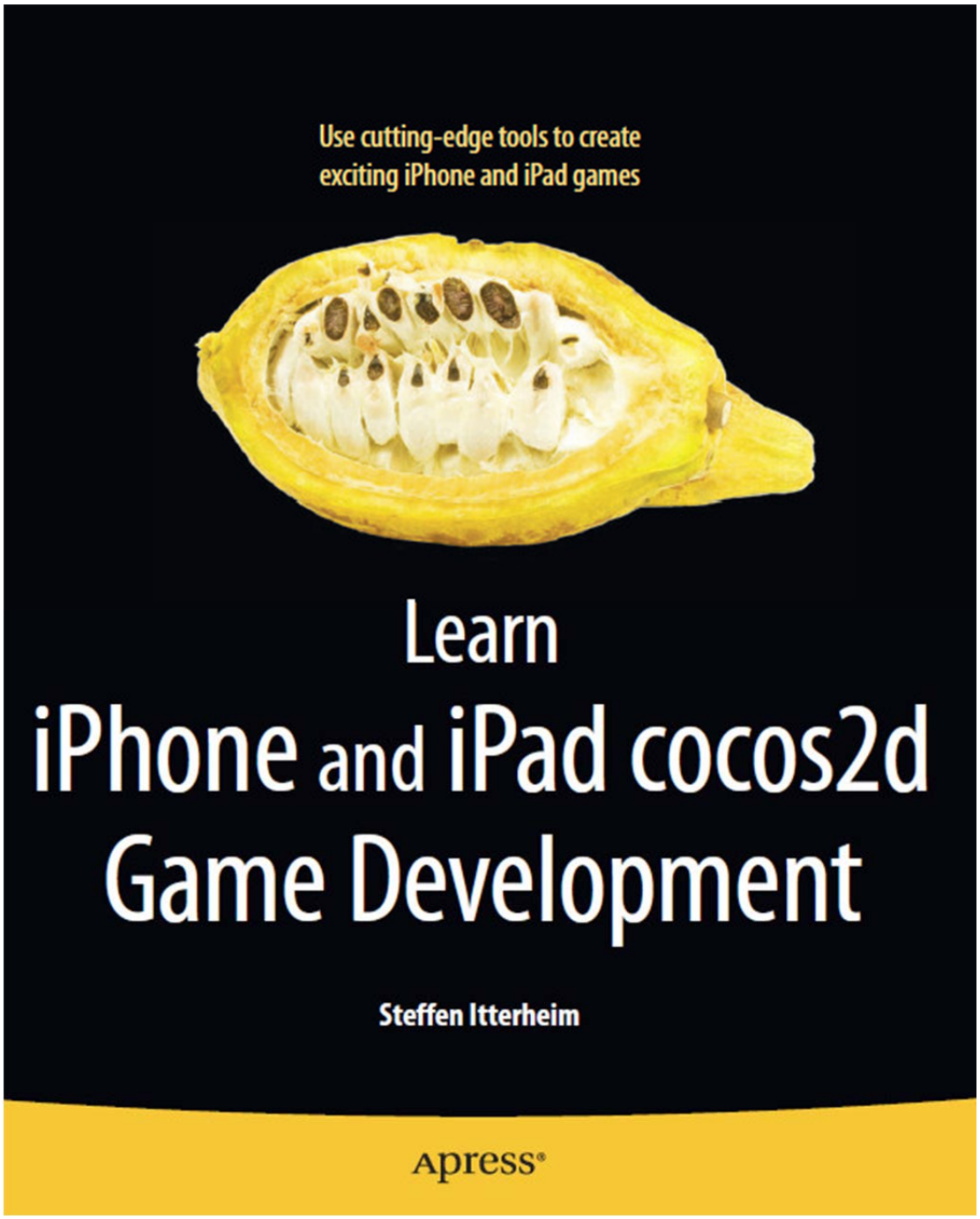

译者：杨栋
邮箱：yangdongmy@gmail.com

第八章
完成滚屏射击游戏

为了完成上一章的滚屏射击游戏，我们将在本章加入各种敌人和子弹，还有Boss怪物。敌人角色和玩家角色将会使用同一个新设计的BulletCache类来发射多个种类的子弹。BulletCache类会重用隐形的子弹以避免频繁地分配和释放子弹所占用的内存。因为屏幕上将会显示大量的敌人角色，所以我们也会给敌人创建一个EnemyCache类。很显然，玩家角色可以向敌人开火。

本章我也会介绍“基于组件的编程方法”（Component-based Programming），你可以通过模块化的方式来扩展游戏的角色种类。我们除了创建射击组件和移动组件，还会创建一个用于显示Boss怪物生命值的组件。毕竟Boss怪物和普通敌人不同，不是一下两下就可以打死的，所以我们需要一个组件来显示Boss的剩余生命值。

添加BulletCache类

在ShootEnUp01项目中，我们会使用BulletCache类来生成新子弹。之前，生成子弹的代码都在GameScene类中，但是GameScene类其实不应该被用于生成和管理子弹的。**列表8-1**展示了BulletCache类的头文件，你可以看到现在它包含了一个CCSpriteBatchNode和隐形子弹的计数器变量：

列表8-1. BulletCache类的头文件代码

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
@interface BulletCache : CCNode
{
        CCSpriteBatchNode* batch;
        int nextInactiveBullet;
}
-(void) shootBulletAt:(CGPoint)startPosition velocity:(CGPoint)velocity
                                        frameName:(NSString*)frameName;
@end
```

为了重构GameScene类中的子弹代码，我必须把初始化代码和子弹射击代码都移到BulletCache类中去（见**列表8-2**）。我也把CCSpriteBatchNode放进了一个成员变量，这样就不用每次使用CCNode的getChildByTag方法来获取精灵批处理对象了。因为我会把BulletCache类加到GameScene中去，所以我把CCSpriteBatchNode放在了BulletCache类中。

注：把BulletCache这样的CCNode添加到场景中虽然会增加场景的层级数，但是并不会带来任何坏处。如果你不希望增加场景层级数的话，你也可以像以前一样把CCSpriteBatchNode添加到GameScene类中，然后在GameScene中创建一个获取方法，这样BulletCache类就可以获取CCSpriteBatchNode了。不过这样的话你就多写了一个不必要的函数。我的一个原则是：优先考虑代码的可读性，如

果有必要，之后可以重构代码以提高性能。

列表8-2. BulletCache中有一个可以重用的“子弹池”

