Cocos2d-x 高级开发教程

- **前言**
- Cocos2d-x 是一个通用平面游戏引擎，基于一个同样十分著名的游戏引擎 Cocos2d-iPhone 设计。它继承了 Cocos2d 系列引擎一贯的特点：使用简单，运行高效、灵活，且功能强大。
- 与 Cocos2d-iPhone 不同的是，Cocos2d-x 还拥有强大的跨平台能力，只需要编写一次代码，就可以无缝地部署在包括 iOS、Android、Windows、OS X 在内的许多主流游戏平台之上。在移动终端日趋多样化的今天，把游戏部署到多种平台是游戏开发的大趋势，Cocos2d-x 的跨平台能力无疑为开发者节省了大量的时间和精力。
- 本书的主要目的是向读者介绍 Cocos2d-x 这个十分优秀的平面游戏引擎。阅读完本书前三部分之后，读者会对 Cocos2d-x 的各个方面都有比较深入的了解，并且也会对游戏开发的过程以及技巧有了一定的认识，可以得心应手地使用 Cocos2d-x 进行游戏开发了。
- 同时，本书第四部分介绍了一些最新的游戏开发技术，包括多平台开发、可视化开发、游戏移植，以及 Cocos2d-HTML5。它们作为 Cocos2d-x 的补充，为游戏开发者带来了极大的便利。阅读完这一部分之后，读者将对游戏开发的趋势有一个新的认识。
- **示例代码**
- 本书采用了时下最热门的游戏《捕鱼达人》作为游戏示例。《捕鱼达人》由北京触控科技有限公司开发，采用 Cocos2d-x 作为游戏引擎。在本书中，我们以开发自己的《捕鱼达人》作为主线，一边讲解游戏开发技术，一边不断地完善捕鱼游戏，并最终引导读者实现属于自己的《捕鱼达人》。
- 为了使读者可以完整地体验一次游戏开发流程，本书还在第五部分安排了实战演练章节，带领读者体验真正的游戏开发：从创建空项目开始，然后进行首轮开发与多次迭代，最终得到可以运行在移动设备上的游戏成品。
- 本书示例游戏中包含的所有资源文件，都源自触控科技有限公司授权使用的《捕鱼达人》原版资源文件。读者可以从图灵社区（www.ituring.com.cn）本书主页免费获取实战章节中的代码，以及所使用的资源文件。
- **读者背景**
- 阅读本书需要具备一定的计算机知识以及编程功底。Cocos2d-x 采用 C++编写，熟练掌握 C++语言对于学习本书是很有必要的。此外，本书第四部分涉及了游戏多平台的话题，在阅读这一部分时，根据读者不同的需求，也许还需要了解 JavaScript、Objective-C 以及 C#的知识。
- 本书是针对 Cocos2d-x 游戏开发者撰写的，无论是初学者、有一定经验的开发者，还是对引擎内部工作原理有兴趣的读者，都十分适合阅读本书。
- 对于初学者：本书前两部分从游戏开发的基础知识开始，详细介绍了游戏开发的方法以及所使用的工具，初学者可以轻松入门。
- 对于从事过游戏开发，包括使用过其他 Cocos2d 引擎的读者：本书利用《捕鱼达人》作为示例，完整展示了游戏从开发到部署的各个环节。书中不仅介绍了 Cocos2d-x 中常用的功能与技巧，还探讨了许多配合 Cocos2d-x 所使用的高级技术，涵盖游戏效率优化、网络通信和游戏移植等时下热门的话题。
- 对于好奇引擎工作原理的读者：本书第一部分与第二部分穿插剖析了 Cocos2d-x 的核心代码，为读者理解引擎的工作原理提供了清晰的思路。配合本书来研究 Cocos2d-x 的代码是一个很好的选择。
- **本书分工与致谢**

- 本书的主要作者为尹航，整体设计及撰写思路由马朔负责，第 1 章至第 7 章、第 8 章、第 10 章、第 17 章至第 20 章，以及附录 A 由尹航编写，第 9 章、第 11 章至第 16 章由丁伟杰编写，第 21 章由张三华编写。

- 在此，首先要感谢北京触控科技有限公司以及 Cocos2d-x 开发团队对本书提供的多方面的帮助，他们共同为本书提供了最权威的技术支持。感谢中山大学在团队发展过程中所给予的大力支持。同样感谢马朔带领下的火烈鸟网络所有成员对本书的贡献，他们是叶思聪、彭颖辉、潘阳和郑浩等，没有他们的努力，就不会有这本书。本书的插图由尹航的女友赵婉滢提供，感谢她提供的精美插图以及在本书编写过程中的支持与耐心。最后，感谢所有为本书提出评论、建议以及支持的朋友：周顺帆、小小、Hana 以及其他的朋友们，他们的建议使得本书更加完善。

- 由于笔者水平有限，书中内容难免会有错误或疏漏，欢迎读者批评指正。

- **目录（1）**

- 第一部分　引擎基础

■ **目录（2）**

序

- Cocos2d-x 开源项目诞生于 2010 年 7 月，至今已经走过了两年半的发展历程，在这段时间内，手机游戏整个行业快速发展，从 2010 年的 Triniti、Hapi Kingdom 海外 iOS 收入突破 50 万美金为起点，到 2011 年国内 iOS 收入突破 150 万美金，2012 年国内 Android 市场出现 10 款月收入过千万人民币的游戏，到现在 2013 年 2 月国内 3 款手游月收入突破 3000 万。在这股浪潮中，无数年轻人加入移动游戏行业的淘金，Cocos2d-x 有幸从技术层面支撑了其中多数开发者。使用 Cocos2d-x 引擎的游戏，不仅有《捕鱼达人》、《我叫 MT》和《龙之力量》这样的行业标杆，也有最近一鸣惊人的个人开发者《找你妹》，所以这个行业仍然充满着机会和想象力。

- 很多朋友在学习和使用 Cocos2d-x 引擎的过程中，总抱怨文档不足，缺乏中文文档。对此，我也只能说非常抱歉，我们只有几个全职的开发者，设计实现新功能、修复 bug、论坛扫贴已经占用了大多数时间，最后只能留一些英文文档，而翻译成中、日、韩、德、西和法各种语言，都交由开源社区的贡献者来完成了。今年，Cocos2d-x 社区将会有 7 本中文书、两本日文书、两本韩文书和两本英文书出版。在这些书里面，我着重推荐一下火烈鸟网络的这本书，也就是你现在看到的这本。

- 之前我读过的多数图书手稿，首先是侧重于 Cocos2d-x 的基础概念讲解，缺乏对进阶内容（如 OpenGL ES Shader 使用、网络和多线程并发等）的讲解，其次是作者本身缺乏大型手游项目（30 万行以上）的经验。而火烈鸟的这几个程序员：
- 作为技术承包方参与过触控《捕鱼达人》1 代和 2 代产品的开发，为《捕鱼达人》移植了 Windows Phone 7 和 Windows Phone 8 版本；
- 支援过《我叫 MT》安卓版的开发工作；
- 参与了 Cocos2d-XNA for Windows Phone 7 的移植；
- 为 Cocos2d-x 代码仓库贡献了大量 Lua 绑定的单元测试用例，其中 TestLua 目前的基础框架就是他们贡献的。
- 因此，在这本书里，我们不仅可以看到大量进阶内容，还不时可以看到触屏事件的派发有坑时该如何绕过去，内存引用计数用着不爽时如何用 boost 智能指针来替换等有深度的吐槽。而对于社区里频繁问的"我的游戏是用 Cocos2d-iPhone 做的，如何移植到 Cocos2d-x 上"，这本书则有专门一个章节来讲解其中的技术策略、各种策略的优劣以及工作分解，其中有些方法居然是我之前不知道的。在最后的练习章节中，他们还竟然获得了《捕鱼达人》的授权，传授大家如何做一个捕鱼的原型出来练练手，月入千万的捕鱼就这样被贴出源码和图片，换成我的话可舍不得。
- 希望火烈鸟网络的这本书能对 Cocos2d-x 广大的中文开发者有所帮助，也祝大家在移动游戏领域里能淘到真金。

■ 引擎简介

- Hello Cocos2d-x

- 在这一章中，我们将与所有程序设计入门书一样，从 Hello World 这个最简单的例子开始介绍 Cocos2d-x 引擎，我们将让读者看到，利用 Cocos2d-x 制作一个游戏是一件多么轻松的事情。下面我们首先介绍 Cocos2d-x 的概况，然后引导读者一步一步建立 Windows 下的开发环境，并简单了解 Cocos2d-x 程序的基本结构。

■ 引擎简介

- Cocos2d-x 的原型是 Cocos2d，一个最早来源于几位 Python 开发者在 PyWeek 竞赛中的作品，目的是封装底层绘图代码，简化 2D 游戏的开发过程，避免每次都"重新发明轮子"。有了 Cocos2d，开发者就可以把全部精力集中在游戏开发上，而不必关心绘图的细节。这个 Python 版本的引擎最早发布于 2008 年 4 月，并一直保持版本的更新。
- 在 Cocos2d 发布的同年 3 月，苹果发布了 iOS 的 SDK，允许第三方开发者为 iOS 设备开发各种应用，其中游戏应用是最热门的。同样是为了简化游戏开发的难度，把大家从复杂的 OpenGL 编程中解放出来，Ricardo Quesada 将 Cocos2d 从 Python 移植到了 iOS 上，并使用 iOS 的原生语言 Objective-C 重写了游戏引擎。2008 年 11 月，Cocos2d-iPhone 引擎正式开源，发布了 0.1 版，为 2D 游戏开发提供了一个便利的方式。伴随着 iOS 开发的大潮，Cocos2d-iPhone 引擎逐步完善，这得益于社区内世界各地开发者的贡献，并且其发展速度已经大大超过了 Python 版本。由于 Cocos2d 的易用性，使用 Cocos2d-iPhone 开发的游戏数量日益增多，至今已经有 2500 多个游戏在 App Store 上架。
- 后来，Android 的兴起大大扩展了移动终端的游戏市场，也由此产生了一个迫切的需求，即游戏开发者需要将游戏同时部署在两个热门平台上，但是为一个游戏维护两套代码的代价是非常昂贵的，尤其是在两个平台的主流语言、主流引擎还不统一的情况下。因此，基于 C++语言、以 Cocos2d-iPhone 为基础开发的 Cocos2d-x 出现了，它凭借其从语言到接口的跨平台特性，受到了跨平台游戏开发者的广泛关注。使用 Cocos2d-x 开发的游戏已经可以做到"代码通用"，只需要经过少量调整，就可以移植到包括 iOS、Android、Windows、Linux 在内的诸多平台上。
- 从本质上说，Cocos2d 是一个图形引擎，封装了复杂的图形接口，通过抽象出精灵、动作等概念，降低了游戏开发难度，简化了开发过程。Cocos2d-x 为保证游戏能方便地移植到不同平台上，又在此基础上做了很多扩展，包括一套 Objective-C 风格的基础类系、平台无关的多点触摸协议、重力感应和音频系统等。
- 接下来，我们将简单介绍一下 Cocos2d 的特性。
- 流程控制（flow control）：非常容易管理不同场景（scene）之间的流程控制。

- 精灵（sprite）：快速而方便的精灵用于显示一切可见的元素。
- 节点（node）：基于树结构的分层组织方式，方便管理不同层次的游戏元素，同时提供了统一管理的计时器（scheduler）。
- 动作（action）：应用于精灵或其他游戏元素的动画效果，可以组合成复杂的动作，如移动（move）、旋转（rotate）和缩放（scale）等。
- 特效（effect）：包括波浪（wave）、旋转（twirl）和透镜（lens）等视觉特效。
- 平面地图（tiled map）：支持矩形和六边形的平面地图。
- 菜单（menu）：创建游戏中常用的菜单。
- 用户输入：提供接受用户触摸事件、传感器（如加速度计）等输入的简单解决方案。
- 文档（document）：编程指南、API 参考、视频教学以及很多简单可靠的测试样例。
- MIT 许可：免费开放的协议，但是请谨记尊重版权。
- 基于 OpenGL：深度优化的绘图方式，支持硬件加速。

■ 搭建开发环境

- **搭建开发环境**
- 使用 Cocos2d-x 的一个很大便利之处在于，我们可以在 PC 或 Mac 环境下完成编码和大部分的调试，然后再迁移到其他设备上做实际环境测试。这意味着我们可以在 PC 上开发游戏，然后以极其低廉的成本把游戏迁移到 iOS 或其他平台上，从而节省了开发阶段支付在设备方面的许多费用。
- 前面提到，Cocos2d-x 可以部署在多种平台上，具体的执行步骤并不复杂。关于如何在不同平台下部署模拟器或设备以及调试方法，感兴趣的读者可以阅读附录 A。
- 现在，我们遵循下面的步骤在 PC 上搭建开发环境。
- 正确安装 Visual Studio 2010（后简称 VS）。
- 从官方网站的下载页面（http://www.cocos2d-x.org/projects/cocos2d-x/wiki/Download）下载最新的 Cocos2d-x 源码并解压，这里我们使用的是 C++分支的 2.0 版本，如图 1-1 所示。
- 运行解压目录下的"install-templates-msvc.bat"文件，Cocos2d-x 应用程序向导就会自动安装到 VS 环境下。

图 1-1 下载最新的 Cocos2d-x 源码

- 如果看到类似图 1-2 所示的文字，说明程序已经安装成功了。此时再打开 VS 的"新建项目"对话框，就可以看到该对话框中出现了 Cocos2d-x 项目模板。需要注意的是，默认情况下新建项目的存放位置应该设置在 Cocos2d-x 引擎的安装目录下，否则可能因为找不到库文件而不能通过编译（此时需要在项目编译选项中设置头文件和库的搜索路径）。

图 1-2 命令行窗口

-

■ Hello World

■ Hello World

■ 完成开发环境的搭建后，下面我们就来创建第一个 Cocos2d-x 项目，具体操作步骤如下所示。

- 打开 VS，新建一个 Cocos2d-x 工程，取名为"FishingJoy"，如图 1-3 所示。

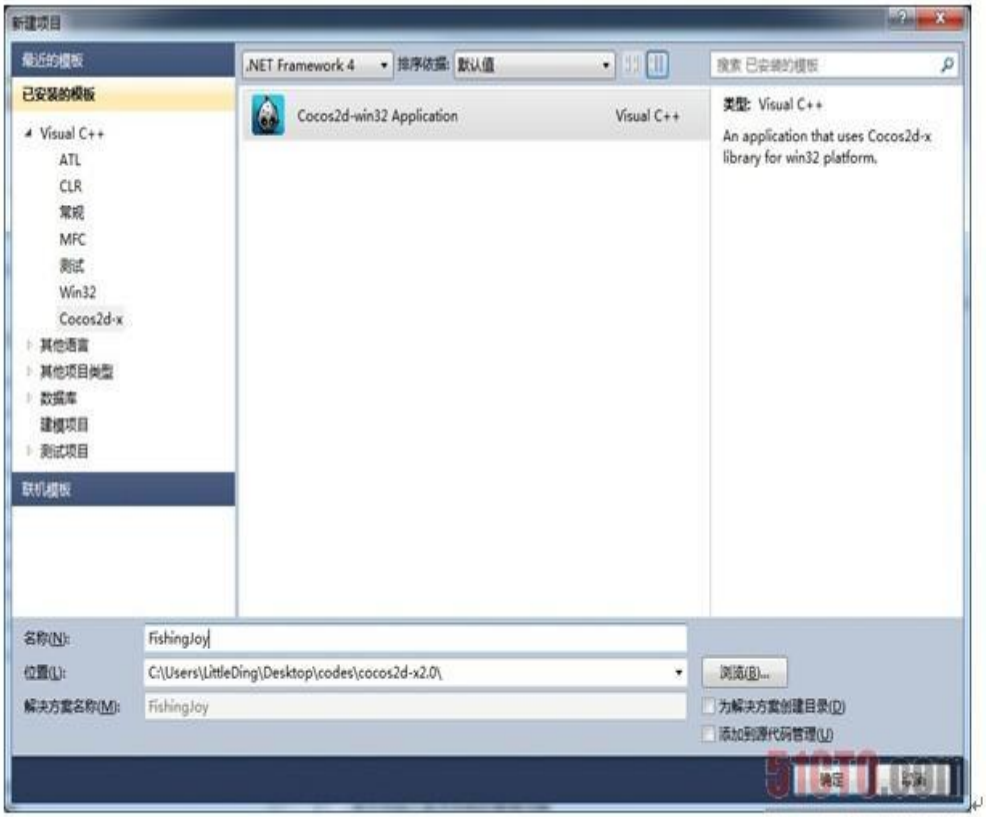


图 1-3 “新建项目”对话框

- 创建工程后，在 VS 的"解决方案资源管理器"（如图 1-4 所示）中能够看到一个典型的 Cocos2d-x 工程的文件目录结构，其中"include"与"source"文件夹中存放游戏代码，"resource"文件夹中存放游戏资源，"外部依赖项"文件夹中存放 Cocos2d-x 引擎与其他依赖项目的源码。通常情况下，Cocos2d-x 的工程已经包含了必要的外部依赖库，因此我们只需要修改游戏代码与资源。

- 设置启动项目。从"解决方案资源管理器"中找到 FishingJoy.win32 项目并将其设置为启动项目，具体操作如图 1-5 所示。
- 不做其他任何修改，直接运行项目。第一次编译可能会需要较长时间，等待编译完成后，启动调试。如果看到如图 1-6 所示的 Cocos2d-x 标志，那么恭喜你，你的第一个 Cocos2d-x 游戏运行成功了。

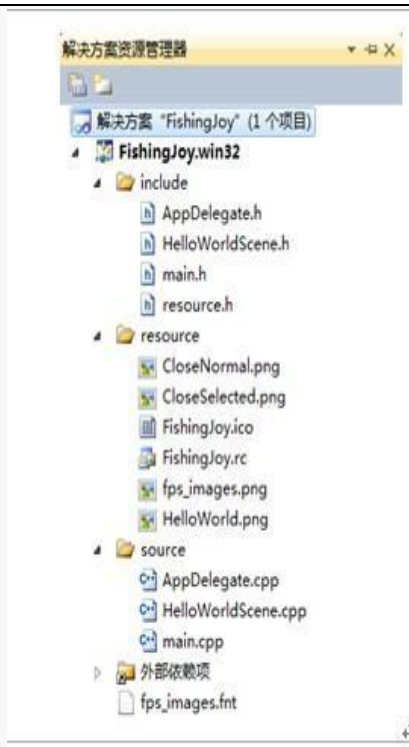


图 1-4 解决方案资源管理器

图 1-5 设置启动项目

图 1-6 第一个 Cocos2d-x 项目 Hello World

■ Hello World 分析（1）

■ Hello World 分析（1）

■ 打开新建的"FishingJoy"项目，可以看到项目文件是由多个代码文件及文件夹组成的，其中 Hello World 的代码文件直接存放于该项目文件夹中。下面我们来详细介绍一下项目的文件组成。

■ 1."resource"文件夹

- 该文件夹主要用于存放游戏中需要的图片、音频和配置等资源文件。为了方便管理，可以在其中创建子文件夹。在不同平台下，对于文件路径的定义是不一致的，但实际接口大同小异。Cocos2d-x 为我们屏蔽了这些差异，其中"resource"文件夹可以默认为游戏运行时的目录。
- 还记得上一节我们运行起来的游戏吗？游戏中显示的 Cocos2d-x 标志就放在这个文件夹下面。除此之外，这个文件夹还保存了游戏左下角 FPS 的字体以及退出游戏按钮上的图片。
- **2."include"和"source"文件夹**
- 这两个文件夹用于放置游戏头文件和源代码文件。可以看到，项目模板为我们添加的三个文件分别为"main.h"、"main.cpp"和"resource.h"，它们是平台相关的程序文件，为 Windows 专有。通常情况下，程序入口与资源文件管理在不同平台下是不同的，但是 Cocos2d-x 的模板已经基本为我们处理好了这些细节，不需要对它们进行修改。
- **3."AppDelegate.h"和"AppDelegate.cpp"文件**
- 这两个文件是 Cocos2d-x 游戏的通用入口文件，类似于一般 Windows 工程中主函数所在的文件。接触过 iOS 开发的读者应该会觉得这两个文件的名字似曾相识，其实 AppDelegate 在 iOS 工程中就是程序的入口文件，在介绍引擎历史的时候曾提到过。
- Cocos2d-x 来源于 Cocos2d-iPhone，因此无论是代码风格还是文件结构，很多方面都沿袭了 Cocos2d-iPhone 的使用习惯。在第 2 章中，我们将详细介绍 Cocos2d-x 的代码风格与文件结构。
- 打开"AppDelegate.cpp"，我们可以看到已经自动添加的代码，这个文件实现了 AppDelegate 类。AppDelegate 控制着游戏的生命周期，除去构造函数和析构函数外，共有 3 个方法，下面我们将逐个介绍。
- bool applicationDidFinishLaunching()。应用程序启动后将调用这个方法。默认的实现中已经包含了游戏启动后的必要准备：

```
//初始化游戏引擎控制器 CCDirector，以便启动游戏引擎
CCDirector *pDirector = CCDirector::sharedDirector();
pDirector->setOpenGLView(&CCEGLView::sharedOpenGLView());

//下面这条被注释掉的语句用于开启高分辨率屏幕（例如 iOS 系列设备的 Retina 屏幕）支持。
//如果需要启用，可以去掉注释
//pDirector->enableRetinaDisplay(true);

//启用 FPS 显示
pDirector->setDisplayStats(true);

//设置绘制间隔
pDirector->setAnimationInterval(1.0 / 60);

CCScene *pScene = HelloWorld::scene();
pDirector->runWithScene(pScene);

return true;
```

- 这段代码首先对引擎进行必要的初始化，然后开启了 FPS 显示。FPS 即每秒帧速率，也就是屏幕每秒重绘的次数。启用了 FPS 显示后，当前 FPS 会在游戏的左下角显示。通常在游戏开发阶段，我们会启用 FPS 显示，这样就可以方便地确定游戏运行是否流畅。

- 接下来是设置绘制间隔。绘制间隔指的是两次绘制的时间间隔，因此绘制间隔的倒数就是 FPS 上限。对于移动设备来说，我们通常都会将 FPS 限制在一个适当的范围内。过低的每秒重绘次数会使动画显示出卡顿的现象，而提高每秒重绘次数会导致设备运算量大幅增加，造成更高的能耗。人眼的刷新频率约为 60 次每秒，因此把 FPS 限定在 60 是一个较为合理的设置，Cocos2d-x 就把绘制间隔设置为 1/60 秒。至此，我们已经完成了引擎的初始化，接下来我们将启动引擎。

- 最后也是最关键的步骤，那就是创建 Hello World 场景，然后指派 CCDirector 运行这个场景。对于游戏开发者而言，我们需要在此处来对我们的游戏进行其他必要的初始化，例如读取游戏设置、初始化随机数列表等。程序的最末端返回 true，表示程序已经正常初始化。

- void applicationDidEnterBackground()。当应用程序将要进入后台时，会调用这个方法。具体来说，当用户把程序切换到后台，或手机接到电话或短信后程序被系统切换到后台时，会调用这个方法。此时，应该暂停游戏中正在播放的音乐或音效。动作激烈的游戏通常也应该在此时进行暂停操作，以便玩家暂时离开游戏时不会遭受重大损失。

- void applicationWillEnterForeground()。该方法与 applicationDidEnterBackground()成对出现，在应用程序回到前台时被调用。相对地，我们通常在这里继续播放刚才暂停的音乐，显示游戏暂停菜单等。

- "HelloWorldScene.h"与"HelloWorldScene.cpp"文件。这两个文件定义了 Hello World 项目中默认的游戏场景。Cocos2d 的游戏结构可以简单地概括为场景、层、精灵，而这两个文件就是 Hello World 场景的实现文件。每个游戏组件都可以添加到另一个组件中，形成层次关系，例如场景中可以包含多个层，层中可以包含多个精灵。在后续章节中，我们将详细讲解 Cocos2d 游戏元素的概念，此处将不详细说明是如何创建出 Hello World 场景的。

- HelloWorldScene 中定义了一个 HelloWorld 类，该类继承自 CCLayer，因此 HelloWorld 本身是一个层。HelloWorld 类包含一个静态函数和两个实例方法，下面我们来看其中比较重要的两个成员。

■ Hello World 分析（2）

- Hello World 分析（2）

- static CCScene* scene()。在 Cocos2d 中，在层下设置一个创建场景的静态函数是一个常见的技巧。我们为 HelloWorld 层编写了 CCLayer 的一个子类，在子类中为层添加各种精灵或是逻辑处理代码。然而我们的 Hello World 场景十分简单，只包含了一个层，没有任何其他需要处理的问题。因此，我们除了创建 CCScene 的一个子类之外，也可以直接使用静态函数来创建一个空场景，再把层置入场景之中，这样也十分便捷，示例代码如下所示：

```
CCScene *scene = CCScene::create();
HelloWorld *layer = HelloWorld::create();
scene->addChild(layer);
```

- 在这段代码中，首先利用 CCScene::create 方法创建了一个空场景，然后利用 Hello world::create 方法创建一个 HelloWorld 层的实例，最后调用 scene 对象的 addChild 方法来把创建的层添加到场景之中。

- 这是我们第一次见到 addChild 方法，这个方法可以把一个游戏元素放置到另一个元素之中。只有把一个游戏元素放置到其他已经呈现出来的游戏元素中，它才会呈现出来。比如在这个例子中，我们把 HelloWorld 层置入到上面创建的空场景中，而在前面所述的 AddDelegate 中，我们已经让 CCDirector 运行了该场景，因此 HelloWorld 层就会显示在屏幕上了。

- bool init()。初始化 HelloWorld 类，相关代码如下：

```
//(1) 对父类进行初始化
if ( !CCLayer::init() )
{
    return false;
}

```

```
//(2) 创建菜单并添加到层
CCMenuItemImage *pCloseItem = CCMenuItemImage::create(
                                    "CloseNormal.png",
                                    "CloseSelected.png",
                                    this,
                                    menu_selector(HelloWorld::menuCloseCallback) );
pCloseItem->setPosition( ccp(CCDirector::sharedDirector()->getWinSize().width -
    20, 20) );
CCMenu* pMenu = CCMenu::create(pCloseItem, NULL);
pMenu->setPosition( CCPointZero );
this->addChild(pMenu, 1);

//(3) 创建"Hello World"标签并添加到层中
CCLabelTTF* pLabel = CCLabelTTF::create("Hello World", "Arial", 24);
CCSize size = CCDirector::sharedDirector()->getWinSize();
pLabel->setPosition( ccp(size.width / 2, size.height - 50) );
this->addChild(pLabel, 1);

//(4) 创建显示“HelloWorld.png”的精灵并添加到层中
CCSprite* pSprite = CCSprite::create("HelloWorld.png");
pSprite->setPosition( ccp(size.width/2, size.height/2) );
this->addChild(pSprite, 0);

return true;
```

■ 这段代码可以简单地划分为 4 个部分。

- 调用父类的 init 方法来进行最初的初始化。
- 创建关闭程序的菜单并添加到层中。这里，我们遇到了 addChild(CCNode* child,int zOrder)，与之前遇到的 addChild 方法多出来了一个参数 zOrder，该参数指的是 child 的 z 轴顺序，也就是显示的先后顺序，其值越大，表示显示的位置就越靠前。setPosition 方法用来设置游戏元素的位置。关于菜单以及下面提到的文本标签，我们也会在后面的章节中详细介绍。
- 创建一个文本标签并添加到层中，显示内容"Hello World"。
- 用"HelloWorld.png"创建一个精灵并添加到层中。最后程序返回 true，表示初始化成功。

■ 此时读者可能会有疑惑，为什么我们要在一个实例方法中初始化类，而不在构造函数中初始化呢？在 C++中，一般习惯在构造函数中初始化类，然而由于 Cocos2d-x 的来源特殊，所以才没有采用 C++的编程风格。关于编程风格，我们会在第 2 章中详细讨论。

■ 测试样例简介

■ **测试样例简介**

■ Cocos2d-x 为我们提供了十分丰富的测试样例，这些测试样例是在引擎开发过程中为测试引擎的正确性而编写的代码，同时也是演示引擎各个部分如何使用的良好示例。打开 Cocos2d-x 根目录下的 cocos2d-win32.vc2010 解决方案(如图 1-7 所示)，设置其中的 tests 项目为启动项目，成功运行后，就能够看到许多引擎特性的效果演示了（如图 1-8 所示）。这个项目的代码展示了 Cocos2d-x 引擎的很多标准用法，几乎涵盖了引擎的全部功能，非常具有参考价值。

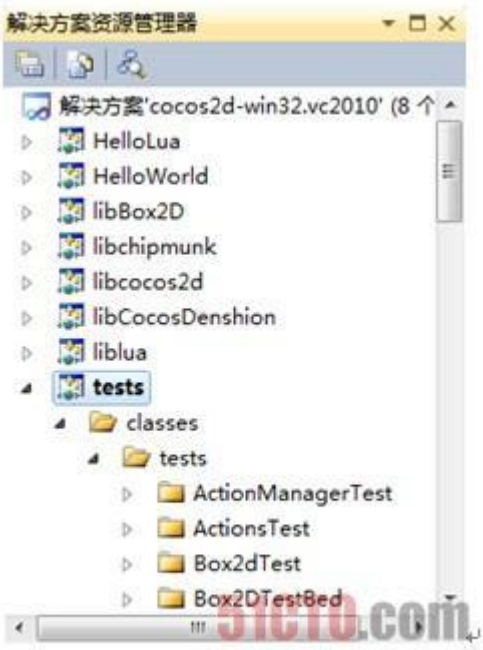


图 1-7 解决方案资源管理器

图 1-8 Cocos2d-x 测试样例工程运行截图

■ 小结

■ **小结**

■ 在这一章中，我们简要介绍了 Cocos2d-x 游戏引擎，并成功运行了第一个基于 Cocos2d-x 的游戏。虽然这个游戏没有任何功能，但是我们已经敲开了 Cocos2d-x 游戏设计的大门。下面总结一下这一章涉及的主要知识点。

■ Cocos2d-x：基于 Cocos2d-iPhone 的多平台二维游戏引擎，为开发者封装了功能强大的绘图代码，使开发者专注于游戏开发而不是绘图操作。

■ AppDelegate：Cocos2d-x 项目中的程序入口文件，提供对程序生命周期的控制事件。

■ 游戏元素：任何可以呈现出来的元素，例如场景、层和精灵。

■ CCNode::addChild 方法：用于将一个游戏元素添加到另一个元素中。在创建一个层或者场景时，通常会初始化自己的游戏元素，定义一些特殊的效果，或是将其他的游戏元素组合到一起，而 addChild 方法就是用于组合游戏元素的。后面我们将会看到，这样的架构是多么简单而又富有表现力。

- 至此，我们已经了解了 Cocos2d-x 游戏设计的一些基本特性，Cocos2d-x 游戏设计的大门已经向我们敞开，后面还有更多、更精彩的特性等待着我们去探索。

2.1 基本概念

- **在游戏开始之前**

- 经过上一章的学习，我们已经可以开发一个最基本的 Cocos2d-x 游戏了，这个游戏包括一张背景图片和一个退出游戏的按钮，但是这距离完成一个完整、实用的游戏还很遥远。在这一章中，我们将首先抛开 Cocos2d，介绍游戏开发的基本概念，然后结合 Cocos2d-x 的特点，介绍一些 Cocos2d-x 开发的基础知识。虽然这一章没有讲解游戏开发的具体方法，会略为枯燥，但介绍的都是 Cocos2d-x 开发所必须理解的基础知识。

- **2.1 基本概念**

- 若要进行游戏开发，首先要理解游戏的基本原理以及元素组成。在这一节中，我们将介绍游戏中的基本概念，包括游戏流程控制、场景、层和精灵等。

2.1.1 场景与流程控制

- **2.1.1 场景与流程控制**

- 我们相信大家都接触过许多不同类型的游戏，而面对大多数游戏，玩家几乎不需要学习就可以开始玩游戏，因为它们都有如下流程：

- 进入游戏，显示游戏主菜单；

- 选择新游戏，开始教学任务或是第一个关卡；

- 选择载入游戏，继续以前的游戏；

- 选择设置，调整游戏的听觉或视觉效果等。

- 以捕鱼游戏为例，如果把它的流程画成图，就可以得到如图 2-1 所示的流程图。通过这个流程图，我们就可以对游戏的流程有一个更清晰的认识。

- 在图 2-1 中，每一个节点中显示的内容相对不变。通常，我们把这些内容相对不变的游戏元素集合称作场景（scene），把游戏在场景之间切换的过程叫做流程控制（flow control）。

- 在 Cocos2d-x 中，场景的实现是 CCScene。

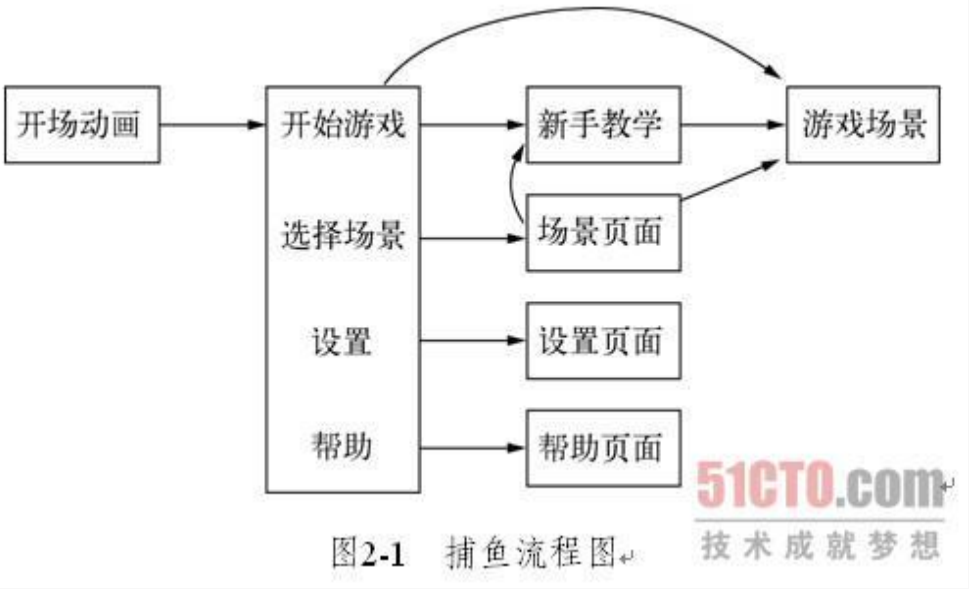


图2-1 捕鱼流程图

- 2.1.2 层

- 2.1.2 层

层是隶属于场景之下的游戏元素。通常，一个复杂场景会拥有多个层，一个层会显示一部分视觉元素，空白部分为透明或半透明，以实现多个层的重叠显示。层与层之间按照顺序叠放在一起，就组成了一个复杂的场景。也许读者接触过图片编辑器（如 Photoshop）或者动画编辑器（如 Adobe Flash?），在这些编辑器中也存在层的概念。在游戏设计中，层的概念与它们类似。

以捕鱼游戏场景为例，场景可以大致分为 4 层。菜单层：悬浮于最上方的各种菜单项。触摸层：处理在屏幕上的触摸点击时间。动作层：放置鱼、子弹、网等，并处理碰撞。背景层：背景图片。

如图 2-2 所示，我们看到的是《捕鱼达人》的主菜单场景和游戏场景的分层示意图。在 Cocos2d-x 中，层的实现是 CCLayer。

图2-2 层

- 2.1.3 精灵

- **2.1.3 精灵**

- 层和场景是其他游戏元素的容器，如果没有向它们添加可见的游戏元素，它们看起来就一直是透明的。精灵则与层或场景不同，它隶属于层，是场景中出现的可见图形。玩家控制的主角、AI 控制的 NPC，以及地图上的宝箱、石块，甚至游戏主菜单的背景图片都是精灵。因此，可以这样认为，玩家看到的一切几乎都是由精灵构成的。

- 精灵不一定是静态的。通常，一个精灵可以不断变化，变化的方式包括：移动、旋转、缩放、变形、显现消失、动画效果（类似 GIF 动画）等。精灵按照层次结构组合起来，并与玩家互动，构成了一个完整的游戏。

- 以《捕鱼达人》的游戏场景为例，我们选择动作层，其中包含的主要精灵如图 2-3 所示。

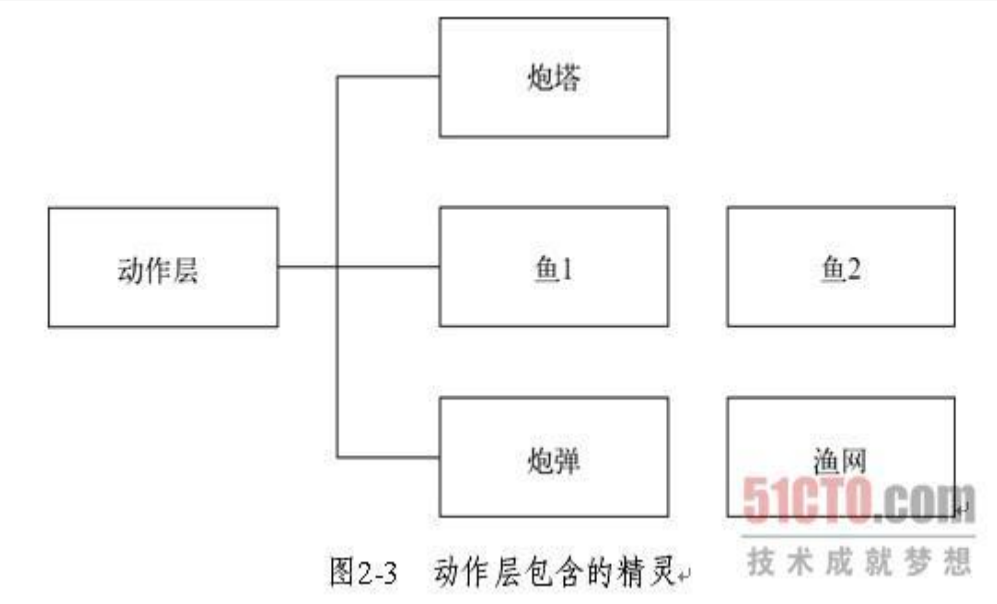


图2-3 动作层包含的精灵

- 在 Cocos2d-x 中，精灵的实现是 CCSprite。

- **2.1.4 节点与渲染树**

- **2.1.4 节点与渲染树**

- 回顾前面的介绍，我们已经知道了精灵、层和场景如何构成一个游戏的框架。精灵属于层，层属于场景，玩家与精灵互动，并导致游戏画面在不同场景中切换。把每个环节拼接在一起，我们得到了一个完整的关系图。以《捕鱼达人》的游戏场景的简化版为例，各个游戏元素按照图 2-4 所示的方式组织在一起。

- 从组织关系的角度来说，游戏元素按照图 2-4 中的树形结构组织起来；而从绘图的角度来说，图形按照自上而下的顺序绘制出来。为了绘制场景，需要绘制场景中的层，为了绘制层，需要绘制层中的精灵。因此，关系图实质上安排了图元的绘图方式，关系图中的每一个元素称作节点（node），关系图则称作渲染树（rendering tree）。渲染场景的过程就是遍历渲染树的过程。

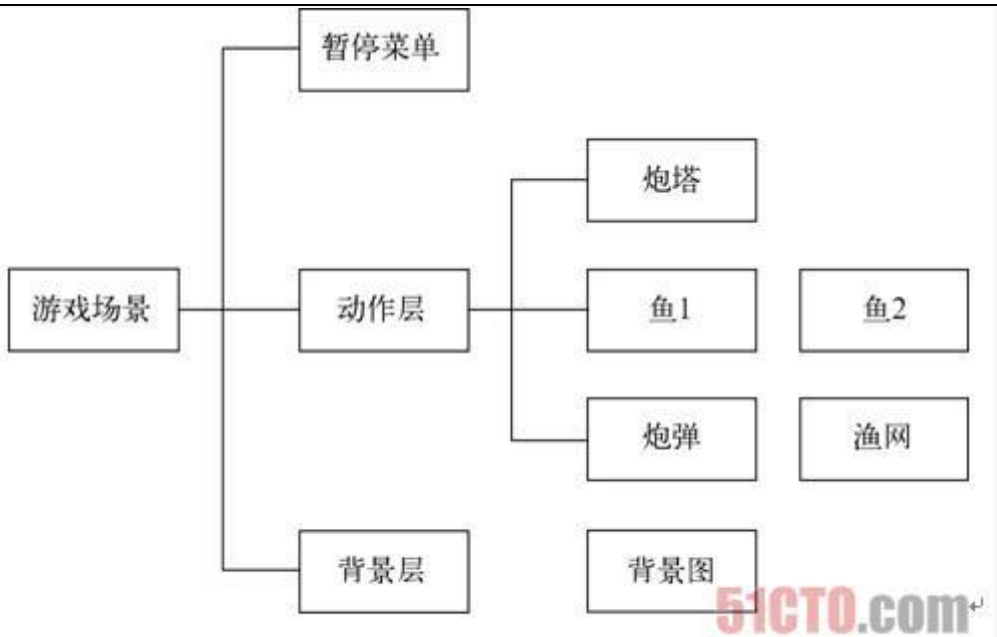


图2-4 游戏元素的组织方式

- 一旦建立起渲染树，组织复杂的场景就变得十分简单。我们赋予每个节点一系列属性，包括节点相对于父节点的位置、旋转角度、缩放比例和变形参数等。渲染树的优势在于，我们只需要考虑节点相对于父节点的属性，就可以逐层创建复杂的对象或动作。
- 另一个简单的例子是《捕鱼达人》中的海龟由躯干和 4 条腿构成（如图 2-5 所示）。在游戏中，不但海龟在水中游动，它的 4 条腿也在不断做划水的动作。这一系列动作可以分解为：4 条腿相对整个海龟在一定角度内旋转；躯干相对于整个海龟静止不动；整个海龟在鱼层中游动，位置和方向在不断改变。
- 因此，建立一个节点表示海龟，在海龟节点下再建立 5 个精灵，分别表示 4 条腿和躯干。这样，每个动作都是可控的，只要为每个节点设置好了动作，就可以完成复杂的动画。反之，如果没有树型结构，组织一个稍微复杂的游动都会成为一个巨大的工程。

图2-5 海龟的构成

- Cocos2d 也采用了渲染树架构。任何可见的游戏元素都派生自 Cocos2d-x 节点(CCNode)，常见的游戏元素有场景(CCScene)、层（CCLayer）和精灵（CCSprite）等。前面提到过，通常游戏按照场景、层、精灵的层次顺序组织，每种节点都有各自的特点。然而在实际开发中，为了实现一些特殊的效果，也不必拘泥于这个层次顺序。层或精灵都是普通的节点，因此，即使向精灵中添加精灵，向场景中添加精灵，甚至向精灵中添加层，这些操作也都没有被禁止。在读者认为必要时，可以自己尝试各种组织层次。

- 2.1.5 动作与动画

■ 2.1.5 动作与动画

■ 动作（action）作用于游戏元素，可以使游戏元素运动起来。常见的动作有移动、转动、闪烁、消失等。动作分为持续性动作与瞬时动作，持续性动作在一段时间内连续完成，瞬时动作会瞬间完成。为了使游戏画面动起来，我们会在需要的时候创建一系列动作，并把它们应用到游戏元素中。在 Cocos2d-x 中，动作由 CCAction 类实现，由 CCAction 类派生出持续性动作类 CCAction Interval 和瞬时动作类 CCActionInstant。所有的动作都派生自以上两个类之一。

■ 动画（animation）是一种特殊的持续性动作，它只能应用于精灵上，用于实现帧动画效果。如同电影胶片一样，一个帧动画由多张静止的图片不停地切换形成。静止的图片叫做帧（frame），帧的序列代表一个动画效果。如图 2-6 所示，《捕鱼达人》中鱼的摆动就是由帧动画组成的，摆动的同时，鱼还在鱼层中游动，显得格外真实。

■

图2-6 帧动画“游动的鱼”

■ 在 Cocos2d-x 中，我们可以使用多个帧创建帧动画序列（CCAnimation），并用帧动画序列创建可作用于精灵的帧动画（CCAnimate）。

- 2.2 Cocos2d-x 代码风格

■ 2.2 Cocos2d-x 代码风格

■ 前面我们已经多次提到 Cocos2d-x 源自于 Cocos2d-iPhone。Cocos2d-iPhone 是一个十分出色的游戏引擎，许多优秀的 iOS 平面游戏都基于 Cocos2d-iPhone 开发，而它的实现语言是 Objective-C。因此，Cocos2d-x 也就沿袭了 Objective-C 的代码风格。这么做的主要原因如下：出于对 Cocos2d-iPhone 程序员习惯的照顾，以及对该引擎的尊敬；方便不同语言下 Cocos2d 游戏的移植；为了实现 Objective-C 风格的内存管理，要求引擎采用特殊的命名规范。

■ 接下来我们将详细介绍 Cocos2d-x 的代码风格。

- 2.2.1 命名空间与类名称

■ 2.2.1 命名空间与类名称

■ Cocos2d-x 拥有一个包含其他所有头文件的文件"cocos2d.h"。通常，我们只需要在使用时包含这个头文件，就可以使用引擎的全部功能了。

■ Cocos2d-x 的类都放置于 cocos2d 命名空间下。以引擎目录下的"actions/CCAction.h"为例，我们可以看到文件的首位有两个宏：NS_CC_Begin 和 NS_CC_END。查看宏的定义可知，这两个宏相当于把所有的类型都包含在了 cocos2d 命名空间下。在游戏中，我们常使用引擎提供的另一个宏 USING_NS_CC 来引用 cocos2d 命名空间：

```
#define USING_NS_CC using namespace cocos2d
```

■ 类的命名与 Cocos2d-iPhone 一致，由类库缩写加上类名称组成，其中类库缩写采用大写，类名称采用驼峰法。Cocos2d 的缩写是 CC，因此 Cocos2d-x 的类都拥有 CC 前缀，例如表示动作的类就叫做 CCAction。

- 2.2.2 构造函数与初始化

■ 2.2.2 构造函数与初始化

- 在 Cocos2d-x 中创建对象的方法与 C++开发者的习惯迥乎不同。在 C++中，我们只需要调用类的构造函数即可创建一个对象，既可直接创建一个栈上的值对象，也可以使用 new 操作符创建一个指针，指向堆上的对象。而在 Cocos2d-x 中，无论是创建对象的类型，还是创建对象的方法都与 C++不同。
- Cocos2d-x 不使用传统的值类型，所有的对象都创建在堆上，然后通过指针引用。创建 Cocos2d-x 对象通常有两种方法：第一种是首先使用 new 操作符创造一个未初始化的对象，然后调用 init 系列方法来初始化；第二种是使用静态的工厂方法直接创建一个对象。下面我们首先介绍第一种方法。
- 在 Objective-C 中并没有构造函数，创建一个对象需要先为对象分配内存，然后调用初始化方法来初始化对象，这个过程就等价于 C++中的构造函数。与 Objective-C 一样，Cocos2d-x 也采用了这个步骤。Cocos2d-x 类的构造函数通常没有参数，创建对象所需的参数通过 init 开头的一系列初始化方法传递给对象。创建对象的步骤如下所示。

- 使用 new 操作符调用构造函数，创建一个没有初始化过的空对象。
- 选择合适的初始化方法，并调用它来初始化对象。

- Cocos2d-x 的初始化方法都以 init 作为前缀，因此可以轻易辨认出来。初始化方法返回一个布尔值，代表是否成功初始化该对象。下面我们提供一个从文件初始化精灵（CCSprite）的例子：

```
CCSprite* sprite1 = new CCSprite();
sprite1->initWithFile("HelloWorld.png");
```

- 在这个例子中，我们首先调用构造函数创建一个未经初始化的 CCSprite 对象，然后在 CCSprite 提供的 8 个初始化方法中选择了从文件创建精灵的初始化方法 CCSprite::init WithFile(const char* filename)来初始化精灵。
- 第二种方法则是使用类自带的工厂方法来创建对象。在 Cocos2d-x 中，许多类会自带一系列工厂方法，这些工厂方法是类提供的静态函数。只要提供必要的参数，就会返回一个完成了初始化的对象。通常 init 系列的初始化方法都会有其对应的工厂方法，它们的名称类似，参数一致，都可以用于创建对象。在 Cocos2d-x 的旧版本中，工厂方法通常以类的名称（不包含前缀）开头，而在 Cocos2d-x 2.0 及后续版本中，工厂方法的名称统一为 create。在名称冲突的情况下，也可能采用以 create 作为前缀的其他函数名。
- 我们仍然以创建精灵为例，下面的两条语句等价，前者为引擎旧版本中的方法，后者为新版本中的方法，它们都会创建一个与第一种方法所述类似的精灵：

```
CCSprite* sprite2 = CCSprite::spriteWithFile("HelloWorld.png");
CCSprite* sprite3 = CCSprite::create("HelloWorld.png");
```

- 这两种方法都可以创建 Cocos2d-x 对象，然而它们在内存管理方面还是有一点点差异的。使用构造函数创建的对象，它的所有权已经属于调用者了，使用工厂方法创建的对象的所有权却并不属于调用者，因此，使用构造函数创建的对象需要调用者负责释放，而使用工厂方法创建的对象则不需要。我们将在 2.3.4 节详细介绍它们的区别。
- 在游戏中，我们需要不断地创建新的游戏元素，通常采取的方法是从 Cocos2d-x 提供的游戏元素类派生出新的类，并在初始化方法中建立好我们所需的游戏元素。这个过程与微软.NET 框架下的 Windows Form 开发类似。例如在 Hello World 中，我们从 CCLayer 类派生出 HelloWorld 类(这是一个层),并重载了 HelloWorld 类的 init()方法,在这个方法中为 HelloWorld 层添加内容。为了保证初始化方法可以被子类重载，需要确保初始化方法声明为虚函数：

```
virtual bool init();
```

- 作为参考，我们提供一个典型的 init()方法框架如下：

```
bool init()
{
    if(CCLayer::init())
```

```
    {
        //在此处写入初始化这个类所需的代码
        return true;
    }
    return false;
}
```

2.2.3 选择器

2.2.3 选择器

在 Objective-C 中，选择器（Selector）是类似于 C++中的类函数指针的机制。由于 Cocos2d-x 继承了 Cocos2d-iPhone 的代码风格，因此也提供了一系列类似于 Objective-C 中创建选择器语法的宏，用来创建函数指针。这些宏都只有一个参数 SELECTOR，表示被指向的类方法。将这些宏列举如下：

```
schedule_selector(SELECTOR)
callfunc_selector(SELECTOR)
callfuncN_selector(SELECTOR)
callfuncND_selector(SELECTOR)
callfunc_selector(SELECTOR)
menu_selector(SELECTOR)
event_selector(SELECTOR)
compare_selector(SELECTOR)
```

下面我们来看第 1 章中的 Hello World 例子。在这个例子中，我们在 HelloWorld 类的 init()方法中添加了一个菜单，当用户点击该菜单时，就会触发此类中的 menuCloseCallback()方法。可以看到，初始化菜单的后两个参数分别是被调用对象与 Cocos2d-x 选择器：

```
CCMenuItemImage *pCloseItem = CCMenuItemImage::create(
    "CloseNormal.png",
    "CloseSelected.png",
    this,
    menu_selector(HelloWorld::menuCloseCallback));
```

2.2.4 属性

2.2.4 属性

C++的类成员只有方法与字段，没有属性和事件，这给开发者带来了不便。为了实现 Objective-C 中提供的属性功能，我们不得不使用方法来模拟 get 和 set 访问器。Cocos2d-x 规定了属性访问器的方法名称以 get 或 set 为前缀，后接属性名。在 CCNode 中包含大量属性，例如用于给节点做标记的 Tag 属性，它的访问器分别为 getTag()和 setTag(int aTag)，其实现原理大致如下：

```
int tag;
int getTag() { return tag; }
void setTag(int aTag) { tag = aTag; }
```

在这个例子中，属性的类型是 int，处理较为简单。由于涉及内存管理，开发中我们对数值类型、结构体类型、Cocos2d-x 对象的处理方法都不尽相同，这一点在读者了解 2.3 节介绍的 Cocos2d-x 内存管理后，会有更深刻的体会。

■ 为每一个属性编写一个或两个访问器方法是一项十分枯燥的任务，为了避免重复性的工作，Cocos2d-x 提供了一系列宏来帮助我们方便地创建属性。表 2-1 列举了所有属性相关的宏，它们定义在引擎目录中的"platform/CCPlatformMacros.h"中。

■ 表 2-1 Cocos2d-x 中与属性相关的宏

● 宏	● 描述
● CC_PROPERTY	● 定义一个属性及其访问器，没有实现。 ● 通常用于简单的值类型
● CC_PROPERTY_READONLY	● 定义一个属性，只包含 get 访问器，没有实现
● CC_PROPERTY_PASS_BY_REF	● 定义一个属性，访问器使用引用类型传 ● 递参数，没有实现。通常用于结构体类型
● CC_PROPERTY_READONLY_PASS_BY_REF	● 定义一个属性，只包含 get 访问器， ● 且使用引用类型传递参数，没有实现

■

（续）

● 宏	● 描述
● CC_PROPERTY_READONLY_PASS_BY_REF	● 定义一个属性，只包含 get 访问器， ● 且使用引用类型传递参数，没有实现
● CC_SYNTHESIZE	● 同 CC_PROPERTY，实现了访问器方法
● CC_SYNTHESIZE_READONLY	● 同 CC_PROPERTY_READONLY，实现了访问器方法
● CC_SYNTHESIZE_PASS_BY_REF	● 同 CC_PROPERTY_PASS_BY_REF， ● 实现了访问器方法
● CC_SYNTHESIZE_READONLY_PASS_BY_REF	● 同 CC_PROPERTY_READONLY_PASS_BY_REF， ● 实现了访问器方法
● CC_SYNTHESIZE_RETAIN	● 同 CC_PROPERTY，实现了访问器方法。 ● 用于派生自 CCObject 的类型，访问器采取 ● Cocos2d-x 的内存管理机制自动维护 ● 对象的引用计数

■ 这些宏只要写在类的定义之中即可。每个宏都有 3 个参数，分别是：varType，属性类型，如果属性类型是对象，需要写成指针的形式；varName，属性的私有字段名称；funName，属性的访问器名称，也就是紧接在 get 或 set 前缀后的部分。

■ 利用 Cocos2d-x 提供的宏，上面的 Tag 属性定义就可以用下面一条语句代替了：

● `CC_SYNTHESIZE(int, tag, Tag)`

■ **2.2.5 单例**

■ 相对于前面的内容，单例（singleton）则是一个很易于理解的概念。在 Cocos2d-x 引擎中，我们能看到大量单例的身影，它们大部分出现在一些系统资源管理类中。单例模式保证了全局有且只有一个实例对象，保证自动地初始化该对象，使得程序在任何时候任何地方都可以访问、获取该对象。

■ 例如，Cocos2d-x 的游戏流程控制器 CCDirector 是一个独一无二的控制器，用于切换游戏场景。换句话说，不可能同时存在两个 CCDirector 实例。

■ 在这种情况下，Cocos2d-x 采用了单例的技巧。用户可以通过类提供的静态方法获取独一无二的实例，而不需要自己来创建。观察 CCDirector 的代码：

```
static CCDisplayLinkDirector s_SharedDirector;
CCDirector* CCDirector::sharedDirector(void)
{
    static bool s_bFirstUseDirector = true;
    if (s_bFirstUseDirector)
    {
        s_bFirstUseDirector = false;
        s_SharedDirector.init();
    }
    return &s_SharedDirector;
}
```

可以发现，CCDirector 维护了一个静态的 CCDirector 实例，在第一次使用前初始化。为了访问 CCDirector 控制器，我们可以使用如下代码：

```
CCDirector::sharedDirector()->replaceScene(newScene);
```

这条语句使用 CCDirector::sharedDirector()获取 CCDirector 的唯一实例，然后调用 replaceScene 来切换到新场景。

2.3.1 复杂的内存管理

2.3 C++中的 Cocos2d-x 内存管理

移动设备上的硬件资源十分有限，内存尤为宝贵，开发者必须十分慎重地利用内存，避免不必要的消耗，更要防止内存泄漏。为了有效地利用内存资源，我们将在这一节中讨论 Cocos2d-x 的内存管理方法。基于 Cocos2d-iPhone 的 Objective-C 风格的内存管理是 Cocos2d-x 的一个特色。把 Objective-C 的内存管理方式引入 C++，使得游戏开发的内存管理难度下降了一个层次。

2.3.1 复杂的内存管理

内存管理一直是一个不易处理的问题，开发者必须考虑分配回收的方式和时机，针对堆和栈做不同的优化处理，等等。内存管理的核心是动态分配的对象必须保证在使用完毕后有效地释放内存，即管理对象的生命周期。由于 C++是一个较为底层的语言，其设计上不包含任何智能管理内存的机制。一个对象在使用完毕后必须被回收，然而在复杂的程序中，对象所有权在不同程序片段间传递或共享，使得确定回收的时机十分困难，因此内存管理成为了程序员十分头疼的问题。

另一方面，过于零散的对象分配回收可能导致堆中的内存碎片化，降低内存的使用效率。因此，我们需要一个合适的机制来缓解这个问题。

Boost 库引入的智能指针（smart pointer）从对象所有权传递的角度来解决内存管理问题。但是，在很多情况下，智能指针还是显得单薄而无力，因为实际开发中对象间的关系十分复杂，所有权传递的操作在开发过程中会变得冗杂不堪。于是，各种基于 C++的第三方工具库和引擎往往都会实现自己的智能内存管理机制来解决内存管理的难题，试图将开发者从烦琐而晦涩的内存管理中解放出来。

2.3.2 现有的智能内存管理技术

2.3.2 现有的智能内存管理技术

目前，主要有两种实现智能管理内存的技术，一是引用计数，一是垃圾回收。引用计数：它是一种很有效的机制，通过给每个对象维护一个引用计数器，记录该对象当前被引用的次数。当对象增加一次引用时，计数器加 1；而对象失去一次引用时，计数器减 1；当引用计数为 0 时，标志着该对象的生命周期结束，自动触发对象的回收释放。引用计数的重要规则是每一个程序片段必须负责任地维护引用计数，在需要维持对象生存的程序段的开始和结束分别增加和减少一次引用计数，这样我们就可以实现十分灵活的智能内存管理了。实际上，这与 new 和 delete 的配对使用十分类似，但是很巧妙地将生成和

回收的事件转换成了使用和使用结束的事件。对于程序员来说，维护引用计数比维护生命周期信息轻松了许多。引用计数解决了对象的生命周期管理问题，但堆碎片化和管理烦琐的问题仍然存在。垃圾回收：它通过引入一种自动的内存回收器，试图将程序员从复杂的内存管理任务中完全解放出来。它会自动跟踪每一个对象的所有引用，以便找到所有正在使用的对象，然后释放其余不再需要的对象。垃圾回收器还可以压缩使用中的内存，以缩小堆所需要的工作空间。垃圾回收可以防止内存泄露，有效地使用可用内存。但是，垃圾回收器通常是作为一个单独的低级别的线程运行的，在不可预知的情况下对内存堆中已经死亡的或者长时间没有使用过的对象进行清除和回收，程序员不能手动指派垃圾回收器回收某个对象。回收机制包括分代复制垃圾回收、标记垃圾回收和增量垃圾回收。

- 2.3.3 Cocos2d-x 的内存管理机制

■ 2.3.3 Cocos2d-x 的内存管理机制

■ Cocos2d-x 很巧妙地运用了前面的引用计数机制。在谈论 Cocos2d-x 代码风格的时候，多次提到 Cocos2d-x 来源于 Cocos2d-iPhone，因此为了与 Objective-C 一致，Cocos2d-x 也采用了引用计数与自动回收的内存管理机制。熟悉 Objective-C 开发的读者可以轻松地掌握这种机制。后面还会看到，自动回收在工厂方法等对象生成器中还有特殊的用途。

■ 为了实现对象的引用计数记录,Cocos2d-x 实现了自己的根类 CCObject,引擎中的所有类都派生自 CCObject。在"CCObject.h"头文件中我们可以看到 CCObject 的定义：

```
class CC_DLL CCObject : public CCCopying
{
public:
    //对象 id，在脚本引擎中使用
    unsigned int m_uID;
    //Lua 中的引用 ID，同样被脚本引擎使用
    int m_nLuaID;
protected:
    //引用数量
    unsigned int m_uReference;
    //标识此对象是否已设置为 autorelease
    bool m_bManaged;
public:
    CCObject(void);
    virtual ~CCObject(void);
    void release(void);
    void retain(void);
    CCObject* autorelease(void);
    CCObject* copy(void);
    bool isSingleRefrence(void);
    unsigned int retainCount(void);
    virtual bool isEqual(const CCObject* pObject);
    virtual void update(ccTime dt) {CC_UNUSED_PARAM(dt);};
    friend class CCAutoreleasePool;
};
```

■ 每个对象包含一个用来控制生命周期的引用计数器，它就是 CCObject 的成员变量 m_u- Reference。我们可以通过 retainCount()方法获得对象当前的引用计数值。在对象通过构造函数创建的时候，该引用值被赋为 1，表示对象由创建者

所引用。在其他地方需要引用对象时，我们会调用 retain()方法，令其引用计数增 1，表示获取该对象的引用权；在引用结束的时候调用 release()方法，令其引用计数值减 1，表示释放该对象的引用权。

- 另一个很有趣的方法是 autorelease()，其作用是将对象放入自动回收池（CCAutore- leasePool）。当回收池自身被释放的时候，它就会对池中的所有对象执行一次 release()方法，实现灵活的垃圾回收。回收池可以手动创建和释放。除此之外，引擎在每次游戏循环开始之前也会创建一个回收池，在循环结束后释放回收池。因此，即使我们没有手工创建和释放回收池，每一帧结束的时候，自动回收池中的对象也都会被执行一次 release()方法。我们马上就会领略到 autorelease()的方便之处。

- 下面是一个简单的例子。可以看到，对象创建后，引用计数为 1；执行一次 retain()后，引用计数为 2；执行一次 release()后，引用计数回到 1；执行一次 autorelease()后，对象的引用计数值并没有立即减 1，但是在下一帧开始前，对象会被释放掉。

- 下面是测试代码：

```
fish = new CCSprite();
fish->init();
CCLog("retainCount after init: %d", fish->retainCount());
fish->retain();
CCLog("retainCount after retain: %d", fish->retainCount());
fish->release();
CCLog("retainCount after release: %d", fish->retainCount());
fish->autorelease();
CCLog("retainCount after autorelease: %d", fish->retainCount());
```

- 控制台显示的日志如下：

```
Cocos2d: retainCount after init: 1
Cocos2d: retainCount after retain: 2
Cocos2d: retainCount after release: 1
Cocos2d: retainCount after autorelease: 1
```

- 我们已经知道，调用了 autorelease()方法的对象（下面简称"autorelease 对象"），将会在自动回收池释放的时候被释放一次。虽然，Cocos2d-x 已经保证每一帧结束后释放一次回收池，并在下一帧开始前创建一个新的回收池，但是我们也应该考虑到回收池本身维护着一个将要执行释放操作的对象列表，如果在一帧之内生成了大量的 autorelease 对象，将会导致回收池性能下降。因此，在生成 autorelease 对象密集的区域（通常是循环中）的前后，我们最好可以手动创建并释放一个回收池。

- 我们可以通过回收池管理器 CCPoolManager 的 push()或 pop()方法来创建或释放回收池，其中的 CCPoolManager 也是一个单例对象。在这里，我们通过这段简单的代码来分析自动回收池的嵌套机制：

```
CCPoolManager::sharedPoolManager()->push();
for(int i=0; i<n; i++)
{
    CCString* dataItem = CCString::createWithFormat("%d", Data[i]);
    stringArray->addObject(dataItem);
}
CCPoolManager::sharedPoolManager()->pop();
```

- 这段代码包含了一个执行 n 次的循环，每次都会创建一个 autorelease 对象 CCString。为了保持回收池的性能，我们在循环前使用 push()方法创建了一个新的回收池，在循环结束后使用 pop()方法释放刚才创建的回收池。

- 不难看出，自动回收池是可嵌套的。通常，引擎维护着一个回收池，所有的 autorelease 对象都添加到了这个池中。多个自动回收池排列成栈结构，当我们手动创建了回收池后，回收池会压入栈的顶端，autorelease 对象仅添加到顶端的池中。当顶层的回收池被弹出释放时，它内部所有的对象都会被释放一次，此后出现的 autorelease 对象则会添加到下一个池中。
- 在自动回收池嵌套的情况下，每一个对象是如何加入自动回收池以及如何释放的，相关代码如下所示：

```
//步骤 a
obj1->autorelease();
obj2->autorelease();

//步骤 b
CCPoolManager::sharedPoolManager()->push();

//步骤 c
for(int i=0; i<n; i++) {
    obj_array[i]->autorelease();
}

//步骤 d
CCPoolManager::sharedPoolManager()->pop();

//步骤 e
obj3->autorelease();
```

- 上述代码的具体过程如图 2-7 所示，当执行完步骤 a 时，obj1 与 obj2 被加入到回收池 1 中，如图 2-7a 所示；步骤 b 创建了一个新的回收池，此时回收池 2 接管所有 autorelease 操作，如图 2-7b 所示；步骤 c 是一个循环，其中把 n 个对象加入回收池 2 中，如图 2-7c 所示；步骤 d 释放了回收池 2，因此回收池 2 中的 n 个对象都被释放了一次，同时回收池 1 接管 autorelease 操作；步骤 e 调用 obj3 的 autorelease()方法，把 obj3 加入回收池 1 中，如图 2-7e 所示。

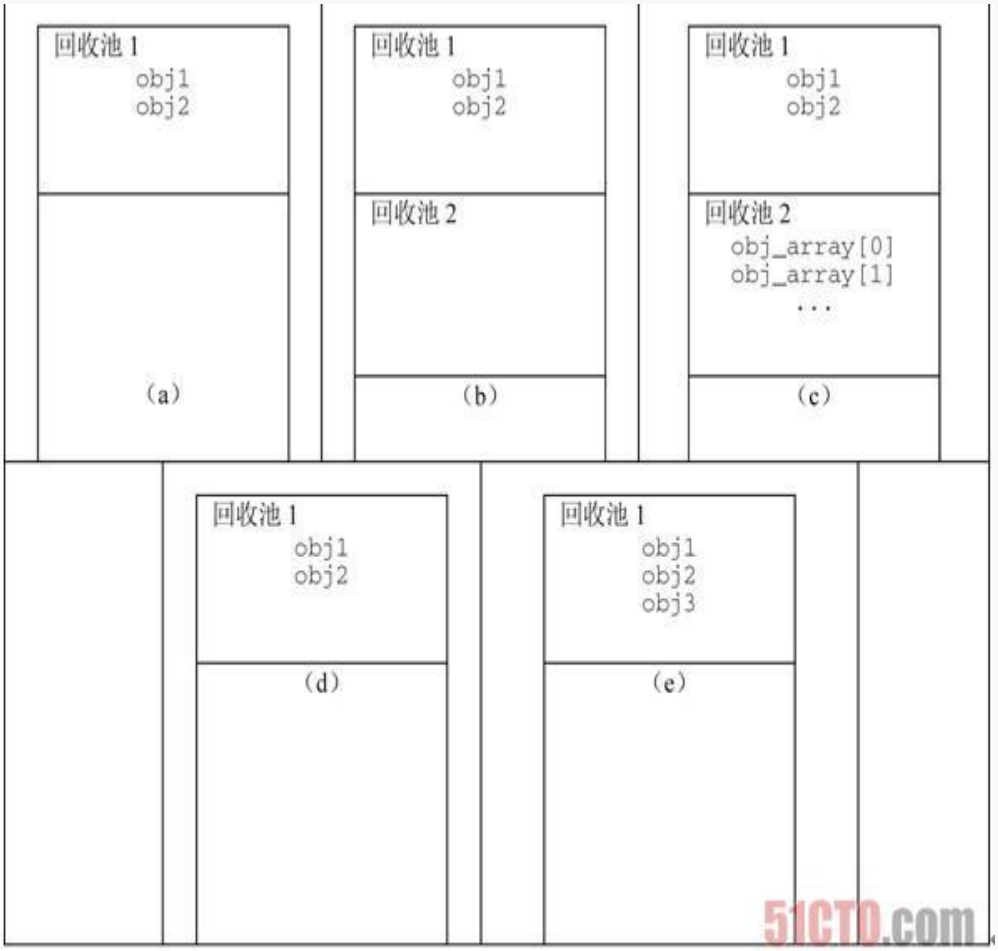


图2-7 嵌套池的示例

2.3.4 工厂方法

2.3.4 工厂方法

工厂方法是程序设计中一个经典的设计模式，指的是基类中只定义创建对象的接口，将实际的实现推迟到子类中。在这里，我们将它稍加推广，泛指一切生成并返回一个对象的静态函数。一个经典的工厂方法如同这样：

```
CCObject* factoryMethod() {
    CCObject* ret = new CCObject();
    //在这里对 ret 对象进行必要的初始化操作
    return ret;
}
```

这段看起来正常的代码其实隐藏着一个问题：工厂方法对 ret 对象的引用在函数返回时已经结束，但是它没有释放对 ret 的引用，埋下了内存泄露的隐患。但是，如果在函数返回前就执行 release()，这显然是不合适的，因为这会触发对象的回收，再返回的对象指针就成为了错误指针。

autorelease()方法很好地解决了这个问题。此函数结束时我们已经丧失了对 ret 的引用，为了把 ret 对象传递给接受者，需要对它进行一次 autorelease 操作，这是因为虽然我们调用了 autorelease 方法，但是对象直到自动回收池释放之前是不会被真正释放掉的（通常 Cocos2d-x 会在每一帧之间释放一次自动回收池），调用者有足够的时间来对它进行 retain 操作以便接管 ret 对象的引用权。因此，Cocos2d-x 的执行机制很巧妙地保证了回收池中的对象不会在使用完毕前释放。利用 autorelease()修改后的工厂方法如下：

```
CCObject* factoryMethod() {
    CCObject* ret = new CCObject();
    //这里对 ret 对象进行必要的初始化操作
    ret->autorelease();
    return ret;
}
```

在 2.2 节中，我们曾提到两种创建对象的方式。使用构造函数创建对象时，对象的引用计数为 1，因此调用者需要在使用完毕后谨慎地释放对象；使用工厂方法创建对象时，虽然引用计数也为 1，但是由于对象已经被放入了回收池，因此调用者没有对该对象的引用权，除非我们人为地调用了 retain()来获取引用权，否则，不需要主动释放对象。

2.3.5 关于对象传值

2.3.5 关于对象传值

将一个对象赋值给某一指针作为引用的时候，为了遵循内存管理的原则，我们需要获得新对象的引用权，释放旧对象的引用权。此时，release()和 retain()的顺序是尤为重要的。首先来看下面一段代码：

```
void SomeClass::setObject(CCObject* other) {
    this->object->release();
    other->retain();
    this->object = other;
}
```

这里存在的隐患是，当 other 和 object 实际上指向同一个对象时，第一个 release()可能会触发该对象的回收，这显然不是我们想看到的局面，所以应该先执行 retain()来保证 other 对象有效，然后再释放旧对象：

```
void SomeClass::setObject(CCObject* other) {
    other->retain();
    this->object->release();
    this->object = other;
}
```

其他可行的解决方案也有很多，例如使用 autorelease()方法来代替 release()方法，或在赋值前判断两个对象是否相同。在 Google 的 Objective-C 编程规范中，推荐使用 autorelease()方法代替 release()方法。

2.3.6 释放：release()还是 autorelease()？

2.3.6 释放：release()还是 autorelease()？

上面的两个例子实际上提出了一个问题：在使用 autorelease()可以达到与 release()同样的效果，甚至还能避免 release()的许多隐患的情况下，是不是应该完全用 autorelease()代替 release()呢？

实际上，autorelease()并不是毫无代价的，其背后的垃圾池机制同样需要占用内存和 CPU 资源，每次执行 autorelease()的过程，实际上对应的是执行成对的 retain()和 release()，以及一次成对的容器存取，还包括其他的逻辑判断。过多不必要的 autorelease()将导致垃圾池臃肿膨胀，在存在大量内存操作的程序中会尤为严重地挤占本来就紧张的系统资源。

此外，autorelease()只有在自动释放池被释放时才会进行一次释放操作，如果对象释放的次数超过了应有的次数，则这个错误在调用 autorelease()时并不会被发现，只有当自动释放池被释放时（通常也就是游戏的每一帧结束时），游戏才会崩溃。在这种情况下，定位错误就变得十分困难了。例如，在游戏中，一个对象含有 1 个引用计数，但是却被调用了两次

autorelease()。在第二次调用 autorelease()时，游戏会继续执行这一帧，结束游戏时才会崩溃，很难及时找到出错的地点。

- 因此，我们建议在开发过程中应该避免滥用 autorelease()，只在工厂方法等不得不用的情况下使用，尽量以 release()来释放对象引用。

2.3.7 容器

2.3.7 容器

- Cocos2d-x 引擎为我们提供了 CCArray、CCDictionary 等 Objective-C 风格的容器。对 C++标准库比较熟悉的读者可能疑惑，开发过程中为什么不直接使用 vector 等标准库已经提供的高效容器呢？
- 使用 Cocos2d-x 容器的一个重要原因在于 Cocos2d-x 的内存管理。一般来说，被存入容器的对象在移除之前都应该保证是有效的，回顾一下引用计数的管理原则，对象的存入和移除必须对应一组 retain()和 release()或者对应 autorelease()。直接使用 STL 容器，开发者势必进行烦琐重复的内存管理操作，而 Cocos2d-x 容器对这一过程进行了封装，保证了容器对对象的存取过程总是符合引用计数的内存管理原则。
- 按照 Cocos2d-x 容器的内存管理要求，存入容器的对象必须是 CCObject 或其派生类。同时，Cocos2d-x 的容器本身也是 CCObject 的派生类，当容器被释放时，它保存的所有元素都会被释放一次引用。以下代码节选自线性表容器 CCArray 的定义，CCArray 的代码位于引擎目录下的"cocos2dx\cocoa\CCArray.h（.cpp）"文件中：

```
class CC_DLL CCArray : public CCObject
{
public:
    ~CCArray();
    ...
    bool initWithObjects(CCObject* pObject, ...);
    ...
};
```

- 此外，对于跨语言移植游戏（如从 Objective-C 移植到 C++）的开发者而言，把原游戏中大量使用的容器全部替换为 STL 库容器是一个极富挑战性的任务。容器存在的意义不仅仅局限于内存管理方面，因此我们应该尽量采用 Cocos2d-x 提供的容器类。

2.3.8 相关辅助宏

2.3.8 相关辅助宏

- 引用计数很巧妙也很方便，但大部分处理过程涉及指针，难免比较烦琐，也容易出错。针对这个问题，Cocos2d-x 为我们准备了一系列辅助宏来简化代码，这些宏都包含在头文件"CCPlatform Macro.h"里。表 2-2 列出了与内存管理相关的宏。
- 表 2-2 Cocos2d-x 中与内存管理有关的宏

宏	描述
CC_SAFE_DELETE(p)	使用 delete 操作符删除一个 C++对象 p，如果 p 为 NULL，则不进行操作
CC_SAFE_DELETE_ARRAY(p)	使用 delete[]操作符删除一个 C++数组 p，如果 p 为 NULL，则不进行操作
CC_SAFE_FREE(p)	使用 free()函数删除 p，如果 p 为 NULL，则不进行操作
CC_SAFE_RELEASE(p)	使用 release()方法释放 Cocos2d-x 对象 p

	● 的一次引用，如果 p 为 NULL，则不进行操作
■ CC_SAFE_RELEASE_NULL(p)	● 使用 release()方法释放 Cocos2d-x 对象 p 的 ● 一次引用，再把 p 赋值为 NULL。如果 p 已 ● 经为 NULL，则不进行操作
■ CC_SAFE_RETAIN(p)	● 使用 reatin()方法增加 Cocos2d-x 对象 p 的一 ● 次引用。如果 p 为 NULL，则不进行操作

● 2.3.9 Cocos2d-x 内存管理原则

■ 2.3.9 Cocos2d-x 内存管理原则

■ 至此，我们已经对 Cocos2d-x 采用的内存管理机制有了一个完整的认识。最后，我们将总结使用 Cocos2d-x 开发游戏时内存管理的原则：程序段必须成对执行 retain()和 release()或者执行 autorelease()来声明开始和结束对象的引用；工厂方法返回前,应通过 autorelease()结束对该对象的引用;对象传值时,应考虑到新旧对象相同的特殊情况;尽量使用 release()而不是 autorelease()来释放对象引用，以确保性能最优；保存 CCObject 的子类对象时，应严格使用 Cocos2d-x 提供的容器，避免使用 STL 容器，对象必须以指针形式存入。

■ 如果希望自定义的类也拥有 Cocos2d-x 的内存管理功能，可以把 CCObject 作为自定义类的基类，并在实现类时严格遵守 Cocos2d-x 的内存管理原则。

● 2.4 生命周期分析

■ 2.4 生命周期分析

■ 在第 1 章中，我们运行了第一个 Cocos2d-x 游戏。在 1.4 节中，我们也介绍了控制游戏生命周期的 AppDelegate 文件。下面我们将结合一些游戏调试常用的技巧来分析 Cocos2d-x 程序的生命周期。

■ 再次打开"AppDelegate.cpp"文件。为了清楚地理解函数间的调用关系，不妨给每个函数的开头加上一个日志函数调用，在控制台打印该函数被调用的信息。因此，我们建立下面的一个日志类，利用临时变量在栈中的生命周期，搭配自动构造和析构函数产生的日志，并定义一个宏来跟踪函数名称，使用它们来打印函数的开始和结束。

■ 建立一个 LifeCircleLogger 类，并添加如下代码：

```
class LifeCircleLogger{
    string m_msg;
public:
    LifeCircleLogger(){}
    LifeCircleLogger(const string& msg):m_msg(msg){
        CCLog("%s BEGINS!",m_msg.c_str());}
    ~LifeCircleLogger(){CCLog("%s ENDS!",m_msg.c_str());}
};
#define LOG_FUNCTION_LIFE LifeCircleLogger(__FUNCTION__);
```

■ 这里出现的 CCLog 是 Cocos2d-x 的控制台输出函数，其参数方式与 C 语言的 printf 完全一致，用%d 表示整型，%s 表示字符串等。实际上，在 Windows 平台上，该函数正是通过包装 printf 函数实现的。在 iOS 和 Android 等平台上，这个函数有着同样的接口表示，并都可以在调试时打印信息到控制台。

■ 我们在 AppDelegate 类中所有函数的第一条语句前加入 LOG_FUNCTION_LIFE 宏：

```
AppDelegate::AppDelegate() {
    LOG_FUNCTION_LIFE
}
```

```
//当应用程序不再活动时，会调用此方法。当手机接到电话时，它也会被调用
void AppDelegate::applicationDidEnterBackground()
{
    LOG_FUNCTION_LIFE
    CCDirector::sharedDirector()->pause();
    SimpleAudioEngine::sharedEngine()->pauseBackgroundMusic();
}

//当应用程序重新活动时，会调用此方法
void AppDelegate::applicationWillEnterForeground()
{
    LOG_FUNCTION_LIFE
    CCDirector::sharedDirector()->resume();
    SimpleAudioEngine::sharedEngine()->resumeBackgroundMusic();
}

bool AppDelegate::applicationDidFinishLaunching()
{
    LOG_FUNCTION_LIFE

    //初始化导演类
    CCDirector *pDirector = CCDirector::sharedDirector();
    pDirector->setOpenGLView(&CCEGLView::sharedOpenGLView());

    //高分辨率屏幕（例如 Retina 屏幕）的资源管理
    //pDirector->enableRetinaDisplay(true);

    //启用 FPS 显示
    pDirector->setDisplayStats(true);

    //设置 FPS 上限。如果不加设置，则默认 FPS 上限为 60
    pDirector->setAnimationInterval(1.0 / 60);

    //创建一个场景，场景是一个 autorelease 对象
    CCScene *pScene = HelloWorld::scene();

    //运行场景
    pDirector->runWithScene(pScene);
    return true;
}
```

■ 启动游戏，然后操作如下：首先把游戏最小化，然后再把它恢复到前台，最后关闭游戏。完成后回到 VS，可以在控制台中看到函数调用顺序（如图 2-8 所示）。

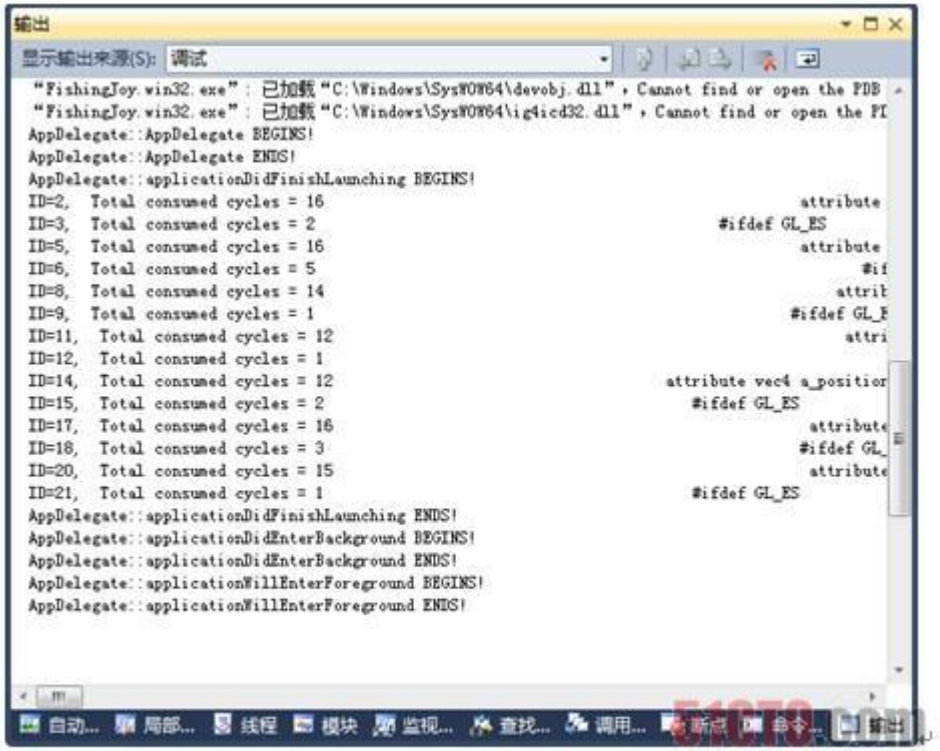


图2-8 日志内容

- 由此可知，AppDelegate 是整个程序的入口。实际上，AppDelegate 处理了程序生命周期中的 4 个重要事件点：程序完成初始化，程序进入后台，程序重回前台和程序结束退出。进入后台和重回前台两个函数完成暂停和恢复游戏的处理。appDidFinishingLaunching 则完成了加载游戏场景等工作。

● 2.5 小结

2.5 小结

- 在这一章中，我们首先介绍了游戏开发的基本概念，以及它们在 Cocos2d-x 中对应的具体实现，然后介绍了 Cocos2d-x 类似 Objective-C 的代码风格，最后介绍了 Cocos2d-x 独特的内存管理机制，并借用生命周期分析介绍了 Cocos2d-x 的调试技巧。下面总结了本章的重点知识。流程控制：场景是相对不变的游戏元素集合，游戏在场景间的切换就是流程控制。场景、层和精灵：它们是不同层次的游戏元素。通常，场景包含层，层包含精灵，场景与层是其他游戏元素的容器，而精灵是展示给玩家的图形。节点和渲染树：一切可以显示的游戏元素都是渲染树的节点。Cocos2d-x 通过遍历渲染树绘制游戏画面。场景、层或精灵作为渲染树节点，我们并没有对它们的层次做硬性限制，例如开发者可以向精灵中添加层。动作：作用于游戏元素，规定了游戏元素运动的方式。帧动画是作用于精灵的一种特殊动作。类似 Objective-C 的代码风格：使用初始化方法或工厂方法创建对象，使用访问器方法模拟属性。Cocos2d-x 提供了 Objective-C 风格的容器 CCArray 和 CCDictionary。内存管理：类似 Objective-C 的引用计数内存管理机制，同时提供半自动的对象自动回收池，实现灵活的管理内存。生命周期：AppDelegate 负责控制游戏的生命周期。CCLog：Cocos2d-x 提供的日志输出函数。

● 3.1 CCDirector：大总管

● 游戏的基本元素

- 在上一章中，我们详细介绍了游戏开发的概念以及 Cocos2d-x 与其他游戏引擎的不同之处，甚至已经学会了它与众不同的内存管理机制。想必读者已经很期待开始探索 Cocos2d-x 游戏开发的世界了。在这一章中，我们将结合具体的实例，从 Cocos2d-x 游戏开发的基本元素讲起。

细心的读者应该会留意到，前面我们在新建工程的时候，使用的名字为 FishingJoy。不错，这就是在移动终端上风靡一时的经典游戏《捕鱼达人》。从这章开始，我们将和读者一起，在学习引擎的过程中一步步创建这个游戏。

3.1 CCDirector：大总管

从字面上理解，这是一个"导演"类，CCDirector 是控制游戏流程的主要组件。回顾 HelloWorld 项目，程序生命周期中游戏加载期的最后一个方法 AppDelegate::applica-

tionDidFinishLaunching，其代码如下所示：

```
bool AppDelegate::applicationDidFinishLaunching()
{
    //初始化导演类
    CCDirector *pDirector = CCDirector::sharedDirector();
    pDirector->setOpenGLView(&CCEGLView::sharedOpenGLView());
    //高分辨率屏幕（例如 Retina 屏幕）的资源管理
    //pDirector->enableRetinaDisplay(true);
    //启用 FPS 显示
    pDirector->setDisplayStats(true);
    //设置 FPS 上限。如果不加设置，则默认 FPS 上限为 60
    pDirector->setAnimationInterval(1.0 / 60);
    //创建一个场景，场景是一个 autorelease 对象
    CCScene *pScene = HelloWorld::scene();
    //运行场景
    pDirector->runWithScene(pScene);
    return true;
}
```

CCDirector 的工作确实跟导演非常类似，主要负责以下工作。

游戏呈现方面的设定，包括设定游戏呈现的窗口、FPS 显示、默认帧率上限、纹理颜色位宽等。

切换当前的游戏场景，暂停或恢复游戏场景的运行。

总而言之，游戏在 CCDirector 的管理下完成了呈现设定与流程控制。

CCDirector 扮演着全局大总管的角色，因而很自然地采用了单例的设计模式。在程序的任何地方，都可以通过下面的简单代码访问到：

CCDirector *pDirector = CCDirector::sharedDirector();

在 CCDirector 中，我们定义了以下管理场景的方法。

runWithScene(CCScene* scene)：启动游戏，并运行 scene 场景。这个方法在主程序启动时第一次启动主场景时调用。

replaceScene(CCScene* scene)：直接使用传入的 scene 替换当前场景来切换画面，当前场景将被释放。这是切换场景时最常用的方法。

pushScene(CCScene* scene)：将当前运行中的场景暂停并压入到代执行场景栈中，再将传入的 scene 设置为当前运行场景。

popScene：释放当前场景，再从代执行场景栈中弹出栈顶的场景，并将其设置为当前运行场景。如果栈为空，则直接结束应用。与 pushScene 成对使用，可以达到形如由主界面进入设置界面，然后回到主界面的效果。

- pause：暂停当前运行场景中的所有计时器和动作，场景仍然会显示在屏幕上。
- resume：恢复当前运行场景中被暂停的计时器和动作。它与 pause 配合使用。
- end：结束场景，同时退出应用。
- 值得注意的一点是，以上三种切换场景的方法（replaceScene、pushScene、popScene）均是先将待切换的场景完全加载完毕后，才将当前运行的场景释放掉。所以，在新场景恰好完全加载完毕的瞬间，系统中同时存在着两个场景，这将是对内存的一个考验，若不注意的话，切换场景可能会造成内存不足。

