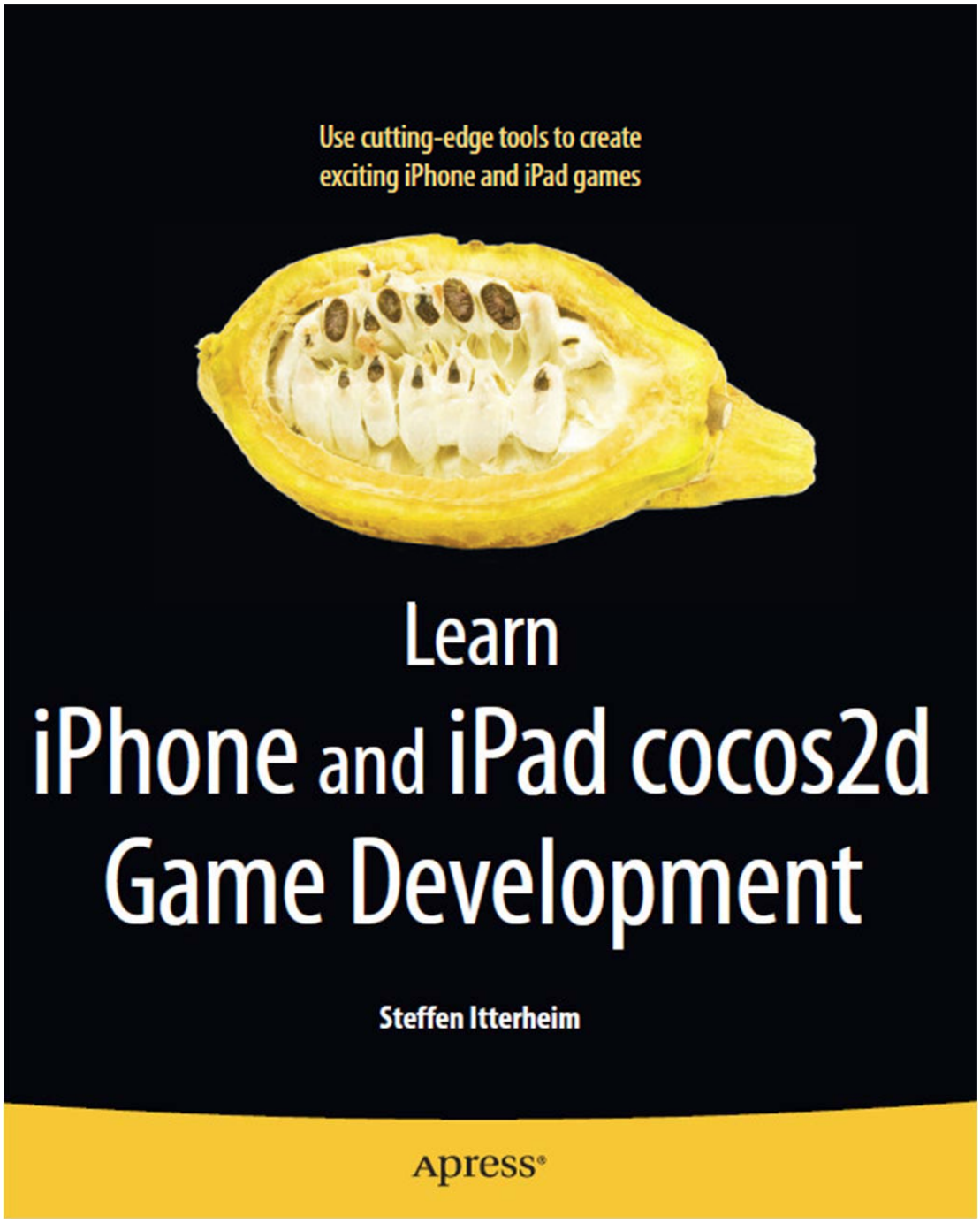

译者：杨栋
邮箱：yangdongmy@gmail.com

第三章
基础知识

本章将会为你介绍cocos2d游戏引擎的构成要素。你将会在每一个编写的游戏里用到这些类，所以了解它们是些什么样的类和这些类是如何在一起工作的，将会帮助你写出更好的游戏。有了这些知识，你会发现使用cocos2d很容易。

配套本章的Xcode项目叫作 “Essentials”。它包含了所有在此讨论的内容，外加一些额外的例子。源代码附带详尽的注释，所以读起来就像本书的附录。

我们将以讨论cocos2d游戏引擎结构作为开始。每个游戏引擎在管理和呈现屏幕上的游戏对象的方式都是不一样的。所以一开始我们就要了解有哪些元素存在以及它们之间的关系。

cocos2d的单例

cocos2d很好的利用了单例设计模式。因为此模式经常引起争议，所以我想有必要在此解释一下单例。原则上，单例是在程序生命周期里只被实例化过一次的类。为了确保这一点，我们利用类的一个静态方法来生成和访问对象。因此，你是通过以“shared”开头的方法来访问cocos2d的单例对象的，而不是用alloc/init或者静态autorelease初始化方法。以下是一些最常用到的cocos2d单例类和访问它们的方法：

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

单例的好处是它可以在任何时间任何地点被任何类所调用。它接近于全局类的作用，更像一个全局变量。如果你需要在任何地方都能用到某些数据或者方法，单例是很好的选择。音频就是个很好的例子：因为任何一个类，不管是玩家，敌人，菜单按钮，或是过场动画，都可能需要播放声效或者改变背景音乐。因此，使用单例来播放音频是很好的选择。同样，如果存在全局的游戏状态，比如说玩家军队的大小和每支部队排的数目，你可以把这些信息存到一个单例中，把这些信息从一个关卡传到另一个关卡。**列表3-1**演示了如何实现单例。这些代码使用了最少的代码实现了MyManager类的单例。SharedManager提供了访问MyManager单一实例的静态方法。如果实例不存在，一个MyManager的实例将会被分配和初始化；否则已经存在的实例会被返回。

列表3-2：MyManager类实现单例

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

不过，单例也有不好的方面。因为单例很容易实现，而且可以在任何地方访问到，它们可能会被用在不该用的地方。

例如，你可能觉得你的游戏只有一个玩家对象，所以为什么就不能把玩家这个类变成单例呢？一切看起来都没有什么问题 - 直到你认识到不管什么时候这个玩家进入下一个关卡，这个玩家不仅带着上一关卡的得分，而且还有上一关卡的最后一帧动画信息，健康值，和所有已经捡到的物品，并且由于他在离开上一关卡的时候还在“狂暴”状态下，在新关卡开始的时候，他还处于之前的状态中。

