# cocos2dx card\_test项目程序交付文档

## 一、项目概述

### 1.1 项目背景

本项目是一款基于cocos2d-x引擎开发的卡牌游戏，核心玩法围绕卡牌的移动、匹配及回退操作展开。游戏通过模型-视图-控制器（MVC）架构实现数据与界面的分离，确保代码的可维护性和可扩展性。

### **1.2 技术栈**

- 引擎：cocos2d-x（2D游戏开发引擎，提供图形渲染、事件处理等核心功能）

- 语言：C++（核心逻辑开发）

- 架构：MVC架构（模型层负责数据存储，视图层负责界面展示，控制器层负责业务逻辑）

## **二、系统架构设计**

### **2.1 整体架构**

项目采用MVC架构，各层职责如下：

- 模型层（Models）：存储游戏核心数据（如卡牌属性、游戏状态、回退记录等），不依赖视图和控制器。

- 视图层（Views）：负责界面渲染和用户交互展示，通过控制器与模型层交互。

- 控制器层（Controllers）：协调模型层和视图层，处理业务逻辑（如卡牌移动、回退操作等）。

- 管理器（Managers）：提供通用功能支持（如回退管理）。

- 配置（Configs）：存储静态配置数据（如卡牌资源路径、关卡配置等）。

### **2.2 模块关系图**



## **三、模块详细设计**

### 3.1 模型层（Models）

#### 3.1.1 CardModel（卡牌数据模型）

- 功能：存储单张卡牌的核心属性（如牌面、花色、位置、ID等）。

- 核心成员：

- \_face：牌面类型（CardFaceType，如A、2...K）。

- \_suit：花色类型（CardSuitType，如红桃、黑桃等）。

- \_position：卡牌位置（Vec2，坐标系位置）。

- \_id：卡牌唯一标识（用于区分不同卡牌）。

- \_playIndex：所属游戏区牌堆索引（关联到PlayFieldPile）。

- 核心方法：

- getValue()：返回牌面数值（如A对应1，K对应13）。

- getPosition()/setPosition()：获取/设置卡牌位置。

#### **3.1.2 GameModel（游戏状态模型）**

- 功能：存储整个游戏的状态（如游戏区牌堆、牌堆区域等）。

- 核心成员：

- \_playFieldPiles：游戏区牌堆集合（std::vector<PlayFieldPile>），每个牌堆包含一叠卡牌（std::deque<CardModel>）。

