只有两个部分：

1. RequiredProgramMainCPPInclude头文件的内容。
2. IMEPLEMENTAPPLICATION宏的含义。

对于前者，这个头文件包含了：

ModuleManager.h 模块管理器头文件，提供注册模块需要的辅助类，这是提供给IMPLEMENT_APPLICATION宏使用的。

LaunchEngineLoop.h 提供我们用的GEngineLoop定义。

LaunchEngineLoop.cpp 头文件包含cpp显得非常奇怪，根据官方注释，cpp提供了一些需要的定义。

对于后者，实际上可以理解为和之前介绍的模块实现宏IMPLEMENT_MODULE一样，提供了模块的注册实现。

14.4 走得更远

整个程序就打印了一行日志输出，也许很多读者颇为怀疑笔者是不是在逗你玩。请勿慌张，先看看我们已经拥有了什么：

1. 可以引入虚幻引擎已有的运行时模块。
2. 可以包含虚幻引擎的类定义，从而使用虚幻引擎提供的公共API。

在这个基础上，我们能走得更远：我们可以开发出一个引擎独立的Slate应用程序。若读者有兴趣，请按照以下步骤操作。

14.4.1 预先准备

请读者尽可能准备一个源码版的引擎进行试验。如果不知道如何获取Github上的虚幻引擎源码，请查看虚幻引擎官网上关于如何获取源码的资料。

如果读者不介意直接修改BlankProgram，那就可以直接对BlankProgram代码进行操作（可以预先备份到别的文件夹）。否则，就需要复制一份BlankProgram，然后重新命名文件夹名字，并对.Target.cs和.Build.cs中对应的名字进行修正。

笔者更推荐前一种方案。

14.4.2 增加模块引用

在.Build.cs文件的PrivateDependencyModuleNames中，添加到Slate、SlateCore和StandaloneRenderer模块的引用。前两者是为了提供Slate访问的接口，很好理解，后者是为了让Slate能够独立于整个引擎渲染（可以看作一个轻量级的Slate窗口渲染器）。当你修改完之后的代码应该是这样：

```
PrivateDependencyModuleNames.AddRange(  
    new string[] {  
        "Core",  
        "Slate",  
        "SlateCore",  
        "StandaloneRenderer"  
    }  
)
```

14.4.3 添加头文件引用

在BlankProgram.h头文件中加入：

```
#include "SlateBasics.h"
#include "StandaloneRenderer.h"
#include "SlateApplication.h"
```

14.4.4 修改Main函数为WinMain

为了能够创建一个Windows窗口，我们必须要让引擎的入口点设置为WinMain，修改Main函数定义为：

```
int
WINAPI WinMain(
    _In_ HINSTANCE hInInstance ,
    _In_opt_ HINSTANCE hPrevInstance ,
    _In_ LPSTR,
    _In_ int
nCmdShow
)
```

14.4.5 添加LOCTEXT_NAMESPACE定义

在BlankProgram.cpp最开头添加上一行LOCTEXT_NAMESPACE定义，内容随意，这是为了适配虚幻引擎的本地化机制。

14.4.6 添加SlateStandaloneApplication

此时GEngineLoop的初始化会缺少参数，因此需要略微修改，之后添加以下代码以启动SlateApplication，并进入消息循环：

```
GEngineLoop.PreInit(GetCommandLineW());  
FSlateApplication::InitializeAsStandaloneApplication(  
    GetStandaloneRenderer());  
while(!GIsRequestingExit)  
{  
    FSlateApplication::Get().Tick();  
    FSlateApplication::Get().PumpMessages();  
}
```

14.4.7 链接CoreUObject

由于Slate需要CoreUObject模块中的一些定义，因此需要在.Target.cs文件中把链接选项设置为true，同时顺便也把编译为命令行应用程序设置为false，即：

```
UEBuildConfiguration.bCompileAgainstCoreUObject = true;  
OutLinkEnvironmentConfiguration.bIsBuildingConsoleApplication = f
```

14.4.8 添加一个Window

为了让我们的Slate系统打开一个窗口，我们需要手动增加一个Window。在FSlateApplication初始化后，添加以下代码：

```
TSharedPtr<SWindow> MainWindow =  
    SNew(SWindow)  
        .ClientSize(FVector2D(800, 200));  
    FSlateApplication::Get().AddWindow(MainWindow.ToSharedRef());
```

14.4.9 最终代码

如果一切顺利，最终完成的代码应该是这样：

```
#pragma once  
  
#include "Core.h"  
#include "SlateBasics.h"  
#include "StandaloneRenderer.h"  
#include "SlateApplication.h"  
  
#include "BlankProgram.h"  
#include "RequiredProgramMainCPPInclude.h"  
DEFINE_LOG_CATEGORY_STATIC(LogBlankProgram, Log, All);
```

```
IMPLEMENT_APPLICATION(BlankProgram , "BlankProgram");

int
WINAPI WinMain(_In_ HINSTANCE hInInstance, _In_opt_ HINSTANCE
    hPrevInstance, _In_ LPSTR, _In_ int nCmdShow)
{
    GEngineLoop.PreInit(GetCommandLineW());
    FSlateApplication::InitializeAsStandaloneApplication(
        GetStandardStandaloneRenderer());
    TSharedPtr<SWindow> MainWindow =
        SNew(SWindow)
        .ClientSize(FVector2D(800, 200));
    FSlateApplication::Get().AddWindow(MainWindow.ToSharedRef());
    while (!GIsRequestingExit)
    {
        FSlateApplication::Get().Tick();
        FSlateApplication::Get().PumpMessages();
    }
    return 0;
}
```

退出Visual Studio，然后进入你的源码版引擎目录，运行里面的GenerateProjectFiles批处理文件，以调用UBT更新工程文件的设置与依赖。笔者目前不确定这一步是否必须执行，但笔者每次都是用这样的方式刷新了工程文件。

在BlankProgram（或者你自己复制出来的项目）上单击鼠标右键，设置为启动项目，然后单击鼠标右键，构建当前项目。构建完成后，直

接运行。不出意外，你会看到一个弹出的Slate窗口，如图14-1所示。

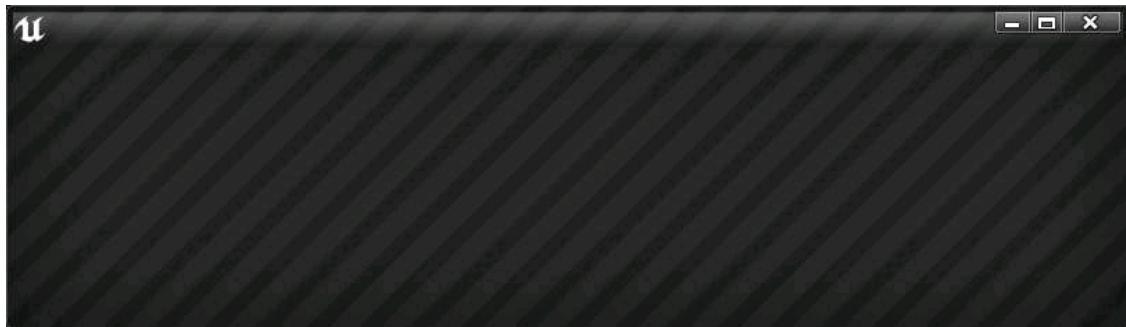


图14-1 激动人心的Slate窗口，它没有依赖引擎来运行

14.5 剥离引擎独立应用程序

此时就有读者想得更远，希望能够不依赖虚幻引擎运行自己写好的引擎独立应用程序。同时也为了验证笔者刚刚说的是不是真的——真的不需要引擎来运行吗？

一个最简单的方式就是去\Engine\Binaries\Win64文件夹下找到BlankProgram.exe，拷贝到你的Epic运行器文件夹对应目录。没错，就是虚幻引擎那个登录账号的运行器的目录。在笔者电脑上，它位于EpicGames\Launcher\Engine\Binaries\Win64。双击一下，你依然会发现程序能正常工作，验证了笔者的说法：你的程序不需要依赖一个引擎也能运行。

需要注意的是，这种应用程序依赖特定文件结构和Shader着色器，所以如果你真的希望把你的程序交给没有虚幻引擎的人使用，笔者衷心建议直接打包运行器，作为已经配置完毕的依赖，会大幅度简化你的工作。

第15章 插件开发

问题

我希望写一个插件，运用我学到的知识来扩展虚幻引擎！我该怎么做？

15.1 简介

插件系统作为虚幻引擎的重要组成部分，被赋予了非常多的重任。不严谨地说，“插件”扛起了虚幻引擎的三分之一边天。在我们的日常使用中，几乎一直在跟插件打交道。比如用于版本控制的Git、Subversion、P4插件；用于VR的GearVR、Oculus、SteamVR插件；用于移动平台多媒体播放的各类Media Player插件等。截至目前为止，虚幻引擎官方自带的插件已经有88个！其中有些是虚幻官方开发的，而有些则是第三方公司、团队开发后被纳入官方主版本库的。

15.2 开始之前

“既然插件这么厉害，那我们赶紧开始吧！”

看完刚才的简介，想必你也是非常激动。不过常言道冲动是魔鬼。

在我们开始学习虚幻引擎插件开发之前，笔者觉得有必要跟你说明一些问题。为了通俗点，接下来给大家讲一个段子。

青年问禅师系列

青年1：在嘛？

禅师：在，请说。

青年1：听说学虚幻引擎4的，会写插件才能算是一个完整的人，是一个脱离了低级趣味的人。插件能分分钟做出一些毁天灭地的东西出来。

禅师：额……

青年1：我想问一下怎么写虚幻引擎4插件？

禅师：你读过《提问的智慧》么？

青年1：没读过。

禅师：下一位！

青年2：我读过《提问的智慧》。

禅师：好吧，你想学插件的动机是什么？

青年2：我在做项目的时候，发现蓝图不够用了。想通过插件实现一些蓝图做不了的事。禅师：嗯，目的很明确。

禅师：那么，简单阐述下弱指针与强指针的区别。

青年2：指针是什么，缝衣服的针？

禅师：下一位！

青年3：禅师，我精通C和C++，也知道弱指针与强指针是什么。

禅师：嗯，不错。基础可以。

青年3：谢谢禅师。

青年3：听说虚幻引擎以前授权要几百万元人民币，现在开源了。我想研究虚幻引擎源码。

禅师：用过蓝图吗？

青年3：蓝图是那种给美术策划用的东西，说实话，我不屑于用它。

禅师：不建议一上来就接触源码。你应该先用蓝图系统熟悉虚幻引擎的整套机制，再研究源码。这样理解起来更容易。

禅师：下一位！

青年4：禅师，我会C++，也会蓝图。

禅师：那么你有什么问题？

青年4：我有一个项目，用蓝图开发的。但后期发现性能不行。我想优化它。

禅师：性能问题直接把蓝图项目转成C项目。把性能瓶颈的那部分抽出来用C实现就可以了。并不需要写成插件。

禅师：下一位！

15.3 创建插件

15.3.1 引擎插件与项目插件

虚幻引擎4的插件有两种：引擎插件和项目插件。引擎插件放在Engine\Plugins目录下，而项目插件则放在项目根目录(.uproject同级目录)下的Plugins中。那么引擎插件与项目插件有什么区别呢？用一种不太严谨的说法就是，把项目插件放到引擎插件目录下它就成了引擎插件。同理，引擎插件放到项目里它就成了项目插件。另外，引擎插件有一点特别的是它可以被加载到使用此引擎的所有项目里，而项目插件只能在当前项目里用。

引擎插件与项目插件除了存放的目录不相同外，其他差别不大。所以，在这里只介绍项目插件如何创建。

创建一个完整的插件需要创建插件配置文件、模块配置、创建源码目录等操作。所幸的是，虚幻引擎4提供了插件创建工具，并提供了几种常见的插件模板，用于我们快速创建插件。

使用方法为：执行“Editor”→“Plugins”命令，在打开的窗口右下角单击“New Plugin”，选择相应的插件模板。

15.3.2 插件结构

笔者使用Blank模板创建了一个名叫“MyBlank”的插件。插件工具随后生成了相应的文件，如图15-1所示。

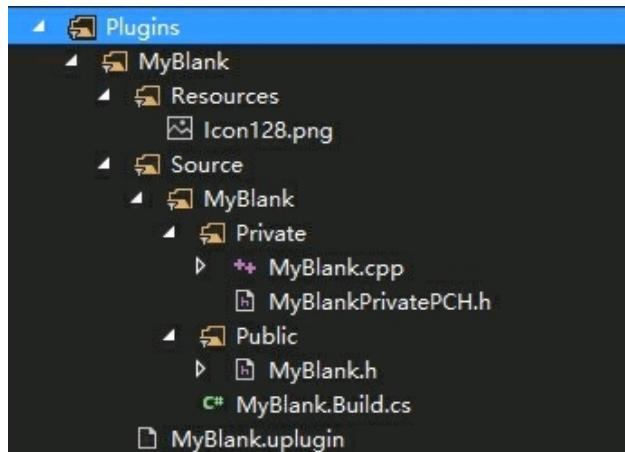


图15-1 MyBlank插件

下面我们来初步认识一下一个基础插件所包含的这些文件都有什么作用。

.uplugin

在插件目录最外层的是.uplugin文件。在虚幻引擎4启动的时候，会在Plugins目录里面搜索所有的.uplugin文件，每个.uplugin代表一个插件。

```
{  
    "FileVersion": 3,  
    "Version": 1,  
    "VersionName": "1.0",  
    "FriendlyName": "MyBlank",  
    "Description": "",  
    "Category": "Other",  
    "CreatedBy": "",  
    "CreatedByURL": "",  
    "DocsURL": "",  
    "MarketplaceURL": "",  
    "SupportURL": "",  
    "EnabledByDefault": false,  
    "CanContainContent": false,  
    "IsBetaVersion": false,  
    "Installed": false,  
    "Modules": [  
        {  
            "Name": "MyBlank",  
            "Type": "Developer",  
            "LoadingPhase": "Default"  
        }  
    ]  
}
```

```
    ]  
}  
]
```

以上为MyBlank.uplugin的内容。可以看出，它是一个.json格式的文件。前几条为插件的基本介绍，用于显示在插件面板。比如重要的是“Modules”的配置。“Modules”即模块，一个插件可以包含多个模块，如果要添加新的模块，只需要在Modules里面添加即可。写法如下：

```
"Modules": [  
    {  
        "Name": "MyBlank",  
        "Type": "Developer",  
        "LoadingPhase": "Default"  
    },  
    {  
        "Name": "another",  
        "Type": "Developer",  
        "LoadingPhase": "Default"  
    }  
]
```

每个模块，需要配置名字、类型和加载阶段。根据需要的类型(Type)可以配置成以下类型：

1. **Runtime** 在任何情况下都会加载。
2. **RuntimeNoCommandlet** Runtime模式但不包含命令。

3. **Developer** 只在开发（Development）模式和编辑模式下加载，打包（Shipping）后不加载。
4. **Editor** 只在编辑器启动时加载。
5. **EditorNoCommandlet** Editor模式但不包含命令。
6. **Program** 独立的应用程序。

加载阶段（LoadingPhase） 用于控制此模块在什么时候加载和启动， 默认为“Default”， 一般情况下此项不需要配置。可以用 **PreDefault**、**PostConfigIni** 。 PostConfigIni将会让此模块在虚幻引擎关键模块加载前加载。而PreDefault则是让模块在一般模块前加载。通常情况下，这个配置只用于你的游戏模块直接引用了插件模块里的资源或者用到了插件里面的类型定义。

还有两个配置在模板里没有体现，分别是**WhitelistPlatforms** 和 **BlacklistPlatforms**， 即平台白名单和黑名单。因为虚幻引擎4是跨平台的， 所以可以用这两个配置定义模块只能在哪些平台被加载或在哪些平台不加载。

Resources 目录 即资源目录， 通常情况下其里面至少包含此插件的图标。

Source 目录 即此插件的源码目录， 通常情况下，在Source目录里， 每个目录代表一个模块。如刚才创建的插件，在Source目录下就只有一个名为“MyBlank”的目录， 即“MyBlank”模块（模块名与插件名可以相同）。在MyBlank目录下， 又包含MyBlank.Build.cs文件和Public、Private两个目录。通常头文件放在Public目录下， 源文件放在Private目录下。

.Build.cs .Build.cs文件为模块的配置文件。此配置文件为.cs文件（即微软的C#语言），所以它可以支持包括C#的打印输出、变量定义、函数定义等特性。通常情况下，此配置文件在实际开发中，我们最常配置的是**PrivateDependencyModuleNames** 和 **PublicDependencyModuleNames**，即私有依赖与公共依赖。简而言之就是，我们写的源码，如果需要引用其他模块，则需要对其进行依赖。如果源码在Private目录下，则需要添加相应的私有依赖，Public亦然。

PCH PCH.h文件为预编译头，通常在模块的Private目录下。关于预编译头的相关知识请自行查询。这里需要注意的是，在虚幻引擎4中，要求所有源文件 (.cpp文件) 都必须在第一行Include此模块的PCH文件中。

15.3.3 模块入口

虚幻引擎4的模块继承自IModuleInterface，在IModuleInterface中包含7个虚函数：**StartupModule**、**PreUnloadCallback**、**PostLoadCallback**、**ShutdownModule**、**SupportsDynamicReloading**、**SupportsAutomaticShutdown**、**IsGameModule**，其中**StartupModule**为模块入口。

15.4 基于Slate的界面

15.4.1 Slate简介

Slate是虚幻引擎中的自定义用户界面系统的名称。目前的编辑器中

的界面大部分都是使用Slate来构建的。同时Slate也可以作为我们游戏开发时的UI框架（除了UMG以外又多了一项选择）。以及虚幻引擎4的UMG系统也是由Slate构建出来的。学会Slate有助于我们更深入地理解虚幻引擎4的GUI架构。不管是用于项目开发，还是插件编写，都可以用到它。

所以在本节中，笔者将带大家深入了解Slate。

15.4.2 Slate基础概念

关于Slate的基础概念，在官方文档

(<https://docs.unrealengine.com/latest/CHN/Programming/Slate/index.html>)里已经有详细的解释。在这里，笔者将主要通过实例代码来讲解。希望读者在看完这些内容后，能把官方文档再详细地阅读一遍。

15.4.3 最基础的界面

创建最简单的Slate界面的方法是创建一个带界面的插件。因为不管如何，我们要使用Slate，需要引用一些Slate模块、引用头文件等。而利用虚幻引擎4编辑器创建插件模板，会帮我们完成这些初始化工作。在这里我们主要是关注Slate本身就可以了。

