

面向对象大作业： Zenith 棋类对战平台

第二阶段：详细设计与实现报告

[GitHub Repo](#) | [Demo Video](#) | [Live Demo](#)

FastAPI

React + TypeScript

WebSocket

MySQL

本项目已经完成了所有基础要求与附加要求！

目录 (Table of Contents)

1. 引言

- 1.1 项目背景与愿景
- 1.2 核心功能矩阵
- 1.3 技术选型解析

2. 系统架构设计

- 2.1 架构分层
- 2.2 组件交互图
- 2.3 混合通信模型

3. 面向对象设计与模式应用

- 3.1 策略模式——AI 系统的核心
- 3.2 模板方法模式——游戏规则的抽象与复用
- 3.3 单例模式——全局游戏状态的唯一入口
- 3.4 核心类图

4. 关键模块详解

- 4.1 在线房间与会话管理
- 4.2 实时对战交互协议
- 4.3 游戏规则引擎

- [4.4 存档与回放系统](#)

5. 数据库设计

- [5.1 实体关系图](#)

- [5.2 关键字段说明](#)

6. 接口设计

- [6.1 核心 REST API](#)

- [6.2 WebSocket 事件](#)

7. 测试与验证

- [7.1 测试场景 A: 黑白棋规则边界测试](#)

- [7.2 测试场景 B: 在线对战异常流程](#)

- [7.3 测试场景 C: 僵尸房间自动销毁与显式离开](#)

- [7.4 测试场景 D: 存档与加载](#)

8. 总结与展望

1. 引言

1.1 项目背景与愿景

本项目 **Zenith Board Battle Platform** 旨在通过**面向对象 (Object-Oriented)** 的设计思想，解决多棋种规则的统一抽象、即时状态同步的复杂性以及 AI 策略的灵活扩展问题。在第一阶段实现了基础游戏逻辑（五子棋、围棋）的基础上，第二阶段重点引入了**实时在线对战、进阶 AI 策略、黑白棋 (Reversi)** 以及完善的**房间管理与回放系统**。

我们的愿景是打造一个不仅能玩游戏，更能作为**现代软件工程实践范本**的系统。它集成了用户认证、即时通讯、智能决策和数据持久化等核心模块，展示了如何在一个分布式系统中维持代码的清晰度与健壮性。我们特别关注解决分布式系统中的状态一致性问题（如断线重连、房间状态同步），以及如何通过设计模式降低系统的复杂度。

1.2 核心功能矩阵

模块 (Module)	功能点 (Feature)	复杂度	描述 (Description)
核心对弈	多棋种支持	★★★	支持五子棋 (Gomoku), 围棋 (Go), 黑白棋 (Reversi/Othello) 的完整规则判定。
在线大厅	房间管理	★★★	支持创建房间、加入房间、实时列表刷新、空房间自动销毁机制，确保资源不被浪费。
实时对战	状态同步	★★★★★	基于 WebSocket 的毫秒级落子同步，支持断线重连与状态恢复，解决“僵尸房间”问题。

模块 (Module)	功能点 (Feature)	复杂度	描述 (Description)
智能 AI	多级策略	★★★★	集成 Greedy、Minimax (Alpha-Beta 剪枝)、MCTS 算法，支持人机对战及 AI 自我对弈演示。
交互系统	准备/悔棋/投降	★★★★	包含“准备大厅”、“请求交换阵营”、“请求悔棋”等复杂的握手流程，确保对战公平性。
回放系统	云端存档	★★★	支持将对局保存至云端，随时加载复盘或继续本地演练，支持 JSON 格式的导入导出。

1.3 技术选型解析

- Backend (FastAPI)**: 选择 FastAPI 是因为它原生支持异步 (AsyncIO)，这对于处理大量 WebSocket 长连接至关重要。其基于 Pydantic 的类型系统保证了前后端数据契约的严谨性，自动生成的文档也极大地方便了开发调试。
- Frontend (React + TypeScript)**: TypeScript 的静态类型检查极大减少了前端逻辑错误，React 的组件化架构使得复用 UI (如棋盘、控制栏、模态框) 变得极其简单。Vite 提供了极速的构建体验。
- Communication (WebSocket)**: 传统的 HTTP 轮询无法满足棋类对战的实时性要求。WebSocket 提供了全双工通信通道，实现了服务器主动推送 (Server Push)，使得落子、状态更新、对手离开等事件能即时触达客户端。
- Database (MySQL)**: 使用 MySQL 存储用户关系数据，同时利用其 JSON 字段特性存储非结构化的棋局历史 (`moves_json`)，兼顾了关系查询 (如用户战绩统计) 与文档存储 (如复杂棋局回放) 的优势。

