James 项目 - NBA 球星数据展示系统

1. 项目概述

James 是一个基于 Spring Boot + Redis + Kafka + MySQL 技术栈的 NBA 球星数据展示系统，主要用于展示 2000 年 - 2010 年的 NBA 球星数据和比赛数据，同时练习后端开发能力，包括进程并发安全、数据库高并发处理、数据库索引设计等核心技能。

2. 需求分析

2.1 功能需求

2.1.1 数据存储需求

- **球员基本信息**：存储球员的基本资料，包括姓名、年龄、身高、体重、位置、所属球队等
- **球员统计数据**：存储球员的比赛统计数据，包括得分、篮板、助攻、抢断、盖帽等
- **比赛数据**：存储比赛的基本信息，包括比赛日期、对阵双方、比分、比赛结果等
- **球队信息**：存储球队的基本信息，包括球队名称、所在城市、成立时间等

2.1.2 数据查询需求

- 球员信息查询：根据球员姓名、ID、位置等条件查询球员信息
- 球员统计查询：查询球员的赛季统计、生涯统计等数据
- 比赛数据查询：查询特定比赛的详细数据
- 球队信息查询：查询球队的基本信息和历史数据

2.1.3 数据展示需求

- 球员数据可视化展示
- 比赛数据统计分析

- 数据排行榜展示
- 历史数据对比分析

2.2 非功能需求

2.2.1 性能需求

- 系统响应时间：页面加载时间 < 2 秒
- 并发处理能力：支持 1000 + 并发用户访问
- 数据查询性能：复杂查询响应时间 < 500ms

2.2.2 可靠性需求

- 系统可用性：99.9%
- 数据一致性：最终一致性
- 故障恢复：支持快速故障恢复

2.2.3 安全性需求

- 用户认证与授权
- 数据加密传输
- 防 SQL 注入
- 防 XSS 攻击

3. 系统架构设计

3.1 整体架构

用户层 → 前端展示层 → API网关层 → 业务服务层 → 数据访问层 → 数据存储层

3.2 技术架构

3.2.1 前端技术栈

- HTML5 + CSS3 + JavaScript
- Vue.js 或 React.js
- Element UI 或 Ant Design
- ECharts 数据可视化

3.2.2 后端技术栈

- Spring Boot 2.7.x
- Spring Data JPA
- Spring Security
- Spring Kafka
- Redis 6.x
- MySQL 8.0
- Kafka 3.x

3.3 分层架构设计

3.3.1 表现层（Controller）

- 处理 HTTP 请求
- 参数校验
- 响应结果封装
- 异常统一处理

3.3.2 业务逻辑层（Service）

- 核心业务逻辑处理
- 事务管理
- 并发控制

- 业务规则校验

3.3.3 数据访问层（Repository）

- 数据库操作
- 缓存操作
- 消息发送
- 数据转换

3.3.4 数据存储层

- MySQL：存储结构化数据
- Redis：缓存热点数据
- Kafka：处理异步消息

4. 数据库设计

4.1 数据库表结构设计

4.1.1 球员表（players）

```
CREATE TABLE players (
  id BIGINT PRIMARY KEY AUTO_INCREMENT,
  player_id VARCHAR(50) NOT NULL COMMENT '球员ID',
  full_name VARCHAR(100) NOT NULL COMMENT '球员全名',
  first_name VARCHAR(50) NOT NULL COMMENT '名字',
  last_name VARCHAR(50) NOT NULL COMMENT '姓氏',
  position VARCHAR(10) COMMENT '位置',
  height DECIMAL(5,2) COMMENT '身高(米)',
  weight DECIMAL(5,2) COMMENT '体重(公斤)',
  birth_date DATE COMMENT '出生日期',
  draft_year INT COMMENT '选秀年份',
  draft_round INT COMMENT '选秀轮次',
  draft_pick INT COMMENT '选秀顺位',
```

```
    team_id VARCHAR(50) COMMENT '所属球队ID',
    is_active BOOLEAN DEFAULT FALSE COMMENT '是否现役',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
TIMESTAMP,
    UNIQUE KEY uk_player_id (player_id),
    INDEX idx_team_id (team_id),
    INDEX idx_position (position)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='球员基本信息表';
```

