《神灵选拔赛》

支持 AI 与自定义角色的回合制对战游戏

设计说明书

作者：邓博誉

一、类设计

本项目共由角色（character）、状态（status）、场地状态（field_status）、技能(cmove)、场地(field)、战斗记录页(note_page)和战斗处理器(battle_handler)类所构成。这些类的关系如图 1 所示。在主函数中，通过选择调用三个不同的函数，构建相应的“战斗处理器”对象，以进入三种不同的游戏模式，即：

1，PVP（玩家与玩家进行对战）模式；

2，PVA（玩家与 AI 进行对战）模式；

3，AVA（AI 与 AI 自动对战并给出推荐的角色能力调整方案）模式。

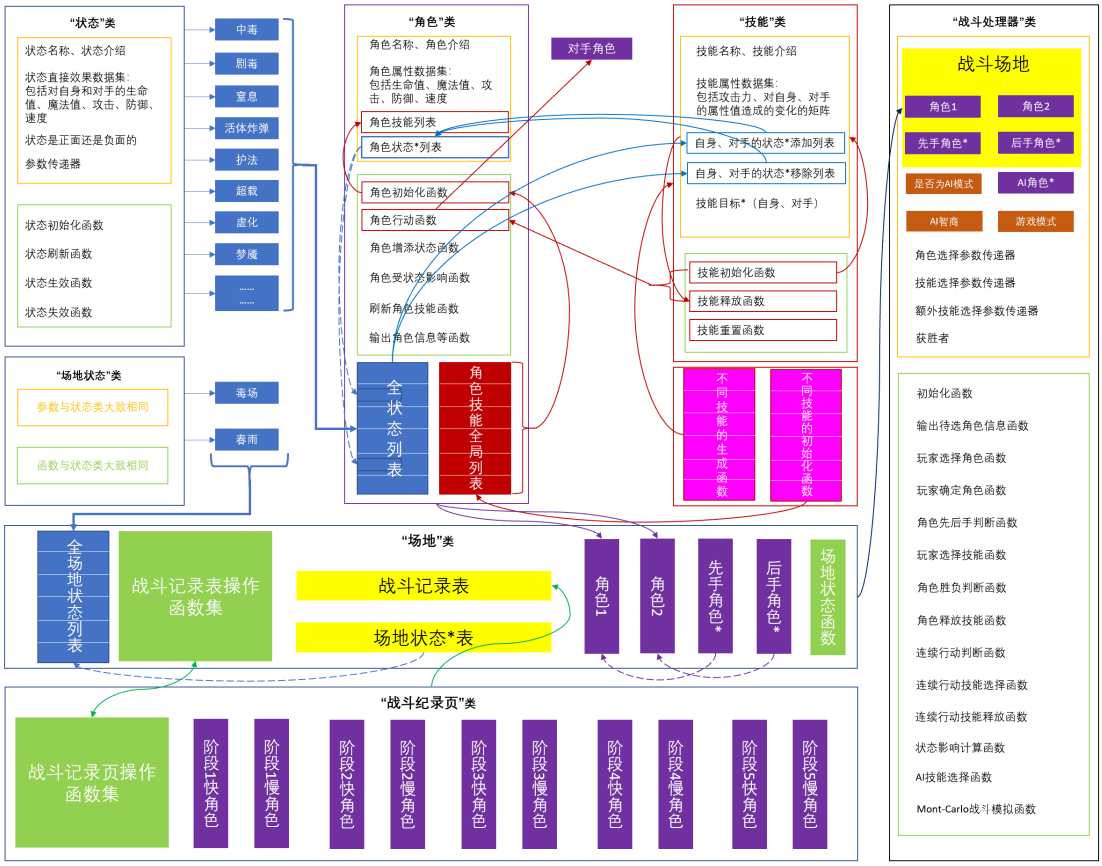

图 1 项目类图

角色类

角色类的数据成员及其含义和用途说明如下：

cName：角色名称。在本游戏中此名称被用作角色的唯一标识名，所以不允许存在名称相同的角色；

cIntroduction：角色介绍。在玩家选择角色时输出给玩家，为玩家提供参考；

maxHP：角色最大生命值，即角色的初始生命值；

maxMP：角色最大魔法值，即角色的初始魔法值；

ctr_atk：角色的攻击力，与技能攻击力和对手防御力共同计算得到伤害值；

ctr_def：角色的防御力，与对手的攻击力和技能的攻击力共同计算得到伤害值；

ctr_spd：角色的速度值，用来决定回合中先手与后手，以及是否获得额外回合以及获得额外回合的概率；

HP：角色当前生命值，该值降至 0 则判失败；

MP：角色当前魔法值，该值小于技能所需魔法值消耗时，技能便无法释放，转而恢复 50 魔法值；

statL：状态列表，储存的是指向角色自身数据成员中的状态对象的指针，代表目前角色身上生效的状态；

moveL：技能列表，储存角色可供使用的技能；

noteStatL：全状态列表，储存的是角色自身数据成员中所有状态对象的指针，用于便捷地搜索角色状态 ；

CharChoke、CharAquaBlast、……TickTock：这些是游戏中所有的状态的实体对象，作为数据成员存储于每一个角色中。

角色类函数成员及其含义和用途说明如下：

Initialize：初始化角色，用于玩家选择角色之后，将不同角色对应的各种属性和技能赋予角色。同时，该函数也可以在快速自定义角色中使用。

TakeTurn：角色行动，包括技能初始化和技能施放，用于玩家选定了角色要用的技能，角色需要施放该技能的时候。

