

# 扑克消除游戏 - 程序设计文档

## 文档版本

- 版本号: v1.0
- 创建日期: 2025-10-29
- 适用项目: 扑克消除游戏 (Cocos2d-x)
- 作者: 王雅楠

## 目录

- [项目概述](#)
- [架构设计](#)
- [代码结构详解](#)
- [扩展指南](#)
- [配置系统](#)
- [开发规范](#)
- [测试与调试](#)
- [常见问题FAQ](#)

## 1. 项目概述

### 1.1 项目简介

这是一款基于 **Cocos2d-x** 引擎开发的扑克牌消除类休闲游戏，采用 **TriPeaks** (三峰纸牌) 核心玩法，玩家通过匹配点数相邻的卡牌来消除主牌区的所有卡牌。

### 1.2 核心玩法

- 匹配规则:** 点数相差  $\pm 1$  即可匹配 (例如: 2可以匹配1或3, K可以匹配Q或A)
- 主牌区:** 包含多列竖直堆叠的卡牌，只有顶部卡牌可以匹配
- 手牌区:** 显示当前可用于匹配的卡牌
- 备用牌堆:** 玩家可以从此抽取新的手牌
- 胜利条件:** 消除主牌区的所有卡牌
- 失败条件:** 备用牌堆用完且无可匹配的卡牌

### 1.3 技术栈

- 游戏引擎: Cocos2d-x 3.17
- 编程语言: C++11/14
- 构建工具: CMake

- **数据格式:** JSON (RapidJSON)
- **平台支持:** Windows

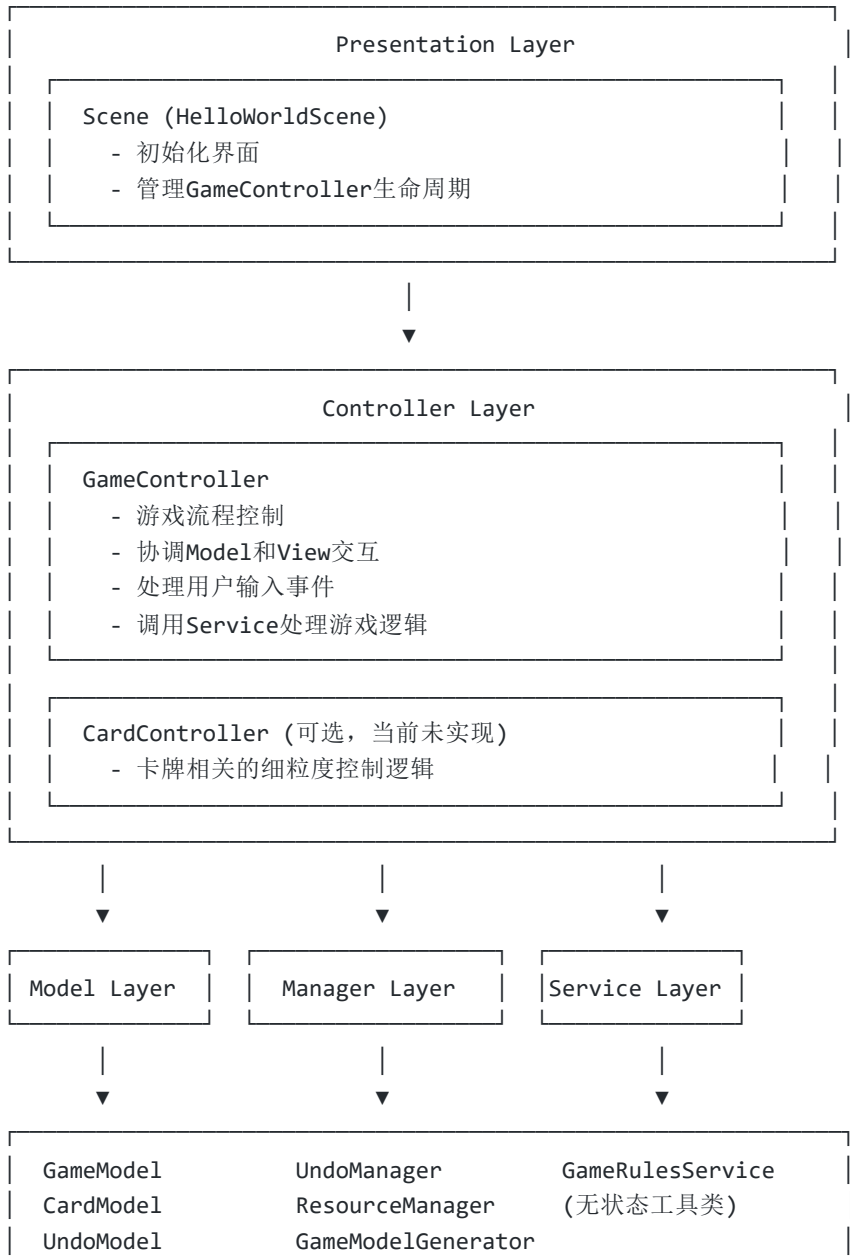
## 1.4 项目目标

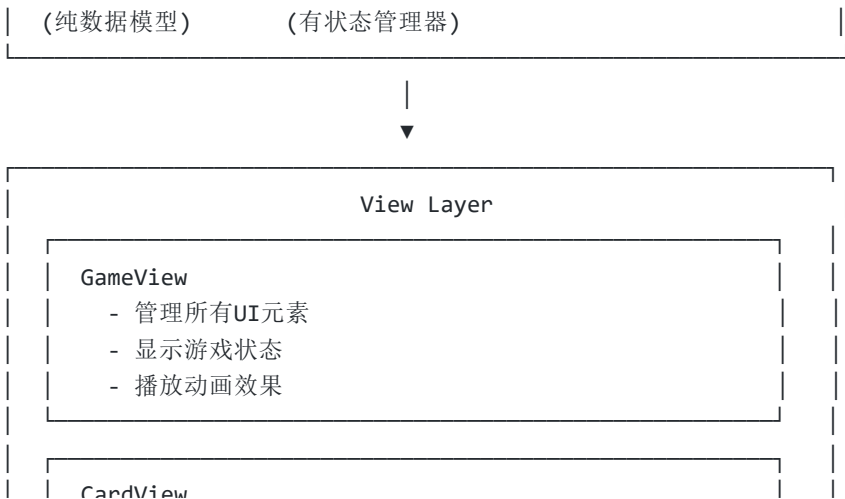
- **高可维护性:** 清晰的代码架构，易于理解和修改
- **高扩展性:** 支持快速添加新卡牌类型、新游戏规则、新撤销功能
- **高复用性:** 模块化设计，组件可复用于其他卡牌游戏
- **高性能:** 优化内存使用，流畅的动画表现

## 2. 架构设计

### 2.1 整体架构

本项目采用 **MVC (Model-View-Controller)** 架构模式，并结合 **Manager-Service** 设计模式，实现了职责清晰、低耦合的代码结构。





## 2.2 设计模式

### 2.2.1 MVC 模式

- **Model (数据模型)**
  - GameModel：游戏状态数据
  - CardModel：卡牌数据
  - UndoModel：撤销操作数据
  - **特点:** 纯数据，不包含业务逻辑，只提供数据访问接口
- **View (视图层)**
  - GameView：游戏主界面
  - CardView：卡牌显示组件
  - **特点:** 只读取Model数据，通过回调通知Controller用户操作
- **Controller (控制器)**
  - GameController：游戏总控制器
  - **特点:** 协调Model和View，调用Service处理业务逻辑

### 2.2.2 Service 模式

- **GameRulesService:** 无状态的游戏规则服务
  - 纯静态方法
  - 不持有任何数据
  - 只通过参数传递数据
  - 提供游戏规则判断和执行逻辑

### 2.2.3 Manager 模式

- **UndoManager:** 撤销管理器
- **ResourceManager:** 资源管理器
- **GameModelGenerator:** 游戏模型生成器
- **特点:** 可以持有状态数据，管理特定领域的功能

## 2.2.4 Factory 模式

- **GameModelGenerator**: 负责创建和初始化GameModel
- 根据LevelConfig生成不同配置的游戏模型

## 2.2.5 Command 模式

- **UndoModel**: 封装撤销操作的命令对象
- 支持撤销 (undo) 和重做 (redo)

## 2.3 数据流向

### 2.3.1 初始化流程

```
AppDelegate
└─> HelloWorldScene::init()
    └─> GameController::initGame(levelId)
        ├──> LevelConfigLoader::loadLevelConfig(levelId) // 加载配置
        ├──> GameModelGenerator::generateGameModel() // 生成Model
        ├──> GameView::create(gameModel) // 创建View
        └─> UndoManager::init() // 初始化撤销管理器
```

### 2.3.2 用户操作流程

```
用户点击卡牌
└─> CardView 触发点击事件
    └─> GameView::_onCardClickCallback(cardId)
        └─> GameController::handleCardMatch(cardId, handCardId)
            ├──> GameRulesService::canCardsMatch() // 判断规则
            ├──> UndoManager::addUndoAction() // 记录撤销
            ├──> GameRulesService::executeCardMatch() // 执行匹配
            ├──> GameModel 更新数据
            └─> GameView::updateGameDisplay() // 刷新界面
```

### 2.3.3 撤销操作流程

```
用户点击撤销按钮
└─> GameView::_onUndoCallback()
    └─> GameController::handleUndo()
        └─> UndoManager::undo()
            ├──> UndoModel::executeUndo(gameModel) // 执行撤销
            ├──> GameModel 数据回滚
            └─> GameController::updateViewAfterUndo() // 更新视图
                └─> GameView 播放撤销动画
```

## 2.4 架构优势

### 2.4.1 职责分离

- **Model**: 只负责数据存储

- **View:** 只负责显示
- **Controller:** 只负责协调
- **Service:** 只负责业务逻辑
- **Manager:** 只负责特定领域的管理

## 2.4.2 低耦合

- View 不直接修改 Model (通过回调通知Controller)
- Model 不依赖 View (纯数据类)
- Service 无状态 (不依赖任何实例)
- Controller 是唯一的协调者