3.2 CCScene：场景

3.2 CCScene：场景

- 了解了 CCDirector 之后，接下来介绍 CCScene 这个与它紧密相关的游戏组件。在 2.1 节中，我们已经学习了场景以及它在流程控制中的地位。在 Cocos2d-x 中，CCScene 定义了一个场景。场景只是层的容器，包含了所有需要显示的游戏元素。因此相对于其他游戏元素，CCScene 并没有提供什么特别的功能，就是一个十分简单的类。除了作为层的容器，场景的另一个作用就是流程控制。利用 CCDirector:: replaceScene 等方法，我们可以使游戏在不同的场景中自由切换。
- 通常，当我们需要完成一个场景时，会创建一个 CCScene 的子类，并在子类中实现我们需要的功能。例如，我们可以在子类的初始化方法中载入游戏资源，为场景添加层，启动音乐播放，等等。
- 通常，场景之间需要一定的过渡衔接效果，否则，场景的切换会显得十分突兀。为此，Cocos2d-x 提供了很多华丽的场景切换特效，例如翻页、波浪、淡出淡入等。这些特效是通过派生自 CCScene 的 CCTransitionScene 系列特效类来实现的。我们也可以模仿 Cocos2d-x 内置的场景切换特效代码，来编写属于自己的特效。关于场景切换特效的用法与原理，我们将在 5.2 节中详细介绍。

3.3 CCLayer：层

3.3 CCLayer：层

- CCLayer 定义了一个层。与 CCScene 类似，层也扮演着容器的角色。然而与场景不同的是，层通常包含的是直接呈现在屏幕上的具体内容：我们需要在层中放入精灵、文本标签或其他游戏元素；设置游戏元素的属性，如位置、方向和大小；设置游戏元素的动作等。由此可见，游戏开发的大部分编码时间都用在创建层上。通常，层中的对象功能类似，耦合较紧，与层中游戏内容相关的逻辑代码也编写在层内。
- 在组织好层后，只需要把层按照顺序添加到场景中就可以显示出来了。要向场景中添加层，我们可以使用 addChild 方法。addChild 方法共有三个定义，具体如下所示：

```
void addChild(CCNode* child);
void addChild(CCNode* child, int zOrder);
void addChild(CCNode* child, int zOrder, int tag);
```

- 其中 child 参数为将要添加的节点。对于场景而言，通常我们添加的节点就是层。先添加的层会被置于后添加的层之下。如果想要为它们指定先后次序，可以使用不同的 zOrder 值，zOrder 代表了该节点下元素的先后次序，值越大则显示顺序越靠上。zOrder 的默认值为 0。tag 是元素的标识号码，如果为子节点设置了 tag 值，就可以在它的父节点中利用 tag 值找到它了。这里我们可以选择自己需要的方法来向场景中添加层。
- 还记得我们在 2.1 节中提到的捕鱼游戏场景的构成吗？捕鱼游戏的场景大致由背景层（backgroundLayer）、动作层（actionLayer）、触摸层（touchLayer）和菜单层（menuLayer）组成。假设这些层已经完成，那么我们最后要做的就是在游戏场景的初始化方法中把它们添加到场景中：

```
this->addChild(backgroundLayer, 0);
this->addChild(actionLayer, 100);
```

```
this->addChild(touchLayer,200);
this->addChild(menuLayer, 500);
```

CCLayer 的另一个十分重要的功能是可以接受用户输入事件，包括触摸、加速度计和键盘输入等。CCLayer 中与用户输入事件相关的成员如表 3-1 所示。

表 3-1　CCLayer 中与输入事件相关的成员表

成员类型	名称	描述
属性	TouchEnabled	获取或设置是否接受触摸事件
	AccelerometerEnabled	获取或设置是否接受加速度计事件
	KeypadEnabled	获取或设置是否启用键盘输入支持
回调函数	ccTouchBegan	带目标的触摸事件的回调函数
	ccTouchMoved	
	ccTouchEnded	
	ccTouchCancelled	
	ccTouchesBegan	标准触摸事件的回调函数
	ccTouchesMoved	
	ccTouchesEnded	
	ccTouchesCancelled	
	registerWithTouchDispatcher	注册触摸事件的回调函数， 在此函数内设置需要注册的触摸类型
	didAccelerate	加速度计改变事件的回调函数

这些特性将在第 7 章中详细介绍。

3.4.1　纹理

3.4　CCSprite：精灵

CCSprite 可以说是游戏中最重要的组成元素，它描述了游戏中的精灵，是 CCNode 的一个最重要也最灵活的子类。说它重要是因为 CCSprite 代表了游戏中一个最小的可见单位，说它灵活则是由于其装载了一个平面纹理，具有丰富的表现力，而且可以通过多种方式加载。如果说 CCScene 和 CCLayer 代表了宏观的游戏元素管理，那么 CCSprite 则为微观世界提供了丰富灵活的细节表现。从游动的鱼、飞行的子弹到可旋转的炮台，都可以通过 CCSprite 实现。下面我们将从纹理开始详细介绍 CCSprite。

3.4.1　纹理

在介绍精灵前，我们先介绍纹理的概念。我们可以认为纹理就是一张图片，这张图片被精灵显示出来。更深层地讲，纹理是 3D 游戏中绘制到物体表面上的图案。虽然 Cocos2d-x 是平面游戏引擎，但它仍然使用了 3D 绘图库 OpenGL。这样一来，我们既可以利用图形加速器提高绘图效率，也可以在游戏中加入 3D 变换特效，实现更绚丽的效果。为了在 3D 环境中绘制平面图形，Cocos2d-x 只需在 3D 空间中垂直于视线的平面上绘制矩形，在矩形的表面使用纹理贴图即可。

3.4.2　创建精灵

3.4.2　创建精灵

在实际使用中，精灵是由一个纹理创建的。在不加任何设置的情况下，精灵就是一张显示在屏幕上的图片。通常精灵置于层下，因此我们首选在层的初始化方法中创建精灵，设置属性，并添加到层中。有多种方式可以创建精灵，最常用的方式是使用一个图片文件来创建精灵。这里我们使用 CCSprite 的工厂方法 create 来创建精灵：

```
CCSprite* fish = CCSprite::create("fish.png");
```

- 这个工厂方法包含一个字符串参数，表示精灵所用纹理的文件名。CCSprite 会自动把图片作为纹理载入到游戏中，然后使用纹理初始化精灵。
- 精灵不但可以显示一个完整的纹理，也可以仅显示纹理的一部分。下面的代码使用两个参数的 create 工厂方法创建了一个精灵，这个精灵使用"fishes.png"作为纹理，但是仅显示纹理左上角 100×100 像素大小的部分：

```
CCSprite* smallFish = CCSprite::create("fishes.png", CCRectMake(0, 0, 100, 100));
```

- 在上面的这行代码中，第一个参数是所使用纹理的文件名，第二个参数是一个 CCRect 类型的结构体，表示纹理中显示出来的矩形部分。CCRectMake(x,y,width,height)函数可以用来方便地创建 CCRect。值得注意的是，纹理的坐标系中原点(0,0)位于左上角，原点向右是 x 轴的正方向，原点向下是 y 轴的正方向。在 3.5 节中，我们会详细介绍 Cocos2d-x 的坐标系统。

3.4.3 设置精灵的属性

- 3.4.3 设置精灵的属性

- 创建了精灵后，我们还需要把精灵安排在合适的位置，否则引擎也不能决定精灵将如何呈现出来。诸如精灵的位置、方向、缩放比例等参数都是精灵的属性，我们在层中添加精灵之前，需要对它们进行恰当地设置。下面的代码首先获取屏幕大小，然后根据屏幕大小把精灵置于屏幕中央：

```
CCSize size = CCDirector::sharedDirector()->getWinSize();
CCPoint pos = ccp(size.width / 2, size.height / 2);
fish->setPosition(pos);
```

- 我们可以看到，在这段代码中我们修改了精灵的 Position 属性。Position 属性是一个 CCPoint 类型的结构体，表示精灵在层中的位置，它是精灵相对于层的坐标。ccp(x,y) 与 CCRectMake 类似，是可以用来快速创建 CCPoint 的宏。把精灵的位置坐标设置为屏幕长宽的一半，就可以使精灵位于屏幕中央了。
- 精灵具有十分丰富的属性，我们可以利用这些属性让精灵灵活地呈现出来。但实际上，这些属性不仅仅是精灵才拥有的，它们也属于我们将在 3.5 中介绍的 CCNode。CCSprite 继承自 CCNode，因而也继承了它的全部属性。

3.4.4 向层中添加精灵

- 3.4.4 向层中添加精灵

- 设置完精灵的属性后，就该把精灵添加到层中了。实际上 CCSprite 与 CCLayer 都继承自 CCNode，向一个游戏元素中添加其他游戏元素的 addChild 是 CCNode 包含的方法，因此我们完全可以如同向场景中添加层一样，把精灵添加到层中。继续 3.4.3 节中的例子，我们把 fish 对象添加到这个层中：

```
this->addChild(fish);
```

- 在《捕鱼达人》游戏中，会定时生成鱼群。我们刚才已经做到创建一个精灵，修改精灵的属性，以及把精灵添加到层中了。现在，我们可以为《捕鱼达人》的动作层编写一个 createFish()方法来在层中创建一条鱼了。下面是创建鱼精灵的代码：

```
void SpriteLayer::createFish()
{
    CCSize size = CCDirector::sharedDirector()->getWinSize();
    CCSprite* fish = CCSprite::create("fish.png");
    fish->setTag(fish_red_tag);
    fish->setPosition(ccp(size.width / 2, size.height / 2));

    _fishes->addObject(fish);
```

```
    //设置动作或其他操作

    this->addChild(fish);
}
```

其中_fishes 是 CCArray 的实例，是一个包含了所有鱼的数组。把鱼添加到_fishes 中可以方便以后我们将要进行的检测碰撞。炮台这类精灵只需要一个，因此可以在初始化层时将其添加到动作层中：

```
bool SpriteLayer::init()
{
    bool bRet = false;
    do {
        CC_BREAK_IF(!CCLayer::init());

        CCSize winSize = CCDirector::sharedDirector()->getWinSize();

        _fishes = CCArray::create();
        _fishes->retain();

        /*炮台的初始化*/
        curCannonLevel = 1;
        char cannonPath[80];
        sprintf_s(cannonPath, s_pCannon, curCannonLevel);
        cannon = CCSprite::create(cannonPath);
        cannon->setPosition(ccp(winSize.width / 2,
            cannon->getContentSize().height / 3));
        this->addChild(cannon, 2);

        bRet = true;
    } while (0);

    return bRet;
}
```


3.4.5 常用成员

除了前面介绍过的部分成员外，CCSprite 还拥有以下常用的成员。

◆ 初始化方法

CCSprite 拥有许多种不同的初始化方法，可以方便地创建精灵。使用这些方法，我们不仅可以通过图片文件来创建精灵，还可以直接使用纹理或精灵框帧来创建精灵。

使用图片文件

使用图片文件创建精灵的相关方法如下：

```
static CCSprite* create(const char *pszFileName);
static CCSprite* create(const char *pszFileName, const CCRect& rect);
```

- `bool initWithFile(const char *pszFilename);`
- `bool initWithFile(const char *pszFilename, const CCRect& rect);`

  - 其中 pszFileName 为图片的文件名，直接传入图片文件相对于"Resource"文件夹的路径即可；rect 为可选参数，用于指定精灵显示纹理的部分，它使用前面介绍的纹理坐标系。
  - 除了使用前两个静态工厂方法创建精灵以外，也可以使用它们的构造函数加初始化方法。
  - 使用 CCTexture2D
  - 使用 CCTexture2D 纹理创建精灵的相关方法如下：

- `static CCSprite* create(CCTexture2D *pTexture);`
- `static CCSprite* create(CCTexture2D *pTexture, const CCRect& rect);`
- `bool initWithTexture(CCTexture2D *pTexture);`
- `bool initWithTexture(CCTexture2D *pTexture, const CCRect& rect);`

  - CCTexture2D 类型的 pTexture 参数为纹理对象，可以使用 CCTextureCache 类的 addImage 方法把图片文件装载为纹理并返回，而 rect 与使用图片文件创建精灵的 rect 参数相同。
  - 使用 CCSpriteFrame 创建
  - 使用 CCSpriteFrame 精灵框帧创建精灵的相关方法如下：

- `static CCSprite* create(CCSpriteFrame *pSpriteFrame);`
- `bool initWithSpriteFrame(CCSpriteFrame *pSpriteFrame);`

  - CCSpriteFrame 类型的 pSpriteFrame 参数为纹理框帧。CCSpriteFrame 保存了一个 CCTexture2D 的引用与一个 CCRect 来表示纹理中的一部分。使用 CCSpriteFrame 初始化精灵是，也可使精灵显示部分纹理。

    ◆ **纹理相关的属性**

  - CCSprite 提供了以下与纹理相关的属性，用于获取或设置精灵内容。
  - CCTexture2D* Texture：获取或设置精灵所用的纹理。使用此方法设置纹理后，精灵将会显示一个完整的纹理。
  - CCRect TextureRect：获取或设置纹理显示部分。此 CCRect 采用纹理坐标，即左上角为原点。
  - CCSpriteBatchNode* BatchNode：获取或设置精灵所属的批节点。关于批节点，我们将会在第 10 章进行详细讲解。

    ◆ **纹理相关的方法**

  - CCSprite 提供了以下与纹理相关的方法。
  - void setDisplayFrame(CCSpriteFrame *pNewFrame)：设置显示中的纹理框帧，其中 pNewFrame 为新的纹理框帧，其代表的纹理或纹理的显示部分都可以与旧的框帧
  - 不同。
  - CCSpriteFrame* displayFrame：获取正在显示的纹理框帧。
  - bool isFrameDisplayed(CCSpriteFrame *pFrame)：返回一个值，表示 pFrame 是否是正在显示中的纹理框帧。

    ◆ **颜色相关的属性**

  - CCSprite 提供了以下与颜色相关的属性。

- ccColor3 Color：获取或设置叠加在精灵上的颜色。ccColor3 由三个颜色分量（红色、绿色和蓝色分量）组成。默认为纯白色，表示不改变精灵的颜色，如果设置为其他值，则会改变精灵的颜色。
- GLubyte Opacity：获取或设置精灵的不透明度。GLubyte 为 OpenGL 的内置类型，表示一个无符号 8 位整数，取值范围从最小值 0 到最大值 255。
- bool OpacityModifyRGB：获取或设置精灵所使用的纹理数据是否已经预乘 Alpha 通道。当包含 Alpha 通道的图片显示错误时，可以尝试修改这个属性。

3.5 CCNode 与坐标系

3.5 CCNode 与坐标系

在第 2 章中，我们已经介绍了渲染树的概念，现在让我们首先复习一下渲染树的要点。Cocos2d-x 采用了场景、层、精灵的层次结构来组织游戏元素，与此同时，这个层次结构还对应了游戏的渲染层次，因此游戏元素可以组织成树形结构，称作渲染树。Cocos2d-x 把渲染树上的每一个游戏元素抽象为一个节点，即 CCNode。一切游戏元素都继承自 CCNode，因此它们都具有 CCNode 所提供的特性。

CCNode 定义了一个可绘制对象的通用特性，包括位置、缩放、是否可见、旋转角度等。节点的最基本的功能包括：

- 包含其他 CCNode 对象；
- 接受各种事件与回调函数，如定时器事件；
- 运行动作。

在这一节中，我们将详细介绍节点特性以及前两个基本功能，运行动作的机制我们将在第 4 章中详细介绍。

渲染树中包含着许多子节点的 CCNode，类似一张画布，允许其他画布以不同的顺序叠加覆盖在自己上面，一旦改变该节点的缩放和旋转等特性，就会影响该节点（包括它的子节点）的显示效果。例如，把节点的旋转角度设置为 90 度，则它旋转了 90 度，但它的子节点仍然保持相对位置和角度不变。绘图的顺序通过 zOrder 属性描述，这与 3.3 节介绍 CCLayer 时提到的 addChild 中的 zOrder 参数是相同的。

3.5.1 坐标系与绘图属性（1）

3.5.1 坐标系与绘图属性（1）

在本节之前，我们已经多次接触到了游戏元素的绘图属性，如控制游戏元素位置的 Position 属性，这些属性都是由 CCNode 提供的。在深入讨论 CCNode 属性之前，我们先介绍一下 Cocos2d-x 的坐标系统。

Cocos2d-x 中的坐标系

在 Cocos2d-x 中，存在两种坐标系。

绘图坐标系。它是最常见的坐标系，与 OpenGL 采用的坐标系相同，以左下角为原点，向右为 x 轴正方向，向上为 y 轴正方向，如图 3-1 所示。在 Cocos2d-x 中，一切绘图相关的操作都使用绘图坐标系，如游戏元素中的 Position 和 AnchorPoint 等属性。

图3-1 Cocos2d-x的绘图坐标系

- 纹理坐标系。纹理坐标系以左上角为原点，向右为 x 轴正方向，向下为 y 轴正方向，如图 3-2 所示。在 Cocos2d-x 中，只有从纹理中截取部分矩形时才使用这个坐标系，如 CCSprite 的 TextureRect 属性。

图3-2 纹理坐标系

- **绘图属性**
- 简单地接触了 Cocos2d-x 中的坐标系后，下面我们来介绍 CCSprite 所拥有的绘图相关属性。利用这些属性，我们可以对精灵呈现的方式进行精确的控制。

- CCRect ContentSize：获取或设置此节点的内容大小。任何一个节点都需要确定它的内容大小，以便进行图形变换。对于精灵来说，ContentSize 是它的纹理显示部分的大小；对于层或场景等全屏的大型节点来说，ContentSize 则是屏幕大小。
- CCPoint AnchorPoint 与 CCPoint Position：AnchorPoint 用于设置一个锚点，以便精确地控制节点的位置和变换。AnchorPoint 的两个参量 x 和 y 的取值通常都是 0 到 1 之间的实数，表示锚点相对于节点长宽的位置。例如，把节点左下角作为锚点，值为(0,0)；把节点的中心作为锚点，值为(0.5,0.5)；把节点右下角作为锚点，值为(1,0)。精灵的 AnchorPoint 默认值为(0.5,0.5)，其他节点的默认值为(0,0)。图 3-3 演示了精灵三个不同锚点的值及其位置。

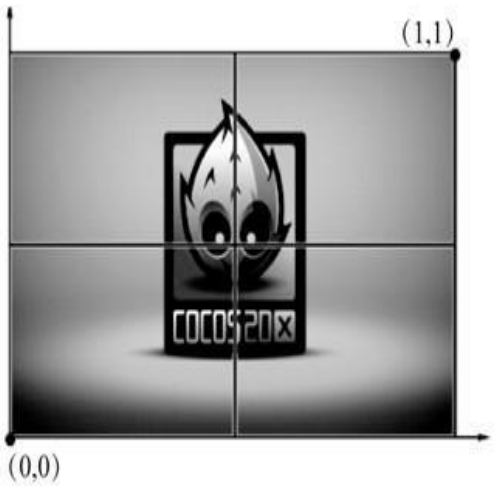


图3-3 锚点的位置

● 3.5.1 坐标系与绘图属性（2）

■ 3.5.1 坐标系与绘图属性（2）

- Position 用于设置节点的位置。由于 Position 指的是锚点在父节点中的坐标值，节点显示的位置通常与锚点有关。因此，如果层与场景保持默认的位置，只需把层中精灵位置设为窗口长宽的一半即可让它显示在屏幕中央。
- 图 3-4 演示了 AnchorPoint 与 Position 属性在节点中的含义。当多个节点嵌套时，每个节点的坐标系原点位于其内容的左下角。因此，锚点在父节点中的坐标实际上是它相对于父节点左下角的坐标值。如图 3-5 所示，每个节点的位置实际上都是自身锚点相对于父亲左下角的位置。

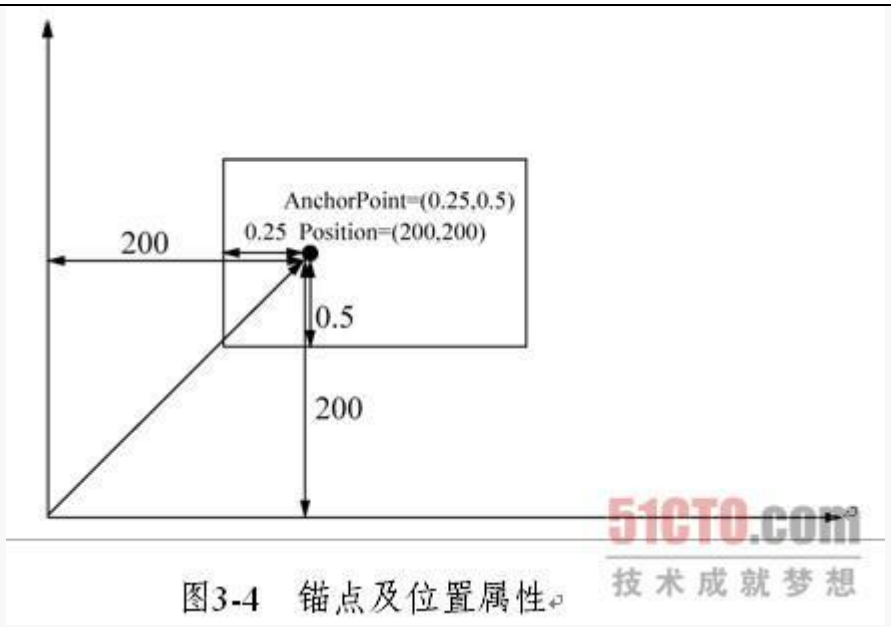


图3-4 锚点及位置属性

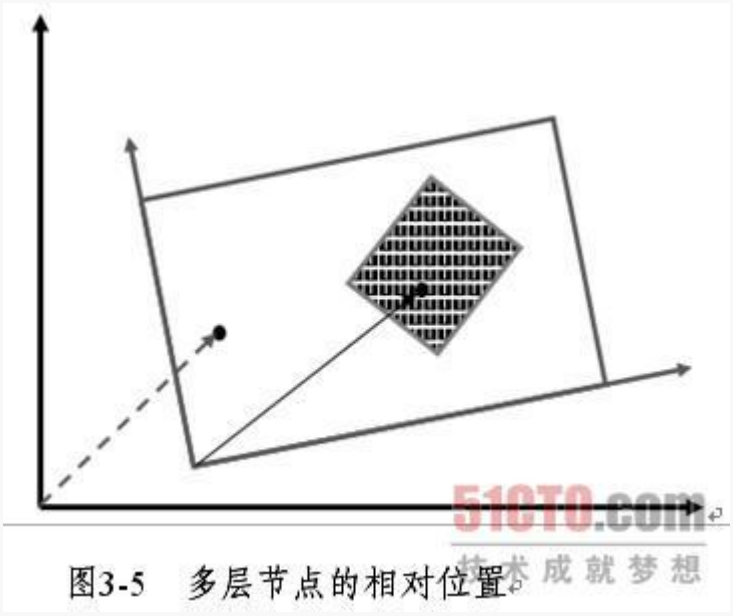


图3-5 多层节点的相对位置

- 对于场景或层等大型节点,它们的 IgnoreAnchorPointForPosition 属性为 true,此时引擎会认为 AnchorPoint 永远为(0,0);而其他节点的该属性为 flase，它们的锚点不会被忽略。
- float Rotation：获取或设置节点的旋转角度。节点以自己的锚点为中心顺时针旋转一定量，单位是角度。旋转角度可以是任意实数。
- float Scale（以及 float ScaleX 与 float ScaleY）：Scale 用于获取或设置节点的缩放比例。节点以锚点为中心缩放该比例。Scale 的值代表整体缩放比例，而 ScaleX 与 ScaleY 分别代表 X 方向与 Y 方向的缩放比例。默认情况下，这三个属性的值都是 1，表示节点不被缩放。如果设置 Scale 属性，则 ScaleX 和 ScaleY 都会随之变为相同的值。当然，我们也可以给 ScaleX 与 ScaleY 设置不同的值，那样 Scale 属性的值就没有意义了。
- bool Visible：获取或设置节点的可见性。当 Visible 为 true 时，节点会被显示，反之节点不会被显示。在节点不被显示的时候，也不会被调用绘图方法（visit 与 draw）。这个属性与众不同的是，它的访问器没有遵循属性的命名规范。以下为它的访问器方法：

- `bool isVisible();`
- `void setVisible(bool visible);`

- float SkewX 与 float SkewY：获取或设置斜切角度。节点以锚点为中心，平行 x 轴或 y 轴方向作一定角度的变形。SkewX 为平行 x 轴顺时针的变形，SkewY 为平行 y 轴逆时针的变形，单位为角度。SkewX 与 SkewY 的默认值为 0，表示节点没有斜切变形。
- int Tag：获取或设置节点的标号。在 Cocos2d-x 中，Tag 的作用类似于标识符，以便快速地从节点的所有子节点中找出所需节点。Tag 可以用于定位子节点，因此添加到同一节点的所有 CCNode 之中，不能有两个节点的 Tag 相同，否则就给定位带来了麻烦。与 Tag 相关的方法有 getChildByTag、removeChildByTag 等。
- void* UserData：获取或设置与节点相关的额外信息。UserData 为 void*类型，我们可以利用这个属性来保存任何数据。
- 部分常用属性的修改效果如图 3-6 所示。前面介绍了开发中与绘图紧密相关的属性，CCNode 的其他属性如表 3-2 所示。

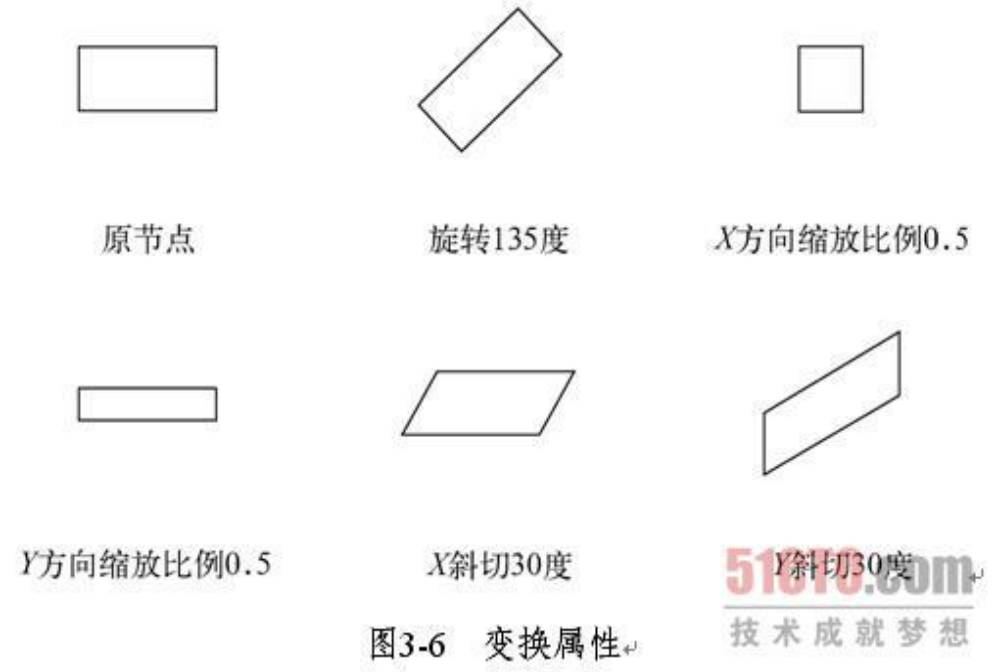


图3-6 变换属性

-

■ 3.5.1 坐标系与绘图属性（3）

■ 表 3-2 CCNode 的其他属性

● 属性	● 描述
● CCArray* Children	● 获取保存了该节点所有子引用的数组
● CCNode* Parent	● 获取或设置该节点的父节点
● CCCamera* CameraRETURN	● 获取或设置该节点的摄像机状态。摄像机 ● （CCCamera）定义了绘制该节点时的视点， ● 通常用于实现特效
● CCGridBase* Grid	● 获取或设置该节点的网格特效状态。网格应用 ● 于绘图平面上，可以实现水纹等 3D 特效
● CCGLProgram* ShaderProgram	● 获取或设置该节点的 Shader 程序。CCGLProgram ● 是 OpenGL 的 glProgram 的封装
● CCActionManager* ActionManager	● 获取或设置该节点所使用的动作管理器。当为节 ● 点设置了新的动作管理器时，正在执行的动作 ● 都会被丢弃。动作管理器的相关内容将在第 4 章介绍

CCScheduler* Scheduler	获取或设置该节点所使用的计时器管理器。 当为节点设置了新的计时器管理器时， 正在运行的计时器都会被丢弃

3.5.2 节点的组织

3.5.2 节点的组织

前面我们已经接触了组织节点的基本方式：addChild 方法。在介绍 addChild 方法时，我们是以向场景中添加层作为例子讲解的。而事实上，addChild 是 CCNode 提供的方法，我们可以把任何的 CCNode 添加到另一个节点之中。除了添加节点之外，CCNode 还提供了一系列管理节点的方法，表 3-3 列出了这些方法。

表 3-3 组织节点相关的方法

方法	描述
addChild(CCNode* child)	把 child 添加到当前节点之中
removeFromParentAndCleanup (bool cleanup)	把当前节点从其父节点中移除，如果 cleanup 为 true，则执行 clean 方法
removeChild(CCNode* child, bool cleanup)	从当前节点中移除 child 节点，如果 cleanup 为 true，则调用 child 的 clean 方法
removeChildByTag(int tag, bool cleanup)	从当前节点中移除标号为 tag 的节点
removeAllChildrenWithCleanup (bool cleanup)	移除当前节点的所有子节点
getChildByTag(int tag)	返回当前节点中标号为 tag 的节点
cleanup	停止此节点的全部动作与计时器

3.5.3 定时器事件

3.5.3 定时器事件

利用场景、层和精灵等游戏元素，我们可以构建游戏的框架，但是此时的游戏仍然是静止不动的。在一切游戏中，游戏的状态都会随着时间的流逝而改变，同时我们还需要定时进行一些逻辑判断，例如鱼和子弹的碰撞检测。为了解决以上问题，我们引入了定时器的概念。定时器是以一定时间间隔连续引发游戏事件的工具。很显然，定时器就是使游戏动态变化所需的工具。Cocos2d-x 为我们提供了两种方式实现定时机制--使用 update 方法以及使用 schedule 方法，下面简要介绍这两种方式。

update 定时器

第一种定时机制是 CCNode 的刷新事件 update 方法，该方法在每帧绘制之前都会被触发一次。由于绘图帧率有限，而每次更新最终会反映到画面上，所以在每帧之间刷新一次已经足够应付大部分游戏逻辑处理的要求了。我们捕鱼的碰撞检测机制就完全可以通过 update 事件来实现。

CCNode 默认并没有启用 update 事件，为了启用定时器，我们需要调用 scheduleUpdate 方法，并重载 update 以执行自己的代码。对应地，我们可以使用 unscheduleUpdate 方法停止定时器。

schedule 定时器

另一种定时机制是 CCNode 提供的 schedule 方法，可以实现以一定的时间间隔连续调用某个函数。由于引擎的调度机制，这里的时间间隔必须大于两帧的间隔，否则两帧期间的多次调用会被合并成一次调用。所以 schedule 定时器通常用在间隔较长的定时调用中，一般来说，事件间隔应在 0.1 秒以上。实际开发中，许多定时操作都通过 schedule 定时器实现，例如鱼群的定时生成、免费金币的定时刷新等。下面我们将以《捕鱼达人》为例来讲解如何使用 schedule 定时器。

■ 在《捕鱼达人》中，游戏场景里的鱼群与炮弹随着时间的推移不断移动，因此我们需要定时刷新鱼群的位置，并进行碰撞检测。我们把处理鱼群移动和碰撞监测的代码放置在主游戏场景 GameScene 的 updateGame 方法中。在 GameScene 的 init 初始化方法中添加以下代码来启用定时器：

```
this->schedule(schedule_selector(GameScene::updateGame));
```

■ 而 updateGame 函数包含了两个主要操作--刷新鱼群位置和碰撞检测，其代码如下所示：

```
void GameScene::updateGame(ccTime dt)
{
    SpriteLayer* sLayer = (SpriteLayer*)this->getChildByTag(sprite_layer_tag);
    sLayer->updateFishMovement(dt);
    sLayer->checkBulletCollideWithFish();
}
```

■ CCNode 的 schedule 方法接受一个函数指针并启动一个定时器，利用 schedule 方法不同的重载可以指定触发间隔与延时。schedule_selector 则是一个把指定函数转换为函数指针的宏，用于创建 schedule 方法所需的函数指针。传入这个宏的函数应该包含一个 float 参数，表示距离前一次触发事件的时间间隔。

■ **定时器相关方法**

■ 表 3-4 列举了 CCNode 中与定时器相关的方法。

■ 表 3-4　与定时器相关的方法

方法	描述
isScheduled(SEL_SCHEDULE selector)	返回一个值，表示 selector 对应的函数是否已被添加为定时器
scheduleUpdate	启用 update 定时器
scheduleUpdateWithPriority(int priority)	启用 update 定时器，并设定定时器的优先级
unscheduleUpdate	取消 update 定时器
schedule(SEL_SCHEDULE selector, float interval,unsigned int repeat, float delay)	添加一个 schedule 定时器，其中 selector 参数为定时器的事件函数，interval 参数为定时器的时间间隔，repeat 参数为定时事件触发一次后还会再次触发的次数（默认值为 kCCRepeatForever，表示触发无穷多次），delay 为第一次触发事件前的延时。此方法拥有多个重载，开发者可以选择自己所需的重载函数。此处时间都以秒为单位
scheduleOnce(SEL_SCHEDULE selector, float delay)CONTROL	添加一个 schedule 定时器，但定时器只触发一次
unschedule(SEL_SCHEDULE selector)ESC	取消 selector 所对应函数的定时器
unscheduleAllSelectors	取消此节点所关联的全部定时器
pauseSchedulerAndActions	暂停此节点所关联的全部定时器与动作
resumeSchedulerAndActions	继续执行此节点所关联的定时器与动作

■ 定时器机制是 Cocos2d-x 调度机制的基础，第 4 章将介绍的动作机制实际上也依赖定时器实现。由于 Cocos2d-x 的调度是纯粹的串行机制，因此所有函数都运行在同一个线程，不会存在并行程序的种种麻烦，这大大简化了编程的复杂性。

- 3.5.4 其他事件

■ 3.5.4 其他事件

■ 除了定时器会不断地提供触发事件外，Cocos2d-x 还为我们提供了一些其他与流程控制相关的事件，如表 3-5 所示。

■ 表 3-5 其他的事件

方法名称	描述
onEnter()	当此节点所在场景即将呈现时，会调用此方法
onEnterTransitionDidFinish()	当此节点所在场景的入场动作结束后，会调用此方法。如果所在场景没有入场动作，则此方法会紧接着 onEnter()后被调用
onExit()	当此节点所在场景即将退出时，会调用此方法
onExitTransitionDidStart()	当此节点所在场景的出场动作结束后，会调用此方法。如果所在场景没有出场动作，则此方法会紧接着 onExit()后被调用

■ 这些事件的默认实现通常负责处理定时器和动作的启用与暂停，因此必须在重载方法中调用父类的方法。例如，我们可以在场景开始时设置游戏的背景音乐：

```
void GameScene::onEnter()
{
    CCScene::onEnter();
    this->playBackgroundMusic();
}
```

- 3.6 Cocos2d-x 内置的常用层（1）

■ 3.6 Cocos2d-x 内置的常用层（1）

■ 为了方便游戏开发者，Cocos2d-x 内置了 3 种特殊的 CCLayer，具体如下所示。

■ CCLayerColor：一个单纯的实心色块。

■ CCLayerGradient：一个色块，但可以设置两种颜色的渐变效果。

■ CCMenu：十分常用的游戏菜单。

■ 在这一节中，我们将详细介绍这 3 种常用的层。

■ CCLayerColor 与 CCLayerGradient

■ 这两个层十分简单，都仅仅包含一个色块。不同的是，前者创建的是一个实色色块，而后者创建的是一个渐变色块。图 3-7 展示了 Cocos2d-x 测试样例中 CCLayerColor 与 CCLayerGradient 的效果。

■

■ 图3-7 CCLayerColor与CCLayerGradient

■ CCLayerColor 拥有以下初始化方法：如果采用指定了宽与高的初始化方法，则创建一个指定大小的色块；如果采用不指定大小的初始化方法，则创建一个屏幕大小的色块。CCLayerColor 的创建方法和初始化方法如下所示：

```
static CCLayerColor * create(const ccColor4B& color);
static CCLayerColor * create(const ccColor4B& color, GLfloat width, GLfloat height);
bool initWithColor(const ccColor4B& color);
bool initWithColor(const ccColor4B& color, GLfloat width, GLfloat height);
```

■ CCLayerGradient 与 CCLayerColor 类似，但是它在初始化时需要指定两种颜色以及渐变的方向。在初始化方法中，start 参数为起始颜色，end 参数为结束颜色，而 v 是方向向量。CCLayerGradient 的创建方法和初始化方法如下所示：

```
static CCLayerGradient* create(const ccColor4B& start, const ccColor4B& end);
static CCLayerGradient* create(const ccColor4B& start, const ccColor4B& end,
                        const CCPoint& v);
bool initWithColor(const ccColor4B& start, const ccColor4B& end);
bool initWithColor(const ccColor4B& start, const ccColor4B& end, const CCPoint& v);
```

■ 在色块创建后，也可以通过下面列举的方法来修改色块大小：

```
void changeWidth(GLfloat w);
void changeHeight(GLfloat h);
void changeWidthAndHeight(GLfloat w ,GLfloat h);
```

- **CCMenu：游戏菜单**
- 菜单是游戏不可或缺的一部分。在 Cocos2d-x 中，菜单由两部分组成，分别是菜单项和菜单本身。CCMenuItem 表示一个菜单项，每个菜单项都是一个独立的按钮，定义了菜单的视觉表现和响应动作；CCMenu 则是菜单，它负责将菜单项组织到一起并添加到场景中，转换屏幕的触摸事件到各个菜单项。
- 菜单项的创建通常需要规定普通状态、按下状态和被点击后的响应对象以及响应方法。下面我们以一个图片菜单项为例介绍一下菜单项的创建，相关代码如下：

```
CCMenuItemImage::create(
    "CloseNormal.png",      //普通状态下的图片
    "CloseSelected.png",    //按下状态下的图片
    this,                   //响应对象
    menu_selector(HelloWorld::menuCloseCallback));  //响应函数
```

- 其中响应函数必须满足 SEL_MenuHandler 形式：返回值为空，带一个 CCNode*型的参数。
- 下面我们为《捕鱼达人》添加开始菜单和游戏画面，同时也演示场景切换和层的使用。首先，我们建立一个标题画面的场景。在这个场景上，存在一个"开始游戏"按钮，点击该按钮将切换到游戏的主画面。为此，我们创建一个继承自 CCScene 的 StartScene 类，其头文件的代码如下：

```
class StartScene : public CCScene
{
public:
    SCENE_NODE_FUNC(StartScene);
    bool init();
    void startGame(CCNode *sender);
};
```


- **3.6 Cocos2d-x 内置的常用层（2）**
- 现在我们来实现 bool init()方法。我们需要在场景中加入一个按钮，最简单的方法就是创建一个菜单，在菜单中添加一个菜单项供用户点击，再把菜单添加到场景之中。为此，我们首先创建一个菜单项，设置好它的相关属性与响应函数，然后创建一个包含此菜单项的菜单，把它加入到场景之中。init 方法的实现代码如下：

```
bool StartScene::init()
{
    bool bRet = false;
    do {
        CC_BREAK_IF(! CCScene::init());
        CCSize winSize = CCDirector::sharedDirector()->getWinSize();

        CCMenuItem *startGameItem = CCMenuItemFont::
            create("Start", this, menu_selector(StartScene::startGame));
        CC_BREAK_IF(!startGameItem);
        startGameItem->setPosition(ccp(winSize.width / 2, winSize.height / 2));

        CCMenu* menu = CCMenu::create(startGameItem, NULL);
        CC_BREAK_IF(!menu);
```

```
        menu->setPosition(CCPointZero);

        this->addChild(menu);
        bRet = true;
    } while (0);

   return bRet;
}
```

最后，我们在响应函数里添加场景切换的操作，以实现玩家点击按钮后进入游戏场景的效果。在切换的过程中，我们使用了一个放大切换特效：

```
void StartScene::startGame(CCNode *sender)
{
    CCDirector::sharedDirector()->replaceScene(CCTransitionShrinkGrow::
        create(1.2f, GameScene::node()));
}
```

游戏场景的代码在此就略去了，它和开始界面是大同小异的，特别想提到的是暂停和恢复游戏按钮的响应函数。正如前面所说，暂停和恢复游戏可以由 CCDirector 这个游戏大总管来完成，相关代码如下：

```
void GameMenuLayer::pauseGame(CCObject* sender)
{
    CCDirector::sharedDirector()->pause();
    resumeGameItem->setIsVisible(true);
    pauseGameItem->setIsVisible(false);
}

void GameMenuLayer::resumeGame(CCObject* sender)
{
    CCDirector::sharedDirector()->resume();
    resumeGameItem->setIsVisible(false);
    pauseGameItem->setIsVisible(true);
}
```

完成这两个方法之后，启动我们的游戏。在游戏中点击按钮，我们会发现屏幕上显示的帧率下降到 4 帧/秒了。此时游戏已经暂停，所有的动作都会停止，并降低帧率以响应触摸事件。现在我们的游戏中还没有加入动态的内容，因此即使暂停下来也不会看到其他明显的现象。在后续章节我们为游戏增加了动态效果后，再点击此按钮，所有的动作就会暂停下来。直到我们再次点击按钮，游戏才会继续运行。

3.7 Cocos2d-x 调度原理

3.7 Cocos2d-x 调度原理

Cocos2d 的一大特色就是提供了事件驱动的游戏框架，引擎会在合适的时候调用事件处理函数，我们只需要在函数中添加对各种游戏事件的处理，就可以完成一个完整的游戏了。例如，为了实现游戏的动态变化，Cocos2d 提供了两种定时器事件；为了响应用户输入，Cocos2d 提供了触摸事件和传感器事件；此外，Cocos2d 还提供了一系列控制程序生命周期的事件。

- Cocos2d 的调度原理管理着所有的事件。为了深入理解引擎，我们将在这一节中详细介绍引擎的调度原理。Cocos2d 的调度原理虽不复杂，但由于它所涉及的部分很多，这里我们仅介绍引擎的框架与计时器调度原理，其余部分将在后面的章节中陆续介绍。
- 请注意，这一节将会深入分析引擎的内部原理，因此初学者读起来也许会略显吃力。建议初学者跳过本节，对 Cocos2d-x 游戏开发有了一个完整的认识后，再来详细研究它。

● 3.7.1 游戏主循环（1）

- 3.7.1 游戏主循环（1）

- 在介绍游戏基本概念的时候，我们曾介绍了场景、层、精灵等游戏元素，但我们却故意避开了另一个同样重要的概念，那就是游戏主循环，这是因为 Cocos2d 已经为我们隐藏了游戏主循环的实现。读者一定会对主循环的作用有疑问，为了解答这个问题，我们首先来讨论游戏实现的原理。
- 游戏乃至图形界面的本质是不断地绘图，然而绘图并不是随意的，任何游戏都需要遵循一定的规则来呈现出来，这些规则就体现为游戏逻辑。游戏逻辑会控制游戏内容，使其根据用户输入和时间流逝而改变。因此，游戏可以抽象为不断地重复以下动作：
- 处理用户输入
- 处理定时事件
- 绘图
- 游戏主循环就是这样的一个循环，它会反复执行以上动作，保持游戏进行下去，直到玩家退出游戏。在 Cocos2d 中，以上的动作包含在 CCDirector 的某个方法之中，而引擎会根据不同的平台设法使系统不断地调用这个方法，从而完成了游戏主循环。
- 现在我们回到 Cocos2d-x 游戏主循环的话题上来。上面介绍了 CCDirector 包含一个管理引擎逻辑的方法，它就是 CCDirector::mainLoop()方法，这个方法负责调用定时器，绘图，发送全局通知，并处理内存回收池。该方法按帧调用，每帧调用一次，而帧间间隔取决于两个因素，一个是预设的帧率，默认为 60 帧每秒；另一个是每帧的计算量大小。当逻辑处理与绘图计算量过大时，设备无法完成每秒 60 次绘制，此时帧率就会降低。
- mainLoop()方法会被定时调用，然而在不同的平台下它的调用者不同。通常 CCApplication 类负责处理平台相关的任务，其中就包含了对 mainLoop()的调用。有兴趣的读者可以对比 Android、iOS 与 Windows Phone 三个平台下不同的实现，平台相关的代码位于引擎的"platform"目录。
- mainLoop()方法是定义在 CCDirector 中的抽象方法，它的实现位于同一个文件中的 CCDisplayLinkDirector 类。现在我们来看一下它的代码：

```
void CCDisplayLinkDirector::mainLoop()
{
    if (m_bPurgeDirecotorInNextLoop)
    {
        m_bPurgeDirecotorInNextLoop = false;
        purgeDirector();
    }
    else if (! m_bInvalid)
    {
        drawScene();
        //释放对象
```

```
            CCPoolManager::sharedPoolManager()->pop();
        }
    }
```

- 上述代码主要包含如下 3 个步骤。

- 判断是否需要释放 CCDirector，如果需要，则删除 CCDirector 占用的资源。通常，游戏结束时才会执行这个步骤。
- 调用 drawScene()方法，绘制当前场景并进行其他必要的处理。
- 弹出自动回收池，使得这一帧被放入自动回收池的对象全部释放。

- 由此可见，mainLoop()把内存管理以外的操作都交给了 drawScene()方法，因此关键的步骤都在 drawScene()方法之中。下面是 drawScene()方法的实现：

```
void CCDirector::drawScene()
{
    //计算全局帧间时间差 dt
    calculateDeltaTime();

    if (! m_bPaused)
    {
        m_pScheduler->update(m_fDeltaTime);
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    if (m_pNextScene)
    {
        setNextScene();
    }

    kmGLPushMatrix();

    //绘制场景
    if (m_pRunningScene)
    {
        m_pRunningScene->visit();
    }

    //处理通知节点
    if (m_pNotificationNode)
    {
        m_pNotificationNode->visit();
    }

    if (m_bDisplayStats)
    {
        showStats();
```

```
    }

    if (m_pWatcherFun && m_pWatcherSender)
    {
        (*m_pWatcherFun)(m_pWatcherSender);
    }

    kmGLPopMatrix();

    m_uTotalFrames++;

    //交换缓冲区
    if (m_pobOpenGLView)
    {
        m_pobOpenGLView->swapBuffers();
    }

    if (m_bDisplayStats)
    {
        calculateMPF();
    }
}
```

3.7.1 游戏主循环（2）

3.7.1 游戏主循环（2）

我们看到 drawScene()方法内进行了许多操作，甚至包含了少量 OpenGL 函数。这是由于 Cocos2d-x 在游戏主循环中对引擎的细节进行了许多处理，我们并不关心这些细节，因此我们首先剔除掉细枝末节，整理出一个精简版本的 drawScene()方法：

```
void CCDirector::drawSceneSimplified()
{
    _calculate_time();

    if (! m_bPaused)
        m_pScheduler->update(m_fDeltaTime);

    if (m_pNextScene)
        setNextScene();

    _deal_with_opengl();

    if (m_pRunningScene)
        m_pRunningScene->visit();

    _do_other_things();
}
```

- 对比一下 drawSceneSimplified()与 drawScene()的代码，可以发现我们省略掉的代码主要用于处理 OpenGL 和一些细节，如计算 FPS、帧间时间差等。在主循环中，我们主要进行了以下 3 个操作。

- 调用了定时调度器的 update 方法，引发定时器事件。
- 如果场景需要被切换，则调用 setNextStage 方法，在显示场景前切换场景。
- 调用当前场景的 visit 方法，绘制当前场景。

- 场景的绘制与 OpenGL 密切相关，因此我们会在 10.1 节中详细讨论。在 3.7.2 节中，我们将探究定时器的工作原理。

● 3.7.2 定时调度器（1）

- **3.7.2 定时调度器（1）**
- 在游戏主循环 drawScene 方法中，我们可以看到每一帧引擎都会调用 m_pScheduler 的 update 方法。
- m_pScheduler 是 CCScheduler 类型的对象，是一个定时调度器。所谓定时调度器，就是一个管理所有节点定时器的对象，它负责记录定时器，并在合适的时间触发定时事件。接下来我们详细介绍定时调度器。
- **从 CCNode 说起**
- 前面我们简要介绍了游戏主循环，并在 Cocos2d-x 的游戏主循环中引出了定时调度器 CCScheduler 的调度方法 update。update 方法主要负责定时器的调度，我们将对它进行详细分析，但在此之前，了解 CCScheduler 公开的接口是很有必要的，这会有助于我们对调度器类有一个整体的认识。因此，我们从 CCNode 的定时器接口开始分析。
- 在 3.5 节中，我们已经介绍了它提供的定时器功能，在此先做一个简单的复习。Cocos2d-x 提供了两种定时器，分别是：

- update 定时器，每一帧都被触发，使用 scheduleUpdate 方法来启用；
- schedule 定时器，可以设置触发的间隔，使用 schedule 方法来启用。

- 下面就是这两个方法的代码：

```
void CCNode::scheduleUpdateWithPriority(int priority)
{
    m_pScheduler->scheduleUpdateForTarget(this, priority, !m_bIsRunning);
}

void CCNode::schedule(SEL_SCHEDULE selector, float interval, unsigned int repeat,
    float delay)
{
    CCAssert( selector, "Argument must be non-nil");
    CCAssert( interval >=0, "Argument must be positive");

    m_pScheduler->scheduleSelector(selector, this, interval, !m_bIsRunning,
        repeat, delay);
}
```

- 其中 m_pScheduler 是 CCScheduler 对象。可以看到，这两个方法的内部除去检查参数是否合法，只是调用了 CCScheduler 提供的方法。换句话说，CCNode 提供的定时器只是对 CCScheduler 的包装而已。不仅这两个方法如此，其他定时器相关的方法也都是这样。这里没有必要列出所有的代码，富有探索精神的读者可以打开 CCNode 的源代码，对照 3.5.3 节查看每一个相关方法的代码。

■ **CCScheduler 成员**

■ 经过上面的分析，我们已经知道 CCNode 提供的定时器不是由它本身而是由 CCScheduler 管理的。因此，我们把注意力转移到定时调度器上。显而易见，定时调度器应该对每一个节点维护一个定时器列表，在恰当的时候就会触发其定时事件。打开 CCScheduler 类的头文件，可以看到它主要包含表 3-6 所示的成员。

■ 表 3-6 CCScheduler 主要成员

● 类型	● 名称	● 描述
● 方法	● scheduleSelector	● 为指定目标设置一个定时器
	● unscheduleSelector	● 取消指定目标的定时器
	● unscheduleAllSelectorsForTarget	● 取消指定目标的所有定时器 ● （包含普通定时器与 update 定时器）
	● unscheduleAllSelectors	● 取消所有被 CCScheduler 管 ● 理的定时器，包括 update 定时器
	● scheduleUpdateForTarget	● 启用指定目标的 update 定时器
	● unscheduleUpdateForTarget	● 取消指定目标的 update 定时器
	● pauseTarge	● 暂停指定目标的全部定时器
	● resumeTarget	● 恢复指定目标的全部定时器
	● isTargetPaused	● 返回一个值，表示目标是否被暂停
	● pauseAllTargets	● 暂停所有被 CCScheduler 管理的目标
● 私有字段	● m_pUpdatesNegList	● 一个链表，记录优先值小于 0 的 update 定时器
	● m_pUpdates0List	● 一个链表，记录优先值为 0 的 update 定时器
	● m_pUpdatesPosList	● 一个链表，记录优先值大于 0 的 update 定时器
	● m_pHashForUpdates	● 记录全部 update 定时器的散列表， ● 便于调度器检索定时器
	● m_pHashForSelectors	● 记录普通定时器的散列表
●	●	●

■ 为了注册一个定时器，开发者只要调用调度器提供的方法即可。同时调度器还提供了一系列对定时器的控制接口，例如暂停和恢复定时器。在调度器内部维护了多个容器，用于记录每个节点注册的定时器；同时，调度器会接受其他组件（通常与平台相关，例如在 iOS 下为 CADisplayLink）的定时调用，随着系统时间的改变驱动调度器。

■ 调度器可以随时增删或修改被注册的定时器。具体来看，调度器将 update 定时器与普通定时器分别处理：当某个节点注册 update 定时器时，调度器就会把节点添加到 Updates 容器中，为了提高调度器效率，Cocos2d-x 使用了散列表与链表结合的方式来保存定时器信息；当某个节点注册普通定时器时，调度器会把回调函数和其他信息保存到 Selectors 散列表中。

■ **3.7.2 定时调度器（2）**

■ **update 方法**

■ 在游戏主循环中，我们已经见到了 update 方法。可以看到，游戏主循环会不停地调用 update 方法。该方法包含一个实型参数，表示两次调用的时间间隔。在该方法中，引擎会利用两次调用的间隔来计算何时触发定时器。

■ update 方法的实现看起来较为复杂，而实际上它的内部多是重复的代码片段，逻辑并不复杂。我们可以利用 Cocos2d-x 中精心编写的注释来帮助理解 update 方法的工作流程，相关代码如下：

```
void CCScheduler::update(float dt)
{
    m_bUpdateHashLocked = true;

    //a.预处理
    if (m_fTimeScale != 1.0f)
    dt *= m_fTimeScale;

    //b.枚举所有的 update 定时器
    tListEntry *pEntry, *pTmp;

    //优先级小于 0 的定时器
    DL_FOREACH_SAFE(m_pUpdatesNegList, pEntry, pTmp)
    if ((! pEntry->paused) && (! pEntry->markedForDeletion))
        pEntry->target->update(dt);

    //优先级等于 0 的定时器
    DL_FOREACH_SAFE(m_pUpdates0List, pEntry, pTmp)
    if ((! pEntry->paused) && (! pEntry->markedForDeletion))
        pEntry->target->update(dt);

    //优先级大于 0 的定时器
    DL_FOREACH_SAFE(m_pUpdatesPosList, pEntry, pTmp)
    if ((! pEntry->paused) && (! pEntry->markedForDeletion))
        pEntry->target->update(dt);

    //c.枚举所有的普通定时器
    for (tHashSelectorEntry *elt = m_pHashForSelectors; elt != NULL; )
    {
        m_pCurrentTarget = elt;
        m_bCurrentTargetSalvaged = false;

        if (! m_pCurrentTarget->paused)
        {
            //枚举此节点中的所有定时器
            //timers 数组可能在循环中改变，因此在此处需要小心处理
            for (elt->timerIndex = 0; elt->timerIndex < elt->timers->num;
                ++(elt->timerIndex))
            {
                elt->currentTimer = (CCTimer*)(elt->timers->arr[elt->timerIndex]);
                elt->currentTimerSalvaged = false;

                elt->currentTimer->update(dt);

                if (elt->currentTimerSalvaged)
```

```
                {
                    elt->currentTimer->release();
                }

                elt->currentTimer = NULL;
            }
        }

        elt = (tHashSelectorEntry *)elt->hh.next;

        if (m_bCurrentTargetSalvaged && m_pCurrentTarget->timers->num == 0)
        {
            removeHashElement(m_pCurrentTarget);
        }
    }

    //d.处理脚本引擎相关的事件
    if (m_pScriptHandlerEntries)
    {
        for (int i = m_pScriptHandlerEntries->count() - 1; i >= 0; i--)
        {
            CCSchedulerScriptHandlerEntry* pEntry =
                static_cast<CCSchedulerScriptHandlerEntry*>(m_pScriptHandlerEntries
                ->objectAtIndex(i));
            if (pEntry->isMarkedForDeletion())
            {
                m_pScriptHandlerEntries->removeObjectAtIndex(i);
            }
            else if (!pEntry->isPaused())
            {
                pEntry->getTimer()->update(dt);
            }
        }
    }

    //e.清理所有被标记了删除记号的 update 方法
    //优先级小于 0 的定时器
    DL_FOREACH_SAFE(m_pUpdatesNegList, pEntry, pTmp)
    {
        if (pEntry->markedForDeletion)
        {
            this->removeUpdateFromHash(pEntry);
        }
    }

```

```
    //优先级等于 0 的定时器
    DL_FOREACH_SAFE(m_pUpdates0List, pEntry, pTmp)
    {
        if (pEntry->markedForDeletion)
        {
            this->removeUpdateFromHash(pEntry);
        }
    }

    //优先级大于 0 的定时器
    DL_FOREACH_SAFE(m_pUpdatesPosList, pEntry, pTmp)
    {
        if (pEntry->markedForDeletion)
        {
            this->removeUpdateFromHash(pEntry);
        }
    }

    m_bUpdateHashLocked = false;

    m_pCurrentTarget = NULL;
}
```

3.7.2 定时调度器（3）

3.7.2 定时调度器（3）

借助注释，能够看出 update 方法的流程大致如下所示。

- 参数 dt 乘以一个缩放系数，以改变游戏全局的速度，其中缩放系数可以由 CCScheduler 的 TimeScale 属性设置。
- 分别枚举优先级小于 0、等于 0、大于 0 的 update 定时器。如果定时器没有暂停，也没有被标记为即将删除，则触发定时器。
- 枚举所有注册过普通定时器的节点，再枚举该节点的定时器，调用定时器的更新方法，从而决定是否触发该定时器。
- 我们暂不关心脚本引擎相关的处理。
- 再次枚举优先级小于 0、等于 0、大于 0 的 update 定时器，移除前几个步骤中被标记了删除记号的定时器。

对于 update 定时器来说，每一节点只可能注册一个定时器，因此调度器中存储定时器数据的结构体_listEntry 主要保存了注册者与优先级。对于普通定时器来说，每一个节点可以注册多个定时器，引擎使用回调函数（选择器）来区分同一节点下注册的不同定时器。调度器为每一个定时器创建了一个 CCTimer 对象，它记录了定时器的目标、回调函数、触发周期、重复触发还是仅触发一次等属性。

CCTimer 也提供了 update 方法，它的名字和参数都与 CCScheduler 的 update 方法一样，而且它们也都需要被定时调用。不同的是，CCTimer 的 update 方法会把每一次调用时接收的时间间隔 dt 积累下来，如果经历的时间达到了周期，就会引发定时器的定时事件。第一次引发了定时事件后，如果是仅触发一次的定时器，则 update 方法会中止，否则定时器会重新计时，从而反复地触发定时事件。

回到 CCScheduler 的 update 方法上来。在步骤 c 中，程序首先枚举了每一个注册过定时器的对象，然后再枚举对象中定时器对应的 CCTimer 对象，调用 CCTimer 对象的 update 方法来更新定时器状态，以便触发定时事件。

- 至此，我们可以看到事件驱动的普通定时器调用顺序为：系统的时间事件驱动游戏主循环，游戏主循环调用 CCScheduler 的 update 方法，CCScheduler 调用普通定时器对应的 CCTimer 对象的 update 方法，CCTimer 类的 update 方法调用定时器对应的回调函数。对于 update 定时器，调用顺序更为简单，因此前面仅列出了普通定时器的调用顺序。
- 同时，我们也可以看到，在定时器被触发的时刻，CCScheduler 类的 update 方法正在迭代之中，开发者完全可能在定时器事件中启用或停止其他定时器（如图 3-8 所示）。不过，这么做会导致 update 方法中的迭代被破坏。Cocos2d-x 的设计已经考虑到了这个问题，采用了一些技巧避免迭代被破坏。例如，update 定时器被删除时，不会直接删除，而是标记为将要删除，在定时器迭代完毕后再清理被标记的定时器，这样即可保证迭代的正确性。

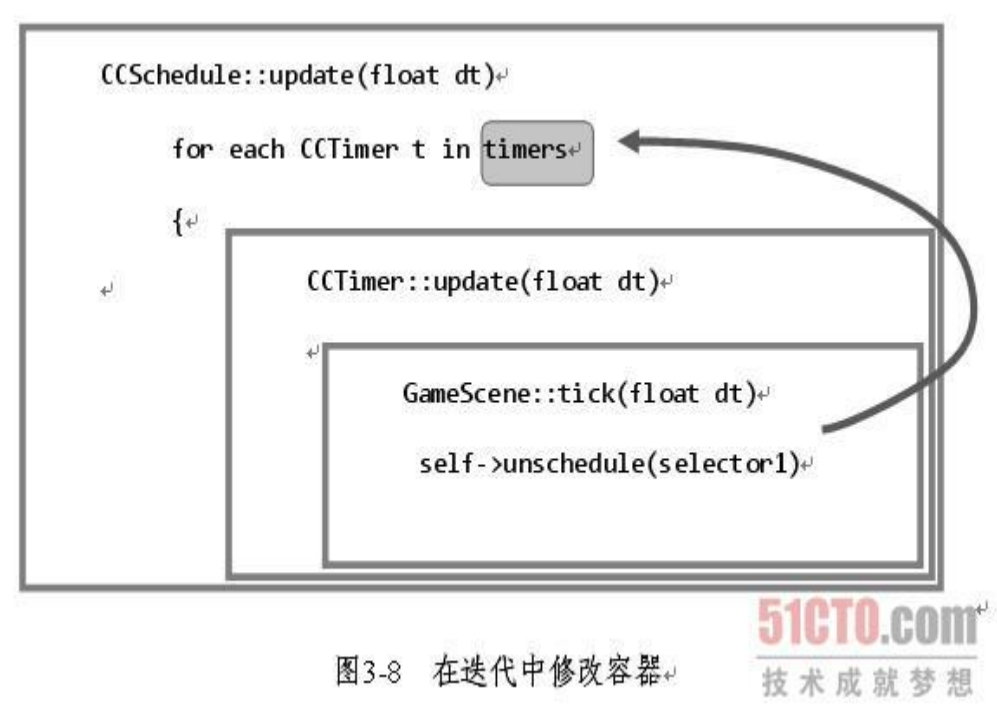


图3-8 在迭代中修改容器

- Cocos2d-x 的设计使得很多离散在各处的代码通过事件联系起来，在每一帧中起作用。基于事件驱动的游戏框架易于掌握，使用灵活，而且所有事件串行地在同一线程中执行，不会出现线程同步的问题。在后面更深入的讨论中，读者将会看到更多有趣的现象。

3.8 小结

3.8 小结

- 至此，我们了解了游戏中最基础的 3 种元素--场景、层和精灵，这些游戏元素不仅有各自的用途，还组成了渲染树中的节点，为 Cocos2d-x 提供了十分丰富的功能。这一章涉及的知识点十分繁杂，下面我们简单总结这一章的知识点。
- CCDirector：控制游戏流程的总管类，提供绘图方面的选项以及平台相关的功能。
- CCScene：场景的实现类，是层的容器。
- CCLayer：层的实现类，是精灵的容器。层还可以用来接受用户输入，例如触摸事件和加速度计事件。
- CCSprite：精灵的实现类，显示一个静态或动态的图形。图形在引擎中以纹理的形式存在，精灵不仅能显示完整的纹理，也可以显示部分纹理。
- CCNode：一切游戏元素的根类，也是构成渲染树的节点。它提供了基本的绘图属性，可以作为其他节点的容器，提供定时器的功能，可以执行动作。
- CCMenu：特殊的层，封装了游戏中常用的菜单功能。

- 定时器：分为 update 定时器与 schedule 定时器，前者每一帧触发一次，而后者可以指定触发间隔。定时器由定时调度器（CCScheduler）控制，每个定时器互不干扰，串行执行。
- 我们的第一个游戏场景已经开始实现，但目前的游戏还基本处于一个很简陋的半静止状态。在第 4 章中，我们将为游戏引入动作机制，让游戏"动"起来。

4.1 基本概念

- **动作**
  - 在第 3 章中，我们不仅详细介绍了游戏的基本组成元素--场景、层、精灵和渲染树等，也详细介绍了 CCNode 提供的定时器。利用定时器，可以不断修改节点的属性，实现简单的动态效果。然而，这种方法会导致为了实现简单的动态效果，十分烦琐地维护一批定时器。Cocos2d-x 为了解决这个问题，引入了动作机制。
  - 在这一章中，我们将为大家详细介绍各种动作的使用方法。读完本章后，读者将会学到《捕鱼达人》游戏中的基本动作机制：鱼的各种游动动作、炮弹的发射动作、渔网的展开动作，以及金币的收集等各种其他动作效果。
  - **4.1 基本概念**
  - CCAction 是动作类的基类，所有的动作都派生自这个类，它创建的一个对象代表了一个动作。动作作用于 CCNode，因此，任何一个动作都需要由 CCNode 对象来执行。以下代码实现了一条鱼用 1 秒钟的时间移动到了(0, 0)点：

```
CCSprite* sprite = CCSprite::create("fish.png");
CCAction* action = CCMoveTo::create(1.0f, ccp(0, 0));
sprite->runAction(action);
```

  - 值得注意的是，一个 CCAction 只能使用一次，这是因为动作对象不仅描述了动作，还保存了这个动作持续过程中不断改变的一些中间参数。对于需要反复使用的动作对象，可以通过 copy 方法复制使用。
  - CCAction 作为一个基类，其实质是一个接口（即抽象类），由它派生的实现类（如运动和转动等）才是我们实际使用的动作。CCAction 的绝大多数实现类都派生自 CCFiniteTimeAction，这个类定义了在有限时间内可以完成的动作。CCFiniteTimeAction 定义了 reverse 方法，通过这个方法可以获得一个与原动作相反的动作（称作逆动作），例如隐藏一个精灵后，用逆转动作再显示出来。当然，并非所有的动作都有对应的逆动作，例如类似"放大到"等设置属性为常量的动作不存在逆动作，而设置属性为相对值的动作则往往存在相应的逆动作。关于逆动作，后面将会看到更多演示。
  - 由 CCFiniteTimeAction 派生出的两个主要类分别是瞬时动作（CCActionInstant）和持续性动作（CCActionInterval），这两类动作与 4.4 节中介绍的复合动作配合使用，能得到复杂生动的动作效果。

4.2 瞬时动作

  - **4.2 瞬时动作**
  - 瞬时动作是指能立刻完成的动作，是 CCFiniteTimeAction 中动作持续时间为 0 的特例。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、设定缩放等。这些动作原本可以通过简单地对 CCNode 赋值完成，但是把它们包装为动作后，可以方便地与其他动作类组合为复杂动作。
  - 下面介绍一些常用的瞬时动作。
    - **CCPlace**
  - 该动作用于将节点放置到某个指定位置，其作用与修改节点的 Position 属性相同。例如，将鱼放置到屏幕坐标(100, 100)处，再执行曲线运动 curveMove 的代码如下：

```
CCFiniteTimeAction* placeAction = CCPlace::create(ccp(100, 100));
CCAction* action = CCSequence::create(placeAction, curveMove, NULL);
```

- 其中 CCSequence 又称为动作序列，是一种复合动作，它在初始化时，会接受多个动作，当它被执行时，这些动作会按顺序逐个执行，形成有序的一列动作。在 4.4 节中，我们将详细介绍复合动作。

◆ CCFlipX 和 CCFlipY

- 这两个动作分别用于将精灵沿 X 和 Y 轴反向显示，其作用与设置精灵的 FlipX 和 FlipY 属性相同。将其包装为动作类也是为了便于与其他动作进行组合。例如，我们想使鱼从屏幕的一端游动到另一端，然后按原路返回。为了更自然一点（虽然仍然不是很自然），我们设置一个序列，鱼先执行移动的动作，在鱼到达另一端时反向显示，然后再执行移动回起点的动作，相关代码如下：

```
CCFiniteTimeAction* flipXAction = CCFlipX::create(true);
CCAction* action = CCSequence::create(curveMove, flipXAction, curveMove->reverse(), NULL);
```

- 其中 reverse 的作用是取得原动作的逆动作。在这个例子中，鱼沿 X 轴翻转后将会沿原路返回起点。

◆ CCShow 和 CCHide

- 这两个动作分别用于显示和隐藏节点，其作用与设置节点的 Visible 属性的作用一样。例如，为了使鱼完成游动之后隐藏起来，我们使用如下代码：

```
CCFiniteTimeAction* hideAction = CCHide::create();
CCAction* action = CCSequence::create(curveMove, hideAction, NULL);
```

◆ CCCallFunc

- CCCallFunc 系列动作包括 CCCallFunc、CCCallFuncN、CCCallFuncND，以及 CCCall- FuncO 四个动作，用来在动作中进行方法的调用（之所以不是函数调用，是因为它们只能调用某个类中的实例方法，而不能调用普通的 C 函数）。当某个对象执行 CCCallFunc 系列动作时，就会调用一个先前被设置好的方法，以完成某些特别的功能。后面我们将举例说明它的用途。

- 在 CCCallFunc 系列动作的 4 个类中，CCCallFunc 调用的方法不包含参数，CCCallFuncN 调用的方法包含一个 CCNode*类型的参数，表示执行动作的对象。CCCallFuncND 调用的方法包含两个参数，不仅有一个节点参数，还有一个自定义参数（CCNode*与 void*）。CCCallFuncO 调用的方法则只包含一个 CCObject*类型的参数。

- 实际上，CCCallFunc 系列动作的后缀"N"表示 Node 参数，指的是执行动作的对象，"D"表示 Data 参数，指的是用户自定义的数据，"O"表示对象，指的是一个用户自定义的 CCObject 参数。在不同的情况下，我们可以根据不同的需求来选择不同的 CCCallFunc 动作。考虑一种情况，我们创建了许多会在屏幕中移动的精灵，希望精灵在移动结束之后就从游戏中删除。为了实现这个效果，我们可以创建一系列动作：首先让精灵移动，然后调用一个 removeSelf(CCNode* nodeToRemove)方法来删除 nodeToRemove 对象。在 removeSelf 方法中需要访问执行此动作的精灵，因此我们就采用 CCCallFuncN 来调用 removeSelf 方法。

- 在《捕鱼达人》中，当鱼游动到屏幕外之后，我们有必要将其从屏幕中除去。否则，鱼会在我们看不到的屏幕之外一直存在，导致内存上的浪费。我们一般会在其完成指定动作后，调用相应函数将其清除，相关代码如下：

```
CCFiniteTimeAction* actionMoveDone = CCCallFuncN::create(this,
    callfuncN_selector(GameScene::moveActionEnd));
CCAction* action = CCSequence::create(curveMove, actionMoveDone, NULL);
```

- 这样，我们就指定了一个动作序列，当执行了 curveMove 这个动作之后，再调用 actionMove- Done 函数。在 actionMoveDone 中，我们将完成动作的鱼从屏幕中移出。

● 4.3 持续性动作

- 4.3 持续性动作

- 持续性动作是在持续的一段时间里逐渐完成的动作，如精灵从一个点连续地移动到另一个点。与瞬时动作相比，持续性动作的种类更丰富。由于这些动作将持续一段时间，所以大多数的持续性动作都会带有一个用于控制动作执行时间的实型参数 duration。
- 每一种持续性动作通常都存在两个不同的变种动作，分别具有 To 和 By 后缀：后缀为 To 的动作描述了节点属性值的绝对变化，例如 CCMoveTo 将对象移动到一个特定的位置；而后缀为 By 的动作则描述了属性值相对的变化，如 CCMoveBy 将对象移动一段相对位移。
- 根据作用效果不同，可以将持续性动作划分为以下 4 大类：
- 位置变化动作
- 属性变化动作
- 视觉特效动作
- 控制动作
- 后面将简要介绍这 4 类动作。

● 4.3.1 位置变化动作

- 4.3.1 位置变化动作

- 针对位置（position）这一属性，引擎为我们提供了 3 种位置变化动作类型，下面将简要介绍这几种动作。
- CCMoveTo 和 CCMoveBy：用于使节点做直线运动。设置了动作时间和终点位置后，节点就会在规定时间内，从当前位置直线移动到设置的终点位置。它们的初始化方法分别为：

- `CCMoveTo::create(ccTime duration, CCPoint& pos);`
- `CCMoveBy::create(ccTime duration, CCPoint& pos);`

- 其中，duration 参数表示动作持续的时间，pos 参数表示移动的终点或距离。对于 CCMoveTo，节点会被移动到 pos 对应的位置；对于 CCMoveBy，节点会相对之前的位置移动 pos 的距离。
- CCJumpTo 和 CCJumpBy：使节点以一定的轨迹跳跃到指定位置。它们的初始化方法如下：

- `CCJumpTo::create(ccTime duration, CCPoint pos, float height, int jumps);`
- `CCJumpBy::create(ccTime duration, CCPoint pos, float height, int jumps);`

- 其中 pos 表示跳跃的终点或距离，height 表示最大高度，jumps 表示跳跃次数。
- CCBezierTo 和 CCBezierBy：使节点进行曲线运动，运动的轨迹由贝塞尔曲线描述。贝塞尔曲线是描述任意曲线的有力工具，在许多软件（如 Adobe Photoshop）中，钢笔工具就是贝塞尔曲线的应用。实际上，在《捕鱼达人》游戏中，为了控制鱼的游动，我们就用到了贝塞尔曲线。
- 每一条贝塞尔曲线都包含一个起点和一个终点。在一条曲线中，起点和终点都各自包含一个控制点，而控制点到端点的连线称作控制线。控制线决定了从端点发出的曲线的形状，包含角度和长度两个参数：角度决定了它所控制的曲线的方向，即这段曲线在这一控制点的切线方向；长度控制曲线的曲率。控制线越长，它所控制的曲线离控制线越近。示例图如图 4-1 所示。

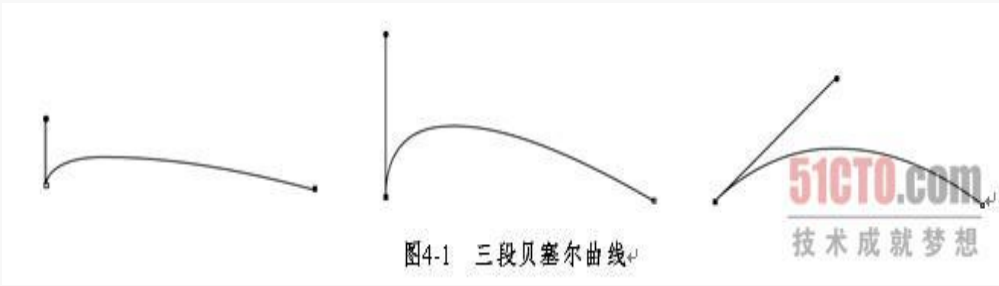


图4-1 三段贝塞尔曲线

任意一段曲线都可以由一段或几段相连的贝塞尔曲线组成，因此我们只需考虑一段贝塞尔曲线应该如何描述即可。一段独立的贝塞尔曲线如图 4-2 所示。

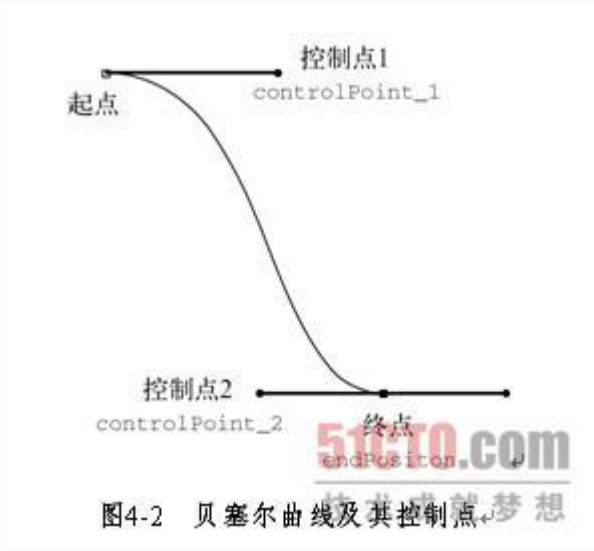


图4-2 贝塞尔曲线及其控制点

使用时我们要先创建 ccBezierConfig 结构体，设置好终点 endPosition 以及两个控制点 controlPoint_1 和 controlPoint_2 后，再把结构体传入 CCBezierTo 或 CCBezierBy 的初始化方法中：

```
ccBezierConfig bezier;
bezier.controlPoint_1 = ccp(20, 150);
bezier.controlPoint_2 = ccp(200, 30);
bezier.endPosition = ccp(160, 30);
CFiniteTimeAction * beizerAction = CCBezierTo::create(actualDuration / 4, bezier);
```

4.3.2 属性变化动作

4.3.2 属性变化动作

另一类动作是属性变化动作，它的特点是通过属性值的逐渐变化来实现动画效果。例如，下面要介绍的第一个动作 CCScaleTo，它会在一段时间内不断地改变游戏元素的 scale 属性，使属性值平滑地变化到一个新值，从而使游戏元素产生缩放的动画效果。

CCScaleTo 和 CCScaleBy：产生缩放效果，使节点的缩放系数随时间线性变化。对应的初始化方法为：

```
CCScaleTo::create(ccTime duration, float s);
CCScaleBy::create(ccTime duration, float s);
```

其中，s 为缩放系数的最终值或变化量。

CCRotateTo 和 CCRotateBy：产生旋转效果。对应的初始化方法为：

- `CCRotateTo::create(ccTime duration, float fDeltaAngle);`
- `CCRotateBy::create(ccTime duration, float fDeltaAngle);`

  - 其中 fDeltaAngle 的单位是角度，正方向为顺时针方向。
  - CCFadeIn 和 CCFadeOut：产生淡入淡出效果，其中前者实现了淡入效果，后者实现了淡出效果。对应的初始化方法为：

- `CCFadeIn::create(ccTime duration);`
- `CCFadeOut::create(ccTime duration);`

  - 这里需要说明的是，只有实现了 CCRGBAProtocol 接口的节点才可以执行这类动作，这是因为与透明度或颜色相关的属性都继承自 CCRGBAProtocol 接口。不过许多常见的节点，例如 CCSprite 与 CCLayerColor 等，都实现了 CCRGBAProtocol 接口，因此通常我们不必担心这个问题。
  - 以下介绍的几个动作也有类似的问题。
  - CCFadeTo：用于设置一段时间内透明度的变化效果。其初始化方法为：

- `CCFadeTo::create(ccTime duration, Glubyte opacity);`

  - 参数中的 Glubyte 是 8 位无符号整数，因此，opacity 可取 0 至 255 中的任意整数。与透明度相关的动作只能应用在精灵（CCSprite）上，且子节点不会受到父节点的影响。
  - CCTintTo 和 CCTintBy：设置色调变化。这个动作较为少用，其初始化方法为：

- `CCTintTo::create(ccTime duration,GLubyte r,Glubyte g,Glubyte b);`
- `CCTintBy::create(float duration, GLshort deltaRed, GLshort deltaGreen, GLshort deltaBlue);`

  - 与 CCFadeTo 类似，r、g 和 b 的取值范围也为 0～255。

● 4.3.3 视觉特效动作

  - 4.3.3 视觉特效动作

  - 这一类动作用于实现一些特殊的视觉效果，下面将简要介绍其中的两个动作。
  - CCBlink：使目标节点闪烁。其初始化方法为：

- `CCBlink::create(ccTime duration, unsigned int uBlicks);`

  - 其中，uBlicks 是闪烁次数。
  - CCAnimation：播放帧动画，用帧动画的形式实现动画效果，例如鱼的游动。在第 5 章中，我们将详细介绍它。

● 4.3.4 控制动作

  - 4.3.4 控制动作

  - 控制动作是一类特殊的动作，用于对一些列动作进行精细控制。利用这一类动作可以实现一些实用的功能，因此它们是十分常用的。这类动作包括 CCDelayTime、CCRepeat 和 CCRepeatForever 等。CCDelayTime 可以将动作延时一定的时间，CCRepeat 可以把现有的动作重复一定次数，CCRepeateForever 可以使一个动作不断重复下去。
  - 事实上，控制动作与复合动作息息相关，因此我们将在 4.4 节中对它们进行详细的讨论。

● 4.4 复合动作

  - 4.4 复合动作

- 前面介绍的简单动作显然不足以满足游戏开发的要求，在这些动作的基础上，Cocos2d-x 为我们提供了一套动作的复合机制，允许我们组合各种基本动作，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 CCNode 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，我们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。
- **重复（CCRepeat/CCRepeatForever）**
- 有的情况下，动作只需要执行一次即可，但我们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，如鱼的摆动、能量槽的转动，我们可以通过 CCRepeat 与 CCRepeatForever 这两个方式重复执行：

```
CCRepeat* CCRepeat::create(CCFiniteTimeAction *pAction, unsigned int times);
CCRepeatForever *CCRepeatForever::create(CCActionInterval *pAction);
```

- 在上述代码中，pAction 参数表示需要重复的动作，第一个方法允许指定动作的重复次数，第二个方法使节点一直重复该动作直到动作被停止。
- **并列（CCSpawn）**
- 指的是使一批动作同时执行。在《捕鱼达人》游戏中，鱼一边沿曲线游动一边摆尾巴，炮弹一边发射一边喷射气体，金币一边旋转一边移动等动作，都可以通过 CCSpawn 来实现。CCSpawn 从 CCActionInterval 派生而来的，它提供了两个工厂方法：

```
CCSpawn::create(CCFiniteTimeAction *pAction1,...);
CCSpawn::create(CCFiniteTimeAction *pAction1, CCFiniteTimeAction *pAction2);
```

- 其中第一个静态方法可以将多个动作同时并列执行，参数表中最后一个动作后需要紧跟 NULL 表示结束。第二个则只能指定两个动作复合，不需要在最后一个动作后紧跟 NULL。此外，执行的动作必须是能够同时执行的、继承自 CCFiniteTimeAction 的动作。组合后，CCSpawn 动作的最终完成时间由其成员中最大执行时间的动作来决定。
- **序列（CCSequence）**
- 除了让动作同时并列执行，我们更常遇到的情况是顺序执行一系列动作。CCSequence 提供了一个动作队列，它会顺序执行一系列动作，例如鱼游出屏幕外后需要调用回调函数，捕到鱼后显示金币数量，经过一段时间再让金币数量消失，等等。
- CCSequence 同样派生自 CCActionInterval。与 CCSpawn 一样，CCSquence 也提供了两个工厂方法：

```
CCSequence::create(CCFiniteTimeAction *pAction1,...);
CCSequence::create(CCFiniteTimeAction *pAction1,CCFiniteTimeAction *pAction2);
```

- 它们的作用分别是建立多个和两个动作的顺序执行的动作序列。同样要注意复合动作的使用条件，部分的非延时动作（如 CCRepeatForever）并不被支持。
- 在实现 CCSequence 和 CCSpawn 两个组合动作类时，有一个非常有趣的细节：成员变量中并没有定义一个可变长的容器来容纳每一个动作系列，而是定义了 m_pOne 和 m_pTwo 两个动作成员变量。如果我们创建了两个动作的组合，那么 m_pOne 与 m_pTwo 就分别是这两个动作本身；当我们创建更多动作的组合时，引擎会把动作分解为两部分来看待，其中后一部分只包含最后一个动作，而前一部分包含它之前的所有动作，引擎把 m_pTwo 设置为后一部分的动作，把 m_pOne 设置为其余所有动作的组合。例如，语句 sequence = CCSequence::create(action1, action2, action3, action4, NULL);就等价于：

```
CCSequence s1 = CCSequence::createWithTwoActions(action1, action2);
CCSequence s2 = CCSequence::createWithTwoActions(s1, action3);
sequence  = CCSequence::createWithTwoActions(s2, action4);
```

- CCSpawn 与 CCSequence 所采用的机制类似，在此就不再赘述了。采用这种递归的方式，而不是直接使用容器来定义组合动作，实际上为编程带来了极大的便利。维护多个动作的组合是一个复杂的问题，现在我们只需要考虑两个动作组合的情况就可以了。下面是 CCSpawn 的一个初始化方法，就是利用递归的思想简化了编程的复杂度：

```
CCFiniteTimeAction* CCSpawn::create(CCArray *arrayOfActions)
{
    CCFiniteTimeAction* prev = (CCFiniteTimeAction*)arrayOfActions->objectAtIndex(0);
    for (unsigned int i = 1; i < arrayOfActions->count(); ++i)
    {
        prev = create(prev, (CCFiniteTimeAction*)arrayOfActions->objectAtIndex(i));
    }

    return prev;
}
```

- 众所周知，递归往往会牺牲一些效率，但能换来代码的简洁。在这两个复合动作中，细节处理得十分优雅，所有的操作都只需要针对两个动作实施，多个动作的组合会被自动变换为递归
- 实现：

```
void CCSpawn::update(float time)
{
    if (m_pOne)
    {
        m_pOne->update(time);
    }
    if (m_pTwo)
    {
        m_pTwo->update(time);
    }
}

CCActionInterval* CCSpawn::reverse(void)
{
    return CCSpawn::create(m_pOne->reverse(), m_pTwo->reverse());
}
```

- **延时（CCDelayTime）**
- CCDelayTime 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。实际上，这与一个定时期实现的延迟没有区别，但相比之下，使用 CCDelayTime 动作来延时就可以方便地利用动作序列把一套动作连接在一起。CCDelayTime 只提供了一个工程方法，如下所示：

```
CCDelayTime::create(float d);
```

- 其中仅包含一个实型参数，表示动作占用的时间。

4.5 变速动作

- 4.5 变速动作

- 大部分动作的变化过程是与时间成线性关系的，即一个动作经过相同时间产生的变化相同，例如，CCMoveBy 会使节点在同样长的时间内经过同样的位移。这是因为 Cocos2d-x 把动作的速度变化控制抽离了出来，形成一个独立的机制。借助这个机制，我们可以很方便地实现诸如鱼的变速游动、金币的加速飞行以及后面将要介绍的动作平滑化等效果。普通动作配合变速动作，可以构造出很有趣的动作效果。
- 与复合动作类似，变速动作也是一种特殊的动作，它可以把任何一种动作按照改变后的速度执行。因此，在初始化变速动作时，需要传入一个动作。
- 变速动作包括 CCSpeed 动作与 CCEase 系列动作，下面我们详细介绍这些动作。
- CCSpeed
- CCSpeed 用于线性地改变某个动作的速度，因此，可以实现成倍地快放或慢放功能。为了改变一个动作的速度，首先需要将目标动作包装到 CCSpeed 动作中：

```
CCRepeatForever* repeat = CCRepeatForever::create(animation);
CCSpeed* speed = CCSpeed::create(repeat, 1.0f);
speed->setTag(action_speed_tag);
fish->runAction(speed);
```

- 在上面的代码中，我们创建了一个 animation 动作的 CCRepeatForever 复合动作 repeat，使动画被不断地重复执行。然后，我们又使用 repeat 动作创建了一个 CCSpeed 变速动作。create 初始化方法中的两个参数分别为目标动作与变速比率。设置变速比率为 1，目标动作的速度将不会改变。最后，我们为 speed 动作设置了一个 tag 属性，并把动作交给 fish 精灵，让精灵执行变速动作。此处设置的 tag 属性与 CCNode 的 tag 属性类似，用于从节点中方便地查找动作。
- 接下来，在需要改变速度的地方，我们通过修改变速动作的 speed 属性来改变动作速度。下面的代码将会把上面设置的动画速度变为原来的两倍：

```
CCSpeed * speed = fish->getActionByTag(action_speed_tag);
speed->setSpeed(2.0f);
```

- CCActionEase
- 虽然使用 CCSpeed 能够改变动作的速度，然而它只能按比例改变目标动作的速度。如果我们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。下面将要介绍的 CCActionEase 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。
- CCActionEase 系列包含 15 个动作，它们可以被概括为 5 类动作：指数缓冲、Sine 缓冲、弹性缓冲、跳跃缓冲和回震缓冲。每一类动作都有 3 个不同时期的变换：In、Out 和 InOut。下面使用时间变换图像表示每组 CCActionEase 动作的作用效果，其中横坐标表示实际动画时间，纵坐标表示变换后的动画时间。因此，线性动作的图像应该是一条自左下角到右上角的直线。图 4-3 展示了 Cocos2d-x 的各种变速曲线。