创建带界面的插件，如图15-2所示。

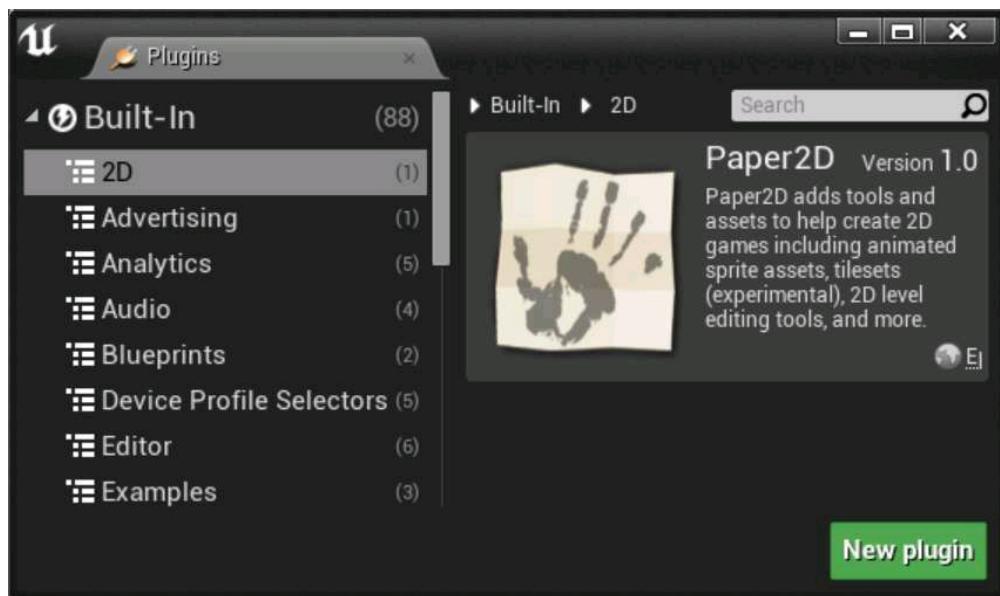


图15-2 创建插件

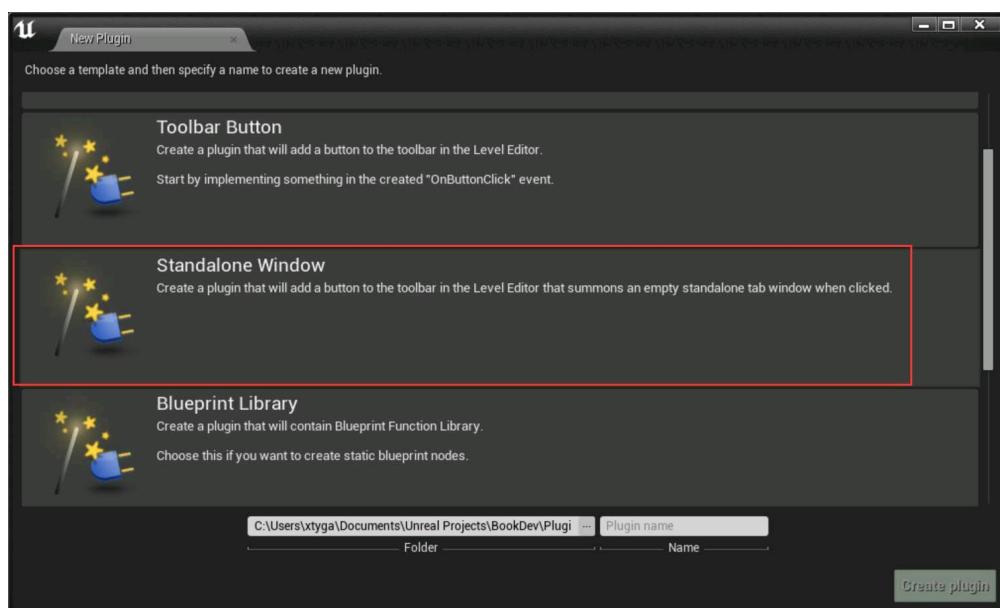


图15-2 创建插件（续）

输入一个名字后，点击创建即可生成插件模板代码。然后切换到VS，会提示解决方案有更新。点击“全部覆盖”即可。

编译项目后运行，在虚幻引擎4编辑器的工具栏上会多一个按钮。点击即会出现一个窗口，如图15-3所示。

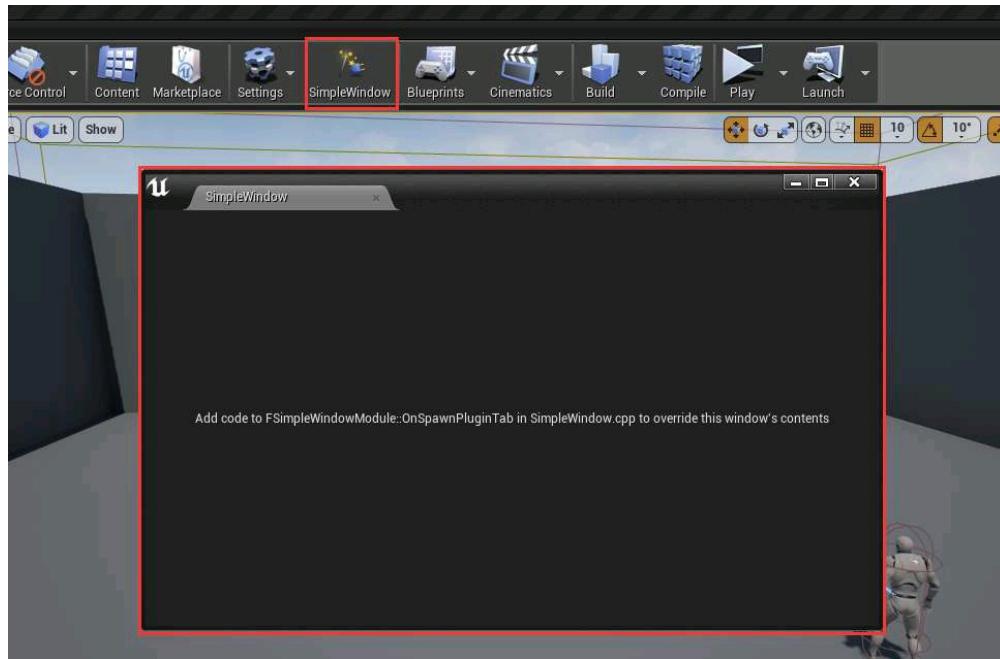


图15-3 生成窗口

至此，我们已经成功创建了一个点界面的插件。接下来，我们来简单分析一下，生成界面插件的一些过程。

首先，从目录结构上可以看出。在Public目录下，生成了4个文件。因为笔者创建的插件名叫“SimpleWindow”。所以模板生成的4个文件名分别叫SimpleWindow.h、SimpleWindowCommands.cpp、SimpleWindowCommand.h和SimpleWindowStyle.h（插件名不同，这几个文件名会有相应的变化）。

SimpleWindow 类为插件主类，插件入口与出口都在此类中。

SimpleCommand 类注册了一个OpenPluginWindow的命令。

SimpleWindowStyle 类则是定义UI样式的类。

这里着重分析一下StartupModule入口函数。StartupModule会在模块

被载入时调用。

```
void FSimpleWindowModule::StartupModule()
{
    //初始化UI样式
    FSimpleWindowStyle::Initialize();
    FSimpleWindowStyle::ReloadTextures();
    //注册命令
    FSimpleWindowCommands::Register();
    //实例化一个命令，并进行行为绑定
    PluginCommands = MakeShareable(new FUICommandList);
    PluginCommands->MapAction(
        FSimpleWindowCommands::Get().OpenPluginWindow,
        FExecuteAction::CreateRaw(this, &FSimpleWindowModule::
            PluginButtonClicked),
        FCanExecuteAction());
    //获取LevelEditor模块的引用
    FLevelEditorModule& LevelEditorModule
        = FModuleManager::LoadModuleChecked<FLevelEditorModule>("LevelEditor");
    {
        //创建一个菜单扩展，并把它添加到场景编辑器里
        // MenuExtender就是菜单栏
        TSharedPtr<FExtender> MenuExtender = MakeShareable(new
            FExtender());
        MenuExtender ->AddMenuExtension(
```

```
        "WindowLayout",
        EExtensionHook::After,
        PluginCommands ,
        FMenuExtensionDelegate::CreateRaw(this, &
            FSimpleWindowModule::AddMenuExtension));
LevelEditorModule.GetMenuExtensibilityManager()->AddExtender(
    MenuExtender);
}

{
    //创建一个工具栏扩展，并添加到工具栏
    TSharedPtr<FExtender> ToolbarExtender = MakeShareable(new
        FExtender);
    ToolbarExtender ->AddToolBarExtension(
        "Settings", EExtensionHook::After,
        PluginCommands ,
        FToolBarExtensionDelegate::CreateRaw(this, &
            FSimpleWindowModule::AddToolbarExtension));
    LevelEditorModule.GetToolBarExtensibilityManager()
        AddExtender(ToolbarExtender);
}

//注册一个Tab容器
FGlobalTabmanager::Get()->RegisterNomadTabSpawner(
    SimpleWindowTabName , FOnSpawnTab::CreateRaw(this, &
        FSimpleWindowModule::OnSpawnPluginTab))
    .SetDisplayName(LOCTEXT("FSimpleWindowTabTitle", "SimpleWindow"))
    .SetMenuType(ETabSpawnerMenuType::Hidden);
```

```
}
```

在StartupModule里只是完成了初始化工作。那么具体流程是什么样的呢？

首先，我们在工具栏点击插件的按钮时，因为ToolbarExtnder与PluginCommands关联，而PluginCommands的执行函数是**FSimpleWindowModule::PluginButtonClicked()**，所以，当点击工具栏上的按钮后，最终执行了PluginButtonClicked()函数。

在**PluginButtonClicked** 里，调用了FGlobalTabmanager的Invoke函数。因为在StartupModule里，已经注册了这个Tab，所以，Invoke会启动这个Tab。而在注册的时候，绑定了Tab生成的处理函数为OnSpawnPluginTab()。

在**OnSpawnPluginTab** 函数内部，生成了具体的UI。

```
FText WidgetText = FText::Format(
    LOCTEXT("WindowWidgetText", "Add code to {0} in {1} to
        override this window's contents"),
    FText::FromString(TEXT("FSimpleWindowModule::OnSpawnPluginTab
        ")),
    FText::FromString(TEXT("SimpleWindow.cpp"))
);

SAssignNew(TestButtonPtr, SButton);
TestButtonPtr = SNew(SButton);
return SNew(SDockTab)
```

```
.TabRole(ETabRole::NomadTab)
[
    SNew(SBox)
        .HAlign(HAlign_Center)
        .VAlign(VAlign_Center)
    [
        SNew(STextBlock).Text(WidgetText)
    ]
];
```

接下来就这段生成UI的代码，我们开始正式进入Slate代码的学习。

15.4.4 SNew与SAssignNew

就像创建一个新的UObject对象是用NewObject()一样。在Slate中，创建一个新的UI，有**SNew** 与**SAssignNew** 两种方式。

SNew与SAssignNew的主要区别是返回值类型不一样。SNew返回**TSharedPtr**，而SAssignNew返回**TSharedRef**。

一般我们需要存储一个UI对象，会在头文件里声明一个变量用于记录。如在SimpleWindow.h中添加以下一行代码：

```
//声明一个Button类型的变量
TSharedPtr<SButton> TestButtonPtr;
```

在cpp文件中实例化这个按钮的时候，有两种方法。

```
SAssignNew(TestButtonPtr , SButton);  
TestButtonPtr = SNew(SButton);
```

可能细心的读者会发现，TestButtonPtr类型不是TSharedPtr吗？
SNew的返回值是TSharedRef。类型不一样应该会报错的吧！但其实不会，在虚幻引擎4中**TSharedRef**可以直接转换成**TSharedPtr**，但这个转换不是双向的，TSharedPtr转换成TSharedRef需要用到AsShared()函数。

下面来演示一下：

```
SNew(SBox)  
    .HAlign(HAlign_Center)  
    .VAlign(VAlign_Center)  
    [  
        //TestButtonPtr为TSharedRef所以要用AsShared进行转换  
        TestButtonPtr ->AsShared()  
    ]  
//////////  
//////////  
SNew(SBox)  
    .HAlign(HAlign_Center)  
    .VAlign(VAlign_Center)  
    [
```

```
//SNew返回值是TSharedRef  
SNew(SButton)  
]
```

15.4.5 Slate控件的三种类型

在Slate中，控件分为三种类型：

1. **Leaf Widgets** 不带子槽的控件。如显示一块文本的STextBlock。
2. **Panels** 子槽数量为动态的控件。如垂直排列任意数量子项，形成一些布局规则的SVerticalBox。
3. **Compound Widgets** 子槽显式命名、数量固定的控件。如拥有一个名为Content的槽（包含按钮中所有控件）的SButton。

在刚才的代码中，我们所演示的控件有**SDockTab**、**SButton**、**SBox**。这三种控件都属于**CompoundWidgets**。

观察以下这段代码的写法，会发现一个共性：这三种控制都是用中括号来指定具体的内容。

```
SNew(SDockTab)  
[  
    SNew(SBox)  
    [  
        SNew(SButton)  
        [
```

```
        SNew(STextBlock)
    ]
]
];
```

那么是否可以说，Slate的容器添加内容都是以这种中括号的形式来添加呢？答案是不一定。我们先来看一下，这种中括号的方法是如何来实现添加子控件的。

以下内容涉及比较深入的内容，不理解也没关系。以SButton为例，在SButton.h中：

```
/**
 * Slate's Buttons are clickable Widgets that can contain arbitrary
 * widgets as its Content().
 */
class SLATE_API SButton
    : public SBorder
{
public:
    SLATE_BEGIN_ARGS( SButton )
        : _Content()
        , _ButtonStyle( &FCoreStyle::Get().GetWidgetStyle< FButtonStyle>"Button" )
    .....
    .....
    .....
```

```
{}

SLATE_DEFAULT_SLOT( FArguments, Content )

.....
.....
};
```

事实上，在Slate中，`SLATE_DEFAULT_SLOT`指的就是那对中括号，而Content则是指中括号里面的具体内容。

在SButton构造函数中。先判断是否有设置`_Text`，如果没设置的话就会取出Content的内容。因为SButton继承自SBorder，所以调用了父类(SBorder)的构造函数，将中括号的内容放到了SBorder的中括号内。

SBorder如何处理Content呢？答案是：

```
ChildSlot
    .HAlign(InArgs._HAlign)
    .VAlign(InArgs._VAlign)
    .Padding(InArgs._Padding)
[
    InArgs._Content.Widget
];
```

直接添加到ChildSlot里。

解释这个过程只是为了说明一个道理：控件最终是要放到插槽里。在Slate系统里，容器并不是直接存储控件的，而是容器里的

Slot（插槽）来存放控件。插槽的数量决定这个容器是属于什么类型：没有**Slot**的叫**Leaf Widgets**；**Slot**数量不固定的叫**Panels**；有明确的**Slot**的叫**Compound Widgets**。

15.4.6 创建自定义控件

虽然Slate提供的控件非常多，但是很多时候我们还是需要自定义一些控件。而且不同类型的控件需要写在不同的文件里，方便管理。同时，为了在本书示例中，排除代码干扰，让读者更好地理解。接下来的Slate内容将与插件代码进行分离。

自定义Slate控件需要继承**SWidget** 或其子类。同时，建议每种控件都有独立的头文件与源文件。在虚幻引擎4中，创建任何类都需要遵循一些规则，Slate也不例外。为了方便创建控件，笔者开发了一个工具，用于创建虚幻引擎4中常见类的代码（项目地址：
<https://coding.net/u/Sanwuthree/p/UPCA>）。

此工具提供exe运行包。把它放到项目根目录后双击运行，其会自动分析当前项目的插件结构。

接下来我们创建一个自己的Slate控件，如图15-4所示。

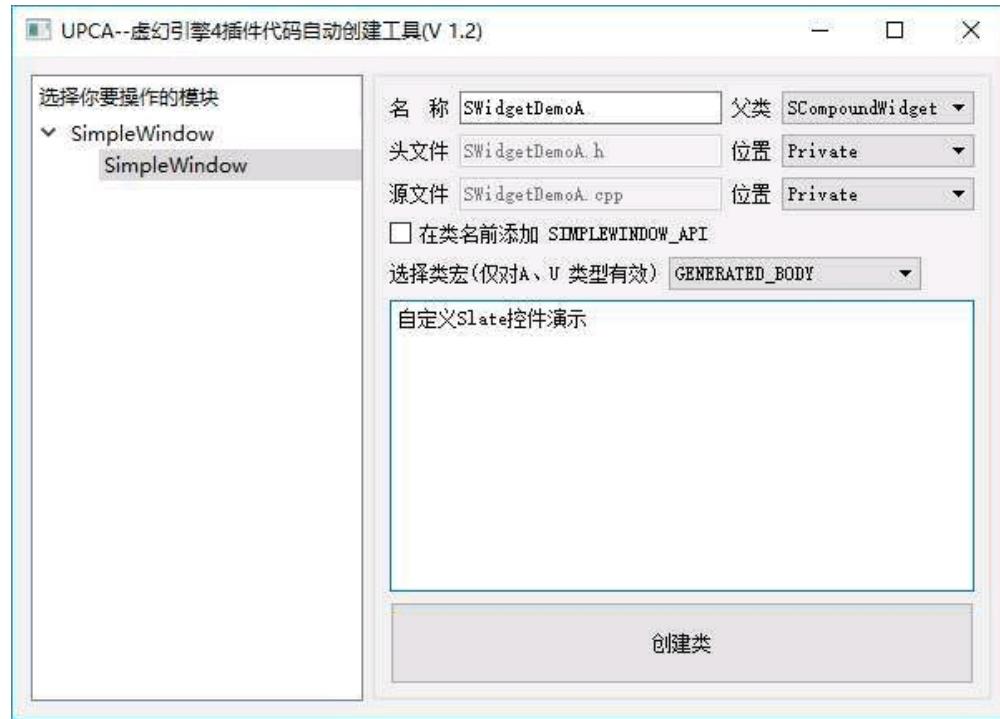


图15-4 创建Slate控件

创建完成后，还需要重新生成解决方案文件，以便在VS中编辑，如图15-5所示。

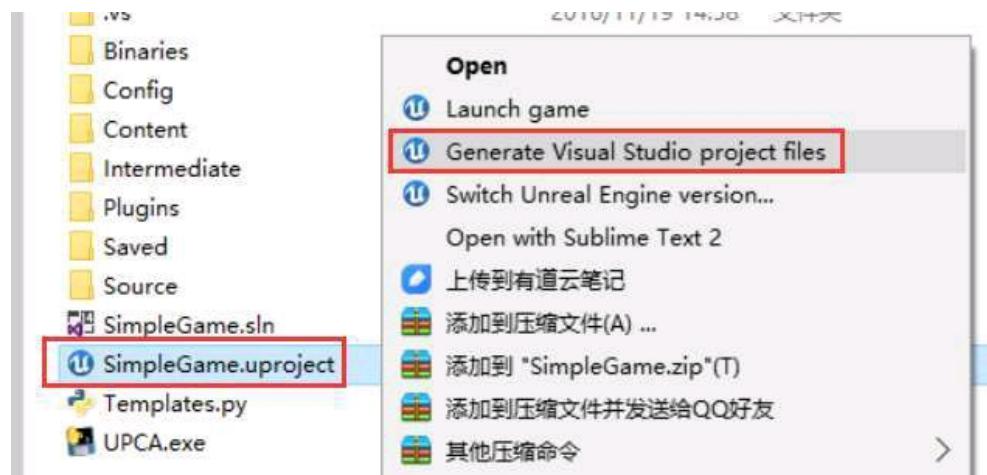


图15-5 重新生成解决方案文件

生成的代码如下：

在这个最基础的Slate控件类中，我们选择的是继承SCompoundWidget。SCompoundWidget可以自定义Slot。

SLATE_BEGIN_ARGS和SLATE_END_ARGS为固定的控件格式。
Construct函数为控件构造函数，会在实例化此控件的时候执行。
SCompoundWidget自带一个名叫ChildSlot的子槽。我们如果要添加控件，需要添加到ChildSlot里。在源文件中，ChildSlot添加了一个按钮（SButton）。

