# 第 3 章

# Cocos-Lua 基础

## 3.1 Cocos2d-Lua 开发环境配置

工欲善其事，必先利其器。在进入 Cocos2d-Lua 学习之前，需要先搭建 Cocos2d-Lua 的开发环境。

### 3.1.1 安装 Cocos2d-Lua

首先，在 http://cocos2d-lua.org/download/index.md 下载最新的 Cocos2d-Lua 版本。然后解压 zip 包到一个路径，路径以及文件夹的名称不能有中文或空格。

**注**：编写本教材时最新的版本为 Quick-Cocos2dx-Community 3.7.6。

**1. 在 Mac 系统下安装引擎**

进入解压后的引擎目录，在终端中执行引擎根目录下的 setup_mac.sh 脚本，执行脚本的命令前不需要加入 sudo。当提示输入密码的时候，请输入当前用户的登录密码。正确的安装过程如图 3-1 所示。安装完成后，单击 Launchpad 中的 Player3 启动引擎。

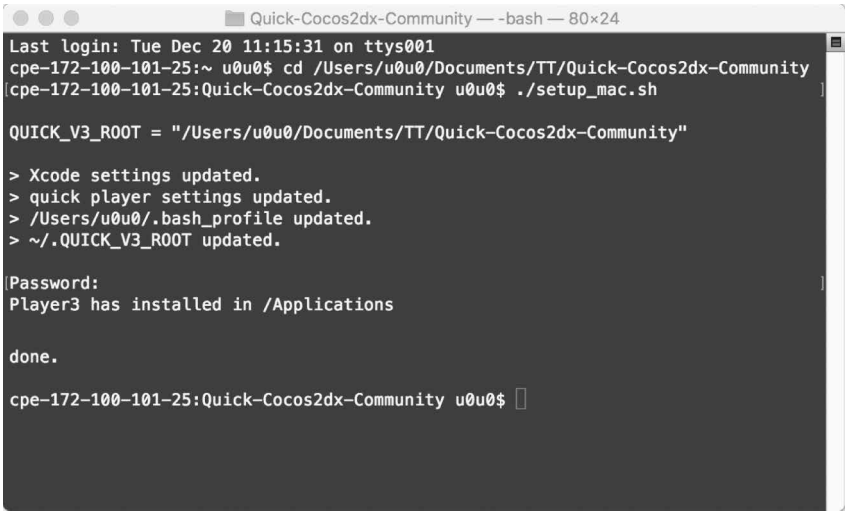



图 3-1　setup_mac.sh

部分 Mac 用户可能会遇到环境配置后，新建工程的 mac_ios 项目不能编译通过。这时需要在终端手动输入下面的命令配置 Xcode 环境变量。

```
defaults write com.apple.dt.Xcode IDEApplicationwideBuildSettings -dict
#路径替换为自己的 Quick root
defaults write com.apple.dt.Xcode IDEApplicationwideBuildSettings -dict -add QUICK_V3_ROOT
"/User/u0u0/Quick_cocos2dx_Community"
defaults write com.apple.dt.Xcode IDESourceTreeDisplayNames -dict
defaults write com.apple.dt.Xcode IDESourceTreeDisplayNames -dict -add QUICK_V3_ROOT QUICK_
V3_ROOT
```

**2. 在 Windows 系统下安装引擎**

在解压后的引擎目录中，双击 setup_win.bat 开始执行安装脚本。正确的安装过程如图 3-2 所示。安装完成后，双击 Windows 系统桌面上的 Player3 快捷方式启动引擎。


图 3-2 setup_win.bat

在 Player 中，可以进行创建项目、运行项目等操作。切换到“示例”标签，可运行引擎自带的各种测试程序，如图 3-3 所示。

## 3.1.2 安装 VS Code 与 QuickXDev

脚本语言的优点是无须编译且 log 功能强大，但开发环境简陋。幸运的是，Cocos2d-Lua 社区的 lonewolf 基于强大的 VS Code 编辑器，开发了 QuickXdev 插件，为 Cocos2d-Lua 代码编写提供了不少便利。

VS Code 的下载地址为 https://code.visualstudio.com/，下载对应开发系统的安装包，双击开始安装，安装过程中注意勾选 Create a desktop icon，方便以后启动 Sublime。

安装完 VS Code 后，进行 QuickXdev 开发插件的安装。步骤如下：

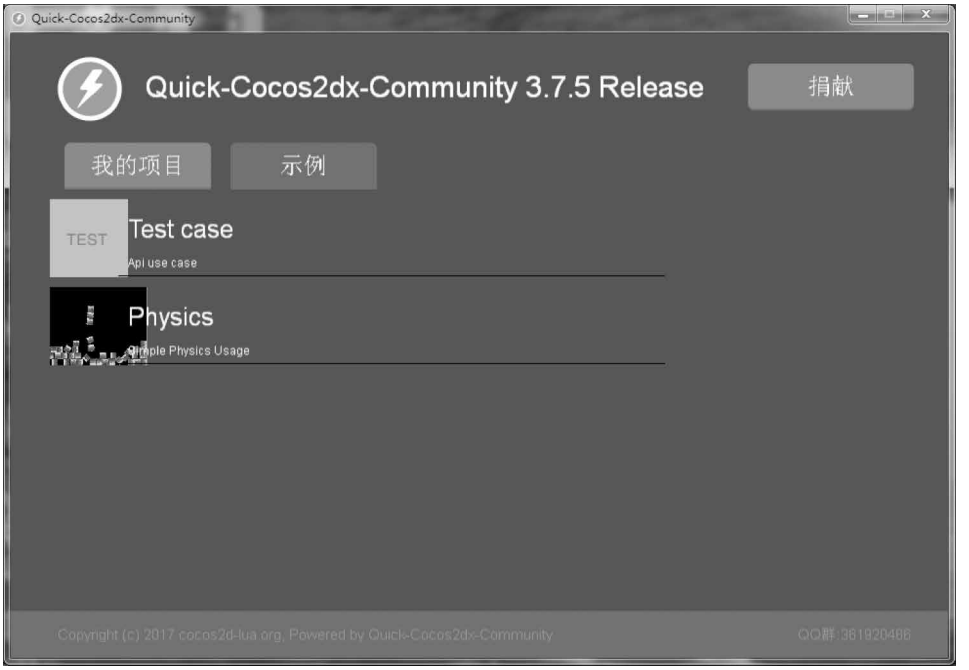



图 3-3　Player

(1) 启动 VS Code,单击左侧的 Extensions 按钮,如图 3-4 所示步骤(1)。

(2) 在插件搜索输入框中输入 QuickXdev,如图 3-4 所示步骤(2)。

(3) 单击 Install 按钮等待安装完毕,如图 3-4 所示步骤(3)。

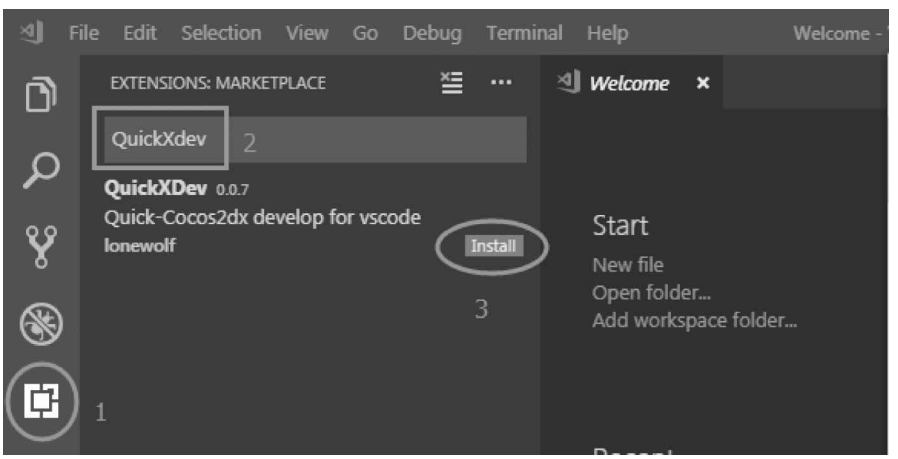



图 3-4　安装 QuickXdev 插件

QuickXdev 插件会自动读取 setup.bat 设置的环境变量,无须配置。

### 3.1.3 测试开发环境

最终完成了环境搭建，但它是否能正常工作呢？接下来测试一下整个开发环境。

首先启动 Player3，单击“新建项目”按钮，出现如图 3-5 所示的界面。

(1) 单击 Select 按钮选择项目根文件夹。

(2) 在 Project package name 栏输入项目的包名。为了 Android 的兼容性，包名中以“.”分隔的每一个单词都不能以数字开头。包名应为 3 个或 4 个单词。

(3) Portrait 为竖屏，Landscape 为横屏。

(4) Create Project 创建项目。

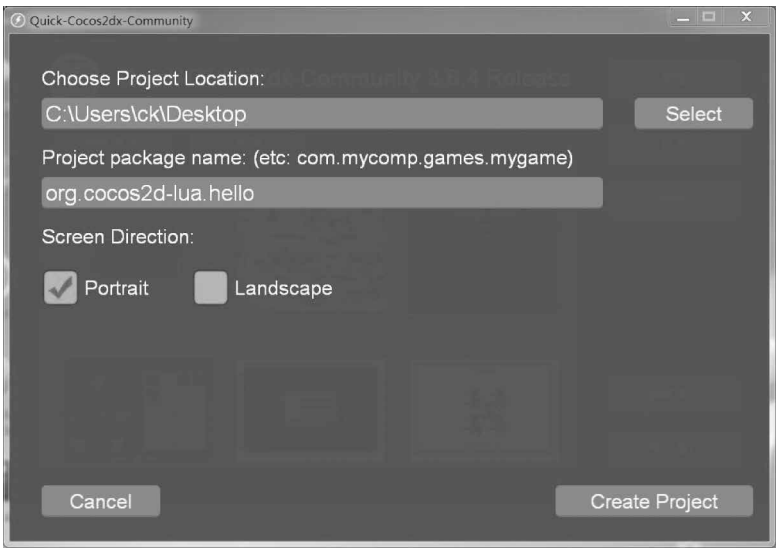



图 3-5 新建项目

项目创建完成，回到 Player3 主界面，单击“导入项目”按钮，导入刚才创建的项目，如图 3-6 所示。

(1) 单击 Select 按钮，选择项目路径。

(2) 单击 Open Project 按钮，开始导入项目。

成功导入项目后它会自动运行，这样就可以看到“Hello，World”界面了。

Player3 主界面的“我的项目”标签页下，也会自动创建这个 helloworld 项目的快捷启动项。以后需要再次启动这个项目时，只需选中它并单击“打开”按钮即可。

接下来尝试修改 helloworld 的代码。

首先，把项目文件夹导入 VS Code。方法很简单，启动 VS Code，拖动“D:”目录下的 helloworld 文件夹到 VS Code 窗口界面。松开后在 VS Code 的左边栏中就会显示整个文件结构。然后，选择 src/app/scenes/MainScene.lua 文件，如图 3-7 所示。

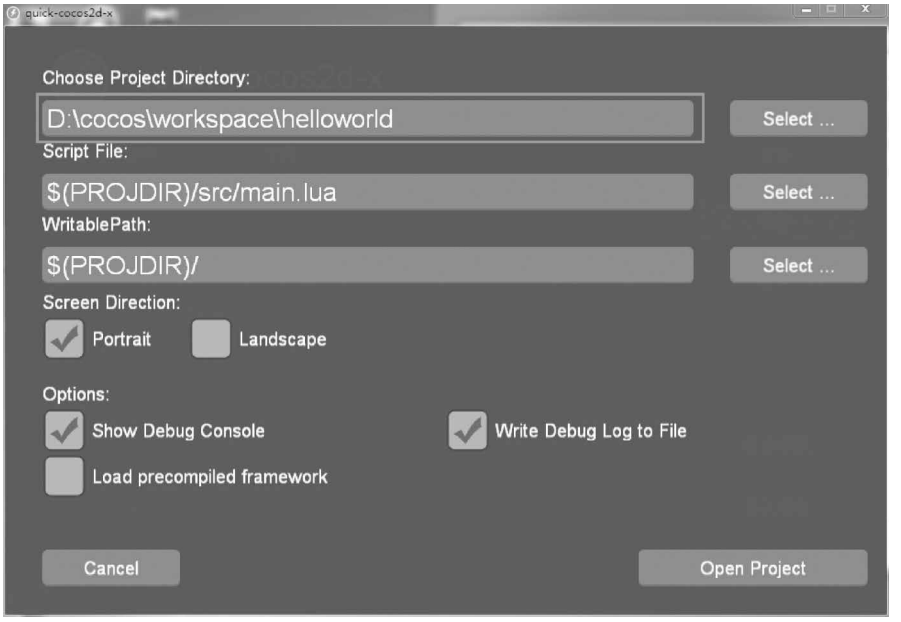



图 3-6　导入项目

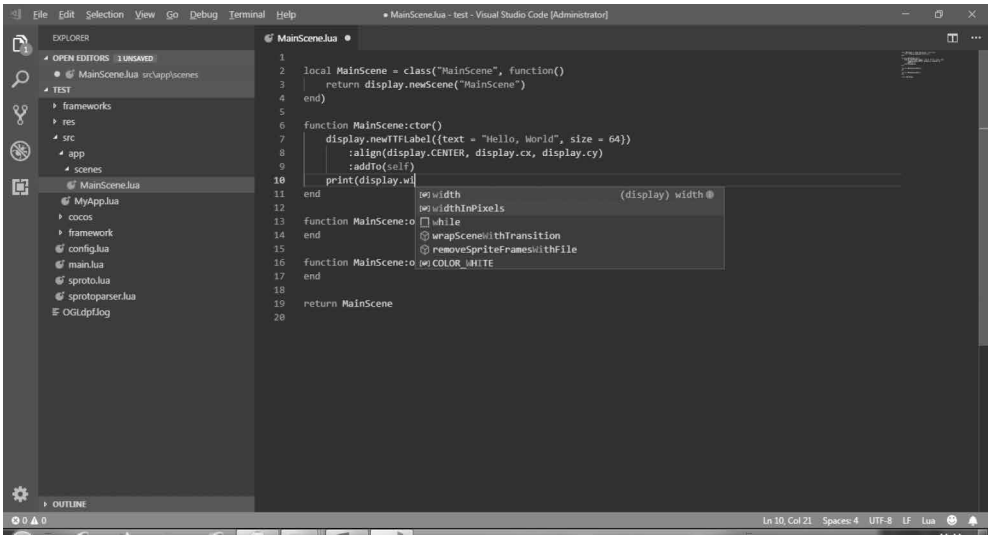



图 3-7　VS Code 打开项目

输入 print 后能自动提示 Lua 标准接口，输入 disp 后能自动提示 Cocos2d-Lua 的接口，表明 QuickXDev 插件正常工作。

选择 display. width，按 Enter 键，补全整个单词。保存文件，切换到 Player3 的 helloworld 项目模拟器，选择 View→Refresh 命令刷新模拟器，可以看到 Player3 的控制台

有新加入的 log 信息。更方便的调试方式是在 VS Code 中直接启动 QuickXDev，在 MainScene.lua 代码编辑器右击，从弹出的快捷菜单中选择“在 Player 中运行”命令，如图 3-8 所示。

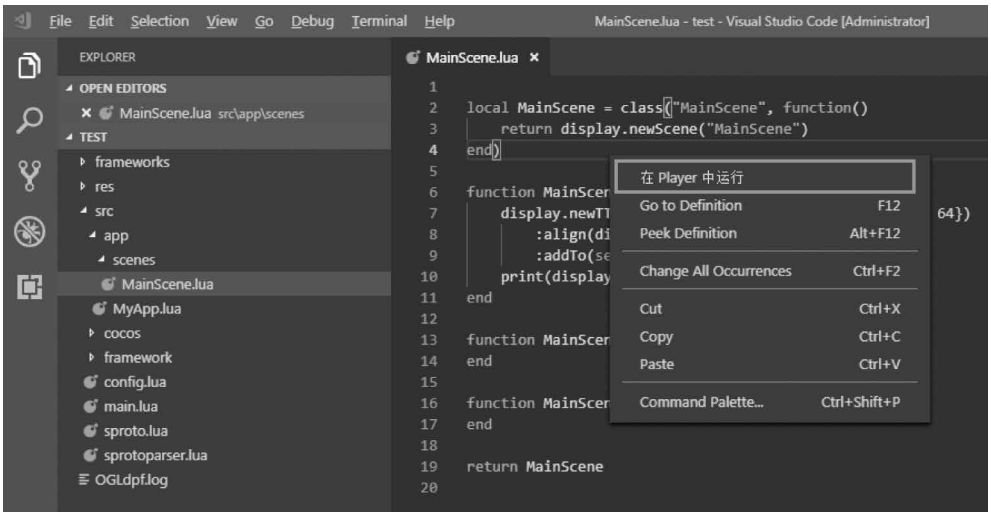



图 3-8　VS Code 运行 Player

## 3.2　引擎架构与目录结构

### 3.2.1　引擎架构

在第 1 章介绍了 Cocos2d 引擎家族历史和 Cocos2d-Lua 的诞生。我们知道 Cocos2d-Lua 是由 Cocos2d-x 衍生出来的，它们之间的关系如图 3-9 所示。

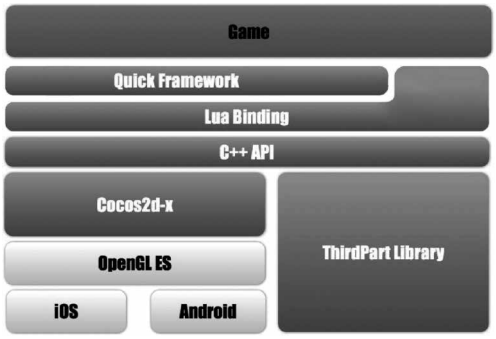



图 3-9　Cocos2d-Lua 框架图

白色区域的模块与具体平台相关，引擎渲染居于跨平台的 OpenGL ES。在 OpenGL ES 之上是 Cocos2d-x 引擎核心架构，Cocos2d-x 与 ThirdPart Library 对外暴露 C++ API。

Lua Binding 把 C++API 映射为 Lua API，而 Quick Framework 基于 Lua API 进行二次封装并扩展了接口。

游戏在最上层，可以同时使用 Lua API 和 Quick 框架。

Cocos2d-x 内部又可以细分为多个模块，如图 3-10 所示。


图 3-10　Cocos2d-x 模块

## 3.2.2　引擎文件结构

以下介绍基于当前最新 Cocos2d-Lua 版本 v3.7.6。进入 Cocos2d-Lua 安装路径，可以看到如图 3-11 所示的目录结构。

| 名称 | 修改日期 | 大小 | 种类 |
|---|---|---|---|
| build | 2017年4月22日 下午7:37 | -- | 文件夹 |
| CHANGELOG | 2019年2月14日 下午10:28 | 17 KB | Hex-Ed...pp 文稿 |
| cocos | 2019年2月14日 下午10:28 | -- | 文件夹 |
| external | 2017年4月22日 下午7:37 | -- | 文件夹 |
| licenses | 2014年12月24日 下午7:41 | -- | 文件夹 |
| quick | 2017年6月15日 下午9:35 | -- | 文件夹 |
| README.md | 2019年1月17日 下午8:26 | 3 KB | Markdown |
| setup_mac.sh | 2018年12月21日 下午11:00 | 2 KB | Shell Script |
| setup_win.bat | 2015年10月19日 下午2:25 | 489 字节 | Visual...app 文稿 |
| VERSION | 2019年2月14日 下午10:28 | 14 字节 | Hex-Ed...pp 文稿 |