## 2.4.3 高内聚

- 每个类都有明确的单一职责
- 相关功能聚合在同一模块
- 易于理解和维护

## 2.4.4 可测试性

- Service 可以独立单元测试
- Model 可以独立测试数据逻辑
- Controller 可以通过 Mock 对象测试

# 3. 代码结构详解

## 3.1 目录结构

Classes/	# 游戏核心代码
├── models/	# 数据模型层 (Model)
│   ├── CardModel.h/cpp	# 卡牌数据模型
│   ├── GameModel.h/cpp	# 游戏数据模型
│   └── UndoModel.h/cpp	# 撤销操作数据模型
├── views/	# 视图层 (View)
│   ├── CardView.h/cpp	# 卡牌视图组件
│   └── GameView.h/cpp	# 游戏主视图
├── controllers/	# 控制器层 (Controller)
│   ├── GameController.h/cpp	# 游戏总控制器
│   └── CardController.h/cpp	# 卡牌控制器 (可选)
├── services/	# 服务层 (Service)
│   └── GameRulesService.h/cpp	# 游戏规则服务 (无状态)
├── managers/	# 管理器层 (Manager)
│   ├── UndoManager.h/cpp	# 撤销管理器
│   ├── ResourceManager.h/cpp	# 资源管理器
│   └── GameModelGenerator.h/cpp	# 游戏模型生成器

```

|
|— configs/                                # 配置系统
|   |— models/                            # 配置数据模型
|   |   |— LevelConfig.h/cpp             # 关卡配置模型
|   |   |— loaders/                      # 配置加载器
|   |       |— LevelConfigLoader.h/cpp    # 关卡配置加载器
|
|— utils/                                  # 工具类
|   |— CardGameUtils.h/cpp               # 卡牌游戏工具
|   |— GameConstants.h                   # 游戏常量定义
|
|— AppDelegate.h/cpp                      # 应用程序入口
|— HelloWorldScene.h/cpp                 # 游戏主场景

configs/                                # 配置文件（JSON）
|— levels/                              # 关卡配置
|   |— level_1.json                     # 第1关配置

Resources/                              # 资源文件
|— cards/                               # 卡牌资源
|   |— card_config.json                 # 卡牌配置
|   |— *.png                           # 卡牌图片
|— fonts/                               # 字体文件

```

## 3.2 核心类详解

### 3.2.1 Model 层

#### CardModel (卡牌数据模型)

```

class CardModel {
public:
    // 枚举
    enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }; // 花色
    enum Rank { ACE, TWO, ..., KING };           // 点数

    // 基本属性
    int _cardId;                                // 唯一ID
    Suit _suit;                                  // 花色
    Rank _rank;                                  // 点数

    // 状态属性
    bool _isFaceUp;                             // 是否正面朝上
    bool _isSelected;                            // 是否被选中
    bool _isMovable;                             // 是否可移动
    bool _isVisible;                             // 是否可见

    // 位置属性
    Vec2 _position;                              // 位置坐标
    int _zOrder;                                 // 渲染层级

    // 游戏属性
    std::string _cardArea;                       // 所在区域 (playfield/stack/hand)
    int _areaIndex;                              // 区域内索引

```

```

// 核心方法
int getValue() const;           // 获取卡牌值
std::string getCardName() const; // 获取卡牌名称
bool isRedSuit() const;        // 是否红色花色
void flipCard();               // 翻转卡牌

// 序列化
std::string serialize() const;
bool deserialize(const std::string& jsonStr);
};

```

## 职责:

- 存储单张卡牌的所有数据
- 提供卡牌数据的访问接口
- 支持序列化和反序列化

## 使用场景:

- 表示游戏中的一张卡牌
- 保存游戏状态时序列化卡牌数据
- 网络同步时传输卡牌信息

## GameModel (游戏数据模型)

```

class GameModel {
public:
    // 游戏状态枚举
    enum GameState { NONE, PLAYING, PAUSED, FINISHED, GAME_OVER };

    // 基本信息
    int _levelId;           // 关卡ID
    int _gameId;            // 游戏ID
    GameState _gameState;   // 游戏状态
    float _gameTime;       // 游戏时间
    int _score;             // 当前分数
    int _targetScore;       // 目标分数

    // 卡牌管理
    std::vector<CardModel*> _allCards;      // 所有卡牌
    std::vector<CardModel*> _playfieldCards; // 主牌区卡牌
    std::vector<CardModel*> _stackCards;    // 备用牌堆
    std::vector<CardModel*> _handCards;     // 手牌区

    // 游戏规则
    int _maxUndoCount;      // 最大撤销次数
    int _currentUndoCount;  // 当前撤销次数
    bool _allowHint;        // 是否允许提示
    bool _allowUndo;        // 是否允许撤销

    // 游戏进度
    int _movesCount;        // 移动次数
    int _hintsUsed;         // 使用提示次数
};

```

```

// 核心方法
bool initGame(int levelId);
void startGame();
void pauseGame();
void endGame();

CardModel* getCardById(int cardId);
bool moveCardToArea(int cardId, const std::string& area, int index = -1);
CardModel* getTopHandCard();

bool canUndo() const;
int calculateStars() const;
};

```

## 职责:

- 存储游戏的所有状态数据
- 管理卡牌集合
- 提供数据访问和修改接口
- 不包含业务逻辑（逻辑在Service和Controller中）

## UndoModel (撤销操作数据模型)

```

class UndoModel {
public:
    // 操作类型枚举
    enum ActionType {
        ACTION_MOVE_CARD,        // 移动卡牌
        ACTION_FLIP_CARD,        // 翻转卡牌
        ACTION_DEAL_CARD,        // 发牌
        ACTION_SHUFFLE,          // 洗牌
        ACTION_SELECT_CARD       // 选中卡牌
    };

    // 操作基本信息
    ActionType _actionType;      // 操作类型
    int _cardId;                 // 卡牌ID
    std::string _fromPosition;   // 起始位置
    std::string _toPosition;     // 目标位置
    float _timestamp;           // 时间戳

    // 操作前状态
    Vec2 _positionBefore;        // 操作前位置
    bool _faceUpBefore;          // 操作前是否正面
    bool _selectedBefore;        // 操作前是否选中

    // 核心方法
    bool executeUndo(GameModel* gameModel);    // 执行撤销
    bool executeRedo(GameModel* gameModel);    // 执行重做
    std::string getActionDescription() const;  // 获取操作描述
};

```

## 职责:



- 存储单次操作的数据
- 封装撤销和重做的逻辑
- 记录操作前后的状态

## 3.2.2 Controller 层

### GameController (游戏总控制器)

```
class GameController {
public:
    // 游戏流程控制
    bool initGame(int levelId);
    bool startGame();
    void pauseGame();
    void resumeGame();
    void restartGame();
    void endGame();

    // 用户操作处理
    bool handleCardClick(int cardId);
    bool handleCardMatch(int playfieldCardId, int handCardId);
    bool handleStackCardFlip(int stackCardId);
    bool handleUndo();
    bool handleHint();

    // 游戏更新
    void updateGame(float deltaTime);

    // 访问器
    GameView* getGameView() const;
    GameModel* getGameModel() const;

private:
    // 核心组件
    GameModel* _gameModel;
    GameView* _gameView;
    ResourceManager* _resourceManager;
    UndoManager* _undoManager;
    GameModelGenerator* _levelGenerator;

    // 辅助方法
    void recordUndoAction(...);
    void updateGameUI();
    bool checkGameEndCondition();
};
```

#### 职责:

- 管理游戏生命周期
- 协调Model和View的交互
- 处理用户输入事件
- 调用Service执行业务逻辑
- 管理子管理器 (UndoManager等)

## 设计要点:

- Controller不直接实现业务逻辑，而是调用Service
- Controller负责数据流转和组件协调
- Controller持有Model、View、Manager的引用

## 3.2.3 View 层

### GameView (游戏主视图)

```
class GameView : public Layer {
public:
    // 创建和初始化
    static GameView* create(const GameModel* gameModel, ResourceManager* rm);
    virtual bool init(const GameModel* gameModel, ResourceManager* rm);

    // 回调设置
    void setCardClickCallback(std::function<void(int)> callback);
    void setUndoCallback(std::function<void()> callback);
    void setHintCallback(std::function<void()> callback);

    // UI更新
    void updateGameDisplay();
    void updateScoreDisplay();
    void updateTimeDisplay();

    // 动画播放
    void playCardMoveAnimation(int cardId, const Vec2& pos, float duration);
    void playCardMatchAnimation(int cardId1, int cardId2);
    void playWinAnimation();
    void playLoseAnimation();

    // 提示效果
    void showHintEffect(const std::vector<int>& cardIds);
    void showStackHintEffect(int cardId);

    // 游戏结果
    void showGameResultWithStars(bool isWin, int score, int stars);

private:
    const GameModel* _gameModel;    // 只读GameModel
    ResourceManager* _resourceManager;
    std::map<int, CardView*> _cardViews;

    // UI元素
    Label* _scoreLabel;
    Label* _timeLabel;
    MenuItem* _undoButton;
    MenuItem* _hintButton;
};
```

## 职责:

- 管理所有UI元素

- 显示游戏状态
- 播放动画效果
- 响应用户输入（通过回调通知Controller）

#### 设计要点:

- View只读取Model数据（const GameModel\*）
- View不直接修改Model
- View通过回调函数将用户操作通知给Controller

### 3.2.4 Service 层

#### GameRulesService (游戏规则服务)

```
class GameRulesService {
public:
    // 规则判断
    static bool canCardsMatch(CardModel* card1, CardModel* card2);
    static bool checkGameWon(const GameModel* gameModel);
    static bool checkGameLost(const GameModel* gameModel);
    static bool hasAvailableMoves(const GameModel* gameModel);

    // 操作执行
    static bool executeCardMatch(GameModel* gameModel, int cardId1, int cardId2);
    static void shuffleCards(std::vector<CardModel*>& cards);
    static bool dealCards(GameModel* gameModel, int count, const std::string& area);

    // 辅助方法
    static std::vector<std::pair<int, int>> findMatchingPairs(const GameModel* gm);

private:
    GameRulesService() = delete;    // 禁止实例化
    ~GameRulesService() = delete;
};
```

#### 职责:

- 提供无状态的游戏规则判断
- 执行游戏规则相关的操作
- 不持有任何数据，纯静态方法

#### 设计要点:

- 所有方法都是静态方法
- 通过参数传递数据
- 不依赖任何实例变量
- 可以独立单元测试

### 3.2.5 Manager 层

#### UndoManager (撤销管理器)

```

class UndoManager {
public:
    bool init(GameModel* gameModel, int maxUndoCount);

    // 撤销/重做
    bool addUndoAction(UndoModel* undoModel);
    bool undo();
    bool redo();

    // 查询
    bool canUndo() const;
    bool canRedo() const;
    UndoModel* getLastUndoAction() const;
    int getUndoCount() const;

    // 历史管理
    void clearUndoHistory();
    void clearRedoHistory();
    std::string serializeUndoHistory() const;

private:
    GameModel* _gameModel;
    std::vector<UndoModel*> _undoHistory;
    std::vector<UndoModel*> _redoHistory;
    int _maxUndoCount;
};

