显示层动画 / 内容层动画 / 3D动画 / 转场动画

Broadview
www.broadview.com.cn


# iOS动画

## 核心技术与案例实战

· 郑微　编著 ·

基于Swift 3.0，舞动酷炫的iOS动画

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

# iOS动画
## 核心技术与案例实战

· 郑微　编著 ·


電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

## 内 容 简 介

目前，APP Store 上的应用已经超过 150 万个，而纵观排名较为靠前的应用，无一例外都有着一个共同的特点，那就是良好的用户体验。动画作为用户体验中最复杂、最绚丽的技术已经备受开发人员和产品设计人员的重视。而如何将炫酷的动画效果快速高效地展现出来已经成为 iOS 开发工程师面临的首要挑战。

本书以“iOS 核心动画架构＋实战代码”的形式阐述如何根据不同的应用场景设计高效、可靠、复杂的动画效果，为读者带来了丰富的实战动画案例，更从动画系统架构的角度阐释动画的原理，因此本书不仅面向读者“授之以鱼”更加“授之以渔”。



**图书在版编目（CIP）数据**

iOS 动画：核心技术与案例实战 / 郑微编著. —北京：电子工业出版社，2017.2
ISBN 978-7-121-30748-5

Ⅰ. ①i… Ⅱ. ①郑… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2016）第 316808 号

策划编辑：杨中兴
责任编辑：黄爱萍
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：720×1000　1/16　印张：13.25　字数：212 千字
版　　次：2017 年 2 月第 1 版
印　　次：2017 年 2 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

# 序　言

平时在家我经常会下载一些 iOS 排名比较靠前的应用或者游戏来玩，这些应用或者游戏都有比较显著的特点：界面优美、运行流畅、效果炫酷。这里暂且不去讨论 iOS 的硬件性能和 UI 美工设计是否优美，只是单纯从动画效果的角度看，它们对 iOS 动画效果的应用有着非常精致的把控。

在日常工作中，每开发一款 APP，我们都会绞尽脑汁想让这款应用与众不同。其实不用太过于纠结系统，因为 iOS 的硬件都很棒。也不必太过于纠结美工，相信一个稍微靠谱的美工做出来的 UI 都不会太差。只需要选择和设计一些比较优美的动画，就可以让自己的应用上一个新的台阶。

在工作过程中大家都经历过这三个阶段。第一阶段初入江湖。在这个阶段如果想设计一个比较炫酷的动画效果，要么请教“大神”，要么进行网络搜索，而很少有自己的想法。这主要是因为大家对动画的架构、常用 API、常用效果没有一个全面的认识，这个阶段基本属于代码收集阶段。第二阶段渐入佳境。相信大家在这个阶段都会有一些自己的思想，通过不断的尝试、对 API 不断地调整都能够实现最终想要的效果。总体来说这一阶段属于代码整理阶段。第三阶段登堂入室。需求来了之后在开发人员的大脑中很快被分解为若干子功能，迅速定位子功能需要实现的代码块。通过“搬砖+修改”的模式实现快速开发。这一阶段基本属于代码灵活运用阶段。

如果从零开始一步步完成这样三个阶段，相信大家都能做到，但是这会花费非常多的时间。在工作中我也曾被第一阶段和第二阶段反复困扰过，走了不少弯路，花费了大量的时间和精力，查看了各种官方手册和相关书籍，直到进入第三阶段才体会到“一览众山小”的感觉。所以我很想把 iOS 关于动画的相关知识为大家抽丝剥茧地整理一番，以帮助更多的人花费更少的时间掌握尽可能多的知识。

## 1．这本书有哪些特点

### （1）层次分明

iOS 动画效果非常丰富，本书一共 16 章，根据动画实现方式及效果分为 4 卷。第一卷（第 1～5 章）介绍显示层动画效果，第二卷（第 6～12 章）介绍内容层动画效果，第三卷（第 13～14 章）介绍 3D 动画效果，第四卷（第 15～16 章）介绍转场动画效果。这种划分有利于读者在学习的过程中对所要查找的动画效果快速定位，以及将知识点分类掌握。

第一卷为显示层动画效果，即利用 UIView 图层显示的效果实现各种动画。常见的有位置动画、颜色动画、淡入淡出动画、旋转动画、关键帧动画、逐帧动画等，文中针对要显示的动画效果一般采用 3 张图渐进描述，对于复杂的动画效果多采用 6 张图或 9 张图描述动画的渐变过程。其效果分别如图 1～图 4 所示。


图 1　位置动画


图 2　颜色渐变动画

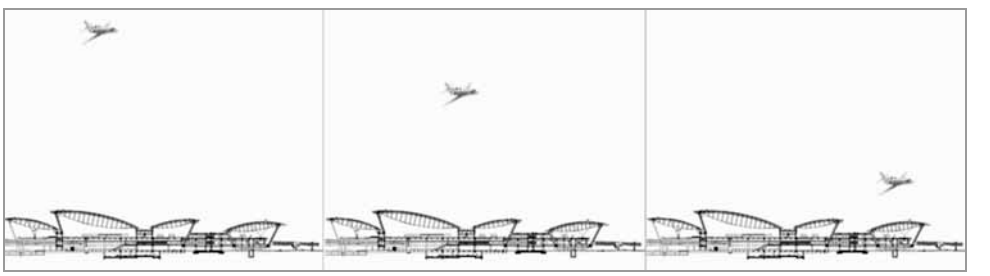


图 3　关键帧动画

图 4　逐帧动画

第二卷为内容层动画效果。内容层动画依赖视图的 Layer 图层，结合常用 Layer 子类，如 CAEmitterCell 粒子动画、CAGradientLayer 扫描动画、CAShapeLayer 图表类动画、CAReplicatorLayer 图层快速复制动画等实现内容层动画展示，如图 5～图 8 所示。

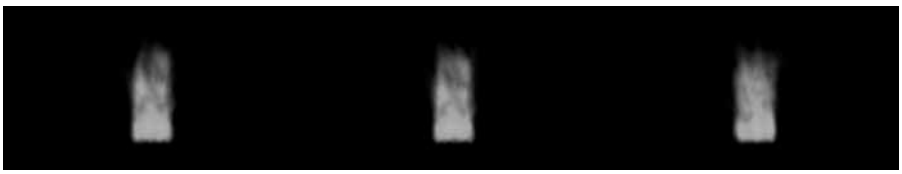


图 5　CAEmitterCell：粒子“鬼火”效果

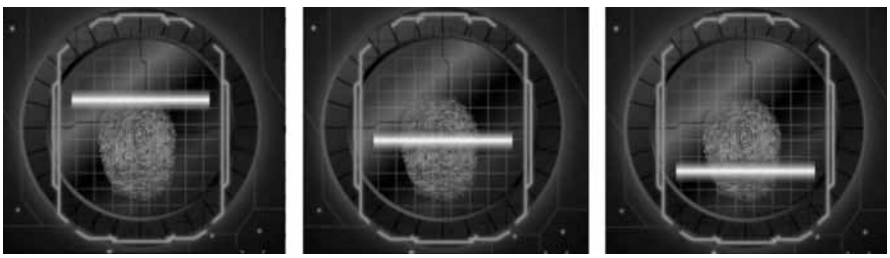


图 6　CAGradientLayer：指纹扫描效果

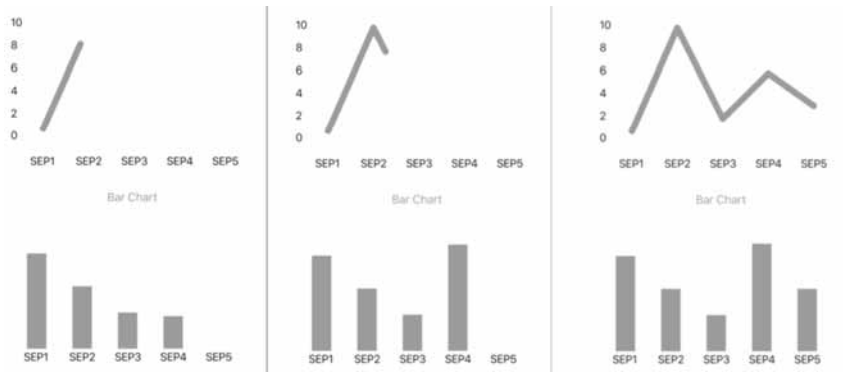



图 7　CAShapeLayer：动态图表效果

图 8　CAReplicatorLayer：“恒星”公转效果

第三卷为 3D 动画效果。3D 动画效果以矩阵变换为基础，利用 *x*、*y*、*z* 与变换矩阵相互作用实现各种 3D 效果，如图 9 所示。比如 Cover Flow 的 3D 动画展示效果，如图 10 所示。

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} * \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

coordinate　　transform　　transformed coordinate

图 9　变换矩阵原理解析

图 10　Cover Flow 展示效果

第四卷为转场动画效果。转场动画常用于多视图场景下视图切换，如常见的水滴、翻页、波纹效果，或者自定义视图控制器转场动画，如图 11～图 12 所示。

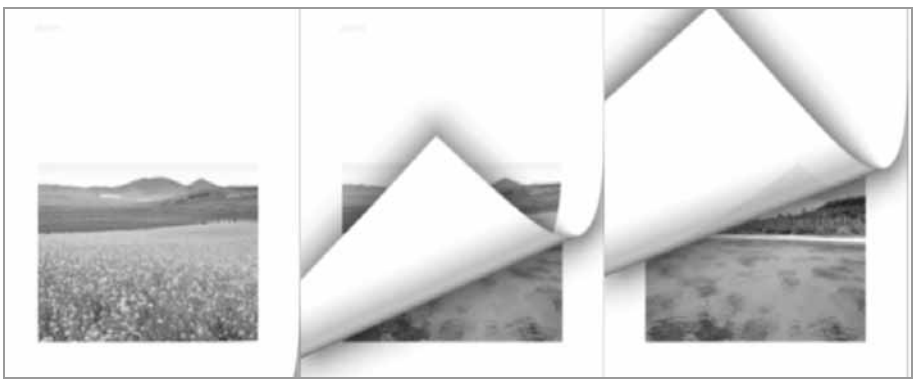


图 11　转场翻页效果


图 12　自定义转场蒙版效果

（2）内容丰富

对于相同类型、相同知识点的动画，书中做了详细的归纳和总结，这种归纳和总结有利于读者对动画整体架构的把握和快速精准的使用，如图 13～图 15 所示是部分动画合集知识点。


图 13　UIView 常用动画合集

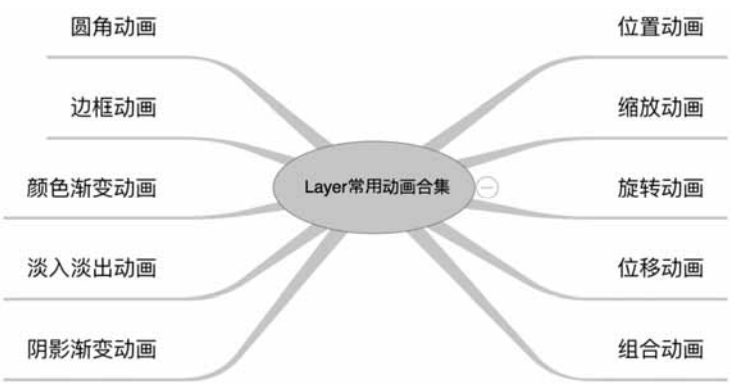



图 14　Layer 常用动画合集

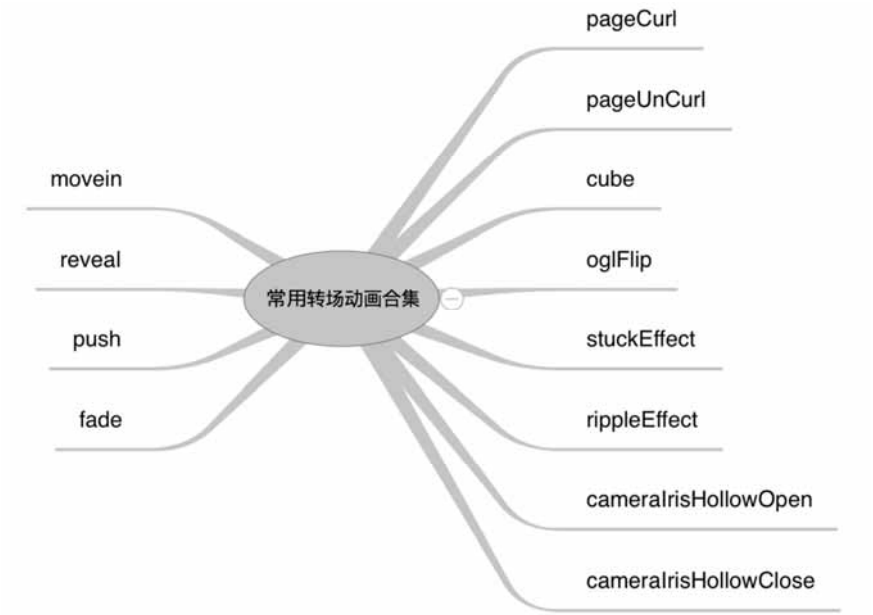



图 15　常用转场动画合集

（3）适用性强

本书针对每个章节给出适合阅读的人群，便于读者过滤出适合自己的核心内容。

（4）实用性强

在讲解每个动画案例的同时，尽可能贴近实际使用场景，如第二卷的各种 Layer 层动画实战案例、Button 按钮相关动画效果等，如图 16～图 17 所示。


图 16　按钮水纹点击效果

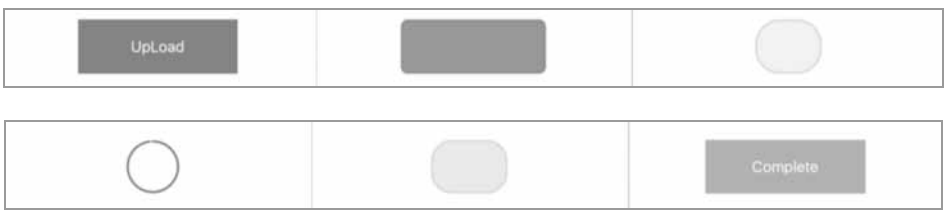



图 17　按钮登录效果

## 2．iOS 动画架构一览

根据不同维度对动画架构进行划分，可以很好地帮助大家理解 iOS 动画的结构及不同类型动画之间的相互联系。如图 18～图 20 所示。


图 18　iOS 动画分层结构

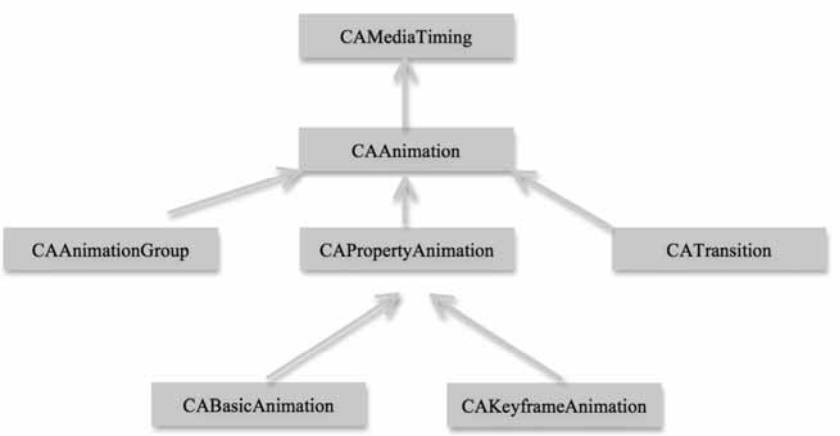



图 19　核心动画类结构

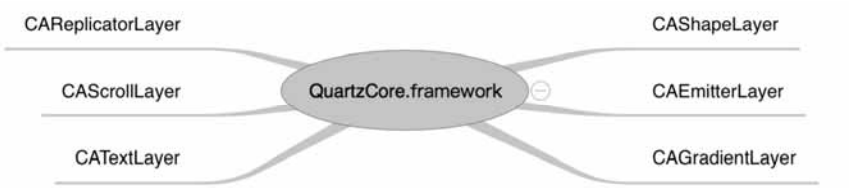



图 20　QuartzCore 框架下常用图层

## 3．如何使用这本书

俗语说得好：“工欲善其事，必先利其器”，要想很好地使用这本书，必

须先准备好一定的工具。本书中的所有代码都是基于 Swift 语言开发的，建议使用 Xcode 8.0、SDK 10.0 以上版本调试。源码下载地址详见：http://www.broadview.com.cn/30748。

工具和源码准备好之后呢？大家都知道几乎没有一本书百分之百适合我们，但或许有某个章节或者某个知识点刚好是大家想要学习和掌握的，本书也不例外。为了方便大家更好地使用这本书，特将本书的特点描述如下，希望广大读者根据自己的实际情况进行相关内容的阅读学习。

- **iOS 初级开发工程师**：建议从第 1 章动画入门开始，循序渐进地阅读。
- **iOS 中级开发工程师**：建议浏览或者跳过第一卷，重点关注第二、三、四卷。
- **iOS 高级开发工程师**：建议挑选工作中需要或感兴趣的章节阅读，如第 9 章“粒子动画”、第 10 章“光波扫描动画”。
- **iOS 超级开发工程师**：建议从整体架构和动画整理归纳的角度阅读本书。

## 致谢

在此感谢电子工业出版社的杨中兴编辑为本书提出的宝贵意见，感谢各位技术博主对本书的大力支持。最后感谢一直深爱并默默支持我的妻子，感谢她对我的关心和照顾，使得我可以抽出更多时间全身心地编写此书。

由于时间仓促，书中难免存在不足之处，欢迎大家批评指证。

郑　微

2016 年 12 月于武汉

# 目　录

## 第一卷　显示层动画

## 第二卷　内容层动画

# 第一卷

# 显示层动画

# 第 1 章　动画之旅启航：登录按钮动画效果

## 本章内容

- 掌握动画分析步骤“三步曲”
- 掌握 UIView 下 Frame 属性动画移动效果

在本章，将实现简单的“登录界面按钮移动效果”，并通过这个动画效果为大家介绍动画设计和分析的思路。本章的目标不仅仅是让大家弄清楚动画效果是如何通过代码来实现的，更重要的是希望大家通过对本章节的学习，掌握动画设计和分析的思路。并以这个思路为基础，设计更为复杂、绚丽的动画效果。

首先先来看看动画设计中的三个角色：产品设计师、算法分析师以及伟大的程序员都有哪些职责。

（1）产品设计师：告诉大家想做一个什么样的动画。

（2）算法分析师：分析动画的实现原理并设计相应的动画算法。

（3）程序员：思考如何用代码实现算法。

在一般中小规模的公司中，开发人员往往都是身兼数职。不仅要编写代码还要参与到算法的设计中去，甚至参与到动画原型的设计中去。所以弄清楚动画设计过程中的不同角色，以及搞清楚动画的分析过程是非常有必要的。

# 1.1　动画分析方法

如图 1.1 所示是我们想要实现的动画效果，那么如何来分析它呢？其实产品设计师在设计动画时，如果能够将动画分解为单帧图像，或者能够较为慢速地展现动画的变化过程，那么对于算法分析师和程序员分析动画的原理，以及设计合适的展现算法起着非常重要的作用。图 1.1 描述了登录按钮从左到右逐渐移动的效果，并最后停留在视图层中间位置这一过程。


图 1.1　登录界面图标移动分帧效果

这个动画效果非常简单，可以用一句话来描述其实现算法，即图像的水平方向位置坐标和时间呈线性渐变关系。接下来思考如何用代码实现这个效果。按照动画的展示过程，这里将动画分为：动画起始阶段、动画进行阶段和动画结束阶段。

### 1．动画起始阶段

在动画启动的瞬间，希望动画从屏幕可视界面外飞入进来。如图 1.2 所示的登录按钮是需要实现的动画起始位置。在 iOS 视图中，左上角为视图的原点（0，0），水平向右为 $x$ 轴递增方向，竖直向下为 $y$ 轴递增方向，只有当 View 视图

位于手机屏幕展示坐标系之内，大家才能看到（虚线区域内控件不可见），否则登录按钮是不可见的。所以在动画的起始阶段可以将动画的位置属性设置在界面之外。

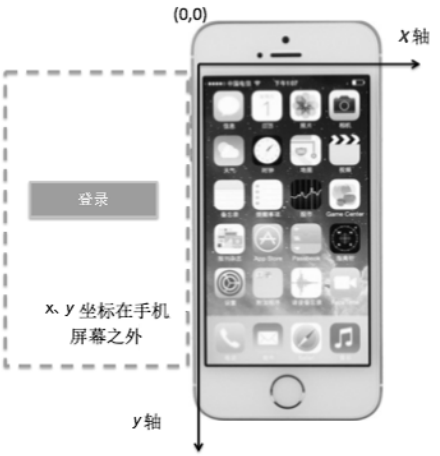



图 1.2　动画起始阶段登录按钮所在位置

### 2．动画进行阶段

经过前面的分析，大家已经了解了这个动画效果的实现算法，即登录按钮的坐标沿水平方向随时间线性变化。如表 1.1 所示描述了不同时间段登录按钮的坐标变化情况。幸运的是大家不需要手动设计这一过程，甚至不需要手动写线性渐变的方法，因为 iOS 在 UIView 的显示层已经帮我们把这个功能集成了。iOS 在 UIView 图层中不仅集成了动画的线性渐变方法，而且动画的加速、减速以及复杂的动画变化时间函数、运动路径函数也已经为大家集成好了，所以只需要学会如何使用这些丰富的 API 即可，且这个功能只需要几行代码就可以实现。

表 1.1　6S 下 QQ 图标移动效果：QQ 图标 $x$、$y$ 坐标随时间变化关系表

| | 0.25s | 0.5s | 0.75s | 1.0s |
|---|---|---|---|---|
| View($x$,$y$)坐标 | （−394,$y$） | （−256,$y$） | （−118,$y$） | （20,$y$） |

### 3．动画结束阶段

在动画效果结束之后没有触发新的回调事件，只是更新了当前登录按钮的最后位置，所以图片最终停留在视图层的中间位置。

## 1.2　登录按钮移动动画效果：闭包形式

首先创建一个单视图工程，创建好之后可以看到如图 1.3 所示的工程文件目录结构：

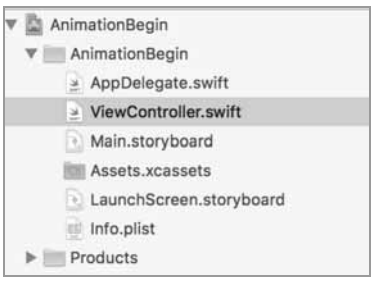



图 1.3　单视图工程文件结构

**动画实现的第一阶段：动画起始阶段**

在开始正式添加动画代码之前需要为应用添加一个背景图片。在 Main.storyboard 中为整个工程添加一个已经准备好的背景图片，背景图片依托在 UIImageView 上。如图 1.4 所示为当前工程的 Main.storyboard 中图层结构，其中 View Controller 为整个工程的视图控制器，login 为 UIImageView 登录背景图片。如图 1.5 所示是准备好的背景图片，通过图 1.5 可以看出，要想实现图 1.1 所示的动画效果，只需为整个登录界面添加一个登录按钮即可。

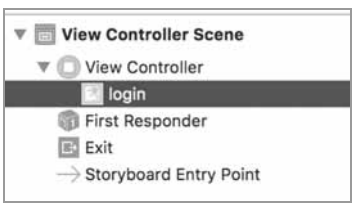



图 1.4　Main.storyboard 中图层结构


图 1.5　登录背景图片

动画起始阶段代码需要放在什么位置才合适呢？要想弄清楚这个问题先搞清楚 ViewController.swift 中几个方法的执行顺序。需要关注以下 3 个方法。

```
viewDidLoad()
viewWillAppear()
viewDidAppear()
```

在应用启动之后，在 viewDidLoad 中会装载所有的 View 视图，注意，虽然所有 View 视图都被装载进来，但是这时所有的 View 视图并不是可见的。程序接着调用 viewWillAppear 方法，这是视图在展现之前需要调用的方法。而最后调用 viewDidAppear，表明所有的视图已经可见。经过以上分析，大家应该清楚，在动画起始阶段可以将所有的初始化代码放置在 viewDidLoad()方法中。具体实现代码如下所示。

```
1  var loginButton:UIButton?
2  override func viewDidLoad() {
3      super.viewDidLoad()
4      loginButton = UIButton(frame: CGRect(x:-394,y:230,
                   width:self.view.frame.width-20*2,height:50))
5     loginButton!.backgroundColor = UIColor(red: 50/255.0, green:
185/255.0, blue: 170/255.0, alpha: 1.0)
```

```
    6       loginButton!.setTitle("登录", for: UIControlState. normal)
    7      self.view.addSubview(loginButton!)
        }
```

代码第 1 行创建了一个 UIButton 登录按钮。第 3 行重写 viewDidLoad 方法，表明应用启动之后首先通过调用 viewDidload 方法加载各种 UI 组键。第 4 行设置当前 UIButton 登录按钮的位置，按钮的 *x* 坐标设置在整个界面之外，因此当前 Button 按钮是不可见的。第 5 行为登录按钮添加一个淡绿色背景。第 6 行设置登录按钮 Title 内容。最后一行将按钮添加到 self.view 图层上。

**动画实现的第二阶段和第三阶段：动画进行阶段和动画结束阶段**

要想实现应用打开动画即展现的效果，需要在 View 视图整体展现之前完成动画实现的第二阶段和第三阶段的设置（因为如果视图已经显示了才设置动画效果，那么会有动画不连贯的现象），所以这部分功能只能放置在 viewWillAppear 方法中。

这里使用的 UIButton 按钮和 UI 控件都是继承 UIView 类，UIView 类中有一个动画方法可以完成我们想要实现的功能：

```
    open class func animate(withDuration duration: TimeInterval,
animations: @escaping () -> Swift.Void)
```

该方法属于类方法，类名可以直接调用，表明为当前的 UIView 添加一个动画效果，它的每个参数的含义如下。

- duration：表明动画执行周期。
- animations：表明动画执行内容。

注意，这里 animations 是一个闭包，使用闭包的方式将动画代码追加进去。在闭包中只需要将动画的结束状态设置完成，那么动画从开始到结束的中间过程，iOS 都会自动实现。下面为 viewWillAppear()中的动画实现代码。

```
        override func viewWillAppear(_ animated: Bool) {
    1       UIView.animate(withDuration: 1, animations: {
    2       self.loginButton!.frame = CGRect(x:20,
                    y:self.loginButton!.frame.origin.y,
```

```
                width:self.loginButton!.frame.width,
                height:self.loginButton!.frame.height))
        })
    }
```

animate 方法中，duration 表明动画执行周期为 1s，动画闭包部分表明登录按钮最终的位置，即最终停留在手机屏幕的中间位置。

## 1.3 登录按钮移动动画效果：方法形式

除了使用闭包的方法之外，还可以使用另外一种方式实现这个动画效果，即通过 commit 相关方法的形式来实现。通过修改 viewWillAppear()中的内容，可以实现相同的动画效果。下面是动画移动效果的另外一种代码实现方式。

```
    override func viewWillAppear(animated: Bool) {
1       UIView.beginAnimations(nil, context: nil)//动画开始
2       UIView.setAnimationDuration(1)//动画周期设置
3       loginButton!.frame = CGRect(x:20,
            y:loginButton!.frame.origin.y,
            width:loginButton!.frame.width,
            height:loginButton!.frame.height)//动画位置设置
4       UIView.commitAnimations()//动画提交
    }
```

代码第 1 行表明动画开始，这里先忽略需要传递的参数，可以先传递两个 nil。第 2 行设置动画执行周期，这里将动画周期设置为 1s。第 3 行将登录按钮设置在屏幕中间位置。代码最后一行将动画效果提交到系统上运行。

其实无论是第 1.2 节的动画实现方法抑或是第 1.3 节的动画实现方法，都可以把动画实现的过程总结为如图 1.6 所示的 3 个步骤。

而第 1.2 节和第 1.3 节实现动画的唯一区别就是一个使用闭包的形式，而另一个使用 beginAnimations 和 commitAnimations 方法的形式启动动画。

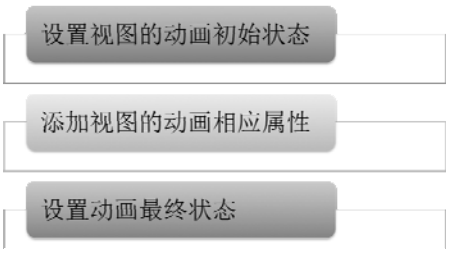



图 1.6　动画实现步骤

## 1.4　UIView 视图中常见动画的属性分析

我们在第 1.2 节和第 1.3 节主要依靠 UIView 下的 frame 属性来实现登录按钮从左到右的进入效果。那么 UIView 下的其他属性是不是也可以有类似的效果呢？要想回答这个问题，首先需要弄清楚 UIView 都有哪些常见的属性。

### 1．位置属性：frame bounds center

frame、bounds、center 都是描述 UIView 的位置信息，不同的是 frame 可以对 *x* 坐标、*y* 坐标、width、height 四个属性进行操作，frame 的 *x* 坐标和 *y* 坐标相对于父控件的原点来计算，而 bounds 一般只能对 width、height 进行操作，它的 *x*、*y* 坐标只相对于自身而言，center 描述的是 *x*、*y* 信息，即 UIView 的中心位置。下面是 CGRect、origin、size 的代码描述。

```
public struct CGRect {
    public var origin: CGPoint
    public var size: CGSize
    public init()
    public init(origin: CGPoint, size: CGSize)
}
```

再来看看三者的数据类型。frame 是 CGRect 类型，它是一个结构体，在结构体中包含 origin 和 size 两个属性。其中 origin 描述 UIView 的 *x*、*y* 坐标起始位置信息，size 描述 UIView 的 width、height 宽高信息。我们再来看看 origin 的 CGPoint 和 size 的 CGSize 又是什么。

```
public struct CGSize {
    public var width:
                    CGFloat
    public var height:
                    CGFloat
    public init()
    public init(width:
 CGFloat, height: CGFloat)
```

```
public struct CGPoint {
    public var x: CGFloat
    public var y: CGFloat
    public init()
    public init(x: CGFloat,
y: CGFloat)
}
```

CGpoint 中包含了 UIView 的 *x*、*y* 坐标，而 CGSize 中包含了 UIView 的 Width、Height 信息。通过对 frame 中数据类型的追本溯源，可以得到以下结论：CGRect 分别对应 *x* 坐标、*y* 坐标、width、height 四个属性。这四个属性表明当前 UI 在它的父控件上的位置，如 self.view 上。

通过以上分析可以知道，可以通过 *x*、*y* 坐标修改 UIView 的移动位置，还可以通过修改 width 或者 height 来修改 UIView 的拉伸、收缩效果。对于 bounds 属性使用最多的还是 width、height 属性，center 则经常使用 *x*、*y* 坐标属性。

### 2．透明度属性：alpha（透明度属性、范围 0-1、浮点型）

UIView 的 alpha 透明度属性也可以用作动画效果。当 alpha 为 0 时，表明 UIView 已经隐藏，当 alpha 为 1 时 UIView 显示。结合这一特征可以通过修改 alpha 在动画开始、结束时的值，实现 UIView 的淡入淡出效果。

### 3．Layer 属性：圆角渐变、边框颜色、阴影、3D 等高级动画效果

UIView 是视图显示的容器，负责内容显示和事件响应。每个 UIView 都有一个 Layer 图层，在这个图层中承载的是视图的内容，所以结合 Layer 可以实现很多高级的动画效果。当然除了这些之外，UIView 还有很多其他属性，在后面的章节中会为大家一一呈现。

## 1.5　本章小结

通过对本章的学习，相信大家基本上掌握了动画分析的基本步骤，在这里总结一下动画实现的三个步骤：

（1）设置视图的动画初始状态。

（2）添加视图的动画相应属性。

（3）设置视图的动画最终状态。

在通过帧分解等方法了解了动画的实现原理之后，通过这三个步骤可以很方便地实现各种动画效果。本章结合登录按钮移动效果为大家展示了动画分析和实现的全过程，并通过代码详细介绍了通过闭包方式，以及 beginAnimations 和 commitAnimations 方法的形式实现动画。

对于 UIView 中常见的动画属性，结合 UIView 对视图的位置、透明度、几何形状给大家做了简要的分析，在后面的章节中会结合具体的代码，为大家呈现缤纷多彩的动画效果。

# 第 2 章　显示层初级动画效果合集



本章内容

- 掌握 UIView 显示层常用动画效果合集：位置、几何形状、旋转、缩放、淡入淡出、颜色渐变……
- 掌握动画效果的组合使用
- 理解抽奖转盘动画效果的实现方法

本章将对 UIView 显示层常用的所有动画效果做一个小结，重点掌握动画中位置、几何形状、旋转、缩放、淡入淡出、颜色渐变等动画效果，并对 CoreGraphics 框架中的 CGAffineTransform 属性做一定的了解。本章还是以第 1 章中的登录按钮为例来实现这些动画效果。

## 2.1　UIView 显示层初级动画属性一览

UIView 显示层动画效果的实质还是通过修改 UIView 的各种属性来实现的。比如，如果想实现移动动画效果，则可以设置 UIView 的 frame 位置属性，如果想实现淡入淡出效果，则可以设置 UIView 的 alpha 属性。所以要想弄清楚 UIView 显示层都有哪些动画效果，首先要了解 UIView 的各种常用属性。

UIView 是大多数 UI 组件的父类，其常用的属性和方法有很多，不过涉及

动画效果的属性和方法并不是很多。下面是为大家整理的 UIView 动画效果经常涉及的属性。

```
public init(frame: CGRect)
public var bounds: CGRect
public var center: CGPoint
public var alpha: CGFloat
@property(nullable, nonatomic,copy)UIColor  *backgroundColor
public var transform: CGAffineTransform
```

# 2.2　初级动画效果合集

## 2.2.1　位置动画

在第 1 章中，利用 UIView 的 frame 属性实现了登录按钮的移动动画效果，下面将结合 UIView 的 center 属性来实现这一效果。界面的初始化部分和登录按钮的添加代码请参考第 1 章，这里就不再赘述。

UIView 的 center 属性表明当前 UI 的中心位置，我们想让登录按钮最终停留在界面的中心位置，如图 2.1 所示。


图 2.1　位置动画最终效果

在 iPhone 6s plus 中，当前登录按钮的 center 可以通过 print 的形式打印出来。

最终登录按钮将停留在 center 为（207.0, 255.0）的位置上。所以要想实现第 1 章的动画移动效果只需要将按钮的最终中心位置设置在（207.0, 255.0）即可。下面是具体代码实现。

```
override func viewWillAppear(animated: Bool) {
//      位置动画
1       UIView.beginAnimations(nil, context: nil)//动画开始
2       UIView.setAnimationDuration(1)//动画周期设置
3       loginButton!.center = CGPoint(x:207,y:255)//动画位置设置
4       UIView.commitAnimations()//动画提交
5       print("center:\(loginButton!.center)");
}
```

代码第 1 行使用 UIView 的类方法表明动画开始，第 2 行设置动画执行周期为 1s，第 3 行设置动画中心位置，第 4 行提交动画，最后一行打印当前动画的中心位置。

## 2.2.2 几何形状动画

可以根据不同的应用场景，把登录按钮进行压缩或者拉伸，动画效果如图 2.2 所示。通过观察图 2.2 中第一张图和最后一张图，基本可以弄明白这个动画效果的演变过程。在动画的变化过程中，登录按钮的宽度被压缩，高度被拉伸。所以通过修改登录 UIButton 的宽高即可实现图中的动画效果。


图 2.2　动画拉伸压缩效果

下面是具体的代码实现。

```
override func viewWillAppear(_ animated: Bool) {
        //      几何形状
1        UIView.beginAnimations(nil, context: nil)//动画开始
2        UIView.setAnimationDuration(1)//动画周期设置
3        loginButton!.bounds = CGRect(x:0,y:0,
                 width:loginButton!.frame.size.width*0.7,
                 height:loginButton!.frame.size.height*1.2)
4        UIView.commitAnimations()//动画提交
}
```

该代码与第 2.1 节的代码类似，重复部分就不再赘述，其核心代码在第 3 行，通过修改 UIButton 的 Bounds 属性来实现登录按钮宽度被压缩，高度被拉伸的效果。登录按钮宽度修改为原来的 0.7 倍，相当于宽度被压缩。高度修改为原来的 1.2 倍，相当于高度被拉伸。Bounds 中 $x$、$y$ 坐标都设置为 0，因为 Bounds 中的 $x$、$y$ 是相对于自身而言，实际并不修改登录按钮的位置。

### 2.2.3　位置+形状动画

如果想让位置和形状一起变化如何来实现呢？比如实现如图 2.3 所示的效果。登录按钮一边向整个应用的左边对齐，一边宽度压缩、高度拉伸。细心的读者可能会发现，利用位置动画和几何形状动画的组合动画即可实现这种效果。


图 2.3　登录按钮位置+形状动画效果

使用 Bounds+center 属性可以实现图 2.3 的动画效果，不过在本小节为大家介绍另外一种动画实现方法。Bounds 属性的 width、height 可以修改拉伸缩放，center 的 x、y 属性可以修改位置。使用 UIButton 的 frame 属性，可以同时囊括 x、y、width、height。下面是具体实现代码。

```
override func viewWillAppear(_ animated: Bool) {
//      位置+形状动画
1       UIView.beginAnimations(nil, context: nil)//动画开始
2       UIView.setAnimationDuration(1)//动画周期设置
3       loginButton!.frame = CGRect(x:0 ,y:230,
                width:loginButton!.frame.size.width*0.7,
                height:loginButton!.frame.size.height*1.2)
4       UIView.commitAnimations()//动画提交
}
```

## 2.2.4 淡入淡出动画

在第 2.2.1 节实现的是登录按钮移动动画效果，这种类型的动画按照功能来分还可以称之为进入动画。在本小节将为大家介绍另外一种控件进入动画：淡入动画。如图 2.4 所示为淡入动画效果图。


图 2.4　淡入动画效果

从图 2.4 中可以看出按钮的透明效果逐渐变弱，登录按钮逐渐出现在视野之内。回顾第 2.1 节中 UIView 的常用属性，alpha 的透明度属性可以实现这一效果。

**注意**：默认情况下 UIView 的子控件中 alpha 的值都为 1，即可见状态。要想实现淡入过程，在代码中，需要先把 UIButton 的 alpha 属性设置为 0，然后在动画执行效果中将动画的透明度属性设置为 1.0。下面是具体的实现代码。

```
1 loginButton!.alpha = 0;//登录按钮初始化时，将其透明度设置为 0，即隐藏状态
  override func viewWillAppear(_ animated: Bool) {
                        //      淡入淡出
2       UIView.beginAnimations(nil, context: nil)//动画开始
3       UIView.setAnimationDuration(2)//动画周期设置
4       loginButton!.alpha = 1;
5       UIView.commitAnimations()//动画提交
  }
```

## 2.2.5　颜色渐变动画

在第 2.2.1 节登录界面初始化为可用状态，颜色设置为青绿色。当按钮或者控件变为不可用状态时需要将登录按钮设置为灰色，表明当前不可点击。这一过程可以使用动画来描述。如图 2.5 所示描述了登录界面按钮从可用状态到不可用状态的颜色渐变过程。


图 2.5　淡入动画效果

下面是具体的代码实现方法：

```
override func viewWillAppear(_ animated: Bool) {
//      颜色渐变
1        UIView.beginAnimations(nil, context: nil)//动画开始
2        UIView.setAnimationDuration(2)//动画周期设置
3        loginButton!.backgroundColor = UIColor.gray
4        UIView.commitAnimations()//动画提交
}
```

### 2.2.6 缩放动画：基于 UIView 的 transform 属性

UIView 有一个非常重要的动画属性 transform，该属性继承于 CGAffineTransform，“CG”实际上是 CoreGraphics 框架的缩写。可见 transform 属性是核心绘图框架与 UIView 之间的桥梁，借助于这一属性可以制作很多高级的动画效果。

transform 最常用的三种动画分别是缩放、旋转和位移。在第 2.2.2 节介绍了 UIView 的缩放效果，即几何尺寸的变化，下面将采用 transform 属性完成这一动画效果。下面是具体的代码实现。

```
override func viewWillAppear(_ animated: Bool) {
//      CGAffineTransform:缩放
1        UIView.beginAnimations(nil, context: nil)//动画开始
2        UIView.setAnimationDuration(1)//动画周期设置
3        loginButton!.transform = CGAffineTransform(scaleX: 0.7,
                                                    y: 1.2)
4        UIView.commitAnimations()//动画提交
}
```

这段代码与第 2.2.2 节的缩放动画代码有很多重复的地方，这些重复的地方就不一一赘述。重点来关注代码的不同点，即代码的第 3 行 CGAffineTransform 方法是缩放动画的核心。

```
CGAffineTransform(scaleX: 0.7, y: 1.2)
```

该方法有两个输入参数，分别控制 UI 组件宽高的缩放比。当 sx 设置为 1.0

的时，表明 UI 组件宽度没有缩放；当 sx 设置为 0.5 时，表明 UI 组件宽度缩小为原来的一半。这里设置登录按钮宽度为原来的 0.7 倍，高度设置为原来的 1.2 倍。

### 2.2.7　旋转动画：基于 UIView 的 transform 属性

若想让登录按钮旋转 45°如何来实现呢？通过第 2.2.6 节对 transform 属性的介绍可以知道，CGAffineTransform 类中会有响应的旋转动画 API。下面是具体的实现代码。

```
    override func viewWillAppear(_ animated: Bool) {
    //      CGAffineTransform:旋转
    1       UIView.beginAnimations(nil, context: nil)//动画开始
    2        UIView.setAnimationDuration(1)//动画周期设置
    3        let angle:CGFloat = CGFloat(M_PI_4);
    4        loginButton!.transform = CGAffineTransform(rotationAngle:
angle)
    5        UIView.commitAnimations()//动画提交
    }
```

iOS 的 SDK 中提供了很多宏，方便开发者直接使用，在上面的代码中，使用了描述角度的宏定义。若想让登录按钮旋转 45°，那么对应的弧度为：

```
    public var M_PI_4: Double { get } /* pi/4          */
```

rotationAngle 方法是以弧度为单位进行旋转的，这里需要特别注意，下面是旋转 45°时的动画效果，如图 2.6 所示。

### 2.2.8　位移动画：基于 UIView 的 transform 属性

位移动画的实现与旋转动画和缩放动画的实现类似，关键是对 transform 属性的认识。位移动画，顾名思义，就是修改登录按钮当前的位置，即登录按钮的 $x$、$y$ 坐标，不过与之前讲到的位置动画有所不同。在第 2.2.1 节中，修改 $x$ 值、$y$ 值直接反应在登录按钮的左上角位置。如图 2.7 所示。

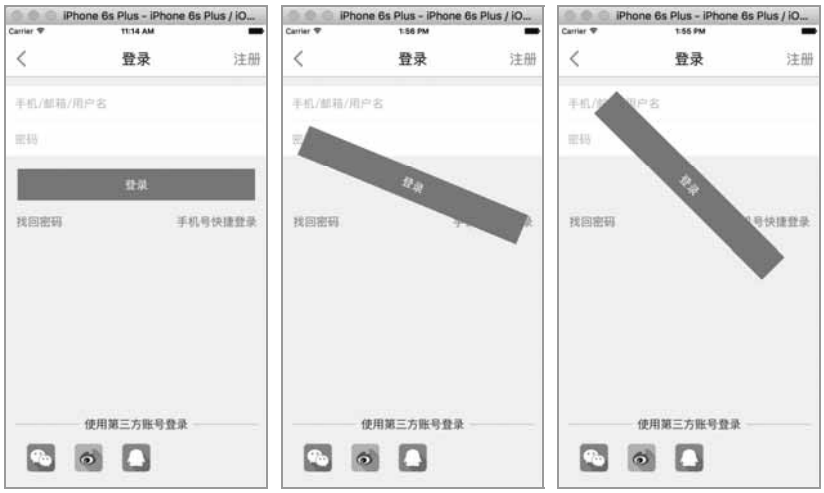



图 2.6　旋转动画效果


图 2.7　位置动画 *x*、*y* 位置

位移动画是设置当前的登录按钮相对于 *x*、*y* 轴移动了多少，比如想让登录按钮在 *x* 轴负方向移动 20，在 *y* 轴正方向移动 100。实现代码如下所示。

```
override func viewWillAppear(_ animated: Bool) {
//      CGAffineTransform:移动
1        UIView.beginAnimations(nil, context: nil)//动画开始
2        UIView.setAnimationDuration(1)//动画周期设置
3        loginButton!.transform =
```

```
            CGAffineTransform(translationX: 0, y: 300)
4           UIView.commitAnimations()//动画提交
}
```

