畅销书全面升级

著名移动开发专家关东升系统论述Cocos2d-Lua开发的百科全书

CocoChina、CSDN、51CTO、9Tech等四大专业社区联袂推荐

Cocos2d-x创始人王哲作序

清华游戏开发丛书

Cocos2d-x实战

Lua卷

关东升◎著

（第2版）

清华大学出版社

清华游戏开发丛书

Cocos2d-x实战

Lua卷

关东升◎著

（第2版）

清华大学出版社
北京

内 容 简 介

本书是介绍 Cocos2d-x Lua 游戏编程和开发技术的书籍。书中介绍了使用 Cocos2d-x Lua 中的核心类、瓦片地图、物理引擎、音乐音效、Cocos2d-x GUI 控件、Cocos2d-x 中的 3D 特性、数据持久化、网络通信、性能优化、多平台移植、程序代码管理、两大应用商店发布产品。

全书分为 6 篇：基础篇、进阶篇、数据与网络篇、优化篇、多平台移植篇和实战篇。

基础篇包括第 1～9 章，内容涵盖了 Lua 语言基础、Cocos2d-x Lua 介绍、环境搭建、标签、菜单、精灵、场景、层、动作、特效、动画和用户事件。

进阶篇包括第 10～15 章，内容涵盖了游戏音乐与音效、粒子系统、瓦片地图、物理引擎、Cocos2d-x GUI 控件和 Cocos2d-x 中的 3D 特性。

数据与网络篇包括第 16～18 章，内容涵盖了数据持久化、基于 HTTP 网络通信和 Node. js 与 WebSocket 网络通信。

优化篇包括第 19 章性能优化。

多平台移植篇包括第 20 和第 21 章，分别是移植到 Android 平台和移植到 iOS 平台。

实战篇包括第 22～25 章，分别是使用 Git 管理程序代码、项目实战：迷失航线手机游戏、发布到 Google play 应用商店和发布到苹果 App Store 应用商店。

图书在版编目(CIP)数据

Cocos2d-x 实战. Lua 卷/关东升著. —2 版. —北京：清华大学出版社，2017
(清华游戏开发丛书)
ISBN 978-7-302-45730-5

Ⅰ. ①C…　Ⅱ. ①关…　Ⅲ. ①移动电话机－游戏程序－程序设计 ②便携式计算机－游戏程序－程序设计　Ⅳ. ①TN929.53 ②TP368.32

中国版本图书馆 CIP 数据核字(2016)第 288774 号

责任编辑：盛东亮
封面设计：李召霞
责任校对：白　蕾
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：北京富博印刷有限公司
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：186mm×240mm　　**印　张**：29　　**字　　数**：639 千字
版　　次：2015 年 4 月第 1 版　2017 年 2 月第 2 版　　**印　　次**：2017 年 2 月第1 次印刷
印　　数：1～2500
定　　价：89.00 元

产品编号：070833-01

序

FOREWORD

欢迎来到 Cocos 的开发世界。

Cocos2-x 自发布第一个版本以来，历经几年的成长，到如今使用者已遍布全球，数不清的采用 Cocos 引擎开发的游戏横扫各个畅销榜单，我自己也成了其中很多游戏的忠实玩家。Cocos 引擎能一步一步走到今天，我很欣慰。感谢许多业界朋友的帮助，也感谢广大开发者的鼎力支持。

近两年，手机游戏行业在移动互联网世界的崛起是有目共睹的。行业格局在变化，Cocos2-x 不改初衷，开源免费始终如一，便捷高效步步提升，跨平台特性也日益完善。我们的引擎团队不断地努力改进，尽可能降低游戏开发的门槛，让更多有想法、有创意的朋友，不管是专业还是非专业出身的开发者，都能着手去实现。

关东升老师是国内著名的移动开发专家，精通多种开发技术，也有多年的开发经验，是一位不可多得的良师益友。这次关老师携手赵大羽先生倾力创作这套“Cocos2-x 实战”系列图书，共包括 5 册，分别是“C++卷”、“JS 卷”、“Lua 卷”和“工具卷”，其中“Lua 卷”与“JS 卷”更是填补了国内市场的空白。

这套图书系统地论述了 Cocos2-x 游戏开发理论与实践，涵盖 Cocos2-x 开发的几乎所有方面的知识领域。全部内容深入浅出，全面系统，对入门和进阶都大有裨益，非常值得阅读，我在这里郑重推荐给大家。

除了撰写图书，关老师还开设了超过 400 课时的 Cocos 引擎在线课程，我很敬佩他的专业精神，也非常感谢他一直以来对 Cocos2-x 的支持。关老师的书籍和在线课程在业内有相当高的人气，相信能为许多想要进入 Cocos 开发世界的朋友提供极大的帮助。

希望大家能从关老师的书籍和在线课程中学到更多，我也期待能有更多的开发者加入 Cocos2-x 大家庭。最后祝愿各位都能马到成功！

前言

PREFACE

“Cocos2d-x 实战”系列图书第 1 版出版后，得到业界一致好评，随着 Cocos2d-x 版本的变化，很多 API 有了较大变化，很多读者希望升级 Cocos2d-x 实战系列图书。

经过几个月努力，我们终于在 2016 年 7 月完成初稿，几个月来我们智捷 iOS 课堂团队夜以继日，我几乎推掉一切社交活动，推掉很多企业邀请讲课的机会，每天工作 12 小时，不敢有任何松懈和徘徊，只做一件事情——编写此书。书中每一个文字、每一幅图片、每一个实例都是精心制作。

这次“Cocos2d-x 实战”系列图书升级版包括如下 4 本书：

(1)《Cocos2d-x 实战：C++卷》(第 2 版)。

(2)《Cocos2d-x 实战：JS 卷——Cocos2d-JS 开发》(第 2 版)。

(3)《Cocos2d-x 实战：Lua 卷》(第 2 版)。

(4)《Cocos2d-x 实战：工具卷》(第 2 版)。

本书是 Cocos2d-x 游戏引擎 Lua 卷，就是使用 Cocos2d-x 的 Lua 语言 API。关于本系列图书的具体进展请读者关注智捷课堂官方网站 http://www.51work6.com。

关于源代码

为了更好地为广大读者提供服务，专门为本书建立了一个网站，具体网址是 www.51work6.com/book/cocos23.php，读者可以对书中内容发表评论，提出宝贵意见。

勘误与支持

我们在网站 www.51work6.com/book/cocos23.php 中建立了一个勘误专区，及时地把书中的问题、失误和纠正反馈给广大读者，您如果发现有什么问题，可以在网上留言，也可以发送电子邮件到：eorient@sina.com，我们会在第一时间回复您；也可以在新浪微博中与我们联系：@tony_关东升。

本书主要由关东升撰写，智捷课堂团队的赵大羽、赵志荣、关锦华参与了部分内容的编写。感谢赵大羽老师手绘了书中全部草图，并从专业的角度修改书中图片，力求更加真实完

美地奉献给广大读者。

在此感谢清华大学出版社的盛东亮编辑给我们提供了宝贵的意见。

最后感谢家人对我的宽容以及对我的关心和照顾，使我能抽出这么多时间，投入全部精力专心地编写此书。

由于时间仓促，书中难免存在不妥之处，请读者原谅，并提出宝贵意见。

关东升

2017年1月

于北京

目 录

CONTENTS

第二篇 进 阶 篇

第三篇 数据与网络篇

第四篇 优 化 篇

第五篇 多平台移植篇

第六篇 实 战 篇

第1章 准备开始

当你拿到这本书的时候，你会说："哇！这么厚！我应该怎么开始呢？"

本章我们不讨论技术，而是告诉大家本书的结构，书中的一些约定，以及如何使用本书的案例。

1.1 本书学习路线图

图 1-1 是本书学习路线图，也是本书的内容结构。

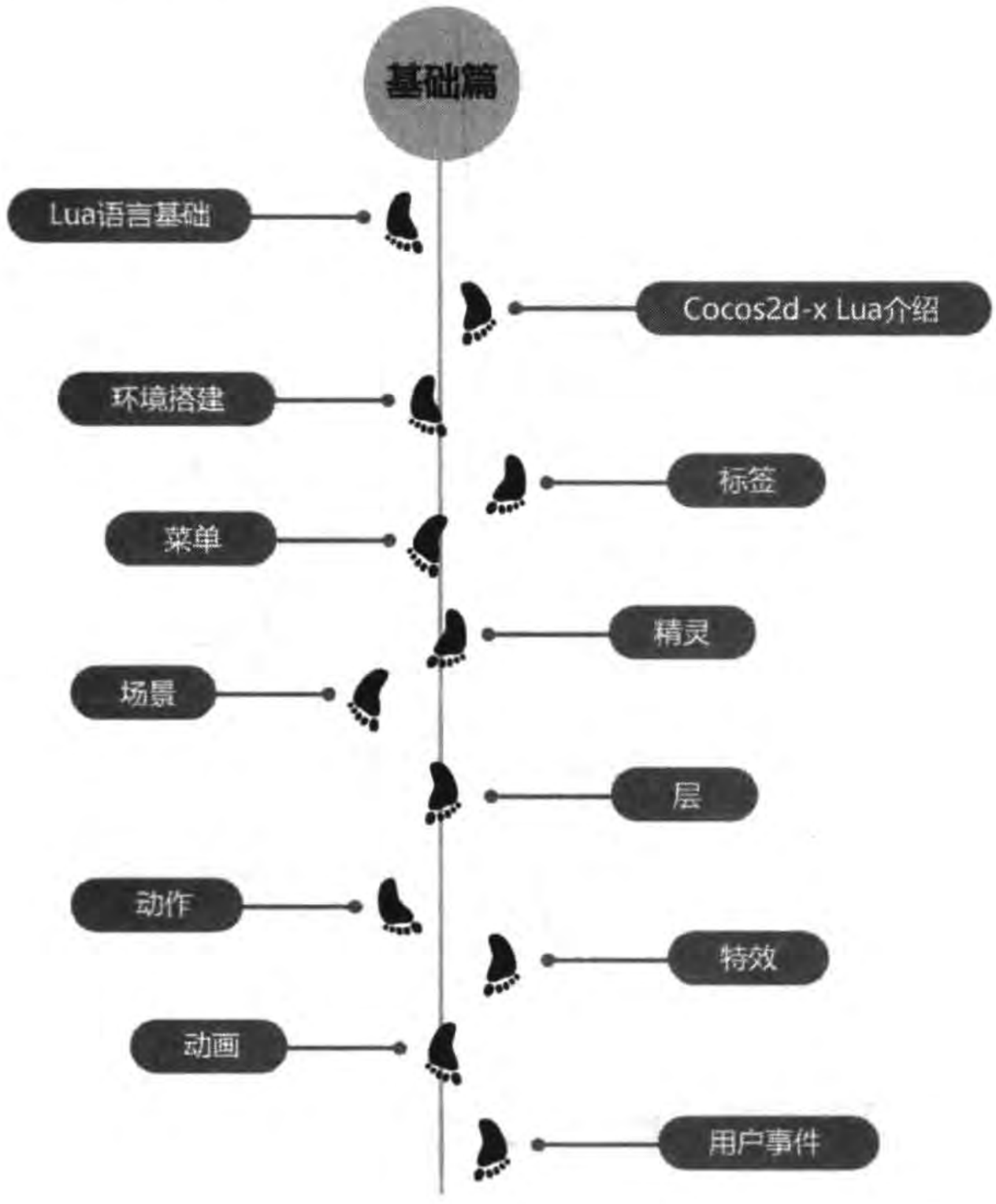


图 1-1 学习路线图

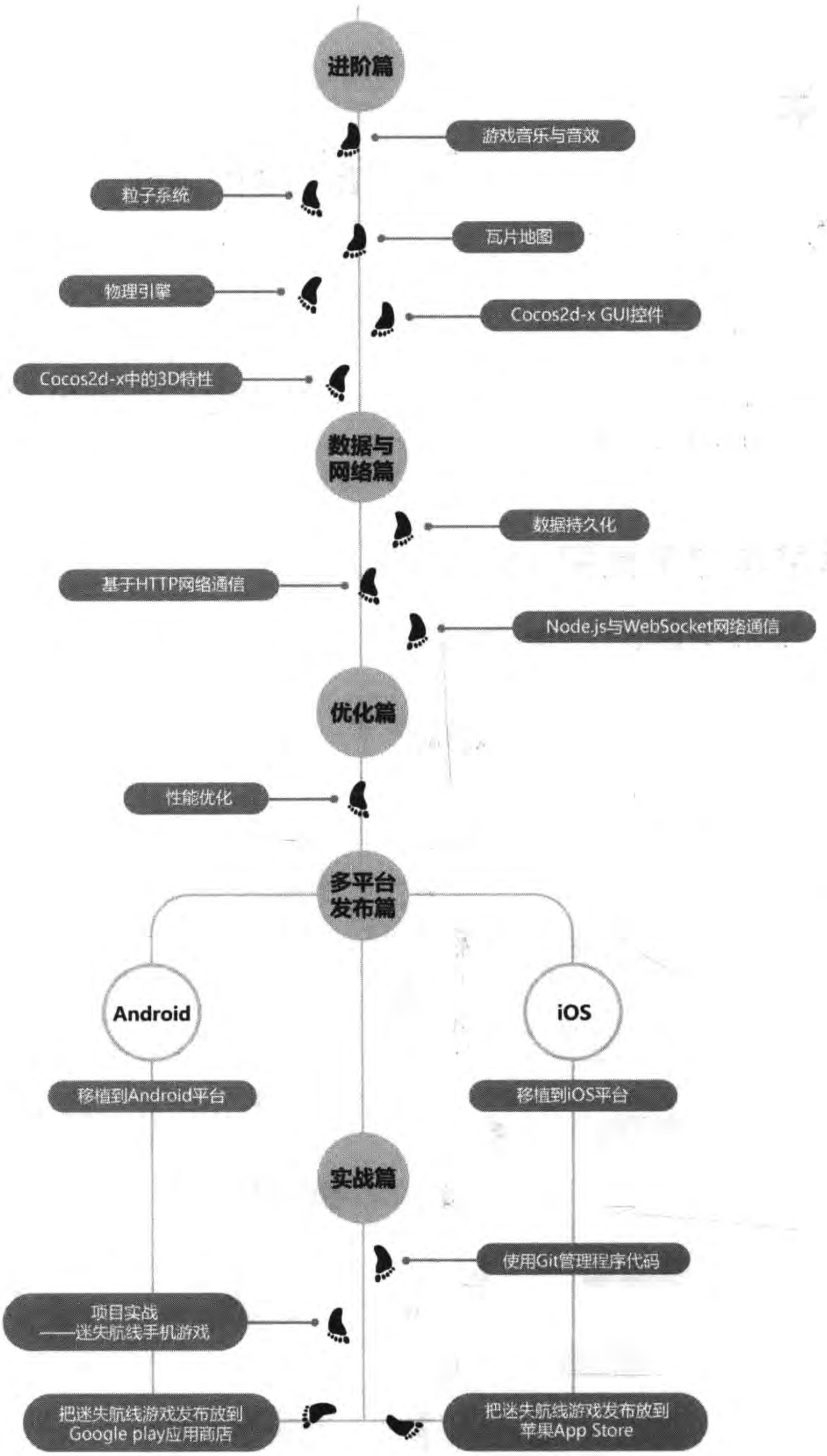

进阶篇
游戏音乐与音效
粒子系统
瓦片地图
物理引擎
Cocos2d-x GUI控件
Cocos2d-x中的3D特性
数据与网络篇
数据持久化
基于HTTP网络通信
Node.js与WebSocket网络通信
优化篇
性能优化
多平台发布篇
Android
iOS
移植到Android平台
移植到iOS平台
实战篇
使用Git管理程序代码
项目实战
——迷失航线手机游戏
把迷失航线游戏发布放到Google play应用商店
把迷失航线游戏发布放到苹果App Store

图 1-1 （续）

本书分为6篇：基础篇、进阶篇、数据与网络篇、优化篇、多平台移植篇和实战篇。为了降低广大读者学习成本和减少学习曲线，在平台发布之前都是使用Cocos Code IDE等工具开发。这样当完成学习Cocos2d-x Lua大部分知识点后，再介绍多平台移植篇，然后再介绍实战篇。

1.2　使用实例代码

作为一本介绍编程方面的书，本书有很多实例代码，下载本书代码并解压，会看到图1-2所示的目录结构。

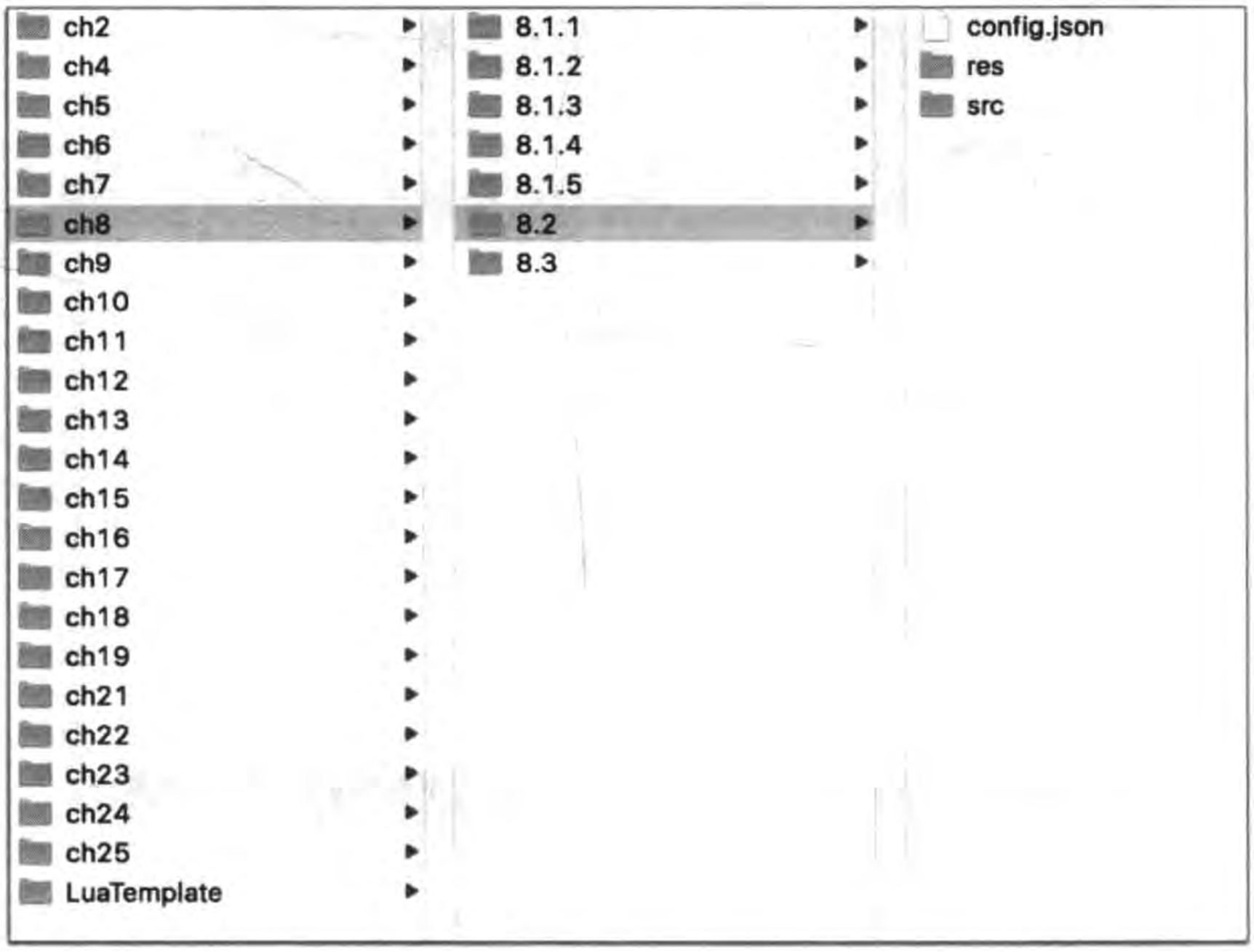


图1-2　实例代码目录结构

图中的ch8表示第8章代码，在ch目录下一般是各个小节的内容，例如8.2表示第8.2节，8.1.3表示第8.1.3节。在小节目录下一般是src和res等目录，这就是程序代码。由于每一个实例都是一个完整的Cocos2d-x Lua工程，整个由Cocos2d-x Lua模板生成的工程内容多达几百兆字节，这里给大家的只是其中核心的代码和资源文件。使用的时候，需要把代码中的目录内容复制到生成工程中。如图1-3所示，使用鼠标把8.2(表示第8.2节的实例)目录中的所有内容拖曳到Cocos2d-x Lua API工程目录上，然后松开鼠标，这时候会提示是否覆盖文件，选择覆盖这些文件。

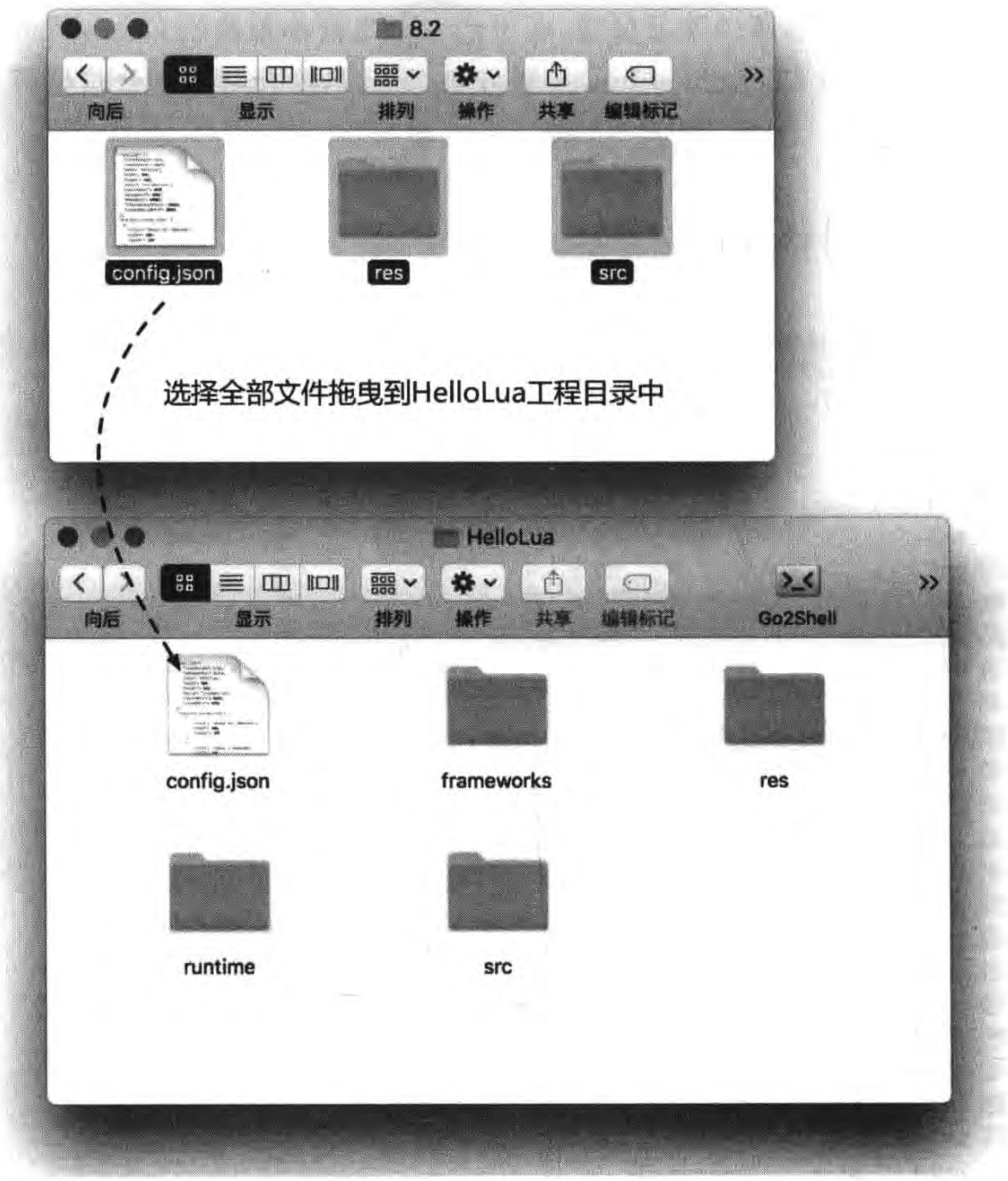


图 1-3　使用实例代码

此外，不同的开发环境还需要不同的配置，这些细节问题会在相应章节介绍。

第一篇　基　础　篇

本篇共 8 章，介绍 Cocos2d-x 游戏开发基础知识，其中包括：Lua 语言基础、环境搭建、字符串、标签、菜单、精灵、场景、层、动作、特效、动画和用户事件。

本篇各章如下：

第 2 章　Lua 语言基础

第 3 章　Cocos2d-x Lua API 与环境搭建

第 4 章　Hello Cocos2d-x

第 5 章　标签和菜单

第 6 章　精灵

第 7 章　场景与层

第 8 章　动作、特效和动画

第 9 章　用户事件

第 2 章

Lua 语言基础

Cocos2d-x Lua 游戏引擎采用的是 Lua 脚本语言。Lua 是葡萄牙语“月亮”的意思，1993 年由巴西里约热内卢天主教大学(Pontifical Catholic University of Rio de Janeiro)的一个研究小组开发的。设计目标是可扩展性，使其非常容易地在应用中加入 Lua 脚本。

Lua 是免费、小巧、快速而且容易移植的，由标准 C 编写而成，是最快、最高效的脚本语言之一。Lua 的内核小于 120KB，而 Python 的内核大约 860KB，Perl 的内核大约 1.1MB。对于游戏开发，Lua 是比较好的选择。

2.1 Lua 开发环境搭建

要想编写和运行 Lua 脚本，需要 Lua 编辑工具和 Lua 运行测试环境，下面分别加以介绍。

2.1.1 安装 LDT 工具

最简单的 Lua 编辑工具是使用一些文本编辑工具，但是它们往往缺少语法提示，有的语法关键字还没有高亮显示，最重要的是它们大部分不支持调试。考虑到易用性，推荐大家使用 Eclipse[①] 的 Lua Development Tools(LDT)插件，下载地址为 http://www.eclipse.org/koneki/ldt/，打开 LDT 网站，如图 2-1 所示，在网页 Installation 的 Standalone product 部分可以下载 Eclipse+LDT 版本，下载解压后就可以使用。

提示 如果你已经安装了 Eclipse 可以直接安装 LDT 插件，如图 2-1 所示，网页 Installation 的 Existing Eclipse installation 部分介绍了插件安装方式，这种安装方式需要注意 Eclipse 版本与 LDT 插件版本的兼容，这种方式比较复杂，笔者不推荐这种安装方式。

---

① Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台。就其本身而言，它只是一个框架和一组服务，用于通过插件组件构建开发环境。幸运的是，Eclipse 附带了一个标准的插件集，包括 Java 开发工具(Java Development Kit，JDK)。——引自百度百科 http://baike.baidu.com/subview/23576/9374802.htm

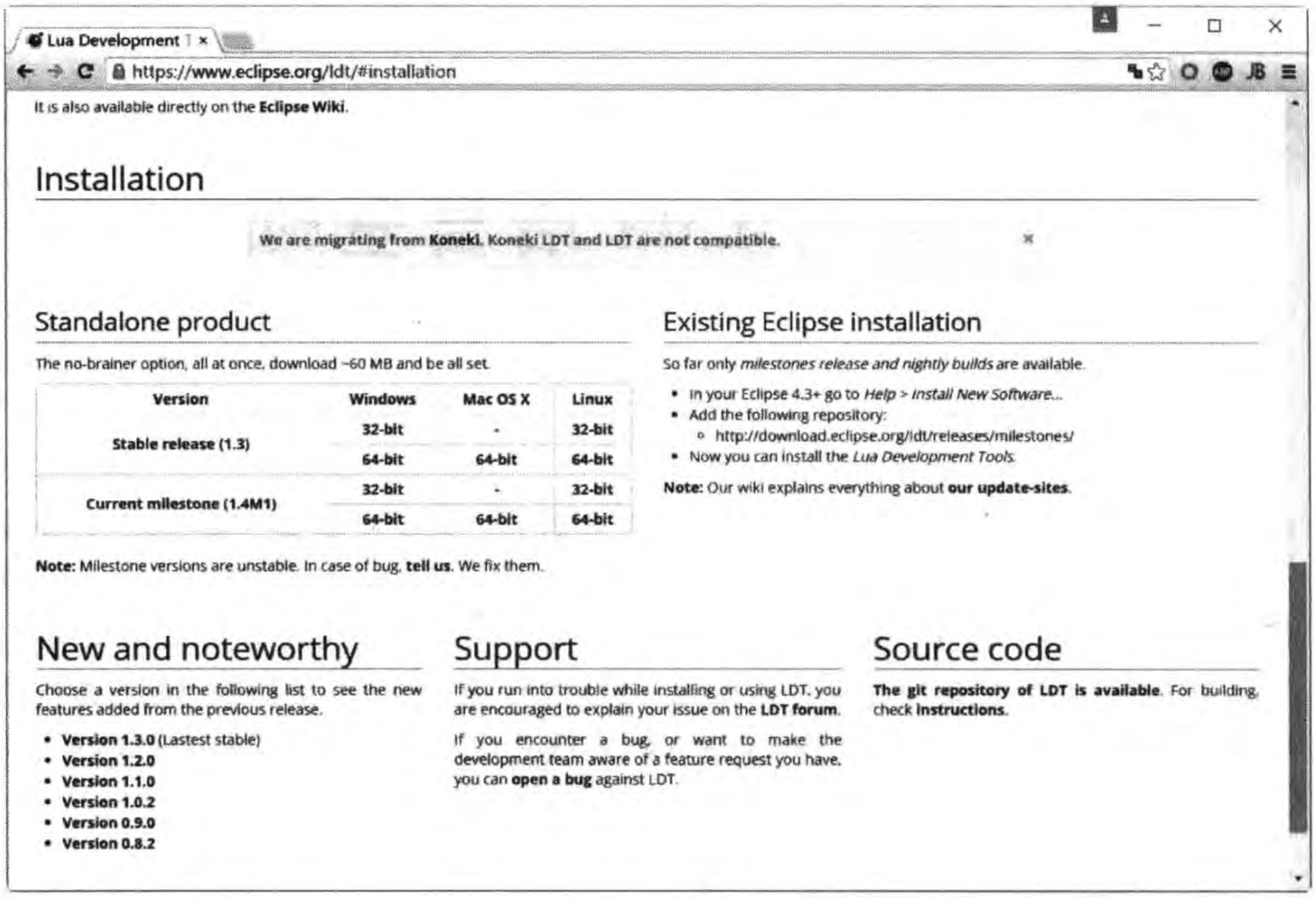


图 2-1　LDT 插件下载

Eclipse是基于Java开发的，要想运行Eclipse必须安装JRE(Java运行环境)或JDK(Java开发工具包)。下面介绍一下JDK下载和安装，图2-2展示了JDK 8下载界面，它的下载地址是http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html，其中有很多版本，注意选择对应的操作系统，以及32位还是64位安装的文件。

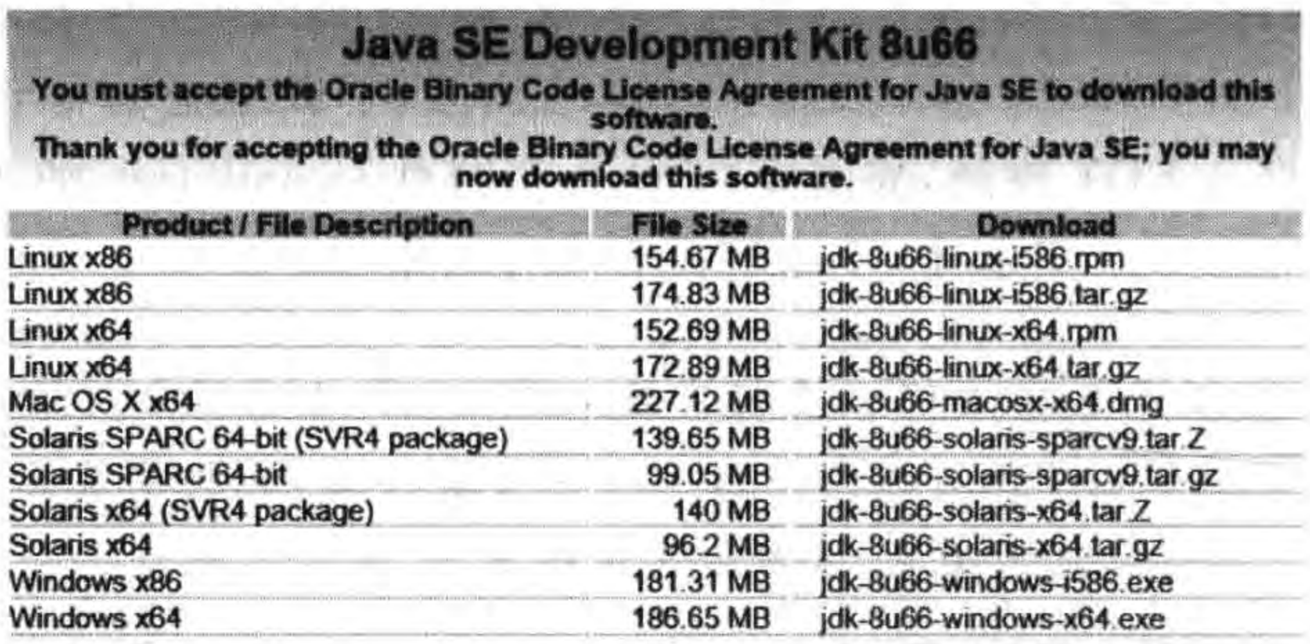


图 2-2　下载 JDK

默认安装完成之后，JDK需要设置系统环境变量，主要是设置JAVA_HOME环境变量。打开环境变量设置对话框，如图2-3所示，在用户变量(上半部分，只影响当前用户)或系统变

量(下半部分,影响所有用户)中添加环境变量,一般情况下在用户变量中设置环境变量。

图 2-3 环境变量设置对话框

在用户变量部分单击“新建”按钮,弹出的对话框如图 2-4 所示,变量名为 JAVA_HOME,变量值为 C:\Program Files\Java\jdk1.8.0_66,注意变量值的路径。

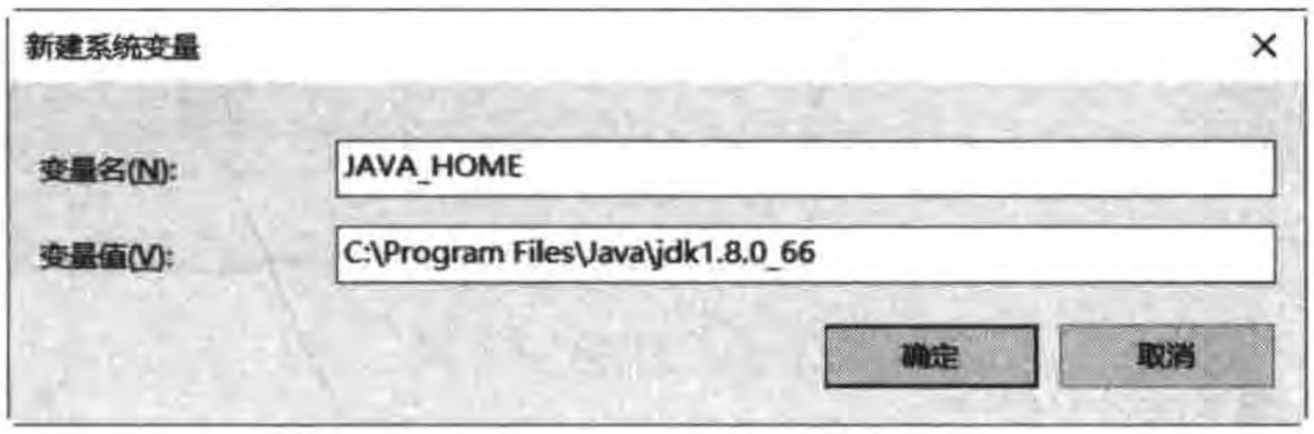


图 2-4 设置 JAVA_HOME

为了防止安装多个 JDK 版本对于环境的影响,可以在环境变量 Path 追加 C:\Program Files\Java\jdk1.8.0_66\bin 路径,如图 2-5 所示,在用户变量中找到 Path。双击打开 Path 修改对话框,如图 2-6 所示,追加 C:\Program Files\Java\jdk1.8.0_66\bin,注意 Path 之间用分号分隔。

JDK 安装好之后,在 LDT 解压目录下找到 Lua Development Tools 可执行文件,然后双击运行就可以了。首次运行 LDT 需要选择工作空间,工作空间是项目所在的目录,如图 2-7 显示了在 Workspace 字段中输入工作空间的路径,如果不想每次启动 LDT 时都选择工作空间,可以选中 Use this as the default and do not ask again 复选框。

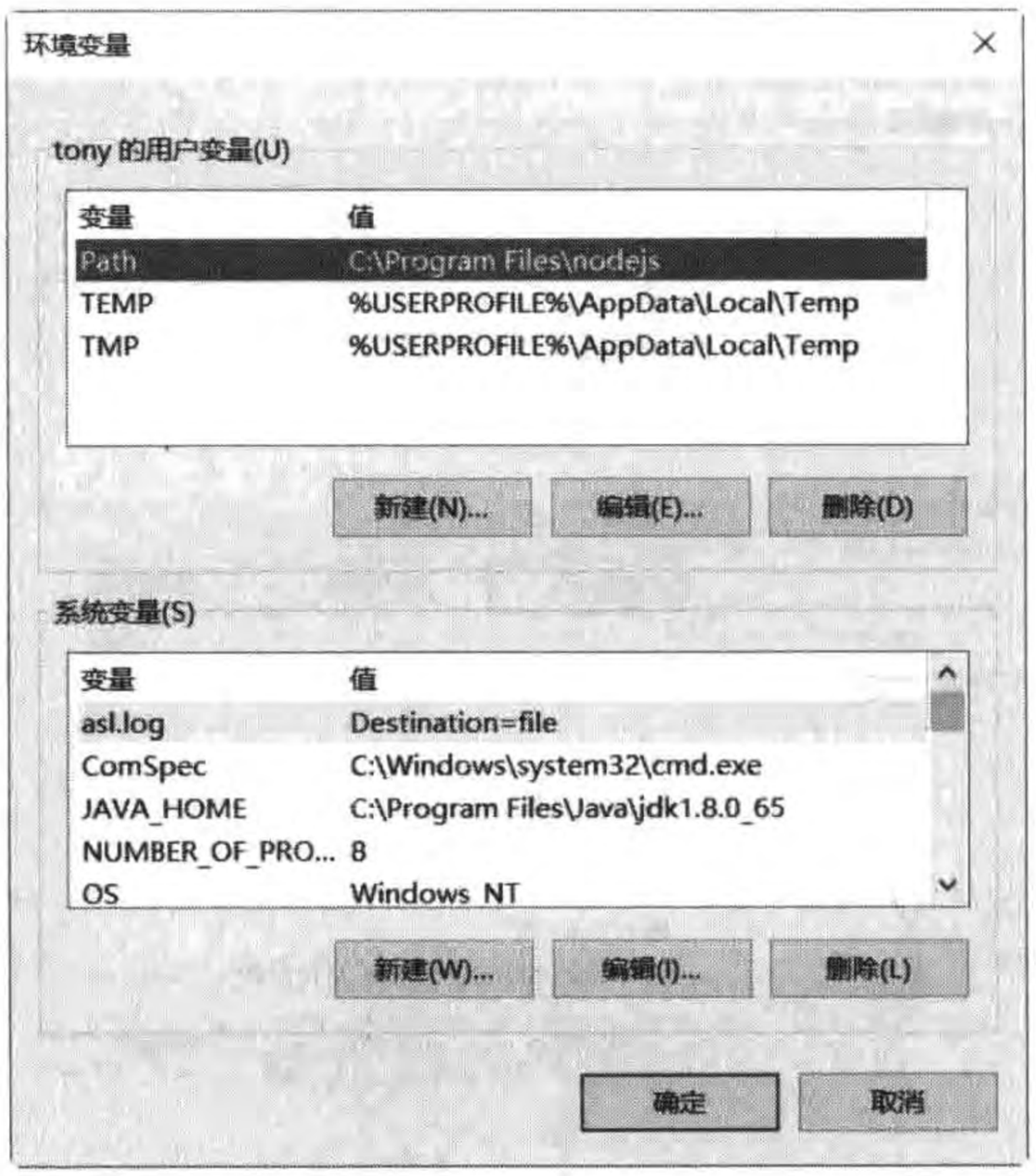


图 2-5　环境变量 Path 设置对话框

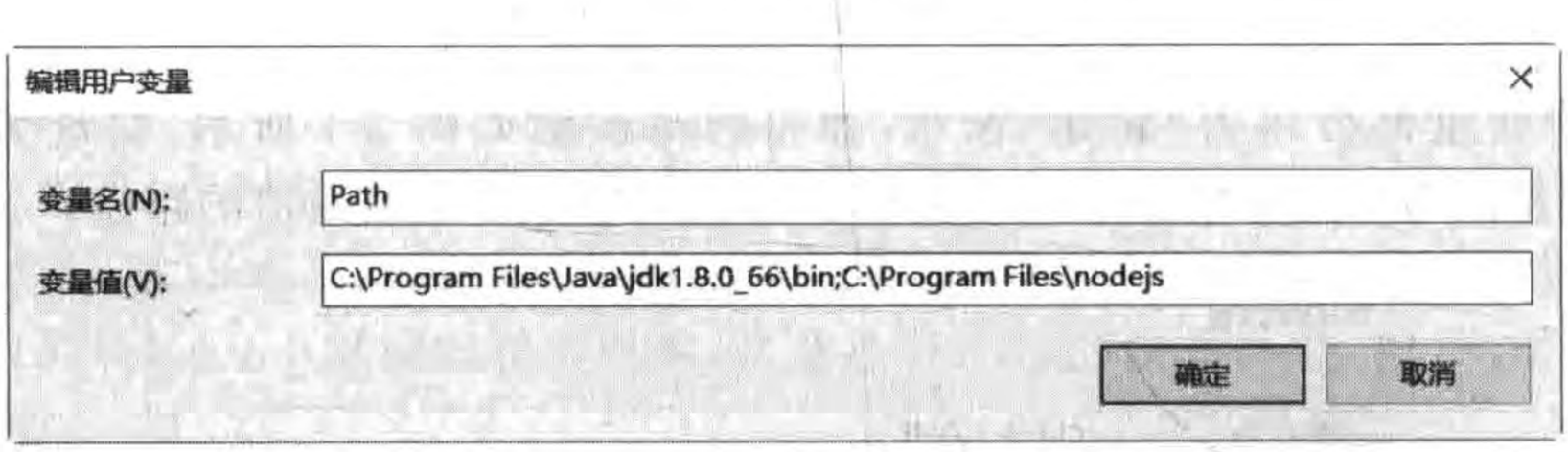


图 2-6　Path 修改对话框

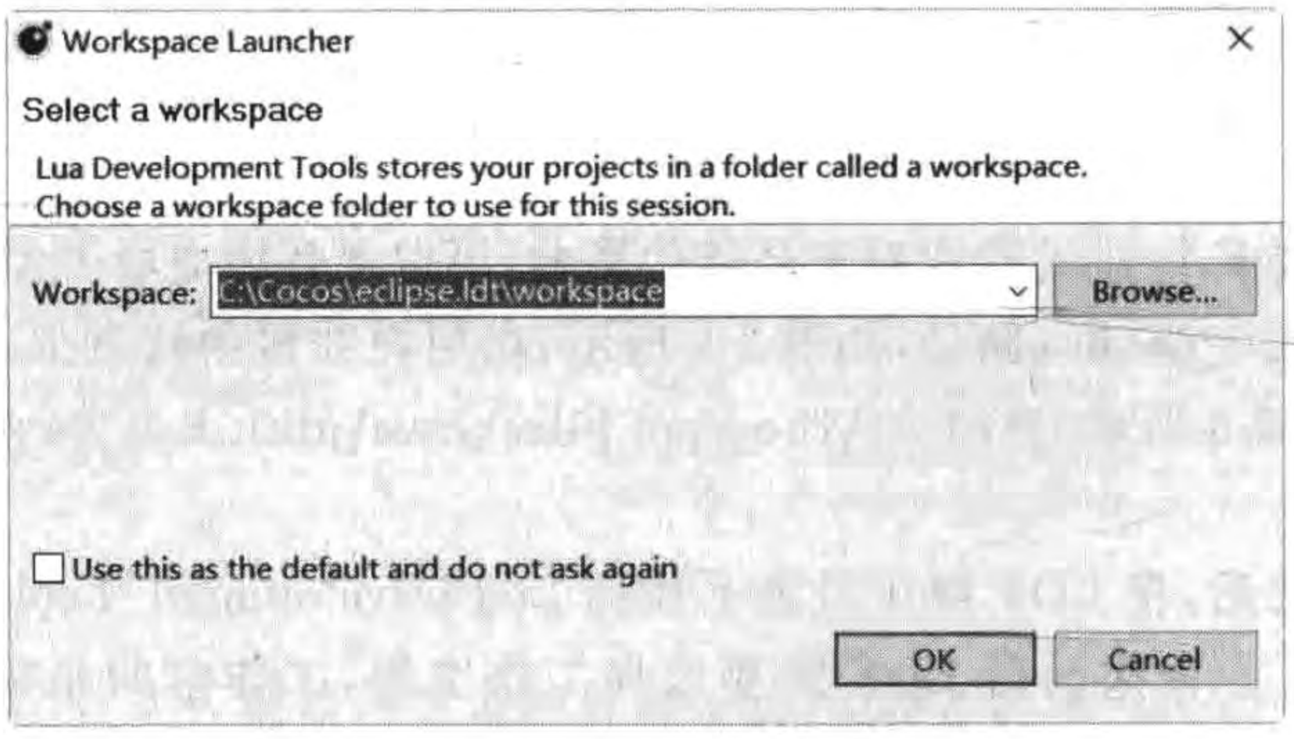


图 2-7　选择工作空间

2.1.2　HelloLua 实例测试

搭建好环境后需要测试，首先需要使用 LDT 工具创建工程，选择 File→New→Lua project，弹出 Create a Lua project 对话框，如图 2-8 所示，在 Project name 文本框输入工程名 HelloLua，其他项目保存默认值，然后单击 Finish 按钮就完成工程的创建了。

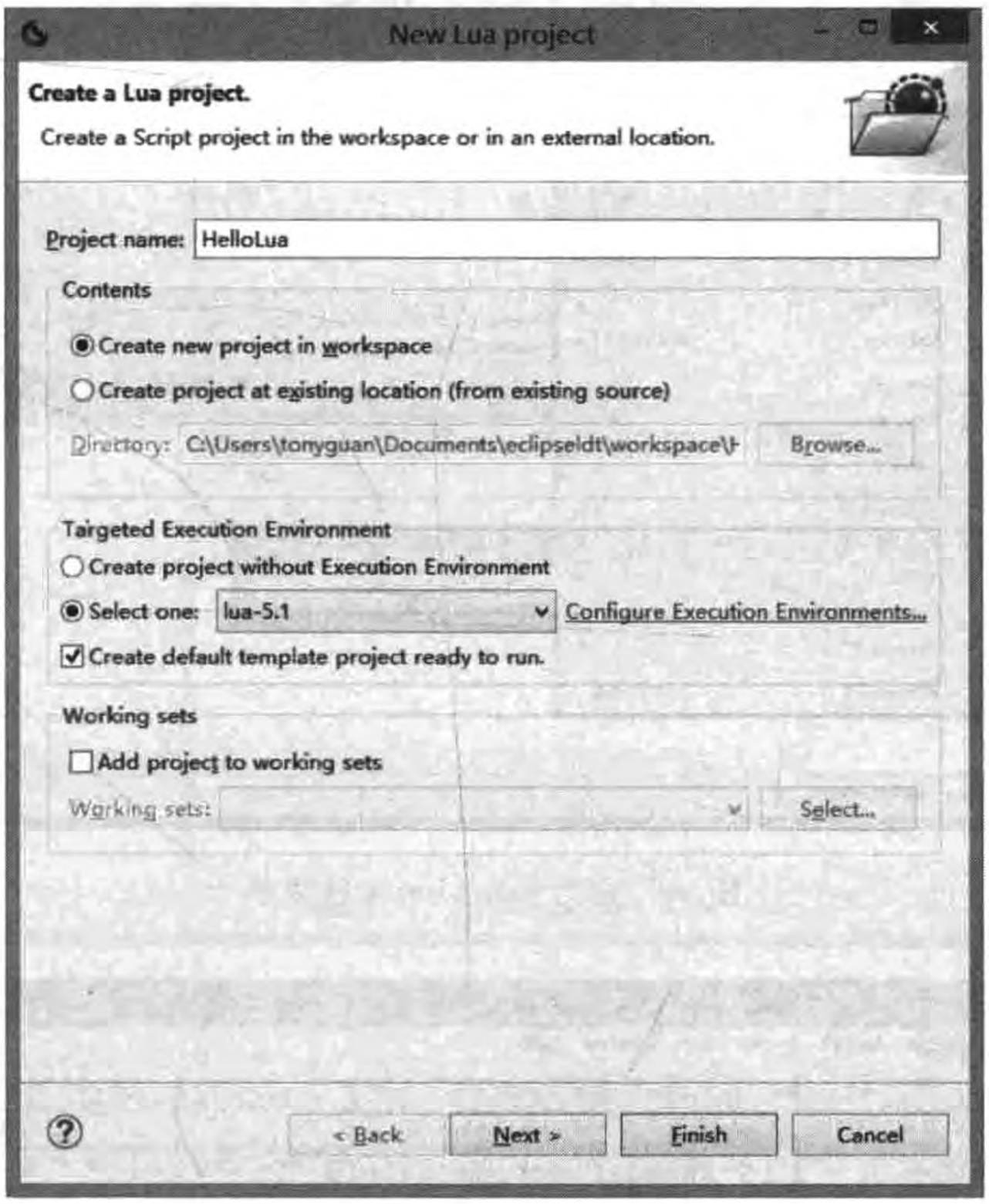


图 2-8　创建 Lua 工程

在 Eclipse 中打开 main.lua，并在编辑界面中输入如下代码：

```
local function main()
  local msg = 'Hello Lua!'
  print(msg)
end
main()
```

其中代码 local msg='Hello Lua!'是把字符串赋值给 msg 变量，print(msg)是将 msg 变量内容输出到控制台。如果要想运行 main.lua 文件，如图 2-9 所示，在左边的导航面板中选择 main.lua 文件，右击选择 Run As→Lua Application，然后会运行 main.lua 文件，将运行结果输出到控制台，如图 2-10 所示。

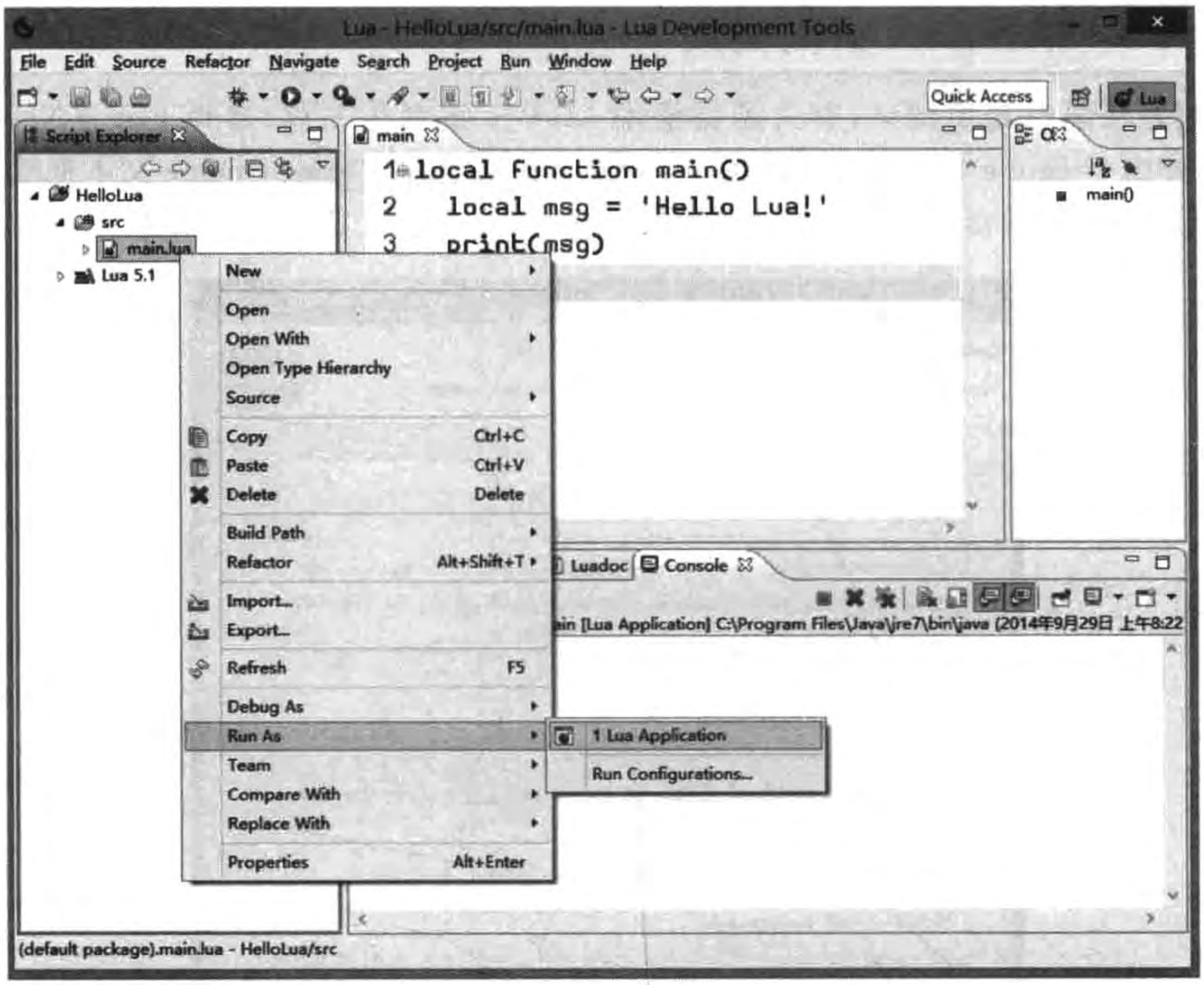


图 2-9　运行 main.lua 文件菜单

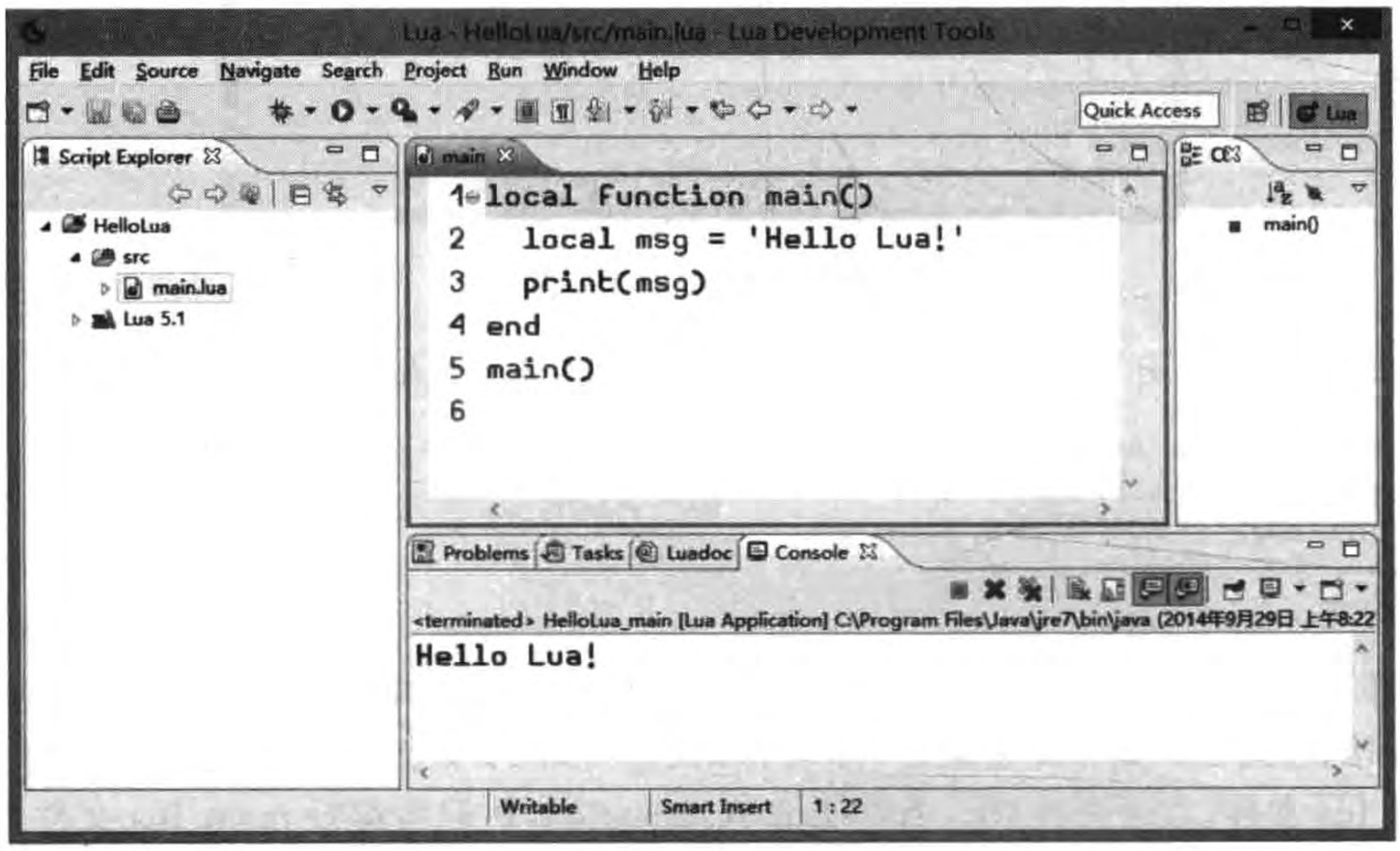


图 2-10　运行结果

如果想调试程序，可以设置断点，如图 2-11 所示，单击行号前面位置，设置断点。

图 2-11 设置断点

调试运行过程是选中 main.lua 文件，右击选择 Debug As→Lua Application，程序运行到第 3 行挂起，并进入调试视图，如图 2-12 所示。在调试视图中可以查看程序运行的堆栈、变量、断点、计算表达式和单步执行程序等操作。图 2-12 中的①区域是 Debug 窗口，这里可以查看程序调用过程，这个过程就是运行堆栈。②区域是变量窗口，这里可以查看当前变量的值，此外，在该区域单击相应的标签还可以打开断点和表达式等窗口。③区域是代码窗口，可以在该窗口中进行单步运行等调试操作。

调试工具栏的主要按钮如图 2-13 所示。

提示 如果编写的文件中有中文，而且想在 Windows、Mac OS X 和 Linux 等平台能够正常显示，就需要设置文件的字符集为 UTF-8。如图 2-14 所示，在 LDT 中选择 Window→Preferences，在 Preferences 对话框中选择 General→Content Types，然后选择对话框右边的 Text，之后在 Default encoding 中输入 UTF-8，最后单击后面的 Update 按钮更新字符集为 UTF-8。设置完成单击 OK 按钮关闭对话框。

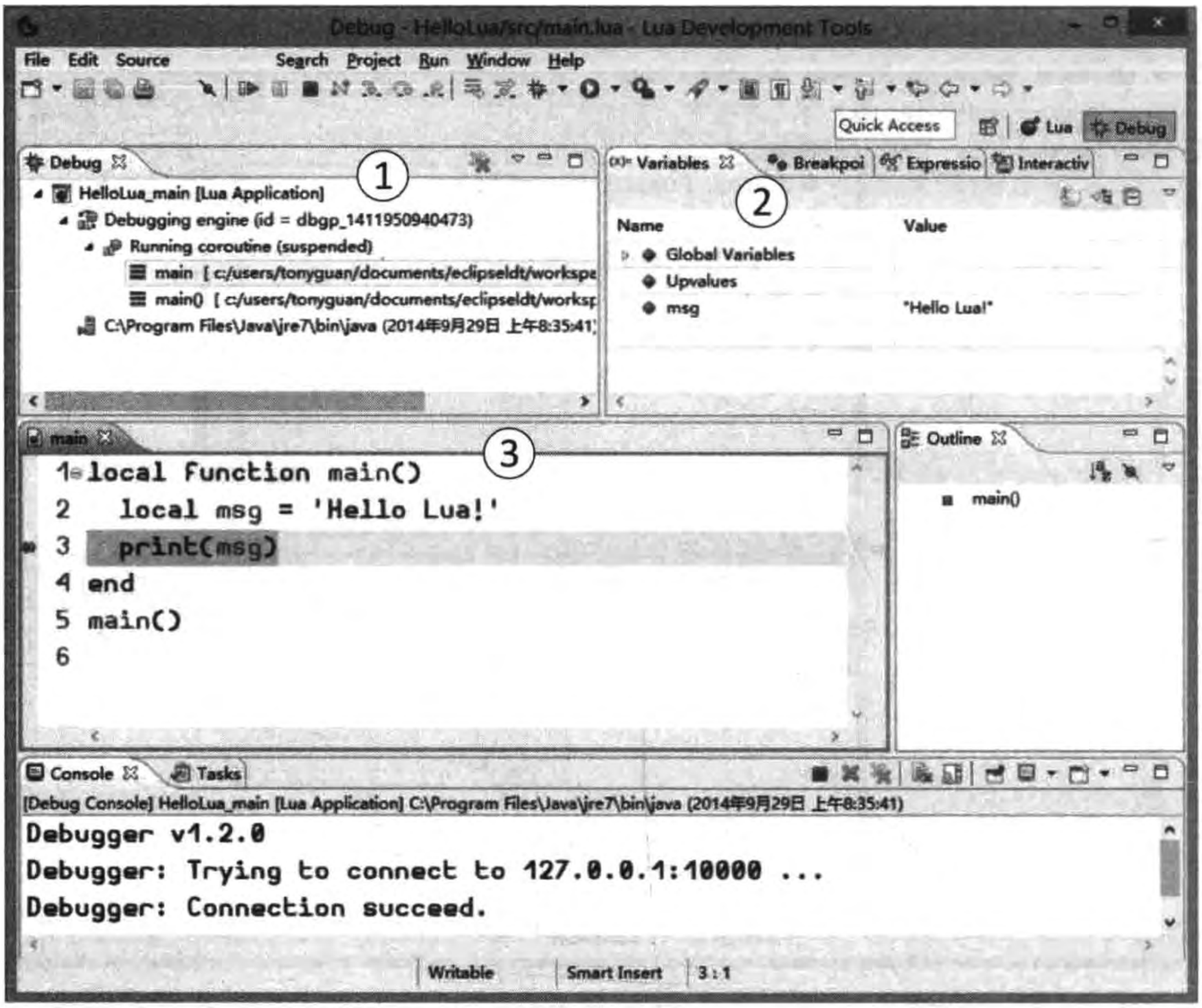


图 2-12　运行到断点挂起

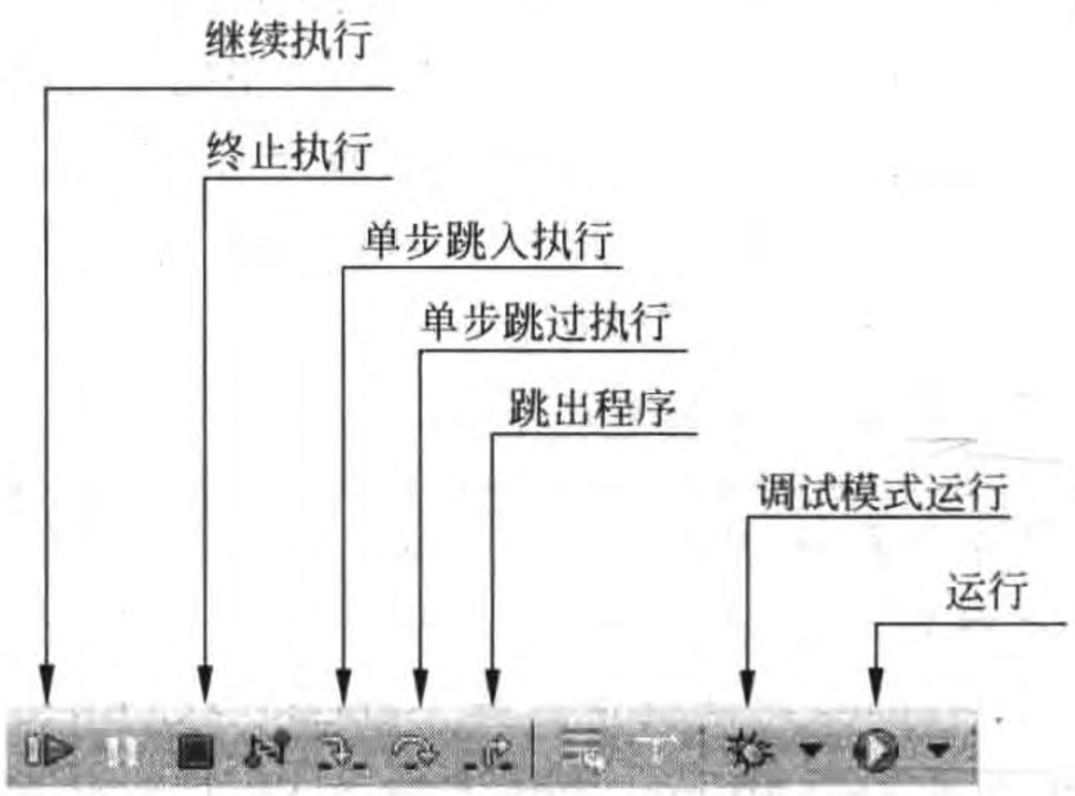


图 2-13　调试工具栏按钮

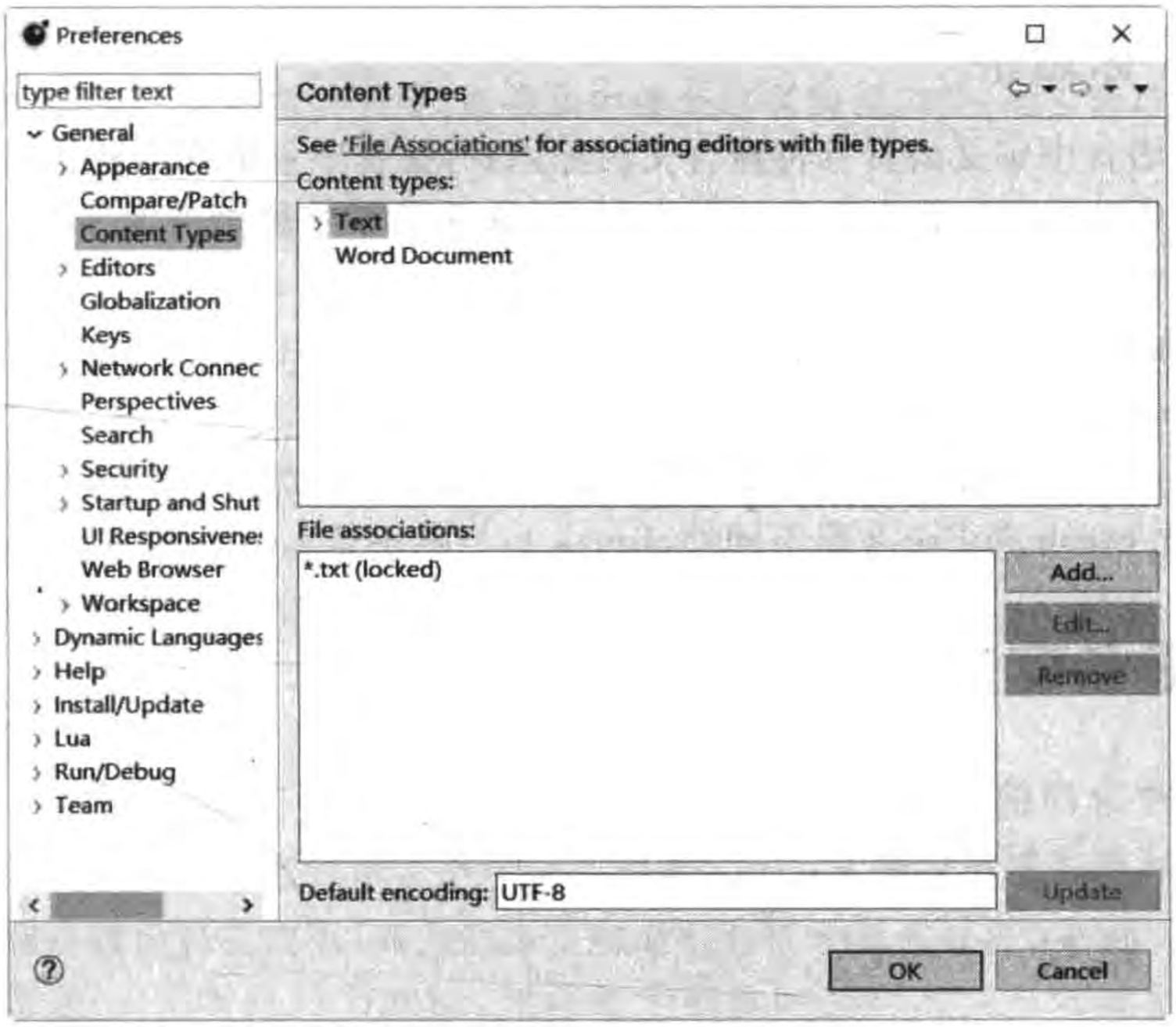


图 2-14 设置文件的字符集为 UTF-8

2.2 标识符和保留字

任何一种计算机语言都离不开标识符和保留字，下面详细介绍 Lua 标识符和关键字。

2.2.1 标识符

标识符就是给变量、函数和对象等指定的名字。构成标识符的字符有一定的规范，Lua 语言中标识符的命名规则为：

(1) 区分大小写，Myname 与 myname 是两个不同的标识符。

(2) 标识符首字符可以以下画线(_)、美元符($)或者字母开始，不能是数字。

(3) 标识符中其他字符可以是由下画线(_)、美元符($)、字母或数字组成的。

例如，identifier、userName、User_Name、_sys_val、身高、$change 等为合法的标识符，而 2mail 和 room# 为非法的标识符。其中，使用中文“身高”命名的变量是合法的。

注意 Lua 中的字母是采用 ASCII 编码而非 Java 等采用的 Unicode 编码[①]，所以中文、日文、韩文等字符不能作为 Lua 的字母使用。

---

① Unicode 是统一编码制，是国际上通用的 16 位编码制，它包含了亚洲文字编码，如中文、日文、韩文等字符。

2.2.2 保留字

保留字是语言中定义的具有特殊含义的标识符,保留字不能作为标识符使用。Lua 语言中定义了一些具有专门意义和用途的保留字,这些保留字称为关键字,下面列出了 Lua 语言中的关键字:

and、break、do、else、elseif、end、false、for、function、if、in、local、nil、not、or、repeat、return、then、true、until 和 while。

上述很多保留字目前没有必要知道其含义,但是要记住的是,在 Lua 中关键字大小写是敏感的,因此 break 和 Break 是不同的,Break 也当然不是 Lua 的保留字。

2.3 常量和变量

在 2.1 节中介绍使用 Lua 编写一个 HelloLua 的小程序,其中就用到变量。常量和变量是构成表达式的重要组成部分。

2.3.1 常量

在 C 语言等其他语言中,常量只能在声明时初始化,不能在使用过程中改变它的值。在 C 和 C++中可以通过在标识符的前面加上关键字 const 实现。但是在 Lua 中的常量并没有这样限制,例如:

```
NUM = 100
```

事实上 Lua 中的常量与变量没有区别,只是命名规范要求常量标识符必须大写。

2.3.2 变量

在 Lua 中声明变量,可以在标识符的前面使用 local 关键字修饰或不使用 local 关键字修饰,示例代码如下:

```
local scoreForStudent = 0.0
scoreForStudent = 0.0
```

上面语句声明 scoreForStudent 变量,并且初始化为 0.0,变量的类型是数值类型。使用 local 关键字修饰的变量是局部变量,局部变量的作用范围是它所在的函数。没有 local 关键字修饰的变量是全局变量。

变量赋值之前值为 nil,下面声明的两个变量值均为 nil:

```
local scoreForStudent
scoreForStudent
```

2.3.3 命名规范

在使用常量和变量时,命名要规范,这样程序可读性好。

1. 常量名

基本数据类型的常量名为全大写，如果是由多个单词构成，可以用下划线隔开，例如：

```
local YEAR = 60
local WEEK_OF_MONTH = 3
```

2. 变量名

变量的命名有几个风格，主要以清楚易懂为主。有些程序员为了方便，使用一些单个字母来作为变量名称，如 j 和 i 等，这会造成日后程序维护的困难，命名变量时发生同名的情况也会增加。单个字母变量一般只用于循环变量，因为它们的作用只是在循环体内。

在过去，计算机语言对变量名称的长度会有所限制，但现在计算机语言已无这种限制，因此我们鼓励用清楚的名称来表明变量作用，通常会以小写字母作为开始，并在每个单字开始时第一个字母使用大写，例如：

```
local maximumNumberOfLoginAttempts = 10
local currentLoginAttempt = 0
```

像这样的命名可以让人一眼就看出这个变量的作用。

除了常量和变量命名要规范，其他的语言对象也要讲求命名规范。对于对象等类型，它的命名规范通常是，大写字母作为开始，并在每个单字开始时第一个字母使用大写，例如 HelloWorldApp。函数名，往往由多个单词合成，第一个单词通常为动词，通常会以小写字母作为开始，并在每个单字开始时第一个字母使用大写，例如 balanceAccount 和 isButtonPressed。

2.4　注释

Lua 程序有两类注释：单行注释(--)和多行注释(--[[...--]])。这些注释方法与 C、C++和 Java 都完全不同。

1. 单行注释

单行注释可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本，然而，它也可以用来注释掉连续多行的代码段。以下是几种风格注释的例子：

```
if x > 1 then
    -- 注释 1
else
    return false -- 注释 2
end

-- if x > 1 then
-- else
--   return false
-- end
```

2. 多行注释

一般用于连续多行的注释文本，但它也可以对单行进行注释。以下是几种风格注释的

例子：

```
--[[
if x > 1 {
    //注释 1
} else {
    return false;          //注释 2
}
--]]

--[[
if x > 1 {
} else {
    return false;
}
--]]
```

Lua 多行注释有一个其他语言没有的优点，就是可以嵌套。上述示例的最后一种情况就是实现了多行注释嵌套。

在程序代码中对容易引起误解的代码进行注释是必要的，但应避免对已清晰表达信息的代码进行注释。需要注意的是，频繁的注释有时反映出代码的低质量。当被迫要加注释的时候，考虑一下重写代码使其更清晰。

2.5 Lua 数据类型

数据类型在任何的计算机语言中都比较重要。Lua 是动态类型类似于 JavaScript，变量类型的确定是在赋值的时候，而不是在声明的时候。

2.5.1 数据类型

Lua 数据类型可以分为数值类型(number)、布尔类型(boolean)、字符串类型(string)、自定义类型(userdata)、函数类型(function)、线程(thread)、表类型(table)和空值(nil)。

1. 数值类型

数值类型包括整数和浮点数，整数可以是十进制和十六进制，由一串数字序列组成。如果是 0x 开始则表示它为一个十六进制数。

浮点数必须包含一个数字、一个小数点或 e(或 E)，如 3.1415、−3.1E12、0.1e12 和 2E−12。

2. 布尔类型

布尔类型有两种值：true 和 false。

3. 字符串类型

字符串是若干封装在双括号(")或单括号(')内的字符，如：

```
"fish"
'fish'
```

```
"5467"
"a line"
```

4. 自定义类型

在 Lua 中，通过自定义类型可以将任意的 C 语言数据存储到 Lua 变量中，实现代码交互。

5. 函数类型

在 Lua 中函数可以作为一种数据类型使用，函数可以赋值给一个变量，也可以作为参数传递给其他的函数。

6. 线程类型

表示一个线程，可以同时执行多个线程，每个线程拥有自己独立的栈、局部变量和指令指针。

7. 表类型

表类型实现了一组关联数组类型。表类型是使用{}表示的，示例如下：

```
local point = {x = 10,y = 20}
print(point["x"])                    -- 采用下标方式访问
print(point.y)                       -- 采用字典方式访问
```

8. nil

nil 是 Lua 中特殊的类型，表示没有任何有效值的情况。变量没有被赋值以前默认值是 nil，如果变量被赋值为 nil，则 Lua 的垃圾收集器会删除该变量，释放它所占用的内存。

2.5.2　type 函数

type 函数可以返回定变量或数值的类型。示例代码如下：

```
print(type("Hello Lua"))
print(type(100))
print(type(100.0))
print(type(true))
print(type(print))
print(type(nil))
print(type({x = 10, y = 20}))
```

输出结果如下：

```
string
number
number
boolean
function
nil
table
```

2.5.3　数据类型转换

Lua 提供了类型转换函数，这些转换包括转换成字符串和转换成数字等。

1. 转换成字符串

可以将布尔类型和数值类型转换为字符串类型，由 tostring()函数实现转换。示例代码如下：

```
local found = false
print(tostring(found))                          --输出"false"

local num1 = 10
local num2 = 10.0
local num3 = 10.01
print(tostring(num1))                           --输出 "10"
print(tostring(num2))                           --输出 "10"
print(tostring(num3))                           --输出 "10.01"

local tb = tonumber({x = 10, y = 20})
print(tostring(tb))                             --输出 nil                    ①
```

第①行代码是试图将 table 类型转换为字符串类型，结果是 nil。

2. 转换成数字

把非数字的原始值转换成数字的函数 tonumber()。示例代码如下：

```
local num = tonumber("10")                      --返回十进制数 10
local num = tonumber("AF", 16)                  --返回十六进制数 175          ①
local num = tonumber("0xA")                     --返回 10                     ②
local num = tonumber("56.9")                    --返回 56.9
local num = tonumber("0102")                    --返回 102

local num = tonumber("12345red")                --返回 nil                    ③
local num = tonumber("red")                     --返回 nil                    ④
local num = tonumber(true)                      --返回 nil                    ⑤
local num = tonumber({x = 10, y = 20})          --返回 nil                    ⑥
```

上述第①行代码将字符串"AF"转换为十六进制数值，tonumber 函数有两个参数，第二个参数是基数，基数表示数值的进制。

第②行代码将字符串"0xA"转换为十进制数值，tonumber 函数默认基数是十进制。

第③～⑥行代码返回 nil 表示无法转换成有效的数字，从第⑤行代码可见布尔类型不能转换为数值类型，从第⑥行代码可见 table 类型也不能转换为数值类型。

2.6 运算符

运算符是进行科学计算的标识符，常量、变量和运算符组成表达式，表达式是组成程序的基本部分。

2.6.1 算术运算符

Lua 中的算术运算符用来进行整型和浮点型数据的算术运算，Lua 中的算术运算包括＋、－、＊、/、％和^，这些运算符对整型和浮点型数据都有效。二元运算符说明如表 2-1 所示。

表 2-1 二元算术运算

运 算 符	名 称	说 明	示 例
+	加	求 a 加 b 的和	a+b
-	减	求 a 减 b 的差	a-b
*	乘	求 a 乘以 b 的积	a * b
/	除	求 a 除以 b 的商	a/b
%	取余	求 a 除以 b 的余数	a%b
^	幂	求 a 的 b 次幂	a^b

下面看一个算数运算符示例：

```
--声明一个整型变量
local intResult = 1 + 2
print(intResult)

intResult = intResult - 1
print(intResult)

intResult = intResult * 2
print(intResult)

intResult = intResult / 2
print(intResult)

intResult = intResult + 8
intResult = intResult % 7
print(intResult)

print("--------")
--声明一个浮点型变量
local doubleResult = 10.6
print(doubleResult)

doubleResult = doubleResult - 1
print(doubleResult)

doubleResult = doubleResult * 2
print(doubleResult)

doubleResult = doubleResult / 2
print(doubleResult)

doubleResult = doubleResult + 8
doubleResult = doubleResult % 7
print(doubleResult)
```

输出结果如下：

```
3
2
4
```

```
2
3
--------
10.6
9.6
19.2
9.6
3.6
```

上述例子中分别对整型和浮点型数据进行算数运算,具体语句不再解释。

2.6.2 关系运算符

关系运算是比较两个表达式大小关系的运算,它的结果是真(true)或假(false),即布尔型数据。如果表达式成立则结果为 true,否则为 false。关系运算符有 6 种:>、<、>=、<=、==和~=。关系运算符的说明如表 2-2 所示。

表 2-2 关系运算符

运算符	名 称	说 明	示 例
==	等于	a 等于 b 时返回 true,否则 false。==与=的含义不同,可以比较两个不同类型值	a==b
~=	不等于	与==恰恰相反	a~=b
>	大于	a 大于 b 时返回 true,否则 false	a>b
<	小于	a 小于 b 时返回 true,否则 false	a<b
>=	大于等于	a 大于等于 b 时返回 true,否则 false	a>=b
<=	小于等于	a 小于等于 b 时返回 true,否则 false	a<=b

下面是一个关系运算符示例:

```
local value1 = 1
local value2 = 2

if value1 == value2 then
    print("value1 == value2")
end

if value1 ~= value2 then
    print("value1 ~= value2")
end

if value1 > value2 then
    print("value1 > value2")
end

if value1 < value2 then
    print("value1 < value2")
end

if value1 <= value2 then
```

```
    print("value1 <= value2")
end
```

输出结果如下：

```
value1 ~= value2
value1 < value2
value1 <= value2
```

2.6.3　逻辑运算符

逻辑运算符是对布尔型变量进行运算，其结果也是布尔型。Lua 逻辑运算符有 3 个：and、or 和 not，如表 2-3 所示。

表 2-3　逻辑运算符

运算符	名　称	说　明
and	逻辑与	对于表达式 a and b，如果 a 为 false 或 nil 时，结果一定为“假”，决定最后结果是为 a，所以返回值为 a；如果 a 为非 false 和非 nil，最后的结果无论为“真”还是“假”，都是由 b 决定的，即：b 为 false 或 nil，结果为“假”，b 为其他值，结果为“真”，由于 b 决定结果，最后返回值为 b
or	逻辑或	对于表达式 a or b，如果 a 为非 false 和非 nil 时，结果一定为“真”，决定最后结果是为 a，所以返回值为 a；如果 a 为 false 或 nil，最后的结果无论为“真”还是“假”，都是由 b 决定的，即：b 为 false 或 nil，结果为“假”；b 为其他值，结果为“真”，由于 b 决定结果，最后返回值为 b
not	逻辑反	a 为 true 时，值为 false；a 为 false 时，值为 true

提示　Lua 的 and 和 or 运算符与其他语言逻辑与和逻辑或有很大的不同。在其他语言中逻辑运算的结果只有 true 和 false，而在 Lua 中逻辑运算的结果则不一定是 true 或 false。在 Lua 中 false 和 nil 被认为是“假”，而其他值都是“真”。此外，Lua 中 and 和 or 运算符具有“短路”特性，例如：a and b 或 a or b，最后的返回值是与决定最终结果的操作数(a 和 b)有关的。

下面是逻辑运算符 and 和 or 用法的示例：

```
print("========== or 示例 ==========")
print(10 or false)              --决定最后结果是 10,返回值是 10
print(10 or true)               --决定最后结果是 10,返回值是 10
print(true or 0)                --决定最后结果是 true,返回值是 true
print(false or 10)              --决定最后结果是 10,返回值是 10
print(nil or 10)                --决定最后结果是 10,返回值是 10
print(100 or 20)                --决定最后结果是 100,返回值是 100
print(20 or 100)                --决定最后结果是 20,返回值是 20

print("========== and 示例 ==========")
print(10 and false)             --决定最后结果是 false,返回值是 false
```

```
print(10 and true)          -- 决定最后结果是 true,返回值是 true
print(true and 0)           -- 决定最后结果是 0,返回值是 0
print(false and 10)         -- 决定最后结果是 false,返回值是 false
print(nil and 10)           -- 决定最后结果是 nil,返回值是 nil
print(100 and 20)           -- 决定最后结果是 20,返回值是 20
print(20 and 100)           -- 决定最后结果是 100,返回值是 100
```

2.6.4 运算优先级

运算优先级决定表达式的运算顺序,如果运算顺序不同,同样的表达式就有不同的结果。运算优先级的说明如表 2-4 所示。

表 2-4 运算优先级

运算优先级顺序	运 算 符	运算优先级顺序	运 算 符
1	^	4	<> <= >= ~= ==
2	* /	5	and
3	+ -	6	or

如表 2-4 所示的运算符优先顺序是从上到下,从高到低。

2.7 控制语句

结构化程序设计中的控制语句有 3 种,即顺序、分支和循环语句,而且只能用这 3 种结构来完成程序。Lua 程序通过控制语句来执行程序流,完成一定的任务。Lua 中的控制语句有以下几类:

(1) 分支语句: if。

(2) 循环语句: while、repeat、for。

(3) 与程序转移有关的跳转语句: break、return。

2.7.1 分支语句

分支语句提供了一种控制机制,使得程序具有“判断能力”,能够像人类的大脑一样分析问题。分支语句又称条件语句,条件语句使部分程序可根据某些表达式的值被有选择地执行。

由 if 语句引导的选择结构有 if 结构、if-else 结构和 elseif 结构 3 种。

如果条件表达式为 true 就执行语句组,否则就执行 if 结构后面的语句。语法结构如下:

```
-- if 结构
if 条件表达式 then
      语句组
end
```

```
-- if…else 结构
if 条件表达式 then
    语句组 1
else
    语句组 2
end

--elseif 结构
if 条件表达式 1 then
    语句组 1
elseif 条件表达式 2 then
    语句组 2
elseif 条件表达式 3 then
    语句组 3
…
elseif 条件表达式 n then
    语句组 n
else
    语句组 n + 1
end
```

if 引导的条件语句示例代码如下：

```
local score = 95

print('------- if 结构示例 --------')
if score >= 85 then
  print("您真优秀!")
end

if score < 60 then
  print("您需要加倍努力!")
end

if score >= 60 and score < 85 then
  print("您的成绩还可以,仍需继续努力!")
end

print('------ if - else 结构示例 ------')
if score < 60 then
  print("不及格")
else
  print("及格")
end

print('------ elseif 结构示例 ------')

local testscore = 76
local grade

if testscore >= 90 then
  grade = 'A'
elseif testscore >= 80 then
  grade = 'B'
```

```
elseif testscore >= 70 then
  grade = 'C'
elseif testscore >= 60 then
  grade = 'D'
else
  grade = 'F'
end
print("Grade = " .. grade)
```

运行结果如下：

```
------- if 结构示例 --------
您真优秀!
------- if-else 结构示例 ------
及格
-------elseif 结构示例 -------
Grade = C
```

2.7.2 循环语句

循环语句使语句或代码块的执行得以重复进行。Lua 支持 3 种循环类型：while、repeat 和 for。

while 和 for 循环是在执行循环体之前测试循环条件，而 repeat 是在执行完循环体之后测试循环条件。这就意味着 for 和 while 循环可能连一次循环体都未执行，而 repeat 将至少执行一次循环体。

1. while 语句

while 语句是一种先判断的循环结构，语句格式如下：

```
[initialization]
while termination do
    body
    [iteration]
end
```

其中中括号([])部分可以省略，initialization 是初始化语句，termination 是条件表达式，如果为 true 则执行循环体。iteration 是迭代语句，主要用于改变循环变量。

下面程序代码是通过 while 实现了查找平方小于 100 000 的最大整数：

```
local i = 0

while i * i < 100000 do
  i = i + 1
end

print(i .." * " .. i .." = " .. i*i)
```

结果是输出：

```
317 * 317 = 100489
```

这段程序的目的是找到平方数小于 100 000 的最大整数。使用 while 循环需要注意几点：while 循环语句中只能写一个表示式，而且是一个布尔型表达式；如果循环体中需要循环变量，就必须在 while 语句之前做好初值的处理，如上例中先给 i 赋值为 0；在循环体内部，必须通过语句更改循环变量的值，否则将会发生死循环。

2. repeat 语句

repeat 语句的使用与 while 语句相似，不过 repeat 语句是事后判断的循环结构，语句格式如下：

```
[initialization]
repeat
    body
    [iteration]
until termination
```

其中 termination 是条件表达式，与 while 不同，termination 为 true 则退出循环体。

下面程序代码是使用 repeat 实现了查找平方小于 100 000 的最大整数：

```
i = 0
repeat
  i = i + 1
until (i * i >= 100000)
print(i .. " * " .. i .. " = " .. i * i)
```

结果是输出：

```
317 * 317 = 100489
```

当程序执行到 repeat 语句时，首先无条件执行一次循环体，然后计算条件表达式，它的值必须是布尔型，如果值为 false，则再次执行循环体，然后到条件表达式准备再次判断，如此反复，直到条件表达式的值为 true 时跳出循环。

3. 典型 for 语句

典型 for 语句是应用最为广泛的一种循环语句，也是功能最强的一种。一般格式如下：

```
forvar = exp1,exp2,exp3 do
    body
end
```

其中 var 是循环变量，从 exp1 开始起，直到 exp2 的值为止，步长为 exp3。exp3 可以省略，默认值为 1。

下面是使用典型 for 循环实现平方表的程序代码：

```
local i
print("n n * n")
print("----------")

for i = 1, 10 do
  print(i .. " " .. i * i)
end
```

运行结果：

```
n   n*n
---------
1   1
2   4
3   9
4   16
5   25
6   36
7   49
8   64
9   81
10  100
```

这个程序步长为1，循环变量i的取值范围是大于等于1，小于等于10。因此，最后的结果是打印出1～10的平方。

4. 迭代 for-in 语句

迭代for语句可以帮助我们方便地遍历数组(table类型)。一般格式如下：

```
for i,v in ipairs(a) do
    body
end
```

其中a是数组变量，ipairs函数从a数组中取出循环变量i和元素v。

使用迭代for循环示例代码如下：

```
local numbers = {106, 22, 30, 48, 56, 60, 70, 80, 90, 109 }

for i,v in ipairs(numbers) do
  print(i .. " : " .. v)
end
```

运行结果：

```
1 : 106
2 : 22
3 : 30
4 : 48
5 : 56
6 : 60
7 : 70
8 : 80
9 : 90
10 : 109
```

上述代码numbers是数组变量。

2.7.3 跳转语句

跳转语句包括break语句和返回语句return。

1. break 语句

break 语句可用于 switch 引导的分支结构以及上述 3 种循环结构，它的作用是强行退出循环结构，不执行循环结构中剩余的语句。break 语句格式如下：

```
break
```

break 语句示例如下：

```
local numbers = {106, 22, 30, 48, 56, 60, 70, 80, 90, 109 }

for i,v in ipairs(numbers) do
  if i == 3 then
    break
  end
  print(i .. " : " .. v)
end
```

在示例中当条件 i==3 的时候执行 break 语句，程序运行结果是：

```
1 : 106
2 : 22
```

2. 返回语句 return

return 语句可以从当前函数中退出，返回到调用该函数的语句处。返回语句有两种格式：

```
return expression
return
```

2.8 table 类型

表类型是 Lua 非常重要的数据类型，也是 Lua 唯一能够描述数据结构的类型。表类型很灵活，可以描述多种数据结构。表类型本身是一种基于“键值对”字典结构，它可以描述数组、链表、队列和字典(或 Map)等。此外，在面向对象的技术中，table 还可以描述对象。

下面分别介绍表类型描述的两种典型数据结构——字典和数组，进而了解表类型的创建、初始化、访问和遍历等过程。

2.8.1 字典

表类型使用{}创建和初始化，{}表示方式被称为 table 构造器(constructor)，最简单的表示方式是空{}，但是在描述字典和数组时构造器的表示方式是不同的。字典结构的 table 创建、初始化和访问过程示例如下：

```
Student1 = {id = "100",name = "Tony",age = 18}                          ①
Student2 = {["id"] = "100",["name"] = "Tony",["age"] = 18}              ②

print("Student1 ID:" .. Student1["id"])                                 ③
print("Student1 Name:" .. Student1["name"])
print("Student1 Age:" .. Student1["age"])
```

```
print("Student2 ID:" .. Student2.id)                                    ④
print("Student2 Name:" .. Student2.name)
print("Student2 Age:" .. Student2.age)
```

上述第①和第②行代码分别创建了 Student1 和 Student2 两个 table 变量。第②行代码的键表示方式比较复杂,键是在["和"]之间包裹起来。而第①行代码的键表示方法非常简洁。

在访问这些 table 变量的时候,有两种方法可以使用,第③行代码 Student1["id"]是访问 Student1 的 id 键对应的内容,这里的键可以放置在双引号(")之间。而另一种访问方式见第④行代码 Student2.id,这种方式是通过点访问符(.)直接访问 id。

采用哪种方式创建以及采用哪种方式访问完全是自己的兴趣使然。

下面介绍字典结构的遍历。遍历字典结构 table 可以使用迭代 for-in 语句实现,示例代码如下:

```
Student1 = {id = "100",name = "Tony",age = 18}

for k,v in pairs(Student1) do                                           ①
  print(k .. " : " .. v)
end
```

运行结果如下:

```
id : 100
name : Tony
age : 18
```

第①行代码使用 pairs 函数,该函数可以从字典结构 table 变量中返回键(k)和值(v)。

2.8.2 数组

数组结构是一种特殊的 table 结构,创建和初始化数组结构的 table 类型也是使用构造器{}实现的。

数组结构的 table 创建、初始化和遍历过程示例如下:

```
local studentList = {"张三","李四","王五","董六"}                        ①

print("------遍历 1--------")
for i,v in ipairs(studentList) do                                       ②
  print(i .. " : " .. v)
end

print("------遍历 2--------")
for k,v in pairs(studentList) do                                        ③
  print(k .. " : " .. v)
end

print("------遍历 3--------")
for i = 1, #studentList do                                              ④
  print(i .. " : " .. studentList[i])
end
```

运行结果如下：

```
------遍历 1-------
1：张三
2：李四
3：王五
4：董六
------遍历 2-------
1：张三
2：李四
3：王五
4：董六
------遍历 3-------
1：张三
2：李四
3：王五
4：董六
```

上述第①行代码是创建并初始化数组结构 table 变量 studentList，这种结构不再需要键了。

接着通过第②、③、④行代码进行遍历 studentList 数组，其中第②行代码使用了 ipairs 函数，ipairs 函数返回数组索引(i)和值(v)。

注意 ipairs 函数不能使用于字典结构 table 的遍历。

第③行代码使用 pairs 函数，该函数可以从 studentList 变量中返回键(k)和值(v)。对于数组结构键就是索引。

第④行代码使用典型的 for 循环，其中 # studentList 是获得数组的长度，对于旧版本的 Lua 可以使用 table. getn(studentList)函数获得长度。

注意 典型的 for 循环不能使用于字典结构 table 的遍历。

2.9 字符串类型

在 Lua 中字符串可以用双引号或单引号括起来表示，示例如下：

```
local s1 = "Hello Lua"
local s2 = 'Hello Lua'
```

有的时候需要将字符串拼接起来，在 Java 中使用＋号将字符串连接，而在 Lua 中使用双点“..”将字符串拼接起来。示例代码如下：

```
local s = "Hello" .. " Lua"
```

获得字符串长度需要使用 String 库提供的 string. len(s)函数。示例代码如下：

```
local slen = string.len(s)
```

其中 s 是字符串。

String 库还提供了很多函数,下面分别介绍。

2.9.1 字符串截取

字符串截取由 String 库提供的 string.sub(s, n, m)函数来实现,该函数截取从第 n 个字符到第 m 个字符之间的字符串。示例代码如下:

```
local s = "Hello Lua"

local sub1 = string.sub(s,1,3)                                    ①
print(sub1)                    -- 输出 Hel
local sub2 = string.sub(s,2, -2)                                  ②
print(sub2)                    -- 输出 ello Lu
```

输出结果如下:

```
Hel
ello Lu
```

上述第①行代码是截取第 1～3 个字符串,与 Java 等语言不同的是 Lua 中字符串的索引下标从 1 开始。

索引可以有负数,－1 表示最后一个字符,－2 表示倒数第 2 个字符,所以第②行代码截取结果是 ello Lu,索引 2 的字符是 e,索引－2 的字符是 u。

2.9.2 字符串转换

String 库提供了字符串转换相关函数,包括大写、小写、重复和翻转等。示例代码如下:

```
local s = "Hello Lua"

local s1 = string.lower(s)                                        ①
print(s1)

local s2 = string.upper(s)                                        ②
print(s2)

local s3 = string.rep("Hello ", 3)                                ③
print(s3)

local s4 = string.reverse("Hello")                                ④
print(s4)
```

输出结果如下:

```
hello lua
HELLO LUA
Hello Hello Hello
olleH
```

上述第①行代码是使用函数 string. lower(s)将 s 字符串转换为小写字符串，输出结果为 hello lua。

第②行代码是使用函数 string. upper(s)将 s 字符串转换为大写字符串，输出结果为 HELLO LUA。

第③行代码是使用函数 string. rep("Hello", 3)重复拼接 3 次，输出结果为 Hello Hello Hello。

第④行代码是使用函数 string. reverse("Hello")翻转字符串，输出结果为 olleH。

2.9.3　字符串查询

String 库提供的 string. find(s, sub)函数是从 s 字符串中查找 sub 字符串，函数有两个返回值，示例代码如下：

```
local s = "Hello Lua"

local i, j = string.find(s, "Hello")                                    ①

print(i, j)
```

输出结果如下：

```
1   5
```

上述第①行代码是使用函数 string. find(s, "Hello")在"Hello Lua"中查找"Hello"，返回值 i 为找到的开始位置，j 为结束的位置。

2.9.4　字符串格式化

String 库提供的 string. format("format",...)函数是对字符串进行格式化输出，其中的格式化所需要参数有 s、q、c、d、E、e、f、g、G、i、o、u、X 和 x，如表 2-5 所示。这些参数与 C 语言中的 printf("format",...)函数类似。

表 2-5　格式化参数

格式化参数	说　明
s	格式化输出字符串
q	接受一个字符串并将其转化为可被 Lua 编译器安全读入的格式
c	格式化输出字符
d	格式化输出数字
E	格式化输出科学计数法 E 大写
e	格式化输出科学计数法 e 小写
f	格式化输出浮点数
g	格式化输出紧凑浮点数
G	格式化输出紧凑浮点数
i	格式化输出数字
o	格式化输出八进制数字

续表

格式化参数	说　明
u	格式化输出无符号数字
X	格式化输出十六进制数,大写表示
x	格式化输出十六进制数,小写表示

示例代码如下:

```
local s = string.format("%s %q", "Hello", "Lua")
print(s)

local s = string.format("%c%c%c", 76,117,97)
print(s)

local s = string.format("%e, %E", 3.14, 3.14)
print(s)

local s = string.format("%f, %g, %G", 3.14, math.pi, math.pi)
print(s)

local s = string.format("%d, %i, %u", 10, 10, -10)
print(s)

local s = string.format("%o, %x, %X", 10, 10, 10)
print(s)
```

输出结果如下:

```
Hello "Lua"
Lua
3.140000e+000, 3.140000E+000
3.140000, 3.14159, 3.14159
10, 10, 4294967286
12, a, A
```

2.10 函数

将程序中反复执行的代码封装到一个代码块中,这个代码块模仿了数学中的函数,具有函数名、参数和返回值。

Lua 中的函数很灵活,可以有全局函数和局部函数等,函数还可以作为一个值传递。

2.10.1 使用函数

使用函数首先需要定义函数,然后在合适的地方调用该函数,函数的语法格式如下:

```
function 函数名(参数列表) {
    语句组
    [return 返回值]
}
```

在 Lua 中定义函数的关键字是 function，函数名需要符合标识符命名规范，参数列表可以有多个，之间用逗号(,)分隔，极端情况下参数可以没有。

如果函数需要返回值，可使用 return 语句将值返回，如果没有返回值，则函数体中可以省略 return 语句。

函数定义示例代码如下：

```
function rectangleArea(width, height)                    ①
  local area = width * height
  return area                                            ②
end

print("320x480 的长方形的面积:" .. rectangleArea(32, 64))    ③
```

上述第①行代码是定义计算长方形的面积 rectangleArea，它有两个参数，分别是长方形的宽和高。

第②行代码是返回函数计算结果。

调用函数的过程是通过第③行代码中的 rectangleArea(32, 64)语句实现，调用函数时需要指定函数名和参数值。

2.10.2 变量作用域

变量可以定义在函数体外，即全局变量；可以在函数内定义，即局部变量。局部变量作用域是在函数内部有效，如果超出函数体就会失效。

示例代码如下：

```
local global = 1                                         ①

function f()
  local local1 = 2                                       ②
  global = global + 1                                    ③
  return global
end

f()

print(global)                                            ④
print(local1)                                            ⑤
```

上述第①行代码定义了全局变量 global。

第②行代码定义了局部变量 local1，local1 是在函数体内部定义的。

第④行代码是打印全局变量 global。

第⑤行代码是打印局部变量 local1，该语句在运行时会发生错误，这是因为 local1 是局部变量，作用域是在 f 函数体内部。

2.10.3 多重返回值

Lua 的函数可以返回多个值，它的函数表达式可以同时赋值给多个变量，这是在其他语

言中没有见到的。C 和 C++等语言要想返回多个值,可以通过给函数传递指针类型的参数实现,在 Java 中可通过传递引用类型参数实现。

多重返回值示例代码如下:

```
--定义计算长方形面积和周长函数
function caleRectangle(width, height)

  local area = width * height
  local perimeter = (width + height) * 2

  return area,perimeter                                    ①
end

--获得计算长方形面积和周长
local area, perimeter = caleRectangle(10, 15)              ②
print("宽 10 高 15,长方形面积:" .. area .. " 长方形周长:" .. perimeter)
```

上述第①行代码是函数 caleRectangle 的返回语句,该语句中返回多个值,多个值之间用逗号“,”分隔。

第②行代码是调用 caleRectangle 函数,并且可以直接赋值给变量 area 和 perimeter,这些变量之间也是用逗号“,”分隔。

2.11 闭包函数

一门计算机语言支持闭包函数的前提如下:

(1) 支持函数类型,能够将函数作为参数或返回值传递。

(2) 支持函数嵌套。

这两个前提在 Lua 中都是满足的,Lua 中可以在一个函数中定义另一个函数,函数还可以作为一个“值”进行传递,即作为函数的参数或作为返回值返回。

2.11.1 嵌套函数

在此之前定义的函数都是全局函数,它们定义在全局作用域中,也可以把函数定义在另外的函数体中,称作嵌套函数。

下面看一个示例:

```
function calculate(opr, a, b)                              ①

  --定义 + 函数
  function add(a, b)                                       ②
    return a + b
  end

  --定义 - 函数
  function sub(a, b)                                       ③
    return a - b
```

```
end

local result

if opr == "+" then
  result = add(a, b)                                    ④
else
  result = sub(a, b)                                    ⑤
end

return result                                           ⑥
end

local res1 = calculate("+", 10, 5)                      ⑦
print("10 + 5 = " .. res1)

local res2 = calculate("-", 10, 5)                      ⑧
print("10 - 5 = " .. res2)
```

上述第①行代码定义 calculate 函数，它的作用是根据运算符进行数学计算，它的参数 opr 是运算符，参数 a 和 b 是要计算的数值。

在 calculate 函数体内，第②行代码定义了嵌套函数 add，对两个参数进行加法运算。

第③行代码定义了嵌套函数 sub，对两个参数进行减法运算。

第④行代码是在运算符为＋号情况下使用 add 函数进行计算，并将结果赋值给 result。

第⑤行代码是在运算符为“－”号情况下使用 sub 函数进行计算，并将结果赋值给 result。

第⑥行代码是返回函数变量 result。

第⑦行代码调用 calculate 函数进行加法运算。

第⑧行代码调用 calculate 函数进行减法运算。

程序运行结果为：

```
10 + 5 = 15
10 - 5 = 5
```

在函数嵌套中，默认情况下嵌套函数的作用域是在外函数体内。

2.11.2　返回函数

返回函数是指把函数作为另一个函数的返回值使用，如下例：

```
--定义计算长方形面积函数
function rectangleArea(width, height)
  local area = width * height
  return area
end

--定义计算三角形面积函数
function triangleArea(bottom, height)
  local area = 0.5 * bottom * height
```

```
  return area
end

function getArea(type)                                          ①
  local returnFunction                                          ②
  if type == "rect" then   -- rect 表示长方形
    returnFunction = rectangleArea                              ③
  else                     -- tria 表示三角形
    returnFunction = triangleArea                               ④
  end
  return returnFunction                                         ⑤
end

-- 获得计算三角形面积函数
local area = getArea("tria")                                    ⑥
print("底 10 高 15,三角形面积: " .. area(10, 15))               ⑦

-- 获得计算长方形面积函数
local area = getArea("rect")                                    ⑧
print("宽 10 高 15,计算长方形面积: " .. area(10, 15))           ⑨
```

上述第①行代码定义函数 getArea(type),它的返回值是一个函数。

第②行代码是声明 returnFunction 变量保存要返回的函数名。

第③行代码是在类型 type 为 rect(即长方形)情况下,把前面定义的 rectangleArea 函数名赋值给 returnFunction 变量。

类似第④行代码是在类型 type 为 tria(即三角形)情况下,把前面定义的 triangleArea 函数名赋值给 returnFunction 变量。

第⑤行代码是将 returnFunction 变量返回。

第⑥和⑧行代码是调用函数 getArea,返回值 area 是函数类型。

第⑦和⑨行代码中的 area(10,15)是调用函数。

上述代码运行结果如下:

```
底 10 高 15,三角形面积: 75.0
宽 10 高 15,计算长方形面积: 150.0
```

2.11.3 使用闭包表达式

还可以采用匿名函数形式的闭包表达式。修改上面的示例代码如下:

```
function getArea(type)
  local returnFunction

  if type == "rect" then   -- rect 表示长方形
    returnFunction = function(width, height)                    ①
      local area = width * height
      return area
    end
  else                     -- tria 表示三角形
    returnFunction = function(bottom, height)                   ②
```

```
        local area = 0.5 * bottom * height
        return area
    end
  end
  return returnFunction
end

-- 获得计算三角形面积函数
local area = getArea("tria")
print("底 10 高 15,三角形面积: " .. area(10, 15))

-- 获得计算长方形面积函数
local area = getArea("rect")
print("宽 10 高 15,计算长方形面积: " .. area(10, 15))
```

采用匿名函数赋值给 returnFunction 变量，第①和第②行代码采用的就是闭包表达式。

2.12 Lua 中的面向对象

面向对象是一种新兴的程序设计方法和设计规范，其基本思想是使用对象、类、继承、封装、消息等基本概念来进行程序设计。从现实世界中客观存在的事物(即对象)出发来构造软件系统，并且在系统构造中尽可能运用人类的自然思维方式。

面向对象最重要的两个概念就是：对象和类。

对象是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位。一个对象由一组属性和对这组属性进行操作的一组函数组成。

类是具有相同属性和函数的一组对象的集合，它为属于该类的所有对象提供了统一的抽象描述，其内部包括属性和函数两个主要部分。但是需要注意的是，Lua 语言中并没有提供类的定义能力，但可以把表类型变量当成对象使用。

2.12.1 Lua 中的对象

在 Lua 语言中虽然不能定义类，但可以将表类型变量当成对象，对象包括数据和操作两个主要部分。下面是使用表类型描述学生对象：

```
Student = {id = 100, name = "Tony"}                                    ①

function Student.toString()                                            ②
  local s = "Name:" .. Student.name .. " id:" .. Student.id
  return s
end
```

第①行代码定义 Student 对象，其中的 id 和 name 是两个数据成员。

第②行代码的 Student. toString( ) 函数是对象的操作，toString( ) 函数只能属于 Student 对象，这种属于特定对象的函数被称为“方法”，出于习惯在本书中仍然将这些方法

称为函数。

在面向对象的语言中一般都有代替本身对象的代词,Java 和 C++是用 this,Objective-C 中是 self,而 Lua 可以使用 self 表示自身对象。

```
Student = {id = 100, name = "Tony"}

function Student:toString()                                    ①
  local s = "Name:" .. self.name .. " id:" .. self.id          ②
  return s
end

print(Student:toString())                                      ③
```

上述第①行代码是定义 Student:toString()函数,其中 Student 与 toString()之间是用冒号":",而不是点".",冒号表示的函数参数中省略了 self,第①行代码的函数也可以写成 function Student. toString(self)。

第②行代码中使用 self 代替了 Student 对象。

第③行代码是访问 Student 对象的 toString()函数,也可以用冒号":"访问 toString()函数。

2.12.2 类的实现

Lua 中并没有类的定义,但是可以自己实现类。类是对象的模板,一个类能够创建出多个对象。实现类的思路是把 Student 作为原型对象,原型就是一个模板,通过 Student 原型来创建其他的对象,那么就可以将 Student 原型对象称为"类"了。创建的对象遇到一个未知操作时,会到 Student 原型中去查找,这也符合类的继承原则。

Student 类的示例代码如下:

```
Student = {id = 100, name = "Tony"}

function Student:toString()
  local s = "Name:" .. self.name .. " id:" .. self.id
  return s
end

function Student:create(o)                                     ①
  o = o or{}                                                   ②
  setmetatable(o, self)                                        ③
  self.__index = self                                          ④
  return o
end

student1 = Student:create({id = 200, name = "Tom"})            ⑤
print(student1:toString())

student2 = Student:create({id = 300, name = "Ben"})            ⑥
print(student2:toString())
```

上述第①行代码定义 Student:create(o)函数，其中 o 是 table 类型的参数。

第②行代码 o=o or{}是当参数 o 没有创建的时候，重新创建一个空的 table。

第③行代码 setmetatable(o, self)是把 self 当作元表(Metatable)设置给 o 对象，元表是 Lua 中的重要概念，每一个 table 都可以加上元表，元表可以改变相应的 table 的行为。

第④行代码 self.__index=self 表示首先在当前对象中查找操作，如果当前对象中没有，就会到原型 Student 中查找操作。

第⑤行代码是通过 Student 原型(类)创建 student1 对象。

第⑥行代码是通过 Student 原型(类)创建 student2 对象。

本章小结

通过对本章的学习，读者可以了解 Lua 语言的基本语法，包括数据类型、表达式及对象等概念。

第3章 Cocos2d-x Lua API与环境搭建

在开始详细介绍 Cocos2d-x Lua API 之前，有必要先了解一下手机游戏引擎有哪些，了解一下 Cocos2d-x 的前世今生。之后还会介绍开发工具。

3.1 移动平台游戏引擎介绍

游戏引擎是指一些已编写好的游戏程序模块。游戏引擎包含以下子系统：渲染引擎（即“渲染器”，含二维图像引擎和三维图像引擎）、物理引擎、碰撞检测系统、音效、脚本引擎、电脑动画、人工智能、网络引擎以及场景管理。

在目前移动平台游戏引擎中主要可以分为 2D 和 3D 引擎。2D 引擎主要有 Cocos2d-iphone、Cocos2d-x、Corona SDK、Construct 2、WiEngine 和 Cyclone 2D，3D 引擎主要有 Unity、Unreal Development Kit、ShiVa 3D 和 Marmalade。此外，还有一些针对 HTML 5 的游戏引擎：X-Canvas 和 Sphinx 等。

这些游戏引擎各有千秋，目前得到市场普遍认可的 2D 引擎是 Cocos2d-x，3D 引擎是 Unity，但是 Cocos2d-x 也在向 3D 游戏引擎发展，在目前 Cocos2d-x 3.11 版本中已经包含了 3D 功能，而 Unity 也在向 2D 游戏引擎发展。

3.2 Cocos2d 家谱

在介绍 Cocos2d-x 之前有必要先介绍一下 Cocos2d 的家谱，图 3-1 展示了 Cocos2d 的家谱。

Cocos2d 最早是由阿根廷的 Ricardo 和他的朋友使用 Python 开发的，后移植到 iPhone 平台，使用的语言是 Objective-C。随着在 iPhone 平台取得了成功，Cocos2d 引擎变得更加多元化。其中各个引擎介绍如下：

(1) ShinyCocos。使用 Ruby 对 Cocos2d-iphone 进行封装，使用 Ruby API 开发。

(2) CocosNet。在 MonoTouch 平台上使用的 Cocos2d 引擎，采用. NET 实现。

(3) Cocos2d-android。为 Android 平台使用的 Cocos2d 引擎，采用 Java 实现。

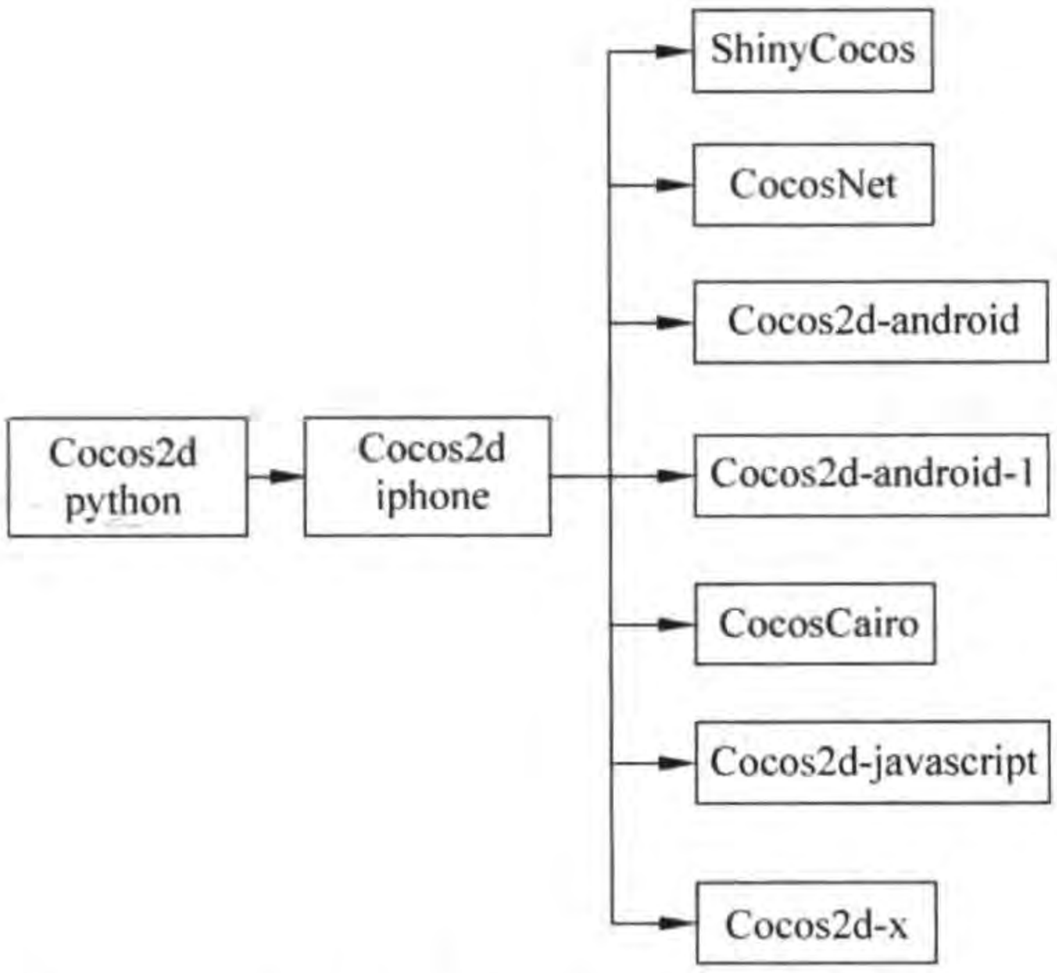


图 3-1　Cocos2d 的家谱

(4) Cocos2d-android-1。为 Android 平台使用的 Cocos2d 引擎，采用 Java 实现，由国内人员开发。

(5) Cocos2d-javascript。采用 JavaScript 脚本语言实现的 Cocos2d 引擎。

(6) Cocos2d-x。采用 C++实现的 Cocos2d 引擎，它是由 Cocos2d-x 团队开发的分支项目。另外 Cocos2d-x 还采用了 JavaScript 和 Lua 脚本绑定技术，可以为开发者提供基于 JavaScript 和 Lua 语言的 API。

此外，历史上 Cocos2d 还出现过很多分支，随着技术的发展这些分支大都消亡了，其中最有生命力的当属 Cocos2d-x 引擎。

3.3　Cocos2d-x 设计目标

Cocos2d-x 设计目标如图 3-2 所示。横向能够支持各种操作系统，桌面系统包括 Windows、Linux 和 Mac OS X，移动平台包括 iOS、Android、WinPhone、Bada、BlackBerry 和 MeeGo 等。纵向方面向下能够支持 OpenGL ES1.1、OpenGL ES1.5 和 OpenGL ES2.0，以及 DirectX11 等技术，向上支持 JavaScript 和 Lua 脚本绑定。

简单地说，Cocos2d-x 的设计目标是实现跨平台，用户不再为同一款游戏在不同平台发布而进行编译。而且 Cocos2d-x 为程序员考虑得更多，很多程序员可能对于 C++不熟悉，针对这种情况可以使用 JavaScript 和 Lua 语言开发游戏，如图 3-3 所示。

如图 3-3 所示，通过 Cocos2dJavaScript 引擎，可以开发网页游戏和本地游戏。图中，A 线路和 B 线路都是给掌握 JavaScript 语言的程序员准备的，通过 A 线路，使用 Cocos2dJavaScript 引擎开发基于 HTML5 的网页游戏。同样的 JavaScript 代码，也可以通过 B 线路，使用 JS binding(JS 绑定)技术通过 Cocos2d C++引擎开发本地游戏。这样同样

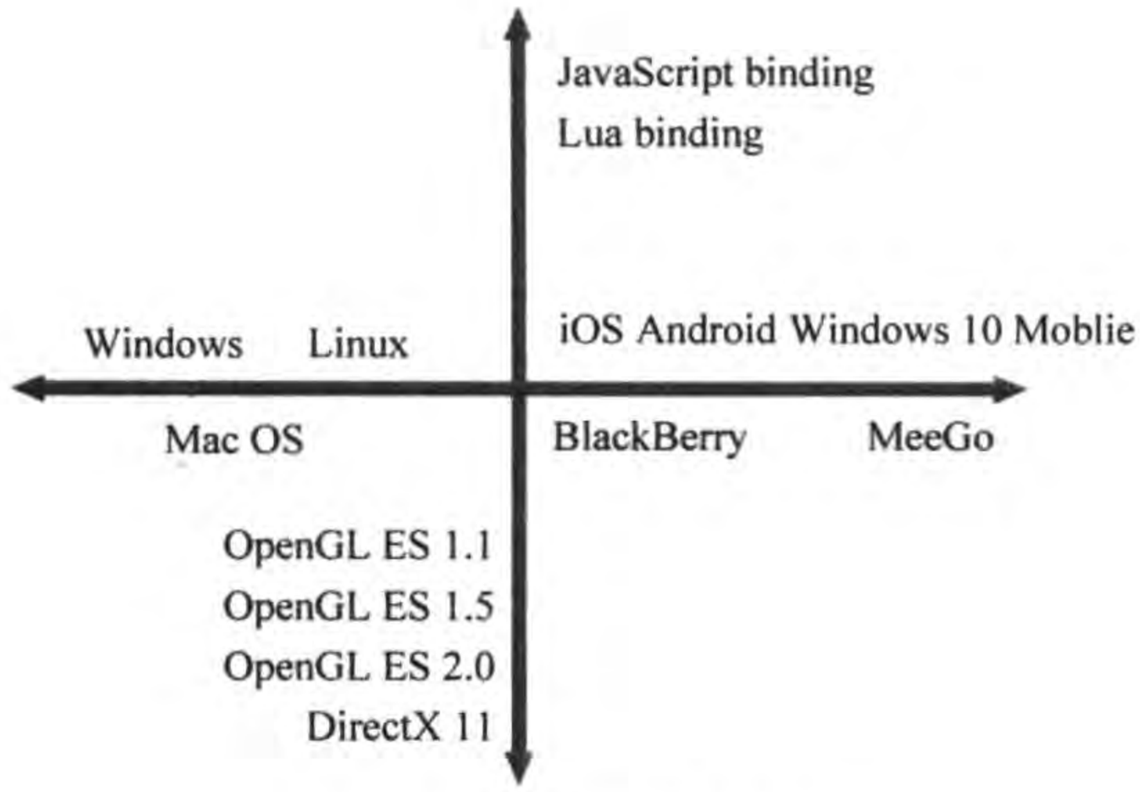


图 3-2　Cocos2d-x 设计目标

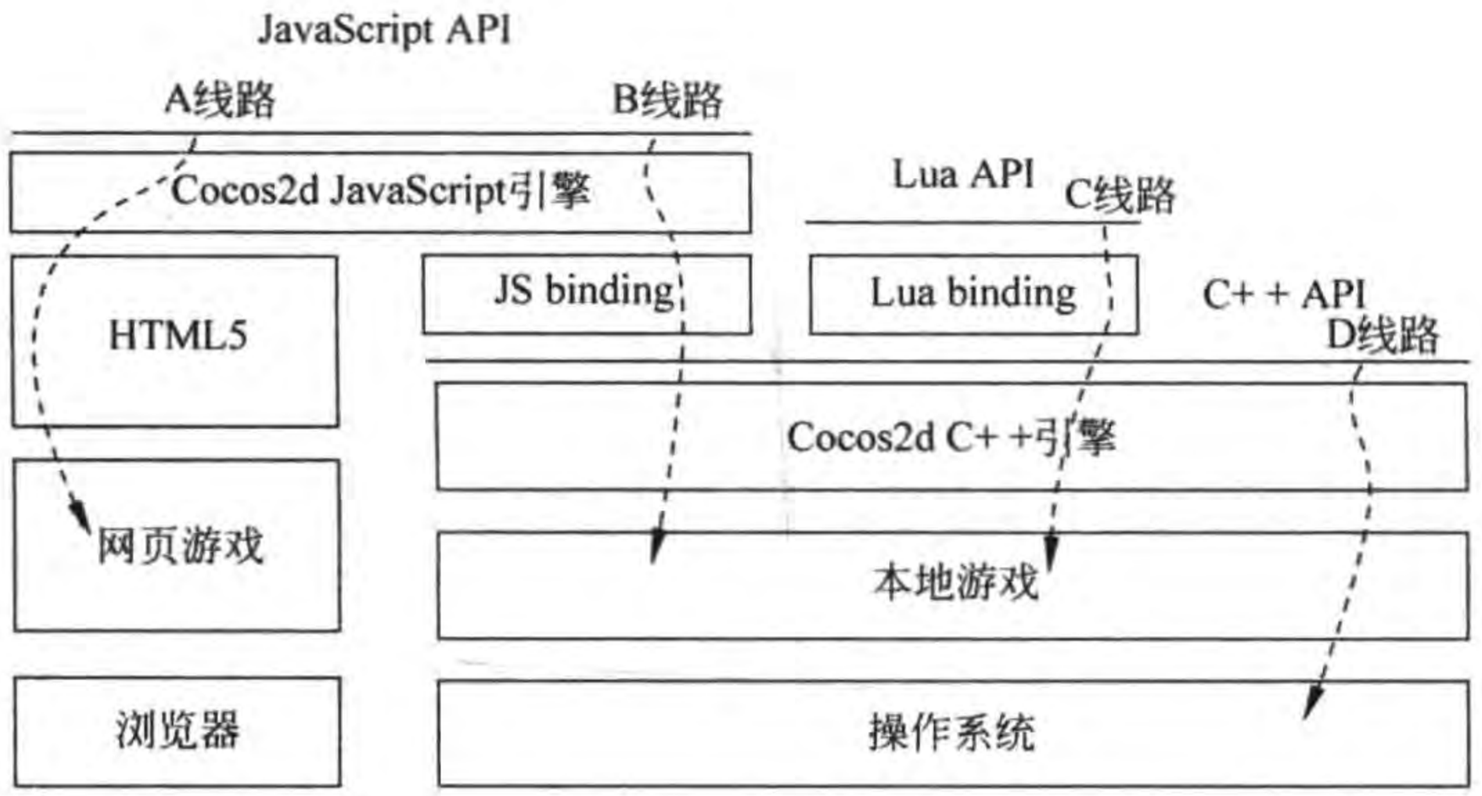


图 3-3　Cocos2d-x 游戏引擎架构

的 JavaScript 代码就可以实现在不同平台下运行。

图 3-3 中的 C 线路是本书介绍的，是为熟悉 Lua 脚本的程序员准备的。使用 Lua binding(Lua 绑定)技术通过 Cocos2d C++引擎开发本地游戏。

图 3-3 中的 D 线路是给熟悉 C++的程序员准备的。通过 C++API 访问 Cocos2d C++引擎，开发本地游戏。

通过 Cocos2d-x 引擎，Cocos2d-x 团队构建了自己的技术生态圈，使得游戏开发越来越简单。

3.4　搭建 Cocos2d-x Lua API 开发环境

使用 Cocos2d-x Lua API 开发游戏，主要的程序代码是 Lua 语言，因此，凡是能够开发 Lua 语言工具都适用于 Cocos2d-x Lua API 游戏开发。如果选择 IDE(集成开发)工具，可

以使用前一章介绍的 Eclipse＋Lua Development Tools。如果不考虑 IDE 工具，使用文本编辑器也是不错的选择。

提示　Cocos 团队曾经开发过 Cocos2d-x JS 和 Lua API 的 IDE 工具——Cocos Code IDE，但是这个工具已经被 Cocos 团队放弃维护了，因此新版本的 Cocos2d-x JS 和 Lua API 开发已经很难配置 Cocos Code IDE 工具。

3.4.1　安装 Visual Studio 开发工具

Eclipse＋Lua Development Tools 等 IDE 开发工具，只能编写 Cocos2d-x Lua API 程序，而不能编译和运行，为了编译和运行 Cocos2d-x Lua API 工程，则需要安装另外的一些 IDE 工具，每个平台 IDE 工具不同，Windows 系统是 Visual Studio 工具，Mac OS X 和 iOS 系统是 Xcode，此外还有 Android 是 Eclipse 和 Android Studio 等。

为了降低广大读者的学习成本，本书在第 19 章之前所有案例都是基于 Windows 系统 Win32 平台，这需要安装 Visual Studio 开发工具。下面介绍 Visual Studio 开发工具。

Visual Studio 是微软公司开发的基于 Windows 操作系统下的 IDE(集成开发)工具，Visual Studio 可以使用 C++、C＃和 Visual Basic 等语言开发基于 Windows 的本地或 Web 应用。使用 Visual Studio 开发 Cocos2d-x 主要是使用它的 C++语言进行编译和运行，还有它的标准库，而一般情况下不需要使用 VC++特有的类库或函数库。

Visual Studio 目前最新版本是 2015，但是 Cocos2d-x 3.11 主要支持使用 2013 版本。与 Xcode 和 Eclipse 免费的不同，Visual Studio 有多个版本，其中只有 Visual Studio Community 版本是免费的，读者可以在如下地址 https://www.visualstudio.com/downloads/download-visual-studio-vs 找到 Visual Studio Community 版。

3.4.2　下载和使用 Cocos2d-x Lua API 官方案例

首先到 Cocos 官方网站下载 Cocos2d-x 开发包，目前 Cocos2d-x 3.11 版已经发布。Cocos2d-x 3.11 下载解压后的目录结构如图 3-4 所示。

如果要运行官方的案例可以进入到 build 文件夹，build 文件夹中的内容如图 3-5 所示，这里包含各个平台编译和运行案例的工程文件等，主要的文件说明如下：

(1) cocos2d_libs.xcodeproj 是 Cocos2d-x 引擎的 Xcode 工程文件，通过这个工程文件可以将 Cocos2d-x 引擎编译成为 Mac OS X 和 iOS 下的静态库文件。

(2) cocos2d_tests.xcodeproj 是 Cocos2d-x 案例的 Xcode 工程文件。

(3) android-build.py 是 Python 脚本文件，可以将 Cocos2d-x 引擎编译成为 Android 平台下的动态库文件。

(4) cocos2d-win8.1-universal.sln 是 Visual Studio 的解决方案，可以编译为 Windows 8 和 Windows Phone 8 平台所需文件。

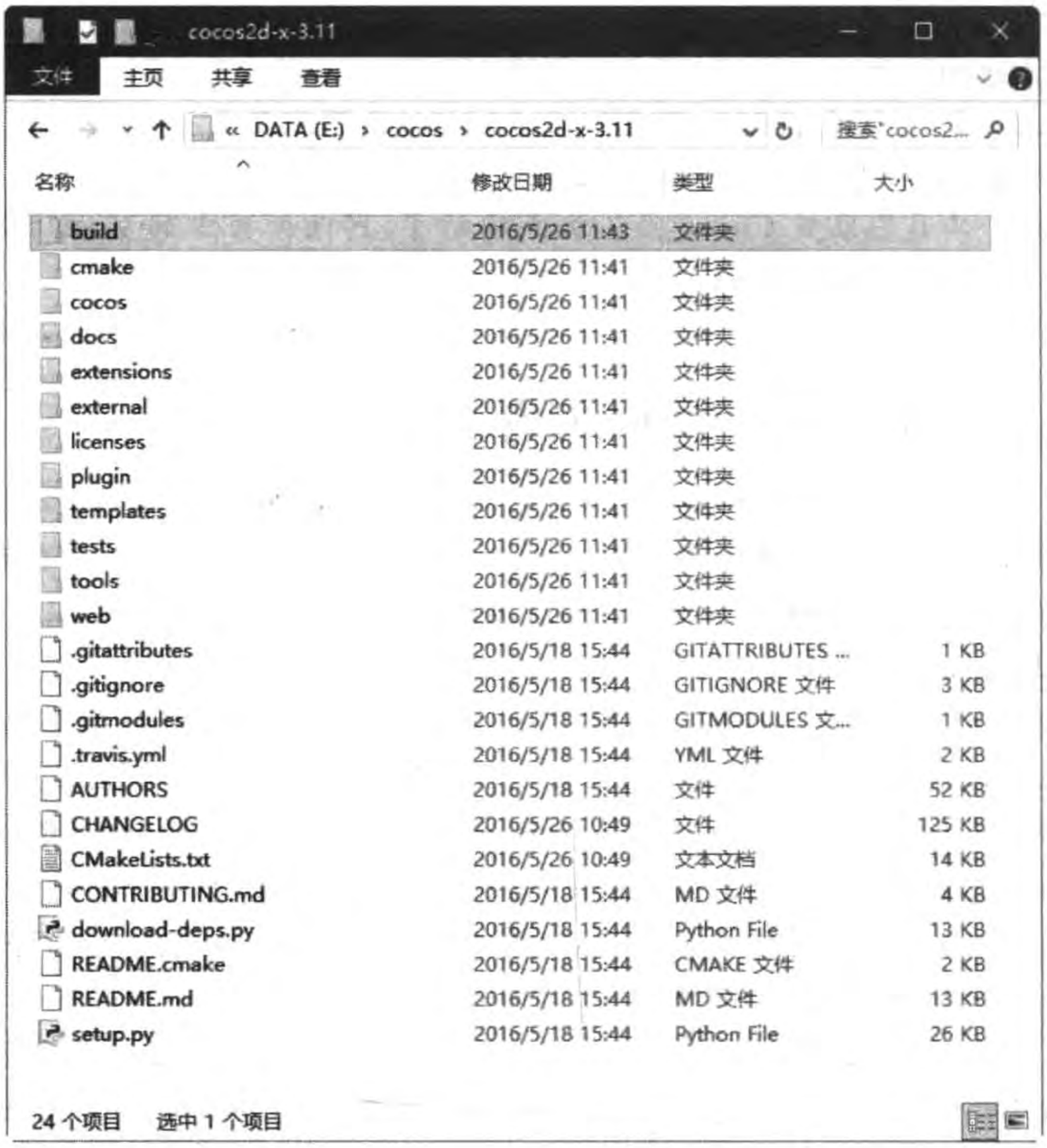


图 3-4 Cocos2d-x 开发包内容

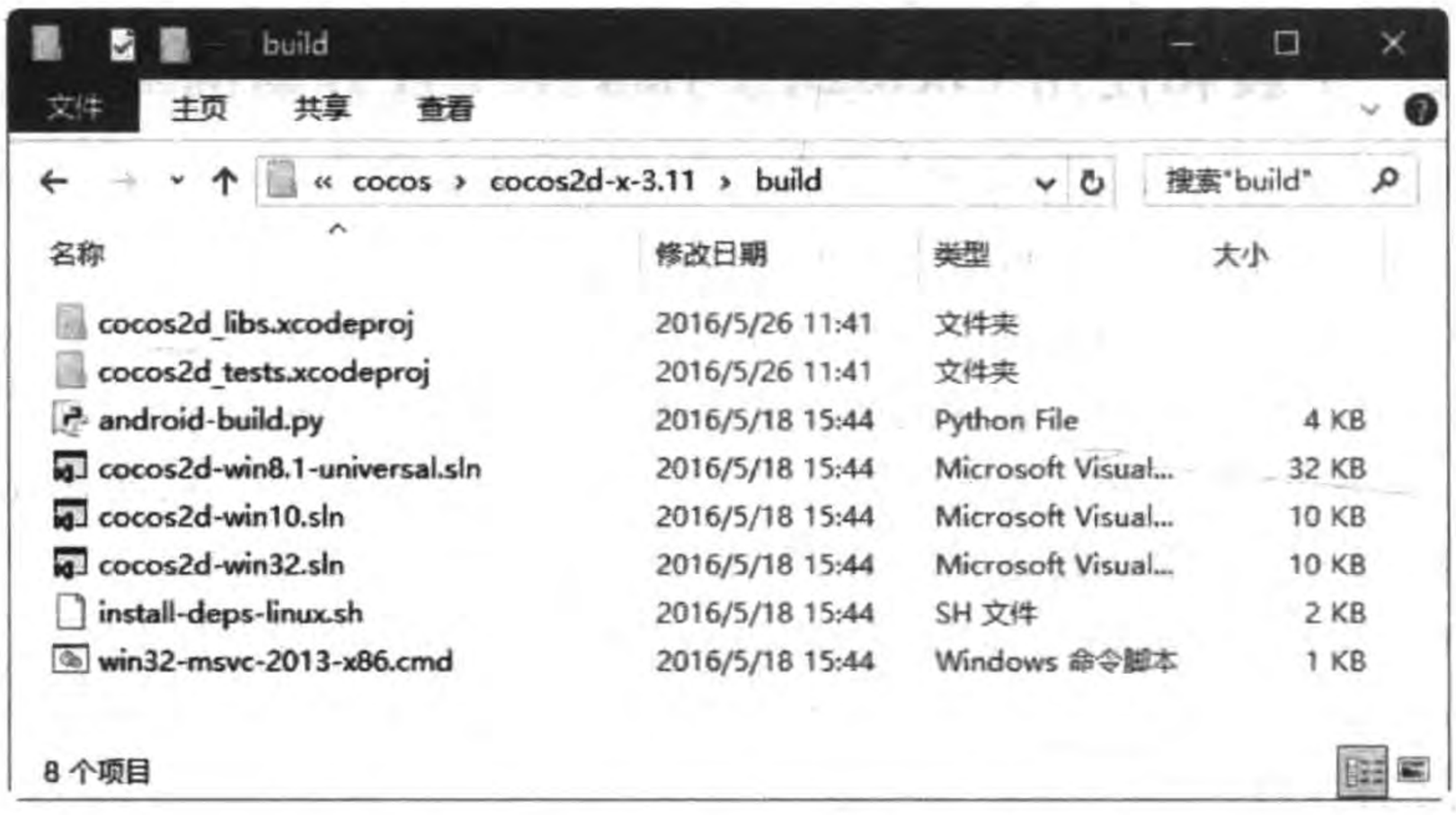


图 3-5 Cocos2d-x 开发包 build 文件夹内容

(5) cocos2d-win10.sln 是 Visual Studio 的解决方案，可以编译为 Windows 10 平台所需文件。

(6) cocos2d-win32.sln 是 Visual Studio 的解决方案，可以编译为 Win32 平台所需文件。

如果启动 cocos2d-win32.sln 解决方案，则进入图 3-6 所示的 Visual Studio 界面，其中的 lua-tests 工程是 Cocos2d-x 官方提供的案例工程。需要选中 lua-tests 工程，在右击弹出的快捷菜单中选择“设置启动项目”命令，然后运行上方工具栏中的运行调试按钮▶，运行 lua-tests 工程。

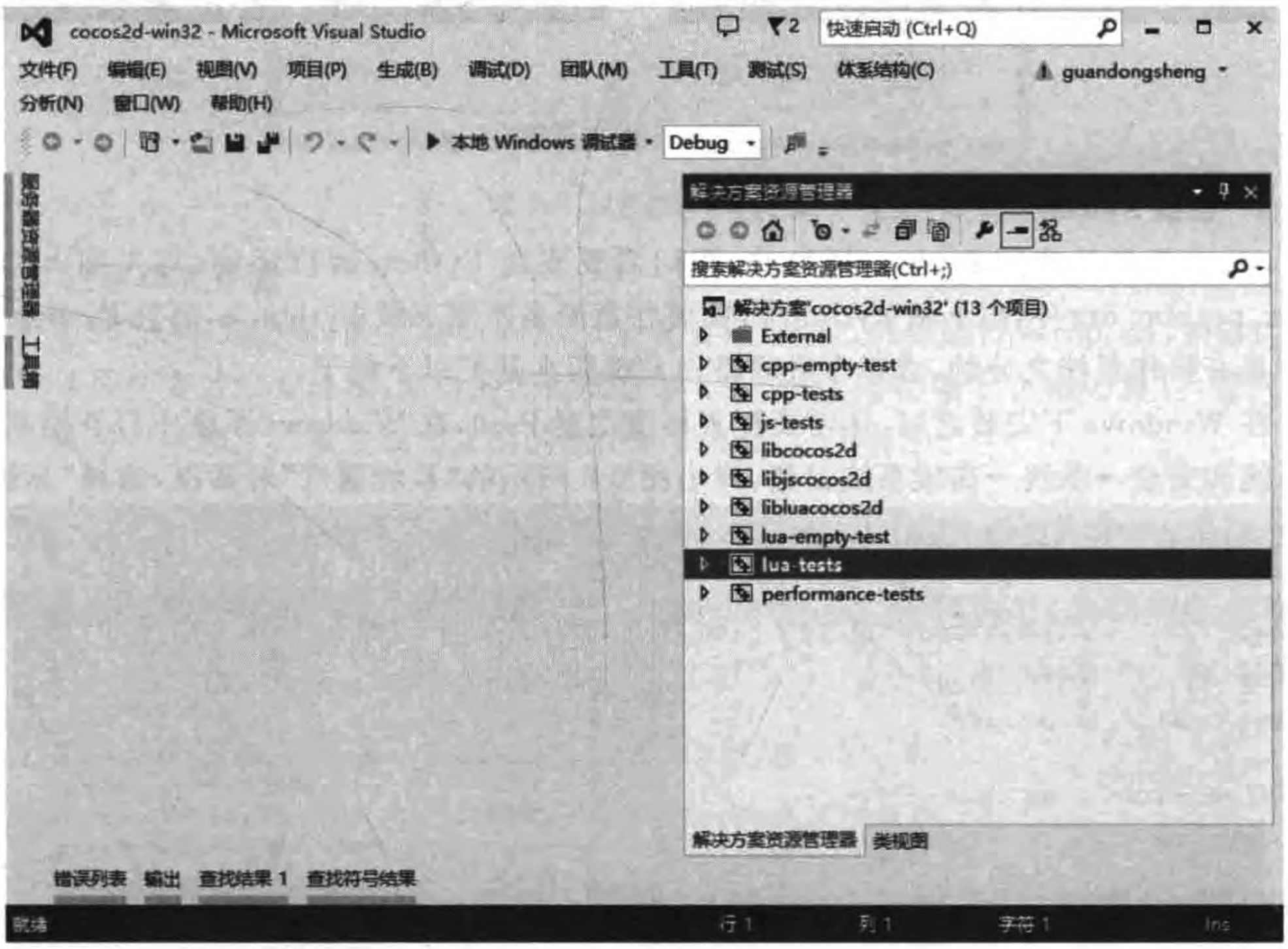


图 3-6　Cocos2d-x 案例

首次运行编译 Cocos2d-x 需要的时间会长一些，运行起来会出现一个 Windows 的窗口，如图 3-7(a)所示，选择其中的一个菜单项可以运行相应的示例，如图 3-7(b)所示。

如果想查看 lua-tests 源代码，不能通过 Visual Studio 查看，需要到< Cocos2d-x 安装根目录>\tests\lua-tests\src 文件夹下，使用文本编辑工具打开查看。

3.4.3　配置 Cocos2d-x 环境

配置 Cocos2d-x 环境可以帮助设置一些环境变量，通过在终端中执行 cocos 命令实现。cocos 命令是 Cocos2d-x 提供的工具，可以帮助创建、编译、打包工程等。

(a) (b)

图 3-7 运行案例

1. 安装 Python

cocos 工具是使用 Python 开发的，我们需要安装 Python 运行环境，首先到 https://www.python.org/网站下载 Python 2，需要注意的是不要下载 Python 3，而且 Python 安装文件是有操作系统之分的，选择合适的操作系统版本就可以下载了。

在 Windows 下安装之后，还需要配置环境变量 Path，在 Windows 系统中打开控制面板→系统和安全→系统→高级系统设置，弹出图 3-8 所示的“系统属性”对话框，选择“高级”标签，然后单击“环境变量”按钮，弹出图 3-9 所示的“环境变量”对话框。

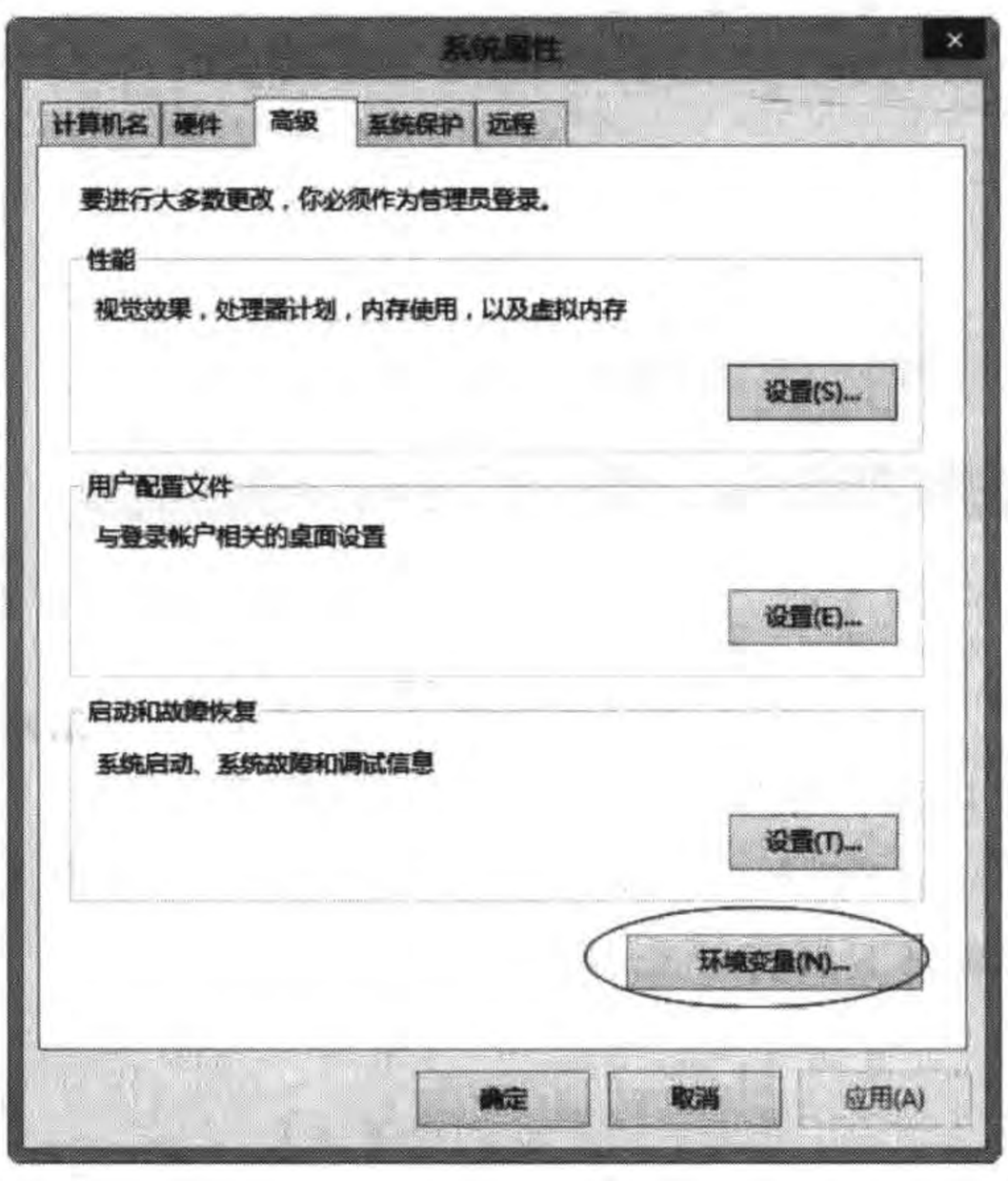


图 3-8 “系统属性”对话框

图 3-9 “环境变量”对话框

在“环境变量”对话框中设置 Path 变量，Path 变量在用户变量（上半部分，只影响当前用户）和系统变量（下半部分，影响所有用户）都有，读者可以根据自己的喜好决定在哪里设置。本例是设置的系统变量，双击系统变量中 Path 变量，弹出图 3-10 所示的对话框，在 Path 后面追加“C:\Python27”，注意两个 Path 之间要用英文分号“;”分隔。

图 3-10　设置 Path 变量

2. 设置环境变量

开启到 DOS 等终端，进入到<Cocos2d-x 安装根目录>，然后运行 setup.py，如果这个过程中发现哪些变量没有设置，窗口的光标就会停止在那儿，等待输入正确的路径，输入完成后回车就可以继续了。设置成功之后的输出结果如图 3-11 所示。

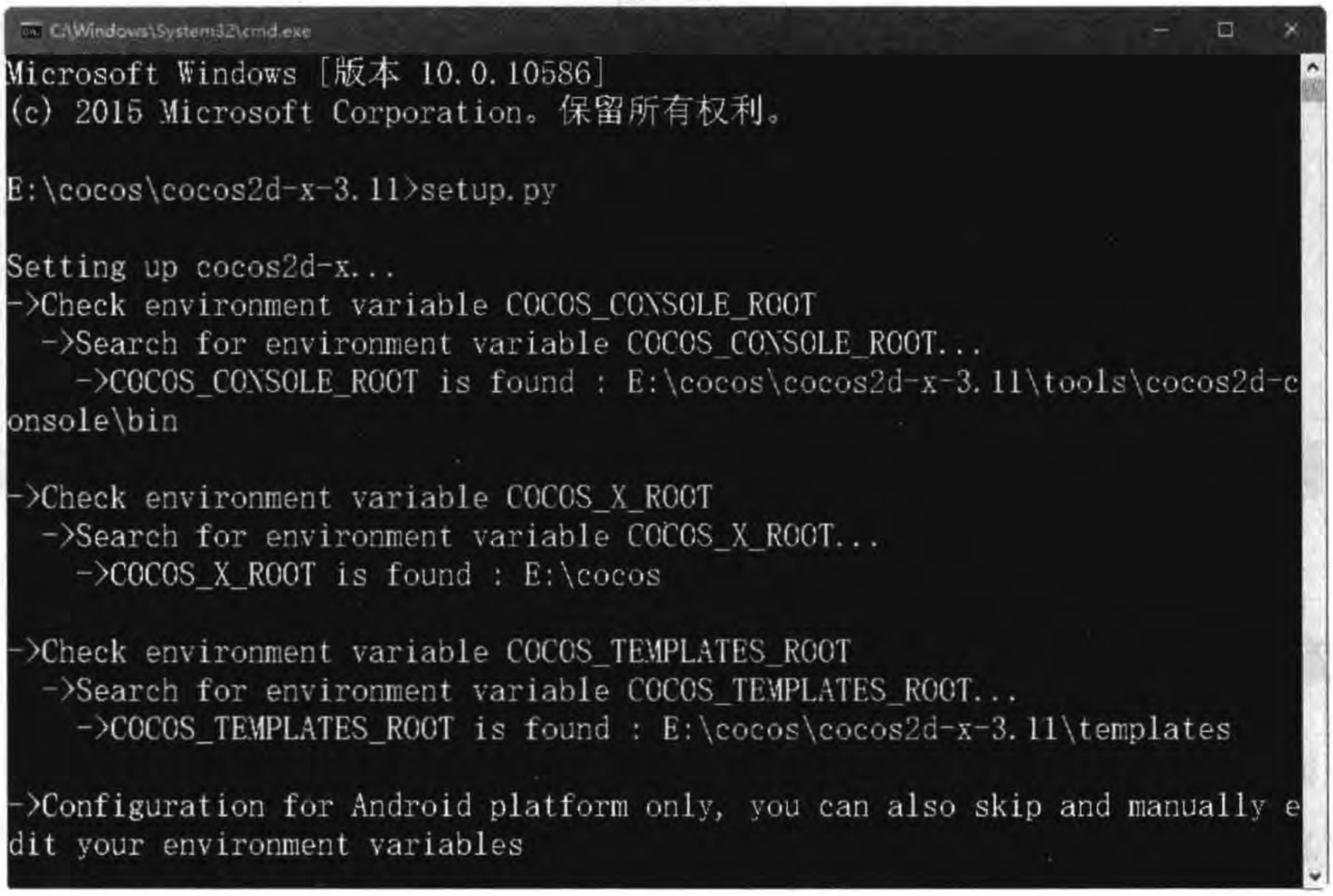


图 3-11　设置环境变量

提示 设置过程中有关 Android 的设置，如 NDK_ROOT、Android_SDK_ROOT 等先不用设置。会在 Android 平台移植的时候再介绍。

3.4.4 使用 API 文档

从 Cocos2d-x Lua API 官方下载的开发包中没有 API 文档，可以使用 Cocos2d-x C++ API 官方的在线 API 文档。通过 http://www.cocos2d-x.org/docs/api-ref/选择 Cocos2d-x C++版本进入在线 API 文档，如图 3-12 所示。

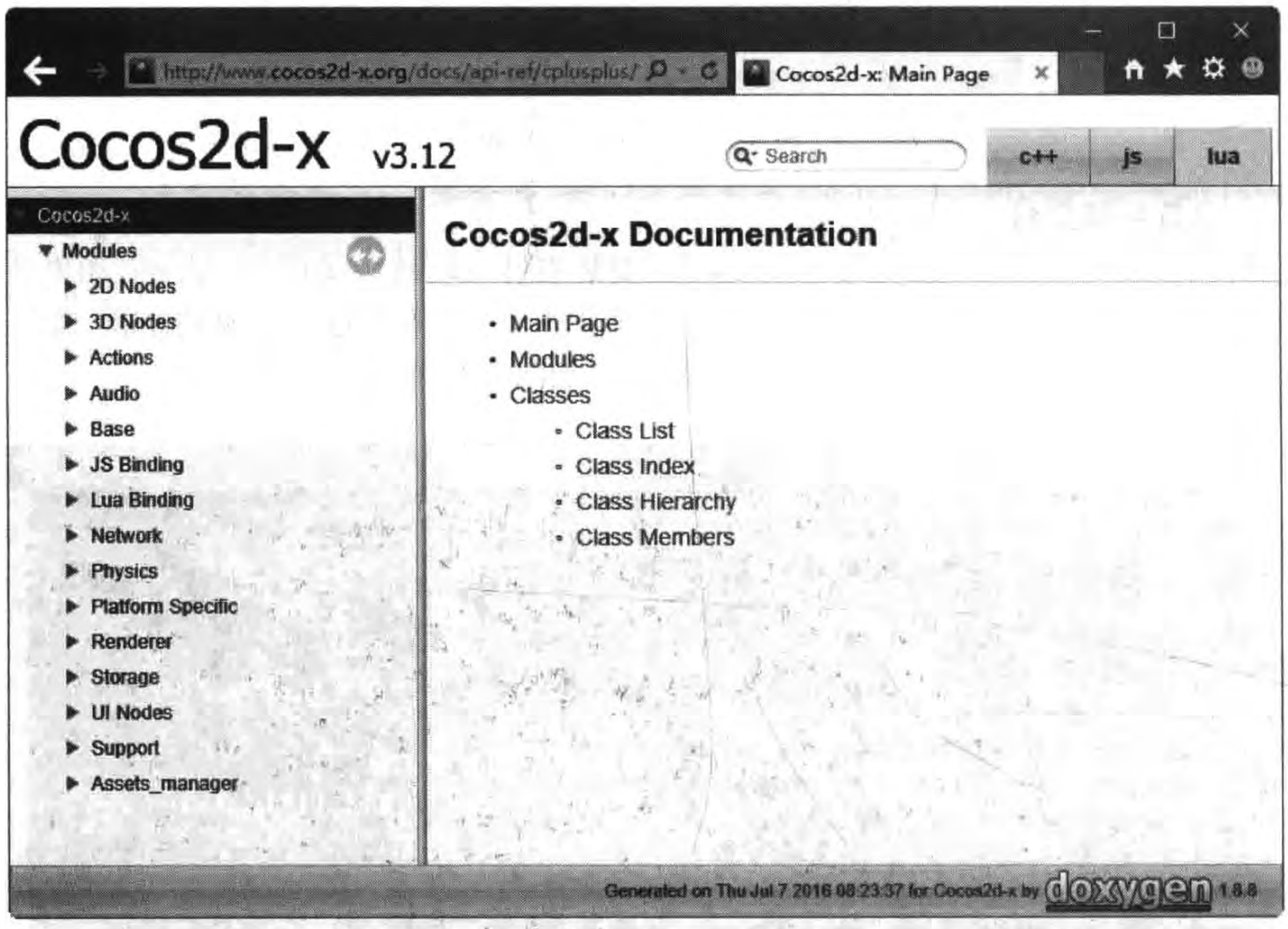


图 3-12 Cocos2d-x 在线文档

本章小结

通过对本章的学习，读者可以了解移动平台游戏引擎，Cocos2d 家谱和 Cocos2d-x 设计目标，还可以学习到 Cocos2d-x Lua API 开发环境的搭建。

第 4 章

Hello Cocos2d-x

第 3 章介绍了 Cocos2d-x Lua API 开发环境，本章从一个 HelloLua 入手，介绍 Cocos2d-x Lua API 的基本开发流程以及 Cocos2d-x 生命周期和 Cocos2d-x 核心知识体系。

4.1 第一个 Cocos2d-x Lua API 游戏

编写的第一个 Cocos2d-x Lua API 程序，命名为 HelloLua，HelloLua 运行界面如图 4-1 所示，界面中有一个 Hello World 标签和一个图片精灵。

图 4-1 HelloLua 运行结果

4.1.1 创建工程

创建 Cocos2d-x Lua API 工程可以通过 Cocos2d-x 提供的 cocos new 命令实现，创建完成的工程可以直接使用文本工具编写 Lua 代码，然后再通过 cocos run 命令运行。

配置好环境后，能够通过 DOS 等终端进入 cocos 目录(< Cocos2d-x 安装根目录>\tools\

cocos2d-console\bin),然后在 DOS 等终端中执行如下指令:

```
cocos new HelloLua - p com.work6 - l lua - d D:/projects
```

其中,D:/projects 为 HelloLua 的工程生成文件夹。通过上面的指令在 D:/projects 文件夹下面生成了名为 HelloLua 的 Cocos2d-x Lua API 工程。

4.1.2 工程文件结构

打开 HelloLua 工程根目录,内容如图 4-2 所示。其中 res 文件夹存放资源文件,src 文件夹存放主要的程序代码。

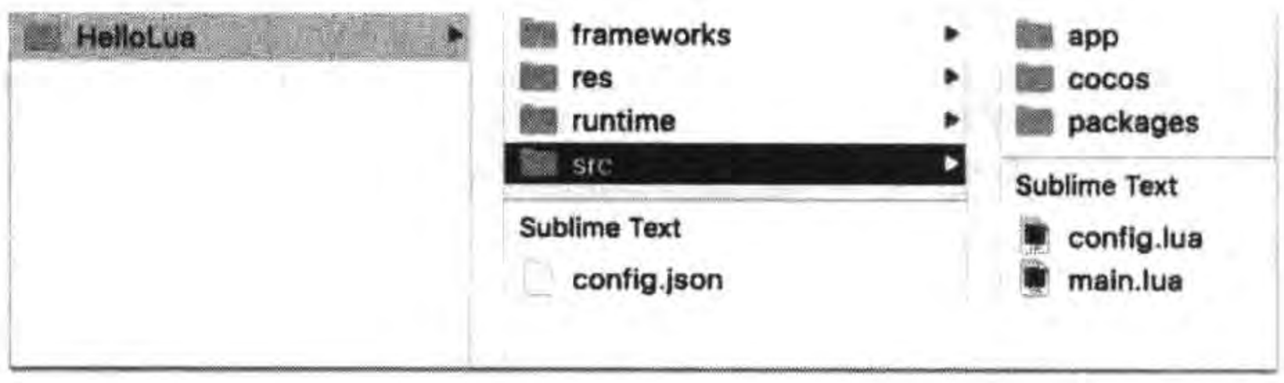


图 4-2 HelloLua 工程中的内容

4.1.3 重构 HelloLua 工程

在 4.1.2 节介绍的 HelloLua 工程中场景是通过 Cocos Studio 工具创建和设计的,由于 Cocos Studio 的内容超出了本书介绍范围,故不使用 Cocos Studio 工具创建场景。另外,HelloLua 工程的架构采用 MVC 设计模式,这个 MVC 设计模式的架构是基于 quick-cocos2d-x 框架,本书不采用 quick-cocos2d-x 框架,因此需要重构 HelloLua 工程,并把 HelloLua 工程作为本书的工程模板。

注意 quick-cocos2d-x 框架是由触控科技成都团队开发的,它在 Cocos2d-x Lua API 绑定基础上封装,虽然经过封装开发变得更加简单,但失去了 Cocos2d 一贯的 API 特征,使得熟悉 Cocos2d 其他 API 的人不容易上手 quick-cocos2d-x。因此本书不采用 quick-cocos2d-x 框架。

首先,删除 HelloLua 工程 src 目录下的一些文件和文件夹,包括 app 文件夹、packages 文件夹和 config.lua 文件,其他的文件和文件夹不变,如图 4-3 所示。

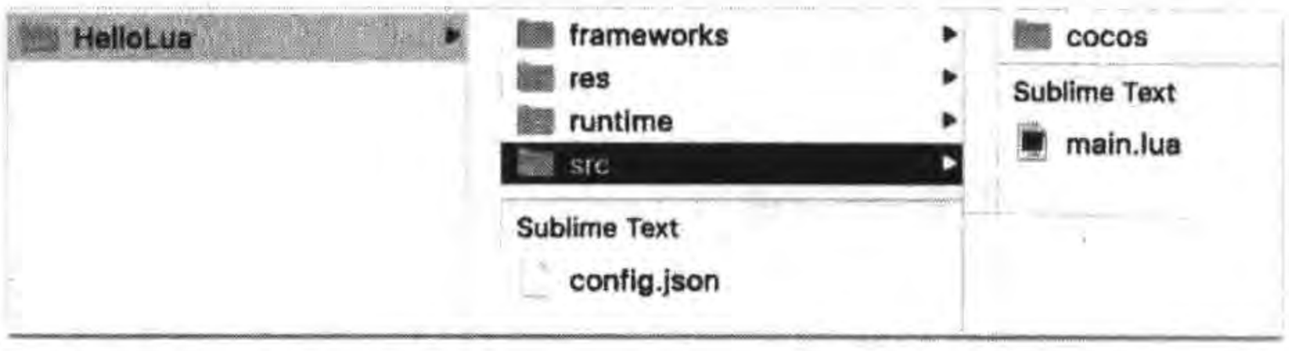


图 4-3 重构 HelloLua 工程中的文件结构

修改 main.lua 主要代码如下：

```
require "cocos.init"

-- cclog
cclog = function(...)                                    ①
    print(string.format(...))
end

local function main()
    collectgarbage("collect")
    -- avoid memory leak
    collectgarbage("setpause", 100)
    collectgarbage("setstepmul", 5000)

    cc.FileUtils:getInstance():addSearchPath("src")
    cc.FileUtils:getInstance():addSearchPath("res")

    local director = cc.Director:getInstance()
    director:getOpenGLView():setDesignResolutionSize(960, 640, 0)

    --设置是否显示帧率和精灵个数
    director:setDisplayStats(true)

    --设置帧率
    director:setAnimationInterval(1.0 / 60)

    --创建场景
    local scene = require("GameScene")
    local gameScene = scene.create()

    if cc.Director:getInstance():getRunningScene() then
        cc.Director:getInstance():replaceScene(gameScene)
    else
        cc.Director:getInstance():runWithScene(gameScene)
    end

end

local status, msg = xpcall(main, __G__TRACKBACK__)
if not status then
    error(msg)
end
```

上述第①行代码声明 cclog 日志函数，它原来是 local 修饰，这里我们删除了 local，这是说明该函数是全局函数，可以在其他模块中使用 cclog 函数。cclog 函数内容很简单，就是封装了 Lua 提供的 print 函数，可以把 cclog 函数当成 print 函数别名。声明 cclog 函数的目的是为了模仿 Cocos2d-x C++API 中 cc.log 函数，这样使得熟悉 Cocos2d-x C++API 的开发人员能够快速熟悉 Cocos2d-x Lua API。

然后需要在工程中添加 GameScene.lua，并添加如下代码：

```
local size = cc.Director:getInstance():getWinSize()
```

```
local GameScene = class("GameScene",function()
    return cc.Scene:create()
end)

function GameScene.create()
    local scene = GameScene.new()
    scene:addChild(scene:createLayer())                                    ①
    return scene
end

function GameScene:ctor()

end

--创建层
function GameScene:createLayer()                                           ②
    cclog("GameScene init")
    local layer = cc.Layer:create()

    local label = cc.Label:createWithSystemFont("HelloWorld", "Arial", 36) ③
    label:setPosition(cc.p(size.width/2, size.height - 100))
    layer:addChild(label)

    local sprite = cc.Sprite:create("HelloWorld.png")                      ④
    sprite:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(sprite)

    return layer
end

return GameScene
```

第①行代码是将创建的层添加到场景中。

第②行代码是声明创建成函数。

第③行代码是创建标签对象。

第④行代码是创建精灵对象。

4.1.4 运行 HelloLua 工程

Cocos2d-x Lua API 工程通过 Lua 绑定技术可以在本地平台下运行，包括 Win32、Linux、Android、iOS、Mac OS X、Windows 和 Windows Phone 等平台。可以通过两种方式运行 Cocos2d-x Lua API 工程：一种是通过相应的 IDE 工具编译并运行；另一种是通过 cocos run 命令运行。