- \_stackArea：牌堆区域（包含待发牌堆deck、顶部展示牌topCard及位置信息）。

- 核心方法：

- pushPlayFieldByIndex()：向指定游戏区牌堆添加卡牌。

- pushStackArea()：向牌堆区域添加卡牌。

- stackMoveToStackTop()：将牌堆顶部卡牌移至展示区。

- findCardAtPosition()：根据位置查找卡牌。

#### **3.1.3 UndoModel（回退数据模型）**

- 功能：存储卡牌操作的历史记录，支持回退功能。

- 核心成员：

- \_model：回退记录队列（std::deque<CardModel>），存储每次操作前的卡牌状态。

- 核心方法：

- push()：添加一条回退记录。

- undo()：取出最近一条回退记录并移除。

- peek()：查看最近一条回退记录（不移除）。

### 3.2 视图层（Views）

#### **3.2.1 CardView（卡牌视图）**

- 功能：渲染单张卡牌的视觉效果，处理卡牌点击事件。

- 核心成员：

- \_tagId：关联的CardModel的ID（用于映射数据模型）。

- \_onClickCallback：点击事件回调（通知控制器处理点击逻辑）。

- 核心方法：

- create(const CardModel& model)：根据卡牌模型创建视图。

- moveToPosition()：动画移动卡牌到指定位置。

- shake()：卡牌抖动动画（用于错误提示）。

#### **3.2.2 PlayFieldView（游戏区视图）**

- 功能：展示游戏区的所有牌堆，处理游戏区卡牌的交互。

- 核心成员：

- \_pileCardTagIds：记录每个牌堆中卡牌的tagId（用于查找视图对应的模型）。

- 核心方法：

- setUpWithCardModels()：根据GameModel初始化游戏区视图。

- moveToStackTop()：将指定牌堆的卡牌移动到牌堆顶部（带动画）。

- unMatched()：卡牌不匹配时的视觉反馈（如抖动）。

#### **3.2.3 StackView（牌堆视图）**

- 功能：展示牌堆区域的卡牌（待发牌堆和顶部展示牌）。

- 核心方法：

- setUpWithCardModels()：根据GameModel初始化牌堆视图。

- moveToStackTop()：将牌堆中的卡牌移至顶部展示区（带动画）。

### 3.3 控制器层（Controllers）

#### 3.3.1 GameController（游戏主控制器）

- 功能：协调游戏各模块，管理游戏生命周期（如启动游戏、回退操作等）。

- 核心成员：

- \_gameModel：当前游戏状态模型。

- \_undoManager：回退管理器实例。

- \_playFieldController/\_stackController：游戏区/牌堆控制器。

- 核心方法：

- startGame(int levelId)：根据关卡ID启动游戏（加载配置、初始化模型和视图）。

- undoOperation()：执行回退操作（调用UndoManager）。

#### **3.3.2 PlayFieldController（游戏区控制器）**

- 功能：处理游戏区卡牌的移动逻辑（如移动到牌堆顶部）。

- 核心方法：

- moveToStackTop()：验证卡牌移动合法性（如牌面差值是否为1），更新模型并通知视图刷新。

- moveUndo()：回退游戏区卡牌操作（恢复卡牌位置和状态）。

#### **3.3.3 StackController（牌堆控制器）**

- 功能：处理牌堆区域的卡牌移动逻辑（如从待发牌堆移至顶部展示区）。

- 核心方法：

- moveToStackTop()：验证牌堆卡牌移动合法性，更新模型并通知视图刷新。

- moveUndo()：回退牌堆卡牌操作。

### 3.4 管理器（Managers）

#### **3.4.1 UndoManager（回退管理器）**

- 功能：统一管理回退操作，协调模型层和视图层的回退逻辑。

- 核心方法：

- record(const CardModel& model)：记录卡牌操作前的状态（调用UndoModel::push()）。

- undo()：执行回退（调用UndoModel::undo()，并通知控制器更新视图）。

- canUndo()：判断是否有可回退的记录。

### **3.5 配置层（Configs）**

#### 3.5.1 CardResConfig（卡牌资源配置）

- 功能：根据CardModel生成卡牌资源路径（如牌面图片、花色图片）。

- 核心方法：

- generateBigNumberPath()/generateSuitPath()：生成大数字、花色图片的资源路径。

#### **3.5.2 LevelConfig（关卡配置）**

- 功能：存储关卡初始卡牌配置（游戏区和牌堆的初始卡牌）。

- 核心成员：

- playfield：游戏区初始卡牌配置（std::vector<CardConfig>）。

- stack：牌堆初始卡牌配置（std::vector<CardConfig>）。

## 四、核心功能实现流程

### 4.1 卡牌初始化流程

1. 加载关卡配置：GameController::startGame()调用LevelConfigLoader加载关卡配置（LevelConfig）。

2. 生成卡牌模型：根据LevelConfig中的CardConfig，通过GameModelFromLevelGenerator生成CardModel实例，添加到GameModel的\_playFieldPiles和\_stackArea。

3. 初始化视图：PlayFieldController和StackController分别调用PlayFieldView::setUpWithCardModels()和StackView::setUpWithCardModels()，根据GameModel创建卡牌视图。

### 4.2 卡牌移动流程（以游戏区卡牌移至牌堆顶部为例）

1. 用户交互：CardView触发点击事件，通过\_onClickCallback通知PlayFieldController。

2. 合法性验证：PlayFieldController::moveToStackTop()检查卡牌是否为牌堆顶部（tagId是否匹配）、牌面是否与目标牌堆顶部卡牌差值为1。