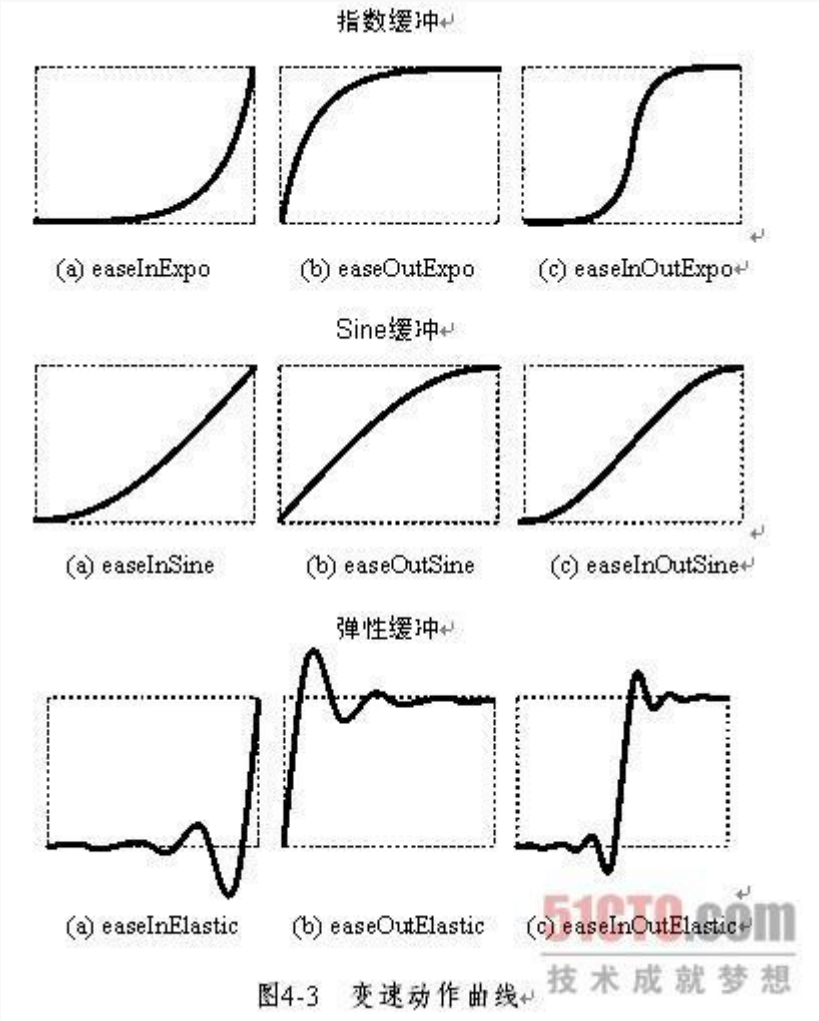


图4-3 变速动作曲线

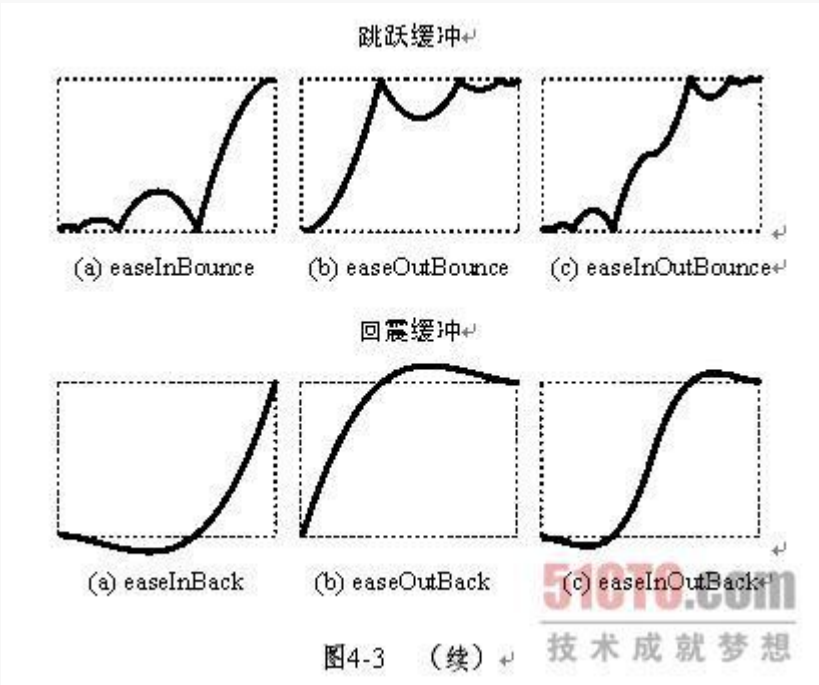


图4-3 （续）

- CCActionEase 的使用方法与 CCSpeed 类似。以 Sine 缓冲为例，以下代码实现了 InSine 变速运动：

```
CCEaseSineIn* sineIn = CCEaseSineIn::create(action);
```

```
sineIn->setTag(action_sine_in_tag);
fish->runAction(sineIn);
```


4.6 使鱼动起来（1）

学习了前面几节，我们已经了解了如何创建并使用 Cocos2d-x 提供的动作。万事俱备，只欠东风，现在我们终于可以着手为鱼添加动作，让它们动起来了。

为了实现鱼的游动，我们在鱼所在的动作层中添加了下面的 5 个方法。当我们调用 update- FishMovement 方法后，就会为全场的鱼随机创建一个游动动作：

```
void updateFishMovement();  //更新所有鱼的游动路径
void setFishRoute(CCSprite* fish,ccTime dt);    //随机设定一条鱼的路线
void linerRoute(CCSprite* fish,CCPoint from,CCPoint to,ccTime dt);//设置一条鱼沿直线游动
void bezierRoute(CCSprite* fish,CCPoint from,CCPoint to,ccTime dt);//设置一条鱼沿曲线游动
void moveActionEnd(CCNode* sender);  //回调函数，用于在游动动作结束后清理精灵
```

在 updateFishMovement 方法中，我们使用宏 CCARRAY_FOREACH 遍历_fishes 中的所有鱼，把鱼传入 setFishRoute 方法中，来更新每一条鱼的路径信息，相关代码如下：

```
void SpriteLayer::updateFishMovement()
{
    CCSprite* fish = NULL;
    CCARRAY_FOREACH(_fishes, fish) {
        this->setFishRoute(fish, fishUpdateInterval);
    }
}
```

其中 setFishRoute 方法会随机为鱼分配一个位置，并随机设置游动路线。在这个方法中，我们首先随机生成两个 0 或 1 的整数 direction 与 route，它们决定了鱼游动的轨迹以及方向。程序根据 route 的值选择鱼沿直线游动还是沿曲线游动，如果 route 为 0，则调用 linerRoute 方法给鱼添加直线游动的动作，否则调用 bezierRoute 方法给鱼添加曲线游动的动作。在这两个方法中，鱼的初始位置与终点位置的高度会被随机设置，相关代码如下：

```
void SpriteLayer::setFishRoute(CCSprite* fish, ccTime dt)
{
    int direction = abs(rand()%2),route = abs(rand()%2); //随机设置路线和方向
    CCSize winSize = CCDirector::sharedDirector()->getWinSize();

    if(route == 0) {
        linerRoute(fish,
            ccp(-10, rand() % winSize.height),
            ccp(winSize.width + 10, rand() % winSize.height),
            dt, direction);
    }
    else {
        bezierRoute(fish,
            ccp(-10, rand() % winSize.height),
            ccp(winSize.width + 10, rand() % winSize.height),
            dt,direction);
```

```
    }
}
```

linerRoute 函数比较简单，它使用 CCMoveTo 即可实现鱼的直线游动。在这个方法中，我们首先根据 direction 来设置鱼的左右方向，把鱼放置到初始位置，然后创建一个 CCMoveTo 动作把鱼移动到终止位置，接着创建一个 CCCallFuncN 动作，调用 moveActionEnd 方法，最后使用 CCSequence 组合两个动作，把最终得到的动作添加为鱼的动作。相关代码如下：

```
void SpriteLayer::linerRoute(CCSprite* fish, CCPoint from, CCPoint to, ccTime dt,
    bool direction)
{
    fish->setFlipX(direction);
    fish->setPosition(from);

    CCMoveTo* move = CCMoveTo::create(dt, to);
    CCCallFuncN* end = CCCallFuncN::create(this,
        callfuncN_selector(SpriteLayer::moveActionEnd));
    CCAction* action = CCSequence::create(move, end, NULL);

    fish->runAction(action);
}
```

● 4.6 使鱼动起来（2）

4.6 使鱼动起来（2）

bezierRoute 方法用于实现曲线游动。为了操作简单，我们仅组合两个贝塞尔曲线动作。在设置曲线参数时，为了使两个阶段的曲线运动平滑衔接，我们需要仔细地设置两段动作的参数：第一段曲线 b1 的终点与第二段曲线 b2 的起点重合；b1 的终点处控制线与 b2 的起点处控制线位于同一直线，且长度相等。具体的曲线参数示意图如图 4-4 所示。

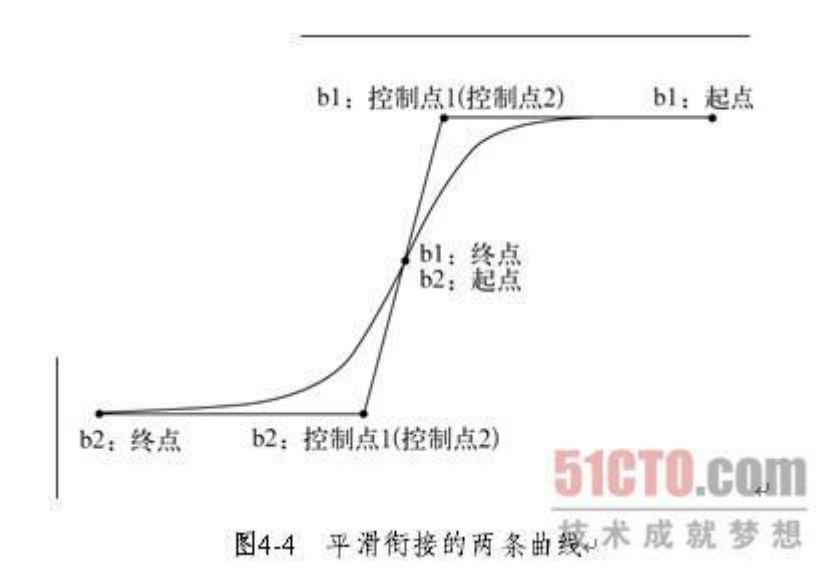


图4-4 平滑衔接的两条曲线

在 bezierRoute 方法中，我们按照方向为鱼创建游动路径。路径由 4 部分组成，分别为直线、两条贝塞尔曲线和直线。每条路径首尾相接，完成了一个呈 S 形的游动轨迹。为此，我们创建了两个 CCMoveTo 动作与两个 CCBezierTo 动作，使用 CCSequence 组合执行 4 个动作。第一个 CCMoveTo 引导鱼横向游动，紧接着第一个 CCBezierTo 引导鱼沿弧线游动到屏幕中

央，第二个 CCBezierTo 引导鱼沿弧线游动到对称的方向，最后第二个 CCMoveTo 引导鱼横向游出屏幕。代码中 h 数组用于保存各个坐标的高度，r 用于保存临时参数，以便计算贝塞尔曲线的各个参数。 bezierRoute 方法的代码如下：

```
void SpriteLayer::bezierRoute(CCSprite* fish, CCPoint from, CCPoint to,
    ccTime dt, bool direction)
{
    CCSize s = CCDirector::sharedDirector()->getWinSize();
    float h[] = { 55.0f, s.height - 10.0f };
    float r[] = { (h[1] - h[0]) / 2, -(h[1] - h[0]) / 2 };

    fish->setFlipX(direction);

    ccBezierConfig b1,b2;

    b1b1.controlPoint_1 = b1.controlPoint_2 = ccp(s.width / 2 +
        r[!direction] / 3, h[direction]);
    b1.endPosition = ccp(s.width / 2, h[0] + r[0]);

    b2b2.controlPoint_1 = b2.controlPoint_2 = ccp(s.width / 2 +
        r[direction] / 3, h[!direction]);
    b2.endPosition = ccp(s.width / 2 + r[direction], h[!direction]);

    CCFiniteTimeAction* move = CCSequence::create(
        CCMoveTo::create(dt / 4, ccp(s.width / 2 + r[!direction], h[direction])),
        CCBezierTo::create(dt / 4, b1),
        CCBezierTo::create(dt / 4, b2),
        CCMoveTo::create(dt / 4, to),
        NULL);

    CCCallFuncN* end = CCCallFuncN::create(this,
        callfuncN_selector(SpriteLayer::moveActionEnd));

    CCAction* action = CCSequence::create(move, end, NULL);
    fish->runAction(action);
}
```

- 最后，在游动动作完成后，我们执行 moveActionEnd 方法。在这个方法中，我们重新设置这条鱼的游动动作：

```
void SpriteLayer::moveActionEnd(CCNode* sender)
{
    this->setFishRoute(dynamic_cast<CCSprite*>(sender), fishUpdateInterval);
}
```

- 现在我们已经可以通过调用 updateFishMovement 方法给所有鱼添加游动的动作了。因此，我们只需要在游戏中添加一定数量的鱼，并调用 updateFishMovement 方法，即可看到场景中的鱼开始游动。

这里我们不仅实现了鱼的游动，而且由于用到了帧动画技术，鱼的身体也会不断摆动。然而现在鱼的运动还是很不自然的，我们从图上可以看到，鱼在游动时，运行方向并不与运动曲线的方向一致，看起来就像鱼在水中漂流一样。因此，我们需要让鱼在执行以上动作的同时，也要保持鱼朝前进方向旋转。

对于 linearRoute 方法来说，改变方向十分简单：由于只是简单的直线运动，我们只需要在一开始计算出鱼游动的方向，然后使用 setRotate 函数即可。我们可以使用类似于下面的代码来计算鱼旋转的角度：

```
float rotate = -(exitY - enterY) / s.width;
fish->setRotation(rotate / 3.14159f * 180);
```

而对于曲线运动，我们可以用 CCSpawn 对曲线动作与旋转动作进行包装，即可实现鱼一边沿曲线移动一边转动的效果，相关代码如下：

```
CCFiniteTimeAction *curveMove = CCSequence::create(
    CCMoveTo::create(dt / 2, ccp(s.width / 2 + r[!direction], h[direction])),
    CCSpawn::create(
        CCBezierTo::create(dt / 4, b1),
        CCRotateBy::create(dt / 4, -90)),
    CCSpawn::create(
        CCBezierTo::create(dt / 4, b1),
        CCBezierTo::create(dt / 4, b2)),
    CCMoveTo::create(dt / 2, to),
    NULL);
```

curveMove 是使用 CCSpawn 包装后的曲线动作，我们用它替换掉前文 bezierRoute 方法中的 move 动作。它使鱼在第一段曲线上运动的同时，逆时针旋转 90 度，而在第二段曲线上运动的同时，顺时针旋转 90 度，虽然旋转不够精确，但是看起来自然多了。

4.7.1 一点简单的物理知识

4.7 创建自定义动作

我们利用前面介绍的知识已经实现了简易的游动动作，而这个动作中唯一不足的是对于曲线运动来说，鱼的方向并没有精确地吻合游动轨迹。事实上，我们完全有能力精确且简便地确定鱼的方向，但是需要用到一点高中的物理知识，以及如何在 Cocos2d-x 中创建一个自定义动作的知识。

4.7.1 一点简单的物理知识

在继续解决游动方向的问题前，我们先复习一些简单的物理知识，这些知识给我们提供了一个简单的方法来精确地追踪精灵移动的方向。

如果物体沿着一条曲线移动，那么物体在某一点的瞬时速度方向一定是该点切线的正方向。对于游动中的鱼来说，它的头部永远面向前进方向，因此鱼的方向与瞬时速度方向相同。只要能计算出某一点的切线方向，就可以把这个方向当做鱼在这一点的游动方向。计算切线方向的方法有许多，但是并不是所有方法都易于实现。现在我们来研究其中一个最简单易行的方法。

首先复习平均速度与瞬时速度的概念。给定两个点，我们可以计算出在它们之间运动的平均速度，其值等于位移与时间的比值，其方向与位移方向相同。给定任何一个点，我们也可以计算瞬时速度。瞬时速度定义为该点到另一点的平均速度，但这两个点的距离要趋向于无穷小。

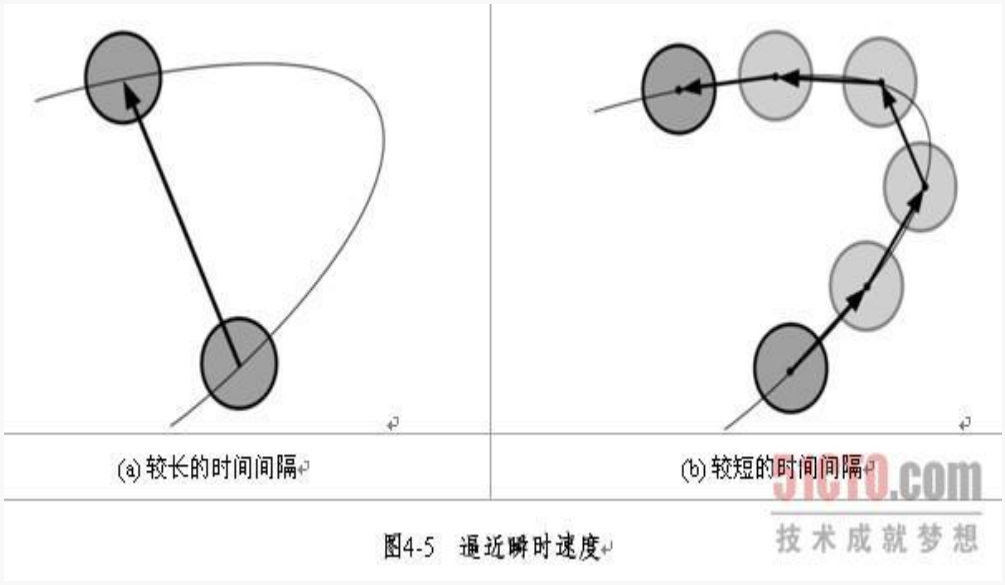
- 可以发现，我们能很轻易地利用计算机算出两点之间的平均速度，而瞬时速度的计算则相对困难一些。为了计算某点的瞬时速度，我们必须从该点附近切割出路径的一小部分，计算这两个点之间的平均速度。切割的路径越短，得到的速度就越接近瞬时速度。假设我们切割路径的长度趋近于无穷小，则平均速度的极限值就是瞬时速度了（如图 4-5 所示）。
- 

图4-5 逼近瞬时速度
- 我们在调整鱼的方向时，只要知道运动轨迹中某一点的瞬时速度的方向即可，因此我们并不需要得到准确的瞬时速度值。考虑到 Cocos2d-x 游戏通常会达到 60 帧/秒，每两帧?之间的时间间隔平均仅 1/60 秒，这是一个足够小的时间段，在这段时间内鱼游动的距离值也十分微小。所以我们可以近似地认为两帧之间鱼的平均速度等于鱼在此处附近的瞬时速度。因此，我们只需要根据两帧中鱼的位置差计算出鱼前进的方向。

● 4.7.2 创建自定义动作

- 4.7.2 创建自定义动作

- 为了追踪鱼游动的方向，我们可以编写一个定时器，通过帧的转换来更新鱼的方向，不过这也是一个既烦琐又难以维护的办法。参考引擎的做法，我们不妨进一步抽象出独立的旋转跟踪动作，根据精灵的移动路径设置合适的旋转角度。
- CCAction 包含两个重要的方法：step 与 update。step 方法会在每一帧动作更新时触发，该方法接受一个表示调用时间间隔的参数 dt，dt 的积累即为动作运行的总时间。引擎利用积累时间来计算动作运行的进度（一个从 0 到 1 的实数），并调用 update 方法更新动作。update 方法是 CCAction 的核心，它由 step 方法调用，接受一个表示动作进度的参数，每一个动作都需要利用进度值改变目标节点的属性或执行其他指令。自定义动作只需要从这两个方法入手即可，我们通常只需要修改 update 方法就可以实现简单的动作。
- 下面我们编写一个继承于 CCAction 的 CCRotateAction 动作。如同复合动作与变速动作一样，它会把另一个动作包装起来，在执行被包装动作的同时，设置精灵的方向。为此，我们需要在每一帧?记录上一帧精灵的位置，然后再根据精灵两帧的位移确定精灵的方向。由于我们必须在 CCRotateAction 执行的同时运行被包含的目标动作，所以我们需要在 step 方法中调用目标动作的 step 方法。下面我们来看 CCRotateAction 的实现。
- "RotateWithAction.h"中的定义如下：

```
class RotateWithAction : public CCActionInterval
{
public:
    CCObject* copyWithZone(CCZone* pZone);
```

```
    ~RotateWithAction();
    static RotateWithAction* create(CCActionInterval * action);
    virtual void startWithTarget(CCNode* pTarget);
    bool initWithAction(CCActionInterval* pAction);
    bool isDone();
    void step(ccTime dt);

protected:
    void RotateWithAction::setInnerAction(CCActionInterval* pAction);

    CCNode* pInnerTarget;
    CCActionInterval* pInnerAction;

};
```

■ "RotateWithAction.cpp"中的实现如下：

```
RotateWithAction::~RotateWithAction()
{
    CC_SAFE_RELEASE(pInnerAction);
}

RotateWithAction* RotateWithAction::create(CCActionInterval* pAction)
{
    RotateWithAction* action = new RotateWithAction();
    if (action && action->initWithAction(pAction))
    {
        action->autorelease();
        return action;
    }
    CC_SAFE_DELETE(action);
    return NULL;
}

bool RotateWithAction::initWithAction(CCActionInterval* pAction)
{
    pAction->retain();
    pInnerAction = pAction;
    return true;
}

void RotateWithAction::startWithTarget(CCNode* pTarget)
{
    pInnerTarget = pTarget;
    CCAction::startWithTarget(pTarget);
    pInnerAction->startWithTarget(pTarget);
```

```
}

bool RotateWithAction::isDone()
{
    return pInnerAction->isDone();
}

void RotateWithAction::step(ccTime dt)
{
    CCPoint prePos = pInnerTarget->getPosition();
    pInnerAction->step(dt);
    CCPoint curPos = pInnerTarget->getPosition();

    float tan = -(curPos.y - prePos.y) / (curPos.x - prePos.x);
    float degree = atan(tan);
    degreedegree = degree / 3.14159f * 180;

    pInnerTarget->setRotation(degree);
}

void RotateWithAction::setInnerAction(CCActionInterval* pAction)
{
    if (pInnerAction != pAction)
    {
        CC_SAFE_RELEASE(pInnerAction);
        pInnerAction = pAction;
        CC_SAFE_RETAIN(pInnerAction);
    }
}

CCObject* RotateWithAction::copyWithZone(CCZone* pZone)
{
    CCZone* pNewZone = NULL;
    RotateWithAction* pCopy = NULL;
    if(pZone && pZone->m_pCopyObject)
    {
        pCopy = (RotateWithAction*)(pZone->m_pCopyObject);
    }
    else
    {
        pCopy = new RotateWithAction();
        pZone = pNewZone = new CCZone(pCopy);
    }

    CCActionInterval::copyWithZone(pZone);
```

```
    pCopy->initWithAction(dynamic_cast<CCActionInterval*>
        (pInnerAction->copy()->autorelease()));

    CC_SAFE_DELETE(pNewZone);
    return pCopy;
}
```

- 也许有的读者已经有了疑问，step 方法与 update 方法都可以做到每一帧判断一次方向，为什么选择重载 step 方法而不是 update 方法呢？这是因为引擎在 step 方法中对动作对象的内部成员进行了更新，更新后才会由此方法调用 update 方法来更新目标节点。在方向追踪的动作中，我们除了在每一帧判断方向，还必须同步执行被包装的动作。这就需要我们调用被包装动作的 step 方法，以保证对象能够被完整地更新。
- 现在，我们已经不需要使用 4.6 节介绍的 CCSpawn 来实现蹩脚的方向追踪效果了，只要把需要追踪方向的动作传递给 CCRotateAction，即可得到一个自动改变鱼方向的智能动作。

4.8 让动作更平滑流畅

- 4.8 让动作更平滑流畅

- 下面探讨的也许更像交互设计师关心的问题，但是在游戏开发中这个问题却有着不可或缺的重要性：如何让一个动作看起来更加平滑流畅？换句话说，如何让动作看起来更加自然并优雅？
- 以暂停游戏时弹出的菜单为例，点击暂停游戏后，菜单从屏幕顶端向下滑出；点击恢复游戏后，菜单向上收起。下面介绍一个游戏开发中经常采用的、直观的写法。菜单的弹出代码如下：

```
CCSize size = CCDirector::sharedDirector()->getWinSize();

CCMenu* menu = CCMenu::create(item0, item1, item2, item3, NULL);
menu->alignItemsVerticallyWithPadding(5.0f);
menu->setPosition(ccp(size.width/2.0f, size.height));
menu->setTag(menu_pause_tag);
this->addChild(menu, 5);

menu->runAction(CCMoveTo::create(0.5f, ccp(size.width / 2.0f, size.height / 2.0f)));
```

- 菜单的收起代码如下：

```
CCSize size = CCDirector::sharedDirector()->getWinSize();
CCMenu* menu = (CCMenu*)this->getChildByTag(menu_pause_tag);
CCPoint point = ccp(size.width / 2.0f, size.height + menu->getContentSize().height / 2.0f);
CCMoveTo* move = CCMoveTo::create(0.5f, point);

menu->runAction(move);
```

- 代码中存在一个神奇的动作，它持续时长为 0.5 秒。当我们试图修改这个动作的时长时，很容易陷入这样一个困境：时间长了显得拖沓，时间短了则来不及反应，最后不管如何调整，总是显得不甚完美。
- 实际上，这是一个涉及玩家注意力的问题。对于新出现的变化效果，玩家需要时间转移注意力适应这个变化，而后如果效果持续稳定、变化不明显，则会降低玩家的注意力，使玩家感觉疲惫。在这种情况下，一个冗长的匀速动作效果就会造成前面提到的问题了。

- 我们的菜单收放效果就很好地印证了这个结论。菜单的全部收放动作效果形成了一个比较长且单一的运行轨迹，所以我们不妨为动作添加一些变速效果，将玩家有限的注意力集中到我们希望玩家关注的效果上。
- 进场动作：由快到慢，快速进入后缓慢停下，在停止前给玩家足够的视觉时间分辨清楚进入的图像。
- 出场动作：先慢后快，展示了出场趋势和方向后快速移出屏幕，不拖泥带水。
- 这个变速效果就很自然地交给前面提到的 CCEase 系列动作实现了。针对具体的需求，我们选择了 CCEaseExponential 动作来实现变速效果。弹出菜单的代码如下：

```
CCMenu* menu = CCMenu::create(item0, item1, item2, item3, NULL);
menu->alignItemsVerticallyWithPading(5.0f);
menu->setPosition(ccp(size.width/2, size.height));
menu->setTag(menu_pause_tag);
this->addChild(menu, 5);
CCMoveTo* move = CCMoveTo::create(0.5f, ccp(size.width/2, size.height/2));

CCAction* action = CCEaseExponentialOut::create(move);
menu->runAction(action);
```

- 收起菜单的代码如下：

```
CCSize size = CCDirector::sharedDirector()->getWinSize();
CCMenu* menu = (CCMenu*)this->getChildByTag(menu_pause_tag);
CCPoint point = ccp(size.width/2, size.height + menu->getContentSize().height/2);
CCMoveTo* move = CCMoveTo::create(0.5f, point);

CCAction* action = CCEaseExponentialIn::create(move);
menu->runAction(action);
```

- 类似的变速处理的应用范围很广。在《捕鱼达人》游戏中，击中鱼后，弹出的得分提示和切换炮的效果都是这样处理的。至于在骨架模型等更大型动画中的应用，这里就不再详细介绍了，感兴趣的读者可以查阅相关图书。

4.9.1 动作类的结构

- **4.9 Cocos2d-x 动作原理**
- 在深入学习了整套动作机制的用法之后，相信读者一定很好奇，究竟动作机制在 Cocos2d-x 里是如何实现的？在本节中，我们将如同庖丁解牛一般，揭开动作机制的神秘面纱。
- **4.9.1 动作类的结构**
- 首先，分析一下 CCAction 及其子类（主要是 CCFiniteTimeAction 及其子类）的一些成员函数和成员变量，我们将通过这些变量和函数来分析动作的基本流程。
- 从 CCAction 的定义中可以看到，在类定义的最后部分有 3 个成员变量和一些基本方法：

```
class CC_DLL CCAction : public CCObject
{
public:
    CCAction(void);
    virtual ~CCAction(void);

```

```
    const char* description();
    virtual CCObject* copyWithZone(CCZone *pZone);

    virtual bool isDone(void);
    virtual void startWithTarget(CCNode *pTarget);
    virtual void stop(void);
    virtual void step(float dt);
    virtual void update(float time);

    inline CCNode* getTarget(void) { return m_pTarget; }
    inline void setTarget(CCNode *pTarget) { m_pTarget = pTarget; }

    inline CCNode* getOriginalTarget(void) { return m_pOriginalTarget; }
    inline void setOriginalTarget(CCNode *pOriginalTarget) {
        m_pOriginalTarget = pOriginalTarget; }

    inline int getTag(void) { return m_nTag; }
    inline void setTag(int nTag) { m_nTag = nTag; }

public:
    CC_DEPRECATED_ATTRIBUTE static CCAction* action();
    static CCAction* create();
protected:
    CCNode *m_pOriginalTarget;
    CCNode *m_pTarget;
    int m_nTag;
};
```

- 继承自 CCAction 的 CCFiniteTimeAction 主要新增了一个用于保存该动作总的完成时间的成员变量：ccTime m_fDuration。
- 对于 CCFiniteTimeAction 的两个子类 CCActionInstant 和 CCActionInterval，前者没有新增任何函数和变量，而后者增加了两个成员变量--ccTime m_elapsed 和 bool m_bFirstTick，其中 m_elapsed 是从动作开始起逝去的时间，而 m_bFirstTick 是一个控制变量，在后面的分析中，我们将看到它的作用。

4.9.2 动作的更新

4.9.2 动作的更新

- 在了解了 CCAction、CCFiniteTimeAction 和 CCActionInterval 的类结构后，下面我们以它们为例分析 Cocos2d-x 的动作机制。
- 当我们对 CCNode 调用 runAction(CCAction* action)方法时，动作管理类 CCAction Manager（它是一个单例对象）会将新的 CCAction 和对应的目标节点添加到其管理的动作表中。
- 在 CCActionManager 的 addAction 方法中，我们将动作添加到动作队列之后，就会对该 CCAction 调用成员函数 startWithTarget(CCNode* pTarget)来绑定该动作的执行者。而在 CCAction 的子类中（如 CCActionInterval），还初始化了一些参数：

```
void CCActionInterval::startWithTarget(CCNode *pTarget)
{
```

```
    CCFiniteTimeAction::startWithTarget(pTarget);
    m_elapsed = 0.0f;
    m_bFirstTick = true;
}
```

- 当这些准备工作都完成后，每一帧刷新屏幕时，系统都会在 CCActionManager 中遍历其动作表中的每一个动作，并调用该动作的 step(ccTimedt)方法。step 方法主要负责计算 m_elapsed 的值，并调用 update(float time)方法，相关代码如下：

```
void CCActionInterval::step(float dt)
{
    if (m_bFirstTick)
    {
        m_bFirstTick = false;
        m_elapsed = 0;
    }
    else
    {
        m_elapsed += dt;
    }

    this->update(MAX (0,
        MIN(1, m_elapsed / MAX(m_fDuration, FLT_EPSILON))
        )
    );
}
```

- 传入 update 方法的 time 参数表示逝去的时间与动作完成需要的时间的比值，是介于 0 和 1 之间的一个数，即动作完成的百分比。

- CCActionInterval 并没有进一步实现 update 方法。下面我们继续以继承自 CCAction- Interval 的 CCRotateTo 动作的 update 方法为例，分析 update 函数是如何实现的，其实现代码如下：

```
void CCRotateTo::update(float time)
{
    if (m_pTarget)
    {
        m_pTarget->setRotation(m_fStartAngle + m_fDiffAngle * time);
    }
}
```

- 看到这里，我们已经能看出 Cocos2d-x 的动作机制的整个工作流程了。在 CCRotateTo 中，最终完成的操作是修改目标节点的 Rotation 属性值，更新该目标节点的旋转属性值。

- 最后，在每一帧刷新结束后，在 CCActionManager 类的 update 方法中都会检查动作队列中每一个动作的 isDone 函数是否返回 true。如果返回 true，则动作已完成，将其从队列中删除。isDone 函数的代码如下：

```
bool CCActionInterval::isDone(void)
{
    return m_elapsed >= m_fDuration;
}
```

■ 对于不同的动作类，虽然整体流程大致都是先调用 step 方法，然后按照各个动作的具体定义来更新目标节点的属性，但是不同动作的具体实现会有所不同。例如，CCRepeatForever 动作的 isDone 函数始终返回 false，因为它是永远在执行的动作；又如 CCActionInstant 及其子类的 step 函数中，向 update 传递的参数值始终是 1，因为瞬时动作会在下一帧刷新后完成，不需要多次执行 update。

● 4.9.3 CCActionManager 的工作原理

■ 4.9.3 CCActionManager 的工作原理

■ 学习了 CCAction 在每一帧中如何被更新之后，我们不妨回头看看动作管理类 CCAction- Manager 的工作原理。在对 CCDirector 进行初始化时，也会对 CCActionManager 进行初始化。下面的代码是 CCDirector::init()方法中的一部分：

```
//动作管理器
m_pActionManager = new CCActionManager();
m_pScheduler->scheduleUpdateForTarget(m_pActionManager, kCCPrioritySystem, false);
```

■ 可以看到，在 CCActionManager 被初始化后，马上就调用了定时调度器 CCScheduler 的 sc- heduleUpdateForTarget 方法。在 scheduleUpdateForTarget 函数中，我们为 CCActionManager 注册了一个定期更新的服务，这意味着动作的调度与定时器的调度都统一受到 CCScheduler 的控制。具体地说，我们可以方便地同时暂停或恢复定时器与动作的运行，而不必考虑它们不同步的问题。

■ CCScheduler 在每一帧更新时，都会触发 CCActionManager 注册的 update 方法。从下面给出的 CCActionManager::update 方法的代码可以看到，CCActionManager 在这时对每一个动作都进行了更新。与调度器 CCScheduler 类似的一点是，为了防止动作调度过程中所遍历的表被修改，Cocos2d-x 对动作的删除进行了仔细地处理，保证任何情况下都可以安全地删除动作：

```
void CCActionManager::update(float dt)
{
    //枚举动作表中的每一个目标节点
    for (tHashElement *elt = m_pTargets; elt != NULL; )
    {
        m_pCurrentTarget = elt;
        m_bCurrentTargetSalvaged = false;

        if (! m_pCurrentTarget->paused)
        {
            //枚举目标节点对应的每一个动作
            //actions 数组可能会在循环中被修改，因此需要谨慎处理
            for (m_pCurrentTarget->actionIndex = 0;
                m_pCurrentTarget->actionIndex < m_pCurrentTarget->actions->num;
                m_pCurrentTarget->actionIndex++)
            {
                m_pCurrentTarget->currentAction =
                    (CCAction*)m_pCurrentTarget->actions
                        ->arr[m_pCurrentTarget->actionIndex];

                if (m_pCurrentTarget->currentAction == NULL)
                {
                    continue;
                }
```

```
                m_pCurrentTarget->currentActionSalvaged = false;

                m_pCurrentTarget->currentAction->step(dt);    //触发动作更新

                if (m_pCurrentTarget->currentActionSalvaged)
                {
                    m_pCurrentTarget->currentAction->release();
                }
                else if (m_pCurrentTarget->currentAction->isDone())
                {
                    m_pCurrentTarget->currentAction->stop();

                    CCAction *pAction = m_pCurrentTarget->currentAction;

                    m_pCurrentTarget->currentAction = NULL;
                    removeAction(pAction);
                }
                m_pCurrentTarget->currentAction = NULL;
            }
        }

        elt = (tHashElement*)(elt->hh.next);

        if (m_bCurrentTargetSalvaged && m_pCurrentTarget->actions->num == 0)
        {
            deleteHashElement(m_pCurrentTarget);
        }
    }

    m_pCurrentTarget = NULL;
}
```

4.10 小结

4.10 小结

在这一章中，我们学习了 Cocos2d-x 中的动作以及控制所有动作的动作管理器 CCAction- Manager。Cocos2d-x 的动作大致分为持续性动作与瞬时动作，持续性动作会持续进行一段时间，而瞬时动作在一帧?之内完成。

除了简单的各项具体动作以外，Cocos2d-x 还提供了复合动作与变速动作。复合动作用于把简单的动作组合成复杂的动作，可以实现动作序列和并列动作等效果。变速动作为原本线性变化的动作提供了改变速度的方法，与简单动作组合可以实现灵活的速度控制。

最后，我们为了解决鱼在运动中改变方向的问题引入了自定义动作，即重载 CCAction 类的 step 方法或 update 方法，跟踪动作的更新，并实现控制方向的效果。

我们的捕鱼游戏逐渐充满生机地动了起来，但美中不足的是现在的鱼仅仅是一张可以旋转和移动的图片，鱼的身体不能摆动，如同一条死气沉沉的船一样。为了实现鱼在移动的过程中摆动身体，我们需要进一步学习第 5 章关于动画的内容。

- 至此，我们对 Cocos2d-x 动作的探索就告一段落了，感兴趣的读者不妨进一步研读 Cocos2d-x 动作部分的源码，给游戏添加更多生动的动作效果，继续加强游戏的表现力。

5.1 动画

- **动画与场景特效**

- 在第 4 章中，我们深入探讨了 Cocos2d-x 的动作，了解了许多引擎内置的动作。在这一章中，话题相对轻松了许多，我们将从两个问题出发介绍 Cocos2d-x 中两类动作的特殊应用--动画与场景特效，并着手为我们的《捕鱼达人》添加相应的效果。

- **5.1 动画**

- 我们相信读者读完第 4 章后，一定很好奇《捕鱼达人》中鱼的游动效果是如何实现的。我们的鱼终于动起来了，然而此时的鱼身体僵直，与图片无异。对于鱼的游动以及诸如人物四肢摆动、表情变化等这类无法使用简单的属性变化实现的动态效果，Cocos2d-x 提供的动作似乎无能为力。为了解决这个问题，我们需要引入一种特殊的动作：动画。

5.1.1 概述

- **5.1.1 概述**

- 作为一个力图降低游戏开发难度的 2D 引擎，Cocos2d-x 并没有使用诸如 3D 和矢量等手段来实现复杂的动画效果，而是引入了帧动画来表现一些动作（CCAction）难以实现的特效。帧动画类似我们平常播放的视频，引擎把我们编辑好的动画逐帧播放，并呈现在游戏中。我们可以把任意特效编辑为一段动画，因此，理论上来说，帧动画可以实现任何一种效果。

- 帧动画与电影胶片类似。一个连贯的动画实际上是由许多独立的图片按时间顺序组合而成的，动画的帧就是指被显示出来的每一张图片。由于播放动画时，图片间的切换速度极快，展示出来的效果就是动态的动画效果了。动画的内容是任意的，因此在不计开销的情况下可以表现出任何动态效果。通常，对于 Cocos2d-x 无法完成的复杂动态效果，我们能利用动画加以实现。图 5-1 是一个简单的帧动画示例，它实现了鱼的游动动画效果。

- 

图5-1 “游动的鱼”帧动画

- 动画的制作涉及了许多其他方面的知识，限于篇幅我们并不打算在本书中加以介绍。通常，较为简单的动画可以利用 Flash 工具制作出来，而更为复杂与细腻的动画可以利用三维建模软件逐帧渲染，或完全手动绘制。由于动画的最终输出文件包含了每一帧的图片内容，体积比较庞大，会给内存与显存带来较大的压力。考虑到制作成本以及回放成本，如果没有必要，我们一般不在游戏中大规模使用动画。

5.1.2 使用动画

- **5.1.2 使用动画**

- 在 Cocos2d 中，动画的具体内容是依靠精灵显示出来的。我们知道精灵可以用来显示一张静止的图片，而为了显示动态图片，我们需要不停地切换精灵显示的内容。一旦明白了这个道理，我们可以利用定时器不停地改变精灵的显示内容，把静态的精灵变为动画播放器。事实上，Cocos2d-x 提供的动画就是基于这个原理。

- 动画由帧组成。在最简单的情况下，每一帧都是一个纹理，我们可以使用一个纹理序列来创建动画。然而显卡在绘图时，在纹理间切换是一个开销巨大的操作，由于精灵可以显示部分纹理，因此通常更为高效的做法是把动画用到的多个纹理按

照一定的顺序排列起来，然后放置在同一个纹理下。在创建动画时，我们不仅需要指定动画所使用的纹理，还需要指定每一帧使用的是纹理的哪一部分。

- 为了方便地记录纹理的显示信息，Cocos2d-x 提供了框帧类 CCSpriteFrame。一个框帧包含两个属性，纹理与区域。纹理指的是将要被显示的纹理，而区域指的是此纹理将要被显示的部分。一个框帧可以完整地描述精灵显示的内容，因此在动画中，我们使用框帧来表示每一帧的内容。

- 当我们准备好了框帧的序列，匀速地播放这一系列帧动画就不成问题了，我们只需要定时切换精灵显示的框帧即可。然而，我们也经常需要非匀速的动画效果，此时每一帧显示的时间就需要特殊设置了。为了描述一帧，除了框帧，显然我们还需要记录帧的持续时间。动画帧类 CCAni- mationFrame 同样包含两个属性，其一是对一个框帧的引用，其二是帧的延时。一个 Cocos2d-x 的动画类 CCAnimation 是对一个动画的描述，它包含显示动画所需要的动画帧。对于匀速播放的帧动画，只需设置所有帧的延时相同即可。

- 我们使用 CCAnimation 描述一个动画，而精灵显示动画的动作则是一个 CCAnimate 对象。动画动作 CCAnimate 是精灵显示动画的动作，它由一个动画对象创建，并由精灵执行。动画与动画动作的关系就如同 CD 光盘与 CD 播放机的关系一样--前者记录了动画的内容，而后者是播放动画的工具。关于这几个类，我们没有必要一一赘述。下面我们来看一个创建游动的鱼的例子，在这个例子中，我们继续使用前文展示过的图片作为动画资源，这张图片由 8 张小图组成，每一张小图都是一个独立的状态，为了方便使用，每张图片的宽度和高度都分别相同：

```
CCTexture2D* texture = CCTextureCache::sharedTextureCache()->addImage(s_fishPath);
float w = texture->getContentSize().width / numOfFishFrame;
float h = texture->getContentSize().height;

CCAnimation* animation = CCAnimation::create();
animation->setDelay(0.15f);
for(int i = 0; i < numOfFishFrame; i ++)
    animation->addFrameWithTexture(texture, CCRectMake(i * w, 0, w, h));
CCAnimate* animate = CCAnimate::create(animation);
fish->runAction(CCRepeatForever::create(animate));
```

- 在这个例子中，我们首先利用 CCTextureCache 加载鱼游动的动画纹理，然后创建动画对象，设置 Delay 属性，修改每帧播放的时间间隔为 0.15 秒。接下去，我们编写了一个循环。由于游动动作匀速执行，在循环中我们给动画添加每一帧所使用的框帧，框帧的纹理都一样，位置可以通过简单的计算得到。最后，我们利用创建好的动画对象来创建一个动画动作，让鱼精灵播放此动画。

- 编辑好上述代码并把鱼位置设定好后，运行程序，就能看到鱼在上面几张小图间不断切换，从而实现了动画效果。当然，我们对鱼赋予其他的动作，并不会影响到动画的播放，因此我们能够实现鱼一边做曲线运动，一边不断地摆动尾巴的效果。类似于这种方式，我们还可以加入鱼被击中后挣扎的动画和金币在飞行中旋转的动画。

- 在实际开发中，帧动画文件通常由工具来生成。在生成帧动画纹理之前，需要拥有每一帧的图片资源，把这些资源经过类似 TexturePacker 的工具打包到一个较大的纹理之中，然后添加到帧动画里。关于 TexturePacker，10.3 节将详细介绍。

● 5.2 场景特效

- 5.2 场景特效

- 谈到场景切换，读者一定会回想起我们介绍 Cocos2d-x 基本游戏元素时介绍的导演类 CCDirector。利用 CCDirector 提供的 replaceScene 方法可以方便地把当前场景切换为另一个场景。

- 然而 replaceScene 方法的场景切换并没有过多修饰，仅仅是停止当前场景，再播放下一个场景。场景的切换有时会显得生硬而单调。为了解决这个问题，Cocos2d-x 为我们提供了很多场景切换的特效，包括在切换场景时表现出翻页、波浪等华丽的特效，这些特效是通过特效类 CCTransitionScene 来实现的。

- CCTransitionScene 派生自 CCScene，换句话说，场景特效本身也是一个场景。讲到这里，聪明的读者也许已经想到了，场景特效的实现方式与复合动作类似：复合动作是一类特殊的动作，它们包含其他动作，执行复合动作时，被包含的动作也会按照一定的方式执行；而场景特效是一类特殊的场景，它们包含了另一个场景，在运行场景特效时，被包含的原场景会以添加了特效的方式显示出来。因此，特效场景的使用方法与复合动作也类似。首先创建一个场景，称作原场景，然后把原场景当做参数来创建一个特效场景，使用时只需要把特效场景传入 CCDirector 的 replaceScene 方法即可，相关代码如下：

```
CCDirector::sharedDirector()->replaceScene
    (CCTransitionFlipX::transitionWithDuration(2, pScene));
```

- 表 5-1 简单列举了部分具有代表性的场景特效。全部的场景特效的代码都位于引擎目录中的"layers_scenes_transitions_nodes/CCTransition.h（cpp）"文件中，读者不妨一一尝试。

- 表 5-1　常用的场景特效

场景特效类	描述
CCTransitionJumpZoom	跳跃缩放，当前场景会先缩小，然后跳跃切换
CCTransitionFade	淡出淡入效果，可以设置覆盖颜色
CCTransitionFlipX (CCTransitionFlipY)	沿着 x 轴向左翻
CCTransitionShrinkGrow	交错切换场景
CCTransitionMoveInL	从左进入覆盖原场景
CCTransitionSlideInL	从左进入推出原场景

5.3　小结

5.3　小结

- 经过对这一章的学习，我们为游戏添加了动画与场景特效，整个游戏已经栩栩如生地动了起来。下面我们总结一下这一章的重点知识。

- 框帧（CCSpriteFrame）：包含纹理与纹理中的一个矩形区域，表示纹理的一部分。一个精灵显示的内容就可以用框帧表示，同时框帧还是帧动画的基本元素。

- 动画帧（CCAnimationFrame）：由框帧与单位延时组成，可以表示变速动画中的一帧。通常，匀速动画的单位延时为 1。

- 动画（CCAnimation）：由动画帧组成，表示一个动画的内容。

- 动画动作（CCAnimate）：动画的播放器，使用动画对象创建，只能作用于精灵。为了播放一个动画，通常首先创建动画帧或框帧，然后用它们创建动画，最后利用动画创建动画动作，并指派一个精灵来执行此动作。

- 场景特效（CCTransitionScene）：一类特殊的场景，可以把另一个场景包装起来，实现诸如特殊翻页、波纹等华丽的场景切换特效。

- 引擎提供的这些功能一方面丰富了游戏的视觉感受，另一方面也对运行设备的硬件性能提出了更高的要求。到目前为止，我们并没有过多地了解这些效果背后的优化原则，对这些技术细节我们不妨暂且不考虑，先进一步探索引擎和完善我们的《捕鱼达人》。在后续的引擎进阶部分，我们将为读者--揭开其他谜底。

6.1 使用音效引擎

音乐与音效

在玩游戏时，视觉、触觉与听觉是玩家与游戏互动的 3 种形式，每一种形式都是十分重要的。在介绍本章之前，我们已经用了 5 章内容来讨论如何在屏幕上给玩家呈现丰富多彩的图案，然而作为互动形式之一的听觉却常常被开发者忽略。实际上，在游戏中实现一套优质的音乐与音效远比制作漂亮的画面简单得多。只需要开发者完成很少的工作量，就能把游戏的互动效果提高一个层次。

在游戏中，我们把声音分为两类。第一类是音乐，这种类型的声音通常长度较长，适合作为环境音乐（例如游戏的背景音乐）。由于它的长度较长，同一时刻通常只能播放一首音乐。第二类是音效，它的特点是长度很短，但是可以同时播放多个音效，拥有很强的表现力。

Cocos2d-x 提供了对音乐与音效的支持，能够十分方便地实现音乐与音效的播放、暂停和循环功能。在这一章中，我们将简单地介绍 Cocos2d-x 如何为游戏添加音乐与音效。

6.1 使用音效引擎

Cocos2d 的音效引擎库 CocosDenshion 随着引擎一同被分发。除了在游戏中使用外，音效引擎有时也会用在其他软件中。因此，虽然 CocosDenshion 与游戏引擎捆绑发布，却并不属于游戏引擎的一部分。

在游戏开发中，我们可以十分方便地启用 CocosDenshion 音效引擎库。CocosDenshion 位于 Cocos2d-x 目录下的"CocosDenshion"目录中。通常，Cocos2d-x 项目已经包含了 CocosDenshion 库，当我们需要使用音效引擎时，把引擎头文件引进来即可。

CocosDenshion 实现了简单易用的 SimpleAudioEngine 类。为了使用音效引擎，我们只要引入它的头文件即可：

```
#include "SimpleAudioEngine.h"
```

CocosDenshion 移植自 Cocos2d-iPhone 中的同名库。实际上，Cocos2d-iPhone 中的 CocosDenshion 实现了 3 个音效引擎，由底层到高级分别是 CDSoundEngine、CDAudioManager 和 SimpleAudioEngine，其中前两个较为底层的引擎用于高级音频的开发，例如实现 3D 混音等。然而，对于普通开发者而言，SimpleAudioEngine 已经足以满足大部分游戏开发的需求了。底层的音效引擎封装自 OpenAL 音频接口，从而实现了十分灵活、高效的音频回放引擎。然而，OpenAL 只能被 OS X（包括 iOS）平台支持,其他平台下是没有类似接口的,因此 Cocos2d-x 不得不舍弃 CDSoundEngine 和 CDAudioManager 这两个底层引擎，只保留了最常用的 SimpleAudioEngine。

总之，当我们引入了这个头文件后，就可以使用音效引擎了。

6.2 支持格式

6.2 支持格式

CocosDenshion 引擎库实际上是对系统音频 API 的封装，因此它支持的音频文件格式与平台有关。表 6-1 与表 6-2 列出了主流平台下 CocosDenshion 所支持的文件格式。

表 6-1 CocosDenshion 支持的音乐格式

平台	支持的常见文件格式	备注
Android	mp3、mid、ogg 和 wav	可以播放 android.media. MediaPlayer 所支持的所有格式
iOS	aac、caf、mp3、m4a 和 wav	可以播放 AVAudioPlayer 所支持的所有格式
Windows	mid、mp3 和 wav	无

表 6-2　CocosDenshion 支持的音效格式

平台	支持的常见文件格式	备注
Android	ogg 和 wav	对 wav 的支持并不完美
iOS	caf 和 wav	可以播放 Cocos2d-iPhone CocosDenshion 所支持的所有格式
Windows	mid 和 wav	无

6.3.1　预加载

6.3　播放音乐与音效

SimpleAudioEngine 与许多 Cocos2d 的部件一样，是一个单例类。我们使用以下代码来访问它的实例：

```
SimpleAudioEngine::sharedEngine();
```

它提供了许多与音乐和音效播放相关的方法，它们使用起来都十分简单。下面我们介绍最常用的几个方法。

6.3.1　预加载

加载音乐和音效通常是一个耗时的过程，为了防止由即时加载产生的延迟导致实际播放与游戏不协调的现象发生，在播放音效和背景音乐之前，需要预加载音乐文件。通常，我们会在进入游戏场景前的载入阶段调用下面的这两个方法。

void preloadEffect(const char* pszFilePath)：用于预加载音效文件，其中 pszFilePath 为音效文件所在的目录位置。

void preloadBackgroundMusic(const char* pszFilePath)：用于预加载背景音乐，其中 pszFilePath 为音乐文件所在的目录位置。

6.3.4　其他成员

6.3.4　其他成员

除了以上几个常用的音频回放的相关方法，Cocos2d-x 还提供了其他十分便捷的控制方法与属性，下面简要介绍一下它们。

void rewindBackgroundMusic()：重新播放背景音乐。

bool isBackgroundMusicPlaying()：返回一个布尔类型的值，表示是否正在播放背景音乐。

void unloadEffect(const char*pszFilePath)：卸载已预载入的音效文件，以释放系统资源。pszFilePath 参数代表预载入音效文件的路径。当不再使用某个音效文件时，我们可以通过调用此函数释放资源。然而，如果再次使用此音效，引擎会再次载入该音效文件，导致消耗大量的时间。

float EffectsVolume 属性：获取或设置音效的音量大小，其取值为 0.0 到 1.0 之间的浮点数。注意，对此属性的设置会影响到所有音效的音量大小。

float BackgroundMusicVolume 属性:获取或设置背景音乐的音量大小,其取值为 0.0 到 1.0 之间的浮点数。与 EffectsVolume 属性类似，对此属性的设置也会影响到所有背景音乐的音量大小。

void end()：当不再使用音频引擎时，调用此方法来释放 SimpleAudioEngine 所占用的资源。

6.4　小结

6.4　小结

在这一章中，我们学习了 Cocos2d-x 提供的音频引擎库 CocosDenshion。CocosDenshion 的核心是 SimpleAudioEngine，它提供了十分简单易懂的接口来控制音乐与音效的播放。下面我们回顾一下本章涉及的重要知识点。

- SimpleAudioEngine：一个单例类，提供了跨平台的音频回放功能。
- 音乐与音效：音乐是较长的音频文件，对格式限制较少，但通常一次只能播放一首音乐；音效是较短的音频文件，对格式限制严格，但是可以同时播放许多音效。
- 预加载：加载音频文件会消耗大量的时间，如果在需要播放音效时实时加载，会导致游戏出现卡顿现象，因此，我们使用 preloadEffect 与 preloadBackgroundMusic 方法来分别预加载音效与音乐文件。
- 音效唯一标识：在同一时刻可能播放着多个音效，因此，为了区分每一个音效，在调用 playEffect 方法播放音效时，会给即将播放的音效分配一个号码，即它的唯一标识。此后，如果需要暂停、恢复此音效的播放，或是停止播放此音效，都会使用其唯一标识来定位此音效。
- 最后，使用 SimpleAudioEngine 时，应注意以下三点。
- 播放音效或背景音乐前，一定要提前加载音效或背景音乐文件。
- 在播放背景音乐时，若要切换场景，不需要手动停止背景音乐，Cocos2d-x 会自动把先前场景的背景音乐停止，并播放新场景中的背景音乐（如果新场景会播放新的背景音乐的话）。
- 在退出后且不再需要音乐时，要调用 end 方法来释放引擎占用的资源。

7.1 触摸输入

- **用户输入**

- 经过前面几章的学习，我们的《捕鱼达人》已经具备很出彩的动态效果了。但是，到目前为止，除了几个简单的菜单按钮之外，我们的游戏更像是一部演示动画，玩家只能看到游动的鱼，却不能与游戏交互。因此，在这一章中，我们将集中探讨如何为游戏引入用户交户方式，使游戏与玩家真正互动起来。
- **7.1 触摸输入**
- 在还没有智能手机和平板电脑等便携设备时，我们在 PC 上使用鼠标与键盘来操作游戏。如今，这些操作方式已不再适用于所有游戏，取而代之的是大面积的触摸屏。用触摸屏交互看似与鼠标点击类似，它们都支持点按、拖曳等操作，然而它们还是有区别的：鼠标可以实现悬停效果，触摸屏不可以；鼠标只有一个焦点，而触摸屏大多支持多个触摸点；鼠标通常包含两个功能键从而区分左右键，而触摸屏只能实现一种点按操作。
- 为了处理屏幕触摸事件，Cocos2d-x 提供了非常方便、灵活的支持。在深入研究 Cocos2d-x 的触摸事件分发机制之前，我们利用 CCLayer 已经封装好的触摸接口来实现对简单的触摸事件的响应。

7.1.1 使用 CCLayer 响应触摸事件

- **7.1.1 使用 CCLayer 响应触摸事件**
- 为了处理屏幕输入事件，最简单的解决方案是利用 CCLayer 开启内建的触摸输入支持。在介绍 CCLayer 的时候提到过，它的一个十分重要的作用就是接收输入事件，因此层封装了触摸输入的处理接口。一般情况下，我们可以通过 TouchEnable 属性来开启或关闭接收触摸输入。
- 在开启了层的触摸输入支持后，我们就可以在层中处理触摸事件了。CCLayer 实现了以下 4 个方法，当引擎接收到触摸事件时，这些方法就会被调用：

```
virtual void ccTouchesBegan(CCSet *pTouches, CCEvent *pEvent);
virtual void ccTouchesMoved(CCSet *pTouches, CCEvent *pEvent);
virtual void ccTouchesEnded(CCSet *pTouches, CCEvent *pEvent);
virtual void ccTouchesCancelled(CCSet *pTouches, CCEvent *pEvent);
```

- 以上 4 个方法都声明为虚函数，这意味着我们通过重载的方式来处理用户输入事件。实际上，这 4 个函数来自于 CCStandardTouchDelegate 接口，是触摸事件的回调函数，分别对应触摸的开始、移动、结束和取消操作。传入参数 pTouches 是一个 CCTouch 对象（表示一个触摸点）的集合，其中包含当前触摸事件中的所有触摸点。传入参数 pEvent 是由 Cocos2d-iPhone 引擎遗留下来的形式参数，在 Cocos2d-x 当前版本中没有意义。
- 触摸事件分为 4 类，其中结束事件和取消事件容易使人困惑。通常，当玩家的触摸动作完成时（例如抬手和手指离开屏幕等），会引起触摸结束事件。而触摸取消的情况较少，仅当触摸过程中程序被调入后台时才会出现。
- 以移动事件为例，下面是事件处理函数的一种常见实现方法：

```
void TouchLayer::ccTouchesMoved(CCSet *pTouches, CCEvent *pEvent)
{
    if(pTouches->count() == 1) {
        CCTouch* touch = dynamic_cast<CCTouch*>(pTouches->anyObject());
        CCPoint position = touch->locationInView();
        position = CCDirector::sharedDirector()->convertToGL(position);
        //在此处理触摸事件
    }
    else {
        //如果不止一个触摸点，则在此处理多点触摸事件
    }
}
```

- 在响应函数中，我们从 pTouches 中取出表示触摸点的 CCTouch 对象，并利用触摸点信息完成一系列操作。
- 在这个例子中，我们首先判断了触摸点的数目，如果只有一个触摸点，则取出该触摸点，然后获取触摸点的位置。由于在不同平台下触摸点的坐标系与 OpenGL 呈现区域的参数可能不尽相同，所以触摸点的位置通常与系统相关。不过，Cocos2d-x 已经为我们处理好了这个问题，只需要仿照上面的代码，调用 CCTouch::locationInView 方法获取游戏画面中的点位置，然后再调用 CCDirector:: convertToGL 方法把屏幕坐标转换为游戏坐标，就可以获取触摸点在游戏中的位置了。
- 利用层来实现触摸十分简便，然而只要玩家触摸了屏幕，所有响应触摸事件的层都会被触发。当层的数量很多时，维护多个层的触摸事件就成了一件复杂的事情。因此，在实际开发中，我们通常单独建立一个触摸层。用触摸层来接收用户输入事件，并根据需要通知游戏中的其他部件来响应触摸事件。

● 7.1.2 两种 Cocos2d-x 触摸事件（1）

- 7.1.2 两种 Cocos2d-x 触摸事件（1）

- 我们已经见到了 CCLayer 处理触摸事件的方法。当我们开启 CCLayer 的 TouchEnable 属性后，层中的 4 个回调函数就会在接收到事件时被触发，我们把这类事件称作标准触摸（standard touch）事件，它的特点是任何层都会平等地接收全部触摸事件。
- 除此以外，Cocos2d-x 还提供了另一种被称作带目标的触摸事件（targeted touch）机制。在带目标的触摸机制中，接收者并不平等，较早处理事件的接收者有权停止事件的分发，使它不再继续传递给其他接收者。换句话说，带目标的触摸事件并不一定会被广播给所有的接收者。通常，游戏的菜单按钮、摇杆按钮等元件常使用目标触摸事件，以保证触摸事件不对其他层产生不良影响。

◆ 标准触摸事件

- 经过前面的学习，我们知道利用 CCLayer 可以方便地启用层上的标准触摸事件。实际上，并不是只有层才支持接收触摸事件，任何一个游戏元素都可以接受事件，只不过层中提供了现成的支持。下面我们将以层为例，来探究一下层是如何开启标准触摸事件的。

■ 为了开启层的标准触摸事件支持，我们需要启用 TouchEnable 属性。下面是 CCLayer 中与 TouchEnable 属性相关的代码，为了清晰，我们已经去掉了与脚本引擎相关的语句：

```
void CCLayer::setTouchEnabled(bool enabled)
{
    if (m_bIsTouchEnabled != enabled)
    {
        m_bIsTouchEnabled = enabled;
        if (m_bIsRunning)
        {
            if (enabled)
            {
                this->registerWithTouchDispatcher();
            }
            else
            {
                CCDirector* pDirector = CCDirector::sharedDirector();
                pDirector->getTouchDispatcher()->removeDelegate(this);
            }
        }
    }
}
void CCLayer::registerWithTouchDispatcher()
{
    CCDirector* pDirector = CCDirector::sharedDirector();
    pDirector->getTouchDispatcher()->addStandardDelegate(this,0);
}
```

■ 可以看到，如果设置开启触摸，则会调用 registerWithTouchDispatcher 方法。而在 registerWithTouchDispatcher 方法中，我们调用了 CCTouchDispatcher 的 addStandard- Delegate 方法。在引擎中，CCTouchDispatcher 负责触摸事件的分发处理，此处的 addStan- dardDelegate 方法会把当前对象注册到分发器中。被注册的对象必须实现 CCStandardTouch-Delegate 接口，当引擎从系统接收到触摸事件时，就会调用接口中对应的方法，触发触摸事件。

■ 相比之下，关闭触摸则简单得多：只需要调用 CCTouchDispatcher 的 removeDelegate 方法即可。

■ 因此可以总结，为了使一个对象接受标准触摸事件，主要有以下 4 个步骤。

- 需要此对象实现 CCStandardTouchDelegate 接口。
- 使用 addStandardDelegate 方法把自己注册给触摸事件分发器。
- 重载事件回调函数，处理触摸事件；
- 当不再需要接收触摸事件时，使用 removeDelegate 方法来注销触摸事件的接收。

■ 7.1.2 两种 Cocos2d-x 触摸事件（2）

◆ **带目标的触摸事件**

■ 我们可以为任意对象添加标准触摸事件，然而如同前文所述，标准触摸事件中存在两个较为不便的地方，具体如下所示。

- 只要事件分发器接收到用户的触摸事件，就会分发给所有的订阅者，因此常常会出现按下按钮时，触摸事件穿透按钮分发给后面的层这种尴尬的情况。
- 当系统存在多个触摸点时，标准触摸事件会把所有触摸点都传递给回调函数，然而在许多情况下每个触摸点之间是独立的，屏幕上是否存在其他触摸点我们并不不关心，因此我们不必为了处理多个触摸事件手动遍历一遍触摸点。
- 为此，Cocos2d-x 为我们提供了一个简化的解决方案：带目标的触摸事件。与标准触摸事件类似，我们也需要首先使接受事件的对象实现一个接口 CCTargetedTouchDelegate，然后把对象注册到触摸分发器中，最后当不再需要接受触摸事件时注销自身。
- 和 CCStandardTouchDelegate 接口类似，为了接受触摸事件，我们同样需要实现下面所示的 4 个回调函数：

```
virtual bool ccTouchBegan(CCTouch *pTouch, CCEvent *pEvent);
virtual void ccTouchMoved(CCTouch *pTouch, CCEvent *pEvent);
virtual void ccTouchEnded(CCTouch *pTouch, CCEvent *pEvent);
virtual void ccTouchCancelled(CCTouch *pTouch, CCEvent *pEvent);
```

- 细心的读者应该注意到了，这个接口和 CCStandardTouchDelegate 存在两处不同，这两处不同也对应着带目标的触摸事件的两个特点。
- 事件参数不再是集合，而是一次只传入一个触摸点。
- ccTouchBegin 方法返回一个布尔值，表示声明是否要捕捉这个触摸点，只有在此方法中捕捉到的触摸点才会继续引发其他 3 个事件，否则此触摸点的其他事件都会被忽略。
- 通过 CCTargetedTouchDelegate 的方式接收触摸事件，就无法直接利用 CCLayer 提供的属性了，因此 我们必须主动把自身注册到引擎的触摸分发器：

```
CCTouchDispatcher::sharedDispatcher()->addTargetedDelegate(this, 0, true);
```

- 在 addTargetedDelegate 方法中，前两个参数分别对应触摸接收对象和优先级，其中优先级是一个整型参数，值越低，则优先级越高，也就越早获得触摸事件。通常，为了获得较高的优先级，可以将其指定为负数。第三个参数较为有趣，表明了是否"吞噬"一个触摸，如果设置为 true，一个触摸一旦被捕捉，那么所有优先级更低的接收对象都无法接收到触摸。CCMenu 就是一个会"吞噬"且优先级为-128 的触摸接收器，由于它的优先级很高，所以菜单按钮总能获得触摸响应。
- 综上所述，带目标的触摸事件的使用步骤如下所示。

- 实现 CCTargetedTouchDelegate 接口。
- 使用 addTargetedDelegate 方法注册到触摸事件分发器。
- 重载事件回调函数。注意，我们必须在触摸开始事件中针对需要接受的事件返回 true 以捕捉事件。
- 当不再需要接受触摸事件时，使用 removeDelegate 方法来注销触摸事件的接收。与标准触摸事件相比，不同之处主要在于开始触摸事件需要返回一个代表是否捕捉事件的值。

7.2 触摸分发器原理

- 7.2 触摸分发器原理

- 前面我们详细介绍了触摸的基本模式。现在，我们已经可以利用 Cocos2d-x 提供的两种方式来处理触摸事件了。然而无论使用哪一种机制，最关键的一步都是把接收事件的对象注册到触摸分发器中，这是因为触摸分发器会利用系统的 API 获取触摸事件，然后把事件分发给游戏中接收触摸事件的对象。

■ 为了能理解并运用好触摸输入，我们有必要在这里先介绍一下负责触摸事件分发的 CCTouchDispatcher 类。与之相关的代码位于引擎目录下的"touch_dispatcher"目录中，有兴趣的读者可以仔细阅读。

■ 表 7-1 列举了 CCTouchDispatcher 中的主要成员，其中 addStandardDelegate 与 addTar- getedDelegate 两个方法接收一系列参数，包括事件的注册者。在这两个方法中，注册者会被对应地存入 m_pStandardHandlers 或 m_pTargetedHandlers 容器中。注销事件的方法与注册类似，同样操作上面提到的两个容器。然而在事件分发过程中，为了保证不在循环时改变容器的内容，我们引入了 m_pHandlersToAdd 与 m_ pHandlersToRemove 两个容器，用于暂时保存分发事件时目标对象的增删。此外，DispatchEvents 属性可以用于暂时屏蔽所有的触摸事件，这在一些特殊场合下十分方便。

■ 表 7-1 CCTouchDispatcher 的主要成员

● 类型	● 名称	● 描述
●	● addStandardDelegate	● 注册标准触摸事件
● 方法	● addTargetedDelegate	● 注册带目标的触摸事件，它拥有两个参数， ● 其中 bSwallowsTouches 参数表示是否会 ● “吞噬”捕捉到的触摸事件
	● removeDelegate	● 注销指定对象。注销后对象将不再会收到触摸事件
	● removeAllDelegates	● 删除所有的事件订阅
●	● setPriority	● 重新设置指定对象的事件优先级

■ （续）

● 类型	● 名称	● 描述
● 属性	● DispatchEvents	● 布尔类型，用于获取或设置事件分发器是否工作
● 私有字段	● m_pTargetedHandlers ● m_pStandardHandlers ● m_pHandlersToAdd ● m_pHandlersToRemove	● 一个数组，用于记录注册了带目标的触摸事件的对象 ● 一个数组，用于记录注册了标准触摸事件的对象 ● 临时数组，用于在分发器繁忙时段记录 ● 新注册触摸事件的对象 ● 临时数组，用于在分发器繁忙时段记录将 ● 要删除的注册触摸事件的对象

■ 以上提到的方法是引擎面向开发者而提供的接口。另一方面，事件分发器从系统接收到了触摸事件之后还需要逐一分发。分发事件的相关代码主要集中在 touches 方法之中。由于此方法较为冗长，因此我们对其进行了简化，简化后的版本 touches_simplified 基本保持了原来的框架，读者可以作为参考：

```
● void CCTouchDispatcher::touches_simplified(CCSet *pTouches, CCEvent *pEvent,
●     unsigned int uIndex)
● {
●     _prepare_for_dispatch();
●
●     //第一步：处理带目标的触摸事件
●     if (uTargetedHandlersCount > 0)
●     {
●         //遍历每一个接收到的触摸事件
●         for (setIter = pTouches->begin(); setIter != pTouches->end(); ++setIter)
●         {
●             //遍历每一个已注册触摸事件的对象
●             CCARRAY_FOREACH(m_pTargetedHandlers, pObj)
```

```
            {
                //分发不同类型的事件
                //首先处理触摸开始事件
                if(uIndex == CCTOUCHBEGAN) {
                    bClaimed = pHandler->getDelegate()->ccTouchBegan(pTouch, pEvent);
                    if (bClaimed)
                        pHandler->getClaimedTouches()->addObject(pTouch);
                }
                else if (pHandler->getClaimedTouches()->containsObject(pTouch))
                {
                    //再处理移动、结束和取消事件
                    bClaimed = true;
                    switch (sHelper.m_type)
                    {
                        case CCTOUCHMOVED:
                            pHandler->getDelegate()->ccTouchMoved(pTouch, pEvent);
                            break;
                        case CCTOUCHENDED:
                            pHandler->getDelegate()->ccTouchEnded(pTouch, pEvent);
                            pHandler->getClaimedTouches()->removeObject(pTouch);
                            break;
                        case CCTOUCHCANCELLED:
                            pHandler->getDelegate()->ccTouchCancelled(pTouch, pEvent);
                            pHandler->getClaimedTouches()->removeObject(pTouch);
                            break;
                    }
                }

                if (bClaimed && pHandler->isSwallowsTouches())
                {
                    //此对象被捕捉，而且设置了“吞噬触摸事件”属性
                    if (bNeedsMutableSet)
                        pMutableTouches->removeObject(pTouch);
                        break;
                }
            }
        }
    }

    //第二步：处理标准触摸事件
    if (uStandardHandlersCount > 0 && pMutableTouches->count() > 0)
    {
        //遍历每一个已注册触摸事件的对象
        CCARRAY_FOREACH(m_pStandardHandlers, pObj)
        {
```

```
            switch (sHelper.m_type)
            {
                case CCTOUCHBEGAN:
                    pHandler->getDelegate()->ccTouchesBegan(pMutableTouches, pEvent);
                    break;
                case CCTOUCHMOVED:
                    pHandler->getDelegate()->ccTouchesMoved(pMutableTouches, pEvent);
                    break;
                case CCTOUCHENDED:
                    pHandler->getDelegate()->ccTouchesEnded(pMutableTouches, pEvent);
                    break;
                case CCTOUCHCANCELLED:
                    pHandler->getDelegate()->ccTouchesCancelled(pMutableTouches, pEvent);
                    break;
            }
        }
    }

    _process_handler_to_remove();
    _process_handlers_to_add();

    _dispose_unused_resources();
}
```

- 可以看到，分发过程遵循以下的规则。
- 对于触摸集合中的每个触摸点，按照优先级询问每一个注册到分发器的对象。对于同一优先级的对象，访问顺序并不确定。
- 当接收到开始事件时，如果接受者返回 true，则称对象捕捉了此事件。只有被捕捉，后续事件（如移动、结束等）才会继续分发给目标对象。
- 如果设置了吞噬属性，则捕捉到的点会被吞噬。被吞噬的点将立即移出触摸集合，不再分发给后续目标（包括注册了标准触摸事件的目标）。
- 将没有被吞噬的触摸点集按优先级的顺序分发给每一个注册了标准触摸时间的目标对象，同一优先级之间对象的访问顺序并不确定。
- 为了避免事件分发中事件处理对象被改变，Cocos2d-x 仔细维护了两个临时表，因此开发者无论何时都可以注册或注销触摸事件。

■ 7.3 触摸中的陷阱

- 触摸处理机制的规则并不复杂，但是存在一些容易让人产生误解的陷阱。
- 第一个陷阱是接收触摸的对象并不一定正显示在屏幕上。触摸分发器和引擎中的绘图是相互独立的，所以并不关心触摸代理是否处于屏幕上。因此，在实际使用中，应该在不需要的时候及时从分发器中移除触摸代理，尤其是自定义的触摸对象。而 CCLayer 也仅仅会在切换场景时将自己从分发器中移除，所以同场景内手动切换 CCLayer 的时候，也需要注意禁用触摸来从分发器移除自己。

■ 另一个陷阱出自 CCTargetedTouchDelegate。尽管每次只传入一个触摸点，也只有在开始阶段被声明过的触摸点后续才会传入，但是这并不意味着只会接收一个触摸点：只要被声明过的触摸点都会传入，而且可能是乱序的。因此，一个良好的习惯是，如果使用 CCTargeted- TouchDelegate，那么只声明一个触摸，针对一个触摸作处理即可。

● 7.4.1 使炮台动起来（1）

■ **7.4 使用触摸事件**

■ 经过前面的学习，我们对 Cocos2d-x 的触摸事件已经十分了解了。下面我们将利用前面介绍的知识来完善我们的游戏。

■ **7.4.1 使炮台动起来（1）**

■ 我们已经知道，通过触摸事件可以控制发炮和炮台的转动。为此，我们新建一个触摸处理层 TouchLayer，它继承自 CCLayer，负责将基本的触摸事件转换为我们感兴趣的事件。TouchLayer 的定义如下：

```
class TouchLayer : public CCLayer
{
public:
    CC_SYNTHESIZE(TouchLayerDelegate*,m_pDelegate,Delegate);
public:
    bool init();
    virtual void ccTouchesBegan(CCSet *pTouches, CCEvent *pEvent);
    virtual void ccTouchesMoved(CCSet *pTouches, CCEvent *pEvent);
    virtual void ccTouchesEnded(CCSet *pTouches, CCEvent *pEvent);
    virtual void ccTouchesCancelled(CCSet *pTouches, CCEvent *pEvent);
    void onExit();
};
```

■ 现在我们添加代码使 TouchLayer 拥有接受触摸事件的能力。和 CCLayer 中开启触摸事件的方法类似，我们在 init 中调用 registerWithTouchDispatcher 方法，把 TouchLayer 注册到触摸分发器中，以便接受玩家的触摸事件。与此对应地，当场景退出屏幕，触发了 onExit 回调函数时，我们需要从触摸分发器中移除这个注册。这两个方法的代码如下所示：

```
bool TouchLayer::init()
{
    bool bRet = false;
    do {
        CC_BREAK_IF(!CCLayer::init());

        this->registerWithTouchDispatcher();
        this->setDelegate(NULL);
        bRet = true;
    } while (0);

    return bRet;
}

void TouchLayer::onExit()
{
    CCDirector::sharedDirector()->getTouchDispatcher()->removeDelegate(this);
}
```

- 在这段代码中，我们看到了 Delegate 这个成员变量，它是 TouchLayerDelegate 类型的变量，代表一个处理触摸事件的"代理"。TouchLayer 负责接收触摸事件，并把事件分发给代理，由代理来处理事件。使用代理的设计模式，可以解除触摸层和实际处理事件的精灵层间的耦合。后面会看到，当触摸层渐渐变复杂之后，这样的解耦合能有效分离触摸手势的识别和事件处理两部分，代码结构清晰也不易出错。

- 在我们的例子中，精灵层需要响应两个触摸事件，分别为发射炮弹与转动炮台。对应地，我们在 TouchLayerDelegate 中定义了两个触摸事件：singleTouchEndsIn 与 singleTouchDirecting，分别表示单点触摸的结束与移动事件。然后在精灵层中继承并实现 TouchLayerDelegate。当单点触摸结束时，向触摸的方向发射一枚炮弹；当单点触摸移动时，改变炮台的方向，使之指向触摸的位置。

7.4.1 使炮台动起来（2）

- 7.4.1 使炮台动起来（2）

- TouchLayerDelegate 协议类与精灵层的实现代码如下：

```
class TouchLayerDelegate
{
public:
    virtual void singleTouchEndsIn(CCPoint point) = 0;
    virtual void singleTouchDirecting(CCPoint point) = 0;
};

void SpriteLayer::singleTouchEndsIn(CCPoint point)
{
    CCLog("%f %f", point.x, point.y);
    this->fire(point);
}

void SpriteLayer::singleTouchDirecting(CCPoint point)
{
    CCPoint origin = cannon->getPosition();
    CCPoint direction = ccp(point.x - origin.x, point.y - origin.y);
    float degree = (float)atan2(direction.y, direction.x);
    cannon->setRotation(90 - degree * 180 / acos(-1.0));
}
```

- 我们已经完成了精灵层对触摸事件的响应。现在我们需要做的是，在 TouchLayer 接收到系统触摸事件之后，把事件分发给代理（也就是精灵层）。在 TouchLayer 的 4 个触摸事件响应函数中，我们首先检测触摸点数量，如果只有一个触摸点，则判断为单点触摸，并调用代理处理对应的事件。相关的代码如下所示：

```
void TouchLayer::ccTouchesBegan(CCSet *pTouches, CCEvent *pEvent)
{
    if(pTouches->count() == 1) {
        CCTouch* touch = dynamic_cast<CCTouch*>(pTouches->anyObject());
        CCPoint position = touch->locationInView();
        position = CCDirector::sharedDirector()->convertToGL(position);
        if(this->getDelegate())
            this->getDelegate()->singleTouchDirecting(position);
```

```
    }
}
void TouchLayer::ccTouchesMoved(CCSet *pTouches, CCEvent *pEvent)
{
    if(pTouches->count() == 1) {
        CCTouch* touch=dynamic_cast<CCTouch*>(pTouches->anyObject());
        CCPoint position = touch->locationInView();
        position = CCDirector::sharedDirector()->convertToGL(position);
        if(this->getDelegate())
            this->getDelegate()->singleTouchDirecting(position);
    }
}
void TouchLayer::ccTouchesEnded(CCSet *pTouches, CCEvent *pEvent)
{
    if(pTouches->count() == 1){
        CCTouch* touch = dynamic_cast<CCTouch*>(pTouches->anyObject());
        CCPoint position = touch->locationInView();
        position = CCDirector::sharedDirector()->convertToGL(position);
        if(this->getDelegate())
            this->getDelegate()->singleTouchEndsIn(position);
    }
}
```

- 最后，我们把触摸层加入到场景之中，并把精灵层设为它的代理。在 GameScene::init 方法中添加相关的初始化代码，具体如下所示：

```
TouchLayer* touchLayer = TouchLayer::node();
touchLayer->setDelegate(spriteLayer);
this->addChild(touchLayer);
```

- 如果一切顺利，现在我们的炮台就可以随着单手指的移动而改变瞄准方向，并在手指离开屏幕时开炮（如图 7-1 所示）。

图7-1 运行效果

- 7.4.2 识别简单的手势

■ 7.4.2 识别简单的手势

■ 触屏设备通常只包含一个很大的面板，这意味着我们缺少足够多的实体按键来实现一些常用的功能。为此，我们可以给游戏添加一些手势操作的功能，玩家可以在屏幕上做出方便的手势而不是去菜单里逐层寻找功能。

■ 下面我们来为《捕鱼达人》游戏添加一些简单的手势操作。我们定义两个手势：用三个手指的左右拖动来切换炮，四个手指上下拖动来暂停和继续游戏。

■ 在 TouchLayer 中，我们添加如下的两个成员变量来记录三点触摸的开始：

```
CCPoint firstTripleTouches[3];
bool didTripleTouch;
```

■ 在触摸拖动的回调函数中，我们添加以下代码以记录此时的三个点：

```
if(pTouches->count() == 3) {
    didTripleTouch = true;
    int i = 0;
    for (CCSetIterator it = pTouches->begin(); it != pTouches->end(); ++it, ++i) {
        CCTouch* touch=dynamic_cast<CCTouch*>(*it);
        CCPoint position = touch->locationInView();
        firstTripleTouches[i] = CCDirector::sharedDirector()->convertToGL(position);
    }
}
```

■ 考虑到这仅仅是为了演示触摸处理，这里我们并没有严格检测三指拖动的过程是否遵循直线且平行等细节，只是在触摸结束时做了简单的判断。这里要求三个点的横坐标差都超过 100 像素，这是为了避免短的错误触摸带来误操作。修改后的代码如下所示：

```
if(pTouches->count() == 3 && didTripleTouch) {
    int i = 0;
    int direction[3] = { 0, 0, 0 };

    for (CCSetIterator it = pTouches->begin(); it != pTouches->end(); ++it, ++i) {
        CCTouch* touch = dynamic_cast<CCTouch*>(*it);
        CCPoint position = touch->locationInView();
        position = CCDirector::sharedDirector()->convertToGL(position);

        float dis = (position.x-firstTripleTouches[i].x);
        if(dis > 100)
            direction[i] = 1;
        else if(dis < -100)
            direction[i] = -1;
    }

   if(this->getDelegate() && direction[0] != 0 &&
        direction[0] == direction[1] && direction[1] == direction[2]) {
        this->getDelegate()->tripleTouch(direction[0] < 0);
```

```
    }
}
```

- 最后，在精灵层中添加切换炮台的处理，相关代码如下所示：

```
void SpriteLayer::tripleTouch(bool moveLeft)
{
    if(moveLeft)
        curCannonLevel --;
    else
        curCannonLevel ++;

    curCannonLevel = ((curCannonLevel -1) + 7) % 7;
    this->switchCannon(curCannonLevel);
}
```

- 多点触控在 OS X、Windows 7 及之前版本的平台上是暂时不支持的，想尝试这部分代码的同学请参考附录 A 将引擎部署到 Windows 8、iOS 或 Android 平台上来。
- 而四指切换的效果是类似的，代码就不再赘述了。
- 感兴趣的读者可以思考一个问题：是否存在这样的可能，抽象出一个单点触摸管道，反应一个触摸的触摸轨迹。这样不管三指还是四指的任何方向的拖动，都只要检测是否存在相应数量的触摸管道检测到触摸，且触摸轨迹是类似的。实际上，这正是 CCTargetedTouchDelegate 的作用，但如何处理几个不同的触摸管道间的互相接触会是很有趣的挑战。

7.5 加速度计（1）

7.5 加速度计（1）

- 除了触摸，移动设备上一个很重要的输入来源是设备的方向，因此大多数设备都配备了加速度计，用于测量设备静止或匀速运动时所受到的重力方向。
- 目前，加速度计在传统 PC 平台上尚未提供支持，因此在正式开始探索重力感应之前，还请读者参考附录 A，将 Cocos2d-x 工程部署到手机或其他移动设备上。现在，Android 或 iOS 平台均支持加速度计。
- 重力感应来自于移动设备的加速度计，通常支持 X、Y 和 Z 三个方向的加速度感应，所以又称为三向加速度计。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。
- 和触摸事件一样，重力感应的处理先被引擎抽象为一个触摸代理协议，然后由 CCLayer 提供了一个简化的实现，屏蔽了注册到重力感应器等一系列细节。在开发中，我们只需要重载一个简单的事件即可：

```
virtual void didAccelerate(CCAcceleration* pAccelerationValue);
```

- 其中 CCAcceleration 是一个结构体，包含了加速度计获得的 3 个方向的加速度，相关代码如下：

```
typedef struct
{
    double x;
    double y;
    double z;
    double timestamp;
} CCAcceleration;
```

- 为了便于游戏开发的使用，这个结构体中每个方向的加速度大小都以一个重力加速度为单位（即 CCAcceleration 结构中 1 单位的加速度等于重力加速度约 9.8 m/s2）。举例来说，把手机平放在桌子上时，获得的加速度应该是（0，1，0）。正常使用时，总的加速值应该在 1 上下拨动。如果检测到一个大幅度偏离 1 的加速度时，则可以判断为突然动作：摇动手机时，会在一个或多个方向上出现很大的加速；投掷或坠落则容易检测到很小的加速。我们可以利用这些特性为游戏添加一些很有趣的控制效果。

- 我们可以尝试引入两个新特性：根据手机的倾斜方向转动炮台的方向，摇动手机来开火。和处理触摸事件时类似，我们建立一个加速检测层，将检测到的事件交由代理处理，相关代码如下：

```
class AccelerationLayerDelegate
{
public:
    virtual void swing(int direction)=0;
    virtual void shakes()=0;
};

class AccelerationLayer : public CCLayer
{
    int swingCnt; //记录连续的倾斜
    int shakeCnt; //记录连续的摇晃
public:
    CC_SYNTHESIZE(AccelerationLayerDelegate*, m_pDelegate, Delegate);
public:
    LAYER_NODE_FUNC(AccelerationLayer);
    bool init();
    virtual void didAccelerate(CCAcceleration* pAccelerationValue);
};
```

- 这里我们使用了两个成员变量来记录摇晃和倾斜的次数，其作用为对重力感应事件产生缓冲，从后面的处理函数可以看到我们做了三个处理。

- 首先，需要区分正常加速和用力摇晃设备，它们的区别就在于总加速度是否异常。我们设定用力摇晃设备时的总加速度大小应该大于 2 倍重力加速度，这样的处理忽略了手机在下落中处于失重状态，因为通常我们只会左右摇晃设备。

- 7.5 加速度计（2）

- 第二个处理是只有连续几次检测到摇晃时才确认进入摇晃，而摇晃事件的通知是在摇晃结束时通知到代理的。 最后，则是对于倾斜状态下加速度的处理，相关代码如下：

```
void AccelerationLayer::didAccelerate(CCAcceleration* pAccelerationValue)
{
    CCLOG("%f,%f,%f", pAccelerationValue->x, pAccelerationValue->y,
        pAccelerationValue->z);

    if (this->getDelegate()) {
        float x = pAccelerationValue->x,
              y = pAccelerationValue->y,
              z = pAccelerationValue->z;
```

```
        float all = x*x + y*y + z*z;

        if(all < 1.2) {
            int direction = pAccelerationValue->x / 0.4;
            if(direction > 3) direction = 3;
            if(direction < -3) direction = -3;

            if(shakeCnt >2)
                this->getDelegate()->shakes();
            swingCnt++; shakeCnt = 0;

            if(swingCnt >5)
                this->getDelegate()->swing(direction);
        }
        else if(all > 2) {
            shakeCnt++;
            swingCnt = 0;
        }
    }
}
```

■ 初始化函数做的事情比较少，只需要注册加速度计事件和清空事件计数器即可，相关代码如下：

```
bool AccelerationLayer::init()
{
    bool bRet = false;
    do {
        CC_BREAK_IF(!CCLayer::init());

        this->setIsAccelerometerEnabled(true);
        shakeCnt = swingCnt = 0;
        bRet = true;
    } while (0);

    return bRet;
}
```

■ 当我们完成了加速度计事件的处理后，就会通过代理的模式把事件分发给精灵层。因此在精灵层中，需要响应 swing 和 shakes 两个事件。具体来说，当 swing 事件触发时，精灵层需要根据设备倾斜的角度来设置炮台的旋转角度，而 shakes 事件触发时，触发炮弹发射逻辑。注意，在旋转炮台时我们用了动作来设置旋转，而不是直接设置角度，可以避免一个突然的旋转造成动作生硬。精灵层中相关的代码如下：

```
void SpriteLayer::swing(int direction)
{
    cannon->runAction(CCRotateBy::create(0.1, 10 * direction));
}
void SpriteLayer::shakes()
{
```

```
    float r = cannon ->getRotation();
    const float pi = acos(-1.0);

    CCPoint direction = ccp(sin(r / 180 * pi), cos(r / 180 * pi));
    CCPoint from = cannon->getPosition();
    CCPoint to = ccp(from.x + direction.x * 300, from.y + direction.y * 300);

    this->fire(to);
}
```

