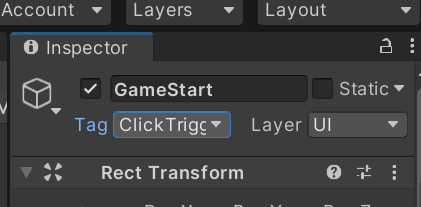
重构日志

2022-8-1

参考winform组件给主窗体发送消息

为每个可点击的物体挂载clicksender组件，参数为主函数所在gameobject

对于非代码控制生成的物体，想要动态挂载clicksender组件，可在场景制作的时候将物体的tag设置为ClickTrigger，这样在场景开始加载的时候即可自动为所有此tag的物体挂载此组件。

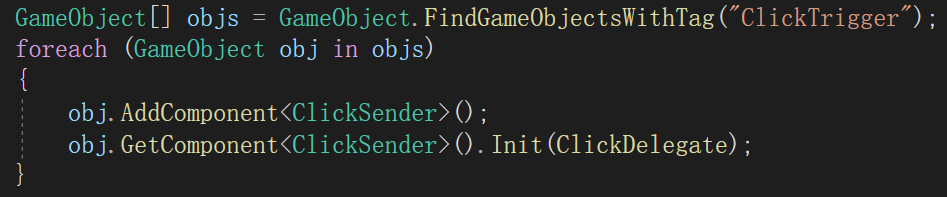


而对于主场景的存档按钮，虽然在不同结点下，名字要保持不一致（后续会改进），因为要判断是哪个按钮被按下。当然也有想过让按钮自己提供一套委托返回给主函数执行，但好像主函数控制一切的逻辑性没那么强烈了，和当初各个按钮各管各的差不多。。。



2022-8-8

突然想到重构情况下对于可点击物体向主函数传递消息是主函数提供一个委托，其参数为点击物体，返回值为点击物体，这样主函数就可以决定哪些物体可以向主函数发送点击。



而主函数发送控制命令，则是找到这个物体然后调用这个函数。也就是说物体【指针】是需要知道的，好像不难理解，想让对方的第三个角色受击，理想情况下可以：gamecontext·.player2.bucketManager.bucket[3].injuired()，也很合理。

但想了想消息管理，认真反思为什么那么混乱，关键是本应该解耦的关系却依然存在，它只应该有2部分：主函数和控制物体的【控件】，也就是说gamecontext·.player2.bucketManager这部分都应该是多余的（实际情况更为复杂【1】），只需要main（）和bucket，形成一对多而不需要中间的manager，player2甚至context。其中bucket应当**包含**所属player：player2，自身序号serial：3，这样在接收到消息的时候通过检查发来的数据包进行匹对（主函数向所有可控物体发送数据包，数据包中应当包含哪个物体以及要执行的操作，很像命令需要提前知道），匹对成功则该物体执行自身受击流程。这应该才是消息系统解耦的关键：去掉gamecontext·.player2.bucketManager.bucket这个冗长的关系。尤其是一些含糊不清，难以简单规定【范畴】的关系。

因此，对于一对一的关系：主函数对于AttackManager，不应该使用消息模式进行控制。这种【滥用】，很早以前大家也有所窥见。同时AttackManager作为主函数的拓展，也是必需的。

逻辑冗余也是混乱的表现之一：明明gamecontext·.player2.bucketManager.bucket[3]就包含了所属关系，而在bucket中却依然有play归属，自身序号等属性。

hh扯远了，既然解耦是一个很重要的理念（个人觉得不应该强求而破坏【封装】【1】），但该重构版本依然用的gamecontext·.player2.bucketManager.bucket[3].injuired()这个流程，所以。。。。。。所以怎么办捏，俺也不知道。

[1].重构版本首先着眼（考虑不周）的改进地方就是更加完整以及合理的封装，拉进主控和控件的关系；

2022-8-9

BattleScene主函数应该掌控哪些东西？大循环应该是游戏开始，进行，结束的流程控制，在游戏进行过程中要进行boss，player1，player2的循环控制，然后再对当player1回合时判断各个物体点击要干什么。那么游戏进行中的过程要不要再开一个monobehavior类封装起来呢，player回合内的点击控制要不要封装起来呢。封装起来的话，主函数就剩下一个游戏开始()，游戏进行()，游戏结束()的循环，这和主函数控制一切的理念是相悖的，单从主函数是看不见某个物体被点击然后干了什么。是否是过度封装呢。

