第 3 章

Cocos2d-x 引擎

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、动画、人工智能、网络引擎以及场景管理。

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 Unity3D、Unreal Development Kit、ShiVa 3D 和 Marmalade。此外，还有一些针对 HTML 5 的游戏引擎：Cocos2d-html5、X-Canvas 和 Sphinx 等。

这些游戏引擎各有千秋，但是目前得到市场普遍认可 2D 引擎是 Cocos2d-x，3D 引擎是 Unity3D。

3.1 Cocos2d 家谱

在我们介绍 Cocos2d-x 之前，有必要先介绍一下 Cocos2d 的“家谱”，如图 3-1 所示。

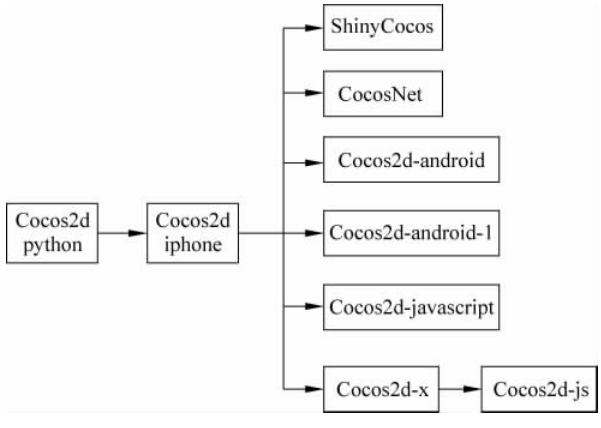

图 3-1　Cocos2d 的“家谱”

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

- ShinyCocos：使用Ruby对Cocos2d-iphone进行封装，使用Ruby api开发。
- CocosNet：在MonoTouch平台上使用Cocos2d引擎，采用.NET实现。
- Cocos2d-android：为Android平台使用的Cocos2d引擎，采用Java实现。
- 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。
- Cocos2d-javascript：采用JavaScript脚本语言实现。
- 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。
- Cocos2d-js：采用JavaScript API的Cocos2d引擎，它可以绑定在Cocos2d-x上开发基于本地技术的游戏；它也可以依托浏览器运行，开发基于Web的网页游戏，它也是由Cocos2d-x团队开发的分支项目。

此外，历史上Cocos2d还出现过很多其他分支，随着技术的发展这些分支逐渐消亡了，其中最有生命力的当属Cocos2d-x引擎。

3.2 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、Windows Phone、Bada、BlackBerry和MeeGo等。纵向方面向下能够支持OpenGL ES 1.1、OpenGL ES 1.5和OpenGL ES 2.0，以及DirectX 11等技术，向上支持JavaScript和Lua脚本绑定。

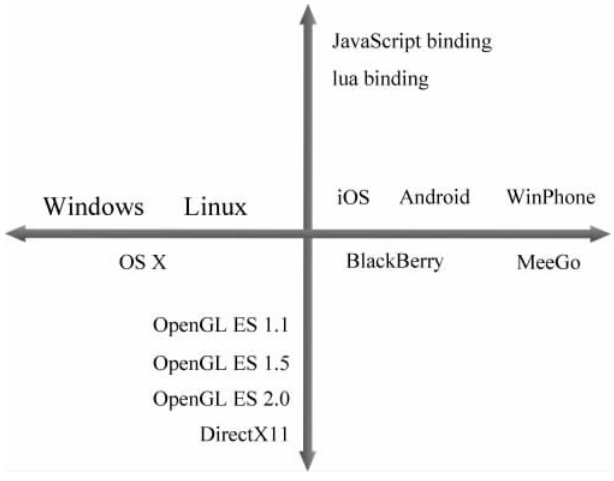

图3-2 Cocos2d-x设计目标

简单地说，Cocos2d-x 的设计目标是为了实现跨平台，使我们不再为同一款游戏在不同平台发布而进行编译了。而且 Cocos2d-x 为程序员考虑得更多，很多程序员可能对 C++不熟悉，针对这种情况可以使用 JavaScript 和 Lua[①] 开发游戏，如图 3-3 所示。

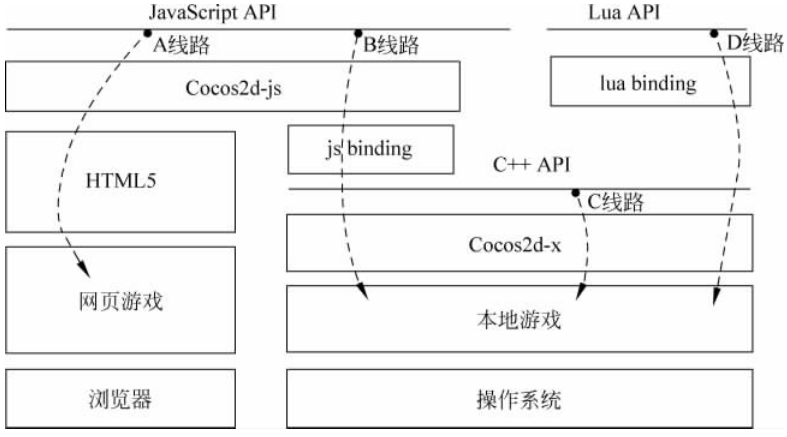

图 3-3　Cocos2d-x 绑定 JavaScript 和 Lua 脚本

由图 3-3 可见，通过 Cocos2d-x 和 Cocos2d-js 引擎，程序员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 脚本的程序员准备的，通过 A 线路，使用 Cocos2d-js 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)技术透过 C++API 访问 Cocos2d-x 引擎，开发本地游戏。这样，同样的 JavaScript 代码就可以实现在不同平台下运行了。

图 3-3 中的 C 线路是本书介绍的，它是给熟悉 C++的程序员准备的。通过 C++API 访问 Cocos2d-x 引擎，开发本地游戏。

图 3-3 中的 D 线路是为熟悉 Lua 脚本的程序员准备的。使用 lua binding(lua 绑定)技术通过 C++API 访问 Cocos2d-x 引擎，开发本地游戏。

通过 Cocos2d-x 和 Cocos2d-js 引擎，Cocos2d-x 团队构建了自己的技术生态圈，通过这些引擎使得我们开发游戏越来越简单。

3.3　第一个 Cocos2d-x 游戏

我们编写的第一个程序一般都命名为 HelloWorld，从它开始再学习其他的内容。这一节介绍的第一个 Cocos2d-x 游戏也命名为 HelloWorld。

---

① Lua 是一个小巧的脚本语言。是巴西里约热内卢天主教大学(Pontifical Catholic University of Rio de Janeiro)里的一个研究小组于 1993 年开发的，它由 Roberto Ierusalimschy、Waldemar Celes 和 Luiz Henrique de Figueiredo 所组成。——引自于百度百科 http://baike.baidu.com/view/416116.htm?fr=wordsearch

3.3.1 创建工程

在 Cocos2d-x 早期版本中，创建工程是通过安装在 Visual Studio 中的工程模板创建的，而 Cocos2d-x 3.x 创建工程是通过 Cocos2d-x 提供的命令工具 cocos 实现的。配置好环境后，进入终端，并从终端进入 cocos 目录，在终端中执行如下指令：

```
cocos new HelloWorld -p com.work6 -l cpp -d <工程生成目录>
```