```
#import "BulletCache.h"
#import "Bullet.h"

@implementation BulletCache
-(id) init
{
	if ((self = [super init]))
	{
		// 从纹理贴图集中得到相关子弹图片
		CCSpriteFrame* bulletFrame = [[CCSpriteFrameCache sharedSpriteFrameCache]
									spriteFrameByName:@"bullet.png"];

		// 使用子弹图片
		batch = [CCSpriteBatchNode batchNodeWithTexture:bulletFrame.texture];
		[self addChild:batch];

		// 初始化时就生成一定数量的子弹，以后可以重复使用
		for (int i = 0; i < 200; i++)
		{
			Bullet* bullet = [Bullet bullet];
			bullet.visible = NO;
			[batch addChild:bullet];
		}
	}

	return self;
}

-(void) shootBulletAt:(CGPoint)startPosition velocity:(CGPoint)velocity
				frameName:(NSString*)frameName
{
	CCArray* bullets = [batch children];
	CCNode* node = [bullets objectAtIndex:nextInactiveBullet];
	NSAssert([node isKindOfClass:[Bullet class]], @"not a Bullet!");
	Bullet* bullet = (Bullet*)node;
	[bullet shootBulletAt:startPosition velocity:velocity frameName:frameName];

	nextInactiveBullet++;
	if (nextInactiveBullet >= [bullets count])
	{
		nextInactiveBullet = 0;
```

```
        }
}
@end
```

你可以看到shootBulletAt这个方法被修改的最多。现在它接受3个参数：startPosition, velocity和frameName。而之前这个方法只需要一个参数：一个Ship类的指针。这三个参数会被传进Bullet类的shootBulletAt方法中。我也重构了Bullet类的shootBulletAt方法：

```
-(void) shootBulletAt:(CGPoint)startPosition velocity:(CGPoint)vel
                                            frameName:(NSString*)frameName
{
        self.velocity = vel;
        self.position = startPosition;
        self.visible = YES;

        // 通过使用传进来的frameName来改变子弹所用的贴图
        CCSpriteFrame *frame = [[CCSpriteFrameCache sharedSpriteFrameCache]
                                                        spriteFrameByName:frameName];
        [self setDisplayFrame:frame];
        [self scheduleUpdate];
}
```

现在，速度和位置信息都被直接赋给了子弹。这意味着BulletCache类中的shootBulletAt方法必须决定子弹的位置，方向和速度。我正需要这样的结构：因为这样我就对子弹的射击拥有了完全的控制，包括使用setDisplayFrame方法来改变子弹的贴图以显示不同种类的子弹。因为所有的子弹图片都在同一张纹理贴图集中，所以你需要做的只是指定子弹的精灵帧（sprite frame）就可以改变子弹的外观了。实际运行中，系统只要渲染同一张纹理贴图中的不同部份，即可得到不同的子弹贴图，而不需要额外的系统开销。

我在Bullet类中也修补了一个问题：目前只有飞出屏幕右边边界的子弹才会被设为隐形，然后放到重用的等候列表中。通过在子弹的更新方法中使用CGRectIntersectsRect方法检查子弹的boundingBox（边界框）和screenRect，你可以知道子弹是不是已经处于屏幕外面了，如果已经处于屏幕可视区域以外，你可以将子弹标记为“可以重用”，也就是隐形：

```
// 当子弹离开屏幕以后，将其设置为隐形
if (CGRectIntersectsRect([self boundingBox], screenRect) == NO)
{
        …
}
```

为了便于使用和性能方面的考虑，我将screenRect变量设置为一个静态变量。这样其它类就可以方便地访问它，而不需要每次都生成了。screenRect这样的静态变量存在于类的实现（implementation）文件中。静态变量对于类来说就

像全局变量：任何这个类生成的实例（instance）都可以读写它。而成员变量与之不同：一个类实例中的成员变量都是当前实例的本地变量，这个类生成的其它实例不能读写这些本地变量。因为在游戏中屏幕尺寸不会改变，而且所有的Bullet实例都要使用这个尺寸变量，所以将其存为一个所有类的实例都可以使用的静态变量就是很好的选择了。第一个被初始化的子弹将会设置好screenRect变量，然后其它子弹实例都可以使用这个变量了。CGRectIsEmpty方法用于测试screenRect这个变量是否还没有被初始化；因为此变量是静态的，所以你只需要初始化一次就够了。

```
static CGRect screenRect;
…
// 用于确保 screenRect 只会被初始化一次
if (CGRectIsEmpty(screenRect))
{
        CGSize screenSize = [[CCDirector sharedDirector] winSize];
        screenRect = CGRectMake(0, 0, screenSize.width, screenSize.height);
}
```

完成上述修改以后，我要把GameScene类中之前用于射击子弹的方法和成员变量移除。具体就是用BulletCache类的初始化代码替换CCSpriteBatchNode的初始化代码：

```
BulletCache* bulletCache = [BulletCache node];
[self addChild:bulletCache z:1 tag:GameSceneNodeTagBulletCache];
```

我也添加了一个名为bulletCache的获取方法（getter method），这样别的类就可以通过GameScene来访问BulletCache实例了：

```
-(BulletCache*) bulletCache
{
        CCNode* node = [self getChildByTag:GameSceneNodeTagBulletCache];
        NSAssert([node isKindOfClass:[BulletCache class]], @"not a BulletCache");
        return (BulletCache*)node;
}
```

现在，InputLayer可以使用新的BulletCache类，让飞船射出子弹了。子弹的一些属性，比如起始位置，速度和用于表示子弹的贴图，现在由InputLayer中的更新方法里的射击代码来传递：

```
if (fireButton.active && totalTime > nextShotTime)
{
        nextShotTime = totalTime + 0.5f;

        GameScene* game = [GameScene sharedGameScene];
        Ship* ship = [game defaultShip];
        BulletCache* bulletCache = [game bulletCache];