4.1.2 球队表（teams）

```
CREATE TABLE teams (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    team_id VARCHAR(50) NOT NULL COMMENT '球队ID',
    team_name VARCHAR(100) NOT NULL COMMENT '球队名称',
    city VARCHAR(50) COMMENT '所在城市',
    conference VARCHAR(20) COMMENT '联盟',
    division VARCHAR(20) COMMENT '分区',
    founded_year INT COMMENT '成立年份',
    arena VARCHAR(100) COMMENT '主场球馆',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
TIMESTAMP,
    UNIQUE KEY uk_team_id (team_id),
    INDEX idx_conference (conference),
    INDEX idx_division (division)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='球队信息表';
```

4.1.3 赛季统计数据表（season_stats）

```
CREATE TABLE season_stats (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    player_id VARCHAR(50) NOT NULL COMMENT '球员ID',
    season_year VARCHAR(10) NOT NULL COMMENT '赛季年份',
    team_id VARCHAR(50) COMMENT '所属球队ID',
    gp INT COMMENT '出场次数',
    gs INT COMMENT '首发次数',
```

```
    min DECIMAL(6,2) COMMENT '出场时间',
    fgm INT COMMENT '投篮命中数',
    fga INT COMMENT '投篮尝试数',
    fg_pct DECIMAL(5,3) COMMENT '投篮命中率',
    fg3m INT COMMENT '三分命中数',
    fg3a INT COMMENT '三分尝试数',
    fg3_pct DECIMAL(5,3) COMMENT '三分命中率',
    ftm INT COMMENT '罚球命中数',
    fta INT COMMENT '罚球尝试数',
    ft_pct DECIMAL(5,3) COMMENT '罚球命中率',
    oreb INT COMMENT '进攻篮板',
    dreb INT COMMENT '防守篮板',
    reb INT COMMENT '总篮板',
    ast INT COMMENT '助攻',
    stl INT COMMENT '抢断',
    blk INT COMMENT '盖帽',
    tov INT COMMENT '失误',
    pf INT COMMENT '犯规',
    pts INT COMMENT '得分',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
TIMESTAMP,
    UNIQUE KEY uk_player_season (player_id, season_year),
    INDEX idx_season_year (season_year),
    INDEX idx_team_id (team_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='球员赛季统计表';
```

4.1.4 比赛表（games）

```
CREATE TABLE games (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    game_id VARCHAR(50) NOT NULL COMMENT '比赛ID',
    game_date DATE NOT NULL COMMENT '比赛日期',
    home_team_id VARCHAR(50) NOT NULL COMMENT '主场球队ID',
    away_team_id VARCHAR(50) NOT NULL COMMENT '客场球队ID',
    home_team_score INT NOT NULL COMMENT '主场球队得分',
    away_team_score INT NOT NULL COMMENT '客场球队得分',
    season_year VARCHAR(10) NOT NULL COMMENT '赛季年份',
    game_type VARCHAR(20) COMMENT '比赛类型',
```

```
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
TIMESTAMP,
    UNIQUE KEY uk_game_id (game_id),
    INDEX idx_game_date (game_date),
    INDEX idx_season_year (season_year),
    INDEX idx_home_team (home_team_id),
    INDEX idx_away_team (away_team_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='比赛信息表';
```

