**结构型模式重构修改说明文档**

**外观模式 (Facade) 重构**

1. **修改前的代码**

（请在此处插入修改前的代码截图）

2. **修改后的代码**

以下是Game.java修改后的内容，修改位置已经进行了标注

|  |
| --- |
| JavaScript package com.kob.backend.consumer;  import com.alibaba.fastjson.JSONObject; import com.kob.backend.pojo.Bot; import com.kob.backend.pojo.Record; import com.kob.backend.pojo.User; import lombok.Getter; import org.springframework.util.LinkedMultiValueMap; import org.springframework.util.MultiValueMap;  import java.util.\*; import java.util.concurrent.locks.ReentrantLock;  public class Game extends Thread {  private final Integer rows;  private final Integer cols;  private final Integer innerWallsCount;  @Getter  private final int[][] gameMap;  private final static int[] *dx* = {-1, 0, 1, 0};  private final static int[] *dy* = {0, 1, 0, -1};  @Getter  private final Player playerA, playerB;  private Integer nextStepA = null;  private Integer nextStepB = null;  private final ReentrantLock lock = new ReentrantLock();  private String status = "playing"; // playing finished  @Getter  private String loser = ""; // all, A, B   private final static String *addBotUrl* = "http://127.0.0.1:3002/bot/add/";   public Game(Integer rows, Integer cols, Integer innerWallsCount, Integer idA, Bot aBot, Integer idB, Bot bBot) {  this.rows = rows;  this.cols = cols;  this.innerWallsCount = innerWallsCount;  gameMap = new int[rows][cols];   Integer aBotId = -1, bBotId = -1;  String aBotCode = "", bBotCode = "";  if (aBot != null) {  aBotId = aBot.getId();  aBotCode = aBot.getCode();  }  if (bBot != null) {  bBotId = bBot.getId();  bBotCode = bBot.getCode();  }   playerA = new Player(idA, aBotId, aBotCode, rows - 2, 1, new ArrayList<>());  playerB = new Player(idB, bBotId, bBotCode, 1, cols - 2, new ArrayList<>());  }   // zzy修改：将地图生成与验证逻辑封装  public void createGameMap() {  for (int i = 0; i < 1000; i++) {  if (createWalls()) {  break;  }  }  }   public void setNextStepA(Integer nextStepA) {  lock.lock();  try{  this.nextStepA = nextStepA;  } finally {  lock.unlock();  }  }  public void setNextStepB(Integer nextStepB) {  lock.lock();  try{  this.nextStepB = nextStepB;  } finally {  lock.unlock();  }  }   private boolean createWalls() {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  gameMap[i][j] = 0;  }  }  for (int r = 0; r < rows; r++) {  gameMap[r][0] = gameMap[r][cols - 1] = 1;  }  for (int c = 0; c < cols; c++) {  gameMap[0][c] = gameMap[rows - 1][c] = 1;  }  Random rand = new Random();  for (int i = 0; i < innerWallsCount / 2; i++) {  for (int j = 0; j < 1000; j++) {  int r = rand.nextInt(rows);  int c = rand.nextInt(cols);  if (gameMap[r][c] == 1 || gameMap[rows - 1 - r][cols - 1 - c] == 1) {  continue;  }  if (r == rows - 2 && c == 1 || r == 1 && c == cols - 2) {  continue;  }  gameMap[r][c] = gameMap[rows - 1 - r][cols - 1 - c] = 1;  break;  }  }  return checkConnect(rows - 2, 1, 1, cols - 2);  }   private boolean checkConnect(int sx, int sy, int tx, int ty) {  if (sx == tx && sy == ty) return true;  gameMap[sx][sy] = 1;  for (int i = 0; i < 4; i++) {  int x = sx + *dx*[i], y = sy + *dy*[i];   if (checkConnect(x, y, tx, ty)) {  gameMap[sx][sy] = 0;  return true;  }  }  }  gameMap[sx][sy] = 0;  return false;  }   // zzy修改：发送消息逻辑移动至外观类调用  protected void sendAllMessage(String message) {  if (WebSocketServer.*users*.get(playerA.getId()) != null) {  WebSocketServer.*users*.get(playerA.getId()).sendMessage(message);  }  if (WebSocketServer.*users*.get(playerB.getId()) != null) {  WebSocketServer.*users*.get(playerB.getId()).sendMessage(message);  }  }   // zzy修改：玩家移动逻辑封装到外观类中  protected void sendMove() {  lock.lock();  try {  JSONObject resp = new JSONObject();  resp.put("event", "move");  resp.put("a\_move", nextStepA);  resp.put("b\_move", nextStepB);  nextStepA = null;  nextStepB = null;  sendAllMessage(resp.toJSONString());  } finally {  lock.unlock();  }  }   // zzy修改：判断比赛状态  protected void judge() {  List<Cell> cellsA = playerA.getCells();  List<Cell> cellsB = playerB.getCells();   boolean validA = checkValid(cellsA, cellsB);  boolean validB = checkValid(cellsB, cellsA);  if (!validA || !validB) {  status = "finished";  if (!validA && !validB) {  loser = "all";  } else if (!validA) {  loser = "A";  } else {  loser = "B";  }  }  }   private boolean checkValid(List<Cell> cellsA, List<Cell> cellsB) {  int n = cellsA.size();  Cell cell = cellsA.get(n - 1);  if (gameMap[cell.getX()][cell.getY()] == 1) {  return false;  }  for (int i = 0; i < n - 1; i++) {  if (cellsA.get(i).getX() == cell.getX() && cellsA.get(i).getY() == cell.getY()) {  return false;  }  }  for (int i = 0; i < n - 1; i++) {  if (cellsB.get(i).getX() == cell.getX() && cellsB.get(i).getY() == cell.getY()) {  return false;  }  }  return true;  } } |

