match_core.py

代码分析报告

目录

0. 总览与简介

match_core.py 是纸带圈地 AI 比赛的核心游戏逻辑代码，负责按照指定的比赛场地参数，接收双方 AI 文件进行对抗，并将比赛结果记录并输出。

模块主要对外开放的功能接口为 match 函数与 match_with_log 函数：match 函数输入双方玩家 AI 代码与比赛参数后会自动发起一场对抗，向双方函数传递参数，接收其返回值并更新比赛场地，并在结束后输出比赛记录对象；match_with_log 函数则在返回对象的同时自动将其输出为

导入的规范编写的玩家 AI 代码应包括必要的 play 函数及可选的 load 与 summary 函数；导入的函数在执行时会自动挂载异常处理与超时处理系统，对于行为异常的代码将以比赛结果判负的形式直接显示出来。

0.1. 代码实现总览

核心代码的逻辑主要可以分为 3 部分：框架逻辑、游戏执行逻辑以及附加的计时模块。在调用执行接口进行比赛时，框架逻辑负责运行数据的传递与整合，游戏执行逻辑负责具体的游戏规则，计时模块则具体实现了评估玩家 AI 函数的功能。

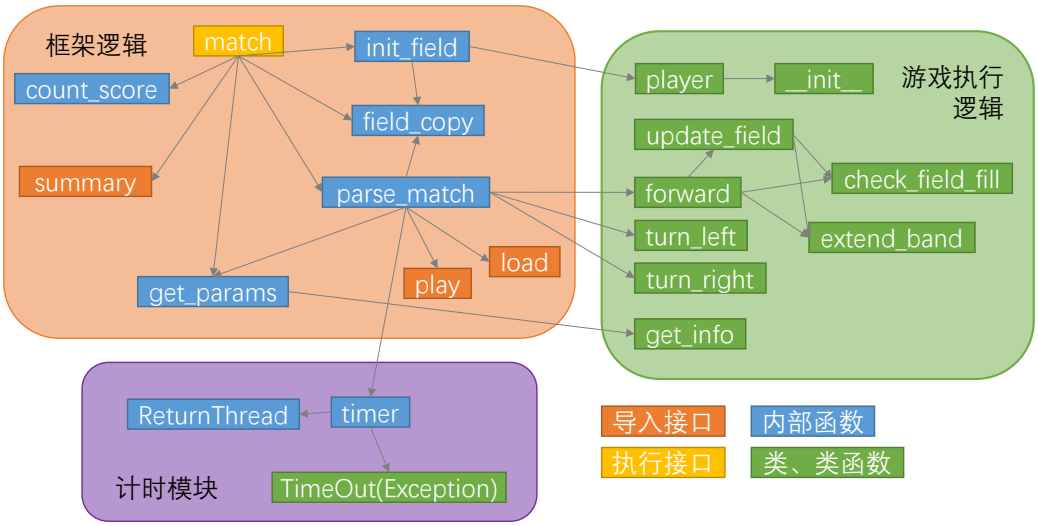

图 1 代码结构图

3 个功能块和其所属的主要函数、类之间的调用关系如上图。下面将逐个介绍各部分所属函数的内容与实现细节。

0.2. 全局变量

核心代码使用数个全局变量存储游戏状态。

比赛时间参数

变量名	默认值	内容
MAX_TURNS	None	最大回合数（双方各操作一次为一回合）
MAX_TIME	None	总思考时间（秒）
TURNS	None	记录每个玩家剩余回合数
TIMES	None	记录每个玩家剩余思考时间

比赛空间参数

变量名	默认值	内容
WIDTH, HEIGHT	None, None	最大回合数（双方各操作一次为一回合）
BANDS	None	纸带判定区 初始化方式为[[None] * HEIGHT for i in range(WIDTH)]
FIELDS	None	已生成区域判定区，初始化方式同上
PLAYERS	[None] * 2	双方玩家对象列表

AI 函数存储空间与跨比赛记忆

变量名	默认值	内容
STAT	None	双方游戏信息
STORAGE	[{}, {}]	双方函数存储，可自由使用

其它

变量名	默认值	内容
LOG_PUBLIC	None	全局比赛记录列表
NULL	lambda storage: None	
FRAME_FUNC	NULL	逐帧处理函数接口，替换以使用

另：函数属性变量 match.DEBUG_TRACEBACK 用于报错输出存储变量，在一方玩家因函数报错而判负时存储错误信息的 traceback 信息。

0.3. 玩家 AI 代码接口

玩家编写的 AI 代码应包含 play 函数作为比赛运行中的双方 AI 进行对抗；此外，还支持可选的 load 函数与 summary 函数，提供额外的功能。

load 函数与 play 函数报错时直接判负，summary 函数报错时将被跳过。

load(storage)（可选）

在比赛开始前运行一次的函数，用于在存储空间 storage 中初始化运行环境等。

play(stat, storage)