代码最终运行效果如图 2.8 所示。


图 2.8　位移动画效果

## 2.2.9　组合动画效果

在掌握了以上几种常用的 UIView 初级动画效果的基础上，还可以将之灵活组合，以实现更为复杂和高级的效果。下面实现一个组合动画效果，如图 2.9 所示。


图 2.9　组合动画效果


图 2.9　组合动画效果（续）

根据图 2.10 所展示的动画演变过程，单击登录按钮之后，登录按钮一边旋转，一边缩放，同时向登录界面的右上角移动，在移动的同时按钮的透明度也在逐渐减弱。最后整个登录按钮消失在登录界面的右上角。下面来总结一下这个复杂的组合动画效果涉及哪些效果：

（1）旋转动画效果

（2）位置动画效果

（3）缩放动画效果

（4）淡入淡出动画效果

虽然这个组合动画看起来比较复杂，但经过层层剖析之后会发现，其实每个动画的知识点都是已经讲解过的，下面是具体代码实现方法。代码分为两个部分，一部分是实例化登录按钮，另一部分是添加登录按钮之后的响应方法。

```
1   var loginButton:UIButton?
2   override func viewDidLoad() {
3       super.viewDidLoad()
4       loginButton = UIButton(frame: CGRect(x:20,y:230,
```

```
                    width:self.view.frame.width-20*2,height:50))
    5       loginButton!.backgroundColor = UIColor(red: 50/255.0,
                    green: 185/255.0, blue: 170/255.0, alpha: 1.0)
    6       loginButton!.setTitle("登录", for:
                                          UIControlState.normal)
    7       loginButton!.addTarget(self,
                  action: #selector(ViewController.loginAction),
                  for: UIControlEvents.touchUpInside)
    8       self.view.addSubview(loginButton!)
        }
```

在 viewDidLoad()方法中实例化一个 UIButton，并将其添加在登录界面的合适位置（注意这里是以 6s plus 模拟器为例，对于其他型号的模拟器，需要适当调整登录按钮的位置坐标），代码第 7 行添加登录按钮响应方法。响应方法具体代码如下所示。

```
    func loginAction() {
    1       UIView.beginAnimations(nil, context: nil)//动画开始
    2       UIView.setAnimationDuration(2)//动画周期设置
    3       let angle:CGFloat = CGFloat(M_PI);
    4       loginButton!.transform = CGAffineTransform(rotationAngle:
angle)
    5  loginButton!.frame = CGRect(x: 400,y: 0,
                           width: loginButton!.frame.width*0.1,
                          height: loginButton!.frame.height*0.1)
    6       loginButton!.alpha = 0;
    7       UIView.commitAnimations()//动画提交
        }
```

代码第 1 行表明动画开始，第 2 行设置动画执行周期为 2s。第 3 行获取动画旋转时的角度（180°），代码从第 4 行到第 6 行分别设置动画旋转效果、位置、缩放效果和淡出效果。代码第 5 行采用 frame 属性将位置动画和缩放动画集成在一起。代码第 7 行为动画提交。可见这段代码并不难理解，只是把我们已经学过的知识进行灵活组合，所以在掌握了基础动画合集的基础上，通过简单的组合，可以形成各种炫酷的效果。

动画就是这么简单！

# 2.3 动画常用属性及回调方法的使用

## 2.3.1 动画常用属性的使用

在第 2.2.1 节位置动画中，有一行代码描述了 UIView 动画执行的周期：UIView.setAnimationDuration(1)。这行语句表明该动画的执行周期设置为 1s，其实在 iOS 动画中还有很多用于设置动画效果的常用属性。下面是常用的 UIView 动画属性设置方法。

（1）setAnimationDelay

（2）setAnimationCurve

（3）setAnimationsEnabled

（4）setAnimationDuration

（5）setAnimationRepeatAutoreverses

（6）setAnimationRepeatCount

setAnimationDelay 方法用于设置动画延迟执行时间间隔，例如 setAnimationDelay(1)，表明动画在启动之后，实际展示效果要等 1s 之后才能显示出来，该方法中传递的整型参数单位为秒。

setAnimationCurve 方法用于设置动画加速、减速效果。setAnimationCurve 传递的参数是一个枚举类型，有以下几种常用类型。

（1）UIViewAnimationCurve.EaseInOut

（2）UIViewAnimationCurve.EaseIn

（3）UIViewAnimationCurve.EaseOut

（4）UIViewAnimationCurve.Linear

EaseInOut 类型表明在动画开始和结束时都呈现减速效果，即开始和结束时动画执行速度较慢。EaseIn 类型表明在动画开始时呈现减速效果，即开始时动画执行速度较慢。EaseOut 类型表明在动画结束时呈现减速效果，即结束时动画执行速度较慢。Linear 类型表明动画在整个执行周期内速度一致，可以认为是匀速运动。

setAnimationsEnabled 方法用于设置动画是否使能，它有 true 和 false 两个参数。当设置为 false 时，动画效果禁止。当设置为 true 时，动画效果使能。

setAnimationDuration 方法用于设置动画执行时间周期，该方法中传递的整型参数单位为秒。例如设置 setAnimationDuration(1)表明动画执行周期为 1s。

setAnimationRepeatAutoreverses 方法用于设置动画是否有重复返回效果。例如实现一个位移动画，登录按钮沿 $y$ 轴正方向移动 300。当 setAnimationRepeatAutoreverses（true）设置为 true 时，效果如图 2.10 所示。


图 2.10　setAnimationRepeatAutoreverses 方法返回效果


图 2.10　setAnimationRepeatAutoreverses 方法返回效果（续）

setAnimationRepeatCount 方法用于设置动画重复执行次数。注意，这里如果将动画重复执行次数设置为 2，动画返回效果设置为 false 时，那么第二次动画并不是在达到图 2.11 中第四幅图的位置返回，而是重新从的第一幅图的位置处开始执行。

## 2.3.2　动画回调方法的使用

动画除了常用的属性设置之外，回调方法也经常使用。动画中常见的回调方法有动画开始时回调和动画结束时回调两种。而回调方法的设置也有两种情况，一种是使用 delegate 委托代理实现动画回调，另一种是通过 set 方法实现回调。先来看看第一种回调方法，

（1）delegate 回调方法

下面是具体实现代码。

```
override func viewWillAppear(_ animated: Bool) {
//      CGAffineTransform:缩放
1       UIView.beginAnimations(nil, context: nil)//动画开始
2        UIView.setAnimationDelegate(self)//设置回调对象
```

```
    3       UIView.setAnimationDuration(1)//动画周期设置
    4       loginButton!.transform = CGAffineTransform(scaleX: 0.7, y:
1.2)
    5       UIView.commitAnimations()//动画提交
    }
```

代码第 2 行设置 self 回调对象，表明动画的回调方法都在 self 对象中实现。下面是回调方法实现。这里想实现动画结束后的停止回调方法，所以重写 animationDidStop 方法，注意，这里是 override 重写，而不是创建一个新的方法。

```
       override func animationDidStop(anim: CAAnimation, finished
flag: Bool) { (swift2.3)
           print("animation stop!")
       }
```

当动画执行完之后就会触发该回调方法打印 log：animation stop!

（2）setAnimationDidStopSelector 自定义回调方法

除了通过 delegate 委托代理的方式设置回调方法之外，还可以使用 setAnimationDidStopSelector 自定义委托代理方法。下面将举例实现自定义委托代理方法。下面想实现一个缩放效果，在缩放效果动画结束之后，通过回调自定义的动画结束方法完成回调。具体实现代码如下：

```
    override func viewWillAppear(_animated: Bool) {
    //      CGAffineTransform:缩放
    1       UIView.beginAnimations(nil, context: nil)//动画开始
    2       UIView.setAnimationDelegate(self)//设置回调对象
    3       UIView.setAnimationDuration(1)//动画周期设置
    4       loginButton!.transform = CGAffineTransform(scaleX: 0.7,
                                                       y: 1.2)
    5       UIView.setAnimationDidStop(#selector
                                  (ViewController.animationEnd))
    6       UIView.commitAnimations()//动画提交
    }
```

在代码的第 5 行通过 setAnimationDidStop 方法设置了一个自定义的动画

回调方法 animationEnd。在自定义回调方法中打印一行 log，表明当前动画已经结束，下面是具体实现代码。动画结束后会打印一行 log 信息 AnimationEnd!

```
    func animationEnd(){
        print("AnimationEnd!")
    }
```

### 2.3.3 案例：抽奖转盘旋转动画效果的简单实现

在一些抽奖 APP 中经常会碰到抽奖转盘动画效果，这种动画效果实现起来比较复杂，在这里结合已经学习过的常用动画合集以及动画属性回调方法实现一个最简单的抽奖转盘连续转动效果。

在第 2.2.7 节介绍了使用 UIView 的 transform 属性实现旋转的效果，那可否利用到转盘上呢？这里将登录按钮替换为 UIImageView（承载一张抽奖转盘图），实现代码如下。

```
override func viewWillAppear(_ animated: Bool) {
//      CGAffineTransform:旋转
1       UIView.beginAnimations(nil, context: nil)//动画开始
2       UIView.setAnimationDelegate(self)
3       UIView.setAnimationDidStop(#selector
                                (ViewController.animationEnd))
4       UIView.setAnimationDuration(0.1)//动画周期设置
5       let angle:CGFloat = CGFloat(M_PI_2);
6       imageView!.transform =
                    CGAffineTransform(rotationAngle: angle)
7       UIView.commitAnimations()//动画提交
}
```

动画最终执行效果如图 2.11 所示。

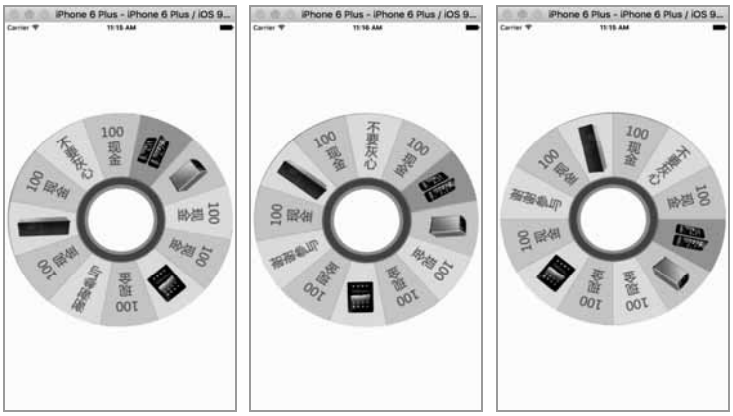


图 2.11　抽奖转盘滚动动画效果

在转盘滚动代码的第 5 行设置 90°滚动效果，如果想实现连续滚动是不是将角度设置大一点就可以了呢？比如滚动 10 圈，这里将滚动角度修改为：

```
let angle:CGFloat = CGFloat(M_PI_2*4*10);
```

事实并非如此美好，当角度改为 M_PI_2*4*10，换算成角度为 360*10，即表示 10 圈。Transform 认为 3600°刚好是 360°的整数倍，那么动画执行完之后又回到了原始状态，所以这时候 UIView 就认为既然起始状态是 0°，旋转之后动画的结束状态和起始状态一致，UIView 就不会进行任何旋转。

如何避免这一问题呢？可以结合第 2.3.2 节的方法，让转盘每次转动 90°，当转盘停下来时，会回调 stop 方法，在 stop 方法中可以让转盘继续旋转，下面是具体实现代码：

```
    func animationEnd() {
1       UIView.beginAnimations(nil, context: nil)//动画开始
2       UIView.setAnimationDelegate(self)
3       UIView.setAnimationDidStop(#selector
                              (ViewController.animationEnd))
4       UIView.setAnimationDuration(0.1)//动画周期设置
5       let angleStart:CGFloat = CGFloat(M_PI_2)
6       index += 1
7       let angle:CGFloat = CGFloat(index)*angleStart
8       imageView!.transform =
```

```
            CGAffineTransform(rotationAngle: angle)
9           UIView.commitAnimations()
        }
```

以上代码有几点需要注意：第一点，该动画设置了 delegate 回调对象。第二点，该动画设置动画执行周期为 0.1s。第三点，该动画每次旋转 90°。

该回调方法中会启动新一轮旋转动画效果，每一次回调，旋转角度都在原来的基础上增加 90°以达到继续旋转的效果。这里初始化 index 为 0，回调方法中每次自加 1 以实现旋转累计增加 90°的效果。

## 2.4 本章小结

本章结合 UIView 的基本属性，详细介绍了 UIView 显示层常用动画效果。并对位置动画、几何形状动画、淡入淡出动画、颜色渐变动画、旋转动画等动画效果做了全面详尽的分析。相信大家通过对本章的学习，今后再使用 UIView 动画合集中的效果时都能游刃有余。

在本章的最后，重点介绍了 UIView 动画的各种属性设置和回调方法。UIView 展示的主题效果受 UIView 动画合集中使用哪种类型的动画效果限制，而 UIView 展示的一些细节则受到动画属性限制。比如动画执行周期、动画运动加速效果、动画是否具有返回效果等。除了常用的动画属性设置之外，回调方法也非常重要。比如抽奖转盘旋转动画，就是将动画的旋转效果结合动画结束回调方法的一个典型应用。

# 第 3 章　显示层关键帧动画

## 本章内容

- 理解关键帧动画的实现原理
- 掌握动画中常用关键帧的设置

iOS 中的动画可以分为很多种类型，根据图层显示可以分为隐式动画和显示动画。根据动画在 iOS 中实现的“位置”可以分为显示层（UIView）动画和内容层（Layer）动画。在第 2 章中为大家详细介绍了 UIView 显示层动画合集，以及 UIView 图层中各种各样的初级动画效果。本章将为大家介绍 UIView 层另一种动画效果：关键帧动画。关键帧动画主要使用在通过动画的几张关键图片描述整个动画的效果的情况下。

## 3.1　关键帧动画实现原理

关键帧动画的实现与 UIView 动画合集中提到的动画效果有一些不同，在 UIView 动画合集中都是通过修改当前 UI 控件的各种属性来实现想要的动画效果，而关键帧动画只需要设置动画的几个关键的显示帧。

下面为大家举一个例子，如图 3.1 所示是飞机降落的动画效果。

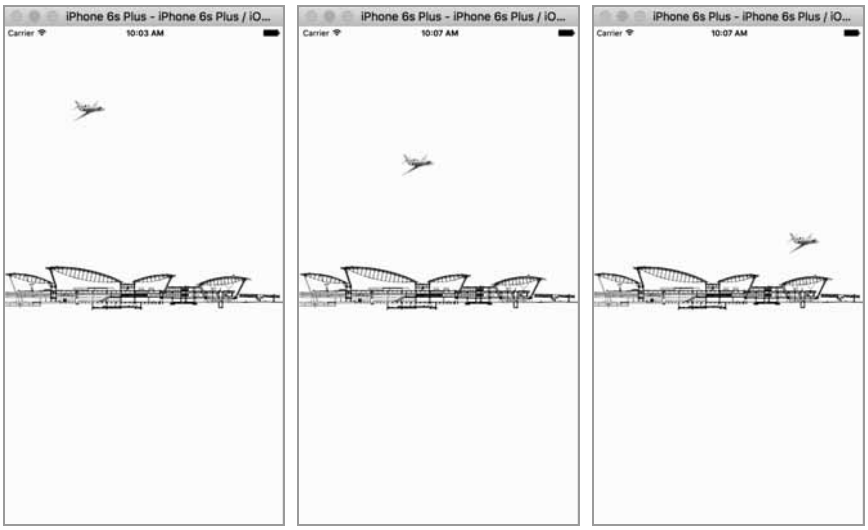



图 3.1　飞机降落动画效果

该动画效果描述了飞机从机场的左上方降落到机场的右下方。使用 UIView 基础动画合集中的效果可以实现这一动画，不过这里将采用关键帧动画来实现，并实现一些对动画更为精确的控制。假设当前动画运动的位置坐标是从（100，100）位置运动到（300，300）位置，整个动画持续时间为 2s。那么就可以在 0s 的时候添加第一帧（起始帧），飞机位置（100，100）作为动画的起始帧。然后在 2s 的时候添加关键帧（或者叫结束帧），飞机位置（300，300）。那么飞机根据这两帧就可以实现降落的运动效果。

添加起始帧和结束帧可以基本完成飞机降落的动画效果，但是要想实现更为精确的控制就必须再多添加几帧。比如飞机在 0.5s 时，控制飞机运动到（150，175）的位置处。可以在动画的 0.5s 时添加一个关键帧，关键帧的内容描述当前飞机运动到（150，175）处。

## 3.2　案例：关键帧动画之飞机降落

如图 3.2 所示，这里为大家准备了两张图，左图描述当前机场，右图描述当前的飞机。

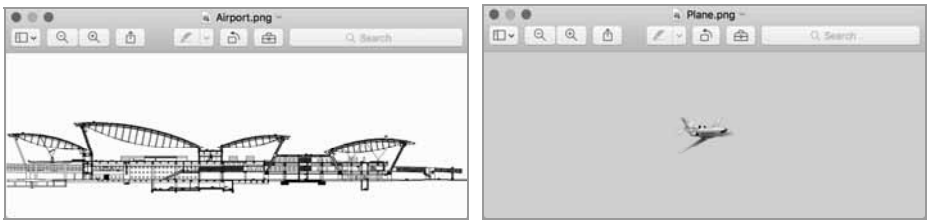



图 3.2　关键帧动画素材图片

下面是机场背景的具体实现代码。

```
1    var imageViewAirport:UIImageView?
2    var imageViewPlane:UIImageView?
3    override func viewDidLoad() {
4        super.viewDidLoad()
5        imageViewAirport = UIImageView()
6        imageViewAirport?.frame = UIScreen.mainScreen().bounds
7        imageViewAirport?.image = UIImage(named: "Airport.png")
8        imageViewAirport?.contentMode    =    UIViewContentMode.
ScaleAspectFit
9        self.view.addSubview(imageViewAirport!)
}
```

代码第 1 行和第 2 行初始化了两个 UIImageView 的实例对象，一个用于加载机场背景，另一个用于加载当前飞机。代码第 5 行实例化 UIImageView 实例对象。第 6 行设置当前飞机场图片大小为整个屏幕大小。第 7 行添加飞机场背景图片。第 8 行调整图片合适的缩放比例。第 9 行将当前的 UIImageView 实例对象添加到 self.view 上。

下面是飞机首帧实现代码。

```
override func viewDidLoad() {
        super.viewDidLoad()
        //机场背景代码…
1       imageViewPlane = UIImageView()
2       imageViewPlane?.frame = CGRect(x: 100, y: 100, width: 50,
                                                    height: 50)
3       imageViewPlane?.image = UIImage(named: "Plane.png")
4       imageViewPlane?.contentMode =
```

```
                                UIViewContentMode.ScaleAspectFit
5       imageViewAirport!.addSubview(imageViewPlane!)
}
```

代码第 1 行实例化 UIImageView 实例对象。第 2 行设置飞机的初始位置和大小。第 3 行添加飞机图片。第 4 行设置飞机图片合适的缩放比例。第 5 行将飞机的实例对象 UIImageView 添加到 self.view 上。

下面是飞机结束帧的实现代码。

```
override func viewWillAppear(_ animated: Bool) {
UIView.animateKeyframes(withDuration: 2, delay: 0,
   options:UIViewKeyframeAnimationOptions.calculationModeCubic,
   animations: {() in
      self.imageViewPlane?.frame = CGRect(x: 300, y: 300,
      width: 50, height: 50)
    }, completion:{(finish) in
      print("done")
    })
}
```

为了实现应用启动后动画效果立即展现，在 viewWillAppear()方法中添加结束帧代码。结束帧实现的核心方法如下。

```
open class func animateKeyframes (
withDuration duration: TimeInterval,
delay: TimeInterval,
options: UIViewKeyframeAnimationOptions = [],
animations: @escaping () -> Swift.Void, completion: (@escaping
(Bool) -> Swift.Void)? = nil)
```

该方法的作用是为飞机添加关键帧。几个参数作用分别如下。

（1）duration：描述动画执行周期。

（2）delay：描述动画延迟执行时间。

（3）options：描述动画执行效果。

（4）animations：关键帧动画添加处。

（5）completion：动画完成回调

在本节设置动画 0s 延迟，动画执行周期为 2s。动画执行效果为一枚举类型，这里设置为 CalculationModeCubic。常见的效果还有下面几类。

```
        public static var CalculationModeLinear: UIViewKeyframe
AnimationOptions { get } // default
        public static var CalculationModeDiscrete: UIViewKeyframe
AnimationOptions { get }
        public static var CalculationModePaced: UIViewKeyframe
AnimationOptions { get }
        public static var CalculationModeCubic: UIViewKeyframe
AnimationOptions { get }
    public static var CalculationModeCubicPaced:
    UIViewKeyframeAnimationOptions { get }
```

CalculationModeLinear 是默认情况下使用的动画效果。该效果表明动画按照匀速线性执行。那么其他几种又是什么意思呢？这里为大家准备了一张图，来描述其他几种动画运动效果，如图 3.3 所示。

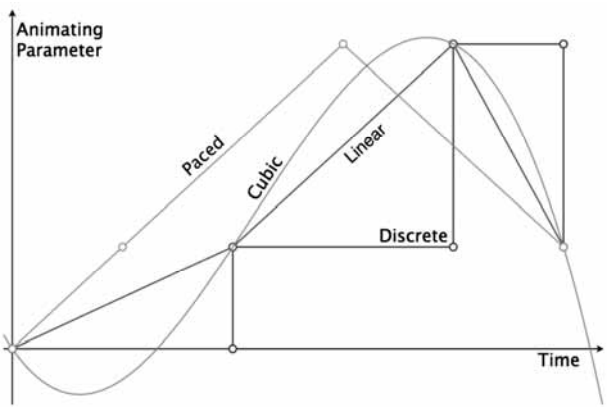



图 3.3　动画运动效果图

在图 3.3 中描绘了不同的动画效果，以及速度随着时间的变化关系，图中 Linear 线段表明了动画的匀速运动效果。

结束帧处的动画最终位置设置在（300，300）处，动画执行完毕之后打印 log（done）。下面是动画最终执行效果，如图 3.4 所示。

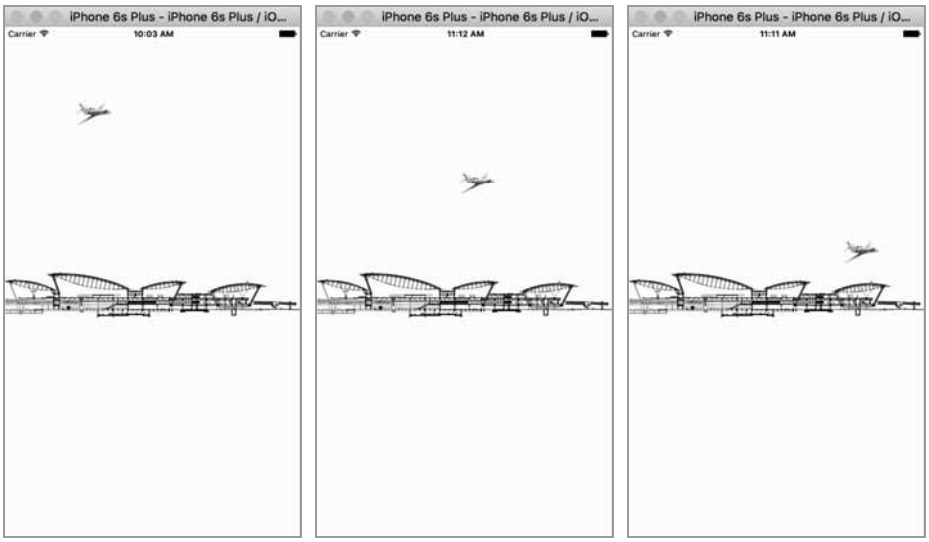



图 3.4　关键帧动画效果

虽然通过上面的代码可以实现飞机的运动效果，但有时候还想实现更为细腻的效果和更复杂的路径。在这种情况下关键帧动画就可以大显身手了，将viewWillAppera()代码修改为下面的形式。

```
override func viewWillAppear(_ animated: Bool) {
1    UIView.animateKeyframes(withDuration: 2, delay: 0, options:
       UIViewKeyframeAnimationOptions.calculationModeCubic,
       animations: {() in
2          UIView.addKeyframe(withRelativeStartTime: 0,
               relativeDuration: 1/2, animations: {() in
               self.imageViewPlane?.frame = CGRect(x: 300, y: 100,
                                          width: 30, height: 30)
               })
3          UIView.addKeyframe(withRelativeStartTime: 1/2,
               relativeDuration: 1/2, animations: {() in
               self.imageViewPlane?.frame = CGRect(x: 300, y: 300,
                                         width: 100, height: 100)
               })
           }, completion:{(finish) in
               print("done")
```

```
            })
        }
```

在这段代码中添加了一个新的方法：

```
open class func addKeyframe (
withRelativeStartTime frameStartTime: Double,
relativeDuration frameDuration: Double,
animations: @escaping () -> Swift.Void)
```

该方法描述了在什么位置，添加一个持续时间为多长的关键帧。此方法的几个参数如下。

（1）frameStartTime：关键帧起始时间。

（2）relativeDuration：关键帧相对持续时间。

（3）animations：关键帧具体实现内容。

代码第一帧表明当前起始帧的初始时间为 0s，持续时间为整个动画周期（2s）的 1/2。动画效果为飞机水平飞到（300，100）处，因为飞机飞离机场越来越远，所以这里将飞机的大小设置为（30，30）。代码第二帧表明当前关键帧起始时间为 1s（即整个动画周期 2s 的 1/2）。该关键帧持续时间为 1s。动画执行效果为垂直俯冲下来，并逐渐变大。如图 3.5 所示为最终实现效果。

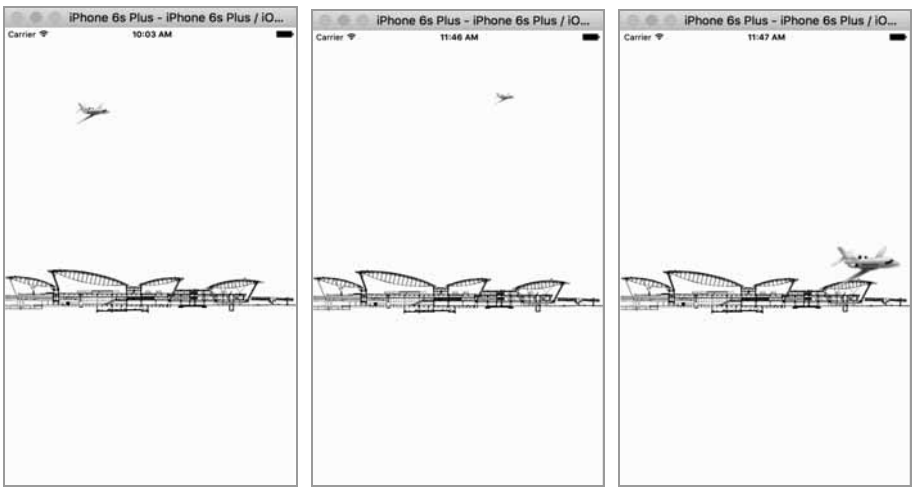



图 3.5　自定义关键帧动画效果

## 3.3 案例：关键帧动画之抽奖转盘滚动

在第 3.2 节利用 UIView 的 transform 属性实现了抽奖转盘无限循环滚动的效果，那么在本小节将利用关键帧动画来实现这一效果。下面是具体实现代码。

```
    var imageView:UIImageView?
    var index:Int = 0
    override func viewDidLoad() {
1       super.viewDidLoad()
2       imageView = UIImageView()
3       imageView?.frame = UIScreen.mainScreen().bounds
4       imageView?.image = UIImage(named: "turntable.png")
5       imageView?.contentMode =
                              UIViewContentMode.ScaleAspectFit
6       self.view.addSubview(imageView!)
7       animationCircle()
    }
```

该段代码实现转盘图片的加载。代码第 2 行实例化一个 UIImageView 实例对象，第 3 行初始化 UIImageView 的位置、大小。第 4 行为 UIImageView 添加一个转盘图片。第 5 行设置图片的拉伸方式。第 6 行将当前实例化的 UIImageView 实例对象添加到 self.view 上。最后启动转盘循环转动方法。转盘循环转动方法如下所示：

```
    func animationCircle() {
    UIView.animateKeyframes(withDuration: 0.2, delay: 0, options:
       UIViewKeyframeAnimationOptions(), animations: {() in
       UIView.addKeyframe(withRelativeStartTime: 0,
         relativeDuration: 1/4, animations: {() in
         self.index += 1
         let angle:CGFloat = CGFloat(M_PI_2)*CGFloat(self.index)
         self.imageView?.transform =
                        CGAffineTransform(rotationAngle: angle)
       })
       UIView.addKeyframe(withRelativeStartTime: 0,
```

```
        relativeDuration: 1/4, animations: {() in
        self.index += 1
        let angle:CGFloat = CGFloat(M_PI_2)*CGFloat(self.index)
        self.imageView?.transform =
                        CGAffineTransform(rotationAngle: angle)
      })
      UIView.addKeyframe(withRelativeStartTime: 2/4,
        relativeDuration: 1/4, animations: {() in
        self.index += 1
        let angle:CGFloat = CGFloat(M_PI_2)*CGFloat(self.index)
        self.imageView?.transform =
                        CGAffineTransform(rotationAngle: angle)
      })
      UIView.addKeyframe(withRelativeStartTime: 3/4,
        relativeDuration: 1/4, animations: {() in
        self.index += 1
        let angle:CGFloat = CGFloat(M_PI_2)*CGFloat(self.index)
        self.imageView?.transform =
                        CGAffineTransform(rotationAngle: angle)
      })
    }, completion:{(finish) in
       self.animationCircle()
    })
  }
```

该段代码创建一个周期为 0.2s、旋转速度为匀速的关键帧动画。在关键帧动画的实现中，添加了 4 个关键帧，分别在 0s、0.05s、0.1s、0.15s 时添加一张关键帧，每帧动画周期为 0.05s。每帧动画功能在上一帧的基础上旋转 90°实现。代码中设置了一个 index 标示符，该标示符负责实现每帧动画旋转角度 90°递增。

## 3.4　本章小结

在本章中为大家介绍了一种利用动画的关键帧实现的动画方式。关键帧动

画与之前介绍的动画实现方式有很大的不同。前两章介绍的动画，只需要改变相应的 UIView 属性即可实现相应的动画效果，而关键帧动画只需要在合适的位置设置一帧关键的 UIView 即可。比如飞机降落动画，只需要在动画降落的几个关键时刻点设置飞机的位置，并根据飞机距离机场的远近设置飞机的大小即可。

用一句话可以概括 UIView 动画和关键帧动画的特点：UIView 动画改变的是动画的属性，而关键帧动画需要设置动画在几个关键时刻的 UIView。

# 第 4 章　显示层逐帧动画

## 本章内容

- 理解逐帧动画的实现原理
- 掌握使用定时器、CADisplayLink、Draw 等方法实现逐帧动画

在前面三章给大家介绍了 UIView 基础动画合集和逐帧动画，其基本原理都是通过在某个时刻设置动画的关键属性或帧来实现的，并非动画的逐帧显示。本章将结合定时器和 CADisplayLink 实现逐帧动画效果。

## 4.1　逐帧动画实现原理

根据字面意思不难理解，逐帧动画实现的动画效果就是将图片一帧帧地逐帧渲染，所以首先需要准备逐帧动画实现的素材。如图 4.1 所示是描述飞机飞行过程中 67 个瞬间的静态图片。

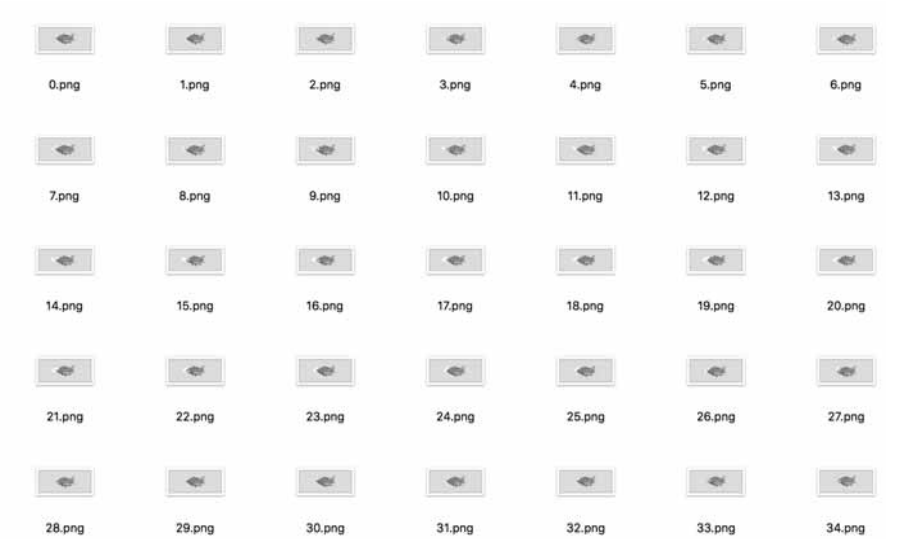



图 4.1　飞机飞行过程中 67 个瞬间的静态图片（此处仅列出 34 张图片，足以表达分帧效果）

将图片序列按照飞机飞行的连续时间状态在很短的时间内依次展示出来，即可实现动画的逐帧展现效果。

## 4.2　基于 NSTimer 的逐帧动画效果

如何让图片按照连续的顺序和一定的时间间隔显示图片呢？这里为大家介绍两种逐帧刷新方法，一种是基于定时器的逐帧刷新，这种方法经常使用在动画帧率不高，且帧率之间的时间间隔并不十分严格的情况下。另一种是基于 CADisplayLink 的帧刷新效果，该方法刷帧频率高达每秒 60 帧，且非常精确。这里先为大家介绍基于定时器的逐帧动画效果，以下是具体实现代码。

```
1   var imageView:UIImageView?
2   var timer:NSTimer?
3   var index:Int = 0
4   override func viewDidLoad() {
5       super.viewDidLoad()
6       imageView = UIImageView()
7       imageView?.frame = UIScreen.mainScreen().bounds
```

```
8       imageView?.contentMode =
                        UIViewContentMode.ScaleAspectFit
9       self.view.addSubview(imageView!)
10      index = 0
11      timer = Timer.scheduledTimer(timeInterval: 0.1,
              target: self,
              selector: #selector(ViewController.refushImage),
              userInfo:nil,
              repeats: true)
    }
```

代码第 1 行到第 3 行实例化三个实例对象，分别是 UIImageView（承载飞机图片）、timer（承载动画的周期执行）、index（承载图片顺序）。代码第 6 行到第 8 行实例化 UIImageView 实例对象，设置 UIImageView 的位置、大小以及图片拉伸方式。代码第 9 行将 UIImageView 添加到 self.view 上。第 10 行初始化图片序号 index 变量。第 11 行初始化一个定时器，设置定时器执行周期为 0.05s，定时器响应对象为当前 self，响应方法为 refushImage()，定时器可以重复执行。定时器响应方法如下所示。

```
func refushImage(){
1       imageView?.image = UIImage(named: "\(index).png")
2       index++
3       if(index == 67){
4           timer?.invalidate()
5           index--
6           imageView?.image = UIImage(named: "\(index).png")
        }
    }
```

代码第 1 行到第 2 行根据 index 序号加载相应的图片并添加到 UIImageView 实例对象上，实现图片序号递增。第 3 行判断动画是否已经执行到最后一张，如果已经执行到最后一张，那么定时器停止，并将动画定格在最后一张。

在定时器的响应方法中，通过设置 index 可以灵活设置动画的效果。比如想实现动画无限重复执行，那么可以将定时器响应方法修改如下。

```
func refushImage(){
    imageView?.image = UIImage(named: "\(index).png")
    index++
    if(index == 67){
        index = 0;
    }
}
```

通过定时器实现的逐帧动画效果如图 4.2 所示。

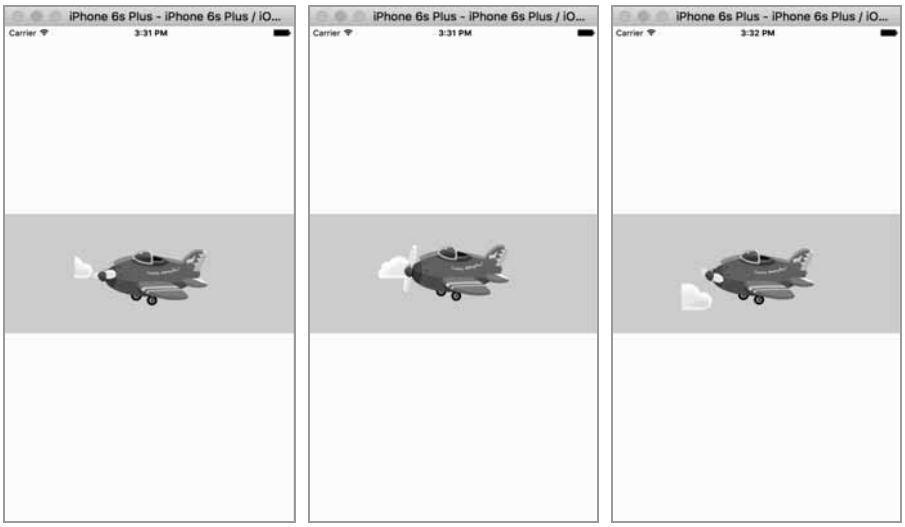



图 4.2　逐帧动画效果

## 4.3　基于 CADisplayLink 的逐帧动画效果

采用定时器实现逐帧动画效果主要适用于相邻两帧之间时间间隔较长或者间隔不均匀的图片序列，但是如果动画帧率超过了每秒 60 帧，或者要求动画播放帧率之间的间隔均匀，那么 CADisplayLink 类的优越性就体现出来了。

CADisplayLink 和 NSTimer 有什么区别呢？

iOS 设备的屏幕刷新频率默认是 60Hz，而 CADisplayLink 可以保持和屏幕刷新率相同的频率将内容渲染到屏幕上，因此它的精度非常高。CADisplayLink

在使用的时候需要注册到 runloop 中，每当刷帧频率达到的时候 runloop 就会向 CADisplayLink 指定的 target 发送一次指定的 selector 消息，相应的 selector 中的方法会被调用一次。下面是采用 CADisplayLink 方法实现的逐帧动画代码。

```
override func viewDidLoad() {
1       super.viewDidLoad()
2       imageView = UIImageView()
3       imageView?.frame = UIScreen.mainScreen().bounds
4       imageView?.contentMode =
                            UIViewContentMode.ScaleAspectFit
5       self.view.addSubview(imageView!)
6       index = 0
7       displaylink = CADisplayLink.init(target: self, selector:
                      #selector(ViewController.refushImage))
8       displaylink?.frameInterval = 1
9       displaylink?.add(to: RunLoop.current,
                      forMode: RunLoopMode.defaultRunLoopMode)
    }
```

代码第 7 七行实例化一个 CADisplayLink 实例对象，设置当前刷帧响应对象以及 refushImage 刷帧响应方法。第 8 行设置刷帧间隔。当 frameInterval 设置为 1 时表示当前的刷帧周期为 1/60s。当 frameInterval 设置为 2 时表示刷帧周期为（1/60）×2。代码第 9 行将当前 displaylink 实例对象添加到 Runloop 中，启动刷帧事件。最终效果如上图 4.2 所示。

## 4.4　基于 draw 方法的逐帧动画效果

在 UIView 中还有一个非常重要的方法：draw()方法。当创建一个新的 View 时，其自动生成了一个 draw()方法，且此方法可以被重写，一旦 draw()方法被调用，Cocoa 就会为我们创建一个图形上下文，在图形上下文中的所有操作最终都会反应在当前的 UIView 界面上。按照这个思路，如果定期调用 draw()方法绘制新的内容，那么就可以实现逐帧动画的效果。

那么 draw()什么时候被调用呢？下面总结一下 draw()触发的机制。

（1）使用 addSubview 会触发 layoutSubviews。

（2）使用 view 的 frame 属性会触发 layoutSubviews（frame 更新）。

（3）直接调用 setLayoutSubviews 方法会触发 layoutSubviews。

如果想实现一个黑洞增长效果，可以预先准备好几张黑洞图片，通过 NSTimer 或者 CADisplayLink 逐帧渲染来实现，但是此方法的缺点是要提前准备好图片。本小节利用 draw()方法，可以把黑洞绘制到 UIView 上，实现逐帧动画的效果。以下是具体实现代码。

```
class BlackHoleView: UIView {
1    var blackHoleRadius:Float = 0
2    func blackHoleIncrease(_ radius: Float){
3        blackHoleRadius = radius
4        self.setNeedsDisplay()
     }
5    override func draw(_ rect: CGRect) {
6        let ctx:CGContextRef = UIGraphicsGetCurrentContext()!
7        ctx.addArc(
             center: CGPoint(x:self.center.x,y:self.center.y),
             radius: CGFloat(blackHoleRadius),
             startAngle: 0,
             endAngle: CGFloat(M_PI * 2),
             clockwise: false)
8        CGContextFillPath(ctx);
     }
}
```

代码第 1 行初始化一个描述黑洞半径的浮点型参数。第 2 行定义一个公开的黑洞半径增长方法，在该方法中获取当前黑洞半径，并调用 setNeedDisplay()方法实现 draw()方法的调用。代码第 5 行重写 draw()方法。第 6 六行获取当前绘图上下文。每次 draw()方法被调用的时候都会得到一个当前绘图上下文，所有在上下文中的操作都会反应到 UIView 视图上。第 7 行调用 addArc 方法绘制

圆形，来表示黑洞。该方法参数描述如下。

```
public func addArc (
 center: CGPoint,
 radius: CGFloat,
 startAngle: CGFloat,
 endAngle: CGFloat,
 clockwise: Int32)
```

（1）center: CGPoint：表明当前绘制圆形中心点的 *x*、*y* 坐标。

（2）radius: CGFloat：表明当前绘制圆形半径。

（3）startAngle: CGFloat：表明当前绘制圆形开始角度。

（4）endAngle: CGFloat：表明当前绘制圆形结束角度。通过合理设置开始、结束的角度还可以绘制扇形。

（5）clockwise: Int32：如果该值为 true，则表示顺时针绘制，为 false 则表示逆时针绘制。

代码最后一行开始绘制圆形。以上是 BlackHoleView 类实现的主要代码。下面是 ViewController 中实现的代码。

```
var blackHole:BlackHoleView?
1    var timer:NSTimer?
2    var index:Float = 0
3    override func viewDidLoad() {
4        super.viewDidLoad()
5        blackHole = BlackHoleView()
6        blackHole?.frame = UIScreen.mainScreen().bounds
7        blackHole?.backgroundColor = UIColor.cyanColor()
8        self.view.addSubview(blackHole!)
9        index = 0
10       timer = Timer.scheduledTimer(timeInterval: 1.0/30,
              target: self, selector: #selector(ViewController.
                  refushImage), userInfo:nil, repeats: true)
     }
```

```
11    func refushImage(){
        blackHole?.blackHoleIncrease(index)
        index += 1
      }
```