接下来，我们把这个控件添加到插件的OnSpawnPluginTab函数中，让它显示出来，如下所示。

```
//Simplewindow.cpp
.....
//添加头文件
#include "SWidgetDemoA.h"
.....
.....
.....
TSharedRef<SDockTab> FSimpleWindowModule::OnSpawnPluginTab(const
FSpawnTabArgs& SpawnTabArgs)
{
    return SNew(SDockTab)
        .TabRole(ETabRole::NomadTab)
        [
            SNew(SWidgetDemoA)//实例化SWidgetDemoA控件
        ];
}
.....
.....
```

运行后，最终效果将会是一个按钮充满整个窗口，如图15-6所示。

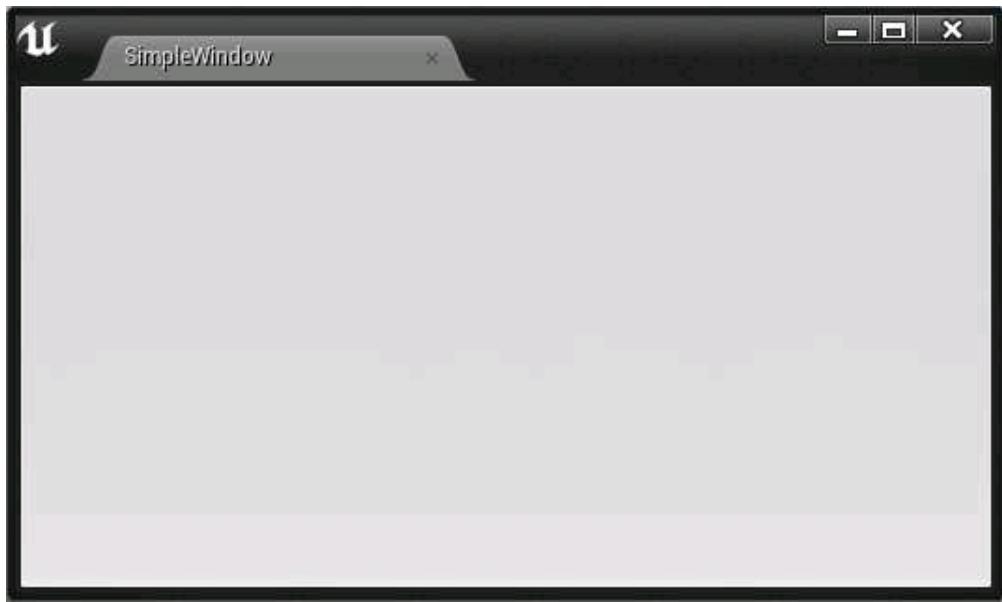


图15-6 最终效果

15.4.7 布局控件

UI布局通常使用Panels类型的控件。常用的有**SVerticalBox**、**SHorizontalBox**、**SOverlay**。接下来以SVerticalBox为例，演示一下垂直布局，如图15-7所示。



图15-7 垂直布局

```
void SWidgetDemoA::Construct(const FArguments& InArgs)
{
    this->ChildSlot
    [
        SNew(SVerticalBox)
        +SVerticalBox::Slot()
        [
            SNew(SButton)
        ]
        +SVerticalBox::Slot()
        [
            SNew(SButton)
        ]
    ];
}
```

在使用SVerticalBox控件时，添加新的子槽只需要

+SVerticalBox::Slot() 即可。

Slot也可以进行嵌套，如图15-8所示：



图15-8 添加新的子槽

```
void SWidgetDemoA::Construct(const FArguments& InArgs)
{
    this->ChildSlot
    [
        SNew(SVerticalBox)
        +SVerticalBox::Slot()
        [
            SNew(SButton)
        ]
        +SVerticalBox::Slot()
        [
            SNew(SHorizontalBox)
```

```
+SHorizontalBox::Slot()
[
    SNew(SButton)
]

+SHorizontalBox::Slot()
[
    SNew(SButton)
]

+SHorizontalBox::Slot()
[
    SNew(SButton)
]

];
}
```

15.4.8 控件参数与属性

每个控件都有自己的属性，不同属性影响了控件的表现形式。以 **SButton** 为例，我们可以设置它的**Text** 属性。

```
SNew(SButton).Text(LOCTEXT("Test","测试文字"))
```

参数是如何定义及使用的呢？这需要使用Slate的 **SLATE_ATTRIBUTE** 宏。接下来的例子将演示如何使用，首先在

SWidgetDemoA.h中添加代码，如下所示。

```
\\"SWidgetDemoA.h\n\n#pragma once\n\nclass SWidgetDemoA : public SCompoundWidget\n{\npublic:\n    SLATE_BEGIN_ARGS(SWidgetDemoA){}\n    SLATE_ATTRIBUTE(FString, InText)\n    SLATE_END_ARGS()\n    void Construct(const FArguments& InArgs);\n};
```

通过SLATE_ATTRIBUTE宏，我们添加了一个FString类型的属性，名叫**InText**。这里需要注意的是，SLATE_ATTRIBUTE必须放在**SLATE_BEGIN_ARGS**和**SLATE_END_ARGS**中间。

然后，在源文件中，我们需要获取到这个属性。

```
\\"SWidgetDemoA.cpp\n\nvoid SWidgetDemoA::Construct(const FArguments& InArgs)\n{\n    FString InString = InArgs._InText.Get();\n    UE_LOG(LogSimpleApp, Warning, TEXT("Attribute is %s"), *\n        InString);\n    ....
```

```
....  
}
```

通过构造函数的InArgs参数获取到InText的值。在Slate中，所有传递的参数、属性、代理等都会存放到构造函数的InArgs里。

最后，进行测试。与一般控件设置属性一样，在**SimpleWindow.cpp** 中，也就是我们创建控件的地方，修改代码为：

```
SNew(SWidgetDemoA).InText(FString("Hello Slate"))
```

如果一切正常，点击工具栏上的按钮后，将会在输出窗口打印“Attribute is Hello Slate”。

15.4.9 Delegate

通常，一个控件除了显示外还需要与它进行交互。进行交互就需要有反馈，比如：点击一个登录按钮，需要产生登录的反馈（提示正在登录、执行登录代码等）。

以Button例，进行一个简单的演示。

```
\\"SWidgetDemoA.h  
#pragma once  
class SWidgetDemoA : public SCompoundWidget  
{
```

```
public:  
    SLATE_BEGIN_ARGS(SWidgetDemoA){}  
        SLATE_ATTRIBUTE(FString, InText)  
    SLATE_END_ARGS()  
    void Construct(const FArguments& InArgs);  
  
private:  
    //点击按错后的反馈  
    FReply OnLoginButtonClicked();  
};
```

在头文件中，只需要定义接收反馈的函数OnLoginButtonClicked。需要注意的是，在Slate中，处理反馈的函数都要返回FReply，告诉系统我们已经处理了这个点击事件。

```
//SWidgetDemoA.cpp  
void SWidgetDemoA::Construct(const FArguments& InArgs)  
{  
    this->ChildSlot  
    [  
        SNew(SButton).OnClicked(this,&SWidgetDemoA::OnLoginButtonClicked);  
    ];  
}  
FReply SWidgetDemoA::OnLoginButtonClicked()  
{  
    //执行具体的代码  
    UE_LOG(LogSimpleApp , Warning, TEXT("按钮被点击了"));  
}
```

```
    return FReply::Handled();  
}  
}
```

运行结果请读者自行测试。

在构造函数中，实例化了一个SButton。然后，设置它的OnClicked，在OnClicked里传入OnLoginButtonClicked函数，这是一般C++回调函数的写法。OnClicked是什么呢？在SButton.h里是这么定义的：**SLATE_EVENT(FOnClicked, OnClicked)**。在Slate的宏，一般遵循(类型名，变量名)的规则。所以FOnClicked是一个类型，OnClicked则是具体名字。

SLATE_EVENT专门用于处理这种事件反馈（或者说“回调”或“代理”）。接下来将以一个登录窗口的例子演示如何自定义SlateEvent。

作为演示，先创建一个新的类——**SEventTest**。

```
\\" SEventTest.h  
#include "SEditableTextBox.h"  
#pragma once  
  
DECLARE_DELEGATE_TwoParams(FLoginDelegate, FString, FString);  
class SEventTest : public SCompoundWidget  
{  
public:  
    SLATE_BEGIN_ARGS(SEventTest){}  
    SLATE_EVENT(FLoginDelegate, OnStartLogin)  
    SLATE_END_ARGS()  
}
```

```
    void Construct(const FArguments& InArgs);

private:

    FReply OnLoginButtonClicked();
    FLoginDelegate OnLoginDelegate;
    TSharedPtr<SButton> LoginButtonPtr;
    TSharedPtr<SEditableTextBox> UserNamePtr;
    TSharedPtr<SEditableTextBox> PasswordPtr;
};
```

DECLARE_DELEGATE_TwoParams即自定义一个代理，并指明此代理将会有两个参数，类型为FString。

SLATE_EVENT(FLoginDelegate,OnStartLogin)即暴露一个接口，让SEventTest在实例化（SNew或SAssignNew）的时候，可以传入一个回调方法。

FLoginDelegate OnLoginDelegate是用于存储此回调方法。因为SLATE_EVENT将回调函数传入后，如果不进行存储，那么在构造函数结束后，将无法获取到此函数。因此，在构造函数里。我们就需要将OnLoginDelegate赋值。

```
// SEventTest.cpp

#include "SimpleWindowPrivatePCH.h"
#include "SEventTest.h"
#define LOCTEXT_NAMESPACE "SEventTest"

void SEventTest::Construct(const FArguments& InArgs)
{
```

```
OnLoginDelegate = InArgs._OnStartLogin;
this->ChildSlot.Padding(50, 50, 50, 50)
[
    SNew(SVerticalBox)
    +SVerticalBox::Slot().AutoHeight()
    [
        SAssignNew(UserNamePtr , SEditableTextBox)
        .HintText(LOCTEXT("Username_Hint", "请输入账号"))
    ]
    + SVerticalBox::Slot().AutoHeight()
    [
        SAssignNew>PasswordPtr , SEditableTextBox)
        .IsPassword(true)
        .HintText(LOCTEXT("Password_Hint", "请输入密码"))
    ]
    + SVerticalBox::Slot().AutoHeight()
    [
        SAssignNew(LoginButtonPtr , SButton)
        .OnClicked(this,&SEventTest::OnLoginButtonClicked)
        .Text(LOCTEXT("Login", "登录"))
    ]
];
}

FReply SEventTest::OnLoginButtonClicked()
{
    FString usn = UserNamePtr->GetText().ToString();
    FString pwd = PasswordPtr->GetText().ToString();
```

```
    OnLoginDelegate.ExecuteIfBound(usr, pwd);  
    return FReply::Handled();  
}  
  
#undef LOCTEXT_NAMESPACE
```

首先来看构造函数。第一行，从InArgs里面取出了OnStartLogin，OnStartLogin就是传入的回调函数。然后将它赋值给了OnLoginDelegate。接下来，我们就可以在任何时候调用OnLoginDelegate，因为它是一个类方法。我们可以在此类的任何地方调用它。

下面是界面布局。在一个SVerticalBox中放了3个SLot，前两个用于给用户输入账号密码，最后一个登录按钮。然后绑定了点击登录按钮后，要执行函数。

OnLoginButtonClicked函数将会在用户点击“登录”按钮后执行。在函数内部，我们首先获取到用户输入的账号与密码。然后，调用OnLoginDelegate的ExecuteIfBound函数将账号密码输出出去。这里需要大家知道的是，Delegate的调用除了ExecuteIfBound还有Execute。用ExecuteIfBound是因为有可能在SEventTest实例化的时候，并没传入回调函数。这时候调用Execute就会发生错误。

那么，类的定义与实现就已经完成了。接下来，进行实际测试。

回到SWidgetDemoA里。将SEventTest添加到构造函数里，让它显示出来。

```
//SWidgetDemoA.h
```

```
#pragma once

class SWidgetDemoA : public SCompoundWidget
{
public:
    SLATE_BEGIN_ARGS(SWidgetDemoA){}
    SLATE_END_ARGS()
    void Construct(const FArguments& InArgs);
private:
    //点击按错后的反馈
    void OnLogin(FString usn, FString pwd);
};

///////////
//SWidgetDemoA.cpp

#include "SimpleWindowPrivatePCH.h"
#include "SWidgetDemoA.h"
#include "SEventTest.h"
#define LOCTEXT_NAMESPACE "SWidgetDemoA"

void SWidgetDemoA::Construct(const FArguments& InArgs)
{
    this->ChildSlot
    [
        SNew(SEventTest)
        .OnStartLogin(this,&SWidgetDemoA::OnLogin)
    ];
}

void SWidgetDemoA::OnLogin(FString usn,FString pwd)
{
```

```
//执行具体的代码  
UE_LOG(LogSimpleApp , Warning, TEXT("开始登录 %s - %s"), *usn, *  
}  
#undef LOCTEXT_NAMESPACE
```

在SWidgetDemoA.h中，只是添加了一个回调函数，用于进行测试。

在SWidgetDemoA.cpp里的构造函数中，通过SNew的方式，实例化了SEventTest。然后，通过.OnStartLogin()的方式，将OnLogin函数传入SEventTest内部。这样，就可以在SEventTest内部调用SWidgetDemoA的OnLogin函数。

最后，记得include"SEventTest.h"。登录界面如图15-9所示。



图15-9 登录界面

15.4.10 自定义皮肤

俗话说：“人靠衣装，佛靠金装”。同样的，做软件开发，UI的样式也是非常重要的一环。在之前的内容中，我们都是使用虚幻引擎4默认的UI样式。现在，我们将一起来研究自定义皮肤样式。

在本节中，将继续使用上一节的“登录界面”的代码。为它进行一些美化。在写代码前，我们先来看一下最终效果，如图15-10所示。



图15-10 自定义皮肤后的效果

在上图中，首先是添加了一张图片，然后，按钮和输入框的背景图片被替换了，并且按钮中显示的字体及大小和颜色也进行了更改。

资源准备。通常情况下，插件所用到的资源都会放在Resources目录下。在本例中，需要准备一张大图、一个字体文件，以及三种按钮状态的图片，如图15-11所示。

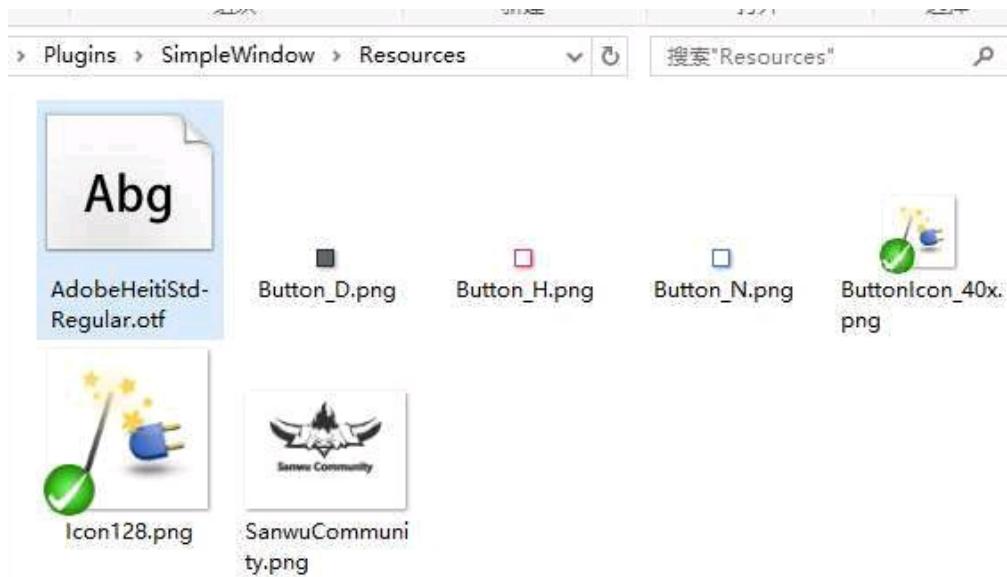


图15-11 资源准备

皮肤定义。在插件模板中，提供了一个非常方便的类用于自定义皮肤样式——SimpleWindowStyle。要增加新的皮肤样式只需要在 SimpleWindowStyle.cpp 中的 Create 函数中添加相应代码即可。如下：

```
// SimpleWindowStyle.cpp
TSharedRef< FSlateStyleSet > FSimpleWindowStyle::Create()
{
    TSharedRef< FSlateStyleSet > Style = MakeShareable(new
        FSlateStyleSet("SimpleWindowStyle"));
    Style->SetContentRoot(IPluginManager::Get().FindPlugin("SimpleWindow")
        ->GetBaseDir() / TEXT("Resources"));
    Style->Set("SimpleWindow.OpenPluginWindow", new IMAGE_BRUSH(TEXT("SimpleWindow.OpenPluginWindow"), ButtonIcon_40x));
    Style->Set("UI.Sanwu", new IMAGE_BRUSH(TEXT("SanwuCommunity"), FVector2D(482, 350)));
}
```

```
    FMargin Button1Margin(2.0 / 10.f, 2.0 / 10.f, 2.0 / 10.f, 2.0 / 10.f);
}

Style->Set("UI.Button1", FButtonStyle()
    .SetNormal(BOX_BRUSH("Button_N", Button1Margin))
    .SetHovered(BOX_BRUSH("Button_H", Button1Margin))
    .SetPressed(BOX_BRUSH("Button_H", Button1Margin))
    .SetDisabled(BOX_BRUSH("Button_D", Button1Margin)));
Style->Set("UI.InputLineText", FEditableTextBoxStyle()
    .SetBackgroundImageNormal(BOX_BRUSH("Button_N", Button1Margin))
    .SetBackgroundImageFocused(BOX_BRUSH("Button_H",
        Button1Margin)))
);

Style->Set("UI.AdobeHS_16", FSlateFontInfo("AdobeHeitiStd-Regular",
    otf", 16));
return Style;
}
```

创建一个新的皮肤样式使用Style->Set的方式添加。其中第一个参数为样式的名称，第二个参数指定具体的样式。在图片创建中，我们使用了IMAGE_BRUSH宏。此宏在SimpleWindowStyle.cpp已经定义，除此之外，还有几个宏用于快速创建样式：

1. **IMAGE_BRUSH** 图片。
2. **BOX_BRUSH** 九宫格缩放样式。
3. **BORDER_BRUSH** 九宫格缩放样式，但不留中间部分。
4. **TTF_FONT_TTF** 字体。