2022-8-10

突然就想到工厂模式了，在游戏里好像并没有任何地方的体现，工厂模式主要是隐藏了构造的复杂性，游戏中要构造的有哪些呢？显然实体卡牌，战场上的角色，，，都应当是【构造】的，但构造的复杂性封装在了instantiate（perfab）中，然后再运行一下自己写的gameobject.getcomponment<XXX>.init()；当然如果perfab不包含一些控件则需要addcomponment。这样看来是可以写一个工厂来构造，考虑到调用组件方式，工厂就显得有些必要了。现在好像明白我想要的组件【主控】是什么样子了，它完全就是一个接口的实现！这个接口会封装一个游戏物体的所有行为。比如已经实现的卡牌缓慢移入到某个位置，它的实时检查自身位置是否逼近终点是应该在一个update（）里面，但它的触发函数，应该就是在这个接口里面！

这样，cardmanager的控制物体方式不再是以gameobject为核心了，而是一个实现对的外接口为核心（面向接口）：handcard，其中找这个物体就handcard.getgameobject【1】,调用移动动作就handcard.moveto(),调用攻击动作就handcard.attackanimation（）。。。相比较之前的gameobject,getcomponment<XXX>.moveto()优雅了不少。而且，对于其他非游戏的开发者，用一个实体（gameobject）再去获取组件（getcomponent）才能调用函数，真的很怪！尤其是gameobject的集合（数组，列表等等等）的使用。

构造的品类并不五花八门，五花八门的应该是各种名字，属性，但还都是属于一个类。如果火史莱姆，风史莱姆各自创建不同的类时，才有必要使用工厂。暂不使用工厂，保留考虑火史莱姆，风史莱姆，打手丘丘人，丘丘人单独建类再使用工厂模式，以便后续加入各种怪物。

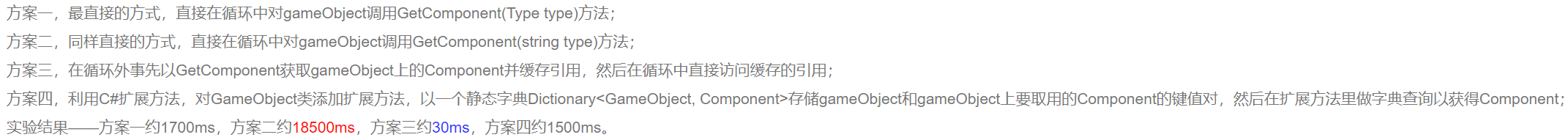
1. 这种调用方式重构前是有的，当时并未重视。

单例模式好像有些可以弄成静态类的说。。。

2022-8-11

Card的克隆模式还是不要了吧，每一张都是构造吧。但是初始化也很有可能需要消耗非常多的资源，比如charaCard可能要加载名字，id以外，还可能从加载技能或其他信息？（暂时想不出来）这些信息每次构造时都需要访问硬盘，确实可以克隆提升性能。

对于接口包含物体（handcard），获取方式：handcard.getgameobject()，对于更改位置，可以在handcard实现的类中缓存对transform的引用：handcard.gettransform();切忌使用handcard.getgameobject().transform。同理，使用自己写的移动组件也不应该用handcard.getgameobject().getcomponent<XXX>.MoveTo(target);而是作为类的属性来调用：handcard.MoveTo();这样做不仅仅是主观上的（学艺不精就不提设计模式了）规范考虑（handcard是主控），在性能上也很有必要：<http://hzhcontrols.com/new-364256.html>