add_status：向角色添加状态。用于角色施放或被施放技能，需要为自己的状态列表中添加响应状态时。

SufferStatus：角色身上的状态生效，即角色受到状态影响时使用此函数，用于双方施放技能完毕后。

print：输出角色基本信息，包括属性值和身上的状态名。用于向玩家输出战斗情况。

showIntroduction：输出角色介绍。在玩家选择角色时用于向玩家输出角色信息。

showMoveInfo：输出角色可用的技能信息，用于玩家选择技能时告知玩家可用的技能和编号。

SetMove：重设技能，用于角色施放技能之前，将所有技能的场地、己方和对方指针设于正确的状态。

状态类

由于不同的状态为状态的派生类，所以不同的状态的成员函数和成员数据均有所不同，无法统一说明。这里介绍一下通常状态对象作用于角色的主要参数和函数：

状态类的数据成员及其含义和用途说明如下：

sta_name：状态名称，作为同一种状态的标识符；

sta_info：状态信息，提供给玩家；

sta_logo：状态标记，此项目中未使用，后续项目可能会用到；

sta_dh：状态在生效时对状态拥有者造成的生命值变化；
sta_dm：状态在生效时对状态拥有者造成的魔法值变化；
sta_da：状态在生效时对状态拥有者造成的攻击力变化；
sta_dd：状态在生效时对状态拥有者造成的防御力变化；
sta_ds：状态在生效时对状态拥有者造成的速度变化；
sta_Odh：状态在生效时对状态拥有者的对手造成的生命值变化；
sta_Odm：状态在生效时对状态拥有者的对手造成的魔法值变化；
sta_Oda：状态在生效时对状态拥有者的对手造成的攻击力变化；
sta_Odd：状态在生效时对状态拥有者的对手造成的防御力变化；
sta_Ods：状态在生效时对状态拥有者的对手造成的速度变化；
iniT：状态持续时间，由技能给予或在初始化函数中设定；
nT：状态剩余时间，剩余时间<0 时状态即执行消失函数并消失，初始化时 nT=iniT，状态每造成一次影响，nT 则减少 1；
sta_pos：布尔量，是否为正面状态，由初始化函数设定。此项目中未使用，但后续开发或许会用到；
sta_neg：布尔量，是否为负面状态，由初始化函数设定。用于标记负面状态，在 Irrawa 的 Tailwind 技能中，只驱散负面状态，用此参数进行识别 。
ParameterDeliver：参数传递器，被用于收集和传递状态每回合生效时的信息给消失函数。例如 NayadBreeze 状态中，每回合都会加速，状态消失时应减去所有加上的速度，即利用此参数进行累加。此参数不限于只有这一个。