行动策略函数，每回合先后手玩家交替运行，接收当前的游戏状态 stat 与存储空间 storage；比赛中通过读取该函数返回值决定双方玩家行动方式。

summary(match_result, storage) （可选）

在一轮比赛结束后运行一次的函数，接收比赛结果 match_result 与存储空间 storage，可用于总结比赛等；storage 中关键字为 memory 的字典将保留。

1. 框架逻辑

框架逻辑负责初始化全局比赛环境，交替向计时模块传递双方函数，以及向双方 AI 代码及数个外界接口的 IO 功能；其中 match 与 match_with_log 函数为整个模块的入口。

1.1. 框架外部接口

match(name1, plr1, name2, plr2, k, h, max_turn, max_time)

整合的比赛函数，输入双方的名字与代码模块，并设定好比赛场地参数之后，match 函数将自动初始化并执行一场比赛。是整个模块调用结构中真正的根节点。

参数：

name1, name2：先后手玩家分别的名称

plr1, plr2：先后手玩家 AI 代码分别导入成的模块，包含 play 等函数

k, h：控制比赛场地大小，生成的比赛场地网格长宽分别为 2k, h

max_turn：比赛最大回合数，两方分别执行该数量回合后将结束比赛结算得分

max_time：比赛最大思考时间（秒），任一方 AI 函数运行总时长超过该数则判负

实现方式：

match 函数首先会根据比赛参数初始化比赛环境，即初始化对应的全局变量，这个过程中会调用 init_field 函数初始化比赛场地与玩家，同时初始化双方使用的存储空间（其中 memory 关键字传入全局保存的字典引用）；随后会传入双方 AI 模块信息调用 parse_match 函数进行比赛，并返回记录比赛结果的元组，并在双方未直接分胜负的情形下调用 count_score 函数统计得分以判胜负；最后调用双方的 summary 函数（如果存在），并将本次比赛结果记录打包输出为字典。

1.2. 框架内部函数

init_field(k, h, max_turn, max_time)

初始化比赛空间参数全局变量；将玩家对象 PLAYERS 中两个对象分别置于比赛场地左右部分正中附近的随机位置。

field_copy()

创建 FIELDS 与 BANDS 的元组拷贝，每回合执行一次

get_params(fields, bands, curr_plr=None)

从全局变量获取当前比赛状态，打包为字典传出。字典内容包含剩余回合数、剩余时间，并从参数输入获取纸片场地列表与纸带场地列表的深拷贝。

可选参数为玩家编号（0 或 1）时会额外返回标记每个玩家“自己”与“对手”的玩家信息引用，返回值将用于玩家 AI 读取与记录比赛状态；否则会额外返回记录双方玩家纸带路径的数组，返回值将保存于比赛记录对象中，额外信息可用于可视化等。

parse_match(funcs)

在已初始化的场地中进行一次比赛，并返回标记比赛结果的元组。输入参数为长度为 2 的列表，其中两个元素分别为先后手玩家代码模块。

实现方式：

首先，系统会执行双方玩家模块的 load 函数，向其传递初始的 storage，若没有声明 load 函数则跳过该步，在该步报错或超时则直接判负。调用方式为作为参数传递给计时模块。

开始比赛后，比赛逻辑会循环指定的回合数次，轮流向计时模块传递先后手玩家的 play 函数及参数，并对其计时，在超时或报错时直接判负；随后，根据函数返回的操作符，决定玩家 AI 代码控制的 player 对象是否左/右转，并调用 forward 函数使其前进一步。若该调用返回值为比赛结果（元组，不为 None）则直接返回该结果，否则向双方存储空间和公用的比赛记录中分别添加当前比赛场地状态，并继续循环。

循环结束后，返回“回合数耗尽”的比赛结果，待由外层 match 函数统计得分。

count_score()

统计得分函数，遍历 FIELDS 二维列表并统计双方玩家 id 出现次数，输出统计结果元组，分别为先手玩家得分、后手玩家得分。

2. 游戏执行逻辑 – player 类

player 类集成了纸带圈地游戏的逻辑，通过数个代表玩家操作的函数连接了玩家 AI 的输出与比赛场地全局变量。

2.1. 类变量

变量名	功能
directions	静态类变量=[(1, 0), (0, 1), (-1, 0), (0, -1)]；指示方向对应坐标变换
id	玩家标记，先手玩家为 1，后手玩家为 2
x, y	玩家在场地内坐标
direction	玩家面对方向
band_direction	纸带轨迹列表，记录纸带生成以来每格对应的玩家方向
field_border	列表，记录玩家到达过的区域的坐标范围（包络矩形）