3. 记录操作：通过UndoManager::record()保存移动前的卡牌状态。

4. 更新模型：从游戏区牌堆移除卡牌，更新StackArea的topCard和topNums。

5. 刷新视图：调用PlayFieldView::moveToStackTop()，通过动画展示卡牌移动效果。

### 4.3 回退操作流程

1. 触发回退：用户点击回退按钮，GameController::undoOperation()调用UndoManager::undo()。

2. 获取回退记录：UndoManager从UndoModel中取出最近的CardModel记录。

3. 恢复模型：PlayFieldController::moveUndo()或StackController::moveUndo()根据回退记录恢复卡牌的位置和状态。

4. 刷新视图：视图层通过moveUndo()方法执行动画，恢复卡牌的视觉位置。

## 五、扩展指南

### 5.1 新增一张卡牌

#### 5.1.1 步骤1：扩展卡牌模型

若新卡牌需要新增属性（如特殊技能），修改CardModel：

// CardModel.h

struct CardModel {

// 原有属性

CardFaceType \_face;

CardSuitType \_suit;

Vec2 \_position;

int \_id;

int \_playIndex;

// 新增属性：特殊技能标识

bool \_hasSpecialSkill;

// 新增方法：获取技能状态

bool hasSpecialSkill() const noexcept { return \_hasSpecialSkill; }

};

#### **5.1.2 步骤2：配置卡牌资源**

在CardResConfig中添加新卡牌的资源路径（若牌面/花色为新类型）：

// CardResConfig.h

// 扩展CARD\_FACES或CARD\_SUITS（若需要新牌面/花色）

static constexpr const char\* CARD\_FACES[14] = { // 原13种+新牌面

"A", "2", ..., "K", "Joker" // 示例：新增 Joker

};

// 生成资源路径时兼容新牌面

std::string generateBigNumberPath(const CardModel& model) {

return std::string("res/number/big\_") + getColor(model) + "\_" + CARD\_FACES[model.getFace()] + ".png";

}

#### **5.1.3 步骤3：添加关卡配置**

在LevelConfig中新增卡牌的初始配置（以关卡1为例）：

// 假设关卡配置文件中添加新卡牌

LevelConfig level1Config;

level1Config.playfield.push\_back({

.face = CF\_JOKER, // 新牌面

.suit = CST\_HEARTS,

.position = Vec2(300, 400)

});

#### **5.1.4 步骤4：适配视图层**

若新卡牌有特殊视觉效果，修改CardView::createFaceSprite()：

void CardView::createFaceSprite(const CardModel& model) {

// 原有逻辑...

if (model.getFace() == CF\_JOKER) {

// 加载特殊卡牌资源（如 Joker 图片）

auto jokerSprite = Sprite::create("res/special/joker.png");

jokerSprite->setPosition(faceSprite->getContentSize() / 2);

faceSprite->addChild(jokerSprite);

}

}

#### **5.1.5 步骤5：处理特殊逻辑**

若新卡牌有特殊交互（如点击时触发技能），在控制器中添加逻辑：

// PlayFieldController.cpp

bool PlayFieldController::moveToStackTop(GameModel& model, int playFiledsIndex, int tagId) {

auto& card = model.getPlayFieldPile().at(playFiledsIndex).cards.back();

if (card.hasSpecialSkill()) {

// 执行特殊技能（如跳过一次验证）

CCLOG("Special skill triggered!");

}

// 原有验证逻辑...

