

1. Introduction

1.1 变革

1.1.1 形态变化

1.1.2 计算架构的变化

1.1.2.1 业务比喻

1.1.3 技术挑战

1.2 现状

1.3 技术优势

1.3.1 无代码交互内容创作

1.3.2 代码复用机制

1.3.3 高性能、低功耗

1.3.4 大规模并发、分布式

1.3.4 自我进化的标准架构

1.4 商业模式

1.4.1 加强朋友间在线互动的最好形式

1.4.2 理想的广告-新型虚拟经济体验

1.4.2.1 广告本身就是产品

1.4.3 真正的“市场经济”

1.4.4 以标准和组件为核心的抽成机制

1.4.5 持续消费

1.5 用户

1.5.1 普通用户

1.5.2 创作用户

1.5.2.1 一个Creation的创作流程

1.5.3 开发者

1.5.4 标准作者

1.6 创造增量价值

2. Reality Interoperable System

2.1 Creation Scene Description (CSD)

2.2 Creation Script

2.2.1 全局变量表 (符号表)

2.2.1.1 动态的符号表

2.2.1.2 符号表

2.2.1.3 符号表定义标准

- 2.2.1.4 版本与兼容性
- 2.2.2 Entity
- 2.2.3 三种变量类型
- 2.2.4 数据抽象：共享变量与解耦
 - 2.2.4.1 共享变量的声明
 - 2.2.4.2 共享变量的初始化
 - 2.2.4.3 函数参数解耦
 - 2.2.4.4 赋值与解耦
- 2.2.5 组件语义化
 - 2.2.5.1 语义化与可视化
 - 2.2.5.2 能否自动生成Machinations
 - 2.2.5.3 复杂系统的仿真
- 2.2.6 组件查询
 - 2.2.6.1 组件组合不是理想的查询方法
 - 2.2.6.2 显式声明
 - 2.2.6.3 RUST ECS
 - 2.2.6.4 Labels/Layers
- 2.2.7 Component + System
 - 2.2.7.1 组件执行顺序
 - 2.2.7.2 Change-driven update
 - 2.2.7.3 ECS
 - 2.2.7.4 ECS参数
 - 2.2.7.5 组件之间的通信
- 2.2.8 用户间通信
 - 2.2.8.1 系统机制
 - 2.2.8.2 组件安全
 - 2.2.8.3 权限控制
- 2.2.9 数据与存档
 - 2.2.9.1 数据配置
 - 2.2.9.2 存档
- 2.2.10 通用性
 - 2.2.10.1 独立类
 - 2.2.10.2 没有返回值的函数调用
 - 2.2.10.3 有返回值

- 2.2.10.4 继承
- 2.2.10.5 结构体
- 2.2.10.6 控制tick的频率
- 2.2.11 赋值与解耦
 - 2.2.11.1 去除直接赋值
 - 2.2.11.2 拉取还是传入
- 2.2.12 符号泛型
- 2.2.13 并发
 - 2.2.13.1 过程式编程
 - 2.2.13.2 Erlang及OOP
 - 2.2.13.3 适合游戏程序的并发模型
- 2.2.14 智能感知
- 2.2.15 交易与交互
 - 2.2.14.1 HelpComponent
- 2.2.15 Components
 - 2.2.15.1 NeuralComponent
 - 2.2.15.2 TagComponent
 - 2.2.15.3 RealityIDComponent
 - 2.2.15.4 HelpComponent
- 2.2.16 最佳实践
 - 2.2.16.1 more granular is better
 - 2.2.16.2 组件顺序
- 2.2.17 关于数据的本质
 - 2.2.17.1 数据泛型
 - 2.2.17.2 接口、协议、参数、数据
- 2.2.18 游戏程序跟传统程序的区别
 - 2.2.18.1 Update机制
 - 2.2.18.2 程序大小与空间数据结构
 - 2.2.18.3 架构复杂度
- 2.2.19 状态机、行为树与AI

2.3 Creation VM

- 2.3.1 Creation Table Engine
 - 2.3.1.1 数据存储：数组
 - 2.3.1.2 无类型定义与垃圾回收

2.3.2 Add、Remove

2.3.3 Change-driven Update

2.3.4 编译

2.3.5 链接和加载

2.3.6 组件关系管理

2.4. CreationXR

2.4.1 Unified XR Input

2.4.2 XR Scene Understanding

2.4.3 Data-driven Architecture

2.4.4 交互

 2.4.4.1 基于空间的交互

 2.4.4.2 基于语音+AI的交互

2.5 Creation AI

2.5.1 Semantics-based Creating

2.5.2 Procedural Content Generation

2.5.3 Intelligent Simulation

2.5.4 Research

2.6 Creation Cloud

2.6.1 Creation Management

 2.6.1.1 CreationID

2.6.2 Creation Code Library

 2.6.2.1 标准管理

 2.6.2.2 组件及包管理

2.6.3 Multi-player Services

 2.6.3.1 Voice Service

2.6.4 端云协同

 2.6.4.1 在云端执行脚本

 2.6.4.2 Client as a Display

 2.6.4.3 在云端计算需要加载的组件

 2.6.4.4 计算实体和组件的关系

2.6.5 并行计算

 2.6.5.1 分布式Creation Table

2.6.6 RPC

2.7 核心编程思想

- 2.7.1 避免全局变量
 - 2.7.2 函数式编程
 - 2.7.3 数据驱动
 - 2.7.4 ECS
 - 2.7.5 包及依赖管理
 - 2.7.6 动态解释
- 2.8 大型动态系统
- 2.8.1 动态编译
 - 2.8.2 动态创建和修改
 - 2.8.3 动态加载场景
 - 2.8.4 动态推送更新
 - 2.8.5 动态分配服务器

3. Reality Create

- 3.1 Creation ID
- 3.2 Creation Simulation
- 3.3 UI组件
 - 3.3.1 Bevy UI
 - 3.3.2 统一编辑态和运行态

4. Reality World

- 4.1 Reality ID
 - 4.1.1 (用户) 组件管理
 - 4.1.2 (用户实体) 权限管理
- 4.2 The Reality World app
 - 4.2.1 真实世界作为底图
 - 4.2.1.1 跟现实世界关联
 - 4.2.1.2 真实地理的意义
 - 4.2.1.3 作为巨大虚拟世界的地图
 - 4.2.2 传送门
 - 4.2.3 Point and Click
- 4.3 源动力
 - 4.3.1 用户：创作和体验自由度
 - 4.3.2 开发者：更活跃的经济市场
 - 4.3.2.1 持续经济
 - 4.3.2.2 广告内容

4.3.3 标准作者：高级抽象能力的巨大收益

4.4 安全和所有权

4.4.1 RealityIDComponent

4.4.2 readonly

4.4.3 重新加载

4.5 稳定性

4.5.1 Reality Verified Components

4.5.2 提前预测过期组件

4.6 经济与交易

4.6.1 及时购买

4.6.2 智能购买

4.6.3 直接发布而不是广告

4.6.3.1 现在的可能做法

4.6.4 市场经济

4.6.4.1 单次购买不具备任何价值

4.6.5 区块链

4.6.5.1 价值关联

4.6.5.2 价值的决定

4.6.5.3 转卖没有创造价值

4.6.6 Royalty

4.6.6.1 标准税

4.6.6.2 组件税

4.6.9 完整的生态

4.7 Social

4.7.1 私人化社交

4.7.2 关注现实

4.8 创造性和开放世界

4.8.1 分工的重要性

4.8.2 共同创造拉近距离

4.8.3 创造游戏性与一般创造

4.8.4 时间创造价值

4.8.5 创造的方式

4.8.6 大世界合成能力

4.8.7 虚拟世界巨大的探索成本

4.9 Third party apps

4.10 社会价值

4.10.1 更好的信息传播媒介即信息表达方法

4.10.2 释放实时模拟程序机制的潜能

4.10.3 复杂系统的模拟

4.11 标准

4.11.1 传统做法的缺点

4.11.2 开放的标准架构

4.11.3 标准管理

4.11.3.1 标准更新通知

4.11.3.2 组件更新通知

4.11.3.3 标准反馈机制

4.11.3.4 始终保持最新（版本管理机制）

4.11.4 跨越标准

4.11.4.1 与相关标准的属性

4.12 自我进化的Metaverse

4.12.1 标准的价值

4.12.2 基于标准的商业模式

4.12.3 自我进化的标准

4.12.3.1 自我进化的核心机制

4.12.4 标准的自我进化

4.12.4.1 符号表的版本兼容性

4.12.4.2 以标准为中心的社区

4.12.4.3 推动组件更新

4.13 用户创作

4.13.1 以组件为最小粒度

4.13.2 以标准为整体思维

4.13.3 反馈和评价

4.13.4 实体和组件管理

5. Applications

6. 核心参考架构

6.1 数据格式

6.1.1 USD

6.1.1.1 新思想

- 6.1.1.2 技术方案
- 6.1.1.3 USDZ
- 6.1.1.4 不足与原因
- 6.1.1.5 对比
- 6.1.2 Alembic formats
- 6.1.3 Unity Prefabs
- 6.2 数据驱动架构
 - 6.2.1 Unity DOTS/ECS
 - 6.2.2 ECS
 - 6.2.3 UE5 MASS
 - 6.2.4 Data-oriented and -driven
 - 6.2.5 Rust ECS
- 6.3 编译器与DSL
 - 6.3.1 Taichi
 - 6.3.1.1 新思想
 - 6.3.1.2 技术方案
 - 6.3.1.3 不足及原因
 - 6.3.1.4 对比
 - 6.3.2 Modular AI
 - 6.3.2.1 新思想
- 6.4 Others
 - 6.4.1 神经网络
 - 6.4.2 开源代码管理, 如pip
 - 6.4.3 区块链/NFT/虚拟货币
 - 6.4.4 Rust
 - 6.4.5 Unity EditorXR and SceneFusion
 - 6.4.6 BEVYengine
 - 6.4.7 工程学, 工业设计
 - 6.4.8 magicavoxel
 - 6.4.9 所有平台的插件架构
 - 6.4.10 传统DCC流程
 - 6.4.11 Meta Builder bot
 - 6.4.12 Houdini: Node-based Workflow
 - 6.4.13 realityOS

- 6.4.14 OpenXR
- 6.4.15 ECS + AI
- 6.4.16 Unreal blueprint
- 6.4.17 Pixar
- 6.4.18 Gaia procedural-worlds.com
- 6.4.19 Google Maps API
- 6.4.20 Procedural content generation
- 6.4.21 casualcreator
- 6.4.22 Scalar, 不鸣科技, 微服务化
- 6.4.23 Google Tilt Brush
- 6.4.24 GitHub
- 6.4.25 Stechfab
- 6.4.26 Game pigeon: games for iMessage

6.5 应用

- 6.5.1 堡垒之夜
 - 6.5.1.1 定义交互类型很重要
 - 6.5.1.2 多人在线服务
 - 6.5.1.3 私密社交
 - 6.5.1.4 Verse Language
- 6.5.2 ROBLOX
- 6.5.3 Niantic
- 6.5.4 SNAPCHAT
- 6.5.5 Meta
- 6.5.6 Omniverse
- 6.5.7 Minecraft
- 6.5.8 Wilder World

7. Programming Language

- 7.1 Script languages
 - 7.1.1 SkookumScript
 - 7.1.1.1 Time-flow logic
 - 7.1.1.2 Conditional flow control
 - 7.1.2 Lua
 - 7.1.3 GameMonkey
 - 7.1.4 Python

7.1.5 JavaScript

7.1.6 TypeScript

7.1.7 SCUMM

7.1.8 Mono-script

7.1.9 AngelScript

7.1.10 Scheme/Guile

7.1.11 ActionScript

7.1.12 mruby

7.2 Erlang

7.2.1 Beam VM

8. 拟娲哲学

8.1 拟娲哲学第一课

8.1.1 元宇宙的社会价值是什么？

8.2 拟娲哲学第二课

8.2.1 RealityIS的本质是什么？

8.3 拟娲哲学第三课

8.3.1 标准及自我进化

8.4 拟娲哲学第四课

8.4.1 Reality World中的“市场经济”机制

8.5 拟娲哲学第五课

8.5.1 怎样构建大规模、大并发系统

1. INTRODUCTION

游戏，作为一种模拟真实世界实时运行的系统和机制，它不光在产品形态上跟一般的应用程序体验不一样，例如一般应用程序一般是功能型的，其功能是明确而具体的，而游戏往往是一种体验，没有直观确定性的功能，每个人获得的体验可能都不一样，它的整个程序组织及其开发工具更是与传统的应用程序不一样，例如传统的应用程序通常是按顺序执行，而游戏为了实现对真实实时世界的模拟，需要以一种不间断的实时轮询的机制。这种轮询不光造就了游戏中各个动态系统的实时性，它对整个应用程序的架构，以及这种应用程序的表达能力，都提供了非常不一样的可能性和结果。

多年来，这种应用程序机制主要被用来制作游戏，而游戏这种程序机制的一些难点，通常需要非常专业的游戏公司才能做出不错的游戏产品。而反观传统的应用程序，由于它们的机制更简单，易于学习和理解，因此被更广泛的使用，不仅对人们的生活带来更大的影响，也大大地促进了社会进步。

今年来，随着虚拟现实和元宇宙概念和趋势的兴起，这种实时模拟系统越来越频繁被用于到游戏之外更泛化的领域，例如：

- 通过手机的AR功能，**Snapchat**给用户提供了丰富的滤镜体验，不同于传统视频和图片，这些滤镜是交互式的，用户可以基于这种交互能力生成还富有表达能力的视频，借助这种能力，**Snapchat**迅速称为一款流行的社交应用。
- 同样是基于手机AR的功能，**Niantic**借助手机后置摄像头的视觉定位能力（VPS），开发了诸如Pokemon Go等应用，这种新的基于真实地理位置的应用跟人们在真实世界中的活动联系起来，并借助3D互动的能力，把人们的生活联系得更紧密，是一款典型用于增强社交关系的应用。
- 以**Roblox**为首的创作类工具，通过简化程序分发和部署、提供统一的多人在线等服务，降低了游戏开发的门槛，使得更多的中小个人创作互动内容更加简单。并通过云原生的多人在线，使社交游戏的效果被放大，成为未来的重要趋势。
- 由类Minecraft沙盒机制延伸的大逃杀沙盒游戏《堡垒之夜》，借助更好的多人在线服务，例如包括对多人实时游戏更友好的在线语音服务等，使得社交属性在《堡垒之夜》被进一步加强。更好的多人在线服务以及堡垒之夜本身逃生类游戏更好地协同机制，使得堡垒之夜的社交属性称为体验最好的社交属性，其开创和举办的虚拟派对Marshmello更是掀起了虚拟演唱会的热潮。

所有这些变化和发展，对互动内容的开发及生态都带来了巨大的影响和变化，这些影响和变化后面，需要全新的技术范式，而这些新的技术范式又将创造增量的价值和体验。

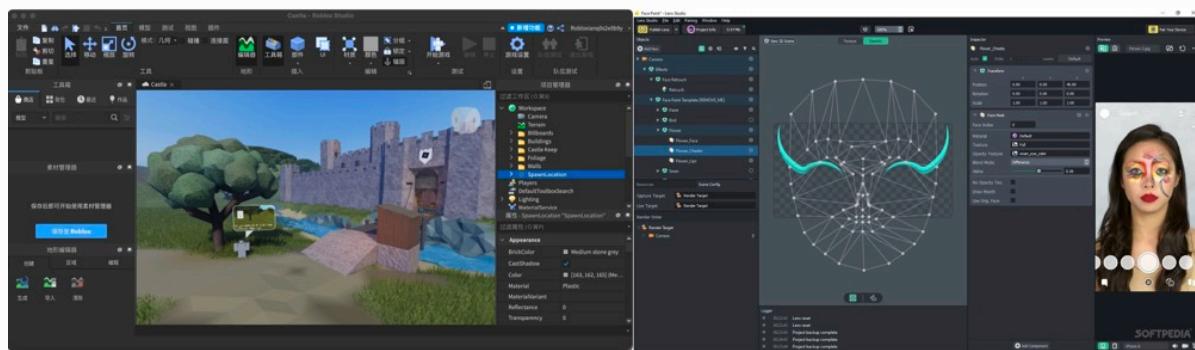
1.1 变革

1.1.1 形态变化

上述这些变化，从上往下看，可以归结为三个维度的变化：

- **开发者：**由专业开发者向普通用户转移
- **开发方式：**互动内容的开发方式由完全开发到基于事件驱动开发
- **玩家体验，**由完全操控到XR辅助

在开发工具方面，Roblox和Snapchat的Lens Studio都面向普通开发者，它们共同的特点包括简化的脚本，以及一键发布，使得开发者不需要花费很大的精力去处理平台相关问题。另一方面是这类工具都是深入集成平台的特定功能，例如Lens Studio底层的AR场景理解算法，以及Roblox内置的多人在线服务。深度集成平台与算法，相比于传统通用游戏引擎，将成为未来的一个方向和优势。



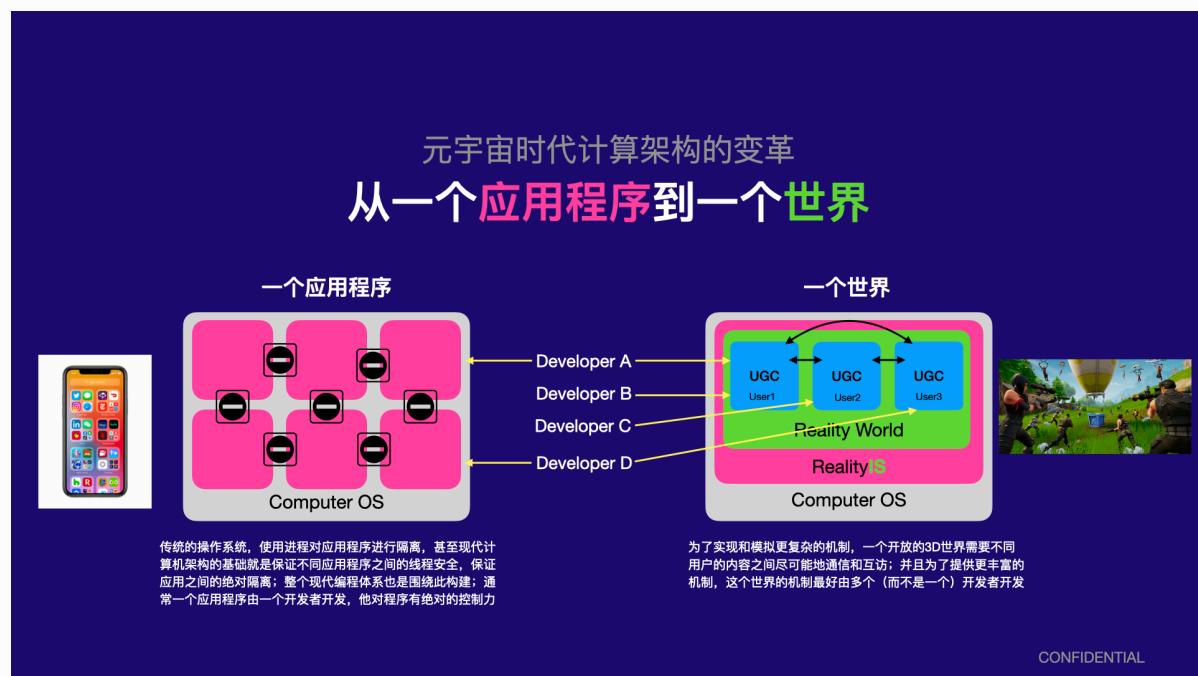
在开发方式方面，传统的流程需要开发者定义所有的逻辑，包括触发逻辑的机制，这部分尤其复杂。在Snapchat基于AR的互动内容开发中，这类互动内容的驱动完全来源于手机对场景理解，这些都由AR算法来提供，因此开发者不需要处理任何交互驱动方面的逻辑，而只需要关注对交互的响应逻辑。这种方式不仅大大简化了整个互动内容的开发，也从根本上对互动内容开发的流程带来了很大的变革。简化逻辑开发的触发机制，触发机制数据化，深度与平台集成，是未来互动内容开发走向平民化的重要方式之一。当然在这个过程中，基于代理的物体位置摆放机制也起到了很大的辅助作用。

同样借助与事件触发机制的数据化，以及基于XR设备对场景的理解，用户对互动内容的部分操作，由原来主动、精细地控制虚拟摄像机和物体，变为基于场景理解算法的自动驱动，这大大简化了交互成本。

这些各个层面的变化，最终都会导致整个互动内容的制作流程会发生较大的变化。

1.1.2 计算架构的变化

从开发的角度，从下往上看，这带来的是计算架构的变革。



传统的计算架构都是为单个应用程序设计的，从硬件到软件，所有一切流程和功能都是针对这个模型设计的，例如一个应用程序的所有源码都会被编译和链接到一起，一个应用程序内的数据可能相互引用，所有需要链接器来重新定位每个引用变量的地址。这样的计算架构，非常适合于处理具有独立功能的应用程序。但它有比较致命的缺点：

- 例如因为所有源代码编译的目标代码都会链接到一起，所以它们从根本上就不支持大规模应用程序，因为这样的应用程序可能由海量的源代码组成。
- 由于源代码之间相互引用，因此它们很难支持应用程序内的独立子程序通信，子程序之间总是需要引用源代码才可以通信，这使得一个应用程序无法成为一个可以自我进化的开放系统，而总是需要一个开发商来进行维护

随着UGC和元宇宙时代的到来，这种大规模的、具有内生开放子系统的多应用交互架构越来越成为最核心的需求，这需要我们在计算架构上做出较大的变革。

1.1.2.1 业务比喻

比如现在腾讯有非常海量的业务，比如微信，音乐，视频等，目前这些业务之间相互是比较独立的，他们组织为相互独立的应用程序，相互比较独立的数据管理，服务器架构和组织，虽然彼此之间存在一定关联，但是这种关联是高度结构化和规则化的，且关联很少。

开放世界则意味着，现在所有这些业务需要在一个应用内组织，它的复杂度是非常高的，架构也非常复杂，数据管理和分布式计算都非常复杂，并且由于传统游戏的逻辑组织方式，在这种体量下根本无法有效管理组织和进行分布式计算

可行的思路：

- 微服务化
- 函数式编程

微服务化是一种软件架构，需要高度依赖于对逻辑的设计和划分，他不是一种基础编程模型，因此无法支撑开放式的设计，这些微服务通常都只能是开发商设计好的，普通用户没法修改，因此本质上不支持开放世界。

1.1.3 技术挑战

从根本上，上述的一些变革带来的技术挑战包括：

- 需要全新支持多应用相互通信的计算架构
- 能够在所有计算轻松在多个服务器之间进行分布式计算的数据和计算架构

当然由这两个底层根本性的挑战，上层还包括一些其他挑战，比如独立程序之间高效的通信标准或者机制，用户对权限控制与代码的分离，普通用户怎样无代码编程等等，我们将在后面进一步分析。

1.2 现状

不能随时随地高效的内容创建，与随时随地方便的将创建的内容进行社交分享，是当前最大的问题，或者说是3D数字世界发展的一些重要基础；前者的难点在于它需要编程（通常需要面向对象的编程能力），所有逻辑都是通过面向对象的方式创建出来的（大部分人都不具备）；后者的难点在于它需要一套新的支持分布式协同编辑和分发的底层数据格式表述，

这与当前大部分引擎私有格式都不一样

1、主要基于场景理解

当前大部分AR应用都是基于场景理解的简单特效/滤镜呈现，以及围绕这些滤镜非常简单的交互，没有围绕场景构建做太多事情，即堡垒之夜的方式，包括更复杂的交互

2、3D内容分发

当前大部分不能分发3D内容本身，而是分享录制的视频；或者只是分享相同的滤镜给朋友，使用体验没有差异性

3、用户创建参与感很弱

绝大部分都是场景理解驱动，仅有纯视觉的体验，没有3D真正复杂的交互和创建体验

4、大部分聚焦虚拟空间

元宇宙即社交方向行业大部分在做虚拟空间：元象、RecRoom、monoAI等

5、3D创建的价值/Minecraft

除了精心的游戏规则设计（这需要很高的开发技巧和能力），3D内容的创建本身也是很有趣，而能够创建自己设计的一定规则的内容就更有趣

大部分没有集中于面向普通用户创建3D内容的能力或体验，Roblox 面向有一定经验的开发者，并且开发模式本质上与传统游戏开发类似，只是分发机制不一样，NV和太极面向用户创作，但是没有面向移动端，而且是纯内容创作

6、底层创新

当前普遍聚焦上层架构，围绕怎样将传统游戏的流程和体验往元宇宙上靠，其结果大多数虚拟空间类产品，聚焦底层协作的主要是Omniverse和太极，其中深入底层性能的只有太极，但在中间层上下协同方面缺乏重点创新，其中太极和元象都在主打云渲染，太极还强调在线协同编辑，Omniverse则支持一般的协同编辑

7、虚拟空间也主要是纯视觉体验

大部分只是在里面行走，不能做太多事情，更不能创作，或者有的也是固定规则的交互，例如Roblox 里面的一些模拟经验的游戏

8、游戏更新的时间

一般需要重新发布的流程，Ubisoft Scalar试图通过云端微服务架构实现及时更新，另外云原生基本上天生就是及时更新的，因为它的资源都在云端

1.3 技术优势

1.3.1 无代码交互内容创作

现在大部分应用都是在设计一个特定场景，提供官方特定的应用、规则或玩法，跟游戏的思路类似，更强调用户的体验

除了游戏引擎，能让用户创作的只有Roblox 只类型的

能够面向普通用户，且针对交互内容进行创作的，只有一家，它是未来最基础的模式和能力，当前没有一家在这个层面，独此一家

用户可以基于RealityWorld 创作游戏，或者简单的作品，跟Roblox 一样，但是它通过两个门槛以及丰富创作的类别，让更多普通用户可以参与，从而大大增加用户群，使得开发者在这个平台上开发游戏有可能具有更大的用户群

1.3.2 代码复用机制

CreationXR runtime，可以被任意第三方app集成

他颠覆了传统开发者生态的方式，传统的小程序或者Roblox 或者Snapchat 只能集中式，只有平台一个入口，而像Niantic 这种仅开放底层能力：

- 开发者接入实际很困难，因为要调用众多的API
- 每个app接入的方式存在冗余，重复，这部分可以共享
- 每个app接入的方式不一样，因此每个app开发的功能本质上类似，但是3D部份却要重复开发，例如要接入预览流等等
- 无法在自己的app里面共享一套开发标准，Unity不算标准，因为他更偏低层，没有定

义太多规范，导致每个人开发不一样，而RealityCreate 是高度规范化的

这使得第三方开发者可以共享RealityCreate 高度规范化的流程，又可以最简单的成本和方式接入自己app，从而是开发者聚焦创作本身，同时能享受云原生，用户协作等等所有Creation XR得好处

这样也可以避免in-house 引擎的问题，in-house 引擎主要的问题是不能被其他app简单使用，有固定的流程，这样：

- 既可以按照in-house 的做法快速提升自己的差异性，而避免陷入Unreal和Unity 那样的通用引擎
- 又能像Unity一样被用于开发独立应用
- 还抓住了生态

这种模式还有一个好处，除了应用部署本身，他还提供如应用的统计分析等功能，应用的存储，这些原本都是要开发者自己去对接的，传统的应用开发看有多么复杂：

- 开发者自己自己Unity开发app程序，其中包括大量的美术资源和逻辑开发是每个游戏开发者比较重复的
- 自己打包部署，这要求一些平台层的技巧和经验，这些其实超出了开发者对内容的聚焦，除了内容和玩法，一个游戏创业公司还需要花很多资源
- 然后还需要对接各种SDK，包括端侧的和云侧的，而各种SDK都要公司自己学习，并且不同公司之间这些工作也是重复的，但是每个公司都的招人去挖坑，有时候这些反而成为一些小团队比较阻碍发布的一些障碍
- 后期的运维也需要一些精力和人力也自己开发经验的投入

总之，游戏开发团队花了较多的精力在一些繁琐的事情上，而且公司之间的这些能力本可以共享的，开发者本应该聚焦内容开发

原因是因为开发者要自己发布app，所以没有办法去统一集成一些东西，这些东西本身没有标准，很难统一

当然大的应用本身需要更灵活的能力，有太多限制反而不利于开发，但是对于一些小内容，尤其是个人内容，这是可行的，而且个人开发者需要这样的生态

Roblox 就是做了这些事情，应用开发者不需要担心其他事情，但：

- 它不支持Roblox之外的分发
- 它不支持链接的方式分享
- 编程模型不一样

- 主要面向PC和手机，没有针对XR的算法层接口封装
- 编程模型不一样所以不可能支持XR设备创作
- 还是典型的专业开发+普通用户玩的模式，没有普通用户的创建模式和创造体验

以此为基础，构建微服务架构

美术资源的重复问题

- 传统互动内容最重要的是玩法、故事，美术相对不是最核心的
- 美术资源能够提供独特的视觉语言，但是这些视觉语言更多是风格化的类型，例如在同一种写实风格类型下，玩家对不同的场景感知的差异就会小很多
- 美术场景通过程序化生成方法是能提供较好的差异性和独特性的
- 如果这种程序化方法更加支持风格化，那么就能满足上述的需求
- 风格化主要表现为纹理或者基础材质，有时网格也有一定的差异，但是这都可以进行研究

1.3.3 高性能、低功耗

现代游戏程序通常基于OOP进行开发，其中的引用关系错综复杂，对现代内存硬件架构极度不友好，需要重新对游戏的运行时内存数据进行更好的管理，并且这些管理又不能给开发者带来成本。

参见2.3.1节。

1.3.4 大规模并发、分布式

基于上述相似的原因，现代游戏程序无法使用大规模并发的需求，例如单台服务器最多只能支持上百人同时在线。这主要是游戏程序内的程序和数据耦合度非常高，导致单台服务器必须加载所有的数据，这样的方式不管单台服务器的内存不够，也会带来多台服务器重复加载，以及重复加载导致的数据同步导致的复杂问题。

参见2.2.13节。

1.3.4 自我进化的标准架构

在一个传统的游戏中，所有的逻辑都是包含在程序中不可修改，所有的关卡、剧情等等都是固定的，跟电影比较类似，唯一不同的可能是游戏具有交互性。

当需要更新程序时，通常通过DLC或Mod等机制对游戏进行扩展或增强，但是这两种机制通常都比较受限，因此本质上，至少一个已经发布的游戏其核心体验是很难改变的。

但一个开放的Metaverse不仅需要可以任意添加独立程序的能力，还需要能够像真实世界一样自动筛选优秀内容的能力，否则用户可能很快就沉入很多垃圾或者质量较低的信息当中。

传统的数字经济，这种内容都是需要平台使用一定的算法进行推荐，这种推荐算法一般由用户对自己的内容设置一些标签，然后平台建立一些相关度的机制。

但真实世界却是相反的，它们由每个根据自己的判断和选择，来促进整个世界的进化。参见4.12节内容。

1.4 商业模式

1.4.1 加强朋友间在线互动的最好形式

传统的在线互动有三种形式：

- 视频电话
- 多人在线游戏，如《刺激战场》
- 《Roblox》类的虚拟房间

其中游戏类的多人在线，还是以游戏为主，这些交互通常只是聊天或者语音，有点像在现场一起玩游戏大家可以相互讨论；整体的活动还是以玩游戏为主，互动是辅助的形式。

房间类的交互，相对私密一点，但是这些应用的游戏性往往很弱，比较局限于形式上的在线互动，目的性和娱乐性都不够强。

Reality World相对上述的模式，存在一下的一些独特区别：

- 互动的规则和内容往往是可以由其中一个用户创作的，具有独特性，针对性，比如针对一个生日专门设置的与朋友相关的场景和互动内容；这种独特性使得**Reality World**的内容更容易在好友之间发起互动
- 可以在互动的过程中进行共同创作，比如你向蛋糕上切一刀下去，所有人都可以看到蛋糕被切成两份，这是因为**Reality World**的场景多人协作特性，而其他互动的内容是固定的，每个玩家只能体验这些设计好的内容
- **Reality World**的内容可以即使创作和分享，不需要提交到商店，然后用户下载，只需

要马上创作之后发送给好友一个链接就可以及时打开

1.4.2 理想的广告-新型虚拟经济体验

在现代数字经济中，除了视频、图片和音乐这种能直接体验的数字内容，其他的大部分内容，其实体和对应的数字表述都是分割的，例如淘宝买的商品只是数字化后的一个记录，你必须收到东西之后才能体验。

另一种与之相关的数字经济是广告，广告作为一个展示产品的方式，在真实世界中，它们往往以视频或者图片的形式呈现。然而在这种方式中，广告语产品通常是割裂的，受限于实物需要场地及运输等问题，我们并不能总是在任意一个广告旁边放上实物，使得任何看到广告的人就可以购买。

然而，对于后者才是理想的广告形态：就是广告本身就是产品，或者说广告可以一键直达产品体验和购买。

1.4.2.1 广告本身就是产品

如果广告的产品就是数字内容，而非实体内容，理论上是可以做到这种一键体验的效果的。比如如果产品是一个游戏或者一个应用程序，就可以直接点击下载，这种形式现在很普遍。然而这种方式并不是最好的模式，因为：

- 所有的产品都得开发一个app
- 用户可能并不想要安装那么多app
- 因为每个app安装除了体验产品还有很多额外的负担：注册，登录，进去了解软件的导航功能等等

所以元宇宙是一个更好的广告平台，每个产品只需要设计一个交互，玩家直接体验一下就是。但是现在的技术并不能做到这种体验，这里面的原因：

- 一是平台无法支撑任何开发者自由开发交互内容，往往只能通过平台发布，这样广告能力很受限
- 即使增加了新功能，也需要所有玩家都更新，有时候一个广告只有少数人有体验需求

总之，Reality World可以做到厂商可以任意发布带有交互的广告，然后任何玩家只要看到它，就可以及时体验和购买，真正做到：产品及广告，广告及产品，这种模式有望重塑一个全新的虚拟经济形式。

如果广告的内容是实体内容，仍然可以虚拟化体验，或者通过交互，相比视频和图片更好的了解产品。

见4.6.3节内容。

1.4.3 真正的“市场经济”

即市场会决定哪些东西更有价值，这是与传统数字经济系统根本性的不同，传统的数字经济都需要由平台实现某种推荐或者排序算法，例如微博的信息，知乎的文章，淘宝的商品，抖音的视频，这就要求基于一定的标签，分类等机制，信息发布者需要去维护这种标签分类。

然而真实世界的经济却不是这样的，我们所有的一切不是由类似国家或中央的官方机构决定的，而是靠人们自己的选择，促进整个世界的运转。

类似真实世界公司之间的销售，产品越好卖的越多，售价也可以随市场调整。

而且这种机制促进作品的不断改进，就是iPhone手机一样，而传统的内容都是一次性发布，缺乏对原产品的改进机会。游戏也一般由于太复杂，发布后不会有大的改进。目前这些数字经济跟真实世界的经济都不一样。

可以认为它们都是“结构化”的经济，而不是市场经济。

真正的市场经济会促使和催生更多的好内容，更多的人参与。而传统的数字经济，都是少数人在参与或获利。

在真实生活中，每个人都在参与经济贡献；而在目前的自媒体时代，只有少数人在参与经济贡献，大部分都是消费者。

这有机会使得整个经济系统的活力更大：传统的数字化经济都是靠阅读量类似不准确的机制，在这种机制下创作者倾向于作弊买量，而不是创作更好的内容。此外，阅读量本身是个不准确的度量，例如用户可能只是打开了页面就关闭了，根本就没有深入了解对应的内容。而这种通过“实际使用”而不是“查看页面”转化而来对产品的经济定义，更容易促进用户进行更好的创作，就像真实世界一样。见4.3节更多描述。

1.4.4 以标准和组件为核心的抽成机制

传统的NFT类的数字交易市场，交易的是一个数字内容，是一个拷贝，这个拷贝除非通过一定的手段跟踪转售记录，或者甚至限制转售，很难保证创作者的权益。

而Reality World交易的主要是组件和标准，这些组件和标准并不会拷贝一份，而只是一个引用，然后运行时动态从源头拉取最新代码。所以他天生就可以保证了解使用者的情况，比如一个标准能够追踪到所有使用其标准的组件，也能够追踪到所有使用这些标准的用户。

这样标准开发者不能够收取所有使用者的费用，并且还有很好的更新机制，通知用户购买相关和最新产品，就像真实世界一样。

1.4.5 持续消费

传统的数字化进程中，数字化产品往往是一次性消费，这导致：

- 软件开发者升级动力不大，对创新及创新的速度是极为不利的，因为新用户会越来越少
- 大量陈旧代码，一方面是平台兼容成本高，一方面是用户使用比较陈旧的技术或体验

需要改变这种局面，才能更大程度地激活数字消费。

参见4.12.4.3节内容。

1.5 用户

Reality World平台有四种类型的用户，当然这里只从创作层面区分，不涉及商业方面的分类或者逻辑：

- 普通用户：类似于抖音平台只观看视频，从来没有或者很少发布内容的用户
- 创作用户：指只在XR设备上，不借助PC编辑器的情况下进行内容创作
- 开发者：使用PC编辑器Reality Create基于标准进行组件开发
- 标准作者：基于对现实世界的理解提出某种抽象，并将其转换为标准，以及持续维护标准的更新

1.5.1 普通用户

尽管普通用户不进行任何形式的创作，但TA仍然是整个经济系统中很重要的一部分，例如：

- TA都其他创作内容的使用和体验产生消费
- TA通过私人社交网络，产生的对好的内容的主动推荐行为，促进了整个市场经济

当然所有人都是普通用户，并且普通用户也有可能转换为其他创作用户。

1.5.2 创作用户

平台很大一部分技术的架构都是为了创作用户，这是区别其他类似平台的关键。

传统面向普通用户的创作有两类主流方式：

- **《堡垒之夜》之类的沙盒游戏**，在这类游戏中，整个世界的规则类型比较一致，比如《堡垒之夜》的堡垒建造与逃生，《我的世界》中的怪物机制等，这些机制内置于系统中，平台提供大量具有固定行为的物件，玩家创作的自由度相对较小：基本上不涉及逻辑本身的构造，只有跟物理位置，物体组合等相关与游戏行为无关的策略
- **AR事件驱动的增强现实体验**，这些应用以《Snapchat》为代表，它提供一些固定的具有互动体验的道具或者滤镜，用户借助摄像机进行体验；这种体验本身不涉及3D的创作，例如制作一个新的滤镜或者一种新的体验，但是它们产生了一个独一无二的视频内容，并且这个内容是跟自己高度相关的。

《Snapchat》的模板只能在PC端制作，《Reality World》则可以及时创作《Snapchat》类似的模板，并且可以选择更丰富的功能组合。

为此，《Reality World》需要支持任意的组件使用，组件之间可以任意协作，这样才能不限制创作，不然就会很容易局限于一个特定的组件包，或者一些特定的互动类型，**任意组件之间可以通信和组合是Reality World独特的功能，它能够释放创作者无限的创意。**

