（普招）软件五组

项目设计报告

目录

[1前言 1](#_Toc498128014)

[1.1项目背景 1](#_Toc498128015)

[1.2目的和意义 1](#_Toc498128016)

[1.3项目特点 1](#_Toc498128017)

[1.4设计原则 1](#_Toc498128018)

[1.4参考文献 2](#_Toc498128019)

[2总体设计 2](#_Toc498128020)

[2.1系统架构 2](#_Toc498128021)

[2.1.1性能指标 4](#_Toc498128022)

[2.1.2主要开发语言 4](#_Toc498128023)

[2.1.3游戏引擎版本 4](#_Toc498128024)

[2.2开发与运行平台 4](#_Toc498128025)

[2.3游戏引擎 5](#_Toc498128026)

[2.3.1 UI树 5](#_Toc498128027)

[2.3.2导演类 6](#_Toc498128028)

[2.3.3内存管理 7](#_Toc498128029)

[2.3.4渲染 9](#_Toc498128030)

[2.3.5游戏循环 9](#_Toc498128031)

[2.3.6事件分发 11](#_Toc498128032)

[2.3.7物理引擎 12](#_Toc498128033)

[2.3.8跨平台 13](#_Toc498128034)

[2.3.9缓存 13](#_Toc498128035)

[2.3.10调度器 14](#_Toc498128036)

[2.3.11动作 15](#_Toc498128037)

[2.4 GameData数据访问对象 16](#_Toc498128038)

[2.4.1专用数据访问模块 16](#_Toc498128039)

[2.4.2不使用数据库+SQL 18](#_Toc498128040)

[2.5 Resource资源组织 19](#_Toc498128041)

[2.6 Lua语言集成 21](#_Toc498128042)

[2.7 工具类 22](#_Toc498128043)

[2.7.1音频控制器 22](#_Toc498128044)

[2.7.2点击粒子 23](#_Toc498128045)

[2.7.3占位符 23](#_Toc498128046)

[2.8 层 23](#_Toc498128047)

[2.8.1设置层 24](#_Toc498128048)

[2.8.2对话层 24](#_Toc498128049)

[2.8.3确认按钮 24](#_Toc498128050)

[2.9 场景组织 25](#_Toc498128051)

[2.8.1场景缓存 27](#_Toc498128052)

[2.8.2场景栈 27](#_Toc498128053)

[2.8.3场景生命周期 28](#_Toc498128054)

[2.8.4跳转表 29](#_Toc498128055)

[2.8.5界面布局 30](#_Toc498128056)

[2.8.6资源加载层 30](#_Toc498128057)

[2.10游戏场景 31](#_Toc498128058)

[2.10.1游戏逻辑和架构 31](#_Toc498128059)

[2.10.1地图层 33](#_Toc498128060)

[2.10.2瓦片地图 33](#_Toc498128061)

[2.10.3物理世界 35](#_Toc498128062)

[2.10.4摄像机 35](#_Toc498128063)

[2.10.5活动对象 37](#_Toc498128064)

[2.10.6触摸监听器 38](#_Toc498128065)

[2.10.7控制面板 39](#_Toc498128066)

[2.10.8帧动画 39](#_Toc498128067)

[2.10.9弹幕系统 40](#_Toc498128068)

[2.10.10互动对象 41](#_Toc498128069)

[2.10.11着色器 42](#_Toc498128070)

[2.10.12碰撞监听器 43](#_Toc498128071)

[2.10.13自定义事件监听器 44](#_Toc498128072)

[2.10.14区域 44](#_Toc498128073)

[2.10.15视差节点 45](#_Toc498128074)

[2.10.16特效 47](#_Toc498128075)

[2.10.17状态机 47](#_Toc498128076)

[2.10.18事件脚本处理 48](#_Toc498128077)

[2.10.19事件过滤 49](#_Toc498128078)

[3详细设计 51](#_Toc498128079)

[3.1项目构建与运行 51](#_Toc498128080)

[3.1.1 Android 平台项目构建与运行 51](#_Toc498128081)

[3.1.2 Linux 平台项目构建与运行 52](#_Toc498128082)

[3.1.3 Windows 平台项目构建与运行 52](#_Toc498128083)

[3.2工具类实现 53](#_Toc498128084)

[3.2.1 AudioController 53](#_Toc498128085)

[3.2.2 ClickParticle 54](#_Toc498128086)

[3.2.3 PlaceHolder 54](#_Toc498128087)

[3.3层实现 55](#_Toc498128088)

[3.3.1 SettingLayer 55](#_Toc498128089)

[3.3.2 ConversationLayer 55](#_Toc498128090)

[3.3.3 ConfirmButton 55](#_Toc498128091)

[3.4场景组织实现 56](#_Toc498128092)

[3.4.1 NonGameplayScenesCache 56](#_Toc498128093)

[3.4.2生命周期 56](#_Toc498128094)

[3.4.3 内部类InnerClass 57](#_Toc498128095)

[3.5 GameData实现 58](#_Toc498128096)

[3.5.1接口 58](#_Toc498128097)

[3.5.2实现 59](#_Toc498128098)

[3.6 Lua集成 62](#_Toc498128099)

[3.6.1嵌入 62](#_Toc498128100)

[3.6.2改造 63](#_Toc498128101)

[3.6.3从 C++ 代码生成 Lua 绑定 64](#_Toc498128102)

[3.6.4对话层Lua实现 65](#_Toc498128103)

[3.7游戏场景实现 65](#_Toc498128104)

[3.7.1制作物理阻挡 65](#_Toc498128105)

[3.7.2双主角切换 66](#_Toc498128106)

[3.7.3物理移动 67](#_Toc498128107)

[3.7.4切换区块 68](#_Toc498128108)

[3.7.5视差节点 69](#_Toc498128109)

[3.7.6电梯 70](#_Toc498128110)

[3.7.7碰撞监听器架构 70](#_Toc498128111)

[3.7.8自定义监听器使用 71](#_Toc498128112)

[3.7.9着色器 72](#_Toc498128113)

[3.7.10活动对象创建与组成 73](#_Toc498128114)

[3.7.11状态机实现 75](#_Toc498128115)

[3.7.12人工智能 76](#_Toc498128116)

[3.8弹幕系统实现 77](#_Toc498128117)

[3.8.1系统概述 77](#_Toc498128118)

[3.8.2弹幕类型 77](#_Toc498128119)

[3.8.3弹幕实现 78](#_Toc498128120)

[3.8.4性能优化 80](#_Toc498128121)

[3.8.5技术难点 81](#_Toc498128122)

# 1前言

## 1.1项目背景

游戏开发是近来非常流行的话题，其中手机游戏因其上手难度低，配置要求低等特点广受手机用户欢迎。

Cocos2dx是一款免费开源的游戏引擎，已经有不少使用此引擎的游戏成功上线运营。Cocos2dx因其功能强大，简单易用，在国内拥有活跃的开发者社区。Cocos2dx尤其适合手机游戏的开发。因此项目组选用Cocos2dx作为游戏的开发引擎。

## 1.2目的和意义

本项目设计报告文档专注于游戏架构的分析和设计，不仅仅讲解实现方案，更是将解决方案的研制过程融入设计中，这样能够对设计的来龙去脉有更深刻的理解。

## 1.3项目特点

游戏开发很少有完善的客户需求支撑，往往是制作团队根据市场分析，自己拟定游戏需求。游戏开发是一个充满想象力，充分发挥创造力的工作，在游戏真正实现之前很难仅通过设定文档就验证游戏是否好玩。因此游戏项目的需求非常多变。即使游戏上线运营，后续修改也会不停地进行。这就使得相比于具体功能的实现，健壮的，经得起修改的架构更加重要。

## 1.4设计原则

整个项目在设计过程中考虑到了自身实力以及外部条件。

项目团队是初次进行游戏开发，存在经验不足的问题，所以只能进行小型的游戏项目开发工作。在设计过程中经常需要引入新技术，项目团队必须验证技术可行性才可以将新技术应用于项目中。

考虑外部条件，因为项目组只有非常有限的可利用资源，也不利于开展大规模的开发。

本项目的定位更多偏向于游戏开发技术的探索和应用上，因此设计时更加注重设计思想和设计方法的应用。

## 1.4参考文献

[美]Jason Gregory ，[译]叶劲峰，《游戏引擎架构》，电子工业出版社2014

秦春林，《我所理解的Cocos2d-x》，电子工业出版社2014

刘剑卓，郑光龙，《Cocos2d-x 游戏开发技术精解》，人民邮电出版社2015

[美]Robert Nystrom [译]GPP翻译组，《游戏编程模式》，人民邮电出版社2016

[美]Sanjay Madhav [译]刘瀚洋，《游戏编程算法与技巧》，电子工业出版社2017

# 2总体设计

在总体设计环节，本文档不仅将逐一说明各个功能或模块的实现方式，还将就设计原因和过程进行简要分析。

## 2.1系统架构

整个游戏项目的系统架构如图2-1所示。

图2-1

### 2.1.1性能指标

本项目采用的帧数为每秒60帧。如果一帧实际花费的时间比1/60秒还要小，游戏引擎会自动休眠直到时间达到1/60秒。

如图2-2所示。

图2-2

### 2.1.2主要开发语言

C++

### 2.1.3游戏引擎版本

Cocos2dx 3.14.1

## 2.2开发与运行平台

Android（NDK）

Windows（Visual Studio）

Linux（CMake）

## 2.3游戏引擎

Cocos2dx包含了游戏运行时所需的大部分核心组件。在本项目中需要使用的核心功能已经表示在系统架构图中。

需要注意的是这些功能模块并不代表引擎的全部，也不是本项目使用的全部功能特性，而是针对本项目特点选择的，起重要作用的功能。本节将对一些重要的功能特性进行简要介绍，以及在项目中的应用。理解这些知识是有效利用引擎开发游戏的基础。

### 2.3.1 UI树

Cocos2dx中，创建的任何对象必须添加进UI树才可以显示在屏幕上。这些对象一般都是直接继承自Node。如图2-3所示。



图片 2-3

图2-3

每一个添加的子节点都有自己的本地深度，添加进UI树的节点需要注意绘制的顺序。绘制方向从屏幕内朝向屏幕外，这被称为画家算法。如图2-4所示。

并不是任何UI树上的节点都会参与绘制，有时候节点并没有设置任何的图像信息，那么此节点就不会显示出来。通常将此类隐藏节点作为某节点的功能子节点。

图2-4

### 2.3.2导演类

Director是整个cocos2dx引擎的核心，是整个游戏的导航仪。游戏中的一些常用操作就是由Director来控制的，如OpenGL ES的初始化，不同场景间的切换，游戏暂停继续的控制，世界坐标和GL坐标之间的切换，对节点（游戏元素）的控制等，屏幕尺寸的获取等都要由Director类来管理控制的。

场景的切换如图2-5所示。

* 在 项目使用中，导演类主要用于获取屏幕尺寸，配合本项目实现的场景缓存切换场景，获取事件分发器等。

图2-5

### 2.3.3内存管理

C++使用new关键字在运行时给一个对象动态分配内存，并返回堆上的内存地址指针。被动态分配的内存只有在对象不再使用时，通过delete运算符才能释放内存。显示分配内存的方法虽然很容易理解，但是非常容易出错。游戏是一种实时性非常高的系统，很难确保对象能够在合适的时机释放，也很难确保被释放的对象不再被调用，因此经常会遇见野指针，内存泄漏等问题。

C++并没有提供完善的垃圾回收机制。虽然C++11引入了智能指针，但是智能指针会带来不必要的开销，而且要求程序员区分强引用、弱引用、独占指针等的差异。这将使得游戏逻辑代码到处充满复杂的内存管理代码。

Cocos2dx使用引用计数进行内存管理。Ref基类负责对对象进行引用计数管理。在Cocos2dx中，几乎所有对象都继承自Ref基类，即所有继承Ref的子类都可以使用cocos2dx的内存管理机制。

在Cocos2dx中，大部分对象的构造函数都是无法直接访问的，只有通过相应类提供的工厂方法构造对象，并返回对象的指针。在当前帧结束后，游戏引擎会自动对当前帧创建对象的引用计数减1。

每一个对象都会记录自身的引用计数，只要对象被释放前被某些对象“使用”，其引用计数就会增加。当其引用计数减为0后，说明已经没有其他对象在使用该对象，就可以安全地释放该对象所占内存。如图2-6所示。

Cocos2dx的内存管理方法使得程序员可以专注于游戏逻辑的开发，而不用特别担心内存的管理问题。

* 在项目中，只要坚持使用Cocos2dx的内存管理机制，与传统C++编程的内存管理方法区分开，就可以妥善地管理内存。。



图2-6

### 2.3.4渲染

在传统游戏编程中，程序员不得不使用图形编程接口和硬件交互，使得程序难以适应平台的变化和硬件的更新。

Cocos2dx使用OpenGL ES进行绘制。OpenGl ES是一套图形硬件的软件接口，直接和GPU进行交互。程序员只需要将游戏对象按照正确的层级组织起来，通过Cocos2dx封装的操作给对象设置贴图或动画，就可以避免和图形编程语言直接交互。而且大多数情况下，使用Cocos2dx封装的绘制操作将要比自己重新实现高效安全得多。

