目录

**摘要 Ⅱ**

**ABSTRACT Ⅲ**

**第一章 引言 1**

1.1 前导知识 1

1.2 二级标题 1

1.2.1 三级标题 1

1.2.2 三级标题 1

**第七章 结论 3**

**参考文献 4**

**致谢 5**

## **摘要**

关键词：

## ABSTRACT

Key Words:

## 引言

根据2025年2月3日由VGI（Video Game Insights）发布的《2025年大型游戏引擎报告》[1]（《The Big Game Engines Report of 2025》），2024年有28%的游戏由UE（Unreal Engine）引擎开发，有51%的游戏由Unity引擎开发，而在卖出的游戏中，有26%由Unity开发，有31%由UE开发。且随着游戏规模的上升，从Tiny Games（卖出1k份）到Small Games（卖出1k-100k份）到Medium Games（卖出100k-1M份）到Large Games（卖出1M+份），Unity引擎开发的游戏占比为50%-48%-35%-22%，而UE引擎则是从23%-27%-32%-31%。这意味着除去部分大型游戏厂商独占的自主研发引擎，UE引擎是名副其实的最广为使用的大型畅销游戏开发引擎。

但是，UE也是一个以“重（Heavy）”著称的游戏引擎，体现在以下方面：1—开发设备配置要求高，开发成本高；2—使用C++作为开发语言，开发门槛高，开发流程繁重且复杂；3—前沿技术集成度高，理解难度高，优化空间少，开发者对引擎的掌握程度低。这导致了使用UE开发的工作室往往面临高成本低产能的

现状，但是寄希望于销量是高度不确定性的游戏市场中一个不切实际的想法，所以，降低产出游戏内容的成本是使用UE开发的游戏工作室的一个迫在眉睫的任务。

自《魔兽世界》在客户端中使用Lua语言并大获成功后，国内的《大话西游2》首先开始使用Lua语言进行游戏客户端开发，后来，腾讯公司注意到Lua语言的灵活特性并随后开发出了UnLua[2]插件以在UE开发流程中使用Lua语言替代C++的游戏逻辑开发，不需要等待重新编译，只需要重启PIE（Play In Editor）就可以确认修改的逻辑的效果极大地加快了内容开发的速度，并且腾讯公司放弃了持有UnLua并将其开源，现如今UnLua插件是所有使用UE开发的游戏工作室最为常用的插件之一。

但是，游戏厂商的产能依然跟不上玩家的消费速度，并不是由于玩家有多么能消费，而是由于玩家总是更倾向于游玩质量更好的游戏，在《2025年大型游戏引擎报告》[1]中显示，由其余的小型引擎开发的游戏数量占比达到了11%，在最终的销量占比中却仅占2%，这说明玩家宁愿等待更好内容的游戏也不愿意消费低质量内容的游戏，这一现象意味着大型游戏厂商可以继续通过加快游戏内容产出速度来挤占游戏市场。最终，UGC（User-Generated-Content）这个概念被发现并采用.事实上，现在热门的CS（Counter-Strike）和DOTA（Defense of the Ancients）都是由半条命（Half-Life）和魔兽争霸3（WarCraft Ⅲ）的模组衍生而来，而《我的世界》和《Roblox》属于对物块进行创作，《马里奥制造》和《Doom》则是进行关卡的创作[3]，由玩家创作内容供玩家消费成为了一种行之有效的解决方案。

在可以预见的未来内，由UE引擎开发的游戏必然会存在有的游戏在收获了较高的人气后没有产能来维持玩家们的热情的情况，而在市面上，由UE引擎开发的游戏目前还缺少一个明确的UGC框架方案，因此，本文尝试通过结合UE和UnLua来开发出一个可供使用的UGC插件，并通过道具这个在游戏中广泛存在的概念来进行创作开放，主要利用UnLua插件来解决UE的开发速度问题，用Lua的可变参数特性解决C++语言对可变参数的支持较差以及维护成本高的问题，用Lua的协程来解决C++委托实现的异步逻辑过于复杂的问题，希望能够为UE的UGC开发提供一个可供参考的方案。

在本文中，我们首先需要了解UE，Lua语言，UnLua插件相关的前导知识，然后，结合国内外对于Lua语言在游戏开发中的研究和国内外对于UGC形态的研究，展开对于UGC道具框架的构建过程，然后在实践中对该框架进行一定的性能分析和优劣势分析，最后展望该框架可能的发展形态和进一步能做的改进。

## 前导知识

在该部分中我们分别介绍与UGC道具框架相关的UE，Lua，UnLua知识

## UE相关知识

根据UE官方文档[4]，UE构建了一个庞大的GamePlay框架，其基本类为UObject，在UE进行GC（Garbage Collect）的时候，会选择一些默认的UObject类作为起点进行可达性检验从而回收垃圾，避免OOM（Out of Memory）导致的崩溃，因此，任何在游戏中会以实例存在的类都需要继承自UObject，否则就会发生内存泄漏，并且实际上，如果不继承UObject，实际上是没有加入到框架体系中的，并不会在运行时有效果。

C++在C++20的时候才引入了一个初步的反射机制，所以UE实现了一套属于自己的反射系统，在定义属性或者函数的时候可以通过添加UProperty()宏和UFunuction()宏来实现反射，其原理是在编译期生成属性或者函数的元数据，并生成一个相关的生成函数来按照元数据模板生成实例。

