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

**第五章 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。

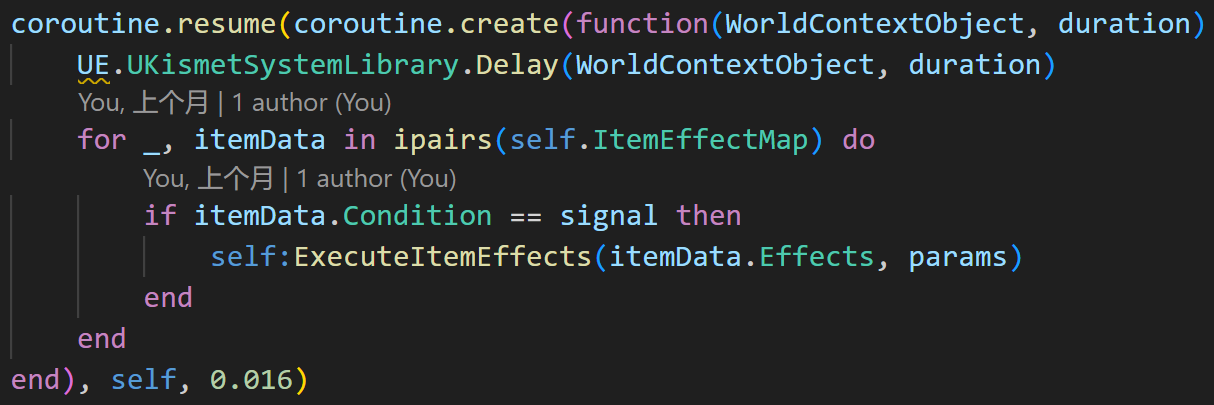
 最后，Lua有自带的协程系统，通过coroutine.create(fn)传入一个函数对象来生成一个协程co，然后通过coroutine.resume(co, ...(params))来执行协程，在实际的使用过程中，我们可能根本不需要提前写任何逻辑，直接在需要的时候一气呵成完成异步逻辑的编写，如图1-2所示。

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。

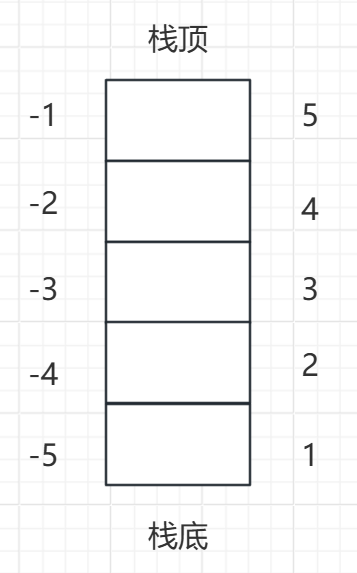
C++和Lua的互动通过Lua的虚拟机栈来实现，如图1-3所示。

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2，加上参数和结果的数量n）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，在小幅度使用时并不会造成明显的性能负载，因此我们要控制框架的规模尽量精简。

## 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

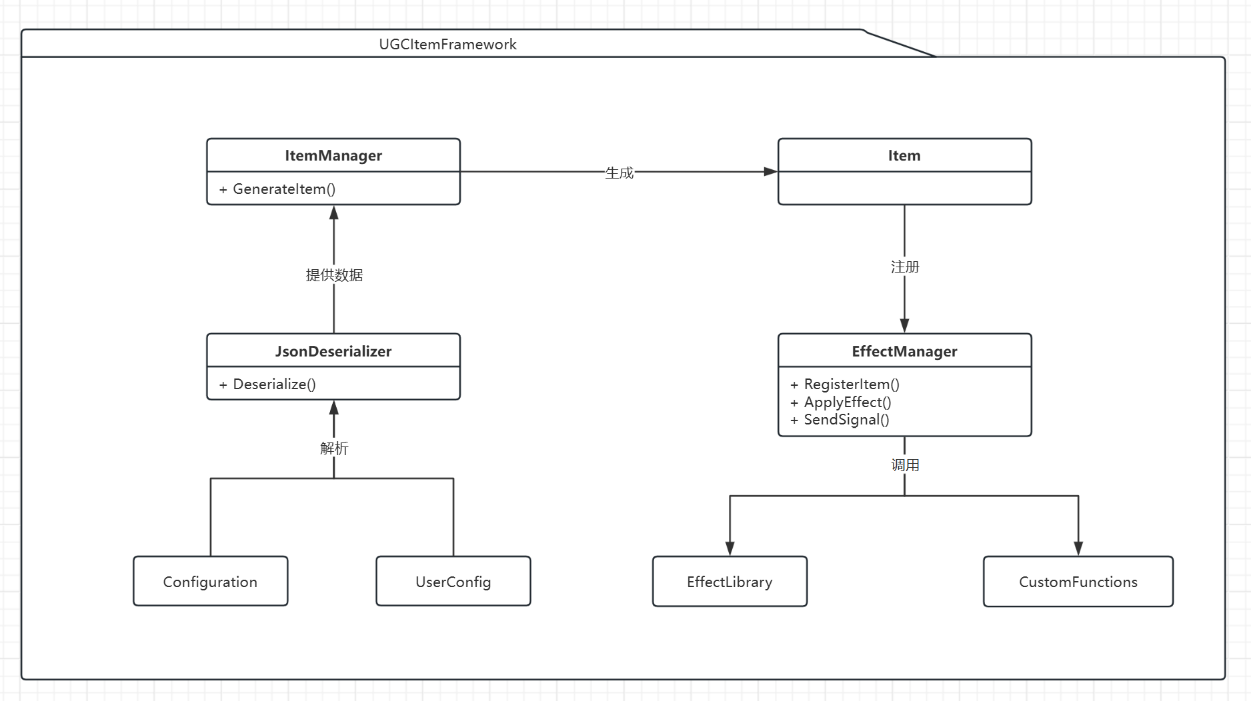
 基于相关工作和前导知识，我们能发现一个面向非专业人士的UGC框架需要以下必要的要素：简化的创作方式，易入门的逻辑结构，丰富的官方接口。而面向专业人士的UGC框架需要的是对自定义代码的支持。整体的框架如图3-1所示。

