A GUIDE TO IOS ANIMATION

KITTEN 的 iOS 动画学习手册

2.0

" TAKE IT EASY !
WE HAVE NO HOMEWORK THIS WEEK ! "

EDITED BY KITTEN YANG

給　更好的你

終于有時間更新第二版了

如果你還能看到這段話

那真是真愛了

我在屏幕這邊給您鞠一躬

你們的支持是本書繼續更新的潤滑劑

自打編寫之初我就堅持

這必須是一本精致的、視覺優先的電子書

每個人心靈深處都有一種創造欲

享受那種從自己手底打磨出一件美好事物的愉悅的感覺

這本書的創作過程就是這樣

更好的第二版 送給更好的你

創作歷程

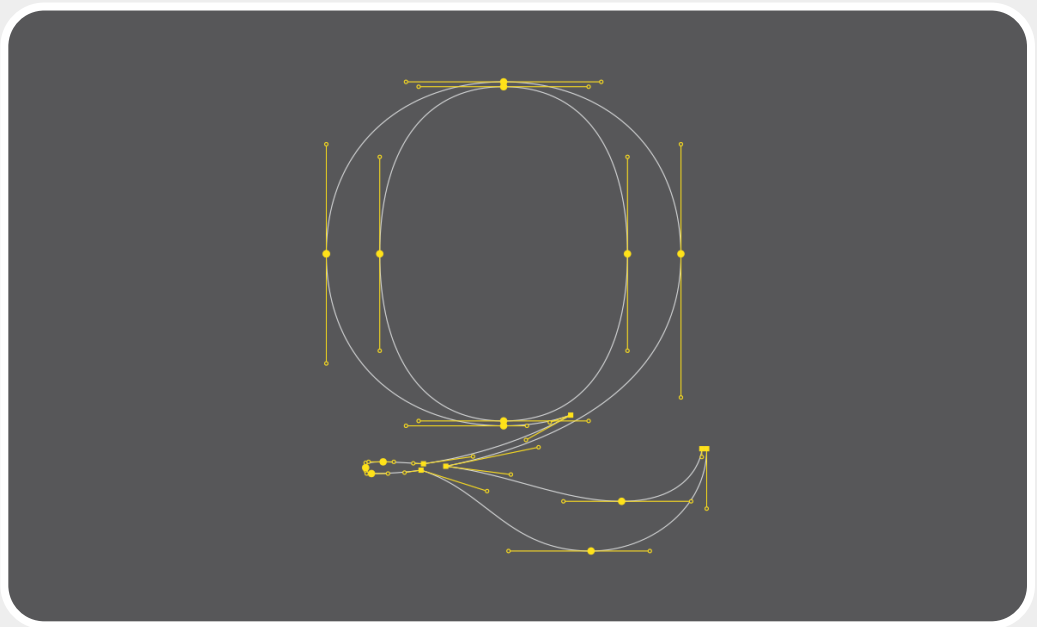
本書的第一版是我業餘時間完成的。當時我還在錘子科技上班。上班一族都有體會，尤其是作爲軟件工程師這一群體，白天已經對着電腦寫了一天代碼，晚上回到家出于本能是拒絕碰代碼的。外加美食和美劇的誘惑，導致在創作第一版的 72 天里，我有半個月的時間因偷懶而没有推進寫書的進度。更疲勞的是，我需要花費很長時間制作篇幅并不長的素材，幾百個純手工制作的素材也是這本書最大耗時之處。但好在，我挺過來了。

第二版。第二版原本打算在 12 月份更新的。但還是拖了 2 個月。當時我已經去了另一家公司，然而巨大的工作壓力讓我呆了 2 個月就毅然辭職了，這才給第二版的更新帶來了時間上的可能性。

在這本電子書中，我力圖通過自己的嘗試向所有軟件工程師亦或所有有意出書的朋友證明：如果你有想法，赶緊行動吧。坦率地講，一開始我認定寫書這種事情離我太遠，自己何德何能去指導别人。但後來我意識到，其實我們所處的社會就是一個分享的社會。没有人是天生就會寫書的，作家亦是如此，更何況作爲一個非文字工作者的軟件工程師。如今我已邁出這一步，相信你也可以。你的能力永遠超乎你想象（不是紅牛軟廣）。

KittenYang

目錄



2

玩轉貝塞爾曲線

首先非常感谢你能购买正版。你们的支持是我给我最大的动力。

提笔写下第一个字，我犹豫了很久，犹豫该以什么方式，什么内容开头。毕竟在我看来，这是一个具有颇具仪式感的时刻，这意味着这件将贯穿我一生的事情总算起步了。写书和写博客的感觉完全不一样。一本完整的书籍需要考虑更多整体上的连贯性，语言的组织，内容的合理编排来引导读者很顺畅地进入你的世界，而不像博客，可以一篇一篇硬切。好在，一切都已踉跄着起步了，虽谈不上万事俱备，但好歹已经出发。在这个出发的地方，我想留下一句话，方便时刻提醒我自己：「不要因为走得太远，就忘了当初为什么出发」。

第一章，我们先来聊聊贝塞尔曲线。贝塞尔曲线的发明人是法国雷诺汽车的工程师皮埃尔·贝塞尔。当年他把贝塞尔曲线应用在了雷诺汽车的设计上。贝塞尔曲线的出现可以说对计算机图形学的发展产生了巨大的推动作用。我们现在得以在电脑上使用 Flash，Illstrator，

CoralDRAW 和 Photoshop 上制作优美的图形，这其中都离不开贝塞尔曲线的功劳。

原理铺垫： 给定n+1个数据点，$p_0(x_0, y_0)$ ... $p_n(x_n, y_n)$，生成一条曲线，使得该曲线与这些点所连结的折线相近。

在数学中，这属于逼近问题。在几何中，可以形象地理解为先用折线段连接这些数据点，勾勒出图形的大致轮廓，然后再用光滑的曲线去尽可能接近地拟合这条折线。

1

在本章的第一节中，我将以 KYAnimatedPageControl 为例，向你介绍贝塞尔曲线在实际生产中的应用。你可以通过点击下方的 Movie 2.1，查看这个 demo 的最终效果。

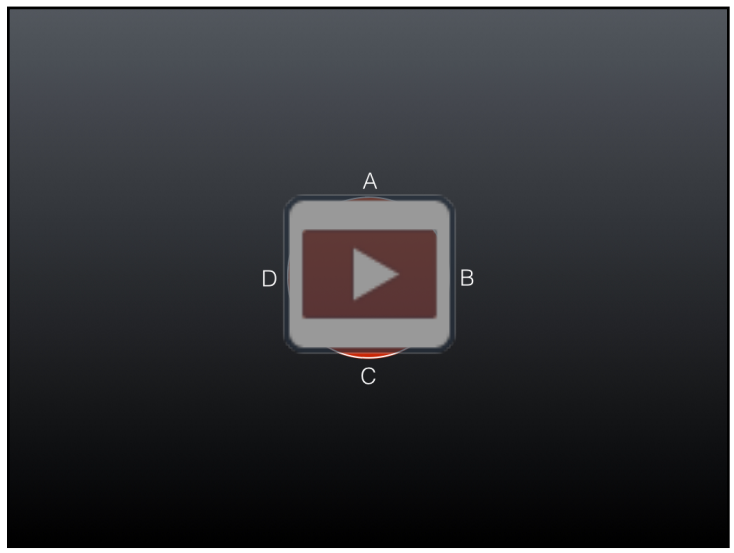
Movie 2.1 小球拖拽形变效果

初看这个效果，直观的感受就是小球发生了形变。所以一个可行的做法是：我们用四条贝塞尔曲线「拼」出这

代码来自 *KYAnimatedPageControl*

个小球的形状，注意是「拼」出，而不是一下子完整地画出来。有了这四条单独的曲线，然后，我们只需要单独控制每条贝塞尔曲线的形状，实时调用 `layer` 的 `[self setNeedsDisplay]` 以重绘 `-(void)drawInContext:(CGContextRef)ctx` 方法，就可以间接地实现控制小球形状的目的了。

KEYNOTE 2.1 如何「拼」出小球



小球由四段 *1/4* 圆弧「拼」成，连接完成之后向内填充颜色。

如左侧 KEYNOTE 2.1 所示，小球由 弧AB,弧BC,弧CD,弧DA 组成。

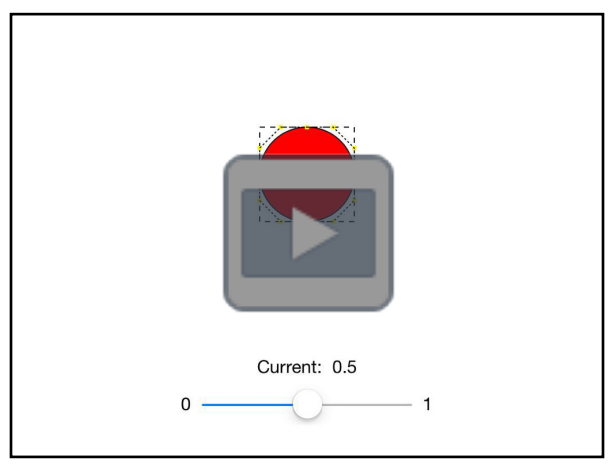
其中 弧AB 是一条由 A,B 加两个控制点 C1,C2 一共四个点绘制的三次贝塞尔曲线。其他弧线段同理。

为了方便传达理念，我以 Keynote 的形式展示了这一思路。你可以逐帧点击下方的 KEYNOTE 2.2 进行预览。双指 Pinch 可以放大观看。

那么问题来了，这些点应该以一个什么样的规律运动呢？

为了方便计算各个点的坐标，我们可以先引入一个外接矩形，也就是你在

Keynote 2.1 小球各点运动轨迹



通过各个点的不同规律的运动，产生变化的形状

右侧 Keynote 中看到的那个被虚线框起来的黑色矩形。

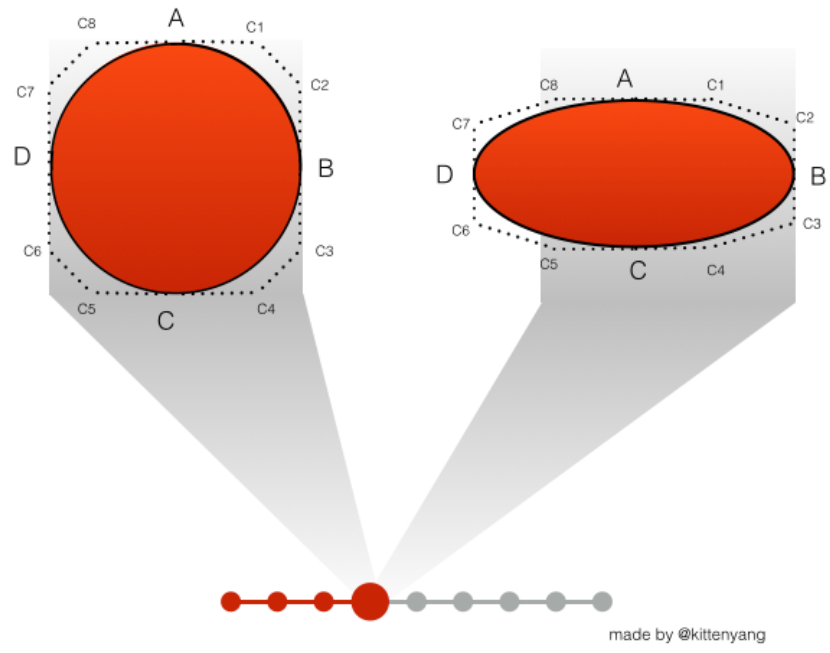
首先计算出这个外接矩形的位置。origin 根据中心点的 x(y) 减去宽度（高度）的 1/2 获得。

```
//outsideRectSize 是外接矩形边长
CGFloat origin_x = self.position.x - outsideRectSize/2 + (pro-
gress - 0.5)*(self.frame.size.width - outsideRectSize);

CGFloat origin_y = self.position.y - outsideRectSize/2;

self.outsideRect = CGRectMake(origin_x, origin_y, outsideRect-
Size, outsideRectSize);
```

其次我们还需要判断当前是向左滑还是向右滑，左滑的时候 B 动 D 不动；右滑的时候 D 动 B 不动。

```
//只要外接矩形在左侧，则改变B点；在右边，改变D点
if (progress <= 0.5) {

    self.movePoint = POINT_B;
    NSLog(@"B点动");

}else{

    self.movePoint = POINT_D;
    NSLog(@"D点动");
}
```

现在有了矩形的位置，接下来我们计算关键点的坐标就可以完全基于这个矩形做为参考了。为了便于对照代码，我用 Keynote 画出了示意图，方便你进行对照。




```
-(void)drawInContext:(CGContextRef)ctx{

    
    CGFloat offset = self.outsideRect.size.width / 3.6;

    
    CGFloat movedDistance = (self.outsideRect.size.width * 1 /
6) * fabs(self.progress-0.5)*2;

    
    CGPoint rectCenter = CGPointMake(self.outsideRect.origin.x
+ self.outsideRect.size.width/2 , self.outsideRect.origin.y +
self.outsideRect.size.height/2);

    
    CGPoint pointA = CGPointMake(rectCenter.x
,self.outsideRect.origin.y + movedDistance);
```

```
    CGPoint pointB = CGPointMake(self.movePoint == POINT_D ?
rectCenter.x + self.outsideRect.size.width/2 : rectCenter.x +
self.outsideRect.size.width/2 + movedDistance*2 ,rectCenter.y);
    CGPoint pointC = CGPointMake(rectCenter.x ,rectCenter.y +
self.outsideRect.size.height/2 - movedDistance);
    CGPoint pointD = CGPointMake(self.movePoint == POINT_D ?
self.outsideRect.origin.x - movedDistance*2 :
self.outsideRect.origin.x, rectCenter.y);


    CGPoint c1 = CGPointMake(pointA.x + offset, pointA.y);
    CGPoint c2 = CGPointMake(pointB.x, self.movePoint ==
POINT_D ? pointB.y - offset : pointB.y - offset + movedDis-
tance);

    CGPoint c3 = CGPointMake(pointB.x, self.movePoint ==
POINT_D ? pointB.y + offset : pointB.y + offset - movedDis-
tance);
    CGPoint c4 = CGPointMake(pointC.x + offset, pointC.y);

    CGPoint c5 = CGPointMake(pointC.x - offset, pointC.y);
    CGPoint c6 = CGPointMake(pointD.x, self.movePoint ==
POINT_D ? pointD.y + offset - movedDistance : pointD.y + off-
set);

    CGPoint c7 = CGPointMake(pointD.x, self.movePoint ==
POINT_D ? pointD.y - offset + movedDistance : pointD.y - off-
set);
    CGPoint c8 = CGPointMake(pointA.x - offset, pointA.y);


    //圆的边界
    UIBezierPath* ovalPath = [UIBezierPath bezierPath];
    [ovalPath moveToPoint: pointA];
    [ovalPath addCurveToPoint:pointB controlPoint1:c1
controlPoint2:c2];
    [ovalPath addCurveToPoint:pointC controlPoint1:c3
controlPoint2:c4];
    [ovalPath addCurveToPoint:pointD controlPoint1:c5
controlPoint2:c6];
    [ovalPath addCurveToPoint:pointA controlPoint1:c7
controlPoint2:c8];
    [ovalPath closePath];
```

```
    //外接虚线矩形
     UIBezierPath *rectPath = [UIBezierPath
bezierPathWithRect:self.outsideRect];
     CGContextAddPath(ctx, rectPath.CGPath);
     CGContextSetStrokeColorWithColor(ctx, [UIColor
blackColor].CGColor);
     CGContextSetLineWidth(ctx, 1.0);
     CGFloat dash[] = {5.0, 5.0};
     CGContextSetLineDash(ctx, 0.0, dash, 2); //1

     CGContextStrokePath(ctx); //给线条填充颜色
     CGContextAddPath(ctx, ovalPath.CGPath);
     CGContextSetStrokeColorWithColor(ctx, [UIColor
blackColor].CGColor);
     CGContextSetFillColorWithColor(ctx, [UIColor
redColor].CGColor);
     CGContextSetLineDash(ctx, 0, NULL, 0); //2
     CGContextDrawPath(ctx, kCGPathFillStroke); //同时给线条和线条
包围的内部区域填充颜色
```

```
     //-----   注：以下所有代码全是为了辅助观察   -----


     //标记出每个点并连线，方便观察，给所有关键点染色 -- 白色,辅助线颜色
-- 白色
     
     CGContextSetFillColorWithColor(ctx, [UIColor
yellowColor].CGColor);
     CGContextSetStrokeColorWithColor(ctx, [UIColor
blackColor].CGColor);
     NSArray *points = @[[NSValue
valueWithCGPoint:pointA],[NSValue
valueWithCGPoint:pointB],[NSValue
valueWithCGPoint:pointC],[NSValue
valueWithCGPoint:pointD],[NSValue valueWithCGPoint:c1],[NSValue
valueWithCGPoint:c2],[NSValue valueWithCGPoint:c3],[NSValue
valueWithCGPoint:c4],[NSValue valueWithCGPoint:c5],[NSValue
valueWithCGPoint:c6],[NSValue valueWithCGPoint:c7],[NSValue
valueWithCGPoint:c8]];
```

```
    [self drawPoint:points withContext:ctx];

    //连接辅助线
    UIBezierPath *helperline = [UIBezierPath bezierPath];
    [helperline moveToPoint:pointA];
    [helperline addLineToPoint:c1];
    [helperline addLineToPoint:c2];
    [helperline addLineToPoint:pointB];
    [helperline addLineToPoint:c3];
    [helperline addLineToPoint:c4];
    [helperline addLineToPoint:pointC];
    [helperline addLineToPoint:c5];
    [helperline addLineToPoint:c6];
    [helperline addLineToPoint:pointD];
    [helperline addLineToPoint:c7];
    [helperline addLineToPoint:c8];
    [helperline closePath];

    CGContextAddPath(ctx, helperline.CGPath);

    CGFloat dash2[] = {2.0, 2.0};
    CGContextSetLineDash(ctx, 0.0, dash2, 2);
    CGContextStrokePath(ctx); //给辅助线条填充颜色

}

//一个提取出来的画点的方法：在 point 位置画一个点，方便观察运动情况
-(void)drawPoint:(NSArray *)points
withContext:(CGContextRef)ctx{

    for (NSValue *pointValue in points) {
        CGPoint point = [pointValue CGPointValue];
        CGContextFillRect(ctx, CGRectMake(point.x - 2,point.y -
2,4,4));
    }
}
```

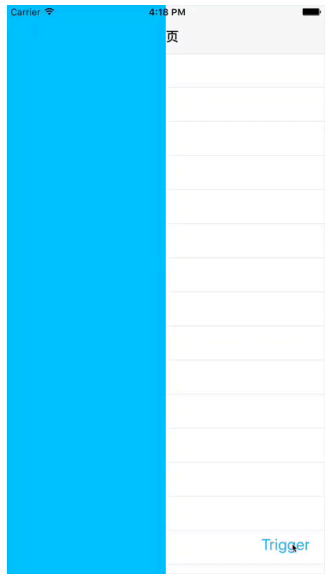
代码中的 (CGContextRef)ctx 字面意思是指上下文，你可以理解为一块全局的画布。也就是说，一旦在某个地方改了画布的一些属性，之

后任何地方使用该画布属性都是改了之后的。比如上面在 //1 中把线条样式改成了虚线，那么在下文 //2 中如果不恢复成连续的直线，那么画出来的依然是虚线样式。

第一个 demo 到这里就告一个段落了。你可以去我的 Github 主页找到 A-GUIDE-TO-iOS-ANIMATION 这个 repo，里面有本册电子书所有 demo 的源码可供学习，并且提供了 Swift 版本。本节的源码参见 AnimatedCircleDemo。当然我还是强烈建议你亲自动手写一遍。以我局限的自学经历来说，很多东西看的时候以为都会了，但只有真的上手才会暴露好多知识漏洞。如果你现在想接着看下一个 demo 的话，你可以继续。但我还是建议你别急。

这一回我们来拆解 GooeySlideMenu 。效果如 Movie 2.2 。授人与鱼不如授人与渔。在介绍这个动画之前，我觉得我有必要向你分享一下我做动画的一般性原则。因为如果你不知道这个通用的方法论，很可能你学会了我教的这个，但下次你看到其它动画的时侯就不会分析了，那这就是我对你的不负责。所以，我必须告诉你我做动画时的心得。其中一条就是：「善于

Movie 2.2 分镜头2 —— 实现蓝色视图边界的反弹动画



拆解」。即把一个复杂的动画分解为几个分动画，然后再把这些分动画逐一解决。

这是我在看过大量动画并亲手实践之后发现的一条规律。然而有一天，我突然意识到，这条规律不仅仅适用于做动画，这其实是一条普世价值观。任何宏观上复杂的事情，都可以通过瓦解的方式细分成一个个小的 ***Task***，然后各个击破。咦，这不就是当年老毛打江山的战术吗？

题外话，其实我练习指弹吉他的过程也深刻悟到了这一点。指弹吉他的难度是比普通弹唱大很多的，因为仅仅凭借两只手要同时担任不同乐器的工作。往往乍一看一首指弹吉他谱子就直接想放弃了。但通过分解的方式就可以很好地解决这个问题。分解乐谱之后，我不再需要考虑整篇还有多长才结束，只要就把属于今天的 ***Task*** 完成就行了，而每天的 ***Task*** 又不是很难，所以每天都会很有成就感，自然就能顺水推舟地进行下去，直到完成既定目标。这又让我联想到了番茄工作法，把一个任务分成不同的小番茄，每个番茄各个击破，直到完成整个任务。这么看

来，很多 solution 本质上的理念都是相同的嘛，莫非这就是传说中的真理？

抱歉，我跑题了。下面回到主题。

我们先把问题拆解成一个淡蓝色的 View 从屏幕左侧移入，这一步成功了，我们再去考虑实现边界的弯曲。为了保障不被其他视图遮挡，我决定把它加在 UIWindow 上。就像 Movie 2.3 这样：

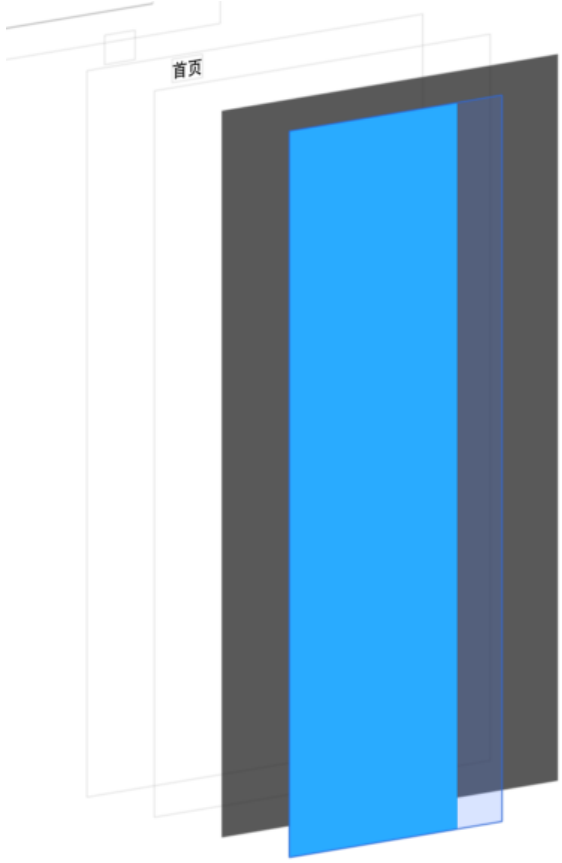
Movie 2.3 GooeySlideMenu


```
self.frame =
CGRectMake(-keyWindow.frame.size.width/2-EXTRAAREA, 0,
keyWindow.frame.size.width/2+EXTRAAREA,
keyWindow.frame.size.height);

self.backgroundColor = [UIColor clearColor];

[keyWindow insertSubview:self belowSubview:helperSideView];
```

你可能会疑问这里的 EXTRAAREA 是什么？你可以看一下 Image 2.1。注意到，淡蓝色的 SlideMenu 其实并没有全部被蓝色填充满，右侧还留出了 30px（即代码中的 EXTRAAREA ） 的透明区域。理由很简单，因为如果不这么做，发生弹性时向右突出的边界就看不到了。至于动画就非常简单了，由于 origin 的末状态为 (0,0)，所以可以直接用 bounds 。

```
[UIView animateWithDuration:0.3 animations:^{
    self.frame = self.bounds;
}];
```

IMAGE 2.1 解释为什么需要留出间隙



拆解的第一步我们已经完成。下面我们继续完成拆解的第二步 —— 实现边界的反弹。效果参见 MOVIE 2.4。

这里就涉及到了两个技术点 —— CADisplayLink 以及贝塞尔曲线。贝塞尔曲线已经在前面有所介绍，这里介绍一下 CADisplayLink。

简单地理解，CADisplayLink 就是一个定时器，每隔 1/60 秒(16.66667ms)刷新一次屏幕。使用的时候，我们要把它添加到一个 runloop 中，并给它绑定一个 target 和 selector，才能在屏幕以 1/60 秒刷新的时候实时调用绑定的方法。

MOVIE 2.4 分镜头1 —— 蓝色视图从左侧移入

例如：

```
self.displayLink = [CADisplayLink displayLinkWithTarget:self
selector:@selector(displayLinkAction:)];

[self.displayLink addToRunLoop:[NSRunLoop mainRunLoop]
forMode:NSDefaultRunLoopMode];
```

另外，CADisplayLink 中还有一些其它属性。

```
@property(readonly, nonatomic) CFTimeInterval duration;
```
每帧之间的时间

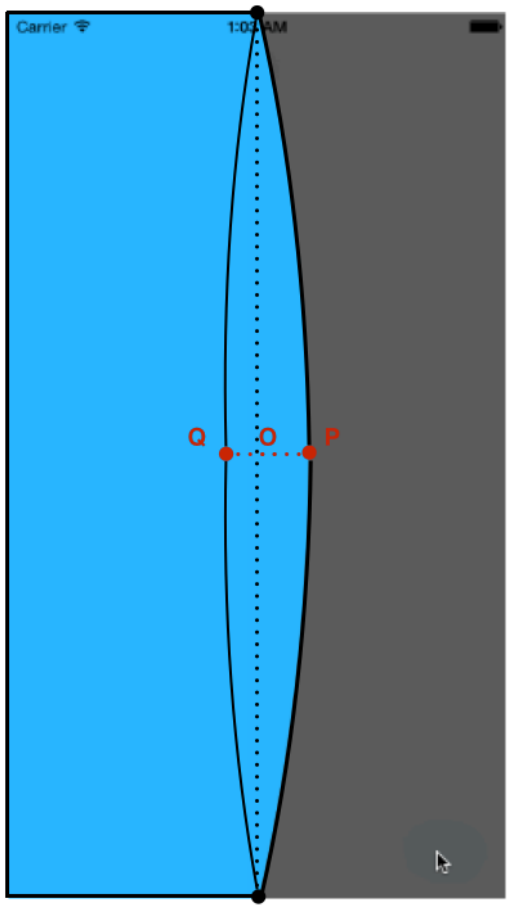
```
@property(nonatomic) NSInteger frameInterval;
```
间隔多少帧调用一次 selector 方法，默认值是 1，即每帧都调用一次。如果每帧都调用一次的话，对于 iOS 设备来说那刷新频率就是 60HZ 也就是每秒 60 次，如果将 frameInterval 设为 2 那么就会两帧调用一次，也就是变成了每秒刷新 30 次。

```
@property(getter=isPaused, nonatomic) BOOL paused;
```
是否暂停当前的定时器，控制 CADisplayLink 的运行。

当我们想结束一个 CADisplayLink 的时候，应该调用：

```
-(void)invalidate;
```
从而从 runloop 中彻底删除之前绑定的 target 跟 selector。

由于 CADisplayLink 绑定的方法会在每次屏幕刷新时被调用，精确度相当之高。正是基于这个特点，CADisplayLink 非常适合 UI 的重绘。

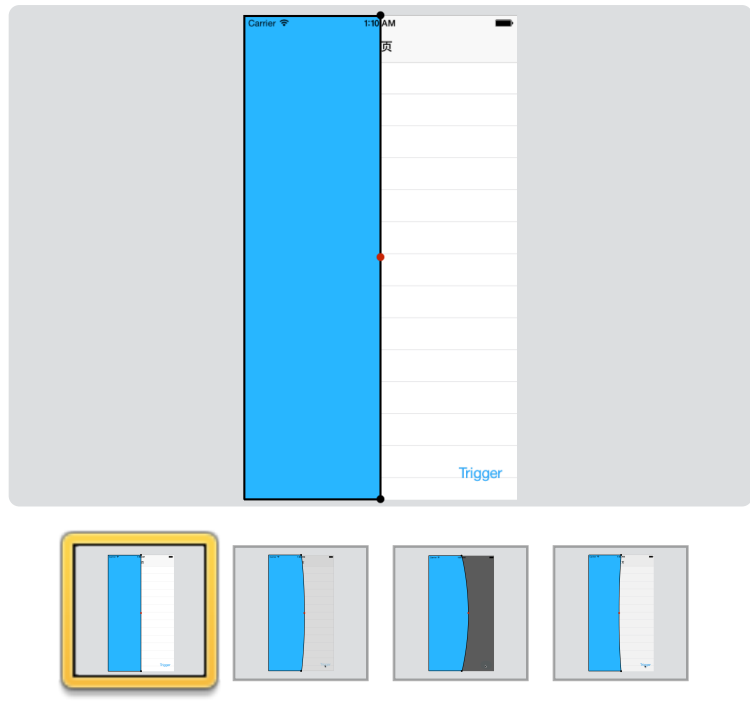


下面又到了算坐标点画贝塞尔曲线的时间了。我们的目标是在 CADisplayLink 绑定的方法 -(void)displayLinkAction:(CADisplayLink *)dis 中实现重绘。这次的计算难度比之前 粘性小球 的要小很多，因为这里我们只需要一条贝塞尔曲线，而且只需要一个控制点。

一图胜千言。左图中红色的点表示控制点，黑色的两个点表示唯一一条贝塞尔曲线的两个端点，其余线段均用直线连接。

接下来的问题就是，如何控制红点的运动？ GALLERY 2.1 很好地说明了红点色运动轨迹。如下：当 Menu 弹出时，那么红点从先 O 点运动到 P 点，再从 P 点运动到 Q 点，最后从 Q 点运动到 O 点；反之，当 Menu 隐藏时，O -> Q -> P -> O.