        // 射击之前设置位置，速度和所用子弹贴图的信息
```

```
        CGPoint shotPos = CGPointMake(ship.position.x + [ship contentSize].width * 0.5f,
                                                                    ship.position.y);
        float spread = (CCRANDOM_0_1() - 0.5f) * 0.5f;
        CGPoint velocity = CGPointMake(1, spread);
        [bulletCache shootBulletAt:shotPos velocity:velocity frameName:@"bullet.png"];
}
```

上述代码重构的结果是我们得到了控制子弹射击的灵活性。我想你现在应该可以想像敌人角色将如何使用相同的代码进行射击了吧？

敌人角色代码的设计和编写

目前为止，关于敌人是什么样的，它们会做什么和它们的行为方式，我们只有很模糊的想法。现在我们需要重新回到画板上，开始设计敌人角色，然后决定敌人在游戏中的行为。

我已经画出了三种不同类型的敌人角色，其中一个将会是我们的Boss怪物。看一下图8-1中的三个怪物，然后试着想像一下它们会干些什么事情。

图8-1. 三种类型的敌人角色。姜饼人，蛇和团状怪物（又叫Boss）。

在开始编程之前，你应该想一下这三个敌人会有哪些相同的行为，这样的话你就可以把相同的行为在编程的时候放到一起。代码设计最重要的目标就是消除重复的代码。让我们看看这些敌人会有哪些相同的行为：

1. 射出子弹
2. 拥有何时，何地，应该射出何种子弹的逻辑代码
3. 飞船的子弹可以打中它们
4. 其它敌人的子弹打不到它们
5. 可以抵挡一次或者多次攻击（拥有健康值）
6. 拥有明确的行为和移动方式
7. 当被消灭时，拥有特定的行为或者动画
8. 将会在紧靠屏幕外的位置出现，然后移动进入屏幕
9. 一旦进入屏幕，将不会再离开屏幕

当你看完上述列表以后，你可能会注意到其中的一些属性和飞船是一样的。飞船可以射出子弹；可以经受多次攻击；然后在被消灭的时候有特定的行为或动画。我们可以认为飞船只是一种特别的敌人类型。

我想到三个可能行得通的设计方向。

第一个设计方向是把飞船，敌人和Boss都放到同一个类中。取决于角色的类型，某些部份的代码会有条件地运行。比如，按照角色类型执行不同的射击动作逻辑。如果只有很少的几种角色类型，这个方法是可行的。不过缺点也很明显：扩展性很差。随着角色类型的增加，你的类会变得越来越大，所有的处理逻辑都堆在一起。当你修改某个部份的代码时，很可能会影响别的代码。

第二个设计方向是创建一个名为Entity的基类，飞船和三个敌人都继承自这个基类。很多程序员都会这样做，而且在角色类型少的时候也可以很好的工作。但是实际上，这个方法和第一个没有多大区别。一些不是所有的子类都用得到的代码会堆积在Entity这个基类中。如果你在Entity类中再添加一个switch用于判断哪些敌人应该执行哪些代码的话，那就更糟糕了。不过，如果你小心点，你可以确保那些针对某个敌人类型的代码只存在于特定的子类中，但是很多时候这些代码会被放到Entity基类中去。

第三个设计方向是使用组件系统。这意味着一些独立的代码可以从Entity类中分离出来，成为单独的组件，然后只添加到那些有需要的子类中，例如生命值显示组件。因为对基于组件的开发讨论可以写上一本书，而且对我们的这个小项目来说也没有很大的必要，所以我将会把类继承和组件设计放在一起来设计我们的角色类。这样你至少可以知道如何将单独的组件复合起来，同时也能了解这样做的好处。

我想指出的是：其实不存在最好的代码设计方式。我们做出的某些选择完全是主观的，基于个人喜好和经验的。如果你愿意在了解自己所做游戏的过程中，不断地重构代码的话，你应该先编写出可以工作的代码，而不是非要追求干净的完美代码。随着经验的增加，你可以在计划阶段就做出一些正确的决定，从而以更快的速度开发更加复杂的游戏。如果这是你的目标的话，你可以一开始制作一些小游戏，然后慢慢给自己增加难度和新的挑战。这是个学习的过程，而且很不幸的是，如果你的野心过大，你的动力也会消失的越快。这也是为什么所有老练的游戏程序员会建议新人先从小游戏做起：模仿制作像Tetris，Pac-Man，或者Asteroids这样的游戏。

Entity类的层级

我在ShootEmUp02项目中创建了Entity这个基类。Entity是一个继承自CCSprite的通用类，它只包含了Ship类的setPosition方法，用于将所有Entity的实例限制在屏幕的可视范围之内。我只做了一点小修改，现在屏幕外面的物体可以移动到屏幕里面，一旦进入屏幕内，它们就不能再离开屏幕。在游戏中，为了展示很快就要介绍的EnemyCache这个类，敌人不会从飞船身边飞过，而是停留在屏幕中间。代码里的屏幕区域测试是用来测试精灵的boundingBox（边界框）是否完全包含在屏幕的可视区域内。如果精灵被完全包含在屏幕内，用于将精灵

保持在屏幕内的代码才会运行：

```
-(void) setPosition:(CGPoint)pos
{
        // 如果精灵的当前位置处于屏幕之外，就没有必要对其位置做任何调整！
        // 这样可以让屏幕外的精灵进入屏幕内
        if (CGRectContainsRect([GameScene screenRect], [self boundingBox]))
        {
                …
        }

        [super setPosition:pos];
}
```

Ship类现在已被ShipEntity类所替代。因为Entity这个基类现在包含了setPosition方法的代码，ShipEntity类中只需要再添加一个initWithShipImage方法就可以了。这个方法和之前的一样，所以我就不在这里重写一遍了。

EnemyEntity类

接下去我们讨论EnemyEntity类，看一下它的内部运行机制。**列表8-3**是它的头文件内容。

列表8-3. EnemyEntity类的头文件

```
#import <Foundation/Foundation.h>
#import "Entity.h"

typedef enum
{
        EnemyTypeBreadman = 0,
        EnemyTypeSnake,
        EnemyTypeBoss,
        EnemyType_MAX,
} EnemyTypes;

