前后端接口清单

本系统共包含 **5 个 Servlet**，提供 **13 个 API 接口**，以及 **6 个 JSP 页面**。

一、用户认证模块（UserServlet）

基础路径：`/user`

接口路径	请求方法	参数	功能说明	返回格式
`/user?action=register`	POST	`username`, `password`	玩家注册	HTML (JavaScript 弹窗)
`/user?action=login`	POST	`username`, `password`, `role`	玩家/管理员登录	HTML (重定向或弹窗)

1.1 玩家注册接口（`/user?action=register`）

1.1.1 前端提交阶段

用户在 `index.jsp` 页面的注册表单中填写用户名和密码，点击"注册账号"按钮后，浏览器会构造一个 POST 请求：

```
<form action="user?action=register" method="post">
    <input type="text" name="username" placeholder="设置用户名" required>
    <input type="password" name="password" placeholder="设置密码" required>
    <button type="submit">注册账号</button>
</form>
```

当表单提交时，浏览器会将表单数据编码为 `application/x-www-form-urlencoded` 格式（例如：`username=test&password=123456`），并通过 HTTP POST 请求发送到服务器地址 `user?action=register`。

1.1.2 后端接收与处理

Tomcat 服务器接收到请求后，会根据 `@WebServlet("/user")` 注解找到 `UserServlet` 类，并调用其 `doPost` 方法。Servlet 首先从请求中提取 `action` 参数：

```
protected void doPost(HttpServletRequest request, HttpServletResponse
response) {
    request.setCharacterEncoding("UTF-8");  // 防止中文乱码
    response.setContentType("text/html;charset=UTF-8");

    String action = request.getParameter("action");  // 获取 "register"

    if ("register".equals(action)) {
        handleRegister(request, response);  // 调用注册处理方法
    }
}
```

1.1.3 数据库操作与响应

`handleRegister` 方法的核心逻辑分为三步：

第一步：检查用户名是否已存在

```
String u = request.getParameter("username");
String p = request.getParameter("password");

try (Connection conn = DBUtil.getConnection()) {
    // 使用 PreparedStatement 防止 SQL 注入
    String checkSql = "SELECT username FROM Player WHERE username = ?";
    PreparedStatement checkPs = conn.prepareStatement(checkSql);
    checkPs.setString(1, u);
    ResultSet rs = checkPs.executeQuery();

    if (rs.next()) {
        // 用户名已存在，返回错误提示
        response.getWriter().write("<script>alert('注册失败：用户名已存在');
window.location='index.jsp';</script>");
        return;
```

```
    }
```

第二步：插入新用户到数据库

如果用户名不存在，则执行插入操作。新用户的 `level` 字段默认设置为 1：

```
    // 插入新用户，默认等级为 1
    String sql = "INSERT INTO Player (username, password, level) VALUES (?,
?, 1)";
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, u);
    ps.setString(2, p);
    ps.executeUpdate();
```

第三步：返回成功响应

注册成功后，服务器返回一段 JavaScript 代码，该代码会在浏览器端执行，显示成功提示并刷新页面：

```
    response.getWriter().write("<script>alert('注册成功！请登录');
window.location='index.jsp';</script>");
}
```

关键点说明：

- 使用 `try-with-resources` 语法（`try (Connection conn = ...)`）确保数据库连接自动关闭。
- 使用 `PreparedStatement` 而非字符串拼接 SQL，有效防止 SQL 注入攻击。
- 返回的 HTML 中包含 JavaScript，浏览器执行后会显示弹窗并跳转页面。

1.2 玩家/管理员登录接口（`/user?action=login`）

1.2.1 前端表单提交

登录表单包含用户名、密码和一个 `role` 单选框（用于区分玩家和管理员身份）：

```
<form action="user?action=login" method="post">
    <input type="text" name="username" placeholder="请输入用户名" required>
    <input type="password" name="password" placeholder="请输入密码" required>
    <input type="radio" name="role" value="player" checked> 玩家
    <input type="radio" name="role" value="admin"> 管理员
    <button type="submit">立即登录</button>
</form>
```