状态类函数成员及其含义和用途说明如下：
SetupStatus：状态设立，作为创造并初始化状态对象使用。
RefStatus：在状态对角色造成效果之前，先运行此函数将状态参数计算至正确值。例如在“剧毒”状态中，每回合其造成的伤害都会增加，那么就在此刷新函数中，依据此状态的内置计数器，每次状态生效前计算出其应有的伤害。
StatusTakeEffect：状态在此函数中对双方角色和场地造成影响。
StatusLoss：状态的消失函数。此函数用于在状态消失时将状态对角色造成的“暂时性”影响复原。

场地状态类

场地状态的设计与角色状态的设计类似，不再赘述。

技能类

技能类的数据成员及其含义和用途说明如下：
mName：技能名称；
mInfo：技能信息；
realName：技能的真实名称，通常情况下无用。但在特殊情况下，如技能被临时更改（比如 Rosie 的 Shadow Mirror 技能，在释放前其名称会被改变以

给玩家反馈更加完善的信息），但又要保持技能的标识不变，则配合下面的 nameChanged 布尔量进行使用；

nameChanged：见 realName 参数介绍；

slf_dh：技能对自身的生命值的变化；

slf_dm：技能对自身的魔法值的变化；

slf_da：技能对自身的攻击力的变化；

slf_dd：技能对自身的防御力的变化；

slf_ds：技能对自身的速度的变化；

opo _dh：技能对对手的生命值的变化；

opo_dm：技能对对手的魔法值的变化；

opo _da：技能对对手的攻击力的变化；

opo _dd：技能对对手的防御力的变化；

opo _ds：技能对对手的速度的变化；

mv_atk：技能的攻击力，与施放者的攻击力和被攻击者的防御力一同计算伤害值；

slf_rmStat：技能将移除的自身状态的指针列表；

opo_rmStat：技能将移除的对手状态的指针列表；

slf_adStat：技能将增添的自身状态的指针列表；

opo_adStat：技能将增添的对手状态的指针列表；

self_target：技能是否以自身为目标，只要不是目标为对手均为 true。用于改变释放技能时输出的字符串信息，以自身为目标的的技能不需要告诉玩家该技能的施放目标。

技能类函数成员及其含义和用途说明如下：

SpellMove：技能在施放前的准备，根据环境将各种参数配置好的函数。

LaunchMove：技能施放，在此函数中，技能与角色与场地进行互动，包括使角色的属性值发生变化和对角色和场地的状态列表进行增减；

ResetMove：此函数用来计算技能的伤害值，伤害值会默认被加到 opo_dh 参数上。函数式为：

$$Dh = -\frac{A_m \times A_c}{D_c}$$

其中A_m为技能攻击力，A_c为施放者攻击力，D_c为受攻者防御力。

场地类

场地类的数据成员及其含义和用途说明如下：

FStatusL：场地的状态指针列表，与角色状态列表使用方式类似，同样场地类中也有场地效果的实体对象；

battleRecord：战斗记录，用来存储战斗历史各个回合的记录页。一些技能可能会需要这些信息（如 Rosie 的 Static Overload 反弹伤害的效果和 Asibi 的 Objective Illusion 技能，均需要获得战斗的历史信息）。

场地类的数据成员及其含义和用途说明如下：

build_begin_state：建立初始场地状态，初始化各个角色，建立战斗记录并建立第一页记录，调用于玩家选择完角色后；

FieldSufferStatus：场地状态对玩家造成影响的函数，用于一回合的最后，双方下一回合选择技能之前；

add_status：向场地状态指针列表中添加场地状态，与角色对应函数功能相似；

NewPage：在战斗记录中建立新的一页；

GetPage：从战斗记录中读取指定的页，如果超过页尾则返回最后一页，超过页头则返回第一页；

GetCurrentPage：从战斗记录中读取当前页；

WriteRecord：将输入的角色指针所对应的角色记录到当前页的指定“行”。

战斗记录页类

战斗记录页类有 C00、C01……C09 共 10 个角色数据成员，分别对应技能施放前、先手释放技能后、后手释放技能后、先手遭受状态后，后手遭受状态后的先后手角色信息。相应地，每一个角色数据成员均有一独立的函数用于写入状态。同时，该类中还搭载了拷贝角色的函数，用于更加方便地进行以上处理。