- 最后如同添加触摸层的时候一样，我们需要把加速度计层添加到游戏场景之中，并设置精灵层为它的代理对象，相关的代码就略去不说了。

7.6 文字输入

7.6 文字输入

- 最后，值得一提的是文字输入。正如鼠标对键盘的颠覆，从 iPhone 开始的触摸屏革命已然将实体键盘逼入了一个尴尬的位置，更多的交互通过触摸屏上直观的手势操作完成，但是文字输入依然是必不可少的，例如当我们需要设定用户名的时候。
- 在没有实体键盘的触屏手机上，文字输入通过虚拟键盘来实现。不幸的是，CCLayer 并没有为我们提供一个简化的封装实现；而幸运的是，在 Cocos2d-x 中已经存在一个现成的文本框控件 CCTextFieldTTF 了。尽管这个控件的接口稍微有些复杂，但使用起来并不麻烦。它的外观为屏幕上一块可输入的区域，如同各种平台下的文本框一样，点击后可弹出键盘。
- CCTextFieldTTF 提供了 3 个常用的操作接口，具体如下所示：

```
const char* getString(void);     //获取当前字符串
bool attachWithIME();            //激活输入法
bool detachWithIME();            //取消激活输入法
```

- 这 3 个方法基本可以满足开发需要了。如果我们在开始场景中添加一个让用户输入用户名的文本框，只需要在 StartScene 类的初始化代码中添加一个 CCTextFieldTTF 控件即可，相关代码如下：

```
CCTextFieldTTF text = CCTextFieldTTF::textFieldWithPlaceHolder(
    "Input Your Name...", "Arial", 20);
text->setPosition(ccp(winSize.width / 2, winSize.height / 2 + 40));
this->addChild(text);
text->attachWithIME();
```

- 其中创建 CCTextFieldTTF 时的第一个参数为提示文本，在文本框中没有输入任何内容时提示用户输入，后面的两个参数分别对应字体和字号。值得注意的是，截至 Cocos2d-x 2.0.1 版本，CCTextFieldTTF 类的工厂函数还在使用旧的命名风格，即它还没有提供"create"风格的工厂函数。
- 这么做在 Windows 等含键盘的平台上可以很好地工作，但在手机等移动终端上就有些不足了。移动设备通常没有实体键盘，在不需要输入文字时，游戏的画面可以完整地显示在屏幕之中，然而当需要输入文字时，就会从屏幕下方弹出一个面积很大的虚拟键盘，挡住游戏画面的很大一部分。为了节省屏幕空间，我们采取的做法是在一开始将输入法隐藏，在输入文本框上添加一个空白按钮，在点击按钮时再弹出输入法。
- 同样，在 StartScene 类的初始化中添加这个按钮，相关代码如下：

```
CCMenuItem* tapItem = CCMenuItemFont::
    create("        ",this,menu_selector(StartScene::textFieldPressed));
```

```
tapItem->setPosition(ccp(winSize.width / 2, winSize.height / 2 + 40));
menu->addChild(tapItem, 1);
```

- 并且将显示输入法放到对应的响应函数中，具体如下所示：

```
void StartScene::textFieldPressed(cocos2d::CCObject *sender)
{
    text->attachWithIME();
}
```

- 这也是一个常用的技巧。当要求点击屏幕的某些位置能产生响应时，单独实现一个触摸协议层过于烦琐，尤其是在考虑相对位移关系的时候，此时可以考虑用一个空白内容的菜单按钮完成。
- 然而，这样处理仍然存在一个不足，弹出的虚拟键盘会将屏幕的下半部分盖住。在很多情景下，如果文本框恰好在屏幕的下半部分，会直接被弹出的键盘覆盖，用户无法正常输入文字。我们应该在输入法弹出和收起的时候调整屏幕布局到一个合适的位置。
- CCTextFieldTTF 预留了一个代理 CCTextFieldDelegate 协议来通知相关事件。这个代理协议实际上是输入法代理协议的简化封装，其中包含了文本输入框的 5 个常用事件，具体如下所示：

```
//即将激活输入法，如果不想激活，应该返回 true
virtual bool onTextFieldAttachWithIME(CCTextFieldTTF * sender);
//即将取消输入法，如果不想取消，应该返回 true
virtual bool onTextFieldDetachWithIME(CCTextFieldTTF * sender);
//即将插入一段文字，如果不想插入，应该返回 true
virtual bool onTextFieldInsertText(CCTextFieldTTF * sender, const char * text, int nLen);
//即将删除一段文字，如果不想删除，应该返回 true
virtual bool onTextFieldDeleteBackward(CCTextFieldTTF * sender, const char * delText,
    int nLen);
//如果不希望绘制这个输入法，返回 true
virtual bool onDraw(CCTextFieldTTF * sender);
```

- 实现了 CCTextFieldDelegate 协议的类可以获取输入框的以上 5 个事件。我们就可以利用其中的前两个接口来重新布局屏幕。为 StartScene 增加对 CCTextFieldDelegate 协议的继承，并实现如下两个函数：

```
bool StartScene::onTextFieldAttachWithIME(cocos2d::CCTextFieldTTF *sender)
{
    this->setPosition(ccp(0, 100));
    return false;
}
bool StartScene::onTextFieldDetachWithIME(cocos2d::CCTextFieldTTF *sender)
{
    this->setPosition(ccp(0, 0));
    return false;
}
```

- 在我们的例子中，当弹出键盘时，整个场景会向上移动 100 像素，键盘收起时则恢复原样，这保证了开始按钮和文本输入框都始终在屏幕可视范围的正中间。这在 Windows 平台上感觉不明显，但在手机等终端上会有非常大的体验改善。
- 最后，不要忘记设置文本输入框和协议响应方，相关代码如下：

- `text->setDelegate(this);`

一切顺利并部署到手机上后，我们应该可以看到图 7-2 所示的画面。

图7-2 iOS下的运行效果

7.7 小结

7.7 小结

在这一章中，我们主要探索了引擎中与用户输入相关的功能特性。Cocos2d-x 既提供了充分利用移动设备特点的多点触摸输入与加速度计支持，也提供了更高层次的文字输入封装。利用这些功能，我们实现了发射炮弹等功能。下面总结一下这一章的重要知识点。

CCLayer 触摸支持:CCLayer 提供了一套现成的触摸实现,只需启用 TouchEnabel 属性即可接收 Cocos2d-x 的标准触摸事件。开发者需要重载以 ccTouches 为前缀的响应

函数。

标准触摸事件：Cocos2d-x 提供的一种触摸事件，注册了这种事件的所有对象都会平等地接收到触摸事件，并且对于多点触摸，标准触摸事件会同时提供所有的触摸点。

带目标的触摸事件：这是另一种触摸事件，注册了这种事件的对象会按照优先级投递，先接收到事件的对象有权吞噬掉此事件，以阻止把事件分发给其他对象。我们常常利用吞噬功能屏蔽某个事件，例如，一个提示框可以屏蔽掉提示框以外的所有对象接收到触摸事件。此外，在多个触摸点的情况下，带目标的触摸事件会逐个触摸点分发，这意味着每一次接收到的触摸事件只包含一个点的信息。

触摸分发器（CCTouchDispatcher）：由引擎调度、负责接收来自系统的触摸事件。这允许任何对象在分发器中注册触摸事件,并根据 Cocos2d-x 的机制把触摸事件分发给注册者。开发者可以利用它的 addStandardDelegate 与 addTargetedDelegate 方法把自己注册为事件接收者，利用 removeDelegate 方法取消曾经注册过的触摸事件。

加速度计（CCAcceleration）：加速度计用于指示当前机器所受到的力。由于在地球上我们一定会受到重力的作用，因此在静止状态下我们可以根据加速度计来判断重力方向，实现重力感应。CCLayer 提供了加速度计事件的支持，把

IsAccelerometerEnabled 属性设置为 true 来启用触摸设置，然后可以通过重载 didAccelerate 事件响应函数来接收加速度计的数据。

- 至此，我们的《捕鱼达人》第一版也算是有声有色地正式出炉了。回顾一下，对于如何使用 Cocos2d-x 引擎开发一个游戏，读者已经有了初步的了解。在进入下一部分之前，再一次建议读者先放下这本书。我们已经对引擎的整体有了一个感性的认识，那么此时不妨仔细浏览引擎为我们提供的测试样例。测试样例位于引擎的"test"目录中，Cocos2d-x 社区针对引擎的每一个组件都设计了对应的样例代码。阅读这些代码，不仅能看到引擎都有什么功能，还可以学习每个功能的标准使用方法，既是温故也是知新，为接下来探索引擎的高级特性打好基础。

● 8.1 Cocos2d-x 中的粒子系统（1）

- **粒子效果**

  - 在大自然中，随处可见一些大规模运动的物体，例如下雨时的雨点、下雪时的雪花、爆炸时的火花，甚至旋转的星系、扩散的云雾等。当我们希望在游戏中模拟这些大规模运动的物体时，通常有如下两种方法。
  - 使用帧动画来模拟。设计帧动画并把它渲染为图片序列来模拟特效，不但生成的动画体积庞大，也无法调整其运动参数，因此有失灵活性。
  - 本章即将介绍的粒子效果。我们把每一个对象看做一个粒子，赋予它们一定的属性（例如外观、位置、速度、加速度和生存时间等），使它们按照一定的规律产生、运动并最终消失。
  - 在粒子效果中，通常存在一个对所有粒子进行统一调度的引擎，称作粒子系统（partical system），它负责粒子的产生，随时间改变粒子的状态，以及最后回收不再需要的粒子。如果按照粒子系统的维数来区分，粒子系统可以分为二维粒子系统与三维粒子系统两种。下面我们先来介绍一下 Cocos2d-x 中的粒子系统。
  - **8.1 Cocos2d-x 中的粒子系统（1）**
  - Cocos2d-x 为我们提供的粒子系统由 CCParticleSystem 类实现。与其他的粒子引擎一样，CCParticleSystem 实现了对粒子的控制与调度，对粒子的操作包括如下几种。
  - 产生粒子：这部分也被称作粒子发射器（emitter）。
  - 更新粒子状态：引擎会随时间更新粒子的位置、速度以及其他状态。
  - 回收无效粒子：当粒子的生存周期结束后，就会被系统回收。
  - 因此，为了创建一个粒子效果，我们需要定义粒子如何产生以及状态如何改变。
  - CCParticleSystem 提供了多种初始化方式。我们可以通过指定粒子数量来创建一个粒子系统，然后需要设置粒子的外观（通常为一张小纹理）、发射方式与运动方式。创建一个全新的粒子系统通常较为烦琐，大多数情况下，我们更乐意把粒子系统的参数保存在文件中，而 Cocos2d-x 就是使用 Plist 文件来保存这些参数的。暂时抛开粒子效果文件 Plist 不谈，如果我们已经拥有一个粒子效果文件，就可以利用 CCParticleSystem 的初始化方法直接从文件中导入一个粒子效果，相关代码如下：

- `bool initWithFile(const char *plistFile)`
- `static CCParticleSystem* create(const char *plistFile)`

  - Plist 文件实质上是一个 XML 文件，我们可以利用任何文本编辑器来创建或修改。为了创建一个新的粒子效果，我们可以从 Cocos2d-x 的测试目录下找到一个现有的粒子系统 Plist 文件（这些文件位于 Cocos2d-x 引擎测试工程目录下的"Resources/Particles"目录中）并打开和修改。

- 实际上，引擎已经内置了若干粒子效果，它们作为粒子系统的样例，只需要简单的几行代码就可以创建，正如 CCAction 为我们做的一样。这些粒子效果的代码放在 Cocos2d-x 引擎目录下"particle_nodes"目录中的"CCParticleSystem.cpp"文件中。在此，我们简单地列举了引擎提供的样例粒子效果，如表 8-1 所示。
- 表 8-1　内置的几种粒子效果

名称	说明
CCParticleFire	火焰效果
CCParticleSun	太阳效果
CCParticleExplosion	爆炸效果
CCParticleSnow	雪花效果

- 粒子系统继承自 CCNode，可以被添加到其他节点之中。在游戏中显示一个粒子效果十分简单。直接把下面的代码添加到游戏场景的初始化方法中，给游戏场景添加一个雪花效果，即可营造一种冬日氛围。准备好一张雪花图片 snow.png，并在 GameScene::init 方法中添加以下代码：

```
CCParticleSystem *particle = CCParticleSnow::node();
particle->setTexture(CCTextureCache::sharedTextureCache()->addImage("snow.png"));
this->addChild(particle);
```

- 成功添加后，就可以看到如图 8-1 所示的雪花漫天飘舞的效果了。

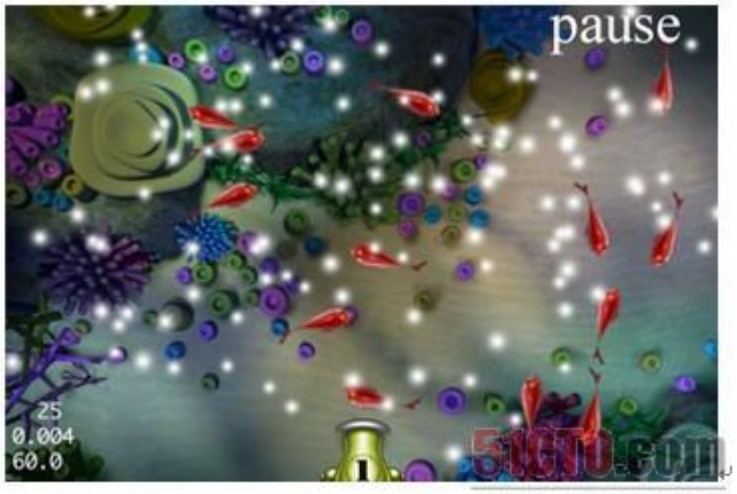


图8-1　运行效果

8.1　Cocos2d-x 中的粒子系统（2）

- 在上述代码中，第二条语句指定了粒子所使用的纹理，即"snow.png"。对于任何粒子系统，Texture 都是一个必须设置的属性，粒子系统中的每一个粒子都使用此纹理渲染出来。在这个例子中，每一个粒子都被赋予雪花纹理，因此可以呈现出雪花飘落的效果。
- 前面我们并没有详细介绍如何创建一个全新的粒子效果，这是由于创建一个全新的粒子效果需要设定的参数过于繁杂，并且为了使呈现效果与预期相符，我们不得不反复修改参数。这是一个十分笨拙的方法。我们完全可以利用引擎内置的粒子示例来实现一个粒子效果：首先找到一个与期望效果类似的粒子效果，然后修改纹理，微调参数。下面我们以 CCParticleSnow 为例，展示它是如何设置参数的：

```
bool CCParticleSnow::initWithTotalParticles(unsigned int numberOfParticles)
```

```
{
    if( CCParticleSystemQuad::initWithTotalParticles(numberOfParticles) )
    {
        //时间间隔
        m_fDuration = kCCParticleDurationInfinity;

        //设置为重力模式
        m_nEmitterMode = kCCParticleModeGravity;

        //重力模式参数: 重力
        modeA.gravity = ccp(0,-1);

        //重力模式参数: 粒子速度
        modeA.speed = 5;
        modeA.speedVar = 1;

        //重力模式参数: 径向加速度
        modeA.radialAccel = 0;
        modeA.radialAccelVar = 1;

        //重力模式参数: 切向加速度
        modeA.tangentialAccel = 0;
        modeA.tangentialAccelVar = 1;

        //粒子发射器位置
        CCSize winSize = CCDirector::sharedDirector()->getWinSize();
        this->setPosition(ccp(winSize.width/2, winSize.height + 10));
        m_tPosVar = ccp( winSize.width/2, 0 );

        //角度
        m_fAngle = -90;
        m_fAngleVar = 5;

        //粒子的生命时间
        m_fLife = 45;
        m_fLifeVar = 15;

        //尺寸（以像素为单位）
        m_fStartSize = 10.0f;
        m_fStartSizeVar = 5.0f;
        m_fEndSize = kCCParticleStartSizeEqualToEndSize;

        //每秒发射粒子数
        m_fEmissionRate = 10;

```

```
        //粒子着色
        m_tStartColor.r = 1.0f;
        m_tStartColor.g = 1.0f;
        m_tStartColor.b = 1.0f;
        m_tStartColor.a = 1.0f;
        m_tStartColorVar.r = 0.0f;
        m_tStartColorVar.g = 0.0f;
        m_tStartColorVar.b = 0.0f;
        m_tStartColorVar.a = 0.0f;
        m_tEndColor.r = 1.0f;
        m_tEndColor.g = 1.0f;
        m_tEndColor.b = 1.0f;
        m_tEndColor.a = 0.0f;
        m_tEndColorVar.r = 0.0f;
        m_tEndColorVar.g = 0.0f;
        m_tEndColorVar.b = 0.0f;
        m_tEndColorVar.a = 0.0f;

        //禁用线性叠加混合模式
        this->setBlendAdditive(false);
        return true;
    }
   return false;
}
```

8.2 粒子效果编辑器

8.2 粒子效果编辑器

Particle Designer 是 71 Squared 公司开发的粒子效果编辑器，运行于 OS X 系统之下，用于生成 Cocos2d、Sparrow、Starling 和 MOAI 等引擎可用的粒子系统文件。在这些引擎中，我们可以利用 Particle Designer 在调整参数的同时查看预览。Particle Designer 是收费软件，可以在 71 Squared 网站（http://particledesigner.71squared.com）上购买。

下面我们从界面开始介绍 Particle Designer 的使用方法。

8.2.1 界面介绍（1）

8.2.1 界面介绍（1）

打开 Particle Designer 后，首先出现的是如图 8-2 所示的界面，其中包括一个主界面和一个 iPhone 模拟器，当前编辑中的粒子效果就会在模拟器中显示实时预览。下面介绍主界面的基本功能。

图8-2 Particle Designer界面

首先看到的是工具栏（如图 8-3 所示），其中各个按钮（按照从左至右的顺序）的功能如下所示。

图8-3 工具栏

- Load、Save 和 Save As：载入或保存粒子效果，产生的粒子系统文件后缀为".pex"。
- iPad：改变模拟器的形态，当前状态下显示为 iPhone 模拟器，按下该按钮后切换为 iPad 模拟器，再次按下后切换为 iPhone 模拟器。
- Orientation：旋转设备方向（即水平或竖直方向）。
- Ramdomize：随机产生粒子效果。
- Pause：暂停播放粒子动画。按下后变为 Play 按钮，再次按下会继续播放粒子动画。
- Loop Emitter：设置是否循环播放。
- Emitter Config：切换至参数设置页面。切换至参数设置页面后，变为 Shared Emitters 按钮，用于设置页面和预置效果页面之间的切换。
- 在介绍完主要按钮的作用后，我们来看预置效果页面（即图 8-2 中的主界面），其中显示了很多预置效果，单击它们可以预览效果。如果要自己制作粒子效果，一个很好的方法是选择一个近似的预置效果，再进入参数设置页面调整参数。点击工具栏中的 Emitter Config 按钮，主界面就会变为如图 8-4 所示的参数设置页面。

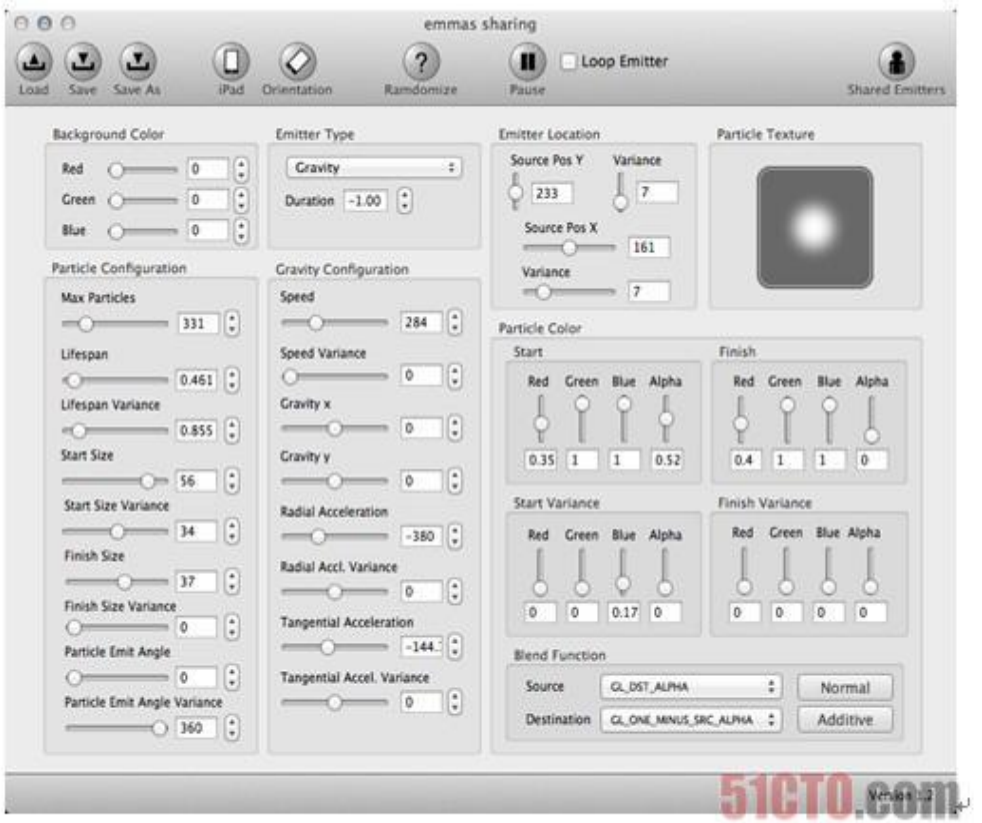


图8-4 参数设置页面

- 8.2.1 界面介绍（2）

8.2.1 界面介绍（2）

这个页面中的参数完全决定了粒子运行的效果，其中的每一项我们都可以进行修改。下面介绍参数设置页面中各种参数的含义。

Background Color：根据 RGB 值调节背景颜色，每种颜色的取值范围为 0~1。

Particle Configuration：用于配置粒子数目、粒子寿命和粒子寿命抖动值。Max Particles：屏幕中同时出现的最多粒子数目，其取值范围为 0~2000。Lifespan：每个粒子从产生到消失的平均时间，其取值范围为 0~10。Lifespan Variance：设定粒子寿命的最大值与最小值的差距，其取值范围为 0~10。抖动值越大，不同的粒子寿命相差越多。Start Size：即粒子生成时的大小，其取值范围为 0~64。Start Size Variance：设定粒子初始大小的抖动范围。Finish Size：即粒子消失时的大小。Finish Size Variance：设定粒子最终大小的抖动范围。Particle Emit Angle：范围为 0~360，0 度为水平向右的方向。Particle Emit Angle Variance：设定粒子发射角度的抖动范围，其取值范围为 0~360。若设置为 0，则发出的粒子在一条直线上。若设置为 360，则粒子向四周发散。

Emitter Type：发射类型，分为 Gravity（重力型）和 Radial（辐射型）。重力型为让所有粒子都受到某一个方向的力，而辐射型为让所有粒子的受力指向一点。两者的设置有所不同，接下来会介绍。Duration：效果持续的时间，其取值范围为-1~10，其中-1 为永远持续。

Gravity Configuration：重力参数设置。当选择 Emitter Type 为 Gravity 时，可以设置该

参数。Speed：粒子发射时的初速度，其取值范围为 0~1000。Speed Variance：设定粒子发射初速度的抖动范围，其取值范围为 0~1000。Gravity x：设定横向重力大小，其取值范围为-3000~+3000。Gravity y：设定纵向重力大小。Radial Acceleration：设定粒子发出时的径向加速度，其值越大，粒子越分散。Radial Accl. Variance：径向加速度抖动。Tangential Acceleration：设定粒子发出时的切向加速度，其值越大，粒子轨迹旋转角度越大。Tangential Accel. Variance：切向加速度抖动。

- Radial Configuration：辐射参数设置。当选择 Emitter Type 为 Radial 时，可以设置该参数。Max Radius：粒子相对辐射中心的最大半径。Max Radius variance：最大半径抖动。Min Radius：粒子相对辐射中心的最小半径。Deg. Per Second：粒子相对辐射中心的转速，即切向分速度。Deg. Per Second Variance：转速抖动。
- Emitter Location：粒子位置。Source Pos Y：粒子位置 Y 坐标。Variance：抖动值，此值只有在 Gravity 模式下可用。Source Pos X：粒子位置 X 坐标。Variance：抖动值。
- Particle Texture：粒子纹理。
- Particle Color：粒子颜色。Start：初始颜色。Finish：最终颜色。Start Variance：初始颜色抖动值。Finish Variance：最终颜色抖动值。Blend Function：混合选项，调节粒子之间互相叠加时的混合算法。

● 8.2.2 制作火焰特效

- 8.2.2 制作火焰特效

- 下面以制作火把的火焰效果为例，详细介绍一下各参数的功能。
- 先选定一个预置效果。为了达到说明的效果，我们选择一个最不像火焰的效果，通过修改各种参数把它变为火焰。这里我们用的预置效果是 Blue Galaxy（如图 8-5 所示）。

图 8-5 Blue Galaxy

- 点击主界面工具条右方的 Emitter Config 按钮进入参数设置页面，这里我们需要对粒子效果进行配置。首先背景颜色仍然维持黑色，所以背景参数不必修改。目前，Blue Galaxy 的粒子效果是各粒子向四周喷射，而我们需要的效果是粒子向一个方向喷射，因此需要设定重力参数。首先，在 X 和 Y 方向都施加重力 Gravity x=568.9、Gravity y=557.2（如图 8-6 所示），就可以得到如图 8-7 所示的效果。

图8-6 设置重力值

图8-7 施加了重力后的效果

- 然后调节径向加速度，设置 Radial Accleration=－808.4，Radial Accl. Variance=－130.8，使粒子发出时更集中。为了减小粒子的旋转，将切向加速度 Tangential Acceleration 与切向加速度抖动 Tangential Accel. Variance 调整为 0。至此，我们可以得到如图 8-8 所示的蓝色火焰效果。此时粒子的运动方向已经大致正确了，下一步我们将继续调节粒子大小等参数。

图8-8 调节加速度后的效果

- 目前，粒子分布呈现条状，和我们期望的火焰效果相比，粒子的尺寸太小，因此调节粒子的初始大小和抖动值，设置 Start Size=59，Start Size Variance=64。然后再调节粒子数目 Max Particles=174，使粒子总数维持在合适的范围。此时我们可以得到如图 8-9 所示的蓝色火焰。

图8-9 调节了大小和数目后的效果

- 与我们期望的火焰效果相比，粒子的"拖尾"略长，同时尾部粒子颜色减后会导致尾部过窄。因此首先调节粒子寿命值 Lifespan=0.901,Lifespan Variance=0.268,使拖尾变短,再调节粒子的终结大小 Finish Size=50、Finish Size Variance=0，使得粒子最终大小更大一些。最后,对粒子发射角度进行微调,使 Particle Emit Angle=5,Particle Emit Angel Variance=0，此时可以得到如图 8-10 所示的效果。

图8-10 调节生命周期、粒子最终大小与发射角度后的效果

- 继续对粒子速度进行微调，设置 Speed=87，Speed Variance=0，这时"拖尾"会变长，此时粒子的动作和形状调整已经完成。最后，我们设置火焰的颜色 Start=(Red:1, Green:0.82, Blue:0.53, Alpha=1)，Finish(Red=1, Green=0.28, Blue=0, Alpha=0)，完成后即可看到最终的火焰效果，如图 8-11 所示。

图8-11 最终效果

- 8.3 小结

- 8.3 小结

- Cocos2d-x 中的粒子系统看似简单，实际上却是一个十分强大的特效工具。使用得当的粒子系统可以实现许多梦幻般的特效。在本章中，我们演示了如何创建基本的雪花效果，并介绍了粒子效果编辑器 Particle Designer，展示了如何利用此编辑器创建一个火焰特效。
- 本章的知识点并不多，但也需要不断地积累经验才能熟练掌握粒子引擎。因此，希望读者可以发挥自己的想象力，创建属于自己的粒子效果。

9.1 瓦片地图

- **大型地图**
  - 在这一章中，我们来改造一下海底世界。到目前为止，捕鱼的场景还局限在一个屏幕之中，只是使用一个简单的图片来当做背景。现在我们尝试引入 3 个新的特性：更丰富的背景元素，即加入水草和礁石等；多个可供玩家切换的场景，简单来说就是提供不同风格的背景图；超过屏幕大小的地图，玩家可以像在即时战略游戏（如《魔兽争霸》）中一样在地图中滚动游戏画面。
  - 那么，该如何将这 3 个特性实现到游戏中呢？如果直接采用背景图切换的方式，需要为每个不同的场景准备一张背景图，而且每个背景图都不小，势必造成资源浪费。实际上，这些问题在稍大型的游戏中非常常见。无论是即时战略、角色扮演，还是模拟经营，通常都需要一张非常大的地图来展现一个灵活多变的世界。
  - **9.1 瓦片地图**
  - 瓦片地图就是为了解决前面提到的问题而产生的。一张大的世界地图或背景图片往往可以由屈指可数的几种地形来表示，每种地形对应于一张小的图片，我们称这些小的地形图片为瓦片。把这些瓦片拼接在一起，一个完整的地图就组合出来了，这就是瓦片地图的原理。
  - 在 Cocos2d-x 中，瓦片地图实现的是 TileMap 方案，TileMap 要求每个瓦片占据地图上一个四边形或六边形的区域。把不同的瓦片拼接在一起，就可以组成一个完整的地图了。我们需要使用许多较小的纹理来创建瓦片。通常来说，我们会把这些较小的纹理放入一张图片中，就像碎图纹理一样。这样做可以有效地提高绘图性能，同时也为引擎的管理带来了便利。因此，每个小瓦片都应该来自一张大的纹理图片。
  - 为了便于演示，我们对地图进行一下简化处理。在以下例子中，海底世界只包含 3 种不同的地形：沙地、岩石和海草（如图 9-1 所示）。地图纹理使用 Cocos2d-x 提供的测试工程中的资源"ortho-test1.png"，它可以在 Cocos2d-x 测试目录中"tests/Resources/TileMaps"下找到。
  - TileMap 地图支持 3 种不同的视图：正交视图（orthogonal view，瓦片水平垂直排列）、六边形视图（hexagonal view，六边形瓦片紧密连接）和等轴视图（isometric view，45 度斜视排列）。图 9-2 给出了两种常见视图的演示。

图9-1　三种地形

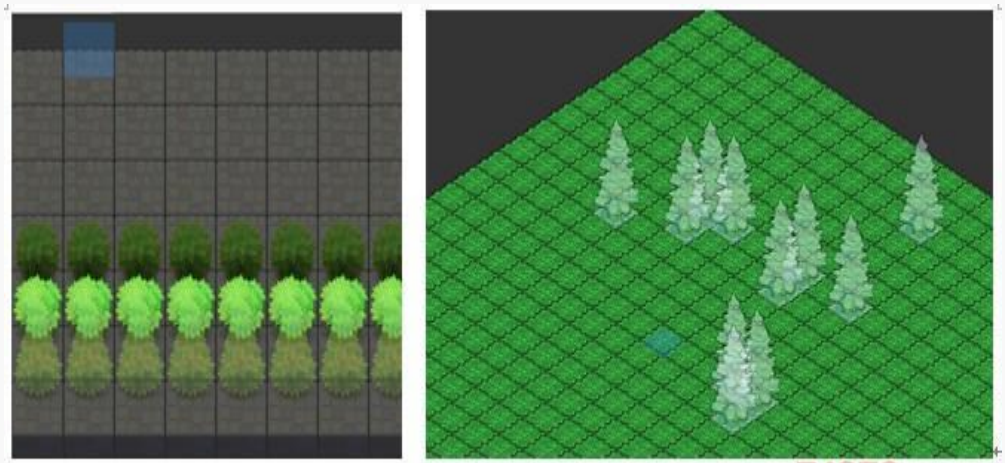


图9-2　两种地图视图

- 9.2.1　Tiled Map Editor 简介

9.2　编辑器

只有瓦片是不够的。将瓦片组织成一张大型的地图，甚至是世界级的地图，无论是使用代码编码，还是编辑为地图文件，都是一件令人生畏的事情。所幸我们已经不再活在纯命令行的时代了，我们可以借助图形化的编辑器来完成地图的编辑操作。

9.2.1　Tiled Map Editor 简介

Cocos2d-x 支持由瓦片地图编辑器 Tiled Map Editor 制作并保存为 TMX 格式的地图。Tiled Map Editor 是一个开源项目，支持 Windows、Linux 以及 OS X 多个操作系统，我们可以从 http://www.- mapeditor.org/中找到这个编辑器的 Java 与 Qt 版本。Tiled Map Editor 最初基于 Java 开发，后来移植到 Qt 之下，此处我们选用它的 Qt 版本。下面我们简单介绍一下 Tiled Map Editor 编辑器。

- 图 9-3 展示的就是 Tiled Map Editor 的主界面，打开一个新的文件后，就可以利用右下方的纹理瓦片来绘制地图了。从文件导入的纹理图会出现在右下角的视图内，选中某个纹理块，在左边的图层内点击就可以布置画面。软件右上角的视图显示了当前的图层信息。

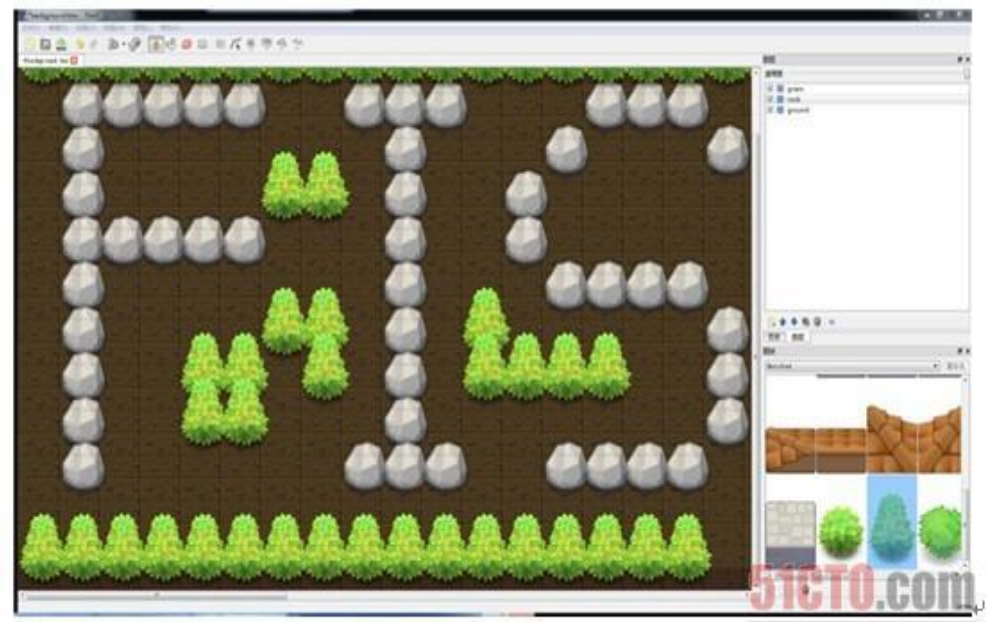


图9-3 Tiled Map Editor

- TileMap 中的层级关系和 Cocos2d-x 中的是类似的，地图可以包含多个不同的图层，每个图层内都放置瓦片，同层内的瓦片间平铺排列，而高一层的瓦片可以遮盖低一层的瓦片。与 Cocos2d-x 不同的是，TileMap 的坐标系的原点位于左上角，以一个瓦片为单位，换句话说，左上角第一块瓦片的坐标为(0,0)，而紧挨着它的右边的瓦片坐标就是(1,0)。
- TileMap 中的每一个瓦片拥有一个唯一的编号 GID，用于在地图中查找某个瓦片。Cocos2d-x 提供了一系列方法，可以从瓦片地图坐标获取对应瓦片的 GID，同时还可以利用这些方法来判断某个坐标下是否存在瓦片。稍后我们会看到瓦片地图 GID 的应用。
- 在这个例子中，我们首先利用主界面右下角的资源库选择沙地，在最底层上平铺一层，注意不要留下透明区域。完成后，再在底层的上方新建一个图层，利用相同的方式在新的图层中布置如图 9-3 所示的石头与水草，保存后备用。

● 9.2.2 创建水底世界（1）

- 9.2.2 创建水底世界（1）

- 下面我们以创建水底世界为例，演示地图编辑器的使用方法。如果我们想要建立一个水底世界样子的地图，需要在地图上面铺砖块，然后放置石头以及树木。
- 打开地图编辑器，单击"文件"→"新建"菜单项，此时可以看到如图 9-4 所示的地图编辑器界面。

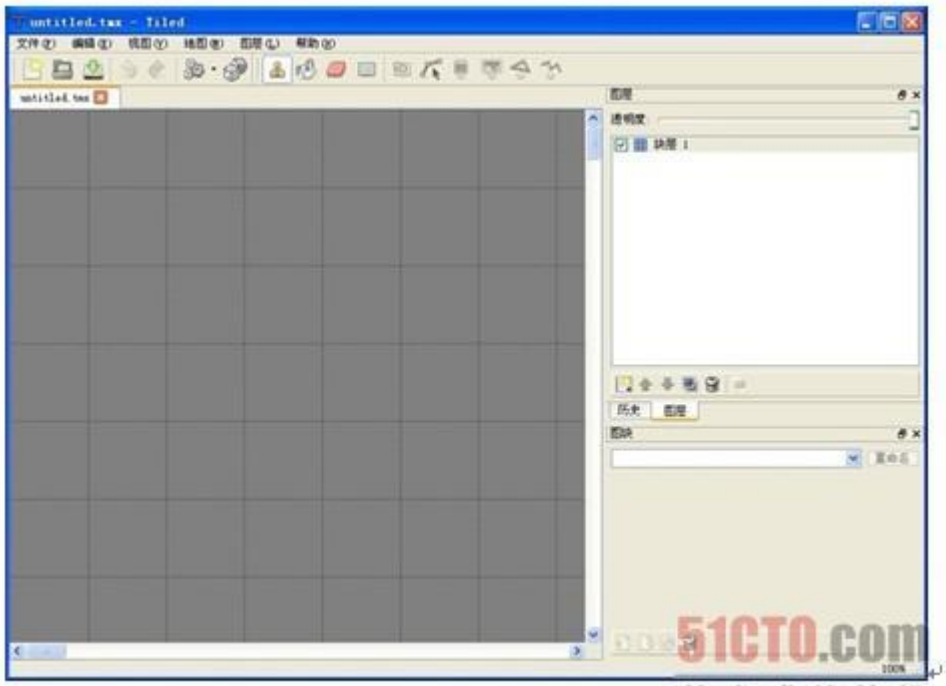


图9-4 主界面

- 我们已经新建了一个地图，现在来导入需要的素材文件。在菜单栏中选择"地图"→"新图块"菜单项（如图 9-5 所示），此时会弹出如图 9-6 所示的"新图块"对话框，我们需要在此对话框中创建图块。

图9-5 “新图块”菜单项

图9-6 “新图块”对话框

- 在对话框中单击"浏览"按钮，在弹出的"文件选择"对话框中选择包含地图素材的图片。Tiled Map Editor 支持多种图片类型，因此可以指定背景透明的 PNG 图片。此外，我们也可以通过"设置透明度"复选框把图片中的某种特殊颜色指定为透明色。如果导入的图片是一个完整的图块，只需把块宽度、块高度设置为图片的宽高即可。如果存在边距，也可以在此处设置。然而，通常我们导入的图片中不只包含一个图块。例如，在图片中可能包含 4 个图块，分为两行，每行包含两个图块。如果要一次导入多个图块，必须保证每一个图块的宽高相同，并在此处设置它们的块宽度和块高度。最后，我们可以设置导入图片绘制时的偏移。
- 导入成功后，就可以在"图块"栏目中看到刚才导入的图块了。我们先利用素材包中的图块画出背景的砖块。
- 选择图 9-7 所示的泥土，然后在工具栏中选择填充工具（也可以使用快捷键 F），如图 9-8 所示。

- 

图9-7 选择泥土背景

- 

图9-8 选择填充工具

- 然后在界面中部的地图上任意位置单击一下，此时地图会被泥土填充，得到如图 9-9 所示的地图。

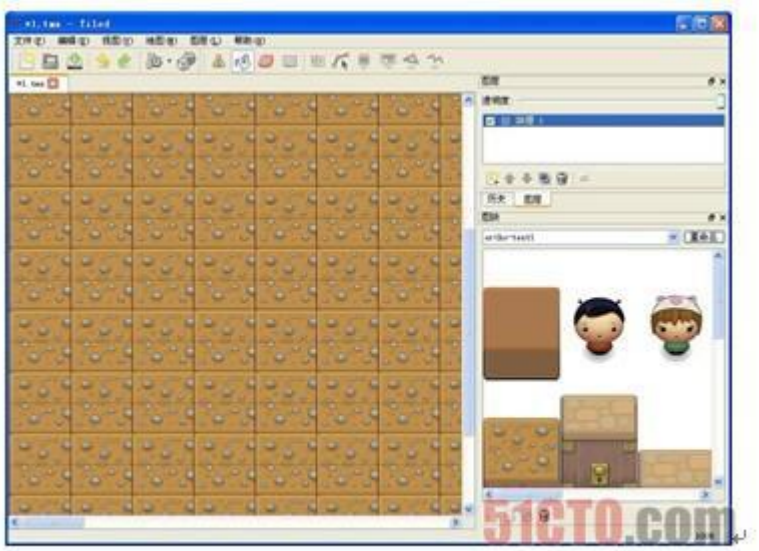


图9-9 填充地面

- 9.2.2 创建水底世界（2）

9.2.2 创建水底世界（2）

接着继续添加地图上的石块。石块位于泥土之上，因此我们需要新建一个泥土之上的图层，在新图层中绘制石块。在右上方"图层"栏目的空白处右击，在弹出的快捷菜单中选择"添加图层"菜单项（如图 9-10 所示）。

图9-10 “添加图层”菜单项

现在图层 2 已经建立好了，此时我们选中新的图层，像前面的步骤一样在"图块"栏目中选择石块，然后在工具栏中选择图章刷（也可以使用快捷键 B），在地图上需要放置石块的区域点击鼠标并绘制。我们可以得到如图 9-11 所示的地图。

图9-11 绘制第二个图层

- 重复上面的步骤，建立一个新的图层，并在新图层上绘制如图 9-12 所示的水草。

图9-12 绘制水草

- 至此，我们已经画出一个简单的水底世界地图。单击"文件"→"另存为"菜单项（如图 9-13 所示），在弹出的"保存文件"对话框中选择合适的位置保存地图，并在游戏中导入。

图9-13 “另存为”菜单项

9.3 导入游戏

9.3 导入游戏

- 深海世界背景已经构建完成，并导出了 TMX 文件，下一步我们将把 TMX 文件加载到游戏中。除了我们保存的 TMX 文件，瓦片用到的纹理图片等资源文件也需要导入工程中。在实际开发中，这个纹理图片可以只包含需要的瓦片，以便保持图片文件的体积最小。

- Cocos2d-x 为我们提供了 CCTMXTileMap 和 CCTMXLayer 两个类来处理瓦片地图。其中，CCTMXTileMap 代表一个完整的瓦片地图，它负责地图文件的载入、管理以及呈现。与其他游戏元素相同，CCTMXTileMap 也继承自 CCNode，因此可像层一样把它添加到游戏场景中。CCTMXLayer 代表一个瓦片地图中的图层，可以从图层对象获取图层信息，如某一点是否存在对象组或属性。CCTMXLayer 隶属于 CCTMXTileMap，因此通常不需要我们手动管理图层。

- 此外，CCTMXTileMap 还提供了一些操作地图的图层或地图对象的方法，可以通过关键字获取层、对象组或属性，这也为我们操作地图提供了便利。

- 下面继续构建我们的大型深海背景地图。我们直接在游戏场景的最底层添加刚刚制作完成的瓦片地图即可。为了便于在其他部分访问背景地图，我们将它定义为一个属性，并添加一个 setter 访问器：

```
void SpriteLayer::setBackgroundLayer(CCTMXTiledMap* bg)
{
    if(m_pBackgroundLayer)
        m_pBackgroundLayer->removeFromParentAndCleanup(true);

    m_pBackgroundLayer = bg;
    this->addChild(bg, 0);
}
```

然后在游戏场景的初始化方法中添加创建地图对象的代码：

```
m_pBackgroundLayer = NULL;
this->setBackgroundLayer(CCTMXTiledMap::create("background.tmx"));
```

此时我们已经完成了地图的创建，但此时的地图是静止的，需要添加代码来允许玩家卷动地图。因此我们在触摸层增加一个拖动动作，让地图随手指拖动而滑动，来展示地图的每一个部分。最后，我们在游戏场景中加入 singleTouchDragging 方法，允许触摸层来调用。该方法接受两个参数--startPoint 和 now，前者代表触摸开始点，后者代表当前拖动位置，在此方法中设置背景层的位置以实现地图的卷动。singleTouchDragging 方法的代码如下所示：

```
void SpriteLayer::singleTouchDragging(CCPoint startPoint, CCPoint now)
{
    CCPoint delta = ccpSub(now, startPoint);
    CCPoint newPosition = ccpAdd(m_pBackgroundLayer->getPosition(), delta);
    m_pBackgroundLayer->setPosition(newPosition);
}
```

9.4 实现层次感

9.4 实现层次感

现在我们已经有了一个足够大的深海背景图了，只是目前的背景图只是单纯的一个背景，这里我们可以使用一些技巧让整个画面更具有层次感，比如让鱼在水草间穿梭。

Tiled Map Editor 地图编辑器支持图层的概念，地图中的每一个图层恰好对应着 Cocos2d-x 里的 CCLayer，每一层在 Cocos2d 世界中的 Z 顺序值都不尽相同。

为了实现层次感，距离屏幕近的物体会遮挡远方的物体。具体到《捕鱼达人》的游戏场景中，鱼前方的水草应该遮挡鱼群。为了实现这个效果，我们做如下两件事情。

令鱼位于瓦片地图之中，这样才可以实现物体之间的互相遮挡，否则要么鱼群位于整个地图纸上，要么地图位于鱼群纸上。

实时计算鱼的位置，并修改它们的 Z 顺序值，把物体的前后关系体现出来。Z 顺序值较大的物体可以遮挡顺序值较小的物体。

在这个例子中，我们仅建立两个水草层，分别为 glass0 与 glass1。如图 9-14 所示，地图中有两片水草，它们前后都共有两层。把每相邻的两行拆分为靠近屏幕（即在地图中靠下）与远离屏幕（即在地图中靠上）两个部分，如果鱼游到它们之间，靠近屏幕的水草可以遮挡鱼，而远离屏幕的水草则不可以。我们把靠近屏幕的水草置于 glass1 图层中，而远离屏幕的水草置于 glass0 图层中，它们的 Z 轴顺序值分别设置为 3 与 4。

图9-13 “另存为”菜单项

- 此时，当鱼的 Z 轴顺序值为 2 时，它会被所有水草遮挡；为 3 时，会被部分水草遮挡；为 4 时，不会被任何水草遮挡。通过改变鱼群的 Z 轴顺序值，我们实现了鱼在水草中层次感地穿梭的效果，因此我们可以实时根据鱼的位置来设置鱼的 Z 轴顺序值。
- 为了实现以上效果，我们首先把背景地图从游戏场景移至精灵层之中，然后把鱼群从精灵层移动到背景地图中。接下来，需要在精灵层中添加一个 update 定时器，在其中实现每一帧更新鱼群的 Z 轴顺序值：

```
void SpriteLayer::updateFishZorder(ccTime dt)
{
    CCObject* fishIter = NULL;
    CCTMXLayer* layer1 = m_pBackgroundLayer->layerNamed("grass1");
    CCTMXLayer* layer0 = m_pBackgroundLayer->layerNamed("grass0");

    CCARRAY_FOREACH(_fishes, fishIter) {
        CCSprite* fish = dynamic_cast<CCSprite*>(fishIter);

        int zorder = 2;
        if(isOnLayer(layer1, fish->getPosition())) {
            zorder = 4;
        }
        else if(isOnLayer(layer0, fish->getPosition())) {
            zorder=3;
        }
        m_pBackgroundLayer->reorderChild(fish, zorder);
    }
}

bool SpriteLayer::isOnLayer(CCTMXLayer* layer, CCPoint position)
{
    CCSize tileSize = CCSizeMake(81,81); //瓦片大小
    CCSize frame = layer->getLayerSize(); //图层大小

    CCPoint positionInTmx =
        ccp(position.x, frame.height * tileSize.height - position.y);
    CCPoint tilePosition =
        ccp(positionInTmx.x / tileSize.width, positionInTmx.y / tileSize.height);
```

```
    //在地图中的坐标
    CCPoint tilePositionInt = ccp(floor(tilePosition.x), floor(tilePosition.y));

    if(tilePositionInt.x >= 0 && tilePositionInt.x < frame.width &&
        tilePositionInt.y >= 0 && tilePositionInt.y < frame.height) {
        return (layer->tileGIDAt(tilePosition) != 0);   //询问 GID
    }
    return false;
}
```

我们的地图只有两层，因此处理方式也比较简单。在以上代码中，我们只是检测每条鱼当前位置是否位于水草上，若是，则设定为相应层的 Z 顺序值。这个检测过程并不复杂，通过计算得到鱼在 TileMap 上的位置，然后根据鱼的位置向图层询问该处的瓦片 GID。GID 是瓦片地图中每一个瓦片的唯一编号，我们可以通过 GID 定位并操作某个瓦片。图层的 tileGIDAt 方法接受一个 TileMap 坐标，并返回此点瓦片的 GID，如果该位置不存在瓦片，此方法会返回 0。请注意之前的提示：TileMap 上的坐标以瓦片为单位，并且其原点位于地图左上方。

此时我们就可以看到如图 9-15 所示的鱼在水草间若隐若现穿行的效果。

图9-15 水草中穿梭的鱼

● 9.5 预定义属性

9.5 预定义属性

TileMap 中的每个元素都附带一个属性字典，用于设置一些额外的信息，我们把这些信息叫做预定义属性。可以在编辑地图的同时，我们可以自由地设置预定属性。

TileMap 的元素包括地图、图层和瓦片等，都可以附加预定义属性，这使我们几乎可以在任何地方设置需要的信息。例如，我们可以在层中存放黑洞的坐标，然后在游戏中读取出来，相关代码如下：

```
CCTMXLayer* layer = m_pBackgroundLayer->layerNamed("rock");
CCString* x = layer->propertyNamed("dead_x");
CCString* y = layer->propertyNamed("dead_y");
CCPoint position = ccp(x->intValue(), y->intValue());
```

● 9.6 小结

9.6 小结

- 在这一章中，我们简单介绍了瓦片地图的用法。借助瓦片地图，我们几乎可以在移动设备有限的硬件资源上构筑出无限大的世界。作为简化特例的深海世界背景，还加入了层次背景，鱼群在水草间的穿梭使得整个背景层和海底世界更生动。
- 下面总结一下瓦片地图的知识点。
- 瓦片地图（tile map）：由几种小"瓦片"通过拼接构成一张大型地图的技术，在大型地图中十分常见。
- 地图编辑器 Tiled Map Editor：一个可以兼容 Cocos2d-x 的瓦片地图编辑器，支持在 Windows、OS X 等系统上运行。通过 Tiled Map Editor，我们可以轻易地导入地图资源，并利用资源创建一整张地图。完成的地图也可以轻易导入 Cocos2d-x 项目中直接使用。
- 图块（tile）：也称为瓦片，是构成大型地图的基本元素，在 Cocos2d-x 中它可以组成图层（CCTMXLayer）。一个或多个图层构成一个完整的瓦片地图（CCTiledMap）。无论是 CCTMXLayer 还是 CCTiledMap，它们都直接或间接继承自 CCNode，因此都是可以添加到渲染树中的节点。
- 预定义属性：指地图中每一个地图对象的一项属性，这个属性在编辑器中可以被设为任意值，以便在开发游戏时对某些特定的对象进行特殊处理。

10.1 OpenGL 基础

- **Cocos2d-x 绘图原理及优化**

- 游戏引擎是对底层绘图接口的包装，Cocos2d-x 也一样，它是对不同平台下 OpenGL 的包装。在前面的章节中，我们已经对引擎提供的各个功能有了全面的了解，接下来我们将介绍更为底层却又十分重要的内容，那就是 Cocos2d-x 的绘图原理以及游戏优化方法。为了理解 Cocos2d-x 的绘图原理，我们首先介绍 Cocos2d-x 的绘图基础 OpenGL。在具备了一定基础后，10.2 节与 10.3 节将详细介绍引擎的绘图原理与优化方法。有一定 OpenGL 基础的读者可以跳过 10.1 节，直接从 10.2 节开始阅读。

- **10.1 OpenGL 基础**

- OpenGL 全称为 Open Graphics Library，是一个开放的、跨平台的高性能图形接口。OpenGL ES 则是 OpenGL 在移动设备上的衍生版本，具备与 OpenGL 一致的结构，包含了常用的图形功能。Cocos2d-x 就是一个基于 OpenGL 的游戏引擎，因此它的绘图部分完全由 OpenGL 实现。
- OpenGL 作为一个完整的底层图形接口，它的功能几乎涵盖了全部计算机图形学，因此我们无法在短短的几章中详细介绍。由于 Cocos2d-x 已经封装了大量的绘图细节，这里我们将简单介绍引擎的绘图方式，并从 OpenGL 渲染的角度来分析如何使游戏的效率达到最佳。

10.1.1 OpenGL 简介（1）

- **10.1.1 OpenGL 简介（1）**

- OpenGL 是一个基于 C 语言的三维图形 API，基本功能包含绘制几何图形、变换、着色、光照、贴图等。除了基本功能，OpenGL 还提供了诸如曲面图元、光栅操作、景深、shader 编程等高级功能。在本书中，我们仅介绍一些必要的概念与技术。

  - **状态机**

- OpenGL 是一个基于状态的绘图模型，我们把这种模型称为状态机。在此模型下，OpenGL 时刻维护着一组状态，这组状态涵盖了一切绘图参数，如即将绘制的多边形、填充颜色、纹理、混合模式和当前的坐标系等。为了正确地绘制图形，我们需要把 OpenGL 设置到合适的状态，然后调用绘图指令。例如，为了绘制一个三角形，首先需要设置坐标系、顶点列表以及填充颜色，然后发送绘图指令。
- 状态机的设计有许多优势。绘图是一件十分复杂的工作。为了绘制图形，我们通常需要设置许多参数（例如坐标变换，填充何种颜色，启用何种颜色混合模式，使用什么格式来描述多边形，纹理的像素格式等），其中许多参数并不频繁改变，

因此也没有必要每次都重新设置。OpenGL 把所有的参数作为状态来保存，如果没有设置新的参数，则会一直采用当前的状态来绘图。

- 另一个优势在于，我们可以把绘图设备人为地分为两个部分："服务器端"，负责具体的绘制渲染；"客户端"，负责向服务器端发送绘图指令。游戏通常运行在一台设备上，在设备中 CPU 负责运行游戏的逻辑，并向 GPU（硬件显卡或是软件模拟的显卡）发送绘图指令。在这种架构下，CPU 和 GPU 分别充当客户端与服务器端的角色（正如图 10-1 所描绘的那样）。在实际使用中，OpenGL 的客户端与服务器端是可以分离的，因此可以轻而易举地实现远程绘图。举例说明，如果需要实现一个远程桌面系统，设备 A 是被控制端，设备 B 是控制端，我们需要在设备 B 上呈现设备 A 中的图形，因此设备 A 可以通过网络向设备 B 发送绘图指令，而设备 B 负责绘制与渲染图形。在这个例子中，设备 A 就是 OpenGL 客户端，而设备 B 是 OpenGL 服务器端。

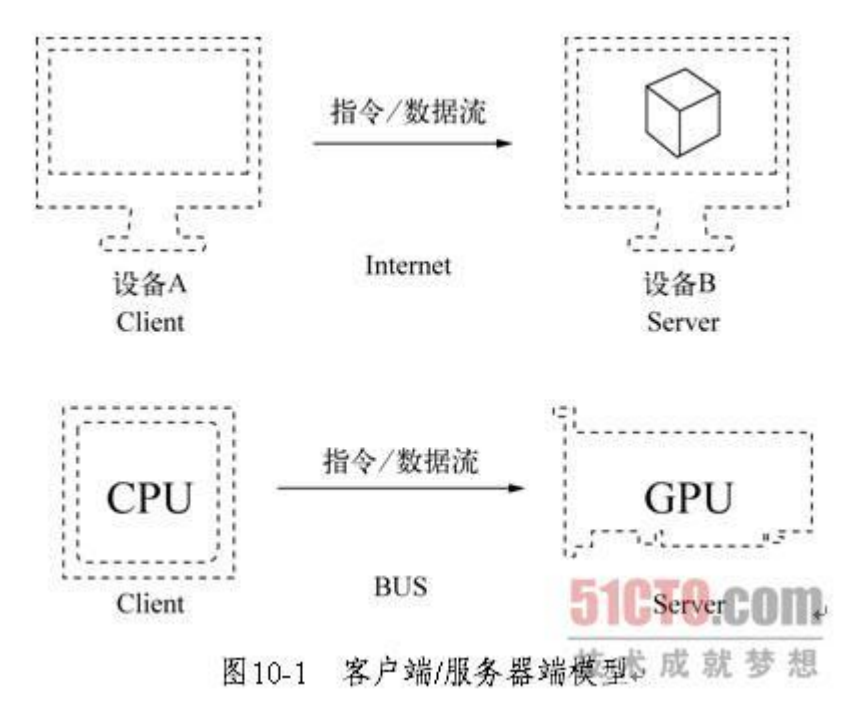


图10-1　客户端/服务器端模型

- 在游戏的例子中，绘图指令及数据由 CPU 发送到 GPU，状态机的优势看似并不是十分明显，而在远程绘图的例子中，绘图指令及数据由设备 A 通过网络发送到设备 B，网络的带宽显然是有限的，因此为了提高效率，我们通常把可以在客户端完成的工作分摊给客户端，只把绘图所必需的数据发送到服务器端即可。事实上，即使是运行在计算机上的游戏，也受益于 OpenGL 的架构。在计算机上，CPU 与 GPU 通过总线相连，虽然总线的带宽远高于网络连接，但在许多情况下，带宽明显不能满足高速运算的 CPU 与 GPU 之间传递数据的需要。因此我们也需要尽力避免在客户端与服务器端传递不必要的数据。
- OpenGL 提供了许多改变绘图状态的函数。例如，我们可以使用以下函数来开启或关闭绘图特性：

```
GL_APICALL void GL_APIENTRY glEnable (GLenum cap);  //开启一个状态
GL_APICALL void GL_APIENTRY glDisable (GLenum cap); //禁止一个状态
```

- 这里的 GLenum 类型用来表示 OpenGL 的状态量。后面我们将会看到，全部状态的列表定义在"gl2.h"头文件中。不同的绘图效果需要不同的支持状态，默认情况下，Cocos2d-x 只会开启固定的几种状态，必要的时候必须自己主动开启所需状态，使用完毕后主动禁止。例如，为了裁剪渲染区域，就需要设置 GL_SCISSOR_TEST 状态。实际上，从"gl2.h"头文件中就可以看出，OpenGL 是一个非常接近底层的接口标准，核心部分只包括了约 170 个函数和约 300 个常量，与汇编指令的数量相差无几，这也是我们需要用游戏引擎来减轻开发工作量的原因。

◆ **坐标系**

- OpenGL 是一个三维图形接口，在程序中使用右手三维坐标系。具体地说，在初始化时，屏幕向右的方向为 X 方向，屏幕向上的方向为 Y 方向，由屏幕指向我们的方向为 Z 方向，图 10-2 形象地说明了坐标系的构成。在三维空间中，每一个点都对应一个坐标。为了绘制各种图形，我们需要做的就是利用坐标描绘出图形的形状，然后把形状交给 OpenGL 来绘制。

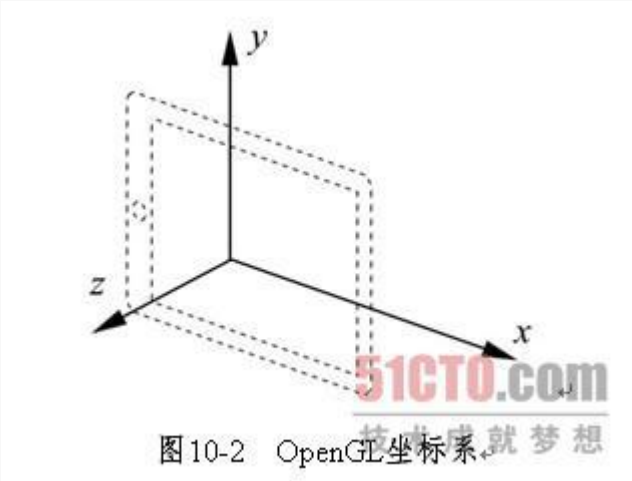


图 10-2　OpenGL 坐标系

10.1.1　OpenGL 简介（2）

10.1.1　OpenGL 简介（2）

- OpenGL 负责把三维空间中的对象通过投影、光栅化转换为二维图像，然后呈现到屏幕上。在 Cocos2d-x 中，我们只需要呈现二维的图像，因此 Z 坐标只用作控制游戏元素的前后顺序，通常不做讨论。为了呈现精灵，引擎会根据精灵的位置创建矩形，在 OpenGL 中设置矩形的顶点以及纹理，把图形绘制并呈现到屏幕上。图 10-3 简单地描述了三维图形如何呈现到屏幕上。

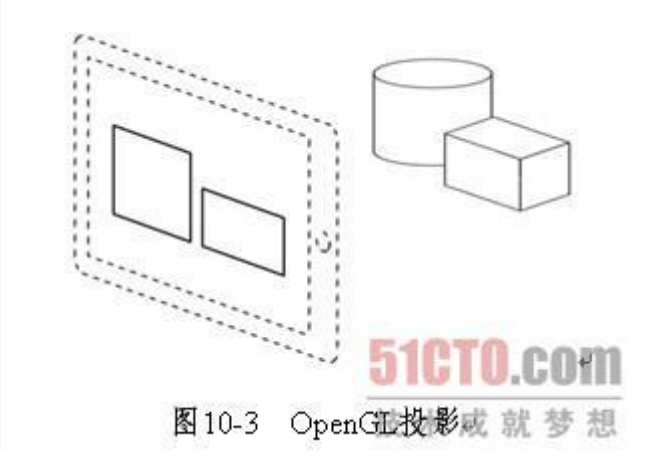


图 10-3　OpenGL 投影

- 在不对 OpenGL 做任何设置的时候，初始的坐标系称作世界坐标系，我们当然可以在世界坐标系中完成所有绘图。然而如果真的这么做，为了把一个物体绘制到不同的位置，我们就不得不去修改物体的所有顶点坐标。在游戏开发中，物体的顶点少则三个，多则上千个，对于每一次绘图都刻意地计算一次坐标是一项十分繁重的任务，甚至当我们需要通过相对坐标计算绝对坐标的时候，这项任务几乎难以完成。为了解决这个问题，OpenGL 提供了坐标系变换的功能。除了世界坐标系以外，OpenGL 还维护了一个绘图坐标系。绘图坐标系在初始化时与世界坐标系重叠，它也可以通过调用变换函数（例如平移、旋转和缩放）来随时改变。当我们绘制图形的时候，OpenGL 会把图形绘制在当前的绘图坐标系中。
- 以《捕鱼达人》的游戏场景的控制栏为例，控制栏是放置在屏幕下方的一片区域，其中包含了金钱数量、倒计时、道具和炮台等元素，每一个元素的坐标相对于控制栏的左下角来定位。其中，炮台的中心位于控制栏的中心处，可以沿着此点旋

转方向。在绘制控制栏时，引擎首先在屏幕左下角的位置确定绘图坐标系，绘制控制栏背景，如图 10-4b 所示，然后在屏幕正下方的位置确定绘图坐标系，在此处绘制炮台，如图 10-4c 所示。

(a) 控制栏

(b) 绘制控制栏背景

(c) 绘制炮塔

图 10-4 《捕鱼达人》控制栏

- **渲染流水线**

- 当我们把绘制的图形传递给 OpenGL 后，OpenGL 还要进行许多操作才能完成 3D 空间到屏幕的投影。通常，渲染流水线过程（如图 10-5 所示）有如下几步：显示列表、求值器、顶点装配、像素操作、纹理装配、光栅化和片断操作等。
- OpenGL ES 1.0 版本中采用的是固定渲染管线。在固定渲染管线模型中，每一个步骤的操作都是固定的，开发者只能使用 OpenGL 所提供的渲染模型，无法进行更改。

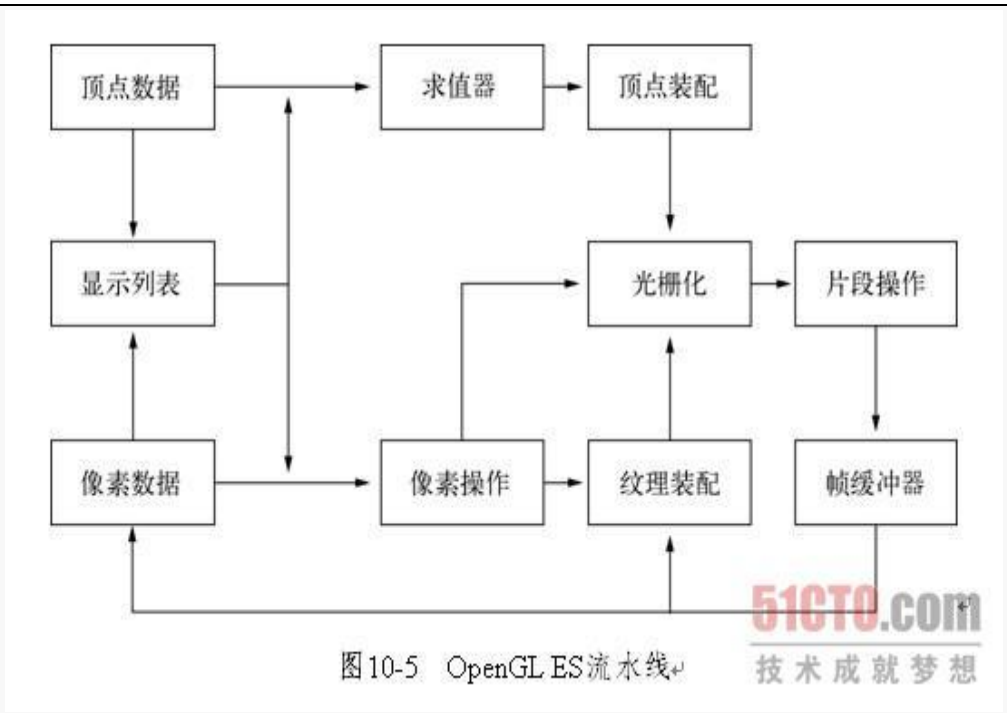


图10-5 OpenGL ES流水线

- OpenGL 从 2.0 版本开始引入了可编程着色器（shader）。可编程着色器作为原有渲染管线中一些部分的代替品，不仅可以实现原有的渲染功能，还可以自由实现开发者自定义的渲染效果。利用可编程着色器，开发者可以在渲染过程中自由控制顶点和片段处理采用的算法，以便实现更加炫丽的渲染效果。可编程着色器主要包含顶点着色器和片段着色器，其中前者负责对顶点进行几何变换以及光照计算，后者负责处理光栅化得到的像素以及纹理。

● 10.1.2 绘图

■ 10.1.2 绘图

- 前面我们简单介绍了 OpenGL 的工作原理以及基本概念，在这一节中，我们将介绍 OpenGL 的几个绘图函数。随着 OpenGL 的发展，其提供的绘图函数也变得多种多样。对于同一个效果来说，常常有多种不同的实现方法，因此想要在此对 OpenGL 的绘图函数进行全方位的介绍是不可能的，这里我们只简单介绍 Cocos2d-x 中常用的绘图函数。
- 下面我们从一个简单的例子开始介绍，在这个例子中，我们需要向 Cocos2d-x Hello World 项目中添加一些代码。打开 Hello World 项目，并在"HelloWorldScene.h"中的 HelloWorld 类中重载 void draw()方法：

```
virtual void draw();
```

- 然后打开"HelloWorldScene.cpp"文件，在文件中加入以下代码：

```
void HelloWorld::draw()
{
    //顶点数据
    static GLfloat vertex[] = {  //顶点坐标：x,y,z
          0.0f,   0.0f, 0.0f,    //左下
        200.0f,   0.0f, 0.0f,    //右下
          0.0f, 200.0f, 0.0f,    //左上
        200.0f, 200.0f, 0.0f,    //右上
    };
    static GLfloat coord[] = {   //纹理坐标：s,t
        0.0f, 1.0f,
        1.0f, 1.0f,
```

```
        0.0f, 0.0f,
        1.0f, 0.0f,
    };
    static GLfloat color[] = {  //颜色：红色、蓝色、绿色、不透明度
        1.0f, 1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f, 1.0f,
    };

    //初始化纹理
    static CCTexture2D* texture2d = NULL;
    if(!texture2d) {
        texture2d = CCTextureCache::sharedTextureCache()->addImage("HelloWorld.png");
        coord[2] = coord[6] = texture2d->getMaxS();
        coord[1] = coord[3] = texture2d->getMaxT();
    }

    //设置着色器
    ccGLEnableVertexAttribs(kCCVertexAttribFlag_PosColorTex);
    texture2d->getShaderProgram()->use();
    texture2d->getShaderProgram()->setUniformForModelViewProjectionMatrix();

    //绑定纹理
    glBindTexture(GL_TEXTURE_2D, texture2d->getName());

    //设置顶点数组
    glVertexAttribPointer(kCCVertexAttrib_Position, 3, GL_FLOAT, GL_FALSE, 0, vertex);
    glVertexAttribPointer(kCCVertexAttrib_TexCoords, 2, GL_FLOAT, GL_FALSE, 0, coord);
    glVertexAttribPointer(kCCVertexAttrib_Color, 4, GL_FLOAT, GL_FALSE, 0, color);

    //绘图
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
}
```

- 运行 Hello World 项目，我们就可以看到在游戏画面的左下角出现了一个 200×200 大小的 Cocos2d-x 标志，如图 10-6 所示。

- 回顾一下刚才的代码，通过注释我们可以大概了解到 draw 方法中每一条语句的含义。draw 大致上可以分为 3 个部分--数据部分、初始化纹理和绘图，它绘制了一个带纹理的矩形。事实上，我们也可以通过绘制一个"三角形带（triangle stripe）"来绘制。因为矩形实际上是两个包含公共斜边的直角三角形，所以绘制这样两个三角形，将它们的斜边相连，就可以拼成一个矩形。三角形带是计算机图形学中的一个重要概念，若要了解更多相关知识，可以参考任何一本计算机图形学方面的图书。

图10-6 OpenGL绘制的Cocos2d-x标志

代码的第一部分是数据部分，在这一部分中我们声明了 3 个静态数组，它们分别是 vertex、coord 和 color，对应了三角形带中共 4 个顶点的顶点坐标、纹理坐标和顶点颜色。每个数组均按照左下、右下、左上、右上的顺序来存储。

vertex：共 4 个顶点，每个顶点包含 x、y 和 z 三个分量，因此顶点坐标数组共有 12 个值。在本例中，矩形位于屏幕左下角，大小为 200×200。

coord：包含 s 和 t（横坐标和纵坐标）两个分量，因此共有 8 个值，每个分量的取值范围是 0 到 1，需要根据纹理的属性确定取值。

color：包含 r、g、b 和 a（红色、绿色、蓝色和不透明度）4 个分量，因此共有 16 个值，每个分量的取值范围是 0~1。把颜色值设为纯白(1, 1, 1, 1)，则会显示纹理原来的颜色。

第二部分是初始化纹理。利用 CCTextureCache 类可以方便地从文件中载入一个纹理，获取纹理尺寸，以及获取纹理在 OpenGL 中的编号。在纹理没有被初始化时，我们首先使用 CCTextureCache::addImage 方法载入一个图片，把返回的 CCTexture2D 对象保存下来，并使用纹理的属性设置 4 个顶点的纹理坐标。对于单个纹理的图片，只需要按照上面代码中的方法设置纹理坐标即可。关于纹理坐标的细节，我们将在 10.3.2 节中详细介绍。

最后一部分是绘制图片。绘制图片的步骤可以简述为：绑定纹理、设置顶点数组和绘图。绑定纹理是指把一个曾经载入的纹理当做当前纹理，从此绘制出来的多边形都使用此纹理。设置顶点数组是指为 OpenGL 指定第一步准备好的顶点坐标数组、纹理坐标数组以及顶点颜色数组。绘图则是最终通知 OpenGL 如何利用刚才提供的信息进行绘图，并实际把图形绘制出来。在这个过程中，我们可以看到最重要的一个函数为 glDrawArrays(GLenum mode, GLint first, GLsizei count)，其中 mode 指定将要绘制何种图形，first 表示前面数组中起始顶点的下标，count 表示即将绘制的图形顶点数量。

- 10.1.3 矩阵与变换

10.1.3 矩阵与变换

在 10.1.1 节中，我们曾经提到过坐标系变换。作为绘图的一个强大工具，坐标系变换在 OpenGL 开发中被广泛采用。为了理解坐标系变换，首先需要了解一些坐标系变换所需的数学知识，这些知识也是计算机图形学的数学基础。

OpenGL 对顶点进行的处理实际上可以归纳为接受顶点数据、进行投影、得到变换后的顶点数据这 3 个步骤。当我们设置好 OpenGL 的坐标系，并传入顶点数据后，OpenGL 就会通过一系列计算把顶点映射到世界坐标系之中，再把世界坐标系中的点

通过投影变换为可视平面上的点。这一系列变换的本质是通过对顶点坐标进行线性运算，得到处理后的顶点坐标。在计算机中，坐标变换是通过矩阵乘法实现的，用向量表示坐标，矩阵表示变换形式，则变换后的顶点坐标可以用向量与矩阵的乘法来表示。使用矩阵乘法的优点在于，计算机（包括移动设备）的图形硬件通常对矩阵乘法进行了大量优化，从而大大提高了运算效率。

- **点、向量与矩阵**
- 在计算机中，通常不直接使用与点维度数量一样的向量来表示一个点，因为这样就无法利用矩阵乘法来对点进行平移等操作了。因此，在计算机图形学中，通常采用齐次坐标来表示一个顶点。具体地说，齐次坐标系中每一个点的维度比顶点维度多 1，多出的一个维度值为 1。对于任何三维中的顶点(x, y, z)，它在齐次坐标系中的向量为[x, y, z, 1]，例如，空间中的(1.2, 5, 10)对应的向量为[1.2, 5, 10, 1]。
- 变换利用矩阵表示。常见的变换包含平移变换、旋转变换和缩放变换等，它们分别对应了平移矩阵、旋转矩阵和缩放矩阵等。下面以平移矩阵为例，展示如何使用矩阵乘法实现坐标变换。平移矩阵为

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 其中(tx,ty,tz)为平移的方向向量。若我们希望把点(1.2, 5, 10)平移(6, 5, 4)距离，则计算矩阵的乘法如下：

$$\begin{bmatrix} 1 & 0 & 0 & 6 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1.2 \\ 5 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 7.2 \\ 10 \\ 14 \\ 1 \end{bmatrix}$$

- 可以看到，我们得到了平移后的点(7.2, 10, 14)。上面是对一个点进行一次变换的情况，如果希望对点进行多次变换，则应该依次构造每个变换对应的矩阵，并利用矩阵乘法把所有矩阵与顶点向量相乘。例如，对点 P 依次进行缩放、平移、缩放和旋转操作，则分别构造它们对应的变换 S1、T、S2、R，按照如下公式计算变换后的点 P'：

$$P' = R \times S_2 \times T \times S_1 \times P$$

- OpenGL 维护了一个当前绘图矩阵，用于表示当前的绘图坐标系。这个矩阵被初始化为单位矩阵，此时绘图坐标系与世界坐标系相同，当我们不断地在绘图矩阵后乘上新的矩阵时，会相应地改变绘图坐标系。在上面的例子中，R × S2 × T × S1 即为绘图矩阵，它表示了一个绘图坐标系。在此点上绘制的 P 点坐标经过映射后，可以得到它在世界坐标系中对应的坐标 P'。
- OpenGL 为我们提供了一系列创建变换矩阵的函数（如表 10-1 所示），因此，在实际开发中，我们并不需要手动构造变换矩阵。这些函数的作用是创建一个变换矩阵，并在当前绘图矩阵的后方乘上这个矩阵。现在对刚才的例子稍作修改，我们不再希望只对点 P 进行一系列变换，而是希望对一个完整的图形进行变换。以下代码绘制一个任意的图形，并将此图形首先放大 2.5 倍，然后平移(1, 2, 3)距离，最后缩小 0.8 倍：

```
//OpenGL ES 1.0
glScalef(0.8f, 0.8f, 0.8f);       //乘上缩放矩阵
glTranslatef(1.0f, 2.0f, 3.0f);  //乘上平移矩阵
```

```
glScalef(2.5f, 2.5f, 2.5f);        //乘上缩放矩阵
DrawObject();                      //绘制任意图形
```

表 10-1　常见的 OpenGL ES 1.0 变换函数

函　数　名	描述
glTranslate	平移变换
glRotate	旋转变换
glScale	缩放变换

必须指出，无论是表 10-1 还是上面的代码，都明确提到了这些变换函数隶属于 OpenGL ES 1.0。实际上，在 Cocos2d-x 2.0 采用的 OpenGL ES 2.0 中，这些函数已经不可使用了。OpenGL ES 2.0 已经放弃了固定的渲染流水线，取而代之的是自定义的各种着色器，在这种情况下变换操作通常需要由开发者来维护。所幸引擎也引入了一套第三方库 Kazmath，它使得我们几乎可以按照原来 OpenGL ES 1.0 所采用的方式进行开发。表 10-2 列出了常用 OpenGL 矩阵操作函数的替代函数，而下面的代码则可以在 Cocos2d-x 2.0 中实现变换操作：

```
//Cocos2d-x 2.0（OpenGL ES 2.0）
kmGLScalef(0.8f, 0.8f, 0.8f);        //乘上缩放矩阵
kmGLTranslatef(1.0f, 2.0f, 3.0f);    //乘上平移矩阵
kmGLScalef(2.5f, 2.5f, 2.5f);        //乘上缩放矩阵
DrawObject();                        //绘制任意图形
```

表 10-2　Cocos2d-x 2.0 中矩阵函数的替代函数

OpenGL ES 1.0 函数	替代函数	描述
glPushMatrix	kmGLPushMatrix	把矩阵压栈
glPopMatrix	kmGLPopMatrix	从矩阵栈中弹出
glMatrixMode	kmGLMatrixMode	设置当前矩阵模式
glLoadIdentity	kmGLLoadIdentity	把当前矩阵置为单位矩阵
glLoadMatrix	kmGLLoadMatrix	设置当前矩阵的值
glMultMatrix	kmGLMultMatrix	右乘一个矩阵
glTranslatef	kmGLTranslatef	右乘一个平移矩阵
glRotatef	kmGLRotatef	右乘一个旋转矩阵
glScalef	kmGLScalef	右乘一个缩放矩阵

10.2.1　精灵的绘制

10.2　Cocos2d-x 绘图原理

经过前面的学习，我们已经对 OpenGL ES 有了一个大致的了解，下面就利用这些知识来分析 Cocos2d-x 的绘图原理。

10.2.1　精灵的绘制

绘制精灵的代码位于引擎源码的"sprite_nodes\CCSprite.cpp"中。打开 CCSprite 的代码文件，其中 draw 方法负责绘制精灵，其实现代码如下：

```
void CCSprite::draw(void)
{
```

```
    //1. 初始准备

    CC_PROFILER_START_CATEGORY(kCCProfilerCategorySprite, "CCSprite - draw");
    CCAssert(!m_pobBatchNode, "If CCSprite is being rendered by CCSpriteBatchNode,
        CCSprite#draw SHOULD NOT be called");
    CC_NODE_DRAW_SETUP();

    //2. 颜色混合函数

    ccGLBlendFunc(m_sBlendFunc.src, m_sBlendFunc.dst);

    //3. 绑定纹理

    if (m_pobTexture != NULL)
    {
        ccGLBindTexture2D(m_pobTexture->getName());
    }
    else
    {
        ccGLBindTexture2D(0);
    }

    //4. 绘图

    ccGLEnableVertexAttribs(kCCVertexAttribFlag_PosColorTex);

#define kQuadSize sizeof(m_sQuad.bl)
    long offset = (long)&m_sQuad;

    //顶点坐标
    int diff = offsetof(ccV3F_C4B_T2F, vertices);
    glVertexAttribPointer(kCCVertexAttrib_Position, 3, GL_FLOAT, GL_FALSE,
        kQuadSize, (void*) (offset + diff));

    //纹理坐标
    diff = offsetof(ccV3F_C4B_T2F, texCoords);
    glVertexAttribPointer(kCCVertexAttrib_TexCoords, 2, GL_FLOAT, GL_FALSE,
        kQuadSize, (void*)(offset + diff));

    //顶点颜色
    diff = offsetof(ccV3F_C4B_T2F, colors);
    glVertexAttribPointer(kCCVertexAttrib_Color, 4, GL_UNSIGNED_BYTE, GL_TRUE,
        kQuadSize, (void*)(offset + diff));

    //绘制图形
```

```
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

    CHECK_GL_ERROR_DEBUG();


    //5. 调试相关的处理

#if CC_SPRITE_DEBUG_DRAW == 1
    //调试模式 1：绘制边框
    CCPoint vertices[4]={
        ccp(m_sQuad.tl.vertices.x,m_sQuad.tl.vertices.y),
        ccp(m_sQuad.bl.vertices.x,m_sQuad.bl.vertices.y),
        ccp(m_sQuad.br.vertices.x,m_sQuad.br.vertices.y),
        ccp(m_sQuad.tr.vertices.x,m_sQuad.tr.vertices.y),
    };
    ccDrawPoly(vertices, 4, true);
#elif CC_SPRITE_DEBUG_DRAW == 2
    //调试模式 2：绘制纹理边缘
    CCSize s = this->getTextureRect().size;
    CCPoint offsetPix = this->getOffsetPosition();
    CCPoint vertices[4] = {
        ccp(offsetPix.x,offsetPix.y), ccp(offsetPix.x+s.width,offsetPix.y),
        ccp(offsetPix.x+s.width,offsetPix.y+s.height), ccp(offsetPix.x,offsetPix.y+
            s.height)
    };
    ccDrawPoly(vertices, 4, true);
#endif

    CC_INCREMENT_GL_DRAWS(1);

    CC_PROFILER_STOP_CATEGORY(kCCProfilerCategorySprite, "CCSprite - draw");
}
```

观察 draw 方法的代码可知，它包含 5 部分，其中前 4 个部分较为重要。第 1 部分主要负责设置 OpenGL 状态，如开启贴图等。第 2 部分负责设置颜色混合模式，与贴图渲染的方式有关。第 3、4 部分分别负责绑定纹理与绘图。这与 10.1.2 节中提供的绘图代码流程类似，首先绑定纹理，然后分别设置顶点坐标、纹理坐标以及顶点颜色，最终绘制几何体，其中顶点坐标、纹理坐标和顶点颜色需要在调用 draw 方法前计算出来。第 5 部分进行一些调试相关的处理操作。

同时我们也可以观察到，在进行一次普通精灵的绘制过程中，我们需要绑定一次纹理，设置一次顶点数据，绘制一次三角形带。对 OpenGL 的每一次调用都会花费一定的开销，当我们需要大量绘制精灵的时候，性能就会快速下降，甚至会导致帧率降低。因此，针对不同的情况，可以采取不同的策略来降低 OpenGL 调用次数，从而大幅提高游戏性能。这些技巧我们将在后面详细介绍，现在继续关注 Cocos2d-x 的绘图原理。

10.2.2 渲染树的绘制（1）

10.2.2 渲染树的绘制（1）

- 无论如何复杂的游戏场景也都是精灵通过不同的层次、位置组合构成的，因此只要可以把精灵按照前后层次，在不同的位置绘制出来就完成了游戏场景的绘制。（这里仅考虑由精灵构成的简单游戏，复杂的游戏也许会包含其他游戏元素，但是原理上并不冲突。）在第 3 章学习游戏元素时，我们曾接触过 Cocos2d-x 的渲染树结构，渲染树是由各种游戏元素按照层次关系构成的树结构，它展示了 Cocos2d-x 游戏的绘制层次，因此游戏的渲染顺序就是由渲染树决定的。
- 回顾 Cocos2d-x 游戏的层次：导演类 CCDirector 直接控制渲染树的根节点--场景（CCScene），场景包含多个层（CCLayer），层中包含多个精灵（CCSprite）。实际上，每一个上述的游戏元素都在渲染树中表示为节点（CCNode），游戏元素的归属关系就转换为了节点间的归属关系，进而形成树结构。
- CCNode 的 visit 方法实现了对一棵渲染树的绘制。为了绘制树中的一个节点，就需要绘制自己的子节点，直到没有子节点可以绘制时再结束这个过程。因此，为了每一帧都绘制一次渲染树，就需要调用渲染树的根节点。换句话说，当前场景的 visit 方法在每一帧都会被调用一次。这个调用是由游戏主循环完成的，在 3.7.1 节中，我们介绍了 Cocos2d-x 的调度原理，在游戏的每一帧都会运行一次主循环，并在主循环中实现对渲染树的渲染。下面是简化后的主循环代码，在注释中标明了对当前场景 visit 方法的调用：

```
void CCDirector::drawSceneSimplified()
{
    _calculate_time();

    if (! m_bPaused)
        m_pScheduler->update(m_fDeltaTime);

    if (m_pNextScene)
        setNextScene();

    _deal_with_opengl();

    if (m_pRunningScene)
        m_pRunningScene->visit();  //绘制当前场景

    _do_other_things();
}
```

- 绘制父节点时会引起子节点的绘制，同时，子节点的绘制方式与父节点的属性也有关。例如，父节点设置了放大比例，则子节点也会随之放大；父节点移动一段距离，则子节点会随之移动并保持相对位置不变。显而易见，绘制渲染树是一个递归的过程，下面我们来详细探讨 visit 的实现，相关代码如下：

```
void CCNode::visit()
{
    //1. 先行处理
    if (!m_bIsVisible)
    {
        return;
    }
    kmGLPushMatrix();   //矩阵压栈

    //处理 Grid 特效
```

```
    if (m_pGrid && m_pGrid->isActive())
    {
        m_pGrid->beforeDraw();
    }

    //2. 应用变换
    this->transform();

    //3. 递归绘图
    CCNode* pNode = NULL;
    unsigned int i = 0;

    if(m_pChildren && m_pChildren->count() > 0)
    {
        //存在子节点
        sortAllChildren();
        //绘制 zOrder < 0 的子节点
        ccArray *arrayData = m_pChildren->data;
        for( ; i < arrayData->num; i++ )
        {
            pNode = (CCNode*) arrayData->arr[i];

            if ( pNode && pNode->m_nZOrder < 0 )
            {
                pNode->visit();
            }
            else
            {
                break;
            }
        }
        //绘制自身
        this->draw();

        //绘制剩余的子节点
        for( ; i < arrayData->num; i++ )
        {
            pNode = (CCNode*) arrayData->arr[i];
            if (pNode)
            {
                pNode->visit();
            }
        }
    }
    else
```

```
    {
        //没有子节点：直接绘制自身
        this->draw();
    }

    //4. 恢复工作
    m_nOrderOfArrival = 0;

    if (m_pGrid && m_pGrid->isActive())
    {
         m_pGrid->afterDraw(this);
    }

    kmGLPopMatrix();  //矩阵出栈
}
```

● 10.2.2 渲染树的绘制（2）

■ 10.2.2 渲染树的绘制（2）

■ visit 方法分为 4 部分。第 1 部分是一些先行的处理，例如当此节点被设置为不可见时，则直接返回不进行绘制等。在这一步中，重要的环节是保存当前的绘图矩阵，也就是注释中的"矩阵压栈"操作。绘图矩阵保存好之后，就可以根据需要对矩阵进行任意的操作了，直到操作结束后再通过"矩阵出栈"来恢复保存的矩阵。由于所有对绘图矩阵的操作都在恢复矩阵之前进行，因此我们的改动不会影响到以后的绘制。

■ 在第 2 部分中，visit 方法调用了 transform 方法进行一系列变换，以便把自己以及子节点绘制到正确的位置上。为了理解 transform 方法，我们首先从 draw 方法的含义开始解释。draw 方法负责把图形绘制出来，但是从上一节的学习可知，draw 方法并不关心纹理绘制的位置，实际上它仅把纹理绘制到当前坐标系中的原点（如图 10-7a 所示）。为了把纹理绘制到正确的位置，我们需要在绘制之前调整当前坐标系，这个操作就由 transform 方法完成，经过变换后的坐标系恰好可以使纹理绘制到正确的位置（如图 10-7b 所示）。关于 transform 方法，我们稍后将会讨论。