为了解决这个问题，你可能会在类里加入一个重置某些变量的方法。看起来问题解决了。但是当你在游戏代码中添加越来越多的功能以后，在转换关卡时，你需要维护的变量也会越来越多。最糟糕的是，某天你的朋友建议你为iPad版本增加个双人模式。但是你发现你的玩家类是个单例，在任何时候你只能有一个玩家对象存在！这可麻烦了：要么你要重写很多代码，或者只能放弃很酷的双人模式了。

你越依赖于单例，类似的问题就会越多。在创建任何一个单例类之前，你都要考虑是否真的需要单例，是否需求会在不久的将来改变。

The Director（导演）

CCDirector类，简称Director（导演），是cocos2d游戏引擎的核心。如果你回想一下第二章的HelloWorld应用，你会记得有很多cocos2d的初始化过程包含了[CCDirector sharedDirector]的调用。Director是一个单例：它保存着cocos2d的全局配置设定，同时管理着cocos2d的场景。

Director的主要用处如下：

1. 访问和改变场景
2. 访问cocos2d的配置细节
3. 访问视图（OpenGL，UIView，UIWindow）
4. 暂停，恢复和结束游戏
5. 在UIKit和OpenGL之间转换坐标

实际上存在四种类型的Director。它们在细节上有所不同。最常用的Director是CCDisplayLinkDirector，它的内部使用了苹果的CADisplayLink类。它是最好的选择，但是只有在iOS 3.1以上的版本中才能使用。其次，你可以使用CCFastDirector。如果你想让Cocoa Touch视图和cocos2d一同工作，你必须转到CCThreadedFastDirector，因为只有这个Director才能完全支持。CCThreadedFastDirector不好的一面是：使用它会很耗电。最后的选择是CCTimerDirector，但这是没有办法的选择，因为它是四种Director里面最慢的。

场景图(The Scene Graph)

有时候又被称为“场景层级”。场景图是由所有目前活跃的cocos2d节点所组成的一个层级图。除了场景本身，每一个节点只有一个父节点，但是可以有任意数量的子节点。

当你将节点添加到其它节点中时，你就在构建一个节点场景图。**图3-1**描绘了一个虚构的游戏场景图。在最上面，你总是放置场景节点(MyScene)，通常跟着的是一个层节点（MyLayer）。在cocos2d里，层节点的作用是接收触摸和加速计的输入。

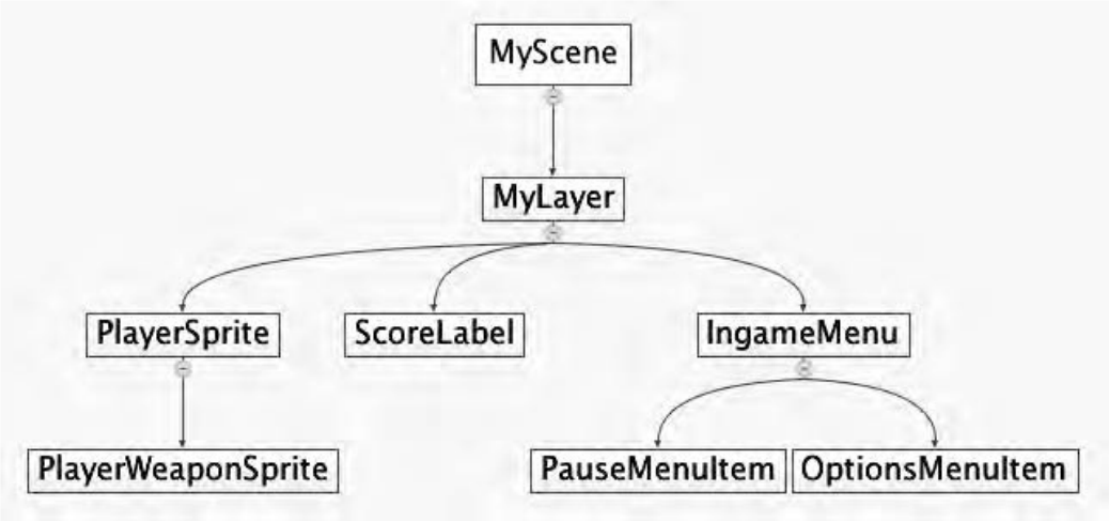

图3-1. 一个简化的由多个不同节点组成的cocos2d场景图。场景图中有一个玩家节点和他的武器节点，游戏的得分，和游戏中用于暂停和改变游戏选项的菜单。

在CCLayer下一层的是你游戏的组成要素，它们大多数是精灵（sprite）节点。它们包括用于显示游戏得分的标签节点，用于显示游戏内菜单的菜单和菜单项目节点，玩家用这些菜单来暂停游戏或者回去主菜单。

在**图3-1**中你会注意到PlayerSprite节点中有个子节点PlayerWeaponSprite。换句话说，PlayerWeaponSprite是附加在PlayerSprite上的。如果PlayerSprite移动，旋转或放大缩小，PlayerWeaponSprite将会跟着做同样的事情而不需要额外的代码。这就是场景图的强大之处：你对一个节点施加的影响将会影响到它的所有子节点。但是有时候这也会产生混淆，因为像位置和旋转都是相对于父节点来说的。

我写了一个叫作“NodeHierarchy”的Xcode样例，你可以在本书提供的源代码

里找到。它演示了在一个层级关系里的节点是如何相互影响的。我想实际的例子比用文字和图片说明要来的更直观和容易理解。

CCNode类的层级.

所有节点都有一个共同的父类：CCNode。它定义了许多除显示节点外的通用的属性和方法。图3-2展示了继承自CCNode的一些最重要的类。这些类是你最常用到的。其实即使你只用这些类，你也可以创造出很有意思的游戏。

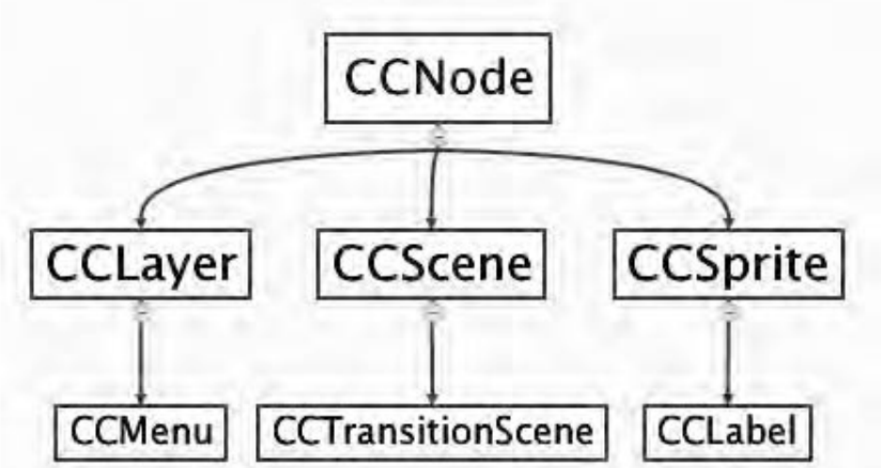

图3-2. CCNode是cocos2d中最重要的类。所有类都继承自CCNode。CCNode定义了通用的属性和方法。

CCNode

CCNode是所有节点的基类。它是一个抽象类，没有视觉表现。它定义了所有节点都通用的属性和方法。

使用节点

CCNode类实现了所有添加，获取和删除子节点的方法。以下是一些处理子节点的方法：

1. 生成一个新的节点 ：

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

2. 将新节点添加为子节点：

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

3. 获取子节点：

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

4. 通过tag删除子节点；cleanup会停止任何运行中的动作：

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

5. 通过节点指针删除节点:

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

6. 删除一个节点的所有子节点：
   [myNode removeAllChildrenWithCleanup:YES];

7. 从myNode的父节点删除myNode：
   [myNode removeFromParentAndCleanup:YES];

addChild中的z参数决定了节点的绘制顺序。拥有最小z值的节点会首先被绘制；拥有最大z值的节点最后一个被绘制。如果多个节点拥有相同的z值，他们的绘制顺序将由他们的添加顺序来决定。当然，这个规则只适用于像sprites那样有视觉表现的节点。

tag参数允许你通过getChildByTag方法来获取指定的节点。

注：如果有多个节点拥有相同的tag数值，getChildByTag将把找到的第一个节点返回。其它节点将不能够再被访问。所以你要确保为你的节点指定独有的tag数值。

动作（Actions）也有tag。不过，节点和动作的tag不会冲突，所以拥有相同tag数值的动作和节点可以和平共处。

使用动作（Actions）

节点可以运行动作。我会在以后多讲一些动作相关的知识。现在你只要知道动作可以让节点移动，旋转和缩放，还可以让节点做一些其它的事情。

1. 以下是一个动作的声明：
   ```
   CCAction* action = [CCBlink actionWithDuration:10 blinks:20];
   action.tag = 234;
   ```

2. 运行这个动作会让节点闪烁：
   ```
   [myNode runAction:action];
   ```

3. 如果你想在以后使用此动作，你可以用tag获取：
   ```
   CCAction* retrievedAction = [myNode getActionByTag:234];
   ```

4. 你可以用tag停止相关联的动作：
   ```
   [myNode stopActionByTag:234];
   ```

5. 或者你也可以用动作指针停止动作：
   ```
   [myNode stopAction:action];
   ```

6. 你可以停止所有在此节点上运行的动作：
   ```
   [myNode stopAllActions];
   ```

预定信息

节点可以预定信息，其实就是Objective-C里面的每隔一段时间调用一次方法。在很多情况下，你需要节点调用指定的更新方法以处理某些情况，比如说碰撞测试。以下是一个最简单的，可以在每一帧都被调用的更新方法：

```
-(void) scheduleUpdates
{
	[self scheduleUpdate];
}
-(void) update:(ccTime)delta
{
	// 此方法每一帧都会被调用
}
```

很简单不是吗？你会注意到我们现在的更新方法是固定的，每一帧都会调用上述方法。delta这个参数表示的是此方法的最后一次调用到现在所经过的时间。如果你想每一帧都调用相同的更新方法，上述做法很适用。不过有时候你需要用到更灵活的更新方法。

如果你想运行不同的方法，或者是每秒调用10次更新方法的话，你应该使用以下代码：

```
-(void) scheduleUpdates
{
	[self schedule:@selector(updateTenTimesPerSecond:) interval:0.1f];
}
-(void) updateTenTimesPerSecond:(ccTime)delta
{
	// 此方法将根据时间间隔来调用，每秒10次
}
```

如果时间间隔（interval）为0的话，你应该使用scheduleUpdate方法。不过，如果你想之后停止对某个指定更新方法的预定信息的话，上述代码更加合适。因为scheduleUpdate方法没有停止预定信息的功能。

更新方法的签名和之前是一样的：delta时间是它唯一的参数。但是这次你可以使用任何名称，而且它会每十分之一秒被调用一次。如果你不想每一帧都判断是否达到了胜利的条件（有可能判断的过程很复杂），每秒调用10次更新方法会比每帧都调用要好。或者，你想让代码在10分钟以后调用运行一个动作，你可以将时间间隔（interval）设置为600。

注：@selector(...)这个语法看起来有点怪。这是Objective-C用来参照指定方法的方式。这里很重要的一点是最后的那个冒号。它告诉Objective-C去找在此指定的方法名，并且此方法只有一个参数。如果你忘记在最后加上冒号，程序还是会继续编译，但是之后会崩溃。在调试控制台（Debugger Console）里，你会看到这样的错误日志：“unrecognized selector sent to instance …”。

@selector(...)里的冒号数量必须和方法的参数数量和名称相匹配。看一下以

下方法：

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

相对应的@selector应该是：

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

通过你自己的选择器（selector）或者用@selector(...)关键词的方式来预定更新方法会有一个很大的问题。默认情况下，如果方法名不存在的话，编译器并不会报错，而是在方法被调用时直接导致程序崩溃。因为调用是在cocos2d内部进行的，所以很难发现问题的根源。幸运的是，有一个相关的编译器报警设置可以使用。图3-3显示“Undeclared Selector”设置已被勾选，示例项目“Essentials”里的这项设置也已被启用。

