卡牌对战系统开发思路与模块设计（结合代表性代码段）

本设计说明系统性梳理了卡牌游戏的核心功能实现，从功能整合、原子操作抽象、BUFF影响机制、PlayerCondition扩展，到整体验证流程，突出模块化、低耦合与高可扩展性。每一部分均结合了代表性代码片段和实现细节，旨在为后续开发和优化提供坚实的理论和实践基础。

---

1. 所有卡牌功能的整合设计

设计原则

- **统一模型化**：所有卡牌通过`CardBattle`类进行标准化建模，包括类型（进攻/防守/增益/减益）、名称、数量、面值等核心字段。
- **集中注册**：在`CardService`静态初始化区，集中登记所有可用卡牌及其属性，便于运维和扩展。
- **类型化分发**：通过`cardType`区分不同卡牌功能，在效果处理时可实现优先级和分组调度。

代表性代码

```
// CardBattle.java
public class CardBattle {
    // 进攻/防守/增益/减益等类型
    private String cardType;
    private int cardNum;
    private String cardName;
    private int cardSize;
    // ... 构造函数等
}
```

```
// CardService.java - 所有卡牌集中初始化
static {
    arrayList.add(new CardBattle("profit", 1, "spring", 1)); // 春
    arrayList.add(new CardBattle("battle", 1, "fire", 1));   // 火
    // ... 其它所有卡牌
}
```

---

2. 抽离所有卡牌效果为原子操作组合

设计原则

- **原子操作抽象**：所有卡牌效果均通过修改金币（AddCoins）、修改血量（AddHP）、修改护盾（AddShield）、丢弃/获得卡牌等基础操作实现。
- **组合实现复杂效果**：每种卡牌效果实为1-3个原子操作的有序组合，便于维护和单元测试。

- **分离效果与判定逻辑**：判定型和即时型效果分离，保障后续BUFF和状态机制能灵活插入。

代表性代码

```
// PlayerService.java - 原子操作定义
public void AddCoins(PlayerAgainst player, int coins) { ... }
public void AddHP(PlayerAgainst player, int hp) { ... }
public void AddShield(PlayerAgainst player, int shield) { ... }
public void killShield(PlayerAgainst player, int shield) { ... }
```

```
// 以Action_Spring为例
private void Action_Spring(PlayerAgainst user, PlayerAgainst target, CardBattle
cardBattle) {
    user.getPlayerCondition().setUsedProfitOrDecreaseCard(true);
    AddHP(user,2); // 原子操作1：加血
    AddStatus(user, List.of(new Status("spring_judge",1))); // 原子操作2：加状态
}
```

- 其它卡牌效果均可见类似模式，如火卡（攻击、判定）、酒卡（攻击、加状态）、柳卡（加盾、加状态）等。

---

3. BUFF影响的实现流程

概括实现方案

- **BUFF抽象为状态（Status）**：统一所有持续性/判定性影响为`Status`对象，分为回合开始（Begin）和回合结束（End）两类。
- **分阶段处理**：在`BeginService`和`EndService`分别结算BUFF，采用接口`BuffAction`/`NextBuffAction`解耦具体实现。
- **分发表调度**：通过`buffActions`和`nextBuffActionMap`将BUFF名称映射到对应的处理函数，实现开放封闭原则。

典型示例

示例1：春天判定BUFF

- 春卡出牌时加`spring_judge`状态到End
- 回合结束时，`EndService`遍历`statusesEnd`，遇到`spring_judge`，分发到`BuffAction_spring_judge`
- 若玩家本回合未受攻击，则在下一回合开始时加3金币并抽1张牌（spring BUFF进入Begin）

```
// PlayerService.java
private void BuffAction_spring_judge(PlayerAgainst user, PlayerAgainst target) {
    if(!user.getPlayerCondition().isAttacked()){
        user.getStatusesBegin().add(new Status("spring",1,"回合开始时获得3金币，且抽
```

```
1张牌"));
    }
}
```

示例2：火焰BUFF（延迟加成）

- 火卡对无护盾目标时为自身叠加`fire`状态
- 下一回合开始时，`BeginService`分发到`BuffAction_fire`，抽一张牌并提升火攻伤害

```
private void BuffAction_fire(PlayerAgainst user, PlayerAgainst target) {
    user.setCards(cardService.RandomGetCardsByNumAndCost(1,2));

user.getPlayerCondition().setFireAdd(user.getPlayerCondition().getFireAdd()+1);
}
```

---

4. PlayerCondition类的引入与原子操作的BUFF兼容

设计原则

- **单独抽象玩家临时状态**：`PlayerCondition`类，记录一回合内所有临时性判定（如是否被攻击、是否能加盾、是否净化等），并作为所有BUFF和原子操作的兼容层。
- **原子操作感知BUFF影响**：所有原子操作（如AddShield、AddHP）需优先检测和修改`PlayerCondition`，确保BUFF对基础行为的全方位影响。
- **高可扩展性**：新增BUFF或修改判定条件只需增加/修改`PlayerCondition`属性及操作，不影响主流程。