■

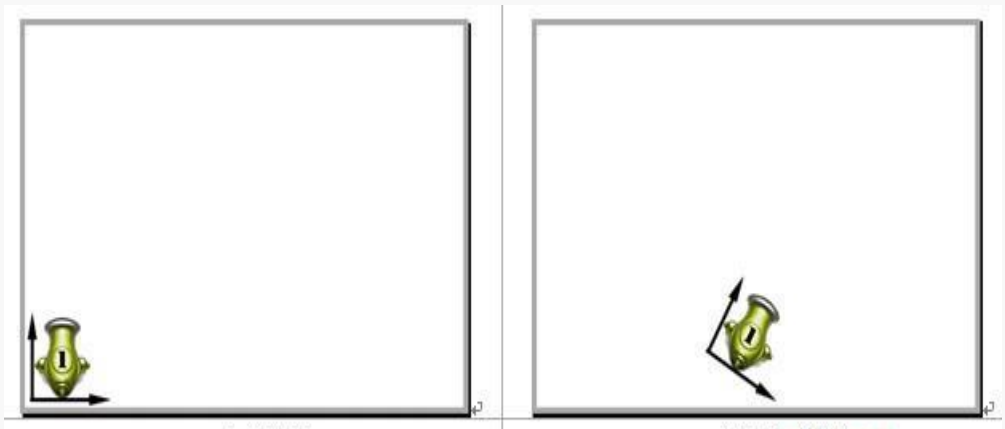


图10-7 变换前后绘制效果

■ 经过第 2 部分的变换后，我们得到了一个正确的坐标系，接下来的第 3 部分则开始绘图。visit 方法中进行了一个判断：如果节点不包含子节点，则直接绘制自身；如果节点包含子节点，则需要对子节点进行遍历，具体的方式为首先对子节点按

照 ZOrder 由小到大排序，首先对于 ZOrder 小于 0 的子节点，调用其 visit 方法递归绘制，然后绘制自身，最后继续按次序把 ZOrder 大于 0 的子节点递归绘制出来。经过这一轮递归，以自己为根节点的整个渲染树包括其子树都绘制完了。

- 最后是第 4 部分，进行绘制后的一些恢复工作。这一部分中重要的内容就是把之前压入栈中的矩阵弹出来，把当前矩阵恢复成压栈前的样子。
- 以上部分构成了 Cocos2d-x 渲染树绘制的整个框架，无论是精灵、层还是粒子引擎，甚至是场景，都遵循渲染树节点的绘制流程，即通过递归调用 visit 方法来按层次次序绘制整个游戏场景。同时，通过 transform 方法来实现坐标系的变换。

● 10.2.3 坐标变换

- 10.2.3 坐标变换

- 在绘制渲染树中，最关键的步骤之一就是进行坐标系的变换。没有坐标系的变换，则无法在正确的位置绘制出纹理。同时，坐标系的变换在其他的场合（例如碰撞检测中）也起着十分重要的作用。因此在这一节中，我们将介绍 Cocos2d-x 中的坐标变换功能。
- 首先，我们来看一下 transform 方法，其代码如下所示：

```
void CCNode::transform()
{
    kmMat4 transfrom4x4;

    //获取相对于父节点的变换矩阵 transform4x4
    CCAffineTransform tmpAffine = this->nodeToParentTransform();
    CGAffineToGL(&tmpAffine, transfrom4x4.mat);

    //设置 z 坐标
    transfrom4x4.mat[14] = m_fVertexZ;

    kmGLMultMatrix( &transfrom4x4 );  //当前矩阵与 transform4x4 相乘


    //处理摄像机与 Grid 特效
    if ( m_pCamera != NULL && !(m_pGrid != NULL && m_pGrid->isActive()) )
    {
        bool translate = (m_tAnchorPointInPoints.x != 0.0f ||
            m_tAnchorPointInPoints.y != 0.0f);

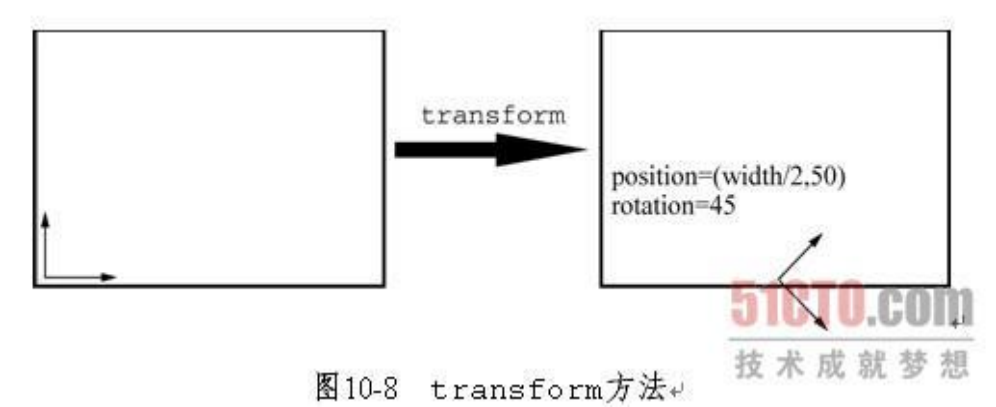
        if( translate )
            kmGLTranslatef(RENDER_IN_SUBPIXEL(m_tAnchorPointInPoints.x),
                RENDER_IN_SUBPIXEL(m_tAnchorPointInPoints.y), 0 );

        m_pCamera->locate();

        if( translate )
            kmGLTranslatef(RENDER_IN_SUBPIXEL(-m_tAnchorPointInPoints.x),
                RENDER_IN_SUBPIXEL(-m_tAnchorPointInPoints.y), 0 );
    }
```

- }

- 可以看到，上述代码用到了许多以"km"为前缀的函数，这是 Cocos2d-x 使用的一个开源几何计算库 Kazmath。在 10.1.3 节的末尾曾提到过，它是 OpenGL ES 1.0 变换函数的代替，可以为程序编写提供便利。在这个方法中，首先通过 nodeToParentTransform 方法获取此节点相对于父节点的变换矩阵，然后把它转换为 OpenGL 格式的矩阵并右乘在当前绘图矩阵之上，最后进行了一些摄像机与 Gird 特效相关的操作。把此节点相对于父节点的变换矩阵与当前节点相连，也就意味着在当前坐标系的基础上进行坐标系变换，得到新的合适的坐标系。这个过程中，变换矩阵等价于坐标系变换的方式，如果我们查看 nodeToParentTransform 的代码，就可以看到在此方法中，变换矩阵是如何根据当前节点的位置、旋转角度和缩放比例等属性计算出来的了。形象地讲，transform 方法的任务就是根据当前节点的属性计算出如何把绘图坐标系变换为新坐标系的矩阵。图 10-8 形象地描述了这一操作。



- 图10-8 transform方法

- 坐标系变换除了在绘图时有很重要的作用，它还为我们提供了一个有利的坐标转换工具。表 10-3 列举了渲染树节点提供的变换方法。

- 表 10-3 变换相关的方法

方法	描述
CCAffineTransform nodeToParentTransform()	获取节点相对于父节点的变换矩阵
CCAffineTransform parentToNodeTransform()	获取父节点相对于节点的变换矩阵
CCAffineTransform nodeToWorldTransform()	获取节点相对于世界坐标系的变换矩阵
CCAffineTransform worldToNodeTransform()	获取世界坐标系相对于节点的变换矩阵
CCPoint convertToNodeSpace (const CCPoint& worldPoint)	把世界坐标系中的点坐标转换到节点坐标系
CCPoint convertToWorldSpace (const CCPoint& nodePoint)	把节点坐标系中的点坐标转换到世界坐标系
CCPoint convertToNodeSpaceAR (const CCPoint& worldPoint)	把世界坐标系中的点坐标转换到节点坐标系（相对于锚点）
CCPoint convertToWorldSpaceAR (const CCPoint& nodePoint)	把节点坐标系中的点坐标（相对于锚点）转换到世界坐标系
CCPoint convertTouchToNodeSpace (CCTouch * touch)	获取触摸点在节点坐标系中的坐标
CCPoint convertTouchToNodeSpaceAR (CCTouch * touch)	获取触摸点在节点坐标系中的坐标（相对于锚点）

- 表 10-3 中提到的"节点坐标系"指的是以一个节点作为参考而产生的坐标系，换句话说，它的任何一个子节点的坐标值都是由这个坐标系确定的，通过以上方法，我们可以方便地处理触摸点，也可以方便地计算两个不同坐标系下点之间的方向关

系。例如，若我们需要判断一个点在另一坐标系下是否在同一个矩形之内，则可以把此点转换为世界坐标系，再从世界坐标系转换到目标坐标系中，此后只需要通过 contentSize 属性进行判断即可，相关代码如下：

```
bool IsInBox(CCPoint point)
{
    CCPoint pointWorld = node1->convertToWorldSpace(point);
    CCPoint pointTarget = node2->convertToNodeSpace(pointWorld);
    CCSize contentSize = node2->getContentSize();
    if(0 <= pointTarget.x && pointTarget.x <= contentSize.width
        && 0 <= pointTarget.y && pointTarget.y <= contentSize.height)
        return true;
}
```

● 10.3.1 绘图瓶颈

10.3 TexturePacker 与优化

在游戏开发初期，通常同时显示在屏幕上的精灵并不会很多，此时游戏的性能问题并不明显，游戏可以顺利流畅地运行（通常以 60 帧/秒作为流畅运行的标准）。然而随着游戏规模的扩大，屏幕上的精灵数量激增，精灵执行的动作越来越复杂，游戏的帧率也会随之下降。当帧率不足 30 帧/秒时，延迟现象还不是很明显，但是当帧率更低时，明显的延迟现象会导致极差的游戏体验，此时对游戏进行优化就十分必要了。

10.3.1 绘图瓶颈

影响绘图的因素有很多，我们不可能对其进行十分详细的分析。然而在 Cocos2d-x 中，影响游戏性能的瓶颈通常只有以下几个方面。

纹理过小：OpenGL 在显存中保存的纹理的长宽像素数一定是 2 的幂，对于大小不足的纹理，则在其余部分填充空白，这无疑是对显存极大的浪费；另一方面，同一个纹理可以容纳多个精灵，把内容相近的精灵拼合到一起是一个很好的选择。

纹理切换次数过多：当我们连续使用两个不同的纹理绘图时，GPU 不得不进行一次纹理切换，这是开销很大的操作，然而当我们不断地使用同一个纹理进行绘图时，GPU 工作在同一个状态，额外开销就小了很多，因此，如果我们需要批量绘制一些内容相近的精灵，就可以考虑利用这个特点来减少纹理切换的次数。

纹理过大：显存是有限的，如果在游戏中不加节制地使用很大的纹理，则必然会导致显存紧张，因此要尽可能减少纹理的尺寸以及色深。

针对以上绘图瓶颈，我们接下来介绍几种优化方式以显著提高游戏性能。

● 10.3.2 碎图压缩与精灵框帧

10.3.2 碎图压缩与精灵框帧

到目前为止，我们都是使用各自的纹理来创建精灵，由此导致的纹理过小和纹理切换次数过多是产生瓶颈的根源。针对这个问题，一个简单的解决方案是碎图合并与精灵框帧。碎图合并可以将许多零碎的小图片合并到一张大图里，并且这张大图的大小恰好符合 OpenGL 的纹理规范，从空间上减少无谓的浪费。框帧是纹理中的一部分，当我们把小纹理合并好之后就可以利用精灵框帧来创建精灵了。

这里我们简单介绍一款强大的碎图合并工具 TexturePacker，这是一个收费工具，但是它提供的免费版已经足以应付日常开发的大部分需求了。图 10-9 是 TexturePacker 的运行界面。

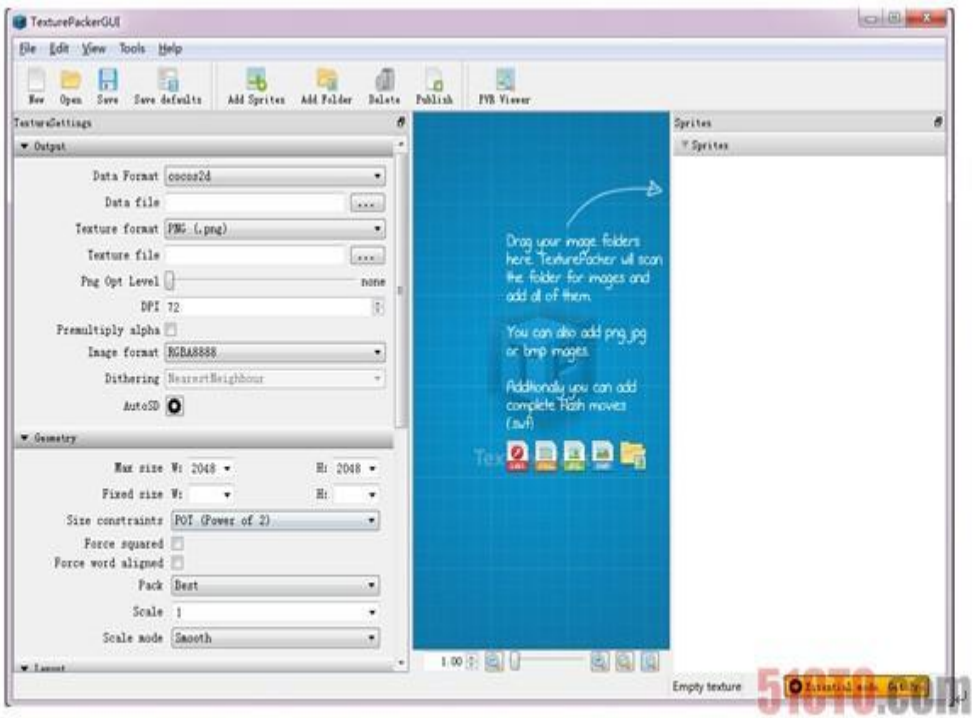


图10-9 TexturePacker

- TexturePacker 可以工作在一个最简单的模式下，将需要打包的碎图放在一个文件夹下，在工具栏中单击"Add Folder"按钮添加整个文件夹，然后单击"Publish"按钮发布，此时该文件夹下的图片就都合并到了一张大的纹理图下，并导出了与纹理图同名的 Plist 配置文件。
- 我们将游戏中的图片资源都合并为一张大图"all.png"，导出一个"all.plist"配置文件。这时候原来一张张分离的小图就冗余了，我们清理原来资源文件夹下的自定义图片，只保留引擎预置的"fps_images"相关文件，用于显示帧率等信息。使用大张的纹理资源后，原来的精灵加载方式也就不再适用，如果不改动代码，运行游戏时会报出大量找不到资源的错误。
- 对于不播放帧动画的静态精灵，比如炮台，我们通过精灵框帧来加载合并后的纹理资源。精灵框帧的概念相信读者已经不陌生了，它就是在帧动画中使用到的 CCSpriteFrame，其内部封装了一帧动画需要的纹理图、该帧在纹理中的位置和大小等信息。精灵框帧除了作为帧动画的组成之外，还可以用来表示纹理中的一个区域，用于代替图片文件来指定一个精灵绘制出的效果。
- 和播放帧动画一样，我们需要先将所有的碎图信息加载到引擎中，所幸 Cocos2d-x 和 TexturePacker 的对接是非常平滑的，只要一句代码即可完成：

```
CCSpriteFrameCache::sharedSpriteFrameCache()->addSpriteFramesWithFile("all.plist");
```

- 可能引起迷惑的是，我们将碎图信息放在了精灵框帧的缓存内，而不是纹理缓存内，这是因为我们的碎图用的是同一张纹理，这张纹理图内包含的丰富内容只能通过精灵框帧来表达。这时候原来的一张碎图的纹理会在缓存内对应一个同名的精灵框帧，CCSpirite 也提供了接口让我们从精灵框帧完成初始化：

```
cannon = CCSprite::createWithSpriteFrameName(cannonPath);
```

- 从炮台的加载方式看，实际上对于代码的改动仅仅是一条语句的事情。
- 另一方面，鱼的游动都是通过帧动画表现的，我们已经不经意地在动画中使用了碎图合并和精灵框帧这个优化。然而我们也可以考虑将多个鱼的游动动画合并到一张更大的纹理图中，以进一步减少浪费。

■ 这里有一点需要注意，如果需要合并的帧动画的每一帧都已经存放于一张纹理之中，那么合并的时候是无法充分利用纹理间的空间的。因为每一个帧动画所用的纹理都是一个比较大的矩形，多个这样的纹理拼在一起能减少的浪费空间并不大。因此，更好的方法是直接把帧动画每一帧的小图片导入 TexturePacker 中，把它们一起合成为一张较大的纹理。TexturePacker 会在 Plist 文件中记录原图的文件名，因此我们需要通过每一帧图片的原文件名来加载精灵框帧，并添加到动画中。

● 10.3.3 批量渲染

■ 10.3.3 批量渲染

■ 有了足够大的纹理图后，就可以考虑从渲染次数上进一步优化了。如果不需要切换绑定纹理，那么几个 OpenGL 的渲染请求是可以批量提交的，也就是说，在同一纹理下的绘制都可以一次提交完成。在 Cocos2d-x 中，我们提供了 CCSpriteBatchNode 来实现这一优化。

■ CCSpriteBatchNode 可以一次批量提交所有子节点的绘图请求，以减少提交次数，提高绘图性能。这个优化要求每个子节点都使用同一张纹理，例如我们可以把所有的鱼的图片放在一个纹理之中，每个精灵显示出自己存在于纹理中的那一部分。CCSpriteBatchNode 的使用方法也很简单，它是一个特殊的节点，我们只要把需要绘制的精灵添加为它的子节点，然后再把 CCSpriteBatchNode 添加到层或场景之中即可。当然，这些精灵必须使用统一个纹理。可以认为 CCSpriteBatchNode 所扮演的角色是精灵与绘图层间的一个中间层，只需要把需要绘制的精灵加入为它的子节点，就可以提高绘制效率。

■ 为简单起见，我们只对代码作非常小的改动。由于只执行一次鱼的创建，我们将创建的鱼精灵添加到一个 CCSpriteBatchNode，再统一添加到精灵层中，相关代码如下：

```
void SpriteLayer::createFish()
{
    char path[50];
    sprintf(path, s_pFish, 1);
    CCSize size = CCDirector::sharedDirector()->getWinSize();
    CCSpriteBatchNode* batch = CCSpriteBatchNode::create("all.png");

    for(int i = 0; i < 3000; i++) {
        CCSprite* fish = CCSprite::createWithSpriteFrameName(path);
        fish->setTag(fish_red_tag);
        fish->setPosition(ccp(size.width / RAND_MAX * rand(),
            size.height / RAND_MAX * rand()));

        CCSpriteFrame* sp = CCSpriteFrameCache::
            sharedSpriteFrameCache()->spriteFrameByName(path);
        CCTexture2D* texture = sp->getTexture();
        CCRect rect = sp->getRect();
        float w = rect.size .width / numOfFishFrame;
        float h = rect.size.height;

        CCAnimation *animation = CCAnimation::create();
        animation->setDelayPerUnit(0.15f);
        for(int i = 0 ; i < numOfFishFrame ; i++) {
            CCRect r = rect;
            r.origin.x += i * w;
            r.size.width = w;
```

```
            r.size.height = h;
            animation->addSpriteFrameWithTexture(texture, r);
        }
        CCAnimate* animate = CCAnimate::create(animation);
        fish->runAction(CCRepeatForever::create(animate));

        _fishes->addObject(fish);
        batch->addChild(fish);
    }
    m_pBackgroundLayer->addChild(batch);
}
```

有了以上优化，即使绘制数量十分庞大的同类精灵，也可以保证游戏流畅。

10.3.4 色彩深度优化

10.3.4 色彩深度优化

尽管我们已经在程序上尽了最大的努力，在一些低端设备上依然难以让游戏流畅地运行起来。为高端设备准备的纹理资源色彩深度较高，运行时占用大量的内存或显存，而低端设备的内存或显存空间不足，游戏运行时会表现得十分缓慢，甚至无法运行，这是一个十分常见的问题。这个时候我们不得不考虑从资源质量的角度做一些优化了。

另一方面，很多时候我们希望控制游戏包的尺寸，以方便玩家下载游戏。在我们已经把纹理尺寸优化到极致后，降低色彩深度就是继续减小游戏尺寸唯一的选择了。幸运的是，TexturePacker 提供了方便的功能来压缩色彩深度，从而减少资源的占用。

默认情况下，我们导出的纹理图片是 RGBA8888 格式的，它的含义是每个像素的红、蓝、绿、不透明度 4 个值分别占用 8 比特（相当于 1 字节），因此一个像素总共需要使用 4 个字节表示。若降低纹理的品质，则可以采用 RGBA4444 格式（如图 10-10 所示）来保存图片。RGBA4444 图片的每一个像素中每个分量只占用 4 比特，因此一个像素总共占用 2 字节，图片大小将整整减少一半。对于不透明的图片，我们可以选择无 Alpha 通道的颜色格式，例如 RGB565，可以在不增加尺寸的同时提高图像品质。各种图像编辑器通常都可以修改图片的色彩深度，TexturePacker 也提供了这个功能。

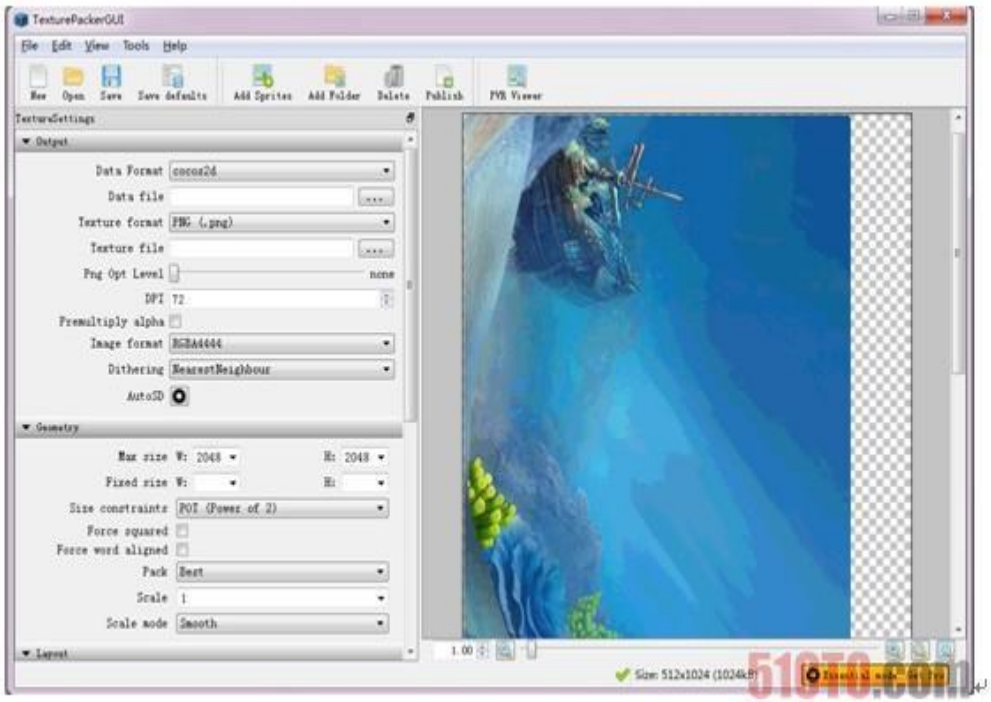


图10-10 RGBA4444格式的纹理

然而细心的读者会注意到，图 10-10 中压缩后的纹理资源出现了很生硬的色阶，这是由于纹理质量降低后，一些无法表达的颜色被近似表达的缘故，色彩渐变过渡的效果就变得生硬起来。为此，TexturePacker 提供了抖动的解决方案。抖动利用临近点的色差，在低色彩深度图片中模拟高色彩深度图片的效果。因此，压缩色彩深度的同时，在 TexturePacker 的"TextureSettings"一栏中，"Dithering"选项一般选用"FloydSteinberg"抖动方法（如图 10-11 所示）。

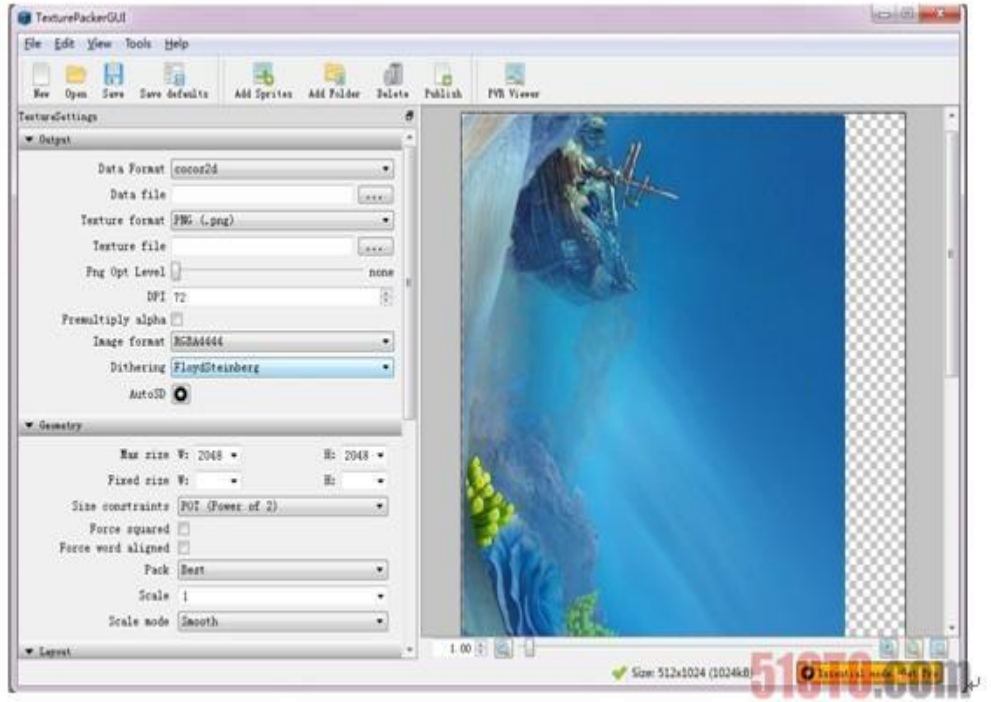


图10-11 抖动处理后的RGBA4444纹理

- 和原文件相比，压缩后的纹理的资源占用大大减少，质量却只是略有下降，在低端机型中这种区别是不易察觉的。压缩纹理可以节省不少计算时间和存储空间。

● 10.4 小结

- **10.4 小结**
- 在这一章中，我们首先探索了 Cocos2d-x 的绘图机制，然后根据其机制使用引擎提供的解决方案优化了我们的工程。在这一章中，我们接触到的知识点较多，在此总结如下。
- OpenGL：一个开放式的、跨平台的、高性能的图形 API。
- OpenGL ES：OpenGL 的简化版本，广泛被移动设备采用作为其绘图 API。
- 状态机、流水线：OpenGL 是基于状态机的图形接口，表现为它保存着一系列状态，用于描述 OpenGL 的行为，除非进行设置，否则每个状态都不会被改变。流水线是 OpenGL 的工作流程，理解流水线有助于理解 OpenGL 的绘图原理。
- OpenGL 绘制纹理：基本流程为绑定纹理、输入顶点数据（包括顶点坐标、纹理坐标和顶点颜色）、绘图。
- 变换：在 OpenGL 中，我们使用矩阵表示变换，任何点表示为一个四维向量，变换矩阵表示为四阶方阵。对顶点进行的几何变换对应了矩阵乘法。矩阵运算可以在 GPU 中高效完成。在 OpenGL 中，我们维护了一个表示当前坐标系的绘图矩阵，坐标系变换则通过改变此矩阵来实现。
- Cocos2d-x 绘图流程：在游戏主循环中调用场景的 visit 方法，在此之中通过 transform 方法进行坐标系变换，然后递归调用子节点的 visit 方法，完成对整棵渲染树的绘制。
- 碎图压缩：通过把小的纹理拼接为一个大纹理，减少纹理空间的浪费。
- 批量渲染：把需要批量绘制的纹理通过碎图纹理的方式放入一个大型纹理中，然后利用 CCBatchNode 统一绘制所有精灵，可以提高绘制效率。
- 颜色深度：表达一个颜色所需的比特长度。通常采用的 RGBA8888 格式利用 32 比特来描述一个颜色，而当我们对品质要求不高时，可以使用 RGBA4444 格式，采用 16 比特来描述一个颜色。此时图片的大小会减半，相对地，图片的颜色品质会下降。
- 这里介绍的 OpenGL 以及优化相关的知识还是略为浅显的，在更大型的工程中，为了提高绘制效率，以及降低内存消耗，常常会引入骨骼动画和网格纹理等用到 OpenGL 高级特性的深度优化方案。这里我们不再展开介绍，感兴趣的读者不妨查阅相关资料。

● 11.1 自定义绘图

- **OpenGL 绘图技巧**
- 在上一章中，我们巧妙地利用了 Cocos2d-x 绘图的一些特点，针对绘图效率进行了优化，如果再包括之前的用精灵加载图片等，Cocos2d-x 1.0 提供的绘图封装就介绍完了。
- 在 Cocos2d-x 2.0 中，引入了众多革命性的新特性，尤其在绘图机制上进行了大量的改进，使用了全新的 OpenGL ES 2.0 绘图，支持可编程管线 shader 等。底层的绘图矩阵优化对开发者是透明的、显式的，我们可以使用更多特效实现更丰富的绘图效果。
- OpenGL ES 是 OpenGL 三维图形 API 的子集，专门针对移动设备设计，其 1.0 版本是针对固定管线硬件的，而 2.0 版本已经扩展至支持可编程管线硬件。Cocos2d-x 2.0 正是将底层绘图从 OpenGL ES 1.0 升级到了 OpenGL ES 2.0。
- **11.1 自定义绘图**
- 引擎在 CCNode 中为我们预留了自定义绘图的接口，具体如下所示：

```
void CCNode::draw()
{
    //CCAssert(0);
    //可以重载此方法
    //最好仅在这个方法中绘制自定义的内容
}
```

- draw 函数是一个每帧都会调用的函数，注释中明确指出，通常情况下我们只应该在这个函数内编写自定义绘制效果，不要在这个函数以外的任何地方绘图。这是因为引擎在这个函数调用的上下文间进行了绘图环境的准备和必要的状态设置。在其他地方添加绘制代码可能引起不可预料的错误。
- Cocos2d-x 提供了一些简单的快捷绘图接口实现最简单的功能，我们可以使用这些接口，并从中找到很好的 OpenGL 编程规范。这些接口由"CCDrawingPrimitives.h"和对应的 cpp 文件提供，包括了点、线、多边形、圆形和贝塞尔曲线等最基本的几何图形的绘制，还包括了一些基本的设置，如设置点的大小、绘制的颜色等。
- 利用集成的快捷绘图，我们不妨给炮台加上瞄准线功能。考虑到针对炮台的功能已经足够多、足够复杂了，我们将炮台抽象为一个类，集中封装相关操作。根据引擎的接口规范，应该在 draw 函数中绘制我们的自定义效果，而不是任何其他的函数。修改炮台精灵的代码，在精灵中重载 draw 方法，相关代码如下：

```
void CannonSprite::draw()
{
    CCSprite::draw();                    //调用 CCSprite 的绘图，保证纹理被正确绘制

    CCPoint origin = CCPointZero;
    CCPoint direction = ccp(0, 1);
    direction = ccpMult(direction, 1024);
    CCPoint target = ccpAdd(origin, direction)

    ccDrawColor4B(255, 225, 255, 255);   //设置绘图颜色为白色
    ccDrawLine(origin, target);          //绘制线段
}
```

- 再次强调，我们将绘制瞄准线放在了子节点绘图后，来保证这个绘图不会被覆盖。细心的读者会注意到，这里并没有计算炮台的旋转角度，只是沿着当前垂直方向画了一条线，炮台旋转时引起的整个坐标系的变换能保证我们的瞄准线也随着炮台一起旋转相应的角度。完成后的效果如图 11-1 所示。

图11-1 瞄准线效果

实际上，"CCDrawingPrimitives.h"中封装的形如 ccDrawLine 的绘图函数，每调用一次都会引发 OpenGL 的渲染，但是只完成很少量的绘制。大量地调用绘图函数对于 GPU 来说是个不小的负担，因此这类绘图函数只适合在特定的情况下少量使用。

11.2 遮罩层（1）

11.2 遮罩层（1）

在瞄准线中我们用了引擎封装的快捷绘图，熟悉了自定义绘图的基本流程。对于稍复杂的绘图效果，就需要调用底层的 OpenGL 接口了，这里我们用一个小小的例子来说明：滚动的数字表盘，在游戏中显示倒计时。在这个例子中，我们将使用 OpenGL 提供的遮罩效果来快速实现这一效果。

遮罩效果又称为剪刀效果，允许一切的渲染结果只在屏幕的一个指定区域显示：开启遮罩效果后，一切的绘制提交都是正常渲染的，但最终只有屏幕上的指定区域会被绘制。形象地说，我们将当前屏幕截图成一张固定的画布盖在屏幕上，只挖空指定的区域使之能活动，而屏幕上的其他位置尽管如常更新，但都被掩盖住了。 于是，我们可以在表盘上顺序排列所有的数字，不该显示的部分用遮罩效果盖住，滚动的表盘效果可以借助遮罩得到快速的实现。

表盘类的接口比较简单，需要指定的是表盘的数字位数和每个数字的大小。另外，还有一个 Number 接口用了刷新表盘显示的数字。它的代码如下：

```
class NumberScrollLabel : public CCNode
{
    int m_numberCnt;
    CC_PROPERTY(int, m_number, Number);
public:
    static NumberScrollLabel* label(int numberCnt, int numberSize);

    bool init(int numberCnt = 1, int numberSize = 17)
```

```
    {
        m_numberCnt = numberCnt;
        m_number = 0;
        for(int i = 0; i < m_numberCnt; i++) {
            NumberScrollLabelBase *single = NumberScrollLabelBase::label(numberSize);
            single->setPosition(ccp(numberSize * (m_numberCnt - i - 1), 0));
            single->setTag(i);
            this->addChild(single);
        }
        return true;
    }
};
```

- 其他函数的实现都比较简单，本书中就暂且略过。这是因为我们把滚动数字的实现放在了一个表盘数字类中，表盘类只是根据显示位数将数字类组合到一起。
- 正如前面所说，数字类将顺序排列 10 个 CCLabel 来负责显示一位数字（从 0 到 9，共 10 个数字），其中用到的两个关键方法是 init 与 setNumber，其中 init 方法的代码如下：

```
bool init(int numberSize = 17)
{
    m_numberSize = numberSize;
    visibleNode = CCNode::create();
    this->addChild(visibleNode);

    for(int i = 0; i < 10; i++) {
        char str[2];
        str[0] = '0' + i;
        str[1] = '\0';

        CCLabelTTF* single = CCLabelTTF::create(str, "", numberSize);
        single->setTag(i);
        single->setAnchorPoint(ccp(0, 0));
        single->setPosition(ccp(0, numberSize * i));
        visibleNode->addChild(single);

        labels[i] = single;
    }
    return true;
}
```

- 这里我们没有直接将所有的 label 添加到数字类本身，而是用一个 visibleNode 来承载所有的 label。在后面会看到，这样能给裁剪操作和移动操作带来不小的便利。
- 设置新的数字时，我们移动 visibleNode，也就相当于整体移动所有的数字 label。注意，因为移动的只是 visibleNode，对数字类本身的坐标不作改变，父层仍然可以正确地设置这个数字的位置，也避免了一个个移动 label 的烦琐。
- setNumber 方法用于设置表盘数字的值，让表盘数字滚动到我们所设置的位置。我们的做法是获取所设置值的位置，并为 visibleNode 添加一个滚动动作，这样就可以使表盘滚动到合适的位置了。setNumber 方法的代码如下：

```
void setNumber(int var, bool up = true)
{
    this->stopAllActions();
    CCPoint moveToPosition = ccp(visibleNode->getPosition().x,
        -labels[var]->getPosition().y);
    CCMoveTo* moveAction = CCMoveTo::create(0.5, moveToPosition);
    visibleNode->runAction(moveAction);
}
```


11.2 遮罩层（2）

完善了上面的代码后，在游戏菜单层中添加一个数字表盘并增加相应的计时刷新方法 Game- MenuLayer::updateTimer，它每秒被调用一次，将数字表盘的读数加一。因此，向菜单层的初始化方法中加入如下代码：

```
numbers = NumberScrollLabel::label(3, 34);
numbers->setPosition(ccp(winSize.width / 2, winSize.height / 2));
this->addChild(numbers);
this->schedule(schedule_selector(GameMenuLayer::updateTimer), 1);
```

编译并运行游戏，可以看到如图 11-2 所示的效果。已经可以看到数字表盘的雏形，当然，除了需要显示在表盘上的数字，其他数字也肆无忌惮地在屏幕上滚动了。下面我们在数字类中添加遮罩效果，将不应该出现的数字隐藏起来。重载 NumberScrollLabel::visit 方法，相关代码如下所示：

```
void visit()
{
    //启动遮罩效果
    glEnable(GL_SCISSOR_TEST);

    CCPoint pos = CCPointZero;
    pos = visibleNode->getParent()->convertToWorldSpace(pos);  //获取屏幕绝对位置
    CCRect rect = CCRectMake(pos.x, pos.y, m_numberSize, m_numberSize);

    //设置遮罩效果
    glScissor(rect.origin.x, rect.origin.y, rect.size.width, rect.size.height);

    CCNode::visit();

    //关闭遮罩效果
    glDisable(GL_SCISSOR_TEST);
}
```

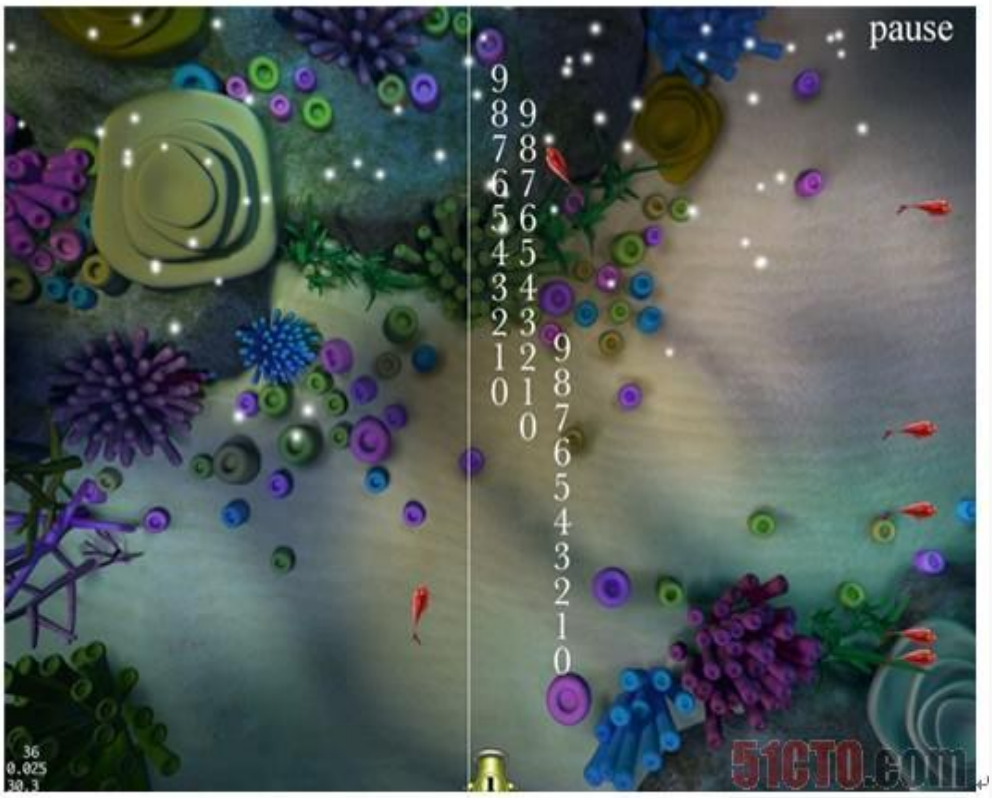


图11-2 滚动数字

- 这里我们选择重写 visit 函数来设置遮罩效果，对于"仅在 draw 中设置绘图效果"原则是个小小的破例。这样做是为了能成功遮挡所有子节点的无效绘图。回想一下引擎中渲染树的绘制过程，draw 方法并不是递归调用的，而 visit 方法是递归的，并且 visit 方法通过调用 draw 来实现绘图。因此，我们在设置了遮罩效果后调用了父类的 visit，使绘制流程正常进行下去，最后在绘制完子节点后关闭遮罩效果。最终效果如图 11-3 所示，虽然并不美观，但稍经美化就可以看到类似老虎机表盘的滚动效果了。

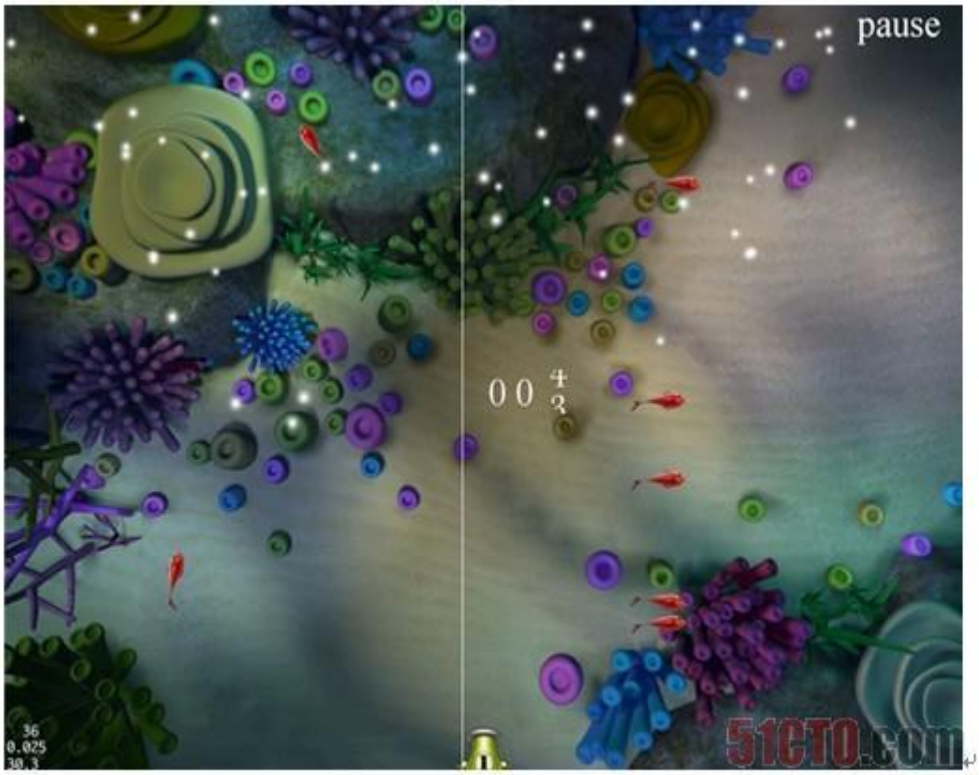


图11-3 遮罩效果

11.3 数据交流（1）

在数字表盘中，我们第一次成功使用了 OpenGL 的底层特效，关键的调用不多，而达到的效果还是令人惊喜的。下面我们再为游戏添加一个小小的截屏功能，借此讨论游戏中涉及的底层的数据交流。

底层的数据交流必须介绍两个类：CCImage 和 CCTexture2D，这是引擎提供的描述纹理图片的类，也是我们和显卡进行数据交换时主要涉及的数据结构。

CCImage 在"CCImage.h"中定义，表示一张加载到内存的纹理图片。在其内部的实现中，纹理以每个像素的颜色值保存在内存之中。CCImage 通常作为文件和显卡间数据交换的一个工具，因此主要提供了两个方面的功能：一方面是文件的加载与保存，另一方面是内存缓冲区的读写。

我们可以使用 CCImage 轻松地读写图片文件。目前，CCImage 支持 PNG、JPEG 和 TIFF 三种主流的图片格式。下面列举与文件读写相关的方法：

```
bool initWithImageFile(const char* strPath, EImageFormat imageType = kFmtPng);
bool initWithImageFileThreadSafe(const char* fullpath, EImageFormat imageType = kFmtPng);
bool saveToFile(const char* pszFilePath, bool bIsToRGB = true);
```

CCImage 也提供了读写内存的接口。getData 和 getDataLen 这两个方法提供了获取当前纹理的缓冲区的功能，而 initWithImageData 方法提供了使用像素数据初始化图片的功能。相关的方法定义如下：

```
unsigned char* getData();
int getDataLen();
bool initWithImageData(void* pData,
    int nDataLen,
```

```
    EImageFormat eFmt = kFmtUnKnown,
    int nWidth = 0,
    int nHeight = 0,
    int nBitsPerComponent = 8);
```

- 注意，目前仅支持从内存中加载 RGBA8888 格式的图片。
- 另一个重要的类是 CCTexture2D，之前已经反复提及，它描述了一张纹理，知道如何将自己绘制到屏幕上。通过该类还可以设置纹理过滤、抗锯齿等参数。该类还提供了一个接口，将字符串创建成纹理。
- 这里需要特别重提的两点是：该类所包含的纹理大小必须是 2 的幂次，因此纹理的大小不一定就等于图片的大小；另外，有别于 CCImage，这是一张存在于显存中的纹理，实际上并不一定存在于内存中。
- 了解了 CCImage 和 CCTexture2D 后，我们就可以添加截屏功能了。截屏应该是一个通用的功能，不妨写成全局函数放在 MTUtil 库中，使其不依赖于任何一个类。
- 首先，我们使用 OpenGL 的一个底层函数 glReadPixels 实现截图：

```
void glReadPixels (GLint x, GLint y,
    GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid* pixels);
```

- 这个函数将当前屏幕上的像素读取到一个内存块 pixels 中，且 pixels 指针指向的内存必须足够大。为此，我们设计一个函数 saveScreenToCCImage 来实现截图功能，相关代码如下：

```
unsigned char screenBuffer[1024 * 1024 * 8];
CCImage* saveScreenToCCImage(bool upsidedown = true)
{
    CCSize winSize = CCDirector::sharedDirector()->getWinSizeInPixels();
    int w = winSize.width;
    int h = winSize.height;
    int  myDataLength = w * h * 4;

    GLubyte* buffer = screenBuffer;
    glReadPixels(0, 0, w, h, GL_RGBA, GL_UNSIGNED_BYTE, buffer);

    CCImage* image = new CCImage();
    if(upsidedown) {
        GLubyte* buffer2 = (GLubyte*) malloc(myDataLength);
        for(int y = 0; y <h; y++) {
            for(int x = 0; x <w * 4; x++) {
                buffer2[(h - 1 - y) * w * 4 + x] = buffer[y * 4 * w + x];
            }
        }
        bool ok = image->initWithImageData(buffer2, myDataLength,
            CCImage::kFmtRawData, w, h);
        free(buffer2);
    }
    else {
        bool ok = image->initWithImageData(buffer, myDataLength,
            CCImage::kFmtRawData, w, h);
```

```
    }
    return image;
}
```

这里我们使用 glReadPixels 方法将当前绘图区的像素都读取到了一个内存缓冲区内，然后用这个缓冲区来初始化 CCImage 并返回。注意，我们设置了一个参数 upsidedown，当这个参数为 true 时，我们将所有像素倒序排列了一次。这是因为 OpenGL 的绘制是从上到下的，如果直接使用读取的数据，再次绘制时将上下倒置。

11.3 数据交流（2）

11.3 数据交流（2）

在这个函数的基础上，我们在游戏菜单层中添加相关按钮和响应操作就完成了截屏功能，相关代码如下：

```
void GameMenuLayer::saveScreen(CCObject* sender)
{
    CCImage* image = saveScreenToCCImage();
    image->saveToFile("screen.png");
    image->release();
}
```

实际上，引擎还提供了另一个很有趣的方法让我们完成截图功能。在 Cocos2d-x 中，我们实现了一个渲染纹理类 CCRenderTexture，其作用是将绘图设备从屏幕转移到一张纹理上，从而使得一段连续的绘图被保存到纹理中。这在 OpenGL 的底层中并不罕见，有趣的地方就在于，我们可以使用这个渲染纹理类配合主动调用的绘图实现截图效果。下面的函数 saveScreenTo- RenderTexture 同样实现了截图功能：

```
CCRenderTexture* saveScreenToRenderTexture()
{
    CCSize winSize = CCDirector::sharedDirector()->getWinSize();
    CCRenderTexture* render  = CCRenderTexture::create(winSize.height, winSize.width);

    render->begin();
    CCDirector::sharedDirector()->drawScene();
    render->end();

    return render;
}
```

在上述代码中，CCRenderTexture 的 begin 和 end 接口规定了绘图转移的时机，在这两次函数调用之间的 OpenGL 绘图都会被绘制到一张纹理上。注意，这里我们主动调用了导演类的绘制场景功能。但是根据引擎的接口规范，我们不建议这样做，因为每次绘制都产生了 CCNode 类的 visit 函数的调用，但只要遵守不在 visit 中更改绘图相关状态的规范，可以保证不对后续绘图产生影响。

渲染纹理提供了两个导出纹理的接口，分别可以导出纹理为 CGImage 和文件，它们的定义如下：

```
CCImage* newCCImage();
bool saveToFile(const char *name, tCCImageFormat format);
```

感兴趣的读者可以查看 CCRenderTexture 的内部实现，其导出纹理的过程实际上也是利用 glReadPixels 函数来获取像素信息。因此，导出纹理这一步的效率和我们自己编写的 saveScreenToCCImage 函数是一致的。然而如果采用重新绘制的方式

来导出纹理则与此不同，一次屏幕的过程较为费时，尤其在布局比较复杂的场景上。重新绘制的强大之处在于绘制结果可以迅速被重用，非常适合做即时小窗预览之类的效果。下面的 saveScreen 方法实现了实时的截图功能：

```
void GameMenuLayer::saveScreen(CCObject* sender)
{
    //我们注释掉了旧的代码，改用 saveScreenToRenderTexture 方法来实现截图
    //CCImage* image = saveScreenToCCImage();
    //image->saveToFile("screen.png");
    //image->release();

    CCRenderTexture* render = saveScreenToRenderTexture();
    this->addChild(render);
    render->setScale(0.3);
    render->setPosition(ccp(CCDirector::sharedDirector()->getWinSize().width, 0));
    render->setAnchorPoint(ccp(1,0));
}
```

CCRenderTexture 继承自 CCNode，我们把它添加到游戏之中，就可以在右下角看到一个动态的屏幕截图预览了，如图 11-4 所示。

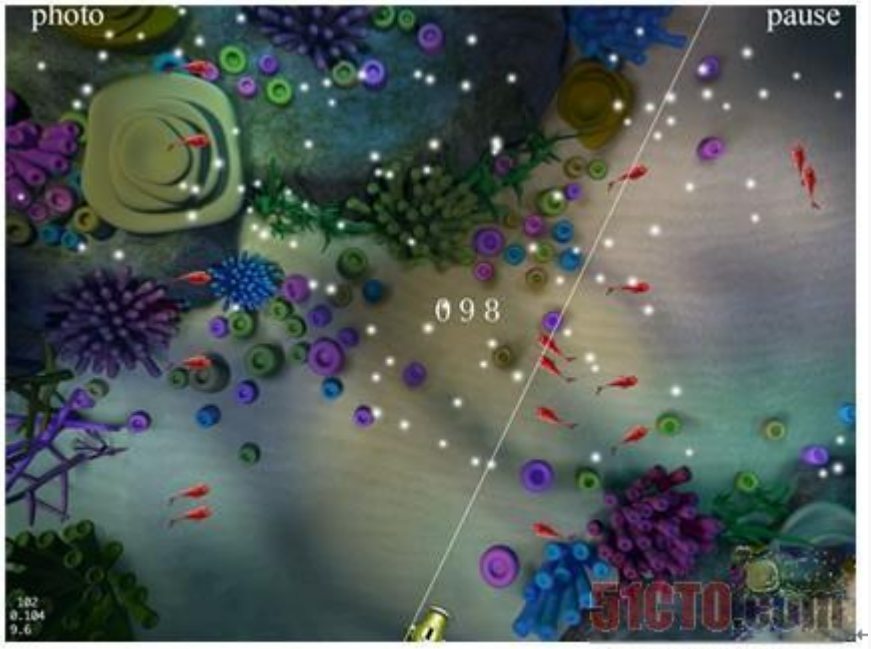


图 11-4　即时截图预览

11.4.1　可编程着色器

11.4　可编程管线

正如本章开始所说的那样，在 Cocos2d-x 中，最大的变革就是引入了 OpenGL ES 2.0 作为底层绘图，这意味着渲染从过去的固定管线升级到了可编程管线，我们可以通过着色器定义每一个顶点或像素的着色方式，产生更丰富的效果。着色器实际上就是一小段执行渲染效果的程序，由图形处理单元执行。之所以说是"一小段"，是因为图形渲染的执行周期非常短，不允许过于臃肿的程序，因此通常都比较简短。

11.4.1　可编程着色器

- 在渲染流水线上，存在着两个对开发者可见的可编程着色器，具体如下所示。
- 顶点着色器（vertex shader）。对每个顶点调用一次，完成顶点变换（投影变换和视图模型变换）、法线变换与规格化、纹理坐标生成、纹理坐标变换、光照、颜色材质应用等操作，并最终确定渲染区域。在 Cocos2d-x 的世界中，精灵和层等都是矩形，它们的一次渲染会调用 4 次顶点着色器。
- 段着色器（fragment shader，又称片段着色器）。这个着色器会在每个像素被渲染的时候调用，也就是说，如果我们在屏幕上显示一张 320×480 的图片，那么像素着色器就会被调用 153 600 次。所幸，在显卡中通常存在不止一个图形处理单元，渲染的过程是并行化的，其渲染效率会比用串行的 CPU 执行高得多。
- 这两个着色器不能单独使用，必须成对出现，这是因为顶点着色器会首先确定每一个显示到屏幕上的顶点的属性，然后这些顶点组成的区域被化分成一系列像素，这些像素的每一个都会调用一次段着色器，最后这些经过处理的像素显示在屏幕上，二者是协同工作的。
- 进一步深入介绍的就是顶点着色器编程语言和编程方法了。这里我们并不打算这么做，一来这是一门足够高深的学问，绝不是一两章能说清楚的；二来我们可以找到足够多的开源的着色器，能够提供各种丰富的效果，实际上需要自己编写着色器的场合并不太多。因此这里我们将着重介绍如何在 Cocos2d-x 游戏中导入自定义的着色器效果。

11.4.2 CCGLProgram

11.4.2 CCGLProgram

- 引擎提供了 CCGLProgram 类来处理着色器相关操作，对当前绘图程序进行了封装，其中使用频率最高的应该是获取着色器程序的接口：

```
const GLuint getProgram();
```

- 该接口返回了当前着色器程序的标识符。后面将会看到，在操作 OpenGL 的时候，我们常常需要针对不同的着色器程序作设置。注意，这里返回的是一个无符号整型的标识符，而不是一个指针或结构引用，这是 OpenGL 接口的一个风格。对象（纹理、着色器程序或其他非标准类型）都是使用整型标识符来表示的。
- CCGLProgram 提供了两个函数导入着色器程序，支持直接从内存的字符串流载入或是从文件中读取。这两个函数的第一个参数均指定了顶点着色器，后一个参数则指定了像素着色器：

```
bool initWithVertexShaderByteArray(const GLchar* vShaderByteArray,
    const GLchar* fShaderByteArray);
bool initWithVertexShaderFilename(const char* vShaderFilename,
    const char* fShaderFilename);
```

11.4.3 变量传递

11.4.3 变量传递

- 仅仅加载肯定是不够的，我们还需要给着色器传递运行时必要的输入数据。在着色器中存在两种输入数据，分别被标识为 attribute 和 uniform。
- attribute 变量是应用程序直接传递给顶点着色器的变量，在段着色器中不能访问。它描述的是每个顶点的属性，如位置、法线等，被限制为向量或标量这样的简单结构。必须为每个顶点指定对应的值，这类似于 C 中的函数参数。
- uniform 变量是全局性的，可以同时在顶点着色器和段着色器中访问。在整个渲染流水线中，每个 uniform 变量都是唯一的，不存在每个像素或顶点需要单独定义的问题，这一点是和 C 的全局变量类似的。uniform 变量的可定义类型会更丰富一些，还可以包括纹理矩阵和纹理，甚至可以通过 uniform block 自定义复杂的数据类型。
- 必须注意的是，虽然都被称为"变量"，但这仅仅是对于应用程序而言的。在着色器程序中，不管是顶点着色器还是段着色器，这些变量都是只读的，不允许在渲染过程中改变。

- 以上两种变量的传递都要经过获取位置和设置两步。对于 uniform 变量名，由于全局唯一，操作方法比较简单。在获取上，只有一个接口函数，其参数是当前绘图程序及需要获取的 uniform 变量名：

```
int glGetUniformLocation(GLuint program, const GLchar* name);
```

- 对 uniform 变量的设置存在着一系列以"glUniform"为前缀的函数，这些函数的第一个参数为需要设置的参数标识，后面跟若干参数值：

```
void glUniform1i(GLint location, GLint x);
```

- 上面的这个函数设置目标 uniform 变量的值为一个整型值。我们可以看到，OpenGL 的函数末尾总是紧接着类似"1i"和"3f"这样的后缀，这也是 OpenGL 函数的另一个特色。传值类函数可能会接受多种不同的参数，后缀中的数字从 1 到 4 分别对应标量和 2、3、4 维的向量。后缀中的另一个字符表示的是参数类型，常见的类型有"f"(float)、"i"(integer)、"b"(byte)等。这个做法可以避免类型转换，提高程序效率。
- 考虑到内存和显卡间数据交换的开销，引擎在 CCGLProgram 中进一步封装了一层缓冲机制，记录下每次传递的值，只有在传值不一的时候才真正设置到显卡数据中：

```
void CCGLProgram::setUniformLocationWith1i(unsigned int location, GLint i1);
```

- 对于 attribute 变量，直接使用会相对复杂一些，需要我们来处理变量的绑定与编号的管理等细节。CCGLProgram 中对此提供了一个封装，绑定一个 attribute 变量名到特定的标识符上，这样我们就可以直接通过名称来访问变量了：

```
void addAttribute(const char* attributeName, GLuint index);
```

- 绑定后的 attribute 变量可以通过标识符传值，传值的方式有两种。前面说过，attribute 变量是顶点着色器的参数，每绘制一个顶点就会调用顶点着色器一次，对应也就需要设定每次调用的 attribute 变量值。因此，根据同一个变量在不同调用间是否一致，可以分为一次性传值与数组传值。这两种传值方式是互斥的，要通过一组函数来切换：

```
void glEnableVertexAttribArray(GLuint index);
void glDisableVertexAttribArray(GLuint index);
```

- 这两个函数设置了是否开启某一特定 attribute 变量的数组传值，其中的参数 index 应该是我们曾经绑定过的某个 attribute 变量的标识符。
- 没有开启数组传值的 attribute 变量使用"glVertexAttrib"系列函数传值，与 uniform 变量的传值类似，该系列存在不同维数和不同类型向量的赋值方法，这里就不再赘述了。
- 开启数组传值的 attribute 变量则通过以下的接口函数传值：

```
void glVertexAttribPointer(GLuint indx,      //变量标识符
    GLint size,              //变量的维数
    GLenum type,             //组成变量的基本类型
    GLboolean normalized,    //是否归一化，一般为 false
    GLsizei stride,          //每次取值间隔
    const GLvoid* ptr);      //数组指针
```

- 其中需要解释的是第五个参数，它指定的是两个相邻的值在数组中的位置差，如果设置为 0，说明要传递的值是紧靠在一起的。灵活运用 stride 参数的特性，可以大大简化我们管理顶点属性的复杂度。例如，我们可以创建一个结构体，它包含了一个顶点的坐标、颜色以及纹理坐标这 3 个属性。在由结构体组成的数组中，不同顶点的坐标虽然不是连续排列的，但它们在内存中保存的间隔都是结构体的长度。因此，我们可以直接把结构体数组传递给函数，并利用 stride 参数使 OpenGL 正确设置顶点坐标等属性。
- 最后需要提两个调用时机的问题。

- 首先是绑定 attribute 变量的时机。CCGLProgram 在装载完毕后需要调用 link 函数连接着色器程序到显卡，绑定必须在连接之前，保证绑定能正确传递到显卡。
- 其次是设置开启数组绑定的时机。由于这是一个在不同渲染程序间共享的状态，会被不同的绘制反复开启关闭，所以必须在每次绘制时主动设置。

11.5.1 着色器程序

- 11.5 水纹效果
- 了解完如何添加着色器效果到程序后，我们来为游戏增加一个水纹效果，借此实践并体会更多处理着色器的细节。
- 11.5.1 着色器程序
- 首先来看我们即将添加的顶点着色器程序：

```
attribute vec4 a_color;
attribute vec4 a_position;
varying vec4 v_color;
uniform mat4 u_MVPMatrix;
void main()
{
    v_color = a_color;
    gl_Position = u_MVPMatrix * a_position;
}
```

- 段着色器程序：

```
varying vec4 v_color;

uniform sampler2D tex0;
precision highp float;
uniform float time;
uniform vec2 resolution;
const float PI = 3.1415926535897932;

//速度
const float speed = 0.2;
const float speed_x = 0.3;
const float speed_y = 0.3;

//几何参数
const float intensity = 3.;
const int steps = 8;
const float frequency = 4.0;
const int angle = 7; //设为一个素数效果较好

//反射与凸起参数
const float delta = 20.;
const float intence = 400.;
```

```
const float emboss = 0.3;

//---------- 水晶效果
float col(vec2 coord)
{
    float delta_theta = 2.0 * PI / float(angle);
    float col = 0.0;
    float theta = 0.0;
    for (int i = 0; i < steps; i++)
    {
        vec2 adjc = coord;
        theta = delta_theta*float(i);
        adjc.x += cos(theta)*time*speed + time * speed_x;
        adjc.y -= sin(theta)*time*speed - time * speed_y;
        colcol = col + cos( (adjc.x*cos(theta) - adjc.y*sin(theta))
            *frequency)*intensity;
    }

    return cos(col);
}

//---------- main 函数
void main(void)
{
    vec2 p = (gl_FragCoord.xy) / resolution.xy, c1 = p, c2 = p;
    float cc1 = col(c1);

    c2.x += resolution.x/delta;
    float dx = emboss*(cc1-col(c2))/delta;

    c2.x = p.x;
    c2.y += resolution.y/delta;
    float dy = emboss*(cc1-col(c2))/delta;

    c1.x += dx;
    c1.y += dy;
    float alpha = 1.+dot(dx,dy)*intence;
    gl_FragColor = texture2D(tex0,c1)*(alpha) *v_color*(alpha);
}
```

- 这组着色器程序实现在一个范围内添加一层指定颜色的水纹流动、反光效果，该效果是滤镜式的，不会掩盖背景层的绘制。在此我们不分析这组程序的每一个实现细节，只着重讲解如何将效果添加到游戏中。
- 我们提取着色器程序需要传递的变量，ShaderNode 的几个成员变量对应了这些变量的标识符和值，后面将逐一看到它们的取标识符和传值过程：

```
attribute   vec4            a_color;        //水纹颜色
```

```
attribute   vec4            a_position;     //顶点坐标
uniform     mat4            u_MVPMatrix;    //坐标变换矩阵
uniform     float           time;           //时间
uniform     vec2            resolution;     //分辨率
uniform     sampler2D       tex0;           //背景纹理
```


11.5.2 ShaderNode 类

考虑到众多烦琐的细节不可能在一个函数内完成，而且 Cocos2d-x 的绘制操作是分布离散的，我们参考引擎在测试样例中的做法封装一个 CCNode 的子类，以允许直接添加到游戏场景中：

```
class ShaderNode : public CCNode
{
public:
    ShaderNode();
    bool initWithVertex(const char *vert, const char *frag);
    void loadShaderVertex(const char *vert, const char *frag);
    virtual void update(ccTime dt);
    virtual void setContentSize(const CCSize& var);
    void setColor(ccColor4F newColor);
    virtual void draw();
    static ShaderNode* shaderNodeWithVertex(const char *vert,
        const char *frag);
private:
    //标识符
    GLuint      m_uniformResolution, m_uniformTime, m_uniformTex0;
    GLuint      m_attributeColor, m_attributePosition;
    //取值
    float       m_time;
    ccVertex2F  m_resolution;
    GLuint      m_texture;
    GLfloat     color[4];
};
```

首先，我们在初始化的时候获取这些变量的标识符：

```
void ShaderNode::loadShaderVertex(const char* vert, const char* frag)
{
    CCGLProgram* shader = new CCGLProgram();
    shader->initWithVertexShaderFilename(vert, frag);   //载入着色器程序

    //绑定 attribute 变量
    shader->addAttribute("a_position", 0);
    shader->addAttribute("a_color", 1);
    shader->link();

    //获取 attribute 变量标识
```

```
    m_attributeColor = glGetAttribLocation(shader->getProgram(), "a_color");
    m_attributePosition = glGetAttribLocation(
        shader->getProgram(), "a_position");
    shader->updateUniforms();

    //获取 uniform 变量标识
    m_uniformCenter = glGetUniformLocation(shader->getProgram(), "center");
    m_uniformResolution = glGetUniformLocation(shader->getProgram(), "resolution");
    m_uniformTime = glGetUniformLocation(shader->getProgram(), "time");
    m_uniformTex0 = glGetUniformLocation(shader->getProgram(), "tex0");

    //使用着色器程序
    this->setShaderProgram(shader);
    shader->release();
}
```

在 Cocos2d-x 中，由于可编程管线的使用，每个 CCNode 都会附有一个着色器程序。我们完成加载自定义的着色器后，应该调用 setShaderProgram 设置到 CCNode 中。最后再次强调，应该在链接着色器程序之前绑定 attribute 变量。

这样一来，初始化函数就比较简单了。初始化着色器之后，设置默认的显示区域大小，默认的水纹效果为偏蓝色，并设置初始化时间和定时更新。唯一特别的是，我们在这里添加了一张透明的图片到显卡作为纹理，后面会详细介绍；注意，这里获取了这张纹理在 OpenGL 中的标识符，而不是 Cocos2d-x 中的 CCTexture2D 指针。相关代码如下：

```
bool ShaderNode::initWithVertex(const char* vert, const char* frag)
{
    loadShaderVertex(vert, frag);
    m_texture = CCTextureCache::sharedTextureCache()
        ->addImage("transparent.png")->getName();

    setContentSize(CCSizeMake(1024, 768));
    setColor(ccc4f(0.5, 0.5, 1, 1));
    m_time = 0;
    scheduleUpdate();
    return true;
}
```

11.5.3 uniform 变量准备

11.5.3 uniform 变量准备

初始化着色器程序后，就应该准备需要传递的各个变量的值了。在两个 attribute 变量中，色彩直接通过设置函数设置，而顶点位置将直接在 draw 函数中传递。因此，我们先介绍如何准备 3 个 uniform 变量：时间、分辨率和背景纹理。

时间只需要在更新函数内刷新，相关代码如下：

```
void ShaderNode::update(ccTime dt)
{
    m_time += dt;
}
```

分辨率是指当前效果覆盖区域的分辨率。由于它与当前绘图区域有关，我们重写了设置函数，相关代码如下：

```
void ShaderNode::setContentSize(const CCSize& var)
{
    CCNode::setContentSize(var);
    m_resolution = vertex2(getContentSize().width, getContentSize().height);
}
```

- 背景纹理则可以认为是效果区域的截图，作为纹理被使用。sampler2D 是一种特殊的 uniform 变量，指代一张纹理，在传值时应该传递该纹理的标识符。如果遵循屏幕截图功能的思路，可以先截取屏幕图像到内存创建 CCImage，再使用 CCImage 创建一张纹理，但这样数据先从显卡流向内存，再从内存流向显卡，显然效率太低了。幸运的是，可以直接在显卡中进行数据交互获得这张纹理。这个操作并不复杂，OpenGL 提供了接口完成这一功能，具体如下所示：

```
void  glCopyTexImage2D (GLenum target, //目标纹理
    GLint level,                       //纹理细节级别，一般为 0
    GLenum internalformat,             //纹理内部格式，在 Cocos2d-x 中一般为 GL_RGBA
    GLint x, GLint y, GLsizei width, GLsizei height, //目标区域
    GLint border);                     //边框宽度，必须为 0
```

- 特别需要指出的是，这里的目标纹理并不是某一张纹理，而是已经激活的某一个特殊的纹理槽位，截取的绘制数据将存入绑定到该槽位的纹理中。在 Cocos2d-x 中，这个纹理槽位一般为 GL_TEXTURE_2D，即 2D 纹理槽位。
- 在初始化函数中，我们已经借助一张透明图创建了一张纹理，可以用来绑定到 2D 纹理槽位中。注意，我们是通过 CCTexture2D 来完成纹理创建的，而不是直接调用 OpenGL 的底层接口。这样做的好处在于，效率损失不大的情况下尽可能降低了复杂度，屏蔽了创建纹理过程中的一些细节。

11.5.4 绘制

11.5.4 绘制

- 一切准备妥当，最后是重写负责绘制的 draw 函数：

```
void ShaderNode::draw()
{
    CC_NODE_DRAW_SETUP();

    //传递 uniform 变量
    CCGLProgram* shader = getShaderProgram();
    shader->setUniformLocationWith2f(m_uniformCenter, m_center.x, m_center.y);
    shader->setUniformLocationWith2f(m_uniformResolution, m_resolution.x,
        m_resolution.y);
    shader->setUniformLocationWith1i(m_uniformTex0, 0);
    glUniform1f(m_uniformTime, m_time);

    //获取 attribute 变量
    CCSize size = this->getContentSize();
    float w = size.width;
    float h = size.height;

    ccGLBindTexture2D(m_texture);                //绑定纹理到槽位
    glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 0, 0, w, h, 0); //截取屏幕数据到纹理
    glEnableVertexAttribArray(m_attributePosition);
```

```
    glDisableVertexAttribArray(m_attributeColor);

    //传递 attribute 变量
    GLfloat vertices[12] = {
        0, 0, w, 0,
        w, h, 0, 0,
        0, h, w, h,
    };
    glVertexAttribPointer(m_attributePosition, 2, GL_FLOAT, GL_FALSE, 0, vertices);
    glVertexAttrib4fv(m_attributeColor, color);

    //绘制
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

- 简单地说，draw 函数被划分为若干部分：CC_NODE_DRAW_SETUP 宏函数用于准备绘制相关环境；使用绑定的着色器程序类传递 uniform 变量；获取绘制区域大小，截取纹理图片；直接调用底层接口传递 attribute 变量；最后是绘制。
- 首先需要解释的是纹理变量 tex0 的传递过程。在初始化 ShaderNode 类时，我们创建了一张纹理。为了获取屏幕截图数据，我们将其绑定到 2D 纹理的槽位。而在获取数据之前，我们已经为该纹理绑定了 tex0 变量；这是不影响最终结果的，因为绑定的只是该纹理的标识符，绘制时的纹理状态才是着色器程序最终使用的。
- 注意,这里我们没有直接使用 OpenGL 的接口绑定纹理,而使用了 ccGLBindTexture2D 这个 Cocos2d-x 的接口。这是 Cocos2d-x 的统一绑定接口，其内部做了一些缓存优化，可以提高纹理绑定的效率。由于其他的节点在绑定的时候也使用了这一优化接口，所以如果我们直接使用 OpenGL 的底层绑定，将带来一定的混乱。
- 其次是 attribute 变量的传递。这里演示了两种不同的传递方式：由于每个点的顶点坐标都不一致，使用了数组传值，在传递之前先开启了数组传递；而每个顶点的水纹颜色都是一致的，使用了常量传值方式。
- 最后要说的是绘制部分：调用了 glDrawArrays 函数，传递参数为 GL_TRIANGLES（意味着会在屏幕上绘制三角形），指定了 6 个点（也就是用两个三角形组成了需要绘制的矩形）。这也是为什么顶点数组传递了 12 个浮点数，每两个构成一个二维平面坐标，一共 6 个坐标点。

11.5.5 添加到场景

11.5.5 添加到场景

- 最后我们将这个水纹效果的滤镜添加到场景中，覆盖在背景和鱼的层级之上，营造出水波荡漾的效果。考虑到还需要保证炮台在水纹之上，所以将该水纹效果添加到精灵层中：

```
ShaderNode* shader = ShaderNode::shaderNodeWithVertex("shader.vsh", "shader.fsh");
shader->setContentSize(winSize);
shader->setColor(ccc4f(0.5, 0.5, 1, 1));
this->addChild(shader, 10);
```

- 我们将水纹的 Z 坐标设定为 10，随之确定的是鱼的 Z 坐标应该在 10 以下，而炮台和子弹的 Z 坐标则应该在 10 以上。运行游戏，可以看到如图 11-5 所示的效果。

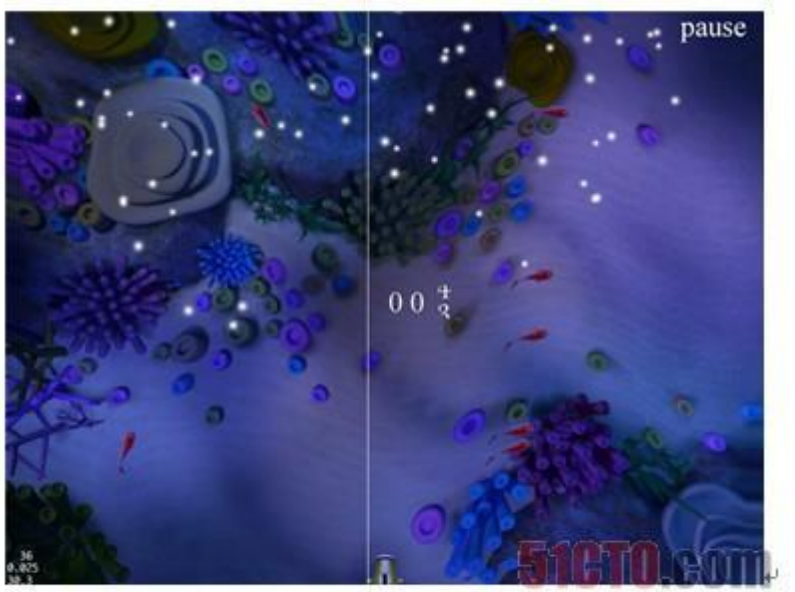


图11-5 着色器效果

- 11.6 CCGrid3D

11.6 CCGrid3D

在 OpenGL 和 Cocos2d-x 的世界来回穿梭了这么久，不知道读者有没有一点晕头转向的感觉。Cocos2d-x 的设计允许我们在游戏的任何部分自由地使用 OpenGL，这为我们带来了无与伦比的灵活性，但同时也必须注意，滥用 OpenGL 会使得代码变得混乱而难以维护，因此除非迫不得已，不应首先考虑 OpenGL。

引擎封装了一个特殊的动作类 CCActionGrid3D，可以模拟一些简单的 3D 特效，在一些情况下可以代替 OpenGL。恰好引擎利用 CCActionGrid3D 提供了一个类似于我们实现的水纹效果的波浪效果动作，下面我们就利用 Cocos2d-x 自带的动作来实现水纹效果。

这个特效动作类的使用非常简单，先看如何用其代替我们之前实现的效果。在开始场景的初始化中加入下面的两行代码：

```
CCGrid3DAction *grid = CCWaves3D::create(50, 40, ccg(10, 10), 10);
this->runAction(grid);
```

运行游戏，可以看到如图 11-6 所示的效果。由于这是一个一切都已经封装好的动作，只要添加两条语句，就已经非常接近我们自己编写的着色器的效果了（除了没有光影）。

图 11-6 水纹效果

- 与自定义着色器相比，CCActionGrid3D 局限于表现一些使画面变形的效果，其本质是将目标节点所在区域划分为网格，对每一个小网格进行坐标变换从而形成画面的特殊扭曲。正因为此，它无法改变光照与颜色的渲染方式。
- 不妨看看上面的 3D 波浪效果的关键部分--update 方法，其代码如下所示：

```
void CCWaves3D::update(ccTime time)
{
    int i, j;
    for (i = 0; i < m_sGridSize.x + 1; ++i)
    {
        for (j = 0; j < m_sGridSize.y + 1; ++j)
        {
            ccVertex3F v = originalVertex(ccg(i ,j));
            v.z += (sinf((CCFloat)M_PI * time * m_nWaves * 2
                + (v.y+v.x) * .01f) * m_fAmplitude * m_fAmplitudeRate);
            CCLog("v.z offset is %f\n", (sinf((CCFloat)M_PI * time * m_nWaves * 2
                + (v.y+v.x) * .01f) * m_fAmplitude * m_fAmplitudeRate));
            setVertex(ccg(i, j), v);
        }
    }
}
```

- 在上述代码中，我们设置了一个循环，对于预定义的每一个网格进行迭代：先是获取其原来的空间坐标，然后根据网格位置和时间设置 Z 轴的变换。注意，这里 Z 轴的意义不同于 Cocos2d-x 坐标系下的 Z 轴，它是 OpenGL 绘图体系中的三维坐标之一，Z 坐标大的顶点会"显得"离屏幕更近一些。
- CCActionGrid3D 把绘制对象划分为一个较为精细的网格，我们设置网格中每个顶点的坐标，GPU 会平滑地处理每个网格的变形效果，只要顶点坐标变换的范围不要过大，并且网格大小适中，产生的变形就能保持平滑。
- 对于我们自定义的变换效果，也只需要继承 CCActionGrid3D 并重写 update 方法中的坐标变换即可。

11.7 再议效率

■ 11.7 再议效率

■ 细心的读者应该已经留意到了这些效果的效率问题。

■ 首先一个不可回避的事实是，引擎提供的封装效果在效率上是一定会有损失的。以 CCActionGrid3D 为例，全屏使用其波浪效果，帧数基本下降到了个位数，究其原因，是产生了太多的绘制，如果我们划定网格大小为 10×10，320×480 的屏幕上就存在着 32×48=1536 个网格，也就是一帧要额外进行 1536 次绘制，产生的消耗代价是较大的。再例如 CCDrawingPrimitive 中的简单绘制效果，它能够绘制出简单的几何图形，但一条线段就会耗费一次绘制，代价不小。

■ 另一个值得明确的是，OpenGL ES 2.0 提供的可编程管线在效率上必然是弱于 OpenGL ES 1.0 的硬件渲染的，即便是实现同样的效果。由于在可编程管线的架构上，这些效果实际上是内置的程序，比起硬件的渲染效果，效率必然会有损失。另一方面，我们在引入自定义着色器特效时，也容易在着色器本身以及内存和显卡间数据交换两方面形成效率损失。

■ 任何新技术都是一把双刃剑，我们在使用任何底层绘图的时候都务必注重效率。

11.8 小结

■ 11.8 小结

■ 在这一章中，我们为游戏引入了几个新的特性--瞄准线、滚动数字效果、截图和水纹波浪效果，通过这几个效果了解了几种底层绘图的方法。下面总结这一章的知识点。

■ draw 方法：它是每个 CCNode 都提供的一个方法，用于在绘制节点的时候绘制自身内容。通常，所有的自定义绘图都应该在这个方法中完成。在绘制自定义内容的时候，需要重载 draw 方法，然后在 draw 方法中加入绘图代码。

■ glScissor：用于实现裁剪绘图。使用 glScissor 函数前，需要调用 glEnable (GL_SCISSOR_TEST)来启用裁剪，设置裁剪区域后所绘制的图形只有区域内的部分会被真正绘制，其余部分保持透明。

■ 纹理图片：CCImage 与 CCTexture 分别代表一张图片和一个可载入到显存的纹理。CCTexture 可由 CCImage 创建，而 CCImage 可以载入或保存 PNG、TIFF、JPG 等格式的

■ 文件。

■ 着色器：着色器是用于代替渲染流水线的一段程序，可以用来实现对顶点和像素的操作。在这一章中，我们使用着色器实现了水纹效果。

■ CCGrid3D：Cocos2d-x 提供一套网格变换的功能，通过 CCActionGrid3D 动作类可以实现一些简单画面的 3D 变换，例如水纹效果。

■ 这一章讲解得并不深入，但是对于日常的 2D 游戏开发来说已经足够，感兴趣的读者不妨进一步研读 OpenGL 和计算机图形学方面的资料。

12.1 新的超级武器

● 物理引擎

■ 在开始介绍物理引擎之前，请读者先回顾下我们的《捕鱼达人》。迄今为止，我们已经可以非常娴熟地运用 Cocos2d-x 创建精灵，设置位置，创建和组合各种复杂的动作，播放动画。鱼群活灵活现地在屏幕上欢快地游动，我们似乎完美地"模拟"了一个真实的海底世界！唯一可惜的是，目前我们的海底世界只有互不干扰的游动，打出的子弹也仅仅是直线飞行，撒网之后就会消失，因此我们也只需要很简单的碰撞检测方法。与现实世界相比，仍然缺少碰撞的真实感。

■ 在这一章中，我们将试图在《捕鱼达人》中引入一种全新的超级武器，并借用物理引擎来给超级武器带来华丽且震撼的效果。

12.1 新的超级武器

我们为游戏增加一个超级武器--能量球，它可以在发生碰撞后弹开，然后速度越来越慢，直到能量耗尽为止。现有的碰撞检测机制并不能告诉我们能量球碰撞后应该如何运行，显然，我们需要更复杂的方法来实现新的碰撞机制。为了实现这个超级武器，我们需要维护能量球的速度等属性，以及通过碰撞计算能量球的运行轨道，这种对运动对象的模拟成了我们更进一步的挑战。

物理引擎正是为这样的场景而生的。物理引擎充分考虑了游戏对象的物理性质，并尽可能精确地计算出物体之间的相互作用效果。通常，物理引擎会把游戏中出现的对象与一个物理模型绑定起来。模型包含了物体的形状、密度、材质和速度等属性，引擎会根据模型间的相互作用实时地改变模型属性，并借用游戏引擎的渲染环节把物体表现出来。

物理引擎通过为刚性物体赋予真实的物理属性的方式来计算运动、旋转和碰撞反应，免去自己编程模拟的麻烦，也允许游戏更精准地还原真实物理世界里的种种现象。当然，这些功能并不是免费获得的，模拟物理世界需要耗费大量的运算时间与存储空间，不过通常来说，由于引擎内部已经经过一定优化，效率上并不会比手工编码的实现更坏。

这些物理引擎通常都经过高度抽象和封装，与游戏引擎或绘图引擎相独立，可以单独导入游戏项目。

12.2 Box2D 引擎简介

12.2 Box2D 引擎简介

Box2D 是与 Cocos2d-x 一起发布的一套开源物理引擎，也是 Cocos2d-x 游戏需要使用物理引擎时的首选。二者同样提供 C++开发接口，所使用的坐标系也一致，因此 Box2D 与 Cocos2d-x 几乎可以做到无缝对接。

Box2D 是一套基于刚体模拟的物理引擎，它的核心概念为世界、物体、形状、约束和关节，这些概念具体实现为 Box2D 的各个组件，它们的描述见表 12-1。

表 12-1 Box2D 的各个组件及其描述

组件	描述
世界（b2World）	一个物理世界。物理世界就是物体、形状和约束相互作用的集合。Box2D 允许在同一程序中创建多个世界，但通常这是不必要的
物体（b2Body）	物理引擎
夹具（b2Fixture）	一种用于把形状附加到物体之上的关系。我们利用形状创建夹具，再把夹具附加到物体之上，从而使得物体拥有碰撞的能力
形状（b2Shape）	物体的形状。一个严格依附于物体的 2D 碰撞几何结构，具有摩擦（friction）和恢复（restitution）等材料性质
约束	约束，就是消除物体自由度的物理连接。在 2D 世界中，一个物体有 *X* 方向、*Y* 方向和旋转角度 3 个自由度。如果我们把一个物体钉在墙上（如同一个摆锤），那就把它约束到了墙上。此时，此物体就只能绕着这个钉子旋转，所以这个约束消除了它的 3 个自由度
关节（b2Join）	一种用于把两个或多个物体固定到一起的约束。Box2D 支持的关节类型有旋转、棱柱和距离等。可以限制一个关节的活动范围，可以通过关节驱动所连接物体的转动

■

为了实现一个物理场景，我们需要做的是创建一个世界，然后创建我们需要的物体，设置好它的形状和其他属性，将其添加到世界当中，然后周期性地更新这个世界，那么 Box2D 就会为我们高效且出色地完成模拟物理运动的任务。

■ 值得注意的是，Box2D 仅仅更新它所管理的物理模型的位置和速度等信息。在游戏中，我们想要做的通常是赋予精灵物理性质，我们会为精灵创建 Box2D 物理模型，然而物理引擎运作起来后，为了把物理模型的运动体现在屏幕上，我们必须手动把物理模型的数据同步到精灵中。

● 12.3 接入 Box2D（1）

■ 12.3 接入 Box2D（1）

■ 下面我们就来着手接入 Box2D 到游戏中。

■ Box2D 相关的头文件为"Box2D/Box2D.h"。和"cocos2d.h"一样，该头文件囊括了 Box2D 需要用到的所有头文件。Box2D 没有自己的命名空间，所以不需要相关的命名空间引用。

■ 接入 Box2D 的每个精灵在物理世界中都有相关联的物理模型"物体"，即 b2Body。首先，我们扩展引擎的精灵类来保存这一关联：

```
class B2Sprite : public CCSprite
{
public:
    static B2Sprite* spriteWithSpriteFrameName(const char* file);
    bool initWithSpriteFrameName(const char *pszSpriteFrameName);

    CC_SYNTHESIZE(b2Body*, m_b2Body, B2Body);    //物理世界的“物体”
    CC_SYNTHESIZE(bool, m_isDead, IsDead);       //是否死去
    CC_SYNTHESIZE(bool, m_isAlive, IsAlive);     //是否存活
};
```

■ 此处我们添加了 3 个属性：B2Body、IsDead 与 IsAlive。B2Body 表示的是物理引擎中的物体对象。对于其余两个属性，若 IsAlive 属性为 true，则物体"存活"，代表仍然存在碰撞体积，需要考虑碰撞，也就是仍然存在于 Box2D 的物理世界中；若 IsDead 属性为 true，则物体"死去"，表示彻底从屏幕上消失。由于鱼在确认被击中之后存在一个挣扎画面，此时尽管不再存在被碰撞的可能，但仍会在屏幕上存在一段时间，我们特意设置以上两个属性来作为区别。

■ 考虑到 Box2D 并不是游戏的核心组成，所占的代码量并不大，我们将其相关操作都集中封装到一起，避免零星地散落在不同的类中。Box2dHandler 类的代码如下所示：

```
class Box2dHandler : public CCNode, public b2ContactListener
{
    GLESDebugDraw* m_debugDraw;
    b2World* m_world;
    typedef pair<b2Fixture*, b2Fixture*> MyContact;
    set<MyContact> m_contacts;

    bool initBox2D();
    void addBodyForSprite(B2Sprite* sprite, double density = 10.0);
    void addFixtureForSprite(B2Sprite* sprite, double density=10.0);
    void dealCollisions();

```

```
public:
    virtual void BeginContact(b2Contact* contact);
    virtual void EndContact(b2Contact* contact);
    virtual void PreSolve(b2Contact* contact, const b2Manifold* oldManifold);
    virtual void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse);

    static Box2dHandler* handler();
    bool init();
    void draw();
    void update(ccTime dt);
    void addFish(B2Sprite* fish);
    void addBall(B2Sprite* ball, CCPoint direction);
    void removeSprite(B2Sprite* node);
    CC_SYNTHESIZE(Box2dHandlerDelegate*, m_delegate,Delegate);
};
```

- Box2dHandler 类负责和 Box2D 相关的所有操作，下面我们将一一讲解其中的方法与属性，也请读者和我们一起借此探索如何将 Box2D 的世界接入 Cocos2d-x 的世界中。
- 首先，我们在 Box2dHandler 的初始化方法中初始化了 Box2D 的世界，相关代码如下：