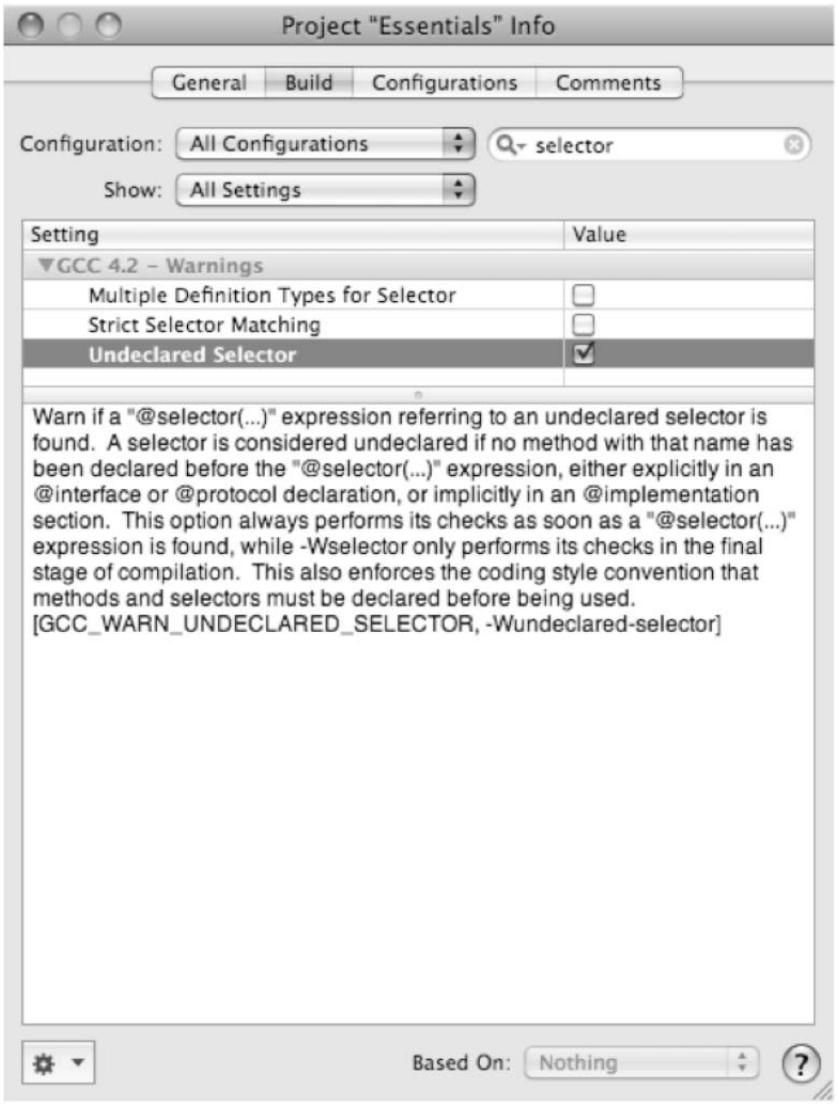

图2-3：启用构建设置（Build Setting）中的“Undeclared Selector”设置

接下去我们讨论如何停止对预定方法的调用。

以下代码会停止节点的所有选择器，包括那些已经在scheduleUpdate里面设置了预定的选择器：

```
[self unscheduleAllSelectors];
```

以下代码会停止某个指定的选择器（假设选择器名称是updateTenTimesPerSecond）：

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

注：此方法不会停止scheduleUpdate中设置的预定更新方法。

还有一个挺有用的设置和停止选择器预定的方法。很多时候你需要在设置好的预定方法里面停止调用某个指定的方法，同时因为参数和方法名可能发生变化，你又不想重复相同的方法名和参数，这时你可以用以下的方法设置（预定的控制器只会运行一次）：

```
-(void) scheduleUpdates
{
	[self schedule:@selector(tenMinutesElapsed:) interval:600];
}
-(void) tenMinutesElapsed:(ccTime)delta
{
	// 用_cmd关键词停止当前方法的预定
	[self unschedule:_cmd];
}
```

_cmd关键词是当前方法的缩写。上述代码只会让tenMinutesElapsed方法运行一次。实际上你也可以用_cmd来设置方法调用的预定。假设你需要调用一个方法，这个方法会使用不同的时间间隔来调用，每次方法被调用以后，时间间隔都会发生变化。你的代码看起来会是像下面这样：

```
-(void) scheduleUpdates
{
	// 像之前一样预定第一次更新
	[self schedule:@selector(irregularUpdate:) interval:1];
}
-(void) irregularUpdate:(ccTime)delta
{
	// 首先，停止方法调用的预定
	[self unschedule:_cmd];

	// 这里我们用随机数来决定下次调用此方法需要经过的时间
	float nextUpdate = CCRANDOM_0_1() * 10;

	// 然后用_cmd来代替选择器，用新的时间间隔来重新预定方法调用
	[self schedule:_cmd interval:nextUpdate];
}
```

用_cmd关键词可以让你避免预定（schedule）或者停止预定（unschedule）错

误的方法。从长期来说是很有好处的。

最后一个预定方法调用的问题是安排更新方法的优先次序。请先看一下以下代码：

```
// 在A节点里
-(void) scheduleUpdates
{
	[self scheduleUpdate];
}

// 在B节点里
-(void) scheduleUpdates
{
	[self scheduleUpdateWithPriority:1];
}

// 在C节点里
-(void) scheduleUpdates
{
	[self scheduleUpdateWithPriority:-1];
}
```

这可能需要些时间来消化。所有的节点还是在调用同样的-(void)update:(ccTime) delta方法。但是因为使用了优先级设置，C节点将会被首先运行。然后是调用A节点，因为默认情况下优先级设定为0。B节点最后一个被调用，因为它的优先级的数值最大。更新方法的调用次序是从最小的优先级数值到最大的优先级数值。

你可能想知道什么时候会用到这个优先级功能。坦率地说很少会用到它。不过按照我过去的经验，在某些极端情况下你可能需要用到这个功能，比如在进行物理效果模拟之前或者之后，为参与模拟的对象添加力量。在宣布此项功能的同时也提到了物理效果的更新说明了上述用处。有的时候，通常是在项目后期，你可能发现了一个很奇怪的bug，这个bug是和时间的选择（timing）有关的，这迫使你在完成所有的对象自我更新之后，运行玩家对象的更新方法。

直到你有一天需要用到优先级设置这个功能以解决特定的问题，现在你可以忽略它。

场景和层

CCNode，CCScene和CCLayer这些类是没有视觉表现的。它们是在内部作为场景图的抽象概念来使用的。CCLayer最典型的应用是把各个节点组织起来，还有接收触摸输入和加速计输入的信息 - 前提是上述接收功能已被启用。

CCScene

CCScene对象总是场景图里面的第一个节点。通常CCScene的子节点都是继承自CCLayer。CCLayer包含了各个游戏对象。因为大多数情况下场景对象本身不包含任何游戏相关的代码，而且很少被子类化，所以它一般都是在CCLayer对象里通过+(id)scene这个静态方法来创建的。我已经在第二章谈到过这个方法，但是我想通过以下代码来刷新一下你的记忆：

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

第一个创建场景的地方是在AppDelegate中aplicationDidFinishLaunching方法结束处。你在那里用Director的runWithScene方法开始运行第一个场景：

```
// 用以下代码运行第一个场景
[[CCDirector sharedDirector] runWithScene:[HelloWorld scene]];
```

在其它情况下，用replaceScene方法来替换已有的场景：

```
// 用replaceScene来替换所有以后需要变化的场景
[[CCDirector sharedDirector] replaceScene:[HelloWorld scene]];
```

注：你可以在HelloWorld场景里顺利运行这些代码。一个新的HelloWorld实例会被生成，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！