在每帧处理的后半部分，游戏引擎遍历UI树，找出所有需要参与绘制的对象。在遍历的结果上，通过排序使用相同纹理的对象，剔除不可见的对象等方法优化绘制过程。尤其是在Cocos2dx 3.0版本后，绘制系统更加简洁与高效。老版本中需要程序员将需要绘制的图像信息通过特殊的对象管理起来，再从这些对象中取得图像信息。然而在新版本中使用了自动批绘制技术。由引擎自动进行绘制优化工作。

* 在新版本的游戏引擎中，程序员不需要特别关心绘制的顺序以及绘制的优化，可以将重点更多放在游戏逻辑的编写上。

### 2.3.5游戏循环

整个游戏程序的核心流程控制，称为游戏循环。游戏不断按照顺序执行一系列动作，直至玩家退出游戏。

游戏在每一帧中执行一次游戏循环。每一次游戏循环就算作一帧。

核心流程依照开发游戏项目的不同而略有差异。在本项目开发中，依照本项目特点，使用如图2- 7所示的一套核心流程。



图2-7

1. 处理用户输入：检测玩家输入事件，如有事件则处理和分发，使得在当前帧里就可以及时响应结果。
2. 动画计算：执行所有注册的Action计算，更新结点的属性值。Action通常在之前的帧里面注册，具有比较高的处理优先级。
3. 物理模拟：处理碰撞，分发接触事件。
4. 自定义游戏逻辑：执行开发人员自定义的游戏逻辑代码，依照前几阶段处理的结果进行加工，检查游戏执行状态。
5. 遍历UI树：由游戏引擎遍历UI树，找出需要绘制的游戏元素。
6. 绘制：由OpenGL ES完成相关绘制工作。
7. 释放内存：基于引用计数，释放在该帧里创建的但是已经不再起作用的对象占用的内存。

* 了解游戏循环，可以清楚地知道所写代码实际的执行时机。程序员主要进行自定义逻辑的编写。应该意识到有些操作的结果并不会在当前帧立刻显现，而是会顺延至下一帧或更久。
* 游戏循环并不区分NonGameplayScene以及GameplayScene。这两个概念只是项目设计时用于区分两种设计思想差别较大的场景。事实上所有的场景都遵循同样的游戏循环，不同的是可能某些循环阶段在当前场景中并不会起效。

### 2.3.6事件分发

相比于直接的模块调用，事件不依赖于事件响应者。事件分发机制将事件的发送者以及响应者分开。事件的发送者只需要向Cocos2dx的事件分发器发送一个事件，事件的响应者可以向事件分发器订阅自己感兴趣的一类或几类事件。如图2-8所示。



图2-8

应用事件机制有以下显而易见的好处：

1. 事件机制使得游戏模块的实现不需要依赖于其他模块的实现，非常适合多人协同开发。
2. 事件机制极大地解除了模块间的耦合，使得模块独立程度更高。
3. 事件机制可以将事件分发给多个订阅者，提高了事件处理的灵活程度。
4. 事件的订阅者可以动态地修改，分发的事件也可以被拦截，进而阻止某些事件响应。在游戏编程中非常有意义。

* 在项目中并不是所有的模块交互都必须使用事件分发机制。事件分发机制虽然给应用程序编程带来了极高的灵活度，但是会损失部分性能。必须有限制的使用事件机制。
* 优先使用参数传递的方式进行模块交互。如果模块中的某些处理结果需要许多模块的响应，或者参数传递需要经过许多函数调用，则使用事件机制。

### 2.3.7物理引擎

对于一个平台跳跃闯关游戏，需要时时刻刻检测活动对象的状态，防止不同的物体相互穿透。实现检测的方法与机制有很多。但最直观和灵活，效果最好的当属利用物理引擎。

事实上现今的游戏开发早已离不开物理引擎，而且这些物理引擎多半都是作为第三方组件整合进游戏引擎中，亦或是单独使用。Cocos2dx集成了Chipmunk2D以及Box2D两大物理引擎，但是Cocos2dx只对Chipmunk2D进行了封装。所以说如果在Cocos2dx中使用封装好的物理引擎接口，实际上是在使用Chipmunk2D物理引擎。

物理引擎提供两大功能，碰撞检测和物理模拟。

碰撞检测：任何添加进入物理引擎管理的物体相互接触时，会进行一系列的检测操作。物理引擎能够区分对象的类型，使得某些碰撞被抛弃，抛弃的结果就是两个对象会互不影响地交叉而过。对于允许碰撞的两个对象，程序可以在对象接触时注册一系列的碰撞处理函数来对碰撞进行处理，就是给碰撞添加游戏逻辑。

物理模拟：允许碰撞的两个物体会进行碰撞模拟。物理引擎将根据经典力学对物体进行模拟，以产生真实的碰撞效果。可以设定物体的阻力，摩擦力，弹性等属性来定义不同的物体。

* 碰撞检测和物理模拟是本项目的重要功能模块。几乎所有在游戏中活动的对象都会接受物理引擎的管理。

### 2.3.8跨平台

Cocos2dx在生成可执行程序时，会依据不同的目标平台采用不同的编译与链接方式，使用不同的底层实现。针对不同的平台，也会使用不同的程序入口。这些都是经过引擎封装的。程序员只需进行一次游戏逻辑的编写，只需要很小的改动就可以将代码应用于不同的平台。

Cocos2dx整合的渲染引擎OpenGL ES，物理引擎Chipmunk2D都有各个平台的实现。Cocos2dx整合的第三方模块都是支持跨平台的。

* Cocos2dx自带的功能可能不足以满足本项目的要求，因此不可避免的要引入第三方库或其他的解决方案。为了跨平台，必须考虑到所引入解决方案的跨平台特性。不能够使用限定于某一平台的解决方案。

### 2.3.9缓存

在程序运行时，访问存储介质，将数据加载进入内存是非常耗时的操作，尤其是在游戏运行时读取大量数据，势必会影响游戏性能。所以几乎所有游戏会提前将此次游戏内所需资源加载进内存中。

Cocos2dx对资源的读取进行了封装。每次通过资源路径名读取资源时，如果缓存中没有此资源，则引擎先把资源加载进入缓存。再从缓存中返回资源对象。如果资源已经存在于缓存，则直接返回资源对象。

不仅如此，Cocos2dx在读取资源时可以进行一定程度的处理和优化。例如会将纹理大小扩展为2次幂，会依据plist文件只载入一大块图片中的某些部分等。最重要的是，游戏引擎负责把资源转换成为可以供渲染引擎使用的资源格式。例如不管图片资源的格式是png，jpg，还是bmp，游戏引擎都会将图片资源统一转化为渲染系统使用的数据格式，这也是让资源显示出来的唯一有效的方式。

* 即使游戏引擎优化了资源的处理和转换，仍然需要提前载入游戏所需的资源。因为游戏引擎并不能预知游戏到底需要哪些资源。
* 在本项目的游戏场景中需要加载大量的资源数据，在每次进入游戏场景前，通过扫描设定数据载入所有可能需要的资源，全部资源加载完毕后，再转入游戏场景。

### 2.3.10调度器

游戏中的对象有时候会需要进行不止一帧的更新操作，有时候某些函数将要持续执行好几秒。此时就需要使用调度器将自定义逻辑代码添加至调度器中。这样才能在每次游戏循环中执行自定义逻辑代码。

程序员可以显示指定需要调度执行的代码，可以指定调度的次数，调度的时间间隔等参数，调度的优先级等等。

每一段申请执行调度的代码都必须接受一个时间参数，表示上一帧所花费的实际时间。程序员可以依据这个时间来计时。通常情况下每一帧花费的时间都是相同的。

事实上游戏循环中的几个阶段都向调度器注册有逻辑更新代码。游戏引擎注册的调度更新往往优先执行。如图2-9所示。动画更新和物理模拟中很重要的一个概念就是插值运算。假如游戏逻辑要求某物体在10秒内移动10米。通过插值运算，即使某一帧因为性能原因花费了大量时间，假设是5秒，那么下一帧模拟时，动画更新和物理模拟就接受参数5秒，使用5秒这个参数更新游戏世界。在此例中，角色会在5秒后突然移动5米。如果固定了模拟的时间，那么角色在5秒后也只会向前移动1/60秒移动的距离。通过差值运算，游戏可以保证在性能不同的运行设备上都可以有较一致的游戏表现。不会因为设备性能过差导致游戏更新非常缓慢，也不会因为设备性能强劲导致游戏更新速度太快。



图2-9

* 因为每一帧花费的时间并不完全一致，所以自定义逻辑代码必须能够对传入的不同时间进行正确的更新处理，不能够假设执行每一段代码的时间间隔都是固定不变的。

### 2.3.11动作

动作系统也称为动画系统。主要是在每一帧根据时间和物体的状态更改物体的位置等属性，使得在一系列帧中物体能呈现出一组连续的动画。Cocos2dx提供了许多实用的动作。

几乎所有游戏引擎的动画系统都基于线性插值（Linear Interpolation）。线性插值接受两个已知的端点值和一个时间段中的时间点，返回该时间点在该两端点值之间对应的值。

在Cocos2dx中， Action类是动作类的基类，Action又继承于Ref类，每一个Action的子类都是一个独立的资源对象。每一个Action对象定义了一种修改。因为每一个Action都记录着执行Action对象的状态，所以每一个Action只能作用于一个UI树节点上。如果需要将同样的Action应用于多个UI树节点，则必须先clone出数个相同的Action对象，再分别应用。

动作系统支持控制动作的动作，Sequence动作能够组合多个子动作，子动作按照添加顺序一个一个执行；Spawn动作也可以包含多个子动作，但是所有的动作都是同时执行的，但不一定同时结束。

动作系统支持回调函数动作，可以像使用任何动作一样使用回调函数动作，也可以和任何动作组合起来使用。这样就可以在动作中穿插代码逻辑，而且根据游戏循环，其执行优先级往往比自定义游戏逻辑更高。

* 动作是项目中非常常用的功能，在使用时往往要组合其他动作。
* 执行回调函数动作，代码只会在下一帧开始执行。

## 2.4 GameData数据访问对象

### 2.4.1专用数据访问模块

为方便管理存档数据和设定数据，将读取数据从游戏逻辑中抽取出来，做成一个独立的数据模块。这个数据模块向游戏逻辑隐藏存档与设定数据的具体存储细节，并提供接口对象。

GameData实现为一个单例对象，供游戏逻辑使用。

GameData 中总共有两种类型的数据：设定数据，存档。但对游戏逻辑来说，它不需要分清 GameData 里面哪些是设定数据，哪些是存档；GameData 对象里面的就仅仅是数据而已，需要什么就读取什么。

设定数据就是记录游戏中各种对象的属性，例如敌人的生命值，道具的价格，主角的动画等等。

存档则是记录当前的游戏进度，玩家已经获得到了哪些道具，可以前往的地点等等数据。支持保存和切换。

相比与直接编写进代码中，将修改限定在单一文件中，使修改变得更容易且不易出错。

在项目中，所有的设定数据使用Json存储。要操作 JSON 文件，就需要一个 JSON 引擎。nlomann::json 有 STL-like 的操作接口。函数的命名和迭代器的使用方式都和 STL 一致，能够快速上手，而不用再去查它的手册。这是我们选用它的理由。最重要的是，nlomann::json，即Json For Modern C++支持跨平台。即使在不同的平台下也可以确保有一致的表现。



图2-10

具体实现中，由GameData对象持有所有设定数据的副本，并由GameData负责读取和写入相关的文件中。游戏中的其他对象只有通过GameData才可以访问存储在Json中的数据。如图2-10所示。

### 2.4.2不使用数据库+SQL

现在几乎所有的应用程序的存储方案用的都是数据库，但我们的游戏使用Json文件读取，主要考虑到以下因素：

1. 开发效率：

游戏中的数据天生是树形，JSON 天生也是树形，直接映射操作更方便

一个存档下有若干个可操纵的人物，有若干个已经解锁的地点，有若干个已经获得的装备，有若干个已经获得的符卡，每个人物可装备若干个道具，可装备若干个的符卡，一个地点下有若干个关卡，有若干个对话。

如果用数据库来存储，理论上每个一对多的关系都要都建一张表。以上那么多一对多的关系，再算上没提到的关系，我们要建的表的数量可能会达到 20多个，表数量的增多会在一定程度上增加维护和操作的成本。就算在实践中我们使用一些特殊方法将一些表合并，不太考虑范式等设计方法，使表的数量减少了，但这同时会使得数据的操作变得很不直观，难以理解。因为每查找一个数据就需要查阅数个表，给修改带来很大不便。