通过上面的指令，我们在<工程生成目录>下面生成了名为 HelloWorld 的 Cocos2d-x 工程。打开 HelloWorld 目录，其中的内容如图 3-4 所示。

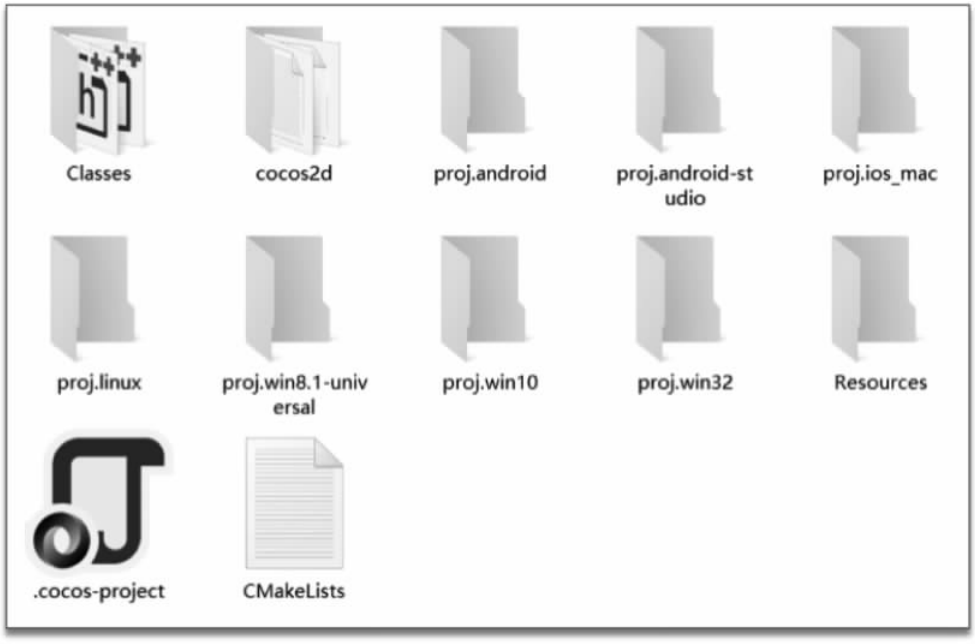

图 3-4 HelloWorld 工程中的内容

从图 3-4 中可以看出，生成的工程代码是适合于多平台的，其中 Classes 目录是放置一些通用类(与平台无关的)，我们编写的 C++代码主要放置在该目录下面。图 3-4 中 cocos2d 目录是放置 Cocos2d-x 引擎的源代码，其中包括了音效引擎、物理引擎等。

图 3-4 中的 proj.android、proj.android-studio、proj.ios_mac、proj.linux、proj.win8.1-universal、proj.win10 和 proj.win32 目录是放置与特定平台有关系的代码，其中 proj.android 和 proj.android-studio 是 android 平台特定代码；proj.ios_mac 是 iOS 和 Mac OS 运行需要的特定代码。proj.win32 是 Win32 平台运行所需要的特定代码，它可以在 Windows 下运行，模拟器是 Win32 窗口；proj.win8.1-universal 是 Windows Phone 8.1 平台运行所需要的特定代码；proj.win10 是 Windows 平台运行所需特定代码。proj.linux 是

Linux 平台运行所需要的特定代码。

图 3-4 中 Resources 目录是放置工程需要的资源文件，这个目录中的内容是共享于全部平台下的。

3.3.2 Visual Studio 工程文件结构

如果是在 Windows 下，可以通过 Visual Studio 工具编译和运行 Cocos2d-x 工程，通过 Visual Studio 工具打开 proj.win32 目录下面的 Visual Studio 解决方案 HelloWorld.sln 来进行编译和运行。

进入到 proj.win32 目录下，双击 HelloWorld.sln 解决方案文件，启动 HelloWorld 界面，如图 3-5 所示。

在图 3-5 所示的解决方案中，HelloWorld 工程的 src 文件夹中的内容是与图 3-4 的 Classes 目录内容对应的。HelloWorld 工程的 win32 文件夹中的 main.cpp 和 main.h 是 win32 平台特有程序代码，通过它启动 Win32 窗口。

如果想看一下效果，可以单击 ▶ 本地 Windows 调试器 按钮运行，如图 3-6 所示。

图 3-5 在 Visual Studio 中启动 HelloWorld.sln 解决方案

图 3-6 程序运行效果

3.3.3 Xcode 工程文件结构

如果在 Mac OS X 下，可以通过 Xcode 工具编译和运行 Cocos2d-x 工程，通过 Xcode 工具打开 proj.ios_mac 目录下面的 HelloWorld.xcodeproj 工程文件。

进入到 proj.ios_mac 目录下，双击 HelloWorld.xcodeproj 工程文件，启动 HelloWorld 界面，如图 3-7 所示。

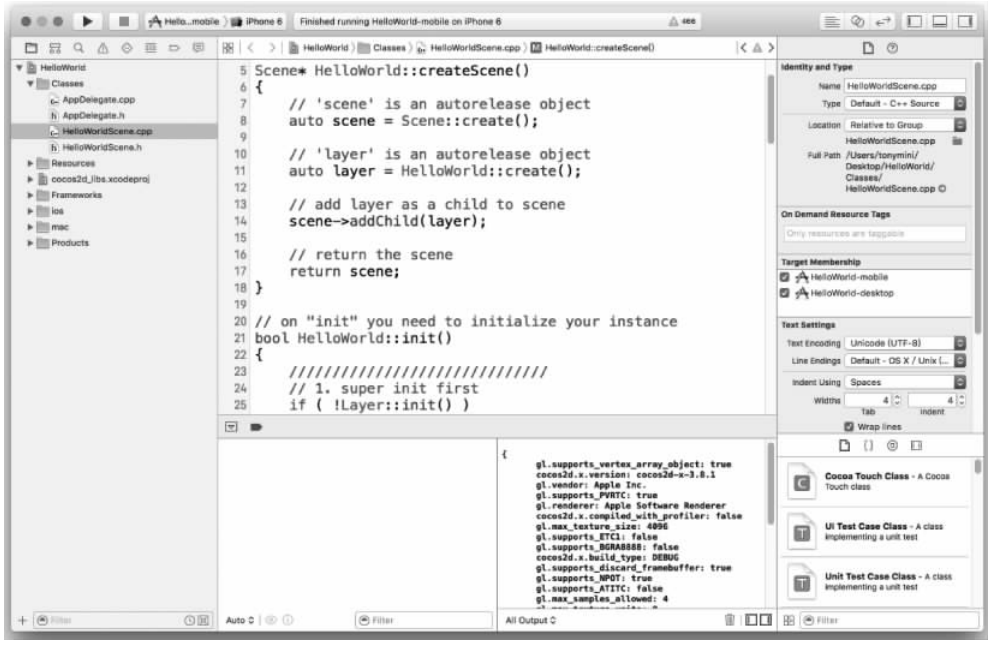

图 3-7 在 Xcode 中启动 HelloWorld. xcodeproj 工程

图 3-7 所示的 Xcode 中 HelloWorld 工程的 Classes 组中的内容是与图 3-4 的 Classes 目录内容对应的。HelloWorld 工程的 ios 组是 iOS 平台特有程序代码，mac 组是 Mac OS X 平台特有程序代码。

