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语言在低代码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，在定义类、属性、函数的时候可以通过添加UClass()宏、UProperty()宏、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。

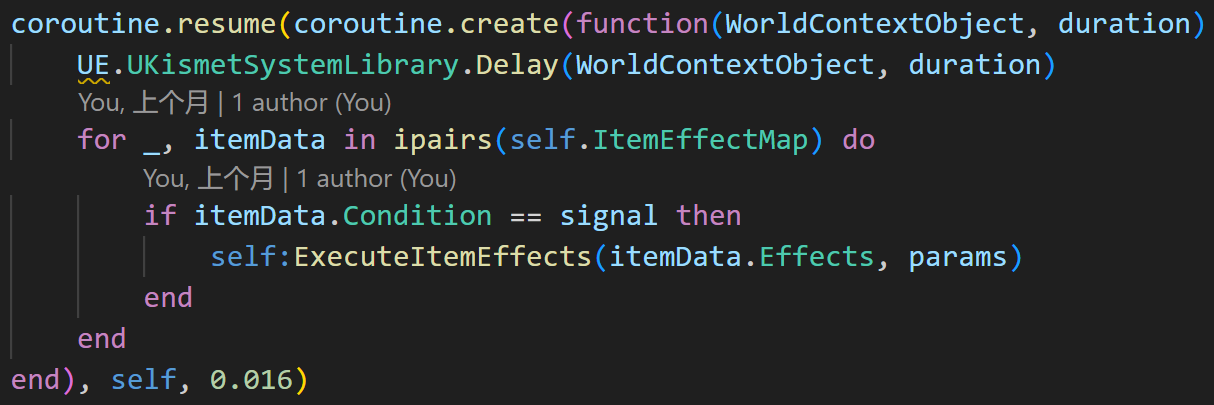
 最后，Lua有自带的协程系统，通过coroutine.create(fn)传入一个函数对象来生成一个协程co，然后通过coroutine.resume(co, ...(params))来执行协程，在实际的使用过程中，我们可能根本不需要提前写任何逻辑，直接在需要的时候一气呵成完成异步逻辑的编写，如图1-2所示。

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。

## UnLua相关知识

在知晓Lua的相关知识后，UnLua的理解实际上就顺理成章了，已知UE会对宏定义的类、属性、函数进行反射，而Lua万物都可以表示成表的形式，且能够容纳各种变量和函数（即所有的初等公民），那么我们定义一个名为“UE”的全局表，将所有能够得到反射数据的类都装入全局表下，例如图1-2，Delay函数的实际位置就位于UE.UKismetSystemLibrary.Delay，然后在C++让类继承接口IUnLuaInterface，通过截取并重写生成函数，使得最终生成的类实例带有Lua代码的逻辑。

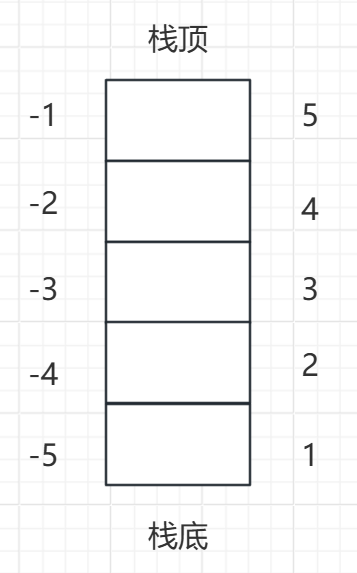
C++和Lua的互动通过Lua的虚拟机栈来实现，如图1-3所示。

图1-3 Lua虚拟机栈结构

当C++想要借助Lua实现某些功能的时候，可以通过压入函数，压入参数，执行函数，弹出结果的方式来完成，所以在实际性能损耗上大概相较于单纯使用C++要多2+n（压入和执行，加上参数和结果的数量）次中间调用，在大量使用Lua的时候，的确有可能会出现性能问题，但是，相较于会出现10次以上中间调用的UE的蓝图编程，Lua依然是更好的辅助编程选项，在小幅度使用时并不会造成明显的性能负载。

## 相关工作

国内外对于在UE上实现的UGC框架的直接研究较少，所以本文会寻找两类相关的研究文章：一类是研究Lua语言在低代码开发中的应用的文章；一类是研究各大游戏中出现的UGC框架的文章。

