

## 实战 1.1 命令行带 AI 的黑白棋游戏

### 一、实验目的

1. 复习并使用的 TypeScript 语言
2. 复习 TypeScript 中的数组、类等知识点
3. 为学习 ArkTS 版本的黑白棋做算法准备

### 二、实验原理

1. Typescript
2. 二维数组的应用

### 三、实验仪器材料

1. 计算机实训室电脑一台
2. VS Code
3. Node.js、npm

### 四、游戏逻辑

黑白棋游戏逻辑的核心在于一个严谨的规则验证与执行系统。游戏基于 8 行 8 列的棋盘，每个格子有三种状态：空、黑棋或白棋。玩家双方轮流在棋盘上落子，每一步都必须符合特定的游戏规则。

当玩家选择落子位置时，系统会启动验证流程。关键规则是必须形成有效的“夹击”态势，即在至少一个直线方向（包括水平、垂直或对角线）上，能够将连续的对方棋子夹在新落的棋子与已有的己方棋子之间。系统会从落子点向八个方向扫描，检查每个方向是否满足“对方棋子-连续对方棋子-己方棋子”的模式。只有满足此条件的方向才被视为有效，该方向上的对方棋子会被标记为待翻转。

如果没有任何方向满足夹击条件，走法则被视为无效。反之，只要有一个方向符合要求，走法即被认可。确认有效后，系统会执行落子操作：先在目标位置放置己方棋子，然后翻转所有被标记的对方棋子。这个过程会立即改变棋盘上的势力分布，是游戏的核心策略。

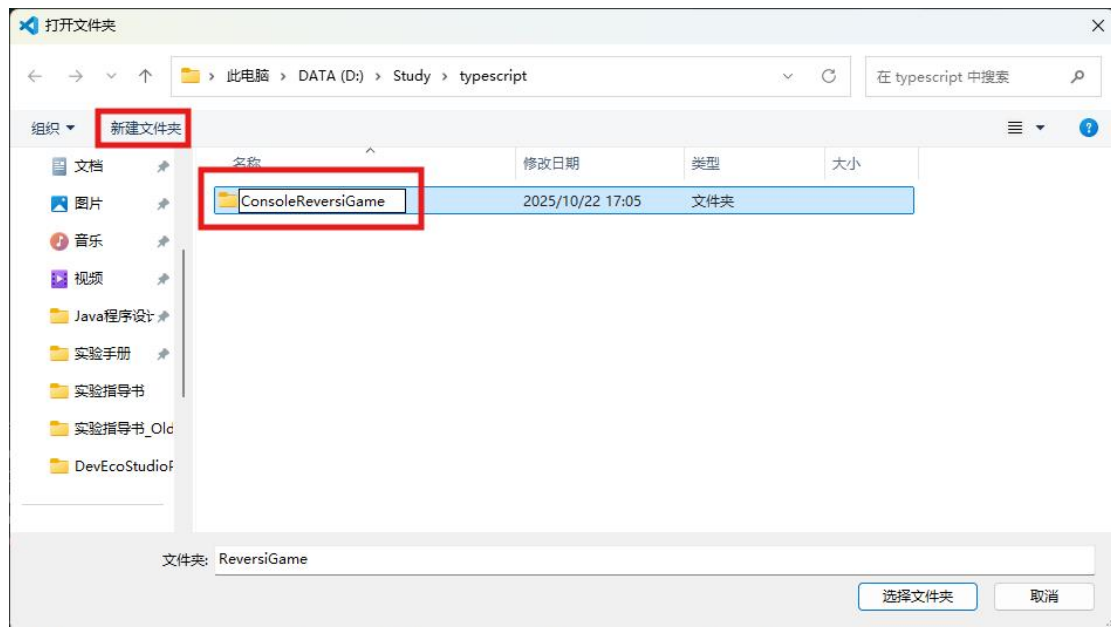
游戏支持双方轮流对弈，若一方无有效位置可下则轮空。为实现人机对战，系统内置了 AI 逻辑模块。AI 会扫描整个棋盘找出所有合法落子点，然后从中随机选择作为决策。这种基础策略确保了游戏的自动运行，同时为后续开发更复杂的 AI 算法奠定了基础。