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相关设计

下面是Item相关的设计，涉及Item和ItemManager的具体设计。

## 3.2.1 Item设计

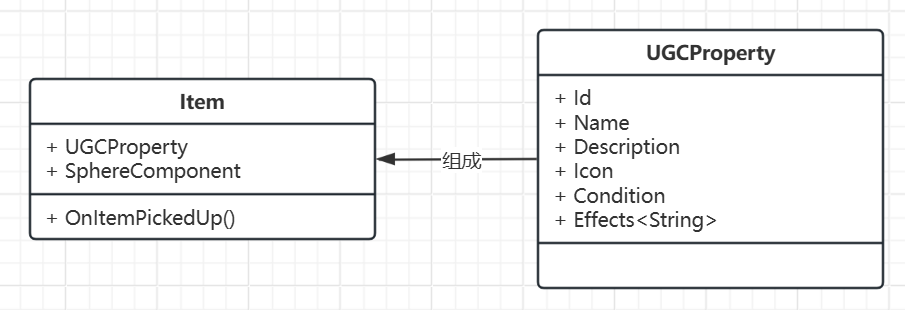
 在本文的设想中，Item本身是一种条件-效果的载体，其余的信息仅用于标识该道具，但是，在局内应该是一个可以被拾取的无实体但可见的物品，为了响应与角色的接触事件，所以Item仍然需要继承Actor类（Actor类是UE对于需要出现在游戏内的任何物体所设计的基类，为UObject类的子类），为了同时满足对游戏内“拾取”操作的适配以及对UGCItemFramework的适应，Item需要有一个SphereComponent来进行碰撞检测，需要UGC的相关属性，为了使该道具能够被辨识且能够满足条件-效果的设计，我们设置一个UProperty类的结构体UGCProperty，包含Id，Name，Description，Icon，Effects，Condition属性，为了能够简洁且统一地处理数据，我们全部设计为String属性，其中Effects由于有多个效果的需要来满足更复杂的逻辑，我们用UE引擎的TArray<String>（本质上是可被反射和序列化的Vector数组）来承载，如图3-2所示。

图3-2 Item的具体类设计

## 3.2.2 ItemManager设计

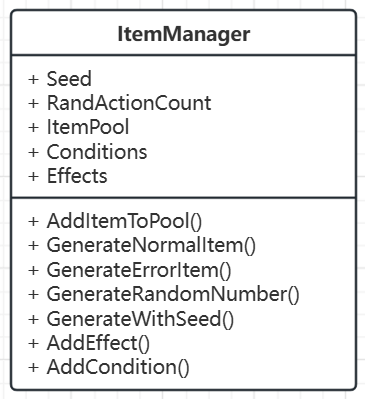
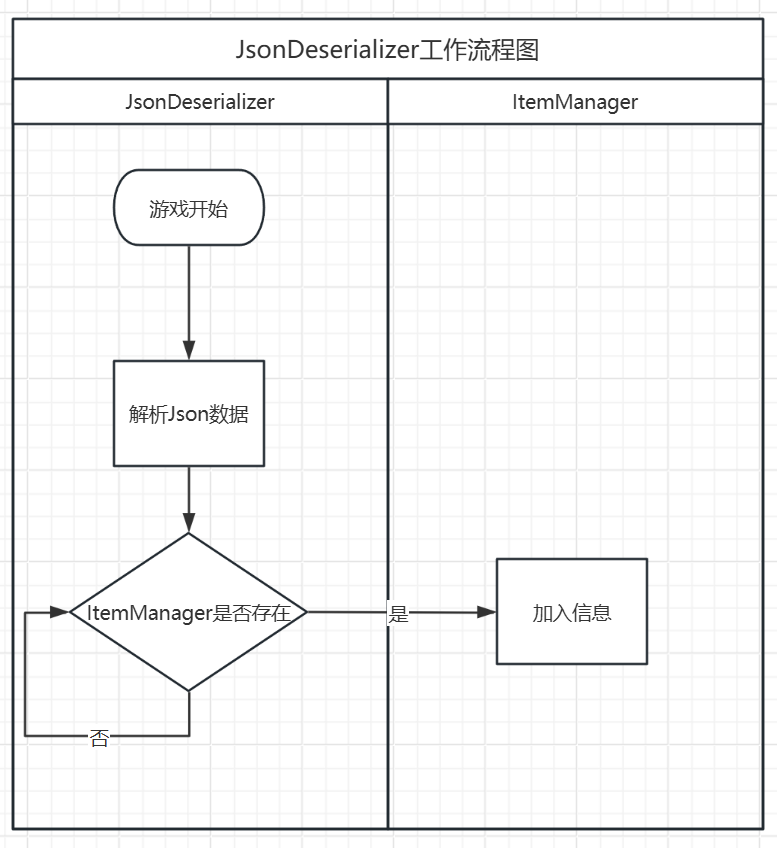
在完成Item的设计后，我们就需要ItemManager来对Item实现管理，在实际的游玩设想中，我们希望使用一局游戏的种子Seed和需要随机数的行为RandActionCount来实现两个效果：1—对于每个种子，应该尽可能地接近真随机，且对于每个尺度的道具池都有较为平均的击中概率；2—对于同一个种子，我们使用相同的行动顺序能够得到相同的道具，使用不同的行动顺序会获得不同的道具。本文决定采取XorShift算法[11]来操控随机数，XorShift算法可以实现对模232遍历232-1个余数，并且由于232足够大，对于道具这个一般最多来到四位数的概念来讲都足够大，造成的概率波动也足够小（1000/232≈0.00000023283），因此，我们每次生成道具时都对Seed做一次XorShift算法的操作来作为RandNumber，对于生成道具，本文设想用生成配置原有的道具来模拟游戏的正常道具生成，用生成一个随机的新道具来模拟创作者创作的新道具或者官方提供的随机道具模式，所以，对于生成正常的道具，我们采用ItemPool[GenerateWithSeed() % ItemPool.Num()]来获取该生成的道具，对于生成随机的道具，我们采用(RandNumber ^ 2 + RandActionCount \* RandNumber) % Conditions.Num()和(RandNumber ^ 2 + RandActionCount \* RandNumber) % Effects.Num()来获取条件和效果组成新的随机道具。最终的类图如3-3.