GALLERY 2.1 控制点的运动轨迹演示



那么问题就化归到了数学问题：

“如何产生一组变化的数值，*O* 增加到某个正数，再从这个正数（也就是最大值 *P* 点）递减到一个负数，最后从这个负数（也就是最小值 *Q* 点）递增到 *O* ?”

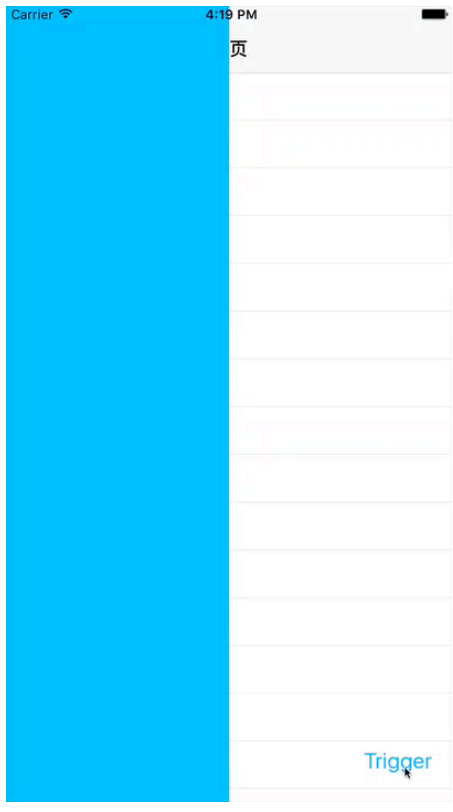
这里介绍两个思路：

- Layer 自定义 Property 的动画。
- 辅助视图。

因为我们的目标很明确，就是需要产生一组变化的数值，这个数值满足上面所述的先递增再递减的规律，至于这个实现的过程就可以是上面提到的两种方案。考虑到方案一我会在之后单独拿出一章来介绍，所以这里我先介绍辅助视图这个技巧。说起这个思路，还得多亏我看到了这篇博客：Recreating Skype's Action Sheet Animation。

效果请看 Movie 2.5。

Movie 2.5 利用两个辅助视图创建弹性动画

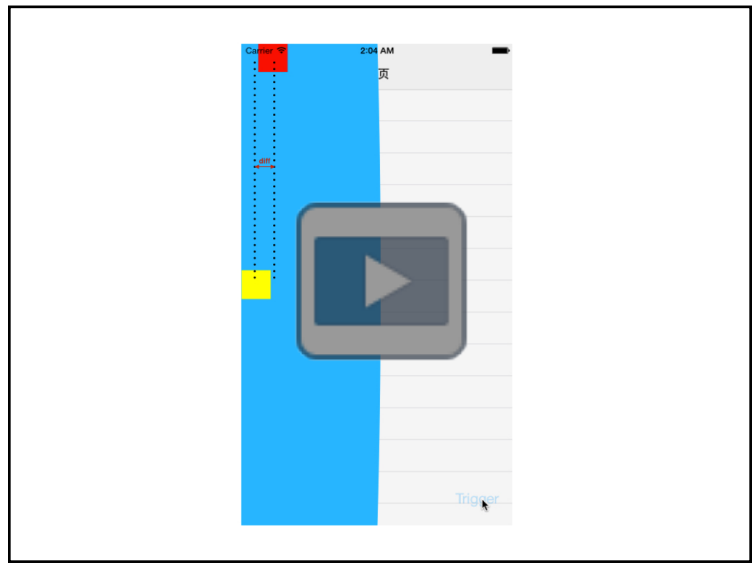


没错，我们创建了两个辅助视图，设置起点和终点都一样，利用弹性动画天生的回弹特性，我们只要赋予两个

辅助视图以不同的动画参数，并且实时计算出两个辅助视图的横坐标 X 之差，就可以间接地得到一组从 0 增至一个正数后，递减至一个负数，最后再回到 0 的数据。

你也可以点击右侧的 Keynote 2.3 明白我的意思，即：

Keynote 2.3 慢动作解释两个辅助视图的作用



『创建两个辅助视图，设置起点和终点都一样，利用弹性动画自身的震荡特点，赋予两个辅助视图以不同的动画参数，实时计算两个辅助视图的横坐标 X 之差，就可以间接地得到一组从 0 增加到一个正数后，递减到一个负数，最后再回到 0 的数据。』

好了，是时候写代码了。

```
//把操作封装到一个公开的方法中
-(void)trigger{

    if (!triggered) {

        [keyWindow insertSubview:blurView belowSubview:self];

        [UIView animateWithDuration:0.3 animations:^{

            self.frame = self.bounds;
```

```
        }];

        [self beforeAnimation];
        [UIView animateWithDuration:0.7 delay:0.0f
usingSpringWithDamping:0.5f initialSpringVelocity:0.9f
options:UIViewAnimationOptionBeginFromCurrentState | UIViewAni-
mationOptionAllowUserInteraction animations:^{

            helperSideView.center =
CGPointMake(keyWindow.center.x,
helperSideView.frame.size.height/2);

        } completion:^(BOOL finished) {
            [self finishAnimation];
        }];

        [UIView animateWithDuration:0.3 animations:^{
            blurView.alpha = 1.0f;

        }];
```

```
        [self beforeAnimation];

        [UIView animateWithDuration:0.7 delay:0.0f
usingSpringWithDamping:0.8f initialSpringVelocity:2.0f
options:UIViewAnimationOptionBeginFromCurrentState | UIViewAni-
mationOptionAllowUserInteraction animations:^{

            helperCenterView.center = keyWindow.center;

        } completion:^(BOOL finished) {
            if (finished) {
                UITapGestureRecognizer *tapGes = [[UITapGesture-
Recognizer alloc]initWithTarget:self
action:@selector(tapToUntrigger:)];
                [blurView addGestureRecognizer:tapGes];
                [self finishAnimation];
            }
        }];

        [self animateButtons];
```

```
        triggered = YES;

    }else{
        [self tapToUntrigger:nil];
    }

}
```

[self beforeAnimation] 和 [self finishAnimation] 用于控制 CA-DisplayLink 什么时候应该移除。用到了动画的累加计数：每开始一个动画时计数器加 1，每停止一个动画时计数器减 1，当两个动画都完成时，计数器为 0，此时移除 CADisplayLink。

```
//动画之前调用
-(void)beforeAnimation{

    if (self.displayLink == nil) {

        self.displayLink = [CADisplayLink
displayLinkWithTarget:self
selector:@selector(displayLinkAction:)];

        [self.displayLink addToRunLoop:[NSRunLoop mainRunLoop]
forMode:NSDefaultRunLoopMode];
    }
    self.animationCount ++;
}

//动画完成之后调用
-(void)finishAnimation{

    self.animationCount --;
    if (self.animationCount == 0) {

        [self.displayLink invalidate];
        self.displayLink = nil;
```

```
    }
}
```

以上几处代码只是实现了辅助视图的运动以及 Menu 侧滑的运动，接下来的代码才是重头戏，因为我们要尝试让边界发生回弹。

```
//CADisplayLink 绑定的方法
-(void)displayLinkAction:(CADisplayLink *)dis{

    CALayer *sideHelperPresentationLayer   =  (CALayer
*)[helperSideView.layer presentationLayer];
    CALayer *centerHelperPresentationLayer =  (CALayer
*)[helperCenterView.layer presentationLayer];

    CGRect centerRect = [[centerHelperPresentationLayer
valueForKeyPath:@"frame"]CGRectValue];
    CGRect sideRect = [[sideHelperPresentationLayer
valueForKeyPath:@"frame"]CGRectValue];

    diff = sideRect.origin.x - centerRect.origin.x;

    [self setNeedsDisplay];

}

//调用 [self setNeedsDisplay] 时触发
- (void)drawRect:(CGRect)rect {

    UIBezierPath *path = [UIBezierPath bezierPath];
    [path moveToPoint:CGPointMake(0, 0)];
    [path
addLineToPoint:CGPointMake(self.frame.size.width-EXTRAAREA,
0)];

    [path
addQuadCurveToPoint:CGPointMake(self.frame.size.width-EXTRAAREA
, self.frame.size.height)
```

```
controlPoint:CGPointMake(keyWindow.frame.size.width/2+diff,
keyWindow.frame.size.height/2)];

    [path addLineToPoint:CGPointMake(0,
self.frame.size.height)];
    [path closePath];

    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextAddPath(context, path.CGPath);
    [_menuColor set];
    CGContextFillPath(context);
}
```

我在后面提到了 Presentation Layer 的作用 —— 即可以实时获取 Layer 属性的当前值。而我们的这个 demo 中要获取的正是这两个辅助视图实时的 X 坐标，从而才能计算出 diff。有了这个 diff，我们再调用 [self setNeedsDisplay]; （这个方法会触发 UIView 的 drawRect 或 CALayer 的 drawRectInContext），同时在 -(void)drawRect:(CGRect)rect 中绘制边界。曲线用贝塞尔曲线，其余都是直线，用 addLineToPoint: 就可以解决。唯一需要注意的是控制点的 X 坐标需要加上 diff:

```
CGPointMake(keyWindow.frame.size.width/2+diff,
keyWindow.frame.size.height/2)
```

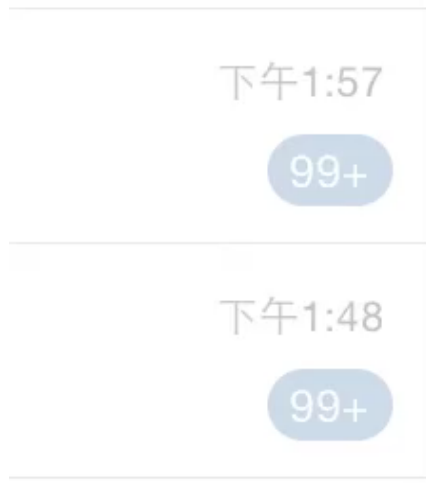
至此，这个 demo 到这里也结束了，剩下的就需要靠你动手实践了。亲身体验和看一遍的效果是完全不同的。源码我已经上传到了

Github，你可以去 GooeySlideMenuDemo 看看。当然，再次提醒，Swift 也一并提供了。

这一节我们学习了使用 CADisplayLink 以及复习了 Presentation Layer 的功能，还复习了绘制贝塞尔曲线知识。

第三小节，我们来尝试复刻一下手机 QQ 里那个起泡拖拽的交互。见 Movie 2.6 。

Movie 2.6 手机 QQ 起泡拖拽交互



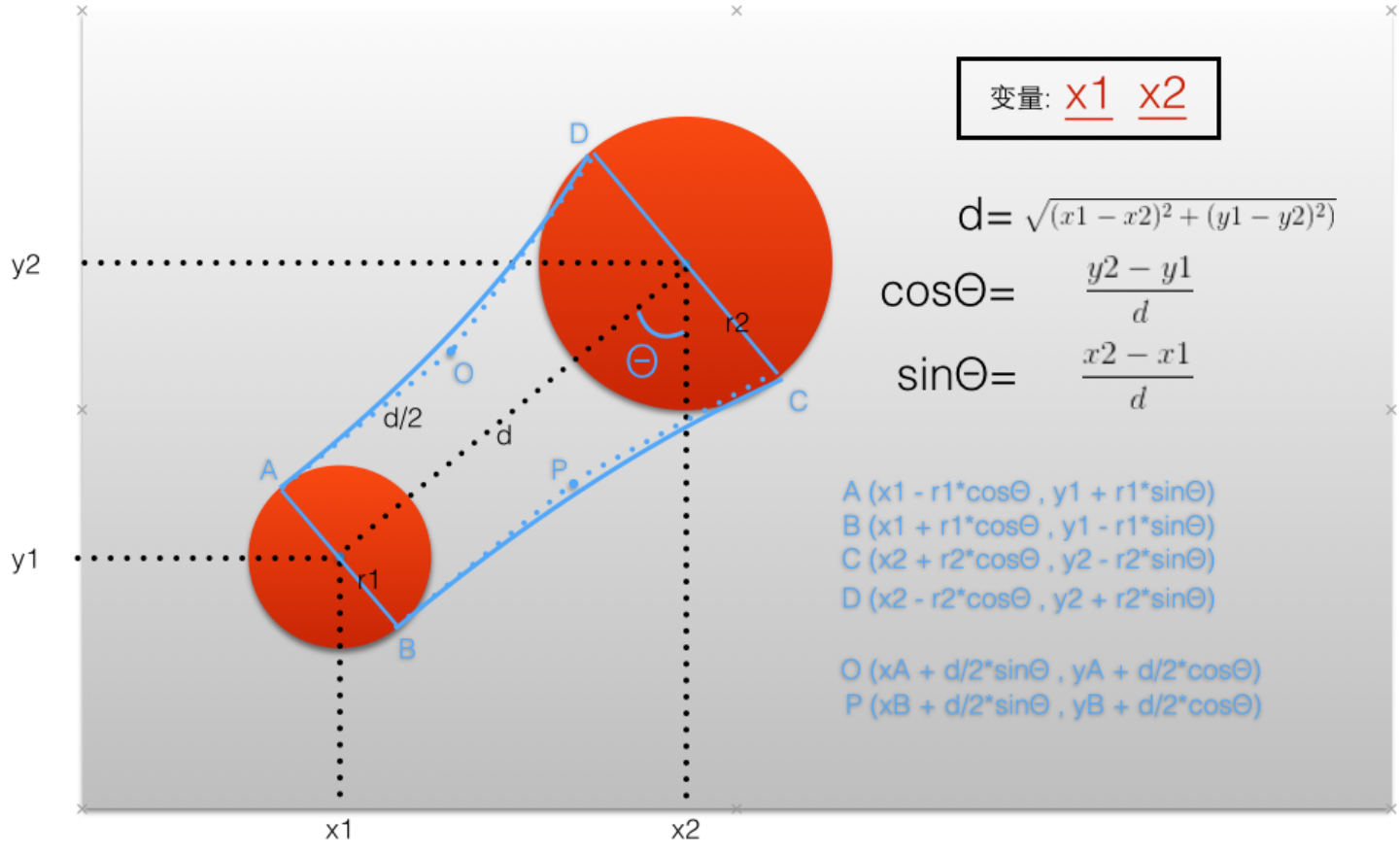
思路如下：

★ 计算出粘性边界关键点的通用坐标。

★ 根据算出的坐标通式，用贝塞尔曲线画出粘性部分的边界，在手势的 UIPanGestureStateChanged 方法里实时重绘。

这个交互中，最核心的要数计算关键点的通用坐标了。一图胜千言。下面我绘制了一幅分析图，方便你计算坐标位置，其中 OA ⊥ AB，PB

⊥ AB，且 OA=PB=d/2。这样一来，问题就转化成了一个高中数学求点坐标的题目了。



我已经在图中计算出六个关键点 A,B,C,D,E,F,G 的坐标位置了。接下来就是要把数学表达式转化成代码了。首先我们创建一些需要的变量：

```
    CGFloat r1;
    CGFloat r2;
    CGFloat x1;
    CGFloat y1;
    CGFloat x2;
    CGFloat y2;
    CGFloat centerDistance;
    CGFloat cosDigree;
    CGFloat sinDigree;
    CGPoint pointA; //A
    CGPoint pointB; //B
    CGPoint pointD; //D
    CGPoint pointC; //C
    CGPoint pointO; //O
    CGPoint pointP; //P
```

根据上图中的计算公式，我们不难用代码表示出这几个变量：

```
    x1 = backView.center.x;
    y1 = backView.center.y;
    x2 = self.frontView.center.x;
    y2 = self.frontView.center.y;

    centerDistance = sqrtf((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
    if (centerDistance == 0) {
        cosDigree = 1;
        sinDigree = 0;
    }else{
        cosDigree = (y2-y1)/centerDistance;
        sinDigree = (x2-x1)/centerDistance;
    }

    r1 = oldBackViewFrame.size.width / 2 - centerDistance/self.viscosity;

    pointA = CGPointMake(x1-r1*cosDigree, y1+r1*sinDigree);  // A
    pointB = CGPointMake(x1+r1*cosDigree, y1-r1*sinDigree); // B
    pointD = CGPointMake(x2-r2*cosDigree, y2+r2*sinDigree); // D
    pointC = CGPointMake(x2+r2*cosDigree, y2-r2*sinDigree);// C
    pointO = CGPointMake(pointA.x + (centerDistance / 2)*sinDigree, pointA.y
+ (centerDistance / 2)*cosDigree);
    pointP = CGPointMake(pointB.x + (centerDistance / 2)*sinDigree, pointB.y
+ (centerDistance / 2)*cosDigree);
```

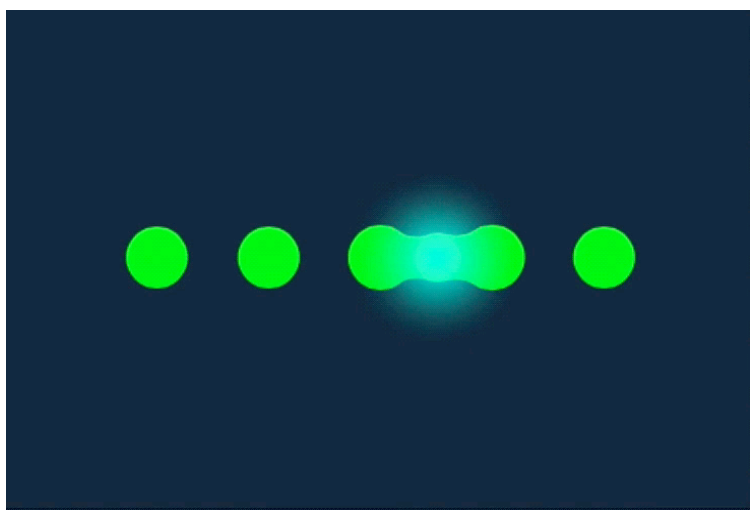
然后，用贝塞尔曲线把这些点连起来，就有了粘性小球完整绘制。最后，我们要做的就是在小球绑定的手势的 Change 方法里面调用这个绘制函数，已达到小球的形状随着手指的移动而变化，就有了粘性的交互了。

```
if (ges.state == UIGestureRecognizerStateChanged){
    ...
    // 绘制函数
    [self drawRect];
}
```

原理其实十分简单，代码已经上传至 **Github**，支持 **Objective-C** 和 **Swift** 。

这一小节，让我们来膜拜一下 *yoavlt* 的　项目。这个项目是一个「圆球」效果的 **loading** 视图，　我也提交了 pr 。同样原理另一个项目是 **LiquidFloatingActionButton**，是一个「圆球」效果的菜单。我们来看一下这两个效果。

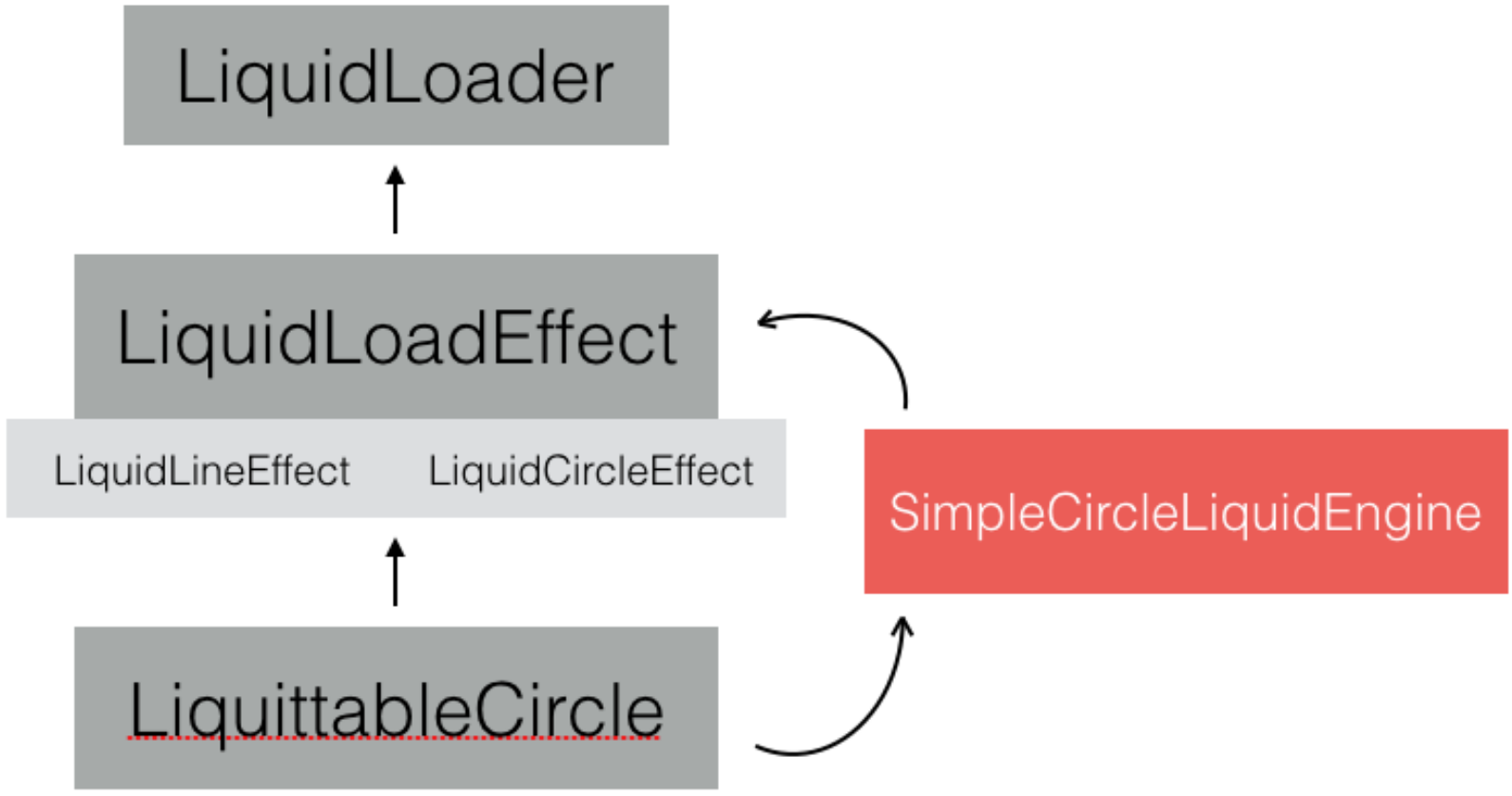
Movie 2.7 LiquidLoader



Movie 2.8 LiquidFloatingActionButton

下面我们来看一下 LiquidLoader 是怎么实现的。

首先我们来看一看架构。



稍微解释一下。对外使用的类就是 LiquidLoader，其中用到了 LiquidLoadEffect 这个类，这个类是个基类，有两个子类继承于它 —— LiquidLineEffect 和 LiquidCircleEffect，用来实现两种样式。LiquidLoadEffect 这个类中用到了 LiquittableCircle，这个类用来绘制小球；除此之外，LiquidLoadEffect 中还用到了一个「引擎」—— SimpleCircleLiquidEngine，这个引擎的作用就是判断当前静止的圆球与运动的圆球之间的距离，从而决定是否显示粘连的部分。

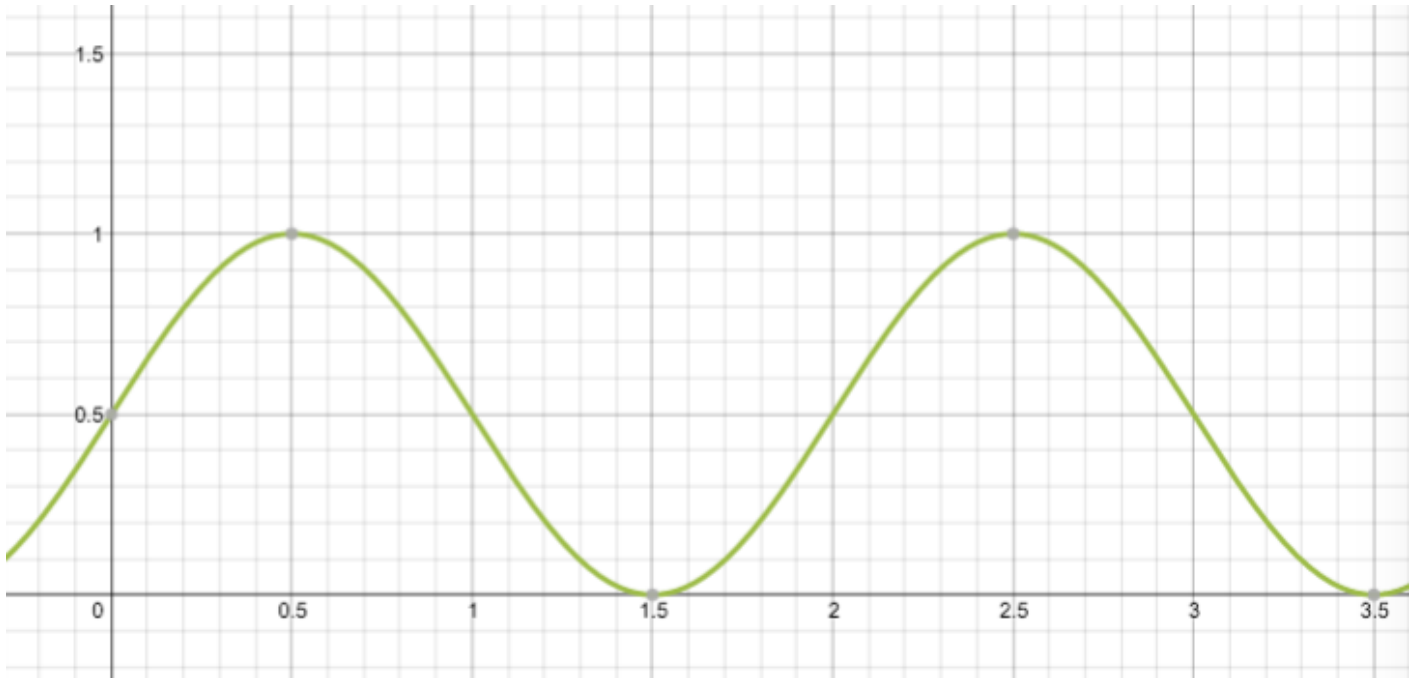
具体到代码中。首先，国际惯例，创建一个定时器 `CADisplayLink`，绑定到一个方法：`update()`。这个方法中我们去更新一个动态变量，取名为 `key`，根据 `key` 我们来移动小球的位置，同时用「引擎」来判断

