第1章

UIKit Dynamics

苹果公司在 iOS 7 版本中引入了 UIKit Dynamics 框架，使用该框架，开发者可以很容易地将真实的物理模拟动作应用在 UIView 上。在之前的版本中，开发者只能将这种真实的动作效果整合到部分程序中，比如可滑动的单元格和下拉刷新动画等。苹果公司在 iOS 7 和 iOS 8 版本中向前迈进了一大步，将这些动画加入到核心 OS 库中，同时也在极力鼓励开发者使用它们实现动画效果。

UIDynamicItem 协议和支持该协议的动态元素使用户体验得到了极大提升。在程序中添加重力、碰撞、弹跳、瞬间位移等效果变得异常简单。介绍这些动态元素的 API 很简单，且很容易实现，应用这些功能提升用户体验易如反掌。

1.1 示例程序

示例程序(如图 1-1 所示)是一个基本的表格元素，用来展示 UIKit Dynamics 各种不同的功能。在这个程序中一共展示了从重力感应到属性设置等 7 个效果，每个效果都会在后面小节中详细展开介绍。除了表视图和基本的导航视图，示例程序并不包含任何专门针对 UIKit Dynamics 的功能。

注意

在同一个视图中使用 UIKit Dynamics 和自动布局可能会导致一些布局问题。通常，这是由于自动布局与 UIKit Dynamics 争抢视图上的正确位置，导致视图出现无法预料的错位而致。如果视图没有像预想那样呈现，开发者可以检查自动布局的相关设置来查看是否出现了冲突。

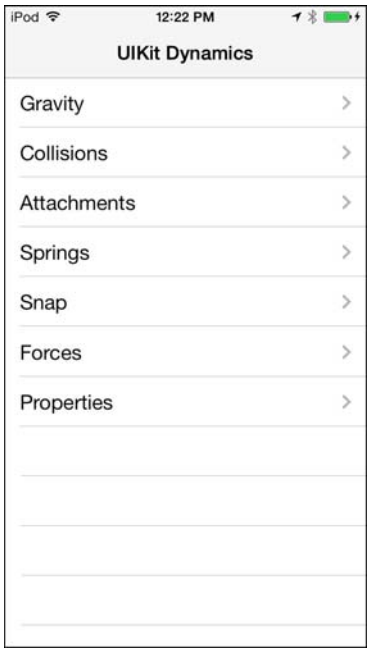

图 1-1　简单查看一下用于展示 UIKit Dynamics 各项功能的示例程序

1.2　UIKit Dynamics 介绍

UIKit Dynamics 是一组新的类和方法，最初是在 iOS 7 版本的 iDevices 中引入的。简单来说，UIKit Dynamics 通过在 UIView 视图中整合现实中的一些行为，提供了一种易于实现的方法来提升应用的用户体验。用最简短的术语来解释 UIKit Dynamics，其实它就是 UIKit 的基础物理引擎，不过它并不像传统的物理引擎一样是专为游戏开发而设计的。苹果公司提供了一些游戏框架，其中都包含了物理引擎，比如 SpriteKit。

当程序创建一个新的 UIDynamicAnimator 并将其添加到 UIView 中时，动态行为就会被激活。每个动画元素都可以对其属性和行为进行自定义，比如重力、碰撞检测、密度、摩擦力以及下面小节中将介绍的额外一些属性。

一共有 6 个附加类可以支持 UIDynamicAnimator 元素的自定义设置，分别是 UIAttachmentBehavior、UICollisionBehavior、UIDynamicItemBehavior、UIGravityBehavior、UIPushBehavior 和 UISnapBehavior。每个元素都允许指定自定义属性并且会在相应的视图中以真实的行为和动画反映出来。

1.3　UIKit Dynamics 具体实现

创建一个新的动画并将它添加到一个视图中，只需要两行代码就可以实现上述操作。示例中 self.view 对象即为将要使用 UIKit Dynamics 行为的对象。每一个特定的动态元素必须使用 addBehavior:方法添加到动画对象中。

```
UIDynamicAnimator *animator = [[UIDynamicAnimatoralloc]
initWithReferenceView:self.view];
```

```
[animatoraddBehavior:aDynamicBehavior];
```