委托是UE实现异步逻辑编写的主要手段，通过事件的触发来产生回调，同样需要使用宏定义DECLARE\_DELEGATE来声明委托，使其能够被序列化和反射。完成委托逻辑的编写如下：1—使用宏定义声明委托；2—创建委托实例；3—绑定委托要触发的回调函数；4—在指定位置执行委托的Execute()或者Broadcast()。除了第一步声明以外，每增加一个异步逻辑都需要重复后面的三个步骤。

## Lua相关知识

Lua 是一种动态类型，解释执行的脚本语言，其本身完全由C语言编写而成，通过构造一个以栈为基本结构的虚拟机，将代码编译后按照Lua内部的机器码解释执行。因此Lua是一种可扩展的扩展语言，具有可移植性、嵌入的便捷性、轻量性、高效性[5]。

Lua具有许多思想十分先进的特征。

首先，Lua使用全局表\_G来管理自己，例如自带的print函数，位置上就位于\_G[print]，同理，那些基本的关键字全部都位于\_G表下，且无法修改。基于这个特性，如果我们用类似于Python的包的思想来看待Lua的文件，在一个文件的初始定义一个空表local M = {}，然后通过M.xxx的方式来添加内容，在文件末尾return M，我们就能将整个文件表示为一个表，我们可以将表合并到全局表下，也可以在别的文件require这个表使用一些静态函数功能，如果我们对获取外部表的方法进行权限约束，还能够诞生沙盒（Sandbox）的概念，沙盒只能获取权限能获取的外部变量或函数以及文件内部的变量和函数，这是一个比Package思想更加轻便简洁又更加严谨的特征。

其次，Lua具有名为闭包的特征（闭包还包含了函数对象的思想），它将函数视为和变量同级的初等公民，任何一个函数在作为一个对象传递的时候都会形成一个闭包，它将同时包含逻辑以及逻辑需要的变量环境，如图1.1所示

图1-1 Lua的闭包演示

并且，如果创建一个local counter2 = createCounter()，计数将重新从1开始，除此之外，我们还看到了C++的λ函数的影子，并且Lua的使用方式明显更加简洁，将函数作为初等公民的思想，在接下来的协程中会享受到明显的便捷。

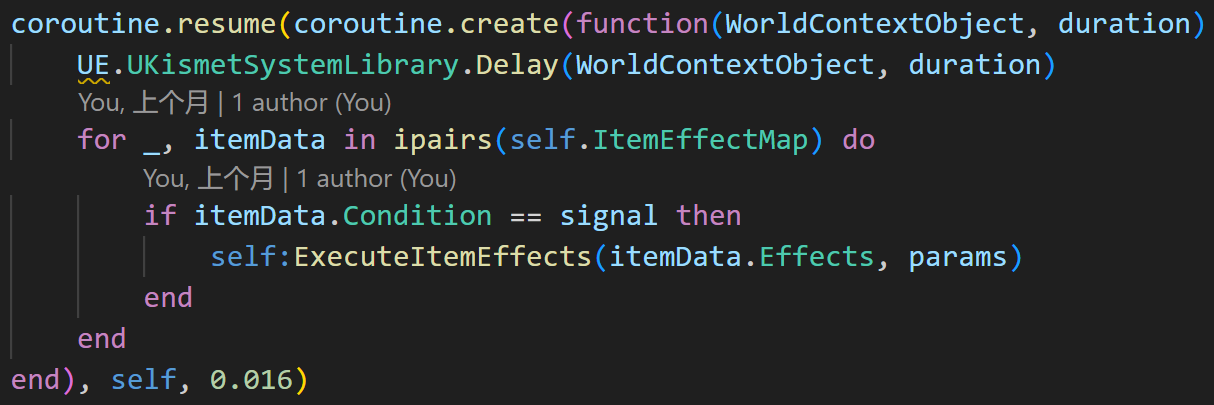
 最后，Lua有自带的协程系统，通过coroutine.create(fn)传入一个函数对象来生成一个协程co，然后通过coroutine.resume(co, ...(params))来执行协程，在实际的使用过程中，我们可能根本不需要提前写任何逻辑，直接在需要的时候一气呵成完成异步逻辑的编写，如图1-2所示。

图1-2 一次性写完的让逻辑延后0.016s执行的协程

我们能够以coroutine.resume(coroutine.create(fn), ...(params))的方式轻松地运行一个协程并完成其在任务时间线上的操纵。

## 第七章 结论

## 参考文献

[1] VG Insights. The Big Game Engines Report of 2025[EB/OL]. 2025. <https://vginsights.com/insights/article/the-big-game-engines-report-of-2025>.

[2] UnLua[EB/OL]. <https://github.com/Tencent/UnLua>

[3] Liu Y, Duan H, Cai W. User-Generated Content and Editors in Games: A Comprehensive Survey[J]. arXiv preprint arXiv:2412.13743, 2024.

[4] Gameplay Framework in Unreal engine[EB/OL]. <https://dev.epicgames.com/documentation/en-us/unreal-engine/gameplay-framework-in-unreal-engine>

[5] Ierusalimschy R, de Figueiredo L H, Celes W. The evolution of Lua[C]//Proceedings of the third ACM SIGPLAN conference on History of programming languages. 2007: 2-1-2-26.

## 致谢