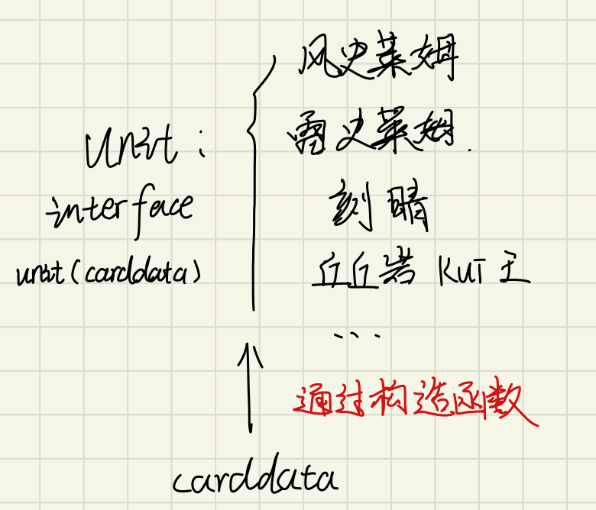


2022-8-12

更正一下，从主函数调用攻击管理器确实是一对一的设计，之前想的这种调用方式不要走消息管理，消息管理还是用在“所有物体监听同一个控制命令”这种一对多的情况之下。主函数中调用就直接attackmanager.instance.attack（），而不是messagemanager.send（attackmanager，resource，target）。这种方式从主函数里面看逻辑性更强，但耦合度也高。之前不知道咋解耦合，想了想就是去掉主函数，确实管理器并不“单一”，想反，一个物体可以对应多个管理器，这时候更像是一种[script]的做法，每个物体的行为都在自己内部，一旦被点击，其他管理器作为观察者各自完成动作。这和重构前版本十分相似，但还是要强调几点不同：  
1，不要使用一个消息管理中心来分发消息！各个消息管理器是独立的，不应设计这种“耦合”，通过消息管理中心分发消息，还叫什么观察者呢。  
2，不要设计以及使用一些很“内部”，很“直接”的函数，比如准备攻击时地块变亮，bucketmanager.instance().lighten()首先排除，sendmessage（target，“变亮”）过于“内部”，也排除。正确的方法是sendmessage（“开始攻击”），就这么简单，当bucketmanager观察到“开始攻击”就开始让其他地块变亮。这样，attackmanager也在观察到这个命令后准备攻击，audiomanager观察到之后开始播放选中音效。。。这样才是观察者模式，才是一对多的关系。

因此，一种是强逻辑的主函数模式，主要难点在封装、另一种是弱逻辑的【script】模式，难点在解耦。本次重构完全采用主函数模式，注重逻辑封装，忽略解耦。

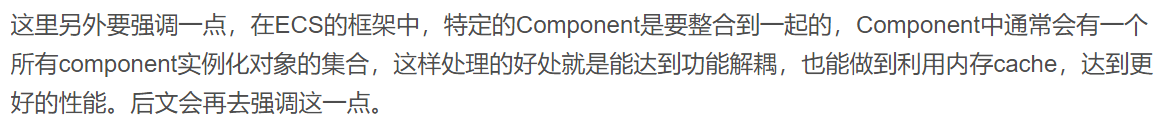
2022-8-20

赛季末冲分结束啦，继续干活，想到一个奇技淫巧：当从carddata转换成战场上的unit时候，都是在unit的构造函数里面判断是不是角色卡，是不是怪物卡，这个逻辑能不能写在carddata里面实现一个多态，只需要每个卡牌数据类重写一下强制类型转换函数就行了吧。这样就是要在carddata进行一个细分，而unit是个大类甚至是个接口：

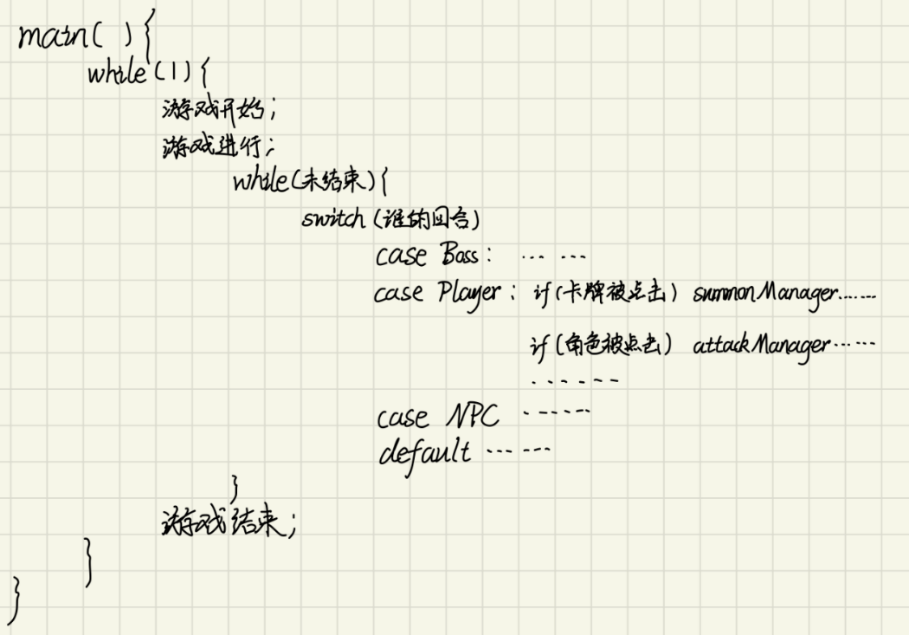
额画出来才发现unit构造函数也可以实现多态，就先这样吧data就不细分了。对于接口实现的选择，就用工厂了。

