互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第3章

基础知识

本章将介绍 cocos2d 游戏引擎中最基本的一些构成要素。在今后创建的每个游戏中，你会经常使用这些类。所以，了解哪些类是可用的，并理解它们之间的协作方式，将有助于你编写出更出色的游戏。有了这些知识作为基础，你会发现用 cocos2d 编写游戏比想象中简单得多。

本章附带一个名为 Essentials 的 Xcode 项目，其中包含了在此讨论的所有内容以及一些额外的示例。源代码中添加了大量的注释，你可以把它们当作本书附录来阅读。

本章将首先对 cocos2d 游戏引擎的架构进行高度概括。由于在不同游戏引擎中，游戏对象的管理方式以及在屏幕上的显现方式各不相同，因此最好先对 cocos2d 中每个构成元素的意义以及它们的组合方式有所理解。

3.1 cocos2d 中的单件类

cocos2d 充分利用了 Singleton 设计模式。鉴于它常常成为开发者热议的话题，所以我想有必要在此提一下。从理论上讲，单件(Singleton)就是一个普通的类，但它在应用程序的整个生命周期中只被实例化一次。为了确保这一点，cocos2d 采用静态方法来创建并访问这个对象实例。所以，要访问单件对象，不应使用 alloc/init 或静态的自动释放初始化器，而应该调用以 shared 开头的方法。下面列举 cocos2d 中一些最常用的单件类以及访问它们的方法：

```
CCActionManager* sharedManager = [CCActionManager sharedManager];
CCDirector* sharedDirector = [CCDirector sharedDirector];
CCSpriteFrameCache* sharedCache = [CCSpriteFrameCache
        sharedSpriteFrameCache];
CCTextureCache* sharedTexCache = [CCTextureCache sharedTextureCache];
CCTouchDispatcher* sharedDispatcher = [CCTouchDispatcher
        sharedDispatcher];
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
CDAudioManager* sharedManager = [CDAudioManager sharedManager];
SimpleAudioEngine* sharedEngine = [SimpleAudioEngine sharedEngine];
```

单件的好处在于它可以在任何地方被任何类使用。可以把它理解为一个全局类(请参照“全局变量”的概念来加深理解)。如果在很多不同的地方需要使用同一组数据和方法，那么这时候单件就非常有用。音频就是一个很好的示例，因为任何类(无论是玩家类、敌方类、菜单类或过场动画类)都可能需要播放音效或切换背景音乐，所以选择用单件来播放音频是明智之举。与此类似，你可能希望将全局的游戏状态信息(可能是玩家持有的兵器数目，也可能是军队的一个排有多少人，等等)存储在一个单件中，这样就可以在游戏的不同关卡中使用同一份数据了。

如代码清单 3-1 所示，单件的实现相当简洁。这个示例用最少的代码实现了一个单件类：MyManager。其中，静态方法 sharedManager 提供了对 MyManager 的单一实例的访问：如果实例不存在，就为 MyManager 实例分配内存并完成初始化；如果存在，就返回实例。

代码清单 3-1 用单件实现示例类 MyManager

```
static MyManager *sharedManager = nil;

+(MyManager*) sharedManager
{
        if (sharedManager == nil)
        {
                sharedManager = [[MyManager alloc] init];
        }
        return sharedManager;
}
```

不过，单件也有它的缺点。正因为它们易于实现、方便使用，而且可以在其他任何类中使用，所以人们常会在一些不应该用单件来实现的场合使用它们。

例如，你可能想：我只有一个玩家对象，为什么不把这个玩家类设为一个单件呢？这种想法似乎很完美。但你会突然发现，只要玩家进入下一个关卡，他就不仅带着上一轮的分数，而且还保持着上一轮最后的动作、血量、捡到的所有道具。甚至，他刚进入下一关就已经在“狂暴”模式(Berserk Mode)下了，而这恰恰是因为他离开上一关的时候就处于这个模式。

为了解决这个问题，你会去添加几个方法，在关卡发生变化时对一些变量进行重置。到目前为止，一切都进行得很顺利，但是当你为游戏设计更多特性时，你会发现转换关卡时需要添加好多变量，而且还要对很多已有变量进行维护。更糟糕的是，假如有一天朋友建议你为 iPad 版本添加双人模式，这时候你想起玩家类是个单件！任何时候都只能有一个玩家实例！于是你将面临一个非常令人头疼的问题：要么重写大量代码进行彻底重构，要么放弃双人模式。

越是依赖于单件，就会碰到越多这样的麻烦事。所以，在你创建一个单件类之前，务必要考虑清楚，对于这个类以及它的数据，是否真的只需要一个实例？这个设计以后是否会发生变化？

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

3.2 Director 类

CCDirector 类(简称为 Director)是整个 cocos2d 游戏引擎的核心。如果回想一下在第 2 章中学到的 HelloWorld 程序，就会想起 cocos2d 中很多初始化过程都包含了[CCDirectorsharedDirector]这个调用。Director 被设计为单件，这个设计是非常合理的：它存储了 cocos2d 中大量的全局配置信息，而且管理着所有的 cocos2d 场景。

Director 的主要用途包括：

- 切换场景
- 存储 cocos2d 配置信息
- 访问视图(OpenGL、UIView、UIWindow)
- 暂停、恢复以及终止游戏
- 在 UIKit 和 OpenGL 之间转换坐标

事实上，一共有 4 种 Director 可供选择，但它们之间只有细微的差别。最常用的一种是 CCDisplayLinkDirector，它的内部采用苹果公司官方的 CADisplayLink 类来实现。这是一个很好的选择，但它只在 iOS 3.1 或更高的版本上可用。另一种选择是，可以使用 CCFastDirector。如果想同时使用 cocos2d 和 Cocoa Touch 视图，就必须使用 CCThreadedFastDirector，因为只有它可以同时支持 cocos2d 和 Cocoa Touch 视图。CCThreadedFastDirector 的缺点在于耗电量比较大，如果你非常注重这一点，那么应选用 CCTimerDirector。不过，不到万不得已，最好不要用它，因为它是所有 Director 中最慢的一个。

3.3 场景图

场景图有时也称为场景层次体系，是一个由所有处于活动状态的 cocos2d 节点构成的层次体系。除了场景以外，每个节点只有一个父节点，同时可以拥有任意个数的子节点。

当向节点添加其他节点时，就是在构建一个节点场景图。图 3-1 显示了某个游戏场景图中的一部分。场景图的顶层通常是一个场景节点，接下来是一个层节点。层节点在 cocos2d 中负责接收触摸输入和加速计输入。