场景和内存

当你替换一个场景时，新场景被加载进内存，但是旧的场景还没有从内存中释放。这会让内存使用量在短时间内忽然增大。替换场景的过程很关键，因为很多时候你会因为系统内存不够而收到内存警告或者导致程序崩溃。如果你在开发过程中，发现游戏在场景转换过程中占用很多内存的话，你应该尽早和尽量多的进行测试。

注：在你替换场景的时候，cocos2d会把自己占用的内存清理干净。它会移除所有的节点，停止所有的动作，并且停止所有选择器的预定。我之所以提到这一点，是因为我有时候看到开发者会直接调用cocos2d的removeAll方法，那是没有必要的。你应该相信cocos2d的内存管理能力。

如果你在替换场景的时候使用过渡效果（transitions）的话，这个问题就更明显了。在过渡的过程中，新的场景首先被生成，然后运行过渡效果，只有在过渡效果完成以后，旧的场景才会被清理出内存。在创建场景的那个图层中添加日志可以帮助你更好的了解你的场景。

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

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

	// 总是在dealloc方法里调用[super dealloc]
	[super dealloc];
}
```

注意观察这些日志信息。如果你发现在场景转换过程中，dealloc里的日志信息没有被发送出去的话，你碰到了大麻烦！你的整个场景都在内存泄漏，应该释放的内存没有得到释放。这样的事情是不大可能由cocos2d本身导致的。大多数情况下是由于错误的retain或者没有释放节点。

有件事情你永远都不应该尝试，那就是首先把一个节点添加到场景中作为它的子节点，然后又自己把此节点retain下来。相反，你应该用cocos2d的方式来访问创建的节点，或者至少是弱引用节点指针，而不是直接retain节点。只要你让cocos2d来管理节点的内存使用，你就不会遇到麻烦。

推进（Pushing）和弹出（Popping）场景

在讨论转换场景的同时，我想提一下pushScene和popScene这两个来自Director的方法。有时候它们会有些用处。他们的作用是在不从内存里移除旧场景的情况下运行新的场景，目的是让转换场景的速度更快。但是这里有个问题：如果你的场景很简单，同时互相分享内存，那么它们本身的加载速度就很快。而如果你的场景很复杂，需要消耗很长时间加载，那么它们就会互相争抢宝贵的内存 - 导致内存使用量迅速上升。

pushScene和popScene最大的问题是可以互相叠加。你可以推进一个场景，同时运行一个新的场景。然后这个新场景推进另一个场景，而那个场景也会推进又一个场景。如果你没有管理好场景的推进和弹出，最终你会忘记弹出场景，或者将同一个场景弹出多次。更加糟糕的是这些场景都共享着同一块内存。

不过在有一个情况下pushScene和popScene很有用：如果你要在很多地方使用一个通用的场景，比如包含改变音乐和声音音量菜单的“设置场景”。你可以推进“设置场景”以显示它。“设置场景”的“回去”按钮则会调用popScene让游戏回到之前的场景。不管你是在主菜单，游戏中，或是其它一些地方打开“设置场景”，这个方法都能很好的工作。你从此不再需要跟踪“设置场景”最后一次是在哪里打开的。

不过，你还是需要测试“设置场景”在各种情况下的表现，以确定在任何情况下都有足够的内存可用。理想状态下，“设置场景”本身应该是很简单轻巧的。

用以下代码在任意一个地方显示“设置场景”：

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

如果你身处“设置场景”，但又想关闭“设置场景”时，你可以调用popScene。这样你会回到之前还保留在内存里的场景：

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

CCTransitionScene

所有过渡效果的类都继承自CCTransitionScene。

注：我先在这里警告一下：在游戏里不是每个过渡效果都很有用，即使它们看起来很好看。玩家们最关心的是过渡的速度。即使3秒钟他们都会觉得长。我设置过渡效果的时间不会超过一秒，或者干脆完全不用。

你绝对要避免在转换场景是随机选择过渡效果。玩家们不关心这些。而作为开发者，你可能对于过渡效果太兴奋了。如果你不清楚该为哪个场景转换使用哪个过渡效果，那就不要用。换句话说，可以使用并不代表你一定要用。

虽然过渡效果的名称和需要的参数数量很多，但是过渡效果只给场景转换代码增加了一行代码而已。以下是很流行的淡入淡出过渡效果：它在一秒内过渡到了白色：

```
// 用我们想要在下一步显示的场景初始化一个过渡场景
CCFadeTransition* tran = [CCFadeTransition transitionWithDuration:1
                                                    scene:[HelloWorld scene]
                                                    withColor:ccWHITE];

// 使用过渡场景对象而不是HelloWorld
[[CCDirector sharedDirector] replaceScene:tran];
```

你可以把CCTransitionScene与replaceScene和pushScene结合使用，但是你不能将过渡效果和popScene一起使用。

有很多种过渡效果可以使用，大多是和方向有关的，比如从哪个地方开始过渡到哪个地方过渡结束。以下是目前可以使用的过渡效果和描述：

1. **CCFadeTransition**: 淡入淡出到一个指定的颜色，然后回来。
2. **CCFadeTRTransition** (还有另外三个变化): 瓦片（tiles）反转过来揭示场景。
3. **CCJumpZoomTransition**: 场景跳动着变小，新场景则跳动着变大。
4. **CCMoveInLTransition** (还有另外三个变化): 场景移出，同时新的场景从左边，右边，上方或者下方移入。
5. **CCOrientedTransitionScene** (还有另外六个变化): 这种过渡效果会将整个场景翻转过来。
6. **CCPageTurnTransition**: 翻动书页的过渡效果。
7. **CCRotoZoomTransition**: 当前场景旋转变小，新的场景旋转变大。
8. **CCShrinkGrowTransition**: 当前场景缩小，新的场景在其之上变大。

9. CCSlideInLTransition（还有另外三个变化）：新的场景从左边，右边，上方或者下方滑入。
10. CCSplitColsTransition（还有另外一个变化）：将当前场景切成竖条，上下移动揭示新场景。
11. CCTurnOffTilesTransition:将当前场景分成方块，用分成方块的新场景随机的替换当前场景分出的方块。

CCLayer

有时候在同一个场景里你需要多个CCLayer。你可以参照以下代码生成这样的场景：