图 3-11　引擎根目录

(1) build：引擎 Xcode 的 lib 工程。

(2) cocos：Cocos2d-x 引擎 C++代码。

(3) external：引擎集成的第三方库。

(4) licenses：引擎以及第三方库的 license 说明文件。

(5) quick：Quick 框架代码以及工具。

(6) setup_mac.sh：Mac 下设置 Cocos2d-Lua 环境变量的脚本文件。

(7) setup_win.bat：Windows 下设置 Cocos2d-Lua 环境变量的脚本文件。

(8) VERSION：当前引擎版本。

(9) CHANGELOG：引擎更新日志。

quick 目录下的 Quick 框架将是本书的重点内容，目录结构如图 3-12 所示。

| 名称 | 修改日期 | 大小 | 种类 |
|---|---|---|---|
| bin | 2019年2月14日 下午10:28 | -- | 文件夹 |
| cocos | 2018年8月12日 下午8:58 | -- | 文件夹 |
| framework | 2019年2月24日 下午9:41 | -- | 文件夹 |
| lib | 2017年4月22日 下午7:37 | -- | 文件夹 |
| player | 2019年2月24日 下午10:36 | -- | 文件夹 |
| samples | 2019年2月14日 下午10:28 | -- | 文件夹 |
| template | 2017年6月15日 下午9:35 | -- | 文件夹 |
| welcome | 2017年4月22日 下午7:37 | -- | 文件夹 |

图 3-12　quick 目录

(1) bin：Quick 框架相关的可执行脚本，包含项目创建和代码加密等工具。

(2) cocos：Cocos2d-x Lua Binding 对应的 Lua 常量定义文件。

(3) framework：Quick 框架的 Lua 源码。

(4) lib：Quick 框架的 C++源码。

(5) player：模拟器的源码及其工程。

(6) samples：Quick 框架接口用例。

(7) template：项目创建模板。

(8) welcome：模拟器启动后显示的欢迎界面。

其中，framework 是 Quick 的核心，其目录结构如图 3-13 所示。

| 名称 | 修改日期 | 大小 | 种类 |
|---|---|---|---|
| platform | 2018年12月21日 下午11:00 | -- | 文件夹 |
| protobuf | 2019年2月14日 下午10:28 | -- | 文件夹 |
| AppBase.lua | 2017年6月21日 上午7:17 | 1 KB | Lua Source File |
| audio.lua | 2019年1月17日 下午8:26 | 5 KB | Lua Source File |
| crypto.lua | 2014年12月24日 下午7:41 | 5 KB | Lua Source File |
| debug.lua | 2015年10月13日 下午5:49 | 5 KB | Lua Source File |
| device.lua | 2015年10月2日 下午4:35 | 10 KB | Lua Source File |
| display.lua | 2018年12月21日 下午11:00 | 52 KB | Lua Source File |
| functions.lua | 2018年12月21日 下午11:00 | 37 KB | Lua Source File |
| init.lua | 2017年6月27日 下午9:50 | 5 KB | Lua Source File |
| json.lua | 2015年10月13日 下午5:51 | 3 KB | Lua Source File |
| network.lua | 2016年1月23日 下午7:50 | 10 KB | Lua Source File |
| NodeEx.lua | 2019年2月24日 下午9:41 | 10 KB | Lua Source File |
| scheduler.lua | 2014年12月24日 下午7:41 | 4 KB | Lua Source File |
| shortcodes.lua | 2017年5月11日 下午10:03 | 11 KB | Lua Source File |
| SimpleTCP.lua | 2019年2月14日 下午10:28 | 5 KB | Lua Source File |
| toluaEx.lua | 2015年10月13日 下午6:11 | 2 KB | Lua Source File |
| WidgetEx.lua | 2017年7月8日 下午2:48 | 757 字节 | Lua Source File |

图 3-13　framework 目录

(1) platform：Quick 与平台相关的接口。

(2) protobuf：protoc-gen-lua 的 Lua 代码。

(3) AppBase.lua：Lua 程序基类，定义 app 全局变量以及其功能实现。
(4) audio.lua：背景音乐和音效的播放与管理。
(5) crypto.lua：加解密、数据编码库。
(6) debug.lua：提供调试接口。
(7) device.lua：提供设备相关属性的查询，以及设备功能的访问。
(8) display.lua：与显示图像、场景相关的功能。
(9) functions.lua：提供一组常用函数，以及对 Lua 标准库的扩展。
(10) init.lua：Quick 框架的初始化。
(11) json.lua：JSON 的编码与解码。
(12) network.lua：网络接口封装，检查 WiFi 和 3G 网络情况等。
(13) NodeEx.lua：对 cc.Node 的功能扩展。
(14) scheduler.lua：全局计时器、计划任务，该模块在框架初始化时不会自动载入。
(15) shortcodes.lua：一些经常使用的短代码，如设置旋转角度。
(16) SimpleTCP.lua：socket 长链接的抽象管理。
(17) toluaEx.lua：tolua 的功能扩展。
(18) WidgetEx.lua：ccui.Widget 的功能扩展。

### 3.2.3 项目文件结构

以第 2 章环境搭建中创建的 helloworld 项目为例，这里介绍的目录结构如图 3-14 所示。

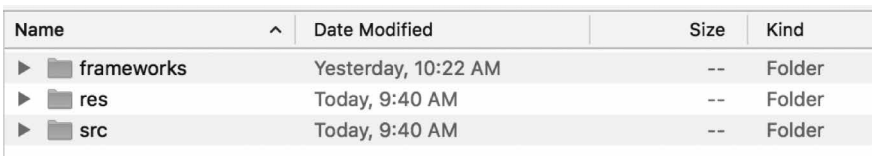



图 3-14 helloworld 目录

(1) frameworks：iOS、Android 等平台的工程文件。
(2) res：项目资源文件夹(图片、音频等)。
(3) src：项目源码存放文件夹(.lua 文件)。
src 文件夹将是游戏逻辑代码的存放位置，如图 3-15 所示。
Quick 框架 src 目录有一定约束，结构说明如下。
(1) app：游戏界面以及逻辑等。
① MyApp.lua：游戏实例，管理整个 App。
② scenes：游戏场景文件夹。
③ MainScene.lua：游戏的第一个场景。
(2) cocos：cocos2d-Lua/quick/cocos/的备份，将随项目一起打包。

| 名称 | 修改日期 | 大小 | 种类 |
|---|---|---|---|
| app | 今天 下午10:16 | -- | 文件夹 |
| MyApp.lua | 2017年4月30日 下午10:22 | 325 字节 | Lua Source File |
| scenes | 今天 下午10:16 | -- | 文件夹 |
| MainScene.lua | 2018年12月21日 下午11:00 | 2 KB | Lua Source File |
| cocos | 2019年2月16日 上午11:46 | -- | 文件夹 |
| config.lua | 2019年2月14日 下午10:28 | 304 字节 | Lua Source File |
| framework | 2019年2月16日 上午11:46 | -- | 文件夹 |
| main.lua | 2018年12月21日 下午11:00 | 324 字节 | Lua Source File |

图 3-15　src 目录

(3) config.lua：工程配置文件，包括分辨率适配等信息。

(4) framework：cocos2d-Lua/quick/framework/的备份，将随项目一起打包。

(5) main.lua：Lua 程序入口。

## 3.3　MVC 框架

### 3.3.1　什么是 MVC

MVC 是模型(model)、视图 (view)和控制器(controller)的缩写。MVC 是一个设计模式，它强制性地使应用程序的输入、处理和输出分开。

(1) Model(模型)：程序员编写程序应有的功能(实现算法等)，数据库专家进行数据管理和数据库设计(可以实现具体的功能)。Model 是应用程序中用于处理应用程序数据逻辑的部分。通常，模型对象负责数据的存储与运算。

(2) View(视图)：界面设计人员进行图形界面设计。View 是应用程序中处理数据显示的部分。通常，视图是依据模型数据创建的。

(3) Controller(控制器)：负责转发请求，对请求进行处理。Controller 是应用程序中处理用户交互的部分。通常，控制器负责从视图读取数据，控制用户输入并向模型发送数据。

**1. 理想 MVC 模型**

MVC 依赖关系如图 3-16 所示。

从依赖关系看，Model 不依赖 View 和 Controller，而 View 和 Controller 依赖 Model。

理想 MVC 关注两个分离：

(1) 从 Model 中分离 View。

(2) 从 View 中分离 Controller。

从 Model 中分离 View，主要基于以下几点考虑。

(1) 不同的关注点：Model 关注内在的不可视的逻辑，而 View 关注外在的可视的逻辑。

(2) 多种表现形式：同一个 Model 往往需要多种 View 表现形式。

(3) 提高可测试性：相对 Model 而言，View 是不容易测试的。

Classic MVC 模型是一种理想化的形式，实际开发中，不可能完全做到 View 和 Controller 的分离。

**2. 真实 MVC 模型**

实际上涉及 UI 的项目，真实的 MVC 模型如图 3-17 所示。

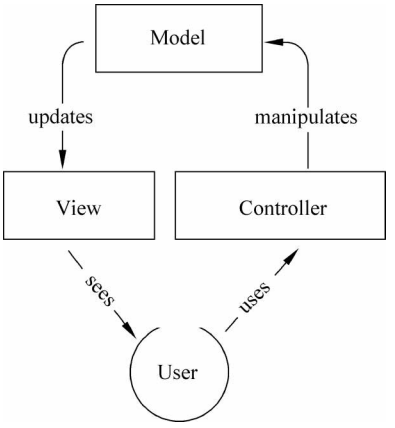



图 3-16 MVC 依赖关系

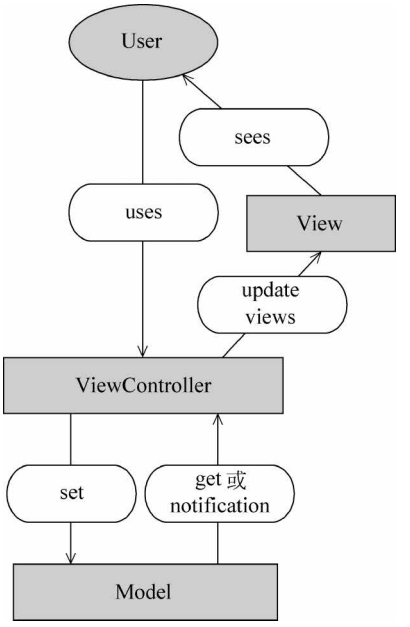



图 3-17 真实 MVC 模型

View 和 Controller 紧密相关，在苹果的 Objective-C 中，定义为 ViewController。ViewController 是业务逻辑实现的核心，负责读取数据显示和处理用户输入等。与界面无关的数据逻辑部分抽象为 Model，如数据库的表更新与读取。界面上独立的显示元素被抽象为 View，如显示文字的 Label 和等待用户单击的 Button 等。

与理想模型不同的是，View 不与 Model 直接交互，View 与 Model 的更新都由 ViewController 完成。

## 3.3.2 Cocos2d-Lua 中的类实现

MVC 是面向对象的一种设计模式，需要类这个机制。前面已经介绍了 Lua 实现面向对象的基本原理，Cocos2d-Lua 结合引擎本身需求，做了完善与扩展，形成了一个叫 class 的函数。

打开 Cocos2d-Lua 引擎下的 quick/framework/functions.lua 文件，可以找到 function class(classname,super)函数。class 函数带有两个参数，其中 classname 表示的是类名，

super 表示的是父类。关于 class 函数的用法可以参考 class 函数的注释，下面主要解析 class 函数的实现。

```
function class(classname, super)
    local superType = type(super)
    local cls

    if superType ~= "function" and superType ~= "table" then
        superType = nil
        super = nil
end
```

这一段主要判断父类的类型。从函数的说明可以看出，super 的类型可以是 nil、function 和 table。如果 super 的类型不为 function 或者 table，那么此时就把父类赋为 nil。当父类为 nil 时，表示类是独立的类，没有父类。

class 的具体实现分两大部分：父类为 C++Object；父类为 table 或 nil。

### 1. 父类为 C++Object

这部分的代码如下：

```
if superType == "function" or (super and super.__ctype == 1) then
    -- inherited from native C++Object
    cls = {}
    if superType == "table" then
        -- copy fields from super
        for k,v in pairs(super) do cls[k] = v end
        cls.__create = super.__create
        cls.super = super
    else
        cls.__create = super
        cls.ctor = function() end
    end

    cls.__cname = classname
    cls.__ctype = 1

    function cls.new(...)
        local instance = cls.__create(...)
        -- copy fields from class to native object
        for k,v in pairs(cls) do instance[k] = v end
        instance.class = cls
        instance:ctor(...)
        return instance
    end
```

第一句判断父类类型：

```
if superType == "function" or (super and super.__ctype == 1) then
```

Cocos2d-Lua 很大一部分接口是 C++Lua Binding 封装过来的，如 cc.Sprite、cc.Layer 等。满足此条件判断，新类直接或间接继承于 C++Lua Binding。

```
if superType == "table" then
    -- copy fields from super
    for k,v in pairs(super) do cls[k] = v end
    cls.__create = super.__create
    cls.super = super
```

如果父类的类型是 table，那么该父类一定是继承于 function 的一个子类，程序会直接把父类中的方法用复制的方式赋值给当前类，同时设置当前类的 super 属性为父类。

```
else
    cls.__create = super
    cls.ctor = function() end
end
```

如果传入的父类类型是 function，那么程序会把当前类的创建函数设置成传入的函数。

在上面的操作完成之后，程序会把类名等参数赋给类，设置类的__ctype 属性为 1，并且创建一个 new()函数方便类的实例化。

```
cls.__cname = classname
cls.__ctype = 1

function cls.new(...)
    local instance = cls.__create(...)
    -- copy fields from class to native object
    for k,v in pairs(cls) do instance[k] = v end
    instance.class = cls
    instance:ctor(...)
    return instance
end
```

在实际应用中，如果父类是引擎封装的 cc.Xxx，则必须用 function 方式实现继承。例如，下面新建一个 MySpriteClass 的类：

```
local MySpriteClass = class("MySpriteClass", function(pic)
    return cc.Sprite:create(pic)
end)
return MySpriteClass
```

通过 className.new()函数创建实例：

```
local mySprite = MySpriteClass.new("xx.png")
```

如果 MySpriteClass 作为父类被继承，则可以不用 function 方式。其原因是 MySpriteClass

的__ctype属性为1,能被正确地继承。例如,下面新建一个OtherSpriteClass的类:

```
local OtherSpriteClass = class("OtherSpriteClass", MySpriteClass)
return OtherSpriteClass
```

### 2. 父类为table或nil

接下来考虑父类为Lua object的部分。实际上,该部分才是传统的Lua table继承。代码如下:

```
else
    -- inherited from Lua Object
    if super then
        cls = {}
        setmetatable(cls, {__index = super})
        cls.super = super
    else
        cls = {ctor = function() end}
    end

    cls.__cname = classname
    cls.__ctype = 2 -- lua
    cls.__index = cls

    function cls.new(...)
        local instance = setmetatable({}, cls)
        instance.class = cls
        instance:ctor(...)
        return instance
    end
end
```

首先,判断父类是否为空,如果不为空,那么就将子类的metatable的__index赋值为父类,并且将子类的super设置成传入的父类对象。接下来再将子类的__index赋值为自己,这样就能同时访问父类和自己的方法。可以看到这里和前面Lua面向对象的部分一致。最后创建一个new函数方便类的实例化。

### 3. 两种继承的异同

前面一直未解答为什么会有两种继承实现方式,下面就比较一下它们之间的异同。首先做个测试:

```
print(type(cc.Node))
print(type(cc.Node:create()))

--> table
--> userdata
```

结果说明,cc.Node是一个table,但它创建出来的实例对象是userdata。userdata用来

描述应用程序或者使用 C 实现的库创建的新类型，Cocos2d-x 的 Lua API 创建的对象实例都是 userdata。

cc. Node 是不能直接继承的，它只是一个普通的 table 变量，里面存放了相应 userdata 的实例化函数，经过 cc. Node：create()创建的实例 userdata 才是引擎渲染用的节点对象。这类 userdata 节点需要使用 class 函数的 C++object 的继承逻辑。

游戏中的一些数据模块，可以直接用 class("dataModule")创建纯粹的 Lua 类，它们完全遵从 Lua 的 metatable 继承方式，使用 class 函数的 table 继承逻辑。

由于引擎 C++object binding 与原生 Lua table 的不同，需要两种继承方式。

这两种类的继承实现方式派生出来的类(table)，成员上略有异同，如图 3-18 所示。

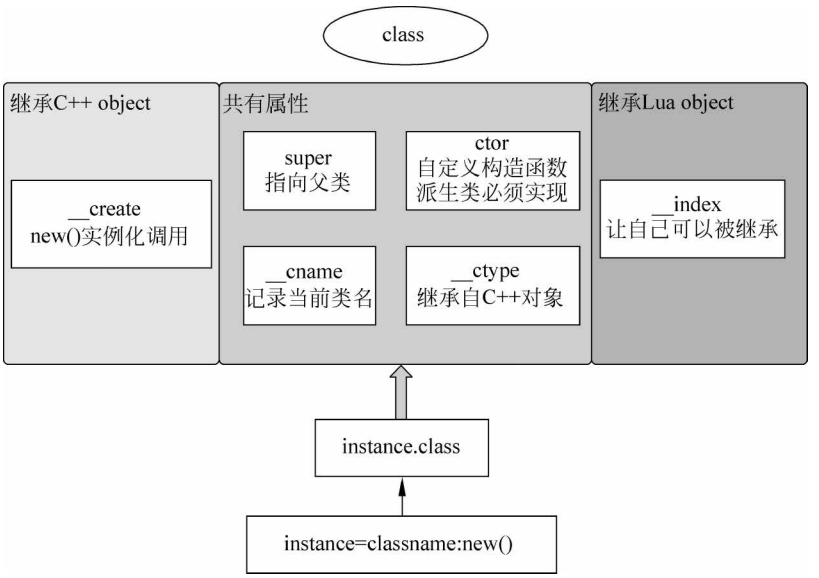



图 3-18 quick class

由 classname. new( ) 生成的 instance 包含了一个 class 属性。class 指向了类原型，并具有 super、ctor、__ cname 和__ ctype 4 个属性。

继承自 C++的类，new 方法使用__ create()函数创建实例，继承自 Lua 的类 new 方法使用{}创建实例。

### 3.3.3 Cocos2d-Lua 中的 MVC 实现

MVC 的实现在 quick/framework/AppBase. lua 中。本质上，MVC 是一个编码习惯的约定，并不能强制用户必须写 MVC 风格的代码，但遵循 MVC 规则可以让代码更易于维护。

AppBase. lua 中定义了一个 AppBase 基类，作为所有 Quick 游戏的 Lua 入口，由 player3 新建的工程有一个 MyApp. lua，它就是继承于 AppBase 类。AppBase 的功能类似

于 iOS 工程中的 AppDelegate.m 实现的功能，具体有下面几个功能：

（1）程序前后台切换事件的接收与分发。

（2）为 framework 提供程序退出接口。

（3）提供创建 scene 和 view 的接口，并约定它们的存放路径。

这里需要重点分析一下第 3 条，为了直观理解，看下一个引擎自带 test 的代码结构。进入目录 quick/samples/test/src/，结构如图 3-19 所示。

| 名称 | 修改日期 | 大小 | 种类 |
|---|---|---|---|
| app | 2017年6月24日 下午3:35 | -- | 文件夹 |
| MyApp.lua | 2017年4月30日 下午10:22 | 325 字节 | Lua Source File |
| scenes | 2018年12月21日 下午11:00 | -- | 文件夹 |
| BaseLayer.lua | 2017年7月6日 下午10:49 | 630 字节 | Lua Source File |
| MainScene.lua | 2018年12月21日 下午11:00 | 2 KB | Lua Source File |
| views | 2018年12月21日 下午11:00 | -- | 文件夹 |
| Test_AccelerometerEvent.lua | 2018年12月21日 下午11:00 | 802 字节 | Lua Source File |
| Test_Audio.lua | 2018年12月21日 下午11:00 | 3 KB | Lua Source File |
| Test_CocosStudio.lua | 2018年12月21日 下午11:00 | 1 KB | Lua Source File |
| Test_KeypadEvent.lua | 2017年5月1日 下午10:14 | 640 字节 | Lua Source File |
| Test_NodeEvent.lua | 2017年5月1日 下午9:59 | 1 KB | Lua Source File |
| Test_NodeFrameEvent.lua | 2017年5月1日 下午10:11 | 607 字节 | Lua Source File |
| Test_NodeTouchEvent.lua | 2017年6月24日 下午3:15 | 1 KB | Lua Source File |
| config.lua | 2019年2月14日 下午10:28 | 304 字节 | Lua Source File |
| main.lua | 2018年12月21日 下午11:00 | 324 字节 | Lua Source File |