1.2.2 后端路由分发

`doPost` 方法根据 `action` 参数调用 `handleLogin` 方法，该方法首先提取三个参数：

```
private void handleLogin(HttpServletRequest request, HttpServletResponse
response) {
    String u = request.getParameter("username");
    String p = request.getParameter("password");
    String role = request.getParameter("role");  // "player" 或 "admin"
```

1.2.3 管理员登录分支

如果 `role` 参数为 `"admin"`，系统执行硬编码的管理员验证逻辑（管理员账号固定为 `admin/admin`）：

```
    if ("admin".equals(role)) {
        if ("admin".equals(u) && "admin".equals(p)) {
            HttpSession session = request.getSession();  // 获取或创建 Session
            session.setAttribute("isAdmin", true);       // 标记管理员身份
            session.setAttribute("currentUser", "Administrator");
            response.sendRedirect("admin.jsp");          // 跳转到管理后台
        } else {
            // 账号密码错误，返回提示
            response.getWriter().write("<script>alert('管理员账号或密码错误！');
window.location='index.jsp';</script>");
        }
        return;
    }
```

Session 机制说明：`request.getSession()` 会检查请求中是否携带 `JSESSIONID` Cookie。如果有，则返回对应的 Session 对象；如果没有，则创建一个新的 Session，并将 `JSESSIONID` 写入响应的 Cookie 中。Session 存储在服务器内存中，可以跨多个请求保持用户登录状态。

1.2.4 玩家登录分支

对于玩家登录，系统首先检查玩家是否试图使用 `admin` 用户名登录（防止玩家账号和管理员账号冲突）：

```
    // 严禁玩家使用 admin 账号登录
    if ("admin".equals(u)) {
        response.getWriter().write("<script>alert('该账号为管理员账号，请选择【管理员】身份登录'); window.location='index.jsp';</script>");
        return;
    }
```

然后，系统连接数据库，查询 `Player` 表验证账号密码：

```
    try (Connection conn = DBUtil.getConnection()) {
        String sql = "SELECT * FROM Player WHERE username = ? AND password = ?";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, u);
        ps.setString(2, p);
        ResultSet rs = ps.executeQuery();

        if (rs.next()) {
            // 登录成功：将用户信息存入 Session
            HttpSession session = request.getSession();
            session.setAttribute("currentUser", u);
            session.setAttribute("currentLevel", rs.getInt("level"));

            // 跳转到玩家大厅
            response.sendRedirect("player.jsp");
        } else {
            // 登录失败：账号或密码错误
```

```
            response.getWriter().write("<script>alert('登录失败：账号或密码错
误'); window.location='index.jsp';</script>");
        }
    }
```

关键点说明：

- `session.setAttribute("currentUser", u)` 将用户名存入 Session，后续页面（如 `player.jsp`、`game.jsp`）可通过 `session.getAttribute("currentUser")` 获取当前登录用户。
- `response.sendRedirect("player.jsp")` 发送 302 重定向响应，浏览器会自动请求 `player.jsp` 页面。
- 如果查询结果集为空（`rs.next()` 返回 `false`），说明账号或密码错误。

二、游戏对战模块（GameServlet）

基础路径：`/play`

接口路径	请求方法	参数	功能说明	返回格式
`/play?action=start&level=?`	POST	`level` (0/1/2)	开始新游戏	JSON: `{"status":"started", "gameId":1}`
`/play?action=move&x=?&y=?`	POST	`x`, `y` (落子坐标)	玩家落子，AI 响应	JSON: `{"ai_x":7, "ai_y":8, "winner":0, "oldLevel":10, "newLevel":15}`

返回字段说明：

- `winner`: 0=继续, 1=玩家赢, 2=AI赢, 3=平局
- `oldLevel`, `newLevel`: 仅在游戏结束时返回，用于显示等级变化