战斗处理器类

战斗处理器类的数据成员及其含义和用途说明如下：

BattleField：场地数据成员，所谓战斗的场地；

p1Character、p2Character：双方角色，当游戏模式为 PVA 时，p2Character 为 AI 控制角色；

*fasterCharacter、*slowerCharacter：较快角色、较慢角色的指针，用于操作先后手；

*AICharacter：AI 控制角色的指针，用于 AI 操作角色；

characterList：角色列表，用户从该列表中选择角色，一个角色被选后会从列表里删除；

AIMode：战斗是否有 AI 参加；

AIIQ：当 AIMode=true 时，此参数被用作设置 AI 的强度，默认值为 20；

GameMode：1：PVP；2：PVA；3：AVA;

p1MoveNum、p2MoveNum：双方角色选择的技能标号，用于参数传递；

fasterMoveNum、slowerMoveNum、speedMoveNum：较快、较慢角色、额外选择的技能标号，用于参数传递；

Winner：0：暂未分出胜负；1：p1 获胜；2：p2 获胜；

战斗处理器类的函数成员及其含义和用途说明如下：

Initialize：初始化各数据成员，包括创建角色名单和所有技能；

showCharacterList：根据预定的格式打印角色名单；

ChooseCharacter：引导玩家在角色列表中选择角色，输入玩家编号和角色列表，将选择的角色自动赋予相应的数据成员；

DecideCharacter：引导双方选择各自角色的函数，p1 先选，p2 后选；

JudgeSpeed：判断角色速度，将 p1Character 和 p2Character 分别赋予 *fasterCharacter，*slowerCharacter 指针，若速度相同则随机分配；

GeneralChooseMove：引导双方选择技能的函数；

IfWin：判断战斗是否分出胜负的函数；

CastMove：双方角色施放技能的函数，返回值为获胜信息：0：暂未分出胜负；1：p1 获胜；2：p2 获胜；

IfSpeed：判断先手角色是否可以获得额外行动机会；

SpeedChooseMove：引导玩家在额外行动机会中选择技能；

SpeedCastMove：额外行动技能施放，返回值为获胜信息：0：暂未分出胜负；1：p1 获胜；2：p2 获胜；

DoStatus：状态分别生效并产生影响，依次为先手角色状态、后手角色状态、场地状态，返回值为获胜信息：0：暂未分出胜负；1：p1 获胜；2：p2 获胜；

AIChooseMove：AI 选择技能函数，输入 AI 能力和 AI 角色标号，输出技能编号；

同时，该类的源程序中搭载了 MontCarlo 战斗模拟函数。

二、主要技术难点和实现方案

模块化程序结构搭建

本软件在设计之初即要求被设计成可拓展性强，支持方便地增添角色、技能和状态。只需要额外加入代码，而不需要改变原有代码即可加入更多的游戏内容。所以模块化的设计非常重要。虽然回合制对战游戏的战斗过程看似简单，但其实有很多错综复杂的逻辑，难以实现模块化，这也是目前市面上可见到的回合制对战游戏均未有类似功能的原因之一。

本游戏将角色、技能、状态、场地四大回合制战斗游戏元素分别以不同的类实现，并尽量将其交互的接口简化。当对其中的一项元素的内容进行扩展时，只需要额外编写扩展项的对象生成函数或派生类，然后将新的元素加入交互的列表中即可，不需要对其它元素中的信息进行更改。本项目的后半部分开发过程，即在尝试、使用这种特点并获得了很大的效率提高。

本游戏的整体框架在实现 Asibi 角色之前便已经完成，Asibi 角色及与其相关的技能和状态时，均在游戏可以正常使用之后才被添加。本人在添加 Asibi 角色时，并没有更改任何其它角色和整体框架的代码，只是增加了与 Asibi 相关的状态派生类、技能生成函数、角色对象，随后将这些元素加入到其应该有的列表中（如将 Asibi 角色加入 CharacterList，将 Nightmare 状态加入 Character 类定义中的角色状态列表中等等）。图 2 和图 3 在一定程度上验证了开发 Asibi 角色时的过程以及带来的收益，从图中的代码变更记录可看出，