5. OTF_FONTOTF 字体。

注：关于“九宫格缩放”请读者自行查找相关资料。

皮肤样式定义完成后，使用起来就非常简单了。引用 SimpleWindowStyle.h 后，只需要在对应的控件中，设置其相应的属性即可。

由于不同的控件设置皮肤样式不同，并且一个控件也可能有多个属性用于设置样式。所以，具体使用哪个属性来设置样式，需要打开对应控件的头文件，从头文件中的 SLATE_ATTRIBUTE 查找。因为虚幻引擎 4 源码的注释非常完善，所以找起来也不难。

在本例中，使用皮肤样式，用到了两种方法。分别是 FSimpleWindowStyle::Get().GetBrush("style_name") 与 &FSimpleWindowStyle::Get().GetWidgetStyle<FXXXStyle> ("style_name")。

```
//SEventTest.cpp
#include "SimpleWindowPrivatePCH.h"
#include "SEventTest.h"
#include "SimpleWindowStyle.h"
#define LOCTEXT_NAMESPACE "SEventTest"
void SEventTest::Construct(const FArguments& InArgs)
{
    this->ChildSlot.Padding(50, 50, 50, 50)
    [
        SNew(SVerticalBox)
```

```
+ SVerticalBox::Slot().VAlign(VAlign_Top)
[
    SNew(SHorizontalBox)
        +SHorizontalBox::Slot().HAlign(HAlign_Center)
    [
        SNew(SImage).Image(FSimpleWindowStyle::Get().GetEditorIcon("Sanwu"))
    ]
]
+SVerticalBox::Slot().AutoHeight().Padding(0,10,0,10)
[
    SAssignNew(UserNamePtr , SEditableTextBox)
        .Padding(FMargin(5, 3, 5, 3))
        .Style(&FSimpleWindowStyle::Get().GetWidgetStyle<
            FEditableTextBoxStyle >("UI.InputLineText"))
        .HintText(LOCTEXT("Username_Hint","请输入账号"))
    ]
+
+SVerticalBox::Slot().AutoHeight().Padding(0, 10, 0, 10)
[
    SAssignNew(UserNamePtr, SEditableTextBox)
        .Padding(FMargin(5,3,5,3))
        .Style(&FSimpleWindowStyle::Get().GetWidgetStyle<
            FEditableTextBoxStyle >("UI.InputLineText"))
        .IsPassword(true)
        .HintText(LOCTEXT("Password_Hint", "请输入密码"))
    ]
+
+SVerticalBox::Slot().AutoHeight().Padding(0, 10, 0, 10)
```

```
[  
    SAssignNew(LoginButtonPtr, SButton)  
        .HAlign(HAlign_Center)  
        .ContentPadding(FMargin(0, 10, 0, 10))  
        .ButtonStyle(&FSimpleWindowStyle::Get().GetWidgetStyle<  
            FButtonStyle>("UI.Button1"))  
    [  
        SNew(STextBlock)  
            .ColorAndOpacity(FLinearColor(0.033, 0.604, 0.604))  
            .Font(FSimpleWindowStyle::Get().GetFontStyle("UI.Adob  
                "))  
            .Text(LOCTEXT("Login", "登录"))  
    ]  
]  
];  
}  
#undef LOCTEXT_NAMESPACE
```

15.4.11 图标字体

在虚幻引擎4中，图标字体使用的是**Font Awesome**，一个完全免费的图标字体库（<http://fontawesome.io/>）。图标字体可能在没有设计师的配合或者没精力设计图标的情况下，用代码快速生成漂亮的图标。

具体使用方法如下：

1. 在Build.cs中添加模块引用（通常在PrivateDependencyModuleNames中添加“EditorStyle”）。
2. 在需要使用图标字体的地方，引用相应头文件（EditorStyleSet.h和EditorFontGlyphs.h）。
3. 设置字体Font为“FontAwesome”。
4. 设置具体显示的图标。

以下示例代码将展示添加两个图标字体。

```
//WidgetDemoA.cpp演示字体图标使用

...
#include "EditorStyleSet.h"
#include "EditorFontGlyphs.h"

.....
.....
void SWidgetDemoA::Construct(const FArguments& InArgs)
{
    this->ChildSlot
    [
        SNew(SHorizontalBox)
        +SHorizontalBox::Slot().AutoWidth()
        [
            SNew(STextBlock)
            .Font(FEditorStyle::Get()).GetFontStyle("FontAwesome.1")
            .Text(FEditorFontGlyphs::Android)
        ]
        + SHorizontalBox::Slot().AutoWidth()
```

```
[  
    SNew(STextBlock)  
        .Font(FEditorStyle::Get()).GetFontStyle("FontAwesome.1")  
        .Text(FEditorFontGlyphs::Apple)  
    ]  
];  
}
```

FEditorFontGlyphs指定具体的图标样式，因为样式繁多，读者可以从(<http://fontawesome.io/icons/>)中找到自己想要的图标。但因为版本更新速度不同，fontawesome中有些图标可能在虚幻引擎4中没有，最终效果如图15-12所示。



图15-12 最终效果

15.4.12 组件继承

继承是实现代码重用、扩展功能的重要手段。在Slate的UI界面的实际开发中，如果用虚幻引擎4默认的组件，则需要设置大量的属性，代

码量偏多。而用组件继承的方式，可以有效降低代码量，让代码结构更清晰。同时，继承也大量用于扩展已有组件的功能。

接下来，笔者将用一个最简单的继承SButton的例子来讲解在Slate中如何实现组件继承。

首先，新建一个类名叫SMyButton。具体代码如下：

```
// MyButton.h
#pragma once
class SMyButton : public SButton
{
public:
    SLATE_BEGIN_ARGS(SMyButton){}
        SLATE_ATTRIBUTE(FText, Icon)
        SLATE_ATTRIBUTE(FText, Text)
    SLATE_END_ARGS()
    void Construct(const FArguments& InArgs);
};
```

在SMyButton头文件中，定义此类继承SButton。然后，通过SLATE_ATTRIBUTE暴露两个属性接口，笔者想实现的结果是上一节学习的图标字体。创建一个SMyButton的时候，只需要传入两个FText参数就能创建一个带小图标的按钮。

```
// MyButton.cpp
#include "SimpleWindowPrivatePCH.h"
```

```
#include "SMyButton.h"
#include "EditorStyleSet.h"
#include "EditorFontGlyphs.h"
#include "SimpleWindowStyle.h"
#define LOCTEXT_NAMESPACE "SMyButton"
void SMyButton::Construct(const FArguments& InArgs)
{
    SButton::Construct(SButton::FArguments()
        .ContentPadding(FMargin(10, 5, 10, 5))
        .ButtonStyle(&FSimpleWindowStyle::Get()).GetWidgetStyle <FButtonStyle>("UI.Button1"))
    .Content()[]
        SNew(SHorizontalBox)
        +SHorizontalBox::Slot().AutoWidth().AutoWidth().Padding(5, 0)
    [
        SNew(STextBlock)
        .Font(FEditorStyle::Get()).GetFontStyle("FontAwesome.1")
        .Text(InArgs._Icon)
    ]
    +SHorizontalBox::Slot().AutoWidth().Padding(5, 0, 5, 0)
    [
        SNew(STextBlock)
        .Font(FSimpleWindowStyle::Get()).GetFontStyle("UI.Adob")
    ))
    .Text(InArgs._Text)
}
```

```
]);
}

#undef LOCTEXT_NAMESPACE
```

在SMyButton构建函数中，调用父类(SButton)构造函数即完成继承。在SButton构造函数中，我们传入一些参数来对SButton进行定制化。如ContentPadding指定Button内部元素的外边距，ButtonStyle指定按钮样式。Content指定按钮内部的内容是什么？在Content内部，实例化了一个SHorizontalBox用于进行横向排列元素。这里排列了两个STextBlock，第一个STextBlock设置字体为FontAwesome，就是用于创建图标字体。从InArgs里获取Text的内容，即头文件中SLATE_ATTRIBUTE的内容。

最终，使用的时候，我们的代码将会变得非常简洁。最终效果如图15-13所示。



图15-13 最终效果

```
void SWidgetDemoA::Construct(const FArguments& InArgs)
```

```
{  
    this->ChildSlot  
    [  
        SNew(SVerticalBox)  
        + SVerticalBox::Slot()  
        .VAlign(VAlign_Top)  
        [  
            SNew(SHorizontalBox)  
            +SHorizontalBox::Slot()  
            .HAlign(HAlign_Center)  
            [  
                SNew(SMyButton)  
                .Icon(FEditorFontGlyphs::Android)  
                .Text(LOCTEXT("Android", "安卓"))  
            ]  
        ]  
    ];  
}
```

15.4.13 动态控制Slot

相信大家学习到现在，应该知道Slot的属性决定一个控件的显示位置、边距、对齐方式等表现形式。那么问题来了，如果想在运行时动态改变一个控件的显示怎么办？我们之前写的代码，Slot的属性都是在代码里写好了的，后期没办法修改。我们知道，改变一个控件的方法是用SAssignNew，存储控件的指针。那么Slot是否也可以用同样的方式呢？

答案是肯定的。但Slot不是用SAssignNew，而是用Slot的Expose函数。具体使用如下所示：

```
\\"SWidgetDemoA.h\n\nclass SWidgetDemoA : public SCompoundWidget\n{\npublic:\n    SLATE_BEGIN_ARGS(SWidgetDemoA){}\n    SLATE_END_ARGS()\n    void Construct(const FArguments& InArgs);\nprivate:\n    //点击按错后的反馈\n    FReply OnLoginButtonClicked();\n    FReply ChangeSlotProperty(int32 type);\n    SHorizontalBox::FSlot* ButtonSlot; //定义 Slot指针\n};
```

在头文件中，通过SHorizontalBox::FSlot*ButtonSlot;的方式，定义一个SHorizontalBox的Slot指针。接下来，在创建Slot的时候，就可以通过Expose的方式把Slot指针赋值过来。

```
void SWidgetDemoA::Construct(const FArguments& InArgs)\n{\n    this->ChildSlot\n    [\n        SNew(SVerticalBox)
```

```
+ SVerticalBox::Slot().VAlign(VAlign_Top)
[
    SNew(SHorizontalBox)
    +SHorizontalBox::Slot().HAlign(HAlign_Center)
        .Expose(ButtonSlot) //将Slot赋值给ButtonSlot
    [
        SNew(SMyButton)
            .Icon(FEditorFontGlyphs::Android)
            .Text(LOCTEXT("Android", "安卓"))
    ]
]
+SVerticalBox::Slot().AutoHeight()
[
    SNew(SHorizontalBox)
    +SHorizontalBox::Slot()
    [
        SNew(SButton)
            .OnClicked(this, &SWidgetDemoA::ChageSlotProp)
            .Text(LOCTEXT("0", "Left"))
    ]
    + SHorizontalBox::Slot()
    [
        SNew(SButton)
            .OnClicked(this, &SWidgetDemoA::ChageSlotProp)
            .Text(LOCTEXT("1", "Right"))
    ]
]
```

```
[  
    SNew(SButton)  
        .OnClicked(this, &SWidgetDemoA::ChageSlotProp  
        .Text(LOCTEXT("2", "Center"))  
    ]  
    + SHorizontalBox::Slot()  
    [  
        SNew(SButton)  
            .OnClicked(this, &SWidgetDemoA::ChageSlotProp  
            .Text(LOCTEXT("3", "Fill"))  
    ]  
]  
];  
}  
  
FReply SWidgetDemoA::ChageSlotProperty(int32 type)  
{  
    switch (type)  
    {  
        case 0:  
            ButtonSlot->HAlign(HAlign_Left);  
            break;  
        case 1:  
            ButtonSlot->HAlign(HAlign_Right);  
            break;  
        case 2:  
            ButtonSlot->HAlign(HAlign_Center);  
            break;  
    }  
}
```

```
case 3:  
    ButtonSlot->HAlign(HAlign_Fill);  
    break;  
}  
return FReply::Handled();  
}
```

在上述代码中的构造函数里，首先用两个SVerticalBox::Slot将整个界面分为上下两个部分。上面部分有一个SHorizontalBox::Slot盛放一个按钮控件，并把这个Slot赋值给头文件中定义的ButtonSlot。在下面部分的SVerticalBox::Slot中，放了4个SHorizontalBox::Slot，每个里面有一个按钮。点击按钮后会执行SWidgetDemoA::ChageSlotProperty函数，并传入不同的参数。通过判断参数的不同，来确定点击的是哪个按钮，如图15-14所示。



图15-14 动态设置Slot属性演示

在SWidgetDemoA::ChageSlotProperty函数中，用Switch的方式来确定点击不同按钮后执行的操作，即分别设置ButtonSlot的横向对齐方式。

15.4.14 自定义容器布局

在之前的例子中，我们一般都是使用SCompoundWidget组件，通过设置属性来控制控件的布局。但除了SCompoundWidget外，还有一种容器组件（还记得之前说的三种控件类型吧）——Panel类型的组件。比

如SVerticalBox就是一种垂直布局组件。但有些时候这些布局方式并不能满足我们的需求。这时候就要自己控制容器布局方式。

首先，我们来看一个自定义的容器布局演示，如图15-15和图15-16所示：

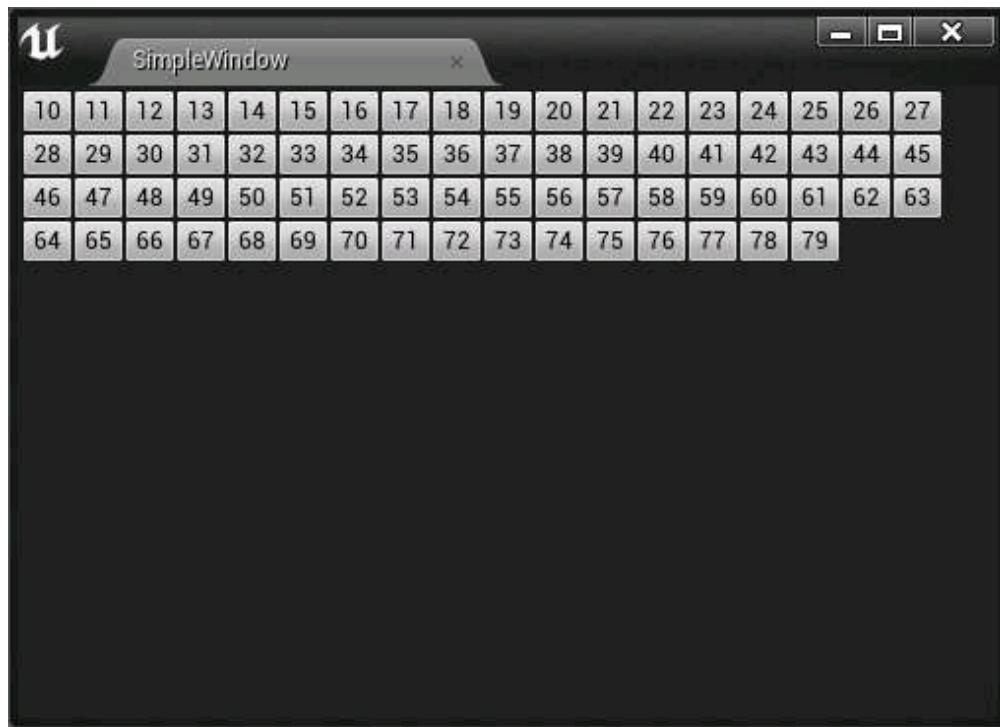


图15-15 自定义容器布局1

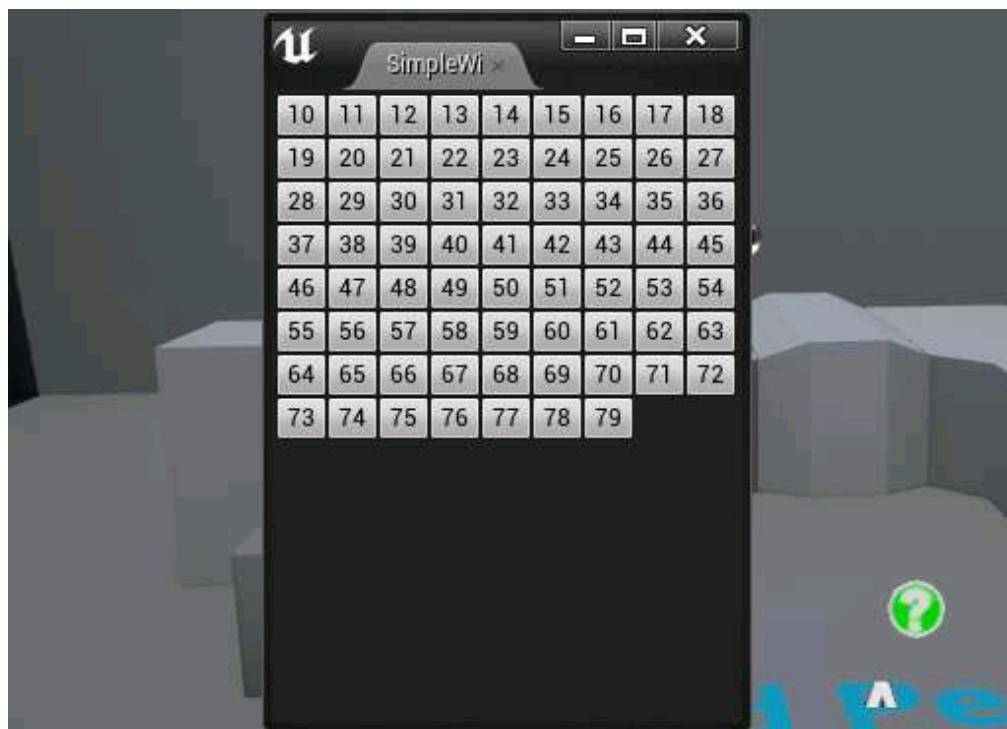


图15-16 自定义容器布局2

在上面两张图中，通过调整窗口的大小，容器中的元素布局会发生变化，简单来说，就是自适应窗口的宽度。

在Slate中，容器布局是靠重写**OnArrangeChildren**函数 来完成的。

示例代码如下：

```
// AutoLayout.h
#pragma once
#include "SBoxPanel.h"
///自动排列的容器组件。
class SAutoLayout :public SBoxPanel
{
public:
```

```
class FSlot : public SBoxPanel::FSlot
{
public:
    FSlot()
        : SBoxPanel::FSlot()
    {}

    FSlot& AutoHeight()
    {
        SizeParam = FAuto();
        return *this;
    }

    FSlot& MaxHeight(const TAttribute< float >&
                      InMaxHeight)
    {
        MaxSize = InMaxHeight;
        return *this;
    }

    FSlot& FillHeight(const TAttribute< float >&
                      StretchCoefficient)
    {
        SizeParam = FStretch(StretchCoefficient);
        return *this;
    }

    FSlot& Padding(float Uniform)
    {
        SlotPadding = FMargin(Uniform);
        return *this;
    }
}
```

```
}

FSlot& Padding(float Horizontal, float Vertical)
{
    SlotPadding = FMargin(Horizontal, Vertical);
    return *this;
}

FSlot& Padding(float Left, float Top, float Right,
                float Bottom)
{
    SlotPadding = FMargin(Left, Top, Right,
                          Bottom);
    return *this;
}

FSlot& Padding(const TAttribute<FMargin>::FGetter&
                InDelegate)
{
    SlotPadding.Bind(InDelegate);
    return *this;
}

FSlot& HAlign(EHorizontalAlignment InHAlignment)
{
    HAlignment = InHAlignment;
    return *this;
}

FSlot& VAlign(EVerticalAlignment InVAlignment)
{
    VAlignment = InVAlignment;
```

```
        return *this;
    }

    FSlot& Padding(TAttribute<FMargin> InPadding)
    {
        SlotPadding = InPadding;
        return *this;
    }

    FSlot& operator[](TSharedRef<SWidget> InWidget)
    {
        SBoxPanel::FSlot::operator[](InWidget);
        return *this;
    }

    FSlot& Expose(FSlot*& OutVarToInit)
    {
        OutVarToInit = this;
        return *this;
    }

};

static FSlot& Slot()
{
    return *(new FSlot());
}

public:

    SLATE_BEGIN_ARGS(SAutoLayout){}
    SLATE_ATTRIBUTE(FMargin, ContentMargin)
    SLATE_SUPPORTS_SLOT(SAutoLayout::FSlot)
    SLATE_END_ARGS()
```

```
FORCENOINLINE SAutoLayout()
: SBoxPanel(Orient_Horizontal)
{
}

virtual void OnArrangeChildren(const FGeometry&
    AllottedGeometry , FArrangedChildren& ArrangedChildren)
const override;

public:

    void Construct(const FArguments& InArgs);

    FSlot& AddSlot() {
        SAutoLayout::FSlot& NewSlot = *new FSlot();
        this->Children.Add(&NewSlot);
        return NewSlot;
    }

private:
    FMargin ContentMargin;
};
```