2.1 开始新游戏接口（`/play?action=start&level=?`）

2.1.1 前端触发流程

用户在 `game.jsp` 页面选择难度（0=简单，1=中级，2=困难），点击"开始新游戏"按钮。前端 JavaScript 执行 `startGame()` 函数：

```
function startGame() {
    let level = document.getElementById("difficultyLevel").value;  // 获取难度值
    isGameActive = true;  // 标记游戏已激活
    isThinking = false;
    drawBoard();  // 清空 Canvas 画布

    // 通过 fetch API 发送异步 POST 请求
    fetch('play?action=start&level=' + level, { method: 'POST' })
        .then(res => res.json())  // 将响应解析为 JSON
        .then(data => {
            document.getElementById("status").innerText = "游戏开始，你是黑棋，请落子";
        });
}
```

`fetch` API 会构造一个 HTTP POST 请求，URL 参数中包含 `action=start` 和用户选择的 `level` 值（如 `level=1`）。

2.1.2 后端初始化游戏状态

`GameServlet` 的 `doPost` 方法接收到请求后，首先检查 Session 中是否已有棋盘状态：

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) {
    request.setCharacterEncoding("UTF-8");
    response.setContentType("application/json;charset=UTF-8");

    String action = request.getParameter("action");  // 获取 "start"
    HttpSession session = request.getSession();
```

```
    int[][] board = (int[][]) session.getAttribute("board");

    // 如果 board 为空，或 action 是 "start"，则初始化新游戏
    if (board == null || "start".equals(action)) {
        board = new int[BOARD_SIZE][BOARD_SIZE];  // 创建 15×15 的二维数组
        session.setAttribute("board", board);      // 存入 Session
        session.setAttribute("stepCount", 0);      // 初始化步数为 0

        String levelStr = request.getParameter("level");
        int difficulty = (levelStr != null) ? Integer.parseInt(levelStr) :
1;
        session.setAttribute("difficulty", difficulty);  // 保存难度到 Session
```

关键点说明：`board` 是一个 `int[][]` 数组，用于在服务器端存储棋盘状态。数组元素的值：0 表示空位，1 表示玩家黑子，2 表示 AI 白子。将这个数组存入 Session，可以确保同一用户的多轮落子请求都能访问到同一个棋盘状态。

2.1.3 创建数据库记录

接下来，系统需要为该局游戏在数据库中创建一条记录。`createNewGame` 方法执行以下操作：

```
    String currentUser = (String) session.getAttribute("currentUser");
    if (currentUser == null) currentUser = "Unknown";

    int newGameId = createNewGame(currentUser);  // 创建 Game 表记录
    session.setAttribute("currentGameId", newGameId);

    response.getWriter().write("{\"status\":\"started\", \"gameId\":" +
newGameId + "}");
}
```

`createNewGame` 方法的具体实现：

```
private int createNewGame(String username) {
    int nextGameId = 1;
    try (Connection conn = DBUtil.getConnection()) {
        // 查询该玩家当前最大的 game_count
```

```
        String query = "SELECT MAX(game_count) FROM Game WHERE username =
?";
        PreparedStatement psQuery = conn.prepareStatement(query);
        psQuery.setString(1, username);
        ResultSet rs = psQuery.executeQuery();
        if (rs.next()) {
            nextGameId = rs.getInt(1) + 1;  // 最大值 + 1 作为新局号
        }
        // 插入新游戏记录，is_win=0 表示进行中
        String insert = "INSERT INTO Game (username, game_count, is_win,
game_time) VALUES (?, ?, 0, NOW())";
        PreparedStatement psInsert = conn.prepareStatement(insert);
        psInsert.setString(1, username);
        psInsert.setInt(2, nextGameId);
        psInsert.executeUpdate();
    } catch (Exception e) { e.printStackTrace(); }
    return nextGameId;
}
```

系统返回 JSON 响应 `{"status":"started", "gameId":1}`，前端收到后更新页面状态提示。

2.2 玩家落子接口（`/play?action=move&x=?&y=?`）

2.2.1 前端坐标计算与请求发送

用户点击 Canvas 画布时，浏览器触发 `canvas.onclick` 事件处理器。JavaScript 需要将鼠标点击的像素坐标转换为棋盘的行列坐标：

