棋类对战平台 – 第二阶段设计文档

目录

设计思路

在第一阶段五子棋和围棋的基础上，第二阶段需要扩展黑白棋、AI 对战、用户管理、录像回放和网络对战等功能。我的整体设计思路是采用分层架构，将系统划分为表示层（GUI）、业务层（Platform）、游戏层（Game/Player/AI）和数据层（Board/User），各层职责清晰、耦合度低。

对于新增的黑白棋功能，我选择继承现有的 Board 和 Game 抽象基类，只需实现黑白棋特有的翻转规则和胜负判断，无需修改已有的五子棋和围棋代码，保证了第一阶段功能的稳定性。AI 系统采用策略模式，定义统一的 AIStrategy 接口，三级 AI 分别实现随机落子、评估函数和 Alpha-Beta 剪枝算法，可在运行时动态切换。用户管理模块采用单例模式的 UserManager 统一管理注册、登录和战绩统计，数据持久化到本地 JSON 文件。录像回放功能通过 Replayer 类记录每一步棋的时间戳和位置，支持前进、后退和跳转。网络对战采用 C/S 架构，服务器维护权威游戏状态，客户端通过回调机制同步界面。

整体设计遵循面向对象的 SOLID 原则：Board 只管棋盘状态（单一职责）、新增游戏类型只需继承基类（开闭原则）、HumanPlayer 和 AIPlayer 可互换使用（里氏替换）、接口精简不强迫实现多余方法（接口隔离）、高层模块依赖抽象接口（依赖倒置）。

选用的设计模式

工厂模式：GameFactory 根据游戏类型字符串创建对应的 Game 实例，AIFactory 根据等级创建对应的 AI 策略，PlayerFactory 根据玩家类型创建 HumanPlayer 或 AIPlayer。工厂模式将对象创建逻辑封装起来，调用方无需关心具体类的构造细节，便于后续扩展新的游戏类型或 AI 算法。

策略模式：AI 模块定义 AIStrategy 接口，包含 get_move()和 get_level()两个方法。RandomAI、EvalAI、MCTSAI 三个具体策略类分别实现不同级别的算法。AIPlayer 持有 AIStrategy 接口引用，可在创建时或运行时切换不同策略，实现了算法与使用者的解耦。

单例模式：UserManager 采用单例模式，通过 get_instance()类方法获取全局唯一实例，确保整个应用中用户数据的一致性，避免多个实例导致的数据不同步问题。

外观模式：GamePlatform 作为外观类，整合了 Game、Player、UserManager、Replayer 等子系统，对外提供 create_game()、make_move()、save_replay()等简洁统一的接口，GUI 层只需与 GamePlatform 交互，无需了解底层复杂的协作关系。

模板方法模式：Game 抽象基类定义了游戏流程的骨架，包括 make_move()中的落子→检查胜负→切换玩家的固定流程，具体的规则判断由子类 GomokuGame、GoGame、OthelloGame 各自实现。

观察者模式：网络模块中 NetworkClient 通过回调函数（on_state_update、on_game_over 等）通知 GUI 状态变化，GUI 注册这些回调后，当服务器推送消息时自动触发界面刷新，实现了网络层与表示层的松耦合。

关键类和函数的简单说明

棋盘模块（board.py）

Board 是所有棋盘的抽象基类，定义了棋盘的基本属性和操作接口。__init__()初始化棋盘大小和二维网格，place_stone()在指定位置放置棋子并检查合法性，get_stone()获取某位置的棋子颜色，is_valid_position()判断坐标是否在棋盘范围内，copy()返回棋盘的深拷贝用于 AI 搜索。

```
class Board:
    def __init__(self, size):
        self.size = size
        self.grid = [[None for _ in range(size)] for _ in range(size)]

    def place_stone(self, row, col, color):
        if not self.is_valid_position(row, col):
            raise ValueError("无效位置")
        if self.grid[row][col] is not None:
            raise ValueError("该位置已有棋子")
        self.grid[row][col] = color
```