XR设备上的创作用户必须购买资源，因为他们只能基于已有的资源进行创作，当然有一些资源包或者组件是帮助程序化生成内容的，这类组件可以生成一些随机不固定的内容。资源的类型包括：

- 静态资源：模型（如树木、汽车、弓箭等）、纹理、粒子特效、动画等，Reality World官方应该提供较多的基础资源
- 功能组件，组件是最基本的行为，它们是用户看不见的逻辑代码，它们用来控制物体在游戏世界中的行为；不同组件包之间的组件可以任何组合，创作者需要区分它们的功能，才能生成更好的合理的交互逻辑。这是创作体验的一部分。

- 部件或者物体，由一定的组件组合形成，具有某些特定逻辑功能的游戏对象或者实体，这些实体是直接存在于游戏场景结构中的元素，这些物体可能包含模型、动画、以及能够良好控制这些模型及其行为的组件组合，它们通常是用户直接放置在场景中就可以使用的，类似《堡垒之夜》当中的物体；它们有些也是用于帮助创作一个场景的结构性的组件，例如一个包含TAG的Entity，一个NPC怪物等；也包含一些特殊内置功能的部件，例如Layer表等。

1.5.2.1 一个Creation的创作流程

- 创作者首先浏览Creation商店，下载或者购买一些基本的资源，如上面介绍的静态资源、功能组件和物体等。
- 开发者选择一个物体，将其拖入初始的空场景中
- 然后选择物体对其属性进行编辑，其中可以对其添加组件，组件按类型进行组织，每个组件有结构化的描述及说明，说明应该怎样使用该组件
- 修改组件的属性参数等，组件通过参数来改变物体的行为或者视觉，例如如果是程序化生成组件，则可以生成不同的场景，如果是粒子特效组件，则形成不同的视觉效果
- 播放预览
- 将链接发送给好友
- 好友点击链接加入一起体验
- 如果好友具有权限，可以进行共同编辑，这些编辑也可以是同时在线协同的

1.5.3 开发者

开发者只能基于标准开发组件，一个组件必须支持某个标准，当然一个组件可以支持多个标准，来实现不同标准之间的通信。

标准和组件是隔离的。除来自标准之外的符号，其他符号都是私有变量。

1.5.4 标准作者

某个标准的负责人，当然标准可以转卖，当前负责人不一定是创始作者。

此外，标准不一定需要编程，它仅仅涉及定义与某个抽象相关的数据结构。因此，不具备编程能力，但是具有较强抽象能力的人也可以创建和维护标准。但是从更好的数据结构定义角度，由编程人员维护标准是最合适的，但是编程人员的抽象能力往往不够。所以理想的情况下是某个标准后面有抽象能力较强和编程能力较强两者的组合。

参见4.11和4.12节内容。

1.6 创造增量价值

元宇宙代表的不仅是一种新的体验，它将对整个社会甚至数字信息化的进程带来全方位的影响，这种影响不仅仅是一种新的技术或者一种新的功能那么简单，它将包括对计算架构以及全新的信息表述方式这种深层次的变革。

很显然，这种由用户驱动的全新体验需求，用当下的技术架构是做不到的，而且它的限制的根源来源于更底层的计算架构。在近几十年的计算机时代中，底层的计算架构基本上没有发生太根本性变化，例如我们能感知到的：

- 近10年编译架构基本上没有太大变化，一些10年前的经典著名基本上现在还是适用的
- 近10年编程语言的发展也没有革命性的变化，不仅至2010年之后很少推出全新的语言，大部分语言设计也只是针对一些开发体验层面的选择，很多语言的核心思想甚至早在2000年之前就确定



编译和编程语言是计算架构最重要的指向，因为它们连接计算机硬件和应用软件，它们的变革往往能够决定上层软件形态的变革，从而决定技术带给消费者的变革。

当元宇宙带来真正大规模、互操作、大并发等等这些传统计算模型不能应付的需求时，我们需要全新的思路，在计算架构层面创造全新的增量价值，才有机会驱动整个元宇宙的变革。

2. REALITY INTEROPERABLE SYSTEM

结构抽象，对于XR端的选择，如果卡包太多，会导致选择操作很复杂，参考淘宝购物，大家会把想买的商品放进购物车，最后一次性付款

让用户在手机端坐好归类，精选出确定或者大概率会使用的组件，然后简化实际的选择，甚至通过更加友好的命名规范来使用语言选择

总之就是需要更好的类型化，并且将用户对资源的选择过程中，融入分类，形成一个天然的筛选过程

定义创作的过程：

- 不仅仅是Create中的时间
- 像备忘录笔记一样随时记录
- 资源的选择准备过程
- 看别人的创作也是创作思考过程
- 甚至逻辑上的组织可能有一部份是非视觉相关的，所以资源管理本身要融入创作过程，即它不光是资源管理，也包含一定的逻辑组织，例如故事大纲结构等等，就像策划在组织表格的时候，比如编写人物故事对话
- 一定不能仅依赖于资源购买+XR端选择这样的传统模式，即交互复杂，也不符合实际的创作流程

2.1 CREATION SCENE DESCRIPTION (CSD)

需要加入用户版权信息。

USD的asset resolution机制，使得可以直接加载creation.id的内存，而不需要单独写加载模块，但是需要在自定义resolution中加入权限验证，例如传入消费的app以及用户信息(Reality ID)。

USDZ可能是将一些Behavior转换为Schema，因为这些Schema是C++库，所以需要将USD的C++库放进iOS系统中；虽然USD提供有Python binding，但是它只是接口的封装，因为USD本身也是一门语言，语言本身需要解释或者编译，USD是将新的Schema生成解析的C++文件，所以需要放置在运行时，所以就不能动态定义，只能是系统级别的Schema，开发者通过脚本定义的数据结构还是需要增加一层解析；

为此USD文件中需要包含一些非USD的片段，例如定义一个特殊的Component或者Model，这可以通过asset resolution来与USD一起适配工作。这一部分可以不是USD语言，可以是自己定义的语法；

实际上USD文件不应该让用户看到和编辑，用户看到的是工程或者可视化的描述，或者属性表述。用户或者开发者也不需要编辑这么复杂的结构，USD本质上还是面向数据结构或者程序员的，RW的结构应该更简单、扁平。

2.2 CREATION SCRIPT

Creation有一个核心目标：

- 它应该像Lua一样简单，轻量的运行时。
- 它的说明文档只有50页pdf，每个开发者只需要了解这20页文档，不再需要阅读其他资料已学习更高级的技巧。
- 它同时面向专业开发者和普通用户
- 尽可能少的系统层API，不要全部暴露引擎层的API，甚至可以将引擎层的API修改不同的易于业务侧理解的API名字，它的所有API应该像Houdini一样，全部专注于业务，不需要开发者了解的就不暴露给开发者

Creation Script的核心目标是：

- 首要构建一套能够便于普通用户编辑、和共享逻辑的架构；所以他必须能够动态更新
- 次要目标是简化开发者的开发体验，例如只专注于逻辑，而不是复杂的面向对象组织设计，再比如简化多线程的开发，开发者应该感知不到并发编程；例如不能为了方便对Component进行管理，就要求用户去配置一个包引用文件，而应该自动管理
- 应该是图灵完备的，不能限制开发者的能力
- 他应该基于一个已有的脚本语言，使他专注于上层架构，而不是去构建一套底层语言
- 复杂对象的构建在宿主，脚本主要做轻量计算
- 无垃圾回收，所有堆上的对象均有宿主分配和管理

传统语言几乎都是为了面向对象而生的，所以包含很多为支持面向对象的功能，如果数据驱动是需要的核心，是不是应该有一种新的语言，目前看来Lua更接近这种语言

编译器用途：

- 例如用于检查ECS的结构，不合法的类结构不能被加入到最终程序中
- 例如检查Component 数据内存分配大小等等
- 用于将底层面向对象的能力禁止面向开发者，但是保持底层能力对面向对象的使用

类型检查

TypeScript有很好的的类型检查，但是往往强类型的语言也有一定的限制，例如不允许像Lua一样，在同一作用域内相同的变量名称改变类型，另外对于动态语言来讲，一般类型检查这种功能在运行时也是存在的，但本质上这个功能对于运行时不再必须，如果你能保证被检查过的源码没有被修改，一般语言不会把这个功能作为一个可选项

因此有必要设计一种类型检查，它可以被移除，使得仅在编辑时发生作用，而一旦发布之后，实际的运行时不需要这个类型检查的功能，但是还是会保留全部的源代码信息。并且自定义的类型检查可以容许更少的限制。

怎样为Lua添加新的语法

参考TypeScript相对于JavaScript 添加的功能。

Unity DOTS为什么没有默认把System中的并行性指定去掉，是因为他希望兼容传统的Component脚本，而按照传统的写法，没法去控制行为，所以只能开发者实现并行计算。

否则就需要像Roblox一样，需要用户自己将脚本挂在entity上，这就增加了复杂性

所以需要避免让开发者手动将Component与System之间进行关联

Minecraft通过直接在属性中进行编写MOLANG代码来避免该问题

例如开发者看到的文件或者对象只有Component，这样迫使开发者对数据进行抽象；

对于System，我们首先不需要开发者去关联一个Component和一个System，他们应该自动关联，例如通过Component来打开对应的System文件进行编辑

Component除了自身的逻辑属性，另外一些属性用来控制系统结构，例如System执行的顺序等，可以在Component中明确区分两种数据，或者这些固定结构的数据就以一个Component本身的Property形式显示，避免开发者写错

但System可能需要多个Component的数据，因此可以设计为一个System必须对应一个主Component（即使这个Component可能只是一个名字，而并没有任何数据，是有这种情况的，就是某些逻辑本身不包含任何数据，他可能就是一个对多种逻辑进行计算的一个组合逻辑，但是这种应该很少才对，毕竟大部分System应该关心的是自己，其他的是作为查询条件），这样仍然可以将System隐藏在Component中，但是System自身能够指定引用的其他Component作为查询条件，也即是在定义ArchType，可以在System的顶部使用类似。

XXXComponent a;

BBBComponent b;

然后在正文中就可以引用这些对象，解释器会自动将该变量从Entity中进行查询，并且检查如果Entity不包含该Component的时候进行自动添加，当然也可以检查冗余，即如果没有任何使用则不需要添加

但是Component的版本号在哪里设置，使用明空间加版本号？

BBBBComponent(reality:name,1.2.3)

怎么默认指定？

开发者肯定是先下载了一个包含Component的包再进行编程，但是仍然可能有冲突，所以最好是需要明确指出，而不是自动分配，因为总有一个地方需要明确指出，使用单独的配置文件看起来并不是一个很好的方案

Python嵌入 (Embed in Python)。Python极其易于学习并且被广泛采用。Taichi的前端语法是Python的子集，这使得任何一个Python程序员都能够轻易地学习、使用Taichi。我们使用Python AST灵活的自省 (inspection) 机制来把Python的AST转化为Taichi的AST，随后进入我们自己的编译和运行时系统。将Taichi的前端嵌入进Python有如下好处：

- 容易运行。嵌入在解释性的Python语言而不是编译性的语言中，大大方便了Taichi程序的运行，因为母体语言的预先编译 (ahead-of-time compilation) 不再需要。

- 容易重用已有的Python基础设施并与其交互，包括IDE (PyCharm等)、包管理器 (pip)、已有的Python包 (如matplotlib、numpy、torch等)。

即时 (Just-in-time, JIT) 编译。JIT不但提供了极强的编程灵活性，还延迟了“编译期常量”的需求。比如，在物理模拟器中，时间步长 Δt 通常被实现成运行时变量，而使用JIT的时候则可以被处理成编译期常量。这允许编译器进行更多的优化，如常量折叠 (constant folding)。同时，Taichi支持模板元编程，伴随着JIT的懒惰编译技术大量节省了不必要的编译时间。另外，对于无法运行Python的环境，如移动端设备，我们也提供提前编译 (Ahead of time, AOT) 相关设施

对大众用户、或者偏美术、艺术类用户，最好的脚本语言是什么？图形化的吗？他应该具有两个特征：

- 对一般用户友好
- 适合数据驱动
- 适合DSL编译处理

面向数据编程：

传统的游戏开发是面向事件编程的，例如我们写的所有逻辑几乎都是在每一个frame的某个事件中发生的事情，例如在Unreal的蓝图中，它的起点也是针对某个事件，这有两个缺点：

- 事件的粒度，游戏逻辑中，几乎主要逻辑都是事件驱动，因此详细的事件非常多，所以大部分都是开发者自定义事件，这些事件由开发者自定义的状态机来进行管理，Framework层只有几个基本的游戏生命周期事件，开发者自定义事件之间没有标准，或者非常复杂，不方便维护
- 事件与逻辑不一致，像UE的蓝图是基于事件编程，如果我们要把这种能力开发给普通用户，这种没有标准的事件定义并不适合让用户去学习，这些事件通常也没有必然的逻辑联系，不容易理解，例如一个逻辑可能对应多个事件，显然用户需要了解的是逻辑，而非事件
- 数据代表的是逻辑，所以平台大部分都在设计这种数据，用户也便于理解，同时它代表的不是最小逻辑，而是逻辑模块，所以他将逻辑内部的实现细节（众多的碎片事件）进行隐藏
- 数据也代表接口，数据的标准有助于构建结构化表达，是逻辑更清晰，以一种更清晰的方式组织

面向数据编程，就像Houdini 中面向Node编程一样，一段Python 代码是受限的，它的输入输出是node，同时又可以使用到python 本身的任意语言特性和能力；与此类似，面向数据编程也是针对一个特定的“数据”编写脚本，用户编写的是System，他的输入输出是数据Component

但跟传统的ECS架构不一样的是，它的Component 和System之间不是一一对应的，甚至不是自动挂载的，这个机制非常重要，一个Component 理论上可以被多个System消费，当然通常一个特定的Entity实例只有一对Component 与System的组合，因为理论上对一个实例的一个类型的数据，应该只有一个操作逻辑，除非多个System之间逻辑不一样，他们可能偏重数据的某一部份，但是这种情况通常是数据的粒度太大了，可以再进一步细分，当然可能有一些特例需要维持较大的数据。

所以这就使得app包里可能包含一个数据定义的多种逻辑，所以我们不能使用自动挂载，而需要依赖于版本制定，在开发或生成内容的时候，我们将一个System添加至一个Entity，他就生成对应的System引用和版本号，加载的时候是根据这个进行逻辑挂载，当然一个独立的应用本身在顶层也会记录所有引用的Component和System，这样便于预加载

比如如果数据是涉及动画，则数据包含动画时间和当前frame number 等动画信息

把数据当作一个类似Houdini 里面的可视节点，System是另一种类型的节点，拖动任意一个节点到一个Entity会自动加载变量，以及相应的组件版本依赖关系

Creation Script

它的语法部分接近Lua，但是因为所有Table全部由Table Engine接管，脚本中只有索引和基本类型，不存在堆中分配的对象，所以不需要GC

脚本只有简单的计算和函数形式，所有操作对象统一，函数只有简单的数值操作，对象都由底层分配

脚本负责函数内的栈上操作

底层负责堆上内存分配

这得益于统一的数据结构和内存管理

2.2.1 全局变量表（符号表）

符号表的核心意义在于，让不同开发者开发的组件之间可以相互交互，因为如果不是这样，那么我们整个应用程序就只能依赖于单个开发者开发的组件，或者每个开发者开发的组件只能独立工作而不能相互交互（在传统的应用开发流程中，开发者通过变量赋值和引用手动建立了来自两个不同开发者开发的组件）

符号表隐含的逻辑是，组件之间的交互涉及的变量应该足够少，大部分应该是内部状态，例如COC中的Cannon：

```
Damage per second: 11
Hitpoints      : 650/650

Range          : 4-11 tiles
Damage type    : Area Splash
Targets        : Ground
Favorite target: Any
```



符号表使我们更加小心地定义我们的变量，使得不那么随意，符号表的全局通用性地位使我们更好地抽象逻辑，确保设置确实的反应逻辑状态的变量。

此外，符号表是一种很好的机制，使得我们很容易将一个组件的内部临时变量和对外表现特征的重要状态变量区分出来，逻辑更好清晰。使得代码逻辑更清晰，他人阅读更容易理解。

```
#version 1.0.0
namespace Global{
    iHP           //表示血量
    iHealth       //
    iCoin         //
    vfPosition
    sLabelName   //用于UI显示名称
}
```

当然符号表更重要的价值是，实现组件之间相互调用的隐式参数传递，组件之间的依赖通常是比较少的变量，在OOP中这些变量即是两个类之间方法调用的参数，如果我们不区分公共和私有变量，则只要两个组件之间有依赖，就需要加载所有的数据，而实际上有些数据是不必要的，因此它有利于提升性能，只加载每个组件需要的数据。

此外，设置out变量的组件，对应的属性数据会被自动填充，因此避免了手动的变量定义、初始化和参数的传递，这是隐式函数调用的核心机制。当然，开发者应该保证一个全局变量应该同时具有消费者和逻辑处理，否则这个计算可能没有用处，这个通过设计来规避，编译器可以协助提示，因为一个正常的设计肯定是两者都有的；当然也有可能一个组件既充当生产者也充当消费者。

这是最核心的机制，因为如果不这样，要想在两个组件之间进行交互，就必须定义一种协议，这种协议通常就是传统程序语言中的引用机制，因为这样才能保证运行时变量可以识别，这就需要在B中定义公共变量，然后A组件需要引用B组件，这就形成耦合，但是不引用，语言本身的机制没法保证哪怕是同一个名字的变量执行同一个地址。

因此，通过符号表，整个事情变得简化：

- 共享变量只需要定义相同的名字，而不需要引用，这个相同的名字由符号表保证，而不是字符串，因为字符串又设计对应的解析，带来解释负担。而传统编译器中的符号表天生就是用来将多个名字相同的符号指向同一个内存地址。
- 避免了通用语言中只能通过引用来实现参数依赖和传递，而这种依赖是导致没法简化编程甚至无代码编程的根本原因。

当然，符号表机制不适合通用语言，它是游戏这种Update机制相结合才能发挥作用，因为两个Update之间的时间很小，有可能通过帧之间的变量共享来实现函数调用和参数传递。否则这种机制就不生效。

此外，这种机制必须配合解释器或者编译器工作，比如单纯集成Lua或者C#是不够的。

2.2.1.1 动态的符号表

在程序中，符号表应该是动态的，以提升解释的性能。因为符号可能会非常多。

首先在创建内容的时候，根据选择的组件计算所有用到的符号表，然后动态生成一个符号表，这可以是云端的一个接口。

当需要修改实体对象时，重新生成新的符号表。

2.2.1.2 符号表

所有交互都来源于符号表

符号表基本上就是为了交互而生

符号表引用就是为了避免直接引用，因为直接引用引入了代码编程思想，限制了组合的能力，除非组件之间完全独立无交互，只要交互通过直接引用就耦合了类型，符号表这是实现类型解耦的关键

符号表需要是公共资源的形式管理，用户可以下载很多开发者定义和上传的符号表，这些符号表由开发者上传，比较有名的符号表可能会得到很多其他开发者的支持和兼容，从而实现跨开发者之间的协作

多个开发者定义的符号表也可能有重复，需要统一的机制，这可以通过：

- 强化全局表，把开发者定义的符号表发展为全局表
- 全局表分类
- 建立表之间的映射和关联，这比较复杂

所有的类型引用都通过数据查找

但是数据不能完全代表类型

某种程度上，数据的组合才代表类型

ECS的最大问题可能是无法区分类型，只有定义字符串或者枚举数值，tag之类的？这些类型由变量的值而不是符号本身决定，这些值可以由开发者或者用户控制

但是tag需要是针对组件，而不是实体

- 实体可以任意复杂，他不可能具有单一类型
- 类型可能被多种组件访问，内存无法控制

定义多个标签，每个组件都包含自己的标签，或者跨组件之间的数据必须是结构体符号，这样结构体本身就包含了类型

这可能导致数据属性重复的结构体，但这没关系，本身数据就是有不同意义的，即使数据结构差不多

考虑只针对某种类型的怪物的血量造成伤害

这个类型标志应该是跟HP关联在一起，还是跟怪物实体呢？

需要一个万能标签，每个组件一个，但是名字相同，编译器保证重复问题

原子“数据”

一个组件的数据或者数据列表应该是有意义的，这个数据的组合应该充当类型，表示一类功能对应的数据，比如HP和Health 通常是组合的，所以我们就不应该单独使用HP，如果有多种类型都需要使用HP，这就需要定义一个结构体

当然这没问题，符号表的意义并不是全部打散成独立的符号，而是为了避免类型引用，但有以下问题：

- 理论上可以为同一实体添加多个HP，这需要开发者用户来避免

所以类型看起来更多应该是由组件的数据合作隐式定义，而不是由符号表来定义，符号表应该足够松散，这样来保证最大的灵活性

组件定义的TAG变量不来源于符号表，但是和符号表类似的定义，编译器特殊处理

或者TAG数组组件本身，是一个用来取代引用组件类型名称本身的一个标志

2.2.1.3 符号表定义标准

尽管从概念上，符号表表示的是一个变量属性，或者是变量的定义，它的目的看起来就是为解耦变量的声明与引用服务。

但是从系统的需求上看，一个符号表应该是包含一个完整的对某一些事情或某一类逻辑的一个完整数据结构，因此它定义的是一个标准。参见2.7节。

2.2.1.4 版本与兼容性

- 修改名称：可以做一个映射，编译器直接处理，甚至组件可以一键升级，代码替换最新名字
- 删除符号：如果发现缺失相关符号，相关的组件不再被解释/编译仅程序，并提示用户组件过期，需要更新；这部分通知通过Reality World进行管理
- 新增符号：会通知开发者，建议开发者支持

2.2.2 Entity

ID + TAG

每个实体有个隐藏private变量：TAG和TARGET用来定义组件本身的标识符，以及用作查询条件的标志符

在Entity上增加TAG，因此充当一个Entity的类型区分，创作者可以在Creation中创建一个Layer表对其进行管理，例如可以统一命名等。

对于System的TARGET，它指向Entity的TAG属性，虽然System对于一个Creation只有一个实例，但是System本身只包含方法，System的状态数据（private私有变量）是存储在每个Entity内部的，它具有多个实例。因此对于TARGET来说，每个Entity的值都不一样，因此Entity可以处理不同的类型目标Entity。

2.2.3 三种变量类型

```
#version 1.0
#order 5

namespace SomeComponent{
    query Global.HP as hp
    query Global.HP[] as hps

    public Global.HP as hp

    private fTemp = 100.
    private TARGET = 20
}
```

三种类型变量说明：

- private 是每个组件的私有变量，只有该组件内部可以访问，这可以通过直接声明实现，因为没有从全局符号表引用声明，所以其他组件没有可能对其进行访问；private是实体的实例变量
- public 是每个组件的公共变量，可以被实体内跨组件访问，也可以被其他实体访问；public也是实体的实例变量；通常public变量由使用该变量的组件定义，可以多个组

件中都重复定义，会被映射为同一个变量

- query不是组件本身的变量，它是指向外部变量的引用，它只能访问那些定义为public的实体

2.2.4 数据抽象：共享变量与解耦

变量的声明与程序执行指令的耦合，是现代计算架构最大的限制，这可能有2种原因：

- 现代编程语言主要是还是基于硬件计算架构的，它上面所做的抽象，更多是围绕用开发者（人的）的角度怎么去理解和组织机器代码的结构，而没有围绕人类理解事物真正的逻辑去进行调整。
- 现代的应用程序架构都是围绕单个独立的应用程序设计，即使有多个应用程序之间需要交互的需求，这些少量且相对比较固定的需求可以使用一些标准的形式解决，例如HTTP协议。

在未来的元宇宙时代，我们需要一些全新的编程架构，所以最重要的是：

- 我们需要在硬件架构和应用架构之间，建立起一个数据抽象架构，将传统硬件计算架构的限制隐藏起来，并且上层应用架构的能力可以更大的释放。

所以，RealityIS的核心逻辑，是通过对编译过程的改造，在操作系统和应用程序之间建立一个新的转换层，是能够将基于数据抽象的程序架构，转换为传统基于硬件架构的程序结构。

2.2.4.1 共享变量的声明

```
#version 1.0.0
#order    1000

namespace SomeComponent{
    public Global.iHP as iHP = 5
    public Global.vfPosition = (1000.0, 234.5, 400.8)
    public Global.sLabelName = "Super Man"

    private fTemp = 500.6

    fn Update(fTime){
        fTemp = fTemp * fTime
        iHP = fTemp
    }
}
```

由于每个组件定义变量的顺序可能不一样，所以编译器要进行排序。

此外，由于这里使用全局符号名与局部简写名字的区别，在解释器中，需要去除本地变量名称，使用统一的全局名称，因为不同组件对于同一个全局变量，可能使用不同的局部变量简写。

对于需要处理其他物体的组件，定义一个数组的形式，解释器自动查找：

```
#version 1.0.0
#order 1000

namespace HandleOtherComponent{
    outer Global.iHP[]
    outer Global.vPosition[]

    out vPosition as position

    fn Upadte{
        for i in iHP{
            if(distance(vPosition[i]-position)<10){
                iHP[i]=-1
                vPosition[i]=(10,10,10)
            }
        }
    }
}
```

上述的类型用于由一些特定对象发起的行为，如果是全局组件，全局组件每个场景只有一个Entity可以拥有

```
#version 1.0.0
#order 10

namespace PhysicalComponent{
    unique Global.Collider[] clliders
}
```

定义了global的属性解释器会解释唯一性，这也意味着有些组件是没有方法的，但没关系，从用户角度来说它仍然是一个功能，例如：

```
#version 1.0.0
#order 100

namespace ClidderComponent{
    public Global.Collider
    public TAG = 100
}
```

也就是组件分为三类：

- 一类对自身进行处理
- 一类对其他对象进行修改，这种情况有一个发起的对象
- 一类是没有特定发起的对象，它是对于某些类型的通用行为，例如物理引擎

2.2.4.2 共享变量的初始化

生产者和消费者都可以设置初始值，其中一个修改会复写另一个，因为对于共享变量，一个实体只有一份数据。

初始化的方式：

- 编辑器中修改，每次选择一个组件，设置后，下次其他组件加载的也是修改后的值。
- 在运行时，用户选择一个物体，对其进行修改，运行态用户只看得见一份共享变量，组件只有私有变量可以单独设置。
- 通过代码的形式加载组件，按时间覆写数值。但一般情况下，组件的初始值应该由其定义默认设置，否则容易造成迷惑。一旦实例运行一段时间之后，值应该由持久化的数据进行加载初始化。

2.2.4.3 函数参数解耦

- 每个组件只需要对感兴趣的数据进行交互
- 游戏特有的Update机制保证两个对同样数据感兴趣的数据都能够按顺序被执行

剩下只要我们保证两个组件之间的逻辑顺序，它们之间就可以以间接的方式实现交互。在这个过程中，共同感兴趣的参数充当了函数参数，同时不需要引用其他组件进行调用。

这是一种全新的机制，只有这样，才能实现两个独立组件之间的交互，而且，除了对函数调用进行解耦，它还有以下三个好处：

- 每个组件只需要处理自己感兴趣的数据，不需要去关心跟其他组件之间的交互，例如

就不会出现面向对象编程中常见的你调我还是我调用你的问题，这让我们将编程思想回归到本质，每个函数都只是处理自己感兴趣的数据，并输出相关的数据，但这个函数式编程还不太一样。现实生活中，大部分功能都是在处理数据，这种思维理解起来更简洁。

- 简化并发相关的问题，大部分并发相关的问题都是由于面向对象编程中的对象相互调用导致的。不仅不利于并行计算，而且容易导致各种并发问题，例如死锁，资源抢占等等，因为不可预期的多个对象会访问同一个对象。
- 还能减少因为变化导致的重构、重新发布程序等问题。当一个函数参数发生变化时，必须要修改调用者的代码，它可能使得其他组件不再可用或者崩溃，因而使得其他组件不太愿意更新到最新引用，从而导致系统进化很慢。函数解耦可以使得一个组件的修改至少并不会影响程序的运行。当然如果逻辑发生了较大的变化，程序的计算结果可能逻辑上是不及预期的。这种要有好的机制保证及时更新，但这种机制是系统更上一层的机制，在系统层，我们首先应该保证平台运行的稳定性。

2.2.4.4 赋值与解耦

传统的赋值语句本身就包含了耦合，它由一个读取语句和一个写入语句构成，虽然从逻辑上来说没问题，其中包含安全问题

- 读取语句引入耦合，原本我只想给资源增加一定的增加，我并不需要关心总量
- 写入是重写，覆盖操作，具有破坏性

尽管程序中有`+=`操作，但是由于它与写操作是同一权限，能够执行`+=`操作的变量意味着可以执行写操作，因此仍然具有破坏性

需要将原值读取出来，进行一定的计算，例如加减计算，然后进行赋值，这使得原本对值进行增量的操作逻辑，很容易写到实体外部，即增量来源的地方，这种操作引入了不必要的耦合，因为增量实体根本不需要包含这样的逻辑，但是如果要隔离权限，开发者必须得写一个类似`Add(delta)`的方法

Machinations 的资源流的一个重要基础，它将两个实体之间的联系，即经济的流动，都看成是增量的，这样经济之间的流动过程中，各个实体就只需要关心自己的事情，然后输入输出一个增量就可以，这个增量本身对两个实体没有任何耦合

生活中大部分其实就是这样的流程，想象一下，你要取100元钱，银行直接给你100，而不是让你让银行先把所有钱给你，你拿掉100，再把剩下的钱存回去，如果你在数钱的时候被旁边的人抢走了怎么办呢？

Creation Script 中，组件之间（即对外部变量）的修改支持：

- 增量式修改
- 并且一个组件之间多个修改保证原子性

这使得每个组件只专注于构造自己的增量，大大减少耦合，并且不需要编写复杂的方法来实现增量修改

如果实在是需要依赖于总量进行计算的，这种少数情况可以先读取总量，计算合适的增量，再对其进行修改

默认增量与默认赋值相比，虽然能实现的功能都没有差异，但是它避免了大部分情况下的耦合，毕竟大部分操作是增量，而且在ECS中在Component中不适合写方法

编译器也可以进行优化，系统生成更优化的代码，当然这个功能本身不是为代码优化服务的，它是为用户开发效率服务的，能够简化逻辑关系，让用户逻辑更清晰

2.2.5 组件语义化

符号表机制带来的最大价值，是它消除了模块之间的显示调用参数传递，使得一个组件退化为一个功能，而隐藏了传统编程中要使功能运行起来的参数传递，因为这就需要变量的声明和初始化，而变量的定义有涉及类型系统。这就不可避免导致对编程的学习。

当组件抽象为一个功能，它就可以语义化，一个功能用一个语义表示，而语义是所有人可以理解的。一旦用户选择一个语义，对应的功能及其跟其他功能之间的交互就会自动适配和工作，用户最多需要设置一些特定的属性值。

2.2.5.1 语义化与可视化

符号表及其游戏Update机制，联合起来解决了两个函数之间参数隐式传递的问题，使得不需要开发者或者用户显式指定两个模块之间的参数，既简化了逻辑开发，又使得函数或者代码功能的语义化成为可能

语义化以后的程序，形成可以用自然语言描述的结构，有了这个基础，再结合自然语言人工智能和语音方面的进展，可以做到最简化的内容创作

- 自然语言
- 结构化语言

这是两种完全不同的机制，虽然自然语言中本身也是有句子语法结构的，但是自然语言本身并没有编程语言那么高度结构化，例如即使包含语法错误，人与人之间的交流也可能因为相同经历、知识等原因对其进行纠错，从而忽略错误的语法，然而机器执行的结构化语言也不同，除非是基于AI进行学习，但是那样又需要大量的数据学习

可视化图通常由一些：

- 节点和
- 节点之间的连接关系

来表示，节点表示一个流程的功能，而连接关系不仅表示了某种流程顺序，他还表示了流程之间需要遵循的参数约定

既然符号表解决了参数传递，那么剩下只需要显式声明节点的功能定义，则整个图可以构成结构化的描述，同时也是可视化的

由于自然语言不具备精确定义，所以需要定义明确的节点结构

这种结构最好是一种描述语言，他具有基础的语法结构，而基于这些结构进行扩展可以构建具体、复杂的实例

所谓语言结构，即包括它的参数（连接属性），它的功能属性都是明确的

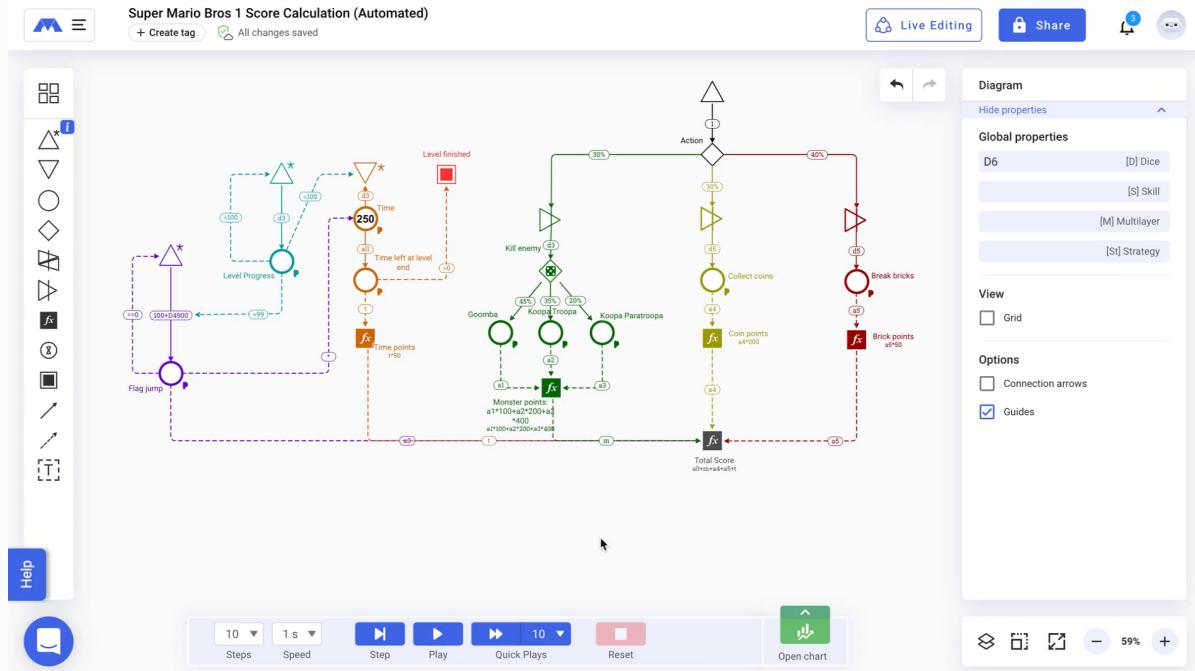
相反，只是任意定义一些没有联系的基类，则无助于构建有价值的可视化图，因为尽管他可以连接一些节点构成图，但是这个图的流程和意义是完全没有意义的，没有价值的

2.2.5.2 能否自动生成Machinations

Machinations 提供了这样一个基础，他能构建起易于理解具有一定逻辑的可视化图，但是它的结构是以经济流动为基础的，有些逻辑不一定有明显的经济因素，所以可以对其进行扩展，有了这样严谨的语言结构，再把组件定义为这样的逻辑单元节点，则节点的语义就明确而且有意义了。

这样可以方便用户涉及Gameplay的玩法

对于每个Machinations中的元素，设计对应的Component，只要从这些Component集成的组件，就自动遵循相应的功能或者接口需求，就可以自动生成设计的可视化图。



或者甚至可以根据Machinations反向生成初始代码。

或者甚至如果Machinations是一种通用编程语言，所有的组件就要求开发者这样去做，这对于用户来讲，就更好控制逻辑行为。

可以对Machinations进行深入分析，加一定的改造。

2.2.5.3 复杂系统的仿真

复杂系统的仿真，对现代工业很多研究直观重要，并且由于真实世界中大部分深刻都机制都深藏于复杂系统中，往往那些直观简单的结构化机制并不足以洞悉这些系统的原理和影响，所以我们需要更好的程序机制来支持复杂系统的仿真。

然后由于复杂系统的特征，现在计算架构并不能很好的处理这类任务，例如：

- 复杂系统往往非常庞大，使用传统面向对象的机制通常无法维护这么庞大的系统交互关系，很难建模
- 复杂系统一般都是实时系统，它并不太适用于传统的应用程序架构，所以现在大部分仿真任务都是使用游戏引擎来执行
- 复杂系统往往还包含人的交互影响，是一个交互式系统

所以目前并没有很好的解决复杂问题仿真的程序机制或者系统软件，RealityIS有机会在这方面提供更好的基础架构和能力。

2.2.6 组件查询

To read or write data, you must first find the data you want to change.

The main way of processing DOTS data is ECS queries. Iterating over all entities that have a matching set of components, is at the center of the ECS architecture.