@interface EnemyEntity : Entity
{
        EnemyTypes type;
}

+(id) enemyWithType:(EnemyTypes)enemyType;
+(int) getSpawnFrequencyForEnemyType:(EnemyTypes)enemyType;
-(void) spawn;
@end
```

上述代码中没有特别令人兴奋的东西。EnemyTypes这个枚举用于区别当前支持的三种不同类型的敌人，EnemyType_MAX则被用作循环的上限值。EnemyEntity类有一个成员变量用于储存类型，这个成员变量的值会被用于switch条件中，根据敌人的类型来判断需要执行的代码。

EnemyEntity类中有很多代码需要讨论，所以我将它们分成不同的主题来讨论，讨论每个主题时我将只提供相关的代码。首先讨论的是**列表8-4**中的initWithType方法：

列表8-4. 用类型（Type）初始化一个敌人

```
-(id) initWithType:(EnemyTypes)enemyType
{
	type = enemyType;

	NSString* frameName;
	NSString* bulletFrameName;
	int shootFrequency = 300;
	switch (type)
	{
		case EnemyTypeBreadman:
			frameName = @"monster-a.png";
			bulletFrameName = @"candystick.png";
			break;
		case EnemyTypeSnake:
			frameName = @"monster-b.png";
			bulletFrameName = @"redcross.png";
			shootFrequency = 200;
			break;
		case EnemyTypeBoss:
			frameName = @"monster-c.png";
			bulletFrameName = @"blackhole.png";
			shootFrequency = 100;
			break;
		default:
			[NSException exceptionWithName:@"EnemyEntity Exception"
					reason:@"unhandled enemy type" userInfo:nil];
	}
	if ((self = [super initWithSpriteFrameName:frameName]))
	{
		// 创建游戏逻辑的组件
		[self addChild:[StandardMoveComponent node]];

		StandardShootComponent* shootComponent = [StandardShootComponent node];
		shootComponent.shootFrequency = shootFrequency;
		shootComponent.bulletFrameName = bulletFrameName;
		[self addChild:shootComponent];
```

```
        // 一开始，敌人都都被设定为隐形
        self.visible = NO;

        [self initSpawnFrequency];
    }
    return self;
}
```

上述代码开始的时候，我根据敌人的类型，使用switch来给不同类型的敌人赋予默认值－包括用到的角色精灵帧名称（贴图），子弹的精灵帧名称和射击的频率。switch在默认情况下会抛出一个异常，因为之所以会进入默认情况通常是因为程序员在EnemyTypes枚举中添加了一个新的类型，但是忘记了扩展相对应的switch条件。通过在switch的默认情况中抛出异常，当你忘记扩展代码时，程序会直接崩溃，并且告诉你:”喂，你忘记更新我了！”。这样就不用花无谓的时间在调试上了。

你也可以在把值赋给self之前运行所有的代码，只是不要忘记调用[super init…]这个方法。否则super类就不会被正确地初始化，导致奇怪的bug和崩溃。

在EnemyEntity中用到的组件类（component classes）包含了可以相互替换的代码。我将在稍后讨论组件类。现在你只需要知道StandardMoveComponent可以让敌人角色移动，而StandardShootComponent可以让它们射击。

现在，让我们专注于initSpawnFrequency方法。相关代码展示在**列表8-5**中：

列表8-5. 控制敌人角色的生成

```
static CCArray* spawnFrequency;
-(void) initSpawnFrequency
{
    // 初始化敌人角色的生成速度
    if (spawnFrequency == nil)
    {
        spawnFrequency = [[CCArray alloc] initWithCapacity:EnemyType_MAX];
        [spawnFrequency insertObject:[NSNumber numberWithInt:80]
                                                    atIndex:EnemyTypeBreadman];
        [spawnFrequency insertObject:[NSNumber numberWithInt:260]
                                                    atIndex:EnemyTypeSnake];
        [spawnFrequency insertObject:[NSNumber numberWithInt:1500]
                                                    atIndex:EnemyTypeBoss];

        // 马上生成一个敌人
        [self spawn];
    }
}
```

```
+(int) getSpawnFrequencyForEnemyType:(EnemyTypes)enemyType
{
	NSAssert(enemyType < EnemyType_MAX, @"invalid enemy type");
	NSNumber* number = [spawnFrequency objectAtIndex:enemyType];
	return [number intValue];
}

-(void) dealloc
{
	[spawnFrequency release];
	spawnFrequency = nil;
	[super dealloc];
}
```

我把每种敌人的生成速度值放在一个叫做spawnFrequency的静态CCArray变量中。之所以使用静态变量，是因为敌人角色的实例并不需要生成速度值，而是各个敌人类型需要这个值。第一个生成的EnemyEntity实例会发现spawnFrequency的值是nil，从而执行initSpawnFrequency方法中的代码，将spawnFreqency进行初始化。

因为CCArray只能储存对象，不能储存像整数这样的简单数据类型，所以我们要用NSNumber类中的numberWithInt初始化方法来包装所需要的整数值。我使用了insertObject而不是addObject把数组元素添加到CCArray中。因为insertObject不仅可以确保数组中每个元素的索引与枚举中的敌人类型值相同，而且可以让其他阅读这些代码的程序员知道我们用的索引所代表的意义。这里的索引和敌人类型是相同的，你可以清楚地看到各个敌人类型被赋予的值。

spawnFrequency变量会在dealloc方法中被释放，然后被设为nil。这个步骤很重要。因为spwanFrequency是个静态变量，所以第一个被释放的EnemyEntity对象会运行它的dealloc方法，从而释放spawnFrequency所占用的内存。如果你不在释放spawnFrequency以后将其设为nil的话，接下去被释放的EnemyEntity对象将会再次尝试释放spawnFrequency变量；但是因为这个变量已经被释放，所以再次释放的操作将会导致程序崩溃。如果在释放以后，将spawnFrequency设为nil的话，任何对此变量进行的释放操作会被自动忽略。我在之前说过这个原则，现在再强调一遍：在Objective-C中，向已经被设为nil的对象发送释放消息是没有问题的；你发送的消息会被自动忽略。

以下代码是用于生成角色的方法：