使用数据库得不偿失。如果用 JSON，那我们可以直接将这些树形的数据直接组织成一颗树，GameData 操作这棵树来读取和改动存档，比数据库操作要更简单明了。

1. 运行效率

JSON 引擎会将整个文件全部读入内存，并将数据在内存中组成树形结构，每次查找可以直接读取内存获取数据。

由于可以方便地记录一对多的关系，通常情况下有关数据都集中在一起，这也使得CPU缓存友好，缓存命中率更高。

对于游戏来说效率是非常重要的性能指标。将庞大的数据库系统应用在中小型的游戏中显然是冗余的，内存占用对于低端设备很不友好。

所以JSON文件读取效率 >> 文件型数据库查询效率。

1. 跨平台

即使不考虑运行效率和内存占用，也很难找到一个真正的跨平台使用的数据库。跨平台是一个不得不考虑的因素。

Sql Lite是一个可行的跨平台解决方案，但是正如前所说，不支持一对多的数据存储。即使不考虑一对多，对于团队协作也带来了额外负担。相比于通过GitHub，团队直接同步json文件，数据库专用存储文件难以直观阅读的特性使得代码合并变得不可预测，冲突难以解决。如果通过网络同步数据库，问题是团队没有时常开启的能够充当数据库的设备，数据库维护成了额外的负担。

所以从开发效率，运行效率，和跨平台三方面考虑，我们使用JSON存储游戏数据。

* 事实上，有很多的现行商业游戏不使用专业的数据库软件来存储游戏数据。任何能够记录数据的文件，甚至是Excel也被用来存储游戏中的数据。从成本考虑，从开发效率考虑，直接读写文件都是可行的做法。

## 2.5 Resource资源组织

游戏中使用到的数据大致有以下三种。

1. Json设定数据
2. 宏定义头文件
3. 图片
4. 音频

此三类数据统称为资源。其中 Json设定数据 和 宏定义头文件 记录如何使用图片和音频。

资源使用易于辨识的目录组织起来，全部放在Resource文件夹中。本项目使用的资源目录组织如下：

|  |  |
| --- | --- |
| 目录 | 说明 |
| avatar | 游戏中主要角色的头像和立绘 |
| background | 供各个场景使用的背景图片 |
| bgm | 背景音乐 |
| character | 游戏中角色使用的帧动画 |
| Effect | 游戏场景播放的特效 |
| emitter | 发射器所需的子弹图片 |
| enemy | 敌人使用的帧动画 |
| fonts | 游戏使用的字体 |
| gamedata | Json设定数据以及存档 |
| gameplayScene | 游戏场景专用的各种素材，包括地图和纹理 |
| item | 道具图标 |
| menu | 菜单界面用到的纹理 |
| particle | 粒子系统使用的图片 |
| se | 声效 |
| shaders | 自定义着色器 |
| Spellcard | 符卡图标 |
| wordArt | 艺术字 |

## 2.6 Lua语言集成

随着项目的开发，代码行数会越来愈多。即使采取各种措施降低模块间编译依赖，但仍然会消耗大量的时间在重新编译与链接上。对于需要经常改动查看效果的代码问题尤为严重，慢速的项目编译构建速度影响了项目的迭代速度，我们需要一种不重新构建项目即可更改项目内各种逻辑的方法。

使用JSON记录数据，在游戏中读取JSON数据是一个好办法。但最初引入 JSON 是为了解决存档与数据存储问题，因为 JSON 天生的树形结构很适合复杂数据的表达，但 JSON 在游戏逻辑表达方面有所欠缺。

由于游戏逻辑并不如数据存储一样具有规律性，使用JSON记录数据块来代表代码逻辑的话，需要增设大量的标记位以表示游戏逻辑以及参数。因为只有GameData才可以访问Json文件，不同游戏逻辑的区分只能交给GameData来处理，但是这与GameData只负责数据访问的职责相违背，GameData不应该负责游戏逻辑的处理。为了使数据访问与游戏逻辑相隔离，我们不能再进一步设计更复杂的 JSON 表达形式。

我们需要表达能力更强的 嵌入式编程语言 来解决这个问题。Lua 是经过综合考虑之后的选择。Lua 与 C/C++ 的亲和力很强，混合编程较为容易，因为 Lua 从一开始就是作为 C 语言项目的嵌入脚本来设计的。

在项目中使用Lua，主要考虑过以下几点特性：

1. 无需重新编译：Lua是一门解释执行的脚本语言，可以在不改动宿主程序代码的情况下进行修改，甚至在程序运行时修改。
2. 可移植：Lua几乎可以运行在任何平台上。
3. 易于使用：Lua语法简单，使用灵活，没有太多复杂特性。
4. 健壮：如果使用Lua编写游戏逻辑，即使出现了错误，游戏程序也不会立刻崩溃。 错误能够立刻察觉出来。

* 虽然Lua有很多的好处，也早已成功应用于游戏开发中，但是应用Lua仍然需要一定程度的前期学习。出于实用角度考虑，本项目只会将Lua应用于需要经常改变并调试的功能里。

## 2.7 工具类

### 2.7.1音频控制器

Cocos2dx引擎团队在制作引擎时也同时整合了同一团队制作的声音播放引擎SimpleAudioEngine。这个引擎提供了最基本的音频操作功能。例如播放音乐，音效，调整音量大小等，对于简单的游戏项目来说足够使用。

但是该声音引擎完全不含任何与游戏逻辑相关的内容，例如无法回放上一首播放的音乐，无法暂停与恢复音乐的播放，无法区分当前播放的音乐与将要播放的音乐是否相同等。

游戏充满了大量的场景切换，伴随着的是不同背景音乐间的切换，内置声音引擎本身不足以满足项目实现要求。

虽然Cocos2dx最新版已整合更完善的声音引擎，但是仍处于试验阶段；引入第三方声音引擎工作量又太大。所以本项目决定使用AudioController类对SimpleAudioEngine做进一步的封装。

AudioController类记录当前播放的音乐，先前播放的音乐以实现音乐区分的功能。能够自动判断当前背景音乐和将要播放的背景音乐是否相同，如果是同一首音乐，则不会从头再播放音乐而是继续播放音乐。支持记录先前的音乐，用于在播放大量背景音乐后再自动返回指定的音乐。提供预设音效，例如封装播放点击音效的代码，使得修改仅仅局限于音频控制器内。

封装最大的意义是将判断音乐是否播放的逻辑转移到控制器内，而不是分散在每一个需要播放音乐的场景内。

### 2.7.2点击粒子

点击粒子是指在屏幕上的空白区域点击时会显示的粒子特效。在屏幕上添加粒子的代码经过封装。只需要在任何需要类似需求的代码处执行此段代码，任何一个方框，层，甚至是场景都可以注册点击粒子特效。

点击粒子被设计为一种特殊的事件监听器，专门监听点击事件，拥有足够的优先级以相应事件，也不会阻止事件的传递。

播放的粒子特效完全支持定制。

### 2.7.3占位符

为支持快速原型开发，同时使得程序编写工作不需要受制于素材，在缺乏素材的时候需要使用占位符来替代，可以等待以后素材就位后再逐一替换。

占位符类封装了Cocos2dx的简单绘制函数。正常来说没有游戏会直接使用这些简陋的图形，但是在开发阶段却很有用。

通过占位符类，可以创建矩形或圆，可以添加文字说明，可设置背景颜色，文字颜色，图形大小。

* 使用这些简单的图块来临时代替美术素材。

## 2.8 层

层有别于场景。同一时刻游戏中只能有一个场景，却可以存在许多的层次。出于简洁考虑，一些功能被设置为依附于层次的。好处就是几乎可以在任何场景调用这些层次。

### 2.8.1设置层

如果要能够修改全局的属性，那么就需要能够访问到全局的属性。层的特性使得玩家可以在游戏中任何地方进入。所以在层中不仅能够方便地修改一些全局属性，更可以把层当作一个跳转中心。

* 如果从游戏场景进入设置层，设置层需要负责暂停游戏。

### 2.8.2对话层

对话层按照故事脚本显示文本，显示人物，播放音乐和音效。

对话层能够在不同的地方被调用，很适合用作故事讲述和背景介绍。

对话层最核心的功能是读取故事脚本，在本项目中使用Lua语言编写故事脚本。

* 只有对话层使用Lua语言编写的脚本。

### 2.8.3确认按钮

有一些操作，玩家点击后并不会立刻生效，而是需要进一步确认或取消。看似是一个很基础的功能，但是会对架构设计产生重大影响。这意味着操作的触发和操作的处理必须分离开。

为了达到触发与实现分离的效果，增加了确认按钮。之所以把确认按钮当作一个层来实现，是因为确认按钮具有许多符合层的特性。

触发确认按钮的模块必须提供两个操作，一个是希望执行的操作，另一个是操作执行完之后本模块应进行的处理。点击确认，就会立刻执行提供的操作。如果点击取消，那么什么事也不会进行。如图2-11所示。



图2-11

## 2.9 场景组织

我们将游戏中的所有场景大致分为两类：

1. Non-Gameplay 场景：指道具库，主界面，设置，存档界面等那些非玩家操作角色实际冒险的场景。
2. Gameplay 场景：指玩家操作角色实际冒险的场景。

这么分类的原因主要是这两类场景实现方式明显不同，运行机制明显不同。一个是简单的菜单组合，一个是地形，人工智能，弹幕，物理引擎，用户操作等组件之间的事件检测、分发和处理。

游戏中所有的场景跳转关系如图2-12所示。

图2-12

### 2.8.1场景缓存

在Cocos2dx中，创建并切换场景是一个耗时的工作。如果能够将已经创建的场景缓存起来，那么接下来就不需要再重复创建场景，可以直接进行场景切换。

据此引入场景缓存，Cocos2dx原本并未实现此类功能。由项目成员自行设计并使用。

使用很简单，每次创建的场景指针都会加入一个容器中。每次切换场景前都从缓存中查找场景指针，取出并替换即可。

* 由于游戏场景会占用大量内存，当从非游戏场景切换到游戏场景时，需要清空场景缓存。

### 2.8.2场景栈

只有通过导演类才能够进行场景间的切换。导演类提供三种切换场景的方法：push，pop和replace。场景栈的操作如图2-13所示。

Push操作将指定场景推入由导演类管理的栈顶中，并切换进指定场景。

Pop操作则将栈顶场景弹出，并切换进原栈顶下方的场景。

Replace操作直接将栈顶场景替换为制定场景。



图2-13

需要注意的是如果当前场景栈为空，就会退出游戏。原则上只要场景栈不为空就不会出现异常问题，但是不加以约束的往场景栈添加场景也会使得场景栈过于庞大。场景栈的管理只能由程序员通过约定来进行。

通过观察场景组织结构，可以发现存在一对多，一对一，单向，双向的场景。场景栈管理就是指控制场景栈尽量小。

对于跳转顺序严格固定的场景，直接使用Replace。

对于有多个出口或入口的场景，如果需要返回，就需要配合使用Push以及Pop。如果不需要返回，则直接使用Replace。

当进入游戏场景时都需要清空场景栈，只保留一个游戏场景。

* 场景栈和场景缓存是不同的。场景缓存只是为了减少创建场景的耗时，存放的都是静态场景，而且存储的都是场景指针。而场景栈只关心场景的跳转关系，同样保存的是场景指针。

### 2.8.3场景生命周期

Cocos2dx中，一个对象并不总是直接加载完成的。在一个对象的生命周期中，可能经历许多的添加和被移除的过程。当一个对象仅仅是暂时移除但是并不需要释放内存时，就需要进行一些特殊处理。同理，当一个对象被添加时，也需要进行一些特殊的处理。因为有些处理只有在对象处在UI树中才有意义。

以场景的生命周期为例，当一个场景被创建时，仅仅拥有最基本的静态布局，被添加进项目组设计的场景缓存中。当该场景被拿来使用时，就需要再读取GameData取得最新的动态数据，并将这些数据添加进场景中。当场景被替换后，就需要将这些动态数据移除。如图2-14所示。

通过场景栈和场景缓存管理的场景在内存中只有一个副本，任何修改都是针对于这一个副本。



图2-14

### 2.8.4跳转表

跳转关系是自上而下的，不同人开发不同场景时势必要根据自己的场景重定向跳转以便于测试，而这种跳转重定向在代码层面是很难合并到一起。

所以我们额外增加一个场景 JumpTableScene，统一地来做跳转重定向，只要在 JumpTableScene 里面点击相应的按钮就可以跳转到相应场景。

这样，就可以并行开发所有场景。只是其他场景没有完成的时候，跳转到其他场景的代码无法正常运行，甚至不能通过编译。

所以，先给所有的场景一个统一的空实现，不含任何内容。

### 2.8.5界面布局

每一个添加进UI树的节点都要指定其相对于父类的坐标，而且还要指定其在父类节点里的层级关系。

如果将所有的节点都添加进一个根节点中，随着界面元素的增多，越来越复杂的层级关系和越来越复杂的坐标定位会使得代码非常难以维护。

一个容易理解和维护的界面编码方式为将相关子节点统一添加进一个父类节点中，该父类节点就可以称为布局。布局可以是一个层，也可以是一个简单的Sprite对象，使用方式非常灵活，但原理都是封装。这样每次修改界面时，可以将修改限定在某些布局内，而不会移动其他布局。如果修改布局的位置，布局所有子节点的位置也会同时移动，非常方便。如图2-15所示。