移动的小球和静止的小球之间的距离。以上就是主干线。下面我挖几处细节和你讲讲。

首先，既然小球的位置是通过实时改写 `center` 实现的，那么我们怎么指定动画的曲线函数呢？比如 **Movie 2.7** 中，仔细看你会发现小球在两端有个减速的过程，中间段加速。要实现这样的曲线函数，我们就要依赖三角函数了。

```
func sineTransform(key: CGFloat) -> CGFloat {
    return sin(key * CGFloat(M_PI)) * 0.5 + 0.5
}
```

通过这个网站，我们可以得到如下函数图像：



根据高中物理知识我们知道，加速度看斜率，所以图中 `key` 取值 1 和 2 的时候加速度最大，此时小球正在经过在中点位置。

我们再来看另一个细节：

```
circles.each { circle in
    if self.moveCircle != nil {
        self.engine?.push(self.moveCircle!, other: circle)
    }
}
```

这段代码也是在 update() 方法里，通过遍历所有静止小球，让每个静止的小球与运动的小球进行比较，比较两球圆心之间的距离是否超过某个事先规定好的阈值。

```
func isConnected(circle: LiquittableCircle, other: Liquittable-
Circle) -> Bool {
    let distance = circle.center.minus(other.center).length()
    return distance - circle.radius - other.radius < radius-
Thresh
}
```

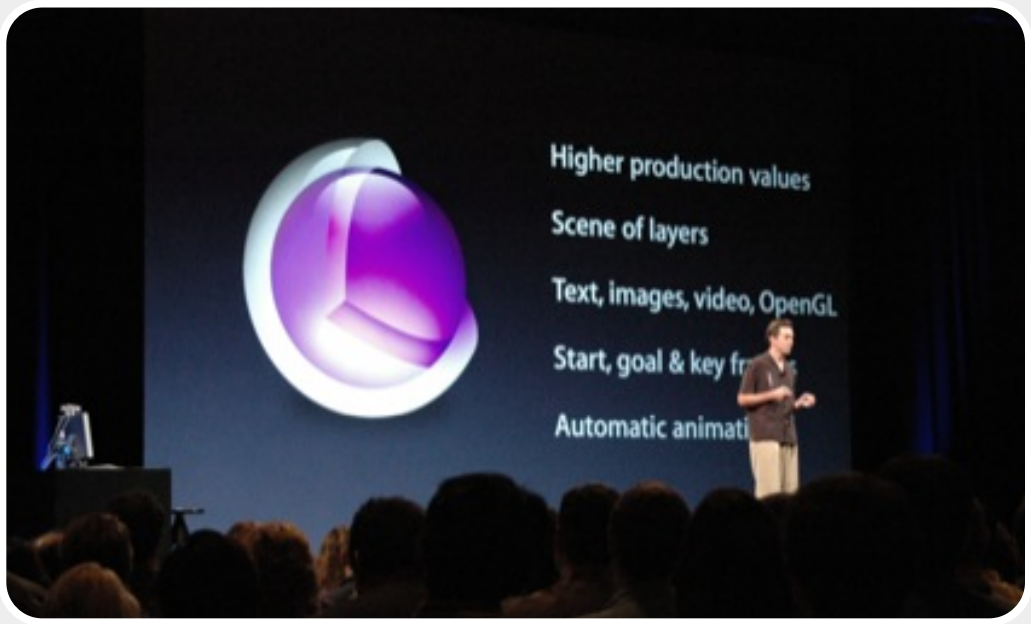
原先的项目中并没有设置 duration 的接口，我在提交的 pr 中增加了这个功能。核心代码如下：

```
override func update() {
    switch key {
    case 0.0...2.0:
        key += 2.0/(duration*60)
    default:
        key = 0.0
    }
}
```

只要让动态变量 key 每次增加的值等于 2.0/总帧数即可。之所以是 2.0 是由三角函数决定的，因为上面的三角函数一个周期为 2 。

除了上面这些之外，更令我大开眼界的是作者写的一大批工具类。身体力行地诠释了什么叫「工欲善其事必先利其器」。相信你看过源码也会有和我一样的感触。

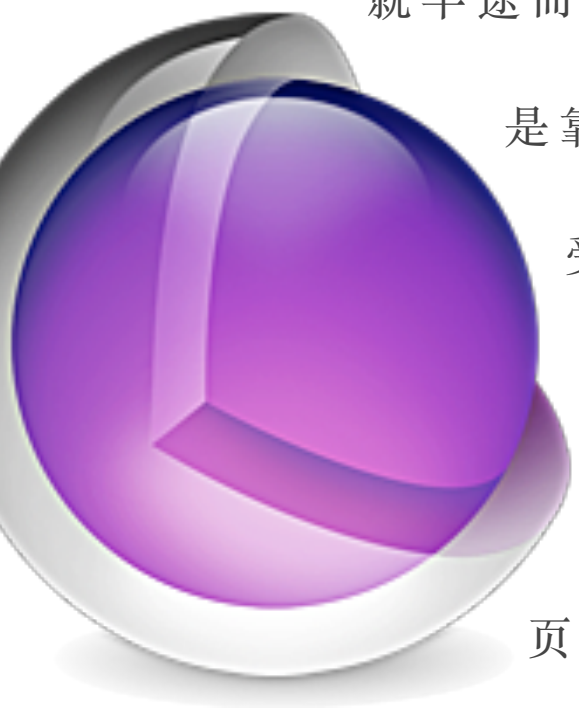
好啦，这个项目的分析就到这里就结束了，你可以在这里翻看源码。



3

CORE ANIMATION

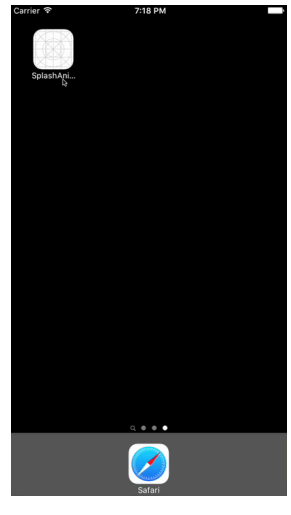
这一章节，我将会把 Core Animation 技巧做个总结。当然，你不用担心我会像学校的老师一样很枯燥地向你灌输知识点，这是别人的做法。事实上从我局限的切身感受来说，我一向认为结果导向的学习方法才是最有效率的：「你要去什么地方，就直接去你想去的那个地方」。所以贯穿我整册电子书的教学模式都是：「我精心挑选一些涵盖尽可能多知识点的例子，让你直接接触这些生动的(半)成品，期间涉及到了知识点，我再发散出去详细介绍」。通过这些活生生的案例让你在实战中了解动画的相关知识。或许你会想，但是这样不够系统性啊，我担心自己只是学到了冰山一角。我想说的是，朋友，我还是那句话，「等你都准备好了，或许你就没有动力了」。不用那么在意这些形式上的东西，你想想你多少次心血来潮地准备好了你认为完备的一切，结果没坚持几天就半途而废了。正所谓「星星之火，可以燎原」。知识从来不是靠看来的，都是靠找来的。我自身就是这种学习模式的受益者。通过这种结果导向的方式学到的每一个知识点就像一颗颗散落的珍珠，随着你越捡越多，当你有一天捡得差不多了，这时你再回过头去看那些当时看几页就犯困的理论书，你会发现一切看起来都是那么顺利，顺便也从全局范畴上对之前零散的知识点进行了一次结构性的巩固。然



后你就会惊喜地发现，所有知识点就像珍珠被串上了线一样，一切都联系起来了。

本章第一个例子，我们来看看如何实现类似 Twitter 的启动动画。效果请参见 **Movie 3.1**。说实话，当时我第一次打开 Twitter 看到这个效果的时候，*I was totally blown away*😱

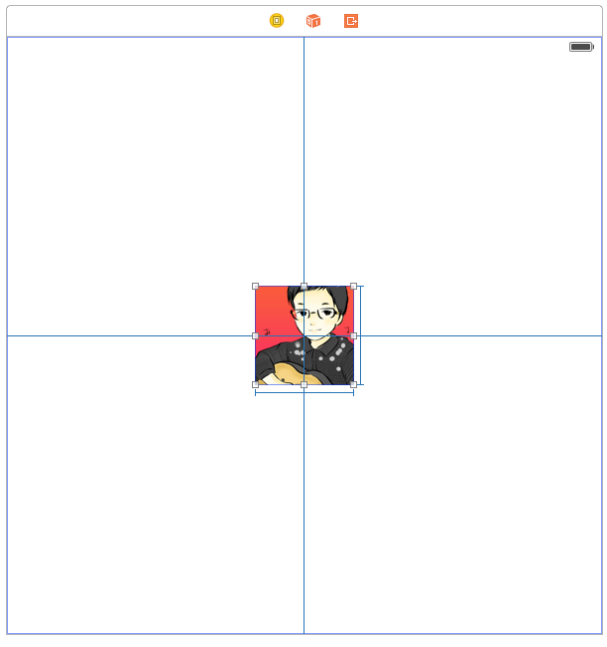
Movie 3.1 模拟 Twitter 的启动动画



在开始之前，我们先了解一下 Layer 的 mask 属性。

```
@property(strong) CALayer *mask;
```

Image 3.1 大小为 100*100 的 photoImage



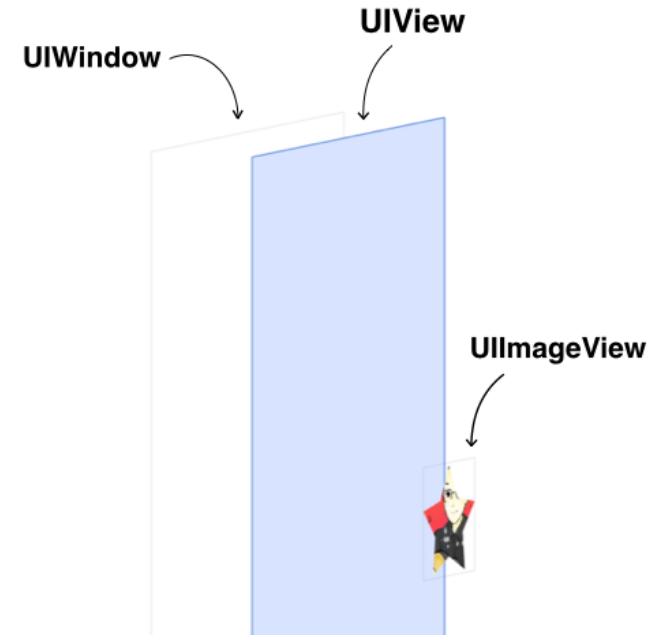
可以发现 mask 也是一个 CALayer。所以当我们使用时，就需要单独创建一个 CALayer 作为 mask。比如我提前准备好了需要遮罩的图片视图 photoImage —— **Image 3.1**。大小为 100*100 的 UIImageView，连好 IBOutlet。在 ViewController.m 的

-(void)viewDidLoad 方法中创建 mask，赋值给 photoImage 的 mask 属性。

```
CALayer *maskLayer = [CALayer layer];
maskLayer.contents = (id)[UIImage imageNamed:@"logo"]
.CGImage; //必须是CGImageRef

maskLayer.frame = CGRectMake(0, 0,
_photoImage.frame.size.width, _photoImage.frame.size.height);

_photoImage.layer.mask = maskLayer;
```

运行之后，可以看到 photoImage 已经被遮罩了。

通过 View Hierarchy 可以看到视图的层级关系。为了实现开头那个 zoom out 的欢迎动画，我把我的的思路用 Gallery 3.1 做了展示。



解释一下，首先毫无疑问一开始显示 LaunchScreen 。接下来 LaunchScreen 消失了，即将进入 NavigationController 时，我们在 -(void)viewDidLoad 中为其设置遮罩，并且把遮罩的位置约束在和 LaunchScreen 中星星同一个位置，这样看起来就好像星星一直停在原地。不过虽然星星的位置不动了，但是 LaunchScreen 一旦结束之后就会露出一个黑底(UIWindow)，立刻就露馅。解决方法也是

简单粗暴，只要把 UIWindow 的背景色改成和 LaunchScreen 一样的颜色就行了。

下面展示代码。首先在 func application didFinishLaunchingWithOptions 中设置 window 的颜色：

```
self.window!.backgroundColor = UIColor(red: 128/255, green: 0/255, blue: 0/255, alpha: 1)
```

然后在 -(void)viewDidLoad 中写上如下代码，让 mask 和 navigationController.view 在不同时间做放大的动画：

```
    // logo mask
    self.navigationController!.view.layer.mask = CALayer()
    self.navigationController!.view.layer.mask.contents = UIImage(named: "logo.png")!.CGImage
    self.navigationController!.view.layer.mask.position = self.view.center
    self.navigationController!.view.layer.mask.bounds = CGRect(x: 0, y: 0, width: 60, height: 60)

    // 接下来就是让 mask 做动画
    let transformAnimation = CAKeyframeAnimation(keyPath: "bounds")
    transformAnimation.delegate = self
    transformAnimation.duration = 1
    transformAnimation.beginTime = CACurrentMediaTime() + 1 //延迟一秒
```

GALLERY 3.1 解释启动动画的视图层级

顺便说一句：这些图都是用 *Keynote* 做的。

```
    let initalBounds = NSValue(CGRect:
self.navigationController!.view.layer.mask.bounds)
    let secondBounds = NSValue(CGRect: CGRect(x: 0, y: 0, width:
50, height: 50))
    let finalBounds = NSValue(CGRect: CGRect(x: 0, y: 0, width:
2000, height: 2000))

    transformAnimation.values = [initalBounds, secondBounds,
finalBounds]
    transformAnimation.keyTimes = [0, 0.5, 1]
      transformAnimation.timingFunctions = [CAMediaTimingFunc-
tion(name: kCAMediaTimingFunctionEaseInEaseOut), CAMediaTiming-
Function(name: kCAMediaTimingFunctionEaseOut)]
      transformAnimation.removedOnCompletion = false
transformAnimation.fillMode = kCAFillModeForwards

self.navigationController!.view.layer.mask.addAnimation(transfo
rmAnimation, forKey: "maskAnimation")


    //最后是 navigationController.view 的动画
    UIView.animateWithDuration(0.25,
        delay: 1.3,
        options: UIViewAnimationOptions.TransitionNone,
        animations: {
            self.navigationController!.view.transform =
    CGAffineTransformMakeScale(1.05, 1.05)
        },
        completion: { finished in
            UIView.animateWithDuration(0.3,
                delay: 0.0,
                options: UIViewAnimationOptions.CurveEaseInOut,
                animations: {
                    self.navigationController!.view.transform =
    CGAffineTransformIdentity
                },
                completion: nil
            )
    })
```

然而，如果你现在运行以上的代码，你会发现仍有不足。因为 mask 一开始就存在，所以你可以在程序启动时就能透过 mask 看到后面的视图。而我们希望的是 mask 后面的内容是在 mask 扩大的过程中逐渐显示出来的，正如一开始效果视频中展示的那样。这里的 solution 就要善于「作弊」，这也是我在做动画的过程中深谙的一个技巧。我们要做的就是在 NavigationController.view 和 mask 之间再加一层背景色为白色的图层，让这个图层挡住背后的内容，同时这个图层在 mask 动画的同时做透明度渐变到 0 的动画。

```
//在 NavigationController.view 和 mask 之间再加一层背景色为白色的图层
var maskBgView = UIView(frame:
self.navigationController!.view.layer.frame)
maskBgView.backgroundColor = UIColor.whiteColor()
self.navigationController!.view.addSubview(maskBgView)
self.navigationController!.view.bringSubviewToFront(maskBgView)


//让这层起遮挡作用视图做一个渐隐的动画
UIView.animateWithDuration(0.1,
    delay: 1.35,
    options: UIViewAnimationOptions.CurveEaseIn,
    animations: {
        maskBgView.alpha = 0.0
    },
    completion: { finished in
        maskBgView.removeFromSuperview()

})
```

下面延伸相关知识。首先，我们就来谈谈 CAKeyframeAnimation 。顾名思义，CAKeyframeAnimation 就相当于 Flash 里的关键帧动画，如果你用过 Flash 制作动画的话你就知道，如果我们要实现一个简单的位置平移、大小缩放、形状变换，我们只需要使用补间动画就可以实现。具体操作就是给出动画的起始状态和结束状态两个关键帧，中间的动画过程只需要设置一个补间即可，剩下的事情软件会自动完成。而这里的起始状态和结束状态的概念，也被沿用到了 CAKeyframeAnimation 里所说的关键帧 。

CAKeyframeAnimation 中我们通过 keyPath 就可以指定动画的类型。比如 let transformAnimation = CAKeyframeAnimation(keyPath: "bounds") 中的 bounds 就是指定了动画类型：让 layer 的 size 发生动画。关于 keyPath 的可选值，你可以查看 CALayer 的 API 文档：

```
/* The bounds of the layer. Defaults to CGRectZero. Animatable.
*/

    /** Geometry and layer hierarchy properties. **/
    var bounds: CGRect

    /* The position in the superlayer that the anchor point of
the layer's
     * bounds rect is aligned to. Defaults to the zero point.
Animatable. */

    var position: CGPoint
```

```
    /* Defines the anchor point of the layer's bounds rect, as
a point in
     * normalized layer coordinates - '(0, 0)' is the bottom
left corner of
     * the bounds rect, '(1, 1)' is the top right corner. De-
faults to
     * '(0.5, 0.5)', i.e. the center of the bounds rect. Animat-
able. */

    var anchorPoint: CGPoint

    /* A transform applied to the layer relative to the anchor
point of its
     * bounds rect. Defaults to the identity transform. Animat-
able. */

    var transform: CATransform3D

   ......
```

类似这一些 Animatable 的属性，都可以作为 CAAnimation 的 keyPath 。然后，我们把每个关键帧的对应参数赋值给 CAKeyframeAnimation 的 values 属性。代码中，我设置了3个关键帧，transformAnimation.values = [initalBounds, secondBounds, finalBounds]。并且设置对应的时间点 transformAnimation.keyTimes = [0, 0.5, 1] ，也就是动画一开始时 bounds 处于 initalBounds 状态；duration 一半时间时处于 secondBounds 状态；动画结束时处于 finalBounds 状态。

但是我们还要注意。当你给一个 CALayer 添加动画的时候，动画其实并没有改变这个 layer 的实际属性。取而代之的，系统会创建一个原始 layer 的拷贝。在文档中，苹果称这个原始 layer 为 Model Layer ，而这个复制的 layer 则被称为 Presentation Layer 。 Presentation Layer 的属性会随着动画的进度实时改变，而 Model Layer 中对应的属性则并不会改变。所以如果你想要获取动画中每个时刻的状态，请使用 layer 的

```
func presentationLayer() -> AnyObject!
```

Hold on a second. 此时如果你只是做了上面的步骤，你会发现效果并不是我们所想的那样。你会发现动画在结束之后突然回到了初始状态。这里就可以引出 removedOnCompletion 和 fillMode 了。

removedOnCompletion 的官方解释是：

```
/* When true, the animation is removed from the render tree once its
 * active duration has passed. Defaults to YES. */
```

也就是默认情况下系统会在 duration 时间后自动移除这个 CAKeyframeAnimation。当 remove 了某个动画，那么系统就会自动销毁这个 layer 的 Presentation Layer ，只留下 Model Layer 。 而前面提到 Model Layer 的属性其实并没有变化，所以也就有了你前面看到的结果，视图在

一瞬间回到了动画的初始状态。要解决这种情况，你需要先把 removedOnCompletion 设置为 false ，然后设置 fillMode 为kCAFillModeForwards 。关于 fillMode ，它有四个值：

- **kCAFillModeRemoved** 这个是默认值，也就是说当动画开始前和动画结束后，动画对layer都没有影响，动画结束后，layer会恢复到之前的状态。
- **kCAFillModeForwards** 当动画结束后，layer会一直保持着动画最后的状态。
- **kCAFillModeBackwards** 这个和 kCAFillModeForwards 是相对的，就是在动画开始前，你只要将动画加入了一个layer，layer便立即进入动画的初始状态并等待动画开始。你可以这样设定测试代码，将一个动画加入一个layer的时候延迟5秒执行。然后就会发现在动画没有开始的时候，只要动画被加入了 layer，layer 便处于动画初始状态， 动画结束后，layer 也会恢复到之前的状态。
- **kCAFillModeBoth** 理解了上面两个，这个就很好理解了，这个其实就是上面两个的合成。动画加入后立即开始，layer便处于动画初始状态，动画结束后layer保持动画最后的状态。

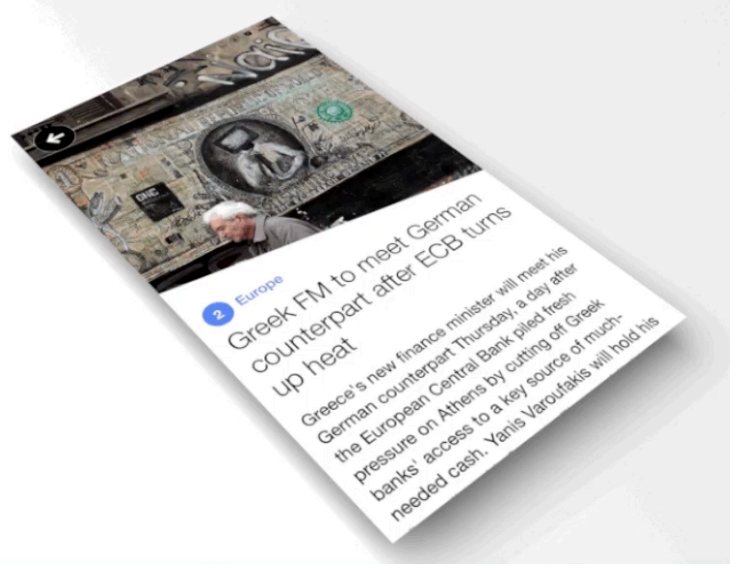
你除了可以设置 removedOnCompletion 为 false, fillMode 为 kCAFillModeForwards 之外，这里还有个 trick ，就是你可以在 addAnimation 之前显式地把 Model Layer 的对应属性设置为结束时的状态，这样同样也能避免之前动画结束后复位的问题。

但设置 removedOnCompletion 和 fillMode 不是正确的方式。正确的做法可以参考 WWDC 2011 中的 session 421 - Core Animation Essentials. 为了保证教程的连贯性，我把视频放在了这章的结尾，你可以在这章结束之后再看这个 session。

除此之外，这个 demo 还有一个比较关键的地方在于需要不断地调 delay 才行，确保每个动画都能衔接上。所以一个好动画离不开一堆好参数。 这一节到这里也就结束了。源码请见 。OC/Swift are both supported.

下一个案例也是关于 mask 的应用，只不过这回套了一个外衣 —— 转场动画，顺便也可以介绍一下它的使用。

先看效果 Movie 3.2。



Movie 3.2 圆圈遮罩的转场动画

尽管核心用的依然是 CoreAnimation ，但是首先我们先来认识下实现转场动画的基本步骤。

iOS7 开始苹果推出了自定义转场的 API 。从此，任何可以用 CoreAnimation 实现的动画，都可以出现在两个 ViewController 的切换之间。并且实现方式高度解耦，这也意味着在保证代码干净的同时想要替换其他动画方案时只需简单改一个类名就可以了，真正体会了一把高颜值代码带来的愉悦感。同时，iOS7 还推出了

手势驱动的转场动画，同样高度解耦。想必随着大屏 iPhone 的普及，软件层面的交互优化将显得格外具有意义。如今的 App 要是还不支持手势滑动返回，就真的太不骄傲了。

苹果在 UINavigationControllerDelegate 和 UIViewController-TransitioningDelegate 中给出了几个协议方法，通过返回类型就可以很清楚地知

道各自的具体作用。你只需要重载它们，然后 return 一个动画的实例对象，一切都搞定了。使用准则就是：UINavigationController push-ViewController 时重载 UINavigationControllerDelegate 的方法；UIViewController presentViewController 时重载 UIViewController-TransitioningDelegate 的方法。

UINavigationControllerDelegate:

```
 - (nullable id <UIViewControllerInteractiveTransitioning>)
navigationController:(UINavigationController *)navigationCon-
troller interactionControllerForAnimationController:(id
<UIViewControllerAnimatedTransitioning>) animationController;

 - (nullable id <UIViewControllerAnimatedTransitioning>)
navigationController:(UINavigationController *)navigationCon-
troller
animationControllerForOperation:(UINavigationControllerOperati
on)operation fromViewController:(UIViewController *)fromVC
toViewController:(UIViewController *)toVC;
```

UIViewControllerTransitioningDelegate:

```
- (nullable id <UIViewControllerAnimatedTransitioning>) anima-
tionControllerForPresentedController: (UIViewController *)pre-
sented presentingController:(UIViewController *)presenting
sourceController: (UIViewController *)source;

- (nullable id <UIViewControllerAnimatedTransitioning>) anima-
tionControllerForDismissedController: (UIViewController *)dis-
missed;

- (nullable id <UIViewControllerInteractiveTransitioning>)
interactionControllerForPresentation:(id
<UIViewControllerAnimatedTransitioning>)animator;

- (nullable id
<UIViewControllerInteractiveTransitioning>)interactionControll
erForDismissal:(id <UIViewControllerAnimatedTransitioning>)
animator;
```

那么接下来就体现了解耦带来的好处了。具体步骤：

- 1、创建继承自 NSObject 并且声明 UIViewControllerAnimatedTran-sitioning 的的动画类。

- 2、重载 UIViewControllerAnimatedTransitioning 中的协议方法。

UIViewControllerAnimatedTransitioning:

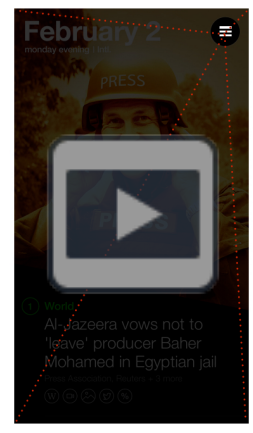
```
@protocol UIViewControllerAnimatedTransitioning <NSObject>
```

```
// This is used for percent driven interactive transitions, as
well as for container controllers that have companion anima-
tions that might need to
// synchronize with the main animation.
- (NSTimeInterval)transitionDuration:(nullable id
<UIViewControllerContextTransitioning>)transitionContext;
// This method can only  be a nop if the transition is interac-
tive and not a percentDriven interactive transition.
- (void)animateTransition:(id
<UIViewControllerContextTransitioning>)transitionContext;


@optional

// This is a convenience and if implemented will be invoked by
the system when the transition context's completeTransition:
method is invoked.
- (void)animationEnded:(BOOL) transitionCompleted;

@end
```

3、没有了。

在准备动手实践之前，我依旧先打算让你过目一下动画的实现思路。请看 Keynote 演示 KEYNOTE 3.1。

KEYNOTE 3.1 演示遮罩动画的思路




```
- (NSTimeInterval)transitionDuration:(id
<UIViewControllerContextTransitioning>)transitionContext{
    return self.duration;
}
```

```
- (void)animateTransition:(id
<UIViewControllerContextTransitioning>)transitionContext{

    
    self.transitionContext = transitionContext;

    ViewController * fromVC = (ViewController *)[transitionCon-
text
viewControllerForKey:UITransitionContextFromViewControllerKey];
    SecondViewController *toVC = (SecondViewController *)[tran-
sitionContext
viewControllerForKey:UITransitionContextToViewControllerKey];
    UIView *contView = [transitionContext containerView];

    UIButton *button = fromVC.button;

    UIBezierPath *maskStartBP =  [UIBezierPath
bezierPathWithOvalInRect:button.frame];
    [contView addSubview:fromVC.view];
    [contView addSubview:toVC.view];


    
    CGPoint finalPoint;
    //判断触发点在那个象限，从而计算出覆盖的最大半径
    if(button.frame.origin.x > (toVC.view.bounds.size.width /
2)){
        if (button.frame.origin.y <
(toVC.view.bounds.size.height / 2)) {
            //第一象限
            finalPoint = CGPointMake(button.center.x - 0,
button.center.y - CGRectGetMaxY(toVC.view.bounds)+30);
        }else{
            //第四象限
            finalPoint = CGPointMake(button.center.x - 0,
button.center.y - 0);
        }
    }else{
        if (button.frame.origin.y <
(toVC.view.bounds.size.height / 2)) {
            //第二象限
```

```
            finalPoint = CGPointMake(button.center.x -
CGRectGetMaxX(toVC.view.bounds), button.center.y -
CGRectGetMaxY(toVC.view.bounds)+30);
        }else{
            //第三象限
            finalPoint = CGPointMake(button.center.x -
CGRectGetMaxX(toVC.view.bounds), button.center.y - 0);
        }
    }

    CGFloat radius = sqrt((finalPoint.x * finalPoint.x) +
(finalPoint.y * finalPoint.y));


    UIBezierPath *maskFinalBP = [UIBezierPath
bezierPathWithOvalInRect:CGRectInset(button.frame, -radius, -ra-
dius)];
    //创建一个 CAShapeLayer 作为 toView 的遮罩。并让遮罩发生 path 属
性的动画。
    CAShapeLayer *maskLayer = [CAShapeLayer layer];
    maskLayer.path = maskFinalBP.CGPath; //将它的 path 指定为最终
的 path 来避免在动画完成后会回弹
    toVC.view.layer.mask = maskLayer;
    CABasicAnimation *maskLayerAnimation = [CABasicAnimation
animationWithKeyPath:@"path"];
    maskLayerAnimation.fromValue = (__bridge
id)(maskStartBP.CGPath);
    maskLayerAnimation.toValue = (__bridge
id)((maskFinalBP.CGPath));
    maskLayerAnimation.duration = [self
transitionDuration:transitionContext];
    maskLayerAnimation.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseInEaseOut];
    maskLayerAnimation.delegate = self;

    [maskLayer addAnimation:maskLayerAnimation forKey:@"path"];

}


#pragma mark - CABasicAnimation的Delegate
```

```
- (void)animationDidStop:(CAAnimation *)anim
finished:(BOOL)flag{

    //告诉 iOS 这个 transition 完成
    [self.transitionContext completeTransition:![self. transi-
tionContext transitionWasCancelled]];
    //清除 fromVC 的 mask
    [self.transitionContext
viewControllerForKey:UITransitionContextFromViewControllerKey].
view.layer.mask = nil;
    [self.transitionContext
viewControllerForKey:UITransitionContextToViewControllerKey].vi
ew.layer.mask = nil;

}
```