以下是新增的GameFacade.java文件

|  |
| --- |
| JavaScript package com.kob.backend.consumer;  import com.alibaba.fastjson.JSONObject; import com.kob.backend.pojo.Bot; import com.kob.backend.pojo.Record; import com.kob.backend.pojo.User; import lombok.Getter; import org.springframework.util.LinkedMultiValueMap; import org.springframework.util.MultiValueMap;  import java.util.\*; import java.util.concurrent.locks.ReentrantLock;   public class GameFacade {  private final Game game;   // zzy修改：初始化GameFacade并绑定Game对象  public GameFacade(Game game) {  this.game = game;  }   // zzy修改：初始化游戏地图  public void initializeGame() {  game.createGameMap();  }   // zzy修改：开始游戏主线程  public void startGame() {  game.start();  }   // zzy修改：处理玩家下一步操作  public void handleNextStep(Integer stepA, Integer stepB) {  game.setNextStepA(stepA);  game.setNextStepB(stepB);  }   // zzy修改：发送比赛结果  public void sendResult() {  JSONObject result = new JSONObject();  result.put("event", "result");  result.put("loser", game.getLoser());  game.sendAllMessage(result.toJSONString());  }   // zzy修改：发送玩家移动信息  public void sendMove() {  game.sendMove();  }   // zzy修改：判断比赛状态  public void judge() {  game.judge();  } } |

3. **重构前后主要差异**

3.1 **重构前：**

1. **Game 类的职责过于繁重**：

* Game 类处理了所有的游戏逻辑，包括游戏地图的创建、玩家操作的处理、游戏状态的管理等。
* 这个类同时也负责与外部系统的交互，直接与 WebSocketServer, RecordMapper 和其他外部服务进行交互。
* 这种职责过多使得 Game 类变得非常庞大，难以维护和理解。游戏逻辑和外部接口被混合在一起，导致代码耦合度高。

1. **与外部系统的耦合**：

* Game 类直接操作外部系统，如 WebSocketServer 和 RecordMapper。例如，发送消息给玩家、更新数据库记录、查询和修改玩家评分等操作都直接写在了 Game 类中。
* 这种直接依赖外部系统的设计使得单元测试变得非常复杂，因为在测试时需要模拟或替代这些外部服务。

1. **缺乏高层次接口**：

* 外部调用需要直接与 Game 类中的多个方法交互，进行复杂的状态管理和操作。没有一个统一且简化的接口供外部使用，外部代码需要了解过多的实现细节，增加了调用的复杂性。