4.1.5 比赛详情表（game_details）

```
CREATE TABLE game_details (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    game_id VARCHAR(50) NOT NULL COMMENT '比赛ID',
    player_id VARCHAR(50) NOT NULL COMMENT '球员ID',
    team_id VARCHAR(50) NOT NULL COMMENT '球队ID',
    is_home BOOLEAN NOT NULL COMMENT '是否主场',
    gp INT COMMENT '出场次数',
    gs INT COMMENT '首发次数',
    min DECIMAL(6,2) COMMENT '出场时间',
    fgm INT COMMENT '投篮命中数',
    fga INT COMMENT '投篮尝试数',
    fg_pct DECIMAL(5,3) COMMENT '投篮命中率',
    fg3m INT COMMENT '三分命中数',
    fg3a INT COMMENT '三分尝试数',
    fg3_pct DECIMAL(5,3) COMMENT '三分命中率',
    ftm INT COMMENT '罚球命中数',
    fta INT COMMENT '罚球尝试数',
    ft_pct DECIMAL(5,3) COMMENT '罚球命中率',
    oreb INT COMMENT '进攻篮板',
    dreb INT COMMENT '防守篮板',
    reb INT COMMENT '总篮板',
    ast INT COMMENT '助攻',
    stl INT COMMENT '抢断',
    blk INT COMMENT '盖帽',
    tov INT COMMENT '失误',
    pf INT COMMENT '犯规',
    pts INT COMMENT '得分',
```

```
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
TIMESTAMP,
    UNIQUE KEY uk_game_player (game_id, player_id),
    INDEX idx_game_id (game_id),
    INDEX idx_player_id (player_id),
    INDEX idx_team_id (team_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='比赛详细数据表';
```

4.2 数据库索引设计

4.2.1 主键索引

- 所有表的 id 字段作为主键
- 使用自增主键保证性能

4.2.2 唯一索引

- player_id：球员唯一标识
- team_id：球队唯一标识
- game_id：比赛唯一标识
- 组合唯一索引：player_id + season_year（赛季统计）

4.2.3 普通索引

- 频繁查询的字段：position、team_id、season_year、game_date 等
- 外键关联字段：team_id、player_id 等

4.2.4 索引优化策略

1. **覆盖索引**：为常用查询创建覆盖索引，避免回表查询
2. **联合索引**：合理设计联合索引，遵循最左前缀原则
3. **索引选择性**：只对选择性高的字段创建索引
4. **索引维护**：定期清理冗余索引和重复索引

5. 并发安全设计

5.1 分布式锁实现

5.1.1 Redis 分布式锁

```
@Component
public class RedisDistributedLock {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    private static final String LOCK_PREFIX = "lock:";
    private static final int DEFAULT_EXPIRE_TIME = 30; // 30秒

    public boolean tryLock(String key, String value, int expireTime) {
        String lockKey = LOCK_PREFIX + key;
        Boolean success = redisTemplate.opsForValue()
                .setIfAbsent(lockKey, value, expireTime, TimeUnit.SECONDS);
        return Boolean.TRUE.equals(success);
    }

    public void releaseLock(String key, String value) {
        String lockKey = LOCK_PREFIX + key;
        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1])
 else return 0 end";
        redisTemplate.execute(new DefaultRedisScript<>(script, Long.class),
                Collections.singletonList(lockKey), value);
    }
}
```

5.1.2 使用示例

```
@Service
public class PlayerService {
```

```
    @Autowired
    private RedisDistributedLock distributedLock;

    public void updatePlayerStats(String playerId, PlayerStats stats) {
        String lockKey = "player:" + playerId;
        String lockValue = UUID.randomUUID().toString();

        try {
            if (distributedLock.tryLock(lockKey, lockValue, 30)) {
                // 执行业务逻辑
                updateStats(playerId, stats);
            } else {
                throw new BusinessException("操作太频繁，请稍后重试");
            }
        } finally {
            distributedLock.releaseLock(lockKey, lockValue);
        }
    }
}
```

5.2 乐观锁实现

5.2.1 版本号机制

```
@Entity
@Table(name = "players")
public class Player {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "player_id")
    private String playerId;

    // 其他字段...

    @Version
```

```
    private Long version;

    // getter和setter
}
```