图 3-19　test 目录结构

深入 AppBase.lua 的代码可以看到，scenes 和 views 这两个文件夹名是被固定了的。scenes 下存放场景文件，views 下存放自定义控件。只有放在这两个目录下，AppBase 的 enterScene 和 createView 才能起作用。

## 3.4　基础概念

Cocos2d-Lua 是一款基于节点树渲染的游戏引擎，并且它将游戏的各个部分抽象成导演、场景、层和精灵等概念。游戏中每个时刻都有一个场景在独立运行，通过切换不同的场景完成整个游戏流程，场景切换的管理由导演执行。它们之间的关系如图 3-20 所示。

一个游戏可以有多个不同的游戏场景，每个场景又可包含多个不同的层，每层可拥有任意个游戏节点（常见是精灵，但也可以是层、菜单和文本等）。

### 3.4.1　导演

一款游戏好比一部电影，它们的基本原理都是一样的，只是游戏具有更强的交互性，所以 Cocos2d-Lua 中把统筹游戏大局的类抽象为导演类（Director）。

导演类是游戏的组织者和领导者，是整个游戏的导航仪和总指挥。它主要负责以下工作：

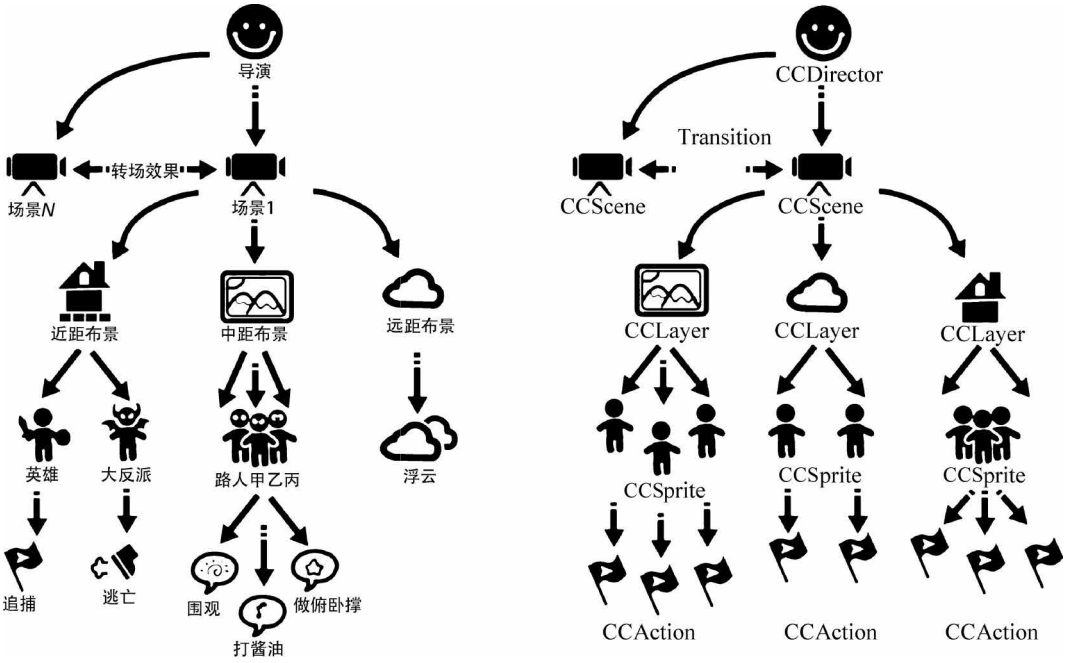



图 3-20　导演

(1) 在开始游戏和结束游戏时游戏的初始化和销毁工作。

(2) 游戏场景的切换,以及场景暂停或恢复的控制。

(3) 设置或获取一些系统信息,如调整 OpenGL 相关的设置和得到屏幕大小等。

还需要说明的是,Director 使用单例模式实现,也就是说一个游戏中只有一个导演。Cocos2d-Lua 中使用 cc.Director 表示导演类,通过下面的方法获取单例对象:

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

下面介绍导演常用方法。

(1) 获取窗口大小。

```
local winSize = sharedDirector:getWinSize()
```

(2) 暂停游戏。暂停所有计时器和动作,但场景仍然会显示在屏幕上。

```
sharedDirector:pause()
```

(3) 恢复游戏。恢复所有计时器和动作。

```
sharedDirector:resume()
```

(4) 退出游戏。

```
sharedDirector:endToLua()
```

(5) 捕捉屏幕。

```
sharedDirector:captureScreen()
```

### 3.4.2 节点

节点(Node)是 Cocos2d-Lua 中可见元素的基础类,场景、层、精灵、标签、菜单等都是继承自 Node。

Node 作为可见元素的节点公共抽象,主要功能如下:

(1) 封装了可见元素的基础属性与方法。

(2) 节点都可以含有子节点。

(3) 节点可以运行动作。

(4) 节点内部有一个跟随节点生命周期的调度器。

Cocos2d-Lua 中使用 cc.Node 表示节点类。下面是 Quick 框架创建节点的方法:

```
local node = display.newNode()
```

下面介绍节点常用方法。

(1) 添加子节点。

```
node:addChild(child, zOrder, name)
```

参数说明:

① child,cc.Node 类型,子节点。

② zOrder,数字类型,可选参数。子节点的 zOrder 将影响渲染顺序。

③ name,字符串类型,可选参数。子节点的名称。

(2) 移除子节点。

```
node:removeChild(childNode)
```

(3) 获取父节点。

```
local parent = node:getParent()
```

(4) 获取子节点。

```
local children = node:getChildren()
```

(5) 设置大小。

```
node:setContentSize(cc.size(width, height))
```

(6) 获取大小。

```
local size = node:getContentSize()
```

（7）设置位置坐标。

```
node:setPosition(cc.p(x, y))
```

（8）设置锚点。

```
node:setAnchorPoint(cc.p(x, y))
```

在 Cocos2d-x 中通过节点树渲染场景，场景是所有节点的根节点，如图 3-21 所示。

这是一个场景的节点树，其中树干是场景，树枝是层，树叶是节点（精灵、文本、按钮等）。

引擎渲染一个场景的时候，从树根往上渲染，底层的节点图像将被上层节点的图像覆盖。


图 3-21　节点树

### 3.4.3　场景

场景（Scene）是容纳其他可见与不可见元素的容器，一个游戏至少需要一个场景。特定时间内只有一个场景是处于活动状态的。游戏里关卡、界面的切换其实就是一个一个场景之间的切换，就像在电影中变换舞台或场地一样。

Cocos2d-Lua 中使用 cc.Scene 表示场景类。下面是 Quick 框架创建场景的方法。

（1）创建普通场景。

```
display.newScene(name)
```

（2）创建带物理引擎的场景。

```
display.newPhysicsScene(name)
```

前面提到，场景的管理是由导演完成的，与场景相关的导演方法有如下几个。

（1）sharedDirector：runWithScene(newScene) 运行游戏的第一个场景。本方法在主程序第一次启动主场景的时候调用。

（2）sharedDirector：pushScene(scene)将当前运行中的场景暂停并压入场景栈中，再将传入的 scene 设置为当前运行场景。

（3）sharedDirector：replaceScene(newScene)使用传入的 scene 替换当前场景，当前场景被释放。这是最常用的场景切换方法。

（4）sharedDirector：popScene() 从场景栈弹出栈顶场景并释放，运行新的栈顶场景。如果栈为空，则结束应用。该方法通常与 pushScene 结对使用。

### 3.4.4　层

层（Layer）是对场景内布局的细分，尽管有 ColorLayer，但层主要起容器作用。假如一个跑酷场景，地图和角色不停地向前滚动，但 UI 却固定不动，这时如果做分层处理，将能更

方便地实现它，即地图和角色放在一个层，UI 放在一个层，互相不影响。

Cocos2d-Lua 中使用 cc.Layer 表示层类。下面是 Quick 框架创建层的方法。

(1) 创建一个普通的层。

```
local layer = display.newLayer()
```

(2) 创建一个有背景填充色的层。

```
local layer = display.newColorLayer(cc.c4b(255,0,0,255))
```

**注**：层作为容器是一个历史遗留产物，在 Cocos2d-x 2.x 版本中，只有层可以获取触摸事件，层作为一个必须的抽象类存在。然而在 Quick 中，所有节点(Node)都可获取触摸事件，层的容器意义已经被节点取代。

### 3.4.5 精灵

如果说场景和层主要起容器的作用，那么精灵(Sprite)是容器中盛放的内容。精灵总绑定一个纹理对象或精灵帧对象，引擎渲染精灵实际上是把精灵绑定的纹理或精灵帧按照属性设定渲染到屏幕上。

可以说，精灵是图像的载体，游戏中看得见的场景如背景图片、房屋、敌人、玩家角色及子弹等，都可以通过精灵实现。

引擎中纹理对应类 Texture2D，精灵帧对应类 SpriteFrame，它们与精灵之间的关系，如图 3-22 所示。

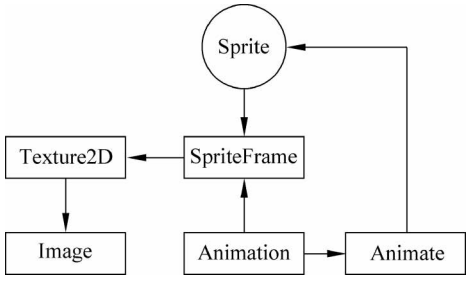


图 3-22 精灵

(1) Image 对应硬盘中不同类型的图片，如 jpg 和 png 等。它知道如何从文件中读取不同类型的图片，生成缓冲数据，供 Texture2D 使用。

(2) Texture2D 代表一个可以被绘制的纹理。

(3) SpriteFrame 的概念是相对于动画而产生的。一个 Sprite 可以拥有多个 SpriteFrame，一个时刻只显示其中一帧，帧之间切换就形成了动画。SpriteFrame 是具有一定区域属性的纹理，依赖于 Texture2D。

(4) Animation 描述一个有序的 SpriteFrame 系列，类似于剪辑过的电影胶卷。

(5) Animate 把 Animation 转化为引擎识别的 Action 类，让 Sprite 可以运行。

Cocos2d-Lua 中使用 cc.Sprite 表示精灵类。下面是 Quick 框架提供的三种创建精灵的方法。

(1) 从图片文件创建。

```
local sprite1 = display.newSprite("hello1.png")
```

(2) 从缓存的图像帧创建。

```
local sprite2 = display.newSprite("#frame0001.png")
```

图像帧的名字就是图片文件名,为了和图片文件名区分,在文件名前添加“#”以表示精灵帧。用“#”表示精灵帧是 Quick 的 display 模块封装的特性,cc.Sprite:create()并不具备这样的特性。

(3) 从 SpriteFrame 对象创建精灵。

```
local frame = display.newFrame("frame0002.png")
local sprite3 = display.newSprite(frame)
```

下面是一个结合层和精灵的示例。

```
function MainScene:ctor()
    local layer = display.newColorLayer(cc.c4b(255,0,0,255))     -- 创建 layer
    self:addChild(layer)                                         -- 添加 layer 到场景
    local sprite = display.newSprite("HelloWorld.png")           -- 创建 sprite
    layer:addChild(sprite,0)                                     -- 添加到 layer
end
```

## 3.5 坐标系

不论在游戏开发中还是在应用开发中,坐标系都是一个比较重要的概念。理解好 Cocos2d-Lua 中的坐标系概念,才能在游戏开发中正确地设置节点的位置坐标。

首先,回顾一下笛卡儿坐标系的概念。Cocos2d-Lua 的底层渲染使用的是 OpenGL,所以它的坐标系和 OpenGL 相同,而 OpenGL 的坐标系来源于高中物理所学的笛卡儿坐标系。

### 3.5.1 笛卡儿坐标系

OpenGL 坐标系为笛卡儿右手系。笛卡儿右手系定义原点在左下角,$x$ 向右,$y$ 向上,$z$ 向外。

向前伸出右手,让拇指和食指成 L 形,大拇指向右,食指向上,其余的手指指向你,这样就建立了一个右手坐标系。拇指、食指和其余手指分别代表 $x$、$y$、$z$ 轴的正方向。如图 3-23 所示,即高中所学的笛卡儿右手系。

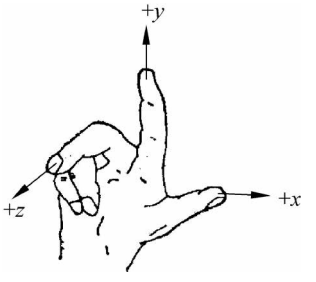


图 3-23 笛卡儿右手系

### 3.5.2 屏幕坐标系和 Cocos2d-Lua 坐标系

上面介绍了笛卡儿坐标系的右手系,现在介绍 OpenGL 坐标系。OpenGL 坐标系是一个三维坐标系,使用的就是**笛卡儿坐标系的右手系**。那么,OpenGL 坐标系与 Cocos2d-Lua 的

坐标系又有什么联系呢？为了更好地理解 Cocos2d-Lua 的坐标系概念，下面来看一个新的概念——屏幕坐标系。

在 iOS、Android 和 Windows Phone 应用开发中，通常以屏幕的左上角为坐标原点，$x$ 轴向右逐渐增大，$y$ 轴向下逐渐增大。有过 iOS 开发经验的都会知道，在 iOS 中，各种控件的位置都是按照这个坐标系摆放及进行数值计算的。这种坐标系称作标准屏幕坐标系。

Cocos2d-Lua 坐标系与标准屏幕坐标系不同，Cocos2d-Lua 坐标系原点为屏幕左下角（多分辨率适配加入后，原点的定义应该是设计分辨率的左下角），$x$ 向右，$y$ 向上。然而，Cocos2d-Lua 作为 2D 游戏引擎，它的坐标系也应该是二维的，那么它的 $z$ 轴有什么意义呢？后面的 ZOrder 中再详细介绍。

图 3-24 表明了屏幕坐标系和 Cocos2d-Lua 坐标系的区别。


图 3-24 Cocos2d-Lua 坐标系与屏幕坐标系

### 3.5.3 世界坐标系和本地坐标系

世界坐标系（World Coordinate）也叫作**绝对坐标系**，是游戏引擎中的抽象概念。它是原点固定在屏幕左下角的 Cocos2d-Lua 坐标系的别称。

本地坐标系（Local Coordinate）也叫**相对坐标系**，是与节点相关联的坐标系。每个节点都有独立的坐标系，以节点的左下角为原点，遵循 OpenGL 坐标系。当节点移动或改变方向时，与该节点关联的坐标系将随之移动或改变方向。

Cocos2d-Lua 中的元素是有父子关系的层级结构，通过 Node 的 setPosition 设定的是当前节点在父节点坐标系上的坐标，即相对坐标。

**注**：一个节点的父节点，是在 addChild 时才决定的。

### 3.5.4 锚点

将一个节点添加到父节点里面时，需要设置其在父节点上的位置，本质上是将节点的锚点（Anchor Point）设置在父节点坐标系上的位置。锚点同时也是旋转一个节点时的中心点，如图 3-25 所示。

锚点的值是节点宽、高的比例因子，它有以下特性。

（1）锚点的两个参数都在 0～1。它们表示的并不是像素点，而是乘数因子。（0.5，0.5）表示锚点位于节点长度乘 0.5 和宽度乘 0.5 的地方，即节点的中心。

（2）在 Cocos2d-Lua 中，Layer 的锚点为默认值（0，0），Sprite 的默认值为（0.5，0.5）。

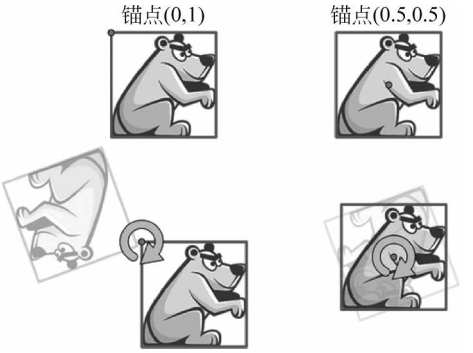



图 3-25　锚点

为了更清楚地展示锚点，下面做两个测试。

(1) 添加一个红色层放在屏幕左下角，再添加一个绿色层在红色层之上。

```
local red = cc.LayerColor:create(cc.c4b(255, 100, 100, 128))
red:setContentSize(display.width / 2, display.height / 2)
local green = cc.LayerColor:create(cc.c4b(100, 255, 100, 128))
green:setContentSize(display.width / 4, display.height / 4)
red:addChild(green)
self:addChild(red)
```

运行结果如图 3-26 所示。


图 3-26　AnchorPoint demo1

(2) 在上面例子的基础上,将红色层的锚点设为(0.5,0.5),绿色层的锚点设为(1,1)。

```
local red = cc.LayerColor:create(cc.c4b(255, 100, 100, 128))
red:setContentSize(display.width / 2, display.height / 2)
red:ignoreAnchorPointForPosition(false)
red:setAnchorPoint(0.5, 0.5)
red:setPosition(display.width / 2, display.height / 2)

local green = cc.LayerColor:create(cc.c4b(100, 255, 100, 128))
green:setContentSize(display.width / 4, display.height / 4)
green:ignoreAnchorPointForPosition(false)
green:setAnchorPoint(1, 1)

red:addChild(green)
self:addChild(red)
```

ignoreAnchorPointForPosition()接口让节点接受 setAnchorPoint 对锚点的修改,在默认情况下,这个值都是 false,无须修改。

运行结果如图 3-27 所示。


图 3-27 AnchorPoint demo2

通过两个例子的比较,锚点和相对坐标的概念应该更加清晰。

### 3.5.5 忽略锚点

Ignore Anchor Point 全称是 ignoreAnchorPointForPosition，作用是将锚点固定在一个地方。如果设置其值为 true，则节点的 Anchor Point 固定为左下角，否则为默认值或用户设定值。

在上一个例子中，把 Layer 的 ignoreAnchorPointForPosition 设置为 true，如下：

```
red:ignoreAnchorPointForPosition(true)
...
green:ignoreAnchorPointForPosition(true)
```

锚点设置将不起作用，运行结果如图 3-28 所示。


图 3-28 忽略锚点

### 3.5.6 ZOrder 与渲染顺序

Cocos2d-Lua 开发 2D 游戏，z 轴的值并不影响物体显示在屏幕上的远近，它只与渲染顺序有关，z 轴值小的 Node 最先被渲染。

Cocos2d-Lua 中，Node 有两个 ZOrder 值如下。

(1) LocalZOrder 是一个父节点的兄弟节点之间排序的 key 值，决定它们在 Scene Graph 渲染树上的顺序。

① 如果两个节点的 LocalZOrder 相同，那么先加入的节点先被渲染。

② Scene Graph 使用 In-Order 算法(http://en.wikipedia.org/wiki/Tree_traversal#In-order)。