1. **难以扩展和维护**：

* 由于所有功能都集中在一个类中，扩展新的功能时可能会影响到其他功能，增加了修改和维护的难度。不同的开发人员在修改代码时，容易引入意外的错误。

3.2 **重构后：**

1. **引入了外观模式 (Facade)**：

* 在 Game 类之外引入了一个新的外观类 GameFacade，将 Game 类中的复杂操作封装起来，提供了简化的高层接口。
* GameFacade 提供了 setNextStepA(), setNextStepB(), getLoser(), startGame() 和 stopGame() 等方法，外部调用者通过这些方法与游戏进行交互，而无需关心内部的实现细节。

1. **职责明确的类结构**：

* Game 类的职责被集中在游戏核心逻辑上，包括游戏地图的创建、判断游戏是否结束等。
* 外部服务的交互（如与 WebSocket 服务器的通信、更新数据库记录）被移到了 GameFacade 类中，Game 类不再直接负责这些操作。这样，Game 类只专注于游戏内部的逻辑，符合单一职责原则（SRP）。

1. **降低了类之间的耦合度**：

* GameFacade 类起到了中介的作用，隔离了外部系统与 Game 类之间的直接依赖。例如，GameFacade 负责将游戏结果传递给 WebSocketServer，更新 RecordMapper 等操作，而 Game 类不再直接依赖这些外部系统。
* 通过这种方式，外部系统的变更不会影响 Game 类的实现，增强了代码的可维护性和扩展性。

1. **更清晰的接口设计**：

* 通过外观类，外部调用者现在只需要关注高层次的接口方法，而不需要关心游戏内部复杂的实现。比如，外部调用者只需通过 GameFacade 提供的接口来启动游戏、设置玩家操作步骤、查询游戏结果等，而不需要直接操作 Game 类中的复杂逻辑。

1. **便于单元测试**：

* 由于 Game 类不再直接依赖外部系统，测试 Game 类变得更加简单。在单元测试中，可以通过模拟 GameFacade 提供的高层接口进行测试，而不必担心复杂的外部依赖。
* 如果需要测试与外部系统（如数据库或 WebSocket）的交互，可以单独对 GameFacade 进行测试，而不需要涉及到 Game 类的核心逻辑。

1. **易于扩展**：

* 当需要修改外部交互方式时，只需要修改 GameFacade 类，而不必修改 Game 类的内部实现。这样，可以保持 Game 类的稳定性，降低系统修改的风险。
* 新的游戏规则或功能扩展也可以在 GameFacade 中进行，不会影响 Game 类的内部结构。