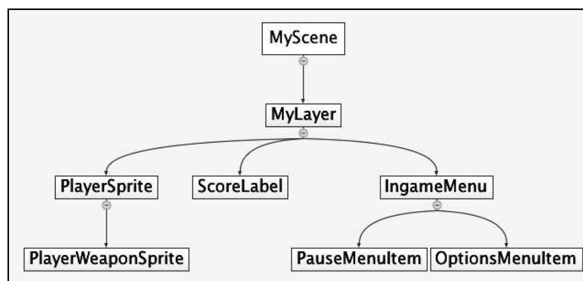

图 3-1 一个简化的 cocos2d 场景图，采用大量不同的节点来表示玩家、武器、分数以及用来暂停或进入游戏的菜单

在 CCLayer 对象的所有子节点中，最上层的节点代表了游戏的各个组成部分。其中大

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

多数都是精灵(sprite)节点，偶尔也有一些标签节点(例如用来显示游戏分数的标签)。菜单节点和菜单项节点用于显示游戏进行时的菜单，玩家可以通过单击该菜单上的按钮暂停游戏或者返回游戏主菜单。

细心的读者可能注意到在图 3-1 中，PlayerSprite 节点下还有一个名为 PlayerWeaponSprite 的节点。也就是说，PlayerWeaponSprite 是从属于 PlayerSprite 的。如果 PlayerSprite 发生移动、旋转或缩放，那么在不添加任何代码的情况下，PlayerWeaponSprite 也会进行同样的操作。这就是场景图的神奇所在：对某个节点采取的大多数操作会同时影响到它的所有子节点。不过，这个特点有时候也会带来一些比较费解的问题，因为子节点的位置和旋转角度突然间都与父节点产生了关联，一下子可能会让人难以适应。

我写了一个名为 NodeHierarchy 的 Xcode 项目，读者可以在随书源码中找到它。这个示例向大家展示了层次体系中各个节点是如何互相影响的。我相信让你看一下真正的效果，会比我用再多的文字和图像去解释要有用得多。

3.4 CCNode 类层次体系

所有节点都有一个公共父节点类——CCNode，它定义了除节点显示外的多数公共属性和方法。图 3-2 显示了从 CCNode 继承的一些最重要的类，将来你会经常使用这些类。而且，即使只用这些类，也能制作出非常出色的游戏。

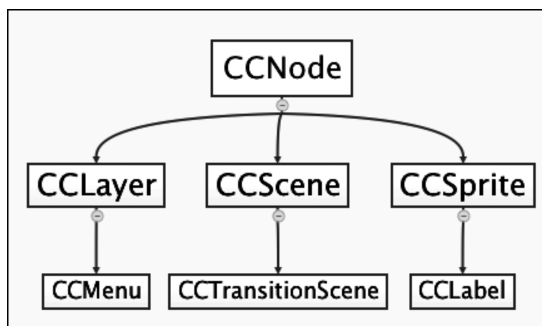

图 3-2 cocos2d 中最重要的节点类，所有节点类都从 CCNode 继承，它定义了公共的属性和方法

3.5 CCNode 类

CCNode 是所有节点的基类。它是一个没有具体显示的抽象类，仅用于定义所有节点的公共属性和方法。

3.5.1 节点的处理方式

CCNode 类实现了添加、获取以及删除子节点的所有方法。下面列举了一些针对子节点的处理方式：

- 创建新节点：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
CCNode* childNode = [CCNode node];
```

- 把新建节点添加为子节点：

```
[myNode addChild:childNode z:0 tag:123];
```

- 获取子节点：

```
CCNode* retrievedNode = [myNode getChildByTag:123];
```

- 利用标记删除子节点；参数“cleanup”将令所有正在运行的动作(action)停止：

```
[myNode removeChildByTag:123 cleanup:YES];
```

- 利用指针删除节点：

```
[myNode removeChild:retrievedNode];
```

- 删除某个节点的所有子节点：

```
[myNode removeAllChildrenWithCleanup:YES];
```

- 将某个节点从其父节点中删除：

```
[myNode removeFromParentAndCleanup:YES];
```

addChild 方法中的参数“z”决定了节点的绘制顺序。首先绘制 z 值最低的节点，最后绘制 z 值最高的节点。如果多个节点拥有相同的 z 值，就按照添加它们的先后顺序进行绘制。当然，这只对有具体显示的节点(比如精灵)有效果。

tag 参数保证了你在使用 getChildByTag 方法时能够区分并获取特定节点。

注意：

如果多个节点拥有相同的标记值，getChildByTag 方法将返回最先匹配该标记值的节点，其她拥有该标记值的节点就不可能再被访问到了。所以，确保每个节点的标记值都是唯一的。

而且，动作也是可以有标记值的。节点和动作的标记值互相不会产生影响，所以，某个节点与某个动作的标记值相同不会产生任何问题。

3.5.2 动作的处理方式

节点也可以运行动作。稍后我会向大家介绍动作的概念。现在，你只要知道动作是指在特定时间内完成移动、旋转、缩放以及其他操作的一些行为。

- 动作的声明：

```
CCAction* action = [CCBlink actionWithDuration:10 blinks:20];
action.tag = 234;
```

- 通过运行上面声明的动作可以令节点闪烁：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
[myNode runAction:action];
```

- 如果之后想获取该动作，可以通过它的标记来获得：

```
CCAction* retrievedAction = [myNode getActionByTag:234];
```

- 可以通过标记来停止指定的动作：

```
[myNode stopActionByTag:234];
```

- 也可以通过指针来停止指定的动作：

```
[myNode stopAction:action];
```

- 还可以同时停止某节点上运行的所有动作：

```
[myNode stopAllActions];
```

3.5.3　消息调度

节点可以对消息进行调度，其实就是指 Objective-C 中每隔一段时间调用一次方法。很多情况下，你可能想对一个节点采用特定的更新方法来进行某个操作，例如碰撞检测。要使特定的更新方法以每帧的频率被调用一次，最简单的做法如下所示：

```
-(void) scheduleUpdates
{
        [selfscheduleUpdate];
}

-(void) update:(ccTime)delta
{
        // this method is called every frame
}
```

非常简单吧？注意，更新方法是有固定格式的，也就是说，它永远是像示例代码这样定义的。参数 delta 代表了该函数自上次被调用到现在所经过的时间。如果希望每帧调用一次更新方法，以上示例是一个比较好的做法。但是如果想要有更好的灵活性，最好还是采用其他更新方法。