代码第 1 行定义一个定时器，用来实现 blackHoleIncrease()方法的实时调用。第 2 行初始化 index，表明黑洞的半径。每次定时时间到半径加 1。第 5 行实例化黑洞类对象。第 6 行和第 7 行设置黑洞类的背景颜色和 frame 大小。第 8 行将黑洞实例对象（UIView 子类）添加到 self.view 上。第 9 行初始化时黑洞半径为 0。第 10 行设置定时器，定时周期为 1.0/30s，响应对象为 self，响应方法设置为 refushImage，定时器循环执行标示符设置为 true。第 11 行响应方法中实现 blackHoleIncrease()以及黑洞半径增加。基于 draw()方法实现逐帧动画显示效果如图 4.3 所示。

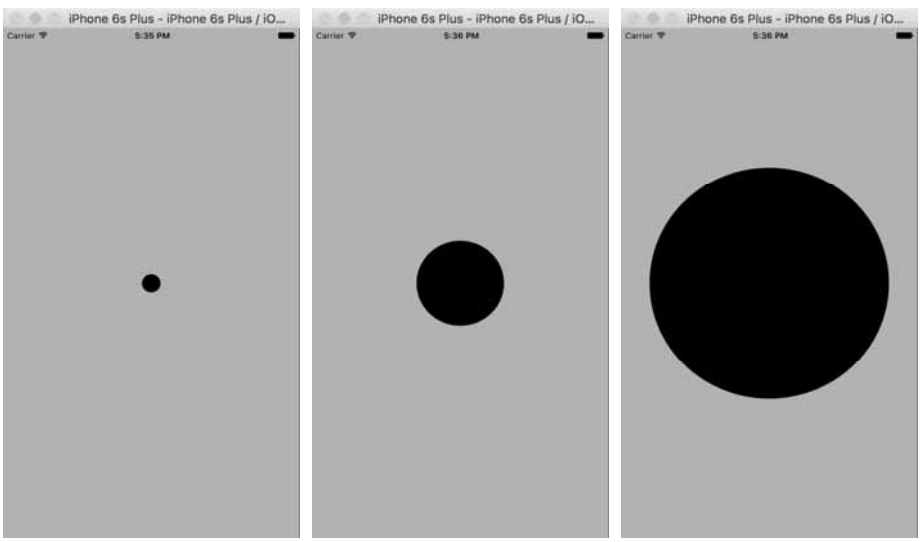



图 4.3　基于 draw()方法实现逐帧动画显示效果

## 4.5　本章小结

本章为大家介绍了逐帧动画的概念，并利用定时器、CADisplayLink、draw()重绘等方式为大家实现了逐帧动画的效果。由上述分析和测试可以得出：

CADisplayLink 精度很高，可以用于实现一些频率较高、帧率要求严格的动画效果。

draw()是 UIView 中重绘的重要方法，在 draw()方法中，对上下文的修改都将直接反应在 UIView 视图上，因此可以通过定期修改 draw()中的内容也可以实现逐帧动画的效果，而且这种动画不需要事先准备大量的素材，可用性较好。

# 第 5 章　GIF 动画效果

## 本章内容

- 掌握 GIF 图像的分解与合成
- 掌握 iOS 下 GIF 图片的展示

目前 iOS 还无法支持原生展现 GIF 图片，市场上支持 GIF 的系统也没有像 JPG、PNG 等图像格式支持得这么全面。因此掌握 GIF 图片的相关分解、合成及展现技术对于开发人员处理各种动画展示效果，以及图片类处理应用有着很实用的价值。在正式进入本章之前先来思考如下几个问题。

（1）GIF 是什么？

（2）GIF 有哪些特点？

（3）GIF 在 iOS 中有哪些使用场景？

（4）GIF 在 iOS 中如何使用？

## 5.1　GIF 图片初识

GIF（Graphics Interchange Format）的原义是“图像互换格式”，是 CompuServe 公司在 1987 年开发的图像文件格式，与常见的静态图像存储格式 JPG、PNG 等类似，GIF 是一种常用于动态图片的储存格式。如图 5.1 所示。


图 5.1　常见图片类型

需要明确的第一点是：GIF 是一种常用于动画效果的图片格式。

## 5.2　GIF 有什么特点

常见的 GIF 图片都是以“文件名.gif”后缀结尾，那么 Gif 图片有什么特点呢？在现阶段使用 GIF 最多的场景就是动画展示，即把一组序列图片存储到 GIF 文件中，然后利用相应的播放器播放出来就能形成动画展示效果。GIF 图片形成的过程如图 5.2 所示。

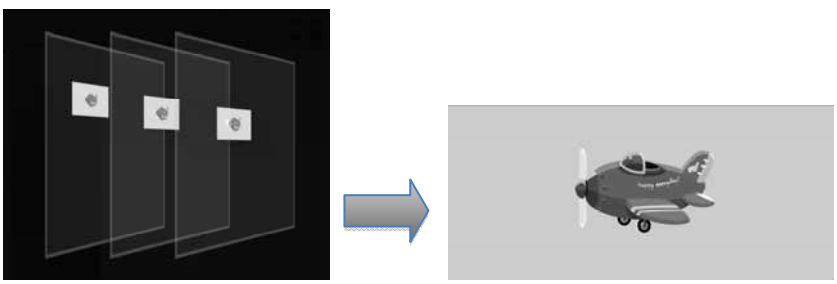


图 5.2　GIF 图片形成过程

通过图 5.2 可以看出，GIF 图片的实质就是由多帧静态图像按照动作发生的时间连续顺序组成图片序列整体。所以现在来回答 GIF 有哪些特点：可以通过单帧图像合成 GIF 或者对 GIF 进行单帧分解。

## 5.3　GIF 在 iOS 中的使用场景

GIF 在 iOS 中的使用场景有以下三个方面。

（1）GIF 图片分解为单帧图片。

（2）一系列单帧图片合成 GIF 图片。

（3）iOS 系统上展示 GIF 动画效果。

在 GIF 的合成和分解方面将会接触到 iOS 图像处理核心框架 ImageIO，作为 iOS 系统中图像处理的核心框架，它为我们提供了各种丰富的 API，本章将要实现的 GIF 分解与合成功能，通过 ImageIO 就可以很方便地实现。GIF 动画展示效果将结合 UIImageView 和定时器，利用逐帧展示的方式为大家呈现 GIF 动画效果。

## 5.4 GIF 分解单帧图片

### 5.4.1 GIF 图片分解过程

GIF 分解为单帧图片的过程如图 5.3 所示。

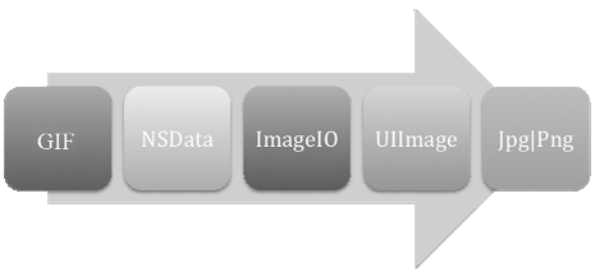



图 5.3 GIF 图片分解过程

整个过程划分为 5 个模块、4 个过程，分别如下。

（1）本地读取 GIF 图片，将其转换为 NSdata 数据类型。

（2）将 NSData 作为 ImageIO 模块的输入。

（3）获取 ImageIO 的输出数据：UIImage。

（4）将获取到的 UIImage 数据存储为 JPG 或者 PNG 格式保存到本地。

在整个 GIF 图片分解的过程中，ImageIO 是处理过程的核心部分。它负责对 GIF 文件格式进行解析，并将解析之后的数据转换为一帧帧图片输出。幸运的是我们并不是“轮子”的创造者，而是只要使用轮子即可。所以在本书中我们不去研究 GIF 分解合成算法的具体实现方式，而是将注意力聚焦在如何使用 ImageIO 框架实现需要的功能上。

## 5.4.2　GIF 图片分解代码实现

在正式分析代码之前，先来看看整个工程的文件结构，如图 5.4 所示。

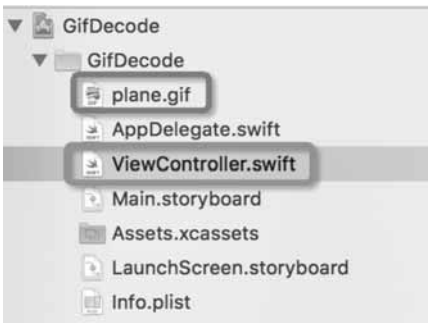



图 5.4　GIF 图片分解工程文件结构

源文件使用的是 plane.gif 文件。ViewController.swift 文件中的 viewDidLoad() 方法中包含了 GIF 图片分解为单帧图片并保存到本地的所有代码。下面就结合“图 5.3 GIF 图片分解过程”来实现这一功能。

功能模块一：读取 GIF 文件并将之转换为 NSdata 类型。

```
    1    let gifPath:NSString = Bundle.main.path(forResource: "plane",
ofType: "gif")! as NSString
    2    let gifData:Data = try! Data(contentsOf: URL(fileURLWithPath:
gifPath as String))
```

代码第 1 行通过 path 方法获取文件名为 plane、文件格式为 gif 的文件地址。第 2 行获取文件信息并加载到 gifData（NSData 类型）变量中。至此已经完成整个处理流程的第一个环节，如图 5.5 所示。


图 5.5　GIF 处理流程示意图（第一个环节）

功能模块二：利用 ImageIO 框架，遍历所有 GIF 子帧。需要注意的是使用 ImageIO 必须把读取到的 NSdata 数据转换为 ImageIO 可以处理的数据类型，这里使用 CGImageSourceRef 实现。其相应功能模块的处理流程如图 5.6 所示。

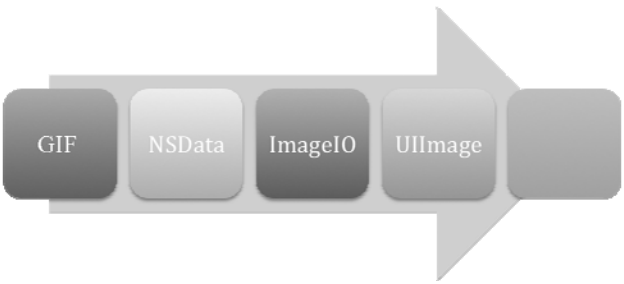



图 5.6　GIF 处理流程示意图

```
1   let gifDataSource:CGImageSource =
        CGImageSourceCreateWithData(gifData as CFData, nil)!
2   let gifImageCount:Int =
                        CGImageSourceGetCount(gifDataSource)
3    for i in 0...gifImageCount-1{
        let imageref:CGImage? =
       CGImageSourceCreateImageAtIndex(gifDataSource, i, nil)
        let image:UIImage = UIImage(cgImage: imageref!,
                        scale:UIScreen.main.scale,
                        orientation:UIImageOrientation.up )
      }
```

下面是 GIF 数据处理流程中 ImageIO 部分功能描述。代码第 1 行实现将 GIF 原始数据类型 NSdata 转换为 ImageIO 可以直接处理的数据类型

CGImageSourceRef。第 2 行获取当前 GIF 图片的分帧个数。我们知道 GIF 图片都是由一帧帧图片组成的，那么这一行就是为了获取构成 GIF 图片的张数。第 3 行对 CGImageSource 数据按照图片的序号进行遍历，将遍历出的结果使用 UIImage 系统方法将之转换为 UIImage。

这里重点为大家介绍两种方法。

CGImageSourceCreateImageAtIndex 方法的作用是返回 GIF 中其中某一帧图像的 CGImage 类型数据。该方法有三个参数，参数 1 为 GIF 原始数据，参数 2 为 GIF 子帧中的序号（该序号从 0 开始），参数 3 为 GIF 数据提取的一些选择参数，因为这里不是很常用，所以设置为 nil。

```
    public func CGImageSourceCreateImageAtIndex(_ isrc: CGImageSource,
_ index: Int, _ options: CFDictionary?) -> CGImage?
```

以下为 UIImage 类的方法，这个方法用于实例化 UIImage 实例对象。该方法有三个参数，参数 1 为需要构建 UIImage 的内容，注意这里的内容是 CGImage 类型，参数 2 为手机物理像素与手机和手机显示分辨率的换算系数，参数 3 表明构建的 UIImage 的图像方向。通过这个方法就可以在某种手机分辨率下构建指定方向的图像，当然图像的类型是 UIImage 类型。

```
    public init(CGImage cgImage: CGImage, scale: CGFloat, orientation:
UIImageOrientation)
```

通过上述两步已经获取了 UIImage，然而 UIImage 并不是通常我们看到的图像格式，此图像格式最大的特点是无法存储为本地可以查看的图片格式，因此如果需要将图像保存在本地，就需要在这之前将已经得到的 UIImage 数据类型转换为 PNG 或者 JPG 类型的图像数据，然后才能把图像存储到本地。

下面是完整的 GIF 图像分解保存代码：

```
    override func viewDidLoad() {
    1       super.viewDidLoad()
    2       let gifPath:NSString = Bundle.main.path(forResource:
                                "plane", ofType: "gif")! as NSString
    3       let gifData:Data = try! Data(contentsOf:
```

```
                        URL(fileURLWithPath: gifPath as String))
4       let gifDataSource:CGImageSource =
            CGImageSourceCreateWithData(gifData as CFData, nil)!
5       let gifImageCount:Int =
                          CGImageSourceGetCount(gifDataSource)
6       for i in 0...gifImageCount-1{
7          let imageref:CGImage? =
                       CGImageSourceCreateImageAtIndex(
                                    gifDataSource, i, nil)
8          let image:UIImage = UIImage(cgImage: imageref!,
                        scale:UIScreen.main.scale,
                        orientation:UIImageOrientation.up )
9          let imageData:Data = UIImagePNGRepresentation(image)!
10         var docs=NSSearchPathForDirectoriesInDomains(
                    .documentDirectory, .userDomainMask, true)
11         let documentsDirectory = docs[0] as String
12         let imagePath = documentsDirectory+"/\(i)"+".png"
13         try? imageData .write(to: URL(fileURLWithPath:
                              imagePath), options: [.atomic])
14         print("\(imagePath)")

       }
   }
```

代码第 1 行使用 UIImagePNGRepresentation 方法将 UIImage 数据类型存储为 PNG 格式的 data 数据类型，第 2 行代码和第 3 行代码获取应用的 Document 目录，第 4 行调用 write 方法将图片写入到本地文件中。如果大家想查看最终写入的效果，可以在最后一行添加 print 信息，将文件写入路径打印出来，观察图像写入是否成功。

### 5.4.3 GIF 图片分解最终实现效果

通过上述代码中的最后一行 print("\(imagePath)")可以获取图片最终保存的路径。进入该路径下可以看到如图 5.7 所示的图片最终分解结果。根据图 5.7 所示，在 Mac 系统下，利用系统图片的查看工具来查看 GIF 图片的分帧结果，对

比图中内容，可以看出 GIF 图片分解的结果是正确的。

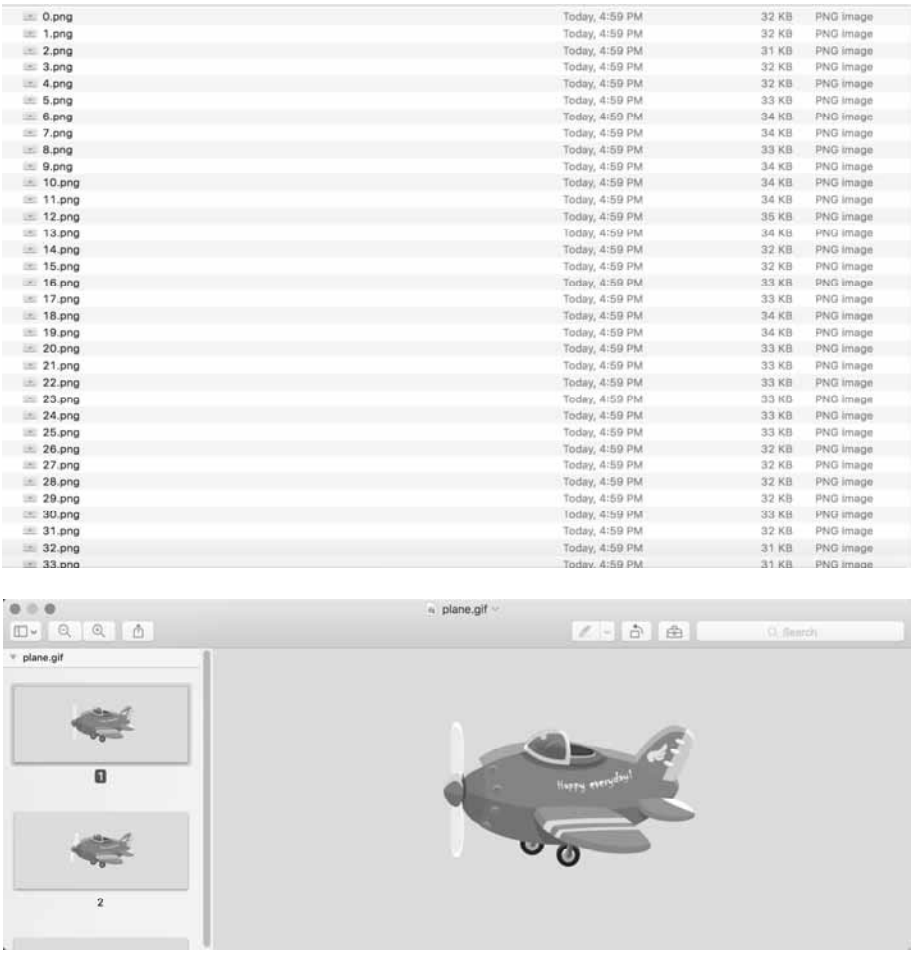



图 5.7　GIF 分解最终效果

# 5.5　序列图像合成 GIF 图像

## 5.5.1　GIF 图片合成思路

前面已经讲解了 GIF 图像如何分解为序列单帧图像，本节再来看看如何通过多帧序列图像合成 GIF 图像。多帧图像合成 GIF 的过程和 GIF 分解多帧图像

的过程互逆，将图 5.3 所示的图片分解过程倒过来推，就是 GIF 图像合成的过程。这里将 5.4 节分解的 67 张序列单帧图像作为需要处理的输入源进行讲述。

从功能上来说，GIF 图片的合成分为以下三个主要部分。

（1）加载待处理的 67 张原始数据源。

（2）在 Document 目录下构建 GIF 文件。

（3）设置 GIF 文件属性，利用 ImageIO 编码 GIF 文件。

### 5.5.2 GIF 图片合成代码实现

如下代码是根据 GIF 构建的三个主要步骤进行编写的。第一部分代码的功能是将 67 张 PNG 图片读取到 NSMutableArray 数组中。代码第 1 行初始化可变数组，第 2 行遍历 67 张本地图片，第 3 行按照图片的命名规律，构建 67 张图片名称，第 4 行加载本地图片。最后一行将读取的图片依次加载到 images 可变数组中。

```
        // Part1:读取 67 张 png 图片
1        let images:NSMutableArray = NSMutableArray()
2        for i in 0...66{// 遍历本地 67 张图片
3            let imagePath = "\(i).png" // 构建图片名称
4            let image:UIImage = UIImage(named: imagePath)!//
5            images.addObject(image)// 将图片添加到数组中
        }
```

代码第二部分的功能是构建在 Document 目录下的 GIF 文件路径。具体实现如下所示。

```
        // Part2:在 Document 目录创建 gif 文件
1     var docs=NSSearchPathForDirectoriesInDomains(
                  .documentDirectory, .userDomainMask, true)
2     let documentsDirectory = docs[0] as String
3     let gifPath = documentsDirectory+"/plane.gif"
4     print("\(gifPath)")
5     let url = CFURLCreateWithFileSystemPath(
```

```
                    kCFAllocatorDefault, gifPath as CFString!,
                    CFURLPathStyle.cfurlposixPathStyle, false)
6     let destion = CGImageDestinationCreateWithURL(url!,
                                kUTTypeGIF, images.count, nil)
```

代码 1 一行和第 2 行获取 Document 路径地址，第 3 行代码通过字符串拼接时组成完整的 Document 路径下 plane.gif 文件路径。为了方便查看 GIF 文件所在路径，第 4 行代码将 GIF 文件路径打印出来。第 5 行代码将 plane.gif 文件路径由 string 类型转换为 URL 类型。最后一行代码是 ImageIO 中构建 GIF 图片非常重要的方法，我们重点来分析该方法的作用和功能。

```
    public func CGImageDestinationCreateWithURL(_ url: CFURL, _ type:
CFString,    _    count:    Int,    _    options:    CFDictionary?)    ->
CGImageDestination?
```

CGImageDestinationCreateWithURL 方法的作用是创建一个图片的目标对象，为了便于大家理解，这里把图片目标对象比喻为一个集合体，如图 5.8 所示。集合体中描述了构成当前图片目标对象的一系列参数，如图片的 URL 地址、图片类型、图片帧数、配置参数等。本代码中将 plane.gif 的本地文件路径作为参数 1 传递给这个图片目标对象，参数 2 描述了图片的类型为 GIF 图片，参数 3 表明当前 GIF 图片构成的帧数，参数 4 暂时给它一个空值。

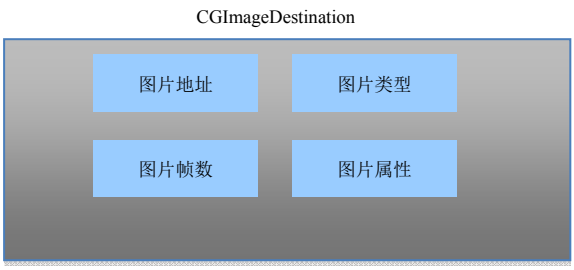



图 5.8　CGImageDestination 结构

到目前为止，待处理图片源已经加载到代码中，GIF 图片 Destination 也已经完成构建，下面就需要使用 ImageIO 框架把多帧 PNG 图片编码到 GIF 图片中，其处理流程如图 5.9 所示。

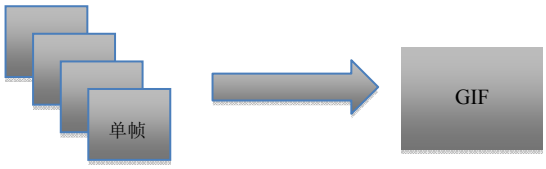



图 5.9　多帧图片合成 GIF 图片

具体实现代码如下：

```
        // Part3:设置 gif 图片属性，利用 67 张 png 图片构建 gif
1   let cgimagePropertiesDic = [kCGImagePropertyGIFDelayTime as
                                    String:0.1]//设置每帧之间播放
时间
2   let cgimagePropertiesDestDic =
          [kCGImagePropertyGIFDictionary
                      as String:cgimagePropertiesDic];
3   for cgimage in images{
4         CGImageDestinationAddImage(destion!,
                  (cgimage as AnyObject).cgImage!!,
                 cgimagePropertiesDestDic as CFDictionary?);
    }// 依次为 gif 图像对象添加每一帧元素

5   let gifPropertiesDic:NSMutableDictionary =
                              NSMutableDictionary()
6   gifPropertiesDic.setValue(kCGImagePropertyColorModelRGB,
                  forKey: kCGImagePropertyColorModel as String)
7   gifPropertiesDic.setValue(16, forKey:
           kCGImagePropertyDepth as String)// 设置图像的颜色深度
8   gifPropertiesDic.setValue(1, forKey:
       kCGImagePropertyGIFLoopCount as String)// 设置 Gif 执行次数
9   let gifDictionaryDestDic = [kCGImagePropertyGIFDictionary
                                  as String:gifPropertiesDic]
10  CGImageDestinationSetProperties(destion!,
     gifDictionaryDestDic as CFDictionary?);//为 gif 图像设置属性
11  CGImageDestinationFinalize(destion!);
```

代码第 1 行设置 GIF 图片属性，设置当前 GIF 中每帧图片展示时间间隔为 0.1s。代码第 2 行构建一个 GIF 图片属性字典，字典使用 GIF 每帧之间的时间

间隔初始化。代码第 4 行使用遍历的方法将已经准备好的图片快速追加到 GIF 图片的 Destination 中。代码第 5 行初始化一个可变字典对象，该字典对象主要用于设置 GIF 图片中每帧图片属性。第 6 行设置图片彩色空间格式为 RGB（Red Green Blue 三基色）类型。第 7 行设置图片颜色深度。一般来说黑白图像也称为二值图像，颜色深度为 1，表示 2 的一次方，即两种颜色：黑和白。灰度图像一般颜色深度为 8，表示 2 的 8 次方，共计 256 种颜色，即从黑色到白色的渐变过程有 256 种。对于彩色图片来说一般有 16 位深度和 32 位深度之说，这里设置为 16 位深度彩色图片。代码第 8 行设置 GIF 图片执行的次数，这里设置为执行一次。代码第 9 行和第 10 行负责将以上图片设置的各种属性添加到 GIF 的 Destination 目标中。最后一行完成 GIF 的 Destination 目标文件构建。

可以打印出当前 GIF 图片的路径，在该路径下可以看到最终生成的 GIF 图片，如图 5.10 所示。


图 5.10　GIF 最终生成效果

# 5.6　Gif 图像展示

## 5.6.1　GIF 图片展示思路

5.4 节和 5.5 节讲解了借助 ImageIO 框架实现图片的分解与合成功能，下面将为大家介绍如何在 iOS 下展示 GIF 图片。iOS 原生并不支持直接显示 GIF 图

片，由前面的分析可知，GIF 图片由一帧帧的单帧图片构成，所以只要实现 GIF 图片的分解，接下来就是多组图片显示的问题了。

在第 4 章中为大家详细介绍了逐帧图片的动画显示方法，使用这种方法可以实现 GIF 图片的展示。这里再为大家介绍另外一种图片展现形式，即基于 UIImageView 展现 GIF 多帧图片。

### 5.6.2 GIF 图片展示：基于 UIImageView

经过对 GIF 图片展示思路的分析可以知道，在 iOS 下展现 GIF 分为两步：第一步分解 GIF 图片为单帧图片，第二步在 iOS 下展现多帧图片。GIF 图片的分解在 5.4 节已经为大家介绍过，所以这里就不再重复了，下面来看看多帧图片如何展示。UIImageView 是一个用来展现图片的 UI 组件，不过它还有一些动画属性可以用来进行逐帧动画展现。

考虑到第一步 GIF 图片已经分解，所以这里把分解之后的 67 张图片先加载进来。如图 5.11 所示。

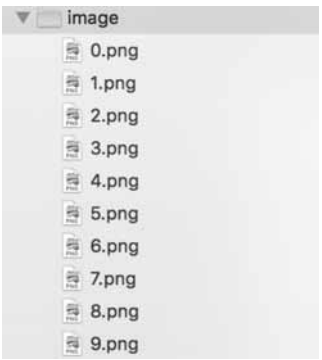



图 5.11 GIF 原始多帧图片

UIImageView 多帧图像展示具体实现代码如下。

```
var images:[UIImage] = []
 for i in 0...66{// 遍历本地 67 张图片
    let imagePath = "\(i).png" // 构建图片名称
    let image:UIImage = UIImage(named: imagePath)!
```

```
5            images.append(image)// 将图片添加到数组中
         }
6        let imageView = UIImageView()
7        imageView.frame = self.view.bounds
8        imageView.contentMode = UIViewContentMode.Center
9        self.view.addSubview(imageView)
10       imageView.animationImages = images
11       imageView.animationDuration = 5
12       imageView.animationRepeatCount = 1
13       imageView.startAnimating()
```

代码第 1 行初始化一个子元素为 UIImage 类型的数组对象。第 2 行到第 5 行通过 for 循环将 67 张图片依次加载到当前数组中。第 6 行实例化一个 UIImageView 实例对象。第 7 行和第 8 行设置 UIImageView 实例对象的 frame 位置属性以及图片的拉伸方式，这里设置为居中显示。第 9 行将 UIImageView 添加到 self.view 图层上。第 10 行将初始化加载的 67 张图片添加到 UIImageView 实例的 animationImages 上，相当于设置 UIImageView 的内容。第 11 行设置 UIImageView 图片动画播放周期。第 12 行设置动画重复次数。最后一行启动 UIImageView 多帧图片展示动画。

GIF 图片最终展示效果如图 5.12 所示。

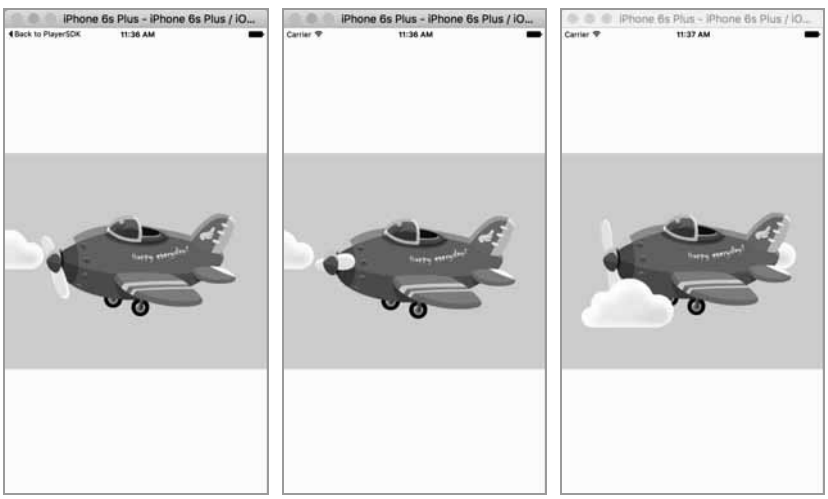



图 5.12　GIF 图像在 iOS 中的展示效果

## 5.7 本章小结

GIF 图像格式是常见的一种动态图片格式，无论是在 Web 端还是在移动端都经常遇到，但是考虑目前 iOS 还无法原生展现 GIF 图片，而对于 GIF 的原生支持暂时也没有像 JPG、PNG 等图像格式支持得这么全面，因此本章从图片的合成与分解角度来为大家讲解 GIF 的知识，结合 ImageIO 框架可以更方便地实现 GIF 图片的合成与分解。

本章最后为大家介绍了 Gif 图片的展示。Gif 图片的最终展示实质仍然是逐帧图像，因此除了本章中提到的使用 UIImageView 方法之外还可使用第 4 章的定时器或者 CADisplayerLink 方法。

本书的第 1～5 章主要是从 UIView 控件展示的角度为大家介绍了一些常见的动画效果，尤其对 UIView 常见的动画种类做了一个总结，方便读者在以后学习和工作过程中快速索引。另外逐帧动画也是显示层一个非常重要的动画效果，在工作中也十分常见。

# 第二卷

# 内容层动画

# 第 6 章　Core Animation: CABasicAnimation 动画效果

本章内容

- 掌握 UIView 显示层动画和 CALayer 内容层动画的区别
- 理解 Core Animation 核心动画架构
- 掌握 CALayer 内容层动画合集

前面为大家介绍了大量 iOS 显示层动画，并对常用动画效果做了一个合集。相信大家通过对合集中动画效果的掌握，已经可以进行一些常用动画效果的开发。从本章开始将正式进入到 iOS 内容层动画。内容层动画具有和显示层动画类似的初级动画效果，但除此之外，其利用内容图层的一些特殊属性还可以实现各种高级的效果，比如环形动画、圆角动画等。在学习内容层动画之前，先来了解一些和内容层相关的基础知识。

## 6.1　UIView 和 CALayer 的区别

UIView 是 iOS 中所有 UI 界面组件的基础，通过 UIView 可以很好地展示

一个 UI 界面组件。前面章节介绍的所有动画效果（包括 UIView 基础动画合集和逐帧动画等）都是通过 UIView 直接展现的，那么从本章开始，所有动画效果都将以 CALayer 为基础，那么什么是 CALayer 呢？先来看看二者之间的不同点。

（1）UIView 继承 UIResponder，因此 UIView 可以处理响应事件，而 CALayer 继承 NSObject，所以它只负责内容的创建、绘制。

（2）UIView 负责对内容的管理，而 CALayer 则是对内容的绘制。

（3）UIView 中的位置属性只有 frame、bounds、center，而 CALayer 除了具备这些属性之外还有 anchorPoint、position。

（4）通过修改 CALayer 可以实现 UIView 无法实现的很多高级功能。

当然 UIView 和 CALayer 的不同点远不止这些，这里就不为大家一一罗列了，只要理解了以上这四点不同，再来看 CALayer 动画就很简单了。

在本书中不区分显式和隐式动画的概念，而把与 UIView 相关的动画统称为显示层动画，把与 CALayer 相关的动画统称为内容层动画。

## 6.2　Core Animation 核心动画

Core Animation 为 iOS 核心动画，它提供了一组丰富的 API 可以用于制作各种高级炫酷的动画效果。Core Animation 来自 iOS 的 QuartzCore.framework 框架，它还具有以下特点。

（1）直接作用于 CALayer 图层上，而非 UIView 上。

（2）Core Animation 的执行过程在后台执行，不阻塞主线程。

（3）可以利用 CALayer 绝大多数属性制作高级动画效果。

下面来看看 Core Animation 下各种常用动画类的继承关系，如图 6.1 所示。

- @protocol CAMediaTiming 有很多动画公共的属性，比如常见的 duration（动画执行周期）、speed（速度）、repeatCount（重复次数）等一些公

共的属性都放在 CAMediaTiming 中。

- CAAnimation 主要用于实现动画的委托代理方法，比如动画开始事件和动画结束事件都是通过 CAAnimation 类来实现的。
- CAPropertyAnimation 为属性动画，分为基础动画和关键帧动画。在本节为大家介绍的 CALayer 内容层动画合集都是通过 CABasicAnimation 来实现的。CAKeyframeAnimation 为关键帧动画，与 UIView 中的关键帧动画实现原理类似。
- CAAnimationGroup 组合动画，顾名思义，利用这个类可以把其他常用动画组合在一起实现。
- CATransition 专场动画，主要用于视图控制器或者多 View 之间的视图切换场景。

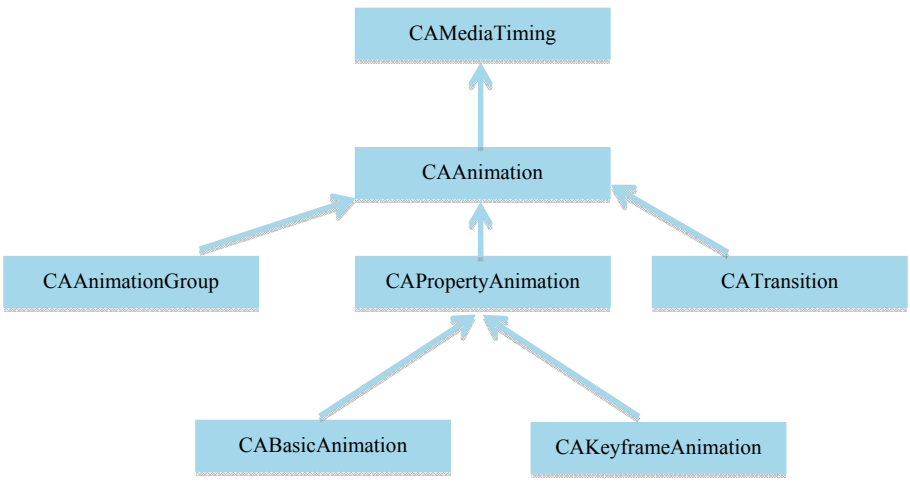



图 6.1　核心动画类结构

# 6.3　CALayer 层动画合集

## 6.3.1　位置动画

与第 2 章介绍的 UIView 层动画合集类似，下面也将为大家介绍一系列 CALayer 层动画。首先在屏幕中间添加一个登录按钮 Button。实现代码如下所示。

```
    override func viewDidLoad() {
```

```
    super.viewDidLoad()
    loginButton = UIButton(frame: CGRect(x: 20, y: 230, width:
self.view.frame.width-20*2,height: 50))
    loginButton!.backgroundColor = UIColor(red: 50/255.0,
            green: 185/255.0, blue: 170/255.0, alpha: 1.0)
    loginButton!.setTitle("登陆", for: UIControlState())
    self.view.addSubview(loginButton!)
}
```

该段代码的主要功能是在登录屏幕的中间位置添加一个登录按钮，这段代码与第 2 章代码类似，这里不做过多的解释。首先想实现位置移动动画效果，具体实现代码如下所示。

```
override func viewWillAppear(_ animated: Bool) {
    //位置
    let animation:CABasicAnimation = CABasicAnimation()
    animation.keyPath = "position"
    let positionX:CGFloat = loginButton!.frame.origin.x
                    +0.5*loginButton!.frame.size.width;
    let positionY:CGFloat = loginButton!.frame.origin.y
                +0.5*loginButton!.frame.size.height+100;
    animation.toValue = NSValue(cgPoint:
                    CGPoint(x:positionX,y: positionY))
    animation.duration = 2.0
    animation.fillMode = kCAFillModeForwards
    animation.isRemovedOnCompletion = false
    loginButton?.layer.add(animation, forKey: nil)
}
```

代码第 1 行初始化一个 CABasicAnimation 动画实例对象。第 2 行设置动画实例对象的效果。比如这里设置的是 position，表明当前是为了修改登录按钮的位置信息。

有一点需要补充的是，position 不同于之前的 frame 属性和 bounds 属性。Frame 属性具有 *x*、*y*、width、height 四个属性，分别描述了 UI 控件在父控件中左上角 *x*、*y* 坐标以及 UI 控件的宽高信息。frame 属性是相对于父控件来说的，

而 bounds 是相对于自身而言的。

```
bounds(x) = 0
bounds(y) = 0
bounds(width) = frame.(width)
bounds(height)= frame.(height)
```

以上描述了 bounds 属性和 frame 属性的关系，那么 position 又是什么呢？先来看看 position 与 frame 的换算公式。下面的公式是在不修改 anchorPoint 的情况下（关于 anchorPoint 会在 3D 相关章节中具体介绍）。

$$position(x) = frame(x)+frame(width)/2$$
$$position(y) = frame(y)+frame(height)/2$$

现在想实现登录按钮向下移动，需要修改 position 的 $y$ 坐标。代码第 3 行、第 4 行获取新的 position 位置。position $x$ 坐标不变，position $y$ 坐标在原来基础上向下移动 100。代码第 5 行将 position 包装成 NSValue，然后把 NSValue 传递给 animation 动画实例对象的 toValue 属性。第 6 行设置动画执行周期为 2s。代码第 7 行和第 8 行让当前动画保持结束之后的状态。代码第 9 行将当前动画添加到登录按钮的 Layer 图层上。如图 6.2 所示为动画最终执行效果。


图 6.2　位置动画效果：toValue 属性

如果想让登录按钮靠屏幕的左边缘对齐，并向下移动 100，那么可以将下面的代码：

```
    animation.toValue  =  NSValue(cgPoint:  CGPoint(x:positionX,y:
positionY))
```

替换为：

```
    animation.byValue = NSValue(cgPoint:CGPoint(x:-20,y:100))
```

通过对比可以发现，这里将原本的 toValue 属性修改为 byValue 属性。toValue 属性表明改变了控件的位置，所以给它一个新的 position 的 *x*、*y* 坐标。byValue 表明在控件原来位置的基础上，沿 *x*、*y* 坐标分别移动了多少。这里想让登录按钮靠齐屏幕的左边边界并向下移动 100，所以这里设置（–20，100）。–20 表明向左移动，100 表明向下移动。如图 6.3 所示为动画具体实现效果。


图 6.3　位置动画效果：byValue 属性

## 6.3.2　缩放动画

在高度不变的情况下缩小宽度，可以实现类似挤压的效果。在宽度不变的情况下缩小高度，可以实现类似拉伸的效果。本节为大家介绍如何通过

transform.scale 属性来实现挤压效果。scale 还有 *x*、*y* 两个属性。*x* 属性表明当前 UI 控件的 width 的缩放系数。*y* 属性表明当前 UI 控件的 height 的缩放系数。下面是具体实现代码。

```
override func viewWillAppear(_ animated: Bool) {
        //缩放
1        let animation:CABasicAnimation = CABasicAnimation()
2        animation.keyPath = "transform.scale.x"
3        animation.fromValue = 1.0
4        animation.toValue = 0.8
5        animation.duration = 2.0
6        animation.fillMode = kCAFillModeForwards
7        animation.isRemovedOnCompletion = false
8        loginButton?.layer.add(animation, forKey: nil)
}
```

代码整体结构与第 6.3.1 节类似，这里就不再重复。重点来看几个不一样的地方。代码第 2 行设置 UI 控件宽度缩放属性名称 transform.scale.x。第 3 行指明当前缩放系数原始值。原始系数设置为 1.0。第 4 行设置最终缩放系数。如图 6.4 所示为最终实现效果。


图 6.4　宽度缩放效果

修改 transform.scale.x 可以修改 UI 控件的宽度，如果想修改 UI 控件的高度，只要修改动画实例的 keyPath 属性为 transform.scale.y 即可。

### 6.3.3　旋转动画

本节为大家介绍动画的旋转效果，使用 transform 的 rotation 属性旋转动画。下面是具体实现代码。

```
override func viewWillAppear(_ animated: Bool) {
      //旋转
1      let animation:CABasicAnimation = CABasicAnimation()
2      animation.keyPath = "transform.rotation"
3      animation.toValue = 3.14/2
4      animation.duration = 2.0
5      animation.fillMode = kCAFillModeForwards
6      animation.isRemovedOnCompletion = false
7      loginButton?.layer.add(animation, forKey: nil)
}
```

代码的第 2 行将动画实例 keyPath 属性修改为 transform.rotation，表明当前想实现旋转动画效果。第 3 行设置旋转最终角度，代码中采用弧度来表示，所以 90°转换为弧度为 3.14/2。如图 6.5 所示为最终动画效果。

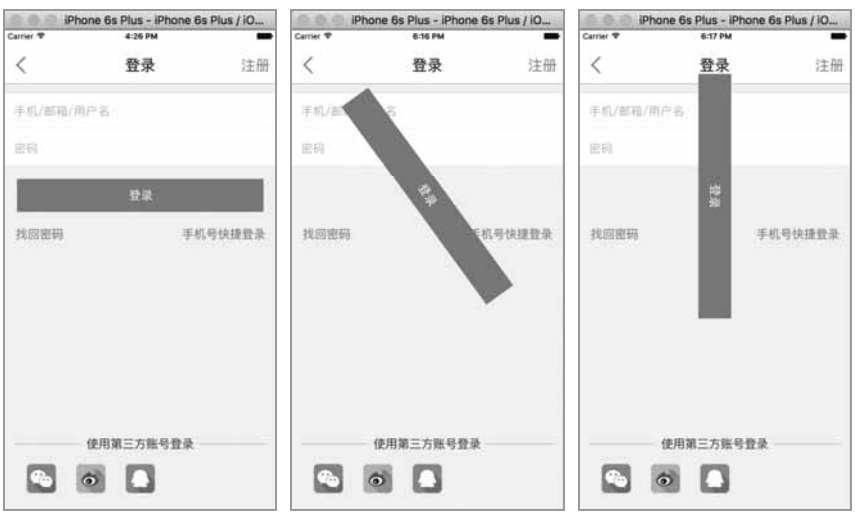



图 6.5　旋转动画效果

### 6.3.4 位移动画

位移动画的实现效果如图 6.2 所示，本节将采用 transform 的 translation 来实现这一动画。Translation 还有 *x*、*y* 两个属性分别表示在 *x*、*y* 方向上移动，下面是具体实现代码。

```
override func viewWillAppear(_ animated: Bool) {
       //位置
1       let animation:CABasicAnimation = CABasicAnimation()
2       animation.keyPath = "transform.translation.y"
3       animation.toValue = 100
4       animation.duration = 2.0
5       animation.fillMode = kCAFillModeForwards
6       animation.isRemovedOnCompletion = false
7       loginButton?.layer.add(animation, forKey: nil)
}
```