2. 系统架构设计

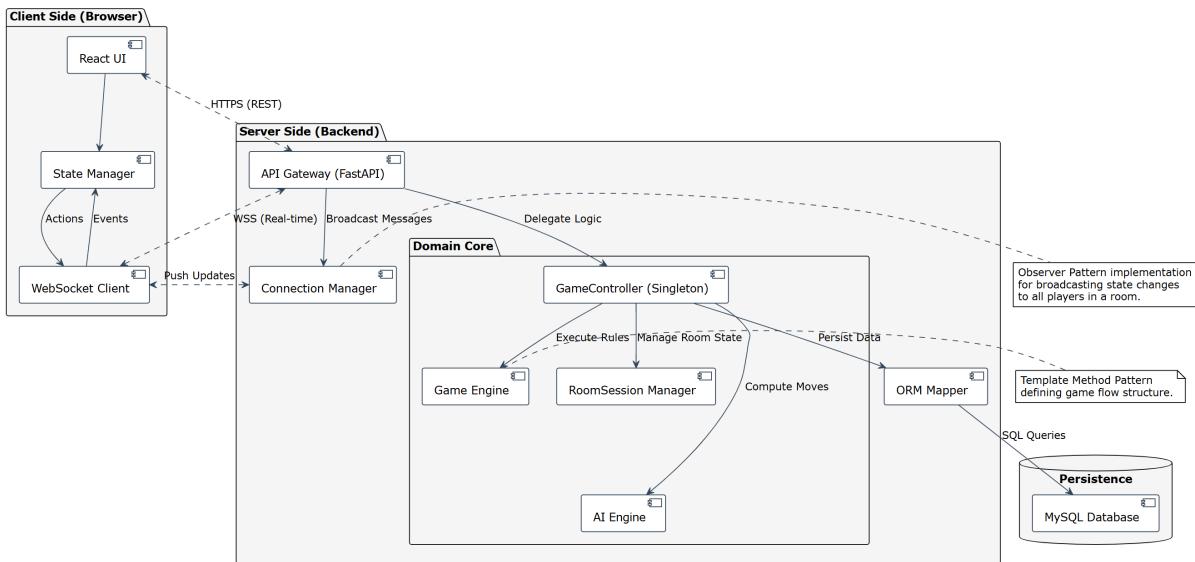
本系统采用经典的分层架构模式，强调关注点分离 (Separation of Concerns)，确保系统的可维护性和可测试性。

2.1 架构分层

- 表现层 (Presentation Layer)**: 运行在浏览器中的 React 单页应用 (SPA)。负责 UI 渲染、用户交互捕获以及与后端的通信适配。它维护了本地的应用状态 (如当前棋盘、用户信息、房间列表)。
- 应用服务层 (Application Service Layer)**: FastAPI 提供的 REST 接口与 WebSocket 端点。负责路由分发、请求校验、用户认证 (JWT)、会话管理。它是系统的入口，负责协调领域层与基础设施层。
- 领域层 (Domain Layer)**: 系统的核心。包含 `GameController` (核心控制器)、`AbstractGame` 及其子类 (游戏规则)、`AIStrategy` (智能策略)。这一层封装了所有的业务规则，不依赖于具体的 HTTP 框架或数据库实现。
- 基础设施层 (Infrastructure Layer)**: 负责数据持久化 (SQLAlchemy) 和底层网络通信 (`ConnectionManager`)。它为上层提供具体的服务实现。

2.2 组件交互图

下图展示了系统内部各模块及其与外部世界的交互关系。前端通过 HTTP 与 WebSocket 双通道与后端通信，后端通过 GameController 协调 AI 和数据库。



2.3 混合通信模型

我们采用了 HTTP 与 WebSocket 互补的通信策略，以达到最佳的性能与开发效率平衡：

- **HTTP REST:** 用于资源导向的操作。例如用户注册、登录（获取 Token）、创建房间（获取 RoomID）、获取回放列表、保存存档。这些操作是无状态的，适合标准的 Request-Response 模型。
- **WebSocket:** 用于事件导向的操作。例如落子（MOVE）、悔棋（UNDO）、聊天/状态同步、断线通知。WebSocket 提供了全双工通信通道，能够维持会话上下文（RoomSession），是实现在线对战的关键。通过 ConnectionManager，后端可以向特定房间内的所有用户广播消息。