```
canvas.onclick = function(e) {
    if (!isGameActive || isThinking) return;  // 检查游戏状态和是否正在思考

    let rect = canvas.getBoundingClientRect();  // 获取 Canvas 相对于视口的位置
    let x = e.clientX - rect.left;  // 鼠标点击的 X 坐标（相对于 Canvas）
    let y = e.clientY - rect.top;   // 鼠标点击的 Y 坐标（相对于 Canvas）

    // 计算最近的交叉点坐标（网格大小为 30 像素，起始偏移 15 像素）
    let col = Math.round((x - 15) / gridSize);  // 列坐标（0-14）
```

```
    let row = Math.round((y - 15) / gridSize);  // 行坐标（0-14）

    // 边界检查
    if (row < 0 || row > 14 || col < 0 || col > 14) return;

    // 立即在 Canvas 上绘制玩家的棋子（优化用户体验）
    drawPiece(row, col, 1);
    isThinking = true;  // 锁定，防止重复点击
    document.getElementById("status").innerText = "AI 思考中...";

    // 发送落子请求到服务器
    fetch('play?action=move&x=' + row + '&y=' + col, { method: 'POST' })
        .then(res => res.json())
        .then(data => {
            // 处理服务器响应（见下文）
        });
}
```

2.2.2 后端处理玩家落子

服务器接收到落子请求后，首先从 Session 中恢复游戏状态：

```
    if ("move".equals(action)) {
        int x = Integer.parseInt(request.getParameter("x"));  // 行坐标
        int y = Integer.parseInt(request.getParameter("y"));  // 列坐标

        // 从 Session 中恢复游戏状态
        int[][] board = (int[][]) session.getAttribute("board");
        Integer stepCountObj = (Integer) session.getAttribute("stepCount");
        int stepCount = (stepCountObj == null) ? 0 : stepCountObj;
        Integer difficultyObj = (Integer)
session.getAttribute("difficulty");
        int difficulty = (difficultyObj == null) ? 1 : difficultyObj;
        Integer gameIdObj = (Integer) session.getAttribute("currentGameId");
        int gameId = (gameIdObj == null) ? 1 : gameIdObj;
```

```
        String currentUser = (String) session.getAttribute("currentUser");
        if (currentUser == null) currentUser = "Unknown";
```

关键点：所有游戏状态（棋盘、步数、难度、游戏ID）都存储在 Session 中，这样无需每次请求都传递整个棋盘数据，大大减少了网络传输量。

2.2.3 更新棋盘并保存步数

后端检查该位置是否已被占用，然后更新棋盘数组并保存到数据库：

```
        if (board[x][y] != 0) return;  // 位置已被占用，直接返回（前端已绘制，这里做二次校验）
        board[x][y] = 1;  // 标记为玩家棋子
        stepCount++;

        // 保存玩家这一步到 Step 表
        saveStepToDB(currentUser, gameId, stepCount, 1, x + "," + y);
```

`saveStepToDB` 方法将每一步落子记录插入数据库：

```
private void saveStepToDB(String user, int gameId, int step, int who, String coord) {
    try (Connection conn = DBUtil.getConnection();
         PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO Step (username, game_count, step_count, by_who, coordination) VALUES (?,?,?,?,?)")) {
        ps.setString(1, user);
        ps.setInt(2, gameId);
        ps.setInt(3, step);
        ps.setInt(4, who);  // 1=玩家, 2=AI
        ps.setString(5, coord);  // "行,列" 格式，如 "7,7"
        ps.executeUpdate();
    } catch (Exception e) { e.printStackTrace(); }
}
```

2.2.4 胜负判断与 AI 响应

系统调用 `checkWin` 方法判断玩家是否获胜。`checkWin` 方法从落子位置向四个方向（横、竖、两个斜线）检查是否有五连珠：

```
        if (checkWin(board, x, y, 1)) {
            // 玩家获胜
            updateGameResult(currentUser, gameId, 1);  // 更新 Game 表的
is_win=1
            int[] lvInfo = updatePlayerLevel(currentUser, difficulty, true);
 // 计算新等级
            session.removeAttribute("board");  // 清除棋盘状态，结束游戏

            String json = String.format(
                "{\"ai_x\":-1, \"ai_y\":-1, \"winner\":1, \"oldLevel\":%d,
\"newLevel\":%d}",
                lvInfo[0], lvInfo[1]);
            response.getWriter().write(json);
            return;
        }
```

如果玩家未获胜，系统调用 AI 算法生成落子位置：