图3-3 ItemManager具体类设计

生成道具的流程如下，收到来自角色的生成道具信号（携带Location信息）—生成道具模板的实例—生成UGCProperty（来自ItemPool或者随机生成）--填充实例的属性。显然为了防止角色捡到还未被填充数据的道具实例，该逻辑必须是同步逻辑，而其余的逻辑也以计算为主，所以我们不需要将其加入到UnLua中进行更加复杂的逻辑编写，放在C++中会更加高效。

## 3.2.3 JsonDeserializer设计

在3.1我们说过，ItemManager所需要的Condition和Effect原材料来自于两个分别来自官方和用户的Json配置文件，所以JsonDeserializer的职责就是承担Json格式翻译和传递解析后的信息到ItemManager。

 Json的格式翻译由UnLua插件携带的rapidjson插件进行解码，我们在JsonDeserializer的ReceiveBeginPlay()中进行数据的解码，解码后用临时的表缓存得到的Condition和Effect，因为ItemManager也在游戏BeginPlay的信号后开始生成，所以在此时ItemManager还未生成，我们需要在完成解析后使用自旋异步逻辑检查ItemManager的生成，在检测到后调用ItemManager的AddEffect()和AddCondition()完成信息传递，如图3-4.

3-4 JsonDeserializer流程图

## 3.2.4 Configuration设计

Configuration的整体编写遵从Json的格式逻辑以及全部参数使用String的约定，但是对于Effect我们需要特别的约定，显然使用一个固定的不能根据参数改变效果的函数是缺乏自由度的，因此我们可以将参数加入到Effect中，我们以最为简洁的形式约定为“函数名-参数名:参数值-…”来传递函数名和参数，例如"Test-origin:1"。

## 3.2 信号系统相关设计

本文设想中，只需要一个轻量化的信号系统就可以完成信号的收发，我们需要在信号的触发中保证两个点：1—一个信号在同一帧内并发触发所有的道具；2—每个道具的效果按照顺序独立生效。在道具效果的生效中，我们要实现两个点：1—我们通过公共的函数接口库实现效果；2—函数可以接受可变参数以实现最低成本的维护成本。具体的设计如下文。

## 3.2.1 EffectManager设计

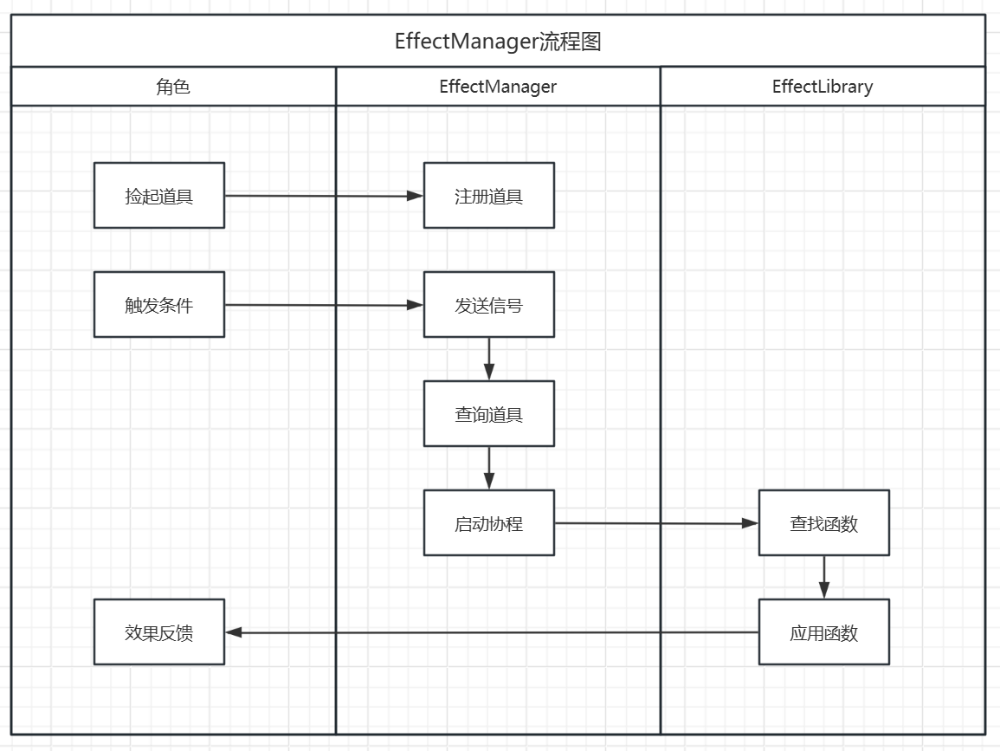
 我们设计EffectManager来管理信号，EffectManager的底层数据结构应该是一个Table，ItemEffectMap，里面的每一个Table代表每个道具，其中每个道具包含Id，Condition和一个Table[Effect]。在角色捡起一个道具时就会触发道具的注册，加入到ItemEffectMap中，我们在每个可以被当作一个信号的地方使用SendSignal(signal, ...)来查询使用该signal为condition的道具，其中...是可变参数，我们可以在后续的维护中加入可变参数维护而不需要改变函数的形式，在完成查询后，我们对每个道具单独启动一个协程来产生效果，由于每个协程是独立执行的，所以每个道具的效果也会单独按照顺序执行，不会存在以下现象：两个同类道具A和B，当A完成效果1后，下一步触发的却是道具B的效果1而不是道具A的效果2，如果效果1会对效果2产生影响的话就会导致编写道具的人的所想和实际效果的不一致。道具的触发一方面接收来自Effect中携带的参数，另一方面接收SendSignal函数中的可变参数，然后在函数库中查询函数，并根据具体有哪些参数来执行逻辑（Lua语言在使用到未定义变量时会返回nil而不是导致未定义行为），比如用于演示的大多效果作用于角色身上，所以我们在最后一步参数的合并中默认再加入角色的实例作为参数。整体的流程如图3-5.

