**卡牌游戏程序设计文档**

# 1. 概述

本文档概述了卡牌游戏的软件架构和程序流程。该游戏使用 Cocos2d-x 框架构建，并遵循模型-视图-控制器（MVC）模式，并使用服务层和管理层进行扩展，以处理数据加载和处理。

# 2. 架构

应用程序分为以下几个层次：

\* 模型（Models）: 表示应用程序的数据结构。

CardModel: 存储单张卡牌的属性，如点数、花色和位置。

\* 视图（Views）: 处理表示层和用户交互。

GameScene: 游戏的主场景，负责显示游戏面板，包括游戏区域和牌堆。

CardSprite: 可视化表示卡牌的自定义精灵。

\* 控制器（Controllers）: 包含应用程序的逻辑，并充当模型和视图之间的中介。

GameController: 管理游戏状态，处理用户输入，并相应地更新视图。

\* 服务（Services）: 负责外部交互，例如从文件加载数据。

CardService: 从 JSON 文件加载卡牌数据。

\* 管理器（Managers）: 管理和处理数据，通常从服务层接收数据。

CardManager: 解析来自 CardService 的原始数据，并创建 CardModel 实例。

# 3. 程序功能工作流程

## 3.1 背景显示

以下序列描述了程序从启动到显示游戏元素的执行流程：

（1）初始化: 应用程序启动，AppDelegate 创建并运行 GameScene。

（2）控制器设置: GameScene 实例化 GameController。

（3）数据加载:

①GameController 调用 CardService 从 JSON 文件（例如 `level\_0.json`）加载关卡数据。

②CardService 读取文件并将其解析为 rapidjson::Document。

（4）数据处理:

①GameController 将 rapidjson::Document传递给 CardManager。

②CardManager 处理 JSON 数据，为 Playfield 和 Stack数组中的每张卡牌创建一个 CardModel。对于 Playfield 卡牌，它会从 JSON 数据中读取位置信息。

（5）UI 设置:

①GameScene 调用其 setupUI 方法来渲染游戏元素。

②它从 GameController 检索游戏区域和牌堆的卡牌列表。

③对于每个 CardModel，它会创建一个相应的 CardSprite。

④游戏区域的卡牌根据其 CardModel 中的 position 数据进行定位。

⑤牌堆的卡牌则按计算好的布局进行排列。

（6）就绪状态: 游戏现已完全加载，并准备好与用户进行交互。

## **3.2 手牌区翻牌替换**

### 3.2.1 功能概述

该功能允许玩家点击手牌区（Stack A）中的任意一张牌，这张被选中的牌会移动到弃牌区（Stack B）的顶部，成为新的顶牌。这个操作是游戏的基本交互之一，用于主动更换弃牌区的顶牌，以便与桌面牌进行匹配。

### **3.2.2. 整体工作流程**

整个流程由 视图层（View） 、控制器层（Controller） 和 模型层（Model）协同完成，确保了UI交互、游戏逻辑和数据状态的清晰分离。

（1）用户输入 (View - CardLayer ) :

①玩家在屏幕上点击属于手牌区（Stack A）的卡牌精灵（Sprite）。

②CardLayer 中的触摸事件监听器 ( onTouchBegan / onTouchEnded ) 会被触发。

③通过点击的坐标，系统会识别出被点击的是手牌区的某一张牌，并获取与之关联的 CardModel 数据对象。

（2）逻辑处理 (Controller - GameController ) :

①CardLayer 在确认点击有效后，会调用 GameController 中负责处理此逻辑的函数，即 moveCardFromAtoB() 。

②GameController::moveCardFromAtoB() 函数执行核心的游戏逻辑：

·它从 \_stackCardsA （手牌区数据栈）中弹出顶部的牌 ( CardModel\* )。

·为了支持后续的回退功能 ，在移动卡牌之前，会将其 owner 属性设置为 CardOwner::STACK\_A ，记录它来自手牌区。

·然后，将这张牌压入 \_stackCardsB （弃牌区数据栈）的顶部。

至此，游戏的数据状态已经更新完毕。

（3）界面刷新 (View - CardLayer ) :

①在 GameController 更新完数据模型后， CardLayer 会调用 refreshStackLayers() 和 refreshPlayfieldLayers() 方法来更新界面显示。