```
-(void) spawn
{
	// 选择一个屏幕可视区域外，紧靠屏幕右侧边界的位置
	CGRect screenRect = [GameScene screenRect];
	CGSize spriteSize = [self contentSize];
	float xPos = screenRect.size.width + spriteSize.width * 0.5f;
```

```
        float yPos = CCRANDOM_0_1() * (screenRect.size.height - spriteSize.height) +
                                                                    spriteSize.height * 0.5f;
        self.position = CGPointMake(xPos, yPos);

        // 最后将生成的角色设定为可视状态，也就是把角色标记为“正在使用中“状态
        self.visible = YES;
}
```

EnemyCache类是用于预先生成所有敌人角色实例的。生成的敌人将会出现在屏幕可视区域外紧靠屏幕右侧边界的位置，每个敌人的Y轴位置都是随机的。一旦成功地生成一个敌人，它的显示属性（visible）会被设为YES，也就是处于可视状态。组件类将会使用敌人的显示属性来决定此敌人是否处于使用状态。如果处于不可见状态，我们就可以通过设置显示属性为YES来激活这个敌人。只有可视的敌人才可以运行它的游戏逻辑。

EnemyCache类

我刚刚提到了EnemyCache类。通过它的名字，你可能会联想到同样拥有预先初始化对象的BulletCache类，你可以快速和简便地重用这些预先初始化好的对象。这样做可以避免在游戏中频繁地生成和释放对象，从而导致游戏运行效率降低。特别是动作类游戏，如果游戏过程中出现错误的话，用户体验就会非常差。让我们在**列表8-6**中看一下EnemyCache类的头文件代码:

列表8-6. EnemyCache类的头文件代码

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"

@interface EnemyCache : CCNode
{
        CCSpriteBatchNode* batch;
        CCArray* enemies;
        int updateCount;
}
@end
```

CCSpriteBatchNode用于存放所有的敌方精灵。enemies这个CCArray用于存放每种敌人的清单。游戏在每一帧中调用更新方法时，updateCount变量会被加1，使用它可以在一定的时间间隔内生成敌方角色。EnemyCache的初始化方法和BulletCache的很相似，因为它们都要初始化一个CCSpriteBatchNode：

```
-(id) init
{
        if ((self = [super init]))
        {
                // 从纹理贴图集中 拿到需要的图片
                CCSpriteFrame* frame = [[CCSpriteFrameCache sharedSpriteFrameCache]
                                                        spriteFrameByName:@"monster-a.png"];
```

```
        batch = [CCSpriteBatchNode batchNodeWithTexture:frame.texture];
        [self addChild:batch];

        [self initEnemies];
        [self scheduleUpdate];
    }
    return self;
}
```

不过，因为用于初始化敌人的代码比较复杂一些，所以我把代码放到了一个单独的方法中，如**列表8-7**所示：

列表8-7. 预先初始化一群敌人，以备将来之用

```
-(void) initEnemies
{
    // 创建一个数组用于存放每一种敌人的数组
    enemies = [[CCArray alloc] initWithCapacity:EnemyType_MAX];

    // 创建每一种敌人类型的数组
    for (int i = 0; i < EnemyType_MAX; i++)
    {
        // 我们根据敌人的类型来设置数组的大小
        // 用于储存需要的敌人对象
        int capacity;
        switch (i)
        {
            case EnemyTypeBreadman:
                capacity = 6;
                break;
            case EnemyTypeSnake:
                capacity = 3;
                break;
            case EnemyTypeBoss:
                capacity = 1;
                break;
            default:
                [NSException exceptionWithName: @"EnemyCache Exception"
                                        reason:@"unhandled enemy type"
                                        userInfo:nil];
                break;
        }

        // 不需要使用alloc，因为enemies数组会保存（retain）任何添加进来的元素
        CCArray* enemiesOfType = [CCArray arrayWithCapacity:capacity];
        [enemies addObject:enemiesOfType];
    }
```

```
        for (int i = 0; i < EnemyType_MAX; i++)
        {
                CCArray* enemiesOfType = [enemies objectAtIndex:i];
                int numEnemiesOfType = [enemiesOfType capacity];
                for (int j = 0; j < numEnemiesOfType; j++)
                {
                        EnemyEntity* enemy = [EnemyEntity enemyWithType:i];
                        [batch addChild:enemy z:0 tag:i];
                        [enemiesOfType addObject:enemy];
                }
        }
}

-(void) dealloc
{
        [enemies release];
        [super dealloc];
}
```

这里比较有趣的部份是enemies这个CCArray，它包含了更多的CCArray对象，每一种敌人类型都有一个CCArray对象。这就是所谓的二维数组。enemies这个成员变量必须使用alloc来创建，否则在离开initEnemies方法以后，enemies将会从内存中被释放。与之不同的是，添加进enemies变量的CCArray对象不需要使用alloc来创建，因为enemies会保存(retain)添加进来的对象。

每一个enemiesOfType的起始大小决定着显示在屏幕上与之类型相关的敌人数量。这样做的好处是你可以控制屏幕上出现的敌人数量。然后我们使用addObject方法将每个enemiesOfType数组加到enemies数组中。如果需要的话，你可以通过这样的方式创造出很深的层级。实际上cocos2d节点的层级是基于CCNode类的，而CCNode包含了CCArray* children，children则进一步包含更多的CCNode类。

我把数组的初始化和敌人的生成放在了不同的循环中来完成。你可以把它们两个放在同一个循环里面，但是那样的话代码逻辑会变得不清晰。分成两个循环所带来的额外系统开销是可以忽略不计的。

通过使用初始化中设置的数组大小，我们生成了相应数量的敌人，然后将它们添加到CCSpriteBatchNode中，最后添加到相对应的enemiesOfType数组中。虽然我们可以通过CCSpriteBatchNode来访问生成的敌人，但是保留一个单独的敌人数组可以让你在以后更方便地使用它，比如在**列表8-8**的生成代码中访问敌人数组：