To identify which entities a system should process, use an [EntityQuery](#). An entity query searches the existing archetypes for those that have the components that match your requirements. You can specify the following component requirements with a query:

```
var queryDescription = new EntityQueryDesc
{
    None = new ComponentType[] { typeof(Frozen) },
    All = new ComponentType[] { typeof(RotationQuaternion),
        ComponentType.ReadOnly<RotationSpeed>() }
};
EntityQuery query = GetEntityQuery(queryDescription);
```

按类型查询组件对用户来讲不太好理解，它让你必须很清楚所有实体中哪些实体具有哪些组件的组合

按类型查找也限制了组件的行为，它只跟类型一致，而实际上，一个攻击可能只针对某些类型的怪物，这些怪物的数据属性是相同的，即可以使用相同的组件，但是因为值的区间不同，它们被分为不同的类别，除非你重复定义Component，这些Component有相同的属性，否则你无法区分它们，即Component充当了类型，但是实际上它只是数据，不能完全充当类型。

怎么对于DOTS中的查询，由于符号表的意义是取代类型引用的，所以ECS中Component的类型需要放到符号表中，换句话说，符号表中的名称同时也表示了符号，因此符号表需要是结构体，像shader中的变量一样，这个结构体的名称或者这个结构体本身就代表了组件类型：

例如符号：

```
struct Position{
    float x
    float y
    float z
}
```

组件中的定义：

```
#version 1.0
#order 1

namespace SomeComponent : Component{
    public Global.Position as position

    fn update(float dt){

    }
}
```

外部引用的组件，使用query修饰符，表示这个变量不是该对象自身的变量，而是查询结果，如果有多重查询，需要定义一种联合查询的方式，但是可以先仅考虑单组件查询，Unity间接使用了多种组件的某些特定组合数据来定义一个System，因为单个组件无法确定数据类型

```
#version 1.0
#order 2

namespace AttackComponent:Component{
    query Global.Position[] as positions
    query Global.HP[] as hps

    public Global.HP as hp
    public Global.Position position

    fn update(float dt){
        if input == 'B' {
            for i in positions{
                if(distance(position-positions)<10){
                    positions[i].x -= hp
                }
            }
        }
    }
}
```

```
        }
    }
}
}
```

2.2.6.1 组件组合不是理想的查询方法

在Unity DOTS中，按组件组合查询有两个好处：

- 最大的好处是直接获取数据，因为System需要操作这些数据，所以直接声明这些类型就不会出错
- 然后才是在这个基础上，将组合定义为一种特定的逻辑类型

目前看起来形成这套机制的核心原因是由于前者；但是这种组合本质上不是真正的查询条件，他可能导致一些意想不到的结果。

因为一个Component能被多个System使用，就说明Component本身可以具备多个意义，例如两个不同的System有可能完全按照相反的逻辑去处理，在这种情况下，用户定义一个Component有可能刚好不是某个System期望的方式，然而最终它还是被当作了查询条件。w

2.2.6.2 显式声明

还是需要某种显式声明的类型，哪怕是Layer这样

感兴趣的数据+TAG数组（多个TAG感兴趣）

只需要声明自己感兴趣的数据就行

由于数据无法表达类型，有用户手动设定

可以像Unity一样定义一个Layer列表，方便用户对对象进行归类

Unity的Component组合查询反而不好理解，太复杂，需要记忆更复杂的东西，相比Tag，tag概念足够简单

- 只要能任意查询，就可以满足通用性
- 符号表满足任意组件间通信

- 组件语义化

2.2.6.3 RUST ECS

You can use empty structs to help you identify specific entities. These are known as "marker components". Useful with **query filters**.

```
/// Add this to all menu ui entities to help identify them
#[derive(Component)]
struct MainMenuUI;

/// Marker for hostile game units
#[derive(Component)]
struct Enemy;

/// This will be used to identify the main player entity
#[derive(Component)]
struct Player;
```

2.2.6.4 Labels/Layers

2.2.7 Component + System

Unity将Component和System区分开，主要是为了将代码和数据分开，使得System中不包含任何数据；即System本身也不能有任何实例数据，因为那样相对于它需要处理的Component而言，其中存在了“全局变量”，这引入了实体间的相关性，耦合，并使得很难定义每个Component之间的状态。

但这也带来了不好的后果：

- 使用者需要分别单独添加Component和System
- 并且使用者必须很小心处理它们之间的关系，比如你定义了Component，但是如果你的系统没有添加相应的System，则数据可能没有用处，这些实体没有任何行为定义；反之，如果引入了System，但是场景中根本就没有对应Component对应的实体，则这些System也无法发挥作用。

Creation同时解决了上述两个问题，方法是将两者融为一体，但是通过编译器将属性抽取为Component（当然Unity也存在类似的转换机制）；更进一步，Creation通过public、private和query标志符区分了公共变量、私有变量和引用变量，使逻辑更清晰。

除此之外，Creation和Unity在功能层面是一致的。

2.2.7.1 组件执行顺序

由于多个未知的程序可以对实体执行操作，所有很难保证顺序的正确性，这比Unity DOTS要复杂，后者由单个开发者开发所有组件，能够严格保证逻辑顺序。但是RealityIS中，同一个实体可能包含来自不同开发者定义的顺序，可能会完全冲突的。

如果允许用户自行去调整执行顺序，会一下子对用户提高了很多要求。

但是我们仔细去分析一下一般的情况，如果能够在做一下假设或者约束，那么问题就会简化很多：

- 1，所有对属性只读的组件都放在最后执行，所有对变量只读的组件之间不用关心任何顺序
- 2，所有对属性可写入或者只读的组件之间都顺序无关

或者更进一步，我们假设：

- 3，每个组件只有一个可写入属性

这样整个问题就可以非常简单地被处理，虚拟机动态调整组件的执行顺序，并且不需要用户或者开发者在关心任何顺序相关的事情。

现实中，这三个条件还是比较容易满足。其中对于第3条，这就有点像函数式编程，每个组件是一个函数，该函数只有一个输出值，并且所有输入参数都是只读的。如果每个组件只有一个可写入的属性，那么就很容易区分出所有只读和可写入两大类组件。即是说，第三个假设可以把所有组件分为两类：只读的组件和只写的组件。

然而即便如此，还是剩下两个问题：

- A和B组件对两个属性交叉读写
- 只读或者只写属性之间由依赖关系

对于第一个问题，在对组件进行自动排序时，将无法推算实际的计算顺序。有两种处理方法：

- 一种是遇到冲突时交由用户来指定，但是由于组件的顺序是全局的，而不是根据用户设置来的，例如两个用户可能设置了相反的顺序，所以这种方法不可取，并且它需要用户介入。

- 第二种方法是由开发者来解决冲突

首先，开发者是最了解逻辑的，而且开发者一旦解决好冲突就不需要用户在做不必要并且增加复杂度的设置。可以把所有代码看成一个整体，当开发者在提交代码的时候，并不只是要考虑自己的代码冲突，还要考虑全局代码冲突。两个开发者提交代码总是有时间先后顺序，所以可以对于后提交代码的人，系统抛出所有可能与之冲突的组件。有开发者通过了解冲突组件的功能，并比较自己组件的功能，来设置依赖关系。

当然这种方式也不是完全可靠，例如开发者可以根本没有看对方组件的功能，随便设置了一个顺序，这时候可能导致错误逻辑。一种可能的方法是，对这些有冲突的组件，后面开发者的设置结果会发送给前面开发者，前面开发者可以对执行顺序提出异议或者同意。

对于上述第二个问题，分三个层面：

- 开发者尽量避免编写这种在一帧之内有复杂依赖的组件，比如如果组件之前都完全没有依赖，那就不会存在这个问题
- 另一方面，具有这种复杂顺序的组件，通常对应的是一个开发者内部的逻辑，这时候它自己设置正确的顺序就好
- 对于组件之间的这种可能性，大多数情况下没有意义，可以不用理会，由用户自行发现问题后反馈给开发者协商处理

2.2.7.2 Change-driven update

事件表的意义：

- 通过建立事件队列，延迟到一起执行，消除一定程度的并发，同时能够做到并行计算，因为一些相同类型的事件可能对应相同的组件，即使同一个事件对应不同的组件，通过按组件类型排序和组织，也能学到按顺序并行；
- 将一部分由状态变化导致的分支转化为事件序列，当然实际处理机制可能不一样

数据库重组涉及更新符号表、以及Table的数据重新调整，符号表和Table API一样，需要符号表API

2.2.7.3 ECS

Unity中ECS的system 只有一个实例，它跟Component 之间的联系由System 对Entity的Query定义，但是带来的结果是系统初始化的时候需要独立完成两个操作：

- 设置每个Entity的Component
- 实例化所有系统会使用的System

这种弱关系的一个可能的结果是有些System 可能查询不到任何具有感兴趣的Component , 而且系统没法自动计算, 只能由开发者手动加载

此外, 这种System 对Component 类型的确定性引用, 使得程序很难动态构建新的场景

而且关键是普通用户很难去理解上述两个操作之间的关系, 例如:

- 如果用户给一个Entity 添加了某个Component , 但是他很难比较明确地要去再添加哪些System , 这可能需要类似于给一个对应表, 而这种对应关系可能有很多
- 如果用户按照System 的功能描述添加了某个System , 他又可能忘了要给一些Entity 添加对应的Component , 程序没办法检查这种情况
- 此外, 更严重的, System 的定义和开发必须了解Component 的定义, 这就回到以前的问题: Component 需要协议, 这很复杂

实际上System 和Component 是严格相关的, 他们分开没有意义, 在OOP中它们就是定义在一起的, 在一起可以避免需要做两个独立看起来不相关的操作, 但实际上也是相关的

所以还是需要和在一起, 使用全局符号表, 但这带来一个新的理解上的问题:

System 全局应该只有一个实例, 但是随着Component 一起加到一个对象, 使得看起来每个Entity 对会执行这个方法

但这问题是否也不大:

- 对于内部组件, 他本来也是需要每个对象执行一次, 在ECS中只是我们强调并行, 其中一份代码对多个数据执行, 而隐藏了System 其实对每个实体执行的感觉, 但是在内部其实是循环, 所以本质上是那个实体都要执行
- 对于特定实体遍历其他实体, 他其实也是那个实体都会执行, 比如一个塔防游戏, 那个炮塔都要遍历行走的怪物, 看看是否在范围内, 如果在范围内就对其造成伤害
- 对于全局的, 那就更好理解, 解释器保证全局只有一个, 那么就是该实体执行了一次

到这里主要的问题在于, 对多个实体的遍历往往带有一个条件形成分支, 并且那个并行的组件A内部都在单独访问所有实体, 又可能造成:

- 并发冲突, 多个线程同时访问同一实体
- 内存局部性破坏, 每个组件分别访问不同的数据

针对上述问题, 解释器要把第二种情况的执行转换为一个一个组件A顺序执行, 在每个顺序执行的组件内对感兴趣的数组进行并发计算

或者更好的方案，对所有感兴趣的实体数组，并发地安顺执行组件A对应的实例

这就是解释器的好处

当然，对于第二种情况，由于System看起来可能被构造多个实例，所以需要解释器只构造一个实例，并记住哪些组件引用了这个实例，也就是在Unity中为了支持Entity的Query，他也需要实现一个数据库来记录这些引用关系，即使针对第一种和第三种组件也是一样的

2.2.7.4 ECS参数

基于Component组合的方式改变了传统编程模型中的参数传递机制，传统的编程模型都需要参数传递，不管是：

- 通用编程语言
- 可视化编程语言
- 声明式编程
- HOUIN 程序化编程

一个System 处理数据，这些数据可以分为：

- 内部定义数据，在对应的Component 中，内部定义通常用来保存帧与帧之间的内部状态
- 外部引用数据，定义一个全局较大的公共属性名称，每个名称有特定含义，仅有引用的名称才会动态组合成所有的数据，这些机制靠编译器来处理，全局属性表之间可能有依赖，编译器自动引入；如果一个插件开发者或者Creation开发者编写的组件需要跨系统引用，需要定义私有名称列表

Global Name

Private Name

这样一来，Table Engine 所做的大部分操作就是进行数组的生成、初始化、复制、排序、修改、SOA等，这部分通过原生的高效实现，通常Table是动态生成的，不知道维度和长度等信息，这就是Table要做的事情

Global Name中的每个属性名称都是具有语义的，相当于宾语，而每个System引用的Name都可以追踪，因此可以归纳出System大致的语义结构：

在什么条件下做什么事情，条件通常是Global Name中某个属性的值

条件 主语 谓语动词 宾语

如果\$NAME<5 System 由System开发者填写，可以多个 Global Name

数据很少是只有一个System 单独消费，通常是多个System 会共享一些component 数据，例如物理引擎组件、动画系统或其他组件会修改位置，而渲染组件会使用这个组件进行渲染

一种常用的模式是：

- 一个或多个组件对某个数据进行修改，通常表现为读写
- 通常一个或者少数几个组件对数据进行消费，通常表现为只读，并且使用目的通常是为了给用户反馈，或者写入到系统或者进行数据存档

由于多个组件会访问数据，因此为了避免数据重复定义，有必要定义一个比较大的标准数据和对应属性名称：

- 其他系统通过引用使用
- 每个系统仅使用部分数据属性，通过显示声明引用

2.2.7.5 组件之间的通信

两种机制：

- 事件，参见2.2.7.2节，本质上是状态处理
- 直接调用，就是正常方法

2.2.8 用户间通信

用户间通信的情况或类型有几下几种：

- 读取和感知，这是最基本权限，让别人可以了解一些你的属性，状态，甚至性格，以及可以怎样与你进行交互
- 交易，所有用户之间的涉及修改的操作都是一种交易，你必须消耗某种类型的资源，另一方获得某种类型的资源

暂时不开放那种会对其他人造成破坏性的功能，例如攻击别人，也就是Reality World没有暴力，没有坏人，没有破坏。

但你可能会因为经验不善而破产，例如地皮是需要租金的，因此你必须赚钱，否则你就没有收入。

当然最好的机制是你的创造力越强，付出的时间越多，收入越高，相比传统的模拟经营游戏主要依赖于时间，这里我们更强调创造力，其实也是现实世界的能力。

2.2.8.1 系统机制

但现实世界有一些全局行为，例如天气对全体城市人员的影响，例如政府的政策等，这些后续再考虑。早期是一个单纯的交易系统。

2.2.8.2 组件安全

不能对其他人的Creation或者实体进行删除或者修改，理论上，对其他人的数据只能读取，所有的修改操作均是交易

- 私有
- 好友之间
- 所有用户（包括陌生人）

2.2.8.3 权限控制

每个公共属性，一般来讲，应该只有一个核心定义，其他的均是对它的引用，就像传统的编程语言中，对象只声明一次，因此它的权限也在那里被定义。

在Creation Script中，由于变量直接变成符号表，因此定义的概念被模糊，符号表之后的引用正确被保证，但是符号表的来源确实不清晰的，因此无法控制变量只能在一个地方被定义。甚至没法区分是谁“定义了”变量。比如，如果两个组件同时定义了一个属性，但分别使用了不同的权限控制，则可能产生歧义：

```
namespace ComponentA {
    public readonly Global.HP as hp
}

namespace ComponentB {
    public readwrite Global.HP as hp
}
```

当上述两个组件被添加到同一个实体了，权限将发生歧义。

实际上，这里由代码开发者来定义权限是不合适的，在传统的游戏开发中，为什么开发者可以定义权限，是因为开发者即是应用的拥有者，或者说开发者是按照拥有所属者的旨意或意愿进行设置的，所以不管怎样，**开发者和应用Owner是同一人**。

随着UGC或用户创作平台的兴起，这种身份的统一性发生了变化：开发者和应用Owner可能是独立不同的人，在这种情况下权限完全由开发者指定是不合适的。

但是，另外一些纯计算的数据属性，只有代码内部才会用到，并且用户不会关心，这些数据显然是应该由开发者控制的，而传统的编程语言中并没有区分这两种变量的权限：逻辑变量和公共变量或者用户变量。尽管开发者可以定义public和private权限，但是这种定义仍然有两个问题：

- 首先它的职责就不是为了区分用户控制与开发中间的区别，因此开发者往往不会有这样的意识
- 开发者定义的权限和用户Owner需求之间可能是有冲突的，因为开发者把权限设定之后就不能更改了，而实际上用户权限是可以发生修改的
- 两者之间的定义没有很强的约束性，例如理论上开发者将所有属性都定义为public都是可以的，而Creation Script保证public属性必须来源于符号表，这就是使得开发者会小心地区分。

Creation Script完美的区分了这两者的定义，并且可以实现用户的控制。

实际生活中，我们所属的东西是资产拥有者可以随便修改的，例如一个图书馆，用户可以设置它是否可以被访问，可以开启和关闭。传统的做法是把这些属性映射到一个数据表，然后由另一个程序去读取并修改这个数据表。这个流程非常复杂，增加了程序的复杂性，而Creation Script相当于是直接“修改程序”。

直接控制数据，甚至直接控制代码，是我们这个世界本来的形式。

2.2.9 数据与存档

提供数据配置表，以及在脚本中访问数据的机制。

数据存储都自动发生，所以用户实体配置的数据都需要存档，但不需要用户指定。

2.2.9.1 数据配置



2.2.9.2 存档

将玩家的进度数据存档，需要存储至云端

2.2.10 通用性

2.2.10.1 独立类

独立的类定义和ECS是等价的，相当于把类的每个方法拆为一个组件，类的实例变量使用全局符号表共享，这样每个组件都可以读写；而Creation Script会区分组件内部和外部变量，因此把一些只有方法内部会使用的变量设置为私有变量，逻辑更加清晰；而传统OOP中，每个类拥有复杂的变量--因此复杂的状态，这些状态有些是表征实例级别状态的，而有些则只是内部两次Update之间的一些临时状态。因此这种划分使类结构逻辑更加清晰，我们能够区分和关注那些真正对外表现自身属性的状态变量。

理论上A和B组件可以完全不需要知道对方的存在，但这种完全无关的交互带来的一个后果是组件执行的顺序非常重要

- 编译器自动识别读写顺序，但这通常最多保证读和写之间，但是多个写之间也有可能有依赖，这种情况无法处理，所以还是需要依赖于后一种情况
- 手动标记执行顺序

2.2.10.2 没有返回值的函数调用

- 在A和B组件之间设置公共变量，如：

```
out Global.iHP as iHP
```

- 如果是A调用B，即A需要向B传递参数，则A的order设置小于B，让A先于B执行即可，这样B执行的时候就可以得到A计算生成的参数
- 如果一个OOP方法内部有多个其他类的方法调用，则按顺序设置多个组件的order

总之，将OOP中的方法调用顺序转换为组件order的顺序

2.2.10.3 有返回值

- 如果B是辅助方法，可以设置为Library而不是组件的形式
- 如果B是实例，具有自己的实例变量，这个时候需要小心地将两者的方法调用关系拆分到两帧之间：A首先或者B上一帧输出的结果，进行计算，相当于B的返回值；然后A将参数输出，B执行的时候或者A的参数，并将计算结果存入到对用的输出参数；然后A在下一帧或者B输出的参数进行计算

当然，如果原来的OOP类特别复杂，就需要小心地进行重构，如果一个OOP方法内有两个即以上的实例间函数返回值调用，上述的方法就不行，需要对逻辑进行进一步梳理，比如如果是前后没有依赖独立的方法调用，则可以很好滴拆分

2.2.10.4 继承

继承通过组合实现

2.2.10.5 结构体

由于底层的Creation Table Engine需要保证数据是简单的数组结构，因为不能设置太复杂的数据结构，所以不支持自定义结构体，只支持基本类型和矢量等基本类型，其中Vector通过内部结构进行处理

当然其实组件的数据本身可以认为是一个结构体，如果两个组件之间需要共享多个变量，可以通过定义多个out参数实现，这就相当于传递一个隐式的结构体。

2.2.10.6 控制tick的频率

2.2.11 赋值与解耦

传统的赋值语句本身就包含了耦合，它由一个读取语句和一个写入语句构成，虽然从逻辑上来说没问题，其中包含安全问题

- 读取语句引入耦合，原本我只想给资源增加一定的增加，我并不需要关心总量
- 写入是重写，覆写操作，具有破坏性

尽管程序中有`+=`操作，但是由于它与写操作是同一权限，能够执行`+=`操作的变量意味着可以执行写操作，因此仍然具有破坏性

需要将原值读取出来，进行一定的计算，例如加减计算，然后进行赋值，这使得原本对值进行增量的操作逻辑，很容易写到实体外部，即增量来源的地方，这种操作引入了不必要的耦合，因为增量实体根本不需要包含这样的逻辑，但是如果要隔离权限，开发者必须得写一个类似`Add(delta)`的方法

Machinations 的资源流的一个重要基础，它将两个实体之间的联系，即经济的流动，都看成是增量的，这样经济之间的流动过程中，各个实体就只需要关心自己的事情，然后输入输出一个增量就可以，这个增量本身对两个实体没有任何耦合

生活中大部分其实就是这样的流程，想象一下，你要取100元钱，银行直接给你100，而不是你让银行先把所有钱给你，你拿掉100，再把剩下的钱存回去，如果你在数钱的时候被旁边的人抢走了怎么办呢？

Creation Script 中，组件之间（即对外部变量）的修改支持：

- 增量式修改
- 并且一个组件之间多个修改保证原子性

这使得每个组件只专注于构造自己的增量，大大减少耦合，并且不需要编写复杂的方法来实现增量修改

如果实在是需要依赖于总量进行计算的，这种少数情况可以先读取总量，计算合适的增量，再对其进行修改

默认增量与默认赋值相比，虽然能实现的功能都没有差异，但是它避免了大部分情况下的耦合，毕竟大部分操作是增量，而且在ECS中在Component中不适合写方法

编译器也可以进行优化，系统生成更优化的代码，当然这个功能本身不是为代码优化服务的，它是为用户开发效率服务的，能够简化逻辑关系，让用户逻辑更清晰

2.2.11.1 去除直接赋值

剩下操作只有：

- 读取 .
- 增量运算，`+=`

其中读取操作主要用于：

- 判断资源是否够用，如果不够用的情况下，资源使用方可以方便显示一些提示信息
- 一些依赖于总量的增量运算，例如增加总量的10%

2.2.11.2 拉取还是传入

资源的两种流动模式：

- 流入模式，一般对应于收集资源，此时需要把一定数量的资源传入一个容器，而不是由该容器去拉取，因为它并没有一个拉取源，而是直接对容器执行一个增量计算
- 拉取模式，当我们需要消耗资源已完成某件事情时，通常由完成该事情的实体从资源容器进行拉取，该实体首先对容器执行一个减量计算（当然需要判断容量是否足够），然后执行自己的处理逻辑；

2.2.12 符号泛型

编程语言中实现泛型来针对变化的类型进行自适应或者运行时解析，这些类型通常具有类似的处理流程或操作

但泛型本质上是编程的范畴，进一步，它：

- 一方面是为了节省重复代码
- 一方面是运行时推导类型，但运行时推导类型的需求往往是由于上面一条为了避免代码重复而导致的

- 当然也有单纯是为了支持变化的类型，或者类型作为变量，从这个角度，泛型的定义反而是为了这个服务

类型变量

实体变量

逻辑变量

```
public Global.HP as hp
```

```
public Global.Health as hp
```

将变量的逻辑映射与变量名区分，在保持逻辑不变的情况下，同一个组件可以处理不同的逻辑，这里“逻辑”本身成为了一个变量，用户可以将一个组件作用至不同的资源类型

传统的泛型要实现类似的功能则非常复杂，你需要把所有变量都转化为对象，并定义接口，然而这种接口非常难定义，因为同样的名称可以用做不同的逻辑，并且这种接口是容易变化的

- 但是数据是不会变化的
- 而且游戏中的逻辑通常修改的资源的数量是比较有限的，通常1，2个，或者3个，以上的很少了

一种不需要约定类型的泛型，只要数据结构类型匹配一样，并且这样类型的检查来源于符号表，而不是运行时

只针对简单结构

复杂结构体，里面的引用名称比较复杂，除非像Lua一样，按索引，但是索引又强调了顺序，顺序通常隐藏在逻辑当中，不过这个顺序倒是可以通过组件说明书说明

2.2.13 并发

面向对象的无序相互引用，通常导致并发，而逻辑上他们不一定有并发，而且我们没有办法从逻辑上去控制这种并发的顺序，太复杂，完全无法预料对象之间以什么样的顺序和时机触发并发。

以组件为单位进行组织，能够更好地控制逻辑的顺序，从而能在逻辑上比较简单地避免掉一些不必要的并发

将共享变量和私有变量区分，私有变量不会触发并发，而共享变量因为从符号表引用，从而编译器能够有可能推导出组件对共享变量之间的依赖关系，比如能够把一些相互独立的组件并行执行

以对象为单位，那个对象执行的逻辑太复杂，存在不可预测的分支、跳转等逻辑，那个对象的指针引用可能导致不可预测的指令执行顺序和序列，因而无法很好地使用指令级并行，因此通常只能依赖线程级并行，但不可预测的混乱的对象引用关系将大大地导致并发问题

通过精心将同一组件的数据组织成数组，不光是能够控制逻辑顺序，从而避免一部分并发，同时将对象级别的线程并行转化为指令级并行，进一步，通过对依赖关系的识别能够更好地将多个独立的组件执行线程级并行，这里的核芯是能够通过全局符号表和显式共享关系识别依赖关系

2.2.13.1 过程式编程

2.2.13.2 Erlang及OOP

2.2.13.3 适合游戏程序的并发模型

2.2.14 智能感知

当靠近一个物体，或者使用一个特定的探索命令的时候，如果其本身不具备识别对方的操作，可以动态查询对方的属性，然后动态提示是否需要安装新的操作脚本。

因为每个脚本需要用户手动选择，不可能自动安装所有脚本，但是系统本身是可以感知的。

比如购买一辆车，需要使用特定的购买方式等。

两个目标或价值：

- **动态移除：**当一个场景中并不包含某个脚本可操作的实体时，可以移除一些组件，或者单纯略过这些组件（出于动态管理的复杂度），比如用户角色可能安装非常多的感知和交互组件，但是在某些场景中根本就没有可与之交互的实体

- **动态添加**: 有些类型的实体，例如一个广告物体，通常其本身并不包含全场景的机制，他们只是临时体验
- **智能购买**: 让用户在试体验某个内容的时候，可以直接一键购买

2.2.15 交易与交互

所有用户之间涉及修改数据的的交互都是交易，当然其他的一般不具备破坏性的交互也可以，例如 读取 数据。

2.2.14.1 HelpComponent

在一个未知的开放世界，很多东西都是未知的，当面对一个新的实体时，需要能够随时随地获取教程，说明怎样与之交互以及带来的影响

2.2.15 Components

2.2.15.1 NeuralComponent

2.2.15.2 TagComponent

2.2.15.3 RealityIDComponent

2.2.15.4 HelpComponent

在一个未知的开放世界，很多东西都是未知的，当面对一个新的实体时，需要能够随时随地获取教程，说明怎样与之交互以及带来的影响

2.2.16 最佳实践

2.2.16.1 more granular is better

Bevy has a smart scheduling algorithm that runs your systems in parallel as much as possible. It does that automatically, when your functions don't require conflicting access to the same data. Your game will scale to run on multiple CPU cores "for free"; that is, without requiring extra development effort from you.

To improve the chances for parallelism, you can make your data and code more granular. Split your data into smaller types / `struct`s. Split your logic into multiple smaller systems / functions. Have each system access only the data that is relevant to it. The fewer access conflicts, the faster your game will run.

The general rule of thumb for Bevy performance is: more granular is better.

2.2.16.2 组件顺序

参见2.2.7.1节内容。

2.2.17 关于数据的本质

- 数据是不变的，但是数据结构是变化的，我们可以把数据定义在任何地方，这就导致好像一个游戏的数据是不可空的
- 针对数据的操作与数据或者数据类型应该是解耦的，暂且称之为数据泛型
- AI编译器或者AI计算平台能够大力发展，模块化，很好的优化，是因为AI的数据结构特征很明显，或者抽象得很好
- 把数据抽象出来，才更容易看清逻辑的本质，比如易于管理，例如能够判断哪些逻辑可以并行，不然逻辑隐藏于一团乱麻之中，人和计算机都不容易识别其中的秩序

2.2.17.1 数据泛型

生活中，我们会发现，有些方法，道理或者逻辑，他们对许多不同的数据或事物都是相通的，我们可以把这些方法应用在不同的领域，比如一辆车子，他其实可以装任何东西，但是在程序员，一个对象所能处理数据却往往包含额外的类型信息，使得方法的处理不够通用

只要逻辑上能够保证合理，这个可以由用户确定，那么一个逻辑应该可以作用在具有不同意义的相同数据类型上，只要用户指定好输入输出

一种新型泛型

编程语言中实现泛型来针对变化的类型进行自适应或者运行时解析，这些类型通常具有类似的处理流程或操作

但泛型本质上是编程的范畴，进一步，它：

- 一方面是为了节省重复代码
- 一方面是运行时推导类型，但运行时推导类型的需求往往是由于上面一条为了避免代

码重复而导致的

- 当然也有单纯是为了支持变化的类型，或者类型作为变量，从这个角度，泛型的定义反而是为了这个服务

类型变量

实体变量

逻辑变量

```
public Global.HP as hp
```

```
public Global.Health as hp
```

将变量的逻辑映射与变量名区分，在保持逻辑不变的情况下，同一个组件可以处理不同的逻辑，这里“逻辑”本身成为了一个变量，用户可以将一个组件作用至不同的资源类型

传统的泛型要实现类似的功能则非常复杂，你需要把所有变量都转化为对象，并定义接口，然而这种接口非常难定义，因为同样的名称可以用做不同的逻辑，并且这种接口是容易变化的

- 但是数据是不会变化的
- 而且游戏中的逻辑通常修改的资源的数量是比较有限的，通常1，2个，或者3个，以上的很少了

一种不需要约定类型的泛型，只要数据结构类型匹配一样，并且这样类型的检查来源于符号表，而不是运行时

只针对简单结构

复杂结构体，里面的引用名称比较复杂，除非像Lua一样，按索引，但是索引又强调了顺序，顺序通常隐藏在逻辑当中，不过这个顺序倒是可以通过组件说明书说明

2.2.17.2 接口、协议、参数、数据

接口是用来保证类之间协作的协议，这个协议没问题，它保证相互协作

但是在编程语言中，除了协议，它还多了一种身份，充当类型，类型被用于帮助语言进行检查，保证程序的合法性

但这是编程语言的需求，实际上并不是协议的需求，比如，在现实中，A和B协作，它们都会自己遵循一种第三方标准，A和B之间事前不需要相互协商，它们可以与任意遵循标准的实体之间进行交互，就算A和B最终发现它们遵循的标准版本不一样，但是大多数情况是一样的

这里的特点有两点：

- 不同在于A和B事前完全互不通信
- A和B在大多数情况下都能相互协作
- 标准往往是来自独立的第三方

程序语言中往往需要引入接口声明，这种耦合不光是协议本身，还包含了很多协议外的跟程序相关的东西，例如特定的程序包、签名的顺序，甚至有时候依赖的顺序，更糟糕的是，还必须把这些内容插入到代码中

你必须从那个接口继承，而不仅仅是遵循一个接口协议

即使你有自己的方法做了协议一样的事情，这还不够，你必须把代码移到接口方法、包装一下等等，你的修改代码

在遵循协议之外，引入了一些额外的负担

另一个情况是：协议通常关注数据，你能把这个数据进行处理，比如我买了一种特定的原材料，我拿过来进行加工，然后生产另一种材料，卖给其他厂家，我们约定的协议是原材料的规格和品类，但不是我们加工的方法或者步骤，我的方法和流程随时可以变动，只要我输入和输出的规格不变

在程序接口中，本意也是关注输入输出的数据规格，这本可以仅通过定义数据结构即可，但是程序需要保证运行时对象初始化、变量赋值、变量的合法性等等各种原因，它把输入输出数据和方法放到了一起，这里面也有更重要的原因是实时性和顺序：调用方需要立即获取返回结果

这使得编程语言的协议约定的更像是方法而不是数据，又加上编译器的类型系统等原因，协议被深度耦合在系统中，增加了复杂性

CreationXR仅关注数据及其结构，并且通过游戏特有的Update机制也能保证返回值被立即取得，但是它弱化了对方法相关的依赖，而数据可以通过公共符号表定义，不管是基本类型还是聚合类型

这种解耦大大的简化了程序组织的复杂性、也增加了灵活性，例如可以随时增加感兴趣的数
据，或者执行不同的逻辑，但其他部分完全不受影响

当然带来的一个新的问题是：这种隐式的参数传递导致组件的实际目标并不是很明确，因为
它很有可能做了一些不可控的事情，这种需要对组件功能进行描述，就像一个产品说明书，
他到底做了什么，这样的说明书是普通用户可以理解的，语义化的

2.2.18 游戏程序跟传统程序的区别

2.2.18.1 Update机制

Update是游戏的核心驱动力：

- 它既是形成动态世界的基础
- 同时又由于每个对象都在实时更新自己的状态，因此使得系统之间的解耦变得可能，
即每个子系统只需要处理自己的状态

传统的应用程序只操作业务规则，没有实体化，他们通常面对的是数据，数据结构，这种数
据通常反应的是规则，而不是对象得概念

2.2.18.2 程序大小与空间数据结构

传统应用程序的程序包大小都比较小，例如一个手机app只有几十最多上百M，但是一个游
戏往往都多大几个G，主机游戏甚至几十到上百G。

不光程序包的大小，计算时加载到显存的数据量更是差异巨大，例如应用程序通常只需要加
载少量有关的数据，常驻内存中的数据通常不多，每个业务逻辑相关的数据通常都比较独
立，即使少量单个逻辑需要的数据量比较大，也仅需在计算的时候即使加载就像，传统的应
用程序通常对实时性要求没那么高。

而游戏程序内的数据通常高度关联，且包含空间数据结构，所以往往数据会非常大，且大多
需要常驻内存，使得现代游戏程序的显存往往是不够的。空间数据结构不仅意味着比一般的
数据量要大，而且为了加速计算，通常还需要包含很多冗余的数据和数据结构来达到实时
性。

当未来的虚拟开放大世界需要更大的数据，这些数据可能远远超出单台计算机能够承载的显
存大小。在这种情况下，这样的大世界将很难有效地运行，需要新的技术架构来支撑这种扩
展需求。

2.2.18.3 架构复杂度

2.2.19 状态机、行为树与AI

2.3 CREATION VM

跟引擎高度一体化，不是独立的虚拟机

2.3.1 Creation Table Engine

Table Engine维护一个Database，主要目标：

- 构造和存储所有基于Data-driven的数据，包括Component的数据、事件列表、Hierarchical Level数据、行为树/状态机等结构，所有数据都已数组的形式组织。
- 对Table数据进行新增、删除、修改、排序等操作。这些操作都需要延迟到Component进行后统一进行处理，而不是立即处理。
- 对于Component的数据，由于所有数据混在一起，需要按照ArchType进行组织。并且块元素的大小进行自动计算。

提供一些标准编程案例

虚拟机的设计：

如果数据都是Table类型的格式，那么动态需要的虚拟机可以设计为处理原生类型，所有Table数据的分配和管理都交给系统层，这样脚本需要生成的代码也为“纯代码”，这些字节码对应的操作数地址的分配则为C++层的编译型代码，而不是动态解释

避免了脚本语言复杂的数据地址分配：例如构造虚拟寄存器或者虚拟栈，以及对应为了构建虚拟寄存器而构造的符号表以及符号表的解释映射等

所有的代码，在虚拟机这一层都是转化为对table的某种操作，这些操作都封装为基础的C++层的代码，自动就处理了内存地址的分配

要拆解Table的基本操作，也就是STL中基础Vector的基础数据操作，应该能够提炼出所有可能的基础操作，然后封装为虚拟机层的高级基础指令

这样整个Table Library 就是一个类似STL的库，它封装一些特定的Table的操作类型和操作方法，面向数据驱动的整个架构，既支持原生C++调用，也支持虚拟机基础指令封装，这就好比Lua的某些方法由C实现，只不过这里的C操作是更低层的操作，并且这里的数据由C定义和分配，而不是像Lua一样由Lua分配再传给C，所以这里脚本语言的定义语言处理特殊的变量操作，例如脚本语言中对变量的使用都翻译为对C对象的使用，没有变量复制，没有数据只有指令

如果脚本语言中不含结构声明，或者没有自己的结构体，只有简单变量，那么整个解释过程是不是会快很多

2.3.1.1 数据存储：数组

为了实现每个组件的单独编译，首先组件访问的数据得是独立的，另外这些数据的布局得是固定的。

对于组件需要访问的每个变量，使用指针的方式是非常复杂的，因为这意味着虚拟机需要动态给每个组件中的每个地址赋值，这几乎是不无做到的，因为虚拟机只处理规则的内容，通用性的规则，但是每个组件对数据的访问确实不一样的，除非是解释器或者虚拟机自己分配的内存，但是这里一个实体的组件的数据不是由组件代码自己分配的（你如传统的寄存器寻址，就是解释器或者编译器直接针对代码设置好寄存器地址），而是由Creation Table自己分配的。

为了实现这种分离，组好的方式就是将组件访问的数据放在一个连续的内存地址中，然后：

- 虚拟机只要动态将起始地址传递给组件，这种规则是通用的
- 解释器将每个组件中的寻址转换为基于相对位置的寻址

这有点像虚拟机中执行函数调用，当然这里只采用栈式方法，所有函数需要访问的变量存储在一个栈中，函数按索引对栈中的内存进行访问。

2.3.1.2 无类型定义与垃圾回收

只有组件类型，组件只有数据，没有新的结构体定义，全局符号表中有一定的基础数据结构，但是其他的数据结构均有组件的数据定义隐式决定。

由于类型定义在脚本语言中，并且是按照单个组件的类型进行定义的，即AOS (array of struct) ，两个原因：

- 第一并不利于高性能计算，高性能计算需要是SOA的形式
- 其次，底层虚拟机并不了解组件的数据结构，导致不能动态构造对象；因此需要在脚本语言层面直接构造对象，这样脚本语言就需要实现复杂对象，及其相应的垃圾回收机制；

对于上述两个问题，Unity使用了离线编译的方式，将组件数据的构造转换成了另外某种形式的中间代码；这样使得可以SOA的形式进行数据管理，但是它可能不支持动态更新，或者至少需要动态更新整个编译后的中间代码？

当CTE试图将上述过程放到运行时动态解析时，性能是需要重点考虑的事情：

- 单纯AOS->SOA的开销
- 以及当场景中有大量对象时，这些对象之间的数据关系等等的判断可能会非常消耗时间

所以需要好好划分阶段，并把部分数据是否可以提前计算出来，例如对于一个Creation：

- 首先确定它引用的所有变量及其组合关系，这部分是否可以预计算成某种格式，即是计算ArchType的时间，这些可以在云端下载资源的时候计算好，下载时动态计算。
- 然后运行时首先就可以根据这些关系初始化数组
- 如果涉及用户在端侧编辑数据，则针对每个对象动态修改，但此时应该不会太影响性能。

因为这种AOS->SOA的转换，使得上层脚本语言定义的对象可以在虚拟机上进行初始化，从而减少脚本层面的复杂度，并使得原生语言管理对象性能更高效。而这种转换机制背后的核心因素是两点：

- 解释器动态识别组件定义的数据结构，并从中提取属性及其符号
- 以及底层ECS的机制将这些属性转换为SOA数组，由于整个CTE都是已知基础数据类型的数组，所以间接地不需要关注上层脚本实际的数据结构，每个元素按照索引进行寻址即可。

整个Table中没有任何未知Struct对象，所谓未知即是用户定义的struct。这里面的核心就是解释器动态对变量的内存地址进行重新映射，通过数组索引+基础变量类型，就能计算出正确的索引位置。因为数据在内存中是没有struct的概念的，struct的作用在于帮助编译器或者解释器计算内存中的索引位置。

当然，我们不能支持用户端自定义struct，否则虚拟机无法识别，就需要复杂的机制来识别结构体。由于游戏的特殊性：它能够将所有数据通过ECS的机制转化为Table，所以我们有就会去除掉结构体或者相关的类型解析，变成更简单的索引计算。

当然，为了考虑性能，这里可能需要考虑AOT的机制，即提前将索引的计算转化为中间形式，不然每次要通过虚拟机中的索引映射方法来动态计算索引，性能开支就会比较大。但是由于这个索引是跟运行时的动态相关联的，所以需要在对象的创建/修改/删除等环节进行重新动态计算。

2.3.2 Add、Remove

对数组的操作一般由Component发起，但是不能立即对Table进行修改，因为其他Component正在访问这些数据，这些修改需要延迟到Component和其他操作执行完毕，需要使用诸如缓存队列之类的架构。原则就是：

- 数组结构的修改需要单独处理，不能影响当前操作

2.3.3 Change-driven Update

在每一帧中，每个组件通常做三种类型的事情：

- 不做任何判断，把整个逻辑完整执行一遍，哪怕其中涉及的数据没有任何变化（因此结果也不会有任何变化），这种计算策略是非常浪费的，但是它是管理成本最低的方式。
- 第二种类型包含逻辑判断，因此一部分计算指令集于某些属性值的不同可能不会被执行
- 第三类是包含一些需要跨越多帧执行的逻辑，例如动画，它们通常只执行一段连续的时间，在这段时间里，由于属性数据都在发生变化，所以它需要像第一种方式一样完整执行；但是一旦这段时间结束，它可能就不再需要被执行。

对于第一和第二种类型，理论上它们都可以归结为一种，因为如果所有输入数据都不发生变化，那么理论上结果也不会有变化，因此可以不用执行。理论上第一种情况可以把某些属性数据作为判断条件，然后第一种情况就变为第二种情况。对于这两种情况，也有可能判断条件会包含多个，因此根据ECS的思路可以拆分，至少拆分成一个组件只包含一个判断条件。

如果一个组件只包含一种判断条件，那么就可以把这个判断条件设置为一个观察值，只有这个值发生变化时才需要完整执行整个组件的逻辑。这就是Change-driven update的核心思路。当然处于简单，我们只判断值是否发生变化，而不是检测更具体的条件，例如一个逻辑条件是变量a大于10，那么a由3变成4也是触发逻辑更新。但是将逻辑判断附加到属性值上则会非常复杂。

这对于一些包含复杂计算或者涉及较大数据加载的组件都是非常有益的。而且对于开发者来讲也不算复杂，可能就是每个组件定义一个或者多个状态变量，并知道这几个变量需要检测即可。当然需要开发者去判断是否这些值的变化能完全决定或覆盖整个逻辑计算。

只有修改过的数据才会Update

Bevy allows you to easily detect when data is changed. You can use this to perform actions in response to changes.

