设计的引言

- 背景：游戏策划在编写策划案的时候，需要非常详尽地考虑各个系统。但是有时仍难以一时间想出来全部的系统。在配表时也是如此，有时候写着写着就发现少了一张表。
- 目标：可以使用ECS的架构来辅助设计师查漏补缺游戏系统。

基本概念

ECS架构是指实体-组件-系统(Entity-Component-System)架构。在游戏开发中，每个基本单元都是一个实体，每个实体又由一个或多个组件构成，每个组件仅仅包含代表其特性的数据（即在组件中没有任何方法）。例如：移动相关的组件MoveComponent包含速度、位置、朝向等属性，一旦一个实体拥有了MoveComponent组件便可以认为它拥有了移动的能力。系统便是来处理拥有一个或多个相同组件的实体集合的工具，其只拥有行为（即在系统中没有任何数据），在这个例子中，处理移动的系统仅仅关心拥有移动能力的实体，它会遍历所有拥有MoveComponent组件的实体，并根据相关的数据（速度、位置、朝向等），更新实体的位置。

1. World：所有Entity都被放在一个World中，World可以对Entity和Component进行维护与查询，提供一些增加或删除Entity、Component的能力。
2. Entity：Entity只是一个容器，它的能力由它所挂接的Component和Component相关的System决定。
3. Component：Component只有数据，不同的游戏机制依赖不同的数据，这些数据由Component维护。有些游戏状态只有一份全局数据，这些数据放在一个单例的Component中。
4. System：一个System可以被理解为一个独立的游戏机制。它挑选出它所需要的那些Component，对它们执行相应的处理。
5. **Group**：其实就是组件类型的元组，充当filer的作用，用于筛选出带有某些组件的实体，以供System遍历。var group =list(Component)