代表性代码

```
// PlayerCondition.java
public class PlayerCondition {
    private boolean isAttacked;
    private boolean canAddShield;
    private int fireAdd;
    private boolean purification;
    private boolean liuBuff;
    // ... 其它判定与标记
}
```

```
// PlayerService.java - AddShield原子操作的BUFF影响
public void AddShield(PlayerAgainst player, int shield) {
    if(shield > 0) {
        if(player.getPlayerCondition().isCanAddShield()) {
            player.getPlayerCondition().setAddedShield(true);

player.getPlayerCondition().setNumOfShieldAdd(player.getPlayerCondition().getNumOf
ShieldAdd() + shield);
```

```
            } else return;
        } else if (shield < 0)
            player.getPlayerCondition().setAttacked(true);
        // ... 其它逻辑
    }
```

- 这样shield相关BUFF（如"悲"无法加盾，"壮志难酬"共享加盾等）都能被统一兼容。

---

5. ToyModel接口验证

验证思路

- **单元测试/集成测试**：设计ToyModel，依次模拟创建玩家、添加卡牌、出牌、触发BUFF、状态切换等全流程。
- **接口验证**：通过ToyModel验证每一个原子操作、BUFF机制和PlayerCondition判定的正确性。
- **高覆盖率**：ToyModel可灵活组合各种状态，模拟极端/边界场景，保障所有功能点可用且无副作用。

```
// ToyModel伪代码
PlayerAgainst p1 = new PlayerAgainst();
PlayerAgainst p2 = new PlayerAgainst();
PlayerService ps = new PlayerService();

p1.setCards(...spring, fire...);
ps.MainOpService(p1, p2, springCard);
assert(p1.getHp() == ...); // 验证加血
assert(p1.getStatusesEnd().contains("spring_judge"));

ps.EndService(p1, p2);
assert(p1.getStatusesBegin().contains("spring"));
ps.BeginService(p1, p2, ..., ..., false);
// ... 更多断言
```

---

6. 分层设计思路简介

为什么要分层、怎么分

- **分层目标**：
    - 保证每个功能点（如卡牌管理、玩家状态、BUFF判定、逻辑调度）都单独封装，职责单一，提升逻辑清晰度和可维护性。
    - 通过数据模型与逻辑代码的分层，确保游戏状态同步一致，便于多人对局和并发操作。
    - 降低各模块之间耦合度，提高扩展性和单元测试的便利性。
- **实际分层方式**：
    1. **实体层（Entity）**：如`CardBattle`、`PlayerAgainst`、`PlayerCondition`、`Status`，定义所有基础数据结构。

2. **服务层（Service）**：
    - `CardService`：卡牌的查找、合并、抽取、丢弃等原子操作。
    - `PlayerService`：卡牌效果执行、BUFF判定、回合流程、原子操作的统一入口。
3. **主控调度层（MainService）**：负责WebSocket事件、房间管理、玩家管理、游戏主流程调度。

- **举例说明每层调用逻辑**（以一次完整出牌流程为例）：
    - `MainService.handleRoundEndMessage()` 负责接收客户端回合结束请求，组织数据并调用核心逻辑。
    - 调用 `PlayerService.MainService(player1, player2, list1, list2, room)`，分发到回合内出牌、BUFF结算等流程。
    - `MainService` 仅负责数据调度、消息广播，不直接处理卡牌或状态逻辑，所有实际业务通过`PlayerService`和`CardService`实现。
    - `CardService` 只专注于卡牌操作本身（如扣牌、抽牌、合成），不关心玩家状态、BUFF等上层逻辑。
    - `PlayerService` 作为中枢，既调用`CardService`完成卡牌操作，也管理`PlayerAgainst`和`PlayerCondition`的状态变更，确保BUFF与回合流程协同运作。

分层带来的好处总结

- **逻辑通顺，层次分明**：每个文件/模块只关心一类问题，流程清晰可追溯。
- **数据同步安全**：主控层统一调度，所有状态变更集中落地，杜绝并发错误。
- **极高的可扩展性**：无论是增加卡牌、BUFF、状态类型、还是调整流程，只需在单一模块内做小范围更改。
- **易于测试与维护**：每一层都可独立Mock并验证，接口清晰，便于定位和修复问题。
- **可复用性强**：原子操作、状态管理与卡牌流程完全解耦，未来可用于更多衍生玩法/模式。

总结：分层设计的正确性

通过上述分层设计，每个模块各司其职，既能保证全局数据一致、同步和安全，也为后续功能扩展和高并发对战场景打下理论和工程基础。这种分层模式也是业界多人对战游戏后端的主流做法，能够最大化实现系统的健壮性和灵活性。