One of the main use cases is optimization – avoiding unnecessary work by only doing it if the relevant data has changed. Another use case is triggering special actions to occur on changes, like configuring something or sending the data somewhere.

Filtering

You can make a `query` that only yields entities if specific `components` on them have been modified.

Use `query filters`:

- `Added<T>`
 - : detect new component instances
 - if the component was added to an existing entity
 - if a new entity with the component was spawned
- `Changed<T>`
 - : detect component instances that have been changed
 - triggers when the component is accessed mutably
 - also triggers if the component is newly-added (as per `Added`)

(If you want to react to removals, see the page on [removal detection](#). It works differently and is much trickier to use.)

```
/// Print the stats of friendly players when they change
fn debug_stats_change(
    query: Query<
        // components
        (&Health, &PlayerXp),
        // filters
        (Without<Enemy>, Or<(Changed<Health>,
    Changed<PlayerXp>)>,
    >,
) {
    for (health, xp) in query.iter() {
        eprintln!(
            "hp: {}+{}, xp: {}",
            health.hp, health.extra, xp.0
        );
    }
}

/// detect new enemies and print their health
fn debug_new_hostiles(
    query: Query<(Entity, &Health), Added<Enemy>>,
) {
    for (entity, health) in query.iter() {
        eprintln!("Entity {:?} is now an enemy! HP: {}", entity, health.hp);
    }
}
```

Checking

If you want to access all the entities, as normal, regardless of if they have been modified, but you just want to check the status, you can use the special [ChangeTrackers](#) query parameter.

```
/// Make sprites flash red on frames when the Health changes
fn debug_damage(
```

```

        mut query: Query<(&mut Sprite, ChangeTrackers<Health>)>,
    ) {
        for (mut sprite, tracker) in query.iter_mut() {
            // detect if the Health changed this frame
            if tracker.is_changed() {
                sprite.color = Color::RED;
            } else {
                // extra check so we don't mutate on every frame
                // without changes
                if sprite.color != Color::WHITE {
                    sprite.color = Color::WHITE;
                }
            }
        }
    }
}

```

This is useful for processing all entities, but doing different things depending on if they have been modified.

For `resources`, change detection is provided via methods on the `Res` / `ResMut` system parameters.

```

fn check_res_changed(
    my_res: Res<MyResource>,
) {
    if my_res.is_changed() {
        // do something
    }
}

fn check_res_added(
    // use Option, not to panic if the resource doesn't exist
    yet
    my_res: Option<Res<MyResource>>,
) {
    if let Some(my_res) = my_res {
        // the resource exists

        if my_res.is_added() {
            // it was just added
        }
    }
}

```

```
// do something
}
}
}
```

Note that change detection cannot currently be used to detect **states** changes (via the **state resource**) (bug).

What gets detected

Changed detection is triggered by **DerefMut**. Simply accessing components via a mutable query, without actually performing a **&mut** access, will *not* trigger it.

This makes change detection quite accurate. You can rely on it to optimize your game's performance, or to otherwise trigger things to happen.

Also note that when you mutate a component, Bevy does not track if the new value is actually different from the old value. It will always trigger the change detection. If you want to avoid that, simply check it yourself:

```
fn update_player_xp(
    mut query: Query<&mut PlayerXp>,
) {
    for mut xp in query.iter_mut() {
        let new_xp = maybe_lvl_up(&xp);

        // avoid triggering change detection if the value is
        // the same
        if new_xp != *xp {
            *xp = new_xp;
        }
    }
}
```

Change detection works on a per-system granularity, and is reliable. A system will not detect changes that it made itself, only those done by other systems, and only if it has not seen them before (the changes happened since the last time it ran). If your system only runs sometimes (such as with **states** or **run criteria**), you do *not*

have to worry about missing changes.

Beware of frame delay / 1-frame-lag. This can occur if Bevy runs the detecting system before the changing system. The detecting system will see the change the next time it runs, typically on the next frame update.

If you need to ensure that changes are handled immediately / during the same frame, you can use **explicit system ordering**.

However, when detecting component additions with **Added** (which are typically done using **Commands**), this is not enough; you need **stages**.

2.3.4 编译

由于System是不依赖于数据及数据结构的，它只包含一个相对索引地址，每个System使用的所有数据都可以通过这个相对索引地址进行查找，所以编译器只是计算了每个变量的一个索引地址，通过堆而不是堆栈指针的方式。

因此，每个组件在开发完成之后它的编译工作就结束了。

而在实际运行的时候，用户请求一个实体，云端会根据这个实体配置（对组件的引用），对实体的数据进行组织，它会根据System对数据的使用定义，将这些数据精心组织在Creation Table中，然后再将适当的数组及其索引发送给System的代码进行执行即可。

所以：

- 在编辑器Reality Create中，开发者每写完一个组件（例如一个System）都会进行编译，除非他再次修改组件源代码，否则不需要再重新编译，属于一种AOT的形式。
- 对于用户，它通常直接在Reality World app中进行操作，TA做的事情主要是修改实体的配置数据，当这些数据发生修改之后，这个过程不会涉及代码重新编译，只有Creation Table对数据的内存布局进行重新调整。

所以，尽管整个代码的组织方式看起来很复杂，得益于这种数据分离的机制，编译逻辑相对还是比较简单的。

2.3.5 链接和加载

在传统的编译过程中，因为源代码之间相互引用了类型及内存地址，所以它们需要链接在一起。虽然为了实现如增量更新等，能够避免改动一个问题就需要重新编译整个系统，但是链接过程是省不了的，链接的过程即是把各个源代码中相互引用的部分链接起来。

链接的机制对于大型实时系统的限制如下：

- 增加了启动时的加载时间
- 使得程序规模很难伸缩，因为更大规模的程序意味着更大规模的链接时间
- 同时，如果链接文件增多，很难管理到底要加载那些动态库，如果每个动态库只使用一点信息，那系统内存会导致大量的浪费。

虽然静态类型的语言其链接过程只需要发生一次，但是对于动态语言来讲，这样的链接过程需要在加载的时候执行，这增加了应用程序启动加载时的时间。

所以，为了解决大型系统的动态解释问题，我们必须要能够将程序分成很小的碎片，并且去除相互间的依赖关系，从而彻底去除掉链接这个环境。

具体需要做到几点：

- 源代码之间没有相互类型引用，或者说源代码没有复杂的类型系统，只有基本类型
- 源代码访问的数据地址通过运行时动态分配，即不需要通过编译器实现指定和计算数据地址
- 当然数据的动态分配要足够简单，否则也会增加性能开支，参见Creation Table相关内容
- 源代码要足够碎片化，使得系统可以尽可能加载更少的代码

最终，RealityIS几乎可以完全抛弃动态链接这一部分的计算过程。

2.3.6 组件关系管理

维护一个表，记录所有当前程序中使用的组件，并根据组件中的数据定义，管理实体对象内存数据的布局，组件的执行顺序等事情。

2.4. CREATIONXR

跟手机最大不同是：

- 手具有在三维世界中的位置，具备创建立体视觉物体的基础，不再仅限于平面
- 由于SLAM，人在三维世界的移动也具有3D位置，所以人身的移动也是交互的一种输入
- XR眼镜的屏幕更大，世界不再仅限于手机大小的屏幕尺寸

当然前两者在手机也是可以做到，只是体验没那么好

硬件设备的交互只限于手势识别、定位等基础接口，应用层要定义真正的交互接口：

- 随着手指的移动，生成不同风格的笔刷
- 当用户勾勒一个多边形之后，生成封闭的几何物体
- 当用户将两个多边形拼在一起，自动合并成物体
- 对几何表面的纹理涂鸦、材质编辑，喷绘
- 可能涉及很多物理模拟，这样更加真实

由于手势位置不是绝对精准的，所以snapping 算法很关键……

手势识别很关键

传统PC或者手机二维空间创建3D很大的问题在于，每一个操作都需要一个菜单，或者说每一个功能都是一个菜单，一个3D软件基本就是就是一个菜单的几何，用户需要首先原则一种模式，然后在该模式下原则具体的功能进行操作，菜单可能上百，想象就是堡垒之夜都好多菜单

但当选择菜单以后，实际的原子3D操作并不多，在XR中，这一切都可以通过手势来大大简化，例如：

- 单手五指收拢就是缩小，张开就是放大
- 食指移动就是移动
- 左手掌可以控制一个物体，左手手势充当一些控制模式

2D vs. 3D

传统2D鼠标没有前后深度上的概念，然而：

- 场景是3D的
- 当前的摄像机主要是特定于某些比较近的物体

这意味着，当你需要对某些距离比较远的物体进行编辑时，必须要将摄像机移动到这些地方附近，然而PC上移动摄像机非常麻烦，因为没有3D距离，我们只能借助鼠标或者屏幕上左右上下的概念，这种平面向3D的映射使得只能实现相对于当前位置：

- 左右上下移动
- 或者左右上下旋转

这意味着移动到一个较远的地方会非常麻烦，而3D的交互则不一样，他可以直接将手指触及到的一个点拉到眼前

2.4.1 Unified XR Input

2.4.2 XR Scene Understanding

2.4.3 Data-driven Architecture

数据驱动应该仅关注用户逻辑层，引擎层面的开发还是使用传统的面向对象结构，或者有一些数据驱动，但它不是ECS架构，而是为了便于如跨平台性、渲染管线配置等这样的目的，除此之外，面向对象具有更好的能力。

2.4.4 交互

要想在XR设备上进行交互（包括内容创作），并且面向更大众的用户，必须具有更低交互门槛，传统的互动程序如游戏的操作门槛还是比较高。例如，使用手势触摸等按键控制人物在3D空间中进行行走，使用复杂的按键组合控制角色完成一些复杂的东西，仍然是游戏门槛比较高的其中一个部门。

当然，操控技巧本身被当做游戏机制很核心的一部分，它能带来玩法的乐趣，这无可厚非，但是互动内容背后本身所表达的故事、系统之间交互的机制等仍然才是互动内容的核心，它表达的东西会更多。

所以，和计算架构一样，我们也需要在交互领域做一些基础创新。

2.4.4.1 基于空间的交互

空间交互式互动内容交互的主要形式，包括移动摄像头、移动场景、选择物体、移动物体等等，常用的一些操作方式，如手机上的滑动、点击、双击等手势，以及PC上的键盘和鼠标，或者主机游戏机上的遥感和控制手柄。

大部分互动内容最频繁和最核心的操作是关于Camera的移动，这既可以是移动角色，也可以是移动场景，但不管怎样，几乎都需要一种机制能够控制在整个空间进行操作。这样的手势操作通常比较复杂，尤其对于较远处物体的操作，会随着距离和遮挡等问题变得更复杂。

以角色对参考系移动世界，和以世界为参考系移动角色，这两种操作类似，但是当你需要同时支持这两者时，事情会变得复杂，尤其移动物体会面临更多空间条件。《堡垒之夜》针对此设计了一种统一的架构，在手机模式中，它将物体与角色之间，借助固定的屏幕中央位置保持一种相对关系，因此可以把物体的移动操作与玩家的Camera结合起来。大大简化了这种交互的方式。



尽管这种方式简化了操作，但是如果在XR的环境（包括手机、AR和VR眼镜），由于设备本身具备定位功能，因此它跟真实世界的3维空间关联起来，我们可以借助人在真实3维空间的移动与虚拟Camera结合起来，这使得：

- 虚拟空间是可以相对静止不动的
- 真实空间人的移动充当了虚拟Camera的移动

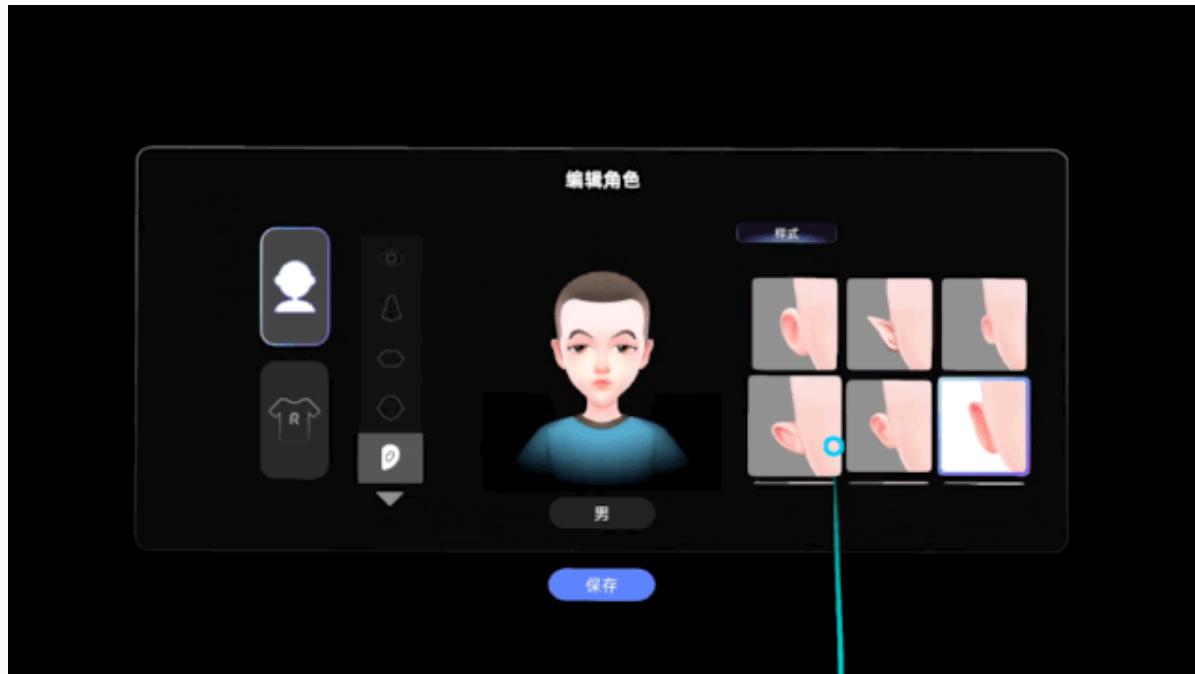
借助上述的方式，我们有望可以进一步简化XR的空间交互门槛。

2.4.4.2 基于语音+AI的交互

大部分传统的互动内容交互都是只涉及到3D的空间交互，但是当这种交互转换到XR设备时，会导致一个新的问题：

- 即原本在手机上只需要点击屏幕选择的方式，也需要变成一种空间交互

这大大增加了门槛，例如下图所示，每一个物品的选择都需要移动虚拟遥感来确定要操作的内容。这在传统的手机或者PC上原本是很简单的事情。



针对此，除了上述的空间变换方法，另一个重要的方面是让所操作的内容尽可能语义化，然后我们就可以借助语音来辅助操作。传统的方式很难使用语音辅助，因为信息都是按结构描述的，信息通过没有语义。我们可能需要对每个元素加个标签之类的来辅助语音。

RealityJS组件的语义特性，使得我们可能在创作内容的时候，尽可能较少对空间交互的依赖，从而使整个创作过程更简单。

2.5 CREATION AI

2.5.1 Semantics-based Creating

基于语义的内容创作

2.5.2 Procedural Content Generation

2.5.3 Intelligent Simulation

2.5.4 Research

行为分析与研究

2.6 CREATION CLOUD

2.6.1 Creation Management

2.6.1.1 CreationID

2、场景到达及时性

游戏中的场景都很大，而且都预设一定的流程和路线，比如：

- 每个玩家都必须从起点，通过前面所有游戏设计师设计的关卡，才能到达某个场景点；
- 即使是静态场景如塔防，三消游戏，模拟经营游戏，它的状态也不一样，你必须从零开始把前面的等级都完成了，才能看到该场景的某个状态
- 即使如世界相对比较静态的开放场景，它的整个世界都很大，你很难让另一个玩家直接定位到某个你指定的地方，他们可能要有一会才能到达，例如一般会分为一些区，玩家可能能进去一个大区一个固定的位置，但是剩下的要独立行走一段时间，并且要知道方向

然而我们需要的是让被分享的玩家能够最及时的、立刻呈现某个兴趣点，并且朋友看到的是同一状态，甚至同一个Camera的位置，这种设计通常是小场景的，独立的，无比较复杂状态的操作

2.6.2 Creation Code Library

代码库是以标准为单位对其进行分类，每个标准会对应无数个实现该标准的组件，但是标准并不包含组件，它只作为用户选择组件或者开发者开发者的一种分类，例如为了实现不同标准之间的交互，某个组件可能使用来自两个标准的符号。

2.6.2.1 标准管理

标准管理比较简单，它就是一个数据结构的定义。

当然围绕标准会有社区和讨论，标准作为一种组件分类和检索依据，可以查询所有与该标准相关的组件。

标准的名称是唯一的。

不同的开发者都可以定义类似的标准，只是你需要去发展自己的标准，例如通过自己开发更丰富的组件，或者邀请别的开发者针对你的标准开组件。

每个标准只包含两个版本，以减少版本管理的复杂度，以及始终保持用户组件更新，参见4.11.3.4节的内容。

2.6.2.2 组件及包管理

组件包的管理包括两个部分：

- 开发者表写组件时，对引用的标准符号进行解释和加载
- 用户对实体添加或者修改组件时，自动加载组件，以及动态修改Creation Table的布局

与传统的包管理方式不同，这里不需要开发者或者用户手动维护版本号，参见2.7.5节的内容。

2.6.3 Multi-player Services

2.6.3.1 Voice Service

Um, and it's true that there are negative people out there sometimes, but the far majority of encounters are positive. Um, and also the far majority of social engagement on fortnight isn't with random strangers. It's not what many to many with millions of people are participating. It's players together with their friends, talking with their friends, kind of as an isolated group, uh, wandering through a much larger outside world. And a lot of the decisions we've made in the game have really contributed to the positivity here. One is that we have voice chat, so you can chat with your friends, but voice chat only works with people in your squad that either you're explicitly friends with and you explicitly joined up with, or you're friends explicitly joined up with.

And epic is con conscientiously making an effort to do this in everything we do. Uh, for example, we're moving to a web RTC based, uh, voice coms framework, um, in Fortnite for voice and text and video chat, so that we can start integrating with other services, you know, other platforms, other stores, other echo systems, other chat clients, um, and have shared social experiences across different game clients. You know, we already have some standards for identity and authentication. We can expand them from there with new standards, for friends and connectivity.

2.6.4 端云协同

现有引擎架构很难做到端云协同，例如大家想到的：

- UI和交互放端侧
- 云端可以共享的在多个用户之间共享

但这些都很难，或者在原有引擎架构下拆分很难。

除此之外，其实还有另外一些协同，例如AOT的预编译等，这种需要软件架构跟自己流程的联合设计。

2.6.4.1 在云端执行脚本

大部分组件都应该在云端执行，尤其考虑到很多逻辑实际上跟用户显示是无关的。

2.6.4.2 Client as a Display

一个核心思路：端侧只需要存储和计算跟显示相关的内容，比如大部分UI和视觉效果相关的内容。其他的逻辑和数据，如果它们虽然是核心数据，但是不会直接显示，也不应该存在于客户端内存中。

所以脚本中要区分哪些是显示组件，哪些数据是跟显示相关的。

对于那些与显示相关的数据，它们在客户端内存中都会有存储的值，但是这些值有可能是服务器计算出来的，如果是服务器写入的值，服务器会自动处理，使得对于端侧来讲，这个值好像就是端侧自己某个逻辑计算的，它随时可以从内存中获取到变量的值。

所以RealityIS要将这一切隐藏起来，使得对于端侧来讲，他像一个虚拟内存，端侧随时都可以获取到这些变量的值。

2.6.4.3 在云端计算需要加载的组件

2.6.4.4 计算实体和组件的关系

2.6.5 并行计算

传统的面向对象很难抽取出小颗粒的计算，所以几乎无法做并行计算，一个游戏必须在一个机器上运行，因为它需要加载所有的数据和代码。而当这个“游戏”是一个无穷的元宇宙世界时，这种计算会变得越来越低效和昂贵。

传统的做法，如果不对数据做精心的管理，最多只能做到三个层面：

- 模块级的多线程
- 流水线级的多线程
- 虚拟化流水线

前者的主要问题是，它的数据仍然没有拆分，所以理论上它可以在一个机器内很好地做多线程，但是当分配到多台机器时，每台机器都要拷贝几乎所有的、相同的数据，使得并行计算的管理难度大大增加。

流水线级的多线程，因为流水线之间的顺序，也会增加管理和调度的难度。

这部分主要针对 GPU，由于 GPU 计算是高度并行的，线程之间依赖比较小，所有理论上可以分区域或者分块进行计算。但这不是绝对的，例如纹理采样，阴影，后处理等技术，通过是需要对纹理进行任意采样的，采用虚拟化流水线这一块有很多问题需要处理。目前看起来仅有类似 Epic Games 的几何裁剪是合理的，保证计算是维持在一个上限，而现代 GPU 计算这样的上限通常问题不大。但缺点是这部分数据管理的开销也不小。

但不管怎样，GPU 的渲染部分通过裁剪，目前看起来是存在比较完善的理论和工程实践了，所以最大的问题是 CPU 的逻辑。

当所有计算处于一个应用程序内，逻辑计算将会非常复杂，所以从这个角度来看，OOP 必然不合适，Unreal Engine 的方式还没有存在较大的问题，是因为它的游戏逻辑部分的规模还不够大。

所以必然要将数据和逻辑拆分成一段一段小的计算单元，不管对于数据还是计算指令，这样就使得可以无限并行化，因此 ECS 类似的数据驱动几乎是未来唯一的解决方案，它在一个程序内部天生地将数据和逻辑区分出来。

2.6.5.1 分布式 Creation Table

当计算和数据能够被划分为小块数据时，一个游戏世界不再被看做是一个不可分割的整体，它的数据和程序都可以被简单地划分为多个独立的数据，因此一个游戏世界很容易被分配到多台服务器上运行。

每个客户端只需要保证知道不同的游戏对象对应的服务器地址即可。

整个并行计算架构变为：

- 渲染在客户端执行，通过很好的几何裁剪保证性能的上限
- 逻辑在云端被很好地分布计算

2.6.6 RPC

一般游戏中

2.7 核心编程思想

2.7.1 避免全局变量

2.7.2 函数式编程

2.7.3 数据驱动

2.7.4 ECS

2.7.5 包及依赖管理

包管理的机制主要是避免用户或者开发者触碰和配置别人的源代码。

但与传统的包管理思想不同， Reality World的主要创新在于不需要用户或者开发者配置版本号之类的事情，这不管对于开发者还是用户来讲，整个流程和思路都简化了很多。

2.7.6 动态解释

在一个开放世界，很多事情都在实时变化，所以传统的编译型平台肯定不再合适。动态解析是必不可少的，只是要做到：

- 局部解析，每个局部组件可以单独解释，而不需要改了一个脚本需要其他人都重新解释代码
- 解释效率要足够高

对于RealityIS，每个组件在发布的时候就编译好了，后续对组件的使用都不会重新对组件进行编译。当然这种情况下，底层的虚拟机需要将组件需要的数据地址计算正确并给到组件源代码。这种需求对于面向对象的方法是不太可行的，但是有了Creation Table将所有数据转换为数组的形式后，组件中的所有地址都是一个相对地址的偏移，所以能够简化这件事情。

为了保证大规模的动态代码执行，整个编译系统必须以一个组件为单位进行，而不能跟其他的代码有任何形式的关联或交叉编译。当然，传统的多个代码链接在一起的过程式因为这些代码之间存在引用，例如编译器为了解释某个类型，或者将变量执行某个内存地址，或者为某个对象分配多大的内存数据，这些导致需要交叉关联。而在RealityIS中，每个组件跟其他组件之间是无关联的，至少组件不需要知道其他组件的任何信息，而即使简洁的关系也是由

虚拟机来决定的，所以对于RealityIS来说，对每个组件执行独立的编译就足够了。

所以，RealityIS是在Runtime的时候，根据用户实体配置实时加载编译好的组件机器码或者字节码，然后根据配置进行实时数据分配，并将实时的数据地址传递给组件指令进行执行。

尽管动态分配组件的数据内存分布会影响运行时性能，但这样的时机只发生在第一次加载程序，以及用户对实体的组件进行修改的时候，因此总体上不会对运行时性能造成太大的影响。

除此之外，由于动态解释的效率比较低，因此又不能将所有的逻辑和数据都使用动态脚本进行解析，那样整个系统的效率将会非常低。理想的状态是能够结合脚本语言的灵活性与原生系统语言的高性能，然后一般的语言机制却很难做到这样，因为数据跟程序逻辑通常是高度耦合的，它通常都是有一台机器同时执行数据的分配和逻辑的解释。比如现在的脚本语言，它们都包含自己独立的虚拟机，使得脚本中的一些都是由该虚拟机管理的。虚拟机本身就是一台能够执行通用计算的机器。即使脚本语言也能够跟自定义的宿主程序进行交互，整个交互的过程却是非常复杂，比如这种复杂的过程肯定不适合普通的用户去配置，而且他也要求暴露一些底层的接口给平台，这样将会带来安全风险。

将组件的数据内存分配交给虚拟机来做，能够简化这个事情，让宿主层来管理和加载数据，将大大提升系统的性能。这样脚本语言省掉很多事情，只专注于唯一需要专注的逻辑计算指令。比如：

- 不用担心复杂数据的分配和寻址
- 不需要垃圾回收管理复杂的数据
- 由于省掉数据分配，以及更简单的组件开发规则（例如不需要定义复杂的面向对象机制），整个解释器也会变得非常简单

所以RealityIS整体上类似AOT，但是它也不是全部AOT，因为它的数据组织部分需要在Runtime执行，这样保证既有比较好的动态性，又有比较好的性能保证。

2.8 大型动态系统

在未来的开放Metaverse中，整个系统会非常庞大，使得不可能使用单独应用程序的思维和架构来管理这样的系统。在这样的系统中，系统内部的一些时时刻刻都在运行，也时时刻刻都在发生变化，无论是程序还是用户内容都是如此。

因此RealityIS应该是一个完全动态的系统，并且这些动态体现在多个层面。

2.8.1 动态编译

首先，程序是最核心的动态内容，而且由于整个世界代码量很大，因此必须完全动态解释。

起码要满足两个方面的需求：

- 每个文件独立编译，不能放到一次性编译
- 避免复杂的链接过程

对于链接过程，又必须做到两点：

- 程序之间不能有引用，因为引用就意味着要加载巨大的代码
- 只加载需要的组件及源代码，而不需要加载大量的不会被执行的代码

当然，参见2.7.6节的内容，为了避免这样大规模程序实时解释的，RealityIS使用了特殊的架构来保证整个程序在解释时的性能问题。

对于源代码的动态解释，它只发生在代码开发的时候，也就是开发者在Reality Create中编写代码的过程中，以及发布组件的时候。同时为了保证支持大规模的代码执行，这种编译只针对单个组件，不能对多个组件进行交叉编译。当然RealityIS也可以避免这样的问题。

除此之外，RealityIS在其他地方，并不需要编译。整个过程相对比较轻量，更多的是动态一些实体与组件关系的管理等工作。

2.8.2 动态创建和修改

在Reality World这样完全开放的世界中，用户可以随时修改内容，而不是只能事先设置好固定的内容。例如用户可以一边玩一边修改场景，这种创建的过程本身也是一种玩法。

由于RealityIS中的所有组件都是编译好的，所以这种修改不涉及到代码的修改，因此动态操作是非常容易的。

当用户修改一个实体的时候，因为整个逻辑都是数据驱动的，因此运行时只涉及到数据的调整。然而，因为RealityIS的Runtime会承担一部分解释以及给指令分配正确内存地址的事情，所以用户的数据调整会导致Creation Table Engine对内存中的数据布局进行调整。当然这种调整直发生在修改的时候，所以总体应该不会影响性能。

2.8.3 动态加载场景

Reality World是一个非常巨大的世界，因此我们不能按照传统的方式一下子加载整个程序，而是只需要动态加载用户需要的数据。这里面可能要涉及很多跟地理位置相关的数据管理。

2.8.4 动态推送更新

当用户A修改了内容，其他与这个内容相关的用户当前的场景必须动态更新。

2.8.5 动态分配服务器

同时，由于整个内容的创建、加载、更新和推送等等，都是动态发生的，任务复杂而且计算量较大，所以需要动态的分配服务器，不能有一些太固定的规则，或者需要人工进行某些配置的工作，而且应该是可以自行伸缩的。

3. REALITY CREATE

设计原则

第一目标是全部程序动态化，任何整个Creation都可以动态下载，所以不用编写C++代码，也就意味着底层必须高度优化，脚本的转换部份也要高度优化，可以去除一些不必要的面向对象属性

3.1 CREATION ID

3.2 CREATION SIMULATION

3.3 UI组件

将UI元素集于Creation Script构建成组件，然后整个编辑器可以集于Creation Script来创建，即整个编辑器当作一个Creation。

UI组件的做法其实可以按照3D Renerer的做法类似，唯一的区别就是Camera不一致，3D的渲染部分肯定也是需要继承到原生C++代码中，UI渲染完全也是类似的思路。

在传统的编辑器中，编辑相关的功能只是存在于编辑器中，不会包含在运行时，这块仍然需要处理，但是至少整个编辑器的构建可以使用统一的架构。

另外有一部分功能是编辑器特有的，包括代码的提示，调试等等功能，这部分在Runtime部分还是需要从虚拟机中拿掉。

3.3.1 Bevy UI

A custom ECS-driven UI framework built specifically for Bevy

- Built directly on top of Bevy's ECS, Renderer, and Scene plugins
- Compose UIs dynamically in code or declaratively using the Bevy Scene format
- Use a familiar "flex box" model to layout your UIs

3.3.2 统一编辑态和运行态

4. REALITY WORLD



核心产品，就是以现实世界的地面平面特征为底板

创作元素、模板、行为组件分类中以建筑类、城市装扮类为核心或者优先，以小世界合成大城市的方式，重新定义我们的世界

- 除了建筑之外，还有许多城市元素，例如广告，交通
- 城市主题结构将不仅仅是建筑，可能非常多奇观创造，非常多元
- 游览城市将是一种很独特的体验
- 要有机制在城市中构造文化

同时整个基础仍然是可以局部独立物体可分享的方式

第一期产品整体会有三种体验

- 大的现实世界
- 独立分享
- 以Code的形式创造平行世界

所有的建筑内容不会是静态的，它会为创作者带来收益，成长或者升级，类似模拟经营的体验，但是这里主要是靠创造的艺术性、文化性等，通过创造的独立性吸引流量，从而形成区域等级中心区等等，较热门的区域会带来更高的收益，形成城市文化（创作本身蕴含着文化），这可能也会形成区域协作，共同定义一些文化，可以类似Everdale机制协作共建

可以有不同的做法，融入建造和模拟经营，全面建设城市：

- 一种可以通过限制资源类型，鼓励交易，鼓励合作，便游戏一点
- 一种只融入少量游戏元素，以创造为极限，不限制资源

前者早期发展更快，后者早期参与较弱，可以两者结合

<http://creation.id/=qwe&app=appid&cam=6dof>

坐标的概念使得大家可以在RealityWorld 之外大量宣传一个地点，就在现实世界一样，甚至大家回去找这样的攻略和列表，而不是通过里面的游览

默认打开当前位置，所以去哪里都可以看看

4.1 REALITY ID

用户中心

4.1.1 (用户) 组件管理

组件版本升级，等等，保证组件是最新的。

这里编译之后列出所有问题，用户也可以直接发送信息给开发者，要求更新组件以支持某些新的标准。

这里是用户对象编译发生的地方，因为这里设置的东西基本上不会再修改，当然也应该支持在运行时动态编译，这种是少数情况

4.1.2 (用户实体) 权限管理

4.2 THE REALITY WORLD APP

4.2.1 真实世界作为底图



尽管对于元宇宙来讲，我们可以构造一个任意的虚拟世界，但是一个纯虚拟世界至少有以下缺陷：

- 它将跟现实世界完全脱节，这种割裂感会非常大，因为虚拟世界里面的内容很难跟现实世界有一个联系，要想让未来的3D成为人们日常生活的一部分，这个虚拟世界一定是和现实世界有关联的，否则它就摆脱不了类似游戏的概念，人们把它当做一个专门的娱乐方式，偶尔进去体验一下，而不是时时跟它保持联系和连接。
- 无法促进地理上靠近的人之间进行互动，在一个纯虚拟世界中，真实的地理位置在其中无法产生较好的关联，因此它们的互动通常只是现实世界的好友之间的互动，或者说通过某些游戏内容的机制促进的具有类似爱好和兴趣的陌生人之间的互动。但是城市作为一个重要的文化载体和符号，它本身也是具有丰富的信息在里面的，而且人与城市之间的关系是现代文明中人类不可忽视的重要体验，所以怎样利用好这种真实世界的地理关系，也是未来元宇宙成为人们日常生成一部分的重要部分之一。
- 元宇宙作为现实世界的延展，其实前面两部分都说明了，真实世界与虚拟世界的关联和关系，会成为未来元宇宙重要的核心机制，否则它不仅会对我们的生活造成割裂，并且它无法成为人们日常生活的一部分，就像今天的泛娱乐类应用如抖音、微信等。并且作为未来科技生活重要的一面，我们希望它要能够用来提升人们的生活品质，这包括两个层面：使人们感到更加快乐，以及帮助人们提供更加丰富的数字化服务。

所以Reality World将以现实世界真实地图为底板进行打造，以围绕现实世界与虚拟世界的紧密联系为核心设计原则，开发能够通过元宇宙的丰富数字化机制来提升人们生活品质的开放虚拟世界。

4.2.1.1 跟现实世界关联

Reality World世界的底座是真实世界的平面地图，并且保留城市主要的道路信息等。这样做有几个好处：

首先虚拟世界跟真实世界是有关联的，这种关联不仅是指地理上的位置关联，而是我们有机会去表达跟一个城市相关的信息，例如当前城市的某些指数，城市的一些文化风貌，可以抽象成某种游戏机制，这样同一个城市中的市民都可以感受到类似的与该座城市独有的体验。这些体验往往都是关于现实的信息，它包含人与城市和周围环境和人之间的关系，所以这种机制形成了对现实的增强。

最重要的是关系，地理位置是一种重要的关系，它不仅仅是位置的关系，它是一种把大家拉在一起，这里的人都有共同的一些认知的关系。所以地理位置实际上隐藏着很多信息，是非常重要的概念。

当然，与真实世界不一样，真实世界的外观和细节在虚拟世界中不是最重要的，人们希望一个不一样的世界，人们希望能够改变现实世界，通过这种机制，人们有可能创造出一个不一样的虚拟世界，这个世界代表着人们期望、向往和想象中的一个世界。也代表着人们对现实世界以及人与人之间关系的思考。

不过，与虚拟世界中的建筑物等外观不一样类似，尽管地面的道路位置是保留的，但是道路的名称是可以更改的。这是世界的道路结构及其视觉位置能够帮助人们在虚拟世界更好地导航，所以即便这些道路的名字被修改了，人们仍然能够很好地关联它们。但是如果这是一个纯虚拟而巨大的虚拟世界，人们则很难记住那么多的地址名称。

这样的元宇宙世界将能够提升人们现实生活的生活品质。

4.2.1.2 真实地理的意义

地理不仅仅是地图上的位置，它蕴藏着很多意义，几乎可以说跟我们大部分的日常生活，以及日常生活之外更重要的人与社会的关系层面，这些信息更加重要，它关乎人的情感、情绪、对生活的体验和品质等。

真实世界是关于大家通过一定相邻的地理位置彼此聚在一起，然后因此而共同关心和关注某些相同的事情，进而形成某些相关联的关系、信用、世界观、文化等等。

尽管有时候我们跟周围的人并不直接认识，但是我们跟他们之间仍然潜藏这某些联系，这些联系并不一定是显式可见的，但是它们却是客观存在的。然而对于远在一个我们并不知道的地方，所有这些联系都不存在，或者说很弱。比如说对于所有中国人，我们之间仍然存在一些联系和关系，但是对于大部分人，这种关系很弱。

因此，从某种程度上说，地理信息甚至是比亲情更重要的意义，亲情之间的联系反而是比较简单的，我们大部分的情感和精力也许会更多花在这种基于地理位置信息上。当然我们把同事等关系也归结到地理信息相关，例如我们跟另一个城市的同事往往没有本地同事之间关系那么紧密。同样的，本地同学之间通常也要比其他城市的同学之间关系紧密，因为他们之间更有可能会有更多的联系。

元宇宙怎样表达这些意义呢？

地理信息，或者说人与社会的关系，是一种抽象的信息，参见第4.10.1节的内容，我们很难用传统结构化的方式进行表达，有时候甚至也很难使用电影或者小说这种叙事的方式进行表达。由第4.10.3节可以看到，这种信息类型最好的表达的方式是能够模拟复杂系统机制的游戏程序。

因此，在Reality World中，只要我们提供足够好的平台技术，让普通大众能够表达自己的机制，就能够释放这种能力，因为大众不同的人能够抽象提炼出这种关系。从而形成关于这些关系的表达。