OthelloBoard 继承 Board，新增 count_stones()方法统计指定颜色的棋子数量，用于黑白棋的比分显示和胜负判断。

```
class OthelloBoard(Board):
    def __init__(self, size=8):
        super().__init__(size)

    def count_stones(self, color):
        count = 0
        for row in self.grid:
            for stone in row:
                if stone == color:
```

```
                    count += 1
        return count
```

游戏模块（game.py）

Game 是游戏逻辑的抽象基类，维护当前玩家、游戏状态、落子历史等属性。make_move()是模板方法，子类需实现具体的落子规则；check_winner()检查胜负；get_valid_moves()返回当前玩家的所有合法位置；undo()实现悔棋，从历史记录中恢复上一状态。

```
class Game:
    def __init__(self, board_size):
        self.board = None  # 子类初始化具体棋盘
        self.current_player = 'black'
        self.game_over = False
        self.winner = None
        self.move_history = []

    def switch_player(self):
        self.current_player = 'white' if self.current_player == 'black' else 'black'
```

OthelloGame 实现黑白棋规则，__init__()中调用_setup_initial_position()放置初始四颗棋子。make_move()先调用_get_flips()获取需要翻转的棋子列表，然后执行翻转；get_valid_moves()遍历所有空位，检查每个位置是否能翻转对手棋子；check_winner()在棋盘填满或双方都无合法位置时比较棋子数量判定胜负。

```
class OthelloGame(Game):
    def __init__(self, size=8):
        super().__init__(size)
        self.board = OthelloBoard(size)
        self._setup_initial_position()

    def _setup_initial_position(self):
        mid = self.board.size // 2
        self.board.grid[mid-1][mid-1] = 'white'
        self.board.grid[mid-1][mid] = 'black'
        self.board.grid[mid][mid-1] = 'black'
        self.board.grid[mid][mid] = 'white'

    def _get_flips(self, row, col, color):
        """获取落子后需要翻转的棋子位置列表"""
        opponent = 'white' if color == 'black' else 'black'
        directions = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
        all_flips = []
```

```
        for dr, dc in directions:
            flips = []
            r, c = row + dr, col + dc
            while 0 <= r < self.board.size and 0 <= c < self.board.size:
                if self.board.grid[r][c] == opponent:
                    flips.append((r, c))
                elif self.board.grid[r][c] == color:
                    all_flips.extend(flips)
                    break
                else:
                    break
                r, c = r + dr, c + dc
        return all_flips
```

玩家模块（player.py）

Player 是玩家的抽象基类，定义 is_human()判断是否为人类玩家，get_move()获取落子位置。HumanPlayer 的 set_move()由 GUI 调用设置待落子位置，get_move()返回该位置；AIPlayer 持有 AIStrategy 实例，get_move()委托给策略对象计算。

```
class HumanPlayer(Player): def __init__(self, color, user=None): super().__init__(color) self.user = user self.pending_move = None def is_human(self): return True def set_move(self, row, col): self.pending_move = (row, col) def get_move(self, game): move = self.pending_move self.pending_move = None return move class AIPlayer(Player): def __init__(self, color, strategy): super().__init__(color) self.strategy = strategy def is_human(self): return False def get_move(self, game): return
```

self.strategy.get_move(game, self.color)AI 模块（ai/）

RandomAI 实现一级 AI，get_move()从所有合法位置中随机选择一个返回，算法简单但完全随机，作为基准对照。

```
class RandomAI:
    def get_move(self, game, color):
        valid_moves = game.get_valid_moves()
        if valid_moves:
            return random.choice(valid_moves)
        return None

    def get_level(self):
        return 1
```

EvalAI 实现二级 AI，采用评估函数计算每个合法位置的分数，选择得分最高的位置。_evaluate_position()综合考虑位置权重（角落最高、边缘次之、中心较低）和棋型价值（能翻转的棋子数），确保能稳定战胜一级 AI。

```
class EvalAI:
    def __init__(self):
        self.position_weights = [
            [100, -20, 10, 5, 5, 10, -20, 100],
            [-20, -50, -2, -2, -2, -2, -50, -20],
            # ... 8x8 权重矩阵
        ]

    def get_move(self, game, color):
        valid_moves = game.get_valid_moves()
        best_move = None
        best_score = float('-inf')
        for row, col in valid_moves:
            score = self._evaluate_position(game, row, col, color)
            if score > best_score:
                best_score = score
                best_move = (row, col)
        return best_move
```