4. **UML 类图**

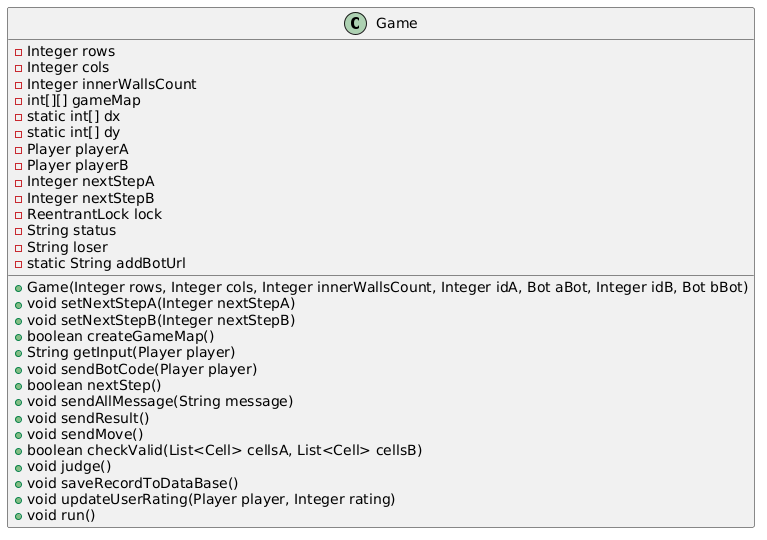


图4-1 重构前

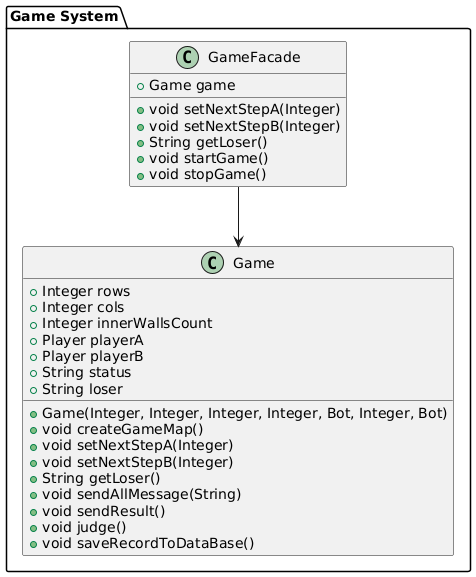


图4-2 重构后

5. **修改的原因**

5.1 **修改前的弊端：**

1. **高耦合度**：Game 类直接与 WebSocket 和数据库交互，这导致代码的维护和扩展变得困难。
2. **不清晰的职责分离**：Game 类负责管理游戏逻辑，同时处理与外部系统的交互，违反了单一职责原则（SRP）。
3. **难以测试**：由于 Game 类直接依赖外部系统，单元测试变得困难，测试时需要模拟多个外部服务。

5.2 **修改后的优点：**

1. **降低耦合度**：通过引入 GameFacade，Game 类与外部系统的交互被隐藏在外观类中，从而简化了接口，降低了类之间的耦合度。
2. **清晰的职责分离**：将游戏核心逻辑和外部依赖分离，使每个类的职责更加明确，符合单一职责原则（SRP）。
3. **易于扩展和测试**：通过外观模式，后续如果需要替换或修改外部系统（例如数据库或 WebSocket 服务器），不需要修改 Game 类，只需要调整 GameFacade 即可。

**装饰器模式 (Decorator)重构**

1. **修改前的代码**

|  |
| --- |
| JavaScript package com.kob.matchingsystem.utils;  import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Component; import org.springframework.util.LinkedMultiValueMap; import org.springframework.util.MultiValueMap; import org.springframework.web.client.RestTemplate;  import java.util.ArrayList; import java.util.List; import java.util.concurrent.locks.ReentrantLock;  @Component public class MatchingPool extends Thread{  private static List<Player> *players* = new ArrayList<>(); // 对这个列表加锁  private final ReentrantLock lock = new ReentrantLock();  private static RestTemplate *restTemplate*;  private final static String *startGameUrl*= "http://localhost:3000/pk/start/game/";   @Autowired  public void setRestTemplate (RestTemplate restTemplate){  MatchingPool.*restTemplate* = restTemplate;  };   public void addPlayer(Integer userId, Integer rating, Integer botId){  lock.lock();  try {  *players*.add(new Player(userId, rating, botId, 0));  } finally {  lock.unlock();  }  }  public void removePlayer(Integer userId){  lock.lock();  try {  List<Player> newPlayers = new ArrayList<>();  for(Player player : *players*){  if(!player.getUserId().equals(userId)){  newPlayers.add(player);  }  }  *players* = newPlayers;  }finally {  lock.unlock();  }  }   private void increasingWaitingTime(){ // 所有玩家等待时间加一  lock.lock();  try {  for(Player player : *players*){  player.setWaitingTime(player.getWaitingTime() + 1);  }  }finally {  lock.unlock();  }  }  private void matchPlayers(){ // 尝试匹配所有玩家  System.*out*.println(*players*);  lock.lock();  try {  boolean[] used = new boolean[*players*.size()];  for(int i = 0; i < *players*.size(); i++){  if(used[i])continue;  for(int j = i + 1; j < *players*.size(); j++){  if(used[j])continue;  Player a = *players*.get(i), b = *players*.get(j);  if(checkMatched(a, b)){  used[i] = used[j] = true;  sendResult(a, b);  break;  }  }  }  List<Player> newPlayers = new ArrayList<>();  for(int i = 0; i < *players*.size(); i++){  if(!used[i]){  newPlayers.add(*players*.get(i));  }  }  *players* = newPlayers;  } finally {  lock.unlock();  }  }  private boolean checkMatched(Player a, Player b){  int ratingDelta = Math.*abs*(a.getRating() - b.getRating());  int waitingTime = Math.*min*(a.getWaitingTime(), b.getWaitingTime());  return waitingTime \* 10 >= ratingDelta;  }  private void sendResult(Player a, Player b){ // 返回匹配结果  MultiValueMap<String, String> data = new LinkedMultiValueMap<>();  data.add("aId", a.getUserId().toString());  data.add("aBotId", a.getBotId().toString());  data.add("bId", b.getUserId().toString());  data.add("bBotId", b.getBotId().toString());  *restTemplate*.postForObject(*startGameUrl*, data, String.class);  }   @Override  public void run() {  while(true){  try {  Thread.*sleep*(1000);  increasingWaitingTime();  matchPlayers();  } catch (InterruptedException e) {  e.printStackTrace();  break;  }  }  } } |