每一个 UIDynamicAnimator 都是独立的，多个动画对象可以同时运行。对于一个持续运行的动画对象，对其的引用必须有效。当动画对象上的所有元素都处于静止状态时，动画对象此时不执行任何计算且处于暂停状态，不过实际操作中建议将不再使用的动画对象移除。

【游戏开发者的经验】

物理模拟对于游戏开发者而言已经使用了很多年了，很多难学的课程都已经学过了。现在物理层的处理技术已经蔓延到普通应用的开发中，下面介绍一些每位开发者都可以从中获益的基本原则。

当向游戏或应用添加物理特性时，请采取小步推进的方式。在多个互动代码段中试图找到出现的错误几乎是不可能的，采用越小步骤得到最终结果，程序也就越容易优化和调试。

在物理层进行处理时，有一些限制和边界在计算机模拟中无法体现。在 1997 年发布的经典游戏“死亡赛车”(Carmageddon)中，物理层处理是基于无上限帧率的。当计算机的处理速度变得越来越快后，帧率得到了大幅提升，在特定的公式中通过创建变量可以得到意想不到的结果。当把任何一种计算类型运用到物理引擎中时，需要确保其最大值和最小值都是符合要求且经过测试的。

预见下面这种意外情况：处理碰撞事件时，当 30 个对象发生重叠后，结果就会变得很扭曲。UIKit Dynamics 可以很好地确保开发者不会出现类似对象超过边界等情况，在处理上述碰撞场景时也能很完美地加以解决。不过当处理许多对象的复杂操作时也不能完全保证不出现边界情况和 bug。随着使用物理引擎的增加，越来越需要进行测试和调试，要能够预料到那些不期而遇和非常规情况下应该遵循的物理定律。

1.3.1　重力效果

重力效果被认为是最容易实现的 UIDynamicItem，同时也是实践中最常用的。苹果公司在 iOS 8 中重点强调了对重力相关元素的使用，用户操作重力相关的互动操作不需要再经过锁屏界面了。在 iOS 8 锁屏界面中，使用 UIGravityBehavior 函数可以实现向上拖动照相机图标并在到达中点之前松开手指，使屏幕再次返回锁屏状态的功能。在之前的 iOS 7 版本中引入 UIKitDynamics 时，这一功能还只能通过手工撰写计时器和传统动画相结合的方式才能实现。

下面的代码片段将在 frogImageView 视图中设置重力效果，该视图是 self.view 的子视图。首先为需要呈现动画的封闭视图创建一个新的 UIDynamicAnimator 对象，本例中呈现动画的视图即 self.view。创建一个新的 UIGravityBehavior 对象并初始化，初始化数组为使用重力效果的视图集合。设置本例中的重力行为参数：y 轴向下力度为 0.1。行为参数设置完成后，使用 addBehavior:方法将其添加到 UIDynamicAnimator 对象。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UIGravityBehavior* gravityBehavior = [[UIGravityBehavioralloc]
➥initWithItems:@[frogImageView]];
```