首先，我们定义了一个继承自SBoxPanel的类，SBoxPanel属于Panel类型的控件。接着在SAutolayout类中定义了一个FSlot并实现了一些常用的属性控制函数。从SLATE_BEGIN_ARGS宏开始定义SAutolayout类的属性，其中定义了一个ContentMargin，用于设置子元素之间的间隙。重写OnArrangeChildren函数。

AddSlot函数，用于动态向容器中添加子元素。其中，实现过程是向SAutoLayout类的Children（可以理解为一个放元素的数组）添加Slot。

```
\\" SAutoLayout.cpp

#include "SimpleWindowPrivatePCH.h"
#include "SAutoLayout.h"

void SAutoLayout::Construct(const FArguments& InArgs)
{
    const int32 NumSlots = InArgs.Slots.Num();
    ContentMargin = InArgs._ContentMargin.Get();
    for (int32 SlotIndex = 0; SlotIndex < NumSlots; ++SlotIndex)
    {
        Children.Add(InArgs.Slots[SlotIndex]);
    }
}

void SAutoLayout::OnArrangeChildren(
    const FGeometry& AllottedGeometry ,
    FArrangedChildren&
    ArrangedChildren)const
{
    //areaSize即容器的尺寸
    FVector2D areaSize = AllottedGeometry.GetLocalSize();
    float startX = ContentMargin.Left;
    float startY = ContentMargin.Top;
    float currentMaxHeight = 0.f;
    for (int32 ChildIndex = 0; ChildIndex < Children.Num(); ++ChildIndex)
    {
        const SBoxPanel::FSlot& CurChild = Children[ChildIndex];
```

```
const EVisibility ChildVisibility = CurChild.GetWidget()->GetVisibility();

//获取元素的尺寸
FVector2D size = CurChild.GetWidget()->GetDesiredSize();

//Accepts用于判断一个元素是否需要参与计算。
if (ArrangedChildren.Accepts(ChildVisibility))
{
    if (size.Y>currentMaxHeight)
    {
        currentMaxHeight = size.Y;
    }

    //判断StartX是否超过容器的宽度。
    if (startX+size.X<areaSize.X)
    {
        ArrangedChildren.AddWidget(ChildVisibility, AllottedGeometry.MakeChild(CurChild.GetWidget(), FVector2D(startX, startY), FVector2D(size.X, size.Y)));
        startX += ContentMargin.Right;
        startX += size.X;
    }
    else{
        //超过宽度后，将StartX设置为最左边
        startX = ContentMargin.Left;
        //同时StartY增加一定的高度
        startY += currentMaxHeight + ContentMargin.BottomOffset;
        ArrangedChildren.AddWidget(ChildVisibility, AllottedGeometry.MakeChild(CurChild.GetWidget(), FVector2D(startX, startY), FVector2D(size.X, size.Y)));
    }
}
```

```
        startX += size.X + ContentMargin.Right;
        currentMaxHeight = size.Y;
    }
}
}
}
```

在源文件中，首先在构造函数中，取出子元素数量。因为除了用AddSlot函数动态添加子元素外，也可以直接在实例化SAutoLayout的时候直接添加子元素（如SButton后面可以直接接中括号，把要添加的元素放进去），然后取出了传入的ContentMargin。

OnArrangeChildren会传入两个参数，其中AllottedGeometry为分配给此容器的尺寸，ArrangedChildren则用于设定子元素的具体位置。

整个排列过程实现原理为：先获取容器的尺寸，然后定义*StartX*与*StartY*，都默认为0（表示左上角），然后循环获取所有的子元素，并得到子元素的尺寸。每添加一个子元素，则*StartX*会加上子元素的宽度，这样，*StartX*会越来越大。当*StartX*超过容器的宽度后，将*StartX*归0（这样，超始位置又回到了最左边）。然后，将*StartY*增加子元素的高度。通过这样的方式，实现一个从左到右，从上到下的子元素排列。

最后，使用此容器只需要实例化它，然后通过AddSlot的方式添加子元素即可，如下所示：

```
this->ChildSlot
```

```
[  
    SAssignNew(LayoutPtr, SAutoLayout)  
];  
for (int32 i = 10; i < 80; i++)  
{  
    LayoutPtr->AddSlot()  
    [  
        SNew(SButton)  
        .Text(FText::FromString(FString::FromInt(i)))  
    ];  
}
```

15.5 UMG扩展

UMG系统允许我们以一种可视化的方式编辑UI，相比Slate这种以代码方式编写UI，可视化操作效率高很多。但虚幻引擎4官方提供的UMG控件类型有限，很多时候需要自己根据需求制作。自定义UMG控件有两种方式，一种是用UMG自己来制作，另一种就是将Slate控件转化成UMG控件。下面通过一个例子演示如何将Slate控件转化成UMG控件。为了连贯性，将继续使用之前的代码，也就是将SAutoLayout控件移植到UMG中。

头文件定义 。这里将参照VerticalBox的UMG实现。首先，创建两个类——AutoLayout与AutoLayoutSlot，分别用于定义AutoLayout的UMG容器与其窗口内元素。

```
//AutoLayout.h

#pragma once

#include "Components/PanelWidget.h"
#include "SAutoLayout.h"
#include "AutoLayoutSlot.h"
#include "AutoLayout.generated.h"

UCLASS()
class UAutoLayout : public UPanelWidget
{
    GENERATED_UCLASS_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Sanwu|"
              AutoLayout")
    FMargin ContentMargin;

public:
#if WITH_EDITOR
    virtual const FText GetPaletteCategory() override;
#endif

protected:
    //UPanelWidget
    virtual UClass* GetSlotClass() const override;
    virtual void OnSlotAdded(UPanelSlot* Slot) override;
    virtual void OnSlotRemoved(UPanelSlot* Slot) override;
    // End UPanelWidget
    virtual void ReleaseSlateResources(bool bReleaseChildren) override;
    virtual TSharedRef<SWidget> RebuildWidget() override;
    TSharedPtr<SAutoLayout> MyAutoLayout;
};

}
```

AutoLayout作为一个布局容器，这里选择继承UPanelWidget。

然后用UPROPERTY定义了一个FMargin类型的属性ContentMargin。UPROPERTY属性定义会将ContentMargin暴露在编辑器的Editor面板，用于使用的时候，可以在面板直接编辑此变量。ContentMargin的作用在SAutolayout中是用于调整容器内元素的间距。

接下来要做的是重写一些函数。GetPaletteCategory用于设定此UMG组件在UMG面板中的分类，这里需要注意的是在重写前需要判断WITH_EDITOR宏。因为UMG是可以脱离编辑器运行的，就是说它可以运行在Editor模式下，我们编辑好UI，最终打包后（Runtime），编辑器模块是不存在的。如果在Runtime模式下尝试调用一个定义在Editor模式下的函数，是会报错的，所以需要进行判断。

```
\AutoLayout.cpp
#include "SimpleWindowPrivatePCH.h"
#include "AutoLayout.h"
#include "SimpleWindowStyle.h"

UAutoLayout::UAutoLayout(const FObjectInitializer& ObjectInitializer)
: Super(ObjectInitializer)
{
}

const FText UAutoLayout::GetPaletteCategory()
{
    return FText::FromString(FString("Sanwu"));
}
```

```
UClass* UAutoLayout::GetSlotClass() const
{
    return UAutoLayoutSlot::StaticClass();
}

void UAutoLayout::OnSlotAdded(UPanelSlot* InSlot)
{
    if (MyAutoLayout.IsValid())
    {
        CastChecked<UAutoLayoutSlot >(InSlot)->BuildSlot(MyAutoLayout
            ToSharedRef());
    }
}

void UAutoLayout::OnSlotRemoved(UPanelSlot* InSlot)
{
    if (MyAutoLayout.IsValid())
    {
        TSharedPtr<SWidget> Widget = InSlot->Content->GetCachedWidget();
        if (Widget.IsValid())
        {
            MyAutoLayout ->RemoveSlot(Widget.ToSharedRef());
        }
    }
}

void UAutoLayout::ReleaseSlateResources(bool bReleaseChildren)
{
    Super::ReleaseSlateResources(bReleaseChildren);
}
```

```
TSharedRef<SWidget> UAutoLayout::RebuildWidget()
{
    MyAutoLayout = SNew(SAutoLayout).ContentMargin(ContentMargin)
    for (UPanelSlot* PanelSlot : Slots)
    {
        if (UAutoLayoutSlot* TypedSlot = Cast<UAutoLayoutSlot >(
            PanelSlot))
        {
            TypedSlot->Parent = this;
            TypedSlot->BuildSlot(MyAutoLayout.ToSharedRef());
        }
    }
    return BuildDesignTimeWidget(MyAutoLayout.ToSharedRef());
}
```

接下来定义AutoLayoutSlot。

```
//AutoLayoutSlot.h
#pragma once
#include "Components/PanelSlot.h"
#include "SAutoLayout.h"
#include "AutoLayoutSlot.generated.h"

UCLASS()
class UAutoLayoutSlot : public UPanelSlot
{
    GENERATED_UCLASS_BODY()
```

```
public:  
    // UPanelSlot interface  
    virtual void SynchronizeProperties() override;  
    // End of UPanelSlot interface  
    virtual void ReleaseSlateResources(bool bReleaseChildren) override;  
    void BuildSlot(TSharedRef<SAutoLayout> InAutoLayout);  
  
private:  
    SAutoLayout::FSlot* Slot;  
};  
////////////////////////////////////////////////////////////////  
// AutoLayoutSlot.cpp  
#include "SimpleWindowPrivatePCH.h"  
#include "AutoLayoutSlot.h"  
UAutoLayoutSlot::UAutoLayoutSlot(const FObjectInitializer&  
    ObjectInitializer)  
:Super(ObjectInitializer)  
{  
void UAutoLayoutSlot::SynchronizeProperties(){}
void UAutoLayoutSlot::ReleaseSlateResources(bool bReleaseChildren)
{
    Super::ReleaseSlateResources(bReleaseChildren);
    Slot = NULL;
}
void UAutoLayoutSlot::BuildSlot(TSharedRef<SAutoLayout> InAutoLayout)
{
    Slot = &InAutoLayout->AddSlot();
    Content==NULL?SNullWidget::NullWidget:Content->TakeWidget
```

```
];
}
```

可以看到，AutoLayoutSlot的代码非常简单。核心部分只有BuildSlot函数。在BuildSlot中，只是判断了一个Content是否为空，如果为空就添加一个空组件，否则就取出Content里面的内容。最终效果如图15-17和图15-18所示：

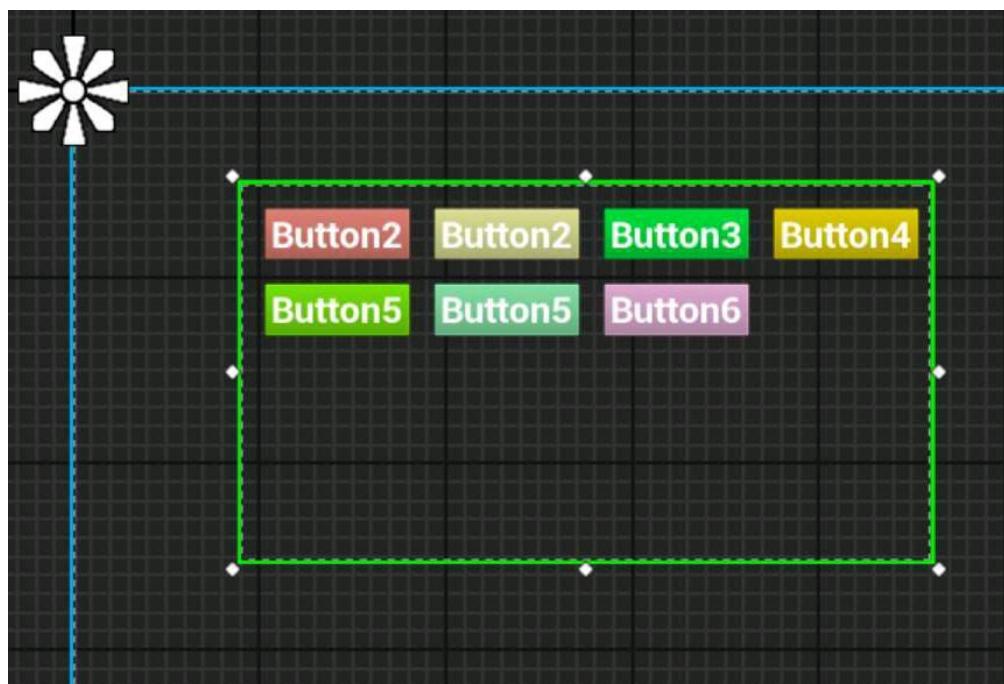


图15-17 UMG扩展示例1

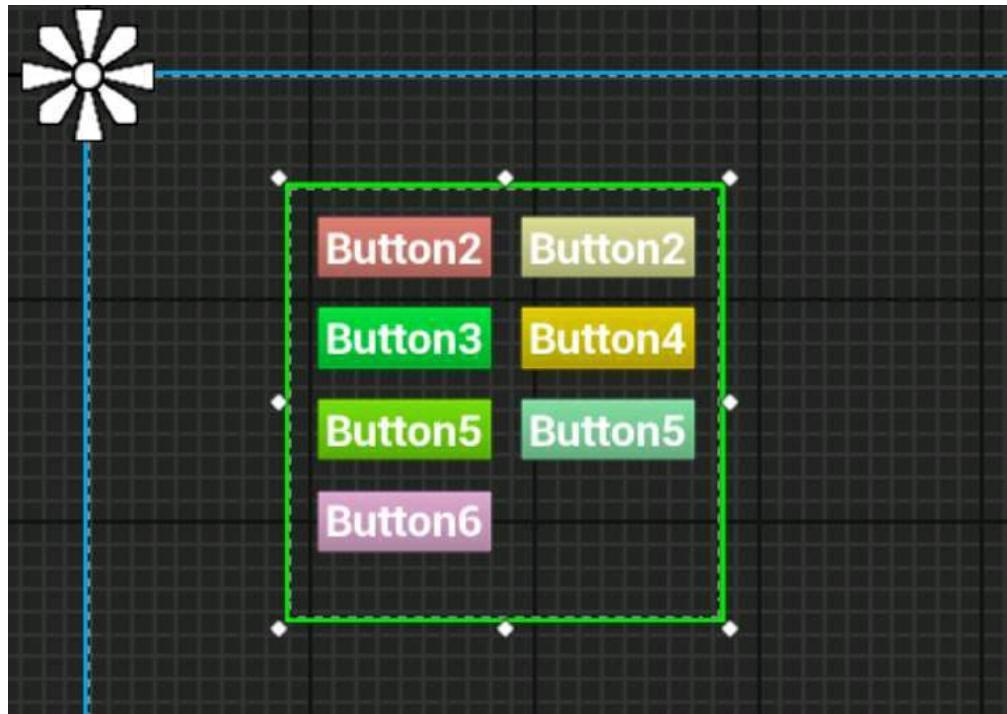


图15-18 UMG扩展示例2

15.6 蓝图扩展

15.6.1 蓝图函数库扩展

扩展蓝图节点，最简单的应该是基于UBlueprint Function Library的扩展。也就是我们一般称的“蓝图函数库”。蓝图函数库的主要作用是把一些逻辑封装成一个单独的节点，增加节点的复用性。且蓝图函数库里面的函数是“全局”函数，也就是说，我们可以在任何地方使用蓝图函数库里面的节点。