```
        GobangAI ai = new GobangAI();
        ai.setDifficulty(difficulty);  // 设置 AI 难度（影响算法策略）
        Point aiMove = ai.think(board);  // AI 思考，返回坐标 Point(x, y)

        if (aiMove.x != -1) {
            board[aiMove.x][aiMove.y] = 2;  // 更新棋盘，标记为 AI 棋子
            stepCount++;
            saveStepToDB(currentUser, gameId, stepCount, 2, aiMove.x + "," +
aiMove.y);

            // 判断 AI 是否获胜
            if (checkWin(board, aiMove.x, aiMove.y, 2)) {
                updateGameResult(currentUser, gameId, 2);
```

```
                int[] lvInfo = updatePlayerLevel(currentUser, difficulty,
false);
                session.removeAttribute("board");

                String json = String.format(
                    "{\"ai_x\":%d, \"ai_y\":%d, \"winner\":2,
\"oldLevel\":%d, \"newLevel\":%d}",
                    aiMove.x, aiMove.y, lvInfo[0], lvInfo[1]);
                response.getWriter().write(json);
                return;
            }
        } else {
            // AI 无法落子（棋盘已满），平局
            response.getWriter().write("{\"ai_x\":-1, \"ai_y\":-1,
\"winner\":3}");
            return;
        }

        // 游戏继续，返回 AI 落子坐标
        session.setAttribute("stepCount", stepCount);  // 更新 Session 中的步
数
        String json = String.format("{\"ai_x\":%d, \"ai_y\":%d,
\"winner\":0}", aiMove.x, aiMove.y);
        response.getWriter().write(json);
    }
```

2.2.5 前端渲染与结算弹窗

前端收到响应后，根据 `winner` 字段的值执行不同的逻辑：

```
    fetch('play?action=move&x=' + row + '&y=' + col, { method: 'POST' })
        .then(res => res.json())
        .then(data => {
            const handleResponse = () => {
                // 绘制 AI 落子（如果有）
                if (data.ai_x !== -1) {
```

```
                    drawPiece(data.ai_x, data.ai_y, 2);
                }

                // 判断游戏是否结束
                if (data.winner === 1 || data.winner === 2) {
                    isGameActive = false;
                    // 显示结算弹窗，展示等级变化和头衔变化
                    showResult(data.winner === 1, data.oldLevel,
data.newLevel);
                } else if (data.winner === 3) {
                    alert("平局！");
                    isGameActive = false;
                } else {
                    // 游戏继续，解锁点击
                    document.getElementById("status").innerText = "轮到你了";
                    isThinking = false;
                }
            };