```
[gravityBeahviorsetXComponent:0.0f yComponent:0.1f];
[animatoraddBehavior:gravityBehavior];
```

注意

动态元素必须作为引用视图的子视图；如果动态元素不是子视图，动画对象就不会移动。

UIKit Dynamics 使用自身的物理系统，苹果工程师将其戏称为 UIKit Newtons。虽然它同标准的公式没有直接的关联，苹果公司还是做到了非常近似的效果。力度 1.0 大致等于 9.80655 m/s^2，即地球的重力。要使用地球重力十分之一的力，也就是 0.1 的力度。在 UIKit Dynamics 框架中使用重力效果不需要特别指定方向，默认就是向下的重力。如果 yComponent 参数为负值，重力方向才是向上的。同样，重力可以指定为沿 X 轴。元素还有密度属性，我们会在后面的 1.3.7 节中详细进行介绍。

运行重力效果示例代码，结果是 imageView 视图以大约十分之一地球重力的速度滑落(如图 1-2 所示)，并且完全滑出了屏幕。这是因为我们没有设置边界或碰撞事件，对象并不知道要碰到哪个边界才能停止运动，所以就一直滑落下去。

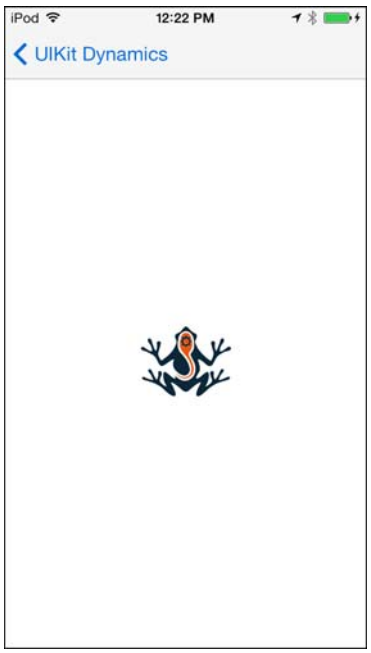

图 1-2　带有重力效果的图片视图在重力效果示例中向下滑到屏幕底端

1.3.2　碰撞效果

上一节中介绍了重力，不过应用了重力的对象会从屏幕底端滑出并一直无限地滑下去。这是因为我们没有在对象下降过程中设置碰撞点来阻止它。

我们对上一个示例程序进行修改，对封闭视图添加碰撞边界，同时添加第二个图片对象。碰撞示例的开始部分同重力示例一样，不过这里使用了两个图片视图。

创建 UICollisionBehavior 对象的过程同创建 UIGravityBehavior 对象的过程类似。对象通过将要应用效果的一些 UIView 视图进行初始化，本例中即两个 UIImageViews。除了视图之外，碰撞行为还需要在以下三个值中指定一个值作为参数：使用 UICollisionBehaviorModeItems 参数会使元素相互碰撞；使用 UICollisionBehaviorModeBoundaries 参数，元素虽然不会发生碰撞，但是会和边界发生碰撞；使用 UICollisionBehaviorModeEverything 参数会使元素在相互之间和同边界都发生碰撞。

要想程序中的对象和边界发生互动关系，需要先对边界进行定义。最简单的定义方法就是通过在 UICollisionBehavior 对象上设置一个称为 translatesReferenceBoundsIntoBoundary 的布尔值参数，在示例中我们将其用于 self.view 对象。边界也可以被设置为遵循某种路径，使用 addBoundaryWithIdentifier:forPath:方法使其遵循 NSBezierPath 表示的路径，或者使用 addBoundaryWithIdentifier:fromPoint:toPoint:方法使其遵循基于两个点表示的路径。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UIGravityBehavior* gravityBehavior = [[UIGravityBehavioralloc]
➥initWithItems:@[frogImageView, dragonImageView]];

[gravityBehaviorsetXComponent:0.0f yComponent:1.0f];


UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[frogImageView, dragonImageView]];

[collisionBehaviorsetCollisionMode:UICollisionBehaviorModeBoundaries];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;

[animatoraddBehavior:gravityBehavior];
[animatoraddBehavior:collisionBehavior];
```

UICollisionBehavior 还提供了一个代理回调用于遵照 UICollisionBehaviorDelegate 协议。

```
collisionBehavior.collisionDelegate = self;
```

UICollisionBehaviorDelegate 函数有 4 个回调方法，两个用于碰撞开始，另两个用于碰撞结束。每一个回调集都有一个方法用来定义边界是否出现碰撞。所有方法都提供了对导致回调方法触发的对象的引用。检测碰撞开始的方法还提供了一个 CGPoint 对象，用于记录碰撞发生的确切位置。示例代码会在对象检测到碰撞时更新标签的显示。

```
-(void)collisionBehavior:(UICollisionBehavior *)behavior
➥beganContactForItem:(id<UIDynamicItem>)item
➥withBoundaryIdentifier:(id<NSCopying>)identifier atPoint:(CGPoint)p
{
    if([item isEqual:frogImageView])
         collisionOneLabel.text = @"Frog Collided";
    if([item isEqual:dragonImageView])
```

```
            collisionTwoLabel.text = @"Dragon Collided";
    }


    -(void)collisionBehavior:(UICollisionBehavior *)behavior
    ➥endedContactForItem:(id<UIDynamicItem>)item
    ➥withBoundaryIdentifier:(id<NSCopying>)identifier
    {

       NSLog(@"Collision did end");
    }
```