到这里这个简单的动画就全部结束了。源码请见 KYPingTransitionDemo。事实上，我们看到的绝大多数动画基本都是这些标准动画的叠加。比如平移、缩放、旋转等，只不过这些效果串在一起看起来就感觉非常复杂了，更何况动画的时间往往比较短。常言道，唯快不破。

顺便，我开源了一个类似的转场动画 Movie 3.3，只需给定一个任意 point，就可以通过圆圈放大的动画进行转场。地址在 A-GUIDE-TO-iOS-ANIMATION 下的 KYBubbleTransition。如果你对其中的类似 气泡晃动效果感兴趣，可以来 KYFloatingBubble 看

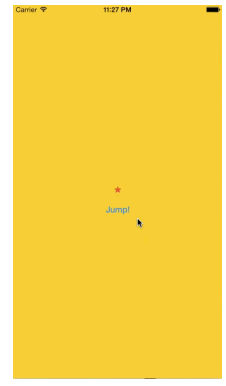
Movie 3.3 气泡放大的转场动画

看，没有任何特殊技巧，完全用的 Core Animation 基本功。

如果让我站在目前局限的角度评价一下 CoreAnimation 的话，我觉得最大感悟就是 ——「大道至简」。我看到过的所有真正牛逼的国外动画师到最后用的并不是什么黑科技或者深奥的算法来做动画，当然也有，但更多的还是用最基础的 CoreAnimation ，通过各种基本动画的组合(duration,delay,timeFunction,damping,velocity...)，以及合理的参数调节，让一款优秀的动画跃然于屏幕之上。所以下面我将尝试给你介绍几个动画，完完全全用的就是 CoreAnimation 中的「基本单位」—— translation、rotation、scale... 打造出乍看起来很「复杂」的动画效果。相信看完这部分，你就可以应付实际应用中绝大多数动画需求了，毕竟日常还是 CoreAnimation 用到的最多。*let's get started!* 

这个例子是一个图片弹跳切换的效果，最早出现在锤子日历中。当你对某一条内容加星之后，就会出现这个活泼的切换动画。效果参见 **Movie 3.4**。

Movie 3.4 图片弹跳+翻转动画



动画分为两个阶段：一个弹上去的阶段，一个落下来的阶段。弹上去的过程让视图绕 y 轴旋转 90°，此时第一阶段的动画结束。在代理方法 animationDidStop 中开始第二个动画——下落。在这个阶段一开始立刻替换图片，随后在落下的同时让视图继续旋转 90°。然后你可能有疑问了，这怎么才转了 180°？那不是动画结束之后图片是反过来的吗？对，所以我们要在下落动画结束之后 removeAllAnimations。

代码也非常短：

```
//上弹动画
-(void)animate{

    ⛴
    if (animating == YES) {
        return;
    }

    animating = YES;

    ⛴
```

```
    CABasicAnimation *transformAnima = [CABasicAnimation
animationWithKeyPath:@"transform.rotation.y"];
    transformAnima.fromValue = @(0);
    transformAnima.toValue = @(M_PI_2);
    transformAnima.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseInEaseOut];

    
    CABasicAnimation *positionAnima = [CABasicAnimation
animationWithKeyPath:@"position.y"];
    positionAnima.fromValue = @(self.starView.center.y);
    positionAnima.toValue = @(self.starView.center.y - 14);
    positionAnima.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseOut];

    
    CAAnimationGroup *animGroup = [CAAnimationGroup animation];
    animGroup.duration = jumpDuration;
    animGroup.fillMode = kCAFillModeForwards;
    animGroup.removedOnCompletion = NO;
    animGroup.delegate = self;
    animGroup.animations = @[transformAnima,positionAnima];

    [self.starView.layer addAnimation:animGroup
forKey:@"jumpUp"];

}


//下落动画
- (void)animationDidStop:(CAAnimation *)anim
finished:(BOOL)flag{

    if ([anim isEqual:[self.starView.layer
animationForKey:@"jumpUp"]]) {

        
        self.state = self.state==Mark?non_Mark:Mark;
        NSLog(@"state:%ld",_state);
        CABasicAnimation *transformAnima = [CABasicAnimation
animationWithKeyPath:@"transform.rotation.y"];
        transformAnima.fromValue = @(M_PI_2);
        transformAnima.toValue = @(M_PI);
```

```
        transformAnima.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseInEaseOut];

        CABasicAnimation *positionAnima = [CABasicAnimation
animationWithKeyPath:@"position.y"];
        positionAnima.fromValue = @(self.starView.center.y -
14);
        positionAnima.toValue = @(self.starView.center.y);
        positionAnima.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseIn];

        CAAnimationGroup *animGroup = [CAAnimationGroup anima-
tion];
        animGroup.duration = downDuration;
        animGroup.fillMode = kCAFillModeForwards;
        animGroup.removedOnCompletion = NO;
        animGroup.delegate = self;
        animGroup.animations = @[transformAnima,positionAnima];

        [self.starView.layer addAnimation:animGroup
forKey:@"jumpDown"];

    }else if([anim isEqual:[self.starView.layer
animationForKey:@"jumpDown"]]){

        
          [self.starView.layer removeAllAnimations];
          animating = NO;

    }

}
```

关于两个阶段动画时间上的思考。理论上，在忽略空气阻力的情况下，两个阶段的动画一定是相等的，无论物体的初速度为多少。但是现实世界中，我们看到的现象是：如果以一个初速度往上抛一个物体，你的感觉是物体上抛过程花费的时间短，下落花费的时间长。因为上抛是

个减速运动，而下落是个加速运动。这个感觉很重要，虽然违背物理规律，但是我们要做的就是更真实地模拟人对现实世界的感知。所以我们在做动画中需要做这些违背物理规律的事情：即把下落的时间设置得比上弹的长。以期符合真实感知。更何况，CoreAnimation 中是不能设置初速度的，默认都是从零速度开始运动，所以我们必须通过缩短上抛时间以此来模拟出这个初速度，让动画看起来更真实。

```
#define jumpDuration 0.125
#define downDuration 0.215
```

一个优秀动画的特点除了复杂之外，也就是短时间内输出足够多的信息量；另一个特点，就是细节决定质量。在这个 demo 中，我在星星弹跳的底部加了一个阴影，随着视图的上弹下落阴影的 bounds 也会有动画。千万别认为这些属于情怀的东西「然并卵」，用户不是傻子，你用心设置的细节都会被用户察觉到。然后别宣传，要等用户自己去发现。这和提前就告诉用户带去的心理感受是完全不一样的。这些东西也是我在锤子科技耳濡目染学到的，在以后的动画教程中，我除了在介绍技术层面的东西之外，也会额外穿插一些人性化、感性方面的小心得，小体会，拿出来权当供各位参考。好了，至于这个阴影动画的时机，我设置在了两个动画的一开始，也就是 animationDidStart 中：

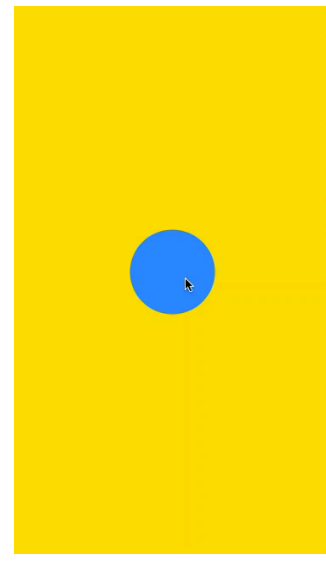
```
- (void)animationDidStart:(CAAnimation *)anim{

    if ([anim isEqual:[self.starView.layer
animationForKey:@"jumpUp"]]) {
        [UIView animateWithDuration:jumpDuration delay:0.0f
options:UIViewAnimationOptionCurveEaseOut animations:^{

            _shadowView.alpha = 0.2;
            _shadowView.bounds = CGRectMake(0, 0,
_shadowView.bounds.size.width*1.6,
_shadowView.bounds.size.height);

        } completion:NULL];

    }else if ([anim isEqual:[self.starView.layer
animationForKey:@"jumpDown"]]){

        [UIView animateWithDuration:jumpDuration delay:0.0f
options:UIViewAnimationOptionCurveEaseOut animations:^{

            _shadowView.alpha = 0.4;
            _shadowView.bounds = CGRectMake(0, 0,
_shadowView.bounds.size.width/1.6,
_shadowView.bounds.size.height);

        } completion:NULL];

    }
}
```

第三节到这里也就结束了。源码请见 **JumpStarDemo**。

第4节要呈现的效果如 Movie 3.5 所示：

这是一个下载按钮的动画，涉及到了 `cornerRadius,bounds,strokeEnd` 等相关属性的动画。

Movie 3.5 下载按钮动画



首先我们讲讲这个 `cornerRadius` 。顾名思义，这个属性是用来绘制矩形的圆角，具体这个值表示的意义是这样的。

正如图中展示的那样，如果你让想让一个正方形变成圆形，那么你所要做的就是把 `cornerRadius` 这个值变成边长的 1/2。

同理，如果是一个矩形，想让两头变为圆角，只需要把 `cornerRadius` 设置成矩形高的 1/2 即可。

设置了 `cornerRadius` 之后别忘了，记得开启 `self.clipsToBounds = YES;` 或者 `self.layer.masksToBounds = YES;` 把圆角之外的部分「切除」。

这个 demo 并没有非常复杂的算法，纯粹就是不同动画在不同时间上的叠加，只要条理清晰，想清楚一个动画结束后该接什么动画就行了。下面我们着重看看代码。

```
-(void)tapped:(UITapGestureRecognizer *)tapped{
    originframe = self.frame;

    if (animating == YES) {
        return;
    }

    for (CALayer *subLayer in self.layer.sublayers) {
        [subLayer removeFromSuperlayer];
    }

    self.backgroundColor = [UIColor colorWithRed:0.0
green:122/255.0 blue:255/255.0 alpha:1.0];

    animating = YES;

    self.layer.cornerRadius = self.progressBarHeight/2;
    CABasicAnimation *radiusAnimation = [CABasicAnimation
animationWithKeyPath:@"cornerRadius"];
    radiusAnimation.duration = 0.2f;
    radiusAnimation.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseOut];
    radiusAnimation.fromValue = @(originframe.size.height/2);

    radiusAnimation.delegate = self;
    [self.layer addAnimation:radiusAnimation
forKey:@"cornerRadiusShrinkAnim"];
}
-(void)animationDidStart:(CAAnimation *)anim{

    if ([anim isEqual:[self.layer
animationForKey:@"cornerRadiusShrinkAnim"]]) {

        [UIView animateWithDuration:0.6f delay:0.0f
usingSpringWithDamping:0.6 initialSpringVelocity:0.0
options:UIViewAnimationOptionCurveEaseOut animations:^{
```

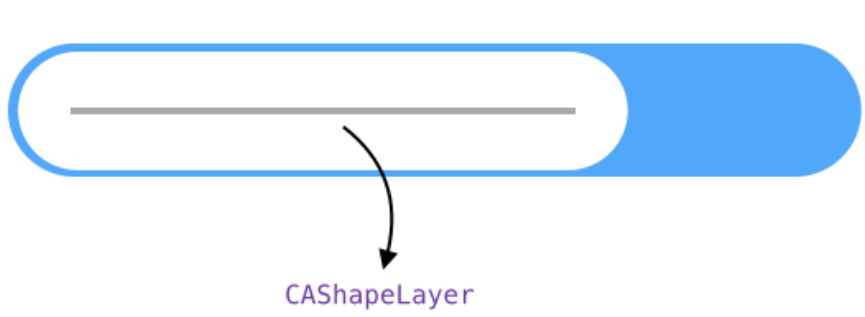
```
            self.bounds = CGRectMake(0, 0, _progressBarWidth,
_progressBarHeight);
        } completion:^(BOOL finished) {
            [self.layer removeAllAnimations];
            [self progressBarAnimation];
        }];
    }
}
```

圆变成进度条了，接下来就是进度条动画了。正如我上面代码中写的那样，在上一个动画结束之后调用了 [self progressBarAnimation];

```
-(void)progressBarAnimation{
    CAShapeLayer *progressLayer = [CAShapeLayer layer];
    UIBezierPath *path = [UIBezierPath bezierPath];
    [path moveToPoint:CGPointMake(_progressBarHeight/2,
self.bounds.size.height/2)];
    [path
addLineToPoint:CGPointMake(self.bounds.size.width-_progressBarH
eight/2, self.bounds.size.height/2)];

    progressLayer.path = path.CGPath;
    progressLayer.strokeColor = [UIColor whiteColor].CGColor;
    progressLayer.lineWidth = _progressBarHeight-6;
    progressLayer.lineCap = kCALineCapRound;

    [self.layer addSublayer:progressLayer];

    CABasicAnimation *pathAnimation = [CABasicAnimation
animationWithKeyPath:@"strokeEnd"];
    pathAnimation.duration = 2.0f;
    pathAnimation.fromValue = @(0.0f);
    pathAnimation.toValue = @(1.0f);
    pathAnimation.delegate = self;
    [pathAnimation setValue:@"progressBarAnimation"
forKey:@"animationName"];
    [progressLayer addAnimation:pathAnimation forKey:nil];

}
```

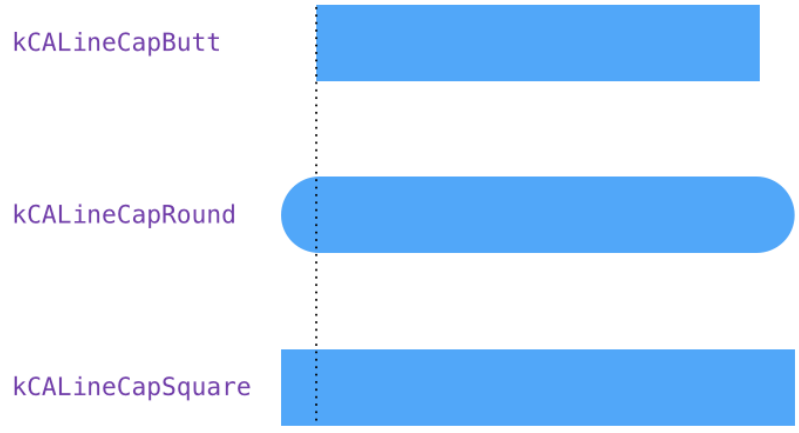
关于进度条动画，我不打算使用 popup 的方式介绍。我将单独拿出篇幅来讲讲。

首先你要知道这类进度的动画，都是用的 strokeEnd 属性。而 strokeEnd 不是 CALayer 的属性，而是其子类 CAShapeLayer 的一个特有的属性。所以我们必须创建一个 CAShapeLayer。其次，一个必须赋值的参数就是 path。Demo 中，我们绘制了一条直线作为 CAShapeLayer 的 path。



如何设置直线的起始点才能让白色进度条距离四周的间距相等呢？结论是 x = _progressBarHeight/2. 证明如下：

因为我们设置了 progressLayer.lineCap = kCALineCapRound; lineCap 指的是线段的线帽，也就是决定一条线段两段的封口样式，有三种样式可以选择：



- kCALineCapButt: 默认格式，不附加任何形状;
- kCALineCapRound: 在线段头尾添加半径为线段 lineWidth 一半的半圆;
- kCALineCapSquare: 在线段头尾添加半径为线段 lineWidth 一半的矩形

由于我们之前设置了 kCALineCapRound, 而半圆的半径就是 lineWidth/2, 所以起始点的 x 坐标应该满足公式 x = space + lineWidth/2, 又 ∵ lineWidth = _progressBarHeight - space*2 ∴ x = _progressBarHeight/2, 也就是说起始点的 x 坐标与 lineWidth 的值并没有关系。

所以，只要保证了 path 的起点 x 坐标等于外围进度条（demo 中的蓝色进度条）高度的 1/2 ，那么无论设置 path 的 lineWidth 为多少都可以让白色进度条距离四周的间距相等。

关于 @property CGFloat strokeStart; 和 @property CGFloat strokeEnd; 这两个属性，正如它的名字一样，定义了线段的开始和结

束，并且取值都在[0,1]之间。默认 strokeStart 为 0，strokeEnd 为 1。通过设置不同的值，可以控制线条的展示状态。

注意 [pathAnimation setValue:@"progressBarAnimation" forKey:@"animationName"]; 这一句，就是我们之前说的判断不同 anim 的第二种方法：KVO.

当进度条动画走完后，我们先让进度条做一个透明度到 0 的动画，之后立马同时开始一个 cornerRadius 动画和一个 bounds 动画，让进度条恢复到圆形状态。

```
-(void)animationDidStop:(CAAnimation *)anim
finished:(BOOL)flag{

    if ([[anim
valueForKey:@"animationName"]isEqualToString:@"progressBarAnima
tion"]){

        [UIView animateWithDuration:0.3 animations:^{
            for (CALayer *subLayer in self.layer.sublayers) {
                subLayer.opacity = 0.0f;
            }
        } completion:^(BOOL finished) {
            if (finished) {
                for (CALayer *subLayer in self.layer.sublayers)
{
                    [subLayer removeFromSuperlayer];
                }

                self.layer.cornerRadius =
originframe.size.height/2;
```

```
                CABasicAnimation *radiusAnimation = [CABasicAni-
mation animationWithKeyPath:@"cornerRadius"];
                radiusAnimation.duration = 0.2f;
                radiusAnimation.timingFunction = [CAMediaTiming-
Function functionWithName:kCAMediaTimingFunctionEaseOut];
                radiusAnimation.fromValue =
@(_progressBarHeight/2);

                radiusAnimation.delegate = self;
                [self.layer addAnimation:radiusAnimation
forKey:@"cornerRadiusExpandAnim"];

            }
        }];

    }
}

-(void)animationDidStart:(CAAnimation *)anim{

    if ([anim isEqual:[self.layer
animationForKey:@"cornerRadiusExpandAnim"]]){

        [UIView animateWithDuration:0.6f delay:0.0f
usingSpringWithDamping:0.6 initialSpringVelocity:0.0
options:UIViewAnimationOptionCurveEaseOut animations:^{
            self.bounds = CGRectMake(0, 0,
originframe.size.width, originframe.size.height);
            self.backgroundColor = [UIColor
colorWithRed:0.18039215686274510 green:0.8
blue:0.44313725490196076 alpha:1.0];
        } completion:^(BOOL finished) {
            [self.layer removeAllAnimations];
            [self checkAnimation];
            //-----
            animating = NO;
        }];
    }
}
```

进度条恢复到圆形状态之后，我们就该进度打勾的动画了。正如你在上面看到的那样，我在代码最后也就是 **bounds** 动画结束时 removeAllAnimations ，并调用了 [self checkAnimation];

打勾动画的思路依然是给一个 CAShapeLayer 指定一个勾形的 path，然后进行 strokeEnd 的动画。

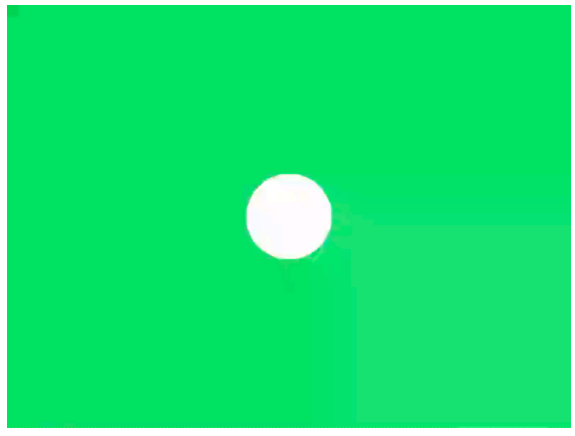
```
-(void)checkAnimation{
    CAShapeLayer *checkLayer = [CAShapeLayer layer];
    UIBezierPath *path = [UIBezierPath bezierPath];
    CGRect rectInCircle = CGRectInset(self.bounds,
self.bounds.size.width*(1-1/sqrt(2.0))/2,
self.bounds.size.width*(1-1/sqrt(2.0))/2);
    [path moveToPoint:CGPointMake(rectInCircle.origin.x +
rectInCircle.size.width/9, rectInCircle.origin.y +
rectInCircle.size.height*2/3)];

    [path addLineToPoint:CGPointMake(rectInCircle.origin.x +
rectInCircle.size.width/3,rectInCircle.origin.y +
rectInCircle.size.height*9/10)];
    [path addLineToPoint:CGPointMake(rectInCircle.origin.x +
rectInCircle.size.width*8/10, rectInCircle.origin.y +
rectInCircle.size.height*2/10)];

    checkLayer.path = path.CGPath;
    checkLayer.fillColor = [UIColor clearColor].CGColor;
    checkLayer.strokeColor = [UIColor whiteColor].CGColor;
    checkLayer.lineWidth = 10.0;
    checkLayer.lineCap = kCALineCapRound;
    checkLayer.lineJoin = kCALineJoinRound;
    [self.layer addSublayer:checkLayer];
```

```
    CABasicAnimation *checkAnimation = [CABasicAnimation
animationWithKeyPath:@"strokeEnd"];
    checkAnimation.duration = 0.3f;
    checkAnimation.fromValue = @(0.0f);
    checkAnimation.toValue = @(1.0f);
    checkAnimation.delegate = self;
    [checkAnimation setValue:@"checkAnimation"
forKey:@"animationName"];
    [checkLayer addAnimation:checkAnimation forKey:nil];
}
```

现在你已经学完了这个 demo 。理论上，所有描线的动画你都可以用这种方式先指定一个 path 然后改变 strokeEnd, strokeStart 来实现。

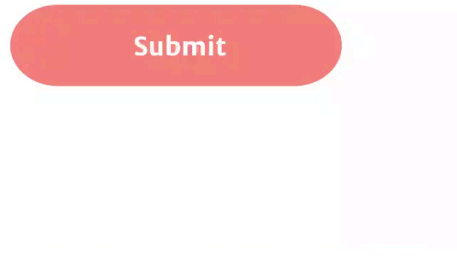
比如我在 dribbble 上找到的两则动画：

Movie 3.6 线条动画原型 001



Paperclip Loader *-by Jokūbas*

Movie 3.7 线条动画原型 002



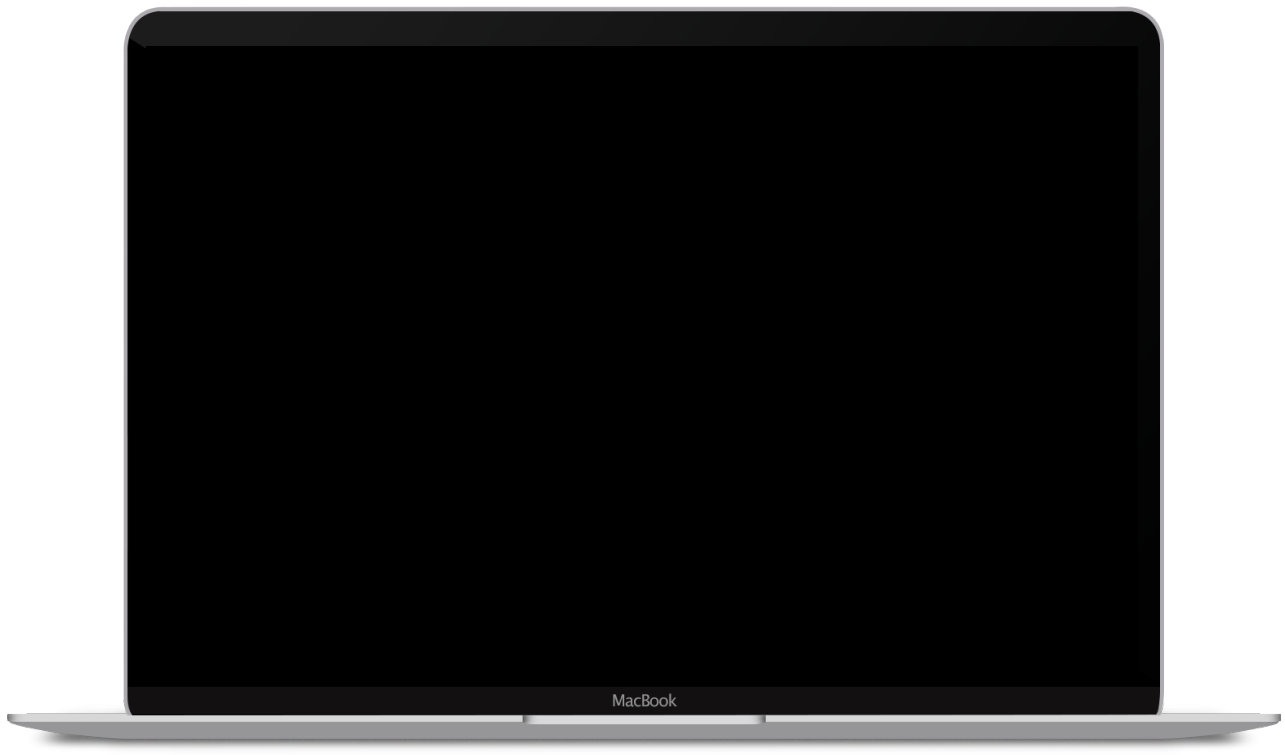
Submit *-by Lars Lundberg*

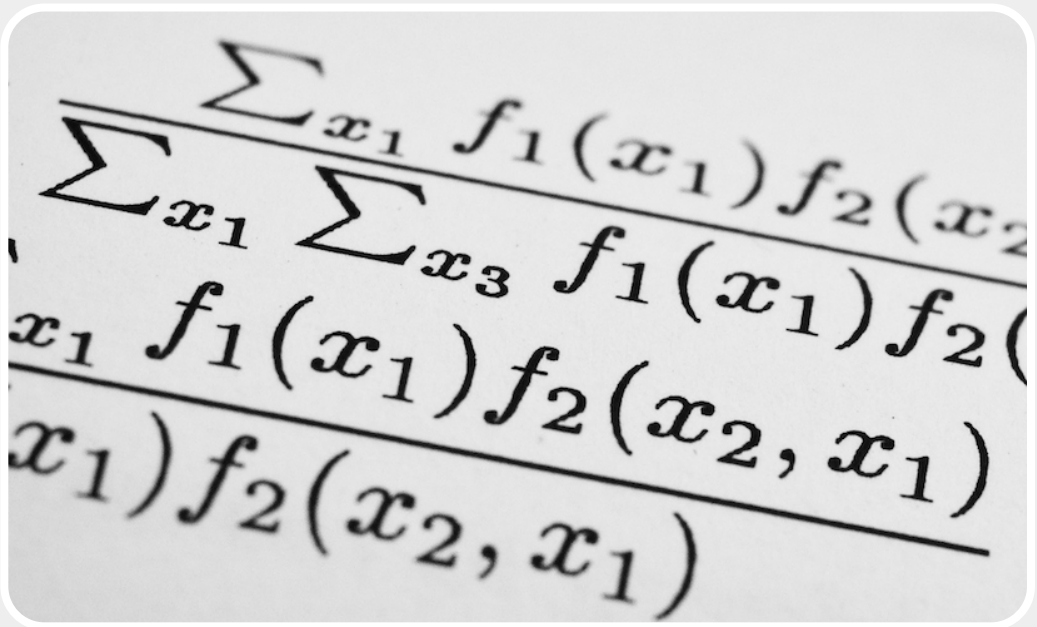
最后，我还想补充的一点是关于 CAKeyframeAnimation 中 @property CGPathRef path; 这一属性。这是经典的路径动画。你只需指定一个指针类型的 CGPathRef ,剩下的事情就交给 CAKeyframeAnimation 吧——视图的 anchorPoint 就会沿着你给出的这条 path 运动。比如右侧