1. 通过 IDE 工具编译并运行

打开< HelloLua 工程根目录>\frameworks\runtime-src 文件夹，如图 4-4 所示，文件夹中 proj 开头的文件夹内部，就是某个平台 IDE 工程(或者是解决方案)。其中，proj. win32 和 proj. win8. 1-universal 中是 Visual Studio 解决方案，需要安装 Visual Studio 工具。

proj. ios_mac 中是 Xcode 工程文件，需要在 Mac OS X 下安装 Xcode 才可以打开运行。

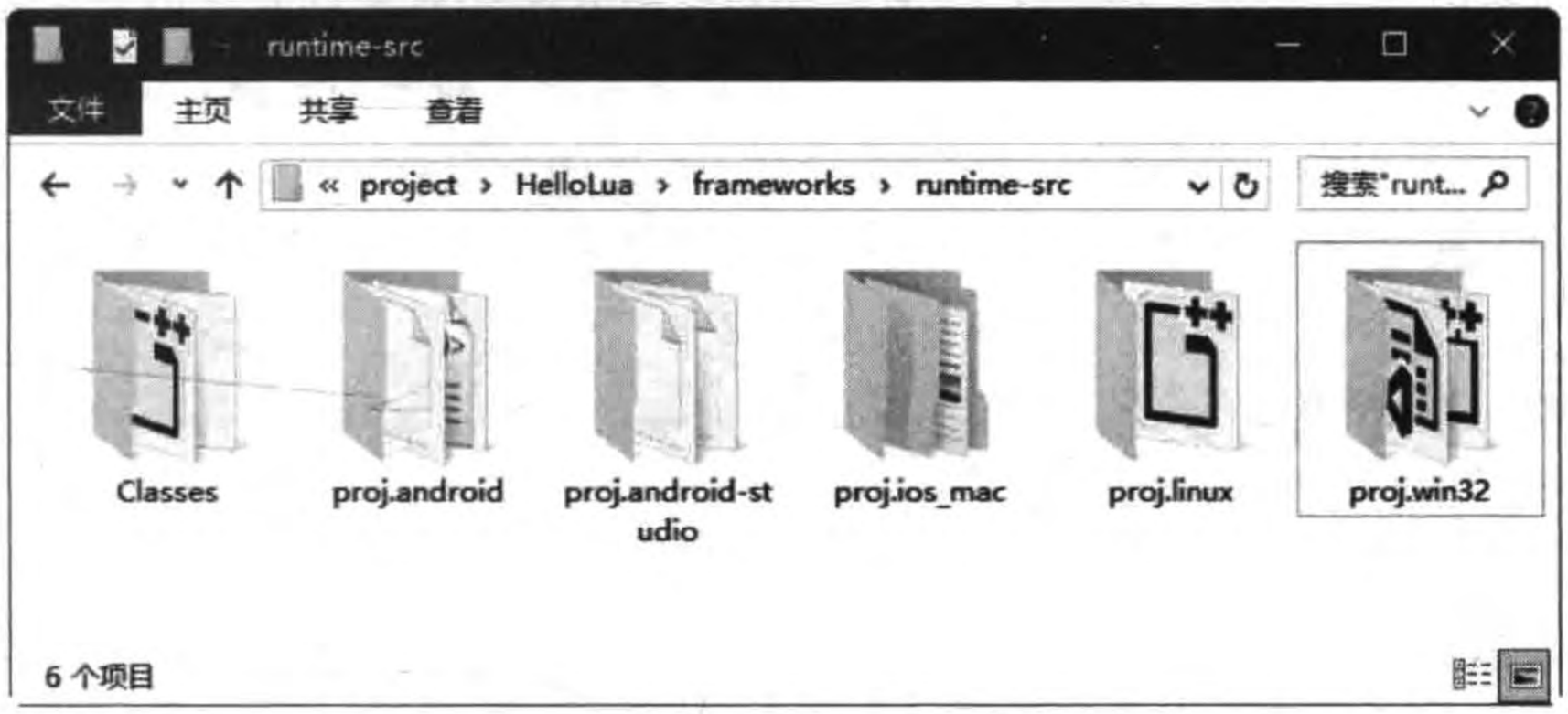


图 4-4 <HelloLua 工程根目录>\frameworks\runtime-src 文件夹

如果在 Windows 下进行开发可以打开 proj. win32 文件夹中的 HelloLua. sln 解决方案文件，如图 4-5 所示，不需要任何修改和设置，直接单击工具栏中的运行调试按钮▶，则可运行工程。

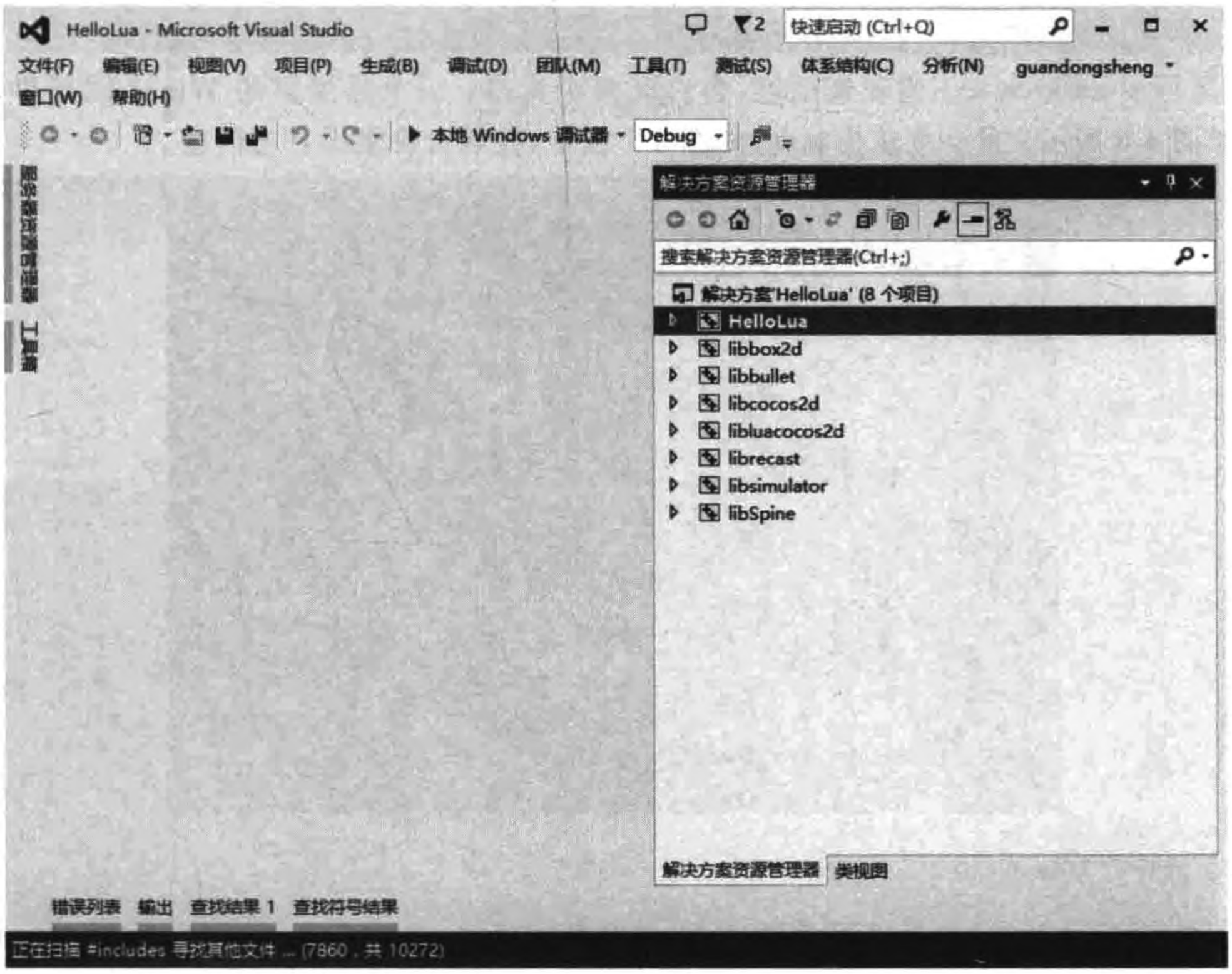


图 4-5 打开 proj. win32 文件夹中的 HelloLua. sln 解决方案文件

2. 通过 cocos run 命令运行

通过 IDE 工具编译并运行方式,需要安装相应的 IDE 工具软件并且由 IDE 启动运行。而事实上开发人员不能使用这些 IDE 工具调试 Lua 程序。如果不想通过 IDE 工具启动运行游戏工程,则可以选择使用 Cocos2d-x 提供的 cocos run 命令。

首先通过 DOS 等终端进入< HelloLua 工程根目录>,然后在 DOS 等终端中执行如下指令:

```
cocos run HelloLua -p win32
```

4.1.5 调试 HelloLua 工程

使用前面介绍的 IDE 工具不能调试 Lua 程序,更不能在 Lua 程序中设置断点,但 cc.log 函数可以将日志信息输出到 IDE 控制台中。而使用 cocos run 指令 cc.log 函数不能输出日志信息,也不能改变模拟器大小,每次修改都需要重新运行,很麻烦。

Cocos2d-x 本身提供一个非常灵活的模拟器工具,它位于< Cocos2d-x 安装根目录>\tools\simulator\frameworks\runtime-src\proj.win32 文件夹中,在安装了 Visual Studio 工具后运行 simulator.sln 解决方案,并编译。然后在< Cocos2d-x 安装根目录>\tools\simulator\runtime\win32 文件夹中可以看到编译获得的 simulator.exe。为了使用方便可以在桌面上创建 simulator.exe 的快捷图标。

运行 simulator.exe 启动模拟器,会启动两个窗口:一个是黑屏的 Win32 图形界面窗口,如图 4-6 所示;另一个是控制台,如图 4-7 所示,控制台会输出日志信息。

图 4-6 Win32 图形界面窗口

simulator.exe 模拟器图像界面窗口中有很多菜单,单击右上角的 Cocos 图标,打开图 4-8 所示的文件菜单,通过该菜单可以动态运行 Cocos2d-x Lua API 或 Cocos2d-x Lua

API 游戏工程。有时需要调整模拟器窗口大小，可以通过图 4-9 所示的视图菜单，动态改变模拟器窗口。另外，一个工程如果修改了其中的文件，也不必重新启动模拟器，按 F5 键重新加载文件即可。

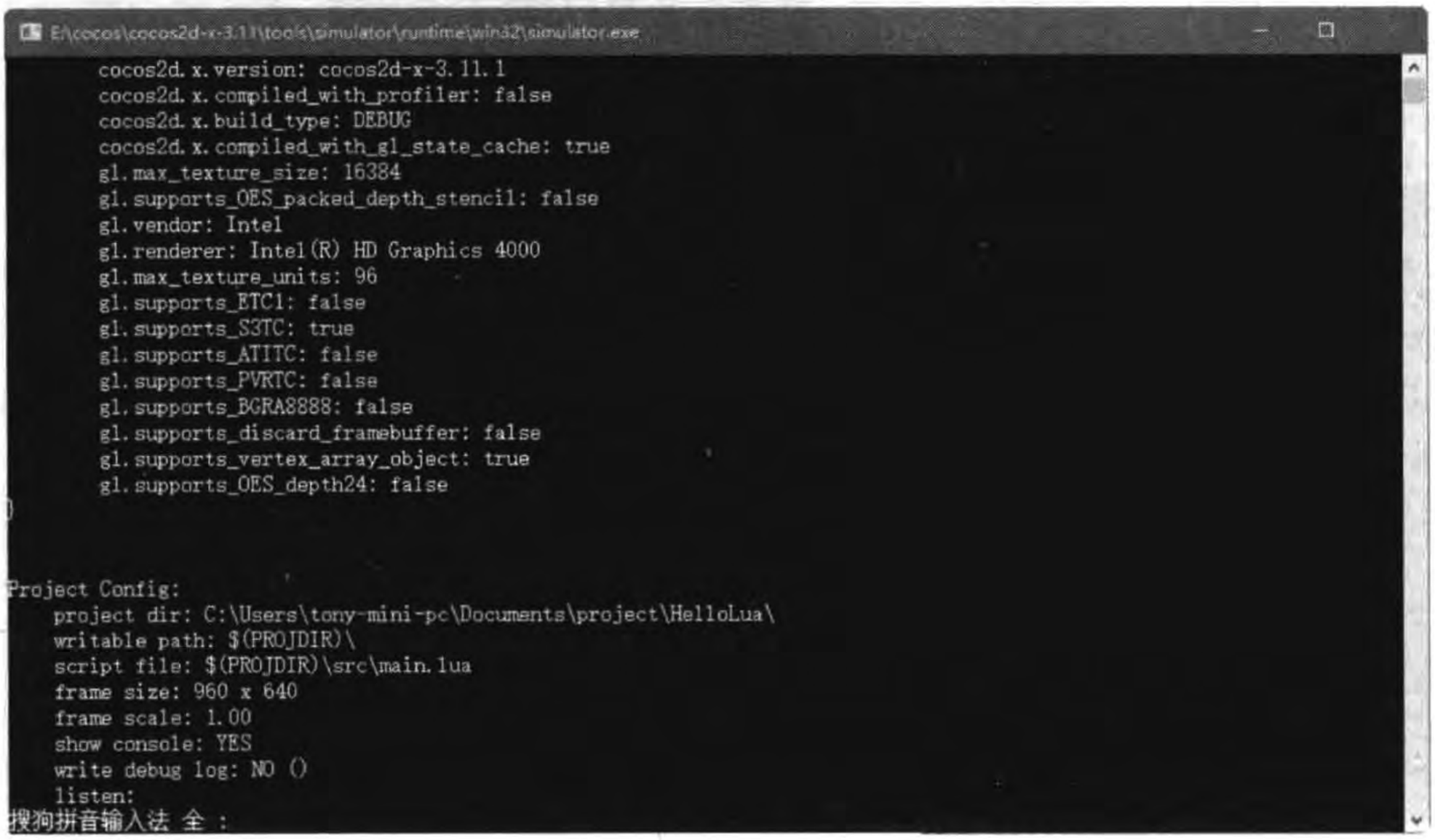


图 4-7 控制台

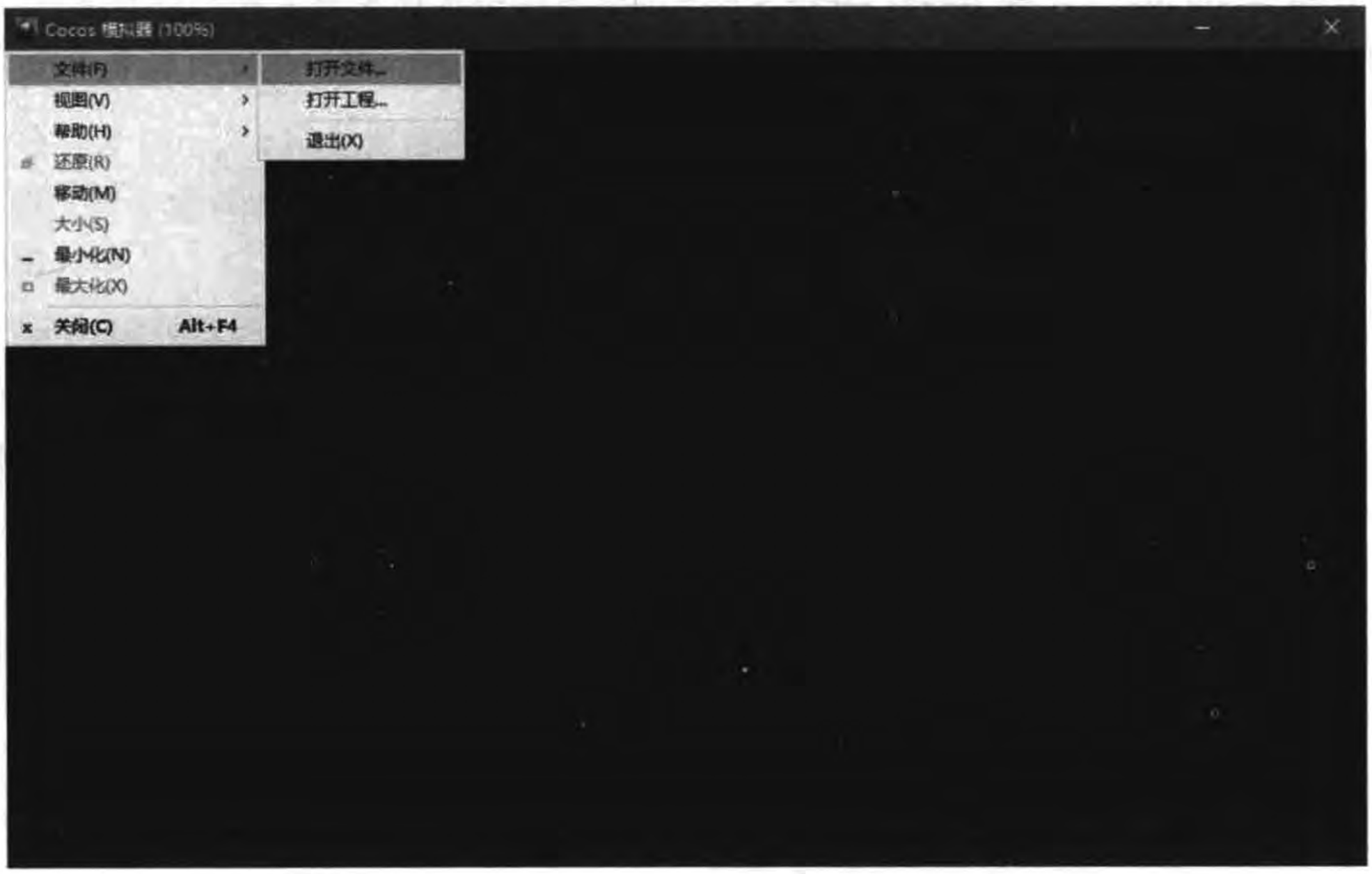


图 4-8 模拟器文件菜单

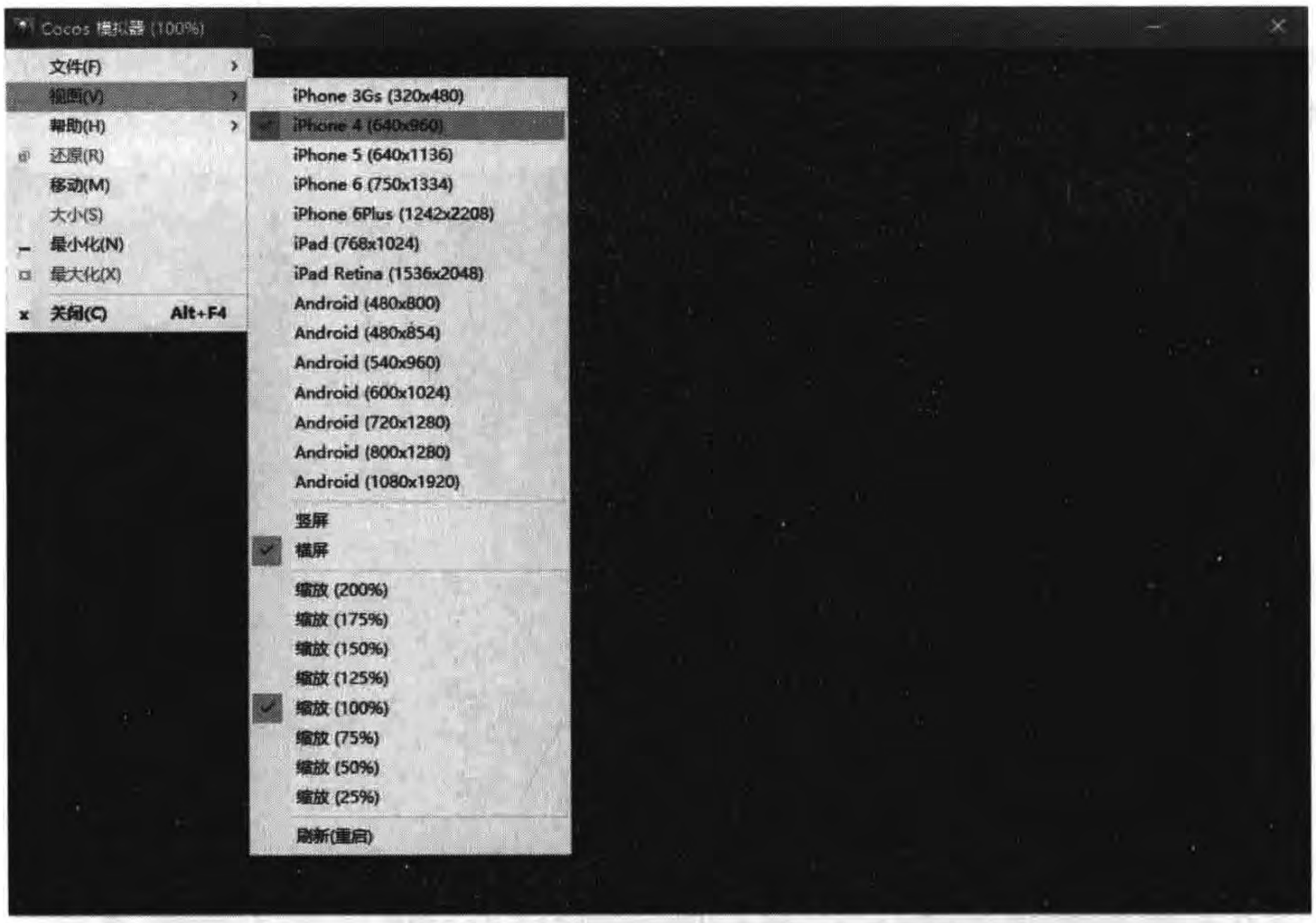


图 4-9　改变模拟器视图大小

使用 simulator.exe 模拟器时，在图 4-8 所示的界面中选择菜单文件→打开文件，在弹出对话框中查到< HelloLua 工程根目录>下的 config.json 文件，就可以运行游戏工程了。

config.json 文件主要内容如下：

```
{
  "init_cfg": {                                                    ①
        "isLandscape": true,                                       ②
        "name": "HelloLua",                                        ③
        "width": 960,                                              ④
        "height": 640,                                             ⑤
        "entry": "src/main.lua",                                   ⑥
        "consolePort": 6050,
        "uploadPort": 6060
    }
  ⋮
}
```

上述第①行代码是初始配置信息。

第②行代码是设置横屏显示还是竖屏显示。

第③行代码 name 属性是设置模拟器上显示的标题。

第④和第⑤行代码是设置屏幕的宽和高。

第⑥行代码是设置入口文件。

4.2 Cocos2d-x 核心概念

Cocos2d-x Lua API 中有很多概念，这些概念很多来源于动画、动漫和电影等行业，例如导演、场景和层等概念，当然也有些传统的游戏的概念。Cocos2d-x Lua API 中核心概念如下：

(1) 导演；
(2) 场景；
(3) 层；
(4) 节点；
(5) 精灵；
(6) 菜单；
(7) 动作；
(8) 效果；
(9) 粒子运动；
(10) 地图；
(11) 物理引擎。

本节介绍导演、场景、层、精灵、菜单以及对应的类，节点的概念将会在 4.3 节详细介绍；其他概念在后面的章节中介绍。

4.2.1 导演

导演类 Director 用于管理场景对象，采用单例设计模式，在整个工程中只有一个实例对象。由于是单例模式能够保存一致的配置信息，便于管理场景对象。获得导演类 Director 实例语句如下：

```
local director = cc.Director:getInstance()
```

其中 cc 是 Cocos2d-x Lua API 中类的命名空间，Director 是导演类，getInstance()函数用于获得调用实例。

导演对象职责如下：

(1) 访问和改变场景。
(2) 访问配置信息。
(3) 暂停、继续和停止游戏。
(4) 转换坐标。

4.2.2 场景

场景类 Scene 是构成游戏的界面，类似于电影中的场景。场景大致可以分为以下几类：

(1) 展示类场景。播放视频或在图像上输出简单的文字,来实现游戏的开场介绍、胜利和失败提示、帮助介绍。

(2) 选项类场景。主菜单、设置游戏参数等。

(3) 游戏场景。这是游戏的主要内容。

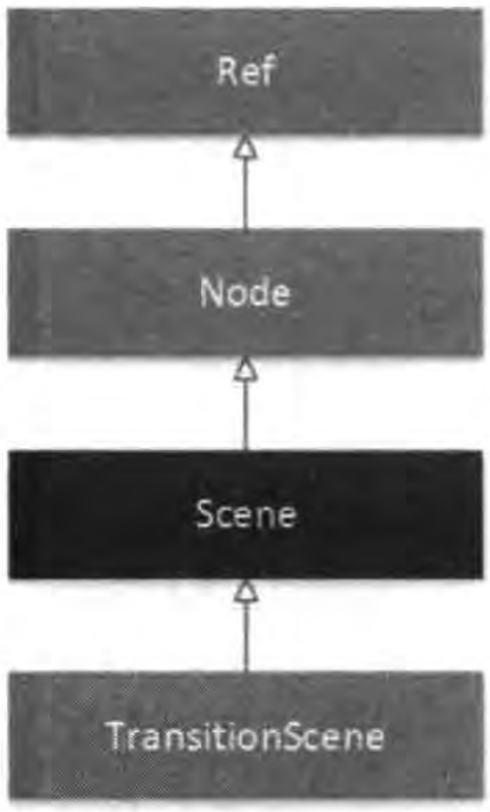


图 4-10 Scene 类图

场景类 Scene 的类图如图 4-10 所示,从类图可见 Scene 继承了 Node 类,Node 是一个重要的类,很多类都从 Node 类派生而来,其中有 Scene、Layer 等。

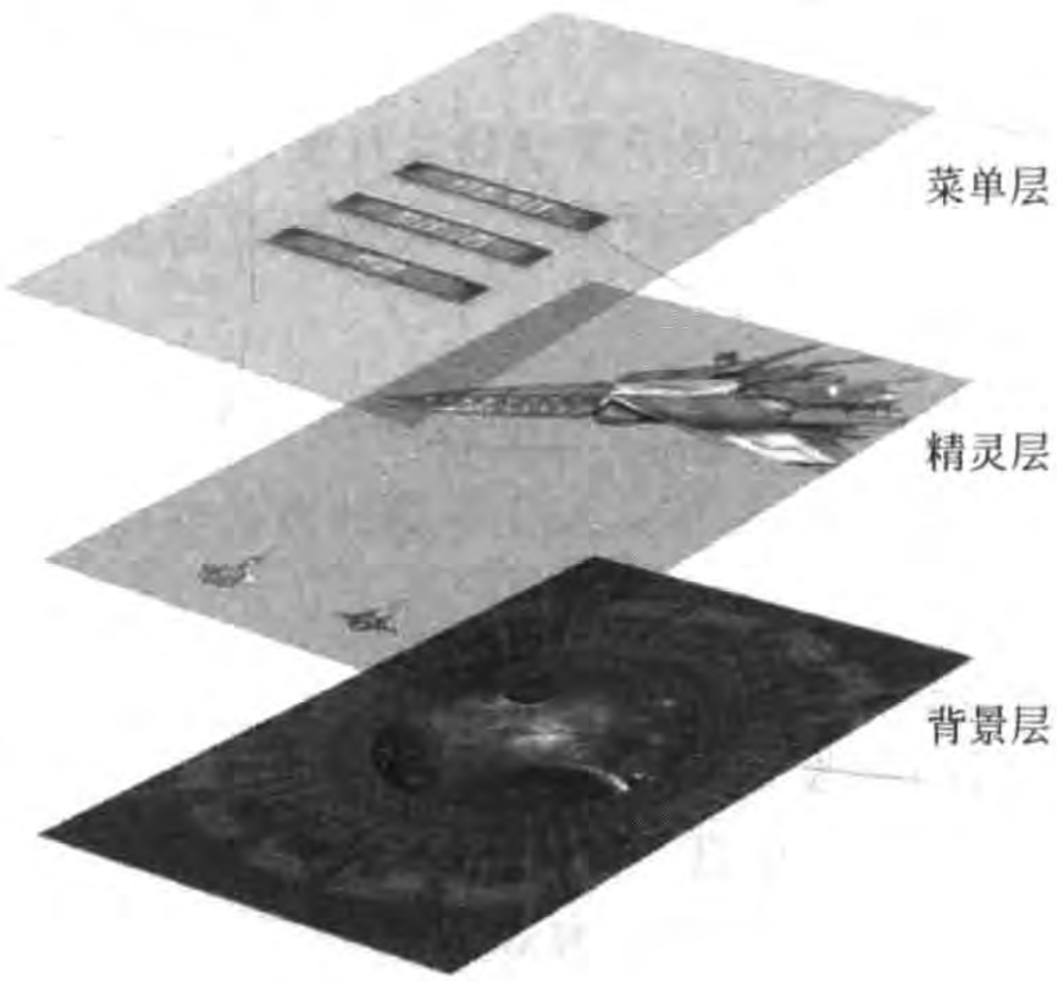
4.2.3 层

层是写游戏的重点,大约 99%以上的时间是在层上实现游戏内容。层的管理类似于 Photoshop 中的图层,它也是一层一层叠在一起。图 4-11(a)是一个简单的主菜单界面,它是由三个层叠加实现的,如图 4-11(b)所示。

为了让不同的层可以组合产生统一的效果,这些层基本上都是透明或者半透明的。层的叠加是有顺序的,如图 4-11(b)所示从上到下依次是菜单层→精灵层→背景层。Cocos2d-x Lua API 是按照这个次序来叠加界面的。这个次序同样用于事件响应机制,即菜单层最先接收到系统事件,然后是精灵层,最后是背景层。在事件的传递过程中,如果有一个层处理了该事件,则排在后面的层就不再接收该事件了。每一层又可以包括很多各式各样的内容要素,如文本、链接、精灵、地图等内容。

主菜单画面

(a)



(b)

图 4-11 层叠加

层类 Layer 的类图如图 4-12 所示。

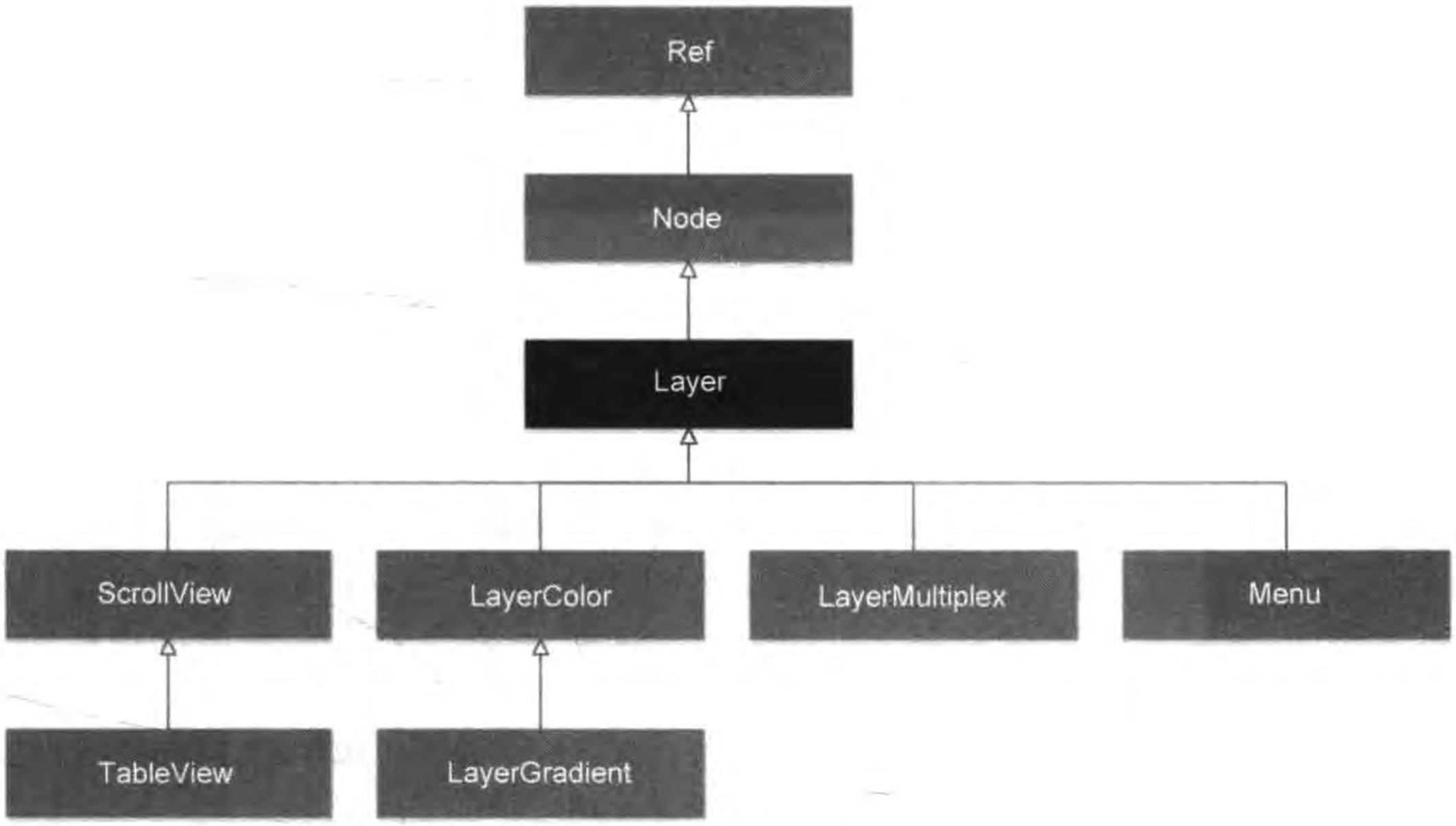


图 4-12 Layer 类图

4.3 Node 与 Node 层级架构

Cocos2d-x Lua API 采用层级(树状)结构管理场景、层、精灵、菜单、文本、地图和粒子系统等节点(Node)对象。一个场景包含了多个层,一个层又包含多个精灵、菜单、文本、地图和粒子系统等对象。层级结构中的节点可以是场景、层、精灵、菜单、文本、地图和粒子系统等任何对象。

节点的层级结构如图 4-13 所示。

这些节点有一个共同的父类 Node,Node 类图如图 4-14 所示。Node 类是 Cocos2d-x 最为重要的根类,它是场景、层、精灵、菜单、文本、地图和粒子系统等类的根类。

4.3.1 Node 中重要的操作

Node 作为根类有很多重要的函数,下面分别介绍:

(1) 创建节点。local childNode=cc. Node:create()。

(2) 增加新的子节点。node:addChild (childNode, 0, 123),第 2 个参数是 Z 轴绘制顺序,第 3 个参数是标签。

(3) 查找子节点。local node=node:getChildByTag(123),通过标签查找子节点。

(4) node:removeChildByTag(123, true)通过标签删除子节点,并停止该节点上的一切

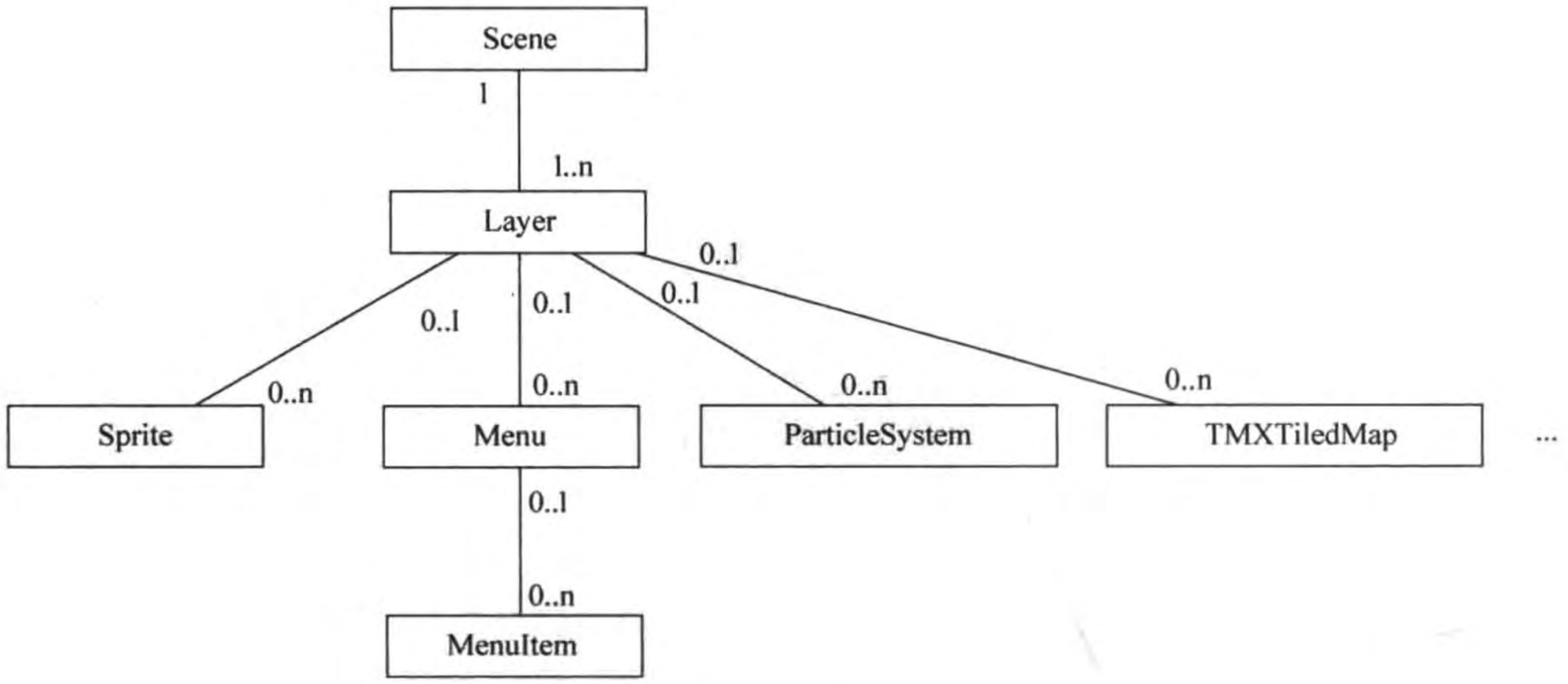


图 4-13　节点的层级结构

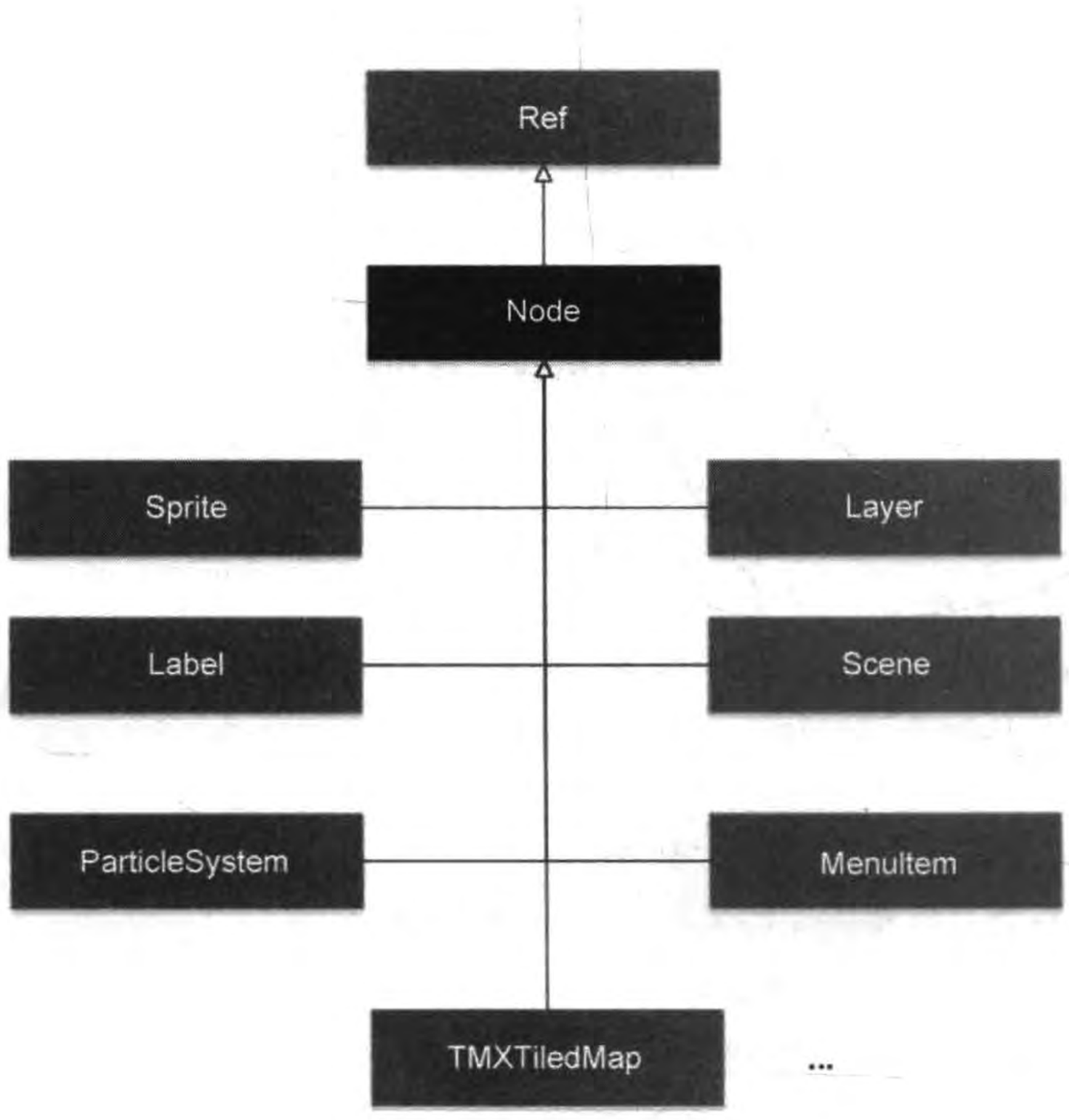


图 4-14　Node 类图

动作。

(5) node:removeChild(childNode, true)删除 childNode 节点,并停止该子节点上的一切动作。

(6) node:removeAllChildrenWithCleanup(true)删除 node 节点的所有子节点,并停止这些子节点上的一切动作。

(7) node:removeFromParentAndCleanup(true)从父节点删除 node 节点,并停止该节点上的一切动作。

4.3.2 Node 中重要的属性

此外,Node 还有两个非常重要的属性: position 和 anchorPoint。

position(位置)属性是 Node 对象的实际位置。position 属性往往还要配合使用 anchorPoint 属性,为了将一个 Node 对象(标准矩形图形)精准地放置在屏幕某一个位置,需要设置该矩形的 anchorPoint(锚点),anchorPoint 属性是相对于 position 的比例,anchorPoint 计算公式是(w1/w2, h1/h2),图 4-15 所示锚点位于节点对象矩形内,w1 是锚点到节点对象左下角的水平距离,w2 是节点对象宽度; h1 是锚点到节点对象左下角的垂直距离,h2 是节点对象高度。(w1/w2, h1/h2)计算结果为(0.5,0.5),所以 anchorPoint 为(0.5,0.5),anchorPoint 的默认值就是(0.5,0.5)。

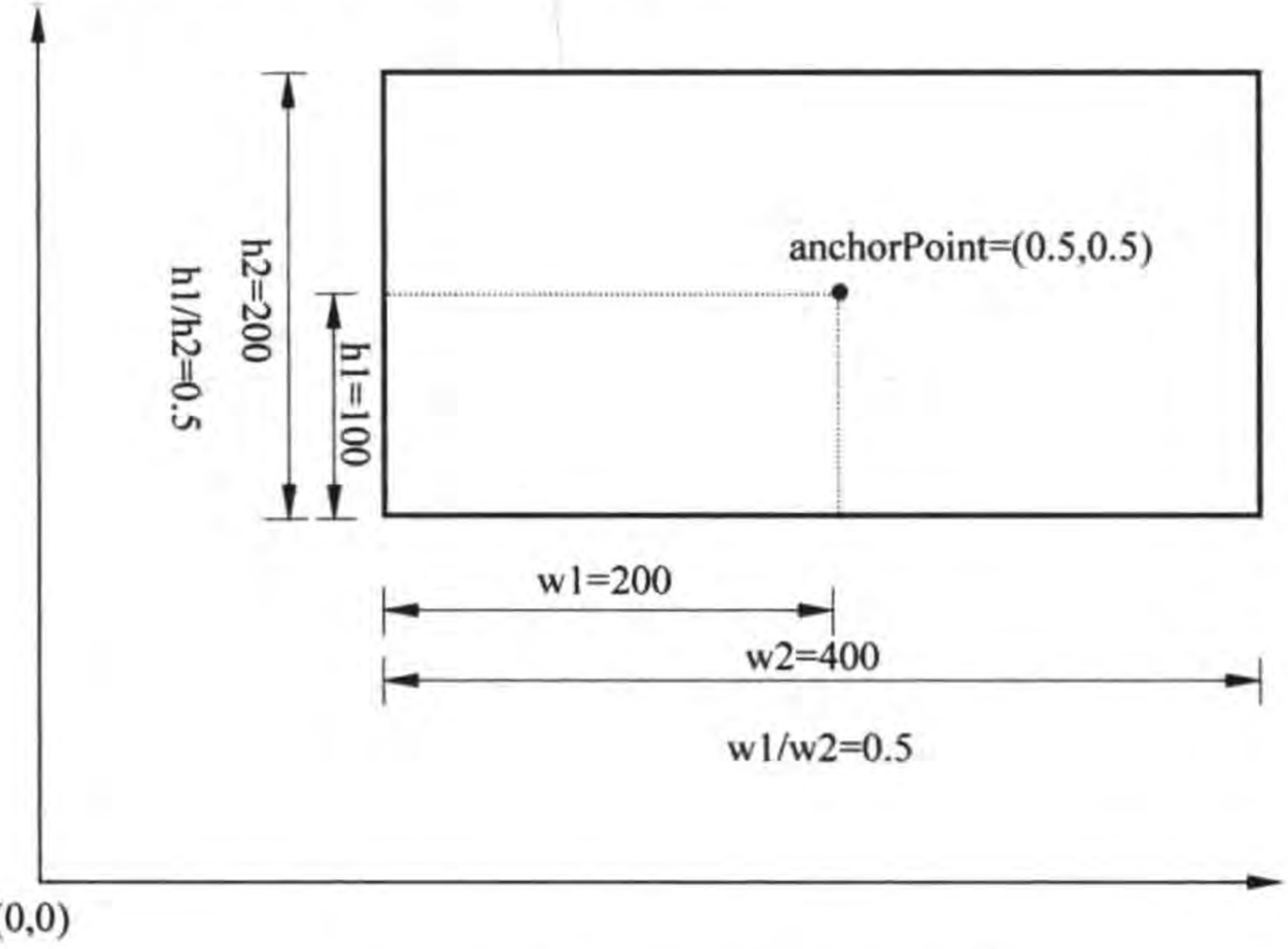


图 4-15 anchorPoint 为(0.5,0.5)

图 4-16 展示的是 anchorPoint 为(0.67, 0.5)的情况。

anchorPoint 还有两个极端值: 一个是锚点在节点对象矩形右上角,如图 4-17 所示,此时 anchorPoint 为(1,1); 另一个是锚点在节点对象矩形左下角,如图 4-18 所示,此时 anchorPoint 为(0,0)。

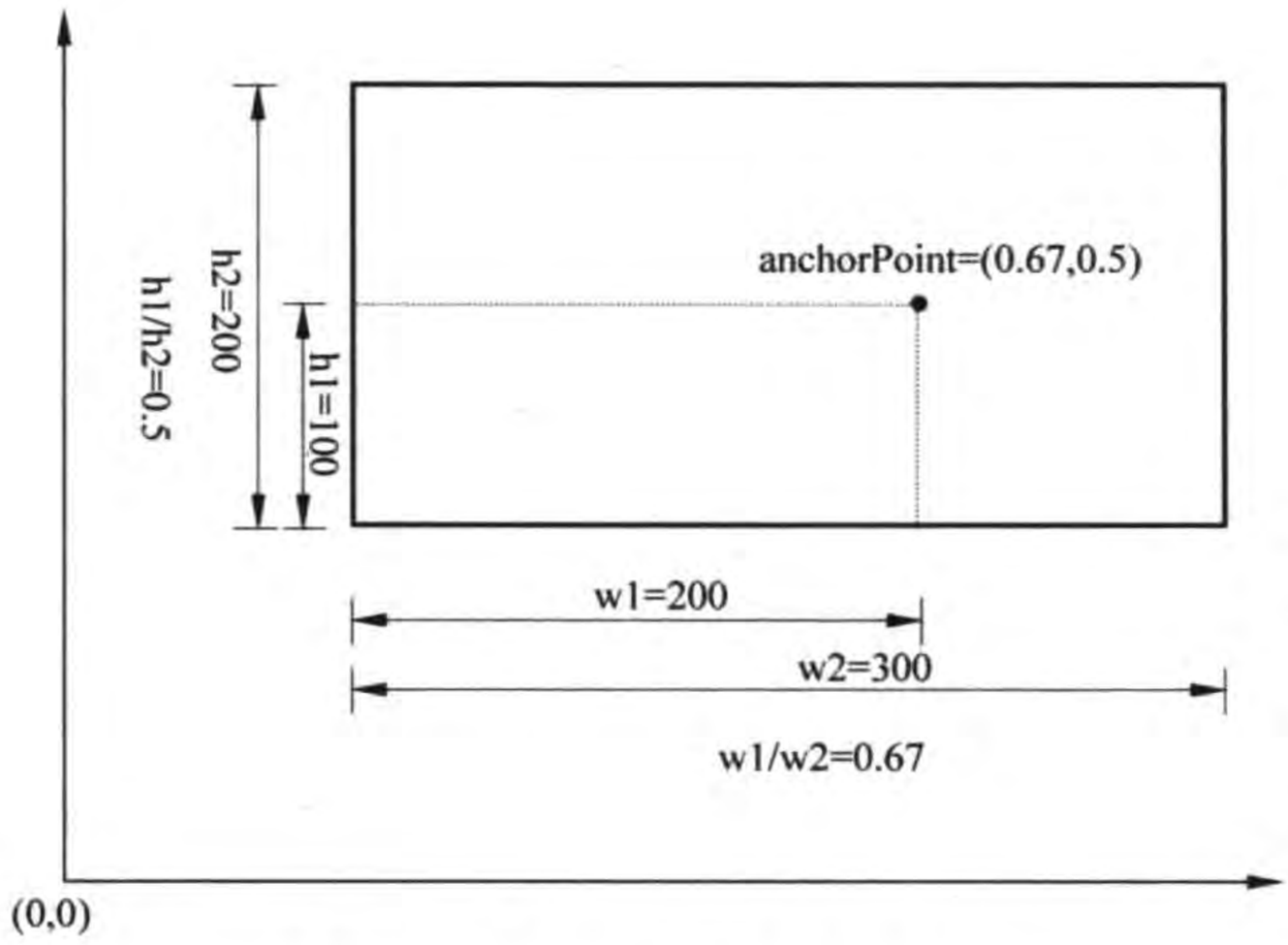


图 4-16　anchorPoint 为(0.67,0.5)

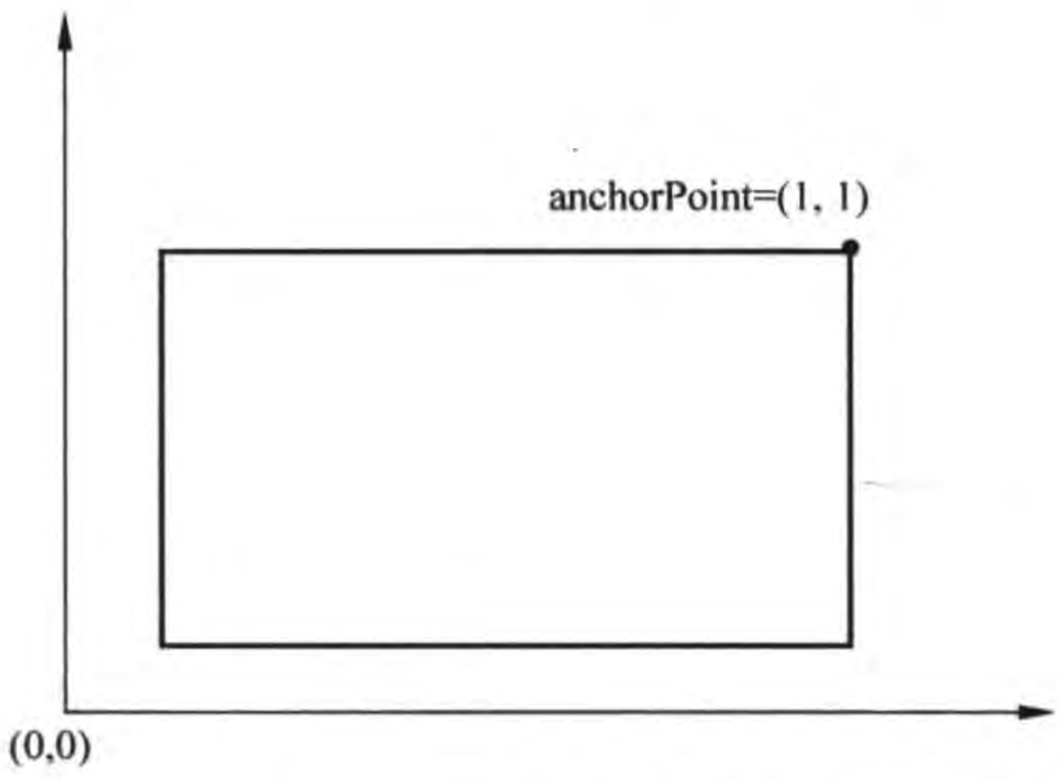


图 4-17　anchorPoint 为(1,1)

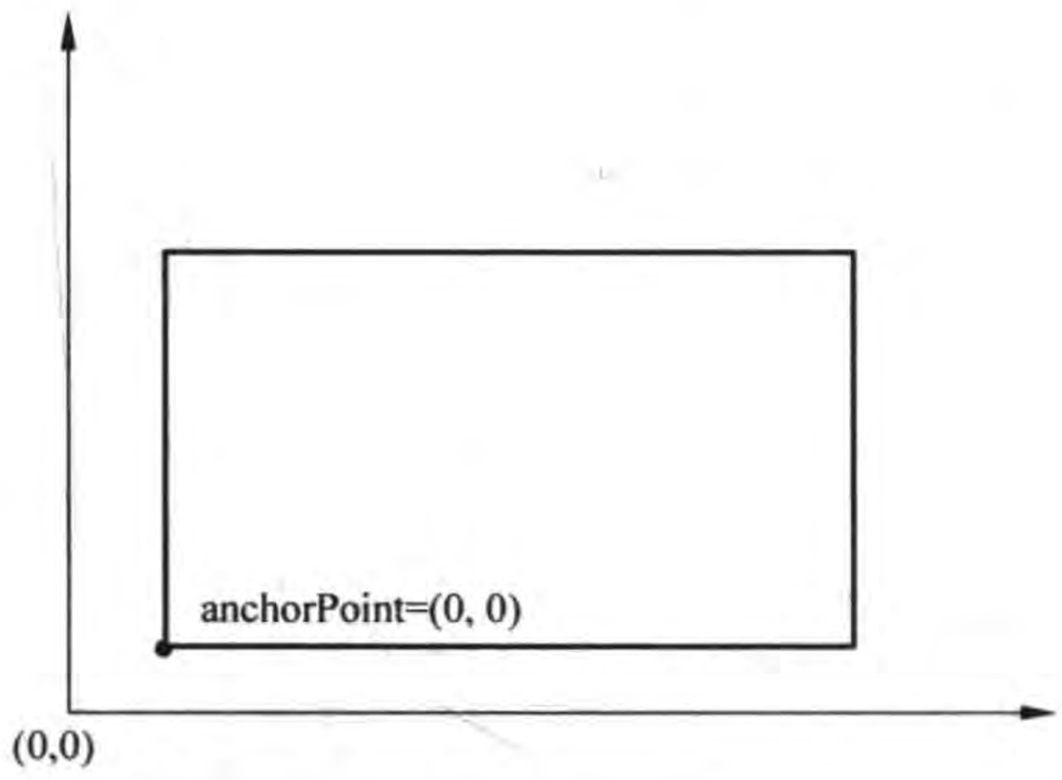


图 4-18　anchorPoint 为(0, 0)

为了进一步了解 anchorPoint 使用,修改 HelloLua 中的 GameScene.lua 代码如下:

```
function GameScene:createLayer()
    cclog("GameScene init")
local layer = cc.Layer:create()

local sprite = cc.Sprite:create("HelloWorld.png")
sprite:setPosition(cc.p(size.width/2, size.height/2))
layer:addChild(sprite)

local label = cc.Label:createWithSystemFont("Hello World ", "Arial", 46)
label:setPosition(cc.p(size.width/2,
    size.height - label:getContentSize().height))
label:setTextColor(cc.c4b(0, 255, 255, 255))
label:setAnchorPoint(cc.p(1.0, 1.0))
```

```
    layer:addChild(label)

    return layer
end
```

运行结果如图 4-19 所示，Hello World 标签设置了 anchorPoint 为(1.0,1.0)。

图 4-19　Hello World 标签的 anchorPoint 为(1.0,1.0)

4.3.3　游戏循环与调度

每一个游戏程序都有一个循环在不断运行，它是由导演对象来管理和维护的。如果需要场景中的精灵运动起来，可以在游戏循环中使用定时器(Scheduler)对精灵等对象的运行进行调度。因为 Node 类封装了 Scheduler 类，所以也可以直接使用 Node 中定时器相关函数。

Node 中定时器主要相关函数如下：

(1) scheduleUpdateWithPriorityLua(nHandler, priority)。每个 Node 对象只要调用该函数，那么这个 Node 对象就会定时地每帧回调用一次 nHandler 函数。priority 是优先级，priority 值越小越先执行。

(2) unscheduleUpdate ()。停止 scheduleUpdateWithPriorityLua 的调度。

为了进一步了解游戏循环与调度的使用，修改 HelloLua 实例。修改后的 GameScene.lua 文件代码如下：

```
local size = cc.Director:getInstance():getWinSize()

local GameScene = class("GameScene",function()
    return cc.Scene:create()
end)
```

```
function GameScene.create()
    local scene = GameScene.new()
    scene:addChild(scene:createLayer())
    return scene
end

function GameScene:ctor()

end

--创建层
function GameScene:createLayer()

    cclog("GameScene init")
    local layer = cc.Layer:create()

    local sprite = cc.Sprite:create("HelloWorld.png")
    sprite:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(sprite)

    local label = cc.Label:createWithSystemFont("Hello World ", "Arial", 46)                ①
    label:setPosition(cc.p(size.width/2,
        size.height - label:getContentSize().height))
    layer:addChild(label)

    local function update(delta)                                                            ②
    local x,y = label:getPosition()
    label:setPosition(cc.p(x + 2, y - 2))
end

--开始游戏调度
layer:scheduleUpdateWithPriorityLua(update, 0)                                              ③

local function onNodeEvent(tag)                                                             ④

    if tag == "exit" then                                                                   ⑤
        --停止游戏调度
        layer:unscheduleUpdate()                                                            ⑥
    end
end
layer:registerScriptHandler(onNodeEvent)                                                    ⑦

    return layer
end
return GameScene
```

上述第①行代码定义了模块级标签对象 label。

第②行代码定义的 update(delta)函数是调度函数。

第③行代码 layer:scheduleUpdateWithPriorityLua(update, 0)是开启游戏调度,按照帧率进行调度,优先级 0 是默认值。

第④行代码是层处理事件回调函数。

第⑤行代码是判断是否为退出层事件，如果是退出层事件则调用第⑥行代码停止调度。
第⑦行代码 layer:registerScriptHandler(onNodeEvent)是注册层事件监听器。

4.4 Cocos2d-x 坐标系

在图形图像和游戏应用开发中，坐标系是非常重要的，在 Android 和 iOS 等平台应用开发时使用的二维坐标系的原点在左上角。而在 Cocos2d-x Lua API 坐标系中，它的原点在左下角，而且 Cocos2d-x Lua API 坐标系又分为世界坐标和模型坐标。

4.4.1 UI 坐标

UI 坐标就是 Android 和 iOS 等应用开发时使用的二维坐标系，它的原点在左上角(见图 4-20)。

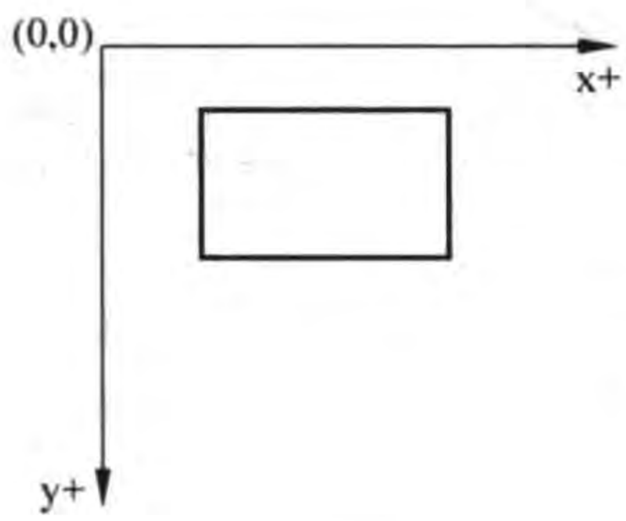


图 4-20 UI 坐标

UI 坐标原点在左上角，x 轴向右为正，y 轴向下为正。在 Android 和 iOS 等平台使用的视图、控件等都是遵守这个坐标系。然而在 Cocos2d-x Lua API 默认不是采用 UI 坐标的情况下，有时也会用到 UI 坐标，例如在触摸事件发生时会获得一个触摸对象(Touch)，触摸对象(Touch)提供了很多获得位置信息的函数，代码如下：

```
cc.p touchLocation = touch:getLocationInView()
```

使用 getLocationInView()函数获得触摸点坐标事实上就是 UI 坐标，它的坐标原点在左上角，而不是 Cocos2d-x Lua API 默认坐标，可以采用下面的语句进行转换：

```
cc.p touchLocation2 = cc.Director:getInstance():convertToGL(touchLocation)
```

通过上面的语句就可以将触摸点位置从 UI 坐标转换为 OpenGL 坐标，OpenGL 坐标就是 Cocos2d-x Lua API 默认坐标。

4.4.2 OpenGL 坐标

上面提到了 OpenGL 坐标是三维坐标。由于 Cocos2d-x Lua API 底层采用 OpenGL 渲染，因此默认坐标就是 OpenGL 坐标，不过只采用两维(x 和 y 轴)。如果不考虑 z 轴，OpenGL 坐标的原点在左下角(见图 4-21)。

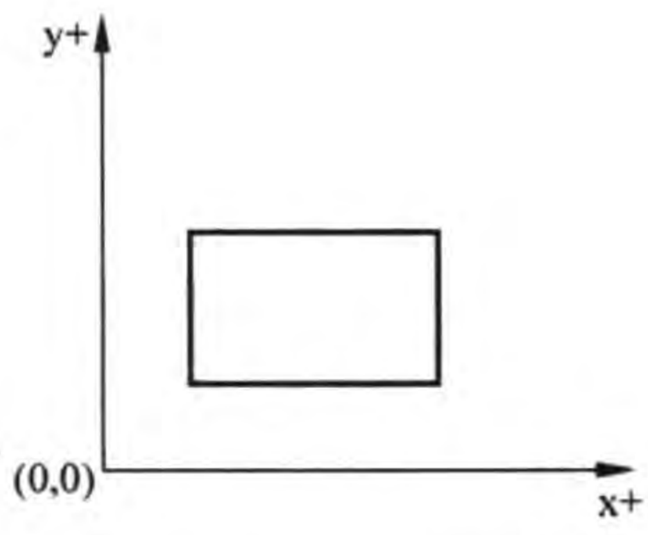


图 4-21 OpenGL 坐标

提示 三维坐标根据 z 轴的指向不同分为左手坐标和右手坐标。右手坐标是 z 轴指向屏幕外，如 4-22 左图所示。左手坐标是 z 轴指向屏幕里，如 4-22 右图所示。OpenGL 坐

标是右手坐标，而微软平台的 Direct3D[①] 是左手坐标。

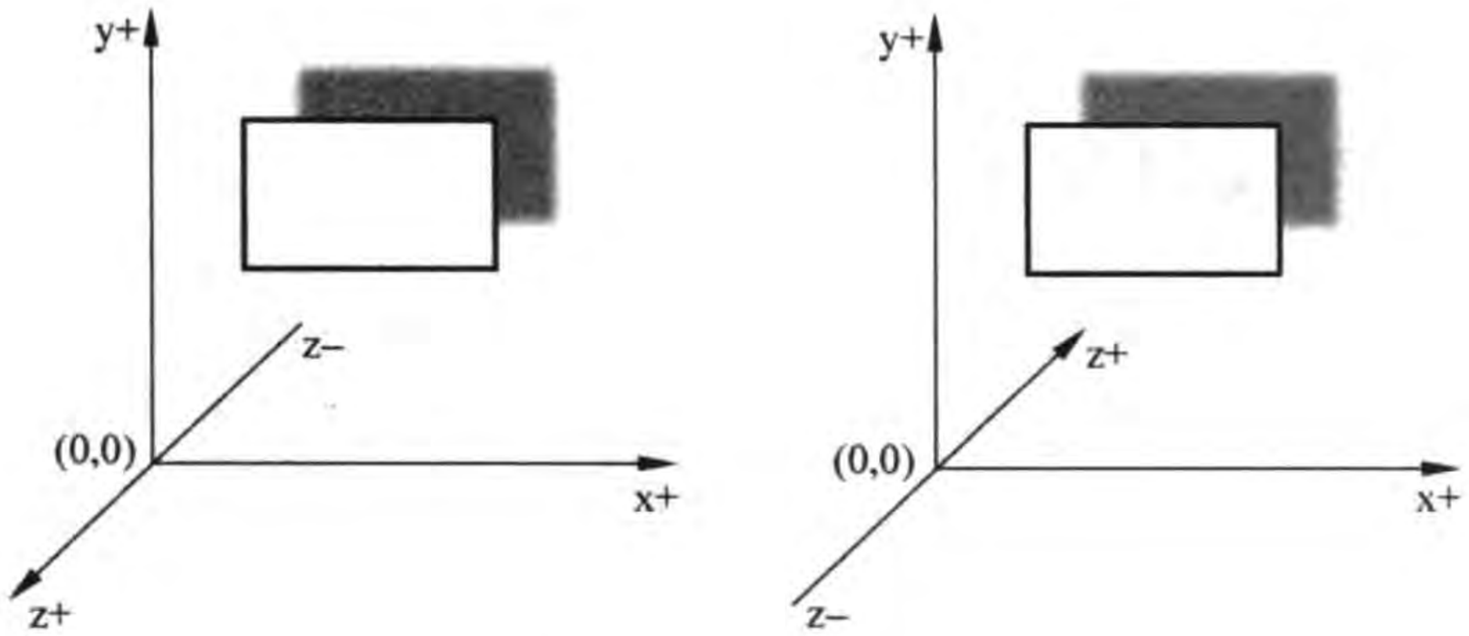


图 4-22　三维坐标

4.4.3　世界坐标和模型坐标

由于 OpenGL 坐标分为世界坐标和模型坐标，所以 Cocos2d-x Lua API 的坐标也有世界坐标和模型坐标之分。

你是否有过这样的问路经历：张三会告诉你向南走 1 公里，再向东走 500 米。而李四会告诉你向右走 1 公里，再向左走 500 米。这里两种说法或许都可以找到你要寻找的地点。张三采用的坐标是世界坐标，把地球作为参照物，表述位置使用地理的东、南、西和北。而李四采用的坐标是模型坐标，让你自己作为参照物，表述位置使用你的左边、你的前边、你的右边和你的后边。

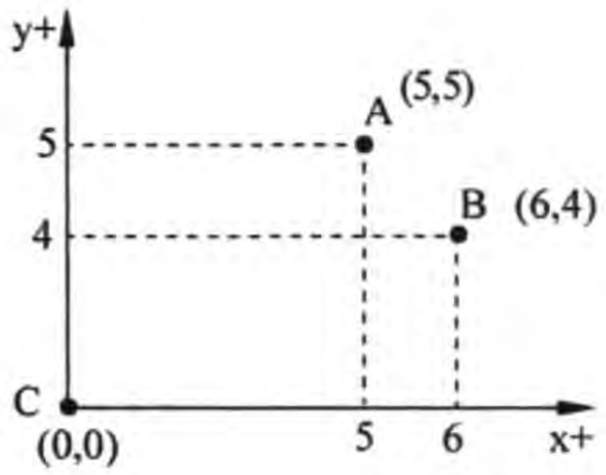


图 4-23　世界坐标和模型坐标

如图 4-23 所示，从图中可以看到 A 的坐标是(5,5)，B 的坐标是(6,4)，事实上这些坐标值就是世界坐标。如果采用 A 的模型坐标来描述 B 的位置，则 B 的坐标是(1,−1)。

有时候需要将世界坐标与模型坐标互相转换。可以通过 Node 对象的如下函数实现：

(1) convertToNodeSpace(worldPoint)：将世界坐标转换为模型坐标。

(2) convertToNodeSpaceAR(worldPoint)：将世界坐标转换为模型坐标，AR 表示相对于锚点。

(3) convertTouchToNodeSpace(touch)：将世界坐标中触摸点转换为模型坐标。

① Direct3D(简称 D3D)是微软公司在 Microsoft Windows 操作系统上所开发的一套 3D 绘图编程接口，是 DirectX 的一部分，目前广为各家显卡所支持。与 OpenGL 同为计算机绘图软件和计算机游戏最常使用的两套绘图编程接口之一。——引自维基百科 http://zh.wikipedia.org/wiki/Direct3D

（4）convertTouchToNodeSpaceAR(touch)：将世界坐标中触摸点转换为模型坐标，AR 表示相对于锚点。

（5）convertToWorldSpace(nodePoint)：将模型坐标转换为世界坐标。

（6）convertToWorldSpaceAR(nodePoint)：将模型坐标转换为世界坐标，AR 表示相对于锚点。

下面通过两个示例了解一下世界坐标与模型坐标互相转换。

1. 世界坐标转换为模型坐标

图 4-24 展示了世界坐标转换为模型坐标的实例运行结果。

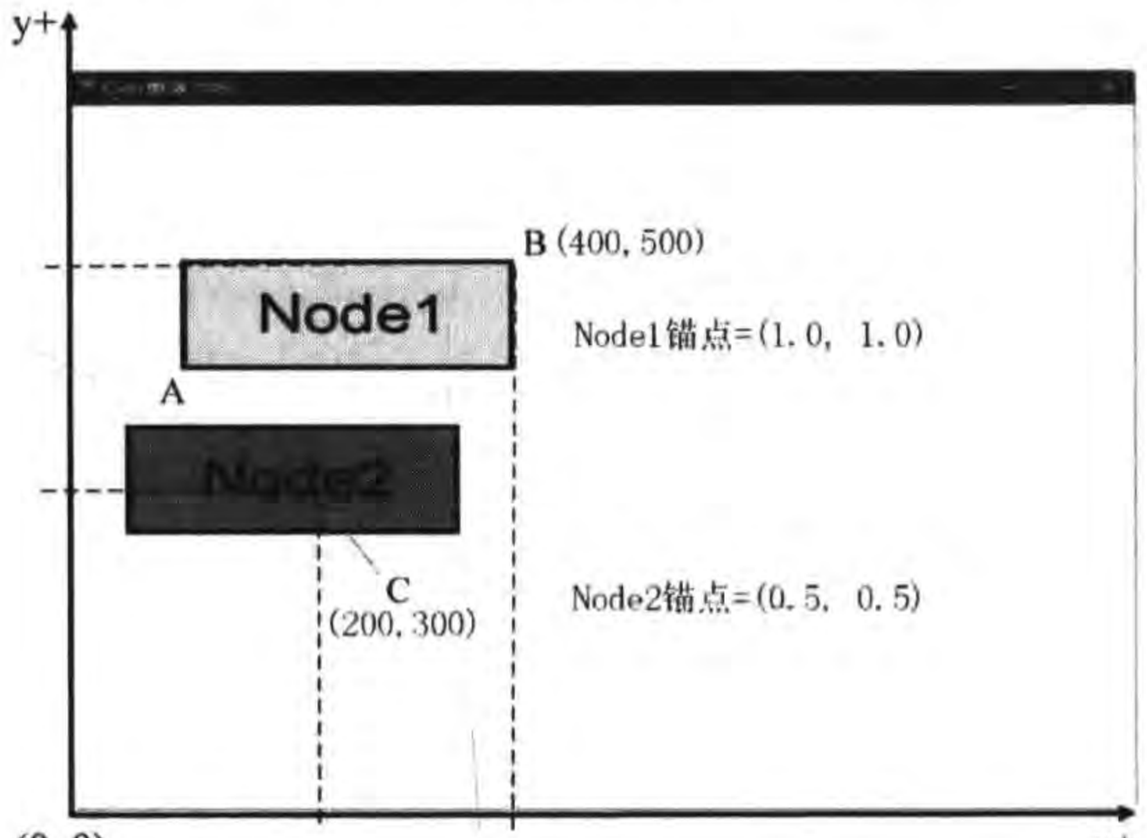


图 4-24　世界坐标转换为模型坐标

在游戏场景中有两个 Node 对象，其中 Node1 的坐标是(400，500)，大小是 300×100 像素。Node2 的坐标是(200，300)，大小也是 300×100 像素。这里的坐标事实上就是世界坐标，它的坐标原点是屏幕的左下角。

编写代码如下：

```
function GameScene:createLayer()
    cclog("GameScene init")
    local layer = cc.Layer:create()
      --创建背景
    local sprite = cc.LayerColor:create(cc.c3b(255, 255, 255))                    ①
    layer:addChild(sprite)                                                        ②

    local closeItem = cc.MenuItemImage:create(
        "CloseNormal.png",
        "CloseSelected.png")
    closeItem:setPosition(cc.p(size.width - closeItem:getContentSize().width/2,
                               closeItem:getContentSize().height/2))
    local menu = cc.Menu:create(closeItem)
    menu:setPosition(cc.p(0, 0))
```

```
    layer:addChild(menu)

    -- 创建 Node1
    local node1 = cc.Sprite:create("node1.png")                                    ③
    node1:setPosition(cc.p(400,500))
    node1:setAnchorPoint(cc.p(1.0, 1.0))

    layer:addChild(node1, 0)                                                       ④
    -- 创建 Node2
    local node2 = cc.Sprite:create("node2.png")                                    ⑤
    node2:setPosition(cc.p(200,300))
    node2:setAnchorPoint(cc.p(0.5, 0.5))
    layer:addChild(node2, 0)                                                       ⑥

    local posX,posY = node2:getPosition()
    local point1 = node1:convertToNodeSpace(cc.p(posX,posY))                       ⑦
    local point3 = node1:convertToNodeSpaceAR(cc.p(posX,posY))                     ⑧

    cclog("Node2 NodeSpace = ( %f, %f)",point1.x, point1.y)
    cclog("Node2 NodeSpaceAR = ( %f, %f)",point3.x,point3.y)

    return layer
end
```

第①和第②行代码是创建背景精灵对象,这个背景是一个白色 900×640 像素的图片。

第③和第④行代码是创建 Node1 对象,并设置了位置和锚点属性。

第⑤和第⑥行代码是创建 Node2 对象,并设置了位置和锚点属性。

第⑦行代码将 Node2 的世界坐标转换为相对于 Node1 的模型坐标。

而第⑧行代码是类似的,它是相对于锚点的位置。

运行结果如下:

```
Node2 NodeSpace = (100.000000, -100.000000)
Node2 NodeSpaceAR = (-200.000000, -200.000000)
```

结合图 4-24 解释一下,Node2 的世界坐标转换为相对于 Node1 的模型坐标,就是将 Node1 的左下角作为坐标原点(图 4-24 中的 A 点),不难计算出 A 点的世界坐标是(100, 400),那么 convertToNodeSpace 函数就是 C 点坐标减去 A 点坐标,结果是(100,-100)。

而 convertToNodeSpaceAR 函数要考虑锚点,因此坐标原点是 B 点,C 点坐标减去 B 点坐标,结果是(-200,-200)。

2. 模型坐标转换为世界坐标

图 4-25 展示了模型坐标转换为世界坐标的实例运行结果。

在游戏场景中有两个 Node 对象,其中 Node1 的坐标是(400, 500),大小是 300×100 像素。Node2 是放置在 Node1 中的,它对于 Node1 的模型坐标是(0, 0),大小是 150×50 像素。

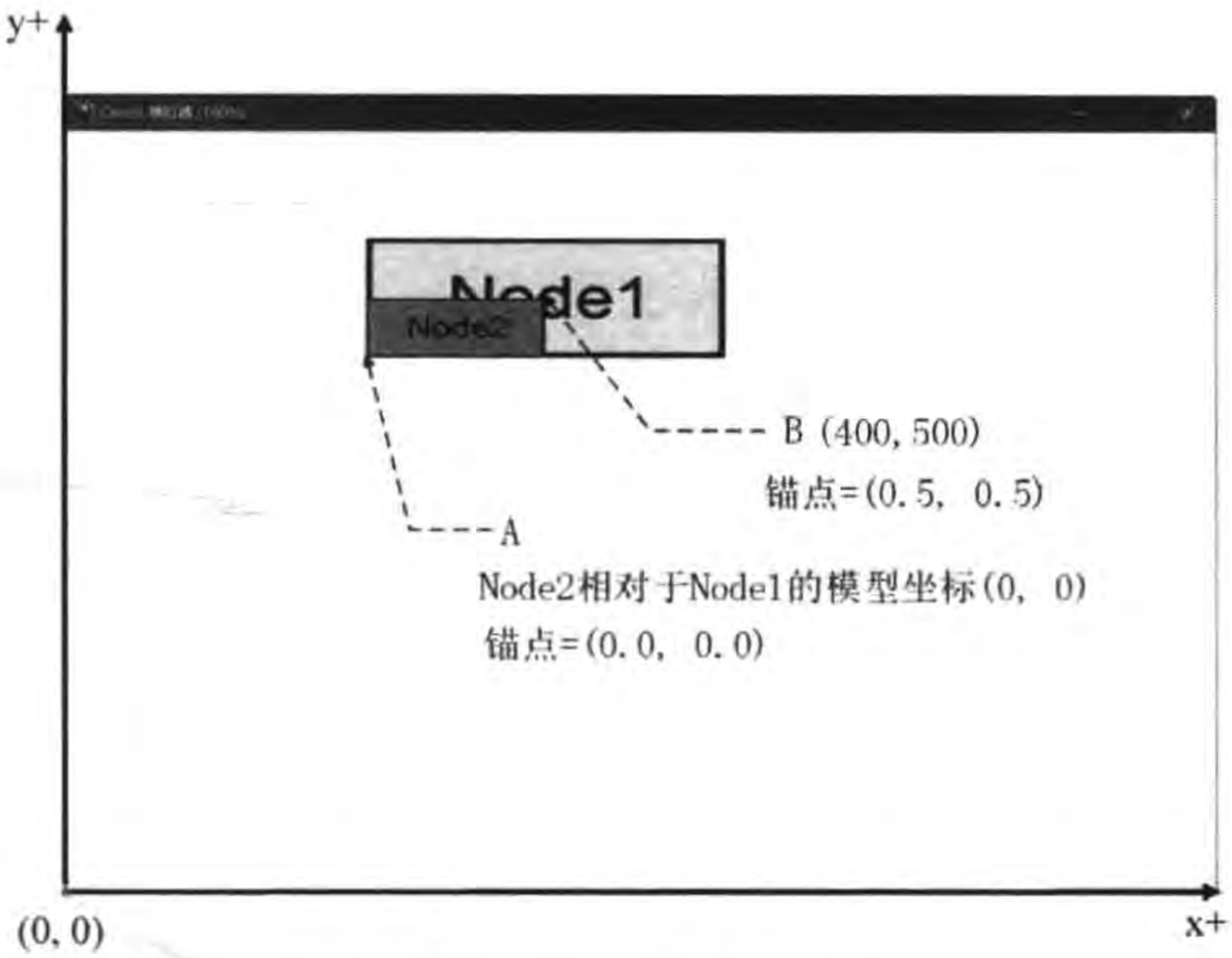


图 4-25 模型坐标转换为世界坐标

编写代码如下：

```
function GameScene:createLayer()
    cclog("GameScene init")
    local layer = cc.Layer:create()

    --创建背景
    local sprite = cc.LayerColor:create(cc.c3b(255, 255, 255))
    layer:addChild(sprite)

    local closeItem = cc.MenuItemImage:create(
        "CloseNormal.png",
        "CloseSelected.png")
    closeItem:setPosition(cc.p(size.width - closeItem:getContentSize().width/2,
                               closeItem:getContentSize().height/2))
    local menu = cc.Menu:create(closeItem)
    menu:setPosition(cc.p(0, 0))
    layer:addChild(menu)

    --创建 Node1
    local node1 = cc.Sprite:create("node1.png")
    node1:setPosition(cc.p(400,500))
    layer:addChild(node1, 0)

    --创建 Node2
    local node2 = cc.Sprite:create("node2.png")
    node2:setPosition(cc.p(0.0, 0.0))                                    ①
    node2:setAnchorPoint(cc.p(0.0, 0.0))
    node1:addChild(node2, 0)                                             ②

    local posX,posY = node2:getPosition()
    local point1 = node1:convertToWorldSpace(cc.p(posX,posY))            ③
```

```
    local point3 = node1:convertToWorldSpaceAR(cc.p(posX,posY))                    ④

    cclog("Node2 WorldSpace = (%f, %f)",point1.x, point1.y)
    cclog("Node2 WorldSpaceAR = (%f, %f)",point3.x,point3.y)

    return layer
end
```

上述主要关注第②行代码，它是将 Node2 放到 Node1 中，这是与之前代码的区别。这样第①行代码设置的坐标就变成了相对于 Node1 的模型坐标了。

第③行代码将 Node2 的模型坐标转换为世界坐标。而第④行代码是类似的，它是相对于锚点的位置。

运行结果如下：

```
Node2 WorldSpace = (250.000000,450.000000)
Node2 WorldSpaceAR = (400.000000,500.000000)
```

如图 4-24 所示的位置，也可以用世界坐标描述，修改第①和第②行代码如下：

```
local node2 = cc.Sprite:create("node2.png")
node2:setPosition(cc.p(0.0, 0.0))
node2:setAnchorPoint(cc.p(0.0, 0.0))
this:addChild(node2, 0)
```

本章小结

通过对本章的学习，读者可以了解 Cocos2d-x Lua API 开发环境的搭建，以及熟悉 Cocos2d-x Lua API 核心概念，这些概念包括导演、场景、层、精灵和菜单等节点对象。此外，还重点学习了 Node 和 Node 层级架构。最后还介绍了 Cocos2d-x Lua API 的坐标系。

第5章

标签和菜单

游戏中文字与菜单很常用，字符串、标签和菜单经常结合在一起使用，因此本章主要介绍 Cocos2d-x Lua API 中的标签与菜单。

5.1 游戏中的文字

游戏场景中的文字包括静态文字和动态文字。静态文字如图 5-1 所示游戏场景中①号文字“COCOS2DX”，动态文字如图 5-1 所示游戏场景中的②号文字“Hello World”。

图 5-1 场景中的文字

静态文字一般是由美工使用 Photoshop 绘制在背景图片上，这种方式的优点是表现力很丰富，例如，①号文字“COCOS2DX”中的“COCOS”、“2D”和“X”设计的风格不同，而动态文字则不能，但是静态文字无法通过程序访问，无法动态修改内容。

动态文字一般是需要通过程序访问，需要动态修改内容。Cocos2d-x Lua API 可以通

过标签类实现。

5.2 使用标签

要想在游戏场景中显示动态文字,可以通过 Cocos2d-x Lua API 提供的标签类 Label 和 LabelAtlas 实现,也可通过 Cocos2d-x GUI 标签控件实现。本章介绍标签类 Label 和 LabelAtlas 实现方式。

5.2.1 Label 类

Label 类是 Cocos2d-x 3.x 后推出的新的标签类,这种标签通过使用 FreeType[①] 来使它在不同的平台上有相同的视觉效果。由于使用更快的缓存代理,它的渲染也将更加快速。Label 提供了描边和阴影等特性。Label 类可以创建系统字体标签、TTF 字体标签和位图字体标签。

提示 系统字体标签就是当前平台系统中安装的字体,我们要确保所用字体在系统中是存在的。TTF 字体[②]标签是加载 TTF 字体文件显示文字的标签。位图字体标签加载两个文件:一个纹理图集(.png)和一个字体坐标文件(.fnt)。

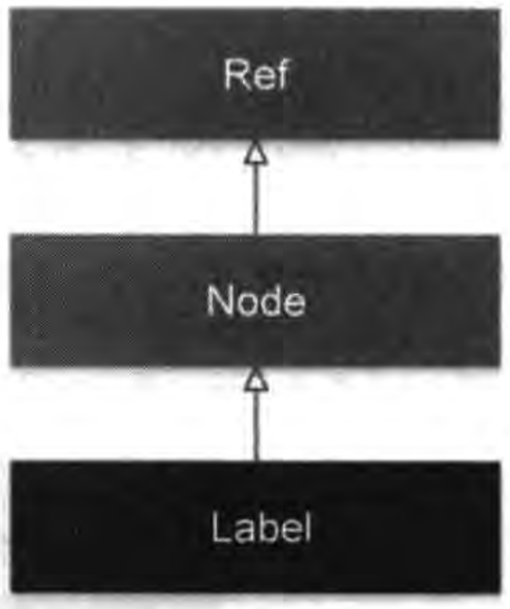


图 5-2 Label 类图

Label 类的类图如图 5-2 所示。

创建 Label 类静态 create 函数常用的有如下几个:

```
Label:createWithSystemFont(text,                                    -- 是要显示的文字
      font,                                                         -- 系统字体名
      fontSize,                                                     -- 字体的大小
      dimensions = size(0,0),                                       -- 可省略,参考 LabelTTF 定义
      vAlignment = TEXT_ALIGNMENT_LEFT,                             -- 可省略,参考 LabelTTF 定义
      vAlignment = VERTICAL_TEXT_ALIGNMENT_TOP                      -- 可省略,参考 LabelTTF 定义
)

Label:createWithTTF(const std::string & text,
      fontFile,                                                     -- 字体文件
      fontSize,
      dimensions = size(0,0),
      hAlignment = TEXT_ALIGNMENT_LEFT,
      vAlignment = VERTICAL_TEXT_ALIGNMENT_TOP
    )
```

① FreeType 库是一个完全免费(开源)的、高质量的且可移植的字体引擎,它提供统一的接口来访问多种字体格式文件。——引自百度百科 http://baike.baidu.com/view/4579855.htm

② TTF(TrueTypeFont)是 Apple 公司和 Microsoft 公司共同推出的字体文件格式,随着 Windows 的流行,已经变成最常用的一种字体文件表示方式。——引自百度百科 http://baike.baidu.com/view/1003637.htm

```
Label:createWithTTF(ttfConfig,                                    -- 字体配置信息
      text,
      hAlignment = TEXT_ALIGNMENT_LEFT,
      int maxLineWidth = 0                                        -- 可省略,标签的最大宽度
    )

Label:createWithBMFont(const std::string& bmfontFilePath,  -- 位图字体文件
      text,
      hAlignment = TEXT_ALIGNMENT_LEFT,
      int maxLineWidth = 0,
      imageOffset = p(0,0)                                        -- 可省略,在位图中的偏移量
    )
```

其中 createWithSystemFont 是创建系统字体标签对象,createWithTTF 是创建 TTF 字体标签对象,createWithBMFont 是创建位图字体标签对象。

5.2.2　实例:使用系统字体和 TTF 字体

下面用实例说明通过 Label 类使用系统字体和 TTF 字体,如图 5-3 所示。

图 5-3　使用系统字体和 TTF 字体

GameScene.lua 的 GameScene:createLayer()函数如下:

```
-- 创建层
function GameScene:createLayer()
cclog("GameScene init")
local layer = cc.Layer:create()

-- 上下两个控件的距离
local gap = 120

local label1 = cc.Label:createWithSystemFont("世界你好 1", "Arial", 36)          ①
label1:setPosition(cc.p(size.width/2, size.height - gap))
layer:addChild(label1, 1)

local label2 = cc.Label:createWithTTF("世界你好 2", "fonts/STLITI.ttf", 36)      ②
```

```
    label2:setPosition(cc.p(size.width/2, size.height - 2 * gap))
    layer:addChild(label2, 1)

    local ttfConfig = {}                                                    ③
    ttfConfig.fontFilePath = "fonts/Marker Felt.ttf"
    ttfConfig.fontSize = 32

    local label3 = cc.Label:createWithTTF(ttfConfig, "Hello World1")        ④
    label3:setPosition(cc.p(size.width/2, size.height - 3 * gap))
    layer:addChild(label3 , 1)

    ttfConfig.outlineSize = 4                                               ⑤
    local label4 = cc.Label:createWithTTF(ttfConfig, "Hello World2")        ⑥
    label4:setPosition(cc.p(size.width/2, size.height - 4 * gap))
    label5:enableShadow(cc.c4b(255,255,255,128), cc.size(4, -4))            ⑦
    label5:setColor(cc.c3b(255, 0, 0))                                      ⑧
    layer:addChild(label4, 1)

    return layer
end
```

上述第①行代码是通过 createWithSystemFont 函数创建系统字体标签对象。

第②行代码是通过 createWithTTF 创建 TTF 字体标签对象。

第③行代码 local ttfConfig = {}是声明一个 ttfConfig 变量，ttfConfig 的属性如下：

```
fontFilePath                    -- 字体文件路径
fontSize,                       -- 字体大小
glyphs = GLYPHCOLLECTION_DYNAMIC,  -- 字体库类型
customGlyphs                    -- 自定义字体库
outlineSize                     -- 字体描边
distanceFieldEnabled            -- 开启距离字段字体开关
```

第④行代码 cc. Label: createWithTTF(ttfConfig, "Hello World1")是通过指定 ttfConfig 创建 TTF 字体标签。

第⑤行代码 ttfConfig. outlineSize = 4 设置 ttfConfig 的描边字段。

第⑥行代码 cc. Label: createWithTTF(ttfConfig, "Hello World2")是重新创建 TTF 字体标签。

第⑦行代码 label5: enableShadow(cc. c4b(255, 255,255,128), cc. size(4, -4))是设置标签的阴影效果。

第⑧行代码 label5: setColor(cc. c3b(255, 0, 0))是设置标签的颜色。

5.2.3 实例：使用位图字体

位图字体标签加载两个文件：一个图片集(. png)和一个字体坐标文件(. fnt)。图 5-4 展示了图片集文件

图 5-4 图片集文件

BMFont.png 的内容，对应还有一个字体坐标文件 BMFont.fnt。

字体坐标文件 BMFont.fnt 代码如下：

```
info face = "AmericanTypewriter" size = 64 bold = 0 italic = 0 charset = "" unicode = 0 stretchH =
100 smooth = 1 aa = 1 padding = 0,0,0,0 spacing = 2,2
common lineHeight = 73 base = 58 scaleW = 512 scaleH = 512 pages = 1 packed = 0
page id = 0 file = "BMFont.png"
chars count = 95
char id = 124 x = 2 y = 2 width = 9 height = 68 xoffset = 14 yoffset = 9 xadvance = 32 page = 0 chnl =
0 letter = "|"
char id = 41 x = 13 y = 2 width = 28 height = 63 xoffset = 1 yoffset = 11 xadvance = 29 page = 0 chnl
 = 0 letter = ")"
char id = 40 x = 43 y = 2 width = 28 height = 63 xoffset = 4 yoffset = 11 xadvance = 29 page = 0 chnl
 = 0 letter = "("
... ...
char id = 32 x = 200 y = 366 width = 0 height = 0 xoffset = 16 yoffset = 78 xadvance = 16 page = 0
chnl = 0
letter = "space"
```

使用 LabelBMFont 需要注意的是图片集文件和坐标文件需要放置在相同的资源目录下，文件命名相同。图片集合和坐标文件是可以通过位图字体工具制作而成的，位图字体工具的使用请参考本系列丛书的工具卷(《Cocos2d-x 实战：工具卷》)。

下面用实例说明通过 Label 类使用位图字体，如图 5-5 所示。

图 5-5　使用位图字体

下面看一下 GameScene.lua 的 GameScene:createLayer()函数如下：

```
--创建层
function GameScene:createLayer()
    cclog("GameScene init")
    local layer = cc.Layer:create()
```

```
    -- 上下两个控件的距离
    local gap = 90

    local label1 = cc.Label:createWithBMFont("fonts/bitmapFontChinese.fnt", "中国")      ①
    label1:setPosition(cc.p(size.width/2, size.height - 3 * gap))
    layer:addChild(label1, 1)

    local label2 = cc.Label:createWithBMFont("fonts/BMFont.fnt", "Hello World")          ②
    label2:setPosition(cc.p(size.width/2, size.height - 4 * gap))
    layer:addChild(label2, 1)

    return layer
end

return GameScene
```

第①和第②行代码通过 Label 类的 createWithBMFont 函数创建位图字体标签对象。

注意 标签中显示的文字字符一定包含在位图字体图片集中，例如代码第①行要显示中文，那么位图字体图片集 bitmapFontChinese.png 文件中应当包含这些中文字。

5.2.4 LabelAtlas 类

LabelAtlas 是图片集标签，其中的 Atlas 本意是“地图集”、“图片集”，这种标签显示的文字是从一个图片集中取出的，因此使用 LabelAtlas 需要额外加载图片集文件。LabelAtlas 比 LabelTTF 快很多。LabelAtlas 中的每个字符必须有固定的高度和宽度。

下面用实例说明通过 LabelAtlas 类使用图片集字体，如图 5-6 所示。

图 5-6 LabelAtlas 实现的图片集文字

LabelAtlas 实现的图片集文字主要代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local label = cc.LabelAtlas:create("HelloWorld",
                "fonts/tuffy_bold_italic-charmap.png", 48, 66, string.byte(" "))      ①
    label:setPosition(cc.p(size.width/2 - label:getContentSize().width / 2,
                           size.height - label:getContentSize().height))
    layer:addChild(label, 1)

    local sprite = cc.Sprite:create("HelloWorld.png")
    sprite:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(sprite, 0)

    return layer
end
```

上述第①行代码是创建一个 LabelAtlas 对象，create 函数的第一个参数是要显示的文字，第二个参数是图片集文件(见图 5-7)，第三个参数是字符高度，第四个参数是字符宽度，第五个参数是开始字符。

使用 LabelAtlas 需要注意的是图片集文件需要放置在 res 目录下。

图 5-7　图片集文件

5.3　使用菜单

菜单中又包含了菜单项，菜单项类是 MenuItem，每个菜单项都有三个基本状态：正常、选种和禁止。回顾一下 MenuItem 类图如图 5-8 所示。

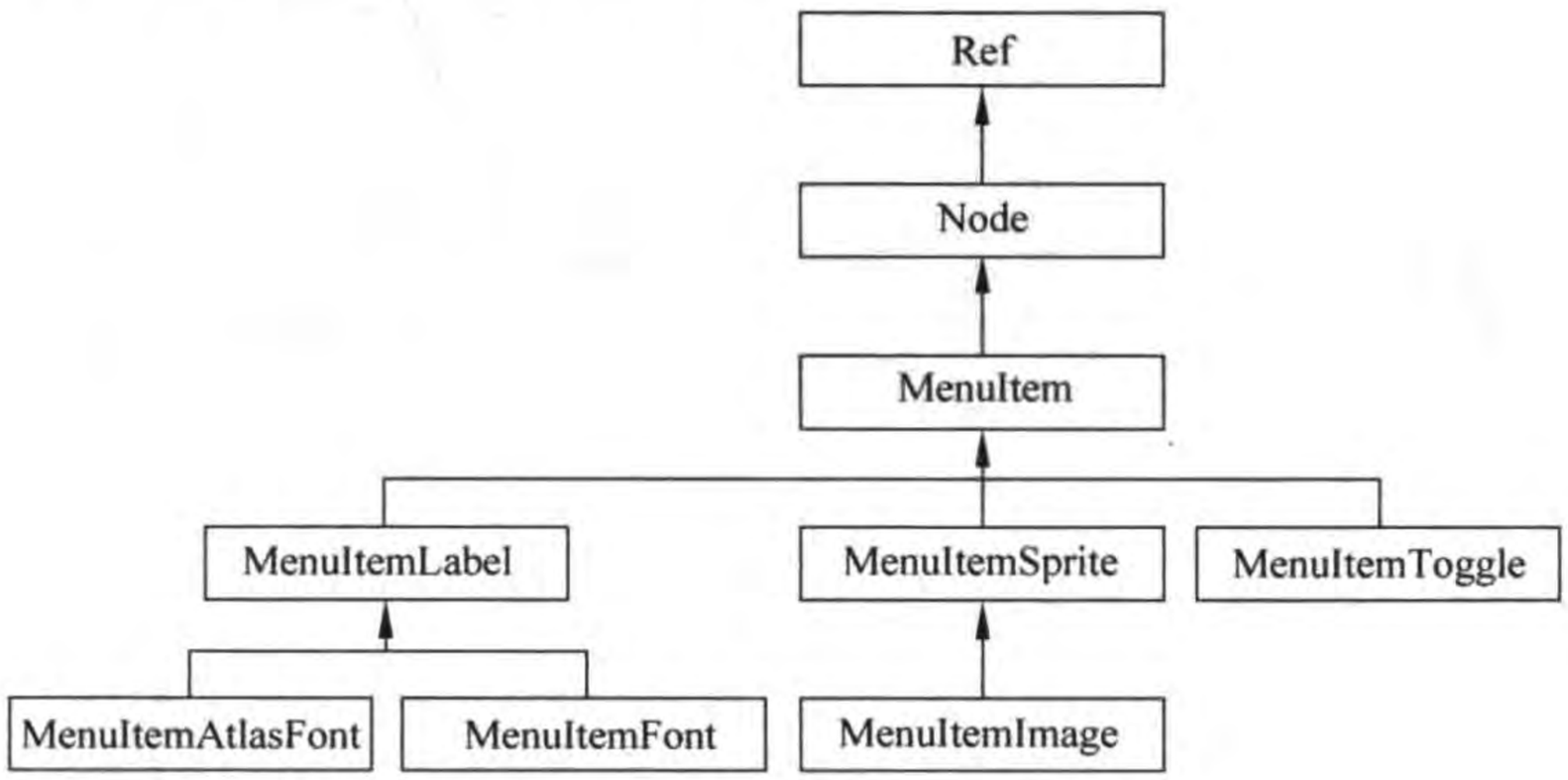


图 5-8　MenuItem 类图

菜单是按照菜单项进行分类的，从 MenuItem 类图中可见 MenuItem 的子类有 MenuItemLabel、MenuItemSprite 和 MenuItemToggle。其中 MenuItemLabel 类是文本菜单，它有 2 个子类：MenuItemAtlasFont 和 MenuItemFont。MenuItemSprite 类是精灵菜单，它的子类是 MenuItemImage，属于图片菜单。MenuItemToggle 类是开关菜单。

下面重点介绍文本菜单、精灵菜单、图片菜单和开关菜单。

5.3.1 文本菜单

文本菜单的菜单项只能显示文本，包括 MenuItemLabel、MenuItemFont 和 MenuItemAtlasFont。MenuItemLabel 是个抽象类，具体使用时只使用 MenuItemFont 和 MenuItemAtlasFont 两个类。

提示 目前的 Cocos2d-x 3.11 版本 Lua 中 MenuItemAtlasFont 还没有可移植性，因此还不能使用。可以使用 LabelAtlas 和 MenuItemLabel 结合来实现 MenuItemAtlasFont 效果。

本节通过一个实例来介绍文本菜单的使用，如图 5-9 所示，其中菜单 Start 是文本字体菜单项，菜单 Help 是 Atlas 字体菜单项。

图 5-9 文本菜单实例

GameScene.lua 的 GameScene:createLayer()函数如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local sprite = cc.Sprite:create("menu/background.png")
```

```
sprite:setPosition(cc.p(size.width/2, size.height/2))
layer:addChild(sprite)

cc.MenuItemFont:setFontName("Times New Roman")                              ①
cc.MenuItemFont:setFontSize(86)                                             ②

local item1 = cc.MenuItemFont:create("Start")                               ③
local function menuItem1Callback(sender)                                    ④
    cclog("Touch Start Menu Item.")
end
item1:registerScriptTapHandler(menuItem1Callback)                           ⑤

local labelAtlas = cc.LabelAtlas:create("Help",
    "menu/tuffy_bold_italic-charmap.png", 48, 65, string.byte(' '))         ⑥
local item2 = cc.MenuItemLabel:create(labelAtlas)                           ⑦
local function menuItem2Callback(sender)
    cclog("Touch Help Menu Item.")
end
item2:registerScriptTapHandler(menuItem2Callback)

local mn = cc.Menu:create(item1, item2)                                     ⑧
mn:alignItemsVertically()                                                   ⑨
layer:addChild(mn)                                                          ⑩

return layer
end
```

上述第①和第②行代码是设置文本菜单的文本字体和字体大小。

第③行代码是创建 MenuItemFont 菜单项对象，它是一个一般文本菜单项，create 函数的参数是菜单项的文本内容。

第④行代码是定义 MenuItemFont 菜单项单击事件的回调函数。

第⑤行代码是注册 MenuItemFont 菜单项单击事件，其参数就是前面定义的回调函数。

第⑥行代码是定义一个 Atlas 标签。

第⑦行代码是通过 Atlas 标签创建 MenuItemLabel 菜单项对象。

第⑧行代码 cc.Menu:create(item1, item2)是创建菜单对象，把之前创建的菜单项添加到菜单中，create 函数是这些菜单项的数组。

第⑨行代码 mn:alignItemsVertically()是设置菜单项垂直对齐。

第⑩行代码 layer:addChild(mn)是把菜单对象添加到当前层中。

5.3.2　精灵菜单和图片菜单

精灵菜单的菜单项类是 MenuItemSprite，图片菜单的菜单项类是 MenuItemImage。由于 MenuItemImage 继承于 MenuItemSprite，所以图片菜单也属于精灵菜单。为什么叫精灵菜单呢？那是因为这些菜单项具有精灵的特点，可以让精灵动起来，具体使用时是把一个精灵放置到菜单中作为菜单项。

精灵菜单项类 MenuItemSprite 的一个创建函数 create 定义如下：

```
cc.MenuItemSprite:create ( normalSprite,                    -- 菜单项正常显示时的精灵
  selectedSprite,                                           -- 选择菜单项时的精灵
  disabledSprite                                            -- 菜单项禁用时的精灵
)
```

使用 MenuItemSprite 比较麻烦，在创建 MenuItemSprite 之前要先创建 3 种不同状态的精灵（即 normalSprite、selectedSprite 和 disabledSprite）。MenuItemSprite 还有一些 create 函数，在这些函数中可以省略 disabledSprite 参数。

如果精灵是由图片构成的，可以使用 MenuItemImage 实现与精灵菜单同样的效果。MenuItemImage 类的一个创建函数 create 定义如下：

```
cc.MenuItemImage:create (normalImage,                       -- 菜单项正常显示时的图片
  selectedImage,                                            -- 选择菜单项时的图片
  disabledImage                                             -- 菜单项禁用时的图片
)
```

MenuItemImage 还有一些 create 函数，在这些函数中可以省略 disabledImage 参数。

本节通过一个实例介绍精灵菜单和图片菜单的使用，如图 5-10 所示。

图 5-10　精灵菜单和图片菜单实例

GameScene.lua 的 GameScene:createLayer()函数如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()
    local director = cc.Director:getInstance()
    local sprite = cc.Sprite:create("menu/background.png")
    sprite:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(sprite)

    -- 开始精灵
    local startlocalNormal = cc.Sprite:create("menu/start - up.png")                          ①
    local startlocalSelected = cc.Sprite:create("menu/start - down.png")                      ②
    local startMenuItem = cc.MenuItemSprite:create(startlocalNormal, startlocalSelected)      ③
    startMenuItem:setPosition(director:convertToGL(cc.p(700, 170)))                           ④
    local function menuItemStartCallback(sender)
        cclog("Touch Start.")
    end
```

```
startMenuItem:registerScriptTapHandler(menuItemStartCallback)

-- 设置图片菜单
local settingMenuItem = cc.MenuItemImage:create(
    "menu/setting - up.png",
    "menu/setting - down.png")                                          ⑤
settingMenuItem:setPosition(director:convertToGL(cc.p(480, 400)))       ⑥
local function menuItemSettingCallback(sender)
    cclog("Touch Setting.")
end
settingMenuItem:registerScriptTapHandler(menuItemSettingCallback)

-- 帮助图片菜单
local helpMenuItem = cc.MenuItemImage:create(
    "menu/help - up.png",
    "menu/help - down.png")                                             ⑦
helpMenuItem:setPosition(director:convertToGL(cc.p(860, 480)))          ⑧
local function menuItemHelpCallback(sender)
    cclog("Touch Help.")
end
helpMenuItem:registerScriptTapHandler(menuItemHelpCallback)

local mn = cc.Menu:create(startMenuItem, settingMenuItem, helpMenuItem) ⑨
mn:setPosition(cc.p(0, 0))                                              ⑩
layer:addChild(mn)

return layer
end
```

上述第①和第②行代码是创建两种不同状态的精灵。

第③行代码是创建精灵菜单项 MenuItemSprite 对象。

第④行代码是设置开始菜单项(startMenuItem)位置，注意这个坐标是(700，170)，由于(700，170)的坐标是 UI 坐标，需要转换为 OpenGL 坐标。

第⑤和⑦行代码是创建图片菜单项 MenuItemImage 对象。

第⑥和⑧行代码是设置图片菜单项位置。

第⑨行代码是 Menu 对象。

第⑩行代码是菜单的位置 mn:setPosition(cc.p(0，0))。

由于背景图片大小是 1136×640，而 Cocos Code IDE 工具模板生成的窗口大小是 960×640，所以需要重新设置大小，修改 main.lua 代码如下：

```
local function main()
    collectgarbage("collect")
    -- avoid memory leak
    collectgarbage("setpause", 100)
    collectgarbage("setstepmul", 5000)

    cc.FileUtils:getInstance():addSearchPath("src")
    cc.FileUtils:getInstance():addSearchPath("res")
    cc.Director:getInstance():getOpenGLView():setDesignResolutionSize(1136, 640, 0)    ①
```

```
    -- create scene
    local scene = require("GameScene")
    local gameScene = scene.create()

    if cc.Director:getInstance():getRunningScene() then
        cc.Director:getInstance():replaceScene(gameScene)
    else
        cc.Director:getInstance():runWithScene(gameScene)
    end

end
```

需要在第①行修改 setDesignResolutionSize(1136, 640, 0)代码。

5.3.3 开关菜单

开关菜单的菜单项类是 MenuItemToggle,它是一种可以进行 2 种状态切换的菜单项。本节通过一个实例介绍其他复杂类型的开关菜单的使用。如图 5-11 所示是一个游戏音效和背景音乐设置界面,可以通过开关菜单实现这个功能,美术设计师为每一个设置项目(音效和背景音乐)分别准备了两张图片。

图 5-11 开关菜单实例

GameScene.lua 中 MenuItemImage 菜单项的代码如下:

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local director = cc.Director:getInstance()

    local sprite = cc.Sprite:create("menu/setting-back.png")
    sprite:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(sprite)

    -- 音效
    local soundOnMenuItem = cc.MenuItemImage:create("menu/on.png", "menu/on.png")          ①
    local soundOffMenuItem = cc.MenuItemImage:create("menu/off.png", "menu/off.png")       ②
    local soundToggleMenuItem = cc.MenuItemToggle:create(soundOnMenuItem,
```

```
                                                soundOffMenuItem)                    ③
soundToggleMenuItem:setPosition(director:convertToGL(cc.p(818, 220)))
local function menuSoundToggleCallback(sender)
    cclog("Sound Toggle.")
end
soundToggleMenuItem:registerScriptTapHandler(menuSoundToggleCallback)

-- 音乐
local musicOnMenuItem = cc.MenuItemImage:create("menu/on.png", "menu/on.png")      ④
local musicOffMenuItem = cc.MenuItemImage:create("menu/off.png", "menu/off.png")   ⑤
local musicToggleMenuItem = cc.MenuItemToggle:create(musicOnMenuItem,
                                                     musicOffMenuItem)             ⑥
musicToggleMenuItem:setPosition(director:convertToGL(cc.p(818, 362)))
local function menuMusicToggleCallback(sender)
    cclog("Music Toggle.")
end
musicToggleMenuItem:registerScriptTapHandler(menuMusicToggleCallback)

-- Ok 按钮
local okMenuItem = cc.MenuItemImage:create(
    "menu/ok - down.png",
    "menu/ok - up.png")
okMenuItem:setPosition(director:convertToGL(cc.p(600, 510)))

local mn = cc.Menu:create(soundToggleMenuItem, musicToggleMenuItem,okMenuItem)     ⑦
mn:setPosition(cc.p(0, 0))
layer:addChild(mn)

    return layer
end
```

上面第①行代码是创建音效开的图片菜单项。

第②行代码是创建音效关的图片菜单项。

第③行代码是创建开关菜单项 MenuItemToggle。

第④～⑥行代码创建了背景音乐开关菜单项。

第⑦行代码是通过上面创建的开关菜单项创建 Menu 对象。

本章小结

通过对本章的学习，使读者了解了 Cocos2d-x Lua API 文字和菜单的相关知识，本章介绍了标签类 Label 和 LabelAtlas，另外，在菜单部分还介绍了文本菜单、精灵菜单、图片菜单和开关菜单等。

第6章 精　灵

前面用到了精灵对象但没有深入地介绍，本章将深入地介绍精灵的使用。精灵是游戏中非常重要的概念，围绕着精灵还有很多概念，如精灵帧、缓存、动作和动画等。

6.1 Sprite 精灵类

Sprite 类图如图 6-1 所示，从图中可见 Sprite 是 Node 子类，Sprite 包含很多类型，如广告牌精灵类 BillBoard。

Sprite 类直接继承了 Node 类，具有 Node 基本特征。此外，Cocos2d-x 现在还提供了 3D 精灵类 Sprite3D，关于 Cocos2d-x 中的 3D 特征将在第 15 章详细介绍。

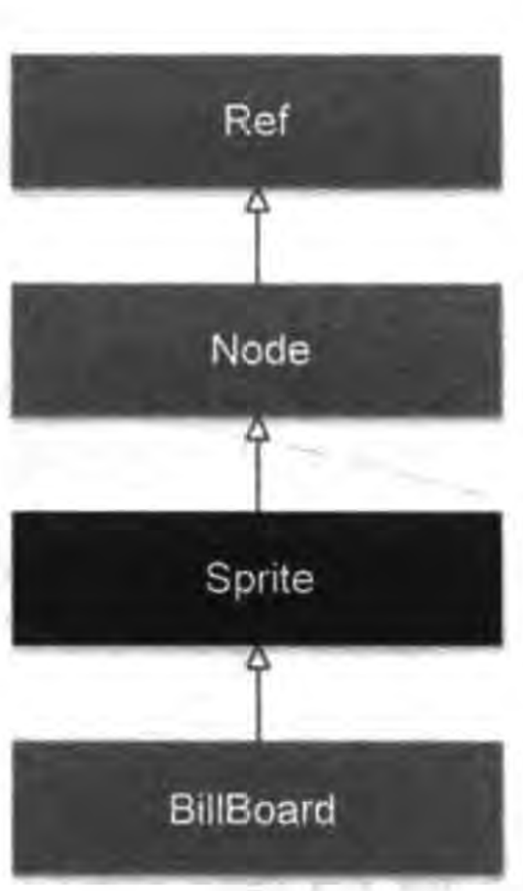


图 6-1　Sprite 类图

6.1.1 创建 Sprite 精灵对象

创建精灵对象有多种方式，其中常用的函数如下：

(1) cc.Sprite:create ()。创建一个精灵对象，纹理[①]等属性需要在创建后设置。

(2) cc.Sprite:create (filename)。指定图片创建精灵。

(3) cc.Sprite:create (filename, rect)。指定图片和裁剪的矩形区域来创建精灵。

(4) cc.Sprite:createWithTexture (texture)。指定纹理创建精灵。

(5) cc.Sprite:createWithTexture(texture, rect, rotated=false)。指定纹理和裁剪的矩形区域来创建精灵，第三个参数表示是否旋转纹理，默认不旋转。

(6) cc.Sprite:createWithSpriteFrame(pSpriteFrame)。通过一个精灵帧对象创建另一

① 纹理(texture)，表示物体表面细节的一幅或几幅二维图形，也称纹理贴图，当把纹理按照特定的方式映射到物体表面上时能使精灵看上去更加真实。

个精灵对象。

(7) cc. Sprite:createWithSpriteFrameName (spriteFrameName)。通过指定帧缓存中精灵帧名创建精灵对象。

上述 create 函数在前面介绍过,而且 create 函数比较简单,就不再介绍了。

6.1.2　实例:使用纹理对象创建 Sprite 对象

使用纹理 Texture2D 对象创建 Sprite 对象是通过 createWithTexture 函数实现的。本节通过一个实例介绍纹理对象创建 Sprite 对象,如图 6-2 所示,其中地面上的草是放在背景(见图 6-3)中的,场景中的两棵树是从如图 6-4 所示的“树”纹理图片中截取出来的,图 6-5 展示了树的纹理坐标,注意它的坐标原点在左上角。

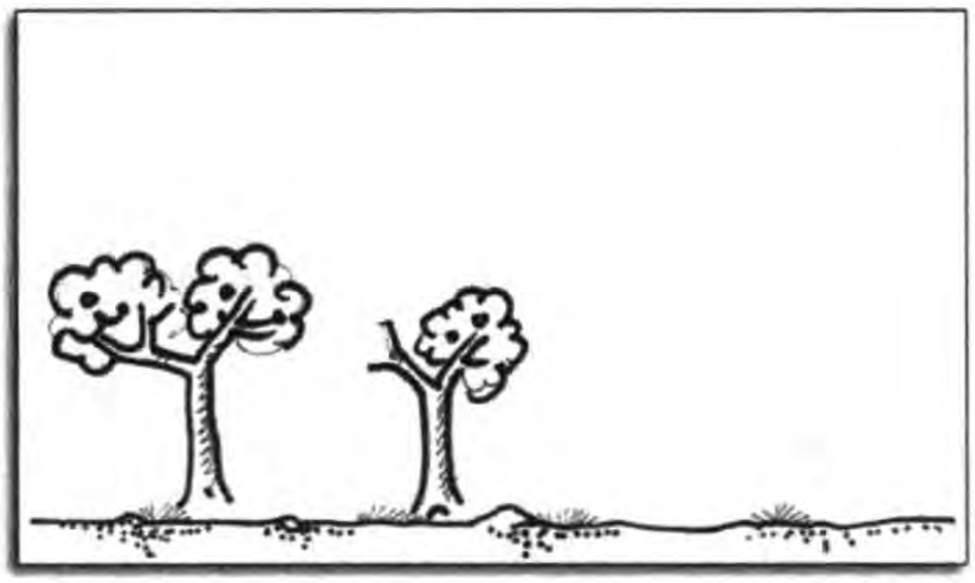


图 6-2　创建 Sprite 对象实例

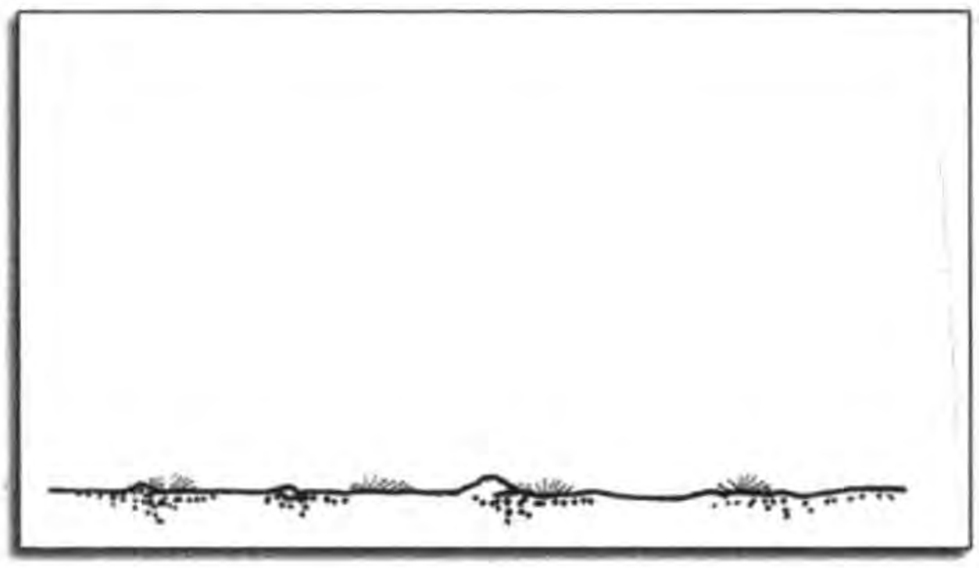


图 6-3　场景背景图片

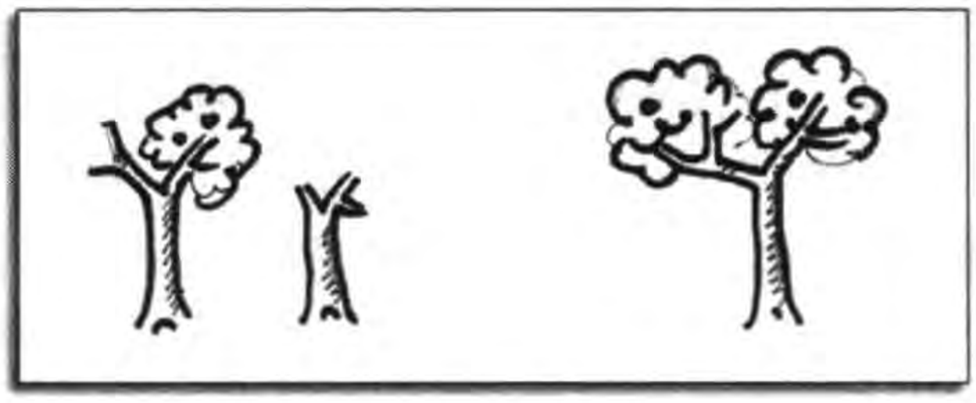


图 6-4　“树”纹理图片

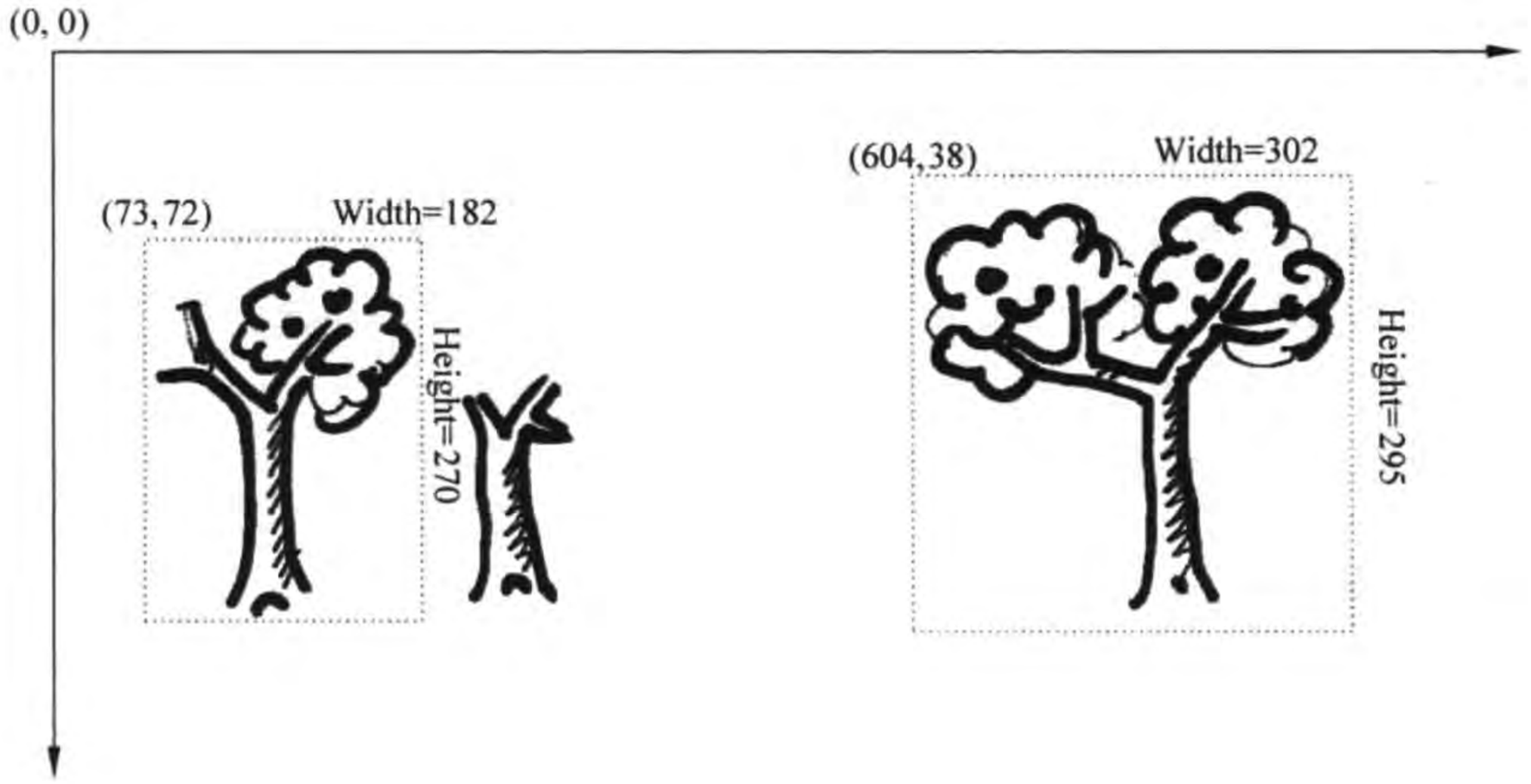


图 6-5　“树”纹理坐标

GameScene.lua 中 createLayer 函数代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local sprite = cc.Sprite:create("background.png")                                    ①
    sprite:setPosition(cc.p(size.width/2, size.height/2))                                ②
    layer:addChild(sprite)

    local tree1 = cc.Sprite:create("tree1.png",cc.rect(604, 38, 302, 295))               ③
    tree1:setPosition(cc.p(200,230))
    layer:addChild(tree1,0)

    local cache = cc.Director:getInstance():getTextureCache():addImage("tree1.png")      ④
    local tree2 = cc.Sprite:create()                                                     ⑤
    tree2:setTexture(cache)                                                              ⑥
    tree2:setTextureRect(cc.rect(73, 72,182,270))                                        ⑦
    tree2:setPosition(cc.p(500,200))
    layer:addChild(tree2,0)

    return layer
end
```

上述第①行代码 cc.Sprite:create("background.png")是通过 background.png 图片创建精灵，background.png 图片如图 6-3 所示。

第②行代码是设置背景的位置。

第③行代码 cc.Sprite:create("tree1.png",cc.rect(604, 38, 302, 295))通过 tree1.png 图片和矩形裁剪区域创建精灵，矩形裁剪区域为(604, 38, 302, 295)，如图 6-5 所示。

rect 类可以创建矩形裁剪区，rect 构造函数如下：

```
cc.rect(x, y, width, height)
```

其中 x，y 是 UI 坐标，坐标原点在左上角，width 是裁剪矩形的宽度，height 是裁剪矩形的高度。

第④行代码通过纹理缓存 TextureCache 创建纹理 Texture2D 对象，通过 Director 的 getTextureCache()函数可以获得 TextureCache 实例，TextureCache 的 addImage("tree1.png")函数可以创建纹理 Texture2D 对象，其中的 tree1.png 是纹理图片名。

第⑤行代码创建一个空的 Sprite 对象，还要通过后面的很多函数设置它的属性。

第⑥行代码 tree2:setTexture(cache)是设置纹理。

第⑦行代码 tree2:setTextureRect(cc.rect(73, 72,182,270))是设置纹理的裁剪区域。

6.2 精灵的性能优化

游戏是一种很耗费资源的应用，特别是在移动设备中的游戏，所以性能优化是非常重要的。本节只是介绍精灵相关的性能优化，其他方面的优化将会在第 19 章中再具体介绍。

精灵的性能优化可以使用精灵表和缓存来实现。下面从这两个方面介绍精灵的性能优化。

6.2.1 使用纹理图集

纹理图集(Texture Atlas)也称为精灵表(Sprite Sheet),它是把许多小的精灵图片组合到一张大图中。使用纹理图集(或精灵表)有如下优点:

(1) 减少文件读取次数,读取一张图片比读取一堆小文件要快。

(2) 减少 OpenGL ES 绘制调用并且加速渲染。

(3) 减少内存消耗。OpenGL ES 1.1 仅仅能够使用 2 的 n 次幂大小的图片(即宽度或者高度是 2、4、8、64……)。如果采用小图片 OpenGL ES 1.1 会分配给每个图片 2 的 n 次幂大小的内存空间,即使这张图片达不到这样的宽度和高度,也会分配大于此图片的 2 的 n 次幂大小的空间。那么运用这种图片集的方式将会减少内存碎片。虽然在 Cocos2d-x 2.0 后使用了 OpenGL ES 2.0,它不会再分配 2 的几次幂的内存块了,但是减少读取次数和绘制的优势依然存在。

(4) Cocos2d-x 全面支持 Zwoptex 精灵表制作工具(http://www.zwopple.com/zwoptex/)和 TexturePacker 精灵表制作工具(http://www.codeandweb.com/texturepacker),所以创建和使用纹理图集是很容易的。

通常可以使用纹理图集制作工具(Zwoptex 和 TexturePacker)来设计和生成纹理图集文件(见图 6-6),以及纹理图集坐标文件(plist)。

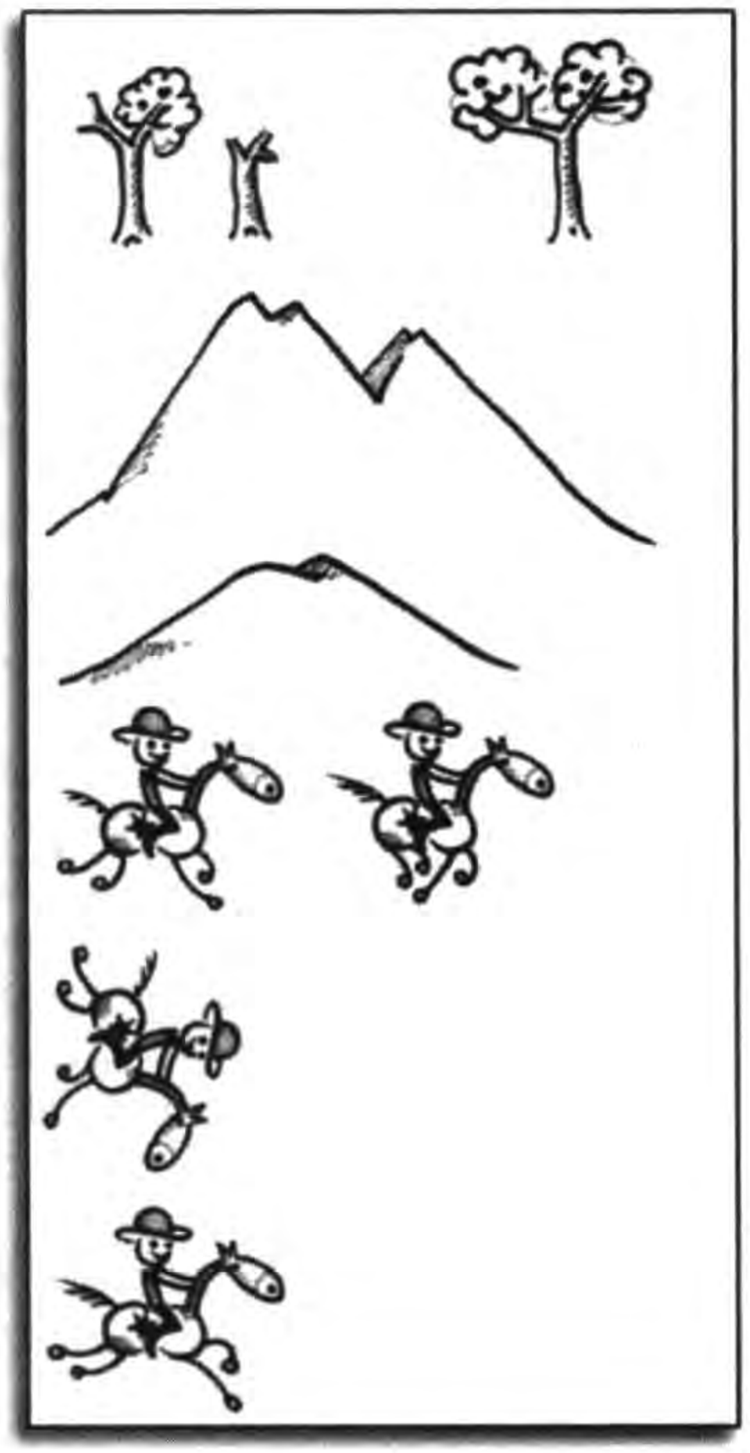


图 6-6 精灵表文件 SpirteSheet.png

plist 是属性列表文件,它是一种 XML 文件,SpirteSheet.plist 文件代码如下:

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<! DOCTYPE plist PUBLIC " - //Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList - 1.0.dtd">
<plist version = "1.0">
    <dict>
        <key> frames </key>
        <dict>                                                                    ①
            <key> hero1.png </key>                                                ②
            <dict>
                <key> frame </key>
                <string>{{2,1706},{391,327}}</string>                             ③
                <key> offset </key>
                <string>{6,0}</string>
                <key> rotated </key>
```

```
            <false/>
            <key> sourceColorRect </key>
            <string>{{17,0},{391,327}}</string>
            <key> sourceSize </key>
            <string>{413,327}</string>                                    ④
    </dict>
    ……
    <key> mountain1.png </key>
    <dict>
        <key> frame </key>
        <string>{{2,391},{934,388}}</string>
        <key> offset </key>
        <string>{0, -8}</string>
        <key> rotated </key>
        <false/>
        <key> sourceColorRect </key>
        <string>{{0,16},{934,388}}</string>
        <key> sourceSize </key>
        <string>{934,404}</string>
    </dict>
    …… ……
</dict>
<key> metadata </key>
<dict>
    <key> format </key>
    <integer> 2 </integer>
    <key> realTextureFileName </key>
    <string> SpirteSheet.png </string>
    <key> size </key>
    <string>{1024,2048}</string>
    <key> smartupdate </key><string> $ TexturePacker:SmartUpdate:5f186491d3aea289
c50ba9b77716547f:abc353d00773c0ca19d20b55fb028270:755b0266068b8a3b8dd250a2d186c02b $
</string>
    <key> textureFileName </key>
    <string> SpirteSheet.png </string>
  </dict>
</dict>
</plist>
```

上述代码是 plist 文件,其中第①～④行代码描述了一个精灵帧(小的精灵图片)位置。第②行代码是精灵帧的名字,一般情况下它的命名与原始的精灵图片名相同。

第③行代码描述了精灵帧的位置和大小,{2,1706}是精灵帧的位置,{391,327}是精灵帧的大小。由于不需要自己编写 plist 文件,其他的属性就不再介绍了。

提示 Zwoptex 和 TexturePacker 等纹理图集工具的使用请读者参考本系列丛书的工具卷(《Cocos2d-x 实战：工具卷》)。

使用精灵表文件最简单的方式是 Sprite 的 create (filename, rect)函数,其中创建矩形 rect 对象可以参考坐标文件中第③行代码的{{2,1706},{391,327}}数据。使用 create 函数

代码如下：

```
local mountain1 = cc.Sprite:create("SpirteSheet.png",cc.rect(2,391, 934, 388))
mountain1:setAnchorPoint(cc.p(0, 0))
mountain1:setPosition(cc.p( - 200,80))
layer:addChild(mountain1,0)
```

创建纹理 Texture2D 对象也可以使用精灵表文件，代码如下：

```
local cache = cc.Director:getInstance():getTextureCache():addImage("SpirteSheet.png")
local hero1 = cc.Sprite:create()
hero1:setTexture(cache)
hero1:setTextureRect(cc.rect(2,1706,391,327))                                    ①
hero1:setPosition(cc.p(800,200))
layer:addChild(hero1,0)
```

上述第①行代码中的 setTextureRect 函数是使用坐标文件中描述的数据。

6.2.2 使用精灵帧缓存

缓存的分类如下：

(1) 纹理缓存(TextureCache)。使用纹理缓存可以创建纹理对象。

(2) 精灵帧缓存(SpriteFrameCache)。能够从精灵表中创建精灵帧缓存，然后再从精灵帧缓存中获得精灵对象，反复使用精灵对象可以节省内存消耗。

(3) 动画缓存(AnimationCache)。动画缓存主要用于精灵动画，精灵动画中的每一帧都是从动画缓存中获取的。

本节主要介绍精灵帧缓存(SpriteFrameCache)，使用精灵帧缓存涉及的类有 SpriteFrame 和 SpriteFrameCache。使用 SpriteFrameCache 创建精灵对象的主要代码如下：

```
cc.SpriteFrameCache:getInstance():addSpriteFrames("SpirteSheet.plist")           ①
local mountain1 = cc.Sprite:createWithSpriteFrameName("mountain1.png")           ②
```

上述第①行代码是通过 SpriteFrameCache 创建精灵帧缓存对象，它是采用单例设计模式进行设计的，getInstance()函数可以获得 SpriteFrameCache 单一实例，addSpriteFrames 函数是将精灵帧添加到缓存中，其中 SpirteSheet. plist 是坐标文件。可以多次调用 addSpriteFrames 函数，以添加更多的精灵帧到缓存中。

第②行代码是通过 Sprite 的 createWithSpriteFrameName 函数来创建精灵对象，其中参数 mountain1. png 是 SpirteSheet. plist 坐标文件中定义的精灵帧名(见 SpirteSheet. plist 文件中的第②行代码)。

下面通过一个实例来介绍精灵帧缓存的使用，如图 6-7 所示，在游戏场景中有背景、山和英雄[①]三个精灵。

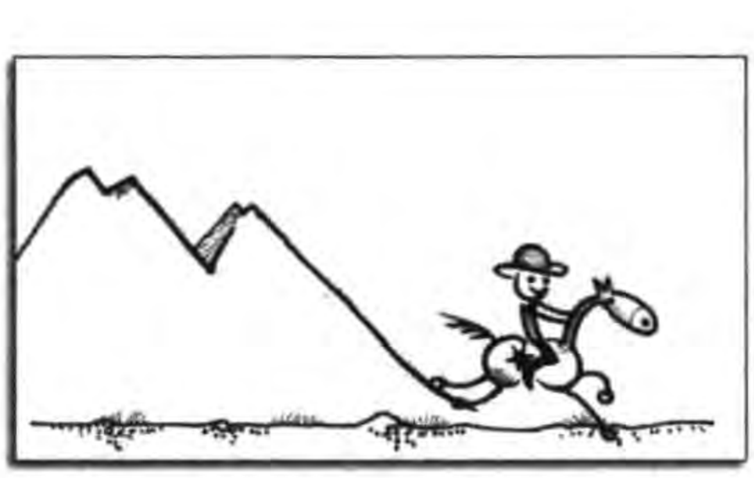


图 6-7 使用精灵帧缓存实例

① 我们把玩家控制的精灵称为“英雄”，把电脑控制的反方精灵称为“敌人”。

GameScene.lua 中 createLayer 函数代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local sprite = cc.Sprite:create("background.png")                                    ①
    sprite:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(sprite)

    local frameCache = cc.SpriteFrameCache:getInstance()                                 ②
    frameCache:addSpriteFrames("SpirteSheet.plist")                                      ③

    local mountain1 = cc.Sprite:createWithSpriteFrameName("mountain1.png")               ④
    mountain1:setAnchorPoint(cc.p(0, 0))
    mountain1:setPosition(cc.p(-200,80))
    layer:addChild(mountain1,0)

    local heroSpriteFrame = frameCache:getSpriteFrameByName("hero1.png")                 ⑤
    local hero1 = cc.Sprite:createWithSpriteFrame(heroSpriteFrame)                       ⑥
    hero1:setPosition(cc.p(800,200))
    layer:addChild(hero1,0)

    return layer
end
```

上述第①行代码是创建一个背景精灵对象，这个背景精灵对象并不是通过精灵缓存创建的，而是直接通过精灵文件创建的，事实上也可以将这个背景图片放到精灵表中。

第②行代码是获得精灵缓存对象，第③行代码是通过 addSpriteFrames 函数为精灵缓存添加精灵帧。在前面的介绍中使用一条语句 cc.SpriteFrameCache:getInstance():addSpriteFrames("SpirteSheet.plist")替代第②和③行两条语句。在这里分成两条语句是因为后面还要使用 frameCache 变量。

第④行代码是使用 Sprite 的 createWithSpriteFrameName 函数创建精灵对象，其中参数是精灵帧的名字。

第⑤和第⑥行代码是使用精灵缓存创建精灵对象的另外一种函数，其中第⑤行代码是使用精灵缓存对象 frameCache 的 getSpriteFrameByName 函数创建 SpriteFrame 对象，SpriteFrame 对象就是“精灵帧”对象，事实上在精灵缓存中存放的都是这种类型的对象。第⑥行代码是通过精灵帧对象创建的。第⑤和⑥行代码使用精灵缓存的方式主要应用于精灵动画，相关的知识将在精灵动画部分详细介绍。

精灵缓存不再使用后要移除相关精灵帧，否则再有相同名称的精灵帧，就会出现一些奇怪的现象。移除精灵帧的缓存函数如下：

(1) removeSpriteFrameByName(name)。指定具体的精灵帧名移除。

(2) removeSpriteFrames()。指定移除精灵缓存。

(3) removeSpriteFramesFromFile(plist)。指定具体的坐标文件移除精灵帧。

(4) removeUnusedSpriteFrames()。移除没有使用的精灵帧。

读者不能简单地移除精灵缓存中的精灵帧，这个问题将会在第 16 章(TODO)具体介绍。

本章小结

通过对本章的学习，使读者了解 Cocos2d-x Lua API 中精灵的相关知识和如何创建精灵对象。此外，还介绍了精灵的性能优化，性能优化方式包括使用精灵表和精灵帧缓存。

第7章

场景与层

前面简单地介绍了场景与层对象，本章将更加深入地介绍场景切换和层的生命周期问题。多个场景必然涉及场景切换、场景过渡动画和层的生命周期等相关知识。

7.1 场景与层的关系

虽然前面介绍了场景和层，但是本节还要深入介绍场景与层的关系。在第4章中介绍了节点的层级结构，在节点的层级结构中可以看到场景与层之间的关系如图7-1所示，即一个场景(Scene)与多个层(Layer)对应，而且层的个数至少是1，不能为0。

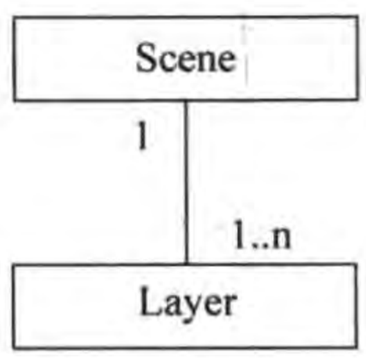


图7-1 场景与层的对应关系

7.2 场景切换

前面介绍的实例都是单个场景，但是实际游戏应用中往往涉及多个场景，多个场景必然需要切换。

7.2.1 场景切换相关函数

场景切换是通过导演类Director实现的，其中的相关函数如下：

(1) runWithScene(scene)。该函数可以运行场景，但是只能在启动第一个场景时调用该函数，如果已经有一个场景运行则不能调用该函数。

(2) replaceScene(scene)。切换到下一个场景。用一个新的场景替换当前场景，当前场景被终端释放。

(3) pushScene(scene)。切换到下一个场景。将当前场景挂起放入到场景堆栈中,然后再切换到下一个场景中。

(4) popScene()。与 pushScene 配合使用,可以返回到上一个场景。

(5) popToRootScene()。与 pushScene 配合使用,可以返回到根场景。

注意 replaceScene 和 pushScene 的区别：replaceScene 会释放和销毁场景,如果需要保持原来场景的状态,replaceScene 函数则不适用；pushScene 并不会释放和销毁场景,原来场景的状态可以保持,但是游戏中不能同时有太多的场景对象运行。

使用 replaceScene 代码如下：

```
local SettingScene = require("SettingScene")
local settingScene = SettingScene.create()
cc.Director:getInstance():replaceScene(settingScene)
```

其中 Setting 是下一个要切换的场景。使用 pushScene 代码如下：

```
local SettingScene = require("SettingScene")
local settingScene = SettingScene.create()
cc.Director:getInstance():pushScene(settingScene)
```

从 Setting 场景回到上一个场景使用代码如下：

```
Director::getInstance()->popScene();
```

下面用实例介绍场景切换相关函数。图 7-2 中有 2 个场景：GameScene 和 SettingScene(设置)。在 GameScene 场景单击“游戏设置”菜单可以切换到 SettingScene 场景,在 SettingScene 场景中单击 OK 按钮可以返回到 GameScene 场景。

首先需要在工程中添加一个 SettingScene 场景(SettingScene.lua),可以使用任何文本编辑工具添加 SettingScene.lua 文件。

在 GameScene.lua 中编写代码如下：

```
require "Cocos2d"
require "Cocos2dConstants"

local SettingScene = require("SettingScene")

local size = cc.Director:getInstance():getWinSize()

local GameScene = class("GameScene",function()
    returncc.Scene:create()
end)

function GameScene.create()
    local scene = GameScene.new()
    scene:addChild(scene:createLayer())
    return scene
end
```

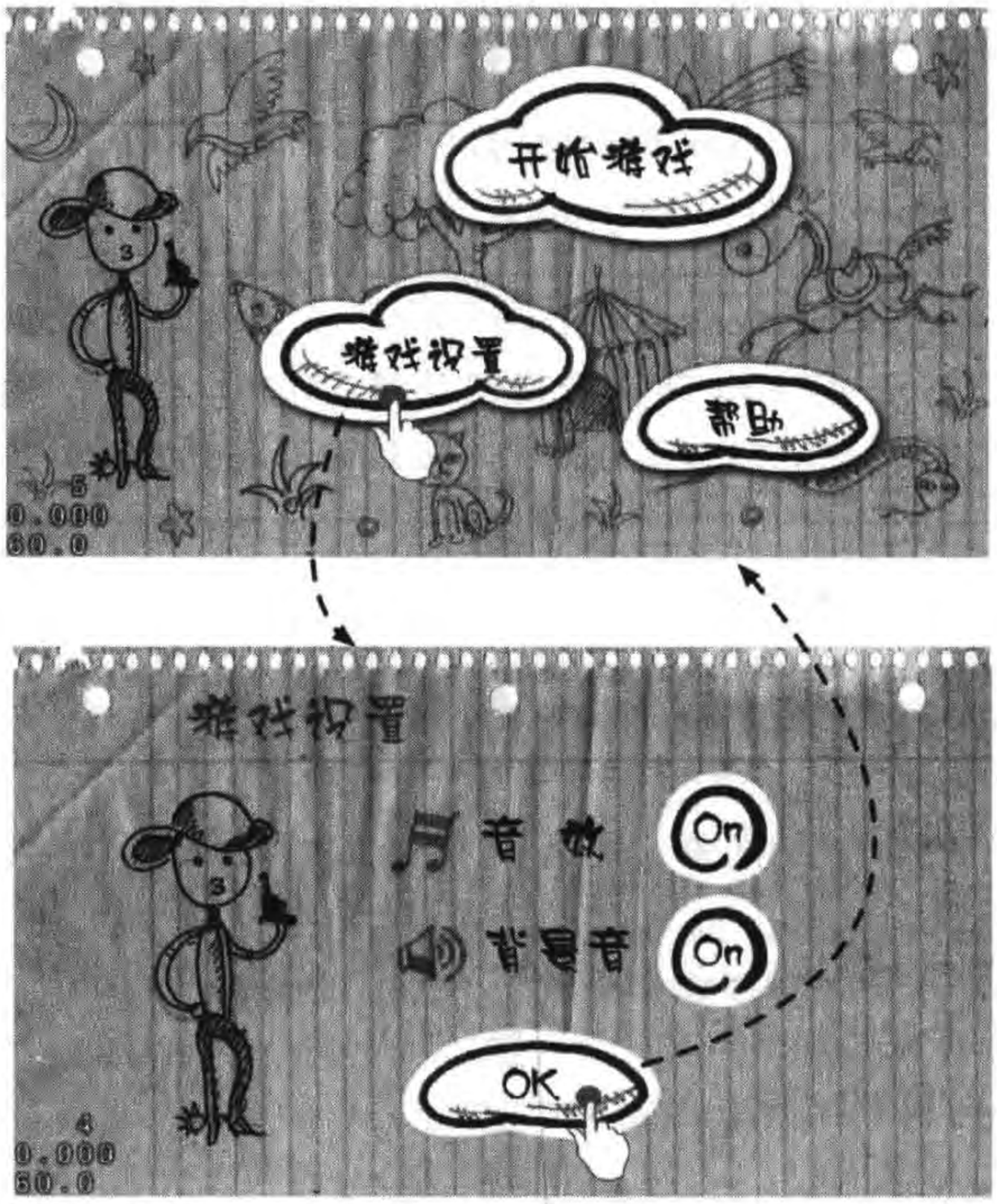


图 7-2　场景之间的切换

（上图为 GameScene 场景，下图为 SettingScene 场景）

```
function GameScene:ctor()
end

 -- create layer
function GameScene:createLayer()

    local layer = cc.Layer:create()
    local director = cc.Director:getInstance()

    local bg = cc.Sprite:create("background.png")
    bg:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(bg)

    -- 开始精灵
    local startlocalNormal = cc.Sprite:create("start - up.png")
    local startlocalSelected = cc.Sprite:create("start - down.png")
    local startMenuItem = cc.MenuItemSprite:create(startlocalNormal, startloca lSelected)
    startMenuItem:setPosition(director:convertToGL(cc.p(700, 170)))
    local function menuItemStartCallback(sender)
        cclog("Touch Start.")
```

```
end
startMenuItem:registerScriptTapHandler(menuItemStartCallback)

-- 设置图片菜单
local settingMenuItem = cc.MenuItemImage:create("setting-up.png", "setting-down.png")
settingMenuItem:setPosition(director:convertToGL(cc.p(480, 400)))
local function menuItemSettingCallback(sender)                                    ①
    cclog("Touch Setting.")
    local settingScene = SettingScene.create()                                    ②
    //director:replaceScene(settingScene)                                         ③
    director:pushScene(settingScene)                                              ④
end
settingMenuItem:registerScriptTapHandler(menuItemSettingCallback)

-- 帮助图片菜单
local helpMenuItem = cc.MenuItemImage:create("help-up.png", "help-down.png")

helpMenuItem:setPosition(director:convertToGL(cc.p(860, 480)))
local function menuItemHelpCallback(sender)
    cclog("Touch Help.")
end
helpMenuItem:registerScriptTapHandler(menuItemHelpCallback)

localmn = cc.Menu:create(startMenuItem, settingMenuItem, helpMenuItem)
mn:setPosition(cc.p(0, 0))
layer:addChild(mn)

    return layer
end


return GameScene
```

上述第①行代码定义的函数 menuItemSettingCallback，是用来在用户单击“游戏设置”菜单时回调。

第②行代码是设置创建对象。

第③行代码是使用 replaceScene 函数进行场景切换。

第④行代码是使用 pushScene 函数进行场景切换。

SettingScene.lua 代码如下：

```
require "Cocos2d"
require "Cocos2dConstants"

local size = cc.Director:getInstance():getWinSize()

local SettingScene = class("SettingScene",function()
    return cc.Scene:create()
end)

function SettingScene.create()
    local scene = SettingScene.new()
```

```
    scene:addChild(scene:createLayer())
    return scene
end


function SettingScene:ctor()
end

-- create layer
function SettingScene:createLayer()
local layer = cc.Layer:create()
local director = cc.Director:getInstance()

local bg = cc.Sprite:create("setting-back.png")
bg:setPosition(cc.p(size.width/2, size.height/2))
layer:addChild(bg)

-- 音效
local soundOnMenuItem = cc.MenuItemImage:create("on.png", "on.png")
local soundOffMenuItem = cc.MenuItemImage:create("off.png", "off.png")
local soundToggleMenuItem = cc.MenuItemToggle:create(soundOnMenuItem,
                                    soundOffMenuItem)
soundToggleMenuItem:setPosition(director:convertToGL(cc.p(818, 220)))
local function menuSoundToggleCallback(sender)
    cclog("Sound Toggle.")
end
soundToggleMenuItem:registerScriptTapHandler(menuSoundToggleCallback)

-- 音乐
local musicOnMenuItem = cc.MenuItemImage:create("on.png", "on.png")
local musicOffMenuItem = cc.MenuItemImage:create("off.png", "off.png")
local musicToggleMenuItem = cc.MenuItemToggle:create(musicOnMenuItem,
                                    musicOffMenuItem)
musicToggleMenuItem:setPosition(director:convertToGL(cc.p(818, 362)))
local function menuMusicToggleCallback(sender)
    cclog("Music Toggle.")
end
musicToggleMenuItem:registerScriptTapHandler(menuMusicToggleCallback)

-- Ok 按钮
local okMenuItem = cc.MenuItemImage:create(
    "ok-down.png",
    "ok-up.png")
okMenuItem:setPosition(director:convertToGL(cc.p(600, 510)))
local function menuOkCallback(sender)                                    ①
    cclog("Ok Menu tap.")
    director:popScene()                                                  ②
end
okMenuItem:registerScriptTapHandler(menuOkCallback)
```

```
    local mn = cc.Menu:create(soundToggleMenuItem, musicToggleMenuItem,okMenuItem)
    mn:setPosition(cc.p(0, 0))
    layer:addChild(mn)

    return layer
end

return SettingScene
```

上述第①行代码定义的函数 menuOkCallback，是用来在设置场景用户单击 OK 菜单时回调。

第②行代码是使用 popScene 函数返回到 GameScene 场景。

7.2.2　场景过渡动画

场景切换时可以添加过渡动画，场景过渡动画是由 TransitionScene 类和它的子类展示的。TransitionScene 类图如图 7-3 所示。

从图 7-3 所示的类图中可以看到，TransitionScene 类的直接子类有 13 个，而且有些子类还有子类，全部的过渡动画类大概有 30 多个。幸运的是过渡动画类使用方式大同小异，均类似如下代码：

```
local settingScene = SettingScene.create()
local ts = cc.TransitionJumpZoom:create(1.0, settingScene)                    ①
cc.Director:getInstance():pushScene(ts)                                       ②
```

第①行代码是创建过渡动画 TransitionScene 对象，它的 create 函数有两个参数：一是动画持续时间；二是场景对象。

第②行代码中 pushScene 函数使用的参数是过渡动画 TransitionScene 对象，而不是场景对象。

这里不介绍全部 30 多个过渡动画，只介绍一些有代表性的过渡动画，如下：

(1) TransitionFadeTR。网格过渡动画，从左下到右上。

(2) TransitionJumpZoom。跳动的过渡动画。

(3) TransitionCrossFade。交叉渐变过渡动画。

(4) TransitionMoveInL。从左边推入覆盖的过渡动画。

(5) TransitionShrinkGrow。放缩交替的过渡动画。

(6) TransitionRotoZoom。类似照相机镜头旋转放缩交替的过渡动画。

(7) TransitionSlideInL。从左侧推入的过渡动画。

(8) TransitionSplitCols。按列分割界面的过渡动画。

(9) TransitionSplitRows。按行分割界面的过渡动画。

(10) TransitionTurnOffTiles。生成随机瓦片方格的过渡动画。

很多动画效果需用心体会，需要广大读者自己运行起来观察。

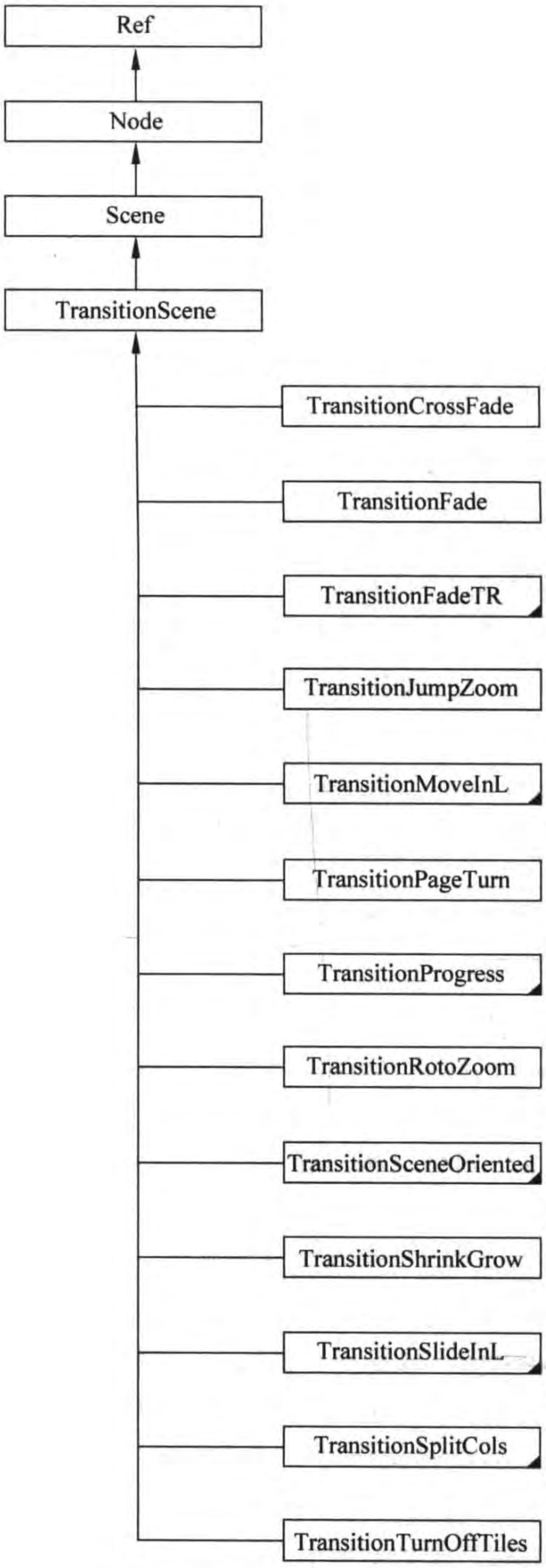


图 7-3　TransitionScene 类图

7.3　场景的生命周期

游戏运行过程中场景(Scene)是有生命周期的，不同的生命周期会触发不同的事件，例如可以在进入场景事件过程中做一些初始化处理，而在退出场景事件过程中释放一些资源。

7.3.1　生命周期函数

场景(Scene)以及所有节点(Node)的生命周期事件如下：

(1) enter。进入场景时触发。

(2) enterTransitionFinish。进入场景并且过渡动画结束时触发。

(3) exit。退出场景时触发。

(4) exitTransitionStart。退出场景并且开始过渡动画时触发。

(5) cleanup。场景对象被清除时触发。

提示　GameScene 场景中的场景(Scene)继承于节点(Node)，这些生命周期事件根本上是从 Node 继承而来。事实上所有 Node 对象(包括场景、层、精灵等)都有这些事件，具体实现代码与 GameScene 场景类似。

为 GameScene 场景添加生命周期函数相关代码如下：

```
require "Cocos2d"
require "Cocos2dConstants"

local SettingScene = require("SettingScene")
local size = cc.Director:getInstance():getWinSize()

local GameScene = class("GameScene",function()
    return cc.Scene:create()
end)

function GameScene.create()
    local scene = GameScene.new()
    scene:addChild(scene:createLayer())
    return scene
end

function GameScene:ctor()                                                    ①
    cclog("GameScene init")
    --场景生命周期事件处理
    local function onNodeEvent(event)
        if event == "enter" then                                             ②
            self:onEnter()
        elseif event == "enterTransitionFinish" then                         ③
            self:onEnterTransitionFinish()
        elseif event == "exit" then                                          ④
```

```
            self:onExit()
        elseif event == "exitTransitionStart" then                    ⑤
            self:onExitTransitionStart()
        elseif event == "cleanup" then                                ⑥
            self:cleanup()
        end
    end
    self:registerScriptHandler(onNodeEvent)                           ⑦
end

function GameScene:onEnter()                                          ⑧
    cclog("GameScene onEnter")
end

function GameScene:onEnterTransitionFinish()                          ⑨
    cclog("GameScene onEnterTransitionFinish")
end

function GameScene:onExit()                                           ⑩
    cclog("GameScene onExit")
end

function GameScene:onExitTransitionStart()                            ⑪
    cclog("GameScene onExitTransitionStart")
end

function GameScene:cleanup()                                          ⑫
    cclog("GameScene cleanup")
end

...

return GameScene
```

上述第①行代码 GameScene:ctor()是 GameScene 场景构造函数,在此函数中进行 GameScene 场景的初始化,也就是触发 init 事件。

第②～⑥行代码是场景生命周期事件处理,根据事件 event 对象判断是哪一种类型的场景生命周期事件,其中第②行代码是判断 enter 事件,是场景进入时触发,判断为 true 情况下,通过代码 self:onEnter()调用第⑧行的 GameScene:onEnter()函数。

第③行代码是判断 enterTransitionFinish 事件,是场景进入并且过渡动画完成时触发,判断为 true 情况下,通过代码 self:onEnterTransitionFinish()调用第⑨行的 GameScene:onEnterTransitionFinish()函数。

第④行代码是判断 exit 事件,是场景退出时触发,判断为 true 情况下,通过代码 self:onExit()调用第⑩行的 GameScene:onExit()函数。

第⑤行代码是判断 exitTransitionStart 事件,是场景退出并且过渡动画开始时触发,判断为 true 情况下,通过代码 self:onExitTransitionStart()调用第⑪行的 GameScene:onExitTransitionStart()函数。

第⑥行是判断 cleanup 事件，是场景销毁时触发，判断为 true 情况下，通过代码 self:cleanup()调用第⑫行的 GameScene:cleanup()函数。

如果 GameScene 是第一个场景，当启动 GameScene 场景时，它的调用顺序如图 7-4 所示。

提示　GameScene init 事件并不是从节点继承的事件(enter、enterTransitionFinish、exit、exitTransitionStart 和 cleanup)，“GameScene init”日志信息是在 GameScene:ctor()构造函数中输出的。本书把 ctor()构造函数调用与其他的几个事件调用放在一起讨论，因为 ctor()构造函数是用于初始化场景的，它相当于 init 事件的作用，因此本书将 ctor()构造函数调用都称为“init 事件”触发。

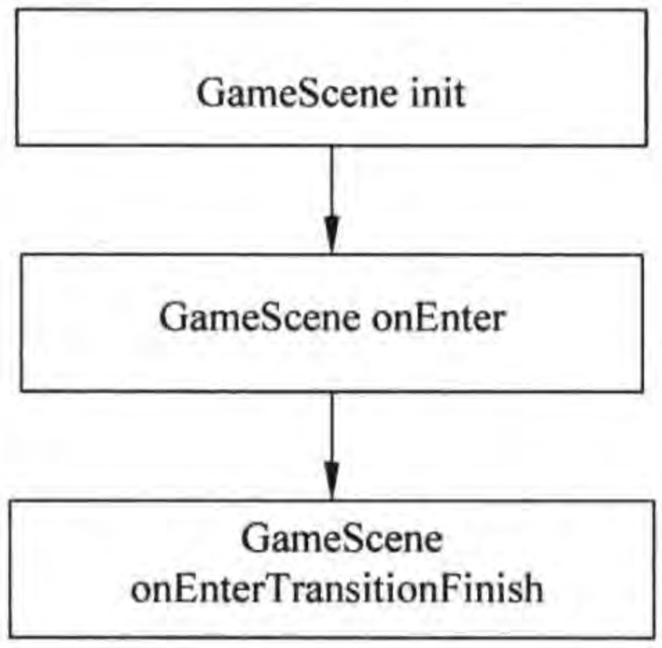


图 7-4　第一个场景启动顺序

7.3.2　多场景切换生命周期

在多个场景切换时，场景的生命周期会更加复杂。本节介绍多场景切换生命周期。多个场景切换时分为以下几种情况：

(1) 使用 pushScene 函数实现从 GameScene 场景进入 SettingScene 场景。

(2) 使用 replaceScene 函数实现从 GameScene 场景进入 SettingScene 场景。

(3) 使用 popScene 函数实现从 SettingScene 场景返回到 GameScene 场景。

参考 GameScene 重写 SettingScene 中的几个生命周期函数，代码如下：

```
function SettingScene:ctor()
    -- self.visibleSize = cc.Director:getInstance():getVisibleSize()
    cclog("SettingScene init")
    --场景节点事件处理
    local function onNodeEvent(event)
        if event == "enter" then
            self:onEnter()
        elseif event == "enterTransitionFinish" then
            self:onEnterTransitionFinish()
        elseif event == "exit" then
            self:onExit()
        elseif event == "exitTransitionStart" then
            self:onExitTransitionStart()
        elseif event == "cleanup" then
            self:cleanup()
        end
    end

    self:registerScriptHandler(onNodeEvent)
end
```

```
function SettingScene:onEnter()
    cclog("SettingScene onEnter")
end

function SettingScene:onEnterTransitionFinish()
    cclog("SettingScene onEnterTransitionFinish")
end

function SettingScene:onExit()
    cclog("SettingScene onExit")
end

function SettingScene:onExitTransitionStart()
    cclog("SettingScene onExitTransitionStart")
end

function SettingScene:cleanup()
    cclog("SettingScene cleanup")
end
```

(1) 当使用 pushScene 函数时,生命周期函数的调用顺序如图 7-5 所示。

(2) 当使用 replaceScene 函数时,生命周期函数的调用顺序如图 7-6 所示,与图 7-5 不同的是多出 GameScene 中 cleanup 事件,这也说明 replaceScene 函数会释放场景对象。

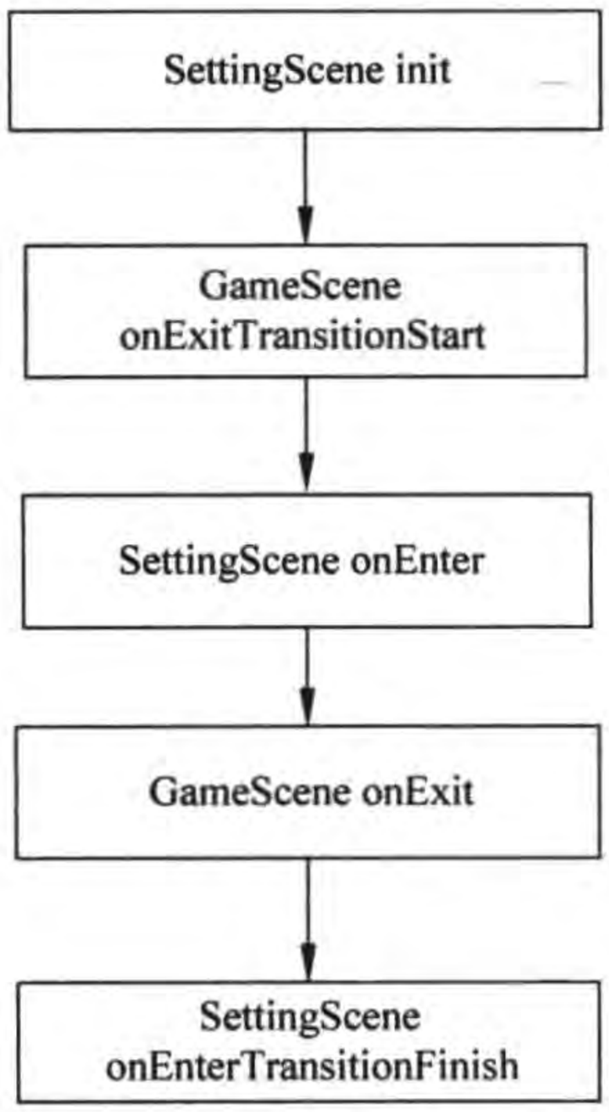


图 7-5 使用 pushScene 函数时的生命周期事件顺序

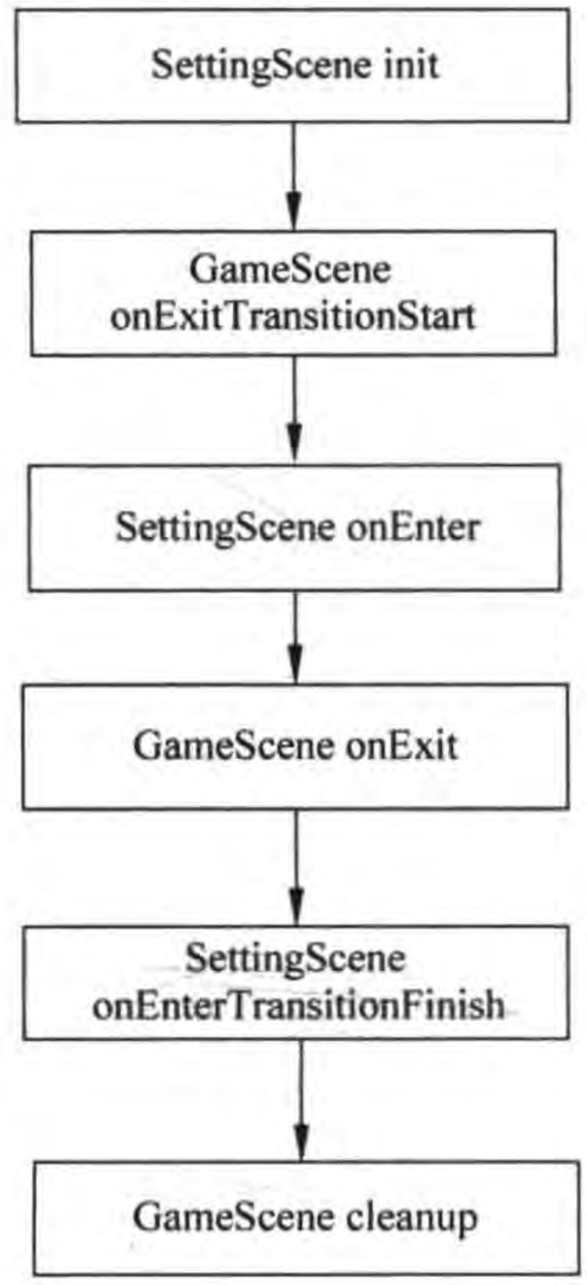


图 7-6 使用 replaceScene 函数时的生命周期事件顺序

（3）使用 popScene 函数时，生命周期函数的调用顺序如图 7-7 所示，从图中可见调用 popScene 函数时触发 SettingScene 的 cleanup 事件，这说明 popScene 函数会释放 SettingScene 场景对象，当回到 GameScene 场景时并不会触发 init 事件，而是触发 enter 事件。

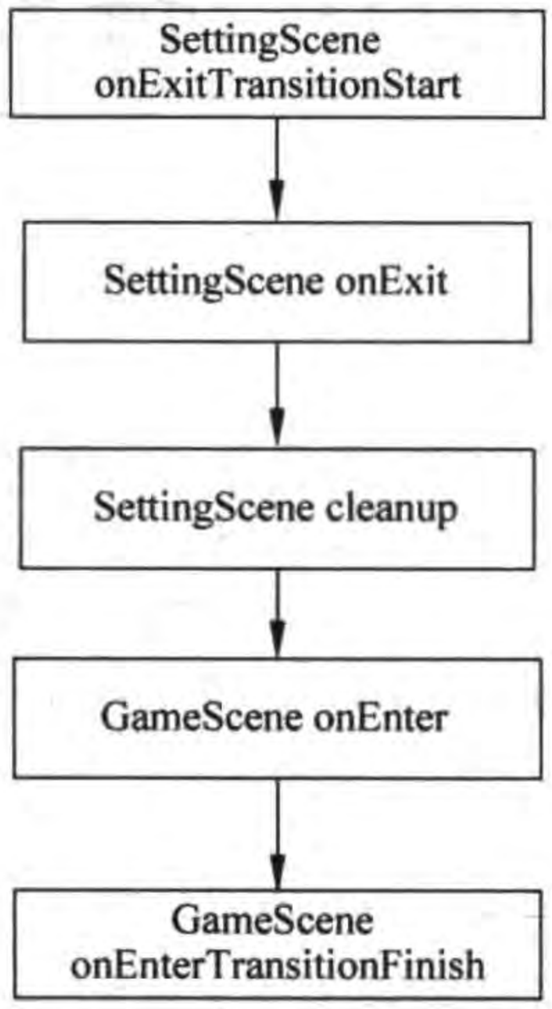


图 7-7　使用 popScene 函数时的生命周期事件顺序

本章小结

通过对本章的学习使广大读者掌握场景和层等概念，重点是场景和层的关系、场景的生命周期和场景之间的切换。

第 8 章

动作、特效和动画

游戏的世界是一个动态的世界，无论是玩家控制精灵还是非玩家控制精灵，甚至背景都可能是动态的。在 Cocos2d-x Lua API 中的 Node 对象可以有动作、特效和动画等动态特性。因为在 Node 类中定义了这些动态特性，因此精灵、标签、菜单、地图和粒子系统等都具有这些动态特性。本章介绍 Cocos2d-x Lua API 中的动作、特效和动画。

8.1 动作

动作(Action)包括了基本动作和基本动作的组合，这些基本动作有缩放、移动、旋转等，而且这些动作变化的速度也可以设定。

动作类是 Action，类图如图 8-1 所示。

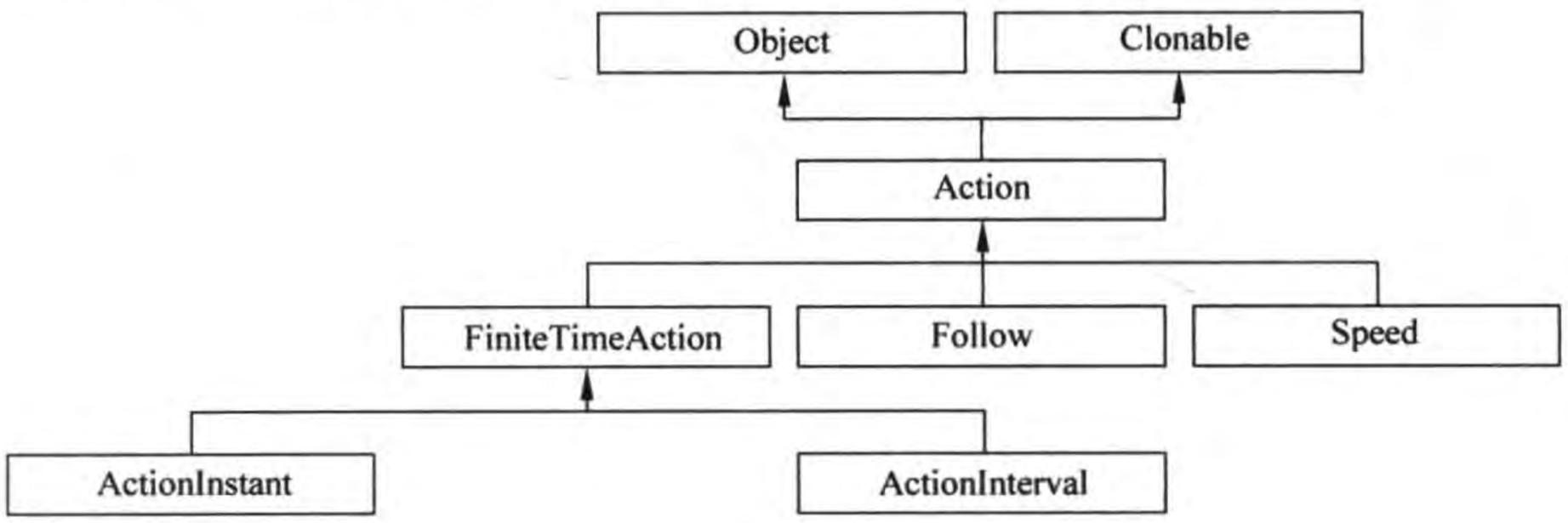


图 8-1 动作类图

从图 8-1 中可以看出，Action 的一个子类是 FiniteTimeAction，FiniteTimeAction 是一种受时间限制的动作，Follow 是一种允许精灵跟随另一个精灵的动作，Speed 是在一个动作运行时改变其运动速率。

此外，FiniteTimeAction 有两个子类：ActionInstant 和 ActionInterval。ActionInstant 和 ActionInterval 是两种不同风格的动作类，ActionInstant 封装了一种瞬时动作，ActionInterval 封装了一种间隔动作。

Node 类中关于动作的函数如下：

(1) runAction(Action * action)。运行指定动作，返回值仍然是一个动作对象。

(2) stopAction(Action * action)。停止指定动作。

(3) stopActionByTag(int tag)。通过指定标签停止动作。

(4) stopAllActions()。停止所有动作。

8.1.1　瞬时动作

瞬时动作就是不等待、马上执行的动作。瞬时动作的基类是 ActionInstant，瞬时动作 ActionInstant 类图如图 8-2 所示。

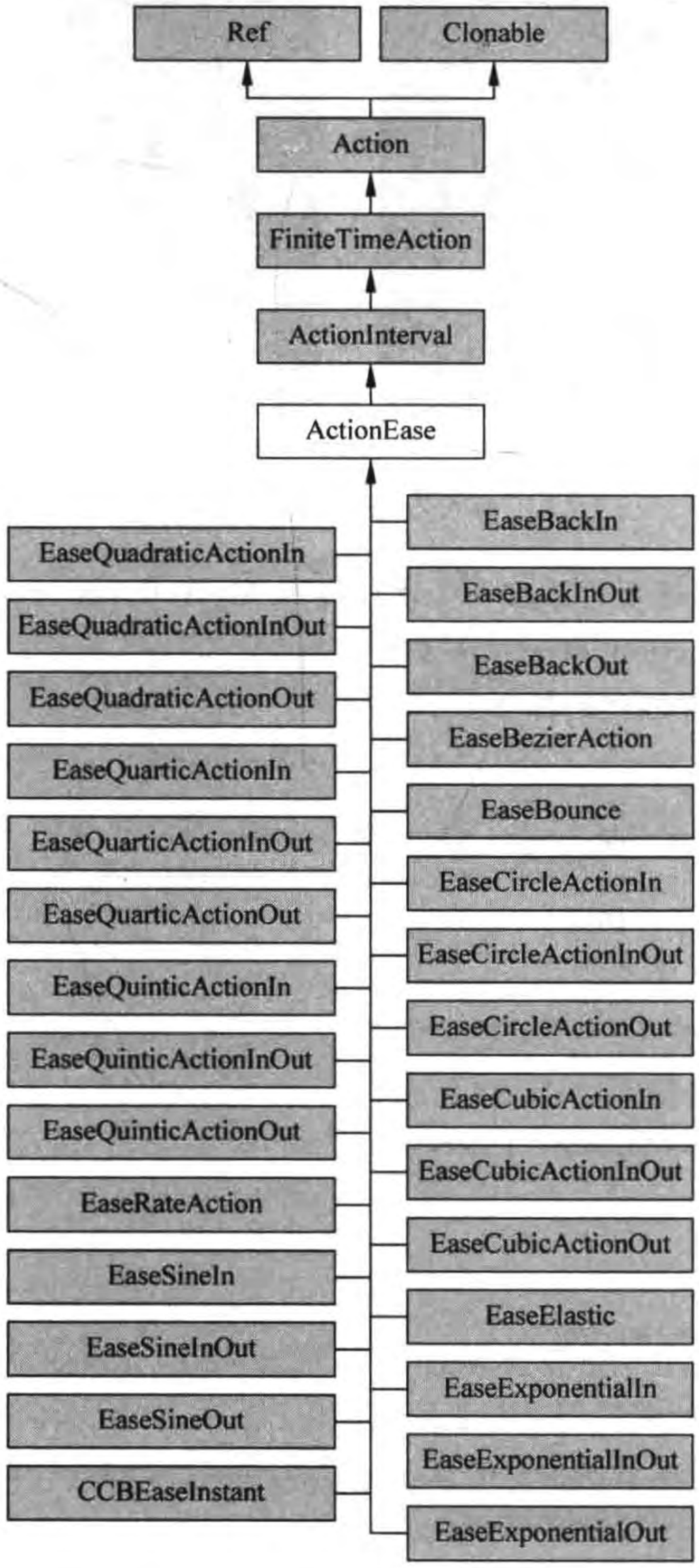
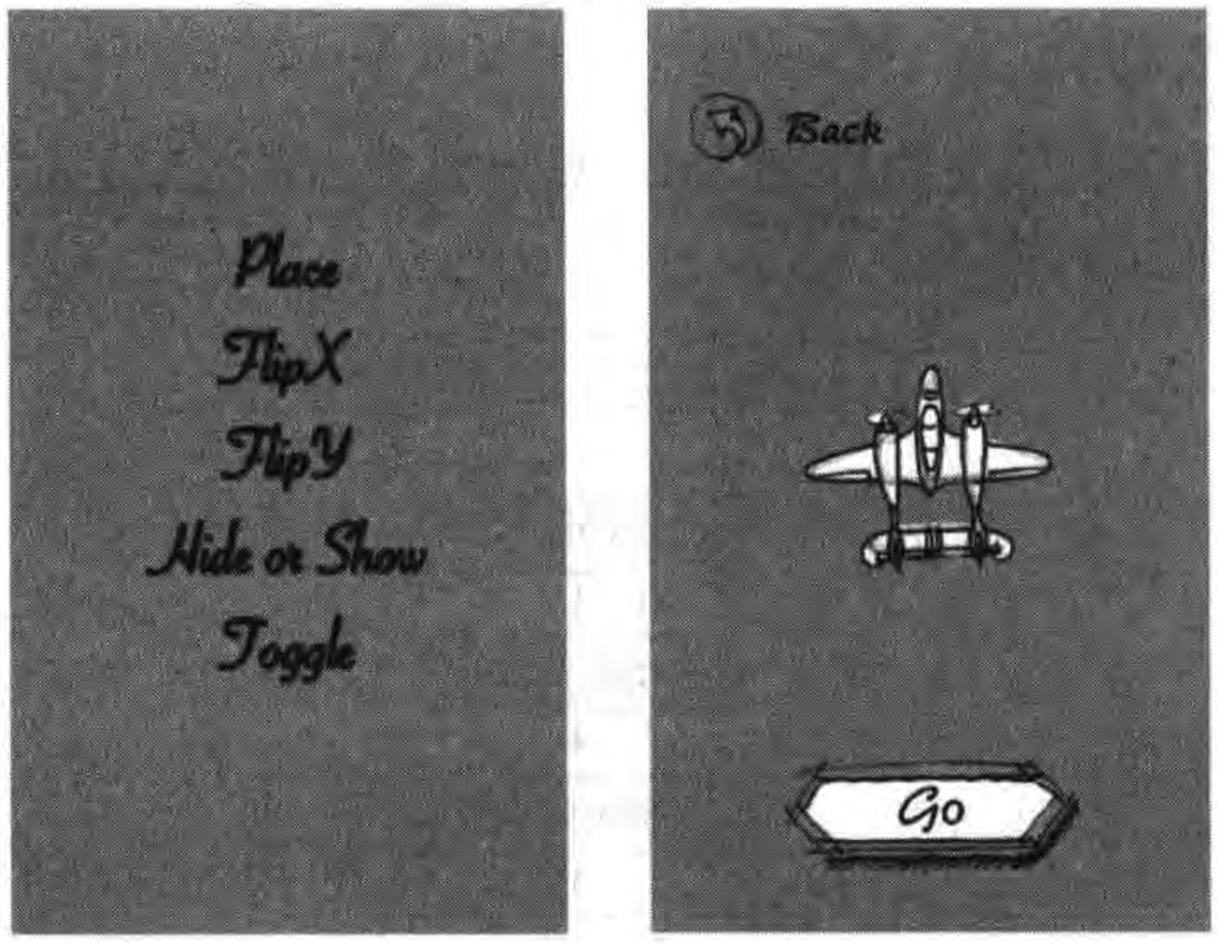


图 8-2　瞬时动作类图

下面通过实例介绍瞬时动作的使用，如图 8-3 所示，图(a)是一个操作菜单场景，选择菜单可以进入到图(b)所示的动作场景，在图(b)动作场景中单击 Go 按钮可以执行选择的动作效果，单击 Back 按钮可以返回到菜单场景。



(a) (b)

图 8-3 瞬时动作实例

由于需要两个场景，还要再添加一个动作场景 MyActionScene，具体添加过程可以通过文本编辑工具创建 MyActionScene.lua 文件。

GameScene.lua 文件的代码如下：

```
-- 定义常量
PLACE_TAG              = 102                                    ①
FLIPX_TAG              = 103
FLIPY_TAG              = 104
HIDE_SHOW_TAG          = 105
TOGGLE_TAG             = 106                                    ②

--操作标识
actionFlag =  -1                                                ③

size = cc.Director:getInstance():getWinSize()
...