### 6.3.5 圆角动画

利用 CALayer 图层可以实现很多高级的功能，比如圆角效果、阴影效果等。这些常用效果借助于 CABasicAnimation 都可以以动画的形式展现出来。下面是登录按钮圆角动画的具体实现代码。

```
override func viewWillAppear(_ animated: Bool) {
             //圆角
1       let animation:CABasicAnimation = CABasicAnimation()
2       animation.keyPath = "cornerRadius"
3       animation.toValue = 15
4       animation.duration = 2.0
5       animation.fillMode = kCAFillModeForwards
6       animation.isRemovedOnCompletion = false
7       loginButton?.layer.add(animation, forKey: nil)
}
```

代码第 2 行设置 Layer 图层圆角属性 cornerRadius，第 3 行设置圆角半径为 15。如图 6.6 所示为最终实现效果。


图 6.6　圆角效果

### 6.3.6　边框动画

通过 Layer 图层可以设置 borderColor 边框颜色和 borderWidth 边框宽度，这种效果利用 CABasicAnimation 也可以实现。下面就用代码展示边框逐渐增强效果，如下所示。

```
override func viewWillAppear(_ animated: Bool) {
            //边框
1       loginButton?.layer.borderColor =
                                    UIColor.gray.cgColor
2       loginButton?.layer.cornerRadius = 10.0
3       let animation:CABasicAnimation = CABasicAnimation()
4       animation.keyPath = "borderWidth"
5       animation.toValue = 10
6       animation.duration = 2.0
7       animation.fillMode = kCAFillModeForwards
8       animation.isRemovedOnCompletion = false
9       loginButton?.layer.add(animation, forKey: nil)
}
```

代码第 1 行设置登录按钮边框颜色为灰色，第 2 行设置按钮圆角效果。第 4

行设置动画属性为 borderWidth，表明当前动画改变的是边框宽度效果。第 5 行设置边框最终宽度为 10。如图 6.7 所示为最终实现效果。


图 6.7　边框增强效果

## 6.3.7　颜色渐变动画

通过 Layer 图层 backgroundColor 可以实现登录按钮的颜色渐变效果，在这个 demo 中将实现登录按钮背景颜色从绿色到红色的渐变过程。下面是具体实现代码。

```
override func viewWillAppear(_ animated: Bool) {
        //颜色
1        let animation:CABasicAnimation = CABasicAnimation()
2        animation.keyPath = "backgroundColor"
3        animation.fromValue = UIColor.green.cgColor
4        animation.toValue = UIColor.red.cgColor
5        animation.duration = 2.0
6        animation.fillMode = kCAFillModeForwards
7        animation.isRemovedOnCompletion = false
8        loginButton?.layer.add(animation, forKey: nil)
}
```

代码第 3 行表示初始状态登录按钮颜色，第 4 行表明最终登录按钮的背景颜色。如图 6.8 所示为颜色渐变效果。


图 6.8　背景颜色渐变效果

除了修改背景颜色渐变效果之外，还可以修改登录按钮的边框颜色的渐变效果。具体实现代码如下所示。

```
override func viewWillAppear(_ animated: Bool) {
            //颜色
1       loginButton?.layer.borderWidth = 5
2        let animation:CABasicAnimation = CABasicAnimation()
3        animation.keyPath = "borderColor"
4        animation.fromValue = UIColor.green.cgColor
5        animation.toValue = UIColor.cyan.cgColor
6        animation.duration = 2.0
7        animation.fillMode = kCAFillModeForwards
8        animation.isRemovedOnCompletion = false
9        loginButton?.layer.add(animation, forKey: nil)
}
```

代码第 1 行设置登录按钮边框线条宽度为 5。代码第 3 行设置边框颜色动画属性。代码第 4 行和第 5 行修改边框颜色从绿色变为青色。如图 6.9 所示为最终实现效果。


图 6.9　边框颜色渐变效果

## 6.3.8　淡入淡出动画

使用 UIView 的 alpha 属性可以实现 UI 控件淡入淡出效果。在 Layer 图层中有一个类似的属性可以实现类似的淡入淡出动画效果。淡入效果的具体实现代码如下。

```
override func viewWillAppear(_ animated: Bool) {
        //淡入
1        let animation:CABasicAnimation = CABasicAnimation()
2        animation.keyPath = "opacity"
3        animation.fromValue = UIColor.green.cgColor
4        animation.toValue = 1.0
5        animation.duration = 2.0
6        animation.fillMode = kCAFillModeForwards
7        animation.isRemovedOnCompletion = false
8        loginButton?.layer.add(animation, forKey: nil)
}
```

Opacity 属性和 alpha 属性类似，通过设置 0～1.0 的浮点数可以实现透明效果。代码第 2 行设置动画属性 opacity。第 4 行设置 opacity 最终属性为 1.0，默

认情况下登录按钮的 opacity 属性为 0，表明按钮初始状态为隐藏，之后慢慢显现，最终出现在界面上。如图 6.10 所示为最终实现效果。


图 6.10　淡入效果

## 6.3.9　阴影渐变动画

通过阴影渐变可以实现很多高级的效果，比如随着日光的移动，地面物体投影渐变过程等。本节将实现登录按钮红色阴影效果渐变动画。具体实现代码如下所示。

```
override func viewWillAppear(_ animated: Bool) {
      //阴影渐变
1   loginButton?.layer.shadowColor =
                              UIColor.red.cgColor
2       loginButton?.layer.shadowOpacity = 0.5
3       let animation:CABasicAnimation = CABasicAnimation()
4       animation.keyPath = "shadowOffset"
5       animation.toValue = NSValue(cgSize:
                                  CGSize(width:10,height:10))
6       animation.duration = 2.0
7       animation.fillMode = kCAFillModeForwards
```

```
8       animation.isRemovedOnCompletion = false
9       loginButton?.layer.add(animation, forKey: nil)
}
```

代码第 1 行设置阴影背景颜色为红色。第 2 行设置阴影透明度为半透明 0.5。第 4 行设置阴影动画效果属性 shadowOffset。第 5 行设置阴影投影角度，分别向 *x*、*y* 轴方向渐变。如图 6.11 所示为最终实现效果。


图 6.11　投影渐变效果

## 6.4　本章小结

本章首先为大家区分了 UIView 和 CALayer 动画的区别，同时这也是本书动画的整体分类划分。其次从架构上介绍了 CALayer 图层的各种动画类之间的从属关系和各动画类的具体使用场景。最后为大家总结了一系列的 CALayer 内容层动画合集。下面再为大家做一个详细的总结。

（1）位置动画：position、transform.translation.x、transform.translation.y、transform.translation.z。

（2）缩放动画：transform.scale.x、transform.scale.y。

（3）旋转动画：transform.rotation。

（4）颜色动画：backgroundColor、borderColor。

（5）淡入淡出动画：opacity。

（6）高级效果：圆角动画（cornerRadius）、边框动画（borderWidth）、阴影动画（shadowOffset）。

# 第 7 章　Core Animation: CAKeyframeAnimation、CAAnimationGroup 动画

## 本章内容

- 理解 CALayer 层关键帧动画的使用方法
- 掌握 CAAnimationGroup 组合动画的使用方法
- 掌握关键帧动画每帧细节控制（弹幕）

CAKeyframeAnimation 是 CALayer 层下的关键帧动画类，利用该类可以实现类似 UIView 的关键帧动画效果。CAKeyframeAnimation 是 CApropertyAnimation 的一个子类，与 CABasicAnimation 原理类似，都是通过修改当前 CALayer 图层的 value 属性来实现动画效果。不同的是 CABasicAnimation 一般只能使用 fromValue、toValue、byValue，即只能修改一个 value 值。而 CAKeyframeAnimation 则可以修改一组 value 值来实现对动画更为精确细腻的控制。

CAAnimationGroup 组合动画效果，顾名思义，可以把各种 CA 类动画组合起来形成各种效果组合在一起的嵌套动画形式。在第 2 章为大家介绍了各种 UIView 动画效果组合在一起的形式，在本章将结合 CAAnimationGroup 实现这一效果。

# 7.1　CAKeyframeAnimation 动画属性要点

CAKeyframeAnimation 的使用很简单，只需在合适的位置设置相应的关键帧即可。而选取合适的位置、设置合适的关键帧都离不开 CAKeyframeAnimation 的各种属性。下面就对 CAKeyframeAnimation 的各种常用属性做一个解析。

（1）values：该属性是一个数组类型，数组中的每个元素都描绘了一个关键帧的相关属性。比如描述关键帧位置的动画时，values 描述的是位置信息。描述关键帧淡入淡出动画时，values 描述的是透明度渐变信息。

（2）keyTimes：默认情况下，关键帧在动画的展示周期内是均匀播放的，但是如果设置了这个属性，就可以精确控制每个关键帧显示的周期。这个属性的取值范围在 0～1 之间。所以每个关键帧显示的周期为 keyTimes*duration。该属性在讲解 UIView 关键帧动画时已经为大家介绍过，本章就不再赘述了。

（3）path：如果通过 values 属性可以对动画进行比较细腻的控制，那么通过 path 属性则可以对动画的细节部分控制得更为精确。因为通过设置 CGPathRef\CGMutablePathRef 可以让动画按照自己绘制的路径随心所欲地运行。

# 7.2　CAKeyframeAnimation 淡出动画效果

通过 7.1 节的分析可以知道，通过设置大量的 values 值，能够实现多关键帧的动画效果。本节将为大家实现一个颜色渐变效果。下面是具体实现代码。

```
var loginButton:UIButton?
    override func viewDidLoad() {
        super.viewDidLoad()

1       let view:UIView = UIView()
2       view.backgroundColor = UIColor.red
3       view.frame = CGRect(x: 100, y: 100, width: 200, height: 200)
4       self.view.addSubview(view)
5       let animation:CAKeyframeAnimation = CAKeyframeAnimation()
```

```
6       animation.duration = 10.0
7       animation.keyPath = "opacity"
8       let valuesArray:[NSNumber] = [
                                NSNumber(value: 0.95 as Float),
                                NSNumber(value: 0.90 as Float),
                                NSNumber(value: 0.88 as Float),
                                NSNumber(value: 0.85 as Float),
                                NSNumber(value: 0.35 as Float),
                                NSNumber(value: 0.05 as Float),
                                NSNumber(value: 0.00 as Float)]
9       animation.values = valuesArray
10      animation.fillMode = kCAFillModeForwards
11      animation.isRemovedOnCompletion = false
12      view.layer.add(animation, forKey: nil)
    }
```

代码第 1 行到第 4 行实例化一个 UIView 实例对象，并将其背景颜色设置为红色，添加到self.view的合适位置上。第5行设置关键帧动画实例对象animation。第 6 行设置动画周期为 10s。第 7 行设置动画属性为 opacity 透明度属性。第 8 行设置透明度属性渐变范围。这里设置一个数组，该数组表明 UIView 的透明度从非透明状态一直保持到透明度为 0.85，然后急剧变为透明度 0。第 9 行将 values 数组的值赋值给动画实例对象 values 属性。第 10 行和第 11 行设置动画的最终保持状态。最后一行将动画添加到当前图层上。如图 7.1 所示为最终实现效果。

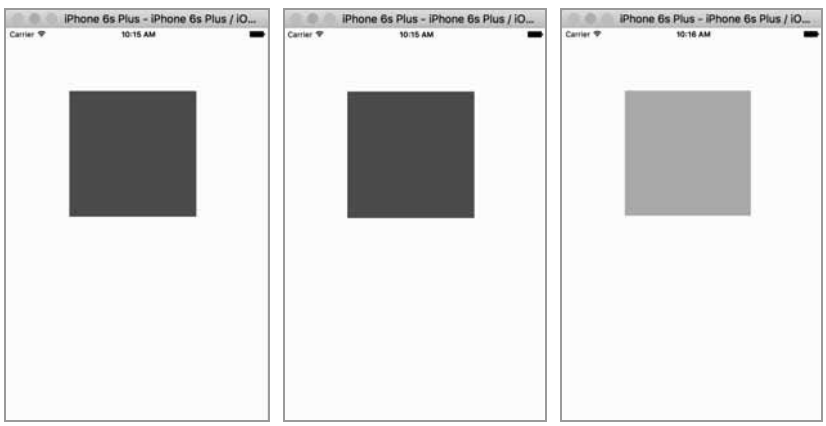



图 7.1　关键帧透明度渐变动画

## 7.3　CAKeyframeAnimation 任意路径动画

除了通过 values 的方式设置关键帧的路径之外，还可以通过 path 属性设置动画关键帧的执行路径。在 iOS 中可以通过使用 Quartz 2D 绘制各种各样的线段或者形状，总结起来即有点、线、圆三大类，当然还有比较复杂的比如贝济埃曲线等。既然使用 Quartz 2D 可以绘制这些线段或者圆弧，那么关键帧的 path 属性当然可以按照绘制的路径进行移动。下面是按直线运动的关键帧动画实现代码。

```
1 var imageView:UIImageView = UIImageView()
   override func viewDidLoad() {
       super.viewDidLoad()
2      imageView.frame = CGRect(x:50,y:50,width:50,height:50)
3      imageView.image = UIImage(named: "Plane.png")
4      self.view.addSubview(imageView)
5      let pathLine:CGMutablePath = CGMutablePath()
6      pathLine.move(to: CGPoint(x:50,y:50))
7      pathLine.addLine(to: CGPoint(x:300,y:50))
8      let animation:CAKeyframeAnimation = CAKeyframeAnimation()
9      animation.duration = 2.0
10      animation.path = pathLine
11     animation.keyPath = "position"
12     animation.fillMode = kCAFillModeForwards
13     animation.isRemovedOnCompletion= false
14     imageView.layer.add(animation, forKey: nil)
   }
```

代码首先实例化一个 UIImageView 实例对象，第 2 行到第 4 行设置当前 UIImageView 实例对象的 frame 属性和 image 内容（飞机图片），最后把实例对象添加到 self.view 上。第 5 行创建一个可变路径对象 pathLine。第 6 行首先指明可变对象的起始点坐标（50，50）。第 7 行添加一条直线连接（50，50）到（300，50）两点。第 8 行实例化关键帧动画实例对象，第 9 行到第 11 行设置动画的执行周期为 2s，执行路径即代码第 7 行绘制的线段，关键帧动画属性设置

为 position，表明动画按照位置属性形成关键帧。第 12 行和第 13 行保持动画完成之后的最终状态。最后一行将关键帧动画添加到 imageView 实例对象的 Layer 图层上。如图 7.2 所示为最终实现效果。

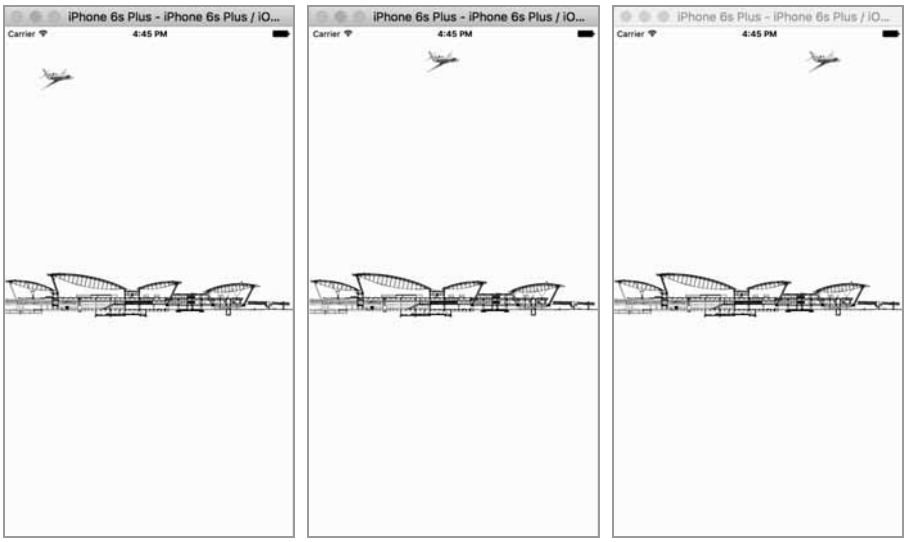



图 7.2　飞机按照自定义 path（直线）路径运动

通过 Quartz2D 可以很方便地绘制直线、圆弧，下面将实现飞机围绕圆弧运动的动画效果。该功能实现代码与飞机按照 path 路线直线运动较为类似，这里仅列举不同的代码部分。具体实现代码如下所示。

```
        ……
    1    let pathArc:CGMutablePath = CGMutablePath()
    2    pathLine.move(to: CGPoint(x:50,y:50))
    3     pathLine.addArc(center: CGPoint(x:200,y:200), radius: 150,
startAngle: 0, endAngle:CGFloat(M_PI_2), clockwise: true)
      ……
    4      animation.path = pathArc
```

该段代码首先将 path 的起始点设置在（50，50），然后又绘制了一个以（200，200）为圆心，150 为半径的圆形，该圆形的起始角度为 0°，最终角度为 360°。如图 7.3 所示描绘了 path 起始点以及圆形的起始点和最终点。

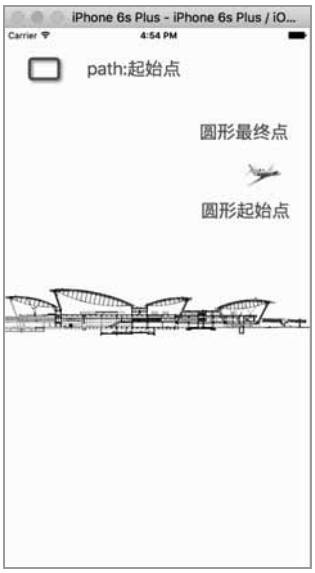



图 7.3　path 各关键点坐标

根据图 7.3 可以知道，该动画效果先从 path 起始点沿直线运动到圆形起始点，然后在该点按照圆形飞行一周，最后停留在圆形最终点。动画效果如图 7.4 所示。

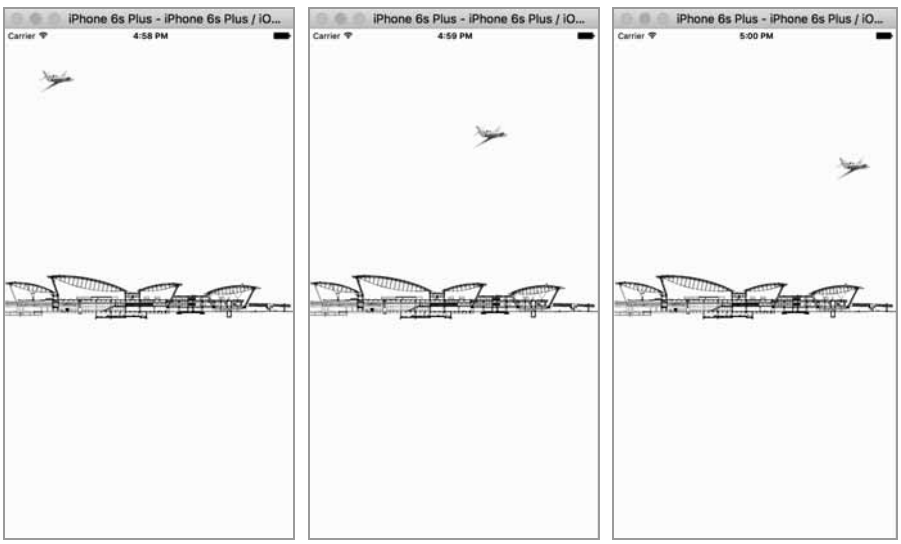



图 7.4　飞机按照自定义 path（直线+圆形）路径运动

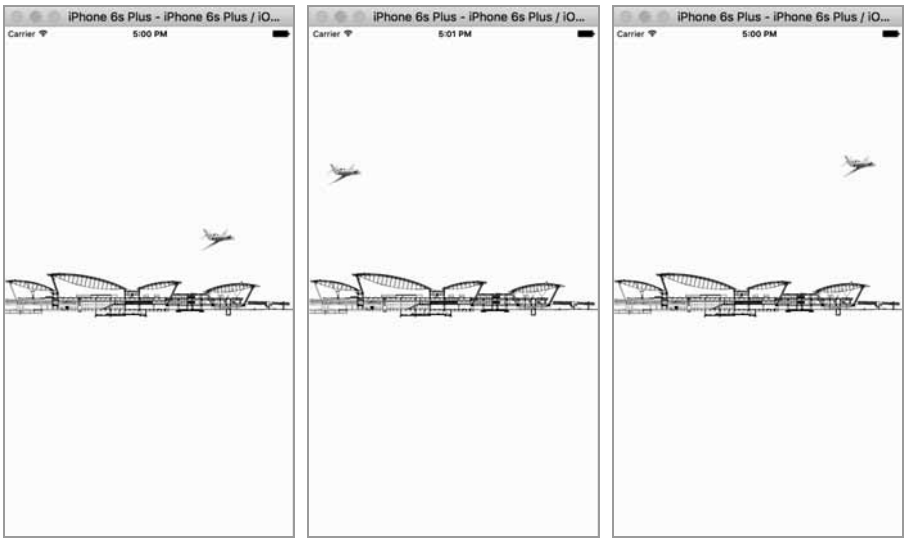



图 7.4　飞机按照自定义 path（直线+圆形）路径运动（续）

从图 7.4 中可以看出，前面三张图片是直线运动，后面三张图为圆形运动，与代码中设置的功能相互印证。

## 7.4　CAAnimationGroup 组合动画效果

第 2 章为大家介绍了使用 UIView 各种初级动画效果实现登录按钮一边旋转、一边缩小、一边向右上角移动消失的动画场景，本章将结合 CAAnimationGroup 重现这一动画场景。下面是具体实现代码。

```
    var loginButton:UIButton?
    override func viewDidLoad() {
        super.viewDidLoad()
1       loginButton = UIButton(frame: CGRect(x:20,y:230,
                    width:self.view.frame.width-20*2,height:50))
2       loginButton!.backgroundColor = UIColor(red: 50/255.0,
                  green: 185/255.0, blue: 170/255.0, alpha: 1.0)
3       loginButton!.setTitle("登陆", for:UIControlState.normal)
4       self.view.addSubview(loginButton!)
5       let rotate:CABasicAnimation = CABasicAnimation()
```

```
    rotate.keyPath = "transform.rotation"
    rotate.toValue = M_PI
    let scale:CABasicAnimation = CABasicAnimation()
    scale.keyPath = "transform.scale"
    scale.toValue = 0.0
    let move:CABasicAnimation = CABasicAnimation()
    move.keyPath = "transform.translation"
    move.toValue = NSValue(cgPoint:CGPoint(x:217,y:-230))
    let animationGroup:CAAnimationGroup = CAAnimationGroup()
    animationGroup.animations = [rotate,scale,move];
    animationGroup.duration = 2.0;
    animationGroup.fillMode = kCAFillModeForwards;
    animationGroup.isRemovedOnCompletion = false
    loginButton?.layer.add(animationGroup, forKey: nil)
}
```

代码第 1 行到第 4 行实例化 UIButton 实例对象，并把 UIButton 按钮放在登录界面合适的位置。设置登录按钮颜色为青绿色，设置按钮 title 标题为“登录”字样。最后将登录按钮添加在 self.view 上。第 5 行到第 7 行实例化一个 CABasicAnimation 实例对象，设置旋转动画属性 transform.rotation，最后设置旋转角度为 180°。第 8 行到第 10 行实例化一个 CABasicAnimation 实例对象，设置动画属性为 transform.scale，最后设置缩放效果从 1.0 缩放到 0.0。第 11 行到第 13 行实例化一个 CABasicAnimation 实例对象，设置位移动画属性 transform.translation，将登录按钮从当前位置移动到右上角。第 14 行实例化一个 animationGroup 实例对象。第 15 行将旋转、缩放、位移三个 CABasicAnimation 基础动画实例添加到 animationGroup 实例对象中。第 16 行到第 18 行设置动画周期，最终停留状态。最后一行将动画效果添加到登录按钮上。如图 7.5 所示为组合动画最终实现效果。


图 7.5　组合动画最终实现效果

## 7.5　本章小结

在本章为大家详细介绍了 CALayer 层关键帧动画和组合动画的使用方法，并对比 UIView 关键帧动画的异同，通过修改一组 CAKeyframeAnimation 中的 value 值来实现对动画更为精确和细腻的控制，比如设置某关键帧的起始时间和持续时间。

本章最后结合 Quartz 2D 为大家介绍自定义路径的关键帧动画执行方法。其实自定义关键帧路径不仅只有直线和圆形，通过几条直线的组合还可以实现三角、长方形、五角形等复杂路径的关键帧动画执行效果。同时如果修改圆形的结束弧度，比如 45°、90°，那么可以实现关键帧弧形运动的动画效果。组合动画的使用则较为简单，使用的时候只需把几种简单动画组合在一起形成组合效果即可。

# 第 8 章　综合案例：登录按钮动画效果

## 本章内容

- 掌握水纹按钮动画效果设计方法
- 掌握 UIView 显示层动画以及 Layer 内容层动画的综合使用

在前面 7 章已经为大家介绍了 UIView 显示层动画和 CALayer 内容层动画的各种实现方法，本章将通过两个具体的实例为大家回顾这两种动画的使用方法，以及构建复杂动画效果的设计思路。

## 8.1　综合案例 1：水纹按钮动画效果实现原理

当单击按钮时，会以单击点为圆心形成水纹扩散效果。扩散形状为圆形，扩散颜色为粉红色。在扩散的过程中按钮的状态为不可点击。如图 8.1 为该动画的分帧实现效果。

为了进一步弄清楚该动画的细节，理清动画的每一个展现细节，需要多做几次实验。当按钮的位置不同时，会形成以不同单击点为圆心的水纹形状。如图 8.2 所示，当单击不同点的时候按钮的水纹圆心不同。

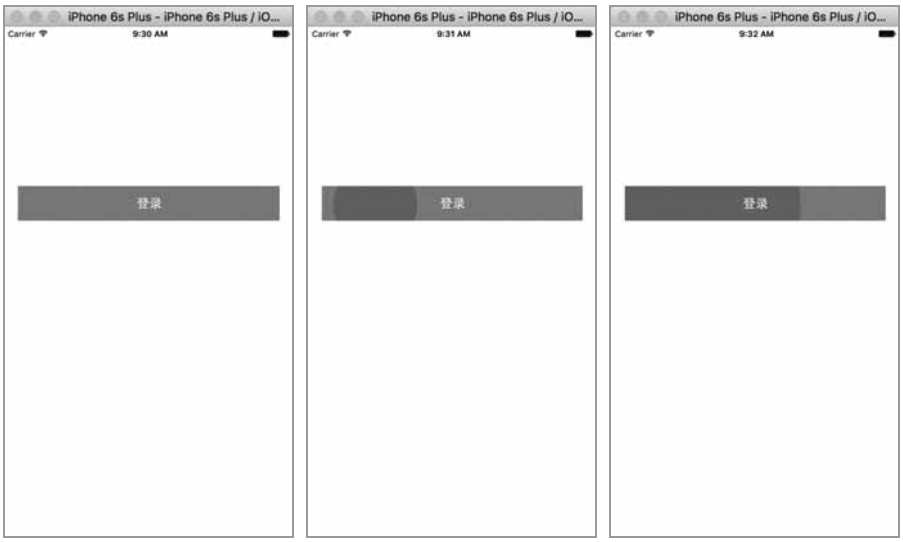



图 8.1　水纹按钮动画效果

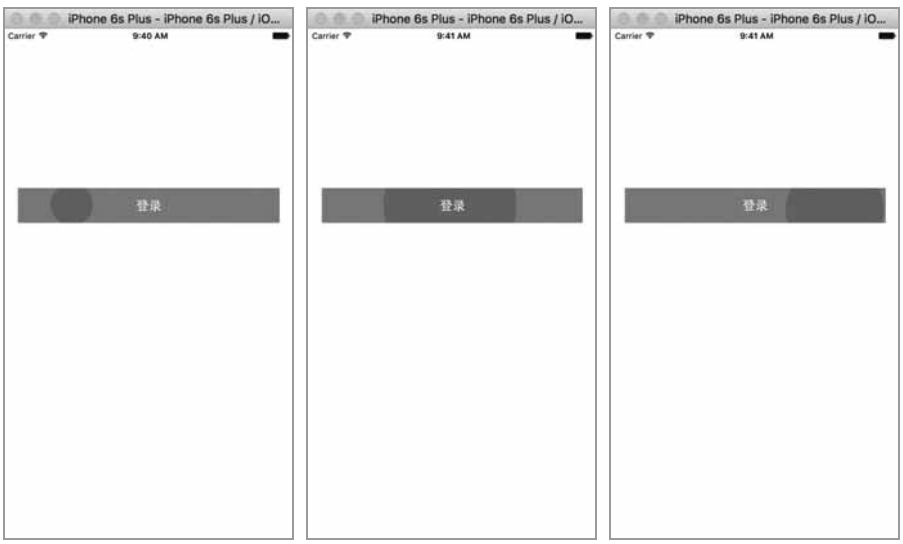



图 8.2　点击不同点时水纹动画效果

在细节上需要注意的第一个问题是如何获取当前按钮单击时的坐标呢？大家知道所有的 UIView 图层都能响应 Touch 方法，根据 Touch 方法可以获取当前单击的具体坐标。但是这里需要处理的是 UIButton。Button 的首要特点就是具有单击之后的响应方法，所以所有的动画效果处理都要在 Button 的响应方法

中来完成，所以这里使用UIView的Touch方法获取单击点坐标的方法不太合适。不过 UIButton 的响应方法可以传递多种事件，比如传递 UIButton 或者 event 事件，而从 event 事件中即可获取当前单击的位置。

经过以上分析相信已经完美解决第一个细节问题了，接下来来思考第二个细节问题。这里并没有准备大量的水纹单帧图片，而水纹的颜色是可以任意设置的，所以该动画效果并非逐帧动画。既然动画效果并非逐帧动画，而 UIView 和 Layer 的所有基础动画类型中也没有相似的动画实现方法，那么只可能是基于 NSTimer+Draw 重绘动画或者基于 CADisplayerLink＋Draw 重绘动画来实现。

第三个问题是，水纹的具体展示形状是什么样的呢？反复单击按钮，如图 8.3 所示。

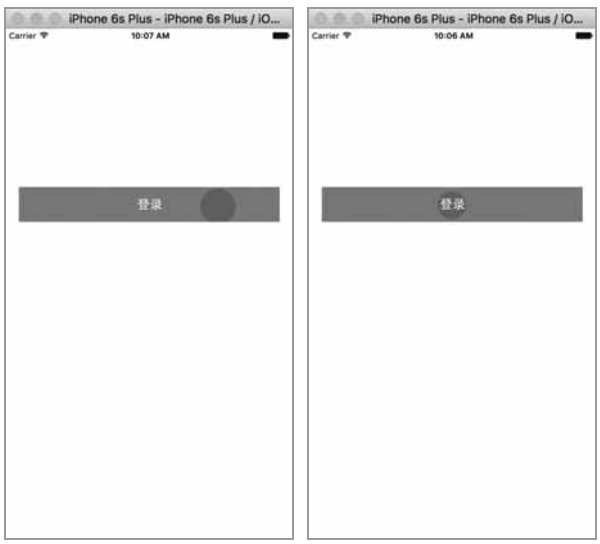



图 8.3　水纹圆形效果

在 CoreGraphics 框架下，可以使用 CGContext 下的 addArc 方法绘制圆形。

该方法的作用是在 Context 的上下文绘制一个圆形。如下代码所示。

```
public func addArc ( center: CGPoint,
                     radius: CGFloat,
                     startAngle: CGFloat,
```

```
                        endAngle: CGFloat,
                        clockwise: Int32)
```

（1）center: CGPoint：表明圆心 $x$、$y$ 坐标。

（2）radius: CGFloat：当前绘制圆形半径。

（3）startAngle: CGFloat：绘制圆形其实角度。

（4）endAngle: CGFloat：绘制圆形结束角度。

（5）clockwise: Int32：绘制圆形方向，顺时针或者逆时针。

## 8.2 水纹按钮动画效果具体代码实现

这里将水纹动画效果根据功能划分为以下五个主要的功能模块。

（1）UIButton 坐标获取模块。

（2）Draw 圆形绘制模块。

（3）定时器刷新模块。

（4）其他模块。

（5）调用模块。

下面先来看一下 UIButton 坐标获取模块代码。

```
func startButtonAnimation(_ msenderBt:UIButton,mevent:UIEvent){
1       self.userInteractionEnabled = false
2       let button:UIView = msenderBt as UIView
3       let touchSet:NSSet=mevent.touches(for: button)! as NSSet
4       var touchArray:[AnyObject] = touchSet.allObjects
                                                as [AnyObject]
5       let touch1:UITouch = touchArray[0] as! UITouch
6       let point1:CGPoint = touch1.location(in: button)
7       self.circleCenterX = point1.x
8       self.circleCenterY = point1.y
```

```
9       timer = Timer.scheduledTimer (timeInterval: 0.02,
                    target: self,
                    selector: #selector(MyButton.timeaction),
                    userInfo: nil,
                    repeats: true);
10       RunLoop.main.add(timer!,
                            forMode:RunLoopMode.commonModes)
}
```

代码第 1 行设置当前按钮点击状态为不可点击。因为该方法是在 UIButton 响应事件中调用，既然 UIButton 已经响应，因此这里必须要将 Button 的点击属性设置为 false，直到该动画完成才能实现下次点击。第 2 行将传递过来的 UIButton 向上转型为 UIView。第 3 行获取当前 event 事件在 UIView 的事件集。第 4 行获取当前点击的所有事件对象。第 5 行获取第一个点击事件的 UITouch 对象。第 6 行获取当前点击的 UITouch 对象在 Button 中的位置。第 7 行到第 8 行获取当前点击点的 *x*、*y* 坐标。第 9 行和第 10 行实例化一个定义器用于定时刷新 Button 水纹动画。并将定时器添加到 mainRunLoop 中。

第二个模块在 draw 方法中绘制圆形。具体实现代码如下所示。

```
override func draw(_ rect: CGRect) {
1       let ctx:CGContextRef = UIGraphicsGetCurrentContext()!
2       let endangle:CGFloat = CGFloat(M_PI*2)
3       ctx.addArc(center:
                    CGPoint(x:circleCenterX,y:circleCenterY),
                    radius:viewRadius,
                    startAngle:0,
                    endAngle:endangle,
                    clockwise:false)
4       let stockColor:UIColor = targetAnimColor
5       stockColor.setStroke()
6       stockColor.setFill()
7       CGContextFillPath(ctx)
    }
```

代码第 1 行获取当前绘制上下文的 Context。第 2 行设置绘制的角度为 360°，

即圆的一周。第 3 行使用 addArc 方法绘制圆形。第 4 行获取当前动画绘制颜色，这里将颜色设置为粉红色。第 5 行和第 6 行填充当前圆形的内部颜色和边缘颜色。最后一行完成圆形的绘制。

下面是定时器刷新模块实现代码。

```
func timeaction(){
1       countNum += 1
2       let dismissTime:DispatchTime = DispatchTime.now() +
          Double(Int64(0*NSEC_PER_SEC)) / Double(NSEC_PER_SEC)
3       DispatchQueue.main.asyncAfter(deadline: dismissTime,
          execute: {() in
4           self.viewRadius += 5
5           self.setNeedsDisplay()
        })
6       if(countNum>50){
7           countNum = 0
8           viewRadius = 0
9           timer?.invalidate()
10          DispatchQueue.main.asyncAfter(deadline: dismissTime,
              execute: {() in
11              self.viewRadius = 0
12              self.setNeedsDisplay()
            })
13          self.userInteractionEnabled = true
        }
    }
```

该方法每 0.2s 刷新一次。countNum 表明该方法被调用了多少次。代码第 2 行获取一个 dismissTime 时间，这里将时间设置为 0。修改合适的 dismissTime 可以产生不同的绘制效果。在主线程中增加圆形绘制半径（＋5），并调用 setNeedsDisplay()方法触发 Draw 完成重绘。第 6 行判断该方法被调用 50 次之后，清空 countNum 和 viewRadius，并将定时器设置为无效状态。最后使能当前 UIButton 的点击功能。

下面主要完成 init 初始化方法和各参数的初始化。

```
1    var viewRadius:CGFloat = 0
2    var countNum:Int = 0
3    var timer:Timer?
4    var circleCenterX:CGFloat = 0
5    var circleCenterY:CGFloat = 0
6    var targetAnimColor:UIColor = UIColor(red: 216.0 / 255.0,
 green: 114.0 / 255.0, blue: 213.0 / 255.0, alpha: 0.8)
7    override init(frame: CGRect) {
8        super.init(frame: frame)
9        self.backgroundColor = UIColor(red: 50/255.0, green:
                    185/255.0, blue: 170/255.0, alpha: 1.0)
    }
10    required init(coder aDecoder: NSCoder) {
       super.init(coder: aDecoder)!
    }
```

代码第 1 行和第 2 行设置当前圆形半径初始值为 0，定时器刷新方法初始值为 0。第 3 行定义一个定时器实例对象。第 4 行和第 5 行设置当前点击圆形的圆心坐标（x，y）。第 6 行设置点击颜色为粉红色。第 7 行至最后，完成 init 初始化方法。

以上完成之后，只需要在最终使用的位置调用该类即可。以上内容都在自定义 MyButton：UIButton 类中实现。

```
override func viewDidLoad() {
1       super.viewDidLoad()
2       let loginButton:MyButton = MyButton(frame: CGRect(x: 20,
         y: 230, width: self.view.frame.width-20*2,height: 50))
3       loginButton.setTitle("登陆", for: UIControlState())
4       loginButton.addTarget(self, action:
              #selector(ViewController.loginAction(_:event:)),
              for: UIControlEvents.touchUpInside)
5       self.view.addSubview(loginButton)
    }
    func loginAction(_ sender:UIButton,event:UIEvent) {
```

```
        let bt:MyButton = sender as! MyButton
        bt.startButtonAnimation(sender, mevent: event)
    }
```

代码第 2 行通过当前 frame 位置实例化 UIButton。第 3 行设置当前按钮的 title 内容。第 4 行设置当前登录按钮的响应方法。最后一行将登录按钮添加到 self.view 上。在登录按钮响应方法中，将传递过来的 UIButton 向下转型为 MyButton，并调用 startButtonAnimation 方法，实现水纹动画效果。最终动画效果如图 8.1 所示。

## 8.3 综合案例 2：登录按钮动画效果实现原理

下面将为大家介绍一些常用的登录按钮动画效果。由于该效果比较复杂，这里将拷贝三个相同的登录按钮，用三个相同的登录按钮在不同时刻的表现状态来展示这一动画的渐变过程。如图 8.4 所示为该动画效果分解示意图。

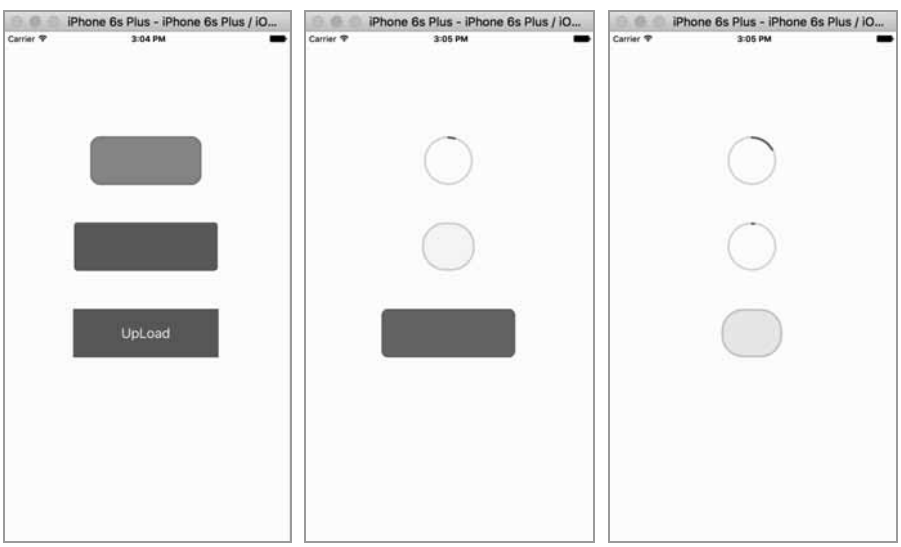



图 8.4　登录按钮动画效果分解示意图

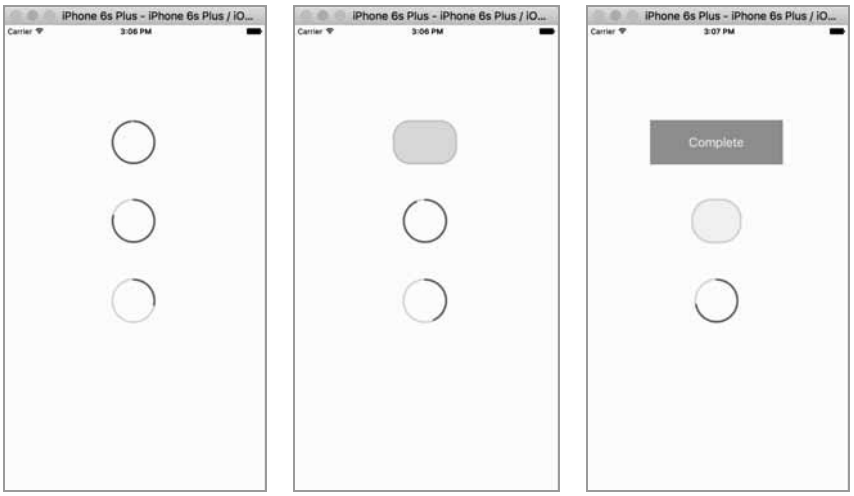



图 8.4　登录按钮动画效果分解示意图（续）

从图 8.4 中可以看出，该动画效果极其复杂，因此这里采取分过程的方法，将动画划分为三个阶段来实现。

（1）第一阶段：动画从 UpLoad 蓝色长方形条状按钮到刚开始形成圆形。如图 8.5 所示。


图 8.5　动画第一阶段

该动画第一阶段包括从长方形到圆角的圆角动画，颜色逐渐消失的颜色淡出动画，以及最后的长方形消失、圆弧显示出来这一过程的动画。第一阶段涉及的各种动画效果，在前面章节都已经为大家介绍过。

（2）第二阶段：动画从圆弧显示出来开始，顺时针绘制圆弧，直至圆弧最终绘制完成。如图 8.6 所示展示了这一动画过程。

图 8.6　动画第二阶段

动画第二阶段主要涉及圆弧绘制动画，以及在绘制圆弧动画时的蓝色填充效果。

（3）第三阶段：从圆弧绘制完成到圆弧伸展为长方形，在伸展过程中，伴随着颜色变化、圆角动画等基础动画效果。如图 8.7 所示为动画第三阶段。


图 8.7　动画第三阶段

虽然该动画效果比较复杂，但是经过分析之后可以看出，每个阶段的每个子效果都是非常容易实现的基础动画效果。

## 8.4　登录按钮动画效果代码实现

### 8.4.1　第一阶段动画

要想完美实现第一阶段的动画效果还需要仔细观察，在图 8.5 中可以看出，初始化的长方形登录按钮包含外部边框和内部两部分，所以第一步初始化 Button 时需要首先完成这两部分功能。下面是具体实现代码。

```
let FreshBlue:UIColor = UIColor(red: 25.0 / 255.0, green:
            155.0 / 255.0, blue: 200.0 / 255.0, alpha: 1.0)
let FreshGreen:UIColor = UIColor(red: 150.0 / 255.0, green:
             203.0 / 255.0, blue: 25.0 / 255.0, alpha: 1.0)
var view:UIView?
var viewborder:UIView?
var button_x:CGFloat = 0
var button_y:CGFloat = 0
var button_w:CGFloat = 0
var button_h:CGFloat = 0
var label:UILabel?
var circleView:CircleView?
override init(frame: CGRect) {
```

```
12      super.init(frame: frame)
13      button_x = frame.origin.x;
14      button_y = frame.origin.y;
15      button_w = frame.size.width;
16      button_h = frame.size.height;
17      view = UIView(frame:CGRect(x: 0, y: 0,
                            width: button_w, height: button_h))
18      view!.backgroundColor = FreshBlue;
19      viewborder = UIView(frame:CGRect(x: 0, y: 0,
                            width: button_w, height: button_h))
20      viewborder!.backgroundColor = UIColor.clearColor()
21      viewborder!.layer.borderColor = FreshBlue.CGColor;
22      viewborder!.layer.borderWidth = 3.0;
23      self.addSubview(view!)
24      self.addSubview(viewborder!)
}
```