2. **修改后的代码**

新是*BaseMatchingPool*.java内容

|  |
| --- |
| JavaScript package com.kob.matchingsystem.utils;  import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Component; import org.springframework.util.LinkedMultiValueMap; import org.springframework.util.MultiValueMap; import org.springframework.web.client.RestTemplate;  import java.util.ArrayList; import java.util.List; import java.util.concurrent.locks.ReentrantLock;  */\*\**  *\* zzy：BaseMatchingPool是原始的匹配池功能实现类，负责添加玩家、移除玩家、增加等待时间、*  *\* 尝试匹配玩家并发送匹配结果。*  *\*/* @Component public class BaseMatchingPool implements MatchingPoolDecorator {  private static List<Player> *players* = new ArrayList<>(); // zzy：存储所有玩家的列表  private final ReentrantLock lock = new ReentrantLock(); // zzy：用于线程安全的锁  private static RestTemplate *restTemplate*; // zzy：用于发送请求的RestTemplate实例  private final static String *startGameUrl* = "http://localhost:3000/pk/start/game/"; // zzy：游戏开始的URL   // zzy：自动注入RestTemplate  @Autowired  public void setRestTemplate(RestTemplate restTemplate) {  BaseMatchingPool.*restTemplate* = restTemplate;  }   // zzy：添加玩家到匹配池  @Override  public void addPlayer(Integer userId, Integer rating, Integer botId) {  lock.lock(); // zzy：获取锁  try {  *players*.add(new Player(userId, rating, botId, 0)); // zzy：将新玩家添加到列表  } finally {  lock.unlock(); // zzy：释放锁  }  }   // zzy：从匹配池中移除玩家  @Override  public void removePlayer(Integer userId) {  lock.lock(); // zzy：获取锁  try {  List<Player> newPlayers = new ArrayList<>();  for (Player player : *players*) {  if (!player.getUserId().equals(userId)) { // zzy：如果玩家ID不匹配，则保留  newPlayers.add(player);  }  }  *players* = newPlayers; // zzy：更新玩家列表  } finally {  lock.unlock(); // zzy：释放锁  }  }   // zzy：增加所有玩家的等待时间  @Override  public void increasingWaitingTime() {  lock.lock(); // zzy：获取锁  try {  for (Player player : *players*) {  player.setWaitingTime(player.getWaitingTime() + 1); // zzy：每个玩家等待时间加1  }  } finally {  lock.unlock(); // zzy：释放锁  }  }   // zzy：尝试匹配所有玩家  @Override  public void matchPlayers() {  lock.lock(); // zzy：获取锁  try {  boolean[] used = new boolean[*players*.size()]; // zzy：用于标记已经匹配的玩家  for (int i = 0; i < *players*.size(); i++) {  if (used[i]) continue; // zzy：如果当前玩家已匹配，跳过  for (int j = i + 1; j < *players*.size(); j++) {  if (used[j]) continue; // zzy：如果对方玩家已匹配，跳过  Player a = *players*.get(i), b = *players*.get(j); // zzy：获取待匹配玩家  if (checkMatched(a, b)) { // zzy：检查是否匹配  used[i] = used[j] = true; // zzy：标记玩家已匹配  sendResult(a, b); // zzy：发送匹配结果  break; // zzy：匹配成功，跳出内层循环  }  }  }  // zzy：移除所有已匹配的玩家  List<Player> newPlayers = new ArrayList<>();  for (int i = 0; i < *players*.size(); i++) {  if (!used[i]) {  newPlayers.add(*players*.get(i));  }  }  *players* = newPlayers; // zzy：更新玩家列表  } finally {  lock.unlock(); // zzy：释放锁  }  }   // zzy：判断两个玩家是否匹配  private boolean checkMatched(Player a, Player b) {  int ratingDelta = Math.*abs*(a.getRating() - b.getRating()); // zzy：计算玩家评分差距  int waitingTime = Math.*min*(a.getWaitingTime(), b.getWaitingTime()); // zzy：选择最小的等待时间  return waitingTime \* 10 >= ratingDelta; // zzy：如果等待时间足够，认为玩家匹配  }   // zzy：发送匹配结果  private void sendResult(Player a, Player b) {  MultiValueMap<String, String> data = new LinkedMultiValueMap<>();  data.add("aId", a.getUserId().toString());  data.add("aBotId", a.getBotId().toString());  data.add("bId", b.getUserId().toString());  data.add("bBotId", b.getBotId().toString());  *restTemplate*.postForObject(*startGameUrl*, data, String.class); // zzy：发送匹配结果  }   // zzy：启动匹配池的执行  @Override  public void run() {  while (true) {  try {  Thread.*sleep*(1000); // zzy：每秒钟执行一次  increasingWaitingTime(); // zzy：增加所有玩家的等待时间  matchPlayers(); // zzy：尝试匹配玩家  } catch (InterruptedException e) {  e.printStackTrace();  break; // zzy：如果线程被中断，退出循环  }  }  } } |