-- create layer
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local bg = cc.Sprite:create("Background.png")
    bg:setPosition(cc.p(size.width/2,
        size.height/2))
    layer:addChild(bg)
```

```
local function OnClickMenu(tag, menuItemSender)                              ④
    cclog("tag =  %d", tag)
    actionFlag = menuItemSender:getTag()                                     ⑤

    local scene = require("MyActionScene")                                   ⑥
    local nextScene = scene.create()                                         ⑦
    local ts = cc.TransitionJumpZoom:create(1, nextScene)
    cc.Director:getInstance():pushScene(ts)

end

local placeLabel = cc.Label:createWithBMFont("fonts/fnt2.fnt", "Place")      ⑧
local placeMenu = cc.MenuItemLabel:create(placeLabel)                        ⑨
placeMenu:setTag(PLACE_TAG)                                                  ⑩
placeMenu:registerScriptTapHandler(OnClickMenu)                              ⑪

local flipXLabel = cc.Label:createWithBMFont("fonts/fnt2.fnt", "FlipX")
local flipXMenu = cc.MenuItemLabel:create(flipXLabel)
flipXMenu:setTag(FLIPX_TAG)
flipXMenu:registerScriptTapHandler(OnClickMenu)

local flipYLabel = cc.Label:createWithBMFont("fonts/fnt2.fnt", "FlipY")
local flipYMenu = cc.MenuItemLabel:create(flipYLabel)
flipYMenu:setTag(FLIPY_TAG)
flipYMenu:registerScriptTapHandler(OnClickMenu)

local hideLabel = cc.Label:createWithBMFont("fonts/fnt2.fnt", "Hide or Show")
local hideMenu = cc.MenuItemLabel:create(hideLabel)
hideMenu:setTag(HIDE_SHOW_TAG)
hideMenu:registerScriptTapHandler(OnClickMenu)

local toggleLabel = cc.Label:createWithBMFont("fonts/fnt2.fnt", "Toggle")
local toggleMenu = cc.MenuItemLabel:create(toggleLabel)
toggleMenu:setTag(TOGGLE_TAG)
toggleMenu:registerScriptTapHandler(OnClickMenu)

local mn = cc.Menu:create(placeMenu,flipXMenu,flipYMenu, hideMenu, toggleMenu)
mn:alignItemsVertically()
layer:addChild(mn)

return layer
end

return GameScene
```

上述第①和第②行代码定义了 5 个常量，这 5 个常量对应 5 个菜单项。

第③行代码的操作标识用来保存用户单击的菜单项 tag 属性，用于下一个场景的判断。

第④行代码定义菜单回调函数 OnClickMenu，5 个菜单项目单击时都会调用这个函数。

第⑤行代码 actionFlag＝menuItemSender：getTag()是取出菜单项 tag 属性赋值给操作标识变量 actionFlag。

第⑥行代码 local scene＝require("MyActionScene")是加载 MyActionScene.lua 文件。

第⑦行代码 local nextScene＝scene.create()是创建 MyActionScene 场景对象。

第⑧行代码定义了位图标签对象 placeLabel。

第⑨行代码是创建菜单项 placeMenu。

第⑩行代码 placeMenu:setTag(PLACE_TAG)设置菜单项的 tag 属性，tag 属性是 Node 类中定义的，是一个整数，它可以为 Node 对象设置一个标识。依次为其他 4 个菜单项也设置 tag 属性。

第⑪行代码是注册菜单项单击事件，其参数就是前面定义的回调函数。

MyActionScene.lua 代码如下：

```
--精灵隐藏
local hiddenFlag = true                                                    ①

⋮

-- create layer
function MyActionScene:createLayer()
    cclog("MyActionScene actionFlag =  %d", actionFlag)
    local layer = cc.Layer:create()

    local bg = cc.Sprite:create("Background.png")
    bg:setPosition(cc.p(size.width/2,
        size.height/2))
    layer:addChild(bg)

    local sprite = cc.Sprite:create("Plane.png")
    sprite:setPosition(cc.p(size.width / 2, size.height / 2))
    layer:addChild(sprite)

    local backMenuItem = cc.MenuItemImage:create("Back - up.png","Back - down.png")
    backMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(120, 100)))

    local goMenuItem = cc.MenuItemImage:create("Go - up.png","Go - down.png")
    goMenuItem:setPosition(size.width/2, 100)

    local mn = cc.Menu:create(backMenuItem, goMenuItem)

    mn:setPosition(cc.p(0, 0))
    layer:addChild(mn)

    local function backMenu(pSender)                                       ②
        cclog("MyActionScene backMenu")
        cc.Director:getInstance():popScene()
    end

    local function goMenu(pSender)                                         ③
        cclog("MyActionScene goMenu")
        local p = cc.p(math.random() * size.width, math.random() * size.height)   ④

        if actionFlag == PLACE_TAG then
```

```
        sprite:runAction(cc.Place:create(p))                                    ⑤
    elseif actionFlag == FLIPX_TAG then
        sprite:runAction(cc.FlipX:create(true))                                 ⑥
    elseif actionFlag == FLIPY_TAG then
        sprite:runAction(cc.FlipY:create(true))                                 ⑦
    elseif actionFlag == HIDE_SHOW_TAG then
        if hiddenFlag then
            sprite:runAction(cc.Hide:create())                                  ⑧
            hiddenFlag = false
        else
            sprite:runAction(cc.Show:create())                                  ⑨
            hiddenFlag = true
        end
    else
        sprite:runAction(cc.ToggleVisibility:create())                          ⑩
    end
    backMenuItem:registerScriptTapHandler(backMenu)
    goMenuItem:registerScriptTapHandler(goMenu)

    return layer
end

return MyActionScene
```

上述第①行代码定义了布尔成员变量，用来保存精灵隐藏的状态。

第②行代码定义了精灵成员变量。

第②和第③行代码是定义菜单回调函数。

第④行代码用来获得一个屏幕中的随机点，其中 math.random()函数可以产生 0～1 之间的随机数。

第⑤行代码 sprite:runAction(cc.Place:create(p))可以执行一个 Place 的动作，Place 动作是将精灵等 Node 对象移动到 p 点。

第⑥行代码 sprite:runAction(cc.FlipX:create(true))是执行一个 FlipX 的动作，FlipX 动作是将精灵等 Node 对象沿水平方向翻转。

第⑦行代码 sprite:runAction(cc.FlipY:create(true))是执行一个 FlipY 的动作，FlipY 动作是将精灵等 Node 对象沿垂平方向翻转。

第⑧行代码 sprite:runAction(cc.Hide:create())是执行一个 Show 的动作，Show 动作是将精灵等 Node 对象隐藏。

第⑨行代码 sprite:runAction(cc.Show:create())是执行一个 Show 的动作，Show 动作是将精灵等 Node 对象显示。

第⑩行代码 sprite:runAction(cc.ToggleVisibility:create())是执行一个 ToggleVisibility 的动作，ToggleVisibility 动作是将精灵等 Node 对象显示/隐藏切换。

8.1.2　间隔动作

间隔动作执行完成需要一定的时间，用户可以设置 duration 属性来设置动作的执行时

间。间隔动作基类是 ActionInterval。间隔动作 ActionInterval 类图如图 8-4 所示。

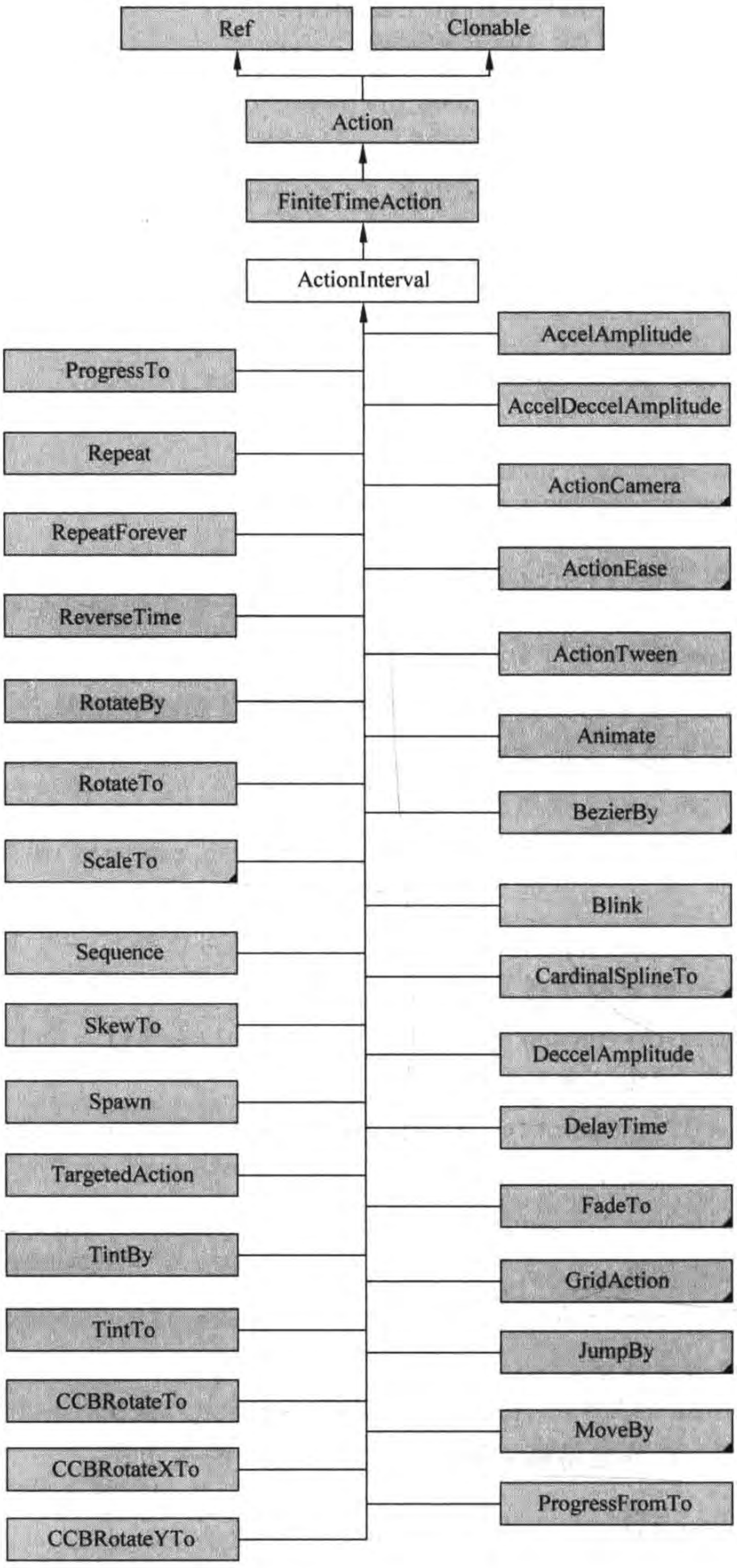
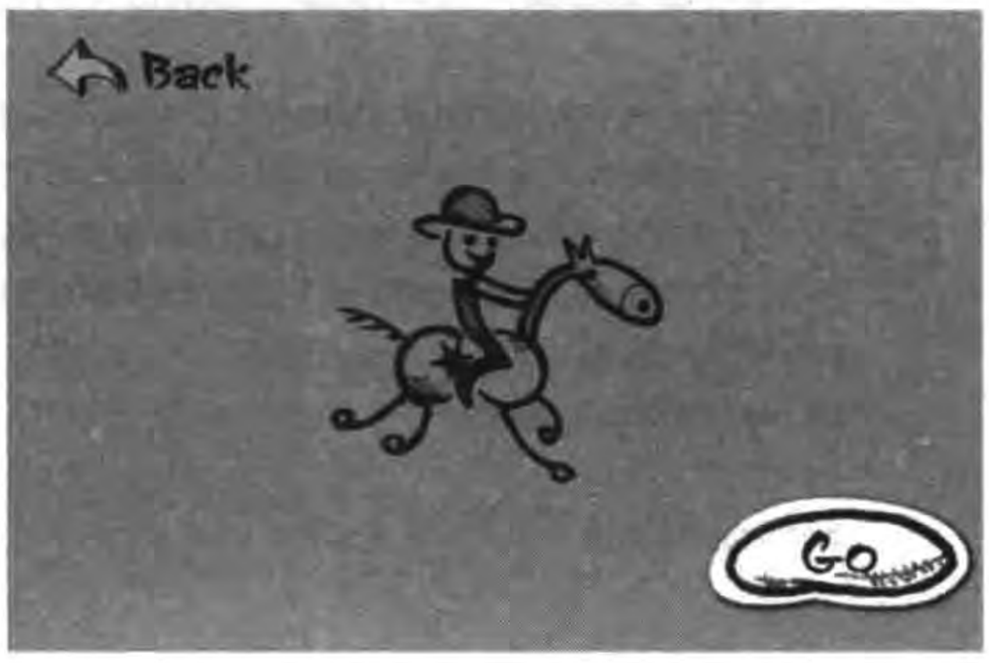


图 8-4　间隔动作类图

下面通过实例介绍间隔动作的使用，如图 8-5 所示，图(a)是一个操作菜单场景，选择菜单可以进入到图(b)动作场景，在图(b)动作场景中单击 Go 按钮可以执行选择的动作效果，单击 Back 按钮可以返回到菜单场景。

(a)



(b)

图 8-5　间隔动作实例

GameScene. lua 代码如下：

```
kMoveTo          = 102                                          ①
kMoveBy          = 103
...
kFadeIn          = 115
kFadeOut         = 116                                          ②

-- 操作标识
actionFlag =  - 1

...

-- create layer
function GameScene:createLayer()

    local layer = cc.Layer:create()

    ⋮

local pItmLabel1 = cc.Label:createWithBMFont("fonts/fnt2.fnt","MoveTo")
local pItmMenu1 = cc.MenuItemLabel:create(pItmLabel1)
pItmMenu1:setTag(kMoveTo)
pItmMenu1:registerScriptTapHandler(OnClickMenu)

local pItmLabel2 = cc.Label:createWithBMFont("fonts/fnt2.fnt", "MoveBy")
local pItmMenu2 = cc.MenuItemLabel:create(pItmLabel2)
pItmMenu2:setTag(kMoveBy)
pItmMenu2:registerScriptTapHandler(OnClickMenu)

⋮
```

```
local pItmLabel15 = cc.Label:createWithBMFont("fonts/fnt2.fnt", "FadeOut")
local pItmMenu15 = cc.MenuItemLabel:create(pItmLabel15)
pItmMenu15:setTag(kFadeOut)
pItmMenu15:registerScriptTapHandler(OnClickMenu)

local mn = cc.Menu:create(pItmMenu1,pItmMenu2,pItmMenu3,pItmMenu4,pItmMenu5,
    pItmMenu6,pItmMenu7,pItmMenu8,pItmMenu9,
    pItmMenu10,pItmMenu11,pItmMenu12,
    pItmMenu13,pItmMenu14,pItmMenu15)
    mn:alignItemsInColumns(3, 3, 3, 3, 3)                                    ③
    layer:addChild(mn)

    return layer
end
```

上述第①和第②行代码定义 15 个常量,这 15 个常量对应 15 个菜单项。

第③行代码设置菜单项是分列显示,菜单项分为 5 行,第一个参数 3 表示第一行有 3 列,以此类推。

MyActionScene.lua 主要代码如下:

```
function MyActionScene:createLayer()
    cclog("MyActionScene actionFlag = %d", actionFlag)
    local layer = cc.Layer:create()

    local bg = cc.Sprite:create("Background.png")
    bg:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(bg)

    local sprite = cc.Sprite:create("hero.png")
    sprite:setPosition(cc.p(size.width / 2, size.height / 2))
    layer:addChild(sprite)

    local backMenuItem = cc.MenuItemImage:create("Back - up.png","Back - down.png")
    backMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(140, 65)))

    local goMenuItem = cc.MenuItemImage:create("Go - up.png","Go - down.png")
    goMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(820, 540)))

    local mn = cc.Menu:create(backMenuItem, goMenuItem)

    mn:setPosition(cc.p(0, 0))
    layer:addChild(mn)

    local function backMenu(pSender)
        cclog("MyActionScene backMenu")
        cc.Director:getInstance():popScene()
    end
```

```
local function goMenu(pSender)
    cclog("MyActionScene goMenu")
    local p = cc.p(math.random() * size.width, math.random() * size.height)
    if actionFlag == kMoveTo then
        sprite:runAction(cc.MoveTb:create(2,cc.p(size.width - 50, size.height - 50)))  ①
    elseif actionFlag == kMoveBy then
        sprite:runAction(cc.MoveBy:create(2,cc.p( - 50, - 50)))                        ②
    elseif actionFlag == kJumpTo then
        sprite:runAction(cc.JumpTo:create(2,cc.p(150, 50),30,5))                       ③
    elseif actionFlag == kJumpBy then
        sprite:runAction(cc.JumpBy:create(2,cc.p(100, 100),30,5))                      ④
    elseif actionFlag == kBezierBy then
        local bezier = {
            cc.p(0, size.height / 2),
            cc.p(300, - size.height / 2),
            cc.p(100, 100)
        }                                                                              ⑤
        sprite:runAction(cc.BezierBy:create(3,bezier))                                 ⑥
    elseif actionFlag == kScaleTo then
        sprite:runAction(cc.ScaleTo:create(2, 4))                                      ⑦
    elseif actionFlag == kScaleBy then
        sprite:runAction(cc.ScaleBy:create(2, 0.5))                                    ⑧
    elseif actionFlag == kRotateTo then
        sprite:runAction(cc.RotateTo:create(2,180))                                    ⑨
    elseif actionFlag == kRotateBy then
        sprite:runAction(cc.RotateBy:create(2, - 180))                                 ⑩
    elseif actionFlag == kBlink then
        sprite:runAction(cc.Blink:create(3, 5))                                        ⑪
    elseif actionFlag == kTintTo then
        sprite:runAction(cc.TintTo:create(2, 255, 0, 0))                               ⑫
    elseif actionFlag == kTintBy then
        sprite:runAction(cc.TintBy:create(0.5,0, 255, 255))                            ⑬
    elseif actionFlag == kFadeTo then
        sprite:runAction(cc.FadeTo:create(1, 80))                                      ⑭
    elseif actionFlag == kFadeIn then
        sprite:setOpacity(10)
        sprite:runAction(cc.FadeIn:create(1))                                          ⑮
    elseif actionFlag == kFadeOut then
        sprite:runAction(cc.FadeOut:create(1))                                         ⑯
        end
    end
    backMenuItem:registerScriptTapHandler(backMenu)
    goMenuItem:registerScriptTapHandler(goMenu)

    return layer
end
```

在上述代码 goMenu 函数中是运行间隔动作，间隔动作中有很多类都是 XxxTo 和 XxxBy 的命名。XxxTo 是指运动到指定位置，这个位置是绝对的。XxxBy 是指运动到相

对于本身的位置，这个位置是相对的。

第①和第②行代码中的 MoveTo 和 MoveBy 是移动函数，第一个参数是持续的时间，第二个参数是移动到的位置。

第③和第④行代码中的 JumpTo 和 JumpBy 是跳动函数，第一个参数是持续的时间，第二个参数是跳动到的位置，第三个参数是跳到的高度，第四个参数是跳动的次数。

第⑥行代码 sprite:runAction(cc.BezierBy:create(3,bezier))是执行贝塞尔曲线动作。第⑤行代码是设置贝塞尔曲线，其中 cc.p(0, size.height / 2)是贝塞尔曲线第一控制点，cc.p(300, − size.height / 2)是贝塞尔曲线第二控制点，cc.p(100, 100)是贝塞尔曲线结束点。

提示 贝赛尔(Bézier)曲线是法国数学家贝塞尔在工作中发现，任何一条曲线都可以通过与它相切的控制线两端的点的位置来定义。因此，贝赛尔曲线可以用 4 个点描述，其中两个点描述两个端点，另外两个描述每一端的切线。贝赛尔曲线可以分为：二次方贝赛尔曲线(图 8-6)和高阶贝赛尔曲线(图 8-7 是三次方贝赛尔曲线)。

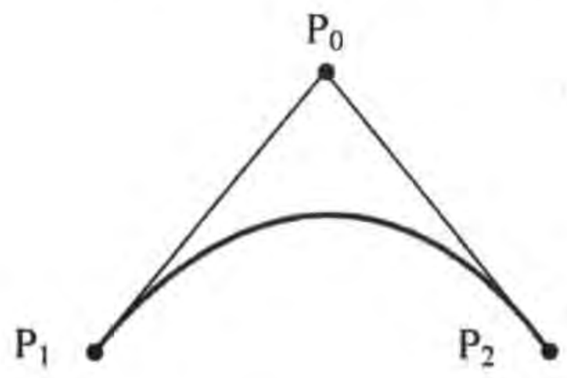


图 8-6 二次方贝赛尔曲线

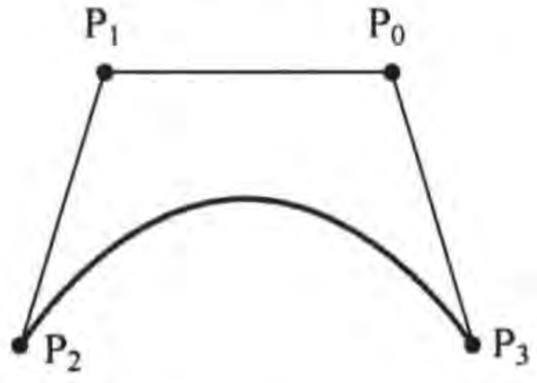


图 8-7 三次方贝赛尔曲线

第⑦和第⑧行代码中的 ScaleTo 和 ScaleBy 是缩放动作函数，第一个参数是持续的时间，第二个参数是缩放比例。

第⑨和第⑩行代码中的 RotateTo 和 RotateBy 是旋转动作函数，第一个参数是持续的时间，第二个参数是旋转角度。

第⑪行代码 sprite: runAction(cc.Blink::create(3, 5))是闪烁动作函数，第一个参数是持续的时间，第二个参数是闪烁次数。

第⑫和第⑬行代码中的 TintTo 和 TintBy 是染色动作函数，第一个参数是持续的时间，第二、三、四参数是 RGB 颜色值，取值范围 0～255。

第⑭行代码 sprite: runAction(cc.FadeTo::create(1, 80))是不透明度变换动作函数，第一个参数是持续的时间，第二个参数是不透明度，参数 80 表示不透明的占 80%。

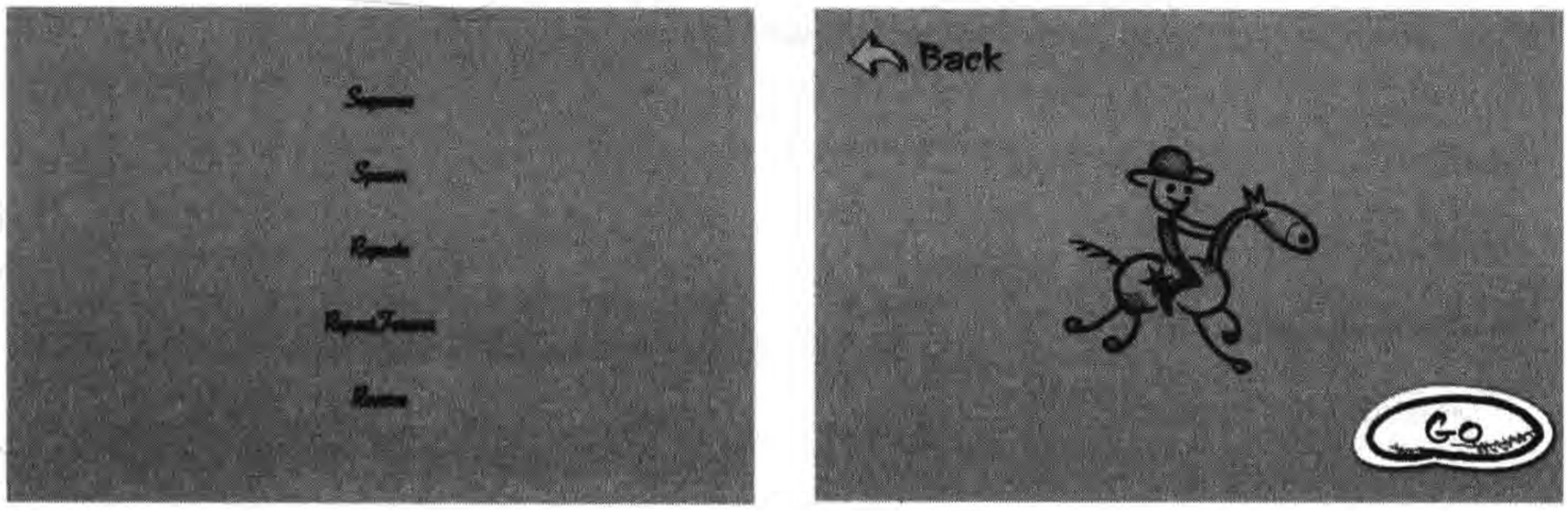
第⑮和第⑯行代码中的 FadeIn 和 FadeOut 是淡入(渐显)和淡出(渐弱)动作函数，参数是持续的时间。在设置 FadeIn 之前先通过 sprite:setOpacity(10)语句设置精灵的不透明度，取值范围是 0～255，0 为完全透明，255 为完全不透明。

8.1.3 组合动作

动作往往不是单一的，而是复杂的组合。可以按照一定的次序将上述基本动作组合起

来，形成连贯的一套组合动作。组合动作包括顺序、并列、有限次数重复、无限次数重复、反动作和动画。动画将在下一节介绍，本节重点介绍顺序、并列、有限次数重复、无限次数重复和反动作。

下面通过实例介绍组合动作的使用，如图 8-8 所示，图(a)是一个操作菜单场景，选择菜单可以进入到图(b)动作场景，在图(b)动作场景中单击 Go 按钮可以执行所选择的动作效果，单击 Back 按钮可以返回到菜单场景。



(a)　　(b)

图 8-8　组合动作实例

GameScene 场景代码与 8.1.1 节类似，不再赘述，这里重点介绍 MyActionScene 场景，MyActionScene.lua 主要代码如下：

```
require "Cocos2d"
require "Cocos2dConstants"

local sprite

...

-- create layer
function MyActionScene:createLayer()
    cclog("MyActionScene actionFlag =  %d", actionFlag)
    local layer = cc.Layer:create()
      ...

local function backMenu(pSender)
    cclog("MyActionScene backMenu")
    cc.Director:getInstance():popScene()
end

local function goMenu(pSender)
    cclog("MyActionScene goMenu")

    if actionFlag == kSequence then
        self:OnSequence(pSender)
    elseif actionFlag == kSpawn then
```

```
            self:OnSpawn(pSender)
        elseif actionFlag == kRepeate then
            self:OnRepeat(pSender)
        elseif actionFlag == kRepeatForever1 then
            self:OnRepeatForever(pSender)
        elseif actionFlag == kReverse then
            self:OnReverse(pSender)
        end
    end
    backMenuItem:registerScriptTapHandler(backMenu)
    goMenuItem:registerScriptTapHandler(goMenu)

    return layer
end

<回调函数代码>

return MyActionScene
```

回调函数 OnSequence 代码如下：

```
function MyActionScene:OnSequence(pSender)
    cclog("MyActionScene OnSequence")

    local p = cc.p(size.width/2, 200)

    local ac0 = sprite:runAction(cc.Place:create(p))                                    ①
    local ac1 = sprite:runAction(cc.MoveTo:create(2,cc.p(size.width - 130, size.height -
200)))                                                                                  ②
    local ac2 = sprite:runAction(cc.JumpBy:create(2, cc.p(8, 8),6, 3))                  ③
    local ac3 = sprite:runAction(cc.Blink:create(2,3))                                  ④
    local ac4 = sprite:runAction(cc.TintBy:create(0.5,0,255,255))                       ⑤

    sprite:runAction(cc.Sequence:create(ac0, ac1, ac2, ac3, ac4, ac0))                  ⑥

end
```

上述代码实现了顺序动作演示，其中主要使用的类是 Sequence，Sequence 是派生于 ActionInterval 属性的间隔动作。Sequence 的作用就是顺序排列若干个动作，然后按先后次序逐个执行。第①行代码是创建 Place 动作。第②行代码是创建 MoveTo 动作。第③行代码是创建 JumpBy 动作。第④行代码是创建 Blink 动作。第⑤行代码是创建 TintBy 动作。第⑥行代码执行 Sequence，Sequence 的 create 函数参数是动作数组。

OnSpawn 函数是在演示并列动作时调用的函数，其代码如下：

```
function MyActionScene:OnSpawn(pSender)
    cclog("MyActionScene OnSpawn")
    local p = cc.p(size.width/2, 200)

    sprite:setRotation(0)                                                               ①
    sprite:setPosition(p)                                                               ②

    local ac1 = sprite:runAction(cc.MoveTo:create(2,cc.p(size.width - 100, size.height -
```

```
100)))                                                                    ③
    local ac2 = sprite:runAction(cc.RotateTo:create(2, 40))               ④

    sprite:runAction(cc.Spawn:create(ac1,ac2))                            ⑤

end
```

上述代码实现了并列动作演示，其中主要使用的类是 Spawn，Spawn 也是从 ActionInterval 继承而来，其作用就是同时并列执行若干个动作，但要求动作都必须是可以同时执行的。比如移动式翻转、改变色、改变大小等。第①行代码 sprite:setRotation(0)设置精灵旋转角度为 0°，即保持原来状态。第②行代码 sprite:setPosition(p)是重新设置精灵位置。第③行代码创建 MoveTo 动作。第④行代码创建 RotateTo 动作。第⑤行代码 sprite:runAction(cc.Spawn:create(ac1,ac2))执行并列动作，create 函数的参数是动作类型数组。

OnRepeat 函数是在演示重复动作时调用的函数，其代码如下：

```
function MyActionScene:OnRepeat(pSender)
    cclog("MyActionScene OnRepeat")
    local p = cc.p(size.width/2, 200)

    sprite:setRotation(0)
    sprite:setPosition(p)

    local ac1 = sprite:runAction(cc.MoveTo:create(2,cc.p(size.width - 100, size.height -
100)))                                                                    ①
    local ac2 = sprite:runAction(cc.JumpBy:create(2,cc.p(10, 10), 20,5))  ②
    local ac3 = sprite:runAction(cc.JumpBy:create(2,cc.p(-10, -10),20,3)) ③

    local seq = cc.Sequence:create(ac1, ac2, ac3)                         ④

    sprite:runAction(cc.Repeat:create(seq,3))                             ⑤
end
```

上述代码实现了重复动作演示，其中主要使用的类是 Repeat，它也是从 ActionInterval 继承而来。第①行代码是创建 MoveTo 动作。第②行代码是创建 JumpBy 动作。第③行代码是创建 JumpBy 动作。第④行代码是创建顺序动作对象 seq。第⑤行代码 sprite:runAction(cc.Repeat:create(seq,3))是重复运行顺序动作 3 次。

OnRepeatForever 函数是在演示无限重复动作时调用的函数，其代码如下：

```
function MyActionScene:OnRepeatForever(pSender)
    cclog("MyActionScene OnRepeatForever")
    local p = cc.p(size.width/2, 500)

    sprite:setRotation(0)
    sprite:setPosition(p)

    local bezier = {                                                      ①
        cc.p(0, size.height / 2),
        cc.p(10, - size.height / 2),
```

```
        cc.p(10,20)
    }                                                                    ②
    local ac1 = sprite:runAction(cc.BezierBy:create(2,bezier))           ③
    local ac2 = sprite:runAction(cc.TintBy:create(0.5, 0, 255, 255))     ④
    local ac1Reverse = ac1:reverse()                                     ⑤
    local ac2Repeat = sprite:runAction(cc.Repeat:create(ac2, 4))         ⑥

    local ac3 = sprite:runAction(cc.Spawn:create(ac1,ac2Repeat))         ⑦
    local ac4 = sprite:runAction(cc.Spawn:create(ac1Reverse,ac2Repeat))  ⑧
    local seq = cc.Sequence:create(ac3, ac4)                             ⑨

        sprite:runAction(cc.RepeatForever:create(seq))                   ⑩
end
```

上述代码实现了无限重复动作演示,其中主要使用的类是 RepeatForever,它也是从 ActionInterval 继承而来。第①和第②行代码是定义贝塞尔曲线控制点。第③行代码是创建贝塞尔曲线动作 BezierBy。第④行代码是创建动作 TintBy。第⑤行代码是创建 BezierBy 动作的反转动作(OnRveverse 函数将在后面介绍)。第⑥行代码是创建重复动作。第⑦和第⑧行代码是创建并列动作。第⑨行代码是创建顺序动作。第⑩行代码 sprite:runAction(cc.RepeatForever:create(seq))是执行无限重复动作。

OnReverse 函数是在演示反动作时调用的函数,其代码如下:

```
function MyActionScene:OnReverse(pSender)
    cclog("MyActionScene OnReverse")
    local p = cc.p(size.width/2, 300)

    sprite:setRotation(0)
    sprite:setPosition(p)

    local ac1 = sprite:runAction(cc.MoveBy:create(2,cc.p(40, 60)))       ①
    local ac2 = ac1:reverse()                                            ②
    local seq = cc.Sequence:create(ac1, ac2)                             ③

    sprite:runAction(cc.Repeat:create(seq,2))                            ④
end
```

上述代码实现了反动作(支持顺序动作的反顺序动作,反顺序动作不是一个类)演示,不是所有的动作类都支持反动作。XxxTo 类通常不支持反动作,XxxBy 类通常支持。

第①行代码是创建一个移动 MoveBy 动作。

第②行代码是调用 ac1 的 reverse()函数执行反动作。

第③行代码是创建顺序动作。

第④行代码 sprite:runAction(cc.Repeat:create(seq,2))是执行反动作。

8.1.4 动作速度控制

基本动作和组合动作实现了针对精灵的各种运动和动画效果的改变。但这样的改变速度是匀速的、线性的。通过 ActionEase 及其子类可以使精灵以非匀速或非线性速度运动,

这样看起来效果更加逼真。

ActionEase 的类图如图 8-9 所示。

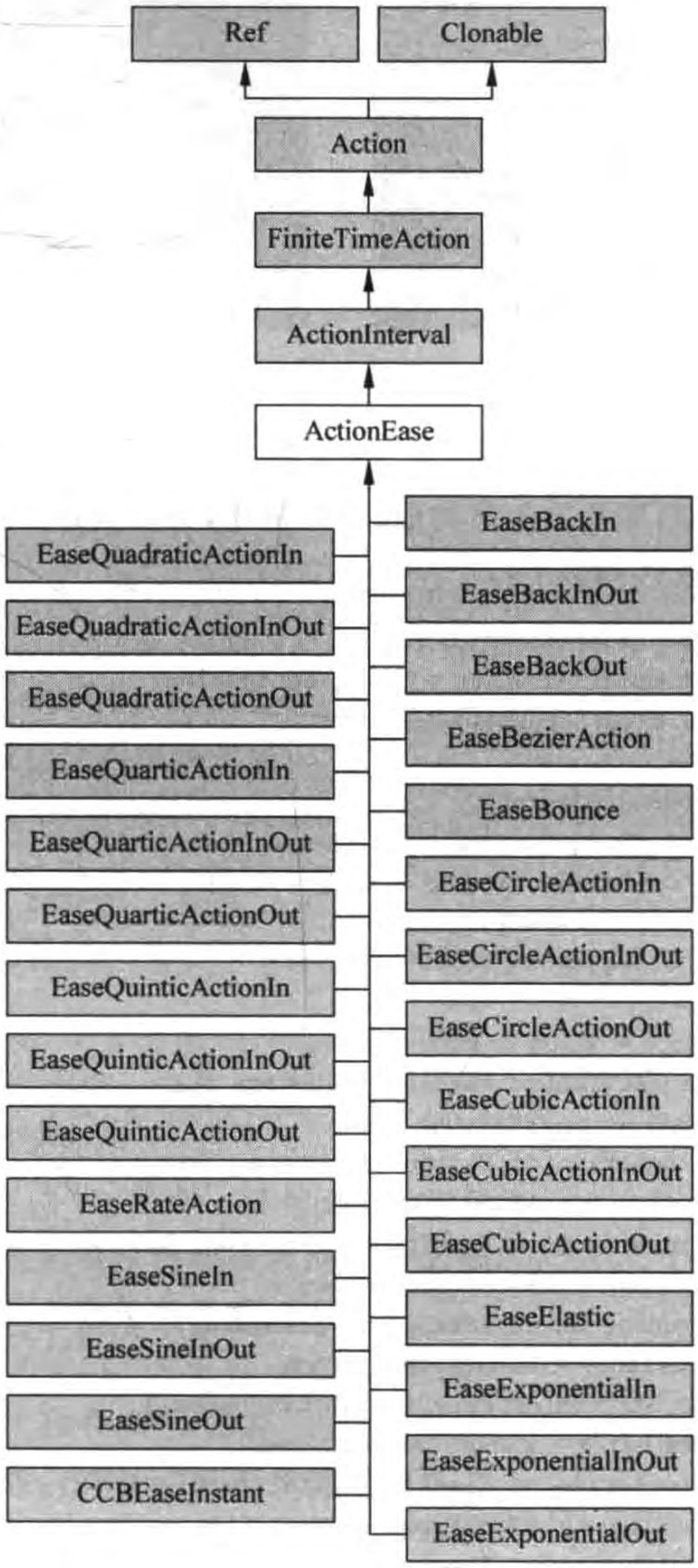
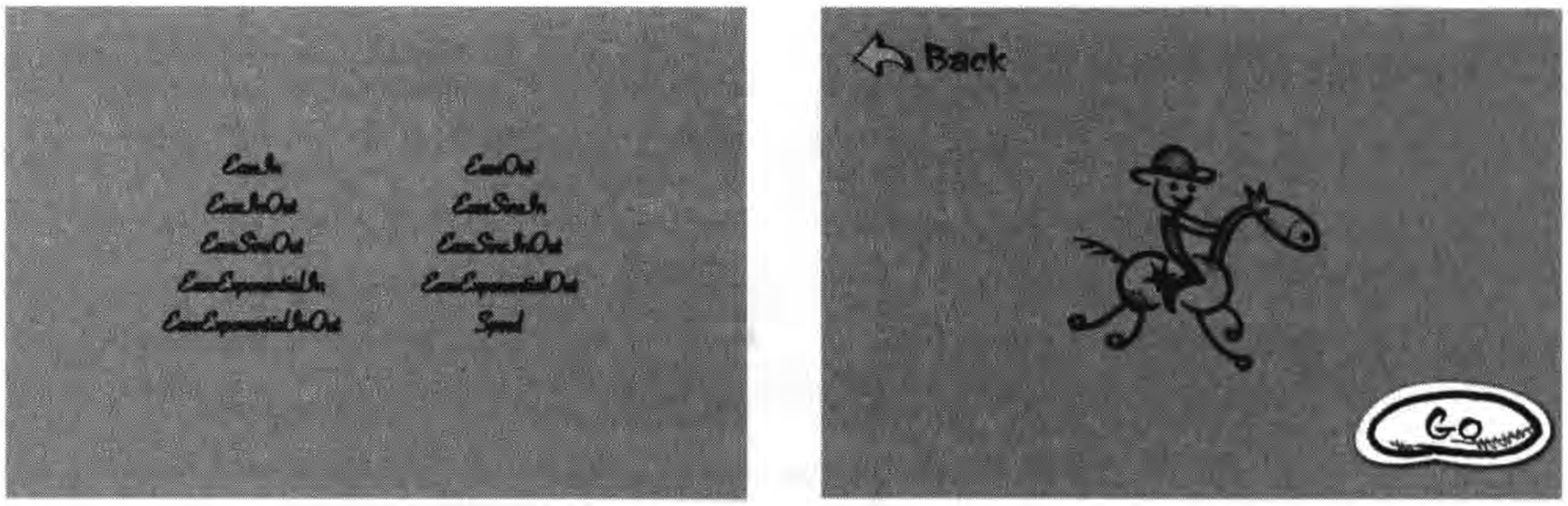


图 8-9　ActionEase 类图

下面通过一个实例介绍这些动作中速度的控制，如图 8-10 所示，图(a)是一个操作菜单场景，选择菜单可以进入到图(b)动作场景，在图(b)动作场景中单击 Go 按钮可以执行所选择的动作效果，单击 Back 按钮可以返回到菜单场景。



(a) (b)

图 8-10 动作速度控制实例

GameScene 场景代码与 8.1.1 节类似，不再赘述，下面重点介绍 MyActionScene 场景，MyActionScene.lua 主要代码如下：

```
function MyActionScene:createLayer()
    cclog("MyActionScene actionFlag =  %d", actionFlag)
    local layer = cc.Layer:create()
    ...
    local function goMenu(pSender)
        cclog("MyActionScene goMenu")
    local ac1 = cc.MoveBy:create(2, cc.p(200, 0))
    local ac2 = ac1:reverse()
    local ac = cc.Sequence:create(ac1, ac2)

    if actionFlag == kEaseIn then
        sprite:runAction(cc.EaseIn:create(ac, 3))                    ①
    elseif actionFlag == kEaseOut then
        sprite:runAction(cc.EaseOut:create(ac, 3))                   ②
    elseif actionFlag == kEaseInOut then
        sprite:runAction(cc.EaseInOut:create(ac, 3))                 ③
    elseif actionFlag == kEaseSineIn then
        sprite:runAction(cc.EaseSineIn:create(ac))                   ④
    elseif actionFlag == kEaseSineOut then
        sprite:runAction(cc.EaseSineOut:create(ac))                  ⑤
    elseif actionFlag == kEaseSineInOut then
        sprite:runAction(cc.EaseSineInOut:create(ac))                ⑥
    elseif actionFlag == kEaseExponentialIn then
        sprite:runAction(cc.EaseExponentialIn:create(ac))            ⑦
    elseif actionFlag == kEaseExponentialOut then
        sprite:runAction(cc.EaseExponentialOut:create(ac))           ⑧
    elseif actionFlag == kEaseExponentialInOut then
        sprite:runAction(cc.EaseExponentialInOut:create(ac))         ⑨
    elseif actionFlag == kSpeed then
        sprite:runAction(cc.Speed:create(ac, (math.random() * 5)))   ⑩
    end
```

```
    end

    backMenuItem:registerScriptTapHandler(backMenu)
    goMenuItem:registerScriptTapHandler(goMenu)

    return layer
end
```

上述第①行代码 sprite:runAction(cc.EaseIn:create(ac, 3))是以 3 倍速度由慢至快。第②代码 sprite:runAction(cc.EaseOut:create(ac, 3))是以 3 倍速度由快至慢。第③行代码 sprite:runAction(cc.EaseInOut:create(ac, 3))是以 3 倍速度由慢至快再由快至慢。

第④行代码 sprite:runAction(cc.EaseSineIn:create(ac))是采用正弦变换速度由慢至快。第⑤行代码 sprite:runAction(cc.EaseSineOut:create(ac))是采用正弦变换速度由快至慢。第⑥行代码 sprite:runAction(cc.EaseSineInOut:create(ac))是采用正弦变换速度由慢至快再由快至慢。

第⑦行代码 sprite:runAction(cc.EaseExponentialIn:create(ac))是采用指数变换速度由慢至快。第⑧行代码 sprite:runAction(cc.EaseExponentialOut:create(ac))是采用指数变换速度由快至慢。第⑨行代码 sprite:runAction(cc.EaseExponentialInOut:create(ac))是采用指数变换速度由慢至快再由快至慢。

第⑩行代码 sprite:runAction(cc.Speed:create(ac, (math.random() * 5)))是随机设置变换速度。

8.1.5　函数调用

在顺序动作执行的中间或者结束后，可以回调某个函数，在该函数中执行任何处理。函数调用可以分为无参数函数调用和有参数函数调用。

函数调用类图如图 8-11 所示，函数调用类是 CallFunc，CallFunc 可以是无参数函数调用或有参数函数调用，也可以将自身作为参数进行函数调用。

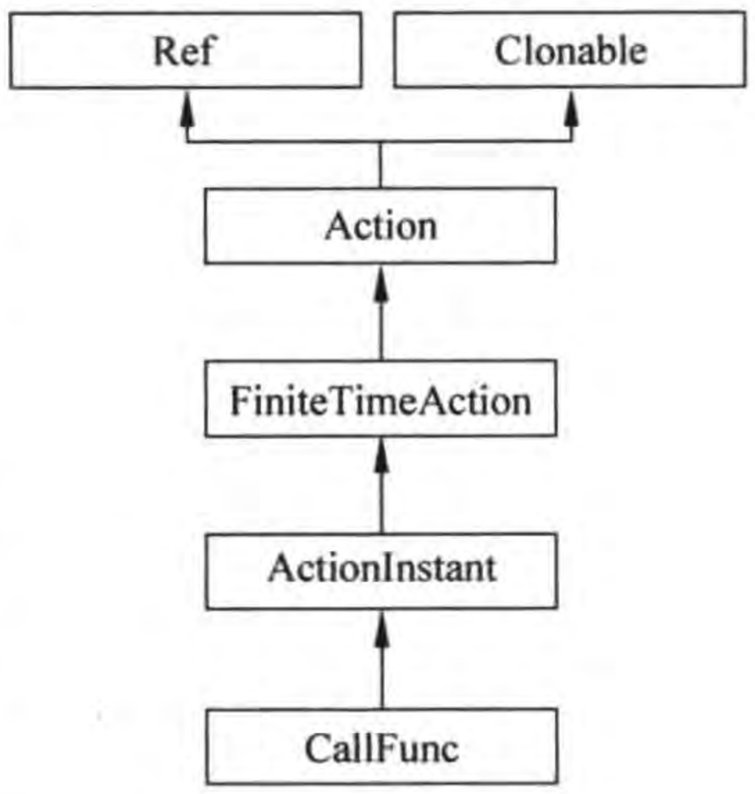
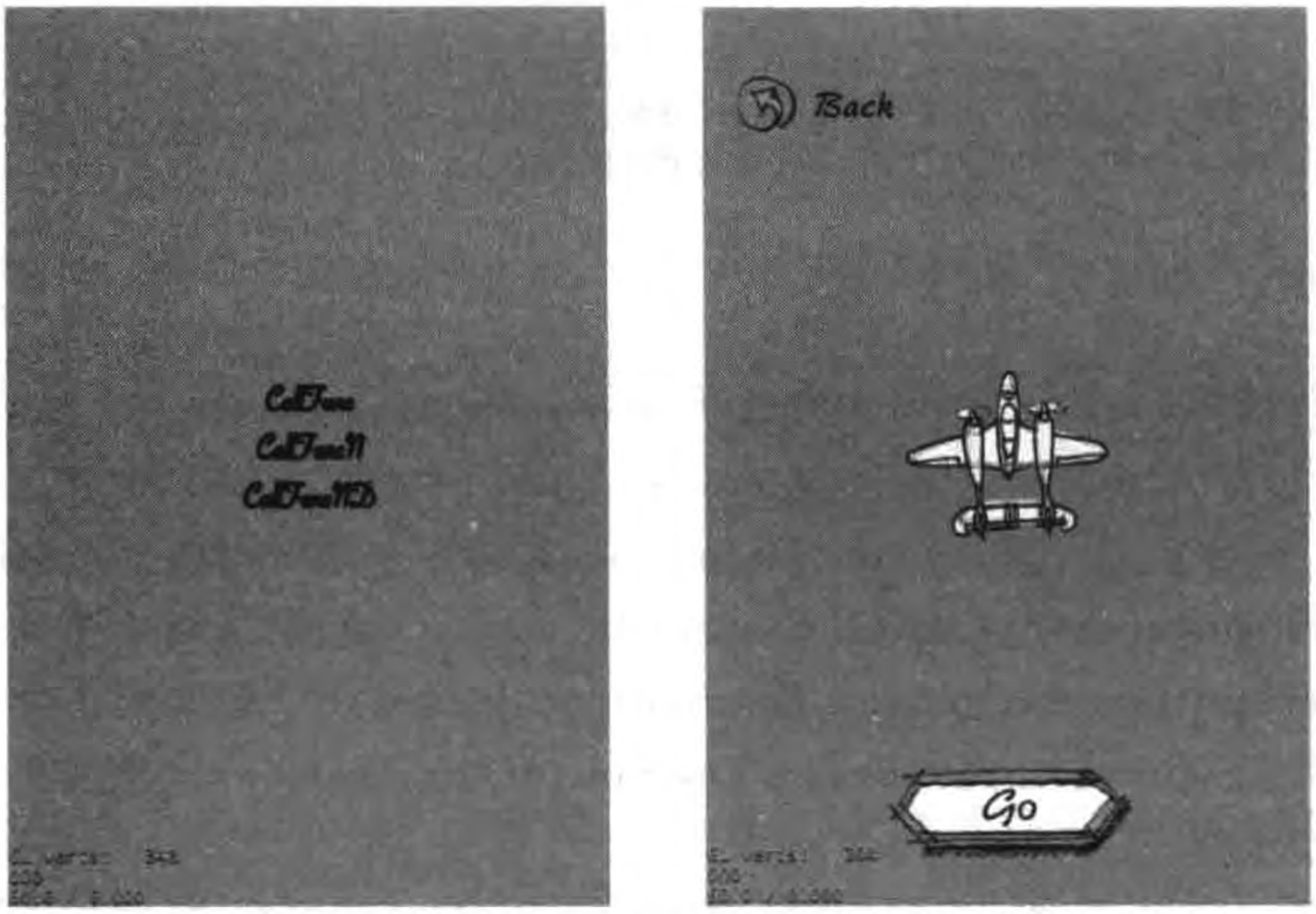


图 8-11　函数调用类图

下面通过一个实例介绍动作中的函数调用，如图 8-12 所示，图(a)是一个操作菜单场景，选择菜单可以进入到图(b)的动作场景，在图(b)的动作场景中单击 Go 按钮可以执行所选择的动作效果，单击 Back 按钮可以返回到菜单场景。其中 CallFunc 菜单项是无参数函数调用，CallFuncN 菜单项是将自身(精灵对象)作为参数进行函数调用，CallFuncND 菜单项是有参数函数调用。

GameScene 场景代码与 8.1.1 节类似，不再赘述，下面重点介绍 MyActionScene 场景，MyActionScene.lua 主要代码如下：



(a) (b)

图 8-12 函数调用实例

```
function MyActionScene:createLayer()
    cclog("MyActionScene actionFlag =  %d", actionFlag)
    local layer = cc.Layer:create()
    ...

    local function backMenu(pSender)
        cclog("MyActionScene backMenu")
        cc.Director:getInstance():popScene()
    end

    local function goMenu(pSender)
        cclog("MyActionScene goMenu")
        local ac1 = cc.MoveBy:create(2, cc.p(200, 0))
        local ac2 = ac1:reverse()
        local ac = cc.Sequence:create(ac1, ac2)

        if actionFlag == func then
            self:OnCallFunc()
        elseif actionFlag == funcN then
            self:OnCallFuncN()
        else
            self:OnCallFuncND()
        end
    end

    backMenuItem:registerScriptTapHandler(backMenu)
    goMenuItem:registerScriptTapHandler(goMenu)

    return layer
end
```

```
local function CallBack1()                                                    ①
    sprite:runAction(cc.TintBy:create(0.5, 255, 0, 255))                      ②
end

function MyActionScene:OnCallFunc()                                           ③
    cclog("MyActionScene OnCallFunc")
    local ac1 = cc.MoveBy:create(2, cc.p(100, 100))
    local ac2 = ac1:reverse()

    local acf = cc.CallFunc:create(CallBack1)                                 ④
    local seq = cc.Sequence:create(ac1, acf, ac2)
    sprite:runAction(cc.Sequence:create(seq))

end

local function CallBack2(pSender)                                             ⑤
    local sp = pSender
    sp:runAction(cc.TintBy:create(1, 255, 0, 255))                            ⑥
end

function MyActionScene:OnCallFuncN()                                          ⑦
    cclog("MyActionScene OnCallFuncN")
    local ac1 = cc.MoveBy:create(2, cc.p(100, 100))
    local ac2 = ac1:reverse()

    local acf = cc.CallFunc:create(CallBack2)                                 ⑧
    local seq = cc.Sequence:create(ac1, acf, ac2)
    sprite:runAction(cc.Sequence:create(seq))

end

local function CallBack3(pSender, table)                                      ⑨
    local sp = pSender
    cclog("CallBack3 %d", table[1])
    sp:runAction(cc.TintBy:create(table[1], table[2], table[3], table[4]))    ⑩
end

function MyActionScene:OnCallFuncND()                                         ⑪
    cclog("MyActionScene OnCallFuncND")
    local ac1 = cc.MoveBy:create(2, cc.p(100, 100))
    local ac2 = ac1:reverse()

    local acf = cc.CallFunc:create(CallBack3, {1, 255, 0, 255})               ⑫
    local seq = cc.Sequence:create(ac1, acf, ac2)
    sprite:runAction(cc.Sequence:create(seq))

end
```

上述第①行代码是声明 CallBack1()函数，第②行代码 sprite:runAction(cc.TintBy:create(0.5, 255, 0, 255))是执行 TintBy 动作。CallBack1()是无参数函数，它是在第③行 OnCallFunc()函数中调用的(见第④行代码)，在该行中创建 CallFunc 对象。

第⑤行代码是声明 CallBack2(pSender)函数，第⑥行代码 sp:runAction(cc.TintBy:create(1, 255, 0, 255))是执行 TintBy 动作。CallBack2(pSender)是有参数函数，其中

pSender 参数是精灵对象。CallBack2 函数是在第⑦行 OnCallFuncN()函数中调用的(见第⑧行代码)。

第⑨行代码是声明 CallBack3(pSender, table)函数,第⑩行代码 sp:runAction(cc.TintBy:create(table[1], table[2], table[3], table[4]))是执行 TintBy 动作,其中 table 参数是函数调用时传递过来的,table 是 Lua 的 table 数据类型,它包含了 4 个成员,table 保存的数据是 RGBA 颜色值。CallBack3 函数是在第⑪行代码 OnCallFuncND()函数中调用的(见第⑫行代码)。

8.2 特效

Cocos2d-x Lua API 提供了很多特效,这些特效事实上属于间隔动作,特效 GridAction 类也称为网格动作,它的类图如图 8-13 所示。

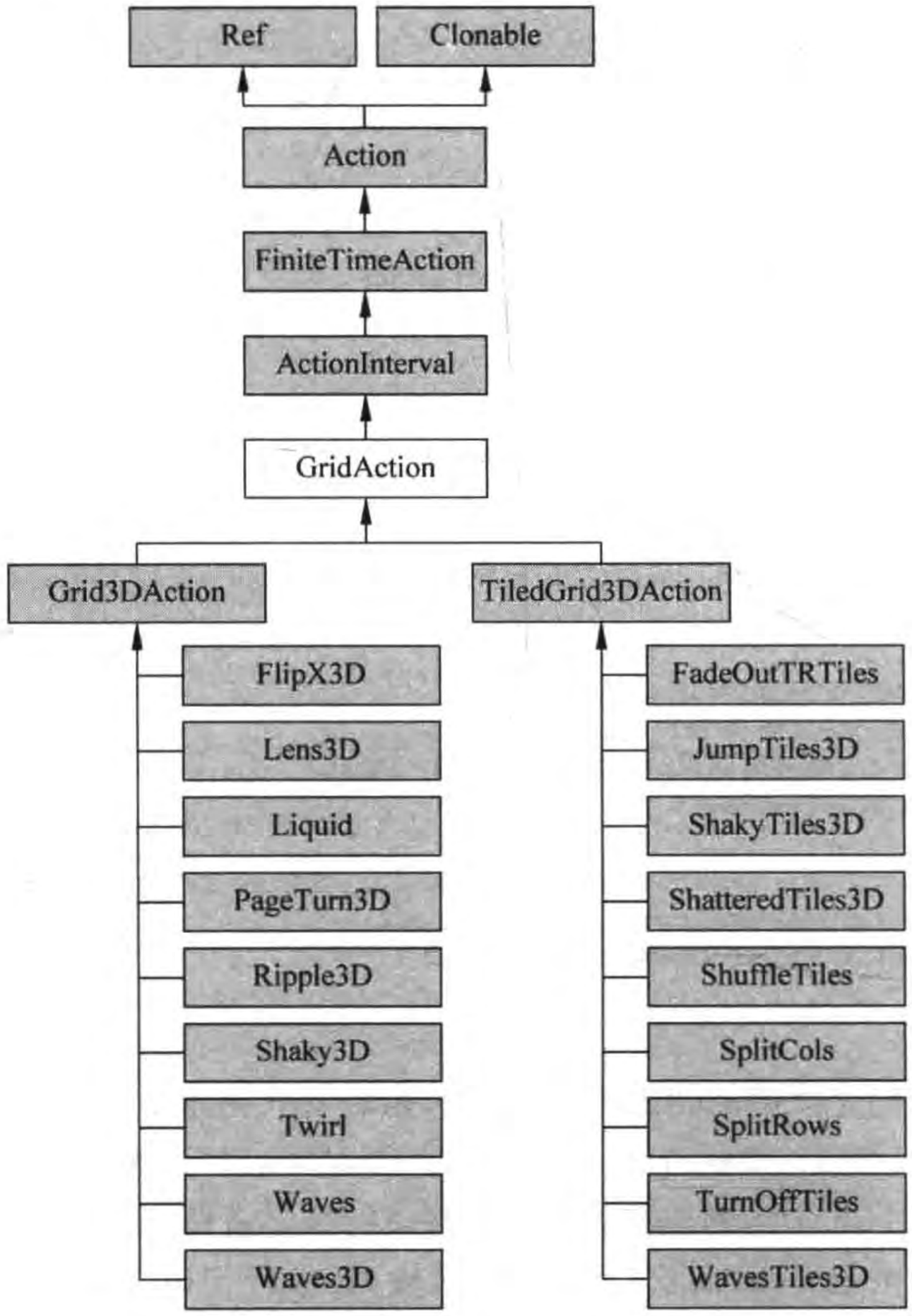


图 8-13 网格动作类图

8.2.1　网格动作

从图 8-13 所示的类图可知，GridAction 有两个主要的子类：Grid3DAction 和 TiledGrid3DAction。TiledGrid3DAction 系列的子类中会有瓦片效果，图 8-14 所示是 Waves3D 特效(Grid3DAction 子类)，图 8-15 所示是 WavesTiles3D 特效(TiledGrid3DAction 子类)，比较这两个会看到瓦片效果的特别之处是界面被分割成多个方格。

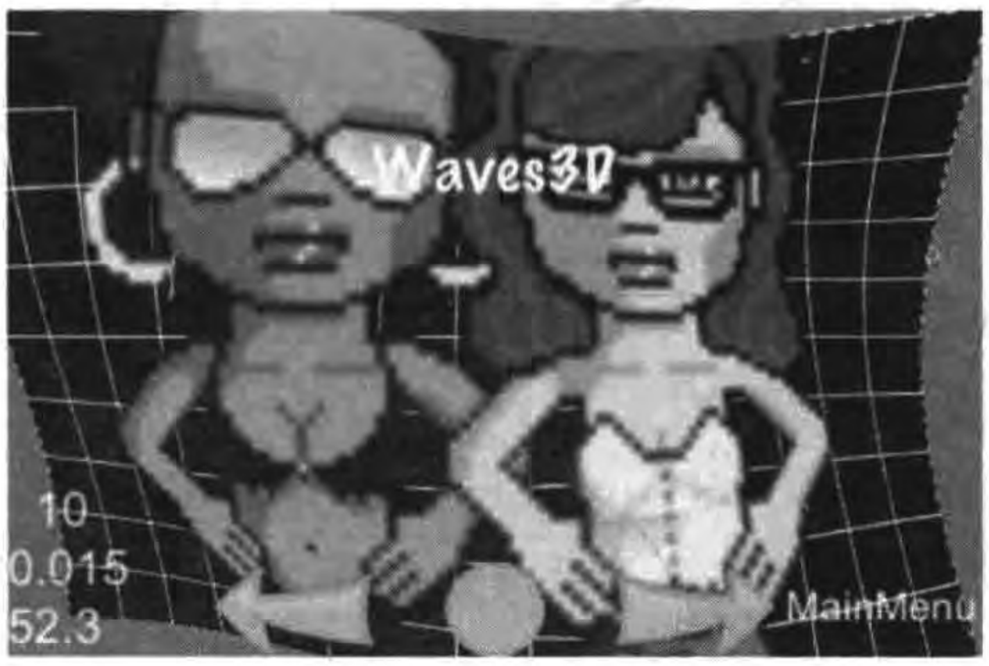


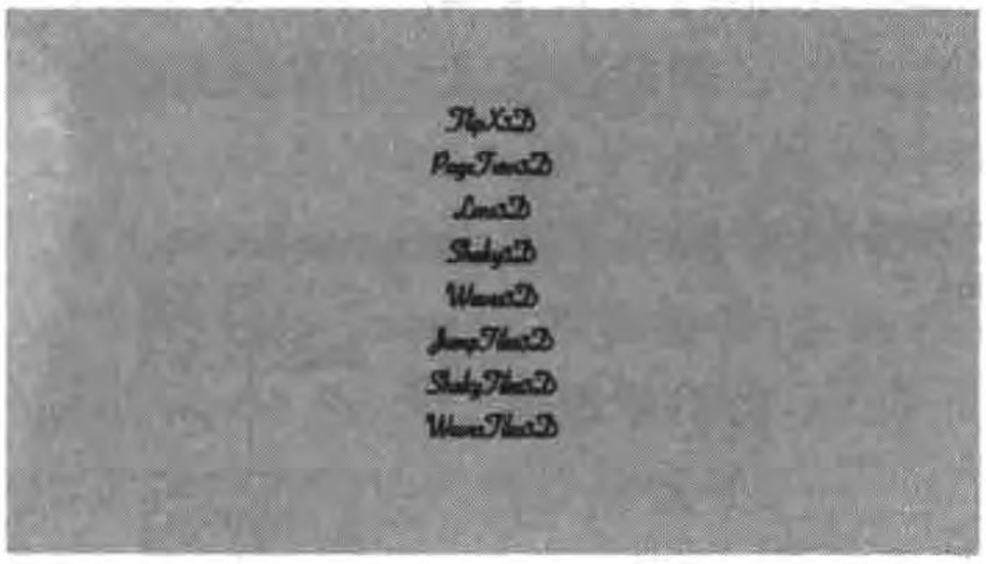
图 8-14　Waves3D 特效

图 8-15　WavesTiles3D 特效

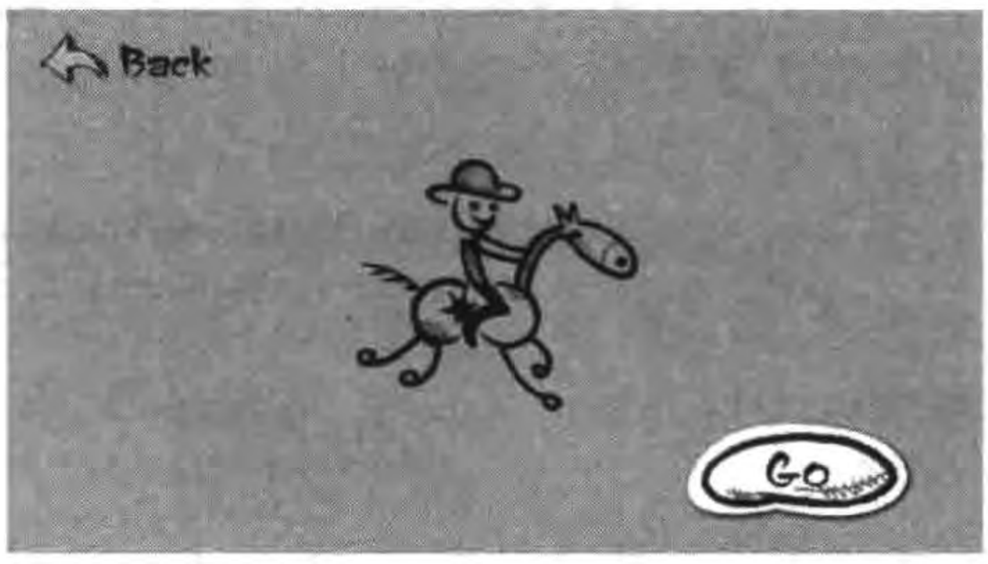
网格动作都是采用 3D 效果，给用户提供了非常震撼和绚丽的体验，但同时也给内存和 CPU 造成了巨大的压力和负担，如果不启用 OpenGL 的深度缓冲，3D 效果就会失真，但是启用对于显示性能会造成负面影响。

8.2.2　实例：特效演示

下面通过一个实例介绍几个特效的使用，如图 8-16 所示，图(a)是一个操作菜单场景，选择菜单可以进入到图(b)动作场景，在图(b)动作场景中单击 Go 按钮可以执行所选择的特效动作，单击 Back 按钮可以返回到菜单场景。



(a)



(b)

图 8-16　特效实例

GameScene 场景代码与 8.1.1 节类似，不再赘述，下面重点介绍 MyActionScene 场景，MyActionScene.lua 主要代码如下：

```
...
local sprite                                                                    ①
local gridNodeTarget                                                            ②
...
function MyActionScene:createLayer()
    cclog("MyActionScene actionFlag = %d", actionFlag)
    local layer = cc.Layer:create()

    gridNodeTarget = cc.NodeGrid:create()                                       ③
    layer:addChild(gridNodeTarget)                                              ④
    ...
    local function goMenu(pSender)
        cclog("MyActionScene goMenu")
        local ac1 = cc.MoveBy:create(2, cc.p(200, 0))
        local ac2 = ac1:reverse()
        local ac = cc.Sequence:create(ac1, ac2)

        if actionFlag == kFlipX3D then
            gridNodeTarget:runAction(cc.FlipX3D:create(3.0))                    ⑤
        elseif actionFlag == kPageTurn3D then
            gridNodeTarget:runAction(cc.PageTurn3D:create(3.0, cc.size(15,10)))  ⑥
        elseif actionFlag == kLens3D then
            gridNodeTarget:runAction(cc.Lens3D:create(3.0, cc.size(15,10),
                                                      cc.p(size.width/2,size.height/2), 240))  ⑦
        elseif actionFlag == kShaky3D then
            gridNodeTarget:runAction(cc.Shaky3D:create(3.0, cc.size(15,10), 5, false))  ⑧
        elseif actionFlag == kWaves3D then
            gridNodeTarget:runAction(cc.Waves3D:create(3.0, cc.size(15,10), 5, 40))     ⑨
        elseif actionFlag == kJumpTiles3D then
            gridNodeTarget:runAction(cc.JumpTiles3D:create(3.0, cc.size(15,10), 2, 30)) ⑩
        elseif actionFlag == kShakyTiles3D then
            gridNodeTarget:runAction(cc.ShakyTiles3D:create(3.0, cc.size(16,12), 5, false))  ⑪
        elseif actionFlag == kWavesTiles3D then
            gridNodeTarget:runAction(cc.WavesTiles3D:create(3.0, cc.size(15,10), 4, 120))    ⑫
        end
    end

    backMenuItem:registerScriptTapHandler(backMenu)
    goMenuItem:registerScriptTapHandler(goMenu)

    return layer
end
```

上述第①行代码是声明 MyActionScene 模块内使用 sprite 变量，是 Sprite 类型。第②行代码是声明 MyActionScene 模块内使用 gridNodeTarget 变量，是 NodeGrid 类型，NodeGrid 是网格动作管理类，它的类图如图 8-17 所示。

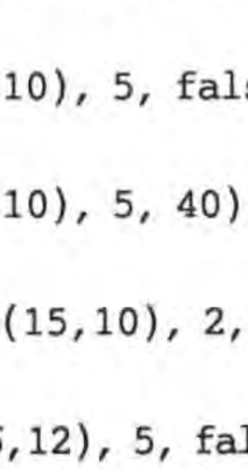


图 8-17　NodeGrid 类图

第③行代码 cc. NodeGrid: create( )是创建 NodeGrid 对象，第④行代码是将创建的 NodeGrid 对象 gridNodeTarget 添加到当前层中。

代码 goMenu 函数是运行特效动作，第⑤行代码是使用 FlipX3D 实现 *X* 轴 3D 翻转特效，create 函数的参数是持续时间。

第⑥行代码是使用 PageTurn3D 实现翻页特效，create 函数的第一个参数是持续时间，第二个参数是网格的大小。

第⑦行代码是使用 Lens3D 实现凸透镜特效，create 函数第一个参数是持续时间，第二个参数是网格大小，第三个参数是网透镜中心点，第四个参数是透镜半径。

第⑧行代码是使用 Shaky3D 实现晃动特效，create 函数第一个参数是持续时间，第二个参数是网格的大小，第三个参数是晃动的范围，第四个参数是是否伴有 *Z* 轴晃动。

第⑨行代码是使用 Waves3D 实现 3D 波动特效，create 函数第一个参数是持续时间，第二个参数是网格的大小，第三个参数是波动次数，第四个参数是振幅。

第⑩行代码是使用 JumpTiles3D 实现 3D 瓦片跳动特效，create 函数第一个参数是持续时间，第二个参数是网格的大小，第三个参数是跳动次数，第四个参数是跳动幅度。

第⑪行代码是使用 ShakyTiles3D 实现 3D 瓦片晃动特效，create 函数第一个参数是持续时间，第二个参数是网格的大小，第三个参数是晃动的范围，第四个参数是是否伴有 *Z* 轴晃动。

第⑫行代码是使用 WavesTiles3D 实现 3D 瓦片波动特效，create 函数第一个参数是持续时间，第二个参数是网格的大小，第三个参数是波动次数，第四个参数是波动振幅。

8.3　动画

与动作密不可分的还有动画，动画又可以分为场景过渡动画和帧动画。场景过渡动画在第 6 章中介绍过，下面只介绍帧动画。

8.3.1　帧动画

帧动画就是按一定时间间隔、一定的顺序、一帧一帧地显示帧图片。美工要为精灵的运动绘制每一帧图片，因此帧动画会由很多帧组成，按照一定的顺序切换这些图片就可以了。

在 Cocos2d-x Lua API 中播放帧动画涉及两个类：Animation 和 Animate。类图如图 8-18 所示，Animation 是动画类，它保存有很多动画帧；Animate 类是动作类，它继承于 ActionInterval 类，属于间隔动作类，它的作用是将 Animation 定义的动画转换为动作进行执行，这样就看到动画播放的效果了。

8.3.2　实例：帧动画使用

下面通过一个实例介绍帧动画的使用，如图 8-19 所示，单击 Go 按钮开始播放动画，这时播放按钮标题变为 Stop，单击 Stop 按钮可以停止播放动画。

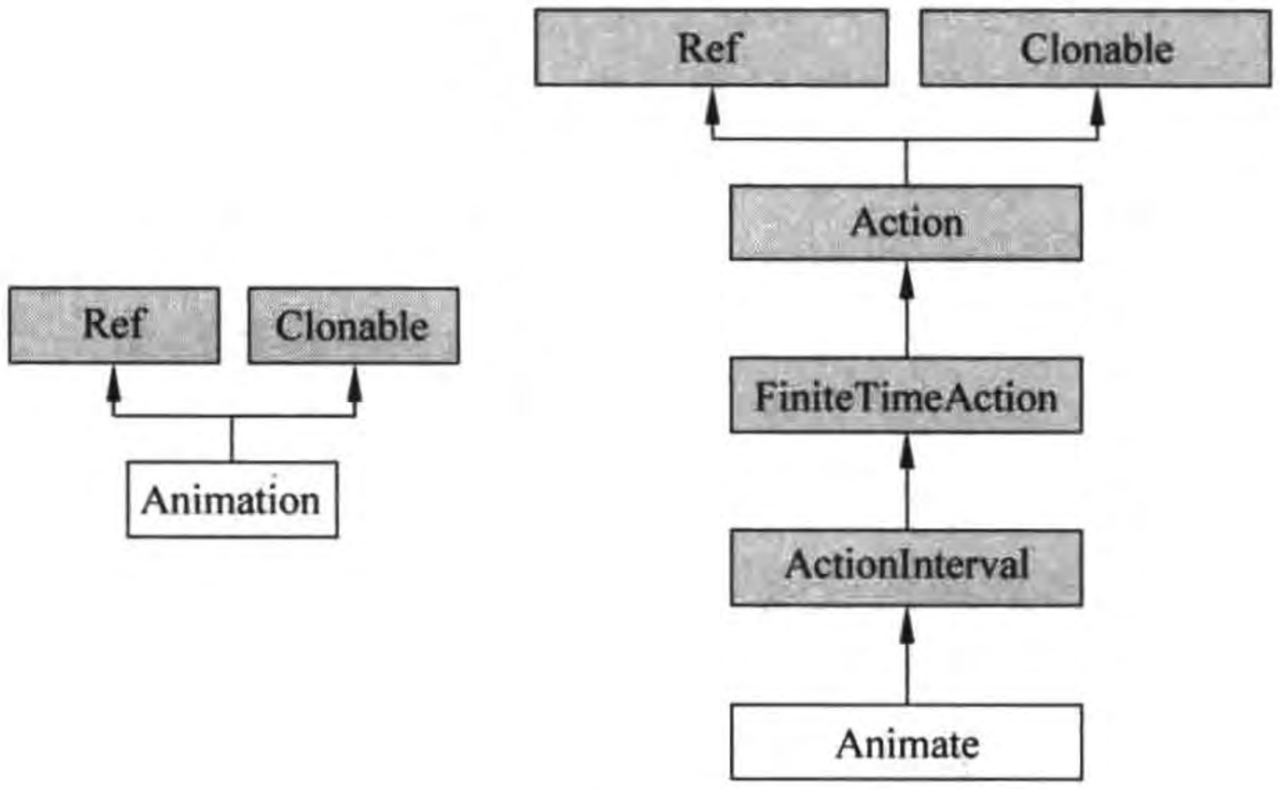
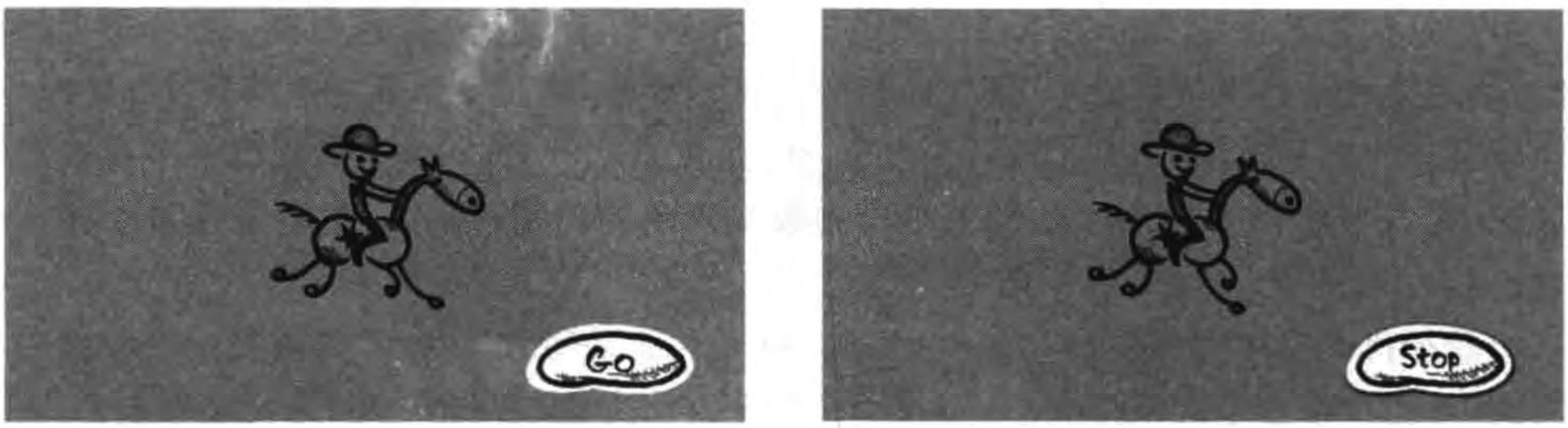


图 8-18　帧动画相关类图



(a)　　(b)

图 8-19　帧动画实例

GameScene.lua 文件的代码如下：

```
local isPlaying = false                    -- 播放标识                    ①
local size = cc.Director:getInstance():getWinSize()

...

-- create layer
function GameScene:createLayer()
local layer = cc.Layer:create()

local spriteFrame = cc.SpriteFrameCache:getInstance()
spriteFrame:addSpriteFrames("run.plist")

local bg = cc.Sprite:createWithSpriteFrameName("background.png")
bg:setPosition(cc.p(size.width/2, size.height/2))
layer:addChild(bg)

local sprite = cc.Sprite:createWithSpriteFrameName("h1.png")
sprite:setPosition(cc.p(size.width/2, size.height/2))
layer:addChild(sprite)
```

```
-- toggle 菜单
local goSprite = cc.Sprite:createWithSpriteFrameName("go.png")
local stopSprite = cc.Sprite:createWithSpriteFrameName("stop.png")

local goToggleMenuItem = cc.MenuItemSprite:create(goSprite, goSprite)
local stopToggleMenuItem = cc.MenuItemSprite:create(stopSprite, stopSprite)
local toggleMenuItem = cc.MenuItemToggle:create(goToggleMenuItem,
                                                stopToggleMenuItem)
toggleMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(930, 540)))

local mn = cc.Menu:create(toggleMenuItem)
mn:setPosition(cc.p(0, 0))
layer:addChild(mn)

local function OnAction(menuItemSender)

    if not isPlaying then

        --////////////////动画开始//////////////////////
        local animation = cc.Animation:create()                                    ②
        for i = 1,4 do
            local frameName = string.format("h%d.png",i)                           ③
            cclog("frameName = %s",frameName)
            local spriteFrame = spriteFrame:getSpriteFrameByName(frameName)        ④
            animation:addSpriteFrame(spriteFrame)                                  ⑤
        end

        animation:setDelayPerUnit(0.15)              -- 设置两个帧播放时间         ⑥
        animation:setRestoreOriginalFrame(true)      -- 动画执行后还原初始状态     ⑦

        local action = cc.Animate:create(animation)                                ⑧
        sprite:runAction(cc.RepeatForever:create(action))                          ⑨
        --////////////////动画结束//////////////////////
        isPlaying = true
    else
        sprite:stopAllActions()                                                    ⑩
        isPlaying = false
    end
end
toggleMenuItem:registerScriptTapHandler(OnAction)

return layer
end

return GameScene
```

上述第①行代码是声明一个布尔变量 isPlaying，用来保存播放状态，true 表示正在播放，false 表示停止播放。

第②行代码是创建一个 Animation 对象，它是动画对象，然后通过循环将各个帧图片放到 Animation 对象中。第③行代码是获得帧图片的文件名，string. format("h%d. png",i)是对字符串进行格式化。第④行代码是通过帧名创建精灵帧对象。第⑤行代码把精灵帧对

象添加到 Animation 对象中。

第⑥行代码 animation:setDelayPerUnit(0.15)是设置两个帧播放时间,这个动画播放的是 4 帧。第⑦行代码 animation:setRestoreOriginalFrame(true)是动画执行完成是否还原到初始状态。第⑧行代码是通过一个 Animation 对象创建 Animate 对象。第⑨行代码 cc.Animate:create(animation)是执行动画动作,无限循环方式。

第⑩行代码 sprite:stopAllActions()是停止所有的动作。

本章小结

通过对本章的学习,可以使广大读者熟悉 Cocos2d-x Lua API 中动作、特效和动画等动态特性。其中动作介绍了瞬时动作、间隔动作、组合动作、动作速度控制以及函数调用等,在特效部分介绍了网格动作,动画部分主要介绍了帧动画。

第9章

用户事件

在移动平台中用户输入的方式有触摸屏幕、键盘输入和各种传感器(如加速度计和麦克风等)。这些用户输入被封装成为事件,例如,在 iOS 平台有触摸事件和加速度事件等,在 Android 和 Windows Phone 平台有触摸事件、键盘事件和加速度事件等。

Cocos2d-x Lua API 具有跨平台特点,能够接收并处理的事件包括触摸事件、键盘事件、鼠标事件、加速度事件和自定义事件等。需要注意的是,在 Cocos2d-x Lua API 中有些事件在一些平台是无法接收的,这跟平台的硬件有关系,例如,在 iOS 平台就无法接收键盘事件,而在 PC 和 Mac OS X 平台不能接收触摸事件和加速度事件。

9.1 事件处理机制

在很多图形用户技术中,事件处理机制一般都有 3 个重要的角色:事件、事件源和事件处理者。事件源是事件发生的场所,通常就是各个视图或控件,事件处理者是接收事件并对其进行处理的一段程序。

在 Cocos2d-x Lua API 事件处理机制中也有这 3 个角色。

1. 事件

事件类是 Event,它的类图如图 9-1 所示,它的子类有:EventTouch(触摸事件)、EventMouse(鼠标事件)、EventKeyboard(键盘事件)、EventAcceleration(加速度事件)和 EventCustom(自定义),以及 EventCustom 的子类 PhysicsContact(物理引擎中接触事件)。

2. 事件源

事件源是 Cocos2d-x Lua API 中的精灵、层、菜单等节点对象。

3. 事件处理者

Cocos2d-x 中的事件处理者是事件监听器类 EventListener,类图如图 9-2 所示,它的子类有:EventListenerTouchOneByOne(单点触摸事件监听器)、EventListenerTouchAllAtOnce(多点触摸事件监听器)、EventListenerKeyboard(键盘事件监听器)、EventListenerMouse(鼠标事件监听器)、EventListenerAcceleration(加速度事件监听器)、EventListenerController(游戏手柄事件监听器)、EventListenerFocus(焦点事件监听器)和 Even-

tListenerCustom(自定义事件监听器),以及 EventListenerCustom 的子类 EventListener-PhysicsContact(物理引擎中接触事件监听器)。

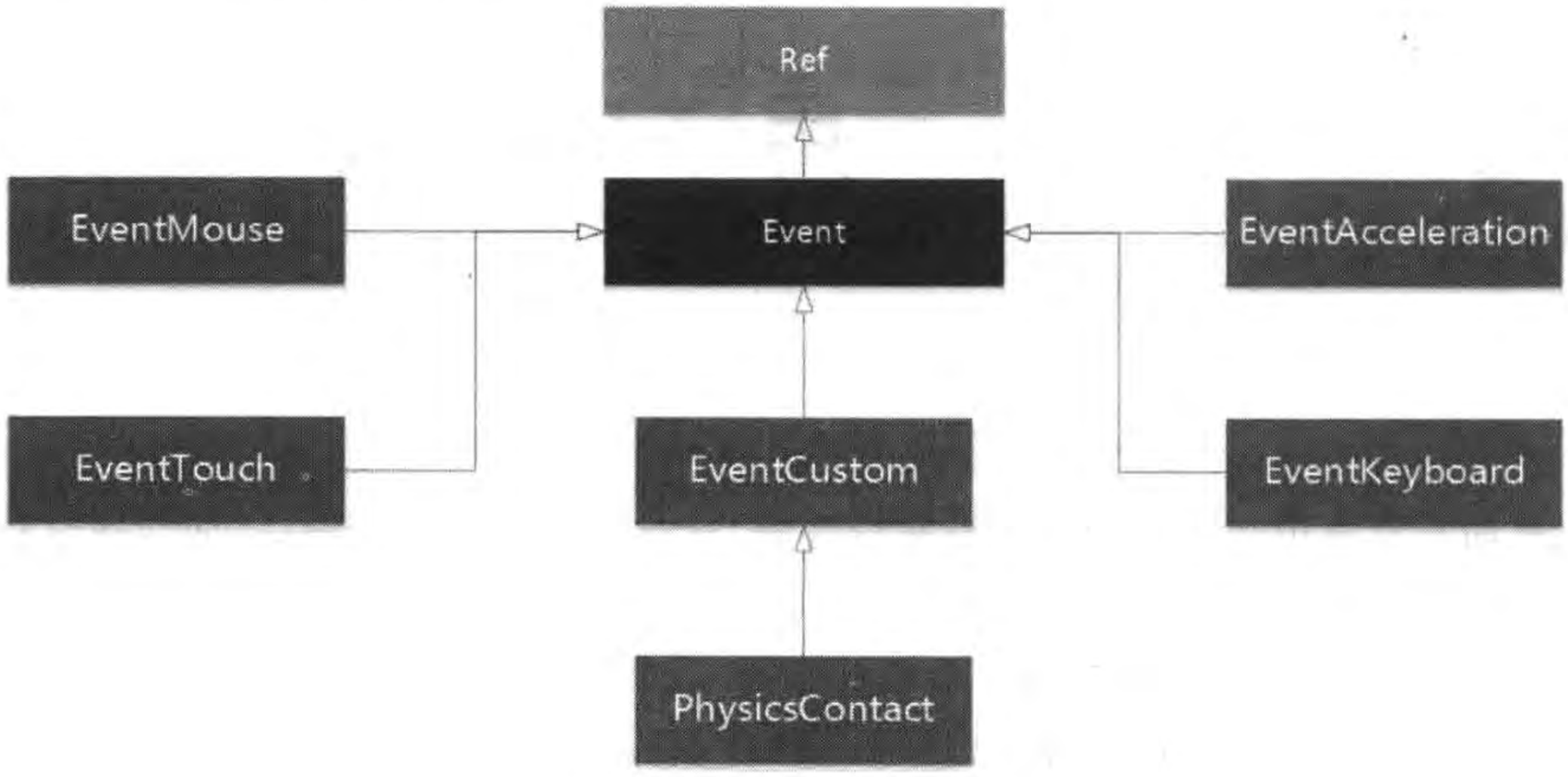


图 9-1　事件类图

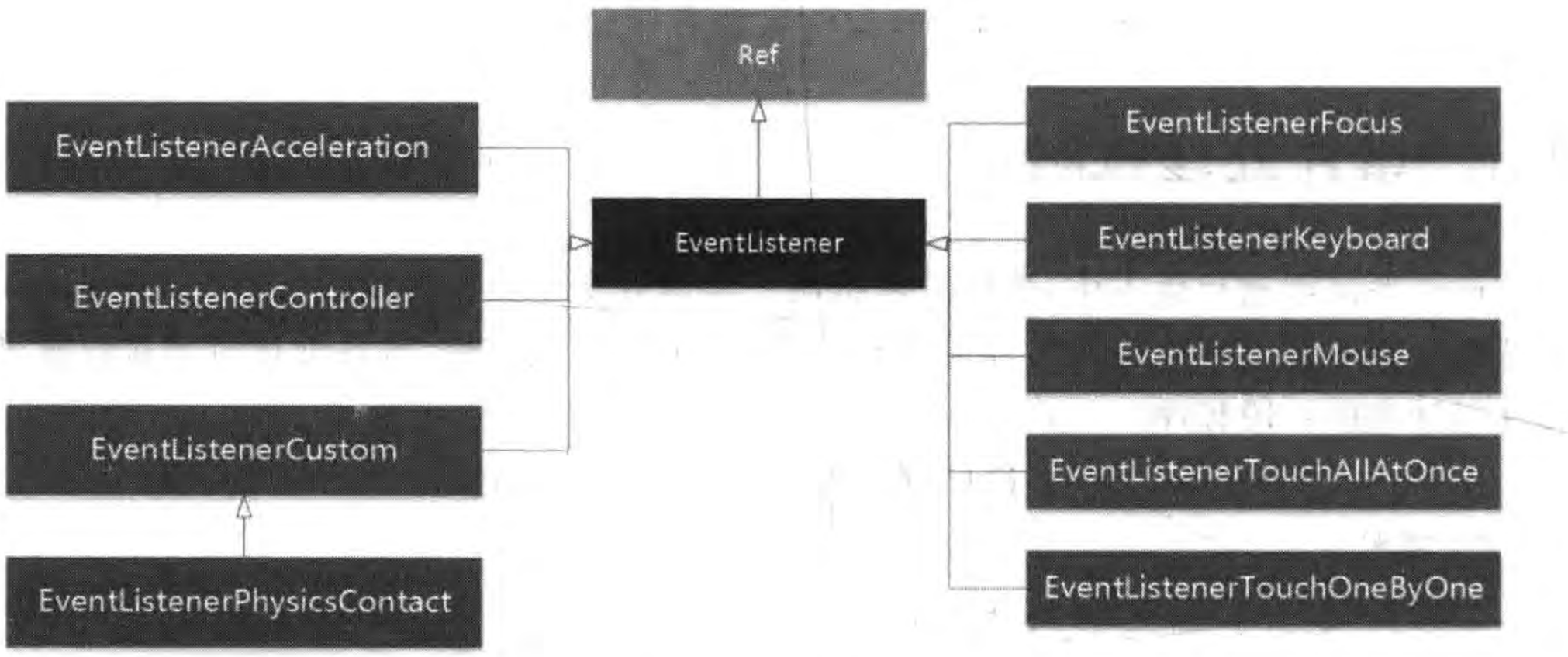


图 9-2　事件监听器类图

9.1.1　事件分发器

从上面的类图和命名上可以看出事件监听器与事件具有对应关系,例如,键盘事件(EventKeyboard)只能由键盘事件监听器(EventListenerKeyboard)处理,它们之间需要在程序中建立关系,这种关系的建立过程被称为注册监听器。Cocos2d-x Lua API 提供一个事件分发器(EventDispatcher)负责管理这种关系,具体来说事件分发器负责注册监听器、注销监听器和事件分发。

EventDispatcher 类采用单例模式设计,通过 cc. Director:getInstance():getEventDis-

patcher()语句获得事件分发器 EventDispatcher 对象。

EventDispatcher 类中添加事件监听器到事件分发器函数如下：

(1) addEventListenerWithFixedPriority(listener, fixedPriority)。指定固定的事件优先级注册监听器，事件优先级决定事件响应的优先级别，值越小优先级越高。

(2) addEventListenerWithSceneGraphPriority(listener, node)。把精灵显示优先级作为事件优先级，如图 9-3 所示的实例精灵 BoxC 优先级是最高的，按照精灵显示的顺序 BoxC 在最前面。参数 node 是要触摸的精灵对象。

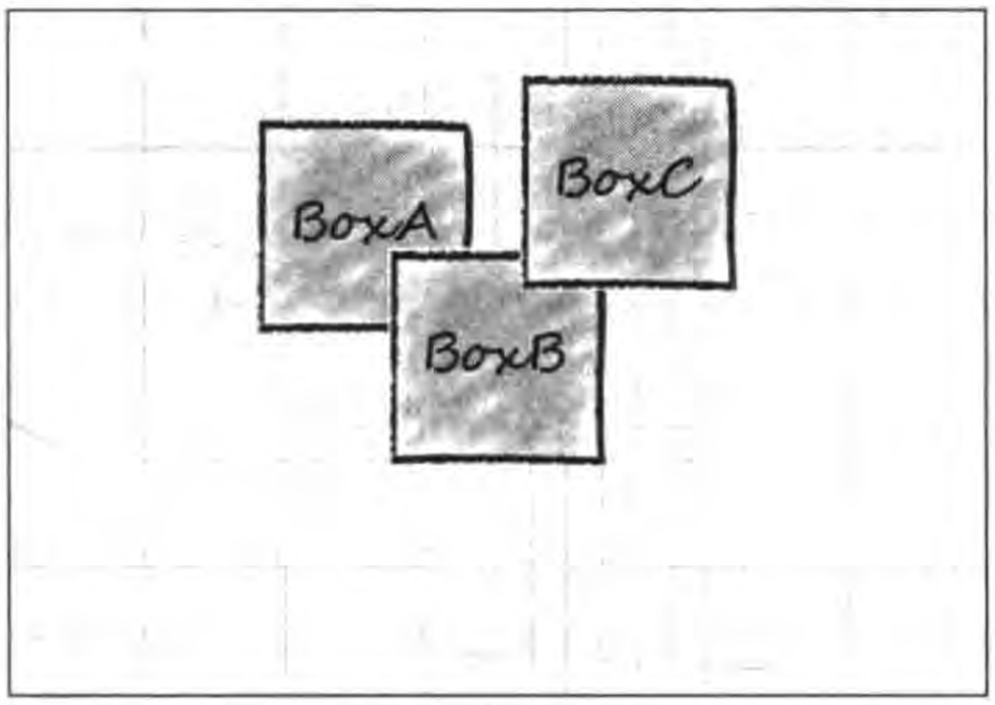


图 9-3　精灵显示优先级作为事件优先级

当不再进行事件响应的时候应该注销事件监听器，主要的注销函数如下：

(1) removeEventListener(listener)。注销指定的事件监听器。

(2) removeCustomEventListeners(customEventName)。注销自定义事件监听器。

(3) removeAllEventListeners()。注销所有事件监听器，需要注意的是使用该函数之后，菜单也不能响应事件了，因为它也需要接受触摸事件。

事件监听器注册之后，如果不再使用需要注销，具体的在什么事件中注销，要看注册的位置。如果注册的位置在 ctor 构造函数中，也就是在 init 事件中，那么注销过程可以在 cleanup 事件中处理，或者不处理，因为监听器所在的 Node 销毁了，监听器也会销毁。但是如果注册的位置在 enter 事件中，需要在 exit 事件中注销。如果注册的位置在 enterTransitionFinish 事件中，需要在 exitTransitionDidStart 事件中注销。

9.1.2　触摸事件

理解一个触摸事件可以从时间和空间两方面考虑。

1. 触摸事件的时间

触摸事件的时间如图 9-4 所示，可以用不同的“按下”、“移动”和“抬起”等分别表示触摸是否刚刚开始、是否正在移动或处于静止状态，以及何时结束。此外，触摸事件的不同阶段都可以有单点触摸或多点触摸，是否支持多点触摸还要看设备和平台。

触摸事件有两个事件监听器：EventListenerTouchOneByOne 和 EventListener-

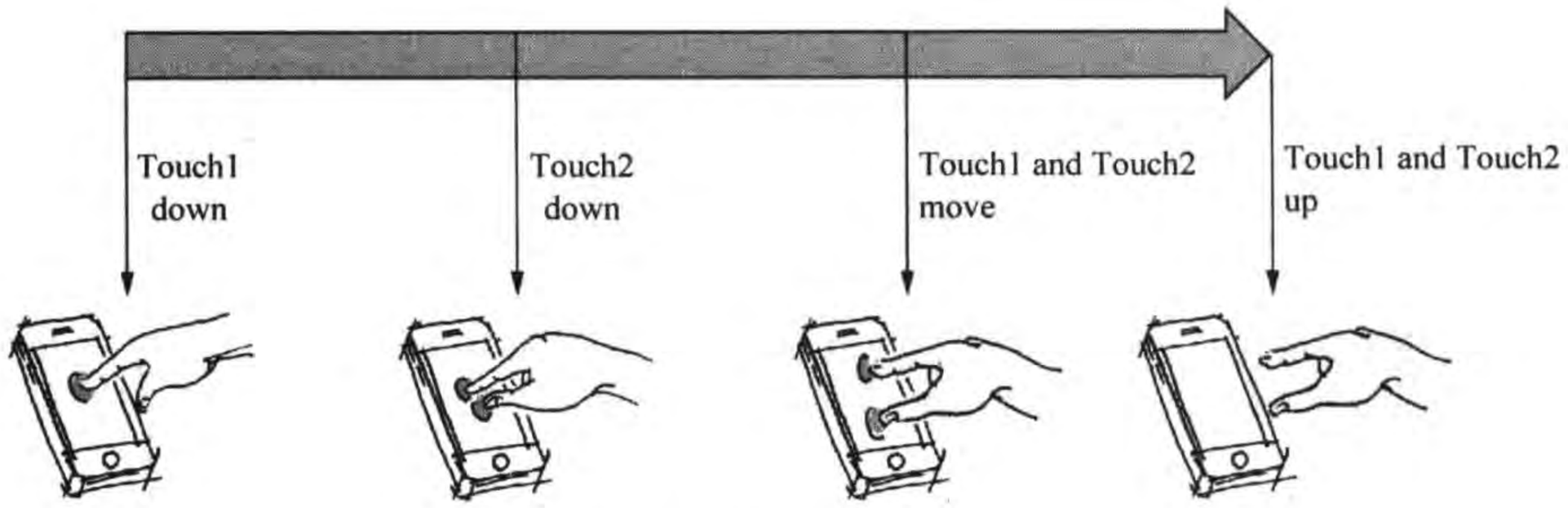


图 9-4　触摸事件的阶段

TouchAllAtOnce,分别对应单点触摸和多点触摸。这些监听器有一些触摸事件响应属性,这些属性对应着触摸事件不同阶段。通过设置,这些属性能够实现事件与事件处理者函数的关联。

EventListenerTouchOneByOne 中触摸事件响应属性如下:

(1) EVENT_TOUCH_BEGAN。当一个手指触碰屏幕时触发该事件所指定的函数。如果函数返回值为 true,则可以触发后面的两个事件(EVENT_TOUCH_MOVED 和 EVENT_TOUCH_ENDED)所指定的函数,否则不回调。

(2) EVENT_TOUCH_MOVED。当一个手指在屏幕移动时触发该事件所指定的函数。

(3) EVENT_TOUCH_ENDED。当一个手指离开屏幕时触发该事件所指定的函数。

(4) EVENT_TOUCH_CANCELLED。当单点触摸事件被取消时触发该事件所指定的函数。

EventListenerTouchAllAtOnce 中触摸事件响应属性如下:

(1) EVENT_TOUCHES_BEGAN。当多个手指触碰屏幕时触发该事件所指定的函数。

(2) EVENT_TOUCHES_MOVED。当多个手指在屏幕上移动时触发该事件所指定的函数。

(3) EVENT_TOUCHES_ENDED。当多个手指离开屏幕时触发该事件所指定的函数。

(4) EVENT_TOUCHES_CANCELLED。当多点触摸事件被取消时触发该事件所指定的函数。

使用这些属性的代码片段演示了它们的使用,代码如下:

```
local function touchBegan(touch, event)
    return false
end

local function touchMoved(touch, event)
    cclog("touchMoved")
```

```
end

local function touchEnded(touch, event)
    cclog("touchEnded")
end

local function touchCancelled(touch, event)
    cclog("touchCancelled ")
end

local listener = cc.EventListenerTouchOneByOne:create()                    ①
listener:registerScriptHandler(touchBegan,cc.Handler.EVENT_TOUCH_BEGAN )   ②
listener:registerScriptHandler(touchMoved,cc.Handler.EVENT_TOUCH_MOVED )
listener:registerScriptHandler(touchEnded,cc.Handler.EVENT_TOUCH_ENDED )
listener:registerScriptHandler(onTouchCancelled,cc.Handler.EVENT_TOUCH_CANCELLED )
local eventDispatcher = pLayer:getEventDispatcher()
eventDispatcher:addEventListenerWithSceneGraphPriority(listener, pLayer)
```

上述第①行代码 cc.EventListenerTouchOneByOne:create()创建单点触摸事件监听器对象。

第②行代码注册 EVENT_TOUCH_BEGAN 触摸事件，当事件触发时回调 touchBegan 函数。该函数是前面定义的 touchBegan(touch, event)函数。其他触摸事件的阶段也需要采用类似的代码，这里不再赘述。

2. 触摸事件的空间

空间就是每个触摸点(Touch)对象包含的当前位置信息，以及之前的位置信息(如果有的话)，下面的函数是可以获得触摸点之前的位置信息：

```
getPreviousLocationInView()      -- UI 坐标
getPreviousLocation()            -- OpenGL 坐标
```

下面的函数可以获得触摸点当前的位置信息。

```
getLocationInView()              -- UI 坐标
getLocation()                    -- OpenGL 坐标
```

9.1.3　实例：单点触摸事件

为了让读者掌握 Cocos2d-x Lua API 中的事件机制，下面以触摸事件为例，使用事件分发器实现单点触摸事件。该实例如图 9-3 所示，场景中有 3 个方块精灵，显示顺序如图 9-3 所示，拖曳它们可以移动，事件响应优先级按照它们的显示顺序。

GameScene.lua 文件中 GameScene 初始化相关代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    --贴图的纹理图片宽高必须是 2 的 n 次幂,128×128
    local bg = cc.Sprite:create("BackgroundTile.png", cc.rect(0, 0, size.width, size.height)) ①
    --贴图的纹理参数,水平重复平铺,垂直重复平铺
```

```
    bg:getTexture():setTexParameters(gl.LINEAR, gl.LINEAR, gl.REPEAT, gl.REPEAT)          ②
    bg:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(bg, 0)                                                                  ③

    local boxA = cc.Sprite:create("BoxA2.png")                                             ④
    boxA:setPosition(cc.p(size.width/2 - 120 , size.height/2 + 120))
    layer:addChild(boxA, 10, kBoxA_Tag)

    local boxB = cc.Sprite:create("BoxB2.png")
    boxB:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(boxB, 20, kBoxB_Tag)

    local boxC = cc.Sprite:create("BoxC2.png")
    boxC:setPosition(cc.p(size.width/2 + 120, size.height/2 + 160))
    layer:addChild(boxC, 30, kBoxC_Tag)                                                    ⑤

    -- 创建一个事件监听器 OneByOne 为单点触摸
    local listener1 = cc.EventListenerTouchOneByOne:create()                               ⑥
    -- 设置是否吞没事件,在 onTouchBegan 方法返回 true 时吞没
    listener1:setSwallowTouches(true)                                                      ⑦
    -- EVENT_TOUCH_BEGAN 事件回调函数
    listener1:registerScriptHandler(touchBegan,cc.Handler.EVENT_TOUCH_BEGAN )              ⑧
    -- EVENT_TOUCH_MOVED 事件回调函数
    listener1:registerScriptHandler(touchMoved,cc.Handler.EVENT_TOUCH_MOVED )              ⑨
    -- EVENT_TOUCH_ENDED 事件回调函数
    listener1:registerScriptHandler(touchEnded,cc.Handler.EVENT_TOUCH_ENDED )              ⑩

    local eventDispatcher = cc.Director:getInstance():getEventDispatcher()                 ⑪
    -- 添加监听器
    eventDispatcher:addEventListenerWithSceneGraphPriority(listener1, boxA)                ⑫
    local listener2 = listener1:clone()
    eventDispatcher:addEventListenerWithSceneGraphPriority(listener2, boxB)                ⑬
    local listener3 = listener2:clone()
    eventDispatcher:addEventListenerWithSceneGraphPriority(listener3, boxC)                ⑭

    return layer
end
```

在场景的层中初始化了背景和 3 个方块精灵。第①～③行代码是创建并添加背景,如图 9-3 所示的背景是由一个 128×128 纹理图片(BackgroundTile.png)反复贴在图上,这样可以减少内存消耗,第①行代码是创建背景精灵对象,注意背景的大小仍然是整个屏幕。第②行代码是设置贴图的纹理,其中的 gl.REPEAT 是 OpenGL 所需参数,表示需要重复贴图。第③行代码是添加背景精灵到当前层。

第④和第⑤行代码是创建了 3 个方块精灵,在添加它们到当前层时使用含有 3 个参数的 addChild(child,localZOrder,tag)函数,这样可以通过 localZOrder 参数指定精灵的显示顺序。

第⑥行代码是创建一个单点触摸事件监听器对象。第⑦行代码是设置是否吞没事件,如果设置为 true,那么触摸事件不会传递给下一个 Node 对象。第⑧行代码是设置监听器的 EVENT_TOUCH_BEGAN 事件回调函数。第⑨行代码是设置监听器的 EVENT_

TOUCH_MOVED 事件回调函数。第⑩行代码是设置监听器的 EVENT_TOUCH_ENDED 事件回调函数。

第⑪～⑭行代码是添加监听器，其中第⑫行代码通过 addEventListenerWithSceneGraphPriority 函数为 boxA 精灵添加事件监听器，addEventListenerWithSceneGraphPriority 添加的事件优先级顺序与精灵显示顺序保持一致，当多个精灵发生重叠的时候，触摸到的是 localZOrder 最大的那个精灵。

第⑬和第⑭行代码是为另外两个精灵添加事件监听器，其中 listener1:clone()和 listener2:clone()用于获得新的事件监听器对象。因为每一个事件监听器只能被添加一次，addEventListenerWithSceneGraphPriority 和 addEventListenerWithFixedPriority 会在添加事件监听器时设置一个注册标识，一旦设置了注册标识，该监听器就不能再用于注册其他事件监听了，因此需要使用 clone()函数获得一个新的监听器对象，把这个新的监听器对象用于注册。

触摸事件回调函数代码如下：

```
local function touchBegan(touch, event)                                    ①
    cclog("touchBegan")
    -- 获取事件所绑定的 node
    local node = event:getCurrentTarget()                                  ②
    -- 获取当前点击点所在相对按钮的位置坐标
    local locationInNode = node:convertToNodeSpace(touch:getLocation())    ③
    local s = node:getContentSize()                                        ④
    local rect = cc.rect(0, 0, s.width, s.height)                          ⑤
    -- 点击范围判断检测
    if cc.rectContainsPoint(rect, locationInNode) then                     ⑥
        cclog("sprite x = %d, y = %d ", locationInNode.x, locationInNode.y)
        cclog("sprite tag = %d", node:getTag())
        node:runAction(cc.ScaleBy:create(0.06, 1.06))                      ⑦
        return true                                                        ⑧
    end

    return false
end

local function touchMoved(touch, event)                                    ⑨
    cclog("touchMoved")
    -- 获取事件所绑定的 node
    local node = event:getCurrentTarget()

    local currentPosX, currentPosY = node:getPosition()                    ⑩
    local diff = touch:getDelta()                                          ⑪
    -- 移动当前按钮精灵的坐标位置
    node:setPosition(cc.p(currentPosX + diff.x, currentPosY + diff.y))     ⑫

end

local function touchEnded(touch, event)                                    ⑬
    cclog("touchEnded")
    -- 获取事件所绑定的 node
```

```
    local node = event:getCurrentTarget()

    local locationInNode = node:convertToNodeSpace(touch:getLocation())
    local s = node:getContentSize()
    local rect = cc.rect(0, 0, s.width, s.height)

    -- 点击范围判断检测
    if cc.rectContainsPoint(rect, locationInNode) then
        cclog("sprite x = %f, y = %f ", locationInNode.x, locationInNode.y)
        cclog("sprite tag = %d", node:getTag())
        node:runAction(cc.ScaleTo:create(0.06, 1.0))
        end
    end
```

上述第①行代码是定义回调函数 touchBegan。第②行代码是获取事件所绑定的精灵对象,其中 event:getCurrentTarget()语句返回当前触摸的 Node 对象。第③行代码是获取当前触摸点相对于 node 触摸对象的坐标。第④行代码是获得 node 触摸对象的尺寸。第⑤行代码是通过 node 触摸对象的尺寸创建 rect 变量。第⑥行代码 cc. rectContainsPoint (rect, locationInNode)是判断触摸点是否在 node 触摸对象范围内。第⑦行代码是放大 node 触摸对象。第⑧行代码返回 true,表示可以回调第⑨行代码 touchMoved 函数和第⑬行代码 touchEnded 函数。

在 touchMoved 回调函数中,第⑩行代码是 node 触摸对象的当前位置,并把位置坐标赋值给 currentPosX 和 currentPosY 两个变量。第⑪行代码 local diff=touch:getDelta()是获得 node 触摸对象当前位置和上次位置之差。第⑫行代码是通过 diff 重新设置 node 触摸对象的位置。

9.1.4 实例: 多点触摸事件

上一节介绍了单点触摸事件,本节通过一个实例来介绍多点触摸事件。将图 9-3 所示的实例简化一下,保持场景中有一个方块精灵,如图 9-5 所示,可以拖曳和移动精灵。

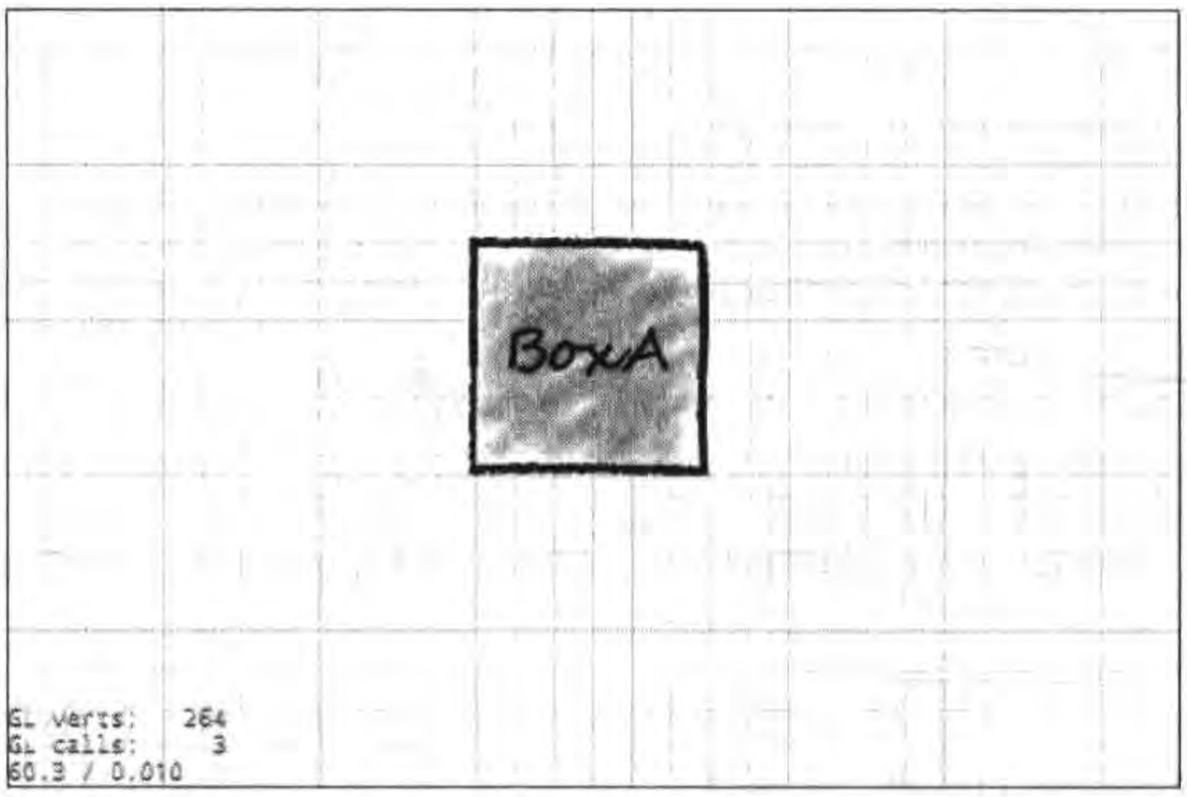


图 9-5 iOS 多点触摸事件实例

GameScene.lua 文件 GameScene 初始化相关代码如下：

```
-- 点击范围判断检测
local function isContainsPoint(touch, node)                                    ①
    local locationInNode = node:convertToNodeSpace(touch:getLocation())
    local s = node:getContentSize()
    local rect = cc.rect(0, 0, s.width, s.height)

    if cc.rectContainsPoint(rect, locationInNode) then
        return true
    end
    return false
end

local function touchesBegan(touches, event)                                    ②
    cclog("touchBegan")
    -- 获取事件所绑定的 node
    local node = event:getCurrentTarget()

    if isContainsPoint(touches[1], node) then                                  ③
        cclog("sprite tag = %d", node:getTag())
        node:runAction(cc.ScaleBy:create(0.06, 1.06))
    end
end

local function touchesMoved(touches, event)                                    ④
    cclog("touchMoved")
    -- 获取事件所绑定的 node
    local node = event:getCurrentTarget()

    if isContainsPoint(touches[1], node) then                                  ⑤
        local currentPosX, currentPosY = node:getPosition()
        local diff = touches[1]:getDelta()
        -- 移动当前按钮精灵的坐标位置
        node:setPosition(cc.p(currentPosX + diff.x, currentPosY + diff.y))
    end
end

local function touchesEnded(touches, event)                                    ⑥
    cclog("touchEnded")
    -- 获取事件所绑定的 node
    local node = event:getCurrentTarget()

    if isContainsPoint(touches[1], node) then                                  ⑦
        node:runAction(cc.ScaleTo:create(0.06, 1.0))
    end
end

-- create layer
function GameScene:createLayer()

    local layer = cc.Layer:create()

    --贴图的纹理图片宽高必须是 2 的 n 次幂,128×128
```

```
    local bg = cc.Sprite:create("BackgroundTile.png", cc.rect(0, 0, size.width, size.height))
    -- 贴图的纹理参数,水平重复平铺,垂直重复平铺
    bg:getTexture():setTexParameters(gl.LINEAR, gl.LINEAR, gl.REPEAT, gl.REPEAT)
    bg:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(bg, 0)

    local boxA = cc.Sprite:create("BoxA2.png")
    boxA:setPosition(cc.p(size.width/2 - 120 , size.height/2 + 120))
    layer:addChild(boxA, 10, kBoxA_Tag)

    -- 创建一个事件监听器 OneByOne 为单点触摸
    local listener = cc.EventListenerTouchAllAtOnce:create()
    -- EVENT_TOUCHES_BEGAN 事件回调函数
    listener:registerScriptHandler(touchesBegan,cc.Handler.EVENT_TOUCHES_BEGAN )
    -- EVENT_TOUCHES_MOVED 事件回调函数
    listener:registerScriptHandler(touchesMoved,cc.Handler.EVENT_TOUCHES_MOVED )
    -- EVENT_TOUCHES_ENDED 事件回调函数
    listener:registerScriptHandler(touchesEnded,cc.Handler.EVENT_TOUCHES_ENDED )

    local eventDispatcher = self:getEventDispatcher()
    -- 添加监听器
    eventDispatcher:addEventListenerWithSceneGraphPriority(listener, boxA)

    return layer
end
```

上述第①行代码定义的 isContainsPoint(touch，node)函数是判断触摸点是否在 node 对象范围内,由于在触摸事件中需要多次用到这种判断,于是将这种判断封装成一个函数。

第②行代码的 touchesBegan(touches，event)函数是多点触摸事件 EVENT_TOUCHES_BEGAN 触发时回调的函数,其中第③行代码是使用 isContainsPoint 函数来判断第一个触摸点是否在 node 对象范围内,如果判断为 true 则放大 node 对象。

第④行代码的 touchesMoved(touches，event)函数是多点触摸事件 EVENT_TOUCHES_MOVED 触发时回调的函数,其中第⑤行代码是使用 isContainsPoint 函数来判断第一个触摸点是否在 node 对象范围内,如果判断为 true 则移动 node 对象。

第⑥行代码的 touchesEnded(touches，event)函数是多点触摸事件 EVENT_TOUCHES_ENDED 触发时回调的函数,其中第⑦行代码是使用 isContainsPoint 函数来判断第一个触摸点是否在 node 对象范围内,如果判断为 true 则缩小 node 对象。

9.1.5 键盘事件

Cocos2d-x Lua API 中的键盘事件与触摸事件不同,它没有空间方面信息。键盘事件不仅可以响应键盘,还可以响应设备的菜单。

键盘事件监听器是 EventListenerKeyboard。EventListenerKeyboard 中键盘事件响应属性如下:

(1) EVENT_KEYBOARD_PRESSED。当键按下时触发该事件所指定的函数。

(2) EVENT_KEYBOARD_RELEASED。当键抬起时触发该事件所指定的函数。

实现键盘事件处理的主要代码如下：

```
local function onKeyPressed(keyCode, event)                                    ①
    local buf = string.format("%d 键按下!",keyCode)
    local label = event:getCurrentTarget()                                     ②
    label:setString(buf)                                                       ③
end

local function onKeyReleased(keyCode, event)                                   ④
    local buf = string.format("%d 键抬起!",keyCode)
    local label = event:getCurrentTarget()
    label:setString(buf)
end

-- create layer
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local statusLabel = cc.Label:createWithSystemFont("没有键盘事件接收到!", "", 40)
    statusLabel:setAnchorPoint(cc.p(0.5, 0.5))
    statusLabel:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(statusLabel)

    -- 创建一个键盘监听器
    local listener = cc.EventListenerKeyboard:create()                         ⑤
    listener:registerScriptHandler(onKeyPressed,
                                   cc.Handler.EVENT_KEYBOARD_PRESSED )         ⑥
    listener:registerScriptHandler(onKeyReleased,
                                   cc.Handler.EVENT_KEYBOARD_RELEASED )        ⑦
    local eventDispatcher = self:getEventDispatcher()                          ⑧
    -- 添加监听器
    eventDispatcher:addEventListenerWithSceneGraphPriority(listener, statusLabel)  ⑨

    return layer
end
```

上述第①行代码 onKeyPressed 函数是在键按下时调用的，其中参数 keyCode 是键的编号，event 是键盘事件对象。第②行代码 event:getCurrentTarget()是获得键盘事件源。第③行代码是设置 label 控件的内容。第④行代码 onKeyReleased 函数是在键抬起时调用的，该函数的参数与 onKeyPressed 函数相同。

第⑤行代码是创建一个键盘监听器。第⑥行代码是注册 EVENT_KEYBOARD_PRESSED 事件，当事件触发时回调 onKeyPressed 函数。第⑦行代码是注册 EVENT_KEYBOARD_RELEASED 事件，当事件触发时回调 onKeyReleased 函数。

第⑨行代码 addEventListenerWithSceneGraphPriority 是添加事件监听器，设置事件优先级。

9.2 加速度计与加速度事件

在很多移动设备的游戏中都使用到了加速度计，Cocos2d-x Lua API 提供了访问加速度计传感器的能力。本节首先介绍一下加速度计传感器，然后再介绍如何在 Cocos2d-x Lua API 中访问加速度计。

9.2.1 加速度计

加速度计是一种能够感应设备某个方向上线性加速度的传感器。广泛应用于航空、航海、宇航及武器的制导与控制中。线加速度计的种类很多，在 iOS 等移动设备中目前采用的是三轴加速度计，可以感应设备上 X、Y、Z 轴方向上线性加速度的变化。如图 9-6 所示，iOS 和 Android 等设备三轴加速度计的坐标系是右手坐标系，即设备竖直向上，正面朝向用户，水平向右为 X 轴正方向，竖直向上为 Y 轴正方向，Z 轴正方向是从设备指向用户方向。

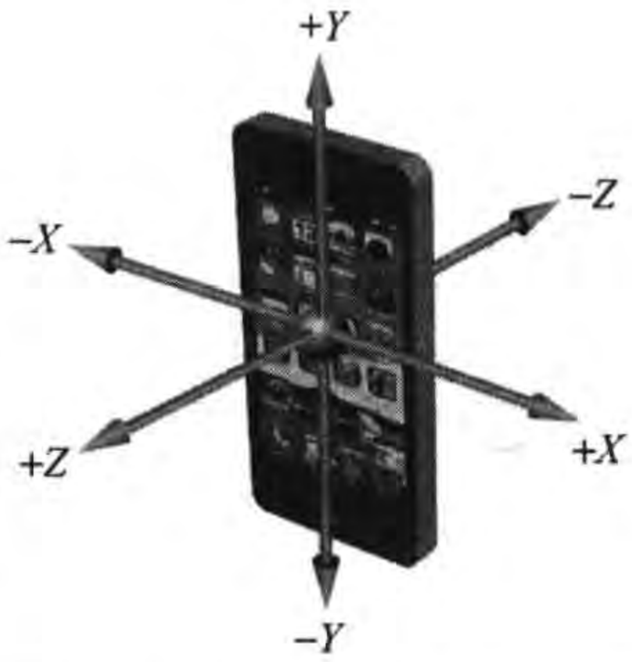


图 9-6 iOS 上三轴加速度计

提示 有人将加速度计称为重力加速度计，这种观点是错误的。作用于 3 个轴上的加速度是指所有加速度的总和，包括了由重力产生的加速度和用户移动设备产生的加速度。在设备静止的情况下，这时的加速度是重力加速度。

9.2.2 加速度计事件

在 Cocos2d-x Lua API 中提供了加速度计事件，可以帮助读者访问加速度计。加速度计事件需要设备的支持。与触摸事件类似，可以通过事件分发器和层加速度计事件两种方式访问加速度计数据。

使用事件分发器之前需要添加加速计事件监听器，在此之前需要先启用此硬件设备，使用如下代码：

```
Node:setAccelerometerEnabled(true)
```

加速度计事件监听器是 EventListenerAcceleration，在事件响应方面加速度计事件监听器与其他不同，事件响应可以通过静态 create 函数指定，create 代码如下：

```
cc.EventListenerAcceleration:create(accelerometerListener)
```

参数 accelerometerListener 是回调函数，回调函数 accelerometerListener(event，x，y，z，timestamp)其中有多个参数，其中参数说明如下：

(1) x：获得 x 轴方向上的加速度。单位为 g，1g=9.81ms^{-2}。

(2) y：获得 y 轴方向上的加速度。

(3) z：获得 z 轴方向上的加速度。

(4) timestamp：时间戳属性，用来表示事件发生的相对时间。

9.2.3　实例：运动的小球

下面通过一个实例介绍通过层加速度计事件实现访问加速度计。该实例场景如图 9-7 所示，场景中有一个小球，首先把移动设备水平放置，屏幕向上，然后左右晃动移动设备来改变小球的位置。

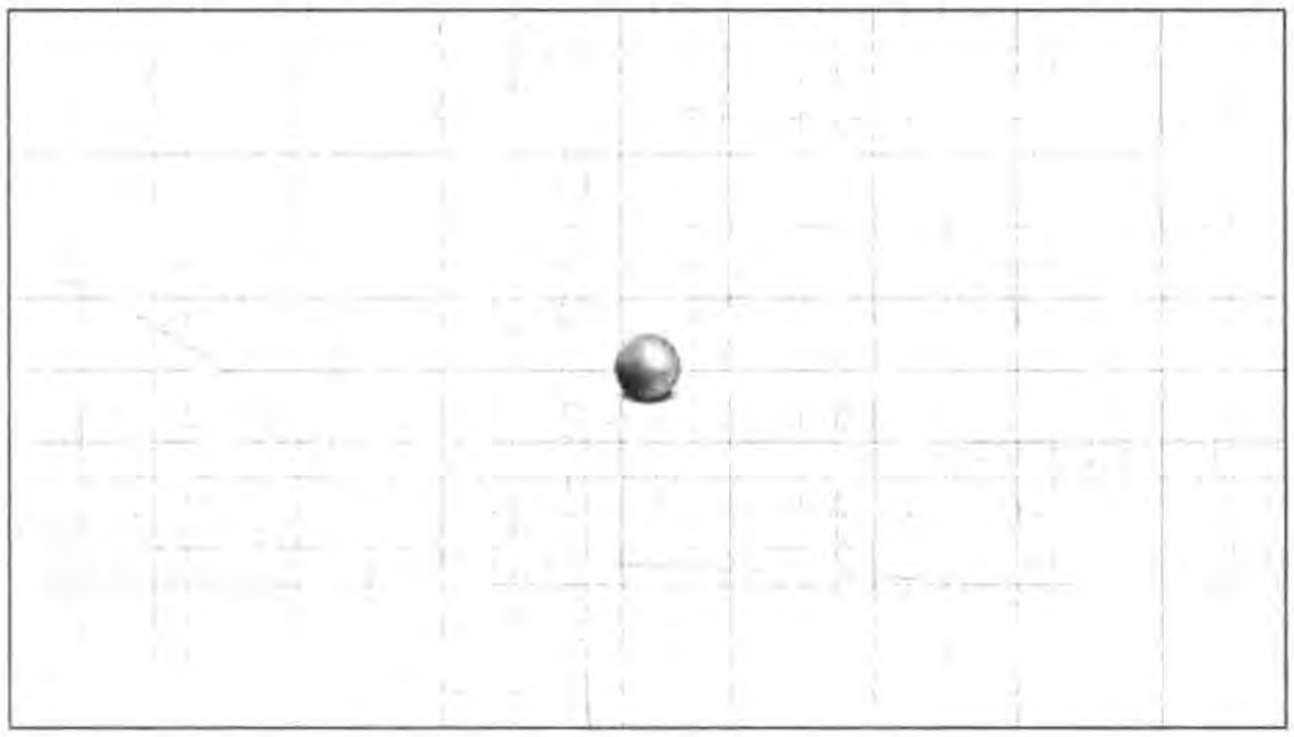


图 9-7　访问加速度计实例

GameScene.lua 文件的代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()
    layer:setAccelerometerEnabled(true)                                    ①

    --贴图的纹理图片宽高必须是 2 的 n 次幂,128 × 128
    local bg = cc.Sprite:create("BackgroundTile.png", cc.rect(0, 0, size.width, size.height))
    --贴图的纹理参数,水平重复平铺,垂直重复平铺
    bg:getTexture():setTexParameters(gl.LINEAR, gl.LINEAR, gl.REPEAT, gl.REPEAT)
    bg:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(bg, 0)

    local ball = cc.Sprite:create("Ball.png")
    ball:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(ball, 10, kBall_Tag)

    local function accelerometerListener(event,x,y,z,timestamp)            ②
        cclog("{x = %f, y = %f}", x, y)

        local ball = layer:getChildByTag(kBall_Tag)                        ③
        local s = ball:getContentSize()                                    ④
        local p0x,p0y = ball:getPosition()                                 ⑤
```

```
local p1x = p0x + x * SPEED                                          ⑥
if (p1x - s.width/2) < 0 then                                        ⑦
    p1x = s.width/2                                                  ⑧
end

if (p1x + s.width / 2) > size.width then                             ⑨
    p1x = size.width - s.width / 2                                   ⑩
end

local p1y = p0y + y * SPEED
if (p1y - s.height/2) < 0 then
    p1y = s.height/2
end
if (p1y + s.height/2) > size.height then
    p1y = size.height - s.height/2
end
ball:runAction(cc.Place:create(cc.p( p1x, p1y)))                     ⑪

end

-- 创建一个加速度监听器
local listerner = cc.EventListenerAcceleration:create(accelerometerListener)
self:getEventDispatcher():addEventListenerWithSceneGraphPriority(listerner,ball)

return layer
end
```

上述第①行代码开启当前层对象的加速计事件。第②行代码是声明加速度计事件回调函数,在该函数中的第③行代码 layer:getChildByTag(kBall_Tag)是通过 tag 属性获得小球精灵对象。

第④行代码 ball:getContentSize()是获得小球尺寸大小。第⑤行代码 local p0x,p0y= ball:getPosition()是获得小球的位置,并分别赋值给 p0x 和 p0y 两个变量。第⑥行代码 local p1x= p0x+x * SPEED 是获得小球的 x 轴方向移动的位置,但是需要考虑左右超出屏幕的情况。第⑦行代码(p1x-s. width/2)<0 是判断是否超出左边屏幕,这种情况下需要通过第⑧行代码 p1x=s. width/2 重新设置它的 x 轴坐标。第⑨行代码(p1x+s. width/2)>size. width 是判断是否超出右边屏幕,这种情况下需要通过第⑩行代码 p1x=size. width-s. width / 2 重新设置它的 x 轴坐标。类似的判断 y 轴也需要,代码就不再解释了。

重新获得小球的坐标位置后,通过第⑪行代码 ball:runAction(cc. Place:create(cc. p(p1x, p1y)))执行一个动作,使小球移动到新的位置。

本章小结

通过对本章的学习,使得读者了解 Cocos2d-x Lua API 的用户输入事件处理,这些事件包括触摸事件、键盘事件、鼠标事件、加速度事件和自定义事件等。

第二篇　进　阶　篇

本篇共 6 章，介绍 Cocos2d-x 游戏开发高级内容，其中包括：游戏音乐与音效、粒子系统、瓦片地图、物理引擎、GUI 控件和 3D 特性。

本篇各章如下：

第 10 章　游戏背景音乐与音效

第 11 章　粒子系统

第 12 章　瓦片地图

第 13 章　物理引擎

第 14 章　Cocos2d-x GUI 控件

第 15 章　Cocos2d-x 中的 3D 特性

第 10 章

游戏背景音乐与音效

游戏中音频的处理非常重要，它分为背景音乐播放与音效播放。背景音乐是长时间循环播放的，不能多个同时播放，它会长时间占用较大的内存。而音效是短的声音，例如单击按钮、发射子弹等声音，能多个同时播放，占用内存较小。在 Cocos2d-x Lua API 中提供了一个音频引擎——Audio Engine，通过引擎能够很好地控制游戏背景音乐与音效优化播放。

10.1 Cocos2d-x Lua API 中音频文件

Cocos2d-x Lua API 是跨平台的游戏引擎，而各个平台支持的音频文件是不同的。首先介绍一下各个平台的音频文件。

10.1.1 音频文件介绍

音频多媒体文件主要是存放音频数据信息，音频文件在录制的过程中把声音信号通过音频编码变成音频数字信号并保存到某种格式文件中。在播放过程中再对音频文件解码，解码出的信号通过扬声器等设备就可以转换成音波。在音频文件编码的过程中数据量很大，所以有的文件格式对于数据进行了压缩，因此音频文件可以分为无损格式和有损格式。

无损格式指非压缩数据格式，文件很大，例如 WAV、AU、APE 等文件，一般不适合移动设备。

有损格式对数据进行了压缩，压缩后丢掉了一些数据，例如 MP3、WMA（Windows Media Audio）等文件。

1. WAV 文件

WAV 文件是目前最流行的无损压缩格式。WAV 文件的格式灵活，可以储存多种类型的音频数据。由于文件较大，不太适合于移动设备这些存储容量小的设备。

2. MP3 文件

MP3（MPEG Audio Layer 3）格式现在非常流行，MP3 是一种有损压缩格式，它尽可能地去掉人耳无法感觉和不敏感的部分。MP3 是利用 MPEG Audio Layer 3 将数据以 10∶1 甚

至 12：1 的压缩率，压缩成容量较小的文件。这么高的压缩比率非常适合于移动设备这些存储容量小的设备。

3. WMA 文件

WMA(Windows Media Audio)格式是微软发布的文件格式，也是有损压缩格式。它与 MP3 格式不分伯仲。在低比特率渲染情况下，WMA 格式显示出比 MP3 更多的优点，压缩率比 MP3 更高，音质更好。但是在高比特率渲染情况下 MP3 还是占有优势。

4. CAFF 文件

CAFF(Core Audio File Format)文件是苹果开发的专门用于 Mac OS X 和 iOS 系统的无压缩音频格式。它被设计用来替换旧的 WAV 格式。

5. AIFF 文件

AIFF(Audio Interchange File Format)文件是苹果开发的专业音频文件格式。AIFF 的压缩格式是 AIFF-C(或 AIFC)，将数据以 4：1 压缩率进行压缩，专门应用于 Mac OS X 和 iOS 系统。

6. MID 文件

MID 文件是 MIDI(Musical Instrument Digital Interface)格式，即专业音频文件格式，允许数字合成器和其他设备交换数据。MID 文件主要用于原始乐器作品、流行歌曲的业余表演、游戏音轨以及电子贺卡等。

7. Ogg 文件

Ogg 文件全称为 OGGVobis(oggVorbis)，是一种新的音频压缩格式，类似于 MP3 等的音乐格式。Ogg 是完全免费、开放和没有专利限制的。Ogg 文件格式可以不断地进行大小和音质的改良，而不影响原有的编码器或播放器。

10.1.2 Cocos2d-x 跨平台音频支持

Cocos2d-x Lua API 与 cocos2d-iphone 不同，它是为跨平台而设计的游戏引擎，因此也要考虑对于音频跨平台的支持。

Cocos2d-x Lua API 对于背景音乐播放各个平台格式支持如下：

(1) Android 平台与 android. media. MediaPlayer 所支持的格式相同。android. media. MediaPlayer 是 Android 多媒体播放类。

(2) iOS 平台推荐使用 MP3 和 CAFF 格式。

(3) Windows 平台支持 MIDI、WAV 和 MP3 格式。

(4) Windows Phone 8 平台支持 MIDI 和 WAV 格式。

Cocos2d-x Lua API 对于音效播放各个平台格式支持如下：

(1) Android 平台支持 Ogg 和 WAV 文件，但最好是 Ogg 文件。

(2) iOS 平台支持使用 CAFF 格式。

(3) Windows 平台支持 MIDI 和 WAV 文件。

(4) Windows Phone 8 平台支持 MIDI 和 WAV 格式。

10.2　使用 AudioEngine 引擎

Cocos2d-x Lua API 提供了一个音频 AudioEngine 引擎，AudioEngine 引擎可以独立于 Cocos2d-x Lua API 单独使用，它本质上封装了 OpenAL[①] 音频处理库。

AudioEngine 引擎有以下常用的函数：

(1) preloadMusic(pszFilePath)。预处理背景音乐文件，将压缩格式的文件进行解压处理，如 MP3 解压为 WAV。

(2) playMusic(pszFilePath, bLoop=false)。播放背景音乐，参数 bLoop 控制是否循环播放，默认情况下为 false。

(3) stopMusic()。停止播放背景音乐。

(4) pauseMusic()。暂停播放背景音乐。

(5) resumeMusic()。继续播放背景音乐。

(6) isMusicPlaying()。判断背景音乐是否正在播放。

(7) playEffect(pszFilePath)。播放音效。

(8) pauseEffect(nSoundId)。暂停播放音效，参数 nSoundId 是 playEffect 函数返回 ID。

(9) pauseAllEffects ()。暂停所有播放音效。

(10) resumeEffect(nSoundId)。继续播放音效，参数 nSoundId 是 playEffect 函数返回 ID。

(11) resumeAllEffects()。继续播放所有音效。

(12) stopEffect(nSoundId)。停止播放音效，参数 nSoundId 是 playEffect 函数返回 ID。

(13) stopAllEffects ()。停止所有播放音效。

(14) preloadEffect (pszFilePath)。预处理音效音频文件，将压缩格式的文件进行解压处理，如 MP3 解压为 WAV。

10.2.1　音频文件的预处理

无论是播放背景音乐还是音效，在播放之前都要进行预处理，这个过程是对音频文件进行解压等处理，预处理只需要在整个游戏运行过程中处理一次就可以了。如果不进行预处理，则会发现在第一次播放这个音频文件时候感觉很“卡”，用户体验不好。

预处理相关函数有两个：preloadMusic 和 preloadEffect。下面代码是预处理背景音乐和音频：

---

① OpenAL(Open Audio Library)是自由软件界的跨平台音频处理库。它设计给多通道三维位置音效的特效表现。其 API 风格模仿自 OpenGL。

```
--初始化  音乐
AudioEngine.preloadMusic( "sound/Synth.mp3" )
--初始化  音效
AudioEngine.preloadEffect( "sound/Blip.wav" )
```

这些预处理过程代码放置到什么地方比较适合呢？由于放置到任何一个场景层中，当进入到这个层时都比较“卡”，所以最好不要放置到场景层中，最好放置到 main.lua 的 main()函数中，因为它是程序的入口，代码如下：

```
local function main()
    collectgarbage("collect")
    -- avoid memory leak
    collectgarbage("setpause", 100)
    collectgarbage("setstepmul", 5000)

    cc.FileUtils:getInstance():addSearchPath("src")
    cc.FileUtils:getInstance():addSearchPath("res")
    cc.Director:getInstance():getOpenGLView():setDesignResolutionSize(1136, 640, 0)

    --初始化  音乐
    AudioEngine.preloadMusic( "sound/Synth.mp3" )
    --初始化  音效
    AudioEngine.preloadEffect( "sound/Blip.wav" )

    -- create scene
    local scene = require("GameScene")
    local gameScene = scene.create()

    if cc.Director:getInstance():getRunningScene() then
        cc.Director:getInstance():replaceScene(gameScene)
    else
        cc.Director:getInstance():runWithScene(gameScene)
    end

end
```

main.lua 的 main()函数在游戏启动时调用。在游戏启动时一般会有一个启动界面，启动界面一般会有一个延迟展示，这段时间是初始化的最佳时机。

10.2.2 播放背景音乐

背景音乐的播放代码如下：

```
AudioEngine.playMusic("sound/Jazz.mp3", true);
AudioEngine.stopMusic;
```

背景音乐的播放代码放置到什么地方比较适合呢？例如，在 SettingScene 场景中，主要代码如下：

```
function SettingScene:ctor()
    cclog("SettingScene init")
    --播放代码                                                    ①
```

```
--场景节点事件处理
local function onNodeEvent(event)
    if event == "enter" then
        self:onEnter()
    elseif event == "enterTransitionFinish" then
        self:onEnterTransitionFinish()
    elseif event == "exit" then
        self:onExit()
    elseif event == "exitTransitionStart" then
        self:onExitTransitionStart()
    elseif event == "cleanup" then
        self:cleanup()
    end
end

self:registerScriptHandler(onNodeEvent)
end


function SettingScene:onEnter()
    cclog("SettingSceneonEnter")
    --播放代码                                                    ②
end

function SettingScene:onEnterTransitionFinish()
    cclog("SettingScene onEnterTransitionFinish")
    --播放代码                                                    ③
end

function SettingScene:onExit()
    cclog("SettingScene onExit")
end

function SettingScene:onExitTransitionStart()
    cclog("SettingScene onExitTransitionStart")
end

function SettingScene:cleanup()
    cclog("SettingScene cleanup")
end
```

关于播放背景音乐，理论上可以将播放代码 AudioEngine. playMusic("sound/Jazz. mp3", true)放置到 3 个位置(代码中的①、②、③)。下面分别分析一下它们有什么不同。

1. 代码放到第①行

代码放到第①行，如果前面场景中没有调用背景音乐停止语句，则可以正常播放背景音乐。但是如果前面场景层 GameScene onExit 函数有调用背景音乐停止语句，那么会出现背景音乐播放几秒钟后停止的现象。

为了解释这个现象，可以参考 7.3.2 节多场景切换生命周期。使用 pushScene 函数实现从 GameScene 场景进入 SettingScene 场景，生命周期函数调用顺序如图 10-1 所示。

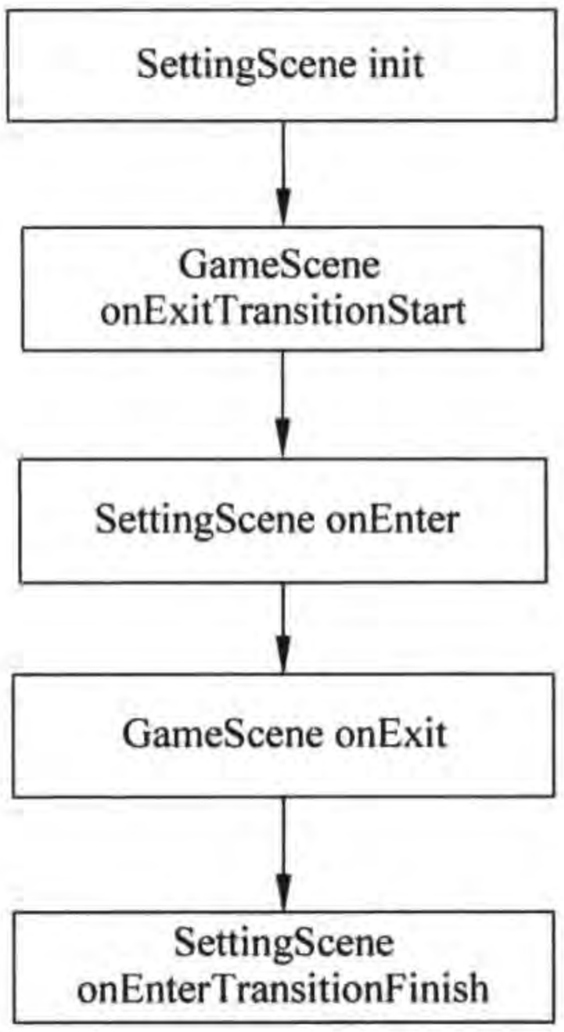


图 10-1　生命周期事件顺序

从图 10-1 可知，GameScene onExit 调用是在 SettingScene init 之后，这样当在 SettingScene init 中开始播放背景音乐后，过一会调用 GameScene onExit，将会停止背景音乐的播放。

2. 代码放到第②行

代码放到第②行，如果前面场景中没有调用背景音乐停止语句，则可以正常播放背景音乐。如果前面的场景层 GameScene onExit 函数有背景音乐停止语句，也会出现背景音乐播放几秒钟后停止的现象。原因与代码放到第①行情况一样。

3. 代码放到第③行

一般推荐代码放到第③行，因为 onEnterTransitionFinish 函数是在进入层而且过渡动画结束时调用，代码放到这里不用考虑前面场景是否有调用背景音乐停止语句，而且也不会出现先听到声音，后出现界面的现象。

综上所述，是否能够成功播放背景音乐，与前面场景是否有调用背景音乐停止语句有关，也与当前场景中播放代码放在哪个函数里有关。如果前面场景没有调用背景音乐停止语句，问题也就简单了，可以将播放代码放置在①、②、③任何一处。但是如果前面场景调用背景音乐停止语句，在 onEnterTransitionFinish 函数播放背景音乐会更好一些。

10.2.3　停止播放背景音乐

停止背景音乐播放代码放置到什么地方比较适合呢？例如，在 GameScene 场景中，主要代码如下：

```
function GameScene:ctor()
    cclog("GameScene init")
    -- 场景生命周期事件处理
    local function onNodeEvent(event)
        if event == "enter" then
            self:onEnter()
        elseif event == "enterTransitionFinish" then
            self:onEnterTransitionFinish()
        elseif event == "exit" then
            self:onExit()
        elseif event == "exitTransitionStart" then
            self:onExitTransitionStart()
        elseif event == "cleanup" then
            self:cleanup()
        end
    end
    self:registerScriptHandler(onNodeEvent)
```

```
end

function GameScene:onEnter()
    cclog("GameScene onEnter")
    AudioEngine.playMusic(MUSIC_FILE, true)
end

function GameScene:onEnterTransitionFinish()
    cclog("GameScene onEnterTransitionFinish")
end

function GameScene:onExit()
    cclog("GameScene onExit")
    --停止播放代码                                                    ①
end

function GameScene:onExitTransitionStart()
    cclog("GameScene onExitTransitionStart")
    --停止播放代码                                                    ②
end

function GameScene:cleanup()
    cclog("GameScene cleanup")
    --停止播放代码                                                    ③
end
```

关于停止背景音乐播放，理论上可以将停止播放代码 AudioEngine.stopMusic()放置到 3 个位置(代码中的①、②、③)。下面分别分析一下它们有什么不同。

1. 代码放到第①行

代码放到第①行(即在 GameScene onExit 函数)，如果后面场景中调用背景音乐播放，则可能导致播放背景音乐异常。关于这个问题在前面已经介绍过了。

2. 代码放到第②行

代码放到第②行(即在 GameScene onExitTransitionStart 函数)，如果后面场景 init 函数中调用背景音乐播放，则可能导致播放背景音乐异常。

3. 代码放到第③行

代码放到第③行(即在 GameScene cleanup 函数)，这个函数是在层对象清除时调用，在此处停止背景音乐播放是比较好的选择。但是如果采用 replaceScene 函数实现从 GameScene 场景进入 SettingScene 场景，情况会有所不同，这种情况下 SettingScene onEnterTransitionFinish 函数调用完成后会调用 GameScene cleanup 函数(参考图 7-6)，因此也会导致播放背景音乐异常。

事实上，在场景过渡过程中不停止播放背景音乐也是一个很好的解决问题的方案，当在下一个场景播放新的背景音乐后，前一个场景的背景音乐播放自然就会停止，因为背景音乐不能同时播放多个。

10.3 实例：设置背景音乐与音效

为了进一步介绍背景音乐和音效播放 API 的使用，下面通过一个实例给大家介绍一下。图 10-2 有两个场景：GameScene 和 SettingScene。在 GameScene 场景单击“游戏设置”菜单可以切换到 SettingScene 场景，在 SettingScene 场景中可以设置是否播放背景音乐和音效，设置完成后单击 OK 菜单可以返回到 GameScene 场景。

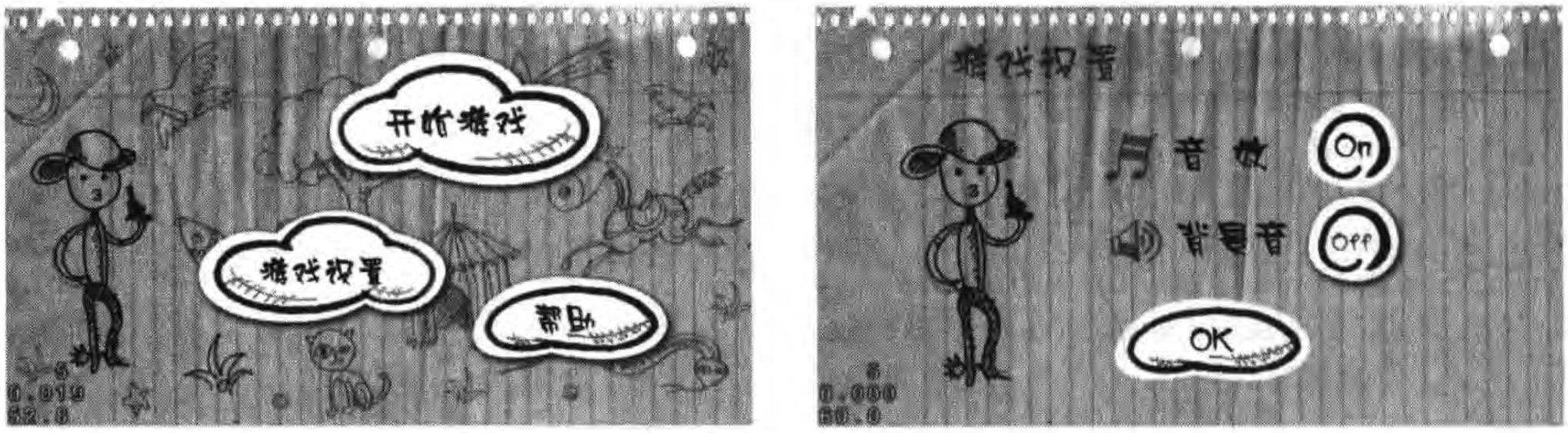


图 10-2 设置背景音乐与音效

（左图 GameScene 场景、右图 SettingScene 场景）

10.3.1 GameScene 场景实现

GameScene 场景是游戏中的主菜单场景。GameScene.lua 文件代码如下：

```
require "Cocos2d"
require "Cocos2dConstants"
require "AudioEngine"                                                    ①

EFFECT_FILE = "sound/Blip.wav"                                           ②
MUSIC_FILE = "sound/Jazz.mp3"                                            ③
isEffect = true                                                          ④
size = cc.Director:getInstance():getWinSize()

...

