黑白棋游戏项目

项目简介

这是一个基于Java实现的棋类游戏集合，包括黑白棋（Reversi）、五子棋（Gomoku）和和平棋（Peace）三种游戏模式。项目采用面向对象的设计方法，实现了不同棋类游戏的规则和逻辑。设计遵循了面向对象的继承和多态原则，通过抽象基类实现代码复用，提高了代码的可维护性和可扩展性。游戏支持两名玩家交替落子，并根据不同游戏规则判定胜负。

项目结构

项目由以下关键类组成，形成了一个完整的棋类游戏框架：

- `Game.java`: 抽象基类，定义了所有棋类游戏的共同特性和行为。作为所有具体游戏的父类，提供了游戏的基本框架。
- `Board.java`: 棋盘类，实现8×8的棋盘数据结构和基本操作，是所有游戏共用的棋盘实现。
- `Piece.java`: 棋子枚举类，定义了黑棋、白棋和空位三种状态，并提供了棋子的可视化表示。
- `Player.java`: 玩家类，包含玩家名称和所使用的棋子类型，表示参与游戏的用户。
- `GameManager.java`: 游戏管理器，负责管理不同游戏模式的切换和创建，是用户与游戏交互的主要接口。
- `ReversiGame.java`: 黑白棋（翻转棋）游戏实现，继承自Game类，实现了黑白棋特有的规则逻辑。
- `GomokuGame.java`: 五子棋游戏实现，继承自Game类，实现了五子棋特有的规则逻辑。
- `PeaceGame.java`: 和平棋游戏实现，继承自Game类，实现了和平棋特有的规则逻辑。

运行截图

```
  A B C D E F G H    游戏信息              游戏列表
  ---------------    ----------------     ----------------
1 ● · · · · · · ·    游戏编号: 1          游戏列表:
2 · · · · · · · ·    游戏类型: peace      1. peace
3 · · · · · · · ·    玩家1: ●             2. reversi
4 · · · ○ ● · · ·    玩家2: ○ ←           3. Gomoku
5 · · · ● ○ · · ·
6 · · · · · · · ·
7 · · · · · · · ·
8 · · · · · · · ·

命令: [坐标] - 落子, 数字 - 切换游戏, peace/reversi/gomoku - 添加新游戏, pass - 跳过, quit - 退出
请输入命令: 3
已切换到游戏3
按回车键继续...
```

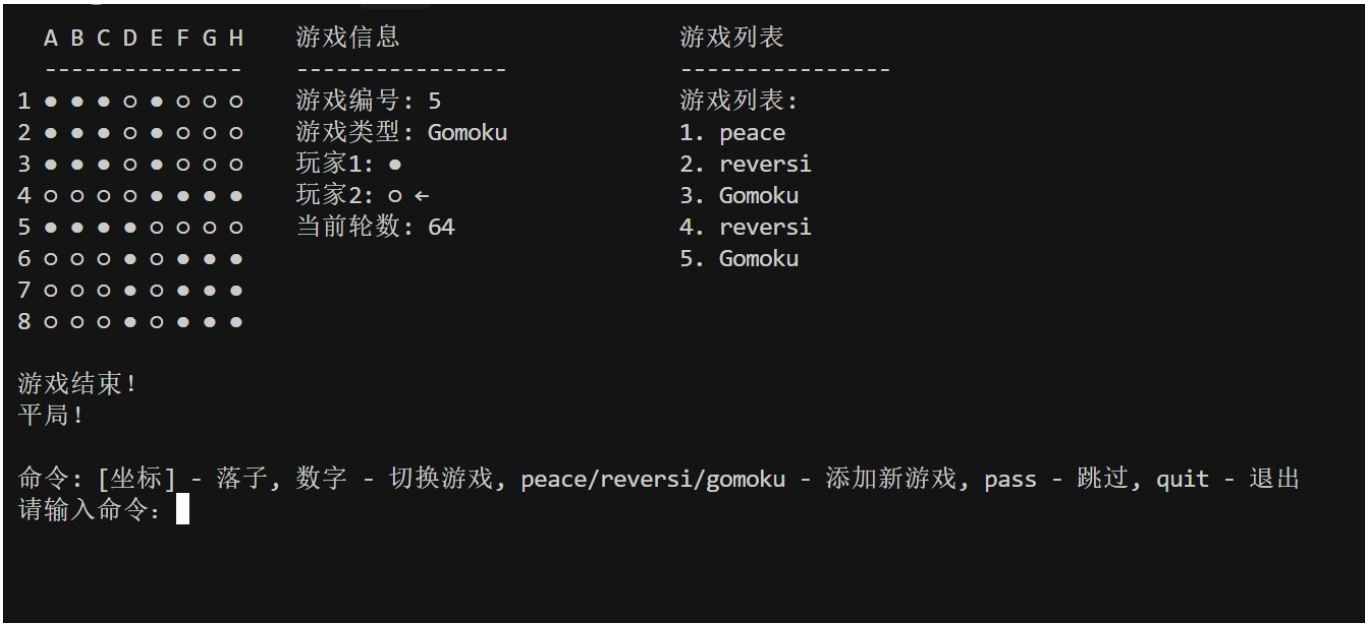
```
  A B C D E F G H    游戏信息              游戏列表
  ---------------    ----------------     ----------------
1 · · · · · · · ·    游戏编号: 3          游戏列表:
2 · · · · · · · ·    游戏类型: Gomoku     1. peace
3 · · · · · · · ·    玩家1: ●             2. reversi
4 · · ● · · · · ·    玩家2: ○ ←           3. Gomoku
5 · · · · · · · ·    当前轮数: 1
6 · · · · · · · ·
7 · · · · · · · ·
8 · · · · · · · ·

命令: [坐标] - 落子, 数字 - 切换游戏, peace/reversi/gomoku - 添加新游戏, pass - 跳过, quit - 退出
请输入命令: 4d
```

```
  A B C D E F G H    游戏信息              游戏列表
  ---------------    ----------------     ----------------
1 · · · · · · · ·    游戏编号: 3          游戏列表:
2 · · · · · · · ·    游戏类型: Gomoku     1. peace
3 · · ● ○ · · · ·    玩家1: ● ←           2. reversi
4 · · ● ○ · · · ·    玩家2: ○             3. Gomoku
5 · · ● ○ · · · ·    当前轮数: 11
6 · · ● ○ · · · ·
7 · · ● ● · · · ·
8 · · ○ · · · · ·

游戏结束!
玩家1 获胜!

命令: [坐标] - 落子, 数字 - 切换游戏, peace/reversi/gomoku - 添加新游戏, pass - 跳过, quit - 退出
请输入命令:
```