添加 Asibi 角色的各种元素时，未对其它无关文件进行更改。证明本软件在很大程度上实现了自定义角色和模块化。

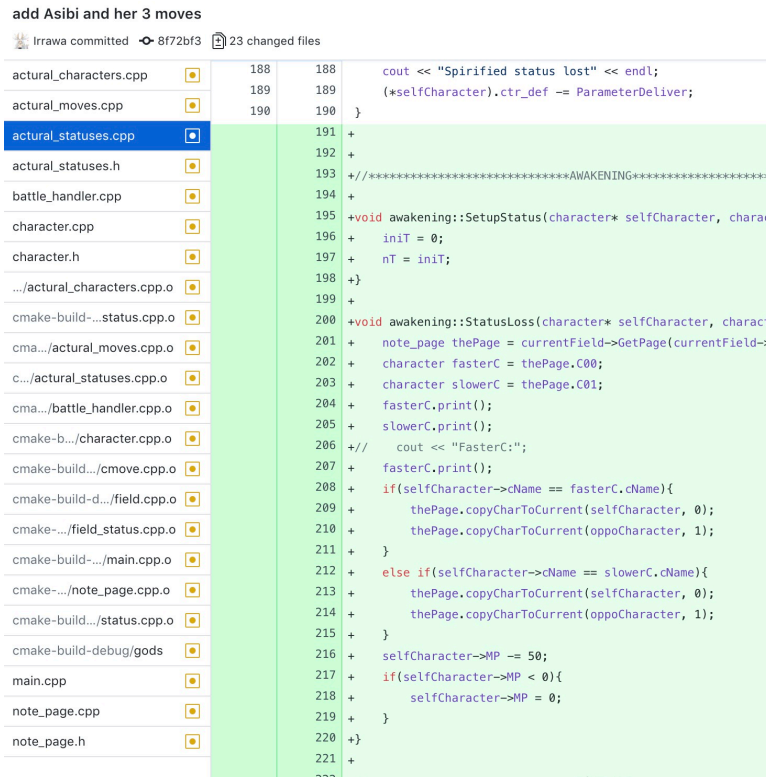

图 2 创建 Asibi 角色时的代码变更记录截图

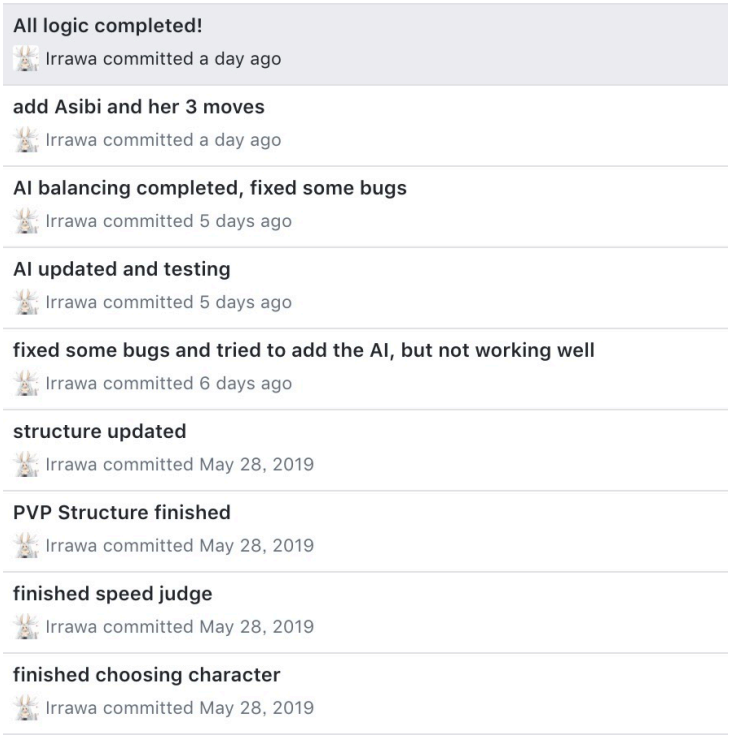

图 3 Asibi 的开发在主体框架完成之后才进行

技能与状态的多样性

本回合制游戏的状态与技能效果非常多样，虽然只有 20 个技能，但其广度已经达到甚至超过大部分目前主流回合制战斗游戏的广度。例如：返回数回合之前的状态、回合数计数、时间跳跃等技能，均难得在回合制游戏中见到。技能复制、效果叠加时重制持续时间、效果与人物属性的关联等，也是很少能在回合制游戏中见到的。单单实现这些功能或许并不困难，但将这些多样性与扩展性结合起来，则是本程序最困难，最有亮点的地方，将在下章详细阐述。