5.2.2 乐观锁更新

```
@Service
public class PlayerService {

    @Autowired
    private PlayerRepository playerRepository;

    @Transactional
    public Player updatePlayer(Player player) {
        try {
            return playerRepository.save(player);
        } catch (ObjectOptimisticLockingFailureException e) {
            throw new BusinessException("数据已被其他用户修改，请刷新后重试");
        }
    }
}
```

6. 高并发处理设计

6.1 缓存策略

6.1.1 Redis 缓存设计

```
@Service
public class PlayerCacheService {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;
```

```
    @Autowired
    private PlayerRepository playerRepository;

    private static final String PLAYER_CACHE_PREFIX = "player:";
    private static final int PLAYER_CACHE_EXPIRE = 3600; // 1小时

    public Player getPlayerById(String playerId) {
        String cacheKey = PLAYER_CACHE_PREFIX + playerId;
        Player player = (Player) redisTemplate.opsForValue().get(cacheKey);

        if (player == null) {
            player = playerRepository.findByPlayerId(playerId);
            if (player != null) {
                redisTemplate.opsForValue().set(cacheKey, player, PLAYER_CACHE_EXPIRE, TimeUnit
.SECONDS);
            }
        }

        return player;
    }

    public void updatePlayerCache(String playerId, Player player) {
        String cacheKey = PLAYER_CACHE_PREFIX + playerId;
        redisTemplate.opsForValue().set(cacheKey, player, PLAYER_CACHE_EXPIRE, TimeUnit.
SECONDS);
    }

    public void deletePlayerCache(String playerId) {
        String cacheKey = PLAYER_CACHE_PREFIX + playerId;
        redisTemplate.delete(cacheKey);
    }
}
```

6.1.2 缓存穿透防护

```
@Service
public class CachePenetrationGuard {

    @Autowired
```

```
    private RedisTemplate<String, Object> redisTemplate;

    private static final String NULL_VALUE = "NULL";
    private static final int NULL_CACHE_EXPIRE = 60; // 1分钟

    public <T> T getWithPenetrationGuard(String key, Function<String, T> loader) {
        Object value = redisTemplate.opsForValue().get(key);

        if (value != null) {
            if (NULL_VALUE.equals(value)) {
                return null;
            }
            return (T) value;
        }

        T result = loader.apply(key);
        if (result == null) {
            redisTemplate.opsForValue().set(key, NULL_VALUE, NULL_CACHE_EXPIRE, TimeUnit.
SECONDS);
        } else {
            redisTemplate.opsForValue().set(key, result, 3600, TimeUnit.SECONDS);
        }

        return result;
    }
}
```

6.2 消息队列异步处理

6.2.1 Kafka 生产者

```
@Configuration
public class KafkaProducerConfig {

 @Value("${spring.kafka.bootstrap-servers}")
 private String bootstrapServers;

 @Bean
 public ProducerFactory<String, String> producerFactory() {
 Map<String, Object> configProps = new HashMap<>();
```

```
 configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);
 configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
 configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
 return new DefaultKafkaProducerFactory<>(configProps);
}

 @Bean
 public KafkaTemplate<String, String> kafkaTemplate() {
 return new KafkaTemplate<>(producerFactory());
}
}
```

6.2.2 异步消息发送

```
@Service
public class PlayerStatsService {

   @Autowired
   private KafkaTemplate<String, String> kafkaTemplate;

   private static final String STATS_UPDATE_TOPIC = "player-stats-update";

   public void updatePlayerStatsAsync(String playerId, PlayerStats stats) {
      // 同步更新缓存
      updatePlayerCache(playerId, stats);

      // 异步更新数据库
      String message = JSON.toJSONString(new StatsUpdateMessage(playerId, stats));
      kafkaTemplate.send(STATS_UPDATE_TOPIC, playerId, message);
   }
}
```

