===========================================================

King of Bots (KOB) 项目技术文档详解

纯文本版本

===========================================================

【项目概述】

King of Bots 是一个基于Web的贪吃蛇对战游戏平台，支持真人对战和AI Bot对战。

项目采用微服务架构，前端使用Vue3，后端使用Spring Boot，实现了用户认证、

匹配系统、游戏引擎、Bot管理等完整功能。

核心玩法:

- 双人贪吃蛇对战，13×14的游戏地图

- 支持手动操作和AI Bot自动对战

- 实时匹配系统，根据rating匹配对手

- 完整的用户系统和排行榜功能

【技术架构总览】

整体架构图:

┌─────────────┐ ┌──────────────┐ ┌─────────────┐

│ 前端Web │◄──►│ 主后端 │◄──►│ 数据库 │

│ (Vue3) │ │ (Backend) │ │ (MySQL) │

└─────────────┘ └──────────────┘ └─────────────┘

│

▼

┌──────────────┐

│ 微服务集群 │

│ │

│ ┌──────────┐ │ ┌─────────────┐

│ │匹配系统 │◄┼───►│ Redis │

│ │Matching │ │ │ (缓存) │

│ └──────────┘ │ └─────────────┘

│ │

│ ┌──────────┐ │

│ │Bot运行 │ │

│ │系统 │ │

│ └──────────┘ │

└──────────────┘

===========================================================

【前端架构 (Vue3)】

技术栈:

- Vue 3.2.13 - 响应式框架

- Vue Router 4.0.3 - 路由管理

- Vuex 4.0.0 - 状态管理

- Bootstrap 5.1.3 - UI框架

- Canvas API - 游戏渲染

- WebSocket - 实时通信

核心目录结构:

web/src/

├── components/ # 公共组件

│ ├── GameMap.vue # 游戏地图组件

│ ├── PlayGround.vue # 游戏场地

│ ├── MatchGround.vue # 匹配界面

│ └── ResultBoard.vue # 结果展示

├── views/ # 页面组件

│ ├── pk/ # 对战相关

│ ├── user/ # 用户相关

│ ├── record/ # 对战记录

│ └── ranklist/ # 排行榜

├── store/ # Vuex状态管理

│ ├── index.js # 主store

│ ├── pk.js # 对战状态

│ ├── user.js # 用户状态

│ └── record.js # 记录状态

└── assets/scripts/ # 游戏引擎

├── AcGameObject.js # 游戏对象基类

├── GameMap.js # 游戏地图类

├── Snake.js # 贪吃蛇类

├── Wall.js # 墙体类

└── Cell.js # 单元格类

【关键技术实现】

1. 游戏引擎设计

游戏对象基类 - 实现游戏循环:

export class AcGameObject {

constructor() {

this.has\_called\_start = false;

this.timedelta = 0;

this.uuid = this.create\_uuid();

}

// 游戏主循环

step(timestamp) {

if (!this.has\_called\_start) {

this.start();

this.has\_called\_start = true;

} else {

this.timedelta = timestamp - this.last\_timestamp;

this.update();

}

this.last\_timestamp = timestamp;

}

}

2. 蛇类的核心实现

export class Snake extends AcGameObject {

constructor(info, gamemap) {

super();

this.cells = [new Cell(info.r, info.c)]; // 蛇身体

this.speed = 5; // 移动速度

this.direction = -1; // 移动方向

this.status = "idle"; // 状态：idle/move/die

}

// 检查蛇长度是否增加

check\_tail\_increasing() {

if (this.step <= 10) return true;

if (this.step % 3 === 1) return true;

return false;

}

// 移动到下一步

next\_step() {

const d = this.direction;

this.next\_cell = new Cell(

this.cells[0].r + this.dr[d],

this.cells[0].c + this.dc[d]

);

this.status = "move";

this.step++;

}

}

3. Vuex状态管理

pk.js - 对战状态管理:

export default {

state: {

status: "matching", // matching/playing

socket: null, // WebSocket连接

gamemap: null, // 游戏地图数据

gameObject: null, // 游戏对象实例

loser: "none", // 游戏结果

},

mutations: {

updateSocket(state, socket) {

state.socket = socket;

},

updateGame(state, game) {

state.gamemap = game.map;

state.a\_id = game.a\_id;

// ...更多状态更新

}

}

}

===========================================================

【后端架构 (Spring Boot)】

技术栈:

- Spring Boot 2.7.1 - 主框架

- Spring Security - 安全认证

- MyBatis-Plus 3.5.2 - ORM框架

- MySQL 8.0 - 数据库

- Redis - 缓存

- WebSocket - 实时通信

- JWT - 身份验证

- Maven - 项目管理

【核心模块设计】

1. 数据库设计

用户表:

CREATE TABLE user (

id INT AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(100) NOT NULL UNIQUE,

password VARCHAR(100) NOT NULL,

photo VARCHAR(1000),

rating INT DEFAULT 1500, -- ELO评分系统

openid VARCHAR(100), -- 第三方登录ID

INDEX idx\_rating (rating)

);

Bot表:

CREATE TABLE bot (

id INT AUTO\_INCREMENT PRIMARY KEY,

user\_id INT NOT NULL,

title VARCHAR(100) NOT NULL,

description VARCHAR(300),

content TEXT NOT NULL, -- Bot代码

createtime DATETIME,

modifytime DATETIME,

FOREIGN KEY (user\_id) REFERENCES user(id)

);

对战记录表:

CREATE TABLE record (

id INT AUTO\_INCREMENT PRIMARY KEY,

a\_id INT NOT NULL, -- 玩家A ID

a\_sx INT NOT NULL, -- 玩家A起始x坐标

a\_sy INT NOT NULL, -- 玩家A起始y坐标

b\_id INT NOT NULL, -- 玩家B ID

b\_sx INT NOT NULL,

b\_sy INT NOT NULL,

a\_steps TEXT, -- 玩家A操作序列

b\_steps TEXT, -- 玩家B操作序列

map TEXT NOT NULL, -- 地图信息

loser VARCHAR(10), -- 失败者：A/B/all

createtime DATETIME,

INDEX idx\_user (a\_id, b\_id),

INDEX idx\_time (createtime)

);

2. WebSocket实现

@Component

@ServerEndpoint("/websocket/{token}")

public class WebSocketServer {

// 存储所有连接的用户

final public static ConcurrentHashMap<Integer, WebSocketServer> users =

new ConcurrentHashMap<>();

@OnOpen

public void onOpen(Session session, @PathParam("token") String token) {

// JWT token验证

Integer userId = JwtAuthentication.getUserId(token);

this.user = userMapper.selectById(userId);

if (this.user != null) {

users.put(userId, this);

} else {

this.session.close();

}

}

@OnMessage

public void onMessage(String message, Session session) {

JSONObject data = JSONObject.parseObject(message);

String event = data.getString("event");

if ("start-matching".equals(event)) {

startMatching(data.getInteger("bot\_id"));

} else if ("move".equals(event)) {

move(data.getInteger("direction"));

}

}

// 开始游戏的静态方法

public static void startGame(Integer aId, Integer aBotId,

Integer bId, Integer bBotId) {

// 创建游戏实例

Game game = new Game(13, 14, 20, aId, botA, bId, botB);

game.createMap();

game.start();

// 通知双方玩家

JSONObject resp = new JSONObject();

resp.put("event", "start-matching");

resp.put("game", gameData);

// 发送给双方...

}

}

3. 游戏核心逻辑

public class Game extends Thread {

private final Integer rows = 13, cols = 14;

private final int[][] g; // 游戏地图

private final Player playerA, playerB;

private Integer nextStepA = null, nextStepB = null;

private ReentrantLock lock = new ReentrantLock();

// 地图生成算法

public void createMap() {

// 1. 初始化边界墙

for (int r = 0; r < rows; r++) {

g[r][0] = g[r][cols - 1] = 1;

}

for (int c = 0; c < cols; c++) {

g[0][c] = g[rows - 1][c] = 1;

}

// 2. 随机生成内部墙体（保证对称）

Random random = new Random();

for (int i = 0; i < inner\_walls\_count / 2; i++) {

int r = random.nextInt(rows);

int c = random.nextInt(cols);

// 确保两个起点可达

g[r][c] = g[rows - 1 - r][cols - 1 - c] = 1;

}

// 3. 验证连通性

if (!check\_connectivity(rows - 2, 1, 1, cols - 2)) {

createMap(); // 重新生成

}

}

@Override

public void run() {

for (int i = 0; i < 1000; i++) {

if (nextStep()) { // 获取双方操作

judge(); // 判断合法性

if (status.equals("playing")) {

sendMove(); // 广播移动

} else {

sendResult(); // 发送结果

break;

}

} else {

// 超时处理

status = "finished";

sendResult();

break;

}

}

}

}

4. JWT身份验证

@Component

public class JwtUtil {

public static final long JWT\_TTL = 60 \* 60 \* 1000L \* 24 \* 14; // 14天

public static String createJWT(String subject) {

JwtBuilder builder = getJwtBuilder(subject, null, getUUID());

return builder.compact();

}

public static Claims parseJWT(String jwt) throws Exception {

SecretKey secretKey = generalKey();

return Jwts.parserBuilder()

.setSigningKey(secretKey)

.build()

.parseClaimsJws(jwt)

.getBody();

}

}

JWT过滤器:

@Component

public class JwtAuthenticationTokenFilter extends OncePerRequestFilter {

@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain) {

String token = request.getHeader("Authorization");

if (token != null && token.startsWith("Bearer ")) {

token = token.substring(7);

try {

Claims claims = JwtUtil.parseJWT(token);

String userid = claims.getSubject();

// 设置SecurityContext...