图2-15

### 2.8.6资源加载层

游戏场景是占用资源最多的场景，每次进入游戏场景都需要将大量的数据读取进内存中。在此期间会造成游戏的严重卡顿，甚至假死。在性能不足的移动设备上尤为明显。所以需要增加额外的层次隔离这种不良观感。

资源加载层就是一类专用的层次。游戏场景的创建，初始化的工作都由资源加载层完成。每完成一类初始化工作，资源加载层就会移动进度条显示加载进度。这样用户就可以直观地看到目前的加载进度，虽然游戏可能仍然会假死，但是玩家能够看到进度条的变化，不会轻易把游戏关掉。

游戏场景是一个需要严格控制构建顺序的场景。如图2-16所示。



图2-16

## 2.10游戏场景

游戏场景应用到了最多的开发技术。其复杂程度使得很难一次性完成全部设计，故设计方法和过程将占很大比重。

### 2.10.1游戏逻辑和架构

Cocos2dx引擎本身不包含任何的游戏逻辑。即Cocos2dx本身没有内置任何的游戏执行流程和目标管理。Cocos2dx不为任何特定游戏类型设计。程序员必须手动检查游戏状态，编写玩家需要达成什么样的目标才算做过关等。由于Cocos2dx为了使引擎更加通用，所以没有提供任何方便特定游戏类型实现的功能。

很多成熟的游戏引擎会将游戏逻辑整合进引擎，或者支持将游戏逻辑当作组件，依据需要增添改删。著名的例子有RpgMaker，专门开发角色扮演游戏的引擎。还有虚幻引擎，特别适合3d游戏，尤其是射击游戏的开发。

因为不同类型的游戏除了某些基础的绘制模块或物理模块可能是通用的，在此之上的任何特定游戏逻辑都是截然不同的。即使两款游戏都有赛车模块，但因为游戏重心的不同，一个可能是专业的赛车模拟游戏，一个可能更加偏向于娱乐性。前者的逻辑复杂程度自然而然比后者高得多。

很多的商业引擎支持快速定制，只要缴纳一定的费用，就可以获得一套适用的引擎，专门用于开发某些类型的游戏。这样开发游戏特别类似现行的网站开发。架构已经设计完成，团队可以立即投入内容的开发工作中。

然而Cocos2dx的定位，免费，开源，决定了项目团队不会得到引擎开发团队的支持，有问题只能找社区或者内部解决，毕竟开源。

所以使用Cocos2dx开发游戏，必须从头到尾设计整个架构。当然架构在此特指游戏场景这一个场景的内部组织结构。

本项目组开发的游戏逻辑很简单，玩家只需要经过一系列的平台跳跃，击败沿途的敌人，最终打到关底BOSS就可以视为通过当前关卡。

但是为了支持平台跳跃，必须引入地图，物理引擎等特性；为了击败敌人，又必须引入伤害检测机制以及攻击方式；为了打倒BOSS通关，又必须增加目标检测。

正因为复杂度如此之高，项目团队决定对游戏场景采用原型开发。即已熟悉技术为主，先开发一个可以执行的原型，可能很简陋，甚至达不到要求，但是可以方便的增加新功能。

最终设计完成的游戏场景架构如图2-17所示。



图2-17

### 2.10.1地图层

整个游戏场景最基础的部分就是地图层。地图层位于场景中最中心的位置。游戏中大部分活跃的对象均是地图层的子节点。

在同一个层次中的子节点相互之间通信比较方便。

### 2.10.2瓦片地图

瓦片地图可以由许多图块层以及许多对象层组合而成。之所以称为瓦片，是因为整个地图是由许许多多的基础图块重复组合而成的。每一个图块在地图中都有一个唯一的编号用以区分其他图块。

图块层由许多大小相等的图块组成，通过图块的组合来绘制出游戏的地形。实际上在图块层上绘制的所有图块都仅仅是图块，不含任何的游戏逻辑信息，例如此处为墙，此处为地板。

如果需要给图块指定游戏逻辑信息，可以给图块所属的图块层添加标记信息，但这样做的话会影响到这个层的所有图块。如果指定一个图块层为不可穿越的，那么这个图块层的所有图块都不能穿越。这就要求必须使用不同的图块层分开绘制地形信息和装饰物信息。

虽然不能直接访问到图块层的某个图块的信息，由于每个图块都有唯一编号，也可以直接检测图块的编号，然后根据图块的意义进行处理。

但是更好的实践是使用对象层来承载游戏逻辑，而图块层仅仅用于绘制，这样就可以将地形绘制和游戏逻辑分隔开来。以后即使需要修改游戏逻辑信息，只需要修改对象层信息，而不需要修改图块层，因为图块层的绘制比较麻烦，精美的图形需要经过精心的打磨，不方便随意变动。

对象层可以由任意多个，一般会给每一类游戏对象专门指定一个层次。例如给玩家角色一个层次，给游戏中的敌人一个层次，给游戏中可以活动的其他对象一个层次。在每个层次中，可以简单地划分某一坐标或某一区块，然后添加自定义的属性信息，例如指定某一坐标为某一敌人的刷新位置。对象层中的信息都不会参与绘制，对象层的用处就是提供标记信息供游戏读取。

瓦片地图实质上是XML文件，而且也不是Cocos2dx专用的。Cocos2dx封装的TMXTiledMap类提供了方便的接口，用于解析瓦片地图中的图块层和对象层。在代码中创建地图时，返回的并不是一个层，而是一个继承自Node类的一个TMXTiledMap对象，需要再将这个对象添加到地图层中。该对象含有所有的瓦片地图信息，其中图块层已经全部绘制出来。但对象层的信息仍然需要程序员读取并解析。

### 2.10.3物理世界

由于Cocos2dx引擎只是整合了第三方的物理引擎，引擎并没有提供原生实现，物理引擎本身就是一个可选项，所以创建场景时必须指定允许启动物理引擎。

当启动物理引擎时，由于物理引擎本身就可以独立于游戏运行，物理引擎所处的“世界”和场景所处的“世界”本身并无对应关系，由游戏引擎负责将物理引擎中的对象和UI树上的对象绑定起来。

刚体代表了在物理世界里的一个对象，刚体只是一个容器，真正参与物理模拟的是依附于刚体的形状。只要给UI树上的对象绑定刚体，物理引擎就可以和Cocos2dx的UI树共同工作。

在物理世界中，刚体的驱动完全取决于所受到的力。这就使得游戏中刚体的移动变得很真实。刚体会受到摩擦力，阻力等的作用。

地图中会有许多玩家不可以穿越的区域，例如地面，墙壁等，统称阻挡区域。而这些阻挡区域通常使用物理引擎制作。在瓦片地图的对象层中，可以划定某些地形为不可穿越区域。

当加载地图时，由程序员读取地图上的不可穿越区域，在这些区域上生成一些不可穿越的，透明的静态刚体。这些刚体完全不受任何外界影响，就静静地出现在我们在地图中为它们规划好的区域。当角色的刚体触碰到这些静态刚体后，角色刚体会被阻挡，这样就可以实现阻挡的效果。

### 2.10.4摄像机

当地图创建完毕后，就需要想办法把地图动起来。可以想象整个地图就是一个巨大的画面，但是玩家看到的屏幕只是显示这个巨大画面的一个部分而已。当角色移动时，玩家应该能够看到地图的不同区域。这称为视角跟随。

cocos2dx提供了一个继承自Action的Follow类，可以非常容易地实现可视区域的自动跟随。创建跟随类的同时要指定跟随的节点对象，以及跟随的区域。跟随的节点对象就是玩家的角色，区域就是整个地图的宽和高。指定跟随的元素必须在同一个层中，即在地图层中。



图2-18

这个跟随类实质是一个Action，执行这个Action的是承载游戏角色的那个父节点对象。当被跟随的对象在父节点中的位置改变时，该Action将保证被跟随的对象始终保持在屏幕正中心的位置。实际上这个Action采用了相对运动的原理，当玩家向右移动时，父节点对象，即地图层会向左同步移动相同的距离。由于Cocos2dx整合了物理引擎，在移动整个地图层时也会同步移动物理刚体，达到物理引擎和UI树的协作效果。

一个需要理解的关键点是，整个游戏的绝对坐标系从来不会改变。也就是说玩家观看游戏的“窗口”位置绝对不会改变。如果要在窗口中显示不同的内容，就必须移动窗口内的对象。如图2-18所示。

然而这个Follow类会将所追踪的Node元素至于屏幕的正中间，在很多情况下，无关紧要的空白地形区域会占据至少五分之二的界面，十分影响画面观感。

故引入一个摄像机节点，将摄像机添加进地图层中，整个视角跟随的是摄像机的位置，而不是玩家的位置。我们需要做的就是根据玩家的位置更新摄像机的位置即可。摄像机可以放在角色的右上方位置，这样可以展示更多的右部区域。对摄像机位置的更新也很有讲究，例如如果角色快速下降，摄像机应该能够转移到角色下方，好让玩家观察到地图下方的内容。

摄像机不仅用来作为被视角跟随的对象，还用来记录地图层的偏移量。而偏移量将作为以后制作视差节点时重要的参数。

### 2.10.5活动对象

所谓活动对象就是能够显示在地图层中并有自身处理逻辑的对象。活动的意义是强调这个对象是有行为的，而不论行为是主动地还是被动的。

在我们的游戏项目中，所有角色，所有敌人，所有机关全被称为活动对象。这些对象全部添加进UI树中，自身封装了许多的属性值和逻辑信息。

对于角色，角色能够接受用户的操作而执行很多的动作，而在地图层看来角色仅仅是一个UI树的节点。只有活动对象才可以使用状态机，加载帧动画，以及设置发射器。

### 2.10.6触摸监听器

为了能够在游戏场景中接收到玩家的操作信息，需要给接受用户操作的节点设置触碰监听器。触摸监听器是一种特殊的监听器类型，相比于普通的事件监听器只能接受一个事件对象，触摸监听器还可以接受到触摸点的信息。

Cocos2dx只提供了四种触摸操作的事件监听器，没有提供双击，长按等操作的监听器，只能自己来编写。

游戏引擎将触摸事件分发作为一种默认行为。当触摸刚按下，触摸移动，触摸取消，触摸结束时，游戏引擎都会自动分发这4种触摸事件对象。所以如果需要相应这4种触摸事件，就需要给监听器设置对应的事件处理函数。

本游戏将整个屏幕分为三个主要触摸区域，由于触摸监听器能够接收到触摸位置的信息，所以可以简单的判断玩家触摸了哪些区域，并执行相应的动作。游戏场景支持的触摸区域划分如图2-19所示。



图2-19

顺带一提，Cocos2dx同样支持键盘操作，按下键盘按键后同样会分发4种按键事件。

### 2.10.7控制面板

控制面板的实现为一个优先级非常高的层。该层次直接作为场景的子节点，永远覆盖在地图层之上。事实上触摸监听器就是注册在控制面板中，由控制面板接受触摸操作。

除了接受触摸操作，控制面板还有各种按钮，代表角色可以执行的其他操作。这些按钮同样注册有触摸监听器，当点击这些按钮后，触摸事件就不会再继续分发。

控制面板也负责显示玩家的血条，敌人的血条等信息，这些信息并不响应点击。也提供返回按钮方便地退出游戏。

需要注意的是，由于控制面板和活动对象根本不是一个层次的UI节点。彼此很难直接访问到对方。相互间通信通过自定义事件分发机制，最大限度的保证模块间的独立性。

### 2.10.8帧动画

动画实际上就是由一幅幅图片连续播放组合而成的。Cocos2dx整合了完善的动画播放机制，能够指定动画播放的持续时间，帧间隔，缓存动画等。但是Cocos2dx没有提供任何切换动画播放的逻辑，一个动画播放完就播放完了，不会产生其他的任何动作。甚至如果没有停止掉先前的动画就播放新的动画，角色的动画就会在两个动画间交替播放，造成非常差的观感。就如同Cocos2dx内置的音频系统一样，必须由程序员手动实现控制动画的停止和播放逻辑。

角色在不同的状态下应该播放不同的动画。实际上就是不停切换角色上显示的图片。在游戏场景的中有专门的一个初始化工作就是生成动画。例如角色行走的动画，角色向上跳跃的动画。

在项目应用中，角色动画的切换是最复杂的一种，因为根本无法预测玩家接下来的操作，也就是说无法预先设定好动画序列。

可以为每一种动画设置一个标志位，在一段每帧都执行的代码中根据角色当前的运动状态，切换角色应该执行的动画。角色的标志位除了正在执行的动作，还应该有角色面朝的方向，这样才可以正确处理角色的转向。

也可以将这些动画切换的操作放在各个操作的处理代码中，例如如果点击了跳跃，那么就在使角色跳跃的同时切换动画。但是如果跳跃之后就不操作了，还是需要根据角色之后的状态重新设置动画。这样做虽然好理解，还不需要加标志位，但是切换动画的操作很分散，代码不容易维护。