6.2.3 Kafka 消费者

```
@Service
public class StatsUpdateConsumer {
```

```
    @Autowired
    private PlayerStatsRepository statsRepository;

    @KafkaListener(topics = "player-stats-update", groupId = "stats-update-group")
    public void consumeStatsUpdate(String message) {
        StatsUpdateMessage updateMessage = JSON.parseObject(message, StatsUpdateMessage
.class);

        try {
            // 更新数据库
            updateDatabaseStats(updateMessage.getPlayerId(), updateMessage.getStats());
        } catch (Exception e) {
            // 处理消费失败，可发送到死信队列
            log.error("更新球员统计数据失败", e);
        }
    }
}
```

7. 前端展示设计

7.1 页面结构

7.1.1 首页

- 热门球员展示
- 最新比赛数据
- 数据排行榜
- 搜索功能

7.1.2 球员详情页

- 球员基本信息
- 生涯统计数据
- 赛季表现趋势
- 比赛记录

7.1.3 比赛详情页

- 比赛基本信息
- 双方球队数据
- 球员表现统计
- 比赛亮点

7.1.4 数据统计页

- 球员排行榜
- 球队排行榜
- 历史数据对比
- 数据可视化图表

7.2 数据可视化

7.2.1 球员得分趋势图

```
// 使用ECharts实现
const scoreChart = echarts.init(document.getElementById('scoreChart'));
const option = {
  title: {
    text: '球员得分趋势'
  },
  tooltip: {
    trigger: 'axis'
  },
  legend: {
    data: ['得分', '篮板', '助攻']
  },
  xAxis: {
    type: 'category',
    data: ['2000', '2001', '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010']
  },
  yAxis: {
```

```
    type: 'value'
  },
  series: [
    {
      name: '得分',
      type: 'line',
      data: [28.4, 29.7, 31.4, 28.7, 30.0, 31.4, 27.3, 28.4, 26.8, 25.3, 27.1]
    },
    {
      name: '篮板',
      type: 'line',
      data: [7.3, 7.6, 7.9, 7.6, 7.9, 7.8, 6.7, 7.6, 7.3, 7.2, 7.0]
    },
    {
      name: '助攻',
      type: 'line',
      data: [6.4, 6.1, 6.0, 6.0, 6.9, 6.6, 6.0, 7.2, 7.1, 7.0, 7.3]
    }
  ]
};
scoreChart.setOption(option);
```

7.2.2 球员能力雷达图

```
const radarChart = echarts.init(document.getElementById('radarChart'));
const option = {
  title: {
    text: '球员能力雷达图'
  },
  tooltip: {},
  radar: {
    indicator: [
      { name: '得分', max: 100 },
      { name: '篮板', max: 100 },
      { name: '助攻', max: 100 },
      { name: '防守', max: 100 },
      { name: '三分', max: 100 },
      { name: '罚球', max: 100 }
    ]
```

```
  },
  series: [{
    name: '球员能力',
    type: 'radar',
    data: [
      {
        value: [95, 80, 85, 75, 65, 88],
        name: '球员A'
      }
    ]
  }]
};
radarChart.setOption(option);
```

8. 核心代码框架

8.1 Spring Boot 应用入口

```
@SpringBootApplication
@EnableCaching
@EnableKafka
public class JamesApplication {
 public static void main(String[] args) {
 SpringApplication.run(JamesApplication.class, args);
}
}
```