代码第 1 行和第 2 行初始化登录按钮刚开始的蓝色以及最后恢复的绿色。第 3 行和第 4 行设置登录按钮内部 View 以及外部边框 View。第 5 行到第 8 行用于记录当前登录按钮的 x、y、width、height 四个属性。第 9 行定义登录按钮初始化时的 UpLoad 文字信息。第 10 行定义 Circle 圆形 View。

在 init 方法中第 13 行到第 16 行获取当前 view 的 *x*、*y*、width、height 四个属性。第 17 行和第 18 行实例化 View，并设置登录按钮内部 View 的 frame 属性和背景颜色。第 19 行到第 22 行实例化登录按钮外部边框 View，并将外部边框颜色设置为蓝色，内部颜色清空。第 23 行和第 24 行完成登录按钮内部 View 以及外部轮廓 View 的添加。

init 方法主要用于完成各种 View 的初始化工作，另外，在 init 方法中还可以将 label 的初始化工作也放到这部分来实现。下面是具体实现代码。

```
override init(frame: CGRect) {
       super.init(frame: frame)
       ……
1      label = UILabel(frame:CGRect(x: 0, y: 0,
```

```
                                width: button_w, height: button_h))
2       label!.text = "UpLoad"
3       label!.textAlignment = NSTextAlignment.center
4       label!.textColor = UIColor.white
5       label!.font = UIFont.systemFont(ofSize: 20.0)
6       self .addSubview(label!)
    }
```

代码第 1 行实例化一个 UILabel，第 2 行到第 4 行为 UILabel 实例对象分别添加 UpLoad 内容，文字居中对齐属性，设置当前 text 的内容颜色，并采用 20 号系统字体。最后一行将 label 添加到当前 Button 的 View 对象上。

完成以上的代码之后就可以在 ViewController 中调用该 Button 观察实现的效果。下面是最终调用代码。

```
1   var singleTap1:UITapGestureRecognizer?
2   var singleTap2:UITapGestureRecognizer?
3   var singleTap3:UITapGestureRecognizer?
4   var buttonview1:ButtonView?
5   var buttonview2:ButtonView?
6   var buttonview3:ButtonView?
7   override func viewDidLoad() {
8       super.viewDidLoad()
9       buttonview1 = ButtonView(frame:CGRect(x: 100, y: 150,
                                    width: 210, height: 70))
10      buttonview2 = ButtonView(frame:CGRect(x: 100, y: 275,
                                    width: 210, height: 70))
11      buttonview3 = ButtonView(frame:CGRect(x: 100, y: 400,
                                    width: 210, height: 70))
12      singleTap1 = UITapGestureRecognizer(target: self, action:
                        #selector(ViewController.viewAction1))
13      singleTap2 = UITapGestureRecognizer(target: self, action:
                        #selector(ViewController.viewAction2))
14      singleTap3 = UITapGestureRecognizer(target: self, action:
                        #selector(ViewController.viewAction3))
15      buttonview1?.addGestureRecognizer(singleTap1!)
16      buttonview2?.addGestureRecognizer(singleTap2!)
```

```
17      buttonview3?.addGestureRecognizer(singleTap3!)
18      self.view.addSubview(buttonview1!)
19      self.view.addSubview(buttonview2!)
20      self.view.addSubview(buttonview3!)
    }
```

代码第 1 行到第 6 行创建三个 Button 的 View 对象以及三个 Tap 的点击事件实例对象。第 9 行到第 11 行通过 frame 属性实例化三个 Button 按钮。第 12 二行到第 14 行为每个 Tap 点击事件添加一个响应方法(因为这里使用的是 View 模拟的 UIButton，所以为每个 View 添加一个 Tap 事件）。第 15 行到第 17 行为 Button 添加各自的响应 Tap 事件。最后分别将三个 Button 实例对象添加到 self.view 上。

如图 8.8 所示为最终实现的效果。


图 8.8　登录按钮初始化效果

经过以上的努力，三个登录按钮已经可以展现在界面上了，下面来实现图 8.5 的动画效果。登录按钮内部 View 以及外部边框在第一阶段动画过程中有各自的动画展示效果。下面是具体实现代码。

```
func startAnimation(){

1       label?.removeFromSuperview()
```

```
2       let animMakeBigger:CABasicAnimation = CABasicAnimation()
3       animMakeBigger.keyPath = "cornerRadius"
4       animMakeBigger.fromValue=5.0
5       animMakeBigger.toValue=button_h/2.0
6       let animBounds:CABasicAnimation = CABasicAnimation()
7       animBounds.keyPath = "bounds"
8       animBounds.toValue = NSValue(
              cgRect:CGRect(x: button_x+(button_w-button_h)/2,
              y: button_h,
              width: button_h,
              height: button_h))
9       let animAlpha:CABasicAnimation = CABasicAnimation()
10       animAlpha.keyPath = "opacity"
11       animAlpha.toValue = 0
12       let animGroup:CAAnimationGroup = CAAnimationGroup()
13       animGroup.duration = 1
14       animGroup.repeatCount = 1
15       animGroup.removedOnCompletion = false
16       animGroup.fillMode=kCAFillModeForwards
17       animGroup.timingFunction = CAMediaTimingFunction(
                    name:kCAMediaTimingFunctionEaseInEaseOut)
18       animGroup.animations =
                          [animMakeBigger,animBounds,animAlpha]
19       let animborder:CABasicAnimation = CABasicAnimation()
20       animborder.keyPath = "borderColor"
21       animborder.toValue = UIColor(red: 224.0/255,
          green: 224.0/255, blue:224.0/255, alpha: 1.0).CGColor
22       let animGroupAll:CAAnimationGroup = CAAnimationGroup()
23       animGroupAll.duration = 1
24       animGroupAll.repeatCount = 1
26       animGroupAll.removedOnCompletion = false
26       animGroupAll.fillMode=kCAFillModeForwards ;
27       animGroupAll.timingFunction = CAMediaTimingFunction(
                    name:kCAMediaTimingFunctionEaseInEaseOut)
28       animGroupAll.animations =
                          [animMakeBigger,animBounds,animborder]
```

```
29        animGroupAll.delegate = self
30        animGroupAll.setValue("allMyAnimationsBoard",
                                        forKey: "groupborderkey")
31        CATransaction.begin()
32        view?.layer.add(animGroup,
                                        forKey: "allMyAnimation")
33        viewborder?.layer.add(animGroupAll, forKey:
                                          "allMyAnimationsBoard")
34        CATransaction.commit()
      }
```

圆角动画：代码第 1 行移除当前 UILabel 的 UpLoad 文字内容。第 2 行和第 3 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为圆角渐变动画类型。第 4 行和第 5 行设置动画的变化范围从圆角半径 5.0 开始到 Button 按钮高度的一半。因为这里将圆角半径最终设置为 Button 按钮的一半，刚好可以构成一个圆形。

位置动画：代码第 6 行和第 7 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为位置动画类型。第 8 行设置 Button 按钮最终位置停留在当前 View 中间，并且宽高相等。

透明度动画：代码第 9 行和第 10 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为淡入淡出动画类型。代码第 11 行将透明度最终设置为 0。注意这里只能将该动画用于登录按钮的内部 View，而外部轮廓 View 不能使用该动画属性。

组合动画：代码第 12 行实例化 CAAnimationGroup 实例对象。第 13 行到第 17 行设置组合动画实例对象动画周期为 1s，重复次数为 1 次，不保留最后状态，并设置动画开始和结束时的动画效果。第 18 行将圆角动画、位置动画、透明度动画分别添加到组合动画中。

边框颜色渐变动画：代码第 19 行和第 20 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为边框颜色渐变动画类型。第 21 行将边框颜色设置为灰色。

组合动画：代码第 22 行实例化 CAAnimationGroup 实例对象。第 23 行到第 27 行设置组合动画实例对象的动画周期为 1s，重复次数为 1 次，不保留最后状态，并设置动画开始和结束时的动画效果。第 28 行将圆角动画、位置动画、边框颜色渐变动画分别添加到组合动画中。

第 31 行开始提交动画。第 32 行到第 33 行分别将组合动画 animGroup 和 animGroupAll 分别添加到 view 和 viewboard 上。第 34 行动画提交立即执行。如图 8.9 所示为最终实现效果。

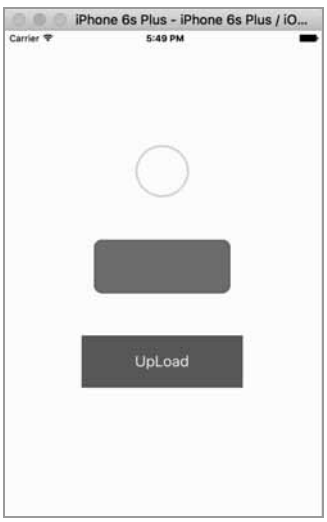



图 8.9　第一阶段动画效果

## 8.4.2　第二阶段动画

第二阶段动画的核心是绘制圆环，这里新建一个类 CircleView，采用贝济埃曲线绘制。下面是具体实现代码。

```
1   protocol CircleDelegate{
2      func circleAnimationStop()
 }
3   class CircleView: UIView,CAAnimationDelegate {
4      var lineWidth:NSNumber = 3.0
5      var strokeColor:UIColor = UIColor(red: 25.0 / 255.0, green:
```

```
                    155.0 / 255.0, blue: 200.0 / 255.0, alpha: 1.0)
6       var circle:CAShapeLayer = CAShapeLayer()
7       var delegate:CircleDelegate?
8       override init(frame: CGRect) {
9           super.init(frame: frame)
10          let startAngle:CGFloat = -90.0/180.0 * CGFloat(M_PI)
11          let endAngle:CGFloat = -90.01/180.0 * CGFloat(M_PI)
12          let circlePath:UIBezierPath = UIBezierPath(arcCenter:
          CGPoint(x: frame.size.width/2,y: frame.size.height/2),
            radius: frame.size.height/2-2,
            startAngle: startAngle,
            endAngle: endAngle,
            clockwise: true)
13          circle.path = circlePath.cgPath
14          circle.lineCap = kCALineCapRound
15          circle.fillColor = UIColor.clear.cgColor
16          circle.lineWidth = lineWidth as CGFloat
17          self.layer.addSublayer(circle)
    }
}
```

代码第 1 行创建一个协议，负责圆环完成之后回调到 ButtonView 类中。第 4 行和第 5 行设置线条宽度为 3.0，设置自定义颜色 strokeColor。第 6 行设置 CAShapeLayer 实例对象。第 7 行设置 delegate 代理属性。第 10 行和第 11 行设置圆圈动画的起始角度和最终停止角度。第 12 行使用贝济埃曲线绘制一个圆环。第 13 行到第 16 行设置圆环路径和线段的拐角形状，清空当前的圆环内部 view，将圆环的线条宽度设置为 3.0。第 17 行将圆环添加到 self.layer 图层上。

圆环绘制完之后，下面就要实现圆环滚动动画。下面是具体实现代码。

```
func strokeChart(){
1       circle.lineWidth = lineWidth as CGFloat
2       circle.strokeColor = strokeColor.cgColor
3       let pathAnimation:CABasicAnimation = CABasicAnimation()
4       pathAnimation.keyPath = "strokeEnd"
5       pathAnimation.delegate = self
```

```
    6       pathAnimation.duration = 3.0
    7       pathAnimation.timingFunction = CAMediaTimingFunction(
                        name:kCAMediaTimingFunctionEaseInEaseOut)
    8       pathAnimation.fromValue = 0.0
    9       pathAnimation.toValue = 1.0
    10      pathAnimation .setValue("strokeEndAnimationcircle",
                                    forKey: "groupborderkeycircle")
    11      circle.add(pathAnimation, forKey:
                                      "strokeEndAnimationcircle")
        }
    12    func animationDidStop(_ anim: CAAnimation,
           finished flag: Bool) {
           delegate!.circleAnimationStop()
        }
```

代码第 1 行和第 2 行设置圆环线条宽度以及圆环填充颜色。第 3 行和第 4 行实例化 CABasicAnimation 实例对象，并设置动画属性为 strokeEnd，表明当前该动画为逐渐绘制出来的效果。第 5 行到第 7 行设置当前动画回调对象，设置动画周期为 3s，设置动画起始和结束时的加速效果。第 8 行和第 9 行设置动画显示从无到有的效果。第 10 行设置动画 key－value 键值对。第 11 行将当前动画添加到圆环上并设置动画关键 key。第 12 行设置圆环绘制完成之后的回调方法，该方法也表明动画的第三阶段开始。

到这里代码第二部分已经完成，那么如何由第一部分进入第二部分呢？在第一部分动画已经设置了 delegate 回调对象，因此这里可以在第一部分动画完成的回调方法中启动第二部分动画。下面是具体实现代码。

```
    override func animationDidStop(anim: CAAnimation, finished flag:
Bool) {
    1       if(flag){
    2          let animType = anim.valueForKey("groupborderkey")
    3          let animType1 = anim.value(forKey: "groupborderkey1")
    4          if(animType != nil){
    5             if ((animType as! NSString).isEqual(
                     to: "allMyAnimationsBoard")){
```

```
6                   circleView?.strokeChart()
                }
7           } else if(animType1 != nil){
                if((animType1 as! NSString).isEqual(
                          to: "allMyAnimationsBoardspread1")) {
                    label = UILabel(frame:CGRect(x: 0, y: 0,
                             width: button_w, height: button_h))
                    label!.text = "Complete"
                    label!.textAlignment = NSTextAlignment.center
                    label!.textColor = UIColor.white
                    label!.font = UIFont.systemFont(ofSize: 20)
                    self.addSubview(label!)
                }
            }
        }
    }
```

代码第 1 行首先判断动画是否完成，第 2 行和第 3 行通过 key 获取当前动画类型。第 4 行和第 5 行通过动画 value 关键词可以判断动画是否为第一阶段，如果为第一阶段动画，那么调用 strockChart 方法，开启第二阶段动画。如图 8.10 所示为具体实现效果。

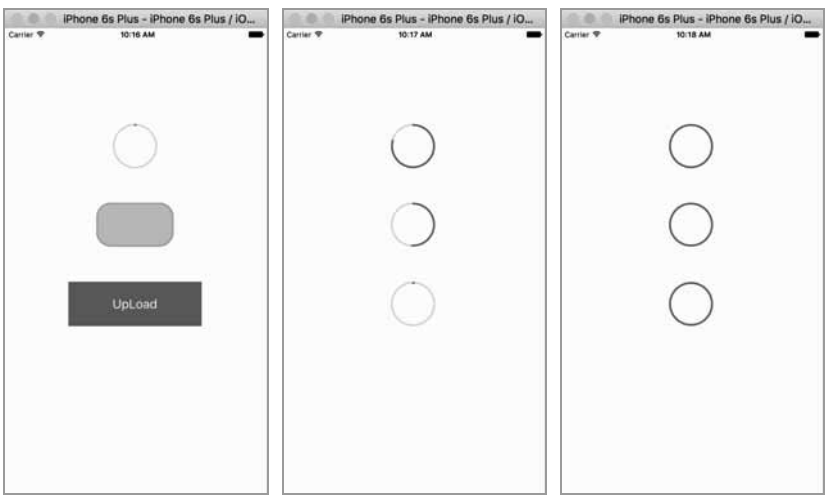



图 8.10　第二阶段动画效果

### 8.4.3 第三阶段动画

第三阶段动画和第一阶段动画类似。第一阶段实现动画收缩到圆环效果，第三阶段实现从圆环到展开效果。不同的是，在展开过程中颜色最终变为草绿色。下面是具体实现代码。

```
func startAnimationSpread(){
1       let animMakeBigger:CABasicAnimation = CABasicAnimation()
2       animMakeBigger.keyPath = "cornerRadius"
3       animMakeBigger.fromValue=button_h/2.0
4       animMakeBigger.toValue=0
5       let animBounds:CABasicAnimation = CABasicAnimation()
6       animBounds.keyPath = "bounds"
7       animBounds.fromValue = NSValue(cgRect:
              CGRect(x: button_x+(button_w-button_h)/2,
              y: button_h, width: button_h, height: button_h))
8       animBounds.toValue = NSValue(cgRect:CGRect(x: 0, y: 0,
                          width: button_w, height: button_h))
9       let animAlpha:CABasicAnimation = CABasicAnimation()
10      animAlpha.keyPath = "opacity";
11      animAlpha.toValue = 1.0
12      let animBackground:CABasicAnimation = CABasicAnimation()
13      animBackground.keyPath = "backgroundColor"
14      animBackground.toValue = FreshGreen.cgColor
15      let group:CAAnimationGroup = CAAnimationGroup()
16      group.duration = 1
17      group.repeatCount = 1
18      group.removedOnCompletion = false
19      group.fillMode=kCAFillModeForwards
20      group.timingFunction = CAMediaTimingFunction(
                    name:kCAMediaTimingFunctionEaseInEaseOut)
21     group.animations =
           [animMakeBigger,animBounds,animAlpha,animBackground]
22     let animBorder:CABasicAnimation = CABasicAnimation()
23     animBorder.keyPath = "borderColor"
24     animBorder.toValue = FreshGreen.cgColor
```

```
25      let allGroup:CAAnimationGroup = CAAnimationGroup()
26      allGroup.duration = 1
27      allGroup.repeatCount = 1
28      allGroup.isRemovedOnCompletion= false
29      allGroup.fillMode=kCAFillModeForwards
30      allGroup.timingFunction = CAMediaTimingFunction(
                        name:kCAMediaTimingFunctionEaseInEaseOut)
31      allGroup.animations =
                [animMakeBigger,animBounds,animAlpha,animBorder]
32      CATransaction.begin()
33      view?.layer.add(group,
                                forKey: "allMyAnimationspread1")
34      allGroup.setValue("allMyAnimationsBoardspread1",
                                forKey: "groupborderkey1")
35      allGroup.delegate = self
36      viewborder?.layer.add (allGroup,
                          forKey: "allMyAnimationsBoardspread1")
37      CATransaction.commit()
    }
```

圆角动画：代码第 1 行和第 2 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为圆角渐变动画类型。第 3 行和第 4 行设置动画的变化范围从圆环半径开始到 0，因为展开效果最终使得登录按钮展开为长方形。

位置动画：代码第 5 行和第 6 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为位置动画类型。第 7 行设置 Button 按钮的最终位置停留在当前 View 中间，并且宽高最终展开为长方形。

透明度动画：代码第 9 行和第 10 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为淡入淡出动画类型。第 11 行将透明度最终设置为 1。

背景颜色渐变动画：代码第 12 行到第 13 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为颜色渐变动画类型。第 14 行设置背景颜色为绿色。

组合动画：代码第 15 行实例化 CAAnimationGroup 实例对象。第 16 行到第 20 行设置组合动画实例对象动画周期为 1s，重复次数为 1 次，不保留最后状态，

并设置动画开始和结束时的动画效果。第 21 行将圆角动画、位置动画、透明度动画、背景颜色渐变动画分别添加到组合动画中。

边框颜色渐变动画：代码第 22 行和第 23 行实例化 CABasicAnimation 动画实例对象，并将动画属性设置为边框颜色渐变动画类型。第 24 行将边框颜色设置为绿色。

组合动画：代码第 25 行实例化 CAAnimationGroup 实例对象。第 26 行到第 27 行设置组合动画实例对象动画周期为 1s，重复次数为 1 次，不保留最后状态，并设置动画开始和结束时的动画效果。第 31 行将圆角动画、位置动画、透明度动画、边框颜色渐变动画分别添加到组合动画中。

代码从第 32 行开始提交动画。第 33 行到第 34 行分别将组合动画 animGroup 和 animGroupAll 添加到 view 和 viewboard 上。第 37 行动画提交立即执行。如图 8.11 所示为最终实现效果。

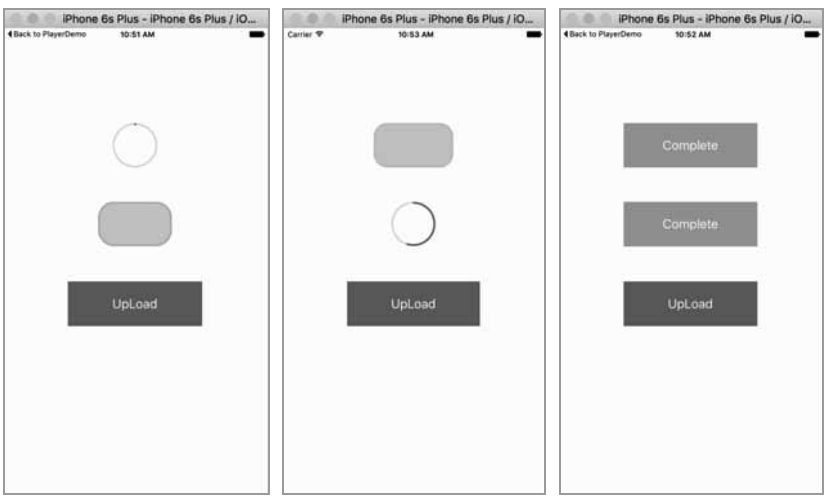



图 8.11　最终动画效果

## 8.5　本章小结

本章通过两个实例介绍了水纹登录按钮和登录按钮收缩、展开动画效果。

水纹动画效果其核心是利用定时器定时刷帧的方式实现水纹效果。而其中的水纹图片，则是通过 Draw 方法重绘得到的最终效果。该案例所涉及的知识都为第一卷为大家介绍的相关知识点。

登录按钮收缩、展开动画效果主要体现在对 Layer 图层相关动画的综合使用上，在该动画中大量使用了 CABasicAnimation 中的位置、圆角、背景颜色渐变等各种动画效果。

至此为大家介绍了视图层和内容层的所有基础动画效果，基础动画效果的实现较为简单，而通过对这些基础知识点的灵活掌握可以组合出各种各样绚丽复杂的特效动画。

# 第 9 章　CAEmitterCell 粒子动画效果

## 本章内容

- 理解 iOS 粒子系统组成
- 掌握火焰燃烧、烟雾等粒子动画效果

读到这里相信大家已经被 iOS 各种各样的动画实现方法和系统提供的动画特效功能所震撼，其实 iOS 远不止我们所了解的这些，它还有更多、更丰富的动画特性等待大家挖掘。iOS 除了前 8 章为大家介绍的动画效果之外，还有一些非常炫酷的粒子特性，通过这些粒子特性可以设计出各种各样炫酷的动画效果。在本章将结合 CAEmitterLayer 和 CAEmitterCell 这两个类为大家实现火焰燃烧、霓虹等粒子动画效果。

## 9.1　iOS 粒子系统概述

在 iOS 系统中，粒子系统由两部分组成：CAEmitterLayer 和 CAEmitterCell。

（1）CAEmitterLayer 为粒子发射图层。该图层主要用于控制粒子展现范围、粒子发射位置、粒子发射形状、渲染模式等属性。通过 CAEmitterCell 构建的发射单元都受到 CAEmitterLayer 图层节制，可以说粒子展现必须在

CAEmitterLayer 图层上来实现。

（2）CAEmitterCell 粒子发射单元，用于对粒子系统中的单个粒子做更加精细的控制。比如控制粒子的移动速度、方向、范围。在 CAEmitterCell 类中提供了几十种粒子属性参数设置，所以结合这些属性可以制作各种炫酷的粒子特效动画。

如何在 iOS 中使用这两个类呢？首先这里使用的 CAEmitterLayer 属于 CALayer 图层，所以实例化 CAEmitterLayer 图层之后将其添加到相应 View 的 layer 图层上即可。而 CAEmitterLayer 实例对象有一个 emitterCells 属性，该属性描绘了所有的粒子实例和相应特性，因此将 CAEmitterCell 实例对象传递给 CAEmitterLayer、emitterCells 属性就可以实现粒子动画效果。

## 9.2　案例：粒子火焰效果

iOS 中火焰效果是由一个个 iOS 的粒子特性描绘出来的，所以每个粒子的具体特性将直接决定火焰的具体形状，下面对粒子常用属性做一个分析。

（1）birthRate：表明火焰效果中每秒粒子的组成个数，个数越多，火焰越大越逼真。如图 9.1 所示为不同粒子个数下火焰的实际效果。

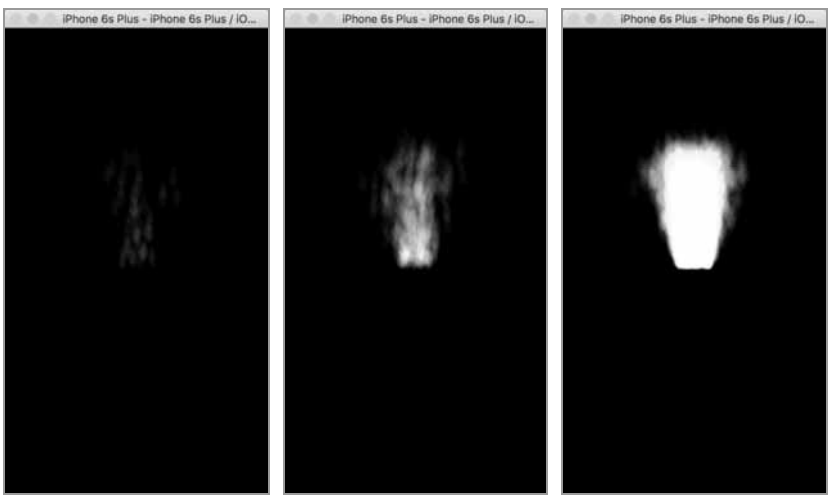



图 9.1　每秒粒子数分别为 50、500、5000 时的火焰燃烧效果

通过图 9.1 可以看出，虽然每秒粒子数反应了火焰的燃烧效果，但并不是粒子数越多越好。粒子数过多一方面会掩盖每个粒子的一些具体细节，另一方面会给 iOS 图像渲染造成很大的负担，所以这里将粒子个数设置为 500 较为合适。

（2）velocity：表明当前粒子速度信息，不同的粒子速度控制火焰的燃烧剧烈程度以及火焰的燃烧方向。如图 9.2 所示为不同粒子速度下火焰的燃烧情况。

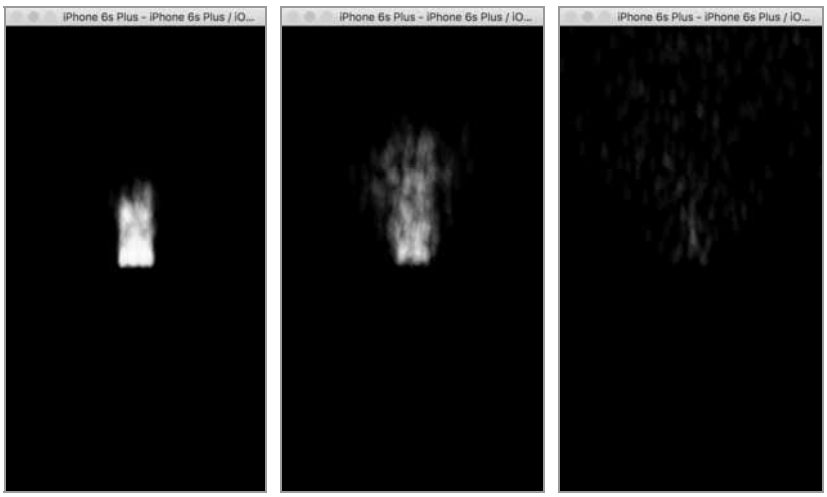



图 9.2　粒子速度分别为 1、100、1000 时的火焰燃烧剧烈程度

通过图 9.2 可以看出，粒子速度越大，火苗喷涌速度越快。所以当粒子速度为 100 时，模拟火焰燃烧效果较为合适，当粒子速度达到 1000 时，可以用来模拟火山喷涌效果。

当然，除了以上介绍的两种属性之外，粒子系统还有大量的属性设置用于更加细微地描述当前的粒子动画效果，不过由于篇幅关系这里不再做进一步的探讨。

## 9.3　案例：“鬼火”火焰效果代码实现

下面是粒子“鬼火”火焰效果的具体实现。

```
override func viewDidLoad() {
```

```
1       super.viewDidLoad()
2       self.view.backgroundColor = UIColor.black
3       let emitterCell = CAEmitterCell()
4       emitterCell.name = "fire"
5       emitterCell.emissionLongitude  = CGFloat(M_PI)
6       emitterCell.velocity        = -1//粒子速度 负数表明向上燃烧
7       emitterCell.velocityRange       = 50//粒子速度范围
8       emitterCell.emissionRange       = 1.1//周围发射角度
9       emitterCell.yAcceleration       =-200//粒子y方向的加速度分量
10      emitterCell.scaleSpeed          = 0.3//缩放比例 超大火苗
11      emitterCell.color = UIColor(red: 0.8 ,
                green:0.4 ,blue:0.2 ,alpha:0.1).CGColor
12      emitterCell.contents = UIImage(
                                    named: "fire.png")!.cgImage
13      let emitterLayer = CAEmitterLayer()
14      emitterLayer.position = self.view.center// 粒子发射位置
15      emitterLayer.emitterSize = CGSize(width: 50,
                                      height: 10)// 控制火苗大小
16      emitterLayer.renderMode = kCAEmitterLayerAdditive
17      emitterLayer.emitterMode = kCAEmitterLayerOutline
18      emitterLayer.emitterShape = kCAEmitterLayerLine
19      emitterLayer.emitterCells = [emitterCell]
20      self.view.layer.addSublayer(emitterLayer)
21      emitterLayer.setValue(500,
                    forKeyPath: "emitterCells.fire.birthRate")
22      emitterLayer.setValue(1,
                    forKeyPath: "emitterCells.fire.lifetime")
    }
```

代码第 2 行设置当前视图背景颜色为黑色。第 3 行定义粒子单元 cell。第 4 行定义该粒子单元名称为 fire 火焰。第 5 行定义该粒子系统在 xy 平面的发射角度。第 6 行设置粒子速度，当前设置为−1，表明沿 *y* 轴向反方向运动。第 7 行设置粒子速度范围为 50。第 8 行设置发射角度为 1.1。第 9 行设置粒子在 *y* 轴方向上的加速度分量。第 10 行描绘粒子的缩放系数。第 11 行设置当前火焰颜色

为红色。既然要实现火焰燃烧的粒子效果，这里必须要设置粒子的“种子”内容。第 12 行设置粒子效果的内容。第 13 行定义粒子的发射图层。第 14 行确定粒子的发射位置。第 15 行控制当前火苗的大小。第 16 行到第 18 行确定当前火苗渲染模式以及发射源模式。第 19 行将当前的粒子单元部署到粒子发射图层中。第 20 行将粒子发射图层添加到当前视图控制器的 Layer 图层上。最后两行设置粒子的生成速度以及粒子生命周期。如图 9.3 所示为最终实现的“鬼火”燃烧效果。

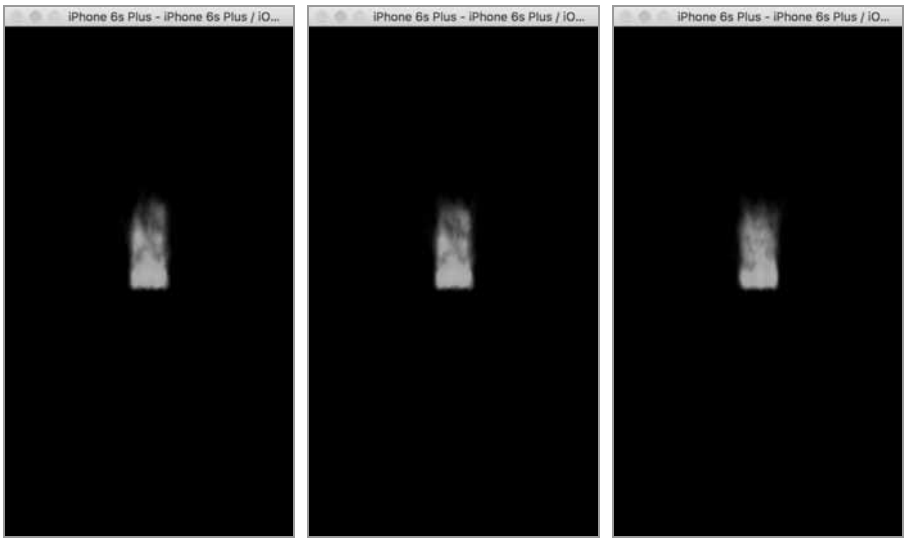



图 9.3　鬼火燃烧效果

## 9.4　案例：霓虹效果代码实现

霓虹效果的实现和火焰燃烧类似，都是通过 iOS 的粒子来进行描述的，同样，在开始进行代码实现之前，需要先准备好一张霓虹效果的“种子”图片，如图 9.4 所示。


图 9.4　粒子效果种子素材

下面是代码具体实现。

```
override func viewDidLoad() {
1       super.viewDidLoad()
2       self.view.backgroundColor = UIColor.black
3       let emitterCell = CAEmitterCell()
4       emitterCell.name = "nenolight"
5       emitterCell.emissionLongitude  = CGFloat(M_PI*2)
6       emitterCell.velocity = 50// 粒子速度
7       emitterCell.velocityRange = 50// 粒子速度范围
8       emitterCell.scaleSpeed = -0.2
9       emitterCell.scale = 0.1
10      emitterCell.greenSpeed = -0.1
11      emitterCell.redSpeed = -0.2
12      emitterCell.blueSpeed = 0.1
13      emitterCell.alphaSpeed = -0.2
14      emitterCell.birthRate = 100
15      emitterCell.lifetime = 4
16      emitterCell.color = UIColor.white.cgColor
17      emitterCell.contents = UIImage(
                              named: "neonlight.png")!.CGImage
18      let emitterLayer = CAEmitterLayer()
19      emitterLayer.position = self.view.center// 粒子发射位置
20      emitterLayer.emitterSize = CGSize(x:2,y:2)// 控制粒子大小
21      emitterLayer.renderMode = kCAEmitterLayerBackToFront
22      emitterLayer.emitterMode = kCAEmitterLayerOutline
23      emitterLayer.emitterShape = kCAEmitterLayerCircle
24      emitterLayer.emitterCells = [emitterCell]
25      self.view.layer.addSublayer(emitterLayer)
    }
```

代码第 2 行设置当前视图背景颜色为黑色。第 3 行定义粒子单元 cell。第 4 行定义该粒子单元名称为 nenolight 霓虹效果。第 5 行定义该粒子系统在 xy 平面的发射角度。第 6 行设置粒子速度，当前设置为 50。第 7 行设置粒子速度范围为 50。第 8 行设置速度缩放比例为−0.2。

第 9 行设置缩放比例为 0.1。第 10 行到第 11 行分别设置 R、G、B、alpha 的颜色速度渐变效果。第 14 行设置粒子生成速度。第 15 行设置粒子生命周期。第 16 行设置粒子的背景颜色。第 17 行设置粒子效果的内容。第 18 行定义粒子的发射图层。第 19 行确定粒子的发射位置。第 20 行控制当前霓虹的大小。第 21 行到第 23 行确定当前渲染模式以及发射源模式。第 24 行将当前的粒子单元部署到粒子发射图层中。最后一行将粒子发射图层添加到当前视图控制器的 Layer 图层上。如图 9.5 所示为霓虹最终生成效果。

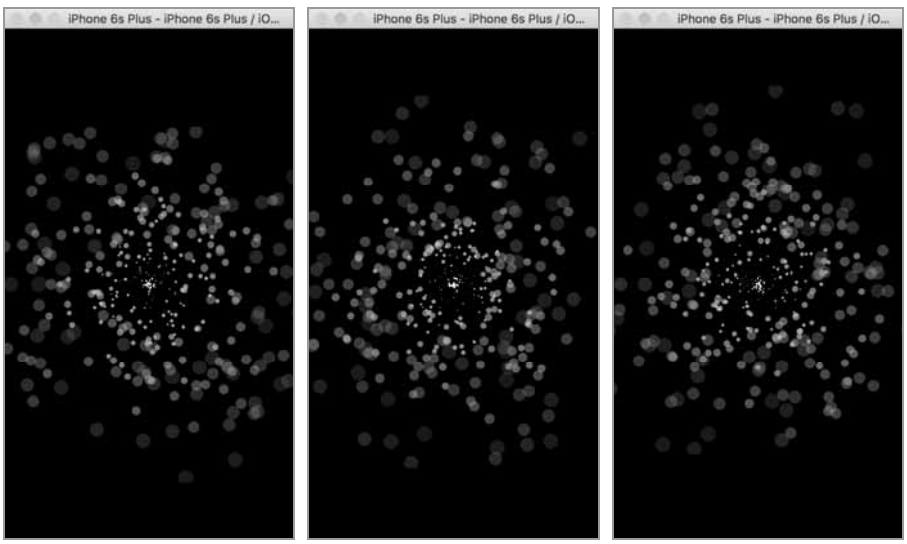



图 9.5　霓虹粒子动画效果

## 9.5　本章小结

无论是现在主流的 Cocos2d 还是绚丽的 Unity3D，都具有类似的绚丽粒子效

果。而在 iOS 使用 CAEmitterCell 和 CAEmitterLayer 就可以模拟出这些绚烂的粒子效果。iOS 中粒子单元使用简单，属性强大，可以设置包括方向、速度、扩展形状等在内的各种动画效果。本章结合火焰和霓虹效果为大家展示了 iOS 中的粒子效果的基本使用。对于粒子效果大家可以自己通过修改粒子的各种属性进一步挖掘粒子属性的深层使用。

# 第 10 章　CoreAnimation: CAGradientLayer 光波扫描动画效果

## 本章内容

- 掌握光波扫描“梯度”的基本原理
- 实现指纹“光波扫描”效果、音响音量跳动效果

我们在观看科幻大片或者开启重磅音响时,经常会看到如图 10.1 所示的“光波扫描”和如图 10.2 所示的“音量跳动”动画效果，每次看到这些炫酷的动画效果都让人激动不已。本章将使用 QuartzCore 框架下的 CAGradientLayer 类揭开“指纹光波扫描”动画和“音量跳动动画”的神秘面纱。

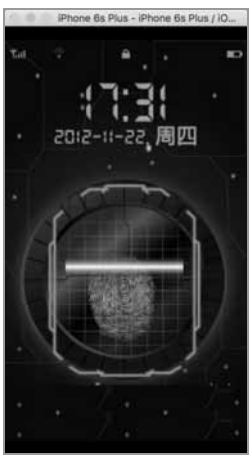



图 10.1　光波扫描效果

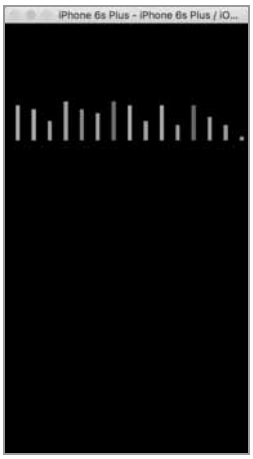



图 10.2　音量跳动效果

# 10.1　CAGradientLayer 追本溯源

先来追本溯源一下 CAGradientLayer 这个单词。CAGradientLayer 可以拆解为三个主要组成部分：CA、Gradient、Layer。CA 为 CoreAnimation 的缩写，表明当前的动画使用的是 iOS 框架下的核心动画部分。Gradient 为梯度的意思，描述了当前动画的特点，实现一些梯度功能的动画效果。比如位置的梯度变化、颜色的梯度渐变等。Layer 表明当前动画并非直接作用于 UIView 显示层上，而是作用在 Layer 内容层。到这里大家就基本搞清楚了 CAGradientLayer 类的基本含义，接下来想一想如何实现这一动画效果。

相信读者已经阅读了本书的第 1 章内容，具备了动画分析的基本思想。按照第 1 章为大家总结的动画分析方法开始分析。要想利用动画实现的三步曲完成动画的设计，首先要搞清楚动画的算法原理。下面对动画进行逐帧分解，如图 10.3 所示为动画分解效果。

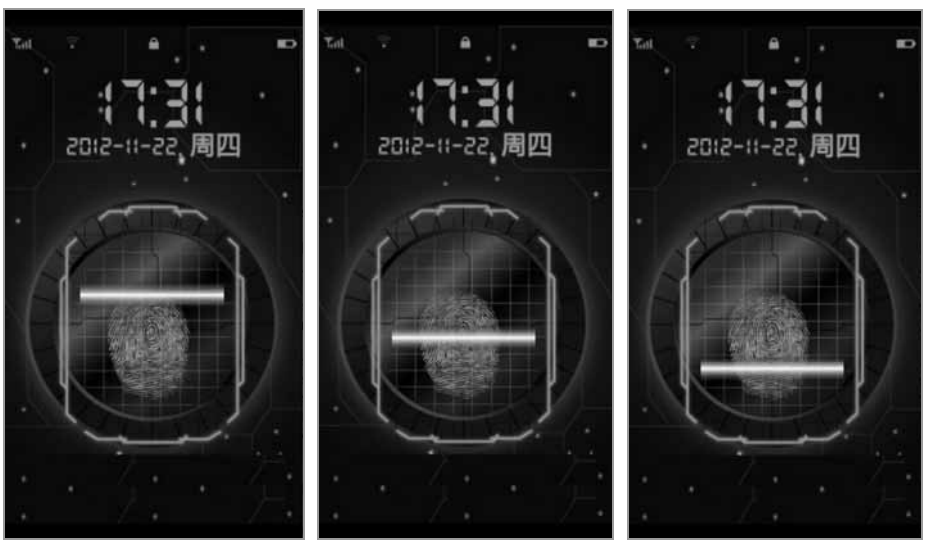



图 10.3　指纹扫描动画多帧分解效果

# 10.2 光波效果实现原理分析

通过对动画的分解，基本上可以弄清楚动画的演变过程。接下来还需要弄清楚以下几个主要问题。

（1）光波的执行方向。

（2）光波的颜色梯度。

（3）光波的“彗星拖尾”效果。

只有先把这些问题研究透彻了，才能更好地实现图 10.1 所示的指纹扫描效果。下面就从“原理+代码”的角度来阐释这些概念的具体含义。

## 10.2.1 光波方向

如图 10.4 所示，左图为光波从上到下执行顺序的效果图，右图为光波从左到右执行顺序的效果图。

图 10.4 光波方向

效果已经为大家展示清楚了，那么接下来思考如何通过代码实现这种效果。下面是光波方向设置的核心代码。gradientLayer 为实例化的 CAGradientLayer 实例对象。将 startPoint 设置为（0，0），endPoint 设置为（1，0）即可实现光波从上到下的扫描效果。

```
//      设置光波起始位置和终止位置坐标
        gradientLayer.startPoint = CGPoint(x:0,y:0)
        gradientLayer.endPoint = CGPoint(x:1,y:0)
```

接下来再为大家画一张原理图来分析 startPoint 和 endPoint 的原理。在实例化 CAGradientLayer 时需要设置其 frame。而 startPoint 和 endPoint 与当前 frame 存在一定的映射关系，如图 10.5 所示为二者之间坐标系的映射关系。