```

## 职责:

- 管理撤销和重做历史
- 执行撤销和重做操作
- 维护操作栈

## 设计要点:

- 持有GameModel引用（但不拥有）
- 持有UndoModel列表（拥有所有权）
- 支持序列化历史记录

## GameModelGenerator (游戏模型生成器)

```

class GameModelGenerator {
public:
    GameModel* generateGameModel(int levelId, LevelConfigLoader* loader);
    GameModel* generateGameModel(LevelConfig* levelConfig);

    void setRandomSeed(unsigned int seed);
    unsigned int getRandomSeed() const;

private:
    unsigned int _randomSeed;

    bool initializeGameModel(GameModel* gm, LevelConfig* config);
    bool createCardCollection(GameModel* gm, LevelConfig* config);

```

```
std::vector<CardModel*> generateRandomCards(int count);  
void shuffleCards(std::vector<CardModel*>& cards);  
};
```

#### 职责:

- 将静态配置转换为运行时数据
- 生成和初始化GameModel
- 处理卡牌随机生成
- 管理随机种子

## 4. 扩展指南

---

### 4.1 如何添加新卡牌

#### 4.1.1 添加新卡牌类型（扩展花色或点数）

**场景:** 假设要添加第5种花色 "JOKER"

**步骤1:** 扩展 CardModel 枚举

```
// Classes/models/CardModel.h  
  
class CardModel {  
public:  
    enum Suit {  
        CLUBS = 0,  
        DIAMONDS = 1,  
        HEARTS = 2,  
        SPADES = 3,  
        JOKER = 4           // ✅ 新增：小丑牌花色  
    };  
  
    enum Rank {  
        ACE = 0,  
        TWO = 1,  
        // ...  
        KING = 12,  
        JOKER_SMALL = 13,   // ✅ 新增：小王  
        JOKER_BIG = 14      // ✅ 新增：大王  
    };  
};
```

**步骤2:** 更新获取名称的方法

```
// Classes/models/CardModel.cpp  
  
std::string CardModel::getSuitName() const {  
    switch (_suit) {  
        case HEARTS: return "Hearts";  
        case DIAMONDS: return "Diamonds";
```

```

        case CLUBS: return "Clubs";
        case SPADES: return "Spades";
        case JOKER: return "Joker";      // ✅ 新增
        default: return "Unknown";
    }
}

std::string CardModel::getRankName() const {
    // ...
    case KING: return "King";
    case JOKER_SMALL: return "Small Joker";    // ✅ 新增
    case JOKER_BIG: return "Big Joker";        // ✅ 新增
    default: return "Unknown";
}

```

### 步骤3: 更新游戏规则 (如果需要)

```

// Classes/services/GameRulesService.cpp

bool GameRulesService::canCardsMatch(CardModel* card1, CardModel* card2) {
    // ✅ 新增: 大小王可以匹配任意卡牌
    if (card1->_rank == CardModel::JOKER_BIG ||
        card1->_rank == CardModel::JOKER_SMALL ||
        card2->_rank == CardModel::JOKER_BIG ||
        card2->_rank == CardModel::JOKER_SMALL) {
        return true;
    }

    // 原有逻辑
    int rank1 = static_cast<int>(card1->_rank);
    int rank2 = static_cast<int>(card2->_rank);
    return abs(rank1 - rank2) == 1;
}

```

### 步骤4: 添加资源文件

```

Resources/cards/
├─ joker_small.png    # ✅ 小王图片
├─ joker_big.png      # ✅ 大王图片
└─ card_config.json   # 配置文件中添加新卡牌路径

```

### 步骤5: 更新关卡配置 (可选)

```

// configs/levels/level_1.json
{
    "playfieldCards": [
        {"cardFace": 13, "cardSuit": 4, "position": {...}}, // 小王
        {"cardFace": 14, "cardSuit": 4, "position": {...}}  // 大王
    ]
}

```

### 步骤6: 更新 GameModelGenerator

```
// Classes/managers/GameModelGenerator.cpp

std::vector<CardModel*> GameModelGenerator::generateRandomCards(int count) {
    std::vector<CardModel*> cards;
    int cardId = 1;

    // 生成普通卡牌 (4种花色 × 13张点数)
    for (int s = 0; s < 4; s++) {
        for (int r = 0; r < 13; r++) {
            CardModel* card = new CardModel(
                static_cast<CardModel::Suit>(s),
                static_cast<CardModel::Rank>(r),
                cardId++
            );
            cards.push_back(card);
        }
    }

    // ✅ 新增：生成大小王
    cards.push_back(new CardModel(
        CardModel::JOKER,
        CardModel::JOKER_SMALL,
        cardId++
    ));
    cards.push_back(new CardModel(
        CardModel::JOKER,
        CardModel::JOKER_BIG,
        cardId++
    ));

    return cards;
}
```

## 4.1.2 添加新卡牌功能属性

**场景:** 添加"炸弹卡", 点击后清除所有相同点数的卡牌

**步骤1:** 扩展 CardModel 属性

```
// Classes/models/CardModel.h

class CardModel {
public:
    // ✅ 新增：特殊卡牌类型
    enum SpecialType {
        SPECIAL_NONE = 0,
        SPECIAL_BOMB = 1,          // 炸弹卡
        SPECIAL_WILD = 2,          // 万能卡
        SPECIAL_FREEZE = 3         // 冻结卡
    };

    SpecialType _specialType;      // ✅ 新增：特殊类型
    int _specialValue;            // ✅ 新增：特殊值

    // ✅ 新增：检查是否为特殊卡牌
```

```

    bool isSpecialCard() const;
    SpecialType getSpecialType() const;
};

```

## 步骤2: 实现特殊卡牌逻辑

// Classes/services/GameRulesService.h/cpp

```

class GameRulesService {
public:
    // ✅ 新增: 处理特殊卡牌效果
    static bool executeSpecialCard(GameModel* gameModel, int cardId);

private:
    static bool executeBombCard(GameModel* gameModel, CardModel* bombCard);
    static bool executeWildCard(GameModel* gameModel, CardModel* wildCard);
};

// 实现
bool GameRulesService::executeSpecialCard(GameModel* gameModel, int cardId) {
    CardModel* card = gameModel->getCardById(cardId);
    if (!card || !card->isSpecialCard()) return false;

    switch (card->_specialType) {
        case CardModel::SPECIAL_BOMB:
            return executeBombCard(gameModel, card);
        case CardModel::SPECIAL_WILD:
            return executeWildCard(gameModel, card);
        default:
            return false;
    }
}

bool GameRulesService::executeBombCard(GameModel* gameModel, CardModel* bombCard) {
    // 找到所有相同点数的卡牌
    CardModel::Rank targetRank = bombCard->_rank;
    std::vector<CardModel*> cardsToRemove;

    for (CardModel* card : gameModel->_playfieldCards) {
        if (card->_rank == targetRank && card->_isFaceUp) {
            cardsToRemove.push_back(card);
        }
    }

    // 移除卡牌
    for (CardModel* card : cardsToRemove) {
        auto it = std::find(gameModel->_playfieldCards.begin(),
                           gameModel->_playfieldCards.end(), card);
        if (it != gameModel->_playfieldCards.end()) {
            gameModel->_playfieldCards.erase(it);
            gameModel->_handCards.push_back(card);
        }
    }

    // 增加分数

```

## 步骤3: 在Controller中处理特殊卡牌



```
// Classes/controllers/GameController.cpp

bool GameController::handleCardClick(int cardId) {
    CardModel* card = _gameModel->getCardById(cardId);
    if (!card) return false;

    // ✅ 新增：检查是否为特殊卡牌
    if (card->isSpecialCard()) {
        recordUndoAction(UndoModel::ACTION_USE_SPECIAL_CARD, cardId, "", "");

        if (GameRulesService::executeSpecialCard(_gameModel, cardId)) {
            updateGameUI();
            checkGameEndCondition();
            return true;
        }
        return false;
    }

    // 原有逻辑...
    return handleCardMatch(cardId, topHandCard->_cardId);
}
```

#### 步骤4: 更新 UndoModel 支持新操作

```
// Classes/models/UndoModel.h

class UndoModel {
public:
    enum ActionType {
        ACTION_MOVE_CARD,
        ACTION_USE_SPECIAL_CARD,    // ✅ 新增：使用特殊卡牌
        // ...
    };

    std::vector<int> _affectedCardIds; // ✅ 新增：受影响的卡牌ID列表
};

// 实现撤销逻辑
bool UndoModel::executeUndo(GameModel* gameModel) {
    switch (_actionType) {
        case ACTION_USE_SPECIAL_CARD:
            // 将所有受影响的卡牌恢复到主牌区
            for (int cardId : _affectedCardIds) {
                CardModel* card = gameModel->getCardById(cardId);
                if (card) {
                    gameModel->moveCardToArea(cardId, "playfield");
                    card->_position = _positionBefore;
                }
            }
            return true;
        // ...
    }
}
```

## 4.2 如何添加新类型的撤销功能

### 4.2.1 撤销系统架构

当前撤销系统基于 **Command Pattern (命令模式)** 设计：

```
UndoManager (管理器)
├-> std::vector<UndoModel*> _undoHistory (撤销栈)
├-> std::vector<UndoModel*> _redoHistory (重做栈)
└-> 方法
    ├──> addUndoAction(UndoModel*)    // 添加操作到撤销栈
    ├──> undo()                        // 执行撤销
    └-> redo()                          // 执行重做

UndoModel (命令对象)
├-> ActionType _actionType             // 操作类型
├-> 操作前状态数据
├-> 操作后状态数据
└-> 方法
    ├──> executeUndo(GameModel*)       // 执行撤销逻辑
    └-> executeRedo(GameModel*)         // 执行重做逻辑
```

### 4.2.2 添加新撤销类型的完整流程

**场景1:** 添加"批量移动卡牌"的撤销功能

**步骤1:** 扩展 ActionType 枚举

```
// Classes/models/UndoModel.h

class UndoModel {
public:
    enum ActionType {
        ACTION_NONE = 0,
        ACTION_MOVE_CARD = 1,
        ACTION_FLIP_CARD = 2,
        ACTION_DEAL_CARD = 3,
        ACTION_SHUFFLE = 4,
        ACTION_SELECT_CARD = 5,
        ACTION_BATCH_MOVE = 6,      // ✓ 新增：批量移动
        ACTION_USE_SPECIAL_CARD = 7 // ✓ 新增：使用特殊卡牌
    };
};
```