(2) GlobalZOrder 是绕过 Scene Graph 渲染树直接提升渲染等级的一个入口。GlobalZOrder 依然是值小的先渲染。

① 当 GlobalZOrder 为 0,节点遵循 Scene Graph 渲染。

② 在默认情况下,所有节点的 GlobalZOrder 初始化为 0。

③ 如果两个节点的 GlobalZOrder 不为 0 且相同,那么渲染顺序不可预期。

④ SpriteBatchNode 不支持 GlobalZOrder。

**注**:通常情况下,不建议使用 GlobalZOrder。

设置 LocalZOrder 的相关接口如下:

```
Node:addChild(child, zorder, tag)
Node:add(child, zorder, tag)
Node:addTo(target, zorder, tag)
Node:zorder(z)
Node:setLocalZOrder(z)
```

设置 GlobalZOrder 的相关接口如下:

```
Node:setPositionZ(zorder)
Node:setGlobalZOrder(zorder)
```

## 3.6 文本标签

在游戏中,文字占有很重要的位置,游戏的介绍、提示和对话等都需要使用到文字。Cocos2d-Lua 在文字渲染方面提供了非常灵活的机制,既可以直接使用系统字体,也可以使用自定义字体。

**注**:Cocos2d-Lua 内部以 UTF-8 格式处理字符串,源代码文件必须使用 UTF-8 格式存储。

### 3.6.1 TTF 文本标签

TTF(True Type Font)是一种字库规范,是苹果公司和 Microsoft 公司共同推出的字体文件格式。随着 Windows 的流行,TTF 变成最常用的一种字体文件格式。

TTF 类型的文本标签是通过系统 TTF 字体渲染文字,它使用简单,支持任意字体大小和字距的文本。

TTF 文本标签创建后,如果修改文本标签内容,则会重新创建一个新的 OpenGL 纹理,就跟重新创建一个新的 TTF 文本标签一样。这意味着它的刷新效率不高。在文本标签内容需要经常更新的应用场景,推荐使用 BMFont 文本标签。

#### 1. display 中的 TTF 接口

TTF 文本标签的创建分为系统 TTF 和外置 TTF 两种。所谓系统 TTF,就是手机或计算机操作系统自带的 TTF 字体,而外置 TTF 就是把 ttf 后缀的字体文件放在游戏的 res

目录下的 TTF 字体。

Quick 的 display.newTTFLabel(params)对两种 TTF 创建进行了统一封装与简化。

参数 params 为 table 类型,有以下可选域。

(1) text:显示的文本。

(2) font:字体名,如果是外置 TTF 字体,那么指定为字体路径名。

(3) size:字体大小,因为是 TTF 字体,所以可以任意指定尺寸。

(4) color:文字颜色(可选),用 cc.c3b()创建颜色,默认为白色。

(5) align:文字的水平对齐方式(可选),仅在指定了 dimensions 参数时有效。

(6) valign:文字的垂直对齐方式(可选),仅在指定了 dimensions 参数时有效。

(7) dimensions:文本显示区域的尺寸(可选),用 cc.size ()创建。

(8) x:x 坐标(可选)。

(9) y:y 坐标(可选)。

其中,align 参数的可用值有如下几个。

(1) cc.TEXT_ALIGNMENT_LEFT,水平左对齐。

(2) cc.TEXT_ALIGNMENT_CENTER,水平居中对齐。

(3) cc.TEXT_ALIGNMENT_RIGHT,水平右对齐。

而 valign 参数可用的值有如下几个。

(1) cc.VERTICAL_TEXT_ALIGNMENT_TOP,垂直顶部对齐。

(2) cc.VERTICAL_TEXT_ALIGNMENT_CENTER,垂直居中对齐。

(3) cc.VERTICAL_TEXT_ALIGNMENT_BOTTOM,垂直底部对齐。

创建 display.newTTFLabel 文本标签示例代码如下。

```
-- 创建系统 TTF 文本标签
local ttfLabelSys = display.newTTFLabel({
    text = "display.newTTFLabel sys TTF",
    font = "",
    size = 30,
    align = cc.TEXT_ALIGNMENT_CENTER,
    x = display.cx,
    y = display.cy + 400
})
self:addChild(ttfLabelSys)

-- 创建外置 TTF 文本标签
local ttfLabelSrc = display.newTTFLabel({
    text = "display.newTTFLabel font in src",
    font = "CreteRound - Italic.ttf",
    size = 30,
    align = cc.TEXT_ALIGNMENT_CENTER,
    x = display.cx,
    y = display.cy + 350
```

```
})
self:addChild(ttfLabelSrc)
```

通过调用 label：setString(newStr)，可以更新文本标签显示的文字。

**2. cc 中的 TTF 接口**

cc.Label 是引擎提供的底层的文本标签接口，display 和 ccui.Text 都是基于它进行的二次封装或扩展。

系统 TTF 字体创建函数为 cc.Label：createWithSystemFont(text，font，size，dimensions，halign，valign)。

参数说明如下。

(1) text：显示的文本。

(2) font：系统字体名称。

(3) size：字体大小，因为是 TTF 字体，所以可以任意指定尺寸。

(4) dimensions：文本显示区域的尺寸(可选)，同 display.newTTFLabel。

(5) halign：文字的水平对齐方式(可选)，同 display.newTTFLabel。

(6) valign：文字的垂直对齐方式(可选)，同 display.newTTFLabel。

创建 cc.Label：createWithSystemFont 文本标签示例代码如下。

```
cc.Label:createWithSystemFont("cc.Label sys font", "", 30,
    cc.size(0, 0), cc.TEXT_ALIGNMENT_CENTER,
    cc.VERTICAL_TEXT_ALIGNMENT_CENTER)
    :addTo(self)
    :pos(display.cx, display.cy + 200)
    :setTextColor(cc.c4b(255, 0, 0, 255)) -- 颜色字体颜色
```

外置 TTF 字体创建函数为 cc.Label：createWithTTF (text，font，size，dimensions，halign，valign)。

参数说明如下。

(1) text：显示的文本。

(2) font：外置字体路径。

(3) size：字体大小，因为是 TTF 字体，所以可以任意指定尺寸。

(4) dimensions：文本显示区域的尺寸(可选)，同 display.newTTFLabel。

(5) halign：文字的水平对齐方式(可选)，同 display.newTTFLabel。

(6) valign：文字的垂直对齐方式(可选)，同 display.newTTFLabel。

创建 cc.Label：createWithTTF 文本标签示例代码如下。

```
cc.Label:createWithTTF("cc.Label TTF font", "CreteRound-Italic.ttf", 30,
    cc.size(0, 0), cc.TEXT_ALIGNMENT_CENTER,
    cc.VERTICAL_TEXT_ALIGNMENT_CENTER)
    :addTo(self)
    :pos(display.cx, display.cy + 150)
```

```
    :setColor(cc.c4b(255, 0, 0, 255)) -- 设置字体颜色
```

**注**：cc.Label 创建的系统 TTF 文本和外置 TTF 文本，在设置字体颜色时所使用的接口不一样，而 display.newTTFLabel 在创建文本时的颜色参数是统一的。

**3. ccui 中的 TTF 接口**

ccui.Text 与 Cocos Studio 编辑器中的 Label 控件对应，是在 cc.Label 上进行的扩展。

TTF 字体创建函数为 ccui.Text：create(text,font,size)，同 display.newTTFLabel 一样，不区分系统 TTF 和外部 TTF。

参数说明如下。

(1) text：显示的文本。

(2) font：外置字体路径。

(3) size：字体大小，因为是 TTF 字体，所以可以任意指定尺寸。

创建 ccui.Text：create 文本标签示例代码如下。

```
-- 创建系统 TTF
ccui.Text:create("ccui.Text sys TTF", "", 30)
    :addTo(self)
    :pos(display.cx, display.cy - 50)
-- 创建外置 TTF
ccui.Text:create("ccui.Text TTF font", "CreteRound-Italic.ttf", 30)
    :addTo(self)
    :pos(display.cx, display.cy - 100)
```

## 3.6.2 BMFont 文本标签

使用 BMFont 字体的文本标签，支持 FNT 类型的文件，它适用于需要频繁更新内容的文本标签。BMFont 文本标签使用图片显示文本，所有字符都被整合到一张纹理图片上，通过纹理坐标控制字符串的显示，每次更新只是改变纹理坐标而不用新建纹理，这有利于性能优化；同时图片可以使用美术设计的艺术字体，为游戏展现力提供强有力的支撑。

在使用 BMFont 文本标签之前，需要添加字体文件到项目工程的 res 目录下，包括一个图片文件(xxx.png)和一个字体坐标文件(xxx.fnt)。fnt 文件中包含了对应图片的名字(图片包含了所有要绘制的字符)、图片中的字符对应的 Unicode 编码、字符在图片中的坐标、宽和高等信息。字体文件需要使用专门的编辑工具制作，如 Glyph Designer、Hiero 和 BMFont 等，本节随后将对字体制作软件进行介绍。

font-issue1343-hd.fnt　　font-issue1343-hd.png

图 3-29　BMFont

典型的字体文件如图 3-29 所示。

### 1. display 中的 BMFont 接口

BMFont 文本标签使用 display.newBMFontLabel(params)新建。其中,参数 params 是 table 类型,可用参数如下。

(1) text:要显示的文本。

(2) font:字体文件名。

(3) align:文字的水平对齐方式(可选)。

(4) maxLineWidth:最大行宽(可选)。

(5) offsetX:图像的 X 偏移量(可选)。

(6) offsetY:图像的 Y 偏移量(可选)。

(7) x:x 坐标(可选)。

(8) y:y 坐标(可选)。

创建 display.newBMFontLabel 文本标签示例代码如下。

```
local labelbmf = display.newBMFontLabel({
    text = "Hello",
    font = "helvetica-32.fnt",
})
```

### 2. cc 中的 BMFont 接口

cc 层 BMFont 文本标签创建函数为 cc.Label:createWithBMFont (font,text,halign,maxLineWidth,imageOffset)。

参数说明如下。

(1) font:fnt 字体文件路径。

(2) text:要显示的文本。

(3) halign:文字的水平对齐方式(可选)。

(4) maxLineWidth:最大行宽(可选)。

(5) imageOffset:cc.p()类型,图像的 X、Y 偏移量(可选)。

创建 cc.Label:createWithBMFont 文本标签示例代码如下。

```
cc.Label:createWithBMFont("helvetica-32.fnt", "cc.Label bmfont",
    cc.TEXT_ALIGNMENT_LEFT, 0, cc.p(0, 0))
    :addTo(self)
    :pos(display.cx, display.cy + 100)
```

### 3. ccui 中的 BMFont 接口

ccui.TextBMFont 与 Cocos Studio 编辑器中的 BitmapLabel 控件对应,是在 cc.Label 上进行的扩展。

ccui 层 BMFont 文本标签创建函数为 ccui.TextBMFont:create(text,font)。

参数说明如下。

(1) text:要显示的文本。

(2) font：字体文件名。

创建 ccui.TextBMFont：create 文本标签示例代码如下。

```
ccui.TextBMFont:create("ccui.TextBMFont", "helvetica-32.fnt")
    :addTo(self)
    :pos(display.cx, display.cy - 150)
```

### 3.6.3　图集文本标签

除了 TTF 和 BMFont 文本标签，引擎还支持一种简单的图集文本标签。图集文本标签是由一连串等宽图片拼接成的一张整图，这些图片中符号的 ASCII 码是连续的，如图 3-30 所示。


图 3-30　AtlasLabel

图集文本标签通常是 0～9 这几个数字，在游戏中用来显示资源数量等。

**1. cc 中的图集文本标签**

cc 中的图集文本标签创建函数为 cc.Label：createWithCharMap（charMapFile，itemWidth，itemHeight，startCharMap）。

参数说明如下。

(1) charMapFile：图集图片的路径。

(2) itemWidth：一个字符图片的宽。

(3) itemHeight：一个字符图片的高。

(4) startCharMap：图集的第一个字符的 ASCII 码。

创建 cc.Label：createWithCharMap 图集文本标签示例代码如下。

```
local labelCM = cc.Label:createWithCharMap("number.png", 19, 35,
    string.byte(0))
    :addTo(self)
    :pos(display.cx, display.cy + 50)
```

由于创建接口不提供默认显示文本的参数，因此需要通过 setString()设置需要显示的文本。

```
labelCM:setString("9876543210")
```

**注**：显示的文本必须在图集中存在。

**2. ccui 中的图集文本标签**

ccui.TextAtlas 与 Cocos Studio 编辑器中的 AtlasLabel 控件对应，是在 cc.Label 上进行的扩展。

ccui 中的图集文本标签创建函数为 ccui.TextAtlas：create（text，charMapFile，itemWidth，itemHeight，startChar）。

参数说明如下。

(1) text：要显示的文本。

(2) charMapFile：图集图片的路径。

(3) itemWidth：一个字符图片的宽。

(4) itemHeight：一个字符图片的高。

(5) startChar：图集的第一个字符。

创建 ccui.TextAtlas：create 图集文本标签示例代码如下。

```
ccui.TextAtlas:create("9898964345", "number.png", 19, 35, "0")
    :addTo(self)
    :pos(display.cx, display.cy - 200)
```

### 3.6.4 Mac 下使用 Glyph Designer 制作字体

Glyph Designer 是一款 Mac 位图字体生成工具，它能读取系统的 TrueType，生成 Cocos2d-Lua 支持的 fnt 位图字体格式。官方网站为 https://71squared.com/glyphdesigner。

使用步骤如下：

(1) 启动 Glyph Designer，选择 File→New 命令，在左上的搜索框中输入需要的字体集名(这里使用 helvetica)。

(2) 设置字体尺寸为 32，默认情况下，Glyph Designer 自动调整字体图集尺寸为最小可能值以适配所有可能的图像。

(3) 在右边 Glyph Fill 里面选择颜色。

(4) 在 Included Glyphs 里面单击 NEHE 按钮，然后在区域内输入所要用到的字符。

(5) 单击 Export 按钮导出文件。

(6) 选择导出文件类型。

步骤图解如图 3-31 所示。

### 3.6.5 Windows 下使用 BMFont 制作字体

在 Windows 中，最常用的字库图集制作工具是 BMFont。官方网站为 http://www.angelcode.com/products/bmfont/。

使用方法如下：

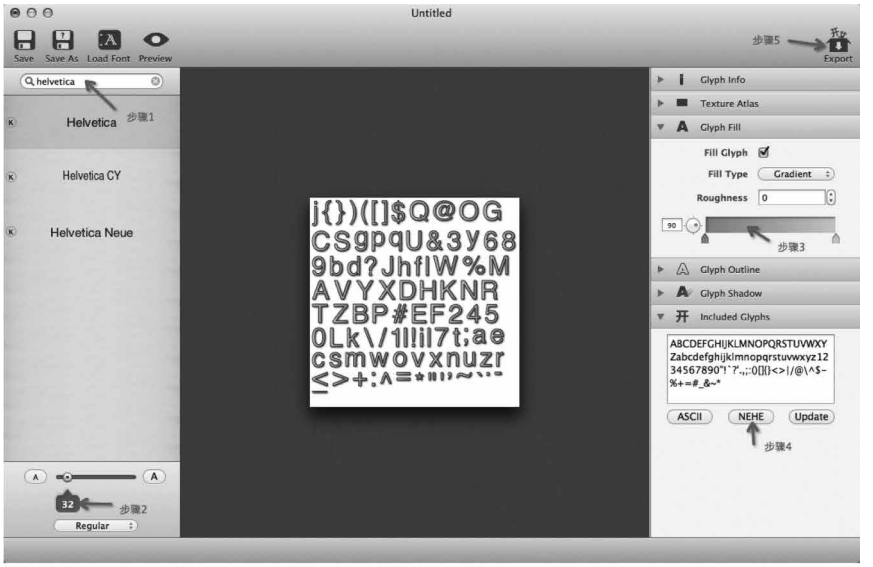
(1) 打开 DMFont 软件，界面如图 3-32 所示，右边的列表是字体库。

(2) 在 Windows 中新建一个 txt 文本，在里面输入需要的文字。

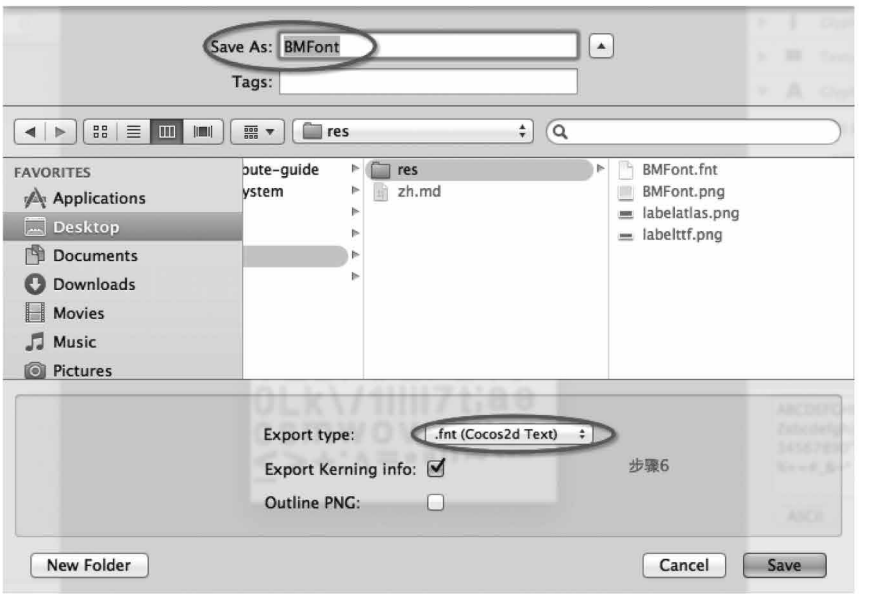
**注**：一定要保存为 UTF-8 格式，否则软件无法识别。

(3) 在 Edit 菜单中选择 Selects chars from file 命令，载入刚才新建的 txt 文件，会发现刚才输入的字符在 BMFont 中已经被选中。

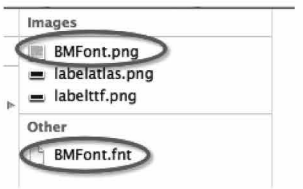
(4) 在 Options 菜单中选择 Font Setting 命令，设置字体，再设置其中的 Font 和 Charset(默认的 Unicode 即可)，如图 3-33 所示。




