**重构说明文档**

**外观模式 (Facade) 重构**

1. **修改前的代码**

|  |
| --- |
| 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 ReentrantLock lock = new ReentrantLock();  private String status = "playing"; // playing finished  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<>());  }   private String mapToString(){  StringBuilder res = new StringBuilder();  for(int i = 0; i < rows; i++){  for(int j = 0; j < cols; j++){  res.append(gameMap[i][j]);  }  }  return res.toString();  }   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 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;  }  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);  }  public void createGameMap(){  for(int i = 0; i < 1000; i ++){  if(createWalls()){  break;  }  }  }   private String getInput(Player player){ // 将当前的局面信息 编码成字符串  Player me, you;  if(Objects.equals(player.getId(), playerA.getId())){  me = playerA;  you = playerB;  } else {  me = playerB;  you = playerA;  }  return mapToString() + '#' +  me.getSx().toString() + '#' +  me.getSy().toString() + "#(" +  me.stepsToString() + ")#" +  you.getSx().toString() + '#' +  you.getSy().toString() + "#(" +  you.stepsToString() + ")#";  }  private void sendBotCode(Player player){  if(player.getBotId() == -1)return;  MultiValueMap<String, String> data = new LinkedMultiValueMap<>();  data.add("userId", player.getId().toString());  data.add("botCode", player.getBotCode());  data.add("input", getInput(player));  String resp = WebSocketServer.restTemplate.postForObject(addBotUrl, data, String.class);  System.out.println(resp);  }  private boolean nextStep(){ // 等待两名玩家下一步操作  try{ // 前端蛇的移动比较慢，等前端  Thread.sleep(200);  } catch (InterruptedException e) {  e.printStackTrace();  }   sendBotCode(playerA);  sendBotCode(playerB);   for(int i = 0; i < 50; i++){  try{  Thread.sleep(100);  lock.lock();  try{  if(nextStepA !=null && nextStepB != null){  playerA.getSteps().add(nextStepA);  playerB.getSteps().add(nextStepB);  return true;  }  } finally {  lock.unlock();  }  } catch(InterruptedException e){  e.printStackTrace();  }  }  return false;  }  private 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);  }  }  private void sendResult(){  JSONObject resp = new JSONObject();  resp.put("event", "result");  resp.put("loser", loser);  saveRecordToDataBase();  sendAllMessage(resp.toJSONString());  }  private 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();  }  }  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;  }  private 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 if(!validB){  loser = "B";  }  }  }   private void saveRecordToDataBase(){  Integer aRating = WebSocketServer.userMapper.selectById(playerA.getId()).getRating();  Integer bRating = WebSocketServer.userMapper.selectById(playerB.getId()).getRating();  if("A".equals(loser)){  aRating -= 2;  bRating += 5;  } else if("B".equals(loser)){  aRating += 5;  bRating -= 2;  }  updateUserRating(playerA, aRating);  updateUserRating(playerB, bRating);   Record record = new Record(  null,  playerA.getId(),  playerA.getSx(),  playerA.getSy(),  playerB.getId(),  playerB.getSx(),  playerB.getSy(),  playerA.stepsToString(),  playerB.stepsToString(),  mapToString(),  loser,  new Date()  );  WebSocketServer.recordMapper.insert(record);  }   private void updateUserRating(Player player, Integer rating){  User user = WebSocketServer.userMapper.selectById(player.getId());  user.setRating(rating);  WebSocketServer.userMapper.updateById(user);  }   @Override  public void run() {  try {  Thread.sleep(2000); // 前端有两秒跳转的时间  } catch (InterruptedException e) {  throw new RuntimeException(e);  }  for(int i = 0; i < 1000; i++){ // 最多1000回合一定结束  if(nextStep()){  judge();  if("playing".equals(status)){  sendMove();  } else {  sendResult();  break;  }  } else {  status = "finished";  lock.lock();  try{  if(nextStepA == null && nextStepB == null){  loser = "all";  } else if(nextStepA == null){  loser = "A";  } else {  loser = "B";  }  } finally {  lock.unlock();  }  sendResult();  break;  }  }   } } |