3. 面向对象设计与模式应用

在本平台的架构与实现中，我们深入贯彻了面向对象的设计思想，并通过恰当应用设计模式，有效提升了系统的可维护性、可扩展性和鲁棒性。这部分将结合具体代码结构，详细阐述所选模式及其背后的设计考量。

3.1 策略模式——AI 系统的核心

设计考量：

棋类游戏的 AI 算法多种多样，从简单的贪心到复杂的蒙特卡洛树搜索，且每种算法针对不同棋种的实现细节也各不相同。若将这些逻辑硬编码在游戏规则中，将导致代码臃肿、难以管理和扩展。

解决方案：

我们采用了策略模式来解耦 AI 算法与其使用环境。

- **Context (上下文):** GameController 类。它不直接包含 AI 算法的实现，而是持有一个 AIStrategy 接口的引用。当需要 AI 落子时，它只需调用策略接口的统一方法，无需关心具体是哪种 AI。

- **Strategy Interface (策略接口):** `AIStrategy` (一个抽象基类)。它定义了所有 AI 算法必须遵循的统一接口，即 `make_move(game: AbstractGame, player: Player) -> Tuple[int, int]`。
- **Concrete Strategies (具体策略):** 实现了 `AIStrategy` 接口的各个具体 AI 算法类，例如：
 - `GreedyGomokuAI` / `GreedyReversiAI`: 实现简单的贪心策略，计算每一步的即时收益。
 - `MinimaxReversiAI` / `MinimaxGomokuAI`: 实现基于 Alpha-Beta 剪枝的极大极小搜索算法，向后预判多步。
 - `MCTSReversiAI`: 实现基础的蒙特卡洛树搜索算法，通过模拟对局寻找最优解。

体现的 OO 原则：

- **Open-Closed Principle (OCP - 开闭原则):** 当需要引入新的 AI 算法 (例如，更强大的神经网络 AI) 时，只需创建新的具体策略类，实现 `AIStrategy` 接口，无需修改 `GameController` 或现有的 AI 代码。
- **Dependency Inversion Principle (DIP - 依赖倒置原则):** `GameController` 依赖于 `AIStrategy` 抽象，而不是具体的 AI 实现。

3.2 模板方法模式——游戏规则的抽象与复用

不同棋类游戏的核心流程 (如落子前的校验、落子后的状态更新、胜负判定、玩家切换) 有共通之处，但具体的规则细节 (如合法性、棋子翻转、提子、连珠) 各异。

解决方案：

我们利用**模板方法模式**来统一管理这些相似流程。

- **Abstract Class:** `AbstractGame`。它定义了棋类游戏的通用骨架算法 `make_move(x, y)`。这个方法包含一系列步骤，其中一些步骤是通用的 (如记录历史、切换玩家)，另一些则是抽象的 (Hook Methods)，需要由子类具体实现。
- **Concrete Classes:** `GoGame`, `GomokuGame`, `ReversiGame`。它们继承 `AbstractGame`，并实现或覆盖其抽象方法及 Hook Methods，以适配各自的棋类规则。
- **Hook Methods (钩子方法):**
 - `_create_board()` : 创建具体棋种的棋盘实例。
 - `_is_valid_move(x, y)` : 检查具体棋种的落子合法性 (例如围棋的气、黑白棋的夹击)。
 - `check_game_over()` : 检查具体棋种的终局条件和胜利者。

体现的 OO 原则：

- **Don't Repeat Yourself (DRY):** 避免了在每个棋类实现中重复编写相同的流程逻辑。
- **Single Responsibility Principle (SRP - 单一职责原则):** `AbstractGame` 专注于定义游戏流程，具体规则的实现则下放给子类。

3.3 单例模式 —— 全局游戏状态的唯一入口

在多线程/多进程环境中，尤其是在 ASGI (Asynchronous Server Gateway Interface) 应用中，确保对共享资源的访问一致性至关重要。`GameController` 管理着所有在线游戏的内存状态 (`_active_games`, `_room_sessions`)，它必须是唯一的。