(a) 生成字体文件




(b) 保存文件




(c) 字体文件

图 3-31　Glyph Designer 制作字体的使用步骤

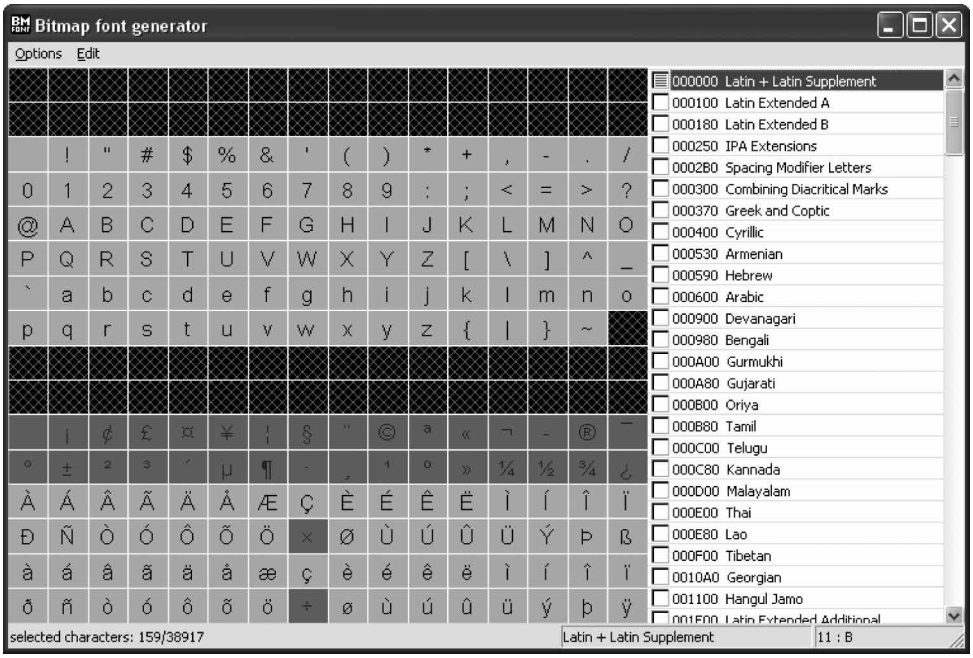



图 3-32　BMFont 主界面

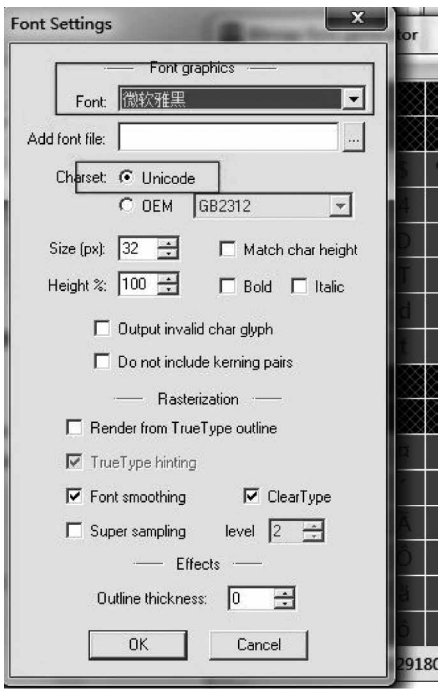



图 3-33　Font Settings

(5) 在 BMFont 上找到 Options 菜单,然后选择 Export options 命令,为了让 Cocos2d-Lua 支持,需要按照图 3-34 所示进行设置。

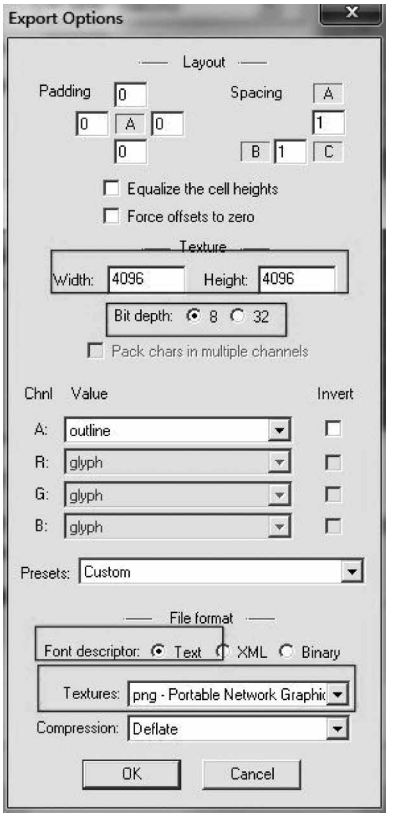



图 3-34 导出选项

① Width 和 Height 的值,4096 即是最大取值,不可超过该值。

② Bit depth,这里选用的是 8 位色深。也可以选择带 alpha 通道的 32 位色深。

(6)在 BMFont 上找到 Options 菜单,然后选择 Save bitmap font as 命令,会发现保存路径下多出了一个 fnt 文件和一个 png 文件。

## 3.7 按钮

不同于文本标签,按钮是图形显示与用户触摸控制的整合抽象。它响应用户触摸,做出显示上的反馈,并触发回调事件。

### 3.7.1 ccui.Button

ccui.Button 与 Cocos Studio 编辑器中的 Button 控件对应。它的工作原理类似于键盘,每单击一次就触发一次事件,松手之后复原。

按钮创建函数为 ccui.Button:create(normalImage,selectedImage,disableImage,textType)。参数说明如下。

(1) normalImage:普通状态下显示的图片(可选)。

(2) selectedImage:按下状态下显示的图片(可选)。

(3) disableImage:禁用状态下显示的图片(可选)。

(4) textType:图片来源,0 从文件,1 从精灵帧缓存(可选)。

如果要创建文本按钮,则接口的 4 个参数均不传值。

创建文本按钮示例代码如下。

```
local btn = ccui.Button:create()
btn:setTitleText('文本按钮')
btn:setTitleFontSize(24)
btn:setTitleColor(cc.c3b(0, 255, 0))
btn:addTo(self)
btn:pos(display.cx, display.cy + 400)
btn:addTouchEventListener(function(ref, eventType)
    if cc.EventCode.BEGAN == eventType then
        print("began")
    elseif cc.EventCode.MOVED == eventType then
        print("move")
    elseif cc.EventCode.ENDED == eventType then
        print("end")
    elseif cc.EventCode.CANCELLED == eventType then
        print("cancel")
    end
end)
```

Button 内部封装有一个 Label,通过成员函数 setTitleText、setTitleFontSize 和 setTitleColor 设置按钮上文本标签的内容、字体大小和颜色。

所有 ccui 控件都派生自基类 ccui.Widget,它们都有共同的触摸事件监听注册函数 addTouchEventListener。通常,监听函数是一个匿名函数,参数 ref 是触发事件对象,eventType 是触摸事件类型。

创建图片按钮示例代码如下。

```
-- 图片按钮
local btn = ccui.Button:create("button/btn_n.png", "button/btn_p.png", "button/btn_d.png", 0)
btn:addTo(self)
btn:pos(display.cx, display.cy + 300)
-- btn:setEnabled(false)
btn:addTouchEventListener(function(ref, eventType)
    if cc.EventCode.BEGAN == eventType then
        print("began")
    elseif cc.EventCode.MOVED == eventType then
        print("move")
```

```
    elseif cc.EventCode.ENDED == eventType then
        print("end")
    elseif cc.EventCode.CANCELLED == eventType then
        print("cancel")
    end
end)
```

文本按钮和图片按钮效果如图 3-35 所示。

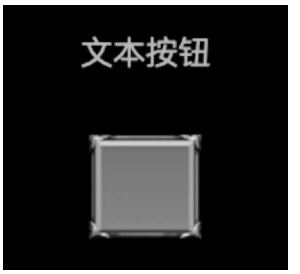



图 3-35 ccui.Button

## 3.7.2 ccui.CheckBox

ccui.CheckBox 与 Cocos Studio 编辑器中的 CheckBox 控件对应。它的工作原理类似于电灯开关,每单击一次,切换一次状态。

按钮创建函数为 ccui.CheckBox:create(backGround,backGroundSelected,cross,backGroundDisabled,frontCrossDisabled,textType)。

参数说明如下。

(1) backGround:图片状态下显示的背景图片(可选)。

(2) backGroundSelected:按下状态下显示的背景图片(可选)。

(3) cross:勾选图片图片状态(可选)。

(4) backGroundDisabled:禁用状态下显示的背景图片(可选)。

(5) frontCrossDisabled:勾选图片禁用状态(可选)。

(6) textType:图片来源,0 从文件,1 从精灵帧缓存(可选)。

创建开关按钮示例代码如下。

```
local btnCB = ccui.CheckBox:create("button/btn_n.png", "button/btn_p.png", "button/
checkbox_on.png", "button/btn_d.png", "button/checkbox_off.png", 0)
    :addTo(self)
    :pos(display.cx, display.cy + 100)

btnCB:setSelected(true)
-- set after setSelected(), make checkbox_off.png show correctly
-- btnCB:setEnabled(false)
btnCB:addEventListener(function(sender, eventType)
    if eventType == ccui.CheckBoxEventType.selected then
        print("change to selected")
    elseif eventType == ccui.CheckBoxEventType.unselected then
        print("change to unselected")
    end
end)
```

开关按钮的内部状态变化由 addEventListener 监听,运行效果如图 3-36 所示。

图 3-36 ccui.CheckBox

# 3.8 场景转换

## 3.8.1 概念

在基础概念介绍中，介绍了场景是引擎渲染的基础，一个游戏可能拥有多个场景，但在任何时候有且只有一个场景是处于激活状态的。

Cocos2d-Lua 中游戏界面以场景为单位，每个场景完成特定的逻辑功能，将不同功能的场景连接起来就组成了整个游戏。一般地，游戏在不同场景之间切换时，为了避免场景切换突兀，程序设计者们往往需要在切换时加入一定的过渡衔接效果。

基础概念中介绍了 display.replaceScene 场景切换 API，但是只使用了一个参数，实际上这个 API 接受 4 个参数，后面的 3 个参数均与场景转换特效相关。

display.replaceScene 的源码如下：

```
function display.replaceScene(newScene, transitionType, time, more)
    if sharedDirector:getRunningScene() then
        if transitionType then
            newScene = display.wrapSceneWithTransition(newScene, transitionType, time,
            more)
        end
        sharedDirector:replaceScene(newScene)
    else
        sharedDirector:runWithScene(newScene)
    end
end
```

假如传入了 transitionType 参数，那么内部将调用 display.wrapSceneWithTransition 生成一个新的场景，然后再调用 sharedDirector：replaceScene 进行场景切换。场景转换特效的实现，实际上是把目标场景生成一个带转场特效的新场景完成的。

## 3.8.2 带转场的场景

display.wrapSceneWithTransition(scene，transitionType，time，more)有 4 个参数，具体作用如下。

1. scene

目标场景。

2. transitionType

指定场景切换使用的动画效果，它接收以下字符串。

(1) crossFade，淡出当前场景的同时淡入下一个场景。

(2) fade，淡出当前场景到指定颜色，默认颜色为 cc.c3b(0，0，0)。可用 more 参数设定翻转颜色。

(3) fadeBL,从左下角开始淡出场景。

(4) fadeDown,从底部开始淡出场景。

(5) fadeTR,从右上角开始淡出场景。

(6) fadeUp,从顶部开始淡出场景。

(7) flipAngular,当前场景倾斜后翻转成下一个场景,默认从左边开始翻转,通过 more 参数可以修改翻转方式。more 可选参数如下:

① cc. TRANSITION_ORIENTATION_LEFT_OVER 从左边开始;

② cc. TRANSITION_ORIENTATION_RIGHT_OVER 从右边开始;

③ cc. TRANSITION_ORIENTATION_UP_OVER 从顶部开始;

④ cc. TRANSITION_ORIENTATION_DOWN_OVER 从底部开始。

**注**:这 4 个值实质上只会产生两种结果,因为 cc. TRANSITION_ORIENTATION_LEFT_OVER 与 cc. TRANSITION_ORIENTATION_UP_OVER 的值都为 0x0, cc. TRANSITION_ORIENTATION _DOWN_OVER 与 cc. TRANSITION_ORIENTATION _RIGHT_OVER 的值都为 0x1。

另外,经验证 Player 模拟器尚有个 Bug。即当指定以上任意特定的附加参数时,用它模拟的运行结果都是一样的,根本体现不了不同参数之间的差异。用其他模拟器(如 Xcode 中的 iOS Simulator)就能正常显示。

(8) flipX,水平翻转,默认从左往右翻转,可用的附加参数同上。

(9) flipY,垂直翻转,默认从上往下翻转,可用的附加参数同上。

(10) zoomFlipAngular,倾斜翻转的同时放大,可用的附加参数同上。

(11) zoomFlipX,水平翻转的同时放大,可用的附加参数同上。

(12) zoomFlipY,垂直翻转的同时放大,可用的附加参数同上。

(13) jumpZoom,跳跃放大切换场景。

(14) moveInB,新场景从底部进入,现有场景同时从顶部退出。

(15) moveInL,新场景从左侧进入,现有场景同时从右侧退出。

(16) moveInR,新场景从右侧进入,现有场景同时从左侧退出。

(17) moveInT,新场景从顶部进入,现有场景同时从底部退出。

(18) pageTurn,翻页效果,如果指定附加参数为 true,则表示从左侧往右翻页。

(19) rotoZoom,旋转放大切换场景。

(20) shrinkGrow,收缩交叉切换场景。

(21) slideInB,新场景从底部进入,直接覆盖现有场景。

(22) slideInL,新场景从左侧进入,直接覆盖现有场景。

(23) slideInR,新场景从右侧进入,直接覆盖现有场景。

(24) slideInT,新场景从顶部进入,直接覆盖现有场景。

(25) splitCols,分成多列切换入新场景。

(26) splitRows,分成多行切换入新场景,类似百叶窗。

(27) turnOffTiles,当前场景分成多个块,逐渐替换为新场景。

3. time

转场动画的持续时间。

4. more

参数 2 可能需要的额外参数。

### 3.8.3 场景转换示例

通常不需要调用 display.wrapSceneWithTransition 生成转场 scene,可以直接用封装好的 display.replaceScene 完成同样的功能。例如:

```
display.replaceScene(nextScene, "fade", 0.5, cc.c3b(255, 0, 0))
```

上面的代码使用红色渐变作转场切换效果,fade 特效可以接收额外的参数,第 4 个参数 more 用来设定渐变色。

测试转场需要综合前面所学的 Lable、Button 等知识。修改初始化工程的 MainScene 的 ctor 函数如下:

```
function MainScene:ctor()
    local btn = ccui.Button:create()
    btn:setTitleText('Click to Second Scene')
    btn:setTitleFontSize(24)
    btn:setTitleColor(cc.c3b(0, 255, 0))
    btn:addTo(self)
    btn:pos(display.cx, display.cy)
    btn:addTouchEventListener(function(ref, eventType)
        if cc.EventCode.ENDED == eventType then
            local secondScene = import("app.scenes.SecondScene"):new()
            display.replaceScene(secondScene, "fade", 0.5, cc.c3b(255, 0, 0))
        end
    end)
end
```

这里创建了一个按钮,单击之后切换到 SecondScene。SecondScene.lua 的代码如下:

```
local SecondScene = class("SecondScene", function()
    return display.newScene("SecondScene")
end)

function SecondScene:ctor()
    local label = display.newTTFLabel({
        text = "This is Second Scene",
        font = "",
        size = 30,
        align = cc.TEXT_ALIGNMENT_CENTER,
        x = display.cx,
```

```
        y = display.cy + 400
    })

    label:center():addTo(self)
end

function SecondScene:onEnter()
end

function SecondScene:onExit()
end

return SecondScene
```

SecondScene 简单创建一个 Lable，通过 Label 显示文字的不同就能判断场景已经切换。也可以修改 display.replaceScene 中的 transitionType 参数测试不同的转场特效。

## 3.9 动作

在之前的章节中已经介绍了游戏的场景、导演、层和精灵等构成游戏画面的基本元素的概念，但游戏不仅是由静态画面构成的，更多的时候游戏是动态效果的呈现，这也是游戏与应用的主要区别。因此，决定一个游戏引擎好坏的重要因素是引擎对动作和动画的支持程度。

Cocos2d-Lua 中，动作是用来描述游戏节点行为规范的一个类，引擎支持很多动作，其中 Action 类是所有动作的基类，它创建的每一个对象都代表一个动作。动作作用于 Node，因此，每个动作都需要由 Node 对象执行，它本身并不是一个能在屏幕中显示的对象。

Cocos2d-Lua 引擎中的动作分为基础动作和高级动作两大类。基础动作包含瞬时动作和有限时间动作两类。高级动作分为复合动作与变速动作，它们都是在基础动作上组合变化得来的。

### 3.9.1 瞬时动作

顾名思义，瞬时动作是指能立刻完成的动作，这中间不产生任何动画效果。更准确地说，这类动作是在下一帧会立刻执行并完成的动作，如设定位置和设定缩放等。

这些动作原本可以通过简单地对 Node 赋值完成，但是把它们封装为动作后，可以方便地与其他动作类组合为复杂动作。

1. Place

该动作用于将节点放置到某个指定位置，作用与 setPosition 相同，但这里它是一个动作，意味着它可以用在复合动作中。用法如下：

```
local place = cc.Place:create(cc.p(10, 10))
```

2. FlipX 与 FlipY

这两个动作只能作用于 Sprite，它们分别将 Sprite 沿 $X$ 轴或 $Y$ 轴反转显示，其作用与 setFlippedX 或 setFlippedY 相同，将其包装成动作是为了便于与其他动作进行组合。用法如下：

```
local flipxAction = cc.FlipX:create(true)
```

参数为 true 则翻转，参数为 false 则不翻转。

3. Show 与 Hide

这两个动作分别用于显示和隐藏节点，作用与 setVisible 相同。用法如下：

```
local hideAction = cc.Hide:create()
```

4. RemoveSelf

该动作执行的时候，把自己从父节点上移除。它通常用在顺序执行的复合动作的最后一个动作，以实现特效播放完毕自动移除节点。

```
local removeAction = cc.RemoveSelf:create()
```

5. CallFunc

函数动作 CallFunc 很有用，它可以用在复合动作中判断某个动作是否执行结束，然后启动其他逻辑。用法如下：

```
local callback = cc.CallFunc:create(function() print("hello world") end)
```

## 3.9.2 有限时间动作

与瞬时动作不同，连续动作需要至少 1 个游戏帧来完成。

1. MoveTo 与 MoveBy

用于使节点从当前坐标点匀速直线运动到目标点。用法如下：

```
local moveTo = cc.MoveTo:create(2, cc.p(0, 0))
```

(1) 参数 1：运动总时间。

(2) 参数 2：moveTo 是目标终点，moveBy 是相对偏移向量。

2. JumpTo 与 JumpBy

使节点以一定的轨迹匀速跳跃到指定位置。用法如下：

```
local jumpTo = cc.JumpTo:create(2, cc.p(10, 0), 50, 2)
```

(1) 参数 1：运动总时间。

(2) 参数 2：JumpTo 是目标终点，JumpBy 是相对偏移向量。

(3) 参数 3：跳跃高度。

(4) 参数 4：跳跃次数。

### 3. BezierTo 与 BezierBy

使节点沿贝塞尔曲线运动。每条贝塞尔曲线都包含一个起点和一个终点。在一条曲线中，起点和终点各自包含一个控制点，而控制点到端点的连线称作控制线。控制点决定了曲线的形状，包含角度和长度两个参数，如图 3-37 所示。

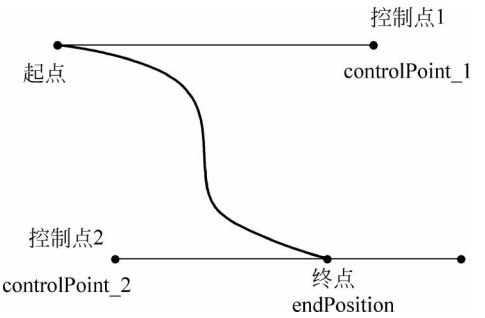



图 3-37　贝塞尔曲线

用法如下：

```
local action = cc.BezierTo:create(2, {cc.p(display.right, display.top), cc.p(200, 200), cc.p(50, 100)})
```

(1) 参数 1：运动总时间。

(2) 参数 2：包含贝塞尔曲线数据的 table，依次是：

① 控制点 1 的坐标；

② 控制点 2 的坐标；

③ BezierTo 是目标终点，BezierBy 是相对偏移向量。

### 4. ScaleTo 与 ScaleBy

使节点的缩放系数随时间线性变化。用法如下：

```
local action = cc.ScaleTo:create(2, 0.5)
```

(1) 参数 1：持续时间。

(2) 参数 2：ScaleTo 是最终缩放系数，ScaleBy 是相对缩放系数。

### 5. RotateTo 与 RotateBy

使节点围绕锚点旋转。用法如下：

```
local action = cc.RotateTo:create(2, 180)
```

(1) 参数 1：持续时间。

(2) 参数 2：RotateTo 是最终角度，RotateBy 是相对旋转角度。

### 6. FadeIn、FadeOut 和 FateTo

FadeIn 淡入，透明度变化范围为 0～255。FadeOut 淡出，透明度变化范围为 255～0。用法如下：

```
local action = cc.FadeIn:create(2)
```