8.2 球员 Controller

```
@RestController
@RequestMapping("/api/players")
public class PlayerController {

 @Autowired
 private PlayerService playerService;

 @GetMapping("/{playerId}")
 public Result<PlayerDTO> getPlayerById(@PathVariable String playerId) {
 PlayerDTO player = playerService.getPlayerById(playerId);
```

```
 return Result.success(player);
}

 @GetMapping
 public Result<PageResult<PlayerDTO>> getPlayers(
 @RequestParam(required = false) String name,
 @RequestParam(required = false) String position,
 @RequestParam(defaultValue = "1") int page,
 @RequestParam(defaultValue = "10") int size) {

 PageResult<PlayerDTO> result = playerService.getPlayers(name, position
, page, size);
 return Result.success(result);
}

 @GetMapping("/{playerId}/stats")
 public Result<PlayerStatsDTO> getPlayerStats(@PathVariable String playerId
) {
 PlayerStatsDTO stats = playerService.getPlayerStats(playerId);
 return Result.success(stats);
}

 @GetMapping("/{playerId}/games")
 public Result<PageResult<GameDTO>> getPlayerGames(
 @PathVariable String playerId,
 @RequestParam(defaultValue = "1") int page,
 @RequestParam(defaultValue = "10") int size) {

 PageResult<GameDTO> games = playerService.getPlayerGames(playerId, page
, size);
 return Result.success(games);
}
}
```

8.3 球员 Service

```
@Service
public class PlayerServiceImpl implements PlayerService {

   @Autowired
   private PlayerRepository playerRepository;

   @Autowired
   private PlayerStatsRepository statsRepository;
```

```
@Autowired
private GameRepository gameRepository;

@Autowired
private PlayerCacheService cacheService;

@Override
public PlayerDTO getPlayerById(String playerId) {
    Player player = cacheService.getPlayerById(playerId);
    if (player == null) {
        throw new ResourceNotFoundException("球员不存在");
    }
    return PlayerConverter.toDTO(player);
}

@Override
public PageResult<PlayerDTO> getPlayers(String name, String position, int page, int size) {
    Pageable pageable = PageRequest.of(page - 1, size, Sort.by("fullName").ascending());
    Page<Player> playerPage = playerRepository.findPlayers(name, position, pageable);

    List<PlayerDTO> content = playerPage.getContent().stream()
            .map(PlayerConverter::toDTO)
            .collect(Collectors.toList());

    return new PageResult<>(
            content,
            playerPage.getTotalElements(),
            playerPage.getTotalPages(),
            page,
            size
    );
}

@Override
public PlayerStatsDTO getPlayerStats(String playerId) {
    List<SeasonStats> seasonStats = statsRepository.findByPlayerId(playerId);
    if (seasonStats.isEmpty()) {
        throw new ResourceNotFoundException("球员统计数据不存在");
    }

    PlayerStatsDTO statsDTO = new PlayerStatsDTO();
```

```
        statsDTO.setPlayerId(playerId);
        statsDTO.setSeasonStats(seasonStats.stream()
                .map(StatsConverter::toDTO)
                .collect(Collectors.toList()));

        // 计算生涯总数据
        calculateCareerStats(statsDTO, seasonStats);

        return statsDTO;
    }

    private void calculateCareerStats(PlayerStatsDTO statsDTO, List<SeasonStats> seasonStats
) {
        CareerStats careerStats = new CareerStats();

        for (SeasonStats stats : seasonStats) {
            careerStats.setGp(careerStats.getGp() + stats.getGp());
            careerStats.setGs(careerStats.getGs() + stats.getGs());
            careerStats.setMin(careerStats.getMin() + stats.getMin());
            careerStats.setFgm(careerStats.getFgm() + stats.getFgm());
            careerStats.setFga(careerStats.getFga() + stats.getFga());
            careerStats.setFg3m(careerStats.getFg3m() + stats.getFg3m());
            careerStats.setFg3a(careerStats.getFg3a() + stats.getFg3a());
            careerStats.setFtm(careerStats.getFtm() + stats.getFtm());
            careerStats.setFta(careerStats.getFta() + stats.getFta());
            careerStats.setOreb(careerStats.getOreb() + stats.getOreb());
            careerStats.setDreb(careerStats.getDreb() + stats.getDreb());
            careerStats.setReb(careerStats.getReb() + stats.getReb());
            careerStats.setAst(careerStats.getAst() + stats.getAst());
            careerStats.setStl(careerStats.getStl() + stats.getStl());
            careerStats.setBlk(careerStats.getBlk() + stats.getBlk());
            careerStats.setTov(careerStats.getTov() + stats.getTov());
            careerStats.setPf(careerStats.getPf() + stats.getPf());
            careerStats.setPts(careerStats.getPts() + stats.getPts());
        }

        // 计算命中率
        if (careerStats.getFga() > 0) {
            careerStats.setFgPct((double) careerStats.getFgm() / careerStats.getFga());
        }
        if (careerStats.getFg3a() > 0) {
```

```
            careerStats.setFg3Pct((double) careerStats.getFg3m() / careerStats.getFg3a());
        }
        if (careerStats.getFta() > 0) {
            careerStats.setFtPct((double) careerStats.getFtm() / careerStats.getFta());
        }

        statsDTO.setCareerStats(careerStats);
    }
}
```