## 2.1 Lua语言在游戏开发中的应用

在探索面向低编程能力的群体的Lua编程中，Msiska M F和van Zijl L[6]对于Lua的可视化编程给出了一种环境和编译器的实现，用于面向自闭症谱系障碍儿童（children with ASDs）的治疗，其界面的实现类似于Scratch[7]，已经编写好的代码块以积木的形象出现在操作台上，而使用者只需要拖动并连接积木块，使积木块上的说明文字能够形成一定的逻辑形式，实际上就会自动转换成合法的Lua代码，对于一些空置的变量位置会设计默认值填充来避免编译失败的问题。该方案对于专业能力较少或者缺失的人群是十分友好的，但是，图形化的界面意味着逻辑的展现是不直观的，对于层次结构较为复杂的逻辑阅读性也是会很差的，由于在转换成语言前积木仅仅是一个前端表示，维护时进行的查找工作也很难进行。Wood C, Mentzelopoulos M, Protopsaltis A[8]利用Unity3D开发了一款游戏EdCCDroid，游戏利用 H.U.D.来为用户提供关于游戏的信息，并允许用户在游戏内编程。提供了类似moveForward()，turnRight()/turnLeft()，open()/close()，findTargets()/aimAtTarget等接口供玩家调用。还提供了for，if等逻辑的教学关卡来教玩家学会编程。由前导知识Lua是一门解释执行语言，是可以像Python那样在交互式解释器中实时执行代码行的，也是该方案能够实现的核心原因。结合上述方案，我们发现在低代码开发的方案中，虽然前端的呈现方式各有不同，但是一个不变的核心观念是官方的开发人员都需要开发足够完备的预设，比如代表指定代码块的积木和有既定效果的代码接口，还需要实现一定的逻辑结构使其能够完成更加复杂的逻辑。

## 2.2 各大游戏中出现的UGC框架

提到UGC就不可能绕过《我的世界》（《Minecraft》）这款伟大的UGC游戏，首先，其基础玩法就是基于物块的UGC，在 Dezuanni M, O’Mara J, Beavis C[9]的研究中，儿童们沉浸于在游戏中生存，收集资源（方块），然后“表演性地构建自我”，可见基于物块的UGC框架是朴素且易懂的，即便是8-9岁的孩子也能利用这样的编辑结构来进行自我表达。实际上，《我的世界》中还有名为红石系统的进阶UGC系统，但是孩子们在面对红石的时候仍然挣扎，但能够模糊地感知红石“有点像《我的世界》里的电”。事实上，红石系统给出的是能源，开关，和逻辑门这几样基本的东西，借此红石就能够像电路一样实现一些基础逻辑。另一款在UGC方面大受成功的是《Roblox》，Brown L, Friesen A, Jeffrey A[10]给出了《Roblox》的编辑器Roblox Studio的研究，Roblox Studio给出的是一个轻量化的游戏引擎编辑器，提供了三维的检视界面方便创作者进行三维资产的位置调整，并且使用衍生自Lua的语言Luau作为该轻量编辑器的脚本语言，然后通过对脚本进行缺陷检测、类型检测、类型推断等来辅助创作者进行开发。相比于直接基于物块的《我的世界》，《Roblox》的创作方式显然更加专业化，创作者需要至少对编程有所了解，并且对游戏系统中广泛存在的观察者模式编程有初始的概念，即便存在类型检查和逻辑检查来辅助，创作者排查问题的效率依然会偏慢，而且在Lua基础上再衍生语言的话会导致知识范围极化和狭窄化，使用AI（Artificial Intelligence）进行辅助编程的出错率也会提升。好在《Roblox》广泛的知名度依然还是吸引了一些知名的游戏模组创作者参与创作，脚本编程由于上限更高会产生更多有趣的玩法，从而间接鼓励了普通玩家学习游戏脚本编程。

## 框架设计与实现

本章主要介绍UGC道具框架（UGCItemFramework）的整体设计与实现。该框架通过Lua语言的灵活性和弱类型特性来解决C++实现UGC框架过于死板，维护成本过高的问题。实现轻量化的信号管理器来避免使用C++的委托编程，并且设计了标签和编程两个层面的UGC层供业余和专业两类创作者人群进行创作，提供更加缓和的学习曲线和保证UGC的上限。并通过异步逻辑和协程来保障框架的并发性能和避免循环逻辑导致的卡死。接下来本章将先整体概述，然后详细讲解框架细节和实现细节。