这个 Movie 3.8 中 Loading 的例子，我也放在了 A-GUIDE-TO-iOS-ANIMATION 下的 KY-LoadingHUD 下，代码中无非就是多了个 path 属性，相信已经不用我解释了，你一眼就能看懂。

Movie 3.8 path 属性的效果

最后，附上 *WWDC 2011 Session 421 —— Core Animation Essentials*




$\frac{\sum_{x_1} f_1(x_1) f_2(x_2}{\sum_{x_1} \sum_{x_3} f_1(x_1) f_2($
$x_1) f_2(x_2, x_1)$
$x_1 f_1(x_1) f_2(x_2, x_1)$

4

動畫中的數學

如果要我说我最喜欢哪种类型的动画，我想应该就是这一节的主题——「结合数学知识的动画」。尽管我在前面的章节中后多或少都涉及到了数学知识的使用，但毕竟不是主角。而这一章，我将把数学的角色晋升到第一位置，让数学的魅力延伸到程序中来。除此之外，你还必须努力习惯动手在纸上画草图，习惯找到数学和程序之间的衔接点。这将会非常有趣，也会很有成就感。

这一章的第一个知识点，我们来聊聊动画曲线（**timingFunction**）。

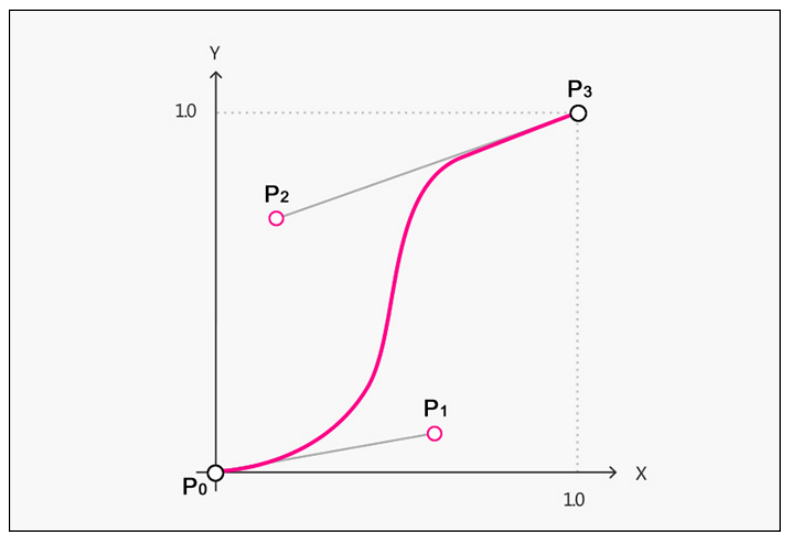
众说周知，我们常用的 **CoreAnimation** 动画曲线只有默认提供的四个枚举值：

```
  CA_EXTERN NSString * const kCAMediaTimingFunctionLinear
    __OSX_AVAILABLE_STARTING (__MAC_10_5, __IPHONE_2_0);

  CA_EXTERN NSString * const kCAMediaTimingFunctionEaseIn
    __OSX_AVAILABLE_STARTING (__MAC_10_5, __IPHONE_2_0);

  CA_EXTERN NSString * const kCAMediaTimingFunctionEaseOut
    __OSX_AVAILABLE_STARTING (__MAC_10_5, __IPHONE_2_0);

  CA_EXTERN NSString * const kCAMediaTimingFunctionEaseInEase-
Out
    __OSX_AVAILABLE_STARTING (__MAC_10_5, __IPHONE_2_0);
```

当然，你完全可以自己创建时间曲线。使用 CAMediaTimingFunction 中的

```
+ (instancetype)functionWithControlPoints:(float)c1x
:(float)c1y :(float)c2x :(float)c2y;
```

方法就可以创建一个 timingFunction。请看下方这幅图，就是对上面这个方法的解释。有没有觉得很熟悉，没错，又是贝塞尔曲线。正如我 前面 说的那样，这绝对是对计算机图形学领域具有里程碑意义的学术成果。



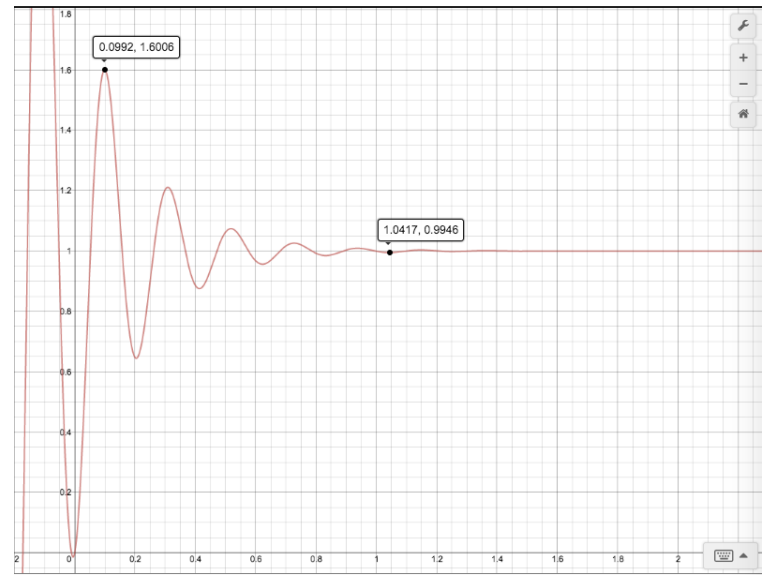
具体讲讲自定义 timingFunction 的方法。其中的 c_1x, c_1y 代表第一个关键点，也就是图中 *p1* 点， c_2x, c_2y 代表第二个关键点，对应图中 p2 点。从图中可以看出， c_1x, c_1y,c_2x, c_2y 的范围都是 [0,1] 。这和 CA-Layer 的 anchorPoint 很类似。然而头疼的是，每次自定义都要计算两个关键点的坐标的确是一件让人望而却步的事情。

不过，历史的经验告诉我们，绝大多数问题你都不会是第一个想到的，我们都是站在巨人的肩上看更远的风景。尤其对于这种被众多领域

普遍使用的东西，各种方便生成贝塞尔曲线关键点的工具可谓是层出不穷。前人早就已经开发一大堆了，步骤简单到只需通过拖拽控制点描绘出你想要的曲线，工具就立刻自动生成了关键点坐标。这里推荐我常用的一个工具，就是这个网站 —— **Rob La Placa** 。或者有款 **Xcode**插件 —— **CATweaker**。

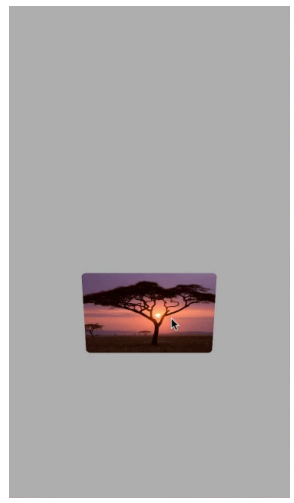
然后，我们可以更进一步。抛开使用 CAMediaTimingFunction，毕竟这限制了动画只能从 0 变化到 1，不能做到完全自定义，比如先从 0 ease-out 变化到 1，再 ease-in 变化回到 0；或者更复杂，比如想要以震荡曲线 **Image 4.1** 的规律运动。所以，我们就会很自然地想到，如果能把任意一条数学中的函数图像转化成自定义的动画曲线，那该有多好。

Image 4.1 震荡曲线



这次的案例，我们将介绍实现一个平滑的手势驱动动画。还是先请预览最终效果 **Movie 4.1**。

Movie 4.1 手势驱动动画最终效果



整体思路是这样的：

- 设定最大滑动距离为 120。
- 随着滑动距离绝对值（离开初始位置的距离，竖直向上或竖直向下）的增加，逐渐接近最大滑动距离。这个过程中，视图同时做三个变换：
  - 第一个是 translation .让视图的 center 的位移等于手指的位移；
  - 其次是 scale . 从 1.0 到 0.8；
  - 另一个变换是 Rotate(绕 x 轴且带透视效果)，从0增长到1，之后立即从1减小到0。

以上三个分运动叠加在一起，就是 **Movie 4.1** 的效果了。

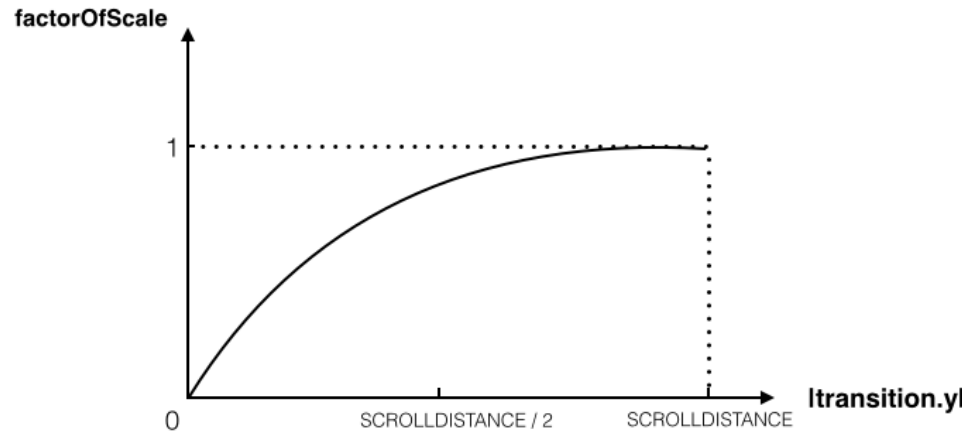
既然是三个分运动，我们还是把他们分解开来，各个击破。 循序渐进的成就感才是最有持久的。首先是位移，这个非常容易。

```
-(void)panGestureRecognized:(UIPanGestureRecognizer *)pan{

     static CGPoint initialPoint = currentPhoto.center;
     CGFloat factorOfAngle = 0.0f;
     CGFloat factorOfScale = 0.0f;
    CGPoint transition = [pan translationInView:self.view];

    if (pan.state == UIGestureRecognizerStateBegan) {
         initialPoint = self.center;
      }else if{
       if(pan.state == UIGestureRecognizerStateChanged){
        self.center = CGPointMake(initialPoint.x,initialPoint.y
+ transition.y);
       }
    }
}
```

然后是实现 scale 变换。先确定以什么样的动画曲线进行动画：我们让视图以平滑的二次函数曲线从 1.0 缩小到 0.8，所以这个差值 0.2 的函数曲线就可以遵循如下的二次函数：

滑动距离到达最大规定距离时，动画也就到达了末状态，我们使用一个系数 **factorOfScale** 。下面就可以建模出一道高中数学题了：

IMAGE 4.2 scale 的函数图像



已知一条开口向下的二次函数曲线，顶点为*(SCROLLDISTANCE, 1)*,且经过*(0,0)*和*(2*SCROLLDISTANCE,0)*两点，定义域为*[0,SCROLLDISTANCE]*，求该曲线的方程。

如果你还记得初中数学知识的话，我们可以很快地求解得到 $y = -\frac{1}{s^2}x(x-2s)$ 。转换成代码就是：

```
factorOfScale =
MAX(0,-1/(SCROLLDISTANCE*SCROLLDISTANCE)*Y*(Y-2*SCROLLDISTANCE)
);
```

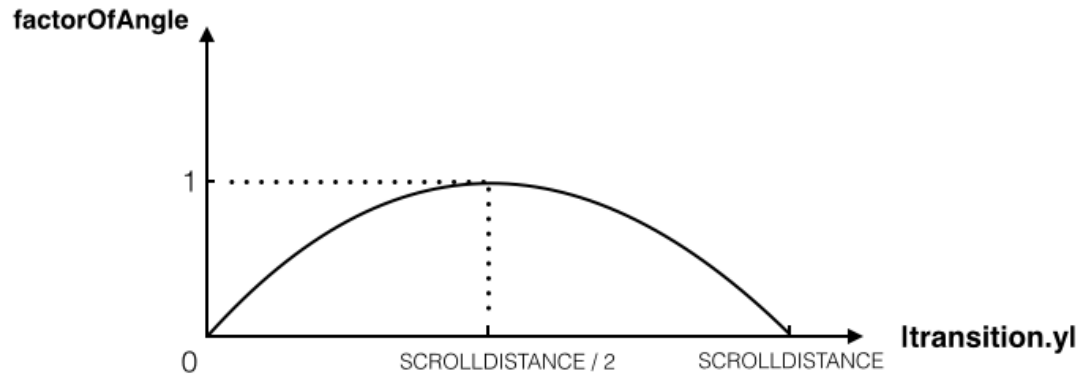
接下来创建一个 CATransform3D ，并把这个 CATransform3D 赋值给 layer 的 transform 属性。把这段代码放在手势绑定的方法中：

```
if(pan.state == UIGestureRecognizerStateChanged){

      ...

      CGFloat Y =MIN(SCROLLDISTANCE,MAX(0,ABS(transition.y)));

        //一个开口向下,顶点(SCROLLDISTANCE,1),过(0,0),
(2*SCROLLDISTANCE,0)的二次函数
        factorOfScale =
MAX(0,-1/(SCROLLDISTANCE*SCROLLDISTANCE)*Y*(Y-2*SCROLLDISTANCE)
);

      CATransform3D t = CATransform3DIdentity;
      t = CATransform3DScale(t, 1-factorOfScale*0.2,
1-factorOfScale*0.2, 0);
```

```
        self.layer.transform = t;

    ...
}
```

最后就是 Rotate 变换。根据我们设想的那样，我们需要让图片先往里转到最大值，比如 36°；随后向外旋转回到 0°。结合平滑的运动曲线，因此很容易想到右面这条函数曲线。

IMAGE 4.3 rotate 的函数图像



有没有觉得曲线很熟悉？是的，同样也可以近似地看成一条一元二次曲线，变量为 transition.y（准确的说，应该是ABS(transition.y)),且 0 <= ABS(transition.y) <= SCROLLDISTANCE)。建模完成，又到了复习高中数学知识的时间了：

已知一条开口向下的二次曲线，顶点为*(SCROLLDISTANCE/2,1)*，且经过*(0,0)*和*(SCROLLDISTANCE,0)*两点，定义域为*[0,SCROLLDISTANCE]*，求该曲线的方程。

求解得到 $y = -\frac{4}{s^2}x(x-s)$ ，转换成代码就是

```
factorOfAngle = MAX(0,-4/(SCROLLDISTANCE*SCROLLDISTANCE)*Y*(Y-SCROLLDISTANCE));
```

在之前的代码中为 CATransform3D t 追加变换，最终得到的是：

```
-(void)panGestureRecognized:(UIPanGestureRecognizer *)pan{

    static CGPoint initialPoint;
    CGFloat factorOfAngle = 0.0f;
    CGFloat factorOfScale = 0.0f;
    CGPoint transition = [pan translationInView:self.superview];
    if (pan.state == UIGestureRecognizerStateBegan) {

        initialPoint = self.center;

    }else if(pan.state == UIGestureRecognizerStateChanged){


        self.center = CGPointMake(initialPoint.x,initialPoint.y + transition.y);

        CGFloat Y =MIN(SCROLLDISTANCE,MAX(0,ABS(transition.y)));

        //一个开口向下,顶点(SCROLLDISTANCE/2,1),过(0,0),(SCROLLDISTANCE,0)的二次函数
        factorOfAngle = MAX(0,-4/(SCROLLDISTANCE*SCROLLDISTANCE)*Y*(Y-SCROLLDISTANCE));
        //一个开口向下,顶点(SCROLLDISTANCE,1),过(0,0),(2*SCROLLDISTANCE,0)的二次函数
        factorOfScale = MAX(0,-1/(SCROLLDISTANCE*SCROLLDISTANCE)*Y*(Y-2*SCROLLDISTANCE));

        CATransform3D t = CATransform3DIdentity;
```

```
        t.m34  = 1.0/-1000;
        t = CATransform3DRotate(t,factorOfAngle*(M_PI/5),
transition.y>0?-1:1, 0, 0);
        t = CATransform3DScale(t, 1-factorOfScale*0.2,
1-factorOfScale*0.2, 0);

        self.layer.transform = t;

    }

}
```

本节内容到这里也就结束了。你可以在 **这里** 下载到源码。

下面这个案例，我把线条动画和数学知识结合在了一起。

通过这个案例，可以很好地向你展示如何自己归纳出一个数学公式，并把它用到一个自定义动画中。

首先，我们还是先看最终效果 **Movie 4.2**：

OK，可以看到随着手指在屏幕上滑动距离的改变，线条一开始逐渐靠拢，到达一定位置后开始弯曲，最终合并成了一个圆。你可能也已经注意到，我已经把这个动画封装到了一个上拉、下拉刷新的控件中，并且用在了大象公会这

Movie 4.2 自定义的下拉线条动画

款独立开发的 App 中。

下面让我讲讲我思考这个动画的整个过程。首先，最终控制这个动画进度的是一个 CALayer 内部的自定义属性：

```
@property(nonatomic,assign)CGFloat progress;
```

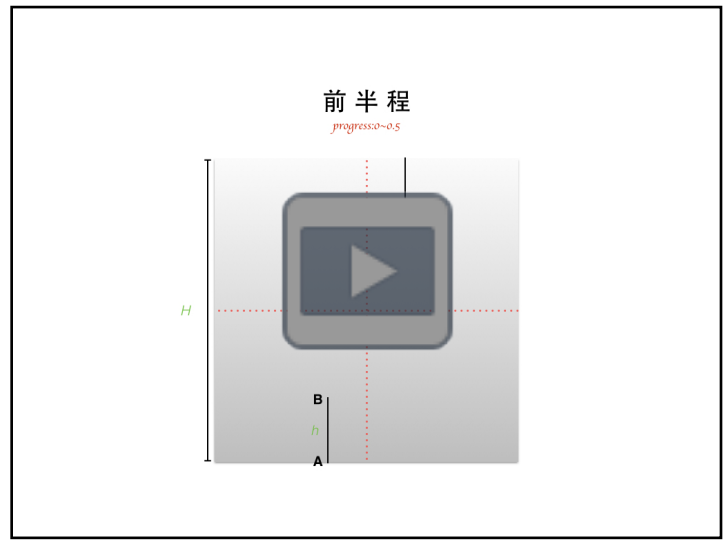
无论你是通过手指滑动产生偏移量，还是滑动 UISlider 改变一个数值，最终都将转化到这个属性的改变。然后，在这个属性的 setter 方法里，我们让 layer 去实时重绘，就像这样：

```
-(void)setProgress:(CGFloat)progress{
    self.curveLayer.progress = progress;
    [self.curveLayer setNeedsDisplay];
}
```

至于重绘的算法，这属于细节上要考虑的事了。我们做一个动画的步骤是先把宏观上的思路理清，再去考虑细节上的实现。就像开发一个 App 一样，一开始肯定是先考虑架构，再去往这个框架里添砖加瓦，修修补补。现在，我们对这个动画的整体思路已经清楚了，下面开始深入到细节去思考具体算法的实现。我把这个动画分成了两部分：*0 ~ 0.5* 和 *0.5 ~ 1.0*. 什么意思呢？我给你做了两个 Keynote：KEYNOTE 4.1，KEYNOTE 4.2。

还是那句话 ——「善于分解」。我们先看前半程，也就是 progress 从一开始的 0 运动到中间状态 0.5 的这一个阶段。这一个阶段两条线段分别从上方和下方两个方向向中间运动，直到接触到中线为止。这一阶段的画线算法非常简单，只要能实时获得 A,B 两点的坐标，剩下用 UIBezierPath 的 moveToPoint,addLineToPoint 就完事了。所以，问题转换成了求 A,B 两点运动的公式（其实只要求出一点，另一点无非就相差了一个线段长度 h）。这里我纠结了好久，该用什么方式像你介绍计算出这两个公式的过程，最后我能想到的只有通过做 Keynote 4.1，Keynote 4.2 这两个演示文稿的方式，剩下的就只能意会不能言传了。其实你只要愿意动笔在纸上尝试推演一番，并不难求得这两个点的运动公式：

Keynote 4.1 线条弯曲动画前半程分析



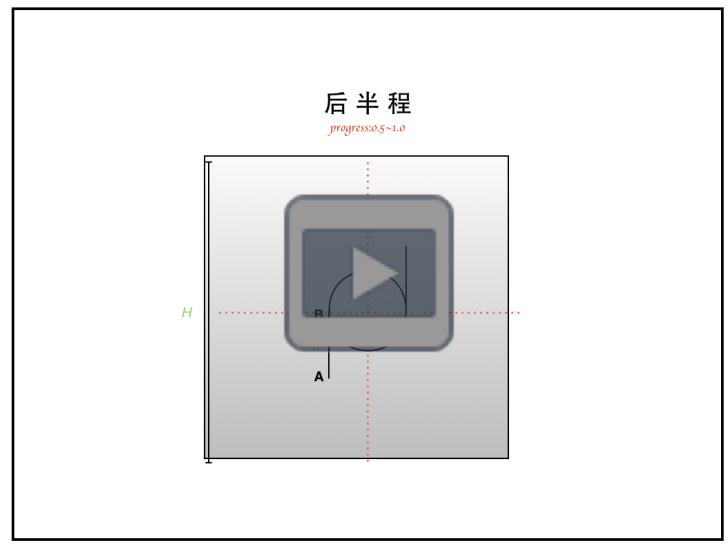
$$y_A = H/2 + h + (1-2*progress) * (H/2 - h)$$

$$y_B = H/2 + (1-2*progress) * (H/2 - h)$$

接下来是动画的第二阶段 *0.5 ~ 1.0*。这个阶段有些许复杂：「B 点保持不动，A 点继续运动到 B 的位置，同时，在顶部根据当前的进度再

画出圆弧」。视觉上给人的感觉就好像尾巴在逐渐缩短，头部在慢慢弯曲。

KEYNOTE 4.2 线条弯曲动画前半程和后半程的分析



在这个过程中，我们不难先求得 A 点的坐标是：

$$y_A = H/2 + h - h*(progress - 0.5) *2$$

比较麻烦的是这个圆弧该怎么画？答案是可以用 UIBezierPath 中提供的

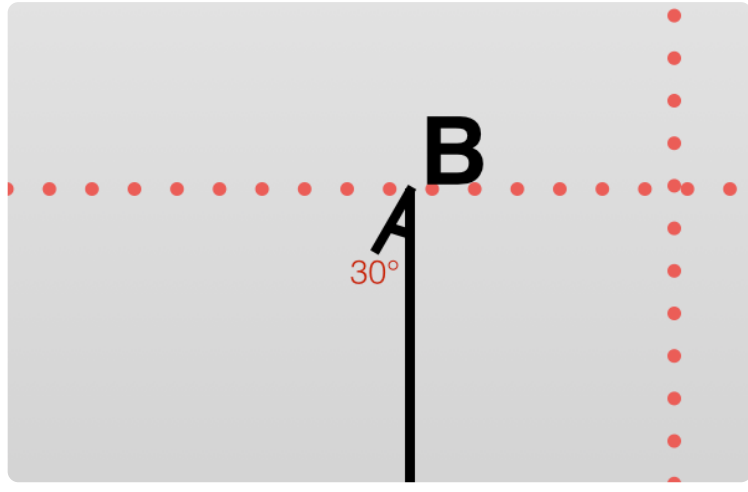
```
- (void)addArcWithCenter:(CGPoint)center radius:(CGFloat)radius
startAngle:(CGFloat)startAngle endAngle:(CGFloat)endAngle
clockwise:(BOOL)clockwise NS_AVAILABLE_IOS(4_0);
```

这个方法绘制出圆弧。具体算法是：以 CGPointMake(self.frame.size.width/2, self.frame.size.height/2) 为圆心，10 为半径，按顺时针方向，从 M_PI(90°) 的起始角度，画到 2*M_PI 的结束角度。

到这里，我们只完成了一条线段的整个过程。同理，也能获得另一条线段的绘制算法。最后，别忘了线段顶端还有个箭头。绘制箭头的算法

GALLERY 4.1：我们以 B 点作为箭头的起始起点，斜向左下方 30° 角延长 3 个单位。弯曲之后也同理，只需要额外加上线段转过的角度即可。

GALLERY 4.1 掩饰线段顶部箭头的绘制思路



相应的代码就是：

```
[arrowPath moveToPoint:pointB];
[arrowPath addLineToPoint:CGPointMake(pointB.x - 3*(cosf(De-
gree)), pointB.y + 3*(sinf(Degree)))];
[curvePath1 appendPath:arrowPath];
```

最终，整个动画完整的绘制算法如下：

```
-(void)drawInContext:(CGContextRef)ctx{

    [super drawInContext:ctx];


    UIGraphicsPushContext(ctx);
    CGContextRef context = UIGraphicsGetCurrentContext();

    //--------- Draw ----------

    //Path 1
```

```
    UIBezierPath *curvePath1 = [UIBezierPath bezierPath];
    curvePath1.lineCapStyle = kCGLineCapRound;
    curvePath1.lineJoinStyle = kCGLineJoinRound;
    curvePath1.lineWidth = 2.0f;

    //arrowPath
    UIBezierPath *arrowPath = [UIBezierPath bezierPath];

    if (self.progress <= 0.5) {

        CGPoint pointA =
CGPointMake(self.frame.size.width/2-Radius, CenterY - Space +
LineLength + (1-2*self.progress)*(CenterY-LineLength));
        CGPoint pointB =
CGPointMake(self.frame.size.width/2-Radius, CenterY - Space +
(1-2*self.progress)*(CenterY-LineLength));
        [curvePath1 moveToPoint:pointA];
        [curvePath1 addLineToPoint:pointB];

        //arrow
        [arrowPath moveToPoint:pointB];
        [arrowPath addLineToPoint:CGPointMake(pointB.x - 3*(cos-
f(Degree)), pointB.y + 3*(sinf(Degree)))];
        [curvePath1 appendPath:arrowPath];

    }else if (self.progress > 0.5) {

        CGPoint pointA =
CGPointMake(self.frame.size.width/2-Radius, CenterY - Space +
LineLength - LineLength*(self.progress-0.5)*2);
        CGPoint pointB =
CGPointMake(self.frame.size.width/2-Radius, CenterY - Space);

        [curvePath1 moveToPoint:pointA];
        [curvePath1 addLineToPoint:pointB];
        [curvePath1
addArcWithCenter:CGPointMake(self.frame.size.width/2, CenterY-
Space) radius:Radius startAngle:M_PI endAngle:M_PI + ((M_PI*9/
10) * (self.progress-0.5)*2) clockwise:YES];

        //arrow
        [arrowPath moveToPoint:curvePath1.currentPoint];
```