图 10.5　frame 坐标系与“梯度”坐标系对应关系

图 10.5 为 frame 坐标系与“梯度”坐标系的对应关系。其中浅色部分为 startPoint 和 endPoint 的坐标系，黑色部分为实例对象 gradientLayer 的 frame 坐标系。这里把 startPoint 和 endPoint 的坐标系暂且称之为“梯度坐标系”。梯度坐标系范围在 0～1 之间，startPoint 和 endPoint 设置为（0，0）和（0，1）。如图 10.6 所示为梯度坐标系原理图和效果图。

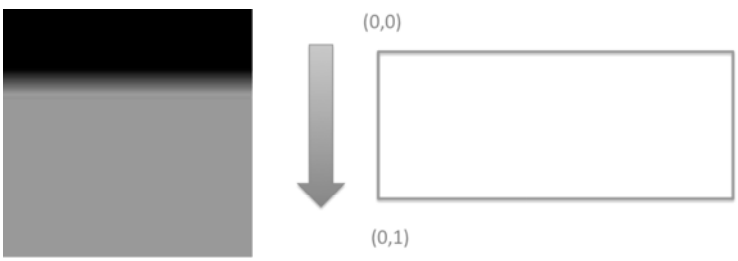



图 10.6　梯度坐标系原理图和效果图

如果想把动画运动方向从上到下改为从左到右，那么只需要将原理图中 startPoint 和 endPoint 分别修改为（0，0）和（1，0）。如图 10.7 所示为水平方向上梯度坐标系原理图和效果图。

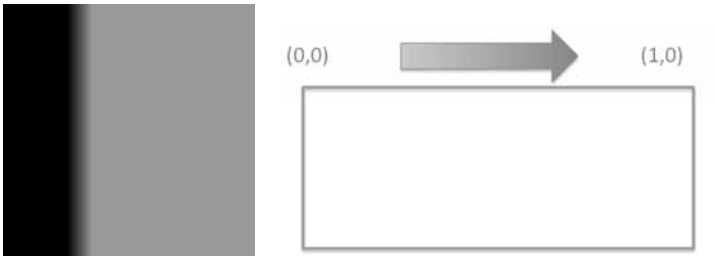



图 10.7　梯度坐标系原理图和效果图

梯度坐标系是一个二维坐标系，除了水平和竖直方向两种情况之外，还有斜线方向。接下来思考一下如何实现斜线方向。如果将 startPoint 设置为（0，0.5），endPoint 设置为（1，0）那么会出现什么效果呢？先来分析原理图，如图 10.8 所示。


图 10.8　动画斜线方向移动效果图

起始坐标设置为（0，0.5），动画结束坐标设置为（1，0），那么把两点连接起来就是动画移动的起始位置和方向。如图 10.9 所示展示了动画的斜线移动方向。

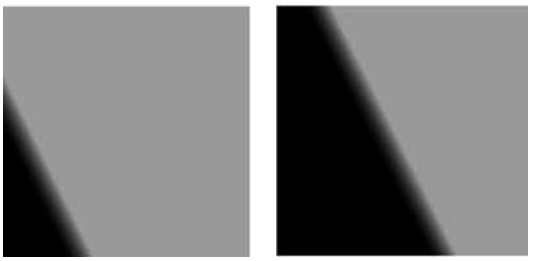


图 10.9　斜线移动方向动画效果

## 10.2.2　光波颜色梯度

从图 10.9 中可以看出光波颜色梯度分为两个等级，一个为黑色，另一个为粉红色。下面看看代码如何实现。

```
        let color = UIColor(red: 216.0 / 255.0, green: 114.0 / 255.0,
                            blue: 213.0 / 255.0, alpha: 1.0)
//      设置光波颜色梯度
        gradientLayer.colors = [UIColor.clear.cgColor,
                                                color.cgColor]
```

GradientLayer 实例对象的 colors 属性是一个颜色数组，其中数组的第一个元素为 UIView 的底色。当扫描动画扫描完成之后就显示出底色“blackColor”。数组的第二个元素为动画逐渐褪去的颜色。如果设置三种颜色会有什么效果呢？如图 10.10 所示为颜色梯度变化。

图 10.10　颜色梯度变化

可以把代码修改为如下形式来实现图 10.10 的梯度变化效果。

```
        let color = UIColor(red: 216.0 / 255.0, green: 114.0 / 255.0,
                            blue: 213.0 / 255.0, alpha: 1.0) // 粉色
        let color1 = UIColor(red: 61.0 / 255.0, green: 226.0 / 255.0,
                            blue: 210.0 / 255.0, alpha: 1.0)// 青色
//      设置光波颜色梯度
        gradientLayer.colors = [UIColor.clear.cgColor,
                                                color1.cgColor,
                                                 color.cgColor]
```

可以看出颜色数组中第一个颜色为 UIView 最终底色，第二个元素为光波渐变颜色，第三个颜色为原本覆盖在 UIView 上的动画颜色。

### 10.2.3　光波“彗星拖尾”效果

在正式讲解光波“彗星拖尾”效果之前，先来认识一下 gradientLayer 的颜

色分割属性。这个属性可以很好地帮助我们理解光波“彗星拖尾”效果的原理。把颜色分割属性按照如下代码进行设置。

```
            let color = UIColor(red: 216.0 / 255.0, green: 114.0 / 255.0,
blue: 213.0 / 255.0, alpha: 1.0)// 粉色
            let color1 = UIColor(red: 61.0 / 255.0, green: 226.0 / 255.0,
blue: 210.0 / 255.0, alpha: 1.0)// 青色
    //      设置光波颜色梯度
            gradientLayer.colors = [UIColor.clear.cgColor,
                                                color1.cgColor,
                                                color.cgColor]
            gradientLayer.locations = [0.0,0.1,0.2]
```

最终光波效果如图 10.11 所示。其中 A 到 B 之间的区域为从黑色渐变到青色，B 到 C 之间的区域为从青色渐变到粉色。经过测量，A 与 B 之间的区域大约为整个视图面积的 10%左右，B 与 C 之间的区域面积为整个视图面积的 10%。结合代码：

```
    gradientLayer.locations = [0.0,0.1,0.2]
```

其中 locations 为颜色渐变比例数组。通过图 10.5 可以知道，在“梯度”坐标系中位置坐标的范围在 0～1 之间。而 locations 数组中的三个值分别对应 A、B、C 三个点的位置坐标。由此可知 locations 中 0.0～0.1 描述的即为 A～B 之间的区域，在这片区域中完成了从黑色到青色的颜色渐变过程。locations 中 0.1～0.2 描述的即为 B～C 之间的区域，在这片区域中完成了从青色到粉红色的颜色渐变过程。

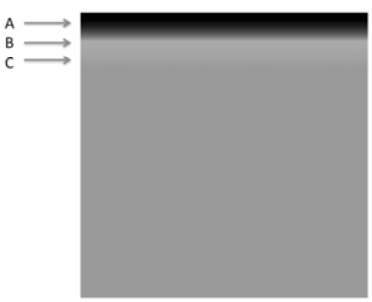



图 10.11　光波颜色“拖尾”效果

在理解了以上的知识之后，再把 A 与 B 之间“彗星拖尾”的效果加深一些。将位置坐标修改为：

```
gradientLayer.locations = [0.0,0.5,0.6]
```

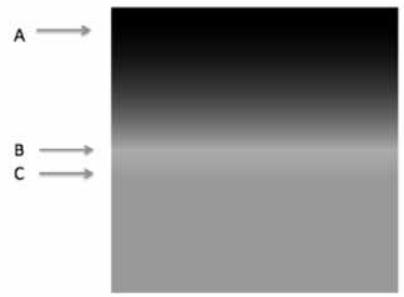



图 10.12　光波颜色“拖尾”增强效果

可见经过修改之后，A 与 B 之间的拖尾效果明显加强了。

## 10.2.4　光波扫描效果

光波的扫描效果采用 CABasicAnimation 动画来实现。只需要把光波 gradientLayer 的 locations 的属性设置成可变状态就可以实现最终的效果。下面是具体代码实现。

```
1       let gradientAnimation:CABasicAnimation = CABasicAnimation()
2       gradientAnimation.keyPath = "locations"//动画属性
3       gradientAnimation.fromValue = [0.0,0.1,0.2];
//动画属性变化起始状态值
4       gradientAnimation.toValue = [0.8,0.9,1.0];
//动画属性变化终止状态值
5       gradientAnimation.duration = 3.0;//动画执行周期
6       gradientAnimation.repeatCount = 10;//动画执行重复次数
7       gradientLayer.add (gradientAnimation, forKey: nil)
//将基础动画添加到 Layer 图层
```

代码第 1 行初始化 CABasicAnimation 动画实例对象，第 2 行指明当前动画需要实现的动画属性。这里修改的是 locations 属性，而 locations 属性对应的效果如图 10.11 和图 10.12 所示。可见，修改 locations 可以修改光波的位置和拖尾

的效果。第 3 行和第 4 行修改动画初始状态的 locations 值以及动画最终状态 locations 值。第 5 行和第 6 行设置动画执行周期为 3s，第 6 行设置动画重复执行 10 次。最后一行将代码添加到 Layer 图层上，启动当前动画效果。最终的动画效果如图 10.13 所示。

图 10.13　光波扫描效果

## 10.3　案例：指纹扫描效果

在学习了第 10.2 节光波原理分析之后，相信大家已经有了足够的知识来实现图 10.1 的指纹扫描效果。这里把整个动画效果按功能分解为三个部分（因为这个动画效果比较简单所以没有按照动画的三步曲进行分析，而是根据功能进行划分）。第一部分实现了指纹背景图片的添加，第二部分设置 Layer 图层的相关属性，第三部分利用 CABasicAnimation 实现光波的运动效果。下面是图 10.1 指纹扫描动画效果的全部代码。

```
//      part1:设置指纹扫描背景图片
1       let image:UIImage = UIImage(named: "unLock.jpg")!
2       let imageView:UIImageView = UIImageView(image: image)
3       imageView.contentMode = UIViewContentMode.ScaleAspectFit
4       imageView.frame = self.view.bounds
5       imageView.center = self.view.center
6       self.view.backgroundColor = UIColor.blackColor()
7       self.view.addSubview(imageView)
//      part2:设置 Layer 图层属性
```

```
8       let gradientLayer:CAGradientLayer = CAGradientLayer()
9       gradientLayer.frame = CGRect(x: 105, y: 330,
                                     width: 200,height: 200)
10       imageView.layer.addSublayer(gradientLayer)
11       gradientLayer.startPoint = CGPoint(x:0,y:0)
12       gradientLayer.endPoint = CGPoint(x:,y:1)
13       gradientLayer.colors = [UIColor.clear.cgColor,
                                 UIColor.white.cgColor,
                                 UIColor.clear.cgColor]
14       gradientLayer.locations = [0.0,0.1,0.2]
//      part3:设置 CABasicAnimation
15       let gradientAnimation:CABasicAnimation =
                                          CABasicAnimation()
16       gradientAnimation.keyPath = "locations"
17       gradientAnimation.fromValue = [0.0,0.1,0.2];
18       gradientAnimation.toValue = [0.8,0.9,1.0];
19       gradientAnimation.duration = 3.0;
20       gradientAnimation.repeatCount = 10;
21      gradientLayer.add (gradientAnimation, forKey: nil)
```

代码第一部分：添加指纹背景图片，并设置图片拉伸方式和 frame 等属性添加到 self.view 上。代码第二部分：初始化 Layer 属性，并将 Layer 图层添加到 imageView 的 Layer 图层上。代码第 11 行到第 14 行设置 Layer 图层的相关属性，具体属性设置可以参考第 10.2 节光波原理分析。代码第三部分设置 CABasicAnimation 动画。我们需要修改动画的 locations 属性，并设置动画的起始位置为［0.0，0.1，0.2］，终止位置为［0.8，0.9，1.0］。设置动画执行周期 3s，执行次数 10 次。最后一行将动画添加到 Layer 图层上，启动当前动画。

## 10.4　案例：音响音量跳动效果

在这个案例中将实现一个类似音量跳动的动画效果，如图 10.14 所示为最终要实现的音量跳动动画效果图。整个动画效果由 15 根柱状 View 组成，并且不同的 View 之间设定了几组随机颜色。

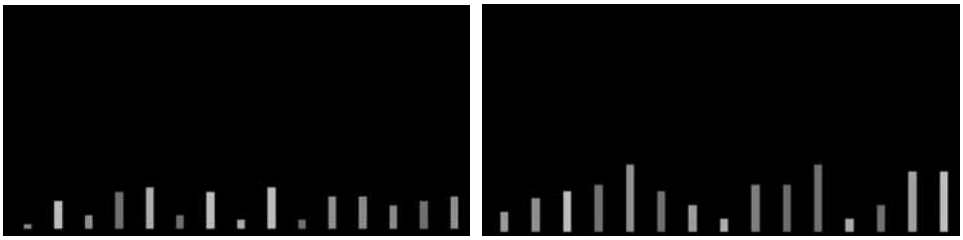


图 10.14　音量跳动动画效果

所以将动画的设计思路总结为以下内容。

（1）动画起始阶段：一组随机颜色。

（2）动画起始阶段：15 根柱状 UIView 视图。

（3）动画进行阶段：如图 10.15 所示。

（4）动画结束阶段。

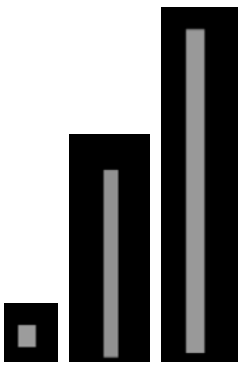


图 10.15　单根柱状图动画渐变过程

### 1．动画起始阶段

先来实现动画的起始阶段，将随机颜色和 15 根柱状图绘制到当前屏幕上。下面是准备的 11 组随机颜色代码。通过颜色的 RGB 属性，生成 11 组随机颜色，并将颜色存储到颜色数组 colorArray 中。代码如下所示。

```
func setcolorArray(){
1       colorArray = NSMutableArray()
2       let  color1:UIColor = UIColor(red: 255.0 / 255.0,
```

```
            green: 127.0 / 255.0, blue: 79.0 / 255.0, alpha: 1.0)
3       let  color2:UIColor = UIColor(red: 138.0 / 255.0,
            green: 206.0 / 255.0, blue: 245.0 / 255.0, alpha: 1.0)
4       let  color3:UIColor = UIColor(red: 216.0 / 255.0,
             green: 114.0 / 255.0, blue: 213.0 / 255.0, alpha: 1.0)
5       let  color4:UIColor = UIColor(red: 51.0 / 255.0,
            green: 207.0 / 255.0, blue: 48.0 / 255.0, alpha: 1.0)
6       let  color5:UIColor = UIColor(red: 102.0 / 255.0,
            green: 150.0 / 255.0, blue: 232.0 / 255.0, alpha: 1.0)
7       let  color6:UIColor = UIColor(red: 255.0 / 255.0,
            green: 105.0 / 255.0, blue: 177.0 / 255.0, alpha: 1.0)
8       let  color7:UIColor = UIColor(red: 187.0 / 255.0,
            green: 56.0 / 255.0, blue: 201.0 / 255.0, alpha: 1.0)
9       let  color8:UIColor = UIColor(red: 255.0 / 255.0,
            green: 163.0 / 255.0, blue: 0.0 / 255.0, alpha: 1.0)
10      let  color9:UIColor = UIColor(red: 203.0 / 255.0,
            green: 93.0 / 255.0, blue: 92.0 / 255.0, alpha: 1.0)
11      let  color10:UIColor = UIColor(red: 61.0 / 255.0,
            green: 226.0 / 255.0, blue: 210.0 / 255.0, alpha: 1.0)
12      let  color11:UIColor = UIColor(red: 25.0 / 255.0,
            green: 146.0 / 255.0, blue: 255.0 / 255.0, alpha: 1.0)
13      colorArray.addObject(color1)
14      colorArray.addObject(color2)
15      colorArray.addObject(color3)
16      colorArray.addObject(color4)
17      colorArray.addObject(color5)
18      colorArray.addObject(color6)
19      colorArray.addObject(color7)
20      colorArray.addObject(color8)
21      colorArray.addObject(color9)
22      colorArray.addObject(color10)
23      colorArray.addObject(color11)
    }
```

在 ViewDidload()方法中将所需要的 15 根柱状 View 绘制到当前视图上，代码如下所示。

```
override func viewDidLoad() {
       super.viewDidLoad()
1       setcolorArray()
2       self.view.backgroundColor = UIColor.blackColor()
3       audioBarNum = 15;
4       for i in 0...audioBarNum {
5          let h:CGFloat = 150
6          let w:CGFloat = (self.view.frame.size.width-10)/
                        CGFloat(audioBarNum)
7          let x:CGFloat = 20
8          let y:CGFloat = 50
9          let view:UIView = UIView(frame:
                              CGRect(x: w*CGFloat(i)+x, y: y,
                              width: w-x, height: h))
10          self.view.addSubview(view)
       }
   }
```

代码第 1 行初始化 11 组随机颜色。第 2 行将 self.view 的背景颜色设置为黑色。第 3 行设置当前动画音量的柱状图的个数。第 4 行采用 for 循环的形式依次添加 15 根柱状图。代码第 5 到第 8 行设置柱状图的 frame 位置坐标。第 9 行初始化单根柱状图。最后一行将 15 根柱状图依次添加到 self.view 图层上。

### 2．动画进行阶段

既然每根音量柱状图都要实现图 10.15 的动画效果，所以在创建每个单根柱状图的时候都可以为其添加一个 CAGradientLayer 实例图层。以上 for 循环中完整代码如下所示：

```
for i in 0...audioBarNum {
          let h:CGFloat = 150
          let w:CGFloat =
          (self.view.frame.size.width-10)/CGFloat(audioBarNum)
          let x:CGFloat = 20
          let y:CGFloat = 50
          let view:UIView = UIView(frame:
```

```
                                CGRect(x: w*CGFloat(i)+x, y: y,
                                width: w-x, height: h))
            self.view.addSubview(view)
1           gradientLayer = CAGradientLayer()
2           gradientLayer.frame = view.bounds;
3           gradientLayer.startPoint = CGPoint(x:0,y:0);
4           gradientLayer.endPoint = CGPoint(x:0,y:1);
5           view.layer.addSublayer(gradientLayer)
6           layerArray.addObject(gradientLayer)
        }
```

代码第 1 行实例化一个 CAGradientLayer 实例对象，第 2 行到第 4 行设置 gradientLayer 图层 frame 位置属性，以及动画的运动方向。startPoint 和 endPoint 的设置可以保证动画如图 10.15 的运动效果。第 5 行为每个单独的柱状图的 Layer 图层添加一个 gradientLayer 实例对象，最后一行将所有的 gradientLayer 实例对象保存在 layerArray 数组中。

完成这一步之后动画也是无法启动的。想让动画模拟音量的跳动，而音量每时每刻又是在不停变化的，所以这里采用定时器的方式，模拟音量的随机变化。每次定时时间到当前柱状音量图就会随机展示图 10.15 的动画效果。具体实现代码如下。

```
    for i in 0...audioBarNum {
    柱状图添加代码（省略）
        }
 1       Timer.scheduledTimer(timeInterval: 0.4,
                target: self,
                selector: #selector(ViewController.colorChange),
                userInfo: nil,
                repeats: true)
    func colorChange(){
    2       for layer in layerArray{
    3          let index:Int = Int(arc4random_uniform(11))
    4           let color:UIColor = colorArray.object(at:index) as!
UIColor
```

```
5           let colors = [
              UIColor.clear.cgColor,
              color.cgColor
            ]
6           let layer = layer as! CAGradientLayer
7           layer.colors = colors
8           layer.locations = [0,1.0]
9           let gradientAnimation:CABasicAnimation =
                                              CABasicAnimation()
10          gradientAnimation.keyPath = "locations"
11          let beginValue = Float(arc4random_uniform(11))/10.0
12          gradientAnimation.fromValue = [beginValue,beginValue]
13          gradientAnimation.toValue = [1.0,1.0]
14          gradientAnimation.duration = 0.4
15          layer.add(gradientAnimation, forKey: nil)
        }
    }
```

该动画模拟音量每 0.4s 变化一次，所以代码第 1 行设置一个每 0.4s 执行一次的定时器，定时时间到则执行一次模拟音量变化动画效果。音量变化动画效果在方法 colorChange()中实现。代码第 2 行遍历我们为每根柱状图添加的 gradientLayer 图层。第 3 行和第 4 行从已经准备好的随机颜色数组中选取一组随机颜色。第 5 行到第 8 行设计 gradientLayer 图层颜色渐变效果中颜色和位置属性。第 9 行定义一个 CABasicAnimation 动画实例对象。第 10 行设置动画是对哪种动画属性起作用。这里是对颜色梯度变换中的 locations 属性起作用。第 11 行到第 13 行设置 locations 属性的变化范围。代码最后两行设置动画变化周期并将动画添加到 Layer 图层。

## 10.5 本章小结

CAGradientLayer 是 QuartzCore 框架下非常重要的一个类，该类在颜色的梯度变换中使用非常广泛。比如在本章实现的指纹光波扫描效果和音量跳动效果，

都是通过该类实现的。CAGradientLayer 掌握起来不是很容易，尤其是对光波的执行方向、颜色梯度、“彗星拖尾”效果的理解，但本章对其做了详细的原理和效果分析。大家可以结合第 10.2 节的知识打造一个个性化的颜色梯度效果，比如 iPhone 手机解锁中最常见的广播扫描解锁效果，如图 10.16 所示。


图 10.16　手机解锁光波动画效果

# 第 11 章　CoreAnimation: CAShapeLayer 打造“动态”图表效果

## 本章内容

- 掌握贝济埃曲线的基本原理
- 实现图表动态渲染

虽然手机屏幕很小，但经过合理的规划并借助图表也可以展示很多内容，比如彩票选号走势 APP，或者是长江证券等金融类 APP。这些应用都有着良好的图表展示效果，比如描述彩票走势的折线图、柱状图，如图 11.1 所示。本章将为大家介绍如何使用 QuartzCore 框架下的 CAShapeLayer 类让图表动起来。

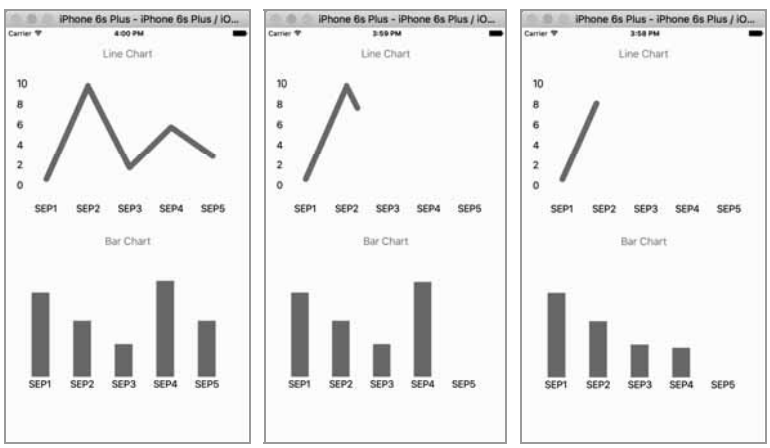



图 11.1　图表动态加载效果

# 11.1　CAShapeLayer 追本溯源

CAShapeLayer 是 QuartzCore 框架下非常重要的一个类，利用它可以实现各种图形绘制类动画效果。CAShapeLayer 来自 iOS 框架下的核心动画部分，Shape 为形状的意思，描述了当前动画的特点，可以实现各类形状的绘制。Layer 表明当前动画并非直接作用于 UIView 显示层上，而是作用在 Layer 内容层上。到这里相信大家大致搞清楚了 CAShapeLayer 类的基本含义，接下来再思考如何实现这一动画效果。

图 11.1 演示了动画的渐变效果，经过仔细观察可以得到以下结论。

（1）折线图：描述折线绘制动画。

（2）柱状图：描述每根柱子从低到高的渐变过程。

折线图可以理解为一根完整的线段，柱状图可以理解为由多条线段组成。这里无论是对于折线图还是柱状图都采用贝济埃曲线来绘制，所以有必要先了解一下贝济埃曲线。

# 11.2　贝济埃曲线

## 11.2.1　初识贝济埃曲线

首先在图上确定四个点 P0、P1、P2、P3，其中 P0 为曲线绘制起点，P3 为曲线绘制终点，在 P0 处设置一条曲线，保证其曲线的切线刚好可以通过 P1 点，同时在 P3 处设置一条曲线，保证其曲线的切线刚好可以通过 P2 点。最后再把曲线连接起来就形成了如图 11.2 所示的曲线效果。

贝济埃曲线就是这样的一条曲线，它是依据四个位置任意点的坐标绘制出的一条光滑曲线。我们按照已知曲线参数方程来确定四个点的思路就可以设计出这种矢量曲线绘制法。

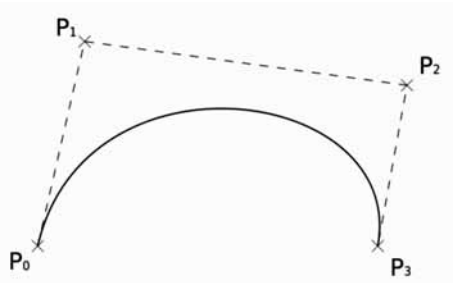



图 11.2　贝济埃曲线

在 iOS 中我们需要关注什么呢？iOS 中有一个 UIBezierPath 类，使用 UIBezierPath 类可以创建基于矢量的路径。这个类位于 UIKit 框架下，因此可以非常方便地在工程中集成。通过贝济埃曲线可以方便地绘制点、线段、弧形、圆形和曲线。那么如何在 iOS 中使用贝济埃曲线呢？

## 11.2.2　贝济埃曲线在 iOS 中的应用

先来总结一下贝济埃曲线的使用步骤。

（1）创建一个贝济埃曲线实例对象。

（2）设置曲线各种属性：如线条颜色、线条粗细。

（3）设置曲线的起始点。

（4）开始绘制贝济埃曲线。

下面就结合具体代码将贝济埃曲线的绘制步骤“代码化”。贝济埃曲线使用第一步，创建一个贝济埃实例对象。

```
let line:UIBezierPath = UIBezierPath()
```

### 1．贝济埃曲线：线条粗细

第二步设置贝济埃曲线的各种属性，我们先来看看贝济埃曲线都有哪些属性。最重要的属性无外乎线条颜色、线条粗细程度和线条拐角。将 lineWidth 属性设置为 1.0 和 10.0 效果如图 11.3 所示。

```
bezierPath.lineWidth = 1.0    bezierPath.lineWidth = 10.0
```

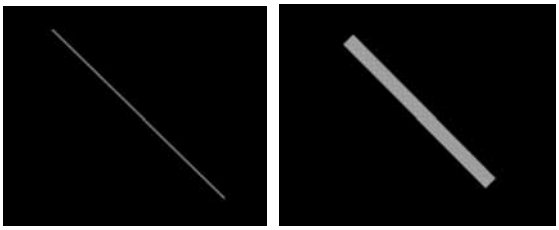


图 11.3　贝济埃曲线不同线条宽度绘制效果

### 2．贝济埃曲线：线条拐角

图 11.3 右图中曲线头尾都是方形，比较尖锐。还可以将其变成圆弧形状。修改 lineCapStyle，将其风格设置为 Round，如图 11.4 所示为曲线不同端点风格。

```
bezierPath.lineCapStyle = CGLineCap.Square
bezierPath.lineCapStyle = CGLineCap.Round
```

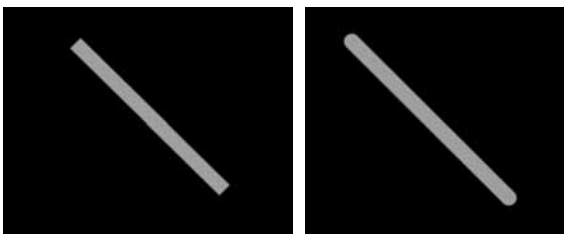


图 11.4　曲线不同端点风格

一条线段通过修改 lineCapStyle 就可以改变线段的头尾形状，如果是两条线段，如何处理两条线段连接处的拐角呢？先来看看默认情况下的拐角形状。如图 11.5 所示左图为默认情况下两条线段的拐角。

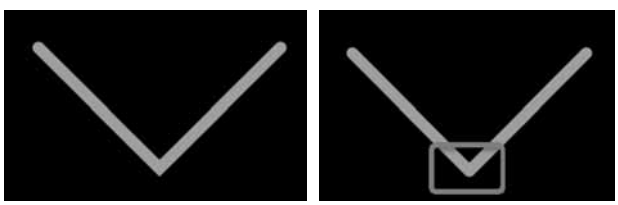


图 11.5　线段连接处拐角

将拐角处的属性设置为：

```
bezierPath.lineJoinStyle = CGLineJoin.Round
```

拐角处的形状如图 11.5 右图所示。

### 3．贝济埃曲线：线条颜色

默认情况下，线条颜色设置为橙色，如果想修改线条颜色如何做呢？

```
    let PNGreen:UIColor = UIColor(red: 77.0/255.0, green: 186.0/255.0,
blue: 122.0/255.0, alpha: 1.0)
    PNGreen.set()
```

使用 RGB 颜色控件定义一个自定义颜色（青色），并调用 set 方法为当前线条着色，经过着色后的线段如图 11.6 所示。

图 11.6　修改线条着色

这里调用 set 方法设置线条颜色，其实这里的颜色还分为边界颜色和填充颜色，要想弄清楚这个问题，需要先理解如何绘制曲线。

### 4．贝济埃曲线：绘制

想要成功地绘制贝济埃曲线，首先需要确定绘制的起点。

```
    bezierPath.move(to:CGPoint(x:150,y:150))
```

从（150，150）处开始绘制，UIBezierPath 类提供了大量的方法方便我们绘制各种形状，下面列举了 UIBezierPath 类常用的绘制方法。

```
    open func move(to point: CGPoint)
    open func addLine(to point: CGPoint)
    open func addCurve(to endPoint: CGPoint,
    controlPoint1: CGPoint, controlPoint2: CGPoint)
    open func addQuadCurve(to endPoint: CGPoint,
    controlPoint: CGPoint)

    open func addArc(withCenter center: CGPoint,
```

```
    radius: CGFloat,
    startAngle: CGFloat,
    endAngle: CGFloat,
    clockwise: Bool)
```

如果想绘制一条直线，那么可以使用下面的代码：

```
        bezierPath.move(CGPoint(x:150,y:150))
        bezierPath.addLine(CGPoint(x:250,y:250))
```

代码第 1 行的作用是确定绘制的起点坐标（150，150），代码第 2 行添加一条直线，直线的起点即（150，150），终点即（250，250），这时会有如图 11.7 所示的效果。如果想绘制图 11.6 右图的效果，可以在代码原有的基础上添加一段代码：

```
        bezierPath.addLine(CGPoint(x:350,y:150))
```

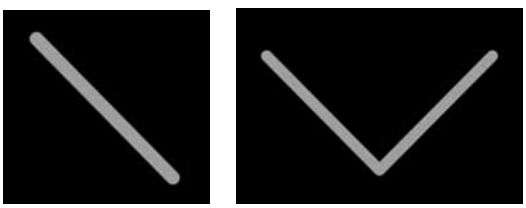


图 11.7　绘制直线和折线

这段代码的作用是在图 11.6 左图的基础上继续添加一条线段，线段的起点为（250，250），终点为（350，150）。如果这时想绘制闭合曲线如何做呢？可以在原有基础上再追加一行代码，如图 11.8 所示。

```
    bezierPath.close()
```

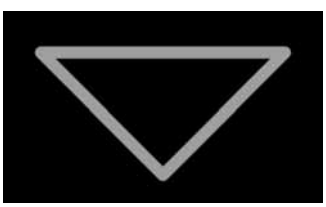


图 11.8　闭合曲线

这里有一个重点知识需要给大家交代清楚，即如何为边框和内部着色。先

来展示一段完整的绘制代码：

```
1       let  color1:UIColor = UIColor(red: 255.0 / 255.0,
            green: 127.0 / 255.0, blue: 79.0 / 255.0, alpha: 1.0)
2       let  color2:UIColor = UIColor(red: 77.0/255.0,
            green: 186.0/255.0, blue: 122.0/255.0, alpha: 1.0)
3       let bezierPath = UIBezierPath()
4       bezierPath.lineWidth = 10.0
5       bezierPath.lineCapStyle = CGLineCap.Round
6       bezierPath.lineJoinStyle = CGLineJoin.Round
7       bezierPath.move(CGPoint(x:150,y:150))
8       bezierPath.addLine(CGPoint(x:250,y:250))
9       bezierPath.addLine(CGPoint(x:350,y:150))
10      bezierPath.close()
11      color1.setStroke()
12      color2.setFill()
13      bezierPath.fill()
14      bezierPath.stroke()
```

代码第 1 行和第 2 行设置两种颜色，一种边框颜色为橙色，一种内部颜色为绿色。代码第 3 行实例化 UIBezierPath 对象。第 4 行到第 6 行设置当前线段的线条宽度、端点的类型、线段拐点类型。第 7 行到第 9 行确定贝济埃曲线的绘制起点，添加了两条绘制线段。第 10 行闭合当前曲线。第 11 行和第 12 行设置当前线段的边框颜色和内部填充颜色，最后两行开始填充边框颜色和内部颜色。代码执行效果如图 11.9 所示。

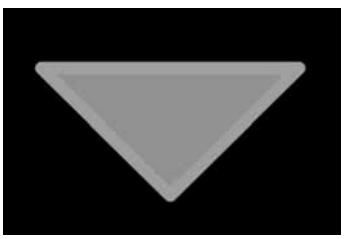


图 11.9　边框及内部填充效果

通过颜色 set 方法也可以同时设置边框颜色和内部颜色。

# 11.3　绘制动态图表

## 11.3.1　动态折线动画

下面将为大家介绍动态折线图表的代码实现。程序整体上分为两部分，第一部分图表初始化，用于初始化图表使用的 CABasicAnimation 动画实例以及 CAShapeLayer 图层实例，第二部分使用 strokeEnd 动画属性开始绘制折线动画效果。下面是动态折线图表实现的具体代码。

```
1 let PNGreen:UIColor = UIColor(red: 77.0/255.0,
        green: 186.0/255.0, blue: 122.0/255.0, alpha: 1.0)
2 class LineChartView: UIView {
3    var chartLine:CAShapeLayer = CAShapeLayer()
4    var pathAnimation:CABasicAnimation = CABasicAnimation()
5    override init(frame: CGRect) {
6        super.init(frame: frame)
7        self.backgroundColor = UIColor.white
8        self.clipsToBounds = true
9        chartLine.lineCap = kCALineCapRound
10       chartLine.lineJoin = kCALineJoinRound
11       chartLine.fillColor = UIColor.whiteColor().CGColor
12       chartLine.lineWidth = 10.0
13       chartLine.strokeEnd = 0.0
14       self.layer.addSublayer(chartLine)
}
}
```

代码第 1 行实例化一个青绿色的 UIColor 实例颜色。第 3 行实例化 CAShapeLayer 图层实例对象。第 4 行实例化 CABasicAnimation 基础动画实例对象。第 5 行初始化当前的 UIView 对象。第 7 行将整个 UIView 的背景颜色设置为白色。第 8 行将 view 超出部分裁剪属性设置为 true。第 9 行和第 10 行设置线条的拐角及连接处的线条类型。第 11 行设置折线的线条填充颜色。第 12 行设置线条的粗细为 10.0。第 13 行设置 stokeEnd 属性为 0.0，表明该动画效果开始绘制折线图时是一点点绘制，而非一次绘制完成。第 14 行将当前的

CAShapeLayer 图层添加到 self.layer 图层上。以上是初始化代码的主要内容，可以看出初始化代码主要完成 CAShapeLayer 图层的实例化以及各种初始化属性设置。下面是对外开放代码以及折线图动态绘制代码部分。

```
    func drawLineChart(){
        chartLine.strokeEnd = 1.0
        chartLine.add(pathAnimation, forKey: nil)
    }
```

该段代码的主要功能是为了设置折线图的最终状态。设置 stockeEnd 为 1，并将动画添加到 CAShapeLayer 图层上。

```
override func draw(_ rect: CGRect) {
1       let line:UIBezierPath = UIBezierPath()
2       line.lineWidth = 10.0
3       line.lineCapStyle = CGLineCap.Round
4       line.lineJoinStyle = CGLineJoin.Round
5       line.move(CGPoint(x:70,y:260))
6       line.addLine(CGPoint(x:140,y:100))
7       line.addLine(CGPoint(x:210,y:240))
8       line.addLine(CGPoint(x:280,y:170))
9       line.addLine(CGPoint(x:350,y:220))
10      chartLine.path = line.cgPath
11      chartLine.strokeColor = PNGreen.cgColor
12      pathAnimation.keyPath = "strokeEnd"
13      pathAnimation.timingFunction = CAMediaTimingFunction(
                              name: kCAMediaTimingFunctionLinear)
14      pathAnimation.fromValue = 0.0
15      pathAnimation.toValue = 1.0
16      pathAnimation.autoreverses = false
17      pathAnimation.duration = 2.0
    }
```

代码第 1 行实例化 UIBezierPath 实例对象，贝济埃曲线在第 11.2 节已经为大家详细介绍过，这里就直接使用贝济埃曲线绘制折线图。代码第 2 行到第 4 行设置贝济埃曲线的线条宽度、线条的拐角和连接处的圆角效果。代码第 5 行设置线段的起始位置，第 6 行到第 9 行依次添加各点的折线效果。第 10 行将贝

济埃曲线添加到 CAShapeLayer 图层的 path 属性上。第 11 行设置折线颜色。第 12 行设置动画的属性为 stokeEnd，表明该折线动画效果为一点点绘制，而非一次出现。第 13 行设置折线绘制速度为匀速绘制。第 14 行和第 15 行设置动画绘制前后的属性起始及终止状态。第 16 行设置动画相反执行效果为 false。最后设置动画周期为 2s。

### 11.3.2　动态柱状图动画

柱状图的代码结构与折线图类似，整体也分为两部分：一部分动态图表初始化，另一部分绘制代码部分。

```
class BarChartView: UIView {
    var chartLine:CAShapeLayer = CAShapeLayer()
    var pathAnimation:CABasicAnimation = CABasicAnimation()
    override init(frame: CGRect) {
        super.init(frame: frame)
        self.backgroundColor = UIColor.white
        self.clipsToBounds = true
        chartLine.lineCap = kCALineCapSquare
        chartLine.lineJoin = kCALineJoinRound
        chartLine.fillColor = UIColor.gray.cgColor
        chartLine.lineWidth = 30.0
        chartLine.strokeEnd = 0.0
        self.layer.addSublayer(chartLine)
}
}
```

动态柱状图初始化代码部分与折线图非常类似，这里就不再重复，它们的核心思想都是通过 CAShapeLayer 图层和 CABasicAnimation 基础动画来实现的。

下面是柱状图绘制代码。代码第 1 行定义一个方法，该方法设置柱状图绘制的 strokeEnd 属性为 1，并将动画添加到 chartLine 上。代码第 5 行定义一条贝济埃曲线。第 6 行到第 8 行设置曲线的宽度，以及拐角和曲线之间的连接属性。第 9 行到第 13 行在视图中合适的位置绘制五根柱状图。第 14 行将绘制的五根柱状图路径添加到 CAShapeLayer 图层上。第 15 行设置柱状图颜色。第 16 行设

置动画属性为 strokeEnd。第 17 行到第 19 行设置动画起始终止阶段的动画速度效果以及动画的 value 值变化情况。第 20 行设置动画执行完毕之后不按照原路返回。最后一行设置动画周期为 1s。

```
1   func drawLineChart(){
2       chartLine.strokeEnd = 1.0
3       chartLine.add(pathAnimation, forKey: nil)
    }
4   override func draw(_ rect: CGRect) {
5       let line:UIBezierPath = UIBezierPath()
6       line.lineWidth = 30.0
7       line.lineCapStyle = CGLineCap.Square
8       line.lineJoinStyle = CGLineJoin.Round
9       for i in 0...4 {
10          let x:CGFloat = CGFloat(60+70*i)
11          let y:CGFloat = CGFloat(100+20*i)
12          line.move(CGPoint(x:x,y:215))
13          line.addLine(CGPoint(x:x,y:y))
        }
14      chartLine.path = line.cgPath
15      chartLine.strokeColor = PNGreen.cgColor
16      pathAnimation.keyPath = "strokeEnd"
17      pathAnimation.timingFunction = CAMediaTimingFunction(
                              name: kCAMediaTimingFunctionLinear)
18      pathAnimation.fromValue = 0.0
19      pathAnimation.toValue = 1.0
20      pathAnimation.autoreverses = false
21      pathAnimation.duration = 1.0
    }
```

动画调用非常简单，该代码主要分为两个部分：一部分初始化并绘制相应坐标系，另一部分开始调用图标绘制方法。初始化代码如下。

```
    var lineChartView1:LineChartView?
    var barChartView1:BarChartView?
    var bezierView1:BezierView?
    override func viewDidLoad() {
```

```
        super.viewDidLoad()
        lineChartView1 = LineChartView(frame: self.view.bounds)
        self.view.addSubview(lineChartView1!)
        barChartView1 = BarChartView(frame: CGRect(x: 0,
                                y: self.view.bounds.height/2.0,
                                width: self.view.bounds.width,
                                height: self.view.bounds.height))
        self.view.addSubview(barChartView1!)
        self.addDrawChartButton()
        self.addAxes()
    }
```

下面是坐标系绘制代码。

```
    func addAxes(){
        //     LineChart
        for i in 1...5{
            let xAxesTitle:String = "SEP"+"\(i)"
            let  xAxesLabel:UILabel  =  UILabel(frame:  CGRect(x:
50+(CGFloat(i)-1)*70,y: 300, width: 50, height: 20))
            xAxesLabel.text = xAxesTitle
            self.view.addSubview(xAxesLabel)
        }
        for i in 0...5{
            let yAxesTitle:String = "\(10-i*2)"
            let yAxesLabel:UILabel = UILabel(frame: CGRect(x: 20,y:
120+(CGFloat(i)-1)*35, width: 20, height: 20))
            yAxesLabel.text = yAxesTitle
            self.view.addSubview(yAxesLabel)
        }
        //     BarChart
        for i in 1...5{
            let xAxesTitle:String = "SEP"+"\(i)"
            let  xAxesLabel:UILabel  =  UILabel(frame:  CGRect(x:
40+(CGFloat(i)-1)*70,y: 600, width: 50, height: 20))
            xAxesLabel.text = xAxesTitle
            self.view.addSubview(xAxesLabel)
```

```
            }
        }
```

最后是动画调用代码。

```
        func addDrawChartButton(){
            let bt_line:UIButton = UIButton()
            bt_line.frame = CGRect(
                (x:self.view.frame.size.width-100)/2,y:20,width:100,
height:50)
            bt_line.setTitle("Line Chart",
                                    forState: UIControlState.Normal)
            bt_line.setTitleColor(PNGreen,
                                    forState: UIControlState.Normal)
            bt_line.addTarget(self, action: "drawChart",
                  forControlEvents: UIControlEvents.TouchUpInside)
            self.view.addSubview(bt_line)
        }
        func drawChart(){
            lineChartView1!.drawLineChart()
            barChartView1!.drawLineChart()
        }
```