如果想看一下效果，可以单击工具栏中的 ▶ 按钮运行，在 iPhone 6 模拟器上运行的效果如图 3-8 所示。

图 3-8 程序运行效果

3.3.4 代码解释

下面解释一下 Cocos2d-x 工程源程序文件，它们是位于 Classes 文件夹中的 4 个文件：AppDelegate.h、AppDelegate.cpp、HelloWorldScene.h 和 HelloWorldScene.cpp。

1. AppDelegate 类

在 AppDelegate.h 和 AppDelegate.cpp 中分别声明和定义了 AppDelegate 类，AppDelegate 类是 Cocos2d-x 引擎要求实现的游戏应用委托对象，在 Cocos2d-x 游戏运行的不同生命周期阶段会触发它的不同函数。

AppDelegate.h 代码如下：

```
#ifndef _APP_DELEGATE_H_
#define _APP_DELEGATE_H_

#include "cocos2d.h"

class AppDelegate : private cocos2d::Application
{
public:
    AppDelegate();
    virtual ~AppDelegate();

    virtual void initGLContextAttrs();

    /*
     * 游戏启动时调用的函数,在这里可以初始化导演对象和场景对象
     */
    virtual bool applicationDidFinishLaunching();

    /*
     * 游戏进入后台时调用的函数
     */
    virtual void applicationDidEnterBackground();

    /*
     * 游戏进入前台时调用的函数
     */
    virtual void applicationWillEnterForeground();
};

#endif //_APP_DELEGATE_H_
```

从上面的代码可以看到，AppDelegate 继承了 cocos2d::Application，cocos2d::Application 是 Cocos2d-x 引擎提供的基类。

AppDelegate.cpp 代码如下：

```
#include "AppDelegate.h"
#include "HelloWorldScene.h"

USING_NS_CC;                                                                    ①

static cocos2d::Size designResolutionSize = cocos2d::Size(480, 320);            ②
static cocos2d::Size smallResolutionSize = cocos2d::Size(480, 320);
static cocos2d::Size mediumResolutionSize = cocos2d::Size(1024, 768);
static cocos2d::Size largeResolutionSize = cocos2d::Size(2048, 1536);           ③

AppDelegate::AppDelegate() {

}

AppDelegate::~AppDelegate()
{
}

//设置 OpengGL 上下文属性
void AppDelegate::initGLContextAttrs()
{
    //设置 OpenGL 上下文属性,现在可以设置 6 个属性
    //red,green,blue,alpha,depth(深度缓存),stencil(模板缓存)
    GLContextAttrs glContextAttrs = {8, 8, 8, 8, 24, 8};

    GLView::setGLContextAttrs(glContextAttrs);
}

//如果使用包管理安装更多的包,不要修改或删除该函数
static int register_all_packages()
{
    return 0; //flag for packages manager
}

bool AppDelegate::applicationDidFinishLaunching() {                             ④
    //初始化 director
    auto director = Director::getInstance();                                    ⑤
    auto glview = director->getOpenGLView();
    if(!glview) {
#if (CC_TARGET_PLATFORM == CC_PLATFORM_WIN32) || (CC_TARGET_PLATFORM == CC_PLATFORM_MAC)
|| (CC_TARGET_PLATFORM == CC_PLATFORM_LINUX)                                     ⑥
        glview = GLViewImpl::createWithRect("HelloWorld", Rect(0, 0,
                 designResolutionSize.width, designResolutionSize.height));
#else
        glview = GLViewImpl::create("HelloWorld");
#endif
        director->setOpenGLView(glview);                                        ⑦
```

```
    }

    director->setDisplayStats(true);                                    ⑧
    director->setAnimationInterval(1.0 / 60);                           ⑨

    //设置设计分辨率
    glview->setDesignResolutionSize(designResolutionSize.width,
                designResolutionSize.height, ResolutionPolicy::NO_BORDER);  ⑩
    Size frameSize = glview->getFrameSize();
    if (frameSize.height > mediumResolutionSize.height)
    {
            director->setContentScaleFactor(MIN(largeResolutionSize.height/
designResolutionSize.height, largeResolutionSize.width/designResolutionSize.width));
    }
    else if (frameSize.height > smallResolutionSize.height)
    {
            director->setContentScaleFactor(MIN(mediumResolutionSize.height/
designResolutionSize.height, mediumResolutionSize.width/designResolutionSize.width));
    }
    else
    {
            director->setContentScaleFactor(MIN(smallResolutionSize.height/
designResolutionSize.height, smallResolutionSize.width/designResolutionSize.width));
    }                                                                   ⑪

    register_all_packages();                                            ⑫

    auto scene = HelloWorld::createScene();                             ⑬

    //run
    director->runWithScene(scene);                                      ⑭

    return true;
}

void AppDelegate::applicationDidEnterBackground() {                     ⑮
    Director::getInstance()->stopAnimation();                           ⑯

    //如果使用 SimpleAudioEngine 播放音乐,则在该方法中暂停
    //SimpleAudioEngine::getInstance()->pauseBackgroundMusic();         ⑰
}
void AppDelegate::applicationWillEnterForeground() {                    ⑱
    Director::getInstance()->startAnimation();                          ⑲

    //如果使用 SimpleAudioEngine 播放音乐,则在该方法中继续
    //SimpleAudioEngine::getInstance()->resumeBackgroundMusic();        ⑳
}
```

上述代码第①行的 USING_NS_CC 是 Cocos2d-x 提供了一个宏，它的作用是用来替换 using namespace cocos2d 语句，随着我们学习深入，你会发现 Cocos2d-x 定义了很多宏，使用起来很方便。

代码第②行～第③行定义了几种分辨率尺寸。

代码第④行定义 applicationDidFinishLaunching()函数是游戏程序启动时调用的函数。

代码第⑤行是初始化导演类 Director，代码第⑥行是通过判断当前运行的平台。代码第⑦行是设置导演类的 OpenGL 视图。代码第⑧行设置是否在屏幕上显示帧率等信息，60.1 就是当前的帧率，显示帧率一般是为了测试，实际发布时设置为不显示的，它会影响游戏的外观。第⑨行代码是设定计时器 1.0 / 60 秒间隔一次，即设定平均帧率为 60。

代码第⑩行～第⑪行是设置解决屏幕适配问题，有关知识将会在后面章节介绍。

代码第⑫行调用 register_all_packages()管理安装。

代码第⑬行创建场景对象 Scene，第⑭行代码是运行该场景，这会使游戏进入该场景。

代码第⑮行定义的 applicationDidEnterBackground()函数是游戏进入后台时调用的函数。第⑯行代码是停止场景中的动画。第⑰行代码是暂停背景音乐，默认是注释掉的，游戏中如果使用 SimpleAudioEngine 播放音乐，则在该方法中暂停播放音乐。

代码第⑱行定义的 applicationWillEnterForeground()函数是游戏进入前台时调用的函数。第⑲行代码是开始场景中的动画。第⑳行代码是继续背景音乐，默认是注释掉的，如果使用 SimpleAudioEngine 播放音乐，则在该方法中实现继续播放音乐。

2. HelloWorld 类

在 HelloWorldScene.h 和 HelloWorldScene.cpp 中分别声明和定义了 HelloWorld 类，HelloWorld 类继承了 cocos2d::Layer 类，它被称为层(Layer)，这些层被放到场景(Scene)中，场景类是 cocos2d::Scene。注意，HelloWorldScene.h 虽然命名为场景，但是它内部定义的 HelloWorld 类是一个层。