4.2.1.3 作为巨大虚拟世界的地图

参见4.2.3节传送门

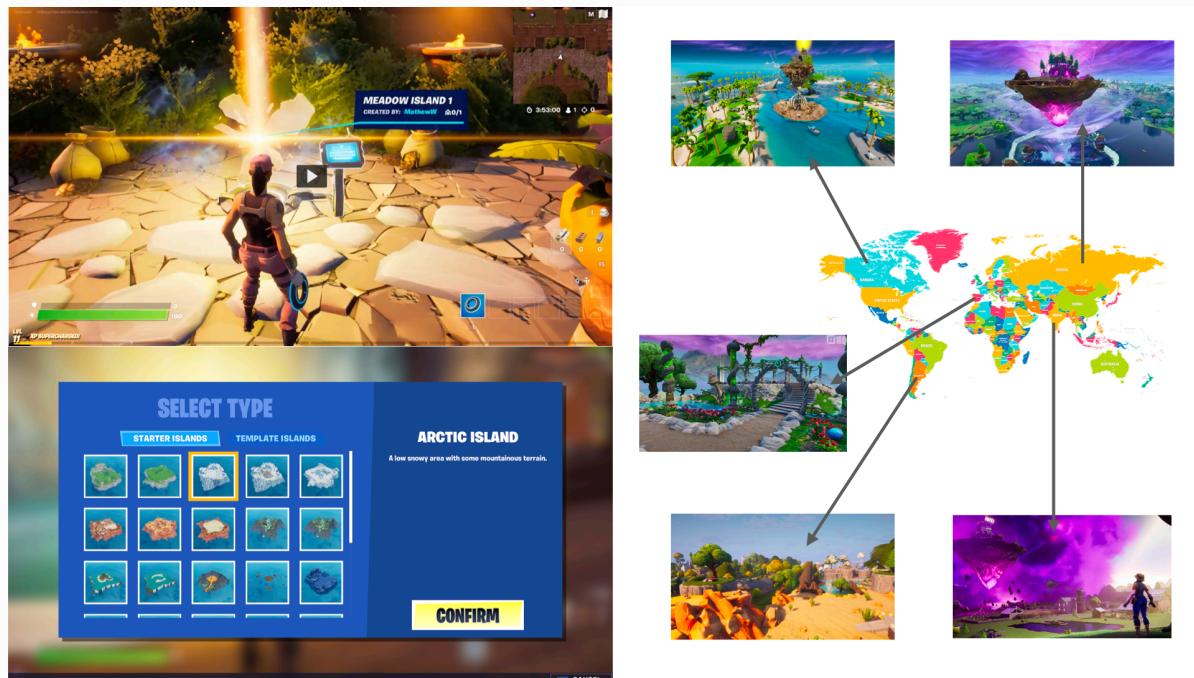
4.2.2 传送门

由于地图的限制，以及复杂度的控制，定义一些标准的传送门机制，点击可以进入私有的Creation

- 不管是Reality World中的内容
- 还是个人独立的Creation

这样Reality World更像是一个多重虚拟世界的入口，他可以去到很多不同的虚拟世界，每

个独立虚拟世界具有更不一样的体验，比如在安全方面没有更多的限制，比如可以使用暴力，有坏人，容易遭到攻击等等。这些虚拟世界可以是一个广告产品的体验，一个独立的游戏，一个其他互动内容等等。



由于Reality World主要是基于现实地板，因此形成一种虚拟与现实结合的体验。

4.2.3 Point and Click

为了简化操控，可以使用Point and Click模式为默认主模式。

4.3 源动力

4.3.1 用户：创作和体验自由度

相比其他大世界，固定的模式，模板，用户完全可以自定义自己所属的任何东西。

- 自己创作的建筑可以有完全不同的行为
- 自己的Avatar也可以设置不同的行为

总之，这种自由性产生了两个结果：

- 控制，而不是仅仅体验，自己设计创作
- 这个世界的体验会更加丰富，而不是可预期的
- 每个用户展示给其他玩家的内容也是丰富，而不是固定的模板

4.3.2 开发者：更活跃的经济市场

平台提供一个更接近真实社会的经济系统和机制，使得Reality World里面的经济生产能够根据市场行为，使优秀的内容更加获益，从而使得整个经济系统更加活跃和繁荣，来为不同的用户创造价值。

4.3.2.1 持续经济

相比于传统的数字经济市场，构建持续消费的经济体制来是开发者获益更多，并且可以持续获益。参见1.4.5节内容。

4.3.2.2 广告内容

通过提供独特的机制，使得广告语产品融为一体，提升广告的效果，从而提高销售数量。见1.4.2节内容。

4.3.3 标准作者：高级抽象能力的巨大收益

标准作者是Reality World里面最具价值的用户：

- TA具有对现实世界最高的抽象和设计能力
- 这种抽象能力使得Reality World的体验能够进化得越来越好，并且覆盖的范围会越来越丰富
- 为了维持这种利益，标准作者还会很好地维护标准社区组件的开发

所以，平台需要给标准作者最高的经济利益，只有TA们才会使整个平台越来越健康。

4.4 安全和所有权

USD的好处在于，可以把每个用户自己创作的Creation保存为独立的文件，这样方便独立的编辑、更新甚至删除；然后又可以把所有用户的内容合成到一个场景，只需要记录它们的相对坐标位置等信息。

在这样一个大世界中，权限有两种：

- 是对于单个用户自己的Creation，其中的组件可能来自多个开发者，这些不同的组件之间可能需要通信
- 第二种是当将所有用户的Creation合成到一个场景中，它们可能使用相同的组件，这时候会导致一些非法的访问

从某种程度上来说，我们真正应该关心的是后者。即是说，对于前者来讲，尽管用户使用了多个开发者编写的代码，但是对于用户来讲，最终的Creation都是他编写的代码，所以对于用户而言，他应该保证的是所有组件组合在一起是否能够正常工作，而不是去分配组件之间的访问权限：一是本质上组件访问的都是他自己的数据，二是这样的关心涉及到了代码的组织和开发逻辑，这不是用户应该关心的。但在这种情况下，确实会存在有些恶意组件破话数据的问题，这种应该小心审核组件，并且依靠举报等方式监管。这本质上是一个监管的问题，对于用户而言，他应该认为他使用的组件应该都是安全的。

如果某个组件对某个属性的修改超出了用户的预期，例如用户可能只希望只读，但是该组件确实读和写。这样的情况也应该是正常的，因为组件对变量的访问权限本身是包含在组件的逻辑之中的。如果用户认为这种权限越界了，用户应该选择使用其他组件，或者修改组件。因为直接修改属性不让其访问，这可能破坏了组件本身的意图，使得其组件的功能完全不生效。

所以readonly或readwrite权限标志应该是用来处理Reality World这种多Creation共享的场景，即我们的数据能够被不认识的人怎么访问。但这种机制会存在一下的问题：

- 直接对单个变量进行声明会显得非常复杂，所有实体的所有变量可能会多达几百个
- 直接对Creation既进行设置又会导致大部分共享交互的机制建立不起来，因为用户倾向于把所有数据设为私有，这使得基本上无法与外界交互
- 诸如RenderComponent这种数据，用户是不可以设置权限的，不过这种情况由系统决定就好了，例如系统不让编辑这部分属性，例如物理和可视相关的属性基本上都属于这种属性

用户还是应该对每个实体的每个变量管理这种权限，由编辑器或者系统动态将所有公共的符号变量总结到一起，并按类别形成一个如iOS系统中Setting的列表，由用户统一设置，这样在打包的时候直接修改这些权限设置，这样这些权限设置到实体级别是不可见的。

但这种Setting表只对Reality World有效，如果不发布至Reality World，则这个功能根本不可见。也即不会允许第三方开发类似Reality World的应用--开放世界。

这里由于符号表的概念，也简化了整个权限设置的复杂度，即同一个符号即使有多个实例，它的全系均设置一次，避免了对所有属性实例分别设置，因为符号本身也是包含了意义在里面，而不仅仅是一个随便定义的变量名字。

4.4.1 RealityIDComponent

每个Entity都应该具有RealityID这个字段，用于区分System的访问权限，知道所属关系

4.4.2 readonly

对于不同用户之间的权限问题，它们只能是readonly，即用户之间只能读取数据，不同写或者修改数据。

在这种情况下，将System强制与一个Component关联是有意义的，这样System是所有有权的，它属于某一个特定的Entity，而每个Entity拥有特定的RealityID，因此可以便于控制。如果System是系统独立方法，就只能拿到Entity之后才能决定其数据是否可用，这种情况下如果权限不够，则会造成浪费。

所以系统应该避免读取没有权限的数据，由于System与实体关联，就可以比较权限，即对于RealityID不等于自己的数据，不能进行修改操作，并且是否可读也取决于用户的设置。

这样对于程序而言，有三种权限：其中readonly和readwrite应该是Creation内部的事情；这里的readonly还有应该拆分成RealityID内部和RealityID之间，比如使用share

- readonly
- readwrite
- share

4.4.3 重新加载

由于用户可以随时修改权限，因此当某个用户修改之后，其他正在Reality World的在线用户应该对其进行重新加载。

4.5 稳定性

需要确保每个Creation在提交之前，运行时是稳定的，否则程序中只要包含这个Creation就可能导致崩溃。在Reality World这种完全开发的世界中，这个问题更是严重。

这里，核心问题其实归结为一个，即程序的稳定性，所以对所有的组件要进行审核，以减轻对后面用户Creation稳定性的检查。

所有需要提交到Reality World的必须是经过Reality World验证过的组件，否则无法发布至Reality World，但是用户自己的Creation则可以使用未经验证的组件，因为这影响的用户范围很小，用户一旦发现问题自己去解决。

4.5.1 Reality Verified Components

对组件进行审核与测试，合法的组件才能被用户使用。

未经审核的组件自能用于小范围测试。

4.5.2 提前预测过期组件

对于那些合法但是比较旧的组件，可能导致跟标准不再兼容等导致程序无法运行，对于这种过期行为要进行判断。从两个层面来保证稳定性：

- 提醒用户及时更新
- 对于未经更新的代码，系统能够在加载时动态判断，然后丢弃与标准不兼容的代码

通过以上的机制，能够保证整个Reality World的稳定性。这对于一个大的动态更新的世界至关重要。因为：

- 一方面又不能限制开发者自由提交代码，这样就不具备开放性，但是这就容易导致不稳定性
- 另一方面必须保证整个系统的更新机制，因为维护太多过期的组件，对于这种系统来说成本非常高，必须促进系统快速更新

4.6 经济与交易

由于安全性的原因，所有两个用户之间的交易，都需要调用一些特殊的系统API，这些API不应该特定于Reality World，而是所有Creation中涉及消费的都可以，因为本质上交易就是两个Reality ID之间发生的事情。

4.6.1 及时购买

3D的东西没法像传统商品一样通过图片的方式浏览就可以获得很好的了解，因为它是一种体验，视觉只是其中很小的元素，甚至视频也不是最好的了解方式

比如传统的广告，我们通常不能获得太多体验上的信息，更多是其他一些非产品因素

所以需要一种新的购买模式：当你在体验一个互动内容的时候直接一键购买

比如你在试驾一辆车，获得不错的体验之后一键购买

比如你跟好友一起玩游戏，看到好友使用的某个交互内容

4.6.2 智能购买

在用户试体验某个内容或者看到某个内容时可以理解购买，就像在商场的购买体验。

4.6.3 直接发布而不是广告

You can play the game with. And that was incredibly interesting to see. Um, I think this is going to be the future of this shared 3d entertainment, medium. Um, it's not about Facebook pages, it's not about advertising. It's about actually delivering meaningful experiences that people can interact with. And that become part of this much larger world, right? So the programming model for the metaverse must incorporate, uh, the assumption that everybody's on objects, they build should be able to interact sensibly and fit and safely with everybody else's objects, your car, you know, built by Ford should be able to interact with your motorcycle built by Dati.

If an architect to be is a major work of architecture in the metaverse, you know, that should work with all the different player models have been introduced into the game and everything should work together. So I think the center, the focus of any programming model, uh, for the metaverse needs to be open world compatibility over time, open interfaces, um, which can evolve and be extended over time.

4.6.3.1 现在的可能做法



对于这样的需求，现有可能的做法是：

- 对于每个广告产品，广告商自己开发一个应用程序并发布，由于开发者具有所有的源代码，所以可以任意交互
- 如果要实现自定义复杂交互，每个广告商需要把源代码交给平台，由平台统一部署发布，并要求用户更新
- 广告可以使用有限的交互，按照一定的标准开发，这样就可以不经过开发商自定发布

显然这些都不是最好的方法，所以RealityIS可以创作全新的商业模式，这里任意广告商可以完全按照自己的设计定义功能丰富的产品，然后在Reality World中自由发布。

4.6.4 市场经济

实际价值由人们主动参与经济的行为决定，而不是单纯的投票或者其他机制决定，确保虚拟货币的数量是由经济行为决定，虚拟货币的价值需要与这种行为产生直接关联。

所以对于Reality World的经济来说，有两点是至关重要的：

- 确保经济的主要推动者是人们的主动经济参与行为
- 确保虚拟经济的货币与人们实际感受到的价值相关联

Reality World通过构建高度开放的世界，避免中央式的干预来实现这样的经济运作。同时标准的发布、反馈以及人们实际使用相关联，通过人们的主动选择来实现价值的筛选与传递。

世界的经济规律是相对确定的，人们不管是在真实世界还是虚拟世界中都需要有类似公平的机制来保证人们的经济活动参与是有意义的，而真实世界的经济规则是人类数年来积累的成果，它也是人们熟悉的思维，只不过虚拟世界可以通过数字技术更少中央集权式的干预。

那既然是跟真实社会一样的经济体验，还有什么意义需要构建一个虚拟世界呢？尽管两者的经济体制相似，但是在虚拟世界中可以创作和体验在真实世界无法实现的事情和体验，这就是虚拟世界的价值，而且这种价值通过经济的机制而得人们觉得也是有意义的。反之，没有任何经济意义的事情可能就是无意义的，人们会把很多事情当成经济活动的一部分，即使是精神上的体验也可以是经济活动的一部分。

经济思维是人们觉得所有参与与付出会有意义的一种心理基础。

4.6.4.1 单次购买不具备任何价值

当然，尽管用户的经济购买行为为整个经济系统产生价值，但是这种价值是一种总体行为，而不是由单词购买决定的。

4.6.5 区块链

区块链解决了两件事情：

- 它通过技术手段定义了物权，并且一旦你拥有物权，别人没法篡改，因为整个任何对该物权的转换都会被记录，而这种转换只有在所有者同意之后才能被执行。
- 它的账本机制，实际上意味着物品可以被任意转换或者说交易，这就为商品的自由交

易创造了可能。反观传统的中央式的数字经济，一件物品的交易通常只发生一次，一般平台不会提供这种无限转卖的机制，即使提供这种机制，通常也是认为不可靠的，因为这些交易账本可能被篡改。

但本质上，区块链只解决关于物品交易的过程，这个过程对于经济活动只是辅助性的，但是它并不是经济活动的全部。例如：

- 它没有解决能否保证虚拟货币与真实价值的映射是否安全可靠的问题，这可能导致上当受骗；
- 它没有解决物品在交易过程中，怎样更公平地决定物品价值的机制；例如在真实世界中，人们的经济交易除了产生物品交换，这种行为还有很多其他经济价值，比如最核心的是决定物品的价值。这些机制跟区块链的理念都差很远。

除此之外，它约游戏互动内容需要大家分享和共同玩游戏才能产生价值的理念是相违背的，区块链更鼓励封闭和秘密的行为。

4.6.5.1 价值关联

区块链不解决价值关联的问题，一定数量的虚拟货币到底关联多少实际物品的价值，以及怎样关联，这不是区块链会考虑的。如果这个问题不解决，也许从源头上就不可控了，后面的物权保障也就没有意义。

4.6.5.2 价值的决定

在交易过程中，物品的价值到底怎样变动，没有更好的机制。

在真实世界中，一个物品的价值肯定不是由投票来决定的，它是由人们的经济行为来决定的。投票是可以被操作的，或者也可能是虚假的。但是真实的经济行为是不会说谎的，即使某些个体存在偏差，但总体而言是客观的，这就是真实世界物品价值决定的机制。

所以，在Reality World，我们从技术上把人们这种经济行为融入到商品价值体系，包括：

- 私人网络之间的口碑，例如如果你决定某个东西好，你会分享给朋友，这样的推荐更靠谱，你用朋友之间的关系来保障你的口碑，而不是随便一句不负责任的话。而为什么你会分享给朋友，是因为虚拟世界的互动大都是需要和朋友一起进行的。
- 竞争，对于相似的功能，可以有多个不同的组件实现，这些组件之间会相互竞争，因此竞争也会指导定价，对标准也是一样。

因此，Reality World是更接近真实世界的运作方式，它保证物品的价值是与你需要付出的代码匹配的。

4.6.5.3 转卖没有创造价值

在传统的NFT系统中，物品被反复和大量转卖，而不是像游戏一样被大量玩家真正的体验。在这些转换过程中，甚至大部分买家和卖家根本就没有去体验它真正的内容，当然也不可能有机会去改进它，或者去增加它的价值。这样的经济活动毫无意义，它对整个经济系统都毫无贡献。

4.6.6 Royalty

对于经济活动中的生产者，有两种激励方式：

- 一次性费用
- 版税

在现实生活中，一次性费用通常发生在商品交易的终端，即商品转移到最终消费者的过程中。但是对于一些比较强势和技术竞争力强的生产者，他们也会使用版税的形式，例如徕卡跟华为的合作，是按照手机销量进行分成。当然这些都会随着一些话语权等因素可以调整。

对于Reality World中的标准和组件开发者而言，有两种不同的影响：

- 版税可能更加容易鼓励开发者提升单个产品的能力
- 而一次性费用可能更容易鼓励开发者开发更多的内容，但也许它没有精力去提升单个产品的竞争力，因为无法转化存量用户的价值，永远只有新用户才能产生收入。

当然开发者也可以开发更多的标准和组件，但是因为版税的收益主要是取决于影响力和知名度，不同的影响力和名气其版税收入的差距非常大，所以开发者更愿意花心思提升单个标准或组件的品质，因为只要有好的影响力，这套机制以保证ta赚取足够的收入。就好比苹果手机，TA需要维持自己的品牌，然后销售就会很高，而其他一些手机厂商则会尝试开发多种不同定位的产品和开发细分市场。

我们显然是需要鼓励开发者制作更好的标准和组件，而不是开发数量更多的标准和组件。所有对这两类开发者使用版税的形式。

4.6.6.1 标准税

即组件开发者，在每销售一件组件时，标准的制定者可以收取一定的版税。

如果组件开发者只是基于标准开发组件，而没有形成任何销售收入，是不需要向标准作者支付费用的。这样降低组件开发者的门槛。

同样，标准作者在没有任何标准税收入之前，也不需要向平台支付费用。

4.6.6.2 组件税

即普通用户使用某个组件开发的内容，在产生收入的时候会收取一定的版税。

当然，对于组件税来说要更复杂一些，因为用户的收入可能来自多个组件的结果，很难清晰判定某单个组件的贡献。这块后续在梳理一下思路。

4.6.9 完整的生态

既要有消费者，生成者，工具制作，供应链，才能全域激活

4.7 SOCIAL

4.7.1 私人化社交

以私人化社交为核心驱动
私人化社交是**大众创作的最大动力**

传统“应用市场型”元宇宙



以Roblox为代表的传统元宇宙
参照应用市场的模式
创作者创作的作品是面对所有人群
因此必须是一个接近游戏成品规格的产品
创作门槛较高
由于面向大众内容同质化严重
创作者群体空间很小
更广泛的大众群体根本无法参与创作

RW以私人社交为核心驱动

- 产生巨大的信息传输动力
- 以及更广泛的创作参与群体

私人关系之间能够产生更多生活化的需求
因此参与创作的群体空间非常大

私人之间的主动分享和推送
是信息传输的巨大的流量和力量
而传统市场模式主要靠平台推送
缺乏内容主动传输的巨大动力

元宇宙时代的生活
更多是圈子和好友之间
而不是陌生人共处一个公共环境

CONFIDENTIAL

互动内容需要和朋友一起玩，会给朋友推荐自己觉得还玩的东西，形成良性经济系统，间接也会导致社区更文明

大多数类Roblox平台都是类似的模式，它们看起来像一个应用市场

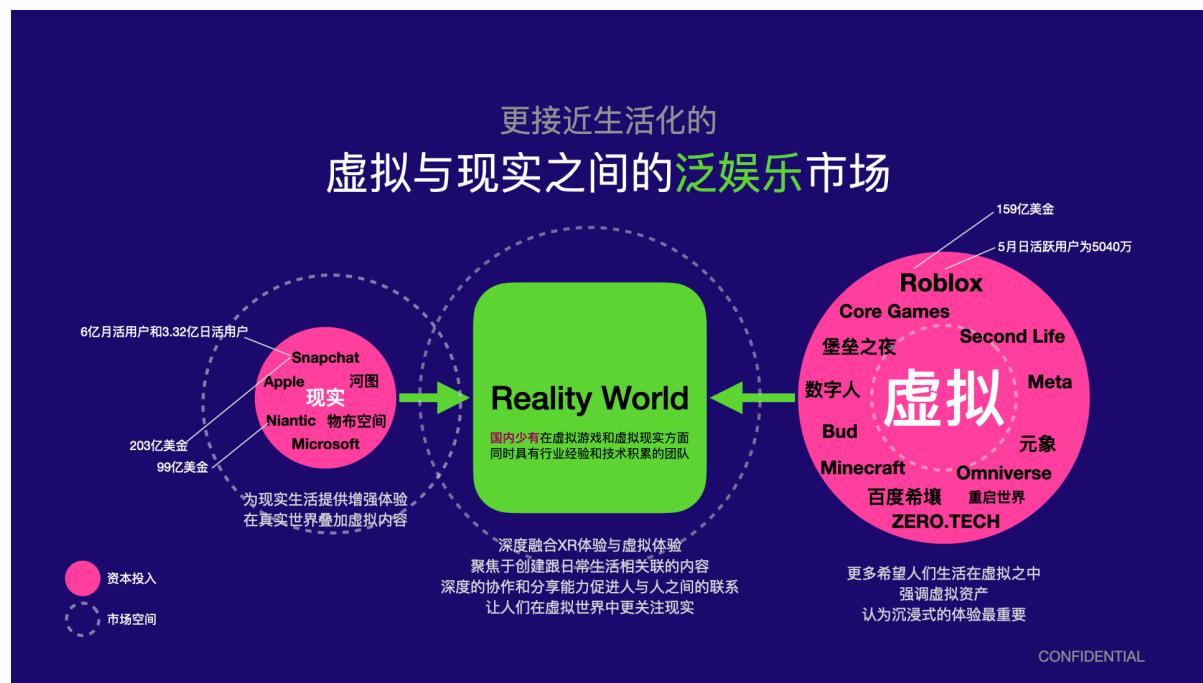
只有私人化才能促进大众创作，就像Snapchat

甚至抖音视频大部分都是围绕自己的

一个应用市场型的应用，是不会激发普通用户的，就像你要求普通用户创作严肃的大片，他们做不到，所以必须私人化

跟微信的模式，借助私人小圈子的强烈分享和创作需求

4.7.2 关注现实



4.8 创造性和开放世界

在游戏和电影等3D载体的娱乐体验中，有三种主要的相对比较独特的类别：

- 游戏性
- 故事性
- 创作性

游戏性主要对应于游戏，它是游戏中的灵魂，有游戏设计师根据自己的经验和创造力设计出的，让玩家产生心流的瞬间体验

故事性主要对应于电影

创作性的最大不同和魅力在于，创作不仅仅是一种被动的体验，它是需要思考、构思、自己个人对生活各方面的理解、感悟和想象力的，因此这种要求更高，它的结果对应的不仅仅是一种游戏态的心流，它对应于成就了，甚至某种精神物质，创作的东西才更对应于价值，才更容易产生交易

如果一个导演或者一个艺术工作者，他持续创作优秀作品的动力主要来源于创作的体验，那么将这种能力释放到普通大众当中，也一定会是不错的体验

过去的科技我们聚焦于改善一些实用产品的体验，它体现在去流程、便捷性、效率等；互联网时代除了提升人们之间的交流效率，他很大的体验改善在于释放了观察世界的能力，就是视频和照片，本质上照片和视频所反应的是每个人感知和观察世界的能力，比如：

- 使用不同视角镜头观察同一世界不同的美感
- 捕捉不同感兴趣的画面表达自己的个性、理解、主张等
- 拍摄感兴趣的视频故事反应自己的兴趣、主张等
- 转载不同的视频和图片信息表达自己的观点、价值观等
- 对已有带有各种历史、技术、文化等信息的图片和视频添加自己的理解，抒发自己的主张、认知、观点：价值等

但所有这些，他都是在观察或者学习了解这个世界，它的工具通常只是镜头，除此之外它几乎没有其他工具，他的表达能力很受限于我们眼睛所能看到的事物

而反观我们的世界为什么多姿多彩，是因为我们用各种工具如创造了建筑、车子、衣服、草地、公园、艺术作品等等整个世界，这些创造的价值是因为他们融入了人们的理解、思考、想象、甚至梦想和期望等等

创造伴随着整个文明，因为人们创造的东西改善了这个世界，提升了人们的认知、理解、生活效率、生活质量等等，所以它促进了文明进程

但是现实世界的创造性是否足够了，受限于很多物理约束，很多创造肯定是受限的

但是创造虚拟的内容相较于物质物品是否具有价值，只要创造的结果提升了人们对世界的认知和理解、提升了生活质量、精神世界，从而也就促进了文明的进程，那么他就和物理世界的创造是等价的，从这个角度说，它甚至可以不需要跟物理世界发生关联，就像一些科幻电影或者一些玄幻小说讲述的故事那样

所以，科技的下一个具有社会价值的使命是释放创造力和表达能力

当前在创造力和表达能力方面最容易实施的是写作：可以基于自己的理解创造新的理解；其次是电影和游戏，但是他们仅面向少数开发者或者电影工作者

4.8.1 分工的重要性

同现实世界一样，虚拟的创造也必然需要分工，不可能所有东西都需要每个创作者从零开始搭建

分工意味着劳动力复用，节省时间，分工也意味着价值的交易

商店数字资产、组件等其实就是分工的产物

现实世界的分工由人类自身驱动，例如行业标准由行业内部讨论决定

开发出能够易于分工协作的架构，是释放创造力和表达能力的重要基础

4.8.2 共同创造拉近距离

现实生活中的人们之间的距离：

- 亲情
- 友情
- 在一起工作或学习

围绕着做一件共同的事情，或者说为了一些共同的目标或者商业目的，合作做一件共同的事情，这是生活中最多的拉近人们距离的方式，这也是我们日常社交圈子扩展的主要来源

因此，创造性不仅仅针对个人，还需要围绕共同目标，共同创造和协同，才能促进人们之间的交流和了解

4.8.3 创造游戏性与一般创造

游戏性有玩法，目标，策略，延续性较大。

一般创造更多只是看一下，即使有交互，交互的目的性也很弱，所以需要把单个一般创作内容的体验，转化为持续，有目标和吸引力的一种体验很重要

一般游戏中很多时间的操作在于探索，探索中一方面是了解环境，一方面是收集资源，所以这些一般的3D内容中要有类似的机制，例如每个内容都可以获得一定的经验，但是经验跟设计交互有关，但经验是共享的

4.8.4 时间创造价值

如果只是玩别人设计的游戏，或者看电影，这通常只是个人视觉上的体验、个人理解的升华、心流，这种心理感觉往往很难传递给其他人，例如当别人给你讲述某个游戏体验时，如果你要获得类似的经历，你必须自己亲自玩一下，他没发通过口述传递给你

即，如果认为这种游戏体验是一种价值，那么只有游戏开发者创造了价值，而大部分玩家也是获得价值，并且这种价值不可转化

但如果希望这个世界会衍生价值和创造价值，则我们希望普通的用户能够创建可以交易的价值

时间可以创造这种价值

如果没有创造，仅仅是体验，这其实又回到了传统游戏行业：

只有少数人可以创造游戏

如果要创造出好玩的体验，需要巨大的时间

玩家都在玩同样一些游戏

只有少数游戏正在被广泛体验，少数人受益

丰富性不够

缺乏游戏之外的很多体验

只有创造和游戏体验结合，才能均分和消耗更多的时间，让用户能够持续投入，而传统游戏的活跃度往往跟一些新游戏或者经典游戏相关

4.8.5 创造的方式

- 终端用户不会直接建模，除非是程序化生成，不需要用户雕琢精细网格，这部分还是要回归传统DCC，那里可以进行更精致微调，在3维空间做不到（这样也就避免将传统DCC的工具引入进来，客户端只需要做跟位置相关的交互，大大简化，现实世界人们加工某个东西也是基于现有物体，而不是从零开始）
- 程序化生成，一些不需要精致网格，并且有自由度的物体，如地面，山脉等等，大部分跟环境有关

最后的交互是一个非常简便符合视觉直观常识的交互集合，用户基本是环境靠基于手势的程序化生成，个性物体靠模板，谢谢模版通过DCC生成，大部分脚本和逻辑也是针对个性物体

粘性，由于创造花费了巨大的时间，因此粘性更高

4.8.6 大世界合成能力

单次创造是局部的，单个局部场景可以合成到一个大场景，如果把这种能力开发，例如基于一块固定类型的地或者环境，组成自己的小世界，就容易让一些志同道合的人一起去构建一个他们喜欢的世界，可以是科幻，武侠等等风格

- 鼓励合作与协作，是非常好的协作例子
- 也给其他人的游览带来更大的吸引力，宏大的，形成众多具有更复杂表达和文化的世界，这种文化的感觉需要复杂性来表现，局部较小的场景往往无法表达一种文化，甚至一个文明
- 文明本身自带故事了

大地图在PC Create上创建，或者提供一些模版，像Minecraft Editor一样

4.8.7 虚拟世界巨大的探索成本

虽然沉浸式、宏大的虚拟世界具有很好的体验，但是相对于影视来说，其探索成本更高，例如看完一部魔戒需要三个小时，但是探索一个中土世界可能总共会花费很多天时间：

- 这对于普通用户来讲可能是不可行的
- 普通用户可能仅仅随便看看，无法深入体验故事
- 玩家对虚拟世界的探索本质上源于未知的体验，这种未知并不是单单一个一个宏大的虚拟世界，而是故事或者玩法，因此需要花大量精力设计，而一旦玩家探索玩所有未知，那么这个世界便不再新奇，除非它像真实世界一样不断会有新奇故事发生，那必须是一个开放世界，用户能够高度自定义或者甚至自我演进

所以，开放世界架构及其重要

4.9 THIRD PARTY APPS

4.10 社会价值

4.10.1 更好的信息传播媒介即信息表达方法

按照信息的组织特征，其可以分为三类：

- 一种是非常简单，能够用结构化的方式简单描述的信息，例如一个公式，一间事情的方法，菜谱，一条朋友圈，一段视频等。这种信息所表示的含义通常是明确的。
- 一种是描述人与人、或者人与事情之间的关系，这种通常比较抽象，它不能有一个很确定的、简单的方式进行描述，比如一个故事，一间艺术品，对他的传播涉及一些解释，甚至一些相关的视觉符号，文化等。
- 第三种是机制，这种机制往往是复杂系统，它既不能像第一种信息那样能够简单描述和传播，也不像电影等艺术品那样可以直接解读，由于机制内子系统构造复杂的相互关系，因此它需要新的媒介进行传播。

电影更多是对人与人或者人与社会之间的关系进行描述，理解和表达，这种关系往往是非结构化的，它很难使用一定的规则、模型、定律、公式等等进行描述，所以这种非常适合于文学、电影、美术、戏剧等形式。

与围绕人的关系情感不同的另一个维度是理解社会运作的机制，比如交通，旅游路线，城市不同的分区，工作与公司的分类等等，这些反应的是社会机制，他们是可以量化和结构化的。



很大的一个特点是：

- 人与人之间的关系往往是容易用比较简单的信息进行表达，尽管这些简短的信息是需要非凡的人对其进行高度理解和抽象提炼，比如一部电影通常就足以讲述一个深刻的道理
- 但是机制却是更复杂的，因为机制本身是一个复杂系统，它由许多相互相关的子系统构成，而这种关系往往不是人易于理解的方式，比如人很容易理解一个公式，但是复杂系统无法表述成一个单一的公式，它是一个多维线性函数，它的理解蕴藏在所有那些关系当中，不同子系统的数据会导致差异很大的关系，有多种因素的影响，而不是一个清晰的逻辑，所以他比如不太能够用一部电影来表述，或者说电影业务能够做一些科普，但是真正的理解你必须去使用那个系统，这种“使用”从数字化的角度来讲就是模拟，而游戏就是这样模拟的核心方式之一

所以，大部分这样的社会机制都可以借助3D来进行学习和理解，甚至参与影响

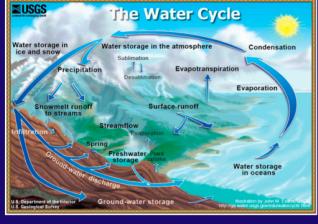
另一方面，人其实是深度跟社会高度耦合的，很多人与人之间的关系也来源于社会机制的影响，所以社会机制本质上也是另一种帮助理解人与人之间关系的一种方式

4.10.2 释放实时模拟程序机制的潜能

如上一节的信息分类，传统的应用程序模型只适合处理结构化的计算，这种计算通常是确定的，其应用结构通常也是不会的，例如微信、淘宝、抖音、支付宝、大众点评等等这样的应用程序。在这些程序当中，通常是由用户发出一个操作指示，然后应用程序按照固定的逻辑执行某个结构基本上不变的计算。

开发模式
从结构化计算到复杂系统机制

复杂系统理论



复杂系统由许多部分组成，这些部分单个看一般都很容易理解，但把它们组合在一起后形成的复杂系统大都能表现出无法预测的惊人特性，很难通过单独拆分分析每个组成部分来解释这种现象。可以通过引入正/负反馈循环来影响整个系统。

开放和自由度是元宇宙的关键



开放的游戏机制
游戏由一家公司开发，有限、固定的机制

开放，即多个开发者开发的程序可以在一个环境运行
因此游戏具有更丰富的机制，玩家有更多选择
因此具有更大的玩家群体



开放的社交行为
游戏一般具有固定的社交行为

但现实中人们个性的表达方式非常丰富
只有高度可定制、个性化、自由的表达方式
才能大大促进人们在元宇宙进行生活、社交和表达

CONFIDENTIAL

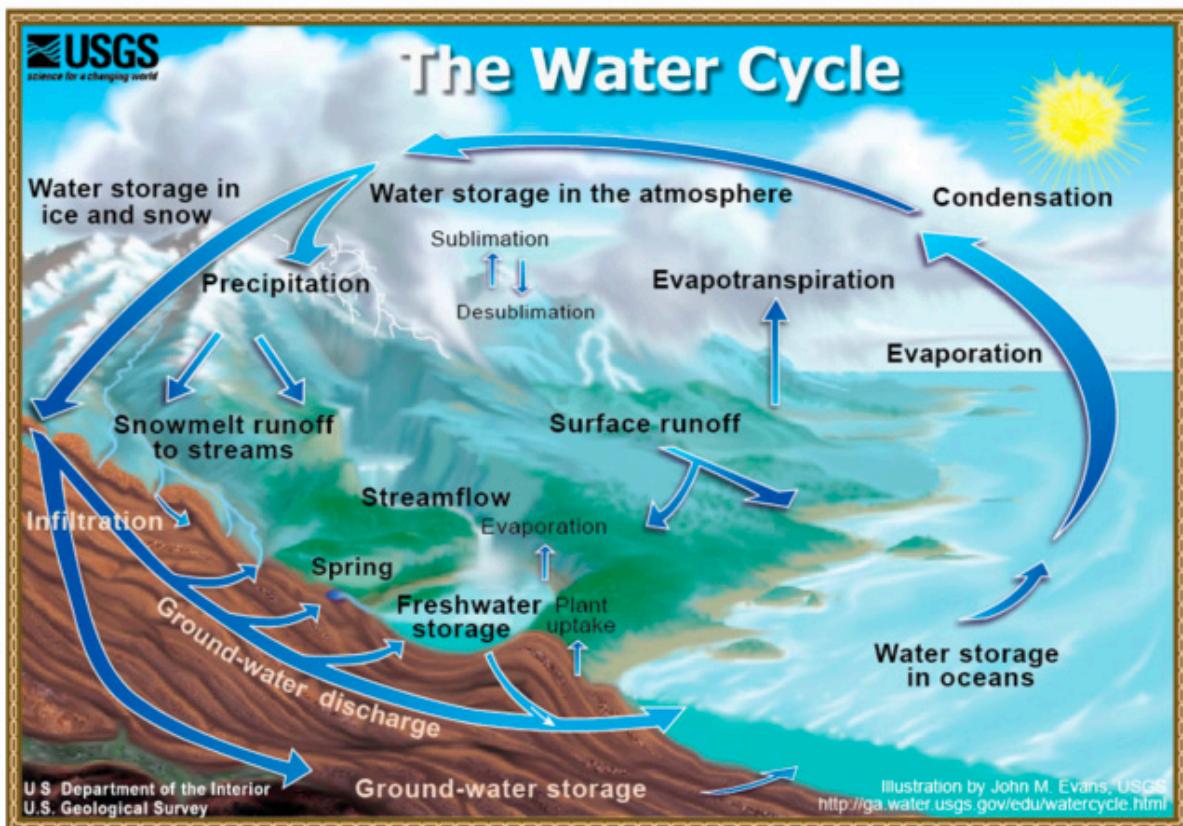
然而，现实当中还有大量的类似复杂系统的机制，这类信息对人们理解这个世界和社会可能更加至关重要，并且这类信息通常不能使用传统的应用程序架构进行表达。游戏程序架构是非常适合处理这类信息的，但是游戏程序目前还没有被广泛使用，不管是在开发工具、开发流程还是开发成本上它都存在着很多问题，还不具备这样的潜能。

所以，RealityJS有机会去释放这样的潜能，使得游戏类实时互动内容的开发变得更加简单，例如跟开发一个应用程序差不多。也许在这样的情况下，这种实时模拟的程序架构也许可以成为整个数字化的常态。这样数字化能够在人类文明进程中进一步发生更大的推动作用，因为人与人之间有一种更高效、更具表达力的信息表达方式。

4.10.3 复杂系统的模拟

复杂系统由许多部分组成，这些部分单个看一般都很容易理解，但把它们组合在一起后形成的复杂系统大都能表现出无法预测的惊人特性，很难通过单独拆分分析每个组成部分来解释这种现象。可以通过引入正/负反馈循环来影响整个系统。

复杂系统其实能够用来表达是真实生活中大量的信息，他对于我们理解人与人之间、人与社会之间等等的关系至关重要。通常这些知识要比我们一般能够从书中看到的信息要多得多。



例如关于管理，其实它也是一个复杂系统，它有很多影响因素，如果我们把这些因素用一个复杂系统来表达，这样学习者可以通过交互的方式，对某些子系统施加影响，来实时看到它们对整个管理体系影响的效果，这样的方式我们不仅可以用来学习这样的一些系统机制，也可以用来对一些机制其进行模拟和预测。