新增MatchingPoolDecorator.java文件

|  |
| --- |
| JavaScript package com.kob.matchingsystem.utils;  */\*\**  *\* zzy：MatchingPoolDecorator接口定义了所有装饰器类需要实现的方法。*  *\* 它将提供一种方式来通过装饰器模式扩展原有的匹配池功能。*  *\*/* public interface MatchingPoolDecorator {  void addPlayer(Integer userId, Integer rating, Integer botId); // zzy：添加玩家  void removePlayer(Integer userId); // zzy：移除玩家  void increasingWaitingTime(); // zzy：增加所有玩家的等待时间  void matchPlayers(); // zzy：尝试匹配所有玩家  void run(); // zzy：启动匹配池的执行 } |

新增MatchingPoolLoggingDecorator.java文件

|  |
| --- |
| JavaScript package com.kob.matchingsystem.utils;  */\*\**  *\* zzy：MatchingPoolLoggingDecorator为匹配池增加日志记录功能。*  *\* zzy：每次操作（如添加玩家、移除玩家等）都会在控制台输出日志。*  *\*/* public class MatchingPoolLoggingDecorator implements MatchingPoolDecorator {  private MatchingPoolDecorator matchingPool; // zzy：被装饰的原始匹配池   // zzy：构造函数，接收一个MatchingPoolDecorator实例作为参数  public MatchingPoolLoggingDecorator(MatchingPoolDecorator matchingPool) {  this.matchingPool = matchingPool;  }   // zzy：添加玩家时输出日志  @Override  public void addPlayer(Integer userId, Integer rating, Integer botId) {  System.*out*.println("Adding player: userId=" + userId + ", rating=" + rating + ", botId=" + botId); // zzy：日志记录  matchingPool.addPlayer(userId, rating, botId); // zzy：调用被装饰的addPlayer方法  }   // zzy：移除玩家时输出日志  @Override  public void removePlayer(Integer userId) {  System.*out*.println("Removing player: userId=" + userId); // zzy：日志记录  matchingPool.removePlayer(userId); // zzy：调用被装饰的removePlayer方法  }   // zzy：增加等待时间时输出日志  @Override  public void increasingWaitingTime() {  System.*out*.println("Increasing waiting time for all players"); // zzy：日志记录  matchingPool.increasingWaitingTime(); // zzy：调用被装饰的increasingWaitingTime方法  }   // zzy：匹配玩家时输出日志  @Override  public void matchPlayers() {  System.*out*.println("Attempting to match players"); // zzy：日志记录  matchingPool.matchPlayers(); // zzy：调用被装饰的matchPlayers方法  }   // zzy：启动匹配池的执行时输出日志  @Override  public void run() {  System.*out*.println("Starting matching pool execution"); // zzy：日志记录  matchingPool.run(); // zzy：调用被装饰的run方法  } } |