HelloWorldScene.h 的代码如下：

```
#ifndef __HELLOWORLD_SCENE_H__
#define __HELLOWORLD_SCENE_H__

#include "cocos2d.h"

class HelloWorld : public cocos2d::Layer                                    ①
{
public:

    static cocos2d::Scene* createScene();                                   ②
    virtual bool init();                                                    ③

    void menuCloseCallback(cocos2d::Ref* pSender);                          ④
```

```
    CREATE_FUNC(HelloWorld);                                                    ⑤
};

#endif //__HELLOWORLD_SCENE_H__
```

从上面的代码第①行可以看出，HelloWorld 继承了 cocos2d::Layer，HelloWorld 是一个层，而不是场景。第②行代码是声明创建当前层 HelloWorld 所在场景的静态函数 createScene()。第③行代码是声明初始化层 HelloWorld 实例函数。第④行代码是声明菜单回调函数 menuCloseCallback，用于触摸菜单事件的回调。第⑤行代码中的 CREATE_FUNC 是一个 Cocos2d-x 定义的宏，它的作用相当于如下代码：

```
static HelloWorld* create()
{
    HelloWorld *pRet = new HelloWorld();
    if (pRet && pRet->init())
    {
        pRet->autorelease();
        return pRet;
    }
    else
    {
        delete pRet;
        pRet = NULL;
        return NULL;
    }
}
```

从上述代码可见，CREATE_FUNC 宏的作用是创建一个静态函数 create，该函数可以用来创建层。

下面我们分别解释一下 HelloWorldScene.cpp 中的几个函数。

HelloWorldScene.cpp 中 createScene()代码如下：

```
Scene* HelloWorld::createScene()
{
    auto scene = Scene::create();                                               ①
    auto layer = HelloWorld::create();                                          ②
    scene->addChild(layer);                                                     ③
    return scene;
}
```

createScene()函数是在游戏应用启动的时候，在 AppDelegate 的 applicationDidFinishLaunching()函数中通过 auto scene = HelloWorld::createScene()语句调用的。在 createScene()中做了三件事情，首先创建了 HelloWorld 层所在的场景对象（见代码第①行），其次创建了 HelloWorld 层（见代码第②行），最后将 HelloWorld 层添加

到场景 scene 中(见代码第③行)。

当调用 HelloWorld::create()语句创建层的时候,会调用 HelloWorld 的实例函数 init(),达到初始化 HelloWorld 层的目的,init ()代码如下:

```
bool HelloWorld::init()
{
    //////////////////////////////
    //1. 初始化父类
    if ( !Layer::init())                                                    ①
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();           ②
    Vec2 origin = Director::getInstance()->getVisibleOrigin();              ③

    //////////////////////////////
    //2. 增加一个菜单项,单击它的时候退出程序
    auto closeItem = MenuItemImage::create(
                                        "CloseNormal.png",
                                        "CloseSelected.png",
                                CC_CALLBACK_1(HelloWorld::menuCloseCallback, this));
                                                                            ④

    closeItem->setPosition(Vec2(origin.x + visibleSize.width - closeItem->getContentSize().
width/2 ,origin.y + closeItem->getContentSize().height/2));                 ⑤
    auto menu = Menu::create(closeItem, NULL);                              ⑥
    menu->setPosition(Vec2::ZERO);                                          ⑦
    this->addChild(menu, 1);                                                ⑧

    //////////////////////////////
    //3. 在下面添加你自己的代码

    auto label = Label::createWithTTF("Hello World", "fonts/Marker Felt.ttf", 24);
                                                                            ⑨
    label->setPosition(Vec2(origin.x + visibleSize.width/2,
                            origin.y + visibleSize.height - label->getContentSize().height));
                                                                            ⑩
    this->addChild(label, 1);                                               ⑪

    auto sprite = Sprite::create("HelloWorld.png");                         ⑫
    sprite->setPosition(Vec2(visibleSize.width/2 + origin.x, visibleSize.height/2 +
origin.y));                                                                 ⑬
    this->addChild(sprite, 0);                                              ⑭
```

```
    return true;
}
```

在 init()函数中主要是初始化 HelloWorld 层,其中包括了里面的精灵、菜单和文字等内容。其中第①行代码是初始化父类 Layer,返回 true 则初始化成功,false 则初始化失败。

第②行代码是视图的可视化尺寸,第③行代码是视图的可视化原点。

第④行代码是创建一个图片菜单项对象,单击该菜单项的时候回调 menuCloseCallback 函数。第⑤行代码是菜单项对象的位置,第⑥行代码是创建 Menu 菜单对象。第⑦行代码是菜单对象的位置。第⑧行代码是把菜单对象添加到当前 HelloWorld 层上,如图 3-6 或图 3-8 所示,运行界面右下角的 ⏻ 就是刚刚添加的菜单对象。

第⑨～⑪行代码是将一个 Hello World 标签对象放置到层中,这个过程是:创建对象→设置对象的位置→把对象添加到层上。第⑨行代码是创建一个 LabelTTF 标签对象。第⑩行代码是设置标签对象位置为水平居中,在垂直方向上与屏幕顶对齐。第⑪行代码是将文本对象添加到层 HelloWorld 上。

第⑫行代码是创建精灵 Sprite 对象,它是图 3-6 中的 Cocos2d-x 的 logo 图标。第⑬行代码是设置精灵对象的位置,这个位置是屏幕的中央。第⑭行代码是将精灵对象添加到层 HelloWorld 上。

菜单回调函数 menuCloseCallback 代码如下:

```
void HelloWorld::menuCloseCallback(Ref * pSender)
{
    Director::getInstance() -> end();

#if (CC_TARGET_PLATFORM == CC_PLATFORM_IOS)                    ①
    exit(0);
#endif
}
```

menuCloseCallback 函数中使用了条件编译语句代码①行所示,用来判断是当前程序运行的是哪个平台,其中 CC_TARGET_PLATFORM 宏保存游戏运行的当前目标平台,CC_PLATFORM_IOS 宏表示 iOS 平台,此外还有定义很多平台的宏,如果需要大家可以查询 API。

3.3.5 Win32 平台下设置屏幕

我们在 Win32 平台运行游戏是为了进行测试游戏应用,那么如何改变屏幕的大小呢?可以修改 AppDelegate.cpp 中的 smallResolutionSize 和 designResolutionSize 常量值代码如下:

```
#include "AppDelegate.h"
#include "HelloWorldScene.h"
```

```
USING_NS_CC;

static cocos2d::Size designResolutionSize = cocos2d::Size(900, 640);                    ①
static cocos2d::Size smallResolutionSize = cocos2d::Size(900, 640);                     ②
static cocos2d::Size mediumResolutionSize = cocos2d::Size(1024, 768);
static cocos2d::Size largeResolutionSize = cocos2d::Size(2048, 1536);

AppDelegate::AppDelegate() {

}

AppDelegate::~AppDelegate()
{
}

void AppDelegate::initGLContextAttrs()
{
    ...
    return true;
}
...
```

在上述代码中，将代码第①行的 designResolutionSize 和 smallResolutionSize 常量都设置为(900, 640)，这里我们暂时不做解释，在后面的章节我们再解释。

3.3.6 工程中添加资源文件

在游戏开发过程中经常需要向工程中添加资源文件(图片、声音、视频和配置文件等)，如果在 Windows 下通过 Visual Studio 工具运行 Win32 工程，我们把这些资源文件复制到资源管理器的 Resources 目录下就可以了。