处理难以预测的动画切换，关键是保证先前播放的动画在播放新动画前首先停止。比较好的解决方案是使用状态机。

### 2.10.9弹幕系统

弹幕系统的作用就是在屏幕上有规律地发射子弹。弹幕系统分两层。

第一层为发射器节点，发射器本身不具备任何图像信息，只是作为子节点添加进活动对象中。这么做的原因是可以像操控其他UI树节点一样操控发射器节点，达到随时添加和移除的目的。由于发射器和活动对象都处在同一UI树中，子节点可以非常方便的获取到活动对象的节点信息。

发射器节点接受活动对象发来的样式配置，而样式配置表示子弹应该怎么样发射。发射器负责分析样式配置，并将样式配置正确地发送给对应的样式。如果传入了一个不存在的样式，那么发射器就会简单的抛弃这个样式。

第二层为样式，所有的子弹都是在这里创建并添加进地图层中。样式中封装了实际的子弹创建和发射逻辑。

弹幕系统的架构如图2-20所示。



图2-20

### 2.10.10互动对象

互动对象不同于活动对象，互动对象并不隶属于地图层，而是作为控制层的子节点。

在控制面板一节中提到过，控制面板除了接受触摸操作，还有各种按钮和负责显示信息的子节点。它们都称为互动对象。

主要有两大类互动对象，冷却按钮和血条。 表面看来控制面板上只是一些普通的按钮和图片，但是它们内部都封装有响应逻辑。

冷却按钮往往对应游戏中的一类道具或者技能，能够自动在点击后将自身禁用，并在冷却时间结束后重新启用；不仅如此，冷却按钮还能够修改自身外观，并实时显示剩余冷却时间；最重要的是，按钮可以分发事件，将逻辑处理和按钮的事件分发隔离开。控制面板并不应该负责游戏逻辑的实现。

血条对应游戏中的具有生命值属性的活动对象，用来显示玩家的生命值，魔法值，也用来显示敌人的血量。血条有经过设计的外观，能够依据实时的游戏信息修改自身长度，直观地显示血量的增长和减少。由于血条和活动对象不在同一层次中，两者通过事件分发传递信息。

### 2.10.11着色器

很多时候如果要更改图像的颜色，就需要准备许多不同颜色的素材，但是如果这些素材仅仅是颜色存在差异。如果这样的话，不仅美工的工作量剧增，代码也要同时维护同一种动画的不同颜色版本，最重要的是会占用更多的存储空间和内存。这里提及的颜色存在差异是指的图像整体，例如整体偏暗，整体发红，整体变蓝等等。

这个时候就必须从渲染的角度考虑。所有参与绘制的图片都有一个统一的阶段进行上色。这里不再加以详细说明，只需要知道上色的过程就是颜色混合的过程。在RGBA颜色空间中，A代表透明度。如果能给一个图片额外的混色过程，就可以动态的修改图片显示的颜色。着色器很适合应用于此种情形。

在项目中需要实现角色受到攻击后，身体会闪红。着色器能够在任意时刻启动，交替变换颜色与透明色和红色的混合。那么角色在每一帧绘制时就会在红色和原色间切换。

着色器并不是只能混合颜色，还可以用来颜色过滤。有些时候美术素材会附带背景色，或许是灰色，或许是黑色，白色。但是在游戏渲染引擎看来，这些背景色都是图片的一部分。如果将这些图片放进游戏中，就会看到一个包含大量背景色的丑陋的矩形图片。通过给需要参与绘制的UI树节点设置颜色混色方式，就可以在渲染阶段过滤掉背景色。

### 2.10.12碰撞监听器

使用物理引擎进行碰撞检测和处理的关键是正确地设置三种掩码。刚体和形状都可以被设置掩码，但一般给刚体设置掩码就可以了，因为一般一个刚体就只有一个形状。

种别掩码categoryBitmask，默认值为0xFFFFFFFF；

接触掩码contackBitmask,默认值为0x0;

碰撞掩码collisionBitmask,默认值为0xFFFFFFFF;

每次两个刚体接触时，物理引擎会拿各自的种别掩码和对方接触掩码进行逻辑AND操作，结果为true的话就会执行接触回调。接触掩码默认值为0，所以默认不会进行接触回调。

当两物体接触的时候一般都会发生碰撞，但是这个碰撞是否进行还需要进行检测。将种别掩码和对方的碰撞掩码进行逻辑AND操作，返回true才进行物理模拟。与接触不同的是，两个刚体碰撞可能只有一个会发生物理模拟，而另一个没有反应，所以每一个需要发生物理模拟的刚体都必须正确地设置碰撞掩码。

可以创建一个碰撞监听器，并注册一组接触回调函数控制接触和碰撞的全过程。如果一开始什么掩码也不设置，接触掩码默认为0，碰撞掩码默认为0xFFFFFFFF。你可能会以为要进行碰撞检测，必须经过接触检测，那么默认情况下不会进行接触回调，那么就不会有这个物理模拟发生。然而默认情况下几乎每个刚体都会进行默认的碰撞的物理模拟。

重点是，接触判断完全是可选的。接触判断发生在物理模拟之前，用于决定是否放弃进行物理模拟。接触回调函数是物理引擎给程序员的接口，便于开发者精确控制两个物体间的碰撞的行为，开发者可以主动控制碰撞的各个阶段，来实现一些可能是物理引擎之外的，和游戏逻辑相关的操作。

如果不写这几个回调函数，那么就是简单地按照前面说的用掩码来判断，来进行默认的碰撞模拟，不会触发接触回调。如果写了这么几个回调函数，并且成功添加监听器，但是你只是简单地返回true或者false，如果设置的合适，那么效果和用默认的物理模拟没什么不同。所以注册这几个回调函数的主要目的就是能够实现一下游戏逻辑相关的操作。

### 2.10.13自定义事件监听器

不同于触摸事件和物理接触事件，自定义事件只能由程序员自己负责创建，发送和接受。

相比于触摸监听器和碰撞监听器，在接受事件的同时都会接收到完整的触摸或者碰撞信息，自定义事件想要传递数据，只能由程序员手动填写进事件中。

不填写数据就分发事件，也可以成功被监听器接收，就像无参函数调用。但是大多数情况下还是希望事件能够携带一些数据。

监听器接收到事件后并取出数据，就可以拿着数据进行正确的逻辑处理。

### 2.10.14区域

随着地图规模的扩大，越来越多的对象将被添加进入地图层中。然而玩家同一时间只能在地图中的一小部分活动，许多屏幕外的对象都在浪费计算量。这就要求对地图进行合理的分块，同一时刻游戏只需要专注于某一区域的模拟。

区域划分在瓦片地图中完成，由对象层指定区域的范围。

划分区域的额外好处是不需要再制作非常连贯的地图，而且能够动态地更改玩家所在地区的背景图片，播放不同的背景音乐等。

同时区域系统还将和视差节点紧密配合，一起提升游戏的观感。

### 2.10.15视差节点

为了营造出整个游戏场景画面的层次感，需要将整个游戏画面的背景与地图层分离。

如果需要突出层次感，最重要的就是距离感。假如坐在快速移动的车上向外看，近处的物体看起来总是比远处的物体移动速度快，这就是视差原理。如图2-21所示。星星移动了10个像素，而三角形只移动了5个像素。



图2-21视差原理



图2-22

游戏项目中将视差原理同时应用于前景和背景。实际上就是在地图层移动的时候，同时移动前景和背景，只是前景移动最快，背景移动最慢。如果地图层移动了100像素，那么前景可能会移动200像素，背景只移动了10像素。

由于这三个层次完全不是同步移动，所以前景层和背景层不能够添加进地图层中。作为单独的层次分别位于地图层的前方和后方。如图2-22所示。

但是游戏中的景物除了具有远近关系外，还必须正确的处理遮挡关系。有些景物不一定要移动的很快，只是为了体现遮挡的效果，那么就必须再引入和地图层同步移动的层次，称为静态背景或者前景。从名字就可以看出来，这两个层次就是专门用来处理遮挡关系的。

所有静态背景的图片都会被地图层中的图像遮挡住，而静态前景会遮挡所有地图层中的图像。

### 2.10.16特效

游戏画面中除了包含帧动画和瓦片地图的图块，还包含其他的装饰性画面表现，这些画面表现一般都被称为特效。

在项目中包含的特效为粒子特效和贴图特效。

使用Cocos2dx的粒子引擎可以很容易的设置粒子特效。对增强画面表现有非常大的作用。但是在项目中只将粒子特效应用于打击特效中，因为项目组没有美工，准确来说是没钱请美工，粒子特效对性能和美术素材的要求都很高，项目组只能从简应用。

贴图特效相比粒子特效方便许多，本质上就是帧动画，但是可以添加进任何UI树节点中。通常用来对一些特殊动作提供反馈信息，当这些动作触发后，将自动播放一段帧动画。

### 2.10.17状态机

状态是人为划分的，例如角色行走是一个状态，跳跃又是一个状态。如果要在不同的状态间正确切换，就必须正确处理不同状态间的差异。例如停止先前的动画，并重设某些属性值。

状态机封装了状态，每一个状态有一个入口和一个出口。入口用于指定进入当前状态时需要进行的处理，而出口表示退出当前状态时需要进行的收尾操作。执行则表示角色在当前状态下持续进行的动作。

状态机保证每一个状态切换时都会先进入出口，然后再进入另一个状态的入口。如图2-23所示。

状态机在项目中应用于动画切换，角色的操作状态切换，以及敌人的人工智能。



图2-23

### 2.10.18事件脚本处理

事件脚本指一段预先设定好的动作流程，用在游戏场景中，可以在游戏场景中添加对话层，也可以对角色数据进行更新，甚至代替玩家执行一系列的操作。可以说事件脚本就是个“剧本”，而剧本决定了游戏的发展方向。

事件脚本处理能够按照剧本顺序执行，并能够在剧本执行完毕后正确返回，将控制权再次交给玩家。

剧本存储在json文件中，通过GameData访问，而剧本的解析和执行则由事件脚本处理类来负责。事件脚本处理类能够：

1. 触发对话：在游戏场景中加载对话层，用于故事讲述。
2. 操作指定活动对象：指定对象将按照剧本向指定位置移动，或进行指定操作，起到剧情演出的效果。
3. 更新游戏：每当通关后，游戏进度应该有所进展，玩家应该得到更多的道具，获得更多的金钱等，这些更新都反应在游戏存档上，原本由GameData 负责对存档的操作。事件脚本允许在游戏场景中更新游戏进度，而不一定要等待游戏通关。

### 2.10.19事件过滤

事件过滤利用事件接收的优先级，通过指定“最高”优先级的监听器，达到优先接受事件的效果，从而决定进行额外的逻辑处理，还是停止继续分发事件。

指定监听器优先级并不需要专门封装一个类。但是经过封装后就可以添加许多方便的功能。通过事件过滤类可以统一管理所有程序员指定的“最高”优先级监听器，最主要的是指定过滤事件的持续时间。

事件过滤器只会监听固定的一些事件类型，实质上就是提前注册了一些最高优先级的监听器，优先接受感兴趣的事件。当接收到事件，在事件过滤器内部将事件传递给所有事件过滤器管理的后续添加的监听器。如果不存在需要响应现有事件的监听器，就继续传播事件。如果在事件过滤器中，响应事件的监听器终止了销毁了事件，事件就不会再继续向下传播。

使用事件过滤器，可以指定角色在一定时间内不会受到伤害，因为伤害事件全在过滤器内部被拦截。当指定时间到达后，事件过滤器自动移除伤害过滤监听器，使得角色可以再次受到伤害。

事件过滤器原理如图2-24所示。



图2-24

# 3详细设计

在详细设计中重点讲述一些关键技术的实现方案，大多是前述总体实现方案的延伸，也包含了开发环境，运行平台相关的内容。

## 3.1项目构建与运行

### 3.1.1 Android 平台项目构建与运行

由于系统限制，Android APP 只能由 Java 代码启动。所以 Android 平台项目包含两个子项目：

1. Eclipse Java 项目（Android SDK, Apache ant 构建）

1. NDK C++ 项目（Android NDK 构建）

Eclipse Java 子项目仅仅作为一个游戏启动器，Java 代码启动整个游戏，加载 C++ 编译成的共享目标文件。而后 Java 代码使用 JNI 调用共享目标文件内的 C++ 代码，启动由 C++ 编写的游戏引擎和游戏开发人员编写的游戏。具体的启动过程，cocos2d-x 引擎已经做好了。所以我们无需关心这个子项目，只需要配置好 Android SDK，AVD 等开发环境即可。

NDK C++ 项目由 Android NDK 构建。引擎本身的构建过程控制，cocos2d-x 团队已经做好了；我们自己编写的游戏代码构建过程需要需要自己控制。在实际实践中，我们修改 `/proj.android/jni/Android.mk, /proj.android/jni/Application.mk` 以控制头文件搜索路径，处理子模块链接依赖。

NDK C++ 项目构建时，Android NDK 使用内置交叉编译器将游戏引擎代码和我们自己编写的游戏代码编译成目标平台的 ELF 共享目标文件。Apache ant 在进行最后的 APK 打包时会将 ELF 共享目标文件一起打包入 APK，这样 APK 在安装时可以安装到 Android 平台上，APP 运行时即可以加载这个 ELF 共享目标文件，执行 C++ 相关代码，启动游戏。

### 3.1.2 Linux 平台项目构建与运行

Linux 平台下我们使用 CMake 来控制项目构建过程。CMake 专为 C/C++ 项目设计，取代以往的 Makefile 或 Autotools 项目。CMake 也可以在 Windows 和 Mac 平台运行，但在我们的项目中仅来控制 Linux 平台下的项目构建过程 。

CMake 是一个依赖分析器。我们将源代码加入 CMake 控制，说明要使用的头文件搜索路径和模块依赖之后，CMake 分析/整合不同目录下各源文件的依赖项，组成树型依赖关系，生成 Makefile，再调用 make 命令开始项目构建过程。使用 Makefile 可以实现增量编译，这样当更改某文件时，其他所有不依赖它的源文件不需要重新编译，直接使用上次的编译结果，大大提高了编译速度。

CMake 同时是一个具备变量作用域，条件控制语句，函数，宏等编译语言设施的可编程系统。我们的游戏项目中也使用了这点来配置不同生成（DEBUG/RELEASE）下不同的构建行为。

虽然 GNU/Linux 平台推荐使用动态链接库。但我们的 Linux 项目构建过程最终生成单个 ELF 可执行文件，所有的外部库依赖都直接链接入 ELF 可执行文件中，以避免在运行过程中由于依赖库版本问题而引起未知错误。

### 3.1.3 Windows 平台项目构建与运行

Windows 下使用 Visual Studio 来构建项目。VS 中以 solutionn 和 proj 组成两级构建系统，以我们的游戏项目为例：

solution: touhou-game.sln

|-- touhou-game.vcxproj

|-- libcocos2d.vcxproj

|-- libbox2d.vcxproj

|-- ...