游戏模式

黑白棋 (Reversi)

- **游戏规则**：在8×8的棋盘上，两名玩家轮流放置自己颜色的棋子。黑方先行。
- **初始布局**：游戏开始时，棋盘中央的四个格子已经放置了两黑两白的棋子，呈对角线分布。具体位置是：(3,3)白、(3,4)黑、(4,3)黑、(4,4)白。
- **落子规则**：
  - 每次落子必须在直线上（横向、纵向或对角线）夹住对方至少一个棋子。
  - 落子后，所有被夹住的对方棋子都会被翻转为己方颜色。
  - 如果没有合法的落子位置，玩家必须选择"Pass"，让对方继续落子。
- **胜利条件**：当棋盘填满或双方都无法落子时，游戏结束，拥有更多棋子的玩家获胜。
- **特殊规则**：如果连续两次Pass（双方都无法落子），游戏立即结束。

五子棋 (Gomoku)

- **游戏规则**：在8×8的棋盘上，两名玩家轮流放置自己颜色的棋子。黑方先行。
- **落子规则**：
  - 每次只能在空位上落子。

    - 每次落子后不会导致任何已有棋子的翻转。
    - 棋子一旦放下，就不能移动或移除。
- **胜利条件**：先形成五个连续的同色棋子（横向、纵向或对角线）的玩家立即获胜。
- **其他规则**：
    - 如果棋盘下满仍然没有玩家达成连五，则游戏以平局结束。
    - 不存在"Pass"操作，玩家每轮必须落子。

和平棋 (Peace)

- **游戏规则**：基于黑白棋的变种，但不翻转对方棋子。这是一种更简单的规则变体。
- **初始设置**：与黑白棋相同，中间四格有特定的棋子排列：(3,3)白、(3,4)黑、(4,3)黑、(4,4)白。
- **落子规则**：
    - 玩家可以在任何空位上落子，不需要夹住对方棋子。
    - 落子后不会翻转任何棋子。
    - 不存在"Pass"操作，只要有空位就必须落子。
- **胜利条件**：当棋盘填满时，拥有更多棋子的玩家获胜。

技术特点

- **面向对象设计**：

    - 采用继承机制实现代码复用，所有游戏共享基类的通用功能。
    - 利用多态实现不同游戏规则的特定行为。
    - 通过抽象方法定义游戏必须实现的核心功能。