多样性与可拓展性的冲突

保持可拓展性固然是很好的，但可拓展性在很多情况下会导致多样性降低，而对于游戏来说，多样性是非常重要的。如果一个游戏中的元素多样性过低，势必会影响到其可玩性。加入本游戏的角色、状态、技能和场地的接口参数均为固定值，那么玩家很容易就可以发现每次行动的最优解，游戏的不确定性所带来的刺激感也将消失。

在设计本游戏时，就已经强调了游戏应支持多样化的拓展，必须保证技能和状态的多样性。为了达到这一目的，本游戏着重在三个方面进行突破：

1，尽量让所有的接口均能获取游戏全局信息；
2，使用派生类而不是对象来制作不同的状态；
3，为每个技能创建独立的初始化函数和刷新函数；

在接口设计上，本游戏大量使用了如下格式的接口：

（character* C1, character* C2, field* F）

由于所有的状态和技能均存储在 character 中，所有的场地状态均存储在 field 中，战斗记录中的数据也在 field 中，所以此格式的接口是可以获得整个战斗的所有信息的。这样一来，无论是角色、技能还是状态，均可以调用战斗中的所有信息来建立函数，调整自己的各项参数，从而形成多种多样得、有个性的角色、有特点的技能和状态，让整个游戏的拓展广度很高。

状态系统是本游戏多样性的重要组成部分。游戏使用了派生类来制作不同的状态，即给予了各个状态拥有自己独特的成员函数、成员数据的机会。如此一来，每个状态在其存在的每一个回合，与角色和场地的交互都是可以自定义的。配合上文所描述的接口，这些状态可以在每回合结束时为场面带来丰富的变化可能性。例如，我们可以很轻易地实现下述的看似很复杂的状态：

“若角色在接下来 3 回合之内损失了超过 300 点生命值，则回复这些生命值的 50%，并为对手施加剧毒状态。”

为了实现此状态，只需要将状态持续回合设定为 3。在 StatusLoss 函数中从 field 的 battleRecord 中读取 3 回合前己方的生命值，获取 3 回合的生命值差 ΔH 判断是否满足条件，若满足，则执行 opo_character -> add_status (&opoCharacter -> Toxic)，和 slf_character->HP += ΔH * 0.5 即可。

考虑到开发的复杂度，技能系统没有像状态系统那样加入派生类（因为在类似回合制战斗游戏中，技能的种类往往远远大于状态的种类，为每一个技能

设置子类不经济，也没有意义）。为了使技能的多样性得到满足，游戏中采用了每个技能所独有生成函数和初始化函数的方式。游戏初始化时分别调用生成函数生成各个技能并赋予各个角色，在角色施放技能之前，通过调用初始化函数来根据战斗数据现场“制作”一个即将被施放的函数，而这个技能初始化函数也拥有上述的“万能接口”。这样一来，就可以在游戏设计时，通过自定义这两个函数，来打造变化多端的技能。本游戏中的 ToxicBlast 和 ShadowMirror 等复杂有趣的技能的实现，便证明了此方法的有效性。

状态系统的复杂性

本项目的状态系统较为复杂，虽然各个状态均为 status 类的派生类，但由于 slicing 的原因，很多情况下难以批量处理复数的 status（例如，无法建立一个可以储存不同的 status 的容器，以至于难以在一个技能中加入任意多的添加状态）。为了解决这个问题，我将原本的设计进行了改进，即：

1，在每一个角色中内置所有的状态，并在角色创建时初始化；
2，角色的状态列表中不存储状态，而是状态的指针，各个指针指向角色自己的响应状态；
3，角色增删状态时，函数传递响应状态的指针即可；

使用以上设计方法，便解决了状态多样性带来的麻烦，同时还从本质上避免了出现同样的状态在一个角色身上复数出现的 bug 的可能性。不过在复制角色时，需要重新定义指针，此过程在战斗记录页类里使用了一个函数来完成。同样，在场地状态中也应用了类似的方法。