} catch (Exception e) {

// token无效

}

}

filterChain.doFilter(request, response);

}

}

===========================================================

【微服务架构】

1. 匹配系统 (MatchingSystem)

端口: 3001

职责: 玩家匹配逻辑

@Component

public class MatchingPool extends Thread {

private static List<Player> players = new ArrayList<>();

private final ReentrantLock lock = new ReentrantLock();

// 匹配算法 - 基于ELO rating

private boolean checkMatched(Player a, Player b) {

int ratingDelta = Math.abs(a.getRating() - b.getRating());

int waitingTime = Math.min(a.getWaitingTime(), b.getWaitingTime());

return ratingDelta <= waitingTime \* 10; // 等待时间越长，匹配范围越大

}

@Override

public void run() {

while (true) {

Thread.sleep(1000); // 每秒匹配一次

lock.lock();

try {

increaseWaitingTime(); // 增加等待时间

matchPlayers(); // 执行匹配

} finally {

lock.unlock();

}

}

}

}

2. Bot运行系统 (BotRunningSystem)

端口: 3002

职责: 执行用户Bot代码

@Component

public class BotPool extends Thread {

private final Queue<Bot> bots = new LinkedList<>();

private final ReentrantLock lock = new ReentrantLock();

public void addBot(Integer userId, String botCode, String input) {

lock.lock();

try {

bots.add(new Bot(userId, botCode, input));

condition.signalAll(); // 唤醒消费者线程

} finally {

lock.unlock();

}

}

private void consume(Bot bot) {

Consumer consumer = new Consumer();

consumer.startTimeout(2000, bot); // 2秒超时

}

}

Bot代码执行器:

public class Consumer {

public void startTimeout(long timeout, Bot bot) {

// 1. 编译用户代码

// 2. 创建沙箱环境

// 3. 执行代码获取下一步操作

// 4. 通过HTTP回调返回结果

}

}

===========================================================

【技术难点与解决方案】

1. 实时同步问题

问题: 双人游戏需要严格的时序同步

解决方案:

- 后端作为权威服务器，统一管理游戏状态

- WebSocket确保低延迟通信

- 客户端只负责渲染，不做游戏逻辑判断

2. 地图生成算法

问题: 需要生成公平且连通的对称地图

解决方案:

对称生成 + 连通性检查

private boolean draw() {

// 1. 生成对称障碍物

for (int i = 0; i < inner\_walls\_count / 2; i++) {

int r = random.nextInt(rows);

int c = random.nextInt(cols);

g[r][c] = g[rows - 1 - r][cols - 1 - c] = 1;

}

// 2. DFS检查连通性

return check\_connectivity(rows - 2, 1, 1, cols - 2);

}

3. Bot代码安全执行

问题: 用户提交的代码可能包含恶意操作

解决方案:

- 独立的Bot运行微服务

- 代码执行时间限制（2秒）

- 沙箱环境隔离

- 资源使用限制

4. 并发安全

问题: 多线程环境下的数据一致性

解决方案:

使用ReentrantLock保护临界区

private final ReentrantLock lock = new ReentrantLock();

public void setNextStep(Integer nextStep) {

lock.lock();

try {

this.nextStep = nextStep;

} finally {

lock.unlock();

}

}

5. 游戏状态管理

问题: 复杂的游戏状态需要精确管理

解决方案:

- 状态机模式：idle -> move -> die

- 帧同步：客户端60fps渲染，服务端5fps逻辑更新

- 插值算法：平滑的移动动画

===========================================================

【性能优化亮点】

1. 前端优化

- Canvas渲染优化: 只重绘变化区域

- 状态管理: Vuex模块化，避免不必要的响应式更新

- 组件懒加载: 路由级别的代码分割

2. 后端优化

- 连接池管理: MyBatis-Plus连接池配置

- 缓存策略: Redis缓存用户信息和排行榜

- 异步处理: 游戏逻辑在独立线程中运行

3. 数据库优化

关键索引设计:

CREATE INDEX idx\_user\_rating ON user(rating);

CREATE INDEX idx\_record\_time ON record(createtime);

CREATE INDEX idx\_bot\_user ON bot(user\_id);

===========================================================

【部署架构】

微服务端口分配:

- 主后端 (Backend): 3000

- 匹配系统 (MatchingSystem): 3001

- Bot运行系统 (BotRunningSystem): 3002

- 前端 (Web): 8080

- MySQL: 3306

- Redis: 6379

服务间通信:

RestTemplate实现HTTP调用

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

调用匹配系统:

private void startMatching(Integer botId) {

MultiValueMap<String, String> data = new LinkedMultiValueMap<>();

data.add("user\_id", this.user.getId().toString());

data.add("rating", this.user.getRating().toString());

data.add("bot\_id", botId.toString());

restTemplate.postForObject(addPlayerUrl, data, String.class);

}

===========================================================

【核心算法详解】

1. ELO评分系统

胜负后评分更新:

private void updateRating() {

Integer ratingA = userMapper.selectById(playerA.getId()).getRating();

Integer ratingB = userMapper.selectById(playerB.getId()).getRating();

if ("A".equals(loser)) {

ratingA -= 2; // 失败者扣分

ratingB += 5; // 胜利者加分

} else if ("B".equals(loser)) {

ratingA += 5;

ratingB -= 2;

}

// 平局不变分

}

2. 蛇长增长规律

check\_tail\_increasing() {

if (this.step <= 10) return true; // 前10步必增长

if (this.step % 3 === 1) return true; // 之后每3步增长1次

return false;

}

3. 碰撞检测算法

private boolean check\_valid(List<Cell> cellsA, List<Cell> cellsB) {

int n = cellsA.size();

Cell cell = cellsA.get(n - 1); // 新蛇头位置

// 检查是否撞墙

if (g[cell.x][cell.y] == 1) return false;

// 检查是否撞自己

for (int i = 0; i < n - 1; i++) {

if (cellsA.get(i).x == cell.x && cellsA.get(i).y == cell.y)

return false;

}

// 检查是否撞对手

for (int i = 0; i < cellsB.size() - 1; i++) {

if (cellsB.get(i).x == cell.x && cellsB.get(i).y == cell.y)

return false;

}

return true;

}

===========================================================

【开发环境配置】

前端环境:

{

"dependencies": {

"vue": "^3.2.13",

"vue-router": "^4.0.3",

"vuex": "^4.0.0",

"bootstrap": "^5.1.3",

"vue3-ace-editor": "^2.2.2" // 代码编辑器

}

}

后端环境:

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-websocket</artifactId>

</dependency>

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-boot-starter</artifactId>

<version>3.5.2</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version>

</dependency>

</dependencies>

===========================================================

【面试重点知识点总结】

1. 微服务架构

- 服务拆分原则: 按业务功能划分（匹配、Bot执行、主业务）

- 服务间通信: RestTemplate HTTP调用

- 数据一致性: 最终一致性，通过消息传递同步状态

2. 并发编程

- ReentrantLock: 保护共享资源

- Condition: 线程间协调

- ConcurrentHashMap: 线程安全的Map实现

- 线程池: 异步任务处理

3. 网络编程

- WebSocket: 全双工实时通信

- HTTP: RESTful API设计

- JSON: 数据交换格式

4. 数据库设计

- 索引优化: 根据查询模式设计索引

- 外键约束: 保证数据完整性

- 分页查询: 大数据量处理

5. 前端技术

- Vue3 Composition API: 更好的逻辑复用

- Vuex状态管理: 组件间状态共享

- Canvas API: 2D图形渲染

- WebSocket客户端: 实时数据接收

6. 安全机制

- JWT Token: 无状态身份验证

- Spring Security: 权限控制

- CORS配置: 跨域资源共享

- 代码沙箱: 安全执行用户代码

7. 算法设计

- 地图生成: 随机算法 + 连通性检查

- 匹配算法: 基于ELO rating的公平匹配

- 碰撞检测: 高效的几何计算

- 路径验证: DFS连通性检查

===========================================================

【项目亮点总结】

1. 完整的微服务架构: 三个独立服务，职责清晰分离

2. 实时游戏引擎: 基于Canvas的流畅游戏体验

3. 智能匹配系统: ELO评分 + 等待时间的公平匹配

4. 安全的代码执行: 沙箱环境执行用户Bot代码

5. 高并发处理: 多线程 + 锁机制保证数据一致性

6. 现代化前端: Vue3 + Composition API + 模块化状态管理

7. 完善的用户系统: JWT认证 + 第三方登录集成

8. 可扩展架构: 微服务设计支持横向扩展

这个项目涵盖了前端开发、后端开发、数据库设计、微服务架构、实时通信、

游戏开发、算法设计等多个技术领域，是一个非常适合展示全栈开发能力的

综合性项目。

===========================================================

文档创建时间: 2024年

项目技术栈: Vue3 + Spring Boot + MySQL + Redis + WebSocket

微服务架构: Backend + MatchingSystem + BotRunningSystem

===========================================================