MCTSAI 实现三级 AI，采用 Alpha-Beta 剪枝搜索算法。get_move()首先检查优先级情况：自己能连五则立即获胜、对手能连五则必须阻挡、自己能活四则必胜。若无紧急情况则调用_alphabeta()进行深度搜索，搜索深度为 4 层，_evaluate_board()对整个棋盘进行评估，综合计算各种棋型的分数。

```
class MCTSAI:
    SCORES = {'five': 100000, 'open_four': 50000, 'four': 5000, 'open_three': 3000}

    def get_move(self, game, color):
        # 优先级检测
        winning_moves = self._find_all_winning_moves(game.board, color)
        if winning_moves:
            return winning_moves[0]

        opponent = 'white' if color == 'black' else 'black'
        block_moves = self._find_all_winning_moves(game.board, opponent)
        if block_moves:
            return block_moves[0]

        # Alpha-Beta 搜索
        return self._alphabeta_search(game, color)
```

```
    def _alphabeta(self, board, depth, alpha, beta, maximizing, color):
        if depth == 0:
            return self._evaluate_board(board, color)
        # ... 递归搜索与剪枝
```

用户模块（user.py）

User 类封装用户信息，包括用户名、密码哈希、对战场次和胜场。get_win_rate()计算胜率，to_dict()和 from_dict()实现序列化与反序列化。

```
class User:
    def __init__(self, username, password):
        self.username = username
        self.password_hash = self._hash_password(password)
        self.games = 0
        self.wins = 0

    def get_win_rate(self):
        if self.games == 0:
            return 0.0
        return self.wins / self.games
```

UserManager 采用单例模式管理所有用户，register()注册新用户并检查用户名是否重复，login()验证密码并返回 User 对象，update_user_stats()更新指定用户的战绩，save()将所有用户数据持久化到 JSON 文件。

```
class UserManager:
    _instance = None

    @classmethod
    def get_instance(cls, data_file='users.json'):
        if cls._instance is None:
            cls._instance = cls(data_file)
        return cls._instance

    def register(self, username, password):
        if username in self.users:
            raise ValueError("用户名已存在")
        user = User(username, password)
        self.users[username] = user
        self.save()
        return user

    def login(self, username, password):
```

```
        if username not in self.users:
            raise ValueError("用户不存在")
        user = self.users[username]
        if not user.verify_password(password):
            raise ValueError("密码错误")
        return user
```

录像回放模块（replay.py）

Replayer 负责录像的记录和回放。record_move()记录每步棋的行列、颜色和时间戳；save()将录像数据序列化为 JSON 保存到文件；load()从文件加载录像；next()和 prev()实现前进后退；goto()跳转到指定步骤。

```
class Replayer:
    def __init__(self):
        self.game_type = None
        self.board_size = 0
        self.moves = []
        self.current_step = 0

    def record_move(self, row, col, color, timestamp):
        self.moves.append({
            'row': row, 'col': col,
            'color': color, 'time': timestamp
        })

    def next(self):
        if self.current_step < len(self.moves):
            move = self.moves[self.current_step]
            self.current_step += 1
            return move
        return None
```

网络模块（network.py）

GameServer 管理网络对战，start()启动服务器监听指定端口，_handle_client()为每个客户端启动独立线程处理消息。_handle_join()处理加入请求并分配颜色，_handle_move()验证落子合法性并广播状态更新，_broadcast_state()将当前棋盘状态推送给所有客户端。

```
class GameServer:
    def __init__(self, host='0.0.0.0', port=9999):
        self.clients = {}  # {socket: {'username': str, 'color': str}}
```

```
        self.board = None
        self.current_player = 'black'

    def _handle_join(self, client_socket, msg):
        username = msg.get('username')
        # 分配颜色
        assigned_color = 'black' if 'black' not in taken_colors else 'white'
        self.clients[client_socket]['color'] = assigned_color
        # 发送颜色分配消息
        self._send_to_client(client_socket, {
            'type': 'color_assigned',
            'color': assigned_color
        })
```