图3-5 EffectManager流程图

## 3.2.2 EffectLibrary及函数设计

在EffectManager中，最后的每个道具效果协程都需要查询EffectLibrary，在C++中，这会是一个非常复杂的反射流程的应用，我们需要查找到该函数并且设置好参数的内存分配，对于每次新加入的参数类型都需要有新的内存分配检测，否则就会导致失败或者内存泄漏。而在本文中，我们就可以借助Lua的可拓展性和自带的GC系统摆脱掉这个非常痛苦的维护成本。我们设计EffectLibrary作为可供调用的静态函数库，借助Lua万物都是表的特性，我们按照EffectLibrary[FunctionName] = Function的形式来编写函数逻辑，当外界查询时，直接使用local EffectLibrary = require("EffectLibrary\_C")来将该函数库作为一个表获取，然后对这个表遍历查找即可。用户的自编写函数也可以以一个非常简单的方式融入，编写一个函数名-函数的表，然后return即可，在EffectLibrary的初始化阶段，我们就可以使用合并表的方式最后得到一个经过用户函数扩展过的函数库。

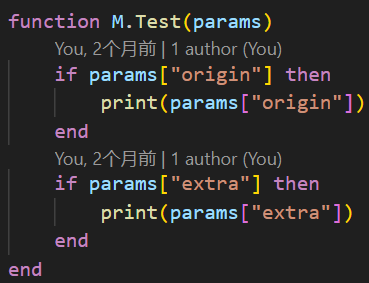
 函数的设计更体现Lua的灵活性，已知参数会来自三个方面：1—道具自带的参数；2—发送信号时带入的参数；3—最终合并参数时带入的默认参数（全局环境）。所以函数的逻辑可能为了短时间的一些Feature进行变化，我们希望最终的维护成本仅仅是添加或者删除一个if，因此，我们可以先对参数的值进行检验，如果是nil，则不执行效果，如果非空，则执行对应的参数的效果。如图3-6。

图3-6 函数逻辑示例

## 3.3 框架实现相关

以上是整个框架各个部分的设计设想，但是在实际开发时，由于UE的“重”和两个框架的连接，在实现时有相当多的细节需要注意，以下是需要注意的实现细节。

## 3.3.1 UE连接UnLua

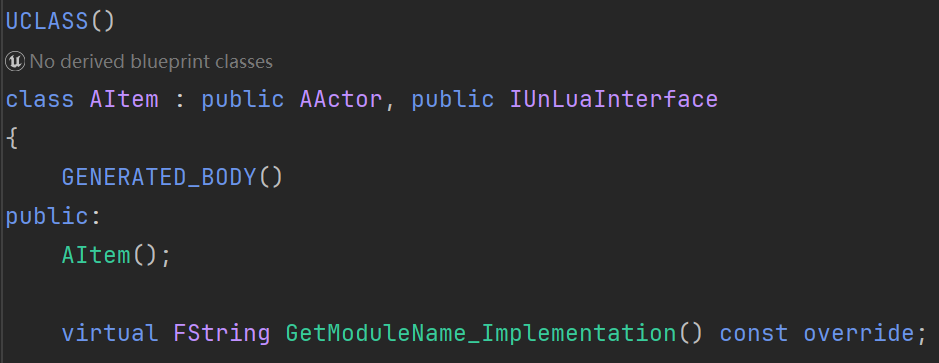
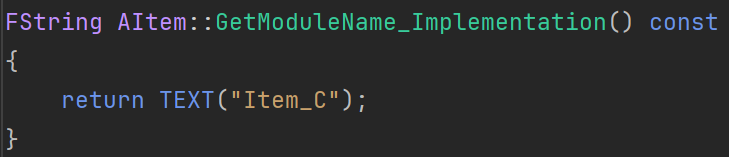
 UE连接UnLua的方法是在声明类的时候加入UClass()宏，并继承IUnLuaInterface接口，然后添加GetModuleName\_Inplementation()虚函数来返回Lua文件名，UnLua框架会根据返回值寻找${项目名}/Script/{文件名}处的Lua文件并建立联系，如图3-7，3-8

图3-8 返回Lua文件名

图3-7 连接UnLua的类的编写

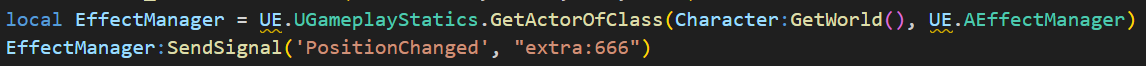
 此时，我们可以在${项目名}/Script/{文件名}的文件中重写逻辑，重写规则如下：相同名字的函数—由Lua文件的函数覆盖C++文件的函数，如果想调用原来的C++函数，可以在Lua文件中使用self.Overridden.{FunctionName}进行调用；不同名字的函数—以Lua文件中的为准。如果Lua中想要获取游戏中的实例，可以使用UE.UGameplayStatic.GetActorOfClass(GetWorld(), ClassName)来获取该World下的实例，如果想要调用实例的方法，就使用实例:函数名来调用，如图3-9。