这种能力是很难通过其他应用程序来实现的，所以这样的系统将给人类的文明带来巨大的影响和推进。

4.11 标准

基于全局和公共符号表构建沟通方式和标准。

4.11.1 传统做法的缺点

Epic CEO在其演讲《Fundamental Principles and Technologies for the Metaverse》中指出，为了实现一个Open Metaverse，像我们今天的各种互操作系统如Web等一样，需要定义非常多的标准用来实现Metaverse内部各个实体、对象等之间的通信，例如关于用户的身份、资产所有权、社交图谱等等。他进一步指出可以参见现在的一些标准如HTML+JavaScript等进行设计。

然而实际上但我们再深入去思考这种方式的时候，会发现也许我们并不能使用同样的方式去设计Metaverse的标准，例如其中两个最重要的原因包括：

- **实时性**：现在的标准指定都是通过文本的形式，然后各个子系统对文本进行解析，这种大量实时的文本解析和字符串处理在游戏程序中可能会导致性能问题。
- **互操作性**：即使可以解决性能问题，它本质上只是一个虚拟机，我们将别人的代码放在我们的环境中初始化和分配变量，因此可以获得直接的变量地址，但是拿到地址之后要进行正确的通信还是需要了解关于函数的定义等等，否则我们只能约定一些固定的调用行为。

以上这套机制假说能够很好的工作，它也只是针对双方约定的接口进行通信，这就限制了自由度。传统的一些互操作系统本质上它们之间的通信非常简单，基本上可以使用一些固定的规则进行描述，且标准之间变更的频率非常低。

而Metaverse是一个更加活跃的大世界，它就像真实世界一样运作，因此它本质上不能使用这种限制比较大的方式，例如我们生活中跟其他实体之间的交互是非常自然的，我们有很大的自由度，没有被严格限制每件事情一定要按怎样的方式做，当然它也存在一部分固定的规定，例如我们要遵循交通规则，法律规则等等。

4.11.2 开放的标准架构

如果Metaverse是要尽可能模拟真实世界，或者说它的整个系统更符合人类的认知，它的标准必须支持两种方式：

- 既要能够像传统的标准那样制定固定、需要公众共同遵守、不太容易变化的标准，如

交通规则

- 又要能够支持局部群体之间定义自己的小标准，并且对于这些小标准，群体之外的参与者只要愿意遵循该小标准对应的协定，TA们就可以很轻松地参与到这个小群体中来。

上述的机制很像现实世界的运作方式，它让人们即有很大的灵活度和自由，同时也受一定的社会约束。

RealityIS的符号表就提供了这样的机制，符号表本身提供了一种定义标准的机制：只要两个独立的程序包含（类比于遵循）这样的符号数据定义，它们自然就遵循了相同的标准。所以，对于全局符号表，这就是对应一些公共标准，而对于一些局部的开发者，它们可以创建自己的局部符号表，从而构建局部小标准，这样理论上来说就是一种完全的自由度，比如你甚至可以定义别人完全不知道的标准，这种自由度是存在的，只是那样你没有办法跟别人进行交互。

所以，你需要去推广你的标准，但是这种推广也不是把你的东西放到一个组件市场或者去做广告，你可以通过两种方式去推广你的标准：

- **用户层面**：你只需要将你的作品放置到这个世界中，当有其他人体验到它时，他可以直接就复制你的组件。虽然这种方式并没有引入新的开发者来遵循你的标准，但是它引入和增加了使用它的用户。实际上我们建立的标准，当被其他开发者支持之后，我们最终的目的还是希望通过更多的支持程序来获得使用的用户，从这个层面来讲，它的结果是一样的。
- **开发者层面**：每个开发者定义的标准都可以发布到一个共享标准库，其实就是共享符号表。其他开发者可以搜索共享库，并通过对其引用以支持这个标准。这样，定义的比较好的标准就容易被更多的开发者引用和支持，因此被更广泛的使用。

符号表形成的标准系统是一套自我自进化的标准架构，在这样的架构下，任何标准不仅能够被其他独立开发者任意支持，以形成标准的推广；并且通过及时购买等方式，标准能够被更直接的通过用户进行普及，从而能够推动那些更好的标准被更多的人群使用。

通过上述两种机制，最终整个系统或者说世界，向着优秀进化的能力使标准实现自我进化，并且带动着整个世界进行自我进化。

4.11.3 标准管理

标准即是整体系统进行自进化的机制，也是实现用户实体功能的机制。它的整个管理和更新必须非常高效。

标准管理借鉴了现代应用程序市场的推送、源代码包管理、Github多版本管理等思想。但它同时也包含一些RealityIS独特的机制。它实现的功能不仅包括开发者向用户的推送，也包括用户向开发者甚至标准作者的反向建议，以及标准作者建议开发者针对新的符号进行开发的建议，总结：

- 标准更改：向组件开发者推送
- 组件更改：向用户推送
- 新增标准建议：标准作者增加新的功能，建议组件开发者支持
- 组件增强：用户对组件的建议
- 标准建议：用户或者开发者对标准作者的建议
- 特性建议：用户可以针对标准作者或者开发者提出新的相反的组件开发建议

整个RealityIS的自我进化功能都是围绕标准的一些列机制来实现的。

同时标准管理的另外一个大的目标是使用户的组件始终保持最新，减少维护旧组件带来的复杂兼容机制。

标准的管理有两条线：

- 一个是自上而下的更新通知
- 一个是自下而上的反馈建议

4.11.3.1 标准更新通知

如果标准本身有更改，会通知到所有支持该标准的开发者，提醒他们升级版本。开发者在收到通知之后，可以发布支持新标准的新版本组件。当然组件开发者需要实现兼容性。包括：

- 新增符号
- 符号重命名
- 删 除 符 号

标准更新机制使得标准能够快速进化，而不会由于信息的滞后甚至不知道标准的改进而导致一个标准迟迟无法快速进化。相比于传统的方法，有两项改进：

- 它以标准为中心，标准的修改可以直接通知到所有关注者；中间没有任何时延。传统

的方法通常需要开发者主动去关注某个标准，以开发者为中心去推动一项标准。传统的方法是一个大家共同来制定标准的过程，这种效率极低，而RealityIS反过来，先定义标准。可以这样做的一个原因是RealityIS简化了标准的定义：它仅关注一个逻辑结构中需要相互通信的参数。

- 它直接告知需要修改的地方，由于标准中的符号跟组件的变量引用关联，所以系统可以计算出哪些组件需要修改，什么变量需要修改。而传统的方法通常是通过文本的方式，如邮件，告知修改的内容，然后开发者再对照修改。这使得开发者能够快速进行修改。

4.11.3.2 组件更新通知

如果组件开发者更新了组件，也会自动通知到所有使用该组件的用户。用户可以选择一键升级，或者用户可以开启自动升级。

同样，由于组件都是结构化的、数据驱动的方式，而不是写死在代码中，所以系统可以较为容易地将所以这些组件信息抽取出来形成列表。用户的所有实体对象都可以很方便地罗列出来，所以就更方便用户对这些组件进行管理。

同时，这也意味着很方便地对实体对象进行管理。

4.11.3.3 标准反馈机制

有以下多种反馈机制：

- (向开发者) 反馈组件功能：用户基于组件的功能理解和需求，用户可以给组件开发者提供反馈意见，以完善或增强某个组件。
- (向标准作者) 反馈标准结构：可以向标准组织提出建议，例如修改、删除或者重命名符号。这里的反馈中可以是用户或者开发者。
- (向标准社区) 征询新功能开发：可以在标准社区发布新的功能需求，开发者可以按照相关需求进行组件开发。

4.11.3.4 始终保持最新（版本管理机制）

为了减少对旧组件的兼容性维护成本，所有组件最好都是保持最新。

这里其中一条可选的做法就是用于只保存最新的标准，这样旧的组件就必须升级。但这可能导致有时候在组件没有更新之前无法工作。

另外就是标准作者可以设定一些旧标准存续的时间，给开发者和用户一段时间进行更新升级。

或者系统默认就是两个版本，其中每一个新版本发布之后，旧版本最多存续固定的时间，如三个月，三个月之后自动删除。这种方式看起来是两者的一个权衡。

4.11.4 跨越标准

从逻辑上讲，组件关注的只有符号，而不是标准，标准只是组件开发者在开发组件的思考过程中的一种参考，他对标准本身没有直接的所属或者依赖关系。

因此，一个组件是可以跨域标准的。

如果我们把每个标准理解为一个子系统，那么这种跨越标准的组件就可以实现标准之间的通信，从而实现标准之间的联系或者关系。

例如整个天气系统包括云层子系统，海洋子系统，天空子系统，陆地子系统等等，然后这些子系统之间是存在一定比较简单的关系，从而形成整个天气复杂系统的。

4.11.4.1 与相关标准的属性

如果把每个标准看做一个更大复杂系统的子系统，那么每个子系统中必然有部分属性是与其他子系统的属性相关的。

因此，每个标准通常会包含少部分与相关标准有关的符号，对于这些符号，它们的属性值通常由内部的机制计算，然后这些值会影响到其他相关标准中与之相关的属性。

但是对于哪些是相关属性，我们不需要去约束它。这只是开发者脑中知道的一个概念。从理论上来说，标准的任何属性也许都可能与其他子系统有某种关系，因此我们不需要限制他，这只是对开发者的一种指导。

4.12 自我进化的METAVERSE

一个不能自我进化的Metaverse就是一个游戏，这显然不是Metaverse的形态。此外，即便我们解决了多程序交互的问题，它只是增加了一个游戏内的系统会更加丰富。然而对于一个好的世界，这种丰富不仅仅是指数量上越来越多，而且需要在丰富上形成层次，甚至对于社会的运行机制，后者是更重要的，因为用户关注和需要的是有层次的信息，而不是更多海量可能存在大量无意义无价值的信息。然而，仅仅向其中增加程序的能力不能保证这种丰富形

成层次。

当一个开发者向其中添加了一个新的程序，怎么能够判断这个程序的价值？并且是要通过用户的视角去评判这个程序的价值？

在真实世界中，社会进化的机制来源于两股力量：

- 少数优秀的人能够创造一些好的东西，这些东西不仅仅是指一个具体的实物，更可能包含一些结构、关系以及社会运作的一些逻辑，这些东西在Reality World就对应标准，标准的数据结构及其数据组合背后反映的是一定深层次的结构、关系和逻辑。
- 这些好的东西会被其他少部分人接触到，不管是地理位置上较近，还是熟人之间介绍等等，这部分机制在现实社会中往往通过广告进行加强。当这一少部分人使用之后觉得真正有价值的东西，他们会形成推广的力量，通过人与人之间的关系把这个有价值的东西推向更大的人群，如此，那些最有价值的东西被逐步挖掘出来。

上述这两种机制导致的结果：

- 人们会觉得创造东西会有价值，你有机会被更多人使用，从而为更多人创造价值，你的创作也有机会被更多人认可
- 人们会觉得社会越来越进步，幸福感更强，因为你感觉这个社会在进步，你越来越能使用到更好的东西

这种进化最根本的力量来自于社会个人，而不是少数中央机构。所以要实现这样的自我进化，我们一定要有类似的机制来释放个人的这种力量，而不是依靠平台，平台没法做这件事情。

4.12.1 标准的价值

符号表是实现多程序/多应用之间进行互操作的核心机制，标准则构建于符号表之上，它是一组语义上相关的符合集合的概念。符号总是存在于一个标准之中，即符号按标准的形式进行组织。

相比于单个符号，标准是对现实世界某些关系或逻辑的抽象。标准中的符号是围绕某一类关系或者某一类事物的核心的数据属性，其中通过这些数据要能够描述该类关系或数据的特征以及各个层面，这些属性应该是便于人类理解的。

所以标准是Reality World的一个核心指标，它也是代表用户创作的最高抽象能力，我们对于现实世界的一些关系的深刻抽象理解都蕴藏在标准的定义及其结构中。

Reality World中的所有组件都是围绕标准来开发的，这保证组件不会太混乱，因为它们是按照很严格的逻辑来逻辑的，这种逻辑由标准来定义。同时他也保证组件之间的交互变得有意义，相关性比较高，因为相对于同一标准的不同组件，它们彼此知道应该怎样协作，因为这些标准中的符号不仅包含一些的相关性，也包括对逻辑开发的指导。

标准的这种定义形式也使得标准易于定义：我们只需要找出描述某类关系的数据，而不是需要去实现或定义对这些数据进行操作的方法，这样就是的标准的定义根本不需要很复杂的组织结构，比如类似USD类多层级的结构。

标准也应该是可以自我进化的，标准的作者可以对标准中的符号进行新增、修改、重命名或者删除等，通过这种机制来实现标准的进化，形成更好的抽象，更好的标准。而动态编译的机制，以及标准更新通知机制，使得使用这些标准的用户或者组件可以得到通知，使得组件在程序中可以正常运行。最终真个系统或者说世界，向着优秀进化的能力是标准的自我进化，好的标准代表着优秀的事情，他会被更多的人使用，它是一个重要的指标

标准本身也是一个类型查找的依据，标准的设计应该围绕某一类主题，而不是泛泛的涉及多个无关内容的标准。这些也是普通用户进行创作时的组件筛选机制之一。

4.12.2 基于标准的商业模式

针对符号表的版权，类比指定标准，符号表标准本身比实现的组件是更高价值的东西

4.12.3 自我进化的标准

4.12.3.1 自我进化的核心机制

4.12.4 标准的自我进化

前者是指一个好的标准，有一定的机制被更多人发现，从而促进了标准的推广，这是一种维度的进化，因为这样好的东西会越来越被更多使用，从用户来看，这个世界变得越来越好。

后者是指，促进标准本身的进化。即对于一个被广泛使用的标准，这个标准并不是100%完美的，它本身还有改进空间，它本身也可以延后

4.12.4.1 符号表的版本兼容性

4.12.4.2 以标准为中心的社区

订阅的机制

4.12.4.3 推动组件更新

参见4.11.3.4节，标准的更新机制（保持最新两个版本）使得标准能够快速进化以及牵引用户更新。

标准的这种版本机制还促进了组件更新，当标准更新了，可能不久之前，比如一个月或者半年的组件将有可能过时，这时候为了持续被其他新用户使用，它必须更新组件，否则新用户无法购买，这样就促进组件开发者快速更新。

如果是已经购买的用户，它可以反馈要求组件更新，这里面就有类似的机制刺激开发者：

- 组件购买是一次性的，所以开发者不用对过期负责
- 但是用户需要升级组件时，这是一次重新购买行为，对开发者来讲有二次收入，当如开发者可以设置老用户优惠，甚至老用户免费升级，所以开发者有足够的动力去做这件事情。

另外，从实际来讲，真实社会也是这个样子的：

- 你买的东西是容易过时的，你可能会重新购买相同产品的新品，比如手机

此外，对于软件来讲，更新很快，你不可能开发一个组件就用几年，这种陈年的老代码后面一定有兼容性问题，而且它不更新也代表着用户体验的还是很久之前的东西，当然这些东西也有可能经得起时间考验。但对于这种情况，有两种思路：

- 对于这种比较稳定的产品，其代码也趋向于稳定，开发者每次可能并不需要花费很多时间就可以更新升级一下组件
- 标准也趋向于稳定，每个标准都是一个进化的过程，到一定的阶段它也会趋于稳定
- 此外，标准开发者为了避免开发者流失，TA也要尽可能保证稳定，否则频繁变化的标准有可能会流失开发者

另外，对于传统的App来讲，比如有时候看到很多非常久的app没有更新升级也能运行，但是因为它是独立程序，只要OS保持一定的兼容时间即可，但是对于在一个应用程序内的开放世界来讲，这个事情会复杂得多，所以前期使用更简单的方式处理。

4.13 用户创作

4.13.1 以组件为最小粒度

4.13.2 以标准为整体思维

4.13.3 反馈和评价

4.13.4 实体和组件管理

5. APPLICATIONS

每个产品要思考3D带来的价值增益，而不是简单把东西搬到3D或XR

1、生日墙



好友A用AR手机或眼镜找一块墙面进行创作，其中可以把背景图色，纯色或某种pattern，然后在墙上设置装饰和定制祝福，其中某些元素包含一些交互；最后将结果发给好友B，好友B找一个立面或者纯虚拟的方式就可以3D查看，如果是立面，也根据语义识别，将背景墙换色

可以点击交互有生日相关的流程，例如点蜡烛，出现特效，唱生日歌，现实特定的信息，好友一起围观等等

2、二维墙面涂鸦类创作

随便找一块空地地面，从地面拉一个垂直面，就可以进行墙面艺术创作

有交互的涂鸦

3、3D脱口秀

4, 移动的灵感氛围



5, 做一个解密游戏



可能包含移动拼图，包含拨一定的顺序点亮按钮，所以需要包含一些特定玩法类型的数据结构

6、知识讲解类会很多

一个模型，有些交互点击展示，普通人可以制作，不只是官方制作少量

7、虚拟房间

比如现在的虚拟聊天类场景，一般都是官方提供的少数固定场景，或者允许一定的定制性，但是通常定制能力有限，比如几何是固定的，只允许改材质，或者只允许增减部件，或者移动位置

在RealityWorld 里用户也可以创建更加丰富多样的聊天环境，然后邀请用户进来聊天

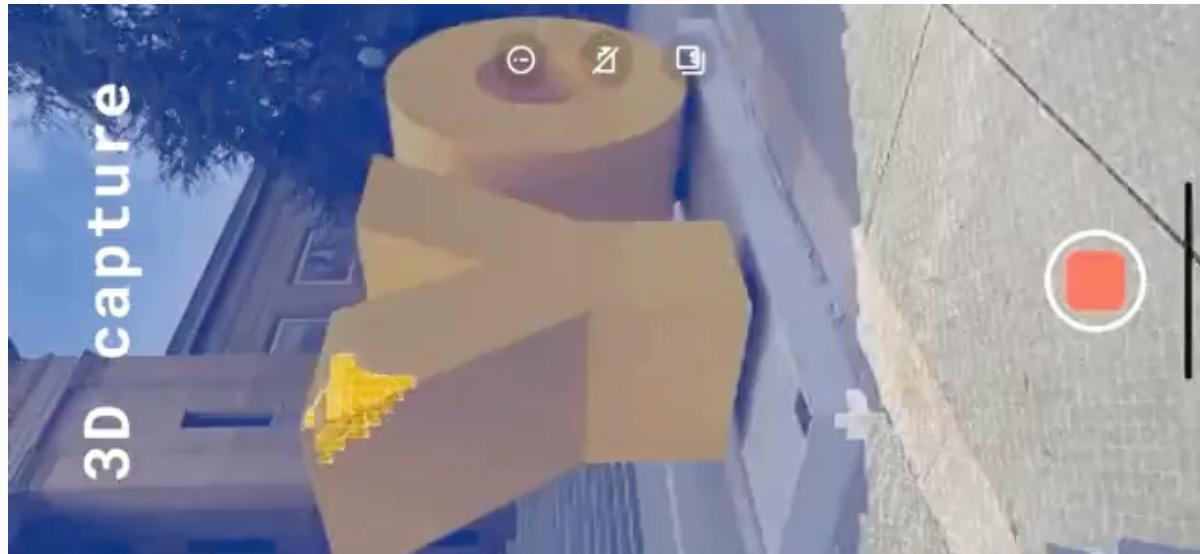
8、个人收藏馆

有一个自己的房间或者特定的场景，可以是自己设计的，有自己的收藏，可以加入一些自己的玩法，好友参观可以赠送Creation, Part

9、去一个浪漫的Creation 中约会

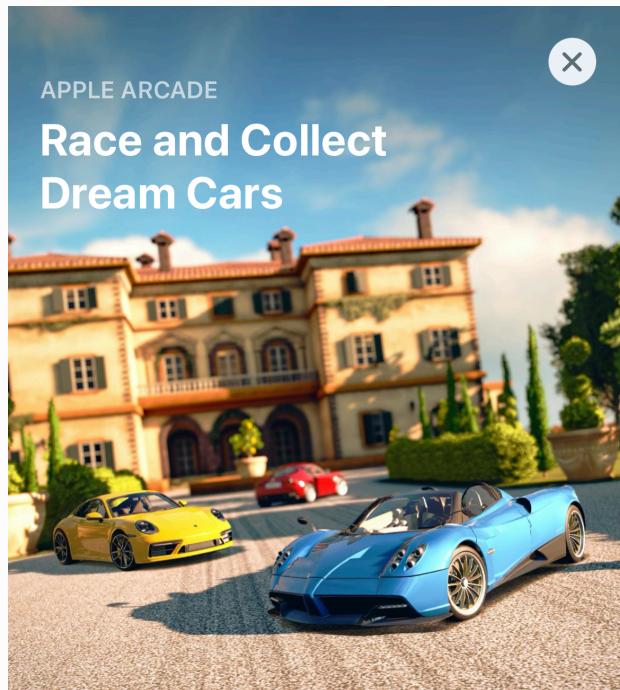
creation.id/=qwe&session=sessionid

10, 基于现实的创作



现实提供粗略的几何与参考材质，在此基础上进行创作

11、



12、连续剧，整个开发周期均可发布

把游戏关卡或者故事一点一点更新，每次玩家玩一小段，像连续剧一样，甚至世界都是一点一点构建；或者有点观看创作过程，过去整个游戏需要一次性做完再走发布流程，这种可以实时把中间创作过程共享出来，可以反馈，主要解决的是中间任何状态都可以发布，而不是要留到最后只做完了再去处理发布相关的事情，这些事情导致不能提前发布，这实际上是一种流程上的创新，带来全新的模式

13、story telling

*Like a well-executed joke, the pleasure is in the experience more than it is in the retelling. You have to be there.



Super Brothers 开创了一种叙事+交互的非常优秀的体验，相对于单纯游戏，他的故事线让整个世界观呈现更完整，相对于电影，他的交互可以让故事的体验更真实

创作部分涉及的内容基本上均可以实现

故事的元素：

- 环境，物理环境通常是静态的，但一些重要物品通常是动态的
- 信息，需要探索不同的地方了解信息
- 交互，对信息的探索是一种交互，其他比如解密，开动机关等等
- 世界状态

14、互动小说/故事

在传统小说的基础上，2D+3D，先文字介绍基本剧情和背景，然后进去3D场景，具有沉浸感，而且因为前面的文字剧情，对场景的探索会更融入，然后条件是需要在3D场景中完成一定的任务才能进入下一章，把游戏的机制融入进来，游戏结合文字剧情，弥补了纯游戏探索需要耗费大量时间，并且剧情比较零碎的感觉

6. 核心参考架构

本文介绍对Reality World有影响或者可以参考的技术架构，通过分析他们的技术原理，识别其背后的技术架构，以及其特定技术架构蕴藏着的对开发者或者用户生态的影响。对于每一个技术方案，也会分析其优缺点，以及Reality World应该怎样吸收这些优点，最重要的，我们应该从这些架构中得到什么更深层次的、有价值的信息以帮助Reality World构建更好的技术架构和技术方案。

每个参考架构按如下的格式进行描述：

- **新思想：**相对传统技术方案，该技术方案该来什么新思想或新思路
- **技术方案：**对相关的核心技术架构进行表述
- **不足及原因：**在Reality World的方向上，该技术方案没有解决什么问题，或者无法解决什么问题；其中的原因是因为技术方案的不足，还是产品定位和方向的问题
- **对比：**Reality World与其对比存在哪些差异，或者说Reality World通过什么样的技术方案来解决这些问题

6.1 数据格式

6.1.1 USD



Universal Scene Description (USD) is the first publicly available software that addresses the need to robustly and scalably interchange and augment arbitrary 3D scenes that may be** **composed** **from many elemental assets.

6.1.1.1 新思想

- **协作**: USD是一个为了大规模协作的高性能可扩展软件平台
- **交换**: USD提供了在多个DCC工具之间进行交换的格式，这通过内置的一些schema实现，包括geometry, shading, lighting和physics等
- **合成**: USD独特的合成特性提供了强大的收益，比如能够将丰富多样的individual asset合成到一个大场景，这允许多人同步协作（而不会导致冲突）

USD的合成引擎对任何特定的domain是无感知的（agnostic），所以它可以被扩展来编码（encode）与合成其他domain。

6.1.1.2 技术方案

Schema

Schema用于从UsdObject编辑、查询和定义结构化的数据（structured data），大部分核心库中的Schema是prim schemas，这又分为两类：1) IsA Schemas；2) API Schemas；3) 另外还有一些Schema称为property schemas。

一个prim可以订阅多个API Schema，但是只能订阅一个IsA Schema，USD提供了工具用于生成Schema的代码。

IsA Schema

IsA Schema用于定义一个prim在Stage中的角色或者目的，它继承自UsdTitled类，并可以指定typeName metadata，例如：

```
UsdPrim::IsA<SomeSchemaClass>()
```

IsA Schema可以是实的或者虚的，例如UsdGeomImageable是虚的IsA Schema，而UsdGeomMesh是实的IsA Schema，因为它包含一个Define()方法可以定义typeName。

API Schema

API Schema是prim的Schema，它们用于提供接口（API）对prim相关的数据进行定义、编辑和提取。它继承自UsdAPISchemaBase类而不是UsdTYPED，因此相对于“is a”可以称为“has a”。API Schema有三类：

- Non-applied API Schemas
- Single and Multiple Apply Schemas
- Multiple-apply schemas

Model, component and Assembly

Kind是一个prim-level的类型系统，它相对于schema type抽象层级更高，对应于Model Hierarchy：

- model, kind的抽象基类
- group
- assembly
- component
- subcomponent

相对于更细碎的asset或者prim，model提供一种将场景结构进行细分的架构；model结构也提供了更好的方式管理和引用资源，否则对更对referenced assets引用和推理会变得复杂。

```
def Xform "TreeSpruce" (
    kind = "component"
)
{
    # Geometry and shading prims that define a Spruce tree...

    def "Cone_1" (
        kind = "subcomponent"
        references = @Cones/PineConeA.usd@
    )
    {
    }
}
```

Asset AssetInfo and Asset Resolution

Asset是能够使用一个字符串标志符 (via asset resolution) 被识别和定位的资源，asset可以是一个文件，或者多个文件组合引用形成的单个文件，它一般有版本控制，为了方便一些如asset dependency analysis等操作，USD定义了一个特殊的字符串类型，asset (represents a resolvable path to another asset)，这样所有的metadata和attributes都能被很快地定位和识别。

尽管USD的composition arcs能够用来合成场景，但是他们并不方便在内存中对资源进行定位和识别，AssetInfo是composition arcs的补充：

```
def Xform "Forest_set" (
    assetInfo = {
        asset identifier = @Forest_set/usd/Forest_set.usd@
        string name = "Forest_set"
    }
    kind = "assembly"
)
{
    # Possibly deep namespace hierarchy of prims, with
    references to other assets
}
```

Asset Resolution是将一个asset path转换为可以定位一个资源的location的过程，默认按照文件路径进行搜索，但是开发者可以自定义定位逻辑，甚至资源不一定需要存储在磁盘中。

Prim, Property and Attribute

USD的命名空间主要由：Prim和Property组成，其中Prim提供对合成场景的组织和索引，它是USD的主要容器，prim可以包含另一个prim，形成一个namespace hierarchy Stage；

Prim，连同它的indices，是Stage中唯一被持久化存储在内存中的数据；对prim进行操作的接口由UsdPrim类提供；prim可以拥有一个schema typename，也可以提供如scene-level instancing, load/unload behavior, and deactivation等操作。

而Property提供real data。有两种类型的Property：

- Attribute
- Relationship

Property可以被组成新的层级且不需要引入新的Prim，这可以提升内存的局部性：

```
#usda 1.0

over MyMesh
{
    rel material:binding =
</ModelRoot/Materials/MetalMaterial>
    color3f[] primvars:displayColor = [ (.4, .2, .6) ]
}
```

Attribute:

```
def Sphere "BigBall"
{
    double radius = 100
    double radius.timeSamples = {
        1: 100,
        24: 500,
    }
}
```

6.1.1.3 USDZ

USD的核心是方便对众多分散的资源进行合成，其中资源已分散的碎片形式分布，这种机制是为了编辑态设计的，此时整个场景还没有编辑完成，需要继续维持这种分散的状态；然而当我们的内容全部编辑完成时，分散的文件却不利于管理，此时的核心需求是分发，它要求一下特征：

1. A single object, from marshaling and transmission perspectives
2. Potentially streamable
3. Usable without unpacking to a filesystem

USDZ通过USD提供的FileFormat plugin机制实现：UsdUsdzFileFormat

6.1.1.4 不足与原因

USD主要聚焦于怎样对合成的场景进行编辑和提取，因此它偏向于“low-memory footprint, higher-latency data access”而不是“high-memory footprint, low-latency access to data”，因为在内存中缓存更多数据，会影响对基于合成结构场景的编辑和提取，带来复杂性，因为在编辑阶段会有更多的数据修改，而运行时阶段则基本上保持数据不变。所以USD不太适合运行时。但是USD提供了一些便利和工具（facilities），使得客户端可以对UsdStage构建一些扩展性的缓存，以提供对USD数据的低延时访问。

USDZ从分发的角度对USD进行了增强。

另一方面，USD主要是为了在DCC之间进行数据交换和协同，这全是编辑时的需求，有很多功能本身对运行时没有任何用处，例如由大量的碎片组合形成的大型场景虽然适合编辑时，但是却不利于运行时加载，所以需要在运行时对USD进行一定的定制，例如是否从Core中删除一些模块，或者去掉一些功能。

数字内容的生命周期：

- 编辑 (USD)
- 分发 (USDZ)
- 运行时 (USDX)

需要在USDZ基础上进行运行时改造，使其分发得是适合运行态得格式，所以可以隐藏分发态，只需要编辑和运行时两个形态；

同时为了保证继续编辑，需要将编辑态和运行态区分，但是

6.1.1.5 对比

- USD太过复杂，有很多额外的属性都是为了合成场景的目的，而合成场景并不是运行时需要的，或者说一旦到了运行时，有些合成属性已经固定了，我们便不再需要那么复杂的属性，例如至少不再需要单一场景（一个微型app）内部所有合成属性，那么该微型场景就应该转化为固定格式场景，而整个合成能力保持在微型场景层面就可以
- 且对运行时不太友好，有很多属性，跟上面一样，有很多不必要的合成细节在运行时执行，这部分要去掉

针对上述特性，有必要开发一个中间格式：

- 它在开发者保存场景至RW时执行预处理，其计算过程主要是提前计算一些合成方面

的计算

- 最终运行时直接加载该格式
- DSL针对该格式就行优化
- 针对每个微型场景，开发者本地会保存原始USD文件，下次他仍然对原始文件进行编辑，然后提交时进行预处理，再保存中间格式至云端内容服务器

兼容第三方，在第三方做插件，转化为RW支持的格式，然后在RW做场景和交互编辑

然而怎么实时多人协同修改，另外用户开发微型场景大部分都是在端侧运行时进行

具有对用户可读的表述形式，和对程序高效加载的二进制形式

6.1.2 Alembic formats

6.1.3 Unity Prefabs

6.2 数据驱动架构

6.2.1 Unity DOTS/ECS

传统的DOTS或者ECS还是仅关注性能层面，但是数据驱动的好处是它让开发者把逻辑区分了出来，所以在这些逻辑的组织层面再加上一层管理，就可以向上层用户层进一步简化逻辑开发

SRP Batcher把材质数据从原来的raw data里面抽取出来，这样就能：

- 让GameObject rendering的性能随着scriptable pipeline得到提升
- GPU中可以缓存材质数据，而不需要每次都切换shader，因为draw call提交的频率远高于材质数据切换和提交的频率

System 对component 的引用比较复杂，Unity为了简化以及不改变原来的代码，让开发者实现一个原来的Component 类，然后自动拆分，这样也许会使得用户不易于彻底使用数据驱动的思想，另一种是通过一个特殊的语法糖包装引用，然后编译器自动将引用转化为通过Component 复制，而不是对象引用

用户不应该关注并行计算：

Unity的EntityQuery似乎可以按上述的思路去优化，甚至并行计算的显示调用都是隐藏的，用户不应该关注并行，用户对单个System 的执行自动转变为并行，包括实际的查询和并行执行，只需要每个system必须在头部引用Components 即可（声明包括是否只读的使用说明）：

- 这个引用声明跟传统的编程类似
- 编译器可以根据这个引用建立archetype
- 编译器根据这个执行自动查询
- 编译器根据这个执行自动并行计算
- 可以通过设定逻辑类型和几何或者外观类型之间的约束和对应关系来控制新类型的创建，但是这些仅发生在编译期间

太极的Megakernel programming 有这样的思想，将传统element-wise的编程，多个计算阶段合并为一个single kernel，编程理解更自然

一开始就要教会用户，怎样基于逻辑或者功能组合来创建，这些都是基本规律和逻辑，设计的时候要考虑高度通用性、抽象性、逻辑性，这样用户理解成本最低，而在设计的时候构思的成本也最低，以及逻辑之间的关系，这其实是本质：

- 构建子逻辑及其类型
- 以及逻辑之间的关系
- 几何与外观的类型
- 几何外观及其类型与逻辑之间的关系
- 然后有一套框架来支撑这个体系

Unity的DOTS做的还不够彻底，它还是为了兼容原来的Component，依赖于编辑器属性把Component和System分离出来，数据和逻辑之间的关联、关系和区别都没有那么明显，不利于深入贯彻数据驱动这一理念

在RW中，所有逻辑和数据必须分开，它们没有办法混到一起，但你可以选择不将组件发布，变成private的，但软件架构一样，并且遵循同样的包管理，中心化的组件管理和加载，版本管理，只是组件不对外公开而已

其中一种让多个开发者遵循公共协定的方法，是由平台来定义数据，平台定义的是一些业务数据，这些数据成为公共接口，多个组件之间就可以相互独立工作；如果平台缺乏某些类型公共接口，开发者也可以自行定义，但是开发者需要选择哪些属性是公共接口，然后其他开发者可以基于这些公共接口开发，这些公共数据接口跟与特定方法相关的数据接口分开

数据驱动的重要性：

- 代码可以重用，所以有机会将一部开发者写的代码共享给其他人
- 将代码从数据剥离出来，才可以做到普通用户能够构建丰富的功能

原则：

Composition over inheritance

虽然ECS相较传统面向对象概念没那么直观，但是OOP及其继承的方式带来的逻辑上的复杂性和可维护性，ECS其实更简化了，它简化的原因是两个：

- 问题分而治之
- 逻辑层次更扁平，组织复杂性降低

开源ECS实现：Flecs

ECS是实现in-game editor的核心

数据驱动也要支持网络服务相关的功能

协议由官方来定义，开发者实现功能，但这种机制会开放给开发者，使得开发者可以在内部实现协议定义，然后通过私仓或者代码文件分享给别人，前者最好，然后优秀的协议专为公共的

6.2.2 ECS

缺点：

- 共享组件：包含关系和处理顺序，这些概念促进耦合
- 跨系统通信：两个独立的组件间需要通信

ECS game engine design

用户感知的应该只有Component，他不需要知道有个System，Component的设计原则应该是一个功能节点，Node，像Houdini中的节点一样：

- 这个节点告诉用户该节点为目标物体添加了什么功能
- 这个功能应该用一个语义上的名字描述，一定要起一个很好的名字，官方要保留一些常见功能的名字列表，有一个总的名字列表，不能开发者随意起名字，除非是他内部私有的节点
- 需要构建一个全局的功能节点列表可以在文档中供用户选择和查询，普通用户真正使

用的是公共列表中的功能节点，官方约定和维护其中名字和数据结构定义

- 开发者可以提交新的节点可以，但需要提交经过官方审核，审核数据规范，跟其他节点之间的兼容性，例如某些类型的节点不适合于某些类型的物体
- 理论上一个节点定义只能有一个System，但是System之间的实现差异很大，尤其性能差异，可以通过一些性能测试方法选择其中最优的版本为默认版本，最好不要让用户选择版本，哪怕是随机选择一个版本也要，例如提交的时候，可以要求开发者提供性能测试结果，所以节点的实现要定义一套好的仿真测试工具和框架
- 一定不要通过用户去查看其中的数据才能理解节点的含义
- 当需要修改参数的时候才会去打开节点的参数

这其实和苹果的USDZ的思想类似，只不过苹果只定义了少量节点，我们可以通过定义几百种节点实现很丰富的功能

- Houdini 有几百种
- Fortnite 也有100多种

通用引擎不会这么做，只有in-house或者堡垒之夜这种沙盒游戏会这么做

6.2.3 UE5 MASS

6.2.4 Data-oriented and -driven

<https://www.dataorienteddesign.com>

Data Oriented Programming unlearning objects (book)

6.2.5 Rust ECS

<https://specs.amethyst.rs/docs/tutorials/>

6.3 编译器与DSL

RW底层需要极高的性能来支撑上层复杂的图形和逻辑计算，同时这种性能优化又要同时对跨平台移植性和开发效率带来好处，所以它不是单纯的性能优化，是一套高度优良的底层框架，具体之前一下几个重要方面：

- 图形管线的深度定制

- 针对数据驱动的优化，数据驱动除了让普通用户能够使用逻辑，还要通过数据驱动来简化开发者逻辑的开发，例如只要按照某种数据结构，不仅能使流程更简单，还会是的底层编译时和运行时能够针对这些数据结构进行优化
- 脚本语言的深入定制，跟上面的数据驱动相结合，用户起来极其简单

6.3.1 Taichi

Born from the MIT CSAIL lab, Taichi was designed to facilitate computer graphics researchers' everyday life, by helping them quickly implement visual computing and physics simulation algorithms that are executable on GPU. The path Taichi took was an innovative one: Taichi is embedded in Python and uses modern just-in-time (JIT) frameworks (for example LLVM, SPIR-V) to offload the Python source code to native GPU or CPU instructions, offering the performance at both development time and runtime.