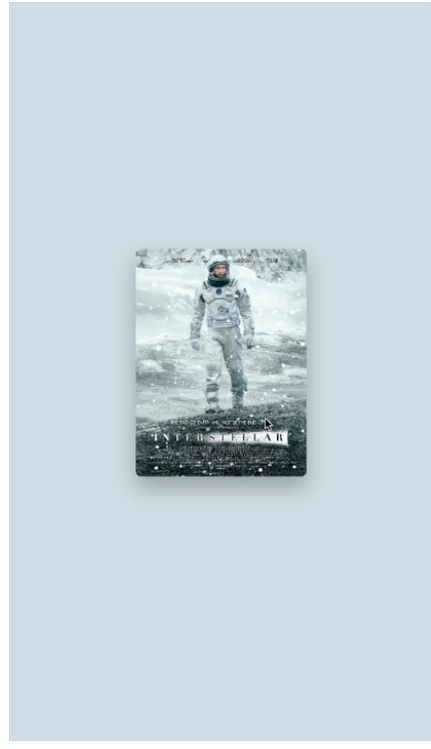
```
        [arrowPath
addLineToPoint:CGPointMake(curvePath1.currentPoint.x - 3*(cos-
f(Degree  - ((M_PI*9/10) * (self.progress-0.5)*2))),
curvePath1.currentPoint.y + 3*(sinf(Degree - ((M_PI*9/10) *
(self.progress-0.5)*2)))))];
        [curvePath1 appendPath:arrowPath];
    }


    //Path 2
    UIBezierPath *curvePath2 = [UIBezierPath bezierPath];
    curvePath2.lineCapStyle = kCGLineCapRound;
    curvePath2.lineJoinStyle = kCGLineJoinRound;
    curvePath2.lineWidth = 2.0f;
    if (self.progress <= 0.5) {

        CGPoint pointA =
CGPointMake(self.frame.size.width/2+Radius, 2*self.progress *
(CenterY + Space - LineLength));
        CGPoint pointB =
CGPointMake(self.frame.size.width/2+Radius,LineLength +
2*self.progress*(CenterY + Space - LineLength));
        [curvePath2 moveToPoint:pointA];
        [curvePath2 addLineToPoint:pointB];

        //arrow
        [arrowPath moveToPoint:pointB];
        [arrowPath addLineToPoint:CGPointMake(pointB.x + 3*(cos-
f(Degree)), pointB.y - 3*(sinf(Degree)))];
        [curvePath2 appendPath:arrowPath];

    }else if (self.progress > 0.5) {

        [curvePath2
moveToPoint:CGPointMake(self.frame.size.width/2+Radius, CenterY
+ Space - LineLength + LineLength*(self.progress-0.5)*2)];
        [curvePath2
addLineToPoint:CGPointMake(self.frame.size.width/2+Radius, Cen-
terY + Space)];
        [curvePath2
addArcWithCenter:CGPointMake(self.frame.size.width/2, (Cen-
terY+Space)) radius:Radius startAngle:0
endAngle:(M_PI*9/10)*(self.progress-0.5)*2 clockwise:YES];
```

```
        //arrow
        [arrowPath moveToPoint:curvePath2.currentPoint];
        [arrowPath
addLineToPoint:CGPointMake(curvePath2.currentPoint.x + 3*(cos-
f(Degree - ((M_PI*9/10) * (self.progress-0.5)*2))),
curvePath2.currentPoint.y - 3*(sinf(Degree - ((M_PI*9/10) *
(self.progress-0.5)*2)))))];
        [curvePath2 appendPath:arrowPath];
    }


    CGContextSaveGState(context);
    CGContextRestoreGState(context);

    [[UIColor blackColor] setStroke];
    [arrowPath  stroke];
    [curvePath1 stroke];
    [curvePath2 stroke];

    UIGraphicsPopContext();
}
```

你仍然可以在我的Github repo —— A-GUIDE-TO-iOS-ANIMATION 下的 AnimatedCurveDemo 找到对应的源码。

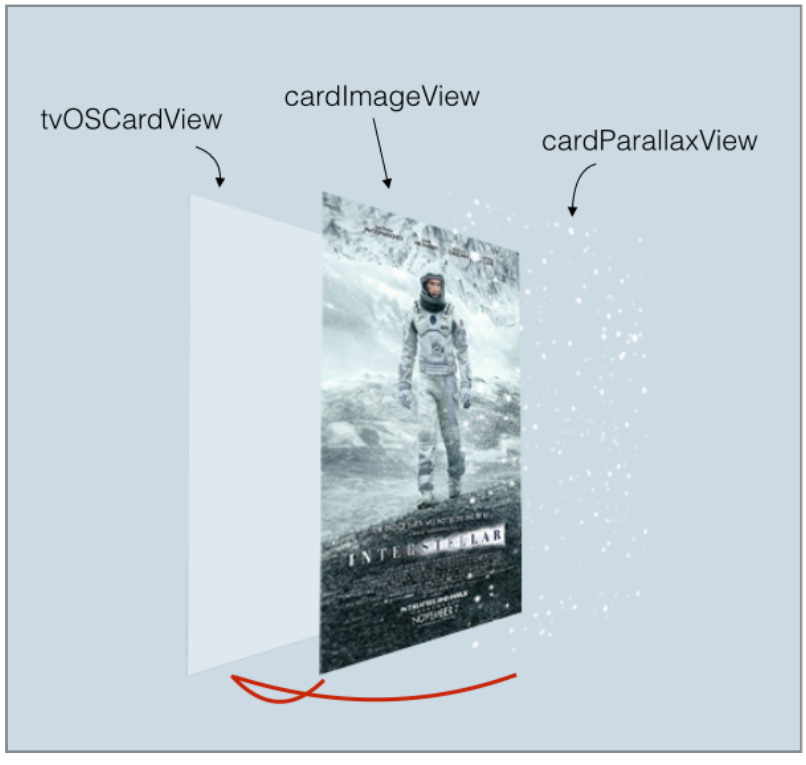
第三个例子是我在发布本册电子书之前临时加的一个例子，也是在看了 9月9日苹果新品发布会之后才有的灵感。

Movie 4.3 模拟 tvOS 中的3D浮动效果



效果请看 **Movie 4.3**。

下面我就带你来实现一下这个效果。首先我要告诉你的是，这个 demo 的代码非常短，也就 100 行左右。其中用到的核心方法就是 CATransform3DRotate。除此之外，没有使用到任何其它关于动画的 API。我会毫无保留地向你展示我的整个思考过程，相信在看完之后，以后你再遇到类似的问题也能举一反三。



首先，我要向你介绍的是整体视图的层级。

tvOSCardView 是我们对外展示的类，其上方有两个 subView —— 分别是 cardImageView 和 cardParallaxView。初始化代码如下：

```
-(void)setUpSomething{

    self.layer.shadowColor = [UIColor blackColor].CGColor;
    self.layer.shadowOffset = CGSizeMake(0, 10);
    self.layer.shadowRadius = 10.0f;
    self.layer.shadowOpacity = 0.3f;


    cardImageView = [[UIImageView
alloc]initWithFrame:self.bounds];
    cardImageView.image = [UIImage imageNamed:@"poster"];
    cardImageView.layer.cornerRadius = 5.0f;
    cardImageView.clipsToBounds = YES;
    [self addSubview:cardImageView];

    UIPanGestureRecognizer *panGes = [[UIPanGestureRecognizer
alloc]initWithTarget:self action:@selector(panInCard:)];
    [self addGestureRecognizer:panGes];


    cardParallaxView = [[UIImageView
alloc]initWithFrame:cardImageView.frame];
    cardParallaxView.image = [UIImage imageNamed:@"5"];
    cardParallaxView.layer.transform =
CATransform3DTranslate(cardParallaxView.layer.transform, 0, 0,
200);
    [self insertSubview:cardParallaxView
aboveSubview:cardImageView];

}
```

为什么需要这样设计这样的视图层级是有原因的。注意到了吗？我们的这个视图带有阴影，但是细心的你又会发现，视图同时还带有圆角，而圆角就必须在设置了 `cornerRadius` 的同时开启 `clipsToBounds` 或 `layer.masksToBounds`。如果此时阴影也是加在这个圆角的视图上，那么

阴影也就会被裁掉。所以这就是为什么我们要把阴影设置在 self.layer 上，然后把圆角设置在 cardImageView 上。

再说说是手势控制了。我先不急给你看代码，我先聊聊一开始我是怎么想的。我想要的效果是手指移到什么地方，什么地方就会「突」起来，并且是绕着视图中点旋转的，而默认 iOS 中的坐标系是以左上角为原点，水平向左为 x 轴正方向，竖直向下为 y 轴正方向。所以可以预感到我们必须以视图中心为原点创建一个虚拟坐标系，以方便后续动画数值的计算。下面给出的代码的作用就是 —— 把系统默认的坐标系统转换成我们虚构的一个坐标系统，这个假设的坐标系以视图中心为坐标系原点，就像下图所展示的那样：

```
-(void)panInCard:(UIPanGestureRecognizer *)panGes{

    CGPoint touchPoint = [panGes locationInView:self];

    if (panGes.state == UIGestureRecognizerStateChanged) {

        CGFloat xFactor = MIN(1, MAX(-1,(touchPoint.x -
(self.bounds.size.width/2)) / (self.bounds.size.width/2)));
        CGFloat yFactor = MIN(1, MAX(-1,(touchPoint.y -
(self.bounds.size.height/2)) / (self.bounds.size.height/2)));

        cardImageView.layer.transform = [self
transformWithM34:1.0/-500 xf:xFactor yf:yFactor];
        cardParallaxView.layer.transform = [self
transformWithM34:1.0/-250 xf:xFactor yf:yFactor];
```

```
    }else if (panGes.state == UIGestureRecognizerStateEnded){

        [UIView animateWithDuration:0.3 animations:^{
            cardImageView.layer.transform = CATransform3DIden-
tity;
            cardParallaxView.layer.transform = CATrans-
form3DIdentity;
        } completion:NULL];
    }
}


-(CATransform3D )transformWithM34:(CGFloat)m34 xf:(CGFloat)xf
yf:(CGFloat)yf{

    CATransform3D t = CATransform3DIdentity;
    t.m34  = m34;
    t = CATransform3DRotate(t, M_PI/9 * xf, 0, 1, 0);
    t = CATransform3DRotate(t, M_PI/9 * yf, -1, 0, 0);

    return t;
}
```

代码中，我通过两个转换公式获得了两个数值 xFactor 和 yFactor，并且取值范围控制在了 [-1,1]，作为我们新建的虚拟坐标系下的坐标点。这样一来视图的左上角、右上角、左下角、右下角就分别被转换成了 (-1,-1)，(1,-1)，(-1,1)，(1,1)。然而为什么是 1,-1 呢？

我们知道 CATransform3D CATransform3DRotate (CATransform3D t, CGFloat angle,CGFloat x, CGFloat y, CGFloat z) 这个方法在给定一个 弧度制 的角度之后，后面的 x,y,z 只需要指定一个任意正数或零或任意负数即可。通常我们使用 -1,0,1。关于苹果规定的 rotate 正方向参见 Gallery 4.2 。

Gallery 4.2 rotation.x/y/z 的正方向

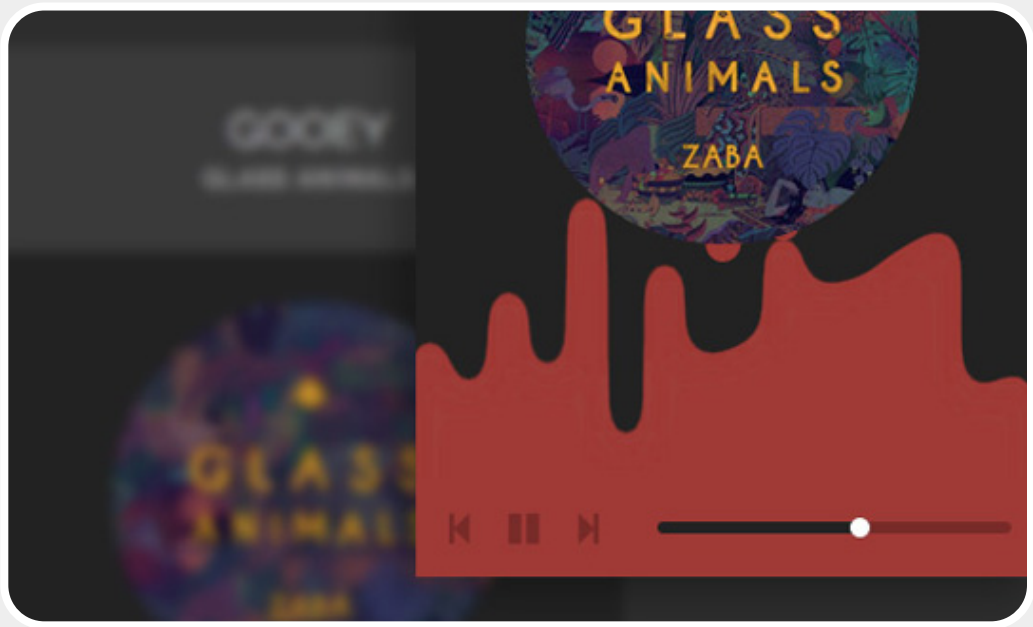
绕 X 轴旋转的正方向

接下来我就开始尝试把这个坐标系统运用到 CATrans-form3DRotate 中。先规定最大偏移角度为 20°，也就是 M_PI/9 。然后我处理的方法依然是分解，把一个多纬度动画拆开为两个分运动逐个分析。我先分析了绕 x 轴旋转的分运动，也就是随着 yFactor 值从 -1 到 1，视图从 -20° 转到 20°。由于绕 x 轴旋转的正方向是上半部分向外，下部分向内，因此一开始的方向为正，与 yFactory 一开始的符号刚好相反，所以把 x 设为 -1 ，即 t = CATransform3DRotate(t, M_PI/9 * yf, -1, 0, 0)。

同理，由于绕 y 轴旋转的正方向是左半部分向外，右半部分向内，而 xFactor 的值是从 -1 到 1 符合绕 y 轴的正方向，所以把 y 设为 1 即可：`t = CATransform3DRotate(t, M_PI/9 * xf, 0, 1, 0)`。

最后我们加了一个回弹动画，让视图在手势结束之后恢复成 CATransform3DIdentity。

到这里，这一节又该结束了，你可以在 **tvOSCardAnimationDemo** 查看源码。



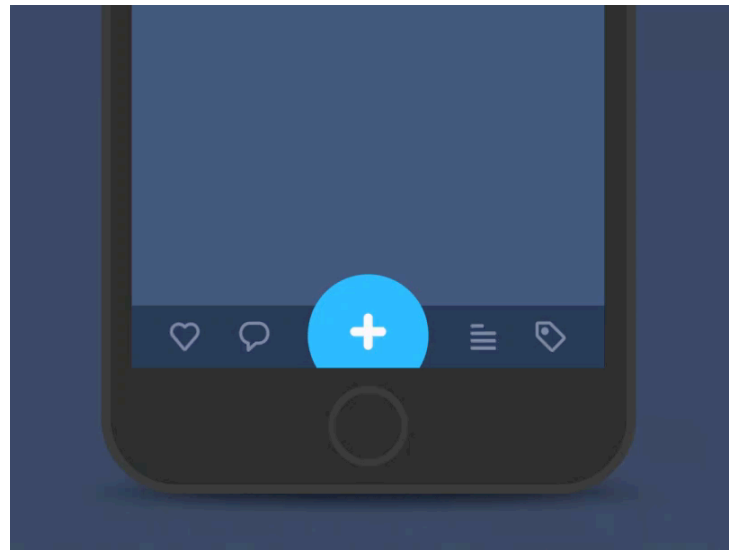
5

自定義屬性動畫

这一节我们将进入一个全新的动画世界。这绝对是一些你未尝涉及过的动画技巧。掌握了这些技巧，你可以实现很多之前望而却步的动效。它就是 —— 自定义属性动画（Custom Property Animation）

比如像 **Movie 5.1** 这种：

Movie 5.1 自定义动画 —— GooeyEffect



事不宜迟，我们赶紧来看看这个效果的实现。

首先，我们创建两个类：`Menu` 和 `MenuLayer`。把 `MenuLayer` 添加到 `Menu` 的 **layer** 上。`Menu` 只负责点击事件、以及作为添加 **item** 的容器，动画的具体实现我们放在 `MenuLayer` 中，这也符合 **UIView** 的 **CALayer** 的先天使命。

在编写这本电子书之前，我就希望我能给带给各位读者的不仅仅只是问题的答案，更应该是解决问题的过程已经我的思考。这才是能帮助你脱离这几个案例依然能自己完成动画的根本方法论。所以，我不想只给你说下一步应该怎么做，我尽量把我自己从第一眼看到这个动画时的心

理活动到最终实现的整个过程呈现给你，希望能对你以后自己实现相应的动画有所借鉴。

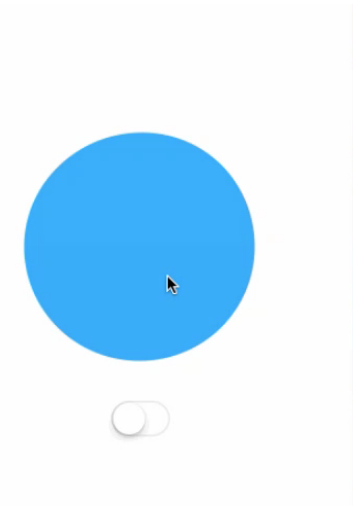
我在第一眼看到这个动效时，坦率地讲，我真的觉得很难实现。但我知道，「天下武功唯快不破」。凡是让人无从下手的动效，很大程度上是因为看不清，也就是动画的播放速度快到让你根本不知道它是怎么动的。那么接下来就很自然地会想到，有什么办法可以让动画慢下来呢？最好是可以手动控制关键帧地进行逐帧播放。如果你有视频剪辑的经验那么你就知道最直接的方法就是拖入 Final Cut Pro 或者 PR 这类剪辑软件中，在时间线进行逐帧浏览。但归功于我是一位重度 Keynote 使用者，我在大学中的每次演讲都是在 Keynote 的帮助下完成的，以至于熟悉到我一直拿它当 PS 用。你之前看到的所有插图都是我在 Keynote 中完成的。然而现在我还要告诉你的是，Keynote 还可以让你逐帧查看视频或 GIF 。

首先你需要将一个视频或 GIF 拖入 Keynote ，然后选中右侧的「影片」板块，通过滑动「标记帧」滑块可以实现逐帧预览。

INTERACTIVE 5.1 演示如何在 Keynote 中逐帧查看视频或 GIF

通过对逐帧图片的慢动作分析，我很快就有了思路：「依然用贝塞尔曲线拼出一个圆，依然通过控制点的运动实现小球的形变」。只是这回关键点的移动规律需要推敲一下。MOVIE 5.2 是我完成的第一版初稿。

MOVIE 5.2 Gooey Effect 第一版



打开控制点可视化后，你会发现其中的秘密：小球的形变其实就是由于顶上三个点的运动产生的。所以核心问题就转化换成了：如何确定控制点的运动轨迹？到这里我们再理一理目前的思路：

> “ 通过 *CAKeyframeAnimation* 改变一个 *layer* 的自定义的属性，在这个 *layer* 的类中监听该属性的变化，一旦该属性变化立即重绘。”

下面我会一边给出代码一边通过警示符号向你做出解释。

```
+(BOOL)needsDisplayForKey:(NSString *)key{
    if ([key isEqualToString:@"xAxisPercent"]) {
        return YES;
    }

    return [super needsDisplayForKey:key];
}
```

```
-(void)drawInContext:(CGContextRef)ctx{

    CGRect real_rect = CGRectInset(self.frame, OFF,OFF);
    CGFloat offset = real_rect.size.width/ 3.6;
    CGPoint center = CGPointMake(CGRectGetMidX(self.frame),
CGRectGetMidY(self.frame));
    CGFloat moveDistance = _xAxisPercent*(offset);
    CGPoint top_left   =
CGPointMake(center.x-offset-moveDistance, OFF);
    CGPoint top_center =  CGPointMake(center.x-moveDistance,
OFF);
    CGPoint top_right  =
CGPointMake(center.x+offset-moveDistance, OFF);

    CGPoint right_top    =  CGPointMake(CGRectGetMaxX(re-
al_rect), center.y-offset);
    CGPoint right_center =  CGPointMake(CGRectGetMaxX(re-
al_rect), center.y);
    CGPoint right_bottom =  CGPointMake(CGRectGetMaxX(re-
al_rect), center.y+offset);
```

```
    CGPoint bottom_left   =  CGPointMake(center.x-offset,
CGRectGetMaxY(real_rect));
    CGPoint bottom_center =  CGPointMake(center.x, CGRectGetMa-
xY(real_rect));
    CGPoint bottom_right  =  CGPointMake(center.x+offset,
CGRectGetMaxY(real_rect));

    CGPoint left_top    =  CGPointMake(OFF, center.y-offset);
    CGPoint left_center =  CGPointMake(OFF, center.y);
    CGPoint left_bottom =  CGPointMake(OFF, center.y+offset);

    UIBezierPath *circlePath = [UIBezierPath bezierPath];
    [circlePath moveToPoint:top_center];
    [circlePath addCurveToPoint:right_center
controlPoint1:top_right controlPoint2:right_top];
    [circlePath addCurveToPoint:bottom_center
controlPoint1:right_bottom controlPoint2:bottom_right];
    [circlePath addCurveToPoint:left_center
controlPoint1:bottom_left controlPoint2:left_bottom];
    [circlePath addCurveToPoint:top_center
controlPoint1:left_top controlPoint2:top_left];
    [circlePath closePath];

    CGContextAddPath(ctx, circlePath.CGPath);
    CGContextSetFillColorWithColor(ctx, [UIColor
colorWithRed:29.0/255.0 green:163.0/255.0 blue:1
alpha:1].CGColor);
    CGContextFillPath(ctx);

}
```

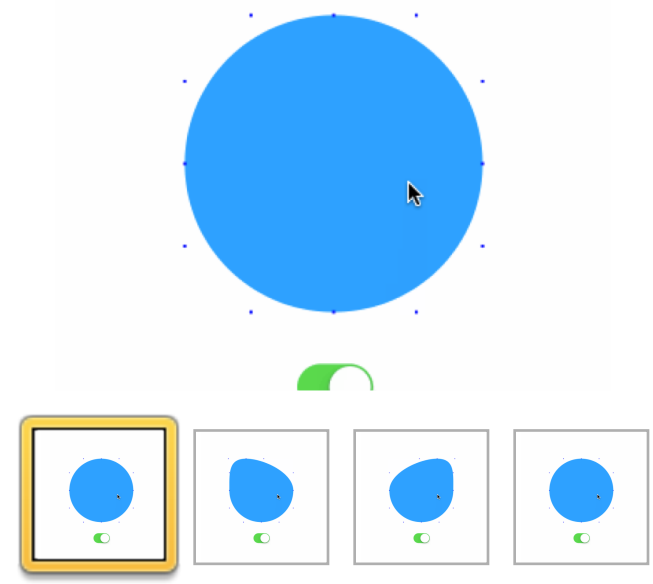
理论上，我们的 MenuLayer 只需要这两段代码就可以了。但我们还需要重载 -(id)initWithLayer:(MenuLayer *)layer 方法拷贝前一个 layer 的相应属性，否则当我们每次调用 -(void)drawInContext:(CGContextRef)ctx 当前 layer 的属性都将是缺省值。

```
-(id)initWithLayer:(MenuLayer *)layer{
    self = [super initWithLayer:layer];
    if (self) {
        //...在这里拷贝layer的所有property
        self.showDebug = layer.showDebug;
        self.xAxisPercent = layer.xAxisPercent;
    }
    return self;
}
```

现在你已经完成了 MenuLayer 中的所有代码。我们回到 Menu 类中，看看应该如何触发 MenuLayer 的重绘以及如何确定控制点的运动轨迹。

关于运动轨迹，思路是分解为 3 步。结合 **Gallery 5.1** 体会。

Gallery 5.1 控制点3个状态下的不同位置



- 第一步：从初始位置开始运动到左侧最远位置，运动距离为小球宽度的1/3.6倍。
- 第二步：从左侧最远位置运动到右侧最远位置。
- 第三步：从右侧最远位置以弹性动画恢复到初始位置。

下面的代码是关于 MenuLayer 的重绘。

```
-(void)openAnimation{
    CAKeyframeAnimation *openAnimation_1 = [[KYSpringLayerAnima-
tion sharedAnimManager]createBasicAnima:@"xAxisPercent"
duration:0.3 fromValue:@(0) toValue:@(1)];
    openAnimation_1.delegate = self;
    [self.menuLayer addAnimation:openAnimation_1
forKey:@"openAnimation_1"];
}
```

我创建了一个 CAKeyframeAnimation 用来刷新 menuLayer 的 @"xAxisPercent" 属性，又因为 menuLayer 对该属性进行了监听，所以 menuLayer 会进行重绘。值得注意的是，我使用的是自定义的动画生成器 —— KYSpringLayerAnimation。这里面的考量在于，通过这个自定义动画生成器，你可以实现各种想要的效果，而所需要的仅仅是一个曲线方程。

下面我们来看看这个动画生成器的源码：

```
-(CAKeyframeAnimation *)createBasicAnima:(NSString *)keypath
duration:(CFTimeInterval)duration fromValue:(id)fromValue
toValue:(id)toValue{

    CAKeyframeAnimation *anim = [CAKeyframeAnimation
animationWithKeyPath:keypath];
    anim.values = [self basicAnimationValues:fromValue
toValue:toValue duration:duration];
    anim.duration = duration;
    anim.fillMode = kCAFillModeForwards;
    anim.removedOnCompletion = NO;

    return anim;
}
```

```
-(NSMutableArray *) basicAnimationValues:(id)fromValue
toValue:(id)toValue duration:(CGFloat)duration{

    NSInteger numOfFrames = duration * 60;
    NSMutableArray *values = [NSMutableArray
arrayWithCapacity:numOfFrames];
    for (NSInteger i = 0; i < numOfFrames; i++) {
        [values addObject:@(0.0)];
    }

    CGFloat diff = [toValue floatValue] - [fromValue float-
Value];
    for (NSInteger frame = 0; frame<numOfFrames; frame++) {

        CGFloat x = (CGFloat)frame / (CGFloat)numOfFrames;
        CGFloat value = [fromValue floatValue] + diff * x;
        values[frame] = @(value);
    }
    return values;
}
```

核心代码就是 anim.values = [self basicAnimationValues:fromValue toValue:toValue duration:duration]; 从而产生 60 个数值。为什么是 60 个？因为 iOS 设备的刷新频率就是 60HZ，也就是说，想要达到流畅细腻的动画，60 个关健帧就足够了。然后我们来看看这 60 个数值是如何被创造出来的。

```
-(NSMutableArray *) basicAnimationValues:(id)fromValue
toValue:(id)toValue duration:(CGFloat)duration{

    NSInteger numOfFrames = duration * 60;
    NSMutableArray *values = [NSMutableArray
arrayWithCapacity:numOfFrames];
    for (NSInteger i = 0; i < numOfFrames; i++) {
```

```
        [values addObject:@(0.0)];
    }

    CGFloat diff = [toValue floatValue] - [fromValue floatVal-
ue];
    for (NSInteger frame = 0; frame<numOfFrames; frame++) {
        CGFloat x = (CGFloat)frame / (CGFloat)numOfFrames;
        CGFloat value = [fromValue floatValue] + diff * x;
        values[frame] = @(value);
    }
    return values;
}
```

现在你点击 Menu 视图调用 -(void)openAnimation 方法就能看到运动轨迹的第一步了。同理，第二步到第三步也是一样，无非就是再创建一个 CAKeyframeAnimation。

```
CAKeyframeAnimation *openAnimation_2 = [[KYSpringLayerAnimation
sharedAnimManager]createBasicAnima:@"xAxisPercent" duration:0.3
fromValue:@(1) toValue:@(-1)];
```