整个游戏逻辑通过精密的坐标计算、方向检测和状态管理，构建了一个完整且自洽的规则引擎，为玩家提供了符合标准的黑白棋对战体验。

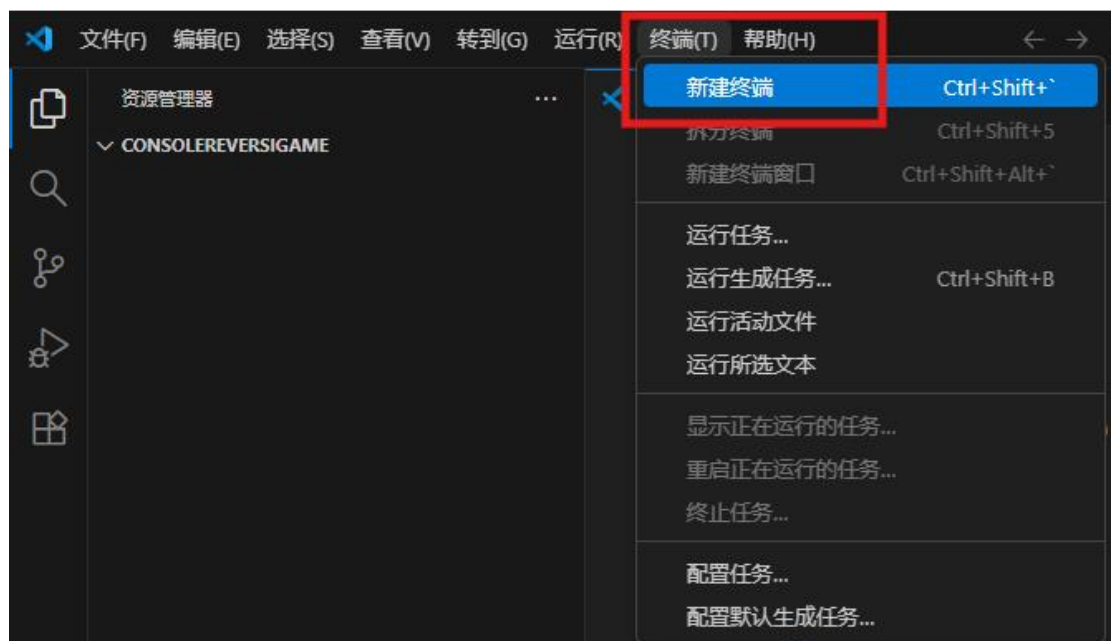
## 五、实验步骤

使用 VS Code 软件

打开 VS Code 软件，在自己电脑的任意目录创建一个游戏源代码文件夹，文件夹的名字是“ConsoleReversiGame”。



1. 在 VS Code 打开命令行终端，安装一些 TypeScript 必备的依赖包。



1) 在 VSCode 的 Terminal 中，执行命令安装 typescript:

```
npm install -g typescript
```

```
PS D:\Study\typescript\ConsoleReversiGame>
PS D:\Study\typescript\ConsoleReversiGame>
PS D:\Study\typescript\ConsoleReversiGame> npm install -g typescript

changed 1 package in 4s
PS D:\Study\typescript\ConsoleReversiGame> []
```

## 2) 安装 ts-node

和 TypeScript 编译器一样，通过 npm 全局安装 ts-node：

```
npm install -g ts-node
```

```
PS D:\Study\typescript\ConsoleReversiGame>
PS D:\Study\typescript\ConsoleReversiGame> npm install -g ts-node

changed 20 packages in 4s
PS D:\Study\typescript\ConsoleReversiGame> []
```

