典型功能流程及完整函数调用链说明

本说明详细梳理 `MainService.java`、`PlayerService.java` 及 `CardService.java` 内部的核心业务流程，逐步列举各流程中的每一次函数调用，突出系统设计的严谨性、逻辑性和面向对象的分层思想。整体流程涵盖了玩家连接、房间管理、回合对战、卡牌操作与BUFF结算等所有典型场景。

---

1. 玩家连接与房间管理全过程

1.1 WebSocket连接建立

- **入口函数：**
  - `MainService.afterConnectionEstablished(WebSocketSession session)`
    - 新增会话到`sessions`列表。
    - 生成连接确认消息并通过`session.sendMessage`返回客户端。

1.2 创建房间

- **前端请求：**
  - `{"type":"createRoom","room":{"uid":"1"}}`
- **服务端处理链条：**
  1. `MainService.handleTextMessage(session, message)`
     （解析type为`createRoom`，路由至下一级）
  2. `MainService.handleCreateRoomMessage(session, messageData)`
     - 解析用户ID
     - 关联`userSessions.put(uid, session)`
     - 调用业务创建房间：
  3. `MainService.CreateRoom(int userId)`
     - `InitService.CreateRoom(userId) → Room`（创建房间对象）
     - 房间对象加入`roomMap`
     - 返回房间ID
  4. 构造结果，`session.sendMessage`返回前端

1.3 加入房间

- **前端请求：**
  - `{"type":"joinRoom","room":{"roomId":"xxx","uid":"2"}}`
- **服务端处理链条：**
  1. `MainService.handleTextMessage(session, message)`
     （解析type为`joinRoom`，路由至下一级）
  2. `MainService.handleJoinRoomMessage(session, messageData)`
     - 解析房间ID和用户ID
     - 关联`userSessions.put(uid, session)`
     - 调用业务加入房间：
  3. `MainService.JoinRoom(String roomId, int userId)`
     - 判断房间是否存在
     - `InitService.joinRoom(room, userId)`（加入房间，更新房主/加入者字段）

- 计算房内真实玩家数量（Set去重），更新`roomNum`
- 构造房间状态消息，广播至所有房内用户和全体sessions
- 返回结果字符串

4. 解析返回内容，组合响应，`session.sendMessage`返回前端

---

2. 游戏对局与回合流程

2.1 开始游戏

- **前端请求：**
  - `{"type":"startGame","room":{"roomId":"xxx","role1":"A","role2":"B"}}`
- **服务端处理链条：**
  1. `MainService.handleTextMessage(session, message)`
     （解析type为`startGame`，路由至下一级）
  2. `MainService.handleStartGameMessage(session, messageData)`
     - 解析房间ID和角色
     - 调用业务开始游戏：
  3. `MainService.StartGame(roomId, role1, role2)`
     - 查询房间对象
     - 判定人数是否满足
     - 使用`InitService.init(roomId, uid1, role1)`和`init(roomId, uid2, role2)`初始化两个`PlayerAgainst`
     - 两对象分别放入`playerAgainstMap`
     - 日志记录并返回结果字符串
  4. 结果返回前端

2.2 回合开始同步

- **前端请求：**
  - `{"type":"RoundBegin","room":{"roomId":"xxx","uid":"1"}}`
- **服务端处理链条：**
  1. `MainService.handleTextMessage(session, message)`
     （解析type为`RoundBegin`，路由至下一级）
  2. `MainService.handleRoundBeginMessage(session, messageData)`
     - 解析房间ID/用户ID
     - 检查房间和玩家对象有效性
     - 在`roundBeginSyncMap`登记当前轮次已准备玩家uid
     - 若两人都准备：
       - 调用`playerService.BeginService(player1, player2, null, null, false)`
         - 分别遍历两个玩家的`statusesBegin`，对带有`next`的BUFF调用`MainNextBuffService`，普通BUFF调用`MainBuffService`，如需发放金币/抽牌也在此阶段处理
       - 构造回合开始消息，推送给两位玩家
     - 否则仅回复“等待对手”消息