NetworkClient 连接服务器并处理消息。connect()建立 TCP 连接并发送 join 请求，_receive_loop()在独立线程中循环接收消息，根据消息类型调用对应的处理函数。on_state_update 等回调函数由 GUI 注册，实现界面同步更新。

```
class NetworkClient:
    def connect(self, host, port, username, preferred_color):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((host, int(port)))
        self._send({'type': 'join', 'username': username, 'color': preferred_color})

    def _handle_state_update(self, msg):
        # 更新本地状态
        self.current_player = msg.get('current_player')
        self.board_data = msg.get('board')
        # 触发回调通知 GUI
        if self.on_state_update:
            self.on_state_update(self.get_game_state())
```

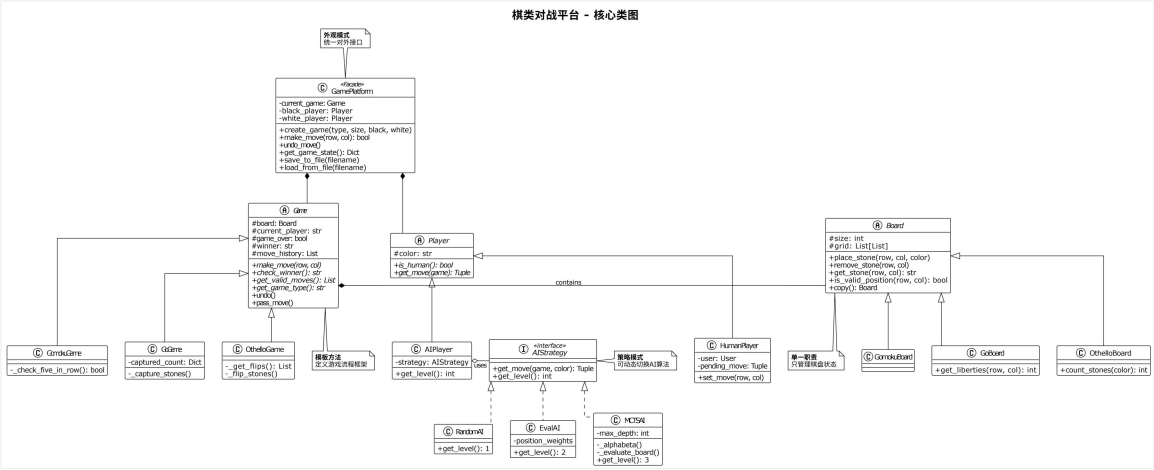
平台模块（platform.py）

GamePlatform 作为外观类，create_game()通过工厂创建游戏和玩家实例；make_move()协调落子流程并记录到录像；get_game_state()返回当前状态供 GUI 显示；save_to_file()和 load_from_file()实现存档功能；save_replay()和 load_replay()实现录像功能。

UML 图

下图展示了系统的核心类关系。Board 和 Game 采用继承结构，各有三个具体子类对应三种棋类游戏；Player 有 HumanPlayer 和 AIPlayer 两个子类，AIPlayer

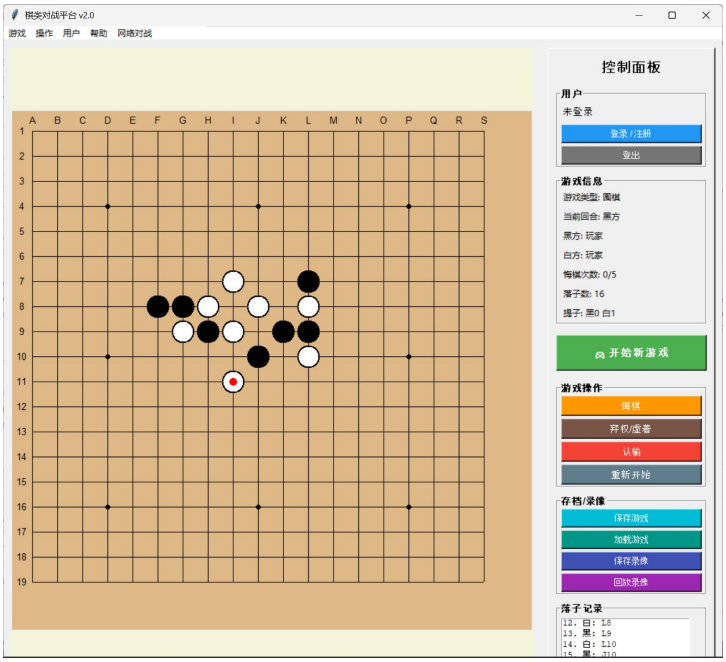
通过组合持有 AIStrategy 接口实现策略模式；GamePlatform 作为外观类组合了 Game 和 Player，对外提供统一接口。