ECS是DDD的架构，DDD：Data-Driven-Development。

优点

Component需要考虑的并不是用什么功能，而是有什么数据。

ECS架构的性能非常好，因为逻辑与数据进行了分离，所以内存连续性好。

模块解耦，组件之间不必知晓其他组件的存在

数据驱动，通过配置不同的组件来完成不同的实体

当游戏中有大量的墙壁或者某些物体的时候，就可以使用ECS架构，提高性能

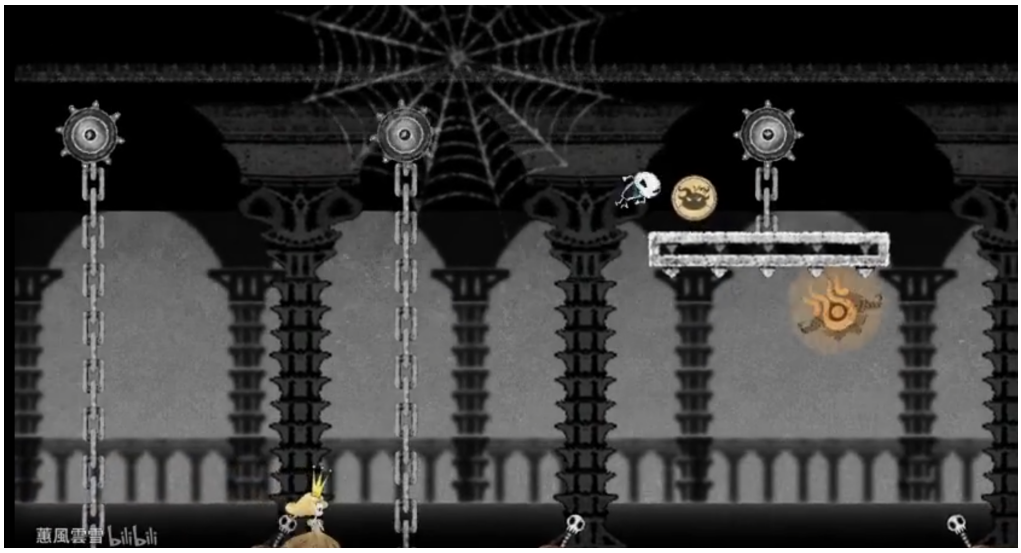
第一层抽象——底层方法论的设计

对数值游戏的本体论研究

目前的引擎主要是以设计数值类游戏为主，因此需要对数值游戏整体进行拆解。在拆解之前，首先要考虑数值游戏的定义。

这里的数值游戏指代以下几个方面：

- 以数据为驱动的游戏：指玩家在游戏中的成长主要以数据的方式呈现。而且成长是可以被数据衡量的，例如《勇者斗恶龙》系列，玩家的等级会不断成长。这里的成长方向必须明确，例如《密特罗德》系列，玩家会不断获得新的能力，但是这种能力无法被衡量。
- 以数据为目标的游戏：指玩家在游戏中是以某个具体数据为目标游戏的，玩家需要完成一系列连续数值或数据序列来完成游戏。例如《开罗游戏》系列，玩家一开局就会被给予一个目标，赚取10000万等。同样的，如果玩家的目标是收集到某个具体的物品也可以理解为数值游戏。例如《星露谷物语》中的黄金钟。要特别注意，这里的目标应该是一个连续的目标。包括离散型数据和来连续型数据。例如在解谜游戏《扣押（DISTRAINT）》中，玩家就需要根据一连串连续的解密数据进行探索。这些数据都是以某一系列bool变量控制的，虽然bool变量并不是连续的数值，但是它们彼此构成了序列。而解谜游戏《dokuro》则不然，它的谜题判断非常丰富，玩家的目标是将公主送到终点，这个目标不可以概括为数值。



这两点对着应游戏数据的两个核心点：方向与终点。因此数值游戏本质上可以看作一个方向明确的箭头。有方向和终点，都可以完成数据的引导。从而让媒体信息流向终点。

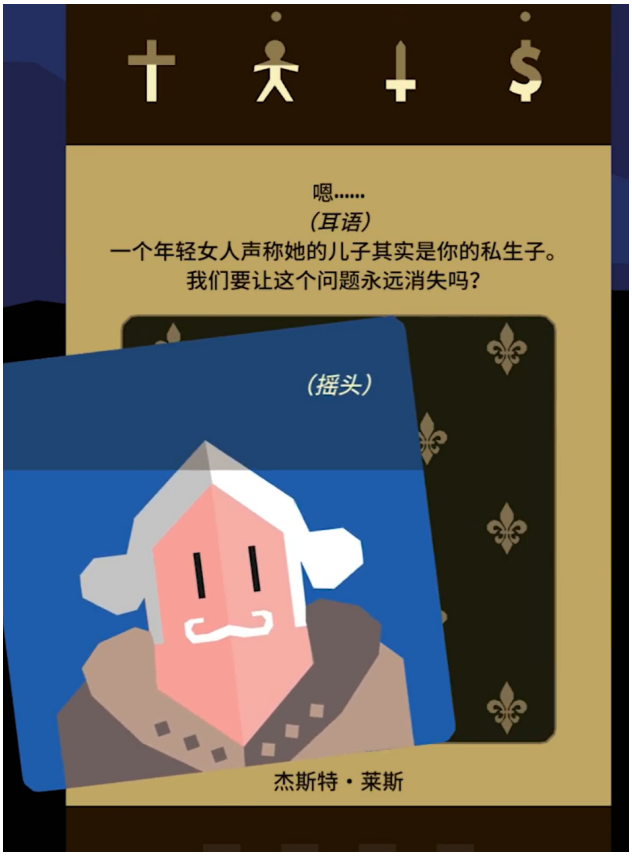
因此数值游戏和其他非数值游戏的核心点就在于，数值游戏是具有**数据驱动力**的。例如《黑暗之魂》系列，就并不是纯粹的数值游戏，因为数据的驱动力不足，玩家无法单纯通过成长来赢得战斗。同样的，《塞尔达传说——旷野之息》中，玩家装备和武器的提升是可以被客观衡量的，虽然玩家自身也会成长，但是和数值的成长并不冲突。

对数值游戏gameplay的拆解

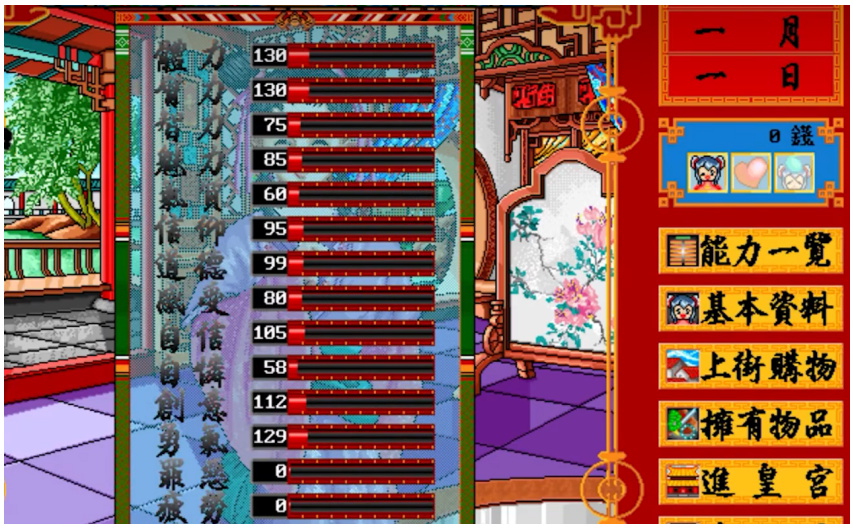
在抽象出了数值游戏过程和终点的箭头之后。下一步就是探究如何让游戏更加"有趣"。在进行理性主义分析之前，首先应该先用经验主义的眼光进行抽象。