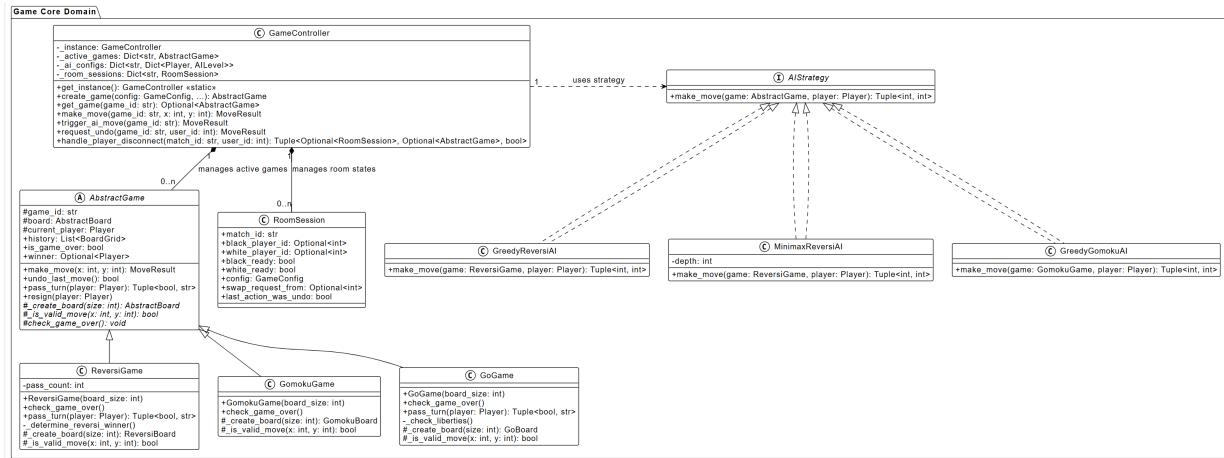
解决方案:

我们将 `GameController` 设计为单例模式。

- **实现方式:** 通过重写类的 `__new__` 方法，确保每次实例化 `GameController` 时，都返回同一个实例。
- **作用:** 保证了无论多少个 WebSocket 连接或 HTTP 请求并发访问，在处理游戏逻辑时，都操作的是内存中同一份实时游戏状态数据，从而避免了数据不一致和竞态条件。这对于房间状态同步至关重要。

3.4 核心类图

下图展示了系统核心类（游戏逻辑、AI、控制器）的继承与关联关系，突出了面向对象的设计。



图解说明：

- `AbstractGame` 是所有游戏的基类，定义了通用接口。
- `ReversiGame`, `GoGame`, `GomokuGame` 继承自 `AbstractGame`，实现了具体规则。
- `GameController` 是核心控制类（单例），聚合了多个 `AbstractGame` 实例（管理进行中的游戏）和 `RoomSession`（管理房间状态）。
- `AIStrategy` 是策略接口，具体 AI 算法类实现该接口，并被 `GameController` 依赖使用。

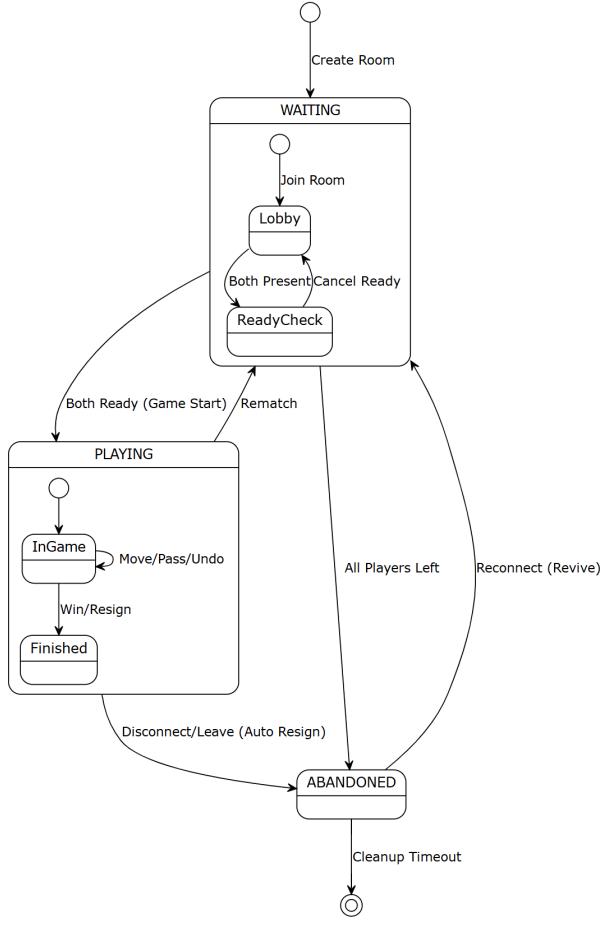
4. 关键模块详解

4.1 在线房间与会话管理

在线对战系统的核心在于如何管理房间的生命周期以及玩家的连接状态。我们引入了 `RoomSession` (内存会话) 与 `MatchStatus` (数据库状态) 双层管理机制。