测试用例和程序测试过程的输入输出

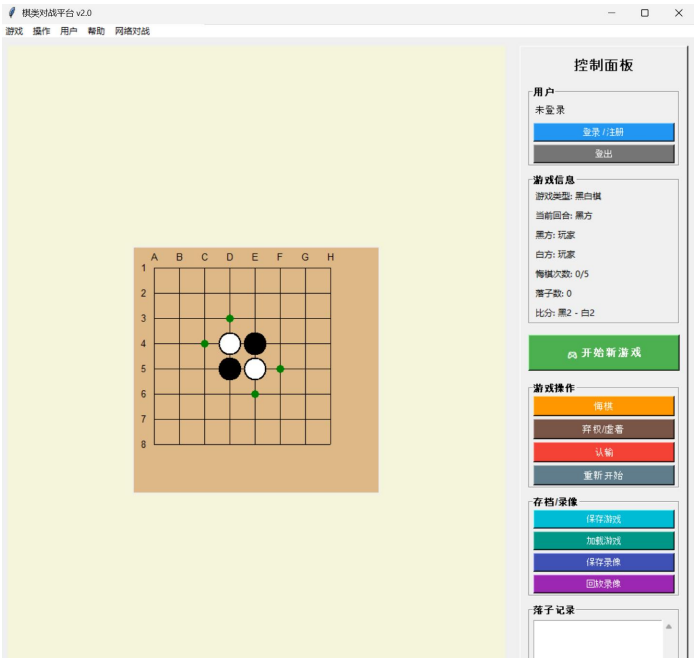
棋盘基本操作测试

测试棋盘的创建、落子、边界检查和重复落子等基本功能，验证 Board 类的核心方法是否正确工作。



黑白棋规则测试

测试黑白棋的初始布局、合法位置判断和翻转逻辑，这是第二阶段新增的核心功能。