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**

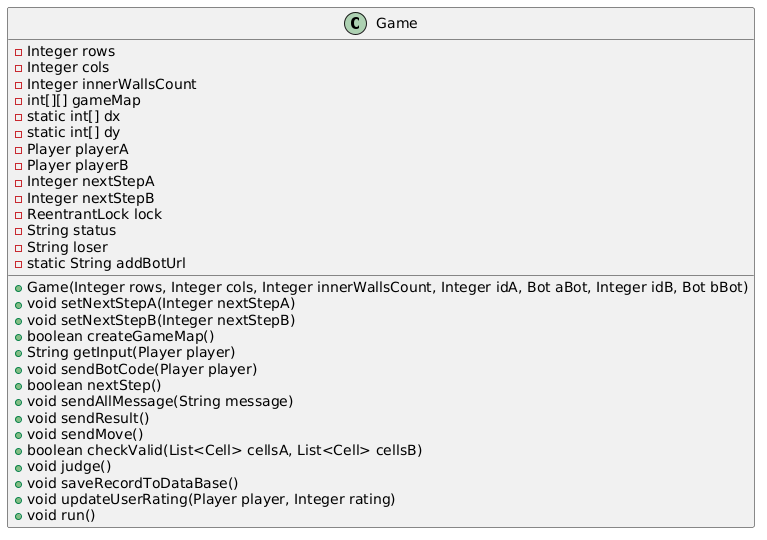


图4-1 重构前

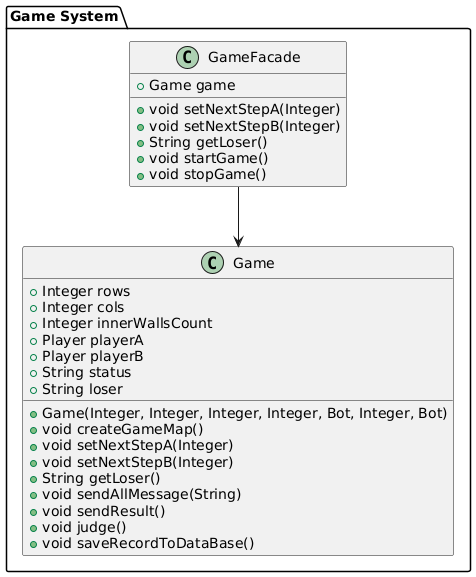


图4-2 重构后

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**

4. **UML 类图**

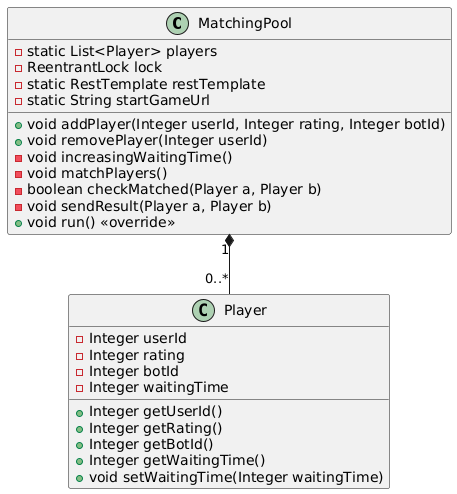


图4-1 重构前

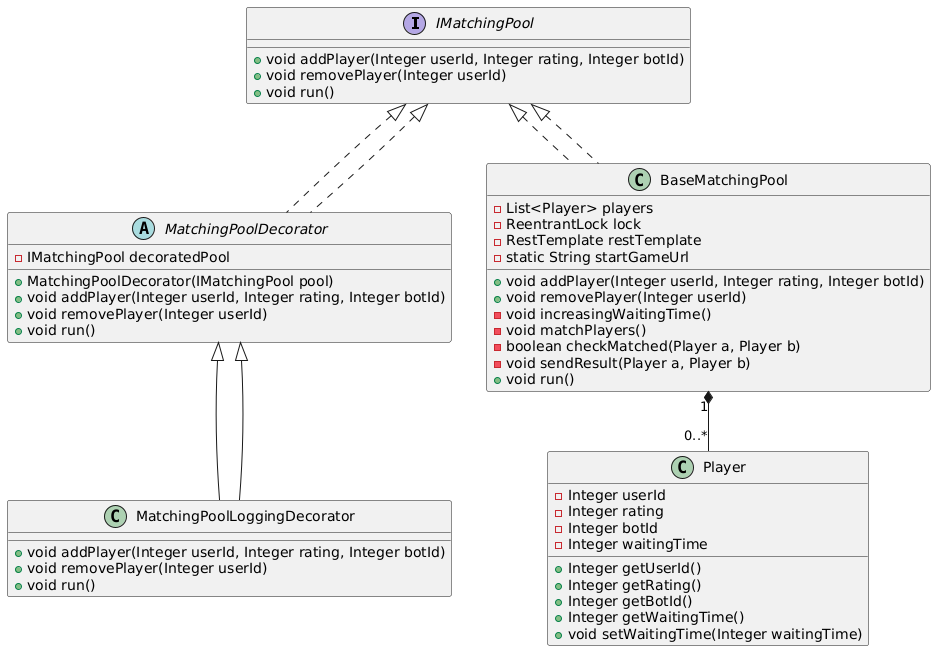


图4-2 重构后

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。

**责任链模式 (Chain of Command)重构**

1. **修改前的代码**

|  |
| --- |
| JavaScript package com.kob.backend.consumer;  import com.alibaba.fastjson.JSONObject; import com.kob.backend.mapper.BotMapper; import com.kob.backend.mapper.RecordMapper; import com.kob.backend.mapper.UserMapper; import com.kob.backend.pojo.Bot; import com.kob.backend.pojo.User; import com.kob.backend.utils.JwtUtil; import io.jsonwebtoken.Claims; 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 javax.websocket.\*; import javax.websocket.server.PathParam; import javax.websocket.server.ServerEndpoint; import java.io.IOException; import java.util.Iterator; import java.util.concurrent.ConcurrentHashMap; import java.util.concurrent.CopyOnWriteArraySet;  import static com.kob.backend.constants.Constants.\*;  @Component @ServerEndpoint("/websocket/{token}") // 注意不要以'/'结尾 public class WebSocketServer {   //线程安全的静态变量存储客户端id和websockeserver的对应关系  final public static ConcurrentHashMap<Integer, WebSocketServer> *users* = new ConcurrentHashMap<>();  //匹配池，线程安全  //final private static CopyOnWriteArraySet<User> matchPool = new CopyOnWriteArraySet<>();   private User user = null;  private Session session = null;  public Game game = null;  private final String addPlayerUrl = "http://localhost:3001/player/add/";  private final String removePlayerUrl = "http://localhost:3001/player/remove/";   public static UserMapper *userMapper*;  public static RecordMapper *recordMapper*;  public static RestTemplate *restTemplate*;  private static BotMapper *botMapper*;  @Autowired  public void setUserMapper(UserMapper userMapper) {  WebSocketServer.*userMapper* = userMapper;  }  @Autowired  public void setRecordMapper(RecordMapper recordMapper) {  WebSocketServer.*recordMapper* = recordMapper;  }  @Autowired  public void setRestTemplate(RestTemplate restTemplate) {  WebSocketServer.*restTemplate* = restTemplate;  }  @Autowired  public void setBotMapper(BotMapper botMapper) {  WebSocketServer.*botMapper* = botMapper;  }   @OnOpen  public void onOpen(Session session, @PathParam("token") String token) throws IOException {  System.*out*.println("连接了一个客户端");  // 建立连接  int userId = -1;  try {  Claims claims = JwtUtil.*parseJWT*(token);  userId = Integer.*parseInt*(claims.getSubject());  } catch (Exception e) {  throw new RuntimeException(e);  }  if(userId == -1) {  session.close();  } else {  this.session = session;  this.user = *userMapper*.selectById(userId);  *users*.put(userId, this);  }  }  @OnClose  public void onClose() {  System.*out*.println("断开了一个客户端的连接");  // 关闭链接  if(user != null) {  *users*.remove(user.getId());  //matchPool.remove(user);  }  }  @OnMessage  public void onMessage(String message, Session session) { // onMessage，一般用来做分类，根据event的内容，转给不同方法处理  System.*out*.println("收到来自客户端的信息"); // 客户端向服务器端发  JSONObject data = JSONObject.