如果想调用另一个方法，或者不想以每帧的频率(而是以每 0.1 秒的频率)调用该方法，就应该采用以下做法：

```
-(void) scheduleUpdates
{
        [self schedule:@selector(updateTenTimesPerSecond:)interval:0.1f];
}

-(void) updateTenTimesPerSecond:(ccTime)delta
{
        // this method is called according to its interval, ten times per second
}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

注意，如果 interval 的值为 0，还是应该用 scheduleUpdate 方法。但是，假如你之后需要取消对一个定时方法的调用，最好还是采取上面的做法，因为 scheduleUpdate 方法做不到这一点。

这里的更新方法在格式上和原来一样，还是只有一个参数“delta”。与之前有所不同的是，该更新方法的名称可以根据你的喜好来修改，而且它以 0.1 秒的频率被调用。例如，当游戏逻辑复杂到你不想以每帧的频率去检测玩家是否胜出时，就可以采用这种做法。假如希望在 10 分钟后调用某方法，可以用选择器选定你的更新方法，并把 interval 设定为 600。

注意：

读者看到@selector(…)语句的时候可能会觉得有些陌生，这在 Objective-C 中用来指定某个特定方法。关键在于，决不能忘记上述示例中 updateTenTimesPerSecond 后面的冒号！该冒号告诉 Objective-C：“去找一个名为 updateTenTimesPerSecond 的方法，这个方法有且只有一个参数。”如果忘记写冒号，编译还是可以通过的，但是程序一运行就会发生崩溃。在 Debugger Console 窗口中，你会看到这样的错误日志：“unrecognized selector sent to instance…”。

@selector(…)中的冒号数一定要与所指定方法的参数个数相同。假设有下述方法：

```
-(void) example:(ccTime)delta sender:(id)sender flag:(bool)aBool
```

那么，对应的@selector 语句就应该是：

```
@selector(example:sender:flag:)
```

不论是在调度消息时还是在其他情况下使用@selector(…)，都需要注意一个很重要的问题：在默认情况下，如果方法名不存在，编译器并不会报错；但是，一旦程序运行时调用了指向不存在方法的 aselector 语句，应用程序就会立即崩溃。由于这个调用是在 cocos2d 内部完成的，因此很难找到问题所在。好在，你可以做一些设置来强制编译器报错。图 3-3 显示的是设置强制报错的界面。本章的“Essentials”Xcode 项目也已经做了相同设置。

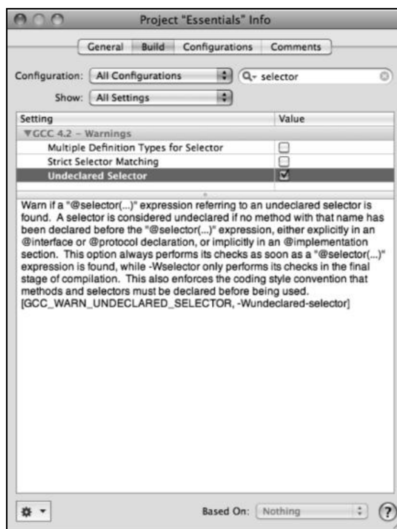

图 3-3　激活 Build 设置中对于 Undeclared Selectors 的警告

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

接下来要讨论的是，如何使这些被调度的方法不再被调用。可以通过取消调度做到这一点。

要使节点中所有用选择器选定的方法(包括使用 scheduleUpdate 指定的方法)停止，可使用：

```
[self unscheduleAllSelectors];
```

要使某个用特定选择器选定的方法停止(假设选择器名为“updateTenTimesPerSecond:”)，可使用：

```
[self unschedule:@selector(updateTenTimesPerSecond:)];
```

注意，该做法不会令使用 scheduleUpdate 指定的更新方法停止。

关于如何使用选择器来进行调度或者取消调度，这里有一个很有用的小窍门。很多时候，可能需要在一个被调度的方法中取消对某个方法的调用，但是因为自定义方法可能会被改名，所以不想在这里重复具体的方法名和参数个数。下面演示了如何用选择器对一个方法进行调度，并在第一次调用后取消调度：

```
-(void) scheduleUpdates
{
        [self schedule:@selector(tenMinutesElapsed:) interval:600];
}

-(void) tenMinutesElapsed:(ccTime)delta
{
        // unschedule the current method by using the _cmd keyword
        [selfunschedule:_cmd];
}
```

此处的关键字_cmd 是当前方法的快捷表达方式。这种方法可以有效地使 tenMinutesElapsed 方法不再被调用。事实上，也可以用_cmd 来执行调度。假设现有一个方法，你需要以不同的时间间隔调用它，那么可以使用如下代码来完成：

```
-(void) scheduleUpdates
{
        // schedule the first update as usual
        [self schedule:@selector(irregularUpdate:) interval:1];
}

-(void) irregularUpdate:(ccTime)delta
{
        // unschedule the method first
        [self unschedule:_cmd];

        // I assume you'd have some kind of logic other than random to determine
        // the next time the method should be called
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        float nextUpdate = CCRANDOM_0_1() * 10;

        // then re-schedule it with the new interval using _cmd as the selector
        [self schedule:_cmdinterval:nextUpdate];
}
```

从长远来看，使用_cmd 关键字能为你减少很多痛苦，因为它帮助你避免了很多因为选择器选错方法而带来的可怕问题。

关于调度，还有最后一个问题要提一下，那就是更新方法的优先级。请看以下代码：

```
// in Node A
-(void) scheduleUpdates
{
        [self scheduleUpdate];
}

// in Node B
-(void) scheduleUpdates
{
        [self scheduleUpdateWithPriority:1];
}

// in Node C
-(void) scheduleUpdates
{
        [self scheduleUpdateWithPriority:-1];
}
```

看懂这段代码可能要花一点时间。所有节点都在调用“– (void) update:(ccTime)delta”方法。但是因为有了优先级设定，所以节点 C 中的更新方法是最先被调用的。接下来是调用节点 A 中的更新方法，因为 scheduleUpdate 的默认优先级为 0。最后是节点 B 中的更新方法被调用，因为它的优先级最高。更新方法是以优先级从低到高的顺序被调用的。

你可能心存疑惑，什么时候会用到这个优先级呢？老实说，它确实不常用，以我多年的经验来看，它会在一些比较少见的情况下发挥优势，例如，在物理模拟器发生自动更新前后对某个物理对象施加力的作用。官方文档对于优先级的解释也能证实我的这个观点，因为它也提到了物理更新。有时候，通常是在项目开发后期，你会发现一些因为计时问题导致的奇怪的 bug，要解决问题，必须要在其他所有对象都更新完之后，再对玩家进行更新。

在碰到与更新的优先级相关的问题之前，你可以放心地跳过这段内容。

3.6 场景和层

与 CCNode 类似，CCScene 和 CCLayer 类也没有具体显示。它们通常仅供内部使用，作为场景图起点的抽象概念。所谓场景图，就是一个以 CCScene 为根节点的继承体系。CCLayer 通常是用来组织节点的一个单元，而且只要进行了相关设置，它还可以用来接收触摸输入及加速计输入。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

3.6.1 CCScene

一个 CCScene 对象往往是场景图中的第一个节点。通常来说，CCScene 节点的第一层子节点一定是 CCLayer 的子类。有趣的是，游戏中的各个对象总是由这些子节点来保存，而不是由父节点 CCScene 保存。由于场景对象本身通常不含有任何游戏相关代码，而且它很少被子类化，因而它往往是用 CCLayer 对象中的静态方法“+(id) scene”来创建的。我在第 2 章已经简单提到过，但还是想在这里再强调一下以加深印象：