图3-9 Lua中实例的获取与使用

## 3.3.2 初始化顺序

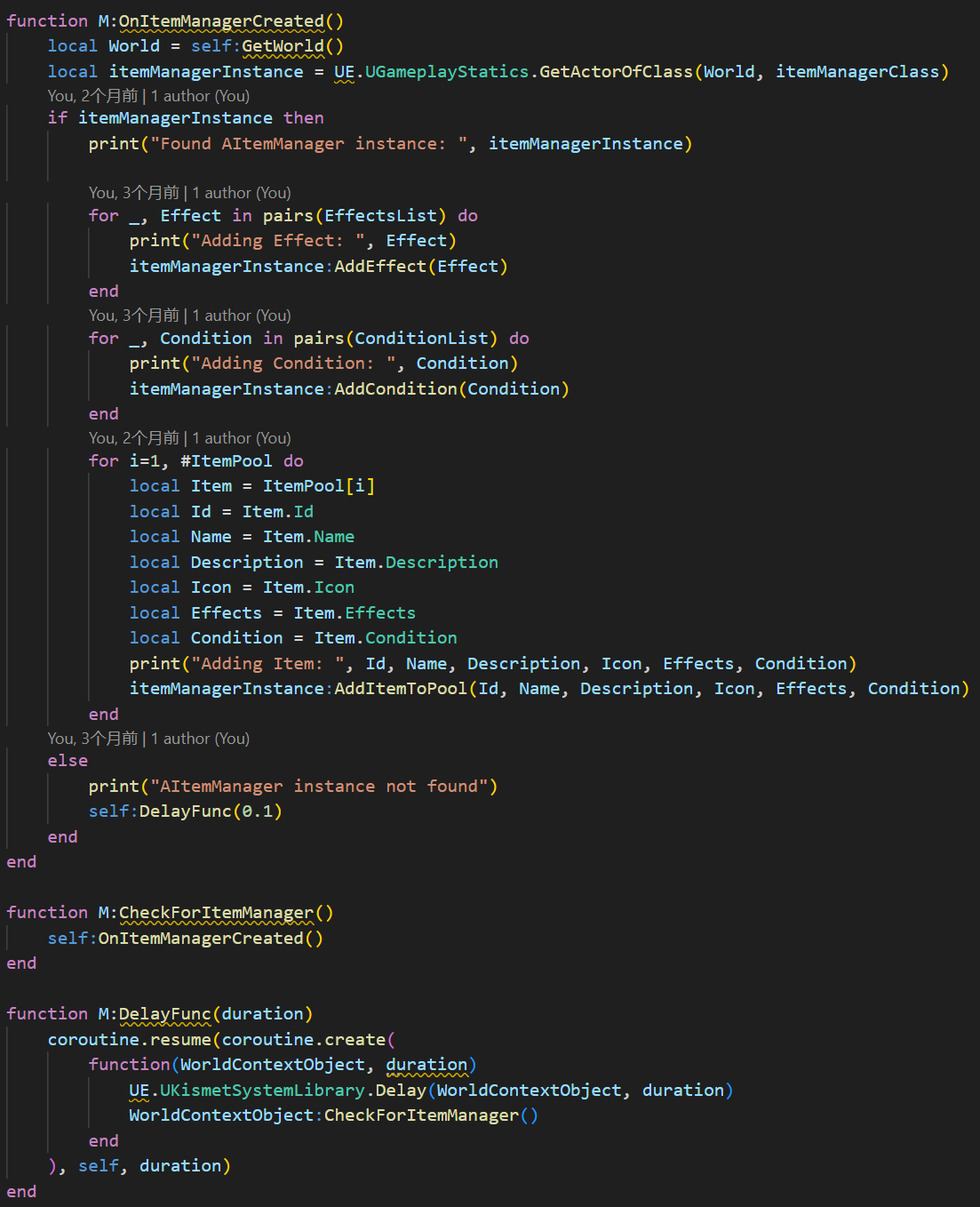
 在框架的初期，各个模块的初始化顺序是一个很重要的问题，模块之前的依赖顺序和初始化顺序应该是倒置的，才能满足正确的初始化效果，这要求我们需要在时间轴上的排轴，最容易理解的方法就是设定自旋来进行检测上一层的模块是否已经存在，那么我们首先需要的就是一个延迟函数，在UE.UKismetSystemLibrary中存在现成的Delay函数，我们使用协程包含这个函数以使得其延迟实际对Lua的执行生效。以JsonDeserializer和ItemManager的初始化举例，JsonDeserializer的信息需要ItemManager初始化完成后再传递，因此，我们在完成解析后启动函数CheckForItemManager()，CheckForItemManager()函数触发检查逻辑，如果不存在ItemManager，则触发Delay函数，创建一个协程在0.1秒后再触发CheckForItemManager()函数,如果存在ItemManager，则完成信息传递并不再重复触发逻辑，实际实现如图3-10.