数值游戏的gameplay存在以下几点：

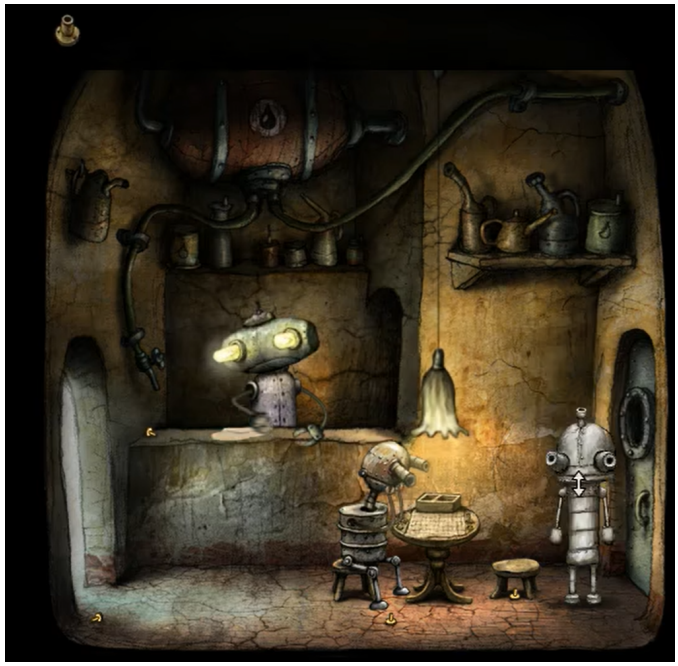
- 生存：指玩家需要维持一个或者若干个数值之间的平衡，不能让其过高或者过低。例如《王权》中，玩家必须保持宗教，民心，军力，金钱。这四个属性的稳定。过高过低都会导致游戏失败。



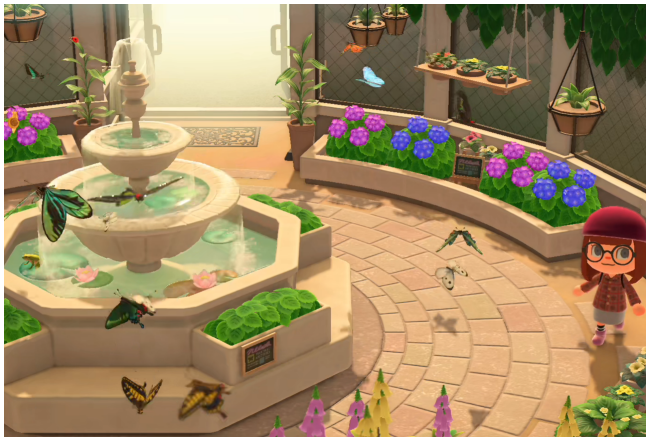
- 养成：养成是指，玩家需要从0开始培养某一个对象。该对象可以是人类，动物，物品等。例如在《皇后养成计划》中玩家就需要完成对女主角的培养。除了传统的养成游戏，SLG的模拟经营，和RPG的等级系统等都是养成的一部分。



- 收集：收集是指玩家可以完成该对象的收集，该操作通常要图鉴，或者完成一系列收集链。完成收集既可以是游戏的核心也可以仅仅作为奖励系统的一部分。所有的解密游戏的gameplay都是收集。例如《机械迷城》中玩家的核心就不断寻找下一个关键道具。