## 3) 在目录中新建一个文件 tsconfig.json，内容加入：

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es2016",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  },
  "ts-node": {
    "esm": false
  }
}
```

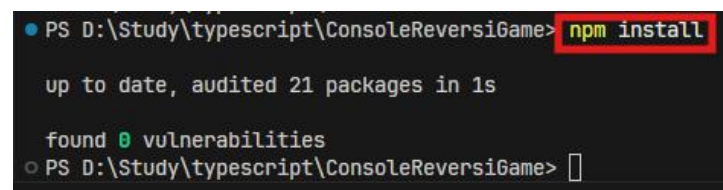
## 4) 新建一个配置 package.json 文件，内容加入：

```
{
  "name": "typescript",
  "version": "1.0.0",
  "description": "",

  "scripts": {
    "build": "tsc",
    "start": "ts-node Game.ts",
    "dev": "node --loader ts-node/esm Game.ts",
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

```
},
"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
  "ts-node": "^10.9.2",
  "typescript": "^5.9.2"
}
}
```

完成配置后，执行命令 `npm install` 初始化项目的依赖：



```
PS D:\Study\typescript\ConsoleReversiGame> npm install
up to date, audited 21 packages in 1s

found 0 vulnerabilities
PS D:\Study\typescript\ConsoleReversiGame>
```

至此，我们的项目就准备完毕，可以开始编写游戏代码了。

## 2. 开始编写游戏

1) 新建游戏的主程序文件 `Game.ts`，内容如下：

```
async function gameLoop() {
  console.log("黑白棋游戏-命令行版")
}

// 启动游戏
gameLoop();
```

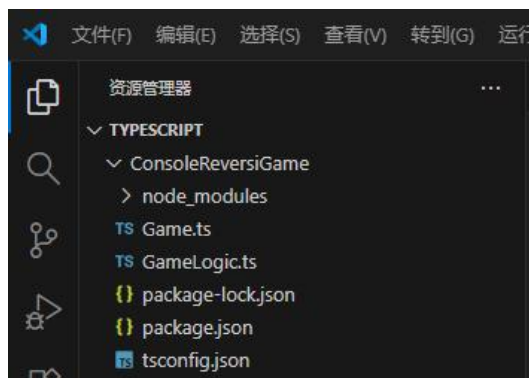
这是一个非常简单的空白程序，在终端执行命令，启动这个初始的空白主程序，命令为：

```
ts-node Game.ts
PS D:\Study\typescript\ConsoleReversiGame> ts-node Game.ts
黑白棋游戏-命令行版
PS D:\Study\typescript\ConsoleReversiGame>
```

这个命令启动并执行 `Game.ts`，打印一段欢迎消息。出现这个界面，证明整体的环境搭建完成且运行正常。

2) 新建游戏的算法逻辑文件 `GameLogic.ts`，此时我们项目的所有文件都准备完毕了，文件

夹中所有的文件截图如下：



我们将在 `GameLogic.ts` 中编写可以复用的游戏逻辑, 在 `Game.ts` 中编写调用代码和界面。这种游戏逻辑和界面分离的设计, 符合面向对象设计原理以及 MVC 模型, 让整体设计更清晰。

3) 在 `GameLogic.ts` 添加代表棋子的枚举数据类型, 并且将这个数据类型导出便于主程序使用, 代码如下:

```
// 定义棋盘上格子的三种状态: E (Empty) - 空, B (Black) - 黑棋, W (White) - 白棋
export enum CType {
    E, // 空
    B, // 黑棋
    W // 白棋
}
```

4) 在 `GameLogic.ts` 添加代表棋子的枚举数据类型 `CType`, 并且将这个枚举数据类型导出便于主程序使用, 代码如下:

```
// 定义棋盘上格子的三种状态: E (Empty) - 空, B (Black) - 黑棋, W (White) - 白棋
export enum CType {
    E, // 空
    B, // 黑棋
    W // 白棋
}
```