图3-10 ItemManager的自旋检查逻辑

## 3.3.3 角色和信号的发送与触发

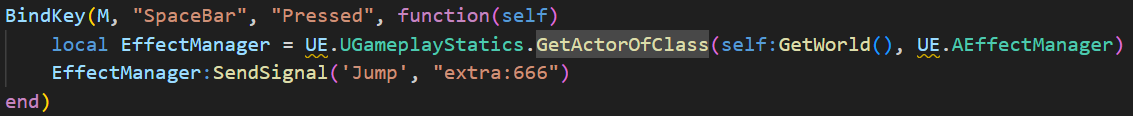
 为了直观体现UGCItemFramework，我们在demo中添加角色作为游玩的对象，该角色负责触发各种各样的信号和接收函数的效果改变，因此我们也将角色加入UnLua中，UnLua对绑键已经有了封装过的接口BindKey({Object}, ButtonName, Action, function)，我们可以使用该接口轻松地完成按键委托的绑定，如图3-11，我们绑定空格键的委托是发送Jump的信号，并且还可以传递额外参数。

图3-11 轻松的绑键操作

但是实际上，发送信号的对象不一定是角色，而如果效果是反馈在角色上的时候，该参数是缺失的，所以，对于常见的作用对象，我们应该在参数的合并阶段加入该对象作为默认参数，如果新增了一系列针对某个其他类的函数，那么在维护的时候加上该类的实例作为参数即可。

并且，当一个效果执行的时候，也有可能会发出信号，如果该信号是另外的道具的触发条件，而另外的道具产生的效果发送的信号又是原来道具的触发条件，就会触发循环，如果信号的发送是一个同步操作的话，就会导致游戏卡在循环的那一帧无法完成计算，所以信号的发送必须是异步且延后的，如果取游戏的常见帧数60帧的话，那么该延迟需要是0.016秒，我们可以粗略地认为当触发循环逻辑时每秒的协程并发数=循环道具数量\*60。

当道具被触发时，如果不采用协程的话，道具内的几个效果是会在游戏的同一帧内被触发的，例如一个道具的实际效果是效果A—延迟1秒—效果B，在协程外就会在一帧内同时实现三个效果，那么这个延迟1秒就失去效果了，所以我们需要使用协程包裹每个道具的触发，每个协程是独立且顺序执行的，就能够满足延迟一秒的效果。

综上所述，当调用SengSignal函数的时候，我们需要启动一个大的协程，该协程自带0.016秒的延迟，在延迟之后查找ItemEffectMap中Condition和Signal对应的道具，并对每个道具启动协程（并传递SengSignal中的可变参数），协程逻辑为顺序触发道具的效果（并传递可变参数），在触发效果的时候合并Effect自带的参数和可变参数和默认参数，并在道具的效果函数中根据改变的事物继续发送信号。整体如图3-12所示。

## 3.4 调试相关实现

图3-12 信号发送和触发的核心逻辑部分

虽然UE是一个完全透明的开源框架，但是现代游戏引擎的代码量早已超过人能够理解的范畴，所以其在运行时其实与黑盒无异，为了能够将调试信息呈现在游戏运行时视觉所能到达的地方，实际上也需要一定的开发工程量，此处也需要介绍相关实现以作为参考。

3.4.1 在屏幕显示道具信息功能的实现

## 为方便在局内显示所指道具的信息，我们需要做一个在局内的屏幕上显示道具信息的功能。在一个标准的第三人称视角游戏中，游戏的实际画面为角色的组件Follow Camera拍摄所得，所以我们获取Follow Camera的位置和向前的向量作射线检测，对第一个检测到的目标进行类判断，如果是框架中的道具，则向视口左上角添加文本显示道具的相关信息，以上的逻辑我们放在Tick逻辑中以在每帧进行判断，这样就能做到实时检测面前的道具并将信息呈现在屏幕上，以便测试实际的效果是否和描述一致，由于是一个附加功能，我们选择UE自带的蓝图编程功能来最快实现。

第一部分是数据的获取和射线检测，我们在角色的蓝图中可以直接获取Follow Camera的世界位置和前向单位向量，然后我们使用Line Trace By Channel（由通道检测线条）结点，以相机位置为起点，相机位置加前向向量乘以10000作为终点进行检测，由于该逻辑对实时性有高要求，所以我们将该逻辑放到Tick事件下，如图3-13。

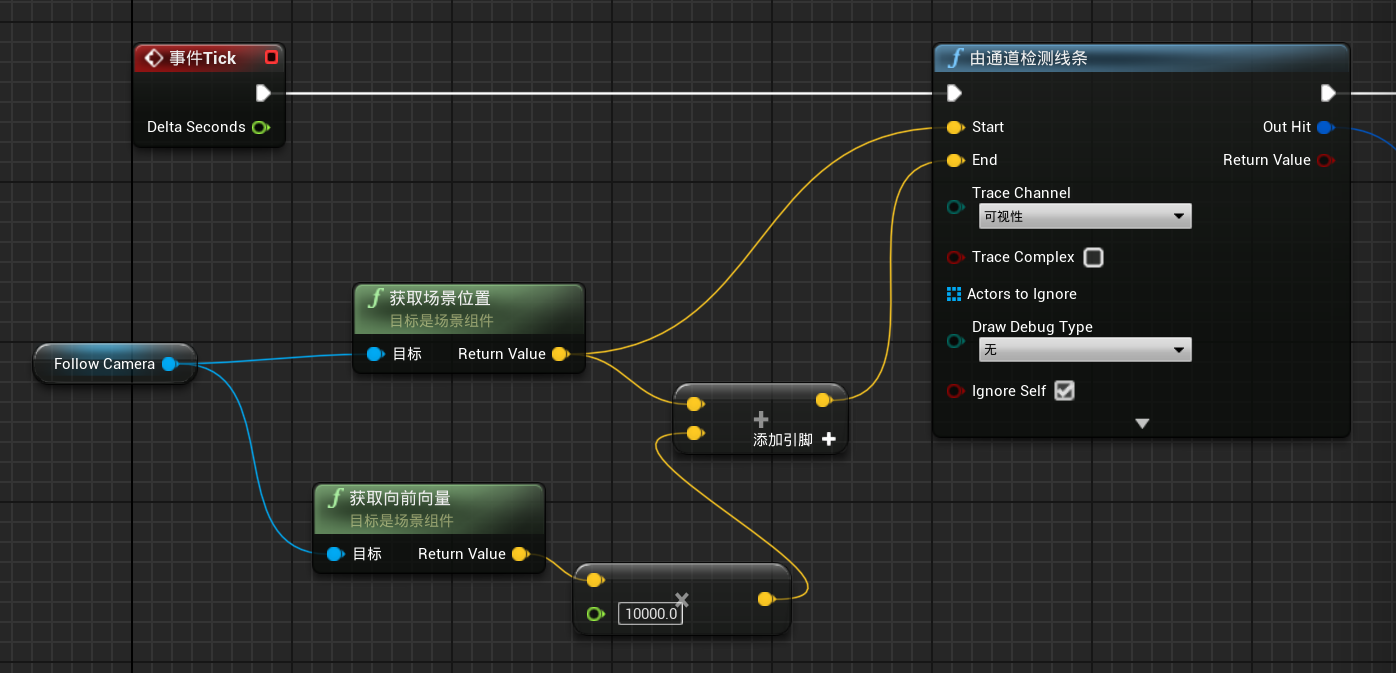
第二部分是类型判断和类型转换，我们使用中断命中结果结点来获取射线命中的第一个Hit Actor，并进行类型判断是否为框架中的道具类Item，如果是则类型转换Hit Actor为Item以便进行下一步的数据获取，具体如图3-14。