**步骤2:** 扩展 UndoModel 数据结构

```
// Classes/models/UndoModel.h

class UndoModel {
public:
    // 原有字段
    ActionType _actionType;
    int _cardId;                // 单卡牌ID
    std::string _fromPosition;
```

```

std::string _toPosition;
Vec2 _positionBefore;
bool _faceUpBefore;

// ✅ 新增：批量操作相关字段
std::vector<int> _cardIds;           // 多卡牌ID列表
std::map<int, Vec2> _positionsBefore; // 每张卡牌的原始位置
std::map<int, bool> _faceUpStates;   // 每张卡牌的翻转状态
std::map<int, std::string> _fromAreas; // 每张卡牌的起始区域

// ✅ 新增：构造函数
UndoModel(ActionType type,
           const std::vector<int>& cardIds,
           const std::string& fromArea,
           const std::string& toArea);
};

```

### 步骤3: 实现新操作的撤销逻辑

```

// Classes/models/UndoModel.cpp

bool UndoModel::executeUndo(GameModel* gameModel) {
    switch (_actionType) {
        // ... 原有case ...

        // ✅ 新增：批量移动的撤销
        case ACTION_BATCH_MOVE: {
            // 遍历所有卡牌，恢复到原始状态
            for (int cardId : _cardIds) {
                CardModel* card = gameModel->getCardById(cardId);
                if (!card) continue;

                // 移回原区域
                std::string fromArea = _fromAreas[cardId];
                gameModel->moveCardToArea(cardId, fromArea);

                // 恢复位置
                card->_position = _positionsBefore[cardId];

                // 恢复翻转状态
                card->_isFaceUp = _faceUpStates[cardId];
            }

            // 更新游戏分数（回退）
            gameModel->_score -= _cardIds.size() * 100;
            gameModel->_movesCount--;

            return true;
        }

        default:
            return false;
    }
}

bool UndoModel::executeRedo(GameModel* gameModel) {

```

```

switch (_actionType) {
    // ✅ 新增：批量移动的重做
    case ACTION_BATCH_MOVE: {
        for (int cardId : _cardIds) {
            CardModel* card = gameModel->getCardById(cardId);
            if (!card) continue;

            // 重新移动到目标区域
            gameModel->moveCardToArea(cardId, _toPosition);
            card->_isFaceUp = true;
        }
    }
}

```

#### 步骤4: 实现业务逻辑 (Service层)

// Classes/services/GameRulesService.h/cpp

```

class GameRulesService {
public:
    // ✅ 新增：批量移动卡牌
    static bool batchMoveCards(
        GameModel* gameModel,
        const std::vector<int>& cardIds,
        const std::string& toArea
    );
};

bool GameRulesService::batchMoveCards(
    GameModel* gameModel,
    const std::vector<int>& cardIds,
    const std::string& toArea
) {
    // 验证所有卡牌都存在且可移动
    for (int cardId : cardIds) {
        CardModel* card = gameModel->getCardById(cardId);
        if (!card || !card->_isMovable) {
            return false;
        }
    }

    // 执行批量移动
    for (int cardId : cardIds) {
        CardModel* card = gameModel->getCardById(cardId);
        gameModel->moveCardToArea(cardId, toArea);
        card->_isFaceUp = true;
    }

    // 更新分数
    gameModel->_score += cardIds.size() * 100;
    gameModel->_movesCount++;

    return true;
}

```

#### 步骤5: 在Controller中记录撤销操作

```
// Classes/controllers/GameController.cpp

bool GameController::handleBatchMove(
    const std::vector<int>& cardIds,
    const std::string& toArea
) {
    if (!_isGameRunning || !_gameModel) return false;

    // ✅ 创建批量撤销对象
    UndoModel* undoModel = new UndoModel(
        UndoModel::ACTION_BATCH_MOVE,
        cardIds,
        "playfield", // fromArea
        toArea        // toArea
    );

    // ✅ 记录每张卡牌的原始状态
    for (int cardId : cardIds) {
        CardModel* card = _gameModel->getCardById(cardId);
        if (card) {
            undoModel->_positionsBefore[cardId] = card->_position;
            undoModel->_faceUpStates[cardId] = card->_isFaceUp;
            undoModel->_fromAreas[cardId] = card->_cardArea;
        }
    }

    // ✅ 添加到撤销管理器
    if (_undoManager) {
        _undoManager->addUndoAction(undoModel);
    }

    // ✅ 执行业务逻辑
    if (GameRulesService::batchMoveCards(_gameModel, cardIds, toArea)) {
        updateGameUI();
        return true;
    }

    return false;
}
```

## 步骤6: 更新View层的撤销动画

```
// Classes/controllers/GameController.cpp

void GameController::updateViewAfterUndo(UndoModel* undoModel) {
    if (!undoModel || !_gameView) return;

    switch (undoModel->_actionType) {
        // ✅ 新增: 批量移动的撤销动画
        case UndoModel::ACTION_BATCH_MOVE: {
            for (int cardId : undoModel->_cardIds) {
                CardModel* card = _gameModel->getCardById(cardId);
                CardView* cardView = _gameView->getCardView(cardId);

                if (card && cardView) {
```

```

        // 播放移动动画
        Vec2 targetPos = card->_position;
        cardView->setVisible(true);
        cardView->playMoveAnimation(targetPos, 0.3f);

        // 更新翻转状态
        cardView->updateCardDisplay();
    }
}
break;
}

// ... 其他case ...
}
}

```

### 4.2.3 高级撤销功能示例

**场景2:** 添加"游戏状态快照"撤销功能（支持撤销多步操作）

**步骤1:** 创建新的撤销模型

```

// Classes/models/UndoSnapshotModel.h

class UndoSnapshotModel : public UndoModel {
public:
    // 完整的游戏状态快照
    std::string _gameStateSnapshot;    // 序列化的完整游戏状态

    UndoSnapshotModel();

    // 保存当前游戏状态
    void captureGameState(const GameModel* gameModel);

    // 恢复游戏状态
    bool restoreGameState(GameModel* gameModel);

    // 重写撤销逻辑
    virtual bool executeUndo(GameModel* gameModel) override;
    virtual bool executeRedo(GameModel* gameModel) override;
};

// 实现
void UndoSnapshotModel::captureGameState(const GameModel* gameModel) {
    if (!gameModel) return;
    _gameStateSnapshot = gameModel->serialize();
}

bool UndoSnapshotModel::restoreGameState(GameModel* gameModel) {
    if (!gameModel || _gameStateSnapshot.empty()) return false;
    return gameModel->deserialize(_gameStateSnapshot);
}

bool UndoSnapshotModel::executeUndo(GameModel* gameModel) {

```

```
    return restoreGameState(gameModel);  
}
```

## 步骤2: 在关键时刻保存快照

```
// Classes/controllers/GameController.cpp  
  
void GameController::saveGameStateSnapshot() {  
    if (!_undoManager || !_gameModel) return;  
  
    UndoSnapshotModel* snapshot = new UndoSnapshotModel();  
    snapshot->_actionType = UndoModel::ACTION_SNAPSHOT;  
    snapshot->captureGameState(_gameModel);  
  
    _undoManager->addUndoAction(snapshot);  
}  
  
// 在关键操作前保存快照  
bool GameController::handleComplexOperation() {  
    // 保存快照  
    saveGameStateSnapshot();  
  
    // 执行复杂操作  
    // ...  
}
```

## 4.2.4 撤销功能扩展的设计原则

### 1. 单一职责原则

- 每个ActionType对应一种明确的操作
- 撤销逻辑封装在UndoModel内部

### 2. 开闭原则

- 通过枚举扩展新类型，不修改已有代码
- 使用switch-case或虚函数实现多态

### 3. 依赖倒置原则

- UndoModel依赖GameModel接口，不依赖具体实现
- 通过GameModel的公共方法修改数据

### 4. 最小知识原则

- UndoModel只知道如何撤销，不关心业务逻辑
- 业务逻辑在Service层，撤销逻辑在Model层

## 撤销功能扩展checklist:

- ☐ 定义新的ActionType
- ☐ 扩展UndoModel数据字段
- ☐ 实现executeUndo()逻辑

- ☐ 实现executeRedo()逻辑
- ☐ 在Controller中记录操作
- ☐ 实现View层的撤销动画
- ☐ 编写单元测试
- ☐ 更新文档

## 扑克消除游戏 - 程序设计文档（第二部分）

### 5. 配置系统

#### 5.1 配置系统架构

配置系统采用 **数据驱动设计 (Data-Driven Design)**，通过JSON文件配置关卡数据，实现代码与数据分离。

```
配置文件 (JSON)
  ↓ 读取
LevelConfigLoader (加载器)
  ↓ 解析
LevelConfig (配置模型)
  ↓ 转换
GameModelGenerator (生成器)
  ↓ 生成
GameModel (运行时数据)
```

#### 5.2 关卡配置文件结构

##### 5.2.1 关卡配置JSON格式

文件路径: configs/levels/level\_1.json

```
{
  // 基本信息
  "levelId": 1,
  "levelName": "关卡 1",
  "levelDescription": "第一个关卡",
  "difficulty": 1,

  // 游戏规则
  "targetScore": 1000,           // 目标分数（通关分数）
  "timeLimit": 0,                // 时间限制（0为无限制）
  "maxUndoCount": 10,           // 最大撤销次数
  "allowHint": true,             // 是否允许提示
  "allowUndo": true,            // 是否允许撤销

  // 星级分数线
  "star1Score": 500,             // 1星分数
  "star2Score": 1000,           // 2星分数
  "star3Score": 1500,           // 3星分数

  // 解锁条件
```



```
"requiredLevelId": 0,           // 需要通关的前置关卡ID（0表示无前置）
"requiredStars": 0,            // 需要的星级数量

// 主牌区卡牌配置（三列竖直堆叠）
"playfieldCards": [
  {
    "cardFace": 2,              // 点数（0=A, 1=2, ..., 12=K）
    "cardSuit": 2,              // 花色（0=梅花, 1=方块, 2=红桃, 3=黑桃）
    "position": {"x": 250, "y": 880}, // 初始位置
    "isFaceUp": false          // 是否正面朝上
  },
  // ... 更多卡牌 ...
],

// 备用牌堆配置
"stackCards": [
  {
    "cardFace": 11,
    "cardSuit": 1,
    "position": {"x": 100, "y": 250}
  },
  // ... 更多卡牌 ...
]
}
```

5.2.2 配置字段说明

字段名	类型	说明	默认值
levelId	int	关卡唯一ID	必填
levelName	string	关卡名称	"关卡 X"
difficulty	int	难度级别（1-5）	1
targetScore	int	通关目标分数	1000
timeLimit	int	时间限制（秒，0为无限制）	0
maxUndoCount	int	最大撤销次数	10
allowHint	bool	是否允许提示	true
allowUndo	bool	是否允许撤销	true
star1Score	int	1星分数线	500
star2Score	int	2星分数线	1000
star3Score	int	3星分数线	1500
playfieldCards	array	主牌区卡牌列表	[]
stackCards	array	备用牌堆卡牌列表	[]