修改后的MatchingPool.java文件

|  |
| --- |
| JavaScript package com.kob.matchingsystem.utils;  import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Component;  */\*\**  *\* zzy：MatchingPool类作为系统的统一入口，*  *\* 通过装饰器模式增强匹配池的功能。*  *\*/* @Component public class MatchingPool extends Thread {  private final MatchingPoolDecorator matchingPool;   // zzy：构造函数，注入装饰的匹配池实例  @Autowired  public MatchingPool(BaseMatchingPool baseMatchingPool) {  this.matchingPool = new MatchingPoolLoggingDecorator(baseMatchingPool); // zzy：通过装饰器增强功能  }   // zzy：对外提供的添加玩家接口  public void addPlayer(Integer userId, Integer rating, Integer botId) {  matchingPool.addPlayer(userId, rating, botId);  }   // zzy：对外提供的移除玩家接口  public void removePlayer(Integer userId) {  matchingPool.removePlayer(userId);  }   // zzy：启动匹配池执行  @Override  public void run() {  matchingPool.run();  } } |

3. **重构前后主要差异**

3.1 **代码结构差异**

**重构前**

* 所有功能都集中在一个类 MatchingPool 中。
* MatchingPool 包含多个职责：玩家管理、日志记录、线程同步等，违背了单一职责原则 (SRP)。
* 扩展新功能（如日志记录）需要修改原类代码，违背了开闭原则 (OCP)。

**重构后**

* 功能被拆分为多个独立类：
* **BaseMatchingPool**：核心匹配功能（玩家管理和匹配逻辑）。
* **装饰器类**：对基础功能进行增强：
* **MatchingPoolLoggingDecorator**：添加日志记录功能。
* 未来可扩展更多装饰器（如性能监控、错误处理等）。
* **MatchingPool**：提供统一入口，管理装饰器链。
* 装饰器模式使扩展功能更易于实现，无需修改核心代码，符合开闭原则。

3.2 **代码职责差异**

**重构前**

* MatchingPool 既负责玩家管理（如 addPlayer、removePlayer），又负责线程同步和日志记录。
* 日志记录等辅助功能与主要功能耦合，导致代码复杂度增加，难以单独测试和维护。

**重构后**

* **单一职责**：
* BaseMatchingPool 专注于核心逻辑（玩家添加、移除、匹配）。
* 装饰器类专注于辅助功能（如日志记录）。
* **职责分离** 提高了代码可读性和可维护性，每个类都只专注于一项功能。