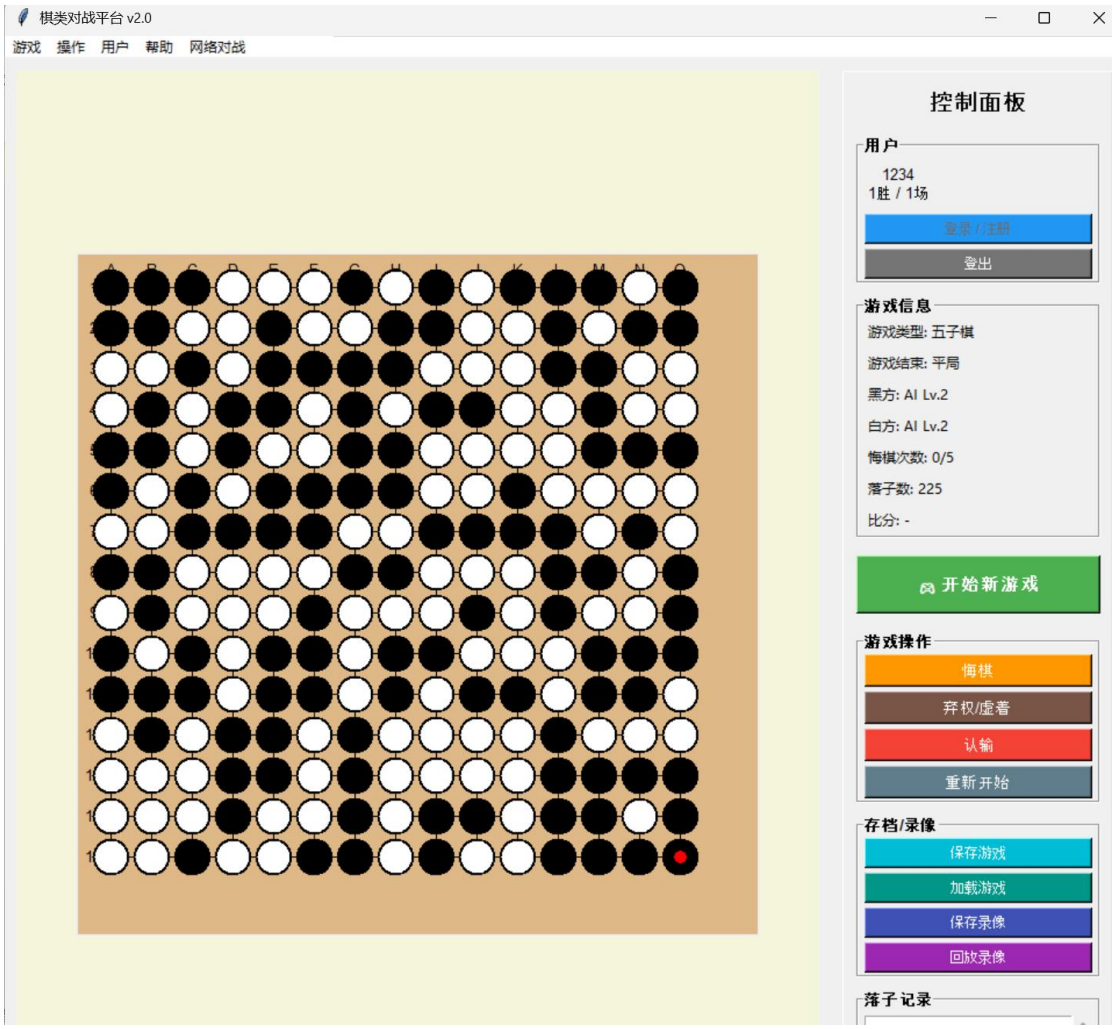


用户系统测试

测试用户注册、登录、密码验证和战绩更新等功能，验证用户账户管理模块的正确性。可以看到，**登录之后，正常显示用户的胜场，用户名等信息**

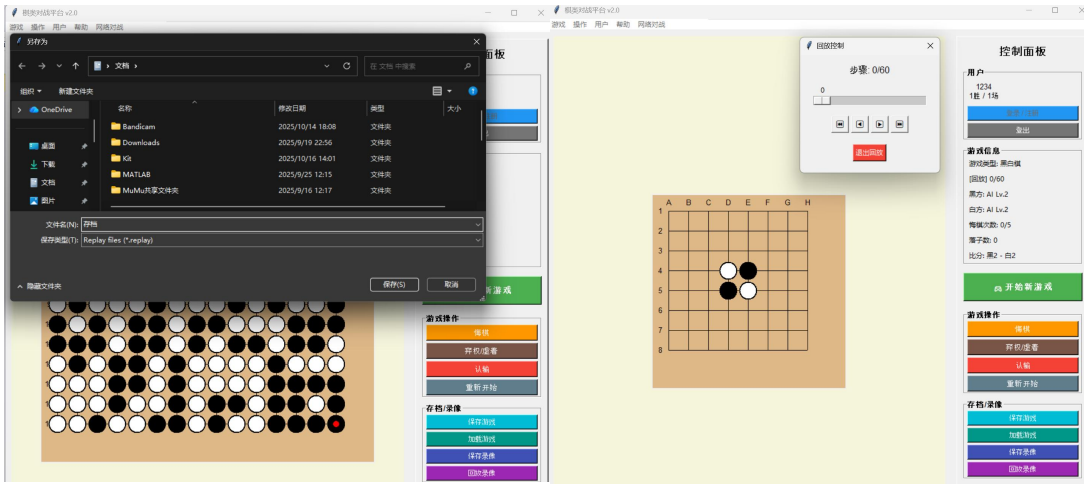
AI 功能测试(三级 AIbonus)