*parseObject*(message);  String event = data.getString("event");   if("start-match".equals(event)) {  startMatch(data.getInteger("bot\_id"));  } else if("stop-match".equals(event)) {  stopMatch();  } else if("move".equals(event)) {  move(data.getInteger("d"));  }  }  @OnError  public void onError(Session session, Throwable error) {  error.printStackTrace();  }   //在机器人对战时，人的输入不接收  private void move(int d) {  if(game.getPlayerA().getId().equals(user.getId())) {  if(game.getPlayerA().getBotId() == -1)game.setNextStepA(d);  } else if(game.getPlayerB().getId().equals(user.getId())) {  if(game.getPlayerB().getBotId() == -1)game.setNextStepB(d);  }  }  public static void startGame(Integer aId, Integer aBotId, Integer bId, Integer bBotId) {  User a = *userMapper*.selectById(aId);User b = *userMapper*.selectById(bId);  Bot aBot = *botMapper*.selectById(aBotId);Bot bBot = *botMapper*.selectById(bBotId);   Game game = new Game(*ROWS*, *COLS*, *INNER\_WALLS\_COUNT*, a.getId(), aBot, b.getId(), bBot);  game.createGameMap();  if(*users*.get(a.getId()) != null){ //在玩家匹配时意外断开，但是玩家仍在匹配池中的情况，此时WebSocketServer是空，会空指针  *users*.get(a.getId()).game = game;  }  if(*users*.get(b.getId()) != null){  *users*.get(b.getId()).game = game;  }  game.start();   JSONObject resp = new JSONObject();   resp.put("a\_id", game.getPlayerA().getId());  resp.put("a\_sx", game.getPlayerA().getSx());  resp.put("a\_sy", game.getPlayerA().getSy());   resp.put("b\_id", game.getPlayerB().getId());  resp.put("b\_sx", game.getPlayerB().getSx());  resp.put("b\_sy", game.getPlayerB().getSy());   resp.put("map", game.getGameMap());   JSONObject respA = new JSONObject();  JSONObject respB = new JSONObject();  respA.put("opponent\_name", b.getUsername());  respA.put("opponent\_photo", b.getPhoto());  respA.put("event", "match-found");  respA.put("me", "A");  respA.put("game", resp);   respB.put("opponent\_name", a.getUsername());  respB.put("opponent\_photo", a.getPhoto());  respB.put("event", "match-found");  respB.put("me", "B");  respB.put("game", resp);   if(*users*.get(a.getId()) != null){  *users*.get(a.getId()).sendMessage(respA.toJSONString());  }  if(*users*.get(b.getId()) != null){  *users*.get(b.getId()).sendMessage(respB.toJSONString());  }   }  //先点的左下角后点的右上角  private void startMatch(Integer botId){  System.*out*.println("调试信息：开始匹配");  MultiValueMap<String, String> data = new LinkedMultiValueMap<>();  data.add("userId", user.getId().toString());  data.add("rating", user.getRating().toString());  data.add("botId", botId.toString());  String resp = *restTemplate*.postForObject(addPlayerUrl, data, String.class);  System.*out*.println(resp); // matchPool.add(user); // while(matchPool.size() >= 2){ // Iterator<User> iterator = matchPool.iterator(); // 先进去的是a，左下 // User a = iterator.next(), b = iterator.next(); // matchPool.remove(a); // matchPool.remove(b); // // // }  }  private void stopMatch(){  System.*out*.println("调试信息：停止匹配");  MultiValueMap<String, String> data = new LinkedMultiValueMap<>();  data.add("userId", user.getId().toString());  *restTemplate*.postForObject(removePlayerUrl, data, String.class);  //matchPool.remove(user);  }  public void sendMessage(String message) { // 服务器端向客户端发送  synchronized (session) {  try{  session.getBasicRemote().sendText(message);  } catch (IOException e) {  e.printStackTrace();  }  }  } } |