而如果在 Mac OS X 下通过 Xcode 工具运行工程，则需要将这些资源文件添加到 Xcode 工程中，添加到工程的具体步骤是：在工程导航面板中，右键选择 Resources 组，弹出右键菜单如图 3-9 所示，选择菜单中的 Add Files to "HelloWorld"弹出选择文件对话框，如图 3-10 所示，选中 Destination→Copy items if needed 可以使文件或文件夹从原始位置复制到当前工程目录中。Added foldes 项目适用于添加文件夹时候，选中 Create groups 时，该文件夹将在 Xcode 中作为组(group)，组在 Xcode 中颜色为黄色；选中 Create folder references 时，该文件夹将在 Xcode 中作为文件夹表示，文件夹在 Xcode 中颜色为蓝色。在 Add to targets 选

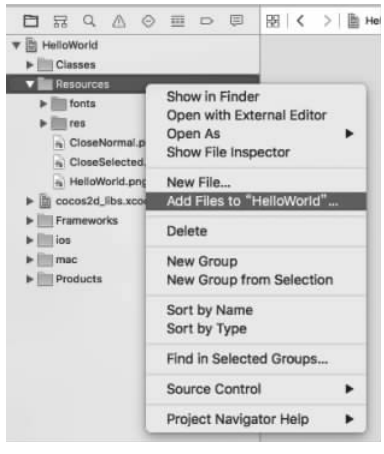

图 3-9 添加资源文件到工程

中其中的 Targets，可以使这些资源文件编译到 Targets 中，并随产品一起发布。

提示 Xcode 中一个 Targets 编译之后是一个产品，这个产品可能是一个可执行程序包，也可以是一个库或框架。

图 3-10 选择资源文件对话框

在图 3-10 所示的对话框中选中要添加的资源文件或文件夹后，单击 Add 按钮添加这些文件到 Xcode，如图 3-11 所示，添加了两个图片文件到 Xcode 的 Resources 组下面。

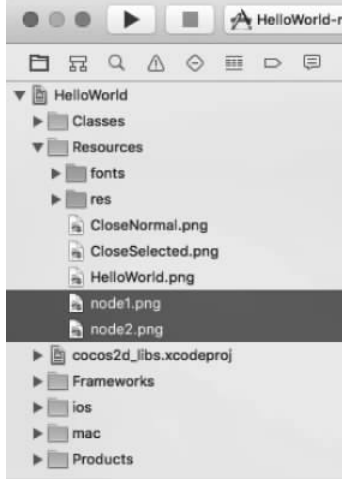

图 3-11 添加资源文件完成

3.4 Cocos2d-x 核心概念

Cocos2d-x 中有很多概念，这些概念多来源于动画、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 中的核心概念如下：

- 导演
- 场景
- 层
- 节点
- 精灵
- 菜单
- 动作
- 效果
- 粒子运动
- 地图
- 物理引擎

本节我们介绍导演、场景和层概念以及对应的类。由于节点概念很重要，我们会在下一节继续介绍，而其他的概念则在后面的章节中介绍。

3.4.1 导演

导演类 Director(v3.0 之前是 CCDirector)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：

```
auto director = Director::getInstance();
```

导演对象职责如下：

- 访问和改变场景；
- 访问 Cocos2d-x 的配置信息；
- 暂停、继续和停止游戏；
- 转换坐标。

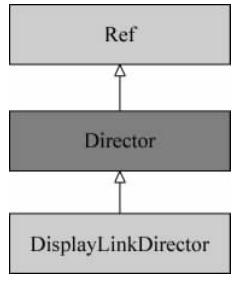

图 3-12 Director 类图

Director 类图如图 3-12 所示。

从图 3-12 所示的类图中还可以看到它有一个子类：DisplayLinkDirector。

3.4.2 场景

场景类 Scene(v3.0 之前是 CCScene)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：

- 展示类场景：播放视频或简单的在图像上输出文字，来实现游戏的开场介绍、胜利和失败提示、帮助介绍。
- 选项类场景：主菜单、设置游戏参数等。
- 游戏场景：这是游戏的主要内容。

场景类 Scene 的类图如图 3-13 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，其中包括 Scene、Layer 等。

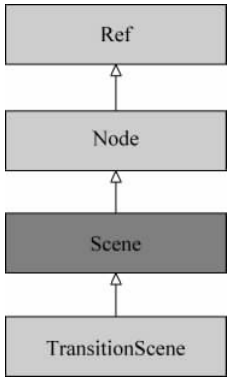

图 3-13　Scene 类图

3.4.3　层

层是我们编写游戏程序的重点，我们大约会花费 99％以上的时间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 3-14 是一个简单的主菜单界面，它是由三个层叠加实现的。

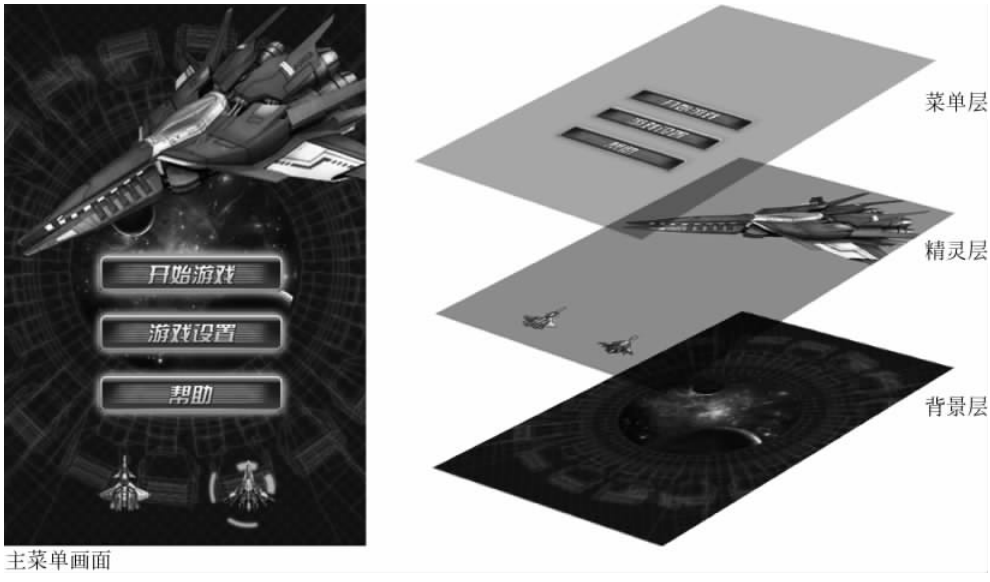

图 3-14　层叠加

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，从上到下依次是：菜单层→精灵层→背景层。Cocos2d-x 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，则排在后面的层将不再接收到该事件了。每一层又可以包括很多各式各样的内容要素：文本、链接、精灵、地图等。