```
+(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;
}
```

另一个方式是通过创建CCScene的子类，然后在各个场景的init方法中生成CCLayer层和其它对象。

如果你有一个滚动的背景，背景上有个静止的框围绕着背景（上面可能包含一些用户界面元素），这种情况下你可能需要在同一个场景中使用多个层。通过使用两个分开的层，你可以调整背景层的位置来使其移动，同时前景层保持不动。另外，根据层的z-order属性的不同，同一层的物体要么在另一层物体的前面或者后面。当然，你也可以不用层而达到相同的效果。不过那样的话就要求背景上的各个物体要分开移动。这样做非常没有效率。

和场景一样，层没有大小的概念。层是一个组织的概念。比如，如果你对一个层使用动作，那么所有在这个层上的物体都会受到影响。这意味着你可以让同一层上的所有物体一起移动，旋转和缩放。通常，如果你想让一组物体执行相同的动作和行为，层是很好的选择。比如说让所有的物体一起滚动；有时候你可能想让他们一起旋转，或者将他们重新排列然后覆盖在其它物体上面。如果所有这些物体是同一个层的子节点，你就可以通过改变层的属性或者在层上执行动作，来达到影响层上所有子节点的目的。

注：有人建议不要在同一个场景里使用过多的CCLayer对象。这是一个误解。使用层和使用其它的节点一样，并不会因为使用多个层而降低运行效率。不过，如果你的层接收触摸或者加速计事件的话就不一样了。因为接收处理外来事件很耗费资源。所以，你不应该使用很多接收外来事件的层。比较好的处理方式

是：只使用一个层来接收和处理事件。如果需要的话，这个层应该通过转发事件的方式来通知其它节点或类。

接收触摸事件

CCLayer类是用来接收触摸输入的。不过你要首先启用这个功能才可以使用它。你通过设置isTouchEnabled为YES来让层接收触摸事件：

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

此项设定最好在init方法中设置。你可以在任何时间将其设置为NO或者YES。

一旦启用isTouchEnabled属性，许多与接收触摸输入相关的方法将会开始被调用。这些事件包括：当新的触摸开始的时候，当手指在触摸屏上移动的时候，还有在用户手指离开屏幕以后。很少会发生触摸事件被取消的情况，所以你可以在大多数情况下忽略它，或者使用ccTouchesEnded方法来处理。

1. 当手指首次触摸到屏幕时调用的方法：
   ```
   -(void) ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent*)event
   ```

2. 手指在屏幕上移动时调用的方法：
   ```
   -(void) ccTouchesMoved:(NSSet *)touches withEvent:(UIEvent*)event
   ```

3. 当手指从屏幕上提起时调用的方法：
   ```
   -(void) ccTouchesEnded:(NSSet *)touches withEvent:(UIEvent*)event
   ```

4. 当触摸事件被取消时调用的方法：
   ```
   -(void) ccTouchesCancelled:(NSSet *)touches withEvent:(UIEvent*)event
   ```

取消事件的情况很少发生，所以在大多数情况下它的行为和触摸结束时相同。

很多情况下，你可能想知道触摸是在哪里开始的。因为触摸事件由Cocoa Touch API接收，所以触摸的位置必须被转换为OpenGL的坐标。以下是一个用来转换坐标的方法：

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

上述方法只对单个触摸有效，因为我们使用了[touches anyObject]。为了跟踪多点触摸的位置，你必须单独跟踪每次触摸。

默认情况下，层接收到的事件和苹果UIResponder类接收到的是一样的。
cocos2d也支持有针对性的触摸处理。和普通处理的区别是：它每次只接收一次触摸，而UIResponder总是接收到一组触摸。有针对性的触摸事件处理只是简单的把一组触摸事件分离开来，这样就可以根据游戏的需求提供所需的触摸事件。更重要的是，有针对性的处理允许你把某些触摸事件从队列里移除。这样的话，

如果触摸发生在屏幕某个指定的区域，你会比较容易识别出来；识别出来以后你就可以把触摸标记为已经处理，并且其它所有的层都不再需要对这个区域再次做检查。

在你的层中添加以下方法可以启用有针对性的触摸事件处理：

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

注：如果你把registerWithTouchDispatcher方法留空，你将不会接收到任何触摸事件！如果你想保留此方法，而且使用它的默认处理方式，你必须调用[super registerWithTouchDispatcher]这个方法。

现在，你将使用一套有点不一样的方法来代替默认的触摸输入处理方法。它们几乎完全一样，除了一点：用 (UITouch *)touch 代替 (NSSet *)touches 作为方法的第一个参数：

```
-(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 {}
```

这里很重要的一点是：ccTouchBegan返回的是一个布尔值（BOOL）。如果你返回了YES，那就意味着你不想让当前的触摸事件传导到其它触摸事件处理器。你实际上是“吞下了”这个触摸事件。

接收加速计事件

和触摸输入一样，加速计必须在启用以后才能接收加速计事件：

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

同样的，层里面要加入一个特定的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);
}
```

你可以通过加速参数来决定任意三个方向的加速度值。

CCSprite

CCSprite是最常用到的类。它使用图片把精灵（sprite）显示在屏幕上。生成精灵最简单的方法是把图片文件加载进CCTexture2D材质里面，然后将它赋给精灵。你必须把需要用到的图片文件放进Xcode的Resources组中，否则你的应用

程序将无法找到指定的图片文件：

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

我想问你个问题：你认为这个精灵会被系统放置在屏幕的哪个地方？可能和你想的相反，精灵贴图的中心点和精灵的左下角位置是一致的。生成的精灵被放置在（0，0）点，也就是屏幕的左下角。因为精灵贴图的中心点和精灵的左下角位置一致，导致贴图只能显示一部份（也就是贴图的右边上半部份）。比如，假设图片大小是80x30px，你必须将精灵移动到坐标（40，15）才能将精灵贴图与屏幕的左下角完美对齐，从而看到完整的贴图。

乍看上去这样安排位置很不寻常，不过将贴图的中心点和精灵的左下角位置设为一致有很大的好处。一旦你开始使用精灵的旋转或缩放属性，精灵的中心点将会保持在它的位置上。

警告：iOS设备上的文件名是区分大小写的！在模拟器上测试时并不区分大小写。但是在iOS设备上实际测试时，程序就会因为大小写错误而崩溃。

这个要求导致了很多让开发者头痛的问题，这也是另一个为什么要经常在设备上做实际测试的原因。为自己确立一个文件命名规则，并且坚持用下去。对我自己而言，我全部使用小写，词和词之间则用下划线分开。

定位点揭秘

每个节点都有一个定位点，但是只有当此节点拥有贴图时，这个定位点才有用。默认情况下，anchorPoint属性设置为（0.5，0.5）或者贴图尺寸的一半。它是一个抽象的因素，一个乘数，而不是一个特定的像素尺寸。

和你想的恰恰相反，定位点和节点的位置没有关系。虽然当你改变anchorPoint属性的时候，你看到精灵在屏幕上的位置发生了变化。但那是错觉，因为节点的位置并没有改变；改变的是精灵里贴图的位置！

anchorPoint定义的是贴图相对于节点位置的偏移。你可以通过把贴图的宽和高乘以定位点来得到贴图的偏移值。顺便提一下，有一个只读的anchorPointPixels属性可以得到贴图的像素偏移值，所以你不需要自己计算。