-- create layer
function GameScene:createLayer()
    cclog("GameScene init")
    local layer = cc.Layer:create()

    local bg = cc.Sprite:create("background.png")
    bg:setPosition(cc.p(size.width/2,
        size.height/2))
    layer:addChild(bg)

    -- 开始精灵
    local startlocalNormal = cc.Sprite:create("start-up.png")
```

```
local startlocalSelected = cc.Sprite:create("start-down.png")
local startMenuItem = cc.MenuItemSprite:create(startlocalNormal, startlocalSelected)
startMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(700, 170)))
local function menuItemStartCallback(sender)
    cclog("Touch Start.")
    if isEffect then
        AudioEngine.playEffect(EFFECT_FILE)                                    ⑤
    end
end
startMenuItem:registerScriptTapHandler(menuItemStartCallback)

-- 设置图片菜单
local settingMenuItem = cc.MenuItemImage:create(
    "setting-up.png",
    "setting-down.png")
settingMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(480, 400)))
local function menuItemSettingCallback(sender)
    cclog("Touch Setting.")
    local scene = require("SettingScene")
    local settingScene = scene.create()
    local ts = cc.TransitionJumpZoom:create(1, settingScene)
    cc.Director:getInstance():pushScene(ts)

    if isEffect then
        AudioEngine.playEffect(EFFECT_FILE)
    end

end
settingMenuItem:registerScriptTapHandler(menuItemSettingCallback)

-- 帮助图片菜单
local helpMenuItem = cc.MenuItemImage:create(
    "help-up.png",
    "help-down.png")

helpMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(860, 480)))
local function menuItemHelpCallback(sender)
    cclog("Touch Help.")

    if isEffect then
        AudioEngine.playEffect(EFFECT_FILE)
    end
end
    helpMenuItem:registerScriptTapHandler(menuItemHelpCallback)

    local mn = cc.Menu:create(startMenuItem, settingMenuItem, helpMenuItem)
    mn:setPosition(cc.p(0, 0))
    layer:addChild(mn)

    return layer
end

function GameScene:onEnter()
    cclog("GameScene onEnter")
```

```
end

function GameScene:onEnterTransitionFinish()
    cclog("GameScene onEnterTransitionFinish")
    AudioEngine.playMusic(MUSIC_FILE, true)                    ⑥
end

function GameScene:onExit()
    cclog("GameScene onExit")
end

function GameScene:onExitTransitionStart()
    cclog("GameScene onExitTransitionStart")
    AudioEngine.stopMusic()                                    ⑦
end

function GameScene:cleanup()
    cclog("GameScene cleanup")
    -- AudioEngine.stopMusic()
end

return GameScene
```

上述第①行代码 require "AudioEngine"是引入 AudioEngine.lua 文件。第②行代码是声明全局变量 EFFECT_FILE,该变量保存了音效文件。第③行代码是声明全局变量 MUSIC_FILE,该变量保存了背景音乐文件。第④行代码是声明全局变量 isEffect,该变量保存了音效播放状态。

第⑤行代码 AudioEngine.playEffect(EFFECT_FILE)是播放音效。第⑥行代码 AudioEngine.playMusic(MUSIC_FILE, true))是在 enterTransitionFinish 事件中开始持续播放背景音乐。第⑦行代码 AudioEngine.stopMusic()是在 exitTransitionStart 事件中停止播放背景音乐。

10.3.2 SettingScene 场景实现

设置场景(SettingScene)的 SettingScene.lua 文件代码如下:

```
require "Cocos2d"
require "Cocos2dConstants"
require "AudioEngine"

local MUSIC_FILE = "sound/Synth.mp3"                           ①

...

function SettingScene:ctor()
    cclog("SettingScene init")
    -- 场景节点事件处理
    local function onNodeEvent(event)
        if event == "enter" then
            self:onEnter()
```

```
    elseif event == "enterTransitionFinish" then
        self:onEnterTransitionFinish()
    elseif event == "exit" then
        self:onExit()
    elseif event == "exitTransitionStart" then
        self:onExitTransitionStart()
    elseif event == "cleanup" then
        self:cleanup()
    end
end

    self:registerScriptHandler(onNodeEvent)
end

function SettingScene:createLayer()

    local layer = cc.Layer:create()

    local bg = cc.Sprite:create("setting-back.png")
    bg:setPosition(cc.p(size.width/2,
        size.height/2))
    layer:addChild(bg)

    -- 音效
    local soundOnMenuItem = cc.MenuItemImage:create("on.png", "on.png")
    local soundOffMenuItem = cc.MenuItemImage:create("off.png", "off.png")
    local soundToggleMenuItem = cc.MenuItemToggle:create(soundOffMenuItem,
                                    soundOnMenuItem )
soundToggleMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(818, 220)))
local function menuSoundToggleCallback(sender)                              ②
    cclog("Sound Toggle.")
    if isEffect then
        AudioEngine.playEffect(EFFECT_FILE)                                 ③
    end
    if soundToggleMenuItem:getSelectedIndex() == 1 then -- 选中状态 Off -> On   ④
        isEffect = false
    else
        isEffect = true
    end

end
soundToggleMenuItem:registerScriptTapHandler(menuSoundToggleCallback)

    -- 音乐
local musicOnMenuItem = cc.MenuItemImage:create("on.png", "on.png")
local musicOffMenuItem = cc.MenuItemImage:create("off.png", "off.png")
local musicToggleMenuItem = cc.MenuItemToggle:create(musicOffMenuItem, musicOnMenuItem)
musicToggleMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(818, 362)))
local function menuMusicToggleCallback(sender)                              ⑤
    cclog("Music Toggle.")
    if isEffect then
        AudioEngine.playEffect(EFFECT_FILE)
    end
    if musicToggleMenuItem:getSelectedIndex() == 1 then -- 选中状态 Off -> On   ⑥
```

```
            AudioEngine.stopMusic()
        else
            AudioEngine.playMusic(MUSIC_FILE, true)
        end
    end
    musicToggleMenuItem:registerScriptTapHandler(menuMusicToggleCallback)

    -- Ok 按钮
    local okMenuItem = cc.MenuItemImage:create(
        "ok - down.png",
        "ok - up.png")
    okMenuItem:setPosition(cc.Director:getInstance():convertToGL(cc.p(600, 510)))
    local function menuOkCallback(sender)
        cclog("Ok Menu tap.")
        cc.Director:getInstance():popScene()
        if isEffect then
            AudioEngine.playEffect(EFFECT_FILE)
        end
    end
    okMenuItem:registerScriptTapHandler(menuOkCallback)

    local mn = cc.Menu:create(soundToggleMenuItem, musicToggleMenuItem,okMenuItem)
    mn:setPosition(cc.p(0, 0))
    layer:addChild(mn)
    return layer
end

function SettingScene:onEnter()
    cclog("SettingScene onEnter")
end

function SettingScene:onEnterTransitionFinish()
    cclog("SettingScene onEnterTransitionFinish")
    AudioEngine.playMusic(MUSIC_FILE, true)                                    ⑦
end

function SettingScene:onExit()
    cclog("SettingScene onExit")
end

function SettingScene:onExitTransitionStart()
    cclog("SettingScene onExitTransitionStart")
end

function SettingScene:cleanup()
    cclog("SettingScene cleanup")
End
```

上述第①行代码是声明模块变量 MUSIC_FILE,该变量保存了音效文件。第②行代码 menuSoundToggleCallback 是用户单击音效开关按钮时回调函数。第③行代码 AudioEngine.playEffect(EFFECT_FILE)是播放音效。第④行代码判断开关菜单状态是否从 Off→On,如果是则将开关变量 isEffect 设置为 false,否则为 true。

第⑤行代码 menuMusicToggleCallback 是用户单击播放背景音乐开关按钮时回调函数。第⑥行代码判断开关菜单状态是否从 Off→On，如果是则停止音乐，否则播放音乐。

第⑦行代码 AudioEngine. playMusic(MUSIC_FILE, true))是在 enterTransitionFinish 事件中开始持续播放背景音乐。

本章小结

本章介绍了 Cocos2d-x Lua API 引擎在不同平台所支持的音频文件格式，还介绍了 Cocos2d-x Lua API 中音频引擎 Audio Engine。

第11章

粒子系统

人们经常会在游戏中看到火和爆炸等动画效果，怎样才能制作得如此逼真、而且又不需要消耗大量的内存？可以使用“粒子系统”来创建这些动画效果。“粒子系统”是本章要介绍的主要内容。

11.1 问题的提出

如果游戏场景中有一堆篝火（见图 11-1），篝火一直不停地燃烧，如何实现呢？通过之前学习过的内容，应首先考虑到使用帧动画。

图 11-1 帧动画

首先需要准备多张帧图片，然后编写如下代码实现：

```
local size = cc.Director:getInstance():getWinSize()

function GameScene:createLayer()

    local layer = cc.Layer:create()
    --////////////////动画开始//////////////////////
    local animation = cc.Animation:create()
    for i = 1,17 do
        local frameName = string.format("fire/campFire%02d.png",i)
```

```
    cclog("frameName =  %s",frameName)
    animation:addSpriteFrameWithFile(frameName)
end
animation:setDelayPerUnit(0.11)                    --设置两个帧播放时间
animation:setRestoreOriginalFrame(true)            --动画执行后还原初始状态

local sprite = cc.Sprite:create("fire/campFire01.png")

sprite:setPosition(cc.p(size.width/2 , size.height/2))
layer:addChild(sprite)

    local action = cc.Animate:create(animation)
    sprite:runAction(cc.RepeatForever:create(action))
    --//////////////////动画结束//////////////////////
    return layer
end

return GameScene
```

这个示例运行一下看看动画效果，如果每一帧能够设计得很好，那么运行的效果也会不错。但是另外的问题出现了，那就是性能，需要将大量的帧图片渲染到屏幕上，而且每一帧图片很大，这样程序会消耗大量的内存，有可能导致内存的溢出。

因此，这种帧动画方案解决这种火焰效果不是很理想。事实上粒子系统是解决这类问题的最佳方案。

11.2　粒子系统基本概念

所谓“粒子系统”是模拟自然界中的一些粒子的物理运动的效果，如烟雾、下雪、下雨、火、爆炸等。单个或几个粒子无法体现出粒子运动规律性，必须有大量的粒子才能体现运行的规律。而且大量的粒子不断消失，又有大量的粒子不断产生。微观上粒子运动是随机的、不确定的，而宏观上是有规律的，它们符合物理学中的“测不准原理”。

11.2.1　实例：打火机

下面通过一个最简单的实例来初步了解一下粒子系统。图 11-2 所示是一个 Zippo 打火机，它的火苗就是粒子系统。

图 11-2　火焰粒子系统

只需下面的几行代码就可以实现这个实例。

```
function GameScene:createLayer()

    local layer = cc.Layer:create()
```

```
    local bg = cc.Sprite:create("zippo.png")

    bg:setPosition(cc.p(size.width/2, size.height /2))
    layer:addChild(bg)

    local particleSystem = cc.ParticleFire:create()                                    ①
    particleSystem:setPosition(cc.Director:getInstance():convertToGL(cc.p(270, 380)))  ②
    layer:addChild(particleSystem)                                                     ③

    return layer
end
```

上述第①行代码是创建火焰粒子系统对象，ParticleSystem 是粒子系统基类，子类 ParticleFire 是火焰粒子系统类，图 11-3 是粒子系统类图，从类图中可见粒子系统也是派生子节点类 Node。ParticleSystemQuad 类是 ParticleSystem 类的直接子类，ParticleSystemQuad 描述的粒子系统的每个粒子由 4 个点组成，这种粒子系统是针对于支持浮点计算的 CPU 而设计

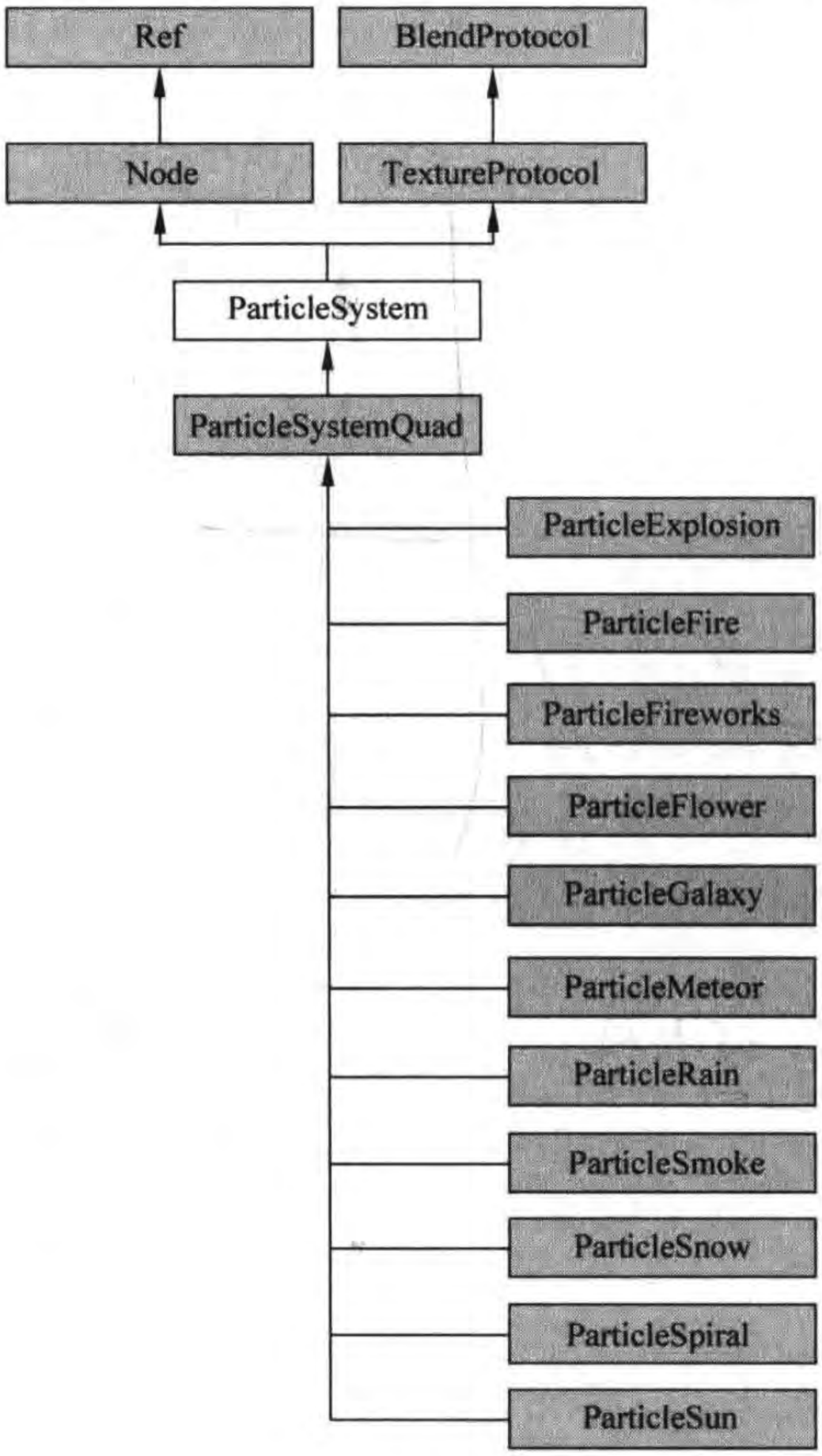


图 11-3　粒子系统类图

的，如 ARM v7 等，从图 11-3 中可见 ParticleFire 等 11 种粒子系统全部是 ParticleSystemQuad 粒子系统。

第②行代码是设置粒子系统的位置。第③行代码是添加火焰粒子系统对象到当前层。

11.2.2　粒子发射模式

粒子系统发射时有两种模式：重力模式和半径模式。重力模式是让粒子围绕一个中心点做远离或紧接运动(见图 11-4(a))，半径模式是让粒子围绕中心点旋转(见图 11-4(b))。

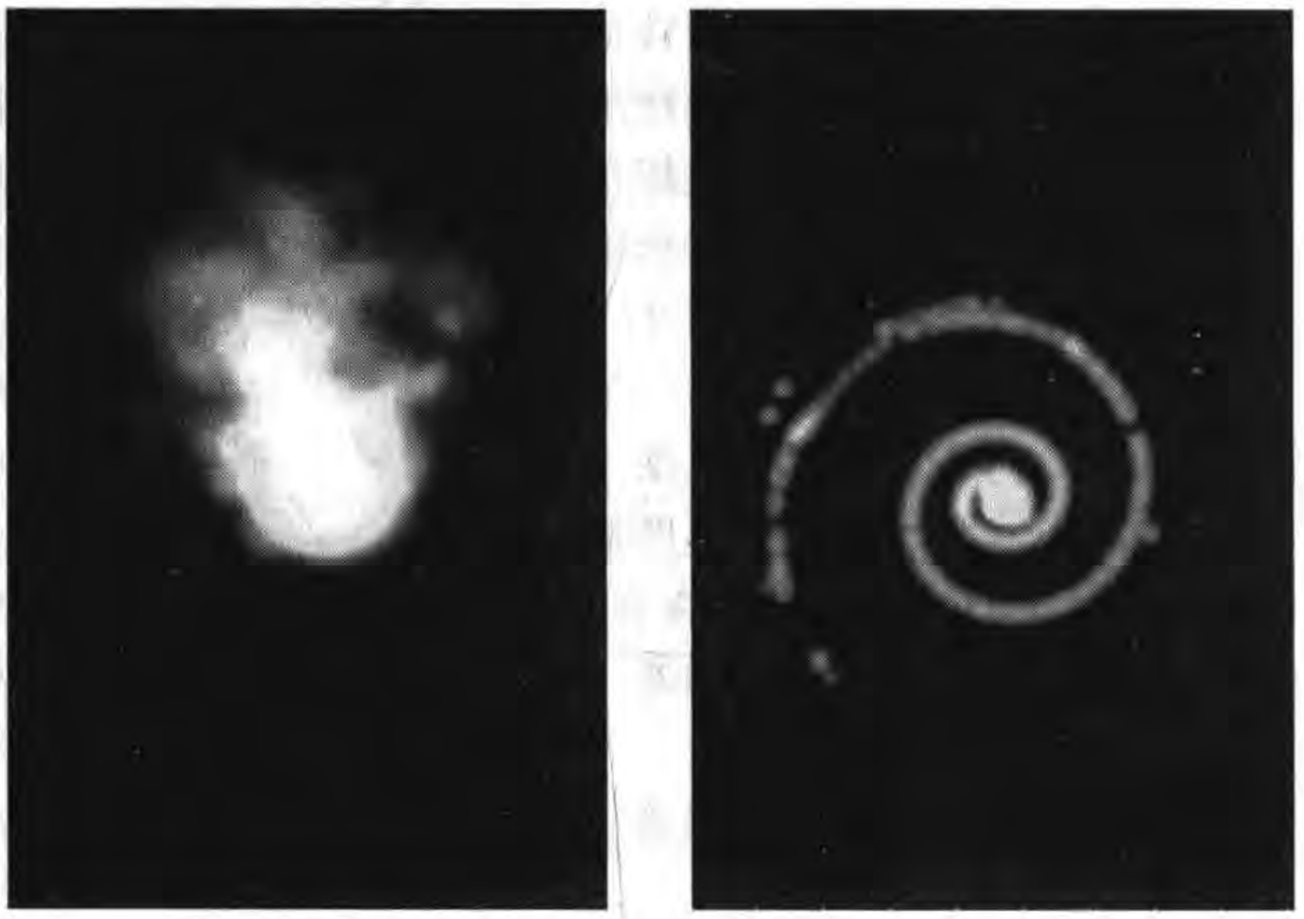


图 11-4　粒子发射模式

11.2.3　粒子系统属性

可能有人会发现上一节介绍的粒子系统实例很简单，事实上并非如此，如果需要调整样式，那就会变得非常麻烦。粒子系统的属性很多，粒子的各种行为都是通过属性控制的，一些属性还与发射模式有关系。表 11-1 所示是粒子系统的属性。

表 11-1　粒子系统属性

属 性 名	行　为	模　式
Duration	粒子持续时间，-1 是永远持续	重力和半径模式
SourcePosition	粒子初始化位置	重力和半径模式
PosVar	粒子初始化位置偏差	重力和半径模式
Angle	粒子方向	重力和半径模式
AngleVar	粒子方向角度偏差	重力和半径模式
StartSize	粒子初始化大小	重力和半径模式
StartSizeVar	粒子初始化大小偏差	重力和半径模式
EndSize	粒子最后大小	重力和半径模式
EndSizeVar	粒子最后大小偏差	重力和半径模式

续表

属 性 名	行 为	模 式
Life	粒子生命期单位秒	重力和半径模式
LifeVar	粒子生命期偏差	重力和半径模式
StartColor	粒子开始颜色	重力和半径模式
StartColorVar	粒子开始颜色偏差	重力和半径模式
EndColor	粒子的结束颜色	重力和半径模式
EndColorVar	粒子的结束颜色偏差	重力和半径模式
StartSpin	粒子的开始旋转角度	重力和半径模式
StartSpinVar	粒子的开始旋转角度偏差	重力和半径模式
EndSpin	粒子的结束旋转角度	重力和半径模式
EndSpinVar	粒子的结束旋转角度偏差	重力和半径模式
Texture	粒子的纹理图片	重力和半径模式
Gravity	粒子的重力	重力模式
Speed	粒子移动的速度	重力模式
SpeedVar	粒子移动的速度偏差	重力模式
TangentialAccel	切向(飞行垂直方向)加速度	重力模式
TangentialAccelVar	切向加速度偏差	重力模式
RadialAccel	径向加速度	重力模式
RadialAccelVar	径向加速度偏差	重力模式
StartRadius	开始半径	半径模式
StartRadiusVar	开始半径偏差	半径模式
EndRadius	结束半径	半径模式
EndRadiusVar	结束半径偏差	半径模式
RotatePerSecond	每秒钟粒子旋转的角度	半径模式
RotatePerSecondVar	每秒钟粒子旋转的角度偏差	半径模式

这么多粒子属性确实很难全部记下来,从这个角度看粒子系统又是比较复杂的。事实上读者不需要将所有的属性都记下来,只需要记住其中常用的。而且很多属性之间是有规律的,它们是成对出现的(xxx 和 xxxVar),例如 startRadius 和 startRadiusVar,后面的 Var 表示 Variance(偏差),即浮动值,表示随机上下浮动的修正值,实际值由原始值(startRadius)+浮动值(startRadiusVar)组成,例如 startRadius=50,startRadiusVar=10,那么随机出来的结果就是 40～60。

Cocos2d-x Lua 在粒子系统基类 ParticleSystem 中定义了这些粒子属性。如果想让 11.2.1 节 Zippo 打火机火焰大一点,可以调整这些属性。修改程序代码如下:

```
function GameScene:createLayer()

    local layer = cc.Layer:create()
    local bg = cc.Sprite:create("zippo.png")

    bg:setPosition(cc.p(size.width/2, size.height /2))
```

```
layer:addChild(bg)

local particleSystem = cc.ParticleFire:create()

--设置粒子的重力
particleSystem:setGravity(cc.p(45, 300))                                    ①
--设置径向加速度
particleSystem:setRadialAccel(58)
--设置粒子初始化大小
particleSystem:setStartSize(84)
--设置粒子初始化大小偏差
particleSystem:setStartSizeVar(73)
--设置粒子最后大小偏差
particleSystem:setEndSize(123)
--设置粒子最后大小偏差
particleSystem:setEndSizeVar(17)
--设置粒子切向加速度
particleSystem:setTangentialAccel(70)
--设置粒子切向加速度偏差
particleSystem:setTangentialAccelVar(47)
--设置粒子生命期
particleSystem:setLife(0.79)
--设置粒子生命期偏差
particleSystem:setLifeVar(0.45)                                             ②

particleSystem:setPosition(cc.Director:getInstance():convertToGL(cc.p(270, 380)))
    layer:addChild(particleSystem)

    return layer
end
```

上述第①和第②行代码设置了粒子系统对象的属性。

11.3　内置粒子系统

从图 10-3 所示的类图中可以看到，Cocos2d-x 中有内置的 11 种粒子，这些粒子的属性都是预先定义好的，读者也可以在程序代码中单独修改某些属性，在上一节的实例中已经实现了这些属性的设置。

11.3.1　内置粒子系统

内置的 11 种粒子系统说明如下：

(1) ParticleExplosion。爆炸粒子效果，属于半径模式。

(2) ParticleFire。火焰粒子效果，属于重力模式。

(3) ParticleFireworks。烟花粒子效果，属于重力模式。

(4) ParticleFlower。花粒子效果，属于重力模式。

(5) ParticleGalaxy。星系粒子效果，属于半径模式。

(6) ParticleMeteor。流星粒子效果,属于重力模式。

(7) ParticleSpiral。漩涡粒子效果,属于半径模式。

(8) ParticleSnow。雪粒子效果,属于重力模式。

(9) ParticleSmoke。烟粒子效果,属于重力模式。

(10) ParticleSun。太阳粒子效果,属于重力模式。

(11) ParticleRain。雨粒子效果,属于重力模式。

这11种粒子系统每一个都有如下两个create函数,通过create函数可以创建粒子对象。

```
cc.ParticleExplosion:create ()
cc.ParticleSystemQuad:createWithTotalParticles(numberOfParticles)
```

其中createWithTotalParticles函数中的numberOfParticles是粒子的初始化个数。

这11种粒子的属性,根据它的发射模式不同也会有所不同,具体情况可以参考表11-1。

11.3.2 实例: 内置粒子系统

下面通过一个实例演示这11种内置粒子系统,如图11-5所示。图(a)是一个操作菜单场景,选择菜单可以进入到动作场景,在图(b)动作场景中演示选择的粒子系统效果,单击右下角返回按钮可以返回到菜单场景。

(a) (b)

图11-5 内置粒子系统实例

下面再看看具体的程序代码,GameScene.lua文件的代码如下:

```
-- 定义常量
kExplosion   = 1                                                    ①
kFire        = 2
...

kSpiral      = 10
kSun         = 11                                                   ②

--操作标识
```

```
actionFlag =- 1                                                    ③

size = cc.Director:getInstance():getWinSize()

...

-- create layer
function GameScene:createLayer()

    local layer = cc.Layer:create()
    local function OnClickMenu(tag, menuItemSender)
        cclog("tag =  %d", tag)
        actionFlag = menuItemSender:getTag()

        local scene = require("MyActionScene")
        local nextScene = scene.create()
        local ts = cc.TransitionSlideInR:create(1, nextScene)
        cc.Director:getInstance():pushScene(ts)

    end

    local pItmLabel1 = cc.Label:createWithBMFont("fonts/fnt8.fnt", "Explosion")
    local pItmMenu1 = cc.MenuItemLabel:create(pItmLabel1)
    pItmMenu1:setTag(kExplosion)
    pItmMenu1:registerScriptTapHandler(OnClickMenu)

    ...

    local mn = cc.Menu:create(pItmMenu1,pItmMenu2,pItmMenu3,
        pItmMenu4,pItmMenu5,
        pItmMenu6,pItmMenu7,
        pItmMenu8,pItmMenu9,
        pItmMenu10,pItmMenu11)
    mn:alignItemsInColumns(2, 2, 2, 2, 2, 1)
    layer:addChild(mn)

    return layer
end

return GameScene
```

上述第①和第②行代码是定义 11 个常量，这 11 个常量对应 11 个菜单项。第③行代码的操作标识用来保存用户单击的菜单项 tag 属性，用于下一个场景的判断。

MyActionScene.lua 主要代码如下：

```
function MyActionScene:onEnterTransitionFinish()
    cclog("MyActionScene onEnterTransitionFinish")

    if actionFlag == kExplosion then                               ①
        system = cc.ParticleExplosion:create()
        pLabel:setString("Explosion")
    elseif actionFlag == kFire then
```

```
        system = cc.ParticleFire:create()
        pLabel:setString("Fire")
    elseif actionFlag == kFireworks then
        system = cc.ParticleFireworks:create()
        pLabel:setString("Fireworks")
    elseif actionFlag == kFlower then
        system = cc.ParticleFlower:create()
        pLabel:setString("Flower")
    elseif actionFlag == kGalaxy then
        system = cc.ParticleGalaxy:create()
        pLabel:setString("Galaxy")
    elseif actionFlag == kMeteor then
        system = cc.ParticleMeteor:create()
        pLabel:setString("Meteor")
    elseif actionFlag == kRain then
        system = cc.ParticleRain:create()
        pLabel:setString("Rain")
    elseif actionFlag == kSmoke then
        system = cc.ParticleSmoke:create()
        pLabel:setString("Smoke")
    elseif actionFlag == kSnow then
        system = cc.ParticleSnow:create()
        pLabel:setString("Snow")
    elseif actionFlag == kSpiral then
        system = cc.ParticleSpiral:create()
        pLabel:setString("Spiral")
    elseif actionFlag == kSun then
        system = cc.ParticleSun:create()
        pLabel:setString("Sun")                                          ②
    end

    system:setPosition(cc.p(size.width / 2, size.height / 2))
    self:addChild(system)
end
```

上述代码在 MyActionScene 场景 enterTransitionFinish 事件(回调 onEnterTransitionFinish()函数)中创建粒子系统对象,而不是在 enter 事件中创建粒子系统对象。这是因为 enter 事件触发时场景还没有显示,如果在 enter 事件中创建爆炸等显示一次的粒子系统,等到场景显示时爆炸已经结束了,看不到爆炸的效果。

第①和第②行代码创建了 11 种粒子系统,这里创建粒子系统时都采用了它们的默认属性值。其中 pLabel:setString("xxx")函数是为场景中标签设置内容,这样在进入场景后可以看到粒子系统的名称。

11.4 自定义粒子系统

除了使用 Cocos2d-x Lua 的 11 种内置粒子系统外,还可以通过创建 ParticleSystemQuad 对象,并设置属性实现自定义粒子系统,通过这种方式完全可以实现各种效果的粒子系统。使用 ParticleSystemQuad 自定义粒子系统至少有两种方式可以实现:代码创建和 plist 文

件创建。

11.4.1　代码创建

所谓代码创建就是完全通过代码方式实现，其中所有的属性全部通过程序代码设置。这要求开发人员对于这些属性值非常熟悉，而且这种方式无法预览，只能通过程序运行看效果，因此比较麻烦。

要想实现图 11-6 所示的下雪粒子系统，本节先介绍通过代码创建方式实现。

图 11-6　下雪粒子系统实例

代码创建的下雪粒子系统，主要代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local bg = cc.Sprite:create("background-1.png")

    bg:setPosition(cc.p(size.width/2, size.height /2))
    layer:addChild(bg)

    local particleSystem = cc.ParticleSystemQuad:createWithTotalParticles(200)          ①

    --设置雪花粒子纹理图片
    particleSystem:setTexture(cc.Director:getInstance()
                                        :getTextureCache():addImage("snow.png"))        ②
    --设置发射粒子的持续时间-1表示永远持续
    particleSystem:setDuration(-1)
    --设置粒子的重力方向
    particleSystem:setGravity(cc.p(0,-240))

    --设置角度以及偏差
    particleSystem:setAngle(90)
    particleSystem:setAngleVar(360)
```

```
    --设置径向加速度以及偏差
    particleSystem:setRadialAccel(50)
    particleSystem:setRadialAccelVar(0)

    --设置粒子的切向加速度以及偏差
    particleSystem:setTangentialAccel(30)
    particleSystem:setTangentialAccelVar(0)

    -- 设置粒子初始化位置偏差
    particleSystem:setPosVar(cc.p(400,0))

    --设置粒子生命期以及偏差
    particleSystem:setLife(4)
    particleSystem:setLifeVar(2)

    --设置粒子开始时的旋转角度以及偏差
    particleSystem:setStartSpin(30)
    particleSystem:setStartSpinVar(60)

    --设置结束时的旋转角度以及偏差
    particleSystem:setEndSpin(60)
    particleSystem:setEndSpinVar(60)

    --设置开始时的颜色以及偏差
    particleSystem:setStartColor(cc.c4b(1,1,1,1))
    --设置结束时的颜色以及偏差
    particleSystem:setEndColor(cc.c4b(1,1,1,1))

    --设置开始时的粒子大小以及偏差
    particleSystem:setStartSize(30)
    particleSystem:setStartSizeVar(0)

    --设置粒子结束时的大小以及偏差
    particleSystem:setEndSize(20.0)
    particleSystem:setEndSizeVar(0)

    --设置每秒钟产生粒子的数量
    particleSystem:setEmissionRate(100)

    particleSystem:setPosition(cc.p(size.width/2, size.height + 50))

    layer:addChild(particleSystem)

    return layer
end
```

上述第①行代码 cc. ParticleSystemQuad: createWithTotalParticles(200)是创建 ParticleSystemQuad 对象,静态 createWithTotalParticles 函数是通过指定初始粒子数来创建粒子对象。

第②行代码是指定粒子的纹理,可以通过指定的纹理图片创建纹理对象 Texture2D,贴图的纹理图片宽高必须是 2 的 n 次幂,大小不要超过 64×64 像素,在美工设计纹理图片时

不用关注太多细节，例如，设计雪花纹理图片的时候，按照雪花是有 6 个角的，很多人会设计成图 11-7 所示的样式，而事实上需要图 11-8 所示的渐变效果的圆点。

图 11-7　雪花图片

图 11-8　雪花粒子纹理图片

11.4.2　plist 文件创建

代码创建方式要维护很多属性，要想手工调整这些属性是非常困难的事情，所以推荐使用 Particle Designer 等粒子设计工具进行所见即所得的设计，这些工具一般会生成一个描述粒子的属性类表文件 plist，然后通过类似下面的语句加载：

```
local particleSystem = cc.ParticleSystemQuad:create("snow.plist")
```

snow.plist 是描述运动的属性文件，plist 文件是一种 XML 文件，代码如下：

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<!DOCTYPE plist PUBLIC " - //Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList -
1.0.dtd">
<plist version = "1.0">
<dict>
    <key>angle</key>
    <real>270</real>
    <key>angleVariance</key>
    <real>5</real>
    <key>blendFuncDestination</key>
    <integer>771</integer>
    <key>blendFuncSource</key>
    <integer>1</integer>
    <key>duration</key>
    <real> - 1</real>
    <key>emitterType</key>
    <real>0.0</real>
    <key>finishColorAlpha</key>
    <real>1</real>
    <key>finishColorBlue</key>
    <real>1</real>
    <key>finishColorGreen</key>
    <real>1</real>
    <key>finishColorRed</key>
    <real>1</real>
    <key>finishColorVarianceAlpha</key>
    <real>0.0</real>
    <key>finishColorVarianceBlue</key>
    <real>0.0</real>
```

```
<key>finishColorVarianceGreen</key>
<real>0.0</real>
<key>finishColorVarianceRed</key>
<real>0.0</real>
<key>finishParticleSize</key>
<real>-1</real>
<key>finishParticleSizeVariance</key>
<real>0.0</real>
<key>gravityx</key>
<real>0.0</real>
<key>gravityy</key>
<real>-10</real>
<key>maxParticles</key>
<real>700</real>
<key>maxRadius</key>
<real>0.0</real>
<key>maxRadiusVariance</key>
<real>0.0</real>
<key>minRadius</key>
<real>0.0</real>
<key>minRadiusVariance</key>
<real>0.0</real>
<key>particleLifespan</key>
<real>3</real>
<key>particleLifespanVariance</key>
<real>1</real>
<key>radialAccelVariance</key>
<real>0.0</real>
<key>radialAcceleration</key>
<real>1</real>
<key>rotatePerSecond</key>
<real>0.0</real>
<key>rotatePerSecondVariance</key>
<real>0.0</real>
<key>rotationEnd</key>
<real>0.0</real>
<key>rotationEndVariance</key>
<real>0.0</real>
<key>rotationStart</key>
<real>0.0</real>
<key>rotationStartVariance</key>
<real>0.0</real>
<key>sourcePositionVariancex</key>
<real>1200</real>
<key>sourcePositionVariancey</key>
<real>0.0</real>
<key>speed</key>
<real>130</real>
<key>speedVariance</key>
<real>30</real>
<key>startColorAlpha</key>
<real>1</real>
<key>startColorBlue</key>
<real>1</real>
```

```
    <key>startColorGreen</key>
    <real>1</real>
    <key>startColorRed</key>
    <real>1</real>
    <key>startColorVarianceAlpha</key>
    <real>0.0</real>
    <key>startColorVarianceBlue</key>
    <real>0.0</real>
    <key>startColorVarianceGreen</key>
    <real>0.0</real>
    <key>startColorVarianceRed</key>
    <real>0.0</real>
    <key>startParticleSize</key>
    <real>10</real>
    <key>startParticleSizeVariance</key>
    <real>5</real>
    <key>tangentialAccelVariance</key>
    <real>0.0</real>
    <key>tangentialAcceleration</key>
    <real>1</real>
    <key>textureFileName</key>
    <string>snow.png</string>
</dict>
</plist>
```

上述 plist 文件描述的属性和属性值都是成对出现的，其中<key>标签描述的是属性，<real>描述的是属性值。plist 文件是描述粒子的属性，使用时还需要有粒子纹理图片，plist 文件中 textureFileName 属性指定了纹理图片，需要将 plist 文件和纹理图片放置到 Resources 目录下面。

提示　描述粒子属性的 plist 文件，可以通过粒子系统设计工具生成，有关粒子系统工具的使用可以参考本系列丛书的工具卷(《Cocos2d-x 实战：工具卷》)。

图 11-6 所示的下雪实例，使用 plist 文件创建，主要代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local bg = cc.Sprite:create("background-1.png")
    bg:setPosition(cc.p(size.width/2, size.height/2))
    layer:addChild(bg)

    local particleSystem = cc.ParticleSystemQuad:create("snow.plist")
    particleSystem:setPosition(cc.p(size.width/2, size.height + 50))
    layer:addChild(particleSystem)

    return layer
end
```

由代码可见，plist 文件创建粒子系统要比代码创建简单很多，这主要是因为采用了

plist 描述粒子属性。

本章小结

通过对本章的学习,使广大读者熟悉了粒子系统的基本概念。然后还介绍了内置粒子系统和自定义粒子系统。

第 12 章

瓦片地图

在游戏中经常会用到背景，有些背景比较大而且复杂，使用一个大背景图片会消耗大量内存，采用一些地图技术构建的大背景可以解决性能问题。本章介绍瓦片(Tiles)地图，以及如何使用瓦片地图构建复杂的大背景。

12.1 地图性能问题

图 12-1 所示的游戏场景是第 8 章介绍的实例，在场景中有 3 个方块精灵(BoxA、BoxB、BoxC)和背景精灵，如果把这个背景叫做“地图”有点牵强。它采用了有规律的纹理。

那么如何设计这个游戏地图呢？在前面的学习过程中其实已经使用了两种方法：采用一张大图片和采用小纹理图片重复贴图。

1. 采用一张大图片

可以让美术设计师制作一个屏幕大小的图片，大小 960×640 像素，如图 12-2 所示。如果是 RGBA8888 格式，则占用内存大小大约 2400K 字节。

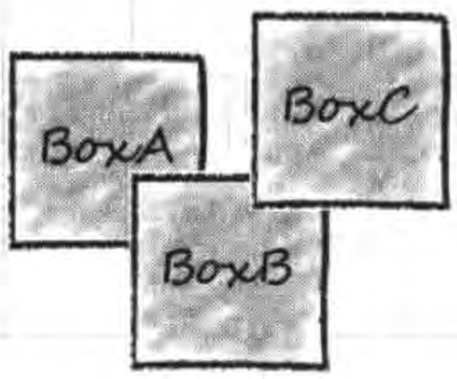


图 12-1 游戏场景

图 12-2 游戏地图

2. 采用小纹理图片重复贴图

在第 8 章介绍的实例采用的就是小纹理图片重复贴图，每个小的纹理图片大小是 128×128 像素，如图 12-3 所示。如果是 RGBA8888 格式，则占用内存大小大约 64K 字节，纹理图

片的宽和高必须是 2 的 n 次幂。

采用小纹理图片重复贴图的主要代码如下：

```
--贴图的纹理图片宽高必须是 2 的 n 次幂,128 × 128
local bg = cc.Sprite:create("BackgroundTile.png", cc.rect
(0, 0, size.width, size.height))
--贴图的纹理参数,水平重复平铺,垂直重复平铺
bg:getTexture():setTexParameters(gl.LINEAR, gl.LINEAR, gl.
REPEAT, gl.REPEAT)
bg:setPosition(cc.p(size.width/2, size.height/2))
layer:addChild(bg, 0)
```

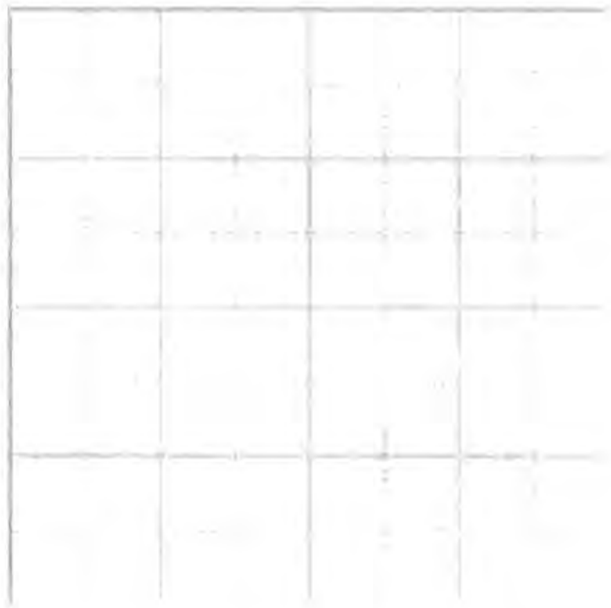


图 12-3　小纹理图片

地图比较大的情况下，在内存很少的移动设备上就会造成内存泄漏。因此需要一些小图片经过拼接构成大地图，这样可以减少内存占用。

但是使用 Texture2D 只能重复贴图，构建的地图比较简单，而且有规律，无法构建如图 12-4 所示的复杂地图，这种复杂地图可以使用瓦片地图。

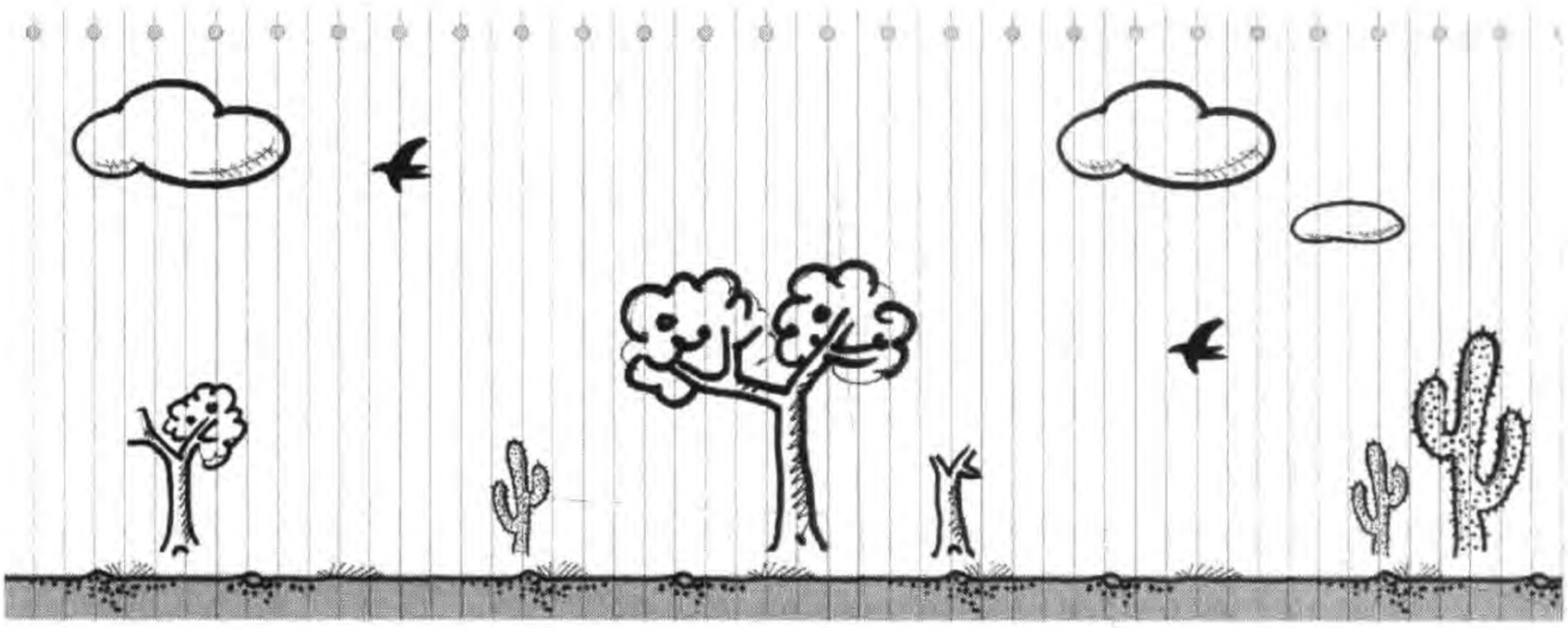


图 12-4　复杂地图

12.2　瓦片地图 API

Cocos2d-x 中访问瓦片地图 Lua API 的主要类有 TMXTiledMap、TMXLayer 和 TMXObjectGroup 等。

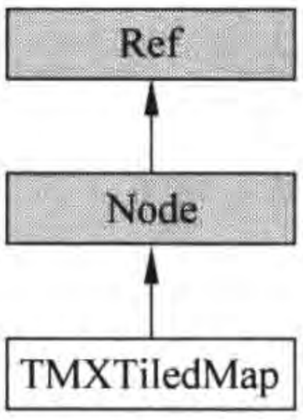


图 12-5　TMXTiledMap 类图

1. TMXTiledMap

TMXTiledMap 是瓦片地图类，它的类图如图 12-5 所示，TMXTiledMap 派生自 Node 类，具有 Node 类的特点。

TMXTiledMap 常用的函数如下：

(1) getLayerlayerName()。通过层名获得 TMXLayer 层对象。

(2) getObjectGroup(groupName)。通过对象层名获得

层中 TMXObjectGroup 对象组集合。

(3) getMapSize()。获得地图的尺寸,它的单位是瓦片。

(4) getTileSize()。获得瓦片尺寸,它的单位是像素。

示例代码如下:

```
local group = _tileMap:getObjectGroup("Objects")
localbackground = _tileMap:getLayer("Background")
```

其中_tileMap 是瓦片地图对象。

2. TMXLayer

TMXLayer 是地图层类,它的类图如图 12-6 所示,TMXLayer 也派生自 Node 类,也具有 Node 类的特点。同时 TMXLayer 也派生自 SpriteBatchNode 类,所以 TMXLayer 对象具有批量渲染的能力,瓦片地图层就是由大量重复的图片构成的,它们需要渲染提高性能。

TMXLayer 常用的函数如下:

(1) getLayerName()。获得层名。

(2) getLayerSize()。获得层尺寸,它的单位是瓦片。

(3) getMapTileSize()。获得瓦片尺寸,它的单位是像素。

(4) getPositionAt(tileCoordinate)。通过瓦片坐标获得像素坐标,瓦片坐标 y 轴方向与像素坐标 y 轴方向相反。

(5) getTileGIDAt(tileCoordinate)。通过瓦片坐标获得 GID 值。

3. TMXObjectGroup

TMXObjectGroup 是对象层中的对象组集合,它的类图如图 12-7 所示,注意 TMXObjectGroup 与 TMXLayer 不同,TMXObjectGroup 不是派生自 Node 类,不具有 Node 类的特性。

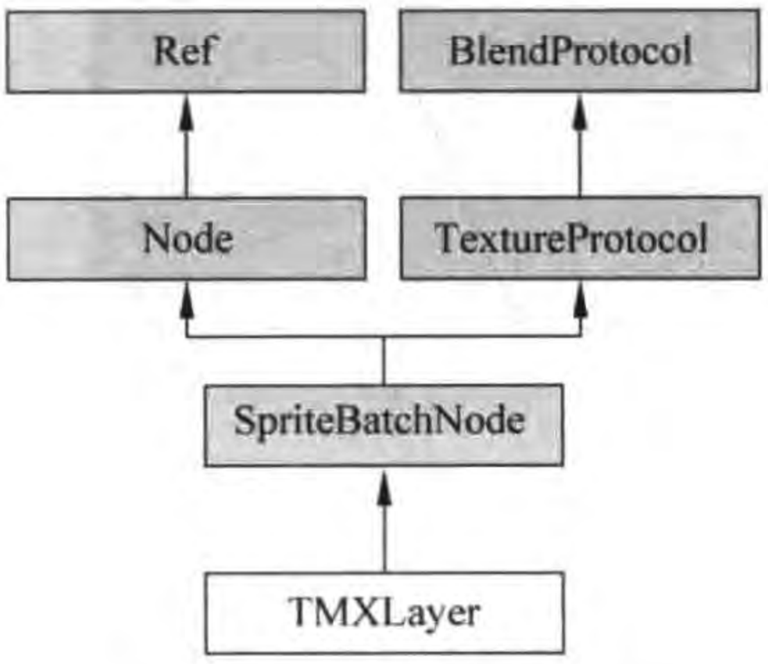


图 12-6　TMXLayer 类图

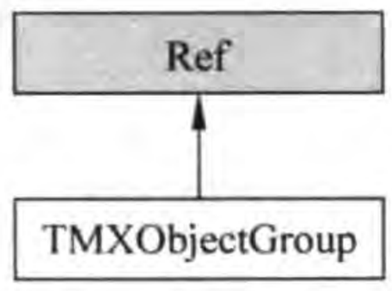


图 12-7　TMXObjectGroup 类图

TMXObjectGroup 常用的函数如下:

(1) getProperty(propertyName)。通过属性名获得属性值。

(2) getObject(objectName)。通过对象名获得对象信息。

(3) getProperties()。获得对象的属性。

(4) getObjects()。获得所有对象。

12.3 实例：忍者无敌

本节再介绍一个完整的实例，使广大读者能够了解开发瓦片地图应用的完整流程。

实例比较简单，如图 12-8 所示，地图上有一个忍者精灵，玩家单击它周围的上、下、左、右，它就能够向这个方向行走。障碍物是无法穿越的，障碍物是除了草地以外的部分，包括树、山、河流等。

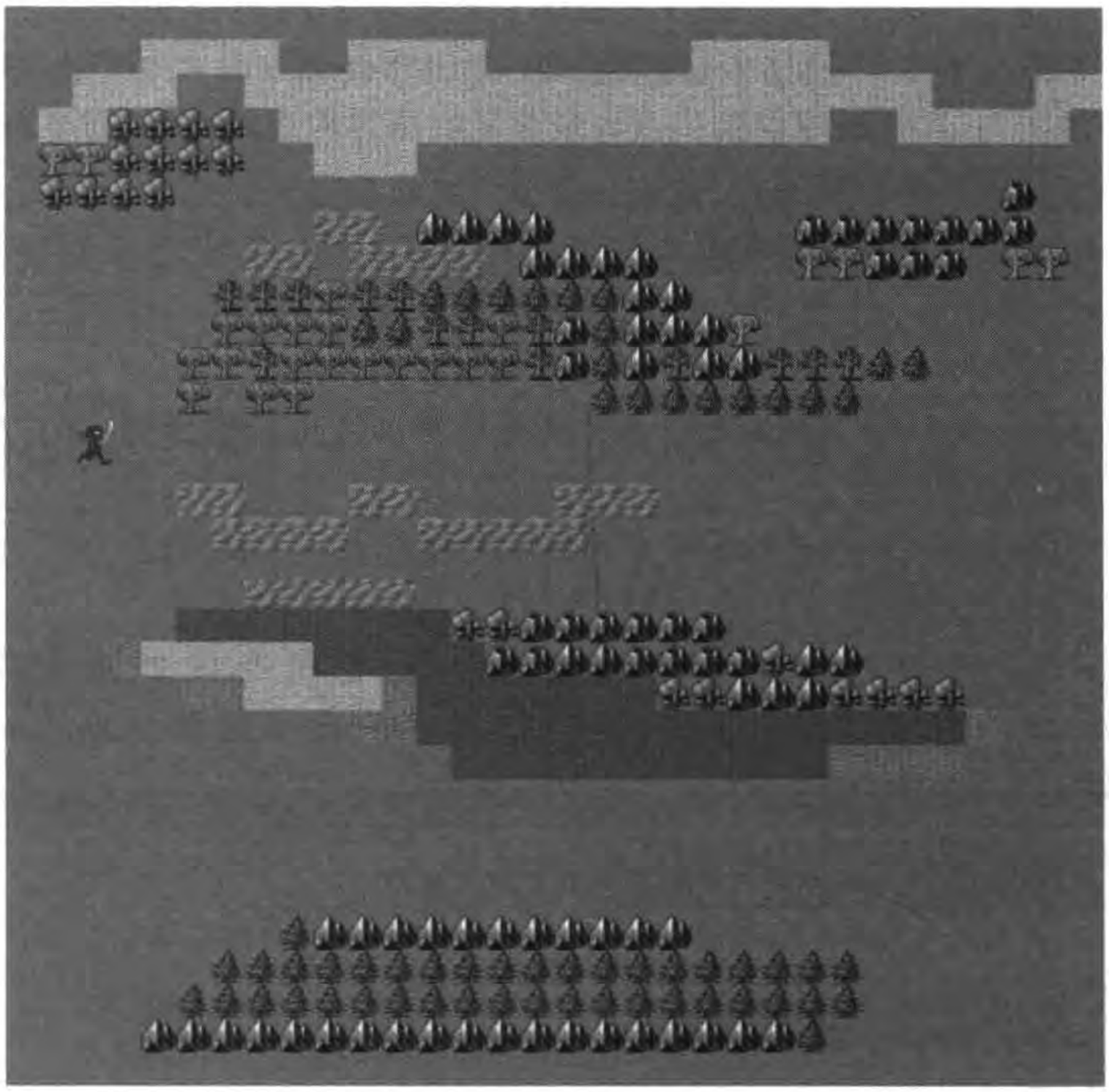


图 12-8 忍者实例地图

12.3.1 设计地图

采用 David Gervais(http://pousse.rapiere.free.fr/tome/index.htm)提供开源免费瓦片集，下载的文件为 dg_grounds32.gif，gif 文件格式会有一定的问题，需要转换为.jpg 或.png 文件。本实例中是使用 Photoshop 转换为 dg_grounds32.jpg。

David Gervais 提供的瓦片集中的瓦片是 32×32 像素，创建的地图大小是 32×32 瓦片。先为地图添加普通层和对象层，普通层按照图 12-9 设计，对象层中添加几个矩形区域对象，制作具体细节请参看智捷 iOS 课堂的另外一本工具书《Cocos2d-x 实战：工具卷》，这

里不再赘述。这个阶段设计完成的结果如图 12-8 所示。保存文件名为 MiddleMap.tmx，保存目录为 Resources\map。

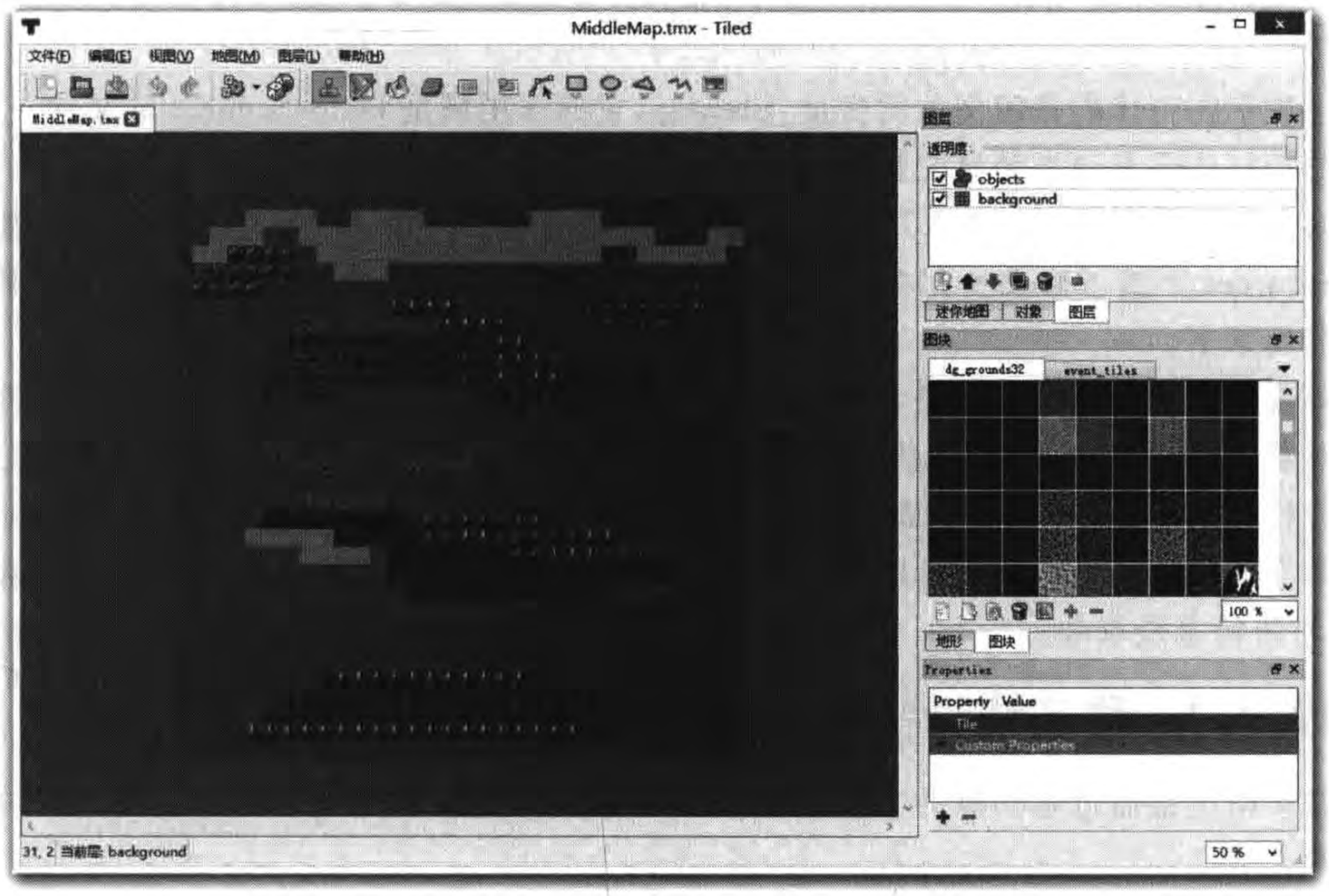


图 12-9　设计地图的普通层

12.3.2　程序中加载地图

地图设计完成就可以在程序中加载地图了。下面再看看具体的程序代码，GameScene.lua 文件的代码如下：

```
local _tileMap                                                    ①
local _player                                                     ②
…
-- create layer
function GameScene:createLayer()

    local layer = cc.Layer:create()

    _tileMap = cc.TMXTiledMap:create("map/MiddleMap.tmx")         ③
    layer:addChild(_tileMap,0,100)                                ④

    local group = _tileMap:getObjectGroup("objects")              ⑤
    local spawnPoint = group:getObject("ninja")                   ⑥

    local x = spawnPoint["x"]                                     ⑦
    local y = spawnPoint["y"]                                     ⑧

    _player = cc.Sprite:create("ninja.png")                       ⑨
```

```
    _player:setPosition(cc.p(x,y))                                    ⑩
    layer:addChild(_player, 2, 200)

    return layer
end
```

上述第①行代码是定义地图变量_tileMap,它的作用域是 GameScene.lua 文件。

第②行代码是定义精灵变量_player,它的作用域与_tileMap 相同。

第③行代码是创建 TMXTiledMap 对象,地图文件是 MiddleMap.tmx,map 是资源目录 Resources 下的子目录。TMXTiledMap 对象也是 Node 对象,需要通过第④行代码添加到当前场景中。

第⑤行代码是通过对象层名 objects 获得层中对象组集合。第⑥行代码是从对象组中通过对象名获得 ninja 对象信息,它的返回值是字典类型,字典类型是一种"键-值"对结构。第⑦行代码从 spawnPoint 按照 x 键取出它的值。类似地,第⑧行代码是获得 y 轴坐标。

第⑨行代码是创建精灵_player。第⑩行代码是设置精灵位置,这个位置是从对象层中 ninja 对象信息获取的。

12.3.3 移动精灵

移动精灵是通过触摸事件实现移动的,需要在层中进行事件处理,应在层中注册单点触摸事件。

下面再看看具体的程序代码,GameScene.lua 文件的代码如下:

```
local _tileMap
local _player
local _layer                                                          ①
...
local function touchBegan(touch, event)
    cclog("touchBegan")
    return true
end

local function touchMoved(touch, event)
    cclog("touchMoved")
end

local function touchEnded(touch, event)
    cclog("touchEnded")
    local touchLocation = touch:getLocation()                         ②
    --转换为当前层的模型坐标系
    touchLocation = _layer:convertToNodeSpace(touchLocation)          ③


    local playerPosX,playerPosY = _player:getPosition()               ④
    local diff = cc.pSub(touchLocation, cc.p(playerPosX,playerPosY))  ⑤

    if math.abs(diff.x) > math.abs(diff.y) then                       ⑥
        if diff.x > 0 then                                            ⑦
```

```
            playerPosX = playerPosX + _tileMap:getTileSize().width
            _player:runAction(cc.FlipX:create(false))                    ⑧
        else
            playerPosX = playerPosX - _tileMap:getTileSize().width
            _player:runAction(cc.FlipX:create(true))                     ⑨
        end
    else
        if diff.y > 0 then                                               ⑩
            playerPosY = playerPosY + _tileMap:getTileSize().height
        else
            playerPosY = playerPosY - _tileMap:getTileSize().height
        end
    end
    cclog("playerPos ( %f , %f) ",playerPosX,playerPosY)
    _player:setPosition(cc.p(playerPosX,playerPosY))

end

-- create layer
function GameScene:createLayer()

    _layer = cc.Layer:create()

    _tileMap = cc.TMXTiledMap:create("map/MiddleMap.tmx")
    _layer:addChild(_tileMap,0,100)

    local group = _tileMap:getObjectGroup("objects")
    local spawnPoint = group:getObject("ninja")

    local x = spawnPoint["x"]
    local y = spawnPoint["y"]

    _player = cc.Sprite:create("ninja.png")
    _player:setPosition(cc.p(x,y))
    _layer:addChild(_player, 2, 200)

    -- 创建一个事件监听器 OneByOne 为单点触摸
    local listener = cc.EventListenerTouchOneByOne:create()
    -- 设置是否吞没事件,在 touchBegan 函数返回 true 时吞没
    listener:setSwallowTouches(true)
    -- EVENT_TOUCH_BEGAN 事件回调函数
    listener:registerScriptHandler(touchBegan,cc.Handler.EVENT_TOUCH_BEGAN )
    -- EVENT_TOUCH_MOVED 事件回调函数
    listener:registerScriptHandler(touchMoved,cc.Handler.EVENT_TOUCH_MOVED )
    -- EVENT_TOUCH_ENDED 事件回调函数
    listener:registerScriptHandler(touchEnded,cc.Handler.EVENT_TOUCH_ENDED )

    local eventDispatcher = self:getEventDispatcher()
    -- 添加监听器
    eventDispatcher:addEventListenerWithSceneGraphPriority(listener, _layer)

    return _layer
end
```

上述第①行代码是定义层变量_layer，_layer是当前GameScene场景中的层，它的作用域是GameScene.lua文件。

第②行代码touch:getLocation()是获得OpenGL坐标，OpenGL坐标的原点是在左下角，touch对象封装了触摸点对象。第③行代码是将触摸点转换相对当前层的模型坐标系。第④行代码local playerPosX,playerPosY = _player:getPosition()是获得精灵的位置。

第⑤行代码是获得触摸点与精灵位置之差。第⑥行代码是比较一下触摸点与精灵位置之差，是y轴之差大还是x轴之差大，哪个差值较大就沿着哪个轴移动。第⑦行代码diff.x>0是沿着x轴正方向移动，否则是沿着x轴负方向移动。第⑧行代码_player:runAction(cc.FlipX:create(false))是把精灵翻转回原始状态。第⑨行代码_player:runAction(cc.FlipX:create(true))是把精灵沿着y轴水平翻转。

第⑩行代码是沿着y轴移动，diff.y>0是沿着y轴正方向移动，否则是沿着y轴负方向移动。

12.3.4 检测碰撞

到目前为止，游戏中的精灵可以穿越任何障碍物。为了能够检测到精灵是否碰撞到障碍物，需要再添加一个普通层(collidable)，它的目的不是显示地图，而是检测碰撞。在检测碰撞层中使用瓦片覆盖background层中的障碍物，如图12-10所示。

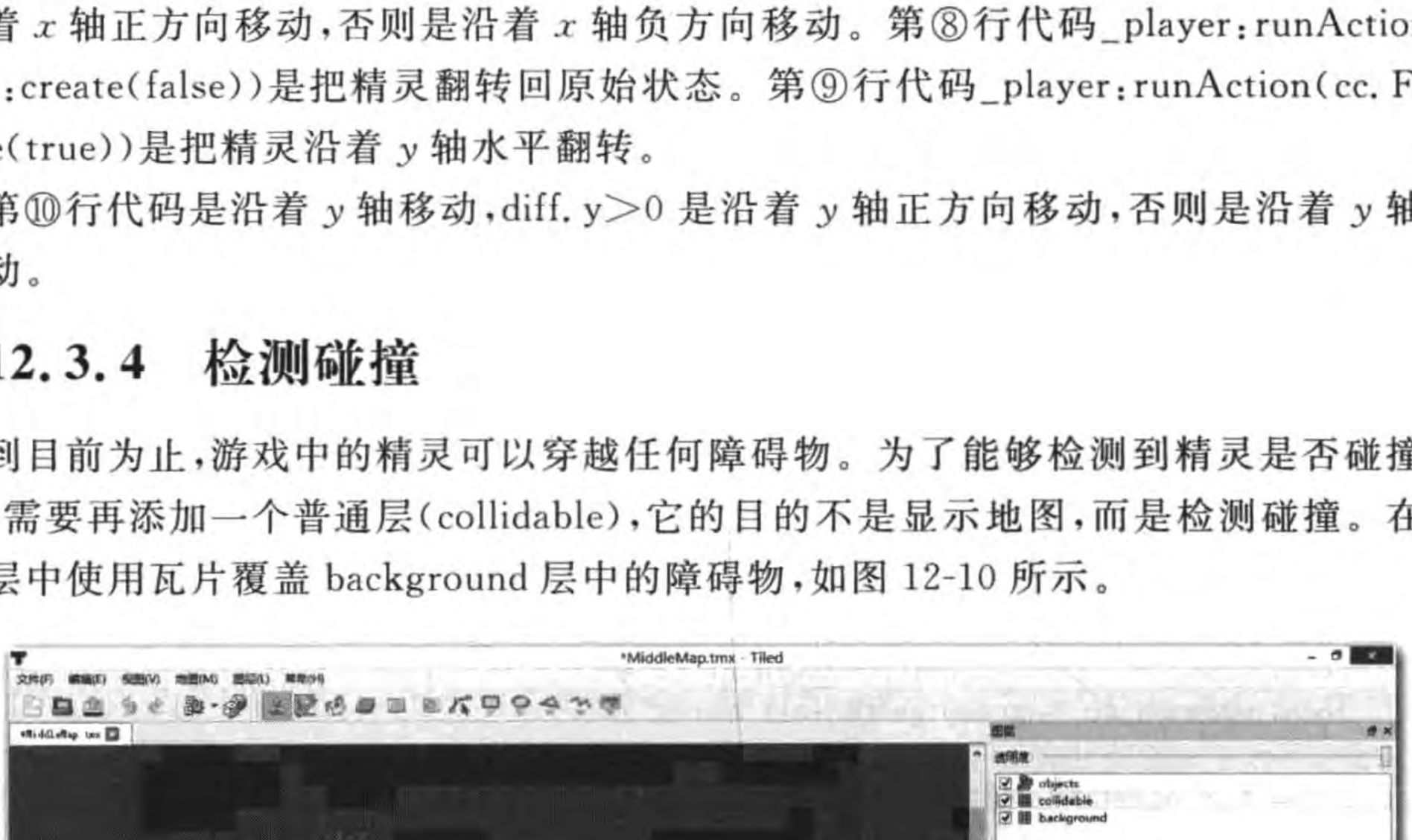


图12-10 检测碰撞层

检测碰撞层中的瓦片集可以是任何的满足格式要求的图片文件。在本例中使用一个32×32像素单色.jpg图片文件collidable_tiles.jpg，它的大小与瓦片一样，也就是说这个瓦片集中只有一个瓦片。导入这个瓦片集到地图后，需要为瓦片添加一个自定义属性，瓦片本

身也有一些属性，例如坐标属性 x 和 y。

要添加的属性名为 Collidable，属性值为 true。添加过程如图 12-11 所示，首先，选择 collidable_tiles 瓦片集中的要设置属性的瓦片。然后单击属性视图中左下角“+”按钮，添加自定义属性，这时会弹出一个对话框，在对话框中输入自定义属性名 Collidable，单击“确定”按钮。这时回到属性视图，Collidable 在属性后面是可以输入内容的，这里输入 true。

图 12-11　添加检测碰撞属性

地图修改完成后，还要修改 GameScene.lua，主要代码如下：

```
local _tileMap
local _player
local _collidable
local _layer

...

local function tileCoordFromPosition(pos)                                    ①
    local x = pos.x / _tileMap:getTileSize().width
    local y = ((_tileMap:getMapSize().height * _tileMap:getTileSize().height) - pos.y)
                                        / _tileMap:getTileSize().height      ②
    return cc.p(x,y)
end

local function setPlayerPosition(position)                                   ③
    --从像素点坐标转化为瓦片坐标
    local tileCoord = tileCoordFromPosition(position)

    --获得瓦片的 GID math.floor 取小于该数的最大整数
    local intX = math.floor(tileCoord.x)
    local intY = math.floor(tileCoord.y)

    local tileGid = _collidable:getTileGIDAt(cc.p(intX, intY))               ④
```

```
    if tileGid > 0 then                                                    ⑤
        local prop = _tileMap:getPropertiesForGID(tileGid)                 ⑥
        local collision = prop["Collidable"]                               ⑦

        if collision == "true" then -- 碰撞检测成功                         ⑧
            cclog("碰撞检测成功")
            AudioEngine.playEffect("empty.wav")
            return
        end
    end
    -- 移动精灵
    _player:setPosition(position)
end
...

local function touchEnded(touch, event)
    cclog("touchEnded")
    ...

    setPlayerPosition(cc.p(playerPosX,playerPosY))                         ⑨

end


-- create layer
function GameScene:createLayer()

    _layer = cc.Layer:create()

      ...

    _collidable = _tileMap:getLayer("collidable")                          ⑩
    _collidable:setVisible(false)                                          ⑪

    return _layer
end

return GameScene
```

上述第①行代码 tileCoordFromPosition 函数是把像素坐标点转换为地图瓦片坐标点。第②行代码是计算 y 轴瓦片坐标(单位是瓦片数),这个计算有点麻烦,瓦片坐标的原点在左上角,而触摸点使用的坐标是 Open GL 坐标,坐标原点在左下角,表达式_tileMap:getMapSize().height * _tileMap:getTileSize().height - pos.y 是反转 y 坐标轴,结果再除以每个瓦片的高度_tileMap:getTileSize().height,就得到 y 轴瓦片坐标了。

第③行代码中 setPlayerPosition 函数是重新设置精灵的位置,在这个函数中可以检测精灵是否与障碍物碰撞。第④行代码_collidable:getTileGIDAt(cc.p(intX, intY)是通过瓦片坐标获得 GID 值。第⑤行代码 tileGid>0 可以判断瓦片是否存在,tileGid == 0 是瓦片不存在的情况。第⑥行代码_tileMap:getPropertiesForGID(tileGid)是通过地图对象的

getPropertiesForGID返回，它的返回值是字典类型。第⑦行代码是取出变量中的Collidable属性。第⑧行代码collision == "true"是碰撞检测成功情况。

第⑨行代码中的setPlayerPosition(cc.p(playerPosX,playerPosY))是在EVENT_TOUCH_ENDED事件触发时调用，它能够重新设置精灵的位置。

第⑩行代码_collidable = _tileMap:getLayer("collidable")是通过层名字collidable创建层。第⑪行代码_collidable:setVisible(false)是设置层隐藏，要么在这里隐藏，要么在地图编辑时将该层透明，如图12-12所示，在层视图中选择层，然后通过滑动上面的透明度滑块来改变层的透明度，在本例中是需要将透明度设置为0，那么隐藏层_collidable:setVisible(false)语句就不再需要了。

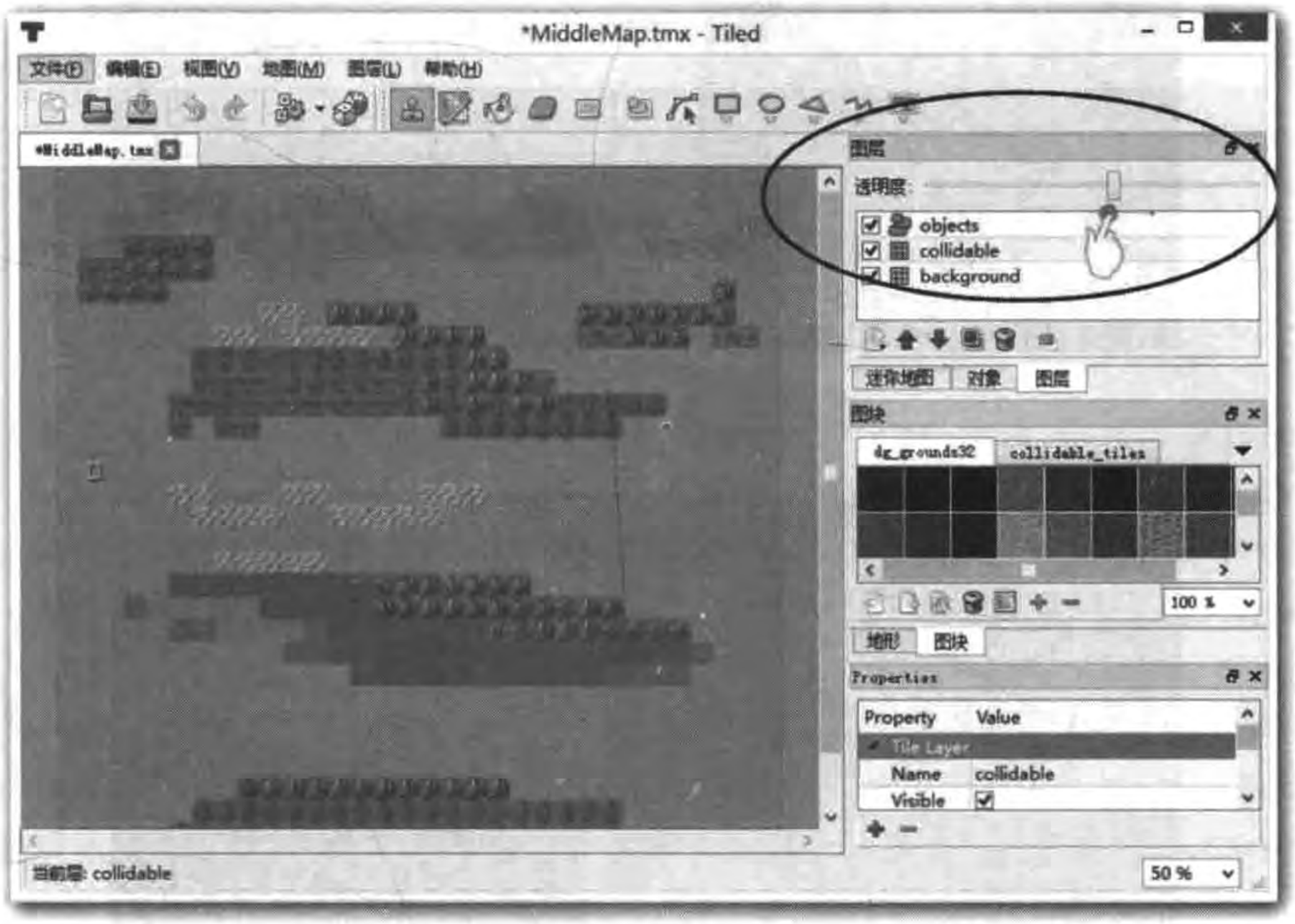


图12-12　设置层透明度

注意　在地图编辑器中，设置层的透明度为0与设置层隐藏，在地图上看起来一样，但是有着本质的区别，设置层透明是无法通过_collidable = _tileMap:getLayer("collidable")语句访问的。

12.3.5　滚动地图

由于地图比屏幕要大，当移动精灵到屏幕的边缘时，那些处于屏幕之外的地图部分应该滚动到屏幕之内。这些需要重新设置视点(屏幕的中心点)，使得精灵一直处于屏幕的中心。

但是精灵太靠近地图边界的时候,有可能不在屏幕的中心。精灵与地图的边界距离的规定是,左右边界距离不小于屏幕宽度的一半,否则会出现如图 12-13 和图 12-14 所示的左右黑边问题。上下边界距离不小于屏幕高度的一半,否则也会出现上下黑边问题。

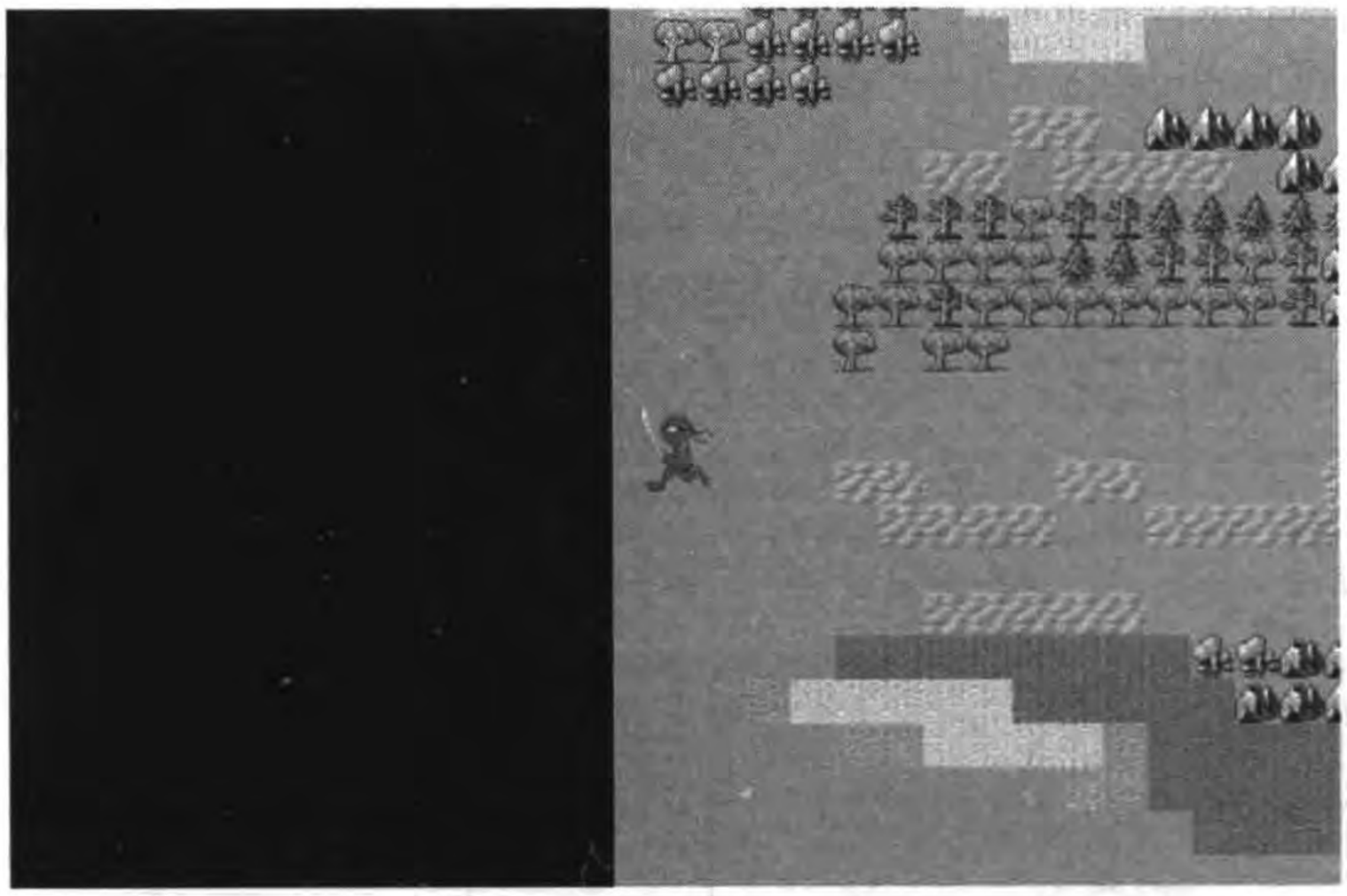


图 12-13　屏幕左边超出地图

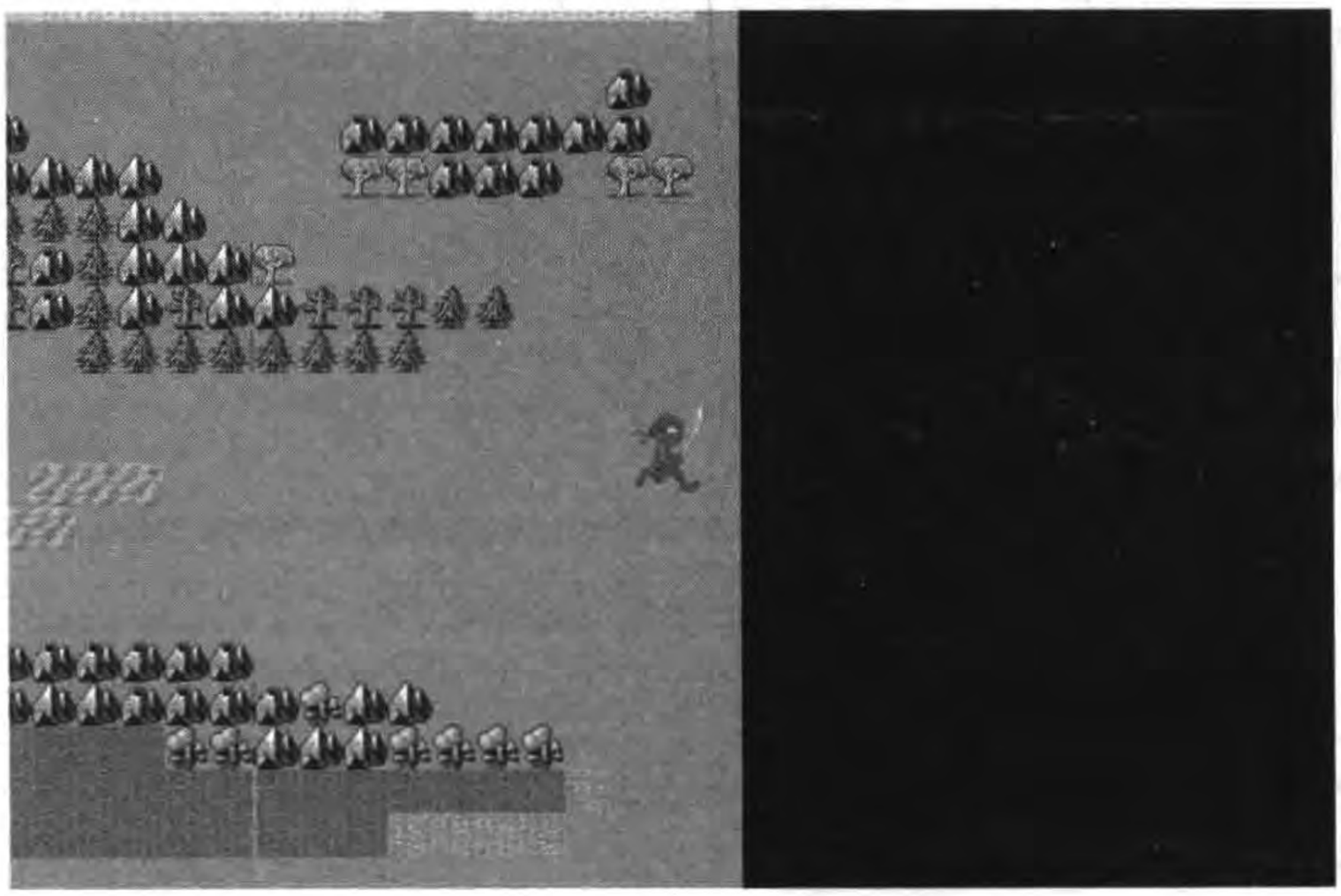


图 12-14　屏幕右边超出地图

重新设置视点的方式很多,本章中采用移动地图位置实现这种效果。

在 GameScene.lua 中再添加一个函数 setViewpointCenter,添加后代码如下:

```
local function setViewpointCenter()

    cclog("setViewpointCenter")
```

```
local playerPosX,playerPosY = _player:getPosition()

--可以防止视图左边超出屏幕之外
local x = math.max(playerPosX, size.width / 2)                                    ①
local y = math.max(playerPosY, size.height / 2)                                   ②
--可以防止视图右边超出屏幕之外
x = math.min(x, (_tileMap:getMapSize().width * _tileMap:getTileSize().width)
        - size.width / 2)                                                         ③
y = math.min(y, (_tileMap:getMapSize().height * _tileMap:getTileSize().height)
        - size.height/2)                                                          ④

    --屏幕中心点
    local pointA = cc.p(size.width/2, size.height/2)                              ⑤
    --使精灵处于屏幕中心,移动地图目标位置
    local pointB = cc.p(x, y)                                                     ⑥
    cclog("目标位置 ( %f , %f) ",pointB.x,pointB.y)

    --地图移动偏移量
    local offset = cc.pSub(pointA, pointB)                                        ⑦
    cclog("offset ( %f , %f) ",offset.x, offset.y)
    _layer:setPosition(offset)                                                    ⑧

End
```

上述第①～④行代码是保证精灵移动到地图边界时不会再移动,防止屏幕超出地图之外,这一点非常重要。其中第①行代码是防止屏幕左边超出地图之外(见图 12-13),math.max(playerPosX, size.width / 2)语句表示当 position.x < size.width / 2 情况下,x 轴坐标始终是 size.width / 2,即精灵不再向左移动。第②行代码与第①行代码类似,不再解释。第③行代码是防止屏幕右边超出地图之外(见图 12-14),math.min(x, (_tileMap:getMapSize().width * _tileMap:getTileSize().width) - size.width / 2)语句表示当 x >(_tileMap:getMapSize().width * _tileMap:getTileSize().width)－size.width/2 时,x 轴坐标是表达式(_tileMap:getMapSize().width * _tileMap:getTileSize().width) - size.width/2 计算的结果。

提示　size 是表示屏幕的大小,_tileMap: getMapSize().width * _tileMap: getTileSize().width)－size.width/2 表达式计算的是地图的宽度减去屏幕宽度的一半。

第④行代码与第③行代码类似,不再解释。

第⑤～⑧行代码实现了移动地图的效果,使得精灵一直处于屏幕的中心。理解这段代码,请参考图 12-15,A 点是目前屏幕的中心点,也是精灵的位置。玩家触摸 B 点,精灵会向 B 点移动。为了让精灵保持在屏幕中心,地图一定要向相反的方向移动(见图 12-15 中虚线)。

第⑤行代码 local pointA = cc.p(size.width/2, size.height/2)是获取屏幕中心点(A 点)。第⑥行代码是获取移动地图目标位置(B 点)。第⑦行代码是计算 A 点与 B 点两者之

差,这个差值就是地图要移动的距离。由于精灵的世界坐标就是地图层的模型坐标,即精灵的坐标原点是地图的左下角,因此第⑧行代码_layer:setPosition(offset)是将地图坐标原点移动到 offset 位置。

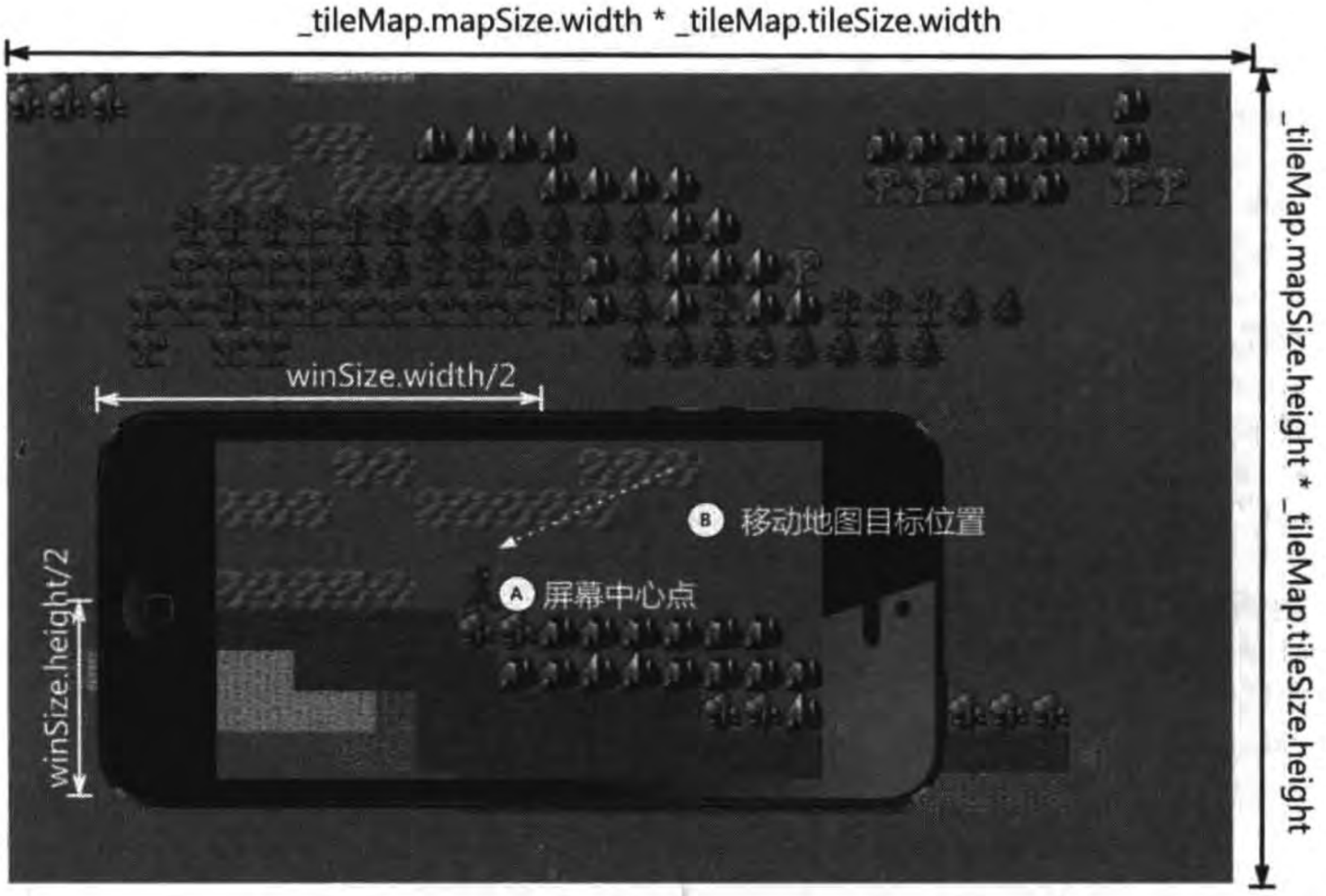


图 12-15 移动地图

本章小结

通过对本章的学习,使广大读者了解到瓦片地图在解决大背景问题的优势。熟悉 Cocos2d-x 中访问瓦片地图 Lua API。掌握瓦片地图的开发过程。

第 13 章

物 理 引 擎

你玩过 Angry Birds(愤怒的小鸟)①(见图 13-1)和 Bubble Ball②(见图 13-2)吗? 在 Angry Birds 中小鸟在空中飞行,它飞行的轨迹是一个符合物理规律的抛物线,通过改变它的发射角度,让它飞得更远。还有建筑物的倒塌也跟人们现实生活中看到的是一样的。Bubble Ball 中的小球沿着木板的滚动非常逼真,它滚动的距离与下落的高度、木板的倾斜角度和材质都有关系。

图 13-1　Angry Birds(愤怒的小鸟)游戏

我们发现这些游戏的共同特点是,场景中的精灵能够符合物理规律,与人们生活中看到的效果基本一样。这种在游戏世界中模仿真实世界物理运动规律的能力,是通过“物理引擎”实现的。严格意义上说“物理引擎”模仿的物理运动规律,是指牛顿的力学运动规律,而

---

① 《愤怒的小鸟》(芬兰语为 Vihainen Lintu,英语为 Angry Birds)是芬兰 Rovio 娱乐推出的一款益智游戏。在游戏中玩家控制一架弹弓发射无翅小鸟来打击建筑物和小猪,并以摧毁关中所有的小猪为最终目的。

② Bubble Ball 是益智类游戏,玩家通过改变场景中的木板的位置和角度,使得小球滚到小旗那里,玩家就赢得此关。它是由 Robert Nay 开发的,当时他只有 14 岁,在他的母亲帮助下,使用 Corona SDK 游戏引擎开发的。

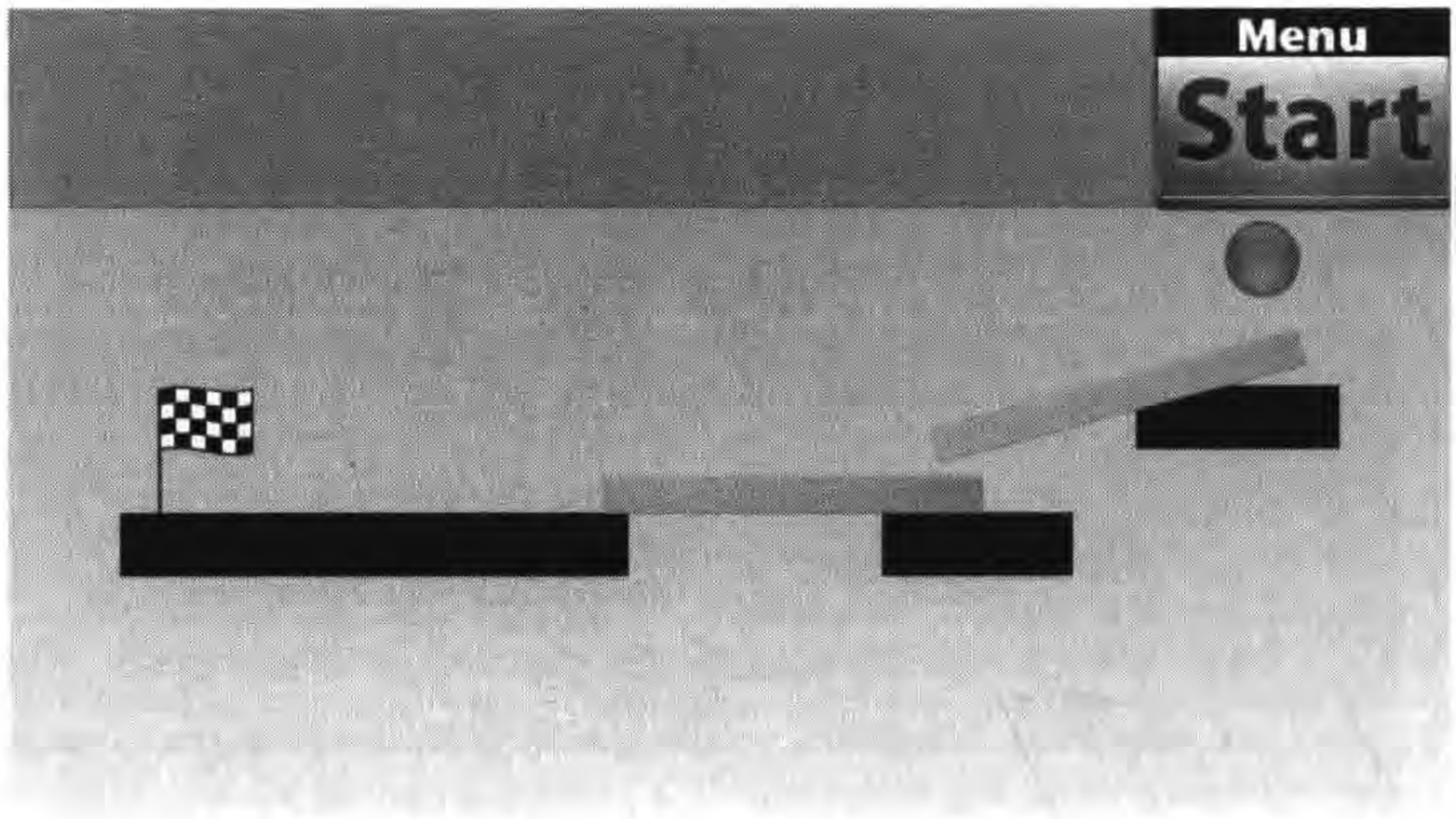


图 13-2　Bubble Ball 游戏

不符合量子力学运动规律。

13.1　使用物理引擎

物理引擎能够模仿真实世界的物理运动规律，使得精灵做出自由落体、抛物线运动、互相碰撞、反弹等效果。

使用物理引擎还可以进行精确的碰撞检测，检测碰撞不使用物理引擎时往往只是将碰撞的精灵抽象为矩形、圆形等规则的几何图形，这样算法比较简单。但是碰撞的真实效果就比较差了，而且自己编写时往往算法没有经过优化，性能也不是很好。物理引擎是经过优化的，所以建议还是使用已有的成熟的物理引擎。

目前主要使用的物理引擎有 Box2D 和 Chipmunk。在 Cocos2d-x 3. x 之后对 Chipmunk 引擎进行了封装，拥有了一套自己的物理引擎，它能够与 Cocos2d-x 结合更加紧密，也不用关心物理引擎的技术细节问题。Cocos2d-x Lua API 是绑定在 Cocos2d-x 之上的，Cocos2d-x Lua API 中绑定了 Cocos2d-x 3. x 中的物理引擎，并提供了相应的 API。

13.1.1　物理引擎核心概念

在详细介绍物理引擎之前，首先介绍物理引擎的一些核心概念。这些概念主要有：

(1) 世界(World)。游戏中的物理世界。

(2) 物体(Body)。构成物理世界的基础，具有位置、旋转角度等特性，它上面的任何两点之间的距离都是完全不变的，它们就像钻石那样坚硬，也可以称为刚体(Rigid Body)。在任何力的作用下，体积和形状都不发生改变的物体叫做刚体(Rigid Body)。在物理学内，理想的刚体是一个固体的，尺寸值有限的，形变情况可以被忽略的物体。不论是否受力，在刚体内任意两点的距离都不会改变。在运动中，刚体上任意两条平行直线在各个时刻的位置

都保持平行。——引自百度百科 http://baike.baidu.com/view/68357.htm。

(3) 形状(Shape)。物体的形状。一个依附于物体的二维碰撞几何结构,具有摩擦和弹性等材料属性。由于物体被抽象成刚体,忽略了形状。但是物体间的摩擦和碰撞是与形状有关的,这时需要将形状依附于物体上。

(4) 接触点(Contact)。管理检测碰撞。

(5) 关节(Joint)。把两个或多个物体固定到一起的约束。

如果将物体比作人的灵魂,那么形状就是人的躯体,从辩证法的观点来讲是内容与形式的关系。当创建一个物体的时候,它是没有形状的。然后根据需要把形状附着于物体上,这时物体就会受到摩擦力、弹力等影响,碰撞检测也是与形状有联系的,一个物体也可以附着多个形状。

13.1.2 物理引擎与精灵关系

物理引擎本身不包括精灵,它与精灵之间是相互独立的,精灵不会自动地跟着物理引擎中的物体做物理运动,通常需要编写代码将物体与精灵连接起来,同步它们的状态。

有些游戏引擎对精灵进行了封装,使它们能够与物理引擎中的物体同步,不需要编写代码实现同步,Cocos2d-x Lua API 中的 Sprite 类就实现了这个目的,再配合其他的一些封装类,使得在 Cocos2d-x Lua API 中使用物理引擎,根本不用关心它们的细节问题。

13.2 Cocos2d-x 中物理引擎

使用 Cocos2d-x 封装好的物理引擎,能够使开发人员不需要再关注物理引擎的使用细节,也不需要自己同步物体与精灵位置和角度了。

13.2.1 Cocos2d-x 物理引擎 Lua API

在 Cocos2d-x 3 版本后物理世界被融入游戏引擎的场景中,可以指定这个场景是否使用物理引擎。为此创建类 Scene 增加如下函数:

(1) createWithPhysics()。创建场景对象。

(2) addChildToPhysicsWorld(child)。增加节点对象到物理世界。

(3) getPhysicsWorld()。获得 PhysicsWorld 物理世界对象。

Cocos2d-x Lua API 在节点类 Node 中增加了 physicsBody 属性,可以将物理引擎中的物体添加到 Node 对象中。

此外,Cocos2d-x Lua API 为物理引擎增加了很多类,其中主要的有如下几类:

(1) PhysicsWorld 类。封装物理引擎世界(World)。

(2) PhysicsBody 类。封装物理引擎物体(Body)。

(3) PhysicsShape 类。封装物理引擎形状(Shape)。

(4) PhysicsContact 类。封装物理引擎碰撞类(Contact)。

(5) EventListenerPhysicsContact 类。碰撞检测监听类。

(6) PhysicsJoint 类。封装物理引擎关节(Joint)。

下面重点介绍 PhysicsShape、EventListenerPhysicsContact 和 PhysicsJoint。

1. PhysicsShape 类

形状类 PhysicsShape 是一个抽象类,它有很多子类,类图如图 13-3 所示,它的子类有:

(1) PhysicsShapeCircle。圆圈。

(2) PhysicsShapeBox。矩形盒子。

(3) PhysicsShapePolygon。多边形。

(4) PhysicsShapeEdgeSegment。有边的线段。

(5) PhysicsShapeEdgeBox。有边的矩形盒子。

(6) PhysicsShapeEdgePolygon。有边的多边形。

(7) PhysicsShapeEdgeChain。有边的链形。

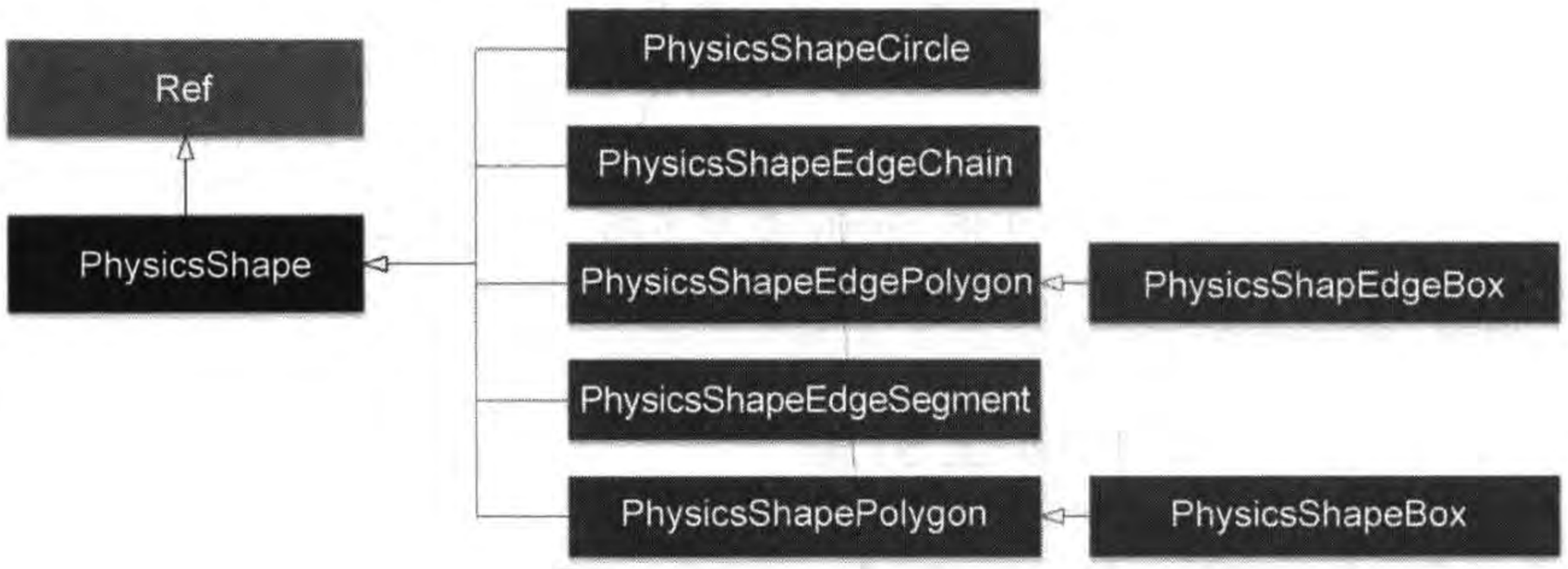


图 13-3　PhysicsShape 类图

2. EventListenerPhysicsContact 类

碰撞检测监听类 EventListenerPhysicsContact。类图如图 13-4 所示,它还有很多子类。

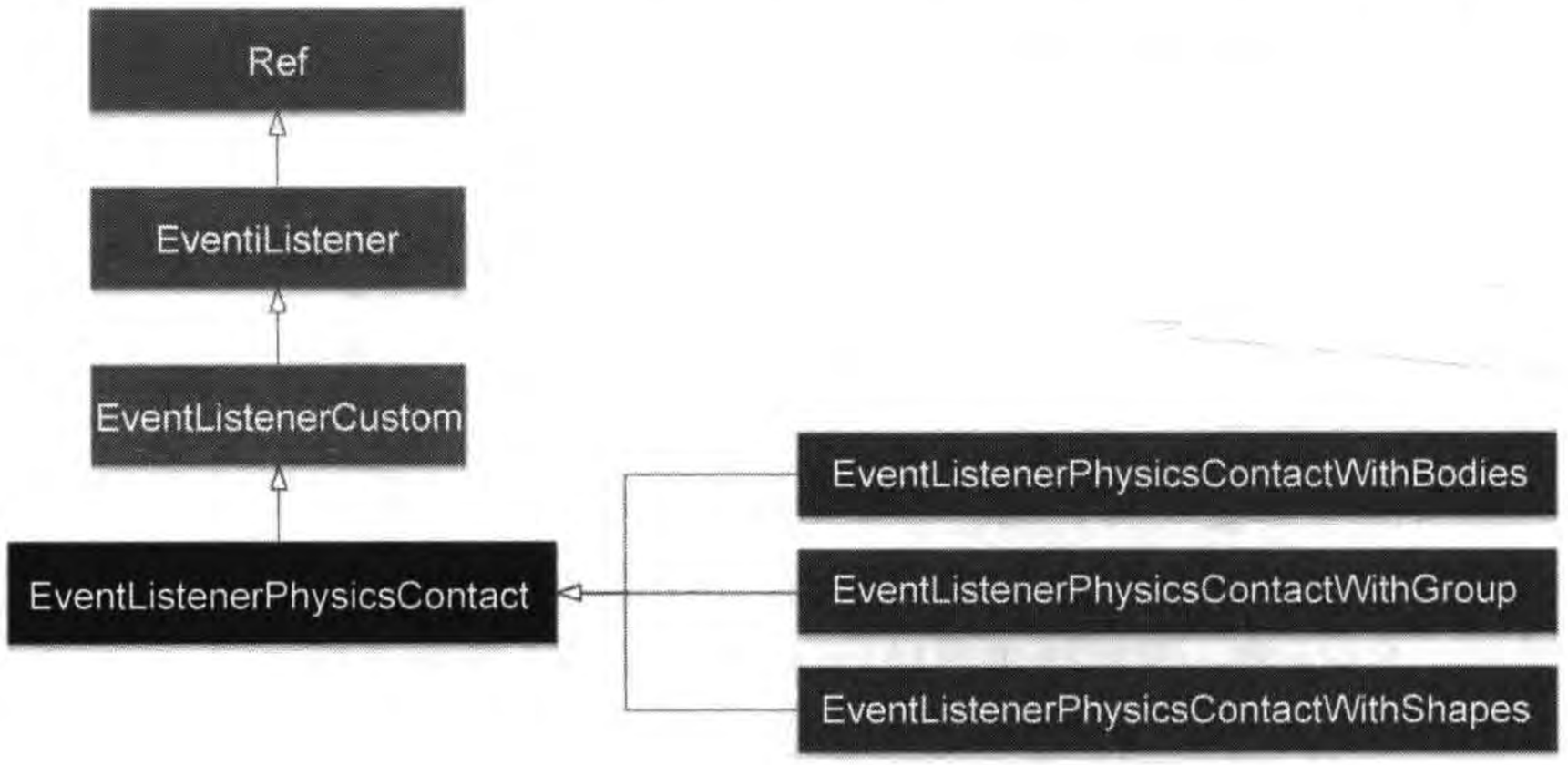


图 13-4　EventListenerPhysicsContact 类

EventListenerPhysicsContact 是碰撞检测事件监听器：

(1) EVENT_PHYSICS_CONTACT_BEGIN。两个物体开始接触该事件所指定的函数，但只触发一次。返回 true 情况可以触发后面的两个事件(EVENT_PHYSICS_CONTACT_PRESOLVE 和 EVENT_PHYSICS_CONTACT_POSTSOLVE)。

(2) EVENT_PHYSICS_CONTACT_PRESOLVE。持续接触时响应，它会被多次调用。返回 true 情况下触发后面的 EVENT_PHYSICS_CONTACT_POSTSOLVE 事件。

(3) EVENT_PHYSICS_CONTACT_POSTSOLVE。持续接触时响应，在触发 EVENT_PHYSICS_CONTACT_PRESOLVE 后触发。

(4) EVENT_PHYSICS_CONTACT_SEPARATE。分离时响应。但只触发一次。

碰撞检测事件响应顺序如图 13-5 所示，注意 EVENT_PHYSICS_CONTACT_BEGIN 和 EVENT_PHYSICS_CONTACT_PRESOLVE 事件所指定的函数，返回类型是布尔值。

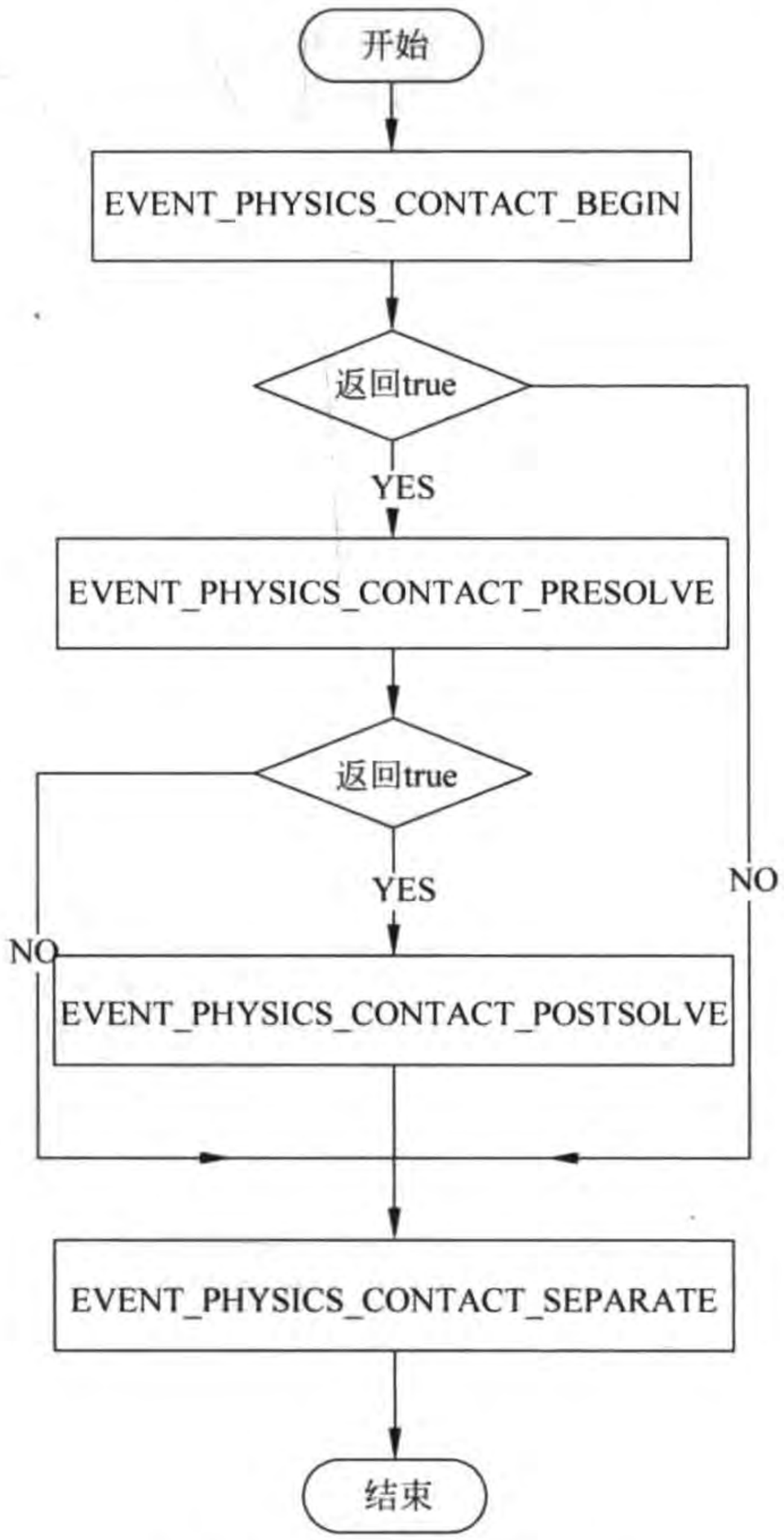


图 13-5　碰撞检测事件响应

如果它们都返回 true,事件的触发的顺序是 EVENT_PHYSICS_CONTACT_BEGIN→EVENT_PHYSICS_CONTACT_PRESOLVE→EVENT_PHYSICS_CONTACT_POSTSOLVE→EVENT_PHYSICS_CONTACT_SEPARATE,否则如图 13-5 所示。总之,EVENT_PHYSICS_CONTACT_BEGIN 和 EVENT_PHYSICS_CONTACT_SEPARATE 会被触发一次,EVENT_PHYSICS_CONTACT_PRESOLVE 和 EVENT_PHYSICS_CONTACT_POSTSOLVE 会多次触发,但是前提是两个物体没有分离。

3. PhysicsJoint 类

关节类 PhysicsJoint 是一个抽象类。类图如图 13-6 所示,它还有很多子类。

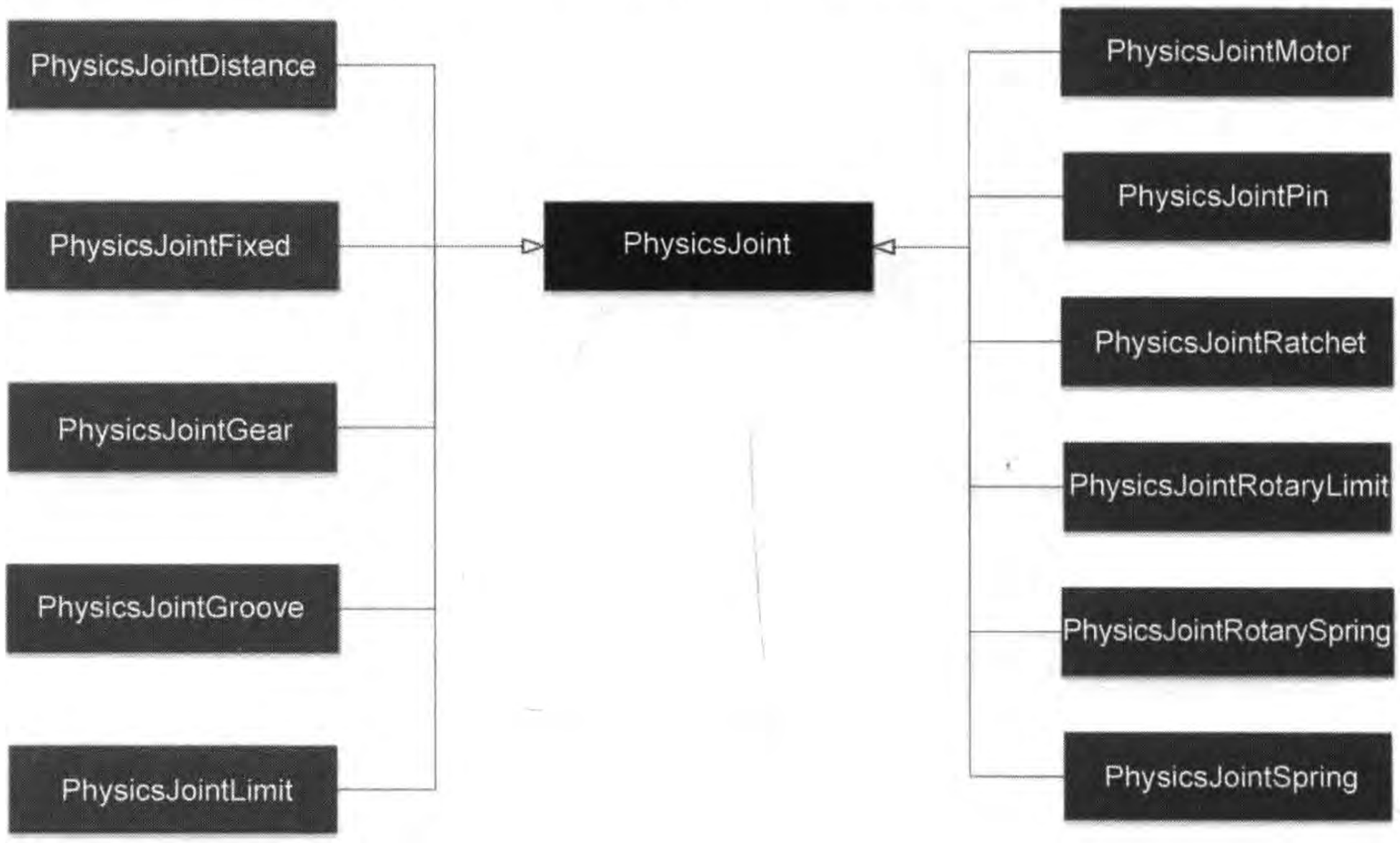


图 13-6　PhysicsJoint 类

在物理引擎中关节类很多,随着 Cocos2d-x 3 版本的增加,PhysicsJoint 的子类会越来越多,本节重点讲述 PhysicsJointDistance 类,其他的关节类的创建和使用与 PhysicsJointDistance 类似,只是约束的形式不同,这里不再介绍。

PhysicsJointDistance 类是距离关节类,它是最简单的关节之一。两个物体上面各自有一点,两点之间的距离必须固定不变。创建 PhysicsJointDistance 的静态函数定义如下:

```
cc.PhysicsJointDistance:construct(
        a,                          -- PhysicsBody 类型
        b,                          -- PhysicsBody 类型
        anchr1,
        anchr2
)
```

其中,a 和 b 参数是两个互相约束的物体;anchr1 是连接 a 物体锚点;anchr2 是 b 物体的锚点。

注意　物理引擎关节所提到的锚点与 Cocos2d-x Lua API 中节点(Node)的锚点 anchorPoint 不同。Cocos2d-x Lua API 中 Node 中的锚点是相对位置的比例，如图 13-7 所示锚点在 Node1 中的中心，则它的相对位置为(0.5,0.5)。物理引擎关节中的锚点不是相对值，而是绝对的坐标点，如图 13-8 所示，采用模型坐标(本地坐标)表示两个物体的锚点，Body1 的锚点位置在 Body1 的中心，它的坐标为(0,0)。Body2 的锚点位置在 Body2 的上边界的中央，它的坐标为(0,Body2. Width /2)，Body2. Width /2 表示 Body2 宽度的一半。

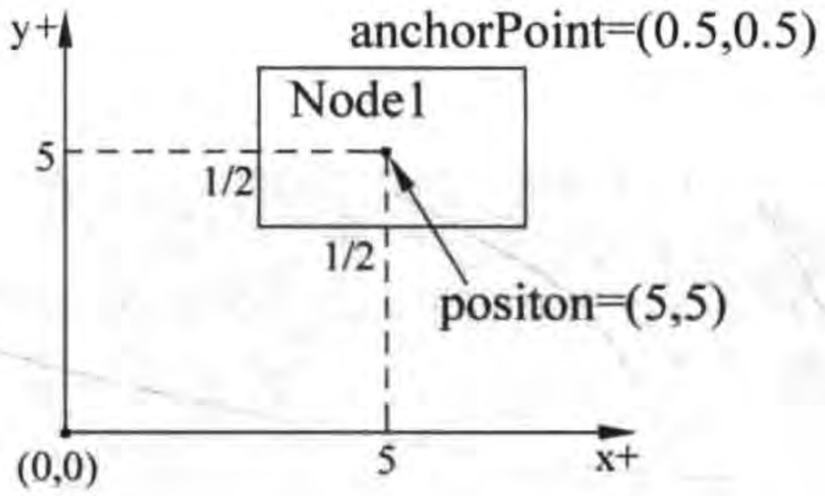


图 13-7　Cocos2d-x 中节点的锚点

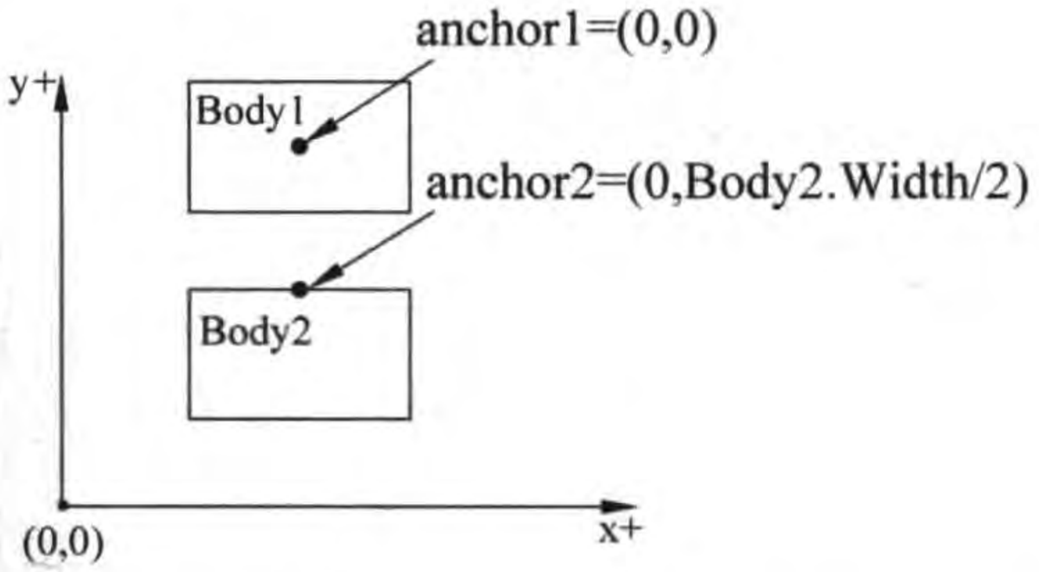


图 13-8　物理引擎关节的锚点

13.2.2　实例：HelloPhysicsWorld

通过一个实例介绍在 Cocos2d-x Lua API 中使用物理引擎的开发过程，熟悉这些 API 的使用。这个实例运行后的场景如图 13-9 所示，当场景启动后，玩家可以触摸单击屏幕，每次触摸时，就会在触摸点生成一个新的精灵，精灵的运行是自由落体运动。

使用物理引擎的一般步骤，如图 13-10 所示。

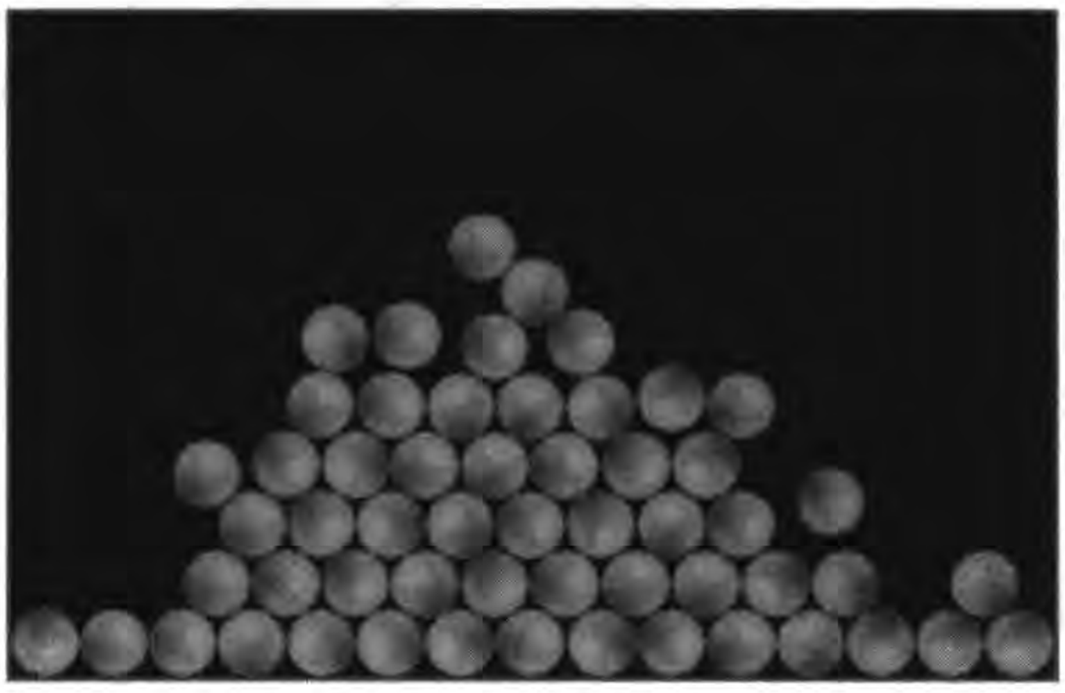


图 13-9　HelloPhysicsWorld 实例

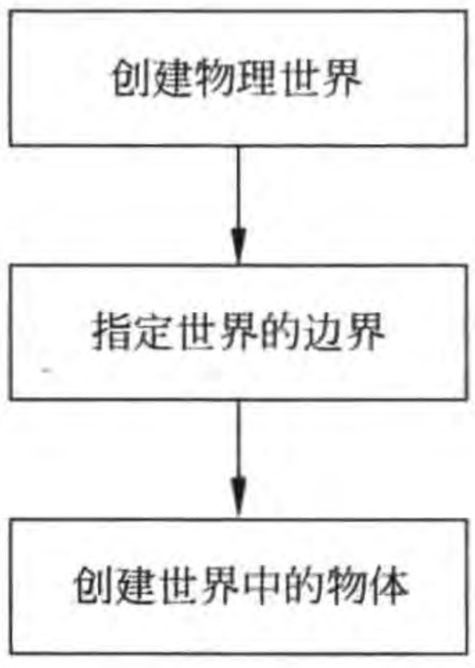


图 13-10　使用物理引擎的一般步骤

这个过程与上帝创建世界的过程类似,上帝首先创建了世界,然后为世界指定了边界,否则万物就会掉到世界之外的混沌里去,最后上帝创建了万物。

当然这只是一个最基本的步骤,有的时候还需要碰撞检测和使用关节等处理。下面就按照这个步骤介绍代码部分。

1. 创建物理世界

在 GameScene 场景中创建物理世界,代码如下:

```
localGameScene = class("GameScene",function()
    local scene = cc.Scene:createWithPhysics()                                    ①
    scene:getPhysicsWorld():setDebugDrawMask(cc.PhysicsWorld.DEBUGDRAW_ALL)       ②
    return scene
end)
```

上述第①行代码 cc. Scene:createWithPhysics()语句是创建带物理引擎世界场景,使用 createWithPhysics()函数在场景中进行初始化物理引擎,可以通过场景的 getPhysicsWorld()函数获取初始化的物理世界(PhysicsWorld)对象。也可以根据需要在这里设置物理世界,其中第②行代码设置在物理世界中绘制调试遮罩,这会把物体的形状绘制出来。因为世界中的物体如果没有与精灵绑定到一起,是看不到它的。这主要用于调试,当调试结束后,游戏发布的时候,需要把它关闭。关闭绘制调试遮罩的场景如图 13-11 所示。

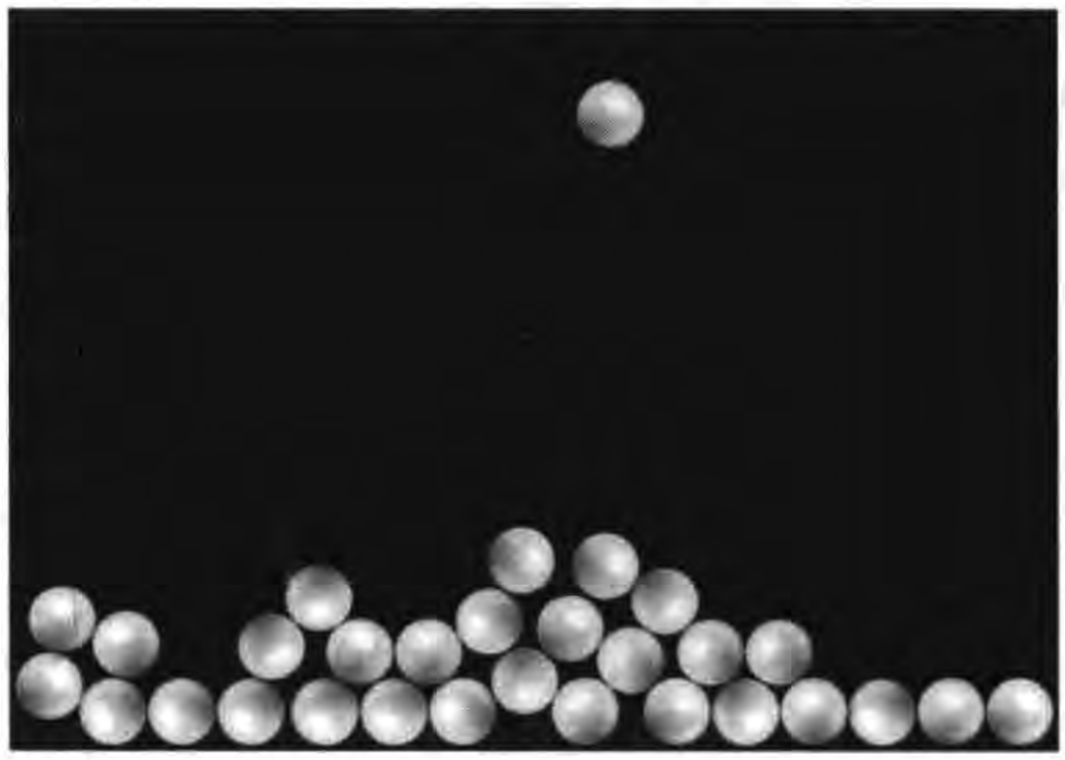


图 13-11 HelloPhysicsWorld 实例

2. 指定世界的边界

在 GameScene 场景中指定世界的边界,代码如下:

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    --定义世界的边界
    local body = cc.PhysicsBody:createEdgeBox(size,
                                cc.PHYSICSBODY_MATERIAL_DEFAULT, 5.0)             ①
    local edgeNode = cc.Node:create()                                             ②
    edgeNode:setPosition(cc.p(size.width/2,size.height/2))                        ③
    edgeNode:setPhysicsBody(body)                                                 ④
    layer:addChild(edgeNode)                                                      ⑤

    local function touchBegan(touch, event)                                       ⑥
        local location = touch:getLocation()
        self:addNewSpriteAtPosition(location)                                     ⑦
        return false
    end
```

```
-- 创建一个事件监听器 OneByOne 为单点触摸
local listener = cc.EventListenerTouchOneByOne:create()
-- 设置是否吞没事件,在 onTouchBegan 方法返回 true 时吞没
listener:setSwallowTouches(true)
-- onTouchBegan 事件回调函数
listener:registerScriptHandler(touchBegan,cc.Handler.EVENT_TOUCH_BEGAN )
local eventDispatcher = self:getEventDispatcher()
-- 添加监听器
eventDispatcher:addEventListenerWithSceneGraphPriority(listener, layer)

return layer
end
```

上述代码是初始化层函数,在这个函数中可以指定世界的边界,世界边界也是一个物体,第①行代码 cc.PhysicsBody:createEdgeBox 是创建物体对象,静态函数 createEdgeBox 指定世界边界是矩形盒子,第一个参数指定矩形的大小;第二个参数是设置材质,cc.PHYSICSBODY_MATERIAL_DEFAULT 常量是默认材质,材质是由结构体 PhysicsMaterial 定义的,结构体 PhysicsMaterial 成员有:density(密度)、friction(摩擦系数)和 restitution(弹性系数)。密度可以用来计算物体的质量,密度可以为零或者为正数。摩擦系数经常会设置在 0.0～1.0 之间,0.0 表示没有摩擦力,1.0 表示强摩擦。弹性系数的值通常设置到 0.0～1.0 之间,0.0 表示物体不会弹起,1.0 表示物体会完全反弹,即称为弹性碰撞;createEdgeBox 函数的第三个参数是设置边的宽度。

第②行代码是创建边界节点对象,这个节点对象用作世界边界对象,第③行代码是设置节点对象的位置。第④行代码 edgeNode:setPhysicsBody(body)是设置与节点相关的物体对象。通过这条语句使得游戏场景中的节点对象(精灵等)与物体关联起来。第⑤行代码是将节点对象添加到层中。

第⑥行代码单点触摸事件的回调函数。第⑦行代码是触摸发生时调用 addNewSpriteAtPosition 函数。

3. 创建世界中的物体

在 GameScene 场景中创建世界中的物体是通过 addNewSpriteAtPosition 函数实现的,代码如下:

```
function GameScene:addNewSpriteAtPosition(pos)
    local sp = cc.Sprite:create("Ball.png")                                    ①
    sp:setTag(1)                                                               ②
    local body = cc.PhysicsBody:createCircle(sp:getContentSize().width / 2)    ③
    sp:setPhysicsBody(body)                                                    ④
    sp:setPosition(pos)
    self:addChild(sp)
end
```

上述第①行代码 cc.Sprite:create("Ball.png")是创建一个精灵对象,第②行代码 sp:setTag(1)是设置精灵的 tag 属性,在检查碰撞时可以通过 tag 属性判断并获得精灵对象。

第③行代码是通过 PhysicsBody 的静态 createCircle 函数创建圆圈形状物体。

PhysicsBody还有很多类似的 create 函数，如 createBox、createCircle、createPolygon 和 createEdgePolygon 等函数，这些函数与物理形状是对应的。这里只详细解释 createCircle 函数，createCircle 函数 API 如下：

```
cc.PhysicsBody:createCircle(float radius,
    material = cc.PHYSICSBODY_MATERIAL_DEFAULT,          --PhysicsMateria类型
    offset = cc.p(0, 0)
)
```

其中第一个参数 radius 是设置圆圈的半径，第二个参数 material 是材质，可以省略，默认值是 cc.PHYSICSBODY_MATERIAL_DEFAULT，所以在初始化层函数中定义世界的边界时，createEdgeBox 语句中的 material 参数也是可以省略的。第三个参数 offset 是偏移量，可以省略，默认值是 cc.p(0, 0)。

第④行代码 sp:setPhysicsBody(body)是设置与精灵相关的物体对象。

13.2.3 实例：接触与碰撞检测

在物理引擎作用的虚拟物理世界中，物体之间会相互作用，其中接触(contact)与碰撞(collision)是非常重要的相互作用方式。下面解释一下这两个概念：

(1) 接触，是两个物体彼此靠近，当边缘发生碰触时，则这两个物体发生了接触，如图 13-12(a)。接触常用来进行接触检测，根据检测结果进行处理。例如，检测到子弹与飞机发生接触，会让飞机发生爆炸，同时让子弹和飞机在屏幕中消失。

(2) 碰撞，是为了防止两个物体接触并碰撞后，发生彼此互穿现象，如图 13-12(b)所示。希望碰撞的结果是两个物体彼此分开，物理引擎能够自动计算出物体上的反作用力，并作用于物体，这会使动态物体向相反方向运动，如图 13-12(c)所示。

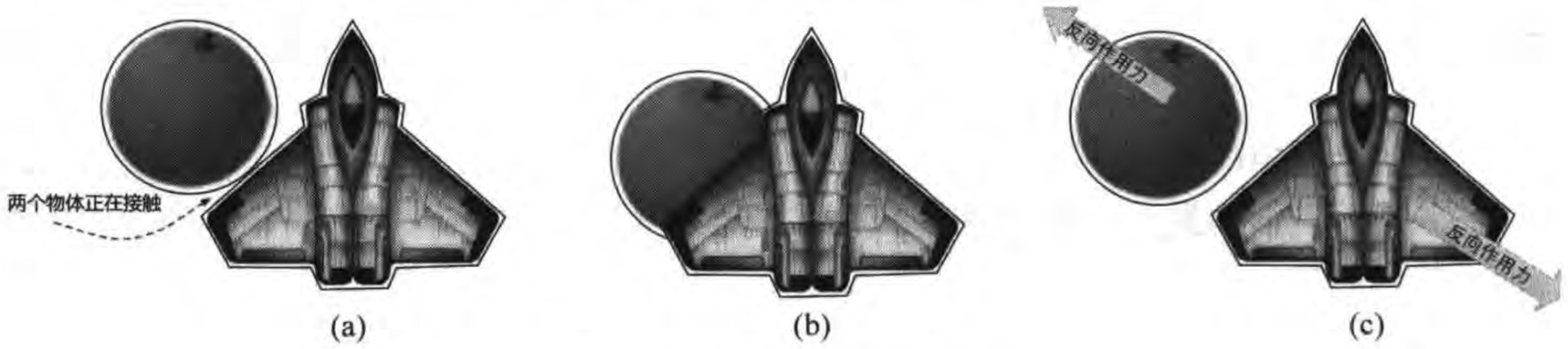


图 13-12 物体接触与碰撞

一个游戏场景中经常会有很多精灵物体，有的时候希望某些物体之间发生接触和碰撞，而有时不需要。物理引擎为物体提供如下三种掩码：

(1) 类别掩码(CategoryBitmask)，相同类型的物体具有相同的类别掩码，可以在一个场景中最多定义 32 种类别掩码。

(2) 接触检测掩码(ContactTestBitmask)，物体 A 的接触检测掩码与物体 B 的类别掩码进行“逻辑与”运算，如果结果为非零值，则物体 A 与物体 B 发生接触，触发接触事件。

(3) 碰撞掩码(CollisionBitmask)，物体 A 的碰撞掩码与物体 B 的类别掩码进行“逻辑

与”运算，如果结果为非零值，则物体 A 与物体 B 发生碰撞，即物体 A 与物体 B 不会发生互穿现象。

Cocos2d-x Lua API 中的物理引擎 PhysicsBody 类提供了三个设置掩码函数分别是：

(1) setCategoryBitmask(bitmask)。设置类别掩码函数。

(2) setContactTestBitmask(bitmask)。设置接触检测掩码函数。

(3) setCollisionBitmask(bitmask)。设置碰撞掩码函数。

假设有三个物体(body1、body2 和 body3)，它们设置如下：

```
body1:setCategoryBitmask(0x01);      //0001
body1:setCollisionBitmask(0x03);     //0011

body2:setCategoryBitmask(0x02);      //0010
body2:setCollisionBitmask(0x01);     //0001

body3:setCategoryBitmask(0x04);      //0100
body3:setCollisionBitmask(0x06);     //0110
```

body1 能与 body1、body1 能与 body2、body3 能与 body3 发生碰撞，body1 不能与 body3 发生碰撞，因为碰撞发生是有方向性的，body2 不能与 body3 发生碰撞，但 body3 能与 body2 发生碰撞。

在 Cocos2d-x Lua API 中使用事件派发机制管理碰撞事件，EventListenerPhysicsContact 是碰撞事件监听器。碰撞检测相关的 API 在前面一节介绍过了，下面通过一个实例介绍碰撞检测的实现。这个实例运行后的场景如图 13-13 所示，当场景启动后，玩家可以触摸点击屏幕，每次触摸时，就会在触摸点生成一个新的精灵，精灵的运行是自由落体运动。当这些精灵之间发生接触时，它们的颜色被设置为黄色，分离后颜色又恢复到原来状态了。

图 13-13　检测碰撞实例

本实例涉及物理引擎中物体之间的碰撞检测,在两个物体接触到分离过程中,会发生一些事件,可以通过注册监听器 EventListenerPhysicsContact 来响应这些事件。

GameScene.lua 中与碰撞检测相关的代码如下:

```
-- 创建层
function GameScene:createLayer()

    local layer = cc.Layer:create()

    ...

    local function onContactBegin(contact)                                    ①
        local spriteA = contact:getShapeA():getBody():getNode()               ②
        local spriteB = contact:getShapeB():getBody():getNode()               ③

        if spriteA and spriteA:getTag() == 1 and spriteB and spriteB:getTag() == 1 then  ④
            spriteA:setColor(cc.c3b(255,255,0))
            spriteB:setColor(cc.c3b(255,255,0))
        end
        cclog("onContactBegin")
        return true
    end

    local function onContactPreSolve(contact)                                 ⑤
        cclog("onContactPreSolve")
        return true
    end

    local function onContactPostSolve(contact)                                ⑥
        cclog("onContactPostSolve")
    end

    local function onContactSeparate(contact)                                 ⑦
        local spriteA = contact:getShapeA():getBody():getNode()
        local spriteB = contact:getShapeB():getBody():getNode()

        if spriteA and spriteA:getTag() == 1 and spriteB and spriteB:getTag() == 1 then
            spriteA:setColor(cc.c3b(255,255,255))
            spriteB:setColor(cc.c3b(255,255,255))
        end
        cclog("onContactSeparate")
    end


    -- 创建一个事件监听器 OneByOne 为单点触摸
    local listener = cc.EventListenerTouchOneByOne:create()
    -- 设置是否吞没事件,在 EVENT_PHYSICS_CONTACT_BEGIN 方法返回 true 时吞没
    listener:setSwallowTouches(true)
    -- onTouchBegan 事件回调函数
    listener:registerScriptHandler(touchBegan,cc.Handler.EVENT_TOUCH_BEGAN )

    local contactListener = cc.EventListenerPhysicsContact:create()
    contactListener:registerScriptHandler(onContactBegin,
```

```
                    cc.Handler.EVENT_PHYSICS_CONTACT_BEGIN)
contactListener:registerScriptHandler(onContactPreSolve,
                    cc.Handler.EVENT_PHYSICS_CONTACT_PRESOLVE)
contactListener:registerScriptHandler(onContactPostSolve,
                    cc.Handler.EVENT_PHYSICS_CONTACT_POSTSOLVE)
contactListener:registerScriptHandler(onContactSeparate,
                    cc.Handler.EVENT_PHYSICS_CONTACT_SEPARATE)

local eventDispatcher = self:getEventDispatcher()
-- 添加监听器
eventDispatcher:addEventListenerWithSceneGraphPriority(listener, layer)
eventDispatcher:addEventListenerWithSceneGraphPriority(contactListener, layer)      ⑧

    return layer
end
```

上述第①、第⑤～⑦行代码定义了事件处理的回调函数。第②和第③行代码是从接触点中取出互相接触的两个节点对象，它们的取值过程有点复杂，首先接触点使用 getShapeA()和 getShapeB()函数获得物体形状，再通过形状的 getBody()函数获得物体，通过物体的 getNode()函数获得与形状相关的节点对象。第④行代码是进行判断，判断从接触点取出的节点对象是否存在，并且判断 tag 属性是否为 1。

第⑧行代码 addEventListenerWithFixedPriority 是指定固定的事件优先级注册监听器，事件优先级决定事件响应的优先级别，值越小优先级越高。

GameScene. lua 中 addNewSpriteAtPosition 函数，代码如下。

```
function GameScene:addNewSpriteAtPosition(pos)
    local sp = cc.Sprite:create("Ball.png")
    sp:setTag(1)
    local body = cc.PhysicsBody:createCircle(sp:getContentSize().width / 2)

    body:setContactTestBitmask(0x1)                                          ①

    sp:setPhysicsBody(body)
    sp:setPosition(pos)
    self:addChild(sp)
end
```

这个函数的代码与上一节介绍的实例基本一致，但是需要注意的是在第①行添加了 body:setContactTestBitmask(0x1)代码，它的作用是设置物体接触时能否触发 EventListenerPhysicsContact 中定义的接触事件。如果两个物体的接触检测掩码(ContactTestBitmask)执行“逻辑与”运算，结果为非零值，表明这两个物体会触发接触检测事件。默认值 0x00000000 表示清除所有掩码位，0xFFFFFFFF 表示所有掩码位都设置为 1。

13.2.4　实例：使用关节

在游戏中可以通过关节约束两个物体的运动。通过一个距离关节实例，介绍一下如何使用关节。

这个实例运行后的场景如图 13-14 所示，当场景启动后，玩家可以触摸点击屏幕，每次触摸时，就会在触摸点和附近生成两个新的精灵，它们的运行是自由落体运动，它们之间的距离是固定的。图 13-15 所示是开启了绘制调试遮罩，从图中可见，调试遮罩不仅会显示物体，还会显示关节。

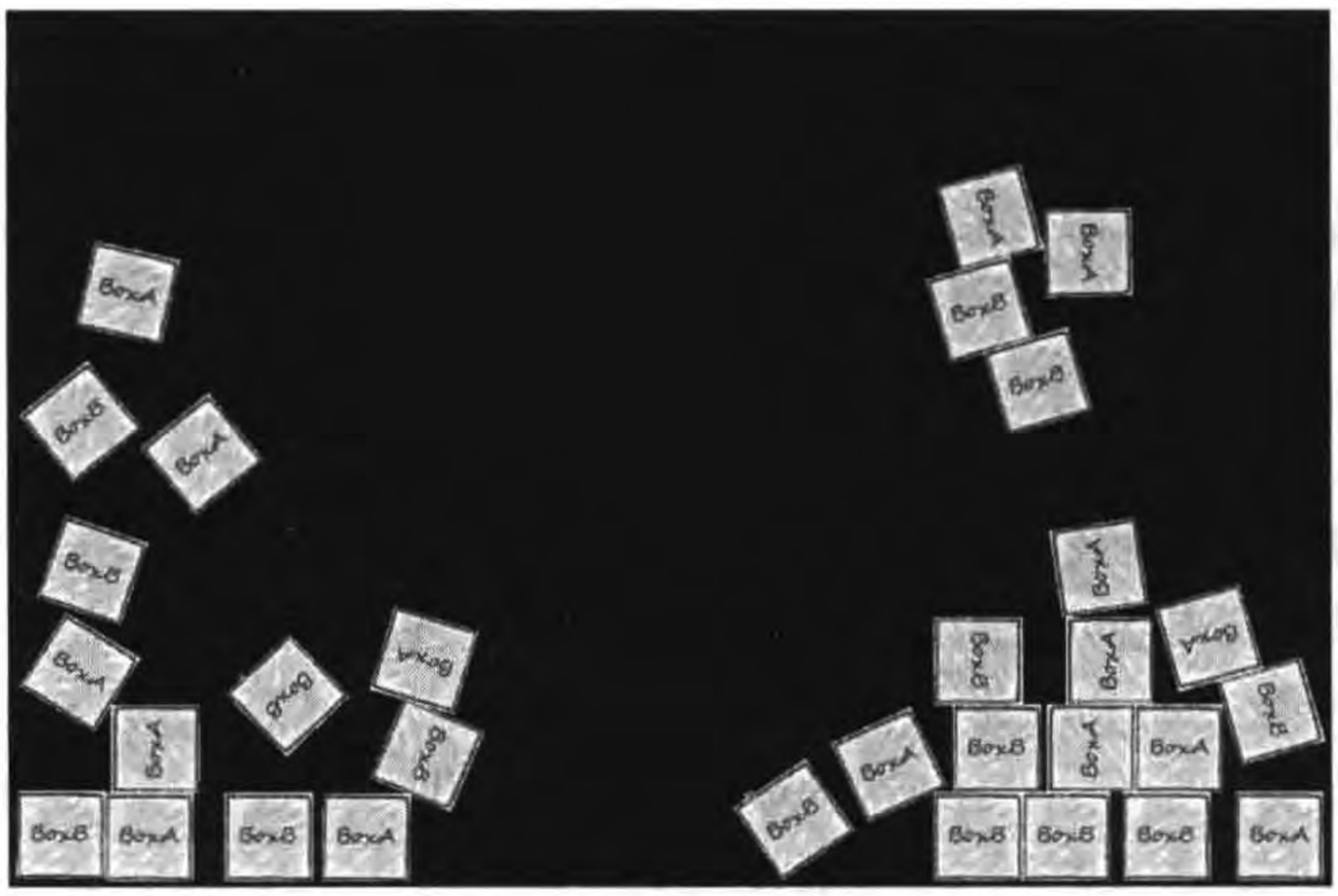


图 13-14　使用距离关节实例

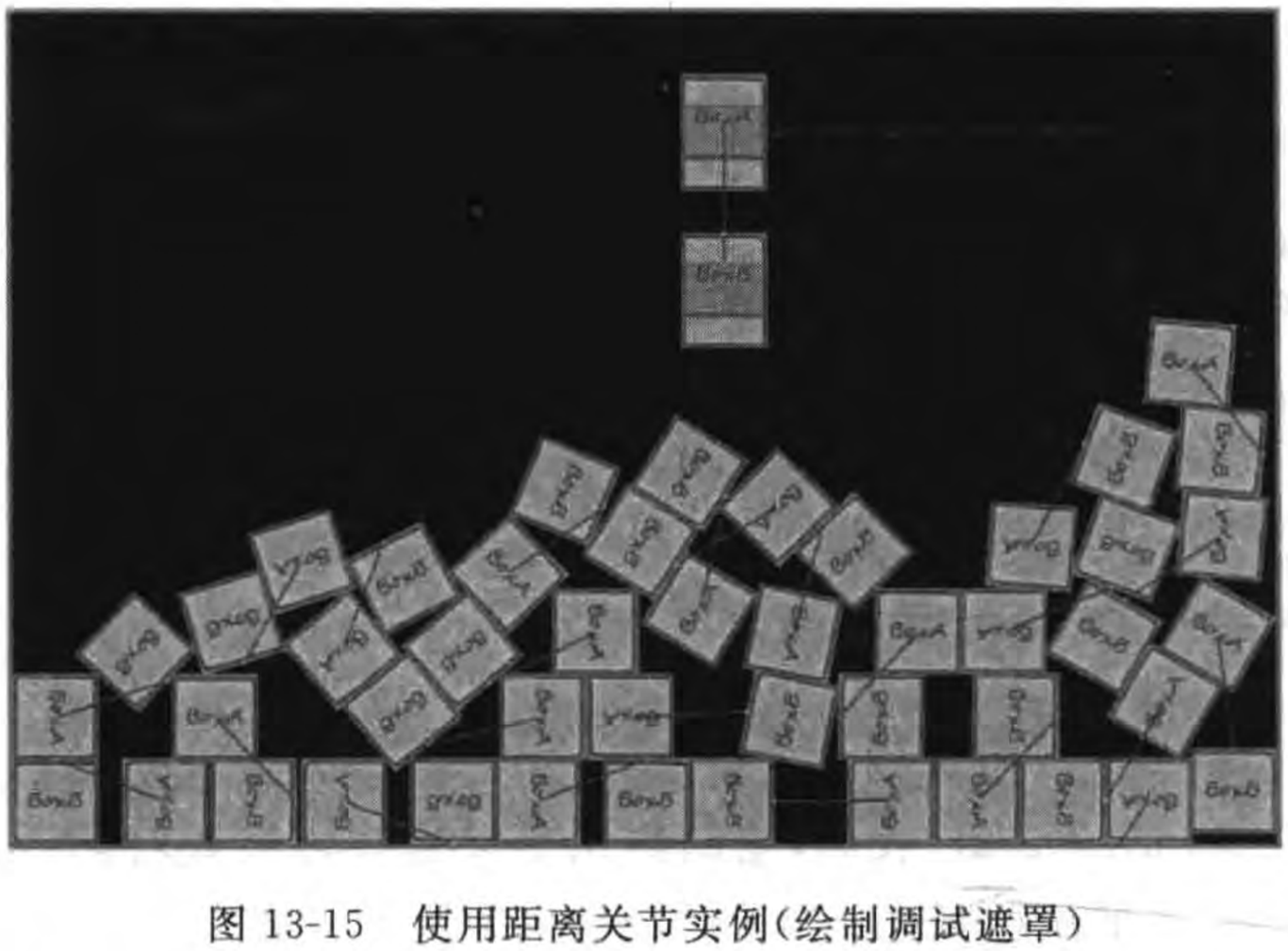


图 13-15　使用距离关节实例(绘制调试遮罩)

实例中创建物理世界和指定世界的边界代码与上一节实例类似，这里也不再解释这些函数代码了，主要介绍关节的相关代码，代码如下：

```
function GameScene:addNewSpriteAtPosition(pos)

    local boxA = cc.Sprite:create("BoxA2.png")                    ①
    boxA:setPosition(pos)                                         ②
```

```
local boxABody = cc.PhysicsBody:createBox(boxA:getContentSize())                    ③
boxA:setPhysicsBody(boxABody)                                                       ④
self:addChild(boxA, 10, 100)                                                        ⑤

local boxB = cc.Sprite:create("BoxB2.png")
boxB:setPosition(pos.x + 100, pos.y - 120)

local boxBBody = cc.PhysicsBody:createBox(boxB:getContentSize())
boxB:setPhysicsBody(boxBBody)
self:addChild(boxB, 20, 101)

local world = cc.Director:getInstance():getRunningScene():getPhysicsWorld()         ⑥

    local joint = cc.PhysicsJointDistance:construct(boxABody, boxBBody,
        cc.p(0, 0), cc.p(0, boxB:getContentSize().width / 2))                       ⑦
    world:addJoint(joint)                                                           ⑧
end
```

上面第①行代码创建精灵 boxA。第②行代码设置它的位置。第③行代码 cc.PhysicsBody:createBox(boxA:getContentSize())是创建矩形盒子物体。第④行代码 boxA:setPhysicsBody(boxABody)是设置与精灵相关的物体对象。第⑤行代码是将精灵添加到当前层中。

创建完成 boxA 和 boxABody，下面又紧接着创建了 boxB 和 boxBBody 对象。创建好它们之后就可以添加关节约束了，第⑥行代码是从场景中获得物理世界(PhysicsWorld)对象。第⑦行代码通过 PhysicsJointDistance 的静态函数 construct 创建距离关节对象，其中锚点坐标采用的是模型坐标(本地坐标)，如果获得的不是模型坐标，可以进行坐标转换。PhysicsBody 中提供两个坐标转换函数：

(1) world2Local(point)。世界坐标转换为模型坐标。

(2) local2World(point)。模型坐标转换为世界坐标。

第⑧行代码 world:addJoint(joint)是将创建关节添加到物理世界中。

本章小结

通过对本章的学习，读者可以了解什么是物理引擎。在本章中介绍了 Cocos2d-x Lua API 中的物理引擎，然后重点介绍了使用物理引擎的一般步骤，还有碰撞检测以及关节的使用。

第 14 章

Cocos2d-x GUI 控件

在 Cocos2d 游戏引擎家族中只有少量 GUI 控件，包括标签(Label)、菜单(Menu 和 MenuItem)和精灵等。事实上对于游戏开发，这些 GUI 控件也能满足开发人员的需要，但是由于 Cocos2d-x 有场景设计工具 Cocos Studio，Cocos Studio 能够使游戏开发人员在一个画布上通过拖曳控件方式实现游戏场景的设计，这样可以提高开发效率。为了配合 Cocos Studio 工具的使用，Cocos2d-x 引擎增加了一些 GUI 控件，这些 GUI 控件的类图如图 14-1 所示。

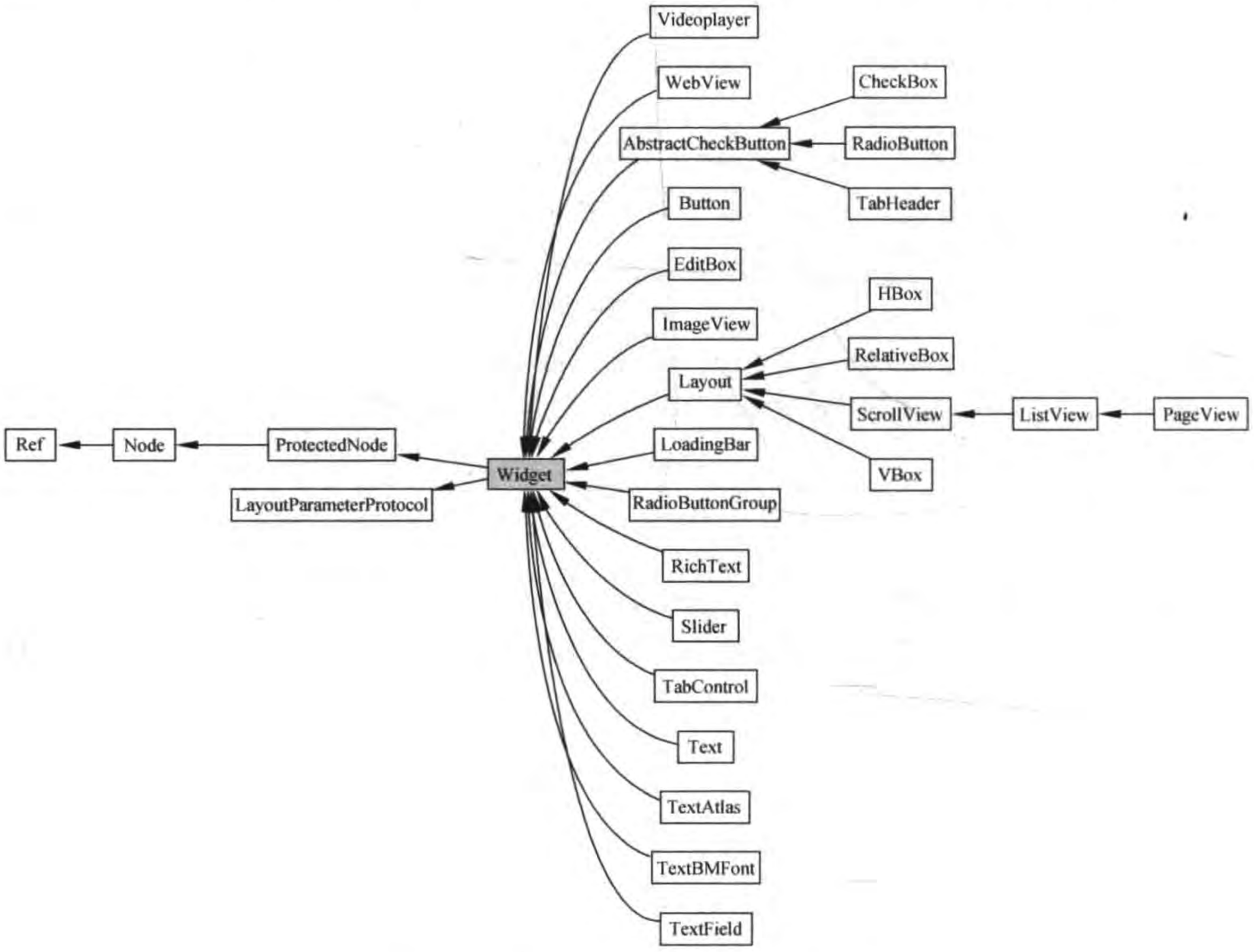


图 14-1　Cocos2d-x Lua APIGUI 控件

从图 14-1 可见这些控件都继承于 Widget，Layout 及其子类属于容器类型控件，它们可以包含其他的控件。接下来就介绍：Button、Text、TextBMFont、RichText、RadioButton、RadioButtonGroup、CheckBox、LoadingBar 和 Silder。

提示　在 Cocos2d-x Lua API 中，GUI 控件都有相同的命名空间 ccui，例如 ccui. Button，ccui 表示 Cocos Studio 中使用的控件。

14.1　按钮

Cocos2d-xLua API 中 GUI 按钮类是 Button。单击 Button 会触发触摸事件，并调用与事件相关的函数，可以为 Button 设置按钮标题、图标等属性。一个 Button 有三种状态：正常状态、选中状态和不可用状态。

Button 类中一个主要的创建函数 create 定义如下：

```
ccui.Button:create(normalImage,                    //正常状态显示的纹理
    selectedImage = "",                            //选中状态显示的纹理，默认值为“”
    disableImage = "",                             //不可用状态显示的纹理，默认值为“”
    texType = TextureResType::LOCAL                //默认值为 LOCAL
)
```

其中 TextureResType 纹理加载资源类型，它有两种类型 LOCAL 和 PLIST，LOCAL 表示纹理从图片文件加载，PLIST 表示纹理从 Atlas 图片集中加载。

下面通过一个实例介绍 Button 的使用。如图 14-2 所示的场景右下角的 ⏻ 按钮、图片和“Hello World”文字，完全可以采用前面章节所介绍的方式来实现。当然也可以使用 Cocos2d-x GUI 控件实现，“Hello World”文字可以使用 Text 类实现，Cocos2d-x 图标可以使用 ImageView 类实现，而右下角 ⏻ 可以使用 Button 类实现。这一节先采用 Button 实现右下角 ⏻ 功能。

图 14-2　HelloWorld 实例

下面看看具体的程序代码。首先看一下 GameScene.lua 文件。其中初始化相关代码如下：

```
-- 创建层
function GameScene:createLayer()

    local layer = cc.Layer:create()

    -- Button 对象事件处理
    local function ButtonCloseCallback(sender, eventType)                    ①
    cclog("Call ButtonCloseCallback...")

    if eventType == ccui.TouchEventType.began then          -- 手指触碰屏幕
        cclog("Touch Down")
    elseif eventType == ccui.TouchEventType.moved then      -- 手指在屏幕上移动
        cclog("Touch Move")
    elseif eventType == ccui.TouchEventType.ended then      -- 手指离开屏幕
        cclog("Touch Up")
    else
        cclog("Touch Cancelled")
    end
end


-- 1. Button 替换 Menu
-- 创建 Button 对象
local button = ccui.Button:create("CloseNormal.png", "CloseSelected.png")    ②
-- 设置 Button 位置
button:setPosition(size.width - button:getContentSize().width / 2,
                   button:getContentSize().height / 2)
-- 添加事件监听器
button:addTouchEventListener(ButtonCloseCallback)                            ③
-- 设置单击时会有放大效果
button:setPressedActionEnabled(true)
layer:addChild(button)

-- 创建 Label 对象
local label = cc.Label:createWithTTF("Hello World", "fonts/Marker Felt.ttf", 24)
label:setPosition(size.width/2, size.height - label:getContentSize().height)
    layer:addChild(label)

    local sprite = cc.Sprite:create("HelloWorld.png")
    sprite:setPosition(size.width/2, size.height/2)
    layer:addChild(sprite)

    return layer
end
```

上述第①行代码是回调函数 ButtonCloseCallback 通过判断 type 参数可以触发触摸事件的不同阶段。

第②行代码是创建 Button 对象，其中的 CloseNormal. png 是正常状态显示的图片、CloseSelected. png 是选中状态时显示的图片。本例中没有提供不可用状态所需图片。第③行代码是为 Button 添加事件监听器。

完成上述编码，可以测试一下。游戏项目启动后可以用手指在按钮上进行按下、抬起和移动等操作，控制台中输出类似如下日志内容：