`-- librecast.vcxproj

整个游戏项目是一个 solution，整个 solution 包含 cocos2d-x 引擎项目和我们自己的游戏子项目，两个子项目各使用自己的构建方案（构建参数不同）。cocos2d-x 引擎使用自带的 libbox2d.vcxproj 等来控制自己的构建过程；我们自己游戏使用 touhou-game.vcxproj 来控制自己的构建过程。

然而 vcxproj 却与 VS 版本依赖，不同 VS 版本对 vcxproj 的处理方式不同，不利于团队协作。具体的解决方案见《Git 协作指南》。

## 3.2工具类实现

### 3.2.1 AudioController

音频控制器采用单例设计模式，使用时需要先获取单例对象。

如果有音频相关需要，则只需要引入音频控制器的头文件，而不需要再引入的头文件SimpleAudioEngine.h以及命名空间CocosDenshion。

音频控制器支持的操作如图3-1所示。

****

图3-1

### ClickParticle

点击粒子特效的实现方案如图3-2所示。

****

图3-2

### PlaceHolder



图3-3

PlaceHolder类提供的函数全为静态函数，可以直接通过类名调用。具体的创建过程如图3-3所示。

## 3.3层实现

### 3.3.1 SettingLayer

设置层可以被添加进不同的场景中，但是在不同的场景中，设置层能够提供的功能应该是有区别的。所以设置层在初始化之前需要按照所在场景，展示不同的界面布局。

在实现时，设置层接受当前场景标识符作为参数。在内部初始化时通过此参数排布不同的按钮。

### 3.3.2 ConversationLayer

对话层提供预设的角色立绘展示位置，提供一个对话框。每次创建时需要指定对话的唯一标识符。

要显示不同的对话，实质就是替换当前显示的文本。

对话层界面本身并无实现难点，关键在和Lua协同工作。

### 3.3.3 ConfirmButton

创建ConfirmButton时必须指定两个参数，一个是希望执行的操作func，另一个是针对本类希望执行的操作callBack。

两个参数类型均为std::function<void()>，只接受没有参数的可调用对象。

如果需要传递带参数的可调用对象，必须使用std::bind，将参数和函数指针以及执行对象绑定，封装为无参数调用对象。

当点击确认按钮后，将按照顺序执行func()以及callBack()。

## 3.4场景组织实现

游戏中的所有场景都指定一个唯一的字符串类型标识符。

### 3.4.1 NonGameplayScenesCache

场景缓存类存储场景时使用Cocos2dx封装的数据结构Map<>，

Map<std::string, Scene\*> 接受两个参数，

* 场景的唯一标识符，便于查找对应的函数指针
* 场景的指针

由于此数据结构经过Cocos2dx封装，任何添加进入集合的UI树节点都会自动增加引用计数，所以即使场景被移除，场景的内存也不会被释放。只有再清空场景缓存中保存的指针，对应的场景的引用计数才会变成0，内存才会被释放。

场景缓存类采用单例设计模式，提供三个函数。

getScene(sceneTag); 如果场景不在缓存中，返回 nullptr

addScene(sceneTag, Scene\*);如果场景已经在缓存中，什么都不做

removeAllScenes();在进入 Gameplay 场景时使用，用于删除所有缓存场景

### 3.4.2生命周期

每一个继承自Node的子节点都至少会含有以下6种可以重载的虚函数。

1. init()初始化节点时调用，只会调用一次
2. onEnter()当子节点被添加进UI树时调用，可以被多次调用，例如移除子节点并再添加进UI树就会再次调用
3. onEnterTransitionDidFinish()子节点完全进入场景，而且过渡结束时候调用，在onEnter()之后调用
4. onExit()子节点被移除时首先调用，可以被多次调用
5. onExitTransitionDidStart()子节点完全移除出场景的时候调用，onExit()结束后调用
6. cleanup() 子节点对象要被销毁时调用

需要注意的是，重载的函数必须首先执行父类的同名函数，否则子类函数将会覆盖父类函数的初始化过程。Node节点在每一个阶段都会进行最基础的初始化工作。重载的意义只是允许开发者在相应阶段附加游戏逻辑。

在项目中，在init()种初始化静态模块，在onEnter()中加载动态的数据，而在onExit()中卸除动态加载的数据。

### 3.4.3 内部类InnerClass

在具体实现某些界面场景时，仅仅使用Cocos2dx自带的UI控件是难以满足需求的。除了使用UIWidget中的控件方便功能实现外，还需要对一些界面元素进行一些封装。这些类只会在特定场景使用，定义在特定场景的类中，故称为内部类。

* EquipScene

SelectCharacterButton：加载进入装备场景后会创建数个角色头像按钮，要求点击角色按钮时能够正确加载界面元素。这就要求按钮能够记录角色信息。但是一般的按钮只能记录简单的标记信息，完全不足以满足需求。所以对按钮进行一下封装，此按钮可以保存一个角色对象信息副本，初始化选择角色按钮时，同时注册点击事件监听器。

ItemMenu：封装一个TableView控件，此控件能够便捷地实现元素在上下方向或左右方向的滑动切换功能。除了控件外，此类还要保存一个道具数组用于控件查询。玩家需要使用这个控件进行装备道具的选择与卸除。创建此类时需要指定两个回调函数，一个是希望进行的操作，就是将相应的道具装备给指定角色，另一个是回调函数，用来刷新界面。如果玩家选择取消，则直接将此类从场景中移除。

SpellCardMenu：类似ItemMenu。

* InventoryScene

characterMenu：显示当前出战的所有角色。创建时需要指定两个类型为 std::function<void(std::string)>的可调用对象作为参数。两个可调用对象接受的参数分别为选择的道具标签，选择的角色标签。如果选择返回，则直接将菜单移除。

* LocationSelectScene

SelectLocationMenuItem：此类封装了地点信息，以及设置了点击回调函数。

## 3.5 GameData实现

### 3.5.1接口

* 如何使用接口

GameData 的接口包括两部分：

1. 在 GameData.h 中定义的 GameData 类成员函数

2. 在 GameData 文件夹下定义的各 struct结构体

游戏逻辑调用 GameData 函数，GameData 函数返回对象，游戏逻辑再去使用返回的对象。这就是 GameData 接口的使用方式。

* 接口返回的对象中不包含与其他实体的联系

一个实体可能会与多个实体相联系，比如说一个角色可以装备多个道具，可以装备多个符卡，

是将这些关系直接放入实体做成vector<tag>，还是说实体就是实体，不包含与其他实体之间联系（即不包含那些 vector可变数组），想知道哪些实体与此实体联系时，再去问 GameData 呢？

我们选用了后者，接口返回的对象中不包含实体之间的关系。这么做是因为：我们想让实体类的定义更小，简单明了，一眼就能看懂这个实体是什么，不用费劲心思去理解实体定义中定义的那些 vector，思考与其他实体之间的联系。在真正需要处理这个实体与其他实体之间联系的时候，再去看 GameData 的接口，了解所需要的那一部分联系。

### 3.5.2实现

* 单例

做成单例保证游戏运行期间只有一个 GameData 对象，防止多对象同时操作造成混乱。

* 工作流程

GameData的工作流程如图3-4所示。

1. 将存档文件和设定文件解析成 DOM 树（init）
2. 游戏逻辑使用 GameData 的接口，GameData 代为操作 DOM 树，这时对存档所做的更改不实时同步到文件中（各个接口）
3. 若游戏逻辑明确表示要存档，GameData 将 DOM 树上做的更改同步至文件中（\*Save 接口）

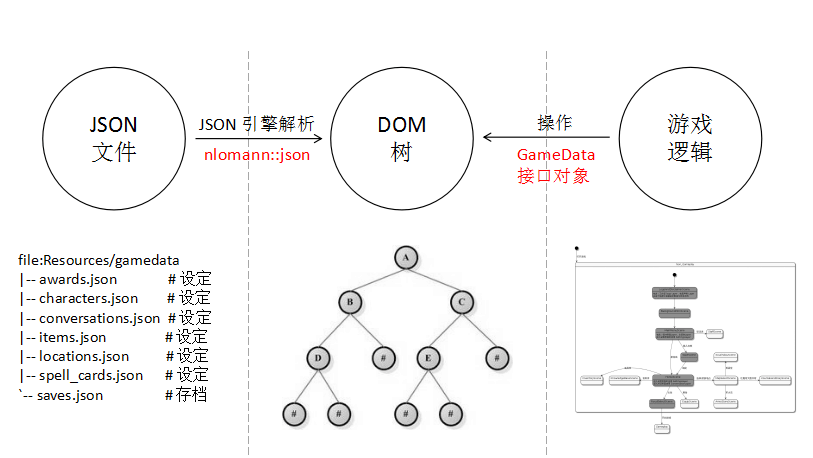


图3-4

* JSON文件组织

file:Resources/gamedata

|-- awards.json

|-- characters.json

|-- conversations.json

|-- items.json

|-- locations.json

|-- saves.json

|-- spell\_cards.json

|-- update.json

其中除 saves.json 是存档数据外，其他 json 文件都是只读的设定数据。

以下对 saves.json 的内部做一些解释：

1. saves.json 中有一些非存档性的玩家设置数据，像背景音乐音量大小，音效音量大小，对话速度大小，这些非存档性的设置数据也一并放入 saves.json 中了。这并没有什么特别的原因，只是因为这些数据量很少，就直接放在这里了。

2. saves.json 包含所有的存档，一个存档在其中是一个子节点，所有的存档都放入该文件中了，这样可以减少操纵多文件问题。

3. saves.json 中有一个 tag 为 0 的存档模板，tag 为 0 的存档作为新建存档的模板，是一个实现上的东西，不算做玩家的实际存档。当玩家需要新建存档时，以其作为模板生成一个编号不为 0 的模板。

4. saves.json 中只存档设定数据（道具，符卡，地点）的编号

* 添加数据

存档数据在 saves.json 中，设定数据在其他 json 文件中。

1. 若要更改存档模板就更改 saves.json 中 tag 为 0 的存档

2. 若要添加设定数据就更改其他 json文件

* 切换存档

如果要在游玩其他存档时切换到A存档，此时不需要再读取文件。其中A存档数据在内存中，通过GameData查询DOM树获取A存档数据，将A存档数据存放在存档缓存cachedSave中，就达到了切换存档的目的。

如果要支持在『在玩 A 存档时，能保存到 B 存档上』，必须要有一些特殊措施来保证。为了简单，我们仍采用缓存更改的方式。如下图所示：

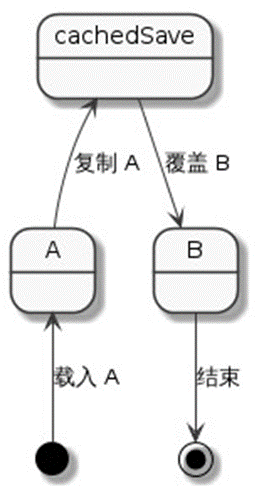


图3-5

将所做的更改不直接做在 A 上，而做在 cachedSave 上，当想保存到 B 时就用 cachedSave 覆盖 B，这样 A 就不会被更改了。在这里，A和B都处在内存中的DOM树中，所有的更改都是针对DOM树的。在存档覆盖结束后，由GameData再负责把最新更改的DOM树写入文件中。如图3-5所示。

