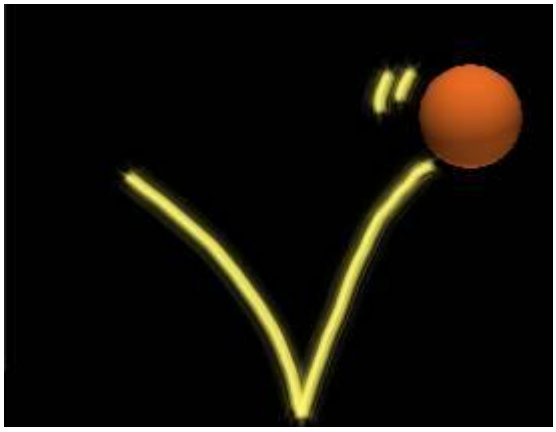
（译）在 cocos2d 里面如何使用物理引擎 box2d：弹球

整理：Taiyangmobile（泰然论坛管理组）

原文链接地址：
http://www.raywenderlich.com/457/intro-to-box2d-with-cocos2d-tutorial-bouncing-balls

程序截图：



这个教程的目的就是让你们熟悉在 cocos2d 里面如何使用 box2d，所采用的例子就是制作一个简单的应用，里面有一个篮球，你可以通过旋转你的 iPhone 来改变重力的方向，同时篮球碰到屏幕边界可以反弹。

这个教程假设你已经学过前面的教程《如何使用 cocos2d 来制作一个简单的 iphone 游戏》，或者有同等相关经验也可以。

好了，让我们开始学习 Box2d 物理引擎吧！

创建一个空的工程

打开 Xcode，选择 cocos2d-0.99.1 Box2d Application template 来创建一个新的工程，并且命名为 Box2D.如果你直接编译并且运行的话，你将会看到一个很酷的例子，里面展示了 Box2d 的许多内容。然后，这个教程的目的，我们将从 0 开始，创建一个篮球反弹的应用，这样我们就可以更好地理解那个范例的具体原理。

因此，让我们把 HelloWorld 模板里面的内容都删除掉，因为我们要从 0 开始。把 HelloWorldScene.h 里面的内容替换成下面的代码：

```
#import "cocos2d.h"
```

```
@interface HelloWorld : CCLayer {
}

+ (id) scene;

@end
```

同时修改 HelloWorldScene.mm 文件：（为什么后缀是.mm，因为 box2d 是 c++写的，而 objective-c++的实现文件必须是.mm 后缀，否则你编译会出 n 个错误！）

```
#import "HelloWorldScene.h"

@implementation HelloWorld

+ (id)scene {

CCScene *scene = [CCScene node];
HelloWorld *layer = [HelloWorld node];
[scene addChild:layer];
return scene;

}

- (id)init {

if ((self=[super init])) {
}
return self;
}

@end
```

最后一步----验证一下，你的 Classes 分组下面的所有文件（比如 HelloWorldScene）是以.mm 文件结尾的，如果是.m，那么请改成.mm，否则等下使用 Box2d 的时候，编译器会报出一大堆莫名其妙的错误！

如果你编译并运行，你应该看到一个黑色的屏幕。好了，现在让我们开始创建 Box2d 场景吧。

Box2D 世界相关理论

在我们开始之前，让我们先交待一下 Box2D 具体是如何运作的。

你需要做的第一件事情就是，当使用 cocos2d 来为 box2d 创建一个 world 对象的时候。这个 world 对象管理物理仿真中的所有对象。

一旦我们已经创建了这个 world 对象，接下来需要往里面加入一些 body 对象。body 对象可以随意移动，可以是怪物或者飞镖什么的，只要是参与碰撞的游戏对象都要为之创建一个相应的 body 对象。当然，也可以创建一些静态的 body 对象，用来表示游戏中的台阶或者墙壁等不可以移动的物体。

为了创建一个 body 对象，你需要做很多事情--首先，创建一个 body 定义结构，然后是 body 对象，再指定一个 shap，再是 fixture 定义，然后再创建一个 fixture 对象。下面会一个一个解释刚刚这些东西。

- 你首先创建一个 body 定义结构体，用以指定 body 的初始属性，比如位置或者速度。
- 一旦创建好 body 结构体后，你就可以调用 world 对象来创建一个 body 对象了。
- 然后，你为 body 对象定义一个 shape，用以指定你想要仿真的物体的几何形状。
- 接着创建一个 fixture 定义，同时设置之前创建好的 shape 为 fixture 的一个属性，并且设置其它的属性，比如质量或者摩擦力。
- 最后，你可以使用 body 对象来创建 fixture 对象，通过传入一个 fixture 的定义结构就可以了。
- 请注意，你可以往单个 body 对象里面添加很多个 fixture 对象。这个功能在你创建特别复杂的对象的时候非常有用。比如自行车，你可能要创建 2 个轮子，车身等等，这些 fixture 可以用关节连接起来。