在《动物之森》中，玩家的一个核心就是完成博物馆的收集。



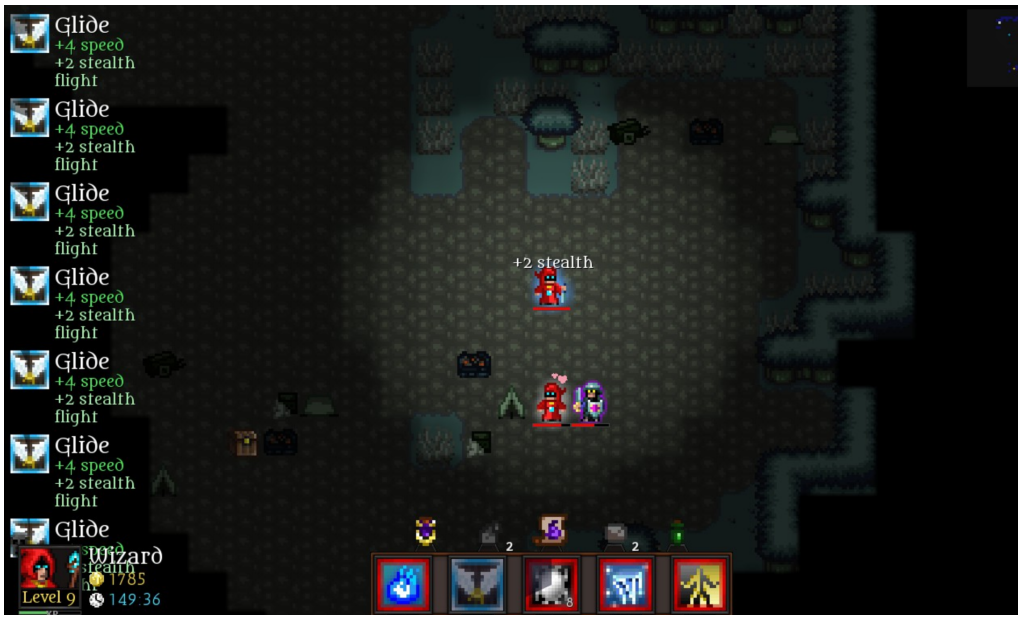
那么，根据我的理论，这三个属性实际上对应着数据的三个处理方式：

- 生存：描述了某一个数据的范围回调。在某个数值到达某一个范围时的回调。

- 养成：描述了数据的增删改查。
- 收集：描述了所有数据的关联性，这强调了多个数据之间的关系。例如解谜游戏的顺序关系，图鉴的拓扑关系等。

gameplay的派生

在此基础上，很多数值游戏又进行了新gameplay的派生。例如《主教之旅2》中，游戏就引入了“战斗系统”，“技能系统”等。



我们不妨进行一个归纳。

- 战斗系统=玩家的生存+等级的养成+敌人血量和玩家攻击的收集： 在游戏中，战斗系统本质就是玩家成长，然后攻击敌人。敌人死亡，玩家继续成长的play circle。敌人通过生存判定了死亡，那么玩家获得了养成判定的成长。
- buff系统=角色属性的养成： buff系统可以增加角色的属性。

以此类推，所有的数值游戏理论上都可以由这些机制派生而来。

基于ECS的数值游戏模块设计

以ECS为基础逻辑，不妨将引擎划分为两个部分：

- 组件（compoments）：组件只负策划案的数值部分
- 系统（systems）：系统只负责策划案的逻辑部分

在组件中，可以定义若干数据，在系统中，只需要完成对数据的处理。而具体的处理方式如上所述。

第二层抽象——对基本组件和系统的提炼

组件和系统的抽象

一个组件只包含其所需的数据。以战斗组件为例，战斗组件可以包含下面的值：

属性	数值
攻击	10
防御	20
暴击	30%

当某一个实体挂载了战斗组件，说明该实体可以战斗。

战斗系统的设计如下：

描述	
运算类型	二元运算，发生在两个实体之间
运算方式	b.hp-=a.atk
回调监视	null

也就是说，只要两个实体都挂载了战斗组件，那么战斗系统就可以被触发。触发之后，会使b的hp发生变化。

运算类型就对应着收集，运算方式对应着养成，回调监视对应着生存。

收集系统除了提供简单的二元关系，还需要提供网状关系和集群关系。这对应着解谜游戏和收集游戏。

例如简单的解密组件可以设计如下：

属性	关系链				
开始	(开始->打开门)				
开始	(开始->打开窗户)				
打开门	(打开门->走出门)				
打开窗户	(打开窗户>翻过窗户)				
走出门	(走出门->逃脱)				
翻过窗户	(翻过窗户>逃脱)				
逃脱	(逃脱->null)				

根据解密组件的内容，可以设计解密系统：

描述		备注	
运算类型	链式运算		
开始节点	开始		
回调监视	null		

简单的成就系统可以设计如下：

属性	数值	分组
开始游戏	10	1
防御	20	
击败100个敌人	30%	1

描述		备注	
运算类型	集群运算		
回调监视	当完成时：当完成50%时：		

第三层抽象——组件的具体设计

组件只负责数据的部分，引擎提供的仅仅是组件的预设模板，组件并没有固定的形式。

（参考帝国时代3的编辑器，提前预留好若干个事件和触发事件，完成游戏的拼接）

组件的分类

引擎中需要挂载多种组件，在设计具体组件之前，需要先设计一个合理的分类标准。

所有的组件代表着able。

例如战斗组件本质上就是说明这个实体是**可战斗的。**是战斗able。因此在下文的组件设计中，核心思路就是区分该组件是否是xxxable的

所有的系统都包含着生存，养成，收集。这三种基本逻辑。对于一个组件的功能也应该检查这三点内容。

- 数值是否可以发生变化?
- 数值是否存在回调?
- 和其他组件的关系?