8.4 数据访问层

```
@Repository
public interface PlayerRepository extends JpaRepository<Player, Long> {

 Optional<Player> findByPlayerId(String playerId);

 @Query("SELECT p FROM Player p WHERE " +
"(:name IS NULL OR p.fullName LIKE %:name%) AND " +
"(:position IS NULL OR p.position = :position)")
 Page<Player> findPlayers(@Param("name") String name,
 @Param("position") String position,
 Pageable pageable);
}
```

8.5 异常处理

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public Result<?> handleResourceNotFound(ResourceNotFoundException e) {
        return Result.error(404, e.getMessage());
    }

    @ExceptionHandler(BusinessException.class)
    public Result<?> handleBusinessException(BusinessException e) {
        return Result.error(400, e.getMessage());
    }
```

```
    @ExceptionHandler(Exception.class)
    public Result<?> handleException(Exception e) {
        log.error("系统异常", e);
        return Result.error(500, "系统内部错误，请联系管理员");
    }
}
```

9. 部署架构

9.1 开发环境

- 本地开发：IntelliJ IDEA + MySQL + Redis + Kafka
- 测试环境：Docker Compose 部署
- 生产环境：Kubernetes 集群

9.2 Docker Compose 配置

```
version: '3'
services:
  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: james
    ports:
      - "3306:3306"
    volumes:
      - mysql-data:/var/lib/mysql

  redis:
    image: redis:6
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data

  kafka:
    image: confluentinc/cp-kafka:latest
    environment:
      KAFKA_BROKER_ID: 1
```

```
  KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
  KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
  ports:
  - "9092:9092"
  depends_on:
  - zookeeper

  zookeeper:
  image: confluentinc/cp-zookeeper:latest
  environment:
  ZOOKEEPER_CLIENT_PORT: 2181
  ports:
  - "2181:2181"

  app:
  build: .
  ports:
  - "8080:8080"
  depends_on:
  - mysql
  - redis
  - kafka
  environment:
  SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/james
  SPRING_DATASOURCE_USERNAME: root
  SPRING_DATASOURCE_PASSWORD: root
  SPRING_REDIS_HOST: redis
  SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
volumes:
  mysql-data:
  redis-data:
```

10. 总结

James 项目是一个完整的 NBA 球星数据展示系统，通过 Spring Boot + Redis + Kafka + MySQL 技术栈实现了以下核心功能：

1. **数据存储与管理**：设计了合理的数据库表结构，支持球员信息、比赛数据、统计数据的存储和管理
2. **高并发处理**：通过 Redis 缓存、Kafka 异步处理、分布式锁等技术保证系统在高并发场景下的性能和稳定性
3. **并发安全**：实现了分布式锁和乐观锁机制，保证数据在并发访问下的一致性
4. **数据库优化**：设计了合理的索引策略，包括主键索引、唯一索引、普通索引和联合索引，提升查询性能
5. **前端展示**：提供了丰富的数据可视化展示，包括趋势图、雷达图等

该项目不仅满足了 NBA 球星数据展示的业务需求，同时也为后端开发能力的练习提供了很好的实践场景，涵盖了现代分布式系统开发的核心技术和最佳实践。

> （注：文档部分内容可能由 AI 生成）