3.3 **可扩展性差异**

**重构前**

* 扩展功能需要修改 MatchingPool 源代码。
* 增加新功能可能引入潜在的代码冲突或逻辑错误。

**重构后**

* 新功能可通过创建新的装饰器类实现，无需修改原有代码。
* 装饰器链可以动态组合，灵活增强匹配池功能。例如，日志记录和性能监控装饰器可以同时生效。

3.4 **测试和维护差异**

**重构前**

* 功能耦合，难以单独测试（如无法单独测试日志记录逻辑）。
* 排查问题时需要梳理所有耦合逻辑，调试难度较高。

**重构后**

* 各个类职责单一，可以单独测试（如仅测试日志装饰器的功能）。
* 排查问题时，可以快速定位到具体的装饰器类或基础功能类。

3.5 **性能和线程安全差异**

**重构前**

* 线程安全逻辑由 ReentrantLock 直接嵌套在每个方法中。
* 锁的使用分散在多个地方，不易集中管理。

**重构后**

* 线程安全逻辑集中在 BaseMatchingPool，通过继承或装饰实现，保证一致性。
* 装饰器增强逻辑在外层处理，不影响核心锁的实现。

3.6 **重构前后主要差异对比表**



**点击图片可查看完整电子表格**

如果有进一步要求，请告诉我！

4. **UML 类图**

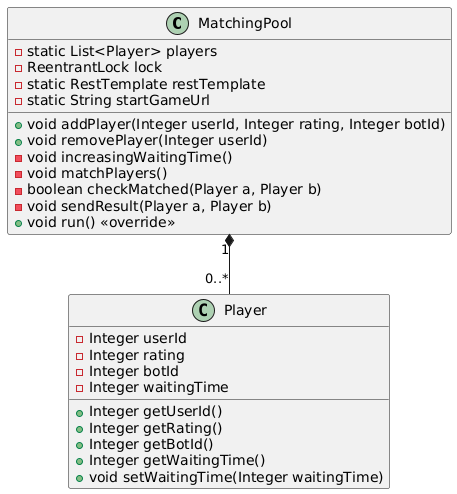


图4-1 重构前

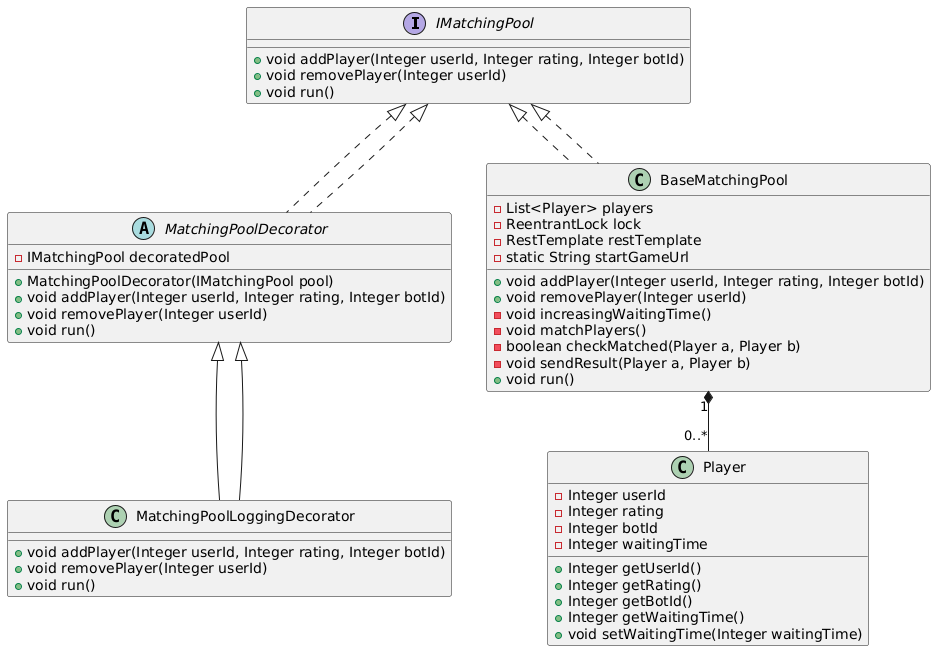


图4-2 重构后

5. **修改的原因**

5.1 **职责过于集中**

* **问题**：  
  MatchingPool 类同时负责多个功能，包括玩家管理、匹配逻辑、线程安全操作和日志记录。  
  **影响**：  
   代码复杂且难以维护，任何修改都会对整个类产生潜在影响，增加了风险。
* **解决**：  
   使用装饰器模式，将职责分离到多个类中。例如：
* 核心匹配功能由 BaseMatchingPool 实现。
* 日志记录功能由 MatchingPoolLoggingDecorator 动态增强。

5.2 **扩展性差**

* **问题**：  
   如果需要新增功能（如性能监控或其他行为），需要直接修改 MatchingPool 类。  
  **影响**：  
   违反了开闭原则（Open-Closed Principle），代码不易扩展且容易引入新问题。
* **解决**：  
   使用装饰器模式，可以通过创建新的装饰器类（如 MatchingPoolPerformanceMonitorDecorator）动态扩展功能，无需修改核心逻辑。

5.3 **测试困难**

* **问题**：  
   所有功能耦合在一个类中，难以单独测试每个功能模块。  
  **影响**：  
   测试覆盖率低，排查问题困难。
* **解决**：  
   重构后，每个功能封装在独立的类中，可以单独测试其逻辑，增强测试能力。

5.4 **不易理解的代码结构**

* **问题**：  
  MatchingPool 类内容庞杂，阅读成本高，开发人员需要理解所有细节才能进行修改。  
  **影响**：  
   开发效率低且容易引入错误。
* **解决**：  
   使用接口（IMatchingPool）定义匹配池功能，具体实现通过类和装饰器组合，结构清晰，易于理解。

5.5 **缺乏灵活性**

* **问题**：  
   功能是硬编码在 MatchingPool 类中，无法根据需求动态切换或组合功能。  
  **影响**：  
   难以满足多样化的需求场景。
* **解决**：  
   装饰器模式允许在运行时动态组合功能，例如可以仅启用日志记录或性能监控，灵活性大幅提升。