            // 延迟 500ms 显示 AI 落子，模拟思考过程
            if (data.ai_x !== -1) {
                setTimeout(handleResponse, 500);
            } else {
                handleResponse();
            }
        });
```

`showResult` 函数会弹出 Modal 弹窗，显示"恭喜胜利"或"遗憾落败"，并展示等级从 `oldLevel` 变化到 `newLevel`，以及头衔是否发生变化。

关键技术点总结：

- **Session 存储棋盘状态**：避免每次传输 15×15 的数组，显著提升性能。
- **异步 AJAX 通信**：使用 `fetch` API 实现无刷新交互。
- **Canvas 实时绘制**：玩家落子后立即绘制，无需等待服务器响应，优化用户体验。

- **AI 算法集成**：`GobangAI.think()` 根据难度返回不同的落子策略。

三、对局复盘模块（ReviewServlet）

基础路径：`/review`

接口路径	请求方法	参数	功能说明	返回格式
`/review`	GET	无	加载复盘页面（转发到 review.jsp）	HTML (JSP 渲染)
`/review?action=getSteps&gameId=?`	GET	`gameId`	获取某局的所有步数记录	JSON 数组: `[{"step":1,"who":1,"x":7,"y":7},...]`

说明：

- `who`: 1=玩家, 2=AI
- 第一个接口通过 `request.getRequestDispatcher("review.jsp").forward()` 转发，不是纯 API

四、排行榜模块（RankServlet）

基础路径：`/rank`

接口路径	请求方法	参数	功能说明	返回格式
`/rank`	GET	无	加载排行榜页面（转发到 rank.jsp）	HTML (JSP 渲染)

说明：

- 该接口在服务器端查询所有玩家数据，计算排名和胜率，然后转发到 JSP 渲染
- 前端排序通过 JavaScript 在客户端完成，无需额外接口

五、管理员模块（AdminServlet）

基础路径：`/admin`

接口路径	请求方法	参数	功能说明	返回格式
`/admin?action=listPlayers`	POST	无	获取所有玩家列表	JSON 数组: `[{"username":"xxx", "password":"xxx", "level":11}, ...]`
`/admin?action=getGames&username=?`	POST	`username`	获取某玩家的所有对局	JSON 数组: `[{"game_count":1, "is_win":1, "time":"2024-01-01 12:00:00"}, ...]`
`/admin?action=getGameSteps&username=?&gameId=?`	POST	`username`, `gameId`	获取某局的所有步数	JSON 数组: `[{"step":1, "who":1, "x":7, "y":7}, ...]`
`/admin?action=addPlayer`	POST	`username`, `password`, `level`	新增玩家	JSON: `{"status":"ok"}`
`/admin?action=addGame`	POST	`username`, `isWin` (1/2)	为玩家新增对局记录	JSON: `{"status":"ok"}`

接口路径	请求方法	参数	功能说明	返回格式
`/admin?action=deletePlayer&username=?`	POST	`username`	删除玩家（级联删除所有数据）	JSON: `{"status":"ok"}`
`/admin?action=deleteGame&username=?&gameId=?`	POST	`username`, `gameId`	删除某局对局（后续局号自动前移）	JSON: `{"status":"ok"}`
`/admin?action=updatePlayer`	POST	`oldUsername`, `newUsername`, `newPassword`, `newLevel`	修改玩家信息（支持改用户名）	JSON: `{"status":"ok"}`

说明：

- 所有接口返回 JSON，便于前端 AJAX 处理
- 错误时返回 `{"error":"错误信息"}`，HTTP 状态码 500

六、JSP 页面（直接访问）

页面路径	功能说明	是否需要登录
`index.jsp`	登录注册首页	否
`player.jsp`	玩家大厅（功能导航）	是（玩家）

页面路径	功能说明	是否需要登录
`game.jsp`	游戏对战页面	是（玩家）
`review.jsp`	对局复盘页面	是（玩家，通过 `/review` 转发）
`rank.jsp`	排行榜页面	是（玩家，通过 `/rank` 转发）
`admin.jsp`	管理员后台	是（管理员）

说明：

- JSP 页面通过 `session.getAttribute("currentUser")` 检查登录状态
- 未登录会自动重定向到 `index.jsp`

接口统计汇总

- **API 接口总数**：13 个
  - 用户认证：2 个
  - 游戏对战：2 个
  - 对局复盘：2 个（1 个页面转发 + 1 个 API）
  - 排行榜：1 个（页面转发）
  - 管理员：8 个
- **JSP 页面总数**：6 个
- **请求方式分布**：
  - GET：3 个（页面加载）
  - POST：10 个（数据操作）

接口调用示例

1. 玩家注册

```
fetch('user?action=register', {
    method: 'POST',
    body: new URLSearchParams({username: 'test', password: '123'})
```

```
})
```

2. 开始游戏

```
fetch('play?action=start&level=1', { method: 'POST' })
    .then(res => res.json())
    .then(data => console.log(data.gameId));
```

3. 玩家落子

```
fetch('play?action=move&x=7&y=7', { method: 'POST' })
    .then(res => res.json())
    .then(data => {
        if(data.winner === 0) {
            drawPiece(data.ai_x, data.ai_y, 2); // 绘制AI落子
        }
    });
```

4. 管理员查询玩家列表

```
fetch('admin?action=listPlayers', { method: 'POST' })
    .then(res => res.json())
    .then(players => {
        players.forEach(p => console.log(p.username));
    });
```