```
Touch Move
Call ButtonCloseCallback...
Touch Move
Call ButtonCloseCallback...
Touch Move
Call ButtonCloseCallback...
Touch Move
Call ButtonCloseCallback...
Touch Move
Call ButtonCloseCallback...
Touch Move
Call ButtonCloseCallback...
Touch Move
Call ButtonCloseCallback...
Touch Move
Call ButtonCloseCallback...
Touch Up
```

14.2　ImageView

ImageView 是显示图片的控件，ImageView 不能接收事件处理。ImageView 中一个主要的创建函数 create 定义如下：

```
ccui.ImageView:create(imageFileName,          //要显示的纹理文件
    texType = TextureResType::LOCAL           //同 Button 类
)
```

下面将图 14-2 所示的场景 Cocos2d-x 图标使用 ImageView 类实现。修改 GameScene. lua 代码片段如下：

```
function GameScene:createLayer()
    ...
    -- 创建 ImageView 对象
    local imageView = ccui.ImageView:create("HelloWorld.png")                    ①
    imageView:setPosition(size.width / 2, size.height / 2)
    layer:addChild(imageView)

    return layer
end
```

上述第①行代码是创建 ImageView 对象，然后设置位置，并将 ImageView 添加到当前场景。

14.3 文本控件

文本控件类似于标签 Label,它是只读的、不能修改,一般用于显示一些文本信息。在 Cocos2d-x Lua API GUI 中的文本控件包括 Text、TextBMFont、TextAtlas 和 RichText。本节重点介绍 Text、TextBMFont 和 RichText 控件。

14.3.1 Text

Text 控件是最为基础的文本控件,能够展示系统默认字体和 TTF 字体的文本信息。Text 类中一个主要的创建函数 create 定义如下:

```
ccui.Text:create(textContent,                    //显示的文本内容
    fontName,                                    //系统默认字体名或 TTF 字体文件名
    fontSize                                     //字体号
)
```

下面将图 14-2 所示的场景中“Hello World”文字,使用 Cocos2d-x Lua API GUI 中 Text 类实现。修改 GameScene.lua 代码片段如下:

```
function GameScene:createLayer()
    ...
    -- 创建 Text 对象
    local label = ccui.Text:create("Hello World", "fonts/Marker Felt.ttf", 24)          ①
    -- 设置文本颜色
    label:setColor(cc.c3b(159, 168, 176))
    label:setPosition(size.width/2, size.height - label:getContentSize().height)
    layer:addChild(label)
    ...
    return true;
}
```

上述第①行代码是创建 Text 对象,然后设置位置,并将 Text 添加到当前场景。

14.3.2 TextBMFont

TextBMFont 控件是通过位图字体显示文本。TextBMFont 类中一个主要的创建函数 create 定义如下:

```
ccui.Text:create(text,                           //显示的文本内容
    filename                                     //位图字体字符坐标文件
)
```

下面将图 14-2 所示的场景中“Hello World”文字,使用 Cocos2d-x Lua API GUI 中 TextBMFont 类实现。修改 GameScene.lua 代码片段如下:

```
function GameScene:createLayer()
    ...
    -- 创建 TextBMFont 对象
```

```
local textBMFont = ccui.TextBMFont:create("Hello World", "fonts/BMFont.fnt")          ①
--缩放因子
local factor = 0.8
--设置缩放
textBMFont:setScale(factor)
--设置文本颜色
textBMFont:setPosition(size.width / 2,
                size.height - textBMFont:getContentSize().height * factor)          ②
layer:addChild(textBMFont, 1)
…
return true;
}
```

上述第①行代码是创建 TextBMFont 对象，由于本例中位图字体比较大，需要缩小后再放到场景中，语句 textBMFont:setScale(factor)用来缩放显示文本，factor 是缩放因子，小于 1.0 就是缩小，大于 1.0 就是放大。另外，缩放文本会影响它的高和宽，因此在第②行代码中计算文本高度时也需要乘以缩放因子 factor。

实例运行结果如图 14-3 所示。

图 14-3 实例运行结果

14.3.3 RichText

RichText 是富文本控件，所谓富文本控件就是可以显示文本、图片等超文本，还可以设置文本的字体、颜色和链接等。此外，可以自定义 RichText 控件的内容。

RichText 控件中可以放置富文本元素 RichElement 对象，RichElement 是一个抽象类，它有四个子类：RichElementText（文本元素）、RichElementImage（图片元素）、RichElementCustomNode(自定义元素)和 RichElementNewLine(换行元素)。RichElement 类图如图 14-4 所示。

RichText 控件提供了如下函数管理富文本元素：

(1) insertElement(element, index)：插入元素。

(2) pushBackElement (element)：在最后元素后面追加元素。

(3) removeElement (index)：根据索引移除元素。

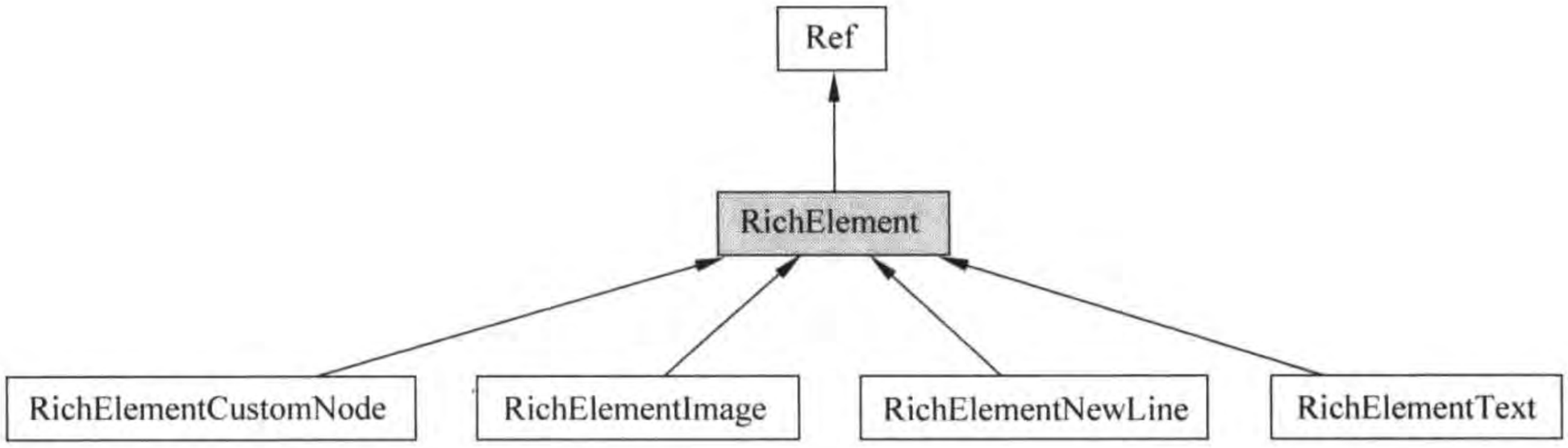


图 14-4　RichElement 类图

(4) removeElement (element)：删除元素。

下面通过一个实例介绍 RichText 控件的使用，这个实例运行后的场景如图 14-5 所示。

图 14-5　RichText 控件实例

修改 GameScene.lua 代码片段如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()
    ...

    -- 创建 RichText 对象
    local richText = ccui.RichText:create()
    -- 设置是否忽略用户定义的内容大小
    richText:ignoreContentAdaptWithSize(false)
    -- 设置内容大小
    richText:setContentSize(cc.size(200, 200))

    -- 创建文本类型的 RichElement 对象
    local re1 = ccui.RichElementText:create(1, cc.c3b(0, 255, 0), 255,
                                "Hello World", "fonts/Marker Felt.ttf", 20)          ①
    local re2 = ccui.RichElementText:create(2, cc.c3b(255, 0, 0), 255,
                                "Last one is red ", "Helvetica", 20)                 ②
    -- 创建图片类型的 RichElement 对象
    local re3 = ccui.RichElementImage:create(3, cc.c3b(255, 0, 0), 255, "CloseSelected.png") ③
```

```
--创建换行 RichElement 对象
local newLine = ccui.RichElementNewLine:create(77, cc.c3b(255, 255, 255), 255)      ④

richText:pushBackElement(re1)
richText:pushBackElement(newLine)                                                    ⑤
richText:pushBackElement(re2)
richText:pushBackElement(re3)
richText:insertElement(newLine, 3)                                                   ⑥
richText:setPosition(size.width / 2, size.height / 2)

layer:addChild(richText)

    return layer
end
```

上述第①和第②行代码是创建文本类型的 RichElementText 对象，RichElementText 类中一个主要的创建函数 create 定义如下：

```
ccui.RichElementText:create(tag,        //设置一个 tag(标签),tag 是一个整数
        color,                          //设置颜色
        opacity,                        //设置不透明度,0 表示完全透明,255 表示完全不透明
        text,                           //显示的文本
        fontName,                       //文本的系统字体或 TTF 文件名
        fontSize,                       //文本的大小
        flags = 0,                      //文本标识,描边、加粗等,可以省略
        url = ""                        //设置超文本链接,可以省略
)
```

第③行代码是创建图片类型的 RichElementImage 对象，RichElementImage 类中一个主要的创建函数 create 定义如下：

```
ccui.RichElementImage:create (tag,      //设置一个 tag(标签)
        color,                          //设置颜色
        opacity,                        //设置不透明度
        filePath                        //设置图片路径
)
```

第④行代码是创建一个换行 RichElementNewLine 对象，使用该对象是为了在多个富文本之间换行，见第⑤和第⑥行代码。RichElementNewLine 类中一个主要的创建函数 create 定义如下：

```
ccui.RichElementNewLine:create(tag,
        color,
        opacity
)
```

14.4　RadioButton 和 RadioButtonGroup

RadioButton 一般称为“单选按钮”、“收音机按钮”，多个 RadioButton 对象具有互斥性，前提是这些 RadioButton 对象是由一个 RadioButtonGroup 对象管理的，但 RadioButtonGroup 并

不是 RadioButton 父容器，RadioButtonGroup 对象只是管理一组 RadioButton 对象。

下面通过一个实例介绍 RadioButton 和 RadioButtonGroup 控件的使用，这个实例运行后的场景如图 14-6 所示，场景会动态创建一组 RadioButton 对象，这组 RadioButton 是垂直居中。

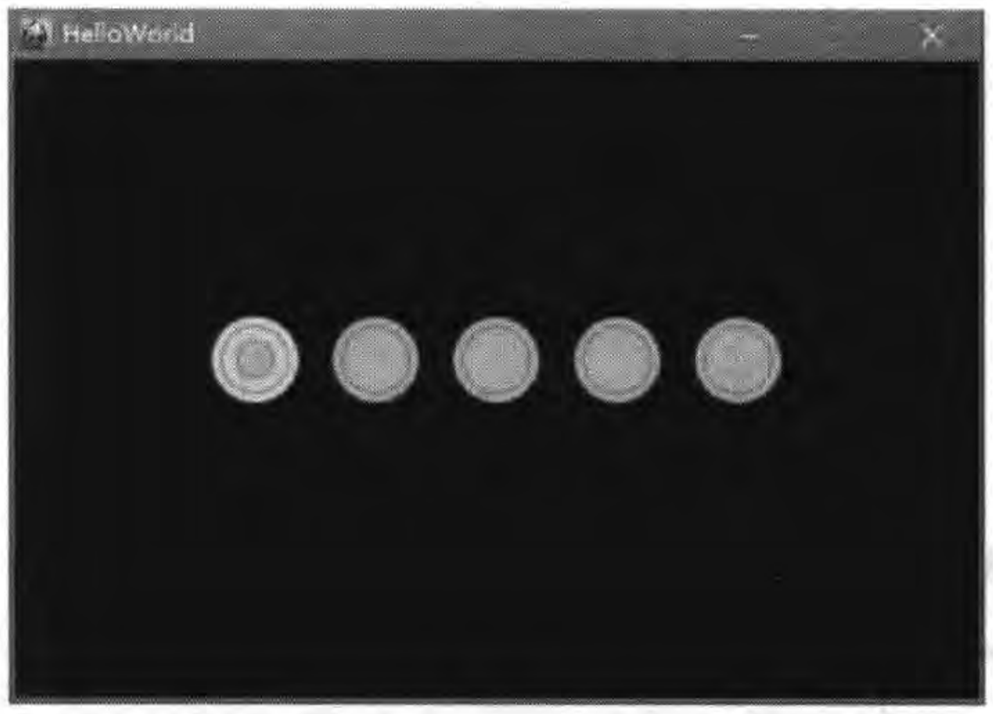


图 14-6　RadioButton 和 RadioButtonGroup 控件实例

修改 GameScene.lua 代码片段如下：

```
-- 创建层
function GameScene:createLayer()

    local layer = cc.Layer:create()

    -- RadioButton 单击事件回调函数
    local function onChangedRadioButtonGroup(sender, index, eventType)
        cclog("RadioButton" .. index .. " Clicked")
    end

    -- 创建 RadioButtonGroup 对象
    local radioButtonGroup = ccui.RadioButtonGroup:create()                          ①
    layer:addChild(radioButtonGroup)

    -- 创建 RadioButton
    local NUMBER_BUTTONS = 5
    local BUTTON_WIDTH   = 60
    local startPosX = size.width / 2.0 - ((NUMBER_BUTTONS - 1) / 2.0) * BUTTON_WIDTH   ②
    for i = 0, NUMBER_BUTTONS - 1 do
        local radioButton = ccui.RadioButton:create("icon/btn_radio_off_holo.png",
                                                    "icon/btn_radio_on_holo.png")     ③
        local posX = startPosX + BUTTON_WIDTH * i
        radioButton:setPosition(posX, size.height / 2 + 10)
        radioButtonGroup:addRadioButton(radioButton)                                 ④
        radioButtonGroup:addEventListener(onChangedRadioButtonGroup)                 ⑤

        layer:addChild(radioButton)                                                  ⑥
    end

    return layer
end
```

上述第①行代码是创建 RadioButtonGroup 对象，需要注意的是 RadioButtonGroup 对象也必须通过 layer:addChild(radioButtonGroup)语句添加到当前层中。

第②行代码是计算左边第一个 RadioButton 对象的位置，其中 BUTTON_WIDTH 是 RadioButton 的宽度，即两个 RadioButton 中心点之间的距离。

第③行代码是循环创建 RadioButton 对象，其中第一个参数是设置 RadioButton 抬起(或未选中)时纹理，第二个参数是设置 RadioButton 按下(或选中)时的纹理。

为了能够让 RadioButtonGroup 一直管理 RadioButton，则需要通过第④行代码 radioButtonGroup:addRadioButton(radioButton)函数将 RadioButton 添加到 RadioButtonGroup 中。

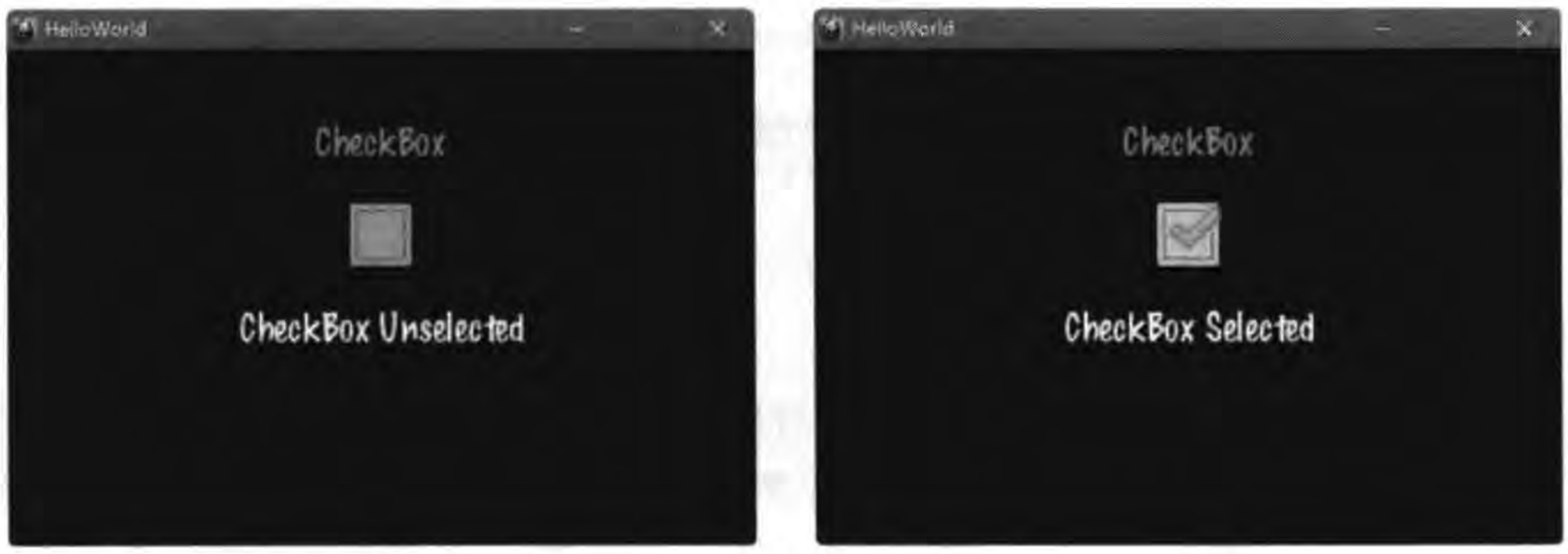
RadioButton 和 RadioButtonGroup 都可以通过 addEventListener 函数添加事件监听器，如果只关注单个 RadioButton 对象的事件处理，则调用 RadioButton 对象的 addEventListener 函数添加事件监听器。如果关注的是同一组相关的 RadioButton 对象的事件处理，则需要调用 RadioButtonGroup 对象的 addEventListener 函数添加事件监听器。本例中是为 RadioButtonGroup 添加事件监听器。

第⑥行代码 layer:addChild(radioButton)是将 RadioButton 对象添加到当前层中，这时需要注意的不仅仅是 RadioButton 对象，RadioButtonGroup 对象也需要添加到当前层中。

14.5　CheckBox

CheckBox 一般称为“复选框”、“多选按钮”，多个 CheckBox 可以给用户多个选择的可能性，而单个 CheckBox 相当于开关控件，给用户提供了两种状态切换的可能性。

下面通过一个实例介绍 CheckBox 控件的使用，这个实例运行后的场景如图 14-7 所示，场景中只有一个 CheckBox 控件，它有两种状态：选中和未选中。



(a)　　(b)

图 14-7　CheckBox 控件实例

修改 GameScene.lua 代码片段如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    -- CheckBox 单击事件回调函数
    local function onChangedCheckBox(sender, eventType)                    ①
        cclog(eventType)
        if eventType == ccui.CheckBoxEventType.selected then
            label2:setString("CheckBox Selected")
        elseif eventType == ccui.CheckBoxEventType.unselected then
            label2:setString("CheckBox Unselected")
        end
    end

    -- 创建 Text 对象,显示静态文本 CheckBox
    local label1 = ccui.Text:create("CheckBox", "fonts/Marker Felt.ttf", 24)    ②
    -- 设置文本颜色
    label1:setColor(cc.c3b(159, 168, 176))
    label1:setPosition(size.width / 2, size.height - 60)
    layer:addChild(label1)

    -- 创建 CheckBox 对象
    local ckb = ccui.CheckBox:create("icon/btn_check_off_holo.png",
                                     "icon/btn_check_on_holo.png")          ③
    -- 设置 CheckBox 位置
    local posX, posY = label1:getPosition()
    ckb:setPosition(cc.p(posX, posY - 60))
    -- 添加事件监听器
    ckb:addEventListener(onChangedCheckBox)                                ④
    layer:addChild(ckb)

    -- 创建 Text 对象,显示 CheckBox 选中状态
    label2 = ccui.Text:create("CheckBox Unselected", "fonts/Marker Felt.ttf", 24)    ⑤
    posX, posY = ckb:getPosition()
    label2:setPosition(cc.p(posX, posY - 60))
    layer:addChild(label2)

    return layer
end
```

上述第②和第⑤行代码都是创建 Text 对象,第②行的 label1 是局部对象用于显示静态文本“CheckBox”,而第⑤行的 label2 是成员变量用于动态显示 CheckBox 对象的状态。

第③行代码是创建 CheckBox 对象,其中第一个参数是设置 CheckBox 未选中时的纹理,第二个参数是设置 CheckBox 选中时的纹理。第④行代码是为 CheckBox 对象添加事件监听器。

第①行代码 onChangedCheckBox 函数是 CheckBox 单击事件回调函数。

14.6　LoadingBar

LoadingBar是加载进度栏，一般应用于耗时的等待，消除用户的心理等待时间。LoadingBar没有事件处理能力，LoadingBar的取值是0.0～100.0之间的float类型浮点数。

下面通过一个实例介绍LoadingBar控件使用，这个实例运行后的场景如图14-8所示，场景中只有一个LoadingBar控件，LoadingBar控件处于活动状态，它的进度从0开始达到100后再重新开始。

(a)　(b)

图14-8　LoadingBar控件实例

修改GameScene.lua代码片段如下：

```
-- 创建层
function GameScene:createLayer()

    local layer = cc.Layer:create()

    --创建 LoadingBar
    loadingBar = ccui.LoadingBar:create("progressbar.png")                    ①
    loadingBar:setPosition(size.width / 2, size.height / 2 + loadingBar:getContentSize().
height / 2.0)
    loadingBar:setDirection(ccui.LoadingBarDirection.LEFT)                    ②

    layer:addChild(loadingBar)
    local count = 0

    local function update(dt)                                                 ③
        count = count + 1
        if count > 100 then
            count = 0
        end
        loadingBar:setPercent(count)                                          ④
```

```
        end
        layer:scheduleUpdateWithPriorityLua(update,0)                    ⑤

        return layer
    end
```

上述第①行代码是创建 LoadingBar 对象，参数 progressbar.png 是 LoadingBar 显示的纹理图片，默认情况下图片的长度则是 LoadingBar 的长度，图片的宽度则是 LoadingBar 的宽度。

第②行代码是 LoadingBar 加载的方向，它有两个常量：LEFT(从左到右)和 RIGHT(从右到左)，默认值是 LEFT。

第⑤行代码 layer:scheduleUpdateWithPriorityLua(update,0)语句是启动游戏调度函数，它将在每一帧(1/60 秒)调用一次第③行代码的 update 函数，在 update 函数中更新 LoadingBar 的值，第④行代码 loadingBar:setPercent(count)语句是设置 LoadingBar 的数值。

14.7 滑块控件

滑块控件是由一个滑杆和一个滑动按钮构成，玩家可以通过手指按住滑动按钮拖曳，来改变滑块控件数值。在游戏设置场景中经常会用到滑块控件设置“英雄”状态等。Cocos2d-x Lua API GUI 滑块类是 Silder。

滑块是一个相对有点复杂的控件，为了了解它的一些基本概念，请查看图 14-9 所示的示意图，从图中可见一个滑块控件基本上由三部分组成：滑块进度栏(ProgressBar)、滑块按钮(SlidBall)和滑杆(Bar)。

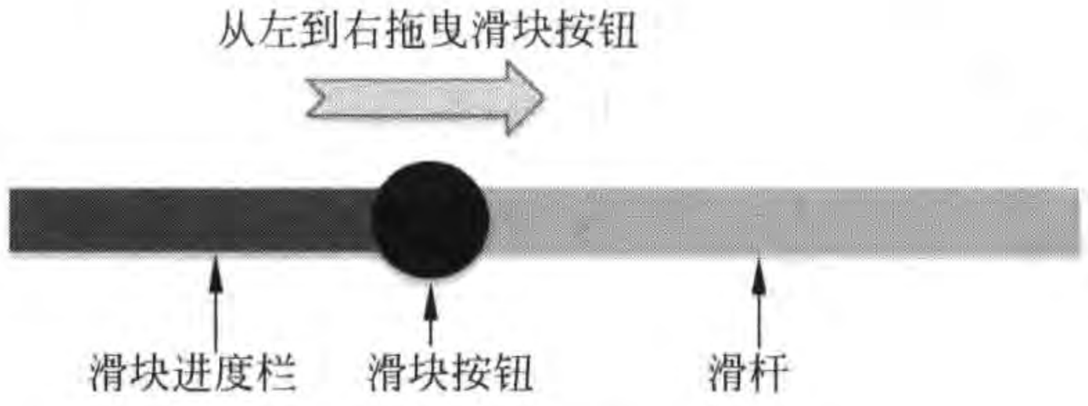


图 14-9 滑块控件示意图

另外，滑块的取值默认是 0.0～100.0 之间的 float 类型浮点数，但可以通过 setMaxPercent(int percent)函数设置最大值。

提示 Percent 虽然是百分数，但是通过设置 MaxPercent 可以超过 100 或小于 100。

下面通过一个实例介绍滑块控件的使用，这个实例运行后的场景如图 14-10 所示，场景中只有一个滑块控件，当用手指拖曳滑块按钮时，它的值将显示在下面的文本控件上，取值

范围是 0～1000。

图 14-10　滑块控件实例

修改 GameScene.lua 代码片段如下：

```
local slider
local label2
…
-- 创建层
function GameScene:createLayer()

    local layer = cc.Layer:create()

-- Slider 滑动事件回调函数
local function onChangedSlider(sender, eventType)                                    ①
    if eventType == ccui.SliderEventType.percentChanged then                         ②
        local percent = slider:getPercent()
        label2:setString(percent)
    end
end

--创建 Text 对象,显示静态文本 Slider
local label1 = ccui.Text:create("Slider", "fonts/Marker Felt.ttf", 24)
--设置文本颜色
label1:setColor(cc.c3b(159, 168, 176))
label1:setPosition(size.width / 2, size.height - 100)
layer:addChild(label1)

--创建滑块控件
slider = ccui.Slider:create()                                                        ③
--加载滑杆纹理
slider:loadBarTexture("sliderTrack.png")                                             ④
--加载滑块按钮纹理
slider:loadSlidBallTextures("sliderThumb.png", "sliderThumb.png", "")                ⑤
--加载滑块进度栏纹理
slider:loadProgressBarTexture("sliderProgress.png")                                  ⑥
--The max percent of Slider.
slider:setMaxPercent(1000)                                                           ⑦
slider:setPosition(size.width / 2.0, size.height / 2.0)
```

```
    slider:addEventListener(onChangedSlider)                                    ⑧
    layer:addChild(slider)

    -- 创建 Text 对象,显示 CheckBox 选中状态
    label2 = ccui.Text:create("0", "fonts/Marker Felt.ttf", 24)
    posX, posY = slider:getPosition()
    label2:setPosition(cc.p(posX, posY - 100))
    layer:addChild(label2)

    return layer
end
```

上述第③行代码是创建 Slider 对象。第④行代码 loadBarTexture 函数是加载滑杆纹理。第⑤行代码 loadSlidBallTextures 函数是加载滑块按钮纹理,其中第一个参数是正常显示时的纹理,第二个参数是按下时的纹理,第三个参数是不可用时的纹理。第⑥行代码 loadProgressBarTexture 函数是加载滑块进度栏纹理。

第⑦行代码是设置滑块的最大值,可以超过 100。

第⑧行代码是为滑块对象添加事件监听器,使用的函数是 addEventListener 函数。第①行代码是回调函数,其中第一个参数是滑块对象,第二个参数是事件类型。第②行是判断如果是滑块值改变事件,则获得滑块值并显示在 label2 文本对象中。ccui. SliderEventType. percentChanged 是滑块值改变事件。

本章小结

在本章中重点介绍了 Button、Text、TextBMFont、RichText、RadioButton、RadioButtonGroup、CheckBox、LoadingBar 和 Silder 等 Cocos2d-x Lua API GUI 控件。

第 15 章

Cocos2d-x 中的 3D 特性

VR(virtual reality,即虚拟现实,简称 VR)近年来越来越火爆,也进一步推动了 3D 游戏增长。Cocos2d-x 在 3.x 版本中逐步增加了一些 3D 特性,现在使用 Cocos2d-x 3.x 引擎已经可以开发 3D 游戏了。本章介绍 Cocos2d-x 中的 3D 特性。

15.1 一些 3D 概念

3D 游戏与 2D 游戏有很大的差别,3D 游戏增加了很多的概念,如模型、相机、视点、视口、投影、视景体、材质和灯光。

15.1.1 网格和模型

3D 世界由点、线、面构成,它们被称为“图元”,在移动平台中面一般就是一个三角形,多个三角形可以构成一个“网格”Mesh。而多个网格构成“模型”Model,如图 15-1 所示海豚模型。

从图 15-1 中可见海豚模型是由很多网格平面构成的,所以模型也是由很多顶点构成的,顶点越多模型越逼真细腻,当然渲染时也是非常耗费 CPU 或 GPU 资源。

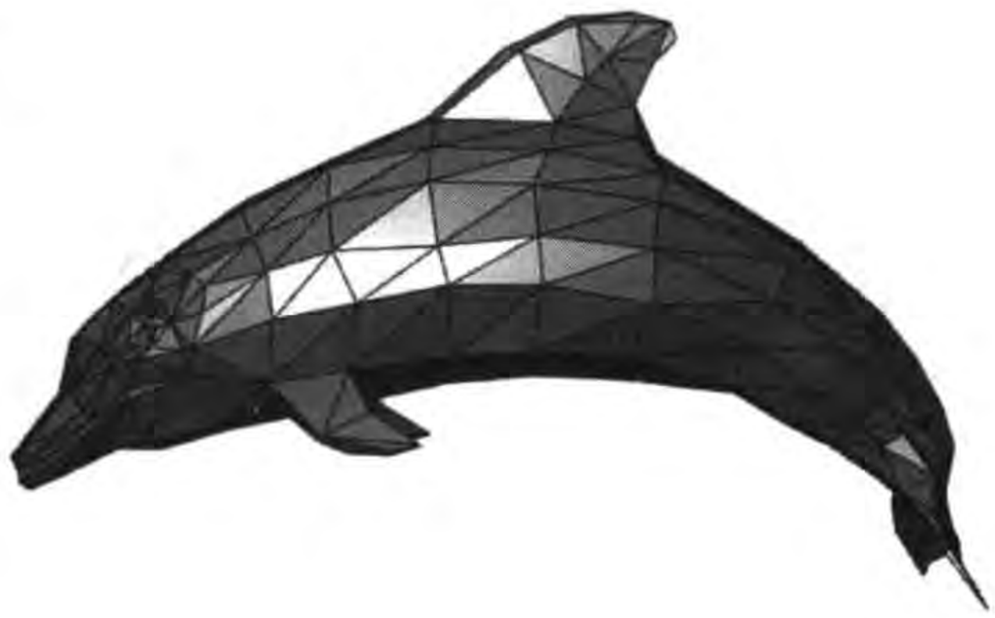


图 15-1 海豚模型

15.1.2 相机

3D 世界需要从一个视点去看它,就相当于观察者的眼睛所在的位置。Cocos2d-x 中视点是通过“相机”Camera 表示的。可以设置相机位置、朝向和竖直方向向量,默认情况下相机位于原点,镜头指向 Z 轴负方向,竖直方向向量(0,1,0),如图 15-2 所示。

15.1.3 投影

投影决定了模型如何投射到屏幕上,Cocos2d-x 提供了两种基本投影方式:透视投影和

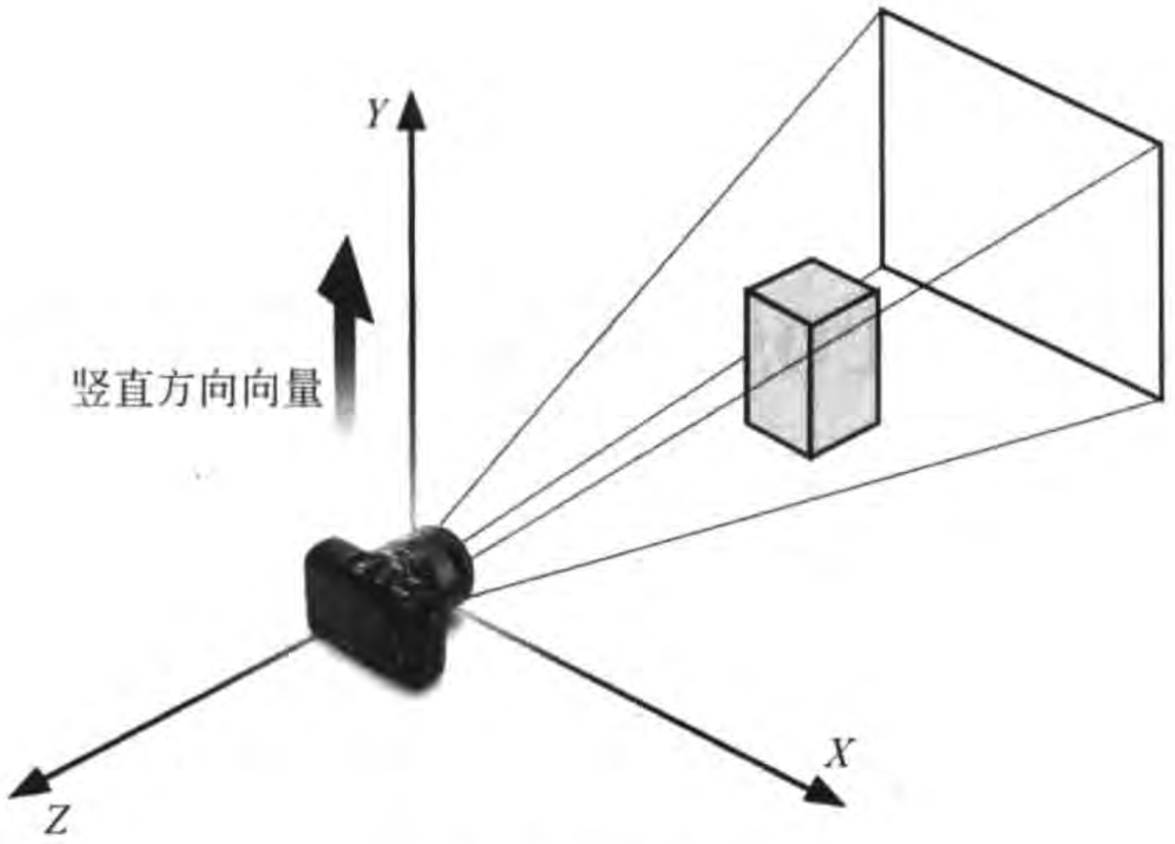


图 15-2　竖直方向向量

正交投影。透视投影是类似于人看东西的效果，即近大远小的透视效果，例如铁道上的两根铁轨会在远处交汇成一点。制作真实的 3D 场景，需要使用透视投影。而正交投影是将模型直接映射到屏幕上，并不影响它们的相对大小，正交投影常用于建筑制图、机械制图和 CAD 软件中。

在透视投影中视景体[①]是一个棱锥台(类似于将金字塔顶部截掉)，如图 15-3 所示，图中标示了透视投影视景体中的一些概念：近裁剪面距离、远裁剪面距离、视野夹角和高宽比，这些概念从图中可见含义，这里不再赘述。

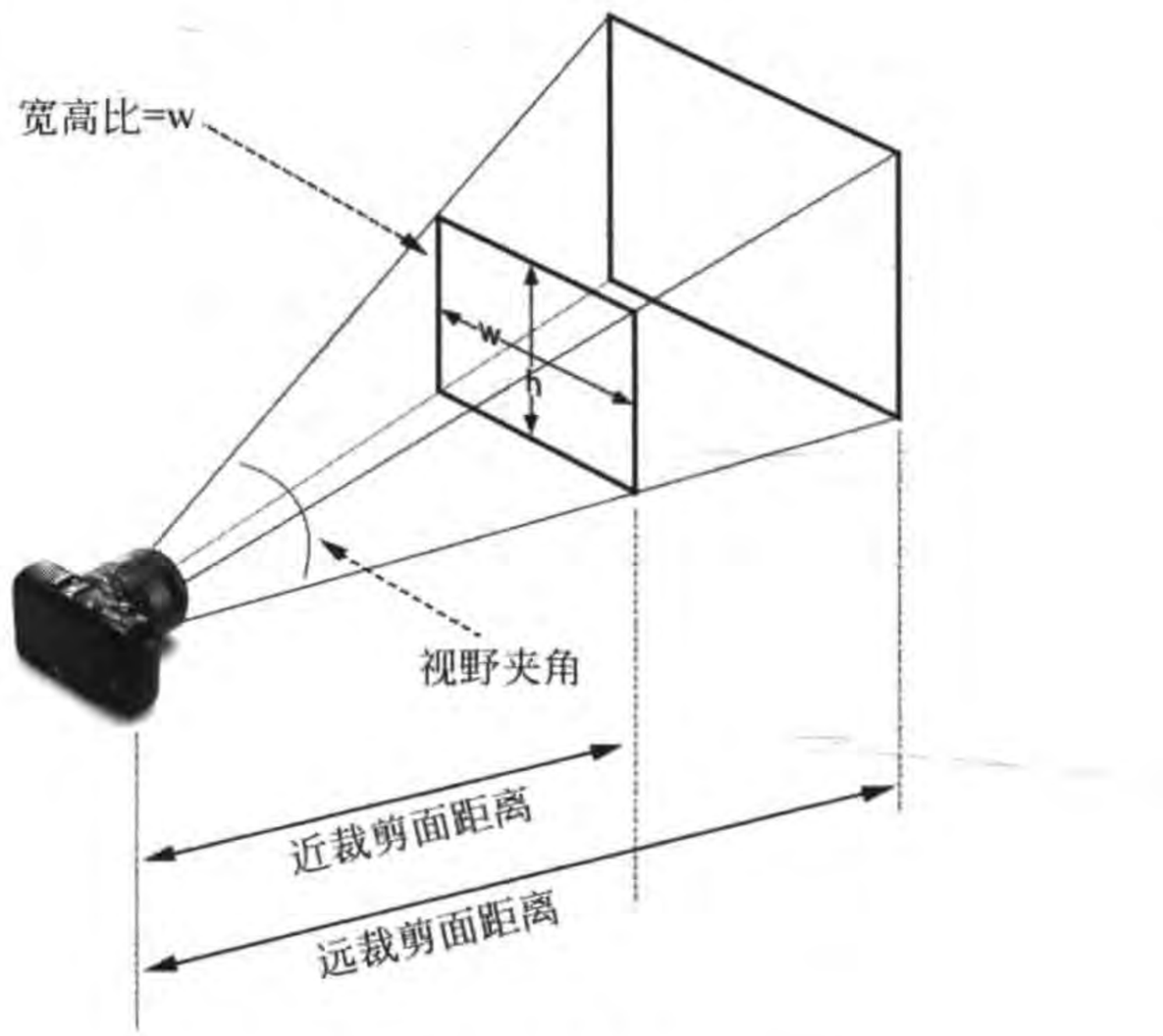


图 15-3　透视投影视景体

① 视景体决定了模型如何被投影到屏幕上，还决定了视野的范围，即哪些模型或模型的哪些部分可见。

在正交投影中，视景体是一个平行六面体，如图 15-4 所示。与透视投影不同，正交投影使用的视景体两端大小相同，因此离相机的距离不会影响模型在图像中的大小。

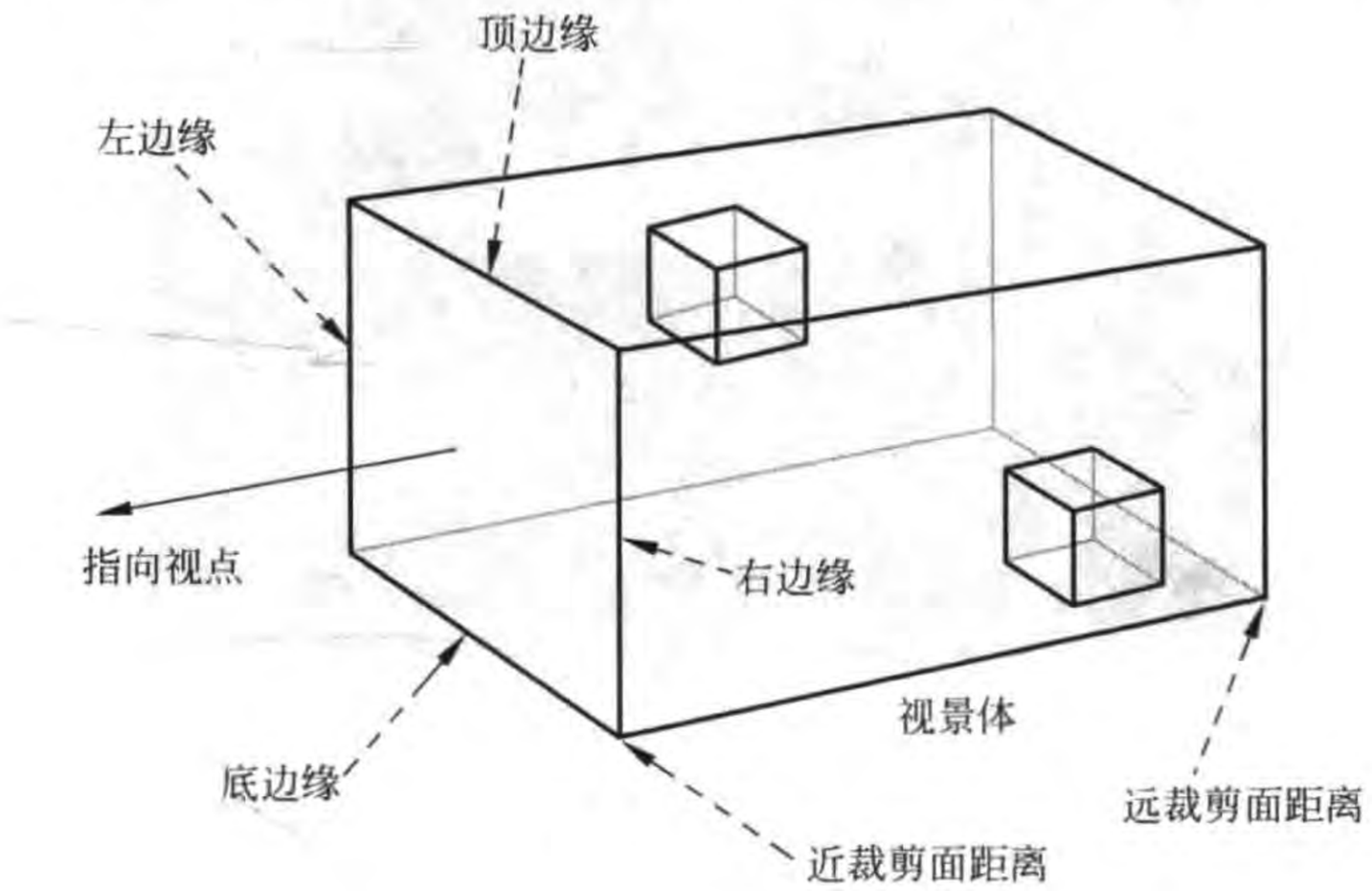


图 15-4　正交投影视景体

15.2　使用 3D 精灵

在 Cocos2d-x 中 3D 精灵类与 2D 精灵类完全不同，也非父子关系。3D 精灵类是 Sprite3D，Sprite3D 类图如图 15-5 所示，从图中可见 Sprite3D 是 Node 直接子类，而非继承自 2D 精灵类 Sprite。

15.2.1　创建 Sprite3D 对象

创建 Sprite3D 对象有多种方式，其中常用的函数如下：

（1）cc. Sprite3D: create( )：创建一个无模型和纹理的 Sprite3D 对象，模型和纹理等属性需要在创建后设置。

（2）cc. Sprite3D: create (modelPath)：指定模型创建 Sprite3D 对象。

（3）cc. Sprite3D:create(modelPath,texturePath)：指定模型和纹理创建 Sprite3D 对象。

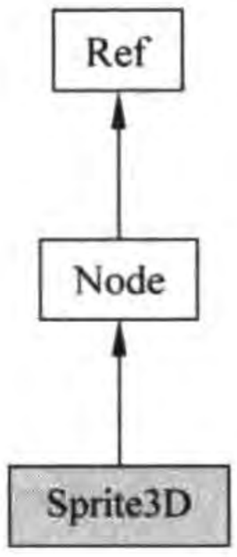


图 15-5　Sprite3D 类图

15.2.2　实例：使用模型和纹理 Sprite3D 对象

模型能够描述 3D 世界中的物体所占用的空间，纹理是在模型上进行贴图，是物体的“皮肤”，所以 3D 世界中物体至少应该被设置模型和纹理两个属性。

下面通过一个实例介绍如何创建 Sprite3D 对象，实例运行情况如图 15-6 所示，当手指在屏幕上按下时飞机开始旋转，当抬起手指时飞机停止旋转。

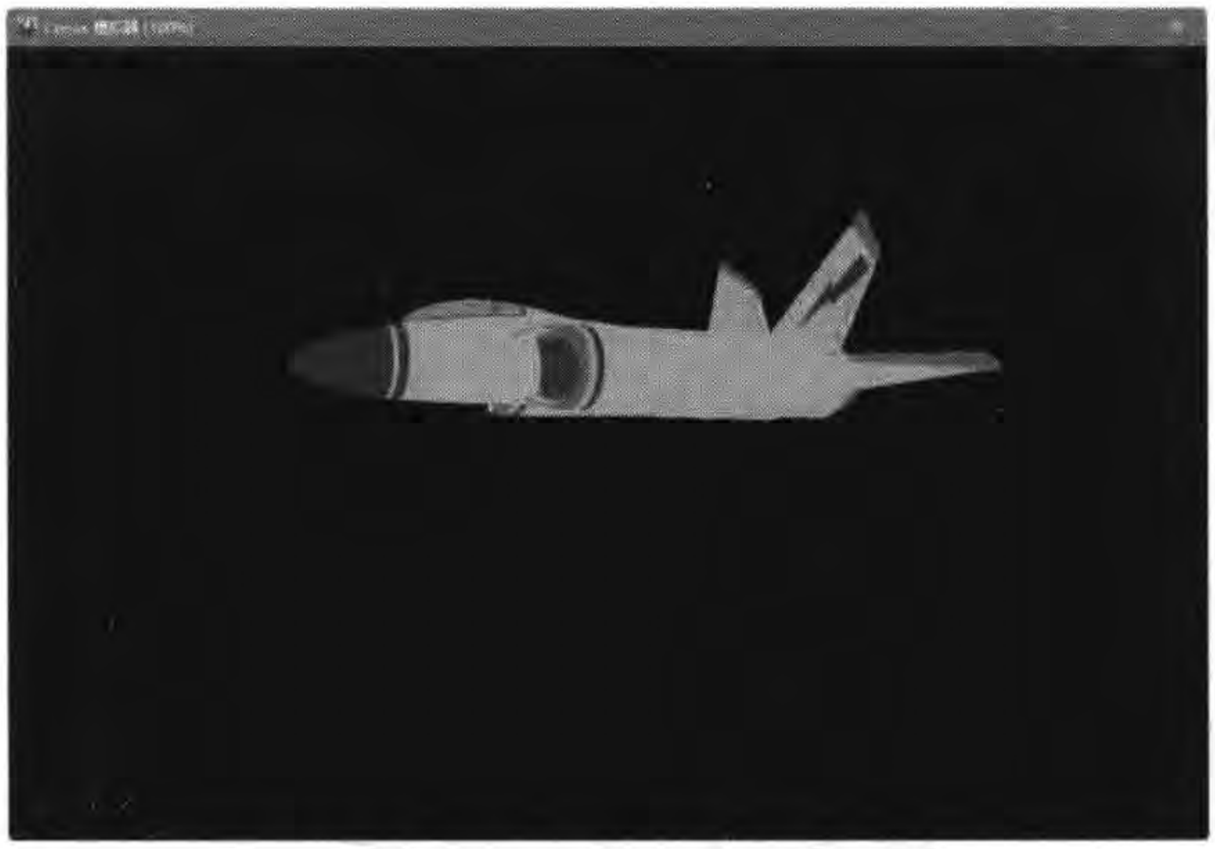


图 15-6　创建 Sprite3D 对象实例

实现该实例需要添加触摸事件、游戏调度函数等，这些技术同样适用于 Cocos2d-x 的 3D 游戏场景，这里不再赘述。

GameScene.lua 的 GameScene:createLayer()函数中初始化场景相关代码如下：

```
function GameScene:createLayer()
    local layer = cc.Layer:create()
    local bg = cc.LayerColor:create()
    layer:addChild(bg)

    -- 创建 Sprite3D 精灵对象
    local sprite = cc.Sprite3D:create("3D/ship.c3b")                    ①
    -- sprite:setTexture("3D/ship.png")                                 ②
    sprite:setScale(10)                                                 ③
    sprite:setPosition(cc.p(size.width / 2, size.height / 2))           ④
    layer:addChild(sprite)                                              ⑤

    ...

    return layer
end
```

其中，第①行代码 Sprite3D:create("3D/ship.c3b")是通过模型文件 ship.c3b 创建 Sprite3D 精灵对象，ship.c3b 模型文件是 Cocos2d-x 专用的二进制模型文件，关于模型格式问题下一节再详细介绍。一般情况下在模型中指定纹理文件信息，但是如果纹理文件路径有误，这需要另外设置纹理文件路径，见第②行代码-sprite:setTexture("3D/ship.png")语句，也可以通过 cc.Sprite3D:create(modelPath,texturePath)函数一并指定模型文件和纹理文件路径。

第③～⑤行代码的几个函数在 Cocos2d-x 的 2D 游戏场景中常用，同样适用于 3D 游戏

场景。其中第③行代码 sprite:setScale(10)语句是放大模型，第④行代码是设置位置，第⑤行代码是将 Sprite3D 精灵对象添加到当前场景中。

GameScene.lua 的 GameScene:createLayer()函数中游戏调度 update(delta)函数代码如下：

```
-- 游戏循环调度函数
 local function update(delta)
     local rotation3D = sprite:getRotation3D()                    ①
     rotation3D.y = rotation3D.y + 3                              ②
     sprite:setRotation3D(rotation3D)                             ③
 end
```

模型的旋转是在 update(delta)函数中实现的，其中第①行代码获得 Sprite3D 精灵当前的 3D 旋转角度，它是个三维向量，有 x、y、z 三个属性，分别表示 x、y、z 轴旋转角度。由于本例只是沿 y 轴，所以可以通过第②行代码的 rotation3D.y = rotation3D.y + 3 语句改变 y 轴旋转角度。第③行代码是设置精灵旋转角度。

GameScene.lua 的 GameScene:createLayer()函数中触摸事件相关代码如下：

```
local function touchBegan(touch, event)
  cclog("touchBegan")
    layer:scheduleUpdateWithPriorityLua(update, 0)               ①
    return true
end

local function touchEnded(touch, event)
    cclog("touchEnded")
  layer:unscheduleUpdate()                                       ②
end

  -- 创建一个事件监听器 OneByOne 为单点触摸
  local listener = cc.EventListenerTouchOneByOne:create()
  -- 设置是否吞没事件，在 EVENT_TOUCH_BEGAN 事件返回 true 时吞没
  listener:setSwallowTouches(true)
  -- EVENT_TOUCH_BEGAN 事件回调函数
  listener:registerScriptHandler(touchBegan, cc.Handler.EVENT_TOUCH_BEGAN)   ③
  -- EVENT_TOUCH_ENDED 事件回调函数
  listener:registerScriptHandler(touchEnded, cc.Handler.EVENT_TOUCH_ENDED)   ④

  local eventDispatcher = self:getEventDispatcher()
  -- 添加监听器
  eventDispatcher:addEventListenerWithSceneGraphPriority(listener, layer)
```

touchBegan 是在手指按下时回调的函数，其中第①行代码 layer:scheduleUpdateWithPriorityLua(update, 0)是开始游戏调度，它会按照游戏循环周期调用 update(delta)函数。

而停止游戏调度是在 touchEnded 函数中，第②行代码 layer:unscheduleUpdate()是结束游戏调度，touchEnded 是手指抬起时回调的函数。

15.3 3D 模型文件格式

Cocos2d-x 引擎目前只能读取两种格式的 3D 模型文件：

(1) 通用的格式 obj 文件。

(2) Cocos2d-x 专用格式 c3t 和 c3b 文件。

obj 文件是 Alias|Wavefront 公司提出的 3D 模型文件格式，很适合用于 3D 软件模型之间的互导，obj 文件格式是一种应用十分广泛的格式，目前几乎所有知名的 3D 编辑器都支持 obj 文件的读写，而且它能够很简单地被解析。obj 文件是一种文本文件，可以直接用写字板打开进行查看和编辑修改。但是，它也有局限性，如不包含动画、材质、纹理路径和粒子系统等信息。

如果直接使用 obj 格式，则需要同时指定模型和纹理路径，示例代码如下：

```
local sprite = cc.Sprite3D:create("3D/boss.obj")
sprite:setTexture("3D/boss.png")
```

或

```
local sprite = cc.Sprite3D:create("3D/boss.obj", "3D/boss.png")
```

c3t 和 c3b 是 Cocos2d-x 专用的文件格式，c3t 是文本格式的 Cocos2d-x 3.x(具体是 3.3 之后版本)3D 模型文件，c3b 是二进制格式的 Cocos2d-x 3.x 之后 3D 模型文件。一般情况下 c3t 用于开发调试阶段，而 c3b 用于游戏发布版本，因为 c3b 很小，访问比较高效。

c3t 和 c3b 模型文件中至少需要包含一个纹理路径，而且目前只支持一个骨骼动画，多个骨骼动画现在还不支持，另外网格最多支持的顶点数是 32 767。

3D 模型文件中除了 obj，还有 fbx 和 dae 等文件格式。

(1) fbx 是一种封闭的二进制模型文件格式，是由 Autodesk 公司出品的一款用于跨平台的免费三维创作与交换格式的软件，通过 fbx 用户能访问大多数三维供应商的三维文件，fbx 文件格式支持所有主要的三维数据元素以及二维音频和视频媒体元素。

(2) dae 是开发的 3D 交互文件格式，一般用于多个图形程序之间交换数字数据。

遗憾的是 Cocos2d-x 不支持 fbx 和 dae 文件格式。Cocos2d-x 提供一个 fbx-conv 工具可以将 fbx 和 dae 等 3D 模型文件转化成 Cocos2d-x 能够访问的文件格式。fbx-conv 命令行如下：

```
fbx-conv [-a|-b|-t] FBXFile
```

可供选择使用的参数如下：

(1) -?：显示帮助文档。

(2) -a：导出文本格式和二进制格式。

(3) -b：导出二进制格式。

(4) -t：导出文本格式。

示例如下：

```
fbx-conv -a ship.dae
```

-a 参数会在当前目录下生成 ship.c3b 和 ship.c3t 两个文件。

15.4　使用相机

在 Cocos2d-x 中相机类是 Camera，Camera 类图如图 15-7 所示，从图中可见 Camera 是 Node 直接子类。

15.4.1　创建和设置 Camera 对象

创建 Camera 对象有多种方式，其中常用的函数如下：

(1) cc.Camera:create()：创建一个 Camera 对象，相机的类型依赖于 Director 的投影类型，通过 cc.Director:getProjection 函数获得 Director 的投影类型。

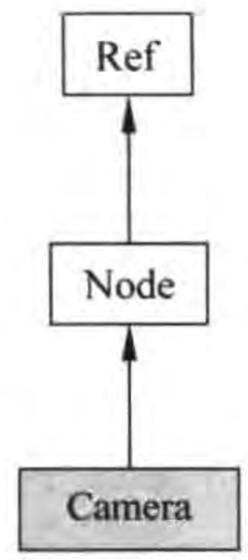


图 15-7　Camera 类图

(2) cc.Camera:createPerspective (fieldOfView, aspectRatio, nearPlane, farPlane)。创建一个透视投影 Camera 对象，其中参数 fieldOfView 是视野夹角，aspectRatio 是宽高比，nearPlane 是近裁剪面距离，farPlane 是远裁剪面距离。

(3) cc.Camera:createOrthographic(zoomX, zoomY, nearPlane, farPlane)。创建一个正交投影 Camera 对象，其中参数 zoomX 是沿 X 轴放大因子，zoomY 是沿 Y 轴放大因子，nearPlane 是近裁剪面距离，farPlane 是远裁剪面距离。

创建好 Camera 对象之后，还需要对 Camera 对象进行设置，其中重要的属性有位置、朝向和竖直方向向量，设置相机的位置通过 setPosition3D(position)函数实现；设置相机的朝向和竖直方向向量属性是通过 Camera 的如下函数实现的：

```
local lookAt  ( local  target,          // target 是相机朝向坐标点
  local  up = Vec3::UNIT_Y              // up 是竖直方向向量
)
```

up 可以省略，默认值是常量 Vec3::UNIT_Y，Vec3::UNIT_Y 取值是(0,1,0)，相当于相机的顶部沿 Y 轴向上，相机镜头指向 Z 轴相反方向，如图 15-2 所示。

15.4.2　实例：使用 Camera 对象

下面通过一个实例介绍如何创建 Camera 对象。事实上在 3D 场景中会有一个默认相机，它的位置是在坐标原点(0,0,0)，默认相机看到的 3D 场景物体，如图 15-6 所示的场景中看到的飞机，没有俯视角。

该实例运行情况如图 15-8 所示，看到的飞机可以有一定的俯视角度。

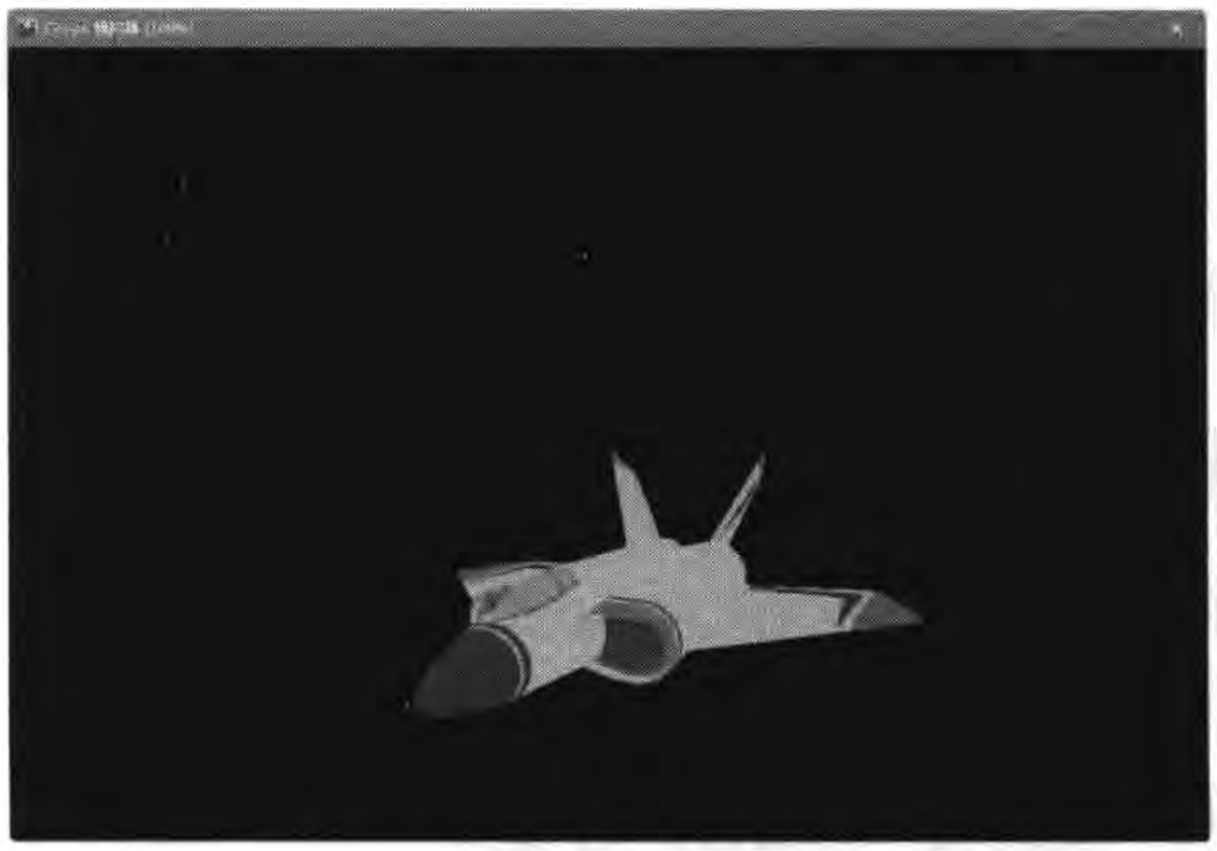


图 15-8　创建 Camera 对象实例

GameScene.lua 的 GameScene:createLayer()函数相关代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()
    local bg = cc.LayerColor:create()
    layer:addChild(bg)

    -- 创建 Sprite3D 精灵对象
    local sprite = cc.Sprite3D:create("3D/ship.c3b")
    -- sprite:setTexture("3D/ship.png")
    sprite:setScale(10)
    sprite:setPosition(cc.p(size.width / 2, size.height / 2))
    sprite:setCameraMask(cc.CameraFlag.USER1)                                    ①
    layer:addChild(sprite)

    -- 创建 Camera 对象
    local camera = cc.Camera:createPerspective(60, size.width / size.height, 1, 1000)    ②
    -- 设置相机位置
    local spritePos = sprite:getPosition3D()                                     ③
    spritePos.y = spritePos.y + 200
    spritePos.z = spritePos.z + 600
    camera:setPosition3D(spritePos)
    -- 设置相机朝向和竖直方向向量
    camera:lookAt(spritePos)                                                     ④
    -- 设置相机 CameraFlag 属性
    camera:setCameraFlag(cc.CameraFlag.USER1)                                    ⑤
    layer:addChild(camera)

    ...

    return layer
end
```

为了让 3D 创建的 Node 对象在新的相机中可见，需要设置 Node 对象 CameraMask 属

性，对应的要设置相机的 CameraFlag 属性，当 Node 的 CameraMask 与相机的 CameraFlag 进行“与运算”，结果非零，则 Node 对象在该相机对象中可见。上述第①行代码是设置 Node 的 CameraMask 属性，cc. CameraFlag. USER1 是 Cocos2d-x 提供的常量，Cocos2d-x 提供了 cc. CameraFlag. USER1～cc. CameraFlag. USER8 可以使用的常量。而第⑤行代码设置的相机 CameraFlag 属性，也需要设置为 cc. CameraFlag. USER1。

第②行代码是创建透视投影 Camera 对象，60 是视野夹角，size. width/size. height 就是宽高比，近裁剪面距离为 1，远裁剪面距离为 1000。

第③行代码 sprite:getPosition3D()语句是获得相机位置，第④行代码是设置相机朝向和竖直方向向量，表达式 camera:lookAt(spritePos)是设置相机的朝向点在 Sprite3D 对象的位置，竖直方向向量采用默认值。

15.5　3D 粒子系统

Cocos2d-x 目前版本已经支持 Particle Universe3D 粒子系统。Particle Universe3D 粒子系统是非常绚丽的 3D 粒子系统。Particle Universe Editor(http://www. fxpression. com/)是一款免费开源的 3D 粒子系统编辑器。

15.5.1　创建 PUParticleSystem3D 对象

Cocos2d-x Lua API 提供了用于呈现 Particle Universe3D 粒子系统引擎——Particle3D，Particle3D 属于 Cocos2d-x Lua API 扩展部分。

在 Particle3D 中 3D 粒子 Node 是 PUParticleSystem3D 类，PUParticleSystem3D 继承于 Node 类。创建 PUParticleSystem3D 对象有多种方式，其中常用的函数如下：

(1) static PUParticleSystem3D: create()。创建一个 PUParticleSystem3D 对象。

(2) static PUParticleSystem3D: create(const std::string &filePath)。创建一个 PUParticleSystem3D 对象，filePath 指定 3D 粒子脚本路径，脚本是 Particle Universe Editor 工具编辑和导出的。

(3) static PUParticleSystem3D: create(const std::string &filePath, const std::string &materialPath)。创建一个 PUParticleSystem3D 对象，filePath 指定 3D 粒子脚本路径，materialPath 指定材质路径。

15.5.2　实例：创建 Particle Universe 3D 粒子

Particle Universe 3D 粒子包括脚本、材质和纹理贴图。Particle Universe 3D 粒子资源目录中有 3 个子目录，目录结构如下：

```
Particle3D
 ├—materials
 ├—scripts
 └—textures
```

scripts 目录中保存了脚本(*.pu)文件。materials 目录中保存一些材质文件,例如 pu_example.material 文件,这些文件是通过 Particle Universe Editor 导出的。textures 目录中保存纹理贴图所需的图片文件。

提示 materials 和 textures 目录名是固定的,不用修改为其他的名字。Particle Universe 3D 粒子会加载 materials 目录中的全部材质(*.material)文件,并到 textures 目录中查找纹理贴图所需的图片文件。

下面通过一个实例介绍如何创建 Particle Universe 3D 粒子系统,实例运行情况如图 15-9 所示,飞机周围会出现绚丽的 3D 粒子特效,当手指在屏幕上按下时飞机开始旋转,当抬起手指时飞机停止旋转。

图 15-9 创建 Particle Universe 3D 粒子系统实例

GameScene.lua 的 GameScene:createLayer()函数相关代码如下:

```
-- 创建层
function GameScene:createLayer()

    local layer = cc.Layer:create()
    local bg = cc.LayerColor:create()
    layer:addChild(bg)

    -- 创建 Sprite3D 精灵对象
    local sprite = cc.Sprite3D:create("3D/ship.c3b")
    -- sprite:setTexture("3D/ship.png")
    sprite:setScale(10)
    sprite:setPosition(cc.p(size.width / 2, size.height / 2))
    sprite:setCameraMask(cc.CameraFlag.USER1)
    layer:addChild(sprite)

    local rootps = cc.PUParticleSystem3D:create("Particle3D/scripts/example_010.pu")    ①
    rootps:setCameraMask(cc.CameraFlag.USER1)                                           ②
```

```
rootps:setScale(5)                                                    ③
rootps:startParticleSystem()                                          ④
sprite:addChild(rootps)                                               ⑤

-- 创建 Camera 对象
local camera = cc.Camera:createPerspective(60, size.width / size.height, 1, 1000)
-- 设置相机位置
local spritePos = sprite:getPosition3D()
spritePos.y = spritePos.y + 200
spritePos.z = spritePos.z + 600
camera:setPosition3D(spritePos)
    -- 设置相机朝向和竖直方向向量
    camera:lookAt(spritePos)
    -- 设置相机 CameraFlag 属性
    camera:setCameraFlag(cc.CameraFlag.USER1)
  layer:addChild(camera)

  ...

    return layer
end
```

第①和第②行代码是创建并设置 3D 粒子系统。其中，第①行代码通过 PUParticleSystem3D 类创建 3D 粒子系统 Node 对象，第②行代码是设置粒子系统的 CameraMask 属性，第③行代码是放大粒子 Node 对象，第④行代码是开始播放粒子系统，第⑤行代码是将 3D 粒子 Node 对象添加到 Sprite3D 对象中，注意本例没有添加到当前层，而是作为 Sprite3D 的子 Node 对象，当 Sprite3D 转动时粒子系统会一起转动。

本章小结

通过对本章的学习，可以了解到一些基本的 3D 概念，包括模型、相机和投影等，读者还学习了 Cocos2d-xLua API 中的 3D 特性，其中包括 3D 精灵对象、相机和 3D 粒子系统。

第三篇　数据与网络篇

本篇共 3 章，介绍 Cocos2d-x 游戏开发中的数据持久化、基于 HTTP 网络通信、Node. js 与 WebSocket 网络通信。

本篇各章如下：

第 16 章　文件访问操作和数据持久化

第 17 章　基于 HTTP 网络通信

第 18 章　Node. js 与 WebSocket 网络通信

第 16 章

文件访问操作和数据持久化

信息和数据在现代社会中扮演着至关重要的角色，已成为人们生活中不可或缺的一部分。我们经常接触的信息有电话号码本、QQ 通信录、消费记录等。作为游戏引擎的 Cocos2d-x Lua API 具有数据文件访问能力，这包括对文件访问操作、数据持久化和数据交换格式的解码/编码等支持。本章重点介绍文件访问操作和数据持久化实现。

16.1 使用 FileUtils 访问文件

FileUtils 类是 Cocos2d-x Lua API 提供的访问文件工具，由于不同的平台在文件格式、存放的目录和安全限制等方面有所不同，Cocos2d-x Lua API 又提供 FileUtils 的平台子类，如图 16-1 所示。

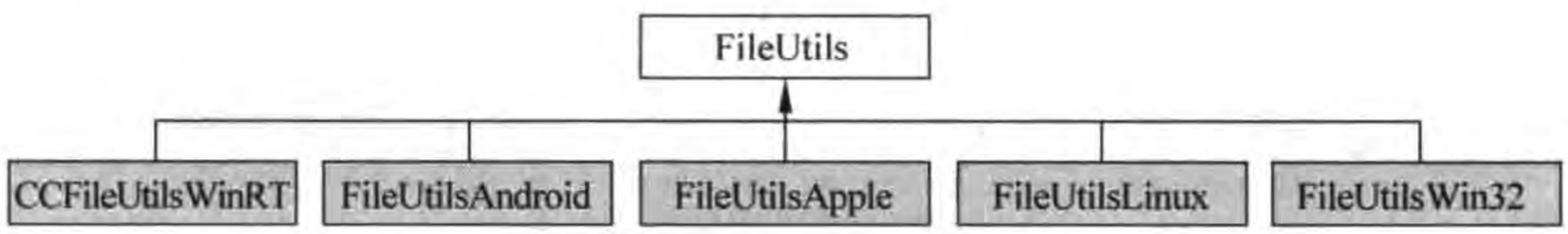


图 16-1 FileUtils 类图

具体使用时，一般情况下不需要直接使用这些子类，除非是某个平台下的特定内容。有关特定平台问题会在后面的章节介绍。

FileUtils 类采用单例设计模式，可以通过如下语句获得它的 FileUtils 实例：

```
local sharedFileUtils = cc.FileUtils:getInstance()
```

getInstance()函数第一次被调用时，Cocos2d-x Lua API 会根据所在平台实例化具体类。如在 iOS 平台，实例化结果是 FileUtilsApple 类的实例。

16.1.1 Cocos2d-x Lua API 中的目录

文件与平台有关，目录也是与平台有关的。在目前的移动平台下，Android、iOS 和 Windows Phone 出于安全考虑，可以访问的目录有资源目录和可写入目录。

1. 资源目录

资源目录只能读不能写入,放到资源目录中的文件一般是游戏中的图片、声音和视频等资源文件,也可以是配置文件等文本文件。不同平台的资源目录是不同的,但是可以将资源文件放置到 Cocos2d-x Lua API 工程中的 res 目录下。

FileUtils 类提供了 fullPathForFilename 函数访问 res 目录,代码如下:

```
local sharedFileUtils = cc.FileUtils:getInstance()
local fullPathForFilename = sharedFileUtils:fullPathForFilename("test.txt")
```

如果 test.txt 文件被放置在 res 目录下,在 Windows 平台下 Cocos Code IDE 工具中运行这段代码,结果可能是:< Cocos2d-x Lua API 工程目录>/res/test.txt。

当然文件也可以放置在其他目录,但是需要设置搜索路径,对目录进行搜索。搜索目录比较费时,可以使用 FileUtils 的 isFileExist 函数判断文件是否存在。代码如下:

```
local isExist = sharedFileUtils:isFileExist("test.txt")
```

isFileExist 函数判断文件是否存在,参数可以是相对路径,也可以是绝对路径。test.txt 写法就是相当路径,就是判断< Cocos2d-x Lua API 工程目录>/res/test.txt 文件是否存在。

2. 可写入目录

可写入目录在移动平台,如 Android、iOS 和 Windows Phone 是采用沙箱设计的,所谓沙箱设计就是只能被自己的应用访问,其他的应用不能或有限制访问。

图 16-2 所示是 iOS 平台模拟器中可写入目录,有关 iOS 平台的沙箱目录,请大家看一下 iOS 相关资料学习,这里不再介绍。

图 16-2 iOS 模拟器中可写入目录

FileUtils 类提供了 getWritablePath()函数获得可写入目录,代码如下:

```
local sharedFileUtils = cc.FileUtils:getInstance()
local writablePath = sharedFileUtils:getWritablePath()
```

16.1.2 实例: 读取文件

获得文件目录之后,接下来就可以对文件进行读写操作了,下面通过一个实例介绍读取文件内容。这个实例的界面如图 16-3 所示,界面中有 2 个菜单可以操作,每个菜单所完成的功能如菜单标签所示。第一个菜单进行资源目录测试,第二个菜单实现读取文件内容。

所有操作信息都是输出到日志中，而没有输出到场景中。

图 16-3　文件读写实例

GameScene.lua 文件中主要代码如下：

```
local sharedFileUtils = cc.FileUtils:getInstance()
...
function GameScene:createLayer()

    local layer = cc.Layer:create()
    --测试资源目录信息
    local function OnClickMenu1(menuItemSender)
        local fullPathForFilename = sharedFileUtils:fullPathForFilename("test.txt")      ①
        cclog("fullPathForFilename path = %s", fullPathForFilename)
        local isExist = sharedFileUtils:isFileExist("test.txt")                          ②
        if isExist == true then
            cclog("test.txt exists")
        else
            cclog("test.txt doesn't exist")
        end
    end

    --读文件
    local function OnClickMenu2(menuItemSender)
        local fullPathForFilename = sharedFileUtils:fullPathForFilename("test.txt")      ③
        cclog("test.txt path = %s", fullPathForFilename)
        local content = sharedFileUtils:getStringFromFile(fullPathForFilename)           ④
        cclog("content : %s",content)
    end

    local pItmLabel1 = cc.Label:createWithBMFont("fonts/fnt8.fnt", "Test ResourcesDir
Info")
    local pItmMenu1 = cc.MenuItemLabel:create(pItmLabel1)
    pItmMenu1:registerScriptTapHandler(OnClickMenu1)

    local pItmLabel2 = cc.Label:createWithBMFont("fonts/fnt8.fnt", "Read File")
    local pItmMenu2 = cc.MenuItemLabel:create(pItmLabel2)
    pItmMenu2:registerScriptTapHandler(OnClickMenu2)
```

```
    local mn = cc.Menu:create(pItmMenu1,pItmMenu2)
    mn:alignItemsVertically()
    layer:addChild(mn)

    return layer
end
```

上述第①和③行代码是获得资源目录中 test.txt 文件的全路径,第②行代码是判断该文件是否存在。第④行代码 getStringFromFile()函数是读取文件的内容。

16.1.3 实例: 路径搜索

有的时候在指定的目录中无法找到想要的文件,需要路径搜索。Cocos2d-x Lua API 提供了路径搜索能力,这是通过 FileUtils 类的一系列函数实现的,这些函数如下:

(1) getSearchPaths ()。获得所有搜索路径集合。

(2) setSearchPaths (searchPaths)。设置搜索路径集合。

(3) addSearchPath (path)。追加搜索路径。

下面通过一个实例介绍一下路径搜索的使用。这个实例的界面如图 16-4 所示,界面中有一个菜单 Search File 可以操作,Search File 菜单是在下面 3 个路径中搜索 test.txt 文件。

(1) "<可写入目录>"。

(2) "< Cocos2d-x Lua API 工程目录>/res/dir1"。

(3) "< Cocos2d-x Lua API 工程目录>/res/dir2"。

路径的搜索是有优先顺序的,可以通过代码控制这个顺序。

图 16-4 路径搜索实例

下面是路径搜索实例的代码片段,通过分析下面代码片段了解一下路径搜索的使用。

```
local sharedFileUtils = cc.FileUtils:getInstance()
...
    local function OnClickMenu1(menuItemSender)

        sharedFileUtils:purgeCachedEntries()                    ①

        local searchPaths = sharedFileUtils:getSearchPaths()    ②
```

```
local writablePath = sharedFileUtils:getWritablePath()

local resPrefix = "res/"

table.insert(searchPaths, 1, resPrefix.."dir2")                          ③
table.insert(searchPaths, 1, resPrefix.."dir1")                          ④
table.insert(searchPaths, 1, writablePath)                               ⑤
sharedFileUtils:setSearchPaths(searchPaths)                              ⑥

local fullPathForFilename = sharedFileUtils:fullPathForFilename("test.txt")   ⑦
cclog("test.txt 's fullPathForFilename is : %s",fullPathForFilename)
    local content = sharedFileUtils:getStringFromFile(fullPathForFilename)
    cclog("File content is : %s",content)

end
```

OnClickMenu1 函数是单击 Search File 菜单回调函数。第①行代码 sharedFileUtils:purgeCachedEntries()函数用来清理搜索文件缓存，一般是更新资源后进行搜索前调用。第②行代码是获得搜索路径的容器，返回值 searchPaths 是一个集合。

第③～⑤行代码是添加搜索路径，其中 table 是 Lua 的内置对象，它提供了 table 类型进行操作的函数，table.insert 函数语法如下：

```
table.insert(table, pos, value)
```

参数 table 是 table 类型集合变量，pos 是插入的位置，value 是插入到集合中的元素。

插入的顺序反映出路径的搜索顺序，第③～⑤行代码依次越高，即搜索路径的顺序如下：

```
"<可写入目录>""<Cocos2d-x Lua API 工程目录>/res/dir1""<Cocos2d-x Lua API 工程目录>/res/dir2"
```

第⑥行代码是设置搜索路径。如果设置完成搜索路径后还有路径想搜索，可以使用 addSearchPath 函数追加搜索路径，优先级排在最后。第⑦行代码是获得搜索到的文件路径。

代码编写完成后，可以运行一下看看效果。由于结果都输出到日志，所以需要查看日志信息。也可以改变一下搜索路径的顺序看看输出的结果有什么不同。

16.2　持久化概述

数据持久化就是数据能够存储起来，然后在需要时可以查找回来，即使是设备重新启动也可以查找回来。Cocos2d-x 3.x 能够支持至少 4 种持久化方式。

1. 普通文本文件

持久化的数据无论什么格式和结构都是要保存到文件中的。可以使用自定义结构化的文本，通过一些访问文件技术实现数据持久化。所谓“结构化”就是数据交换格式。

2. UserDefault

可以存放少量的数据,主要用于保存应用程序设置参数,例如控件的状态、用户使用偏好(背景、字体)设置等,一般而言它不会用来存放大量信息。

3. 属性列表

可以读写属性列表文件实现数据持久化。

4. SQLite 数据库

SQLite 是一个开源嵌入式关系型数据库。

目前 Cocos2d-x Lua API 中只提供了 UserDefault 方式的完善支持,所以本章重点介绍 UserDefault 和属性列表,数据交换格式等内容将在后面的章节介绍。

16.3 UserDefault 数据持久化

UserDefault 可以实现数据的存储,但是它不能泛滥使用,具体讲一般情况下不会使用它保存大量的数据,它没有 SQL 语句那样的灵活。UserDefault 除了保存游戏设置外,还可以长期保存游戏精灵等对象的状态。

UserDefault 类包含了数据读取函数和存储函数。它们是基于键-值对设计。它的数据最后被保存到可写入目录中的 UserDefault. xml 文件中,UserDefault. xml 文件示例代码如下:

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<userDefaultRoot>
    <sound_key>true</sound_key>
    <music_key>true</music_key>
</userDefaultRoot>
```

在 UserDefault. xml 文件中包括两个键值 sound_key 和 music_key,它们保存的是布尔值 true。

16.3.1 UserDefault API

UserDefault 类数据读取函数如下:

(1) getBoolForKey (pKey)。根据键取出布尔值。

(2) getBoolForKey (pKey, defaultValue)。根据键取出布尔值,如果键不存在,返回 defaultValue。

(3) getIntegerForKey (pKey)。根据键取出 int 类型数据。

(4) getIntegerForKey (pKey, defaultValue)。根据键取出 int 类型数据,如果键不存在,返回 defaultValue。

(5) getFloatForKey (pKey)。根据键取出 float 类型数据。

(6) getFloatForKey (pKey, defaultValue)。根据键取出 float 类型数据,如果键不存在,返回 defaultValue。

(7) getDoubleForKey(pKey)。根据键取出 double 类型数据。

(8) getDoubleForKey(pKey, defaultValue)。根据键取出 double 类型数据,如果键不存在,返回 defaultValue。

(9) getStringForKey (pKey)。根据键取出字符串类型数据。

(10) getStringForKey (pKey, defaultValue)。根据键取出字符串类型数据,如果键不存在,返回 defaultValue。

使用取值的实例代码如下:

```
local ret = cc.UserDefault:getInstance():getStringForKey("string")
cclog("string is %s", ret)

local d = cc.UserDefault:getInstance():getDoubleForKey("double")
cclog("double is %f", d)

local i = cc.UserDefault:getInstance():getIntegerForKey("integer")
cclog("integer is %d", i)

local f = cc.UserDefault:getInstance():getFloatForKey("float")
cclog("float is %f", f)

local b = cc.UserDefault:getInstance():getBoolForKey("bool")
if (b){
      cclog("bool is true")
}else{
      cclog("bool is false")
}
```

其中 cc.UserDefault:getInstance()是获得 UserDefault 实例,UserDefault 也是采用单例设计模式。

UserDefault 类数据存储函数如下:

(1) setBoolForKey(pKey, value)。根据键写入布尔值。

(2) setDoubleForKey(pKey, value)。根据键写入 double 类型数据。

(3) setFloatForKey(pKey, value)。根据键写入 float 类型数据。

(4) setIntegerForKey(pKey, value)。根据键写入 int 类型数据。

(5) setStringForKey(pKey, value)。根据键写入字符串类型数据。

使用取值的实例代码如下:

```
cc.UserDefault:getInstance():setStringForKey("string", "value2")
cc.UserDefault:getInstance():setIntegerForKey("integer", 11)
cc.UserDefault:getInstance():setFloatForKey("float", 2.5)
cc.UserDefault:getInstance():setDoubleForKey("double", 2.6)
cc.UserDefault:getInstance():setBoolForKey("bool", false)
```

16.3.2　实例:保存背景音乐和音效设置

下面通过一个实例介绍在游戏项目中如何使用 UserDefault。还记得 9.3 节介绍的“设

置背景音乐与音效"实例吗？如图 16-5 所示，在 SettingScene 场景中可以设置是否播放背景音乐和音效，但是 9.3 节没有实现保存状态的功能，现在将它完善，把选择的状态保存到 UserDefault 中。

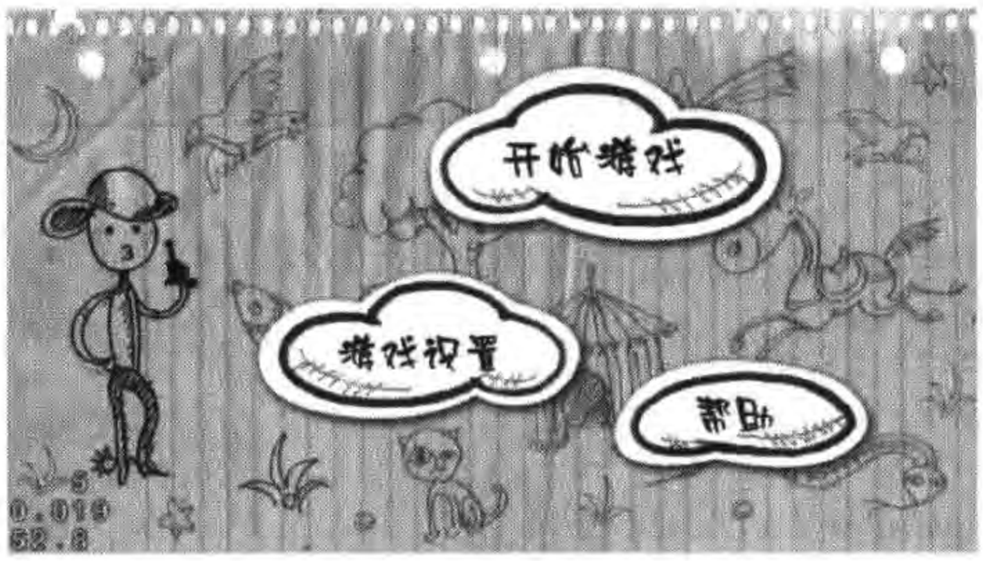


图 16-5　设置背景音乐与音效

GameScene.lua 主要代码如下：

```
require "AudioEngine"

EFFECT_FILE = "sound/Blip.wav"
MUSIC_FILE = "sound/Jazz.mp3"

SOUND_KEY = "sound_key"                                          ①
MUSIC_KEY = "music_key"                                          ②

size = cc.Director:getInstance():getWinSize()
local defaults = cc.UserDefault:getInstance()                    ③

...

-- create layer
function GameScene:createLayer()
    cclog("GameSceneinit")
    local layer = cc.Layer:create()

    ...
    -- 开始菜单
    local function menuItemStartCallback(sender)
        cclog("Touch Start.")
        if (defaults:getBoolForKey(SOUND_KEY)) then              ④
            AudioEngine.playEffect(EFFECT_FILE)
    end
end
startMenuItem:registerScriptTapHandler(menuItemStartCallback)

...
-- 设置菜单
local function menuItemSettingCallback(sender)
    cclog("Touch Setting.")
    local scene = require("SettingScene")
    local settingScene = scene.create()
```

```
    local ts = cc.TransitionJumpZoom:create(1, settingScene)
    cc.Director:getInstance():pushScene(ts)

    if (defaults:getBoolForKey(SOUND_KEY)) then                    ⑤
        AudioEngine.playEffect(EFFECT_FILE)
    end

end
settingMenuItem:registerScriptTapHandler(menuItemSettingCallback)

...
    -- 帮助图片菜单
    local function menuItemHelpCallback(sender)
        cclog("Touch Help.")

        if (defaults:getBoolForKey(SOUND_KEY)) then                ⑥
            AudioEngine.playEffect(EFFECT_FILE)
        end
    end
    helpMenuItem:registerScriptTapHandler(menuItemHelpCallback)

    local mn = cc.Menu:create(startMenuItem, settingMenuItem, helpMenuItem)
    mn:setPosition(cc.p(0, 0))
    layer:addChild(mn)

    return layer
end
...

function GameScene:onEnterTransitionFinish()
    cclog("GameScene onEnterTransitionFinish")
    if defaults:getBoolForKey(MUSIC_KEY) then                      ⑦
        AudioEngine.playMusic(MUSIC_FILE, true)
    end
end

...

return GameScene
```

上述第①和第②行代码是定义两个键，其中 SOUND_KEY 是音效状态键，MUSIC_KEY 是背景音乐播放状态键。

第③行代码 local defaults = cc.UserDefault:getInstance()是定义一个作用域范围，是 GameScene.lua 内的 UserDefault 变量，在后面使用时比较方便。

上述第④～⑥行代码中 defaults:getBoolForKey(SOUND_KEY)是获得 SOUND_KEY 键值，通过取出布尔值判断是否播放音效。第⑦行代码 Defaults:getBoolForKey(MUSIC_KEY)是获得 MUSIC_KEY 键值，通过取出布尔值判断是否播放背景音乐。

SettingScene.lua 主要代码如下：

```
require "AudioEngine"
```

```
local MUSIC_FILE = "sound/Synth.mp3"
local defaults   = cc.UserDefault:getInstance()

...

-- create layer
function SettingScene:createLayer()
    local layer = cc.Layer:create()

    local bg = cc.Sprite:create("setting-back.png")
    bg:setPosition(cc.p(size.width/2,
    size.height/2))
layer:addChild(bg)

-- 音效
...
local function menuSoundToggleCallback(sender)
    cclog("Sound Toggle.")
    if defaults:getBoolForKey(SOUND_KEY) then
        defaults:setBoolForKey(SOUND_KEY, false)
    else
        defaults:setBoolForKey(SOUND_KEY, true)
        AudioEngine.playEffect(EFFECT_FILE)
    end
end
soundToggleMenuItem:registerScriptTapHandler(menuSoundToggleCallback)

-- 音乐
...
local function menuMusicToggleCallback(sender)
    cclog("Music Toggle.")
    if (defaults:getBoolForKey(MUSIC_KEY)) then
        defaults:setBoolForKey(MUSIC_KEY, false)
        AudioEngine.stopMusic()
    else
        defaults:setBoolForKey(MUSIC_KEY, true)
        AudioEngine.playMusic(MUSIC_FILE, true)
    end
    if defaults:getBoolForKey(MUSIC_KEY) then
        AudioEngine.playEffect(EFFECT_FILE)
    end
end
musicToggleMenuItem:registerScriptTapHandler(menuMusicToggleCallback)

--初始化音乐开关菜单状态
if defaults:getBoolForKey(MUSIC_KEY) then                    ①
    musicToggleMenuItem:setSelectedIndex(0) --off            ②
else
    musicToggleMenuItem:setSelectedIndex(1) --on             ③
end
--初始化音效开关菜单状态
if defaults:getBoolForKey(SOUND_KEY) then                    ④
    soundToggleMenuItem:setSelectedIndex(0) --off            ⑤
else
```

```
    soundToggleMenuItem:setSelectedIndex(1) -- on                    ⑥
end

-- Ok 按钮
...
local function menuOkCallback(sender)
    cclog("Ok Menu tap.")
    cc.Director:getInstance():popScene()
    if defaults:getBoolForKey(MUSIC_KEY) then
        AudioEngine.playEffect(EFFECT_FILE)
    end
    okMenuItem:registerScriptTapHandler(menuOkCallback)

    local mn = cc.Menu:create(soundToggleMenuItem, musicToggleMenuItem,okMenuItem)
    mn:setPosition(cc.p(0, 0))
    layer:addChild(mn)
    return layer
end

...

function SettingScene:onEnterTransitionFinish()
    cclog("SettingScene onEnterTransitionFinish")
    if defaults:getBoolForKey(MUSIC_KEY) then
        AudioEngine.playMusic(MUSIC_FILE, true)
    end
    ...

    return SettingScene
```

上述代码是设置开关菜单的状态，第①～③行代码是设置背景音乐开关菜单，其中第②行代码是设置开关菜单为 Off，否则通过第③行代码设置状态为 On。第④～⑥行代码是设置音效开关菜单，其中第⑤行代码是设置开关菜单为 Off，否则通过第⑥行代码设置状态为 On。

其他代码与 GameScene 场景类似，不再赘述。

运行一下先将状态保存，然后重新运行游戏，看是否能够保持状态。

16.4　属性列表数据持久化

有些数据可以保存到 XML 文件中，但是对于 XML 的解析比较复杂。而属性列表文件是一种特定的 XML 文件，Cocos2d-x Lua 中的 table 可以与属性列表文件互相转换，这样就省去了 XML 的解析麻烦。

16.4.1　属性列表概述

属性列表文件是苹果公司定义的，设计的目的是为了 Objective-C 中的字典结构类

NSDictionary 和列表结构类 NSArray 映射到文件。所以从属性列表文件根结构看,它要么属于字典结构,要么属于列表结构。

根为列表结构的属性列表文件代码如下:

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<! DOCTYPE plist PUBLIC " - //Apple//DTD PLIST 1. 0//EN" " http://www. apple. com/DTDs/
PropertyList - 1.0.dtd">
<plist version = "1.0">
    <array>
        ...
    </array>
</plist>
```

根为字典结构的属性列表文件代码如下:

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<! DOCTYPE plist PUBLIC " - //Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList -
1.0.dtd"/>
<plist version = "1.0">
    <dict>
        ...
    </dict>
</plist>
```

Objective-C 中的字典结构和列表结构可以使用 Lua 语言的 table 类型描述。Cocos2d-x Lua 提供了 table 类型与属性列表文件互相映射 API。下面通过两个不同的实例介绍它们访问方式有什么不同。

16.4.2 实例:访问根为字典结构的属性列表文件

下面代码是一个根为字典结构的属性列表文件 NotesList. plist 实例。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<! DOCTYPE plist PUBLIC " - //Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList -
1.0.dtd"/>

<plist version = "1.0">
    <dict>
        <key> root </key>
        <array>
            <dict>
            <key> date </key>
            <string> 2010 - 08 - 04 16:01:03 </string>
            <key> content </key>
            <string>初始化数据。</string>
    </dict>
    <dict>
            <key> date </key>
            <string> 2011 - 12 - 04 16:01:03 </string>
            <key> content </key>
            <string>欢迎使用 MyNote。</string>
```

```
            </dict>
        </array>
    </dict>
</plist>
```

从 NotesList.plist 文件中可以看出它的结构，如图 16-6 所示。根结构是字典结构，在字典中包含列表结构，列表中的每个元素结构又是字典，字典中包含两个键-值对，例如第一个元素：date=2010-08-04 16:01:03 和 content=初始化数据。

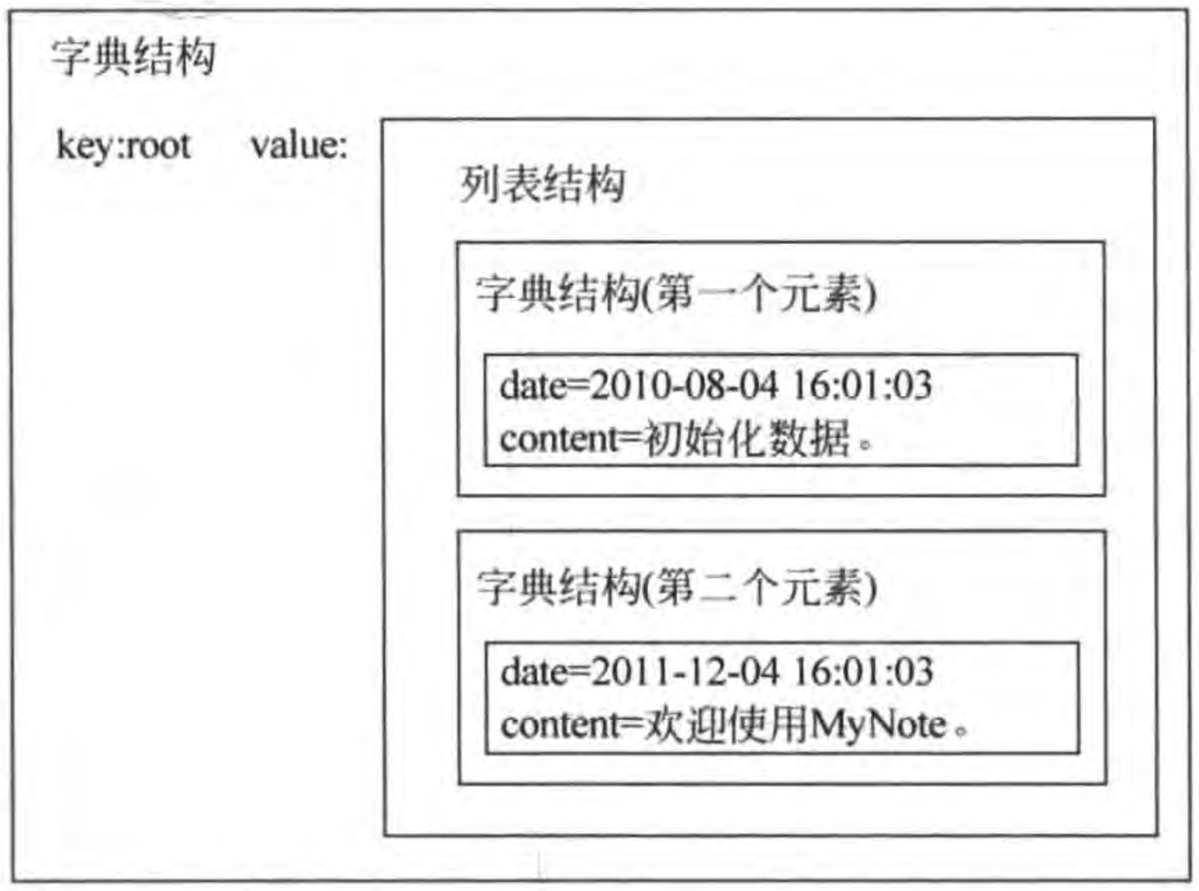


图 16-6　与 NotesList.plist 对应的字典结构对象

FileUtils 通过如下方法读取根为字典结构的属性列表文件：

```
cc.FileUtils:getInstance():getValueMapFromFile(fullPathForFilename)
```

这个实例有一个界面(见图 16-7)，界面中有一个菜单可以操作，单击菜单从 NotesList.plist 文件中读取内容，并把文件内容输出到日志中，而没有输出到场景中。

图 16-7　实例界面

GameScene.lua 主要代码如下：

```
-- create layer
function GameScene:createLayer()
    cclog("GameScene init")
    local layer = cc.Layer:create()

    local function OnClickMenu1(tag, menuItemSender)
        local sharedFileUtils = cc.FileUtils:getInstance()                          ①
        local fullPathForFilename = sharedFileUtils:fullPathForFilename("NotesList.plist")  ②
        local dict = sharedFileUtils:getValueMapFromFile(fullPathForFilename)       ③

    for key, value in pairs(dict) do                                                ④
        for i = 1, table.getn(value) do                                             ⑤
            cclog("----------[%d]----------", i)
            local row = value[i]                                                    ⑥
            local date = row["date"]                                                ⑦
            local content = row["content"]                                          ⑧
            cclog("date :  %s", date)
            cclog("content : %s", content)
        end
        break
    end
end

local pItmLabel1 = cc.Label:createWithBMFont("fonts/fnt8.fnt", "Dictionary Structure")
local pItmMenu1 = cc.MenuItemLabel:create(pItmLabel1)
    pItmMenu1:registerScriptTapHandler(OnClickMenu1)

    local mn = cc.Menu:create(pItmMenu1)
    mn:alignItemsVertically()
    layer:addChild(mn)

    return layer
end
```

单击菜单调用 OnClickMenu1 函数，其中第①行代码是获得 FileUtils 实例。第②行代码是获得资源目录中 NotesList.plist 的全路径。第③行代码通过 getValueMapFromFile 函数读取属性列表文件 NotesList.plist，结果读取到 dict 变量中。

第④行代码 for key, value in pairs(dict) do 是遍历 dict 字典变量，由于它只有一个键-值对，所以这个循环只运行一次，其中 key 是键，value 是值，value 还是一个列表结构。

第⑤行代码 for i=1, table.getn(value) do 是遍历 value 列表变量，其中 table.getn() 可以获得 value 列表长度，i 是循环变量。第⑥行代码 local row = value[i]是获得 value 列表变量中的一个元素。第⑦行代码 local date = row["date"]是从一个元素中取出 date 数据项，第⑧行代码 local content = row["content"]是从一个元素中取出 content 数据项。

运行之后输出结果如下：

```
[LUA-print] ----------[1]----------
[LUA-print] date :2010-08-04 16:01:03
[LUA-print] content : 初始化数据。
```

```
[LUA-print] ----------[2]----------
[LUA-print] date :2011-12-04 16:01:03
[LUA-print] content : 欢迎使用 MyNote。
```

16.4.3　实例：访问根为列表结构的属性列表文件

下面重新设计属性列表文件 NotesList.plist 结构，NotesList.plist 内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<! DOCTYPE plist PUBLIC " -//Apple//DTD PLIST 1.0//EN" " http://www.apple.com/DTDs/
PropertyList-1.0.dtd">

<plist version="1.0">
  <array>
    <dict>
      <key>date</key>
      <string>2010-08-04 16:01:03</string>
      <key>content</key>
      <string>初始化数据。</string>
    </dict>
    <dict>
      <key>date</key>
      <string>2011-12-04 16:01:03</string>
      <key>content</key>
      <string>欢迎使用 MyNote。</string>
    </dict>
  </array>
</plist>
```

这个结构如图 16-8 所示，与图 16-6 所示的结构进行比较，这种结构比较简单。

图 16-8　与 NotesList.plist 对应的列表结构对象

FileUtils 通过了如下方法读取根为列表结构的属性列表文件：

```
cc.FileUtils:getInstance():getValueVectorFromFile(fullPathForFilename)
```

这个实例的界面如图 16-9 所示，界面中有一个菜单可以操作，单击菜单，从 NotesList.

plist 文件中读取内容，并把文件内容输出到日志中，而没有输出到场景中。

图 16-9　实例界面

GameScene.lua 主要代码如下：

```
-- create layer
function GameScene:createLayer()
    cclog("GameScene init")
    local layer = cc.Layer:create()

    local function OnClickMenu1(tag, menuItemSender)
        local sharedFileUtils = cc.FileUtils:getInstance()                                    ①
        local fullPathForFilename = sharedFileUtils:fullPathForFilename("NotesList.plist")    ②
        local vector = sharedFileUtils:getValueVectorFromFile(fullPathForFilename)            ③

        for i = 1, table.getn(vector) do                                                      ④
            cclog("----------[%d]----------", i)
            local row = vector[i]                                                             ⑤
            local date = row["date"]
            local content = row["content"]
            cclog("date :  %s", date)
            cclog("content : %s", content)
        end

    local pItmLabel1 = cc.Label:createWithBMFont("fonts/fnt8.fnt", "Dictionary Structure")
    local pItmMenu1 = cc.MenuItemLabel:create(pItmLabel1)
        pItmMenu1:registerScriptTapHandler(OnClickMenu1)

        local mn = cc.Menu:create(pItmMenu1)
        mn:alignItemsVertically()
        layer:addChild(mn)

        return layer
end
```

单击菜单调用 OnClickMenu1 函数，其中第①行代码是获得 FileUtils 实例。第②行代码是获得资源目录中 NotesList.plist 的全路径。第③行代码是通过 getValueVectorFrom-

File 函数读取属性列表文件 NotesList. plist，结果读取到 vector 变量中。

第④行代码 for i=1，table. getn(vector) do 是遍历 vector 列表变量，其中 table. getn()可以获得 vector 列表长度，i 是循环变量。第⑤行代码 local row = vector[i]是获得 vector 列表变量中的一个元素。

运行之后输出结果如下：

```
[LUA-print] -----------[1]------------
[LUA-print] date :  2010-08-04 16:01:03
[LUA-print] content : 初始化数据。
[LUA-print] -----------[2]------------
[LUA-print] date :  2011-12-04 16:01:03
[LUA-print] content : 欢迎使用 MyNote。
```

本章小结

通过对本章的学习，使广大读者了解了 UserDefault 持久化技术，以及访问文件类 FileUtils。

第 17 章

基于 HTTP 网络通信

现在很多游戏需要网络通信，网络结构有客户端服务器(Client Server，C/S)结构网络和点对点(Peer to Peer，P2P)结构网络。考虑到跨平台需要，Cocos2d-x Lua API 主要采用 C/S 结构网络。P2P 结构网络一般采用蓝牙通信，特定平台一般提供了访问 P2P 的本地 API，例如 iOS 的 Game Kit，但是这些 API 不能使用在具有跨平台特性的 Cocos2d-x 引擎。

本章介绍在 Cocos2d-x 中使用的基于 HTTP 的网络通信 Lua API。

17.1 网络结构

网络结构是网络的构建方式，目前流行的有客户端服务器结构网络和点对点结构网络。

17.1.1 客户端服务器结构网络

客户端服务器结构网络是一种主从结构网络，如图 17-1 所示，服务器一般处于等待状态，如果有客户端请求，服务器响应请求建立连接提供服务。服务器是被动的，有点像在餐厅服务的服务员。而客户端是主动的，像在餐厅吃饭的顾客。

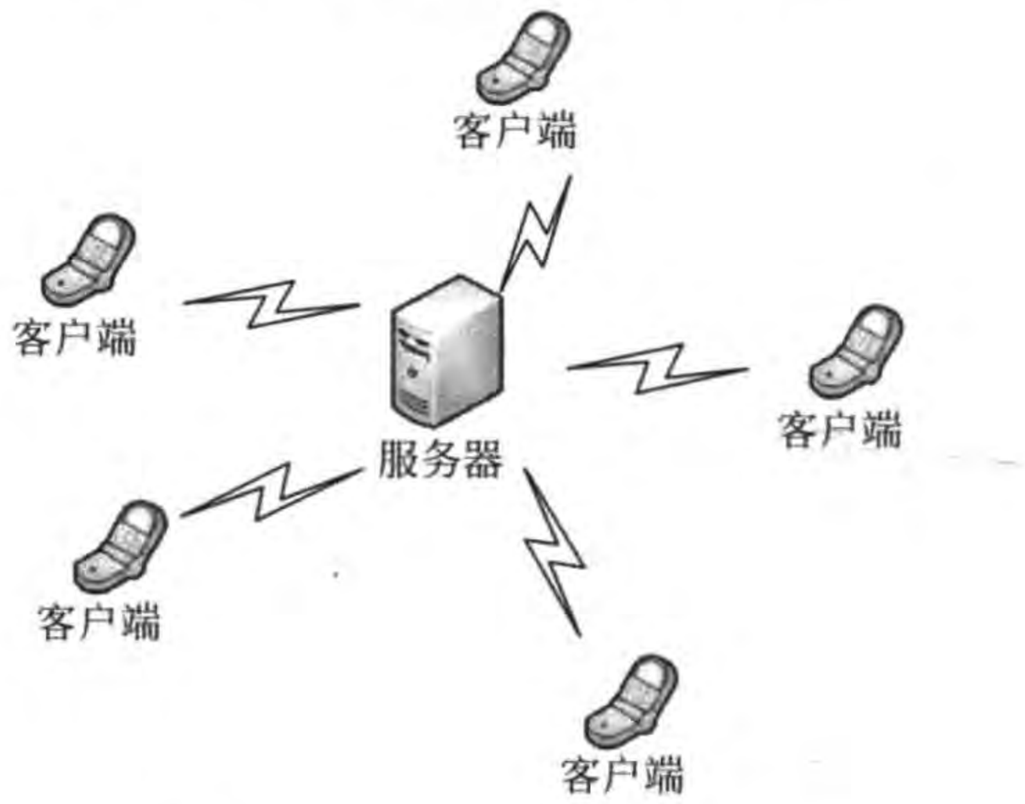


图 17-1 客户端服务器结构网络

事实上，很多网络服务都采用这种结构。例如 Web 服务、文件传输服务和邮件服务等。虽然它们存在的目的不一样，但基本结构是一样的。这种网络结构与设备类型无关，服务器不一定是电脑，也可能是手机等移动设备。如 iPhone 应用的内置 Web 服务，可以在电脑上通过浏览器下载手机中的图片和资料。

17.1.2　点对点结构网络

点对点结构网络也叫对等结构网络，每个节点之间是对等的，如图 17-2 所示，每个节点既是服务器又是客户端，这种结构有点像吃自助餐。

对等结构网络分布范围比较小。通常在一间办公室或一个家庭内，因此它非常适合于移动设备间的网络通信，网络链路层是由蓝牙和 WiFi 实现的。

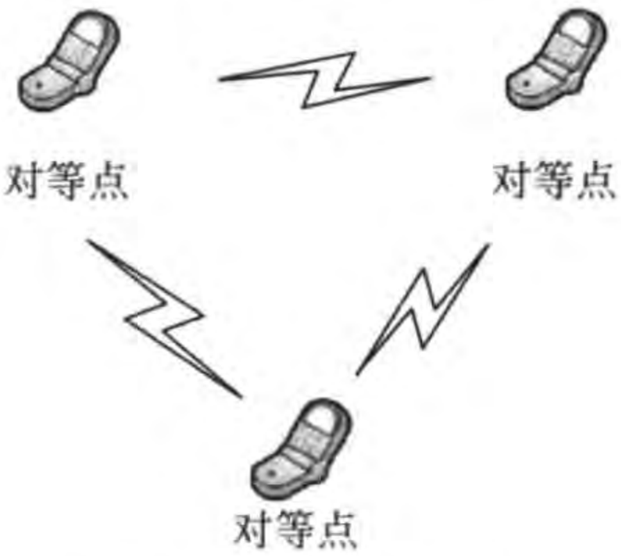


图 17-2　对等结构网络

17.2　HTTP 与 HTTPS 协议

客户端服务器应用层主要采用的是 HTTP 和 HTTPS 等传输协议。因此有必要介绍一下 HTTP 和 HTTPS 协议。如果读者对于这两个协议比较熟悉，可以跳过这一节，继续学习后面的内容。

1. HTTP 协议

HTTP 是 Hypertext Transfer Protocol 的缩写，即超文本传输协议。Internet 的基本协议是 TCP/IP 协议，目前广泛采用的 HTTP、HTTPS、FTP、Archie Gopher 等都是建立在 TCP/IP 协议之上的应用层协议，不同的协议对应着不同的应用。

HTTP 是一个属于应用层的面向对象的协议。由于其简捷、快速的方式，适用于分布式超文本信息传输。它于 1990 年提出，经过几年的使用与发展，不断地完善和扩展。HTTP 协议支持客户端服务器网络结构，是无连接协议，即每一次请求时建立连接，服务器处理完客户端的请求后，应答给客户端然后断开连接，不会一直占用网络资源。

HTTP/1.1 协议共定义了 8 种请求方法：OPTIONS、HEAD、GET、POST、PUT、DELETE、TRACE 和 CONNECT。作为 Web 服务器至少需实现 GET 和 HEAD 方法，其他方法都是可选的。

GET 方法是向指定的资源发出请求，发送的信息显示在 URL 后面，使用 GET 方法应该只用在读取数据，例如静态图片等数据。GET 方法像是使用明信片给别人写信，信的内容写在外面，接触到的人都可以看到，因此不安全。

POST 方法是向指定资源提交数据，请求服务器进行处理。例如提交表单或者上传文件等。数据被包含在请求体中。POST 方法像是把信的内容装入到信封中给别人写信，接触到的人都看不到，因此是安全的。

2. HTTPS 协议

HTTPS 是 Hypertext Transfer Protocol Secure 的缩写，即安全超文本传输协议，是超文本传输协议和 SSL 的组合，提供加密通信及对网络服务器身份的鉴定。

简单说，HTTPS 是 HTTP 的升级版，与 HTTP 的区别是：HTTPS 使用 https://代替 http://，HTTPS 使用端口 443，而 HTTP 使用端口 80 和 TCP/IP 进行通信。SSL 使用 40 位关键字作为 RC4 流加密算法，这对于商业信息的加密是合适的。HTTPS 和 SSL 支持使用 X.509 数字认证，如果需要，用户可以确认发送者是谁。

17.3 使用 XMLHttpRequest 对象开发客户端

在 Web 前端开发中有一种异步刷新技术——AJAX(Asynchronous JavaScript and XML)，AJAX 的核心是 JavaScript 对象 XMLHttpRequest。该对象在 Internet Explorer 5 中首次引入，它是一种支持异步请求的技术。借助于该 XMLHttpRequest 对象使用 JavaScript 语言向服务器提出请求并处理响应。

17.3.1 使用 XMLHttpRequest 对象

由于在 Web 中使用 XMLHttpRequest 对象发出 HTTP 请求很普遍，Cocos2d-x Lua API 对其进行了移植，可以在 Cocos2d-x LuaAPI 中使用 XMLHttpRequest 对象。

XMLHttpRequest 对象中几个常用的函数和属性如下：

(1) open()。与服务器连接，创建新的请求。

(2) send()。向服务器发送请求。

(3) abort()。退出当前请求。

(4) readyState 属性。提供当前请求的就绪状态，其中 4 表示准备就绪。

(5) status 属性。提供当前 HTTP 请求状态码，其中 200 表示成功请求。

(6) responseText 属性。服务器返回的请求响应文本。

(7) onreadystatechange 属性。设置回调函数，当服务器处理完请求后就会自动调用该函数。

其中 open 和 send 函数，以及 onreadystatechange 属性是 HTTP 请求的关键。open 函数有以下 5 个参数可以使用：

(1) request-type：发送请求的类型。典型的值是 GET 或 POST，也可以发送 HEAD 请求。

(2) url：要请求连接的 URL。

(3) asynch：如果希望使用异步连接则为 true，否则为 false。该参数是可选的，默认为 true。

(4) username：如果需要身份验证，则可以在此指定用户名。该可选参数没有默认值。

(5) password：如果需要身份验证，则可以在此指定口令。该可选参数没有默认值。

17.3.2　实例：MyNotes

下面通过一个实例来学习 MyNotes 应用。这个应用(MyNotes)是用来管理和记录备忘录信息的，它具有增加、删除和查询备忘录的基本功能。图 17-3 是 MyNotes 应用的用例图。由于需求很简单，只需要保存备忘录(Note)信息和对应的时间就可以了。

MyNotes 实例的数据是存储在云服务器端数据库中(以后简称“云数据库”)，为此搭建了一个远程服务器，客户端使用 XMLHttpRequest 与服务器通信。由于服务器端已经搭建好了，只考虑客户端这边的开发，客户端界面如图 17-4 所示。界面中有 4 个菜单可以操作，每个菜单都能与云服务器的 WebService 交互，完成操作云数据库的 CRUD 功能。所有从云服务器返回的数据都输出到日志中，而没有输出到场景中。

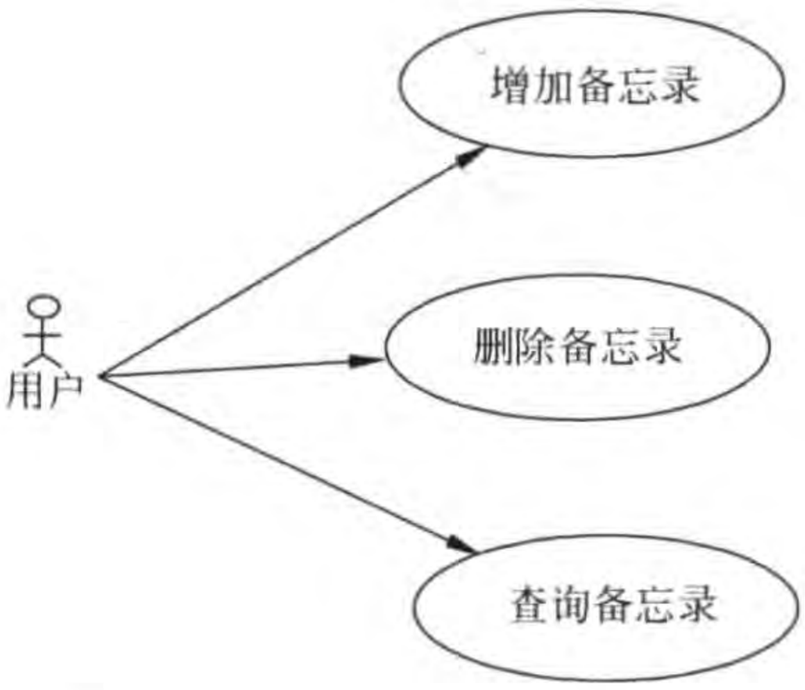


图 17-3　MyNotes 应用的用例图

图 17-4　MyNotes 实例

在介绍客户端开发之前有必要了解一下 MyNotes 实例的 WebService API，它是由智捷课堂(http://www.51work6.com)提供的，读者要想使用这些 WebService API，必须到 http://www.51work6.com 网站注册用户，在注册时需要提供一个邮箱，把这个邮箱作为一个 email 参数提交给 WebService，具体细节可以参看后面代码部分。

WebService API 提供了 4 个操作(CRUD)调用，使用的 URL 地址为 http://www.51work6.com/service/mynotes/WebService.php，采用 HTTP 的请求方法，建议使用 POST 方法，这是因为 GET 请求静态资源，数据传输过程也不安全，而 POST 主要请求动态资源。这些方法的调用都需要传递很多参数，它们之间的关系如表 17-1 所示。

表 17-1　WebService 方法调用与参数关系

调用方法	type 参数	action 参数	id 参数	date 参数	content 参数	email 参数
add	需要	需要	不需要	需要	需要	需要
modify	需要	需要	需要	需要	需要	需要
remove	需要	需要	需要	不需要	不需要	需要
query	需要	需要	需要	不需要	不需要	需要

表 17-1 参数说明如下：

(1) type。是数据交换类型，WebService 提供 3 种类型的数据：JSON、XML 和 SOAP，用户提供其中一个参数，而且全部是大写，它是数据交互类型，本书中主要使用 JSON 数据交换类型。

(2) action。是指定调用 WebService 的方法，这些方法有 add、remove、modify 和 query，分别代表插入、删除、修改和查询处理。

(3) id。一条 Note 信息中的主键，它隐藏在界面之后，当删除和修改时需要把它传给 WebService。

(4) date。一条 Note 信息中的日期字段数据。

(5) content。Note 信息中的内容字段数据。

(6) email。在 www.51work6.com 网站注册时提供给网站的邮箱，注意不是用户 ID。

GameScene.lua 中初始化 GameScene 场景的代码如下：

```
-- 定义常量
local BASE_URL = 'http://www.51work6.com/service/mynotes/WebService.php'                ①
--查询之后修改,用于删除和修改
local selectedRowId = 1483                                                               ②

...

-- create layer
function GameScene:createLayer()

    local layer = cc.Layer:create()
    ...

    <菜单回调函数>
    ...

    local pItmLabel1 = cc.Label:createWithBMFont("fonts/fnt8.fnt", "Insert Data")
    local pItmMenu1 = cc.MenuItemLabel:create(pItmLabel1)
    pItmMenu1:registerScriptTapHandler(onMenuInsertCallback)

    ...

    local pItmLabel4 = cc.Label:createWithBMFont("fonts/fnt8.fnt", "Read Data")
    local pItmMenu4 = cc.MenuItemLabel:create(pItmLabel4)
    pItmMenu4:registerScriptTapHandler(onMenuReadCallback)

    local mn = cc.Menu:create(pItmMenu1,pItmMenu2,pItmMenu3,pItmMenu4)
    mn:alignItemsVertically()
    layer:addChild(mn)

    return layer
end
```

上述第①行代码是定义一个全局变量 BASE_URL，它保存了请求服务器的 URL 网址，注意这里的 URL 网址是大小写敏感的。

第②行代码是定义一个全局变量 selectedRowId，它保存了要删除或修改的记录 ID，这个 ID 的获得是先单击 Read Data 菜单查询一下服务器上的相关数据，查询的结果输出到控制台，然后找到合法的 ID，修改 selectedRowId 值。这种测试的方式很显然不够“智能”，从服务器返回的数据是 JSON 数据，通过程序读取 ID 需要有 JSON 的解码，有关 JSON 的解码问题将会在下一节介绍，本节先不考虑解码问题。因此 selectedRowId 的取值也采用手动赋值方式。

GameScene.lua 中 Read Data 菜单查询代码如下：

```
--查询数据函数
 local function onMenuReadCallback(pSender)
    cclog("onMenuReadCallback")

    local data = string.format("email = %s&type = %s&action = %s",
                        "<www.51work6.com 注册邮箱>", "JSON", "query")          ①
    local url = BASE_URL .. "?" .. data                                          ②
    local xhr = cc.XMLHttpRequest:new()                                          ③
    xhr.responseType = cc.XMLHTTPREQUEST_RESPONSE_JSON                           ④
    xhr:open("GET", url)                                                         ⑤

    local function onReadyStateChange()                                          ⑥
        if xhr.readyState == 4 and xhr.status == 200 then                        ⑦
            local response = xhr.responseText                                    ⑧
            cclog(response)
        end
        xhr:registerScriptHandler(onReadyStateChange)                            ⑨
        xhr:send()                                                               ⑩

 end
```

上述第①行代码是准备要传递给服务器的数据，它们按照“键＝值”形式通过“&”符号连接起来。参数含义参考表 17-1。第②行代码是 BASE_URL 和 data 字符串连接起来，GET 请求的参数是放到 URL 之后用“?”连接起来。

第③行代码是通过 cc.XMLHttpRequest:new()函数创建 XMLHttpRequest 对象。第④行代码是设置应答类型，应答类型有如下 3 种：

(1) XMLHTTPREQUEST_RESPONSE_STRING。应答返回的是文本字符。

(2) XMLHTTPREQUEST_RESPONSE_ARRAY_BUFFER。应答返回的是二进制数据。

(3) XMLHTTPREQUEST_RESPONSE_JSON。应答返回的是 JSON 字符串。

第⑤行代码 xhr:open("GET", url)使用 open 函数与服务器建立连接创建请求对象，第一个参数是设置发送请求类型为 GET 方式，第二个参数是请求的地址。

第⑥行代码 onReadyStateChange()是回调函数。第⑦行代码中 xhr.readyState == 4 是判断 HTTP 就绪状态，XMLHttpRequest 中有 5 种就绪状态。

(1) 0：请求没有发出，在调用 open()函数之前为该状态。

(2) 1：请求已经建立但还没有发出，在调用 send()函数之前为该状态。

(3) 2：请求已经发出正在处理之中。

(4) 3：请求已经处理，响应中通常有部分数据可用，但是服务器还没有完成响应。

(5) 4：响应已完成，可以访问服务器响应并使用它。

第⑧行代码中 xhr.status == 200 是判断 HTTP 状态码，常见的 HTTP 状态码如下：

(1) 401：表示所访问数据禁止访问。

(2) 403：表示所访问数据受到保护。

(3) 404：表示错误的 URL 请求，表示请求的服务器资源不存在。

(4) 200：表示一切顺利。

如果就绪状态是 4 而且状态码是 200 就可以处理服务器的数据了。

第⑨行代码 xhr:registerScriptHandler(onReadyStateChange)是注册请求服务器事件。第⑩行代码是使用 send()函数发送数据。

GameScene.lua 中插入、删除和修改数据代码如下：

```
--插入数据函数
local function onMenuInsertCallback(pSender)                                    ①
    cclog("onMenuInsertCallback")

    local data = string.format("email = %s&type = %s&action = %s&date = %s&content = %s",
        "<www.51work6.com 注册邮箱>", "JSON", "add", "2016-09-18", "Tony insert data")②
    local xhr = cc.XMLHttpRequest:new()
    xhr.responseType = cc.XMLHTTPREQUEST_RESPONSE_JSON
    xhr:open("POST", BASE_URL)                                                  ③

    local function onReadyStateChange()
        if xhr.readyState == 4 and xhr.status == 200 then
            local response = xhr.responseText
            cclog(response)
        end
    end
    xhr:registerScriptHandler(onReadyStateChange)
    xhr:send(data)                                                              ④
end

--删除数据函数
 local function onMenuDeleteCallback(pSender)                                   ⑤
    cclog("onMenuDeleteCallback")

local data = string.format("email = %s&type = %s&action = %s&id = %s",
                "<www.51work6.com 注册邮箱>", "JSON", "remove", selectedRowId)
local xhr = cc.XMLHttpRequest:new()
xhr.responseType = cc.XMLHTTPREQUEST_RESPONSE_JSON
xhr:open("POST", BASE_URL)

local function onReadyStateChange()
    if xhr.readyState == 4 and xhr.status == 200 then
        local response = xhr.responseText
        cclog(response)
```

```
        end
    end
    xhr:registerScriptHandler(onReadyStateChange)
    xhr:send(data)
end

--更新数据函数
local function onMenuUpdateCallback(pSender)                                    ⑥
    cclog("onMenuUpdateCallback")

    local data = string.format("email = %s&type = %s&action = %s&date = %s&content = %s&id = %s",
                    "<www.51work6.com 注册邮箱>", "JSON", "modify", "2016-09-18",
                    "Tom modify data", selectedRowId)

    local xhr = cc.XMLHttpRequest:new()
    xhr.responseType = cc.XMLHTTPREQUEST_RESPONSE_JSON
    xhr:open("POST", BASE_URL)

    local function onReadyStateChange()
        if xhr.readyState == 4 and xhr.status == 200 then
            local response = xhr.responseText
            cclog(response)
        end
    end
    xhr:registerScriptHandler(onReadyStateChange)
    xhr:send(data)
end
```

上述第①、⑤、⑥行代码定义的函数是菜单项 Insert Data、Delete Data、Update Data 单击时回调的函数。这 3 个函数的代码非常相似，只是请求的参数数据不同，下面重点介绍其中的一个，第②行代码设置请求数据，第③行代码 xhr.open("POST", BASE_URL)是通过 POST 方式请求服务器，与 GET 方式不同，不需要在 BASE_URL 后面提供参数数据(URL 中"?"之后内容)。这些参数数据是通过第④行代码 xhr.send(data)发送给服务器的。

提示　在测试删除或修改的时候，不要忘了先查询到一个有效的 ID，然后在代码中修改 selectedRowId 初始值，然后再进行删除或修改。

17.4　数据交换格式

数据交换格式是在两个程序之间对话的"语音"。下面从一个身边的故事开始引入数据交换的概念和意义。

我有一次向同学借一本《小学生常用成语词典》，结果到他家时，他不在，于是我写了下面(见图 17-5)的留言条。

留言条与类似的书信都有一定的格式，我曾经也学习过如何写留言条。它有 4 个部分：称谓、内容、落款和时间，如图 17-6 所示。

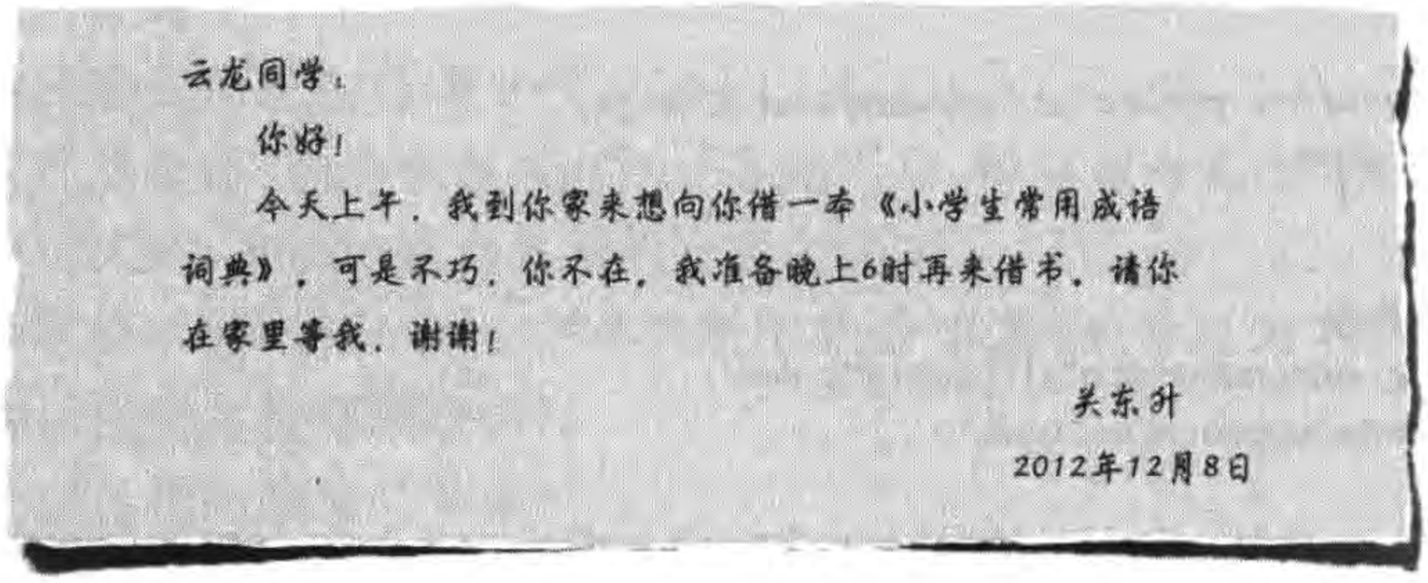


图 17-5　留言条

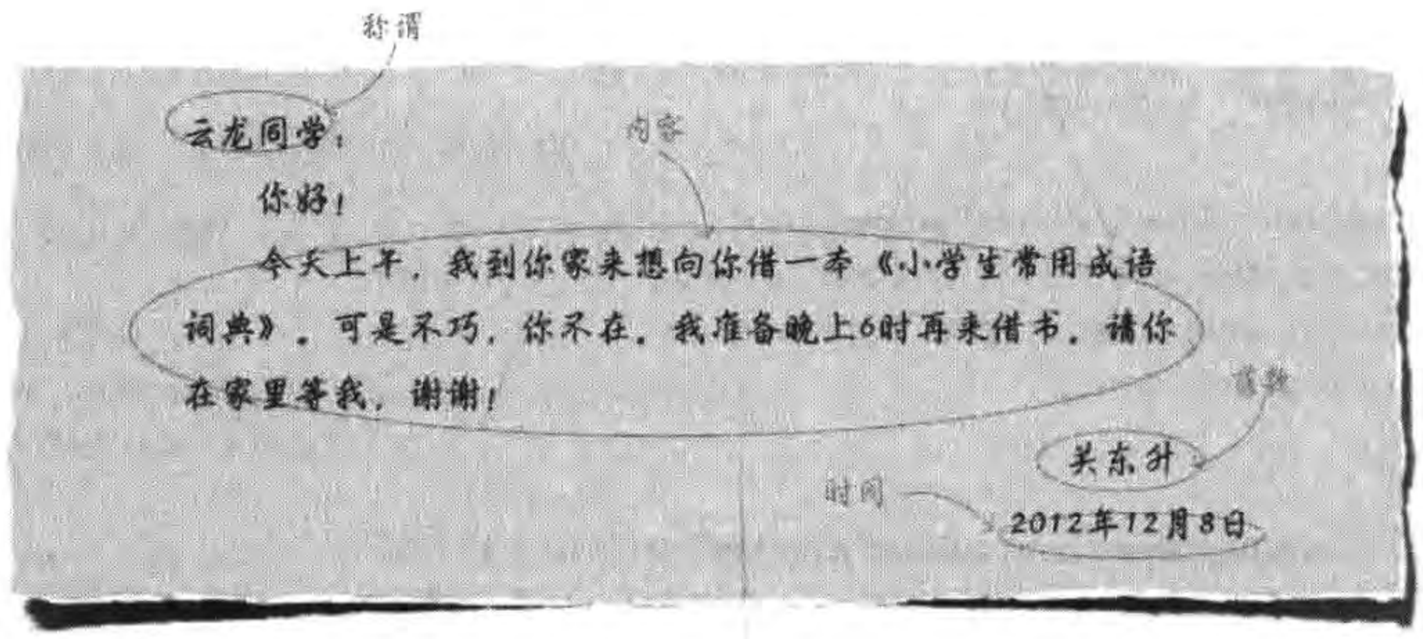


图 17-6　留言条格式

这 4 个部分是不能搞乱的，云龙同学也懂得这个格式，否则就会出笑话。他知道“称谓”部分是称呼他，“内容”部分是要做什么，“落款”部分是谁写给他的，“时间”部分是什么时候写的。

留言条是信息交换的手段，需要“写”与“看”的人都要遵守某种格式。计算机中两个程序之间的相互通信，也是要约定好某种格式。

数据交换格式就像两个人聊天，采用彼此都能听得懂的语言，你来我往。其中的语言就相当于通信中的数据交换格式。有时为防止聊天被人偷听，可以采用暗语。同理，计算机程序之间也可以通过数据加密技术防止“偷听”现象。

数据交换格式主要分为 CSV 格式、XML 格式和 JSON 格式。CSV 格式是一种简单的逗号分隔交换方式。上面的留言条写成 CSV 格式如下：

```
"云龙同学","你好!\n 今天上午,我到你家来想向你借一本《小学生常用成语词典》。可是不巧,你不在。我准备晚上 6 时再来借书。请你在家里等我,谢谢!","关东升","2012 年 12 月 08 日"
```

留言条中的 4 部分数据按照顺序存放，各个部分之间用逗号分割。有时数据量很小，可以采用这种格式。但是随着数据量的增加，问题也会暴露出来。人们总是会搞乱它们的顺序，如果各个数据部分能有描述信息就好了。而 XML 格式和 JSON 格式带有描述信息，这叫做“自描述”的结构化文档。

上面的留言条写成 XML 格式如下：

```
<?xml version = "1.0" encoding = "UTF - 8"?>
<note>
  <to>云龙同学</to>
  <content>你好!\n 今天上午,我到你家来想向你借一本《小学生常用成语词典》。可是不巧,你不在。我准备晚上 6 时再来借书。请你在家里等我,谢谢!</content>
  <from>关东升</from>
  <date>2012 年 12 月 08 日</date>
</note>
```

其中尖括号内容(<to>…</to>等)就是描述数据的标识,在 XML 中称为“标签”。

上面的留言条写成 JSON 格式如下:

```
{to:"云龙同学",content:"你好!\n 今天上午,我到你家来想向你借一本《小学生常用成语词典》。可是不巧,你不在。我准备晚上 6 时再来借书。请你在家里等我,谢谢!,"from:"关东升",date:"2012 年 12 月 08 日"}
```

数据放置在大括号“{}”之中,每个数据项之前都有一个描述的名字(如 to 等),描述名字和数据项之间用冒号(:)分开。描述同样的信息会发现,一般来讲 JSON 所用的字节数要比 XML 少,这也是很多人喜欢采用 JSON 格式的主要原因,因此 JSON 也被称为“轻量级”的数据交换格式。接下来重点介绍 XML 和 JSON 数据交换格式。

Cocos2d-x Lua API 目前只提供了 JSON 解码与编码。本章重点介绍 JSON 格式。

17.5　JSON 数据交换格式

JSON (JavaScript Object Notation)是一种轻量级的数据交换格式。所谓的轻量级是与 XML 文档结构相比而言,描述项目字符少,所以描述相同的数据所需的字符个数要少,那么传输的速度就会提高而流量也会减少。

如果留言条采用 JSON 描述,大体上可以设计成为下面的样子:

```
{"to":"云龙同学",
"content": "你好!\n 今天上午,我到你家来想向你借一本《小学生常用成语词典》。可是不巧,你不在。我准备晚上 6 时再来借书。请你在家里等我,谢谢!",
"from": "关东升",
"date": "2012 年 12 月 08 日"}
```

在移动平台开发对流量要求尽可能少,对速度要求尽可能快,而轻量级的数据交换格式——JSON 就成为理想的数据交换语言。

17.5.1　文档结构

构成 JSON 的文档有两种结构:对象和数组。对象是“名称—值”对集合,它类似于 Objective-C 中的字典类型,数组是一连串元素的集合。

对象是一个无序的“名称—值”对集合,一个对象以“{”(左括号)开始,“}”(右括号)结束。每个“名称”后跟一个“:”(冒号),“名称—值”对之间使用“,”(逗号)分隔,JSON 对象语法表如图 17-7 所示。

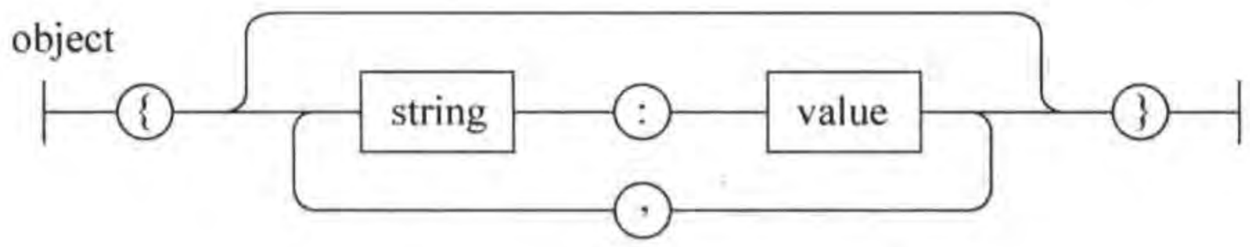


图 17-7　JSON 对象语言结构图

下面是一个 JSON 的对象例子：

```
{
"name":"a.htm",
"size":345,
"saved":true
}
```

数组是值的有序集合，一个数组以“[”(左中括号)开始，“]”(右中括号)结束，值之间使用“,”(逗号)分隔，JSON 数组语法表如图 17-8 所示。

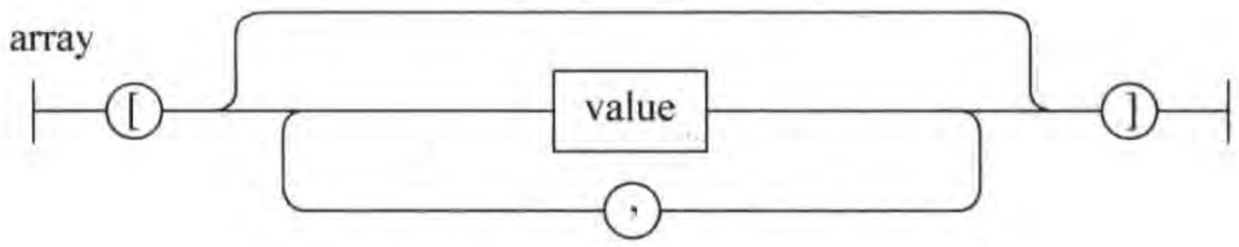


图 17-8　JSON 数组语言结构图

下面是一个 JSON 的数组例子：

```
["text","html","css"]
```

在数组中值可以是双引号括起来的字符串、数值、true、false、null、对象或者数组，而且这些结构可以嵌套，数组中值的 JSON 语法结构如图 17-9 所示。

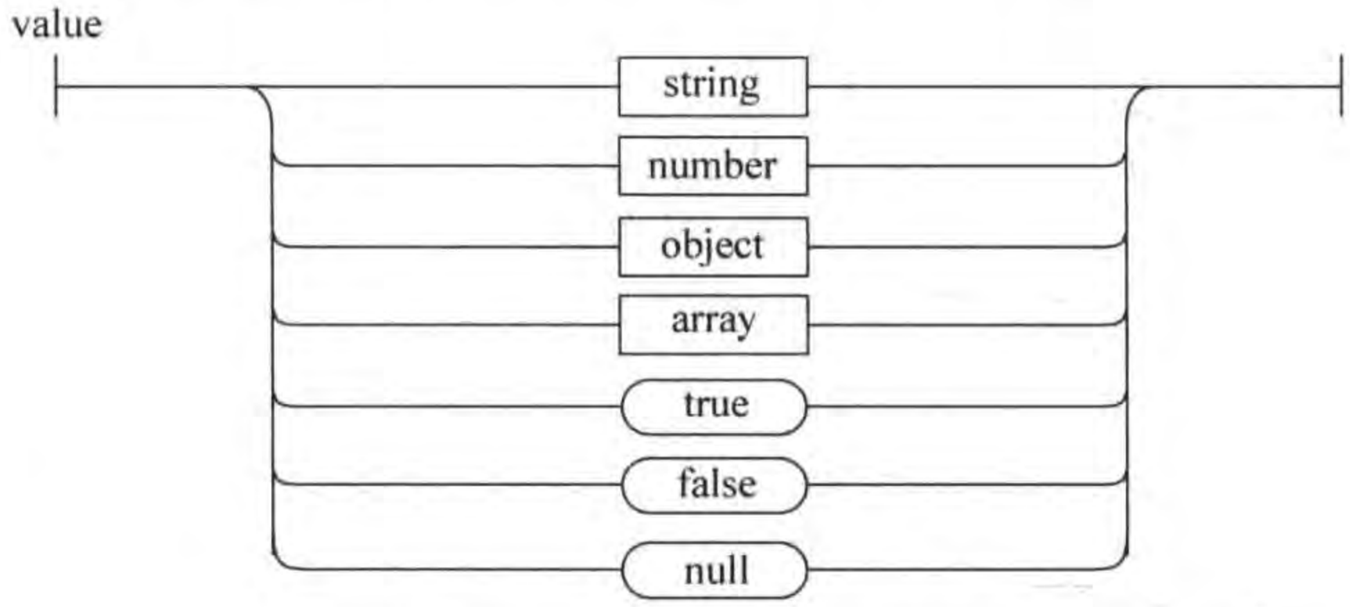


图 17-9　JSON 数值的语法结构图

17.5.2　JSON 解码与编码

Cocos2d-x Lua 封装了 JSON4Lua 库(http://json.luaforge.net/)，提供了 JSON 解码与编码相关函数。

1. JSON 解码

JSON 解码主要是通过 decode 函数实现的，由于 JSON 有对象和数组之分，因此解码 JSON 字符串时，返回的结果有可能是 JSON 对象或 JSON 数组。下面的代码是解码 JSON 字符串返回 JSON 对象的示例：

```
local jsonStr = '{"ID":"1","CDate":"2012-12-23","Content":"发布 iOSBook0"}'          ①
local jsonObj = json.decode(jsonStr)                                                  ②
cclog("ID : %s", jsonObj["ID"])
cclog("CDate : %s", jsonObj["CDate"])
cclog("Content : %s", jsonObj["Content"])
```

日志输出结果是：

```
[LUA-print] id : 1
[LUA-print] CDate : 2012-12-23
[LUA-print] Content : 发布 iOSBook0
```

上述第①行代码是定义 JSON 字符串 jsonStr。第②行代码是解码 jsonStr 字符串，如果成功则返回 JSON 对象。

下面的代码是解码 JSON 字符串返回 JSON 数组的示例：

```
local jsonStr = '[{"ID":"1","CDate":"2012-12-23","Content":"发布 iOSBook0"},
                  {"ID":"2","CDate":"2012-12-24","Content":"发布 iOSBook1"}]'      ①

local jsonArray = json.decode(jsonStr)                                              ②
for i=1, table.getn(jsonArray) do                                                   ③
    local jsonObj = jsonArray[i]                                                    ④
    cclog("**************************")
    cclog("ID : %s", jsonObj["ID"])
    cclog("CDate : %s", jsonObj["CDate"])
    cclog("Content: %s", jsonObj["Content"])
end
```

日志输出结果是：

```
[LUA-print] **************************
[LUA-print] ID : 1
[LUA-print] CDate : 2012-12-23
[LUA-print] Content : 发布 iOSBook0
[LUA-print] **************************
[LUA-print] ID : 2
[LUA-print] CDate : 2012-12-24
[LUA-print] Content : 发布 iOSBook1
```

上述第①行代码是定义 JSON 字符串 jsonStr。第②行代码是解码 jsonStr 字符串，如果成功则返回 JSON 数组。第③行代码 for i=1，table.getn(jsonArray) do 是循环遍历 JSON 数组。第④行代码 local jsonObj = jsonArray[i]是取出 JSON 数组中的一个元素，而 JSON 数组中的元素又是 JSON 对象类型。

2. JSON 编码

JSON 编码是将 JSON 对象或 JSON 数组转变为 JSON 字符串解析，以便于存储和网

络中数据的传输。JSON 编码主要是通过 json. encode(jsonObj)函数实现的。下面的代码是 JSON 对象和数组编码,返回 JSON 字符串示例:

```
local jsonObj = {ID = "1", CDate = "2012 - 12 - 23", Content = "发布 iOSBook0"}                ①
cclog("JSON Object : % s", json.encode(jsonObj))                                              ②

local jsonArray = [{ID = "1", CDate = "2012 - 12 - 23", Content = "发布 iOSBook0"},
                   {ID = "2",CDate = "2012 - 12 - 24",Content = "发布 iOSBook1"}]             ③
cclog("JSON Array : % s", json.encode(jsonArray))                                             ④
```

上述第①行代码是创建并初始化 JSON 对象,它是放在{…}括起来,这说明 jsonObj 表示的是 JSON 对象。第②行代码中 json. encode(jsonObj)是进行 JSON 编码,即返回的是 JSON 字符串。

第③行代码是创建并初始化 JSON 数组,它的右值前后使用[…]括起来,这是 JSON 数组与 JSON 对象的区别。第④行代码中 json. encode(jsonArray)是进行 JSON 编码,即返回的是 JSON 字符串。

日志输出结果是:

```
[LUA - print] JSON Object : {"ID":"1","CDate":"2012 - 12 - 23","Content":"发布 iOSBook0"}
[LUA - print] JSON Array : [{"ID":"1","CDate":"2012 - 12 - 23","Content":"发布 iOSBook0"},
                            {"ID":"2","CDate":"2012 - 12 - 24","Content":"发布 iOSBook1"}]
```

从日志结果可以看出 JSON 对象和 JSON 数组与 JSON 字符串的区别。

17.5.3 实例: 完善 MyNotes

在 Cocos2d-x Lua 中实现 JSON 解码,17.3.2 节的 MyNotes 实例只是从服务器端返回 JSON 数据,但是没有对这些 JSON 数据进行解析。本节就来完善这个实例。

GameScene. lua 中的 Read Data 菜单查询代码如下:

```
--查询数据函数
local function onMenuReadCallback(pSender)
    cclog("onMenuReadCallback")
    ...
    local function onReadyStateChange()
        if xhr.readyState == 4 and xhr.status == 200 then
            local response = xhr.responseText
            cclog(response)
            local jsonObj = json.decode(response)                                             ①
            local resultCode = jsonObj['ResultCode']                                          ②
            if resultCode == 0 then
                cclog("read success.")
                local jsonArray = jsonObj['Record']                                           ③
                for i = 1, table.getn(jsonArray) do                                           ④
                    cclog("---------------- [ % d] ----------------",i)
                    local row = jsonArray[i]                                                  ⑤

                    cclog("ID : % s", row["ID"])                                              ⑥
                    cclog("CDate : % s", row["CDate"])                                        ⑦
                    cclog("Content : % s", row["Content"])                                    ⑧
```

```
                cclog("selectedRowId = %s", selectedRowId)
                selectedRowId = row["ID"]                                    ⑨
            end
        else
            cclog(getErrorMessage(resultCode))
        end
    end
end
xhr:registerScriptHandler(onReadyStateChange)
xhr:send()
```

上述第①行代码开始解码从服务器返回的 JSON 字符串，这个字符串内容如下：

```
{"ResultCode":0,"Record":[
{"ID":"1","CDate":"2012-12-23","Content":"发布 iOSBook0"},
{"ID":"2","CDate":"2012-12-24","Content":"发布 iOSBook1"},
{"ID":"3","CDate":"2012-12-25","Content":"发布 iOSBook2"},
{"ID":"4","CDate":"2012-12-26","Content":"发布 iOSBook3"},
{"ID":"5","CDate":"2012-12-27","Content":"发布 iOSBook4"}]}
```

第②行代码 local resultCode = jsonObj['ResultCode']是取 ResultCode 键对应的数值，如果 ResultCode 大于等于 0 说明操作成功，ResultCode 小于 0 说明操作失败，−1～−7 定义了不同的错误编号。

第③行代码是通过 Record 键获得从服务器返回的记录集，它是一个 JSON 数组。第④行代码是遍历 JSON 数组。第⑤行代码 local row = jsonArray[i]是通过索引获得记录。第⑥～⑧行代码是通过键返回 JSON 对象中的值。第⑨行代码 selectedRowId = row["ID"]是取出 ID 字段的数据赋值给 selectedRowId 全局变量，需要注意的是，循环结束后 selectedRowId 保存最后一条记录的 ID 字段的数据。

当 ResultCode 小于 0 时，通过 getErrorMessage(resultCode)函数将−1～−7 错误编号翻译为错误消息返回，该函数代码如下：

```
--- 获得错误消息
--@param resultCode 错误编码
--@return 错误消息
local function getErrorMessage(resultCode)
    local errorStr = ""
    if resultCode == -7 then
        errorStr = "没有数据."
elseif resultCode == -6 then
    errorStr = "日期没有输入."
elseif resultCode == -5 then
    errorStr = "内容没有输入."
elseif resultCode == -4 then
    errorStr = "ID 没有输入."
elseif resultCode == -3 then
    errorStr = "数据访问失败."
elseif resultCode == -2 then
    errorStr = "您的账号最多能插入 10 条数据."
```

```
    elseif resultCode == -1 then
        errorStr = "用户不存在,请到 http://51work6.com 注册."
    end
    return errorStr
end
```

GameScene.lua 中的插入、删除和修改数据代码如下：

```
--插入数据函数
local function onMenuInsertCallback(pSender)
    cclog("onMenuInsertCallback")
    ...
    local function onReadyStateChange()
        if xhr.readyState == 4 and xhr.status == 200 then
            local response = xhr.responseText
            cclog(response)
            local jsonObj = json.decode(response)
            local resultCode = jsonObj['ResultCode']
            if resultCode > 0 then
                cclog("insert success.")
            else
                cclog(getErrorMessage(resultCode))
            end
        end
    end
    xhr:registerScriptHandler(onReadyStateChange)
    xhr:send(data)
end

--删除数据函数
local function onMenuDeleteCallback(pSender)
    cclog("onMenuDeleteCallback")
    ...
    local function onReadyStateChange()
        if xhr.readyState == 4 and xhr.status == 200 then
            local response = xhr.responseText
            cclog(response)
            local jsonObj = json.decode(response)
            local resultCode = jsonObj['ResultCode']
            if resultCode > 0 then
                cclog("delete success.")
            else
                cclog(getErrorMessage(resultCode))
            end
        end
    end
    xhr:registerScriptHandler(onReadyStateChange)
    xhr:send(data)
end

--更新数据函数
local function onMenuUpdateCallback(pSender)
    cclog("onMenuUpdateCallback")
    ...
```

```
local function onReadyStateChange()
    if xhr.readyState == 4 and xhr.status == 200 then
        local response = xhr.responseText
        cclog(response)
        local jsonObj = json.decode(response)
        local resultCode = jsonObj['ResultCode']
        if resultCode > 0 then
            cclog("update success.")
        else
            cclog(getErrorMessage(resultCode))
        end
    end
    xhr:registerScriptHandler(onReadyStateChange)
    xhr:send(data)
end
```

上述代码与17.3.2节相比基本上没有太多的变化，这里不再赘述。

本章小结

通过对本章的学习，读者可以熟悉基于HTTP网络通信技术。重点需要掌握XMLHttpRequest对象实现HTTP网络通信，熟悉JSON数据交换格式的解码和编码。

第 18 章 Node. js 与 WebSocket 网络通信

在前一章介绍了基于 HTTP 的网络通信，服务器一般会使用 Apache HTTP Server (Apache)和微软的 Internet 信息服务器(internet information services，IIS)等。服务器端的技术包括 Java Servlet/JSP、ASP. NET 和 PHP。一方面 Apache 和 IIS 这些 HTTP 服务器配置起来比较复杂，另外一方面服务器端技术 Java Servlet/JSP、ASP. NET 和 PHP 学习成本很高。

本章给大家推荐更加轻便的通信方式 WebSocket，它是 HTML5 新引入的技术，允许后台随时向前端发送文本或者二进制消息。此外，随着 Node. js 技术蓬勃发展，WebSocket 通信越来越受到广大开发人员的青睐。

在本章中介绍了 Cocos2d-x Lua API 中关于 WebSocket 的通信客户端技术，通过 Node. js 实现 WebSocket 服务器端，以及在服务器端如何访问 SQLite3 数据库。

18. 1 Node. js

2009 年 2 月，Ryan Dahl 在博客上宣布准备编写一个基于 Google V8 JavaScript 引擎[①]，轻量级的 Web 服务器并提供配套库。2009 年 5 月，Ryan Dahl 在 GitHub 上发布了最初版本的部分 Node. js 包。Node. js 使用 V8 引擎，并且对 V8 引擎进行优化，提高 Node. js 程序的执行速度。

Node. js 是一个事件驱动服务端 JavaScript 环境，只要能够安装相应的模块包，就可以开发出需要的服务器端程序，例如 HTTP 服务器端程序、Socket 程序等。

18. 1. 1 Node. js 安装

Node. js 安装包括 Node. js 运行环境安装和 Node. js 模块包管理。首先安装 Node. js 运行环境，该环境在不同的平台下安装文件也不同，可以在 http://nodejs. org/download/

---

① V8 是 Google 开发的开源 JavaScript 引擎，用于 Google Chrome 中。V8 在运行之前将 JavaScript 编译成了机器码，而非字节码或是解释执行它，以此提升性能，它是目前最快的 JavaScript 引擎。基于它的开源和免费很多人移植到自己的平台，这样就可以执行 JavaScript 了，这样安装了 V8 引擎的 JavaScript 程序执行速度大大提升。

页面找到需要下载的安装文件，目前 Node.js 运行环境支持 Windows、Mac OS X、Linux 和 SunOS 等系统平台。由于笔者的电脑是 Windows 8 64 系统，所以下载的是 node-v0.10.26-x64.msi 文件，下载完成进行安装就可以了。

安装完成后需要确认一下 Node.js 的安装路径（C:\Program Files\nodejs\）是否添加到系统 Path 环境变量中，需要打开图 18-1 所示的对话框，在系统变量 Path 中查找是否有这个路径。

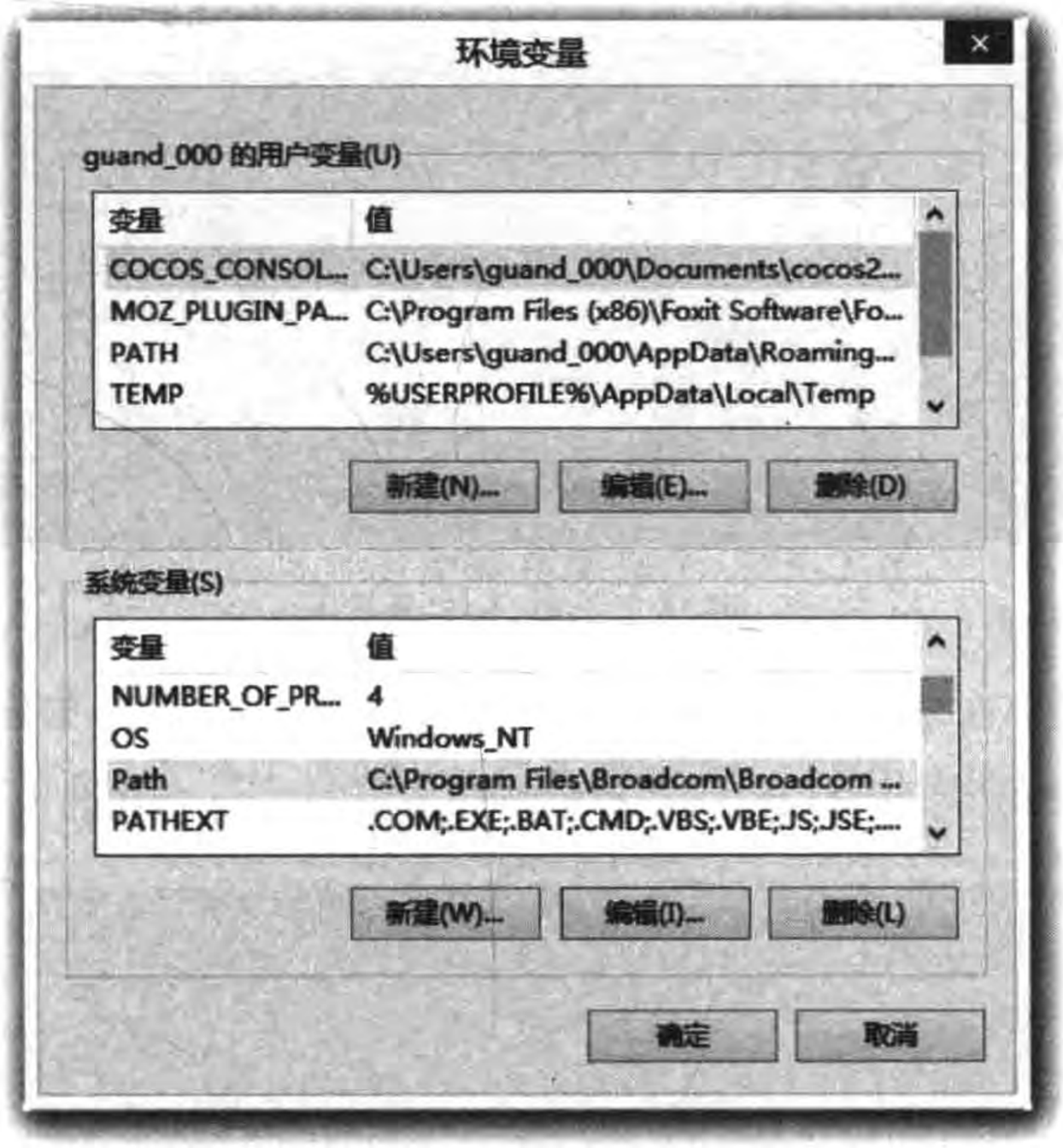


图 18-1　系统变量 Path 配置

18.1.2　Node.js 测试

现在编写一个最简单的 HTTP 服务器程序，客户端通过浏览器请求该服务器，并在浏览器页面中显示 Hello World。服务器端的 app.js 代码如下：

```
var http = require('http');
http.createServer(
    function (req, res) {
        res.writeHead(200, {
            'Content-Type': 'text/plain'
        });
        res.end('Hello World\n');
    }).listen(3000, '127.0.0.1');
console.log('Server running at http://127.0.0.1:3000/');
```

上述 app.js 代码暂且不介绍，它主要完成的任务是在本机上监听 3000 端口，通过

HTTP 服务向客户端输出 Hello World 字符串。

app.js 可以使用任何文本编辑器编辑和修改,但是保存时最好保存为 UTF-8 字符集,这样可以防止代码里的中文等字符乱码。运行时,通过 DOS 终端窗口进入 app.js 文件所在的目录,然后运行指令:

```
node app.js
```

启动界面如图 18-2 所示。

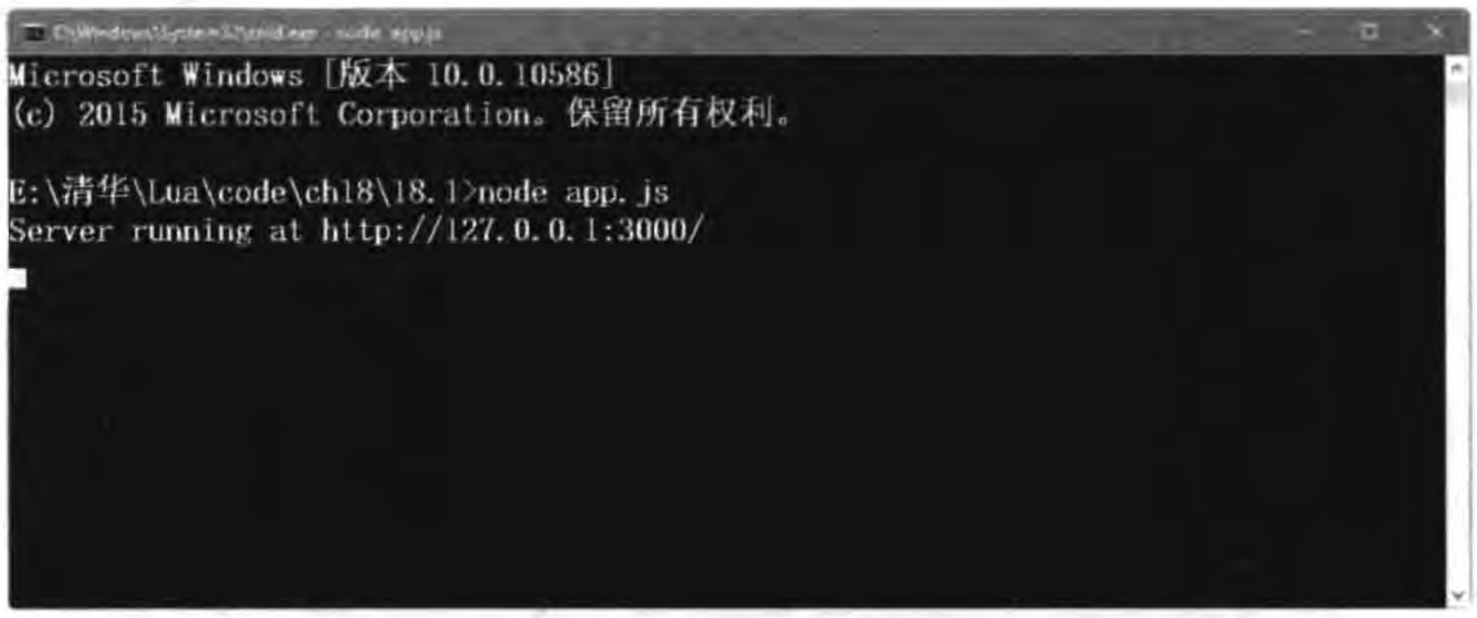


图 18-2　app.js 启动界面

然后打开浏览器,在地址栏中输入 http://127.0.0.1:3000/,如图 18-3 所示,会看到“Hello World”显示在页面中。

图 18-3　在浏览器中请求服务器

到此为止就成功地安装和配置了 Node.js 运行环境。要想让 Node.js 处理更多的工作,还需要安装模块包,模块包的安装将会在下一节介绍。

18.2　使用 WebSocket

WebSocket 是一种全新的协议,不属于 HTTP 无状态协议,协议名为 ws,这意味着一个 WebSocket 地址会是这样的写法: ws://51work6.com。在 HTML5 规范中,WebSocket

的使用需要服务器与浏览器同时支持才能正常运行。

在 Cocos2d-x 和 Cocos2d-x Lua API 移植了 WebSocket 客户端所需要的 WebSocket 库，而服务器端可以选择 Node.js 提供的 WebSocket 库。这样就可以在 Cocos2d-x 下基于 WebSocket 协议构建 C/S 网络通信了，如图 18-4 所示。

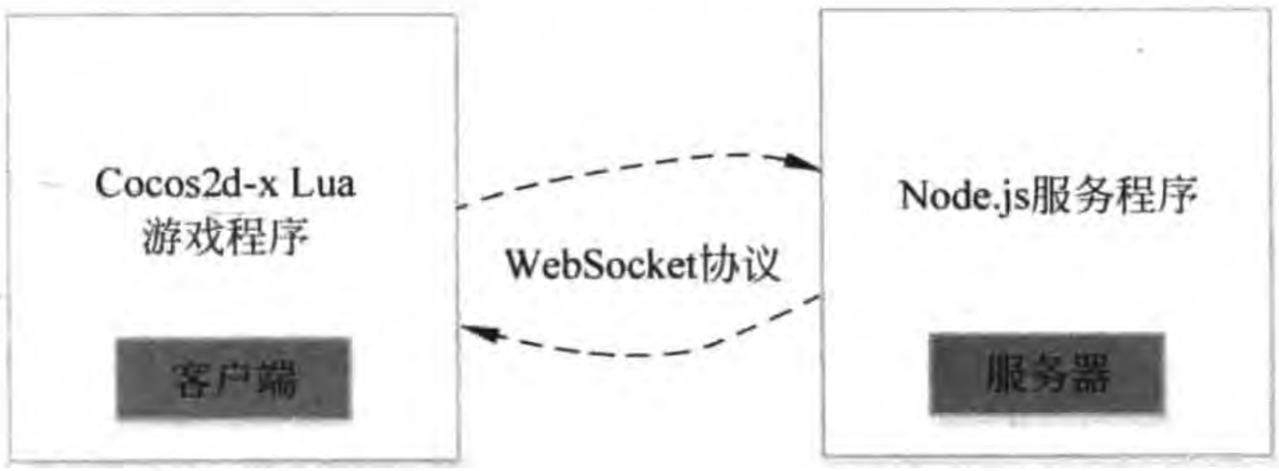


图 18-4　WebSocket 协议下的 C/S 网络通信

18.2.1　使用 Node.js 开发 WebSocket 服务器端程序

要想使用 Node.js 开发 WebSocket 服务器端程序，需要安装 Node.js 模块 WebSocket 模块包。在 Node.js 中模块包的管理使用 npm 指令，安装 WebSocket 模块包指令如下：

```
npm install ws
```

该命令需要在 DOS 终端窗口 js 文件所在的目录下运行，运行过程如图 18-5 所示。

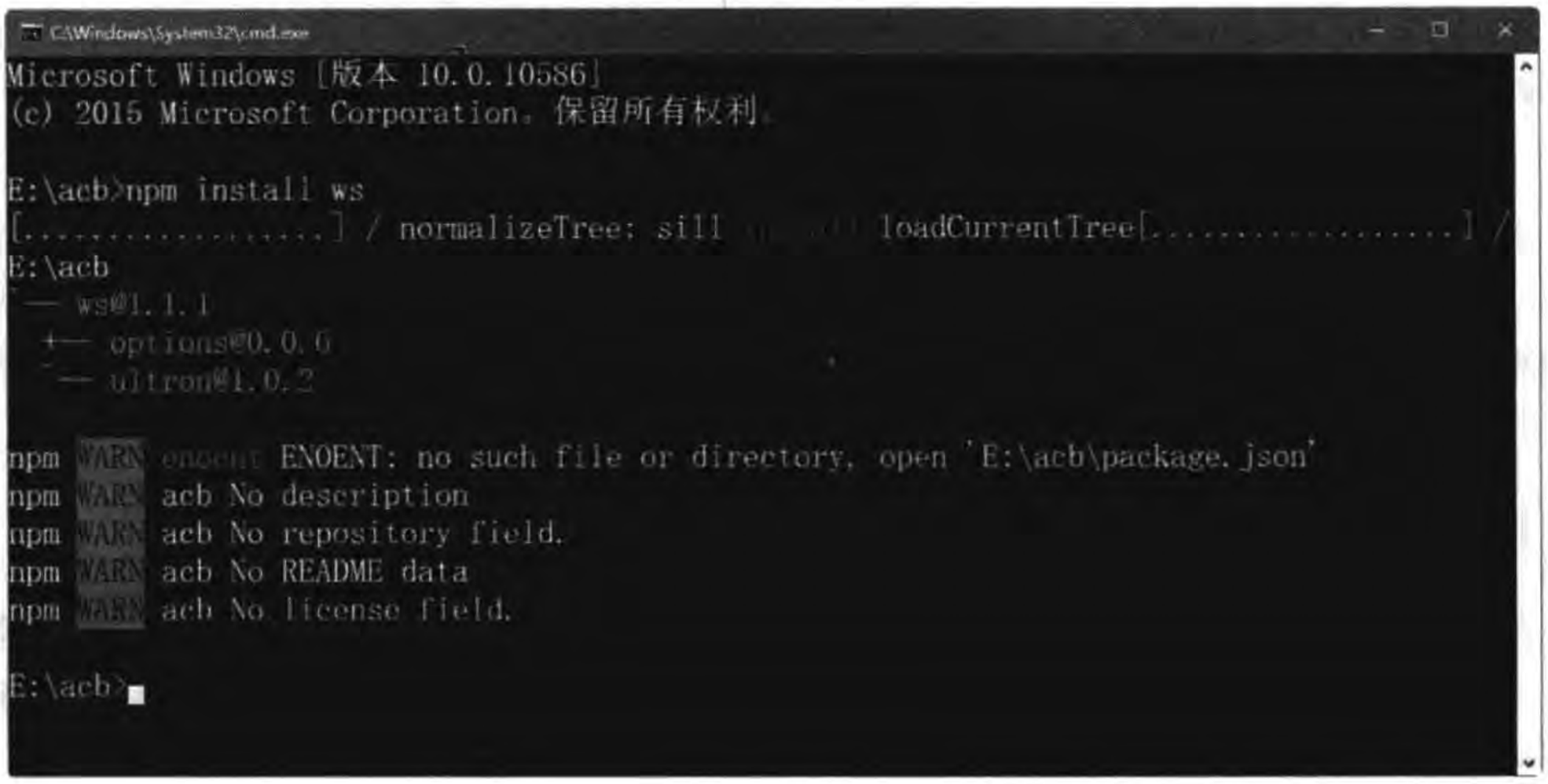


图 18-5　安装 WebSocket 模块包

为了掌握 WebSocket 服务器启动具体的开发细节，下面介绍一个最简单的 WebSocket 案例，这个案例是客户端与服务器双向通信，当客户端与服务器建立连接时，服务器会把 Hello Client 字符串消息发送回客户端。客户端也可以触发服务器端的事件，把 Hello

Server 字符串消息发送给服务器端。

服务器端的 app.js 代码如下：

```
var WebSocketServer = require('ws').Server;                                    ①

var wss = new WebSocketServer({port: 3000});                                   ②

console.log('Server on port 3000.');                                           ③

wss.on('connection', function(ws) {                                            ④
    //向客户端发送消息
    ws.send('Hello Cocos2d-x Lua');                                            ⑤
    //注册接收消息
    socket.on('message', function (data){                                      ⑥
        console.log(data);                                                     ⑦
    });
});
```

上述第①行代码 var WebSocketServer = require('ws').Server 引入 WebSocket 模块。第②行代码是监听 3000 端口。第③行代码中的 console.log 函数是进行日志输出的。第④行代码 wss.on('connection', function(ws) {…}) 是 WebSocketServer 与客户端 WebSocket 建立连接时触发的，其中 wss 是 WebSocketServer 对象，参数 ws 是 WebSocket 对象。

WebSocket 事件包括：

(1) open。连接打开事件。

(2) message。接收消息事件。

(3) close。连接关闭事件。

(4) error。错误发生事件。

上述第⑤行代码 ws.send('Hello Cocos2d-x Lua')是向客户端发送消息，它会触发客户端 message 事件，如果是 Cocos2d-x Lua API 客户端则是 WEBSOCKET_MESSAGE 事件。第⑥行代码 socket.on('message', function (data){…})是注册接收客户端 message 事件。第⑦行代码 console.log(data)是接收客户端后输出日志消息内容。

18.2.2 Cocos2d-x Lua API 客户端

在 Web 应用开发过程中，WebSocket 客户端是使用 JavaScript 语言开发的，运行环境是在浏览器中运行。Cocos2d-x Lua API 提供了 WebSocket 客户端开发所需库，这样在游戏中开发网络通信的相关应用就变得比较简单了。

GameScene.lua 中初始化代码如下：

```
local wsSendText    = nil                                                      ①
…
-- create layer
function GameScene:createLayer()
```

```
local layer = cc.Layer:create()

...