5) 在 `GameLogic.ts` 添加代表棋子落子位置的类 `Coordinate`, 并且将 `Coordinate` 这个数据类型导出便于主程序使用, 注意阅读分析类的定义代码, 如下:

```
// 根据 ArkTS 的严格模式要求, 使用 class 来定义坐标
export class Coordinate {
    row: number; // 行
    col: number; // 列
}
```

```

    constructor(row: number, col: number) {
        this.row = row;
        this.col = col;
    }
}

```

6) 在 `GameLogic.ts` 添加落子到棋盘某个坐标后的结果的数据类型 `MoveCheckResult`，注意看这也是一个类，注意它有两个成员变量，了解各自的作用是什么，代码如下：

```

// 使用 class 来定义 每一步落子后的结果
export class MoveCheckResult {
    isValid: boolean; // 标记当前走法是否有效
    piecesToFlip: Coordinate[]; // 记录如果当前走法有效，可以翻转的对方棋子坐标列表

    constructor(isValid: boolean, piecesToFlip: Coordinate[]) {
        this.isValid = isValid;
        this.piecesToFlip = piecesToFlip;
    }
}

```

7) 接着创建游戏的核心逻辑类 `GameLogic`，同样也要导出这个类。这个类首先定义了 8 个方向，在查找翻转棋子的时候按照这 8 个方向查找。接着定义了一个傻瓜 AI 的落子函数 `findRandomMove` 用于在棋盘上随机落子，这个函数会调用 `findAllValidMoves` 函数找到所有可以落子的有效位置 (`Coordinate`)，然后随机选择其中一个位置 (`Coordinate`) 落子。查找所有可以落子的位置则需要调用 `checkValidMove` 遍历检查棋盘上每个位置，记录下可以落子的所有可能性，最后可以调用 `applyMove` 落子到棋盘中。具体代码如下：

```

// 游戏核心逻辑类
export class GameLogic {
    // 定义 8 个方向的坐标偏移量，用于检查棋子周围的情况
    private directions: Coordinate[] = [
        new Coordinate(-1, 0), new Coordinate(1, 0), // 上、下
        new Coordinate(0, -1), new Coordinate(0, 1), // 左、右
        new Coordinate(-1, -1), new Coordinate(-1, 1), // 左上、右上 (对角线)
        new Coordinate(1, -1), new Coordinate(1, 1) // 左下、右下 (对角线)
    ];

    /**
     * AI 逻辑：从所有有效落子位置中随机选择一个
     * @param player - AI 玩家的棋子类型 (B 或 W)
     */
}

```

```

    * @param board - 当前的 8x8 棋盘状态
    * @returns 返回一个随机选择的有效落子坐标, 如果没有有效走法则返回 null
    */
    public findRandomMove(player: CType, board: CType[][]): Coordinate | null {
        const validMoves = this.findAllValidMoves(player, board);
        if (validMoves.length === 0) {
            return null; // 没有有效走法
        }
        const randomIndex = Math.floor(Math.random() * validMoves.length);
        return validMoves[randomIndex];
    }

    /**
     * 找出指定玩家所有有效的落子位置
     * @param player - 当前玩家的棋子类型 (B 或 W)
     * @param board - 当前的 8x8 棋盘状态
     * @returns 返回一个包含所有有效落子坐标的数组
     */
    public findAllValidMoves(player: CType, board: CType[][]): Coordinate[] {
        const validMoves: Coordinate[] = [];
        for (let row = 0; row < 8; row++) {
            for (let col = 0; col < 8; col++) {
                if (board[row][col] === CType.E) {
                    const result = this.checkValidMove(row, col, player, board);
                    if (result.isValid) {
                        validMoves.push(new Coordinate(row, col));
                    }
                }
            }
        }
        return validMoves;
    }

    /**
     * 检查在给定的坐标落子是否为一步有效的走法
     * @param row - 落子的行号
     * @param col - 落子的列号
     * @param player - 当前玩家的棋子类型 (B 或 W)
     * @param board - 当前的 8x8 棋盘状态
     * @returns MoveCheckResult - 包含走法是否有效以及可翻转棋子列表的对象
     */
    public checkValidMove(row: number, col: number, player: CType, board: CType[][]):
    MoveCheckResult {
        // 规则 1: 落子的位置必须是空的