- **封装**：

    - 每个类都有明确的职责边界和封装的内部实现。
    - 提供清晰的公共接口，隐藏实现细节。
    - 通过访问修饰符控制属性和方法的可见性。

- **代码复用**：

    - 将共同特性抽象到基类中，避免代码重复。
    - 通用算法（如棋盘操作）集中在专门的类中实现。
    - 特定游戏只需要实现自己的独特规则。

- **游戏逻辑**：

    - 实现了复杂的黑白棋翻转规则和胜负判定。
    - 使用数组表示方向向量，简化了多方向检查的代码。
    - 利用坐标系和二维数组高效存储和操作棋盘状态。

代码详解

Game.java（抽象基类）

抽象基类，定义了所有游戏共有的属性和方法：

- **核心属性**：

  - `gameId`：游戏唯一标识符，用于区分不同游戏实例。
  - `gameType`：游戏类型（如"reversi"、"gomoku"、"peace"），标识游戏模式。
  - `board`：棋盘对象，管理棋盘状态。
  - `player1`/`player2`：两名玩家对象。
  - `currentPlayer`：当前行动的玩家。
  - `gameOver`：游戏是否结束的标志。

- **抽象方法**：

  - `placePiece(int row, int col)`：在指定位置放置棋子，返回是否成功放置。由子类实现具体的落子规则。
  - `isGameOver()`：判断游戏是否结束，根据具体游戏规则判定。
  - `getValidMoves()`：获取当前玩家的所有合法落子位置，返回坐标列表。

- **共用方法**：

  - `switchPlayer()`：切换当前玩家，从player1切换到player2，或从player2切换到player1。
  - `getGameId()/getGameType()`：获取游戏ID和类型。
  - `getBoard()`：获取棋盘对象。
  - `getCurrentPlayer()`：获取当前玩家。
  - `isOver()/setGameOver()`：获取/设置游戏状态。
  - `getPlayer1()/getPlayer2()`：获取玩家对象。

Board.java（棋盘类）

棋盘类，实现8×8的棋盘及其基本操作：

- **核心属性**：

  - `SIZE`：棋盘大小常量，固定为8×8。
  - `board[][]`：二维数组表示的棋盘，存储每个位置的棋子状态（黑/白/空）。

- **主要方法**：

  - `initReversiBoard()`：初始化黑白棋棋盘，在中心四格设置初始棋子。
  - `initPeaceBoard()`：初始化和平棋棋盘，与黑白棋初始布局相同。
  - `setPiece(int row, int col, Piece piece)`：在指定位置设置指定类型的棋子。
  - `getPiece(int row, int col)`：获取指定位置的棋子类型。
  - `isInBoard(int row, int col)`：判断坐标是否在棋盘范围内（0-7）。
  - `isEmpty(int row, int col)`：判断指定位置是否为空（没有棋子）。
  - `countPieces(Piece piece)`：统计棋盘上特定颜色棋子的数量，用于计分。
  - `isFull()`：判断棋盘是否已满（没有空位），是某些游戏模式的结束条件。

Piece.java（棋子枚举类）

棋子枚举类，定义了三种棋子状态：

- **枚举值**：

  - `BLACK("●")`：黑棋，用实心圆点表示。
  - `WHITE("○")`：白棋，用空心圆表示。

    - `EMPTY("·")`：空位，用小点表示。

- **属性**：
    - `symbol`：棋子的显示符号，用于棋盘的可视化显示。
- **方法**：
    - `getSymbol()`：获取棋子的显示符号。

Player.java（玩家类）

玩家类，存储玩家信息：

- **属性**：
    - `name`：玩家名称，如"玩家1"、"玩家2"。
    - `piece`：玩家使用的棋子类型（BLACK或WHITE）。
- **方法**：
    - `getName()/setName()`：获取/设置玩家名称。
    - `getPiece()/setPiece()`：获取/设置玩家的棋子类型。

GameManager.java（游戏管理器）

游戏管理器，负责游戏的创建和切换：

- **核心属性**：
    - `games`：游戏列表，存储所有创建的游戏实例。
    - `currentGameIndex`：当前活动游戏的索引。
- **主要方法**：
    - `getCurrentGame()`：获取当前活动的游戏实例。
    - `getAllGames()`：获取所有游戏实例的列表。
    - `switchGame(int gameId)`：根据游戏ID切换当前活动游戏。
    - `switchGame(String gameType)`：根据游戏类型名称切换游戏。
    - `addNewGame(String gameType)`：创建指定类型的新游戏并添加到游戏列表。
    - `placePiece(int row, int col)`：在当前游戏中放置棋子。
    - `pass()`：在黑白棋游戏中执行"Pass"操作，跳过当前玩家的回合。

ReversiGame.java（黑白棋实现）

黑白棋游戏实现，继承自Game类：

- **核心属性**：
    - `lastMoveWasPass`：标记上一步是否是Pass操作。
    - `DIRECTIONS`：方向数组，定义了8个方向的坐标偏移，用于检查合法落子和翻转棋子。
- **主要方法**：