2. **修改后的代码**

修改后的WebSocketServer.java内容

|  |
| --- |
| JavaScript package com.kob.backend.consumer;  import com.alibaba.fastjson.JSONObject; import com.kob.backend.consumer.handler.MessageHandler; import com.kob.backend.consumer.handler.MoveHandler; import com.kob.backend.consumer.handler.StartMatchHandler; import com.kob.backend.consumer.handler.StopMatchHandler; import com.kob.backend.mapper.BotMapper; import com.kob.backend.mapper.RecordMapper; import com.kob.backend.mapper.UserMapper; import com.kob.backend.pojo.Game; import com.kob.backend.pojo.User; import com.kob.backend.utils.JwtUtil; import io.jsonwebtoken.Claims; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Component; import org.springframework.web.client.RestTemplate;  import javax.websocket.\*; import javax.websocket.server.PathParam; import javax.websocket.server.ServerEndpoint; import java.io.IOException; import java.util.concurrent.ConcurrentHashMap;  @Component @ServerEndpoint("/websocket/{token}") public class WebSocketServer {  // zzy: 静态变量存储客户端ID和WebSocketServer实例的对应关系  public static final ConcurrentHashMap<Integer, WebSocketServer> *users* = new ConcurrentHashMap<>();   private MessageHandler chain; // zzy: 责任链的起点  private Session session; // zzy: 当前WebSocket的会话  private User user; // zzy: 当前用户信息  public Game game; // zzy: 当前用户所在的游戏   // Spring Bean注入  public static UserMapper *userMapper*;  public static RecordMapper *recordMapper*;  public static RestTemplate *restTemplate*;  public static BotMapper *botMapper*;   @Autowired  public void setUserMapper(UserMapper userMapper) {  WebSocketServer.*userMapper* = userMapper;  }   @Autowired  public void setRecordMapper(RecordMapper recordMapper) {  WebSocketServer.*recordMapper* = recordMapper;  }   @Autowired  public void setRestTemplate(RestTemplate restTemplate) {  WebSocketServer.*restTemplate* = restTemplate;  }   @Autowired  public void setBotMapper(BotMapper botMapper) {  WebSocketServer.*botMapper* = botMapper;  }   // zzy: 初始化责任链  public WebSocketServer() {  initChain();  }   private void initChain() {  MessageHandler startMatchHandler = new StartMatchHandler();  MessageHandler stopMatchHandler = new StopMatchHandler();  MessageHandler moveHandler = new MoveHandler();   startMatchHandler.setNext(stopMatchHandler);  stopMatchHandler.setNext(moveHandler);   this.chain = startMatchHandler;  }   @OnOpen  public void onOpen(Session session, @PathParam("token") String token) throws IOException {  System.*out*.println("连接了一个客户端");  int userId = -1;  try {  Claims claims = JwtUtil.*parseJWT*(token);  userId = Integer.*parseInt*(claims.getSubject());  } catch (Exception e) {  e.printStackTrace();  }   if (userId == -1) {  session.close();  return;  }   this.session = session;  this.user = *userMapper*.selectById(userId);   if (this.user != null) {  *users*.put(userId, this);  } else {  session.close();  }  }   @OnClose  public void onClose() {  System.*out*.println("断开了一个客户端的连接");  if (user != null) {  *users*.remove(user.getId());  }  }   @OnMessage  public void onMessage(String message, Session session) {  System.*out*.println("收到来自客户端的信息: " + message);  JSONObject data = JSONObject.*parseObject*(message);  String event = data.getString("event");   // zzy: 使用责任链处理事件  chain.handle(event, data, this);  }   @OnError  public void onError(Session session, Throwable error) {  error.printStackTrace();  }   public void sendMessage(String message) {  synchronized (session) {  try {  session.getBasicRemote().sendText(message);  } catch (IOException e) {  e.printStackTrace();  }  }  }   // zzy: 游戏开始逻辑  public static void startGame(Integer aId, Integer aBotId, Integer bId, Integer bBotId) {  User a = *userMapper*.selectById(aId);  User b = *userMapper*.selectById(bId);   Game game = new Game(13, 13, 20, a.getId(), null, b.getId(), null);  game.createGameMap();   if (*users*.get(a.getId()) != null) {  *users*.get(a.getId()).game = game;  }  if (*users*.get(b.getId()) != null) {  *users*.get(b.getId()).game = game;  }  game.start();   JSONObject resp = new JSONObject();  resp.put("a\_id", game.getPlayerA().getId());  resp.put("b\_id", game.getPlayerB().getId());  resp.put("map", game.getGameMap());   if (*users*.get(a.getId()) != null) {  *users*.get(a.getId()).sendMessage(resp.toJSONString());  }  if (*users*.get(b.getId()) != null) {  *users*.get(b.getId()).sendMessage(resp.toJSONString());  }  }   // zzy: Getter方法，提供给处理器使用  public User getUser() {  return user;  }   public RestTemplate getRestTemplate() {  return *restTemplate*;  }   public String getAddPlayerUrl() {  String addPlayerUrl = "http://localhost:3001/player/add/";  return addPlayerUrl;  }   public String getRemovePlayerUrl() {  String removePlayerUrl = "http://localhost:3001/player/remove/";  return removePlayerUrl;  }   public Game getGame() {  return game;  } } |

新增MessageHandler.java文件

|  |
| --- |
| JavaScript package com.kob.backend.consumer.handler;  import com.alibaba.fastjson.JSONObject; import com.kob.backend.consumer.WebSocketServer;  // zzy: 定义抽象事件处理器 public abstract class MessageHandler {  protected MessageHandler next;   public void setNext(MessageHandler next) {  this.next = next;  }   public void handle(String event, JSONObject data, WebSocketServer server) {  if (canHandle(event)) {  process(data, server);  } else if (next != null) {  next.handle(event, data, server);  }  }   // zzy: 判断是否能处理当前事件  protected abstract boolean canHandle(String event);   // zzy: 处理当前事件的具体逻辑  protected abstract void process(JSONObject data, WebSocketServer server); } |