②refreshStackLayers() 会清空并重新绘制手牌区和弃牌区的所有卡牌。

③对于刚刚被移动的卡牌，系统会创建一个 MoveTo 动画，使其从原始位置平滑地移动到弃牌区顶部的新位置。

④同时，手牌区的显示也会更新，被点击的牌将不再显示在手牌区。

### **3.2.3 关键代码实现**

（1）CardLayer.cpp :

①在 \_onStackACardClicked() (或类似的触摸处理函数) 中，获取被点击的 CardModel 并调用 GameController 的方法。

②在 refreshStackLayers() 中，根据 GameController 提供的数据，重新渲染两个牌堆的视图，并为需要移动的牌创建 MoveTo 动画。

（2）GameController.cpp :

①moveCardFromAtoB() : 实现从 \_stackCardsA 到 \_stackCardsB 的数据转移，并设置 owner 属性。

②getStackCardsA() / getStackCardsB() : 提供接口供 CardLayer 获取最新的牌堆数据。

（3）CardModel.h :

①定义了 CardModel 类，包含卡牌的花色、点数、位置以及用于回退功能的 owner 属性。

## **3.3 桌面牌与手牌区手牌匹配**

### **3.3.1功能概述**

此功能是游戏的核心玩法。玩家可以点击桌面（Playfield）上的任何一张可见的牌，如果这张牌的点数与弃牌区（Stack B）顶部的牌点数正好相差1（例如，桌面是3，弃牌区顶部是4；或者桌面是K，弃牌区顶部是A），则匹配成功。匹配成功后，被点击的桌面牌会移动到弃牌区的顶部，成为新的顶牌。此规则不考虑花色。

### **3.3.2整体工作流程**

与3.2类似，此功能也遵循 View -> Controller -> Model -> View 的工作流。

（1）用户输入 (View - CardLayer ):

①玩家点击桌面区域（Playfield A 或 Playfield B）的一张卡牌。

②CardLayer 的触摸事件监听器捕获点击事件，并识别出被点击的桌面牌及其对应的 CardModel 。

（2）逻辑处理与匹配检查 (Controller - GameController ) :

①CardLayer 将被点击的桌面牌 CardModel 和其来源（Playfield A 或 B）传递给 GameController 的 tryMoveCardFromPlayfieldToStack() 函数。

②GameController::tryMoveCardFromPlayfieldToStack() 执行以下核心逻辑：

·首先，检查弃牌区 \_stackCardsB 是否为空。如果为空，则无法匹配，直接返回 false 。

·获取弃牌区顶牌 \_stackCardsB.top() 和被点击的桌面牌的点数。

·执行匹配规则 ：计算两张牌点数的差值，如果绝对值为1，或者一张是A另一张是K，则视为匹配成功。

·如果匹配成功：

i从对应的桌面牌数据源（ \_playfieldCardsA 或 \_playfieldCardsB ）中弹出该卡牌。

ii记录来源 ：根据卡牌的来源，将其 owner 属性设置为 CardOwner::PLAYFIELD\_A 或 CardOwner::PLAYFIELD\_B ，为回退功能做准备。

iii将被点击的桌面牌压入 \_stackCardsB 栈顶。

iiii函数返回 true ，表示移动成功。

·如果匹配失败，函数返回 false 。

（3）界面刷新 (View - CardLayer ) :

①CardLayer 根据 tryMoveCardFromPlayfieldToStack() 的返回值来决定是否更新界面。

②如果返回 true （匹配成功）：

·调用 refreshPlayfieldLayers() 和 refreshStackLayers() 。

·系统会为被点击的桌面牌创建一个 MoveTo 动画，使其从桌面平滑移动到弃牌区的顶部。

·桌面上原来的位置会空出来，可能会显示出其下方的牌（如果存在）。

1. 如果返回 false ，界面不发生任何变化。

### **3.3.3关键代码实现**

（1）CardLayer.cpp :

①在\_onPlayfieldCardClicked() (或类似的触摸处理函数) 中，调用 GameController::tryMoveCardFromPlayfieldToStack() 。

②根据返回值，决定是否调用 refresh... 系列函数来更新UI和执行动画。

（2）GameController.cpp :

①tryMoveCardFromPlayfieldToStack(CardModel\* card, CardOwner origin) : 这是核心逻辑函数。它包含了匹配规则的判断、数据在不同牌堆（ \_playfieldCardsA/B 到 \_stackCardsB ）之间的转移，以及 owner 属性的设置。

（3）CardModel.h :