1.3.3 附着效果

附着效果定义了两个对象间的动态连接，可实现两个移动对象间的绑定关系。默认情况下，UIAttachmentBehaviors 都是固定在对象中心的，附着点可以通过程序自行设置为对象上的任何一点。

本节示例程序的创建以上一节“碰撞效果”中的程序为基础。仍然使用两个图片视图，边界碰撞已经创建好，并且已应用到 UIDynamicAnimator 对象上。创建一个新的 CGPoint 并设置青蛙图片视图的中心点作为其关联点。创建一个新的 UIAttachmentBehavior 对象并使用 initWithItem:attachedToAnchor:对其进行初始化。这里仍然需要对 UICollisionBehavior 进行额外的初始化，从而指定具体的点和其他对象的规范。将碰撞效果和附着效果都添加到动画对象。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[dragonImageView, frogImageView]];

[collisionBehaviorsetCollisionMode:UICollisionBehaviorModeBoundaries];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;

CGPointfrogCenter = CGPointMake(frogImageView.center.x,
➥frogImageView.center.y);

self.attachmentBehavior = [[UIAttachmentBehavioralloc]
➥initWithItem:dragonImageViewattachedToAnchor:frogCenter];

[animatoraddBehavior:collisionBehavior];
[animatoraddBehavior:self.attachmentBehavior];
```

这些对象现在已经被一个长度为其初始距离的可见连接线围住，如果青蛙图片视图移动，那么龙图片视图将会保持中心点不变而随之移动。不过现在青蛙图片还不具有移动的能力，为解决这个问题，示例程序需要实现一个简单的平移手势。当青蛙图片视图在主视图中移动时，中心点位置更新的同时会设置更新的锚点。

```
-(IBAction)handleAttachmentGesture:(UIPanGestureRecognizer*)gesture
{
    CGPointgesturePoint = [gesture locationInView:self.view];

    frogImageView.center = gesturePoint;
    [self.attachmentBehaviorsetAnchorPoint:gesturePoint];
}
```

在移动过程中，碰撞边界始终有效并且覆盖了附着效果，这一点可以通过将龙图片移到视图边界上进行验证。

为了改变两个对象间的附着距离，还可以对附着视图的长度属性进行更新。附着点自身不需要是对象的中心点，可以通过调用 setAnchorPoint 方法设置任意偏移量作为附着点。

1.3.4　弹跳效果

弹跳效果(如图 1-3 所示)是上述附着效果的扩展。UIKit Dynamics 框架可以在 UIAttachmentBehavior 上额外设置一些属性，比如频率和阻尼等。

图 1-3　将弹跳效果应用在龙图片和青蛙图片上，展示了重力效果和设置 UIAttachmentBehavior 阻尼、频率的效果

下面小节的示例程序在创建 UIAttachmentBehavior 对象后又对其添加了三个新的 UIKit Dynamic 属性。第一个是 setFrequency，用于设置对象的振幅或摆动大小；第二个是 setDamping，用于校平动画峰值；第三个是 setLength，该属性也是根据其初始位置进行调整的。为更好地展示上述行为效果，我们仍然在例子中添加了重力效果。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];
```

```
UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[dragonImageView, frogImageView]];

UIGravityBehavior* gravityBeahvior = [[UIGravityBehavioralloc]
➥initWithItems:@[dragonImageView]];

CGPointfrogCenter = CGPointMake(frogImageView.center.x,
➥frogImageView.center.y );

self.attachmentBehavior = [[UIAttachmentBehavioralloc]
➥initWithItem:dragonImageViewattachedToAnchor:frogCenter];

[self.attachmentBehaviorsetFrequency:1.0f];
[self.attachmentBehaviorsetDamping:0.1f];
[self.attachmentBehaviorsetLength:100.0f];

[collisionBehaviorsetCollisionMode: UICollisionBehaviorModeBoundaries];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;

[animatoraddBehavior:gravityBeahvior];
[animatoraddBehavior:collisionBehavior];
[animatoraddBehavior:self.attachmentBehavior];
```