## 3.1 框架整体设计

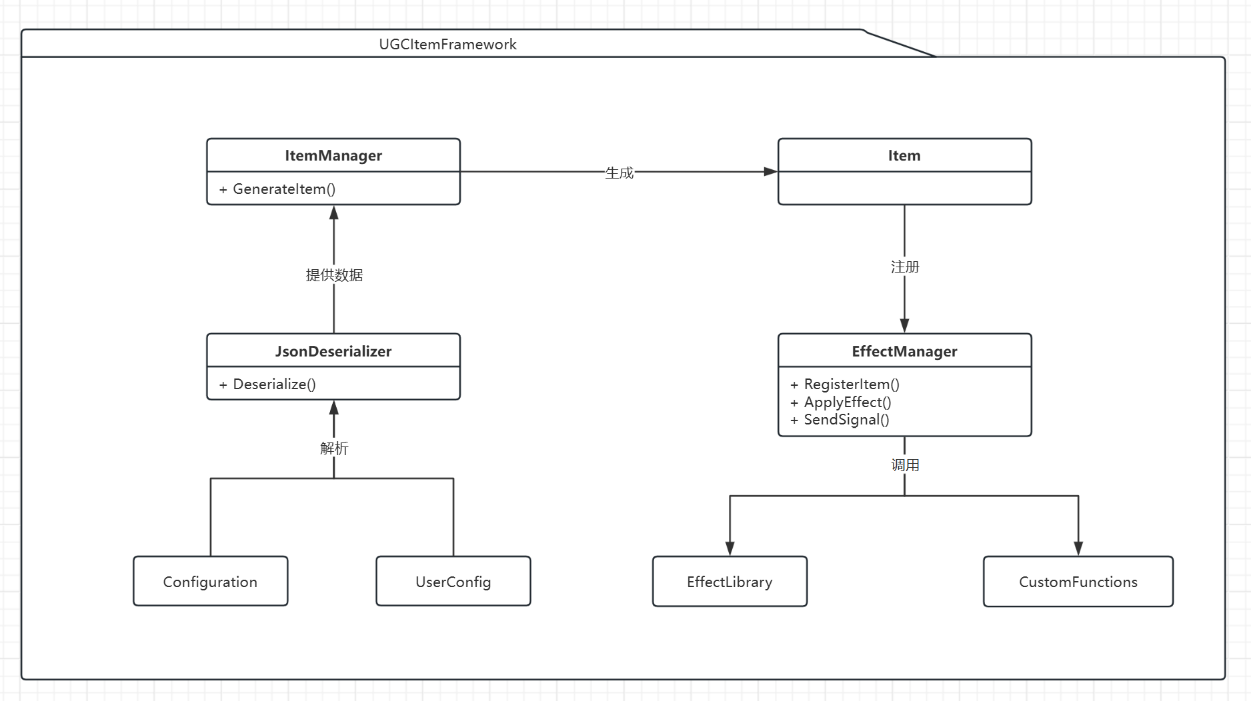
 基于相关工作和前导知识，我们能发现一个面向非专业人士的UGC框架需要以下必要的要素：简化的创作方式，易入门的逻辑结构，丰富的官方接口。而面向专业人士的UGC框架需要的是对自定义代码的支持。整体的框架如图3-1所示。

图3-1 UGCItemFramework整体框架简图

本文决定对基础的创作方式采取标签化的形式，我们使用json格式来描述每个道具，并在游戏开始的初期读取配置文件并保存在道具池数组中，在生成道具的时候我们生成一个基础道具实例然后对UGC相关属性进行填充。对于逻辑结构，本文决定采取条件-效果的观察者模式，然后用条件链式触发节点来实现流程上的连续，每个触发的节点单独编写代码逻辑。前两步足够满足一些游戏模式的更改以及角色属性上的变更和游戏效果的触发，最后一步允许专业创作者进一步的复杂逻辑实现。对于专业创作者，我们创建名为module的文件夹，放置User\_Config.json和Custom\_Function.lua供初期的配置读取流程加载，这样，用户的创作和官方的创作本质上就是同级互通的，可被纳入同一个道具结构体中。

## 3.2 Item设计

在本文的设想中，Item本身是一种条件-效果的载体，其余的信息仅用于标识该道具，但是，在局内应该是一个可以被拾取的无实体但可见的物品，为了响应与角色的接触事件，所以Item仍然需要继承Actor类（Actor类是UE对于需要出现在游戏内的任何物体所设计的基类，为UObject类的子类）

## 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.

[6] Msiska M F, van Zijl L. From visual scripting to Lua[C]//Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference. 2012: 94-99.

[7] Maloney J, Resnick M, Rusk N, et al. The scratch programming language and environment[J]. ACM Transactions on Computing Education (TOCE), 2010, 10(4): 1-15.

[8] Wood C, Mentzelopoulos M, Protopsaltis A. EdCCDroid: An education pilot prototype for introducing code-combat using LUA[C]//Workshop Proceedings of the 11th International Conference on Intelligent Environments. IOS Press, 2015: 353-360.

[9] Dezuanni M, O’Mara J, Beavis C. ‘Redstone is like electricity’: Children’s performative representations in and around Minecraft[J]. E-learning and Digital Media, 2015, 12(2): 147-163.

[10] Brown L, Friesen A, Jeffrey A. Position Paper: Goals of the Luau Type System[J]. arXiv preprint arXiv:2109.11397, 2021.

## 致谢