新增MoveHandler.java文件

|  |
| --- |
| JavaScript package com.kob.backend.consumer.handler;  import com.alibaba.fastjson.JSONObject;  // zzy: 处理“移动”事件 public class MoveHandler extends MessageHandler {  private final Game game; // 引用 Game 对象   public MoveHandler() {  this.game = game;  }   @Override  public void handle(JSONObject message) {  // 提取移动方向  int direction = message.getInteger("d");   // 判断玩家是 A 还是 B，并设置下一步操作  if (game.getPlayerA().getId().equals(user.getId())) {  if (game.getPlayerA().getBotId() == -1) {  game.setNextStepA(direction); // 设置玩家 A 的下一步操作  }  } else if (game.getPlayerB().getId().equals(user.getId())) {  if (game.getPlayerB().getBotId() == -1) {  game.setNextStepB(direction); // 设置玩家 B 的下一步操作  }  }  } } |

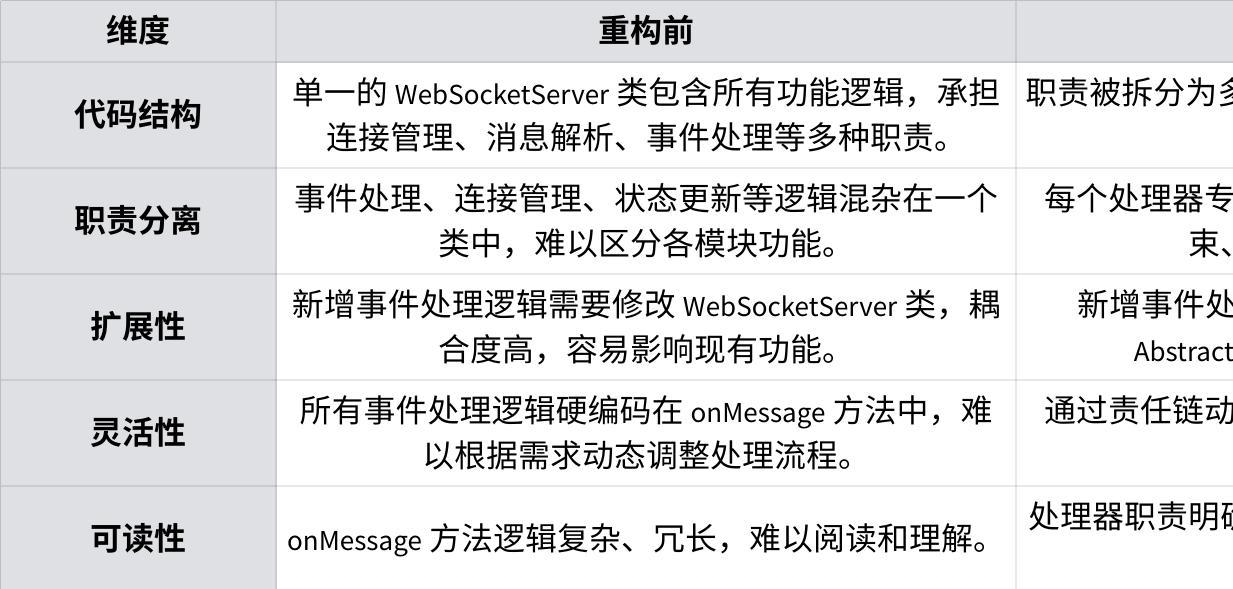
新增的StopMatchHandler.java文件

|  |
| --- |
| JavaScript package com.kob.backend.consumer.handler;  import com.alibaba.fastjson.JSONObject; import com.kob.backend.consumer.WebSocketServer; import org.springframework.util.LinkedMultiValueMap;  // zzy: 处理“停止匹配”事件 public class StopMatchHandler extends MessageHandler {  @Override  protected boolean canHandle(String event) {  return "stop-match".equals(event);  }   @Override  protected void process(JSONObject data, WebSocketServer server) {  System.*out*.println("调试信息：停止匹配");  LinkedMultiValueMap<String, String> requestData = new LinkedMultiValueMap<>();  requestData.add("userId", server.getUser().getId().toString());  server.getRestTemplate().postForObject(server.getRemovePlayerUrl(), requestData, String.class);  } } |

新增的StartMatchHandler.java文件

|  |
| --- |
| JavaScript package com.kob.backend.consumer.handler;  import com.alibaba.fastjson.JSONObject; import com.kob.backend.consumer.WebSocketServer; import org.springframework.util.LinkedMultiValueMap;  // zzy: 处理“开始匹配”事件 public class StartMatchHandler extends MessageHandler {  @Override  protected boolean canHandle(String event) {  return "start-match".equals(event);  }   @Override  protected void process(JSONObject data, WebSocketServer server) {  Integer botId = data.getInteger("bot\_id");  System.*out*.println("调试信息：开始匹配");  LinkedMultiValueMap<String, String> requestData = new LinkedMultiValueMap<>();  requestData.add("userId", server.getUser().getId().toString());  requestData.add("rating", server.getUser().getRating().toString());  requestData.add("botId", botId.toString());  String response = server.getRestTemplate().postForObject(server.getAddPlayerUrl(), requestData, String.class);  System.*out*.println(response);  } } |