图3-13 道具检测part1 射线检测的数据获取

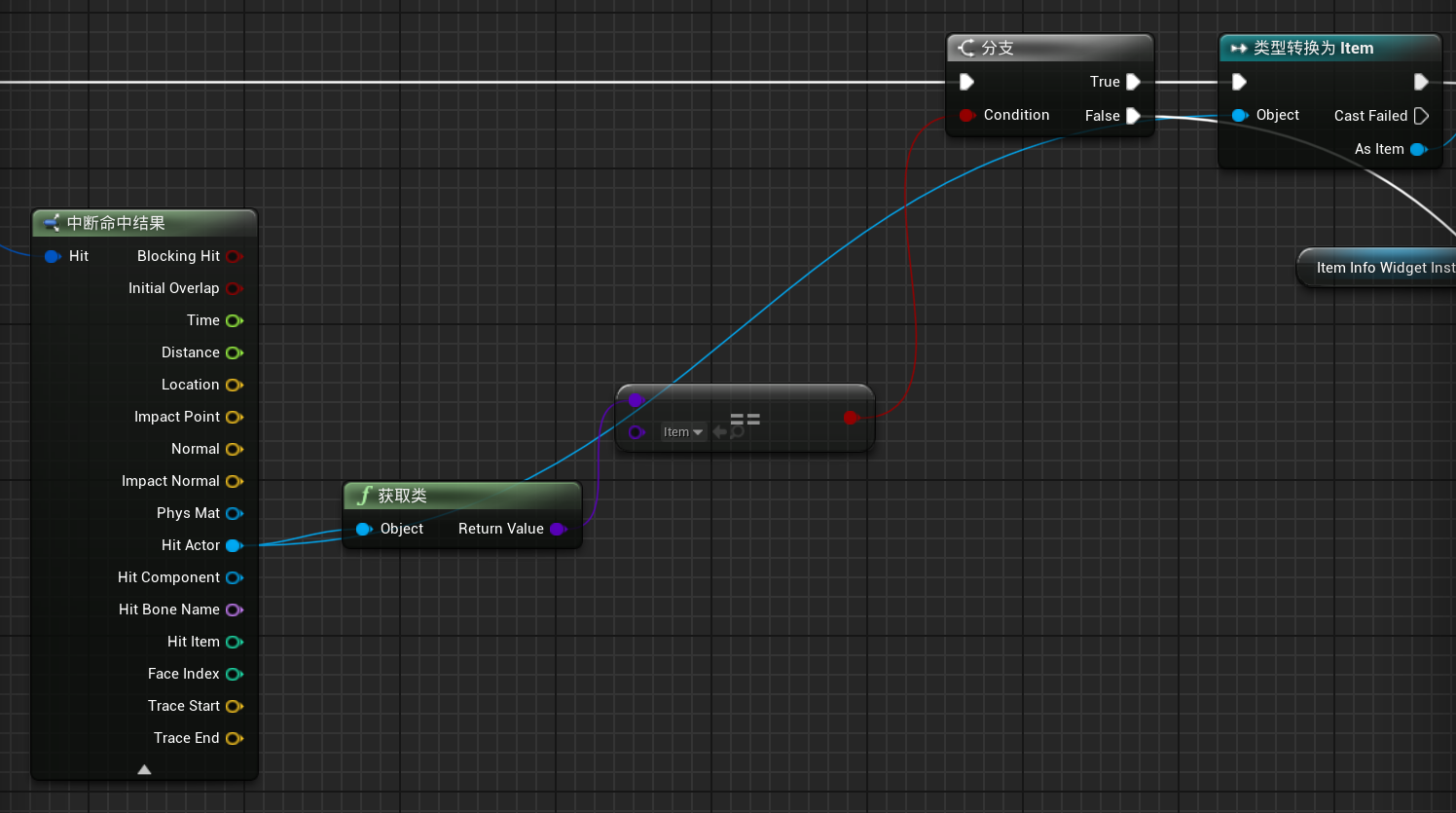
第三部分是获取数据并显示，首先我们事先在游戏开始时向视口添加了Item Info Displayer（就是一个简单的多栏文本框，不作赘述）并设置为隐藏，然后我们获得转换为Item后的UGCProperty并选择Name，Description，Effects，Condition四项属性设置为Item Info Displayer的文本并设置为显示，如果第二部分的类别检测就为false则重新设置为隐藏，具体如图3-15。