5.3 卡牌资源配置

文件路径: Resources/cards/card\_config.json

```
{
    // 卡牌尺寸
    "cardSize": {
        "width": 120,
        "height": 160
    },

    // 卡牌样式
    "cardCornerRadius": 8.0,
    "cardBorderWidth": 2.0,

    // 纹理路径
    "cardBackTexturePath": "cards/back/card_back.png",
    "cardFrontTexturePath": "cards/card_general.png",
    "suitTexturePath": "cards/suits/",
    "rankTexturePath": "cards/number/",

    // 颜色配置
    "cardBackColor": {"r": 0.2, "g": 0.4, "b": 0.8, "a": 1.0},
    "cardFrontColor": {"r": 1.0, "g": 1.0, "b": 1.0, "a": 1.0},
    "redSuitColor": {"r": 1.0, "g": 0.0, "b": 0.0, "a": 1.0},
    "blackSuitColor": {"r": 0.0, "g": 0.0, "b": 0.0, "a": 1.0},

    // 字体配置
    "rankFontPath": "fonts/Marker Felt.ttf",
    "rankFontSize": 18,

    // 动画配置
    "flipAnimationDuration": 0.5,
    "moveAnimationDuration": 0.3,
    "scaleAnimationDuration": 0.2,

    // 主题
    "themeName": "Default",
    "themeDescription": "默认卡牌主题"
}
```

## 5.4 如何创建新关卡

### 步骤1: 复制模板文件

```
# 复制 level_1.json 作为新关卡模板
cp configs/levels/level_1.json configs/levels/level_2.json
```

### 步骤2: 修改关卡ID和基本信息

```
{
    "levelId": 2,                // ✅ 修改关卡ID
    "levelName": "关卡 2",      // ✅ 修改名称
    "levelDescription": "更难的关卡",
    "difficulty": 2,           // ✅ 增加难度
}
```

```

    "targetScore": 1500,                // ✅ 提高目标分数
    "maxUndoCount": 8,                 // ✅ 减少撤销次数

    "requiredLevelId": 1,              // ✅ 需要通关第1关
    "requiredStars": 1                 // ✅ 需要至少1星
}

```

## 步骤3: 调整卡牌布局

### 方法A: 手动配置每张卡牌

```

"playfieldCards": [
    // 第一列 (x=250)
    {"cardFace": 2, "cardSuit": 2, "position": {"x": 250, "y": 880}, "isFaceUp": false},
    {"cardFace": 5, "cardSuit": 0, "position": {"x": 250, "y": 960}, "isFaceUp": false},
    {"cardFace": 7, "cardSuit": 1, "position": {"x": 250, "y": 1040}, "isFaceUp": true},

    // 第二列 (x=540)
    {"cardFace": 4, "cardSuit": 1, "position": {"x": 540, "y": 880}, "isFaceUp": false},
    // ...
]

```

### 方法B: 使用关卡编辑器 (推荐)

如果项目需要大量关卡, 建议开发一个可视化关卡编辑器:

```

// 伪代码: 关卡编辑器界面
class LevelEditor : public Scene {
public:
    void onDragCard(CardView* cardView, Vec2 newPos);
    void onSaveLevel();
    void exportToJSON(const std::string& filePath);
private:
    std::vector<CardModel*> _editorCards;
    int _currentLevelId;
};

void LevelEditor::exportToJSON(const std::string& filePath) {
    rapidjson::Document doc;
    doc.SetObject();
    auto& allocator = doc.GetAllocator();

    doc.AddMember("levelId", _currentLevelId, allocator);

    rapidjson::Value cardsArray(rapidjson::kArrayType);
    for (CardModel* card : _editorCards) {
        rapidjson::Value cardObj(rapidjson::kObjectType);
        cardObj.AddMember("cardFace", static_cast<int>(card->_rank), allocator);
        cardObj.AddMember("cardSuit", static_cast<int>(card->_suit), allocator);

        rapidjson::Value posObj(rapidjson::kObjectType);
        posObj.AddMember("x", card->_position.x, allocator);
        posObj.AddMember("y", card->_position.y, allocator);
        cardObj.AddMember("position", posObj, allocator);
    }
}

```

```

        cardObj.AddMember("isFaceUp", card->_isFaceUp, allocator);

        cardsArray.PushBack(cardObj, allocator);
    }
    doc.AddMember("playfieldCards", cardsArray, allocator);

    // 写入文件
    rapidjson::StringBuffer buffer;
    rapidjson::Writer<rapidjson::StringBuffer> writer(buffer);
    doc.Accept(writer);

    FileUtils::getInstance()->writeStringToFile(buffer.GetString(), filePath);
}

```

## 步骤4: 调整游戏规则参数

```

{
    // ✅ 增加难度: 减少撤销次数
    "maxUndoCount": 5,

    // ✅ 增加难度: 禁用提示
    "allowHint": false,

    // ✅ 增加难度: 添加时间限制
    "timeLimit": 180, // 3分钟

    // ✅ 调整星级要求
    "star1Score": 1000,
    "star2Score": 2000,
    "star3Score": 3000
}

```

## 步骤5: 测试新关卡

```

// 在 HelloWorldScene.cpp 中测试
bool HelloWorld::init() {
    // ...

    // ✅ 加载新关卡进行测试
    if (!_gameController->initGame(2)) // 关卡2
    {
        CCLOGERROR("Failed to load level 2");
        return false;
    }

    // ...
}

```

## 5.5 配置验证与错误处理

### 5.5.1 配置验证器

```

// Classes/configs/LevelConfigValidator.h

class LevelConfigValidator {
public:
    // 验证关卡配置
    static bool validate(const LevelConfig* config, std::string& errorMsg);

    // 验证卡牌配置
    static bool validateCards(const std::vector<CardData>& cards, std::string& errorMsg);

    // 验证分数配置
    static bool validateScores(const LevelConfig* config, std::string& errorMsg);
};

bool LevelConfigValidator::validate(const LevelConfig* config, std::string& errorMsg) {
    if (!config) {
        errorMsg = "Config is null";
        return false;
    }

    // 验证关卡ID
    if (config->levelId <= 0) {
        errorMsg = "Invalid levelId: " + std::to_string(config->levelId);
        return false;
    }

    // 验证星级分数递增
    if (config->star1Score >= config->star2Score ||
        config->star2Score >= config->star3Score) {
        errorMsg = "Star scores must be in ascending order";
        return false;
    }

    // 验证目标分数
    if (config->targetScore <= 0) {
        errorMsg = "Invalid targetScore";
        return false;
    }

    // 验证卡牌配置
    if (!validateCards(config->playfieldCards, errorMsg)) {
        return false;
    }

    return true;
}

bool LevelConfigValidator::validateCards(const std::vector<CardData>& cards, std::string& errorMsg) {
    if (cards.empty()) {

```

## 5.5.2 在加载器中使用验证器

```

// Classes/configs/loaders/LevelConfigLoader.cpp

LevelConfig* LevelConfigLoader::loadLevelConfig(int levelId) {
    std::string filePath = getConfigFilePath(levelId);

```

```

if (!FileUtils::getInstance()->isFileExist(filePath)) {
    CCLOGWARN("Level config file not found: %s", filePath.c_str());
    return createDefaultLevelConfig(levelId);
}

LevelConfig* config = loadFromFile(levelId);

// ✅ 验证配置
std::string errorMsg;
if (!LevelConfigValidator::validate(config, errorMsg)) {
    CCLOGERROR("Level config validation failed: %s", errorMsg.c_str());
    delete config;
    return createDefaultLevelConfig(levelId);
}

return config;
}

```

## 5.6 热重载配置（开发阶段）

**场景:** 在开发阶段，修改JSON文件后不重启游戏即可看到效果

```

// Classes/controllers/GameController.h

class GameController {
public:
    // ✅ 新增：热重载关卡配置
    bool reloadLevelConfig();

    // ✅ 新增：监听文件变化（可选）
    void watchConfigFile();
};

bool GameController::reloadLevelConfig() {
    if (!_levelConfigLoader || _currentLevelId <= 0) {
        return false;
    }

    CCLOG("Reloading level config for level %d...", _currentLevelId);

    // 清除缓存
    _levelConfigLoader->clearCache();

    // 重新加载配置
    LevelConfig* newConfig = _levelConfigLoader->loadLevelConfig(_currentLevelId);
    if (!newConfig) {
        CCLOGERROR("Failed to reload level config");
        return false;
    }

    // 重新生成GameModel
    GameModel* newModel = _levelGenerator->generateGameModel(newConfig);
    if (!newModel) {
        CCLOGERROR("Failed to regenerate GameModel");
        delete newConfig;
    }
}

```

```
        return false;
    }

    // 替换旧Model
    delete _gameModel;
    _gameModel = newModel;

    // 重新创建View
    if (_gameView) {
        _gameView->removeFromParent();
    }
    _gameView = GameView::create(_gameModel, _resourceManager);

    // 重新初始化
    setupEventCallbacks();
    startGame();
}
```

## 6. 开发规范

---

### 6.1 代码命名规范

#### 6.1.1 文件命名

类文件:	PascalCase	(例如: GameController.h, CardModel.cpp)
工具函数文件:	PascalCase	(例如: CardGameUtils.h)
常量文件:	PascalCase	(例如: GameConstants.h)
配置文件:	snake_case	(例如: level_1.json, card_config.json)

#### 6.1.2 类命名

```
// ✅ 正确: 类名使用 PascalCase
class GameController { };
class CardModel { };
class UndoManager { };

// ❌ 错误
class gameController { };
class card_model { };
```

#### 6.1.3 变量命名

```
class Example {
public:
    // ✅ 成员变量: 下划线开头 + camelCase
    int _cardId;
    bool _isSelected;
    std::string _cardName;

    // ✅ 局部变量: camelCase
    void someMethod() {
        int localVariable = 0;
    }
}
```

```

    bool hasValue = false;
}

// ✅ 常量: k前缀 + PascalCase
static const int kMaxUndoCount = 50;
static const float kCardWidth = 120.0f;
};

// ❌ 错误示例
class BadExample {
    int cardId;           // ❌ 缺少下划线前缀
    bool IsSelected;      // ❌ 大写开头
    int MAX_VALUE;        // ❌ 应使用 kMaxValue
};

```

## 6.1.4 函数命名

```

class Example {
public:
    // ✅ 公共方法: camelCase
    void startGame();
    bool checkGameRules();
    int getCardId() const;
    void setCardId(int id);

    // ✅ 私有方法: camelCase (可选: 前缀 _ )
private:
    void updateInternalState();
    bool validateData();
};

// ✅ 全局/静态工具函数: camelCase
namespace GameUtils {
    bool isValidCard(CardModel* card);
    int calculateScore(const GameModel* model);
}

// ❌ 错误示例
void StartGame();          // ❌ 大写开头
bool Check_game_rules();   // ❌ 蛇形命名