6.3.1.1 新思想

Taichi是一门面向物理模拟和计算机视觉计算的领域特定语言，相对于传统使用C++等语言自行实现的算法，taichi提供一下方面的改进：

- 提供了并行计算抽象，使得开发人员不需要特别进行并行计算管理，并且具有较好的一致性
- 简化了代码复杂度性，主要是两个方面，一个是因为不用关心并行计算，开发者专注于单个Kernel，省掉了一定的复杂度，同时逻辑更清晰；另一个是通过将数据和算法分离，使得像稀疏结构这样的算法被自动优化，使得开发者不需要为了性能编写很多复杂的代码，例如对数据结构进行复杂的管理和排布
- 高性能，通过编译器进行了大量针对特定算法的优化，因此性能提升比较大，但比较局限于一些特定的算法，因为这些优化正式针对这些特定算法的数据结构或者形式进行优化的
- 即时编译，Taichi提供即时编译和预编译两种方案，其中针对即时编译，由于能够知道一些运行时的信息，因此Taichi提供了更多的优化
- 跨平台部署，将上层算法全部转换为统一的中间表述，有利于跨平台部署，这也减轻了开发者针对多个平台进行适配的痛苦过程

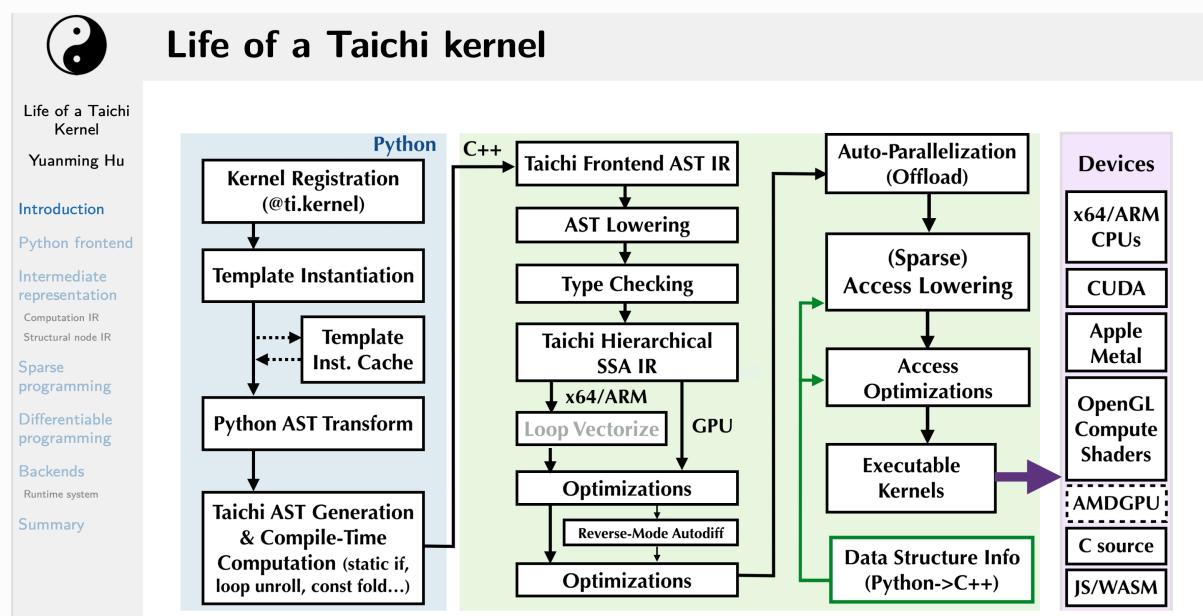
其他方面：

- **Academia**，学术界的研究实现中，往往因为缺乏优化，临时的实现方案，往往导致很难复用，Taichi希望改变这个局面，一方面通过优化的底层技术支持，一方面提供统一的接口

- **Apps & game engine integration**, 由于统一的中间表述，使得Taichi容易跨平台，Taichi可以编译一些跨平台的库供其他平台调用，例如Taichi的AOT (Ahead of time) 模块可以构建并保存在computer shaders，这样可以被其他运行时调用，AOT和JIT是两种模式
- **General-purpose computing**, 虽然早期面向特定的目标如物理模拟，但是也会有更多的通用计算支持，例如TaichiSLAM
- **Maybe a new frontend**, 可以将Python改为其前端

6.3.1.2 技术方案

以下为Taichi语言的核心技术架构：



Taichi的核心是一个编译器（compiler），在这个编译器中，它针对特定的一些计算任务，如物理模拟，稀疏结构，以及向量化的类型等进行优化，通过修改和调整这些数据结构的内容布局，使得这些计算的缓存局部性更好，同时也通过向量化的数据类型系统，使得应用的内存占用更小，从而也减少内存对带宽的占用，不仅提高了计算效率，也减少了内存占用。

为了实现上述的目的，Taichi在前端语言中（目前是Python），通过元编程定义了特定的类型，例如：

```
import taichi as ti

ti.init(arch=ti.gpu)

n = 320
```

```

pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels: # Parallelized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

i = 0
while gui.running:
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
    i = i + 1

```

这些自定义类型构成Taichi语言，它们借助Python的AST控制能力，生成带来Taichi类型信息的中间语言，然后底层的编译器就可以根据这些类型信息进行特定的代码优化和调整。

其中，相对于传统的编译优化过程，Taichi更是针对例如稀疏结构等特定的计算进行了大量的代码调整，生成相对于前端数倍的代码，这个过程不但减少了上层语言的代码量，还对其数据结构及内存排布做了大量的调整，以提升计算性能。

实现上述能力的其中最重要的思路是数据驱动，它将一些特定算法的数据表述从算法指令当中分离出来，从而使得编译器能够对这些数据进行修改，已生成更加优化的数据结构及内存排布。当然，这样的上下文知识必须针对特定的算法，并不是对所有算法都能实现优化。

6.3.1.3 不足及原因

相对于Reality World的产品方向，Taichi存在两个方面的不足：

- **面向算法而不是逻辑**，尽管Taichi可以减少代码量，但是它的核心是面向特定算法结构的优化，它不涉及逻辑层面的考虑，例如怎么帮助开发者实现交互逻辑；
- **不支持动态创建**

Taichi最大的问题在于，它的底层优化特别重，这也是实现性能提升的关键，而这一层优化实现于C++中，因此Taichi的运行时程序只能是两种情况之一：

1. 是运行时带上Taichi的编译器，这种情况可以支持任意的算法修改，这也是PC上的一般模式
2. 如果运行时没有Taichi的编译环境，需要提前将Taichi程序编译，那么这个程序一旦部署就不能修改

第一种模式是比较慢的，因为运行时需要即时编译，并且Taichi的编译过程相对于一般的程序编译要更加复杂，因此这种模式不适合实时的游戏引擎系统；而对于第二种模式，由于算法被编译为固定的底层机器语言，因此程序不能再被动态修改，除非Taichi将所有的解释过程上移至脚本语言（Python）这一层，但尽管如此，虽然能够支持动态程序，但是上层的解释过程仍然非常复杂。

Reality World解决上述问题的方法是只提供脚本语言层面的优化或者解释，而为了保持性能，RW不会进行任何底层的特定优化，同时因为RW要保持计算的通用性，以及它的目标是面向逻辑结构，所以没有必要去做这一层，只需要通过脚本语言层的DSL使得开发过程足够简单就行，RW的架构更像Unity DOTS。

一些Reality Create的内置固定算法适合用Taichi编译吗？

Taichi的另外一个问题在于，它的编译是跟算法数据结构相关的，例如分配的列表是5个还是10个元素，这些都会跟编译器耦合，它是一个与数据有关的优化，而不是一个只与抽象结构有关的优化，或者这些数值即使不是来源于运行时，也是与类型的定义有关，所以本质上它只适合与固定的算法实现。

6.3.1.4 对比

	Taichi	Creation Script
目标	面向底层算法	面向上层语义、功能、逻辑
编译类型	C++侧	脚本侧
优化算法	针对特定算法深度优化，例如针对稀疏结构，以及向量化进行深度的优化	仅针对开发复杂的进行一定的优化，例如简化并行计算，引入一些快捷变量，一些边界限制等

6.3.2 Modular AI

6.3.2.1 新思想

我们需要下一代编译器和编程语言来帮助解决这种碎片化

- 首先，计算机行业需要更好的硬件抽象，硬件抽象是允许软件创新的方式，不需要让每种不同设备变得过于专用化。
- 其次，我们需要支持异构计算，因为要在一个混合计算矩阵里做矩阵乘法、解码JPEG、非结构化计算等等。然后，还需要适用专门领域的语言，以及普通人也可以用的编程模型。
- 最后，我们也需要具备高质量、高可靠性和高延展性的架构。

在GCC之前，每家公司都要开发自己的：前端->优化器->后端，每家公司通常只开发一种前端和一种后端，导致碎片化；GCC将三者分离，减少了碎片化；

LLVM是一系列库的组合，它的模块性凸显了接口和组件的重要性，Key insight:
Compilers as libraries, not an app

- Enable embedding in other applications
- Mix and match components
- No hard coded lowering pipeline

此外，LLVM还让JIT编译（即时编译）能有更多作为。虽然JIT编译器已经是一种著名的技术，但它一开始是用在其他地方。有了LLVM以后，芯片设计、HLS工具、图形处理、都更加便捷，还促进了CUDA和GPGPU的诞生，这些都是很了不起的成就。但更重要的的是，LLVM整合了的碎片化。LLVM出现之前有很多种JIT编译器框架，但LLVM的存在，提升了JIT编译器的基线，让它迸发出更多可能，也让行业可以实现更高层次的创新。

LLVM的主要缺点是不太适合做并行处理优化。

加速器是什么？可以把它高度简化成两个部分：

- 第一个是并行计算单元。因为硅本身的结构也是并行的，加速器要用到许多晶体管，也就需要很多硅来达成这种并行处理能力。
- 第二个部分起控制作用。它的名字不太统一，有人叫它“控制处理器（Control Processor）”，有人叫它“序列器（Sequencer）”。有人希望它小一点，所以会做状态机然后嵌入寄存器。这个部分基本上起到编排并行计算单元的作用。如果并行计算单元是一个大型矩阵乘法单元，控制处理器就会命令它执行一些宏操作，例如从这个内存区加载、执行某一操作、执行另一操作、更新SRAM等。

还有一些加速器很不一样，所以控制逻辑和计算之间的比率也各有不同。正如Patterson和Hennessy所说那样，你可以选择不同的点，但每个点都需要一定程度的编排。但人们常常忘记其他一些相关的工作，比如，你不止需要编排，还要解决启动问题，比如电源管理，还要不断调试排错。如果你想做得尽善尽美，可以对这些部件进行编程；如果你希望简单一点，可以把这些部件做得很小。

当控制处理器和并行计算单元都齐备之后，怎么给它们输入和输出信息？这时就需要一个内存接口。根据抽象等级的不同，这个内存接口可以是小型的block，也可以是支持物联网的芯片，这样加速器就可以和该芯片连接整个网络通信了。这里需要用到像AMBA或类似的技术。

你可以在更大的粒度（granularity）上构建整个ASIC，所有的ASIC都在加速，在这种情况下，你可能正在与PCI通信，并且正在芯片外直接访问内存，但这种“我有一个控制处理器，有一个计算单元和有一个内存接口”的模型，是构建这些东西的一种非常标准的方法。

因此，我的主张是创新编程模型，发展新的应用程序，通过不断创新推动行业向前发展。我们应该对此过程所需的一切实行标准化，通过标准化能够快速完成工作，然后就可以把时间花在真正重要的事情上。

现在有一种相对较新的编译器技术MLIR可以帮上忙。你可以把MLIR看作是一个元编译器，它允许你非常快速地构建加速器/编译器。MLIR的全称是“多级中间表示”，它支持构建分层编译器，并以适用专门领域的方式构建，同时保留领域的复杂性。然后，使用MLIR提供的大量库和例程来做一些事情，比如，用多面体编译器来做循环展开和循环融合等等。

现在，我们开始看到的是，MLIR开始统一异构计算的世界，这也是我希望看到的。所有的大公司现在都在不同程度地使用MLIR，我认为，建立在RISC-V之上的MLIR很有必要，因为一旦开始从下往上整合行业，就可以开始把越来越多的层（layer）拉到一起，重复使用更多的技术。这使得我们可以专注在堆栈中更有趣的部分，而不是一遍又一遍地重新发明轮子。

也许你不会感到惊讶，但我认为答案是编译器，这是真正要走的一条路。

作为编译器编程语言从业者，我认为硬件设计这个领域已经到了重新评估的地步。整个领域是建立在两种技术之上，但实际上主要是一种叫做Verilog的技术，你大概率可能不喜欢Verilog。它有一个非常复杂的标准，当我看它时，不知道它是被设计成一个IR，也即一个不同工具之间的中间表示，还是被设计成让人们直接书写的语言。我认为，它在这两方面都很失败，它真的很难使用，对工具来说也很难生成。

此外，EDA工具、硬件设计工具已经非常成熟，它们非常标准化，有很多大公司正在推动和开发这些工具。但他们的创新速度并不快，设计时并不注重可用性。它们比加速器编译器要差得多，绝对不是以软件架构的最佳实践来构建的，而且成本也非常高。因此，这个领域有巨大的创新机会。

我不是第一个认识到这一点的人。在开源社区，已经构建了一堆工具推动行业向前发展。这些工具非常棒，比如Verilator被广泛使用，Yosys是另一个非常棒的工具，它有很好的定理证明器（Theorem Prover）。

我的担忧在于，这些工具的理想目标是试图像专有工具一样好，而我并不真的认为专有工具有那么好。另外，这些工具的设计者并没有合作。每个工具都在遵循单一僵化的方法，没有实现大程度的模块化或重复使用，可以从其中一些工具中得到网络列表，用它来解析一些Verilog之类的东西。但是，它不是由基于库的设计构建，与LLVM之类的东西不一样。

要创建在语法上正确，并且能表达你想要的东西的Verilog非常困难。此外，因为许多与Verilog有关的工具都有点奇怪，而且很难高质量地预测。生成与工具兼容的Verilog是每个前端工具都必须重新发明的一门黑科技。因此，在堆栈中真的缺失了一种组件，这个组件允许人们在编程模型水平上进行创新，并允许人们找到方法让所有工具都接受它。

有一个叫CIRCT的新开源项目正试图解决这个问题。CIRCT的全称是"Circuit IR for Compilers and Tools (编译器和工具的Circuit IR)"，它构建在MLIR和LLVM之上。CIRCT社区的目的是提升整个硬件设计世界，促进编程模型的创新，并启用一套新的模块化硬件设计工具。它确实运用了很多我们到目前为止一直在讨论的基于库的技术。

此外，它提供了一个可组合的基于库的工具链，可以建立有趣的新的弹性接口连接，你可以建立Chisel社区正在探索的新编程模型，用它来加速Chisel流程。它带来了很多好处，可以让很多人一起工作，推动不同方式的创新。我们正在建立一个真正伟大的小世界，让关心硬件编译器的人在一起工作，这很有趣。这项工作仍处于早期，目标是更快地构建加速器，让加速器变得更快。

- Modular,
- composable &
- layered architecture is what the world of AI needs, and **we are building it for everyone.**

6.4 OTHERS

6.4.1 神经网络

6.4.2 开源代码管理，如pip

6.4.3 区块链/NFT/虚拟货币

NFTCN/Bigverse/Opensea

当前的NFT数字资产市场主要还是偏2D

- NFT资产生成本身很简单，因此也容易复制、山寨
- 创作方式简单，大多数甚至都是一些简单的图像处理及风格化工具或者简单的编辑，而开发3D的内容生成要难得多
- 仅仅是一个市场，无对内容进行价值发现和价值增值的方式和空间，当前主要的机制是低买高卖，空等着增值，比较依赖于所谓的一些估值的机制和服务来判定价值，但实际上现实生活中一个艺术品的价值有时是通过人们的了解、学习、结构、研究、背后的文化价值和社会价值的发掘和演进后，才会慢慢催生一个作品的价值，而一个简单的市场并不足以形成这样的机制，这样作品需要一种能够更生活化的呈现机制，而

不仅仅是一个列表，它应该能够让更多的人对它有更多维度、更多机会的再认知，结构、解读、欣赏、观察、体验，这样才有机会去挖掘它的价值，它绝不是你买来放在那里他就自己会增值

- 仅仅一个市场，作为一个独立的3D或者3D的形式展现，缺乏与之相关的环境，特别是3D作品往往不是单个物体那么简单，它的表达往往和环境等因素有关，作为一个单独的作品，既容易被复制和下载，又缺乏表达能力

拟娲的3D创作更难，并且它的内容不是单个主体，而是融入在环境中，甚至和其他内容一起呈现，环境甚至程序都是一部分，在脱离这个环境，他甚至都无法运行，被复制的风险降低，同时它并不是一个单纯的市场，它更多是处于一种被欣赏的社会状态，它的价值更容易被解读和结构，融入真正的价值评判体系和方式

3D内容不适合交易

不像传统的图像艺术，更多被收藏，互动内容是动态更新的，它也与特定的运行时环境和操作系统高度耦合，他还会进行不断更新，修复bug，因此他更适合按玩家收费，按服务收费，而不是一个一次性的产品买卖，当然仅作为纯资产的数字内容不一样

虚拟地块：

虚拟地块也是没有好的价值支撑，并不是任何一个虚拟空间弄一个唯一地块划分的机制就可以，那样的话就太简单了太容易了，现实很多元宇宙的概念和产品给人的感觉就是太简单太容易了，这是不符合逻辑的，其根本原因是没有价值支撑，没有共同的价值认同基础，比如如果在赛尔达里面建立虚拟地块，那肯定有价值，因为他有价值认同基础，但你随便搭建一个地块，他并不具备共同认同价值，仅仅是像赌博一样少部份人的炒作，大部分人并不认可这些价值

所以，怎样创建价值认同基础是最重要的，因为价值创造和认同是很难的，不然就编程虚拟货币一样一种纯机制，没有价值担保

价值认同需要所有人能够参与，以某种方式，才能形成价值认同，这种参与或者价值认同应该是现实世界一样，需要一种实际的参与方式，而不是仅仅像炒房一样，就是说它需要具备真正的价值，不管是游戏带来精神上的，或者它传达了某种知识或者文化，这样才能形成价值认同

传统数字内容得货币化还存在一个重要问题，它需要一种担保机制，不像实物一样物权是唯一的明确的，当然这也是有国家机构担保负责物权管理，虚拟货币或者区块链虽然在机制上能操作一个数字内容的物权唯一，但是这个数字内容本身是可以复制的，例如一个作者可以在不同的数字平台发布，理论上，只有平台自身可以保证物权唯一，平台必须与作者达成实物或者原始作品的物权协议，比如作者不能在其他平台再使用次作品，或者作者销毁原始电子版，这样保证该平台的物权唯一，但这些机制都依赖于真实世界的法律保障，实际上虚拟货币只要是跟现实世界关联的，它的物权都需要现实世界的法律物权保障

除非是完全虚拟货币，他不需要映射实物，货币的发行本身就是基于区块链大型的，所以它本身就有物权保证了，但是这种也是没有法律保障

拟娲的数字内容是由用户创建的，或者大部分，虽然用户可以导入一部分已有资源，这部分无法控制物权的唯一性，需要借助现实世界的版权机制，但是它的大部分内容是用户基于拟娲平台创建的：

- 首先是它的那个内容在创建时已经基于区块链技术得到物权保障
- 其次是，它的数字内容不像传统的图片或者视频，其很容易被复制保存，也不像传统的3D内容一样容易被下载，它本身是一种私有格式，他必须借助平台才能解析，它的内容也全部存储在云端，这使得它的物权能够进一步被保障：你只能在这个平台才能体验这个作品，但在这个平台它是唯一的，你不能将它复制到其他地方
- 当然拟娲本身还有一套价值认同的机制，就像游戏一样，你必须要发挥你的经验和创造力，付出时间和思考才能创作出好的东西，而其他用户在体验过程中体验到了你的文化艺术表达，你的价值并被认同，用户有一个体验的过程，而不仅仅是一个商品买卖，这类似于商品需要使用才有价值，这个使用就是对数字内容文化艺术表达的体验，使用+交易 才能形成循环：使用发现价值，促进交易，进而促进使用价值的创造
- 静态2D艺术品其实还是纸质更珍贵，无论是创作的难度、仪式感，还是体验欣赏时的专注和仪式感，在数字屏幕上看2D艺术受很多因素影响，比如受屏幕分辨率、色彩、屏幕尺寸等因素影响，但3D数字内容则天生是虚拟的

区块链技术导致的思维有时候很受限，大家会倾向于放大这项技术的功用，认为一切皆可以使用区块链，因为所有事务本质上都涉及物权和交易，但他有缺点：

- 有时候会让一些事情更复杂
- 而更大的问题他可能会限制我们的想象力，例如游戏的分发跟传统的交易是不一样的，传统的交易物品是独立的，或者不可分割的，一次交易对应一件具体的物品，但是游戏程序通常是有持续更新的，当游戏这种数字资产发生变更，他的Token会变更吗，变更了原来的购买就失效了，但不变更怎样让订阅的用户享受到新增服务，所以当前的NFT市场主要聚焦于图片这种通常不会修改的静态内容，它本身不是程序，而3D数字内容会面临更复杂的变更

- 再比如，传统的书画是收藏的概念为主，它本身是由实体唯一性演变出来的方式，通常一个画家不会批量画同一副画，但交互程序天生就是数字出生，以复制的方式为主，他要面对的是玩家数量的概念和模式，单幅画的收藏价格很高，普通人根本承受不起；而游戏面向群体，所以单个复制的成本很低，这才导致普通人可以参与，这是两种完全不同的模式，前者容易导致反复拍卖，多次交易，而后者几乎只有一次性交易，防篡改的需求很低，我们唯一需要保证的是版权而非物权，而区块链恰好不擅长版权
- 3D防篡改的需求远低于图像，因为它的数字内容篡改不仅仅意味着数字存储的内容变更，而很有可能篡改的程序无法运行，因为它的运行时环境，及其他一些依赖如联网等高度相关，对于游戏程序人们一般篡改的是外挂而非程序本身，我们主要保护的是原始的版权，而非交易过程中的物权

6.4.4 Rust

游戏和图形系统相对传统应用架构并行计算的行为更多，要重点关注语言对并行计算的支持

embark.dev

Rust-gpu

Kajiya

rafx

gamedev.rs

6.4.5 Unity EditorXR and SceneFusion

6.4.6 BEVYengine

基于数据驱动的rust游戏引擎

6.4.7 工程学，工业设计

6.4.8 magicavoxel

感觉以Voxel 为基础的创建，一是比精细的三角形便于生成，而且符合物理创建的方式

6.4.9 所有平台的插件架构

Unity unreal engine blender

6.4.10 传统DCC流程

6.4.11 Meta Builder bot

语言或文本至少生成要素，如果有误差，至少大部分要素，然后用户交互专注于精调，或者把要素和交互分开，在用户确定要素之后，相当于有了类型先验，再确定交互有更多背景

在C端，从数据库选择会比较复杂，所以对数据进行类型划分，并具有一定的无法对内容进行组织变得非常重要，大大减少交互成本，而将交互集中于当前场景可以看到的东西，所以的按钮都转成语音或者一些快捷文字，短语，关键字

<https://mp.weixin.qq.com/s/JKe-KrlnwAzljb9ndXpJoQ>

6.4.12 Houdini: Node-based Workflow

3、Houdini

要像Houdini一样把复杂的结构定义，方法构造，流程定义，参数赋值，等全部去掉，用户只专心写逻辑，最后整个编程是像Houdini 那样干净整洁的，没有代码痕迹的

Houdini的node没有参数，只有节点，他把operation分成一些类型，那个类型之间的参数传递是固定的

Houdini的Node network 是一种运行过程，他既可以充当：

- 静态配置，如果所有节点没有延迟，瞬时执行，他就类似于一个静态设置，如果这些节点的指令能在一帧内执行完，这有点类似于一个usd或者其他合适文件的加载过程，相当于在加载过程的顺序执行中，后续的加载可能会修改前面的属性，但不足的方面是加载的过程无法并行化，因为Node network 是一个顺序指令
- 动态动画，如果某些节点的执行需要时间，他就形成动画

Houdini的流程和数据驱动的流程是不一样的，一个是执行计算的过程，一个是单纯的组合配置，这是因为Houdini 实际上是一个内容生成的过程，或者说它是编辑态，而ECS是运行态

内容生成过程中怎么对一个基础模型进行变形，其输入输出参数是确定或者可以推导的，但是逻辑之间的关联则比较复杂，没有规则，需要定义协议

Directable results

由于所有设计都是过程式的，整个创作过程都是过程式的，所以可以在最后一刻进行修改，而传统的硬编码，它具有许多中间形态，没有形成一个整体流程和结构，所以很容易中间某些流程或者逻辑变了，其后面的流程都受到较大的影响，甚至需要重新修改后面的代码和逻辑，迭代成本高

数据驱动有点某种程度上可以实现这个目标

其实就有点像编译流程或者渲染管线一样，他定义好了整个管线之间的参数和接口，那么中间的调整就不会影响那么大，你只需要遵循接口规范就行

Tool building

基于node-based的工作流程可以使得自定义node变得可能，只要遵照node之间的协议，然后node就可以共享，即：

Houdini Digital Assets

上下游的参数形式基本是还是Houdini 本身的node定义的，开发者只是把中间某些处理过程保存为一个.hda 资产，因此不需要定义参数，创作做的只是把一个复杂的生成过程复用，这些复杂的生成过程还是用Houdini 基本的操作，Houdini有海量的基础操作，创作者几乎很少会编写自定义函数代码，或者只需要简单很小片段的代码

所以Reality World要想做到这一层，要对游戏逻辑脚本进行深入分类，并把这些模式术语化，这些分类要能够对比，所有游戏的逻辑组合，以及保证相应的灵活性，使得自定义的成本最低，自定义的模式更简单

A new way of thinking

由于全新的方法，可以定义的能力而不是针对具体问题重复解决，就会产生一种新的创建数

字内容的思维

编程都是在node之间编程，固定的输入输出

跟Houdini 的主要区别是他是离线的，编辑态的，不需要内容审查，兼容性检查，安全性等，而拟娲是运行时的，包括包的大小，仿真模拟，安全，兼容性等问题很多

一方面可以通过编译器要做一些分析，另一方面用户需要在自己的环境跑起来

6.4.13 realityOS

6.4.14 OpenXR

苹果退出了OpenXR意味着：

- 苹果会对XR的理解有较大的差异，苹果也是希望加强这种差异来增强自己的竞争力和差异化
- XR的标准还存在很多变数，标准本身可能面临较大变化
- XR的开发接口会进一步分化，开发者面对更加复杂的概念和开发方式，这种情况下自研的接口封装会更快速相应这种变化，并且简化用户开发

6.4.15 ECS + AI

EntitiesBT

Behavior Tree

在原生的ECS框架下实现具有坚强依赖关系的功能是否很难，例如行为树，这种情况下，如果是像行为树一样相对固定的规则，我们可以像UE blueprint一样定义套框架，然后对于这个框架按太极一样的思路在编译层对代码做重新调整，当然要考虑重新编辑的便捷性，所以可能是拟娲运行时上的一个脚本轻量级JIT

6.4.16 Unreal blueprint

蓝图以及相应的很多编辑器及UI界面，本质上他们并没有简化逻辑的开发，他从两个方面简化：

- 固定类型的定义：对于一些特定固定的类型，例如blueprint 包含的那些类型，他们往往在OOP中有一定的关系，例如一般需要定义那几个类或者实例，那个类需要引用

那些类的实例，以及怎样对这些实例进行初始化，对于实现这些blueprint对应的功能，他们的这种通用结构被设计出来，否则那个开发者还需要定义自己的对象模型，每个人定义的可能很不一样，相应的内存管理，等等都很复杂

- 对应功能的初始化，成员变量复制管理等是很繁琐的事情，按照蓝图的编辑器，他实际上把流程固化了，因此这些基本的流程就简化了
- 剩下用户需要做的就是针对固定结构和架构下的具体某个函数进行代码编写

坏处是整个逻辑都是基于代码的，并且无法自定义，存在一定的冗余，如果不采用它的模式，也可以自己从零实现，那就需要实现整个模块的架构，对象之间的管理等，代价更大，但自由度更大，例如可以使用数据驱动，UE的流程并不一定是最好的，或者对某些特定类型或者环境下不一定最优

蓝图并没有简化逻辑的架构，他相当于UE帮助打了一个架子，开发者对其中的方法进行填充，整个还是面向对象的思路，对于Epic来讲，它需要去实现大量这样的架构，所以它的功能比Unity要复杂得多，但这些实现对于特定的一个游戏并不一定是最优化的，他会牺牲性能来换取这种简单，因为通用的架子往往不是最优的，而它面向对象的深度实现导致代码也很难优化，例如架子里面存在大量交叉引用

像Unity因为没有提供众多这种深度定制的架子，反而能够容易去实现ECS这种优化

相对于Unity，有更多逻辑层面的封装，类似定义了一些特定类型的范式，但这些范式本身是按照OOP的方式定义的，因此他的范式是实现了一些特定的功能，而不是定义了一个框架，所以UE要学习更多的知识，很多知识就是关于这些特定范式的

除了范式本身，Blueprint 的另一个强大的功能在于它把整个一个代码块或者一个复杂的子模块系统，打散成多个以方法为单位的可视编辑模块，这样是的修改单独的模块更加简单，不需要关注函数的输入输出，不需要在代码中寻找修改函数的入口，也不需要引入一些变量的引用、读取或者修改繁琐的代码

6.4.17 Pixar

皮克斯科技与艺术结合

科技与艺术结合的结果是什么

6.4.18 Gaia procedural-worlds.com

程序化静态内容生成应该还是比较成熟的架构了，后期应该能够容易开发，到交互更难

交互的设计也要是程序化的，当然由于ECS本身是组件化的，没有操作顺序依赖，所以这个问题不大存在，但是当需要对静态内容设计多个修改时，操作步骤的影响就出来了，程序化的好处是directable，它简化了对操作的任意修改，传统的软件就是遵循规则和步骤的，这样如果有些历史操作修改了或者需要修改，往往会影响其他大量修改

但脚本只是针对单个物体的行为，整个场景的结架构设计还是应该尽可能的程序化，而Roblox 没有这样的机制，程序化的场景设计使得通常可以更高效地设计大环境，然后精调小物体，而不是对场景每个物体都要独立摆放和设计，因为大环境通常都有一定的随机性

6.4.19 Google Maps API

6.4.20 Procedural content generation

传统的DCC使用deforming, cutting, merging 等代替对三角形的直接操作，从而简化内容创作

PCG使用户专注于用于生成内容的程序化算法，而不是底层繁琐的内容操作，这种算法更符合人的逻辑

PCG的计算过程比较复杂，更适合PC端，移动端要专注玩法部份

答主对生成算法的理解就好像某些时期对火药的理解：用来放烟花的东西，需要研究更好的火药么？不是算法够了，而是设计者不知道设计目的为何，对算法没有要求。游戏核心设计绝不是剧情、场景，而是玩法。玩法是什么，就是给玩家有意思的问题，让玩家解决问题。用算法提出的问题的难度，和解决问题难度是不一样的，玩家即使知道了你如何生成问题，但是他现在也不知道如何有效的解决问题。这就是著名的NP/P问题，我当然知道俄罗斯方块随机生成的方块随机算法，但就没有一个高效的算法去解决俄罗斯方块拼接问题，数独的生成算法很高效，解决数独的算法很慢，生成地图的算法高效，访问地图上的每一个点的哈密顿回路问题就难爆了。为啥觉得生成算法没用，因为不知道要构造什么问题给玩家，设计目的到底是什么，而是漫无目的地去生成那些花花草草，和用火药放烟火一个道理，本来就不是设计的核心，再怎么提升技术也只是装饰。

俄罗斯方块就是例子，真正把算法生成要用于玩法上，就需要了解P/NP问题：用简单的信息是可以构造复杂问题。其实MOBA游戏里，一个时间段的走位规划也是NP问题：哈密顿回路问题，如果要访问和侦察地图上多个战略要点，如何才能走最少的路，消耗最少的时间，达到战略目标。有时候这些战略要点还是其他玩家影响下生成的。

6.4.21 casualcreator

6.4.22 Scalar, 不鸣科技，微服务化

When it comes to cloud technology in gaming, most people associate it with game streaming. However, Scalar is based on cloud computing, which Romell as explained, is quite different from the term we've heard about so often. "Cloud streaming is a distribution model; it improves people's access to games, but it doesn't change, in essence, what games are, or the quality of them. The game is still being run on a single-processing machine placed remotely and then streamed via the cloud to your screen," he said.

"Cloud computing – what Ubisoft Scalar enables – means the processing power for a game isn't tied to a single machine, but a decentralized computation system. The processing is taking place in the cloud. This eliminates the limits of local hardware for players, improves the quality of games, and opens up new possibilities for game developers."

更新不会停服

把游戏引擎的多个组件，如physics, AI等都转成微服务，然后单独在云端计算，所以理论上不受限制，传统的引擎把整个引擎在一台机器运行，由于共享整个场景大量数据，因此不好并行计算

3月17日消息，据外媒报道，在当地时间星期四的GDC演讲中，育碧斯德哥尔摩的总经理Patrick Bach、技术总监Christian Holmqvist和首席技术官/技术总监Per-Olof Romell公布了一项全新云计算技术，该技术被称为Ubisoft Scalar。他们声称该技术将创造全新的游戏类型。他们表示，这项技术将使育碧能够制作比以往更大，更复杂的游戏世界，这些游戏世界可以实时更新，并由大量玩家填充，从而创造新的社交体验。关于使用该技术开发的新作。Bach表示：“育碧斯德哥尔摩正在研究与Scalar一起开发的IP，目标当然是充分利用这项新技术的所有可能性，但现在谈论这个还为时过早”。

6.4.23 Google Tilt Brush

太偏底层，属于基础创作，生成最原始的Mesh，问题：

- 由于控制精细度不够，所以网格和材质都不够精细
- 属于基础创作，那个物体都要从无到有创作，或者基于一些基本的几何体进行创作
- 无法生成复杂几何，更多是概念上的感觉
- 并且这些作品实际上很少有被真正使用或者被当作艺术，因为创作很受限

如果要提供更精细的控制，则面临：

- UI太复杂不好操作
- 手势控制的精度误差比较大

这类创作的作品通常只是半成品，可能作为一个初始场景和概念，然后在PC上在进一步精细化调，但如果是这样，在PC上有更多的方式生成这样的概念

如果直接使用，这类场景通常没有太大用处，只有静态的东西，没法编辑动画，当然如果要在XR设备加入动画编辑，那又是另外一个很复杂，几乎不太可能的事情

所以，在XR设备上，不能直接创作原始几何，因为这样：

- 这样几何是静态的，没有动画
- 太简单

它只能是基于某些标准组件的创作，这些组件通常由PC制作，带有动画和一定的逻辑属性（因为XR上也无法编辑逻辑），并且在XR设备上交互的是PCG的方式，这样使得交互需要的并不是精细位置，而且PCG的参数，这样参数的空间和范围小的多，同时能够生成更多内容，而不是仅仅一个基础几何

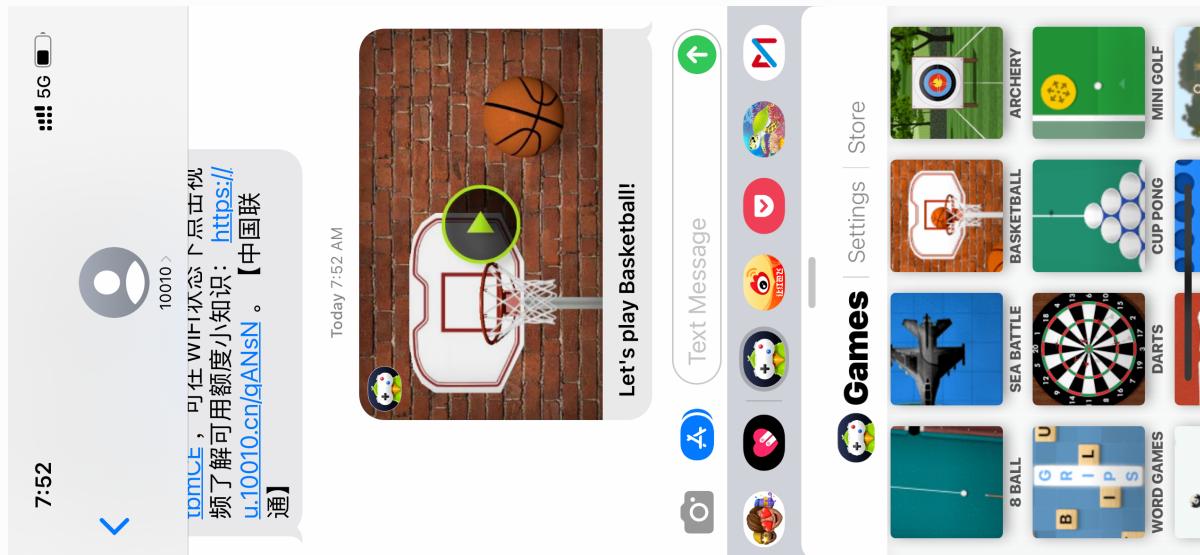
6.4.24 GitHub

多人协作的典范，在协作中体现贡献度(^hello)

比如你发现某个Creation 存在不合理，提出修改

6.4.25 Stechfab

6.4.26 Game pigeon: games for iMessage



6.5 应用

6.5.1 堡垒之夜



Game theme, starter island

组件会随着theme变化

相比堡垒之夜只从模板创建，RW具有能够让用户创造模板的能力，灵活度和可表达能力更强

Accolade 自动控制XP的获取，自动计算平衡，根据时间

资源是引用内部，需要游戏包内登录时下载，是否无法用户自己动态添加

堡垒之夜是实现了固定一套资源和逻辑的数据化，现实世界比游戏规则更复杂，不管是逻辑还是场景，交互，所以必须要支持编程扩展能力

例如堡垒之夜里大部分资源外观是不能修改的，但是生活中的设计往往都设立照片编辑，自定义一些内容，因为他是固定的类，无法组合：

- 无法通过减少或增加功能来修改已有物体
- 无法通过组合创建一个新的类型，新的功能
- 也没有办法创建一个新的资源，所有都依赖于官方构建一个新的类
- 每增加一个新的资源，都要创建一个新的类型，尽管他们有复用，代码会重复，用户需要理解和记住的类型非常多

所有东西或者大部分东西都被视觉化，然后数据驱动，修改设定好的属性

堡垒之夜简化了树形层级结构，所有内容都是扁平

6.5.1.1 定义交互类型很重要

堡垒之夜有122种devices

6.5.1.2 多人在线服务

音频的重要性

6.5.1.3 私密社交

I, I, couldn't be more pleased with the situation in Fortnite. Fortnite is the most positive social experience I've ever interacted with.

Um, and it's true that there are negative people out there sometimes, but the far majority of encounters are positive. Um, and also the far majority of social engagement on fortnight isn't with random strangers. It's not what many to many with millions of people are participating. It's players together with their friends, talking with their friends, kind of as an isolated group, uh, wandering through a much larger outside world. And a lot of the decisions we've made in the game have really contributed to the positivity here. One is that we have voice chat, so you can chat with your friends, but voice chat only works with people in your squad that either you're explicitly friends with and you explicitly joined up with, or you're friends explicitly joined up with.

And, you know, we really are innately trained to, uh, you know, in ordinary circumstances, respect people when we're interacting with them personally far, far more than when we're interacting with them with text. And so I think this is an area where the matter verse will have a major advantage overall. So O other social media, it's that inherently by being focused on small groups and actual friends, engaging in a much larger outside world and in carrying all of the emotional content of voice and perhaps even facial capture in the future, uh, it'll be a much higher empathy platform, only much less subject to abuse where, you know, one nasty action affects millions of people, uh, because of curation.

So, so I think we have a lot of positive things to be excited about there. Um, but the key point, the key challenge for this new medium is that to succeed anything that causes itself, the metaverse must actually be better than all other experiences competing for people's digital time. And that's a massive challenge. This means the metaverse needs to be better than an hour on the metaverse needs to be better than an hour on Facebook or Instagram or an hour on YouTube, uh, or an hour on Netflix.