```

```

    if (board[row][col] !== CType.E) {
        return new MoveCheckResult(false, []); // 如果非空，直接返回无效
    }

    // 确定对手的棋子类型
    const opponent: CType = player === CType.B ? CType.W : CType.B;
    // 用于存储所有可以被翻转的棋子
    let allPiecesToFlip: Coordinate[] = [];

    // 规则 2：遍历 8 个方向，检查是否能形成“夹击”
    for (const dir of this.directions) {
        // 存储当前方向上可能被翻转的棋子
        let piecesInThisLine: Coordinate[] = [];
        // 从落子点的邻近点开始沿当前方向检查
        let currentRow = row + dir.row;
        let currentCol = col + dir.col;

        // 只要还在棋盘内，并且遇到的是对手的棋子，就继续沿该方向前进
        while (currentRow >= 0 && currentRow < 8 && currentCol >= 0 && currentCol < 8
&&
            board[currentRow][currentCol] === opponent) {
            // 将这个对手棋子的坐标记录下来
            piecesInThisLine.push(new Coordinate(currentRow, currentCol));
            currentRow += dir.row;
            currentCol += dir.col;
        }

        // 循环结束后，如果仍在棋盘内，并且遇到了自己的棋子，说明形成了有效的“夹击”
        if (currentRow >= 0 && currentRow < 8 && currentCol >= 0 && currentCol < 8 &&
            board[currentRow][currentCol] === player) {
            // 只有在中间有对手棋子时，这个方向的走法才算有效
            if (piecesInThisLine.length > 0) {
                // 将这个方向上所有可翻转的棋子添加到总列表中
                allPiecesToFlip.push(...piecesInThisLine);
            }
        }
    }

    // 规则 3：如果总的可翻转棋子列表不为空，说明这是一个有效的走法
    if (allPiecesToFlip.length > 0) {
        return new MoveCheckResult(true, allPiecesToFlip);
    }

```



```

        // 如果遍历完所有方向都没有可翻转的棋子，则为无效走法
        return new MoveCheckResult(false, []);
    }

    /**
     * 将一步棋应用到棋盘上，并翻转相应的棋子
     * @param board - 当前的 8x8 棋盘状态
     * @param move - 要执行的落子坐标
     * @param piecesToFlip - 需要翻转的棋子坐标列表
     * @param player - 当前玩家的棋子类型 (B 或 W)
     * @returns 返回更新后的棋盘状态
     */
    public applyMove(board: CType[][], move: Coordinate, piecesToFlip: Coordinate[], player:
    CType): CType[][] {
        // 创建棋盘的深拷贝以避免直接修改原棋盘
        const newBoard = board.map(row => [...row]);
        // 在指定位置落子
        newBoard[move.row][move.col] = player;
        // 翻转所有“夹击”的棋子
        for (const piece of piecesToFlip) {
            newBoard[piece.row][piece.col] = player;
        }
        return newBoard;
    }
}

```

8) 有了基本的傻瓜 AI 逻辑代码后，接着需要在主程序 Game.ts 中完成如下功能：

- a. 绘制界面棋盘
- b. 提醒玩家用户输入落子坐标
- c. 检查落子是否有效并在棋盘上落子
- d. 绘制更新后的棋盘
- e. 换手到 AI 玩家落子
- f. 回到步骤 b，如果游戏结束则退出