2.2. 算法原理

游戏规则

根据原始游戏 paper.io 中各种边界情况的考察，结合一些额外定义的规范，最终的游戏规则可以总结为如下几点

a. 游戏过程中每回合双方轮流向某个方向前进 1 格；出界玩家直接判负
b. 玩家离开领地时会在路径留下纸带；重新返回领地时将所有纸带转换为领地并填充封闭的非领地区域
c. 玩家走入纸带区域时，纸带所属者直接判负
d. 玩家相撞时，若在其中一玩家领地内则该玩家获胜；否则，若双方方向垂直则主动方获胜，若方向相反则进入结算
e. 比赛过程中若发生上述情况则终止比赛，若一方代码报错或总时长耗尽则另一方获胜。
f. 运行指定回合后进入结算，统计双方领地，面积大者获胜

“填色”原理

游戏规则中由纸带到领地的转换涉及到两个步骤：①将纸带从游戏场地移除，并将其占地全部转换为领地；②将领地形状内“空心”的部分填充为指定“颜色”，无论其原始内容（空白或对方领地）。

在代码实现中，领地与纸带的占地分别由名为 FIELDS 与 BANDS 的二维列表表示，算法涉及到对二维列表进行遍历的 $O(n^2)$复杂度操作。为简化算法，增加性能，有必要对其进行优化。

a. 纸带-领地转换

每个玩家对象维护一个 band_direction 列表，包含生成纸带每一步时玩家的前进方向(0-4)。当玩家在自己领地内，无纸带时，该数组对应为空；每当玩家前进一格走到领地外以后，铺设纸带的过程不仅会将 BANDS 列表中玩家当前坐标元素设置成玩家对应标记，还会向 band_direction 列表中加入当前玩家方向。

在玩家重新回到自己领地，将所有纸带占地转换为领地时，首先设置一个坐标 ptr 指向玩家这回合前进方向的上一步（这一步未添加新纸带），同时不断弹出 band_direction 列表末尾，并将 ptr 向每个元素对应方向的反方向不断移动 1 格；ptr 经过路径上的所有坐标都将移除 BANDS 数组内对应内容，并在 FIELDS 数组添加该玩家标记。

该算法将 $O(n^2)$的查询纸带占地操作简化为了 O(n)的回溯操作，且在记录比赛信息时可以额外返回纸带的方向信息，便于可视化时对纸带方向进行显示。

b. 领地填色

填色过程使用了经典的 floodfill 算法，并对其进行了范围优化与 python 化改进。floodfill 算法被使用于对一块封闭区域进行上色，其本质为按次序遍历并标记图的节点，并在到达边界时自动停止。在该“填色”情景下，需要填色的对象为非自己领地的格点，边界则为两种：已有的自己领地格点与场地边框。所有与场地边框不相邻的区域均视为“内部”区域，需要进行上色。

注意到比赛场地可以很大，而双方玩家互动涉及到的区域往往只是很小的区域，对整个比赛场地执行算法会有大量不必要的时间开销。为减少每次填色需要遍历的格点个数，每个玩家都维护了一个 field_border 列表，其内容为[x_min, x_max, y_min, y_max]，分别记录了玩家到达过的横纵坐标的最大值与最小值，初始为玩家出生点外围 3*3 的正方形领地；每当玩家坐标更新时，可能的影响范围扩大，该列表的值也会相应更新。

为忽略边界条件检查、减少额外的变量维护，填色操作的具体实现利用了 python 的数据结构进行简化，其大体流程如下：

> 建立名为 targets 的 set，其初值为遍历 field_border 范围内所有 FIELDS 中该位置不为自己领地的坐标，包含了**所有需要检查的坐标值**
> 在 **targets 不为空**时重复执行：
>> 建立名为 iter 的列表，包含 targets 中弹出的一个坐标，用于**迭代一次 floodfill**
>> 建立名为 fill_pool 的空列表，记录**该次填色需要上色的所有坐标**
>> 逻辑值 in_bound 设为 True，用于标记**这次上色是否为内部区域**
>> 在 **iter 不为空**时重复执行：
>>> iter 中弹出一个坐标记为 curr
>>> 在 curr 周围 4 个方向相邻的坐标内，如果 targets 包含该坐标，则将该坐标从 targets 中移除并添加至 iter
>>> 如果 in_bound 为 True：
>>>> 如果 curr 落在 field_border 上，将 in_bound 设为 False；否则将 fill_pool 中加入 curr
>> 如果 in_bound 为真，则将 fill_pool 中所有坐标填充为当前玩家 id