层类 Layer 的类图如图 3-15 所示。

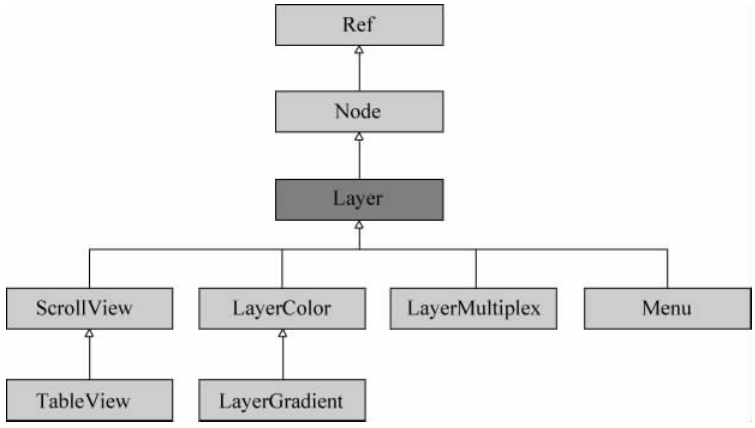

图 3-15 Layer 类图

3.5 Node 与 Node 层级架构

Cocos2d-x 采用层级（树形）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。一个场景可以包含多个层，一个层又可以包含多个精灵、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 3-16 所示。

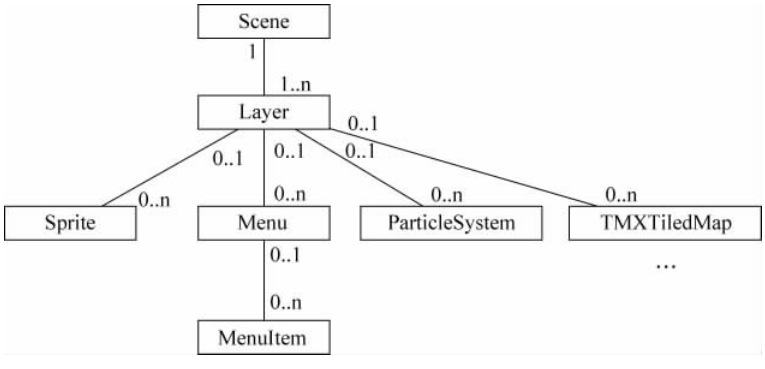

图 3-16 节点的层级结构

这些节点有一个共同的父类 Node，Node 类图如图 3-17 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。

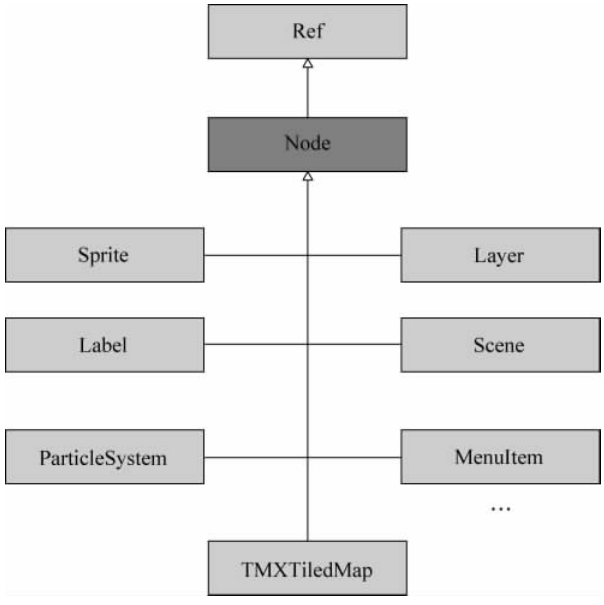

图 3-17　Node 类图

3.5.1　Node 中的重要操作

Node 作为根类，它有很多重要的函数。下面分别介绍一下：

- 创建节点：Node * childNode = Node::create()。
- 增加新的子节点：node->addChild (childNode, 0, 123)，第二个参数 Z 轴绘制顺序，第三个参数是标签。
- 查找子节点：Node * node = node->getChildByTag(123)，通过标签查找子节点。
- node->removeChildByTag(123, true)通过标签删除子节点，并停止所有该节点上的一切动作。
- node->removeChild(childNode, true)删除 childNode 节点，并停止所有该子节点上的一切动作。
- 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。
- node->removeFromParentAndCleanup(true)从父节点删除 node 节点，并停止所有该节点上的一切动作。

3.5.2 Node中的重要属性

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上，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3-18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)。

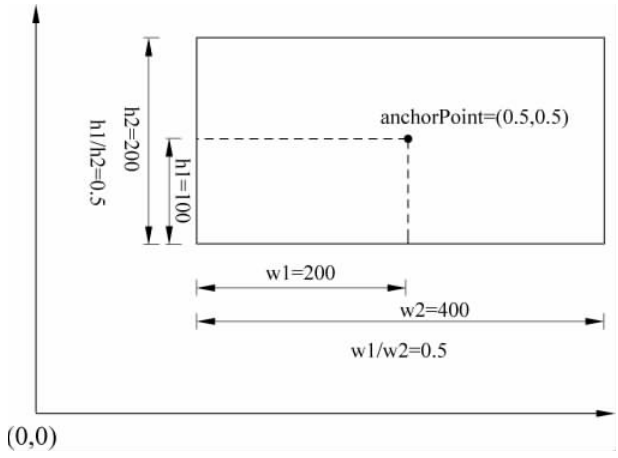

图3-18 anchorPoint为(0.5,0.5)

图3-19是anchorPoint为(0.66，0.5)的情况。

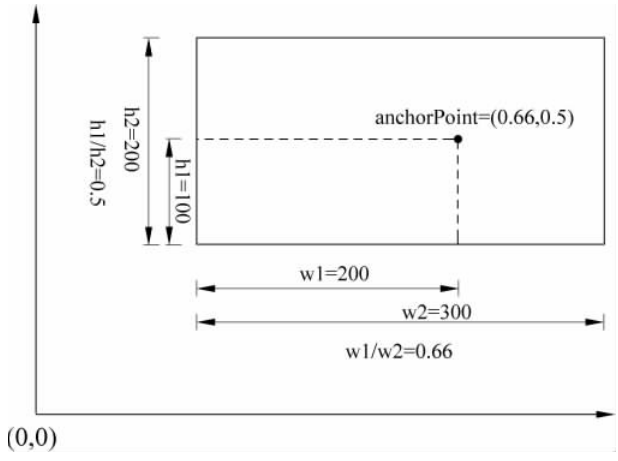

图3-19 anchorPoint为(0.66,0.5)

anchorPoint 还有两个极端值，一个是锚点在节点对象矩形右上角，如果图 3-20 所示，此时 anchorPoint 为(1,1)；另一个是锚点在节点对象矩形左下角，如果图 3-21 所示，此时 anchorPoint 为(0,0)。

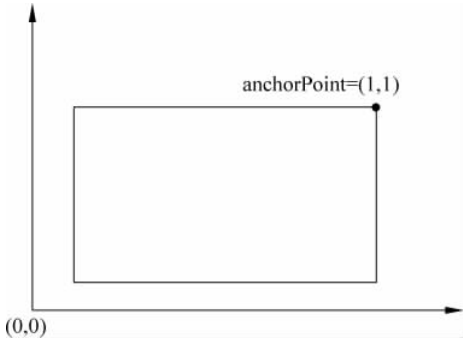

图 3-20　anchorPoint 为(1,1)