①CardFaceType 枚举的定义对于点数比较至关重要，特别是处理 A 和 K 的特殊匹配逻辑。

**3.3 回退功能**

### **3.3.1功能概述**

回退功能为玩家提供了撤销上一步操作的能力。每当玩家执行一次有效的卡牌移动（无论是从手牌区移动还是从桌面区匹配），该操作都会被记录下来。点击“回退”按钮，最近一次移动的卡牌将会从弃牌区（Stack B）返回其原始位置，恢复到操作前的状态。此功能可以连续使用，直到回退到本关卡的初始状态为止。

### **3.3.2 整体工作流程**

实现回退功能的核心在于 记录每一次移动的来源 。这通过在 CardModel 中增加一个 owner 属性来实现。

（1）记录移动 (Controller - GameController ) :

①在执行任何合法的卡牌移动之前（即在 moveCardFromAtoB() 和 tryMoveCardFromPlayfieldToStack() 中）， GameController 会设置被移动卡牌的 owner 属性。

②owner 是一个 CardOwner 枚举，可能的值包括 STACK\_A , PLAYFIELD\_A , PLAYFIELD\_B 。

③例如，当一张牌从 \_playfieldCardsA 移动到 \_stackCardsB 时，这张牌的 owner 会被设置为 CardOwner::PLAYFIELD\_A 。

④初始状态处理 ：在游戏关卡初始化时，所有卡牌的 owner 属性都被设置为 CardOwner::NONE 。这是区分“玩家移动的牌”和“初始状态的牌”的关键。

（2）用户输入 (View - CardLayer ) :

①玩家点击界面上的“回退”按钮 ( \_rollbackButton )。

②CardLayer 中为该按钮绑定的回调函数 \_onRollbackClicked() 被触发。

（3）逻辑处理 (Controller - GameController ) :

①\_onRollbackClicked() 函数会直接调用 GameController::rollbackLastMove() 。

②GameController::rollbackLastMove() 执行回退逻辑：

·检查 \_stackCardsB 是否为空，如果为空则不执行任何操作。

·获取 \_stackCardsB 的顶牌 card 。

·检查是否为初始状态 ：检查 card->owner 的值。如果为 CardOwner::NONE ，说明这张牌是关卡初始时就在弃牌区的牌，不允许回退，函数直接返回。

·如果不为 NONE ，则根据 card->owner 的值，将这张牌从 \_stackCardsB 弹出，并压入其原始的牌堆中（ \_stackCardsA , \_playfieldCardsA , 或 \_playfieldCardsB ）。

③数据状态成功恢复到上一步。

（4）界面刷新 (View - CardLayer ) :

①在 \_onRollbackClicked() 中，调用完 rollbackLastMove() 后，会立即调用 refreshPlayfieldLayers() 和 refreshStackLayers() 来更新整个游戏界面的显示。

②被回退的卡牌会从弃牌区消失，并出现在它原来的位置。这个过程同样可以通过 MoveTo 动画来使其更加平滑自然。

### **3.3.3关键代码实现**

（1）CardModel.h :

①增加了 CardOwner 枚举 ( NONE , STACK\_A , PLAYFIELD\_A , PLAYFIELD\_B )。

②CardModel 类中增加了 owner 成员变量。

（2）GameController.cpp :

①moveCardFromAtoB() / tryMoveCardFromPlayfieldToStack() : 在移动卡牌前，正确设置 card->owner 。

②rollbackLastMove() : 实现回退的核心逻辑，包括检查 owner 是否为 NONE ，以及根据 owner 将卡牌移回正确的牌堆。

（3）CardLayer.cpp :

①\_rollbackButton 和 \_onRollbackClicked() : 负责捕捉用户的回退请求，并调用 GameController 的回退方法，最后刷新界面

# 扩展问题1：如何新加一个卡牌一个新类型的回退功能

在当前的代码结构中，添加一种新的卡牌（如“王牌”）是一个清晰、分步骤的过程，主要涉及数据模型、资源、数据处理和游戏逻辑的扩展。

4.1 步骤如下：

（1）更新数据模型 ( CardModel.h )

①首先，需要让系统“认识”这种新卡牌。这需要在 CardFaceType 枚举中添加一个新的值，例如 Joker 。

②文件 : d:\CocosProjects\cardGame\Classes\models\CardModel.h

③修改 :

|  |
| --- |
| enum class CardFaceType {      A = 1, Two, Three, Four, Five, Six,      Seven, Eight, Nine, Ten, J, Q, K,      Joker // 新增王牌类型  }; |