搜了ecs框架发现很多如出一辙的想法唉，entity不用多说就是gameobject；component也就是gameobject组件化，然后system就是各种管理器啦，不过。。。以前项目不也是吗，所以，俺还要加一点细节。

首先就是entity和component的联系，一个entity挂载很多component，但ECS的挑战在于怎么放进一个高度复杂的项目里。如何归纳和整理组件，如何控制组件的逻辑时序，如何处理组件之间的依赖关系和通信。通信是后话，属于是system的范畴。E和C的考虑首先就是要建立一个（包装），也就是一开始我想要的一个物体的“主控”（详见2022-8-11）。对于框架而言：

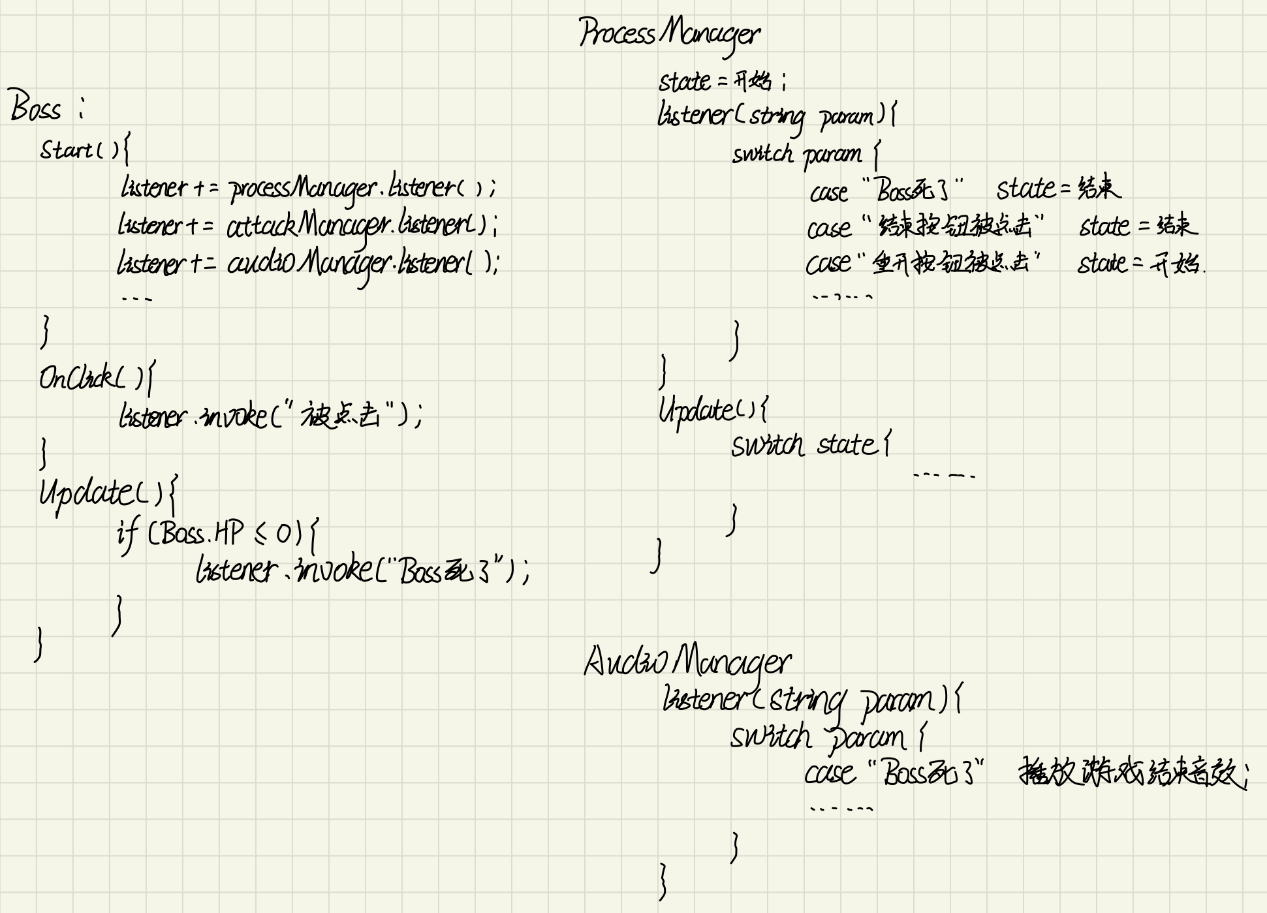


其次就是system，system在本次重构中不采用观察者模式，他更像是主函数中的一块逻辑的扩充，其中主函数自顶向下设计应当是这样的：