如果设置anchorPoint为（0，0）的话，你实际上是把贴图的左下角同节点的位置对齐了。以下代码会把精灵图片完美地同屏幕左下角对齐：

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

注：如果你在使用别的游戏引擎时，习惯了把所有精灵定位点都设为0,0的话，请不要在cocos2d里面这样做。这样做会引起很多麻烦，包括旋转和缩放，父节点和子节点之间的相对位置，还有距离测试和碰撞测试。你要保证anchorPoint

在贴图的中央。相信我。

贴图大小

我要特别提一下贴图大小。目前可用于iOS设备的贴图尺寸必须符合“2的n次方”规定，所以贴图的宽和高必须是2，4，8，16，32，64，128，256，512，1024。在第三代设备上可以达到2048像素。贴图不一定是正方形的，所以8x1024像素的贴图完全没有问题。

在你制作贴图的时候你要考虑到上述尺寸要求，比如在为精灵准备图片时。让我们马上来看看最坏情况下会发生什么事情：假设你的图片尺寸是260x260，用的是32位颜色。在内存里，贴图本来只占279KB左右的空间，但是现在却使用了整整1MB。

这几乎是原尺寸四倍的内存占用，这是因为iOS设备要求任何贴图的尺寸必须符合“2的n次方”规定。260x260像素的贴图到了iOS设备中以后，系统会自动生成一张与260x260尺寸最相近的符合“2的n次方”规定的图片（一张512x512像素的图片），以便于把原贴图放进这个符合规定的“容器”中。而这张512x512像素的图片占用了1MB的内存空间。

为了解决这个问题，你唯一能够做的是确保任何制作的图片尺寸符合“2的n次方”规定。260x260像素的图片其实应该做成256x256像素。这样就不会浪费这么多的内存。如果你有设计师为你工作，你要确保她按照要求制作。

在第六章我会教你如何使用“纹理贴图集”来最大限度的解决这个问题。

CCLabel

当你需要在屏幕上显示文字的时候，CCLabel是最直接的选择。以下代码会生成一个CCLabel对象用于显示文字：

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

如果你想知道iOS设备上有哪些TrueType字体可以使用的话，本章提供的Essentials源代码里提供了一个字体列表。

从生成文字的内部原理来说，TrueType字体被用于CCTexture2D贴图上渲染出文字。因为每次文字改变都会导致系统重新渲染一遍，所以你不应该经常改变文字。重建文字标签的贴图非常耗时。

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

如果你改变标签上的文字，文字始终是居中的。这是因为定位点的关系。你可以通过改变anchorPoint属性将文字居左，居右，置顶或者放置在底部。以下代码通过改变anchorPoint属性来排列对齐标签：

```
// 右对齐
```

```
label.anchorPoint = CGPointMake(1, 0.5f);

// 左对齐
label.anchorPoint = CGPointMake(0, 0.5f);

// 置顶放置
label.anchorPoint = CGPointMake(0.5f, 1);

// 放置在底部
label.anchorPoint = CGPointMake(0.5f, 0);

// 使用实例：将标签放置在屏幕右上角
// 标签文字延展到左下方，并且在屏幕上总是可见
CGSize size = [[CCDirector sharedDirector] winSize];
label.position = CGPointMake(size.width, size.height);
label.anchorPoint = CGPointMake(1, 1);
```

菜单

很快你就会需要一些可以让用户进行操作的按钮，比如进入下一个场景或者将音乐打开或关闭的按钮。这里你使用CCMenu类来生成菜单。CCMenu只支持CCMenuItem节点作为它的子节点。

列表3-2的代码展示了如何设置菜单。你可以在Essentials项目的MenuScene类里找到这些菜单代码。

列表3-2：用文字和图片菜单制作cocos2d菜单

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

// 设置CCMenuItemFont的默认属性
[CCMenuItemFont setFontName:@"Helvetica-BoldOblique"];
[CCMenuItemFont setFontSize:26];

// 生成几个文字标签并指定它们的选择器
CCMenuItemFont* item1 = [CCMenuItemFont itemFromString:@"Go Back!" target:self
selector:@selector(menuItem1Touched:)];

// 使用已有的精灵生成一个菜单项
CCSprite* normal = [CCSprite spriteWithFile:@"Icon.png"];
normal.color = ccRED;
CCSprite* selected = [CCSprite spriteWithFile:@"Icon.png"];
selected.color = ccGREEN;
CCMenuItemSprite* item2 = [CCMenuItemSprite itemFromNormalSprite:normal
                                    selectedSprite:selected target:self selector:@selector(menuItem2Touched:)];

// 用其它两个菜单项生成一个切换菜单（图片也可以用于切换）
[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];

// 用菜单项生成菜单
CCMenu* menu = [CCMenu menuWithItems:item1, item2, item3, nil];
menu.position = CGPointMake(size.width / 2, size.height / 2);
[self addChild:menu];

// 排列对齐很重要，这样的话菜单项才不会叠加在同一个位置
[menu alignItemsVerticallyWithPadding:40];
```

警告：菜单项参数总是用nil作为最后一个参数。如果你忘记添加最后的nil参数，应用程序会在那一行崩溃。

第一个菜单项基于CCMenuItemFont，用于显示一条文字。当点击此菜单项，它会调用menuItem1Touched方法。在程序内部，CCMenuItemFont只是简单的生成一个CCLabel。如果你的场景中已经有一个CCLabel，你可以把它与CCMenuItemLabel类结合在一起使用。

同样的，有两个使用图片的菜单项；一个是CCMenuItemImage，它利用图片文件生成菜单项，内部实际上使用了CCSprite来实现。我在上述代码里使用了另一个类：CCMenuItemSprite。我认为这个类使用起来更加方便，因为它可以重复利用已有的精灵作为参数。你可以改变同一个图片的颜色，作为显示触摸后高亮效果的图片

CCMenuItemToggole只接受两个继承自CCMenuItem的对象作为参数，当用户点击菜单时，菜单将会在两个菜单项之间进行切换。你可以在CCMenuItemToggle里使用文字标签或者图片。

最后，CCMenu本身被生成和放置在场景中。因为所有菜单项都是CCMenu的子节点，它们放置的位置都是相对于CCMenu的。为了不让菜单互相叠加在一起，你必须调用一个CCMenu的排列对齐方法，比如像我在**列表3-2**结束的地方使用的alighItemsVerticallyWithPadding方法。

因为CCMenu包含了所有的菜单项，你可以通过动作来让它们一起滚动。这会让你的菜单看上去不那么呆板。我在Essentials项目中提供了一个例子。你可以在**图3-4**中看到上述代码生成的菜单：

图3-4：列表3-2的代码生成了这里的菜单。你只能在实际的代码运行中看到触摸时文字标签的大小会发生变化，精灵的颜色也会发生变化。

动作（Actions）

动作是用于在节点上运行某些“动作”的轻量级类。你可以通过动作让节点移动，旋转，缩放，着色，淡进淡出和干很多其它的事情。因为动作可以用于任何节点，你可以在精灵，标签，甚至菜单或者整个场景中使用它们！这让它们非常有用。

因为大多数动作都是在一段时间内发生的，比如旋转三秒钟，所以通常需要写一个更新方法，还要添加用于储存中间状态的变量。动作（Actions）把这些逻辑都包装了起来，用参数化的方法来应用动作：