9) 在 Game.ts 使用一个 8\*8 的二维数组 board 来保存棋盘上的棋子，数据类型为：let board: CType[][] = [[]]，我们循环使用 console.log 来打印这个棋盘中的棋子。所以在 Game.ts 中导入 GameLogic.ts 中定义的 CType 类型，然后创建一个打印棋盘的函数 printBoard 来打印棋盘。在主逻辑循环中，初始化棋盘并打印棋盘。

完整的 Game.ts 的代码如下：

```

import { CType } from './GameLogic';

/**
 * 在控制台打印当前棋盘状态
 */
function printBoard(board: CType[][]) {
  console.log('\n   0 1 2 3 4 5 6 7');
  console.log(' +-----+');
  for (let i = 0; i < 8; i++) {
    let rowStr = `${i}|`;
    for (let j = 0; j < 8; j++) {
      switch (board[i][j]) {
        case CType.B: rowStr += '   '; break; // 黑棋 (Nerd Font: nf-fa-circle)
        case CType.W: rowStr += '   '; break; // 白棋 (Nerd Font: nf-fa-circle_o)
        default: rowStr += ' • '; break; // 空
      }
    }
    console.log(rowStr + ' |');
  }
  console.log(' +-----+');
}

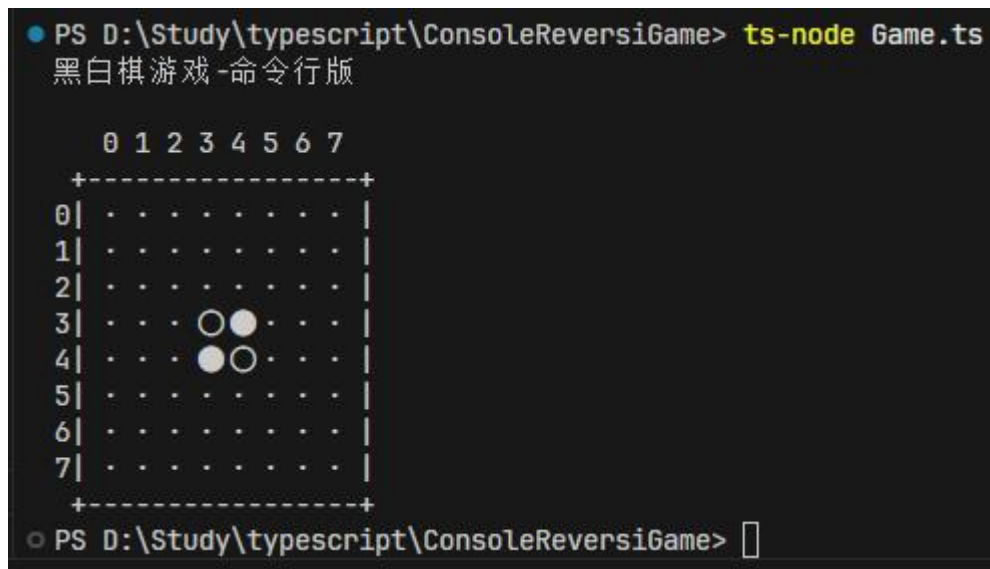
async function gameLoop() {
  console.log("黑白棋游戏-命令行版")
  let board: CType[][] = [
    [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
    [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
    [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
    [CType.E, CType.E, CType.E, CType.W, CType.B, CType.E, CType.E, CType.E],
    [CType.E, CType.E, CType.E, CType.B, CType.W, CType.E, CType.E, CType.E],
    [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
    [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
    [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
  ];

  printBoard(board);
}

// 启动游戏
gameLoop();

```

执行 Game.ts 可以看到如下界面，显示了一个初始的棋盘：



```
PS D:\Study\typescript\ConsoleReversiGame> ts-node Game.ts
黑白棋游戏-命令行版

  0 1 2 3 4 5 6 7
+-----+
0| . . . . . . . |
1| . . . . . . . |
2| . . . . . . . |
3| . . . ○ ● . . |
4| . . . ● ○ . . |
5| . . . . . . . |
6| . . . . . . . |
7| . . . . . . . |
+-----+

PS D:\Study\typescript\ConsoleReversiGame> 
```