这里面有一个不是很高级的东西，就是耦合，一个主函数吧所有manager都用了个遍，如果要加一个点击时播放音效就要加一个audioManager（一个新的system）那么必然违反开闭原则，而且各个manager的执行顺序，执行逻辑也是这样耦合在了一块。但我觉得这种逻辑还是很清晰的，所以是故意把流程控制，点击控制“耦合”在了一块，感觉这样才能体现出游戏的逻辑。

那么system的“高级”作法是什么样呢。其实就是实现2022-8-12所强调几点不同的解耦方法，注意是没有主函数的：



注意不要有一个“消息分发中心”，因为那样的话相当于无效解耦，新增system的话还是需要更改消息分发中心的分发规则，各个manager关注自己的事情就行了。

总结：

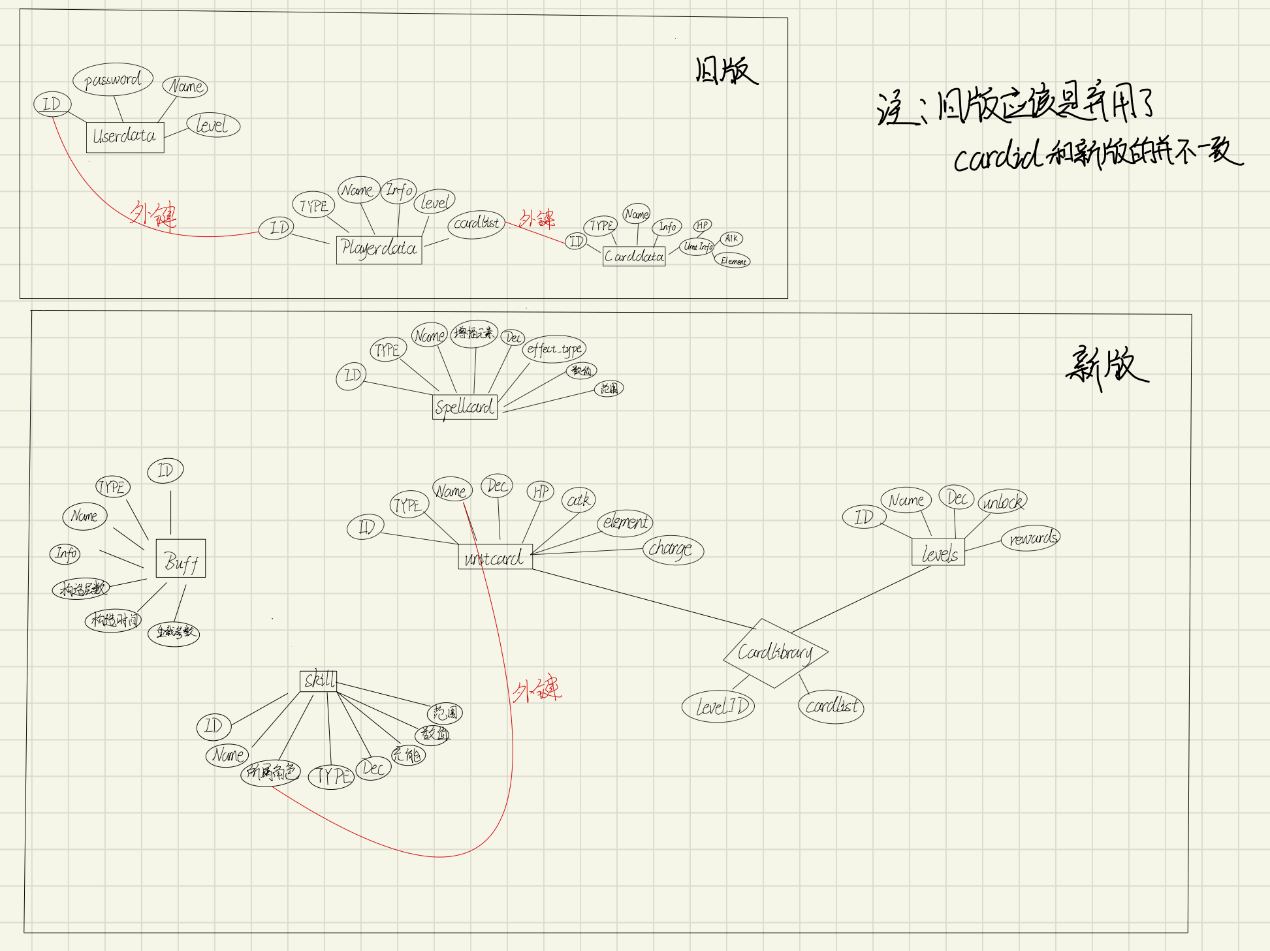
真的是每想通一件事就想歇逼，不再光想不干了，不更了，更新只会填补光想不干的罪恶感（嘿嘿今天最后一次，写个总结，辛苦我自己了）