```

## 6.1.5 枚举命名

```

// ✅ 正确: 枚举类型 PascalCase, 枚举值 UPPER_CASE
enum GameState {
    GAME_STATE_NONE,
    GAME_STATE_PLAYING,
    GAME_STATE_PAUSED
};

enum class Suit { // C++11 强类型枚举 (推荐)
    CLUBS,
    DIAMONDS,
    HEARTS,

```



```

        SPADES
    };

// ❌ 错误示例
enum gamestate {    // ❌ 小写
    None,           // ❌ 应全大写
    playing         // ❌ 应全大写
};

```

## 6.2 注释规范

### 6.2.1 文件头注释

```

/**
 * @file GameController.h
 * @brief 游戏总控制器
 * @author 开发团队
 * @date 2025-10-29
 * @version 1.0
 *
 * @details
 * GameController 是游戏的核心控制器，负责协调 Model 和 View 的交互，
 * 处理用户输入事件，管理游戏生命周期。
 */

#ifndef __GAME_CONTROLLER_H__
#define __GAME_CONTROLLER_H__

// 实现代码...

#endif // __GAME_CONTROLLER_H__

```

### 6.2.2 类注释

```

/**
 * @brief 游戏控制器类
 *
 * 职责：
 * - 管理整个游戏流程和状态
 * - 协调模型和视图之间的交互
 * - 处理用户操作和游戏逻辑
 * - 管理子控制器的生命周期
 *
 * 使用场景：
 * - 游戏初始化、开始、暂停、结束
 * - 处理用户界面交互事件
 * - 协调各个子控制器的协作
 *
 * @code
 * // 创建和初始化游戏控制器
 * GameController* controller = new GameController();
 * controller->initGame(1); // 初始化第1关
 * controller->startGame(); // 开始游戏
 * @endcode
 */

```

```

*/
class GameController {
    // ...
};

```

### 6.2.3 函数注释

```

/**
 * @brief 处理卡牌匹配事件
 *
 * @param playfieldCardId 主牌区卡牌ID
 * @param handCardId 手牌区卡牌ID
 * @return 是否处理成功
 *
 * @details
 * 此方法会执行以下操作：
 * 1. 验证两张卡牌是否可以匹配
 * 2. 记录撤销操作
 * 3. 执行卡牌匹配逻辑
 * 4. 更新分数和UI
 * 5. 检查游戏结束条件
 *
 * @note 此方法会自动播放匹配动画
 * @warning 传入的卡牌ID必须有效，否则返回false
 *
 * @see GameRulesService::canCardsMatch()
 * @see UndoManager::addUndoAction()
 */
bool handleCardMatch(int playfieldCardId, int handCardId);

```

### 6.2.4 成员变量注释

```

class GameModel {
public:
    int _levelId;           ///< 关卡ID
    GameState _gameState;   ///< 游戏状态
    float _gameTime;        ///< 游戏时间（秒）
    int _score;             ///< 当前分数

    /**
     * @brief 所有卡牌的集合
     *
     * 包含游戏中所有卡牌的指针，负责卡牌的生命周期管理。
     * 在析构函数中会删除所有卡牌对象。
     */
    std::vector<CardModel*> _allCards;
};

```

## 6.3 代码风格规范

### 6.3.1 缩进与空格

```
// ✔ 正确：使用4个空格缩进
class Example {
public:
    void method() {
        if (condition) {
            doSomething();
        }
    }
};

// ✔ 运算符两侧加空格
int result = a + b;
bool isValid = (x > 0) && (y < 100);

// ✔ 逗号后加空格
void func(int a, int b, int c);

// ✘ 错误：使用Tab或2个空格
class BadExample {
    void method() { // ✘ 2个空格
        doSomething(); // ✘ Tab
    }
};
```

## 6.3.2 大括号风格

```
// ✔ 正确：K&R风格（推荐）
void method() {
    if (condition) {
        doSomething();
    } else {
        doOtherThing();
    }
}

class Example {
public:
    void method();
};

// ⚠ 可接受：Allman风格（可选）
void method()
{
    if (condition)
    {
        doSomething();
    }
}


// ✘ 错误：不一致的风格
void method() {
    if (condition) // ✘ 混合风格
    {
        doSomething();
    } else {
```

```

        doOtherThing();
    }
}

```

### 6.3.3 行长度限制

//  正确：每行不超过100个字符

```

void createCard(CardModel::Suit suit,
               CardModel::Rank rank,
               int cardId);


```

//  长条件语句换行

```

if (card->_isFaceUp &&
    card->_isMovable &&
    !card->_isSelected) {
    processCard(card);
}

```

//  错误：单行过长


```

void createCard(CardModel::Suit suit, CardModel::Rank rank, int cardId, const Vec2& position, bool isFace

```

## 6.4 内存管理规范

### 6.4.1 对象生命周期管理

//  正确：明确所有权

```

class GameController {
public:
    ~GameController() {
        // 释放自己创建的对象
        delete _gameModel;      // 拥有所有权
        delete _undoManager;    // 拥有所有权


        // Cocos2d-x对象使用引用计数
        CC_SAFE_RELEASE_NULL(_resourceManager);

        // View由场景管理，只需置空
        _gameView = nullptr;
    }

private:
    GameModel* _gameModel;      // 拥有所有权（需要delete）
    GameView* _gameView;       // 不拥有（由父节点管理）
    ResourceManager* _resourceManager; // Cocos对象（引用计数）
    UndoManager* _undoManager;  // 拥有所有权（需要delete）
};

```

### 6.4.2 避免内存泄漏

//  正确：使用智能指针（C++11）

```

class ModernGameController {
private:

```

```

std::unique_ptr<GameModel> _gameModel;
std::unique_ptr<UndoManager> _undoManager;
std::shared_ptr<LevelConfig> _levelConfig;
};

// ✅ 正确：容器管理对象生命周期
class GameModel {
public:
    ~GameModel() {
        // 清理所有卡牌
        for (auto card : _allCards) {
            delete card;
        }
        _allCards.clear();
    }

private:
    std::vector<CardModel*> _allCards; // 拥有所有权
};

// ❌ 错误：内存泄漏
void badFunction() {
    CardModel* card = new CardModel();
    // ... 使用card ...
    // ❌ 忘记delete card
}

// ✅ 正确：及时释放
void goodFunction() {
    CardModel* card = new CardModel();
    // ... 使用card ...
    delete card; // ✅ 及时释放
}

```

### 6.4.3 避免野指针

```

// ✅ 正确：删除后置空
class Example {
public:
    void cleanup() {
        delete _gameModel;
        _gameModel = nullptr; // ✅ 置空，避免野指针

        if (_gameView) {
            _gameView->removeFromParent();
            _gameView = nullptr;
        }
    }

    void safeAccess() {
        if (_gameModel) { // ✅ 使用前检查
            _gameModel->updateScore(100);
        }
    }
};

```

## 6.5 错误处理规范

### 6.5.1 参数验证

```
// ✔ 正确：函数入口验证参数
bool GameController::handleCardMatch(int playfieldCardId, int handCardId) {
    // 验证游戏状态
    if (!_isGameRunning || !_gameModel) {
        CCLOGWARN("Game is not running or model is null");
        return false;
    }

    // 验证参数
    CardModel* playfieldCard = _gameModel->getCardById(playfieldCardId);
    CardModel* handCard = _gameModel->getCardById(handCardId);

    if (!playfieldCard || !handCard) {
        CCLOGERROR("Invalid card ID: playfield=%d, hand=%d",
            playfieldCardId, handCardId);
        return false;
    }

    // 执行逻辑
    // ...
}
```

### 6.5.2 日志输出

```
// ✔ 使用Cocos2d-x日志系统
CCLOG("Info: Game started, level=%d", levelId);
CCLOGWARN("Warning: Low memory detected");
CCLOGERROR("Error: Failed to load level config, id=%d", levelId);

// ✔ 调试信息（Release版本自动去除）
#ifdef COCOS2D_DEBUG
    CCLOG("Debug: Card position=(%f, %f)", card->_position.x, card->_position.y);
#endif

// ✘ 错误：使用printf
printf("This is bad\n"); // ✘ 不要使用printf
```

### 6.5.3 异常处理

```
// ✔ 正确：通过返回值处理错误
bool GameController::initGame(int levelId) {
    if (levelId <= 0) {
        CCLOGERROR("Invalid level ID");
        return false; // ✔ 返回false表示失败
    }

    LevelConfig* config = _levelConfigLoader->loadLevelConfig(levelId);
    if (!config) {
        CCLOGERROR("Failed to load level config");
    }
}
```

```
        return false;
    }

    // ... 其他初始化 ...
    return true;
}

// ✖ 避免：在游戏中使用异常（性能考虑）
// C++异常在游戏中会影响性能，尽量避免使用
void badFunction() {
    try {
        // ...
    } catch (std::exception& e) {
        // ...
    }
}
```

## 扑克消除游戏 - 程序设计文档（第三部分）

---

### 7. 测试与调试

---

#### 7.1 单元测试

##### 7.1.1 测试框架选择

推荐使用 **Google Test (gtest)** 进行单元测试。

**安装gtest (CMake项目)：**

```
# CMakeLists.txt

# 添加gtest
include(FetchContent)
FetchContent_Declare(
    googletest
    URL https://github.com/google/googletest/archive/release-1.12.1.zip
)
FetchContent_MakeAvailable(googletest)

enable_testing()

# 创建测试可执行文件
add_executable(
    card_game_test
    tests/CardModelTest.cpp
    tests/GameRulesServiceTest.cpp
    tests/UndoManagerTest.cpp
)

target_link_libraries(
    card_game_test
    GTest::gtest_main
    card_game_lib # 你的游戏库
```

```
)
```

```
include(GoogleTest)
gtest_discover_tests(card_game_test)
```