```
+(id) scene
{
          CCScene* scene = [CCScene node];
          CCLayer* layer = [HelloWorld node];
          [sceneaddChild:layer];

          return scene;
}
```

这个方法应该加在 AppDelegate 中 applicationDidFinishLaunching 方法的最后。可以用 Director 类的 runWithScene 方法来启动第一个场景：

```
// only use this to run the very first scene
[[CCDirector sharedDirector] runWithScene:[HelloWorld scene]];
```

以后，你可以通过调用 replaceScene 方法来替换现有场景：

```
// use replaceScene to change all subsequent scenes
[[CCDirector sharedDirector] replaceScene:[HelloWorld scene]];
```

注意：

如果在 HelloWorld 场景里运行这些代码，不会有任何问题。一个新的 HelloWorld 实例会被生成，并替换掉原先的 HelloWorld 实例，相当于刷新场景。但是，不要把 self 作为参数传给 replaceScene 方法以达到刷新场景的目的，那样做会让游戏卡死！

3.6.2 场景和内存

当进行场景替换时，新场景往往在旧场景释放前就被加载到内存中了，这会导致内存负荷瞬间增大。所以，场景替换是最容易引发内存警告或者干脆导致程序崩溃的。当你的游戏使用大量内存的时候，就应该尽早并尽量多地对场景切换的情况进行测试。

注意：

当进行场景切换时，cocos2d 会把自己占用的内存清理干净。它会删去所有节点，停止所有动作，并且对所有用选择器选中的方法取消调度。之所以提出这个问题，是因为有时候我看到一些开发者直接调用 cocos2d 的 removeAll 方法。其实这是没有必要的，你应该信任 cocos2d 的内存管理能力。

当开始使用场景过渡效果时，内存问题就显得愈发明显了。这时候，新场景会被创建，

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

然后过渡效果在运行，一直要等到过渡效果完成后，旧场景才会从内存中释放。向场景或创建场景的层中添加日志记录语句会是一个非常好的习惯：

```
-(id) init
{
        if ((self = [super init]))
        {
                CCLOG(@"%@: %@", NSStringFromSelector(_cmd), self);
        }
}

-(void) dealloc
{
        CCLOG(@"%@: %@", NSStringFromSelector(_cmd), self);

        // always call [super dealloc] in the dealloc method!
        [super dealloc];
}
```

注意观察这些日志消息。如果发现在进行场景切换时 dealloc 日志消息从来没有显示出来，这就是一个巨大的预警信号。这意味着整个场景都没有从内存中释放，发生了泄露。cocos2d 本身是决不会产生这种错误的。在大多数情况下，发生这个问题的原因是在保留和释放节点时发生了错误。

记住，绝对不要把节点添加为场景图的子节点，然后再去保留它。可以用 cocos2d 提供的方法来获取这些节点对象，或者宁愿使用一个弱引用去指向这个节点，也不要保留它。只要把内存管理的问题丢给 cocos2d，你就是安全的。

3.6.3 推进和弹出场景

在讨论场景替换的同时，我还要提一下 pushScene 和 popScene 这两个有用的方法。这两个方法用来在不释放旧场景内存的情况下运行新场景，旨在加快场景替换的速度。不过这里有个令人头疼的问题：如果新、旧两个场景对内存需求都不大，可以共享内存，那么无论如何它们切换起来都是很快的；但如果这两个场景都非常复杂，加载起来很慢，那么使用 pushScene 和 popScene 以后，这两个场景互相争夺宝贵的内存资源，内存使用很快就会达到一个非常危险的级别。

使用 pushScene 和 popScene 最大的问题在于，很多场景可以相互叠加地存在于内存之中。你可以推进一个场景，然后在运行新场景时，这个新场景又推进了另一个场景。只要稍不留意，就可能忘记弹出一个场景，或者对于同一个场景弹出太多遍。更可怕的是，所有这些场景都共享着同一块内存。

如果一个场景在很多地方被用到，那么使用 pushScene 和 popScene 会非常方便。用来调节音量和选择背景音乐的 Settings 场景就是一个很好的例子。你可以推进 Settings 场景来显示它，当按下 Settings 场景中的返回按钮时，调用 popScene 就可以返回到显示 Settings 场景前的情形。不论 Settings 场景是在主菜单中、游戏过程中还是在其他地方打开，pushScene 和 popScene 都是适用的。而且，开发者还不用时刻关心 Settings 场景是在哪里

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

被打开的。

但是，你需要在所有可能打开 Settings 场景的情况下做好测试工作来保证内存一定是够用的。理想的情况是：Settings 场景本身足够轻巧，不会在任何地方引发内存不足。

可以使用以下代码在任何地方显示 Settings 场景：

```
[[CCDirector sharedDirector] pushScene:[Settings scene]];
```

如果正处于 Settings 场景下，又想关闭这个场景，调用 popScene 就可以返回到之前还保留在内存中的场景了：

```
[[CCDirector sharedDirector] popScene];
```

3.6.4 CCTransitionScene

场景过渡就是任何从 CCTransitionScene 继承的类，它们可以让你的游戏看起来非常专业。

注意：

在此我要先给出一个警告：并不是所有过渡效果都很有用。虽然它们看起来很优雅，但玩家最在乎的还是过渡的速度，哪怕只有 3 秒，对玩家来说都是难以忍受的。我倾向于在一秒内完成过渡，或者在适当的情况下干脆不使用过渡效果。

切忌在场景转换时采用随机过渡效果。同样身为一个游戏开发者，我能够理解你做出华丽的过渡效果时有多么兴奋，但是玩家并不在乎这些。如果你不确定用哪种过渡效果好，干脆别用了。要知道，“可以”使用过渡效果并不意味着你非得去用它。

要使用过渡效果，只要在场景转换时添加一行代码就可以做到(虽然我不得不承认，那一行代码可能是很长很长的一行，这取决于过渡效果的名称长度以及所需参数的个数)。这里我列举一个常见的淡入淡出过渡的例子，它会在一秒内过渡到纯白背景：