参数：持续时间。

FateTo 从节点当前透明度变化到指定透明度。用法如下：

```
local action = cc.FateTo:create(2,110)
```

(1) 参数 1：持续时间。

(2) 参数 2：目标透明度。

7. TintTo 与 TintBy

使节点着色。用法如下：

```
local action = cc.TintTo:create(1, 0, 255, 0)
```

(1) 参数 1：持续时间。

(2) 参数 2：TintTo 为目标 red 值，TintBy 为相对 red 变化值。

(3) 参数 3：TintTo 为目标 green 值，TintBy 为相对 green 变化值。

(4) 参数 4：TintTo 为目标 blue 值，TintBy 为相对 blue 变化值。

8. Blink

使节点闪烁，内部是设置节点透明度为 0 或 255 来实现闪烁效果。用法如下：

```
local action = cc.Blink:create(1, 2)
```

(1) 参数 1：持续时间。

(2) 参数 2：闪烁次数。

9. Animation

序列帧动画也叫逐帧动画(Frame By Frame)，在时间轴的每帧上逐帧绘制不同的内容，使其连续播放而成动画。序列帧动画后面的章节将重点详细介绍。用法如下：

```
display.addSpriteFrames("grossini-aliases.plist", "grossini-aliases.png")

local frames = display.newFrames("grossini_dance_%02d.png", 1, 14)
local animation = display.newAnimation(frames, 0.2)
local animate = cc.Animate:create(animation)

local sprite1 = display.newSprite("#grossini_dance_01.png")
    :center()
    :addTo(self.backgroundLayer)
    :runAction(animate)
```

## 3.9.3 复合动作

Cocos2d-Lua 提供了一套动作的复合机制，允许组合各种基本动作，产生更为复杂和生动的动作效果。

复合动作是一类特殊的动作，因此它也需要使用 Node 的 runAction 方法执行。而它的特殊之处在于，作为动作容器，复合动作可以把许多动作组合成一个复杂的动作。因此，通常会使用一个或多个动作来创建复合动作，再把动作交给节点执行。复合动作十分灵活，这是由于复合动作本身也是动作，因此也可以作为一个普通的动作嵌套在其他复合动作中。

### 1. DelayTime

DelayTime 是一个“什么都不做”的动作，类似于音乐中的休止符，用来表示动作序列里一段空白期，通过占位的方式将不同的动作段串接在一起。它最常见的用法就是在一个 Sequence 序列动作中，打入若干延时时间，让动作的执行速度慢下来，不至于眼花缭乱，让人反应不过来。

DelayTime 本身不是复合动作，但是它只有放在复合动作中才有存在的意义，所以把它归类到复合动作中来解析。

```
cc.DelayTime:create(delay)
```

参数 delay 表示需要延时的时间。

### 2. Repeat 与 RepeatForever

有的情况下，动作只需要执行一次，但还常常遇到一个动作反复执行的情况。对于一些重复的动作，可以通过 Repeat 与 RepeatForever 这两个方式重复执行：

```
cc.Repeat:create(action, times)
cc.RepeatForever:create(action)
```

RepeatForever 是无限重复执行动作，Repeat 重复执行 times 次动作。

Repeat 动作不能嵌入其他复合动作内使用，它应该是最外层的动作。嵌入其他复合动作会导致不能正确重复动作。

### 3. Spawn

使一个 Node 同时执行一批动作，并列动作必须是能够同时执行并继承自 FiniteTimeAction 的动作，合并之后，动作执行完成时间按照最大的一个动作执行时间计算。Spawn 动作的创建方法如下：

```
cc.Spawn:create(action1, ...)
```

### 4. Sequence

除了让动作同时并列执行，更常遇到的情况是顺序执行一系列动作。Sequence 提供了一个动作队列，它会顺序执行一系列动作。Sequence 同样派生自 ActionInterval。与 Spawn 一样，Sequence 创建方法如下：

```
cc.Sequence:create(action1, ...)
```

### 5. Follow

Follow 是一个节点跟随另一个节点的动作。Follow 的创建方法如下：

```
cc.Follow:create(followedNode, rect)
```

作用是创建一个跟随的动作。

(1) 参数 1：跟随的目标。

(2) 参数 2：跟随范围，离开范围就不再跟随。

Follow 经常用来设置 Layer 跟随 Sprite，可以实现类似摄像机跟拍的效果。下面是 Cocos2d-Lua 中使用 Follow 动画的一个例子：

```
function MainScene:ctor()
    self.backgroundLayer = display.newColorLayer(cc.c4f(128,128,128,255))
    self.backgroundLayer:addTo(self)

    local sprite1 = display.newSprite("1.png")
    sprite1:center()
    local move_right = cc.MoveBy:create(1.5, cc.p(display.width / 2, 0))
    local move_left = cc.MoveBy:create(3, cc.p( - display.width, 0))
    local seq = cc.Sequence:create(move_right, move_left, move_right)
    local rep = cc.RepeatForever:create(seq)
    sprite1:runAction(rep)
    sprite1:addTo(self.backgroundLayer)

    self.backgroundLayer:runAction(cc.Follow:create(sprite1))
end
```

该段代码实现了精灵在地图上移动，地图也跟着移动，但是精灵仍然是在整个界面的中心位置。

### 3.9.4 变速动作

前面介绍的动作都是匀速运动的，但是也需要速度变化的动作，如模拟汽车起步加速的过程等。

#### 1. Speed

可调整速度动作 Speed 不是一个独立的动作，可以把它理解为是对目前动作的“包装”，经过这个“包装”以后，就可以实现慢动作和快进的效果，使用 Speed 来处理很方便。Speed 的创建方法如下：

```
cc.Speed:create(action, speed)
```

作用是让目标动作运行速度加倍。

(1) 参数 1：目标动作。

(2) 参数 2：倍速。

#### 2. ActionEase

Speed 虽然能改变动作的速度，但是只能按比例改变目标动作的速度，如果要实现动作由快到慢、速度随时间改变的变速运动，需要不停地修改它的 speed 属性才能实现，显然这是一个很烦琐的方法。下面介绍的 ActionEase 系列动作通过使用内置的多种自动速度变化来解决这一问题。该类型包含 5 类运动：指数缓冲、Sine 缓冲、弹性缓冲、跳跃缓冲和回震缓冲。每类运动都包含 In、Out 和 InOut 3 个不同的变换，其含义如下。

(1) In：表示动作执行先慢后快。

（2）Out：表示动作执行先快后慢。

（3）InOut：表示动作执行慢-快-慢。

ActionEase 改变的是内部动作的执行速率(注意，并没有改变执行的最终效果和执行的时间)。可以用图 3-38 表示，横轴表示时间，纵轴表示位移。

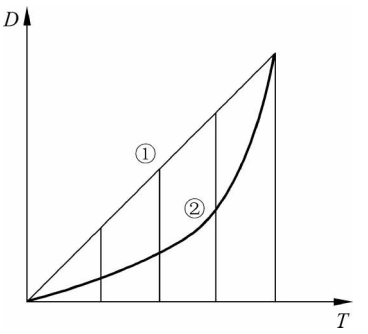


图 3-38　ActionEase

这里假设有一个动作，4s 内按(200，200)的增量进行 MoveBy 移动，那么曲线①所表示的就是其速率的变化情况，可以看出，它是按照匀速速率进行的动作。

现在使用 EaseSineIn 做包装器，包装这个动作，如下：

```
cc. EaseSineIn: create ( cc. MoveBy: create ( 4, cc. p ( 200, 200)))
```

它的位移时间关系就变成图 3-38 中所示曲线②，可以看出，移动是非匀速速率进行的，先慢后快。并且，还可以看出，使用 ActionEase 系列动作，并没有改变位移和时间，但是改变了动作的执行速率，从匀速执行变为非匀速执行。

全部 ActionEase 系列动作的作用效果如图 3-39 所示。

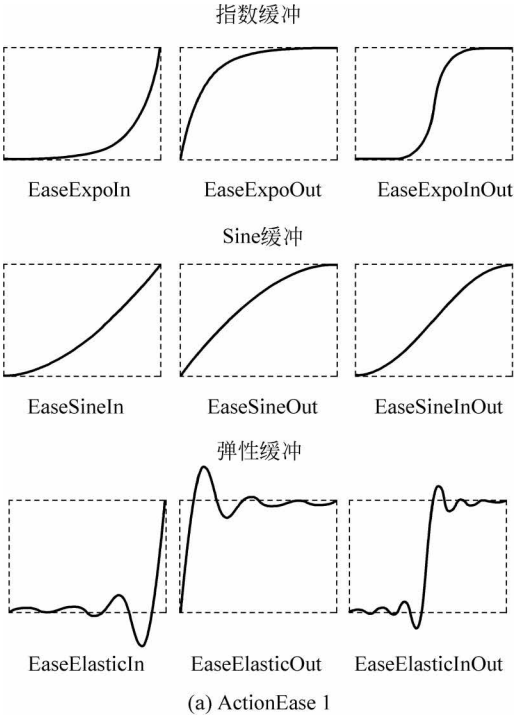
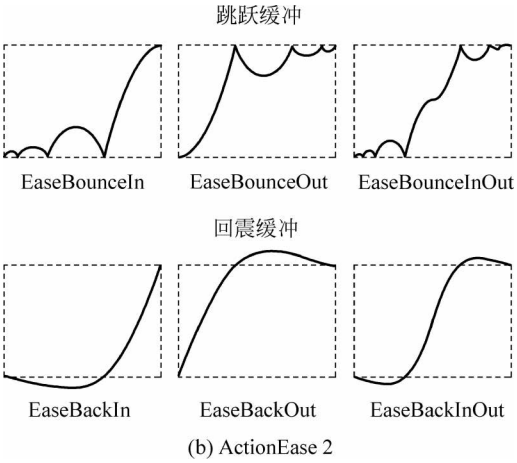


图 3-39　全部 ActionEase 系列动作的作用效果




(b) ActionEase 2

图 3-39 （续）

### 3.9.5 节点与动作相关的接口

前面介绍了各种 Action 的创建，最终这些 Action 是由节点来执行的，具体接口如下：

```
node:runAction(action)
```

如果节点多次调用 runAction 运行不同的动作，那么这些动作会同时运行，效果叠加。既然有运行接口，那么就有停止接口，如下：

```
node:stopAllActions()          -- 停止所有动作
node:stopAction(action)        -- 停止某个指定的 action 对象
node:stopActionByTag(tag)      -- 停止某个 tag 的 action 对象
```

事实上 Action 类也是从 Node 继承而来的，所以 Action 也能设置 tag 标签：

```
node:setTag(tag)               -- Node 设置 tag 为整数
action:setTag(tag)             -- Action 设置 tag 为整数
```

所以也能用 tag 来获取或停止一个动作：

```
node:getActionByTag(tag)
node:stopActionByTag(tag)
```

## 3.10 序列帧动画

序列帧动画指的是逐帧动画(以下简称帧动画)，也就是动画的每一帧都有独立的数据保存，连续播放这些帧形成了动画。常见的 GIF 动画就是逐帧动画。

逐帧动画的播放原理与图 3-40 所示的翻动连续画册一样。

图 3-40　手翻书

很显然，逐帧动画由于每一帧都是数据，都需要单独存储，所以内存空间消耗大。但它的实现简单，数据加载进内存，然后定时播放每一帧就能实现动画，所以在计算机发展早期被广泛应用。

Cocos2d-Lua 对序列帧动画进行了封装，特别对从精灵表单(SpriteSheet)创建帧动画的流程进行了优化，使用起来很方便。在学习如何在 Cocos2d-Lua 中播放帧动画之前，先来了解什么是精灵表单，以及精灵表单如何制作。

## 3.10.1　精灵表单

精灵表单由一张存储多个精灵纹理的大图和一个对应的描述文件组成，描述文件记录每个精灵纹理在大图上的位置区域。

用一张大图来集合精灵的纹理有如下三方面的好处：

(1) 减少磁盘存储空间占用。一张大图可以减少图片压缩与解压次数；大图上相邻的精灵纹理的透明像素可以被进一步压缩掉以减少文件体积。

(2) 减少内存空间占用。由于 OpenGL ES 在加载纹理的时候需要对宽高不足 $2^n$ 的图片进行填充，所以在一张大图上存储多个纹理将有效减少内存空间的占用。

(3)减少 CPU 开销。加载一个精灵表单能一次性加载多张纹理资源，之后对纹理的访问都在内存中进行，避免离散文件的多次加载。同时在图片解压的时候也只需要一次就能完成。在引擎内部，更可以使用批处理指令，来减少绘图指令交互。

关于精灵表单，TexturePacker 官方有两个生动的视频介绍，打开下面的链接地址可以查看视频：https://v.youku.com/v_show/id_XNDE0OTU5MDEzNg、https://v.youku.com/v_show/id_XNDE0OTU5Njg5Mg。

精灵表单的制作需要由 TexturePacker 工具来完成，官网地址为 https://www.codeandweb.com/texturepacker。

TexturePacker 可以免费下载使用，但是完整功能需要付费，如果不付费，转换出来的精灵表单中的某些纹理可能会被替换成其他图片。作为一个游戏开发的必备工具，TexturePacker 价格并不是很高。该软件按年收取费用，到期后不能更新，但在付费期间的版本，即使 license 过期后也能使用。

TexturePacker 的安装很简单，下面主要介绍精灵表单的制作。

(1) 打开软件，单击 New 按钮。

(2) 拖曳图片文件到最右侧的 Sprites 栏中。

(3) 选择 Data Format，默认是 cocos2d，也就是 Cocos2d 系列引擎所支持的格式。单击下拉菜单按钮，可以看到还有 cocos2d-0.99.4 和 cocos2d-original 两个格式，它们是

Cocos2d 引擎早期版本支持的格式。目前的是 Cocos2d-Lua 引擎，选用 cocos2d 格式。

(4) 设置 Data file 路径，即 plist 文件存放路径。这项设置会同时作用于 Texture file 文件的存放路径。

(5) 选择 Texture format 格式，推荐使用 zlib compr. PVR 压缩方案。

(6) 选择 Image format，默认为 RGBA8888，最高图像质量，如果需要进一步压缩文件体积，可以选择 RGB565 等格式。

(7) 保存 TexturePacker 项目文件。方便以后增减图片的时候，重新生成精灵表单。

(8) 单击 Publish 按钮，开始生成精灵表单，生成的. pvr. ccz 和对应的. plist 文件存放在步骤(4)设置的文件夹下。

每个步骤对应的位置编号如图 3-41 所示。

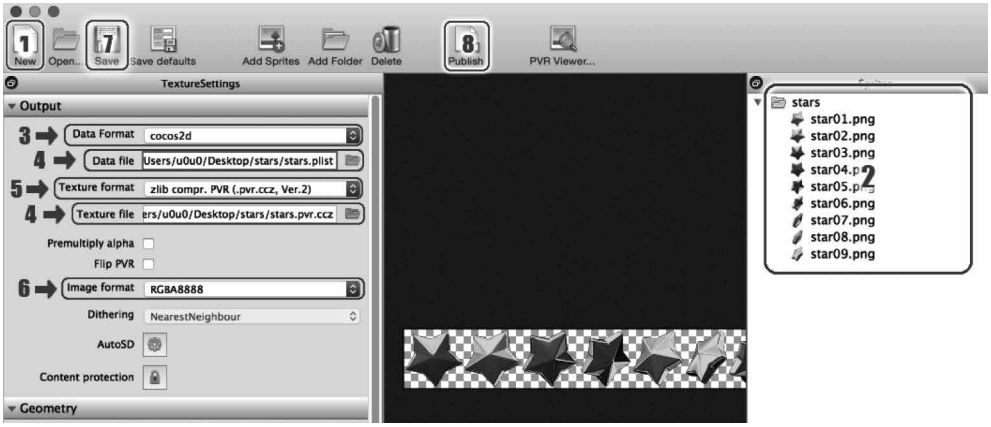



图 3-41 TexturePacker

假定步骤(4)设置的存储文件名为 stars，那么生成的精灵表单对应为 stars. pvr. ccz 和 stars. plist 两个文件。

## 3.10.2 播放序列帧动画

在工具端完成了精灵表单的制作之后，可以切换回游戏代码，体验把刚刚生成的精灵表单加载到游戏中并播放动画。如果一切顺利，将看到一颗转动的星星。

播放精灵表单序列帧动画的步骤如下：

(1) 将导出的 stars. pvr. ccz 和 stars. plist 文件复制到项目的 res 目录下。

(2) 从精灵表单批量导入精灵帧到引擎的帧缓存(有关帧缓存的知识后续章节将会介绍)。

Cocos2d-Lua 中使用 display. addSpriteFrames(plistFilename, image, handler)方法从指定的 SpriteSheet 导入精灵帧，三个参数作用如下。

① plistFilename：string 类型，它是数据文件 . plist 的相对路径。

② image：string 类型，它是纹理文件 . pvr. ccz 的相对路径。

③ handler：function 类型，是可选参数，设置了 handler 则引擎启用异步模式加载精灵表单，并在加载结束后回调 handler。

(3) 生成序列帧数组。精灵表单中的精灵帧可以用来做动画，也可以用来显示设置给某个静态精灵，需要挑选需要的帧并按照期望的播放顺序组成一个序列帧数组。

Cocos2d-Lua 中使用 display.newFrames(pattern,begin,length,isReversed)方法生成序列帧数组。它使用字符串匹配模式，从帧缓存中依次寻找对应名称的帧，然后添加到数组中。参数作用如下。

① pattern：string 类型，字符串匹配模板。

② begin：integer 类型，起始索引。

③ length：integer 类型，表示帧序列数组长度，决定了末尾索引。

④ isReversed：boolean 类型，默认为递增排序，如果设置为 true，则递减排序。

(4) 生成 Animation。Animation 是一个描述帧动画的对象。

Cocos2d-Lua 中使用 display.newAnimation(frames,time)方法创建 Animation 对象，参数如下。

① frames：table 类型，上一步骤生成的序列帧数组。

② time：number 类型，相邻两帧之间的时间间隔。

(5) 播放精灵动画。Animation 对象不能直接被节点的 runAction 播放，因为它不是一个 Action 对象，需要使用 cc.Animate：create(animation)来创建一个 Animate 才能被 runAction 播放。Animate 提供很多灵活性，可以用在复合动作中。

完整的帧动画创建示例代码如下：

```
display.addSpriteFrames("stars.plist", "stars.pvr.ccz")
local frames = display.newFrames("star%02d.png", 1, 9)
local animation = display.newAnimation(frames, 0.2)
local animate = cc.Animate:create(animation)

-- # 表示从精灵帧缓存中获取纹理资源，帧既可以用来创建动画，也可以用来创建精灵
local sprite1 = display.newSprite("#star01.png")
    :center()
    :addTo(self.backgroundLayer)
    :runAction(animate)
```

## 3.10.3 动画缓存

通常情况下，对于一个精灵帧动画，每次创建时都需要加载精灵帧，按顺序添加到数组，再创建帧动画类，这是一个非常烦琐的计算过程。对于使用频率高的动画，将其加入缓存可以有效降低每次创建的消耗。