--////////////////// WebSocket start /////////////////
wsSendText   = cc.WebSocket:create("ws://127.0.0.1:3000")                    ②

local function wsSendTextOpen(strData)                                       ③
    cclog("WebSocket 实例打开")
end

local function wsSendTextMessage(strData)                                    ④
    local strInfo = "response text msg: "..strData
    cclog(strInfo)
end

local function wsSendTextClose(strData)                                      ⑤
    cclog("WebSocket 实例关闭")
end

local function wsSendTextError(strData)                                      ⑥
    cclog("WebSocket 错误发生")
end

wsSendText:registerScriptHandler(wsSendTextOpen,cc.WEBSOCKET_OPEN)           ⑦
wsSendText:registerScriptHandler(wsSendTextMessage,cc.WEBSOCKET_MESSAGE)     ⑧
wsSendText:registerScriptHandler(wsSendTextClose,cc.WEBSOCKET_CLOSE)         ⑨
wsSendText:registerScriptHandler(wsSendTextError,cc.WEBSOCKET_ERROR)         ⑩
--////////////////// WebSocket end /////////////////

    return layer
end
```

上述第①行代码是声明一个 WebSocket 类型的变量 wsSendText，它的作用域范围是 GameScene.lua 文件内部。第②行代码 cc.WebSocket:create("ws://127.0.0.1:3000")是实例化 wsSendText 对象，其中 ws://127.0.0.1:3000 表示 ws 是 WebSocket 协议，主机 IP 是 127.0.0.1，端口是 3000。

为了监听 WebSocket 的各种事件，需要通过第⑦～⑩行代码注册 WebSocket 事件，其中第⑦行代码是注册 WEBSOCKET_OPEN 事件，当事件触发时则调用第③行代码的 wsSendTextOpen 函数。第⑧行代码是注册 WEBSOCKET_MESSAGE 事件，当事件触发时则调用第④行代码的 wsSendTextMessage 函数。第⑨行代码是注册 WEBSOCKET_CLOSE 事件，当事件触发时则调用第⑤行代码的 wsSendTextClose 函数。第⑩行代码是注册 WEBSOCKET_ERROR 事件，当事件触发时则调用第⑥行代码的 wsSendTextError 函数。

GameScene.lua 中的 Send Message 菜单项回调函数代码如下：

```
local function OnClickMenu1(menuItemSender)                                  ①
    if cc.WEBSOCKET_STATE_OPEN == wsSendText:getReadyState() then            ②
```

```
            cclog("Send Text WS is waiting...")
            wsSendText:sendString("Hello WebSocket")                    ③
        else
            local warningStr = "WebSocket 实例还没有准备好!"
            cclog(warningStr)
        end
end

local pItmLabel1 = cc.Label:createWithBMFont("fonts/fnt8.fnt", "Send Message")
local pItmMenu1 = cc.MenuItemLabel:create(pItmLabel1)
pItmMenu1:registerScriptTapHandler(OnClickMenu1)

local mn = cc.Menu:create(pItmMenu1)
mn:alignItemsVertically()
layer:addChild(mn)
```

上述第①行代码定义的函数 OnClickMenu1 是单击 Send Message 菜单项回调的函数，第②行代码是判断 WebSocket 是否为打开状态，如果是则通过第③行代码 wsSendText:sendString("Hello WebSocket")向服务器发送消息，这个动作会触发服务器的 message 事件。

下面看看整个实例的运行情况。首先需要启动服务器，通过 DOS 终端窗口进入 app.js 文件所在的目录，运行指令：node app.js，启动服务器。

客户端与服务器交换分为两个过程：Cocos2d-x Lua API 客户端启动和单击 Send Message 菜单。

1. Cocos2d-x Lua API 客户端启动

启动 Cocos2d-x Lua API 客户端程序，场景启动后的界面如图 18-6 所示。场景启动后与服务器建立连接。

图 18-6　Cocos2d-x Lua API 客户端场景启动后的界面

客户端日志输出结果：

```
[LUA - print] WebSocket 实例打开                                    ①
[LUA - print] response text msg: Hello Cocos2d - x Lua               ②
```

上面日志第①行是 WEBSOCKET _ OPEN 事件触发时输出的，第②行日志是 WEBSOCKET_MESSAGE 事件触发时输出的。

连接过程中服务器端没有日志信息输出。

2. 单击 Send Message 菜单项

客户端启动后，可以单击 Send Message 菜单项，向服务器发出"Hello WebSocket"消息。客户端相关的日志信息如下：

```
Hello WebSocket
```

上面的日志信息是在 app.js 如下代码中输出的。

```
ws.on('message', function (data){
    console.log(data);
});
```

本章小结

通过对本章的学习，读者可以了解基于 Node.js 的 WebSocket 网络通信技术。在本章中我们介绍了采用 Node.js 技术实现的服务器端技术。还介绍了 Cocos2d-x Lua API 提供的 WebSocket 客户端通信技术。

第四篇　优　化　篇

本篇只有 1 章，从多个方面例举了 Cocos2d-x 性能优化技术。

本篇内容如下：

第 19 章　性能优化

第 19 章

性 能 优 化

相对电脑而言，移动设备具有内存小、CPU 速度慢等缺点，而且游戏程序本身也是非常耗费内存的，因此移动开发人员需要尽可能优化游戏程序的性能。性能优化需要考虑的问题很多，本章就来介绍几个重要的优化方法。

19.1 合理使用缓存

为了提高性能，Cocos2d-x 提供几个缓存应用于不同的情况，这些缓存类有纹理缓存(TextureCache)、精灵帧缓存(SpriteFrameCache)、动画缓存(AnimationCache)和着色器缓存(ShaderCache)。前 3 个缓存在第 5 章做过基本用法的介绍。而着色器缓存是为着色器程序提供缓存，在 Open GL 2.0 之后每个渲染节点必须拥有它的着色器程序，对于开发人员很少使用着色器缓存。

在编程过程中缓存(主要是纹理缓存和精灵帧缓存)生命周期是非常重要的。程序什么时候创建缓存，什么时候清除缓存，都很有必要好好研究，麻烦的是不能“一刀切”，而是需要具体问题、具体分析、具体解决。

19.1.1 场景与资源

不同的场景中资源的占用不同，而资源的占用决定了创建缓存和清除缓存的时机。如图 19-1 所示是一个飞机大战游戏场景草图，主要有以下 5 个场景：

(1) 主菜单场景 HelloWorldScene。

(2) 游戏场景 GameScene。

(3) 游戏结束场景 GameOverScene。

(4) 设置场景 SettingScene。

(5) 帮助场景 HelpScene。

这些场景中玩家使用它们的时间并不是均等的。玩家几乎将所有的时间都花在游戏场景和结束场景中，而主菜单场景是进入游戏场景必须要经过的，设置场景和帮助场景可能很少光顾，或许一辈子都不会去帮助场景中查看帮助。

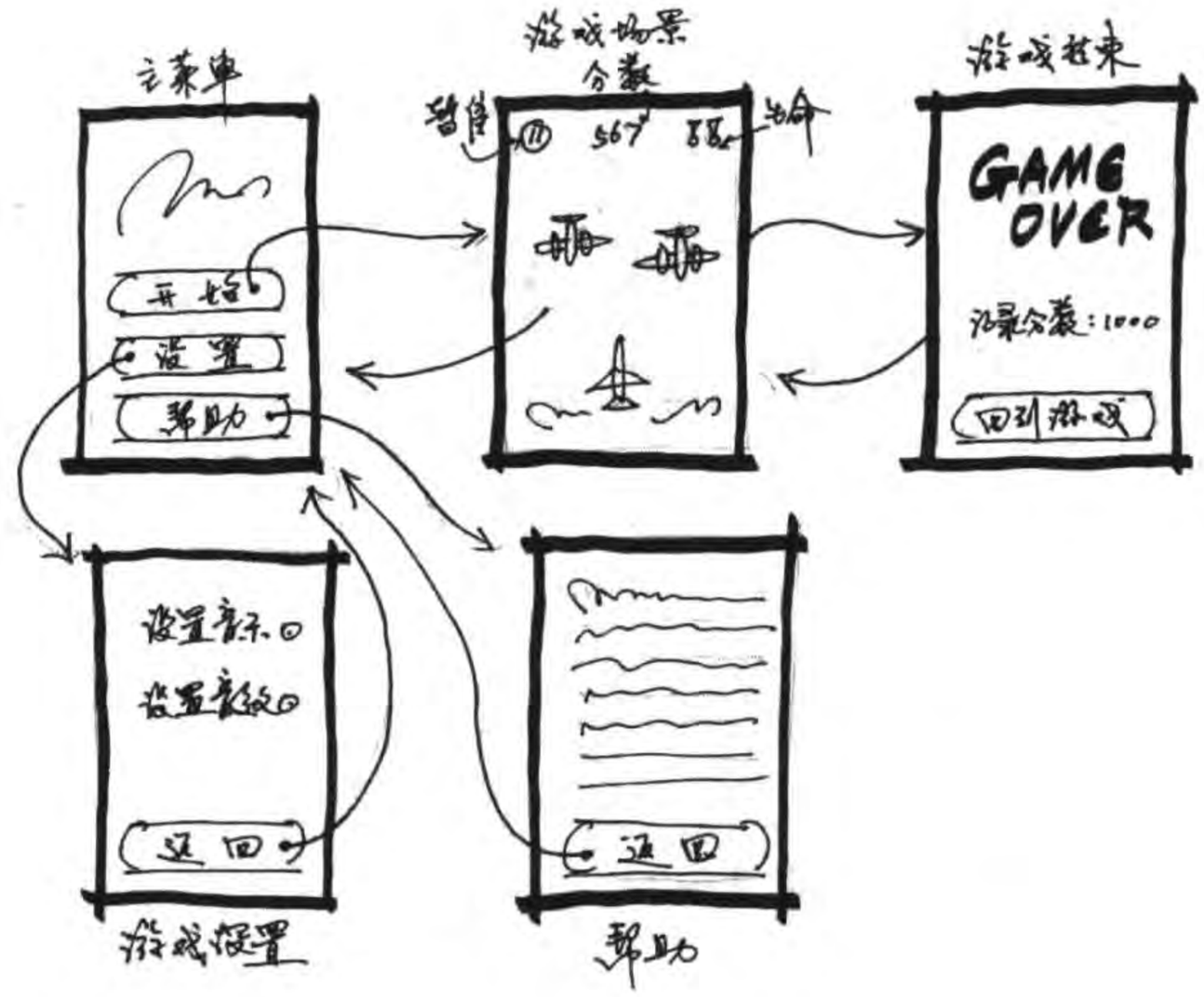


图 19-1　飞机大战游戏场景草图

场景中资源的占用见表 19-1，从中可以看出大体上的时间比例。图片加载到内存由纹理缓存管理，通过精灵帧缓存加载的图片最后也被放到纹理缓存中。玩家花费时间越多的场景（如游戏场景）中的缓存应该长时间保持，而很少光顾的场景（如设置场景和帮助场景）中缓存应该短时间保持，或者不用缓存。

表 19-1　场景中资源的占用

编　　号	游 戏 场 景	玩家花费时间	资　　源
1	主菜单场景	6%	背景、菜单等
2	游戏场景	80%	背景、菜单、飞机、敌人、子弹等
3	游戏结束场景	10%	背景、菜单等
4	设置场景	3%	背景、菜单等
5	帮助场景	1%	背景、菜单等

下面通过飞机大战游戏介绍一下缓存创建和缓存清除的时机。

19.1.2　缓存创建和清除时机

使用纹理和精灵帧缓存的时候，一个原则是“尽可能将所有资源加载到缓存”。但是这个命题正确的前提是：你的设备内存足够大。程序对内存的需求，就像人的欲望一样，永无止境，内存足够大是相对而言的。

事实上作为游戏开发人员，需要照顾那些配置比较低的设备，我们不可能采用“尽可能将所有资源加载到缓存”一刀切的方式，而是按照表 19-1 所示的情况适时地加载资源创建缓存，缓存分为短周期和长周期。

1. 短周期缓存

短周期缓存中的内容一方面玩家花费时间短，另外一方面不跨场景，没有必要长时间缓存。可以在该场景进入函数 onEnter()中创建缓存，在该场景退出函数 onExit()中清除。设置场景的示例代码如下：

```
function SettingScene:ctor()
    cclog("SettingScene init")
    -- 场景节点事件处理
    local function onNodeEvent(event)
        if event == "enter" then
            self:onEnter()
        elseif event == "enterTransitionFinish" then
            self:onEnterTransitionFinish()
        elseif event == "exit" then
            self:onExit()
        elseif event == "exitTransitionStart" then
            self:onExitTransitionStart()
        elseif event == "cleanup" then
            self:cleanup()
        end
    end

    self:registerScriptHandler(onNodeEvent)
end

function SettingScene:onEnter()
    cclog("SettingScene onEnter")
    cc.SpriteFrameCache:getInstance():addSpriteFrames("help/help.plist")

end

function SettingScene:onEnterTransitionFinish()
    cclog("SettingScene onEnterTransitionFinish")
end

function SettingScene:onExit()
    cclog("SettingScene onExit")
    cc.SpriteFrameCache:getInstance():removeSpriteFramesFromFile("help/help.plist")

end

function SettingScene:onExitTransitionStart()
    cclog("SettingScene onExitTransitionStart")
end

function SettingScene:cleanup()
    cclog("SettingScene cleanup")
end
```

这种短周期缓存通过 addSpriteFrames 函数添加帧时，不要在 ctor 函数(init 事件)中实现，因为 ctor 函数(init 事件)在返回到当前场景时不会调用。进入场景事件顺序如图 19-2 所示。

在 onExit()函数中还可以使用 removeSpriteFrameByName(name)函数清除特定名字的帧。退出场景事件顺序如图 19-3 所示。

注意 不要在 onExit()函数中使用 removeSpriteFrames()和 removeUnusedSpriteFrames()函数,removeSpriteFrames()会清除所有缓存里的帧,removeUnusedSpriteFrames()会清除所有缓存中未使用的帧,本场景中没有使用的帧不等于在其他场景中不用。总之,这两个函数在任何情况下都要慎用。

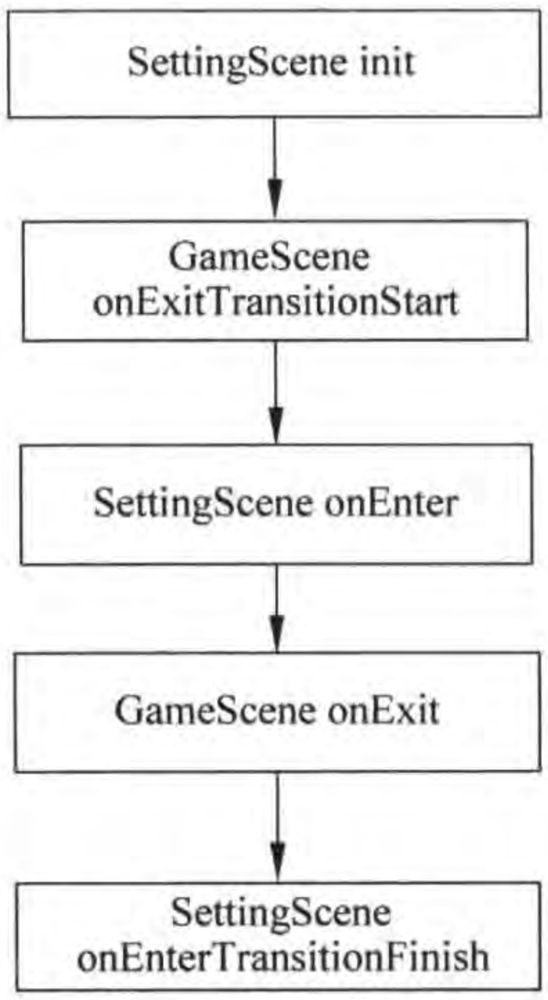


图 19-2 主菜单场景调用 pushScene 函数进入设置场景事件顺序

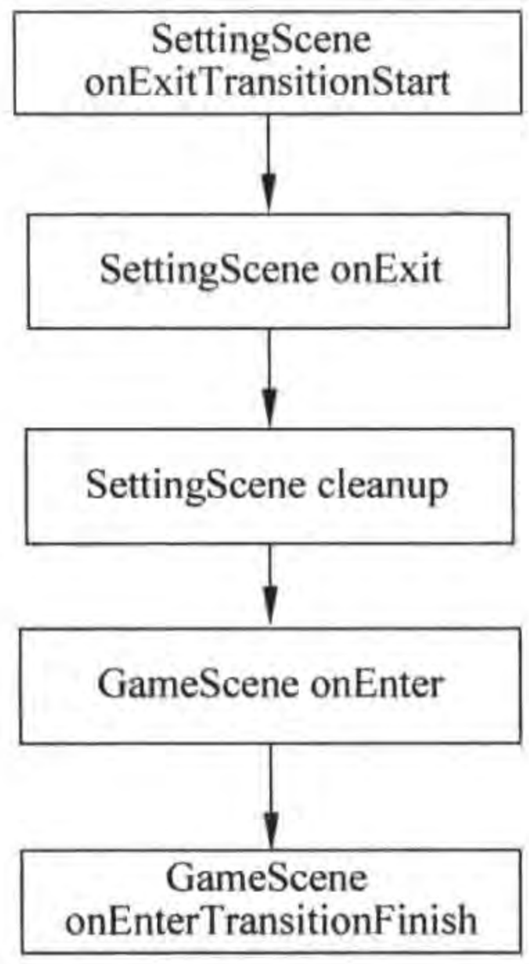


图 19-3 设置场景调用 popScene 函数返回主菜单场景事件顺序

另外,TextureCache 纹理缓存与 SpriteFrameCache 帧缓存具有类似的 API,使用原则也是一样的,这里不再赘述。

2. 长周期缓存

长周期缓存中的内容,一方面玩家花费时间长,另外一方面可能跨场景,为了获得流畅的用户体验,不能经常反复创建和清除缓存,而是长时间缓存。长周期缓存可以分两种情况:贯穿整个游戏长周期缓存和关卡长周期缓存。

1) 贯穿整个游戏长周期缓存

贯穿整个游戏长周期缓存是在游戏刚刚启动、给用户展示 Loading 等信息时创建的缓存,以飞机大战游戏为例,可以在主菜单场景的 ctor()构造函数中添加如下代码:

```
function HelloWorldScene:ctor()
    cclog("HelloWorldSceneinit")
    cc.SpriteFrameCache:getInstance():addSpriteFrames("home/home.plist")
    --场景节点事件处理
    local function onNodeEvent(event)
```

```
    if event == "enter" then
        self:onEnter()
    elseif event == "enterTransitionFinish" then
        self:onEnterTransitionFinish()
    elseif event == "exit" then
        self:onExit()
    elseif event == "exitTransitionStart" then
        self:onExitTransitionStart()
    elseif event == "cleanup" then
        self:cleanup()
    end

    self:registerScriptHandler(onNodeEvent)
end
```

还可以在 main.lua 的 main()函数中添加如下代码：

```
local function main()
    collectgarbage("collect")
    -- avoid memory leak
    collectgarbage("setpause", 100)
    collectgarbage("setstepmul", 5000)

    cc.FileUtils:getInstance():addSearchPath("src")
    cc.FileUtils:getInstance():addSearchPath("res")
    cc.Director:getInstance():getOpenGLView():setDesignResolutionSize(960, 640, 0)

    --create scene
    local scene = require("GameScene")
    local gameScene = scene.create()

    if cc.Director:getInstance():getRunningScene() then
        cc.Director:getInstance():replaceScene(gameScene)
    else
        cc.Director:getInstance():runWithScene(gameScene)
    end

    cc.SpriteFrameCache:getInstance():addSpriteFrames("home/home.plist")

end
```

贯穿整个游戏长周期缓存不需要考虑清除缓存问题。

2）关卡长周期缓存

关卡长周期缓存是在开始某个特定关卡等场景时创建的缓存。以飞机大战游戏为例，可以在游戏场景的 ctor()构造函数中添加如下代码：

```
cc.SpriteFrameCache:getInstance():addSpriteFrames("home/home.plist")
```

与贯穿整个游戏长周期缓存类似，一般不需要考虑清除缓存问题。如果在程序测试的过程中出现了内存问题，应该尽量考虑它的解决方案，比如改变纹理图片格式、清除其他的缓存等方式解决问题。如果这样处理后，加载这些纹理时还是有内存问题，建议将这些资源的加载改成短周期缓存方式。

提示 使用 addSpriteFrames 可以加载多个不同的纹理图集 plist 文件，建议在做纹理图集的时候把相关周期的精灵图片拼接在一起，以便于添加和清除。在 iOS 设备上加载单个纹理图集时图片大小也是有限制的，例如，iPod touch 4 中单个图片大小不能超过 2048×2048 像素，可以将这个场景中需要的精灵图片放在几个不同的纹理图集中，然后依次加载。

19.2 图片与纹理优化

在 2D 游戏中图片无疑是最为重要的资源文件，它会被加载到内存中转换为纹理，由 GPU 贴在精灵之上渲染出来。它能够优化的方面很多，包括图片格式、拼图和纹理格式等，下面从这几个方面介绍一下图片和纹理的优化。

19.2.1 选择图片格式

要回答这个问题，需要先了解一下目前在移动平台所使用的图片文件格式，以及这些图片文件格式在 Cocos2d-x Lua API 是否支持。图片文件格式有很多，在移动平台上主要推荐使用 PNG，JPG 也可以考虑，而其他的文件格式最好转化成 PNG 格式。首先了解一下它们的特点。

1. PNG 文件

PNG 文件格式设计目的是替代 GIF 和 TIFF 文件格式，是一种位图存储格式。PNG 采用无损压缩，可以有 Alpha 通道数据，支持透明，但不支持动画。PNG 可以保存高保真的较复杂的图像，但是文件比较大。PNG 格式具体又分为 PNG8 和 PNG24，后面的数字则是代表这种 PNG 格式最多可以索引和存储的颜色值。

2. JPG

JPG 全名是 JPEG。JPG 图片以 24 位颜色存储单个位图图形。JPG 是与平台无关的格式，支持最高级别的压缩，压缩比率可以高达 100∶1，这种压缩是以牺牲图像质量为代价，来换取更小的文件大小。JPG 不支持透明。JPG 比较支持摄影图像或写实图像作品，这是因为它们颜色比较丰富。而对于所含颜色很少、具有大块颜色相近的区域或亮度差异十分明显的较简单的图片，JPG 就不太适合了。

那么选择 JPG 还是 PNG 呢？很多人认为 JPG 文件比较小，PNG 文件比较大，加载到内存纹理，JPG 占有更少的内存。这种观点是错误的。纹理与图片是不同的两个概念，如果纹理是野营帐篷，那么图片格式就是收纳折叠后的帐篷袋子，装有帐篷的袋子大小，不能代表帐篷搭起来后的大小。特别是在 Cocos2d-x Lua API 平台，JPG 加载后被转化为 PNG 格式，然后再转换为纹理，保存在内存中，这样无形中增加了对 JPG 文件解码的时间，因此无论 JPG 在其他平台表现得多么不俗，但是在移动平台下它一定无法与 PNG 相提并论。

提示　在上文中提到了 Alpha 颜色通道，那么什么是颜色通道？颜色通道是保存图像颜色信息的通道。根据图像颜色模式的不同，颜色通道的种类也各异。例如，RGB 和 CMYK 图像颜色模式，RGB 图像默认有 3 个通道，即红(Red)、绿(Green)、蓝(Blue)。CMYK 图像默认有 4 个通道，分别为青色、洋红、黄色、黑色。这些通道按照一定的比例表示颜色，如图 19-4 所示。有时还会加上 Alpha 通道，Alpha 通道还可以表示透明度。

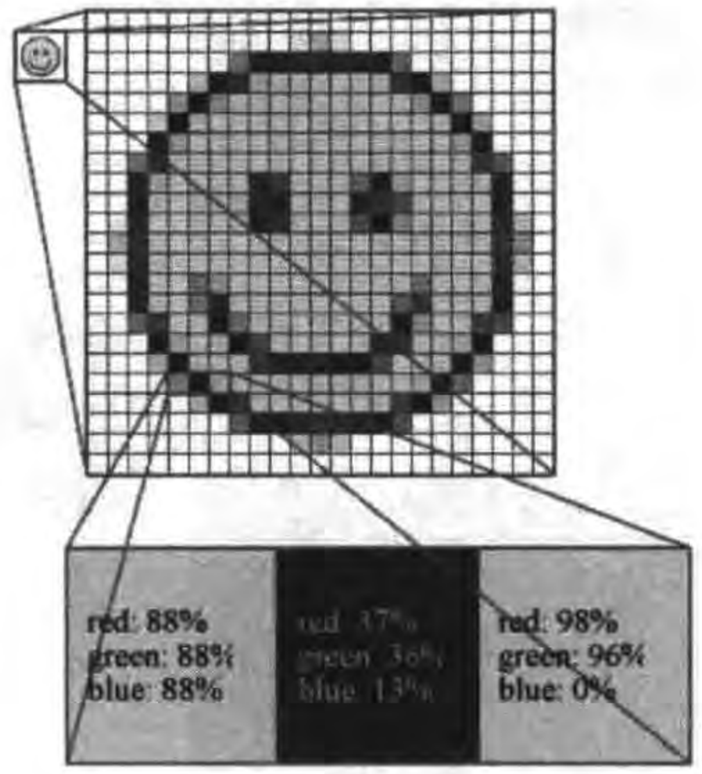


图 19-4　颜色通道

19.2.2　拼图

不知大家是否有过这样的一个疑问，为什么要把场景中小图片都拼接成一个大图片呢？这个问题在使用纹理图集时简单说了一下，这一节将详细介绍。

如果把多个小图拼接为一个大图(纹理图集或精灵表)，可以减少 IO 操作。如果使用散图每次都要针对一个图，创建精灵添加到纹理缓存，如果很频繁大量创建，CPU 和内存的开销会很高。而使用大图，则一次性创建精灵帧缓存，并把它们的纹理添加到纹理缓存中，这样会明显地提高效率。

在进行图片拼接的时候，如果能够满足用户的保真度，大图越小当然是越好。可以通过 TexturePacker 等纹理拼图工具，设置纹理支持 NPOT，这些工具的使用读者可以参考《Cocos2d-x 实战：工具卷》。

那么什么是 NPOT 呢？NPOT 是 non power of two 的缩写，意思是非 2 的 N 次幂。在 OpenGL ES 1.1 时纹理图片要求是 2 的 N 次幂(即 POT)，否则纹理无法创建。在 POT 要求下使用纹理工具拼接成的大图，会有很多的空白区域。如图 19-5 所示，右下角还有一些空白区域，造成了浪费，同时也会增加图片的大小，图 19-5 所示的图片大小是 2048KB。

OpenGL ES 2.0 支持 NPOT，不需要为图片是否为 2 的 N 次幂而苦恼，如图 19-6 所示，是采用 NPOT 拼图，整个图片基本上没有大的空白区域，能充分利用图片空间。图 19-6 所示的图片大小是 1822KB，与图 19-5 相比节省了 200KB。

提示　Cocos2d-x Lua API 本身已经支持了 NPOT，但是具体落实到设备上，还要看设备本身情况，苹果的 iOS 硬件设备比较统一，主流机型都能够支持 NPOT，但是 Android 平台由于碎片化很严重，因此无法确定所需要的设备是否能够很好地支持 NPOT。

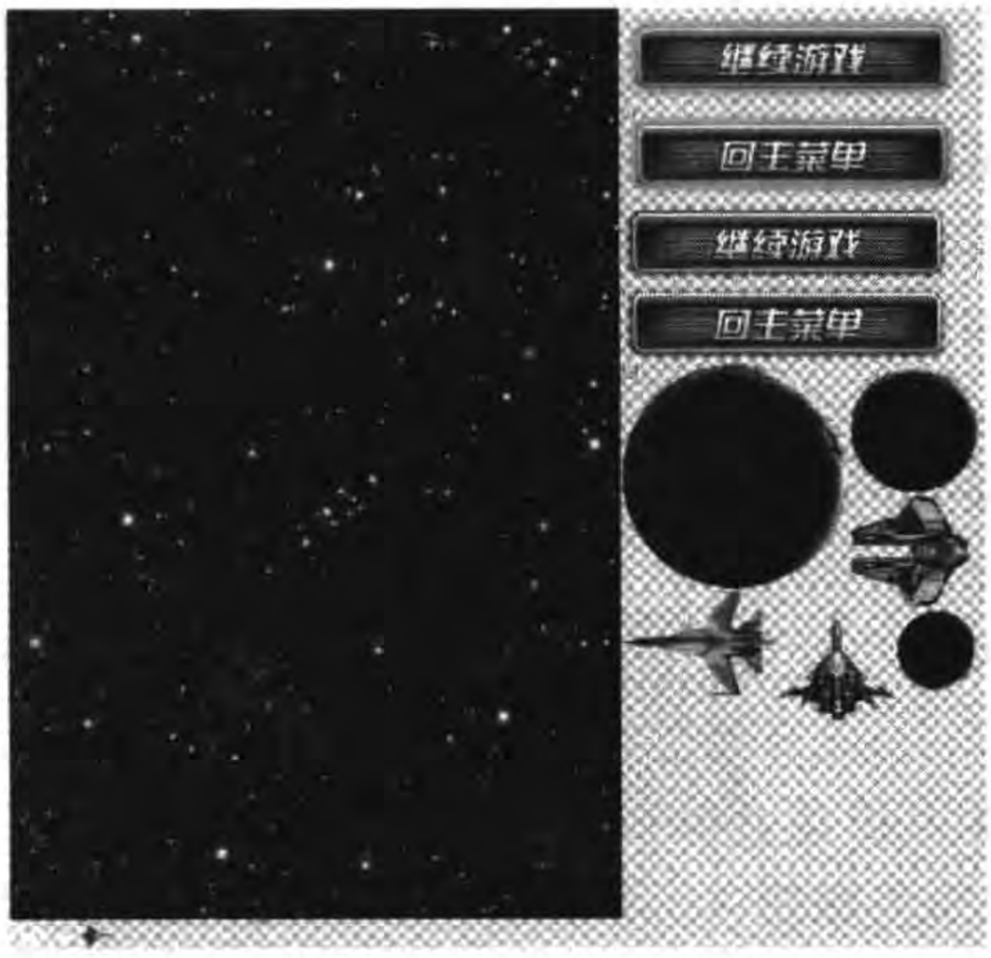


图 19-5　POT 拼图

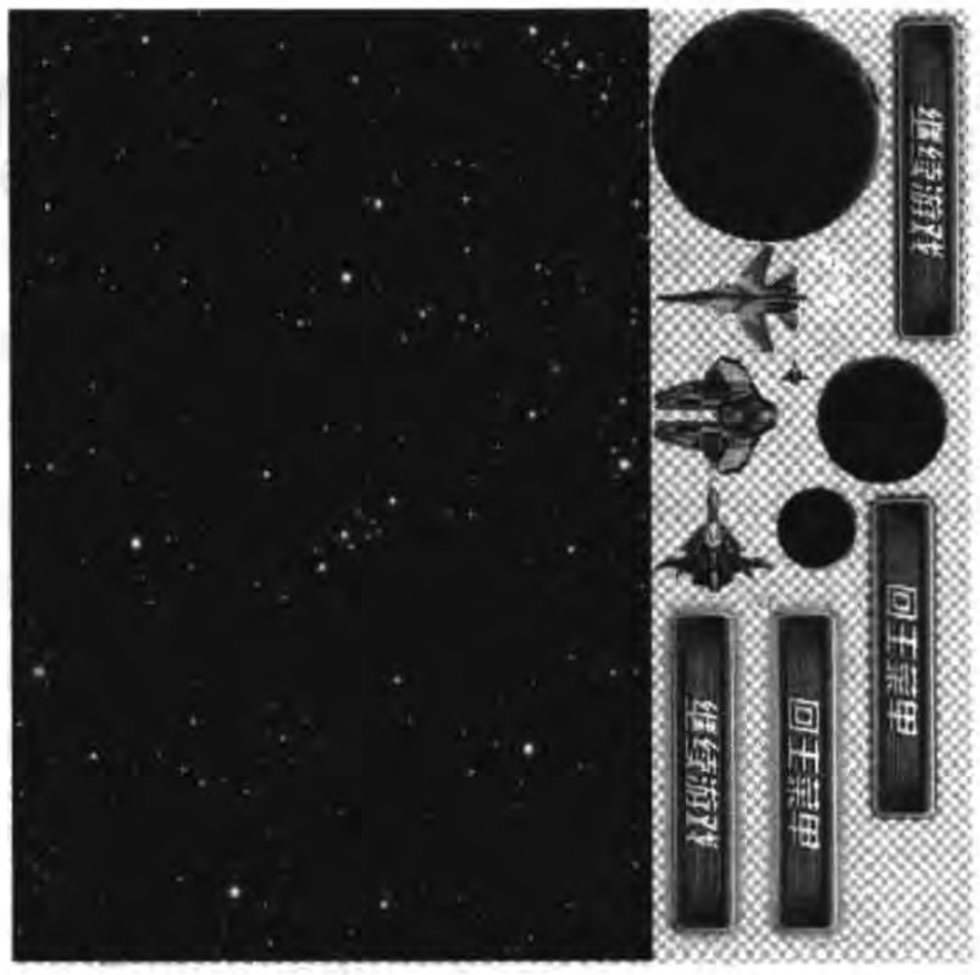


图 19-6　NPOT 拼图

19.2.3　纹理像素格式

纹理优化工作的另一重要指标是纹理像素格式，在能够最大限度满足用户对保真度要求的情况下，选择合适的像素格式，可以大幅提高纹理的处理速度。而且纹理像素格式与硬件有着密切的关系。

下面先了解一下纹理像素的格式，主要的格式有：

(1) RGBA8888。32 位色，它是默认的像素格式，每个通道 8 位(比特)，每个像素 4 个字节。

(2) BGRA8888。32 位色，每个通道 8 位(比特)，每个像素 4 个字节。

(3) RGBA4444。16 位色，每个通道 4 位(比特)，每个像素 2 个字节。

(4) RGB888。24 位色，没有 Alpha 通道，所以没有透明度。每个通道 8 位(比特)，每个像素 3 个字节。

(5) RGB565。16 位色，没有 Alpha 通道，所以没有透明度。R 和 B 通道是各 5 位，G 通道是 6 位。

(6) RGB5A1(或 RGBA5551)。16 位色，每个通道各 4 位，Alpha 通道只用 1 位表示。

(7) PVRTC4。4 位 PVR 压缩纹理格式，PVR 格式是专门为 iOS 设备上面的 PowerVR 图形芯片而设计的。它们在 iOS 设备上非常好用，可以直接加载到显卡上面，而不需要经过中间的计算转化。

(8) PVRTC4A。具有 Alpha 通道的，4 位 PVR 压缩纹理格式。

(9) PVRTC2。2 位 PVR 压缩纹理格式。

(10) PVRTC2A。具有 Alpha 通道的，2 位 PVR 压缩纹理格式。

此外，PVR 格式在保存的时候还可以采用 Gzip 和 Zlib 压缩格式进行压缩，对应的保存

文件为 pvr.gz 和 pvr.ccz。经过压缩文件会更小，加载时使用更少的内存。虽然转化为纹理时需要解压，但对于 CPU 影响很小。

19.2.4　纹理缓存异步加载

在启动游戏和进入场景时，由于需要加载的资源过多就会比较“卡”，用户体验不好。可以采用纹理缓存(TextureCache)异步加载纹理图片，TextureCache 类异步加载函数如下：

```
cc.Director:getInstance():getTextureCache():addImageAsync(filename, loadingCallBack)
```

其中第一个参数是文件路径，第二个参数是回调函数。下面通过一个实例介绍纹理缓存异步加载的使用。有 200 张小图片需要加载到纹理缓存，加载过程会有一个进度显示在界面上，如图 19-7 所示。

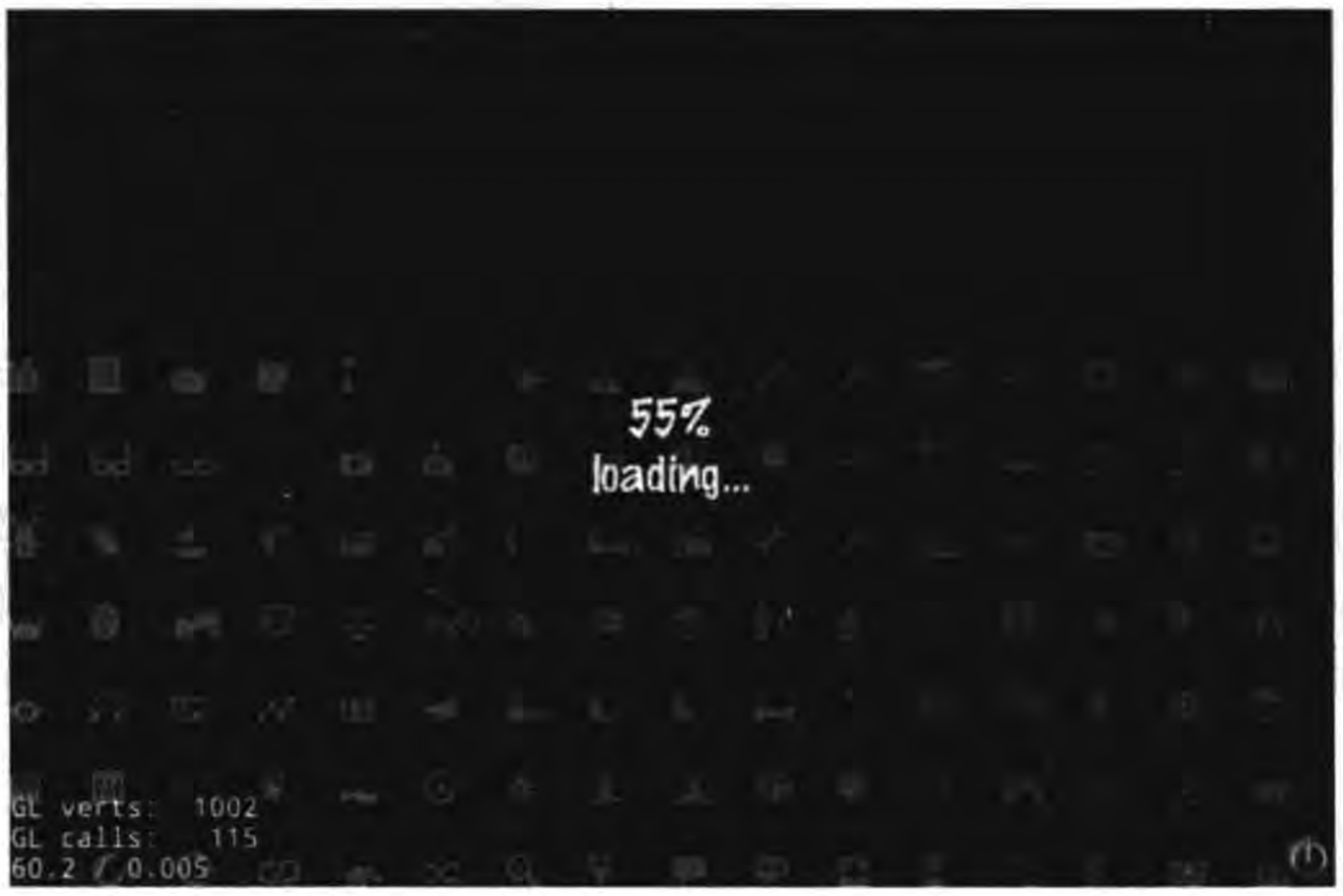


图 19-7　纹理缓存异步加载实例

GameScene.lua 中主要代码如下：

```
function GameScene:createLayer()

    local layer = cc.Layer:create()

    local function loadingCallBack(texture)
        <异步加载处理>
    end

    _labelLoading = cc.Label:createWithTTF("loading...", "fonts/Marker Felt.ttf", 35)
    _labelPercent = cc.Label:createWithTTF("0 % %", "fonts/Marker Felt.ttf", 35)

    _labelLoading:setPosition(cc.p(size.width / 2, size.height / 2 - 20))
    _labelPercent:setPosition(cc.p(size.width / 2, size.height / 2 + 20))

    layer:addChild(_labelLoading)
```

```
    layer:addChild(_labelPercent)

    _numberOfLoadedSprites = 0                                                    ①
    _imageOffset = 0                                                              ②

    local sharedFileUtils = cc.FileUtils:getInstance()                            ③
    local fullPathForFilename = sharedFileUtils:fullPathForFilename("ImageMetaData.plist")
                                                                                  ④

    local vec = sharedFileUtils:getValueVectorFromFile(fullPathForFilename)       ⑤
    _numberOfSprites = table.getn(vec)                                            ⑥
    cclog(_numberOfSprites)

    for i = 1, table.getn(vec) do                                                 ⑦
        local row = vec[i]                                                        ⑧
        local filename = "icons/" .. row["filename"]                              ⑨
        cc.Director:getInstance()
            :getTextureCache():addImageAsync(filename, loadingCallBack)           ⑩
    end

    return layer
end
```

上述第①行代码_numberOfLoadedSprites 变量是已经加载的图片数。第②行代码_numberOfSprites 变量是要加载的全部图片数。

第③行代码 cc. FileUtils:getInstance()是获得 FileUtils 实例。第④行代码是获得资源目录下 ImageMetaData. plist 文件全路径，ImageMetaData. plist 文件是用来描述要加载图标文件名，文件内容如下：

```
<?xml version = "1.0" encoding = "utf - 8"?>
<! DOCTYPE plist PUBLIC " - //Apple//DTD PLIST 1.0//EN"
        "http://www.apple.com/DTDs/PropertyList - 1.0.dtd">
<plist version = "1.0">
  <array>
    <dict>
      <key> filename </key>
      <string> 01 - refresh.png </string>
    </dict>
    <dict>
      <key> filename </key>
      <string> 02 - redo.png </string>
    </dict>
    <dict>
      <key> filename </key>
      <string> 03 - loopback.png </string>
    </dict>
    <dict>
      <key> filename </key>
      <string> 04 - squiggle.png </string>
    </dict>
  ...
  </array>
</plist>
```

ImageMetaData.plist 文件是属性列表文件，内部结构是数组类型，可以通过第⑤行代码 FileUtils 的 getValueVectorFromFile 函数读入到列表变量 vec 中。第⑥行代码 table.getn(vec)是获得列表的长度，然后赋值给成员变量_numberOfSprites，目的是能够计算加载进度。

第⑦行代码 for i=1，table.getn(vec) do 是遍历 vec 列表变量，其中 table.getn()可以获得 vec 列表长度，i 是循环变量。第⑧行代码 local row = vec[i]是获得 vec 列表变量中的一个元素。第⑨行代码中的 row["filename"]是一个元素中 filename 数据项。第⑩行代码是调用 TextureCache 的 addImageAsync 函数，实现异步加载图片缓存，loadingCallBack 是回调函数。

GameScene.lua 中的回调函数 loadingCallBack 的异步加载处理代码如下：

```
local function loadingCallBack(texture)
    _numberOfLoadedSprites   = _numberOfLoadedSprites + 1                                    ①
    local str = string.format("%d%%",(_numberOfLoadedSprites / _numberOfSprites) * 100)      ②
    _labelPercent:setString(str)                                                             ③
    _imageOffset = _imageOffset +1                                                           ④
    local i = _imageOffset * 60
    cclog("i,_imageOffset =  %d,   %d",i,_imageOffset)

    local sprite = cc.Sprite:createWithTexture(texture)                                      ⑤
    sprite:setAnchorPoint(cc.p(0,0))
    layer:addChild(sprite, -1)
    --math.floor 取小于该数的最大整数
    local x =   math.floor(i % size.width)                                                   ⑥
    local y =   math.floor(i / size.width) * 60                                              ⑦

    cclog("x,y =  %d,   %d",x,y)

    sprite:setPosition(cc.p( x, y))

    if _numberOfLoadedSprites == _numberOfSprites then                                       ⑧
        _numberOfLoadedSprites = 0                                                           ⑨
    end
end
```

上述第①行代码是累加_numberOfLoadedSprites 变量。第②行代码是计算出加载进度并转换为字符串，"%d%%"可以显示百分号，其中的%d 是格式化输出数字。%%是输出%，前面%起到转义作用。第③行代码是设置进度标签_labelPercent 的内容。第④行代码是累加_imageOffset 变量。

第⑤行代码 local sprite = cc.Sprite:createWithTexture(texture)是通过纹理对象 texture 创建精灵对象。第⑥行代码是精灵图片的 x 轴坐标，i % size.width 表达式是 i 与屏幕宽度取余数，这样 x 坐标不会超出屏幕。但是需要截取小数部分，math.floor 函数是取小于该数的最大整数，能够将正数的小数截取掉。第⑦行代码是精灵图片的 y 轴坐标，math.floor(i / size.width) * 60 表达式是先通过 i / size.width 计算除数，再乘以 60，60 是每一行高度。

第⑧行代码是判断是否完成任务。第⑨行代码是初始化_numberOfLoadedSprites变量。

19.2.5 背景图片优化

由于背景图片长时间在场景中保存,而且图片很多,可以对其进行一些优化。通过如下几个方面考虑优化。

1. 不要 Alpha 通道

背景图片的特点是不需要透明,所以纹理格式可以采用不带有 Alpha 通道的格式,所以 RBG565 格式比较适合背景图片。

2. 拼图

背景图片与其他的图片纹理格式不同,在创建纹理图集的时候,没有办法将 RBG565 格式的背景图片与其他的纹理图片(如 RGBA4444)做在一个纹理图集,所以基于格式的考虑,可以将多个背景放置在一个纹理图集中,但是要注意这个纹理图集拼接成的大图文件不能太大,一些老设备对于单个文件大小是有限制的,如 iPod touch 4 是单个文件,不能超过 2048×2048 像素大小。

3. 加载到纹理缓存的时机

什么时候加载背景图片到纹理缓存呢? 这个问题主要看背景图片的场景使用频率,如果频率高就要在游戏初始化时加载。对于频率比较低的场景背景图片,可以考虑进入场景时加载。在图片进行加载的时候,由于背景图片比较大,加载时间比较长,可以考虑异步加载。

4. 小纹理图片,重复贴图

如果场景的背景采用单色或有规律的图形,可以采用小纹理图片,重复贴图实现。在第 7 章的案例中采用了一个 128×128 的纹理图片(BackgroundTile.png)反复贴图,这样可以减少内存消耗。核心代码如下:

```
--贴图的纹理图片宽高必须是 2 的 n 次幂,128 × 128
localbg = cc.Sprite:create("BackgroundTile.png", cc.rect(0, 0, size.width, size.height))
--贴图的纹理参数,水平重复平铺,垂直重复平铺
bg:getTexture():setTexParameters(gl.LINEAR, gl.LINEAR, gl.REPEAT, gl.REPEAT)
bg:setPosition(cc.p(size.width/2, size.height/2))
layer:addChild(bg, 0)
```

5. 考虑使用瓦片地图

背景可以考虑采用瓦片地图实现。由于瓦片地图只需要几个小图片就可以构建一个很大的游戏背景,它的性能自不用多说,但是它的缺点也是由于采用几个瓦片拼接而成,背景上有很多重复的区域,如果用户不在乎这些,当然选择瓦片地图构建背景是首选方式。另外,在设计瓦片地图时,地图中的层不要超过 4 层。

6. 背景 z 深度的优化

有时为了达到动态视差效果,背景被分成了几个图片,如图 19-8 所示,通过设置 z 轴顺

序(z-order)可以把云、树木、草地和山分别放置在不同背景图片中。

图 19-8　背景图片实例

19.3　声音优化

在游戏中用的最多的除了图片资源就是声音资源了。这一节来介绍声音文件的优化。

19.3.1　声音格式优化

有关声音的格式在前面章节介绍过了。不同平台对于声音文件的支持是不同的,优化的方式也是不同的。

在前面章节介绍了苹果的声音格式有 CAFF(Core Audio File Format)和 AIFF(Audio Interchange File Format),其中 CAFF 是无压缩音频格式,AIFF 的压缩格式是 AIFF-C(或 AIFC),应用于 Mac OS X 和 iOS 系统。在 Android 平台一般采用 MP3。

我们使用的声音主要用于背景音乐和音效,下面从这两个方面介绍相关的优化技术。

1. 背景音乐优化

背景音乐会在应用中反复播放,它会一直驻留在内存中并耗费 CPU,所以更合适比较小的文件,而压缩文件是不错的选择。压缩文件主要有 AIFC 和 MP3 两种格式,在 iOS 平台首选 AIFC,因为这是苹果推荐的格式。但是获得的原始文件格式不一定是 AIFC,这种情况下需要使用苹果 Mac OS X 中提供的音频转换命令行工具——afconvert,afconvert 工具位于/usr/bin 目录下。要将其转换为 AIFC 格式需要在终端中执行如下命令:

```
$ afconvert -f AIFC -d ima4 Fx08822_cast.wav
```

其中-f AIFC 参数用于转换为 AIFC 格式,-d ima4 参数指定解码方式,Fx08822_cast.wav 是要转换的源文件。转换成功后,会在相同目录下生成 Fx08822_cast.aifc 文件。本例中的源文件 Fx08822_cast.wav 的大小是 295KB,转换之后的 Fx08822_cast.aifc 文件的大小是 82KB。当然,afconvert 工具也可以转换 MP3 等其他压缩格式文件。如果同时有 WAV 文件,就应优先采用 WAV 文件。MP3 本身是有损压缩,如果再经过 afconvert 转换,音频的质量会受到影响。

另外,在制作背景声音时,要求文件不要太大,单个文件播放时间也不要太长,如果原始文件播放时间比较长,可以使用一些工具截取一小段,然后反复播放就可以了。

2. 音乐特效优化

音乐特效用于很多游戏中,如发射子弹、敌人被打死或单击按钮等发出的声音,这些声音都是比较短的。如果追求震撼的3D效果,在iOS平台可以采用苹果专用的无压缩CAFF格式文件,其他格式的文件尽量不要考虑。一般不要使用压缩音频文件,这主要是因为音乐特效通常采用OpenAL技术,它只接受无压缩的音频文件。另外,压缩音频文件都会造成音质的丢失。如果没有CAFF格式的文件,也可以使用afconvert工具将其转换为CAFF格式。在终端中执行如下命令:

```
$ afconvert -f caff -d LEI16 Fx08822_cast.wav
```

其中-f caff参数用于转换为CAFF格式,-d LEI16参数指定解码方式,Fx08822_cast.wav是要转换的源文件。默认音频的采样频率为22050Hz,如果想提高音频采样频率,可以通过如下命令:

```
$ afconvert -f caff -d LEI16@44100  Fx08822_cast.wav
```

其中-d LEI16@44100参数中的44100表示音频采用频率为44100Hz。

综上所述,在iOS平台背景音乐采用AIFC格式是首选,音效采用CAFF格式是首选。Android平台背景音乐采用MP3格式是首选,音效采用OGG或WAV格式是首选。

19.3.2 声音预处理与清除

声音与图片是比较类似的,为了追求好的效果,文件可能就会比较大。为此文件都会压缩,使用时就需要解压处理,因此在使用声音之前有必要进行预处理。在Cocos2d-x Lua API中是使用CocosDenshion声音引擎管理声音的,针对背景音乐和音效,声音预处理示例代码如下:

```
--初始化  音乐
AudioEngine.preloadMusic( "sound/Synth.mp3" )
--初始化  音效
AudioEngine.preloadEffect( "sound/Blip.wav" )
```

预处理的过程就是把声音文件解压、解码等处理后加载到声音缓存中。当再次使用时就不需要重新加载,这样可以提高执行效率。声音缓存的清除,只有音效提供了清除缓存的函数unloadEffect,背景音乐没有清除缓存的函数,音效的清除缓存示例代码如下:

```
//卸载音效
AudioEngine.unloadEffect("sound/Blip.wav");
```

由于没有清除背景音乐函数,这就意味着一旦将背景音乐加载到缓存中,就无法释放了,所以一定要谨慎。而音效的加载和清除的时机与纹理缓存和精灵帧缓存是类似的,要在合适的时机来加载和清除,基本使用的原则也是类似的。

如果它们的声明周期很长,可以在main.lua的main()函数中进行预处理。如果是具

体场景中使用的声音可以在进入到这个场景之前的 ctor()构造函数中进行预处理，一般情况下不需要考虑清除。如果经常测试，可以在场景的 enter 事件处理函数中加载缓存，在该场景的 exit 事件处理函数中清除。

原则上尽可能将更多的资源加载到缓存中，这会使得程序运行起来更流畅。尽可能少清除缓存，频繁的缓存加载和清除给系统带来的开销可能会更大。

本章小结

通过对本章的学习，使读者了解性能优化的方法，其中包括使用缓存、图片和纹理优化和声音优化等。这些内容都是非常重要的，希望读者认真掌握。

第五篇　多平台移植篇

本篇共 2 章，介绍 Cocos2d-x 游戏如何从 Win32 平台移植到 Android 平台和从 Win32 平台到 iOS 平台。

本篇各章如下：

第 20 章　移植到 Android 平台

第 21 章　移植到 iOS 平台

第 20 章

移植到 Android 平台

本章介绍如何在 Win32 下进行游戏工程的开发并移植到本地 Android 平台。

20.1 搭建交叉编译和打包环境

在 Windows 下使用的桌面计算机是基于 x86 架构[①] CPU 的，而手机等移动设备是基于 ARM 架构[②] CPU 的。由于两个 CPU 的指令集不同、架构不同，在 Windows 下编写的程序代码，如果要在手机上运行，就需要交叉编译，即把程序代码在 x86 CPU 下编译成为 ARM CPU 下能够运行的二进制文件。

这个过程事实上跨越很大。不过不用担心，Android 提供了 NDK 开发工具包，可以帮助我们实现交叉编译过程。Cocos2d-x 3.x 也提供了一些工具使得编译和打包更加轻松。

这个编译过程可以在 Windows、Linux 和 Mac OS X 等操作系统上执行，配置和执行过程基本类似。在 Windows 系统中能够满足需要的版本有：Windows XP (32 位)、Vista (32 或 64 位)、Windows 7 (32 或 64 位)和 Windows 8(32 或 64 位)。如果在 Mac 平台，要求 OS X 10.5.8 及以上。如果在 Linux 平台，可以是 Ubuntu 10.04 LTS 及以上。

这几个平台需要的软件都是类似的，这里以介绍 Windows 操作系统为主，其他的操作系统为辅。

编译需要准备的软件有 Cocos2d-x、JDK、Apache Ant、Python、Android SDK、Android NDK。

1. JDK

由于 Ant 等软件需要 JDK，JDK 是必须的，并且要求 JDK 6 以上版本。图 20-1 所示是 JDK 7 下载界面，它的下载地址是 http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html。其中有很多版本，注意选择对应的操作系统，

---

① x86 指令集是美国 Intel 公司为其第一块 16 位 CPU(i8086)专门开发的，美国 IBM 公司 1981 年推出的世界第一台 PC 中的 CPU— i8088(i8086 简化版)使用的也是 x86 指令。

② ARM 架构是一个 32 位精简指令集处理器架构，支持 Thumb(16 位)/ARM(32 位)双指令集，能很好地兼容 8 位/16 位器件。它的优点是体积小、低功耗、低成本、高性能，能广泛地使用在许多嵌入式系统设计。大部分移动设备，如 iPhone、iPad 等都是使用 ARM 架构 CPU。

以及 32 位还是 64 位安装的文件。

Java SE Development Kit 7u51

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux ARM v6/v7 Hard Float ABI	67.7 MB	jdk-7u51-linux-arm-vfp-hflt.tar.gz
Linux ARM v6/v7 Soft Float ABI	67.68 MB	jdk-7u51-linux-arm-vfp-sflt.tar.gz
Linux x86	115.65 MB	jdk-7u51-linux-i586.rpm
Linux x86	132.98 MB	jdk-7u51-linux-i586.tar.gz
Linux x64	116.96 MB	jdk-7u51-linux-x64.rpm
Linux x64	131.8 MB	jdk-7u51-linux-x64.tar.gz
Mac OS X x64	179.49 MB	jdk-7u51-macosx-x64.dmg
Solaris x86 (SVR4 package)	140.04 MB	jdk-7u51-solaris-i586.tar.Z
Solaris x86	95.13 MB	jdk-7u51-solaris-i586.tar.gz
Solaris x64 (SVR4 package)	24.53 MB	jdk-7u51-solaris-x64.tar.Z
Solaris x64	16.28 MB	jdk-7u51-solaris-x64.tar.gz
Solaris SPARC (SVR4 package)	139.38 MB	jdk-7u51-solaris-sparc.tar.Z
Solaris SPARC	98.19 MB	jdk-7u51-solaris-sparc.tar.gz
Solaris SPARC 64-bit (SVR4 package)	23.92 MB	jdk-7u51-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	18.33 MB	jdk-7u51-solaris-sparcv9.tar.gz
Windows x86	123.64 MB	jdk-7u51-windows-i586.exe
Windows x64	125.46 MB	jdk-7u51-windows-x64.exe

图 20-1　下载 JDK

下载并安装完成之后，需要设置系统环境变量，主要是设置 JAVA _HOME 环境变量。打开环境变量设置对话框，如图 20-2 所示，可以在用户变量(上半部分，只影响当前用户)或系统变量(下半部分，影响所有用户)添加环境变量。一般情况下，在用户变量中设置环境变量。

图 20-2　环境变量设置对话框

在用户变量部分单击“新建”按钮，系统弹出对话框，如图 20-3 所示。设置“变量名”为 JAVA_HOME，“变量值”为 C:\Program Files\Java\jdk1.7.0_21。注意变量值的路径。

图 20-3　设置 JAVA_HOME

为了防止安装多个 JDK 版本对于环境的影响，还可以在环境变量 PATH 中追加 C:\Program Files\Java\jdk1.7.0_21\bin 路径，如图 20-4 所示，在用户变量中找到 PATH。双击打开 PATH 修改对话框，如图 20-5 所示，追加 C:\Program Files\Java\jdk1.7.0_21\bin。注意 PATH 之间用分号分隔。

图 20-4　环境变量 PATH 设置对话框

2. Python

关于 Python 的安装在第 3 章介绍过了，这里就不再赘述。

Android SDK 和 Android NDK 安装与配置的过程有一点复杂，这里重点介绍一下。

图 20-5　PATH 修改对话框

20.1.1　Android SDK 安装

Android SDK 下载地址为 https://developer.android.com/studio/index.html，在浏览器中打开该网址。如图 20-6 所示，可以有两个选择：installer_r24.4.1-windows.exe 和 android-studio-bundle-143.2915827-windows.exe。

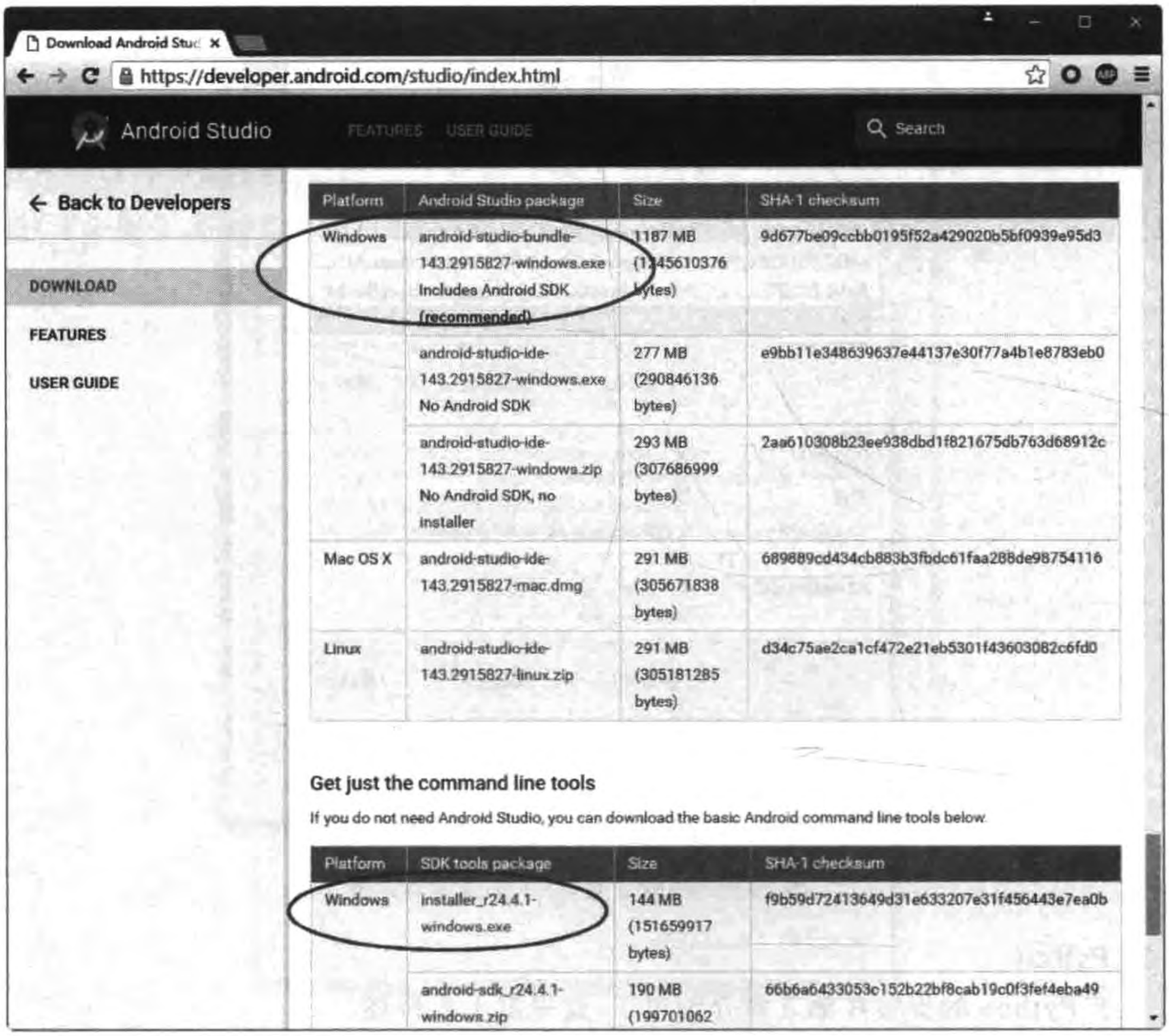


图 20-6　下载 Android SDK

installer_r24.4.1-windows.exe 提供 Android SDK 安装文件，下载完成后双击即可安装。

android-studio-bundle-143.2915827-windows.exe 提供一个工具包，它包括 Android Studio[①] 工具和 Android SDK。在交叉编译过程中，其实并不需要 Android Studio，Android Studio 能够方便管理 Android 应用的发布、运行和调试，方便管理 Android 模拟器等。

提示　开发 Android 应用除了 Android Studio，还有 Eclipse＋ADT 插件。

20.1.2　管理 Android SDK

在 Android SDK 的安装目录下有一个 SDK Manager.exe 文件，双击它则启动 Android SDK 安装管理对话框，如图 20-7 所示，可以下载和管理 Android SDK。

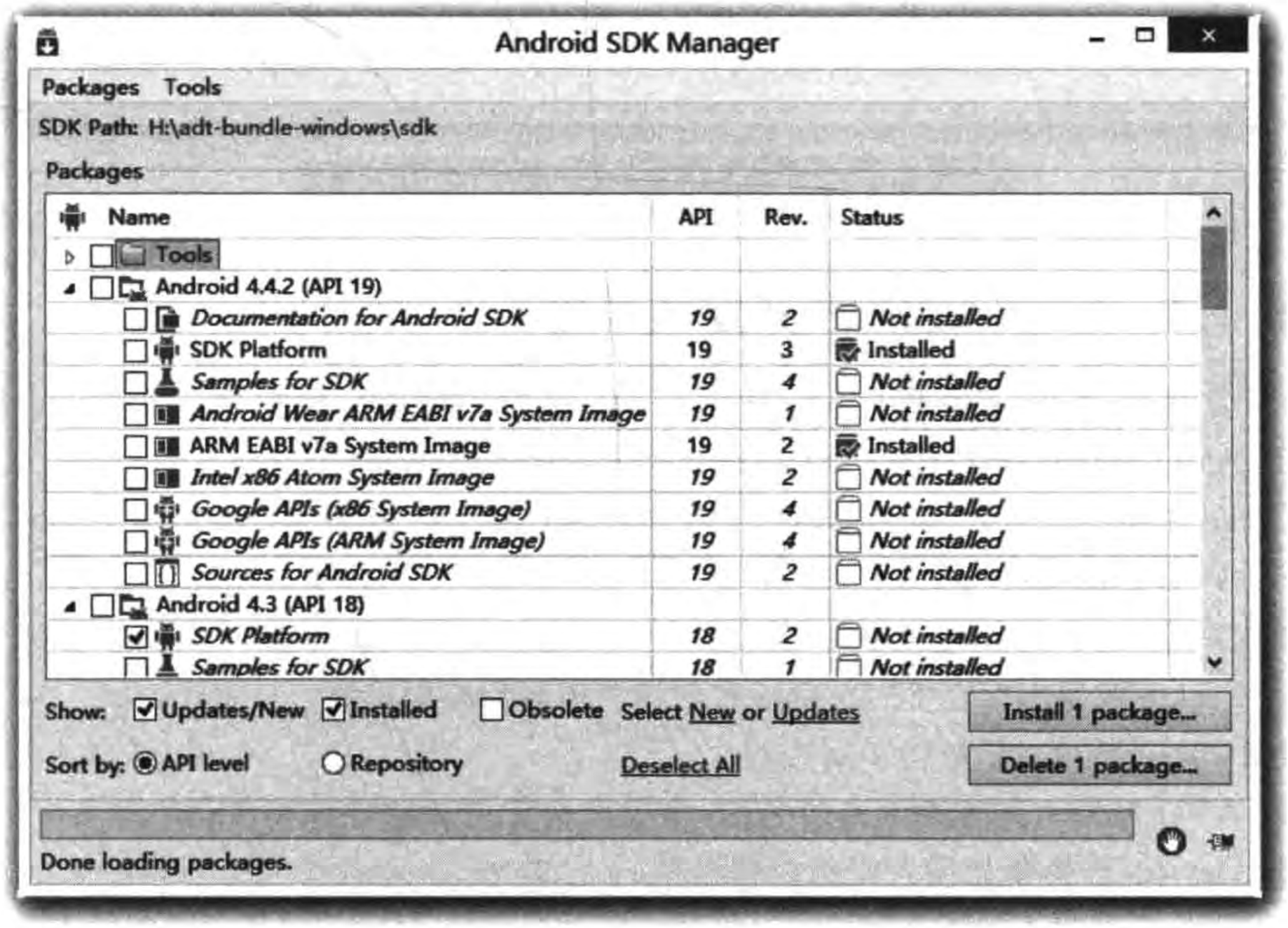


图 20-7　下载和管理 Android SDK

选择需要的 SDK 等内容，单击 Install 1 package 按钮弹出安装许可对话框，如图 20-8 所示。选择 Accept 单选按钮，然后再单击 Install 按钮就开始下载安装了。

① Android Studio 是一个为 Android 平台开发程序的集成开发环境。2013 年 5 月 16 日在 Google I/O 上发布，可供开发者免费使用。——引自维基百科 https://zh.wikipedia.org/wiki/Android_Studio

图 20-8 Android SDK 安装

20.1.3 管理 Android 开发模拟器

在开发这些手机应用程序时,开发环境一般都提供了模拟器。Android SDK 中 Android AVD Manager 工具可以创建和管理 Android 模拟器。

可以在 Android SDK 中直接运行 AVD Manager. exe 或者在 Android Studio 中单击 Android AVD Manager 按钮。

如果采用 AVD Manager. exe 工具打开 Android AVD Manager 对话框,如图 20-9 所示,单击 Create 按钮出现图 20-10 所示的创建模拟器对话框。

在图 20-10 所示对话框中填写内容:

(1) AVD Name,是虚拟设备的名称,由用户自定义。这里输入 AVD1。

(2) Device,选择预先设置好的设备模板。

(3) Target,选择不同的 SDK 版本(依赖当前 SDK 的 platform 中包含哪些版本的 SDK)。

(4) CPU/ABI,选择 CPU 类型,其中 ABI 是 ARM 体系结构。

(5) Keyboard,选中后面的选项,可以在模拟器中出现键盘。

(6) Skin,是皮肤,它的含义其实是仿真器运行尺寸的大小,默认的尺寸有 HVGA-P(320×480)、HVGA-L(480×320)等。也可以通过直接指定尺寸的方式指定屏幕的大小。

(7) Front/Back Camera,选中前、后摄像头开启。这需要计算机中硬件支持。

(8) Memory Options,内存选项,设置模拟器的内存。其中 RAM 是内存,VM Heap 是堆内存大小。

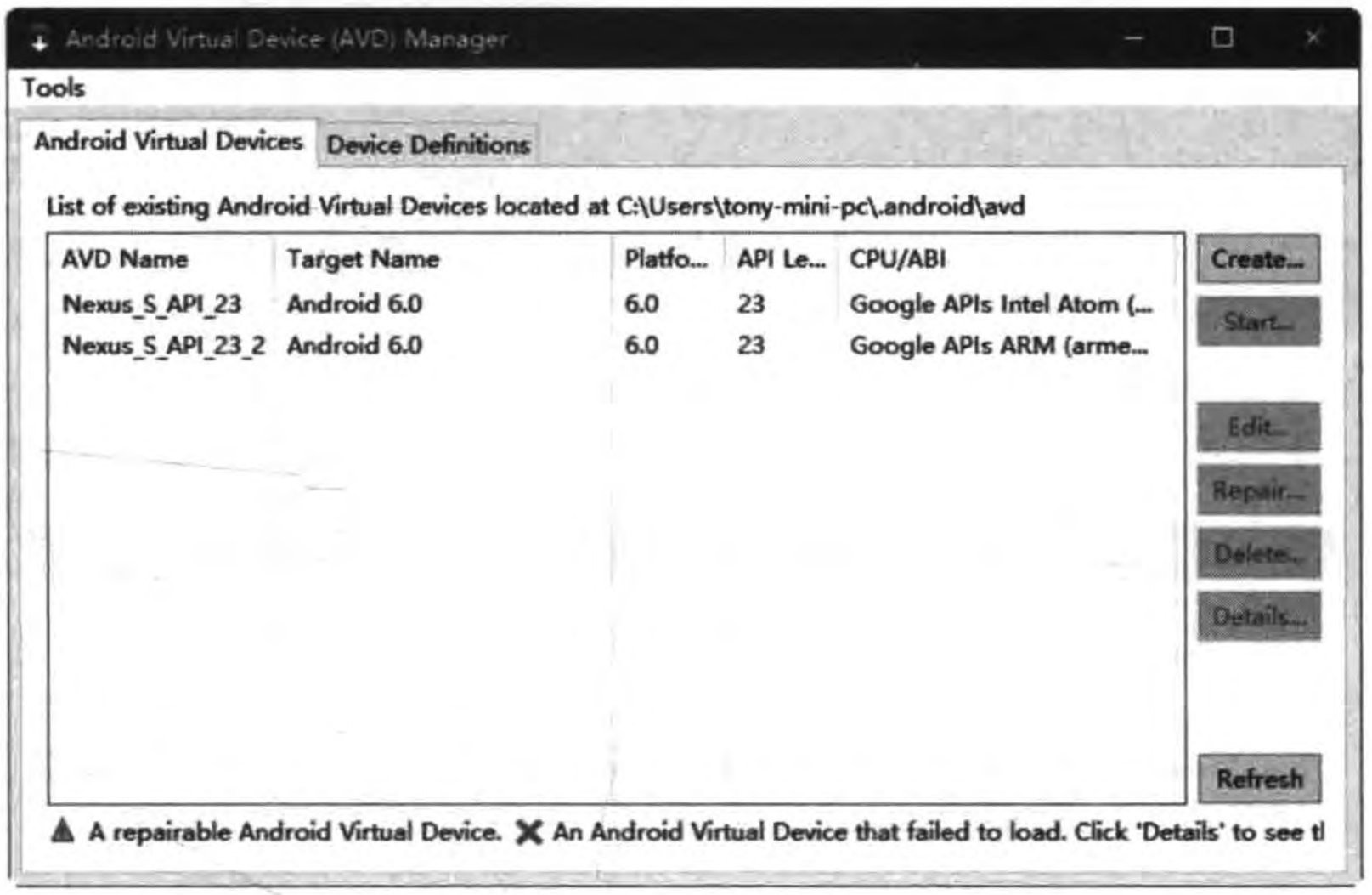


图 20-9　Android AVD Manager 对话框

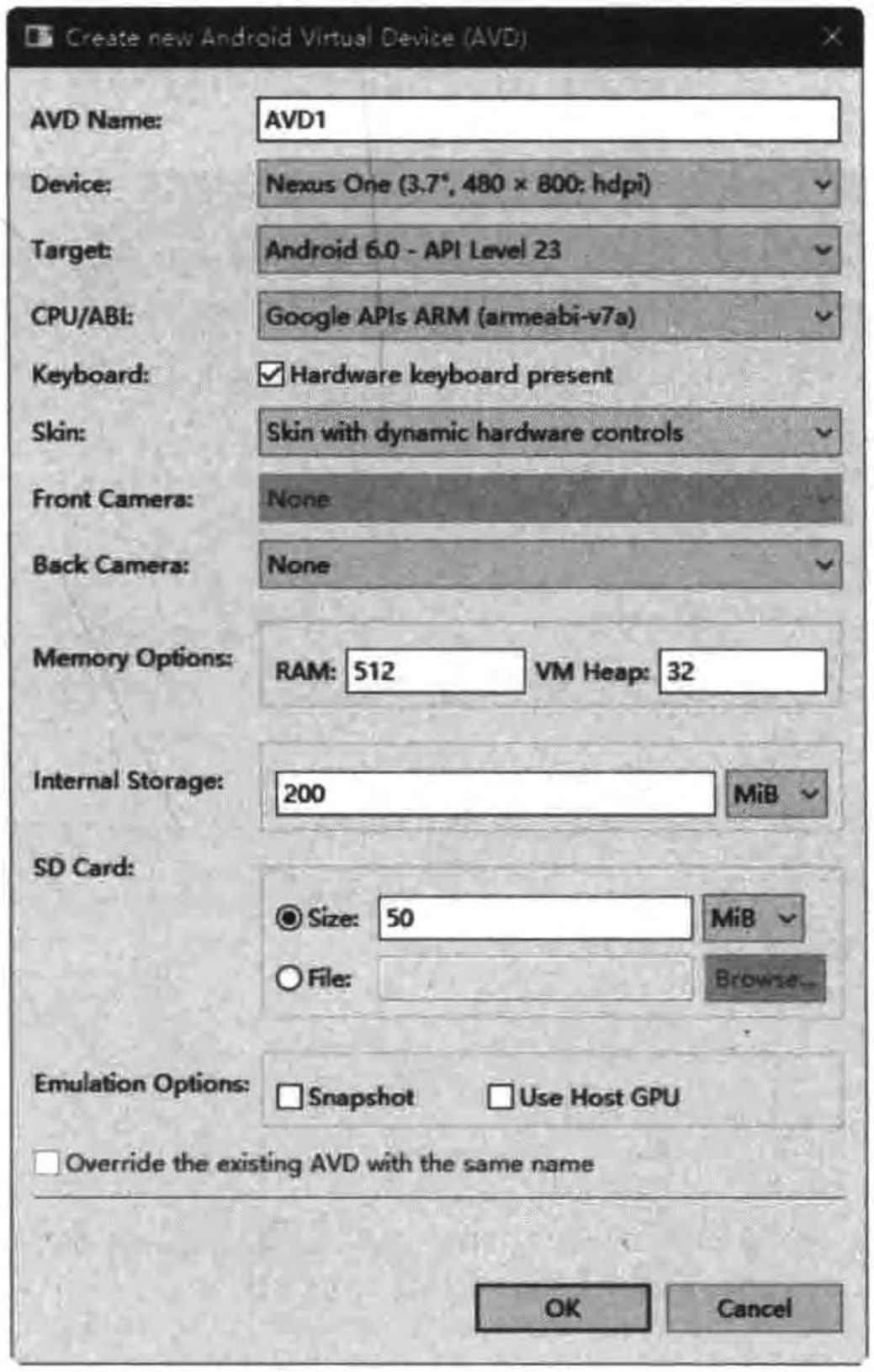


图 20-10　创建模拟器

(9) Internal Storage,内部存储,注意不要与 RAM 内存混淆,它还属于外设,访问它需要 I/O 操作。

(10) SD Card,模拟 SD 卡,它是与内部存储(internal storage)对应的外部存储。可以选择大小或者一个 SD 卡映像文件,SD 卡映像文件是使用 mksdcard 工具建立的。不需要 SD 卡时可以不定义,这里就不定义 SD 卡。

单击 OK 按钮,在 AVD Manager 对话框里就出现了刚刚创建的模拟器。可以在 AVD Manager 窗口中选择一个设备,单击右侧的 Start 按钮,将启动虚拟设备,如图 20-11 所示。

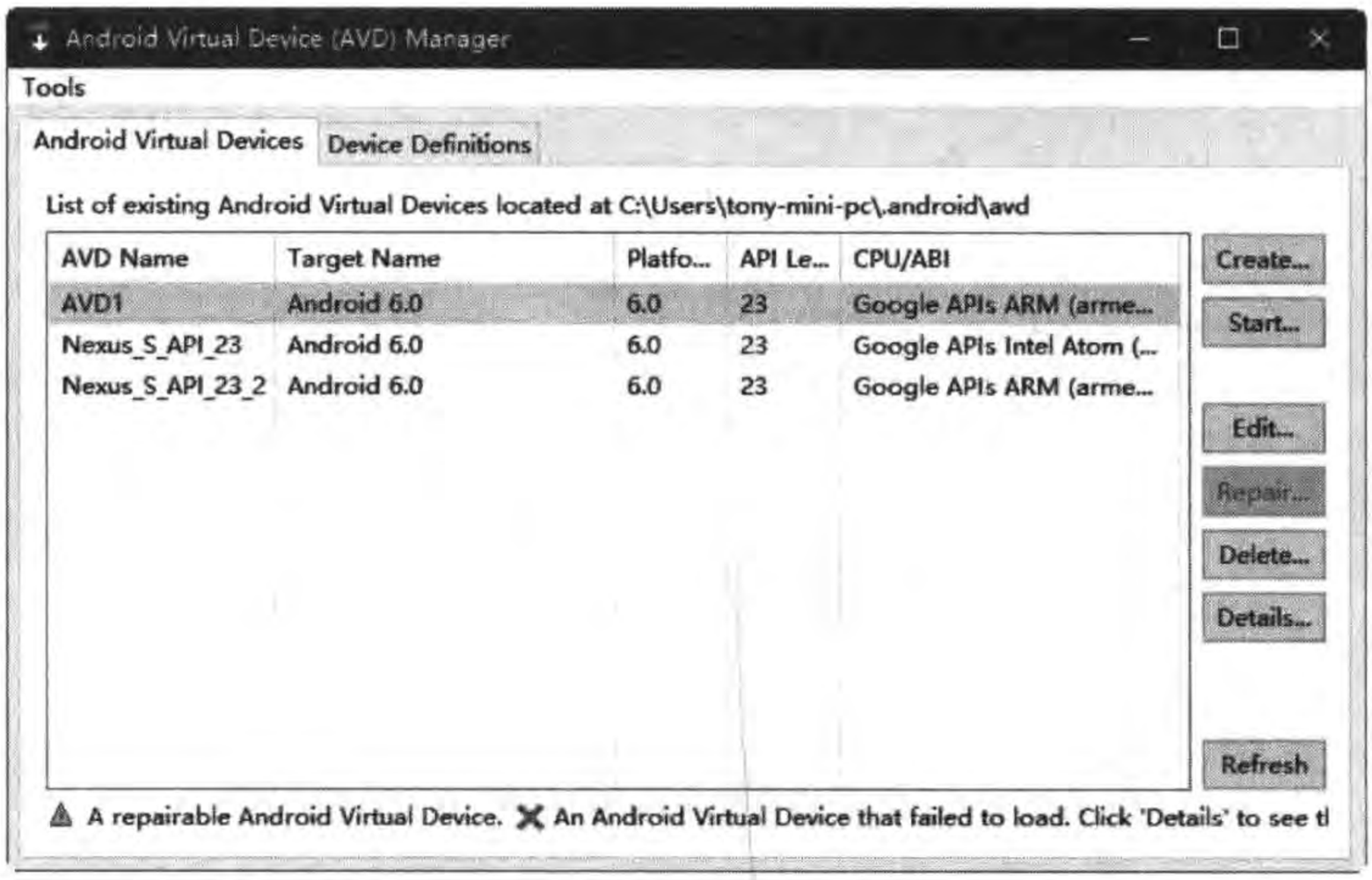


图 20-11 启动虚拟设备

20.1.4 Android NDK 安装

Android NDK(native development kit)是一个工具包,它提供了构建 C(或 C++)的动态库,并能够交叉编译为 so(动态链接库),也可以编译 Java 代码,然后一起打包成 apk(Android 安装包)。

Android NDK 下载地址为 http://developer.android.com/tools/sdk/ndk/index.html。打开下载页面,如图 20-12 所示,其中有很多版本,注意选择对应的操作系统,以及 32 位还是 64 位安装的文件。

文件下载完后需要解压出来,保存好它的位置。

20.1.5 设置环境变量

在第 3 章虽然已经介绍过如何通过 setup.py 工具设置 Cocos2d-x 环境变量,但是没有设置 NDK 等环境变量。

进入到 Cocos2d-x 根目录所在 DOS 终端,运行 setup.py,如图 20-13 所示。如果这个过程中发现哪些变量没有设置,窗口的光标就会停止在哪儿,等待用户输入正确的路径,输

入完成后按 Enter 键即可继续。设置成功之后的输出结果如图 20-14 所示。

图 20-12　下载 Android NDK

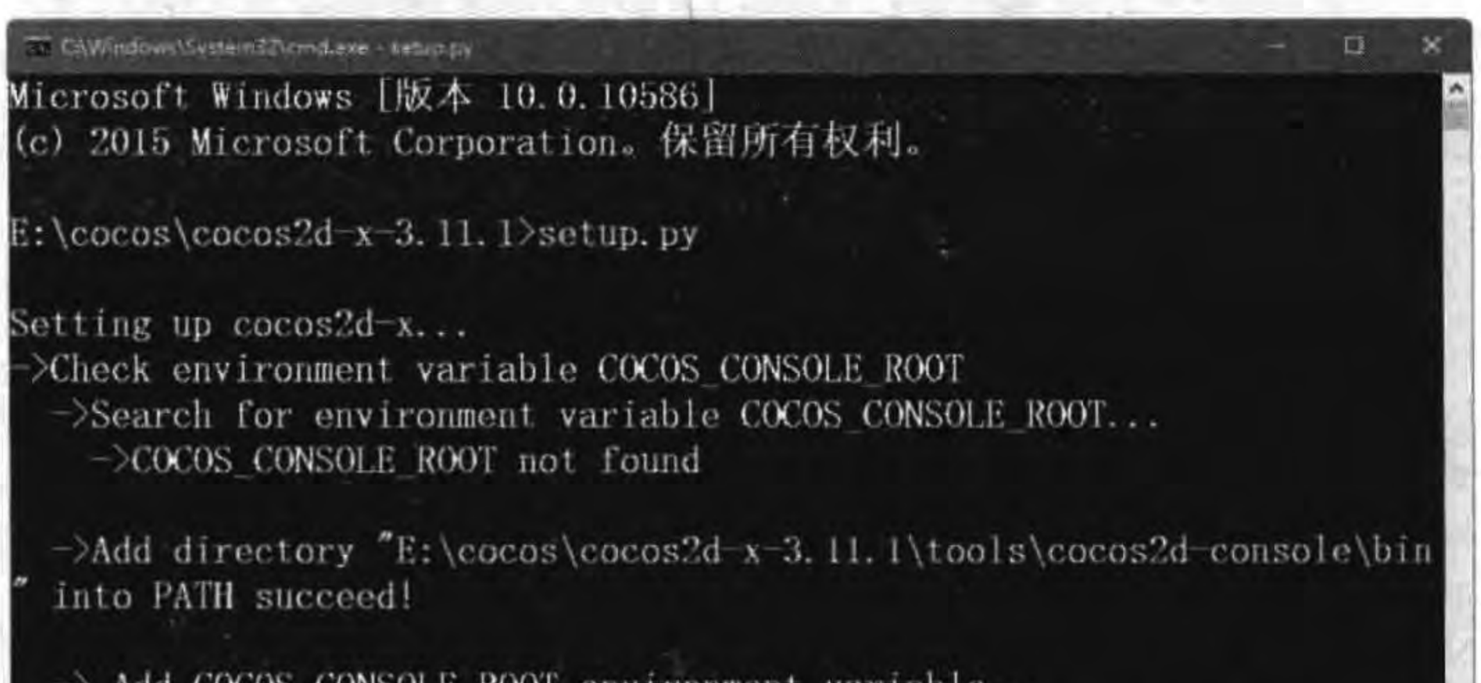


图 20-13　设置环境变量

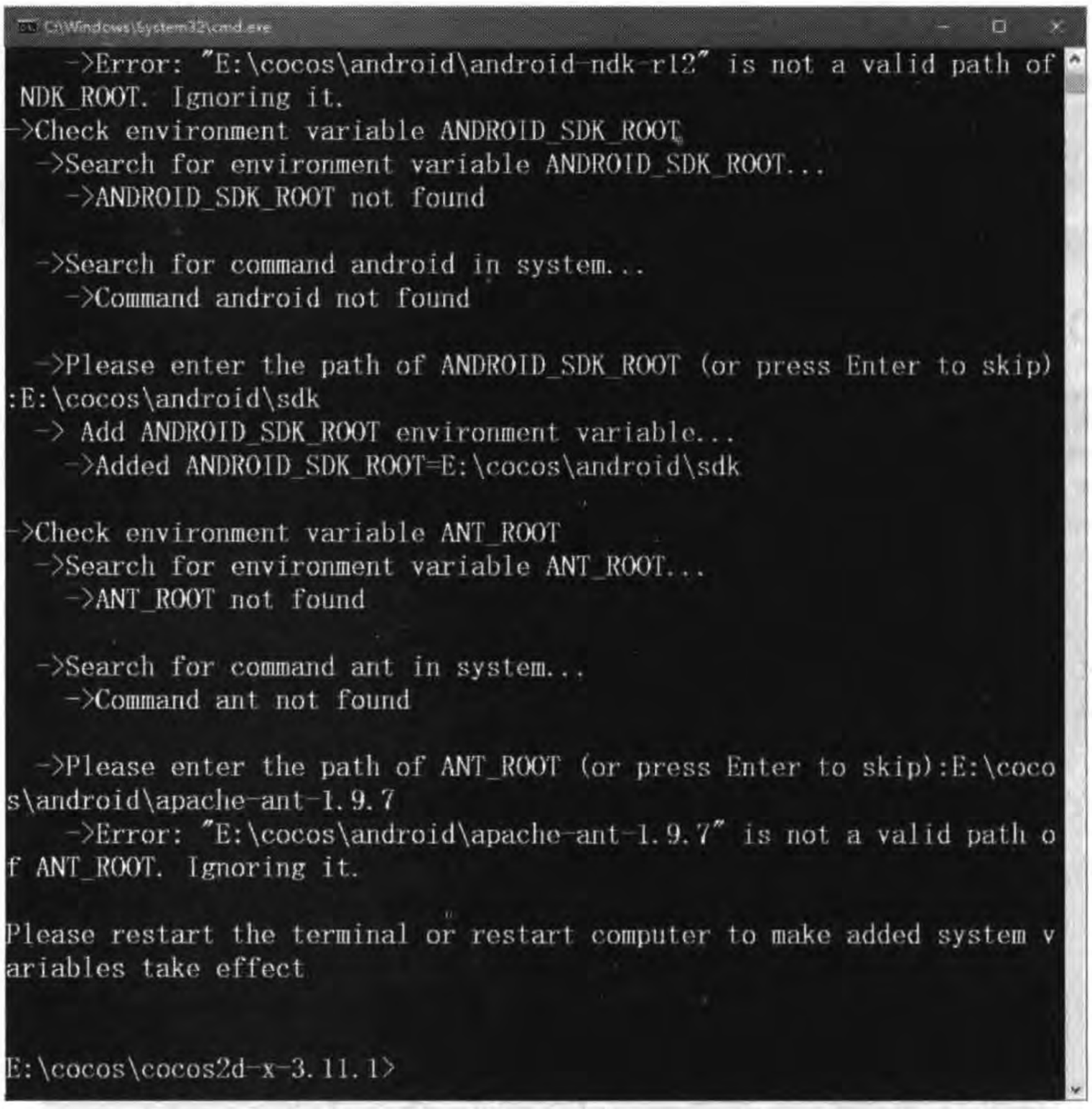


图 20-14　设置环境变量成功

20.2　交叉编译、打包和运行

环境配置成功之后，就可以进行交叉编译了。能够进行交叉编译的方法有很多，但最为方便的是 cocos 工具，在前面已经使用过 cocos new 等指令，现在使用它在 Android 平台进行交叉编译、打包、安装和运行。

20.2.1　使用 cocos 命令行工具

首先通过 DOS 进入到要编译的工程的根目录下，如图 20-15 所示，输入 cocos run -p android 命令。其中，MyGame 是工程根目录，然后按 Enter 键开始执行，如果一切都顺利，则会出现图 20-16 所示的结果。

20.2.2　Android.mk 编译文件

上一节使用 cocos 工具对 C 和 C++源代码进行编译。实际上，cocos 工具是读取<游戏工程目录>\proj.android\jni\目录中的 Android.mk 文件进行交叉编译和打包。

图 20-15　命令行交叉编译

图 20-16　命令行交叉编译成功

Android.mk 是一个编译文件，它是 GNU Makefile 的一小部分，用来向 Android NDK 描述 C 和 C++源代码文件，描述如何进行编译以及打包等操作。默认的 Android.mk 文件内容如下：

```
LOCAL_PATH := $(call my-dir)                                    ①

include $(CLEAR_VARS)                                           ②
```

```
$(call import-add-path, $(LOCAL_PATH)/../../cocos2d)                          ③
$(call import-add-path, $(LOCAL_PATH)/../../cocos2d/external)
$(call import-add-path, $(LOCAL_PATH)/../../cocos2d/cocos)
$(call import-add-path, $(LOCAL_PATH)/../../cocos2d/cocos/audio/include)

LOCAL_MODULE := MyGame_shared                                                 ④

LOCAL_MODULE_FILENAME := libMyGame                                            ⑤

LOCAL_SRC_FILES := hellocpp/main.cpp \
                   ../../Classes/AppDelegate.cpp \
                   ../../Classes/HelloWorldScene.cpp                          ⑥

LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../Classes                               ⑦

...


LOCAL_STATIC_LIBRARIES := cocos2dx_static                                     ⑧

...

include $(BUILD_SHARED_LIBRARY)                                               ⑨

$(call import-module,2d)                                                      ⑩

...
```

下面解释这些项目的含义。第①行代码中 LOCAL_PATH 是定义当前目录变量,该变量必须定义,其中 my-dir 是宏,call my-dir 是返回当前目录。第②行代码 include $(CLEAR_VARS)是清除 LOCAL 开通的变量,但不包含 LOCAL_PATH,这是因为所有的变量都是全局的。第③行代码是添加 LOCAL_PATH 目录。

第④行代码是定义 LOCAL_MODULE 变量,该变量是必须定义的,用来识别 Android.mk 文件中描述的每个模块。名称必须是唯一的,而且不包含任何空格。第⑤行代码定义 LOCAL_MODULE_FILENAME 变量,它是可选的,通过该变量可以重新定义生成文件的名字。本例中生成文件名为 libMyGame.so 的文件。

第⑥行代码是定义 LOCAL_SRC_FILES 变量,它描述了模块中将要编译的源文件列表。第⑦行代码是定义 LOCAL_C_INCLUDES 变量,指定头文件搜索路径逻辑列表。

第⑧行代码是定义 LOCAL_STATIC_LIBRARIES 变量,用来指定要连接的库模块。

第⑨行代码 include $(BUILD_SHARED_LIBRARY)表示编译生成动态链接库(或共享库),文件命名为 lib<库模块名>.so。另外,可以使用 BUILD_STATIC_LIBRARY 告知编译系统生成静态链接库,文件命令为 lib<库模块名>.a。

第⑩行代码 $(call import-module,2d)通过目录名包含另一个模块的 Android.mk,其

中 2d 是模块目录名。

提示　库是一些没有 main 函数的程序代码的集合。库分为静态链接库和动态链接库。它们的区别是：静态链接库可以编译到执行代码中，应用程序可以在没有静态链接库的环境下运行；动态链接库不能编译到执行代码中，应用程序必须在有链接库文件的环境下运行。在微软的 Windows 和 Windows Phone 平台，动态链接库文件是.dll 文件，静态链接库文件是.lib 文件；在 Linux 和 Android 平台，动态链接库文件是.so 文件，静态链接库文件是.a 文件；在 Mac OS X 和 iOS 等平台，动态链接库文件是.dylib 文件，静态链接库文件是.a 文件。

20.3　移植问题汇总

Android 平台版本和设备碎片化很严重，因此当把游戏移植到 Android 平台会遇到很多问题，下面是我们归纳了移植到 Android 平台遇到的一些问题。

20.3.1　Lua 文件编译问题

Lua 编写的游戏在 Android 和 iOS 等平台移植后，很容易被别人获得安装包文件。这些文件可以被解压，Lua 源代码文件的内容很容易被别人看到。我们需要保护 Lua 源代码文件，cocos luacompile 工具能够编译 Lua 源代码文件为 luac 文件，这样源代码内容就不会被他人看见了。lua 编译并非编译成为 C++的二进制机器码，而是字节码(bytecode)。

我们并不需要对 Cocos2d-x Lua API 工程中的所有 Lua 文件编译，只需要对我们自己编写的<游戏工程目录>\src 目录内容编译。

具体步骤是通过 DOS 进入到要编译的工程的根目录下，执行如下命令：

```
cocos luacompile -s <游戏工程全路径>\src\ -d <游戏工程全路径>\src\
```

然后通过 DOS 命令或在资源管理器中删除编译之后 src 目录中的.lua 文件。上面指令同样适用于在 Mac OS X 的终端中执行。

20.3.2　横屏与竖屏设置问题

Cocos2d-x Lua API 工程模板默认情况下是横屏显示的，而有的游戏是竖屏显示的，我们需要在 Android 工程中进行一些设置。打开<游戏工程目录>\frameworks\runtime-src\proj.android\AndroidManifest.xml 文件，AndroidManifest.xml 文件是 Android 工程的配置文件，内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.work6"
      android:versionCode="1"
```

```
    android:versionName = "1.0">

<uses - sdkandroid:minSdkVersion = "9"/>
<uses - feature android:glEsVersion = "0x00020000" />

<application android:label = "@string/app_name"
                android:icon = "@drawable/icon">


    <activity android:name = "org.cocos2dx.cpp.AppActivity"
                android:label = "@string/app_name"
                android:screenOrientation = "portrait"                    ①
                android:theme = "@android:style/Theme.NoTitleBar.Fullscreen"
                android:configChanges = "orientation">

        <! -- Tell NativeActivity the name of our .so -->
        <meta - data android:name = "android.app.lib_name"
                    android:value = "cocos2dcpp" />

        <intent - filter>
            <action android:name = "android.intent.action.MAIN" />
            <category android:name = "android.intent.category.LAUNCHER" />
        </intent - filter>
    </activity>
</application>

<supports - screens android:anyDensity = "true"
                    android:smallScreens = "true"
                    android:normalScreens = "true"
                    android:largeScreens = "true"
                    android:xlargeScreens = "true"/>

    <uses - permission android:name = "android.permission.INTERNET"/>
</manifest>
```

上述第①行代码把 android:screenOrientation = "landscape" 改为 android:screenOrientation="portrait",android:screenOrientation 属性取值 landscape 为横屏显示,portrait 为竖屏显示。

本章小结

通过对本章的学习,读者可以了解到 Cocos2d-x Lua API 工程移植到 Android 平台需要做哪些工作,以及一些具体问题的汇总。

第 21 章

移植到 iOS 平台

本章介绍如何在 Win32 下进行游戏工程的开发并移植到本地 iOS 平台。

21.1 iOS 开发环境搭建

Xcode 是 Mac OS X 和 iOS 应用集成开发的工具。苹果公司于 2008 年 3 月 6 日发布了 iPhone 和 iPod Touch 的应用程序开发包，其中包括 Xcode 开发工具、iPhone SDK 和 iPhone 手机模拟器。第一个 Beta 版本是 iPhone SDK 1.2b1(build 5A147p)，它在发布后立即就能够使用，但是同时推出的 App Store 所需要的固件更新直到 2008 年 7 月 11 日才发布。编写本书时，iOS SDK 7 版本已经发布。

iOS 开发工具最主要的就是 Xcode。自从 Xcode 3.1 发布以后，Xcode 就成为 iPhone 软件开发工具包的开发环境。Xcode 可以开发 Mac OS X 和 iOS 应用程序，其版本是与 SDK 相互对应的。例如，Xcode 3.2.5 与 iOS SDK 4.2 对应，Xcode 4.1 与 iOS SDK 4.3 对应，Xcode 4.2 与 iOS SDK 5 对应，Xcode 4.5 与 iOS SDK 6 对应，Xcode 5 与 iOS SDK 7 对应。

Xcode 4.1 之前还有一个配套使用的工具 Interface Builder，它是 Xcode 套件的一部分，用来设计窗体和视图，通过它可以“所见即所得”地拖曳控件并定义事件等，其数据以 XML 的形式被存储在.xib 文件中。在 Xcode 4.1 之后，Interface Builder 成为 Xcode 的一部分，与 Xcode 集成在一起。

21.1.1 Xcode 安装和卸载

Xcode 必须安装在 Mac OS X 系统上，Xcode 的版本与 Mac OS X 系统版本有着严格的对应关系。Xcode 5 要求 Mac OS X 10.8 以上。

安装可以通过 Mac OS X 的 Dock 中的 App Store 应用，如图 21-1 所示。如果需要安装软件或查询软件，则需要用户登录，这个用户是 App ID 即可，弹出的登录对话框如图 21-2 所示。如果没有登录 App ID，则可以单击“创建 App ID”按钮。

之后，可以在右上角的搜索栏中输入要搜索的软件或工具名称 Xcode 关键字。搜索结果如图 21-3 所示。

图 21-1　启动 App Store 界面

图 21-2　App Store 用户登录界面

图 21-3　搜索 Xcode 工具

单击 Xcode 进入 Xcode 信息介绍界面，如图 21-4 所示，单击“安装 App”按钮开始安装。

图 21-4　Xcode 安装

卸载 Xcode 非常简单。实际上，在 Mac OS X 中应用程序只需直接删除即可。如图 21-5 所示打开应用程序，右击 Xcode，从弹出的快捷菜单中选择“移到废纸篓”即可删除 Xcode 应用。如果想彻底删除，只需清空废纸篓即可。

21.1.2　Xcode 操作界面

打开 Xcode 工具，看到的主界面如图 21-6 所示。该界面主要分成 3 个区域：①号区域是工具栏，其中的按钮可以完成大部分工作；②号区域是导航栏，主要是对工作空间中的内容进行导航；③号区域是代码编辑区，我们的编码工作就是在这里完成的。在导航栏上面还有一排按钮，如图 21-7 所示，默认选中的是“文件”导航面板。关于各个按钮的具体用法，会在以后用到的时候详细介绍。

在选中导航面板时，导航栏下面也有一排按钮，如图 21-8 所示。这是辅助按钮，它们的功能都与该导航面板内容相关。对于不同的导航面板，这些按钮也是不同的。

图 21-5　Xcode 卸载

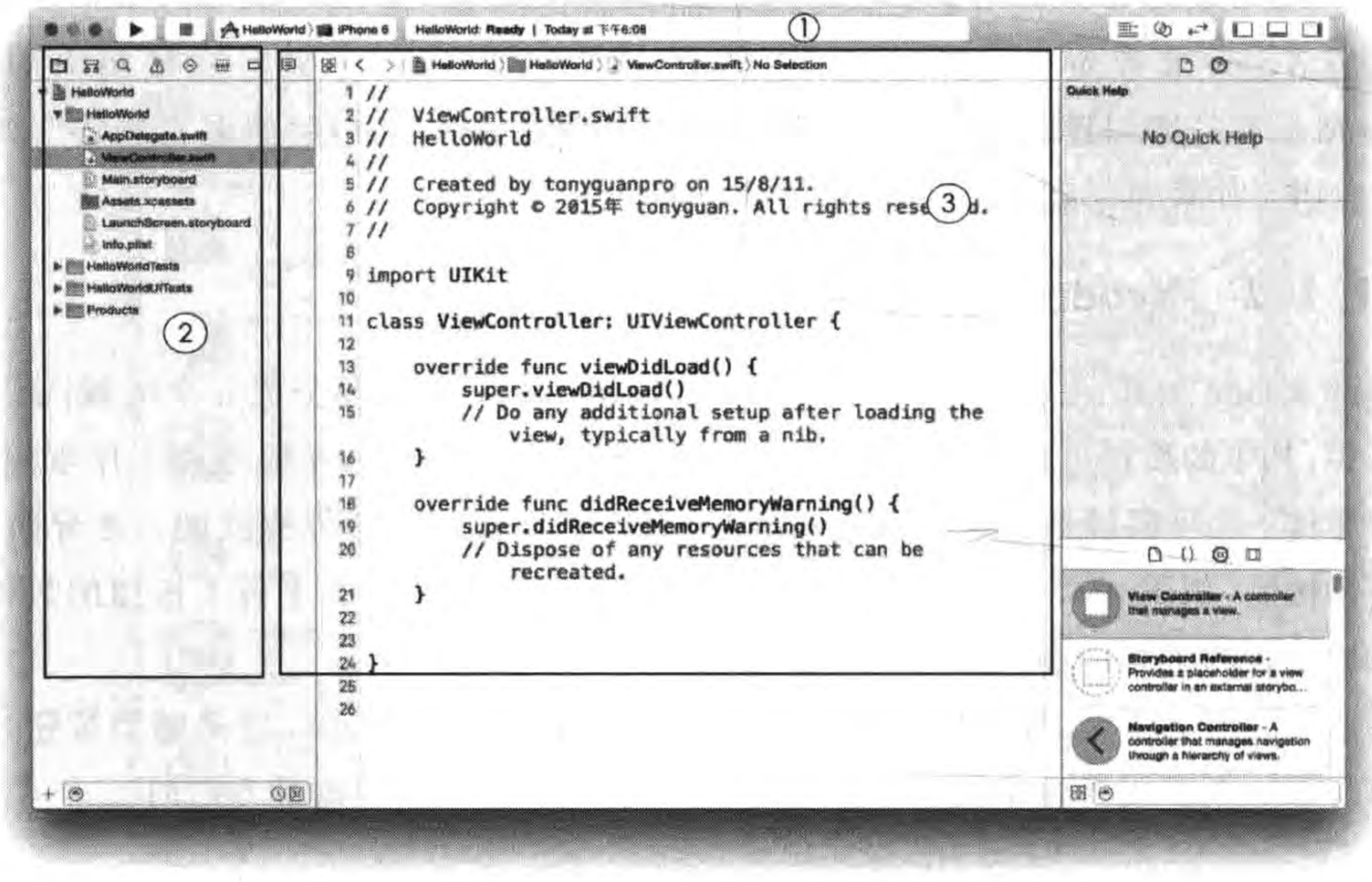


图 21-6　Xcode 主界面

图 21-7　Xcode 导航面板

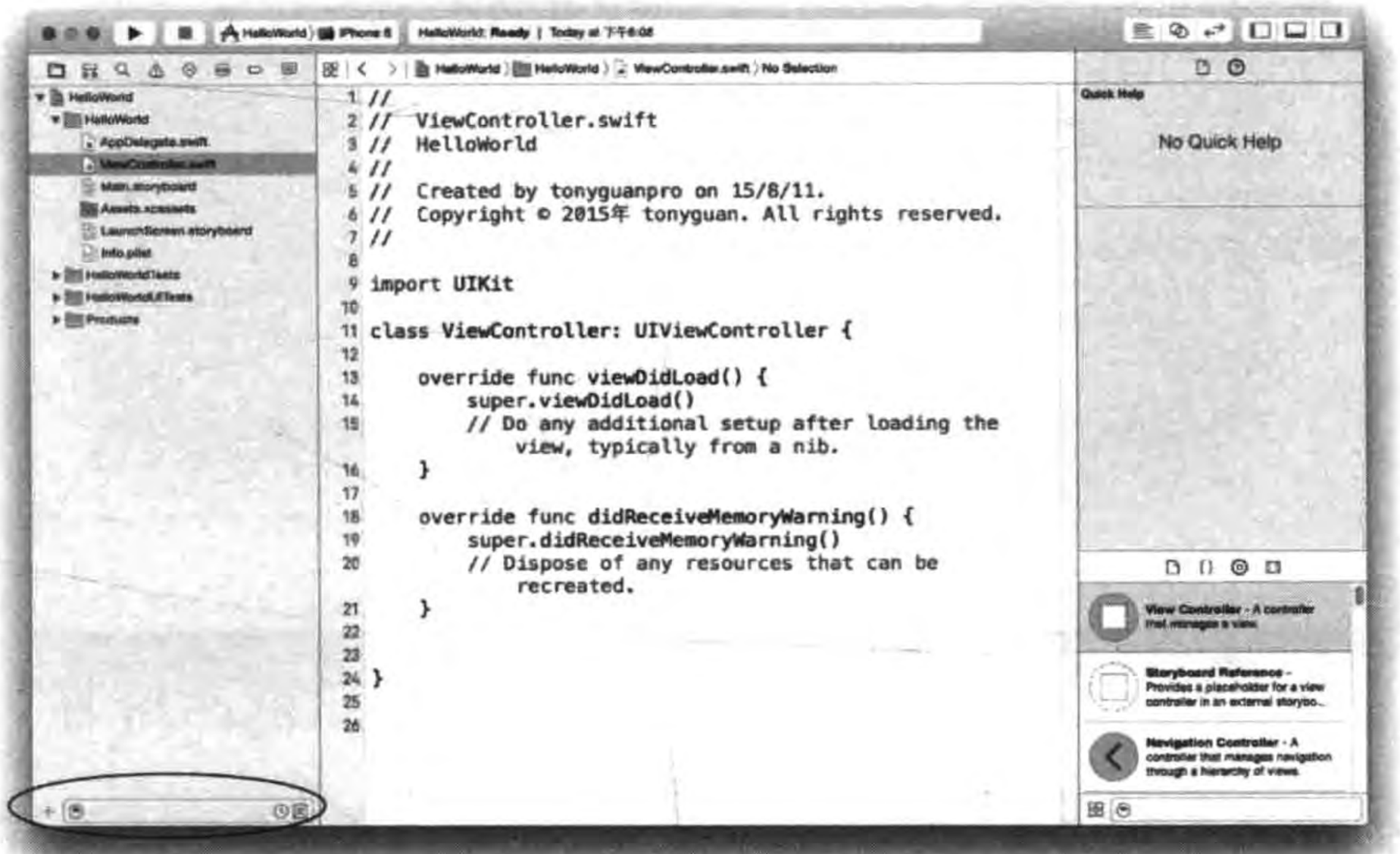


图 21-8　导航面板的辅助按钮

21.2　编译与运行

在 iOS 平台中编译与运行，可以有两种方式：

(1) 使用 cocos run -p ios 命令。

(2) 使用 Xcode 工具

使用 cocos run 命令类似于 Android 平台，这里不再赘述。本节重点介绍使用 Xcode 工具进行编译和运行。

首先找到<工程根目录>/frameworks/runtime-src/proj. ios_mac 下的. xcodeproj 工程文件，双击打开工程，如图 21-9 所示。

打开工程，按照图 21-10 所示，选择 HelloLua-mobile→iPhone 6s 后，再选择运行按钮。运行之前先进行编译，如果是第一次编译运行则比较慢。如果编译并运行成功则如图 21-11 所示界面。

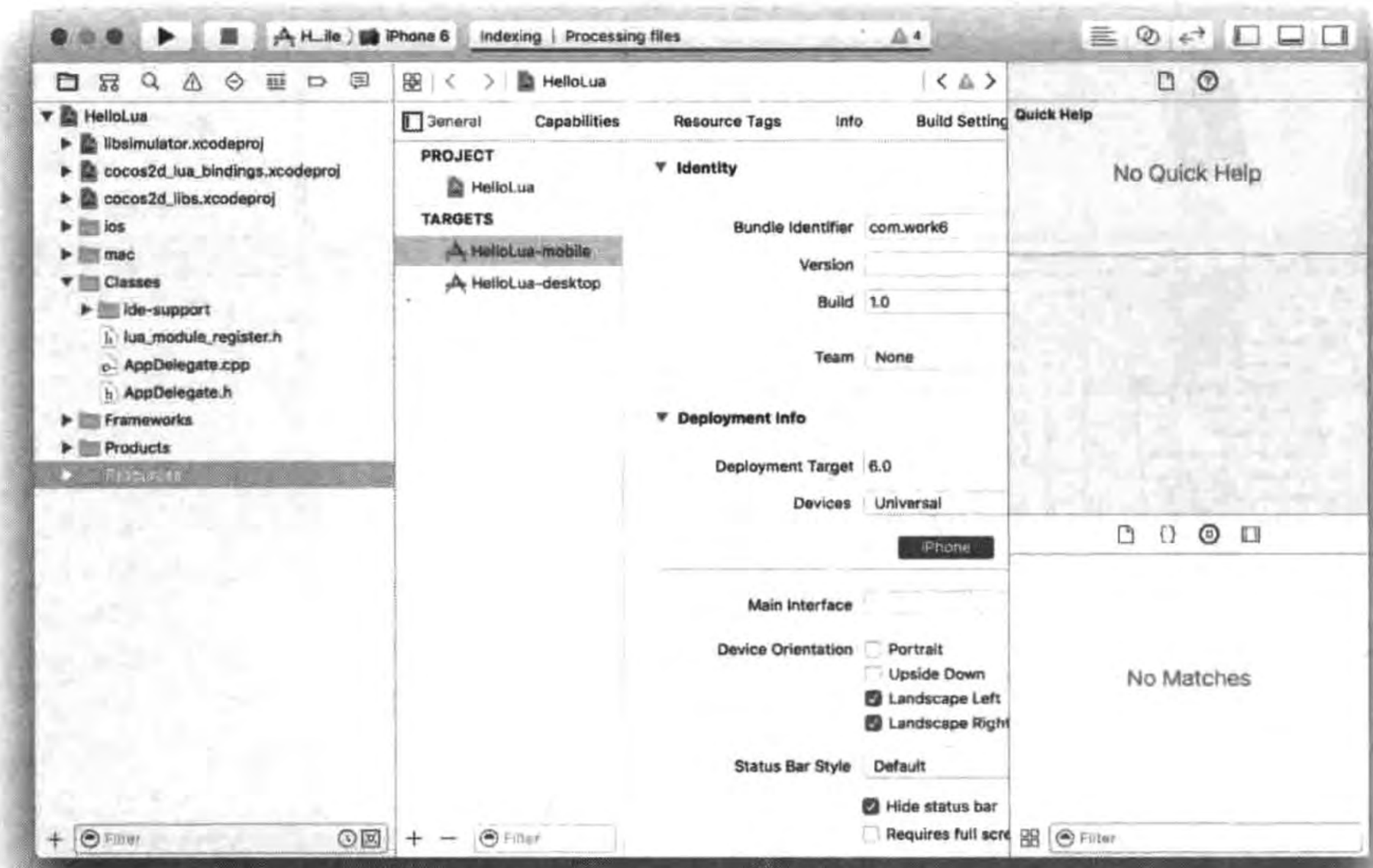


图 21-9　HelloLua. xcodeproj 工程文件

图 21-10　编译并运行工程

图 21-11　运行成功

21.3　移植问题汇总

安装好了 Xcode 工具，并且成功地在 iOS 工程中添加源文件和资源文件，就可以开始移植工作了。下面是我们归纳了移植到 iOS 平台遇到的一些问题。

21.3.1　iOS 平台声音移植问题

如果游戏要运行在 Mac OS X 或 iOS 平台，那么采用苹果特有格式的声音文件可以提升游戏性能。第 10 章介绍过苹果的声音格式有 CAFF 和 AIFF。CAFF 是无压缩音频格式，一般用于音乐特效优化，AIFF 和 AIFF-C 由于经过压缩，文件占用空间较小，一般用于背景音乐。

下面以 10.3 节移植到 iOS 平台为例介绍一下声音移植问题。首先需要将文件格式进行转换，音效转换命令：

```
$ afconvert -f caff -d LEI16@44100  Blip.wav
```

其中 Blip.wav 是文件名。通过该命令我们将所有的 wav 文件转换为 caf 文件。

背景音乐转换命令：

```
$ afconvert -f AIFC -d ima4 Synth.mp3
```

其中 Synth.mp3 是文件名。通过该命令将所有的 mp3 文件转换为 aifc 文件。全部转换完成后文件如图 21-12 所示。

在程序代码中需要判断平台，然后进行选择加载用哪些文件，修改 main.lua 文件，内容

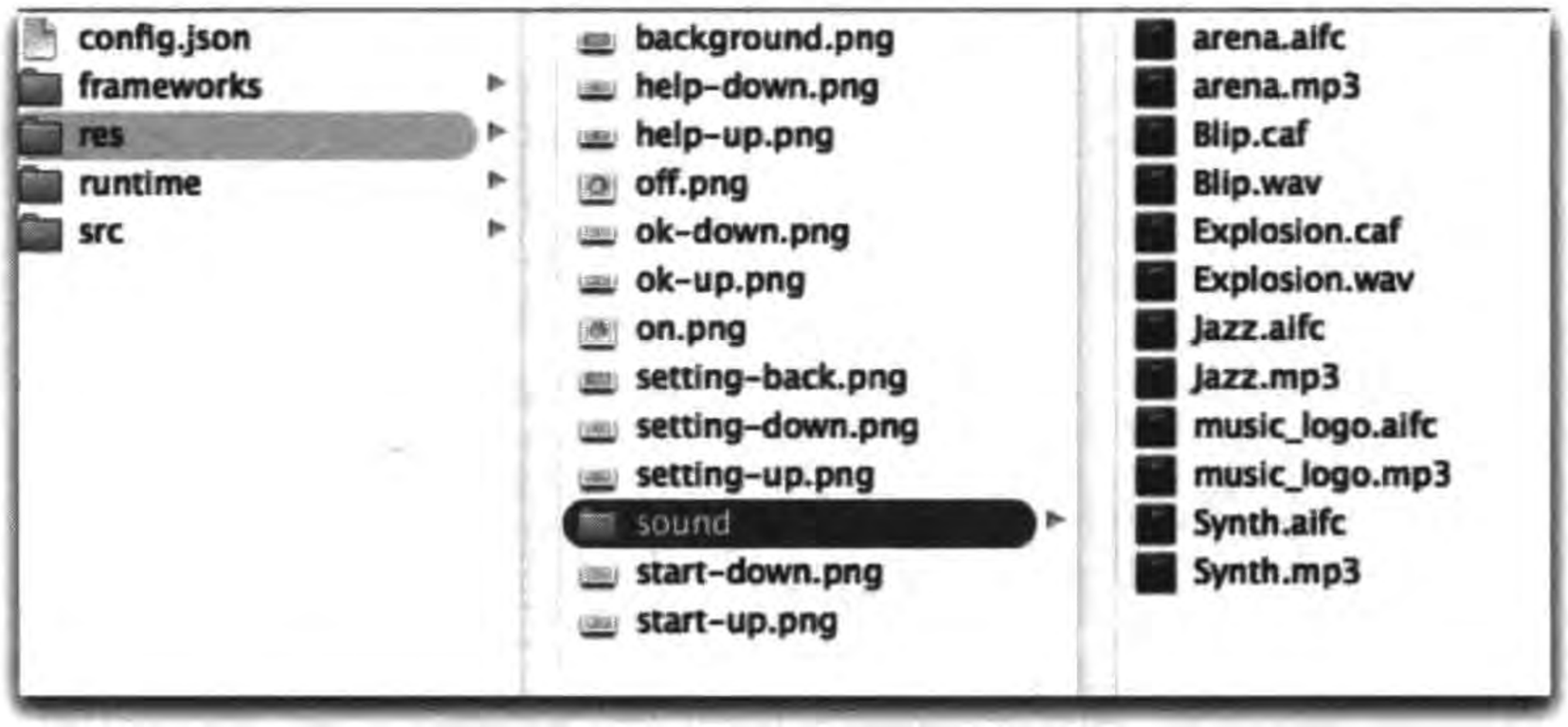


图 21-12　转换之后的文件

如下：

```
local targetPlatform = cc.Application:getInstance():getTargetPlatform()

if (cc.PLATFORM_OS_IPHONE == targetPlatform)                    ①
    or (cc.PLATFORM_OS_IPAD == targetPlatform)  then            ②
    bg_music_1   =   "sound/Jazz.aifc"
    bg_music_2   = "sound/Synth.aifc"
    sound_1    = "sound/Blip.caf"
else
    bg_music_1   =   "sound/Jazz.mp3"
    bg_music_2   = "sound/Synth.mp3"
    sound_1    = "sound/Blip.wav"
end
...
local function main()
    ...
    --初始化   音乐
    AudioEngine.preloadMusic(bg_music_1)
    AudioEngine.preloadMusic(bg_music_2)
    --初始化   音效
    AudioEngine.preloadEffect(sound_1)
    ...


end
```

上述第①～③行代码是判断 iOS 设备，即 iPhone、iPod touch 和 iPad，其中 targetPlatform 是前面定义的变量，cc. Application:getInstance():getTargetPlatform()表达式可以获得当前平台，表示平台的常量如下：

(1) cc. PLATFORM_OS_WINDOWS。

(2) cc. PLATFORM_OS_LINUX。
(3) cc. PLATFORM_OS_MAC。
(4) cc. PLATFORM_OS_ANDROID。
(5) cc. PLATFORM_OS_IPHONE。
(6) cc. PLATFORM_OS_IPAD。
(7) cc. PLATFORM_OS_BLACKBERRY。

21.3.2　使用 PVR 纹理格式

第 19 章(性能优化)已介绍过在 iOS 平台应该使用 PVR 纹理格式,PVR 纹理不需经过解码等处理能够直接被 iOS CPU 使用。

可以使用 TexturePacker 等工具制作纹理图集,需要注意的是,背景图片放到一个纹理集中,不需要 Alpha 通道,可以选择使用 RGB565 格式。而其他图片放到一个纹理图集中,可以采用 RGBA4444 格式。生成的文件如下:

(1) Texture. plist。非 iOS 平台纹理集 plist 文件。
(2) Texture. png。非 iOS 平台纹理集文件。
(3) Texture_bg. plist。非 iOS 平台背景图片纹理集 plist 文件。
(4) Texture_bg. png。非 iOS 平台背景图片纹理集文件。
(5) Texture_PVR_zlib. plist。iOS 平台纹理集 plist 文件。
(6) Texture_PVR_zlib. pvr. ccz。iOS 平台纹理集文件。
(7) Texture_bg_PVR_zlib. plist。iOS 平台背景图片纹理集 plist 文件。
(8) Texture_bg_PVR_zlib. pvr. ccz。iOS 平台背景图片纹理集文件。

修改 main. lua 文件程序代码如下:

```
require "Cocos2d"
require "AudioEngine"

local targetPlatform = cc.Application:getInstance():getTargetPlatform()
local spriteFrameCache = cc.SpriteFrameCache:getInstance()                    ①
...
local function main()
    ...
    --初始化　音乐
    AudioEngine.preloadMusic(bg_music_1)
    AudioEngine.preloadMusic(bg_music_2)
    --初始化　音效
    AudioEngine.preloadEffect(sound_1)

    if (cc.PLATFORM_OS_IPHONE == targetPlatform)
        or (cc.PLATFORM_OS_IPAD == targetPlatform) then                        ②

        spriteFrameCache:addSpriteFrames("Texture/Texture_PVR_zlib.plist")
```

```
            spriteFrameCache:addSpriteFrames("Texture/Texture_bg_PVR_zlib.plist")
        else
            spriteFrameCache:addSpriteFrames("Texture/Texture.plist")
            spriteFrameCache:addSpriteFrames("Texture/Texture_bg.plist")
        end
            ...

end
```

上述第①行代码是通过表达式 cc.SpriteFrameCache:getInstance()获得精灵帧缓存。第②行代码是判断苹果的3个平台。纹理集的加载时机对于游戏程序的性能是有影响的，在性能优化一章介绍过，这里不再赘述。

21.3.3 横屏与竖屏设置问题

Cocos2d-x Lua API 工程模板在 iOS 平台默认情况下是横屏显示的，因此需要修改设置，否则会出现图 21-13 所示的现象，图片会失真，正常显示应如图 21-14 所示。

图 21-13 横屏显示竖屏内容

图 21-14 正常显示

具体设置过程是：打开 Xcode 工程，选择工程(如 HelloLua)中的 TARGETS→HelloLua-mobile→General→Device Orientation，如图 21-15 所示，只选中 Portrait。Device Orientation 属性是设置设备的支持方向，Portrait 表示竖直朝上，Upside Down 表示竖直朝下，Landscape Left 和 Landscape Right 分别表示水平向左和水平向右。

修改完成后就可以运行并查看效果了。

21.4 多分辨率屏幕适配

这一节介绍解决多分辨率屏幕适配问题。只考虑移植到 iOS 平台。

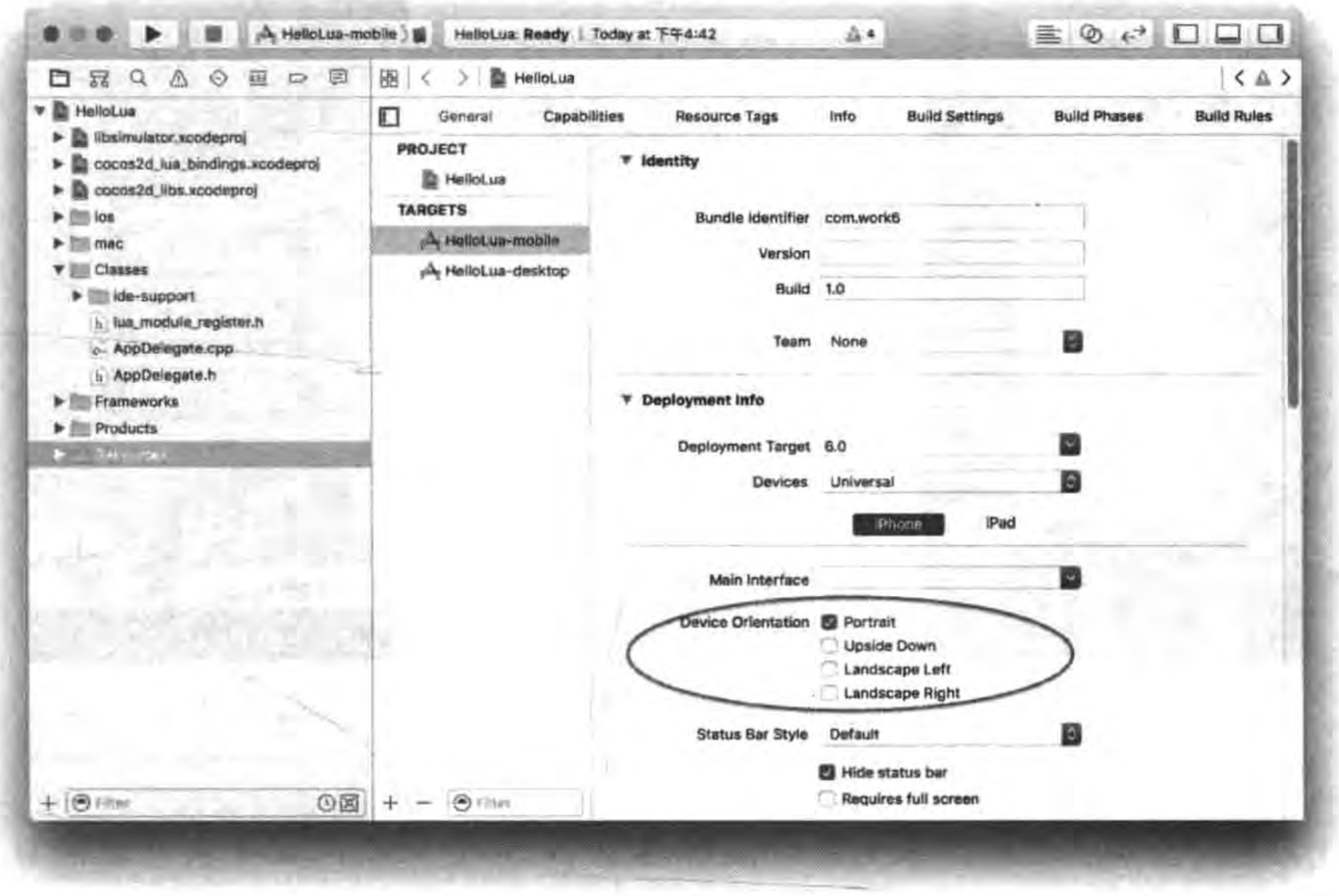


图 21-15　设置竖屏

21.4.1　问题的提出

很多初学者会感到困惑，设备屏幕适配真的有那么大的危害吗？下面比较一个 iOS 平台的实例。图 21-16 所示是屏幕适配有问题的情况，其中图(a)的资源图片是 320×480 像素，屏幕尺寸 iPhone 3.5 英寸 Retina 显示屏[①]，分辨率是 640×960 像素设备，这种情况下图片太小了，周围用黑色填充。图(b)的资源图片是 640×1136 像素，屏幕尺寸是 640×960 像素设备，导致图片上下超出屏幕。图(c)的资源图片是 640×960 像素，屏幕尺寸是 640×1136 像素设备，导致屏幕上下有黑边。

21.4.2　Cocos2d-x Lua API 屏幕适配

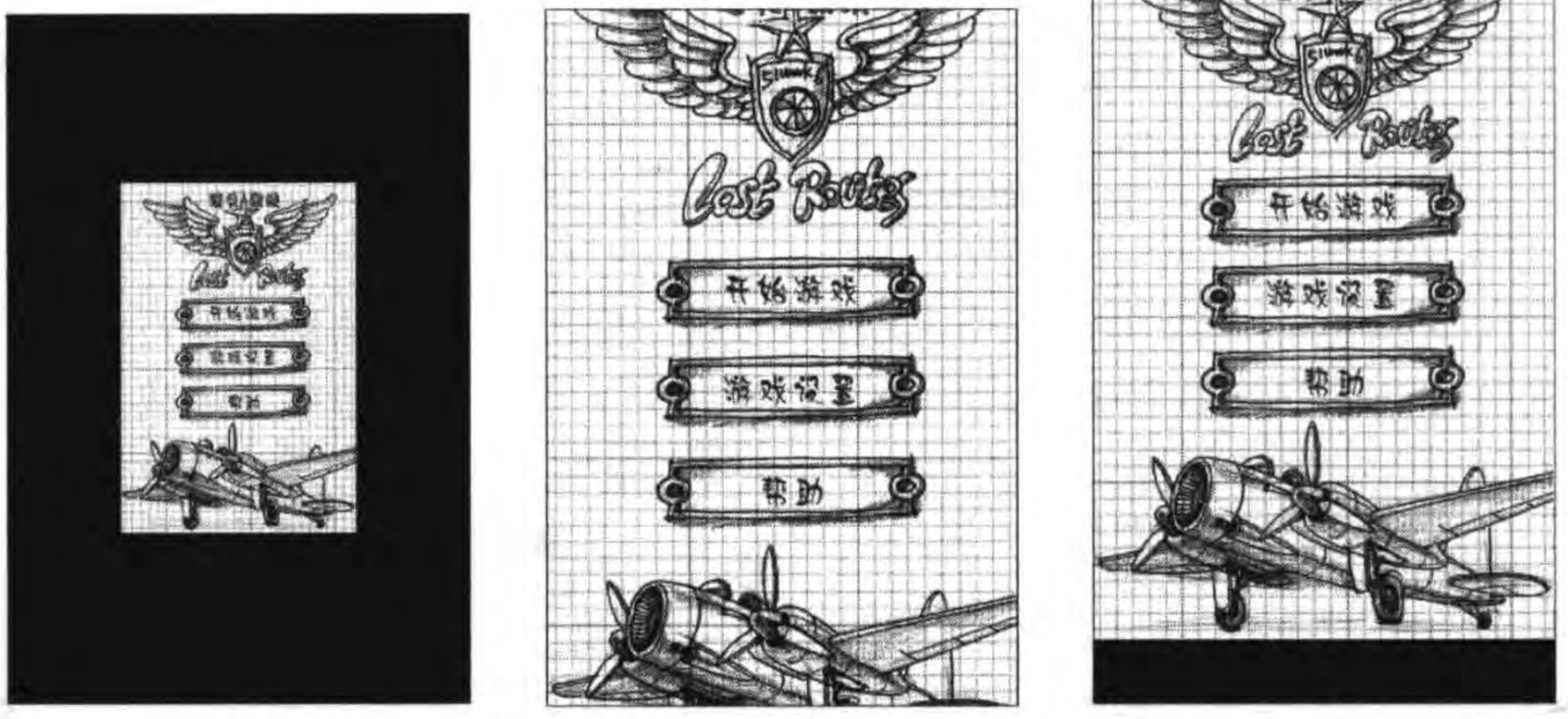
Cocos2d-x 给出了解决屏幕适配问题的方案，首先看看它的原理。Cocos2d-x Lua API 定义了 3 种分辨率：资源分辨率、设计分辨率和屏幕分辨率。

(1) 资源分辨率。也就是资源图片的大小，单位是像素。

(2) 设计分辨率。逻辑上游戏屏幕的大小，在这里设置其分辨率为 320×480 像素，那么在游戏中设置精灵的位置便可以参考该值，如左下角 cc.p(0,0)，右上角 cc.p (320, 480)。

---

① Retina 显示屏，是苹果高清显示屏，它在屏幕上的一个点是 4 个像素，高和宽是普通显示屏的两倍，例如，在 iPhone 3.5 英寸普通显示屏分辨率是 320×480 像素，而 iPhone 3.5 英寸 Retina 显示屏分辨率是 640×960 像素。



(a) (b) (c)

图 21-16 多分辨率屏幕适配

(3) 屏幕分辨率。是以像素为单位的屏幕大小,对于 iPhone 3.5 英寸普通显示屏是 320×480 像素,而对于 iPhone3.5 英寸 Retina 显示屏是 640×960 像素。

从资源文件显示到屏幕上分为两个阶段:资源分辨率到设计分辨率设置和设计分辨率到屏幕分辨率设置,这两个阶段如图 21-17 所示。

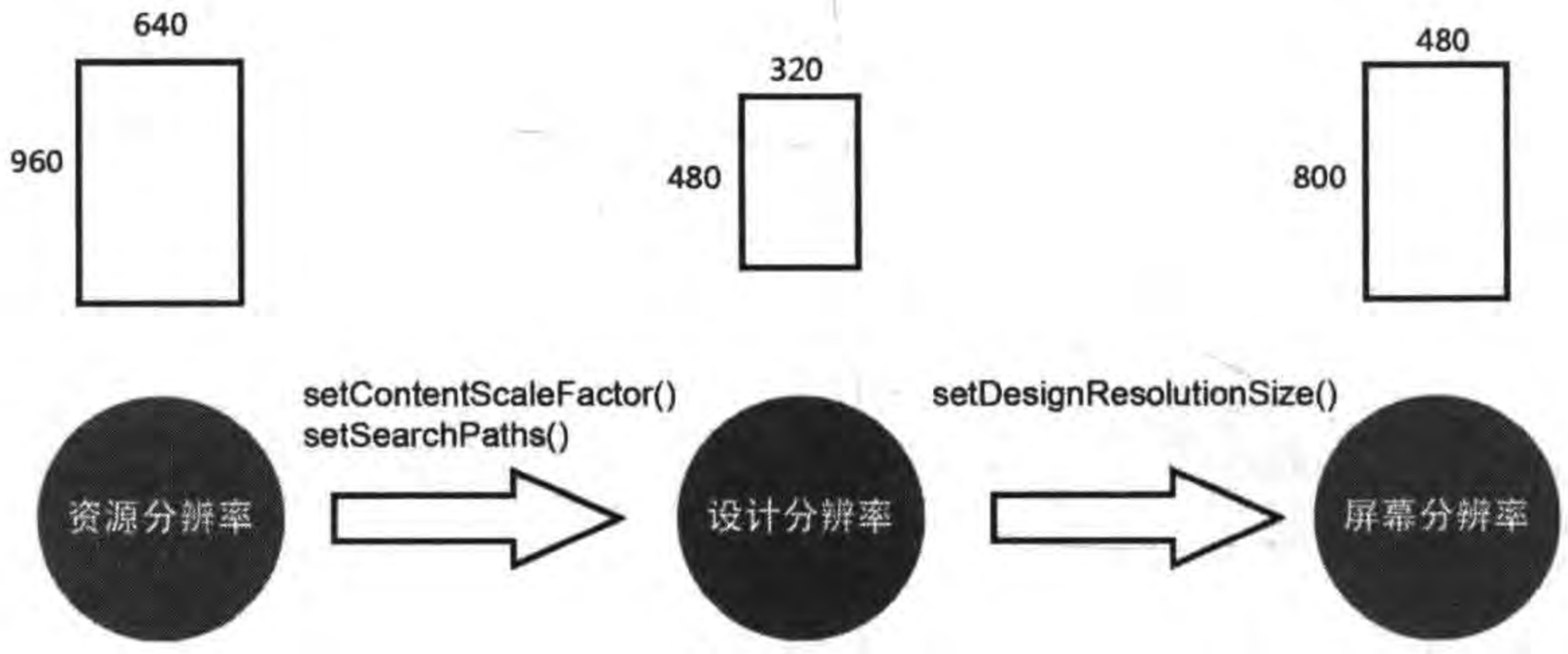


图 21-17 资源文件显示到屏幕上的两个阶段

第一个阶段从资源分辨率到设计分辨率,使用了 Director 的实例函数 setContentScaleFactor()和 setSearchPaths()控制转换过程,这一个转换的目的是统一坐标单位,不管资源分辨率大小,全部统一到设计分辨率的大小,将设计分辨率坐标单位作为标准。第二个阶段从设计分辨率到屏幕分辨率,通过 GLView 的实例函数 setDesignResolutionSize()设置这个过程。

setContentScaleFactor()是设置缩放因子(ContentScaleFactor),它是指资源大小和设计分辨率大小的比例系数,可以通过 resourceSize. height/designSize. height 或 resourceSize.

width/designSize.width 的做法来设置，其中 resourceSize 表示资源的大小，designSize 表示设计分辨率的大小。

setSearchPaths()函数是设置搜索路径，每个路径下面有一套资源文件，如图 21-18 所示有 3 套资源文件，为此创建了 3 个目录，hd 对应 iPhone 3.5 英寸 Retina 显示屏，hd5 对应 iPhone 4 英寸 Retina 显示屏，sd 是普通显示屏。在这些目录中有着相同名字的资源文件。

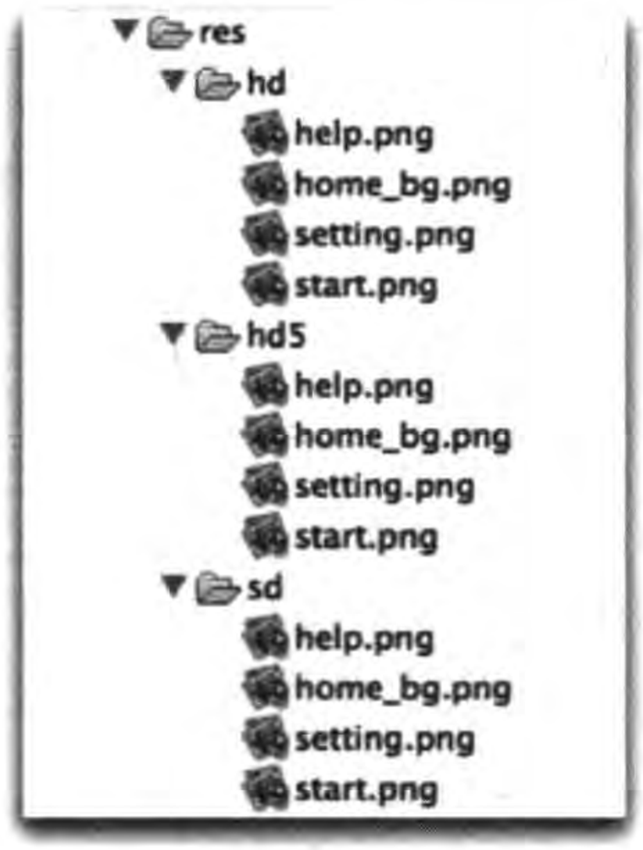


图 21-18　资源目录

下面看看实例代码，main.lua 代码如下：

```
local function main()
    collectgarbage("collect")
    -- avoid memory leak
    collectgarbage("setpause", 100)
    collectgarbage("setstepmul", 5000)

    local sharedFileUtils = cc.FileUtils:getInstance()
    local glview = cc.Director:getInstance():getOpenGLView()

    sharedFileUtils:addSearchPath("src")
    sharedFileUtils:addSearchPath("res")

    --屏幕大小
    local screenSize = glview:getFrameSize()                                    ①
    --设计分辨率大小
    local designSize = cc.size(320,480)                                         ②
    --资源大小
    local resourceSize = cc.size(320,480)                                       ③

    local searchPaths = sharedFileUtils:getSearchPaths()                        ④
    local resPrefix = "res/"
    if screenSize.height > 960 then                                             ⑤
        designSize = cc.size(320, 568)
        resourceSize = cc.size(640, 1136)
        table.insert(searchPaths, 1, resPrefix.."hd5")                          ⑥
    elseif screenSize.height > 480 then
        resourceSize = cc.size(640, 960)
        table.insert(searchPaths, 1, resPrefix.."hd")
    else
        table.insert(searchPaths, 1, resPrefix.."sd")
    end
    sharedFileUtils:setSearchPaths(searchPaths)                                 ⑦

    cc.Director:getInstance():setContentScaleFactor(resourceSize.width/designSize.width)
                                                                                ⑧
    glview:setDesignResolutionSize(designSize.width, designSize.height,
                                   cc.ResolutionPolicy.FIXED_HEIGHT)            ⑨

    -- create scene
    local scene = require("GameScene")
    local gameScene = scene.create()
```

```
    if cc.Director:getInstance():getRunningScene() then
        cc.Director:getInstance():replaceScene(gameScene)
    else
        cc.Director:getInstance():runWithScene(gameScene)
    end
end
```

上述第①～⑨行代码是为了实现屏幕适配而追加的代码,其中第①行代码是获得设备屏幕的大小。第②行代码是设置设计分辨率。第③行代码是设置资源分辨率。第④行代码是声明搜索路径变量 searchPaths,searchPaths 是列表类型,可以容纳多个路径。第⑤行代码是判断屏幕的高度是否大于 960,也就是 iPhone 4 英寸 Retina 显示屏情况,这种情况下重新设置了设计分辨率和资源分辨率大小,然后通过第⑥行代码的 table.insert(searchPaths, 1, resPrefix.."hd5")语句设置搜索路径 hd5。类似地还判断了其他几种情况。第⑦行代码 sharedFileUtils:setSearchPaths(searchPaths)是设置搜索路径。

第⑧行代码是按照资源宽和设计分辨率宽比例作为缩放因子。第⑨行代码 glview:setDesignResolutionSize()是设置设计分辨率大小和适配策略,designSize.width 设计分辨率的宽,designSize.height 设计分辨率的高,cc.ResolutionPolicy.FIXED_HEIGHT 设计分辨率策略,有关分辨率策略将在下一节介绍。

21.4.3 分辨率策略

在上一节中 glview:setDesignResolutionSize(designSize.width, designSize.height, cc.ResolutionPolicy.FIXED_HEIGHT)语句中用到了分辨率策略 cc.ResolutionPolicy.FIXED_HEIGHT,cc.ResolutionPolicy.FIXED_HEIGHT 只是适配策略的一种,Cocos2d-x 3.x 提供了 5 种适配策略,其表示常量如下:

(1) cc.ResolutionPolicy.NO_BORDER。无边策略。

(2) cc.ResolutionPolicy.FIXED_HEIGHT。固定高度。

(3) cc.ResolutionPolicy.FIXED_WIDTH。固定宽度。

(4) cc.ResolutionPolicy.SHOW_ALL。全显示策略。

(5) cc.ResolutionPolicy.EXACT_FIT。精确配合。

1. NO_BORDER

屏幕宽、高分别和设计分辨率宽、高计算缩放因子,取较大者作为宽、高的缩放因子。保证了设计区域总能在一个方向上铺满屏幕,而在另一个方向一般会超出屏幕区域,如图 21-19 所示,设计分辨率是 320×480 像素,屏幕分辨率是 480×800 像素。

提示 图中 SH 表示屏幕分辨率的高,SW 表示屏幕分辨率的宽,DH 表示设计分辨率的高,DW 表示设计分辨率的宽,scaleX 是 *X* 轴缩放因子,scaleY 是 *Y* 轴缩放因子,MAX 函数表示取最大值,这些表示方式与后面篇幅中含义一样,不再解释。

NO_BORDER 没有拉伸图像,同时在一个方向上撑满了屏幕,是我们推荐的策略。但

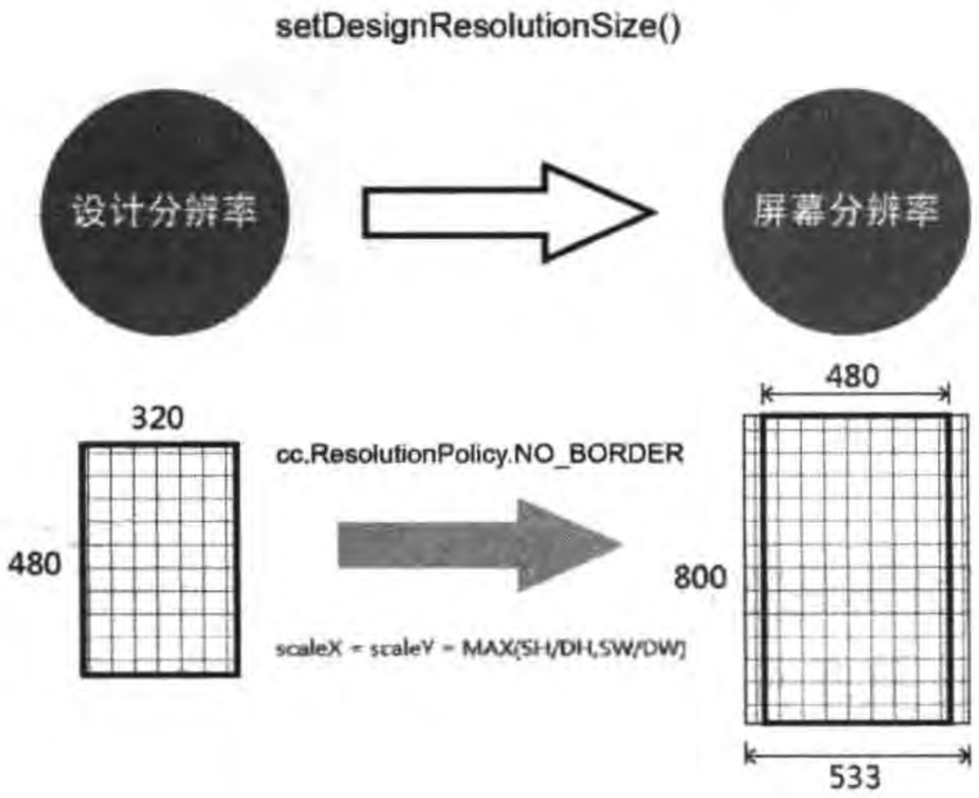


图 21-19　NO_BORDER 策略

是这种策略有一个问题，它不能根据我们的意愿向特定方向填充满屏幕，如果有这样的需要可以使用 FIXED_HEIGHT 和 FIXED_WIDTH。

2. FIXED_HEIGHT 和 FIXED_WIDTH

FIXED_HEIGHT 和 FIXED_WIDTH 可以保证图像按照固定的高或宽无拉伸填充满屏幕，如图 21-20 所示，其中 ceilf 函数是取参数的上限，例如 ceilf(5.61) = 6。

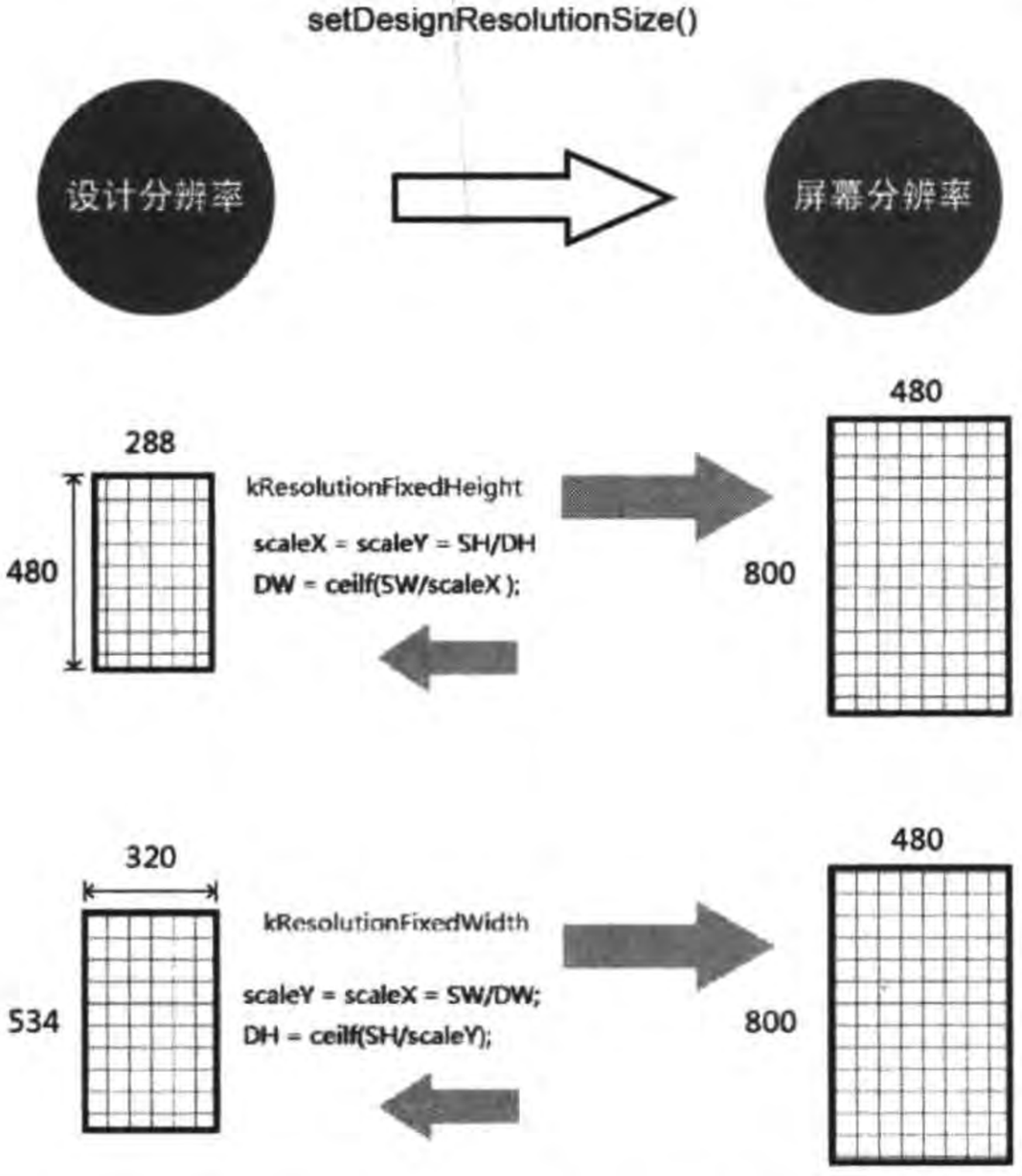


图 21-20　FIXED_HEIGHT 和 FIXED_WIDTH 策略

FIXED_HEIGHT 适合高方向需要填充满、宽方向可以裁剪的游戏界面。FIXED_WIDTH 适合宽方向需要填充满、高方向可以裁剪的游戏界面。FIXED_HEIGHT 经常结

合 setContentScaleFactor(屏幕分辨率高度/设计分辨率高度)使用。FIXED_WIDTH 经常结合 setContentScaleFactor(屏幕分辨率宽度/设计分辨率宽度)使用。

3. SHOW_ALL

屏幕宽、高分别和设计分辨率宽、高计算缩放因子,取较小者作为宽、高的缩放因子。保证了设计区域全部显示到屏幕上,但可能会有黑边。如图 21-21 所示,其中 MIN 函数取最小值。

4. EXACT_FIT

屏幕分辨率宽与设计分辨率宽成比例,作为 x 轴方向的缩放因子,屏幕分辨率高与设计分辨率高成比例,作为 y 轴方向的缩放因子。保证了设计区域完全铺满屏幕,但可能会出现图像拉伸的现象,如图 21-22 所示。

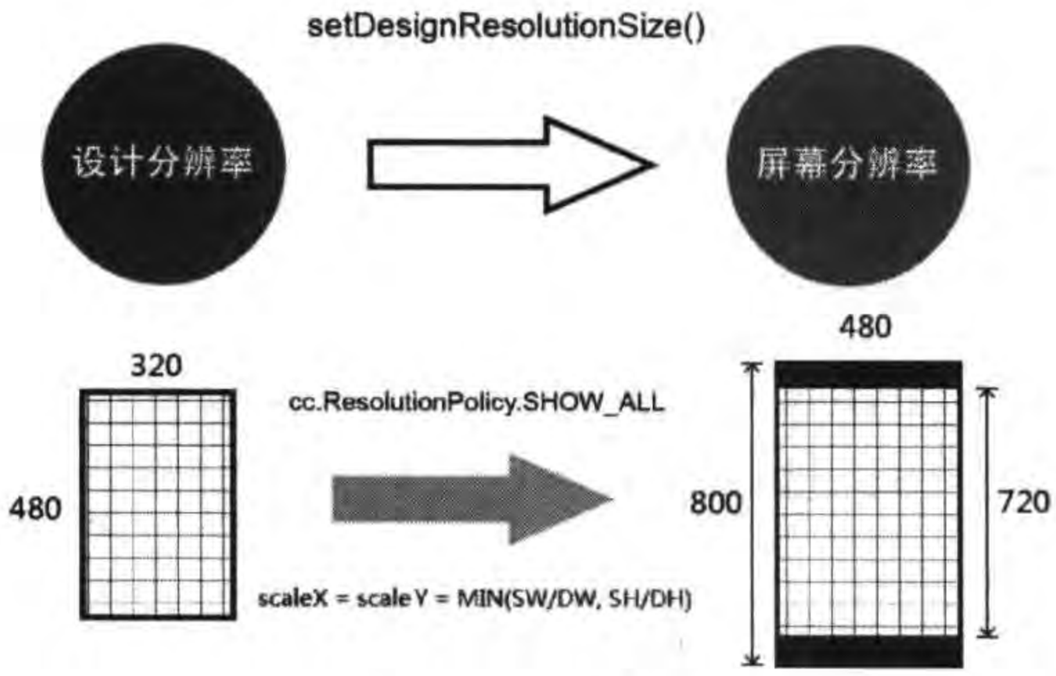


图 21-21 SHOW_ALL 策略

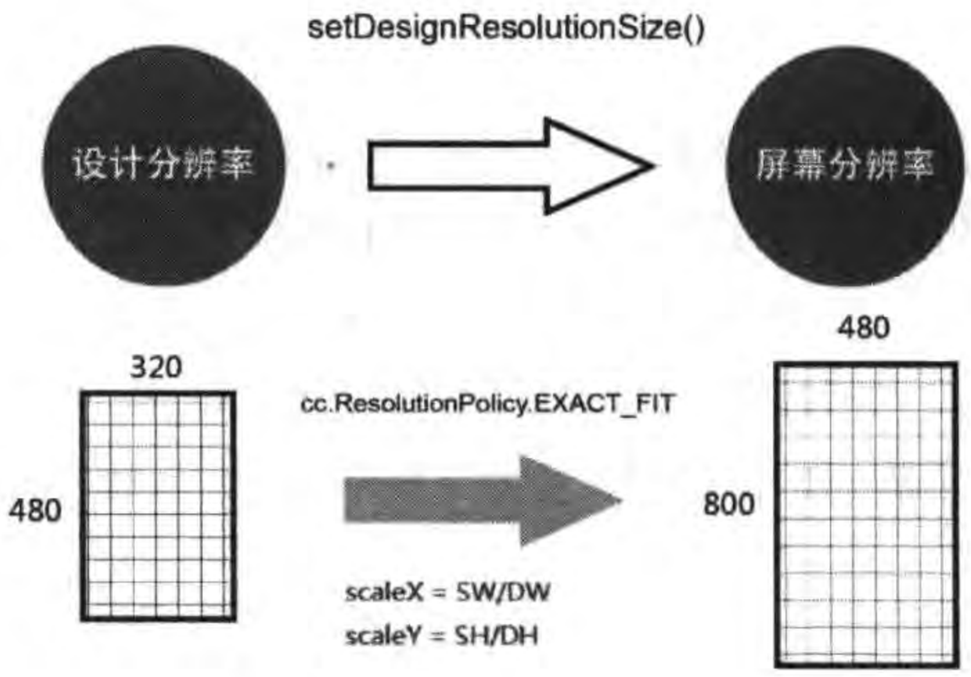


图 21-22 EXACT_FIT 策略

总结上面的 5 种策略,重点推荐使用 FIXED_HEIGHT 和 FIXED_WIDTH,次之是 NO_BORDER。除非特殊需要,一定要全部显示填充屏幕可以使用 EXACT_FIT,一定要全部无变形显示可以使用 SHOW_ALL。

21.4.4 纹理图集资源适配

在游戏中经常会使用纹理图集,那么纹理图集应该放置到什么位置呢? 它的图片资源目录结构一致就可以了。目录结构如图 21-23 所示,为每一套分辨率提供了一套纹理图集。在使用它们的时候,不用考虑 hd、hd5 和 sd 等搜索路径。

代码如下:

```
local targetPlatform = cc.Application:getInstance():getTargetPlatform()
local spriteFrameCache = cc.SpriteFrameCache:getInstance()
...
local function main()
    collectgarbage("collect")
    -- avoid memory leak
    collectgarbage("setpause", 100)
    collectgarbage("setstepmul", 5000)
```

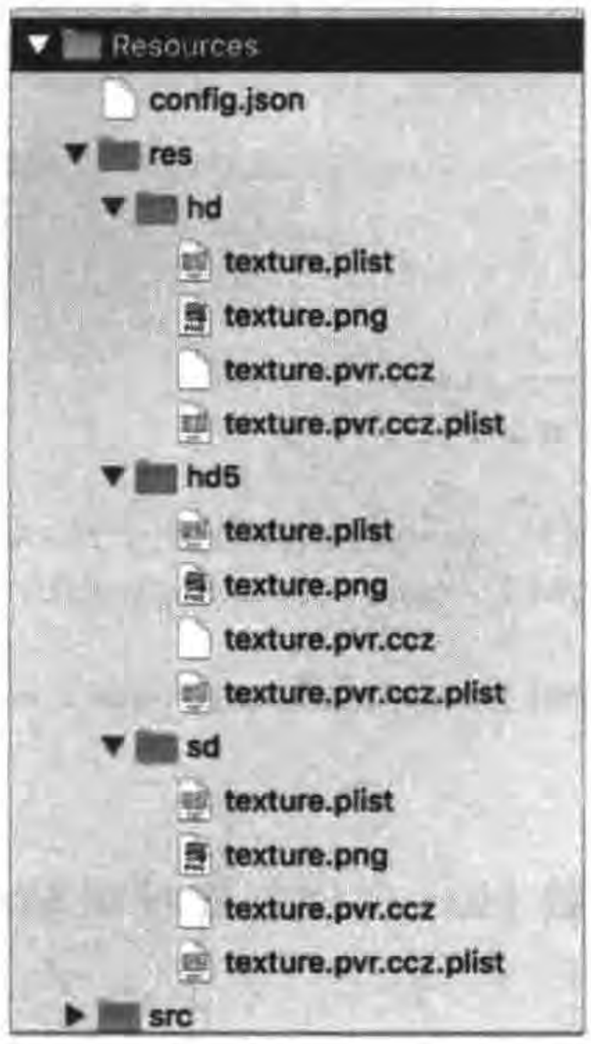


图 21-23　纹理图集资源目录结构

```
local sharedFileUtils = cc.FileUtils:getInstance()
local glview = cc.Director:getInstance():getOpenGLView()

sharedFileUtils:addSearchPath("src")
sharedFileUtils:addSearchPath("res")

--屏幕大小
local screenSize = glview:getFrameSize()
--设计分辨率大小
local designSize = cc.size(320,480)
--资源大小
local resourceSize = cc.size(320,480)

local searchPaths = sharedFileUtils:getSearchPaths()
local resPrefix = "res/"
if screenSize.height > 960 then
    designSize = cc.size(320, 568)
    resourceSize = cc.size(640, 1136)
    table.insert(searchPaths, 1, resPrefix.."hd5")
elseif screenSize.height > 480 then
    resourceSize = cc.size(640, 960)
    table.insert(searchPaths, 1, resPrefix.."hd")
else
    table.insert(searchPaths, 1, resPrefix.."sd")
end
sharedFileUtils:setSearchPaths(searchPaths)

cc.Director:getInstance():setContentScaleFactor(resourceSize.width/designSize.width)
--默认为 1.0
glview:setDesignResolutionSize(designSize.width, designSize.height, cc.ResolutionPolicy.FIXED_
HEIGHT)

if (cc.PLATFORM_OS_IPHONE == targetPlatform)
```

```
    or (cc.PLATFORM_OS_IPAD == targetPlatform) then
    spriteFrameCache:addSpriteFrames("texture.pvr.ccz.plist")          ①
else
    spriteFrameCache:addSpriteFrames("texture.plist")                  ②
end

-- create scene
local scene = require("GameScene")
local gameScene = scene.create()

if cc.Director:getInstance():getRunningScene() then
    cc.Director:getInstance():replaceScene(gameScene)
else
    cc.Director:getInstance():runWithScene(gameScene)
end
```

上述第①和②行代码分别加载 plist 文件创建精灵帧缓存，从代码中可以看出不需要再添加资源路径了。

21.4.5 瓦片地图资源适配

除了纹理图集需要考虑适配问题，瓦片地图也需要考虑适配问题。在设计时可以为每一套分辨率提供一套瓦片地图，需要注意的是，瓦片地图文件 tmx 一般都会跟有一个瓦片集图片，需要把 tmx 文件和瓦片集图片放在同一个目录下，如图 21-24 所示。