图3-14 道具检测part2 Hit Actor的判断和类型转换

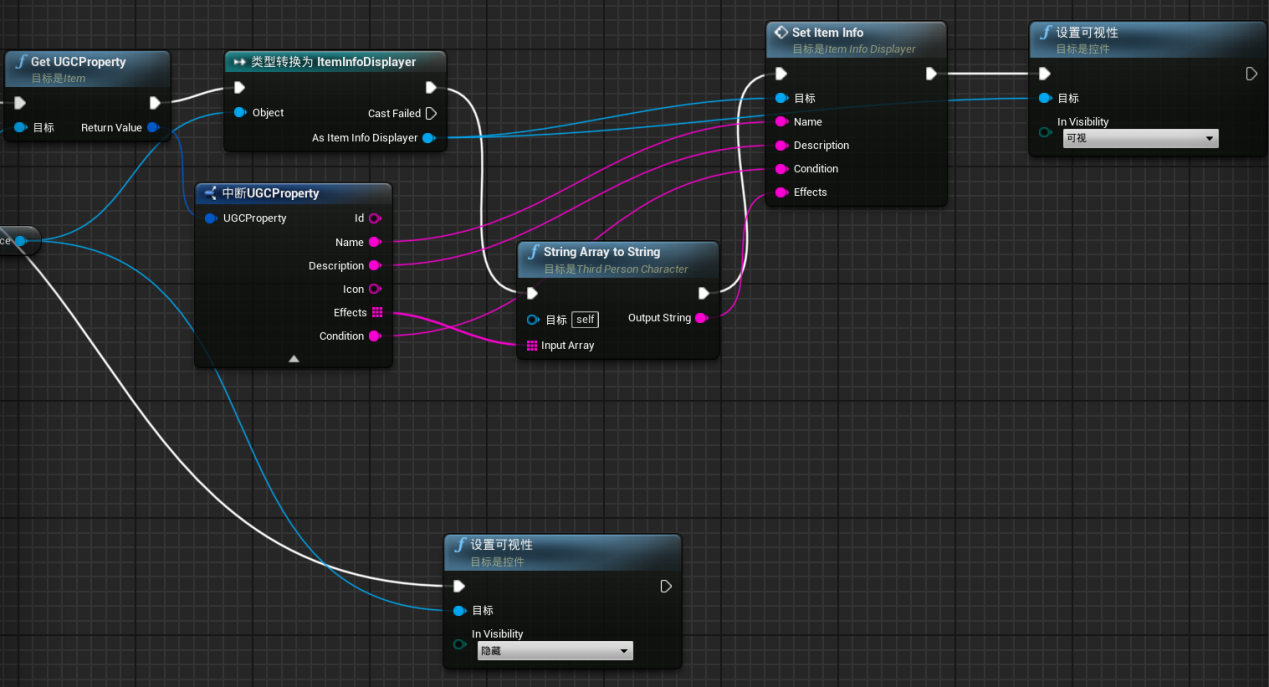
如此一来，我们就实现了道具的实时检测和属性显示，局内可以根据显示的文本和实际的效果对比以进行调试。

图3-14 道具检测part3 道具的属性的显示

## 实验评估

本章主要使用UE自带的性能测试指令Stat Unit对该框架进行性能评测，检测高并发状态下框架对游戏造成的性能损耗。

## 4.1 实验平台

操作系统：Windows 10

CPU处理器：AMD Ryzen™ 5 7500F

内存：32768MB

相关环境：UE版本：4.26，Lua版本：5.4.3，UnLua版本：Commits on Nov 29, 2023

## 4.2 实验方法

使用Stat Unit启动对游戏的监测，由于常态情况下无法完成协程并发上限的测试，本文设计了几个可循环触发的道具来快速触发，由于信号的发送延迟为0.016秒，可以认为循环道具数量\*60即为每秒协程并发上限，我们检测不同组合道具循环触发时对Game线程的影响来判断性能影响。

## 4.3 实验结果

|  |  |
| --- | --- |
| 道具组合 | Game线程平均耗时（ms） |
| 无 | 1.82 |
| 旋转+位置回溯 | 1.96 |
| 位移+旋转 | 2.15 |
| 位移+位移 | 2.07 |
| 旋转+旋转 | 1.94 |
| 位移+位置回溯 | 2.17 |

首先可以确定的是随着实验产生结果而不是直接卡死，说明框架对于循环触发导致的高并发线程情况是有一定的鲁棒性的，这意味着在正常游玩而非刻意毁坏游戏的前提下，该框架至少可以确保120次/秒的道具触发频率下可以稳定运行而不会导致崩溃的，这远远覆盖了实际的游玩场合角色所拥有的道具数量。其次是性能表现上，可以看到120个每秒的并发线程对Game线程造成的性能损耗为从0.12ms到0.35ms，但是考虑到框架的损耗应该是保持一致的，实际上框架自身的消耗应该是小于0.12ms的，而道具效果不同造成的差异达到了0.23ms，将近或者超越了两倍框架带来的损耗，从实际的性能优化角度来看，框架的性能压力应该和道具的数量成正相关，而道具的性能压力主要来自于道具的效果表现，优化道具效果能够比优化框架带来更多的性能增益。在实际的游戏需求中，30帧游戏的时间要求为33ms，60帧游戏的时间要求为16.7ms，120帧游戏的时间要求为8.33ms，240帧游戏的时间要求为4.16ms，可以看到即便对于较为苛刻的高帧游戏需求来看，该框架的0.12ms消耗也远不会导致游戏流畅度的下降（假设垃圾回收机制稳定运作且不发生内存泄漏）。

表4-1 高并发场合框架性能损耗测试结果

## 第五章 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.

[11] Marsaglia G. Xorshift rngs[J]. Journal of Statistical software, 2003, 8: 1-6.

[12] Visual Studio[EB/OL]. <https://visualstudio.microsoft.com>

[13] Rider[EB/OL]. <https://www.jetbrains.com/rider>

[14] LuaPanda[EB/OL]. https://github.com/Tencent/LuaPanda

## 致谢