3. **重构前后主要差异**



**点击图片可查看完整电子表格**

4. **UML 类图**

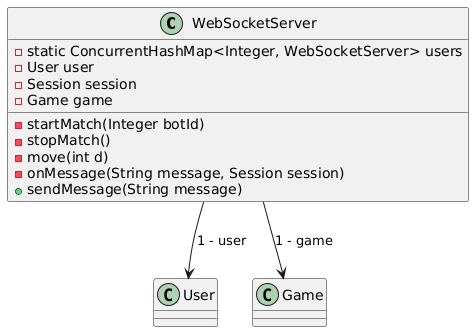


图4-1 重构前

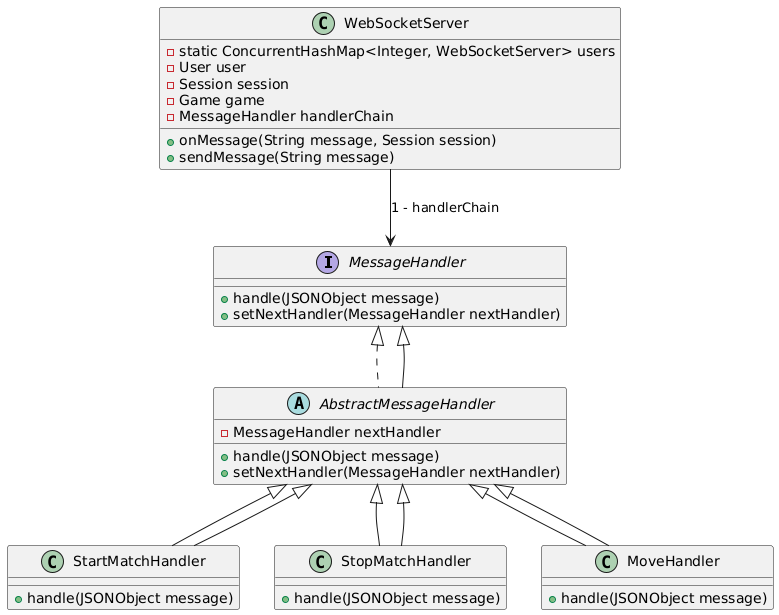


图4-2 重构后

5. **修改的原因**

5.1 **代码可维护性较低**

* **重构前问题**：  
  WebSocketServer 类承担了过多职责，包含用户连接管理、消息解析、事件处理等所有功能，代码耦合度高。修改一处逻辑可能影响其他功能，维护成本高。
* **重构后改进**：  
   职责被拆分为多个独立的处理器类，每个类只负责一个具体功能。代码模块化后，定位和修改问题更加容易，维护成本显著降低。

5.2 **扩展性不足**

* **重构前问题**：  
   新增消息事件时需要修改 WebSocketServer 的 onMessage 方法。这种硬编码方式不符合开闭原则，容易引入错误并增加代码复杂度。
* **重构后改进**：  
   引入责任链模式后，可以通过添加新的处理器类来实现新功能，而无需修改原有代码，符合开闭原则，扩展性更强。

5.3 **代码复杂度过高**

* **重构前问题**：  
   单一的 onMessage 方法处理所有消息类型，逻辑复杂且混乱，难以阅读和理解。随着功能增加，代码复杂度呈指数级增长。
* **重构后改进**：  
   通过责任链模式，将消息处理拆分到多个独立的处理器中，每个处理器逻辑清晰、独立，代码简洁明了，易于阅读和理解。

5.4 **职责不明确**

* **重构前问题**：  
  WebSocketServer 类既处理底层网络连接，又负责应用层的业务逻辑。这种设计违背了单一职责原则，导致职责混杂。
* **重构后改进**：  
   网络连接和事件处理被分离，WebSocketServer 仅负责网络连接管理，消息处理逻辑由责任链中的处理器完成，职责划分更加明确。

5.5 **测试困难**

* **重构前问题**：  
   所有逻辑集中在一个类中，无法单独测试特定功能，导致测试覆盖率低，缺陷难以发现和修复。
* **重构后改进**：  
   每个处理器类独立设计，可以单独测试每种消息类型的处理逻辑，测试覆盖率和效率显著提高。

5.6 **灵活性不足**

* **重构前问题**：  
  onMessage 方法中消息处理逻辑是硬编码的，无法根据需求动态调整处理流程。
* **重构后改进**：  
   责任链模式支持动态调整处理器顺序或内容，灵活应对复杂业务需求。

**策略模式 (Strategy)重构**

1. **修改前的代码**

|  |
| --- |
| JavaScript package com.kob.botrunningsystem.utils;  import java.util.LinkedList; import java.util.Queue; import java.util.concurrent.locks.Condition; import java.util.concurrent.locks.ReentrantLock;  public class BotPool extends Thread {  private final ReentrantLock lock = new ReentrantLock();  private final Condition condition = lock.newCondition();  private final Queue<Bot> bots = new LinkedList<>(); // 消息队列   public void addBot(Integer userId, String botCode, String input){  lock.lock();  try{  bots.add(new Bot(userId, botCode, input));  condition.signalAll(); // 一共两个线程，添加bot后唤醒BotPool线程  } finally {  lock.unlock();  }  }  private void consume(Bot bot) { // 用一个线程去执行，可以控制执行的时间  Consumer consumer = new Consumer();  consumer.startTimeout(2000, bot);  }   @Override  public void run() {  while(true){  lock.lock();  if(bots.isEmpty()){  try {  condition.await(); // await包含释放锁的操作  } catch (InterruptedException e) {  e.printStackTrace();  lock.unlock();  break;  }  } else {  Bot bot = bots.remove();  lock.unlock();  consume(bot); // 比较耗时，unlock要在前面  }  }  } } |