2.3 回合出牌及结算

- **前端请求：**
  - `{"type":"RoundEnd","room":{"roomId":"xxx","uid":"1","cardList1":["spring", "fire", "bamboo"]}}`
- **服务端处理链条：**
  1. `MainService.handleTextMessage(session, message)`
     （解析type为`RoundEnd`，路由至下一级）
  2. `MainService.handleRoundEndMessage(session, messageData)`
     - 解析房间ID、用户ID、出牌列表
     - 查找房间对象、`PlayerAgainst`
     - 记录本玩家本轮出牌（`roundEndDTOMapHistory`），如对手尚未提交则发送"等待对手"消息并返回
     - 若两人均已提交：
       - 取出双方出牌列表
       - `playerService.GetList(cardList1)`与`playerService.GetList(cardList2)`：字符串转卡牌对象List，补足3张
       - 调用`playerService.MainService(player1, player2, list1, list2, room)`
         1. **BeginService**
            - 处理金币发放、回合开始BUFF
         2. **DiscardPlayersCards**
            - 双方本回合出的三张卡依次通过`CardService.DiscardCard`弃掉
         3. **sortCardBattleByPriority**
            - 按卡牌类型优先级排序
         4. **MainOpService循环三次**
            - 对每张卡牌，查找`cardActions`对应实现（如`Action_Spring`、`Action_Fire`等），执行效果
            - 过程涉及卡牌效果、状态添加、数值变化、Buff触发等
         5. **EndService**
            - `updateAndCleanStatuses`减少所有BUFF持续时间并处理过期
            - 针对`statusesEnd`的judge型BUFF调用`MainBuffService`，其余转到`statusesBegin`
            - 处理特殊Buff（如壮志难酬等特殊逻辑）
            - 重置玩家临时状态
       - 回合后统计双方HP判断胜负，构造结算消息和可能的游戏结束消息，推送给两玩家
       - 若游戏未结束且符合条件，自动为双方加护盾，轮数+1，并自动发起新回合（BeginService、推送回合开始消息）

---

3. 卡牌合成与弃牌流程

3.1 卡牌合成

- **前端请求：**
  - `{"type":"synthesize","room":{"uid":"5","cardA":"spring","cardB":"fire","cardC":"bamboo"}}`
- **服务端处理链条：**
  1. `MainService.handleTextMessage(session, message)`
     （type为`synthesize`，路由至下一级）

2. `MainService.handleSynthesizeMessage(session, messageData)`
    - 解析uid和三张卡名
    - 调用`MainService.SynthesizeCards(uid, a, b, c)`
        - `PlayerService.SynthesizeABC(playerAgainst, a, b, c)`
            1. `CardService.DiscardCard`弃掉a
            2. `CardService.DiscardCard`弃掉b
            3. `CardService.GetCardByName`获得c
    - 构造合成结果返回前端

3.2 弃牌获利

- **前端请求：**
    - `{"type":"discardCard","room":{"uid":1,"card":"spring","money":2}}`
- **服务端处理链条：**
    1. `MainService.handleTextMessage(session, message)`
       （type为`discardCard`，路由至下一级）
    2. `MainService.handleDiscardCard(session, messageData)`
        - 解析uid、卡牌名、金币
        - `PlayerService.AddCoins(playerAgainst, money)` 加金币
        - `playerAgainst.setCards(CardService.DiscardCard(...))`弃掉指定卡牌
        - 构造结果返回前端

---

4. 卡牌与BUFF效果的详细结算链

4.1 单张卡牌效果

- 在 `PlayerService.MainOpService(user, target, cardBattle)`：
    - 查找`cardActions`表中对应方法（如cardName="spring"则为`Action_Spring`）
        - 以`Action_Spring`为例：
            - 恢复血量 `AddHP(user,2)`
            - 添加判定状态 `AddStatus(user, List.of(new Status("spring_judge",1)))`
        - 以`Action_Fire`为例：
            - 检查对方护盾，必要时添加火焰状态
            - 破坏护盾 `AddShield(target, -伤害值)`

4.2 状态（Buff）结算

- 回合开始（BeginService）和结束（EndService）阶段分别处理
    - 对于`statusesBegin`，每个状态遍历：
        - 若为`next`型，调用`MainNextBuffService`，如`rain_next`、`war_next`等
        - 其它直接`MainBuffService`，如`mountain`、`sad`等
    - `statusesEnd`在EndService阶段由judge型Buff（如`spring_judge`）决定是否转为Begin状态或产生新状态
- BUFF分派均采用接口+映射表（`BuffAction`、`NextBuffAction`），便于后续功能扩展。

---

5. 房间与玩家状态数据查询

- **前端请求：**
  - `{"type":"fetchall","room":{"roomId":"xxx"}}`
- **服务端处理链条：**
  1. `MainService.handleTextMessage(session, message)`
     （type为`fetchall`，路由至下一级）
  2. `MainService.handleFetchAll(session, messageData)`
     - 解析房间ID，查找房间对象
     - 获取房间内所有玩家ID，查找`PlayerAgainst`对象
     - 汇总所有玩家状态、卡牌、房间信息
     - 广播至房间内所有用户

设计严谨性与逻辑性总结

1. **分层清晰**：网络通信、房间业务、玩家逻辑、卡牌操作、状态系统各自解耦，便于维护与扩展。
2. **并发安全**：所有全局数据结构采用`ConcurrentHashMap`，多线程环境下数据一致性有保障。
3. **同步机制**：回合同步与多客户端消息同步采用双键Map与广播，保证状态一致且无死锁。
4. **Buff与卡牌机制**：采用接口+分发表，所有新卡牌或新BUFF只需注册到映射表即可，无侵入扩展。
5. **数据完整性**：每次业务流程均有详细日志和异常处理，状态变更前后都有清晰记录，便于调试和追溯。

通过上述详细的流程描述及每次函数调用链条，可以证明系统的设计具备高度的正确性与严谨性，完全支撑高并发、复杂博弈和策略丰富的卡牌对战需求。