## 3.6 Lua集成

### 3.6.1嵌入

将Lua语言嵌入宿主开发语言的流程如图3-6所示。



图3-6

### 3.6.2改造

cocos2d-x 提供纯 C++ 项目和纯 Lua 项目的直接支持，但并没有 C++/Lua 混合项目的直接支持。为了支持纯 Lua 项目，cocos2d-x 有以下工具：

* 从 C++ 生成 Lua 绑定的工具：在 cocos2d-x/tools/ 下有两个目录 bindings-generator, tolua。这是 cocos2d-x 团队做给他们自己用的从 cocos2d-x 引擎源源代码生成 Lua 绑定的工具。
* 与 Lua 交互的 C++ 工具类：在 cocos2d-x/cocos/scripting/lua-bindings/manual 目录下，有 LuaEngine, LuaStack, LuaValue 等工具类，还有 tolua\_fix.{h,cpp} 一些用来辅助 Lua 对 C++ 对象生存周期管理的函数。

我们需要改造 cocos2d-x 自带的各种设施，从

* 项目构建
* 生成/注册 Lua 绑定
* 调用 Lua 代码

三方面来完成我们的 Lua 集成与项目改造任务。

### 3.6.3从 C++ 代码生成 Lua 绑定

Lua 绑定说到底是一些供 Lua 调用的 C 代码。由于 Lua 函数调用规则与 C 调用规则（calling convention）不同，这些 Lua 绑定做从解析调用函数，再调用 C 代码的工作。

cocos2d-x/tools/tolua 的工作过程如下：

1. libclang 解析 C/C++ 代码
2. genbindings.py (python 代码) 抽取解析结果，根据
   1. 生成模板（tools/bindings-generator/targets/lua/templates）
   2. 用户自定义的生成规则（tools/tolua/.ini） 生成相应的 Lua 绑定（C 代码） 到 cocos2d-x-3.14.1/cocos/scripting/lua-bindings/auto

要从我们自己写的 C++ 代码生成 Lua 绑定，我们需要：

1. 配置 tools/tolua 的工作环境 tolua 及其依赖的 bindings-generator 需要 ndk-r9b 的工作环境。
2. 自定义生成规则 tools/tolua/<module>.ini。具体规则参考原目录下的各 ini 文件
3. 修改 tools/tolua/genbindings.py，使其使用我们自定义的规则
4. 指定 Lua 绑定的输出目录为 <proj>/Classes/LuaBindings
5. 将生成的 Lua 绑定文件 {h,cpp} 加入各平台编译系统
6. 修改 AppDelegate.cpp 使应用启动时注册 Lua 绑定到 Lua 运行时中

### 3.6.4对话层Lua实现



图3-7

touch/schedule/next/quit 框架仍然在 C++ 端，但具体的对话在 Lua 端一个 table 内存放。当要执行具体的第 i 条对话时，C++ 调用 Lua 端相应代码，并将 i 传递给 Lua，Lua 端函数根据这个 i 调用对应对话。

经过改进的对话层架构如图3-7所示。

## 3.7游戏场景实现

### 3.7.1制作物理阻挡

制作地图时单独分出来一个对象层physics，这个层专门用来绘制阻挡区域，里面的对象指定了地图中的阻挡区域。

cocos2dx封装的物理引擎，支持的形状有矩形，多边形，折线，圆形，但不支持椭圆。所以在制作地图时，只使用前四种图形绘制对象层中的对象。利用这些形状可以绘制更复杂的地形。我们随时可以在代码中操作这些刚体，例如可以随时移除一个静态刚体，达到地面塌陷的效果。

当瓦片地图被添加进地图层中，就可以创建静态刚体。

首先获取到对象层physice中的所有对象，遍历其中的对象。由于瓦片地图本质上就是个XML文件，记录了所有对象的起始位置，大小等信息。

由于在physics对象层中的对象都具有形状信息，不同的形状有不同的游戏逻辑含义，所以需要根据不同形状采用不同的处理方法。

如果对象类型是多边形，则需要沿某一方向读取到多边形的所有顶点，将这些顶点按照顺序放入数组中创建静态刚体，设置掩码使得该刚体和游戏中大部分的对象都可以发生碰撞。

如果对象类型是矩形，同多边形。

如果对象类型是折线，则需要记录构成折线的点，创建边缘刚体。由于折线刚体支持任何游戏对象从其下方穿过，所以需要设置接触掩码，供碰撞监听器调用以判断对象接触的方向。

### 3.7.2双主角切换

如果游戏场景中只有一个玩家操控的对象，那么所有游戏场景中的其他对象都可以放心地保存一个指向角色的指针。因为角色对象总是存在于游戏场景中，绝不会为空。

但是根据游戏设定，支持切换当前操纵的角色。这意味着同一时刻游戏场景中存在两个游戏角色对象，但是只有一个存在于UI树节点中。如果要访问角色对象，就要保存两个指针，还需要判断指针代表的对象是不是激活的对象。

为此引入“指针的指针”。



图3-8

在游戏场景中，除了切换角色的代码可以直接访问角色的指针外，其他对象都只使用CurPlayer指针。这样其他对象就可以不用判断对象是否激活。如图3-8所示。

每次切换角色时，只需要将当前角色从UI树中移除，将另外一个角色加进UI树。为了防止角色对象内存被释放，手动增加角色的引用计数一次。

### 3.7.3物理移动

玩家操控的角色需要符合一定的物理规律，就是速度是逐渐加快的，而停止移动后还应该有微小的惯性，所以最好使用物理引擎来控制角色移动。需要给玩家的刚体设置好质量，密度，回复力，摩擦力，以及禁止旋转等许许多多的物理属性，这些只能通过不断的微调来改善。

实现角色的跳跃很简单，只需要给角色一个向上的冲量。如果需要改变左右移动的速度，就需要引入加速度的概念，每一帧都给角色一个冲量。当水平速度达到设定的最大值时就停止加速。下面的代码每帧执行，判断当前的水平速率和最大速率，并设置冲量，实现平缓加速移动的效果。

if (velocity.x < MAX\_SPEED) {

impluse.x = std::min(MAX\_SPEED / ACCELERATE\_TIME \* dt, MAX\_SPEED - velocity.x);

}

角色不能无限制地在空中跳跃，所以需要引入一个跳跃次数的变量，每一次连续跳跃都要修改这个变量，一旦次数为0就禁止跳跃。直到玩家和地面接触，通过碰撞监听器，调用接触地面的回调函数，在回调函数中复位角色的各种状态，例如恢复可跳跃次数。

### 3.7.4切换区块

在首次初始化瓦片地图的时候，加载对象层中指定的区块信息。区块在瓦片地图中就是一个很大的对象层中的对象。

游戏场景记录当前角色所处的区块的坐标。区块是矩形的，所以游戏场景只需要记录区块的四角坐标。每一帧判断当前角色的坐标是否包含矩形中。

如果检测到角色坐标已经不在矩形中，那么就再次访问地图层对象，取得角色当前真正所在的区块信息，替换当前游戏场景记录的区块坐标。

接下来就需要移除上一个区块的所有内容，包括敌人，事件，装饰物等等对象。

成功移除后，就需要遍历所有的对象层，将坐标属于当前区块的对象按照类别加载进地图层。

最后就是要重设摄像机的跟随区域为当前矩形的大小。

这样就完成了区块的切换。

### 3.7.5视差节点

使用Cocos2dx提供的ParallaxNode类。

以视差背景节点作为说明，将视差动态背景层和静态背景层添加进视差背景节点中，设置跟随的速度。

视差动态背景移动缓慢，可以设置移动比率为10%，而静态背景完全跟随地图层移动，移动比率为100%。

由于视差节点完全脱离于地图层，不会跟随地图层一起移动。所以就需要在每一帧中根据地图层的偏移增量手动移动视差节点。

最重要的是在切换区块的时候，需要重设所有视差节点的位置。所有视差节点的位置都必须设为原点。因为切换区块后，地图层的偏移增量要归零。

这里需要注意地图层的偏移量和偏移增量是不同的概念。偏移量指地图层整体偏移坐标原点的大小，而偏移增量则是记录地图层在当前区块内偏移的大小。

但是玩家可能从一个区块边缘的任何位置切换进入另一个区块。如果只是单纯地将视差节点坐标归零，那么视差节点就会和屏幕坐标系重叠。如果角色从上方切换区块，角色落地前地图层又会产生一段向上的位移增量。这样视差节点层整体也会向上移动。这样的话当角色完全站立静止后，可能会看到背景下方的黑白区域，因为背景被过度移动。

所以在重设视差节点位置的时候不能简单的归零。而是重设为当前地图层的偏移增量。这样视差节点初始就有一定的偏移量，不会轻易的显示出空白区域。

### 3.7.6电梯

游戏中存在一种能够沿着固定轨迹定时移动的活动对象，称为电梯。电梯实质上就是一个可以承载角色或敌人的刚体。

电梯的实现需要解决两大问题，一个是设置运动轨迹，一个是让站在上方的对象保持移动。

电梯的运动轨迹在瓦片地图的对象层中绘制，呈现为一道连贯的折线。在游戏中读取运动轨迹，按照顺序读取折线顶点。读取全部顶点后，按照顺序创建Move动作。每两个相邻顶点创建一个Move动作，最后将所有的Move动作组合为一个Sequence动作。由于电梯还要能返回，还要再反向读一遍顶点，用相同的方法创建一个Sequence动作。最后创建一个无限执行的顺序动作，交替执行刚才创建的两个顺序动作。

由于物理引擎中没有静摩擦力，所以一个刚体站在另一个刚体上，如果脚下的刚体移动，上方的刚体并不会跟着同步移动，上方刚体最终会滑落。为了模拟同步移动的效果，每帧记录电梯移动的增量，并施加给电梯的“乘客”。

### 3.7.7碰撞监听器架构

碰撞监听器注册后，回调函数会自动接收到碰撞的信息。碰撞中包含两个互相碰撞的形状，通过形状可以获取到参与碰撞的两个UI树节点。但是碰撞中并没有包含碰撞的顺序信息。无法直接区分碰撞的的两个节点的实际类型和顺序。所以需要设计一种方法对碰撞进行分类：

获取节点A，获取节点B；

如果节点A和节点B存在，那么就分别获取到两节点的tag。

已经获取到了tag，接下来根据tag区分对象。

以角色为例，如果A和B中任何一个tag为角色的tag，那么，

1. 如果A的tag为角色的tag，创建节点指针ptrA，节点指针ptrB，将ptrA指向节点A，将ptrB指向节点B。那么ptr就指向了角色对象，接下来就可以判断ptrB到底指向哪些对象，并据此进行相应的逻辑处理。
2. 如果B的tag为角色的tag，创建节点指针ptrA，节点指针ptrB，将ptrA指向节点B，将ptrB指向节点A。那么ptr就指向了角色对象，接下来就可以判断ptrA到底指向哪些对象，并据此进行相应的逻辑处理。

其他游戏中的活动对象判断同理。

关键点在于映射。这样在之后就可以直接拿着ptrA和ptrB操纵两个碰撞的对象。

### 3.7.8自定义监听器使用

使用自定义事件分发与自定义监听器主要用来解耦。在不同模块中为了避免到处保存对象指针，只有使用对象监听器才可以将一个层次比较深的代码逻辑和一个表面的代码逻辑关联。在游戏场景中，直接注册了大部分的自定义事件监听器。

自定义事件监听器用来中转所有的自定义事件。尤其是从碰撞监听器和触摸监听器中发出来的自定义事件。这些事件都首先经由自定义事件监听器接受，将事件中包含的参数信息取出，再由这些监听器负责处理逻辑。这样就达到了解耦的目的。将逻辑的触发和逻辑的实现分离。

自定义监听器用来分发所有玩家的操作事件。包括跳跃，向左走，向右走，点击按钮等。

自定义监听器也用来实现道具和技能的逻辑处理。当玩家点击道具或技能时，相应的互动对象会将道具的tag放进事件中，然后发送自定义事件。由自定义事件监听器接受事件，取出tag，根据tag进行相应的逻辑处理。在此期间，还可以继续发送自定义事件。例如角色血条的增长就是通过事件机制获取变化量的。

自定义监听器用来终止游戏。每当能够推进游戏进度的事件发生后，对应的监听器接受事件，判断当前进度。如果已经达到了要求，则退出当前场景。

### 3.7.9着色器

着色器的主要用途已经在总体设计里讲解过。

如果要有效应用着色器，最好能够将着色器和Cocos2dx自带的动作系统结合起来使用，这样就能像使用正常的动作一样使用自定义着色器，例如指定着色器的持续时间。

为了将着色器和Cocos2dx的动作系统一同使用，创建继承自ActionInterval类的BlendAction，并重载必备的虚函数。

该动作类需要三个参数：持续时间，混合颜色，是否反转。

由于动作类实质上也是由调度器进行调度的，BlendAction类每次都被调度，接收到帧间隔时间。BlendAction根据此持续时间修改混合颜色的百分比。百分比初始为0，很快就可以达到100，执行着色器的对象颜色也就完全变成了目标颜色。如果允许反转，那么颜色百分比将从100逐渐减少为0。这样就实现了颜色逐渐恢复正常的效果。