部分需要历史数据的技能处理

原设计中的一些技能（如 Lust Storm 和 Objective Illusion）需要获得之前战斗的信息。因此设计了战斗记录页类，并在场地中加入了 BattleRecord，用来存储不同回合的战斗记录。形象地比喻来说：BattleRecord 就像是一本书，而战斗记录页则是书中的每一页，而每一个角色在回合不同阶段的状态，则是每一页中的每一行。这种数据结构，可以很方便地让角色、技能或状态读取战斗历史中不同阶段吗不同角色的状态，以计算相应的参数。

内存管理与优化

本项目的 AI 算法中，若不加以针对性地设计，会制造大量内存碎片，导致程序长时间运行后速度越来越慢，甚至被系统 kill 掉。在发现此问题后，我重新设计了 AI 算法中树搜索的内存管理方法。不再在每一次循环中创建新的对象，而是在循环开始前先保存当前战场的副本，随后就在当前战场上进行树搜索模拟，在搜索结束后再用之前的副本重新覆盖经模拟后状态未知的当前战场。这项改进使程序的内存占用从数十 GB 减少到 300MB 左右，且峰值也不会超

过 800MB，虽然理论上应该还可以再优化内存，但至少已经可以正常运行了，占用内存不会无止境升高，程序不会越来越慢，也不会被系统 kill 掉。

AI 设计

本项目需要有 AI 设计。在项目中，我引入了朴素 MontCarlo 算法，即在 AI 选择技能时，使用每一个技能 n 次，并且在此技能使用后随机选择双方的技能直至分出胜负，若该技能获胜，计 1 分，否则不计分。最终选择分数最高的技能，若分数相同，则随机选择技能。此算法并不是最优的算法，但在此项目中已足够让 AI 拥有足以与玩家相较量的实力，所以被采用，并利用此算法实现了角色自动平衡功能，是本游戏的另一大亮点——显然，玩家自定义的角色通常会破坏游戏的平衡性，而此功能，则在自动游戏调平衡的方面做了大胆的尝试。

模拟界面

一款没有界面的游戏对玩家是很不友好的，会导致玩家读取关键信息困难，从而降低可玩性。但本课程并没有教学如何制作可视化程序，本人暂时也不具备进行可视化编程的实力。但这并不代表此游戏不能有基础的界面。经过设计，本游戏使用字符与适当的排版，制作了简易却清晰明了的界面，玩家可以快速地锁定战斗中的关键信息。其中角色的属性和状态信息直接打印在模拟场地中，而场地状态则使用 logo 参数打印在场地上，以模拟场地效果。界面效果如图 4 所示。当游戏处于 PVA 模式时，玩家控制的角色必定在下方，符合主流回合制游戏的设计原则。

```
==========================================
O。O。O。O。O。O。O。O。O。O。O。O。O。O。O。O。
[Irrawa]
HP:1200   MP:650   ATK:125   DEF:100   SPD:80
Status: {POISONED}


[Mew]
HP:963   MP:975   ATK:100   DEF:100   SPD:100
Status: {AQUA BLAST} {POISONED}
O。O。O。O。O。O。O。O。O。O。O。O。O。O。O。O。
==========================================
```

图 4 战场信息显示界面示例

计算数据长期存储与断档读取功能

当前流行的游戏通常都有保存游戏数据到本地磁盘的功能。本游戏虽然未提供回合制战斗的数据永久存储，但在 AVA 模式中提供了，对角色平衡状况的存储和读取，以便于游戏调试人员实时保存计算结果，并且可以方便地继续上次未进行完毕的计算。角色最后一次被调整之前的结果将被存储在程序根目录下的“suggested.txt”文件中，而当游戏调试者再次进入此界面时，若根目录下的“suggested.txt”文件存在，则默认从该文件中读取数据并将其赋予 p1（即要调平衡的角色）而不是使用 p1 的默认属性。此功能可让游戏调试人员在不修改程序内置角色默认属性的情况下，方便地获知角色如果被调整到某个属性值的情况下的强弱。这种思想对在线游戏非常有用。