**1. 缓存动画**

使用下面的方法把指定动画加入到动画缓存。

```
display.setAnimationCache(name,animation)
```

其中，name 为 string 类型，用于指定动画名称；animation 为 Animation 动画对象。

**注**：缓存的对象是 Animation，不是 Animate。

**2. 获取缓存的动画**

使用下面的方法获取指定动画名称的动画对象。

```
animation = display.getAnimationCache(name)
```

其中，name 为动画名，如果未找到对应的动画，则返回 nil。

**3. 删除缓存的动画对象**

如果游戏收到系统内存警告，则可以使用如下方法清理动画缓存。

```
display.removeAnimationCache(name)
```

其中，name 为 string 类型，指定删除的动画对象的名称。

**4. 动画缓存使用示例**

结合前面生成的精灵表单，创建好帧动画，添加到动画缓存并起名为 stars，最后使用函数 sprite：runAction()来播放动画。代码如下：

```
function MainScene:ctor()
    self.backgroundLayer = display.newColorLayer(cc.c4f(128,128,128,255))
    self.backgroundLayer:addTo(self)

    -- preload frames to cache
    display.addSpriteFrames("stars.plist", "stars.pvr.ccz")

    local sprite = display.newSprite("#star01.png")
        :center()
        :addTo(self.backgroundLayer)

    local frames = display.newFrames("star%02d.png", 1, 9)
    local animation = display.newAnimation(frames, 0.1)

    -- 添加到 cache
    display.setAnimationCache("stars", animation)
    -- 从 cache 中取出
    animation = display.getAnimationCache("stars")
    -- 清除动画缓存
    display.removeAnimationCache("stars")
    sprite:runAction(cc.Sequence:create(
        cc.Animate:create(animation),
        cc.CallFunc:create(function()
            print("animate play done")
        end)
    ))
end
```

# 3.11 调度器

Cocos2d-Lua 引擎中的调度器是用来周期执行某个函数或延时执行某个函数的。功能类似于定时触发器，但它又与游戏紧密结合。

Cocos2d-Lua 中的调度器分为两种，即全局调度器和节点调度器。

## 3.11.1 全局调度器

在游戏中，经常需要周期性地处理事务，并且这些事务不会因为某个节点的销毁而取消。例如在线游戏的网络心跳包，或某些全局变量的刷新。全局调度器用来解决这类问题。

全局调度器是 Cocos2d-Lua 在 Cocos2d-x 的基础上提出来的，它基于 schedule 进行封装，让 Lua 可以脱离节点使用调度器。

Cocos2d-Lua 框架默认不加载全局调度器模块，需要手动加载：

```
local scheduler = require(cc.PACKAGE_NAME .. ".scheduler")
```

全局调度器模块提供了如下三种调度器以满足各种需求：

(1) 全局帧调度器：scheduleUpdateGlobal(listener)。

(2) 全局自定义调度器：scheduleGlobal(listener,interval)。

(3) 全局延时调度器：performWithDelayGlobal(listener,time)。

前两个调度器的生命周期需要手动管理，全局延时调度器会在回调后自动销毁，但也不是所有情况下都完全可靠，引擎提供了一个注销调度器的接口：scheduler.unscheduleGlobal(handle)。

**1. 全局帧调度器**

顾名思义，全局帧调度器是游戏的每一帧都会触发的调度器，主要用在碰撞检测等每一帧都需要计算的地方。全局帧调度器不依赖任何场景，因此可以在整个游戏范围内实现较为精确的全局计时。

全局帧调度器的示例代码如下：

```
local scheduler = require(cc.PACKAGE_NAME .. ".scheduler")
local function onInterval(dt)
    print("update")
end
scheduler.scheduleUpdateGlobal(onInterval)
```

回调函数 onInterval 的参数 dt 是两次调度之间的时间间隔。

运行示例代码会在控制台不停输出以下信息：

```
cocos2d: update
cocos2d: update
...
```

**2. 全局自定义调度器**

全局帧调度器是全局自定义调度器的特例，自定义调度器可以指定调度时间，提供更高的灵活性。

由于引擎的调度机制，自定义时间间隔必须大于两帧的间隔，否则两帧内的多次调用会被合并成一次调用，所以自定义时间间隔应在 1/60s 以上(引擎默认每秒刷新 60 帧)。

全局自定义调度器的示例代码如下：

```
local scheduler = require(cc.PACKAGE_NAME .. ".scheduler")
local function onInterval(dt)
    print("Custom")
end
scheduler.scheduleGlobal(onInterval, 0.5)
```

每隔 0.5s 控制台输出以下信息：

```
cocos2d: Custom
cocos2d: Custom
...
```

**3. 全局延时调度器**

若在游戏中某些场合，只想实现一个单次的延迟调用，这就需要延迟调度器。scheduler.performWithDelayGlobal()会在等待指定时间后执行一次回调函数，然后自动取消该 scheduler。

全局延时调度器的示例代码如下：

```
local scheduler = require(cc.PACKAGE_NAME .. ".scheduler")
local function onInterval(dt)
    print("once")
end
scheduler.performWithDelayGlobal(onInterval, 0.5)
```

在控制台只会看到一次输出：

```
cocos2d: Once
```

## 3.11.2 节点调度器

Node 是 Cocos2d-Lua 引擎中的基础类，它封装了很多基础方法与属性，其中调度器就是 Node 提供的方法之一。Node 中的调度器只能在 Node 中使用，Node 负责管理调度器的生命周期，当 Node 销毁的时候，会自动注销节点名下的所有调度器。

大部分情况下，我们使用节点调度器，这样能把精灵集中在游戏逻辑实现，而不是调度器的生命周期管理。

节点调度器同样提供了三种调度器：

(1) 节点帧调度器。节点帧调度器在 Cocos2d-Lua 中已归类到节点帧事件，请参考

3.12 节。

(2) 节点自定义调度器。

(3) 节点延时调度器。

下面详细介绍后两种调度器。

**1. 节点自定义调度器**

由于引擎的调度机制,自定义时间间隔必须大于两帧的间隔,否则两帧内的多次调用会被合并成一次调用,所以自定义时间间隔应在 1/60s 以上(引擎默认每秒刷新 60 帧)。

节点自定义调度器的示例代码如下:

```
local action = node:schedule(function ()
    print("schedule")
end, 1.0)
```

事实上,schedule 函数内部是用动作系统来实现的。如果需要提前停止节点调度器,可以用停止动作的方式实现。代码如下:

```
node:stopAction(action)
```

**2. 节点延时调度器**

节点延时调度器等待指定时间后执行一次回调函数。示例代码如下:

```
node:performWithDelay(function ()
    print("performWithDelay")
end, 1.0)
```

提前停止节点延迟调度器的方法依然是用 stopAction()。

## 3.12 事件分发机制

Cocos2d-Lua 中 Quick 框架的事件分发机制与 Cocos2d-x 的不同,它在结合 Lua 语言特性方面做了改进。

Quick 框架的事件按照功能和用途分为:

(1) 节点事件。

(2) 帧事件。

(3) 键盘事件。

(4) 加速计事件。

(5) 触摸事件。

接下来,详细讲解一下 Quick 框架中各种事件的处理方法。

### 3.12.1 节点事件

节点事件在一个 Node 对象进入和退出场景时触发。例如,加入一个层或者其他的

Node 的子类的时候，想在子类进入或者退出时添加一些数据清除的工作，可以通过这个事件来操作。

就事件含义本事来讲，叫场景事件更贴切。但是它能被场景及其所有子节点监听。

```
node:addNodeEventListener(cc.NODE_EVENT, function(event)
    print(event.name)
end)
-- 启用节点事件
node:setNodeEventEnabled(true)
```

**注**：display.newScene()创建的场景，默认开始了节点事件。如果是其他节点，需要主动调用 setNodeEventEnabled 开启节点事件监听。

参数 event 只有 name 属性，值如下。

(1) enter：加载场景。

(2) exit：退出场景。

(3) enterTransitionFinish：转场特效结束。

(4) exitTransitionStart：转场特效开始。

(5) cleanup：场景被完全清理并从内存删除。

例如，把下面的代码加到 MainScene 的 ctor()函数中：

```
local function createTestScene(name)
    local scene = display.newScene(name)
    scene:addNodeEventListener(cc.NODE_EVENT, function(event)
        printf("node in scene [%s] NODE_EVENT: %s",name, event.name)
    end)
    return scene
end

-- 等待 1.0s 创建第一个测试场景
self:performWithDelay(function()
    local scene1 = createTestScene("scene1")
    display.replaceScene(scene1)

    -- 等待 1.0s 创建第二个测试场景
    scene1:performWithDelay(function()
        print("--------")
        local scene2 = createTestScene("scene2")
        display.replaceScene(scene2)
    end, 1.0)
end, 1.0)
```

运行后可以看到如下的输出信息：

```
cocos2d: node in scene [scene1] NODE_EVENT: enter
cocos2d: node in scene [scene1] NODE_EVENT: enterTransitionFinish
```

```
cocos2d: --------
cocos2d: node in scene [scene1] NODE_EVENT: exitTransitionStart
cocos2d: node in scene [scene1] NODE_EVENT: exit
cocos2d: node in scene [scene1] NODE_EVENT: cleanup
cocos2d: node in scene [scene2] NODE_EVENT: enter
cocos2d: node in scene [scene2] NODE_EVENT: enterTransitionFinish
```

在切换场景时如果没有使用特效,那么事件出现的顺序如上。

但如果将测试代码 display. replaceScene(scene2)修改为 display. replaceScene(scene2, "random",1.0),事件出现顺序会变成:

```
cocos2d: node in scene [scene1] NODE_EVENT: enter
cocos2d: node in scene [scene1] NODE_EVENT: enterTransitionFinish
cocos2d: --------
cocos2d: node in scene [scene1] NODE_EVENT: exitTransitionStart
cocos2d: node in scene [scene2] NODE_EVENT: enter
cocos2d: node in scene [scene1] NODE_EVENT: exit
cocos2d: node in scene [scene2] NODE_EVENT: enterTransitionFinish
cocos2d: node in scene [scene1] NODE_EVENT: cleanup
```

造成这种区别的原因就是场景切换特效播放期间,会同时渲染两个场景,所以从事件上看,可以看到第二个场景的 enter 事件出现后,第一个场景的 exit 事件才出现。

因此,在使用节点事件时,不应该假定事件出现的顺序,而是根据特定事件采取特定的处理措施。

通常建议如下。

(1) enter:这里可以做一些场景初始化工作。

(2) exit:如果场景切换使用了特效,可以在这里停止场景中的一些动画,避免切换场景的特效导致帧率下降。

(3) cleanup:适合做清理工作。

### 3.12.2 帧事件

在 Cocos2d-x 中,C++中可以通过重载 update 函数在每帧刷新的时候执行自己需要的一些操作。在 Quick 框架中,这种事件被称为帧事件,意思是每帧刷新时都会执行的事件。

例如,把下面的代码加到 MainScene 的 ctor()函数中:

```
local node = display.newNode()
self:addChild(node)
-- 注册事件
node:addNodeEventListener(cc.NODE_ENTER_FRAME_EVENT, function(dt)
    print(dt)
end)
-- 启用帧事件
node:scheduleUpdate()
```

```
-- 0.5s 后,停止帧事件
node:performWithDelay(function()
    -- 禁用帧事件
    node:unscheduleUpdate()
    print("STOP")

    -- 再等 0.5s,重新启用帧事件
    node:performWithDelay(function()
        -- 再次启用帧事件
        node:scheduleUpdate()
    end, 0.5)
end, 0.5)
```

运行时,屏幕上会不断输出上一帧和下一帧之间的时间间隔(通常为 1/60s),并在第一个 0.5s 时短暂停顿一下。

只有在调用 scheduleUpdate()后,帧事件才会触发。帧事件的回调函数的参数只有一个 dt,它是用来表示时间间隔的。帧事件在游戏中经常用来更新游戏中的数据。例如制作一款射击游戏,就需要通过帧事件来更新游戏中的子弹坐标等参数。

### 3.12.3 键盘事件

监听键盘事件的方式如下:

```
self:setKeypadEnabled(true)
self:addNodeEventListener(cc.KEYPAD_EVENT, function(event)
    print("TestKeypadEvent = " .. event.key)
end)
```

event 为 table,有以下属性值:

(1) code,数值。按键对应的编码。

(2) key,字符串。按键对应的字符串。

Android 设备可以响应 Menu 和 Back 按键事件。对应的 key 值如下:

① menu,菜单键。

② back,返回键。

(3) type,字符串。按键事件类型。

① Pressed,键盘按键按下事件。

② Released,键盘按键弹起事件。

**注**:iOS 设备没有键盘事件。

### 3.12.4 加速计事件

现在的手机都配备了加速计,用于测量设备静止或匀速运动时所受到的重力方向。

重力感应来自移动设备的加速计,通常支持 $X$、$Y$ 和 $Z$ 3 个方向的加速度感应,所以又

称为三向加速计。在实际应用中,可以根据3个方向的力度大小来计算手机倾斜的角度或方向。在Quick中按如下方法监听加速计事件:

```
-- 重力感应器
self:addNodeEventListener(cc.ACCELEROMETER_EVENT, function (event)
    print("AccelerateData:", event.x, event.y, event.z, event.timestamp)
end)
self:setAccelerometerEnabled(true)
```

event属性如下。

(1) event.x,event.y,event.z:设备在$x$、$y$、$z$轴上的角度。

(2) event.timestamp:测量值更新时间。

### 3.12.5 触摸事件

Cocos2d-x3.x的触摸事件分发机制在Cocos2d-x 2.x上进行了大幅改进,取消了只能layer监听触摸事件的限制,不过在使用过程中依然有些烦琐。Quick框架在此基础上进行了二次封装,简化了用法并保持了接口稳定。本节详细介绍Quick触摸事件的用法。

**1. 显示层级**

在Cocos2d-x里,整个游戏的画面是由一系列的Node、Scene、Layer和Sprite等对象构成的。而所有这些对象都是从Node这个基类继承而来。可以将Node称为显示节点,一个游戏画面就是许多显示节点构成的一棵树,如图3-42所示。

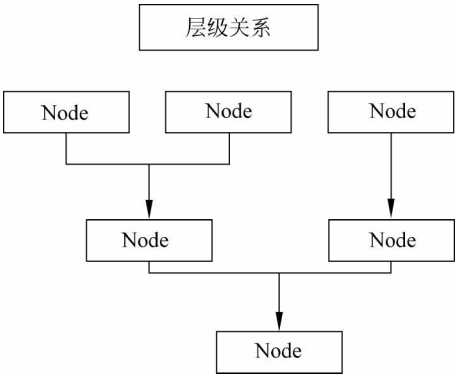


图3-42 Node Tree

在图3-42所示树里,Node所处的垂直位置就是它们的显示层级。越往上的Node,其显示层级就越高。从画面表现上来说,下面的Node是背景,上面的Node是建筑,那么建筑就会挡住一部分背景。

在游戏中的体现就是有的元素显示在上面,有的元素显示在下面,在上面的元素挡住了下面的元素,那么在上面的元素的显示层级就要比在下面的元素的高。

**2. 触摸区域**

在Cocos2d-x的2.x版本中,只有Layer对象才能接受触摸事件。而Layer总是响应整个屏幕范围内的触摸,这就要求开发者在拿到触摸事件后,再做进一步的处理。

例如,有一个需求是在玩家触摸屏幕上的方块时,人物角色做一个动作。那么使用Layer接收到触摸事件后,开发者需要自行判断触摸位置是否在方块之内。当屏幕上有很多东西需要响应玩家交互时,程序结构就开始变得复杂了。所以Cocos2d-x 3.x允许开发者将任何一个Node接受触摸事件,而不局限于Layer。并且触摸事件的开始状态只会出现在这个Node的触摸区域内。

所谓触摸区域，就是一个 Node 及其所有子 Node 显示内容占据的屏幕空间。要注意的是，这个屏幕空间包含了图片的透明部分。如图 3-43 所示的五角星 Sprite 对象，它的触摸区域是包含了透明区域的矩形范围。

触摸事件方式分为两种，一种是单点触摸事件，另一种是多点触摸事件。

图 3-43　Touch Area

**3. 单点触摸事件**

单点触摸事件一个时刻只响应一个触摸点。

通过 setTouchMode 设置 Node 的触摸监听模式，然后通过 addNodeEventListener 设置触摸事件监听回调函数。

```
-- 可以不写这一句，默认为单点触摸
node:setTouchMode(cc.TOUCH_MODE_ONE_BY_ONE)
node:addNodeEventListener(cc.NODE_TOUCH_EVENT, function(event)
    printf("sprite: %s x,y: %0.2f, %0.2f",
           event.name, event.x, event.y)

    if event.name == "began" then
        return true
    end
end)
```

默认节点是不响应触摸的，需要调用下面的接口开启节点的触摸事件监听。

```
node:setTouchEnabled(true)
```

**注**：setTouchEnabled 必须在 addNodeEventListener 之后调用。

当触摸 node 的时候，回调返回 event 信息。

event 是一个 table，具体信息如下。

(1) event.name：事件类型。

① began：手指开始触摸屏幕。在 began 状态时，如果要继续接收该触摸事件的状态变化，事件处理函数必须返回 true。

② moved：手指在屏幕上移动。

③ ended：手指离开屏幕。

④ cancelled：因为其他原因取消触摸操作。通常情况下，cancelled 和 ended 是相同的处理逻辑。

(2) event.x：触摸点 $x$ 坐标。

(3) event.y：触摸点 $y$ 坐标。

**4. 多点触摸**

先设置触摸模式为多点触摸：

```
node: setTouchMode(cc.TOUCH_MODE_ALL_AT_ONCE)
```

然后添加触摸事件回调函数：

```
-- 注册触摸事件
node:addNodeEventListener(cc.NODE_TOUCH_EVENT, function(event)
    -- event.name 是触摸事件的状态：began、moved、ended、cancelled
    -- event.points 包含所有触摸点
    -- 按照 events.point[id] = {x = ?, y = ?} 的结构组织
    for id, point in pairs(event.points) do
        printf("event [ %s] %s = %0.2f, %0.2f",
               event.name, id, point.x, point.y)
    end

    if event.name == "began" then
        return true
    end
end)
```

同样需要在 addNodeEventListener 之后打开 Touch 功能。

```
node:setTouchEnabled(true)
```

在多点触摸时，事件状态的含义有所区别，说明如下。

(1) began：手指开始触摸屏幕。不同于单点触摸，此状态可被触发多次，每一次代表一个或多个手指开始触控屏幕。

(2) moved：由于多点触摸时可能只有部分触摸点移动，所以此时 event.points 中只包含有变化的触摸点数据。

(3) ended：当一个或多个触摸点消失(手指离开了屏幕)时，出现 ended 状态。此时 event.points 中包含删除的触摸点数据。

(4) cancelled：因为其他原因导致触摸点被取消(手指不一定离开了屏幕)。此时 event.points 中包含取消的触摸点数据。通常情况下，cancelled 和 ended 是相同的处理逻辑。

**注**：多点触摸中，回调参数 event.points 的 key 是唯一 ID，在整个触摸状态变化过程中，可以通过这个 ID 来辨识是哪个触摸点发生了状态改变。

**5.触摸事件吞噬**

默认情况下，Node 在响应触摸后(在 began 状态返回 true 表示要响应触摸)，就会阻止事件继续传递给 Node 的父对象(更下层的 Node)，这称为触摸事件吞噬。

Node：setTouchSwallowEnabled()可以改变这个行为。默认为 true 吞噬事件。如果设置为 false，则 Node 响应触摸事件后仍然会将事件继续传递给父节点。

## 3.13 多分辨率适配

随着智能设备的发展，各种屏幕尺寸和分辨率的移动设备层出不穷。为了使游戏更好地适应各种分辨率，减少游戏开发成本，Cocos2d-x 设计了一套完善的多分辨率适配方案。

由于 Cocos2d-Lua 是基于 Cocos2d-x 之上的轻量框架，它使用的依然是 Cocos2d-x 的多分辨率适配方案。本章首先介绍 Cocos2d-x 的多分辨率适配原理，涉及的 API 以 C++的接口为示例，然后进行总结，最后对比 Cocos2d-Lua 中的接口做示例展示。

## 3.13.1 Cocos2d-x 多分辨率适配

### 1. 发展历史

我们知道 Cocos2d-x 是从 Cocos2d-iPhone 派生出来的分支，所以初期 Cocos2d-x 的屏幕适配使用的是和 Cocos2d-iPhone 一样的方案，而 Cocos2d-iPhone 的适配方案遵循当时 iOS 应用开发的屏幕适配方案。