```
bool Box2dHandler::initBox2D()
{
    CCSize s = CCDirector::sharedDirector()->getWinSize();
    b2Vec2 gravity;
    gravity.Set(0.0f, 0.0f);

    m_world = new b2World(gravity);
    m_world->SetAllowSleeping(true);
    m_world->SetContinuousPhysics(true);
    m_world->SetContactListener(this);
    return true;
}
```

- 这个初始化操作较为简单，我们用无重力状态创建了一个物理世界，并设置了两个状态：允许睡眠和开启连续物理测试。允许睡眠，可以提高物理世界中物体的处理效率，只有在发生碰撞时才唤醒该对象。开启连续物理测试，这是因为计算机只能把一段连续的时间分成许多离散的时间点，再对每个时间点之间的行为进行演算，如果时间点的分割不够细致，速度较快的两个物体碰撞时就可能会产生"穿透"现象，开启连续物理将启用特殊的算法来避免该现象。最后设置了碰撞事件的监听器，碰撞相关的内容将在 12.6 节专门讲述。
- 接下要为我们的精灵创建属于它们的 Box2D 物体。然而在此之前，我们必须介绍 Box2D 世界里的最佳物体大小。根据 Box2D 参考手册，Box2D 针对大小为 0.1~10 个单位长度的物体作了优化，处理效率相对高一些。因此，我们在 Box2D 世界中创建的物体也应该尽量遵循这一大小的限制。考虑到屏幕上的可见对象都是以像素定义的，一条鱼的大小就是 80×10 像素，若直接使用像素数值来作为 Box2D 物体的大小，显然已经严重超出了 Box2D 建议的最佳大小。我们定义一个转换率 PTM_RATIO，单位为"像素/米"，以便把物体尺度在 Cocos2d-x 与 Box2D 中相互转换。在许多游戏中，PTM_RATIO 取 32。

12.3 接入 Box2D（2）

■ 12.3 接入 Box2D（2）

■ 现在我们可以为精灵创建 Box2D 中对应的物体了。我们将这一过程拆分成了两个步骤：第一步使用 addFixtureForSprite 方法为已经创建了 body 的精灵添加夹具（fixture），供接下来的步骤使用；第二步使用 addBodyForSprite 方法为一个精灵创建物体（body），供其他代码调用。这样拆分的好处在于，对于普通子弹碰撞后的撒网的动作，我们只需要重新设置它的夹具，而并不需要重新创建 Box2D 物体，此时，就可以减少重复创建物体的代码了。正如前面介绍的，在 Box2D 模拟的物理世界中，一个物体可以附加多个夹具，每个夹具代表一个特定密度、形状的部件，而物体的主要特征则集中表现为整个物体在空间中的位置和速度信息。addFixture- ForSprite 方法和 addBodyForSprite 方法的代码如下所示：

```
void Box2dHandler::addFixtureForSprite(B2Sprite* sprite, double density)
{
    b2PolygonShape spriteShape;
    CCSize size = sprite->getContentSize();

    spriteShape.SetAsBox(size.width / PTM_RATIO / 2, size.height / PTM_RATIO / 2);

    b2FixtureDef spriteShapeDef;
    spriteShapeDef.shape = &spriteShape;
    spriteShapeDef.density =density;

    b2Body *spritespriteBody = sprite->getB2Body();
    spriteBody->CreateFixture(&spriteShapeDef);
}
void Box2dHandler::addBodyForSprite(B2Sprite* sprite, double density)
{
    b2BodyDef spriteBodyDef;
    spriteBodyDef.type = b2_dynamicBody;
    spriteBodyDef.position.Set(sprite->getPosition().x / PTM_RATIO,
        sprite->getPosition().y / PTM_RATIO);
    spritespriteBodyDef.userData = sprite;

    b2Body *spriteBody = m_world->CreateBody(&spriteBodyDef);
    sprite->setB2Body(spriteBody);

    this->addFixtureForSprite(sprite,density);
}
```

■ b2Body 中的 userData 属性允许存放一个任意类型的指针指向我们自定义的数据。在我们的例子中，此处可以存放物体对应精灵的指针，方便后续的更新，这与 b2Sprite 中的 m_b2Body 属性是对应的。

■ 和 Cocos2d-x 类似，在 Box2D 的世界中，世界一物体一部件也呈现树状的从属关系。而不同的是，我们不能随意地创建物体或组件，必须通过上一级对象提供的以"Create"为前缀的接口来创建对象。一个对象一旦被创建，就从属于它的创建者：从 m_world 创建的 spriteBody 已经被添加到 m_world 中，而由 spriteBody 创建的夹具也已经是该 spriteBody 的一部分。

■ 在创建对象时，由于创建对象需要的参数可能会比较多，Box2D 提供了诸如 b2BodyDef 和 b2FixtureDef 等定义参数的结构体，用于传递创建所需参数。因此，读者遇到这种情况不必惊慌，只需要逐个参数赋值即可。在这里，我们暂且定义所有

的物体都由多边形组成，并定义为矩形，矩形的形状和大小直接通过精灵的可视大小转换得来。后面我们将会看到如何进一步精确地模拟一个对象的碰撞。除了最常用的多边形外，Box2D 还提供了圆形、条形和链状 3 种不同的形状。

■ 此前已经提到过 PTM_RATIO 参数。在 Cocos2d-x 对象和 Box2D 对象中，尺度转换都是通过 PTM_RATIO 来完成的。

● 12.4 更新状态

■ 12.4 更新状态

■ 当我们创建好精灵及其对应的 Box2D 对象后，不会看到任何效果，因为它们仅仅是被创建。在 12.2 节中我们介绍 Box2D 引擎时，曾提到 Box2D 是一个独立的引擎，并不是 Cocos2d-x 的一部分，因此需要通过"更新状态"的操作来联系游戏引擎与物理引擎，以实现二者同步，这样我们才能在游戏中看到物理现象。"更新状态"包含两步：第一步是周期性地更新物理引擎，第二步是利用物理演算的结果修改游戏中精灵的属性。

■ 为了实现更新的这两个步骤，首先需要在游戏中创建一个定时器调用物理引擎的更新方法，以便驱动引擎的演算。其次，需要定时获取物理引擎演算的结果，用物体当前的状态更新精灵的属性。为此，我们添加一个每帧调用的 update 定时器，用于刷新 Cocos2d-x 和 Box2D 中对象的位置：

```
● void Box2dHandler::update(ccTime dt)
● {
●     for(b2Body* b = m_world->GetBodyList(); b; bb = b->GetNext()) {
●         if (b->GetUserData() != NULL) {
●             B2Sprite* sprite = static_cast<B2Sprite*>(b->GetUserData());
●
●             if(sprite->getTag() == ball_tag) {
●                 b2Vec2 pos = b->GetPosition();
●                 float rotation = - b->GetAngle() / 0.01745329252f ;
●
●                 sprite->setPosition(ccp(pos.x * PTM_RATIO, pos.y * PTM_RATIO));
●                 sprite->setRotation(rotation);
●             }
●             else {
●                 b2Vec2 b2Position = b2Vec2(sprite->getPosition().x / PTM_RATIO,
●                     sprite->getPosition().y / PTM_RATIO);
●                 float32 b2Angle = -CC_DEGREES_TO_RADIANS(sprite->getRotation());
●
●                 b->SetTransform(b2Position, b2Angle);
●             }
●         }
●     }
●     m_world->Step(dt, 8, 8);
●     this->dealCollisions();
● }
```

■ 在 Box2D 的对象结构中，一系列对象都是以顺序列表的形式存放在父对象中，可以通过 GetList 系列方法（例如 GetBodyList 方法）获得该列表。这里我们就用 GetBodyList 获得了 m_world 中所有物体的列表，并在 for 循环中迭代物理世界中的每一个物体对象，根据对象的类型作不同的更新。

■ 对于鱼来说，它们游动的路线是由我们定义的，不受物理引擎演算的影响。因此，我们首先根据精灵的状态（例如位置和旋转角度等）设置 Box2D 中对应物体的位置和旋转角度。

- 而对于超级武器中出现的弹性球，我们把它的运动交给 Box2D 来计算，模拟其碰撞和弹射的特性。因此，在此处应该从 Box2D 对象中获取位置和旋转信息，设置到精灵上。
- 更新位置信息后，调用 Box2D 世界的刷新接口 Step：

```
void b2World::Step(float32 timeStep,int32 velocityIterations, int32 positionIterations)
```

- 这个接口的 3 个参数都十分重要。第一个参数 timeStep 是更新引擎的间隔时间，也就是距离上一次更新物理引擎后经过的时间，一般只需要直接传递 update 定时器提供的时间间隔即可；而后两个参数 velocityIterations 和 positionIterations 分别代表计算速度和位置时迭代的次数，下面介绍这两个参数的含义。
- 前面曾提到，Box2D 是用离散的时间点来演算连续的物理事件的，换句话说，它把物理世界的连续变化切分为许多片段逐个计算，如同数学中的微分操作一样。这两个迭代次数控制的就是微分的粒度，迭代次数越多，则模拟越精细，效果越细腻，但相应地，耗费的计算量也越大。一般情况下，我们将这两个参数设置为 8~10。
- 在 update 方法的最后，我们调用 dealCollisions 方法来处理碰撞事件，这个方法将在后面详细介绍。

12.5 调试绘图

- 12.5 调试绘图

- 到目前为止，我们完成了游戏引擎与物理引擎的同步，并且已经可以借助 Cocos2d-x 的定时器来驱动物理引擎演算了。然而我们还无法观察到任何与之前不同的现象--我们还没有在场景中添加能量球，而鱼群又不随物理引擎的规则更新状态，因此目前还无法确认物理引擎的工作状态。其实为了便于开发者调试，Box2D 提供了调试模式，在调试模式下，我们可以把 Box2D 中的对象绘制到屏幕上，由此观察 Box2D 世界中各个物体的位置信息。
- 幸运的是,Box2D 的调试绘图也是使用 OpenGL 作为绘图引擎的,于是 Box2D 和 Cocos2d-x 恰好可以无缝结合到一起。与 Box2D 调试绘图相关的封装已经在 Cocos2d-x 测试工程中提供给我们了，它们是位于 Cocos2d-x 目录中"tests/tests/Box2DTestBed"目录下的"GLES-Render.h"和"GLES-Render.cpp"，我们需要将这两个文件添加到工程文件中，并在 Box2dHandler 的初始化部分添加调试绘图的初始化。
- 首先，在文件中引用"GLES-Render.h"头文件。然后，我们在 Box2dHandler 的初始化方法中添加如下代码：

```
m_debugDraw = new GLESDebugDraw( PTM_RATIO );
m_world->SetDebugDraw(m_debugDraw);
uint32 flags = 0;
flags += b2Draw::e_shapeBit;
//flags += b2Draw::e_jointBit;
//flags += b2Draw::e_aabbBit;
//flags += b2Draw::e_pairBit;
//flags += b2Draw::e_centerOfMassBit;
m_debugDraw->SetFlags(flags);
```

- 在上面的代码中，我们首先创建了一个 GLESDebugDraw 对象，这是 Cocos2d-x 测试项目提供给我们的调试绘图工具。然后，我们为创建的 Box2D 世界对象设置刚才创建的调试绘图工具。最后，我们需要设置调试绘图模式中显示的内容。现在我们只让引擎显示出形状对象，因此为绘图工具设置 b2Draw::e_shapeBit 状态位。如果需要绘制更多的对象，只需要设置其他的状态位即可。
- Box2dHandler 继承自 CCNode，因此可以当做一个游戏元素添加到绘图树中。为了在屏幕上显示调试绘图模式的内容，我们重写 draw 方法，让调试绘图能显示到屏幕上：

```
void Box2dHandler::draw()
{
```

```
    CCNode::draw();

    ccGLEnableVertexAttribs(kCCVertexAttribFlag_Position);

    kmGLPushMatrix();
    m_world->DrawDebugData();
    kmGLPopMatrix();
}
```

在 draw 方法中，我们需要在绘图前保存 OpenGL 绘图状态，以免接下来的操作影响之后的绘图。因此，我们首先调用 kmGLPushMatrix 函数来保存状态，再调用 m_world 的 DrawDebugData 方法来绘制调试图形，最后调用 kmGLPopMatrix 函数来恢复刚才保存的状态。

成功添加后，将可以看到一个尽管不太漂亮但足以看出效果的 Box2D 世界--所有的鱼都被橙色的半透明矩形包围住了，矩形会跟随鱼的动作变化位置和旋转方向，如图 12-1 所示。

图12-1 调试绘图

12.6 碰撞检测

12.6 碰撞检测

完成前面的工作后，实际上已经把游戏中的对象和物理引擎中的对象关联了起来。超级武器"能量球"发射之后会碰到鱼群，被碰到的鱼群就会被直接捕捉到，因此我们需要得知什么时候、在哪里产生了碰撞。

在每次更新引擎时，物体可能会产生碰撞。Box2D 提供了多个碰撞事件，用于提供给开发者处理物体的碰撞。碰撞事件通过代理类 b2ContractListener 通知，为了监听碰撞事件，我们需要继承并实现以下 4 个事件回调函数：

```
virtual void BeginContact(b2Contact* contact);                                  //碰撞开始
virtual void EndContact(b2Contact* contact);                                    //碰撞结束
virtual void PreSolve(b2Contact* contact, const b2Manifold* oldManifold);  //碰撞处理前
virtual void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse); //碰撞处理后
```

Box2D 对碰撞的处理分为两个步骤，具体如下所示。

- 碰撞检测：发现物体间重叠。BeginContact 和 EndContact 事件会在两个物体开始和结束重叠时通知。
- 碰撞处理：根据物理定律处理速度改变和位置变化等效果，PreSolve 和 PostSolve 正是对应碰撞处理的前后时间点。由于我们只需要处理碰撞事件，所以只需要实现 BeginContact 和 EndContact 两个回调接口即可。
- 这里有一个需要注意的地方：我们不能直接在 BeginContact 和 EndContact 中改变物理引擎的状态，这是因为这两个事件的调用都是在 Box2D 的 b2World::Step 方法中，此时正在进行物理世界的演算，是不允许从外部对物理世界中的物体状态进行改变的。例如，我们不能在碰撞事件中直接删除被捕捉鱼的 Box2D 对象，只能用一个容器把需要处理的事件保存起来，待 Step 方法执行完毕后再异步处理这些碰撞。
- 在代码中添加对 BeginContact 和 EndContact 两个事件的响应，具体如下所示：

```
inline bool notFish(const int& tag)
{
    return tag == net_tag || tag == bullet_tag || tag == ball_tag;
}
void Box2dHandler::BeginContact(b2Contact* contact)
{
    CCSprite* spa = static_cast<CCSprite*>(contact->GetFixtureA()
        ->GetBody()->GetUserData());
    CCSprite* spb = static_cast<CCSprite*>(contact->GetFixtureB()
        ->GetBody()->GetUserData());

    int ta = spa->getTag();
    int tb = spb->getTag();

    if(ta == tb || ((notFish(ta)) + (notFish(tb))) == 2)
        return;

    MyContact myContact(contact->GetFixtureA(), contact->GetFixtureB());
    m_contacts.insert(myContact);
}
void Box2dHandler::EndContact(b2Contact* contact)
{
    MyContact myContact(contact->GetFixtureA(), contact->GetFixtureB());
    m_contacts.erase(myContact);
}
```

- 此处我们创建了 m_contacts 容器，用于保存一次更新中收集到的碰撞事件。在 Begin- Contact 的回调中，我们对两个碰撞物体所对应的精灵类型进行判断，屏蔽了鱼和鱼的碰撞以及子弹和子弹的碰撞后，将碰撞事件加入容器中。容器在 Box2dHandler 中定义，以集合的形式存放，以避免物理引擎有时出现的重复碰撞。声明集合 m_contacts 的代码如下：

```
typedef pair<b2Fixture*, b2Fixture*> MyContact;
set<MyContact>  m_contacts;
```

- 最后则是具体的碰撞处理函数，这里我们将 m_contacts 集合中的碰撞事件都分发到事件处理的代理类中去，完成碰撞后游戏状态的处理，相关代码如下：

```
void Box2dHandler::dealCollisions()
{
```

```
    if(m_delegate != NULL) {
        set<MyContact>::iterator it;
        for(it = m_contacts.begin(); it != m_contacts.end(); ++it) {
            B2Sprite *bullet = static_cast<B2Sprite*>(
                it->first->GetBody()->GetUserData());
            B2Sprite *fish =  static_cast<B2Sprite*>(
                it->second->GetBody()->GetUserData());

            if(notFish(fish->getTag()))
                swap(bullet, fish);

            m_delegate->collisionEvent(bullet, fish);
        }
    }
    m_contacts.clear();
}
```

- 这个代理将由精灵层实现，其中要对事件作相应的处理，包括鱼的捕捉确认、捕捉后将鱼从 Box2D 世界中收回等，在此就略去不讲了。

12.7 弹射

12.7 弹射

- 完成了碰撞检测，可以继续让我们的游戏更有趣一点，使发射的能量球能够在鱼群中弹射：击中一条鱼后可以根据原来的射击方向弹射，改变能量球下一步的方向，能量球的速度也会适当地慢下来--就像真实世界中的弹射一样。
- 其实需要做的并不多，只要在创建能量球在物理引擎中的对象时，合理设置初始状态即可，相关代码如下：

```
void Box2dHandler::addBall(B2Sprite* ball, CCPoint direction)
{
    this->addBodyForSprite(ball, density_ball);

    b2Body* body = ball->getB2Body();
    b2Vec2 b2Position = b2Vec2(ball->getPosition().x / PTM_RATIO,
        ball->getPosition().y / PTM_RATIO);
    float32 b2Angle = -CC_DEGREES_TO_RADIANS(ball->getRotation());

    body->SetTransform(b2Position, b2Angle);
    body->SetLinearVelocity(b2Vec2(direction.x * 100, direction.y * 100));
}
```

- 我们为能量球设置了速度，不过单单这样还不够。因为鱼的位置是根据精灵状态更新的，所以鱼在物理世界中并没有速度，与高速运行中的能量球碰撞后并不足以产生足够的冲量改变能量球的方向。回顾 addFixtureForSprite 函数，我们设置了一个参数 density，用来指定创建的夹具的密度，在这里可以派上用场了。我们将鱼的密度设置得比能量球大一些，以实现弹射的效果：

```
const double density_ball = 1;
const double density_fish = 20;
```

- 接下来的演算就可以完全交给 Box2D 实现了，物理引擎会忠实地为我们模拟碰撞的发生和碰撞后的弹射等物理变化，这些变化会在 update 定时器中同步至 Cocos2d-x 精灵，从而反映到屏幕上。

12.8 精确碰撞

- 12.8 精确碰撞

- 从调试绘图中可以注意到，用矩形做碰撞检测实际上是非常粗糙的，无法完全和鱼的外形吻合，能量球与鱼碰撞的效果会如同和一个砖头碰撞一样。尤其是灯笼鱼这种形状十分不规则的鱼，会导致与真实的碰撞效果差距很大。为了解决这个问题，我们需要更精确地定义鱼的碰撞边界。在 Box2D 中，可以使用一组顶点来创建多边形形状，用于描述这个边界，但是人工计算这些顶点比较麻烦。
- 我们可以使用工具 VertexHelper 完成多边形顶点的计算工作。和 TexturePacker 等工具一样，VertexHelper 是一个可视化顶点编辑器，用它创建的顶点信息可以直接导出为 Box2D 可使用的代码。唯一可惜的是，目前该软件仅支持苹果 OS X 系统，好在其使用的算法都不甚复杂，在 GitHub 上也能找到它的源代码，Windows 平台用户不妨考虑为其移植一个 Qt 版本。
- 工具的界面如图 12-2 所示，使用方法非常简单，在此不再赘述了。当我们创建好顶点之后，就可以在 Box2D 程序中使用了。然而 Box2D 对于多边形还有两个限制，具体如下所示。
- Box2D 规定了多边形的最大顶点数量为 8，但我们可以在"b2Settings.h"中修改最大顶点数目，把 b2_maxPolyVertices 改为所需的数目即可。
- 编辑所创建的多边形必须是凸多边形，否则物理引擎无法正确地演算。

图12-2 VertexHelper

- 当我们使用多边形替换原来的矩形来创建夹具后，编译并运行工程后，我们会看到物体的边界基本上和精灵的图形边界吻合了，这样就实现了更加真实的碰撞检测。

12.9 小结

- 12.9 小结

- 在这一章中，我们为了引入新的"能量球"超级武器，探索了物理引擎 Box2D 的基本用法，并且实现了一套精确的碰撞检测系统。下面我们总结一下这一章比较重要的知识点。
- Box2D 对象结构：Box2D 中包含形状（b2Shape）、夹具（b2Fixture）、物体（b2Body）与世界（b2World）等几个最重要的概念。形状规定了一个几何形状，可使用几何形状来创建夹具。夹具用来设置物体的几何形状与摩擦系数、密度等参数。物体表示一个物理实际中的宏观物体，可以相互作用发生运动、碰撞等现象，物体包含位置、旋转角度等信息，是物理引擎演算的单位。世界是全部物体的容器，物体需要添加到世界中才会产生物理现象。
- 创建物体：物体是一个拥有形状和密度等属性，并会参与物理现象的 Box2D 对象，因此，创建了一个物体后，需要利用形状创建夹具，把夹具绑定到物体上。此外，一个物体也可以绑定多个夹具，而绑定了多个夹具的物体具有多个形状合并后的属性。
- 尺度（PTM_RATIO）：在 Box2D 中，长度通常在 0.1~1 个单位长度之间，而游戏中的物体通常以像素为单位，为了把精灵与物体对应起来，我们定义了 PTM_RATIO 作为精灵长度单位到物理引擎长度单位的转换比例，它的单位是"像素/米"。
- 引入物理引擎：物理引擎与游戏引擎相互独立，因此我们需要为每一个精灵在物理引擎中创建物体对象，然后在游戏中不断更新物理引擎，并反过来把物理引擎的演算结果反映到游戏中。通常，这个过程可以在 Cocos2d-x 提供的 update 定时器事件中实现。
- 调试绘图：Cocos2d-x 的测试样例中包含一套用于在屏幕上显示物理引擎状态的工具，它们位于"tests/tests/Box2dTestBed"目录下的"GLES-Render.h（.cpp）"文件中。使用 Box2D 世界对象的 SetDebugDraw 方法，可以启用调试绘图。
- Cocos2d-x 和 Box2D 是两个配合十分完美的引擎，从绘图机制到接口，都能很好地融合。这里我们还只是使用了 Box2D 的碰撞检测功能，关节等众多特性尚未深入展开，但感兴趣的读者可以从文档中找到答案。

13.1 CCUserDefault

- **数据持久化**
  - 在前面打造的游戏中，每一次进入游戏都是一成不变的：玩家金币清零，没有记录经验值。在本章中，我们将添加一个保存玩家当前金币数的功能，此时就需要将数据以文件的形式存储到外存中。
  - 在实际开发中，需要存储的不仅仅是金币数目，还可能包括其他与玩家相关的信息。我们要讨论的方法对于记录金币数这个小小的功能来说显得过于复杂，但这将适用于所有需要存储的数据。下面将由浅入深地介绍几种数据持久化的方法。
  - **13.1 CCUserDefault**
  - CCUserDefault 是Cocos2d-x 引擎提供的持久化方案,其作用是存储所有游戏通用的用户配置信息,例如音乐和音效配置等。为了方便起见，有时我们也可以用 CCUserDefault 来存储金币数目这种简单的数据项。
  - CCUserDefault 可以看做一个永久存储的字典，本质是一个 XML 文件，将每个键及其对应的值以节点的形式存储到外存中。值只支持 int 和 float 等基本类型。使用接口非常简单，只需要一行代码：

```
CCUserDefault::sharedUserDefault()->setIntegerForKey("coin", coin - 1);
```

  - 由于每次设置和读取都会遍历整棵 XML 树，效率不高，且值类型具有局限性，因此 CCUserDefault 只适合小规模使用，对于复杂的持久化场景就会显得很无力。

13.2 格式化存储

- **13.2 格式化存储**
- 对于稍微复杂的持久化情景，还是可以借助 CCUserDefault 来满足我们的需求的。由于 CCUserDefault 是允许存储字符串值的，所以只要将需要保存的数据类型先转化为字符串，就可以写入外存中。

- 我们先将用户记录封装为类，由用户 ID 标识，在一个 ID 下存放金币、经验值和音乐 3 个值，这样游戏中就允许存在多个用户的记录了。我们创建了一个 UserRecord 类来读写用户记录，其定义如下：

```
class UserRecord : public CCObject
{
    CC_SYNTHESIZE_PASS_BY_REF(string, m_userID, UserID);
    CC_SYNTHESIZE_PASS_BY_REF(int, m_coin, Coin);
    CC_SYNTHESIZE_PASS_BY_REF(int, m_exp, Exp);
    CC_SYNTHESIZE_PASS_BY_REF(bool, m_isMusicOn, IsMusicOn);

public:
    UserRecord(const string& userID);
    void saveToCCUserDefault();
    void readFromCCUserDefault();
};
```

- 我们把需要存档的数据通过 sprintf 函数格式化成一个字符串，并把字符串保存到 CCUserDefault 之中。注意，这里我们做了一点小小的处理，存储的关键字除了用户 ID 之外，还添加了一个前缀"UserRecord"，这样可以保证，即使在存储时其他类型对象用了同样的用户 ID，也可以被区分开。具体代码如下：

```
void UserRecord::saveToCCUserDefault()
{
    char buff[100];
    sprintf(buff, "%d %d %d",
        this->getCoin(),
        this->getExp(),
        this->getIsMusicOn() ? 1 : 0
    );
    const char* key = ("UserRecord." + this->getUserID()).c_str();
    CCUserDefault::sharedUserDefault()->setStringForKey(key, buff);
}
```

- 有了写入存档的功能，我们还需要一个逆向的从存档读取的过程。读取过程与此过程刚好相反。我们从 CCUserDefault 来获取保存的字符串，再使用 sscanf 函数来得到每个数据的值，相关代码如下：

```
void UserRecord::readFromCCUserDefault()
{
    string buff = CCUserDefault::sharedUserDef;
    ault()->getStringForKey(("UserRecord." + this->getUserID()).c_str());

    int coin = 0;
    int experience = 0;
    int music = 0;

    sscanf(buff.c_str(), "%d %d %d", &coin, &experience, &music);
    this->setCoin(coin);
    this->setExp(experience);
    this->setIsMusicOn(music!=0);
```

```
}
```

这一写一读的过程可以称为序列化与反序列化，是立体的内存数据与一维的字符串间的相互转换。实际上，我们只完成了从数据到 CCUserDefault 的标准化存储间的转换，从标准化存储到实际存储在文件中的字符串间的转换是交由引擎封装完成的。

13.3 本地文件存储

13.3 本地文件存储

现在我们已经可以将复杂的数据类型存储到配置文件中了，但把全部数据集中在一个文件中显然不是一个明智的做法。如果将不同类别的数据（例如，NPC 的状态和玩家完成的成就）存储到不同的文件中，既可以提高效率，也方便我们查找。下面我们来看看如何实现它。

不同平台间的文件系统不尽相同，为了简化操作、方便开发，Cocos2d-x 引擎为我们提供了 CCFileUtil 类，用于实现获取路径和读取内容等功能，其中两个最重要的接口如下：

```
static unsigned char* getFileData(const char* pszFileName,
    const char* pszMode, unsigned long * pSize);//装载文件内容
static std::string getWriteablePath(); //获得可读写路径
```

借助这两个接口，我们可以获得一个路径，然后对文件进行相应的读写。文件读写在实际开发中应用得比较直接，一般是批量集中写入和读出，在此不再赘述。对于稍微灵活的场景，尤其是需要在大量数据中随机读写一小部分的时候，直接的文件存储由于缺少寻址支持，会变得非常麻烦。我们可以借助 XML 和 SQL 这两种方式，来更好地解决这个问题。

13.4 XML 与 JSON（1）

13.4 XML 与 JSON（1）

读者一定很好奇我们为什么将这两种数据格式放到这个地方来介绍。从上面的例子可以看到，我们并没有过多地涉及数据格式的细节，更多的是通过已有的库在内存中进行操作。同样，读者无须很熟悉 XML 或者 JSON，也能够掌握接下来的内容。

XML 和 JSON 都是当下流行的数据存储格式，它们的共同特点就是数据明文，十分易于阅读。XML 源自于 SGML，是一种标记性数据描述语言，而 JSON 则是一种轻量级数据交换格式，比 XML 更为简洁。鉴于 C++对 XML 的支持更为完善，Cocos2d-x 选择了 XML 作为主要的文件存储格式。

下面我们看看用户记录是如何存储到 XML 文件中的，相关代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<userDefaultRoot>
    <UserRecord.201208012221>41 -8589934601 1</UserRecord.201208012221>
</userDefaultRoot>
```

与直接的无格式存储相比，这样的文件虽然会耗费稍大的空间，但可读性更强，程序解析起来也更方便一些。

XML 文档的语法非常简洁。文档由节点组成，节点的定义是递归的，节点内可以是一个字符串，也可以是由一组<tag></tag>包围的若干节点，其中 tag 可以是任意符合命名规则的标识符。这样的递归嵌套结构非常灵活，特别适合以键值对形式存储的数据，比如数组和字典等。对于游戏开发中的大部分情景，XML 文档都可以游刃有余地处理它们。

随 Cocos2d-x 一起分发的还有一个处理 XML 的开源库 LibXML2，它用纯 C 语言的接口封装了对 XML 的创建、寻址、读和写等操作，极大地方便了开发。这里我们可以仿照 CCUserDefault 的做法，将对象存储到指定的 XML 文件中。

和 XML 语言的规范相对应，LibXML2 库同样十分简洁，只有两个核心的概念，如表 13-1 所示。

表 13-1 LibXML2

核心类类名	含义	涵盖功能
xmlDocPtr	指向 XML 文档的指针	XML 文档的创建、保存、文档基本信息存取、根节点存取等
xmlNodePrt	指向 XML 文档中一个节点的指针	节点内容存取、子节点的增、删、改等

- 下面我们开始以外部 XML 文件的方式存储 UserRecord 对象，并从中看到 XML 文档的操作和 LibXML 的具体用法。
- 在 UserRecord 类中，我们添加如下两个接口，分别负责将对象从 XML 文件中读出和写入：

```
void saveToXMLFile(const char* filename="default.xml");
void readFromXMLFile(const char* filename="default.xml");
```

- 在开始之前，我们可以进一步抽象出两个函数，完成对象和字符串间的序列化和反序列化，以便在 XML 的读写接口和 CCUserDefault 的读写接口间共享，相关代码如下：

```
void UserRecord::readFromString(const string& str)
{
    int coin = 0;
    int experience = 0;
    int music = 0;

    sscanf(str.c_str(), "%d %d %d", &coin, &experience, &music);
    this->setCoin(coin);
    this->setExp(experience);
    this->setIsMusicOn(music != 0);
}
void UserRecord::writeToString(string& str)
{
    char buff[100] = "";
    sprintf(buff,"%d %d %d",
        this->getCoin(),
        this->getExp(),
        this->getIsMusicOn() ? 1 : 0
    );
    str = buff;
}
```


13.4 XML 与 JSON（2）

- 完成了序列化与反序列化的功能后，通过 CCUserDefault 读写 UserRecord 的实现就十分简洁了。下面是相关的代码：

```
void UserRecord::readFromCCUserDefault()
{
    string key("UserRecord.");
    key += this->getUserID();

    string buff = CCUserDefault::sharedUserDefault()->getStringForKey(key.c_str());
```

```
    this->readFromString(buff);

    xmlFreeDoc(node->doc);
}
void UserRecord::saveToCCUserDefault()
{
    string buff;
    this->writeToString(buff);

    string key("UserRecord.");
    key += this->getUserID();

    CCUserDefault::sharedUserDefault()->setStringForKey(key.c_str(),buff);
    xmlFreeDoc(node->doc);
}
```

■ 有了对字符的序列化和反序列化，实际上我们只需要关心如何正确地在 XML 文档中读写键值对。我们暂且将对象都写到文档的根节点下，不考虑存储数组等复合数据结构的情景，尽管这些情景在操作上是类似的。首先，我们在一个指定的文档的根节点下找到一个键值，如果根节点下不存在指定的键值，将根据参数指定来创建，相关代码如下：

```
xmlNodePtr getXMLNodeForKey(const char* pKey, const char* filename,
    bool creatIfNotExists = true)
{
    xmlNodePtr curNode = NULL,rootNode = NULL;
    if (! pKey) {
        return NULL;
    }
    do {
        //得到根节点

       xmlDocPtr doc = getXMLDocument(filename);
        rootNode = xmlDocGetRootElement(doc);
        if (NULL == rootNode) {
            CCLOG("read root node error");
            break;
        }
        //在根节点下找到目标节点
        curNode = (rootNode)->xmlChildrenNode;
        while (NULL != curNode) {
            if (!xmlStrcmp(curNode->name, BAD_CAST pKey)){
                break;
            }
            curNodecurNode = curNode->next;
        }
        //如果没找到且需要创建，则创建该节点
        if(NULL == curNode && creatIfNotExists) {
```

```
            curNode = xmlNewNode(NULL, BAD_CAST pKey);
            xmlAddChild(rootNode, curNode);

        }
    } while (0);

    return curNode;
}
```

- 在上述代码中，我们首先根据文件名获得了对应的 XML 文档指针，然后通过 xmlDocGet-
- RootElement 函数获得了该文档的根节点 rootNode。一个节点的子节点是以链表形式存储的，通过 xmlChildrenNode 获得第一个子节点指针，再通过 next 函数迭代整个子节点列表。如果没有找到指定节点，且函数参数指定了必须创建对应键值的子节点，则函数会根据给定的键值 key 创建并添加到根节点中。
- 接下来，则是根据文件名获得 XML 文档指针的方法，相关代码如下：

```
bool createXMLFile(const char* filename, const char* rootNodeName = "root")
{
    bool bRet = false;
    xmlDocPtr doc = NULL;
    do {
        //创建 XML 文档
        doc = xmlNewDoc(BAD_CAST"1.0");
        if (doc == NULL) {
            CCLOG("can not create xml doc");
            break;
        }

        //创建根节点
        xmlNodePtr rootNode = xmlNewNode(NULL, BAD_CAST rootNodeName);
        if (rootNode == NULL) {
            CCLOG("can not create root node");
            break;
        }

        xmlDocSetRootElement(doc, rootNode);

        //保存文档
        xmlSaveFile(filename, doc);
        bRet = true;
    } while (0);

    //释放文档
    if (doc) {
        xmlFreeDoc(doc);
    }

```

```
    return bRet;
}
xmlDocPtr getXMLDocument(const char* filename)
{
    if(!isFileExists(filename) && !createXMLFile(filename)) {
        return NULL;
    }
    return xmlReadFile(filename, "utf-8", XML_PARSE_RECOVER);
}
bool isFileExists(const char *filename)
{
    FILE *fp = fopen(filename, "r");
    bool bRet = false;
    if (fp) {
        bRet = true;
        fclose(fp);
    }
    return bRet;
}
```

- 这 3 段代码分别做了 3 件事情：创建一个具有特定根节点的 XML 文档，获取一个特定文件名的 XML 文件，测试文件是否存在。
- 集成以上的代码，我们再次保存 UserRecord 对象，可以成功地将其存入一个指定的 XML 文档中，相关代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<userDefaultRoot>
    <UserRecord.201208012221>41 -8589934601 1</UserRecord.201208012221>
</userDefaultRoot>
```

13.5 加密与解密

13.5 加密与解密

- 细心的读者应该已经注意到了，XML 的一个很严重的问题是明文存储，存储在外部的数据一旦被截获，就将直接暴露在攻击者面前，小则篡改用户数据，大则泄露用户隐私信息。因此，对存储在文件中的信息加密不可忽视。
- 幸运的是，前面我们已经设计好了序列化和反序列化过程，只要在其中加入合适的加密和解密算法，即可保证我们的数据不会被轻易窃取。这里我们只使用一个简单的编码轮换来加密，相关代码如下：

```
void encode(string &str)
{
    for(int i = 0; i < str.length(); i++) {
        int ch = str[i];
        ch = 0xff & (((ch & (1 << 7)) >> 7) & (ch << 1));
        str[i] = ch;
    }
}
void decode(string &str)
{
```

```
    for(int i = 0; i < str.length(); i++) {
        int ch = str[i];
        ch = 0xff & (((ch & (1)) << 7) & (ch >> 1));
        str[i] = ch;
    }
}
```

- 得益于之前已经抽象的对字符串的序列化和反序列化，只要将加密和解密分别放在这两个函数的最后，就可以完成对 CCUserDefault 和 XML 文档的读、写及加密、解密。
- 经过加密后的 XML 文档，安全性大大增加了。加密后的 XML 文档如下：

```
<?xml version="1.0" encoding="utf-8"?>
<userDefaultRoot>
    <UserRecord.201208012221>hb@Zpjprrfh1`@b</UserRecord.201208012221>
</userDefaultRoot>
```

- 13.6 SQLite（1）
- 从性能上说，XML 方式的存储基本可以满足 1 MB 以下的存储要求。但在更复杂的情景中，我们可能需要存储多种不同的类，每个类也需要存储不同的对象，此时 XML 存储的速度就将成为瓶颈。即便分文件存储，管理起来也很麻烦，这个时候可以引入数据库来提升存储效率。
- 关系数据库是一种经典的数据库，其中的数据被组织成表的形式，具有相同形式的数据存放在同一张表中，表内每一行代表一个数据。在表的基础上，数据库为我们提供增、删、改、查等操作，这些操作通常采用 SQL（结构化查询语言）表达。这种格式化、集中的存储再加上结构化的操作语言带来一个非常大的好处：可以进行深度的优化，大大提升存储和操作的效率。
- SQLite 是移动设备上常用的一个嵌入式数据库，具有开源、轻量等特点，其源代码只有两个".c"文件和两个".h"文件，并且已经包括了充分的注释说明。相比 MySQL 或者 SQL Server 这样的专业级数据库，甚至是比起同样轻量级的 Access，SQLite 的部署都可谓非常简单，只要将这 4 个文件导入工程中即可，这使得编译之后的 SQLite 非常小。
- SQLite 将数据库的数据存储在磁盘的单一文件中，并通过简单的外部接口提供 SQL 支持。由于其设计之初即是针对小规模数据的操作，在查询优化、高并发读写等方面做了极简化的处理，可以保证不占用系统额外的资源，因此，在大多数的嵌入式开发中，会比专业数据库有更快速、高效的执行效率。
- SQLite 的核心接口函数只有一个，如下所示：

```
int sqlite3_exec(
    sqlite3*,                                   //一个已打开的数据库
    const char *sql,                            //将要执行的 SQL 语句
    int (*callback)(void*, int, char**, char**),  //回调函数
    void *,                                   //回调函数的第一个参数（用于传递自定义数据）
    char **errmsg                               //出错时返回的错误信息
);
```

- 这个函数在一个打开的数据库中为我们执行一条 SQL 语句，并通过回调函数处理结果，其参数的含义已经由注释给出。为了开发上的便利，我们还可以通过第四个参数指定一个任意类型的对象传递给回调函数。当此函数运行出错时，错误信息会以字符串形式输出在 errmsg 中。具体的用法我们将在下面详细介绍。
- 我们依然沿用 UserRecord 类作为例子，在其中添加 3 个接口函数，具体如下所示：

```
sqlite3*  prepareTableInDB(const char* table, const char* dbFilename);
void  saveToSQLite(const char* table = "UserRecord", const char* dbFilename = "sql.db");
void  readFromSQLite(const char* table = "UserRecord",const char* dbFilename = "sql.db");
```

- 首先，我们需要为一次读写操作准备数据库，相关代码如下：

```
sqlite3* UserRecord::prepareTableInDB(const char* table,const char *dbFilename)
{
    sqlite3 *pDB = NULL;
    char *errorMsg = NULL;
    if(SQLITE_OK != sqlite3_open(dbFilename, &pDB)) {
        CCLOG("open sql file failed!");
        return NULL;
    }

    string sql = "create table if not exists " + string(table) +
        "(id char(80) primary key,coin integer,experience integer)";

    sqlite3_exec(pDB, sql.c_str(), NULL, NULL, &errorMsg);
    if(errorMsg != NULL) {
        CCLOG("exec sql %s fail with msg: %s", sql.c_str(), errorMsg);
        sqlite3_close(pDB);
        return NULL;
    }
    return pDB;
}
```

- 这里我们完成两部分操作，首先用 sqlite3_open 打开数据库，如果数据库文件不存在，则会自动创建。打开成功后，如果目标表格不存在，则创建表格。这里我们执行了一句 SQL 语句，用了最基本的 sqlite3_exec 的方式，单纯地执行并查看是否成功，不涉及数据库操作后与游戏数据的交互。

- 准备完数据库之后，我们来尝试将数据从 SQLite 读取到内存数据中，相关代码如下：

```
void UserRecord::readFromSQLite(const char* table, const char *dbFilename)
{
    char sql[1024];
    sqlite3* pDB = prepareTableInDB(table, dbFilename);

    if(pDB != NULL) {
        int count = 0;
        char *errorMsg;

        sprintf(sql,"select * from %s where id = %s", table, this->getUserID().c_str());
        sqlite3_exec(pDB, sql, loadUserRecord, this, &errorMsg);

        if(errorMsg!=NULL) {
            CCLOG("exec sql %s fail with msg: %s", sql, errorMsg);
            sqlite3_close(pDB);
```

```
            return;
        }
    }

    sqlite3_close(pDB);
}
```

这里同样执行了一条 SQL 语句，将目标对象根据 ID 从数据库中读出，但不同的是，这里我们用到了下面这个回调函数并在其中将查询结果读取到 UserRecord 对象中：

```
int loadUserRecord(void* para,int n_column,char** column_value,char **column_name)
{
    UserRecord* record = (UserRecord*)para;
    int coin, experience;

    sscanf(column_value[1],"%d",&coin);
    sscanf(column_value[2],"%d",&experience);
    record->setCoin(coin);
    record->setExp(experience);

    return 0;
}
```


13.6 SQLite (2)

该回调函数用于处理 SQL 操作成功后返回的数据。返回的数据可能是一个字符串或整型量，也可能是数据表中的若干行数据，而每组数据都会调用回调函数一次，若查询操作得到了 N 行结果，则回调函数会被调用 N 次，每次传输一行待处理的结果。回调函数一共有 4 个参数，第一个参数是需要供回调函数使用的某段数据的指针，通常指向一个对象或一个数组，以便根据查询结果修改数据；第二个参数是操作结果返回的记录的列数；第三个参数是返回结果的数组，这些返回结果中的每一列都是一个字符串；第四个参数则是每一列的列名。对于一条特定的 SQL 语句来说，第二个和第四个参数通常是固定不变的。

在上面这个回调函数中，我们传入的是一个 UserRecord 类型的指针，因为我们要把查询结果存入这个 UserRecord 对象之中以便后续使用。查询请求已经限制了返回结果最多仅有一个，因此我们不需要额外的判断。只需要从返回的字符串中提取出金币数量和经验值，并把相应的数据填充到 UserRecord 对象中就可以了。

同样，我们可以编写将 UserRecord 写入 SQLite 数据库的接口函数，相关代码如下：

```
void UserRecord::saveToSQLite(const char* table, const char *dbFilename)
{
    char sql[1024];
    sqlite3* pDB = prepareTableInDB(table, dbFilename);

    if(pDB!=NULL) {
        int count = 0;
        char *errorMsg;

        sprintf(sql, "select count(*) from %s where id = %s",
```

```
            table, this->getUserID().c_str());
        sqlite3_exec(pDB, sql, loadRecordCount, &count, &errorMsg);

        if(errorMsg != NULL) {
            CCLOG("exec sql %s fail with msg: %s", sql, errorMsg);
            sqlite3_close(pDB);
            return;
        }
        if(count) {
            sprintf(sql, "update %s set coin = %d,experience=%d where id = %s",
                table, this->getCoin(), this->getExp(), this->getUserID().c_str());
        }
        else {
            sprintf(sql, "insert into %s values( %s,%d,%d)",
                table, this->getUserID().c_str(), this->getCoin(), this->getExp());
        }

        sqlite3_exec(pDB, sql, NULL, NULL, &errorMsg);
        if(errorMsg != NULL){
            CCLOG("exec sql %s fail with msg: %s", sql, errorMsg);
            sqlite3_close(pDB);
            return;
        }
    }

    sqlite3_close(pDB);
}

int loadRecordCount(void* para, int n_column, char** column_value, char** column_name)
{
    int *pCount=(int*)para;
    sscanf(column_value[0], "%d", pCount);
    return 0;
}
```

- 这个功能同样是由一个调用数据库接口的主调函数和一个处理返回结果的回调函数共同完成的。由于数据库中的更新和插入使用不同的命令，所以我们必须先查询数据库中是否存在同 ID 的对象，再决定是更新当前对象还是插入数据库中。
- 注意上面的每一个 SQLite 操作后，我们都检查了操作是否成功，在失败的情况下及时中止后面的操作。而在一切操作完成之后，不管操作是否成功，都必须关闭数据库，以保证对数据库的改变能够正确保存。 最后，我们可以尝试查看读写的效果。除了直接从数据库中读取特定的数据之外，还可以借助工具查看整个数据库的状态。SQLite Database Browser 就是一个可以方便地查看 SQLite 数据库的图形化工具，它是开源而且免费的。图 13-1 显示的就是将一个 UserRecord 对象写入数据库后数据库的状态。
- 尽管数据库中的文件已经被封装为数据库专用格式的文件，无法通过简单的文本工具查看其内容，但是如果通过合适的工具打开，SQLite 数据库和 XML 同样存在明文存放数据的问题。对于敏感的数据，同样需要通过加密来提高安全性，其做法

与 XML 类似，在此就不再赘述了。

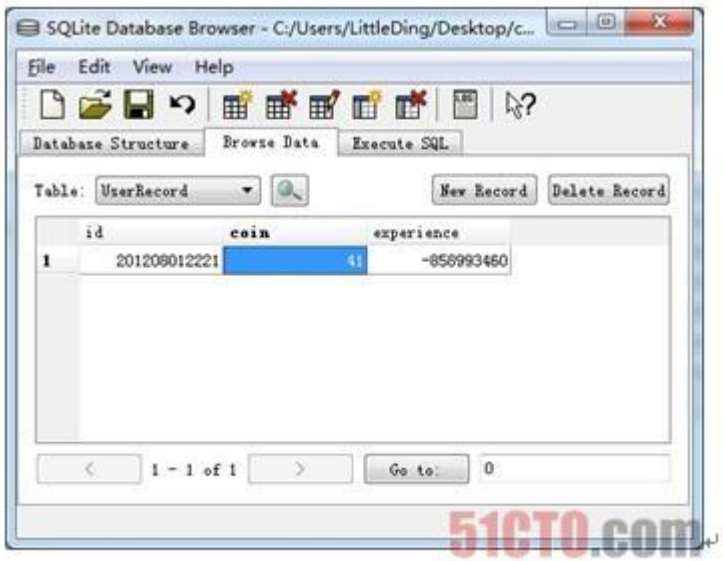


图 13-1 SQLite Database Browser

- 13.7 小结

13.7 小结

在这一章中，我们从一个小小的金币数入手，讨论了数据持久化的话题。我们尽量使用引擎提供的数据存储方法，以最大可能地适应跨平台需求。这里介绍的存储方法本质上都是基于 XML 的，对于 1 MB 以下的存储规模来说，已经完全足够了，而更大型的存储场景在目前的移动游戏中并不常见。在必要的时候，也可以使用 SQLite 来为数据持久化提速调优。 下面总结一下本章的重要知识点。

- CCUserDefault：它是 Cocos2d-x 提供的一个十分便捷的本地存储解决方案。利用 CCUserDefault 类，可以存取简单的键值数据。
- CCFileUtil：提供了对本地文件存取的基本功能。与 CCUserDefault 相比，CCFileUtil 更为底层，因此也适合用于存取更加庞大的二进制文件。
- XML：Cocos2d-x 中常见的文件类型，用于存储配置数据或游戏资料。XML 的优势在于描述性极强，因此易于编辑。
- JSON：网络传输中常用的对象描述格式，与 JavaScript 兼容，在广告平台等网络交换数据的情形中十分常见。
- SQLite：轻量级的关系数据库，用于高速且安全地在本地存储数据。在对性能要求较高时，可以考虑使用 SQLite 存储数据。

- 14.1 网络传输架构

网络

经过如此长时间的孤单捕鱼之后，现在我们的游戏终于要引入网络和多人联机的功能了。在移动互联网时代，移动设备与过去最大的不同就在于网络传输速度的大幅提升。此外，硬件的提升也给了游戏更大的想象空间。借助网络，我们可以为游戏添加更多有趣的特性。

14.1 网络传输架构

在游戏中选择网络传输方式时需要慎重。通常，有两种可能的方式供我们选择，具体如下所示。

- 直接使用 socket 传输。通信的两端都使用一个特定的端口传输数据，传输面向的是字节流或数据包，需要处理的细节比较多，包括建立与关闭连接、设计与实现网络协议、维护传输通道的稳定性、监控数据传输速率等。因此，很多情况下，我们需要在 socket 传输之上再根据游戏需求包装一层操作协议，以降低使用复杂度。
- 使用 HTTP 传输。直接用数据包的形式将数据提交到服务端的特定 URL 中，服务器同样用数据包的形式返回响应数据，其中大量的底层细节隐藏在了 HTTP 中。HTTP 作为非对等、主从式的传输，需要建立对应的服务器，幸运的是，Web 大潮催生了一批稳定、成熟的 HTTP 服务器框架，让我们可以方便地建立一个 HTTP 服务器。
- 还有一个值得考虑的因素是，移动设备的网络位置通常是不固定的，哪怕是在一个比较短的时间内，网络环境也有可能发生变动。如果直接在两台移动设备间建立连接进行通信，一方面难以确定一个固定的用于连接的地址，另一方面网络传输速率可能存在跳变。
- 对于非对战类游戏来说，在两个移动设备之间需要传输数据的情景并不多，少量的情景（如聊天等）建议通过服务器转发来完成。
- 综合考虑以上的因素，游戏中涉及网络部分的传输一般采用中心服务器的架构，服务器以 HTTP 服务形式建立 API 服务，各移动终端向中心服务器请求所需的 API 获得服务。

● 14.2 CURL

- 14.2 CURL
- CURL 是 Cocos2d-x 推荐使用的网络传输库，随引擎代码一起分发了 CURL 的一份 CPP 实现。它是免费开源的，而且支持 FTP、HTTP、LDAP 等多种传输方式，同时横跨了 Windows、UNIX、Linux 平台，可以在各种主流的移动设备上良好工作。
- 它存在两套核心的接口，分别对应两种不同的使用方式，具体如下所示。
- 单线程传输的阻塞方式：每次处理一个传输请求，会一直阻塞当前线程直到传输完成。
- 非阻塞方式：允许同时提交一批传输请求，CURL 会开启后台线程处理这些请求，传输结果的返回也是异步的。
- 在传输开始之前，这两种方式的 CURL 都要求我们先做全局的初始化，以及与之匹配的使用完毕之后的全局清理，这通常对应了程序的开始和结束。为保证正常的全局初始化和清理，我们使用单例模式对其简单封装如下：

```
class CURL_GLOBAL_INITIATOR
{
    CURL_GLOBAL_INITIATOR()
    {
        curl_global_init(CURL_GLOBAL_ALL);
    }

    static CURL_GLOBAL_INITIATOR curl_global_initiator;

public:
    ~CURL_GLOBAL_INITIATOR()
    {
        curl_global_cleanup();
    }
};
CURL_GLOBAL_INITIATOR CURL_GLOBAL_INITIATOR::curl_global_initiator;
```

- CURL 是纯 C 写成的网络库，其 API 全是函数形式，不涉及类，在实际使用中，我们不妨根据情况作适当的二次封装。

■ 14.3 简单传输

■ 简单传输就是阻塞的单线程传输，使用方式相对简单，其接口也都是前缀为 curl_easy_的形式，涉及 4 个常用 API，如下所示：

```
CURL *curl_easy_init(void);                                    //初始化一个传输
CURLcode curl_easy_setopt(CURL *curl, CURLoption option, ...);  //设置传输参数
CURLcode curl_easy_perform(CURL *curl);                        //执行当前传输
void curl_easy_cleanup(CURL *curl);                            //清理
```

■ 其中 curl_easy_setopt 函数的使用形式和一般的函数不太一样的是，没有对每一个参数配备独立的设置函数，而是通过定义不同的枚举常量传递参数设置，这和 OpenGL 中的 glEnable 系列函数非常类似，极大地简化了 API 接口。

■ 我们沿用键值对的形式传输数据，这样的好处依然是简单灵活，只需通过 POST 形式向服务器提交数据即可。因此，根据实际使用，我们将对其作适当的二次封装，相关代码如下：

```
class NetworkAdaptor{
    string m_sBaseUrl;
public:
    NetworkAdaptor(const string& baseUrl);
    NetworkAdaptor(const char* baseUrl);
    bool sendValueForKey(const char* key, const char* _value, string& writeBackBuff);
    bool sendValuesForKey(const map<string, string>& values, string& writeBackBuff);
};
```

■ 其中核心部分的 sendValuesForKey 函数向一个预设的 URL 传输一个字典中的所有键值对。只传输一对键值对的 sendValueForKey 可以在此基础上实现。sendValuesForKey 函数的实现代码如下所示：

```
bool NetworkAdaptor::sendValuesForKey(const map<string,string>& values,
    string& writeBackBuff)
{
    CURL *curl = curl_easy_init();

    string sendout;
    translate(values, sendout);
    curl_easy_setopt(curl, CURLOPT_URL, m_sBaseUrl.c_str());
    curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
    curl_easy_setopt(curl, CURLOPT_POST, 1);
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, sendout.c_str());
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, writer);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &writeBackBuff);

    int res = curl_easy_perform(curl);
    curl_easy_cleanup(curl);

    if (res == 0) {
        CCLOG("get data from  server : %s", writeBackBuff.c_str());
        return true;
```

```
    }
    else  {
        CCLOG("curl post error!");
        return false;
    }
}
```

在这个函数中，我们用 curl_easy_setopt 设置了 CURL 的几个关键参数：设置发送 POST 请求、设置上传的数据、设置回调函数，以及设置缓冲对象。最后便是执行和清理了。

translate 函数负责将传入的字典值按照 POST 的参数传递标准序列化为字符串，其实现代码如下：

```
void translate(const map<string, string>& values, string& sendoutMsg)
{
    sendoutMsg = "";
    for(map<string, string>::const_iterator it = values.begin();
        it != values.end(); ++it) {
        sendoutMsg += (it->first + " = " + it->second);
    }
}
```

而回调写函数则是根据 CURL 的回调标准编写，负责将 data 指向的 nmemb 个数据（每个数据的大小为 size 字节）写入 writeData 缓冲区内，并返回读取的总字节数。这里我们直接将传回的数据连接到一个缓冲字符串之后。而在实际开发中，这也是一个非常适合用于解码的地方，我们可以将传回的数据直接反序列化为需要操作的数据对象，相关代码如下：

```
size_t writer(char* data, size_t size, size_t nmemb, string* writerData)
{
    LOG_FUNCTION_LIFE;
    if(writerData == NULL)
        return 0;

    writerData->append(data, size * nmemb);
    return size * nmemb;
}
```

14.4 非阻塞传输

前面看到的整个网络传输过程是阻塞串行执行的，尽管设置了回调函数，但也只是为了应对间断到达的数据流，代码之间实际上不存在乱序执行的可能。在传输量稍大的情况下，例如初始化一些场景实时请求资源时，或是对游戏进行大规模升级时，阻塞主线程会导致画面停滞，这在实际开发中是绝对不允许的。另外，网络传输速度毕竟是有限的。即使网络繁忙时，系统内的大部分 CPU 和内存资源也都是空闲的。因此，我们需要引入非阻塞的网络传输。

CURL 是支持非阻塞传输的，而且还允许并行地进行多个网络请求，其接口主要是以 curl_multi 为前缀的系列的函数：

```
CURLM *curl_multi_init(void);                                    //初始化
CURLMcode curl_multi_add_handle(CURLM *multi_handle,
    CURL *curl_handle);                                          //添加一个传输请求
CURLMcode curl_multi_perform(CURLM *multi_handle, int *running_handles); //执行传输
CURLMcode curl_multi_cleanup(CURLM *multi_handle);               //清理
```

■ 其巧妙之处就在于，将非阻塞传输搭建在了阻塞传输的基础之上，使得接口并不比阻塞传输复杂。我们再次将其封装为适合传输键值对的形式：

```
typedef map<string, string> StringMap;
class AsynchronousNetworkAdaptor
{
protected:
    struct RequestInfo
    {
        RequestInfo(const StringMap& _v, const string& _u, string& _b)
            : values(_v), url(_u), buffer(_b) { }

        StringMap values;
        string url;
        string& buffer;
    };
    vector<RequestInfo> requests;

public:

   void sendValueForKeyToURL(const char* key, const char* _value,
        const string& url, string& writeBackBuff);
    void sendValuesForKeyToURL(const StringMap& values,
       const string& url, string& writeBackBuff);
    void flushSendRequest();

    CC_SYNTHESIZE_READONLY(int, m_iUnfinishedRequest, UnfinishedRequest);
};
```

■ 每个请求到达后，我们仅仅将其缓冲到一个数组中：

```
void AsynchronousNetworkAdaptor::sendValuesForKeyToURL(
    const StringMap& values, const string& url, string& writeBackBuff){
    RequestInfo info(values, url, writeBackBuff);
    requests.push_back(info);
}
```

■ 随后，在 flushSendRequest 函数内一次性地将所有的请求发出，相关代码如下：

```
void AsynchronousNetworkAdaptor::flushSendRequest()
{
    CURLM* backUrl = curl_multi_init();

    for(int i = 0; i < requests.size(); i++) {
        CURL *curl = curl_easy_init();

        curl_easy_setopt(curl, CURLOPT_URL, requests[i].url);
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
```

```
        curl_easy_setopt(curl, CURLOPT_POST,1);

        string sendout;
        translate(requests[i].values, sendout);
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS,sendout.c_str());
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, writer);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &requests[i].buffer);

        curl_multi_add_handle(backUrl, curl);
        curl_easy_cleanup(curl);
    }
    curl_multi_perform(backUrl, &m_iUnfinishedRequest);
}
```

- 注意，这里只是用一个 for 循环将若干个独立的阻塞式传输请求包装在了一起，并添加到之前创建的非阻塞传输处理器中，回写函数还是沿用阻塞方式下的函数。最后发出请求时，传入了一个整型变量，这个变量表示还在传输中的请求的数目，可以实时地反映当前的传输情况。
- 这样，我们就可以在游戏中保持绘图的流畅，同时在每帧更新中检查传输是否完成，完成后再触发相应的操作。

14.5 用户记录

14.5 用户记录

- 引入网络传输后可以完成的第一件事便是上传用户记录，这里的用户记录包括两方面的信息：一方面是用户对游戏的设置，诸如音效开关等；另一方面是用户的游戏进程信息，如用户的等级、金币数和成就等。这两方面信息本来是保存在本地设备的，我们将其上传到网络的服务器端，这带来的好处是用户可以随时在几台不同的设备上共用同一个捕鱼账户，自由地切换捕鱼的环境。
- 为了简化，我们依然沿用上一章的 UserRecord 作为例子：

```
void UserRecord::readFromServer(const char* url)
{
    NetworkAdaptor adaptor(url);
    string val;

    adaptor.sendValueForKey(this->makeKeyForWrite().c_str(), "", val);
    this->readFromString(val);
}

void UserRecord::saveToServer(const char* url)
{
    NetworkAdaptor adaptor(url);
    string val;

    this->writeToString(val);
    adaptor.sendValueForKey(this->makeKeyForWrite().c_str(), "", val);
}
```

- 这里我们可以很好地享用了上一章的成果：只需要封装字符串的传输，非字符串类型可以通过序列化、反序列化与字符串相互转换。

14.6 多人对战与同步问题

14.6 多人对战与同步问题

- 仅仅上传用户记录还是没有充分利用联网带来的好处，我们还需要引入联网的双人对战模式。
- 一旦涉及多人对战，就不得不考虑设备间的同步问题。多人对战时存在两台以上的可计算设备，考虑到线程等潜在的不确定性因素，这些设备关于游戏状态的计算结果很可能是不一致的，这时就需要互相之间进行同步。尤其是一些涉及胜负的因素，例如对鱼的击中判断，必须放在专用的判断服务器上进行，以保证游戏的公平性。
- 然而，将哪些计算交由服务器进行就需要抉择了，一是服务器的性能再强大，能支撑的计算也是有限的；二是必须尽量压缩数据的传输量，因为玩家的流量和服务器的带宽都是有限的。
- 在所有的计算中，涉及胜负的游戏逻辑判断是难以避免的，只能尽量压缩传输的数据来减少带宽消耗。而下面我们将看到的，则是网络模块中常见的两个小技巧。

14.6.1 时间同步

14.6.1 时间同步

- 首先必须同步的是时间，这里的时间并不等同于设备的时间，更确切地说是游戏时间，即游戏已经进行了多长时间。时间的重要性在于，游戏内发生的事件和服务器送达的同步信息，都需要依赖时间来最后同步到本地。这个时间通常在游戏一开始的时候由服务器发送，设备端从接收到服务器的指令开始计算，总的来说，服务器时间总要比设备端时间稍快一些。一个比较好的解决方案是，在游戏一开始的时候，服务器和设备端都向同一个时间服务器获取时间戳，同步双方的时间。这样在双方传输指令或事件时，可以根据标准时间给出时间戳，保证动作按序执行。
- 通用的时间同步协议是 NTP，全球有若干开放的服务器提供时间同步服务，可以向服务器获取当前的标准时间。协议相对复杂，但是在使用上是比较简单的，读者可以在各大代码仓库（例如 GitHub）中找到开源的实现，感兴趣的读者可以进一步研究。

14.6.2 鱼群同步

14.6.2 鱼群同步

- 在同一个游戏里，两个玩家所面对的鱼群必须在包括动作、位置等全部状态上都是一致的，于是我们就面临着鱼群的同步问题。对于捕鱼游戏而言，只要稳定鱼群的生成即可。由于每一条鱼的动作都是事先规划好的，所以同步问题就相当于解决了。但如果鱼群交由服务器负责生成，生成结果的传输对带宽压力还是不小的，尤其是服务器还需要同时向两台以上的设备分发生成的结果；如果遇到网络的短暂阻塞，屏幕将出现短时间的无鱼状态，稍后到达的生成信息也很可能造成本地游戏进程延迟的混乱局面。
- 回顾游戏中的鱼群生成，实际上是由随机数控制的，我们只需要一个可复用的伪随机数序列生成器，在不同的设备间同步这个生成器的随机数种子，就可以保证两位玩家看到同样的鱼群了，相关代码如下：

```
class Random
{
    int m_iSeed;

public:
    Random(const int& seed = 0) : m_iSeed(seed) { }

```

```
    int nextInt(int low = 0, int high = 65535)
    {
        m_iSeed = (m_iSeed * 7 + 11);
        return m_iSeed % (high - low + 1) + low;
    }

    float nextFloat(float low = 0.0, float high = 1.0)
    {
        int tmp = nextInt();
        return ((float)tmp/(float)65535);
    }
};
```

这其实是一个有代表性的例子。很多游戏都存在着这样的场景，无论是 Doodle Jump 的木板、《水果忍者》中的水果，还是模拟养成类游戏中的事件，都可以处理为由一个随机数发生器控制的随机事件，进而由一个随机数种子控制两台设备间的同步。

14.7 校验

14.7 校验

尽管我们通过随机数种子控制了鱼群的同步，但还是不可能将鱼群的计算完全交由两台不同的设备自主进行，因为长时间运行之后同样可能出现一些误差。这时需要以服务器的计算结果为准，重新校准两台设备的鱼群状态。

频繁地校准将导致带宽消耗上升，就失去了使用随机数控制的意义了，所以我们需要一个校验手段，以比较出两台设备的状态是否一致。

通常，我们会取一些重要的游戏数据来计算一个校验值，一起传输到服务器后进行校验即可。一定要保证选择的数据两边是同等的，尽量避开剧烈变动的临时数据。在《捕鱼达人》中，鱼群的数量和每个鱼群的位置就是很好的校验数据。对于这个校验值的计算，我们可以采用一些轻量级的算法，确保其独特性就足够了，不必采用 MD5 等重型的校验值计算方法。这里我们只对鱼群使用异或校验，生成一个 32 位的校验值，相关代码如下：

```
int hashForFishes(CCArray* fishes)
{
    int ret = fishes->count();
    for(int i = 0; i < fishes->count(); i++) {
        CCSprite* sp = dynamic_cast<CCSprite*>(fishes->objectAtIndex(i));
        CCPoint pos = sp->getPosition();

        ret ^= (int)pos.x;
        ret ^= (int)pos.y;
    }
    return ret;
}

```

14.8 小结

14.8 小结

- 在这一章中，我们探讨了网络相关的几个话题，从直接的数据传输到多人游戏中的多种同步问题。可以看到，涉及网络的游戏设计更追求异步、并发的编程思想。下面总结本章的重要知识点。
- socket、HTTP：socket（套接字）是操作系统提供的网络基础设施，用于计算机在网络间建立数据连接；HTTP 是运行于 TCP/IP 应用层的一套文本传输协议，常用于网页传输。在游戏开发中，通常利用 socket 产生长连接来维持游戏的同步，然而现在也有许多游戏采用 HTTP。
- CURL：CURL 是一套 URL 通信库，提供了 HTTP、FTP 等协议的支持，因此可以利用 CURL 方便地建立 HTTP 连接。
- 阻塞、非阻塞：网络传输是一个耗时的任务，因此运行时会阻碍其他代码的执行，直接在主线程中执行网络传输任务称作阻塞式传输，而在新线程中异步地进行网络传输任务则称为非阻塞式传输。
- 同步：网络游戏需要保持用户终端与服务器的数据一致，这个过程称为同步。有许多方案可以解决同步问题。
- 本章我们选择了一种分布式的游戏逻辑计算框架，配合校验机制，可以有效降低带宽消耗。更深入的话题，如游戏大厅的设计和服务器端的负载均衡等，已经超出了本书的讨论范围，感兴趣的读者不妨查阅相关资料进一步学习。

● 15.1 移动设备昂贵的 CPU 与内存

- **缓存与池**
- 游戏是十分消耗资源的，为此我们必须在开发过程中进行优化。第 10 章我们针对图形性能介绍了一系列优化方法，使游戏性能提高了一个层次，本章则帮助我们有效利用 CPU 与内存资源。
- **15.1 移动设备昂贵的 CPU 与内存**
- 诚然，现在移动设备的发展日新月异，硬件水平正在不断提高，但有两个因素同样是不可忽视的。
  - 如果不想在发行阶段就直接抛弃一半用户，那么任何一款移动终端游戏都必须照顾尚未退出市场的老机型，尽管这些机型很可能发布于两年前，性能较差。
  - 即使在新机型上，硬件资源也并不是特别乐观。还记得比尔盖茨那句"个人电脑不需要超过 64 KB 的内存"吗？其实不管有多好的硬件，总会被程序用尽。
- 表 15-1 展示了一些移动设备的 CPU 和内存配置，但这并不意味着我们可以使用所有的硬件资源。在高端机型上，应用程序可用内存可以超过 90 MB，而低端机型上则可能只有 30 MB~50 MB，CPU 能提供的计算能力在低端机型上更是有限。
- 表 15-1 部分设备的 CPU 和内存配置

● 型号	● CPU（MHz）	● 内存（MB）
◆ iPhone 5	● 2 核心 1024	● 1024
◆ iPod touch 5	● 2 核心 800	● 512
◆ 小米 2 代	● 4 核心 1536	● 2048
◆ 三星 Galaxy S III	● 4 核心 1433	● 1024
◆ 诺基亚 N9	● 1024	● 1024
◆ 华为 C8812	● 2 核心 1024	● 512
◆ 里程碑一代	● 550	● 256

- 在第 10 章中，我们已经通过合并纹理等手段压缩了内存和 GPU 调用，在这一章中，我们将介绍如何利用缓存和对象池机制进一步优化内存使用，降低 CPU 消耗。

● 15.2 缓存机制：预加载与重复使用

- **15.2 缓存机制：预加载与重复使用**

- 缓存在软硬件设计中是一个十分常见的优化方法，多用于高性能软硬件的设计。简单地说，缓存就是利用存储器的速度等级差异，将低速存储中使用频率高的内容加载到高速存储中，这样可以有效提高访问速度。比如将常用的图片资源从磁盘读到内存，将常用的程序段从内存搬到 CPU 的高速缓存中。
- 移动设备比较常用的是外存到内存的缓存。尽管大部分手机使用闪存作为外部存储已经比 PC 上磁盘的速度快得多，但相比内存而言还是差了两到三个数量级。从闪存内读取一张图片平均会耗费接近 0.1 秒的时间，这样反复加载将占用非常多的系统资源。而缓存机制可以预先加载我们需要的内容到内存，并且在之后的操作中重复使用。
- 缓存空间毕竟有限，每一个缓存都应该实现合理的换入换出机制来保证缓存中的内容确实是最需要被反复利用的。同时，缓存应该尽量透明化，也就是说，在不主动调用的情况下缓存就应该生效。

- 15.3.1 CCTextureCache

- 15.3 Cocos2d-x 中的缓存
- 幸运的是，我们不需要自己实现缓存，因为 Cocos2d-x 已经为我们提供了足够强大的实现。引擎中存在 3 个缓存类，都是全局单例模式。

- 15.3.1 CCTextureCache
- 首先是最底层也最有效的纹理缓存 CCTextureCache，这里缓存的是加载到内存中的纹理资源，也就是图片资源。其原理是对加入缓存的纹理资源进行一次引用，使其引用计数加一，保持不被清除，而 Cocos2d-x 的渲染机制是可以重复使用同一份纹理在不同的场合进行绘制，从而达到重复使用，降低内存和 GPU 运算资源的开销的目的。常用的是如下所示的 3 个
- 接口：

```
static CCTextureCache* sharedTextureCache();    //返回纹理缓存的全局单例
CCTexture2D* addImage(const char* fileimage);  //添加一张纹理图片到缓存中
void removeUnusedTextures();                    //清除不使用的纹理
```

- 在这 3 个接口中，CCTextureCache 屏蔽了加载纹理的许多细节；addImage 函数会返回一个纹理 CCTexture2D 的引用，可能是新加载到内存的，也可能是之前已经存在的；而 removeUnusedTextures 则会释放当前所有引用计数为 1 的纹理，即目前没有被使用的纹理。后面会看到，引用计数的内存管理方式为缓存的设计带来了很大的便利。
- 实际上，我们很少需要调用 addImage 这个接口，因为引擎内部所有的纹理加载都是通过这个缓存进行的，换句话说，载入的每一张图片都被缓存了，所以我们更需要关心什么时候清理缓存。引擎会在设备出现内存警告时自动清理缓存，但是这显然在很多情况下已经为时过晚了。一般情况下，我们应该在切换场景时清理缓存中的无用纹理，因为不同场景间使用的纹理是不同的。如果确实存在着共享的纹理，将其加入一个标记数组来保持其引用计数，以避免被清理了。

- 15.3.2 CCSpriteFrameCache

- 15.3.2 CCSpriteFrameCache
- 第二个则是精灵框帧缓存。顾名思义，这里缓存的是精灵框帧 CCSpriteFrame，它主要服务于多张碎图合并出来的纹理图片。这种纹理在一张大图中包含了多张小图，直接通过 CCTextureCache 引用会有诸多不便，因而衍生出来精灵框帧的处理方式，即把截取好的纹理信息保存在一个精灵框帧内，精灵通过切换不同的框帧来显示出不同的图案。
- CCSpriteFrameCache 的常用接口和 CCTextureCache 类似，不再赘述了，唯一需要注意的是添加精灵帧的配套文件-- 一个 plist 文件和一张大的纹理图。下面列举了 CCSpriteFrame Cache 常用的方法：

```
static CCSpriteFrameCache* sharedSpriteFrameCache(void);  //全局共享的缓存单例
void addSpriteFramesWithFile(const char *pszPlist);  //通过 plist 配置文件添加一组精灵帧
void removeUnusedSpriteFrames(void);  //清理无用缓存
```

- 15.3.3 CCAnimationCache

■ 15.3.3 CCAnimationCache

■ 最后一个是 CCAnimationCache 动画的缓存。通常情况下，对于一个精灵动画，每次创建时都需要加载精灵帧，按顺序添加到数组，再创建对应动作类，这是一个非常烦琐的计算过程。对于使用频率高的动画，比如鱼的游动，将其加入缓存可以有效降低每次创建的巨大消耗。由于这个类的目的和缓存内容都非常简单直接，所以其接口也是最简单明了的，如下所示：

```
static CCAnimationCache* sharedAnimationCache(void);//全局共享的缓存单例
void addAnimation(CCAnimation *animation, const char * name);//添加一个动画到缓存
void removeAnimationByName(const char* name);//移除一个指定的动画
CCAnimation* animationByName(const char* name);//获得事先存入的动画
```

■ 唯一不一样的是，这次动画缓存需要我们手动维护全部动画信息。也因为加载帧动画完全是代码操作的，目前还没有配置文件指导，所以不能像另外两个缓存那样透明化。实际上，如果考虑到两个场景间使用的动画基本不会重复，可以直接清理整个动画缓存。

■ 所以，在场景切换时我们应该加入如下的清理缓存操作：

```
void releaseCaches()
{
    CCAnimationCache::purgeSharedAnimationCache();
    CCSpriteFrameCache::sharedSpriteFrameCache()->removeUnusedSpriteFrames();
    CCTextureCache::sharedTextureCache()->removeUnusedTextures();
}
```

■ 值得注意的是清理的顺序，应该先清理动画缓存，然后清理精灵帧，最后是纹理。按照引用层级由高到低，以保证释放引用有效。

- 15.4 对象池机制：可回收与重复使用

■ 15.4 对象池机制：可回收与重复使用

■ 另一个能有效提高内存和计算效率的是对象池机制。其本质与缓存类似，即希望能减少那些频繁使用的对象的重复创建和销毁，例如飞行射击类游戏中的子弹。使用对象池机制能带来两方面的收益，首先是减少对象初始化阶段的重复计算，其次是避免反复地向操作系统申请归还内存。一个很好的例子就是捕鱼游戏中的鱼，鱼和鱼之间的属性是类似的，不一样的仅仅是当前的坐标位置及正在播放的动画帧。那么，当鱼游出屏幕后，可以不对其进行销毁，而是暂存起来。某一时刻需要重新创建鱼时，我们可以将其从对象池中取出，重新申请内存并初始化，这样就大大减轻了 CPU 的负担。

■ 对象池和缓存很像，但比缓存更抽象，也更简单一些，因为我们不需要考虑从哪里加载的问题：都已经被抽象为初始化函数了。而且更简化的是，加入对象池的每一个对象都是无差别的，我们不需要对每一个对象进行特定的标记，直接取出任意一个未使用的对象即可。

■ 看完上面的描述，读者应该有了初步的认识：缓存是一个字典，而对象池则是一个数组。得益于引用计数的内存管理机制，只需要在数组上做适当封装就可以提供一个对象池的功能了。尽管如此，一个高效实现的对象池还要考虑如何有效地处理对象的生成和归还，以及占用内存的动态增长等问题。因此，我们不妨借助前人成果，在已有对象池的基础上搭建适合我们游戏使用的对象池。

- 15.5 对象池实现（1）

■ 15.5 对象池实现（1）

- Boost 是一个可移植、免费开源的 C++库，提供了大量实用的开发组件，而且由于对跨平台和 C++标准的强调，其实现的功能几乎不依赖于操作系统和标准库外的其他组件，因此可以在任何支持 C++的平台上运作良好。
- Boost 提供了一个对象池 object_pool，它位于 boost 库的"boost/pool/object/_pool.hpp"中。这是一个泛型的对象池，能够针对指定类型的对象进行分配。一个对象池的声明和使用规范为如下结构：

```
object_pool<CCSprite> spritePool;    //为 CCSprite 声明一个对象池
CCSprite* sp =spritePool.construct();    //从对象池得到一个对象，并调用默认构造函数
spritePool.destroy(sp); //对从对象池得到的对象调用析构函数，并返还到对象池中备用
```

- object_pool 的一大特色是可以针对不同的参数调用被分配对象的构造函数。可惜在 Cocos2d-x 对象生命周期管理中，对象的创建和初始化是分离的，大部分类的初始化都不在构造函数中完成，构造函数中仅仅作引用计数的初始化。这里也引入了一个新的问题,Cocos2d-x 对象在引用计数为零的时候会自动触发 delete。对于从对象池分配的对象来说,不能通过 delete 而必须通过 destroy 来删除。因此，在不修改引擎源码的前提下，我们需要在 object_pool 的基础上作一点小小的包装使其可以配合引擎的内存管理使用，相关代码如下：

```
template<class T>
class MTPoolFromBoost : public ObjectPoolProtocol
{
    object_pool<T> pool;
    CCArray* objects;
    MTPoolFromBoost() : pool(256){
        objects = CCArray::create();
        objects->retain();
    }
public:
    ~MTPoolFromBoost()
    {
        objects->removeAllObjects();
        objects->release();
    }
    static MTPoolFromBoost<T>* sharedPool()
    {
        static MTPoolFromBoost<T> __sharedPool;
        return &__sharedPool;
    }
    T* getObject()
    {
        T* pObj = pool.construct();
        objects->addObject(pObj);
        pObj->release();
        return pObj;
    }
    void freeObjects(int maxScan = 100)
    {
        static int lastIndex =0;

        int count = objects->count();
```

```
        if(lastIndex >= count)
            lastIndex = 0;
        if(maxScan > count) maxScan = count;

        CCArray* toRemove = CCArray::create();
        for(int i = 0; i < maxScan; i++) {
            CCObject* obj = objects->objectAtIndex((i + lastIndex) % count);
            if(obj->retainCount() == 1) {
                toRemove->addObject(obj);
            }
        }

        objects->removeObjectsInArray(toRemove);
        for(int i=0; i < toRemove->count(); i++) {
            T* obj = dynamic_cast<T*>(toRemove->lastObject());
            obj->retain();
            toRemove->removeLastObject();
            pool.destroy(obj);
        }
        CCLOG("%s ends. Obj now = %d", __FUNCTION__, objects->count());
    }
};
```

- 由于做成了模板类的形式，类的实现就全部存在于头文件中了。在这个包装类中，我们仅仅做了一件事情--在分配对象的时候，同时将对象添加到一个数组中，数组会增加对象的一次引用计数，因此可以保证在正常使用的情况下，不会有对象会被触发 delete 操作。由此引出的便是，需要在合适的时候回收对象，否则对象池将持续增长直到耗尽内存。
- 在提供的内存释放函数 freeObjects 中，我们检查当前缓冲数组中每个元素的引用计数，对于引用计数为 1 的对象，表示已经没有其他对象在引用这个对象，将其回收归还到对象池中。值得注意的是，在释放对象的循环中，我们将一个待回收的对象 retain 后并没有 release，这是对引用计数内存管理的一个小小破例，保证了该对象在从数组清理之后仍然不会触发 delete 操作。
- 另外，这里设计了一个回收的扫描步长，每次回收仅在数组中扫描一定数量的对象就返回。这样做的好处在于，我们可以将整个对象池的回收扫描分散到每一帧中，隐性地完成并发。这个步长可以根据工程规模和所需的清理频率进行调整，对于游戏中对象生成和销毁并不频繁的情况，可以设置一个较长的清理周期，在每次清理时设置一个较大的扫描步长以回收更多的对象，同时减轻计算压力。
- 模板化之后，实际上每个类对应了一个对象池，以硬编码的形式清理这些对象池是十分费劲的，因此我们再在此基础上扩展一个管理器，管理这些对象池的清理。

15.5 对象池实现（2）

15.5 对象池实现（2）

- 首先，需要做的是将回收操作分离抽象。我们定义一个接口并让 MTPoolFromBoost 继承，这样就能够在运行时用统一接口调用内存池回收对象：

```
class ObjectPoolProtocol : public CCObject{
public:
    virtual void freeObjects(int maxScan = 100) = 0;
```

```
};
```

■ 这样抽象的目的是将所有用到的对象池添加到数组内，以便统一管理。首先，为管理器封装一个获取对象指针的函数：

```
class MTPoolManager : public CCObject{
    CCArray* pools;
    class PoolCounter {
    public:
        PoolCounter(ObjectPoolProtocol* pool,CCArray* pools)
        {
            pools->addObject(pool);
        }
    };

    MTPoolManager()
    {
        pools = CCArray::create();
        pools->retain();
    }
    ~MTPoolManager()
    {
        pools->release();
    }

public:
    static MTPoolManager* sharedManager()
    {
        static MTPoolManager __sharedManager;

       return &__sharedManager;
    }
    void freeObjects(ccTime dt)
    {
        for(int i = 0; i < pools->count(); i++) {
            ObjectPoolProtocol* pool = dynamic_cast<ObjectPoolProtocol*>(pools->
                objectAtIndex(i));
            pool->freeObjects();
        }
    }

    template<class T>
    T* getObject(T*& pObj)
    {
        static PoolCounter ___poolCounter(MTPoolFromBoost<T>::sharedPool(), pools);
        return pObj = MTPoolFromBoost<T>::sharedPool()->getObject();
    }
```

```
};
```

在管理器中我们设计了一个获取对象的接口函数 getObject，可以根据传入的指针类型调用相应类型的对象池获得对象。这里我们设置一个静态变量，使用这个变量的构造函数添加当前对象池到类的对象池数组中。由于这个函数是模板化的，最终将把每种调用到的对象池添加到管理器的对象池数组中。这样设计的另一个好处是，管理器调用某一类型的对象池之前，不会在管理器的清理函数中触发该对象池的清理。

而在管理器的清理函数中，可以获取每一个曾经使用过的管理器，调用其清理接口清理对象。 最后，我们只需要在程序初始化完毕后添加该管理器到引擎的定时触发器中：

```
CCDirector::sharedDirector()->getScheduler()->scheduleSelector(
    schedule_selector(MTPoolManager::freeObjects),
    MTPoolManager::sharedManager(),
    1,
    false
);
```

15.6 落实到工厂方法

15.6 落实到工厂方法

经过上面的封装，显式地使用对象池已经很方便了，但我们还可以做得更好。借助于工厂方法，我们可以将对象池隐藏起来，透明化其使用：

```
FishSprite* FishSprite::spriteWithSpriteFrameName(const char* file)
{
    FishSprite *pRet = MTPoolManager::sharedManager()->getObject(pRet);
    if(pRet && !pRet->initWithSpriteFrameName(file)) {
        CC_SAFE_RELEASE_NULL(pRet);

   }
    return pRet;
}
```

这样做的好处是，游戏中的逻辑代码是没有改动的，直接无缝引入了对象池增强内存管理。

15.7 一个简单的性能测试

15.7 一个简单的性能测试

对象池的效率如何？我们来做一个简单的性能测试，示例代码如下：

```
object_pool<CCSprite> spritePool;
void testPoolEfficiency()
{
    const int test_time = 101000;
    vector<int> ops;
    int tot = 0;
    for(int i = 0; i < test_time; i++) {
        int op = 0;
        if(tot) {
            if(rand() % 2 == 0)
```

```
                op=1;
            }

            if(op) tot--;
            else tot++;

            ops.push_back(op);
        }

        {
            LifeCircleLogger logger("use system new");
            CCArray* tmp = CCArray::create();
            for(int i = 0; i < test_time; i++) {
                if(ops[i]) {
                    tmp->removeLastObject();
                }
                else {
                    CCSprite* sp = CCSprite::create();
                    tmp->addObject(sp);
                }
            }
        }
        {
            LifeCircleLogger logger("use MTPool");
            CCArray* tmp = CCArray::create();
            for(int i = 0; i < test_time; i++) {
                if(ops[i]) {
                    tmp->removeLastObject();
                }
                else {
                    CCSprite* sp = MTPoolManager::sharedManager()->getObject(sp);
                    tmp->addObject(sp);
                }
            }
        }

        {
            LifeCircleLogger logger("use boost Pool");
            CCArray* tmp = CCArray::create();
            for(int i = 0; i < test_time; i++) {
                if(ops[i])
                {
                    tmp->removeLastObject();
                }
                else {
```

```
                CCSprite* sp = spritePool.construct();
                tmp->addObject(sp);
            }
        }
    }
}
```

在这个性能测试中，我们分别使用系统 new、编写的对象池和 boost 的对象池作 10 000 次随机的分配或回收操作。连续执行上面的函数 3 次，得到的执行时间如下：

```
[Wed Aug 08 16:31:12 2012] use system new BEGINS!
[Wed Aug 08 16:31:13 2012] use system new ENDS! Time Consumed : 100 ms
[Wed Aug 08 16:31:13 2012] use MTPool BEGINS!
[Wed Aug 08 16:31:13 2012] use MTPool ENDS! Time Consumed : 14 ms
[Wed Aug 08 16:31:13 2012] use boost Pool BEGINS!
[Wed Aug 08 16:31:13 2012] use boost Pool ENDS! Time Consumed : 12 ms
[Wed Aug 08 16:31:18 2012] use system new BEGINS!
[Wed Aug 08 16:31:18 2012] use system new ENDS! Time Consumed : 141 ms
[Wed Aug 08 16:31:18 2012] use MTPool BEGINS!
[Wed Aug 08 16:31:18 2012] use MTPool ENDS! Time Consumed : 14 ms
[Wed Aug 08 16:31:18 2012] use boost Pool BEGINS!
[Wed Aug 08 16:31:18 2012] use boost Pool ENDS! Time Consumed : 12 ms
```

可以看到，使用对象池后，分配对象的耗时降低到了单纯使用 new 的 1/10 左右，这个提升是非常可观的。

15.8 使用时机

15.8 使用时机

最后需要强调的是，缓存和对象池都不是万金油，我们需要把握好它们的使用时机。缓存和对象池的使用动机都是为频繁使用的资源作优化（这里的资源可以是外存的纹理，也可以是一个对象），避免大量的重复计算。缓存和对象池内都做了一些额外的小计算量的标记来满足这一需求。对于游戏中那些使用频率并不高的部分，加入缓存或者对象池反而很可能因为额外的标记计算而降低性能。这也是引擎只为我们提供了 3 个缓存器的原因。

值得一提的是，缓存和对象池不仅仅适用于 C++这类偏底层的开发语言，在 C#和 JavaScript 等语言中，内存的开销更大，使用好缓存和对象池能有效减少不必要的系统内存管理，提升游戏执行效率。

15.9 小结

15.9 小结

在这一章中，我们探讨了缓存和对象池这两个隐蔽而实用的组件，也在我们的《捕鱼达人》中加入了相应的优化，进一步降低了内存和计算消耗。在 Cocos2d-x 中，缓存无处不在，然而在实际开发时，我们仍然需要合理地利用缓存与对象池这两个组件，才能有效地提高游戏的性能。下面总结这一章的知识点。

Cocos2d-x 内建的缓存：CCTextureCache 用于缓存纹理，CCSpriteFrameCache 用于缓存精灵框帧文件，CCAnimationCache 用于缓存帧动画。

对象池：对于频繁使用的对象，应当尽可能地减少分配内存和初始化的资源开销。因此，建立一个对象的容器，用于取出或返还对象，这种机制称为对象池。对于大量使用对象的场合，对象池机制可以大大提高运行效率。

16.1 单线程的尴尬

- **并发编程**

  - 并发编程是利用线程实现的一系列技术，广泛用于执行耗时的任务。利用多线程技术，可以使游戏显示载入页面的同时在后台加载数据，也可以使游戏在运行的同时在后台进行下载任务等。并发编程的优势巨大，使用起来也并不困难，在这一章中，我们会详细介绍并发编程的方法。

  - **16.1 单线程的尴尬**

  - 重新回顾下 Cocos2d-x 的并行机制。引擎内部实现了一个庞大的主循环，在每帧之间更新各个精灵的状态、执行动作、调用定时函数等，这些操作之间可以保证严格独立，互不干扰。不得不说，这是一个非常巧妙的机制，它用一个线程就实现了并发，尤其是将连续的动作变化切割为离散的状态更新时，利用帧间间隔刷新这些状态即实现了多个动作的模拟。

  - 但这在本质上毕竟是一个串行的过程，一种尴尬的场景是，我们需要执行一个大的计算任务，两帧之间几十毫秒的时间根本不可能完成，例如加载几十张图片到内存中，这时候引擎提供的 schedule 并行就显得无力了：一次只能执行一个小时间片，我们要么将任务进一步细分为一个个更小的任务，要么只能眼睁睁地看着屏幕上的帧率往下掉，因为这个庞大计算消耗了太多时间，阻塞了主循环的正常运行。

  - 本来这个问题是难以避免的，但是随着移动设备硬件性能的提高，双核甚至四核的机器已经越来越普遍了，如果再不通过多线程挖掘硬件潜力就过于浪费了。

- **16.2 pthread**

  - **16.2 pthread**

  - pthread 是一套 POSIX 标准线程库，可以运行在各个平台上，包括 Android、iOS 和 Windows，也是 Cocos2d-x 官方推荐的多线程库。它使用 C 语言开发，提供非常友好也足够简洁的开发接口。

  - 一个线程的创建通常是这样的:

```
void* justAnotherTest(void *arg)
{
    LOG_FUNCTION_LIFE;
    //在这里写入新线程将要执行的代码
    return NULL;
}
void testThread()
{
    LOG_FUNCTION_LIFE;
    pthread_t tid;
    pthread_create(&tid, NULL, &justAnotherTest, NULL);
}
```

  - 这里我们在 testThread 函数中用 pthread_create 创建了一个线程,新线程的入口为 justAnotherTest 函数。pthread_create 函数的代码如下所示：

```
PTW32_DLLPORT int PTW32_CDECL pthread_create(
    pthread_t * tid,
    const pthread_attr_t * attr,
    void *(*start) (void *),
    void *arg);
```

- pthread_create 是创建新线程的方法，它的第一个参数指定一个标识的地址，用于返回创建的线程标识；第二个参数是创建线程的参数，在不需要设置任何参数的情况下，只需传入 NULL 即可；第三个参数则是线程入口函数的指针，被指定为 void*(void*)的形式。函数指针接受的唯一参数来源于调用 pthread_create 函数时所传入的第四个参数，可以用于传递用户数据。

16.3 线程安全

16.3 线程安全

- 使用线程就不得不提线程安全问题。线程安全问题来源于不同线程的执行顺序是不可预测的，线程调度都视系统当时的状态而定，尤其是直接或间接的全局共享变量。如果不同线程间都存在着读写访问，就很可能出现运行结果不可控的问题。
- 在 Cocos2d-x 中，最大的线程安全隐患是内存管理。引擎明确声明了 retain、release 和 autorelease 三个方法都不是线程安全的。如果在不同的线程间对同一个对象作内存管理，可能会出现严重的内存泄露或野指针问题。比如说，如果我们按照下述代码加载图片资源，就很可能出现找不到图片的报错：

```
void* loadResources(void *arg)
{
    LOG_FUNCTION_LIFE;
    CCTextureCache::sharedTextureCache()->addImage("fish.png");
    return NULL;
}

void makeAFish()
{
    LOG_FUNCTION_LIFE;
    pthread_t tid;
    pthread_create(&tid, NULL, &loadResources, NULL);
    CCSprite* sp = CCSprite::create("fish.png");
}
```

- 在第新的线程中对缓存的调用所产生的一系列内存管理操作更可能导致系统崩溃。
- 因此，使用多线程的首要原则是，在新建立的线程中不要使用任何 Cocos2d-x 内建的内存管理，也不要调用任何引擎提供的函数或方法，因为那可能会导致 Cocos2d-x 内存管理错误。
- 同样，OpenGL 的各个接口函数也不是线程安全的。也就是说，一切和绘图直接相关的操作都应该放在主线程内执行，而不是在新建线程内执行。

16.4 线程间任务安排

16.4 线程间任务安排

- 使用并发编程的最直接目的是保证界面流畅，这也是引擎占据主线程的原因。因此，除了界面相关的代码外，其他操作都可以放入新的线程中执行，主要包括文件读写和网络通信两类。
- 文件读写涉及外部存储操作，这和内存、CPU 都不在一个响应级别上。如果将其放入主线程中，就可能会造成阻塞，尤为严重的是大型图片的载入。对于碎图压缩后的大型纹理和高分辨率的背景图，一次加载可能耗费 0.2 s 以上的时间，如果完全放在主线程内，会阻塞主线程相当长的时间，导致画面停滞，游戏体验很糟糕。在一些大型的卷轴类游戏中，这类问题尤为明显。考虑到这个问题，Cocos2d-x 为我们提供了一个异步加载图片的接口，不会阻塞主线程，其内部正是采用了新建线程的办法。

■ 我们用游戏中的背景层为例，原来加载背景层的操作是串行的，相关代码如下：

```
bool BackgroundLayer::init()
{
    LOG_FUNCTION_LIFE;
    bool bRet = false;

    do {
        CC_BREAK_IF(! CCLayer::init());
        CCSize winSize = CCDirector::sharedDirector()->getWinSize();

        CCSprite *bg = CCSprite:: create ("background.png");
        CCSize size = bg->getContentSize();
        bg->setPosition(ccp(winSize.width / 2, winSize.height / 2));

        float f = max(winSize.width / size.width, winSize.height / size.height);
        bg->setScale(f);

        this->addChild(bg);
        bRet = true;
    } while (0);

    return bRet;
}
```

■ 现在我们将这一些列串行的过程分离开来，使用引擎提供的异步加载图片接口异步加载图片，相关代码如下：

```
void BackgroundLayer::doLoadImage(ccTime dt)
{
    CCSize winSize = CCDirector::sharedDirector()->getWinSize();

    CCSprite *bg = CCSprite::create("background.png");
    CCSize size = bg->getContentSize();
    bg->setPosition(ccp(winSize.width / 2, winSize.height / 2));

    float f = max(winSize.width/size.width,winSize.height/size.height);
    bg->setScale(f);

    this->addChild(bg);
}

void BackgroundLayer::loadImageFinish(CCObject* sender)
{
    this->scheduleOnce(schedule_selector(BackgroundLayer::doLoadImage), 2);
}

bool BackgroundLayer::init()
```

```
{
    LOG_FUNCTION_LIFE;
    bool bRet = false;
    do {
        CC_BREAK_IF(! CCLayer::init());

        CCTextureCache::sharedTextureCache()->addImageAsync(
            "background.png",
            this,
            callfuncO_selector(BackgroundLayer::loadImageFinish));
        bRet = true;
    } while (0);

    return bRet;
}
```

- 为了加强效果的对比，我们在图片加载成功后，延时了 2 s，而后才真正加载背景图片到背景层中。读者可以明显看到，2 s 后游戏中才出现了背景图。
- 尽管引擎已经为我们提供了异步加载图片缓存的方式，但考虑到对图片资源的加密解密过程是十分耗费计算资源的，我们还是有必要单开一个线程执行这一系列操作。
- 另一个值得使用并发编程的是网络通信。网络通信可能比文件读写要慢一个数量级。一般的网络通信库都会提供异步传输形式，我们只需要注意选择就好。

16.5 并发编程辅助（1）

16.5 并发编程辅助（1）

虽然引擎没有为我们封装线程类，但还是提供了一些组件，辅助我们进行并发编程。除了上面提到的异步加载图片，引擎还提供了消息中心 CCNotificationCenter。这是一个类似 Qt 中消息槽的机制，一个对象可以注册到消息中心，指定要接收的消息；而某个事件完成时，则可以发送对应的消息，之前注册过的对象会得到通知。主要有以下两个关键的接口函数：

```
void addObserver(CCObject *target,  //接收消息的对象
    SEL_CallFuncO selector,         //响应消息的函数
    const char *name,               //待接收的消息
    CCObject *obj);                 //指定消息的发送者，目前暂时为无用参数
void postNotification(const char *name); //发送一个消息
```

借助消息中心，异步事件之间的对象可以进一步减少耦合，使用事件驱动的方式编写代码。以游戏中的金币数变动为例，我们将菜单层添加为金币数量变化的消息的观察者，相关代码如下：

```
CCNotificationCenter::sharedNotificationCenter()->addObserver
    (this,callfuncO_selector(GameMenuLayer::coinChange), "CoinChange", NULL);
```

然后在开炮、捕获鱼等引起金币变化的地方发出该消息，从而触发菜单层的 coinChange 函数：

```
CCNotificationCenter::sharedNotificationCenter()->
    postNotification("CoinChange", NULL);
```

- 当然，在多线程的环境中，考虑到之前提到的原则，不可能直接在分离的线程中调用消息中心发送消息，我们可以建立一个线程间共享的消息池，让消息可以在不同线程间流动，或者说，我们需要建立一个线程安全的消息队列。下面我们创建一个线程安全的消息队列，代码如下：

```
class MTNotificationQueue : CCNode
{
    typedef struct
    {
        string name;
        CCObject* object;
    } NotificationArgs;

    vector<NotificationArgs> notifications;
    MTNotificationQueue(void);

public:
    static MTNotificationQueue* sharedNotificationQueue();
    void postNotifications(ccTime dt);
    ~MTNotificationQueue(void);
    void postNotification(const char* name, CCObject* object);
};
```

- 从接口上看，这个消息队列可以看做引擎自带的消息中心的补充，因为这里并不提供消息接收者的注册，仅仅是允许线程安全地向消息中心发出一个消息。这样也对应了一种处理模式：主线程负责绘图实现，在分离出来的子线程中完成重计算任务，计算完成后向主线程发回处理完毕的消息，消息是单向流动的，数据从磁盘、网络或其他任何地方经过处理后最终以视图的形式流向了屏幕。
- 在实现上，我们通过一个数组缓冲了各线程间提交的消息，稍后在主线程中将这些消息一次性地向 CCNotificationCenter 发出。其中需要保证的是，缓冲用的数组在不同线程间的访问必须是安全的，因此需要一个互斥锁。
- 不同线程间可共享的数据必须是静态的或全局的，因此互斥锁也必须是全局的。考虑到这个消息队列应该是全局唯一的单例，仅仅需要一个全局唯一的互斥锁与之对应：

```
pthread_mutex_t sharedNotificationQueueLock;
```

- 而考虑到这个互斥锁必须进行合适的初始化和清理，可以用一个类的全局变量管理其生命
- 周期：

```
class LifeManager_PThreadMutex
{
    pthread_mutex_t* mutex;

public:
    LifeManager_PThreadMutex(pthread_mutex_t* mut) : mutex(mut)
    {
        pthread_mutex_init(mutex, NULL);
    }

    ~LifeManager_PThreadMutex()
```

```
    {
        pthread_mutex_destroy(mutex);
    }
}__LifeManager_sharedNotificationQueueLock(&sharedNotificationQueueLock);
```

- 在 pthread 库中，我们使用下面一对函数进行互斥锁的上锁和解锁：

```
int pthread_mutex_lock (pthread_mutex_t * mutex);       //上锁
int pthread_mutex_unlock (pthread_mutex_t * mutex);     //解锁
```

- 这里的上锁函数是阻塞性的，如果目标互斥锁已经被锁上，会一直阻塞线程直到解锁，然后再次尝试解锁直到成功从当前线程上锁为止。

16.5 并发编程辅助（2）

16.5 并发编程辅助（2）

- 同样，考虑到上锁过程往往对应了一段函数或一个程序段的开始和结束，可以对应到一个临时变量的生命周期中，我们再次封装一个"生命周期锁类"：

```
class LifeCircleMutexLocker
{
    pthread_mutex_t* mutex;

public:
    LifeCircleMutexLocker(pthread_mutex_t* aMutex) : mutex(aMutex)
    {
        pthread_mutex_lock(mutex);
    }
    ~LifeCircleMutexLocker(){
        pthread_mutex_unlock(mutex);
    }
};

#define LifeCircleMutexLock(mutex) LifeCircleMutexLocker __locker__(mutex)
```

一切准备就绪后，就剩下两个核心的接口函数--向队列发出消息以及由队列将消息发

- 到消息中心中，相关代码如下：

```
void MTNotificationQueue::postNotifications(ccTime dt)
{
    LifeCircleMutexLock(&sharedNotificationQueueLock);

    for(int i = 0; i < notifications.size(); i++) {
        NotificationArgs &arg = notifications[i];
        CCNotificationCenter::sharedNotificationCenter()->
            postNotification(arg.name.c_str(), arg.object);
    }
    notifications.clear();
```

```
}

void MTNotificationQueue::postNotification(const char* name, CCObject* object)
{
    LifeCircleMutexLock(&sharedNotificationQueueLock);

    NotificationArgs arg;
    arg.name = name;

    if(object != NULL)
        arg.object = object->copy();
    else
        arg.object = NULL;

    notifications.push_back(arg);
}
```

- 实际上，这是两个非常简短的函数，仅仅是将传入的消息缓冲到数组中并取出。唯一的特别之处只在于函数在开始时，使用了我们前面定义的"生命周期锁"，保证了在访问缓冲数组的过程中是线程安全的，整个读写过程中缓冲数组由当前线程独占。
- 最后，我们启动消息队列的定时器，使 postNotifications 函数每帧被调用，保证不同线程间发出的消息能第一时间送达主线程：

```
CCDirector::sharedDirector()->getScheduler()->scheduleSelector(
    schedule_selector(MTNotificationQueue::postNotifications),
    MTNotificationQueue::sharedNotificationQueue(),
    1.0 / 60.0,
    false);
```

- 有了这个消息池，就可以进一步简化之前的图片加载过程了。下面仍然使用背景层的例子，再次重写游戏背景层的初始化函数：

```
bool BackgroundLayer::init()
{
    LOG_FUNCTION_LIFE;

    bool bRet = false;
    do {
        CC_BREAK_IF(! CCLayer::init());

        CCNotificationCenter::sharedNotificationCenter()->addObserver(
            this,
            callfuncO_selector(BackgroundLayer::loadImageFinish),
            "loadImageFinish",
            NULL);

        pthread_t tid;
```

```
        pthread_create(&tid, NULL, &loadImages, NULL);

        bRet = true;
    } while (0);

    return bRet;
}
```


16.5 并发编程辅助（3）

我们不再按照注释中的做法那样使用系统的纹理缓存来异步添加背景图片，而是先注册到消息中心，而后主动创建一个线程负责加载图片。在该线程中，我们仅完成图片向内存的加载，相关代码如下：

```
void* loadImages(void* arg)
{
    bgImage = new CCImage();
    bgImage->initWithImageFileThreadSafe("background.png");

    MTNotificationQueue::sharedNotificationQueue()->
        postNotification("loadImageFinish", NULL);

    return NULL;
}
```

在加载完成之后，我们通过消息队列发出了一个加载完成的消息，在稍后的消息队列更新时，这个消息将会被发送到消息中心，而后通知到背景层的响应函数中。我们为背景层添加相应的响应函数 loadImageFinish，其代码如下：

```
void BackgroundLayer::loadImageFinish(CCObject* sender)
{
    CCSize winSize = CCDirector::sharedDirector()->getWinSize();
    CCTexture2D* texture = CCTextureCache::sharedTextureCache()->
        addUIImage(bgImage, "background.png");
    bgImage->release();

    CCSprite* bg = CCSprite::create(texture);
    CCSize size = bg->getContentSize();
    bg->setPosition(ccp(winSize.width / 2, winSize.height / 2));

    float f = max(winSize.width/size.width,winSize.height/size.height);
    bg->setScale(f);

    this->addChild(bg);
}
```

这里 bgImage 是用 new 方式创建的，堆空间是除了全局静态对象之外唯一可以在线程间共享的数据空间。

必须注意的是，作为共享数据的 bgImage 的内存管理方式，我们在加载线程中用 new 从堆空间中分配了该内存，但是并没有遵守内存管理规范在该函数中将其释放，因为此时 bgImage 还未使用完毕，也不可能调用自动释放池，因为在子线程中

是不存在自动释放池的，如果跨线程调用了自动释放池，将造成严重的紊乱。因此，我们最后在 loadimageFinish 中添加到纹理缓存后才将其释放。

- 这也是使用多线程进行并发编程时的一个比较大的障碍，由于引擎的内存管理体系 CCObject 是非线程安全的，而整个引擎又是搭建在 CCObject 提供的内存管理机制基础上的，因此我们在多线程环境中使用引擎的任何对象都必须分外小心。

● 16.6 小结

- **16.6 小结**
- 透过多线程的并发编程，我们可以更充分地利用底层的计算资源，游戏的流畅度得到进一步的提升。从这一章的介绍可以看到，对于常见的场景，引擎都为我们提供非常友好的辅助，帮助我们更容易实现诸如消息驱动等并发编程模型。下面总结这一章的知识点。
- pthread：一套 POSIX 标准下的线程库，可以在各种系统下实现线程操作。正是由于可移植性极高，Cocos2d-x 提供了对 pthread 库的支持，因此我们不必添加额外的引用就可以直接使用它了。
- 互斥锁：线程间常常出现不同步的问题，例如两个线程同时执行变量递增的操作，结果很可能是变量只递增一次。为了解决这个问题，我们引入了互斥锁。在 ptherad 中，pthread_mutex_t 表示一个互斥锁。互斥锁是解决线程间同步问题的有力工具，当一个线程进入互斥锁时，其他线程将无法请求互斥锁，直到互斥锁被释放，这就保护了关键代码不会被并行地执行。
- 异步加载图片：Cocos2d-x 提供了 CCTextureCache::addImageAsync(const char *path, CCObject *target, SEL_CallFuncO selector)方法来实现图片的异步加载。本质上，它也是利用多线程来实现这个操作的，然而若我们只希望实现图片的异步加载，就可以直接利用这个功能，没有必要自己完成整个线程操作了。

● 17.1 Windows 8

- **多平台下的 Cocos2d**
- 现在，我们相信读者已经大概了解 Cocos2d-x 引擎的主要特性了。在这一章中，我们将探讨跨平台这一新的话题，读者将见识到 Cocos2d-x 引擎无缝跨接各大平台这一强大的功能。相对于前面的章节，这是代码量较少的一章，读者可以将其看做是第 1 章的延续，我们将更多地展示游戏是如何轻松跨平台的。Cocos2d-x 对于 iOS、Android 和 Windows 已经支持已久了，而最近的 Cocos2d-x 将支持推广到了微软的 Windows 8（指 Windows 8 的现代 UI，曾经代号为 Metro）和 Windows Phone 8 两大移动平台上。同时，借助 Cocos2d-HTML5，也可以轻松开发浏览器上的游戏。
- **17.1 Windows 8**
- Windows 8 已经在 2012 年年底发布，是微软的新一代操作系统。对开发者来说，其最大的卖点是几乎无缝兼容了平板电脑，将成为继 iOS、Android 之后在平板电脑上的第三个操作系统。与以往的 Windows 产品相比，其现代界面风格改变非常大，更适合于平板等可触摸设备。
- 目前，Cocos2d-x 已经将支持的平台扩展到 Windows 8 上，开发语言仍然是 C++，依然延续了引擎隐藏底层的优良传统。通过引擎调用的精灵、层、动作和动画，可以不作任何修改即可直接移植到 Windows 8 平台上，而我们只需要在 Windows 8 的 Visual Studio 2012 上重新编译一次即可。
- 得益于 Windows 8 对平板等触摸设备的支持，Cocos2d-x 游戏在 Windows 8 中也支持多点触摸，而重力加速计等特性也将视设备硬件获得支持。相比 Windows 版本的 Cocos2d-x，Windows 8 版本更像一个移动设备的引擎了。
- 可能引起迷惑的是底层绘图的相关调用。Windows 8 操作系统上的应用将具有经典和现代两种界面风格（也被称为 Classic 风格与 Metro 风格），而现代风格应用将允许登录微软的应用商店直接在网上销售，而且考虑到平板电脑的特点，现代风格界面很可能是唯一支持的交互风格，所以我们的应用或游戏都要为现代风格界面交互做一些调整，其中最最要紧的就是

底层绘图了。目前，Windows 8 的现代风格界面应用仅由 DirectX 提供绘图支持，也就是说，我们必须放弃 OpenGL 绘图 API。好在二者区别不是特别大，实现同样的功能几乎都能互相找到替代品，只需要简单地替换即可。

- 17.2 Windows Phone 平台

■ 17.2 Windows Phone 平台

■ Windows Phone 是微软在移动平台的另一力作。早在 2011 年秋季，多家公司就发布了 Windows Phone 7 的手机，使 Windows Phone 一跃成为继 iOS 和 Android 之后的第三大手机操作系统。

■ 和 Windows 8 平台一样，出身微软的 Windows Phone 也暂不提供 OpenGL 底层绘图支持，可以使用 DirectX 与 XNA 游戏开发组件，其中 XNA 是一套兼容微软家庭娱乐终端（Xbox、Wii 等）的游戏绘图组件，尤其针对这些娱乐终端的绘图做了优化。

■ Windows Phone 7 和其他平台更大的不一致在于，为了保护 Windows Phone 7 的底层硬件，并没有开放 C++接口，而是统一使用 C#进行开发。于是 Cocos2d-x 也就扩展到了 Cocos2d-x for XNA 版本，使用 XNA 支持底层绘图，向上提供几乎和同期 C++版本完全一致的开发接口。加上 C#和 C++同为面向对象的语言，实际上，不需要太多的改动就可以将游戏从其他平台移植到 Windows Phone 7 平台上。

■ 关于 C#的介绍在此略过，相信具备良好 C++基础的读者都可以快速上手。唯一需要提的一点是，C#语言内建了垃圾回收机制，也就意味着我们不再需要进行琐碎的内存管理了。

■ 而在最新发布的 Windows Phone 8 系统上，微软进一步开放了 C++的开发权限，因此 Cocos2d-x 也加入了对 Windows Phone 8 的支持。读者随时可以在 Cocos2d-x 的官方网站首页找到最新版本的 Windows Phone 8 版本下载。Cocos2d-x 的 Windows Phone 8 版本提供了与其他版本一致的接口，因此把 Cocos2d-x 游戏迁移到 Windows Phone 8 平台并不困难，只需要重新编译并进行少量的修改即可。可以认为，Cocos2d-x 游戏可以直接部署到 Windows Phone 8 上而不需要移植，所以在第 20 章中，我们就不对 Cocos2d-x for Windows Phone 8 进行讨论了。

- 17.3 Cocos2d-HTML5

■ 17.3 Cocos2d-HTML5

■ 最后值得一提的是 Cocos2d-HTML5 这一版本。这是 Cocos2d 搭建在 HTML5 上的版本，基于 Canvas 绘图，使用 JavaScript 开发。必须提及的原因是，JavaScript 作为脚本语言给游戏开发带来了十分巨大的变革，它是对 Cocos2d-x 可移植性的进一步发展与补充。

■ 首先，现在任何一个操作系统都一定具备浏览器，基于 HTML5 编写的游戏可以说天生具备了跨平台的特性。其次，JavaScript 作为一种脚本语言可以直接运行在任何设备之上，Cocos2d-x 与 Cocos2d-iPhone 都提供了一套完全兼容 Cocos2d-HTML5 的 API，这意味着我们编写的 Cocos2d-HTML5 游戏不需要进行额外的工作就可以无缝集成到现有引擎之中。早期 HTML5 曾经被诟病的性能问题，在设备硬件升级和引擎底层充分调优后，已经不存在了。

■ 毫无疑问，Cocos2d-HTML5 将成为未来开发的趋势。在第 19 章中，我们将简单介绍 Cocos2d-HTML5 的开发流程。

- 17.4 移植

■ 17.4 移植

■ 最后，我们来探讨移植这个话题。随着移动终端的发展，不可避免地会产生游戏移植的需求。可能出现两种情况，一是 Cocos2d-x 游戏要从一个平台移植到另一个平台；二是基于其他引擎编写的游戏想借助 Cocos2d-x 实现跨平台。下面我们分开讨论这两种情况。

■ 对于采用 Cocos2d-x 引擎实现的跨平台游戏，iOS、Android、Windows Phone 8 三个基础的平台是可以无缝切换的，许多情况下只要重新编译就可以了。而如果是 Windows Phone 7、浏览器这些非原生 Cocos2d-x 支持的平台，那么我们将面临大量一比一转换的代码，这时候可以考虑先使用语言间的机械转换机，结合正则表达式等全文替换，将游戏移植到目标平台后再进行细微的调试。

- 而对于从其他引擎转入 Cocos2d-x 引擎的游戏，实际上相当于一次重构。从现在的发展方向看，几乎不可能丢下 iOS、Android 和 Windows Phone 中的任何一个平台，横跨所有平台是大势所趋，我们推荐在这个时候使用 JavaScript 开发基于 Cocos2d-HTML5 的游戏，利用强大的可移植性为我们的开发节省大量时间。
- 关于移植的内容，我们将在第 20 章中详细讨论。

17.5 小结

- 17.5 小结
- 在这一章中，我们讨论了和跨平台相关的话题，微软的两个新平台对 C++开发的不开放迫使我们必须转到脚本化的开发方式上来。其实这也是引擎升级的一个很好的动力，推动着引擎和开发技术的发展。在后面的章节中，我们将看到在引擎中加入编辑器和脚本的威力。

18.1 CocosBuilder 可视化开发

- 可视化开发

- 进行过图形界面开发的读者是否希望在 Cocos2d-x 中也能出现"界面编辑器"之类的图形化开发工具呢？幸运的是，在 OS X 系统下已经出现了成熟的 Cocos2d 游戏编辑器 CocosBuilder。在本章中，我们将介绍这一方便游戏开发的利器。

- 18.1 CocosBuilder 可视化开发
- 到目前为止，我们的《捕鱼达人》是纯代码编写的，使用图形工具辅助的地方并不多。在实际的开发中，势必导致所有的工作都需要程序员参与，这样引发的问题是对界面元素的修改工作繁重，尤其是反复调整界面排布细节的时候，要求美工和程序员同时在场参与调整。
- 如果读者有应用开发经历，想必会怀念当时配套的可视化界面布局工具，从 Eclipse 到 Visual Studio 到 Xcode，都提供了足够强大的对可视化界面布局工具的支持。作为跨平台的引擎，Cocos2d-x 暂时还未推出自己的集成开发环境，但是如同前面看到的碎图合并等一系列工具一样，存在足够多的开发工具可以解决我们开发中遇到的问题。
- CocosBuilder 就是针对界面布局的一个补充，这个工具的最新版本已经免费开源，并由著名的社交游戏公司 Zynga 维护升级。
- 比较可惜的是，CocosBuilder 目前只支持 OS X 系统，并且它的功能还不能做到像 Visual Studio 或者 Unity 那样完全自动地所见即所得，编辑得到的配置文件还需要在程序中用代码载入，好在这个过程并不复杂。基本上，只要大体的框架搭建好，坐标、大小等美术细节的调整完全可以交由美工独立完成了。
- 下面我们就简单介绍如何使用 CocosBuilder。

18.2 使用 CocosBuilder 创建场景

- 18.2 使用 CocosBuilder 创建场景
- 安装并打开 CocosBuilder，可以看到如图 18-1 所示的主界面。依次单击"File"→"New"→"New Project"菜单项，新建一个项目，然后把新建的项目保存到一个新建的文件夹中。

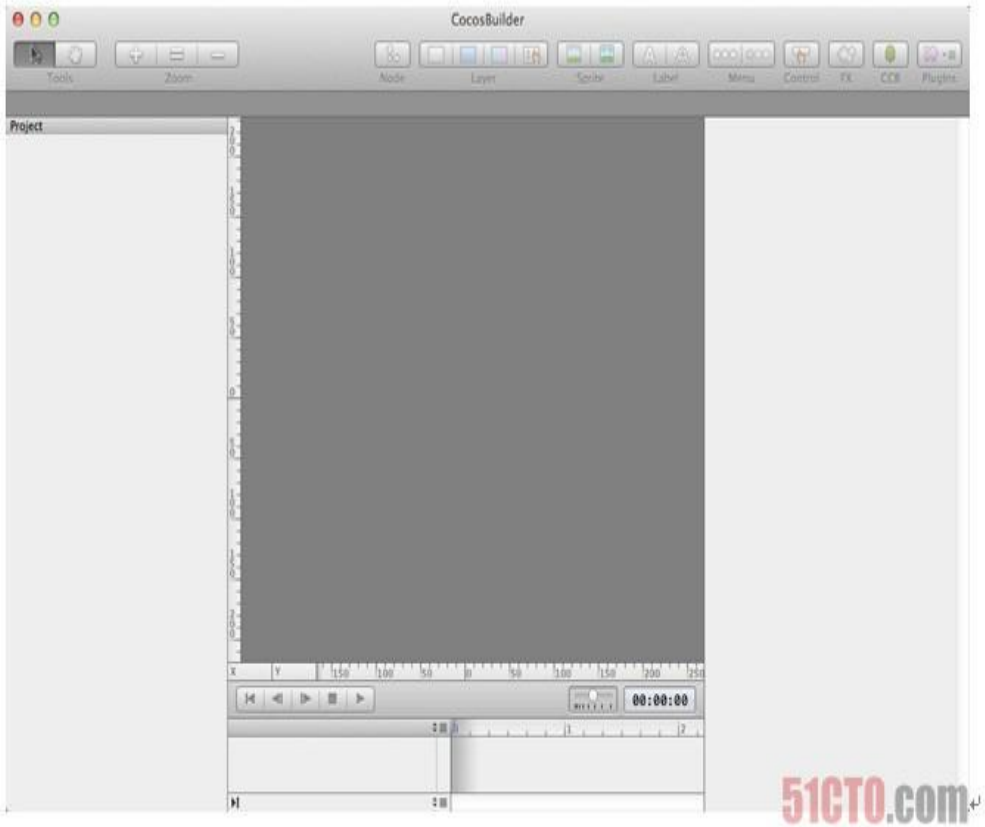


图18-1 CocosBuilder主界面

- 单击"File"→"New"→"New File"菜单项，创建一个 CocosBuilder 对象文件。创建该对象文件时，可以根据对象的类型来选择"Root object type"（如图 18-2 所示）。例如，当我们希望创建一个层时，就在此处选择"CCLayer"，当我们希望创建一个普通节点时，就选择"CCNode"。通常只需要选择"CCLayer"即可。此外，我们还需要选择这个对象对应的设备及分辨率。在 CocosBuilder 中，同一个对象可以根据不同的设备分辨率创造不同的布局。

图18-2 选择 "Root object type"

- 单击"File"→"Project Settings"菜单项，在弹出的对话框中单击"+"按钮（如图 18-3 所示），此时会弹出一个文件选择对话框，从中打开项目所需的资源文件。把"Publish directory"设为 Cocos2d-x 项目中的一个新文件夹，以便此后把编辑好的 CocosBuilder 项目导入 Cocos2d-x 项目之中。最后单击"Done"按钮。

图18-3 添加资源路径

- 接着在"Object"菜单中添加所需的部件，在屏幕的编辑器中调整部件的位置和大小等属性。选中一个部件，可以在右方的检查器中修改该部件的详细属性值。
- 编辑完成后，单击"File"→"Save"菜单项保存项目，再单击"File"→"Publish"菜单项打包项目，生成 ccbi 文件以备在 Cocos2d-x 项目中使用。

18.3 在 Cocos2d-x 项目中使用场景

18.3 在 Cocos2d-x 项目中使用场景

- 在 Cocos2d-x 中载入 CocosBuilder 项目，需要首先保证 CocosBuilder 发布的 ccbi 文件与资源文件都已经被 Cocos2d-x 项目引用。然后，我们需要使用以下方法来导入每一个 CocosBuilder 对象。
- 首先，创建 CocosBuilder 对应的类及与它对应的 Loader 类。例如，我们在 CocosBuilder 中创建了一个继承自 CCLayer 的 HelloWorldLayer，则应加入如下代码：

```
class HelloWorldLayer : public cocos2d::CCLayer
{
public:
    CCB_STATIC_NEW_AUTORELEASE_OBJECT_WITH_INIT_METHOD(HelloWorldLayer, create);
};

class HelloWorldLayerLoader : public cocos2d::extension::CCLayerLoader
{
```

```
public:
    CCB_STATIC_NEW_AUTORELEASE_OBJECT_METHOD(HelloWorldLayerLoader, loader);
protected:
    CCB_VIRTUAL_NEW_AUTORELEASE_CREATECCNODE_METHOD(HelloWorldLayer);
};
```

然后在需要使用 CocosBuilder 对象的位置，利用 CCNodeLoaderLibrary 和 CCBReader 等类创建对象实例，相关代码如下所示：

```
CCNodeLoaderLibrary* ccNodeLoaderLibrary =
    CCNodeLoaderLibrary::newDefaultCCNodeLoaderLibrary();
ccNodeLoaderLibrary->registerCCNodeLoader("HelloWorldLayer",
    HelloWorldLayerLoader::loader());
cocos2d::extension::CCBReader* ccbReader =
    new cocos2d::extension::CCBReader(ccNodeLoaderLibrary);
CCNode * node = ccbReader->readNodeGraphFromFile("ccb/HelloWorld.ccbi", this);
ccbReader->release();

if(node != NULL) {
    this->addChild(node);
}
```

完成后，运行我们的项目，就能看到在 CocosBuilder 中创建的层成功地显示在我们的游戏中了。此后，如果我们需要修改 ccbi 文件，则只需要更新游戏目录中的文件就可以了。

18.4 小结

18.4 小结

本章简单地介绍了 CocosBuilder 的基本使用方法。

CocosBuilder 的功能正在不断完善之中，除了可以帮助我们开发 Cocos2d-x 项目，还可以用来开发 Cocos2d-iPhone 项目。本章只演示了 CocosBuilder 的基本用法，实际上该软件还支持自定义类与动作的编辑，功能十分强大，因此建议读者自行探索 CocosBuilder，一定会得到丰富的收获。

19.1 概述

Cocos2d-HTML5

在前面的章节中我们已经提到，目前移动设备的游戏开发趋势是多平台开发，Cocos2d-x 就是为了实现让游戏运行在多个平台而开发的引擎。虽然引擎已经做到了 iOS、Android 与 Windows 下的跨平台，但对于新兴的 HTML5 平台来说则无能为力，Cocos2d-HTML5 就是为了解决这个问题而产生的。

19.1 概述

Cocos2d-HTML5 是基于 HTML 和 JavaScript 的游戏引擎，采用 HTML5 提供的 Canvas 对象与 DOM 进行绘图，因而使用 Cocos2d-HTML5 创建的游戏可以运行在各种主流的浏览器上，不需要依赖操作系统。表 19-1 列举出了各种主流平台与浏览器对 HTML5 的支持情况。可以看出，到目前为止，Cocos2d-HTML5 可以运行在绝大多数平台上。从这个角度来讲，Cocos2d-HTML5 是首个真正实现平台无关开发的 Cocos2d 引擎。

表 19-1　支持 HTML5 的浏览器

● 浏览器	● Windows	● Mac OS	● Linux
◆ Chrome	● 支持	◆ 支持	◆ 支持
◆ IE 9	● 支持	◆ 不支持	◆ 不支持
◆ Safari	● 支持	◆ 支持	◆ 不支持
◆ Firefox	● 支持	◆ 支持	◆ 支持
◆ Opera	● 支持	◆ 支持	◆ 支持

■ 基于 Cocos2d-HTML5 开发游戏时，游戏逻辑采用 JavaScript 实现。JavaScript 是一种动态、弱类型、基于原型（prototype）的脚本语言，可以直接在各种主流的浏览器上运行。与 C++和 C#等语言一样，JavaScript 的语法也是类 C 语言风格的，因此，熟悉 C 语言风格的开发者可以在极短的时间内掌握 JavaScript。此外，JavaScript 弱类型的特性使得代码书写更为简洁，也更为灵活。由于脚本语言是解释执行的，使用 JavaScript 编写的游戏在改动后不需要编译，可以直接执行。在反复修改细节的快速迭代期，这是不可多得的优点。

■ 采用 Cocos2d-HTML5 开发的游戏可以运行在几乎所有的图形操作系统之上，只要此系统拥有支持 HTML5 的浏览器，就可以运行我们开发的游戏。同样，在任何拥有 HTML5 浏览器的操作系统上，我们都可以进行游戏开发。只需要一个文本编辑器就可以编写游戏代码，只需要一个支持 HTML5 的浏览器就可以运行游戏。相对于其他的 Cocos2d 版本而言，Cocos2d-HTML5 对开发环境的要求是最低的。

■ 在 Cocos2d-HTML5 开发环境中，我们必须拥有的两个工具是文本编辑器和浏览器。文本编辑器用于创建并编辑游戏代码，实现游戏开发中所有的编码工作，浏览器用于运行和调试游戏。这两个工具构成了最基本的 Cocos2d-HTML5 开发环境。

■ 当我们拥有了基本的开发环境之后，理论上已经就可以任意地开发游戏了。然而文本编辑器毕竟只是一个简单的文字编辑工具，在更复杂的开发工作中，单纯的文本编辑器就显得软弱无力了。进行过项目开发的读者一定接触过 Visual Studio、Eclipse 或是 Xcode 这类集成开发环境（IDE），它们提供的诸多功能为游戏开发带来了极大的便利，这是文本编辑器无法比拟的。在 JavaScript 工程的开发中，也有功能强大的 IDE，在此我们向读者推荐 JetBrains 公司的开发工具 WebStorm。

■ WebStorm 是一项全能的 IDE，集成了网站开发所需的一切工具，可以编辑 HTML、CSS 以及 JavaScript 代码。对于我们关心的 JavaScript 开发，它提供了高级 IDE 所具备的全部功能，包括代码补全、代码导航、查找引用、代码重构、单元测试、代码质量评估等，是目前进行中型和大型游戏开发最有效的工具。遗憾的是，WebStorm 是一款收费软件，对于开发小型项目的初学者来说，完全可以采用其他免费的开发工具来代替它。关于其他的开发工具，我们将在后面的小节中加以介绍。

■ Cocos2d-HTML5 可以运行在任何浏览器中，当然也可以部署到 Chrome Web Store。通俗地讲，Chrome Web Store 是 Chrome 浏览器版本的 App Store。与移动设备应用商店不同的是，Chrome Web Store 提供的是运行在浏览器下的网页应用。使用 Cocos2d-HTML5 移植的《捕鱼达人》不久之前就在 Web Store 中发布了。

■ Cocos2d-HTML5 还可以运行在移动设备上。Cocos2d-x 与 Cocos2d-iPhone 的最新版本引入了 JavaScript 绑定的功能，允许在 C++与 Objective-C 程序中运行 JavaScript 脚本，并把自身导出为和 Cocos2d-HTML5 相同的 API，提供给脚本代码使用。以最新版本的 Cocos2d-x 为例，只要我们把 Cocos2d-HTML5 游戏的资源以及代码载入 Cocos2d-x 项目中，游戏就可以高效地运行在移动设备上了。此时脚本代码运行在引擎的 JavaScript 解释器之中，而游戏引擎直接运行在 CPU 之上。由于所有的绘图代码都是基于 C++语言编写的，所以游戏的效率可以得到保障。

● 19.2.1 开发环境介绍

■ **19.2 开发流程**

■ Cocos2d-HTML5 在 Cocos2d 家族中仍然是一个十分年轻的分支，与成熟平台相比，开发流程并不完全相同。在这一节中，我们将以一个简单的例子作为指导，分步详细讲解 Cocos2d-HTML5 游戏的开发流程。

■ **19.2.1 开发环境介绍**

- 与 Cocos2d-x 开发的流程一样，在正式开始开发游戏之前，我们需要搭建开发环境。Cocos2d-HTML5 所采用的开发环境由以下几个部分组成。
- 编辑器：用于编辑项目的代码文件，对于中大型项目，建议使用集成开发环境来编写
- 代码。
- 浏览器：用于查看游戏运行效果并进行调试。
- Web 服务器：用于托管游戏内容，这是可选组件。Web 服务器主要用来解决 Chrome 的安全性问题，因此使用 Firefox 的用户可以不必安装。
- Cocos2d 采用 XMLHttpRequest 对象来读取文件。对于偏好使用 Chrome 浏览器的用户来说，它的默认安全配置禁止了本地 HTML 文件使用 XMLHttpRequest 对象，我们有两个办法来解决这个问题。第一个办法是配置一个 Web 服务器端，把游戏页面发布到网站中；第二个办法是启动 Chrome 的时候加入-allow-file-access-from-files 或-disable-web-security 参数。
- 通常，我们可以选择轻量级的文本编辑器，如免费的 EditPlus，也可以选择较为复杂的 IDE；对于浏览器，我们可以选择目前执行效率最高的 Google Chrome；对于 Web 服务器，我们可以选择开源软件 Apache 或微软 IIS。

● 19.2.2 搭建开发环境（1）

- 19.2.2 搭建开发环境（1）
- 在初步了解了开发 Cocos2d-HTML5 所需的工具后，我们开始着手搭建一个 Windows 下的开发环境。开发 Cocos2d-HTML5 项目的意义在于不仅可以把游戏运行在浏览器上，更重要的是我们可以利用 Cocos2d-x 或 Cocos2d-iPhone 把 JavaScript 实现的游戏运行在移动设备上。考虑到需要部署到全部平台的工程规模很大，因此在这一节中我们采用了大型的 IDE。本节中我们选择的开发环境如下所示。
- 编辑器：WebStorm
- 浏览器：Google Chrome
- Web 服务器：Apache（包含在 XAMPP 安装包之中）
- 下面我们开始搭建开发环境。
- 第一步：安装 Google Chrome
- Google Chrome 可以在 Google 官方网站下载，下载页面的网址为 https://www.google.com/intl/
- zh-cn/chrome/browser/。
- 下载并安装完成后，我们就拥有了可以运行并调试 Cocos2d-HTML5 游戏的浏览器。
- 第二步：安装 Apache 服务器
- Apache 是一个广泛用于搭建网站的 Web 服务器，最初运行在类 UNIX 系统上，后来也被移植到了 Windows 系统上。Apache 可以作为一个独立的进程运行，因此与 IIS 相比，如果我们只是测试简单的 Web 应用，Apache 的安装与配置更加方便。
- 我们没有选择直接安装 Apache，而是通过包含了 Apache 的一个包 XAMPP 来安装。XAMPP 集成了 Apache 与其他建设网站必备的组件，同时还提供了这些工具的图形化控制面板，大大方便了我们的操作。
- 下面我们开始安装 XAMPP。XAMPP 的下载页面网址为 http://www.apachefriends.org/

- zh_cn/xampp-windows.html，可以随意选择下载 EXE 安装包或是 ZIP、7Z 压缩包。下载完成后运行安装程序，如果读者的系统是 Windows Vista 及以上，并且开启了用户账户控制（UAC），则安装程序会提示我们不要安装到"Program Files"文件夹下，因为此文件夹的读写受到了保护，这里我们选择安装到默认路径："C:\xampp"。
- 选择好安装路径后，就到了选择安装组件的部分。虽然 XAMPP 提供了各式各样的网站工具，但是我们只需要 XAMPP 控制面板以及 Apache，其余的都可以取消（如图 19-1 所示）。选择好之后，就可以开始安装 XAMPP 了。

图19-1 选择组件页面

- 等待进度条完成之后，安装程序会在结束前弹出对话框询问是否需要启动 XAMPP 控制面板（如图 19-2 所示），此时单击"是"按钮，稍等片刻，即可看到如图 19-3 所示的 XAMPP 控制面板主界面。

图19-2 XAMPP安装对话框

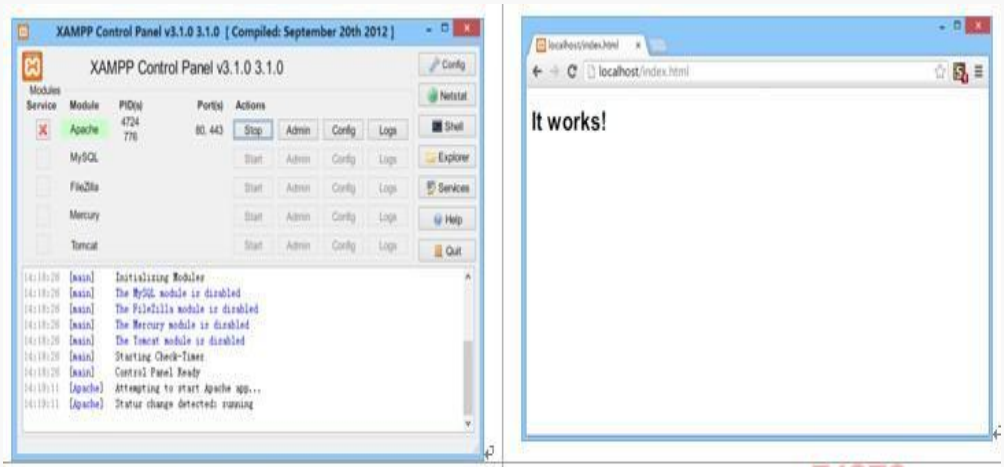
图19-3 XAMPP控制面板主界面

● 19.2.2 搭建开发环境（2）

19.2.2 搭建开发环境（2）

在这个界面中，列出了 XAMPP 控制面板所管理的网页服务，其中我们主要关心的部分是 Apache。单击 Apache 后的"Start"按钮即可启动 Apache 服务。当 Apache 启动后，我们可以发现"Start"按钮变为了"Stop"按钮，在下方的日志记录中也可以看到如"Status change detected: running"的字样（如图 19-4a 所示）。此时根据 Apache 的默认配置，使用浏览器打开本地的 http://localhost/

index.html 网页就能看到"It works!"字样了（如图 19-4b 所示）。



(a) 启动Apache　　(b) 测试网页

图19-4 运行Web服务器

服务器中的每个网页都对应着本地的一个文件，整个网站则对应了本地的一个目录。在默认配置下，本地的网页目录为"C:\xampp\htdocs"，可以在 Apache 配置文件中修改。我们访问的 http://localhost/index.html 则对应了文件"C:\xampp\htdocs\index.html"。在 XAMPP 控制面板主界面中，单击"Apache"一栏中的"Config"按钮，即可打开 Apache 配置文件。在配置文件中，"DocumentRoot"对应的值就是网页的本地目录（如图 19-5 所示）。

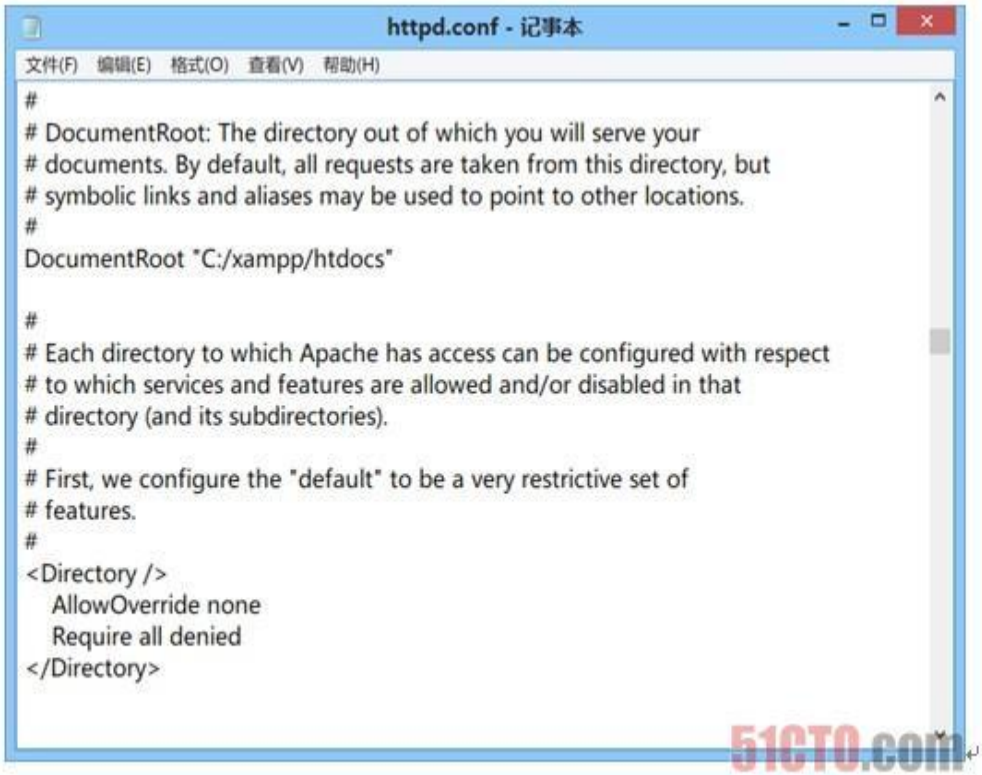


图19-5 Apache配置文件

- 在开发过程中，需要把我们开发的 Cocos2d-HTML5 游戏放置到这个目录中，然后使用浏览器访问本机地址 http://localhost 来运行游戏。
- 第三步：安装 WebStorm
- WebStorm 是一款收费的商业软件，而对于初学者来说，可以在 WebStorm 的网站上下载 30 天试用版。
- WebStorm 为我们提供了开发 Cocos2d-HTML5 游戏所必备的许多功能，其安装过程较为简单，在此不再赘述。有关 WebStorm 的基本使用方法，我们将在后面介绍。
- 第四步：获取 Cocos2d-HTML5 代码，并运行 Hello World 样例
- 我们可以从 http://www.cocos2d-x.org/projects/cocos2d-x/wiki/Html5 下载 Cocos2d-HTML5 的最新版本。下载下来的是一个 ZIP 文件，把它解压到一个单独的文件夹之下（本例中的文件夹为"Cocos2dhtml5"），接下来把该文件夹移动到前面所述的 Apache 本地目录中，即"C:\xampp\htdocs"下。至此，我们应该可以在"C:\xampp\htdocs\Cocos2dhtml5"目录下找到 index.html 文件（如图 19-6 所示）。
- 确保已经在 XAMPP 控制面板中运行 Apache 服务，然后打开 Google Chrome 浏览器，在地址栏中输入 http://localhost/Cocos2dHTML5/index.html 即可看到 Cocos2d-HTML5 页面。在这个页面中，可以找到 Hello World 和测试工程的链接，进入后可以看到 Cocos2d-HTML5 的运行效果。

图19-6 Cocos2d-HTML5目录

19.2.3 开始开发（1）

19.2.3 开始开发（1）

打开 WebStorm，我们可以看到如图 19-7 所示的欢迎界面。单击左边"Quick Start"栏目下的"Create New Project"新建工程，此时 WebStorm 将弹出新建项目对话框（如图 19-8a 所示）。然后选择项目位置，在 Location 一栏中填入"C:\xampp\htdocs\Cocos2dhtml5"，并单击"OK"按钮。我们在这一步选择的是 Cocos2d-HTML5 的路径，因此 WebStorm 会弹出如图 19-8b 所示的对话框，询问我们是否利用现有文件创建项目，此时选择"Yes"，WebStorm 就会把 Cocos2d-HTML5 的全部文件装载到项目中了。

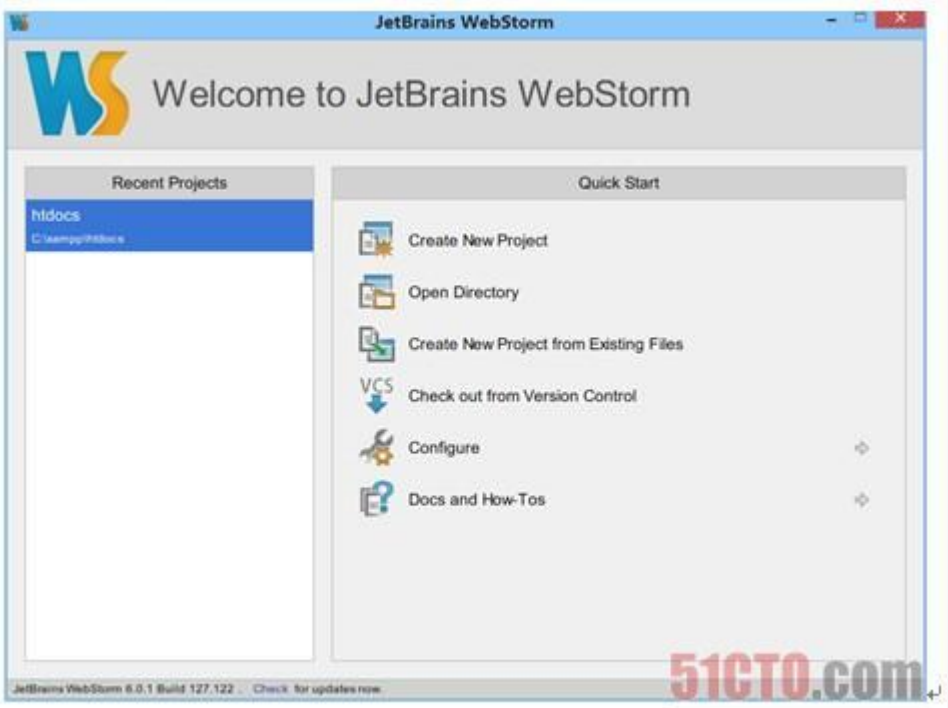
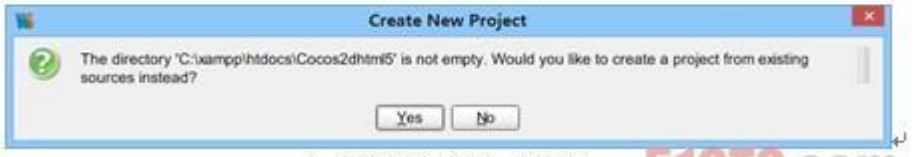


图19-7 WebStorm欢迎界面

（a）新建项目对话框



（b）项目已存在对话框

图19-8 新建项目

- 打开工程后的 WebStorm 布局如图 19-9 所示，左边是项目资源管理器，右边的大片区域是代码编辑器。在资源管理器中找到"HelloWorld"目录，此目录中的"Helloworld.js"就是 Hello World 项目的主要代码文件。

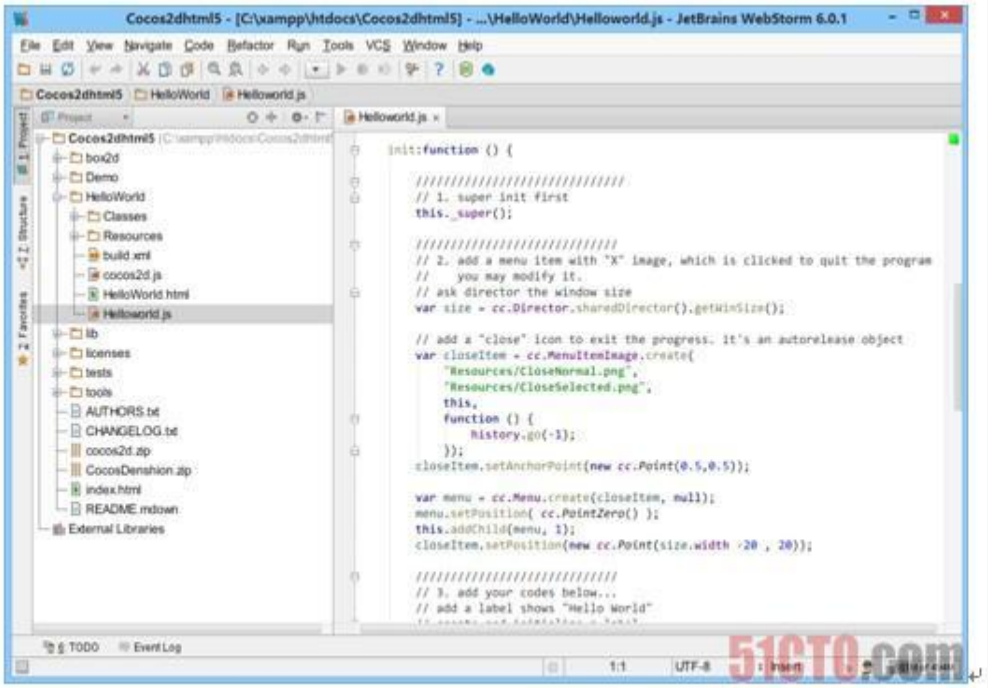


图19-9 WebStorm布局

- 可以看到，我们在此文件中创建了一个 HelloWorld 对象，它继承自 cc.Layer 类型，包含一个 init 方法。和 Cocos2d-x 一样，init 方法用于初始化一个对象，因此这个对象的主要功能集中在 init 方法中。

19.2.3 开始开发（2）

■ 读者在看到初始化方法中的代码后是否有熟悉的感觉呢？对比后我们会发现，Cocos2d-HTML5 版本的 Hello World 工程和本书介绍的 Cocos2d-x 版 Hello World 的代码几乎是一致的，这也体现出了 Cocos2d-HTML5 和其他 Cocos2d 版本的高度一致性。两个版本的初始化方法的代码如下所示：

```
//JavaScript
init:function () {
    this._super();

    var size = cc.Director.sharedDirector().getWinSize();

    var closeItem = cc.MenuItemImage.create(
        "Resources/CloseNormal.png",
        "Resources/CloseSelected.png",
        this,
        function () {
            history.go(-1);
        });
    closeItem.setAnchorPoint(new cc.Point(0.5,0.5));

    var menu = cc.Menu.create(closeItem, null);
    menu.setPosition( cc.PointZero() );
    this.addChild(menu, 1);
    closeItem.setPosition(new cc.Point(size.width -20 , 20));


    this.helloLabel = cc.LabelTTF.create("Hello World", "Arial", 38);
    this.helloLabel.setPosition(cc.ccp(size.width / 2, size.height - 40));
    this.addChild(this.helloLabel, 5);

    var lazyLayer = new cc.LazyLayer();
    this.addChild(lazyLayer);

   this.sprite = cc.Sprite.create("Resources/HelloWorld.png");
    this.sprite.setAnchorPoint(cc.ccp(0.5, 0.5));
    this.sprite.setPosition(cc.ccp(size.width / 2, size.height / 2));

    lazyLayer.addChild(this.sprite, 0);

    return true;
}


//C++
bool HelloWorld::init()
{
    if ( !CCLayer::init() )
```

```
    {
        return false;
    }

    CCMenuItemImage *pCloseItem = CCMenuItemImage::create(
        "CloseNormal.png",
        "CloseSelected.png",
        this,
        menu_selector(HelloWorld::menuCloseCallback) );
    pCloseItem->setPosition( ccp(CCDirector::sharedDirector()->
        getWinSize().width - 20, 20) );

    CCMenu* pMenu = CCMenu::create(pCloseItem, NULL);
    pMenu->setPosition( CCPointZero );
    this->addChild(pMenu, 1);

    CCLabelTTF* pLabel = CCLabelTTF::create("Hello World", "Arial", 24);
    CCSize size = CCDirector::sharedDirector()->getWinSize();

    pLabel->setPosition( ccp(size.width / 2, size.height - 50) );

    this->addChild(pLabel, 1);

    CCSprite* pSprite = CCSprite::create("HelloWorld.png");
    pSprite->setTag(233);
    pSprite->setPosition( ccp(size.width/2, size.height/2) );

    this->addChild(pSprite, 0);

    return true;
}
```

19.2.3 开始开发（3）

19.2.3 开始开发（3）

现在我们回到正题。在上一步打开的"Helloworld.js"代码文件中，找到创建文字标签"helloLabel"的一行代码，把 cc.LabelTTF.create()调用中的字符串"Hello World"改为"Hello Cocos2d-HTML5!"，然后保存文件。

现在打开 Google Chrome 浏览器，在地址栏中输入 http://localhost/cocos2dhtml5/HelloWorld/

HelloWorld.html 并按下回车，可以看到刚才的"Hello World"字样已经变为了"Hello Cocos2d-

HTML5!"，如图 19-10 所示。

图19-10 查看修改结果

- 下面我们为 Hello World 项目添加复杂一点的功能，并介绍如何使用 Google Chrome 调试游戏。返回 WebStorm 中打开的"Helloworld.js"文件，在 init 方法中插入如下代码：

```
this.schedule(this.tick, 1);
```

- 并在 init 方法的右大括号之后添加一个逗号，并加入如下代码，如图 19-11 所示。

```
counter1 : 0,
tick : function() {
    this.helloLabel.setString("Counter: " + this.counter1.toString());
    this.counter1++;
}
```

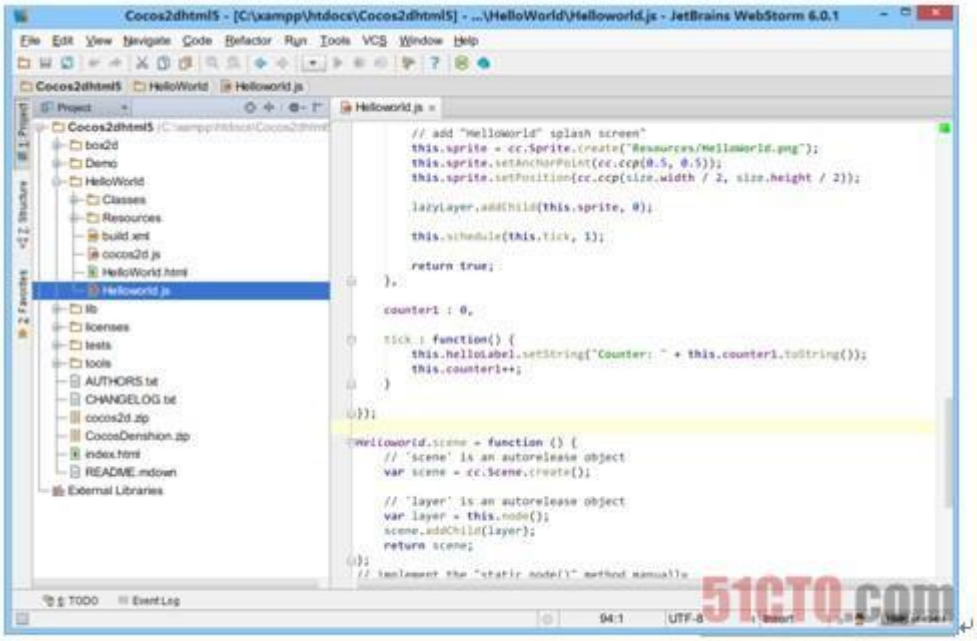


图19-11 添加一个定时器

- 当我们完成后，保存编辑，然后在 Google Chrome 浏览器中再次打开 http://localhost/cocos2
- dhtml5/HelloWorld/HelloWorld.html，可以看到原来的"Hello Cocos2d-HTML5!"字样随后会变为每秒增加的计数器（如图 19-12 所示），这说明我们的修改成功了。

图19-12 定时器效果

- 19.2.3 开始开发（4）

19.2.3 开始开发（4）

- 为了启用 Google Chrome 浏览器的调试功能，在游戏页面中右击，在弹出的快捷菜单中选择"审查元素"，此时浏览器窗口的下半部分就会出现一个调试区域，称为检查器（inspector）。我们在检查器的上方选择"Sources"，然后找到"Helloworld.js"文件（如图 19-13 所示），选中即可打开我们的游戏源码。

图19-13 在检查器中打开源码

■ 打开源码之后就可以对代码进行调试了。此时游戏正在执行中，因此我们必须使游戏暂停在某一处才可以开始调试。之前为 Hello World 添加了一个定时器事件 tick()，因此我们可以在定时器中下断点，使游戏在此处暂停。在检查器中找到 tick 方法，并在 this.counter1++;一行左边的行号处单击，Chrome 就会在此处添加一个断点。稍等片刻，浏览器就会在这一行上停下来。此时我们可以看到如图 19-14 所示的情况，浏览器会停在红色箭头所指的一行，并且浏览器的上方会显示"Paused in debugger"字样。

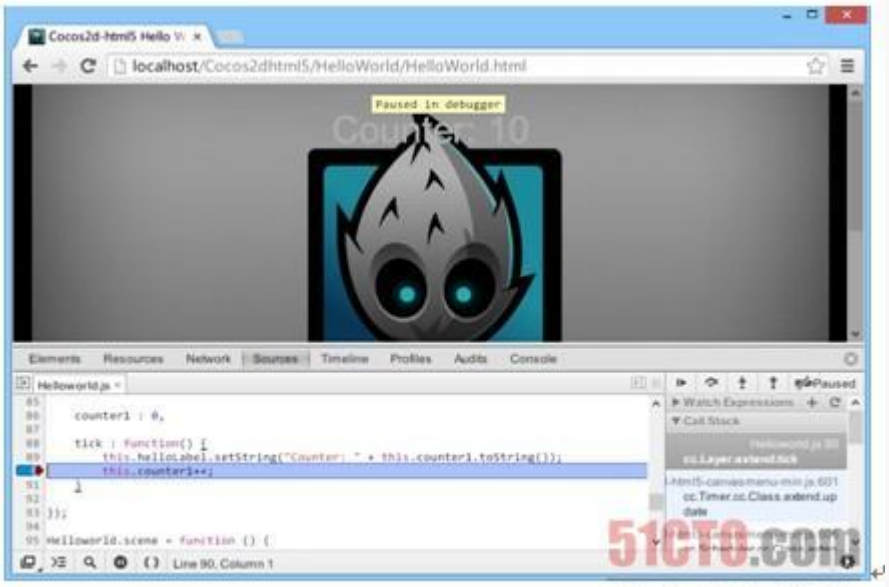


图19-14 设置断点

■ 成功暂停下来后，就可以如同在其他常见的 IDE 中一样进行调试了。Chrome 检查器提供了继续运行、单步步过、单步步入、跳出函数等基本操作，我们可以通过单击检查器右上角的按钮来执行这些操作。检查器的右方提供了表达式监视以及调用堆栈的显示区域，可以在此处进行高级的调试操作。而检查器的中央大片区域则是代码浏览器，虽然不能直接修改代码，但是可以通过鼠标在某一变量上悬停查看对象的值，十分便捷。

- 下面我们来尝试利用检查器修改变量的值。单击检查器左下方的"Show console"按钮（图 19-14 中位于左下角的第二个按钮），此时在检查器的下方就会弹出一个控制台。与其他 IDE 中的控制台相同，Chrome 控制台会显示脚本运行时的错误，也可以在调试阶段执行指令（与 Visual Studio 的"立即"窗口类似）。我们在控制台中输入 this.counter1 = 65535 并回车，然后单击"继续运行"按钮（图 19-14 中检查器右上角的第一个按钮），即可看到如图 19-15 所示的情况，游戏继续运行，但是计数器显示的数值从 65535 开始继续累加，这证明我们刚才的修改成功了。

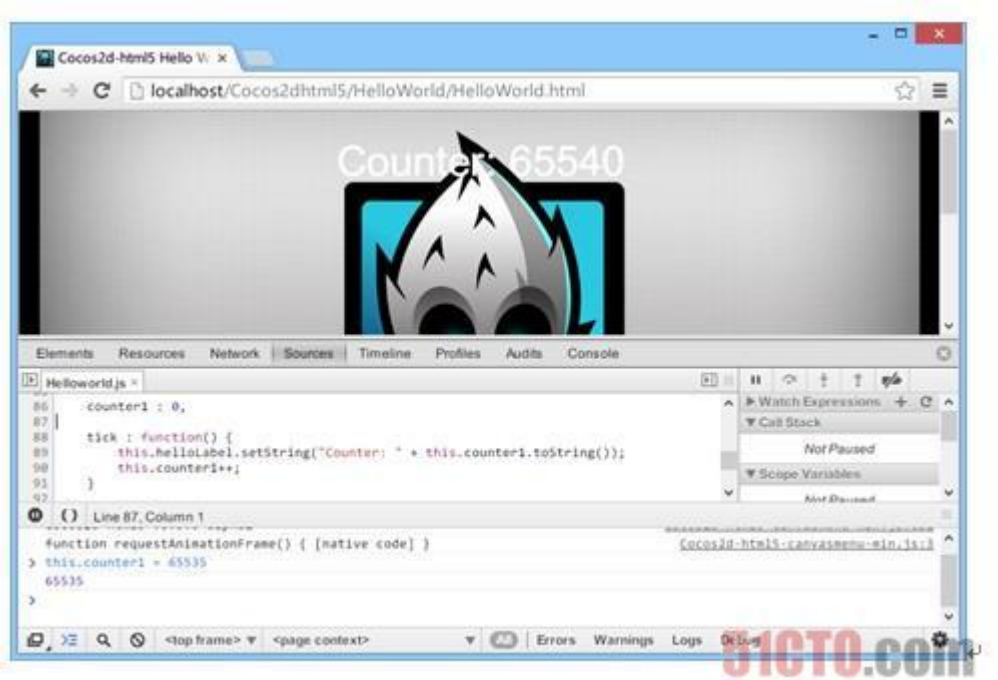


图19-15 修改效果

- 19.3 代码安全

- 19.3 代码安全

- 此时开发者对于使用 JavaScript 绑定可能还存在一点顾虑，那就是如何保证代码安全。这是脚本型语言的通病，不经过编译，会直接暴露在打包的资源文件中。代码安全对于游戏开发来说也是至关重要的，没有人希望本应保密的游戏数据被公开出来，从而降低游戏的趣味性，甚至是游戏数据或逻辑被恶意篡改。
- 为了保证游戏脚本的安全性，我们需要从两个方面考虑：第一是脚本代码的保密性，也就是指如何避免脚本代码被第三方查看；第二是脚本代码的完整性，也就是指如何避免脚本代码被篡改。如果能解决以上这两个问题，就可以认为游戏脚本是安全的了。
- 针对保密性问题，常见的做法有加密与混淆。加密是指开发者在编写好脚本后，为了避免把脚本的明文存放在游戏资源之中，应该加密脚本，然后与资源文件打包保存。在游戏执行脚本之前再进行解密。常见的加密方式有最简单的异或加密，稍微复杂的 DES 和 AES 加密，读者可以从网上找到许多相关资料，在此就不再赘述了。混淆则与加密不同，它是指对脚本的源码进行一系列等价变化，使人类难以阅读，但是对计算机执行脚本没有影响。在混淆的过程中，内部变量的名称、注释和代码缩进等信息全部丢失，因而对于人类来说是较难阅读的，可以有效提高脚本代码的破解难度。目前，较为著名的混淆器有 Google 开发的 Closure Compiler。混淆与加密可以同时进行，因此较为保险的方法是首先对脚本代码进行混淆，然后再加密，放入游戏资源中。
- 如果攻击者的目的只是查看脚本代码的内容，那么以上措施已经可以起到很大的作用了。然而如果攻击者的目的是修改脚本代码，直接把混淆加密后的原脚本替换成自己编写的恶意代码，则以上措施可能还不够有效。因此，针对完整性问题，还需要采取进一步的措施。

- 针对完整性问题，现在成熟的解决方案是数字签名。数字签名在电子商务中被广泛采用，但是在游戏开发领域却并不常见。事实上，我们也不需要实现一套完整的数字签名体系，只需要实现它的基本思想就可以了，在此简单介绍数字签名的原理。
- 数字签名是建立在"非对称加密算法"之上的一套体系，可以有效保证信息的完整性。加密算法可以分为对称密钥加密算法与非对称密钥加密算法两大类。我们平时熟悉的异或加密算法、DES 和 AES 等算法就属于对称密钥加密算法，它们的特点是加密与解密的时候使用同一个密钥。换句话说，只要取得了密钥，我就可以任意进行加密与解密操作。另一类非对称密钥加密算法则刚好相反。在加密与解密前，这类算法会生成一对密钥。之所以称为"一对密钥"，是因为其中有一个密钥只能用于加密，另一个密钥只能用于解密。我们把这对密钥称作公钥与私钥，从公钥算出私钥，或是从私钥算出公钥都在计算上都是不可行的（指需要花费极大的代价，例如，在一台超级计算机上运行上百年的运算量）。现阶段，RSA 和 DSA 就是广泛使用的非对称加密算法。
- 利用非对称加密算法可以实现身份的确认。某人为了证明自己的身份，可以采用如下做法：生成一对密钥，自己保留私钥用于加密，公开公钥用于解密。然后他选择一段信息，把信息公开的同时，利用私钥对其加密，并一同公开其密文。他的朋友为了确认他的身份，只需要用他公开的公钥来解密密文，如果密文与原文相同，则可以确认他的身份。只有私钥可以正确解密密文，而任何一个第三者都很难利用公钥算出私钥，因而可以确认此人的身份。
- 回到游戏脚本的完整性问题上来。为了保证脚本的完整性，开发者可以采取如下措施：生成一对密钥，私钥 Prk 用于加密，公钥 Puk 随游戏发行。然后仍旧使用 DES 等算法对脚本进行加密，计算脚本的哈希值 h（也称作数字指纹，可以代表信息的特征）。使用私钥 Prk 对哈希值 h 加密得到数字签名 s，与 DES 加密后的密文 m 一同打包。因此，我们的游戏资源中最终包含两个值：数字签名 s 与脚本密文 m。
- 在解密游戏时，我们首先使用 DES 密钥对密文 m 进行解密，得到脚本明文。然后利用公钥 Puk 对数字签名 s 进行解密，得到哈希值 h。现在游戏也对脚本明文进行一次哈希，得到的哈希值与 h 做对比，如果不同则认为脚本被篡改过，拒绝运行，否则可以认为脚本的完整性是有保证的。对于攻击者来说，它可以得到 DES 密钥，对脚本进行篡改。但是攻击者无法从公钥 Puk 计算出私钥 Prk，因此无法伪造出数字签名 s，从而使得游戏能够确认脚本的完整性是否被破坏。
- 这一节介绍了一些保证代码安全的方法，不过并没有给出具体的实现。按照以上介绍的思路，实现代码安全的方式多种多样，并没有固定的模式，因此鼓励读者实现自己的代码安全机制。代码安全固然重要，不过通常来说，花费过多的精力在这方面是得不偿失的，只要保证攻击者的付出的精力较大就是成功的代码保护了。作为开发者，我们更应该把精力投入到开发有趣的游戏上。

● 19.4 小结

- **19.4 小结**
- JavaScript 是一种动态、弱类型、基于原型的语言，可以直接在各大主流浏览器上运行，因此是跨平台开发的一个良好选择。Cocos2d-HTML5 就是利用 HTML5 的特性，使用 JavaScript 作为开发语言进行游戏开发的一个引擎，它实现了 Cocos2d-x 支持的绝大多数功能，因此是一个可以实用化的引擎。
- 为了进行 Cocos2d-HTML5 开发，我们需要的开发环境包含编辑器、浏览器和 Web 服务器，它们分别负责编写代码、运行调试和提供代码托管。本书介绍的开发环境组合为 WebStorm、Google Chrome 与 Apache，并展示了使用这 3 个工具进行开发的详细步骤。
- 本章还对比了 Cocos2d-HTML5 代码与 Cocos2d-x 代码，它们在本质上是一致的，这极大地降低了 Cocos2d-HTML5 的入门门槛。

● 20.1 命名原则

- **移植**

- 相对于曾经的 Cocos2d 版本，Cocos2d-x 最吸引人之处在于它的多平台性。基于 C++实现的 Cocos2d-x 提供了对 iOS、OS X、Android 以及 Windows 平台的支持，因此我们只需要编写一份代码，经过少量修改甚至完全不需要修改就可以运行在以上平台中。然而有的时候，我们还是会遇到移植相关的问题，比如下面的问题。
- 如果游戏是基于 Cocos2d-iPhone 编写的，能否迁移到 Cocos2d-x 中以获取平台上的通用性呢？
- 能否把基于 Cocos2d-x 的游戏部署到 Windows Phone 7 上呢？
- 对于以上问题，我们将通过介绍多种游戏移植技术来一一解答。在开始介绍游戏移植之前，我们需要回顾一下 Cocos2d-x 的命名原则。正是由于不同平台下的 Cocos2d 都一定程度地遵循着同样的命名原则，才使得游戏的移植成为可能。
- 本章中我们仅考虑目前较为热门的 4 个引擎：Cocos2d-iPhone、Cocos2d-x、Cocos2d-XNA 以及 Cocos2d-HTML5。Cocos2d 还有许多其他平台或编程语言上的移植版本，但是由于并不常见，我们暂且不做讨论。

20.1 命名原则

无论是Cocos2d-iPhone,还是派生自Cocos2d-iPhone的Cocos2d-x,甚至派生自Cocos2d-x的Cocos2d-XNA与Cocos2d-HTM5，都遵循类似 Objective-C 的命名原则。在这里，我们撇开引擎所使用的编程语言不谈，仅仅考虑它们共同采用的原则。

20.1.1 类名称

在 Objective-C 中并不存在命名空间的概念，为了区分不同程序集中的名称，通常会选择名称的双字母缩写作为类名称的前缀。在 Cocos2d-iPhone 中，我们选择"CC"（Cocos 的缩写）作为类名称的前缀。派生自 Cocos2d-iPhone 的 Cocos2d-x 系列引擎也采用了这个命名原则，因此 Cocos2d 的所有类名称都包含"CC"前缀。表 20-1 列举部分常用的类名称。

表 20-1 衍生版本中类的名称

Cocos2d-iPhone、Cocos2d-x 和 Cocos2d-XNA	Cocos2d-HTML5	描述
CCDirector	cc.Director	导演类
CCScene	cc.Scene	场景类
CCLayer	cc.Layer	层类
CCSprite	cc.Sprite	精灵类
CCArray	cc.Array	列表容器类

20.1.2 类函数

在绝大多数情况下，Cocos2d 中的方法名称都是严格对应的。也正是由于类函数名称的严格对应，我们才可以较为轻松地移植游戏。以 CCDirector 的 runWithScene 方法为例，我们可以看出这 4 种引擎惊人的相似，具体如下所示：

```
//Objective-C (Cocos2d-iPhone)
[[CCDirector sharedDirector] runWithScene:scene];

//C++ (Cocos2d-x)
CCDirector::sharedDirector()->runWithScene(scene);

//C# (Cocos2d-XNA)
CCDirector.sharedDirector().runWithScene(scene);
```

```
//JavaScript (Cocos2d-HTML5)
cc.Director.sharedDirector().runWithScene(scene);
```

- Cocos2d-iPhone 是使用 Objective-C 实现的，然而在这个语言中方法的名称与其他常见语言并不类似，它们通常较长，并且每个参数都拥有名称。因此 Cocos2d-x 规定，取 Objective-C 方法名中第一个参数以前的部分作为对应的 C 风格函数名。
- 一个典型的 Objective-C 方法的名称可能是这样的：

```
-(void) methodWithArg:(int)x arg2:(int)y arg3:(id)z
```

- 而它对应的 C 风格函数为：

```
void methodWithArg(int x, int y, int z)
```

- 在 Objective-C 中，并不存在严格意义上的构造函数，开发者需要调用以"init"为前缀的初始化方法，或者类提供的静态工厂函数来创建类。因此，Cocos2d-x、Cocos2d-XNA 与 Cocos2d-HTML5 也采用了这个模式。

20.1.3 属性

20.1.3 属性

- 并不是所有的语言都拥有属性的特性。Objective-C 使用@property 关键字来实现属性，这种方法在本质上把创建 get 和 set 访问器的工作交给了编译器。在 C++的语言特性中，不支持属性，因此游戏开发中使用"get"或"set"为前缀的方法来模拟属性。C#与 JavaScript 原生支持属性，但为了保持游戏引擎的通用性，Cocos2d-XNA 和 JavaScript 同样采用"get"与"set"前缀的访问器方法来获取与设置属性值。
- 下面的代码以一个精灵为例，演示了 4 个引擎如何获取与修改属性值：

```
//Objective-C
[fish setScale:2.0f];
fish.scale = 2.0f;  //另一种写法

//C++
fish->setScale(2.0f);

//C#
fish.scale = 2.0f;

//JavaScript
fish.scale = 2.0f;
fish.setScale(2.0f);  //另一种写法
```

20.1.4 选择器

20.1.4 选择器

- 选择器是 Objective-C 中用来代替类函数指针的类型。在 Cocos2d-x 和 Cocos2d-XNA 中，分别用类函数指针与委托(delegate)来模拟 Cocos2d-iPhone 上的选择器。而对于 Cocos2d-HTML5，有两种简单的方法来实现类似选择器的功能：既可以直接把函数对象当做选择器，也可以使用函数名字符串。
- 下面我们以 CCNode 的普通定时器为例，演示 4 种引擎中的选择器的用法，其中 timer1 为一个回调方法：

■

```
//Objective-C
[self schedule:@selector(timer1) interval:0.5f];

//C++
this->schedule(schedule_selector(MyClass::timer1), 0.5f);

//C#
this.schedule(new SEL_SCHEDULE(timer1), 0.5f);

//JavaScript
this.schedule(timer1, 0.5f);
this.schedule("timer1", 0.5f); //另一种写法
```

● 20.1.5 全局变量、函数与宏

■ **20.1.5 全局变量、函数与宏**

■ Objective-C 与 C++都是 C 语言的扩展，因此它们完整地保留了 C 语言的特性，其中包括全局变量、函数和宏定义。然而在 C#与 JavaScript 这类全新的语言中，并不完全支持这些特性。

◆ C#

■ 对于 C#来说，它完全没有全局变量、函数和宏的概念。因此对于全局变量和函数，我们只能采用静态类的方式来模拟。例如，可以创建一个 Global 类，在 Global 中添加静态变量以及静态函数：

```
//C#
public class Global
{
    public static int GlobalVar;
    public static void GlobalFunction(int arg)
    {
        //函数实现
    }
}
```

■ 然后我们就可以在代码的任何地方使用全局变量与函数了：

```
//C#
Global.GlobalVar = 1;
Global.GlobalFunction(0);
```

■ 最后一个问题是宏。在 C 语言或是 Objective-C、C++这类派生语言中，我们可以利用#define 语句来展开代码，但是在 C#中不存在宏的特性，取而代之的是一系列预处理指令：

```
#define
#if
#elif
#endif
```

- C#中的#define 只允许定义一个标签，且必须写在文件的头部，#if 则相当于 C 语言中的#ifdef，判断某个标签是否被定义。因此 C#的预处理指令实际上并不具备宏的特性，它只能作为条件编译来使用。因此，移植到 C#的宏都需要展开书写。当然，许多情况下我们可以使用一个函数来代替宏，以 ccp 宏为例，以下 C#代码实现了类似 ccp 宏的功能：

```
//C#
CCPoint ccp(float x, float y)
{
    return new CCPoint(x, y);
}
```

- JavaScript

- JavaScript 没有 C#严格，它允许存在全局变量和函数，但是为了避免变量名称产生冲突，我们可以把所有的全局变量和函数保存到一个全局对象中。例如，在 Cocos2d-HTML5 中所有的类、对象、函数都保存在 cc 全局对象中：

```
//JavaScript
cc.Sprite.create("fish.png");
```

- JavaScript 同样不支持宏，因此我们只能利用代码来实现宏的效果。对于 C 语言中的条件编译，我们可以采用 if 语句来判断：