Um, it'll be actual interactive objects, actual experiences, and actual engaging things and not just ads forced upon you. And, you know, I think we can completely escape an advertising based business model if we take this approach that the only way you ever get to see a commercial thing is if you decide to see it yourself, because it's really cool. And I think we'll see a whole new level of competition among brands to surface really awesome 3d content. Um, and you know, I think you can look to Fortnite and some of these other games is pioneering. A lot of these really exciting engaging non-advertising based, uh, mechanisms for exposing stuff.

6.5.1.4 Verse Language

Now, the next question is about programming model, because if we want to have this huge shared experience with many different types of games, other entertainment experiences, or, you know, any sort of experience at all, what you're talking about is a huge amount of user generated content in the form of 3d assets and also a huge amount of user written code, um, uh, per perhaps an unprecedented amount of user written code. And, you know, there's already some evidence of models like this working.

Um, I think the web with Java script is sort of a microcosm of this. Now the web is a much simpler programming model than the metaverse will have to be because on a website, all of the code that's running in Java script on that site is nominally under the control of the website operator. You can decide exactly what code runs and you don't ever have to deal with random user code being injected that might interact with you.

Um, uh, so the web BES, a closed world programming model, whereas the metaverse will need to be an open world programming model because the goal, it can't just be another app store, right?

You can't have a thousand different experiences and you can be in one at a time. And when you're in that one experience, it dictates everything, right? The, the metaverse has gotta be about interoperability of user creative objects of all different types, right? Because, uh, besides having some core game experiences in

the sort of place you're also going to have, you know, the equivalent of Facebook pages, uh, for every object that, that exists in the physical world, you know, uh, like Ford creates a new car, they're going to want to unveil the car, um, as a user drivable object, right?

Uh, in this virtual world. Um, and that's going to be a really interesting and more powerful way to unveil a car than to just announce it on your Facebook page.

Uh, in this virtual world. Um, and that's going to be a really interesting and more powerful way to unveil a car than to just announce it on your Facebook page.

The Facebook page, you can have text and video, but in the virtual world, you can actually get in the car, you can go around it, you can look at it from all different angles. You can open the doors, you can drive it, you can see how it handles. Um, and you know, you can have a, a huge set of interactions that are much more interesting and organic as it's not just a, an advertisement, uh, for a product, but also something that you can experience and have fun with. And I think we need to look at, uh, all of the crossovers that have occurred, um, in recent years between games and, um, and other brands as kind of an indicator of where the metaverse will go.

6.5.2 ROBLOX

Roblox社区太封闭：

- 不具备开放能力，没有平台开放的模式，别的应用无法调用
- 开发者之间很难共享，没有组件生态，整个开发模式还是传统封闭的app模式，每个开发者单独针对自己的app独立开发，共享复用能力不够，只是构建了一套自分发机制

相同点：

- 所有资源都是云端的，这使得多人协作更简单，不管是对于开发者还是对于用户，都没有本地资源数据，全是云端的，资源上传后Roblox 有个内容审核流程，大概几分钟
- Roblox packages, 可复用的游戏资源，游戏可以实时同步到最新版本，Roblox 的资源分享只在指定好友和组之间，切版本号相对简单，每次提交自动生成一个V1, V2, V3...之类的版本，仅能根据时间判别，功能很有限，不具备大规模协同的基础

- 多人协作编辑：用户有group的概念：My packages和Group packages，有Collaborators的概念，他们可以编辑游戏，其他用户编辑时，选中一个物体会带不同的颜色，对于脚本，当其他人在编辑时，会有颜色提示，但是仍然可以编辑，编辑完以后会有草稿存储，提交之后可以进行合并，可以与服务器版本进行比较，解决冲突之后再提交，也可以回滚，类似版本的概念了
- Data Store用于存储需要持久化存储的数据，只能在Script而非Local Script中调用，按字典的方式存储，有版本号，可以用于一些问题定位和支持，旧版本的内容会在30天后被删除
- 引擎plug-in，用于增强引擎，marketplace，跟其他引擎差不多
- Client-service模式，所有之间传递的参数都是可以Replicated类型的对象，否则传递结果为空，例如Part就是不可以在Server和Client之间进行传递的
- 所有非直接操作的物体都是物理模拟的，物理模拟可以是client或者server计算，一般靠近用户的地方会首先在该用户的client device计算，否则在server端计算
- Roblox在简化3D内容的层面做的很好，这样避免用户向着最高质量的内容创作，比如用方块就能描述树，使得渲染在移动端压力也少了很多
- 数据隐私保护，内容版权

缺点：

- 脚本编程模式，基本上还是Unity传统的模式，主要不同在于天生联网
- 例如脚本的挂载跟场景层级或者具体物体有关，还是像Unity原来的方式一样针对game object编写脚本，脚本中通过script.parent引用所属对象，这天生就使得脚本与特定场景结构或物体耦合，不利于复用，可复用的脚本应该仅关注数据，而不是实际的3D物体，这种数据是逻辑数据，他就使得逻辑和表现分离出来
- 比如对于一个UI按钮的点击事件，还需要写
`script.Parent.MouseButton1Click:Connect(function() end())`
- 脚本就等装的主要是结构，有点类似UE blueprint，或者说定义了一些规范，一些结构，这些规范大多数针对client-server架构的一些调整，没有像Houdini一样有些更多逻辑层面的封装
- PNC/Weapons Kit，有一套固定模版，有点类似于UE中内置的很多系统，平台提供了这些相对比较固定系统的架子，开发者往往修改的是：增加或移除某些组件；修改其中一些参数；或者基于架子代码进行定制修改。其中如果是修改的是一部分函数功能，UE提供了更好的定位方式，Roblox则更容易破坏整个体结构，但Roblox提供了基于模版代码进行定制的能力，灵活性更大，但是难度也更大，例如需要引用、读取和管理变量，寻找入口函数，处理输入输出等
- Configuration，对于NPC kit这样比较复杂的对象，由于有许多公共属性被很多脚本访问，Roblox建议将这些公共属性存储在一个value object中，用一个Configuration container封装，这是典型传统的OOP思想，其中带来的问题有很多，比如多个对象的指针引用，比如如果某些变量被删除其中一些脚本可能不工作，也可能移除了一些脚本而某些变量压根就不被使用，核心问题就是数据和逻辑之间脱离了关系，使得需要额外小心这种关系的维护

- Roblox 中的复用问题一方面通过ModuleScript，知识代码级的复用
- Roblox 的多人在线用户数还是有限的15个人，所以还是采用比较传统的多人同步架构，没有充分发挥现代云计算的能力，比如微服务架构，允许不限制的人数

Roblox 的核心优势在于云原生，他可能会自动处理很多同步问题，例如在Script中调用 Instance.new就会在workspace中创建一个Part并自动同步到在线场景，大部分的脚本都是通过Script编写的，里面特别是对Workspace中part的修改会自动同步到端侧

跟Roblox的最大区别在于，我们需要面向C端的用户，用户可能是不依赖于PC的，而Roblox的整个生态还是比较依赖于传统的游戏开发模式，其创新在于云原生和分发模式，为了实现在C端创作，需要：

- 运行时即是创作态，它比较少有编辑态的概念，像Minecraft 那样
- 需要在逻辑层面做更多的架构来支撑C端创作，有点类似于：Roblox + Houdini，而由此衍生出来的技术和架构要比Roblox 复杂得多，但其结果是会比Roblox 在创作层面更大量的普及

6.5.3 NIANTIC

6.5.4 SNAPCHAT

6.5.5 Meta

6.5.6 Omniverse

6.5.7 Minecraft

强项在于基于像素块，可以自由组合，自由度大，可以构建任意结构的场景或物体，不依赖于DCC输出，因此真正的低门槛

缺点：

- 无法使用强大的DCC输出
- 由于像素块松散组合，通常无法对物体级设置玩法
- 每一块单独构建，手工量极大，当然有些尝试用一些DCC输出的场景体素化后作为输入

Minecraft pc 编辑器

Minecraft MOD

Minecraft 的所有内容都在本地，进度需要自己备份，分享的内容需要自己安装在本地特定的文件夹，网易的版本会做一些联网购买

使用固定的文件夹结构，很多内容混到一起，管理复杂度高，容易造成冗余资源，不方便多人协作编辑，例如每个独立的json都引入ID，这就意味着删除对应的资源还需要解析json文件，显然不可能，这使得备份也会拷贝冗余文件，如果购买了一个Pack，则在新的创作者必须全部导入该Pack，这是传统UE和Unity那种传统的本地文件资源管理方式：

1. 按文件夹进行资源管理，需要开发者自己区分文件夹内资源之间的依赖关系，容易冗余
2. USD是按照资源进行管理

而RealityWorld使用更加先进的USD结构

Molang: 为什么不直接让开发者写脚本？

很大的原因可能是不方便管理，因为Minecraft完全限定于数据驱动，开发者能修改的是两种：

- 组件的属性
- 组件的组合形成新的Entity

其中后者通过json的定义实现，而前者是直接在json中的对象进行赋值操作，那么如果要使用单独的脚本文件，则会涉及的数量非常大，这些脚本文件怎么关联，如果需要手动关联就引入了复杂性，例如在Roblox中需要将脚本手动关联到对应的实体，而实体之间往往还涉及层次路径结构，就会进一步复杂化，Minecraft则直接将脚本写在属性赋值的地方，简化了很多东西

但这同时也意味着开发者无法自定义行为方法，他只能是对固定的结构的值进行修改，而无法定义新的逻辑

为了访问系统内存中的游戏状态与数值，Molang提供了大量的Query function进行查询

因此Molang是一种基于表达式的语言：expression-based language

6.5.8 Wilder World

Wilder World

Liquidity

- One of the biggest problems in the NFT space is that the best content is reserved for the uber-wealthy. We are flipping this paradigm on its head by enabling fully fractionalized NFT ownership, which will not only drive more capital into the space but will make it available to a much wider audience.

No Artist Fees

- Other platforms charge artists between 15% and 30%. We consider this is an antiquated way of thinking, older world not Wilder World. Instead, we have designed our native token to create value for all participants while redistributing wealth directly back into our Wilder community. There's no middleman or platform taking a cut of the artist's hard earned reward.

传统没有现实价值支撑的NFT，大部分的价值来源于，有点像传销一样，转卖，早期的玩家转卖给后续的玩家，赚取差价，而后续的玩家要想赚钱，必须不断有人接龙，然后一旦到了某些不可思议的价值就不会有人接龙

7. PROGRAMMING LANGUAGE

7.1 SCRIPT LANGUAGES

7.1.1 SkookumScript

SkookumScript uses a **multi-pass compiler** to determine what files and components are needed, and automatically manages them in the memory of the parser and runtime, so aspects such as dependencies are always up-to-date.

7.1.1.1 Time-flow logic

Coroutines and methods

Commands that may take time (multiple frames) to complete are called *coroutines* and have identifier names that must start with an underscore `_`.

Commands that start without an underscore such as `println()` complete immediately (within the same frame) and are called *methods*.

7.1.1.2 Conditional flow control

7.1.2 Lua

It provides "meta language" features. You can implement object-oriented structures, or pure procedural functions, etc. It has a very simple C interface, and gives the engine developer a lot of flexibility in the language itself.

Artists tend to love Lua too because it's very approachable, with plain and forgiving syntax. If your codebase is C or C++, I would highly recommend it.

It has good runtime performance when compared to other scripting languages like Python. (...and it has full support for closures.)

It has a small memory footprint (approx 150k), it has excellent C/C++ bindings making it easy to add new game specific APIs, it is easy to pick up, it is flexible - i.e Has elements of OO, imperative and functional - none of which are mandatory, it has good buy-in from mod community from games such as WoW etc.

简单的说，register-based的指令格式设计把stack-based的指令中分几条指令要完成的事情用一条指令搞定了，快当然是快了，难度也加大了。

另外还有一点上面的回答中似乎没有提到，Lua使用的是一遍遍历就生产指令的方式，学过编译原理的，大概都能知道一般分两遍遍历，第一遍生成AST，再一遍遍历AST生成指令，而在Lua中是直接跳过了AST指令这一步的。

还是那句话，快是快了，代码的实现难度也大了些。最早的Lua解释器，也是使用lex、yacc这样的工具来自动生成代码的，后来为了提升性能，作者改成了自己手写的递归下降的分析器。这部分代码是我认为Lua代码中最难理解的一个部分了--因为它要一遍分析干太多的事情了。

我在阅读Lua代码的过程中，能充分感受到作者为了Lua在性能上的提升花费的心血，致敬。

抛开理论不谈，如果要在Lua中实践，我们到底可以做点什么呢？

我认为需要有这几个方面：

首先应该对Lua加强类型系统。Lua的动态性天然支持把不同的组件聚合在一起，我们把不同的Component放在一张表里组合成Entity就足够了。但如果Component分的很细的话，用很多的表组合成一个Entity对象的额外开销不小。不像C++，结构体聚合的额外开销几乎为零。我们完全可以把不同Component的数据直接平坦放在一个table中，只要键值不冲突即可。但是我们需要额外的类型信息方便运行时从Entity中萃取出Component来。另外，如果是C/Lua混合设计的话，某些Component还应该可以是userdata。

从节省空间及方便遍历的角度讲，我们甚至可以把同类的C Component聚合在一大块内存中，然后在Entity的table中只保留一个lightuserdata即可。ECS的System最重要的操作就是遍历处理同类Component，这样天然就可以分为C System和Lua System。数据的内聚性很高，可以直接区分开C data和Lua Data。

然后、就是方便的遍历。ECS 的 System 需要做的就是筛选出它关心的 Entity，针对其中的 Component 做操作。如果需要筛选结果大大少于全体 Entity 数量，遍历逐个判断就会效率很低。好在在 Lua 中，我们可以非常容易地做出 cache，只需要遍历筛选一次，在监控新的 Component 的诞生就可以方便的维护遍历用的集合了。

Squirrel

受lua影响最大，但风格是C/C++风格的，在lua基础上添加了class, array等

<http://squirrel-lang.org/#overview>

V8 JavaScript from Google

7.1.3 GameMonkey

This one is used by several teams. It's faster than Lua and better at threading.

没有维护了

7.1.4 Python

This one has been used in several games (e.g. Civilization IV).

It is very easy to teach to non-programmers/designers. It is even easier to pick up for developers since it essentially reads like pseudocode. Being dynamically typed is just one of the aspects that help to get people with little to no prior coding experience up and running fast with the language.

Possible cons:

- The C bindings for python are much more geared towards extending python with C, than embedding python in C.

7.1.5 JavaScript

7.1.6 TypeScript

7.1.7 SCUMM

7.1.8 Mono-script

The Mono framework is faster than most (perhaps all?) of scripting languages out there because it's not interpreted, and because there's a layer between the compiler and the instruction set, it allows you to program in a variety of languages including C# and dialects of Python, Lua and Javascript.

Possible cons:

- If you're doing console development (including iOS), JITting code is apparently out of the question because you can't mark data pages as executable. The IL it has to be pre-compiled to the target platform.
- Mono has license restrictions. You need a commercial license if you want to use it in an environment where the end user is not allowed/able to upgrade the Mono runtime.

7.1.9 AngelScript

7.1.10 Scheme/Guile

With guile you can have your own DSL (Domain Specific Language) just for your game. Once you get used to the parentheses and prefix notation, scheme is heaven to work with.

<http://www.gnu.org/software/guile/>

libguile

Guile also provides an object library, libguile, that allows other applications to easily incorporate a complete Scheme interpreter.

设计原则：

- 始终定位为一个扩展语言
- Guile使用保守垃圾回收， conservative garbage collection

- it implements the Scheme concept of continuations by copying and reinstating the C stack—but whose practical consequence is that most existing C code can be glued into Guile as is, without needing modifications to cope with strange Scheme execution flows.
- Module system, 它使得extensions可以与之前的模块共存

最开始是基于Emacs Lisp作为Emacs扩展语言的巨大成功，GNU Project提出一种希望可以对所有GNU 应用程序都可以实现类似功能的语言

1.5 Supporting Multiple Languages

Since the 2.0 release, Guile's architecture supports compiling any language to its core virtual machine bytecode, and Scheme is just one of the supported languages. Other supported languages are Emacs Lisp, ECMAScript (commonly known as Javascript) and Brainfuck, and work is under discussion for Lua, Ruby and Python.

This means that users can program applications which use Guile in the language of their choice, rather than having the tastes of the application's author imposed on them.

2.4 Writing Guile Extensions

You can link Guile into your program and make Scheme available to the users of your program. You can also link your library into Guile and make its functionality available to all users of Guile.

2.5 Using the Guile Module System

Guile has support for dividing a program into modules. By using modules, you can group related code together and manage the composition of complete programs from largely in- dependent parts.

Module之间是怎么通信的？完全独立吗？

3.1.1 Latent Typing

没有办法为一个变量定义类型，以及为一个表达式定义返回类型，所有的变量和表达式都必须在runtime的时候确定，一个变量的名字x只不过表示内存中的一个位置，同时由于变量没有类型，所以可以赋予新的类型的值

3.2.1 Procedures as Values

跟其他变量一样处于同一个空间，所以你甚至可以对一个procedure使用一个不同的名字

5.7 An Overview of Guile Programming

5.7.1.2 Four Steps Required to Add Guile

- First, 在Guile中represent应用程序对象，除非是一些简单的内置数据类型如数字，否则我们需要使用foreign object interface创造对应的Scheme数据类型，这些对象受垃圾回收的管理
- Second编写可以被Guile访问的operations
- Third, 在宿主应用程序中需要有一种机制能够调用添加进来的Guile方法
- Finally, 在应用程序的top-level，需要做一些结构调整，使得可以初始化Guile的解释器，以及为Scheme定义foreign objects和primitives

5.7.1.3 How to Represent Dia Data in Scheme

- 该表述必须能够被原始语言decodable，因为原生语言需要获取数据
- The representation must also cope with Scheme code holding on to the value for later use.
- 内存数据同时被C和Scheme访问，不能只是简单地使用垃圾回收机制

One resolution of these issues is for the Scheme-level representation of a shape to be a new, Scheme-specific C structure wrapped up as a foreign object. The foreign object is what is passed into and out of Scheme code, and the Scheme-specific C structure inside the foreign object points to Dia's underlying C structure so that the code for primitives like square? can get at it.

9 Guile Implementation

7.1.11 ActionScript

This is a hybrid dynamic/static typed language used to create Flash games, which can be widely distributed on the web. It is fairly well supported with libraries like Flixel, FlashPunk and Box2d.

7.1.12 mruby

7.2 ERLANG

Erlang 算不上冷门，至少你还知道名字，很多你连名字都没听过的才算冷门。（但是很多冷门的设计理念却非常先进）

Erlang 在高并发方面有优势这个说法，其实非常片面。Erlang 最牛逼的地方是它是目前唯一一个具备软实时（Software Realtime）级别的系统。Java 模仿不了，Go 模仿不了。当然如果你要用 C/Rust 之类来做是可以的，但是其实就是把 Erlang 再做一遍而已。

这个软实时指的是垃圾回收性能平稳。如果做语音类应用，需要网络传输过程不会因为 GC 回收导致延迟抖动，Erlang 是你的开箱即用的最佳选择，没有之一。

“听起来也没多牛逼。不就是 GC 技术的优化嘛。我搞个并发式 GC 算法不就行了？”——说这话的，只能说第一并不了解 GC，第二也根本不知道 Erlang 的恐怖之处。只能说明朋友，你对力量一无所知。这里不想展开八百字复读机式介绍。自己可以看看 Erlang VM 的设计介绍。你会明白为什么 Erlang 里的 GC 才是真正完全并行，绝无 Stop the World 可能，而且回收延迟柔性可预测的。这一切不是没有代价的，代价就是变量必须绝不能被共享，而且不能被修改。这一来 Java 之类的 C 家族语言还玩个啥，凉了。

另外一些回答里，看了一圈，其实很多也只是随便用了一下试试。说几个点：

1、Erlang 是官方自带一套静态类型分析系统的——**dialyzer**，你不需要完全标注所有类型，未标注的可以自动推导；官方建议你在所有项目里都默认使用它来检查项目，如果你遵循这个建议，那么你还能享受自动生成文档的好处；而且官方标准库里也都写了类型。

为什么 Erlang 没有把静态类型分析作为吹的点？

因为静态类型系统（编译期检查）其实有其局限性，特别是分布式系统下，两个系统 A 和 B，假设某数据类型做了升级，那么实际系统升级里，会出现 A 升级了，B 还处于旧版本的情况。这个时候还有个屁的类型一致。所以依赖于静态类型分析保证系统一致，只能对于单个非分布式系统比较好。对于真实的分布式系统，设计出发点根本不是类型一致。而是即使不一致，也要能容忍。这就是另外一个话题了。

额外提一句，Erlang 的类型系统是在不允许你自己定义新类型的基础上，却能够完美的满足你的类型要求的设计。说真的，没有人和我提过这一点，但是当有一天我突然意识到的时候，那一瞬间是极其震惊的……（想想 Haskell）

2、Erlang 自带源代码变换系统，这玩意儿用人话说就是，你可以对你自己的源代码进行变换。比如 Erlang 官方自己的 EUnit 库，它是一个单元测试库。它的原理是什么？实际上就是当你引用 EUnit 的时候，就会导致你的当前模块增加一个 `parse_transform` 标记。然后编译期就知道这个模块需要被外部重写。最终实际上是交给 `eunit_autoexport` 模块来处理。

这个机制不是特权。你自己也可以用。但是这个 feature 确实比较高级，比较少有人讨论。

前端工程师熟悉的 Babel 其实做的就是这件事。只不过差别在于，Erlang 直接把这个做到了内部而已。而且非常简洁。大部分时候都用不到这个。当时当你有那么一两个 feature 真的需要用牛刀的时候，你一定会发出卧槽太爽了的评价。

3、Erlang 的模块系统是我见过最人性化的，简单到小学生都能明白。你不需要 `import` 任何模块。你想使用，就直接使用。Erlang 会为你自动寻找并加载。朋友们，其他语言头部那一堆 `import` 怎么说呢，真的是脱裤子放屁的存在。因为 Erlang 的语法保证了，能够简单的扫描当前文件就能推导出到底使用了哪些模块。

模块可以在不停止系统的情况下安全的热升级。是的，热升级其实 Python、JavaScript 之类的用点 Hack 小技巧，也能模仿个七八分。问题是没有一个敢说“安全”。因为 Erlang 的模块热升级是多版本并存的。假设一个进程真正跑，它使用的是老版本模块。那么升级的时候，新进程会使用新版本。互不干扰。

即使新版本带来了新问题，你还可以无缝的降回去。当然，你愿意，也可以把老的进程干掉一些，直接强制到新版本。其他系统这么做实在太可怕。可是 Erlang 的进程是容错的，状态可恢复而且可升级的，所以这么做还是可行。

模块热更，只是应对一些局部小修改。如果模块间有复杂依赖，需要一次进行多个模块热更怎么办？放心吧。Erlang 有完整的方案。

4、其他语言里，程序基本上就是，一个主入口，然后调用其他第三方模块这样的设计。但是这个设计太简陋。Erlang 的设计是，整个系统是由一系列独立运行的 Application 组成的。没错，其实你只是在为 Erlang 这个系统里开发 Application。包括俗话说的“系统**标准库**”这种玩意儿，Erlang 里也是独立的 Application。

有何区别？每个 Application 都有自己的一个启动过程，自己的一组进程（构成监督树，具备独立的容错性）。相互之间运行时耦合是松散的。所以，A 和 B 两个 Application 你想运行在同一台计算机，或者多台不同的计算机上，代码有差别吗？没有。

你感觉到一丝奇怪的气味没。是的，Erlang 甚至有自己的 Shell 用来管理和控制这整个系统。而这个 Shell 里就是 Erlang 语言本身。完美的一致，简直是操作系统一样。

顺带一提，Erlang 是可以写脚本的，叫做 escript。原汁原味，保证鲜美。

5、一般语言的字符串处理，感觉很方便。但是很多语言内部是只能处理 Unicode 的某一种编码的（UTF-8、UTF-16BE 是流行选择）。如果想要随心所欲的去支持，就必须把字符串当作原始二进制数据处理。但是 Erlang 里根本没有这个问题。

这个展开说比较复杂。很多人抱怨 Erlang 里字符串处理好像不方便。一个重要的原因是，这部分的理解需要稍微深一点的基础知识（不复杂）。以后再展开说

6、Erlang 里面直接包含了几种设计模式。而且只需要这几种设计模式。是的，比如 Erlang 里是自带状态机模式的。说到这里……

7.2.1 Beam VM

大家都知道erlang要解决的问题是“高并发”和“分布式”问题，这样说有点太抽象。

具体来说，erlang在应用层和操作系统层之间又加入了一个细粒度的计算资源分配层（beam vm），这个分配层自动把计算任务分派到os (thread) 层。这其实是高并发处理中一个很理想的环境，计算资源可以更合理的配置。理论上可以做到动态扩大或者缩小所需的硬件计算资源。

有了细粒度的自动计算资源分配，很多时候就不需要在应用层去考虑这个问题了，减少了很多无谓的工作。

这不正是未来所需要的计算模式吗？

8. 拟娲哲学

8.1 拟娲哲学第一课

8.1.1 元宇宙的社会价值是什么？

在虚拟世界，价值由两部分组成：

一种是通过视觉、音效等给你带来即时的快乐；另一种是通过作品承载的故事、对世界的理解、个人的生活经验、知识等信息带给个人的精神力量，这种力量不能单纯比做知识，它更多是丰富我们的精神世界，但是这种丰富可以通过给我某些意识从而使我们在工作中创造更大价值，例如它让我们更加积极、勇敢等

虚拟世界价值的产生：

创造的过程和结果都产生价值，创造过程产生的价值相对于创作者自身，参见威廉莫里斯论著，当然除了创作的过程本身，创作的价值还有一部份来源于别人的认可，比如你创造的是一个完全无意义的人，除非你自己觉得很有意义，否则只能体验到自身对这个创造的体验，但如果你是预期它可以让别人感到快乐，那么这种预期以及实际的反馈会让你感到更大的快乐

在虚拟世界，快乐就是价值

因此，创作的快乐，不仅来源于创作的过程，更来源于作品被其他用户消费和体验的过程，包括反馈，以及改进和再创作

对于创作结果的价值

对应上面的价值

通过数字作品，特别是叙事性视觉艺术作品产生的价值，不光是这种上述的价值本身，她另一个重要的意义在于：表达能力

我们所有的事情一般通过文字形式进行表达，理论上任何概念都可以通过精准的文字进行表述，就像计算机程序一样，任何计算机对一段程序的理解都是一致的，然而人类语言不一样，人类语言的字面描述通常都带有一定的背景信息，同样一段话，不同背景信息的人的理解程度是不一样的，并且这种背景信息有时候不一定是逻辑上的知识，还有文化、艺术、生活经历等复杂因素，因此导致的结果就是，比如：

我告诉你要变得勇敢，这句话字面意思很清晰，但是关于勇敢是什么，他可能有很多解释，到底要做到什么才算勇敢，没有定义

但是我给你看了一部《指环王》或者《霍比特人》，你马上就能获得很多精神上的理解，这里面不光是电影本身包含了更多信息，他还包含了很多视觉语言、以及融入你在看这些诗句内容和故事时产生的自我想象力，等等，这些都是非字面的信息所能表达的

一个作品融入的不光是创作者的经验知识，还有很多逻辑，表达手法等等很复杂的因素

所以这就是创作，它是一种表达形式，它的表达能力超越文字的字面意思，这也就是艺术创作这种事物的价值所在

交流和社交产生价值

8.2 拟娲哲学第二课

8.2.1 RealityIS的本质是什么？

RealityIS的设计过程是从上至下的，即看到上面应用层的开放问题，然后找到问题的根源是编程语言的限制，然后再深入到编程语言以及计算机体系结构的机制，最后得出解决方案。

这跟一般的软件架构过程很类似，由业务层的领域需求，来引导软件架构的设计，只不过这里的“软件架构”深入到了编程语言这一层。然而传统的软件架构是解决特定问题，因此必然导致泛化性不足。

但是，当我们得到这套技术架构之后，再反向向上理解的时候，却发现它具有很大的通用和泛化能力，这一部分原因可能是因为我们的“软件架构”发生在语言这一较低的层次，并且没有改变语言本身的机理。

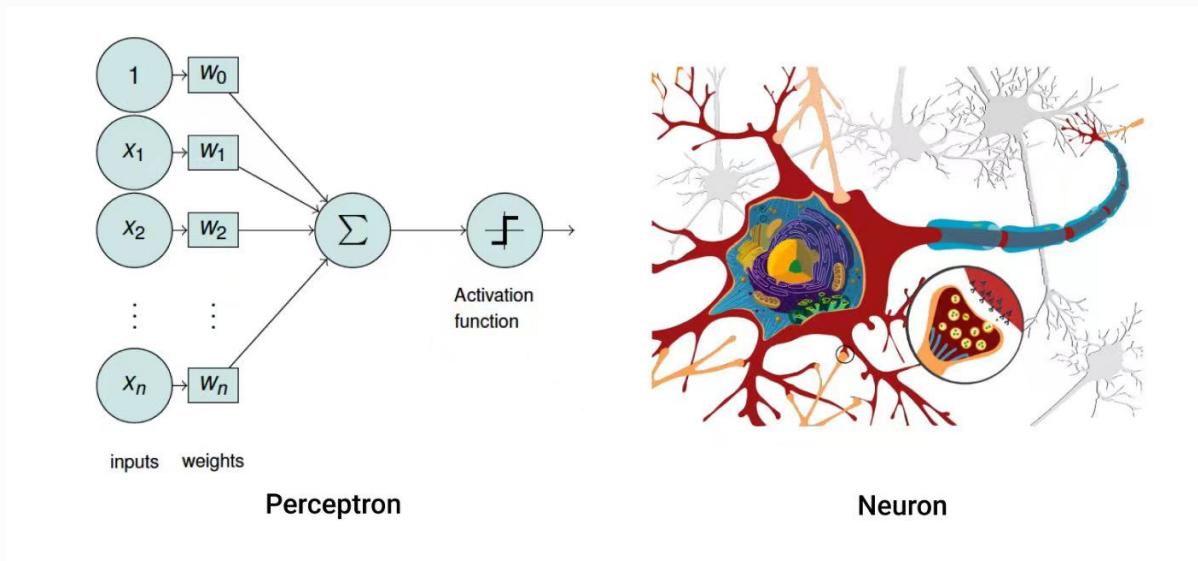
但是另一方面，也由于我们在设计过程中比较注重对数据的理解，当然这里也有如数据驱动、高性能计算、高度并发等技术需求所引向对数据的关注，也有刻意迭代地加深对数据本质的思考和理解。

所以当我们回过头来，对数据有了更深刻的理解和认知之后，会发现，从根源上，RealityIS的这些泛化性能力，来源于将整个程序开发和执行的机制，从传统以硬件处理器为核心的编译架构，转变为了更符合实际物理世界直觉的机制，这是一个根本性转变。

最终，整个RealityIS的能力和思维，都可以理解为是基于数据的编程模型。包括如解耦、并发、泛型、自我进化式的标准机制等等，这些本质上都是以数据为中心去思考才能形成的结果。

所以，它有一种偶然，也有必然；偶然的是我们关注到数据这个中心问题，必然的是数据为中心的概念是一套能够以真实世界类似的机制进行作业的规则。

将来，我们还会继续完善这一概念，最终，RealityIS将变为一个以真实世界的直觉和真实世界的运作方式类似的机制进行整个程序的构建和运行，这将是一种全新的计算架构。

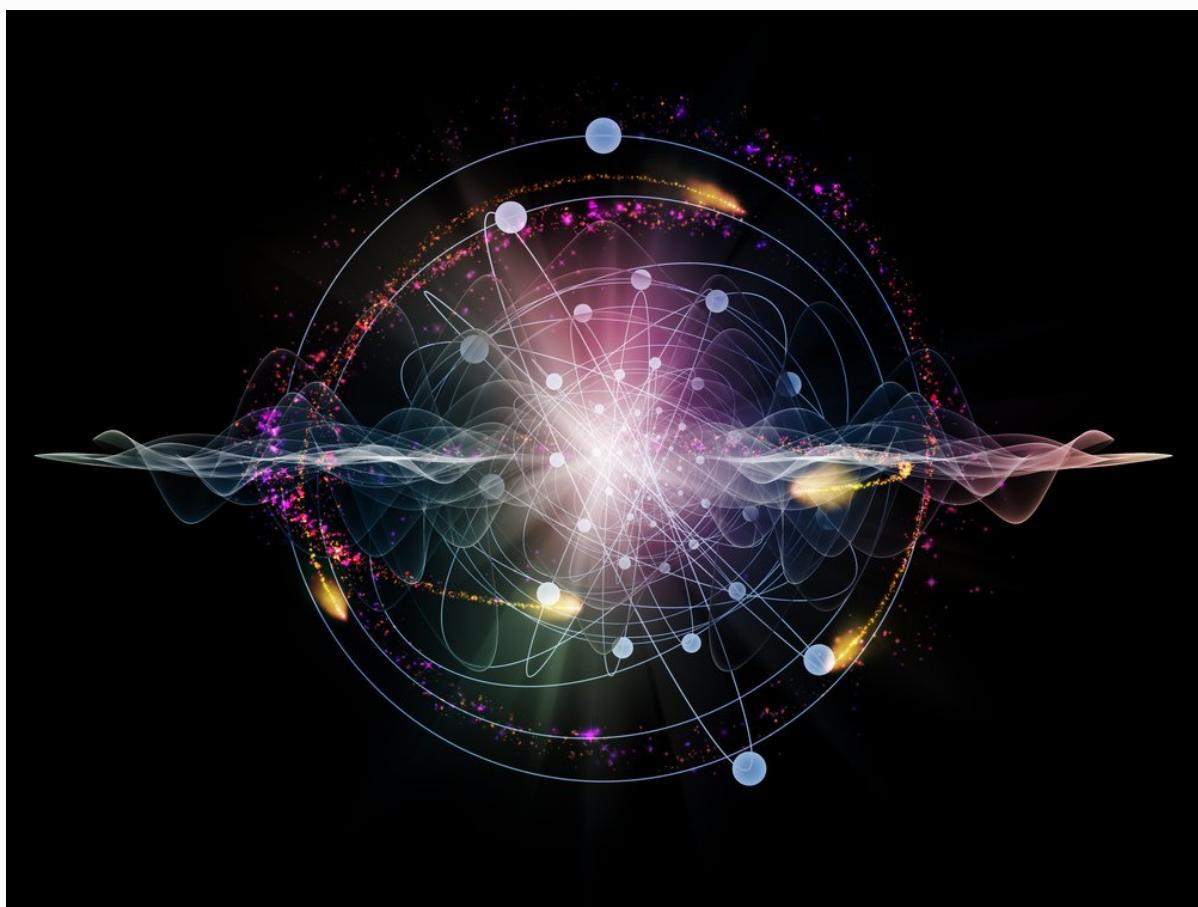


这有点像深度学习，它的很多理念来自于对大脑机制的思考，虽然神经元的机制并不一定是大脑实际运行的机制（实际当然要复杂得多），但是它可能至少是其中核心的一部分，或者说这种思考抓住了一定的本质，所以最终基于神经元这种简单的结构构建的深度学习模型能够在较大程度上模拟大脑的机制。

8.3 拟娲哲学第三课

8.3.1 标准及自我进化

一个不能自我进化的Metaverse就是一个游戏，这显然不是Metaverse的形态。此外，即便我们解决了多程序交互的问题，它只是增加了一个游戏内的系统会更加丰富。然而对于一个好的世界，这种丰富不仅仅是指数量上越来越多，而且需要在丰富上形成层次，甚至对于社会的运行机制，后者是更重要的，因为用户关注和需要的是有层次的信息，而不是更多海量可能存在大量无意义无价值的信息。然而，仅仅向其中增加程序的能力不能保证这种丰富形成层次。



当一个开发者向其中添加了一个新的程序，怎么能够判断这个程序的价值？并且是要通过用户的视角去评判这个程序的价值？

在真实世界中，社会进化的机制来源于两股力量：

- 少数优秀的人能够创造一些好的东西，这些东西不仅仅是指一个具体的实物，更可能包含一些结构、关系以及社会运作的一些逻辑，这些东西在Reality World就对应标准，标准的数据结构及其数据组合背后反映的是一定深层次的结构、关系和逻辑。
- 这些好的东西会被其他少部分人接触到，不管是地理位置上较近，还是熟人之间介绍等等，这部分机制在现实社会中往往通过广告进行加强。当这一少部分人使用之后觉得真正有价值的东西，他们会形成推广的力量，通过人与人之间的关系把这个有价值的东西推向更大的人群，如此，那些最有价值的东西被逐步挖掘出来。

上述这两种机制导致的结果：

- 人们会觉得创造东西会有价值，你有机会被更多人使用，从而为更多人创造价值，你的创作也有机会被更多人认可
- 人们会觉得社会越来越进步，幸福感更强，因为你感觉这个社会在进步，你越来越能使用到更好的东西

这种进化最根本的力量来自于社会个人，而不是少数中央机构。所以要实现这样的自我进化，我们一定要有类似的机制来释放个人的这种力量，而不是依靠平台，平台没法做这件事情。

8.4 拟娲哲学第四课

8.4.1 Reality World中的“市场经济”机制

即市场会决定哪些东西是更有价值的，这是与传统数字经济系统根本性的不同，传统的数字经济都需要由平台实现某种推荐或者排序算法，例如微博的信息，知乎的文章，淘宝的商品，抖音的视频，这就要求基于一定的标签，分类等机制，信息发布者需要去维护这种标签分类。

然而真实世界的经济却不是这样的，我们所有的一切不是由类似国家或中央的官方机构决定的，而是靠人们自己的选择，促进整个世界的运转。

类似真实世界公司之间的销售，产品越好卖的越多，售价也可以随市场调整。

而且这种机制促进作品的不断改进，就是iPhone手机一样，而传统的内容都是一次性发布，缺乏对原产品的改进机会。游戏也一般由于太复杂，发布后不会有大的改进。目前这些数字经济跟真实世界的经济都不一样。



可以认为它们都是“结构化”的经济，而不是市场经济。

真正的市场经济会促使和催生更多的好内容，更多的人参与。而传统的数字经济，都是少数人在参与或获利。

在真实生活中，每个人都在参与经济贡献；而在目前的自媒体时代，只有少数人在参与经济贡献，大部分都是消费者。

这有机会使得整个经济系统的活力更大：传统的数字化经济都是靠阅读量类似不准确的机制，在这种机制下创作者倾向于作弊买量，而不是创作更好的内容。此外，阅读量本身是个不准确的度量，例如用户可能只是打开了页面就关闭了，根本就没有深入了解对应的内容。而这种通过“实际使用”而不是“查看页面”转化而来对产品的经济定义，更容易促进用户进行更好的创作，就像真实世界一样。见4.3节更多描述。

8.5 拟娲哲学第五课

8.5.1 怎样构建大规模、大并发系统