现在，在屏幕中移动青蛙图片会使龙图片由底部上行 100 像素并按照设置好的附着效果和重力效果进行摇摆。

1.3.5 瞬间位移

元素可以在视图中动态地瞬间位移至另一个点，该功能非常容易实现。在示例程序中，动作已经绑定了一个点击手势，点击屏幕中的任何位置都会使指定图片瞬间位移到指定的锚点。每个 UISnapBehavior 一次仅能关联一个单独的元素，并且在初始化过程中元素终点的位置已被设置好。另一个属性阻尼系数也可以被指定，用于影响动作的移动速度。

```
CGPoint point = [gesture locationInView:self.view];
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UISnapBehavior* snapBehavior = [[UISnapBehavioralloc]
➥initWithItem:frogImageViewsnapToPoint:point];

snapBehavior.damping = 0.75f;
[animatoraddBehavior:snapBehavior];
```

1.3.6 推力效果

UIKit Dynamics 还支持对力度的运用，比如推力。UIPushBehavior 的使用比之前的几种动作效果稍微复杂一些，不过同其他物理引擎相比还是很容易使用的。示例仍然使用了一个前面测试程序中用到的 UICollisionBehavior 对象，这确保了在推力效果作用时图片视图始终

处于屏幕范围内。

创建一个新的 UIPushBehavior 并使用一个图片视图初始化它。目前方向和加速度两个属性的值为 0.0。

示例程序还在屏幕中心用一个小黑方块的形式描绘了一个引用。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[dragonImageView]];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;
[animatoraddBehavior:collisionBehavior];

UIPushBehavior *pushBehavior = [[UIPushBehavioralloc]
➥initWithItems:@[dragonImageView]
➥mode:UIPushBehaviorModeInstantaneous];

pushBehavior.angle = 0.0;
pushBehavior.magnitude = 0.0;

self.pushBehavior = pushBehavior;
[animatoraddBehavior:self.pushBehavior];
```

如果现在运行项目，图片视图将会始终固定在屏幕上，因为还没有对推力效果设置任何参数。创建一个新的平移手势，在其关联动作中会计算出 magnitude 与 angle 的新值并应用其中。在该例中，我们计算出一个角度，用于确定推力的来源。角度是基于中间参考点的，运动距离还要根据持续增加的力度进行计算。点击黑色方块的外面区域将会有一个力沿相应的方向作用于图片视图。图片离得越远，力度就越大。

```
CGPoint point = [gesture locationInView:self.view];

CGPoint origin = CGPointMake(CGRectGetMidX(self.view.bounds),
➥CGRectGetMidY(self.view.bounds));

CGFloat distance = sqrtf(powf(point.x-origin.x, 2.0)+powf(point.y-
➥origin.y, 2.0));

CGFloat angle = atan2(point.y-origin.y, point.x-origin.x);
distance = MIN(distance, 100.0f);

[self.pushBehaviorsetMagnitude:distance/100.0];
[self.pushBehaviorsetAngle:angle];

[self.pushBehaviorsetActive:YES];
```

除了可以手动设置角度和加速度的值之外，还可以对指定的目标点使用 setTargetPoint:forItem:方法自动进行计算。有必要对视图中的一部分施加力的效果，受力点并不一定是对象中心点，使用 setXComponent:yComponent:方法可以指定一个 CGPoint 类型的点作为受力点。

有两种类型的推力可以应用——UIPushBehaviorModeContinuous 和 UIPushBehaviorMode-Instantaneous，持续的力会推动对象不断加速，瞬间的力则会直接作用到对象上。

1.3.7　元素属性

动态元素有许多默认的预设属性，这些属性都可以自定义设置，用来表示对象针对物理引擎的不同响应。示例程序(如图 1-4 所示)展示了对一个图片视图修改默认属性和对另一个图片视图保留默认属性的对比效果。

图 1-4　修改动态元素的默认属性，展示在同一力的作用下得到的不同物理响应

要修改对象的属性，首先需要创建一个新的 UIDynamicItemBehavior 并使用相应的视图对象初始化它。结果就是其中一个对象动起来像一个橡皮球，对其应用了重力和碰撞效果后更容易弹跳。具体的属性和描述参见表 1-1。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UIGravityBehavior* gravityBeahvior = [[UIGravityBehavioralloc]
➥initWithItems:@[dragonImageView, frogImageView]];

UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[dragonImageView, frogImageView]];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;

UIDynamicItemBehavior* propertiesBehavior = [[UIDynamicItemBehavior
➥alloc] initWithItems:@[frogImageView]];

propertiesBehavior.elasticity = 1.0f;
```

```
propertiesBehavior.allowsRotation = NO;
propertiesBehavior.angularResistance = 0.0f;
propertiesBehavior.density = 3.0f;
propertiesBehavior.friction = 0.5f;
propertiesBehavior.resistance = 0.5f;

[animatoraddBehavior:propertiesBehavior];
[animatoraddBehavior:gravityBehavior];
[animatoraddBehavior:collisionBehavior];
```