```
// initialize a transition scene with the scene we'd like to display next
CCFadeTransition* tran = [CCFadeTransition transitionWithDuration:1
                          scene:[HelloWorld scene] withColor:ccWHITE];
// use the transition scene object instead of HelloWorld
[[CCDirector sharedDirector] replaceScene:tran];
```

可以把 CCTransitionScene 与 replaceScene 和 pushScene 结合起来使用，但是，正如我前面说过的，千万不要将过渡效果和 popScene 一起使用。

cocos2d 中有很多过渡效果可供使用，虽然很多只是在方向上有所变化，比如说过渡从哪边开始，往哪个方向进行扩展等。下面我会列出现有的过渡效果，并对每个效果做一些简单的描述：

- CCFadeTransition：淡入淡出到一个指定颜色的背景，然后恢复。
- CCFadeTRTransition(还有其他 3 种版本)：瓦片翻转，渐渐显示新场景。
- CCJumpZoomTransition：旧场景弹跳着缩小，新场景弹跳着放大。
- CCMoveInLTransition(还有其他 3 种版本)：旧场景移出，同时新场景从任意方向移入。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

- CCOrientedTransitionScene(还有其他 6 种版本)：整个场景会翻转过来。
- CCPageTurnTransition：翻页效果。
- CCRotoZoomTransition：旧场景边旋转边缩小，新场景边旋转边放大。
- CCShrinkGrowTransition：旧场景缩小，新场景在旧场景基础上放大。
- CCSlideInLTransition(还有其他 3 种版本)：新场景从各个方向掠过旧场景。
- CCSplitColsTransition(还有另一种版本)：旧场景切成竖条，从上方或下方散开，新场景显现。
- CCTurnOffTilesTransition：新场景呈瓦片状替代旧场景。

3.6.5 CCLayer

有时候你的场景中需要不止一个层，那么可以用以下方法来创建场景：

```
+(id) scene
{
        CCScene* scene = [CCScene node];

        CCLayer* backgroundLayer = [HelloWorldBackground node];
        [scene addChild: backgroundLayer];

        CCLayer* layer = [HelloWorld node];
        [scene addChild:layer];

        CCLayer* userInterfaceLayer = [HelloWorldUserInterface node];
        [scene addChild: userInterfaceLayer];

        return scene;
}
```

另一种方法是继承 CCScene 的一个子类，然后在场景的 init 方法中创建层和其他对象。

有一种情况，你可能想在一个场景中使用多个层。如果你有一个滚动的背景，但是在背景四周又有一个静态的边框，也许还有一些用户界面元素。使用两个层可以方便地通过调整背景层的位置来移动它，而此时前景层依然在原处。而且，同一层的所有对象要么全部在另一层所有对象的上层，要么全部在另一层所有对象的下层，这取决于层的 z-order 属性的值。当然，不使用多个层也可以达到这个效果，只要对每个对象分别进行移动即可。显然，不使用层的做法是非常低效的。

与场景一样，层是没有维度的。层的本质是对节点进行分组。比如说，你可以对某一个层施加某个动作，然后这个动作会对该层上的所有对象产生影响。也就是说，你可以统一地移动一个层上的所有对象，还可以统一地对它们进行翻转和缩放。总的来说，当你需要对一组对象施加相同的动作或行为时，可以用层很方便地达到这个效果。比如说，让所有对象一起滚动，有时候你可能想要旋转它们，或者对它们重新排序，进而把它们绘制在其他对象的上面。如果这些对象都是某一个层的子节点，那么可以很轻松地通过修改层的属性或者在该层上运行一个动作来影响层上的所有子节点。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

注意：

有人建议不要在一个场景中使用过多 CCLayer 对象。这是一个误解。你可以使用任意多的层，与其他节点相比，它们对于性能并没有特别大的影响。不过，如果层能够接收用户输入，情况就有所变化了。因为接收触摸或加速计事件的开销是很大的。所以，不要使用太多层来接收触摸或加速计输入。可以只用一个层来接收和处理这些输入，然后在必要情况下，通过这个层将输入事件通知给其他节点或类。

1. 接收触摸事件

CCLayer 类是被设计为可以接收触摸输入的，但是需要显式地启用这个功能。这可以通过把 isTouchEnabled 属性设为 YES 来实现：

```
self.isTouchEnabled = YES;
```

这个操作最好在类的 init 方法中完成，但是可以在其他任何地方对其进行修改。

一旦 isTouchEnabled 属性被设好，许多用来接收触摸输入的方法就都会被调用。以下列出触摸开始、手指在屏幕上移动以及手指离开屏幕时会接收到的事件。很少会发生触摸被取消的情况，所以在大多数情况下，可以忽略这种情况，或者用 ccTouchesEnded 方法来处理。

- 当单指接触到屏幕时被调用：

```
-(void) ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent*)event
```

- 当手指在屏幕上移动时被调用：

```
-(void) ccTouchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
```

- 当单指离开屏幕时被调用：

```
-(void) ccTouchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
```

- 当触摸取消时被调用：

```
-(void) ccTouchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
```

取消事件一般很少发生，所以大多数情况下它的行为和触摸结束类似。

很多情况下，需要知道触摸在什么位置发生。由于触摸时间是由 Cocoa Touch API 接收的，因此一定要把触摸位置转换为 OpenGL 坐标。下面的方法为你实现了这个功能：

```
-(CGPoint) locationFromTouches:(NSSet *)touches
{
        UITouch *touch = [touches anyObject];
       CGPoint touchLocation = [touch locationInView: [touch view]];
        return [[CCDirector sharedDirector] convertToGL:touchLocation];
}
```

这个方法仅对单指触摸有效，因为它使用了[touches anyObject]。要追踪多指触摸的位置，你必须分别追踪每个触摸的位置。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

默认情况下，层接收到的事件和苹果公司的 UIResponder 类类似。cocos2d 也支持有针对性的触摸处理。区别在于：有针对性的触摸一次只接收单点触摸，而 UIResponder 触摸事件总是能接收多点触摸。有针对性的触摸处理只是简单地把多指触摸分割成了独立的事件，这样可以方便你更灵活地针对游戏需求去进行处理。更重要的是，它允许你把特定的触摸事件从事件队列中移除，可以指定一个触摸事件为已处理事件，并阻止将它传送给其他层。这样一来，如果触摸集中在一块特定区域内，你就可以很快识别出来，识别并处理完以后，可以把这些触摸标记为已处理，其他层就都不会再对这个区域做检测。

可以通过向层中添加以下方法来启用有针对性的触摸处理：

```
-(void) registerWithTouchDispatcher
{
        [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                              priority:INT_MIN+1 swallowsTouches:YES];
}
```

注意：

如果把 registerWithTouchDispatcher 方法保持为空，就不会接收到任何触摸事件了！如果既想保留此方法又，想使用默认处理方式，就必须在这个方法中调用 [super registerWithTouchDispatcher]。

现在，将使用一系列与默认触摸输入方法稍有不同的方法。它们唯一的区别在于：默认方法使用“(NSSet *)touches”作为第一个参数，而你要用的方法是以“(UITouch *)touch”作为第一个参数：