图 21-24 瓦片地图资源目录结构

通过程序代码 local tileMap = cc.TMXTiledMap:create("orthogonal-test6.tmx")就可以加载创建瓦片地图对象 TMXTiledMap 了，从代码中可以看出不需要再添加资源路径了。

本章小结

通过对本章的学习，读者可以了解到把 Cocos2d-x Lua API 工程移植到 iOS 平台需要做哪些工作，以及一些具体问题的汇总。

第六篇　实　战　篇

本篇共 4 章，介绍 Cocos2d-x 游戏项目开发中的实际问题，其中包括：使用 Git 管理程序代码版本、迷失航线手机游戏开发过程、发布游戏到 Google play 应用商店和苹果 App Store 应用商店。

本篇各章如下：

第 22 章　使用 Git 管理程序代码版本

第 23 章　Cocos2d-x Lua API 敏捷开发项目实战——迷失航线手机游戏

第 24 章　把迷失航线游戏发布到 Google play 应用商店

第 25 章　把迷失航线游戏发布到苹果 App Store 应用商店

第 22 章

使用 Git 管理程序代码版本

有位朋友曾经做了一个时长 3 个月的项目，就在项目即将结束的时候他的硬盘坏了，数据全部丢失，而他之前从不做备份，结果可想而知。这件事对我的触动很大，后来经常备份程序代码并将其备份在不同的计算机上。有一段时间每天下班，都把程序代码备份到公司的服务器。随着时间的推移，备份到服务器上的数据越来越多，很难快速查到想要的资料。在与同事进行代码整合时，用 U 盘互相复制，由于版本不能及时更新造成了很多问题。其中必须解决的问题有以下两个：

(1) 程序代码的备份。便于查到历史版本，能够进行比较，知道修改过什么地方，是谁修改了这些代码等。

(2) 代码共享与整合。能很容易得到团队其他人的代码，也能够很容易地把代码共享给其他成员。

解决这两个问题时，可以使用版本控制工具。版本控制工具是一种软件，开发人员要习惯使用版本控制工具，每日提交程序代码，提交的代码应该有清晰的注释，成员之间应该及时沟通。

版本控制工具很多，本章主要介绍 Git 代码版本控制工具。

22.1 代码版本管理工具——Git

版本控制的重要性毋庸置疑，必须使用代码版本控制工具，每一个程序员和项目管理人员都必须深刻认识到这一点。

22.1.1 版本控制历史

版本控制的最早方式是将文件复制到文件服务器上，命名为“××年××月××日×××备份”目录，这在现在看来太原始了。作为软件工具，版本控制经历过两个阶段：集中管理模式和分布式管理模式。

(1) 集中管理模式。以一个服务器作为代码库，团队人员本地没有代码库，只能与服务器进行交互。这种类型的版本控制工具有 VSS(Visual Source Safe，微软开发的 Microsoft

Visual Studio 套件中的软件之一)、CVS(Concurrent Versions System,并发版本系统)、SVN(Subversion)等。其中,SVN 是目前这种模式的佼佼者。

(2) 分布式管理模式。这是更为先进的模式,不仅有一个中心代码库,而且每位团队人员本地也都有代码库,在不能上网的情况下也可以提交代码。该类型的版本控制工具有 Git、Mercurial、Bazzar 和 Darcs。

Git 是为了帮助管理 Linux 内核开发项目而开发的一个开源的版本控制工具。之前,BitKeeper 工具是 Linux 内核开发人员使用的主要版本控制工具,它采用许可证管理版本,但 Linus Torvalds(著名的计算机程序员、黑客,Linux 内核的发明人及该计划的合作者)觉得 BitKeeper 不适合 Linux 开源社区的工作,在 2005 年开始着手开发 Git 来替代 BitKeeper。虽然 Git 最初是为了辅助 Linux 内核开发,但我们发现在很多其他软件项目中也都可以使用 Git。

22.1.2 术语和基本概念

版本控制工具涉及很多术语和概念。常用的概念如下:

(1) 代码库(repository)。存放项目代码以及历史备份的地方。

(2) 分支(branch)。为了验证和实验一些想法、版本发布、缺陷修改等需要,建立一个开发主干之外的分支,这个分支被隔离在各自的开发线上。当改变一个分支中的文件时,这些更改不会出现在开发主干和其他分支中。

(3) 合并分支(merging branch)。完成某分支工作后,将该分支上的工作成果合并到主分支上。

(4) 签出(check out)。从代码库获得文件或目录,将其作为副本保存在工作目录下,此副本包含了指定代码库的最新版本。

(5) 提交(commit)。将工作目录中修改的文件或目录作为新版本复制回代码库。

(6) 冲突(conflict)。有时提交文件或目录时可能会遇到冲突,当两个或多个开发人员更改文件中的一些相同行时,将发生冲突。

(7) 解决(resolution)。遇到冲突时,需要人为干预解决,这必须通过手动编辑该文件进行处理,必须有人逐行检查该文件,以接受一组更改并删除另一组更改。除非冲突解决,否则存在冲突的文件无法成功提交到代码库中。

(8) 索引(index)。Git 工具特有的概念。在修改的文件提交到代码库之前做出一个快照,这个快照称为“索引”,它一般会暂时存储在一个临时存储区域中。

22.1.3 Git 环境配置

为了使用 Git,需要到 http://git-scm.com/网站下载 Git 工具,并根据操作系统选择不同安装程序。这里以 Windows 为例介绍一下环境的配置过程。笔者下载的安装程序是 Git-2.9.0-64-bit.exe,双击即可安装。只要根据向导进行安装,安装成功后,在桌面上会看到快捷图标或者在程序菜单中出现 Git Bash,双击 Git Bash 即可运行软件。运行界面如

图 22-1 所示。在 Git Bash 中可以接受 UNIX 命令，也可以执行 Git 命令。

图 22-1　Git Bash 界面

Git Bash 在 Linux 和 Mac OS X 中是不需要安装的，在这两个平台下安装完 Git 软件后，进入终端，即可直接执行 Git 命令。

22.1.4　Git 常用命令

无论项目是否需要与他人协同开发，都会用到 Git 的一些常用命令。在上一节中就用到了 git add、git commit 和 git push 命令。本节将介绍一些常用的 Git 命令，包括 git help、git log、git init、git add、git rm、git commit 和 git status 等，以及 Git 图形界面辅助工具 gitk。

1. git help

第一个要掌握的是 git help，通过它可以自己查找命令的帮助信息，在终端中执行如下命令即可：

```
$ git help <命令>
```

其中，help 后面是要查询的命令。

2. git log

通过 git log 命令可以查看 Git 的日志信息。在终端中执行 git log 命令：

```
$ git log
```

执行结果为：

```
commit 2f027fbad790fa3e61ec9965a18415203f5e9683
Author: tonyguan <si92@sina.com>
Date:   Wed Oct 31 12:57:13 2012 +0800

    a

commit ac29dd648c7e78bfed5682742855683b5242c27f
Author: git on zhao - VirtualBox <git@zhao - VirtualBox>
Date:   Wed Oct 31 12:53:27 2012 +0800

    start
```

3. git init

该命令可以创建一个新的代码库,或者是初始化一个已存在的代码库。例如,想在本地创建一个 myrepo 代码库,可以先使用 mkdir 创建这个目录,然后再执行 git init。在终端中执行如下命令:

```
$ mkdir myrepo
$ cd myrepo
$ git init
Initialized empty Git repository in /Users/tonyguan/myrepo/.git/
```

使用 mkdir 创建 myrepo 目录(它只是一个普通的目录,并不是代码库)后,git init 会在 myrepo 目录下生成一个隐藏的.git 目录。

4. git add

git add 命令用来更新索引,记录下哪些文件有修改,或者添加了哪些文件。该命令并没有更新代码库,只有在提交的时候才将这些变化更新到代码库中。在终端中执行如下命令:

```
$ git add .
```

可以将当前工作目录和子目录下所有新添加和修改的文件添加到索引中。如果只想将某个文件添加到索引中,可以使用如下命令:

```
$ git add filename
```

或

```
$ git add *.txt
```

这里可以指定文件名,也可以使用通配符指定文件名。

5. git rm

git rm 命令用于删除索引或代码库中的文件,然后通过提交命令将变化更新到代码库中。在终端中执行命令:

```
$ git rm filename
```

或

```
$ git rm *.txt
```

6. git commit

git commit 命令用于更新缓存中的索引,但未被保存到代码库中的内容。在终端中执行命令:

```
$ git commit -m 'tony commit'
```

其中-m 设定提交注释信息。

7. git status

git status 命令可以显示当前 git 的状态,包括那些修改、删除和添加了的文件,但是没有提交的信息。在终端中执行命令:

```
$ git status
```

会显示类似如下的内容：

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#   (commit or discard the untracked or modified content in submodules)
#
#   modified:   .DS_Store
#   modified:   HelloWorld (modified content, untracked content)
#
no changes added to commit (use "git add" and/or "git commit -a")
```

8. gitk

gitk 并不是一个命令，而是一个图形工具，用来辅助管理 Git，在终端中直接输入 gitk 就可以启动。图 22-2 所示是 gitk 图形界面。

图 22-2　gitk 图形界面

在 gitk 工具中，可以查看日志、提交信息、注释、分支信息等，使用起来非常方便。

22.2　代码托管服务——GitHub

若开发团队成员来自世界各地，但又不想搭建 Git 服务器，因为这个投入比较大，那么可以选择 Git 代码托管服务。这就有点像买不起房子，可以租房子住。

在众多 Git 代码托管服务中，GitHub(https://github.com)公认是最好的，GitHub 是全球最大的编程社区及代码托管网站，它可以提供基于 Git 版本控制系统的代码托管服务。GitHub 同时提供商业账户和免费账户：免费账户不能创建私有项目；而商业账户中，每个公有项目不能超过 300MB 的存储空间，但是 300MB 的空间限制并非硬性限制，如果不存在滥用情况，可以申请扩增托管空间。如果团队成员分散在不同的地方，使用 GitHub 代码托管服务是一个不错的选择。

22.2.1 创建和配置 GitHub 账号

只有用户注册了账号，GitHub 才能提供服务。进入 https://github.com/pricing 网址(见图 22-3)，可以创建免费账号、收费账号和收费组织。

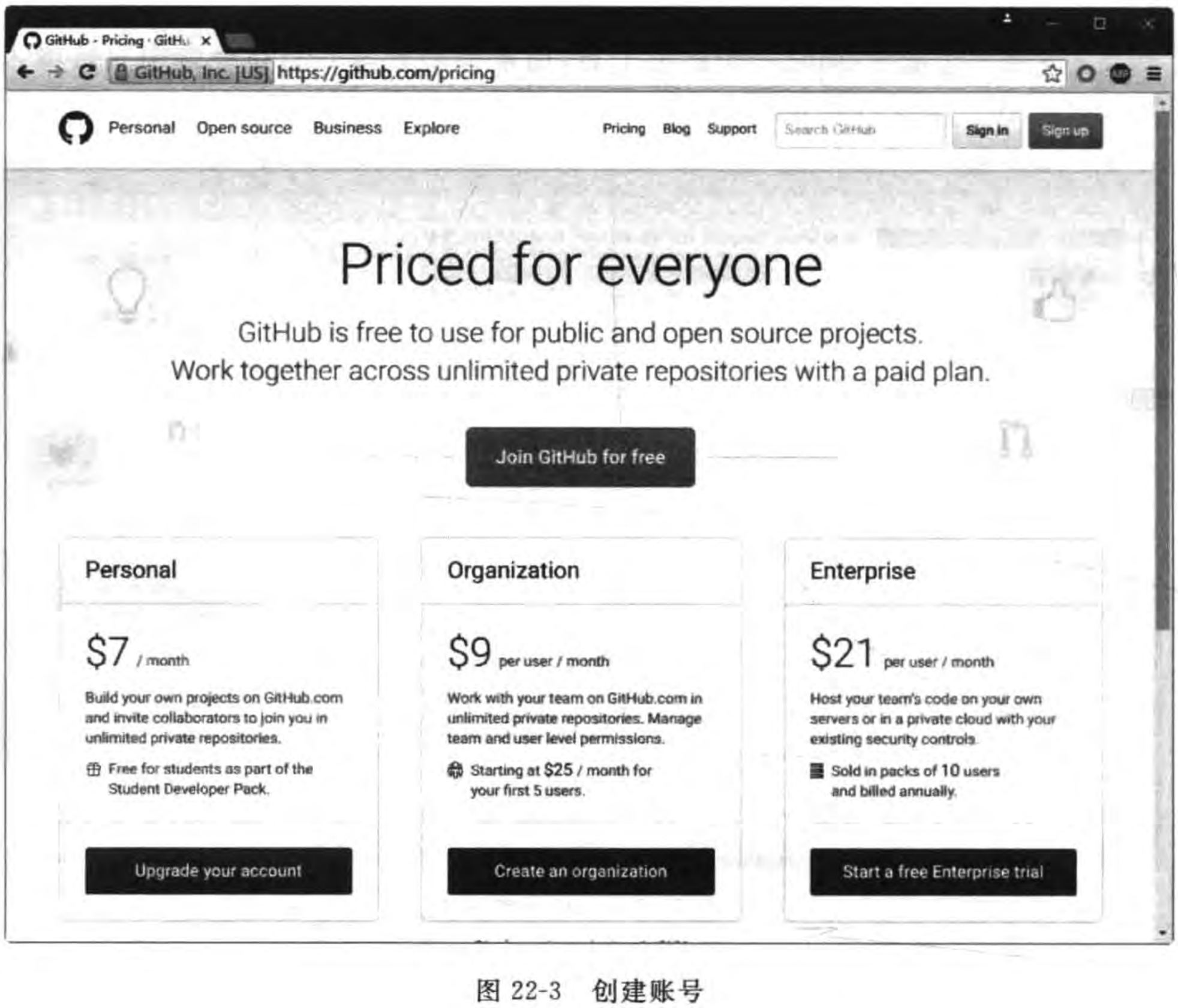


图 22-3 创建账号

这里以创建免费账号为例。单击 Upgrate your accout 按钮，进入创建免费账号页面，输入账号、邮箱和密码，验证通过就可以创建了。进入网址 https://github.com/login 可以登录，这里可以使用刚才创建的账号测试一下。登录成功后的页面如图 22-4 所示。

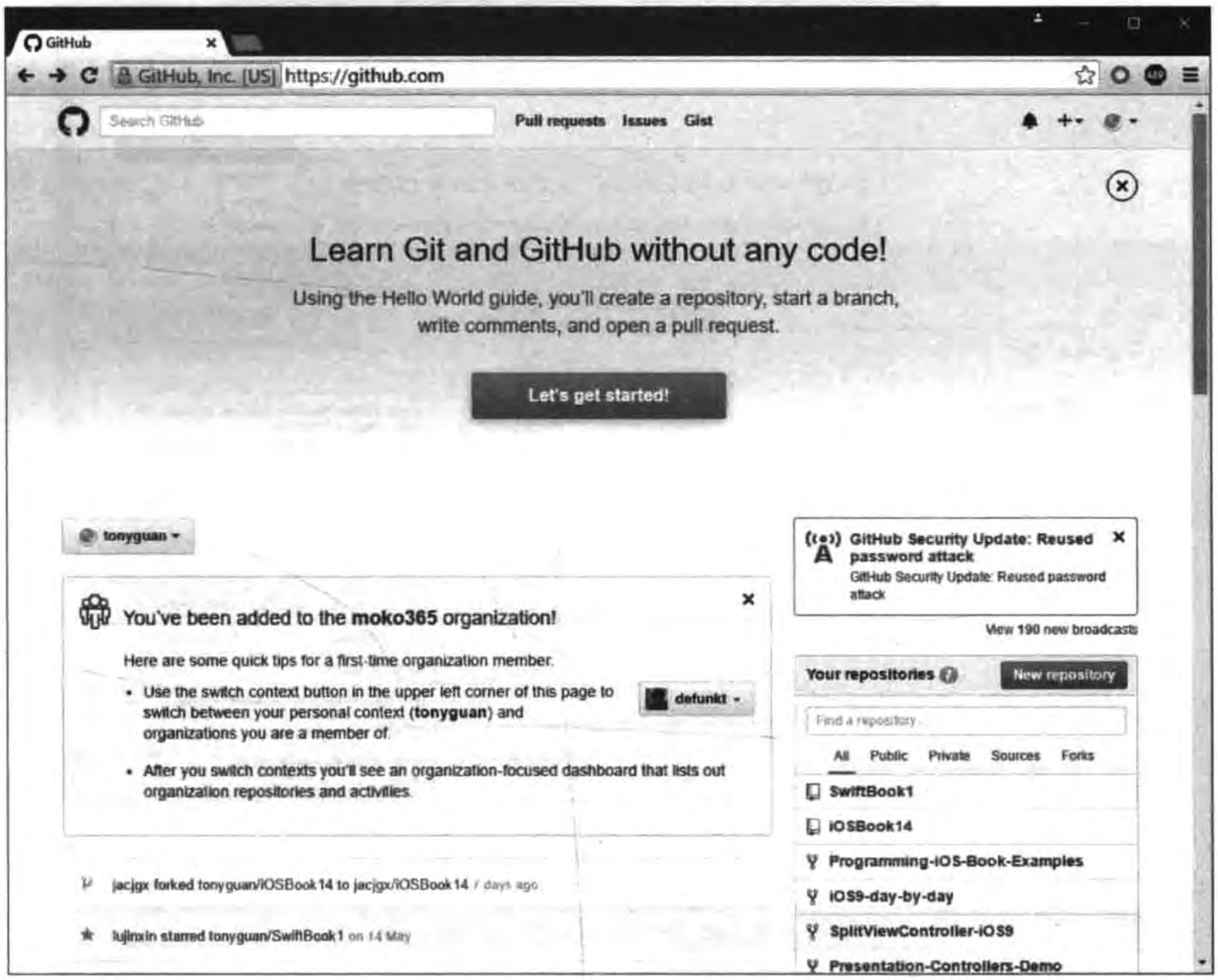


图 22-4　登录成功页面

我们知道，Git 常用的协议是 SSH 协议，这就需要在本机上生成后，将公钥提供给 GitHub 网站。单击页面右上角用户图标，会看到图 22-5 所示下拉菜单，选择其中的 Settings(或直接输入网址 https://github.com/settings/profile)，进入设置页面，然后从右边的 Profile 列表框中选择 SSH and GPG keys 项，再单击 New SSH key 按钮，此时会进入图 22-6 所示的页面。

将本机生成的 SSH 公钥(在 id_rsa.pub 文件中)粘贴到 Key 文本框中，并在 Title 文本框中输入一个标题。另外读者如果对生成 SSH key 不熟悉，可以单击 generating SSH keys 超链接查看帮助，注意在 Windows 下创建 SSH key 需要在 Git Bash 执行 ssh-keygen 命令：

```
$ ssh-keygen -t rsa -C "your_email@example.com"
```

在 Title 文本框中输入名字，可以随便命名。在 Key 文本框中输入生成的公钥，用文本工具打开后将其复制到这里。然后单击 Add SSH key 按钮提交内容，接着再次确认密码才能成功。这里可以提交多个 key，用来管理同一用户在不同机器上的登录情况。

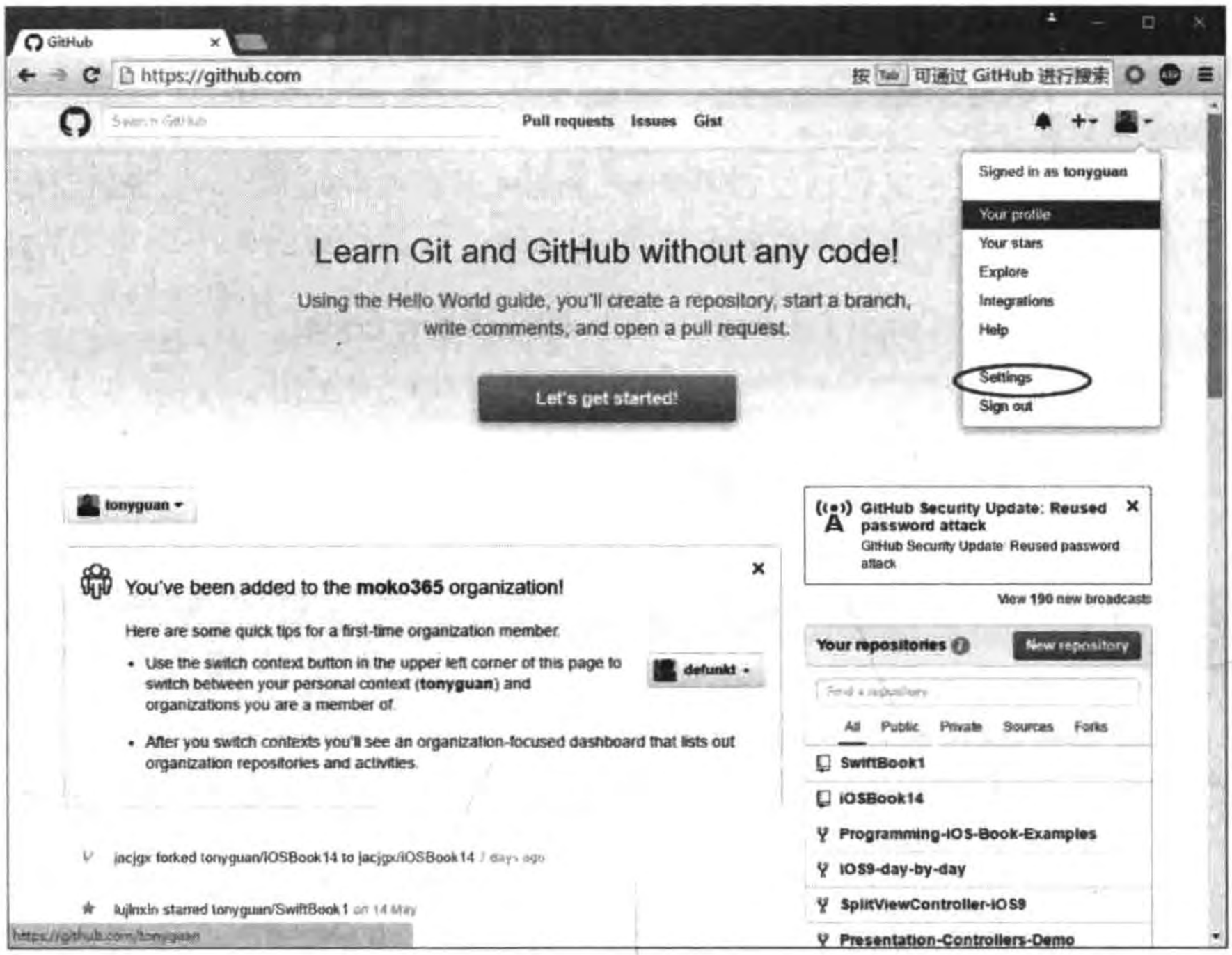


图 22-5　进入用户信息页面

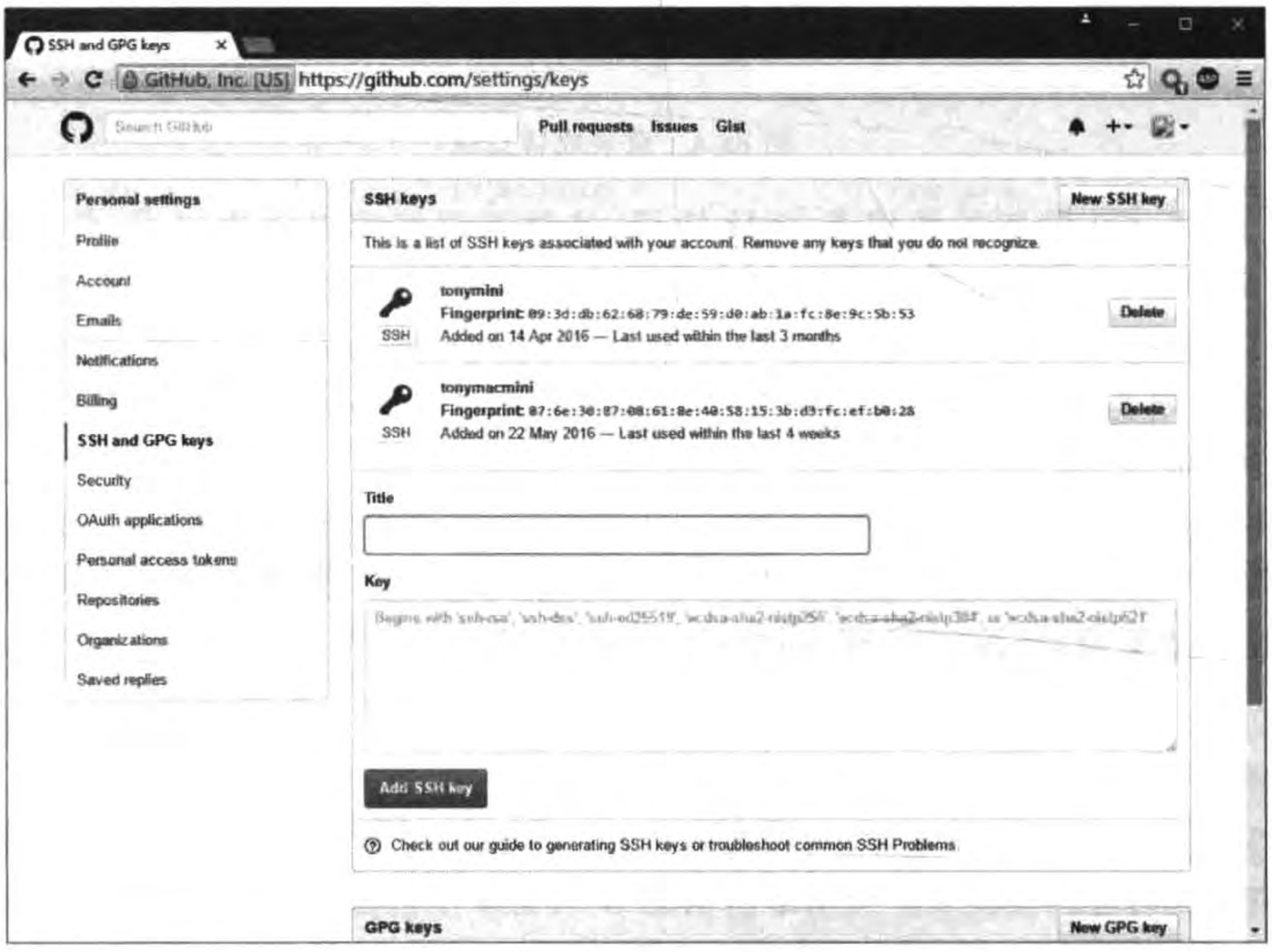


图 22-6　添加 SSH key 成功的页面

22.2.2　创建代码库

在 GitHub 中，代码库可以自己创建，也可以从别人那里派生(fork)过来。图 22-7 所示是代码库列表。其中，Cocos2d-x 是从名为 Cocos2d 的用户那里派生过来的，其他的两个库是自己创建的。

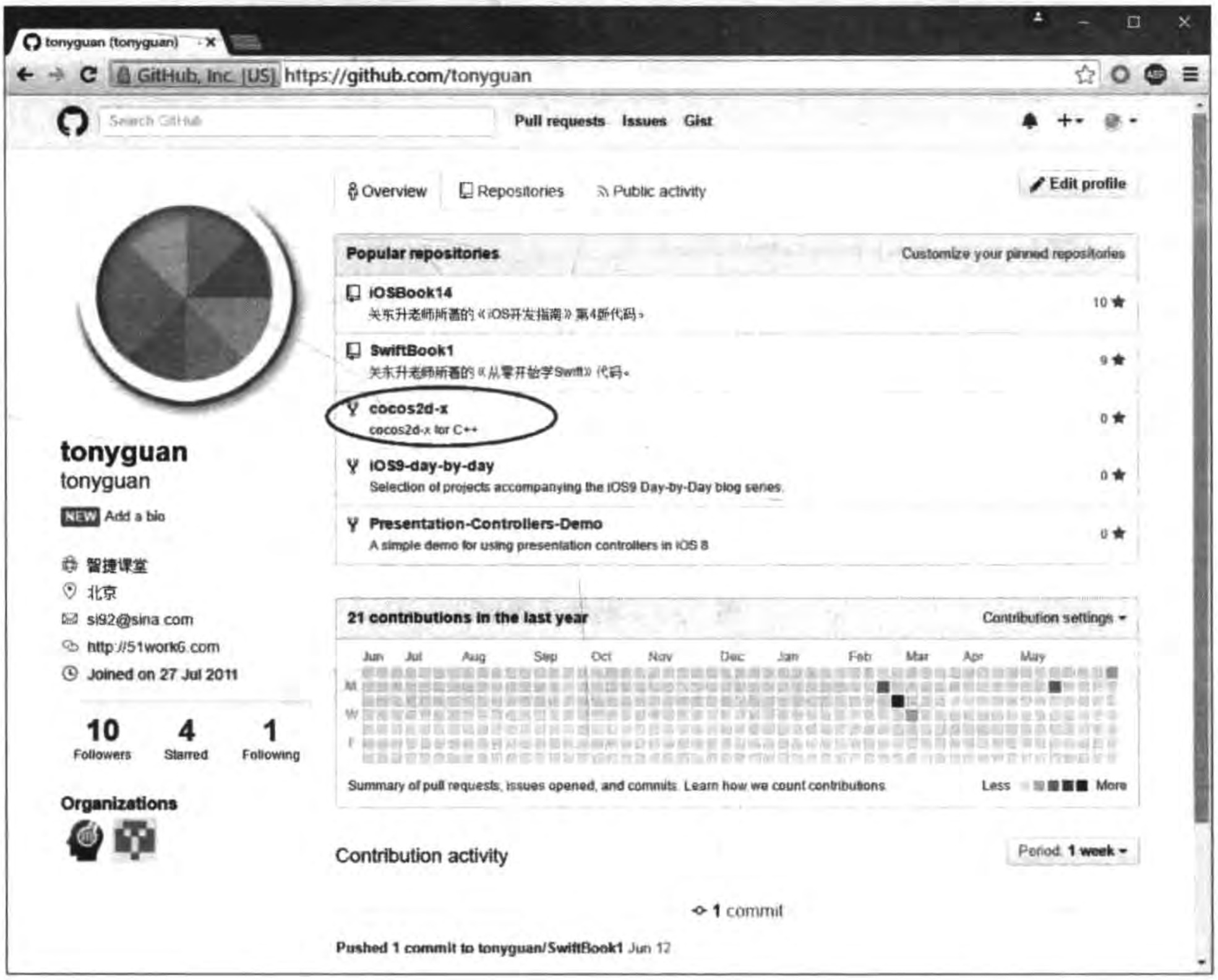


图 22-7　代码库列表

如果要在 GitHub 中创建代码库，可以在登录成功页面中的右下角单击 New repository 按钮，如图 22-8 所示。

此时进入图 22-9 所示的创建代码库页面。可以在其中输入代码库的名字和描述信息。在这里，可以创建私有代码库或公有代码库。需要说明的是，只有付费账户才可以创建私有库。此外，还可以选择是否创建一个 README 文件来说明这个代码库。

至此，单击 Create repository 按钮即可创建代码库，但此时代码库中还是空的，需要在本地计算机中推送一个项目到 GitHub 代码库。下一节会详细介绍这个推送过程。

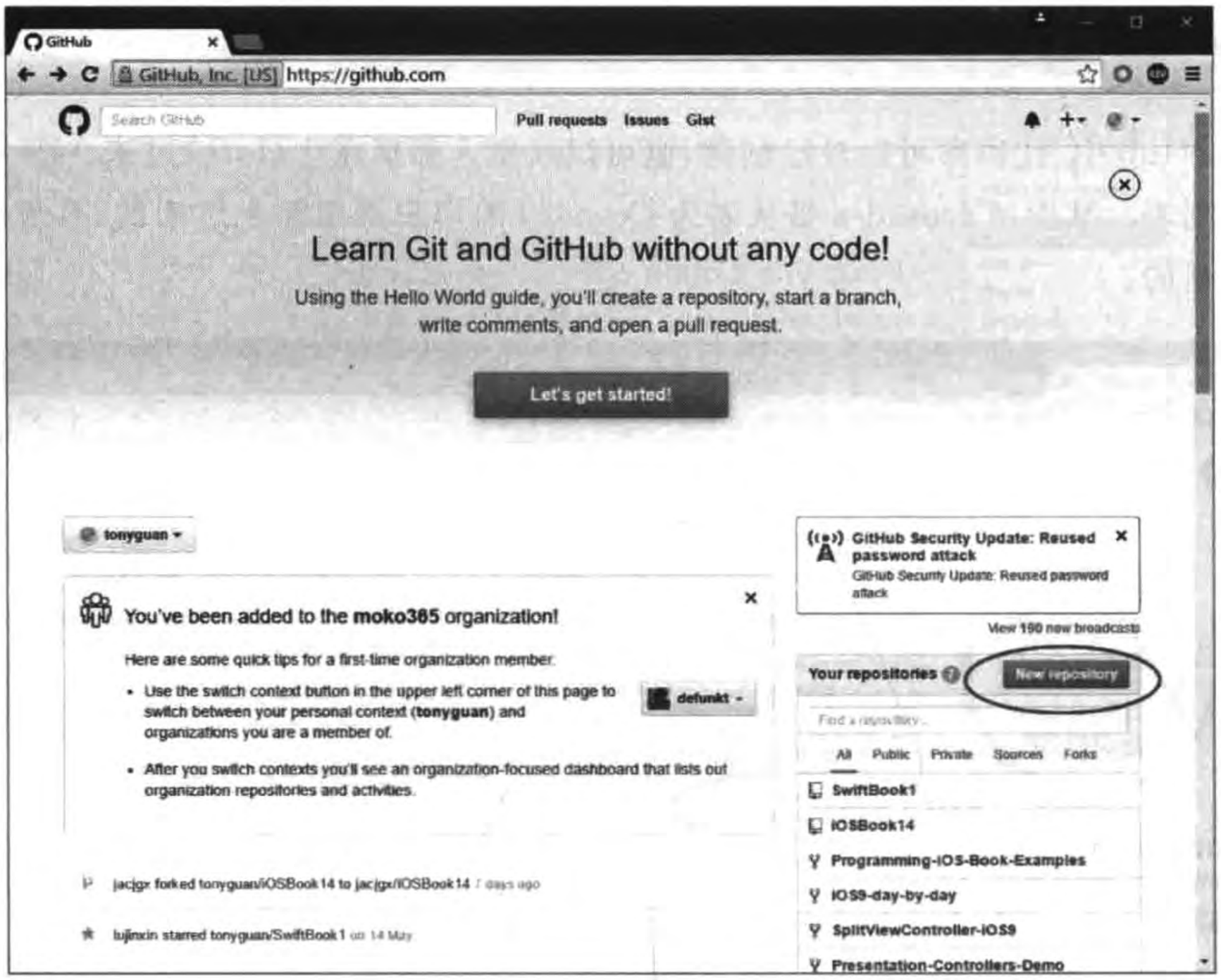


图 22-8　创建代码库

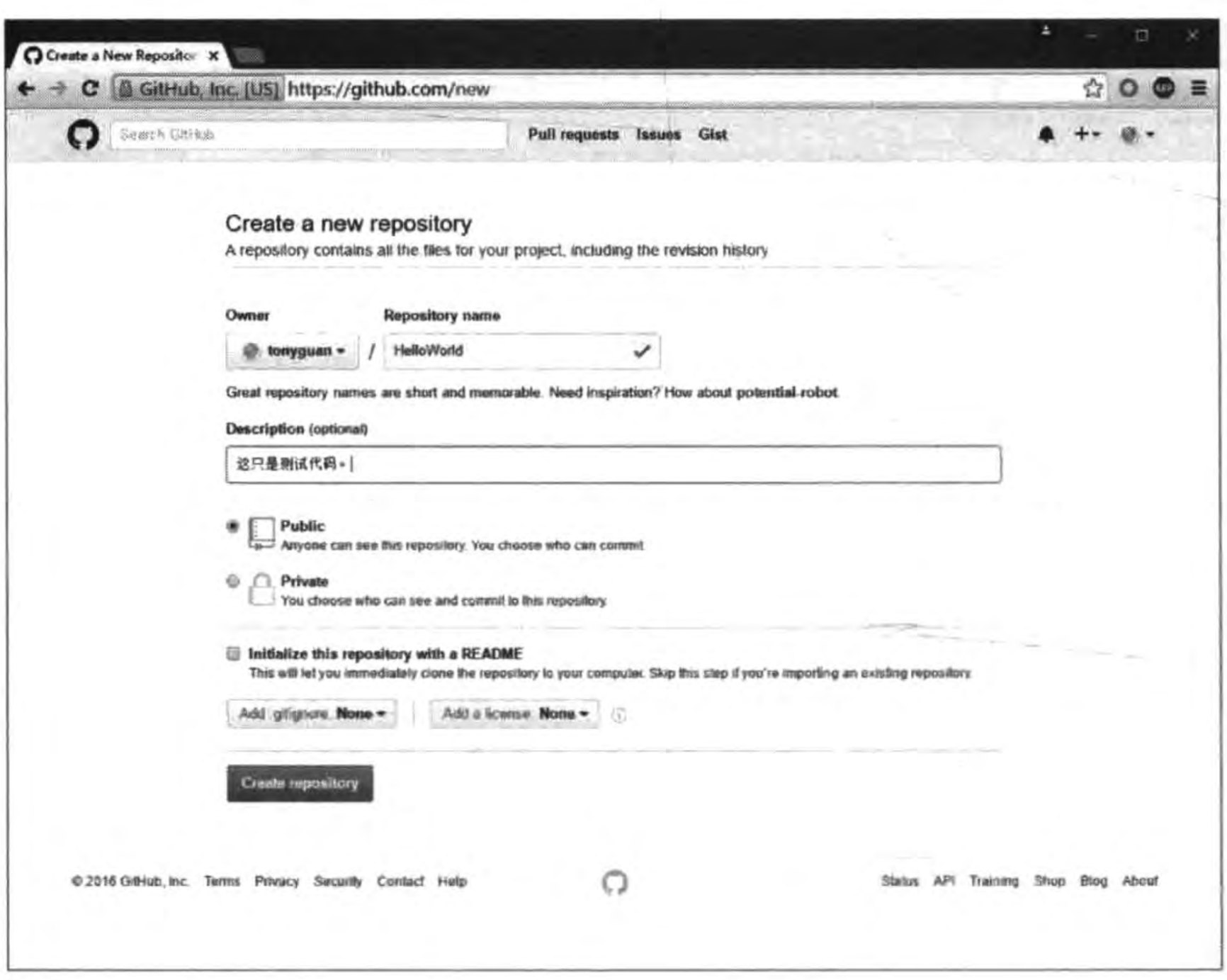


图 22-9　创建代码库页面

22.2.3　删除代码库

有时需要删除代码库，此时可以单击图 22-10 所示的 Setting 标签，进入图 22-11 所示的代码库维护页面。将页面滚动到下面，会看到 Delete this repository 按钮，单击该按钮后再次确认即可完成该操作。这个操作破坏性比较大，操作时一定要谨慎。

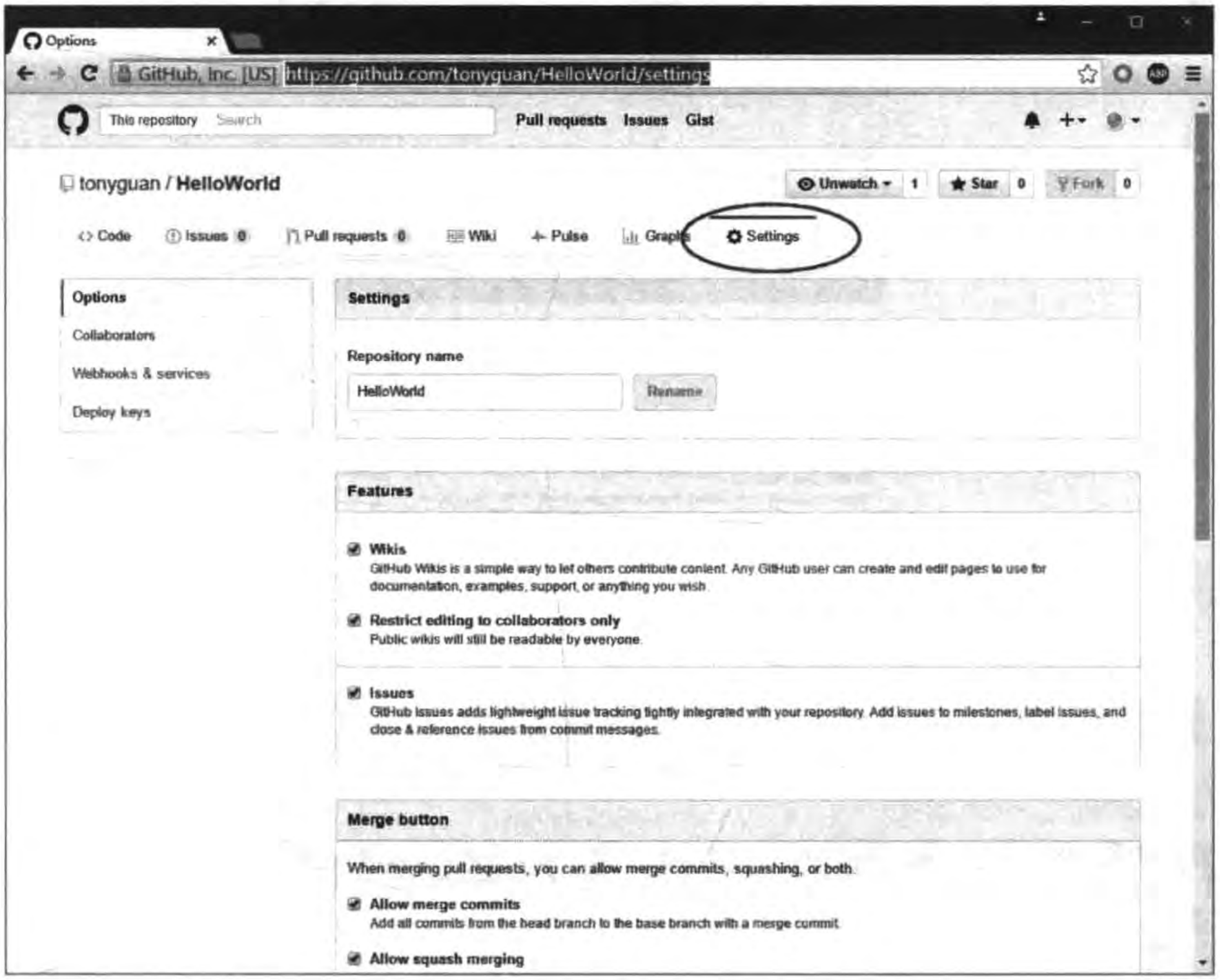


图 22-10　进入 Settings 页面

22.2.4　派生代码库

获得代码库的最简单方式是从别人那里派生代码库。可以修改该代码库，然后提供给开发者。派生与 Git 中的分支很像，可以把它理解为代码库级别的分支。

如果想从 cocos2d 用户那里派生 cocos2d-x 代码库，首先需要在 GitHub 中找到 cocos2d-x 代码库。GitHub 搜索功能在网址 https://github.com/search 的页面中提供了，如图 22-12 所示。

在这个网页的命令栏中输入命令并按 Enter 键即可执行命令。在搜索栏中可以输入关键字 cocos2d-x。单击 Search 按钮即可进行搜索，得到的结果展示在图 22-13 中。

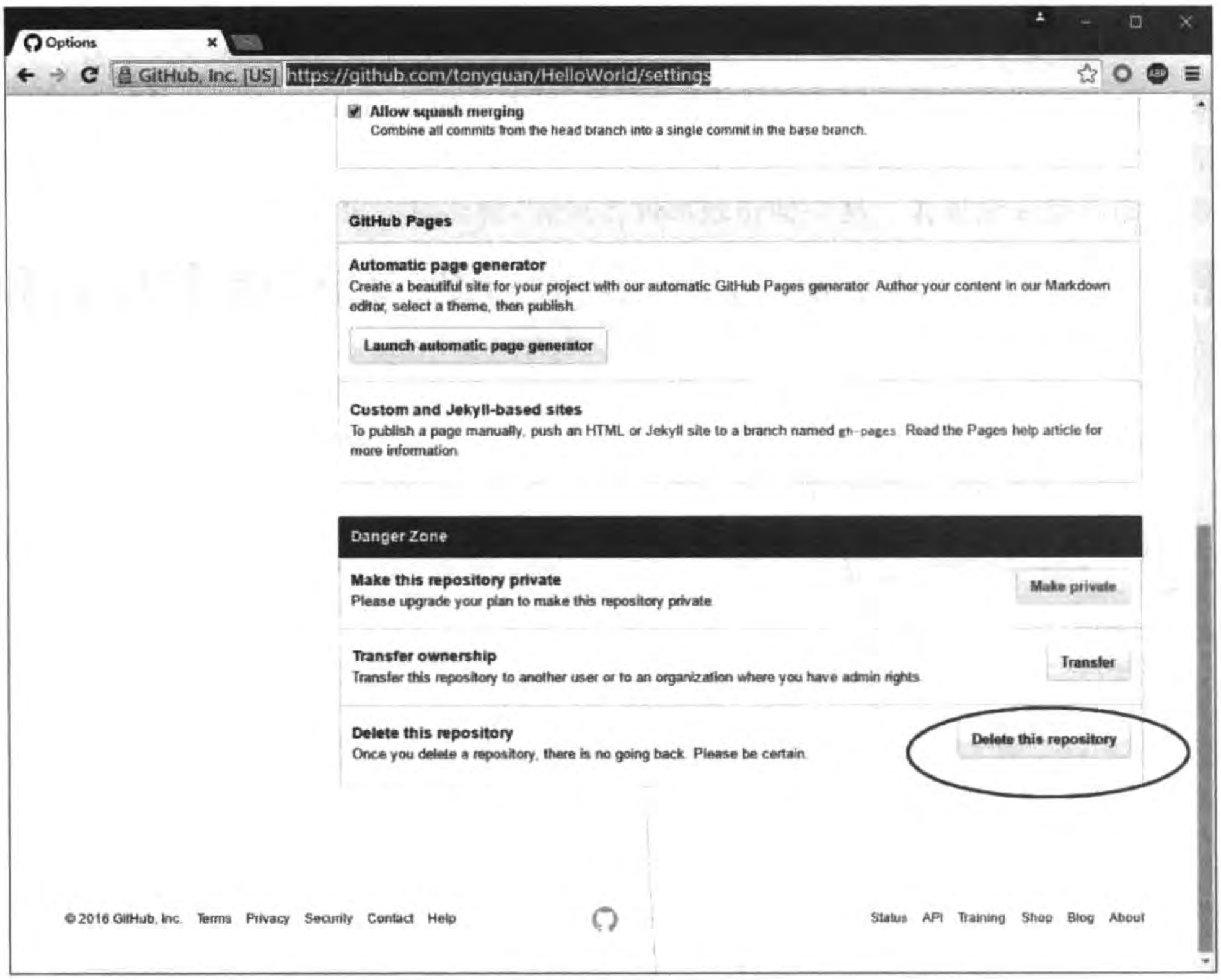


图 22-11　删除代码库

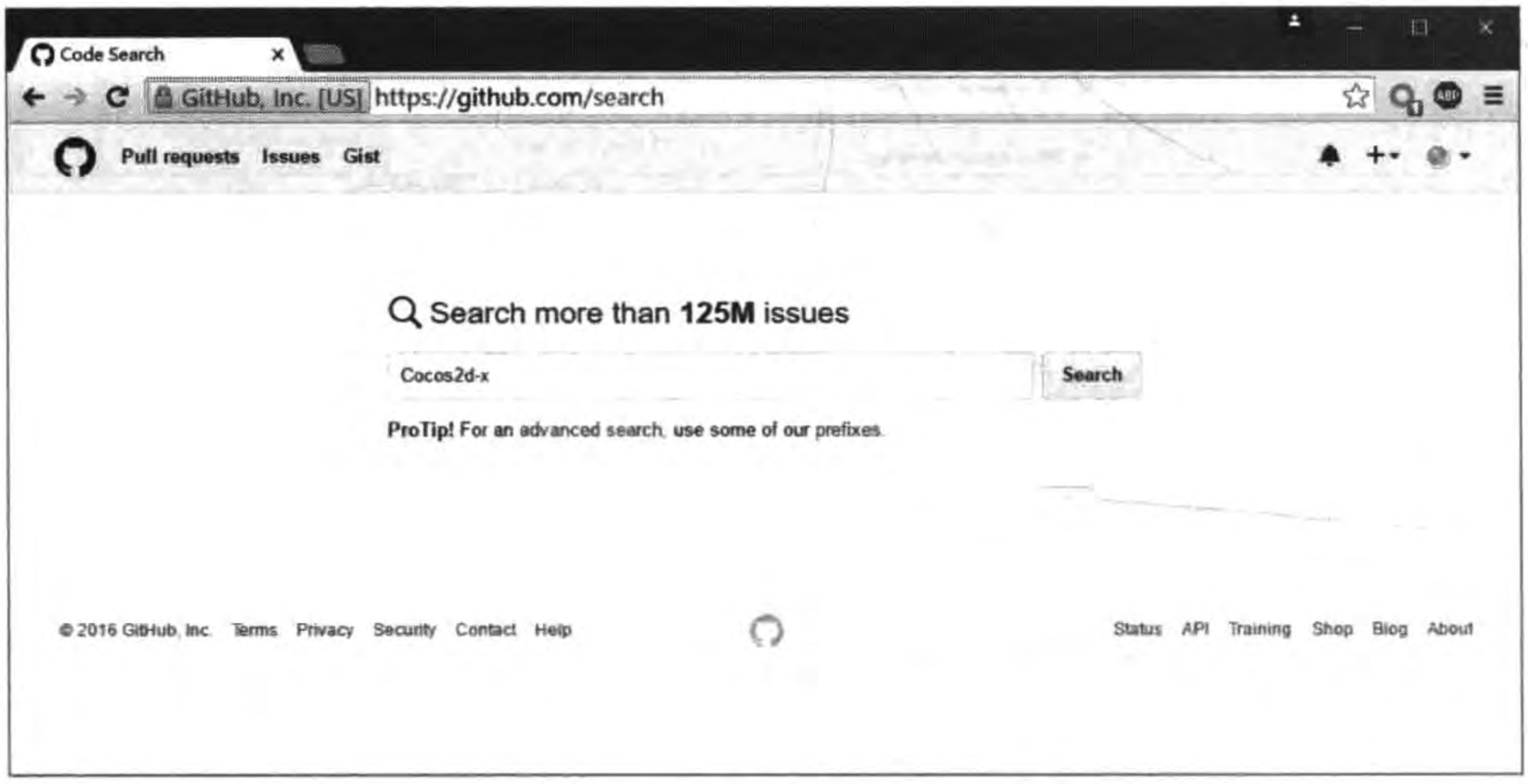


图 22-12　GitHub 中的执行命令和搜索页面

图 22-13　搜索结果

单击 cocos2d/cocos2d-x 超链接，即可进入 cocos2d 账户下的 cocos2d-x 代码库，如图 22-14 所示。

图 22-14　cocos2d 账户的 cocos2d-x 代码库

单击右上角的 Fork 按钮，此时会弹出一个确认对话框。确认之后就可以把 cocos2d-x 代码库派生到当前账户下，如图 22-15 所示。

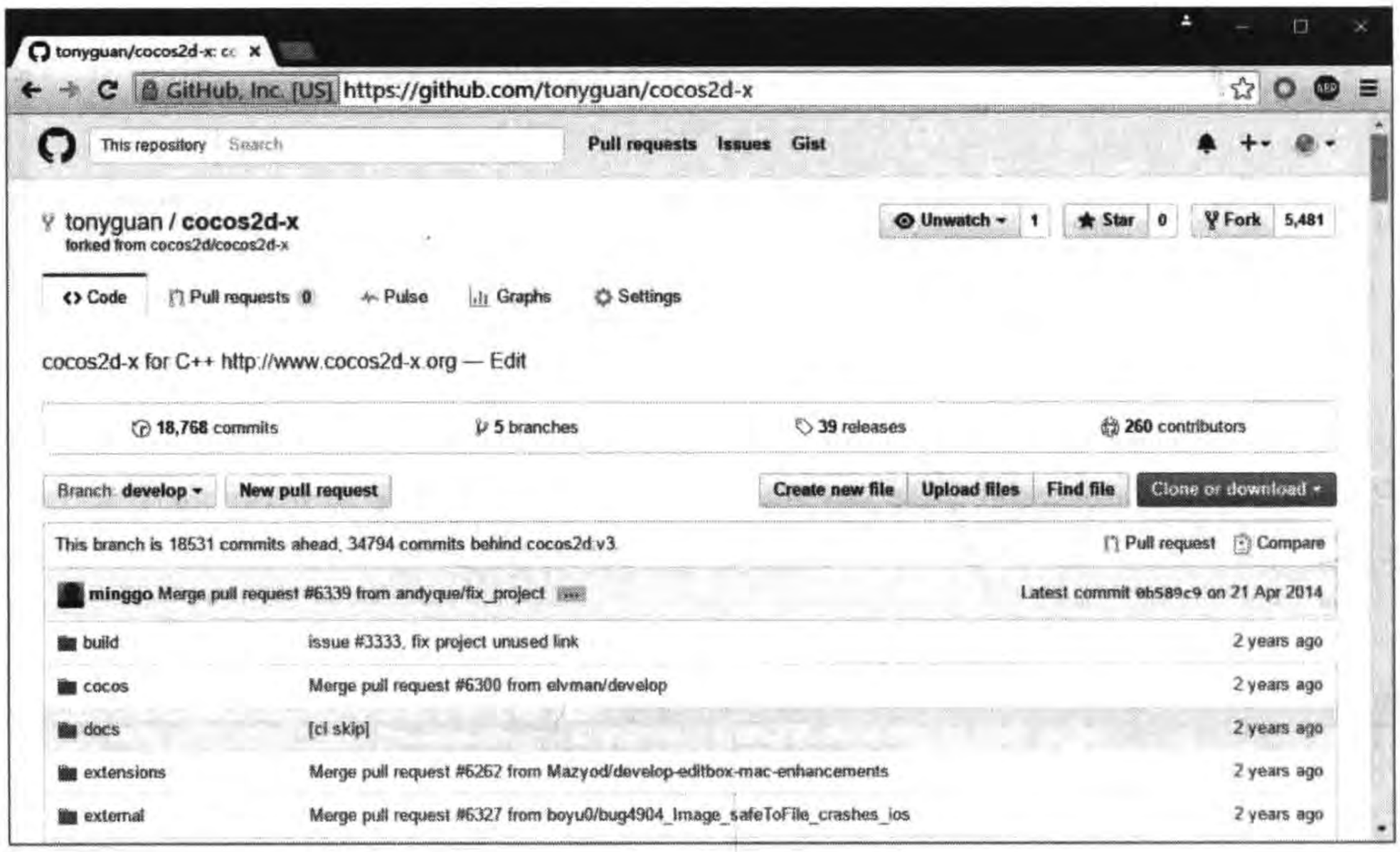


图 22-15　当前账号下的 cocos2d-x 代码库

此时，当前账号可以像使用自己创建的其他代码库一样使用这个库。如果只是想参考别人的代码学习，这样派生过来后工作就结束了。

22.2.5　GitHub 协同开发

使用 GitHub 协同开发有两种模式：一种是比较传统的，其中代码库与开发者之间是一对多的关系模式；另一种是代码库与开发者之间是多对多的关系模式。

一对多的关系模式如图 22-16 所示，这里一个 GitHub 用户作为项目 A 代码库的管理员。这种模式很传统，适合于沟通密切的小团队开发，开发人员基本上不需要登录 GitHub。

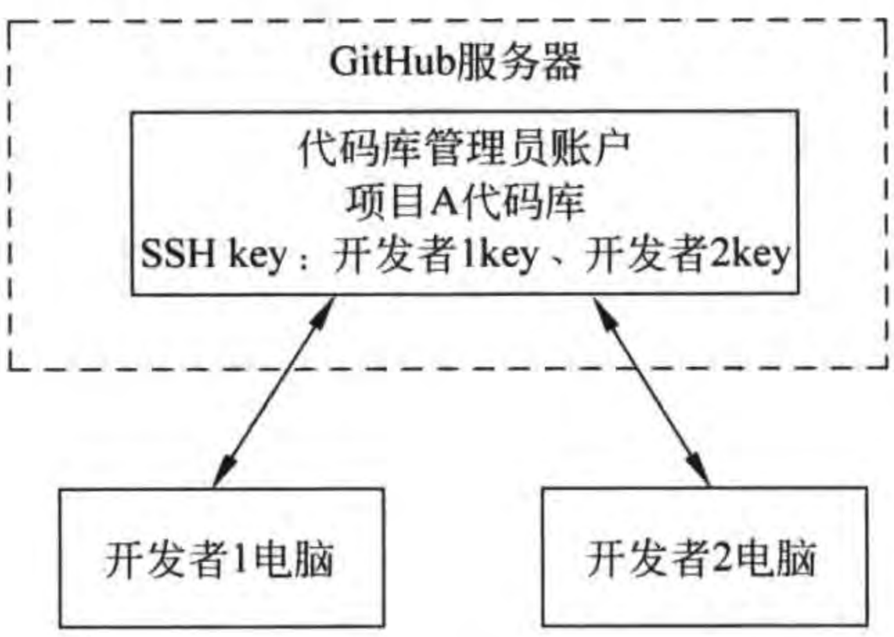


图 22-16　一对多的关系模式

多对多的关系模式如图 22-17 所示。此时在 GitHub 服务器上存在多个 A 项目的代码库，但是只有一个是“主”的，由管理员账户维护。管理员也可能就是一个开发者，其他开发者有自己的账号，维护自己的 A 项目代码库，但是他们的项目代码库是从管理员那里派生过来的，修改完后需要推送回管理员的主代码库中。

这种模式很灵活，适用于复杂项目的大型团队，能够充分利用 GitHub 编程社区网站的功能。

图 22-17　多对多的关系模式

例如，可以实现建立组织、互发邮件和通知等功能。这种模式主要使用 GitHub 的派生功能。

22.3　实例：Cocos2d-x 游戏项目协同开发

如果项目需要一个团队协同开发，就需要搭建 Git 服务器。考虑到成本，一般可以使用 GitHub 作为服务器托管代码。由于 Cocos2d-x 游戏项目一般不会有太多的人员参与，要求成员之间紧密合作，因此 GitHub 一对多模式就能够满足要求。直接把 22.2.2 节创建的 HelloWorld 代码库复制过来使用即可。

现在只关心客户端即可。Xcode、Visual Studio 和 Eclipse 等开发工具都有 Git 客户端功能，但是在 Cocos2d-x 游戏项目中不推荐使用它们，因为每一个工具操作界面差别都很大，推荐直接使用 Git 的协同开发常用命令：git clone、git fetch、git pull、git push 和 git remote add 等。下面通过解决实际工作中的一些问题来介绍 Cocos2d-x 游戏项目协同开发。

问题如下：

(1) 开发者 1 在本地创建 HelloWorld 工程代码库，然后提交到 GitHub 代码库。

(2) 开发者 2 把 GitHub 的 HelloWorld 代码库拷贝到本地代码库，对 HelloWorld 工程进行修改，并重新提交给 HelloWorld 代码库。

(3) 当开发者 2 重新提交给服务器代码库时，开发者 1 如何获得本地代码库？

22.3.1　提交到 GitHub 代码库

开发者 1 在本地创建 HelloWorld 工程代码库，然后提交到 GitHub 代码库。开发者 1 在自己的计算机中创建了 HelloWorld 工程，但是它并不是代码库，需要进行初始化。开发者 1 使用 Cocos new 工具创建了 HelloWorld 的 Cocos2d-x 工程，然后启动 Git Bash，进入到 HelloWorld 目录。执行命令如图 22-18 所示。

图 22-18　初始化本地代码库

上述命令中 cd /d/HelloWorld/是进入到 d 盘下的 HelloWorld 目录。git init 是初始化 HelloWorld 目录为本地代码库。

初始化完成之后开发者 1 就可以将 HelloWorld 本地代码库提交给 GitHub 服务器代码库,但是提交之前要看看 HelloWorld 工程中哪些文件需要提交哪些不需要提交。不应该把 HelloWorld 工程目录下的内容全部提交给服务器,否则每次上传和下载都比较耗时。那些能够在本地重建的文件不用提交,因此,有必要在 HelloWorld 工程目录中建立一个不提交的文件和目录的列表,这个文件是个隐藏文件,命名为.gitignore。.gitignore 部分内容如下:

```
# ignore thumbnails created by windows
Thumbs.db
# Ignore files build by Visual Studio
*.obj
*.exe
...

# Ignore Android SDK, NDK, Eclipse plugin and ANT files
libs/
bin/
obj/
assets/
```

在这个文件中,#号表示注释,可以使用正则表达式。.gitignore 文件非常重要,由于在 Cocos2d-x 中经常在不同平台编译,因此会产生很多编译文件,最多情况下工程目录达到 4GB。如果不加以过滤,那么后果是很严重的。

提示　.gitignore 文件内容可以是本书配套代码中的 gitignore.txt 文件,把其复制到 HelloWorld 工程目录下。但是如果试图在资源管理器下把 gitignore.txt 重新命名为.gitignore,则会发生图 22-19 所示的错误。可以通过 DOS 进入 HelloWorld 工程

目录，通过 ren gitignore. txt . gitignore 命令实现，如图 22-20 所示。其中，ren 是 DOS 的重命名指令。

图 22-19　重命名发生错误

图 22-20　通过 DOS 重命名

这些工作准备好之后，就可以向 GitHub 服务器提交本地代码库。启动 Git Bash，进入到 HelloWorld 目录，执行如下指令：

```
$ git add .
$ git commit -m "tony commit"
$ git remote add origin git@github.com:tonyguan/HelloWorld.git
$ git push -u origin master
```

在执行过程中有时会出现下面的错误：

```
$ git push -u origin master
To git@github.com:tonyguan/HelloWorld.git
! [rejected]          master -> master (fetch first)
error: failed to push some refs to 'git@github.com:tonyguan/HelloWorld.git'
```

这是因为 GitHub 代码库中已经有内容了,这种情况下首先使用如下命令:

```
$ git fetch origin
```

从 GitHub 服务器端取数据,然后运行上面的指令:

```
$ git add .
$ git commit -m "tony commit"
$ git push -u origin master
```

若命令成功执行,则可以在 GitHub 上看到刚刚提交的代码库。

22.3.2 克隆 GitHub 代码库

作为开发者 2,则需要把 GitHub 的 HelloWorld 代码库克隆到本地代码库,对 HelloWorld 工程进行修改,并重新提交给 HelloWorld 代码库。

开发者 2 的计算机上没有 HelloWorld 代码库,他不需要自己创建,而是从 GitHub 上拷贝过来。此时可以在终端中执行如下命令:

```
$ git clone 'git@github.com:tonyguan/HelloWorld.git'
```

然后他也对 HelloWorld 做了一些修改。那么如何推送他的数据到服务器代码库呢? 实际上,这个过程与开发者 1 刚刚的提交方式是一样的。这里就不再介绍了。

22.3.3 重新获得 GitHub 代码库

开发者 1 再次被告知他们的程序有新的版本,他需要从服务器代码库中获取新的程序。使用 git fetch 命令,可以从服务器代码库获取数据。相关命令如下:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 4 (delta 3), reused 4 (delta 3)
Unpacking objects: 100% (4/4), done.
From github.com:tonyguan/HelloWorld
   144841e..6c3b7b0  master     -> origin/master
```

这时打开修改的文件,发现没有变化,这是因为还需要使用 git merge 命令合并 origin/master 到本地 master 分支。相关代码如下:

```
$ git merge origin/master
Updating 144841e..6c3b7b0
Fast-forward
 Classes/HelloWorldScene.cpp | 1 +
 1 file changed, 1 insertion(+)
```

再看看修改的文件是否发生了变化。合并过程中也可能发生冲突,需要人为解决这些冲突再合并,这可以通过 Git 提供的更加简便的命令 git pull 实现。git pull 命令是 git fetch 和 git merge 命令的一个组合。相关代码如下:

```
$ git pull origin master
```

```
remote: Counting objects: 7, done.
remote: Compressing objects: 100 % (1/1), done.
remote: Total 4 (delta 3), reused 4 (delta 3)
Unpacking objects: 100 % (4/4), done.
From github.com:tonyguan/HelloWorld
 * branch            master     -> FETCH_HEAD
   6c3b7b0..76b1c8d  master     -> origin/master
Updating 6c3b7b0..76b1c8d
Fast - forward
 Classes/HelloWorldScene.cpp | 2 +-
 1 file changed, 1 insertion( + ), 1 deletion( - )
```

其中，origin 是远程代码库名，master 是要合并的本地分支。

本章小结

通过对本章的学习，读者可以了解如何使用 Git 进行代码版本控制，具体内容包括 Git 服务器的搭建、Git 常用命令、GitHub 协同开发、使用 GitHub 代码托管服务等。

第 23 章

Cocos2d-x Lua API 敏捷开发项目实战

——迷失航线手机游戏

这一章是项目实战，也是本书的画龙点睛之笔。我想通过一个实际的手机游戏，使用 Cocos2d-x Lua API 从设计到开发整个过程，使读者能够将本书前面讲过的知识点串联起来。了解当下最为流行的开发方法学——敏捷开发。在开发过程中，我们会发现敏捷方法非常适合基于 Cocos2d-x Lua API 开发手机游戏。

23.1 迷失航线游戏分析与设计

本节从计划开发这个项目开始，然后进行分析和设计，设计过程包括原型设计、场景设计、脚本设计和架构设计。

23.1.1 迷失航线故事背景

这款游戏构思的初衷，是出于大英博物馆里珍藏的一份“二战”时期的飞行报告。讲述了一名英国皇家空军的飞行员在执行任务时遭遇了暴风雨，迫降在不列颠的一个不知名的军用机场，看到绿色的跑道和米色的塔楼。等暴风雨平息后再次起飞又遭遇相同的困境，返航后提交飞行报告时却被告知根本不存在这个军用机场。迷惑的飞行员百思不得其解，直到 70 年代的一次飞行中降落在一个刚刚竣工的军用机场，目睹的一切与当年看到的情景完全一样（刚刚粉刷好的绿色的跑道和米色的塔楼）。于是，这份飞行记录成为关键证据，造就了世界知名的未解之谜。

因此，我们想以这个超现实主义的故事为背景，设计并制作一款简单、轻松的射击类游戏，让我们的小飞机穿越时空去冒险。

23.1.2 需求分析

这是一款非过关类的第三视角射击游戏。

游戏主角是一架“二战”时期的老式轰炸机，在迷失航线后穿越宇宙、穿越时空，与敌人激战的同时还要躲避虚拟时空的生物和小行星。

由于是一款手机游戏，因此需要设计的操作简单，节奏明快，适合用户利用空闲或琐碎的时间来放松和娱乐。

在这里我们采用用例分析方法描述用例图，如图 23-1 所示。

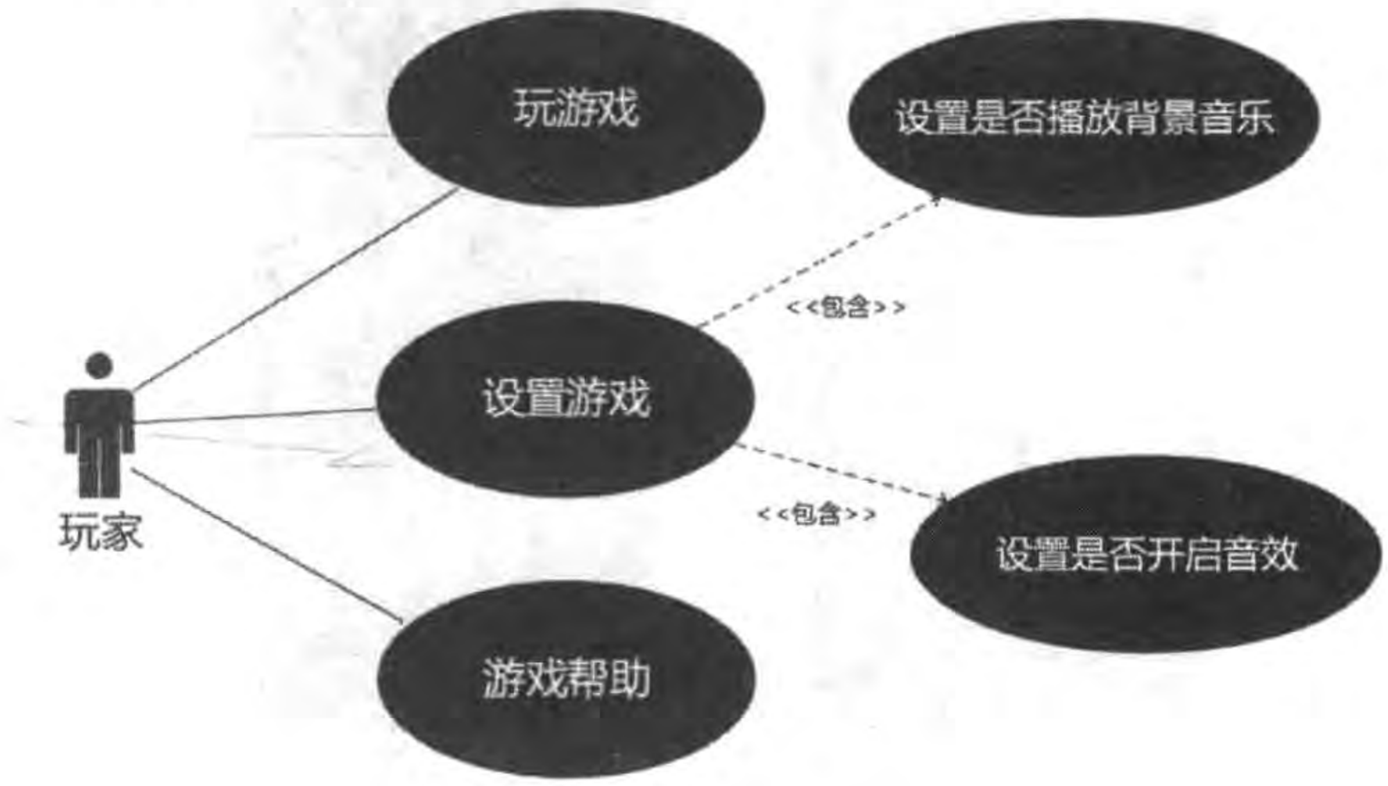


图 23-1　客户端用例图

23.1.3　原型设计

原型设计草图对于应用设计人员、开发人员、测试人员、UI 设计人员以及用户都是非常重要的，该案例的原型如图 23-2 所示。

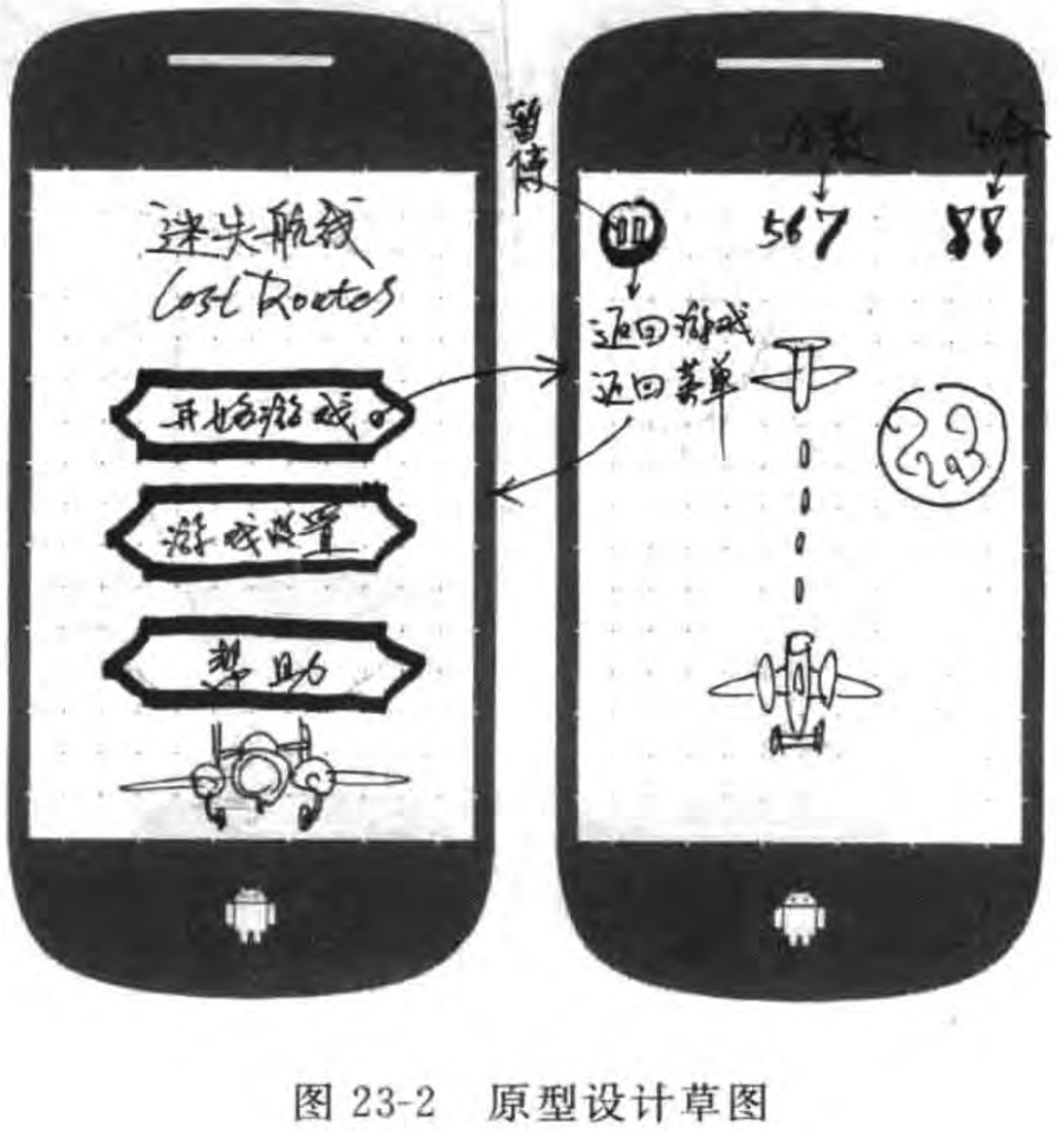


图 23-2　原型设计草图

图 23-2 所示是个草图，它是我们最初的想法，一旦确定了这些想法，我们的 UI 设计师就会将这些草图变成高保真原型设计图（见图 23-3）。

最终我们希望采用另类的圆珠笔手绘风格界面，并把战斗和冒险的场景安排在坐标纸上。这样会给玩家带来耳目一新、超乎想象的个性体验。

图 23-3　高保真原型设计图

23.1.4　游戏脚本

为了在游戏的实现过程中团队的配合更加默契，工作更加有效，我们事先制作了一个简单的手绘游戏脚本(见图 23-4)。

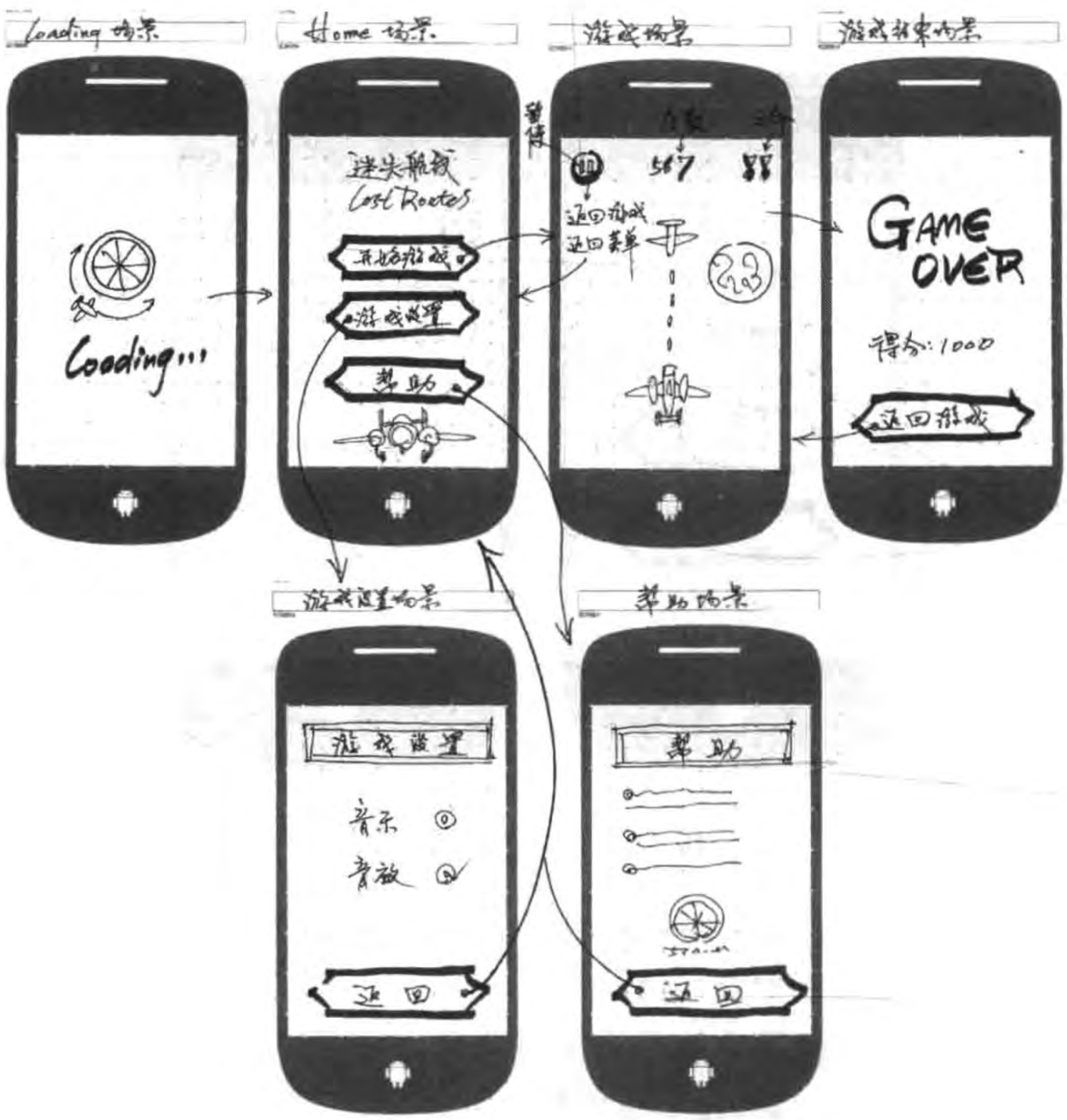


图 23-4　游戏脚本图

脚本描绘了界面的操作及交互流程和游戏的场景，包括场景中的敌人种类、玩家飞机位置、它们的生命值、击毁一个敌人获得的加分情况，同时每次加分超过 1000 分给玩家增加一条生命等。

23.2　任务 1：游戏工程的创建与初始化

在开发项目之前，应该由一个人搭建开发环境，然后把环境复制给其他人使用。

23.2.1　迭代 1.1：创建工程

首先，使用 cocos new 命令创建本地工程。如果使用 Windows 操作系统，则进入 DOS 终端，在 DOS 终端中进入 bin 目录。执行如下指令：

```
cocos new -p com.work6 -l Lua -d <保存工程的目录> LostRoutes
```

具体步骤可参考 4.1.1 节。

23.2.2　迭代 1.2：添加资源文件

我们需要为 LostRoutes 游戏工程准备资源文件，目录结构如下：

```
LostRoutes/res
├──fonts
|       hanyi.ttf
|
├──large
|   └──map
|           blue_bg.tmx
|           blue_tiles.png
|           red_bg.tmx
|           red_tiles.png
|
├──small
|   |
|   └──map
|           blue_bg.tmx
|           blue_tiles.png
|           red_bg.tmx
|           red_tiles.png
|
├──particle
|       explosion.plist
|       fire.plist
|       light.plist
|
├──sound
|       Blip.caf
|       Blip.wav
|       Explosion.caf
|       Explosion.wav
```

```
|       game_bg.aifc
|       game_bg.mp3
|       home_bg.aifc
|       home_bg.mp3
|
└──texture
        loading_texture.plist
        loading_texture.png
        loading_texture.pvr.gz
        loading_texture_pvr.plist
        LostRoutes_Texture.plist
        LostRoutes_Texture.png
        LostRoutes_Texture.pvr.gz
        LostRoutes_Texture.tps
        LostRoutes_Texture_pvr.plist
```

fonts 目录是保存 TTF 字体文件。small/map 目录保存小地图资源文件，large/map 目录保存大地图资源文件。texture 目录是保存纹理图集相关文件。particle 目录是保存粒子系统的文件。sound 目录是保存背景音乐和音效的文件。

这些资源文件的说明如表 23-1 所述。

表 23-1　资源文件说明

运　算　符	目录	说　　明
hanyi. ttf	fonts	TTF 字体文件
blue_bg. tmx	small/map	蓝色背景小地图文件
blue_tiles. png	small/map	蓝色背景小地图中的瓦片
red_bg. tmx	small/map	红色背景小地图文件
red_tiles. png	small/map	红色背景小地图中的瓦片
blue_bg. tmx	large/map	蓝色背景大地图文件
blue_tiles. png	large/map	蓝色背景大地图中的瓦片
red_bg. tmx	large/map	红色背景大地图文件
red_tiles. png	large/map	红色背景大地图中的瓦片
loading_texture. plist	texture	Loading 场景纹理图集 plist 文件(非 iOS 平台)
loading_texture. png	texture	Loading 场景图集文件(非 iOS 平台)
loading_texture. pvr. gz	texture	Loading 场景纹理图集文件(iOS 平台)
loading_texture_pvr. plist	texture	Loading 场景纹理图集 plist 文件(iOS 平台)
LostRoutes_Texture. plist	texture	除 Loading 场景外，其他场景纹理图集 plist 文件(非 iOS 平台)
LostRoutes_Texture. png	texture	除 Loading 场景外，其他场景纹理图集文件(非 iOS 平台)
LostRoutes_Texture. pvr. gz	texture	除 Loading 场景外，其他场景纹理图集文件(iOS 平台)
LostRoutes_Texture_pvr. plist	texture	除 Loading 场景外，其他场景纹理图集 plist 文件(iOS 平台)
explosion. plist	particle	爆炸粒子系统 plist 文件
fire. plist	particle	飞机尾巴喷射火焰粒子系统 plist 文件
light. plist	particle	游戏场景背景中发亮粒子系统 plist 文件
Blip. caf	sound	单击按钮声音效果(iOS 平台)

续表

运　算　符	目录	说　　明
Blip. wav	sound	单击按钮声音效果(非 iOS 平台)
Explosion. caf	sound	爆炸声音效果(iOS 平台)
Explosion. wav	sound	爆炸声音效果(非 iOS 平台)
game_bg. aifc	sound	游戏场景背景声音(iOS 平台)
game_bg. mp3	sound	游戏场景背景声音(非 iOS 平台)
home_bg. aifc	sound	除游戏场景之外,其他场景的背景声音(iOS 平台)
home_bg. mp3	sound	除游戏场景之外,其他场景的背景声音(非 iOS 平台)

23.2.3　迭代 1.3：添加常量文件 SystemConst. lua

在项目中为了方便管理常量,可以在一个 Lua 文件中定义这些创建,在本例中添加 SystemConst. lua 文件,SystemConst. lua 代码如下：

```
local targetPlatform = cc.Application:getInstance():getTargetPlatform()                    ①

--UserDefault 中保存音效播放状态键
SOUND_KEY = "sound_key"
--UserDefault 中保存背景音效播放状态键
MUSIC_KEY = "music_key"
--UserDefault 中保存最高记录键
HIGHSCORE_KEY = "highscore_key"

--本地 iOS 平台
if (cc.PLATFORM_OS_IPHONE == targetPlatform)
        or (cc.PLATFORM_OS_IPAD == targetPlatform) then                                    ②
    --声音
    bg_music_1   =    "sound/home_bg.aifc"                                                 ③
    bg_music_2   =   "sound/game_bg.aifc"
    sound_1    =   "sound/Blip.caf"
    sound_2    =   "sound/Explosion.caf"
    effectExplosion = "sound/Explosion.caf"

    --LostRoutes  Texture 资源
    texture_res = 'texture/LostRoutes_Texture.pvr.gz'
    --LostRoutes Texture plist
    texture_plist = 'texture/LostRoutes_Texture_pvr.plist'
    --loading  Texture 资源
    loading_texture_res = 'texture/loading_texture.pvr.gz'
    --loading Texture plist
    loading_texture_plist = 'texture/loading_texture_pvr.plist'                            ④

    --其他平台包括 Web 和 Android 等
else
    --声音
    bg_music_1    =   "sound/home_bg.mp3"                                                  ⑤
    bg_music_2   =   "sound/game_bg.mp3"
    sound_1   =    "sound/Blip.wav"
```

```
sound_2    =   "sound/Explosion.wav"
effectExplosion = "sound/Explosion.wav"
-- LostRoutes  Texture 资源
texture_res = 'texture/LostRoutes_Texture.png'
-- LostRoutes Texture plist
texture_plist = 'texture/LostRoutes_Texture.plist'
-- loading  Texture 资源
loading_texture_res = 'texture/loading_texture.png'
-- loading Texture plist
loading_texture_plist = 'texture/loading_texture.plist'                    ⑥
end

-- Home 菜单操作标识
HomeMenuActionTypes = {
    MenuItemStart = 100,
    MenuItemSetting = 101,
    MenuItemHelp =  102
}

--定义敌人类型
EnemyTypes = {
    Enemy_Stone =   0, -- 陨石
    Enemy_1 =   1, -- 敌机 1
    Enemy_2 =   2, -- 敌机 2
    Enemy_Planet =   3 -- 行星
}

--定义敌人名称  也是敌人精灵帧的名字
EnemyName = {
    Enemy_Stone =   "gameplay.stone1.png",
    Enemy_1 =   "gameplay.enemy-1.png",
    Enemy_2 =   "gameplay.enemy-2.png",
    Enemy_Planet =   "gameplay.enemy.planet.png"
}

--游戏场景中使用的标签常量
GameSceneNodeTag = {
    StatusBarFighterNode =   301,
    StatusBarLifeNode =   302,
    StatusBarScore =   303,
    BatchBackground =   800,
    Fighter =   900,
    ExplosionParticleSystem =   802,
    Bullet =   803,
    Enemy =   804
}

--精灵速度常量
Sprite_Velocity = {
    Enemy_Stone =   cc.p(0, -150),
    Enemy_1 =   cc.p(0, -80),
    Enemy_2 =   cc.p(0, -100),
    Enemy_Planet =   cc.p(0, -50),
```

```
    Bullet =   cc.p(0, 200)
}

--分值
EnemyScores = {
    Enemy_Stone = 5,
    Enemy_1 = 10,
    Enemy_2 = 15,
    Enemy_Planet = 20
}

--敌人初始生命值
Enemy_initialHitPoints = {
    Enemy_Stone = 3,
    Enemy_1 = 5,
    Enemy_2 = 15,
    Enemy_Planet = 20
}

--我方飞机生命数
Fighter_hitPoints = 5
```

上述第①行代码的表达式可以获得当前平台，第②行代码是判断 iOS 设备，即 iPhone、iPod touch 和 iPad，其中第③和第④行代码是 iOS 设备下所需要文件，而第⑤和第⑥行代码是非 iOS 设备下所需要文件。

23.2.4　迭代 1.4：多分辨率支持

我们的游戏需要发布到多个不同平台，需要考虑多分辨率支持。相关技术内容可以参考 21.4 节。我们可以在资源目录 res 下，创建不同规格资源文件所需子目录。本次我们的美工只是设计了一套规格为 640×960 资源文件，这种文件放到 texture 子目录下。而且地图准备了两套，对应目录 small/map 和 large/map。

多分辨率是在 main.lua 编写的，main.lua 代码如下：

```
require "cocos.init"

--设计分辨率大小
local designResolutionSize = cc.size(320, 568)

--三种资源大小
local smallResolutionSize = cc.size(640, 1136)
local largeResolutionSize = cc.size(750, 1334)

-- cclog
cclog = function(...)
    print(string.format(...))
end

local function main()
    collectgarbage("collect")
```

```
-- avoid memory leak
collectgarbage("setpause", 100)
collectgarbage("setstepmul", 5000)

-----------------
local director = cc.Director:getInstance()
local glview = director:getOpenGLView()

local sharedFileUtils = cc.FileUtils:getInstance()
sharedFileUtils:addSearchPath("src")
sharedFileUtils:addSearchPath("res")

    local searchPaths = sharedFileUtils:getSearchPaths()
    local resPrefix = "res/"

    --屏幕大小
    local frameSize = glview:getFrameSize()

    -- 如果屏幕分辨率高度大于 small 尺寸的资源分辨率高度,选择 large 资源。
    if frameSize.height > smallResolutionSize.height then
        director:setContentScaleFactor(math.min(largeResolutionSize.height /
designResolutionSize.height, largeResolutionSize.width / designResolutionSize.width))
        table.insert(searchPaths, 1, resPrefix .. "large")
        --如果屏幕分辨率高度小等于 small 尺寸的资源分辨率高度,选择 small 资源。
    else
        director:setContentScaleFactor(math.min(smallResolutionSize.height /
designResolutionSize.height, smallResolutionSize.width / designResolutionSize.width))
        table.insert(searchPaths, 1, resPrefix .. "small")
    end
    --设置资源搜索路径
    sharedFileUtils:setSearchPaths(searchPaths)

    -- 设置设计分辨率策略
     glview: setDesignResolutionSize ( designResolutionSize. width, designResolutionSize.
height, cc.ResolutionPolicy.FIXED_WIDTH)

    --设置是否显示帧率和精灵个数
    director:setDisplayStats(true)

    --设置帧率
    director:setAnimationInterval(1.0 / 60)

    --创建场景
    local scene = require("LoadingScene")
    local loadingScene = scene.create()

    if director:getRunningScene() then
        director:replaceScene(loadingScene)
    else
        director:runWithScene(loadingScene)
    end
end
```

上述代码在前面章节已经介绍过了,这里不再赘述。

23.2.5　迭代 1.5：发布到 GitHub

LostRoutes 工程创建完成并编译通过后，需要将工程代码分发给开发小组的其他成员，这里使用 GitHub 发布应用的第一个版本。注意，其他成员无须再创建 LostRoutes 工程，只需从 GitHub 上克隆 LostRoutes 工程代码即可。

需要使用 GitHub 账号登录，并创建一个名为 LostRoutes 的代码库。创建完成后，需要将本地代码上传到 GitHub 服务器，具体步骤可参考 22.2 节。其他的成员使用 $ git clone git@github.com:tonyguan/LostRoutes 克隆到本地。

23.3　任务 2：创建 Loading 场景

Loading 场景是玩家看到游戏的第一个界面，我们在这里加载纹理缓存和预处理声音资源。

23.3.1　迭代 2.1：添加场景和层

先来介绍一下添加 Loading 场景和层。Loading 场景的界面如图 23-5 所示。其中的 Loading 文字是动画的，这样可以消除玩家的心理等待时间。

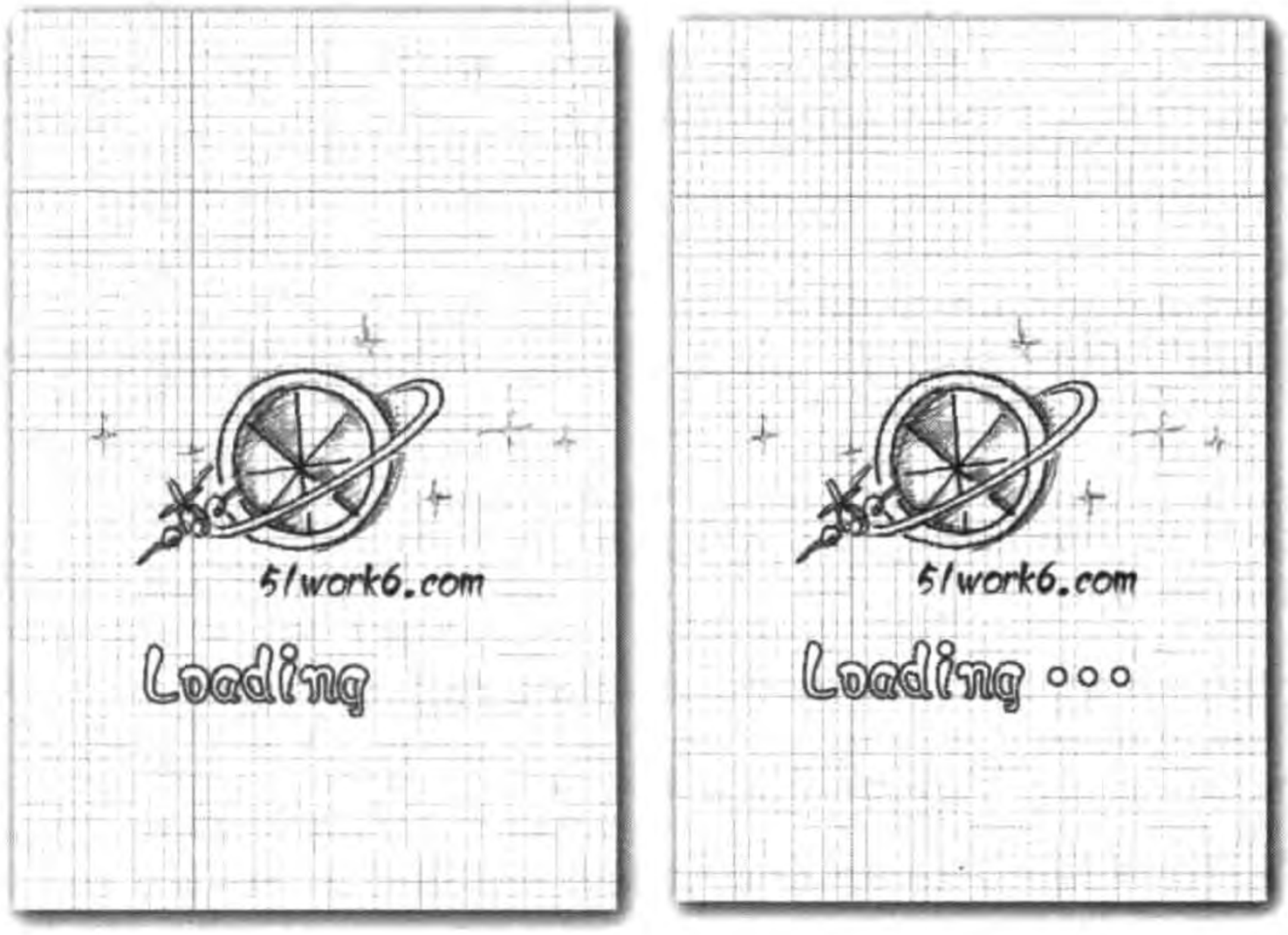


图 23-5　Loading 场景

Loading 场景是在 LoadingScene.lua 文件中实现的，LoadingScene.lua 主要代码如下：

```
require "SystemConst"

local size = cc.Director:getInstance():getWinSize()
```

```
local frameCache = cc.SpriteFrameCache:getInstance()
local textureCache = cc.Director:getInstance():getTextureCache()

local LoadingScene = class("LoadingScene",function()
    return cc.Scene:create()
end)

function LoadingScene.create()
    local scene = LoadingScene.new()
    scene:addChild(scene:createLayer())
    return scene
end

function LoadingScene:ctor()
    --场景生命周期事件处理
    local function onNodeEvent(event)
        if event == "enter" then
            self:onEnter()
        elseif event == "enterTransitionFinish" then
            self:onEnterTransitionFinish()
        elseif event == "exit" then
            self:onExit()
        elseif event == "exitTransitionStart" then
            self:onExitTransitionStart()
        elseif event == "cleanup" then
            self:cleanup()
        end
    end
    self:registerScriptHandler(onNodeEvent)
end

-- 创建层
function LoadingScene:createLayer()
    cclog("LoadingScene init")
    local layer = cc.Layer:create()

    frameCache:addSpriteFrames(loading_texture_plist)                    ①
    local bg = cc.TMXTiledMap:create("map/red_bg.tmx")
    layer:addChild(bg)

    local logo = cc.Sprite:createWithSpriteFrameName("logo.png")
    layer:addChild(logo)
    logo:setPosition(cc.p(size.width/2, size.height/2))

    local sprite = cc.Sprite:createWithSpriteFrameName("loding4.png")
    layer:addChild(sprite)
    local logoX,logoY = logo:getPosition()
    sprite:setPosition(cc.p(logoX, logoY - 130))

    ...
    return layer
end

function LoadingScene:onEnter()
```

```
    cclog("LoadingScene onEnter")
end

function LoadingScene:onEnterTransitionFinish()
    cclog("LoadingScene onEnterTransitionFinish")
end

function LoadingScene:onExit()
    cclog("LoadingScene onExit")
end

function LoadingScene:onExitTransitionStart()
    cclog("LoadingScene onExitTransitionStart")
end

function LoadingScene:cleanup()
    cclog("LoadingScene cleanup")
end

return LoadingScene
```

上述代码是 Loading 场景主要结构，其中定义了一个层，并注册场景事件。而且在第①行代码加载 Loading 场景精灵帧缓存。

23.3.2　迭代 2.2：Loading 动画

Loading 动画是采用帧动画实现，我们的美工帮助设计了 4 帧的 Loading 动画。Loading 动画是在 LoadingScene:createLayer()函数中实现的，createLayer 函数主要代码如下：

```
-- 创建层
function LoadingScene:createLayer()
    cclog("LoadingScene init")
    local layer = cc.Layer:create()

    ...

    ------------------- 动画开始 ----------------------
    local animation = cc.Animation:create()
    for i = 1,4 do
        local frameName = string.format("loding%d.png",i)
        cclog("frameName = %s", frameName)
        local spriteFrame = frameCache:getSpriteFrameByName(frameName)
        animation:addSpriteFrame(spriteFrame)
    end

    animation:setDelayPerUnit(0.5)                    -- 设置两个帧播放时间
    animation:setRestoreOriginalFrame(true)           -- 动画执行后还原初始状态

    local action = cc.Animate:create(animation)
    sprite:runAction(cc.RepeatForever:create(action))
    -------------------- 动画结束 -------------------
```

```
    ...
    return layer
end
```

这些代码也不再解释。

23.3.3 迭代 2.3：异步加载纹理缓存

为了给玩家流畅的体验，一般需要在 Loading 场景加载尽可能多资源，而且这些资源的加载应该是异步的。纹理缓存异步加载可以使用 TextureCache 的 addImageAsync 函数。

LoadingScene.lua 中的相关代码如下：

```
-- 创建层
function LoadingScene:createLayer()
    cclog("LoadingScene init")
    local layer = cc.Layer:create()

    ...
    local function loadingTextureCallBack(texture)                    ①

        frameCache:addSpriteFrames(texture_plist)                     ②
        cclog("loading textrue ok.")

        --初始化  音乐
        AudioEngine.preloadMusic(bg_music_1)                          ③
        AudioEngine.preloadMusic(bg_music_2)                          ④
        --初始化  音效
        AudioEngine.preloadEffect(sound_1)                            ⑤
        AudioEngine.preloadEffect(sound_2)                            ⑥

        local HomeScene = require("HomeScene")
        local scene = HomeScene.create()
        cc.Director:getInstance():pushScene(scene)                    ⑦

    end

    textureCache:addImageAsync(texture_res,loadingTextureCallBack)    ⑧

    return layer
end
```

上述第①行代码定义的 loadingTextureCallBack 函数，它是在第⑧行代码回调的。第②行代码 frameCache:addSpriteFrames(texture_plist)是加载精灵帧缓存。在工程中纹理图集有两个 loading_texture_plist 和 texture_plist，而只有 texture_plist 进行了异步加载，这是因为 loading_texture_plist 只是在当前场景使用一次，texture_plist 纹理会在后面其他场景反复使用，而且 texture_plist 纹理文件比较大，异步加载是有必要的。

第③和第④行代码是预处理声音。第⑤和第⑥行代码是预处理音效。第⑦行代码是跳转到 HomeScene 场景。

第⑧行代码 textureCache:addImageAsync(texture_res,loadingTextureCallBack)是异步加载纹理缓存,异步加载完成后回调 loadingTextureCallBack 函数。

23.4　任务 3：创建 Home 场景

Home 场景是主菜单界面,通过它可以进入到游戏场景、设置场景和帮助场景。

23.4.1　迭代 3.1：添加场景和层

首先需要通过文本编辑工具创建 Home 场景类文件 HomeScene.lua。HomeScene.lua 中主要代码如下：

```
local size = cc.Director:getInstance():getWinSize()
local defaults = cc.UserDefault:getInstance()

local HomeScene = class("HomeScene",function()
    return cc.Scene:create()
end)

function HomeScene.create()
    local scene = HomeScene.new()
    scene:addChild(scene:createLayer())
    return scene
end

function HomeScene:ctor()
    --场景生命周期事件处理
    local function onNodeEvent(event)
        if event == "enter" then
            self:onEnter()
        elseif event == "enterTransitionFinish" then
            self:onEnterTransitionFinish()
        elseif event == "exit" then
            self:onExit()
        elseif event == "exitTransitionStart" then
            self:onExitTransitionStart()
        elseif event == "cleanup" then
            self:cleanup()
        end
    end
    self:registerScriptHandler(onNodeEvent)
end

-- 创建层
function HomeScene:createLayer()
    cclog("HomeScene init")
    local layer = cc.Layer:create()

    local bg = cc.TMXTiledMap:create("map/red_bg.tmx")                    ①
    layer:addChild(bg)
```

```
    local top =   cc.Sprite:createWithSpriteFrameName("home - top.png")
    layer:addChild(top)
    top:setPosition(cc.p(size.width/2, size.height - top:getContentSize().height / 2))

    local buttom = cc.Sprite:createWithSpriteFrameName("home - end.png")
    buttom:setPosition(cc.p(size.width/2, buttom:getContentSize().height / 2))
    layer:addChild(buttom)
    ...
    return layer
end

function HomeScene:onEnter()
    cclog("HomeScene onEnter")
end

function HomeScene:onEnterTransitionFinish()                                    ②
    cclog("HomeScene onEnterTransitionFinish")
    if defaults:getBoolForKey(MUSIC_KEY) then
        AudioEngine.playMusic(bg_music_1, true)
    end
end

function HomeScene:onExit()
    cclog("HomeScene onExit")
end

function HomeScene:onExitTransitionStart()
    cclog("HomeScene onExitTransitionStart")
end

function HomeScene:cleanup()
    cclog("HomeScene cleanup")
end

return HomeScene
```

上述第①行代码 local bg = cc.TMXTiledMap:create("map/red_bg.tmx")是场景瓦片地图背景,red_bg.tmx 是我们设计的红色底纹的瓦片地图,在 Loading 场景中也使用了这个瓦片地图。

第②行代码 HomeScene:onEnterTransitionFinish()是在场景动画结束之后回调,在该函数中我们判断 UserDefault 中保存的 MUSIC_KEY 键是否为 ture,以便是否播放背景音乐 bg_music_1。

23.4.2 迭代 3.2: 添加菜单

在 Home 场景中有三个菜单,HomeScene.lua 中的相关代码如下:

```
-- 创建层
function HomeScene:createLayer()
    cclog("HomeScene init")
```

```
...
local function menuItemCallback(tag, sender)
    -- 播放音效
    if defaults:getBoolForKey(SOUND_KEY) then
        AudioEngine.playEffect(sound_1)
    end

    if tag == HomeMenuActionTypes.MenuItemStart then
        local GamePlayScene = require("GamePlayScene")
        local scene = GamePlayScene.create()
        local ts = cc.TransitionCrossFade:create(1, scene)
        cc.Director:getInstance():pushScene(ts)
    elseif tag == HomeMenuActionTypes.MenuItemSetting then
        local SettingScene = require("SettingScene")
        local scene = SettingScene.create()
        local ts = cc.TransitionCrossFade:create(1, scene)
        cc.Director:getInstance():pushScene(ts)
    else
        local HelpScene = require("HelpScene")
        local scene = HelpScene.create()
        local ts = cc.TransitionCrossFade:create(1, scene)
        cc.Director:getInstance():pushScene(ts)
    end

end

-- 开始菜单
local startSpriteNormal = cc.Sprite:createWithSpriteFrameName("button.start.png")
local startSpriteSelected = cc.Sprite:createWithSpriteFrameName("button.start-on.png")
local startMenuItem = cc.MenuItemSprite:create( startSpriteNormal, startSpriteSelected)
startMenuItem:registerScriptTapHandler(menuItemCallback)
startMenuItem:setTag(HomeMenuActionTypes.MenuItemStart)

-- 设置菜单
local settingSpriteNormal = cc.Sprite:createWithSpriteFrameName("button.setting.png")
local settingSpriteSelected = cc.Sprite:createWithSpriteFrameName("button.setting-on.png")
local settingMenuItem = cc.MenuItemSprite:create( settingSpriteNormal, settingSpriteSelected)
settingMenuItem:registerScriptTapHandler(menuItemCallback)
settingMenuItem:setTag(HomeMenuActionTypes.MenuItemSetting)

-- 帮助菜单
local helppriteNormal = cc.Sprite:createWithSpriteFrameName("button.help.png")
local helpSpriteSelected = cc.Sprite:createWithSpriteFrameName("button.help-on.png")
local helpMenuItem = cc.MenuItemSprite:create(  helppriteNormal,  helpSpriteSelected)
helpMenuItem:registerScriptTapHandler(menuItemCallback)
helpMenuItem:setTag(HomeMenuActionTypes.MenuItemHelp)

local mu = cc.Menu:create(startMenuItem, settingMenuItem, helpMenuItem)

mu:setPosition(size.width/2, size.height/2)
mu:alignItemsVerticallyWithPadding(12)
layer:addChild(mu)

return layer
```

```
end
```

三个菜单都回调 menuItemCallback 函数，我们在 menuItemCallback 函数中判断 tag 参数。

23.5 任务 4：创建设置场景

创建设置场景过程，要通过文本编辑工具创建 SettingScene 场景类文件 SettingScene.lua。SettingScene.lua 中主要代码如下：

```
...
-- 创建层
function SettingScene:createLayer()
    cclog("SettingScene init")
    local layer = cc.Layer:create()

    local bg = cc.TMXTiledMap:create("map/red_bg.tmx")
    layer:addChild(bg)

    local top = cc.Sprite:createWithSpriteFrameName("setting.page.png")
    top:setPosition(cc.p(size.width/2, size.height - top:getContentSize().height /2))
    layer:addChild(top)

    local function menuSoundToggleCallback(sender)
    if defaults:getBoolForKey(SOUND_KEY, false) then
        defaults:setBoolForKey(SOUND_KEY, false)
    else
        defaults:setBoolForKey(SOUND_KEY, true)
        AudioEngine.playEffect(sound_1)
    end
end

--音效.
local soundOnSprite   = cc.Sprite:createWithSpriteFrameName("check-on.png")
local soundOffSprite   = cc.Sprite:createWithSpriteFrameName("check-off.png")
local soundOnMenuItem = cc.MenuItemSprite:create(soundOnSprite, soundOnSprite)
local soundOffMenuItem = cc.MenuItemSprite:create(soundOffSprite, soundOffSprite)
local soundToggleMenuItem = cc.MenuItemToggle:create(soundOnMenuItem,
                                                     soundOffMenuItem)
soundToggleMenuItem:registerScriptTapHandler(menuSoundToggleCallback)

local function menuMusicToggleCallback(tag, sender)
    if  defaults:getBoolForKey(MUSIC_KEY, false) then
        defaults:setBoolForKey(MUSIC_KEY, false)
        AudioEngine.stopMusic()
    else
        defaults:setBoolForKey(MUSIC_KEY, true)
        AudioEngine.playMusic(bg_music_2, true)
    end
    if defaults:getBoolForKey(SOUND_KEY) then
```

```
        AudioEngine.playEffect(sound_1)
    end
end
--音乐.
local musicOnSprite   = cc.Sprite:createWithSpriteFrameName("check - on.png")
local musicOffSprite   = cc.Sprite:createWithSpriteFrameName("check - off.png")
local musicOnMenuItem = cc.MenuItemSprite:create(musicOnSprite, musicOnSprite)
local musicOffMenuItem = cc.MenuItemSprite:create(musicOffSprite, musicOffSprite)
local musicToggleMenuItem = cc.MenuItemToggle:create(musicOnMenuItem,
                                                     musicOffMenuItem)
musicToggleMenuItem:registerScriptTapHandler(menuMusicToggleCallback)

local  menu = cc.Menu:create(soundToggleMenuItem, musicToggleMenuItem)
menu:setPosition(cc.p(size.width / 2 + 70, size.height /2 + 60))
menu:alignItemsVerticallyWithPadding(12)
layer:addChild(menu, 1)

--Ok菜单事件处理
local function menuOkCallback(sender)
    cc.Director:getInstance():popScene()
    if defaults:getBoolForKey(SOUND_KEY) then
        AudioEngine.playEffect(sound_1)
    end
end
--Ok菜单
local okNormal = cc.Sprite:createWithSpriteFrameName("button.ok.png")
local okSelected = cc.Sprite:createWithSpriteFrameName("button.ok - on.png")
local okMenuItem = cc.MenuItemSprite:create(okNormal, okSelected)
okMenuItem:registerScriptTapHandler(menuOkCallback)

local okMenu = cc.Menu:create(okMenuItem)
okMenu:setPosition(cc.p(190, 50))
layer:addChild(okMenu)

--设置音效和音乐选中状态
if  defaults:getBoolForKey(MUSIC_KEY, false) then
    musicToggleMenuItem:setSelectedIndex(0)
else
    musicToggleMenuItem:setSelectedIndex(1)
end
    if  defaults:getBoolForKey(SOUND_KEY, false) then
        soundToggleMenuItem:setSelectedIndex(0)
    else
        soundToggleMenuItem:setSelectedIndex(1)
    end

    return layer
end
...
```

上述设置场景代码在前面章节中多次解释，这里不再赘述。

23.6 任务5：创建帮助场景

创建帮助场景过程，要通过文本编辑工具创建 HelpScene 场景类文件 HelpScene.lua。HelpScene.lua 中主要代码如下：

```
...

-- 创建层
function HelpScene:createLayer()
    cclog("HelpScene init")
    local layer = cc.Layer:create()

    local bg = cc.TMXTiledMap:create("map/red_bg.tmx")
    layer:addChild(bg)

    local top = cc.Sprite:createWithSpriteFrameName("help.page.png")
    top:setPosition(cc.p(size.width/2, size.height - top:getContentSize().height /2))
    layer:addChild(top)

    -- Ok 菜单事件处理
    local function menuOkCallback(sender)
        -- 播放音效
        if defaults:getBoolForKey(SOUND_KEY) then
            AudioEngine.playEffect(sound_1)
        end
        cc.Director:getInstance():popScene()
    end

    -- Ok 菜单
    local okNormal = cc.Sprite:createWithSpriteFrameName("button.ok.png")
    local okSelected = cc.Sprite:createWithSpriteFrameName("button.ok-on.png")
    local okMenuItem = cc.MenuItemSprite:create(okNormal, okSelected)
    okMenuItem:registerScriptTapHandler(menuOkCallback)

    local okMenu = cc.Menu:create(okMenuItem)
    okMenu:setPosition(cc.p(190, 50))
    layer:addChild(okMenu)

    return layer
end
...
```

上述代码比较简单，不再赘述。

23.7 任务6：游戏场景实现

创建游戏场景过程，要通过文本编辑工具创建 GamePlayScene 场景类文件 GamePlayScene.lua。

23.7.1　迭代 6.1：创建敌人精灵

由于敌人精灵比较复杂，我们不能直接地使用 Sprite 类，而是根据需要进行封装，我们需要继承 Sprite 类，并定义敌人精灵类 Enemy 的特有函数和成员。

Enemy.lua 主要代码如下：

```
local Enemy = class("Enemy",function()
    return cc.Sprite:create()
end)

function Enemy.create(enemyType)                                    ①
    local sprite = Enemy.new(enemyType)                             ②
    return sprite
end

function Enemy:ctor(enemyType)                                      ③

    --精灵帧
    local enemyFramName = EnemyName.Enemy_Stone
    --生命值
    local hitPointsTemp = 0
    --速度
    local velocityTemp = nil

    if enemyType   == EnemyTypes.Enemy_Stone then                   ④
        enemyFramName = EnemyName.Enemy_Stone
        hitPointsTemp = Enemy_initialHitPoints.Enemy_Stone
        velocityTemp = Sprite_Velocity.Enemy_Stone
    elseif  enemyType   == EnemyTypes.Enemy_1 then
        enemyFramName = EnemyName.Enemy_1
        hitPointsTemp = Enemy_initialHitPoints.Enemy_1
        velocityTemp = Sprite_Velocity.Enemy_1
    elseif  enemyType   == EnemyTypes.Enemy_2 then
        enemyFramName = EnemyName.Enemy_2
        hitPointsTemp = Enemy_initialHitPoints.Enemy_2
        velocityTemp = Sprite_Velocity.Enemy_2
    elseif  enemyType   == EnemyTypes.Enemy_Planet then
        enemyFramName = EnemyName.Enemy_Planet
        hitPointsTemp = Enemy_initialHitPoints.Enemy_Planet
        velocityTemp = Sprite_Velocity.Enemy_Planet
    end                                                             ⑤

    self:setSpriteFrame(enemyFramName)                              ⑥
    self:setVisible(false)                                          ⑦

    --设置敌人精灵的基本属性
    self.hitPoints = 0                          --当前的生命值      ⑧
    self.initialHitPoints = hitPointsTemp       --初始的生命值
    self.velocity = velocityTemp                --速度
    self.enemyType = enemyType                  --敌人类型          ⑨

    local body = cc.PhysicsBody:create()                            ⑩
```

```
if enemyType == EnemyTypes.Enemy_Stone
    or enemyType == EnemyTypes.Enemy_Planet then
    body:addShape(cc.PhysicsShapeCircle:create(self:getContentSize().width / 2 - 5))     ⑪
elseif enemyType == EnemyTypes.Enemy_1 then
    local verts = {
        cc.p( - 2.5, - 45.75),
        cc.p( - 29.5, - 27.25),
        cc.p( - 53,  - 0.25),
        cc.p( - 34,43.25),
        cc.p(28, 44.25),
        cc.p(55, - 2.25)}
    body:addShape(cc.PhysicsShapePolygon:create(verts))                                  ⑫
elseif enemyType == EnemyTypes.Enemy_2  then
    local verts = {
        cc.p(1.25, 32.25),
        cc.p(36.75,  - 4.75),
        cc.p( 2.75,  - 31.75),
        cc.p( - 35.75, - 3.25)}
    body:addShape(cc.PhysicsShapePolygon:create(verts))
end

self:setPhysicsBody(body)                                                            ⑬
body:setCategoryBitmask(0x01)                    -- 0001                             ⑭
body:setCollisionBitmask(0x02)                   -- 0010                             ⑮
body:setContactTestBitmask(0x01)                 -- 0001                             ⑯
    self:spawn()                                                                     ⑰

    <游戏调度代码>


end
```

上述第①行代码定义 Enemy.create(enemyType)工厂函数,其中 enemyType 参数是敌人类型。第②行代码 Enemy.new(enemyType)是创建 Enemy 对象,它将调用第③行代码的 Enemy:ctor(enemyType)构造函数。

第④和第⑤行代码是根据不同敌人类型获得精灵帧名、生命值(承受的打击次数)、速度。

第⑥行代码 self:setSpriteFrame(enemyFramName)是设置敌人精灵帧名,第⑦行代码 self:setVisible(false)是隐藏该敌人精灵。

第⑧和第⑨行代码是设置敌人精灵的基本属性,在 Lua 中对象定义属性方式是 self.xxx,其中 xxx 是属性名,例如第⑧行的 self.hitPoints 是定义并设置 hitPoints 属性值。

上述第⑩行代码 local body = cc.PhysicsBody:create()创建物理世界中的一个物体。第⑪行代码 body:addShape(cc.PhysicsShapeCircle:create(self:getContentSize().width / 2−5))是在敌人类型是陨石和行星情况下,为物体添加圆形形状,其中 self:getContentSize().width / 2 是半径,−5 是一个修正值,由于美工给我们做的图片如图 23-6 所示,球体与边界有一些空白。

代码第⑫行 body:addShape(cc.PhysicsShapePolygon:create(verts))是为物体添加多边形形状，这是针对飞机形状的敌人(见图 23-7(a))。我们需要指定形状的顶点坐标，其中 verts 是顶点坐标数组，如图 23-7(b)所示。

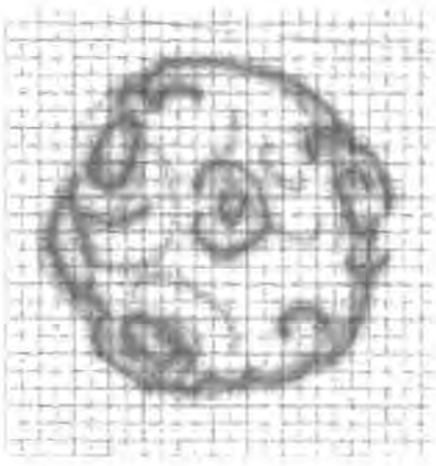


图 23-6　陨石图片的空白

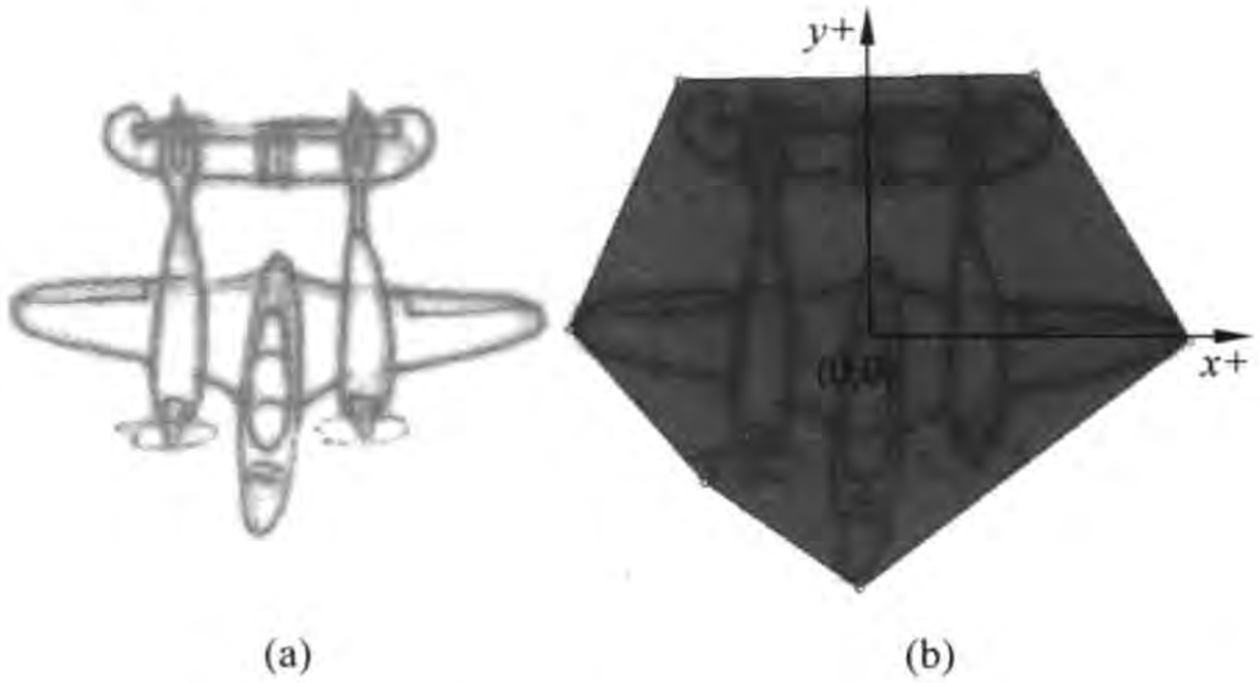


图 23-7　多边形顶点

提示　由于底层封装了 chipmunk 引擎，chipmunk 要求多边形顶点数据必须是按照顺时针，必须是凸多边形(见图 23-7(b))。如果遇到凹多边形情况可以把它分割为几个凸多边形。另外，顶点坐标的原点在图形的中心，并注意它向上是 y 轴正方向，向右是 x 轴正方向，如图 23-7(b)所示。

代码第⑬行 self:setPhysicsBody(body)是将上面定义好的物体对象，设置到敌人精灵对象中，这样就可以使得敌人精灵对象具有物理特性了。

代码第⑭行 body:setCategoryBitmask(0x01)是设置物体类别掩码，第⑮行代码 body:setCollisionBitmask(0x02)是设置物体碰撞掩码，由于 0x01 与 0x02 进行逻辑与运算结果为零，这样就说明敌人精灵对象自己之间不能发生碰撞反应。第⑯行代码 body:setContactTestBitmask(0x01)是设置接触检测掩码，如果我们设置玩家飞机的接触掩码的代码如下：

```
fighterBody:setContactTestBitmask(0x01)
```

则玩家飞机与敌人的接触检测掩码逻辑与运算结果是非零，说明玩家飞机与敌人会发生接触。

代码第⑰行 self:spawn()是调用 spawn()函数产生敌人，我们会在下面介绍该函数。

Enemy.lua 中游戏调度代码如下：

```
function Enemy:ctor(enemyType)
    ...
    local function update(delta)                                    ①

        -- 设置陨石和行星旋转.
        if enemyType   == EnemyTypes.Enemy_Stone then
            self:setRotation(self:getRotation() - 0.5)              ②
```

```
    elseif  enemyType   == EnemyTypes.Enemy_Planet then
        self:setRotation(self:getRotation() + 1)                              ③
    end

    local x,y = self:getPosition()

    self:setPosition(cc.p(x + self.velocity.x * delta, y + self.velocity.y * delta))   ④

    x,y = self:getPosition()

    if  y + self:getContentSize().height /2 < 0  then                         ⑤
        self:spawn()                                                          ⑥
    end
end

self:scheduleUpdateWithPriorityLua(update, 0)                                 ⑦

function onNodeEvent(tag)                                                     ⑧
    if tag == "exit" then
        --停止游戏调度
        self:unscheduleUpdate()                                               ⑨
    end
end
self:registerScriptHandler(onNodeEvent)                                       ⑩
...
end
```

上述第①行代码的 update 函数是游戏调度函数，在该函数中我们需要改变当前对象运动的位置和旋转的角度，这样就可以在场景中看到不断运动的敌人了。其中第②行代码是逆时针旋转陨石类型敌人，−0.5 表示逆时针旋转。第③行代码是顺时针旋转行星类型敌人，+1 表示顺时针旋转。

代码第④行是重新计算本次 update 敌人移动的新位置。第⑤行代码是在判断敌人运动到屏幕之外重新调用生成敌人，图 23-8 所示虚线表示敌人，从图中不难看出，敌人运动到屏幕之外的判断语句是 y + self:getContentSize(). height /2 < 0。注意这样计算的前提是要把敌人的锚点设置为(0.5,0.5)。

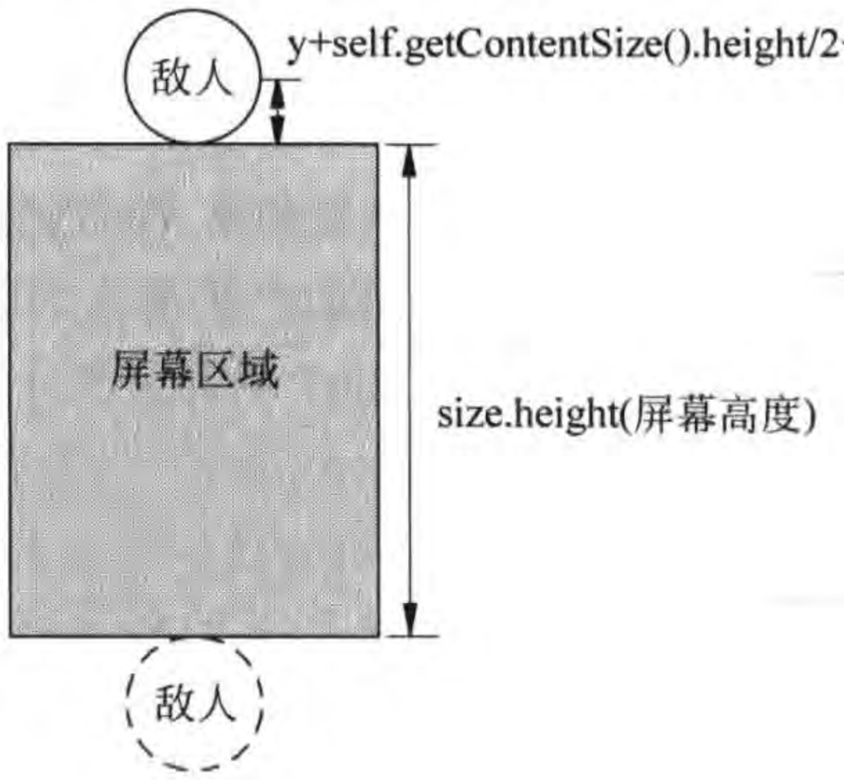


图 23-8　敌人运动示意图

第⑥行代码 spawn 函数是生成敌人，所谓“生成敌人”并不是创建敌人对象，而是重新调整它的坐标，让它从屏幕上边开始向下运动，见图 23-8 所示实线表示的敌人。

第⑦行代码 self:scheduleUpdateWithPriorityLua(update, 0)是通过 Lua 语言开始游戏调度，其中 update 是调度函数，见第①行代码，0 是调度优先级。

第⑧行代码 onNodeEvent 是层事件回调函数，其中第⑨行代码 self:unscheduleUpdate()是停止游戏调度，第⑩行代码 self:registerScriptHandler(onNodeEvent)是注册层事件。

Enemy.lua 中产生敌人函数代码如下：

```
function Enemy:spawn()                                                    ①
    local yPos = size.height + self:getContentSize().height /2            ②

    local xPos = math.random() * (size.width - self:getContentSize().width)
                                        + self:getContentSize().width / 2  ③
    self:setPosition(cc.p(xPos, yPos))                                    ④
    self:setAnchorPoint(cc.p(0.5, 0.5))                                   ⑤

    self.hitPoints = self.initialHitPoints                                ⑥
    self:setVisible(true)                                                 ⑦

end
```

上述第①行代码是定义 Enemy:spawn()函数产生敌人。第②行代码计算出它在屏幕上边界 y 轴坐标。第③行代码是随机产生 x 轴坐标，其中 math.random()是产生 0～1 之间的随机数。

第④行代码是重新设置敌人精灵的位置，第⑤行代码是重新设置敌人精灵的锚点。第⑥行代码是用 initialHitPoints 属性重新设置 hitPoints 属性。第⑦行代码 self:setVisible(true)是设置敌人精灵可见。

23.7.2　迭代 6.2：创建玩家飞机精灵

玩家飞机精灵没有敌人精灵那么复杂，玩家飞机只有一种，我们需要继承 Sprite 类，并定义飞机精灵类 Fighter 的特有函数和成员。

Fighter.lua 主要代码如下：

```
...
local size = cc.Director:getInstance():getWinSize()
local defaults = cc.UserDefault:getInstance()

local Fighter = class("Fighter",function()
    return cc.Sprite:create()
end)

function Fighter.create(spriteFrameName)                                  ①
    local fighter = Fighter.new(spriteFrameName)                          ②
    return fighter
end
```

```
function Fighter:ctor(spriteFrameName)                                    ③
    self.hitPoints = Fighter_hitPoints              --当前的生命值          ④
    self:setSpriteFrame(spriteFrameName)                                  ⑤

    local ps = cc.ParticleSystemQuad:create("particle/fire.plist")        ⑥
    --在飞机下面.
    ps:setPosition(cc.p(self:getContentSize().width / 2, 0))              ⑦
    ps:setScale(0.5)                                                      ⑧
    self:addChild(ps)                                                     ⑨

    local verts = {
        cc.p( -43.5,15.5),
        cc.p( -23.5,33),
        cc.p(28.5,34),
        cc.p(48,17.5),
        cc.p(0, -39.5)}                                                   ⑩

    local body = cc.PhysicsBody:createPolygon(verts)                      ⑪
    body:setCategoryBitmask(0x01)                   --0001                ⑫
    body:setCollisionBitmask(0x02)                  --0010                ⑬
    body:setContactTestBitmask(0x01)                --0001                ⑭
    self:setPhysicsBody(body)                                             ⑮

end
...
```

上述第①行代码定义 Fighter.create(spriteFrameName)工厂函数,其中 spriteFrameName 参数是精灵帧名。第②行代码 Fighter.new(spriteFrameName)是创建 Fighter 对象,它将调用第③行代码的 Fighter:ctor(spriteFrameName)构造函数。

第④行代码 self.hitPoints = Fighter_hitPoints 是初始化 hitPoints 属性。第⑤行代码 self:setSpriteFrame(spriteFrameName)是设置 Fighter 对象精灵帧名。

第⑥~⑨行代码是创建飞机后面喷射烟雾粒子效果,第⑦行代码是设置烟雾粒子在飞机的下面。由于粒子设计人员设计的粒子比较大,我们通过第⑧行代码 ps:setScale(0.5)缩小一半。第⑨行代码 self:addChild(ps)是将粒子系统添加到飞机精灵上。

第⑩行代码是定义飞机顶点坐标。第⑪行代码是创建物理世界中的多边形物体。第⑫行代码是定义物体类别掩码,第⑬行代码是定义碰撞掩码,由于敌人设置也是 0x01 和 0x02,0x01 和 0x02 进行逻辑与运算结果为零,这说明飞机不能与敌人发生碰撞反应。第⑭行代码是设置接触检测掩码,因为设置的是 0x01,敌人设置的也是 0x01,所以玩家飞机和敌人会发生接触。

第⑮行代码 self:setPhysicsBody(body)是将上面定义好的物体对象,设置到 Fighter 对象中,这就可以使得 Fighter 对象具有物理特性了。

Fighter.lua 中的设置位置函数代码如下:

```
function Fighter:setPos(newPosition)
```

```
local halfWidth = self:getContentSize().width /2
local halfHeight = self:getContentSize().height /2
local pos_x = newPosition.x
local  pos_y = newPosition.y

if  pos_x < halfWidth then                                          ①
    pos_x = halfWidth
elseif  pos_x > (size.width - halfWidth) then
    pos_x = size.width - halfWidth
end                                                                 ②

if  pos_y < halfHeight  then                                        ③
    pos_y = halfHeight
elseif pos_y > (size.height - halfHeight) then
    pos_y = size.height - halfHeight
end                                                                 ④

self:setPosition(cc.p(pos_x,pos_y))                                 ⑤
self:setAnchorPoint(cc.p(0.5, 0.5))                                ⑥

end
```

在 setPos 函数是重新设置 Fighter 对象的位置，该函数可以防止飞机超出屏幕。其中第①和第②行是计算 Fighter 对象的 x 轴坐标，第③和第④行代码是计算 Fighter 对象的 y 轴坐标。

第⑤行代码是重新设置 Fighter 对象位置。第⑥行代码是重新设置 Fighter 对象锚点。

23.7.3　迭代 6.3：创建炮弹精灵

Bullet 也没有直接使用 Sprite 类，而是进行了封装，让它继承 Sprite 类。Bullet.lua 主要代码如下：

```
local Bullet = class("Bullet",function()
    return cc.Sprite:create()
end)

function Bullet.create(spriteFrameName)
    local sprite = Bullet.new(spriteFrameName)
    return sprite
end

function Bullet:ctor(spriteFrameName)

    self:setSpriteFrame(spriteFrameName)
    self:setVisible(false)
    self.velocity = Sprite_Velocity.Bullet        --速度

    local body = cc.PhysicsBody:createBox(self:getContentSize())
    body:setCategoryBitmask(0x01)                 --0001                ①
    body:setCollisionBitmask(0x02)                --0010                ②
    body:setContactTestBitmask(0x01)              --0001                ③
```

```
        self:setPhysicsBody(body)

        function onNodeEvent(tag)
            if tag == "exit" then
                --开始游戏调度
                self:unscheduleUpdate()
            end
        end
        self:registerScriptHandler(onNodeEvent)

    end
```

上述第①行代码是定义物体类别掩码,第②行代码是定义碰撞掩码,由于敌人设置也是0x01和0x02,0x01和0x02进行逻辑与运算结果为零,这说明炮弹不能与敌人发生碰撞反应。第③行代码是设置接触检测掩码,因为设置的是0x01,敌人也设置的是0x01,所以炮弹和敌人会发生接触。

Bullet.lua中的发射炮弹函数代码如下:

```
function Bullet:shootBulletFromFighter(fighter)

    local  fighterPosX,fighterPosY = fighter:getPosition()

    self:setPosition(cc.p(fighterPosX,  fighterPosY + fighter:getContentSize().height/2))    ①
    self:setVisible(true)                                                                    ②

    --开始游戏调度
    function update(delta)                                                                   ③

        local x,y = self:getPosition()
        self:setPosition(cc.p(x + self.velocity.x * delta,y + self.velocity.y * delta))     ④
        x,y = self:getPosition()

        if  y > size.height then                                                             ⑤
            self:setVisible(false)                                                           ⑥
            self:unscheduleUpdate()                                                          ⑦
        end

    end
    self:scheduleUpdateWithPriorityLua(update, 0)                                            ⑧
end
```

上述第①行代码是设置发射炮弹的位置,炮弹的位置是在飞机的头部。第②行代码是设置炮弹可见。

第③行代码update(delta)是游戏调度函数,需要在第⑧行代码开启游戏调度。第④行代码是根据时间重新计算炮弹的位置,其中velocity是炮弹的飞行速度。

第⑤行代码判断炮弹是否超出屏幕之外,如果超出则将炮弹设置为不可见(见第⑥行代码),然后通过第⑦行代码停止游戏循环。

23.7.4　迭代6.4：初始化游戏场景

为了能够清楚地介绍游戏场景实现，我们将分成几个部分为大家介绍。

我们先来看看初始化游戏场景，在这一部分中涉及的函数有：ctor、onExit 和 onEnterTransitionFinish 和 createInitBGLayer。

我们可以将 ctor 和 createInitBGLayer 函数一起考虑，这两个函数是在场景初始化的时候调用，且在场景生命周期中只调用一次，例如：游戏结束进入游戏结束场景后再返回的时候，这两个函数不会被调用，因此不经常变化的精灵如背景等，可以在这个函数中初始化。

GamePlayScene.lua 中的初始化主要代码如下：

```
function GamePlayScene.create()
    local scene = GamePlayScene.new()
    return scene
end

function GamePlayScene:ctor()                                          ①
    cclog("GamePlayScene init")

    self:addChild(self:createInitBGLayer())
    --场景生命周期事件处理
    local function onNodeEvent(event)
        if event == "enter" then
            self:onEnter()
        elseif event == "enterTransitionFinish" then
            self:onEnterTransitionFinish()
        elseif event == "exit" then
            self:onExit()
        elseif event == "exitTransitionStart" then
            self:onExitTransitionStart()
        elseif event == "cleanup" then
            self:cleanup()
        end
    end
    self:registerScriptHandler(onNodeEvent)
end

-- create 背景层
function GamePlayScene:createInitBGLayer()                             ②
    cclog("背景层初始化")
    local bgLayer = cc.Layer:create()

    local bg = cc.TMXTiledMap:create("map/blue_bg.tmx")
    bgLayer:addChild(bg,0,GameSceneNodeTag.BatchBackground)

    --放置发光粒子背景
    local ps = cc.ParticleSystemQuad:create("particle/light.plist")
    ps:setPosition(cc.p(size.width / 2, size.height / 2))
    bgLayer:addChild(ps,0,GameSceneNodeTag.BatchBackground)

    --添加背景精灵1.
```

```
    local sprite1 = cc.Sprite:createWithSpriteFrameName("gameplay.bg.sprite-1.png")
    sprite1:setPosition(cc.p(-50, -50))
    bgLayer:addChild(sprite1,0,GameSceneNodeTag.BatchBackground)

    local ac1 = cc.MoveBy:create(20, cc.p(500, 600))
    local ac2 = ac1:reverse()
    local as1 = cc.Sequence:create(ac1, ac2)
    sprite1:runAction(cc.RepeatForever:create(cc.EaseSineInOut:create(as1)))

    --添加背景精灵 2.
    local sprite2 = cc.Sprite:createWithSpriteFrameName("gameplay.bg.sprite-2.png")
    sprite2:setPosition(cc.p(size.width, 0))
    bgLayer:addChild(sprite2,0,GameSceneNodeTag.BatchBackground)

    local ac3 = cc.MoveBy:create(10, cc.p(-500, 600))
    local ac4 = ac3:reverse()
    local as2 = cc.Sequence:create(ac3, ac4)
    sprite2:runAction(cc.RepeatForever:create(cc.EaseExponentialInOut:create(as2)))

    return bgLayer
end
```

上述第①行代码 ctor()是构造函数。第②行代码 createInitBGLayer()是创建背景场景函数,该函数中创建了瓦片地图背景精灵、背景发光粒子,以及两个背景有动画效果的精灵。

在 createInitBGLayer()函数中创建的这些精灵,在退出函数 onExit 中是不能移除的。GamePlayScene.lua 中的 onExit 函数代码如下:

```
function GamePlayScene:onExit()
    cclog("GamePlayScene onExit")
    --停止游戏调度
    if (schedulerId ~= nil) then
        scheduler:unscheduleScriptEntry(schedulerId)                    ①
    end
    --注销事件监听器.
    local eventDispatcher = cc.Director:getInstance():getEventDispatcher()
    if nil ~= touchFighterlistener then
        eventDispatcher:removeEventListener(touchFighterlistener)       ②
    end
    if nil ~= contactListener then
        eventDispatcher:removeEventListener(contactListener)            ③
    end
    --删除 layer 节点以及其子节点
    mainLayer:removeAllChildren()                                       ④
    mainLayer:removeFromParent()                                        ⑤
    mainLayer = nil

end
```

在 onExit 函数中的第①行代码是停止游戏调度,第②行代码注销触摸事件监听器 touchFighterlistener,第③行代码注销接触检测事件监听器 contactListener。第④行代码 mainLayer:removeAllChildren()是移除 mainLayer 的所有的子节点,第⑤行代码

mainLayer:removeFromParent()是移除 mainLayer 层对象。

当然在游戏场景中除了背景层中的精灵外，还有很多精灵等 Node 元素需要在每次显示游戏场景时候都要初始化，我们可以把这些初始化放入到 onEnter、createLayer 和 onEnterTransitionFinish 函数中。主要是在 onEnter 函数中实现这些初始化。

GamePlayScene.lua 中的 onEnter 函数主要代码如下：

```
function GamePlayScene:onEnter()
    cclog("GamePlayScene onEnter")
    self:addChild(self:createLayer())                                    ①
end
```

上述第①行代码通过 self:createLayer()调用 createLayer()函数创建游戏主层，并将其添加到 GamePlayScene 游戏场景中。

GamePlayScene.lua 中的 createLayer 函数主要代码如下：

```
-- 创建 Main 层
function GamePlayScene:createLayer()

    mainLayer = cc.Layer:create()

    -- 添加陨石 1.
    local stone1 = Enemy.create(EnemyTypes.Enemy_Stone)
    mainLayer:addChild(stone1, 10 , GameSceneNodeTag.Enemy)

    -- 添加行星.
    local planet = Enemy.create(EnemyTypes.Enemy_Planet)
    mainLayer:addChild(planet, 10 , GameSceneNodeTag.Enemy)

    -- 添加敌机 1.
    local enemyFighter1 = Enemy.create(EnemyTypes.Enemy_1)
    mainLayer:addChild(enemyFighter1, 10 , GameSceneNodeTag.Enemy)

    -- 添加敌机 2.
    local enemyFighter2 = Enemy.create(EnemyTypes.Enemy_2)
    mainLayer:addChild(enemyFighter2, 10 , GameSceneNodeTag.Enemy)

    fighter = Fighter.create("gameplay.fighter.png")
    fighter:setPos(cc.p(size.width / 2, 70))
    mainLayer:addChild(fighter, 10, GameSceneNodeTag.Fighter)

    ...
<省略菜单、事件处理、游戏循环调度代码>
    ...

    -- 接触检测事件回调函数
    local function touchBegan(touch, event)                              ①
        return true                                                      ②
    end

    -- 接触检测事件回调函数
    local function touchMoved(touch, event)                              ③
```

```
        -- 获取事件所绑定的 node
        local node = event:getCurrentTarget()                                    ④

        local currentPosX, currentPosY = node:getPosition()
        local diff = touch:getDelta()
        -- 移动当前按钮精灵的坐标位置
        node:setPos(cc.p(currentPosX + diff.x, currentPosY + diff.y))            ⑤
    end

    ...
    -- 创建一个事件监听器 OneByOne 为单点触摸
    touchFighterlistener = cc.EventListenerTouchOneByOne:create()
    -- 设置是否吞没事件,在 onTouchBegan 方法返回 true 时吞没
    touchFighterlistener:setSwallowTouches(true)
    -- EVENT_TOUCH_BEGAN 事件回调函数
    touchFighterlistener:registerScriptHandler(touchBegan,cc.Handler.EVENT_TOUCH_BEGAN )
    -- EVENT_TOUCH_MOVED 事件回调函数
    touchFighterlistener:registerScriptHandler(touchMoved,cc.Handler.EVENT_TOUCH_MOVED )

    local eventDispatcher = cc.Director:getInstance():getEventDispatcher()
    -- 添加监听器
    eventDispatcher:addEventListenerWithSceneGraphPriority(touchFighterlistener, fighter)    ⑥

    ...

    --分数
    score = 0                                                                    ⑦
    --记录 0～999 分数
    scorePlaceholder = 0                                                         ⑧

    --在状态栏中设置玩家的生命值
    self:updateStatusBarFighter()                                               ⑨
    --在状态栏中显示得分
    self:updateStatusBarScore()                                                 ⑩


    return mainLayer                                                            ⑪
end
```

上述代码初始化了 4 种类型的敌人和玩家飞机。第①行代码是触摸开始事件回调函数,第②行代码是在该函数中返回 true,只有开始事件函数返回 true 才能触发触摸移动等事件。第③行代码是触摸移动事件回调函数。第④行代码 local node = event:getCurrentTarget( ) 是返回触摸对象,其中返回值是第⑥行代码中 addEventListenerWithSceneGraphPriority 函数的第二个参数 fighter。所以第⑤行代码 node:setPos(cc.p(currentPosX + diff.x, currentPosY + diff.y))是设置玩家飞机位置,因为 node 对象就是第⑥行 fighter 对象。

第⑦行代码是初始化玩家得分变量 score,第⑧行代码是初始化记录 0～999 分数变量 scorePlaceholder。它们都是在 GamePlayScene.lua 开始部分声明的模块变量,声明代码如下:

```
-- 分数
local score = 0
-- 记录 0～999 分数
local scorePlaceholder = 0
```

第⑨行代码是在状态栏中设置玩家的生命值，我们会在任务 6.9 介绍。第⑩行代码是在状态栏中显示得分，我们会在任务 6.10 中介绍。

第⑪行代码是返回游戏主层 mainLayer 对象，它是在 GamePlayScene.lua 开始部分声明的模块变量，声明代码如下：

```
-- 主游戏层
local mainLayer
```

GamePlayScene.lua 中的 onEnterTransitionFinish 函数主要代码如下：

```
function GamePlayScene:onEnterTransitionFinish()
    cclog("GamePlayScene onEnterTransitionFinish")
    if defaults:getBoolForKey(MUSIC_KEY) then
        AudioEngine.playMusic(bg_music_2, true)
    end
end
```

在该函数中我们主要初始化是否播放背景音乐。

23.7.5　迭代 6.5：游戏场景菜单实现

在游戏场景中有三个菜单：暂停、返回主页和继续游戏。暂停菜单位于场景的左上角，点击暂停菜单，会弹出返回主页和继续游戏菜单。因此我们需要在 onEnter 函数中初始化暂停菜单。

GamePlayScene.lua 中的暂停菜单相关代码如下：

```
function GamePlayScene:createLayer()

    ...
    -- 暂停菜单回调函数
    local function menuPauseCallback(sender)
        cclog("menuPauseCallback")
        if defaults:getBoolForKey(SOUND_KEY)  then
            AudioEngine.playEffect(sound_1)
    end

    -- 暂停当前层中的 node
    mainLayer:pause()                                                    ①
    if (schedulerId ~= nil) then
        scheduler:unscheduleScriptEntry(schedulerId)                     ②
    end

    -- layer 子节点暂停
    local pChildren = mainLayer:getChildren()                            ③
    for i = 1, #pChildren , 1 do                                         ④
```

```
        local  child = pChildren[i]                                    ⑤
        child:pause()                                                  ⑥
    end

    --返回主菜单
    local backNormal = cc.Sprite:createWithSpriteFrameName("button.back.png")
    local backSelected = cc.Sprite:createWithSpriteFrameName("button.back-on.png")
    local backMenuItem = cc.MenuItemSprite:create(backNormal, backSelected)
    backMenuItem:registerScriptTapHandler(menuBackCallback)            ⑦

    --继续游戏菜单
    local resumeNormal = cc.Sprite:createWithSpriteFrameName("button.resume.png")
    local resumeSelected = cc.Sprite:createWithSpriteFrameName("button.resume-on.png")
    local resumeMenuItem = cc.MenuItemSprite:create(resumeNormal, resumeSelected)
    resumeMenuItem:registerScriptTapHandler(menuResumeCallback)        ⑧

    menu = cc.Menu:create(backMenuItem,resumeMenuItem)
    menu:alignItemsVertically()
    menu:setPosition(cc.p(size.width / 2, size.height / 2))

    mainLayer:addChild(menu,50,1000)

end

--初始化暂停按钮.
local pauseSprite = cc.Sprite:createWithSpriteFrameName("button.pause.png")
local pauseMenuItem = cc.MenuItemSprite:create(pauseSprite, pauseSprite)
pauseMenuItem:registerScriptTapHandler(menuPauseCallback)              ⑨

local pauseMenu = cc.Menu:create(pauseMenuItem)
pauseMenu:setPosition(cc.p(30, size.height - 28))

mainLayer:addChild(pauseMenu,300,999)
...
end
```

上述第①行代码 mainLayer:pause()是暂停层,第②行代码 scheduler:unschedule-ScriptEntry(schedulerId)是停止游戏调度,开启游戏调度代码如下:

```
schedulerId = scheduler:scheduleScriptFunc(shootBullet, 0.2, false)
```

第③~⑥行代码是暂停子节点,第③行代码 mainLayer:getChildren()是取出所有子节点,第④行是遍历所有子节点,第⑤行代码 pChildren[i]是取出子节点。第⑥行代码 child:pause()是暂停子节点。

第⑦行代码是注册返回主菜单事件,回调 menuBackCallback 函数,该函数我们稍后再介绍。第⑧行代码是注册继续游戏菜单,回调 menuResumeCallback 函数,该函数我们稍后再介绍。第⑨行代码是注册暂停按钮,回调 menuPauseCallback 函数,该函数是在上面定义的。

GamePlayScene.lua 中的返回主页和继续游戏菜单回调函数相关代码如下:

```
function GamePlayScene:createLayer()
    ...
    --返回主页菜单回调函数
    local function menuBackCallback(sender)
        cclog("menuBackCallback")
        cc.Director:getInstance():popScene()
        if defaults:getBoolForKey(SOUND_KEY)  then
            AudioEngine.playEffect(sound_1)
        end
    end
    --继续菜单回调函数
local function menuResumeCallback(sender)

    cclog("menuResumeCallback")
    if defaults:getBoolForKey(SOUND_KEY)  then
        AudioEngine.playEffect(sound_1)
    end

    mainLayer:resume()                                                    ①
    schedulerId = nil
    schedulerId = scheduler:scheduleScriptFunc(shootBullet, 0.2, false)   ②

    --layer子节点继续
    local pChildren = mainLayer:getChildren()
    for i = 1, #pChildren , 1 do
        local  child = pChildren[i]
            child:resume()                                                ③
        end
        mainLayer:removeChild(menu)
    end

    <暂停菜单回调函数>
    ...
end
```

上述代码 menuBackCallback 是返回主页菜单回调函数。menuResumeCallback 是继续游戏菜单回调函数，在该函数中是让游戏场景继续执行，它的处理与暂停相反，其中第①行代码 mainLayer:resume()是继续当前层的 Node 元素。

第②行代码是开启游戏调度代码，schedulerId 和 scheduler 是在 GamePlayScene.lua 开始部分声明的模块变量，声明代码如下：

```
local schedulerId = nil
local scheduler = cc.Director:getInstance():getScheduler()
```

其中 scheduler 变量是游戏调度 Scheduler 类的一个实例，该实例是通过 cc.Director:getInstance():getScheduler()语句获得的。还有 schedulerId 变量是通过 scheduler:scheduleScriptFunc(shootBullet, 0.2, false)语句返回的，它是游戏调度 ID，这个 ID 在停止游戏调度时使用。

第③行代码是暂停它们的子 Node 元素。

23.7.6 迭代 6.6：玩家飞机发射炮弹

玩家飞机需要不断发射炮弹，这个过程不需要玩家控制，需要一个调度计划定时重复发射，我们在 onEnter 函数和 menuResumeCallback 继续游戏菜单回调函数中开始游戏调度，相关代码如下：

```
function GamePlayScene:createLayer()
    ...
    -- 继续菜单回调函数
    local function menuResumeCallback(sender)
        ...
        schedulerId = scheduler:scheduleScriptFunc(shootBullet, 0.2, false)          ①
        ...
    end

    <暂停菜单回调函数>
    ...
     -- 每 0.2 秒调用 shootBullet 函数发射 1 发炮弹
     schedulerEntry = scheduler:scheduleScriptFunc(shootBullet, 0.2, false)          ②

end
```

上述第①行代码是在继续菜单回调函数中开始游戏调度，这个调度语句我们在前文中介绍过。另外，第②行代码是进入到场景的时候调用。scheduler:scheduleScriptFunc(shootBullet, 0.2, false)语句中第一个参数 shootBullet 是调度函数，第二个参数 0.2 是调度时间间隔，单位为秒，第三个参数是是否暂停标识，false 表示反复执行。

调度函数 shootBullet 也是在 onEnter 函数中编写的，并且要保证调度函数 shootBullet 代码在②语句之前定义。

调度函数 shootBullet 代码如下：

```
function GamePlayScene:createLayer()
    ...
    -- 开始游戏调度
    local function shootBullet(delta)                                              ①

        if  nil ~= fighter and fighter:isVisible() then                            ②
            local bullet = Bullet.create("gameplay.bullet.png")                    ③
            mainLayer:addChild(bullet, 0, GameSceneNodeTag.Bullet)                 ④
            bullet:shootBulletFromFighter(fighter)                                 ⑤
        end

    end


    -- 继续菜单回调函数
    local function menuResumeCallback(sender)
        ...
        schedulerId = scheduler:scheduleScriptFunc(shootBullet, 0.2, false)
        ...
```

```
    end

    <暂停菜单回调函数>
    ...
     -- 每 0.2 秒调用 shootBullet 函数发射 1 发炮弹
     schedulerEntry = scheduler:scheduleScriptFunc(shootBullet, 0.2, false)

end
```

上述第①行代码是定义 shootBullet 函数，第②行代码是判断飞机精灵是否可见，如果飞机精灵不可见则不能发射炮弹。第③行代码 local bullet = Bullet.create("gameplay.bullet.png")是创建炮弹精灵。第④行代码是将炮弹添加到当前 mainLayer 层中，第⑤行代码 bullet:shootBulletFromFighter(fighter)是发射炮弹。

23.7.7　迭代 6.7：炮弹与敌人的接触检测

本游戏项目需要检测接触的是：玩家发射的炮弹与敌人之间以及玩家飞机与敌人之间。接触检测中我们引入了物理引擎检测接触，这样会更加精确。为了能够在游戏场景中使用物理引擎，需要将当前场景变成具有物理世界的场景，为此，需要修改 GamePlayScene 类声明，代码如下：

```
local GamePlayScene = class("GamePlayScene",function()
    local scene = cc.Scene:createWithPhysics()                                    ①
    -- scene:getPhysicsWorld():setDebugDrawMask(cc.PhysicsWorld.DEBUGDRAW_ALL)   ②
    -- 0,0 不受到重力的影响
    scene:getPhysicsWorld():setGravity(cc.p(0,0))                                 ③
    return scene
end)
```

上述第①行代码用 cc.Scene:createWithPhysics()语句替代 cc.Scene:create()语句，其中 createWithPhysics()语句是创建一个具有物理世界的场景。

第②行代码是设置在物理世界中绘制调试遮罩，这会把物体的形状绘制出来。

第③行代码 scene:getPhysicsWorld():setGravity(cc.p(0,0))是设置重力，这里重力设置的是 cc.p(0,0)，表示在 x 轴和 y 轴方向都没有重力作用，让物体处于“失重状态”，这样在我们控制物体运动时候不会受到重力的影响。其他的物理世界设置可以根据自己需要而设定。

为了能够检测物体之间的接触，我们需要在 onEnter 函数中注册接触检测事件监听器。相关代码如下：

```
function GamePlayScene:createLayer()
    ...
    -- 接触检测事件回调函数
    local function onContactBegin(contact)                                        ①

        local spriteA = contact:getShapeA():getBody():getNode()                   ②
        local spriteB = contact:getShapeB():getBody():getNode()                   ③
```

```
<省略代码：检测飞机与敌人的接触>

---------------------检测  炮弹与敌人的接触 start---------------------
local enemy2 = nil
if  spriteA:getTag() == GameSceneNodeTag.Bullet
    and  spriteB:getTag() == GameSceneNodeTag.Enemy then                    ④
    --不可见的炮弹不发生接触
    if   spriteA:isVisible() == false then
        return false
    end
    --使得炮弹消失
    spriteA:setVisible(false)
    enemy2 = spriteB
end

if  spriteA:getTag() == GameSceneNodeTag.Enemy
    and spriteB:getTag() == GameSceneNodeTag.Bullet then                    ⑤
    --不可见的炮弹不发生接触
    if  spriteB:isVisible() == false then
        return false
    end
    --使得炮弹消失
    spriteB:setVisible(false)
    enemy2 = spriteA
end

if nil ~= enemy2  then--发生接触                                            ⑥
    self:handleBulletCollidingWithEnemy(enemy2)                             ⑦
end
----------------------检测  炮弹与敌人的接触 end----------------------
return false
end

...
-- 创建一个接触检测事件监听器
contactListener = cc.EventListenerPhysicsContact:create()                   ⑧
contactListener:registerScriptHandler(onContactBegin,
            cc.Handler.EVENT_PHYSICS_CONTACT_BEGIN)                         ⑨

local eventDispatcher = cc.Director:getInstance():getEventDispatcher()
-- 添加监听器
eventDispatcher:addEventListenerWithSceneGraphPriority(touchFighterlistener, fighter)
eventDispatcher:addEventListenerWithSceneGraphPriority(contactListener, mainLayer)  ⑩
    ...
end
```

上述第①行代码是定义接触检测事件回调函数 onContactBegin，该函数将在接触检测事件调用。第②和第③行代码是获得两个接触的精灵对象。在第④和第⑤行代码通过它们的 tag 属性可以实现这个判断，在判断为 ture 情况下，设置炮弹不可见，然后将敌人对象指针赋值给 enemy2 指针。

在第⑥行代码判断 enemy2 指针是否为 nil，如果非 nil 说明炮弹与敌人之间发生了接

触，通过第⑦行代码 self:handleBulletCollidingWithEnemy(enemy2)语句处理接触。

第⑧行代码是创建一个接触检测事件监听器，第⑨行代码注册触摸事件监听器，只监听触摸开始事件和接触检测事件。

第⑩行代码是添加监听器 contactListener 到事件派发器 eventDispatcher。

handleBulletCollidingWithEnemy 函数代码如下：

```
--炮弹与敌人的接触检测
function GamePlayScene:handleBulletCollidingWithEnemy(enemy)
enemy.hitPoints = enemy.hitPoints - 1                                    ①

if  enemy.hitPoints <= 0 then                                            ②
    --爆炸和音效
    local node = mainLayer:getChildByTag(GameSceneNodeTag.ExplosionParticleSystem)  ③
    if nil ~= node  then
        self:removeChild(node)
    end
    local explosion = cc.ParticleSystemQuad:create("particle/explosion.plist")
    explosion:setPosition(enemy:getPosition())
    self:addChild(explosion, 2, GameSceneNodeTag.ExplosionParticleSystem)
    if defaults:getBoolForKey(SOUND_KEY)  then
        AudioEngine.playEffect(sound_2)
    end                                                                  ④

    if enemy.enemyType == EnemyTypes.Enemy_Stone then                    ⑤
        score = EnemyScores.Enemy_Stone + score
        scorePlaceholder =  EnemyScores.Enemy_Stone + scorePlaceholder
    elseif enemy.enemyType ==  EnemyTypes.Enemy_1 then
        score = EnemyScores.Enemy_1 + score
        scorePlaceholder =  EnemyScores.Enemy_1 + scorePlaceholder
    elseif enemy.enemyType ==  EnemyTypes.Enemy_2 then
        score = EnemyScores.Enemy_2 + score
        scorePlaceholder =  EnemyScores.Enemy_2 + scorePlaceholder
    else
        score = EnemyScores.Enemy_Planet + score
        scorePlaceholder =  EnemyScores.Enemy_Planet + scorePlaceholder
    end                                                                  ⑥

    --每次获得 1000 分数，生命值加 1，scorePlaceholder 恢复 0.
    if  scorePlaceholder >= 1000 then                                    ⑦
        fighter.hitPoints = fighter.hitPoints + 1                        ⑧
        self:updateStatusBarFighter()                                    ⑨
        scorePlaceholder = scorePlaceholder - 1000
    end

    self:updateStatusBarScore()                                          ⑩
    --设置敌人消失
    enemy:setVisible(false)
    enemy:spawn()
  end
end
```

上述第①行代码 enemy.hitPoints = enemy.hitPoints-1 是给敌人生命值减 1。第②行代码 enemy.hitPoints <= 0 是判断敌人生命值小于等于 0 情况下,即敌人应该消失、爆炸,并给玩家加分。第③和④行代码是实现敌人被击毁时候爆炸音效和爆炸粒子效果。第⑤和⑥行代码是根据被击毁的敌人类型给玩家加不同的分值。

第⑦~⑨行代码是每次玩家获得 1000 分数,生命值加 1。第⑧行代码是给玩家生命值加 1,第⑨行代码 self:updateStatusBarFighter()更新状态栏中的玩家生命值。第⑩行代码 self:updateStatusBarScore()是更新状态栏中玩家的得分值。

23.7.8 迭代 6.8:玩家飞机与敌人的接触检测

玩家飞机与敌人的接触检测与炮弹与敌人的接触检测类似,都是在接触检测监听事件中实现的。相关代码如下:

```
function GamePlayScene:createLayer()
    ...
    -- 接触检测事件回调函数
    local function onContactBegin(contact)

        local spriteA = contact:getShapeA():getBody():getNode()
        local spriteB = contact:getShapeB():getBody():getNode()

        ------------------ 检测  飞机与敌人的接触 start ----------------------
        if  spriteA:getTag() == GameSceneNodeTag.Fighter
            and spriteB:getTag() == GameSceneNodeTag.Enemy then
            enemy1 = spriteB
        end
        if  spriteA:getTag() == GameSceneNodeTag.Enemy
            and spriteB:getTag() == GameSceneNodeTag.Fighter  then
            enemy1 = spriteA
        end

        if  nil ~= enemy1  then -- 发生接触
            self:handleFighterCollidingWithEnemy(enemy1)
            return false
        end
        ------------------ 检测  飞机与敌人的接触 end ----------------------

        <省略代码: 检测  炮弹与敌人的接触>
        return false
    end

...
    -- 创建一个接触检测事件监听器
    contactListener = cc.EventListenerPhysicsContact:create()
    contactListener:registerScriptHandler(onContactBegin,
                cc.Handler.EVENT_PHYSICS_CONTACT_BEGIN)

    local eventDispatcher = cc.Director:getInstance():getEventDispatcher()
    -- 添加监听器
```

```
    eventDispatcher:addEventListenerWithSceneGraphPriority(touchFighterlistener, fighter)
    eventDispatcher:addEventListenerWithSceneGraphPriority(contactListener, mainLayer)

    ...
end
```

上述代码与炮弹与敌人的接触检测代码类似不再解释了。在发生接触时通过 self:handleFighterCollidingWithEnemy(enemy1)语句处理。

handleFighterCollidingWithEnemy 函数代码如下：

```
--处理玩家与敌人的接触检测
function GamePlayScene:handleFighterCollidingWithEnemy(enemy)

    self:removeChildByTag(GameSceneNodeTag.ExplosionParticleSystem)

    local explosion = cc.ParticleSystemQuad:create("particle/explosion.plist")
    explosion:setPosition(fighter:getPosition())
    self:addChild(explosion, 2, GameSceneNodeTag.ExplosionParticleSystem)
    if  defaults:getBoolForKey(SOUND_KEY)  then
        AudioEngine.playEffect(sound_2)
    end
    --设置敌人消失
    enemy:setVisible(false)                                          ①
    enemy:spawn()                                                    ②

    --设置玩家消失
    fighter.hitPoints = fighter.hitPoints - 1                        ③
    self:updateStatusBarFighter()                                    ④
    --游戏结束
    if fighter.hitPoints <= 0 then                                   ⑤
        cclog("GameOver")
        local GameOverScene = require("GameOverScene")
        local scene = GameOverScene.create(score)                    ⑥
        local tsc = cc.TransitionFade:create(1.0, scene)
        cc.Director:getInstance():pushScene(tsc)
    else
        fighter:setPosition(cc.p(size.width /2, 70))                 ⑦
        local ac1 = cc.Show:create()
        local ac2 = cc.FadeIn:create(1.0)
        local seq = cc.Sequence:create(ac1,ac2)
        fighter:runAction(seq)                                       ⑧
    end
end
```

上述第①行代码 enemy:setVisible(false)是设置敌人不可见。第②行代码 enemy:spawn()重新生成敌人。第③行代码 fighter.hitPoints = fighter.hitPoints-1 是给玩家生命值减 1。第④行代码 self:updateStatusBarFighter()更新状态栏中的玩家生命值。

第⑤行代码是判断游戏是否结束(玩家生命值为 0),在游戏结束时场景切换到游戏结束场景。游戏结束场景切换与其他的场景切换不同,我们需要把当前获得分值传递给游戏

结束场景。

第⑥行代码 GameOverScene. create(score)是创建 GameOverScene 场景对象,其中参数 score 是玩家获得的分数,将该参数传递给 GameOverScene 场景。