表 1-1　UIDynamicItem 属性及其描述

属　　性	描　　述
allowsRotation	一个布尔值，用于指定元素是否根据力度进行旋转，默认值是 YES
angularResistance	一个位于 0.0 到 CGFLOAT_MAX 范围内的 CGFloat 值，用于指示带有角度的阻尼值，这个数值越大，旋转的减速就越快
density	用于表示密度。100×100 对象的默认密度值为 1.0，100×200 对象的默认密度值为 2.0。调整密度的大小会影响重力和碰撞效果的反映
elasticity	有效值介于 0.0 到 1.0，用于指示当对象间发生碰撞时弹力的大小。0.0 表示不发生弹跳，1.0 表示整个力度会应用到碰撞对象上
friction	当两个元素相互滑动时的线性阻力，0.0 表示无摩擦力，1.0 表示最大摩擦力。此外，也可以使用大于 1.0 的值表示额外的阻力
resistance	开放空间中遇到的线性阻力，取值范围为 0.0 到 CGFLOAT_MAX。0.0 表示无阻力，1.0 表示当没有力作用于元素时元素应该停止移动

1.4　深入了解 UIDynamicAnimator 和 UIDynamicAnimatorDelegate

本章前面部分介绍了 UIDynamicAnimator，并且示例程序中都用到了 addBehavior 方法，不过这个类的强大功能远不止这些。除了可以添加动态效果之外，还可以每次移除一个效果或者对一组对象使用 removeBehavior:和 removeAllBehaviors 方法。要得到当前 UIDynamicAnimator 对象所有的动作行为，可以通过行为属性返回的数组进行查看。

还可以通过运行属性来查看动画的运行状态，可以使用 elapsedTime 值确定动画时长。UIDynamicAnimator 还带有一个关联的委托函数，即 UIDynamicAnimatorDelegate。该委托函数给出了两个方法用于处理暂停和重启动作。开发者无法显式暂停 UIDynamicAnimator 对象的运行，当所有元素都处于静止且不再发生运动时动画效果会自动停止。当将任何新的动作效果应用在元素上时，都会使其开始运动并返回激活状态。

```
-(void)dynamicAnimatorDidPause:(UIDynamicAnimator *)animator
{
    NSLog(@"Animator did pause");
}
```

```
-(void)dynamicAnimatorWillResume:(UIDynamicAnimator *)animator
{
    NSLog(@"Animator will resume");
}
```

1.5　小结

无论是从开发者的立场还是从 iOS 系统本身未来的发展来看，UIKit Dynamics 都是一个令人感兴趣的话题。苹果公司在花大力气将软件推广到现实世界，希望能够做到让用户同应用的交互就像同真实世界交互的体验一样。用户期望应用对人类指令的反应同周围现实世界的事务一样。对于苹果公司来说这并不新鲜，传统 iPhone 最大的一个卖点就是动量卷动(momentum scrolling)，现在它们又为开发者提供了工具，以实现向程序中添加这些功能。

本章介绍了 UIKit Dynamics 的基本概念和基本组件，不过这些方法真正的强大之处还在于开发者的运用。这个框架有无限的潜能和各种组合，开发者利用这些功能做出的产品可能令苹果公司自己都感到惊讶。用户体验已经被重新定义，那么可以确定的是软件提供真实的物理响应已经不再是可有可无的，用户也非常期待这样的技术。