测试三级 AI 的等级标识和落子有效性，确保 AI 能在合法位置落子。



存档与录像测试

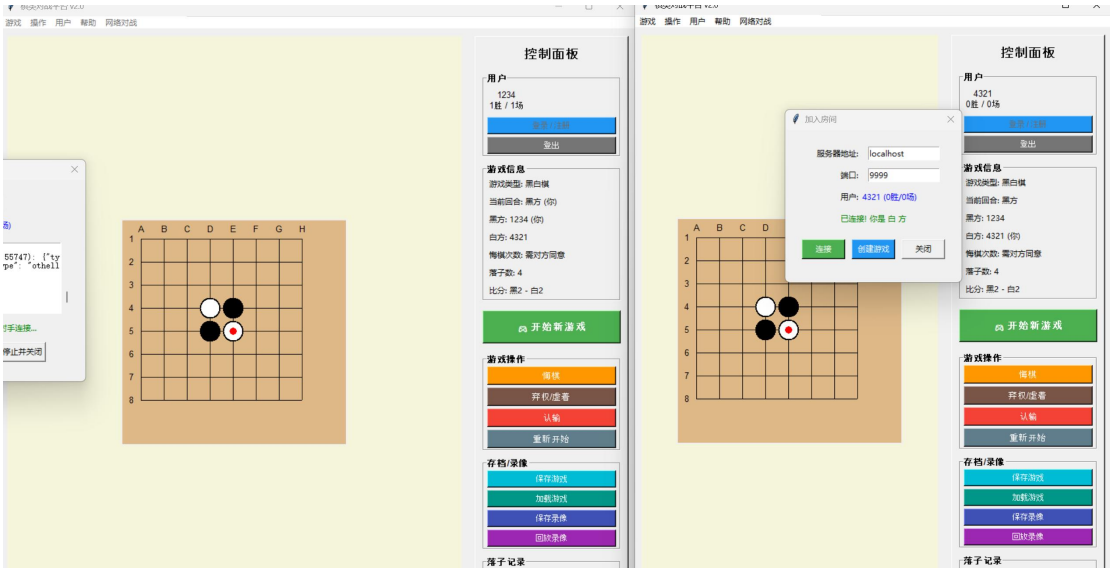
测试游戏存档的保存加载和录像的记录回放功能。右下角的保存录像按钮可以保存数据；导入之后，可以通过拖动进度条到达相应的步数



网络对战测试（bonus）

房主开启服务端，用于通过客户端加入游戏

便可进入对战，然后点击创建游戏便可选择相应的游戏进行对战



总结

本项目完成了作业要求的全部功能。黑白棋方面，实现了 8x8 棋盘的初始布局、合法位置检测、落子翻转和胜负判定，在夹击位置落子后被夹棋子正确翻转，当一方无合法位置时自动弃权，棋盘填满或双方都无合法位置时正确比较棋子数量判定胜负。AI 功能方面，实现了三级 AI 算法：一级 RandomAI 在合法位置随机落子，二级 EvalAI 基于位置权重和棋型评估函数选择最优位置，三级 MCTSAI 使用 Alpha-Beta 剪枝搜索配合优先级检测（连五、活四、冲四、活三）；测试结果显示二级 AI 能稳定战胜一级 AI（5 局胜 5 局），三级 AI 能稳定战胜一级和二级 AI。用户账户管理方面，实现了注册、登录、密码验证功能，平台记录每个账号的对战场次和胜场，对局结束后自动更新战绩，数据持久化保存到本地 JSON

文件，界面上显示用户名和战绩信息。录像与回放方面，实现了对局过程中每步棋的时间戳记录，支持保存录像文件和加载回放，回放模式下可以前进、后退、跳转到指定步骤观看对局过程。网络对战方面（可选+10%），实现了局域网内 TCP/IP 通信的服务器-客户端架构，双方可选择先后手，棋盘状态和落子指令在两个客户端间实时同步。整体设计采用工厂模式、策略模式、单例模式、外观模式等多种设计模式，遵循面向对象设计原则，保证了类的可复用性和可扩展性，新增黑白棋游戏后原有五子棋和围棋功能不受影响。