只要你把所有需要创建的 body 对象都创建好之后，box2d 接下来就会接管工作，并且高效地进行物理仿真---只要你周期性地调用 world 对象的 step 函数就可以了。

但是，请注意，box2d 仅仅是更新它内部模型对象的位置--如果你想让 cocos2d 里面的 sprite 的位置也更新，并且和物理仿真中的位置相同的话，那么你也需要周期性地更新精灵的位置。

好了，现在有一些基本的了解了，还是先看看代码吧！

Box2d World 实战

好了，下载我制作的篮球图片，并且把它添加到工程里去吧。下载完后，直接拖到 Resources 文件夹下，同时确保“Copy items into destination group’s folder (if needed)”被复选中。

接下来，在 HelloWorldScene.mm 文件顶部添加下面的代码：

```
#define PTM_RATIO 32.0
```

这里定义了一个“像素/米”的比率。当你在 cocos2d 里面指定一个 body 在哪个位置时，你使用的单位要是米。但是，我们之前使用的都是像素作为单位，那样的话，位置就会不正确。根据 Box2d 参考手册，Box2d 在处理大小在 0.1 到 10 个单元的对象的时候做了一些优化。这里的 0.1 米大概就是一个杯子那么大，10 的话，大概就是一个箱子的大小。

因此，我们并不直接传递像素，因为一个很小的对象很有 60×60 个像素，那已经大大超过了 box2d 优化时所限定的大小。因此，如果我们有一个 64 像素的对象，我们可以把它除以 PTM_RATIO，得到 2 米---这个长度，box2d 刚好可以很好地用来做物理仿真。

好了，现在来点有意思的东西。在 HelloWorldScene.h 文件顶部添加下列代码：

```
#import "Box2D.h"
```

同时在 HelloWorld 类中添加以下成员变量：

```
b2World *_world;
b2Body *_body;
CCSprite *_ball;
```

然后，在 HelloWorldScene.mm 的 init 方法中加入下面的代码：

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

// Create sprite and add it to the layer
_ball = [CCSprite spriteWithFile:@"Ball.jpg" rect:CGRectMake(0, 0, 52, 52)];
_ball.position = ccp(100, 100);
[self addChild:_ball];

// Create a world
b2Vec2 gravity = b2Vec2(0.0f, -30.0f);
bool doSleep = true;
_world = new b2World(gravity, doSleep);

// Create edges around the entire screen
b2BodyDef groundBodyDef;
groundBodyDef.position.Set(0,0);
b2Body *groundBody = _world->CreateBody(&groundBodyDef);
b2PolygonShape groundBox;
b2FixtureDef boxShapeDef;
boxShapeDef.shape = &groundBox;
groundBox.SetAsEdge(b2Vec2(0,0), b2Vec2(winSize.width/PTM_RATIO, 0));
groundBody->CreateFixture(&boxShapeDef);
groundBox.SetAsEdge(b2Vec2(0,0), b2Vec2(0, winSize.height/PTM_RATIO));
groundBody->CreateFixture(&boxShapeDef);
groundBox.SetAsEdge(b2Vec2(0, winSize.height/PTM_RATIO),
b2Vec2(winSize.width/PTM_RATIO, winSize.height/PTM_RATIO));
groundBody->CreateFixture(&boxShapeDef);
groundBox.SetAsEdge(b2Vec2(winSize.width/PTM_RATIO,
winSize.height/PTM_RATIO), b2Vec2(winSize.width/PTM_RATIO, 0));
groundBody->CreateFixture(&boxShapeDef);

// Create ball body and shape
b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(100/PTM_RATIO, 100/PTM_RATIO);
ballBodyDef.userData = _ball;
_body = _world->CreateBody(&ballBodyDef);

b2CircleShape circle;
```

```
circle.m_radius = 26.0/PTM_RATIO;

b2FixtureDef ballShapeDef;
ballShapeDef.shape = &circle;
ballShapeDef.density = 1.0f;
ballShapeDef.friction = 0.2f;
ballShapeDef.restitution = 0.8f;
_body->CreateFixture(&ballShapeDef);

[self schedule:@selector(tick:)];
```

呃，这里有很多陌生的代码。我们一点点来解释一下。下面，我会一段段地重复上面的代码，那样可以解释地更加清楚一些。

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

// Create sprite and add it to the layer
_ball = [CCSprite spriteWithFile:@"Ball.jpg" rect:CGRectMake(0, 0, 52, 52)];
_ball.position = ccp(100, 100);
[self addChild:_ball];
```