4.1.1 房间生命周期状态机

房间的状态流转不仅依赖于玩家的操作，还受到网络连接状态的影响。



状态详解:

- **WAITING**: 房间已创建，等待对手加入，或者大厅内的准备阶段。此时房间在列表可见。
- **PLAYING**: 双方均已准备就绪，游戏正式开始，棋盘激活。此时房间在列表不可见（已满员）。
- **ABANDONED**: 房间内的所有玩家均已断开连接或显式离开。此状态下的房间将从大厅列表中移除。这是为了防止“僵尸房间”污染大厅。
- **COMPLETED**: 游戏已分出胜负，并已归档保存。

4.1.2 断线重连与自动销毁

为了解决网络不稳定导致的“掉线判负”误伤，以及“人走房空”导致的资源浪费，我们设计了一套健壮的机制：

- **主动离开 (Explicit Leave)**: 当玩家点击 "Leave Room" 时，前端发送 `LEAVE` 消息。后端立即将该玩家移出 Session，若房间变空，则立即将数据库状态更新为 `ABANDONED`。
- **意外断开 (Accidental Disconnect)**: 当 WebSocket 连接断开但未收到 `LEAVE` 消息时，后端触发 `handle_player_disconnect`。如果游戏正在进行，系统会自动判负；如果房间变空，则标记为 `ABANDONED`。
- **幽灵连接复活 (Ghost Connection Revival)**: 若玩家因网络闪断（或 React Strict Mode）导致短暂断开后立即重连，后端检测到房间处于 `ABANDONED` 状态但又有新连接进入，会自动将其复活为 `WAITING` 状态，保证用户体验的连续性。

4.2 实时对战交互协议

前后端通过 WebSocket 进行全双工通信，协议采用 JSON 格式。我们设计了丰富的信令来实现复杂的交互流程。

4.2.1 协议定义

Client -> Server Actions:

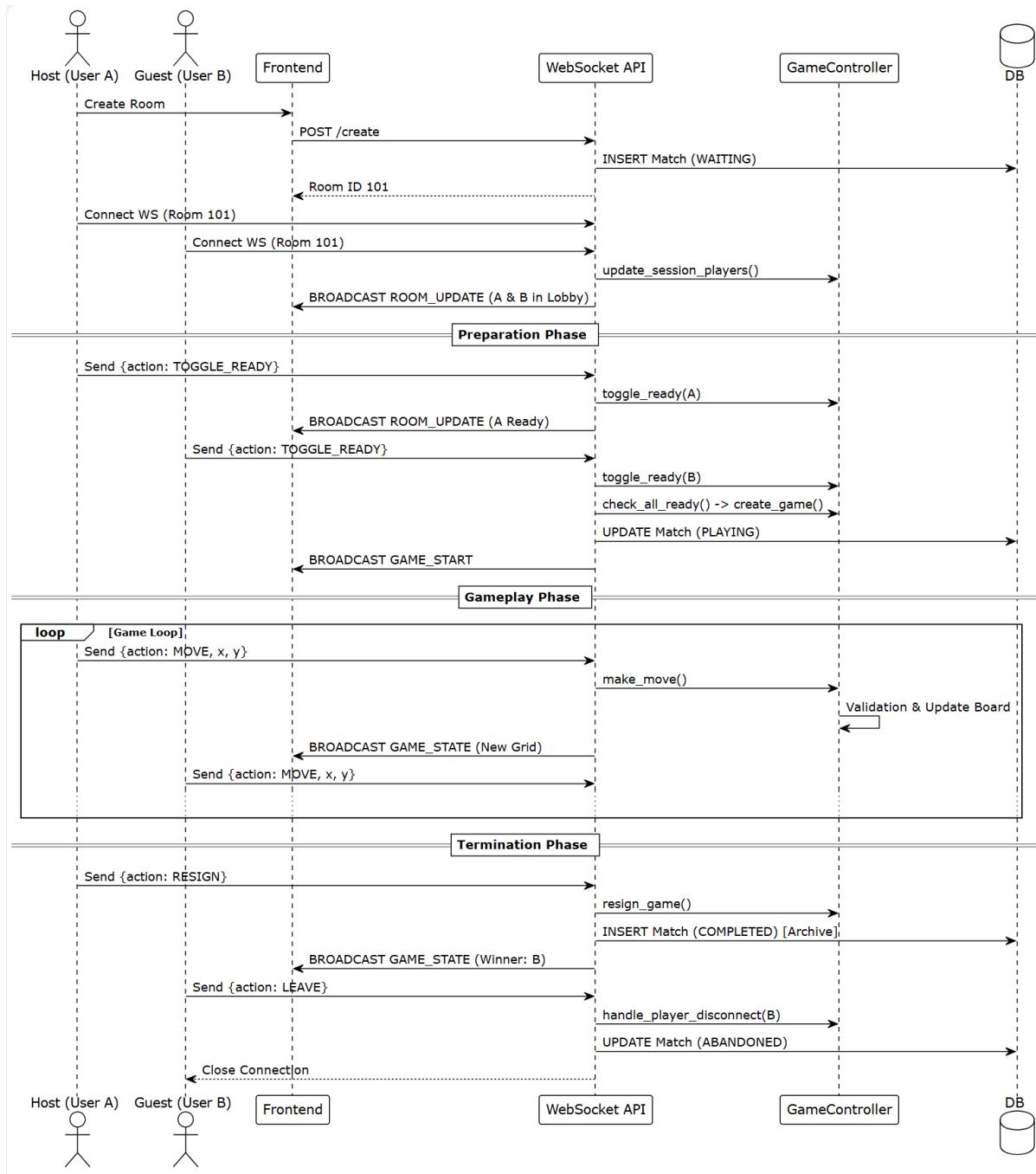
- `MOVE {x, y}`: 请求落子。
- `PASS`: 请求跳过回合（仅围棋/黑白棋）。
- `UNDO`: 请求悔棋（后端校验是否允许）。
- `RESIGN`: 认输。
- `TOGGLE_READY`: 切换准备状态。
- `REQUEST_SWITCH`: 请求交换执黑/执白。
- `APPROVE_SWITCH / REJECT_SWITCH`: 响应交换请求。
- `LEAVE`: 显式离开房间。
- `REMATCH`: 请求再来一局。

Server -> Client Events:

- `ROOM_UPDATE {session}`: 房间大厅状态更新（人员变动、准备状态）。
- `GAME_START {state}`: 游戏正式开始，下发初始盘面。
- `GAME_STATE {state}`: 盘面更新（落子后广播）。
- `ERROR {message}`: 操作非法提示。

4.2.2 对战时序图

下图展示了一场完整对局的交互流程：



4.3 游戏规则引擎

我们将每种棋类的规则封装在独立的类中，确保逻辑的纯粹性。

4.3.1 黑白棋 (Reversi) 翻转算法

黑白棋的核心在于落子后的“夹击翻转”。我们在 `ReversiBoard` 类中实现了 `_get_flippable_pieces(x, y, player)` 方法。

算法步骤：

- 方向向量:** 定义 8 个方向 `[(0,1), (1,1), (-1,0), ...]`。
- 射线探测:** 从落子点 `(x, y)` 开始，沿每个方向延伸。
- 模式匹配:** 寻找 `opponent -> Opponent -> ... -> Self` 的序列。

4. 收集坐标: 如果找到完整序列, 将中间的所有 `opponent` 坐标加入 `flippable` 列表。
5. 执行翻转: 如果 `flippable` 不为空, 则该步合法, 并更新 grid。

```
# 核心算法伪代码
for dx, dy in directions:
    line = []
    for i in range(1, size):
        nx, ny = x + dx*i, y + dy*i
        if grid[ny][nx] == opponent:
            line.append((nx, ny))
        elif grid[ny][nx] == player:
            flippable.extend(line) # 形成夹击, 收集路径上的棋子
            break
        else:
            break # 遇到空格或边界, 未形成夹击
```

4.3.2 悔棋逻辑

在线模式的悔棋比本地模式复杂, 因为涉及回合权的归属。我们实施了严格的限制:

- **规则:** 只有在**轮到对手下** (即我刚下完) 时, 我才能请求悔棋。
- **执行:** 撤销 1 步 (我的那一步), 状态回退到我落子前 (轮到我下)。
- **防抖:** 连续悔棋被禁止 (`last_action_was_undo` 标志位), 防止恶意干扰对手。