如图 11.10 所示为最终效果图。

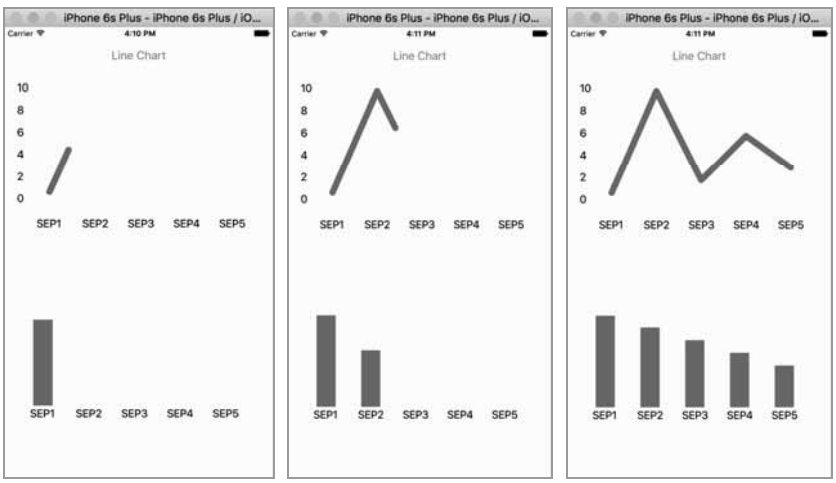



图 11.10　图标最终动画效果

## 11.4　本章小结

本章有两个核心要点，一个是贝济埃曲线的灵活使用，另一个是CAShapeLayer图层的使用，二者结合在一起可以形成各种各样炫酷的动画效果。

贝济埃曲线是 iOS 中非常重要的一类曲线，通过贝济埃曲线绘制直线、圆形、三角形、切线等各种自定义曲线非常方便。

CAShapeLayer 是 CALayer 下一个非常重要的子类，但是相比 CALayer 而言其使用起来更加灵活。尤其是结合各种自定义曲线时，会创造出各种各样的曲线类动画效果，例如在本文中为大家介绍的图表类动画。

# 第 12 章　CAReplicatorLayer：图层复制效果

## 本章内容

- 理解 CAReplicatorLayer 图层复制的含义
- 掌握恒星旋转等图层复制类动画效果设计方法

本章准备为大家介绍如图 12.1 所示的恒星旋转效果。首先来分析这个动画的特点，该动画由星空背景图片、屏幕中心位置恒星以及不停旋转的地球组成。关键点在于地球在旋转的过程中，每隔一段距离留下一帧相同的图片，而且每帧图片都按照一定的顺序不停地旋转，所以要想实现这个效果就需要使用 CAReplicatorLayer 图层，该图层主要用于图层的快速复制。图层复制之后结合关键帧 CAKeyframeAnimation 动画，将关键帧动画路径设置为圆形即可实现这种效果。

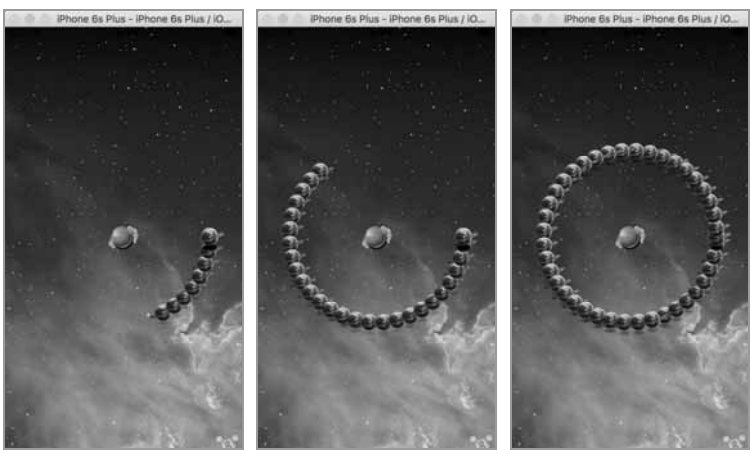


图 12.1　恒星公转效果

# 12.1　CAReplicatorLayer 追本溯源

CAReplicatorLayer 主要由三个部分组成：CA、Replicator、Layer。CA 为 CoreAnimation 的缩写，表明我们当前的动画使用的是 iOS 框架下的核心动画框架。Replicator 表示该图层可以用于图层的快速复制。Layer 表明当前动画并非直接作用于 UIView 显示层上，而是作用在 Layer 内容层上。本章就是使用 CAReplicatorLayer 图层的快速复制效果实现恒星公转效果和音符跳动效果。

# 12.2　恒星旋转动画实现

恒星旋转动画从整体结构上分为三个部分。

（1）UIImageView：星空背景图片、恒星、地球。

（2）CAKeyframeAnimation：关键帧动画及相关属性设置。

（3）CAReplicatorLayer：Replicator 图层关键属性设置。

第一部分实现代码如下所示。

```
let UISCREEN_WIDTH = UIScreen.mainScreen().bounds.size.width
let UISCREEN_HEIGHT = UIScreen.mainScreen().
                                          bounds.size.height
var replicatorLayer:CAReplicatorLayer = CAReplicatorLayer()
var iv_earth:UIImageView?
override func viewDidLoad() {
    super.viewDidLoad()
    let background:UIImageView = UIImageView(frame:
                          CGRect(x: 0, y: 0,
                           width: UISCREEN_WIDTH,
                          height: UISCREEN_HEIGHT))
    background.image = UIImage(named: "background.jpg")
    self.view.addSubview(background)
   iv_earth = UIImageView(frame:
```

```
                CGRect(x: (UISCREEN_WIDTH-50)/2+150,
                y: (UISCREEN_HEIGHT-50)/2, width: 50, height: 50))
11      iv_earth!.image = UIImage(named: "earth.png")
12      let iv_sun = UIImageView(frame: CGRect(x: 0, y: 0,
                                        width: 50, height: 50))
13      iv_sun.center = self.view.center
14      iv_sun.image = UIImage(named: "sun.png")
15      replicatorLayer.addSublayer(iv_earth!.layer)
16      replicatorLayer.addSublayer(iv_sun.layer)
    }
```

代码第 1 行和第 2 行定义当前屏幕宽高信息。第 3 行和第 4 行实例化 CAReplicatorLayer 图层，并定义地球 UIImageView。第 7 行定义 UIImageView 背景图片，第 8 行为 UIImageView 背景添加 background.jpg 图片。第 9 行将背景图片添加到 self.view 上。第 10 行实例化地球 UIImageView 并将其定位在屏幕的指定位置。第 11 行添加 earth.png 图片。第 12 行到第 14 行实例化恒星 UIImageView 实例对象，添加 sun.png 背景图片并添加到屏幕的中心位置。最后两行分别将地球 UIImageView 和恒星 UIImageView 添加到 Layer 图层上。

动画实现部分代码在 viewWillAppear 方法中实现，因此可以达到应用启动之后动画即开始展现的效果。下面是动画实现核心代码。

```
override func viewWillAppear(_ animated: Bool) {
1       let path:UIBezierPath = UIBezierPath()
2       path.addArc(withCenter:
         CGPoint(x: self.view.center.x, y: self.view.center.y),
         radius: 150,
         startAngle: 0,
         endAngle: CGFloat(M_PI*2),
         clockwise: true)
3       path.close()
4       let animation:CAKeyframeAnimation =
                    CAKeyframeAnimation(keyPath:"position")
5       animation.path = path.CGPath
6       animation.duration = 10
```

```
7       animation.repeatCount = MAXFLOAT
8       replicatorLayer.instanceCount = 100
9       replicatorLayer.instanceDelay = 0.2
10      self.view.layer.addSublayer(replicatorLayer)
11      iv_earth?.layer.add(animation, forKey: nil)
    }
```

代码第 1 行实例化一个贝济埃曲线的路径。第 2 行绘制一个圆，该圆的路径就是最终恒星公转时的路径。第 3 行关闭 path，保证贝济埃曲线是一个闭合的曲线。第 4 行实例化关键帧动画实例对象。第 5 行到第 7 行设置动画的路径属性、执行周期、重复次数。第 8 行表明有 100 个重复的 Layer 图层，即 100 个地球 UIImageView 的图层内容。第 9 行动画每 0.2s 复制一个 Layer 图层。第 10 行将 replicatorLayer 图层添加到 self.view.layer 图层上。最后一行在地球 Layer 图层上启动该动画。最终动画效果如图 12.1 所示。

## 12.3　音量跳动动画效果

在第 10 章为大家介绍了使用 CAShapeLayer 制作音量跳动动画效果的方法，本章将结合 CAReplicatorLayer 图层复制效果实现音量跳动动画效果。

下面是最终实现代码。该代码整体分为两个部分，第一部分实例化 CAReplicatorLayer 对象，并设置 Layer 复制图层的相关属性。第二部分使用 CABasicAnimation 实例对象，设置动画周期属性等，启动动画。下面是第一部分 CAReplicatorLayer 代码部分。

```
    override func viewDidLoad() {
1       super.viewDidLoad()
2       let replicatorLayer:CAReplicatorLayer =
                                        CAReplicatorLayer()
3       replicatorLayer.frame = CGRect(x: 0, y: 0, width: 414,
                                              height: 200)
4       replicatorLayer.instanceCount = 20;
5       replicatorLayer.instanceTransform =
```

```
                        CATransform3DMakeTranslation(20, 0, 0)
    replicatorLayer.instanceDelay = 0.2
    replicatorLayer.masksToBounds = true
    replicatorLayer.backgroundColor =
                                        UIColor.black.cgGColor;
}
```

代码第 2 行实例化 CAReplicatorLayer 对象。第 3 行设置该图层的生效范围，生效范围宽度为 414，高度为 200。第 4 行设置图层共复制多少份，这里一共复制 20 份图层，每一份表示一根音量跳动的音量柱。第 5 行设置复制图层之间的渐变效果，这里设置图层沿 *x* 方向，每隔 20 个点复制一份。第 6 行设置图层复制间隔为每 0.2s 一次。第 7 行将图层超出生效范围之外的部分剪切掉。最后一行设置图层背景颜色为黑色。可以看出第一部分代码主要是实现 CAReplicatorLayer 图层的实例化，并设置图层的各种相关属性。下面是第二部分代码。

```
override func viewDidLoad() {
    super.viewDidLoad()
    ◦ ◦ ◦
    let layer = CALayer()
    layer.frame = CGRect(x: 14, y: 200, width: 10, height: 100)
    layer.backgroundColor = UIColor.red.cgColor
    replicatorLayer.addSublayer(layer)
    self.view.layer.addSublayer(replicatorLayer)
    let animation:CABasicAnimation = CABasicAnimation()
    animation.keyPath = "position.y"
    animation.duration = 0.5
    animation.fromValue = 200
    animation.toValue = 180
    animation.autoreverses = true
    animation.repeatCount = MAXFLOAT
    layer.add(animation, forKey: nil)
}
```

代码第 1 行到第 3 行实例化一个 Layer 图层，每一个图层实例对象对应一

个音量跳动音量柱。该图层宽高分别为 10、100。第 4 行将当前图层添加到 replicatorLayer，第 5 行将 replicatorLayer 图层添加到 self.view 的图层上。第 6 行初始化 CABasicAnimation 动画实例。第 7 行设置动画属性为 position.y 属性。第 8 行设置动画周期为 0.5s。第 9 行和第 10 行设置该音量柱高度在 200～180 的范围内波动。第 11 行和第 12 行设置动画完成之后不返回且不断重复。最后一行将动画添加到当前图层上。最终动画效果如图 12.2 所示。

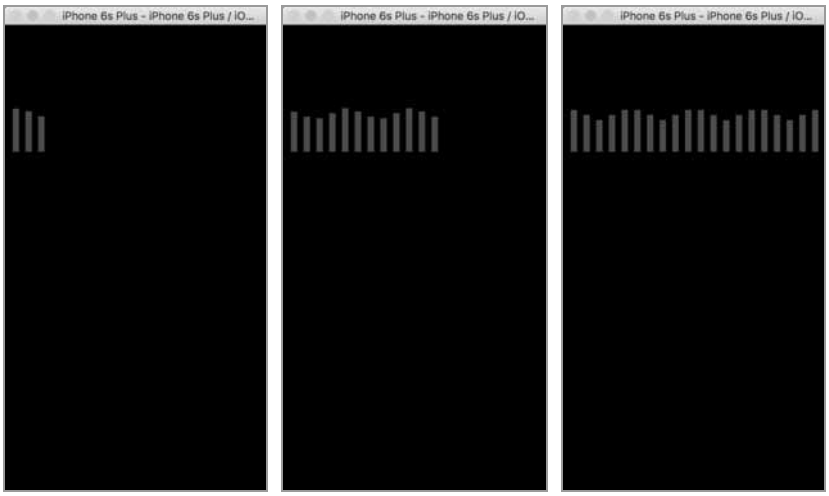



图 12.2　音量跳动动画效果

## 12.4　本章小结

本章通过恒星旋转动画和音符跳动动画详细阐释了 CAReplicatorLayer 图层的基本使用。CAReplicatorLayer 是 Layer 图层下的一个可以复制自己子层的 Layer，该子层的功能可以快速、大量地复制与原来 Layer 具有相同功能的子层。通过给子层添加相应的运动效果、形状效果可以实现各种各样有趣的动画。

# 第三卷

# 3D 动画

# 第 13 章　3D 动画初识

## 本章内容

- 理解 iOS 下 3D 变换的基本原理
- 理解矩阵变换的算法原理
- 掌握常用 3D 动画效果

在第一卷和第二卷中已经为大家详细介绍了 iOS 中常见的 2D 动画效果，本章将为大家介绍一些常用的 3D 动画效果。通过修改 *x*、*y* 轴可以实现动画 2D 效果，如移动、缩放等。3D 则是在原本的基础上修改 *z* 轴，实现投影、拉伸等 3D 效果。iOS 中提供了一个 3D 变幻矩阵（CATransform3D），通过这个矩阵可以实现各种透视、拉伸、移动缩放效果。

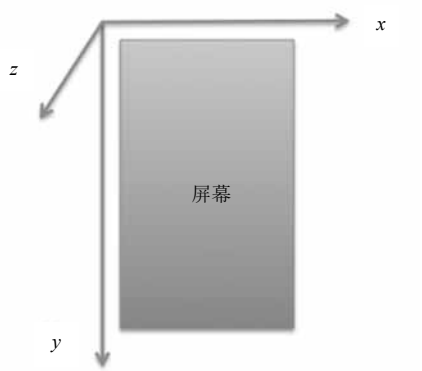



图 13.1　屏幕坐标系示意图

# 13.1 锚点的基本概念

要想实现一些复杂的 3D 动画效果，首先需要搞清楚锚点的概念。AnchorPoint 是一个 CGPoint 类型的值，该值指定了一个基于 bounds 的坐标系位置。也就是说锚点指定了 bounds 相对于 position 的值，同时也是变化时候的中心点。

一般情况下，锚点的默认值为（0.5，0.5），取值范围为 0～1 之间。如图 13.2 所示左边的图表明了锚点的默认位置。

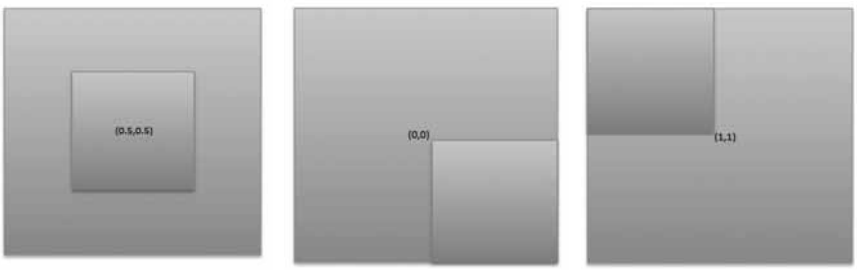



图 13.2 锚点位置

当锚点值为（0，0）时，那么在进行 2D、3D 变幻时都围绕着图层的左上角点进行变幻。当锚点值为（1，0）时，那么在进行 2D、3D 变幻时都围绕着图层的右上角点进行变幻。

在第 13.3 小节会举例演示锚点对于 2D、3D 动画效果的影响。

# 13.2 矩阵变换的基本原理

iOS 中利用 CATransform3D 实现 3D 变换效果。CATransform3D 其实质是定义了一个三维变换（4×4 CGFloat 值的矩阵），利用该矩阵可以实现图层的旋转、缩放、偏移、歪斜和视图透视等效果。

先来看看 CATransform3D 所描述的矩阵都有哪些功能。

```
struct CATransform3D
{
```

```
CGFloat m11（x 缩放）, m12（y 切变）, m13（）, m14（）;
CGFloat m21（x 切变）, m22（y 缩放）, m23（）, m24（）;
CGFloat m31（）     , m32（）     , m33（）, m34（透视）;
CGFloat m41（x 平移）, m42（y 平移）, m43（z 平移）, m44（）;
};
```

该矩阵是一个 4×4 的矩阵，矩阵中每一个值都会在变幻中起到一定的作用。那么如何来理解它呢？假设原来图层的位置是 $x$、$y$、$z$、1。同样定义为一个矩阵，不过这个矩阵的行列是一个 1×4 的矩阵（表示一行四列）。该矩阵与 CATransform3D 相乘最终会得到一个新的矩阵［new$x$，new$y$，new$z$，1］。如图 13.3 所示描述了最终的运算过程。

$$\begin{matrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\ \text{coordinate}\end{matrix} \times \begin{matrix}\begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m31 & m31 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix} \\ \text{transform}\end{matrix} = \begin{matrix}\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \\ \text{transformed coordinate}\end{matrix}$$

图 13.3　矩阵运算

下面还有一些 iOS 官方手册上提供的特殊矩阵对应的变幻效果。如图 13.4 所示。

**注意**：如果不太熟悉矩阵的相关概念，可以参考大学课程中的线性代数部分。

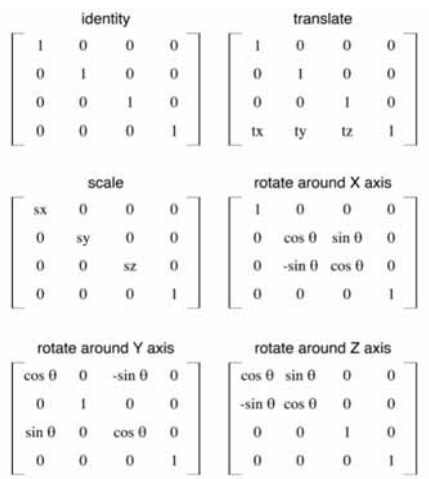



图 13.4　常见矩阵变换形式

# 13.3 3D 旋转效果

在了解了锚点和矩阵的基本知识之后，下面举一个例子来实现实现图像的旋转。我们先来思考一个问题，图层的 3D 旋转是什么样子呢？如图 13.5 所示，围绕图层水平中心位置旋转。

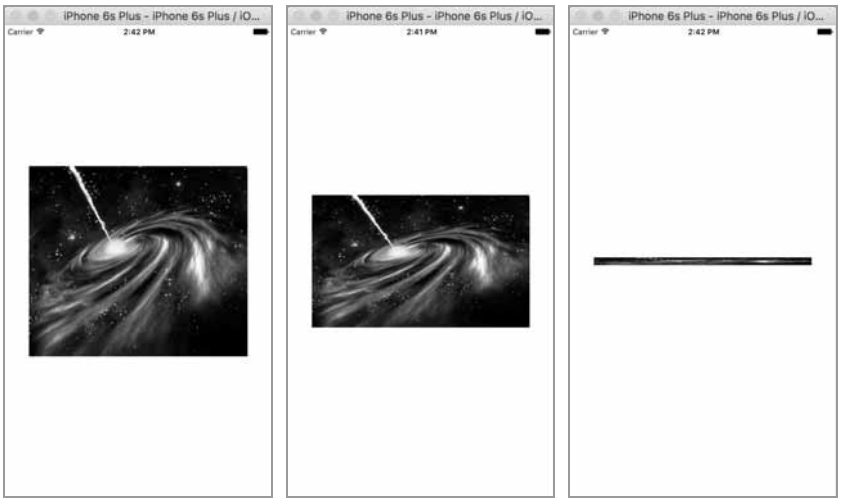



图 13.5 绕图层水平 *x* 中线旋转示意图

从图 13.5 中可以看出，该动画效果围绕图片 *x* 轴方向中线，实现在 *z* 轴方向旋转。下面是具体实现代码。

```
1  var imageView:UIImageView?
2   override func viewDidLoad() {
3       super.viewDidLoad()
4       imageView = UIImageView()
5       imageView?.frame = CGRect(x: 0, y: 0,
                                  width: 400, height: 300)
6       imageView?.center = self.view.center
7       imageView?.image = UIImage(named: "image.jpg")
8       imageView?.contentMode =
                          UIViewContentMode.ScaleAspectFit
9       imageView!.layer.anchorPoint = CGPoint(x: 0.5, y: 0.5)
10      self.view.addSubview(imageView!)
```

```
11      UIView.beginAnimations(nil, context: nil)
12      UIView.setAnimationDuration(4)
13      imageView!.layer.transform = CATransform3DMakeRotation(
                                    CGFloat(M_PI/2.0), 1, 0, 0);
14      UIView.commitAnimations()
    }
```

代码第 4 行实例化一个 UIImageView 实例对象，第 5 行和第 6 行设置该 imageview 的 frame 宽高信息以及 center。表明该 imageview 最终显示在屏幕的中间位置。第 7 行和第 8 行加载一张图片，并设置图片的最终显示方式。第 9 行设置当前锚点的位置为默认位置（0.5，0.5）。第 10 行将 imageview 实例添加到 self.view 上。第 11 行动画开始。第 12 行设置动画执行周期为 4s。第 13 调用动画 CATransform3DMakeRotation 实现旋转。最后一行提交动画代码，动画立即执行。下面是 CATransform3DMakeRotation 方法的相关参数。

```
public func CATransform3DMakeRotation(
                                          angle: CGFloat,
                                        _ x: CGFloat,
                                        _ y: CGFloat,
                                        _ z: CGFloat)
                                        -> CATransform3D
```

该方法实现动画的 3D 旋转。

（1）angle: CGFloat：表明动画的旋转角度。

（2）_ x: CGFloat：表明动画旋转时的旋转轴心。

（3）_ y: CGFloat：表明动画旋转时的旋转轴心。

（4）_ z: CGFloat：表明动画旋转时的旋转轴心。

通过修改第 10 行旋转的轴心位置，还可以实现各种有趣的效果。当代码修改为如下形式时：

```
imageView!.layer.transform = CATransform3DMakeRotation(CGFloat
(M_PI/2.0), 1, 1, 0);
```

表明该动画按照其对角线旋转，动画效果如图 13.6 所示。

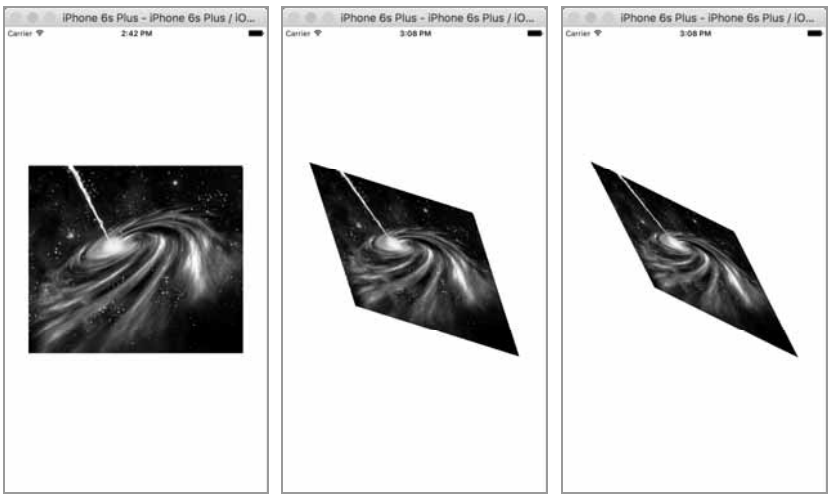


图 13.6　沿 *xy* 对角线旋转效果

修改锚点位置和旋转轴如下：

```
    imageView!.layer.anchorPoint = CGPoint(x:0,y:0)
    imageView!.layer.transform = CATransform3DMakeRotation(CGFloat
(M_PI/2.0), 0, 1, 0);
```

可以得到如图 13.7 所示的效果。

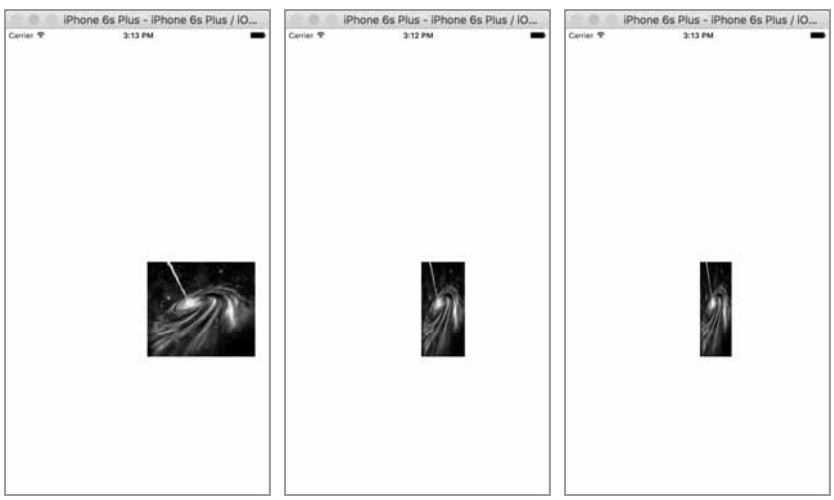


图 13.7　矩阵旋转效果

如果同时修改旋转轴：

```
imageView!.layer.anchorPoint = CGPoint(x:0.5,y:0.5)
imageView!.layer.transform =
        CATransform3DMakeRotation(CGFloat(M_PI/2.0), 1, 1, 1);
```

效果如图 13.8 所示。

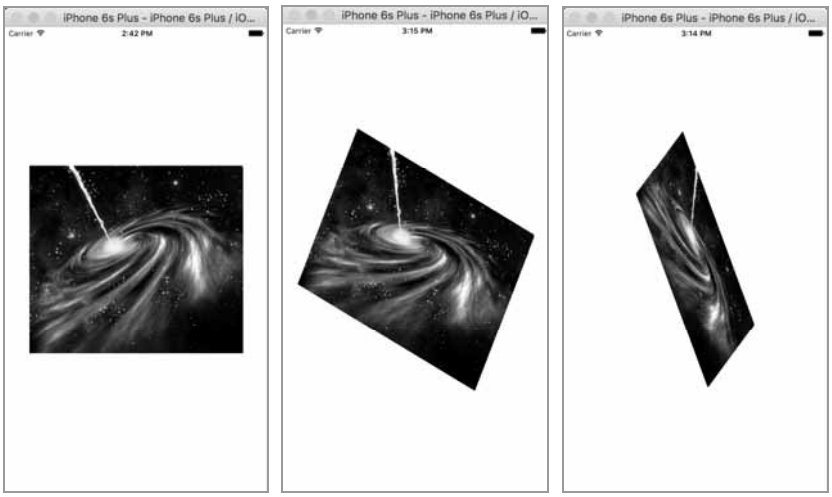



图 13.8　矩阵旋转效果

除了 CATransform3DMakeRotation 方法之外还有一些方法也比较常用，比如用于移动的 CATransform3DMakeTranslation 方法，用于缩放的 CATransform3DMakeScale 方法。这些方法使用起来都比较简单，不是直接作用在矩阵上而是将矩阵变幻做了二次封装，那么如何直接操作矩阵呢？

将动画部分代码修改为以下形式：

```
1       UIView.beginAnimations(nil, context: nil)
2       UIView.setAnimationDuration(4)
3       var transform:CATransform3D = CATransform3DIdentity
4       transform.m22 = 0.5
5       imageView!.layer.transform = CATransform3DScale(
                                                transform, 1, 1, 1)
6       UIView.commitAnimations()
```

代码第 3 行创建一个单位矩阵，该矩阵没有任何缩放、旋转等效果，为图

层最原始的形态。第 4 行手动修改矩阵的（2，2）位置处的值为 0.5。通过图 13.9 矩阵旋转效果可以看出，该值负责图层的 $y$ 轴方向缩放，将 m22 修改为 0.5 之后，表明当前动画为 $y$ 轴方向图片缩放。如图 13.9 所示为最终实现效果。

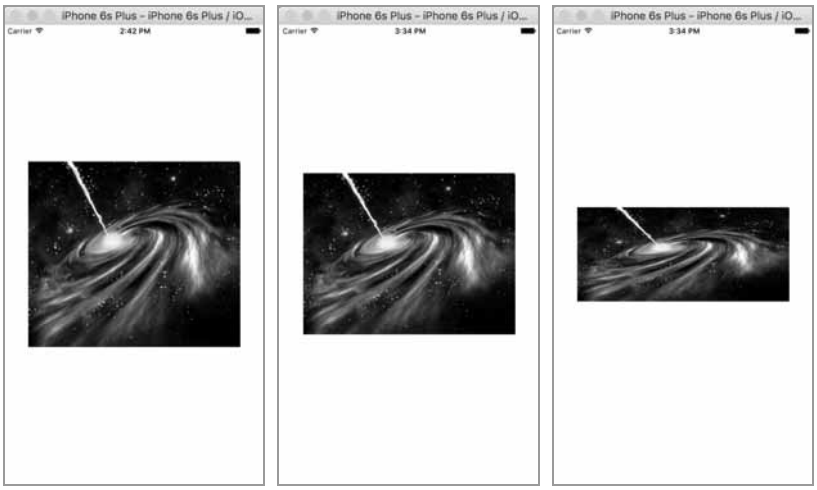


图 13.9　矩阵旋转效果

## 13.4　本章小结

本章作为介绍 3D 入门知识的第一章，首先为大家介绍了锚点和矩阵变幻的一些基本知识。并通过动画旋转效果详细介绍了使用 CATransform3D 进行 3D 旋转的操作流程。在介绍动画旋转效果的同时，通过调整不同的参数，可以实现各种各样有趣的 3D 效果。同时本章作为 3D 的入门部分，也为第 14 章实现 3D cover flow 效果打下了坚实基础。

# 第 14 章　Cover Flow 3D 效果

## 本章内容

- 掌握常用 3D 动画效果 Cover Flow 的设计方法

Cover Flow 是苹果公司首创的 3D 界面展现形式。Cover Flow 具有优美的展示界面和动感的 3D 展示效果，在图片展示、歌曲封面展示、影音展示等场景中应用广泛。在 iOS 中结合第 13 章的 3D 动画效果变换基础知识可以很方便地实现这个动画效果。

## 14.1　案例：Cover Flow 效果实现原理

在开始制作这个动画效果之前先来看看这个动画的实现原理。如图 14.1 所示为 Cover Flow 的最终实现效果。为了便于大家分析，这里以三张图片的 Cover Flow 动画效果为例。

先来看看最左边的一张图片。经过仔细观察可以发现该图片有以下几个特点。

（1）整个图片以 *y* 轴为轴，沿 *z* 轴方向按照一定的角度进行旋转。

（2）在旋转的过程中图片在 *x* 轴、*y* 轴、*z* 轴三个方向具有不同的拉伸比例。

（3）图片在 *z* 轴方向有一定的投影效果。

再来看看右边一张图片。右侧的图片和左侧图片类似，不同的是该图片的旋转角度和左侧图片相反。在第 13 章中已经为大家介绍了 3D 的基础知识，下

面就使用这些知识来实现以上三个技术点。


图 14.1　Cover Flow 效果

（1）整个图片以图片最左侧为轴，沿 *z* 轴方向按照一定的角度进行旋转。对于每一个 UIView 来说，默认情况下锚点为（0.5，0.5）。所以无论是沿着哪个轴旋转都会以图片中心为中心进行旋转。要想让图片以图片的最左侧为轴进行旋转可以将图片的锚点设置为 0。沿 *z* 轴方向旋转可以使用 CATransform3DRotate 方法，设置一定的角度即可实现。

（2）在旋转的过程中图片在 *x* 轴、*y* 轴、*z* 轴三个方向具有不同的拉伸比例。这里考察的是图片的缩放方法，调用 CATransform3DScale 方法，可以在 *x*、*y*、*z* 三个方向上设置不同的图片缩放比例，形成不同的拉伸效果。

（3）图片在 *z* 轴方向有一定的投影效果。CATransform3D 矩阵中，每一个值都有其特殊的含义，要想实现 *z* 轴方向上的投影效果，可以修改 m34 的内容。

## 14.2　案例：Cover Flow 效果代码实现

代码整体分为两部分，一部分是视图初始化，另一部分是 Cover Flow 效果

实现。下面是第一部分视图初始化部分代码。

```
1  var imageViewArray:NSMutableArray?
2  let imageView1:UIImageView = UIImageView(
             frame:CGRect(x: 100, y: 100, width: 200, height: 250))
3  let imageView2:UIImageView = UIImageView(
             frame:CGRect(x: 100, y: 100, width: 200, height: 250))
4  let imageView3:UIImageView = UIImageView(
             frame:CGRect(x: 100, y: 150, width: 300, height: 200))
5 override func viewDidLoad() {
6    super.viewDidLoad()
7    view.backgroundColor = UIColor.black
8    imageViewArray = [imageView1,imageView2,imageView3]
9    for i in 0...(imageViewArray?.count)!-1{
10       let imageView:UIImageView = imageViewArray?.
                                object(at: i) as! UIImageView
11       let imageName:String = "\(i+1).jpg"
12       imageView.image = UIImage(named: imageName)
13       imageView.layer.anchorPoint.y = 0.0
14       view.addSubview(imageView)
    }
  }
```

代码第 1 行到第 4 行实例化三组 UIImageView 实例对象，并将三张图片依次放在 self.view 上的指定位置。因为三张图片的位置相同，所以三张图片只能看到最上面一张，另外两张被顶层图片覆盖掉。代码第 7 行设置当前 view 黑色背景。第 8 行将图片存放在数组 imageViewArray 中。第 9 行依次遍历当前所有图片。第 10 行获取数组中的 UIImageView 实例对象。第 11 行和第 12 行按照一定的顺序将图片加载进去。第 13 行设置当前锚点 *y* 值为 0，表明图片可以沿着图片的边缘旋转而非围绕中心旋转。最后一行将 UIImageView 实例对象添加到当前图层上。

视图初始化完之后接下来就要实现 Cover Flow 动画代码。Cover Flow 这里也分为两部分，一部分用于初始化动画 3D 效果，另一部分使用 CAAnimationGroup 将 3D 动画和位置动画组合在一起，形成 Cover Flow 动态效

果。下面是具体实现代码。

```
override func viewWillAppear(_ animated: Bool) {
1       for i in 0...(imageViewArray?.count)!-1{
2           var imageTransform = CATransform3DIdentity
3           imageTransform.m34 = -0.005;
4           imageTransform = CATransform3DTranslate(
                              imageTransform, 0.0, 50.0, 0.0)
5           imageTransform = CATransform3DScale(
                              imageTransform, 0.95, 0.6, 1.0)
6           if i==0 {
7               imageTransform = CATransform3DRotate(
                imageTransform, CGFloat(M_PI_4/2), 0.0, 1.0, 0.0)
8           }else if i==1{
9               imageTransform = CATransform3DRotate(
                imageTransform, CGFloat(-M_PI_4/2), 0.0, 1.0, 0.0)
            }
10          let imageView:UIImageView = imageViewArray?.
                              object(at:i) as! UIImageView
11          let animation = CABasicAnimation(
                                          keyPath: "transform")
12          animation.fromValue = NSValue(
                        caTransform3D: imageView.layer.transform)
13          animation.toValue = NSValue(
                                  caTransform3D: imageTransform)
14          animation.duration = 0.5
```

代码第 1 行遍历图层中的三个 UIImageView 实例对象。第 2 行创建一个单位矩阵 imageTransform。第 3 行设置矩阵变换中投影系数为-0.005。第 4 行将矩阵进行位置变换。第 5 行将矩阵进行缩放变换。同时控制图层 *x*、*y*、*z* 三个方向上不同的缩放系数。第 6 行和第 7 行实现将顶层图层沿着逆时针旋转。第 8 行和第 9 行实现第二张图层顺时针旋转。第 10 行获取当前需要处理的 UIImageView 实例对象。第 11 行定义一个基础动画类型 CABasicAnimation。设置动画变换属性为 transform。第 12 行和第 13 行设置动画从原始状态变换到第 1 行到第 9 行处理的 transform 过程。最后一行设置动画的变换周期为 0.5s。

```
override func viewWillAppear(_ animated: Bool) {
1    let animBounds:CABasicAnimation = CABasicAnimation()
2      animBounds.keyPath = "position"
3      if i==0 {
4          animBounds.toValue = NSValue(
                                  CGPoint:CGPoint(x:100,y:10))
5      }else if i==1{
6          animBounds.toValue = NSValue(
                                  CGPoint:CGPoint(x:300,y:10))
7      }else{
8          animBounds.toValue = NSValue(
                                  CGPoint:CGPoint(x:200,y:20))
9      }
10     let animGroup:CAAnimationGroup = CAAnimationGroup()
11     animGroup.duration = 0.5
12     animGroup.repeatCount = 1
13     animGroup.removedOnCompletion = false
14     animGroup.fillMode=kCAFillModeForwards
15     animGroup.timingFunction = CAMediaTimingFunction(
                      name:kCAMediaTimingFunctionEaseInEaseOut)
16     animGroup.animations = [animation,animBounds]
17     imageView.layer.add(animGroup, forKey: "\(i)")
}
```

代码第 1 行和第 2 行实例化 CABasicAnimation 实例对象，并设置动画属性为 position。第 3 行到第 8 行判断当前图片如果为顶层图片，那么移动到新的位置（100，10）处。如果为第二张图片那么移动到（300，10）处。如果为最后一张图片，那么停留在（200，20）处。第 10 行实例化 CAAnimationGroup 实例对象。第 11 和第 12 行设置动画周期为 0.5s。动画重复次数为 1 次。第 13 行和第 14 行设置动画最后的保持状态。第 15 行设置动画开始和结束时的加速状态。第 16 行将以上的 transform 和 position 动画添加到 CAAnimationGroup 实例对象中。最后一行将动画添加到各自 UIImageView 实例对象上。如图 14.2 所示为最终实现效果

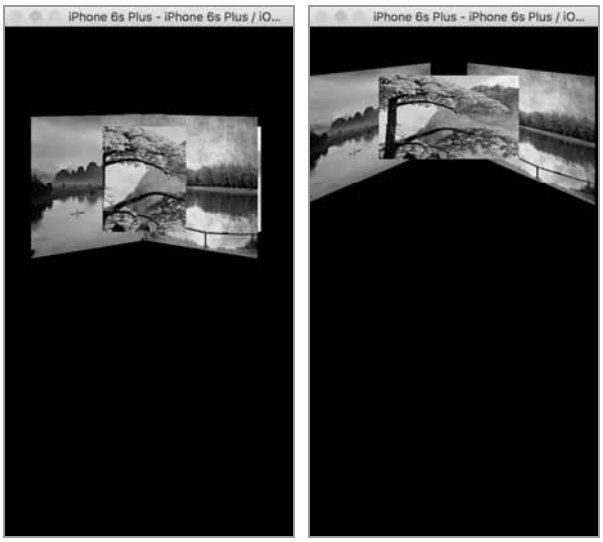



图 14.2　Cover Flow 最终实现效果

## 14.3　本章小结

结合第 13 章的矩阵相关知识，本章将 UIImageView 的 3D 变换应用在 Cover Flow 效果上。经过分解 Cover Flow 动画效果可以看出，该动画其实质相当于只对每张图片进行了 3D 变换，通过每张图片的 3D 变换最终组合成绚丽的 Cover Flow 效果。其实无论该动画如何绚丽其基础仍然是第 13 章的矩阵变换相关知识，所以在掌握矩阵变换基础的知识上再来看这一章的知识会起到事半功倍的效果。

# 第四卷

# 转场动画

# 第 15 章　CoreAnimation: CATransition 转场动画



本章内容

- 理解 CATransition 的使用方法
- 掌握常用转场动画使用场景

转场动画主要用于不同视图场景之间的切换。比如我们经常使用的 PPT，每一页 PPT 都可以作为一个独立的场景，在这个单独的场景中可以添加各种各样的 UI。但是当这一页展示完毕，需要进入到下一页时，添加一个合适的过度动画会使得转场效果比较平滑。在 iOS 中经常使用 CoreAnimation 核心动画中的 CATransition 实现这个功能。

## 15.1　CATransition 初识

CATransition 同 CoreAnimation 核心动画中 CABasicAnimation 等相关类的使用方法类似。主要分为以下三个步骤。

（1）实例化 CATransition，并设置相应的转场动画 key。

（2）设置合适的转场动画属性，比如动画周期、过度方向、动画保持状态等。

（3）将动画效果添加到相应视图的 Layer 图层中。

在第一步设置动画效果时需要注意，iOS 提供了大量的炫酷动画效果。不过总体上来说可以分为公有 API 和私有 API，公有 API 制作的 APP 可以直接上线，私有 API 制作的 APP 有被拒的风险，所以在使用的时候需要尤为注意。

公有 API：

（1）fade，淡入淡出效果，可以使用常量 kCATransitionFade 表示。

（2）push，推送效果，可以使用常量 kCATransitionMoveIn 表示。

（3）reveal，揭开效果，可以使用常量 kCATransitionReveal 表示。

（4）movein，移动效果，可以使用常量 kCATransitionMoveIn 表示。

私有 API：

（1）pageCurl，向上翻页效果，只能用字符串表示。

（2）pageUnCurl，向下翻页效果，只能用字符串表示。

（3）cube，立方体翻转效果，只能用字符串表示。

（4）oglFlip，翻转效果，只能用字符串表示。

（5）stuckEffect，收缩效果，只能用字符串表示。

（6）rippleEffect，水滴波纹效果，只能用字符串表示。

（7）cameraIrisHollowOpen，相机打开效果，只能用字符串表示。

（8）cameraIrisHollowClose，相机关闭效果，只能用字符串表示。

在第二个步骤中，设置动画的周期、最终状态等属性和 CoreAnimation 核心

动画中的 CABasicAnimation 类使用方法相同，这里主要介绍转场动画的方向属性设置。转场动画支持以下四种方向。

（1）kCATransitionFromRight：从右侧转场。

（2）kCATransitionFromLeft：从左侧转场。

（3）kCATransitionFromTop：从顶部转场。

（4）kCATransitionFromBottom：从底部转场。

第三步实现将动画添加到指定的图层上。如果想让整个视图控制器进行转场，那么可以添加到当前的 self.view 上。如果想对某个特定的图层进行转场，那么可以直接添加到相应图层上。

## 15.2 案例：基于 CATransition 的图片查看器

该图片查看器的主要功能是每当点击下一张图片的时候，利用 CATransition 转场动画展示下一张图片。下面是具体实现代码。

```
1   var imageView:UIImageView?
2   override func viewDidLoad() {
3       super.viewDidLoad()
4       imageView = UIImageView()
5       imageView?.frame = CGRect(x:0,y:0,width:300,height:400)
6       imageView?.center = self.view.center
7       imageView?.image = UIImage(named: "1.jpg")
8       imageView?.contentMode = UIViewContentMode.ScaleAspectFit
9       self.view.addSubview(imageView!)
    }
```

代码第 1 行定义一个 UIImageView 实例对象，第 4 行完成这个实例对象的实例化。第 5 行和第 6 行定义 UIImageView 的 *x*、*y*、width、height 信息，并将 UIImageView 最终确定在当前 self.view 的中心位置上。第 7 行加载首张图片。第 8 行设置当前图片的内容拉伸方式。第 9 行将 UIImageView 实例对象添加到

self.view 上。这样就完成了图片查看器的初始化代码。当点击下一张图片时，根据相应的动画展示效果展示一张新的图片。下面是动画按钮点击时相应方法的代码实现。

```
    @IBAction func animationBegin(sender: AnyObject) {
1       imageView?.image = UIImage(named: "2.jpg")
2       let animation:CATransition = CATransition()
3       animation.duration = 2;
4       animation.type = "oglFlip"
5       self.view.layer.add(animation, forKey: nil)
    }
```