2. **修改后的代码**

*修改后Bot*.java内容

|  |
| --- |
| JavaScript package com.kob.botrunningsystem.utils;  import java.util.LinkedList; import java.util.Queue; import java.util.concurrent.locks.Condition; import java.util.concurrent.locks.ReentrantLock;  */\*\**  *\* zzy: BotPool类，使用线程安全队列管理Bot任务*  *\*/* public class BotPool extends Thread {  private final ReentrantLock lock = new ReentrantLock(); // zzy: 锁对象  private final Condition condition = lock.newCondition(); // zzy: 条件变量  private final Queue<Bot> bots = new LinkedList<>(); // zzy: 消息队列   private BotExecutionStrategy botExecutionStrategy; // zzy: 策略对象   */\*\**  *\* zzy: 设置策略对象*  *\* @param botExecutionStrategy 具体的策略实现*  *\*/* public void setBotExecutionStrategy(BotExecutionStrategy botExecutionStrategy) {  this.botExecutionStrategy = botExecutionStrategy;  }   */\*\**  *\* zzy: 添加Bot任务到队列，并唤醒等待线程*  *\* @param userId 用户ID*  *\* @param botCode Bot代码*  *\* @param input Bot输入信息*  *\*/* public void addBot(Integer userId, String botCode, String input) {  lock.lock(); // zzy: 获取锁  try {  bots.add(new Bot(userId, botCode, input)); // zzy: 将Bot任务添加到队列  condition.signalAll(); // zzy: 唤醒等待线程  } finally {  lock.unlock(); // zzy: 释放锁  }  }   */\*\**  *\* zzy: 消费Bot任务，调用策略执行*  *\* @param bot Bot对象*  *\*/* private void consume(Bot bot) {  if (botExecutionStrategy != null) {  botExecutionStrategy.execute(bot); // zzy: 使用当前策略执行Bot任务  } else {  throw new IllegalStateException("BotExecutionStrategy is not set"); // zzy: 策略未设置  }  }   */\*\**  *\* zzy: 主线程运行逻辑，持续消费队列中的任务*  *\*/* @Override  public void run() {  while (true) {  lock.lock(); // zzy: 获取锁  try {  if (bots.isEmpty()) {  try {  condition.await(); // zzy: 等待条件变量  } catch (InterruptedException e) {  e.printStackTrace();  break; // zzy: 线程中断时退出循环  }  } else {  Bot bot = bots.remove(); // zzy: 从队列中取出任务  consume(bot); // zzy: 消费任务  }  } finally {  lock.unlock(); // zzy: 释放锁  }  }  } } |

新增BotExecutionStrategy.java文件

|  |
| --- |
| JavaScript package com.kob.botrunningsystem.utils;  */\*\**  *\* zzy: 定义策略接口，提供统一的Bot执行方式*  *\*/* public interface BotExecutionStrategy {  */\*\**  *\* zzy: 定义执行方法*  *\* @param bot 需要执行的Bot对象*  *\*/* void execute(Bot bot); } |

新增TimeoutExecutionStrategy.java文件

|  |
| --- |
| JavaScript package com.kob.botrunningsystem.utils;  */\*\**  *\* zzy: 具体策略类，使用超时机制执行Bot*  *\*/* public class TimeoutExecutionStrategy implements BotExecutionStrategy {   @Override  public void execute(Bot bot) {  // zzy: 调用Consumer工具类执行Bot，并设置超时时间为2000ms  Consumer consumer = new Consumer();  consumer.startTimeout(2000, bot);  } } |

修改后的BotPool.java文件

|  |
| --- |
| JavaScript package com.kob.botrunningsystem.utils;  import java.util.LinkedList; import java.util.Queue; import java.util.concurrent.locks.Condition; import java.util.concurrent.locks.ReentrantLock;  */\*\**  *\* zzy: BotPool类，使用线程安全队列管理Bot任务*  *\*/* public class BotPool extends Thread {  private final ReentrantLock lock = new ReentrantLock(); // zzy: 锁对象  private final Condition condition = lock.newCondition(); // zzy: 条件变量  private final Queue<Bot> bots = new LinkedList<>(); // zzy: 消息队列   private BotExecutionStrategy botExecutionStrategy; // zzy: 策略对象   */\*\**  *\* zzy: 设置策略对象*  *\* @param botExecutionStrategy 具体的策略实现*  *\*/* public void setBotExecutionStrategy(BotExecutionStrategy botExecutionStrategy) {  this.botExecutionStrategy = botExecutionStrategy;  }   */\*\**  *\* zzy: 添加Bot任务到队列，并唤醒等待线程*  *\* @param userId 用户ID*  *\* @param botCode Bot代码*  *\* @param input Bot输入信息*  *\*/* public void addBot(Integer userId, String botCode, String input) {  lock.lock(); // zzy: 获取锁  try {  bots.add(new Bot(userId, botCode, input)); // zzy: 将Bot任务添加到队列  condition.signalAll(); // zzy: 唤醒等待线程  } finally {  lock.unlock(); // zzy: 释放锁  }  }   */\*\**  *\* zzy: 消费Bot任务，调用策略执行*  *\* @param bot Bot对象*  *\*/* private void consume(Bot bot) {  if (botExecutionStrategy != null) {  botExecutionStrategy.execute(bot); // zzy: 使用当前策略执行Bot任务  } else {  throw new IllegalStateException("BotExecutionStrategy is not set"); // zzy: 策略未设置  }  }   */\*\**  *\* zzy: 主线程运行逻辑，持续消费队列中的任务*  *\*/* @Override  public void run() {  while (true) {  lock.lock(); // zzy: 获取锁  try {  if (bots.isEmpty()) {  try {  condition.await(); // zzy: 等待条件变量  } catch (InterruptedException e) {  e.printStackTrace();  break; // zzy: 线程中断时退出循环  }  } else {  Bot bot = bots.remove(); // zzy: 从队列中取出任务  consume(bot); // zzy: 消费任务  }  } finally {  lock.unlock(); // zzy: 释放锁  }  }  } } |