4.4 存档与回放系统

系统支持将当前的棋局状态保存为云端记录或本地文件。

4.4.1 数据序列化

由于 Python 后端使用 `snake_case` (如 `board_size`) 而 React 前端使用 `camelCase` (如 `boardsize`), 数据交换时的字段映射是一个挑战。

我们利用 **Pydantic** 的 `ConfigDict(populate_by_name=True)` 和 `Field(alias="...")` 特性, 完美解决了这个问题。

- **Save:** 前端发送 camelCase JSON -> 后端 Pydantic 自动解析为 `snake_case` 对象 -> 存入 DB (作为 JSON)。
- **Load:** 后端读取 DB -> Pydantic 序列化为 camelCase JSON -> 前端直接使用。

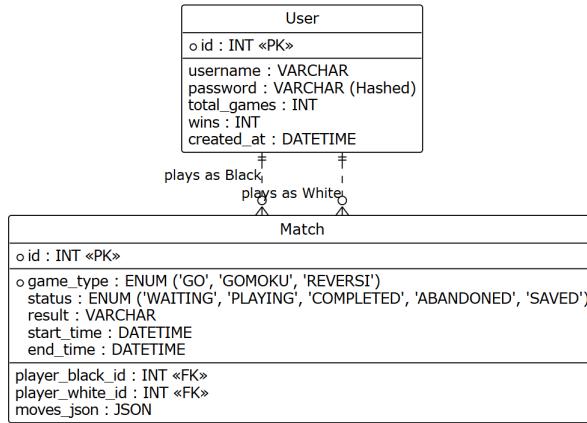
4.4.2 存档策略

- **Local Save:** 直接将 `GameState` 序列化为 `.json` 文件供用户下载。
- **Cloud Save:** 将当前状态作为一条新的 `Match` 记录存入数据库。
 - 如果是未结束的棋局, 状态标记为 `SAVED`, 结果为 `None`。
 - 加载 Cloud Save 时, 实际上是开启了一局新的 **Practice Mode** 游戏, 允许用户基于存档继续对弈 (人机或双人同屏), 但不影响原有的在线房间。

5. 数据库设计

数据库设计力求简洁，核心数据通过 JSON 字段扩展。

5.1 实体关系图



5.2 关键字段说明

- `User` 表:

- `id`: 主键。
- `username`, `password`: 认证信息。
- `total_games`, `wins`: 冗余统计字段，用于快速展示大厅排行榜。

- `Match` 表:

- `id`: 主键。
- `status`: 关键字段，`ENUM ('WAITING', 'PLAYING', 'COMPLETED', 'ABANDONED', 'SAVED')`，决定了房间在大厅的可见性及生命周期状态。
- `moves_json`: 这是一个 `JSON` 类型的非结构化字段，存储了完整的棋局历史。
 - 结构: `{ "meta": { "config": {...} }, "history": [{ "grid": [...] }, ...] }`
 - 优势: 前端可以利用 `history` 数组轻松实现“回放动画”、“上一步/下一步”跳转，而无需后端进行复杂的查询。同时，`meta` 字段存储了游戏配置（如 AI 难度、是否为人工），保证了回放时的上下文完整性。

6. 接口设计

6.1 核心 REST API

Method	Path	Description	Request Model	Response Model
Auth				
POST	/api/auth/login	用户登录	UserLogin	Token

Method	Path	Description	Request Model	Response Model
GET	/api/users/me	获取当前用户信息	(Header Auth)	UserResponse
Room				
POST	/api/rooms/create	创建房间	GameConfig	MatchInfo
GET	/api/rooms	获取房间列表 (仅WAITING)	-	MatchListResponse
POST	/api/rooms/{id}/join	加入房间	-	MatchInfo
Game				
POST	/api/game/start	开始本地/AI游戏	GameConfig	StartGameResponse
POST	/api/game/{id}/move	HTTP落子(Practice)	MakeMoveRequest	MoveResult
POST	/api/game/{id}/trigger_ai	触发AI落子	-	MoveResult
Replay				
GET	/api/replays/me	获取我的回放	-	MatchListResponse
POST	/api/replays/save	保存棋局到云端	SaveGameRequest	MatchInfo
DELETE	/api/replays/{id}	删除回放	-	{message: str}