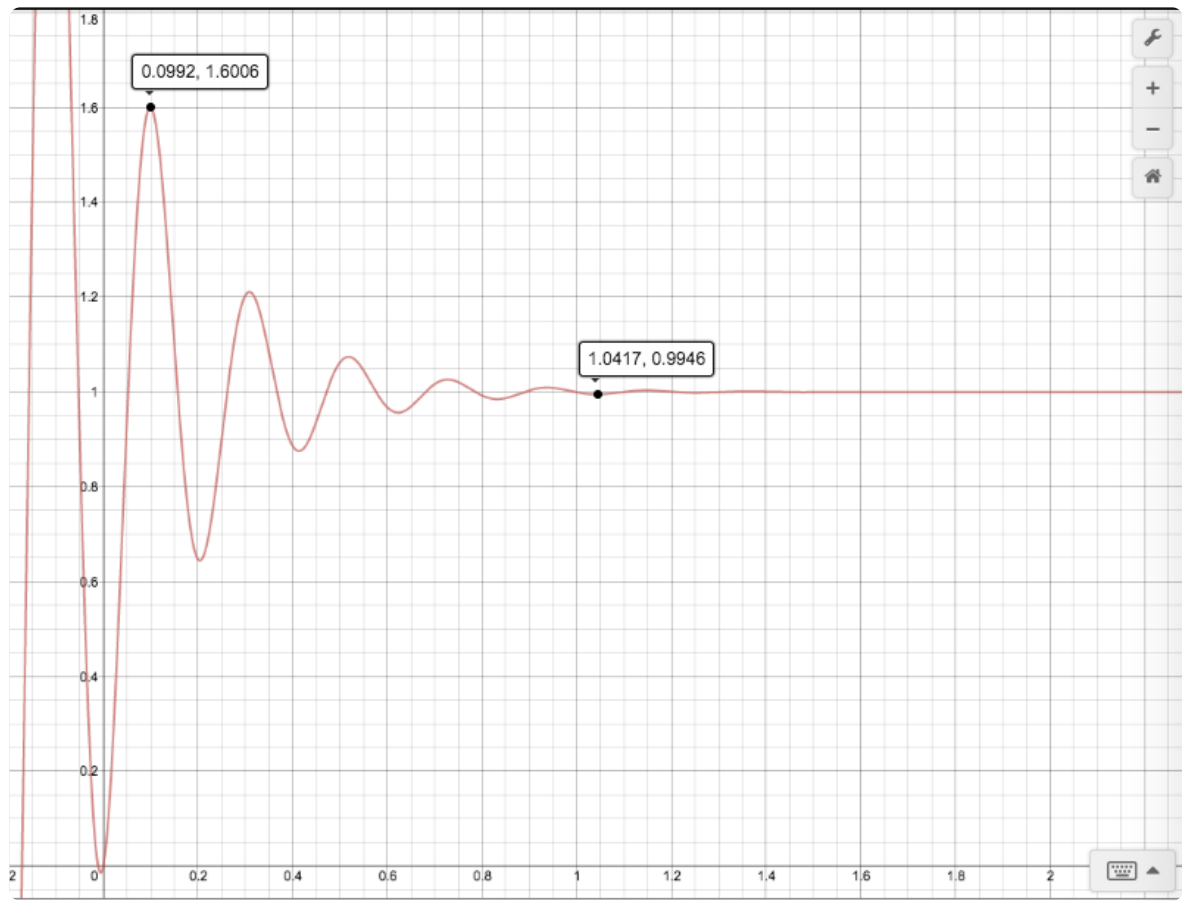
在第一步动画 animationDidStop 之后再添加第二步的动画。通过自定义动画生成器的优势在以上两步中并没有很好体现，因为我们只是使用了一个简单的一次函数，完全可以用 kCAMediaTimingFunctionLin-ear 这些原生曲线代替。但马上你就会用它创建一个弹性动画，你就能体会到它的优势了。

第三步，就是一个弹性动画。唯一要做的只是传入一个震荡方程。在 KYSpringLayerAnimation 中做如下处理：

```
-(CAKeyframeAnimation *)createSpringAnima:(NSString *)keypath
duration:(CFTimeInterval)duration
usingSpringWithDamping:(CGFloat)damping
initialSpringVelocity:(CGFloat)velocity fromValue:(id)fromValue
toValue:(id)toValue{

    CGFloat dampingFactor  = 10.0;
    CGFloat velocityFactor = 10.0;
    NSMutableArray *values = [self
springAnimationValues:fromValue toValue:toValue
usingSpringWithDamping:damping * dampingFactor
initialSpringVelocity:velocity * velocityFactor
duration:duration];

    CAKeyframeAnimation *anim = [CAKeyframeAnimation
animationWithKeyPath:keypath];
    anim.values = values;
    anim.duration = duration;
    anim.fillMode = kCAFillModeForwards;
    anim.removedOnCompletion = NO;

    return anim;
}


-(NSMutableArray *) springAnimationValues:(id)fromValue
toValue:(id)toValue usingSpringWithDamping:(CGFloat)damping
initialSpringVelocity:(CGFloat)velocity
duration:(CGFloat)duration{

    //60个关键帧
    NSInteger numOfFrames  = duration * 60;
    NSMutableArray *values = [NSMutableArray
arrayWithCapacity:numOfFrames];
    for (NSInteger i = 0; i < numOfFrames; i++) {
        [values addObject:@(0.0)];
    }
```

```
    //差值
    CGFloat diff = [toValue floatValue] - [fromValue floatValue];

    for (NSInteger frame = 0; frame<numOfFrames; frame++) {

        CGFloat x = (CGFloat)frame / (CGFloat)numOfFrames;
        CGFloat value = [toValue floatValue] - diff * (pow(M_E, -damping * x) * cos(velocity * x)); // y = 1-e^{-5x} * cos(30x)

        values[frame] = @(value);
    }

    return values;

}
```

这里传入的是一个 **Image 5.1** $y = 1 - e^{-5x} \cdot \cos(30x)$ 的图像震荡函数：$1 - e^{-5x} \cdot \cos(30x)$，如 **Image 5.1** 所示，从而获得了一组震荡变化的值。想要了解更多关于阻尼振动的知识，可以点击这里查看维基百科。



当然，此处必须分享方便查看函数曲线的工具，比如这个 网站 。

接下来使用的方法还是和之前一样，即创建一个 CAKeyframeAnimation ：

```
    CAKeyframeAnimation *openAnimation_3 = [[KYSpringLayerAnimation sharedAnimManager]createSpringAnima:@"xAxisPercent" duration:1.0 usingSpringWithDamping:0.5 initialSpringVelocity:3.0 fromValue:@(-1) toValue:@(0)];
```

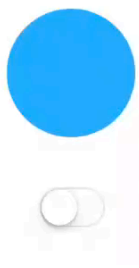
然后在第二步动画的 animationDidStop 之后添加这个动画。

以上，三步动画连起来就有了一开始视频中的效果。

其实，我们还可以使用 之前 提到的使用辅助视图的方法。我们可以通过控制三个辅助的 UIView 并和上面三个小点以相同的运动轨迹运动，通过 UIView 运动过程中的 CALayer 的 Presentation Layer 获取运动过程中实时的位置，也可以画出小球的形状。

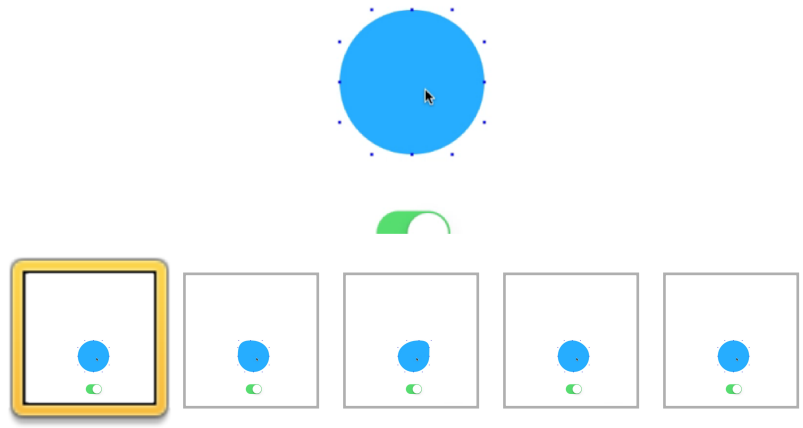
IMAGE 5.2 是我改进之后的效果。

IMAGE 5.2 GooeyMenu 改进效果



你可能注意到了，我仅仅改变了三个控制点的运动规律。实际上，任何形状动画都可以用控制贝塞尔曲线控制点的方法来做到。在改进的例子中，无非就是这样三个状态 。

GALLERY 5.2 第三个状态使用弹性动画



最后完整的代码如下：

◉ "Menu.m"

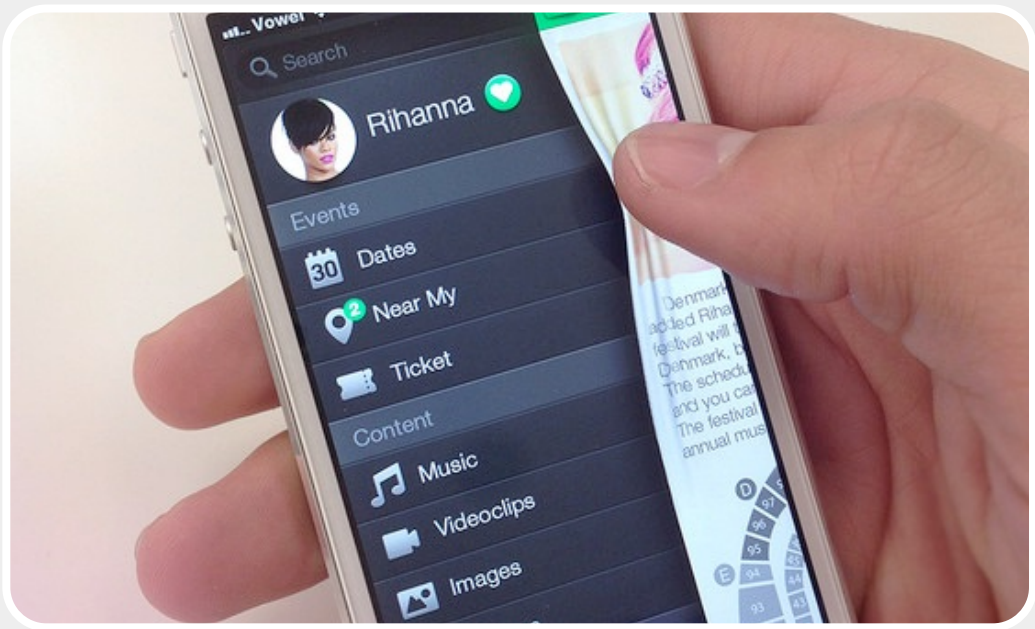
```
    CAKeyframeAnimation *openAnimation_1 = [[KYSpringLayerAnima-
tion sharedAnimManager]createBasicAnima:@"xAxisPercent"
duration:0.3 fromValue:@(0) toValue:@(1)];
    openAnimation_1.delegate = self;
    CAKeyframeAnimation *openAnimation_2 = [[KYSpringLayerAnima-
tion sharedAnimManager]createBasicAnima:@"xAxisPercent"
duration:0.3 fromValue:@(0) toValue:@(1)];
    openAnimation_2.delegate = self;
    CAKeyframeAnimation *openAnimation_3 = [[KYSpringLayerAnima-
tion sharedAnimManager]createSpringAnima:@"xAxisPercent"
duration:1.0 usingSpringWithDamping:0.5
initialSpringVelocity:3.0 fromValue:@(0) toValue:@(1)];
    openAnimation_3.delegate = self;
    [_animationQueue addObject:openAnimation_1];
    [_animationQueue addObject:openAnimation_2];
    [_animationQueue addObject:openAnimation_3];
    [self.menuLayer addAnimation:openAnimation_1
forKey:@"openAnimation_1"];
    self.userInteractionEnabled = NO;
    _menuLayer.animState = STATE1;
```

◉ "MenuLayer.m"

```
-(void)drawInContext:(CGContextRef)ctx{

    CGRect real_rect = CGRectInset(self.frame, OFF,OFF);
    CGFloat offset = real_rect.size.width/ 3.6;
    CGPoint center = CGPointMake(CGRectGetMidX(self.frame),
CGRectGetMidY(self.frame));
    CGFloat moveDistance_1;
    CGFloat moveDistance_2;
    CGPoint top_left;
    CGPoint top_center;
    CGPoint top_right;

    if (_animState == STATE1) {

        
        moveDistance_1 = _xAxisPercent*(real_rect.size.width/2
- offset)/2;
        top_left   =
CGPointMake(center.x-offset-moveDistance_1*2, OFF);
        top_center =  CGPointMake(center.x-moveDistance_1,
OFF);
        top_right  =  CGPointMake(center.x+offset, OFF);

    }else if(_animState == STATE2){

      
        CGFloat hightFactor;
        if (_xAxisPercent >= 0.2) {
            hightFactor = 1-_xAxisPercent;
        }else{
            hightFactor = _xAxisPercent;
        }
        moveDistance_1 = (real_rect.size.width/2 - offset)/2;
        moveDistance_2 =
_xAxisPercent*(real_rect.size.width/3);
        top_left   =
CGPointMake(center.x-offset-moveDistance_1*2 + moveDistance_2,
OFF);
```

```
        top_center =  CGPointMake(center.x-moveDistance_1 +
moveDistance_2, OFF);
        top_right  =
CGPointMake(center.x+offset+moveDistance_2, OFF);

    }else if(_animState == STATE3){

        moveDistance_1 = (real_rect.size.width/2 - offset)/2;
        moveDistance_2 = (real_rect.size.width/3);
        CGFloat gooeyDis_1 =
_xAxisPercent*(center.x-offset-moveDistance_1*2 +
moveDistance_2-(center.x-offset));
        CGFloat gooeyDis_2 =
_xAxisPercent*(center.x-moveDistance_1 +
moveDistance_2-(center.x));
        CGFloat gooeyDis_3 =
_xAxisPercent*(center.x+offset+moveDistance_2-(center.x+offset)
);

        top_left   =
CGPointMake(center.x-offset-moveDistance_1*2 + moveDistance_2 -
gooeyDis_1, OFF);
        top_center =  CGPointMake(center.x-moveDistance_1 +
moveDistance_2 - gooeyDis_2, OFF);
        top_right  =
CGPointMake(center.x+offset+moveDistance_2 - gooeyDis_3, OFF);

    }


    //和之前的代码一样
    .....

}
```

自定义动画的这一章就先到这里了。你可以在 **KYGooeyMenuDemo** 中找到本节的源码。


Search
Rihanna
Events
Dates
Near My
Ticket
Content
Music
Videoclips
Images

6

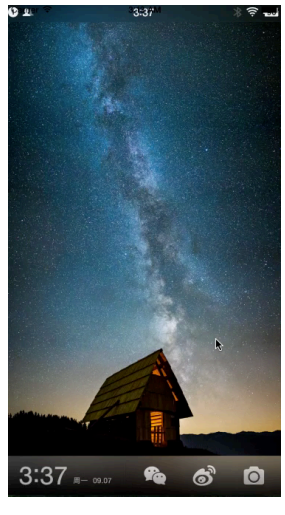
其他效果

这一章节的内容，属于一些我暂时无法归类但都非常有趣的玩意儿。比如我会在这一章中介绍一些动画的小技巧，或者是一些使用得比较少的 iOS SDK 中的动画类。

本章第一节，我打算先介绍的是 —— UIKitDynamics 。可以说 UIKitDynamics 这个 iOS7 中引入的类平常用的并不算多，但是它所展现的视觉效果一定是让你印象深刻的。

下面我通过一个模拟 Smartisan OS 中锁屏界面的 demo 给你介绍一下 UIKitDynamics 的使用方法。具体效果请见 **Movie 6.1** 。

Movie 6.1 模拟 Smartisan OS 锁屏界面



在这个效果中，我一共使用了五个 Behavior：

- UIGravityBehavior
- UIPushBehavior

- UIAttachmentBehavior
- UIDynamicItemBehavior
- UICollisionBehavior

以及一个物理引擎的容器 UIDynamicAnimator，所有 Behavior 都要加进 UIDynamicAnimator 中才能奏效。

首先我们先来实现点击一次发生弹跳的效果。准备工作：设置好封面图，绑定一个单击手势，准备好所需的 Behavior 以及初始化物理引擎容器 UIDynamicAnimator：

```
- (void)viewDidLoad {

    [super viewDidLoad];
    self.lockScreenView = [[UIImageView
alloc]initWithFrame:self.view.bounds];
    self.lockScreenView.image = [UIImage
imageNamed:@"lockScreen"];
    self.lockScreenView.contentMode = UIViewContentModeS-
caleToFill;
    self.lockScreenView.userInteractionEnabled = YES;
    [self.view addSubview:_lockScreenView];

    UITapGestureRecognizer *tap = [[UITapGestureRecognizer
alloc]initWithTarget:self action:@selector(tapOnIt:)];
    [self.lockScreenView addGestureRecognizer:tap];


}

-(void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];
```

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

    UICollisionBehavior *collisionBehaviour = [[UICollisionBe-
havior alloc]initWithItems:@[self.lockScreenView]];

    
    [collisionBehaviour
setTranslatesReferenceBoundsIntoBoundaryWithInsets:UIEdgeInsets
Make(-_lockScreenView.frame.size.height, 0, 0, 0)];
    [self.animator addBehavior:collisionBehaviour];

    
    self.gravityBehaviour = [[UIGravityBehavior alloc]
initWithItems:@[self.lockScreenView]];
    self.gravityBehaviour.gravityDirection = CGVectorMake(0.0,
1.0);
    self.gravityBehaviour.magnitude = 2.6f;
    [self.animator addBehavior:self.gravityBehaviour];

    
    self.pushBehavior = [[UIPushBehavior alloc]
initWithItems:@[self.lockScreenView]
mode:UIPushBehaviorModeInstantaneous];
    self.pushBehavior.magnitude = 2.0f;
    self.pushBehavior.angle = M_PI;
    [self.animator addBehavior:self.pushBehavior];

    
    self.itemBehaviour = [[UIDynamicItemBehavior alloc]
initWithItems:@[self.lockScreenView]];
    self.itemBehaviour.elasticity = 0.35f;

    
    [self.animator addBehavior:_itemBehaviour];

}
```

现在，我们处理点击的手势，点击一次触发一个 pushBehavior，并设置它的 pushDirection 为竖直向上的 CGVectorMake(0.0f, -80.0f)。

```
-(void)tapOnIt:(UITapGestureRecognizer *)tapGes{
    self.pushBehavior.pushDirection = CGVectorMake(0.0f, -80.0f);
    
    self.pushBehavior.active = YES;
}
```

接着我们来看看如何实现手势拖拽。首先处理 UIPanGestureRecognizer。

```
-(void)panOnIt:(UIPanGestureRecognizer *)panGes{

    CGPoint location = CGPointMake(CGRectGetMidX(_lockScreenView.frame), [panGes locationInView:self.view].y);

    if (panGes.state == UIGestureRecognizerStateBegan) {

        [self.animator removeBehavior:self.gravityBehaviour];
        self.attachmentBehaviour = [[UIAttachmentBehavior alloc]initWithItem:self.lockScreenView attachedToAnchor:location];
        [self.animator addBehavior:_attachmentBehaviour];

    }else if (panGes.state == UIGestureRecognizerStateChanged){

        self.attachmentBehaviour.anchorPoint = location;

    }else if (panGes.state == UIGestureRecognizerStateEnded){
```

```
        CGPoint velocity = [panGes
velocityInView:_lockScreenView];
        NSLog(@"v:%@",NSStringFromCGPoint(velocity));

        [self.animator removeBehavior:_attachmentBehaviour];
        _attachmentBehaviour = nil;

        if (velocity.y < -1300.0f) {
            [self.animator removeBehavior:_gravityBehaviour];
            [self.animator removeBehavior:_itemBehaviour];
            _gravityBehaviour = nil;
            _itemBehaviour = nil;

            //gravity
            self.gravityBehaviour = [[UIGravityBehavior alloc]
initWithItems:@[self.lockScreenView]];
            self.gravityBehaviour.gravityDirection =
CGVectorMake(0.0, -1.0);
            self.gravityBehaviour.magnitude = 2.6f;
            [self.animator addBehavior:self.gravityBehaviour];

            //item
            self.itemBehaviour = [[UIDynamicItemBehavior alloc]
initWithItems:@[self.lockScreenView]];
            self.itemBehaviour.elasticity = 0.0f;//1.0 完全弹性碰
撞，需要非常久才能恢复;
            [self.animator addBehavior:_itemBehaviour];

            self.pushBehavior.pushDirection =
CGVectorMake(0.0f, -200.0f);
            self.pushBehavior.active = YES;
        }else{
            [self restore];
        }
    }
}
```

UIAttachmentBehavior，从这个名字就可以看出这个动作是有附着、附加的效果。其实就是视图的锚点（anchorPoint）始终和某个指定

的点保持一段length 距离的效果。这种连接可以是两个 item 之间的连接（两个 item 的 anchorPoint 之间的距离），也可以是一个 item 和一个点之间的连接。其他一些属性：

```
两个点之间的距离：
@property (readwrite, nonatomic) CGFloat length;

阻尼系数（0 ~ 1 之间；默认为 0）：
@property (readwrite, nonatomic) CGFloat damping; // 1: critical damping

弹性系数（值越大 item 的弹性效果就越明显）：
@property (readwrite, nonatomic) CGFloat frequency; // in Hertz
```

通过这些属性，你可以想象成两个点之间是通过一条「木棍」刚性连接着，或者是一根「橡皮筋」弹性连接着。对于这个 demo，首先我们在手势开始时先确定这个目标点：

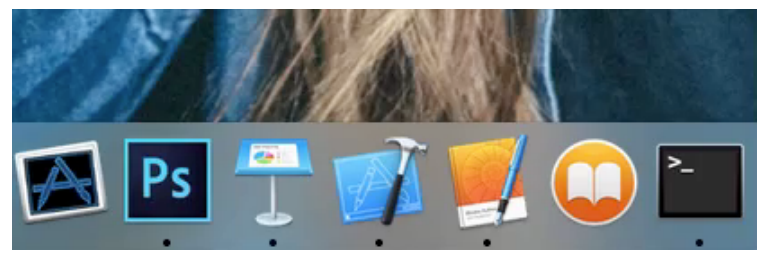
```
CGPoint location =
CGPointMake(CGRectGetMidX(_lockScreenView.frame), [panGes
locationInView:self.view].y);
```

然后创建动作。因为这个 demo 中 item 需要跟手，所以我们不需要弹性也不需要阻尼，都使用默认值。然后让 item 的 anchorPoint 跟着这个目标点，想象成两者之间连着一根「木棍」。

最后在手势结束时根据移动的速度判断是否成功。

```
CGPoint velocity = [panGes velocityInView:_lockScreenView];
```

好了，这个简单的 UIDynamicKit 的 demo 就到这里了。源码参见 UIDynamicsDemo 。除了 UISnapBehavior 之外，这个 demo 基本涉及了所有的动作。类似的弹跳效果还出现在了 Mac OS X 的 dock 中。



至于 UISnapBehavior（吸附动作）其实是所有动作中最简单的。因为它只有这一个方法和一个属性：

```
- (instancetype)initWithItem:(id <UIDynamicItem>)item
snapToPoint:(CGPoint)point;

@property (nonatomic, assign) CGFloat damping; // damping value
from 0.0 to 1.0. 0.0 is the least oscillation.
```

绑定一个 item ，指定一个吸附的目标点，设置一个弹性系数，最后别忘了加到 UIDynamicAnimator 中，一切工作就结束了。然后就只需要静静地看效果(Movie 6.2)。

Movie 6.2
UISnapBehaior 吸附效果的的简单演示

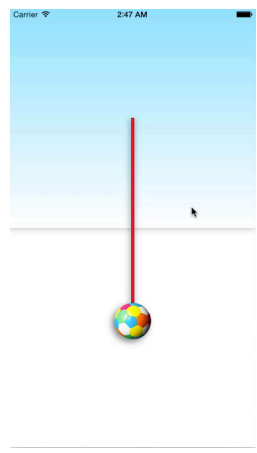
接下去我们来玩点有意思的。我特意抽出这一节向你介绍 UIKitDynamics 中一个神奇的特性 —— Action Block.

```
// When running, the dynamic animator calls the action block on
every animation step.
@property (nonatomic,copy) void (^action)(void);
```

任何一个 UIDynamicBehavior 都能够实时调用这个 action 。举个例子，比如你可以实现下面这些效果：

Movie 6.3 UIDynamicBehavior 中 action 的效果示范01

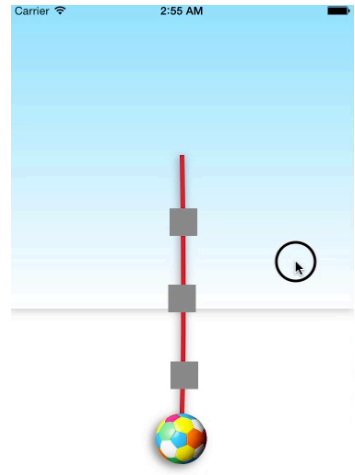
Movie 6.4 UIDynamicBehavior 中 action 的效果示范02



下面我就拿第二个例子作为本章的第二个 demo 。首先一起来看下原理视频：

这个动画的思路其实很简单：给最上方的矩形添加 UIGravityBehavior 和 UICollisionBehavior，在其下方添加三个 UIAttachmentBehavior，分别连接着下方最近的对象。同时，我们把第一个灰块、第三个灰块以及小球这三个视图绑定到同一个 UIGravityBehavior 上面，叫做 viewsGravity，在 viewsGravity 的 action block 中，把第一个灰块、第三个灰块作为一条贝塞尔曲线的两个控制点，实时地绘制出这条曲线，也就是你看到的「绳子」的样子了。

Movie 6.5 UIDynamicBehavior 中 action 的效果示范02 -- 原理解析



然后详细看一下代码细节：

```
- (void)viewDidLoad {
    [super viewDidLoad];

    _panView=[[UIView alloc] initWithFrame:CGRectMake(0, 0,
[[UIScreen mainScreen] bounds].size.width, [[UIScreen main-
Screen] bounds].size.height/2)];
    [_panView setAlpha:0.5];
    [self.view addSubview:_panView];
    [_panView.layer setShadowOffset:CGSizeMake(-1, 2)];
    [_panView.layer setShadowOpacity:0.5];
    [_panView.layer setShadowRadius:5.0];
    [_panView.layer setShadowColor:[UIColor
blackColor].CGColor];
```

```
    [_panView.layer setMasksToBounds:NO];
    [_panView.layer setShadowPath:[UIBezierPath
bezierPathWithRect:_panView.bounds].CGPath];

    UIPanGestureRecognizer *pan=[[UIPanGestureRecognizer alloc]
initWithTarget:self action:@selector(PanTheView:)];
    [_panView addGestureRecognizer:pan];

    CAGradientLayer *grd=[[CAGradientLayer alloc] init];
    [grd setFrame:_panView.frame];
    grd.colors = [[NSArray alloc] initWithObjects:(__bridge
id)([UIColor colorWithRed:0.0 green:191.0/255.0
blue:255.0/255.0 alpha:1].CGColor),(__bridge id)([UIColor
whiteColor].CGColor), nil];
    [_panView.layer addSublayer:grd];

    _ballImageView=[[UIImageView alloc]
initWithFrame:CGRectMake(([[UIScreen mainScreen]
bounds].size.width/2)-30, [[UIScreen mainScreen]
bounds].size.height/1.5, 60, 60)];

    [_ballImageView setImage:[UIImage imageNamed:@"ball"]];
    [self.view addSubview:_ballImageView];
    [_ballImageView.layer setShadowOffset:CGSizeMake(-4, 4)];
    [_ballImageView.layer setShadowOpacity:0.5];
    [_ballImageView.layer setShadowRadius:5.0];
    [_ballImageView.layer setShadowColor:[UIColor
blackColor].CGColor];
    [_ballImageView.layer setMasksToBounds:NO];


    //1
    _middleView = [[UIView alloc]
initWithFrame:CGRectMake(_ballImageView.center.x-15, 200, 30,
30)];
    [_middleView setBackgroundColor:[UIColor grayColor]];
    [self.view addSubview:_middleView];
    [_middleView setCenter:CGPointMake(_middleView.center.x,
(_ballImageView.center.y-_panView.center.y)+15)];

    //2
```

```
    _topView = [[UIView alloc]
initWithFrame:CGRectMake(_ballImageView.center.x-15, 200, 30,
30)];
    [_topView setBackgroundColor:[UIColor grayColor]];
    [self.view addSubview:_topView];
    [_topView setCenter:CGPointMake(_topView.center.x,
(_middleView.center.y-_panView.center.y)+_panView.center.y/2)];


    //3
    _bottomView = [[UIView alloc]
initWithFrame:CGRectMake(_ballImageView.center.x-15, 200, 30,
30)];
    [_bottomView setBackgroundColor:[UIColor grayColor]];
    [self.view addSubview:_bottomView];
    [_bottomView setCenter:CGPointMake(_bottomView.center.x,
(_middleView.center.y-_panView.center.y)+_panView.center.y*1.5)
];

    [self setUpBehaviors];

}
```

代码虽然多但就是简单的布局而已。唯一需要一提的就是 CAGradientLayer 。这是 <QuartzCore/CALayer.h> 中的一个类，估计平时接触的机会也比较少。其实这个类实现的渐变色效果非常有用，有了它你不再需要设计师提供的渐变色的切图了，一个常用的场景就是配图的文字背景。

```
@property(copy) NSArray *colors;  //定义需要显示的颜色，从上到下。
```

- @property(copy) NSArray *locations; //定义颜色的区间分布，比如三种颜色对应三个区间@[@(0.0),@(0.4),@(1.0)],注意必须实在 0~1 之间分布。默认是nil，会平均分布。

- @property CGPoint startPoint;

- @property CGPoint endPoint; //用来定义颜色渐变的方向。[0,0] 是左下角，[1,1] 是右上角。默认是[.5,0] 和 [.5,1]。

下面来看一下如何初始化 UIKitDynamics。