10) 目前 Game.ts 还只能打印一个棋盘，后续要添加游戏的主循环、玩家落子、换手、AI 落子的功能。因此我们需要导入 GameLogic 中的游戏逻辑、坐标，并且导入 readline 函数用于读取命令行的输入，新增的代码如下，注意不要删除之前录入的 printBoard 函数。

```
import { CType, GameLogic, Coordinate } from './GameLogic';
import * as readline from 'readline';

// 创建 readline 接口用于接收用户输入
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

// 创建游戏逻辑实例
const game = new GameLogic();

/**
 * 提示玩家输入并返回坐标
 */
function promptMove(player: CType): Promise<Coordinate> {
  const playerName = player === CType.B ? '黑棋' : '白棋';
  const playerIcon = player === CType.B ? '●' : '○';
  return new Promise((resolve) => {
    rl.question(`轮到 【${playerIcon} ${playerName}】 走，请输入坐标 (格式: 行,列): `,
      (input) => {
        const parts = input.split(',');
        if (parts.length !== 2) {
```

```

        console.log('输入格式错误，请重新输入！');
        return resolve(promptMove(player));
    }
    const row = parseInt(parts[0], 10);
    const col = parseInt(parts[1], 10);

    if (isNaN(row) || isNaN(col) || row < 0 || row > 7 || col < 0 || col > 7) {
        console.log('坐标超出范围，请重新输入！');
        return resolve(promptMove(player));
    }
    resolve(new Coordinate(row, col));
});
}

```

接着修改游戏的主循环函数，在主循环中新建表示当前玩家的变量 `currentPlayer`（初始为黑棋），并且用一个 `while(true)` 循环完成：1. 绘制棋盘，2. 让玩家和 AI 轮番落子，3. 切换玩家，4. 检查游戏是否结束，5. 如果游戏结束则计分判输赢并退出。修改后的 `gameLoop` 函数体如下：

```

/**
 * 游戏主循环
 */
async function gameLoop() {
    console.log("黑白棋游戏-命令行版")
    let board: CType[][] = [
        [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
        [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
        [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
        [CType.E, CType.E, CType.E, CType.W, CType.B, CType.E, CType.E, CType.E],
        [CType.E, CType.E, CType.E, CType.B, CType.W, CType.E, CType.E, CType.E],
        [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
        [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
        [CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E, CType.E],
    ];

    let currentPlayer = CType.B; // 黑棋（玩家）先走

    while (true) {
        printBoard(board);

        const playerValidMoves = game.findAllValidMoves(currentPlayer, board);
        const opponent: CType = currentPlayer === CType.B ? CType.W : CType.B;
    }
}

```

```

const opponentValidMoves = game.findAllValidMoves(opponent, board);

// 游戏结束条件：双方都无棋可走
if (playerValidMoves.length === 0 && opponentValidMoves.length === 0) {
  console.log('游戏结束！双方均无有效走法。');
  break;
}

// 如果当前玩家无棋可走，则轮到对方
if (playerValidMoves.length === 0) {
  const playerName = currentPlayer === CType.B ? '黑棋' : '白棋';
  const playerIcon = currentPlayer === CType.B ? '⬛' : '⬜';
  console.log(`【${playerIcon} ${playerName}】无有效走法，跳过此回合。`);
  currentPlayer = opponent;
  continue;
}

let move: Coordinate | null;

if (currentPlayer === CType.B) { // 玩家回合
  while (true) {
    move = await promptMove(currentPlayer);
    const moveResult = game.checkValidMove(move.row, move.col, currentPlayer, board);
    if (moveResult.isValid) {
      board = game.applyMove(board, move, moveResult.piecesToFlip, currentPlayer);
      break;
    } else {
      console.log('无效的走法，请重新选择位置！');
    }
  }
} else { // AI 回合
  console.log(`轮到【⬜ 白棋】(AI)走...`);
  move = game.findRandomMove(currentPlayer, board);
  if (move) {
    console.log(`AI 选择走在 (${move.row}, ${move.col})`);
    const moveResult = game.checkValidMove(move.row, move.col, currentPlayer, board);
    board = game.applyMove(board, move, moveResult.piecesToFlip, currentPlayer);
  }
}

// 切换玩家
currentPlayer = opponent;
}