（2）添加卡牌资源

①为新的“王牌”创建对应的图片资源（例如 joker.png ），并将其放置在 Resources/ 目录下的相应位置。这确保了视图层可以找到并渲染它。

（3）扩展数据处理逻辑 ( CardManager.cpp )

①CardManager 负责从数据源（如JSON）创建卡牌模型。需要确保 \_createCardModelFromJson 方法能够正确解析代表“王牌”的数据。如果数据格式不变，这一步可能不需要修改，只需确保传入的数据中包含了“王牌”对应的值。

（4）更新视图渲染逻辑 ( CardSprite.cpp )

①CardSprite 负责将 CardModel 渲染为游戏中的卡牌精灵。需要更新其内部逻辑，使其能够根据 CardFaceType::Joker 来加载正确的图片资源。

（5）调整游戏核心逻辑 ( GameController.cpp )

①这是最关键的一步。您需要定义“王牌”在游戏中的行为规则。

②例如，在 tryMoveCardFromPlayfieldToStack 或其他类似的方法中，您需要添加逻辑来判断：

·“王牌”可以移动到哪个牌堆？

·“王牌”是否可以和任何其他牌进行组合？

· 当“王牌”被点击时，应该发生什么？

③这些规则的修改将集中在 GameController 中，因为它负责协调整个游戏流程。

优势总结 : 这种分层结构（Model-View-Controller）的优势在于，添加新卡牌的修改是 局部化 的。数据、视图和逻辑各司其职，您只需要在对应的层次进行扩展，而不需要改动整个系统。

# 扩展问题2：如何新加一个新类型的回退功能

要在当前架构下实现一个灵活的回退功能，最佳实践是引入 命令模式（Command Pattern） 。该模式可以将一个请求封装为一个对象，从而让您能够参数化客户端对象，对请求排队或记录请求日志，以及支持可撤销的操作。

5.1实现步骤：

（1）创建命令基类 ( Command.h )

①定义一个抽象的 Command 接口，包含两个核心方法： execute() 和 undo() 。

|  |
| --- |
| class Command {  public:      virtual ~Command() {}      virtual void execute() = 0;      virtual void undo() = 0;  }; |

（2）创建具体命令类 (例如 MoveCardCommand.h/.cpp )

①为需要支持回退的每一个动作创建一个具体的命令类。例如，要回退一次移动卡牌的操作，可以创建一个 MoveCardCommand 。

②这个类需要存储执行该操作所需的所有上下文信息，例如：

·移动的卡牌 ( CardModel\* )

·源牌堆 ( std::stack<CardModel\*>& )

·目标牌堆 ( std::stack<CardModel\*>& )

·游戏控制器 ( GameController\* )

③execute() 方法 : 执行卡牌的移动逻辑。

④undo() 方法 : 执行与 execute() 相反的操作，即将卡牌从目标牌堆移回源牌堆。

（3）集成到游戏控制器 ( GameController )

①在 GameController 中，引入一个命令历史记录栈，用于存储所有已执行的命令。

|  |
| --- |
| // GameController.h  #include <stack>  #include "commands/Command.h" // 假设的路  径    class GameController {  private:      std::stack<Command\*> \_commandHistory;      // ... 其他成员  }; |

②修改现有的游戏逻辑函数（如 tryMoveCardFromPlayfieldToStack ）。现在，这些函数不再直接修改游戏状态，而是 创建并执行一个命令对象 。

|  |
| --- |
| // GameController.cpp  void GameController::someAction  (CardModel\* card, ...) {      Command\* command = new      MoveCardCommand(this, card,      \_sourceStack, \_destinationStack);      command->execute();      \_commandHistory.push(command);  } |

（4）实现回退功能

①在 GameController 中添加一个公共的 undo() 方法。

|  |
| --- |
| // GameController.cpp  void GameController::undoLastMove() {      if (!\_commandHistory.empty()) {          Command\* lastCommand =          \_commandHistory.top();          \_commandHistory.pop();          lastCommand->undo();          delete lastCommand; // 清理内存      }  } |

（5）连接UI

①在 CardLayer 中添加一个“回退”按钮。

②该按钮的点击回调函数将调用 GameController 的 undoLastMove() 方法。

③CardLayer 在接收到模型变化的通知后，会刷新视图，从而在屏幕上反映出回退后的状态。

优势总结 : 使用命令模式，您可以轻松地添加新的可回退操作，只需创建新的具体命令类即可，而无需修改 GameController 的核心回退逻辑。这使得系统 高度可扩展 ，并且将动作的执行与动作的逻辑解耦，代码更加清晰。