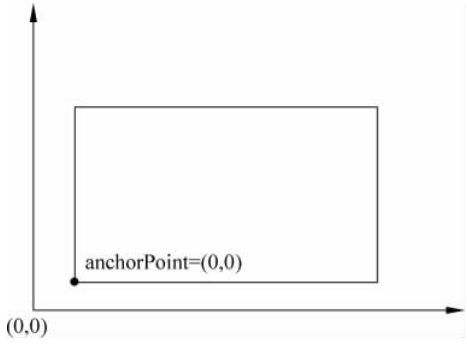

图 3-21　anchorPoint 为(0, 0)

为了进一步了解 anchorPoint 的使用，我们修改 HelloWorld 实例，修改 HelloWorldScene.cpp 的 HelloWorld::init()函数如下，其中加粗字体显示的是我们添加的代码。

```
bool HelloWorld::init()
{
    …

    auto label = Label::createWithTTF("Hello World", "fonts/Marker Felt.ttf", 24);
    label->setPosition(Point(origin.x + visibleSize.width/2,
                             origin.y + visibleSize.height - label->getContentSize().
height));

    label->setAnchorPoint( Vec2(1.0, 1.0));

    this->addChild(label, 1);

    auto sprite = Sprite::create("HelloWorld.png");
    sprite->setPosition(Vec2(visibleSize.width/2 + origin.x, visibleSize.height/2 +
origin.y));
    this->addChild(sprite, 0);

    return true;
}
```

运行结果如图 3-22 所示，Hello World 设置了 anchorPoint 为(1.0,1.0)。

图 3-22 Hello World 的 anchorPoint 为(1.0,1.0)

3.5.3 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,它由导演对象来管理和维护。如果需要场景中的精灵运动起来,我们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,所以我们也可以直接使用 Node 中的定时器相关函数。

Node 中的定时器相关函数主要有:

- void scheduleUpdate(void):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一次自己的 update(float dt)函数。
- void schedule(SEL_SCHEDULE selector, float interval):与 scheduleUpdate 函数功能一样,不同的是我们可以指定回调函数(通过 selector 指定),也可以更加需要指定回调时间间隔。
- void unscheduleUpdate(void):停止 update(float dt)函数调度。
- void unschedule(SEL_SCHEDULE selector):可以指定具体函数停止调度。
- void unscheduleAllSelectors(void):可以停止调度。

为了进一步了解游戏循环与调度的使用,我们修改 HelloWorld 实例。

修改 HelloWorldScene.h 代码,添加 update(float dt)声明,代码如下:

```
class HelloWorld : public cocos2d::Layer
{
public:
    …

    virtual void update(float dt);

    CREATE_FUNC(HelloWorld);

};
```

修改 HelloWorldScene.cpp 代码如下：

```
bool HelloWorld::init()
{
    …

    auto label = LabelTTF::create("Hello World", "Arial", 24);
    label->setTag(123);                                                    ①
    …

    //更新函数
    this->scheduleUpdate();                                                ②
    //this->schedule(schedule_selector(HelloWorld::update),1.0f/60);       ③

    return true;
}

void HelloWorld::update(float dt)                                          ④
{
    auto label = this->getChildByTag(123);                                 ⑤
    label->setPosition(label->getPosition() + Vec2(2,-2));                 ⑥
}

void HelloWorld::menuCloseCallback(Ref* pSender)
{
  //停止更新
    unscheduleUpdate();                                                    ⑦
    Director::getInstance()->end();
  …
}
```

为了能够在 init 函数之外访问标签对象 label，我们需要为标签对象设置 Tag 属性，其中第①行代码就是设置 Tag 属性为 123。第⑤行代码是通过 Tag 属性重新获得这个标签对象。

为了能够开始调度，还需要在 init 函数中调用 scheduleUpdate（见第②行代码）或 schedule（见第③行代码）。

代码第④行的 HelloWorld::update(float dt)函数是调度函数，精灵等对象的变化逻辑都是在这个函数中编写的。这个例子很简单，只是让标签对象动起来，第⑥行代码就是改变它的位置。

为了省电等目的，如果不再使用调度，一定不要忘记停止调度。第⑦行代码 unscheduleUpdate()就是停止调度 update，如果是其他的调度函数可以采用 unschedule 或 unscheduleAllSelectors 停止。

3.6 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，我们在 Android 和 iOS 等平台应用开发的时候使用的二维坐标系的原点是在左上角的。而在 Cocos2d-x 坐标系中的原点是在左下角，而且 Cocos2d-x 坐标系又可以分为：世界坐标和模型坐标。

3.6.1 UI 坐标

UI 坐标就是 Android 和 iOS 等应用开发的时候使用的二维坐标系。它的原点是在左上角的(见图 3-23)。

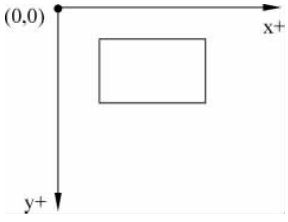

图 3-23 UI 坐标

UI 坐标原点是在左上角，x 轴向右为正，y 轴向下为正。我们在 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 Cocos2d-x 默认不是采用 UI 坐标，但是有的时候也会用到 UI 坐标，例如在触摸事件发生的时候，我们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，如下面的代码所示：

```
Vec2 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 默认坐标，我们可以采用下面的语句进行转换：

```
Vec2 touchLocation2 = 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 默认坐标。

3.6.2 OpenGL 坐标

我们在前面提到了 OpenGL 坐标，OpenGL 坐标是种三维坐标。由于 Cocos2d-x 底层采用 OpenGL 渲染，因此默认坐标就是 OpenGL 坐标，只不过采用两维(x 和 y 轴)。如果不考虑 z 轴，OpenGL 坐标的原点在左下角(见图 3-24)。

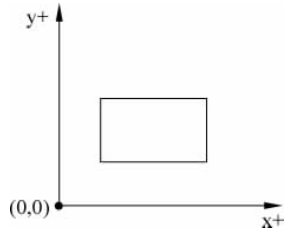

图 3-24 OpenGL 坐标

提示 三维坐标根据 z 轴的指向不同可分为：左手坐标和右手坐标。右手坐标是 z 轴指向屏幕外，如图 3-25(a)所示。左手坐标是 z 轴指向屏幕里，如图 3-25(b)所示。OpenGL 坐标是右手坐标，而微软平台的 Direct3D[①] 是左手坐标。

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，目前为各家的显卡支持。它与 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。——引自于维基百科 http://zh.wikipedia.org/wiki/Direct3D

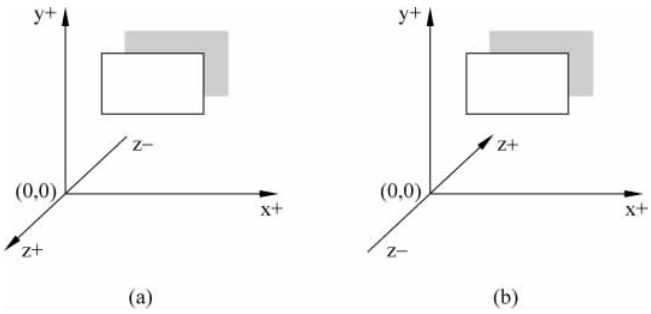

图 3-25　三维坐标

3.6.3　世界坐标和模型坐标

由于 OpenGL 坐标又可以分为世界坐标和模型坐标，所以 Cocos2d-x 的坐标也有世界坐标和模型坐标。

你是否有过这样的问路经历：张三会告诉你向南走 1000 米，再向东走 500 米，即到目的地。而李四会告诉你向右走 1000 米，再向左走 500 米，即到目的地。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，他把地球作为参照物，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，他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。