首先，我们往屏幕中间加入一个精灵。如果你看了前面的教程的话，这里应该没有什么问题。

```
// Create a world
b2Vec2 gravity = b2Vec2(0.0f, -30.0f);
bool doSleep = true;
_world = new b2World(gravity, doSleep);
```

接下来，我们创建了 world 对象。当我们创建这个对象的时候，需要指定一个初始的重力向量。这里，我们设置 y 轴方向为-30，因此，所有的 body 都会往屏幕下面下落。同时，我们还指定了一个值，用以指明对象不参与碰撞时，是否可以“休眠”。一个休眠的对象将不会花费处理时间，直到它与其实对象发生碰撞的时候才会“醒”过来。

```
b2BodyDef groundBodyDef;
groundBodyDef.position.Set(0,0);
b2Body *groundBody = _world->CreateBody(&groundBodyDef);
b2PolygonShape groundBox;
b2FixtureDef boxShapeDef;
boxShapeDef.shape = &groundBox;
groundBox.SetAsEdge(b2Vec2(0,0), b2Vec2(winSize.width/PTM_RATIO, 0));
groundBody->CreateFixture(&boxShapeDef);
groundBox.SetAsEdge(b2Vec2(0,0), b2Vec2(0, winSize.height/PTM_RATIO));
groundBody->CreateFixture(&boxShapeDef);
groundBox.SetAsEdge(b2Vec2(0, winSize.height/PTM_RATIO),
b2Vec2(winSize.width/PTM_RATIO, winSize.height/PTM_RATIO));
groundBody->CreateFixture(&boxShapeDef);
groundBox.SetAsEdge(b2Vec2(winSize.width/PTM_RATIO,
winSize.height/PTM_RATIO), b2Vec2(winSize.width/PTM_RATIO, 0));
```

```
groundBody->CreateFixture(&boxShapeDef);
```

接下来，我们为整个屏幕创建了一圈不可见的边。具体的步骤如下：

- 首先创建一个 body 定义结构体，并且指定它应该放在左下角。
- 然后，使用 world 对象来创建 body 对象。（注意，这里一定要使用 world 对象来创建，不能直接 new，因为 world 对象会做一些内存管理操作。）
- 接着，为屏幕的每一个边界创建一个多边形 shape。这些“shape”仅仅是一些线段。注意，我们把像素转换成了“meter”。通过除以之前定义的比率来实现的。
- 再创建一个 fixture 定义，指定 shape 为 polygon shape。
- 再使用 body 对象来为每一个 shape 创建一个 fixture 对象。
- 注意：一个 body 对象可以包含许许多多的 fixture 对象。

```
// Create ball body and shape
b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(100/PTM_RATIO, 100/PTM_RATIO);
ballBodyDef.userData = _ball;
_body = _world->CreateBody(&ballBodyDef);

b2CircleShape circle;
circle.m_radius = 26.0/PTM_RATIO;

b2FixtureDef ballShapeDef;
ballShapeDef.shape = &circle;
ballShapeDef.density = 1.0f;
ballShapeDef.friction = 0.2f;
ballShapeDef.restitution = 0.8f;
_body->CreateFixture(&ballShapeDef);
```

接下来，我们创建篮球的 body。这个步骤和之前创建地面的 body 差不多，但是有下面一些差别需要注意一下：

- 我们指定 body 的类型为 dynamic body。默认值是 static body，那意味着那个 body 不能被移动也不会参与仿真。很明显，我们想让篮球参与仿真。
- 设置 body 的 user data 属性为篮球精灵。你可以设置任何东西，但是，你设置成精灵会很方便，特别是当两个 body 碰撞的时候，你可以通过这个参数把精灵对象取出来，然后做一些逻辑处理。
- 这里使用了一个不同的 shape 类型--circle shape。
- 在这里，我们需要为这个 fixture 指定一些参数，因此，我们没有使用便捷方法来创建 fixture。后面我们会讲到这些参数的具体意义。

```
[self schedule:@selector(tick:)];
```

最后一件事情就是调度一个 tikck 方法，这个方法默认是 0.1 秒回调一次。注意，这并不是最好的处理方式---最好的方式应该是让 tick 方法有固定的频率（比如每秒 60 次）。然后，这个教程我们就先这样了。

因此，让我们来实现 tick 方法。在 init 方法之后加入下面的代码：

```
- (void)tick:(ccTime) dt {

_world->Step(dt, 10, 10);
for(b2Body *b = _world->GetBodyList(); b; b=b->GetNext()) {
if (b->GetUserData() != NULL) {
CCSprite *ballData = (CCSprite *)b->GetUserData();
ballData.position = ccp(b->GetPosition().x * PTM_RATIO,
b->GetPosition().y * PTM_RATIO);
ballData.rotation = -1 * CC_RADIANS_TO_DEGREES(b->GetAngle());
}
}

}
```