3. **重构前后主要差异**

3.1 **设计模式的引入**

* **重构前**：
* BotPool 类直接包含了所有逻辑，包括任务队列管理和任务执行，职责不清晰。
* 消费逻辑是直接调用 Consumer 类的 startTimeout 方法，紧耦合且难以扩展。
* **重构后**：
* 使用 **策略模式** 将任务执行逻辑从 BotPool 中抽离，形成独立的策略接口 BotExecutionStrategy。
* 可以通过注入不同的策略实现（如 TimeoutExecutionStrategy）动态更改任务执行方式。
* 更符合 **单一职责原则** 和 **开闭原则**。

3.2 **扩展性与灵活性**

* **重构前**：
* 扩展任务的执行方式时需要修改 BotPool 类。
* 没有清晰的扩展点，导致代码难以复用和维护。
* **重构后**：
* 新增任务执行方式时只需实现 BotExecutionStrategy 接口即可，无需修改 BotPool。
* 通过 setBotExecutionStrategy() 方法灵活设置执行策略，提升扩展性。

3.3 **代码耦合度**

* **重构前**：
* BotPool 与任务执行逻辑高度耦合，直接依赖于 Consumer 类，导致类间耦合度过高。
* **重构后**：
* BotPool 只负责队列管理和任务调度，与任务执行逻辑解耦。
* BotExecutionStrategy 接口隔离了 BotPool 和任务执行的具体实现。

3.4 **职责划分**

* **重构前**：
* 单一类 BotPool 同时负责任务的队列管理、调度和执行，职责混乱。
* **重构后**：
* 职责清晰：
* BotPool：负责任务的队列管理和调度。
* BotExecutionStrategy：抽象任务的执行方式。
* TimeoutExecutionStrategy：实现具体的超时任务执行逻辑。

3.5 **测试与维护性**

* **重构前**：
* 测试困难，因为所有逻辑都混杂在一个类中，难以单独测试任务执行逻辑。
* **重构后**：
* 可以单独测试每种 BotExecutionStrategy 的实现，降低测试复杂度。
* BotPool 的逻辑更加简单，易于维护和调试。

4. **UML 类图**

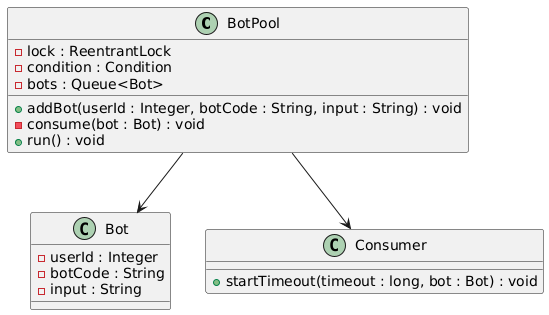


图4-1 重构前

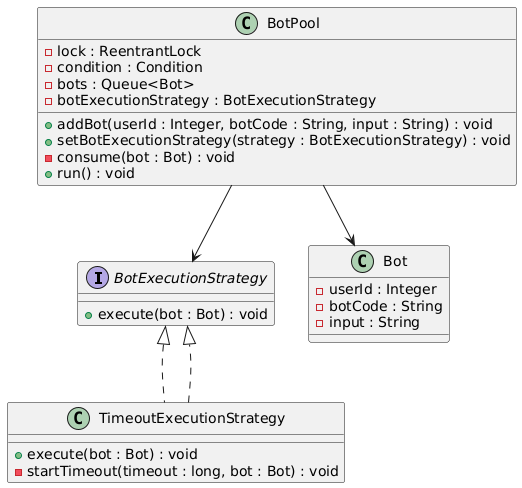


图4-2 重构后

5. **修改的原因**

5.1 **代码可扩展性差**

* **修改前**：BotPool 类直接依赖于 Consumer 类中的固定逻辑，若需要更改任务的执行方式（例如添加新的超时逻辑或非阻塞任务处理），需要修改 BotPool 或 Consumer 类，违反开闭原则 (Open-Closed Principle)。
* **修改后**：通过引入行为型模式的 **策略模式 (Strategy Pattern)**，将任务的执行逻辑抽象为接口 BotExecutionStrategy，实现解耦。BotPool 不再直接依赖于具体的 Consumer，而是依赖于抽象接口，方便切换不同的执行策略（如非超时执行、日志增强等）。

5.2 **职责分离不明确**

* **修改前**：BotPool 同时管理任务队列、线程同步、任务调度和执行逻辑，代码复杂且耦合度高，难以维护。
* **修改后**：将任务执行逻辑交由 BotExecutionStrategy 负责，使 BotPool 专注于任务队列的管理和调度，职责更加明确。

5.3 **增加代码的可测试性**

* **修改前**：BotPool 和 Consumer 紧耦合，无法单独测试任务的执行逻辑，必须依赖完整的队列和线程环境。
* **修改后**：通过将任务执行逻辑抽象为 BotExecutionStrategy，可以独立对不同策略的执行逻辑进行单元测试，而无需依赖 BotPool 的上下文环境。

5.4 **便于扩展新的任务执行策略**

* **修改前**：需要新增任务执行方式时，必须修改 BotPool 或 Consumer，风险较大且容易引入错误。
* **修改后**：只需实现 BotExecutionStrategy 接口即可新增任务执行方式，不需要对现有代码进行改动，符合开闭原则。

5.5 **符合面向对象设计原则 (SOLID)**

* **单一职责原则 (SRP)**：每个类只负责一种功能，例如 BotPool 负责队列管理和调度，BotExecutionStrategy 负责任务执行逻辑。
* **开闭原则 (OCP)**：通过 BotExecutionStrategy 接口，新增或修改任务执行方式无需改动现有代码。
* **依赖倒置原则 (DIP)**：BotPool 依赖于抽象的 BotExecutionStrategy 而非具体实现。