本次也用偏向严谨的概述一下整个项目的框架吧，多加引用，毕竟引用的总是很权威捏：

本项目主要参考了ecs和spring框架：

首先数据部分借鉴了spring框架中的DAO和DTO两层，对于每一个卡牌数据对象会在DTO层实例化，同时也采用原型模式对于新的卡牌不再通过DAO层构造，大幅提升速度。

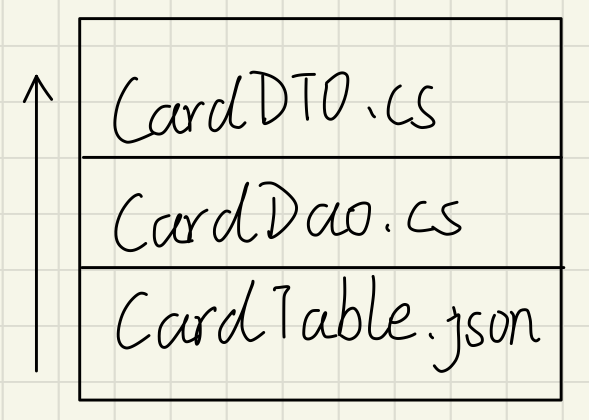
持久化类的具体实现，即类与表的关系基本参考orm框架的一对一模式，采用关系型数据库和类对象一对一的模式的优点在于添加表的时候只需要添加一个新类即可。首先应该重新理清各个表之间的关系：就目前几个表来说应当是这样的：



（艹，画出来才发现好乱，有大量数据冗余甚至冗余出来个新版旧版。此句删掉）

至于DAO层的实现，虽然用的并不是数据库，但原理还是一样的，只需要封装Luban插件对各个表的操作即可，对于用户名和密码将来联网的话就需要改成服务请求了（不会有人直接访问远程数据库吧，不会吧不会吧。此句删掉）。

其中数据的流通是这样的：



其中CardDTO即是一个可以流通的，封装卡牌基本信息的类。主要值得考虑的是持久化类的设计，比如要不要把skill（另外一个表）的信息也写入进去，这样持久化类与表的关系就不是一对一了。

Ecs框架：不想写啦，数据关系还是得重新设计一下，ecs框架分析见2022-8-20；