### 3.7.10活动对象创建与组成

所有活动对象的创建过程都是类似的，如图3-9所示。



图3-9

所有的活动对象都继承自Node节点以便添加进入UI层。

本来Node节点本身就可以设置贴图以显示图片，但是为了支持多图层，以及将逻辑分离，给每一个活动对象增加了一个额外的图片精灵对象，图片精灵实际上只需要图片精灵播放帧动画。

添加发射器节点，说明活动对象可以弹幕系统，这是可选的。

属性值，包含活动对象的游戏逻辑信息，包括刚体的属性，自身的速度值，生命值等属性。

状态更新，因为角色的属性不会一直不变，例如有时候某些属性会随着时间慢慢恢复，这就需要启动调度器来更新这些属性值。

状态机主要是负责角色各种状态间的切换。使用状态及能够正确的切换角色的动画。

活动对象的组成如图3-10所示。



图3-10

### 3.7.11状态机实现

如前所述，状态机实现时需要两个类，一个是状态类，一个是管理状态的状态机类。状态机类记录两个状态。只有通过状态机才可以进行状态的切换。



图3-11



图3-12

对于一个角色来说，每一个状态能够跳转到的状态都是有限的，而且是有顺序的，状态机保证角色不会错误的切换状态。

其中站立和移动状态是最基本的状态。在这两个状态能够切换到大部分的状态。但是只有在站立状态下才可以切换到使用技能状态。

角色的基本状态切换如图3-12所示。

### 3.7.12人工智能

人工智能是一个很深的课题。在项目应用中，考虑到技术实力，只能从最简单的角度入手。

有限状态机经常被用来模拟智能体的行为。事实上状态机几乎是大部分游戏的首选。因为状态全部都是预设好的，智能体在决策时会消耗更少的运算量。而且良好组织的状态可以很好地规范智能体的行为。

智能体使用的状态机和角色使用的状态机没有什么不同。最大的不同，只是触发状态切换不同。角色的状态机由玩家的操控进行切换，而智能体的切换只能由智能体自己选择。



图3-13

智能体的状态切换往往比较简单。基本上只会每隔一段时间才会选择下一个动作。智能体往往存在一个用于决策的中心状态。在这个中心状态中，智能体判断当前的形势，并采取适当的行动：是移动并接近目标，还是直接发起攻击，还是说太久追踪不到敌人而切换到巡逻状态并休眠。如图3-13所示。

## 3.8弹幕系统实现

### 3.8.1系统概述

本弹幕系统主要负责敌人角色和玩家角色的攻击及伤害判定等任务，是面向游戏场景实际开发者服务的。

绚丽的弹幕配合特色的音乐是本游戏的主要玩点。大量的子弹会以一定的方式有规则地发射出来，在游戏场景上形成一定的几何形状。敌弹的攻击判定和自机的被击中判定比眼见的小，方便闪避。

本弹幕系统由发射器和子弹集合构成。角色可以绑定发射器，从而发射多个不同类型的弹幕。发射器负责创建并管理各类型的弹幕。子弹按照设定的参数进行自主地运动，直至销毁或回收。

### 3.8.2弹幕类型

* 固定弹

特点：固定弹的子弹运动轨迹是预先计算的，不会随目标位置变化而变化。

|  |  |
| --- | --- |
| 类型 | 特点 |
| 相对敌机固定 | 主要为全方位弹，以敌机为中心向着四面八方放出弹幕 |
| 相对屏幕固定 | 子弹运动相对固定，不随敌机和自机的变化而变化，可细分为版底弹和雨幕弹 |
| 反射弹 | 子弹在碰到版边的时候会反射回来，在一定频率下能回构成复杂的交叉弹幕 |
| 回旋弹 | 连续射击的子弹构成多条弹幕线，同时各弹幕线进行顺时针或逆时针的旋转，一般用来分割版面以配合其他弹幕使用 |

* 自机依存弹

特点：自机子弹发射时会根据目标的移动而不断调整发射情况。

|  |  |
| --- | --- |
| 类型 | 特点 |
| 奇数弹 | 以奇数股子弹为一组，保持不动中间的子弹必定会命中自机，需要移动规避。 |
| 偶数弹 | 以偶数的子弹数为一组，保持不动子弹就会从身边穿过，稍微移动不慎，就会被击中 |
| 分裂弹 | 子弹发射出一定距离后会分裂多股子弹 |
| 激光 | 射线方向大范围远距离激光 |
| 曲弹 | 子弹轨迹为曲线，切线目标为自机 |
| 跟踪弹 | 子弹行进时，会每隔一段时间定位自机位置，从而改变自身运动方向，从而达到跟踪的效果 |

* 随机弹

|  |  |
| --- | --- |
| 类型 | 特点 |
| 随机出现点 | 以敌机为中心，在一定范围内版面上出现弹幕 |
| 乱弹 | 发射数量和速度都不确定的子弹 |

* 组合弹

组合弹为以上各种类型弹幕中2种以上组合而成的，往往能形成比较繁杂的弹幕。

### 3.8.3弹幕实现

* 实体相关类

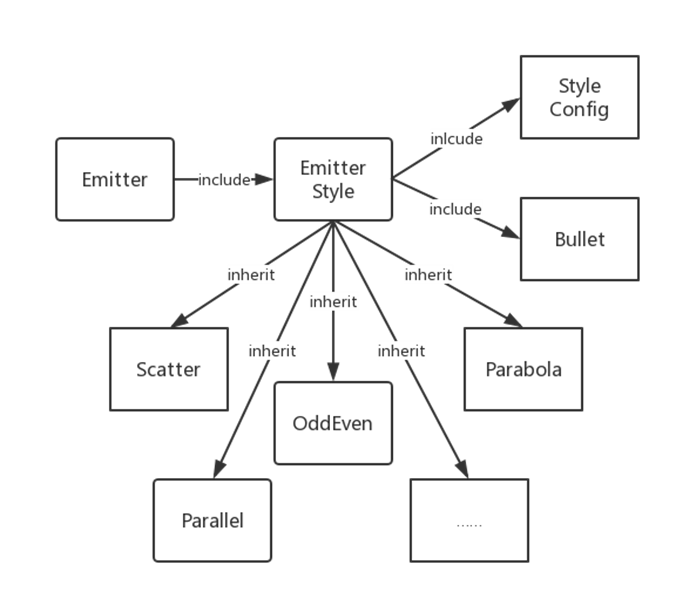


图3-14

* 工作流程

当Emitter创建完成并跟相关角色节点完成绑定后，就可根据StyleConfig参数进行不同类型的弹幕创建及初始化工作，调用Emitter的方法可以对弹幕进行管理。如图3-15所示。

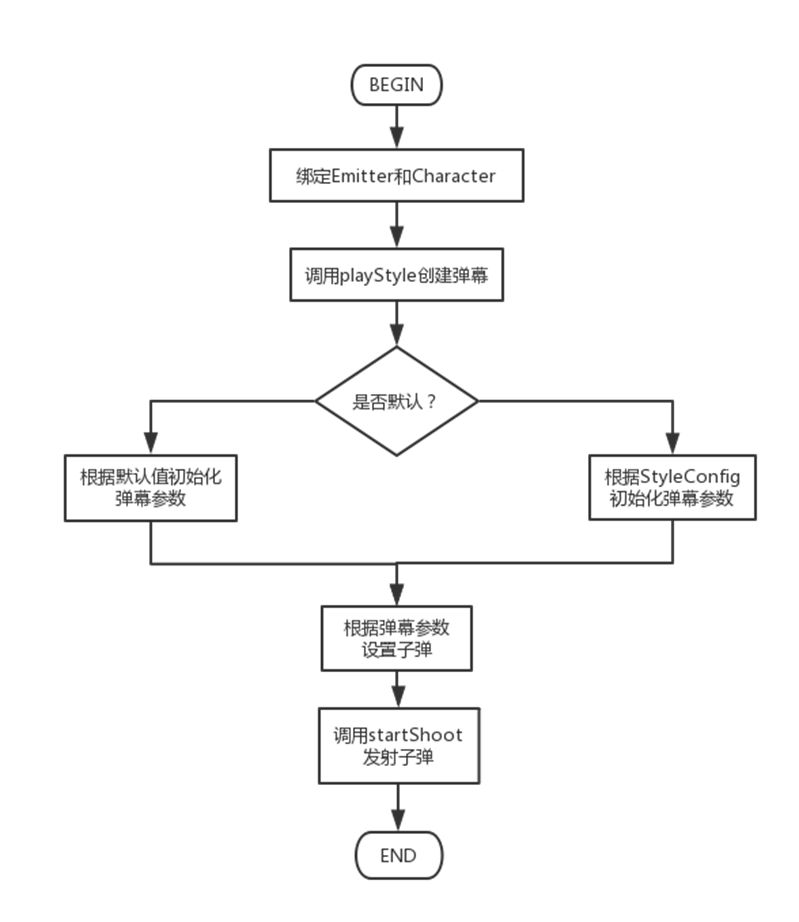


图3-15

### 3.8.4性能优化

大量子弹的不停绘制可能会影响游戏的性能，降低游戏的帧率，目前采用精灵表单批处理方案进行性能优化。

提前将所有子弹素材合成一张总图，利用plist文件配置相关位置信息。在游戏场景加载时，利用plist文件创建子弹的精灵表单，进行批处理操作。当需要绘制子弹时，只需从子弹精灵表单复制部分纹理即可，这样做可以减少单个子弹的绘制资源消耗，提升了游戏的稳定性。

### 3.8.5技术难点

* 子弹方向，距离计算

Cocos2d-x游戏引擎的旋转坐标系与数学上的极坐标系不一致，为了方便理解，在计算偏转角时会以旋转坐标系为主，而在在计算距离向量时会以极坐标为主。

Cocos2d-x没有封装数学函数，可使用的是C标准库的数学函数。Cocos2d-x基本上所有跟角度有关的方法的参数类型是角度制，而标准库的数学函数是参数类型都是弧度制。因此，计算时要经常进行角度和弧度之间的转换，容易出错，计算过程稍有不慎，会导致子弹飞行混乱，可以利用Cocos2d-x提供的角度弧度转换宏简化相关操作。

Cocos2d-x里的Action类的参数对外表现为坐标点，即距离向量。因此在计算距离时不光要计算模长还要计算方向。对于编写者来说，拥有良好几何数学知识是非常有必要的。计算一股子弹的距离向量时通常选择一个特殊的距离向量为基准向量，其他子弹以基准向量为基础进行一些坐标变换得到各自的距离向量。在计算固定弹的距离向量时，通常选择每股的第一颗子弹的距离向量为基准向量；在计算自机依存弹的距离向量时，选择角色和目标的中心连线为基准向量。

* 暂停机制

Cocos2d-x游戏引擎只提供了全局暂停的方法，但未提供局部暂停的方法。因此对于某种类型的弹幕，想要暂停弹幕发射出的子弹，需要另寻他法。

目前的实现方案是为每个弹幕设置一个子弹容器，将每次创建的子弹的节点指针保存在容器中。当需要对某种弹幕的子弹进行操作时，只需遍历弹幕相应的子弹容器即可获取单颗子弹的指针，从而对每个处在结点树上的子弹进行单独的操作。当子弹销毁时也要子弹容器里的相应指针删除。

* 贝塞尔曲线

贝塞尔曲线是计算机图形图像造型的基本工具，是图形造型运用得最多的基本线条之一。它通过控制曲线上的四个点（起始点、终止点以及两个相互分离的中间点）来创造、编辑图形。其中起重要作用的是位于曲线中央的控制线。这条线是虚拟的，中间与贝塞尔曲线交叉，两端是控制端点。移动两端的端点时贝塞尔曲线改变曲线的曲率（弯曲的程度）；移动中间点（也就是移动虚拟的控制线）时，贝塞尔曲线在起始点和终止点锁定的情况下做均匀移动。

贝塞尔曲线示意图如图3-16所示。

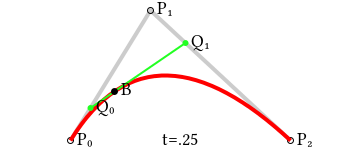
本次在开发过程中尝试利用贝塞尔生成了抛物线型弹幕。Cocos2d-x游戏引擎只封装了贝塞尔曲线的二次方形式，即。

图3-16

在Cocos2d-x中贝塞尔曲线运动的封装类为CCBezierTo和CCBezierBy。这两个Action都需要传入一个参数ccBezierConfig，这是一个结构体，这个结构体有三个字段：CCPoint endPosition（结束点）、CCPoint controlPoint\_1（控制点1）、CCPoint controlPoint\_2（控制点2）。两个控制点的会影响曲线的变化趋势。开始点就是精灵的当前位置，结束点和两个控制点通过ccBezierConfig这个结构体封装。