列表8-8. 生成相应类型的敌人

```
-(void) spawnEnemyOfType:(EnemyTypes)enemyType
{
```

```
        CCArray* enemiesOfType = [enemies objectAtIndex:enemyType];
        EnemyEntity* enemy;
        CCARRAY_FOREACH(enemiesOfType, enemy)
        {
                // 找到第一个还没有被使用的敌人，激活它
                if (enemy.visible == NO)
                {
                        //CCLOG(@"spawn enemy type %i", enemyType);
                        [enemy spawn];
                        break;
                }
        }
}

-(void) update:(ccTime)delta
{
        updateCount++;
        for (int i = EnemyType_MAX - 1; i >= 0; i--)
        {
                int spawnFrequency = [EnemyEntity getSpawnFrequencyForEnemyType:i];
                if (updateCount % spawnFrequency == 0)
                {
                        [self spawnEnemyOfType:i];
                        break;
                }
        }
}
```

上述更新方法每次都会给updateCount加1。虽然没有考虑到每次更新之间的时间间隔，但是由于这些时间间隔的变化通常很小，所以使用简单的updateCount可以让我们的逻辑变得简单一些。接下去的循环有点奇怪，因为它是从EnemyType_MAX-1开始一直循环到 i 变成负数。这里的唯一目的是让拥有高EnemyType数值的敌人类型在游戏中优先生成：如果Boss怪物和蛇要一起生成的话，Boss会优先于蛇出现在游戏中。否则，有可能蛇会先于Boss生成，导致需要Boss的时候不能得到Boss。这是我们的敌人生成逻辑的缺陷，我希望你可以扩展现在的逻辑以改进代码。如果你决定要制作自己的经典射击游戏的话，你肯定是需要进行代码改进的。

我们通过EnemyEntity的getSpawnFrequencyForEnemyType方法来得到spawnFrequency（生成频率）：

```
+(int) getSpawnFrequencyForEnemyType:(EnemyTypes)enemyType
{
        NSAssert(enemyType < EnemyType_MAX, @"invalid enemy type");
        NSNumber* number = [spawnFrequency objectAtIndex:enemyType];
```

```
    return [number intValue];
}
```

首先，上述方法会确保enemyType的值是在确定好的范围之内。然后我们得到敌人类型对象的NSNumber对象，利用NSNumber的intValue方法得到需要的数值。

更新方法中的按模运算符（%）将会返回updateCount和spawnFrequency两个数值相除得到的余数。这意味着只有在updateCount能被spawnFrequency除尽的情况下，也就是相除的返回值是0时，才会生成一个敌人。

然后spawnEnemyOfType方法会得到enemies这个包含各个敌人类型的所有敌人对象的数组。接着我们提取只和当前敌人类型相关的敌人数组，然后进行循环操作。这样就可以避免使用包含所有精灵的CCSpriteBatchNode了。一旦在循环中发现了一个处于“不可视”状态下的敌人，我们就会调用它的spawn方法。如果找不到处于“不可视”状态下的敌人，说明当前屏幕显示了允许的最大数量的敌人，新的敌人将不会被生成，这样就可以有效地限制同一类型敌人在屏幕上的显示数量。

组件类（Component Classes）

组件类是用来扩展游戏逻辑的插件。如果你给一个类添加了一个组件，那么这个类就会执行组件的行为逻辑：移动，射击，动画，显示生命值，等等。

使用组件的一个很大的好处是：因为它们是和CCNode父类进行交互的，所以在编写组件时对将会用到它们的父类会做尽量少的假设。当然，有些时候某个组件可能会要求它的父类是一个EnemyEntity类，不过即使这样，任何一个EnemyEntity都可以使用这个组件。

组件类也可以根据将会使用此组件的类来进行配置。让我们来看个例子。以下是EnemyEntity类中对StandardShootComponent这个组件进行初始化的代码：

```
StandardShootComponent* shootComponent = [StandardShootComponent  node];
shootComponent.shootFrequency = shootFrequency;
shootComponent.bulletFrameName = bulletFrameName;
[self addChild:shootComponent];
```

我们在之前已经根据EnemyType对shootFrequency和bulletFrameName这两个变量进行了设置。通过在EnemyEntity类中添加StandardShootComponent组件，敌人角色就可以每隔一段时间使用指定的子弹进行射击。因为上述组件并不要求特定的父类才能使用它，所以你也可以把它添加到ShipEntity中，让飞船每隔一段时间自动进行射击。或者，你也可以通过打开或者关闭指定的射击组件，用很少的代码就可以创造出玩家角色改变武器的效果。你需要做的只是单独编写射击代码，然后添加到各个类中，通过不同的参数来控制这些组件。剩下你就只要编写在适当的时间关闭或者打开某个射击组件的逻辑代码就行了。更进一步，你可以在另外类似的游戏中应用这些组件代码。对于编写可重复使用的

代码，组件是很好的选择，而且也是很多游戏引擎的标准实现机制。如果你想学习更多关于如何编写游戏组件的知识，请参考我的网站上的博客文章：http://www.learn-cocos2d.com/2010/06/prefer-composition-inheritance

让我们来看一下StandardShootComponent组件的源代码。先看一下头文件：

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"

@interface StandardShootComponent : CCSprite
{
	int updateCount;
	int shootFrequency;
	NSString* bulletFrameName;
}

@property (nonatomic) int shootFrequency;
@property (nonatomic, copy) NSString* bulletFrameName;
@end
```

请注意两件事情。第一件事是：虽然StandardShootComponent没有使用任何的贴图，但是它继承自CCSprite类。继承自CCSprite的原因是因为所有EnemeyEntity对象都会被添加到同一个CCSpriteBatchNode中，而CCSpriteBatchNode只能包含基于CCSprite的对象。同样的限制也延伸到EnemyEntity类的任何子节点上，所以StandardShootComponent必须继承自CCSprite以满足CCSpriteBatchNode的要求。

第二件事是：bulletFrameName这个NSString指针前的@property中带了一个名为 copy 的关键词。这意味着为bulletFrameName 赋值的时候会创建一个新的拷贝。这样做的原因是因为字符串对象通常是自动释放的，但是在这里我们需要一个属于自己的字符串，可以将它保留下来（retain）。不过一旦我们拥有这个bulletFrameName字符串的拷贝，我们就有责任在dealloc中将它释放。代码如**列表8-9**所示：

列表8-9. StandardShootComponent的实现代码