```
-(BOOL) ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event {}
-(void) ccTouchMoved:(UITouch *)touch withEvent:(UIEvent *)event {}
-(void) ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event {}
-(void) ccTouchCancelled:(UITouch *)touch withEvent:(UIEvent *)event {}
```

这里要注意的是，ccTouchBegan 返回一个布尔值。如果返回 YES，就意味着你不想让这个触摸被传送到其他优先级更低的有针对性的触摸处理。这相当于你“吞掉”了这个触摸事件。

2. 接收加速计事件

类似于触摸输入，需要显式地启动加速计来接收加速计事件：

```
self.isAccelerometerEnabled = YES;
```

同样地，还需要向层中添加一个特定方法来接收加速计事件：

```
-(void) accelerometer:(UIAccelerometer *)accelerometer

didAccelerate:(UIAcceleration *)acceleration
{
        CCLOG(@"acceleration: x:%f / y:%f / z:%f",
                   acceleration.x, acceleration.y, acceleration.z);
}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

可以使用 acceleration 参数来决定三维中任意方向的加速度。

3.7 CCSprite 类

CCSprite 是 cocos2d 中最为常用的类，它用一幅图像将精灵显示到屏幕上。要创建一个精灵，最简单的方法就是为精灵指定一个图像文件，而这个图像文件会在 cocos2d 内部被加载到 CCTexture2D 类的图像资源中。为了保证图像文件能够被成功读取，必须把图像文件加到 Xcode 中的 Resources 分组下：

```
CCSprite* sprite = [CCSprite spriteWithFile:@"Default.png"];
[self addChild:sprite];
```

cocos2d 中精灵的定位又有何玄机呢？与一些常见的游戏引擎不同，cocos2d 中精灵的 position 属性指定了图像中心所在的位置。一个刚被初始化的精灵，它的位置默认为(0，0)，此时该精灵就会被放置在屏幕左下角。因为精灵的 position 属性是指图像中心所在的位置，所以该图像只显示其右上角的部分。假设该图像的大小为 80×30 像素，那么必须把精灵的 position 属性设置为(40, 15)才能保证图像在屏幕左下角完全显示。

也许初看起来有些别扭，可是把图像中心作为定位点这个设计确实是非常精妙的。当你使用旋转或缩放属性的时候，可以保证精灵的中心点不会发生移动。

警告：

iOS 设备上的文件名是区分大小写的！而模拟器上的文件名并不区分大小写，所以当你改用真机进行测试时，很可能因为文件名而引发程序崩溃。以上面的代码为例，假如实际文件名为@"default.PNG"，那么用真机调试时就会崩溃。

这是让很多开发者感到头痛的问题，也是为什么需要经常拿真机进行测试的另一个原因。有一种好方法是：设定一个文件命名规范，并严格遵守。就我个人而言，我习惯用小写的文件名，然后用下划线来对单词进行分隔。

3.7.1 定位点揭秘

每个节点都有 anchorPoint(定位点)属性，但是这个属性只在节点需要用纹理来显示时才会发挥作用。任意节点的 anchorPoint 属性默认值都是(0.5, 0.5)或图像尺寸的一半。可以说，它是一个乘数或因子，而并非一个确切的像素值。

有些开发者在修改 anchorPoint 属性的值以后，会发现精灵在屏幕上发生了移动，因而他们认为 anchorPoint 属性会对精灵节点的位置产生影响。其实不然，事实上，精灵节点的位置并没有发生改变，改变的只是精灵纹理的位置！

anchorPoint 属性仅仅决定了纹理相对于精灵节点所在位置的偏移。偏移值是由纹理的实际大小乘以 anchorPoint 属性的值得出的。CCSprite 有一个只读的 anchorPointInPixels 属性，所以你不必手工计算具体的偏移值，仅在需要时修改 anchorPoint 属性的值即可。

如果把 anchorPoint 属性设为(0, 0)，那么纹理的左下角就会贴在节点的 position 属性指定的位置上。以下列代码为例，精灵的图像恰好可以完整地贴在屏幕的左下角：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
CCSprite* sprite = [CCSprite spriteWithFile:@"Default.png"];
sprite.anchorPoint = CGPointMake(0, 0);
[self addChild:sprite];
```

注意：

也许有人为了迎合多年的编程习惯，把所有精灵的 anchorPoint 属性都设为(0, 0)，但这只会在后期带来更多麻烦而已。如果所有精灵的定位点都在屏幕左下角，那么涉及旋转、缩放、子节点相对位置、距离检查、碰撞检测等问题时，情况会变得非常复杂。相信我，你一定会需要把定位点设置在纹理中心的。

3.7.2 纹理大小

纹理大小的问题非常值得一提。到目前为止，iOS 设备只支持尺寸为“2 的 n 次幂”的纹理，所以每张纹理的宽度和高度只可能为：2、4、8、16、32、64、128、512 和 1024 像素，在 3G 设备上可能还有 2048 像素。纹理可以不是方形的，所以，即使是 8×1024 像素的纹理用起来也不会有任何问题。

无论在什么情况下需要制作纹理(比如说要通过图像文件创建一个精灵)，都应该考虑到上述的尺寸要求。我们来设想一个最坏的情形：假设有一幅大小为 260×260 像素的 32 位色图像，它在内存中应该占用 270KB，但是因为受纹理的尺寸限制，它实际占用了 1MB 的内存。

实际占用的内存竟是所需内存的 4 倍之多！由于纹理的尺寸必须是 2 的次幂，因此系统会自动生成一张最接近原图尺寸(但是能包含原图像)的、长宽都为 2 的次幂的纹理。以 260×260 像素的图像为例，系统将生成一张 512×512 像素的纹理，所以会占用 1MB 的内存。

对于这种尴尬的情况，唯一的解决方法就是重新创建一张尺寸为 2 的次幂的纹理。只要把 260×260 像素的图像改为 256×256 像素，就不会导致大量的内存浪费了。如果你是在与一个艺术家合作，那么把这个规定也告诉她吧！

在第 6 章，我会教你如何创建并使用纹理图册来尽可能地解决这个问题。

3.8 CCLabel 类

CCLabel 可以帮你方便地在屏幕上显示文本。以下代码演示了如何用一个 CCLabel 对象来显示文本：