在编辑器里，我们一般直接创建一个Blueprint Function Library（如图15-19所示），然后打开蓝图节点编辑器，开发自己的蓝图函数库。

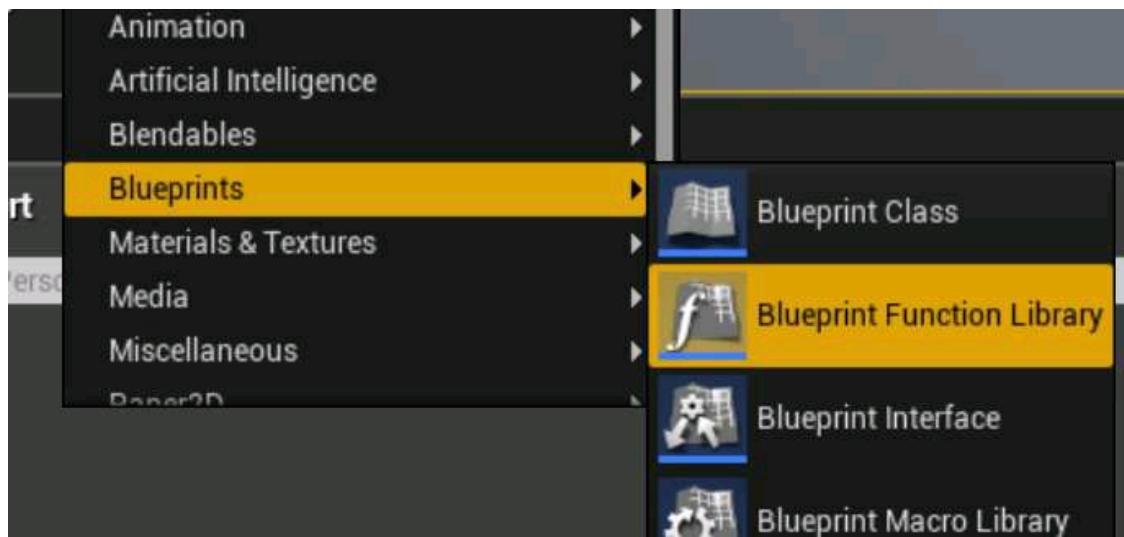


图15-19 创建Blueprint Function Library

而在C++里，我们需要创建自己的类，并继承
UBlueprintFunctionLibrary。然后把你要封装的代码写成静态函数即可。

在这里有一个简单的例子：在源码的Classes目录下，新建两个文件，如图15-20所示。

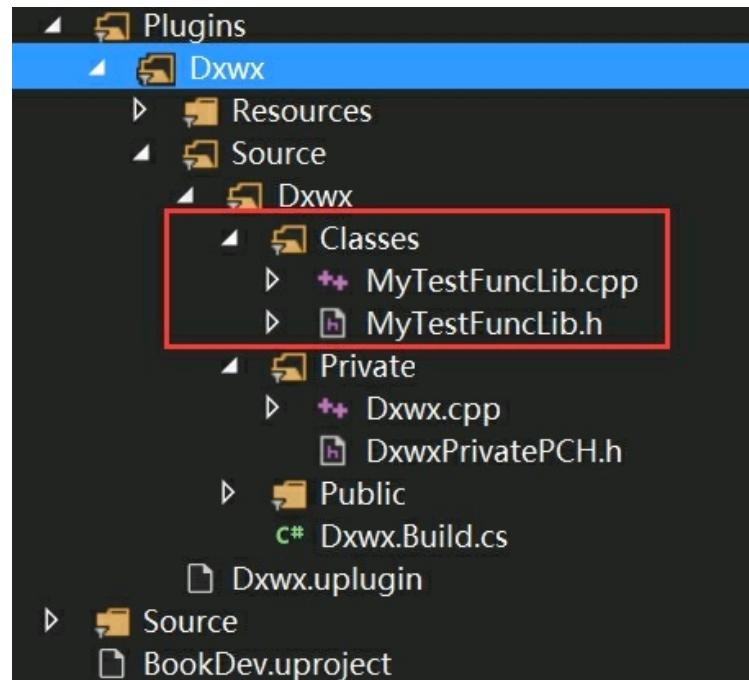


图15-20 新建两个文件

```
// File - MyTestFuncLib.h

#pragma once

#include "Kismet/BlueprintFunctionLibrary.h"
#include "MyTestFuncLib.generated.h"

UCLASS()
class DXWX_API UMyTestFuncLib:public UBlueprintFunctionLibrary
{
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable, Category = "Sanwu|TestFuncLib")
        static FString SayHello();

};
```

以上是头文件的定义，有几点需要解释一下。

首先，引用了BlueprintFunctionLibrary.h，因为我们需要继承它，它就是蓝图函数库的基类。

然后，引用MyTestFuncLib.generated.h，因为所有继承自UObject的类，都需要引用xxx.generated.h，并且**xxx.generated.h**必须是最后一个被**Include**。

接下来我们定义类，在class关键字后加DXWX_API的作用是将这个类暴露出来，可以更方便地给其他模块使用。不过在这个例子中，也可以不写它，并不影响本例的演示。

最后，我们在类里定义了一个叫SayHello的静态函数。这个函数没

有任何参数，只返回一个FString字符串。我们需要用UFUNCTION宏来进行配置SayHello（关于UFUNCTION宏详细说明，请参考对应的章节）。

BlueprintCallable 是指明此函数可以在蓝图里调用。

Category 是函数的分类。在蓝图编辑器里，点击鼠标右键会打开上下文菜单。我们就可以用Category来指定Say Hello这个函数具体放在什么地方。“|”符号的作用是用来进行分级，“Sanwu|TestFuncLib”的意思是说Say Hello函数放在Sanwu分类下的Test Func Lib类目下，如图15-21所示。

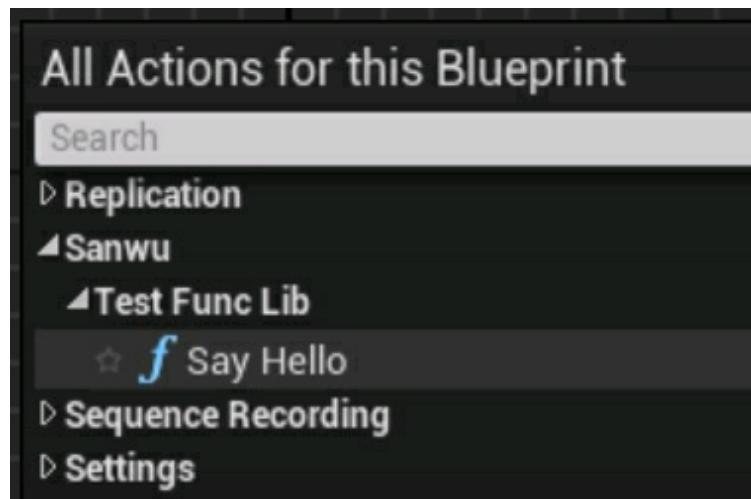


图15-21 SayHello函数位置

```
//File - MyTestFuncLib.cpp
#include "DxwxPrivatePCH.h"
#include "MyTestFuncLib.h"

FString UMyTestFuncLib::SayHello()
{
    return FString("Hello World");
}
```

```
}
```

在Cpp文件中，我们首先引用了项目的pch文件，这是虚幻引擎4规定的所有源文件必须包含项目的pch文件。然后我们实现了Say Hello函数，在函数内我们返回了一个FString，内容为HelloWorld。

最后，在虚幻引擎4中进行一下简单的测试。在关卡蓝图，节点如图15-22所示。

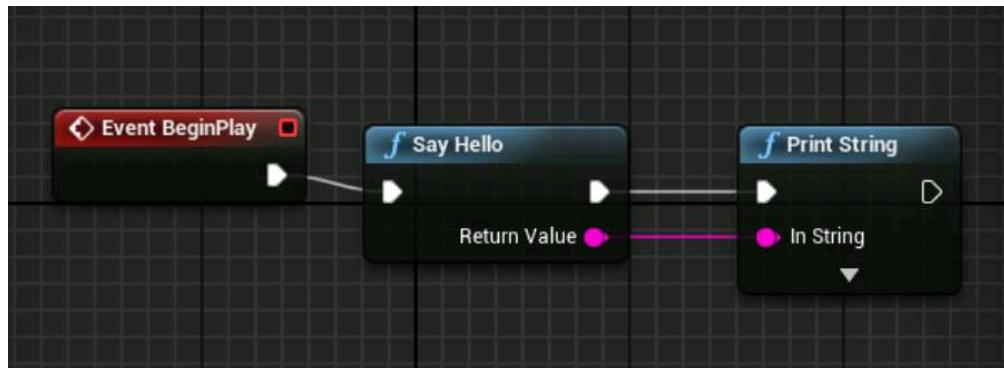


图15-22 关卡蓝图节点

我们在游戏开始的时候，调用Say Hello函数，并把返回值打印到屏幕上。

15.6.2 异步节点

有时候，我们需要一个异步执行的节点，并且这个节点会有一个或多个结果。并且在执行这个节点的时候，可以同时做其他的事。一个典型例子就是Http登录请求，客户端输出用户名和密码，然后向服务端发起一个请求，我们分析一下这个业务。

首先，登录请求需要一些数据：

1. 服务器地址
2. 用户名
3. 密码

然后，发送这个请求后。根据情况不同，会产生不同的结果：

1. 登录成功
2. 连接服务器失败
3. 用户名不存在
4. 密码不正确

最终节点看起来应该是如图15-23所示。

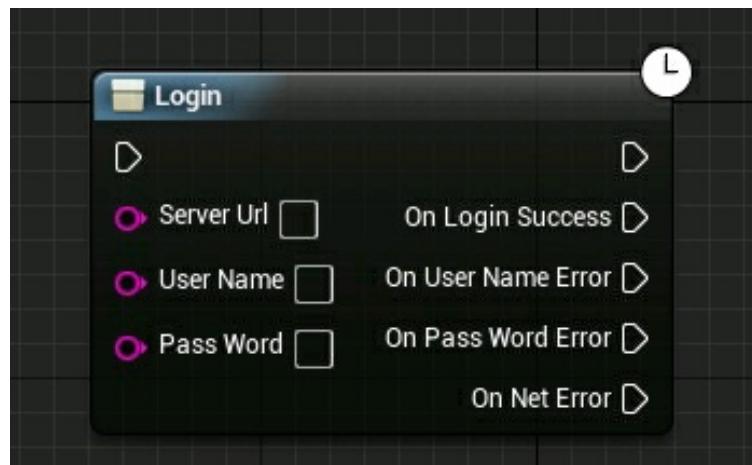


图15-23 最终效果

代码编写

首先，在插件Classes目录下创建AsyTest.h和AsyTest.cpp两个文件。
接着在AsyTest.h初步定义节点结构。

```
// AsyTest.h
```

```
#pragma once

#include "Kismet/BlueprintAsyncActionBase.h"
#include "AsyTest.generated.h"

DECLARE_DYNAMIC_MULTICAST_DELEGATE(FMyTestDelegate);

UCLASS()
class UAsyTest : public UBlueprintAsyncActionBase
{
    GENERATED_UCLASS_BODY()

public:
    UFUNCTION(BlueprintCallable, Category = "Sanwu", meta = (
        BlueprintInternalUseOnly = "true"))
    static UAsyTest* Login(FString ServerUrl, FString UserName, F
        Password);

    UPROPERTY(BlueprintAssignable)
    FMyTestDelegate OnLoginSuccess;

    UPROPERTY(BlueprintAssignable)
    FMyTestDelegate OnUserNameError;

    UPROPERTY(BlueprintAssignable)
    FMyTestDelegate OnPasswordError;

    UPROPERTY(BlueprintAssignable)
    FMyTestDelegate OnNetError;

};
```

异步节点继承自**UBlueprintAsyncActionBase**，所以需要引用Kismet/BlueprintAsyncActionBase.h。

DECLARE_DYNAMIC_MULTICAST_DELEGATE是自定义动态多播代理。这里定义的是一个没有参数的代理。如果要带参数，可以使用带参数的宏，如：

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_TwoParams(FMyTestDelegate , FS  
, a, int32, b);
```

Login函数定义需要注意的是，meta里需要定义 BlueprintInternalUseOnly="true"。这会隐藏默认的节点样式。

4个FMyTestDelegate，即节点的四个引脚。调用相应的引脚，其实就是执行相应的代理。

```
// AsyTest.cpp  
  
#include "SimpleWindowPrivatePCH.h"  
  
#include "AsyTest.h"  
  
UAsyTest::UAsyTest(const FObjectInitializer& ObjectInitializer)  
:Super(ObjectInitializer)  
{  
}  
  
UAsyTest* UAsyTest::Login(FString ServerUrl, FString UserName,  
FString Password)  
{  
    UAsyTest* Instance = NewObject<UAsyTest>();
```

```
    return Instance;  
}
```

在源文件中，初始化Login函数，返回一个UAsyTest指针。

最后，我们添加Http登录功能。使用Http模块，要在Build.cs里添加相应的依赖。打开Build.cs，在PrivateDependencyModuleName里添加“HTTP”模块。

然后，在头文件引用http.h和IHttpRequest.h。最终代码如下：

```
// AsyTest.h  
  
#pragma once  
  
#include "Http.h"  
#include "IHttpRequest.h"  
#include "Kismet/BlueprintAsyncActionBase.h"  
#include "AsyTest.generated.h"  
  
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FMyTestDelegate);  
  
UCLASS()  
class UAsyTest : public UBlueprintAsyncActionBase  
{  
    GENERATED_UCLASS_BODY()  
  
public:  
    UFUNCTION(BlueprintCallable, Category = "Sanwu", meta = (  
        BlueprintInternalUseOnly = "true"))  
    static UAsyTest* Login(FString ServerUrl, FString UserName, F  
        Password);
```

```
UPROPERTY(BlueprintAssignable)
FMyTestDelegate OnLoginSuccess;

UPROPERTY(BlueprintAssignable)
FMyTestDelegate OnUserNameError;

UPROPERTY(BlueprintAssignable)
FMyTestDelegate OnPasswordError;

UPROPERTY(BlueprintAssignable)
FMyTestDelegate OnNetError;

void PostLogin(FString ServerAddr, FString UserName, FString
    PassWord);
void OnLoginComplete(FHttpRequestPtr HttpRequest , FHttpResponse
    HttpResponse , bool bSucceeded);
};

//AstTest.cpp
#include "SimpleWindowPrivatePCH.h"
#include "AsyTest.h"

UAsyTest::UAsyTest(const FObjectInitializer& ObjectInitializer)
:Super(ObjectInitializer)
{
}

UAsyTest* UAsyTest::Login(FString ServerUrl, FString UserName,
    FString Password)
{
```

```
UAsyTest* Instance = NewObject<UAsyTest >();
Instance->PostLogin(ServerUrl, UserName, Password);
return Instance;
}

void UAsyTest::PostLogin(FString ServerAddr, FString UserName,
FString PassWorld)
{
    TSharedRef<class IHttpRequest > HttpRequest = FHttpModule::Get()
        CreateRequest();
    HttpRequest->OnProcessRequestComplete().BindUObject(this, &UAsyTest
        ::OnLoginComplete);
    HttpRequest->SetURL(ServerAddr);
    HttpRequest->SetContentAsString(UserName + "" + PassWorld);
    HttpRequest->SetVerb(TEXT("POST"));
    HttpRequest->SetHeader(TEXT("Content-Type"), TEXT("application/x-www-form-urlencoded"),
        charset=utf-8"));
    HttpRequest->ProcessRequest();
}

void UAsyTest::OnLoginComplete(FHttpRequestPtr HttpRequest,
FHttpResponsePtr HttpResponse , bool bSucceeded)
{
    if ( !bSucceeded )
    {
        OnNetError.Broadcast();
        return;
    }
    if ( HttpResponse ->GetResponseCode( )==200 )

```

```
{  
    if (HttpResponse ->GetContentAsString() == "Success")  
    {  
        OnLoginSuccess.Broadcast();  
    }  
    else if (HttpResponse ->GetContentAsString() == "UserNameError")  
        OnUserNameError.Broadcast();  
    }  
    else if (HttpResponse ->GetContentAsString() == "PasswordError")  
        OnPassWordError.Broadcast();  
    }  
    else {  
        OnNetError.Broadcast();  
    }  
}  
else {  
    OnNetError.Broadcast();  
}  
}
```

15.7 第三方库引用

15.7.1 lib静态链接库的使用

本节我们将涉及Windows平台下静态链接库(.lib)的接入。lib的接入

依赖UBT（Unreal Building System）系统，因此核心代码将在插件的Build.cs中。

首先，我们需要一个静态链接库。与其拿别人的，不如自己创造。如果你对静态链接库不了解，或者不知道如何自己创建，请自行到网上查找相关资料。在这里，笔者对于创建lib只进行一些必要的说明。

1. 启动**VS**，新建一个**Win32**项目，名称为“**LibTest**”，应用程序类型为“静态链接库”。
2. 右键点击“解决方案”，在弹出的菜单中执行“添加”→“类命令”，添加一个名叫“**MyLib**”的类。
3. 在**VS**工具栏上，找到“解决方案平台”，并选择“**x64**”。
4. 分别在**MyLib**类的头文件与源文件中添加以下代码。

```
// MyLib.h
#pragma once
//通过半径计算面积
float getCircleArea(float radius);
///////////
#include "MyLib.h"
float getCircleArea(float radius)
{
    return 3.1415f*(radius*radius);
}
```

在此，lib只定义了一个函数用于圆半径计算。

最后对解决方案单击鼠标右键，选择“生成”。生成完后，我们会得到两个文件“LibTest.lib”和“MyLib.h”。

将这两个文件放到插件目录下，具体放在什么地方没有强制规定。为了更好地理解，笔者将这两个文件放到了“SimpleWindow/ThirdPart/x64”目录下。

接下来，开始修改Build.cs。

```
using UnrealBuildTool;
using System;
using System.IO;
public class Simplewindow : ModuleRules
{
    private string ThirdPartPath
    {
        get{
            return Path.GetFullPath(Path.Combine(ModuleDirectory,
                ..../ThirdPart"));
        }
    }
    private bool LoadThirdPartLib(TargetInfo Target)
    {
        if((Target.Platform == UnrealTargetPlatform.Win64)
            || (Target.Platform == UnrealTargetPlatform.Win32))
        {
            string plartformStr = Target.Platform ==
```

```
        UnrealTargetPlatform.Win64 ? "x64" : "x86";

        string lib_path = Path.Combine(ThirdPartPath,
            plartformStr, "LibTest.lib");
        PublicAdditionalLibraries.Add(lib_path);

        PrivateIncludePaths.Add(Path.Combine(ThirdPartPath,
            plartformStr));
        return true;
    }

    return false;
}

public Simplewindow(TargetInfo Target)
{
    //调用加载静态链接库
    LoadThirdPartLib(Target);
    PublicIncludePaths.AddRange(
        new string[] {
            "Simplewindow/Public"
        });
    .....
    .....
    .....
}

}
```

至此，静态链接库的引入已经全部完成。我们来看看在Build.cs里都干了些什么？

首先，定义了一个获取到ThirdPart目录的属性，通过get方法获得ModuleDirectory上两级的ThirdPart目录。这里的ModuleDirectory指的是Build.cs所在的目录。

LoadThirdPartLib函数用于加载lib,其中传入了TargetInfo（即平台相关信息）。首先对平台进行判断，只有Windows平台的32位与64位才能加载（虽然我们只制作了64位的lib，但还是留出了32位的接口）。然后，通过判断不同平台，选择不同的目录。即x64还是x86。

找到了正确的lib的路径和头文件所在的目录后，将其分别添加到PublicAdditionalLibraries与PrivateIncludePaths。到此，引入完成。

如何使用 `getCircleArea` ? 在需要的地方，引用“`MyLib.h`”然后直接调用`getCircleArea`函数。不过，需要注意的是，在代码示例中，头文件只添加到了`PrivateIncludePaths`，如果需要，也可以添加到`PublicIncludePaths`中。

15.7.2 dll动态链接库的使用

与静态链接库不同，动态链接库(dll)的引入不需要UBT系统，我们可以直接在C++代码中进行动态引入。

所以，我们需要先创建一个dll，步骤与创建lib类似。

1. 启动**VS**，新建一个**Win32**项目，名称为“**DllTest**”，应用程序类型为“动态链接库”。
2. 右键点击“解决方案”，在弹出的菜单中执行“添加”→“类命令”，添加一个名叫“**MyDllTest**”的类。

3. 在**VS**工具栏上，找到“解决方案平台”，并选择“**x64**”。
4. 分别在**MyDllTest**类的头文件和源文件添加以下代码。

```
// MyDllTest.h
#pragma once

#define DLL_EXPORT __declspec(dllexport)

#ifndef __cplusplus
extern "C"
{

#endif

    float DLL_EXPORT getCircleArea(float radius);