```
#import "StandardShootComponent.h"
#import "BulletCache.h"
#import "GameScene.h"

@implementation StandardShootComponent

@synthesize shootFrequency;
@synthesize bulletFrameName;
-(id) init
{
	if ((self = [super init]))
```

```
        {
                [self scheduleUpdate];
        }
        return self;
}

-(void) dealloc
{
        [bulletFrameName release];
        [super dealloc];
}

-(void) update:(ccTime)delta
{
        if (self.parent.visible)
        {
                updateCount++;
                if (updateCount >= shootFrequency)
                {
                        updateCount = 0;
                        GameScene* game = [GameScene sharedGameScene];
                        CGPoint startPos = ccpSub(self.parent.position,
                CGPointMake(self.parent.contentSize.width * 0.5f, 0));
                        [game.bulletCache shootBulletFrom:startPos velocity:CGPointMake(-2, 0)
                                                                frameName:bulletFrameName];
                }
        }
}
@end
```

实际的射击代码会首先检查包含着组件的父类是不是可见，比如说敌人角色是不是处于可见的状态。因为如果父类不可见，显然它不可能会进行射击。BulletCache这个类会使用传入的bulletFrameName和固定的速度值射出子弹。子弹的起始位置与组件的位置是无关的。相反，包含着组件的父类（敌人角色）的位置和 contentSize 属性被用于计算正确的子弹起始位置：在我们的例子中，这个位置是在敌人精灵的左边。

对于普通的敌人，当前的子弹起始位置startPos已经够用，但是对于Boss来说，你可能希望子弹是从它的嘴巴或者鼻子中射出来的。至于如何实现，我把这个任务留给你来完成。

另外，你也可以创建一个单独的BoosShootComponent类，然后只在Boss类中使用。对于StandardMoveComponent，Boss类可能会要求可以在屏幕右边的某个位置悬浮不动。

进行射击

我几乎忘记了-你实际上是要对敌人开火的。在接下去的ShootEmUp03项目中，我们来完成这个任务。

测试子弹是否击中目标的一个理想的地方是BulletCache类。我在这个类中添加了测试的方法。实际上，我添加了三个方法。两个方法是公开的，另一个私有方法则把一些常用的代码组织在了一起（请参考**列表8-10**）。使用两个包裹方法（Wrapper Method）（isPlayerBulletCollidingWithRect 和 isEnemyBulletCollidingWithRect）的原因是：我们可以把如何确认哪种子弹（玩家自己的子弹，还是敌人的子弹）被用于碰撞测试的这个内部细节封装起来。你也可以将 usePlayerBullets 这个参数暴露给其它类，但是这样做的话，如果你想在以后添加第三种类型的子弹，要把当前的bool类型换成enum枚举类型的时候，你就会碰到麻烦。

列表8-10. 子弹的碰撞测试

```
-(bool) isPlayerBulletCollidingWithRect:(CGRect)rect
{
	return [self isBulletCollidingWithRect:rect usePlayerBullets:YES];
}

-(bool) isEnemyBulletCollidingWithRect:(CGRect)rect
{
	return [self isBulletCollidingWithRect:rect usePlayerBullets:YES];
}

-(bool) isBulletCollidingWithRect:(CGRect)rect usePlayerBullets:(bool)usePlayerBullets
{
	bool isColliding = NO;
	Bullet* bullet;
	CCARRAY_FOREACH([batch children], bullet)
	{
		if (bullet.visible && usePlayerBullets == bullet.isPlayerBullet)
		{
			if (CGRectIntersectsRect([bullet boundingBox], rect))
			{
				isColliding = YES;

				// 通过将子弹的可视状态设为NO把子弹移除
				bullet.visible = NO;
				break;
			}
		}
	}
```

```
    return isColliding;
}
```

当然，只有处于可视状态的子弹才能和对手碰撞。通过检查子弹的isPlayerBullet属性，我们可以确保敌人不会射中自己人。碰撞测试是通过使用 CGRectIntersectsRect 来进行的，如果子弹击中了什么东西的话，子弹本省会被设置为隐身状态而消失。

因为EnemyCache类包含了所有的EnemyEntity对象，所以在EnemyCache类中调用碰撞测试方法，用于测试玩家子弹是否已经击中了敌人，是最理想的。现在这个类中包含了一个新的名为checkForBulletCollisions的方法，它会在Enemy Cache类的更新方法中被调用，用于进行碰撞测试:

```
-(void) checkForBulletCollisions
{
    EnemyEntity* enemy;
    CCARRAY_FOREACH([batch children], enemy)
    {
        if (enemy.visible)
        {
            BulletCache* bulletCache = [[GameScene sharedGameScene] bulletCache];
            CGRect bbox = [enemy boundingBox];
            if ([bulletCache isPlayerBulletCollidingWithRect:bbox])
            {
                // 这个敌人被击中了…
                [enemy gotHit];
            }
        }
    }
}
```

在上述代码中，我们可以很方便地遍历所有游戏中的敌人对象，跳过那些当前不可见的敌人对象。我们将每个可见敌人对象的boundingBox值传给BulletCache类中的isPlayerBulletCollidingWithRect方法，可以很快地知道敌人是否已被玩家的子弹击中。如果敌人被击中，EnemyEntity类中的gotHit方法将被调用，此方法会把敌人设置为隐形状态。

对于玩家飞船被敌人子弹击中的情况，我将留给你自己去完成。你必须在ShipEntity类中添加一个更新方法（update method）。然后在类中实现checkForBulletCollisions方法，并且在更新方法中调用它。你必须把isPlayerBulletCollidingWithRect方法替换成isEnemyBulletCollidingWithRect方法，然后要决定一下如果玩家飞船被敌人子弹击中的话会发生什么事情，比如说播放一段声效。

BOSS的生命值显示条

BOSS不应该很容易就被干掉。所以我们需要显示它的生命值显示条给玩家一些反馈：就是BOSS每一次被击中以后，它的生命值就会降低。添加显示条的第一步是在EnemyEntity类中添加名为 hitPoints 的成员变量，此变量用于设置BOSS可以承受的打击数量。initialHitPoints用于存储打击点数的最大值，因为在BOSS被消灭以后，我们需要重置原始的打击点数。以下是修改过的 EnemyEntity 类的头文件：