第⑦和第⑧行代码是在玩家飞机与敌人精灵接触,但玩家还有生命值情况下执行的。首先重新设置玩家飞机的位置,然后再设置显示、淡入等动作效果。

23.7.9 迭代 6.9: 玩家飞机生命值显示

玩家飞机生命值更新在生命值变化的时候才需要。例如,在玩家飞机与敌人发生接触时,或者玩家每次得分超过 1000 分时,需要玩家飞机更新生命值。更新是通过调用 updateStatusBarFighter 函数实现的。

updateStatusBarFighter 函数代码如下:

```
--在状态栏中设置玩家的生命值
function GamePlayScene:updateStatusBarFighter()

    --先移除上次的精灵
    mainLayer:removeChildByTag(GameSceneNodeTag.StatusBarFighterNode)

    local fg = cc.Sprite:createWithSpriteFrameName("gameplay.life.png")
    fg:setPosition(cc.p(size.width - 60, size.height - 28))
    mainLayer:addChild(fg,20,GameSceneNodeTag.StatusBarFighterNode)

    --添加生命值 x 5
    mainLayer:removeChildByTag(GameSceneNodeTag.StatusBarLifeNode)


    if fighter.hitPoints < 0 then
        fighter.hitPoints = 0
    end
    local life = string.format("x %d", fighter.hitPoints)
    local lblLife = cc.Label:createWithTTF(life, "fonts/hanyi.ttf", 18)
    local fgX,fgY = fg:getPosition()
    lblLife:setPosition(cc.p(fgX + 30, fgY))
    mainLayer:addChild(lblLife, 20, GameSceneNodeTag.StatusBarLifeNode)

end
```

23.7.10 迭代 6.10: 显示玩家得分情况

显示玩家得分更新是在击毁一个敌人后执行的。更新是通过调用 updateStatusBarScore 函数实现的。

updateStatusBarScore 函数代码如下:

```
--在状态栏中显示得分
function GamePlayScene:updateStatusBarScore()
```

```
mainLayer:removeChildByTag(GameSceneNodeTag.StatusBarScore)

if score < 0 then
    score = 0
end

local strScore = string.format("%d", score)
local lblScore = cc.Label:createWithTTF(strScore, "fonts/hanyi.ttf", 18)

lblScore:setPosition(cc.p(size.width / 2, size.height - 28))
    mainLayer:addChild(lblScore, 20, GameSceneNodeTag.StatusBarScore)

end
```

23.8　任务7：游戏结束场景

游戏结束场景是游戏结束后由游戏场景进入的。我们需要在游戏结束场景显示最高记录分。最高记录分被保存在 UserDefault 中。在游戏结束场景实现的时候还要考虑接受前一个场景(游戏场景)传递的分数值。

我们先看看 GameOverScene.lua 头文件，代码如下：

```
require "Cocos2d"
require "Cocos2dConstants"

require "SystemConst"

local size = cc.Director:getInstance():getWinSize()
local defaults = cc.UserDefault:getInstance()
local listener

local GameOverScene = class("GameOverScene",function()
    return cc.Scene:create()
end)

function GameOverScene.create(score)                                    ①
    local scene = GameOverScene.new(score)                              ②
    scene:addChild(scene:createLayer())
    return scene
end

function GameOverScene:ctor(score)                                      ③

    self.score = score                                                  ④

    --场景生命周期事件处理
    local function onNodeEvent(event)
        if event == "enter" then
            self:onEnter()
        elseif event == "enterTransitionFinish" then
            self:onEnterTransitionFinish()
```

```
        elseif event == "exit" then
            self:onExit()
        elseif event == "exitTransitionStart" then
            self:onExitTransitionStart()
        elseif event == "cleanup" then
            self:cleanup()
        end
    end
    self:registerScriptHandler(onNodeEvent)
end

-- 创建层
function GameOverScene:createLayer()                                    ⑤
    cclog("GameOverScene init")
    local layer = cc.Layer:create()

    --添加背景地图.
    local bg = cc.TMXTiledMap:create("map/blue_bg.tmx")
    layer:addChild(bg)

    --放置发光粒子背景
    local ps = cc.ParticleSystemQuad:create("particle/light.plist")
    ps:setPosition(cc.p(size.width/2, size.height/2   - 100))
    layer:addChild(ps)

    local top = cc.Sprite:createWithSpriteFrameName("gameover.page.png")
    --锚点在左下角
    top:setAnchorPoint(cc.p(0,0))
    top:setPosition(cc.p(0, size.height - top:getContentSize().height))
    layer:addChild(top)

    local defaults = cc.UserDefault:getInstance()
    local highScore = defaults:getIntegerForKey(HIGHSCORE_KEY, 0)
    if highScore < self.score then
        highScore = self.score
        defaults:setIntegerForKey(HIGHSCORE_KEY,highScore)
    end
    local text = string.format("%i points", highScore)
    local lblHighScore = cc.Label:createWithTTF("最高分: ", "fonts/hanyi.ttf", 25)

    lblHighScore:setAnchorPoint(cc.p(0,0))
    local topX,topY = top:getPosition()
    lblHighScore:setPosition(cc.p(60 , topY - 30))

    layer:addChild(lblHighScore)

    local lblScore = cc.Label:createWithTTF(text, "fonts/hanyi.ttf", 24)
    lblScore:setColor(cc.c3b(75,255,255))
    lblScore:setAnchorPoint(cc.p(0,0))
    local lblHighScoreX,lblHighScoreY = lblHighScore:getPosition()
    lblScore:setPosition(cc.p(lblHighScoreX, lblHighScoreY - 40))
    layer:addChild(lblScore)

    local text2 = cc.Label:createWithTTF("Tap the Screen to Play", "fonts/hanyi.ttf", 24)
    text2:setAnchorPoint(cc.p(0,0))
    local lblScoreX,lblScoreY = lblScore:getPosition()
```

```
text2:setPosition(cc.p(lblScoreX - 10, lblScoreY - 45))
layer:addChild(text2)

--接触检测事件回调函数
local function touchBegan(touch, event)                                    ⑥
    --播放音效
    if defaults:getBoolForKey(SOUND_KEY)  then
        AudioEngine.playEffect(sound_1)
    end
    cc.Director:getInstance():popScene()                                   ⑦
    return false
    end

    --注册  触摸事件监听器
    listener = cc.EventListenerTouchOneByOne:create()
    listener:setSwallowTouches(true)
    -- EVENT_TOUCH_BEGAN 事件回调函数
    listener:registerScriptHandler(touchBegan,cc.Handler.EVENT_TOUCH_BEGAN)  ⑧

    -- 添加  触摸事件监听器
    local eventDispatcher = cc.Director:getInstance():getEventDispatcher()
    eventDispatcher:addEventListenerWithSceneGraphPriority(listener, layer)  ⑨

    return layer
end

function GameOverScene:onEnter()
    cclog("GameOverScene onEnter")
end

function GameOverScene:onEnterTransitionFinish()
    cclog("GameOverScene onEnterTransitionFinish")
end

function GameOverScene:onExit()
    cclog("GameOverScene onExit")
    if nil ~= listener then
        cc.Director:getInstance():getEventDispatcher():removeEventListener(listener)
    end
end

function GameOverScene:onExitTransitionStart()
    cclog("GameOverScene onExitTransitionStart")
end

function GameOverScene:cleanup()
    cclog("GameOverScene cleanup")
end

return GameOverScene
```

上述第①行代码是定义工厂函数 GameOverScene. create(score)，参数 score 是玩家所得分数。第②行代码是通过 GameOverScene. new(score)语句创建 GameOverScene 场景。

第③行代码 GameOverScene:ctor(score)是构造函数，第④行代码 self. score = score

是为 score 属性赋值。

第⑤行代码 GameOverScene:createLayer()是定义创建 GameOverScene 场景函数。第⑥行代码是定义接触检测事件回调函数,在该函数中第⑦行代码通过 cc. Director:getInstance():popScene()语句返回到上一个场景。第⑧行代码是注册触摸事件 EVENT_TOUCH_BEGAN。第⑨行代码是将事件监听 listener 添加到事件派发器中。

本章小结

本章介绍了完整的游戏项目分析与设计、编程过程,使得广大读者能够了解 Cocos2d-x Lua API 手机游戏开发过程。而且通过本章的学习读者能够将前面介绍的知识串联起来。

第 24 章

把迷失航线游戏发布到 Google play 应用商店

当你阅读到本章的时候，恭喜你已经学习了本书的大部分知识，已经使用 Cocos2d-x Lua 开发和测试了迷失航线手机游戏。接下来应该考虑在各个应用商店发布这款游戏了。

现在有很多应用商店可以发布应用和游戏，我们不可能全部介绍，只介绍两大移动平台官方的应用商店。分别是：谷歌公司的 Google play 应用商店和苹果公司的 App Store 应用商店。

本章我们首先介绍将迷失航线游戏发布到 Google play 应用商店。

24.1 谷歌 Android 应用商店 Google play

Android 应用商店很多，也很混乱，Google play 应用商店是谷歌公司官方应用商店，我们重点介绍 Google play。

Google play 应用商店的前身是 Android Market，开发人员可以利用 Google play 应用商店的后台网站（https://play.google.com/apps/publish/）发布自己的应用，然后用户可通过内置在 Android 设备中的 Google play 应用商店购买下载应用、音乐、杂志、书籍、电影和电视节目。或通过 Google play 应用商店网站（https://play.google.com/）购买下载这些内容，图 24-1 所示是针对用户的 Google play 应用商店网站。

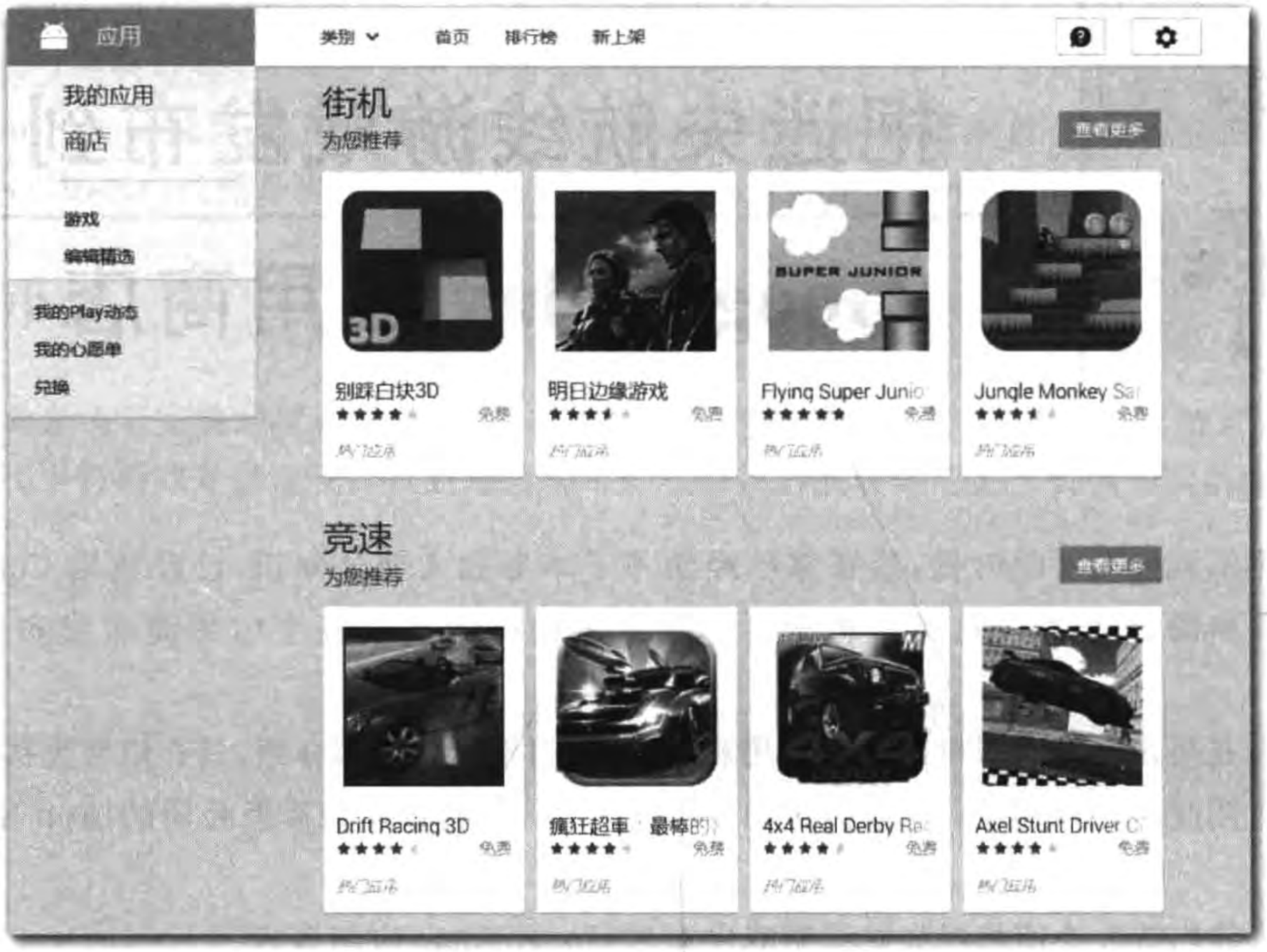


图 24-1 Google Play 应用商店用户网站

24.2 “最后一公里”

设备测试完成后，在发布自己的应用之前，还有“最后一公里”的事情要做，这些事情包括 Lua 文件编译、添加图标和应用程序打包。

24.2.1 Lua 文件编译

为了防止被人看到源代码内容，以及出于新性能的需要，可以使用 cocos luacompile 工具把 Lua 源代码文件编译为 luac 字节码文件。

编译的对象是<游戏工程目录>\src 目录下所有 Lua 文件。具体步骤是通过 DOS 进入到要编译的工程的根目录下，执行命令如下：

```
cocos luacompile -s  <游戏工程全路径>\src\   -d <游戏工程全路径>\src\
cocos luacompile -s  <游戏工程全路径>\src\Sprite\   -d <游戏工程全路径>\src\Sprite\
```

然后通过 DOS 命令或在资源管理器中删除编译之后 src 目录中的.lua 文件。上面指令同样适用于 Mac OS X 的终端中执行。具体细节请参考 20.3.1 节。

24.2.2　添加图标

用户第一眼看到的就是应用的图标。图标是我们的“着装”，给人很好的第一印象非常重要。“着装”应该大方得体，图标设计也是如此，但图标设计已经超出了本书的讨论范围，这里我们只介绍 Android 图标的设计规格以及如何把图标添加到应用中去。

考虑到多种设备适配，我们提供如下几种 Android 应用规格：

(1) 32×32 像素，对应低分辨率，放在 drawable-ldpi 目录下。

(2) 48×48 像素，对应中分辨率，放在 drawable-mdpi 目录下。

(3) 72×72 像素，对应高分辨率，放在 drawable-hdpi 目录下。

(4) 96×96 像素，对应 720p 高清辨率，放在 drawable-xhdpi 目录下。

此外还有高清 1080p 分辨率，文件放到 drawable-xxhdpi 目录等。这些文件命名要统一，本例中统一命名为 icon.png。我们只设计了三个规格的图标，它们的目录结构如下：

```
<LostRoutes 工程目>\frameworks\runtime-src\proj.android\res
├──drawable-hdpi
|       icon.png
├──drawable-ldpi
|       icon.png
└──drawable-mdpi
        icon.png
```

24.2.3　生成数字签名文件

Android 应用程序调试还有发布都需要打包，打包则需要一个数字签名文件。当调试时则有 Android SDK 生成一个用于调试的数字签名文件，开发人员不需要关心。而发布打包则必须要自己创建数字签名文件了。

要生成数字签名文件可以使用 JDK 提供的 keytool 工具，也可以在终端窗口中运行 keytool 命令实现。此外一些基于 Java 的 IDE(如 Eclipse)提供了图形界面工具。本节介绍如何在终端窗口中通过 keytool 工具创建。

在终端窗口中输入如下命令：

```
keytool -genkey -alias android.keystore -keyalg RSA -validity 20000 -keystore android.keystore
```

keytool 文件位于 JDK 的 bin 目录下，要想在任何目录下都可以使用 keytool 命令，则需要将 JDK 的 bin 目录添加到环境变量 PATH 中，否则需要在终端窗口中切换到 JDK 的 bin 目录下。另外，-genkey 是产生秘钥；-alias 是别名，请记住这个别名，后面还要使用；-keyalg 是采用加密方式；-validity 是有效期；-keystore 是数字签名的文件名。

keytool 是一个命令工具，在执行过程中需要询问一些更加详细的信息，如图 24-2 所示。

如果成功则在当前目录下生成一个 android.keystore 文件。

另外，生成的数字证书文件要保管好，记住刚刚设置的别名和密码，以后所有应用都可

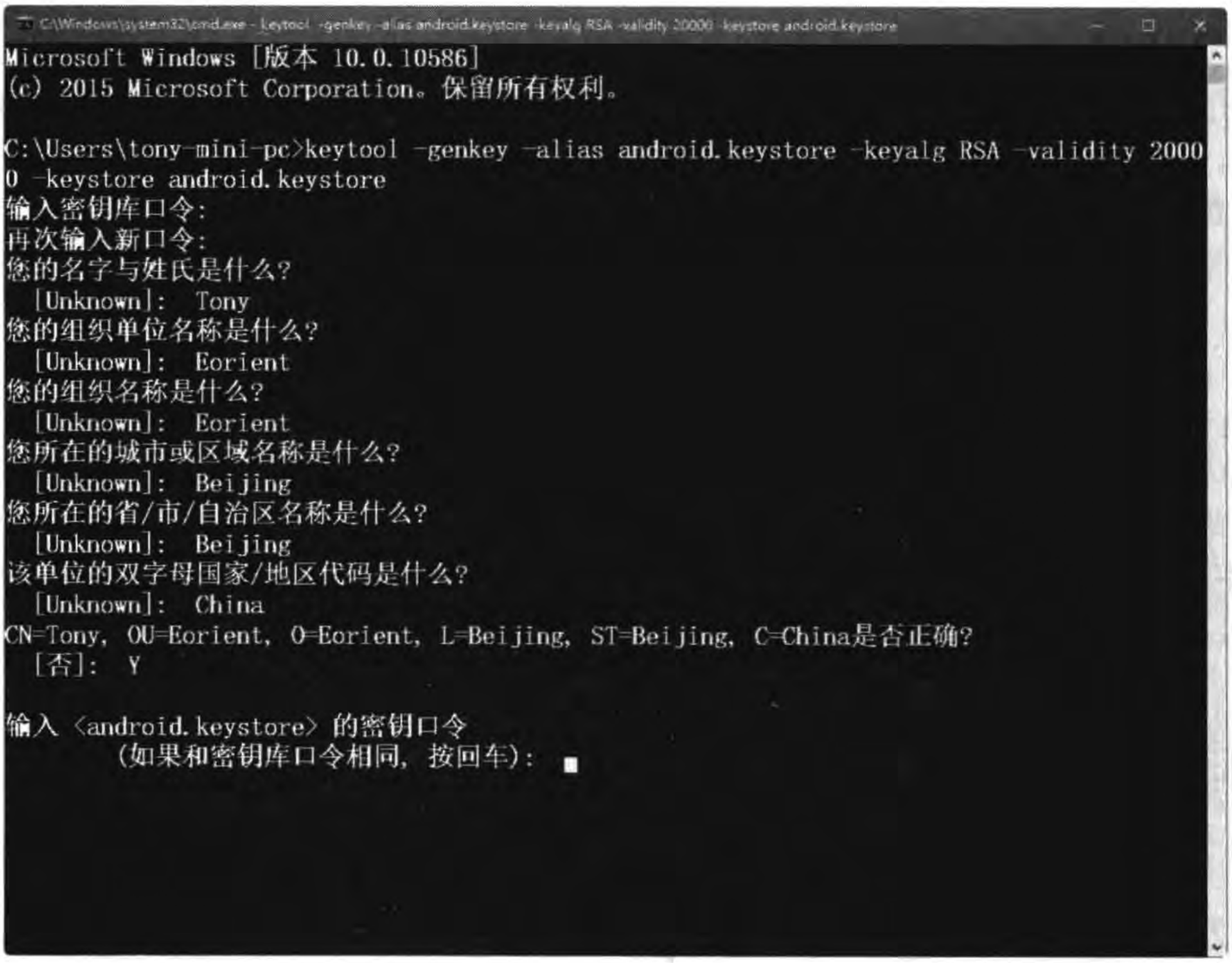


图 24-2 生成数字签名文件

以使用该文件进行数字签名。

24.2.4 应用程序打包

在开发和调试阶段,使用 Cocos2d-x 提供的 cocos 工具编译和运行,也可以使用其为发布打包,但是这种方式比较繁琐,推荐使用 cocos 工具打包。

为发布编译打包的命令如下:

```
cocos compile -p android --m release
```

其中-m 是编译的模式,默认情况下是调试模式,而 release 是发布模式。编译成功之后,就会进行打包,在打包过程中需要指定上一节生成的数字证书文件,如图 24-3 所示需要指定数字证书文件路径,然后回车就开始签名打包。打包成功会在< LostRoutes 目录>\bin\release\android 目录下生成一个 LostRoutes-release-signed.apk 文件。

注意 Cocos2d-x 工程文件路径和数字证书文件路径中都不能有任何的中文。

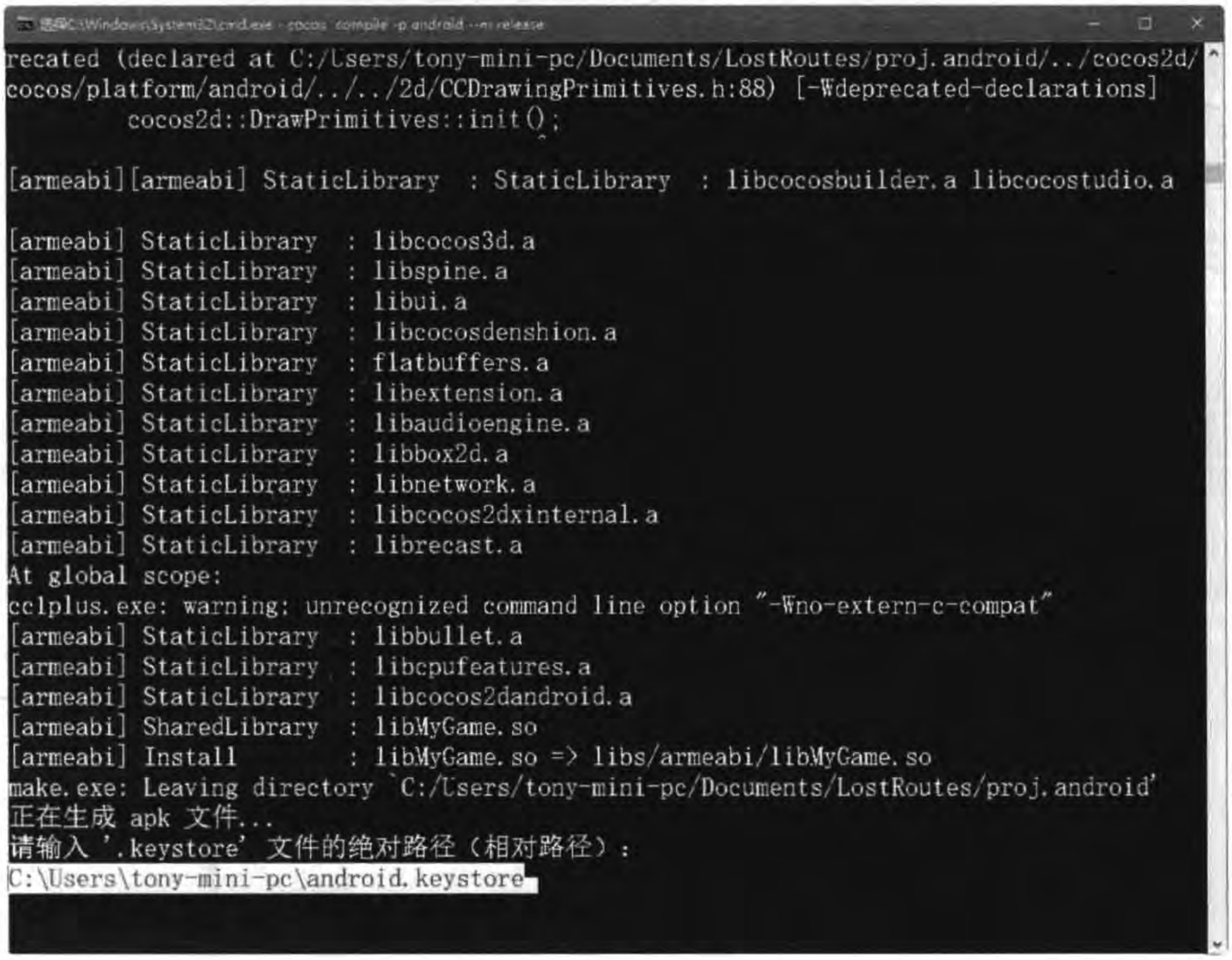


图 24-3　打包文件

24.3　发布产品

程序打包后，就可以发布我们的应用了。发布应用在谷歌提供 https://play.google.com/apps/publish/中完成，发布完成后等待审核，审核通过后就可以到 Google Play Store 上销售了。详细的发布流程分为三个阶段：上传 APK、填写商品详细信息、定价和发布范围。

24.3.1　上传 APK

使用申请的开发者账户登录 https://play.google.com/apps/publish/，会看到图 24-4 所示页面，点击“添加新应用”按钮，弹出图 24-5 所示的对话框，在对话框中选择默认语言和名称，然后单击“上传 APK”按钮，页面会跳转到图 24-6 所示的页面。

在图 24-6 页面中单击“上传您的第一个正式版 APK”按钮，则弹出文件选择对话框，选择刚刚创建的 APK 文件，然后上传就可以了。

图 24-4　登录成功页面

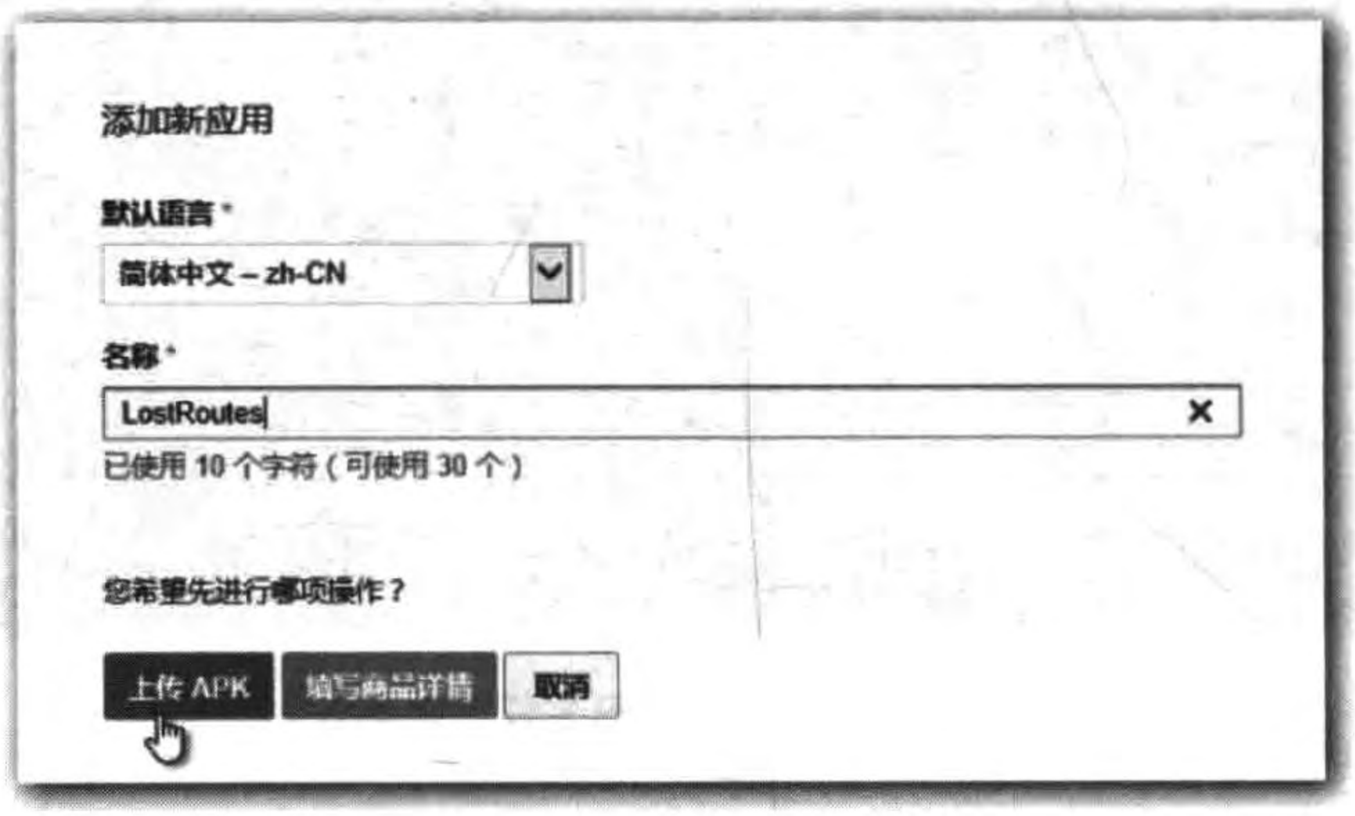


图 24-5　添加新应用对话框

图 24-6　上传 APK 文件

提示　上传成功后会在后台对 APK 文件进行验证，这会与数字签名证书有关，如果证书有问题，或者证书过期，或者证书有效期未到，这些时间导致的证书问题，可能与你制作证书时系统时间不准确有关。你需要在操作系统中选择正确的时区，通过 Internet 同步时间后，重新生成证书，生成 APK 包，再试一试。

24.3.2　填写商品详细信息

上传完成 APK 后，我们可以添加商品详细信息，这里包括文字信息、图片信息和推广视频。单击左边的“商品详情”菜单，打开图 24-7 所示的页面。这些输入项目中有星号标记的是必须输入的，我们可以输入多种不同的语言，通过单击“添加翻译”按钮实现。

图 24-7　商品基本信息

如果将页面向下拖动，我们会看到需要添加的图片和视频资源项目，如图 24-8 所示。从图中可见屏幕截图部分分为手机、7 英寸平板电脑和 10 英寸平板电脑，我们需要按照页面中提示的规格提供真实的应用运行截图，不要有调试信息。在本例中我们的应用只有手机版本，因此平板电脑部分的截图没有提供。屏幕截图部分的下面是高分辨率图标、精选应用图片和宣传图片，高分辨率图标是应用商店显示必须的图标，精选应用图片和宣传图片，

主要用于宣传，这里它们是可选的，我们需要按照页面中提示的规格设计提供图片。在图片的下面还有宣传视频，需要将录制好的视频上传到一些视频网站，然后把视频的网址添加到这里。

图 24-8　商品图片和视频信息

如果将页面向下拖动，在页面的最下面是商品的分类等信息，如图 24-9 所示，请根据实际情况填写相关信息。

图 24-9　商品分类等信息

24.3.3　定价和发布范围

在添加完成商品详细信息后，我们可以添加定价和发布范围，包括商品定价、发布国家和地区、法律协议。单击左边的“定价和发布范围”菜单，打开图 24-10 所示的页面。

“此应用是免费还是付费应用?”中选择“免费”。然后根据我们的销售策略选择发布国家和地区。最下面是三个法律协议，我们应该全部选择，否则就不能发布应用了。

这些信息添加完成后单击页面上面的“保存”，保存添加的信息。如果我们输入的信息没有问题，就可以添加右上角的“可以发布”按钮进行发布了，如图 24-11 所示。如果发布成功，回到管理页面的首页，会看到图 24-12 所示的已发布状态。

上述的流程完成之后，就意味着我们的应用或游戏基本可以在 Google Play 应用商店上线了，谷歌不会再对上传的应用或游戏进行审核了，这一点与苹果和微软不同。但能够在 Google Play 应用商店看到，还需要几个小时的时间。

图 24-10　定价和发布范围

图 24-11　发布应用

图 24-12　发布状态

本章小结

本章介绍了如何在 Google Play 应用商店发布迷失航线游戏，使广大读者了解在 Google Play 应用商店发布应用流程，以及需要注意的问题。

第 25 章

把迷失航线游戏发布到苹果 App Store 应用商店

在前面一章中介绍了把迷失航线游戏发布到 Google play 应用商店。这一章介绍如何把这款游戏发布到苹果的 App Store 应用商店。

25.1 苹果的 App Store

iOS 设备安装应用的渠道有两个：一个是企业开发账户发布的企业内应用；另一个是个人开发者账户在 App Store 发布的应用。除了测试版外，iOS 设备只能够安装企业开发账户发布的企业内应用和个人开发者账户在 App Store 发布的应用，其他的方式是不能发布的。但是针对于“越狱”的 iOS 设备[①]，可以在一些第三方应用商店或论坛安装应用，这种安装方式有很大的隐患。

苹果公司有两个应用商店：App Store 和 Mac App Store，App Store 是为 iOS 设备提供应用软件的，可以单击 OS X 中 iTunes 图标进入，如图 25-1 所示。Mac App Store 是为 OS X 电脑提供应用软件的，可以单击 OS X 中 App Store 图标进入，如图 25-2 所示。

这一章重点介绍 App Store 应用软件发布流程。

---

① iOS 越狱(英语：iOS Jailbreaking)是获取 iOS 设备的 Root 权限的技术手段。iOS 设备的 Root 权限一般是不开放的。由于获得了 Root 权限，在越狱之前无法查看的 iOS 的文件系统也可查看。伴随着越狱，绝大多数情况下也会安装一款名为 Cydia 的软件。Cydia 的安装也被视为已越狱设备的象征。——引自维基百科 http://zh.wikipedia.org/wiki/%E8%B6%8A%E7%8D%84_(iOS)

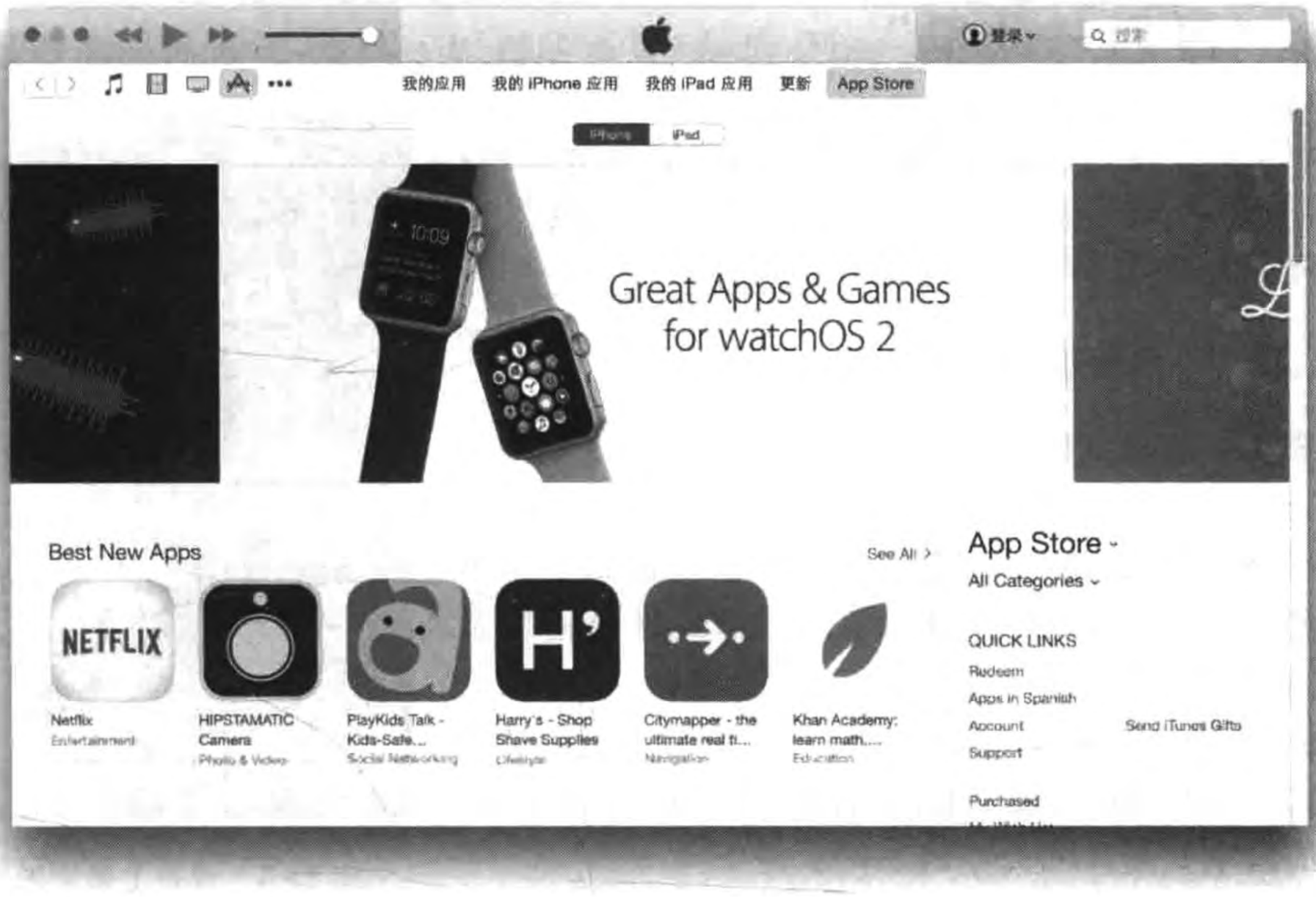


图 25-1　App Store

图 25-2　Mac App Store

25.2 “最后一公里”

iOS应用发布之前还有“最后一公里”的事情要做，这“最后一公里”包括添加图标、添加启动界面和为发布进行编译等工作。

图 25-3　设备上使用的图标

25.2.1　添加图标

用户第一眼看到的就是应用的图标。图标是应用的“着装”，给人很好的第一印象非常重要。“着装”应该大方得体，图标设计也是如此，但图标设计已经超出了本书的讨论范围，这里我们只介绍iOS图标的设计规格以及如何把图标添加到应用中去。

iOS应用使用的图标(App Icon)分为设备上使用的图标(见图25-3)和App Store上使用的图标(见图25-4)，它们之间的区别只是尺寸大小的不同。

此外，iOS上使用的图标还有Spotlight搜索图标、设置图标、工具栏(或导航栏)图标、标签栏图标。这些图标在不同设备上规格不同，而且iOS7与iOS 8和iOS 9在规格方面有比较大的变化，这些图标的规格可以参考苹果的《iOS Human Interface Guidelines》(https://developer.apple.com/library/prerelease/ios/documentation/UserExperience/Conceptual/MobileHIG/IconMatrix.html#//apple_ref/doc/uid/TP40006556-CH27-SW1)。

图 25-4　App Store上使用的图标

这么多规格的图标真是让人很难记住，Xcode 帮我们解决了这一个问题。在 Xcode 有一个新的添加图标方法，在 Xcode 创建通用(包括 iPhone 和 iPad)的工程中，选择打开 Images. xcassets→AppIcon，如图 25-5 所示界面。这个界面中可以很直观地看到各种图标的规格(图标下面的数值代表规格)，例如 29pt，说明规格是 29 点。

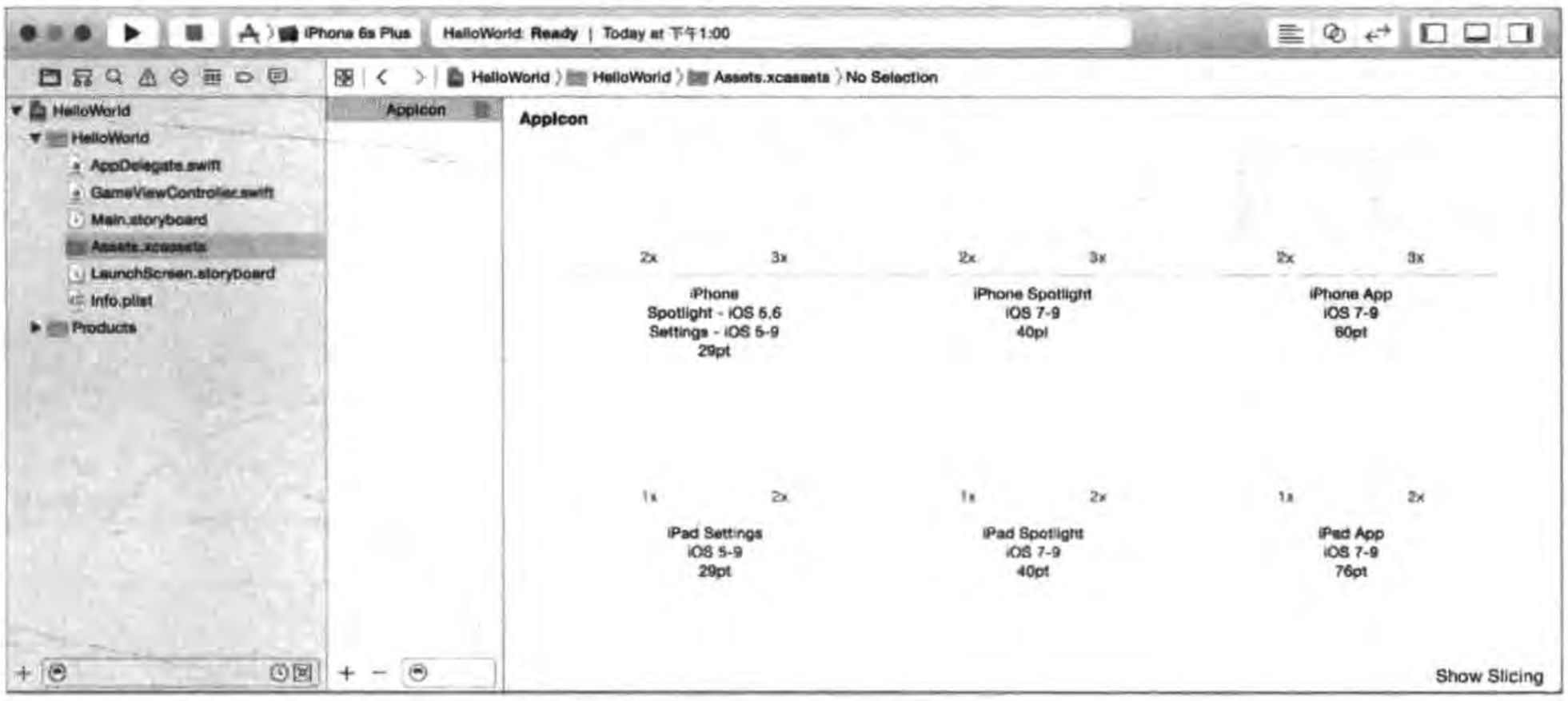


图 25-5　Xcode 中的 AppIcon

提示　点与像素关系是，3x 表示 1 个点 3 倍像素，2x 表示 1 个点 2 倍像素，1x 表示 1 个点 1 倍像素。

图中 3x 是 iPhone 6 Plus 和 iPhone 6s Plus 高分辨率显示屏所用图标，2x 是除 iPhone 6 Plus 和 iPhone 6s Plus 外高分辨率显示屏所用图标，1x 是指普通显示屏所用图标，普通显示屏现在只有 iPad 设备了。

App Store 上图标的添加过程将在下一节介绍，本节主要介绍设备上图标的添加过程，请美术设计师准备好图标，其中最为主要的是 iPhone App 和 iPad App 图标，图 25-6 所示是 LostRoutes(迷失航线)应用所需要图标，由于只是 iPhone 版本，所以只需准备 5 个不同规格的图标即可：58(29×2)像素、87(29×3)像素、80(40×2)像素、120(40×3 和 60×2)像素和 180(60×3)像素。

如果要添加这些图标，首先在 Finder 里打开这些图标所在的文件夹，然后在 Xcode 打开图 25-6 所示的界面，从 Finder 中拖动图标到 Xcode 中对应的位置，如图 25-7 所示。

25.2.2　添加启动界面

启动界面是应用启动与进入到第一个屏幕之间的界面。没有启动界面的应用进入第一个屏幕之前是黑屏，这会影响用户体验。虽然在开发阶段没有什么影响，但是在应用发布前，还是需要添加启动界面的。

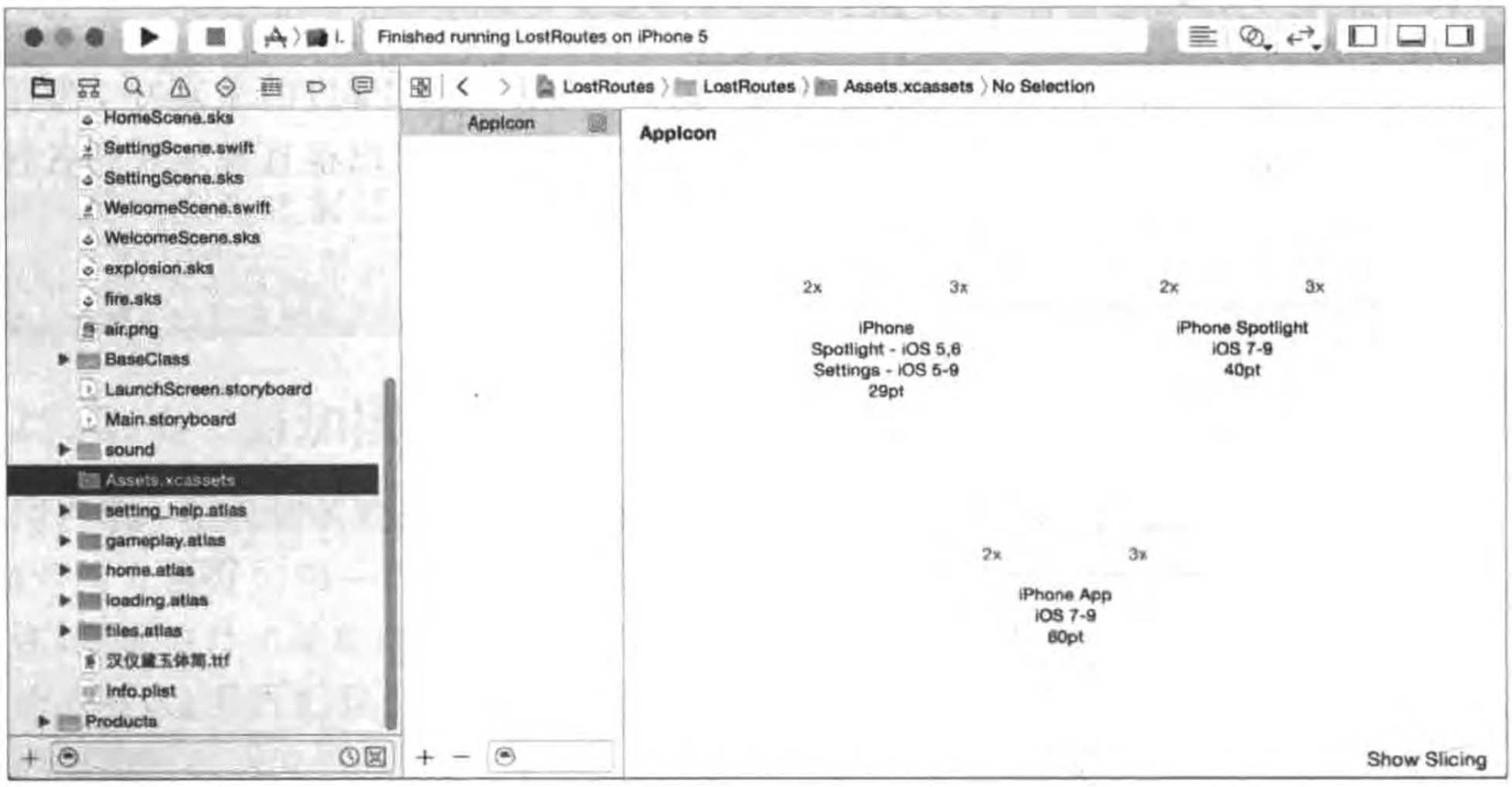


图 25-6　迷失航线中的 AppIcon

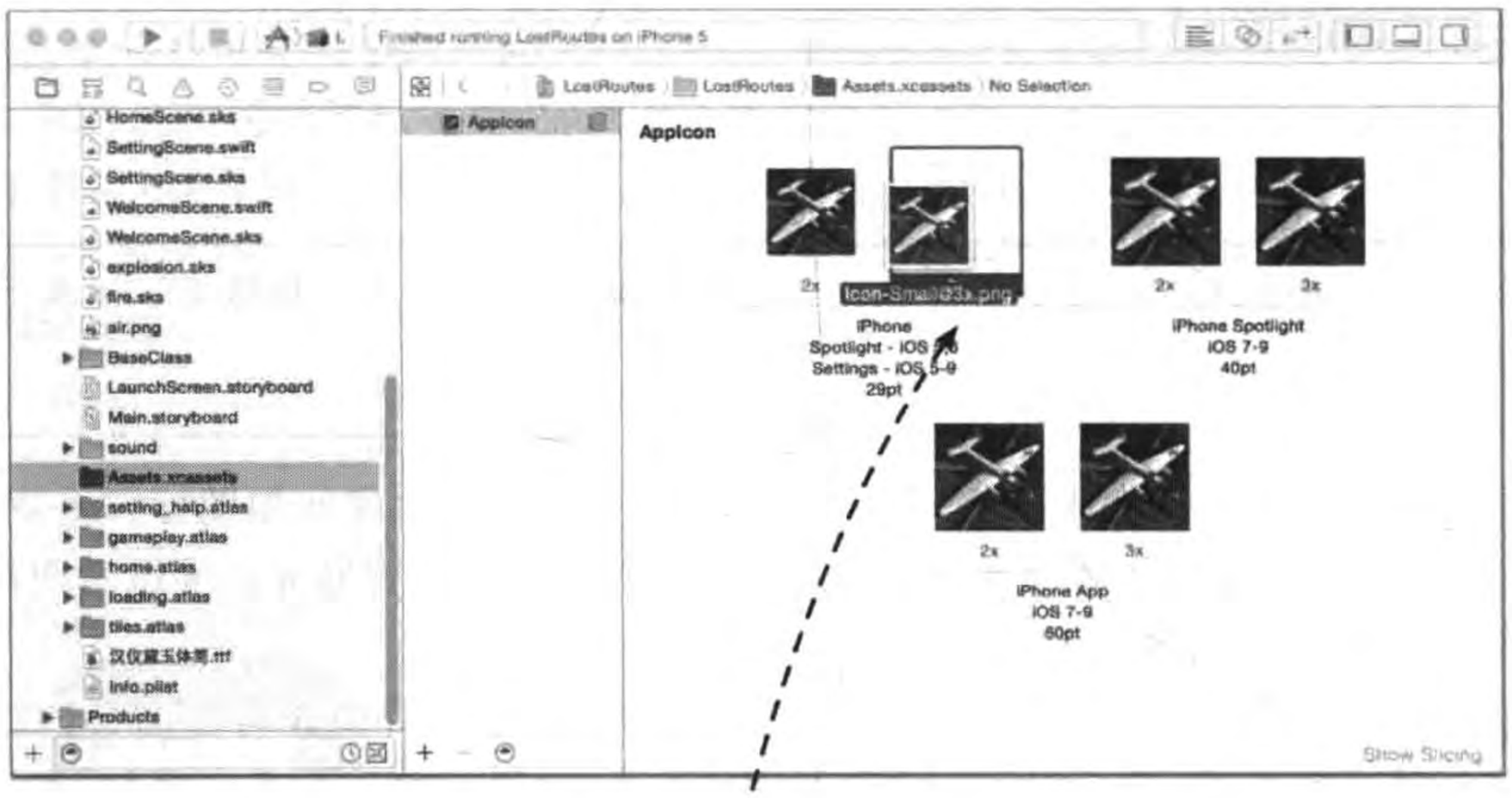


图 25-7　拖动图标到 Xcode

添加启动界面有很多学问，下面先看看两个应用的启动过程，如图 25-8 和图 25-9 所示。

图 25-8　启动界面场景 1

图 25-9　启动界面场景 2

你认为这两个启动界面哪个更好呢？图 25-8 所示的应用进入屏幕④界面时，要经过 3 个界面的变化，界面启动过程中会有大大的 logo 出现，并且设计者还故意让其延迟了几秒钟，让用户看清楚这个 logo。这样做或许很酷，也能宣传自己，但实际上用户既然能够下载你的应用，就肯定知道你是谁了。如果是第一次看，用户可能感觉很新奇，如果每天看会是什么感觉呢？

注意　图 25-9 所示的①界面不是真的已经进入应用了，而是一张图片，它与应用的第一屏幕非常相似，它能够使用户感觉到很快进入了应用，让用户没有感觉到等待，这才是以用户体验为中心的设计。

为了获得更好的用户体验，苹果对于 iOS 上的启动界面推荐采用图 25-9 所示的设计。在 iOS 中，苹果自带的应用都是这样设计的，比如自带的股票应用界面（见图 25-10）。

图 25-10　股票应用启动界面

图 25-10 中的①号图片是启动界面，与第一个屏幕(②号图片)非常类似，就是将动态部分去掉了，它给用户的感觉是应用快速启动，几秒钟后，内容就会填充到屏幕中。

Xcode 6 之后有两种方法添加启动界面：启动图片和启动文件。我们重点介绍启动图片方式。

启动图片方式就是整个的启动界面是一张图片。在 Xcode 中添加启动界面与添加图标非常类似，打开由 Xcode 创建的通用(包括 iPhone 和 iPad)工程，选择打开 Images.xcassets，如图 25-11 所示界面，右击并从快捷菜单中选择 App Icons & Launch Images→New iOS Launch Image 菜单项。

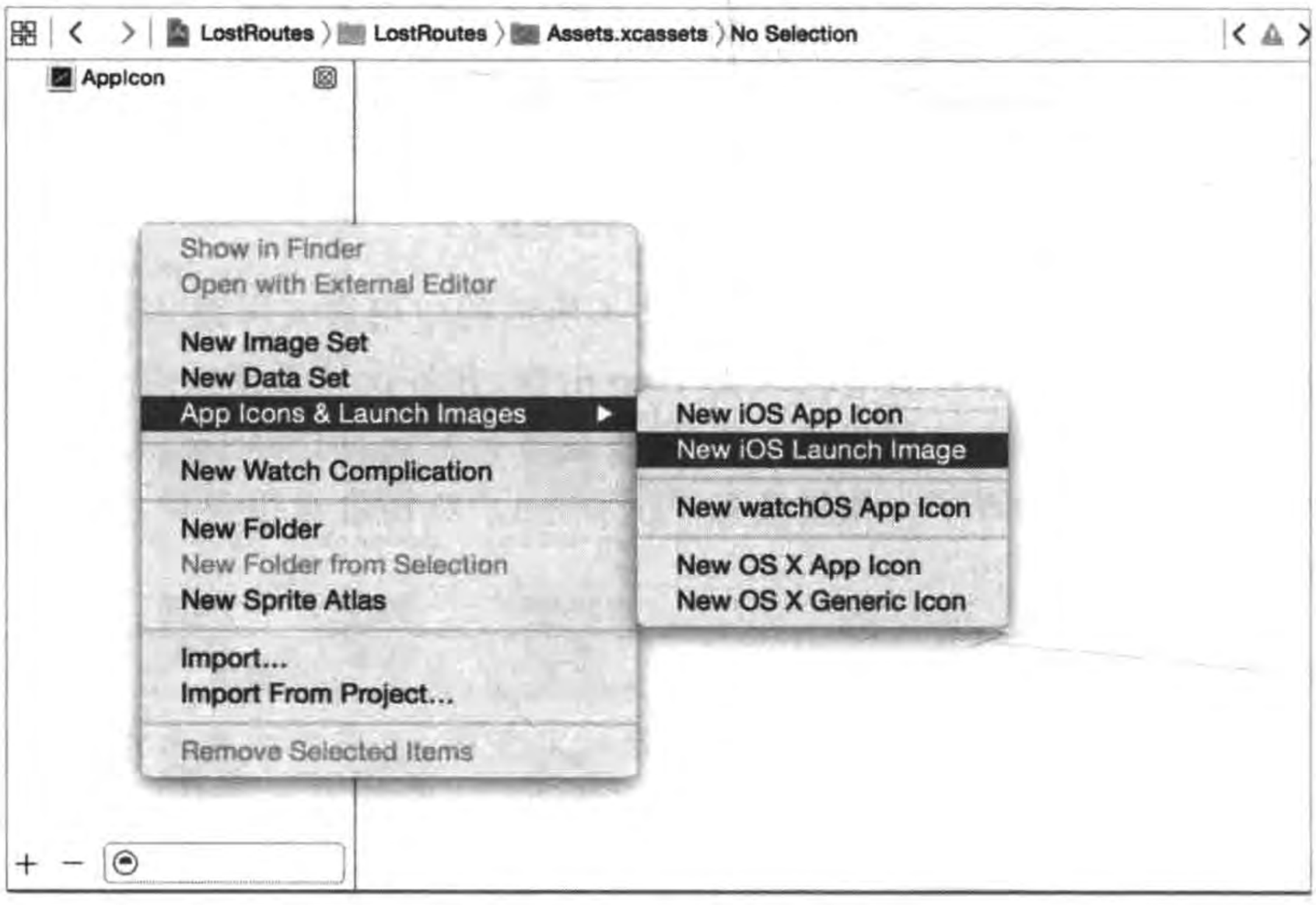


图 25-11　Xcode 中添加 LaunchImage

创建 Launch Image 之后界面如图 25-12 所示，这个界面中我们可以很直观地看到有 20 种不同规格的启动界面。

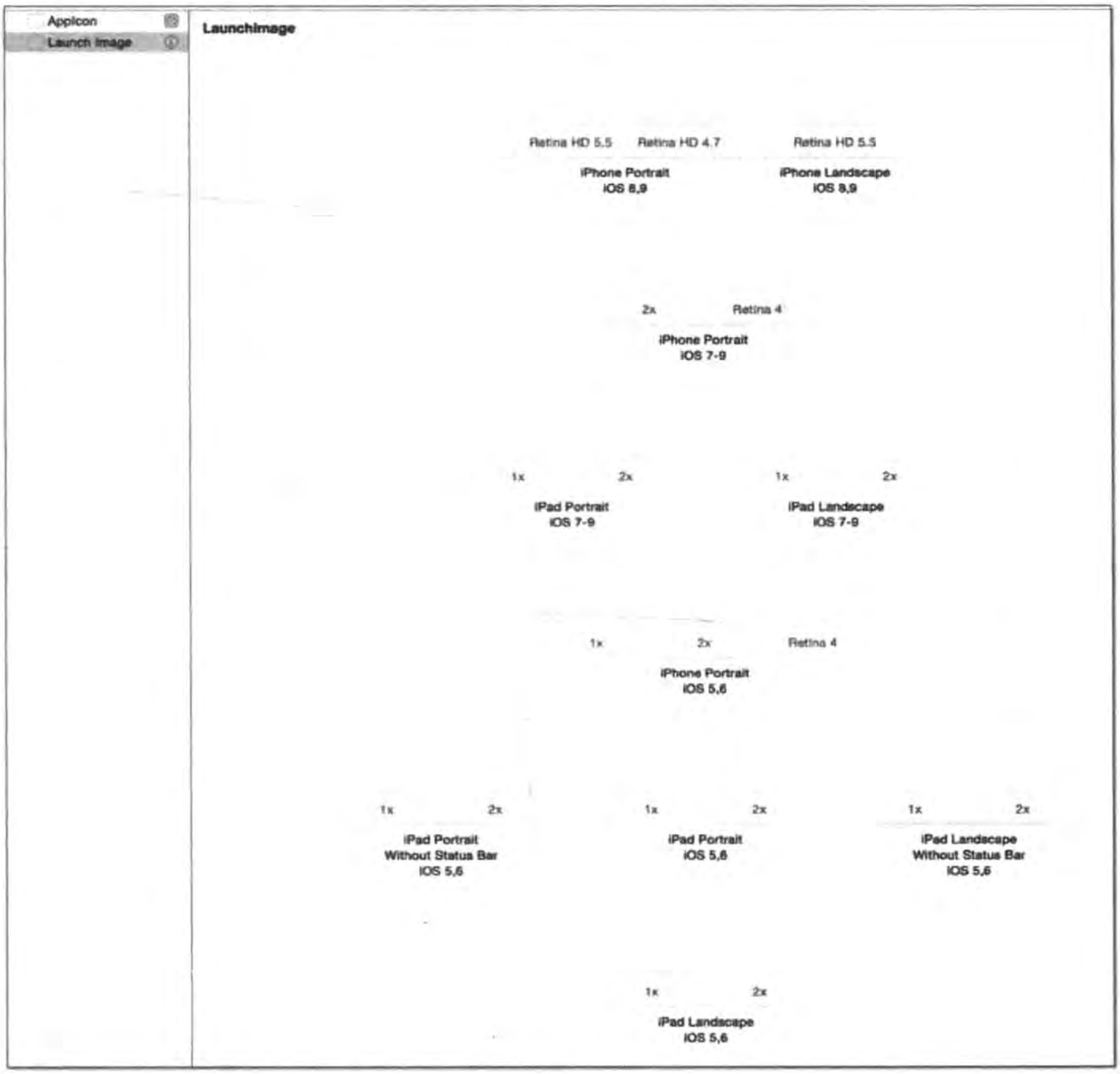


图 25-12　Xcode 中的 Launch Image

请美术设计师为我们准备好启动图片，然后按照上一节类似的方式，将这些图片拖曳到 Xcode 中。然后选中 TARGETS 中 LostRoutes→General→App icons and Launch Images，如图 25-13 所示，将 Launch Screen File 中的内容删除，然后单击 Launch Image Source 后面的 Use Asset Catalog 按钮，在弹出的下拉框中选择 Launch Image。

设置完成之后，可以选择不同的模拟器测试一下了。

25.2.3　修改发布产品属性

在编程过程中，有些产品的属性并不影响开发，即便这些属性设置不正确，一般也不会有什么影响。但是在产品发布时，正确地设置这些属性就很重要了，如果设置不正确，就会

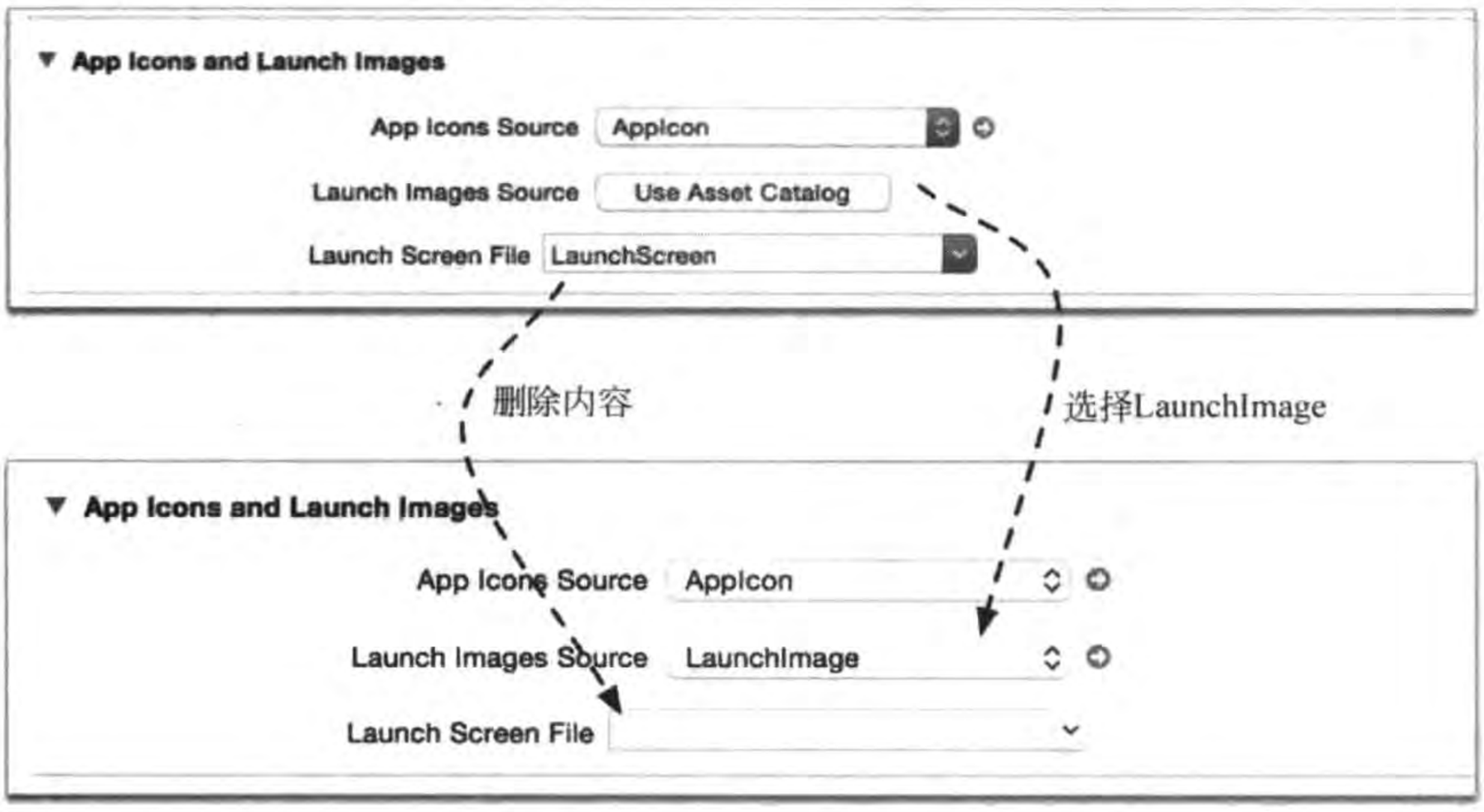


图 25-13　设置启动界面

影响产品的发布。这些产品属性主要是 TARGETS 中的 Identity 和 Deployment Info 属性,如图 25-14 所示。

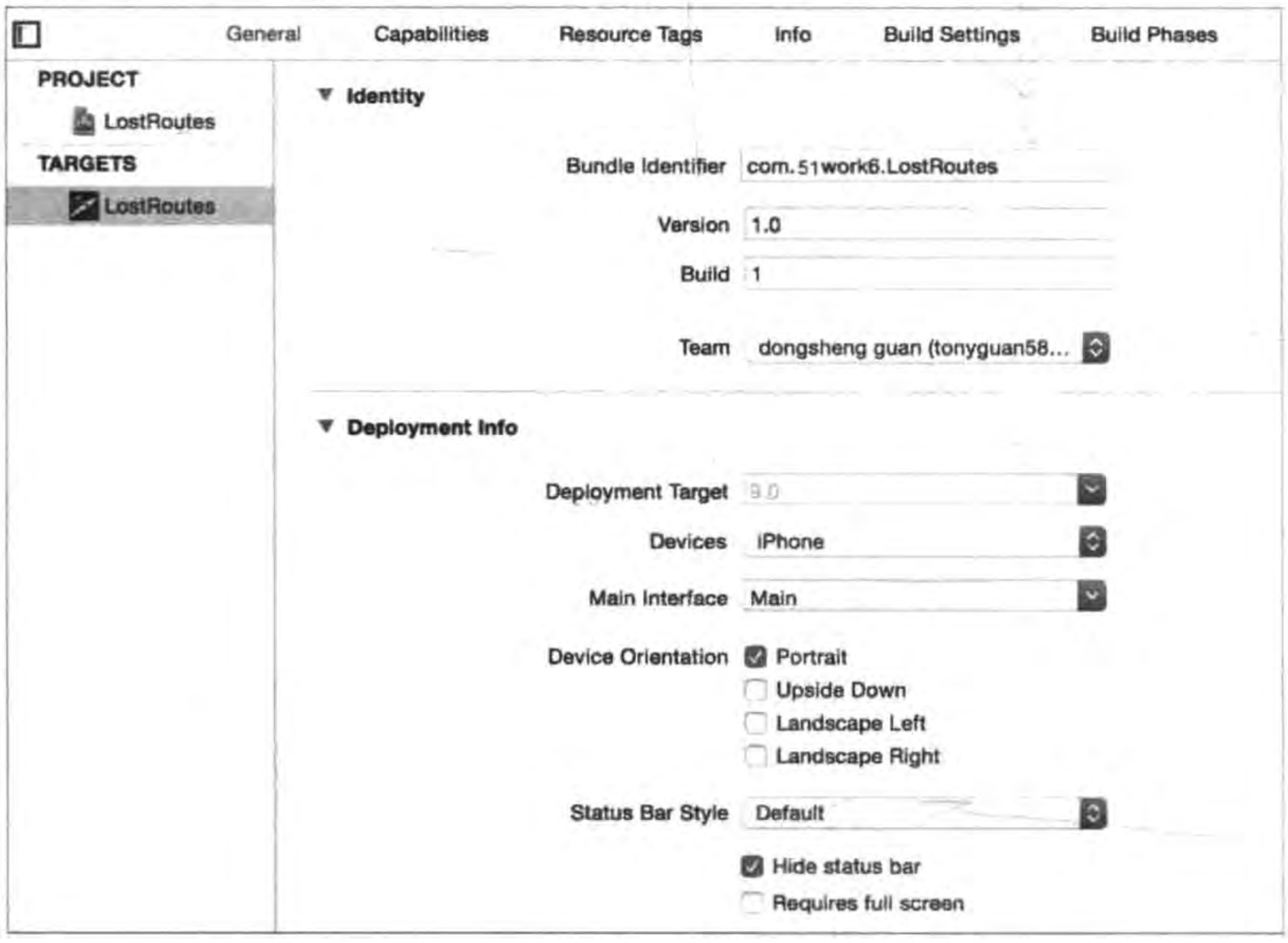


图 25-14　Identity 和 Deployment Info 属性

在这些属性中,Identity 部分主要包括 Bundle identifier(包标识符)、Version(发布版本)、Build(编译版本)和 Team(开发者账号)。在 Deployment Info 主要是 Deployment Target(部署目标)。下面分别介绍它们的含义和重要性。

(1) Bundle identifier(包标识符)。包标识符在开发过程中对我们似乎没有什么影响，但是在发布时非常重要。本例中我们设置的是 com.51work6.LostRoutes。

(2) Version(发布版本)。这个版本号看起来无关紧要，但是在发布时如果这里设定的版本号与 iTunes Connect 中设置的应用的版本号不一致，在打包上传时就会失败。

(3) Build(编译版本)。是编译时设定的版本号。

(4) Team(开发者账号)。测试时这里可以是普通 Apple ID，如果是 App Store 发布，这里必须是开发者账号 Apple ID。

(5) Deployment Target(部署目标)。选择部署目标是开发应用之前就要考虑的问题，这关系到应用所能够支持的操作系统。如果考虑到支持老版本的操作系统(如低于 5.0 版本)，考虑到支持 64 位 ARM CPU 至少需要 iOS 7.0，本例中我们设置为 9.0。

25.3　iOS 设备测试

有些应用只有在真机(设备)上才能运行，所以在应用发布之前一定要在 iOS 真机上测试，本节介绍一下真机(设备)调试所需要的配置过程。在 Xcode 7 之前一个应用要想在 iOS 真机(设备)上运行，需要苹果开发者账号，还需要对设备进行注册、创建开发证书和创建描述文件。Xcode 7 之后可以使用普通 Apple ID(非付费开发者 Apple ID)连接 iOS 真机(设备)运行和调试。

下面介绍一下真机(设备)调试流程，这个流程分为两个步骤：Xcode 设置和设备设置。

25.3.1　Xcode 设置

首先需要在 Xcode 设置账号，这个账号原本是付费的开发者 Apple ID，而现在只需要一个普通的 Apple ID 就可以了。具体步骤是打开 Xcode 使用偏好 Xcode→Preferences，选择 Accounts 标签，如图 25-15 所示单击左下角的“+”账号和密码输入对话框，输入完成之后单击 Add 按钮，如果账号和密码匹配，则添加成功，界面如图 25-16 所示。

接下来就可以对应用进行签名了，这个过程就是通过设置的 Apple ID 生成一个特殊的数字证书，这个证书对应当前这个应用进行签名，这样应用就可以编译了。如图 25-17 所示，选择 TARGETS→LostRoutes→Ceneral→Identity，在 Team 的下拉列表中选择你设置的账号。

当然很多情况下并非那么顺利，如果是第一次运行应用，会有图 25-18 所示的警告，这个警告提示没有该应用的描述文件(provisioning profiles)，描述文件是一个证书签名文件。需要单击“Fix issue”按钮修复该问题，这个过程中可能还会有一些其他的问题，请按照提示修复它们就可以了。成功之后的界面如图 25-19 所示。

最后，可以选择设备进行编译了。

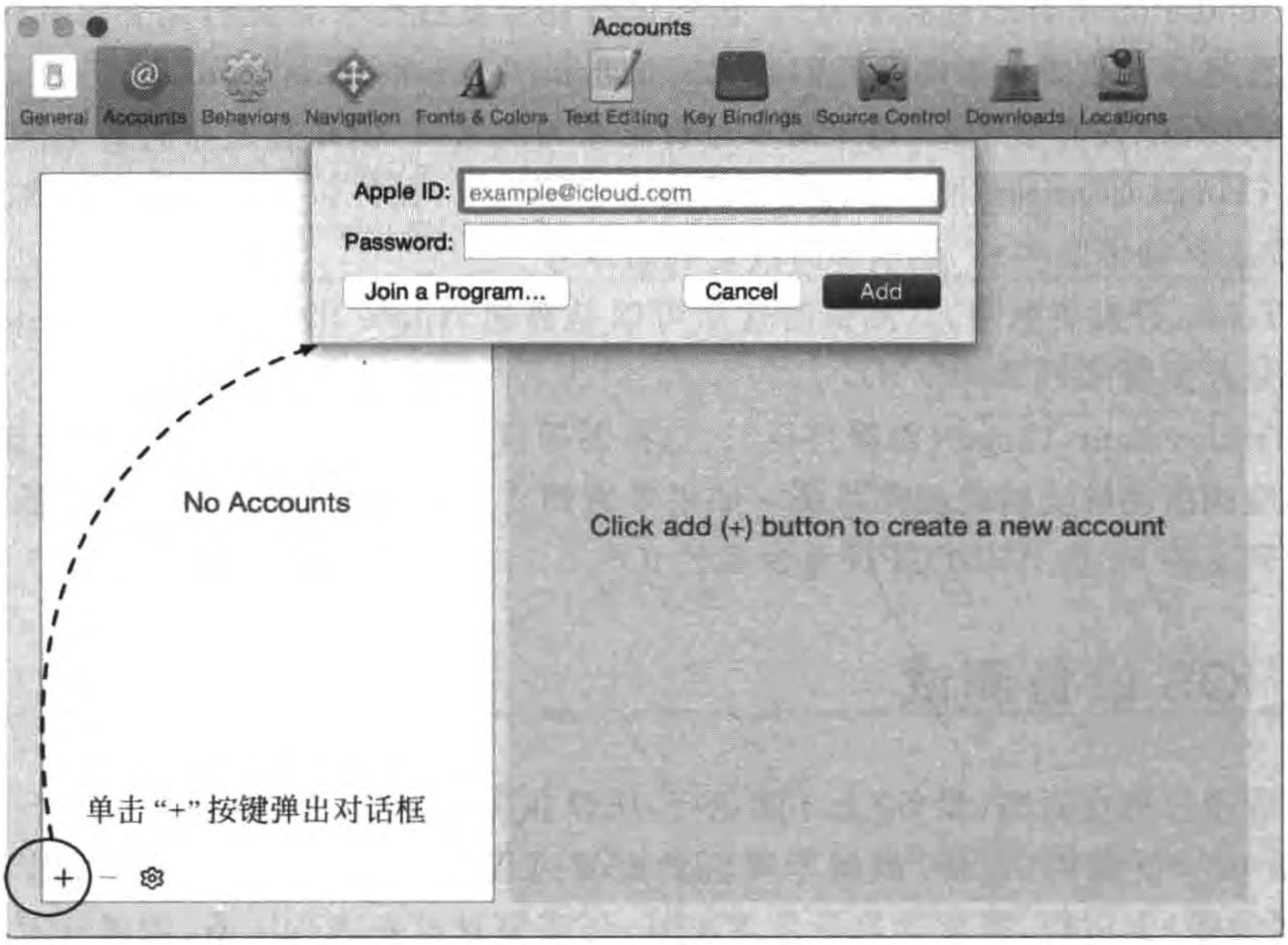


图 25-15　账号和密码输入对话框

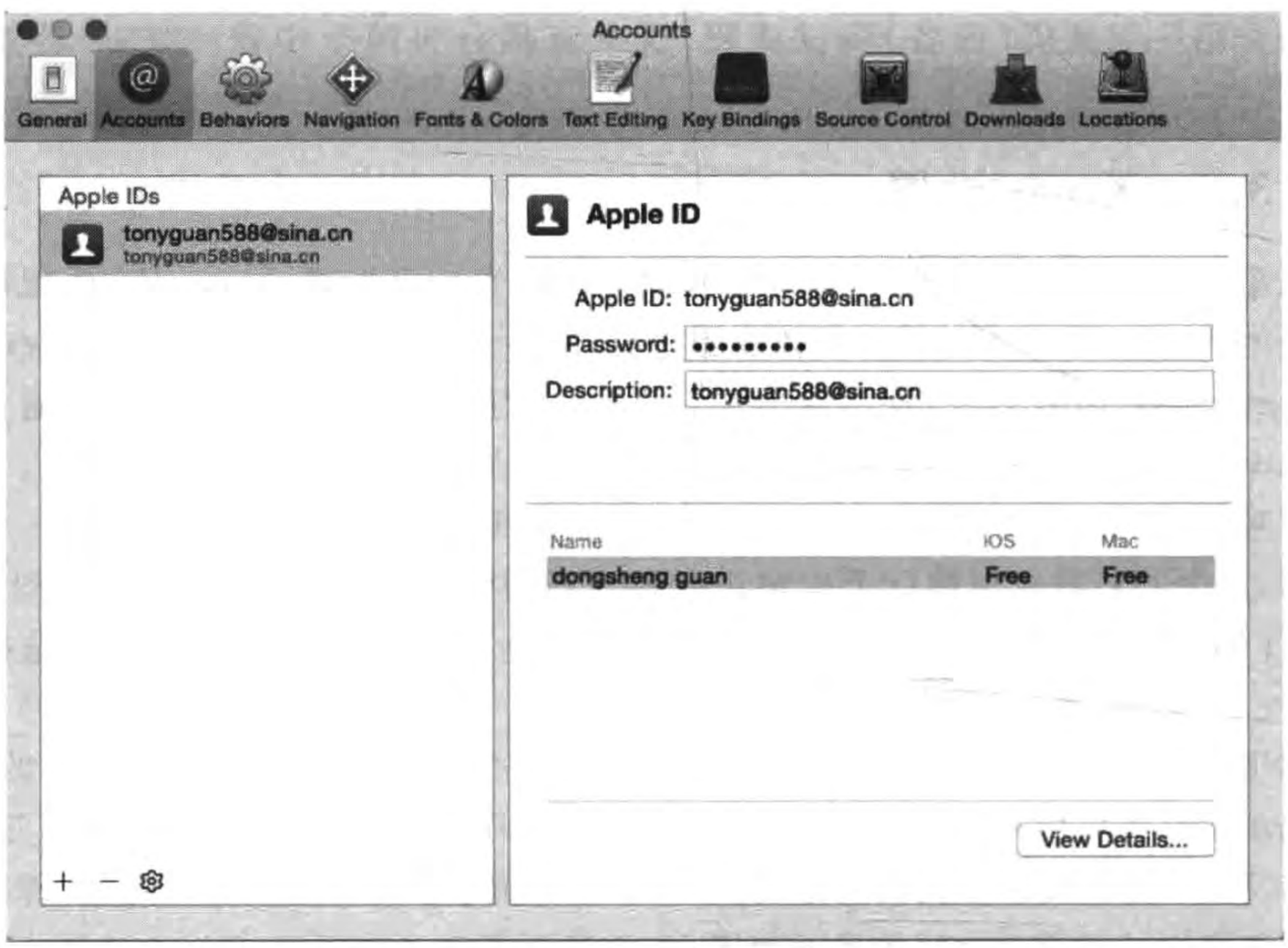


图 25-16　添加完成

图 25-17　签名应用

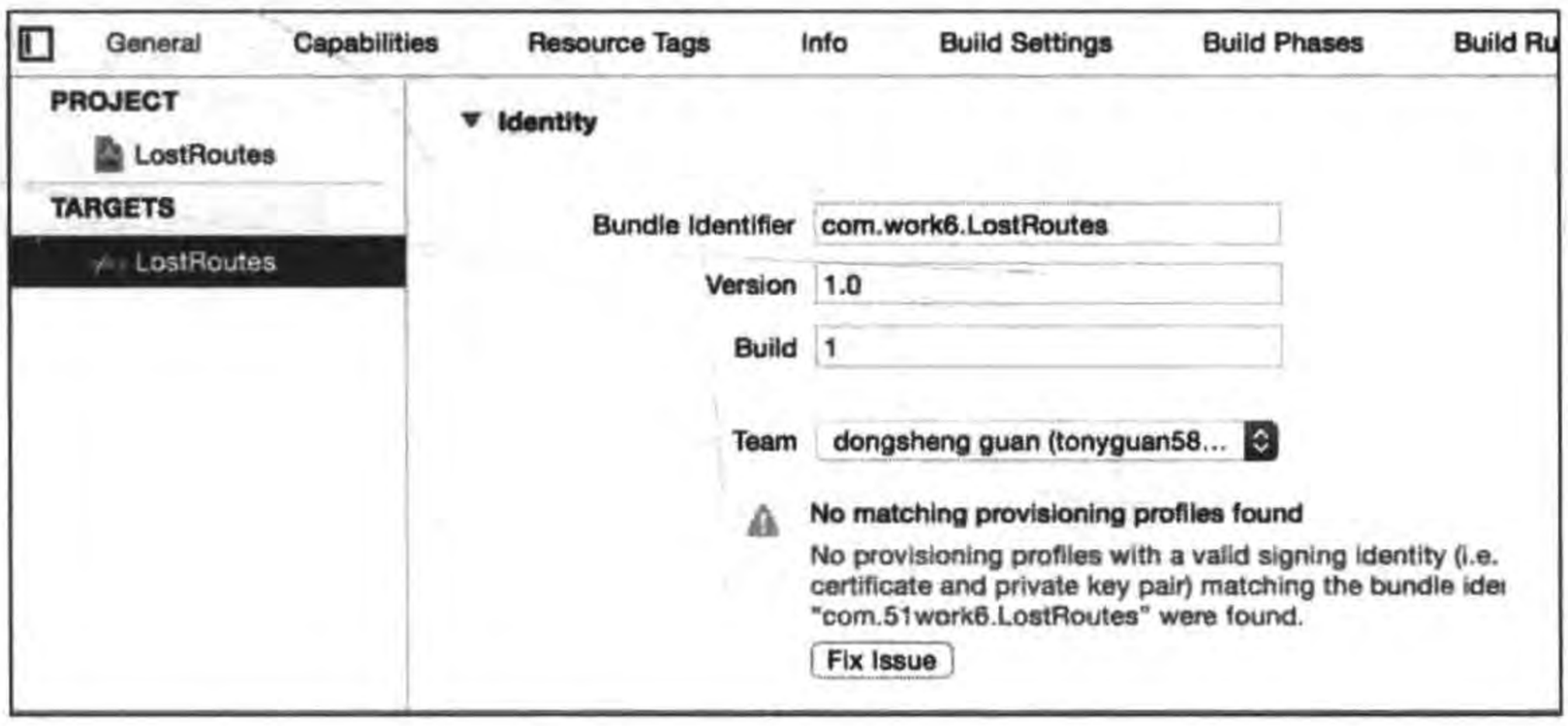


图 25-18　没有描述文件

图 25-19　成功之后的界面

25.3.2　设备设置

上述的设置选择设备进行编译没有问题，但如果运行会出现图 25-20 所示的错误提示。我们会发现该应用已经安装到 iOS 设备上了，但是单击运行会出现图 25-21 所示的不信任提示，接下来需要在 iOS 设备上做一些设置，来信任这个应用。

图 25-21　iOS 不信任提示

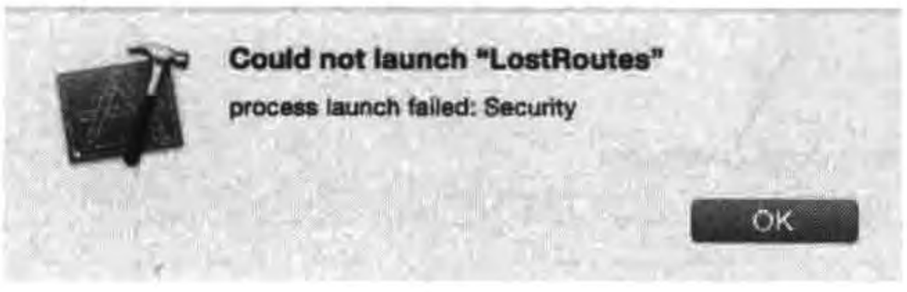


图 25-20　Xcode 错误提示

打开 iOS 的设置，如图 25-22 所示，选择"通用"→"描述文件"，在开发商应用中选择你的账号，单击信任"xxx"按钮，弹出图 25-23 所示的对话框，然后单击"信任"按钮，最后再回到桌面运行刚才的应用就可以运行了。

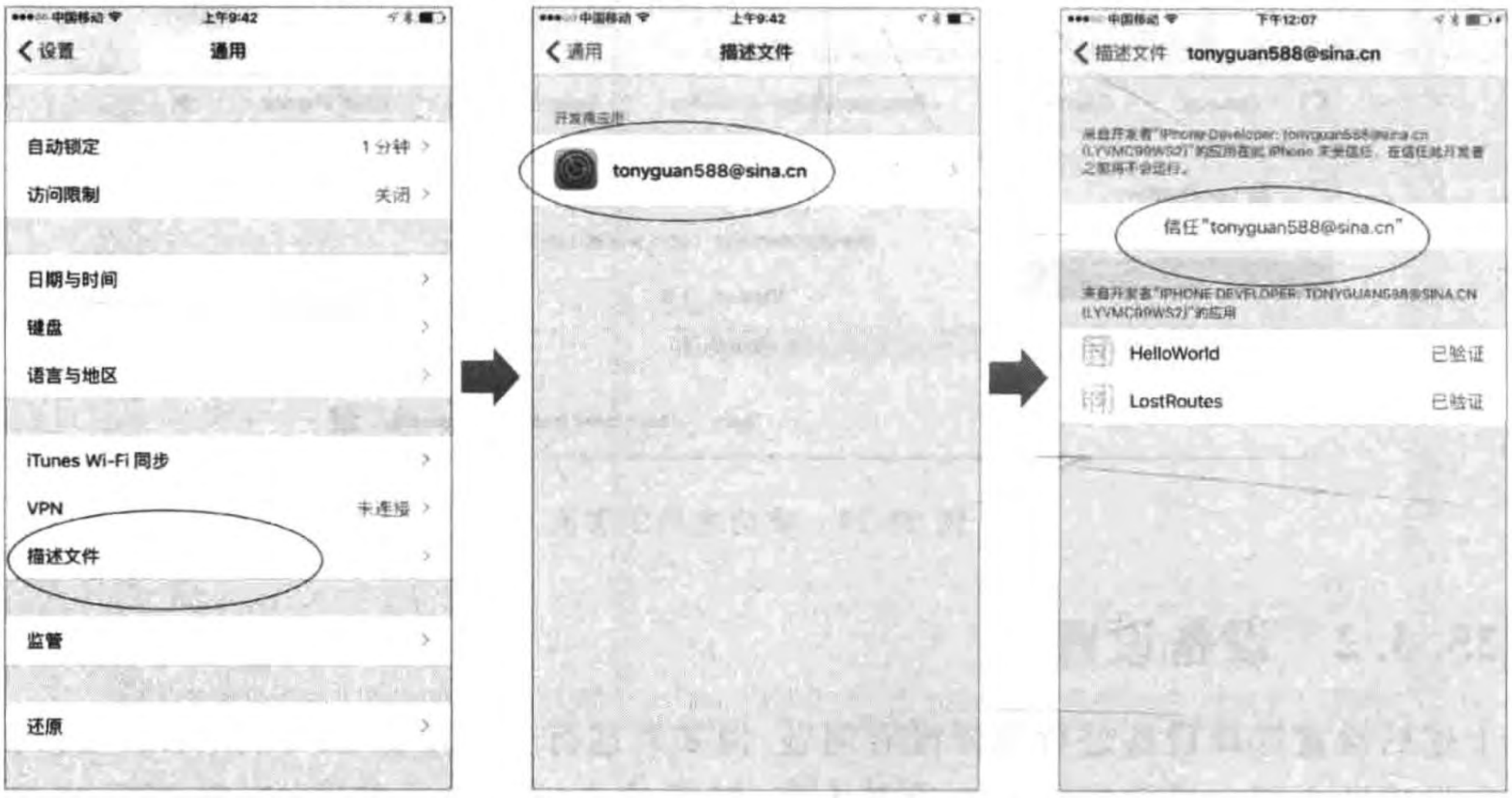


图 25-22　设置信任

图 25-23　设置信任对话框

25.4　为发布进行编译

从编写到发布应用会经历 3 个阶段：在模拟器上运行调试、在设备上运行调试和发布编译。发布编译需要创建开发者证书、创建 App ID、创建描述文件和发布编译。完整的编译发布流程如图 25-24 所示。

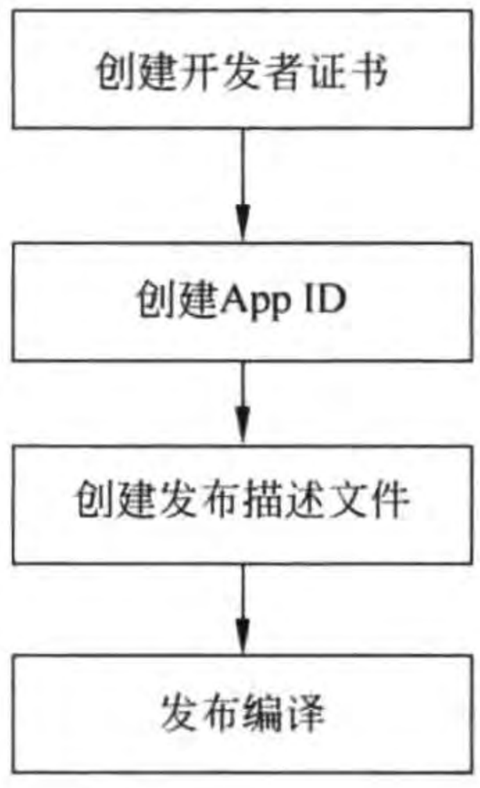


图 25-24　发布编译流程

25.4.1　创建开发者证书

要想在 iOS 设备上调试应用程序，必须具有开发者证书。每个开发人员一次仅允许使用一个开发者证书。证书的管理需要登录 iOS 开发者网站（网址为 https://developer.apple.com/ios/manage/overview/index.action）。登录该网站时，需要苹果的 iOS 开发者账号，登录成功后的界面如图 25-25 所示。

单击 iOS Apps 下的 Certificates（证书）导航菜单，得到的证书管理界面如图 25-26 所示，在此处下载证书和删除证书。

图 25-25　登录 iOS 配置门户网站

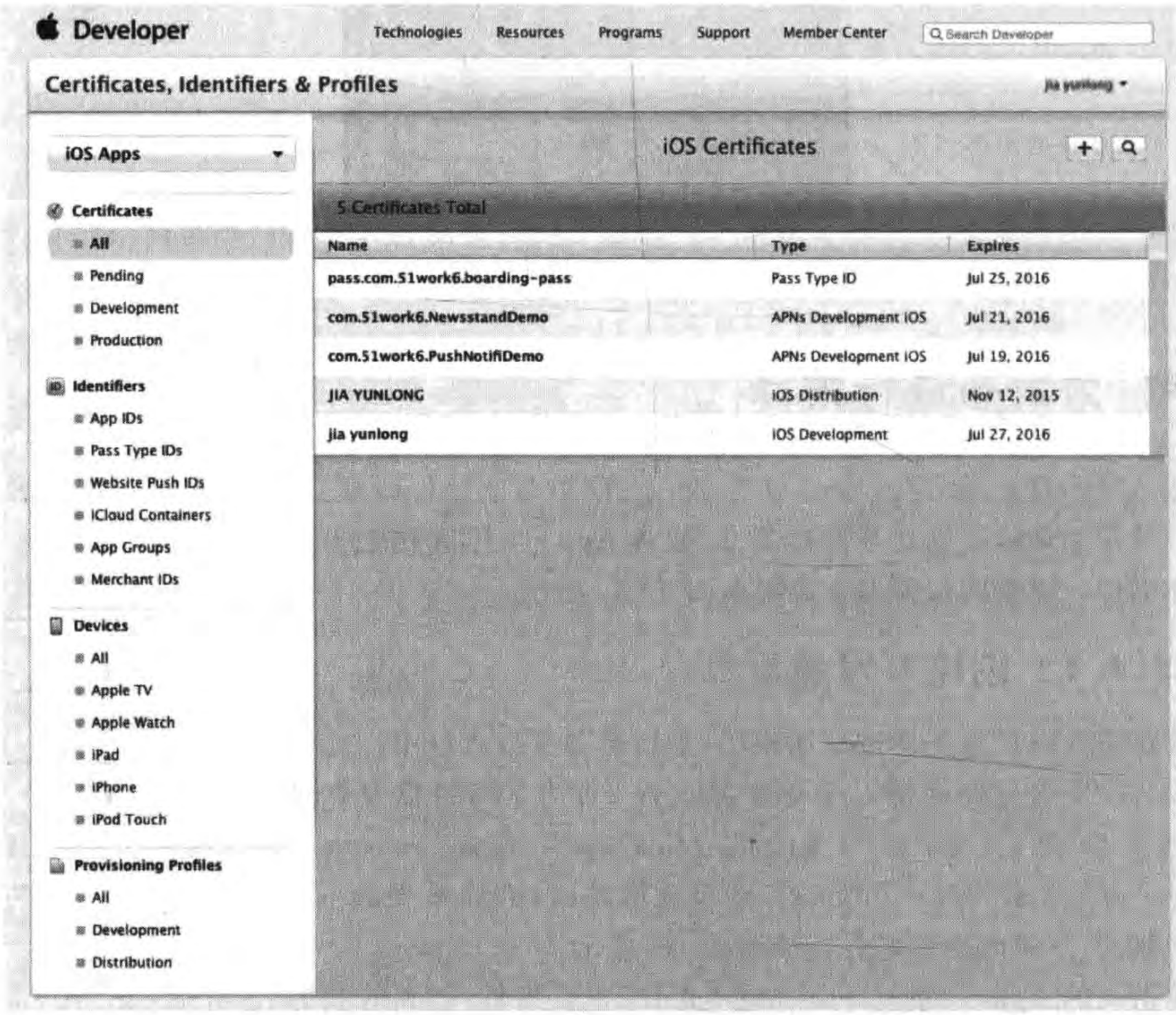


图 25-26　证书管理界面

创建证书过程分成两步：

(1) 钥匙串访问工具请求生成证书。

(2) 提交请求到开发者网站并生成证书。

1. 生成证书签名公钥

在安装有 Mac OS X 操作系统的苹果电脑中打开“应用程序”→“实用工具”→“钥匙串访问”，得到的界面如图 25-27 所示。

图 25-27　钥匙串访问工具

选择“钥匙串访问”→“证书助理”→“从证书颁发机构请求证书”菜单项，此时弹出的对话框如图 25-28 所示，在“用户电子邮件地址”中输入 eorient@sina.com，在“常用名称”中输入“eorient”，然后在“请求是”中选择“存储到磁盘”单选按钮。

在图 25-28 所示的页面中输入信息后，单击“继续”按钮，会弹出图 25-29 所示的证书签名请求文件存储对话框，在这里可以修改文件名和存储位置。

如果默认不修改，则点击“存储”按钮存储文件，此时会在桌面上生成 CertificateSigningRequest.certSigningReques 文件。

2. 提交请求到开发者网站并生成证书

生成 CertificateSigningRequest.certSigningRequest 文件后，重新回到开发者网站提交证书请求文件。单击图 25-26 所示页面右上角的添加按钮 ✚，打开图 25-30 所示的证书类型选择页面，在这个页面中可以选择需要创建的证书。

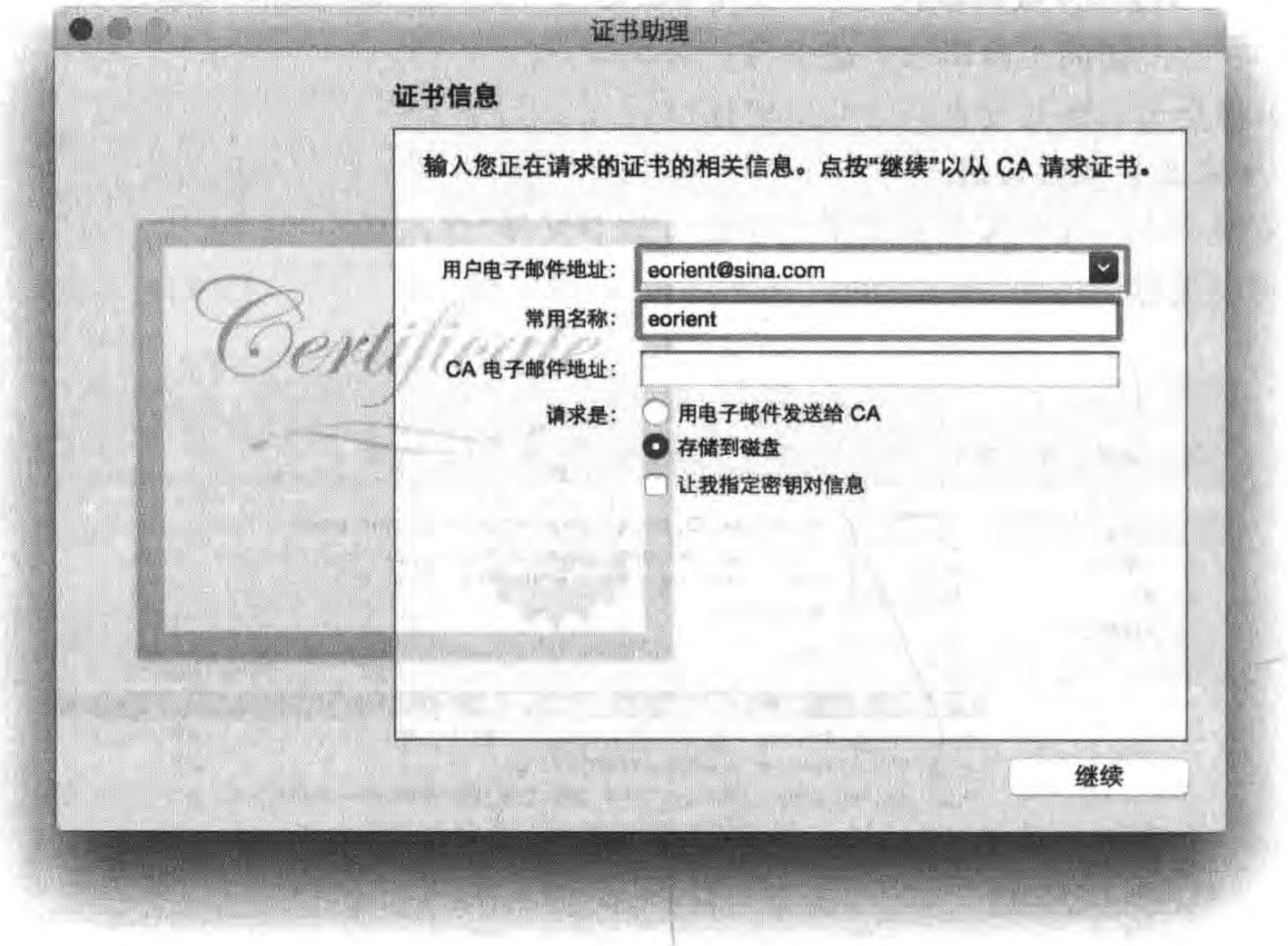


图 25-28　证书助理信息

图 25-29　证书签名请求文件存储对话框

在图 25-30 所示的证书类型选择页面中,有很多概念需要解释一下:

(1) Development。是给开发阶段使用的。

(2) Production。是给发布和团队测试阶段使用的。

(3) iOS App Development。为测试一般的应用使用的。

(4) App Store and Ad Hoc。为在 App Store 或 Ad Hoc 发布使用的,其中 Ad Hoc 也是为团队测试而使用的,允许应用安装到最多 100 个 iOS 设备上,这样可以通过 E-mail 或网站发布将要测试的应用分发给团队其他成员测试。

(5) Apple Push Notification service SSL (Sandbox)。给有推送通知应用测试使用的。

(6) Apple Push Notification service SSL (Production)。给有推送通知应用发布使用的。

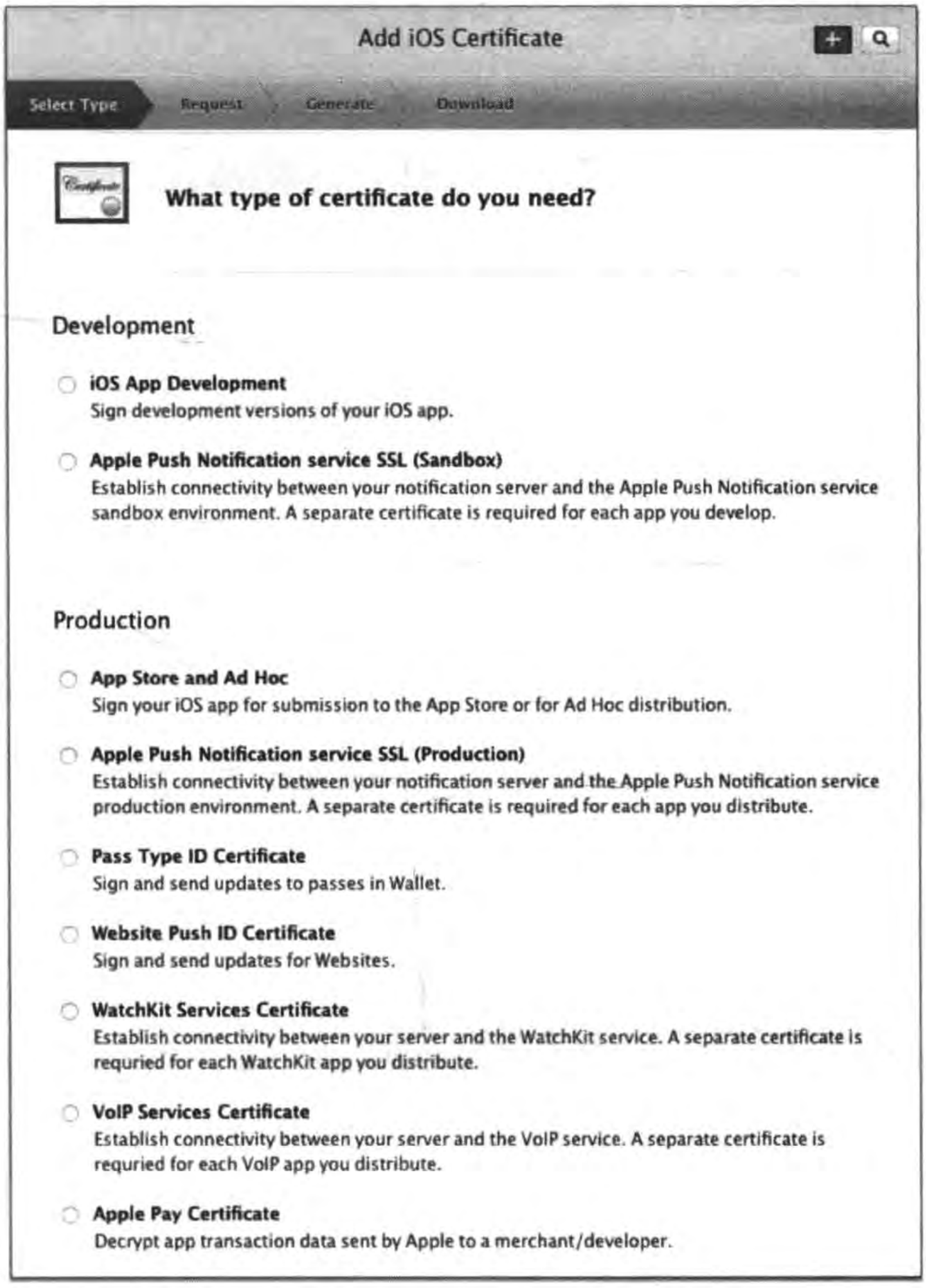


图 25-30　证书类型选择页面

(7) Pass Type ID Certificate。为 PassBook 中的 Pass 使用的。

(8) Website Push ID Certificate。为 Website 使用的。

(9) WatchKit Services Certificate。为 Apple Watch 使用的。

(10) VoIP Services Certificate。为网络电话使用的。

(11) Apple Pay Certificate。为 Apple Pay 应用电话使用的。

我们应该选择 App Store and Ad Hoc 类型，下面的 Continue 按钮就可用了，单击 Continue 按钮进入证书签名请求文件介绍页面，再单击证书请求文件进入图 25-31 所示的证书签名请求文件的上传页面。在页面的最下面找到“Choose File”按钮，选取桌面上的 CertificateSigningRequest. certSigningRequest 文件，然后单击 Generate 按钮就可以生成证

书了,生成后的页面如图 25-32 所示。

图 25-31　提交证书签名请求文件

图 25-32　生成证书成功

在这个页面可以下载证书文件用于发布编译。

25.4.2　创建 App ID

设备注册成功后，还需要为应用创建 App ID，该过程也是在开发者网站完成的。单击左边的 Identifiers→App ID 导航菜单，打开图 25-33 所示页面。

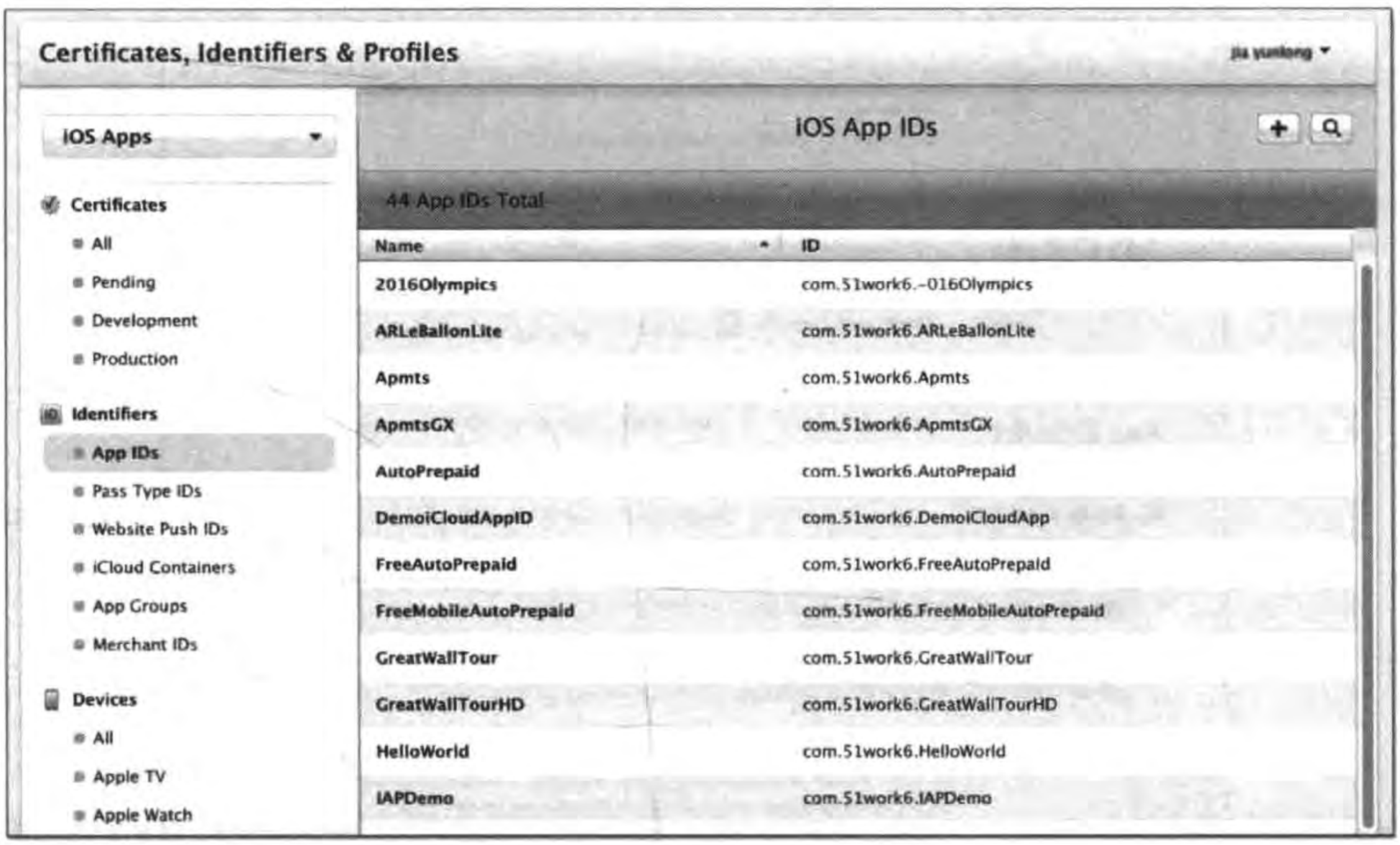


图 25-33　创建 App ID

单击页面右上角的添加按钮 ＋，会打开图 25-34 所示的页面，下面简要介绍一下。

(1) App ID Description。描述，可以输入一些描述应用的信息。

(2) App ID Prefix。应用包种子 ID，它作为应用的前缀，所描述的应用共享了相同的公钥。

(3) App ID Suffix→Explicit App ID。适用于单个应用的后缀，苹果推荐使用域名反写。本例中输入的是 com. work6. LostRoutes，与图 25-35 所示的应用程序 TARGETS 中设定的包标识符保持一致就可以了。

(4) App ID Suffix→Wildcard App ID。适用于多个应用的后缀，苹果推荐使用域名反写。

在图 25-34 所示的界面中完成相应的信息输入后，单击 Continue 按钮提交信息，此时会跳转到创建 App ID 页面。在下面的 App ID 列表中，会发现刚刚创建的 App ID，如图 25-36 所示，单击 Submit 按钮确定创建。

Registering an App ID

The App ID string contains two parts separated by a period (.)—an App ID Prefix that is defined as your Team ID by default and an App ID Suffix that is defined as a Bundle ID search string. Each part of an App ID has different and important uses for your app. Learn More

App ID Description

Name: LostRoutes

You cannot use special characters such as @, &, *, ', "

App ID Prefix

Value: 98Z3R5XU29 (Team ID)

App ID Suffix

● **Explicit App ID**

If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.

To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID: com.work6.LostRoutes

We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

○ **Wildcard App ID**

This allows you to use a single App ID to match multiple apps. To create a wildcard App ID, enter an asterisk (*) as the last digit in the Bundle ID field.

Bundle ID:

Example: com.domainname.*

图 25-34　创建 App ID 的详细页面

General　Capabilities　Resource Tags　Info　Build Settings　Build Phases

PROJECT
LostRoutes
TARGETS
LostRoutes

▼ Identity

Bundle Identifier　com.work6.LostRoutes
Version　1.0
Build　1
Team　JIA YUNLONG (jiayunlongdorg...

图 25-35　应用程序 TARGETS 中设定的包标识符

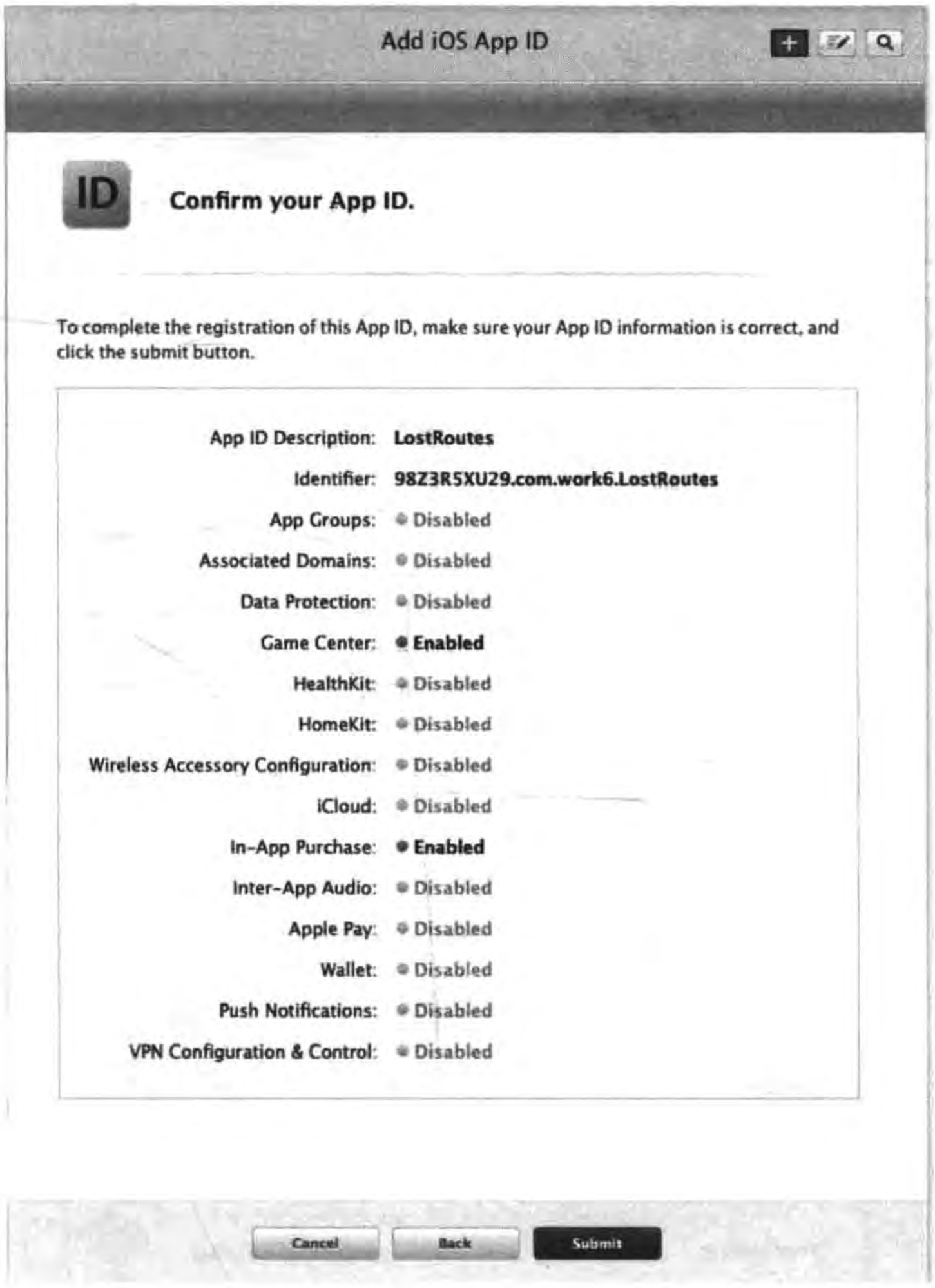
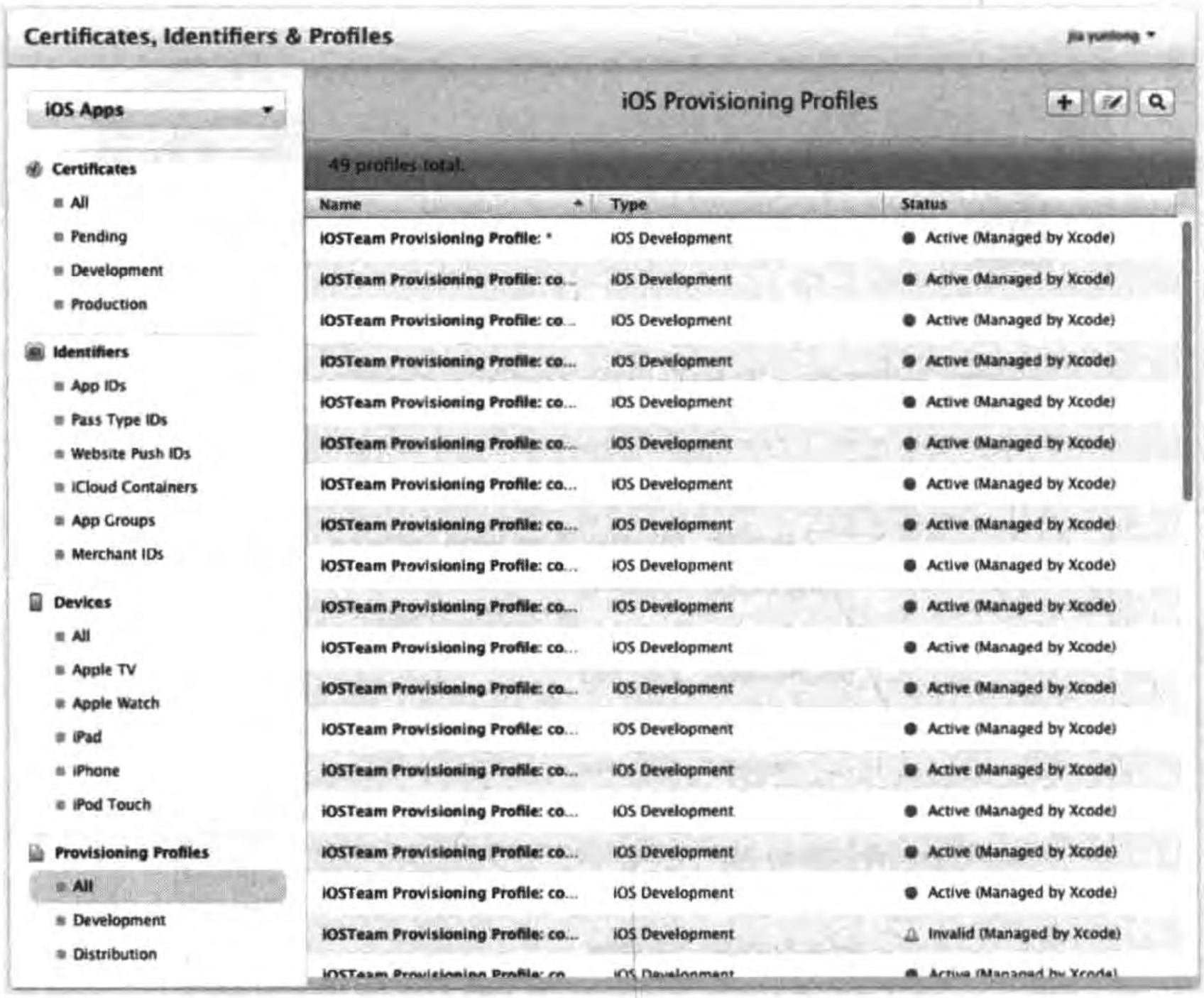


图 25-36　创建完成的 App ID

25.4.3　创建描述文件

描述文件(Provisioning Profiles)是应用在设备上编译时使用的，分为开发描述文件和发布描述文件，分别用于开发和发布。管理描述文件的页面如图 25-37 所示，通过左边的 Provisioning Profiles 导航菜单进入，其中 Development 标签用于管理开发描述文件，Distribution 标签用于管理发布描述文件。本节简要介绍创建开发描述文件的过程，发布描述文件的创建与此类似。

在图 25-37 所示的页面中，单击页面右上角的添加按钮 ，进入创建描述文件选择页面(见图 25-38)，选择 Distribution 中 App Store 类型，单击下面 Continue 按钮进入图 25-39 所示页面，这个页面中选择前面创建好的 App ID。

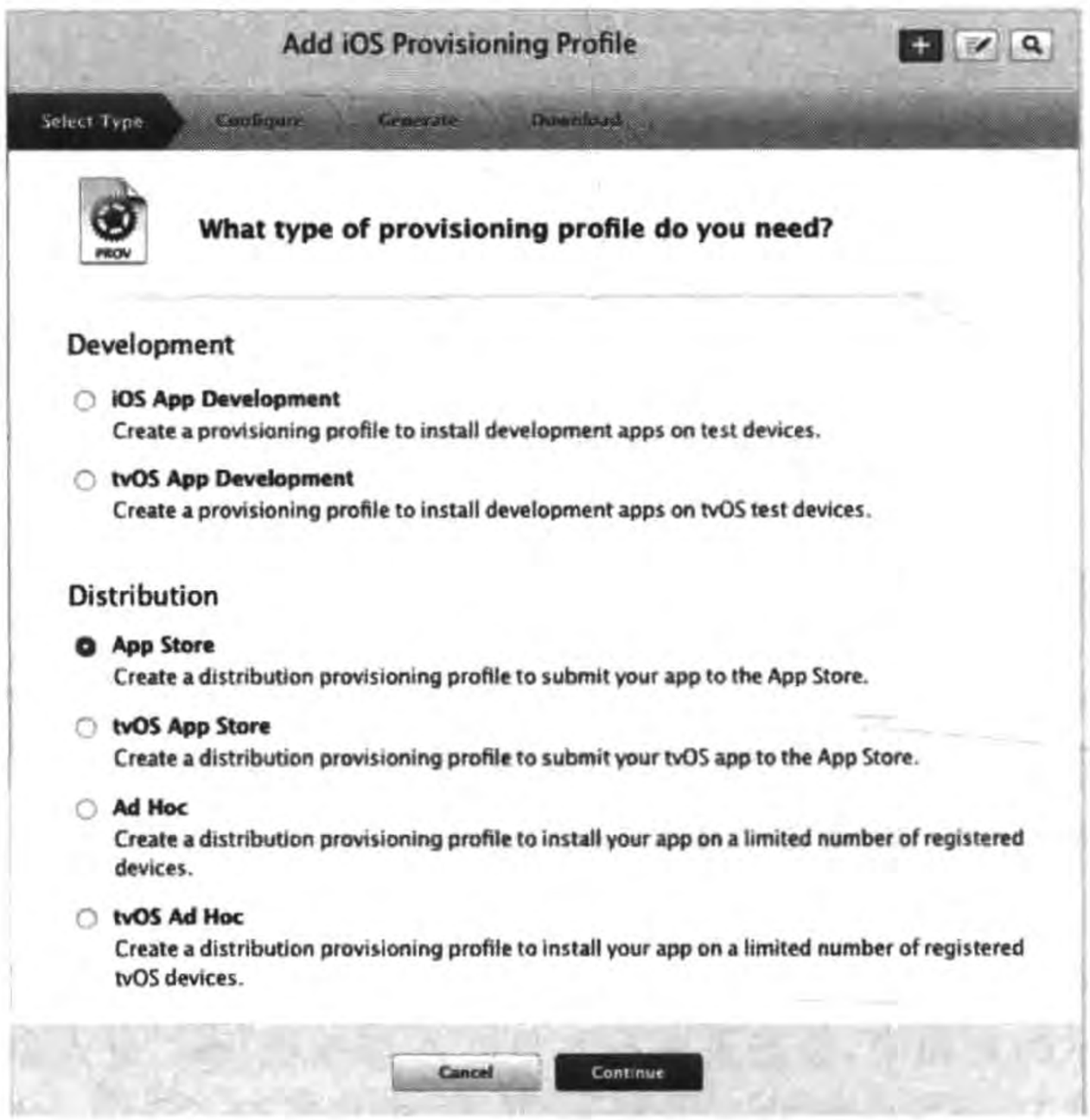


iOS Provisioning Profiles

49 profiles total.

Name	Type	Status
iOSTeam Provisioning Profile: *	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	● Active (Managed by Xcode)
iOSTeam Provisioning Profile: co...	iOS Development	⚠ Invalid (Managed by Xcode)

图 25-37　管理描述文件页面



What type of provisioning profile do you need?

Development

○ **iOS App Development**
Create a provisioning profile to install development apps on test devices.

○ **tvOS App Development**
Create a provisioning profile to install development apps on tvOS test devices.

Distribution

● **App Store**
Create a distribution provisioning profile to submit your app to the App Store.

○ **tvOS App Store**
Create a distribution provisioning profile to submit your tvOS app to the App Store.

○ **Ad Hoc**
Create a distribution provisioning profile to install your app on a limited number of registered devices.

○ **tvOS Ad Hoc**
Create a distribution provisioning profile to install your app on a limited number of registered tvOS devices.

图 25-38　创建描述文件选择页面

图 25-39　选择 App ID 页面

在图 25-39 所示页面，单击下面 Continue 按钮进入图 25-40 所示页面，这个页面中选择前面创建好的证书。然后单击下面 Continue 按钮进入图 25-41 所示生成描述文件页面，这个页面中输入描述文件名，最后单击下面的 Generate 按钮创建描述文件，创建完成后进入图 25-42 所示页面，在这个页面中可以下载这描述文件到本地。

图 25-40　选择证书页面

图 25-41　创建描述文件

图 25-42　创建完成描述文件

25.4.4　发布编译

双击上一节生成的描述文件 LostRoutesDistribution. mobileprovision，把它导入到 Xcode 工具。然后使用 Xcode 打开需要编译的工程或工作空间，选择工程的 PROJECT，再后选择 Build Settings→Code Signing→Provisioning Profile，如图 25-43 所示，在下拉列表中选择 LostRoutesDistribution。

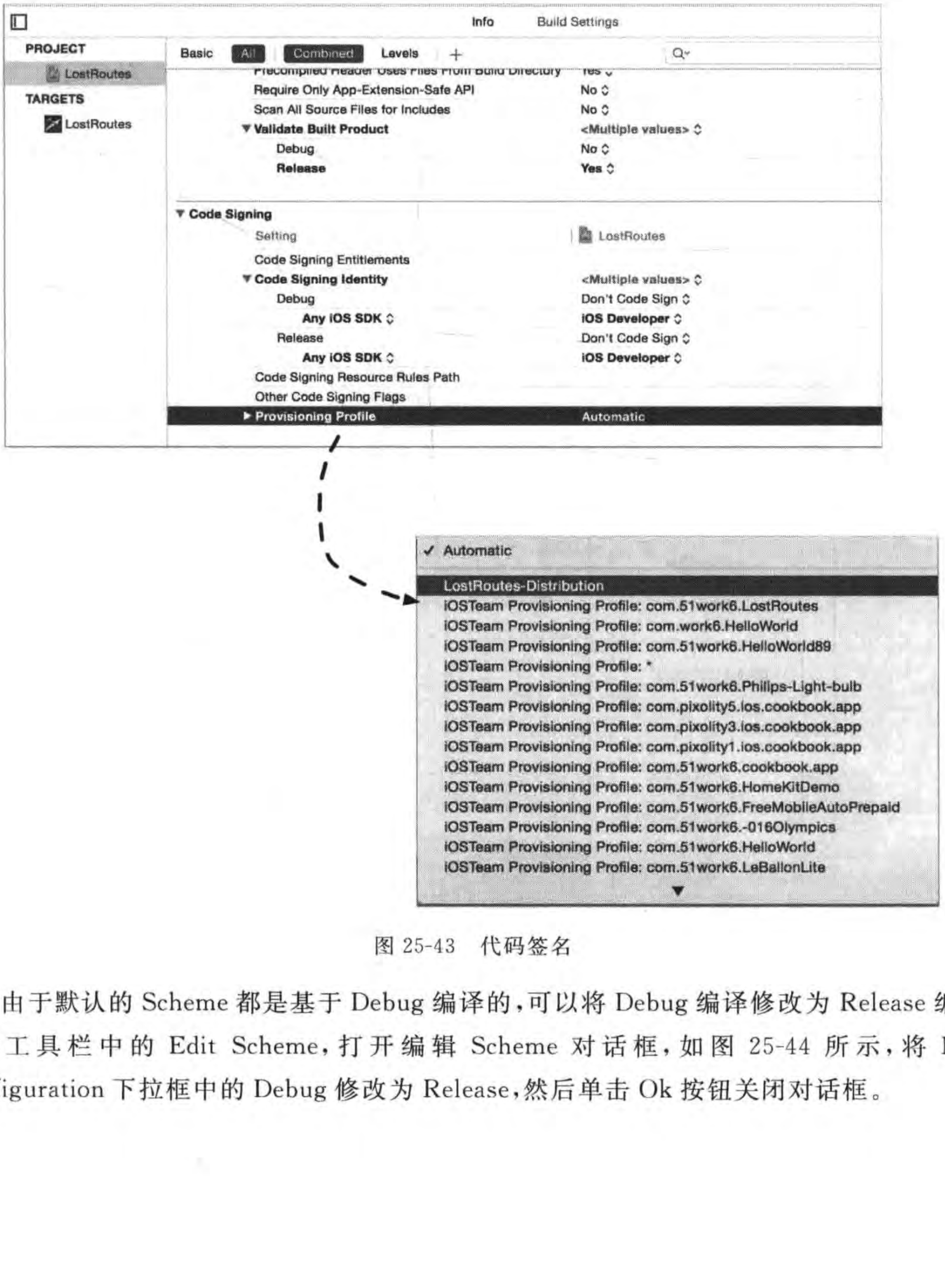


图 25-43　代码签名

由于默认的 Scheme 都是基于 Debug 编译的，可以将 Debug 编译修改为 Release 编译。选择工具栏中的 Edit Scheme，打开编辑 Scheme 对话框，如图 25-44 所示，将 Build Configuration 下拉框中的 Debug 修改为 Release，然后单击 Ok 按钮关闭对话框。

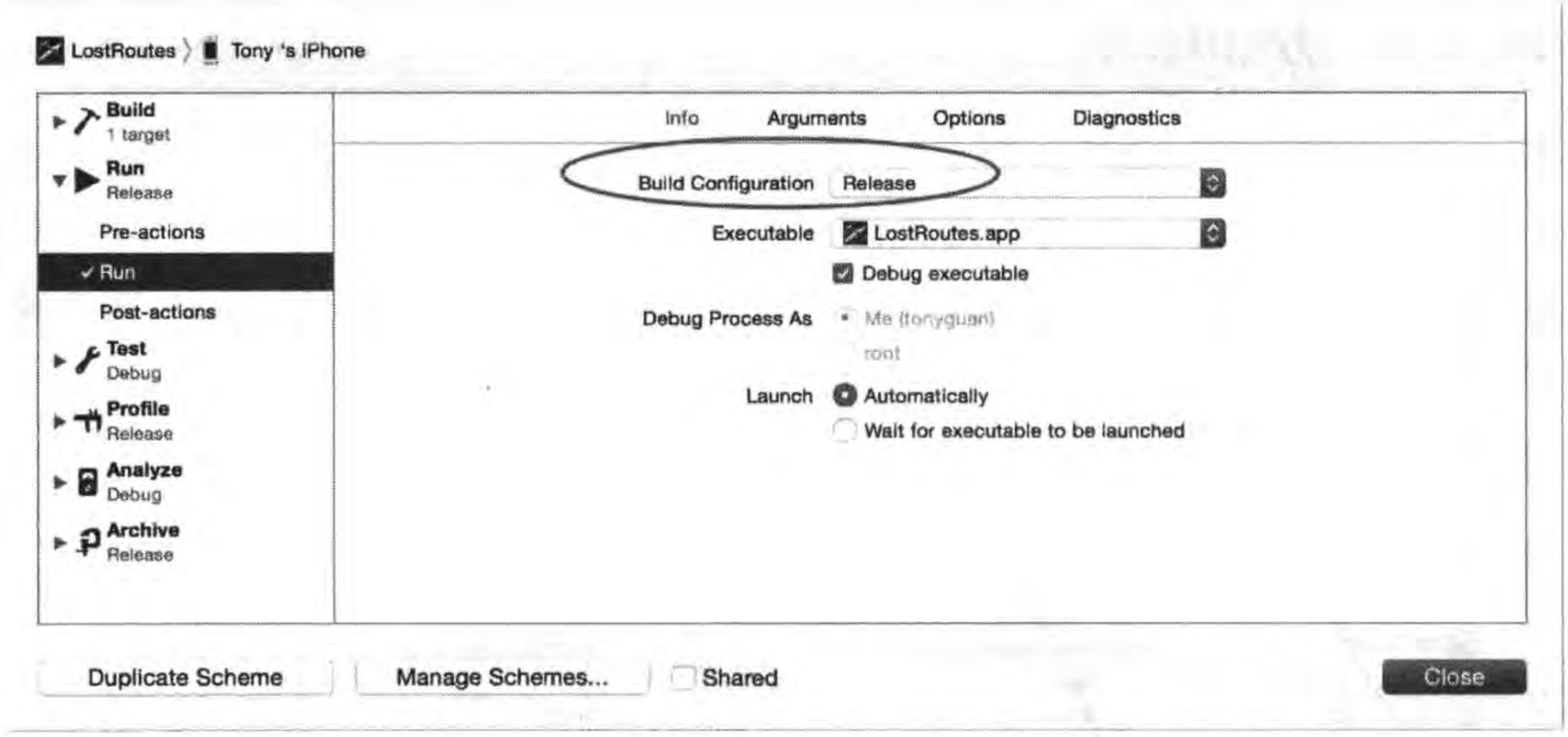


图 25-44　Release 编译

25.5　发布上架

应用程序编译通过就可以发布上架了。发布应用在 iTunes Connect 中完成，发布完成后等待审核，审核通过后就可以在 App Store 上架销售了。详细的发布流程如图 25-45 所示。

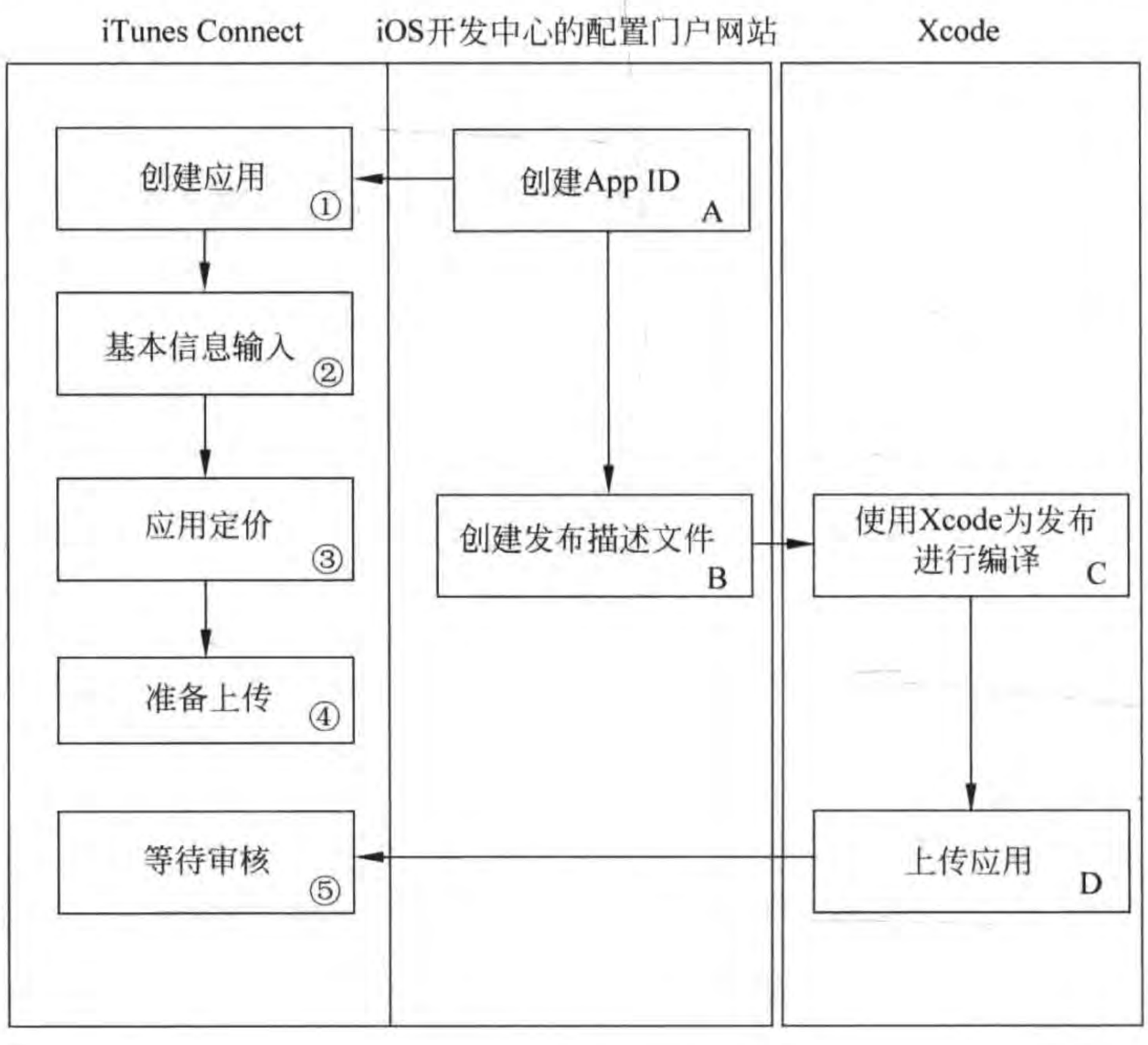


图 25-45　发布流程图

其中第 A 步和第 B 步是在 iOS 开发中心的配置门户网站中完成，这在本章前面已经介绍过了。这里我们介绍其他几个流程，其中主要的流程是在 iTunes Connect 中完成的，而编译和上传应用是在 Xcode 中完成的，上传应用还可以使用 Application Loader 工具完成。

25.5.1　创建应用

通过网址 https://itunesconnect.apple.com/WebObjects/iTunesConnect.woa 打开 iTunes Connect 登录页面，使用苹果开发账号登录，登录成功后的 iTunes Connect 页面如图 25-46 所示。

图 25-46　iTunes Connect 登录成功

单击“我的 App”图标，进入应用管理页面，如图 25-47 所示，在这里可以管理我们的应用，其中显示审核中的、未通过的以及已经上线的所有应用。

图 25-47　应用管理页面

单击左上角的“+”按钮，如图 25-48 所示，在下拉菜单中选择“新 App”菜单，弹出图 25-49 所示的添加新应用对话框，在这里可以输入应用的信息：

图 25-48　添加应用

(1) 平台，选择 iOS；

(2) 名称，为 LostRoutes，这个名称是显示到 App Store 上面的名字，是不能重复的；

(3) 主要语言，选择为 Simplified Chinese(简体中文)；

(4) 套装 ID，即 Bundle ID，它可以从开发者网站创建的 App ID 中选择；

(5) SKU，应用程序编号，具有唯一性，因此建议使用公司的“域名反写＋应用名”，这里我们输入的是 com. work6. LostRoutes。

在图 25-49 中单击创建按钮，进入图 25-50 所示信息输入页面，在这里可以设置游戏的类别，类别是应用的分类，也就是应用会发布到哪个频道，如果选择游戏，还要进行细化分类，因为游戏是 App Store 中最多的应用，所以分得比较细。这两个分类选项可以根据自己的应用进行填写，要求不是特别严格。

图 25-49　添加新应用页面

图 25-50　信息输入页面

25.5.2　应用定价

应用定价或许是我们最关心的了，在图 25-50 页面中单击左边导航菜单“APP STORE 信息”→“价格与定价”，出现图 25-51 所示页面，选择价格标签，出现价格等级选择，这里我

图 25-51　选择发布日期和定价页面

们选择“免费”。理论上,你可以定到很高的价格,是否能够卖得出就要看市场反馈了。这个定价很灵活,以后也可以更改。

定价完成单击页面上部的“存储”按钮保存。

25.5.3 基本信息输入

在图 25-50 页面中单击左边导航菜单“iOS APP”→“1.0 准备提交”,出现基本信息输入页面,其中包含更加详细的部分,包括应用基本数据信息、审核信息、版本发布,以及上传应用图标、截图和介绍视频,下面分别介绍。

1. 上传应用预览视频和截图

为了介绍、推广和宣传应用,App Store 允许我们为应用上传相关的视频和截图,除了 3.5 英寸只能上传截图外,其他的设备可以上传视频或截图,每一种设备所能上传的视频和截图总数不能超过 5 个,我们可以准备好视频和截图,然后把它们拖曳到图 25-52 中的选择文件区域,如果视频或截图的大小规格没有问题就可以上传了。

图 25-52 上传应用预览视频和截图

截图可以让用户了解我们的应用,视频和截图往往比文字描述更形象、更具有说服力。应用可能有很多情景和功能,我们一定要挑选最具特色、最突出的功能截图和演示视频。由于上传的截图和视频不能超过 5 个,一定要把最好的截图和视频放到前面,因为后面的截图和视频需要向后滑动才能出现,这样才能吸引用户对我们的应用产生兴趣,考虑购买我们的应用。

2. 基本信息输入

App 视频预览和屏幕快照的下面是应用的基本信息输入页面，如图 25-53 所示，“描述”对应用很重要，将出现在 App Store 的应用介绍中。用户购买应用时，主要通过这段文字来了解我们的应用到底是做什么的，有什么用。因此，要认真、用心地准备这段文字，描述清楚应用的所有功能，体现出应用的特点、特色等，从而吸引用户来购买。

图 25-53　基本信息输入

“关键词”是在 App Store 上查询该应用的关键词。“技术支持网址”里面需要填写应用技术支持的网址，“营销网址”里面填写应用营销的网址，主要是针对应用做进一步介绍。由

于“描述”的文字和图片数是有限制的，可能不会把应用介绍得很详尽，所以我们可以自己创建一个网页，更详细地介绍我们的应用。

3. 综合信息输入

将图 25-53 的页面向下滚动，则出现如图 25-54 所示的界面，可以在该界面中输入 App 程序图标、版本号、分级和版权声明等信息。

图 25-54　综合信息输入

这里的应用程序图标是在 App Store 上显示的，它要求 1024×1024 像素的 PNG 或 JPEG 格式图片。关于图标设计我们一定要下点工夫，图标能够给用户带来第一印象的感受，所以一定要用心去设计。

单击分级旁边的“编辑”链接弹出图 25-55 所示的分级界面，分级是根据应用中含有色情、暴力等内容的程度进行分级。不同的等级标识使用该应用的年龄段。同时，也会有一些国家根据这个评级高低来限制是否在本国销售。在这个选项中，开发者应该按应用的实际情况来填写，如果与所描述的内容不符，苹果会拒绝审核通过。

4. 应用审核信息

将图 25-47 的页面向下滚动，则出现图 25-56 所示的界面，该界面是应用审核信息输入界面，这里的信息主要是给苹果审核团队的工作人员看的。在“联系信息”中填写开发者团队中负责与苹果审核小组联系的人员的信息，包括姓名、邮箱和电话号码。

在“备注”中，填写应用细节和一些特别的功能，帮助审核人员快速了解该应用。在“演示账户”中，填写应用中的测试账号和密码，提供给审核人员测试，以便更加顺利地通过审核。

编辑评级

对于每项内容描述，请选择对您的 App 描述最为贴切的频率级别。将显示在 App Store 中的 App 评分在您的所有平台上均相同。它基于该 App 评分最高的平台。了解更多

App 不得包含任何暴力、色情、露骨性、诽谤性内容或任何此类资料（文本、图片、图像、照片等），或者 Apple 合理判断可能认为不适宜的其他内容或资料。

Apple 内容描述	无	偶尔/轻微的	频繁/强烈的
卡通或幻想暴力	◉	○	○
现实暴力	◉	○	○
大量露骨或残暴的现实暴力	◉	○	○
低俗笑话	◉	○	○
成人/性暗示题材	◉	○	○
恐怖/惊悚题材	◉	○	○
医学/医疗信息	◉	○	○
使用或提及烟、酒或毒品	◉	○	○
模拟赌博	◉	○	○
色情或裸露内容	◉	○	○
色情及裸体画面	◉	○	○
	否	是	
无限制的网站访问	◉	○	
赌博和竞赛	◉	○	

☐ 面向儿童

ⓘ 您选择的 App 评级为 4+ 岁

取消　完成

图 25-55　分级界面

App 审核信息

联系信息 ?

Guan　Tony

eorient@sina.com

备注 ?

演示账户 ?

用户名　密码

4000

图 25-56　应用审核信息输入界面

5. 版本发布

将图 25-56 的页面向下滚动，则出现图 25-57 所示的版本界面，该界面中可以设置在应用审核通过后自动发布还是手动发布。

图 25-57　版本发布界面

确认这些输入的信息无误后，单击页面上部的"存储"按钮保存输入。

25.5.4　上传应用

基本信息输入完成，选择"iOS APP"→"1.0 准备提交"页面中的"提交以供审核"按钮进行提交，在这之前需要先上传应用，上传应用可以使用 Xcode 或 Application Loader 工具，或者两者结合使用，为了简单我们推荐只使用 Xcode 工具。

首先在 Xcode 中选择菜单 Product→Archive 为应用归档，归档结束打开图 25-58 所示

图 25-58　归档界面

的界面，选中刚刚归档的项目，单击右边的 Validate 按钮进行验证要上传的程序包，这时会弹出验证确认对话框，如图 25-59 所示，然后单击 Validate 按钮开始验证，验证结束之后会有一个是否成功的提示。在上传之前进行验证是非常必要的。

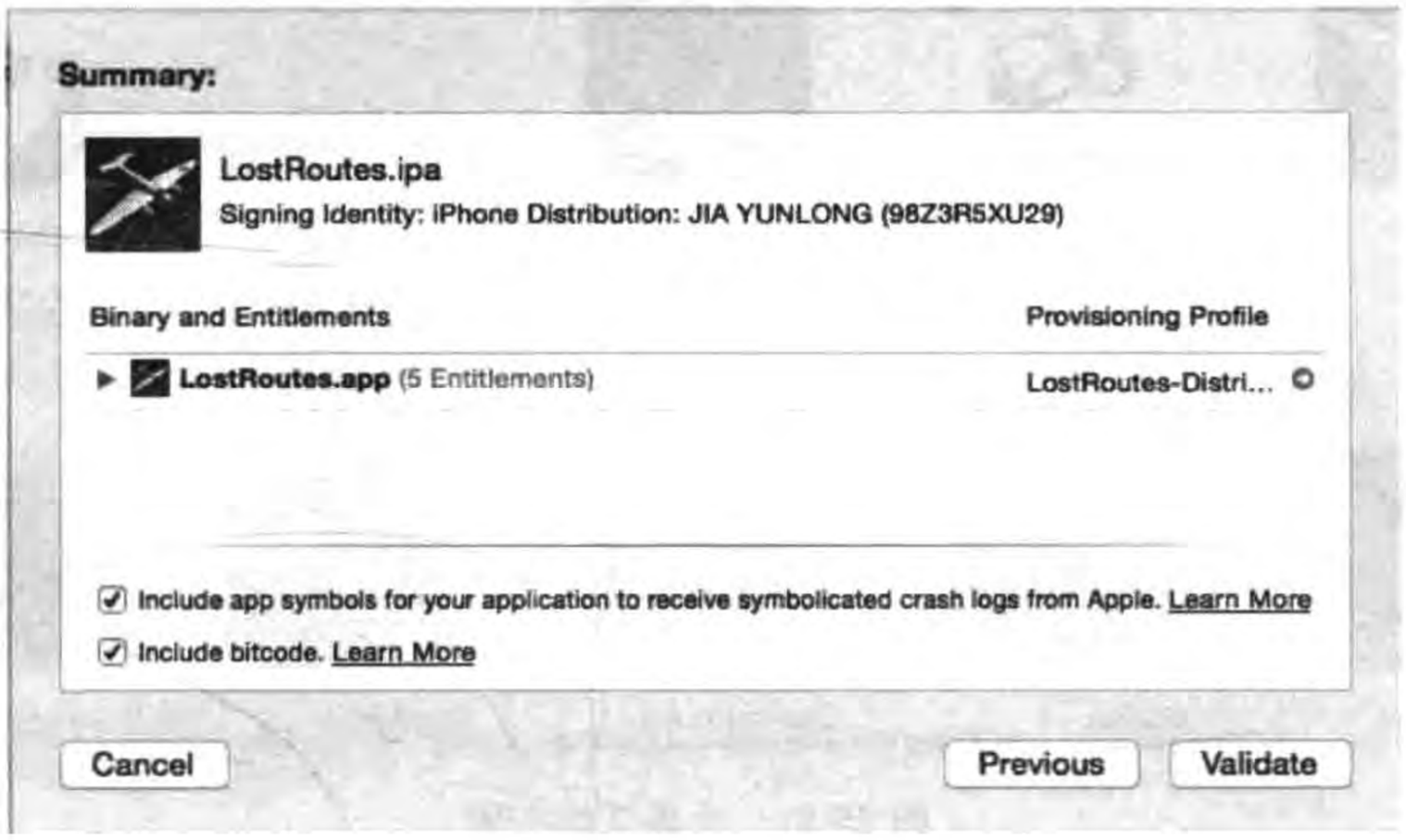


图 25-59　验证程序包

在图 25-58 所示的界面中单击 Upload to App Store 按钮，会弹出如图 25-60 所示对话框，单击 Submit 按钮开始上传程序包，上传结束之后会有一个是否成功的提示。

图 25-60　提交程序包

应用程序上传之后的状态，可以参考 iTunes Connect，登录 iTunes Connect 打开应用会看到已经上传的应用，如图 25-61 所示，单击应用进入应用管理页面，在左边导航菜单选择“iOS APP”→“1.0 准备提交”页面，在版本构建部分单击“＋”按钮，如图 25-62 所示，弹出对话框，在对话框中选择我们上传的应用程序包，然后单击完成按钮。

图 25-61　上传之后应用

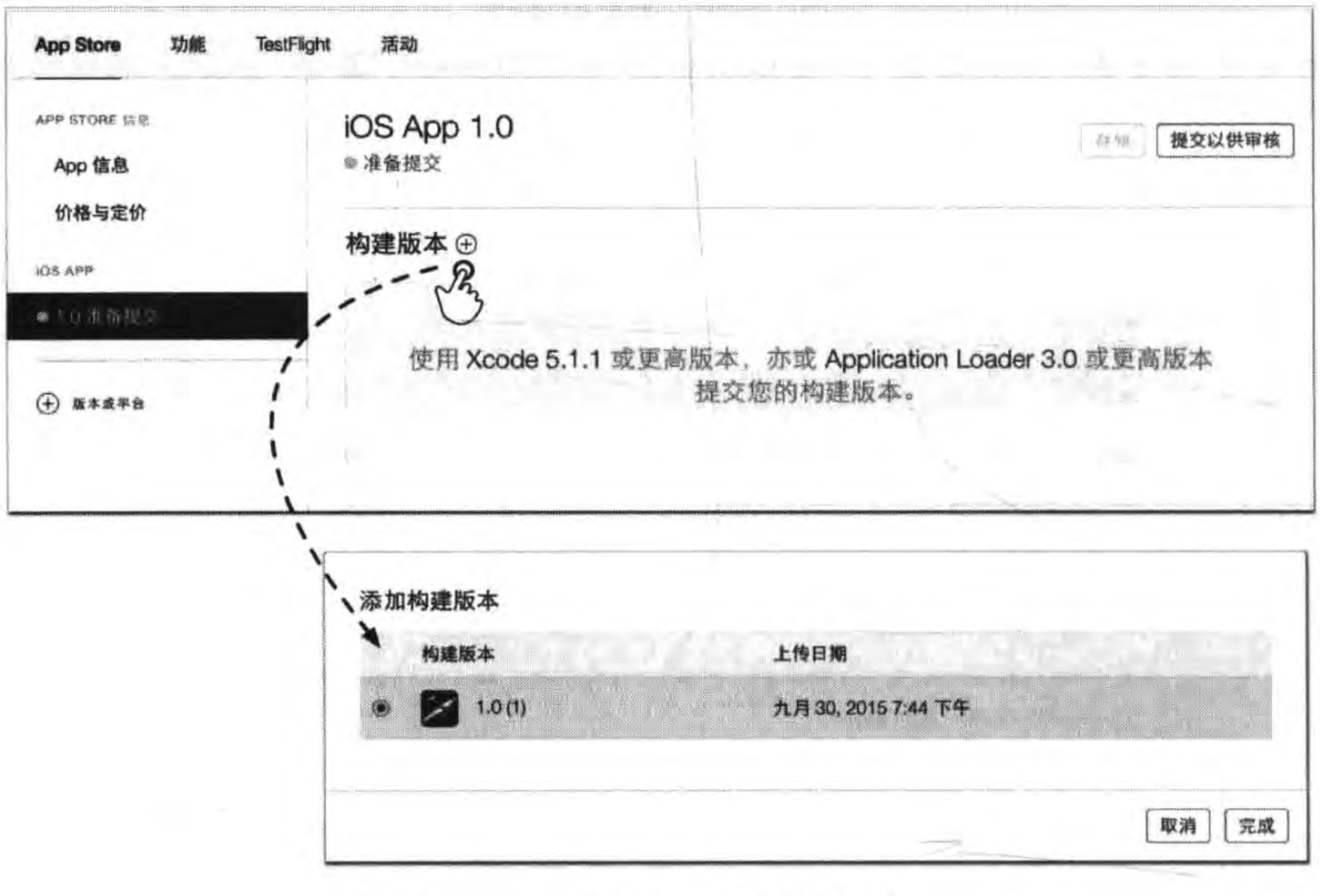


图 25-62　选择上传的应用程序包

25.5.5　提交审核

选择上传的应用程序包后，如果确认无误，就可以单击“提交以供审核”按钮提交审核了。在提交审核之前 iTunes Connect 还需要我们最后确认几件事情，包括出口合规信息、

内容版权和广告标识符，如图 25-63 所示。

图 25-63　最后确认

出口合规信息是询问程序代码中是否有加密算法，美国出口法律规定禁止任何加密的软件流向国外，这里我们选择是即可。内容版权是询问应用中是否有包含、显示和访问第三方内容。广告标识符是询问应用中是否使用了 IDFA 广告标识。这些内容一定要实事求是填写，特别是 IDFA 广告标识，如果选择否它还会有更加详细的询问，如图 25-64 所示。

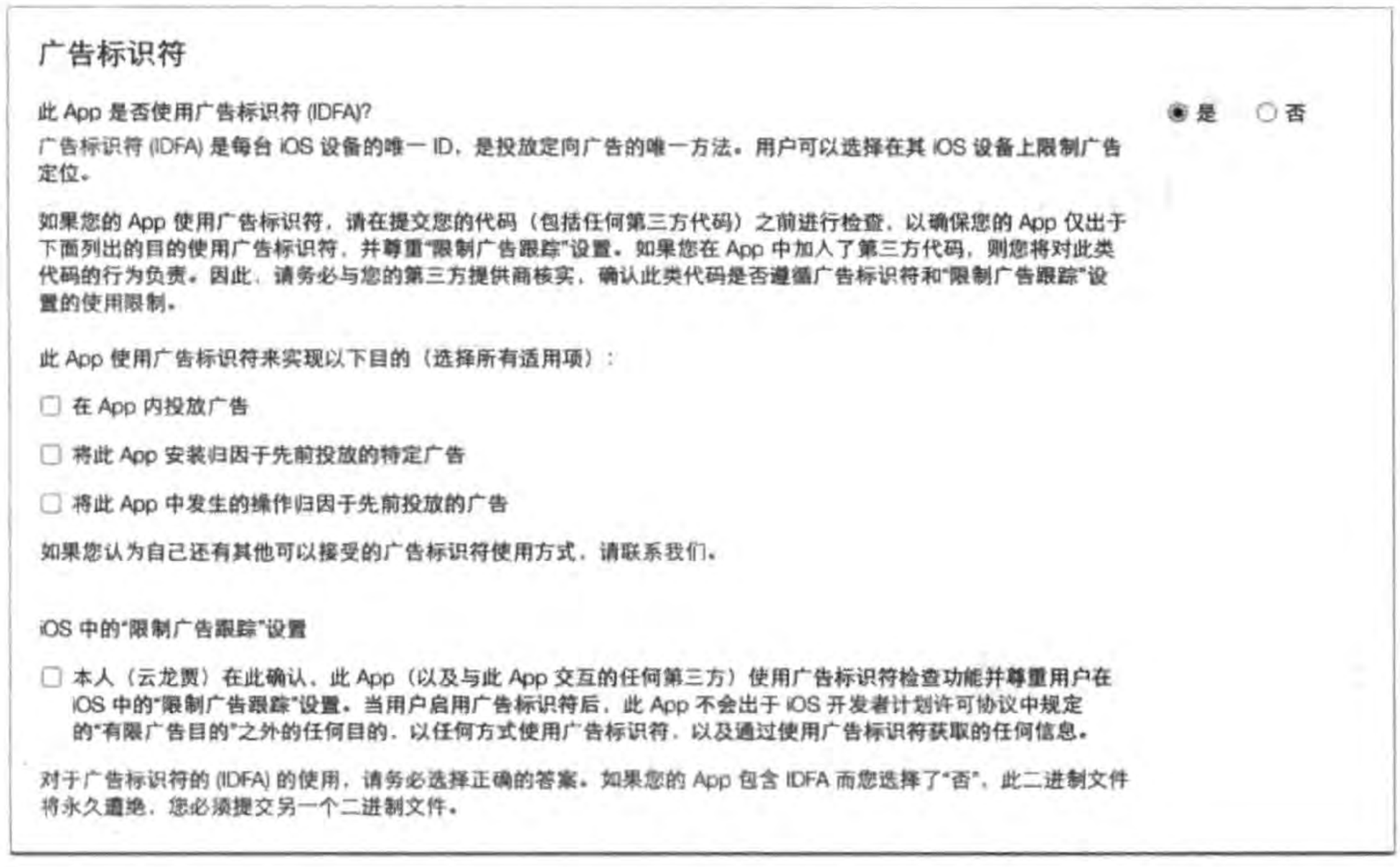


图 25-64　IDFA 广告标识确认

完成这些工作后单击“提交”按钮提交应用,然后应用状态就变成了“正在等待审核”。如果没有任何问题,接下来就是等待了。因为每天有很多程序要发布到 App Store 中,所以等待审核也要排队。如果状态变为“已经上架”就表示审核通过了,并且可以销售了。

25.6 常见审核通不过的原因

App Store 的审核是出了名的严格,相信大家也都略有耳闻。苹果官方提供了一份详细的审核指南,包括 20 多大项、100 多小项的拒绝上线条款,并且条款还在不断增加中。此外,还包含一些模棱两可的条例,所以稍有“闪失”,应用就有可能被拒绝。但是有一点比较好,那就是每次遭到拒绝时,苹果会给出拒绝的理由,并指出你违反了审核指南的哪一条,开发者可以根据评审小组给的回复修改应用重新提交。下面我们讨论一下常见的被拒绝的原因。

25.6.1 功能问题

在发布应用之前,我们一定要对产品进行认真的测试,如果在审核中出现了程序崩溃或者程序错误,无疑这是会被审核小组拒绝的。如果我们想发布一个演示版的程序,通过它给客户演示这也是不会被通过的。应用的功能与描述不相符,或者应用中含有欺诈虚假的功能,那么应用将被拒绝。比如在应用中有某个按钮,但是单击这个按钮时没有反应或者不能单击,这样的程序是不会通过的。

苹果不允许访问私有 API,有浏览器的网络程序必须使用 iOS WebKit 框架和 WebKit JavaScript。还有几点比较头痛的规则,那就是如果你的 App 没有什么显著的功能或者没有长久娱乐价值,也会被拒绝。如果你的应用在市场中已经存在了,在相关产品比较多的时候也可能被拒绝。

25.6.2 用户界面问题

苹果审核指南规定开发者的应用必须遵守苹果《iOS 用户界面指导原则》中解释的所有条款和条件,如果违反了这些设计原则,就会被拒绝上线,所以开发者在设计和开发产品之前一定要认真阅读《iOS 用户界面指导原则》。这些原则中也渗透着苹果产品的一些理念,不仅是为了避免程序被拒绝而看,而且还为了让开发者设计出更好的 App。苹果不允许开发者更改自身按键的功能(包括声音按键以及静音按键),如果开发者使用了这些按键并利用它们做一些别的功能,将会被拒绝。

25.6.3 商业问题

在要发布的应用中,首先不能侵犯苹果公司的商标及版权。简单地说,在应用中不能出现苹果的图标,不能使用苹果公司现在产品的类似名字为应用命名,涉及 iPhone、iPad、iTunes 等相关或者相近的名字都是不可以的。苹果认为这会误导用户,认为该应用是来自

苹果公司的产品。误导用户认为该应用是受到苹果的肯定与认可的，也是不行的。

私自使用受保护的第三方材料（商标、版权、商业机密和其他私有内容），需要提供版权认可。如果你的应用涉及第三方版权的信息，开发者就要仔细考虑考虑了。由于有些开发者对版权法律意识比较淡薄，总会忽视这一点，然而这一点是非常致命的。苹果处理这种被起诉的侵权应用，最轻的处罚是下架应用，有时需要将开发者账户里的钱转到起诉者账户。再严重的就是，起诉者将你告上法庭，除了自己账户中的钱被扣除外，还要另赔付起诉者相关费用。

25.6.4　不当内容

一些不合适、不和谐的内容，苹果当然不会允许上架的。比如具有诽谤、人身攻击的应用，含有暴力倾向的应用，低俗、令人反感、厌恶的应用，赤裸裸的色情应用等。含有赌博性质的应用必须明确表示苹果不是发起者，也没有以任何方式参与活动。

25.6.5　其他问题

关于宗教、文化或种族群体的应用或评论包括诽谤性、攻击性或自私性内容的应用不会被通过，使用第三方支付的应用会被拒绝，模仿 iPod 界面的应用将会被拒绝，怂恿用户造成设备损坏的应用会被拒绝。这里有个小故事，有一款应用，功能是比比谁将设备扔得高，最后算积分，这个应用始终没能上架，因为在测试应用的时候就摔坏了两部手机。此外，未获得用户同意便向用户发送推送通知，要求用户共享个人信息的应用都会被拒绝。

本章小结

通过对本章的学习，我们了解了如何在 App Store 上发布应用，发布之前需要处理哪些问题。发布者需要了解应用的发布流程，更应该熟悉应用审核通不过的一些常见原因，从而在开发时注意，以免等到审核时被拒绝，耽误了应用的上架时间。

Cocos2d-x实战

Lua卷 | 关东升◎著

（第2版）

基于Cocos2d-x引擎开发的产品已经占据移动游戏市场第一名的位置，Cocos2d-x仍继续吸引更多的开发者投入其中。著名移动开发专家关东升老师创作的这套“Cocos2d-x实战”图书，全面系统地论述了Cocos2d-x开发的理论与实践。书中构建了最全面的Cocos2d-x学习体系：基础知识→核心技术→实战特训，是初学者极佳的学习路线！堪称Cocos2d-x开发入门指导的百科全书！

——CocoaChina

关东升老师具有多年的Cocos2d-x教学经验，他的教学视频、技术书籍与技术博客都深受欢迎。“Cocos2d-x实战”系列图书是关东升老师倾力创作的技术经典，书中融汇了多年的教学实践与应用开发经验，凝练了大量贴合开发者实际需求的要点。如果您是一位渴望通过学习Cocos2d-x进入游戏开发领域的程序员，本书应当成为您必读的领航之作。

——CSDN

“Cocos2d-x实战”是关东升老师用创业者的心态撰写的技术图书，并全面超越了同类图书！经过长久的积淀，历经Cocos2d-x众多版本的变化，关老师及其团队不断修改案例、程序，终于完美定稿。借助本书，开发者可以快速进行项目实战，轻松完成“做中学”的体验。

——9ria

时至今日，移动互联网改变了人们的生活方式，同时也改变了IT产业的格局，手机游戏开发无疑成了移动开发领域最火热的技术方向。“Cocos2d-x实战”系统论述了Cocos2d-x全方位的游戏开发知识与技能。无论您是就业者还是创业者，您都会通过本书获益匪浅。

——51CTO

清华大学出版社数字出版网站
WQBook
www.wqbook.com

扫一扫

阅读 | 分享

上架指导：移动开发/游戏开发

ISBN 978-7-302-45730-5

9 787302 457305 >

定价：89.00元