#ifndef __cplusplus
}

#endif

///////////
// MyDllTest.cpp
#include "stdafx.h"
#include "MyDllTest.h"

float DLL_EXPORT getCircleArea(float radius)
{
    return float(3.1415925*(radius*radius));
}
```

最终生成项目后，会得到一个**DllTest.dll**的文件，把它放到插件目录里。同样，对于路径没有强制要求。这次为了不一样，笔者将它放到**Resources**目录里，也就是**Slate**小节中，我们的图片、字体等资源存放的

目录。

接下来，回到插件代码中来测试dll引入。在SimpleWindow.h中，添加以下代码：

```
// dll引入相关
typedef float(*_getCircleArea)(float radius);
_getCircleArea m_getCircleAreaFromDll;
void *dllHandle;
bool loadDll();
```

首先，根据dll的内容，我们使用typedef的方式来定义了getCircleArea的函数原型。并声明以此函数原型为类型的指针m_getCircleAreaFromDll。*dllHandle用于持有dll的句柄。然后，切换到SimpleWindow.cpp中，实现loadDll函数。

```
bool FSimpleWindowModule::loadDll()
{
    FString dllpath = FPaths::GamePluginsDir() / TEXT("SimpleWindow
        Resources/TestDll.dll");
    if (FPaths::FileExists(dllpath))
    {
        dllHandle = FPlatformProcess::GetDllHandle(*dllpath);
        if (dllHandle != NULL)
        {
            m_getCircleAreaFromDll = NULL;
```

```
FString procName = "getCircleArea";
m_getCircleAreaFromDll=
(_getCircleArea)FPlatformProcess::GetDllExport(dllHandle,
procName);
if (m_getCircleAreaFromDll==NULL)
{
    return false;
}
float area = m_getCircleAreaFromDll(100.f);
UE_LOG(LogSimpleApp, Warning, TEXT("第三方DLL计算Area %f",
area));
return true;
}
return false;
}
```

在loadDll函数中，首先需要找到dll文件的路径。然后通过FPlatformProcess::GetDllHandle的方式，将dll载入，并获取到它的句柄。验证句柄的正确性后，通过FPlatformProcess::GetDllExport函数，传入句柄与函数名，可以获取到dll中的函数。

第16章 自定义资源和编辑器

问题

我该如何创建自己的资源类型？怎么写一个自己的编辑器去编辑自己的资源？能不能显示一个**3D**的预览界面？

笔者认为，专业的引擎使用者在拿到虚幻引擎时，一定会针对自身的需求、现有的技术集合及公司的实际情况定制引擎。例如：希望创建自己的资源格式，包含自己想要的数据，然后允许策划在编辑器中填写数据，甚至有可能预览数据在运行时的表现。

并且，对于插件开发者来说，给插件使用者提供自定义的资源与编辑器，能够让插件使用者更好地使用这个插件。

16.1 简易版自定义资源类型

你只需要做一步：在虚幻引擎中新建一个C++类（笔者取名为ExampleDataAsset），继承自UDataAsset类。

编译完成后，你已经可以创建对应的数据资源了。方法是在内容浏览器的空白处单击鼠标右键，执行“其他”→“数据资源”命令。你已经能看到你的ExampleDataAsset了，选择ExampleDataAsset并点击确定。

你会发现内容浏览器中出现了一个数据资源。双击打开，此时会看到空白的细节面板。为了让我们的细节面板能显示一些东西，在生成的 ExampleDataAsset.h 中，添加几个成员变量，让这个类看上去像这样：

```
UCLASS()
class UExampleDataAsset : public UDataAsset
{
GENERATED_BODY()
public:
UPROPERTY(EditAnywhere)
FString StringProperty;
UPROPERTY(EditAnywhere)
float    FloatProperty;
};
```

退出引擎并重新编译，然后重复刚才的创建数据资源的步骤。如果结果正确，当你打开刚刚创建好的数据资源后，会看到这样的面板，如图16-1所示。



图16-1 自定义的DataAsset资源，此时被UPROPERTY标记的变量已经可以编辑

你可以在某个Actor中创建UEExampleDataAsset*类型的成员变量，并将这个资源直接赋值给那个成员变量。为了实现这样的目的，需要按照如下方式改写UEExampleDataAsset的声明。

```
UCLASS(Blueprintable)
class UEExampleDataAsset : public UDataAsset
```

之后你就可以在其他蓝图的类的“变量”面板创建UEExampleDataAsset类型的变量，如图16-2所示。



图16-2 选择变量类型为Example Data Asset

如果你打开了左侧变量面板中对应变量在编辑器中可视的选项（即“眼睛”图标），你就可以在当前蓝图的对象细节面板中找到这个变量，并选择你刚刚创建的UEexampleDataAsset类型的资源，对其进行赋值。

16.2 自定义资源类型

虚幻引擎提供了自定义资源非常便捷的接口。你只需要执行两步：

1. 创建一个继承自UObject的类，该类将会作为你自定义资源的数据模型类。你可以定义自己的字段来存储你希望存储的信息。
2. 创建一个工厂类，该类将会被虚幻引擎识别，然后作为在内容管理器中创建资源的工厂，可以在这里设置你的自定义资源的数据模型的初始值等。

16.2.1 切分两个模块

虚幻引擎的模块分为Runtime、Development和Editor，只有Runtime模块是在运行时包含的，剩余两个模块都不会在发布版本中包含。而在自定义资源中，只有你的数据模型类是需要包含在运行时的，对数据模型进行编辑的类并不需要包含在运行时的构建版本中。所以我们需要分为两个模块。

笔者使用的工程名字为TestProject，当前的模块结构如下：

```
/Source
├── TestProject.Target.cs
├── TestProjectEditor.Target.cs
└── /TestProject
    ├── /Private
    │   └── ExampleDataAsset.h
    │   └── ExampleDataAsset.cpp
    ├── TestProject.Build.cs
    ├── TestProject.h
    └── TestProject.cpp
```

调整文件夹结构，并创建缺少的文件，形成如下的文件结构：

```
/Source
├── TestProject.Target.cs
├── TestProjectEditor.Target.cs
└── /TestProject
    ├── /Private
    │   └── ExampleDataAsset.h
    │   └── ExampleDataAsset.cpp
    ├── TestProject.Build.cs
    ├── TestProject.h
    └── TestProject.cpp
└── /TestProjectEditor
    ├── TestProjectEditor.Build.cs
    ├── /Public
    │   └── TestProjectEditor.h
    └── /Private
        └── TestProjectEditorPrivatePCH.h
        └── TestProjectEditor.cpp
```

也就是说，我们新增了TestProjectEditor模块，该模块用于存放所有编辑器相关但是运行时不需要的类。对应的文件如下。

TestProjectEditor.Build.cs

```
using UnrealBuildTool;
public class TestProjectEditor : ModuleRules
{
public TestProjectEditor(TargetInfo Target)
{
    PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore" });
}
}
```

TestProjectEditor.h

```
#ifndef __TESTPROJECTEDITOR_H__
#define __TESTPROJECTEDITOR_H__
#include "EngineMinimal.h"
#endif
```

TestProjectEditor.cpp

```
#include "TestProjectEditorPrivatePCH.h"
#include "TestProjectEditor.h"
IMPLEMENT_PRIMARY_GAME_MODULE( FDefaultGameModuleImpl ,
    TestProjectEditor , "TestProjectEditor" );
```

其中TestProjectEditorPrivatePCH.h内容可以为空，或者如笔者一样包含Engine.h。

完成创建后，修改TestProjectEditor.Target.cs文件，增加TestProjectEditor模块：

TestProjectEditor.Target.cs

```
using UnrealBuildTool;
using System.Collections.Generic;
public class TestProjectEditorTarget : TargetRules
{
    public TestProjectEditorTarget(TargetInfo Target)
    {
        Type = TargetType.Editor;
    }
    //
    // TargetRules interface.
    //
    public override void SetupBinaries(
        TargetInfo Target,
        ref List<UEBuildBinaryConfiguration> OutBuildBinaryConfigurations
        ref List<string> OutExtraModuleNames
    )
    {
        OutExtraModuleNames.Add("TestProject");
        OutExtraModuleNames.Add("TestProjectEditor");
```

```
}
```

```
}
```

在uproject文件上右键，重新生成解决方案。然后使用Visual Studio 打开解决方案并生成，不出意外，此时已经正常生成了解决方案。

16.2.2 创建资源类

此时的资源类可以不再像“简易版自定义资源类型”那样继承自 UDataAsset类，而是可以直接继承自UObject类。为了保持一致，笔者此处只是直接修改UEExampleDataAsset类继承自UObject类。即修改：

```
class UEExampleDataAsset : public UDataAsset
```

为：

```
class UEExampleDataAsset : public UObject
```

16.2.3 在Editor模块中创建工厂类

如果此时运行项目，会发现已经不能在内容浏览器中创建 ExampleDataAsset实例了。因为我们需要手动定义工厂类来告知虚幻引擎如何创建ExampleDataAsset。具体方法如下：

引入**Editor**模块

修改TestProjectEditor.Build.cs，增加对UnrealEd模块的依赖。当然也要添加TestProject模块的依赖。

TestProjectEditor.Build.cs

```
using UnrealBuildTool;
public class TestProjectEditor : ModuleRules
{
    public TestProjectEditor(TargetInfo Target)
    {
        PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "UnrealEd", "TestProject" });
    }
}
```

增加**UEExampleAssetFactory**类

在TestProjectEditor模块中新增UEExampleAssetFactory类的头文件和定义，修改后的目录结构如下：

```
/Source
└── TestProject.Target.cs
└── TestProjectEditor.Target.cs
└── /TestProject
    └── /Private
        └── ExampleDataAsset.cpp
    └── ExampleDataAsset.h
    └── TestProject.Build.cs
    └── TestProject.h
    └── TestProject.cpp
└── /TestProjectEditor
    └── TestProjectEditor.Build.cs
    └── /Public
        └── TestProjectEditor.h
        └── ExampleDataAssetFactory.h
    └── /Private
        └── TestProjectEditor.cpp
        └── ExampleDataAssetFactory.cpp
```

即在Public文件夹下新增了ExampleDataAssetFactory.h文件，在Private文件夹下新增了ExampleDataAssetFactory.cpp文件。然后将ExampleDataAsset.h文件从Private文件夹中移出，方便引用。

ExampleDataAssetFactory.h

```
#pragma once
#include "Engine.h"
#include "UnrealEd.h"
#include "ExampleDataAssetFactory.generated.h"
UCLASS()
class UExampleDataAssetFactory : public UFactory
```

```
{  
GENERATED_UCLASS_BODY()  
public:  
// UFactory interface  
virtual UObject* FactoryCreateNew(UClass* Class, UObject* InParent,  
FName Name, EObjectFlags Flags, UObject* Context, FFeedbackContext  
* Warn) override;  
// End of UFactory interface  
};
```

ExampleDataAssetFactory.cpp

```
#include "TestProjectEditorPrivatePCH.h"  
#include "ExampleDataAssetFactory.h"  
#include "ExampleDataAsset.h"  
#define LOCTEXT_NAMESPACE "UEExampleDataAssetFactory"  
UEExampleDataAssetFactory::UEExampleDataAssetFactory(const  
FObjectInitializer& ObjectInitializer)  
: Super(ObjectInitializer)  
{  
bCreateNew = true;  
bEditAfterNew = true;  
SupportedClass = UEExampleDataAsset::StaticClass();  
}  
UObject* UEExampleDataAssetFactory::FactoryCreateNew(UClass* Class,  
	UObject* InParent, FName Name, EObjectFlags Flags, UObject*
```

```
Context, FFeedbackContext* Warn)
{
UEExampleDataAsset* NewObjectAsset = NewObject<UEExampleDataAsset>
    (InParent, Class, Name, Flags | RF_Transactional);
return NewObjectAsset;
}

#define LOCTEXT_NAMESPACE
```

UFactory类就是用于实现在内容浏览器中创建对象的工厂类。继承之后需要完成以下两步操作：

1. 提供构造函数，指定当前工厂类支持的类型。即对SupportedClass赋值。
2. 重载FactoryCreateNew函数，指定在内容浏览器中创建对象时应当完成的操作。标准的操作是通过NewObject创建一个资源类的实例，也可以在此时设置实例的一些初始化属性。

此时如果编译，会发现提示无法解析的外部符号。这是因为我们的UEExampleDataAsset类没有被作为DLL导出符号导出。因此我们需要在UEExampleDataAsset类前增加导出标记宏TESTPROJECT_API。

```
class TESTPROJECT_API UEExampleDataAsset : public UDataAsset
```

16.2.4 引入Editor模块

修改.uproject文件（可以使用记事本，或者直接用Visual Studio打

开) :

TestProject.uproject

```
{  
    "FileVersion": 3,  
    "EngineAssociation": "4.11",  
    "Category": "",  
    "Description": "",  
    "Modules": [  
        {  
            "Name": "TestProject",  
            "Type": "Runtime",  
            "LoadingPhase": "Default",  
            "AdditionalDependencies": [  
                "Engine"  
            ]  
        },  
        {  
            "Name": "TestProjectEditor",  
            "Type": "Editor",  
            "LoadingPhase": "Default"  
        }  
    ]  
}
```

这里将会告知工程文件载入TestProjectEditor模块，并指定TestProjectEditor模块的类型为Editor。如果读者是在一个插件中实现自定义的资源文件，那么修改的就是当前插件根目录下的.uplugin文件。

编译并运行整个工程。在内容浏览器中右键，执行“其他”→“ExampleDataAsset”命令，就能够创建一个ExampleDataAsset的实例。双击即可打开一个和简易版类似的属性编辑界面。

回顾整个编辑器的实现过程，虽然步骤烦琐，但是思路非常简单扼要：创建资源类，并创建资源工厂。在资源工厂类中指定生产的对象类型（SupportedClass）及生产的过程（FactoryCreateNew函数）。虚幻引擎通过反射机制识别工厂类，并自动添加到内容浏览器的右键菜单的处理过程中。

16.3 自定义资源编辑器

当我们自定义完资源后，一定希望能够提供一个更符合自己资源的编辑界面来完成编辑。例如提供预览窗口、提供对象树查看等，甚至如同虚幻引擎知名插件——地下城建筑师一样，提供一个基于行为树的编辑界面来定义地下城的生成逻辑。

为了能够实现使用自定义的虚幻引擎编辑器来完成对自定义资源的编辑，实质上需要完成以下四个步骤：

1. 增加继承自FAssetTypeActions_Base类的AssetTypeAction类。
2. 定义你需要打开的编辑器Editor类，继承自FAssetEditorToolkit。
3. 在模块加载的时候注册AssetTypeAction，来告知虚幻引擎针对你的

自定义资源采用你的类处理。

4. 在AssetTypeAction中重载OpenAssetEditor方法，以打开自定义的资源编辑器。

在刚才描述的内容中出现了以下几个新的类型：

FAssetTypeActions_Base 当虚幻引擎在内容浏览器中处理资源操作（创建资源、双击打开编辑资源等）时，会搜寻是否有对应的资源类型操作类被注册，如果没有，就使用默认的类，否则会调用对应类的实例完成操作。

FAssetEditorToolkit 处理资源编辑器操作的抽象类。其提供了包括打开资源编辑器窗口、修改资源编辑器窗口的面板类型、修改资源编辑器窗口的工具栏等功能。

需要注意的是，这两个类型是有包含关系的：

FAssetTypeActions_Base包含全部的资源类型操作，比如资源在右键菜单创建时所属的分类等，而FAssetEditorToolkit是完成了资源编辑 这一个功能的类型，其持有真正创建的编辑器界面的指针，但是我们一般不会直接操作编辑器界面的智能指针，如图16-3所示。

由于整个资源自定义代码较多，逐行呈现并不方便读者理解和阅读，因此笔者按照前文所述的步骤，摘取关键代码供读者参考理解。

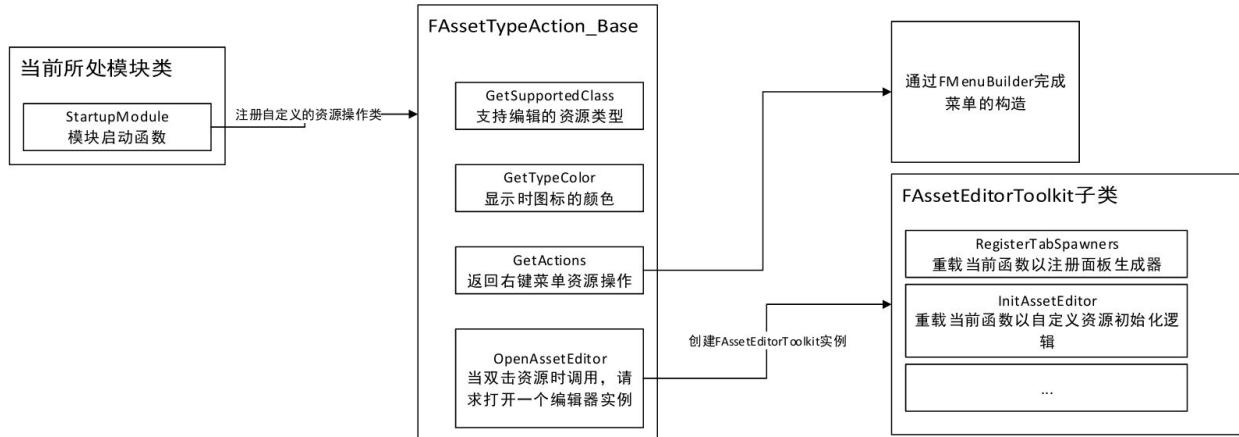


图16-3 自定义资源编辑器中所包含的类的关系

16.3.1 资源操作类

假设读者此时已经创建了自己的资源操作类**FMyAssetTypeActions**的声明，但尚未提供任何成员函数实现。请注意声明时需要在当前模块的.Build.cs文件中引入AssetTools模块，并包含AssetTypeActions_Base.h头文件，随后读者需要完成以下步骤：

1. 实现参数为EAssetTypeCategories::Type类型的名为InAssetCategory的构造函数，并将其放入一个类私有成员变量MyAssetCategory中保存起来。这是为了在GetCategories中使用。
2. 实现GetName函数，返回当前资源类型名称。
3. 实现GetSupportedClass函数，返回自定义资源类型的UClass*类型信息。例如：

```

UClass* FMyAssetTypeActions::GetSupportedClass() const
{
    return UExampleDataAsset::StaticClass();//
    UExampleDataAsset即为你希望编辑的类的名字
}

```

```
}
```

4. 实现GetTypeColor函数，返回FColor类型的颜色变量。例如返回FColor::Green。
5. 实现GetActions函数，可以不做任何处理。
6. 实现GetCategories函数，返回第一条所保存的私有成员变量返回右键菜单资源操作MyAssetCategory的值。
7. 实现GetThumbnailOverlay函数，此时可以返回一个SWidget的Slate控件，以覆盖当前资源的图标。
8. 实现OpenAssetEditor函数，该函数为最核心的函数，其需要创建一个继承自FAssetEditorToolkit类型的子类的实例，然后调用其初始化函数，读者此时可以暂时将此函数置空不写，具体内容会在下文做详细描述。

一旦完成资源操作类的实现，就可以在当前模块的StartupModule函数中对当前资源操作类进行注册。

前文中使用的是简化版的声明：

TestProjectEditor.cpp

```
#include "TestProjectEditorPrivatePCH.h"  
#include "TestProjectEditor.h"  
  
IMPLEMENT_PRIMARY_GAME_MODULE( FDefaultGameModuleImpl ,  
    TestProjectEditor , "TestProjectEditor" );
```

而此时就必须使用完整版的模块类实现，具体方式为：

模块头文件.h

```
#include "Engine.h"
#include "ModuleManager.h"

class FMyModule : public IModuleInterface //FMyModule为模块类名，最
                                            与模块名称一致

{
public:
    /** IModuleInterface implementation */
    virtual void StartupModule() override;
    virtual void ShutdownModule() override;
};
```

模块实现文件.cpp

```
#include "TestProjectEditorPrivatePCH.h"//当前模块预编译头
#include "模块头文件.h"

void FMyModule::StartupModule()
{
    //模块初始化代码
    //...
}

void FMyModule::ShutdownModule()
{
    //模块停止代码
    //...
}
```

```
}
```

```
IMPLEMENT_MODULE(FMyModule, 模块名)//在这里将MyModule类注册为当前模块  
的实现
```

此时即可在StartupModule函数中对当前定义的FMyAssetTypeActions进行注册：

模块实现文件.cpp

```
//...  
  
void FMyModule::StartupModule()  
{  
//...  
  