```
@interface EnemyEntity : Entity
{
        EnemyTypes type;
        int initialHitPoints;
        int hitPoints;
}
@property (readonly, nonatomic) int hitPoints;
```

为了显示BOSS的生命值，我们将会使用一个组件类来实现。以下代码展示了 HealthbarComponent 这个组件的头文件，不过看上很普通：

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"

@interface HealthbarComponent : CCSprite
{
}
-(void) reset;
@end
```

如**列表8-11**中的代码所示，HealthbarComponent类的实现代码更有趣：

列表8-11. HealthbarComponent会根据敌人剩余的打击点数来更新它的 scaleX 属性值

```
#import "HealthbarComponent.h"
#import "EnemyEntity.h"

@implementation HealthbarComponent
-(id) init
{
        if ((self = [super init]))
        {
                self.visible = NO;
                [self scheduleUpdate];
        }
        return self;
}

-(void) reset
```

```
{
        float parentHeight = self.parent.contentSize.height;
        float selfHeight = self.contentSize.height;
        self.position = CGPointMake(self.parent.anchorPointInPixels.x, parentHeight + selfHeight);
        self.scaleX = 1;
        self.visible = YES;
}

-(void) update:(ccTime)delta
{
        if (self.parent.visible)
        {
                NSAssert([self.parent isKindOfClass:[EnemyEntity class]], @"not a EnemyEntity");
                EnemyEntity* parentEntity = (EnemyEntity*)self.parent;
                self.scaleX = parentEntity.hitPoints / (float)parentEntity.initialHitPoints;

        }else if (self.visible){
                self.visible = NO;

        }
}
@end
```

生命值显示条会随着它的 EnemyEntity父类的显示状态变化而变化。reset方法会把生命值显示条放在 EnemyEntity精灵的头顶。因为生命值是通过改变显示条的scaleX属性来实现的，所以显示条本身也需要被重置回默认的状态。

当父类处于可见状态时，HealthbarComponent组件首先在更新方法中确保包含它的类是EnemyEntity类。因为这个组件依赖于只存在EnemyEntity类中的某些属性，我们需要确保包含显示组件的类是EnemyEntity。我们通过改变显示组件的scaleX属性更新显示条的状态，scaleX属性是通过将当前剩余的打击点数除以起始的打击点数所得到的百分比计算而来的。因为目前我没有添加用于检测何时打击点数发生了变化，每一帧都会进行上述计算，不管它是不是需要。虽然由此产生的系统开销是很小的，但是对于更加复杂的计算，从诸如onHit这样存在于EnemyEntity类里的方法中调用HealthbarComponent组件会更好。

注：上述代码中，我把 parentEntity.initialHitPoints转换成了浮点数。如果我不这样做的话，上述除法计算的结果将会得到一个整数。因为整数不能表示小数点后的数字，所以在我们的例子里，计算结果将会是0。把除数转换成浮点数，就可以确保除法计算可以正确地进行，也能得到正确的结果。

在 EnemyEntity类的init方法中，如果敌人类型是 EnemyTypeBoss，HealthbarComponent对象就会被生成和添加到敌人对象中：

```
if (type == EnemyTypeBoss)
{
        HealthbarComponent* healthbar = [HealthbarComponent
                                                spriteWithSpriteFrameName:@"healthbar.png"];
        [self addChild:healthbar];
}
```

我对 spawn 方法进行了扩展，这样就可以在敌人出现的时候把它们的生命值设为初始值。我在这里删除了用于检查出现的敌人是不是BOSS的代码，因为HealthbarComponent是通用的，可以用于任何一种敌人身上：

```
-(void) spawn
{
        // 在屏幕的右侧边缘外部，选择一个地方让敌人出现
        CGRect screenRect = [GameScene screenRect];
        CGSize spriteSize = [self contentSize];
        float xPos = screenRect.size.width + spriteSize.width * 0.5f;
        float yPos = CCRANDOM_0_1() * (screenRect.size.height - spriteSize.height) +
                                                        spriteSize.height * 0.5f;
        self.position = CGPointMake(xPos, yPos);

        // 最后把自己设为可见状态，这也意味着敌人处于“使用状态”
        self.visible = YES;

        // 重置生命值
        hitPoints = initialHitPoints;

        // 重置某些组件
        CCNode* node;
        CCARRAY_FOREACH([self children], node)
        {
                if ([node isKindOfClass:[HealthbarComponent class]])
                {
                        HealthbarComponent* healthbar = (HealthbarComponent*)node;
                        [healthbar reset];
                }
        }
}
```

结语

制作一款完整和优秀的游戏是需要付出很多努力的。制作过程牵涉很多重构，修改已经工作的代码以优化代码设计，还要允许在以后添加新功能时能和现有功能和睦相处。

本章你学习了BulletCache和EnemyCache这些类的价值所在：用它们可以管理类的所有实例，允许你从一个中心点访问它们。同时，把对象（子弹，敌人）放在一起也可以提高运行效率。

Entity类的层级结构说明了你可以将各种游戏对象的类分开，但是并不需要为每一个游戏对象创建单独的类。通过使用组件类和利用cocos2d的节点层级，你可以创建“即插即用”式的拥有特定功能的类（组件）。这可以帮助你使用“复合”而不是“继承”来创建你游戏中的对象。你可以在编写游戏逻辑代码时得到更多的自由度和实现更好的代码重用。

最后，我们学习了如何对敌人进行射击和如何让BulletCache和EnemyCache以直接的方式执行这样的任务。生命值显示条组件完美展示了组件系统是如何工作的。

我们的游戏到现在为止还剩下一些事情可以做。首先也是最重要的，玩家飞船还不会被敌人击中。你也可能想给蛇这个敌人角色添加个生命值显示条，并且给Boss设计一些特殊的移动和射击组件。总的来说，现在我们有一个很好的横向滚屏射击游戏的起始点，等待你增加更多的功能和提高代码的质量。

我将在下一章展示如何在我们的射击游戏中使用粒子效果，添加一些视觉特效。