```
// 以下代码会让myNode在3秒钟内从当前位置移动到（100，200）坐标点
CCMoveTo* move = [CCMoveTo actionWithDuration:3 position:CGPointMake(100, 200)];
[myNode runAction:move];
```

cocos2d的动作可以分为两种类型。一种是“即时动作”，它的效果和设定节点属性一样，例如设定visible或flipX属性。另一种是“时间间隔动作”，这种动作在一段时间之内发生，例如上述代码的移动动作。你不需要在这两种动作完成以后将它们从内存里清理出去，cocos2d会自动释放动作所占用的内存。

重复动作

你可以让动作或者一系列动作重复运行到永远。你可以通过这个特性生成循环动画。以下代码会让一个节点永远旋转下去，就像一个永远旋转的轮子：

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

舒缓动作

CCEaseAction类让cocos2d的动作更加有用。“舒缓动作”允许你改变在一段时间内发生的动作效果。例如，如果你在节点上应用CCMoveTo动作，此节点在整个移动过程中将会保持同一个速度。而如果你使用CCEaseAction的话，你就可以让节点慢慢启动，然后加速向目标移动，或者反过来（快速启动，慢慢减速到达目标）。或者你也可以让节点移动到超过目的地一些，然后再反弹回来。“舒缓动作”可以帮助你创造出通常很费时间才能做出来的动画。以下代码演示了如何应用舒缓动作来改变一个普通动作的行为。rate参数是用来决定舒缓动作的明显程度。此参数只有在大于1的情况下才能看到舒缓动作的效果：

```
// 我想让myNode在3秒钟之内移动到100，200坐标点
CCMoveTo* move = [CCMoveTo actionWithDuration:3 position:CGPointMake(100, 200)];
// 节点应该慢慢启动，然后在移动过程中减速
CCEaseInOut* ease = [CCEaseInOut actionWithAction:move rate:4];
[myNode runAction:ease];
```

注：在上述例子中，舒缓动作是在节点上运行的，而不是在移动动作上运行。当你使用动作时，很容易忘记runAction那行代码里的动作。即使最有经验的cocos2d开发者也会犯这样的错误。如果你看到动作没有如你期望的那样工作的话，记得检查一下你是在运行正确的动作。如果你确定选择了正确的动作，但还是没有得到想要的结果，请确认正确的节点上执行动作。这是另一个很容易犯的错误。

cocos2d实现了以下CCEaseAction类：

1. CCEaseBackIn, CCEaseBackInOut, CCEaseBackOut
2. CCEaseBounceIn, CCEaseBounceInOut, CCEaseBounceOut
3. CCEaseElasticIn, CCEaseElasticInOut, CCEaseElasticOut
4. CCEaseExponentialIn, CCEaseExponentialInOut, CCEaseExponentialOut
5. CCEaseIn, CCEaseInOut, CCEaseOut
6. CCEaseSineIn, CCEaseSineInOut, CCEaseSineOut

在第四章，我将在DoodleDrop项目中使用这里面的一些舒缓动作，这样你可以看到它们的具体运行效果。

动作序列

通常情况下，当你给一个节点添加多个动作时，它们会在同一时间运行。例如，你可以通过添加动作让物体在旋转的同时淡出消失。但是如果你想要一个动作一个动作的运行呢？

有时候，一个动作一个动作的运行会更有用。我们可以使用CCSequence来达到这个目的。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，你可以让一个节点先移动到目标位置，然后在节点到达目标之后让其旋转，最后淡出消失。动作一个跟着一个的运行，直到完成整个动作序列。

以下代码演示了如何让一个标签的颜色从红色变为蓝色，最后变为绿色：

```
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与CCRepeatForever动作结合使用：

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

注：和菜单项一样，一串动作最后总是要用nil来结束。如果你忘记在最后用nil结束参数的话，CCSequence这串代码将会崩溃！

即时动作

你可能会奇怪为什么有基于CCInstantAction的即时动作存在，通过改变节点的属性不是可以达到一样的目的吗？比如那些用来翻转节点，把节点放置的指定的地方，还有用于开关节点的可视性属性的即时动作。

即时动作的存在是为动作序列服务的。有时候在一个动作序列里你必须改变某些节点的属性，像可视性或者位置，改变完成以后会继续当前的动作序列。即时动作让这样的应用变得可能。其中用的最多的可能是CCCallFunc动作。

当你使用一个动作序列时，你可能需要在某个时间得到通知。比如当一个动作序列完成运行以后，你想知道这个动作序列已经完成，得到通知以后，你就可以接着继续另一个动作序列。以下代码重写了之前的颜色改变动作序列的例子，它会在每次完成一个CCTintTo动作以后调用三个CCCallFunc动作中的一个来发送信息：

```
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];
```

上述动作序列将调用以下代码来发送信息。sender这个参数继承自CCNode；这是运行动作的节点。data参数可以是任何值，结构或者指针。只是你必须正确地转换data指针的类型。

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

```
}
-(void) onCallFuncN:(id)sender
{
	CCLOG(@"end of tint2! sender: %@", sender);
}
-(void) onCallFuncND:(id)sender data:(void*)data
{
	// 如下转换指针的方式要求data必须是一个CCSprite
	CCSprite* sprite = (CCSprite*)data;
	CCLOG(@"end of sequence! sender: %@ - data: %@", sender, sprite);
}
```

当然，CCCallFunc也可以和CCRepeatForever一起使用。这样，你所指定的方法将会被重复调用。

cocos2d测试例子

cocos2d提供了很多样例。你可以在cocos2d-iphone文件夹中找到它们。你可以在Xcode里面构建和运行所有的例子，从而了解它们工作的原理。

结语

本章讨论了很多东西！一下子记住那么多内容几乎是不可能的。所以时常回到本章，温故而知新。

通过本章的学习，我们已经准备好，可以开始编写自己的游戏了。

下一章我会和你一起制作你的第一个完整的游戏！