}

### 5.2 新增一种回退功能（如回退多步操作）

#### **5.2.1 步骤1：扩展回退模型**

修改UndoModel以支持多步回退记录（存储完整游戏状态而非单张卡牌）：

// UndoModel.h

include "GameModel.h" // 引入游戏模型

class UndoModel {

private:

// 存储完整游戏状态（原仅存储单张卡牌）

std::deque<GameModel> \_gameStates;

public:

// 记录完整游戏状态

void push(const GameModel& gameModel) {

\_gameStates.push\_back(gameModel);

}

// 回退到上一个状态

GameModel undo() {

auto lastState = \_gameStates.back();

\_gameStates.pop\_back();

return lastState;

}

// 回退N步

GameModel undo(int steps) {

if (steps > \_gameStates.size()) {

steps = \_gameStates.size();

}

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

\_gameStates.pop\_back();

}

return undo();

}

};

#### **5.2.2 步骤2：修改回退管理器**

更新UndoManager以支持多步回退：

// UndoManager.h

class UndoManager : public Node {

public:

// 新增：回退N步

GameModel undo(int steps) {

return \_undoModel.undo(steps);

}

// 记录完整游戏状态（原仅记录单张卡牌）

void record(const GameModel& gameModel) {

\_undoModel.push(gameModel);

}

};

#### **5.2.3 步骤3：更新控制器逻辑**

在GameController中添加多步回退触发方法：

// GameController.h

void undoOperation(int steps) {

if (!\_undoManager->canUndo()) return;

// 获取回退的游戏状态

GameModel lastState = \_undoManager->undo(steps);

// 更新当前游戏模型

setGameModel(lastState);

// 刷新视图

\_playFieldController->setUpView(lastState);

\_stackController->setUpView(lastState);

}

#### **5.2.4 步骤4：适配视图层**

确保视图层能根据完整游戏状态刷新（现有setUpWithCardModels已支持，无需额外修改）。

#### **5.2.5 步骤5：添加用户交互**

在UI中添加多步回退按钮，绑定事件到GameController::undoOperation(int steps)：

// 示例：在GameView中添加按钮

auto undo5StepsBtn = Button::create("res/btn\_undo5.png");

undo5StepsBtn->addClickEventListener([this](Ref\* sender) {

\_gameController->undoOperation(5); // 回退5步

});

this->addChild(undo5StepsBtn);

## 六、项目初期遇到的一些问题

### 1. 层级渲染问题

问题表现：卡片、视图层之间可能出现遮挡顺序错误（例如移动的卡片被其他元素遮挡，或 UI 层级混乱）。

代码中可能的原因：

ZOrder 设置不一致：Cocos2d-x 通过localZOrder（节点树内层级）和globalZOrder（全局层级）控制渲染顺序。例如：

GameView中设置了setGlobalZOrder(-2)，而PlayFieldView和StackView未显式设置全局层级，可能导致视图层整体被其他元素覆盖。

卡片移动时（如StackView::moveToStackTop）仅设置localZOrder，若父节点层级不同，可能无法保证卡片在最上层。

添加节点顺序问题：GameController中先添加\_stackController再添加\_playFieldController，默认后添加的节点层级更高，可能导致堆叠区被游戏区遮挡。

解决思路：

统一层级管理：为不同类型的节点定义明确的层级规则（如：背景层 < 游戏区 < 堆叠区 < 操作按钮 < 提示层），通过globalZOrder区分大层级，localZOrder区分同层内元素（如卡片叠放顺序）。

// 例如在视图初始化时设置全局层级

\_playFieldView->setGlobalZOrder(10); // 游戏区

\_stackView->setGlobalZOrder(20); // 堆叠区（层级更高，避免被游戏区遮挡）

\_undoButton->setGlobalZOrder(100); // 操作按钮（最上层）

动态调整卡片层级：卡片移动时临时提高globalZOrder，确保移动过程中不被遮挡，完成后恢复：

// 在CardView移动时

card->setGlobalZOrder(200); // 临时提升层级

auto callback = CallFunc::create([=]() {

card->setGlobalZOrder(originalZOrder); // 恢复原层级

});