```
CCLabel* label = [CCLabel labelWithString:@"text" fontName:@"AppleGothic"
                 fontSize:32];
[selfaddChild:label];
```

在本章的随书项目“Essentials”中，我列出了一系列可以在 iOS 设备上使用的 TrueType 字体。

cocos2d 内部会以指定的字体作为参数创建一个 CCTexture2D 对象，也就是一张纹理，

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

然后再由该纹理渲染出最后显示的文本。每次文本发生改变，就要做一次上述工作，所以最好不要频繁改变文本(比如说，不要每过一帧就改一下文本)。重建一个标签文本的纹理是非常费时的。

```
[label setString:@"new text"];
```

你会发现，每次修改标签上的文本时，这些文本都会自动显示在标签的中央。如果要让文本相对于标签左对齐、右对齐、上对齐或下对齐，只须修改 anchorPoint 属性就可以轻松做到。以下代码演示了修改方法：

```
// align label to the right
label.anchorPoint = CGPointMake(1, 0.5f);
// align label to the left
label.anchorPoint = CGPointMake(0, 0.5f);
// align label to the top
label.anchorPoint = CGPointMake(0.5f, 1);
// align label to the bottom
label.anchorPoint = CGPointMake(0.5f, 0);

// use case: place label at top-right corner of the screen
// the label's text extends to the left and down and is always completely
// on screen
CGSize size = [[CCDirector sharedDirector] winSize];
label.position = CGPointMake(size.width, size.height);
label.anchorPoint = CGPointMake(1, 1);
```

3.9 菜单

有时候可能需要提供一些按钮来让用户进入另一个场景，或是关掉背景音乐等。这时 CCMenu 类就可以帮上忙了。CCMenu 只能接受 CCMenuItem 对象作为其子节点。

代码清单 3-2 演示了创建菜单的方法。你可以在“Essentials”项目的 MenuScene 类中找到有关菜单的代码。

代码清单 3-2 在 cocos2d 中用 Text 和 Image 菜单项创建菜单

```
CGSize size = [[CCDirector sharedDirector] winSize];

// set CCMenuItemFont default properties
[CCMenuItemFont setFontName:@"Helvetica-BoldOblique"];
[CCMenuItemFont setFontSize:26];

// create a few labels with text and selector
CCMenuItemFont* item1 = [CCMenuItemFont itemFromString:@"Go Back!" target:self
              selector:@selector(menuItem1Touched:)];
// create a menu item using existing sprites
CCSprite* normal = [CCSprite spriteWithFile:@"Icon.png"];
normal.color = ccRED;
CCSprite* selected = [CCSprite spriteWithFile:@"Icon.png"];
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
selected.color = ccGREEN;
CCMenuItemSprite* item2 = [CCMenuItemSprite itemFromNormalSprite:normal
             selectedSprite:selected target:self
             selector:@selector(menuItem2Touched:)];

// create a toggle item using two other menu items (toggle works with images, too)
[CCMenuItemFont setFontName:@"STHeitiJ-Light"];
[CCMenuItemFont setFontSize:18];
CCMenuItemFont* toggleOn = [CCMenuItemFont itemFromString:@"I'm ON!"];
CCMenuItemFont* toggleOff = [CCMenuItemFont itemFromString:@"I'm OFF!"];
CCMenuItemToggle* item3 = [CCMenuItemToggle itemWithTarget:self
              selector:@selector(menuItem3Touched:)
              items:toggleOn, toggleOff, nil];

// create the menu using the items
CCMenu* menu = [CCMenu menuWithItems:item1, item2, item3, nil];
menu.position = CGPointMake(size.width / 2, size.height / 2);
[self addChild:menu];

// aligning is important, so the menu items don't occupy the same location
[menu alignItemsVerticallyWithPadding:40];
```

警告：

menuWithItems(菜单项列表)总是以 nil 作为最后一个参数。这是一个技术要求，如果忘记加上 nil，你的程序可能会在这一行代码崩溃。

创建一个菜单确实需要很多代码。上述例子中的第一个菜单项是一个 CCMenuItemFont 对象，它仅用于显示一个字符串。单击该菜单项，方法 menuItem1 就会被调用。其实，CCMenuItemFont 类的内部只是简单创建了一个 CCLabel 对象。如果你已经有一个 CCLabel 对象，那么可以使用 CCMenuItemLabel 类来创建菜单项。

同样，用于显示图像的菜单项也有两个类：一个是 CCMenuItemImage，它根据文件名参数创建图像，其实它的内部就是采用 CCSprite 类来实现的；另一个就是在上面示例中用到的 CCMenuItemSprite，它是以现成的精灵对象作为输入的，我觉得这个类用起来比 CCMenuItemImage 简单，只要对精灵的颜色稍加调整，甚至不需要加入新的图像资源就可以打造出按钮被单击的高光效果了。

CCMenuItemToggle 类只接受两个继承自 CCMenuItem 的对象作为其参数，单击菜单项，该菜单项的状态就会在由参数指定的两个状态间进行切换。这两个参数可以是文本标签，也可以是图像。

最后一步是创建 CCMenu，并为它定位。由于所有菜单项都是该菜单的子节点，因此它们的位置都是根据到菜单的相对距离得出的。为了避免这些菜单项叠加在一起，你可以调用 CCMenu 一个的对齐方法，比如我在代码清单 3-2 中最后使用的 alignItemsVertically-WithPadding。

由于 CCMenu 是一个包含所有菜单项的节点，因此可以对这个菜单施加动作来让它进行滚动。这样一来，整个菜单场景就不会显得那么呆滞。可以参考“Essentials”项目中的示例。我们通过图 3-4 来看一下到目前为止菜单上有多少内容。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

图 3-4　这是代码清单 3-2 生成的菜单截图。单击菜单时，文本标签的大小会发生变化，精灵的颜色也会不停变换

3.10　动作

动作是轻量级的类，可以用来让节点执行诸如移动、旋转、缩放、变色、消失等很多动作。由于它们能作用在所有节点上，因此可以对精灵、标签甚至菜单或整个场景施加动作。这就是动作的强大之处。

由于动作会执行一段时间，如旋转 3 秒钟，因此通常的做法是写一个更新方法，并加一些变量来存储中间结果。动作封装了这部分的逻辑，并将之转换成了带参数的方法：

```
// I want myNode to move to 100, 200 and arrive there in 3 seconds
CCMoveTo* move = [CCMoveTo actionWithDuration:3 position:CGPointMake(100, 200)];
[myNode runAction:move];
```

动作分为两大类：瞬时动作基本上等同于设置节点的属性，如 visible 和 flipX；延时动作会执行一段时间，如上面的移动动作。另外，不需要清除这两种动作。一旦动作完成，就会自动从节点上清除并释放它所占用的内存。

3.10.1　重复动作

可以使一个动作或一系列的动作不停地重复，用这个方法可以创建无限循环的动画。下面这段代码让一个节点像轮子一样不停旋转：