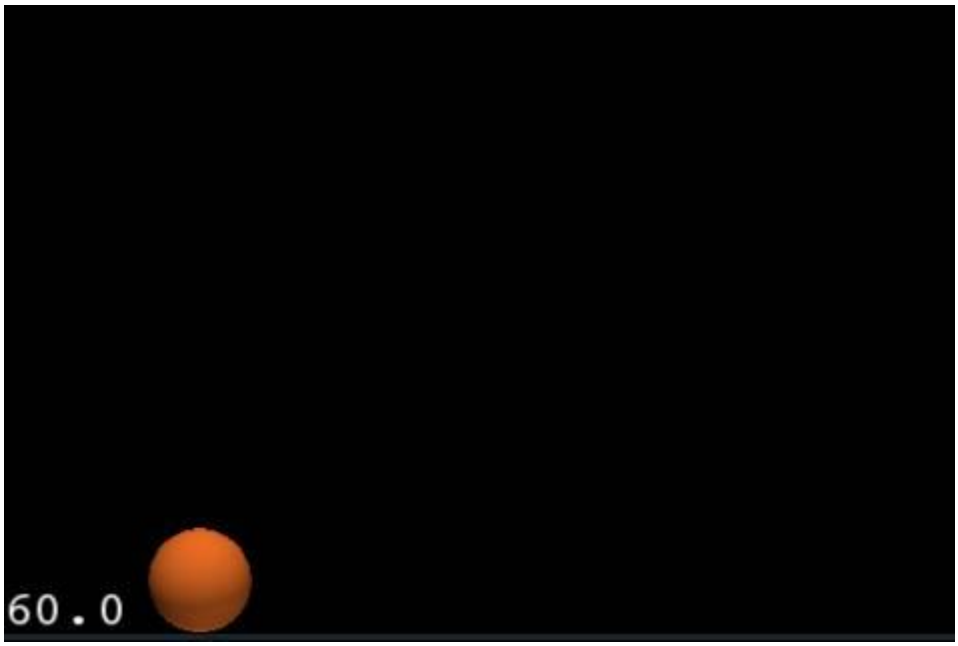
第一件事情就是调用 world 对象的 step 方法，这样它就可以进行物理仿真了。这里的两个参数分别是"速度迭代次数"和"位置迭代次数"--你应该设置他们的范围在 8-10 之间。（译者：这里的数字越小，精度越小，但是效率更高。数字越大，仿真越精确，但同时耗时更多。8 一般是个折中，如果学过数值分析，应该知道迭代步数的具体作用）。

接下来，我们要使我们的精灵匹配物理仿真。因此，我们遍历 world 对象里面的所有 body，然后看 body 的 user data 属性是否为空，如果不为空，就可以强制转换成精灵对象。接下来，就可以根据 body 的位置来更新精灵的位置了。

最后一件事---清理内存！因此，在文件的末尾加入下面的代码：

```
- (void)dealloc {
delete _world;
_body = NULL;
_world = NULL;
[super dealloc];
}
```

编译并运行，你应该可以看到球会往下掉，并且会从屏幕底部往上面弹起来。



关于仿真的一些注意事项

前面我们说后面会讨论 density，friction 和 restitution 参数的意义。

- *Density* 就是单位体积的质量（密度）。因此，一个对象的密度越大，那么它就有更多的质量，当然就会越难以移动.
- *Friction 就是摩擦力。它的范围是 0-1.0， 0 意味着没有摩擦，1 代表最大摩擦，几乎移不动的摩擦。*
- *Restitution* 回复力。它的范围也是 0 到 1.0. 0 意味着对象碰撞之后不会反弹，1 意味着是完全弹性碰撞，会以同样的速度反弹。

建议多去改一改这些参数，看看具体会给小球带来什么影响。一定要去试哦！

完成加速计控制

如果我们可以通过倾斜屏幕让球朝着屏幕的某个方向运行，那将会很棒。首先，我们需要在 init 方法里面加入下面的代码：

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

然后，在文件的某个位置加入下面的方法：

```
- (void)accelerometer:(UIAccelerometer *)accelerometer didAccelerate:(UIAccel
eration *)acceleration {

// Landscape left values
b2Vec2 gravity(-acceleration.y * 15, acceleration.x *15);
_world->SetGravity(gravity);
```

```
}
```

这里就是设置加速计的向量乘以某个数，然后再设置为 world 对象的重力向量。编译并运行（最好编译到设备上面，只有设备上面才有加速计），看看效果吧！

何去何从？

这里是完整源代码。

如果你想学习更多有关 box2d 相关的内容，请看下一篇教程《在 cocos2d 里面如何使用 box2d 制作一个 Breakout 游戏：第一部分》。