## 7.1.2 Model 层测试

### 测试 CardModel:

```
// tests/CardModelTest.cpp

#include <gtest/gtest.h>
#include "models/CardModel.h"

class CardModelTest : public ::testing::Test {
protected:
    void SetUp() override {
        card = new CardModel(CardModel::HEARTS, CardModel::ACE, 1);
    }

    void TearDown() override {
        delete card;
    }

    CardModel* card;
};

// 测试卡牌基本属性
TEST_F(CardModelTest, BasicProperties) {
    EXPECT_EQ(card->_cardId, 1);
    EXPECT_EQ(card->_suit, CardModel::HEARTS);
    EXPECT_EQ(card->_rank, CardModel::ACE);
}

// 测试花色判断
TEST_F(CardModelTest, SuitColor) {
    EXPECT_TRUE(card->isRedSuit());
    EXPECT_FALSE(card->isBlackSuit());

    CardModel blackCard(CardModel::SPADES, CardModel::KING, 2);
    EXPECT_TRUE(blackCard.isBlackSuit());
    EXPECT_FALSE(blackCard.isRedSuit());
}

// 测试翻转功能
TEST_F(CardModelTest, FlipCard) {
    EXPECT_FALSE(card->_isFaceUp);

    card->flipCard();
    EXPECT_TRUE(card->_isFaceUp);

    card->flipCard();
    EXPECT_FALSE(card->_isFaceUp);
}

// 测试卡牌值计算
```



```
GoodModel = .../GoodModel - HEARTC - GoodModel - AGE - 1 \.
```

### 7.1.3 Service 层测试

## 测试 GameRulesService:

```
// tests/GameRulesServiceTest.cpp

#include <gtest/gtest.h>
#include "services/GameRulesService.h"
#include "models/CardModel.h"
#include "models/GameModel.h"

class GameRulesServiceTest : public ::testing::Test {
protected:
    void SetUp() override {
        gameModel = new GameModel();
        gameModel->initGame(1);
    }

    void TearDown() override {
        delete gameModel;
    }

    GameModel* gameModel;
};

// 测试卡牌匹配规则
TEST_F(GameRulesServiceTest, CanCardsMatch) {
    CardModel* card1 = new CardModel(CardModel::HEARTS, CardModel::ACE, 1);
    CardModel* card2 = new CardModel(CardModel::SPADES, CardModel::TWO, 2);
    CardModel* card3 = new CardModel(CardModel::DIAMONDS, CardModel::FIVE, 3);

    // ACE (0) 和 TWO (1) 相差1, 应该可以匹配
    EXPECT_TRUE(GameRulesService::canCardsMatch(card1, card2));

    // ACE (0) 和 FIVE (4) 相差4, 不应该匹配
    EXPECT_FALSE(GameRulesService::canCardsMatch(card1, card3));

    // TWO (1) 和 TWO (1) 相差0, 不应该匹配
    CardModel* card4 = new CardModel(CardModel::CLUBS, CardModel::TWO, 4);
    EXPECT_FALSE(GameRulesService::canCardsMatch(card2, card4));

    delete card1;
    delete card2;
    delete card3;
    delete card4;
}

// 测试K和A是否可以匹配
TEST_F(GameRulesServiceTest, KingAndAceMatch) {
    CardModel* king = new CardModel(CardModel::HEARTS, CardModel::KING, 1);
    CardModel* ace = new CardModel(CardModel::SPADES, CardModel::ACE, 2);

    // KING (12) 和 ACE (0) 相差12, 不应该匹配 (当前规则)
```

## 7.1.4 Manager 层测试

测试 UndoManager:

```
// tests/UndoManagerTest.cpp

#include <gtest/gtest.h>
#include "managers/UndoManager.h"
#include "models/GameModel.h"
#include "models/UndoModel.h"

class UndoManagerTest : public ::testing::Test {
protected:
    void SetUp() override {
        gameModel = new GameModel();
        gameModel->initGame(1);

        undoManager = new UndoManager();
        undoManager->init(gameModel, 10);
    }

    void TearDown() override {
        delete undoManager;
        delete gameModel;
    }

    GameModel* gameModel;
    UndoManager* undoManager;
};

// 测试添加撤销操作
TEST_F(UndoManagerTest, AddUndoAction) {
    EXPECT_EQ(undoManager->getUndoCount(), 0);

    UndoModel* action = new UndoModel(
        UndoModel::ACTION_MOVE_CARD,
        1,
        "playfield",
        "hand"
    );

    EXPECT_TRUE(undoManager->addUndoAction(action));
    EXPECT_EQ(undoManager->getUndoCount(), 1);
    EXPECT_TRUE(undoManager->canUndo());
}

// 测试撤销操作
TEST_F(UndoManagerTest, UndoOperation) {
    // 创建一张卡牌
    CardModel* card = new CardModel(CardModel::HEARTS, CardModel::ACE, 1);
    card->_cardArea = "playfield";
    card->_position = Vec2(100, 200);
    gameModel->allCards.push_back(card);
```

## 7.2 集成测试

## 7.2.1 完整游戏流程测试

```
// tests/IntegrationTest.cpp

#include <gtest/gtest.h>
#include "controllers/GameController.h"
#include "managers/GameModelGenerator.h"

class GameIntegrationTest : public ::testing::Test {
protected:
    void SetUp() override {
        controller = new GameController();
    }

    void TearDown() override {
        delete controller;
    }

    GameController* controller;
};

// 测试完整游戏流程
TEST_F(GameIntegrationTest, FullGameFlow) {
    // 初始化游戏
    EXPECT_TRUE(controller->initGame(1));

    // 开始游戏
    EXPECT_TRUE(controller->startGame());
    EXPECT_FALSE(controller->isGameFinished());

    // 模拟玩家操作
    GameModel* model = controller->getGameModel();
    ASSERT_NE(model, nullptr);

    // 检查初始状态
    EXPECT_GT(model->_playfieldCards.size(), 0);
    EXPECT_GT(model->_handCards.size(), 0);

    // 暂停游戏
    controller->pauseGame();
    EXPECT_EQ(model->_gameState, GameModel::GAME_STATE_PAUSED);

    // 恢复游戏
    controller->resumeGame();
    EXPECT_EQ(model->_gameState, GameModel::GAME_STATE_PLAYING);

    // 结束游戏
    controller->endGame();
    EXPECT_TRUE(controller->isGameFinished());
}
```

## 7.3 性能测试

### 7.3.1 性能基准测试

```

// tests/PerformanceTest.cpp

#include <gtest/gtest.h>
#include <chrono>
#include "services/GameRulesService.h"
#include "managers/GameModelGenerator.h"

class PerformanceTest : public ::testing::Test {
protected:
    void measureExecutionTime(std::function<void()> func, const std::string& name) {
        auto start = std::chrono::high_resolution_clock::now();
        func();
        auto end = std::chrono::high_resolution_clock::now();

        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        std::cout << name << " took " << duration.count() << " microseconds" << std::endl;
    }
};

// 测试卡牌匹配性能
TEST_F(PerformanceTest, CardMatchingPerformance) {
    CardModel* card1 = new CardModel(CardModel::HEARTS, CardModel::ACE, 1);
    CardModel* card2 = new CardModel(CardModel::SPADES, CardModel::TWO, 2);

    measureExecutionTime([&]() {
        for (int i = 0; i < 100000; i++) {
            GameRulesService::canCardsMatch(card1, card2);
        }
    }, "100k card matches");

    delete card1;
    delete card2;
}

// 测试洗牌性能
TEST_F(PerformanceTest, ShufflePerformance) {
    std::vector<CardModel*> cards;
    for (int i = 0; i < 52; i++) {
        cards.push_back(new CardModel(
            static_cast<CardModel::Suit>(i % 4),
            static_cast<CardModel::Rank>(i % 13),
            i + 1
        ));
    }

    measureExecutionTime([&]() {
        for (int i = 0; i < 1000; i++) {
            GameRulesService::shuffleCards(cards);
        }
    }, "1000 shuffle operations");
}

```

## 7.4 调试技巧

### 7.4.1 启用详细日志

```

// Classes/utils/DebugUtils.h

#ifndef __DEBUG_UTILS_H__
#define __DEBUG_UTILS_H__

#include "cocos2d.h"

#if COCOS2D_DEBUG

// 调试宏：打印变量
#define DEBUG_VAR(var) CCLOG("[DEBUG] %s = %s", #var, std::to_string(var).c_str())

// 调试宏：打印位置
#define DEBUG_POS(pos) CCLOG("[DEBUG] %s = (%.2f, %.2f)", #pos, pos.x, pos.y)

// 调试宏：打印卡牌信息
#define DEBUG_CARD(card) \
    CCLOG("[DEBUG] Card[%d]: %s, pos=(%.2f, %.2f), faceUp=%d", \
        card->_cardId, card->getCardName().c_str(), \
        card->_position.x, card->_position.y, card->_isFaceUp)

// 调试宏：打印游戏状态
#define DEBUG_GAME_STATE(model) \
    CCLOG("[DEBUG] Game State: score=%d, playfield=%zu, stack=%zu, hand=%zu", \
        model->_score, model->_playfieldCards.size(), \
        model->_stackCards.size(), model->_handCards.size())

#else

#define DEBUG_VAR(var)
#define DEBUG_POS(pos)
#define DEBUG_CARD(card)
#define DEBUG_GAME_STATE(model)

#endif

#endif // __DEBUG_UTILS_H__

```

## 使用示例:

```

#include "utils/DebugUtils.h"

void GameController::handleCardMatch(int cardId1, int cardId2) {
    CardModel* card1 = _gameModel->getCardById(cardId1);
    CardModel* card2 = _gameModel->getCardById(cardId2);

    DEBUG_CARD(card1); // 只在Debug模式下输出
    DEBUG_CARD(card2);

    if (GameRulesService::canCardsMatch(card1, card2)) {
        CCLOG("Cards can match!");
        DEBUG_GAME_STATE(_gameModel);

        // 执行匹配...
        GameRulesService::executeCardMatch(_gameModel, cardId1, cardId2);
    }
}

```

```

        DEBUG_GAME_STATE(_gameModel);
    }
}

```

## 7.4.2 可视化调试工具

```

// Classes/utils/VisualDebugger.h

```

```

class VisualDebugger {
public:
    // 显示卡牌ID标签
    static void showCardIds(Layer* layer, const std::vector<CardView*>& cardViews);

    // 显示碰撞边界
    static void showBoundingBoxes(Layer* layer, const std::vector<CardView*>& cardViews);

    // 显示卡牌区域分隔线
    static void showAreaBorders(Layer* layer);

    // 显示触摸点
    static void showTouchPoints(Layer* layer);
};

void VisualDebugger::showCardIds(Layer* layer, const std::vector<CardView*>& cardViews) {
#ifdef COCOS2D_DEBUG
    for (CardView* cardView : cardViews) {
        Label* idLabel = Label::createWithSystemFont(
            StringUtils::format("%d", cardView->getCardId()),
            "Arial",
            20
        );
        idLabel->setPosition(cardView->getPosition());
        idLabel->setColor(Color3B::YELLOW);
        idLabel->setTag(999); // 特殊tag用于清理
        layer->addChild(idLabel, 1000);
    }
#endif
}

void VisualDebugger::showBoundingBoxes(Layer* layer, const std::vector<CardView*>& cardViews) {
#ifdef COCOS2D_DEBUG
    for (CardView* cardView : cardViews) {
        Rect bbox = cardView->getBoundingBox();

        DrawNode* drawNode = DrawNode::create();
        Vec2 vertices[] = {
            Vec2(bbox.getMinX(), bbox.getMinY()),
            Vec2(bbox.getMaxX(), bbox.getMinY()),
            Vec2(bbox.getMaxX(), bbox.getMaxY()),
            Vec2(bbox.getMinX(), bbox.getMaxY())
        };
        drawNode->drawPolygon(vertices, 4, Color4F(0, 0, 0, 0), 2, Color4F(1, 0, 0, 1));
        drawNode->setTag(999);
        layer->addChild(drawNode, 1000);
    }
}