其中涉及到的数据结构操作（除迭代式建立初始 set 外），包含 set 移除元素，list 在末

尾添加、删除元素等，均为常数量级的操作；总体访问的单元数不大于 5 倍 field_border 面积，复杂度为 $O(n^2)$。

2.3. 类函数

注：player 类内无静态函数，且除初始化函数外均不接收外部参数；以下内容省略 self

__init__(id, x, y, expand=1, init_direction=None)

根据提供的参数在比赛场地内放置玩家对象，并在 FIELDS 列表中覆盖对应位置的领地。

参数：

id：玩家编号（先手玩家为 1，后手玩家为 2）

x, y：初始坐标

expand：初始领地从玩家位置扩展距离，默认为 1，对应中心与玩家对齐的 3*3 领地

init_direction：初始方向，默认为 None 时初始方向随机

turn_left 与 turn_right

转向函数接口。

左转时方向设为(当前方向+3) % 4，右转时方向设为(当前方向+1) % 4。

forward

前进一格，并执行游戏主体逻辑。若触发游戏结束条件，该函数会返回一个元组；否则返回值为 None；比赛函数调用该函数时将捕捉返回值并决定是否结束比赛。

内容：

玩家首先调用 directions[direction]向当前方向前进一格，若前进后坐标超出场地边缘则返回对手获胜结果；随后以当前坐标更新 field_border 记录的边界。

第二步是判断玩家间相撞的情况，在全局以 PLAYERS[2 - id]的方式引用对手玩家。在对手玩家和自己的坐标重合时调用一系列判断返回相应游戏结果，其中自己领地外的情况需先调用 extend_band 更新纸带，保持纸带终点始终为自己的坐标。

随后，检查 BANDS 列表中当前位置是否为 None，若包含纸带，首先调用 update_field 函数更新场地，随后返回该纸带主人负的结果。

其余情况下调用 update_field 函数更新场地。

update_field

更新玩家在场地引起的效应。首先判断当前位置是否为自己领地：若是自己领地，判断 band_direction 列表是否为空，并在其不为空的时候调用 check_field_fill 将纸带转换为领地，并进行填色；否则调用 extend_band 函数延长纸带。

extend_band

延长纸带函数，将当前方向加入 band_direction 列表末尾，并将 BANDS 列表内当前坐标设置为自己的 id

check_field_fill

由领地外返回领地时执行的填色操作，分为“纸带转换为领地”与“填色”两步，详见“算法原理”部分说明。

get_info

返回玩家当前信息的字典，由 get_params 函数调用

3. 计时模块

对于需要计时的玩家 AI 函数，框架逻辑中将函数与函数需要读取的参数统一传递给计时模块进行运行并计时。若运行中出现错误，计时模块能将异常捕获并输出给

3.1. 原理

利用 threading.Thread 进行计时。Thread.join 函数能在线程运行时阻塞主线程，在输入参数时会设定最大阻塞时间：如果该时间内线程终止就会停止阻塞，否则最长只能阻塞该时间便会释放。

计时模块使用了线程进行计时，并结合函数参数进行输入函数返回值（或捕获异常）的传入传出。调用计时模块执行函数的代码可获得返回值、捕获超时异常、捕获普通异常共三种结果，并对结果进行后续处理。

3.2. 内容

TimeOut 类(继承自 Exception)

自定义的超时异常，用于与函数运行报错区分。

函数 ReturnThread(func, params)

计时线程执行的函数。在其内部以参数 params 执行输入函数 func，并将返回值记录于

函数参数 ReturnThread.result；若捕获到异常则使用 ReturnThread.result 记录异常内容。

函数 timer(timeleft, func, params)

计时模块的接口，包含返回值传递、异常处理、超时处理内容。

参数：

timeleft：输入函数最大可执行时间，使用时输入为当前玩家剩余时间

func：待执行函数，使用时输入为玩家 load 或 play 函数

params：函数调用的参数元组

实现方式：

首先初始化函数为 ReturnThread、参数为(func, params)的线程，在运行该线程时 join 该线程最大 timeleft 秒时间，并使用 time.perf_counter 对阻塞过程进行计时。

在阻塞状态结束后，检查线程存活状态，若线程还在运行，或计时结果大于剩余时间，则抛出超时异常。在线程正常结束的情况下，检查 ReturnThread.result 类型，若为异常实例说明 func 函数运行时报错，将该异常抛出；否则返回该结果与消耗时间的元组。