### **2. 控制器回调等加载问题**

问题表现：点击卡片、按钮时回调未触发，或触发时上下文（如GameModel）已失效，导致逻辑错误。

代码中可能的原因：

回调捕获的变量生命周期问题：例如StackController::setUp中，lambda 回调捕获&model（局部变量），而model是setUp的参数，函数结束后可能失效，导致回调中访问非法内存：

// StackController.cpp 中存在风险的代码

\_stackView->setOnClickCallBack([this, &model](int playFiledsIndex,int tagId) {

moveToStackTop(model, tagId); // model可能已被销毁

});

视图未正确初始化：GameView::setup中注释了\_playFieldView和\_stackView的初始化代码，可能导致视图未添加到节点树，回调无法通过事件分发器传递。

回调未绑定到正确节点：CardView的点击事件通过\_eventDispatcher注册，但如果CardView未被正确添加到父节点（如GameView），事件可能无法被捕获。

解决思路：

避免捕获局部变量引用：控制器应通过成员变量访问GameModel，而非捕获局部参数。例如StackController持有GameModel的指针（需确保生命周期匹配）：

// StackController.h 中添加成员

GameModel\* \_gameModel = nullptr;

// StackController.cpp 中 setUp 方法

void StackController::setUp(GameModel& model) {

\_gameModel = &model; // 存储指针（需保证model生命周期长于控制器）

\_stackView->setOnClickCallBack([this](int playFiledsIndex,int tagId) {

moveToStackTop(\*\_gameModel, tagId); // 使用成员变量

});

}

确保视图初始化完整：取消GameView::setup中对视图初始化的注释，保证\_playFieldView和\_stackView被正确添加到节点树：

// GameView.cpp 中恢复初始化代码

\_playFieldView = PlayFieldView::create();

\_playFieldView->setUpWithCardModels(model);

\_stackView = StackView::create();

\_stackView->setUpWithCardModels(model);

addChild(\_playFieldView);

addChild(\_stackView);

### **3. 组件的生命周期问题**

问题表现：节点重复释放导致崩溃，或未释放导致内存泄漏；控制器、视图初始化 / 销毁顺序混乱。

代码中可能的原因：

内存管理不当：例如GameController中使用CC\_SYNTHESIZE\_RETAIN(UndoManager\*, \_undoManager, UndoManager)，但未在析构函数中释放，可能导致内存泄漏。

初始化顺序错误：GameController::startGame中先调用\_stackController->setUp(\_gameModel)，再初始化\_undoManager，而setUp中可能依赖\_undoManager（若回调中使用），导致空指针。

节点添加 / 移除不匹配：StackView::setUpWithCardModels中调用removeAllChildren()，但未确保子节点被正确释放（若子节点未使用autorelease）。

解决思路：

严格遵循 Cocos 内存管理规则：使用create()创建的节点默认autorelease，通过retain()/release()管理生命周期。例如UndoManager在析构时释放：

// GameController.h 中

~GameController() {

CC\_SAFE\_RELEASE(\_undoManager); // 释放retain的对象

}

保证初始化顺序：依赖项先初始化。例如GameController中先初始化\_undoManager，再传给其他控制器：

// 调整 GameController::startGame 顺序

\_undoManager = UndoManager::create(); // 先初始化

\_stackController->setUndoManager(\_undoManager); // 再传递

\_stackController->setUp(\_gameModel); // 最后调用setUp

移除节点时确保释放：使用removeAllChildrenWithCleanup(true)强制清理子节点，避免悬垂指针：

// StackView::setUpWithCardModels 中

this->removeAllChildrenWithCleanup(true); // 清理子节点并释放

## 七、总结

本项目通过MVC架构实现了卡牌游戏的核心功能，代码结构清晰，各模块职责明确。新增卡牌需从模型、资源、配置、视图、逻辑五个层面进行扩展；新增回退功能则需扩展回退模型、管理器及控制器逻辑，确保完整记录和恢复游戏状态。后续可基于现有架构进一步扩展玩法（如多玩家对战、卡牌技能系统等）。