```

## 在GameView中使用:

```
void GameView::updateGameDisplay() {
    // 清除之前的调试信息
    this->removeChildByTag(999);

    #if COCOS2D_DEBUG
        // 显示调试信息
        std::vector<CardView*> cardViewList;
        for (auto& pair : _cardViews) {
            cardViewList.push_back(pair.second);
        }

        VisualDebugger::showCardIds(this, cardViewList);
        VisualDebugger::showBoundingBoxes(this, cardViewList);
    #endif
}
```

## 7.4.3 内存泄漏检测

### 使用 Valgrind (Linux/Mac):

```
# 编译Debug版本
cmake -DCMAKE_BUILD_TYPE=Debug ..
make

# 使用Valgrind检测内存泄漏
valgrind --leak-check=full \
    --show-leak-kinds=all \
    --track-origins=yes \
    --verbose \
    --log-file=valgrind-out.txt \
    ./bin/TemplateCpp
```

### 使用 Visual Studio 内存检测 (Windows):

```
// 在 main.cpp 开头添加
#ifdef _WIN32 && defined(_DEBUG)
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#endif

int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPTSTR lpCmdLine, int nCmdShow) {
    #if defined(_DEBUG)
        // 启用内存泄漏检测
        _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);

        // 设置内存泄漏报告输出到VS输出窗口
        _CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_DEBUG);
    #endif

    // 运行游戏
```

```
AppDelegate app;  
return Application::getInstance()->run();  
}
```

## 8. 常见问题FAQ

---

### 8.1 架构相关

#### Q1: 为什么使用MVC而不是MVVM?

A: 因为Cocos2d-x不是数据绑定框架，手动实现双向绑定会增加复杂度。MVC模式通过Controller手动协调Model和View，更适合游戏开发的实时性需求。

#### Q2: Service层和Manager层有什么区别?

A:

- **Service:** 无状态的工具类，只提供静态方法，不持有任何数据
- **Manager:** 有状态的管理器，可以持有数据（如UndoManager持有历史栈）

#### Q3: 为什么View层使用 *const GameModel* ? \*

A: 强制View只读取Model数据，不直接修改。所有修改必须通过Controller调用Service完成，保证数据流向单一。

### 8.2 卡牌系统

#### Q4: 如何添加特殊效果的卡牌（如炸弹、冰冻）？

A: 在CardModel中添加 `_specialType` 字段，在游戏RulesService中实现 `executeSpecialCard()` 方法，详见扩展指南 4.1.2。

#### Q5: 卡牌ID是如何分配的?

A: 由GameModelGenerator在生成GameModel时按顺序分配，从1开始递增。每张卡牌的ID在一局游戏中唯一。

#### Q6: 卡牌的ZOrder如何管理?

A:

- 主牌区: 50
- 备用牌堆: 30
- 手牌区: 100起，每次+10（确保最新手牌在最上层）

### 8.3 撤销系统

#### Q7: 撤销系统支持哪些操作类型?

A: 当前支持：



- ACTION\_MOVE\_CARD: 移动卡牌
- ACTION\_FLIP\_CARD: 翻转卡牌
- ACTION\_DEAL\_CARD: 发牌
- ACTION\_SELECT\_CARD: 选中卡牌

扩展方法见扩展指南 4.2。

#### Q8: 撤销历史占用多少内存?

A: 每个UndoModel约 100-200 字节，默认最大50次撤销，总计约 10KB，可忽略不计。

#### Q9: 如何实现"撤销到指定步骤"功能?

A:

```
bool UndoManager::undoToStep(int stepIndex) {  
    while (getUndoCount() > stepIndex && canUndo()) {  
        if (!undo()) return false;  
    }  
    return true;  
}
```

## 8.4 配置系统

#### Q10: 配置文件路径在哪里?

A:

- 关卡配置: configs/levels/level\_X.json
- 卡牌资源配置: Resources/cards/card\_config.json

#### Q11: 如何动态加载关卡配置?

A: 使用LevelConfigLoader::loadLevelConfig(levelId)，会自动从configs/levels/目录加载对应JSON文件。

#### Q12: 配置文件格式错误会崩溃吗?

A: 不会。LevelConfigValidator会验证配置，失败时返回默认配置并记录错误日志。

## 8.5 性能优化

#### Q13: 游戏卡顿怎么办?

A:

1. 检查是否有过多的动态内存分配
2. 减少每帧的卡牌视图更新次数
3. 使用对象池复用CardView
4. 减少日志输出（Release模式）

#### Q14: 如何优化卡牌动画性能?

A:

```
// 使用动作缓存
auto moveAction = MoveTo::create(0.3f, targetPos);
cardView->runAction(moveAction);

// 批量更新位置（避免逐个更新）
for (CardView* view : cardViews) {
    view->setPosition(newPositions[view->getCardId()]);
}
```

### Q15: 内存占用过高怎么办？

A:

1. 检查是否有内存泄漏（使用Valgrind）
2. 及时释放不用的资源
3. 使用纹理图集减少内存占用
4. 限制撤销历史大小

## 8.6 调试相关

### Q16: 如何查看当前游戏状态？

A: 使用调试工具：

```
#include "utils/DebugUtils.h"
DEBUG_GAME_STATE(_gameModel); // 输出分数、卡牌数量等
```

### Q17: 如何定位卡牌点击不响应的问题？

A:

1. 检查CardView的触摸监听器是否正确添加
2. 使用 VisualDebugger::showBoundingBoxes() 查看碰撞区域
3. 检查ZOrder是否正确（是否被其他View遮挡）
4. 检查 \_isMovable 和 \_isVisible 状态

### Q18: 如何调试JSON解析错误？

A:

```
rapidjson::Document doc;
doc.Parse(jsonStr.c_str());
if (doc.HasParseError()) {
    CCLOGERROR("JSON parse error at offset %zu: %s",
               doc.GetErrorOffset(),
               rapidjson::GetParseError_En(doc.GetParseError()));
}
```

## 8.7 扩展开发

### Q19: 如何添加新的游戏模式？

A:

- 1. 创建新的关卡配置JSON文件
- 2. 在GameRulesService中添加新规则判断方法
- 3. 在GameController中处理新模式的逻辑
- 4. 更新UI显示

Q20: 如何支持多语言?

A:

- 1. 使用Cocos2d-x的本地化系统
- 2. 将所有字符串提取到语言包文件
- 3. 使用 LocalizedString::get("key") 获取翻译
- 4. 在配置文件中使用的翻译key而不是直接的文本

附录

A. 文件清单

A.1 核心代码文件

```
Classes/  
├── models/                                (数据模型, 约800行代码)  
│   ├── CardModel.h/cpp                  (约300行)  
│   ├── GameModel.h/cpp                  (约500行)  
│   └── UndoModel.h/cpp                  (约350行)  
│  
├── views/                                (视图层, 约1200行代码)  
│   ├── CardView.h/cpp                  (约400行)  
│   └── GameView.h/cpp                  (约800行)  
│  
├── controllers/                          (控制器, 约1200行代码)  
│   └── GameController.h/cpp            (约1200行)  
│  
├── services/                             (服务层, 约200行代码)  
│   └── GameRulesService.h/cpp          (约200行)  
│  
├── managers/                             (管理器, 约900行代码)  
│   ├── UndoManager.h/cpp              (约450行)  
│   ├── ResourceManager.h/cpp          (约300行)  
│   └── GameModelGenerator.h/cpp        (约350行)  
│  
├── configs/                              (配置系统, 约400行代码)  
│   ├── models/LevelConfig.h/cpp  
│   └── loaders/LevelConfigLoader.h/cpp  
│  
└── utils/                                (工具类, 约200行代码)  
    ├── GameConstants.h  
    └── CardGameUtils.h/cpp
```

总计代码量：约 5,000 行（不含注释和空行）

## A.2 配置文件

```
configs/levels/  
└─ level_1.json          (关卡配置)  
  
Resources/cards/  
└─ card_config.json      (卡牌资源配置)
```

## B. 术语表

术语	英文	说明
主牌区	Playfield	竖直堆叠的卡牌区域，游戏目标是清空此区域
手牌区	Hand	当前可用于匹配的卡牌
备用牌堆	Stack	可抽取的备用卡牌
花色	Suit	梅花、方块、红桃、黑桃
点数	Rank	A, 2-10, J, Q, K
撤销	Undo	回退上一步操作
重做	Redo	重新执行撤销的操作
ZOrder	Z-Order	渲染层级，数值越大越靠前
MVC	Model-View-Controller	模型-视图-控制器架构模式
Service	Service	无状态的业务逻辑服务
Manager	Manager	有状态的功能管理器

## C. 参考资料

- 1. Cocos2d-x官方文档: <https://docs.cocos.com/cocos2d-x/manual/>
- 2. RapidJSON文档: <https://rapidjson.org/>
- 3. Google Test文档: <https://google.github.io/googletest/>
- 4. C++ Core Guidelines: <https://isocpp.github.io/CppCoreGuidelines/>

## 结语

本文档详细介绍了扑克消除游戏的架构设计、代码结构、扩展方法和开发规范。遵循本文档的设计原则和编码规范，可以：

- ✅ **快速上手:** 清晰的架构和代码组织，新开发者可以快速理解项目
- ✅ **易于扩展:** 模块化设计，添加新功能（卡牌、规则、撤销类型）简单明了

- ✓ **易于维护:** 职责分离, 修改某个模块不影响其他部分
- ✓ **易于测试:** 清晰的接口, 支持单元测试和集成测试
- ✓ **高质量代码:** 遵循业界最佳实践, 代码规范统一

**未来扩展方向:**

- 多人对战模式
- 关卡编辑器
- 成就系统
- 排行榜系统
- 更多特殊卡牌
- 自定义皮肤主题

**技术支持:**

如有任何问题, 请参考本文档或查阅相关技术文档。

**文档版本:** v1.0  
**最后更新:** 2025-10-29  
**维护人:** 王雅楠