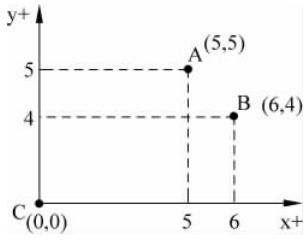

图 3-26　世界坐标和模型坐标

我们看看图 3-26，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)。

有的时候，我们需要将世界坐标与模型坐标互相转换。我们可以通过 Node 对象函数实现：

- Vec2 convertToNodeSpace(const Vec2& worldPoint)：将世界坐标转换为模型坐标。
- Vec2 convertToNodeSpaceAR(const Vec2& worldPoint)：将世界坐标转换为模型坐标，AR 表示相对于锚点。
- Vec2 convertTouchToNodeSpace(Touch * touch)：将世界坐标中触摸点转换为模型坐标。
- Vec2 convertTouchToNodeSpaceAR (Touch * 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。
- Vec2 convertToWorldSpace (const Vec2& nodePoint)：将模型坐标转换为世界坐标。

- Vec2 convertToWorldSpaceAR (const Vec2& nodePoint)：将模型坐标转换为世界坐标，AR 表示相对于锚点。

下面通过两个例子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 3-27 是世界坐标转换为模型坐标实例运行结果。

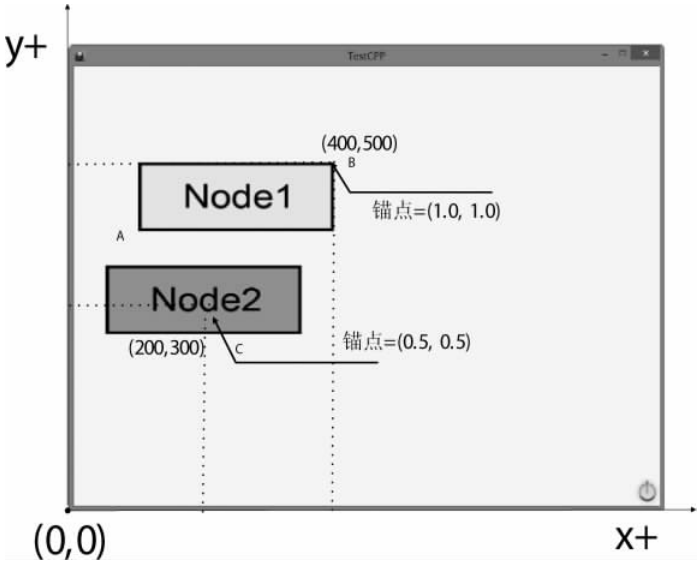

图 3-27 世界坐标转换为模型坐标

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：

```
bool HelloWorld::init()
{

    if ( !Layer::init() )
    {
        return false;
    }

    …
    //创建背景
    auto bg = LayerColor::create(Color4B(255, 255, 255, 255));                    ①
    this->addChild(bg, 0);                                                        ②

    //创建 Node1
```

```
    auto node1 = Sprite::create("node1.png");                                    ③
    node1->setPosition(Vec2(400,500));
    node1->setAnchorPoint(Vec2(1.0, 1.0));
    this->addChild(node1, 0);                                                    ④

    //创建 Node2
    auto node2 = Sprite::create("node2.png");                                    ⑤
    node2->setPosition(Vec2(200,300));
    node2->setAnchorPoint(Vec2(0.5, 0.5));
    this->addChild(node2, 0);                                                    ⑥

    Vec2point1 = node1->convertToNodeSpace(node2->getPosition());
                                                                                 ⑦
    Vec2point3 = node1->convertToNodeSpaceAR(node2->getPosition());              ⑧

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

    return true;
}
```

代码第①～②行是创建背景颜色层对象，它是一个白色的 900×640 大小的层，其中 Color4B(255，255，255，255)是创建一个白颜色对象，四个参数分别表示颜色的 RGBA 值。代码第③～④行是创建 Node1 对象，并设置了位置和锚点属性。代码第⑤～⑥行是创建 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 3-27，我们解释一下：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 3-27 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 3-28 是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，大小是 150×50 像素，Node2 相对于 Node1 的模型坐标是(0，0)。

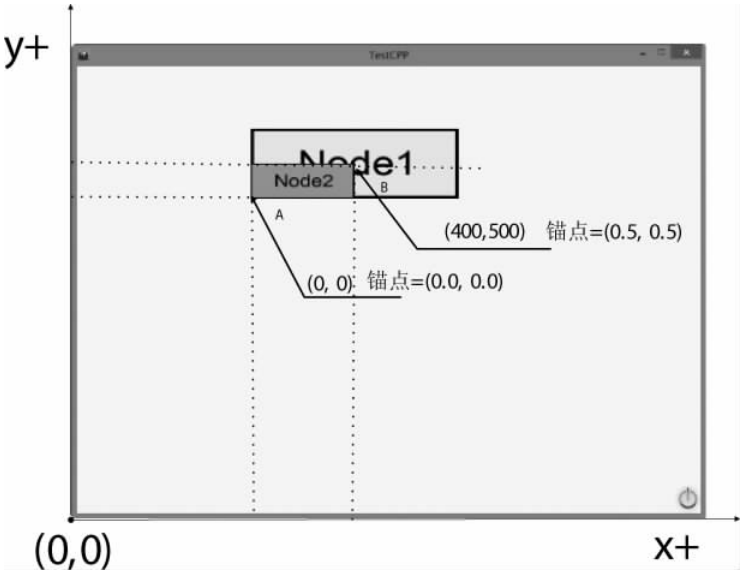

图 3-28 模型坐标转换为世界坐标

编写代码如下：

```
bool HelloWorld::init()
{
    if ( !Layer::init() )
    {
        return false;
    }

    …

    //创建背景
    auto bg = LayerColor::create(Color4B(255, 255, 255, 255));
    this->addChild(bg, 0);

    //创建 Node1
    auto node1 = Sprite::create("node1.png");
    node1->setPosition(Vec2(400,500));
    this->addChild(node1, 0);

    //创建 Node2
    auto node2 = Sprite::create("node2.png");
    node2->setPosition(Vec2(0.0, 0.0));                    ①
    node2->setAnchorPoint(Vec2(0.0, 0.0));                 ②
    node1->addChild(node2, 0);                             ③
```

```
    Vec2point2 = node1->convertToWorldSpace(node2->getPosition());                    ④
    Vec2point4 = node1->convertToWorldSpaceAR(node2->getPosition());                  ⑤


    log("Node2 WorldSpace = (%f,%f)",point2.x,point2.y);
    log("Node2 WorldSpaceAR = (%f,%f)",point4.x,point4.y);

    return true;
}
```

上述代码中第③行是将 Node2 放到 Node1 中，这是与之前的代码的区别。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。

代码第④行将 Node2 的模型坐标转换为世界坐标。而代码第⑤行是类似的，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)
```

图 3-26 所示的位置，可以用世界坐标描述。代码第①～③行修改如下：

```
node2->setPosition(Vec2(250, 450));
node2->setAnchorPoint(Vec2(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 家谱、Cocos2d-x 设计目标，然后从一个 HelloWorld 游戏开始，学习了 Cocos2d-x 的核心概念，这些概念包括：导演、场景、层、精灵和菜单等节点对象。此外，我们还重点学习了 Node 和 Node 层级架构。