```

```
// 游戏结束后计分
let blackCount = 0;
let whiteCount = 0;
for (const row of board) {
  for (const cell of row) {
    if (cell === CType.B) blackCount++;
    if (cell === CType.W) whiteCount++;
  }
}

console.log(`\n--- 最终得分 ---`);
console.log(`  黑棋: ${blackCount}`);
console.log(`  白棋: ${whiteCount}`);
if (blackCount > whiteCount) {
  console.log('恭喜，你赢了！');
} else if (whiteCount > blackCount) {
  console.log('很遗憾，AI 赢了。');
} else {
  console.log('平局！');
}

rl.close();
}
```

执行程序，可以开始在终端界面进行游戏：



11) 当前的 AI 是一个很傻的随机落子 AI，可玩性比较差。我们可以添加一个带落子位置评

分的贪心算法，通过替换 AI 的落子函数让 AI 具备更强的棋力。在 GameLogic 类中添加贪心算法，贪心算法的落子函数代码如下：

```
// 8x8 棋盘上每个位置的战略价值分数
private static readonly positionalValueMatrix = [
    [120, -20, 20, 5, 5, 20, -20, 120], // 角
    [-20, -40, -5, -5, -5, -5, -40, -20], // 边的旁边
    [20, -5, 15, 3, 3, 15, -5, 20],
    [5, -5, 3, 3, 3, 3, -5, 5],
    [5, -5, 3, 3, 3, 3, -5, 5],
    [20, -5, 15, 3, 3, 15, -5, 20],
    [-20, -40, -5, -5, -5, -5, -40, -20],
    [120, -20, 20, 5, 5, 20, -20, 120]
];

/**
 * AI 贪心算法：基于位置价值选择最佳走法
 * @param player - AI 玩家的棋子类型 (B 或 W)
 * @param board - 当前的 8x8 棋盘状态
 * @returns 返回价值最高的有效落子坐标，如果没有有效走法则返回 null
 */
public findGreedyMove(player: CType, board: CType[][]): Coordinate | null {
    const validMoves = this.findAllValidMoves(player, board);
    if (validMoves.length === 0) {
        return null;
    }

    let bestScore = -Infinity;
    let bestMoves: Coordinate[] = [];

    for (const move of validMoves) {
        const score = GameLogic.positionalValueMatrix[move.row][move.col];
        if (score > bestScore) {
            bestScore = score;
            bestMoves = [move];
        } else if (score === bestScore) {
            bestMoves.push(move);
        }
    }

    // 如果有多个价值相同的最佳走法，从中随机选择一个，让 AI 行为不那么固定
    const randomIndex = Math.floor(Math.random() * bestMoves.length);
    return bestMoves[randomIndex];
}
```

```
}
```

请同学们自行替换 AI 的落子函数，并修改游戏的标题为“AI 黑白棋”，让这个游戏更完善。

## 六、实验注意事项

1. 注意教师的操作演示。
2. 学生机与教师机内网连通，能接收和提交实验结果。
3. 按实验要求输入测试数据，并查看输出结果是否符合实验结果。

## 七、思考题

1. 如何将命令行的游戏，移植到鸿蒙 ArkUI 界面下。
2. 利用二维数组还可以制作什么样的游戏？