    - `placePiece(int row, int col)`:
        - 检查落子是否合法。
        - 放置棋子并翻转被夹住的对手棋子。
        - 检查游戏是否结束。
        - 切换玩家并检查下一玩家是否有合法落子。
    - `pass()`:
        - 当玩家没有合法落子位置时执行跳过。
        - 检查连续Pass条件和游戏结束条件。
    - `flipPieces(int row, int col)`:
        - 执行落子后的棋子翻转操作。
        - 沿8个方向检查并翻转被夹住的对手棋子。
    - `isValidMove(int row, int col, Piece piece)`:
        - 检查指定位置是否是有效的落子点。
        - 判断是否能夹住并翻转对手棋子。
    - `isGameOver()`:
        - 检查游戏是否结束（棋盘满或双方都无法落子）。
    - `getValidMoves()`:
        - 计算当前玩家所有合法的落子位置。
    - `getScore(Player player)`:
        - 计算指定玩家的得分（棋子数量）。
    - `getWinner()`:
        - 根据双方棋子数量判断胜者。

GomokuGame.java（五子棋实现）

五子棋游戏实现，继承自Game类：

- **主要方法**:
    - `placePiece(int row, int col)`:
        - 检查落子位置是否为空。
        - 放置棋子。
        - 检查是否形成五连子获胜。
        - 切换玩家。
    - `checkWin(int row, int col)`:
        - 检查从指定位置出发，是否在任一方向上有五个连续同色棋子。

        - 检查横向、纵向和两个对角线方向。
    - `isValidMove(int row, int col, Piece piece)`:
        - 判断是否可以在指定位置落子（只需检查位置是否为空）。
    - `getValidMoves()`:
        - 返回所有空位作为有效落子点。
    - `isGameOver()`:
        - 检查是否有玩家获胜或棋盘已满。

PeaceGame.java（和平棋实现）

和平棋游戏实现，继承自Game类：

- **主要方法**:
    - `placePiece(int row, int col)`:
        - 检查落子位置是否为空。
        - 简单放置棋子，不翻转对手棋子。
        - 检查游戏是否结束（棋盘已满）。
        - 切换玩家。
    - `isValidMove(int row, int col, Piece piece)`:
        - 判断指定位置是否可落子（只需要是空位即可）。
    - `getValidMoves()`:
        - 返回所有空位作为有效落子点。
    - `isGameOver()`:
        - 检查棋盘是否已满。
    - `getScore(Player player)`:
        - 计算指定玩家的得分（棋子数量）。
    - `getWinner()`:
        - 根据双方棋子数量判断胜者。

如何使用

1. **编译代码**:

```
javac *.java
```

2. **创建主类**： 创建一个包含main方法的类，实例化GameManager并使用它来管理游戏。例如：

```
public class Main {
    public static void main(String[] args) {
        GameManager gameManager = new GameManager();

        // 获取当前游戏（默认是和平棋）
        Game currentGame = gameManager.getCurrentGame();

        // 切换到黑白棋
        gameManager.switchGame("reversi");

        // 放置棋子示例
        gameManager.placePiece(2, 3);

        // 在黑白棋中执行Pass操作
        gameManager.pass();

        // 添加新游戏
        gameManager.addNewGame("gomoku");
    }
}
```

3. **游戏交互**： 根据需要，可以实现图形用户界面或命令行界面，显示棋盘和提供用户交互功能：

    - 显示当前玩家
    - 显示棋盘状态
    - 接收用户输入的落子位置
    - 显示游戏结果

扩展性

该项目设计支持轻松添加新的棋类游戏，只需要遵循以下步骤：

1. **创建新游戏类**：

```
public class NewGame extends Game {
    public NewGame(int gameId) {
        super(gameId, "newgame");
        // 初始化特定于新游戏的属性
    }

    // 实现抽象方法
    @Override
    public boolean placePiece(int row, int col) {
        // 实现新游戏的落子逻辑
    }

    @Override
    public boolean isGameOver() {
```

```
        // 实现新游戏的结束条件检查
    }

    @Override
    public List<int[]> getValidMoves() {
        // 实现新游戏的合法落子计算
    }

    // 添加特定于新游戏的方法
}
```

2. **在GameManager中添加支持**:

```
// 在switchGame方法中添加
else if (gameType.equalsIgnoreCase("newgame")) {
    currentGameIndex = /* 索引 */;
    return true;
}

// 在addNewGame方法中添加
else if (gameType.equalsIgnoreCase("newgame")) {
    games.add(new NewGame(newGameId));
}
```

3. **如果需要，扩展Board类**:

```
// 在Board类中添加新的初始化方法
public void initNewGameBoard() {
    // 初始化新游戏的棋盘状态
}
```

通过这种方式，可以在不修改现有代码的前提下，轻松添加新的游戏模式，如围棋、国际象棋等。