代码第 1 行加载一张新的图片到 UIImageView 上。第 2 行实例化 CATransition 实例对象。第 3 行设置当前动画周期为 2s。第 4 行设置动画类型为翻转效果。最后将动画添加到 self.view 的 Layer 图层上。如图 15.1 所示为最终实现效果。

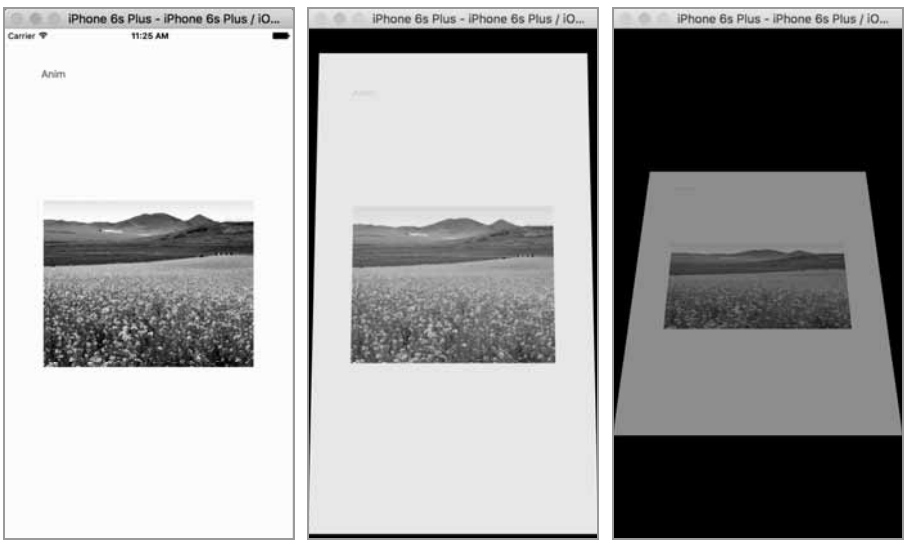



图 15.1　oglFlip 动画效果

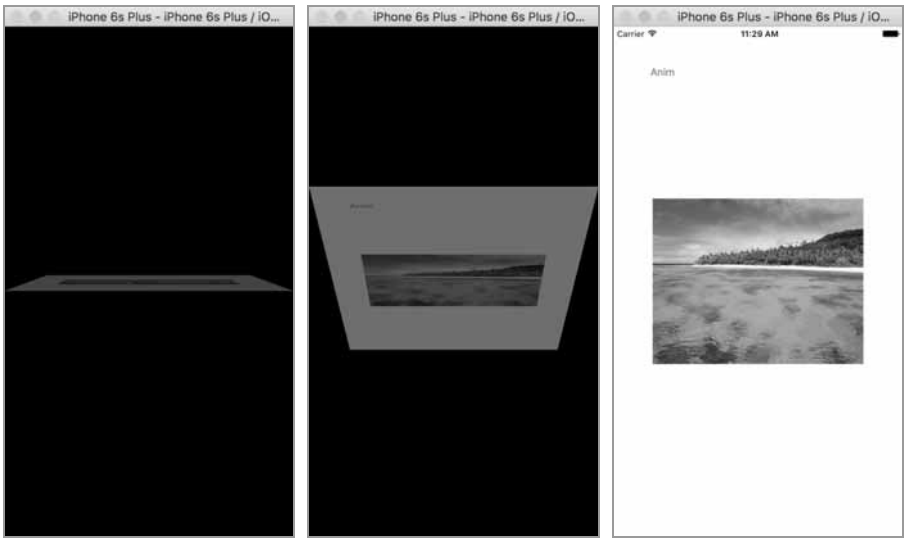



图 15.1　oglFlip 动画效果（续）

当然，这里也可以使用 animation 的 subType 属性，设置当前动画的转场方向。例如：

```
animation.subtype = kCATransitionFromRight
```

方向设置之后，最终动画效果如图 15.2 所示。

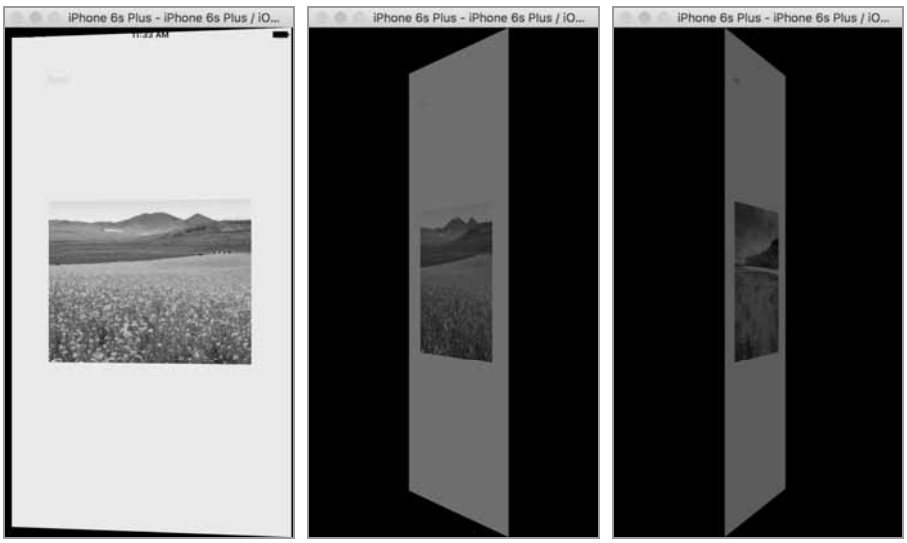



图 15.2　最终动画效果

## 15.3　CATransition 转场动画 key-effect 一览

根据第 15.1 节为大家介绍的内容，这里为大家展示不同转场效果所对应的最终动画效果，方便大家根据动画效果的 key 值快速查找到相应动画效果。如图 15.3～图 15.13 所示。

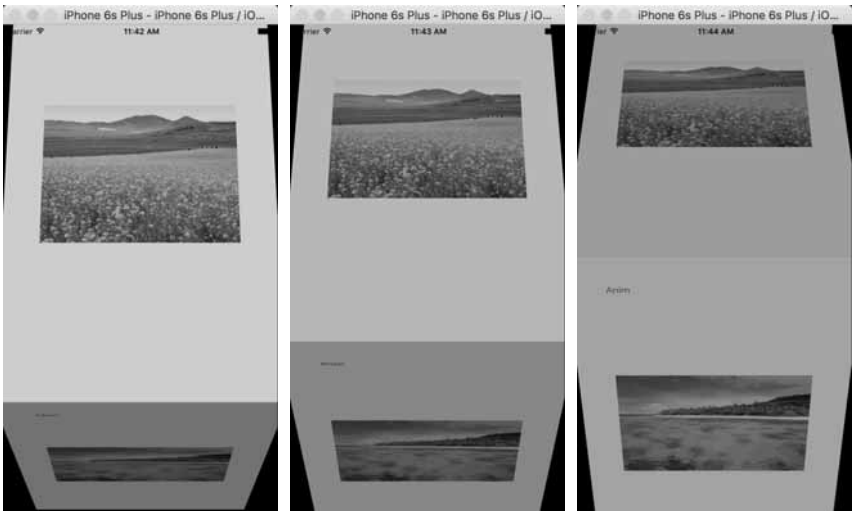



图 15.3　cube 立方体效果


图 15.4　suckEffect 收缩效果

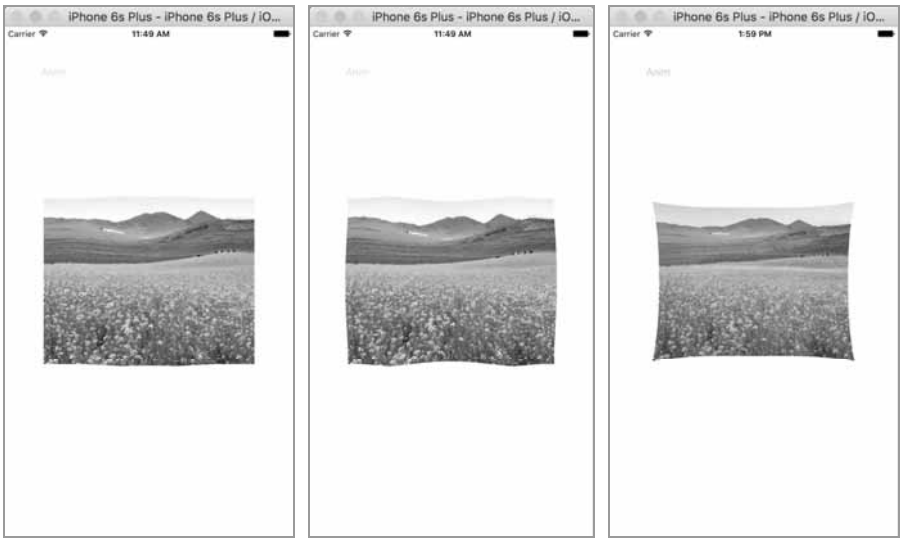



图 15.5　rippleEffect 滴水效果

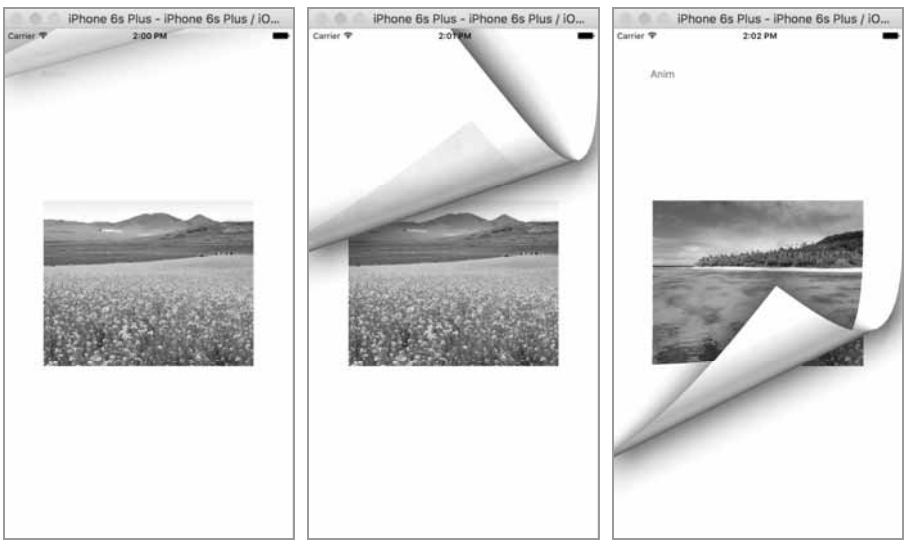



图 15.6　pageUnCurl 向下翻页

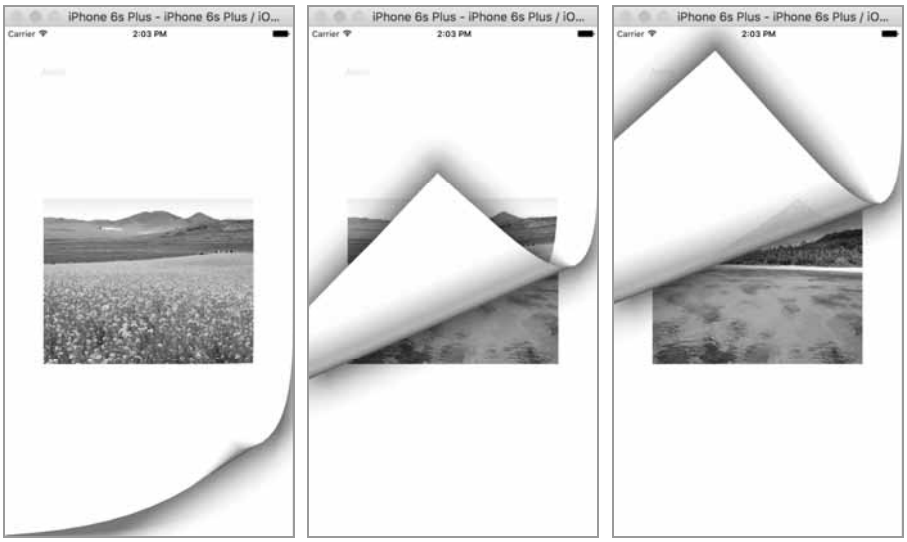



图 15.7　pageCurl 向上翻页

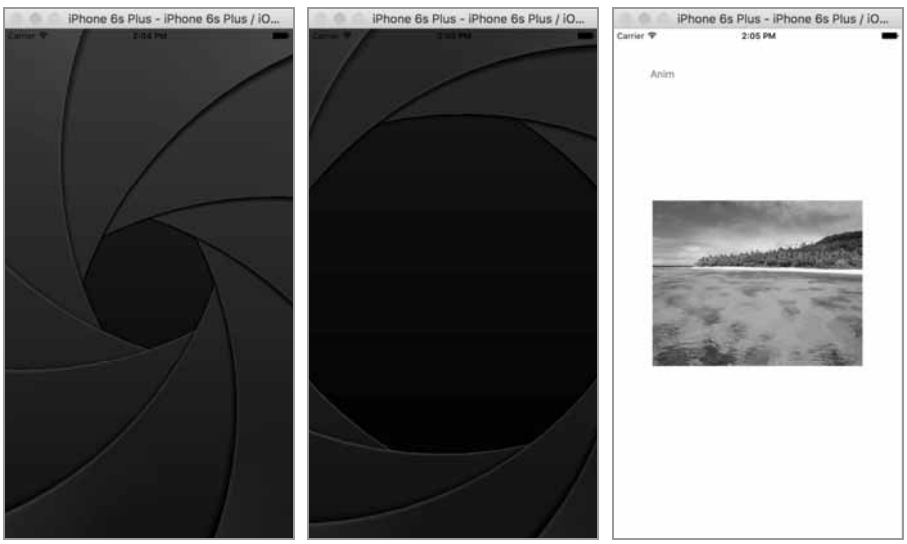



图 15.8　cameraIrisHollowOpen 相机打开

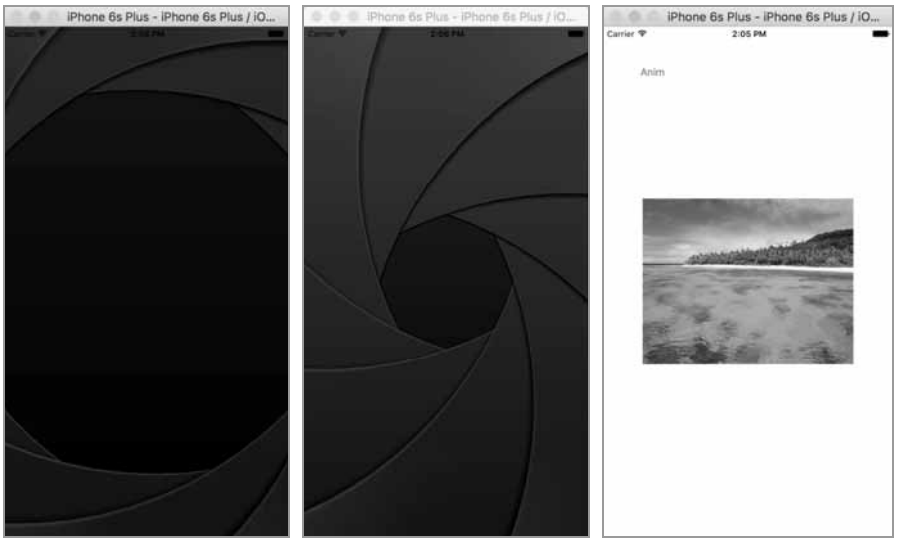



图 15.9　cameraIrisHollowClose 相机关闭

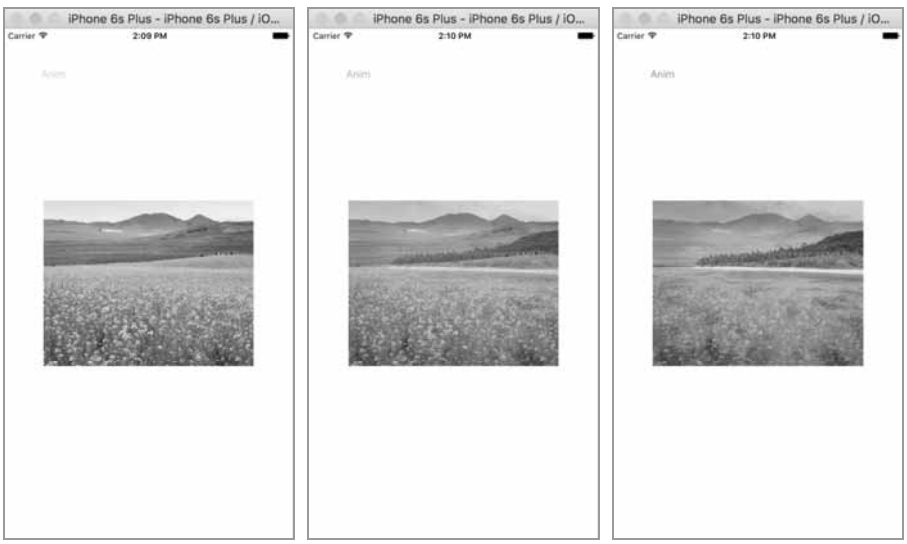



图 15.10　kCATransitionFade 淡化

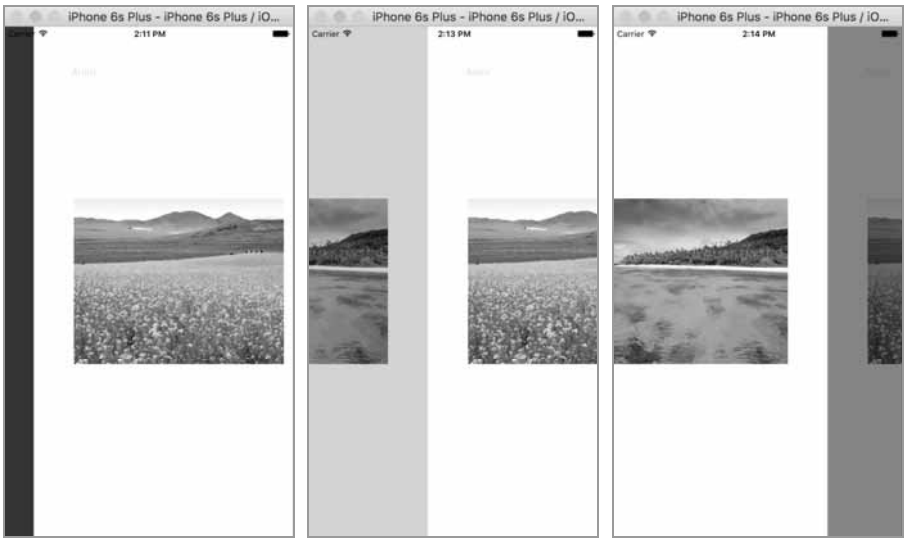



图 15.11　kCATransitionPush 推送效果

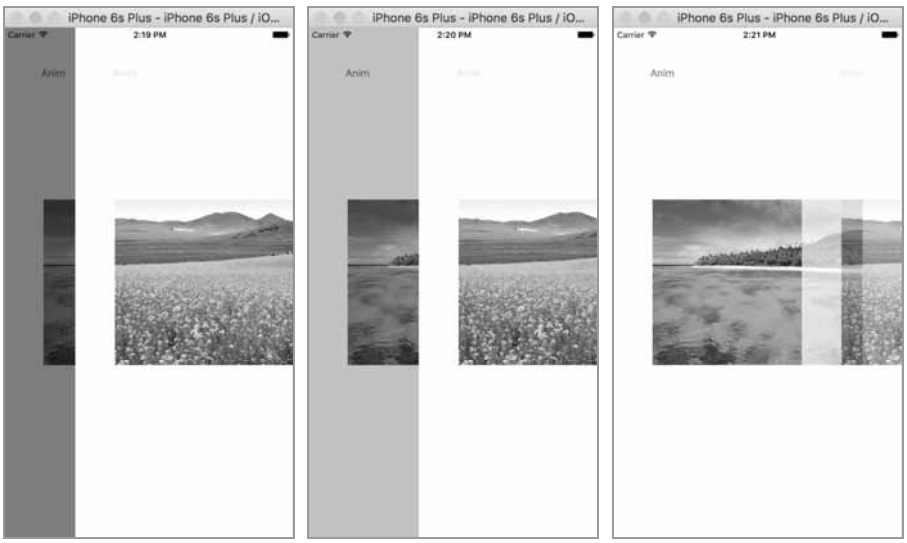



图 15.12　kCATransitionReveal 揭开效果

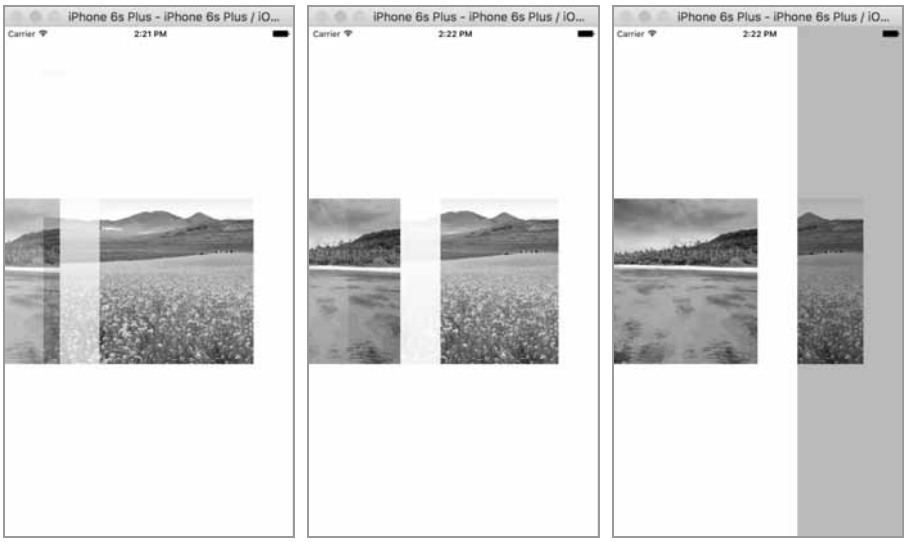



图 15.13　kCATransitionMoveIn 进入效果

## 15.4　本章小结

在不同场景的切换过程中，转场动画的使用非常频繁。本章结合核心动画框架下 CATransition 类为大家介绍了各种丰富的转场动画效果。CATransition 动画效果与 CABasicAnimation 的实现非常类似，关键是需要进行转场动画 type 的选取。iOS 中提供了大量的转场动画供大家学习。在本章的最后为大家列举了 iOS 中常用的动画类型和该类型对应的动画效果，以便大家在以后使用时快速查找。

# 第 16 章　视图过渡动画

本章内容

- 理解视图过渡动画的使用场景
- 掌握视图过渡动画常用使用方法

在使用视图控制器的时候，无外乎 Push 和 Pop 两种操作，但是在进行这两种操作的时候，iOS 很少有比较炫酷的动画效果供大家使用。不过从 iOS 7 之后，这个问题已经不存在了。iOS 提供了一套视图控制器过渡动画 API，可以允许用户自定义视图过渡动画。

## 16.1　视图控制器过渡动画相关协议

要想自定义视图控制器的过渡动画，首先需要先了解以下两个协议。

（1）UINavigationControllerDelegate

（2）UIViewControllerAnimatedTransitioning

UINavigationControllerDelegate 是视图控制器使用的委托代理协议，该协议可以代理视图的以下功能。

（1）拦截导航栏视图控制器。

（2）拦截视图控制器目标 ViewController 和源 ViewController。

在过渡动画中将使用 UINavigationControllerDelegate 的第二个功能，即拦截视图控制器目标 ViewController 和源 ViewController。该功能的回调方法如下所示。

```
optional public func navigationController(
_navigationController: UINavigationController,
animationControllerFor operation:
                    UINavigationControllerOperation,
from fromVC: UIViewController,
 to toVC: UIViewController)
-> UIViewControllerAnimatedTransitioning?
```

该方法中 fromVC 表明该视图控制器在跳转过程中来自哪个视图控制器。toVC 表明该视图控制器在跳转过程中最终跳转到何处去。所以只要拿到这两个视图控制器，在其上的 View 图层中添加想要实现的动画即可实现转场过渡动画效果。那么这些动画添加完之后如何作用在视图控制器上呢？

实际上以上所有的动画操作都将借助于 UIViewControllerAnimated Transitioning 协议，将所有动画效果最终封装成一个实例对象返回给视图控制器 UIViewController－AnimatedTransitioning 协议，定义了视图过渡动画的执行周期和执行内容。

它有两个非常重要的回调方法。

```
public func transitionDuration(
using transitionContext: UIViewControllerContextTransitioning?)
-> NSTimeInterval

public func animateTransition(
using transitionContext: UIViewControllerContextTransitioning)
```

transitionDuration 方法返回转场动画执行周期。animateTransition 方法用于构建转场动画内容。在该方法中可以通过 transitionContext 属性获取当前的 fromViewController 和 toViewController。拿到这两个视图控制器之后就可以设置当前视图控制器的各种动画效果。

# 16.2　视图控制器过渡动画代码实现

如图 16.1 所示为整个工程的文件结构。ViewController.swift 为新建的视图控制器加载文件。NewViewController.swift 为视图控制器跳转的新页面。TransitionAnim.swift 为转场动画的具体实现。

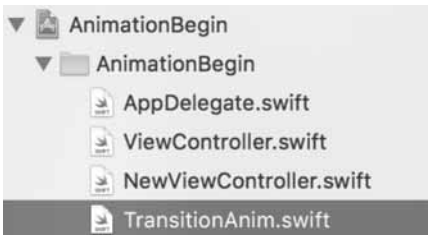



图 16.1　转场动画文件结构

第一步需要先实现 ViewController.swift 中的内容。下面是 ViewController.swift 视图控制器中的详细代码。

```
1  var viewcontroller:NewViewController = NewViewController()
2   override func viewDidLoad() {
3      super.viewDidLoad()
4      self.title = "Main Viewcontroller";
5      self.view.backgroundColor = UIColor.blue
6      navigationController!.delegate = self
   }
7   func navigationController(
          _navigationController: UINavigationController,
    animationControllerFor operation:
                      UINavigationControllerOperation,
    from fromVC: UIViewController,
    to toVC: UIViewController) ->
    UIViewControllerAnimatedTransitioning?{
8      let transitionAnim:TransitionAnim = TransitionAnim()
9      return transitionAnim
   }
10    override func touchesBegan(_ touches: Set<UITouch>, with
```

```
event: UIEvent?){

self.navigationController?.pushViewController(viewcontroller,
animated: false)
        }
```

代码第 1 行实例化一个新的视图控制器对象。第 4 行设置当前视图控制器的 title 标题。第 5 行设置当前视图控制器的背景颜色为蓝色。第 6 行设置视图控制器的回调对象为当前 self。第 7 行实现视图控制器回调方法，拦截视图控制器的跳转行为。在该方法中实现视图控制器跳转动画的实例，并将该实例对象返回。第 10 行在当前视图控制器中设置 touch 方法，实现点击空白处时由当前视图控制器跳转到新的视图控制器中的功能。

第二步，新创建的视图控制器 NewViewController.swift 实现以下内容。

```
    override func viewDidLoad() {
1       super.viewDidLoad()
2       self.view.backgroundColor = UIColor.whiteColor()
3       self.title = "New Viewcontroller"
    }
```

代码第 1 行设置当前视图控制器的背景颜色为白色。第 2 行设置当前视图控制器的标题为 title。

第三步，实现视图控制器转场动画效果。这里想实现一个从下到上的推出动画效果。TransitionAnim.swift 文件中实现代码如下所示。

```
1   class   TransitionAnim:   NSObject,UIViewControllerAnimated
Transitioning {
2     func transitionDuration(using transitionContext:
    UIViewControllerContextTransitioning?) -> NSTimeInterval{
3       return 2
    }
4    func animateTransition(using transitionContext:
              UIViewController ContextTransitioning){
5       let fromVC:UIViewController =
                      transitionContext.viewController(forKey:
```

```
                    UITransitionContextViewControllerKey.from)!
6       let toVC:UIViewController =
                      transitionContext.viewController(forKey:
                      UITransitionContextViewControllerKey.to)!
7       let fromVCRect =
                    transitionContext.initialFrame(for:fromVC)
8       let toVCRect = CGRect(x: 0,y:
                              fromVCRect.size.height*2,
                              width: fromVCRect.size.width,
                              height: fromVCRect.size.height)
9       let fromView:UIView = fromVC.view
10      let toView:UIView = toVC.view
11      fromView.frame = fromVCRect
12      toView.frame = toVCRect
13      transitionContext.containerView()?.addSubview(fromView)
14      transitionContext.containerView()?.addSubview(toView)

15       UIView.animate(withDuration: 2, animations: { () in
16          toView.frame = fromVCRect;
17          toView.alpha = 1;
18          }, completion: { (Bool) in
          transitionContext.completeTransition(true)
       })
    }
}
```

代码第 1 行继承并实现 UIViewControllerAnimatedTransitioning 协议。第 2 行重写 transitionDuration 方法，定义该转场动画持续时间为 2s。第 4 行实现 animateTransition 方法，在该方法中实现具体动画效果。代码第 5 行和第 6 行通过 transitionContext 的 key－value 属性获取当前的 fromViewController 和 toViewContoroller。第 7 行和第 8 行获取当前 fromViewController 的位置和 toViewController 的开始位置。第 9 行到第 12 行分别获取当前 fromView 对象和 toView 对象，并设置该 View 的 frame 位置信息。第 13 行和第 14 行分别将 view 添加到 transitionContext 上。第 15 行设置视图控制器 View 的动画效果。设置

toView 的最终位置和最终透明度效果。第 18 行结束当前自定义视图控制器动画效果。如图 16.2 所示为最终实现效果。

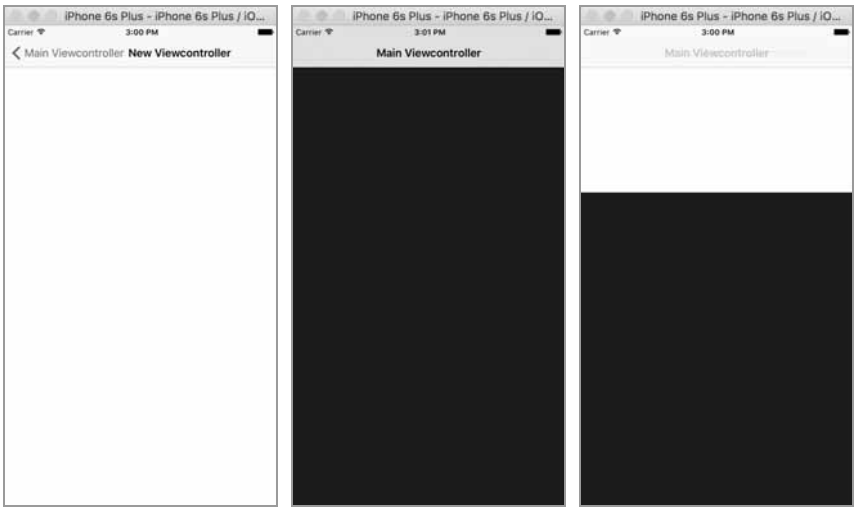



图 16.2　最终实现效果图

## 16.3　侧滑栏动画实现

有时视图控制器的过渡场景并不适合实现一些特有的转场效果，比如侧滑栏效果。而结合之前已经学习的视图层或者内容层动画都可以很方便地实现这种类似的效果。下面就来看看如何实现自定义侧滑栏动画效果。

参考网上的一些新闻类应用，发现在侧滑栏动画实现的同时还具有蒙板或者模糊效果，所以在这个案例中将把动画划分为两个部分，第一部分为蒙板模糊效果，第二部分为侧滑栏滑出及收起效果。下面是第一部分蒙板模糊效果实现代码。

```
func blurimageFromImage(image:UIImage)->UIImage{
1     let blurRadix:UInt32 = 7
2     let img:CGImageRef = image.cgImage!
3     let inProvider:CGDataProviderRef = img.dataProvider!
4     let bitmapdata:CFDataRef = inProvider.data!
5     var inputBuffer:vImage_Buffer = vImage_Buffer()
```

```
6   inputBuffer.data=  (UnsafeMutableRawPointer)(mutating:
                                  CFDataGetBytePtr(bitmapdata))
7   inputBuffer.width=(vImagePixelCount)(img.width)
8   inputBuffer.height=(vImagePixelCount)(img.height)
9   inputBuffer.rowBytes= img.bytesPerRow
10  let pixelBuffer:UnsafeMutableRawPointer =
                        malloc(img.bytesPerRow * img.height);
11  var outputBuffer:vImage_Buffer = vImage_Buffer()
12  outputBuffer.data=pixelBuffer
13  outputBuffer.width=(vImagePixelCount)(img.width)
14  outputBuffer.height=(vImagePixelCount)(img.height)
15  outputBuffer.rowBytes=img.bytesPerRow
16  vImageBoxConvolve_ARGB8888(&inputBuffer, &outputBuffer,
                           nil, 0, 0, blurRadix, blurRadix,
                           nil, UInt32(kvImageEdgeExtend))
17  let colorSpace = CGColorSpaceCreateDeviceRGB()
18  let w:Int=(Int)(outputBuffer.width)
19  let h:Int=(Int)(outputBuffer.height)
20  let ctx:CGContext = CGContext(
            data: outputBuffer.data,
            width: w,
            height: h,
            bitsPerComponent: 8,
            bytesPerRow: outputBuffer.rowBytes,
            space: colorSpace,
            bitmapInfo: image.cgImage!.bitmapInfo.rawValue)!
21   let imageRef:CGImageRef = ctx.makeImage ()!
22   let imagenew:UIImage = UIImage(cgImage:imageRef)
23   free(pixelBuffer)
24   return imagenew;
   }
```

代码第 1 行设置图片模糊效果半径。第 2 行将 UIImage 对象转换为 CGImageRef 类型。第 3 行和第 4 行获取图片的数据信息并将之转换为 CFData 数据类型。第 5 行创建一个 vImage_Buffer 对象。第 6 行到第 9 行获取图片

的 Data 数据内容、width、height 以及每行字节数。第 10 行构建一片空间大小和原始图片所占空间相同大小的数据空间。第 11 行到第 15 行，构建一个输出 vImage_Buffer 对象，并将输入图片的一些属性信息拷贝到输出 vImage_Buffer 对象中。第 16 行调用 vImageBoxConvolve_ARGB8888 方法，生成模糊效果的图片。第 17 行构建 RGB 彩色空间。第 18 行和第 19 行获取输出图片的宽、高。第 20 行调用 CGCreate 方法创建出输出 CGContextRef。第 21 和第 22 行通过创建出的 CGContextRef 构建出最终 UIImage 图片。最后两行释放 pixelBuffer 像素空间并将最终生成的 UIImage 对象返回。

这个方法需要传递的参数为 UIImage，而在点击侧滑栏的时候，往往拿到的都是 UIView 对象，所以这里还需要一个 UIVIew 到 UIImage 的转换方法。该方法的实现代码如下。

```
    func imageFromUIView(view:UIView)->UIImage{
1       UIGraphicsBeginImageContext(view.frame.size)
2       let content:CGContextRef =
                                UIGraphicsGetCurrentContext()!
3       view.layer.render(in:content)
4       let imagenew:UIImage =
                UIGraphicsGetImageFromCurrentImageContext()
5       UIGraphicsEndImageContext();
6       return imagenew
    }
```

代码第 1 和第 5 行配合使用，表明通过 UIGraphics 构建图片上下文开始和结束。第 2 行获取当前上下文。第 3 行将视图 View 的 Layer 图层渲染到当前图层的上下文中。第 4 行将上下文转换为 UIImage 对象。最后一行将 UIImage 对象返回。

通过以上两部分可以实现 UIView 到蒙板效果的实现。下面是第二部分，侧滑栏弹出效果。该代码有界面初始化，侧滑栏弹出和侧滑栏收起三个部分。以下是第一部分界面初始化。

```
let DEVICE_SCREEN_HEIGHT = UIScreen.mainScreen().bounds.height
let DEVICE_SCREEN_WIDTH = UIScreen.mainScreen().bounds.width
class SliderViewController: UIViewController {
  internal var blurView:UIView?
  internal var contentView:UIView?
  override func viewDidLoad() {
      super.viewDidLoad()
      blurView = UIView(frame: CGRect(x: 0, y: 0,
                          width: DEVICE_SCREEN_WIDTH,
                         height: DEVICE_SCREEN_HEIGHT))
      self.view.addSubview(blurView!)
      contentView = UIView(frame:
                  CGRect(x: -DEVICE_SCREEN_WIDTH*0.60, y: 0,
                  width: DEVICE_SCREEN_WIDTH*0.60,
                  height: DEVICE_SCREEN_HEIGHT))
      contentView!.backgroundColor = UIColor(red: 255.0 / 255.0,
       green: 127.0 / 255.0, blue: 79.0 / 255.0, alpha: 1.0)
      self.view.addSubview(contentView!)
  }
```

代码第 1 行和第 2 行定义当前屏幕的宽高。第 4 行和第 5 行定义模糊视图 UIView 和侧滑栏 View。第 8 行和第 9 行初始化模糊视图的 frame 属性并将其添加到 self.view 上。第 10 行到第 11 行初始化侧滑栏 frame 属性，设置其背景颜色为橙色，最后将其添加到 self.view 上。模糊视图和侧滑栏添加到 self.view 上之后还需要实现侧滑栏弹出和收起效果。下面是侧滑栏收起动画效果的具体实现代码。

```
  func sliderVCDismiss(){
      UIView.animate(WithDuration:0.5, animations:
      {()->Void in
          self.contentView!.frame =  CGRect(
                              x: -DEVICE_SCREEN_WIDTH*0.6,
                              y: 0,
                         width: DEVICE_SCREEN_WIDTH*0.6,
                        height: DEVICE_SCREEN_HEIGHT)
```

```
    3            self.contentView!.alpha = 0.9
    4            }, completion: {(finished:Bool)->Void in
                    self.view.alpha=0.0
            })
        }
```

代码第 1 行使用 UIView 调用 animate 方法启动侧滑栏弹出动画。第 2 行和第 3 行设置该侧滑栏的最终停留位置以及侧滑栏的 alpha 透明度属性。在视图控制器中调用该方法则可以实现侧滑栏收起。下面是侧滑栏弹出动画代码。

```
    1   func sliderLeftViewAnimStart(){
    2       var windowview:UIView = UIView()
    3       windowview = UIApplication.shared.
                              keyWindow!.rootViewController!.view
    4       blurView?.layer.contents = blurimageFromImage(
                             imageFromUIView(windowview)).cgImage
    5       self.view.alpha=1.0
    6       UIView.animate(WithDuration:0.5, animations: {()->Void in
    7           self.contentView!.frame = CGRect(x: 0, y: 0, width:
DEVICE_SCREEN_WIDTH*0.6, height: DEVICE_SCREEN_HEIGHT)
    8          self.contentView!.alpha = 0.9
    9          }, completion: {(finished:Bool)->Void in
           })
       }
```

代码第 2 行和第 3 行获取当前视图控制器的 rootViewController 的根视图。第 4 行调用视图模糊方法，并将其结果装载到模糊视图的 Layer 图层中。第 5 行设置 alpha 属性为 1。第 6 行使用 UIView 调用 animate 方法开始侧滑栏弹出动画。第 7 行和第 8 行设置侧滑栏最终弹出的位置及 alpha 属性。如图 16.3 所示为最终实现的效果。

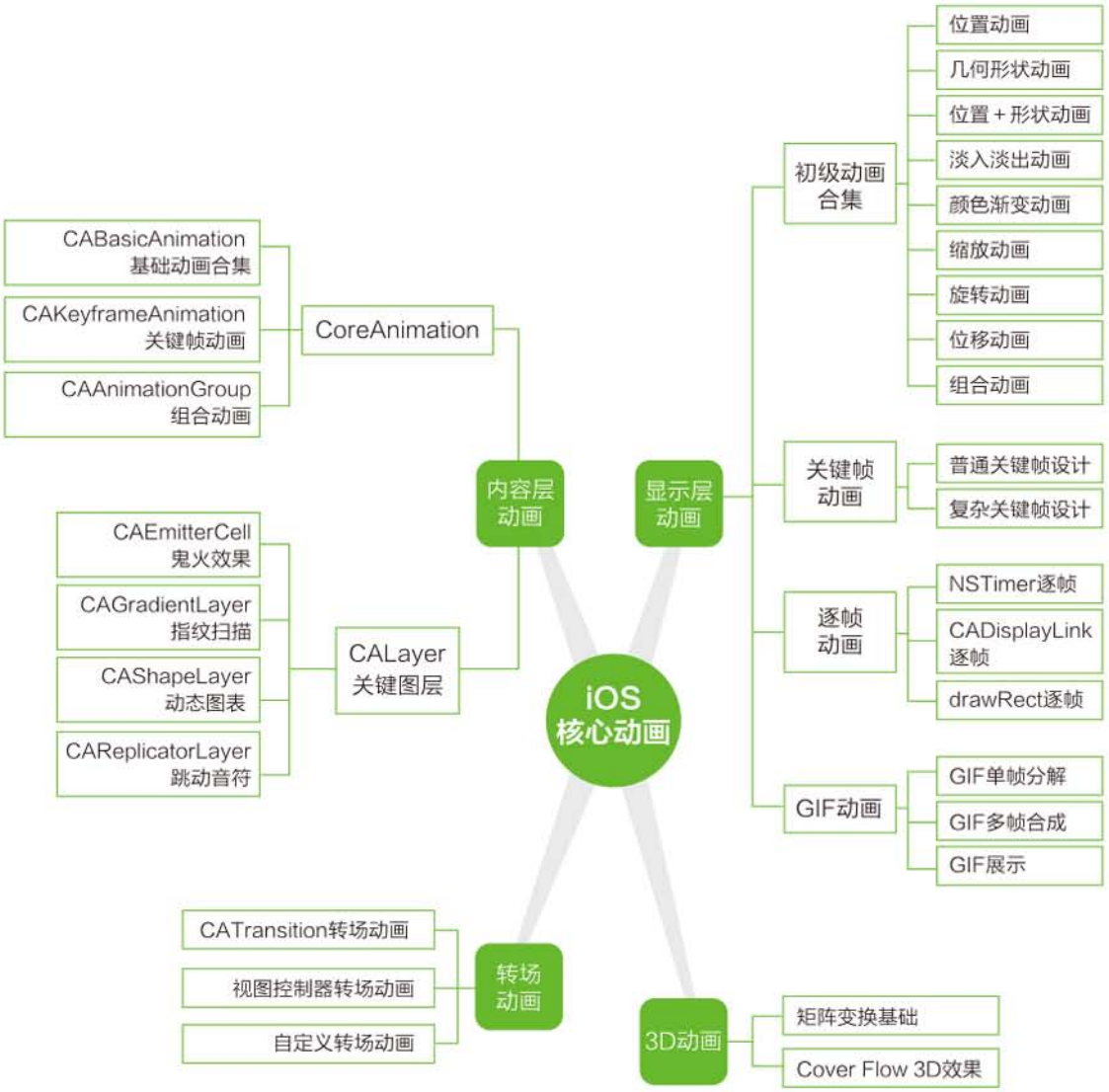

图 16.3　侧滑栏最终效果

## 16.4　本章小结

本章通过 UINavigationControllerDelegate 实现了自定义转场动画，该动画在 iOS 7 之后可以使用。并且通过 UIViewControllerAnimatedTransitioning 类在回调方法中自定义各种特有效果，不过该方法在使用的时候受到视图控制器的跳转限制。在第二个案例中则结合之前学过的 UIView 视图层动画，直接作用在 self.view 上模拟出转场动画场景。

# 反侵权盗版声明

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版




上架建议：计算机>移动开发


博文视点Broadview



@博文视点Broadview

策划编辑：杨中兴
责任编辑：黄爱萍
封面设计：侯士卿