6.2 WebSocket 事件

- **Endpoint:** /ws/game/{room_id}?token={token}
- **认证:** 连接建立时校验 JWT Token, 非法则关闭连接 (Code 1008)。
- **心跳:** 依赖 WebSocket 协议底层的 Ping/Pong 保持连接活跃。

消息Payload示例

Client -> Server: MOVE

```
{
  "action": "MOVE",
  "x": 7,
  "y": 7
}
```

Server -> Client: GAME_STATE

```
{
  "type": "GAME_STATE",
  "state": {
    "gameId": "123",
    "currentPlayer": "WHITE",
    "grid": [[null, "BLACK", ...], ...],
    "isGameOver": false
  }
}
```

7. 测试与验证

为了确保系统的健壮性，我们设计了多组测试用例，覆盖了规则逻辑、并发交互及异常恢复场景。

7.1 测试场景 A：黑白棋规则边界测试

- **前置条件:** 8x8 棋盘，标准开局（中央4子）。
- **操作:** 黑方落子于 (2, 3)。
- **预期结果:**
 1. (3, 3) 处的白子翻转为黑子。
 2. 棋盘状态更新，黑子数量增加 2，白子数量减少 1。
 3. 回合切换至白方。
- **操作:** 此时白方试图在 (0, 0) 落子（无邻接棋子）。
- **预期结果:** 后端返回错误 "Invalid move"，棋盘状态不变，回合不切换。

7.2 测试场景 B：在线对战异常流程

- **场景:** User A (Host) 与 User B (Guest) 正在对战。
- **步骤 1 (正常落子):** A 落子 -> B 收到更新。B 落子 -> A 收到更新。
- **步骤 2 (悔棋限制):**
 - 轮到 A 下。A 请求 UNDO。
 - **结果:** 后端拒绝 "Cannot undo after opponent has moved" (因为是轮到 A，说明 B 刚下完)。
 - A 落子。轮到 B。
 - A 立即请求 UNDO。
 - **结果:** 后端接受，回退 1 步，轮回 A。
- **步骤 3 (断线重连):**
 - B 刷新浏览器（模拟 WebSocket 断开）。
 - **后端:** handle_player_disconnect 触发，检测到游戏正在进行 -> **自动判负** -> B 输，A 赢。
 - A 的界面收到 GAME_STATE (Winner: A)。
 - B 重连进入房间 -> 看到 Lobby 界面（状态重置）或 游戏结束界面。

7.3 测试场景 C：僵尸房间自动销毁与显式离开

- **场景:** A 创建房间 101，此时 Room 101 在大厅列表可见 (WAITING)。
- **步骤:**
 - A 点击 "Leave Room"。
 - 前端发送 LEAVE 消息。
 - 后端收到 LEAVE -> 将 A 移出 Session -> 检测到 Session 为空 -> 删除内存 Game 实例 -> 更新 DB 状态为 ABANDONED -> 关闭连接。

- **验证:**

- 其他用户刷新大厅列表，Room 101 消失。
- 数据库 `matches` 表中 ID 101 的 `status` 为 `ABANDONED`。

7.4 测试场景 D：存档与加载

- **场景:** 本地 Practice 模式，人机对战中途。

- **步骤:**

- 点击 "Save" -> "Save to Cloud"。
- **结果:** 后端创建新 Match (Status: SAVED)，包含当前棋盘状态。
- 点击 "Load" -> "Load from Cloud"。
- **结果:** 列表显示刚才的存档。点击加载，棋盘恢复到保存时的状态，可以继续下棋。

8. 总结与展望

8.1 项目成果

经过两个阶段的迭代开发，**Zenith Board Battle Platform** 已从一个简单的本地棋类程序进化为一个功能完备的在线对战平台。

- **架构层面:** 实现了完全的前后端分离，后端采用异步模型，具备处理高并发连接的能力。
- **功能层面:** 成功攻克了黑白棋复杂规则、WebSocket 实时同步、断线重连恢复等技术难点。特别是解决了“僵尸房间”和“状态不同步”等复杂的分布式系统问题。
- **工程层面:** 严格遵循面向对象设计原则（策略、模板、单例），代码结构清晰，具有良好的可扩展性。

8.2 未来展望

- **AI 升级:** 引入基于深度学习（如 AlphaZero 架构）的 AI 模型，提供大师级挑战。
- **分布式部署:** 引入 Redis 作为 WebSocket 的 Pub/Sub 消息代理，支持多节点集群部署，突破单机连接数限制。
- **移动端适配:** 优化前端响应式布局，开发 React Native 移动端应用。