```
CCRotateBy* rotateBy = [CCRotateBy actionWithDuration:2 angle:360];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:rotateBy];
[myNoderunAction:repeat];
```

3.10.2　流畅动作

CCEaseAction 类的出现使得动作显得更为强大了。流畅动作允许你改变在一段时间内的动作效果。例如，如果你对一个节点使用 CCMoveTo，它会匀速移动到目标点。如果使用 CCEaseAction，就可以让节点由慢到快或由快到慢地移向目标。或者让节点移过目的地一些，再弹回来。流畅动作能帮你创建通常要花很长时间才能做出来的动画。下面这段代

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

码展示了如何使用流畅动作来改变一个普通动作的行为(rate 参数决定了流畅动画的明显程度，只有当它大于 1 时才能看到效果)：

```
// I want myNode to move to 100, 200 and arrive there in 3 seconds
CCMoveTo* move = [CCMoveTo actionWithDuration:3 position:CGPointMake(100, 200)];
// this time the node should slowly speed up and then slow down as it moves
CCEaseInOut* ease = [CCEaseInOut actionWithAction:move rate:4];
[myNode runAction:ease];
```

注意：

在本例中，流畅动作运行在节点上而不是移动动作上。当你在使用动作时，很容易忘记修改 runAction 那行代码，即使是最有经验的 cocos2d 开发者也会犯这样的错误。如果发现动作并未像预期的那样执行，仔细检查下你是否执行了正确的动作。如果执行的的确是正确的动作，但依然没看到期望的结果，就再检查下运行的节点是否正确。这又是另一个常见的错误。

cocos2d 实现了如下几个 CCEaseAction 类：

- CCEaseBackIn、CCEaseBackInOut、CCEaseBackOut
- CCEaseBounceIn、CCEaseBounceInOut、CCEaseBounceOut
- CCEaseElasticIn、CCEaseElasticInOut、CCEaseElasticOut
- CCEaseExponentialIn、CCEaseExponentialInOut、CCEaseExponentialOut
- CCEaseIn、CCEaseInOut、CCEaseOut
- CCEaseSineIn、CCEaseSineInOut、CCEaseSineOut

我会在第 4 章的 DoodleDrop 项目中使用到这里的一些流畅动作，到时候你就能看到它们的效果了。

3.10.3　动作序列

通常情况下，当向一个节点添加几个动作时，它们会在同一时间运行。例如，可以通过添加相应的动作让一个节点一边旋转一边消失。但如果想让动作一个接一个运行呢？

有时候将动作排成序列会更有用，而这正是 CCSequence 的功能。在一个动作序列中，你可以使用任意数量和类型的动作。例如，将一个节点移动到目的地，然后旋转一圈，最后再消失。动作是一个接一个运行的，直到整个序列全部完成。

以下代码演示了如何让一个标签的颜色从红变成蓝，再变成绿：

```
CCTintTo* tint1 = [CCTintTo actionWithDuration:4 red:255 green:0 blue:0];
CCTintTo* tint2 = [CCTintTo actionWithDuration:4 red:0 green:0 blue:255];
CCTintTo* tint3 = [CCTintTo actionWithDuration:4 red:0 green:255 blue:0];
CCSequence* sequence = [CCSequence actions:tint1, tint2, tint3, nil];
[label runAction:sequence];
```

你也可以将动作序列和 CCRepeatForever 动作一起使用：

```
CCSequence* sequence = [CCSequence actions:tint1, tint2, tint3, nil];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:sequence];
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
[label runAction:repeat];
```

> 注意：
> 就像使用菜单项那样，一串动作总是以 nil 结尾。如果你忘记在最后一个参数位置加上 nil，创建 CCSequence 的这行代码就会崩溃！

3.10.4 瞬时动作

瞬时动作可用于翻转节点、移动节点以及设置节点的可视性等。读到这里，读者可能会感到奇怪：这些动作都可以通过修改节点属性来完成，那么瞬时动作的存在又有什么意义呢？

其实，正是动作序列使得这些瞬时动作变得有意义了。有时候在一串动作中，你需要改变节点的某个属性，如可视性和位置，然后再继续运行序列。如何在动作序列中修改这些属性呢？这时候瞬时动作就派上了用场。确实，瞬时动作通常只是用于 CCCallFunc 动作。

使用动作序列时，你也许想在特定时刻收到通知。例如，一个序列完成，紧接着要开始另一个序列。可以使用 CCCallFunc 的 3 个版本来达到这个效果，它们会在序列中轮到自己的时候发出消息。我们来重写一下颜色变换序列，它会在每次 CCTintTo 动作完成后调用一个方法：

```
CCCallFunc* func = [CCCallFunc actionWithTarget:self
                   selector:@selector(onCallFunc)];
CCCallFuncN* funcN = [CCCallFuncN actionWithTarget:self
                   selector:@selector(onCallFuncN:)];
CCCallFuncND* funcND = [CCCallFuncND actionWithTarget:self
                   selector:@selector(onCallFuncND:data:) data:(void*)self];
CCSequence* seq = [CCSequence actions:tint1, func, tint2, funcN, tint3,
                  funcND, nil];
[label runAction:seq];
```

这个序列会一个接一个地调用下列方法(sender 参数总是继承自 CCNode，它是运行这些动作的节点；data 参数可供你任意使用，你可以用它传递值、结构体或其他指针，只要能正确地转换 data 指针的类型即可)：

```
-(void) onCallFunc
{
        CCLOG(@"end of tint1!");
}

-(void) onCallFuncN:(id)sender
{
        CCLOG(@"end of tint2! sender: %@", sender);
}

-(void) onCallFuncND:(id)sender data:(void*)data
{
        // be careful when casting pointers like this!
        // you have to be 100% sure the object is of this type!
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        CCSprite* sprite = (CCSprite*)data;
        CCLOG(@"end of sequence! sender: %@ - data: %@", sender, sprite);
}
```

当然，CCCallFunc 也能和 CCRepeatForever 序列配合使用，你的方法会在适当的时候被反复调用。

3.11 cocos2d 测试案例

cocos2d 自带许多测试案例。在 cocos2d-iphone 文件夹中有一个名为 cocos2d-iphone 的项目，里面包含了许多测试案例可供构建和运行。可以先查看它们的效果，然后阅读代码以了解它们的实现原理。

3.12 本章小结

本章讨论了好多知识！一下子记住这么多内容很难，可以随时回过头来查阅 cocos2d 的场景图以及各种 CCNode 类的使用方法。本章的目的就是为了方便读者在需要时进行参考。

通过本章的学习，再加上你的激情，你已经可以自己写游戏了！

在下一章中，我会和你一起制作一个完整的游戏项目！