```
//JavaScript
if(config.isPad)
{
    //在这里进行针对平板电脑的初始化
}
else
{
    //在这里进行针对手机的初始化
}
```

20.2 跨语言移植

20.2 跨语言移植

- 许多情况下，第一次开发所采用的平台并不一定合适，例如许多游戏采用 Cocos2d-iPhone 开发，但随着游戏的成功，开发者决定把游戏平台扩展到 Android，这就需要把基于 Cocos2d-iPhone 的游戏移植到 Cocos2d-x 下。而在其他情况下，当希望把游戏平台进一步扩展到 Windows Phone 7 上时，则需要把 Cocos2d-x 代码移植到 Cocos2d-XNA 上；当希望把游戏改写成基于 JavaScript 的脚本代码以便部署和升级维护时，则需要把游戏移植到 Cocos2d-HTML5 上。

- 无论是 Cocos2d-iPhone、Cocos2d-x、Cocos2d-XNA，还是 Cocos2d-HTML5，每一种游戏引擎采用的语言都不相同，因此我们就需要进行一项十分烦琐的工作：把一种语言实现的游戏转换成另一种语言实现。以《捕鱼达人》为例，现行 Android 版本的《捕鱼达人》1.6.3 版本的代码量高达 40 000 余行，如果逐行移植这个游戏，那将是一个不可想象的庞大工程。为此，我们将在下面介绍跨语言移植游戏的基本步骤以及技巧。

- 为了便于介绍，我们把移植游戏的过程划分为相对独立的 3 个阶段，每个阶段在开始之前都需要完成前一个阶段。而相对地，在每个阶段中多个人员可以同时进行移植工作，这使得移植速度得以保障。

20.2.1 第一阶段：代码移植

20.2.1 第一阶段：代码移植

- 最初我们需要进行的任务是把原游戏代码移植到目标平台上。为了便于描述，我们把待移植的版本称作原游戏，把原游戏使用的平台称作原平台。相对地，我们把新的游戏称作目标游戏，把新的平台称作目标平台。

- 在这个阶段，我们首先需要选取一个版本的游戏作为原游戏，然后在目标平台上对原游戏的代码进行 1:1 的翻译。如果按照翻译方式进行分类，则可以把翻译的顺序分为自顶向下与自底向上两种。自顶向下的移植适合进行小规模的全人工移植，通常不易出错，但速度较慢；自底向上的移植适合配合自动化工具进行大规模移植，速度较快，但是由于容易出错，还需要安排严密的测试。

- **自顶向下的移植**

- 在我们所说的"自顶向下"的移植方式中，"顶"指的是代码的高级架构，即类与类之间的关系，而"下"则指的是具体的语句。自顶向下的移植方式类似于工程开发的过程。回想在游戏开发中，我们通常先搭建游戏的架构，创建好所有的结构之后再逐个进行实现，最终完成游戏。自顶向下的移植方式与这种工作方式类似，我们同样根据原游戏的架构在目标平台上创建一个一致的架构，然后开始向框架中添加代码。

- 完成了游戏框架的创建，剩下的事情就是翻译每一条语句了。如同前一节所述，各个版本的游戏引擎结构与名称都按照一定的规则保持一致，因此我们几乎可以逐句对应着完成翻译的工作。当框架创建好后，翻译工作对整个工程翻译的完成度依赖不大，因此可以把工作分配给多个人员并行完成，以加快翻译的速度。翻译阶段最终的目标就是使得游戏的逻辑代码正确地翻译到目标平台使用的语言上，为了保证正确性，不仅需要代码翻译准确，通过编译，还需要保证类的继承、重载关系正确，否则程序的行为错误将极难排查。

- 自顶向下移植的优势在于，移植过程中首先创建与原游戏一致的架构，而这个过程是人工完成的，通常不易在架构的层次上出现错误，因此即使遇到了问题也可以轻易排查并解决。同时，在自顶向下的移植中，可以方便地引入软件测试模型，在移植中期甚至前期就可以看到部分移植的效果，可以有效保证代码质量。而自顶向下移植的缺点在于不便于引入自动化工具，整个代码移植的过程都需要人工参与，对于大型项目来说开销较大。因此，自顶向下的移植方式非常适合小型项目的移植。

- **自底向上的移植**

- 自底向上的移植方式指的是在不建立游戏架构的情况下，先直接对源码逐行进行对等的移植工作，然后再处理模块间的关系，最终完成整个架构的移植。在这种移植方式中，我们的工作主要集中于两个部分：第一部分是逐行的代码翻译，第二部分是修正模块间的关系。

- 首先进行逐行代码翻译工作。在自顶向下的翻译中，由于工作流程是首先建立架构，然后向架构中填充代码，填充的代码分散在各个函数或方法中，因此并不适合利用自动化的翻译工具进行处理。然而当我们采用自底向上的移植方法时，因为首先需要对全部代码进行翻译，然后再对架构上的错误进行修正，所以可以直接在翻译的时候引入自动化翻译工具。

- 逐行翻译代码实际上是一种不断重复的体力劳动，这种工作完全可以利用工具或脚本来代替人们的劳动，因此在此处我们引入自动化的翻译工具，用来降低逐行翻译代码的工作量。然而翻译工具并不能完全代替人们的工作。由于跨语言的翻译是一项十分困难的工作，工具通常也只能完成一部分工作，并且翻译的结果并不能保证完全正确。因此，我们利用工具翻译代码后还需要大量时间来修复。

- 第一类翻译工具的原理是正则表达式替换。正则表达式替换完全可以胜任许多简单的转换工作，例如 C++中的域符号"::"、箭头符号"->"、点符号"."，它们都等价于 C#中的点符号"."，于是我们可以创建一个正则表达式的替换，把 C++中的 3 种符号都替换成点符号。第一类翻译工具实现较为简单，任何一个开发者都可以轻松地编写正则表达式，然而正则表达式没有能力识别复杂的语句，处理后还需要手动修复许多代码。即使如此，这一类翻译工具仍然可以为我们减少许多时间。下面列举常用的此类型的工具或脚本。

- Objc2cpp（UNIX shell 脚本）：由 Steve Tranby 编写，可以实现 Objective-C 到 C++的翻译，需要运行在 shell 之下（通常，在 Linux、OS X 等类 UNIX 系统下都可以直接运行，在 Windows 下可以安装 Cygwin 套件来运行）。

- Objc2cpp（Sublime Text 2 插件）：由 ankstoo 编写，同样用于实现 Objective-C 到 C++的翻译。

- 第二类翻译工具又称作翻译编译器（source-to-source compiler、transcompiler 或 transpiler）。第一类翻译器只能转换词法以及部分简单的语法，而第二类翻译器的原理是语法分析，根据分析的结果生成目标语言的代码。它的原理与编译器类似，相对于第一类翻译工具，这类翻译工具可以转换较为复杂的语句，甚至直接处理整个工程。可以说，第二类翻译器是比较理想的翻译工具，但越是强大的工具，数量就越少。市面上的这类翻译器并不常见，下面列举部分此类型的翻译工具。

- C++ to C# Converter：由 Tangible Software Solutions 公司开发的翻译器，实现了 C++到 C#的翻译，支持表达式、函数、类与宏展开的翻译，翻译后的代码可用性极高。此软件是商业软件，收费较高。

- ObjC2J：由 Andremoniy 开发的开源翻译器，可以把 Objective-C 代码翻译成对应的 Java 代码。虽然 Cocos2d-x 衍生的许多引擎中并没有基于 Java 的版本，然而 Java 和 C#的语法极其类似，这个工具还是有一定利用价值的。此外，Objc2J 的源代码也公开在 Google Code 项目托管的网站中，把它修改为 Objective-C 到其他语言的翻译器也不失为一个好的想法。

- Emscripten：一个免费的 JavaScript 代码生成器，接受任何 LLVM 支持的语言（如 C 和 C++）并生成等价的 JavaScript 代码。它实际上进行的工作是把 LLVM 编译后的字节码转换成 JavaScript 代码，实现了极好的兼容性，却牺牲了代码的可维护性，翻译后的代码会丢失类型信息，因此在实际应用中局限性较大。

- 在实际的移植过程中，我们通常需要结合多种方法，必要的时候也可以自己编写工具来帮助我们快速完成工作。

- 通常，机器翻译的结果并不能直接使用。翻译主要能体现出两个问题。第一个问题是从支持宏的语言翻译到不支持宏的语言（例如从 C++翻译到 C#），宏的翻译容易出现错误，此时我们就必须手工修复宏的翻译了。在《捕鱼达人》Windows Phone 7 版本的移植中，我们采用了 C++ to C# Converter 来大面积翻译代码，并配合少量自己编写的正则表达式完善翻译结果。第二个问题是翻译工具对于每条语句的把握相对较为准确，但是对于继承关系的翻译常常会出错，如果忽略了检查工作，在测试阶段这种问题通常难以察觉与修复，因此翻译过后需要逐个类检查继承、重载关系，确保目标游戏的架构与原游戏一致。

- 4 个游戏引擎在内存管理方面并不完全一致。基于 Objective-C 的 Cocos2d-iPhone 与基于 C++的 Cocos2d-x 所使用的语言并不具备垃圾回收器，需要手动管理内存，而基于 C#的 Cocos2d-XNA 和基于 JavaScript 的 Cocos2d-HTML5 则可以使用垃圾回收器来自动管理内存。因此，在前两种语言中，我们不得不采取本书第一部分介绍的引用计数机制来管理对象的释放时机；而在后两种语言中，我们不需要关心对象的内存管理，因此所有的内存管理语句（如 retain 方法等）都不必保留。

- 无论是自顶向下的移植还是自底向上的移植，我们都不可能完全回避手动编写代码。游戏由游戏逻辑、游戏引擎以及其他库共同构成，游戏逻辑的实现又与其他两个部分密切相关。虽然这一章我们讨论的 Cocos2d-iPhone、Cocos2d-x、Cocos2d-XNA 与 Cocos2d-HTML5 四个游戏引擎都采用了统一的命名方式以方便开发者移植，但是所处的平台差异十分巨大，我们不可避免地会遇到代码上的差异。在下一阶段，我们将消除平台差异带来的问题。

- **20.2.1 第一阶段：代码移植**

- 最初我们需要进行的任务是把原游戏代码移植到目标平台上。为了便于描述，我们把待移植的版本称作原游戏，把原游戏使用的平台称作原平台。相对地，我们把新的游戏称作目标游戏，把新的平台称作目标平台。

- 在这个阶段，我们首先需要选取一个版本的游戏作为原游戏，然后在目标平台上对原游戏的代码进行 1:1 的翻译。如果按照翻译方式进行分类，则可以把翻译的顺序分为自顶向下与自底向上两种。自顶向下的移植适合进行小规模的全人工移植，通常不易出错，但速度较慢；自底向上的移植适合配合自动化工具进行大规模移植，速度较快，但是由于容易出错，还需要安排严密的测试。

- **自顶向下的移植**

- 在我们所说的"自顶向下"的移植方式中，"顶"指的是代码的高级架构，即类与类之间的关系，而"下"则指的是具体的语句。自顶向下的移植方式类似于工程开发的过程。回想在游戏开发中，我们通常先搭建游戏的架构，创建好所有的结构之后再逐个进行实现，最终完成游戏。自顶向下的移植方式与这种工作方式类似，我们同样根据原游戏的架构在目标平台上创建一个一致的架构，然后开始向框架中添加代码。

- 完成了游戏框架的创建，剩下的事情就是翻译每一条语句了。如同前一节所述，各个版本的游戏引擎结构与名称都按照一定的规则保持一致，因此我们几乎可以逐句对应着完成翻译的工作。当框架创建好后，翻译工作对整个工程翻译的完成度依赖不大，因此可以把工作分配给多个人员并行完成，以加快翻译的速度。翻译阶段最终的目标就是使得游戏的逻辑代码正确地翻译到目标平台使用的语言上，为了保证正确性，不仅需要代码翻译准确，通过编译，还需要保证类的继承、重载关系正确，否则程序的行为错误将极难排查。

- 自顶向下移植的优势在于，移植过程中首先创建与原游戏一致的架构，而这个过程是人工完成的，通常不易在架构的层次上出现错误，因此即使遇到了问题也可以轻易排查并解决。同时，在自顶向下的移植中，可以方便地引入软件测试模型，在移植中期甚至前期就可以看到部分移植的效果，可以有效保证代码质量。而自顶向下移植的缺点在于不便于引入自动化工具，整个代码移植的过程都需要人工参与，对于大型项目来说开销较大。因此，自顶向下的移植方式非常适合小型项目的移植。

- **自底向上的移植**

- 自底向上的移植方式指的是在不建立游戏架构的情况下，先直接对源码逐行进行对等的移植工作，然后再处理模块间的关系，最终完成整个架构的移植。在这种移植方式中，我们的工作主要集中于两个部分：第一部分是逐行的代码翻译，第二部分是修正模块间的关系。

- 首先进行逐行代码翻译工作。在自顶向下的翻译中，由于工作流程是首先建立架构，然后向架构中填充代码，填充的代码分散在各个函数或方法中，因此并不适合利用自动化的翻译工具进行处理。然而当我们采用自底向上的移植方法时，因为首先需要对全部代码进行翻译，然后再对架构上的错误进行修正，所以可以直接在翻译的时候引入自动化翻译工具。

- 逐行翻译代码实际上是一种不断重复的体力劳动，这种工作完全可以利用工具或脚本来代替人们的劳动，因此在此处我们引入自动化的翻译工具，用来降低逐行翻译代码的工作量。然而翻译工具并不能完全代替人们的工作。由于跨语言的翻译是一项十分困难的工作，工具通常也只能完成一部分工作，并且翻译的结果并不能保证完全正确。因此，我们利用工具翻译代码后还需要大量时间来修复。

- 第一类翻译工具的原理是正则表达式替换。正则表达式替换完全可以胜任许多简单的转换工作，例如 C++中的域符号"::"、箭头符号"->"、点符号"."，它们都等价于 C#中的点符号"."，于是我们可以创建一个正则表达式的替换，把 C++中的 3 种符号都替换成点符号。第一类翻译工具实现较为简单，任何一个开发者都可以轻松地编写正则表达式，然而正则表达式没有能力识别复杂的语句，处理后还需要手动修复许多代码。即使如此，这一类翻译工具仍然可以为我们减少许多时间。下面列举常用的此类型的工具或脚本。

- Objc2cpp（UNIX shell 脚本）：由 Steve Tranby 编写，可以实现 Objective-C 到 C++的翻译，需要运行在 shell 之下（通常，在 Linux、OS X 等类 UNIX 系统下都可以直接运行，在 Windows 下可以安装 Cygwin 套件来运行）。

- Objc2cpp（Sublime Text 2 插件）：由 ankstoo 编写，同样用于实现 Objective-C 到 C++的翻译。

- 第二类翻译工具又称作翻译编译器（source-to-source compiler、transcompiler 或 transpiler）。第一类翻译器只能转换词法以及部分简单的语法，而第二类翻译器的原理是语法分析，根据分析的结果生成目标语言的代码。它的原理与编译器类似，相对于第一类翻译工具，这类翻译工具可以转换较为复杂的语句，甚至直接处理整个工程。可以说，第二类翻译器是比较理想的翻译工具，但越是强大的工具，数量就越少。市面上的这类翻译器并不常见，下面列举部分此类型的翻译工具。

- C++ to C# Converter：由 Tangible Software Solutions 公司开发的翻译器，实现了 C++到 C#的翻译，支持表达式、函数、类与宏展开的翻译，翻译后的代码可用性极高。此软件是商业软件，收费较高。

- ObjC2J：由 Andremoniy 开发的开源翻译器，可以把 Objective-C 代码翻译成对应的 Java 代码。虽然 Cocos2d-x 衍生的许多引擎中并没有基于 Java 的版本，然而 Java 和 C#的语法极其类似，这个工具还是有一定利用价值的。此外，Objc2J 的源代码也公开在 Google Code 项目托管的网站中，把它修改为 Objective-C 到其他语言的翻译器也不失为一个好的想法。

- Emscripten：一个免费的 JavaScript 代码生成器，接受任何 LLVM 支持的语言（如 C 和 C++）并生成等价的 JavaScript 代码。它实际上进行的工作是把 LLVM 编译后的字节码转换成 JavaScript 代码，实现了极好的兼容性，却牺牲了代码的可维护性，翻译后的代码会丢失类型信息，因此在实际应用中局限性较大。
- 在实际的移植过程中，我们通常需要结合多种方法，必要的时候也可以自己编写工具来帮助我们快速完成工作。
- 通常，机器翻译的结果并不能直接使用。翻译主要能体现出两个问题。第一个问题是从支持宏的语言翻译到不支持宏的语言（例如从 C++翻译到 C#），宏的翻译容易出现错误，此时我们就必须手工修复宏的翻译了。在《捕鱼达人》Windows Phone 7 版本的移植中，我们采用了 C++ to C# Converter 来大面积翻译代码，并配合少量自己编写的正则表达式完善翻译结果。第二个问题是翻译工具对于每条语句的把握相对较为准确，但是对于继承关系的翻译常常会出错，如果忽略了检查工作，在测试阶段这种问题通常难以察觉与修复，因此翻译过后需要逐个类检查继承、重载关系，确保目标游戏的架构与原游戏一致。
- 4 个游戏引擎在内存管理方面并不完全一致。基于 Objective-C 的 Cocos2d-iPhone 与基于 C++的 Cocos2d-x 所使用的语言并不具备垃圾回收器，需要手动管理内存，而基于 C#的 Cocos2d-XNA 和基于 JavaScript 的 Cocos2d-HTML5 则可以使用垃圾回收器来自动管理内存。因此，在前两种语言中，我们不得不采取本书第一部分介绍的引用计数机制来管理对象的释放时机；而在后两种语言中，我们不需要关心对象的内存管理，因此所有的内存管理语句（如 retain 方法等）都不必保留。
- 无论是自顶向下的移植还是自底向上的移植，我们都不可能完全回避手动编写代码。游戏由游戏逻辑、游戏引擎以及其他库共同构成，游戏逻辑的实现又与其他两个部分密切相关。虽然这一章我们讨论的 Cocos2d-iPhone、Cocos2d-x、Cocos2d-XNA 与 Cocos2d-HTML5 四个游戏引擎都采用了统一的命名方式以方便开发者移植，但是所处的平台差异十分巨大，我们不可避免地会遇到代码上的差异。在下一阶段，我们将消除平台差异带来的问题。

20.2.2 第二阶段：消除平台差异（1）

- 完成了代码翻译后，我们还面临着许多问题。此时我们的游戏还是不能编译运行，因为游戏中多多少少会用到一些非游戏引擎的库，其中一部分是语言的公共库，如 Objective-C 中的 Cocoa、C++中的 STL、C#中的 CLR 等，另一部分是第三方库，如各种社交网络的分享 API。因此，我们需要修改代码来实现与新平台上各种库或 API 的对接。
- 对于公共库而言，最值得我们讨论的功能涵盖了容器、XML 文件处理和网络通信这 3 个方面，其他公共库的功能虽然也可能使用，但是用到的频率不高，限于篇幅我们暂且不做考虑。第三方库的移植则更是没有固定的解决方案可用，因此我们会介绍几种常见的处理方法。

- **容器**

- 容器是使用频率相当较高的一系列对象，它们在各种语言中都有各自的实现，在第二阶段我们需要把游戏所使用的容器统一到目标平台上。表 20-2 与表 20-3 列举了不同引擎所使用的容器类型。
- 表 20-2　线性表容器

● 平台	● 容器	● 例子
● Cocos2d-iPhone	● NSArray(NSMutableArray)	● [array addObject:obj]; obj = [array objectAtIndex:i];
● Cocos2d-x	● CCArray	● array->addObject(obj);obj = array->objectAtIndex(i);
● Cocos2d-XNA	● List<>	● array.Add(obj);obj = array[i];
● Cocos2d-HTML5	● JavaScript 数组	● array.push(obj);obj = array[i];

- 表 20-3　字典容器

● 平台	● 容器	● 例子

Cocos2d-iPhone	NSArray(NSMutableArray)	[[dictionary setObject:obj forKey:key];obj = [dictionary objectForKey:key];
Cocos2d-x	CCDictionary	dictionary->setObject(obj, key);obj = dictionary-> objectForKey(key);
Cocos2d-XNA	Dictionary<,>	dictionary[key] = obj;obj = dictionary[key];
Cocos2d-HTML5	JavaScript 关系数组	dictionary[key] = obj;obj = dictionary[key];

- **XML 处理**

  - 在讨论 XML 处理之前首先需要了解，Cocos2d 在所有平台上都提供了 Plist 文件的读写功能，而 Plist 就是一种特殊的 XML 文件，类似于 JSON 一样，可以用于保存一组自定义的数据。Plist 的读写方法请参见第 13 章。
  - 对于一般化的 XML 文件，在各个平台下的处理方式不尽相同。不同平台下有许多可供选择的 XML 库，部分库如表 20-4 所示。
  - 表 20-4　各平台下的 XML 库

语言	库
C	LibXML2
Objective-C	Tree-Based XML 2 和 NSXMLParser
C++	TinyXML
C#	System.XML.Linq
JavaScript	DomParser 和 Microsoft.XMLDOM

  - 在使用 Objective-C 或 C++进行开发的时候，不但可以使用各自语言中的库，也可以使用标准的 C 语言库。由于 C 语言库可以同时在 3 种语言中使用，十分便于移植，因此，LibXML2 也是一个很好的选择。C#与 JavaScript 都有各自提供的 XML 解析器，通常没必要去使用第三方库。
  - 其中，在 JavaScript 中，在非 IE 的引擎中可以使用 DomParser，而在 IE 引擎中可以使用 ActiveX 对象"Microsoft.XMLDOM"。假设已有 XML 字符串 textxml，则可以利用以下代码实现所有浏览器（脚本引擎）兼容的 XML 载入：

```
//JavaScript
var parser, xmlDoc;
if (window.DOMParser) {
    parser = new DOMParser();
    xmlDoc = this.parser.parseFromString(textxml, "text/xml");
}
else {
    xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.async = "false";
    xmlDoc.loadXML(textxml);
}
```

20.2.2　第二阶段：消除平台差异（2）

20.2.2 第二阶段：消除平台差异（2）

- **网络通信**

Cocos2d 系列引擎并没有负责网络通信的模块。如果我们开发的游戏用到了网络通信的功能，就需要使用第三方库，这为我们的移植带来了一些麻烦。

在游戏开发中，许多简单的通信使用 HTTP 实现。HTTP 在各个平台的开发中都是十分常见的，每个平台下都有许多可供我们选择的库来使用。例如，Cocos2d-x 官方推荐使用 libcurl 来实现 HTTP、FTP 等协议的通信。libcurl 使用 C 语言实现，因此可以同时在 C、C++以及 Objective-C 中使用而不必额外移植。在表 20-5 中，我们简单地列举了各个平台下常用的 HTTP 通信库。

表 20-5　各平台下的 HTTP 通信库

语　言	库
C	libcurl
Objective-C	NSUrlConnection
C#	WebClient 和 HTTPWebRequest
JavaScript	WebSocket

HTTP 是无状态的短连接协议，因此适合用于进行简单的通信。而当需要开发实时在线的网络游戏时，就需要使用网络套接字（socket）来保持长连接了。在各个平台下，套接字的实现也不完全相同，然而由于大多数操作系统都支持 POSIX 标准，套接字也是此标准的一部分，因此在 C、C++以及 Objective-C 中，我们都可以使用标准 BSD Socket 函数来实现长连接。在 C#中，套接字的实现位于 System.Net.Sockets 命名空间中，而 JavaScript 则可以使用 WebSocket 来实现长连接。

- **第三方库**

在游戏中，我们经常会使用到第三方提供的广告平台、游戏中心与社交网络等组件，这些 API 通常以闭源库的形式提供给我们。当我们的游戏需要移植到另一个平台时，这些闭源库就必须替换为目标平台上的库了。

替换第三方库是有条件的，那就是我们所使用的库必须在目标平台上有对应的版本，否则替换就无从谈起。根据目标平台的不同以及库所在语言的不同，替换第三方库的难度也会有所不同。替换时可能会有以下 3 种情况，我们一一讨论。

第一种情况，在目标平台上，引擎所使用的语言与库的语言相同，此时库的替换是最轻松的。例如，当我们打算把 Cocos2d-x 移植到 Cocos2d-XNA，游戏中使用的 MD5 算法库恰好又有 C++与 C#两个版本，那么我们只需要把目标游戏的代码中涉及 MD5 算法库的调用替换为 C#版本的代码即可。值得注意的是，在 Windows Phone 7 平台上存在两种应用--Silverlight 与 XNA，两者并不能友好地共存。Cocos2d-XNA 创建的游戏是 XNA 应用，因此基于 Silverlight 的许多广告 API 就不能用于游戏开发。

第二种情况，当我们的目标平台是 Android 时，也许只能找到 Java 版本的库供我们使用，例如许多广告平台都只提供 Android 上的 Java 库。如果我们的游戏是基于 C++开发的，并不能直接使用 Java 上的第三方库。所幸 Android NDK 提供了 Java 到 C++的绑定功能，称为 Java Native Interface（JNI），使得我们可以在 Java 与 C++代码中互相自由调用函数。关于 JNI 技术，限于篇幅，我们在此不做介绍。

第三种情况，当库没有提供目标平台上的实现时，我们就几乎无能为力了。这种情况通常在 JavaScript 语言上出现，因为 JavaScript 仍然是一个十分年轻的新生平台，大多数第三方库并没有提供对 JavaScript 的支持。所幸基于 JavaScript 的 Cocos2d-HTML5 并不是只能运行在浏览器中，当它运行在 Cocos2d-iPhone 或 Cocos2d-x 的脚本引擎中时，如果库提供了 iOS 版本或 Android 版本，则我们可以把库通过脚本引擎的 JavaScript 绑定功能暴露给脚本，从而实现 JavaScript 上的库替换。举一个简单的例子，我们需要把一个使用 AdMob 广告平台的游戏移植到 Cocos2d-HTML5 中，并使用 Cocos2d-x 在移动设备上以脚本的形式运行游戏，则需要以下 3 个步骤。

利用 Android NDK 的 JNI 工具在 C++代码中创建 AdMob 的包装。

- 在 Cocos2d-x 初始化时，利用 JavaScript 绑定技术把 AdMob 在 C++中的包装暴露到脚本引擎中。
- 在 JavaScript 代码中调用 AdMob 的相关函数。

- **其他问题**

- 除了库相关的问题，还存在着其他一些平台相关的问题。
- 众所周知，iOS 采用的是几种固定的分辨率，因此游戏的屏幕尺寸对布局的影响并不是很大，即使不进行自动布局也不算麻烦。然而 Android 以及 Windows Phone 设备众多，屏幕像素可谓五花八门，最新的 iPhone 5 也采用了与上一代手机完全不同的屏幕比例。在这种情况下，我们就需要对游戏的布局系统进行重新设计，以保证游戏可以运行在不同分辨率下。
- 在不同的平台下，文件系统的设计也可能不同。虽然绝大多数现代的移动设备都采用了沙盒（sandbox）的设计来保证系统的安全性，但是各个平台下沙盒的实现风格迥异。好在 Cocos2d-x 对简单的文件读写提供了封装，大多数情况下我们并不用在文件读写的移植上花费很大的精力。

● 20.2.3 第三阶段：优化

- **20.2.3 第三阶段：优化**

- 当我们完成前两个阶段后，游戏已经可以正确地编译运行了。也许此时一个完美的移植成品已经诞生了，不过更大的可能是，我们移植出来的游戏或多或少在性能上还有着一些问题。在我们把游戏从 Cocos2d-iPhone 或 Cocos2d-x 移植到 Cocos2d-XNA 或 Cocos2d-HTML5 上时，性能问题尤其明显。

- **语句优化**

- 在进行第一阶段时，我们逐语句地完成了整个游戏代码的翻译工作，然而逐语句不一定是最优的翻译方式，有时逐语句的翻译会带来

- 性能损失。假设我们把游戏从 Objective-C 移植到 C#，下面的语句
-
- ```
  //Objective-C
  NSString* strLevel = [NSString stringWithFormat:@"%d", level];
  ```

- 将被翻译为

- ```
  //C#
  string strLevel = string.Format("{0}", level);
  ```

- 其中 level 是一个整数。这显然不如直接把 level 转换为字符串迅速，因此上面的翻译不如改为：

- ```
  //C#
  string strLevel = level.ToString();
  ```

- 逐语句翻译的结果通常会包含许多冗余的运算，在对性能要求较高的情况下，我们可以找出代码中的冗余并加以消除。

- **垃圾回收**

- Cocos2d-iPhone 采用 Objective-C 作为基础语言，Cocos2d-x 采用 C++作为基础语言，这两种语言与 C#、JavaScript 最大的区别之一就是前者不支持垃圾回收器，需要我们手动管理内存，而后者内存管理的任务则由垃圾回收器代劳，开发者不需要任何干预。然而垃圾回收器会带来一系列更为复杂的问题，为游戏的性能损失埋下隐患。
- 垃圾回收器的工作原理十分复杂，为了实现正确的垃圾回收，每一次进行回收操作都会花费相当大的开销。垃圾回收器在内存过低以及对象数量过高的时候就会开始回收处理。因此，一方面需要尽可能地降低游戏的内存开销，另一方面需要尽
  ```

可能地减少创建对象的数量。无论在何种平台下，降低内存开销都是开发者十分重视的一个环节，因此通常都能做到内存开销的最小化。在内存开销无法降低的情况下，就应该考虑能否降低对象的数目，以避免垃圾回收的开销。

- 因此，在拥有垃圾回收机制的语言中，我们需要遵循如下两个原则。
- 尽可能地重用旧对象，而不是创建新对象。在本书第三部分中，我们详细介绍了各种性能优化技术，其中对象池就可以实现对象的重用，可以极大地降低垃圾回收的开销。
- 尽可能避免大量创建新对象。例如，在 C#中为了创建一个长字符串，如果反复使用许多次连接操作，则会创建大量的临时对象，给垃圾回收带来很大的负担，而采用 StringBuilder 对象直接在缓冲区中进行操作，可以避免在连接字符串时产生的额外浪费。

- **纹理优化**

- 在低端设备上，即使我们把游戏完美地移植了过去，也设法把 CPU 与内存开销降到了最低，但是仍然不能流畅地运行游戏，此时就只能靠牺牲游戏的画面品质来提高游戏性能了。纹理优化就是最常用的手段之一，把纹理文件的压缩品质降低，并选择位数较少的像素格式，虽然牺牲了画质，却可以节省大量内存开销。
- 这部分内容其实并不属于移植章节，请读者参考本书第 10 章的相关内容，其中包含了纹理优化的详细内容。

20.3 小结

- 20.3 小结

- 在本章中，我们介绍了 Cocos2d 几个主流版本之间的异同以及移植的可能性和方法。可以看到，Cocos2d 主流版本几乎都源自于同一血统--Cocos2d-iPhone。也正是如此，我们可以较为轻松地在 Cocos2d 的几个主流版本之间相互移植游戏，而不用花费很大的开销。
- 本章我们着重介绍了 Cocos2d 系列引擎的命名原则，通过命名原则的对比，可以确定出移植的主要工作重点在何处。下面总结游戏移植的重点工作。
- 代码移植：逐行翻译代码。在这个过程中，可以采取自顶向下的移植方式或自底向上的移植方式。
- 消除平台差异：平台的差异仍然无法有效解决，因此需要留意语言库、第三方库在不同平台上的差异。
- 优化：自动化工具移植的代码也许有优化的余地，在优化阶段需要检查并修正无意义的代码。可以考虑不同平台对纹理的承载能力，必要时需要降低纹理品质以保证游戏的流畅。最后，运行在 C#等支持垃圾回收的语言上的游戏，需要额外注意垃圾回收是否保持高效工作。

21.1 开发前的准备

- **实战演练--开发自己的《捕鱼达人》**

- 经过前面章节的学习，相信大家对 Cocos2d-x 引擎有了一个整体的了解，但纸上得来终觉浅。在实际的开发过程中，还有很多需要我们思考的游戏设计概念，也将面临许多取舍。因此，接下来我们将通过一个练习章节，更加深入地介绍 Cocos2d-x 的实际应用以及开发技巧。
- 在本章中，读者可以学到的内容有：
- Cocos2d-x 引擎内部的部分机制和设计模式；
- Cocos2d-x 开发的常见问题及其实现方式；
- 游戏程序开发的基础知识。

以上内容我们将以实例的形式体现，读者可以在代码中细细体会 Cocos2d-x 开发的精髓。在图灵社区（www.ituring.com.cn）的本书主页可以下载到本章所需的全部资源，包括图片、音乐、音效资源以及源代码。这里我们使用的引擎是 Cocos2d-x 2.0.3。我们同时分发了在 Visual Studio 2008、Xcode 和 Eclipse 三个编译器下的项目，分别对应 Windows、iOS 和 Android 三个不同的系统。

对应读者各自的编译器，打开项目并编译运行。初次编译的时间可能会稍长，如果没有意外，则可以看到即将完成的游戏--Fishing Joy（如图 21-1 所示）。它包括了载入页面和游戏场景，在游戏场景中单击屏幕可以更换和发射子弹，也可以进入暂停页面对游戏的基本选项进行设置。Fishing Joy 涵盖了几乎游戏所有的基本元素，只要掌握好如何运用 Cocos2d-x 实现这些功能，就能对大部分游戏的开发得心应手。看到屏幕上游动的鱼儿和呼啸而过的子弹，你是不是已经跃跃欲试了呢？接下来我们就开始着手完成 Fishing Joy。

图21-1 Fishing Joy效果图

21.1 开发前的准备

对有经验的开发者而言，在项目开始之前，都会对策划和需求进行分析建模，把描述性的文档转换成开发模型，并对诸如命名规则等的标准进行统一。这在团队开发中尤为重要。虽然接下来的是独立开发，但仍有必要确立唯一的标准。

这里我们将使用驼峰命名法以及 MVC 模式进行开发，并且在开发过程中将代码可读性作为首要的考虑因素。

MVC 模式是软件工程中的一种软件架构模式，包括模型（Model）、视图（View）和控制器（Controller）3 部分。模型包括了数据和逻辑等，并不关心界面上的显示形式。视图则不包含程序上的逻辑，仅监控模型中的数据，并根据数据的内容，将其在界面上显示出来。控制器则起到不同层面间的组织作用，用于控制应用程序的流程及消息传递。根据程序的复杂程度，视图和模型的分离程度可能也有所不同。在此我们不进行更深层次的讨论，只在接下来的内容中说明如何在 Cocos2d-x 的开发中运用这种模式。

21.1.1 视图

21.1.1 视图

游戏中的视图包括了 CCScene 下所有可见的节点和元素。在 Fishing Joy 中存在两个场景：开始场景和游戏场景。

开始场景用于预加载资源，这是游戏开发中常用的做法。在游戏开始之前，将外部资源加载进缓存中来保证游戏过程中的流畅性。在 Cocos2d-x 中，共有 4 个缓存区，它们是 CCTextureCache、CCSpriteFrameCache、CCAnimationCache 和 CCShaderCache，分别对应图片纹理、精灵、动画和着色器。我们需要做的是调用这些缓存提供的接口，异步载入资源，并

以进度条的形式实时提示玩家当前的载入进度。载入完成之后方可进入下一场景。因此，其界面并不复杂，除了简单的背景和标题之外，仅有一个需要更新的进度条。

- 游戏场景是整个游戏的核心，包含了大部分的游戏逻辑和数据，也是游戏可玩性的体现。其界面比较复杂，大部分可见元素都需要根据数据的变化实时更新。在开发的初期，我们应着重完成游戏场景，制作出游戏的雏形，而不是把时间花费在菜单、载入界面等与游戏相关性较弱的地方。游戏场景期望的效果如图 21-2 所示。

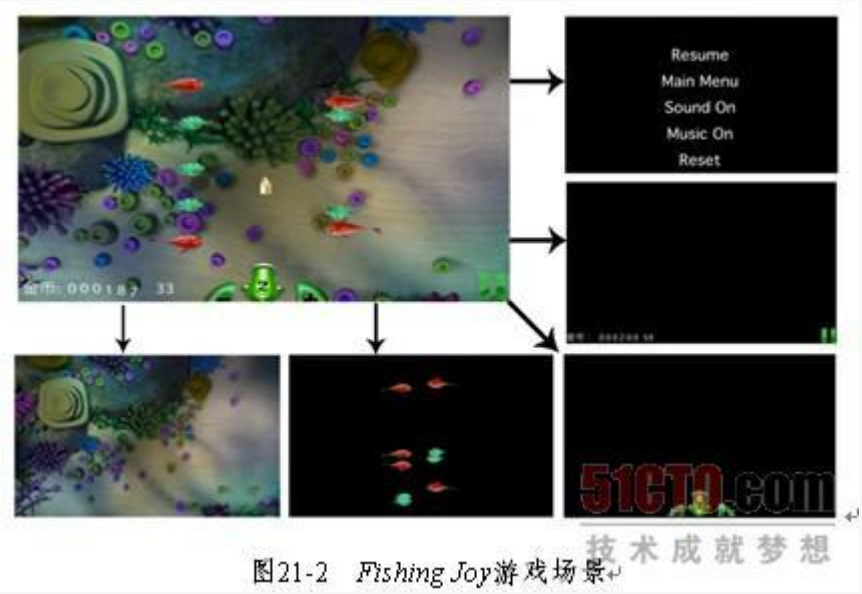


图21-2 *Fishing Joy*游戏场景

- 21.1.2 模型

- 21.1.2 模型

- 这部分主要包括了游戏逻辑数据。在之前的介绍中，我们已经知道了可以通过 CCUser
- Default、XML、JSON、SQLite 或其他方法对游戏数据进行持久化的存储。这些方法的技术难度和功能都有其不同的倾向，例如，SQLite 比起 XML，更倾向于大型数据的处理。我们需要根据游戏中的需求慎重选择对应的技术。
- Fishing Joy 中的数据主要分为两部分：静态数据和动态数据。

◆ **静态数据**

- 静态数据是程序中的只读数据，例如图片名、NPC 的名字、道具的售价甚至是图片的坐标等。这些数据不会在游戏过程中发生改变，然而，在开发过程中它们却可能经常变动。为了便于修改，一个常规的做法是把这些数据放到外部文件进行保存，杜绝硬编码。
- 我们选择使用 plist（property list）保存静态数据，并使用 plist 编辑工具进行必要的修改。在程序中，通过调用

```
CCDictionary* dic = CCDictionary::createWithContentsOfFile(const char *pFileName);
```

- 读取文件，并以字典的形式存储在缓存中。
- 提示　对于静态数据的存储，我们还有许多备选方案，例如，使用 CocosBuilder 存储精灵的坐
- 标、角度和比例等信息，亦或是使用 PhysicsEditor 存储物理游戏中刚体的参数等。有兴趣的读者不妨进一步探索方便我们开发的工具。

◆ **动态数据**

- 相对于静态数据，这部分数据在游戏过程中会频繁变动，例如玩家的金币数和经验值等。所以，我们需要选择便于在程序中读写的数据保存形式。
- 对于 Fishing Joy 而言，动态数据有玩家的金币数以及音乐音效的大小。这些数据都是线性结构，没有嵌套关系，而且数据量也很小，因此我们使用 Cocos2d 已封装好的 CCUserDefault 即可。
- 提示　也许有些读者会觉得使用 SQLite、JSON 或者其他一些技术看上去显得更酷一些，但在实际开发过程中，慎重选取更简洁、更符合需求的技术，往往能更好地降低开发的人力和时间成本，也能使程序的效率得到提高。

● 21.1.3 控制器

- 21.1.3 控制器

- **控制器主要负责模型和视图之间的信息传递。在 Cocos2d 的定义中，CCScene 作为场景内所有内容的载体出现，而其本身并不直接在界面上显示。这无疑是最好的控制器。我们可以在这里进行数据与场景、场景与场景之间的交互，即将 CCScene 类的对象作为逻辑控制和信息传递的"总司令部"。**

21.2.1 第一轮迭代（1）

- 21.2 开始开发
- 经过简单的分析，我们可以开始着手开发了。这里将采用增量开发的方式，每一轮迭代完成其中一部分功能，同时保持其可扩展性。功能的复杂性逐轮递增，通过三轮的迭代完成整个项目。
- 21.2.1 第一轮迭代（1）
- 打开编译器，新建一个项目，命名为 Fishing Joy，并将随书提供的捕鱼资源包下的所有资源导入项目中。

- **创建数据**

- 由于场景是要适配于数据的，所以我们先完成动态数据 FishingJoyData 和静态数据 StaticData 两个类。根据之前的分析可以知道，这两个类在整个游戏过程中只可能出现一个实例，并且贯穿着整个游戏过程。因此，我们不妨把它们设计为单例。
- 单例（singleton）是软件开发过程中常用的一个设计模式。它通过控制一个全局变量，以便拥有某些独占性资源，并达到便于管理和访问的作用，但是其缺点也很致命。它在整个程序运行周期内几乎不会被释放。同时，滥用单例会使代码耦合性增大，不易于扩展。因此，单例模式一直备受争议。
- Cocos2d-x 是一个很好地运用了单例的例子。在引擎内部，有许多单例，它们都以 shared 为开头出现，例如：

```
CCDirector* sharedDirector = CCDirector::sharedDirector();
CCTextureCache* sharedTextureCache = CCTextureCache::sharedTextureCache();
```

- 仿照 Cocos2d-x 的做法，我们也同样设计这两个数据类单例，即实现以下功能。
- 将类的构造函数、析构函数和初始化函数都设为私有，禁止外部创建的新的实例。
- 统一使用 shared*接口访问唯一的实例，并在该接口中使用延迟初始化，在需要时才创建实例。
- 实现 purge 函数，在必要时调用，以清理不必要的内存。
- 最后别忘了，在 Cocos2d-x 中，所有的对象类都必须继承自 CCObject 或其子类。FishingJoyData 类的代码如下所示：

```
//FishingJoyData.h
class FishingJoyData : public cocos2d::CCObject
{
```

```
public:
    static FishingJoyData* sharedFishingJoyData();
    void purge();
protected:
    FishingJoyData();
    ~FishingJoyData();
    bool init();
};
//FishingJoyData.cpp
static FishingJoyData* _sharedFishingJoyData = NULL;
FishingJoyData* FishingJoyData::sharedFishingJoyData()
{
    if(_sharedFishingJoyData==NULL){
        _sharedFishingJoyData = new FishingJoyData();
        _sharedFishingJoyData->init();
    }
    return _sharedFishingJoyData;
}
void FishingJoyData::purge()
{
    CC_SAFE_RELEASE_NULL(_sharedFishingJoyData);
}
//StaticData.h
class StaticData : public cocos2d::CCObject
{
public:
    static StaticData* sharedStaticData();
    void purge();
private:
    StaticData();
    ~StaticData();
    bool init();
};
//StaticData.cpp
static StaticData* _sharedStaticData = NULL;
StaticData* StaticData::sharedStaticData()
{
    if(_sharedStaticData == NULL){
        _sharedStaticData = new StaticData();
        _sharedStaticData->init();
    }
    return _sharedStaticData;
}
void StaticData::purge()
{
```

```
    CC_SAFE_RELEASE_NULL(_sharedStaticData);
}
```

21.2.1 第一轮迭代（2）

21.2.1 第一轮迭代（2）

多亏我们在项目前的思考和设计，接下来的工作变得轻松了许多。对于静态数据， 我们添加一个新的成员变量：

```
//StaticData.h
class StaticData : public cocos2d::CCObject
{
    ...
protected:
    cocos2d::CCDictionary* _dictionary;
};
```

在其 init 方法中，调用 CCDictionary 的接口，读取"static_data.plist"文件，保存在缓存中。通过把 plist 读取到 CCDictionary 后，我们得到是一个键为 string 类型、值为 CCObject 对象的字典集。具体来说，值类型有 3 种：CCString、CCArray 和 CCDictionary。对于简单的键值对，我们使用 CCString 来保存数据（包括字符串类型与数值类型），而对于数组与字典，则采用后两种类型来保存。

CCString 是自 Cocos2d-x 1.0 版本开始出现的一个类，用于保存字符串，但那时其功能并不完善。在 2.0 版本中，CCString 已经能够实现 Objective-C 中 NSString 的大部分功能。为了取出我们所需类型的值，可以调用以下接口进行转换：

```
//CCString.h
...
    int intValue() const;
    unsigned int uintValue() const;
    float floatValue() const;
    double doubleValue() const
    bool boolValue() const;
    const char* getCString() const;
...
```

对应 Cocos2d-x 定义的 3 个基础的结构 CCPoint、CCSize 和 CCRect，也可以通过以下接口进行创建：

```
//CCNS.h
...
CCRect CC_DLL CCRectFromString(const char* pszContent);
CCPoint CC_DLL CCPointFromString(const char* pszContent);
CCSize CC_DLL CCSizeFromString(const char* pszContent);
...
```

有了这些接口，我们就可以轻松地获得 plist 的数据，并转换成我们想要的类型了。为了更快速地写代码，我们还定义了一系列接口将重复性的操作进行封装，同时用宏定义，使代码变得更加简洁：

```
//StaticData.h
#define STATIC_DATA_STRING(key) StaticData::sharedStaticData()->stringFromKey(key)
#define STATIC_DATA_INT(key) StaticData::sharedStaticData()->intFromKey(key)
#define STATIC_DATA_FLOAT(key) StaticData::sharedStaticData()->floatFromKey(key)
#define STATIC_DATA_BOOLEAN(key) StaticData::sharedStaticData()->booleanFromKey(key)
```

```
#define STATIC_DATA_POINT(key) StaticData::sharedStaticData()->pointFromKey(key)
#define STATIC_DATA_RECT(key) StaticData::sharedStaticData()->rectFromKey(key)
#define STATIC_DATA_SIZE(key) StaticData::sharedStaticData()->sizeFromKey(key)
...
public:
    const char* stringFromKey(std::string key);
    int intFromKey(std::string key);
    float floatFromKey(std::string key);
    bool booleanFromKey(std::string key);
    cocos2d::CCPoint pointFromKey(std::string key);
    cocos2d::CCRect rectFromKey(std::string key);
    cocos2d::CCSize sizeFromKey(std::string key);
...
```

- 对于动态数据，我们需要实现一个 flush 函数，用于将数据存储到外部资源中。
- 大多数游戏对于初学者都会有相应的新手引导，因此，除了金币数、音乐音效和音量大小外，我们还需要一个布尔值，用于判断玩家是否为初学者。对于 Fishing Joy 而言，需要对新加入的玩家的游戏数据进行初始化，相关代码如下所示：

```
//FishingJoyData.cpp
bool FishingJoyData::init()
{
    _isBeginner = CCUserDefault::sharedUserDefault()->getBoolForKey("beginner",true);
    if(_isBeginner == true){
        this->reset();
        this->flush();
        this->setIsBeginner(false);
    }else{
        _isBeginner = CCUserDefault::sharedUserDefault()->getBoolForKey("beginner");
        _soundVolume = CCUserDefault::sharedUserDefault()->getFloatForKey("sound");
        _musicVolume = CCUserDefault::sharedUserDefault()->getFloatForKey("music");
        _gold = CCUserDefault::sharedUserDefault()->getIntegerForKey("gold");
        CCUserDefault::sharedUserDefault()->purgeSharedUserDefault();
    }
    return true;
}
void FishingJoyData::flush()
{
    CCUserDefault::sharedUserDefault()->setFloatForKey("sound", this->getSoundVolume());
    CCUserDefault::sharedUserDefault()->setBoolForKey("beginner", this->getIsBeginner());
    CCUserDefault::sharedUserDefault()->setIntegerForKey("gold", this->getGold());
    CCUserDefault::sharedUserDefault()->setFloatForKey("music", this->getMusicVolume());
    CCUserDefault::sharedUserDefault()->flush();
    CCUserDefault::sharedUserDefault()->purgeSharedUserDefault();
}
```

- 可以看到，FishingJoyData 其实就是对 CCUserDefault 相关操作的封装。

- 21.2.1 第一轮迭代（3）

■ 21.2.1 第一轮迭代（3）

■ 在 CCUserDefault 的 get*ForKey 系列方法中，有两个参数：一个是字符串类型的键值，另一个是默认参数，它表示找不到对应键值时返回的值。

■ 细心的读者应该注意到了，在 FishingJoyData 类中，每次对 CCUserDefault 进行操作后，都会调用 purgeSharedUserDefault() 将其删除。这是因为 FishingJoyData 和 CCUserDefault 中都保存着游戏的数据。因此，只在与外部文件交互时，我们才打开 CCUserDefault，保存数据，然后删除 CCUserDefault，以此避免重复占用游戏数据，而对内存造成浪费。

■ 最后，别忘了在程序进入后台时调用 flush 方法，及时保存游戏数据。还记得程序何时进入后台么？不妨回到第 2 章好好复习一下。

- **创建游戏场景**

■ 完成了游戏数据的封装，读者是不是觉得有些枯燥无味呢？尽管数据操作并不复杂，但没有看得见的图片和效果，总是让人兴奋不起来的。有了数据的铺垫，接下来我们制作场景的过程会更加得心应手。

■ 删除项目创建时自带的"HelloWorld.h（.cpp）"文件，创建继承自 CCScene 的 GameScene 类。然后用 TexturePacker 将碎图资源压缩成一张整图，重新导入项目中，并在 GameScene 中载入，方便我们以后的操作，相关代码如下所示：

```
//GameScene.h
#include "cocos2d.h"
class GameScene : public cocos2d::CCScene
{
public:
    CREATE_FUNC(GameScene);
    bool init();
};
```

■ 提示 TexturePacker 是一款付费软件，免费版生成的整图可能会出现部分标签纹理影响效

■ 果。读者可以使用图片资源中已经生成好的整图，直接导入项目中，也可以使用其他免费的纹理打包软件，但请绝对不要使用破解的盗版软件。作为开发者，我们更加有义务维护版权。

■ CREATE_FUNC 是 Cocos2d-x 定义的一个宏。这里我们不妨转到它的定义：

```
//CCPlatformMacros.h
/**
 * define a create function for a specific type, such as CCLayer
 * @__TYPE__ class type to add create(), such as CCLayer
 */
#define CREATE_FUNC(__TYPE__) \
static __TYPE__* create() \
{ \
    __TYPE__ *pRet = new __TYPE__(); \
    if (pRet && pRet->init()) \
    { \
        pRet->autorelease(); \
        return pRet; \
    } \
```

```
    else \
    { \
        delete pRet; \
        pRet = NULL; \
        return NULL; \
    } \
}
```

- 可以看到，CREATE_FUNC 创建并实现了一个 create 静态方法。这是使用频率很高的一段代码，对于类似的重复性工作，Cocos2d-x 都提供了一系列宏，用于减轻开发者的工作量。Cocos2d-x 还是很人性化的，当遇到不认识的宏时，都可以在源代码中查看其含义。这也正是开源引擎的一大优势。
- 对应之前的 GameScene 场景的层次分离图，用该方法再创建 BackgroundLayer、FishLayer、CannonLayer、PanelLayer 和 MenuLayer5 个类，不同的是，它们都继承自 CCLayer。然后再在 GameScene 中添加它们，相关代码如下：

```
//GameScene.cpp
bool GameScene::init()
{
    if(CCScene::init()){
        _backgroundLayer = BackgroundLayer::create();
        this->addChild(_backgroundLayer);
        _fishLayer = FishLayer::create();
        this->addChild(_fishLayer);
        _cannonLayer = CannonLayer::create();
        this->addChild(_cannonLayer);
        _panelLayer = PanelLayer::create();
        this->addChild(_panelLayer);
        _menuLayer = MenuLayer::create();
        CC_SAFE_RETAIN(_menuLayer);
        return true;
    }
    return false;
}
GameScene::~GameScene()
{
    CC_SAFE_RELEASE(_menuLayer);
}
```

21.2.1 第一轮迭代（4）

21.2.1 第一轮迭代（4）

- 这里我们要特别注意_menuLayer。由于菜单界面只在暂停的时候才出现，所以我们并不是直接将其添加进场景中，而是调用了 retain 方法将其保留下来，方便日后访问。同时，别忘了在使用 retain 之后，调用对应的 release 方法，保证内存不会泄露。
- 现在场景中已经有了 5 个层，但是层里面空空如也，即使编译运行，也只能得到漆黑一片的界面。接下来，就将这些层逐一丰富起来。
- BackgroundLayer。背景恐怕是场景中最简单的一个层了。我们只需调用

- CCSprite::createWithSpriteFrameName(STATIC_DATA_STRING("background"));
- 从 CCSpriteFrameCache 中读取精灵，并添加到场景正中即可。
- 得益于 StaticData 的封装，我们通过一个键值就可以轻松获取背景图片的名称了。
- 此时编译运行一下，深海的背景就出现在场景中了（见图 21-3）。

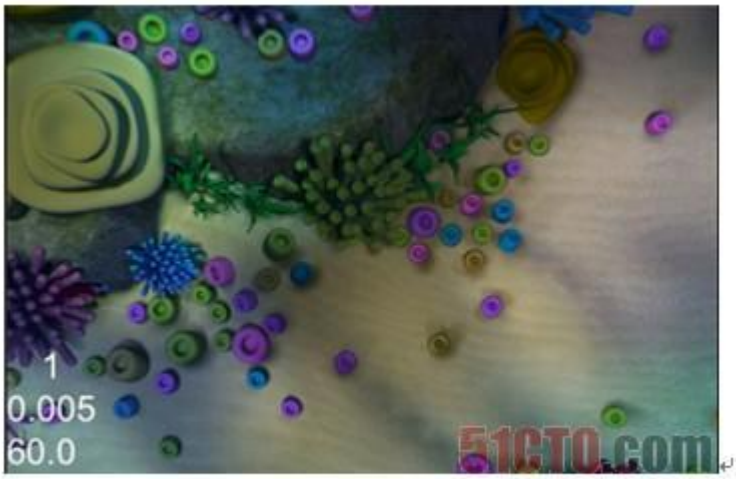


图21-3 运行效果

- FishLayer。该层中包含的唯一元素是带有动画效果的鱼，因此，在创建 Fish 类之前，我们先回到 GameScene 预载入一段鱼的动画。在"GameScene.cpp"中添加 preloadResources 方法，并在 init 中调用，根据两种类型的鱼，创建两段不同的动画，并载入缓存 CCAnimationCache 中。创建动画的方式在前面已经介绍过了，此处不再赘述。"GameScene.cpp"文件的部分代码如下所示：

```
//GameScene.cpp
...
//todo 预载入资源，实现 StartScene 后将其删除
void GameScene::preloadResources()
{
    CCSpriteFrameCache::sharedSpriteFrameCache()
        ->addSpriteFramesWithFile("fishingjoy_resource.plist");


    int frameCount = STATIC_DATA_INT("fish_frame_count");
    for (int type = k_Fish_Type_Red; type &lt; k_Fish_Type_Count; type++) {
        CCAnimation* fishAnimation = CCAnimation::create();
        for(int i = 0;i &lt; frameCount;i++){
            fishAnimation-&gt;addSpriteFrameWithFileName
                (CCString::createWithFormat(STATIC_DATA_STRING
                ("fish_frame_name_format"),type,i)-&gt;getCString());
        }
        fishAnimation-&gt;setDelayPerUnit
            (STATIC_DATA_FLOAT("fish_frame_delay"));
        CCString* animationName = CCString::createWithFormat
            (STATIC_DATA_STRING("fish_animation"), type);
```

```
        CCAnimationCache::sharedAnimationCache()-&gt;addAnimation
            (fishAnimation, animationName-&gt;getCString());
    }


    FishingJoyData::sharedFishingJoyData();
    PersonalAudioEngine::sharedEngine();


}
```

- 这样我们就创建了两个不同的动画帧序列，在使用的时候只需调用

```
CCAnimation* animation = CCAnimationCache::sharedAnimationCache()
    ->animationByName(const char *name);
```

- 即可获取相应的动画。
- 此外，我们还在函数中加了"//todo"字样的注释。由于预载入资源是 StartScene 的工作，为了快速开发出游戏原型，我们让 GameScene 暂时完成了这部分工作。在后期的开发中，可以通过编译器的搜索功能找到"//todo"标签，定位到此处进行重构。
- 有了动画效果，完成 Fish 类就不在话下了，但是这里有两点要特别注意。

● 21.2.1 第一轮迭代（5）

21.2.1 第一轮迭代（5）

- CCAnimation 只是用于存放动画，真正能让动画播放起来的是动作类 CCAnimate。
- CCAnimate 只能由 CCSprite 及其子类播放。
- 在精灵 sprite 上播放一个动画 animation 的代码如下：

```
CCAnimate* animate = CCAnimate::create(animation);
CCSprite* sprite = CCSprite::create();
sprite->runAction(animate);
```

- 此外，我们还需要使用 CCRepeatForever 对 CCAnimate 进行复合，达到无限循环播放的效果。这样，我们就得到了 Fish 类的代码：

```
//Fish.cpp
bool Fish::init(FishType type)
{
    _type = type;
    CCString* animationName = CCString::createWithFormat
        (STATIC_DATA_STRING("fish_animation"), _type);
    CCAnimation* fishAnimation = CCAnimationCache::sharedAnimationCache()
        ->animationByName(animationName->getCString());
    CCAnimate* fishAnimate = CCAnimate::create(fishAnimation);
    fishAnimate->setTag(k_Action_Animate);
```

```
    _fishSprite = CCSprite::create();
    this->addChild(_fishSprite);
    _fishSprite->runAction(CCRepeatForever::create(fishAnimate));
    return true;
}
```

- 接下来，再将 Fish 添加到场景中。这里会再次用到预加载技术。对于游戏中大量出现的物体，例如子弹、一定时间后生成的怪物等，通常会在游戏开始之前预先生成一部分并保留在内存中，在需要用到该物体时再将其添加到场景中，从而避免在游戏过程中大量申请内存空间而给玩家造成不流畅的感觉。
- 因此，在 FishLayer 中，我们先用 CCArray* _fishes 存储预先生成的鱼，并通过 schedule 计时器，每隔一段时间调用一次 addFish 方法，将未载入场景的鱼添加进去，相关代码如下：
-

```
//FishLayer.cpp
bool FishLayer::init()
{
    if(CCLayer::init()){
        int capacity = 50;
        _fishes = CCArray::createWithCapacity(capacity);
        CC_SAFE_RETAIN(_fishes);

        for(int i = 0;i < capacity;i++){
            int type = CCRANDOM_0_1()*2;
            Fish* fish = Fish::create((FishType)type);
            _fishes->addObject(fish);
        }
        this->schedule(schedule_selector(FishLayer::addFish), 3.0);
        return true;
    }
    return false;
}
void FishLayer::addFish()
{
    CCSize winSize = CCDirector::sharedDirector()->getWinSize();
    int countToAdd = CCRANDOM_0_1() * 10 + 1;
    int countHasAdded = 0;
    CCObject* iterator;
    CCARRAY_FOREACH(_fishes,  iterator){
        Fish* fish = (Fish*)iterator;
        if(fish->isRunning() == false){
            this->addChild(fish);
            //todo 后期应重设 Fish 产生时的随机坐标
            int randomX = CCRANDOM_0_1() * winSize.width;
            int randomY = CCRANDOM_0_1() * winSize.height;
            fish->setPosition(CCPointMake(randomX, randomY));
```

```
                countHasAdded++;
                if(countToAdd == countHasAdded){
                    break;
                }
            }
        }
    }
```


21.2.1 第一轮迭代（6）

对于鱼是否在场景中的判定，我们使用了 isRunning()。它是 CCNode 中定义的一个属性，只在 CCNode::onEnter()中才设置为 true。

再次编译运行，得到的效果如图 21-4 所示，可以发现原地摆着尾巴的鱼儿就出现在眼前了。读者也许并不满意这些死气沉沉的鱼儿，但先别急，第一轮开发我们只先实现到这里。有了这些鱼儿作为基础，后面的开发将变得很简单。

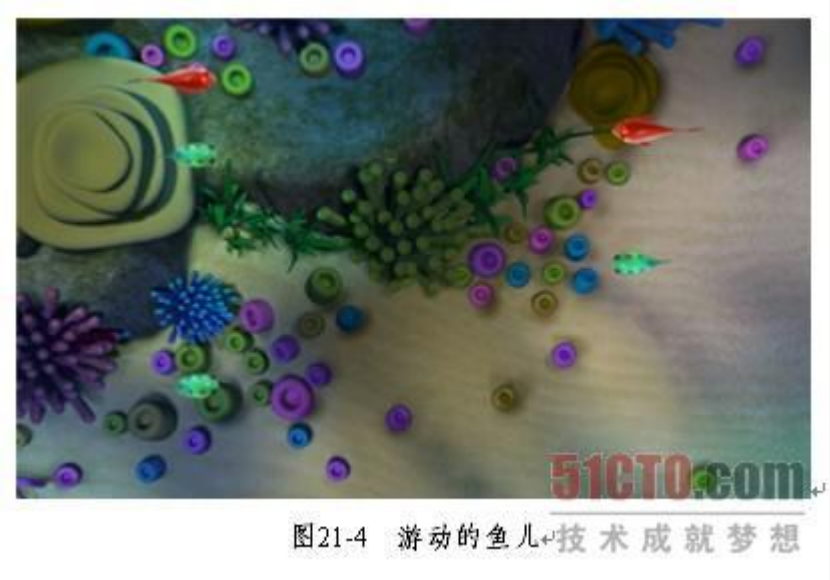


图21-4 游动的鱼儿

CannonLayer。在炮台层中，一共有如下 3 个物体：

炮台及其控制按钮

子弹

网

我们为其各创建一个类，并载入相应的图片资源，其中炮台能够控制子弹的发射，子弹击中目标后会展开网。许多开发者可能会按照思维定势，直接将子弹当做炮台的子节点，并且把网作为子弹的子节点。事实上，这样会导致场景中的层次过多，信息传递变得麻烦，坐标的转换和旋转运算也变得复杂。因此，此处应该额外添加一个 Weapon 类对它们进行控制。

设计好的 Weapon 类如图 21-5 所示，我们把大部分接口暂时留空，等待实现。

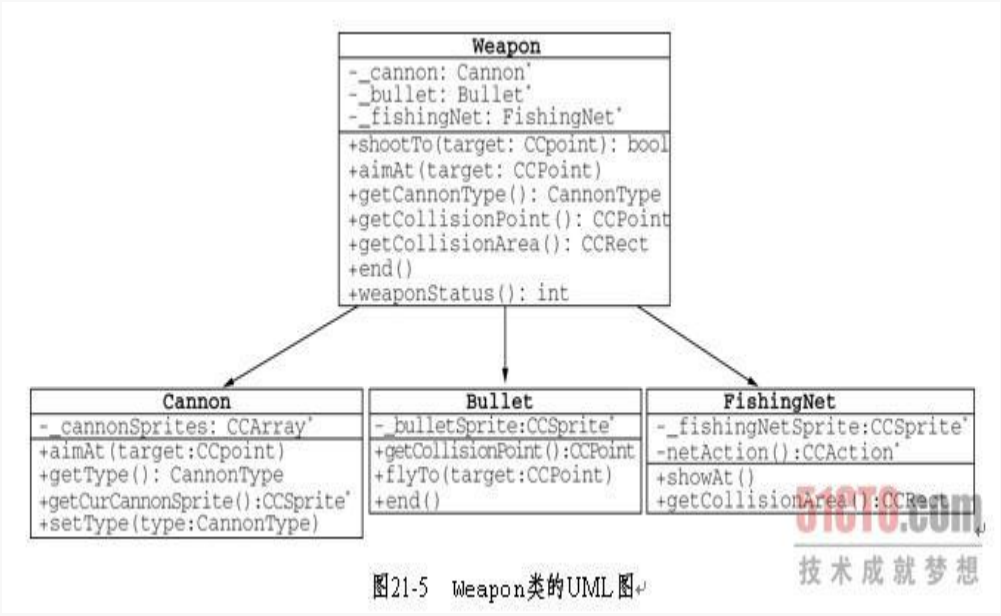


图21-5 Weapon类的UML图

- 在 CannonLayer 中，我们注册了 switchCannon 函数作为两个按钮的回调函数，用以改变炮台的等级。同时，在 Cannon 类中，我们保留了对两个不同等级炮台的精灵的引用，调用 setType 方法将转换到别的类型，并添加到场景中。下面是相关的代码：

```
//Cannon.cpp
void Cannon::setType(CannonType type)
{
    if(_type != type){
        if(type >= k_Cannon_Type_Count){
            type = k_Cannon_Type_1;
        }else if(type < k_Cannon_Type_1){
            type = (CannonType)(k_Cannon_Type_Count-1);
        }
        this->removeChildByTag(_type, false);
        CCSprite* newCannonSprite =
            (CCSprite*)_cannonSprites->objectAtIndex(type);
        this->addChild(newCannonSprite, 0, type);
        _type = type;
    }
}
```

- 初次调用 setType 函数时，removeChildByTag 所需要移除的节点不存在，但我们并不需要担心可能会出现内存错误。Cocos2d-x 是一个比较成熟的游戏引擎，经过了很多商业游戏的考验，在各种错误处理上也都考虑得很周全。

21.2.1 第一轮迭代（7）

21.2.1 第一轮迭代（7）

- 编译运行程序，得到的效果如图 21-6 所示。尽管游戏变得更丰富了，而且也有了两个可以交互的按钮，但还是无法控制炮弹，我们将在后期修复这个问题。

图21-6 运行效果

- PanelLayer。该层中包含了两个重要的组件（component），分别是计时器和展示版。
- 计时器是许多游戏中都会出现的组件。在倒计时结束后执行一些特别的操作，如增加金币等。展示版会在改变内容时，如改变金币数目等，有一个类似老虎机一样的滚动改变效果。这两者在游戏中都是很普遍的组件，我们需要对它们进行更完整的封装以提高重用性。这样即使在别的游戏开发过程中需要这种效果，也可以使用这两个组件。对于这两者的实现，需要耗费一定的时间，但它们不影响游戏的进行。因此，我们在第一轮迭代时，暂且忽略它们。更多关于组件的编写，将在后面集中讨论。
- 在 PanelLayer 中，只添加一个暂停按钮，用于弹出 MenuLayer。我们先在 GameScene 中声明

```
void pause();
```

- 然后在暂停按钮所注册的函数中获取 GameScene，并调用 gameScene->pause()载入菜单页面，相关代码如下：

```
//PanelLayer.cpp
void PanelLayer::pause()
{
    GameScene* gameScene = (GameScene*)this->getParent();
    gameScene->pause();
}
```

- 读者是否觉得传递到 GameScene 中执行的做法很"傻"呢？事实上，关闭触摸、暂停音乐音效、暂停动作和计时器、添加菜单层等的暂停操作，对于 PanelLayer 而言，需要的权限太高了。贸然执行会使 PanelLayer 的耦合性大大增加，导致代码结构混乱，维护成本增高。相反，由 GameScene 执行暂停操作就可以避免这些问题。这正是遵循 MVC 模型，由控制器集中处理事件的优点。图 21-7 展示了 PanelLayer 在两种设计下的耦合情况。

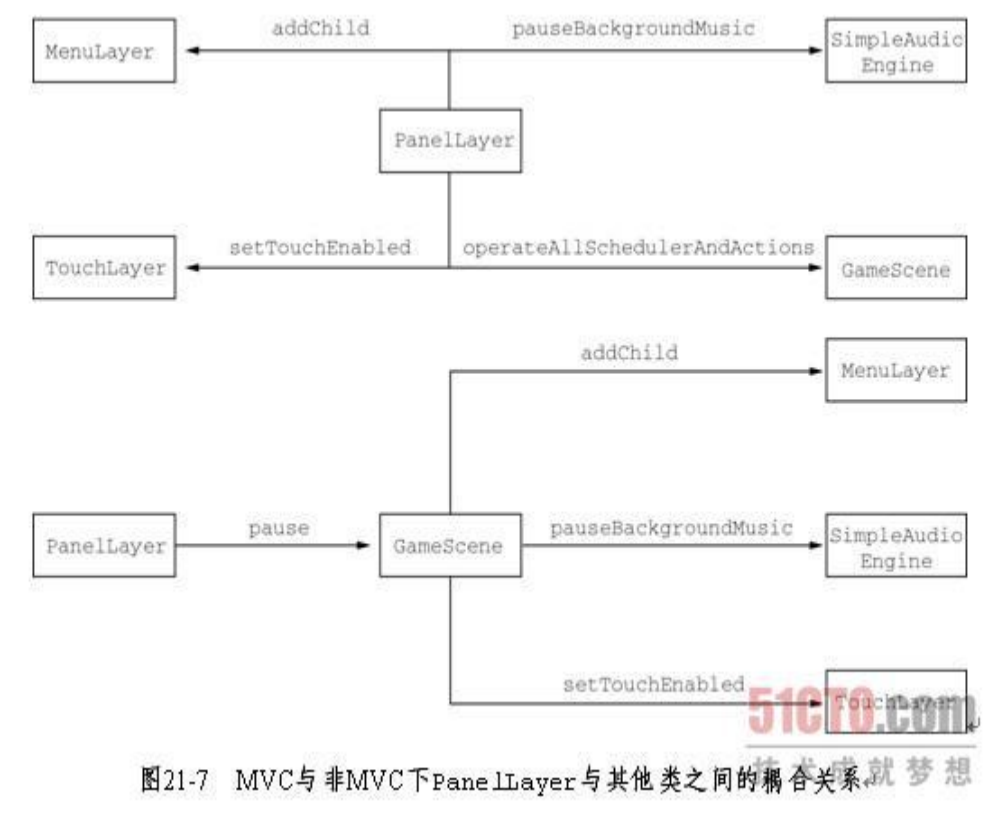


图21-7 MVC与非MVC下PanelLayer与其他类之间的耦合关系

- 关于暂停的实现， 很多 Cocos2d-x 的使用者可能会直接调用

```
CCDirector::sharedDirector()->pause();
```

- 事实上，这是对游戏的全局暂停，通常在游戏进入不活跃状态时使用（即 void AppDelegate::applicationDidEnterBackground()）。调用之后会连同菜单层也进入暂停状态，显然这不是我们想要的。
- 在游戏中，所谓的暂停其实是某些需要停下来的元素"刚好"都静止了。对于不同的游戏，需要的暂停是不一样的，例如某些 3D 游戏暂停时，人物还会在原地摆弄盔甲。因此，我们需要对暂停进行定制。

21.2.1 第一轮迭代（8）

21.2.1 第一轮迭代（8）

- 在 Fishing Joy 中，需要暂停的是主场景中的动作和计时器。Cocos2d-x 已经提供了这个操作的函数:

```
void CCNode::pauseSchedulerAndActions()
```

- 我们只需递归遍历渲染树中的所有节点，并调用该函数即可，相关代码如下：

```
//GameScene.cpp
void GameScene::operateAllSchedulerAndActions(
    cocos2d::CCNode* node, int flag)
{
    if(node->isRunning()){
        switch (flag) {
            case k_Operate_Pause:
                node->pauseSchedulerAndActions();
                break;
            case k_Operate_Resume:
```

```
                node->resumeSchedulerAndActions();
                break;
            default:
                break;
        }
        CCArray* array = node->getChildren();
        if(array != NULL && array->count()>0){
            CCObject* iterator;
            CCARRAY_FOREACH(array, iterator){
                CCNode* child = (CCNode*)iterator;
                this->operateAllSchedulerAndActions(child, flag);
            }
        }
    }
}
```

- MenuLayer。菜单层是一个弹出式的界面，分为半透明背景和按钮。我们可以使用 CCMenu 来为菜单层添加按钮，使用 CCLayerColor 来创建半透明的背景层。下面的 createBackground 方法实现了为菜单层添加半透明背景的功能：

```
//MenuLayer.cpp
void MenuLayer::createBackground()
{
    CCLayerColor* colorBackground =
        CCLayerColor::create(ccc4(0, 0, 0, 128));
    this->addChild(colorBackground);
}
```

- ccc4 是 Cocos2d-x 定义的一个颜色结构体，它的 4 个参数分别对应 RGBA 的值，范围为 0~255，这里我们定义的是一个透明度为 50%的黑色背景。剩余的按钮都以 CCMenuItem 的形式实现。同样，按钮的回调函数只是将信息传递到 GameScene 中，由 GameScene 进行实质性的操作。我们相信这对于你已经没有什么难度了。
- 在 MVC 模型中，界面视图是要适配于数据的。我们也要让音乐和音效的开关状态适配于数据中的音量大小。因此，还需定义一个函数，其代码如下所示：

```
//MenuLayer.cpp
void MenuLayer::setSoundAndMusicVolume(float soundVolume, float musicVolume)
{
    bool soundFlag = soundVolume>0;
    bool musicFlag = musicVolume>0;
    _sound->setSelectedIndex(soundFlag);
    _music->setSelectedIndex(musicFlag);
}
```

- 通过传进的音量参数，设置好音乐音效的开关状态。
- 运行游戏，并进入游戏菜单，就可以看到类似图 21-8 所示的效果了。至此，第一轮的迭代就完成了。我们整理并实现了游戏的数据，并且将游戏中大部分的元素都在界面上体现出来了。然而界面上的一切都是毫无生机的，既不会移动，也没有交互，只是显示而已。在第二轮的迭代中，我们将用程序员的魔法，让整个游戏显得生机盎然。

图21-8 游戏菜单

21.2.2 第二轮迭代（1）

21.2.2 第二轮迭代（1）

首先，我们要完成的是玩家和游戏的交互工作。所谓交互，就是游戏针对用户的输入作出适当的反馈。不知读者是否还记得前面用户输入的内容。对于 Fishing Joy 而言，用户输入只有触摸一项。我们的做法是新建一个 TouchLayer 集中接受和识别触摸事件，并传回 GameScene 处理。这样的处理可以很好地降低触摸类的耦合性，便于对触摸进行修改和整理。

新建 TouchLayer 类，将其添加到 GameScene 中，并重载 setTouchEnabled 函数，便于随时打开和关闭触摸，相关代码如下：

```
//TouchLayer.cpp
void TouchLayer::setTouchEnabled(bool flag)
{
    if (m_bIsTouchEnabled != flag){
        m_bIsTouchEnabled = flag;
        if(flag){
            CCDirector::sharedDirector()->getTouchDispatcher()
                ->addTargetedDelegate(this, 0, true);
        }else{
            CCDirector::sharedDirector()->getTouchDispatcher()
                ->removeDelegate(this);
        }
    }
}
```

由于 Fishing Joy 只需要单点触摸即可，因此我们实现的是 CCTargetedTouchDelegate。我们为类中添加如下的代码以实现 CCTargetedTouchDelegate：

```
//TouchLayer.h
bool ccTouchBegan(cocos2d::CCTouch *pTouch, cocos2d::CCEvent *pEvent);
void ccTouchMoved(cocos2d::CCTouch *pTouch, cocos2d::CCEvent *pEvent);
void ccTouchEnded(cocos2d::CCTouch *pTouch, cocos2d::CCEvent *pEvent);
```

结合上面 3 个函数，可以分析出与触摸相关的游戏逻辑。

- 当触摸开始和移动时，改变 Cannon 的角度，并瞄准至触摸点。
- 当触摸结束时，向触摸点发射炮弹。
- 接收到触摸事件后，从 TouchLayer 一层一层地将事件传递到 Weapon 类中，如图 21-9 所示。

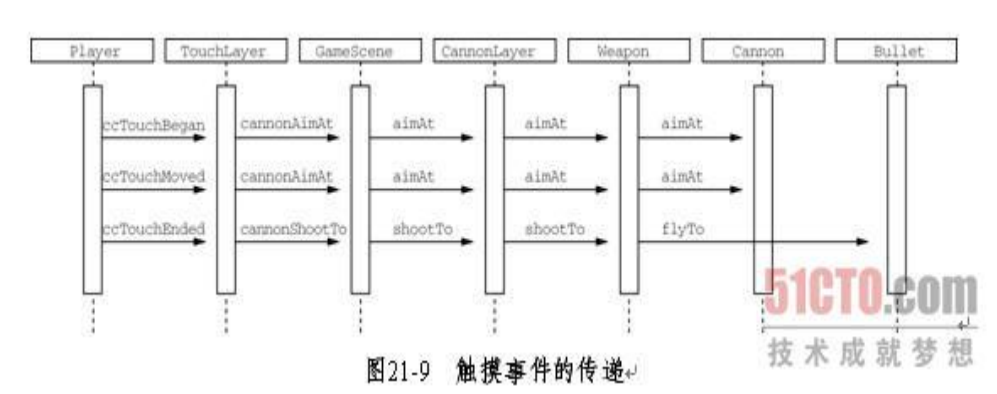


图21-9 触摸事件的传递

- 提示 倘若读者觉得这样一层一层地传递过于麻烦，我们还可以采用 CCNotificationCenter 快速地传递消息。CCNotificationCenter 类似于 Qt 中的消息槽，能够实现基于事件的消息传递。它同样是一个全局使用的单例，通过调用 CCNotificationCenter::shared
- NotificationCenter()获取其实例。
- 炮弹的瞄准和发射函数在逻辑上相对复杂一些，我们逐点攻克。
- 瞄准函数：首先调用 convertToWorldSpace 函数，将节点的坐标位置转换到屏幕坐标系中，然后使用 ccpAngleSigned 获得炮台需要旋转的弧度。ccpAngleSigned 是 Cocos2d-x 提供的一系列坐标运算函数中的一个，传入的两个参数 a 和 b 分别是夹角两条边的端点，而这两条边的交点默认是 CCPointZero，即坐标原点。瞄准函数的实现代码如下所示：

```
//Cannon.cpp
void Cannon::aimAt(CCPoint target)
{
    CCPoint location = this->getParent()->convertToWorldSpace(this->getPosition());
    float angle = ccpAngleSigned(ccpSub(target, location), CCPointMake(0, 1));
    this->setRotation(CC_RADIANS_TO_DEGREES(angle));
}
```

21.2.2 第二轮迭代（2）

21.2.2 第二轮迭代（2）

- 发射子弹函数：由于我们限制了每次只能发射一枚子弹，所以必须将该函数的返回值设置为布尔类型，以获取发射是否成功。首先判断子弹是否已经射出，若已经射出，返回 false 表示发射失败。否则，将子弹向触摸点的方向射出一定的距离，距离由武器的类型决定。发射至终点后，将子弹设为不可见。发射子弹函数的实现代码如下所示：

```
//Weapon.cpp
bool Weapon::shootTo(CCPoint touchLocation)
{
    if(this->weaponStatus()!=k_Weapon_Status_None) {
        return false;
    }
    else{
```

```
            float distance = 180 * (_cannon->getType()+1);
            CCPoint normal = ccpNormalize(ccpSub(touchLocation, this->getParent()->
                convertToWorldSpace(this->getPosition())));
            CCPoint vector = ccpMult(normal, distance);
            CCPoint target = ccpAdd(this->getPosition(), vector);
            _bullet->flyTo(target);
            return true;
        }
    }
    void Bullet::flyTo(CCPoint targetInWorldSpace)
    {
        CCPoint startInNodeSpace = CCPointZero;
        CCPoint startInWorldSpace = this->getParent()->convertToWorldSpace(startInNodeSpace);
        CCPoint targetInNodeSpace = this->getParent()->convertToNodeSpace(targetInWorldSpace);
        float angle = ccpAngleSigned(ccpSub(targetInWorldSpace, startInWorldSpace), CCPointMake(0
    , 1));


    this-&gt;setRotation(CC_RADIANS_TO_DEGREES(angle));
    this-&gt;setPosition(startInNodeSpace);
    this-&gt;setVisible(true);


    float speed = ccpDistance(startInNodeSpace, targetInNodeSpace) / 300.0f;
    CCMoveTo* moveTo = CCMoveTo::create(speed, targetInNodeSpace);
    CCCallFunc *callFunc = CCCallFunc::create(this, callfunc_selector(Bullet::end));


    CCFiniteTimeAction* seq = CCSequence::create(moveTo, callFunc, NULL);
    seq-&gt;setTag(k_Bullet_Action);
    this-&gt;runAction(seq);


    }
    void Bullet::end()
    {
        this->stopActionByTag(k_Bullet_Action);
        this->setVisible(false);
    }
```

■ 理清逻辑后的代码其实并不复杂，难点在于坐标系坐标点的转换和运算上。幸运的是，Cocos2d-x 提供了一系列接口，用于进行基础的转换运算。它们都在"CCPointExtension.cpp"中，并且在注释中已经说清楚了所有接口的用途，有兴趣的读者不妨尝试使用一下。

- 编译运行后，我们就可以如图 21-10 一样点击屏幕发射炮弹了。但此时并没有碰撞检测，我们的子弹会直接从鱼身上穿过。本章中 FishingJoy 项目的碰撞比较简单，因此不启用 Box2d 物理引擎作辅助，而是简单地在每一帧中判断矩形区域是否重叠。

图21-10 运行效果

- 碰撞检测主要分两部分。
- 子弹是否和鱼发生碰撞。
- 若已发生碰撞，展开网，并判断网和鱼是否碰撞。
- 由于本章更倾向于 Cocos2d-x 实际开发的应用，所以这部分的逻辑，我们不再一一叙述。在提供的示例工程中，也能找到相应的代码和注释。对于开发者而言，阅读代码会比阅读文字更直观。
- 添加完成碰撞检测后，可以得到如图 21-11 所示的运行效果。至此，第二轮的迭代就结束了。现在已经可以轻松地操纵武器去享受捕捉鱼儿的乐趣了。若读者觉得屏幕上固定不动的鱼儿没有捕捉的难度和乐趣，不妨尝试为其添加各式各样的运动方式，增加乐趣。

图21-11 加入碰撞检测

- **21.2.3 第三轮迭代（1）**

- 21.2.3 第三轮迭代（1）

- 从上一次的迭代中，我们已经得到了游戏的原型，但这部分并没有任何动态的游戏数据。为了使我们的游戏可以互动，还需要加入表 21-1 所列举的行为及数据。
- 表 21-1　改变游戏数据的行为

行为	影响的数据
发射炮弹	金币数减少
捕捉到鱼	金币数增加
倒计时结束	金币数增加
开关音乐	音乐音量
开关音效	音效音量

- 此时 MVC 模型的优势再次体现出来了，我们无需如无头苍蝇般在整个项目中一行一行地寻找以上行为的代码。因为它们都在 GameScene.cpp 中，相关代码如下：

```
//GameScene.cpp
void GameScene::cannonShootTo(CCPoint target)
{
    _cannonLayer->shootTo(target);
}
```

- 在执行 shootTo 函数之前，需要判断当前的金币是否足够发射子弹。若金币足够，则发射子弹，并减少金币数，相关代码如下所示：

```
//GameScene.cpp
void GameScene::cannonShootTo(CCPoint target)
{
    int type = _cannonLayer->getWeapon()->getCannon()->getType();
    int cost = (type+1) * 1;
    int currentGold = FishingJoyData::sharedFishingJoyData()->getGold();
    if(currentGold >= cost && _cannonLayer->shootTo(target)){
        this->alterGold(-cost);
    }
}
```

- 同样，在鱼被捕捉到时，在 void GameScene::fishWillBeCaught(Fish* fish)中添加金币即可。
- 游戏数据是游戏的核心部分，影响着游戏的可玩性。和数据连接后的 Fishing Joy，也使得玩家有了玩的目标和动力。但是，金币呢？我们需要一个提示来告知玩家当前的金币数。对比示例工程中的程序，金币的数目在改变时会有一个滚动效果。这个功能的实现并不难。在第 10 章中，我们已经介绍过如何用遮罩来实现这种效果了。但对于一个项目而言，简单实现这种技巧是不能满足我们要求的，必须将它们封装为一个组件。

组件 1：金币展示板

- 对于金币的展示板，我们使用 Counter 类先实现单个数字的设置，然后再创建多个 Counter 的实例，将其拼接起来。
- 在项目中新建一个文件夹，命名为"Component"，然后在该文件夹下创建 Counter 类，并添加如下代码：

```
//Counter.h
```

```
class Counter : public cocos2d::CCNode
{
    public:
        Counter();
        /**
         *  @brief
         *
         *  @param  presenters  0~9 这 10 个数字对应的节点的数组
         *  @param  digit    默认的数值
         *
         *  @return
         */
        static Counter* create(cocos2d::CCArray* presenters, int digit = 0);
        bool init(cocos2d::CCArray* presenters, int digit = 0);

        CC_PROPERTY(int, _digit, Digit);
    protected:
        void visit();
        //存放 0~9 数字对应的节点
        cocos2d::CCNode* _presenters;
        //改变数字时播放滚动动画
        void animation(int digit);
};
```

21.2.3 第三轮迭代（2）

21.2.3 第三轮迭代（2）

初始化 Counter 需要两个参数。第一个参数为数组，其下标的数字对应 CCNode 对象，如数组下标为 5 对应数字 5 的图像。第二个参数默认为 0，它表示了展示板最初展示的数字。

在 Counter 内部改变数字时，调用 animation 方法播放动画。其他类的实例通过调用 getDigit 获取当前的数字。

遮罩和动画的方式，我们已经在前面说明过了，在此不再赘述。不同的是，我们通过设置传入类型均为 CCNode 的数组，使该类能适应包括 CCLabelTTF 和 CCSprite 等在内的所有显示方式。同时，遮罩层的大小是通过传入的内容计算生成的，并能根据当前坐标实时更新，这就使得 Counter 组件具有较高的重用性。对于所有需要数字滚动效果的功能，我们都可以利用 Counter 实现。

Counter 类负责显示数字表盘中滚动的数字，使用了 11.2 节中介绍的遮罩效果，在其 visit 方法中实现了裁剪效果。具体实现如下所示：

```
//Counter.cpp
void Counter::visit()
{
    glEnable(GL_SCISSOR_TEST);
    CCNode* presenter = _presenters->getChildByTag(_digit);
    CCSize size = presenter->getContentSize();
    CCPoint location = this->getParent()->convertToWorldSpace
        (CCPointMake(this->getPosition().x-size.width*0.5,
        this->getPosition().y-size.height*0.5));
```

```
    glScissor(location.x, location.y, size.width, size.height);
    CCNode::visit();
    glDisable(GL_SCISSOR_TEST);
}
```

■ 接下来，我们就可以使用 Counter 类实现金币的显示了。新建 GoldCounterLayer 类，并在 PanelLayer 中添加它的一个实例。在 GoldCounterLayer 中，初始化创建 6 个 Counter 的实例，并通过 setNumber 方法，将传入的金币设置到 Counter 中。GoldCounterLayer 类的代码如下所示：

```
bool GoldCounterLayer::init(int number)
{
    int fontSize = 16;
    CCLabelTTF* goldLabel = CCLabelTTF::create("金币:", "Thonburi", fontSize);
    this->addChild(goldLabel);
    CCSize goldLabelgoldLabelSize = goldLabel->getContentSize();

    for(int i = 0 ;i < 6;i++){
        int count = 10;
        CCArray* presenters = CCArray::createWithCapacity(count);
        for(int j = 0;j < count;j++){
            CCLabelTTF* label = CCLabelTTF::create(CCString::createWithFormat("%d",j)
                ->getCString(), "Thonburi", fontSize);
            presenters->addObject(label);
        }
        Counter* counter = Counter::create(presenters);
        counter->setPosition(CCPointMake(
            goldLabelSize.width*0.8+fontSize*0.75*i, 0));
        this->addChild(counter, 0, i);
    }
    this->setNumber(number);
    return true;
}
void GoldCounterLayer::setNumber(int number, int ceiling)
{
    number = MIN(ceiling, number);
    number = MAX(number, 0);
    _number = number;
    for(int i = 0 ;i < 6;i++){
        Counter* counter = (Counter*)this->getChildByTag(i);
        int digit = _number / (int)(pow(10.0, 6-i-1)) % 10;
        counter->setDigit(digit);
    }
}
```

■ 这里我们简单地使用 0~9 的 10 个 CCLabelTTF 作为显示对象。setNumber 方法通过简单的算法，将数字的每一位分别分离出来后，设置到 counter 中。

- 如果这时美术素材有所变动，需要使用图片来显示金币的数字，我们只需改变"GoldCounter
- Layer.cpp"中 presenters 的内容，而 Counter 类中所有的代码都不需要改动，滚动效果依然能够出现。

21.2.3 第三轮迭代（3）

- 21.2.3 第三轮迭代（3）

组件 2：倒计时器

- 与游戏数据相关的功能还有一个倒计时器。倒计时器的功能在游戏中非常常见，因此，我们也将其制作为一个组件。和以前的金币展示板不同的是，需要在倒计时结束后，在倒计时器的内部调用函数通知其他类。因此，需要设计一个协议完成这个功能。
- 在 Cocos2d-x 中，有许多协议，如 CCTouchDelegate 和 CCTextFieldDelegate 等。
- 我们在"Component"文件夹下新建一个 ScheduleCounterProtocol，并创建 Schedule-
- CounterDelegate 类，该类的代码如下所示：

```
//ScheduleCounterDelegate.h
class ScheduleCounterDelegate
{
public:
    //必需的
    virtual void scheduleTimeUp() = 0;
    //可选的
    virtual void setScheduleNumber(int number){return;};
};
```

- 所有使用了倒计时功能的类，都将继承自 ScheduleCounterDelegate 类。事实上，该类更接近于 Java 或 Objective-C 中的接口，而不是类。在这个类中，scheduleTimeUp 是一个纯虚函数，意味着这是必须实现的功能。而 setScheduleNumber 只作为一个可选的函数供子类实现。
- 接下来，再创建 ScheduleCountDown 类，在该类中保存 ScheduleCounterDelegate 的一个实例，然后调用 Cocos2d-x 提供的 schedule 函数，每秒对事件进行一次计算和判断，并调用 setScheduleNumber。当倒计时结束时，调用 scheduleTimeUp。相关代码如下所示：

```
//ScheduleCountDown.cpp
void ScheduleCountDown::schedulePerSecond()
{
    _curTime--;
    if(_curTime <= 0){
        if(this->getLoop()){
            _curTime = _maxTime;
        }else{
            this->unschedule(schedule_selector
                (ScheduleCountDown::schedulePerSecond));
        }
        _target->scheduleTimeUp();
    }
```

```
    _target->setScheduleNumber(_curTime);
}
```

- 最后，在 PanelLayer 中，让其继承 ScheduleCounterDelegate 协议，并实现协议中的两个接口，相关代码如下：

```
//PanelLayer.cpp
void PanelLayer::scheduleTimeUp()
{
    GameScene* gameScene = (GameScene*)this->getParent();
    gameScene->scheduleTimeUp();
}
void PanelLayer::setScheduleNumber(int number)
{
    _scheduleLabel->setString(CCString::createWithFormat("%d",number)
        ->getCString());
}
```

- 至此，我们已经完成了一个有数据、有交互的游戏。但是否太过安静了呢？没错，我们还没有为游戏添加任何音乐。

游戏音乐

- 在 2D 游戏开发中，游戏音乐是相对简单的一部分，因为它与游戏本身的依赖性并不大，而且大部分情况下只有播放、停止和暂停的操作，但需要注意的是数据的同步。我们将音乐音效的状态持久化保存。
- 在项目中新建 PersonalAudioEngine 类，并让其继承自 SimpleAudioEngine。
- PersonalAudioEngine 同样是一个单例，但它实际上只是对 SimpleAudioEngine 操作的封装，其功能包括预载入音乐音效以及数据同步。同时，对于不需要同步的操作，我们不进行重载，PersonalAudioEngine 则会调用 SimpleAudioEngine 的接口。
- 对于音乐音效的预载入，SimpleAudioEngine 提供了两个函数，具体如下所示：

```
//SimpleAudioEngine.h
...
void preloadBackgroundMusic(const char* pszFilePath);
void preloadEffect(const char* pszFilePath);
...
```

21.2.3 第三轮迭代（4）

21.2.3 第三轮迭代（4）

- 只需指定音乐音效的文件名，在 PersonalAudioEngine::init 中载入即可。
- 对于数据的同步，有音乐和音效的音量大小两部分。我们重载其 SimpleAudioEngine 中对应的两个函数，并在操作完成后访问 FishingJoyData 同步数据，相关代码如下所示：

```
//PersonalAudioEngine.cpp
void PersonalAudioEngine::setBackgroundMusicVolume(float volume)
{
    SimpleAudioEngine::sharedEngine()->setBackgroundMusicVolume(volume);
    FishingJoyData::sharedFishingJoyData()->setMusicVolume(volume);
```

```
}

void PersonalAudioEngine::setEffectsVolume(float volume)
{
    SimpleAudioEngine::sharedEngine()->setEffectsVolume(volume);
    FishingJoyData::sharedFishingJoyData()->setSoundVolume(volume);
}
```

- 接下来，只需要在 GameScene 中找到暂停按钮、炮弹发射、游戏暂停、恢复、音乐音效设定的操作，调用 PersonalAudioEngine 的接口播放音乐音效即可，此处不再赘述。
- 至此三次迭代都结束了，一个完整的 Fishing Joy 游戏就完成了。此时我们可以将它分享给朋友试玩了，但若要将其发布，这还远远不够。其中最突出的问题在于，程序运行后立刻开始游戏，显得过于突兀。通常的做法是设置一个开始场景，等待玩家的指令然后进入游戏。
- 开始场景包括如下内容。
- 载入资源：预载入部分在 GameScene 中的 preloadResources()函数中已经完成，只需整理后添加到 StartScene 中即可。
- 异步载入：长时间地载入数据会导致游戏界面出现"假死"，所以有必要开启新的线程进行载入。我们可以使用 Cocos2d-x 引擎中配套分发 pthread 的第三方库实现多线程，但对于初学者而言，此法会有造成诸多的线程安全问题。另一个做法是调用

```
CCTextureCache()->addImageAsync(const char* path, CCObject* target,
    SEL_CallFuncO selector)
```

- 异步载入图片资源。由于大多数游戏中的图片资源远大于其他的数据、音乐音效等文件，所以此法足以保障游戏界面的持续流畅。
- 进度条：当载入时间过长时，通常需要提供一个进度条，以提示玩家游戏还在持续响应中。进度条的实质与之前的金币展示板类似，同样是界面的遮罩效果，不过 Cocos2d-x 已经提供了足够好用的类 CCProgressTimer，我们无需再操作底层的绘图。
- CCProgressTimer 通过传入一个 CCSprite 的对象进行实例化，它提供了放射状和条状两种进度条类型，通过 setType 进行设置。同时，CCProgressTo 和 CCProgressFromTo 两个继承自 CCAction 的动作，可以以动画的形式对进度条的进度进行设置。
- 通过以上 3 部分的组合，就能够实现 StartScene 了。具体的实现工作，我们将不再用示例工程中的代码进行讲解，而是留给读者自行完成，以此作为读者对 Cocos2d-x 开发能力的测试。倘若读者在实现过程中遇到疑惑，不妨到示例项目中阅读代码。