当时的 Cocos2d-iPhone 为了支持 Retina iPhone 设备，使用了-hd 等后缀来区分 iPhone 和 Retina iPhone 的图片资源。在设计游戏的时候，使用 point 坐标系，而非真正的 pixel 坐标系。这一点和苹果在 iOS native 应用开发提出的 point 概念一致，即不用修改代码，就能在 640×960 的设备上运行之前 320×480 的程序，只是图片会看起来模糊，一旦加入@2x 后缀的图片后，iOS 自动加载 @2x 的图片，实现对 Retina iPhone 的支持。

point 坐标系，在一定范围内能解决多分辨率支持的问题。但是当 iPhone 5、iPad 3 出来以后，iOS 需要适配的分辨率达到 5 个，iPhone 6 的发布更加剧了这种情况，如果要做一个 universal 的程序，是相当痛苦的。单纯的 point 坐标系并不能完全解决问题，并且 Android 设备上的分辨率情况还要复杂得多。

为了适应各种奇怪的分辨率屏幕，从 Cocos2d-x 2.0.4 开始，Cocos2d-x 提出了自己的多分辨率支持方案，废弃了之前的 retina 相关设置接口，并提出了 design resolution 的概念。design resolution 是从 point 坐标系进化过来的概念，目的是屏蔽设备分辨率，精灵坐标都在 design resolution 上布局。但要实现这个目标并不简单，Cocos2d-x 提供了一组相关的接口和 5 种分辨率适配策略，至于哪种策略才是最适合游戏的，下面将一一进行分析。

Cocos2d-x 中与分辨率相关的 C++接口如下：

```
Director::getInstance()->getOpenGLView()->setDesignResolutionSize() //设计分辨率及模式
Director::getInstance()->setContentScaleFactor()                    //内容缩放因子
FileUtils::getInstance()->setSearchPaths()                          //资源搜索路径
Director::getInstance()->getOpenGLView()->getFrameSize()            //屏幕分辨率
Director::getInstance()->getWinSize()                               //设计分辨率
Director::getInstance()->getVisibleSize()                           //设计分辨率可视区域大小
Director::getInstance()->getVisibleOrigin()                         //设计分辨率可视区域起点
```

### 2. 基本原则

Cocos2d-x 中图片显示到屏幕有下面两个逻辑过程：

(1) 资源布局到设计分辨率。

(2) 设计分辨率布局到屏幕。

如图 3-44 所示。

**注**：本章后面的图例中的数字计算，均以这张图上的数值为基础，并且 AnchorPoint 为

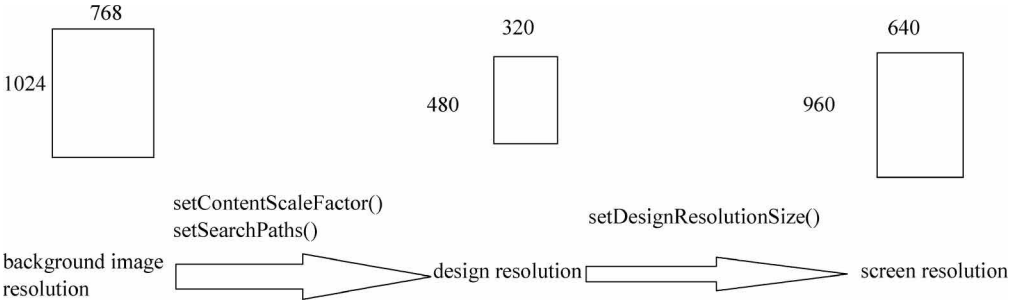



图 3-44 分辨率适配过程

(0.5,0.5),position 为屏幕中心点。

接口 setContentScaleFactor() 和 setSearchPaths() 控制着第一个转换过程。而 setDesignResolutionSize()控制第二个过程。两个过程结合在一起,影响最终的显示效果。

为了方便描述,本章后面采用以下简写:

(1) Resources width 以下简写为 RW,Resources height 以下简写为 RH。

(2) Design width 以下简写为 DW,Design height 以下简写为 DH。

(3) Screen width 以下简写为 SW,Screen height 以下简写为 SH。

**3. 从资源分辨率到设计分辨率**

setSearchPaths()需要根据当前屏幕分辨率来设置最合适的资源文件搜索路径。Cocos2d-x 允许一个游戏中包含多套图片资源,它们对应不同分辨率下的图片,程序可以选择与屏幕分辨率最匹配的图片资源来显示,从而减少图片缩放、提升游戏画质。

setContentScaleFactor()决定了图片显示到设计分辨率的缩放因子,Cocos2d-x 引擎避免游戏开发者直接去关注屏幕大小,所以这个因子是资源宽比设计分辨率宽或资源高比设计分辨率高,即 RH/DH 或 RW/DW,两种不同的因子选择有不同的缩放副作用,如图 3-45 所示。

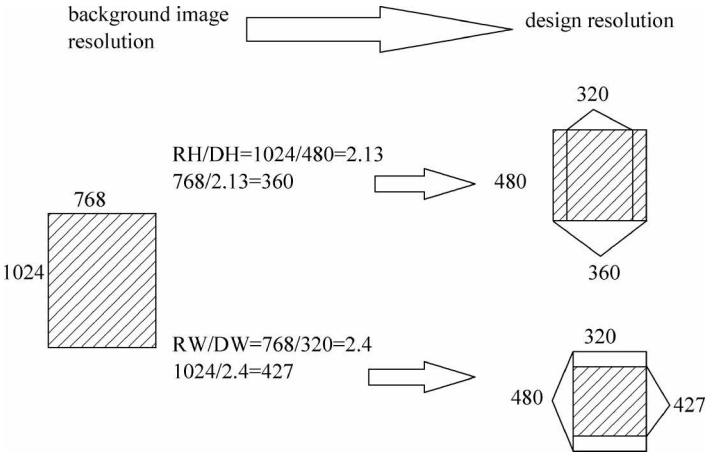



图 3-45 资源分辨率到设计分辨率

如图 3-45 中斜线部分所示：768×1024 的背景图片，采用不同的缩放因子，在 320×480 的设计分辨率上有不同的映射关系。

(1) 用高度比作为内容缩放因子，保证了资源的垂直方向在设计分辨率范围内的全部显示。

(2) 用宽度比作为内容缩放因子，保证了资源的水平方向在设计分辨率范围内的全部显示。

**4. 从设计分辨率到屏幕分辨率**

```
setDesignResolutionSize(DW, DH, resolutionPolicy)
```

有三个参数：设计分辨率宽、设计分辨率高和分辨率策略。前两个很好理解，复杂点在分辨率策略的选择上。Cocos2d-x 中共有 5 种策略模式。

1) ResolutionPolicy::SHOW_ALL

缩放因子为 MIN(SW/DW,SH/DH)。保证了设计区域全部显示到屏幕上，但可能会有黑边。

2) ResolutionPolicy::EXACT_FIT

SW/DW 作为 $X$ 方向的缩放因子，SH/DH 作为 $Y$ 方向的缩放因子。保证了设计区域完全铺满屏幕，但是可能会出现图像拉伸。

3) ResolutionPolicy::NO_BORDER

缩放因子为 MAX(SW/DW,SH/DH)。保证了设计区域总能一个方向上铺满屏幕，而另一个方向一般会超出屏幕区域。

4) ResolutionPolicy::FIXED_HEIGHT

保持传入的设计分辨率高度不变，根据屏幕分辨率修正设计分辨率的宽度。保证设计分辨率能不变形映射到屏幕。

5) ResolutionPolicy::FIXED_WIDTH

保持传入的设计分辨率宽度不变，根据屏幕分辨率修正设计分辨率的高度。保证设计分辨率能不变形映射到屏幕。

ResolutionPolicy::EXACT_FIT、ResolutionPolicy::NO_BORDER、ResolutionPolicy::SHOW_ALL 这三种策略的设计分辨率都是传入值，内部不做修正。其原理如图 3-46(a)所示，其中斜线部分表示 320×480 的设计分辨率映射到不同屏幕分辨率的对应关系。

ResolutionPolicy::NO_BORDER 是之前官方推荐使用的方案，它没有拉伸图像，同时在一个方向上撑满了屏幕，但是 Cocos2d-x 2.1.3 新加入的两种策略将撼动 ResolutionPolicy::NO_BORDER 的地位。

ResolutionPolicy::FIXED_HEIGHT 和 ResolutionPolicy::FIXED_WIDTH 是 Cocos2d-x 2.1.3 以后新加入的策略模式，它们都会在内部修正传入的设计分辨率，以保证屏幕分辨率到设计分辨率无拉伸铺满屏幕。其原理如图 3-46(b)所示，其中左边是设计分辨率，270 和 568 是修正后的值。

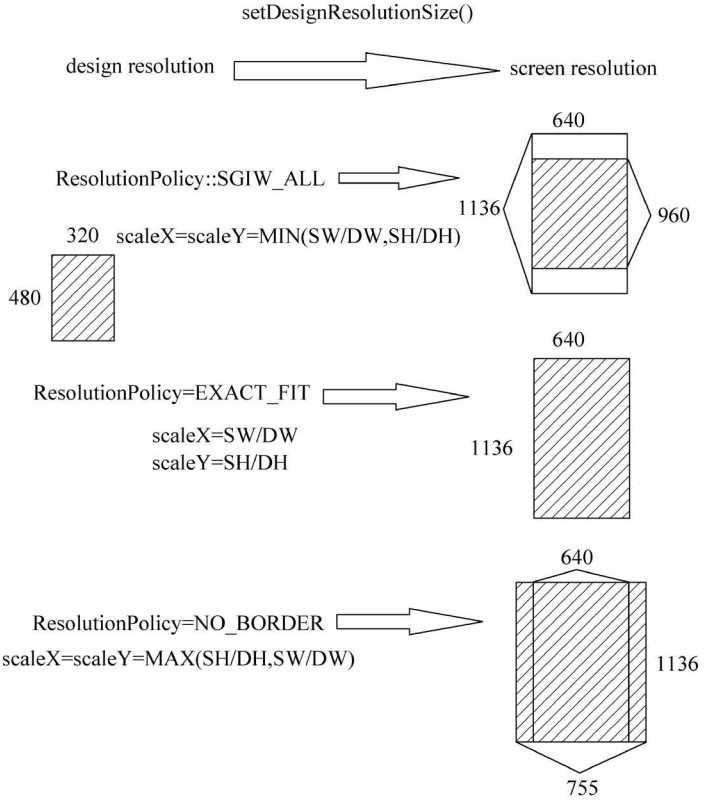



图 3-46 分辨率适配策略

**5. 结合两个过程**

第一过程有两种情况，第二过程有 5 种情况，总共有 10 种可能的方案组合。那么如何确定自己需要的方案？

需要作出选择：是牺牲效果还是牺牲部分显示区域。

这里选择牺牲一个方向的显示区域为例，结果说明两个过程。在游戏里面，背景图的高需要全部显示，而宽方向可以裁减。要实现这个目的，需要保证两个过程都是在宽方向裁减。

（1）第一过程选择 setContentScaleFactor(RH/DH)。

（2）第二过程有两个选择：ResolutionPolicy::NO_BORDER 或 ResolutionPolicy::FIXED_HEIGHT。

为了说明 NO_BORDER 与 FIXED_HEIGHT 的区别，需要结合 visibleOrigin 和 visibleSize，如图 3-47 所示。

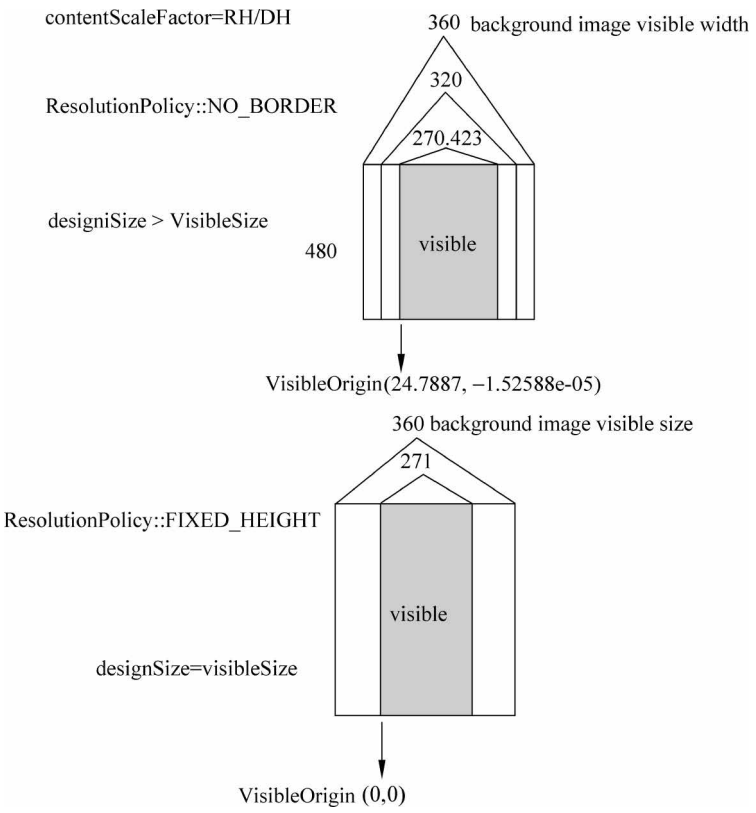



图 3-47　NO_BORDER 与 FIXED_HEIGHT 的区别

（1）ResolutionPolicy::NO_BORDER 情况下，设计分辨率范围并不全是可见区域，布局精灵需要根据 VisibleOrigin 和 VisibleSize 来做判断处理。

(2) ResolutionPolicy::FIXED_HEIGHT 则不同,设计分辨率范围就是可见区域,VisibleOrigin 总是(0,0)。getVisibleSize()= getWinSize(),ResolutionPolicy::FIXED_HEIGHT 达到了同样的目的,但是却简化了代码。

ResolutionPolicy::FIXED_HEIGHT 和 ResolutionPolicy::FIXED_WIDTH 是 ResolutionPolicy::NO_BORDER 的进化,新项目中建议立即开始使用这两种方式。

**6. 小结**

setContentScaleFactor()决定了图片显示到设计分辨率的缩放因子,Cocos2d-x 引擎避免游戏开发者直接去关注屏幕大小,所以这个因子是资源宽比设计分辨率宽或资源高比设计分辨率高,即 RH/DH 或 RW/DW,不同的因子有不同的缩放副作用。

1) ResolutionPolicy::FIXED_HEIGHT

适合高方向需要撑满,宽方向可裁减的游戏,结合 setContentScaleFactor(RH/DH)使用。

2) ResolutionPolicy::FIXED_WIDTH

适合宽方向需要撑满,高方向可裁减的游戏,结合 setContentScaleFactor(RW/DW)使用。

## 3.13.2 Cocos2d-Lua 中的多分辨率适配

前面分析了 Cocos2d-x 的多分辨率适配理论基础,并且介绍了相关的 C++接口。这些接口在 Cocos2d-Lua 有对应的 Lua 设置方法,分两个阶段。

(1) 资源分辨率到设计分辨率。

```
cc.FileUtils:getInstance():addSearchPath()            -- 设置资源搜索路径
cc.Director:getInstance():setContentScaleFactor()     -- 内容缩放因子
```

(2) 设计分辨率到屏幕分辨率,在 Cocos2d-Lua 中,设计分辨率相关设置是由 src/config.lua 文件中的相关常量来配置完成的。

```
-- 设计分辨率的宽、高
CONFIG_SCREEN_WIDTH   = 320
CONFIG_SCREEN_HEIGHT = 480

-- 适配模式: "FIXED_HEIGHT"、"FIXED_WIDTH"或"FILL_ALL"
CONFIG_SCREEN_AUTOSCALE = "FIXED_WIDTH"
```

假设需要做一个竖屏单手操作的游戏,美术已经提供了 iPhone 5 分辨率即 640×1136 的图片资源。针对这个情况,下面介绍如何进行分辨率适配的相关设置。

(1) 新建一个 Cocos2d-Lua 项目,会自动生成模板代码。打开 src/config.lua,修改设计分辨率,如下:

```
CONFIG_SCREEN_WIDTH = 320
CONFIG_SCREEN_HEIGHT = 480
```

```
CONFIG_SCREEN_AUTOSCALE = "FIXED_WIDTH"
```

第1行告诉Cocos2d-Lua引擎,游戏是竖屏的。第3行选择FIXED_WIDTH适配模式,让$X$轴方向能完全显示在屏幕上。

(2) 打开src/app/MyApp.lua,在enterScene前加入资源搜索路径和内容缩放因子。

```
cc.FileUtils:getInstance():addSearchPath("res/")
cc.Director:getInstance():setContentScaleFactor(640 / CONFIG_SCREEN_WIDTH)
```

**注**:主流的开发趋势,已经不再放多套分辨率的资源在游戏中。通常是选择一个主流的分辨率,如1334×750,contentScaleFactor也无须修改。

由于只打算放一套资源在最终的游戏包中,所以资源文件都直接放置在res目录下,如果愿意,可以修改为res/w640这样更清晰的文件夹名称。

由于选择了FIXED_WIDTH适配模式,所以内容缩放因子是RW/DW。

经过上面的设置,引擎已经准备好了竖屏游戏的开发。但是要游戏能自适应各种分辨率,在游戏开发中,还需要遵循下面的原则。

(1) 资源分辨率得尽可能地覆盖可遇见机型的最大宽高比,目前主流手机都是9:16的分辨率,所以选择了640×1136来做美术资源。这样在引擎进行分辨率自适配的裁剪过程中,不会出现黑边。尽管背景图片可能无法完全呈现,但是可通过一些美术手段来优化。例如在FIXED_WIDTH模式下,让背景图上下边缘渐变填充,这样即使被裁剪也无伤大雅。

(2) 游戏场景的背景图,均以屏幕中点来设置position。

(3) 其他精灵元素的位置坐标,应以设计分辨率上的9点相对坐标为基准来进行布局。在元素最靠近的点的坐标上添加偏移量,来确定精灵的坐标。

Quick框架的display模块提供了9点坐标值的便捷获取方式,以及其他与分辨率适配相关的常量如下:

(1) display.widthInPixels,屏幕分辨率的宽。

(2) display.heightInPixels,屏幕分辨率的高。

(3) display.contentScaleFactor,设计分辨率到屏幕分辨率的缩放因子,不同于内容缩放因子。

(4) display.width,设计分辨率的宽。

(5) display.height,设计分辨率的高。

(6) display.cx,设计分辨率中央的$x$坐标。

(7) display.cy,设计分辨率中央的$y$坐标。

(8) display.left,设计分辨率最左坐标。

(9) display.top,设计分辨率最上坐标。

(10) display.right,设计分辨率最右坐标。

(11) display.bottom,设计分辨率最下坐标。

结合 display 模块提供的常量，9 点坐标速查表如图 3-48 所示。

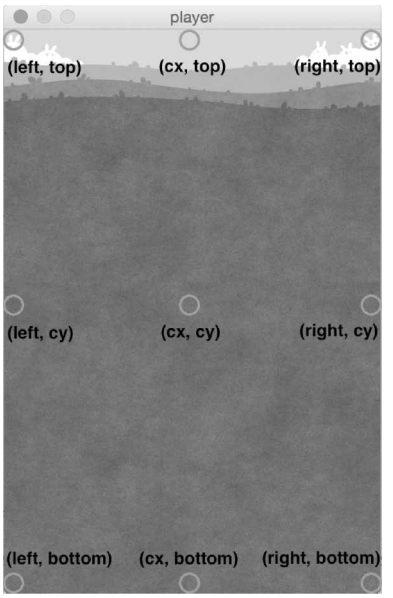



图 3-48　9 点坐标

**注**：图示坐标仅适用于父节点为设计分辨率等大的节点布局。