```
-(void)setUpBehaviors{

    _animator=[[UIDynamicAnimator alloc]
initWithReferenceView:self.view];
    _panGravity=[[UIGravityBehavior alloc]
initWithItems:@[_panView]];
    [_animator addBehavior:_panGravity];

    _viewsGravity=[[UIGravityBehavior alloc]
initWithItems:@[_ballImageView,_topView,_bottomView]];
    [_animator addBehavior:_viewsGravity];

    __weak ViewController *weakSelf = self;
    _viewsGravity.action=^{
        NSLog(@"acting");
        UIBezierPath *path=[[UIBezierPath alloc] init];
        [path moveToPoint:weakSelf.panView.center];
        [path addCurveToPoint:weakSelf.ballImageView.center
controlPoint1:weakSelf.topView.center
controlPoint2:weakSelf.bottomView.center];

        if (!weakSelf.layer) {
            weakSelf.layer=[[CAShapeLayer alloc] init];
            weakSelf.layer.fillColor = [UIColor
clearColor].CGColor;
            weakSelf.layer.strokeColor = [UIColor
colorWithRed:224.0/255.0 green:0.0/255.0 blue:35.0/255.0
alpha:1.0].CGColor;
```

```
            weakSelf.layer.lineWidth = 5.0;

            //Shadow
            [weakSelf.layer setShadowOffset:CGSizeMake(-1, 2)];
            [weakSelf.layer setShadowOpacity:0.5];
            [weakSelf.layer setShadowRadius:5.0];
            [weakSelf.layer setShadowColor:[UIColor
blackColor].CGColor];
            [weakSelf.layer setMasksToBounds:NO];

            [weakSelf.view.layer insertSublayer:weakSelf.layer
below:weakSelf.ballImageView.layer];

        }
        weakSelf.layer.path=path.CGPath;
    };

    //UICollisionBehavior
    UICollisionBehavior *Collision=[[UICollisionBehavior alloc]
initWithItems:@[_panView]];
    [Collision addBoundaryWithIdentifier:@"Left"
fromPoint:CGPointMake(-1, 0) toPoint:CGPointMake(-1, [[UIScreen
mainScreen] bounds].size.height)];
    [Collision addBoundaryWithIdentifier:@"Right"
fromPoint:CGPointMake([[UIScreen mainScreen]
bounds].size.width+1,0) toPoint:CGPointMake([[UIScreen main-
Screen] bounds].size.width+1, [[UIScreen mainScreen]
bounds].size.height)];
    [Collision addBoundaryWithIdentifier:@"Middle"
fromPoint:CGPointMake(0, [[UIScreen mainScreen]
bounds].size.height/2) toPoint:CGPointMake([[UIScreen main-
Screen] bounds].size.width, [[UIScreen mainScreen]
bounds].size.height/2)];
    [_animator addBehavior:Collision];


    //3 UIAttachmentBehaviors
   UIAttachmentBehavior *attach1=[[UIAttachmentBehavior
alloc]initWithItem:_panView attachedToItem:_topView];
    [_animator addBehavior:attach1];

    UIAttachmentBehavior *attach2=[[UIAttachmentBehavior alloc]
initWithItem:_topView attachedToItem:_bottomView];
```

```
    [_animator addBehavior:attach2];

    UIAttachmentBehavior *attach3=[[UIAttachmentBehavior alloc]
initWithItem:_bottomView offsetFromCenter:UIOffsetMake(0, 0)
attachedToItem:_ballImageView offsetFromCenter:UIOffsetMake(0,
-_ballImageView.bounds.size.height/2)];
    [_animator addBehavior:attach3];


    //UIDynamicItemBehavior
    UIDynamicItemBehavior *PanItem=[[UIDynamicItemBehavior al-
loc]
initWithItems:@[_panView,_topView,_bottomView,_ballImageView]];
    PanItem.elasticity=0.5;
    [_animator addBehavior:PanItem];

}
```

我们把 _panView 单独绑定一个 UIGravityBehavior，把 @[_ballImageView,_topView,_bottomView] 也绑定到一个 UIGravityBehavior 上。然后设置一个作用于 _panView 的 UICollisionBehavior。下面是关键，我们创建三个 UIAttachmentBehavior 分别让上一个视图 attach 下方相邻的视图，想象成「一环扣一环」的样子。

有个细节需要提一下，就是这里初始化第三个 UIAttachmentBehavior 的时候用到了

```
-(instancetype)initWithItem:(id <UIDynamicItem>)item1
 offsetFromCenter:(UIOffset)offset1 attachedToItem:(id <UIDy-
 namicItem>)item2 offsetFromCenter:(UIOffset)offset2;
```

这个方法。因为我们希望小球可以绕顶点转起来，所以把小球上的锚点设置在了小球的的顶端，而不是小球默认的正中心，否则你将看不到小球的转动。

接下来是必不可少的一行代码，在 _panView 的手势绑定方法中，我们需要实时调用这句：

```
    [_animator updateItemUsingCurrentState:pan.view];
```

我们来看一下官方文档是怎么说的：

```
// Update the item state in the animator if an external change
was made to this item
-(void)updateItemUsingCurrentState:(id <UIDynamicItem>)item;
```

Asks a dynamic animator to read the current state of a dynamic item, replacing the animator's internal representation of the item's state.A dynamic animator automatically reads the initial state (position and rotation) of each dynamic item you add to it, and then takes responsibility for updating the item's state. If you actively change the state of a dynamic item after you've added it to a dynamic animator, call this method to ask the animator to read and incorporate the new state.

文档中说，当外界变化（比如 _panGravity 开始作用了）作用于 pan.View 上时，去刷新当前 _animator 中所有 item 的 position 和 rotation。而这些 item 包括着 _ballImageView,_topView,_bottomView。 po-

sition 和 rotation 的变化又会触发 _ballImageView,_topView,_bottom-View 这三者绑定的_viewsGravity.action. 在这个 action 中实时绘制一条贝塞尔曲线，就有了你看到的一条弹性的绳子。由于这个 action 会在动画的每一步都会调用（on every animation step），所以动画会显得相当流畅。

这个 demo 主要介绍了 action 的作用以及 updateItemUsingCurrent-State 方法的使用。掌握了这个技巧，你可以创造出更多神奇的物理特效。源码详见 DynamicActionBlockDemo。

第三小节，我们来聊聊另一个非常具有视觉冲击力的效果 —— 粒子效果（ CAEmitterLayer）。在 iOS 5 中，苹果引入了一个新的 CALayer 子类 —— CAEmitterLay-er。CAEmitterLayer 是一个高性能的粒子引擎，被用来创建复杂的粒子动画如：烟雾，火，雨等效果，并且很好地控制了性能。

关于 CAEmitterLayer，*iOS Core Animation: Advanced Techniques* 中给出了解释：

“

CAEmitterLayer 看上去像是许多 CAEmitterCell 的容器，这些 CAEmitterCell 定义了一个例子效果。你将会为不同的例子效果定义一个或多个 CAEmitterCell 作为模版，同时 CAEmitterLayer 负责基于这些模版实例化一个粒子流。一个 CAEmitterCell 类似于一个 CALayer ： 它有一个 contents 属性可以定义为一个 CGImage ，另外还有一些可设置属性控制着表现和行为。

”

下面我们深入到头文件中，并用一个 **demo** 介绍 CAEmitterLayer 的具体用法。

Movie 6.6 CAEmitterLayer 展示下雪效果

比如这个很经典的下雪效果：

代码如下。由于比较简单，我直接用尖帽符做了解释。：

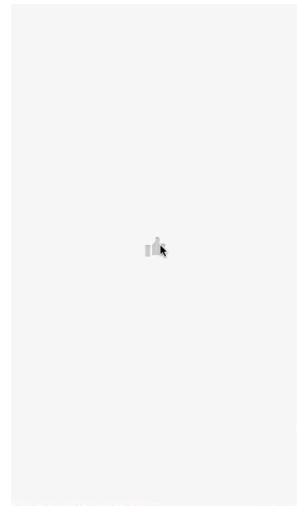
```
    CAEmitterLayer *snowEmitter = [CAEmitterLayer layer];

    snowEmitter.emitterPosition =
CGPointMake(self.view.bounds.size.width / 2.0, -30);

    snowEmitter.emitterSize  =
CGSizeMake(self.view.bounds.size.width * 2.0, 0.0);;
```

```
    snowEmitter.emitterShape = kCAEmitterLayerLine;
    snowEmitter.emitterMode  = kCAEmitterLayerOutline;

    CAEmitterCell *snowflake = [CAEmitterCell emitterCell];

    snowflake.birthRate  = 1.0;
    snowflake.lifetime   = 120.0;
    snowflake.velocity   = -10;
    snowflake.velocityRange = 10;
    snowflake.yAcceleration = 2;
    snowflake.emissionRange = 0.5 * M_PI;
    snowflake.spinRange = 0.25 * M_PI;
    snowflake.contents  = (id) [[UIImage imageNamed:@"snow"]
CGImage];
    snowflake.color   = [[UIColor colorWithRed:0.600 green:0.658
blue:0.743 alpha:1.000] CGColor];

    snowEmitter.shadowOpacity = 1.0;
    snowEmitter.shadowRadius  = 0.0;
    snowEmitter.shadowOffset  = CGSizeMake(0.0, 1.0);
    snowEmitter.shadowColor   = [[UIColor whiteColor] CGColor];


    snowEmitter.emitterCells = [NSArray
arrayWithObject:snowflake];
    [self.view.layer insertSublayer:snowEmitter atIndex:0];
```

这个简单的 demo 就到这里，代码参见 SnowEffectDemo。

下面我再追加一个例子，这个例子是 Github 上的一个开源库 —— MCFireworksButton。我觉得非常具有代表性。不仅效果出色，还涉及到了如何手动控制 CAEmitterLayer 动画的开始与结束。这也正是我想额外介绍的一点。

Movie 6.7 溅出水花的点赞效果



效果参见 Movie 6.8。

先看它的初始化代码：

```
-(void)setup {
    CAEmitterCell *explosionCell = [CAEmitterCell emitterCell];
    explosionCell.name = @"explosion";
    explosionCell.alphaRange = 0.20;
    explosionCell.alphaSpeed = -1.0;
    explosionCell.lifetime = 0.7;
    explosionCell.lifetimeRange = 0.3;
    explosionCell.birthRate = 0;
    explosionCell.velocity = 40.00;
    explosionCell.velocityRange = 10.00;
      explosionCell.contents = (id)[[UIImage
imageNamed:@"Like-Blue"] CGImage];
      explosionCell.scale = 0.05;
      explosionCell.scaleRange = 0.02; //如果你的 contents 的图片太
大了，可以通过设置 scale 属性缩小。

    _explosionLayer = [CAEmitterLayer layer];
    _explosionLayer.name = @"emitterLayer";
    _explosionLayer.emitterShape = kCAEmitterLayerCircle;
```

```
    _explosionLayer.emitterMode = kCAEmitterLayerOutline;
      _explosionLayer.emitterPosition =
CGPointMake(CGRectGetMidX(self.bounds),
CGRectGetMidY(self.bounds));
    _explosionLayer.emitterSize = CGSizeMake(25, 0);
    _explosionLayer.emitterCells = @[explosionCell];
    
    _explosionLayer.renderMode = kCAEmitterLayerOldestFirst;
      _explosionLayer.masksToBounds = NO;
    [self.layer addSublayer:_explosionLayer];

      self.emitterCells = @[explosionCell];
}
```

接下来我们需要手动控制动画的开始和结束。还记得前面提到的 `@property(copy) NSString *name;` 吗？想要手动控制动画的开始和结束，我们必须通过 KVC 的方式设置 cell 的值才行。

- 动画开始：

```
- (void)explode {
    self.explosionLayer.beginTime = CACurrentMediaTime();
    [self.explosionLayer setValue:@500
forKeyPath:@"emitterCells.explosion.birthRate"];
    dispatch_time_t delay = dispatch_time(DISPATCH_TIME_NOW,
0.1 * NSEC_PER_SEC);
    dispatch_after(delay, dispatch_get_main_queue(), ^{
        [self stop];
    });
}
```

对 `[self.explosionLayer setValue:@500 forKeyPath:@"emitterCells.explosion.birthRate"];` 这句话的解读

是：CAEmitterLayer 根据自己的 emitterCells 属性找到名叫 explosion 的 cell，并设置它的 birthRate 为 500。从而间接地控制了动画的开始。

同理，停止动画的思路也一样。通过设置 birthRate 为 0，间接地控制了动画的结束。

- 动画停止：

```
- (void)stop {
    [self.explosionLayer setValue:@0
forKeyPath:@"emitterCells.explosion.birthRate"];
}
```

到这里，这个 demo 基本就讲完了，你可以在 MCFireworksButton-Demo 这里找到源码。

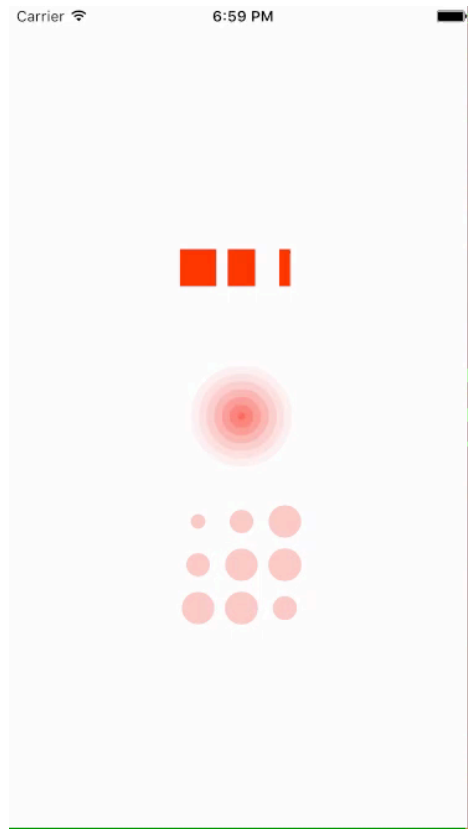
以上两个 demo 涉及了 CAEmitterLayer 会用到的绝大多数知识点，到最后可能真正费时的就是调参数了。没错，参数调节一直是一个优秀的动画的重要组成部分。

这一小节，我们讲介绍一个很小众的类 —— CAReplicatorLayer 。

我们先来看一下它能实现什么样的效果。

从 Movie 6.8 中可以看到，每一个动画中都存在好几个重复的元素。没错，这就是 CAReplicatorLayer 的最大特点，可以重复任意个数的元素按你期望的布局排列，甚至，可以让每个元素拥有动画，制作出这种类似 loading 效果的视觉效果。

Movie 6.8 CAReplicatorLayer Demo



首先我们来看一下官方的解释：

The CAReplicatorLayer class creates a specified number of copies of its sublayers (the source layer), each copy potentially having geometric, temporal and color transformations applied to it.

简单来说，它自己能够重建包括自己在内的 n 个 copies，这些 copies 是原 layer 中的所有 sublayers，并且任何对原 layer 的 sublayers 设置的变化是可以积累的（*accumulative*）。我们以视频中的脉冲动画为例子详细介绍一下 CAReplicatorLayer 的用法。

首先画一个圆形的 CAShapeLayer ,作为 CAReplicatorLayer 将要复制的原型，也可以理解为是第一个元素，下面的代码我设置了第一个元素的起始位置位于（0,0），在下面的介绍中你会认识到这很重要，因为这决定了接下去的元素该怎么排列。

```
let pulseLayer = CAShapeLayer()
pulseLayer.frame = bounds
pulseLayer.path  = UIBezierPath(ovalInRect:
pulseLayer.bounds).CGPath
pulseLayer.fillColor = color.CGColor
```

接下来，创建 CAReplicatorLayer ，并且做一些必要的初始化。其中 instanceCount 指定了一共显示元素的数目，instanceDelay 指定了重复元素之间的时间间隔，单位为秒。最后别忘了把之前创建的 pulseLayer 作为子图层添加到 CAReplicatorLayer 上。

```
let replicatorLayer = CAReplicatorLayer()
replicatorLayer.frame = bounds
replicatorLayer.instanceCount = 8
replicatorLayer.instanceDelay = 0.5
replicatorLayer.addSublayer(pulseLayer)
```

```
pulseLayer.opacity = 0.0
layer.addSublayer(replicatorLayer)
```

此时，你如果运行程序你会看到一个居中的圆。不是说有三个吗？没错，的确有三个，只是重叠在一起了（如果你想让每个元素的位置不一样，请继续往下看）。现在我们让每个圆圈动起来，因为 CAReplicatorLayer 会自动复制其上的所有图层，所以我们只要给第一个元素加上动画，之后复制出来的元素都会继承这个动画，又因为前面我们设置了 instanceDelay ，所以每个动画都会以相同间隔错开，从而连接成一个完整的动画。

下面我们来实现具体的动画，这是一个 CAAnimationGroup 动画，分别是透明度 1 ~ 0 的动画和缩放 0 ~ 1 的动画。

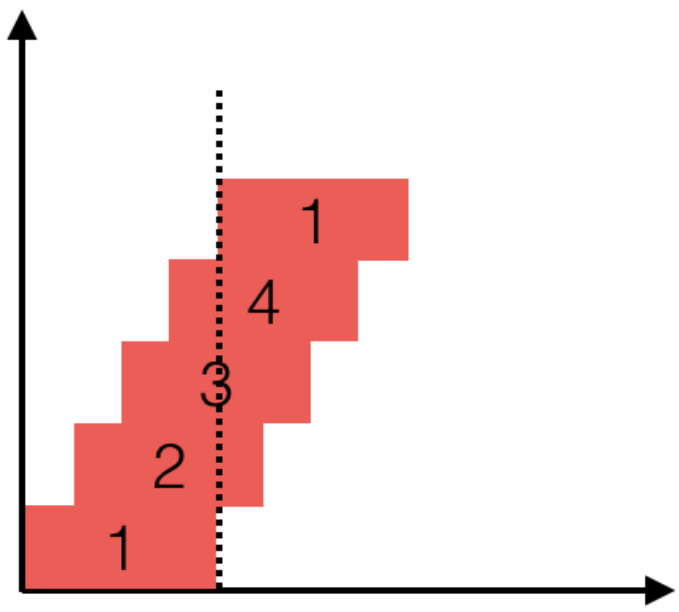
```
func startToPluse() {
    let groupAnimation = CAAnimationGroup()
    groupAnimation.animations = [alphaAnimation(), scaleAnimation()]
    groupAnimation.duration = 4.0
    groupAnimation.autoreverses = false
    groupAnimation.repeatCount = HUGE
    pulseLayer.addAnimation(groupAnimation, forKey: "groupAnimation")
}

private func alphaAnimation() -> CABasicAnimation{
    let alphaAnim = CABasicAnimation(keyPath: "opacity")
    alphaAnim.fromValue = NSNumber(float: 1.0)
    alphaAnim.toValue = NSNumber(float: 0.0)
    return
}
```

```
private func scaleAnimation() -> CABasicAnimation{
    let scaleAnim = CABasicAnimation(keyPath: "transform")
    let t = CATransform3DIdentity
    let t2 = CATransform3DScale(t, 0.0, 0.0, 0.0)
    scaleAnim.fromValue = NSValue.init(CATransform3D: t2)
    let t3 = CATransform3DScale(t, 1.0, 1.0, 0.0)
    scaleAnim.toValue = NSValue.init(CATransform3D: t3)
    return scaleAnim
}
```

这里我还要特别指出一点。你能发现下面三个数值之间的关系吗?

```
replicatorLayer.instanceDelay = 0.5
replicatorLayer.instanceCount = 8
groupAnimation.duration = 4.0
```

如果你注意到了 0.5*8 = 4.0 的话，请不要怀疑这是个巧合。这是动画连贯的必要条件。请看下面张图。



虚线处是一个元素完整动画所需要的时间。如果此时想让第一个元素第二遍动画立刻衔接上，那么就必须满足：

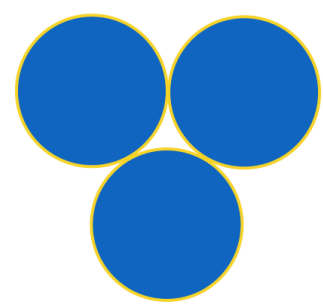
duration = instanceCount * instanceDe-lay。必须满从图中也可以看出。

到这里，我们已经完成了这个动画的全部工作。这个动画是最简单的 CAReplicatorLayer 动画，我们只是对 CAReplicatorLayer 有了一个简单的认识，下面我们来看看如何实现元素的自定义布局。

比如上面视频里的第一个动画，三个水平排列的翻转方块。

这里就要介绍 instanceTransform 这个属性了。这是自定义布局的关键。还记得第一个元素的坐标吗？当我们设定了第一个元素的坐标，之后的元素的坐标都会根据 instanceTransform 进行累加。比如这个并排的翻转方块，第一个元素的坐标是 (0，height*1/2)，设置 instanceTransform 为 replicatorLayer.instanceTransform = CATransform3DMakeTranslation(translationX, 0, 0)，所以第二个元素的位置就是第一个元素的 X 轴坐标平移 translationX 的距离，第三个元素的坐标就是第二个元素的 X 轴坐标平移 translationX 的距离，依此类推。

下面我们再来点旋转。比如像这个布局该怎么实现？



我们可以设置第一个元素的位置位于（0，0），后一个元素始终是前一个元素沿 X 轴坐标平移一段距离，同时绕自身中点旋转 120 度（圆周的三等分）。

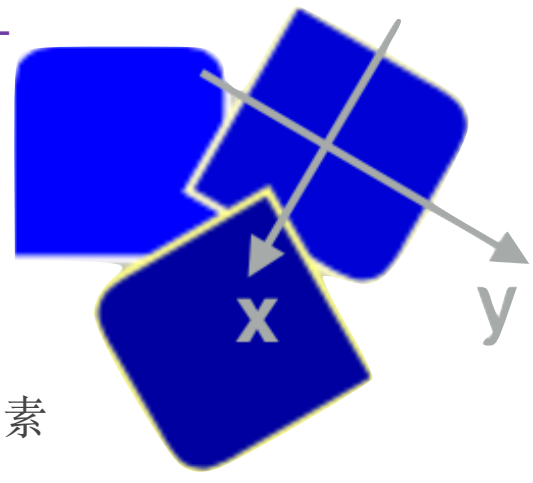
```
var transform = CATransform3DIdentity
transform = CATransform3DTranslate(transform, dotSize, 0, 0.0)
transform = CATransform3DRotate(transform,
(CGFloat(360/replicatorLayer.instanceCount)*CGFloat(M_PI))/CGFl
oat(180.0), 0.0, 0.0, 1.0)
replicatorLayer.instanceTransform = transform
```

考虑到圆形无论怎么旋转都一样，不利于我们的分析，所以我用了一个异形来直观地体现元素之间到底做了怎样的变换。请看下图。

左上角的方块是第一个元素，正朝向为圆角向上。

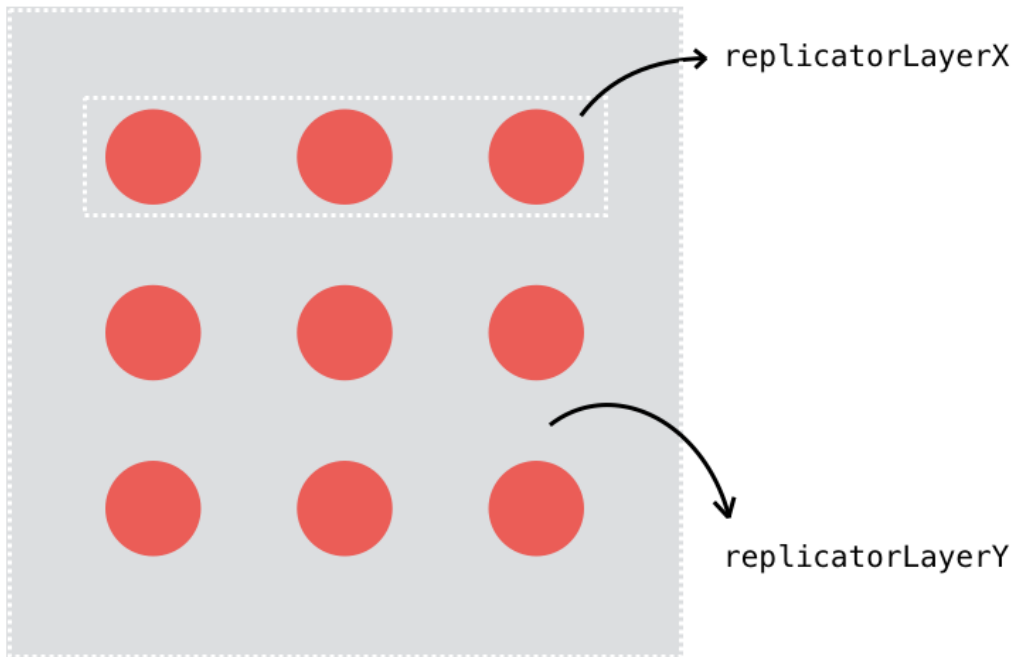
随后，元素中点平移规定距离，再顺时针旋转 120 度，这就是第二个元素的位置了。注意！你知道此时第二个元素的坐标系统是怎么样的吗？因为这直接关系到下一个元素的坐标。

右图中我标注了第二个元素的坐标示意图。



是的，也就是说，当我们设置了 `instanceTransform`，发生了旋转之后，其坐标轴也发生了旋转。这就很好解释了为什么第三个元素会出现在前两个元素的下方，因为我们规定了后一个元素是前一个元素沿着 X 轴正方向平移一段距离同时绕自身中点旋转 120 度获得的。

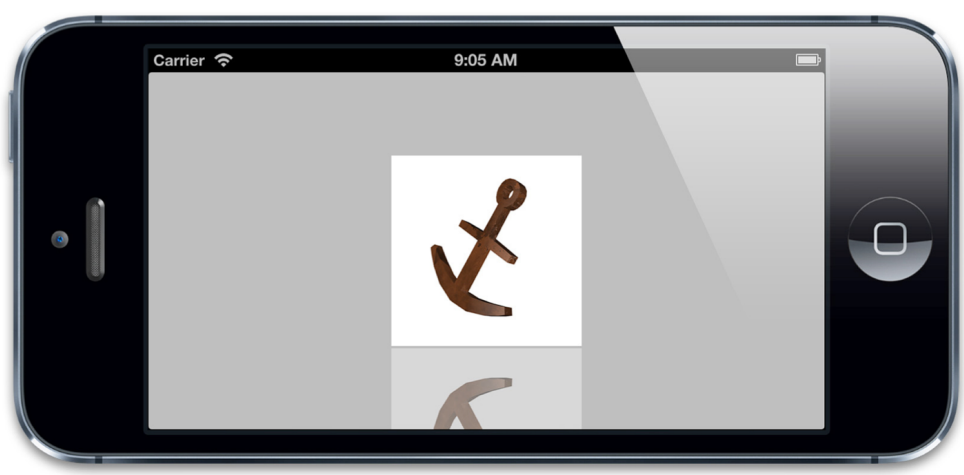
相信现在你对 instanceTransform 有了更深刻的理解。下面我们来看一看视频中第三个 3*3 的矩阵排列的实现思路。



这里我使用的一个技巧就是用两个 CAReplicatorLayer 。replicatorLayerX 先水平复制出三个圆圈，replicatorLayerY 再竖直复制出三个 replicatorLayerX 。就可以创建出一个 3*3 排列的矩阵了。

以上三个 demo 我封装到了一个库里，名字叫 **ReplicatorLoaderDemo-Swift** 。

关于 CAReplicatorLayer 其实还有一个「隐蔽」技能，就是用来做倒影效果。就像下面这种（图片来自 iOS Core Animation: Advanced Techniques）。



你只需要设定元素个数为 2 个，然后做一个 y 方向 -1 的缩放变换就行了。

```
replicatorLayerX.instanceCount = nbColumn
transform = CATransform3DScale(transform, 1, -1, 0)
```

當你翻到這一頁時，說明本書的內容已經全部結束了。非常感謝你能堅持到最後。我會一直更新下去，但恕我不能保證更新頻率。如果你平時看到了那些有意思的交互設計却疑惑如何實現的，請來我的微博 @KITTEN-YANG 找我，我們一起琢磨琢磨。如果你發現書中有什麼錯誤或者你有什麼疑問的地方，請聯系我 *kittenyang@icloud.com* 。

つづく

更多请访问 *Kitten*的时间胶囊
http://kittenyang.com