IAssetTools& AssetTools = FModuleManager::LoadModuleChecked<  
    FAssetToolsModule >("AssetTools").Get(); //获取资源工具模块  
EAssetTypeCategories::Type MyAssetCategoryBit = AssetTools.  
    RegisterAdvancedAssetCategory(FName(TEXT("MyAssetCategory")),  
        LOCTEXT("MyCategoryName", "资源分类名")); //此处为注册自己的资源分  
    类，否则会显示在“其他”这个分类中  
  
AssetTools.RegisterAssetTypeActions(MakeShareable(new  
    FMyAssetTypeActions(MyAssetCategoryBit))); //这里创建并注册实例  
}  
  
//...
```

一旦注册完毕后，当你启动编辑器时，就可以在内容浏览器中右键创建一个资源，只是双击新建的资源没有任何反应，因为我们还尚未提

供OpenAssetEditor类的实现。

16.3.2 资源编辑器类

在完成资源操作类AssetTypeActions定义后，我们需要设计一个继承自FAssetEditorToolkit的类提供给OpenAssetEditor用于产生资源编辑器实例。

具体来说，步骤如下：

创建继承自**FAssetEditorToolkit**的类 对于第一次创建自定义编辑器的读者来说，可以只考虑创建一些自定义的Tab面板，然后放置一个默认的细节面板到其中一个Tab面板中。

1. 重载GetToolkitFName函数，返回当前自定义编辑器的名称（FName类型），命名可以随意，例如：

```
return FName("MyAssetEditor");
```

这只是作为标记使用。

2. 重载GetBaseToolKitName函数，返回当前自定义的编辑器的基础名称（FText类型），例如：

```
return LOCTEXT("BaseToolKitName", "自定义资源编辑器");
```

3. 重载GetWorldCentricTabPrefix函数，返回一个FString作为前缀，内容随意。
4. 重载InvokeTab函数，可以简单直接调用父类版本，以后扩展时再做扩充。

5. 重载GetWorldCentricTabColorScale，可返回FLinearColor::White。
6. 重载RegisterTabSpawners，该函数为注册“Tab生成器”的函数，非常重要，后文会专门叙述。简而言之，此处规定当前编辑器具有哪些Tab面板，并定义这些面板的内容。
7. 创建一个自定义的初始化函数，或者重载InitAssetEditor，笔者推荐创建一个自己的，因为能够设定自己的参数数量和类型。这个函数会在下文的OpenAssetEditor函数中被调用，传入当前正在编辑的对象以初始化编辑器实例。该函数同样会定义面板的布局样式，为了行文方便，此处初始化函数命名为InitObjectEditor，参数推荐为：

EToolkitMode::typeMode 当前编辑器类型参数，将会在OpenAssetEditor函数调用时传入。

const TSharedPtr<IToolkitHost>& InitToolkitHost 该参数为当前编辑器的主人，在OpenAssetEditor函数的参数中，可以直接将对应参数传递给初始化函数。

你的资源类型***Object** 实际被编辑的对象指针。

创建继承自**SSingleObjectDetailsPanel**类的自定义细节面板 需要注意的是，**SSingleObjectDetailsPanel**类是一个抽象类型，因此你需要自己创建一个类继承自该类，并实现GetObjectToObserve函数和PopulateSlot函数，才能作为细节面板使用。

1. 实现GetObjectToObserve函数，返回的是需要编辑的对象的指针。
2. 实现PopulateSlot函数，这里可以修改具体的细节面板样式，或者直接将函数传递过来的PropertyEditorWidget控件引用作为返回值返回。
3. 需要注意的是，读者实现自己的**SSingleObjectDetailsPanel**类时，务必自行实现公开访问类型的用于设置“哪个对象是正在编辑的对

象”的函数，才能实现从外部对当前编辑控件的访问。

实现**OpenAssetEditor**函数 其实到此时，这个函数的实现已经颇为容易。OpenAssetEditor函数包含：正在编辑的对象（列表）TArray<UObject*>&类型和TSharedPtr<IToolkitHost>类型的参数EditWithinLevelEditor。前文有叙述，这个参数需要传递给自定义的初始化函数InitObjectEditor作为参数。OpenAssetEditor函数最简单的实现方式就是直接遍历正在编辑的对象列表，逐个创建编辑器对象实例。

OpenAssetEditor函数实现案例

```
void FCustomCeilingAssetTypeActions::OpenAssetEditor(const
    TArray<UObject*>& InObjects, TSharedPtr<IToolkitHost >
    EditWithinLevelEditor /*= TSharedPtr<IToolkitHost >()
    */)
{
    //FAssetTypeActions_Base::OpenAssetEditor(InObjects);
    for (auto ObjIt = InObjects.CreateConstIterator(); ObjIt;
        ++ObjIt)
    {
        if (UEExampleDataAsset* CustomAsset = Cast<
            UEExampleDataAsset >(*ObjIt))
        {
            TSharedRef<你的资源编辑器类型> Editor(new 你的资源编辑器类
型());
            Editor->InitObjectEditor(
```

```
    EToolkitMode::Standalone,  
    EditWithinLevelEditor,  
    CustomAsset);  
}  
}  
}
```

面板内容注册函数

面板内容注册函数指的是**RegisterTabSpawners** 函数，该函数实质上是指定“如何产生Tab内容”，对于Tab面板的布局，是交给初始化函数完成的。这个函数名为“Register”，反映了当前函数的内容实质上是“注册”，其将产生内容的逻辑，告知负责注册的对象，并在初始化函数中才会真正执行。

面板内容注册函数需要利用到核心对象

const TSharedRef<FTabManager>& TabManager，该对象是当前函数的唯一参数。一个比较典型的注册案例代码如下。

```
TabManager->RegisterTabSpawner(  
    FName(TEXT("Details")),  
    FOnSpawnTab::CreateLambda(  
        [&](const FSpawnTabArgs& Args) {  
            return SNew(SDockTab)  
                .Icon(FEditorStyle::GetBrush("LevelEditor.Tabs.Details"))  
        }  
    )
```

```
.Label(LOCTEXT("DetailsTab_Title", "属性"))

[
//...这里利用SNew(控件类型)产生面板的内容
];
}

)

)

.SetDisplayName(LOCTEXT("DetailsTabLabel", "属性"))
.SetIcon(FSlateIcon(FEditorStyle::GetStyleSetName(), "LevelEditor
Tabs.Details"));


```

这段代码看上去极为复杂，实质上可以这样理解：

1. 从最外层来看，我们调用了TabManager的RegisterTabSpawner函数，产生了对象实例 [\[1\]](#)。这个对象实例此时是匿名的，但是可以在其基础上继续调用函数，我们调用了.SetDisplayName来设置当前面板的名称，并通过.SetIcon设置了图标。
2. RegisterTabSpawner函数接纳两个参数，第一个是当前Tab的 TabId，这个Id会在下文的初始化函数中作为标识符来确定产生的 Tab内容被放置到哪个位置。第二个是一个FOnSpawnTab类型的代理，这个代理会在实际产生内容时调用。
3. 我们创建了一个Lambda表达式形式的代理，当为一个叫Details的 Tab创建内容时，这个代理就会被调用，创建一个SDockTab的控件并返回。

在FOnSpawnTab代理创建的SDockTab中，我们就可以使用先前创建的继承自SSingleObjectDetailsPanel类的自定义细节面板类，使用

SNew创建一个实例放置在面板中，从而完成属性面板的显示。

初始化函数

此时我们的核心问题集中到了调用的初始化函数，即 InitObjectEditor这个函数的实现上。该函数的实现相对复杂，可以分为以下几个阶段。

初始化与赋值阶段 我们已经从参数中获得了正在编辑的对象的指针，这个阶段就是将该指针存储到类成员变量中。

面板布局阶段 一个典型的布局案例可以参考以下代码：

```
const TSharedRef<FTabManager::FLayout>
    StandaloneDefaultLayout = FTabManager::NewLayout(
        "Standalone_MyEditor_Layout2")
    ->AddArea
    (
        FTabManager::NewPrimaryArea()
        ->SetOrientation(Orient_Vertical)
        ->Split
        (
            FTabManager::NewStack()
            ->SetSizeCoefficient(0.1f)
            ->SetHideTabWell(true)
            ->AddTab(GetToolbarTabId(), ETabState::OpenedTab)
```

```
)  
->Split  
(  
FTabManager::NewSplitter()  
->SetOrientation(Orient_Horizontal)  
->SetSizeCoefficient(0.1f)  
->Split  
(  
FTabManager::NewSplitter()  
->SetOrientation(Orient_Vertical)  
->Split(  
FTabManager::NewStack()  
->SetSizeCoefficient(0.5f)  
->SetHideTabWell(true)  
->AddTab(FName(TEXT("Details")), ETabState::OpenedTab)//此  
处的"Details"即为上文所述的TabId  
)  
->Split  
(  
FTabManager::NewStack()  
->SetSizeCoefficient(0.5f)  
->SetHideTabWell(true)  
->AddTab(FName(TEXT("预览")), ETabState::OpenedTab)  
)  
)  
->Split  
(
```

```
FTabManager::NewStack()
->SetSizeCoefficient(0.9f)
->SetHideTabWell(true)
->AddTab(FName(TEXT("编辑器")), ETabState::OpenedTab)
)
)
);
```

可以看到，这其实是一个类似Slate的界面控制系统。当前布局的名字叫作Standalone_MyEditor_Layout2。其定义了如图16-4所示的界面。

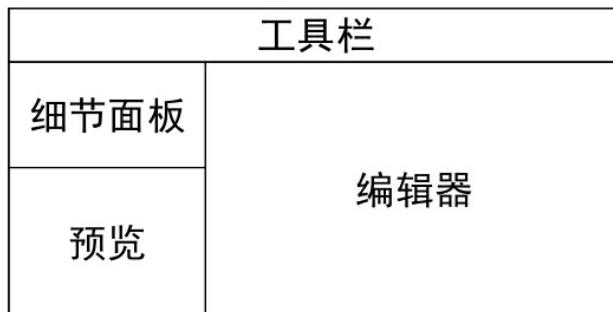


图16-4 示例编辑器的界面

调用**InitAssetEditor**函数

当前函数一定要调用父类的

InitAssetEditor函数以完成初始化。

扩展工具栏阶段 如果读者对工具栏的扩展有需求 [2]，其扩展内容可以放在此处，也可以放在**InitAssetEditor**之前。如果在**InitAssetEditor**函数调用后才调用工具栏扩展函数，则需要调用**RegenerateMenusAndToolbars**函数重新刷新菜单和工具栏。

16.3.3 增加3D预览窗口

创建一个3D预览窗口来完成自己资源的预览也是许多读者希望完成的。例如，如果希望实现一个程序化生成的资源类，自然希望能够直接在3D的视图中预览这个资源生成后的效果，而不是必须要拖动到场景中才能看到实际情况。

这个任务实质上来说并不复杂。一个3D预览窗口包含的结构，如图16-5所示。

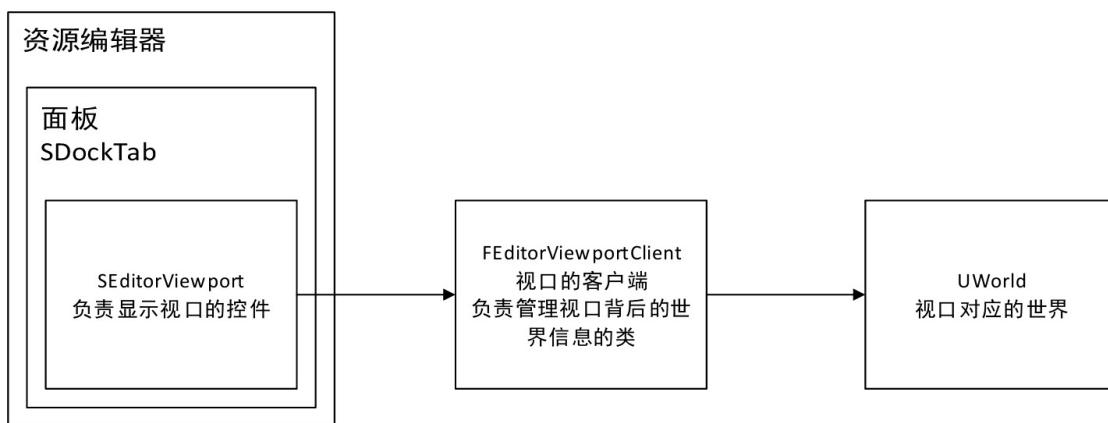


图16-5 3D预览窗口包含的结构

因此，创建自己的3D预览窗口，需要完成的步骤总体来说是三个：

创建自己的**FViewportClient**子类 **FViewportClient**类是负责处理视口背后的世界的类，创建一个自己的继承自这个类的子类，将会允许你有限地修改这个世界的运行规则。

创建自己的**SEditorViewport**子类 这个类将会使用你刚刚创建的**FViewportClient**子类的实例，以在面板中渲染对应的世界。

重载SEditorViewport::MakeEditorViewportClient 重载该函数以返回刚刚创建的FViewportClient的子类，从而将这两个类连接起来。

具体来说步骤如下：

创建自己的**FEditorViewportClient**子类

建立一个继承自FEditorViewportClient类的子类声明，假定为FExampleAssetViewportClient。接下来的步骤并不复杂，其真正的复杂度取决于读者希望在这个视口中显示什么东西。

最简单的步骤如下：

在声明中放置一个**FPreviewScene**类型的成员变量 请注意，这是一个FPreviewScene类型的成员变量，并非是指针！写法非常简单：

```
public:  
FPreviewScene OwnerScene;
```

这是一个值类型的对象，其会在当前FViewportClient类被实例化的同时就初始化，在FPreviewScene的构造函数中完成了对UWorld世界的构建、初始化，连默认的光源都放置好了。这个世界也会随着当前Client的销毁而销毁，避免手动去清理内存。

书写构造函数

这个构造函数需要传入一个
const TWeakPtr<class SEditorViewport>& InEditorViewportWidget作为
参数。一个典型的写法如下：

```
FExampleAssetViewportClient::FExampleAssetViewportClient(const  
    TWeakPtr<class SEditorViewport>& InEditorViewportWidget /*=  
    nullptr*/) : FEditorViewportClient(  
    nullptr,  
    &OwnerScene, //这个参数即刚刚提到的 FPreviewScene 的值类型对象  
    InEditorViewportWidget  
)  
{  
    PreviewScene = &OwnerScene; //PreviewScene 是父类成员变量，需要设置  
    ((FAssetEditorModeManager*)ModeTools)->SetPreviewScene(  
        PreviewScene);  
    DrawHelper.bDrawGrid = true; //打开地平面网格绘制  
    DrawHelper.bDrawPivot = true; //打开坐标轴绘制  
    SetRealtime(true); //设置当前模式为实时模式  
    //...这里可以添加各种组件到 PreviewScene。  
    PreviewMeshComponent = NewObject<UProceduralMeshComponent>(  
        PreviewScene->GetWorld(), TEXT("PreviewMesh"));  
    PreviewScene->AddComponent(PreviewMeshComponent, FTransform::  
        Identity);  
}
```

这里需要注意的有两点：

1. 一定要按照案例所示去设置PreviewScene。
2. 预览场景PreviewScene只能添加组件，而不能直接添加Actor。

重载**Tick**函数 重载后需要手动完成OwnerScene的更新：

```
void FExampleAssetViewportClient::Tick(float DeltaSeconds)
{
    FEditorViewportClient::Tick(DeltaSeconds);
    OwnerScene.GetWorld()->Tick(LEVELTICK_All, DeltaSeconds);
}
```

此时一个ViewportClient类已经准备完成。接下来需要完成控件类的准备：

视口控件类的准备

一个自定义的视口控件类的典型案例如下，假定控件类名称为 SExampleAssetViewport：

视口控件类声明

```
#pragma once
#include "SEditorViewport.h"
class FEditorViewportClient;
```

```
class SExampleAssetViewport :public SEditorViewport {
public:
SLATE_BEGIN_ARGS(SExampleAssetViewport) {}
SLATE_ATTRIBUTE(class UExampleDataAsset*, EditingAsset)//需要编辑的
    源类的对象，通过Slate属性传入
SLATE_END_ARGS()
public:
void Construct(const FArguments& InArgs);
private:
class UExampleDataAsset* EditingAsset;//正在编辑的对象，在此处存储
protected:
virtual TSharedRef<FEditorViewportClient> MakeEditorViewportClient()
    override;//重载该函数以返回自定义的Client
};
```

视口控件类实现

```
//...包含必要的头文件，例如PCH文件、FExampleAssetViewportClient的声明
等
void SExampleAssetViewport::Construct(const FArguments& InArgs)
{
EditingAsset = InArgs._EditingAsset.Get();
SEditorViewport::Construct(SEditorViewport::FArguments());
}
TSharedRef<FEditorViewportClient> SExampleAssetViewport::
    MakeEditorViewportClient()
```

```
{  
    return MakeShareable(new FExampleAssetViewportClient());//重载，返  
    自定义的Client  
}
```

当视口控件类准备完毕后，在上一节所述的“创建SDockTab内容”这一阶段，通过以下代码就能完成视口控件的创建：

```
FOnSpawnTab::CreateLambda(  
[&](const FSpawnTabArgs& Args) {  
    return SNew(SDockTab)  
        .Icon(FEditorStyle::GetBrush("LevelEditor.Tabs.Details"))  
        .Label(LOCTEXT("PreviewTab_Title", "预览"))  
        [  
            SNew(SExampleAssetViewport)  
            .EditingAsset(CurrentEditingAssetObject)//这里给出需要预览的对象指针  
        ];  
}
```

[\[1\]](#) 该对象实例类型为FTabSpawnerEntry &，故可以在其基础上直接调用函数以修改内容。

[\[2\]](#) 工具栏的扩展方式，读者可以参考 FStaticMeshEditor::ExtendToolBar 函数。