第

二

部

分

控

件

篇

# 第十章 UIView和CALayer层

窗口和UIView视图是为iOS应用程序构造用户界面的可视化组件。窗口为内容显示提供背景平台，而UIview视图负责绝大部分的内容描绘，并负责响应用户的交互。

UIView的显示依赖于其内部的CALayer层对象。UIView的显示由一个CALayer类来管理，其本身只负责说明要显示的内容。当访问和调整UIView中如frame、bounds等关于绘制的属性，实际上都是在访问和调整它所包含的CALayer的相关属性。

本章将逐步介绍UIView视图及CALayer层的原理和一些具体的使用方法。

## 10.1 视图UIView

### 10.1.1 UIView概述

UIWindow

UILabel

NSObject

UITableView

UITableViewCell

UITextView

UIScrollView

UICollectionView

UIPickerView

UIResponder

UIProgressView

UIActivationView

UIView

UIButton

UISearchBar

UIDatePicker

UIActionSheet

UIPageControl

UITabBar

UISegmentControl

UIToolBar

UITextField

UIControl

UINavigationBar

UISlider

UISwitch

UIImageView

UIAlertView

UIWebView

图10-1 UIView的继承层次结构

UIView是UIKit框架里面最基础的视图类。UIView类定义了一个矩形的区域，管理该矩形区域内的所有显示内容。在应用程序中还需要响应该区域中发生的触碰事件。视图是应用程序与用户交互的重要机制。

UIView类定义了视图的基本行为，但并不定义其视觉表现，而是UIKit通过子类来为文本框、按键及工具条这样的标准界面元素定义具体的外观和行为。

UIView继承层次如图10-1所示。

以下内容是对UIView视图和UIWindow窗口进行大致介绍。

UIView视图和UIWindow窗口是为iOS应用程序构造用户界面的可视组件。窗口为内容显示提供背景平台，而视图负责绝大部分的内容描画与响应用户的交互。

iOS程序启动后，创建的第一个视图控件就是UIWindow，接着创建视图控制器的view，并将该view添加到UIWindow上，于是控制器的view就显示在屏幕上了。

和桌面Mac OS的应用程序有所不同，iOS应用程序通常只有一个窗口，表示为一个UIWindow类的实例，应用程序在启动时创建这个窗口，并往窗口中加入一个或多个视图，然后将它显示出来。窗口一旦显示出类，基本上就不会再使用到它了，而更多的是对UIView视图的操作。

在iOS应用程序中，窗口对象并没有像关闭框或标题栏这样的区域，所以用户不能直接对其进行关闭或其他操作。

在Mac OS中,NSWindow的父类是NSResponder,而在iOS系统中，UIWindow的父类是UIView。因此UIWindow窗口在iOS系统中也是一个视图对象。

尽管iOS支持多窗口的存在，但是最好不要创建多窗口。比如当希望在自己内容的上方显示警告窗口时，可以使用UIKit提供的警告视图控制器UIAlertController,而不应该在创建一个新的窗口。

创建新项目，模板为iOS中的【Single View App】,创建完成后在ViewController.swift中编写代码来创建一个视图并定义其属性。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import UIKit  class ViewController: UIViewController {  override func viewDidLoad() {  super. viewDidLoad()  //定义视图位置和大小并创建视图  let view = UIView(frame: CGRect(x: 40, y: 100, width: 334, height: 334))  //定义其背景颜色为红色  view.backgroundColor = UIColor.black   //添加视图于根视图  self.view.addSubview(view)  }  } |

运行项目，运行效果如图10-2所示。

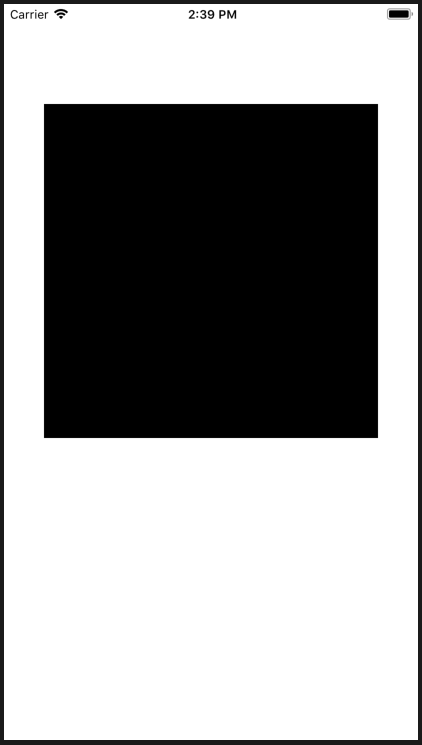


图10-2 使用UIView创建正方形视图

下面将逐一介绍UIView中的各种属性。

### 10.1.2 UIView的外观属性

UIView类的外观属性常用的主要有背景颜色、切边、透明度、显示与隐藏。

背景颜色可以用UIColor来设置单一颜色背景。UIColor是UIKit中存储颜色属性的重要类，其中包含一些常用颜色的方法，如红色、白色、黑色等。一个视图的背景还可以设为一个图片，如把10.1.1节中的代码的5~10行换成：

|  |  |
| --- | --- |
| 7  8  9  10 | let view = UIView(frame: CGRect(x: 40, y: 100, width: 334, height: 334))  let image = UIImage(named: "gezi")  view.backgroundColor = UIColor.init(patternImage:image!)  self.view.addSubview(view) |

点击项目文件夹，在弹出的菜单中选择【Add a File】，添加一张图片到项目中。通过UIImage对象加载资源文件夹的一张图片，然后使用UIColor中的init方法将加载的图片设置为背景。运行效果如图10-3所示。

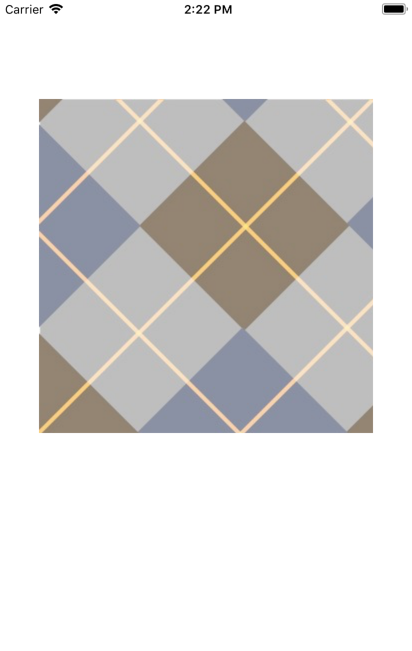


图10-3 设置格子图片为视图背景

UIView可以通过hidden属性设置视图的显示与隐藏。

视图的hidden属性为布尔值类型，表示UIView的视图是否处于隐藏状态，默认值为false。修改10.1.1代码的5~10行如下：

|  |  |
| --- | --- |
| 7  8  9  10 | let view = UIView(frame: CGRect(x: 40, y: 100, width: 334, height: 334))  view.isHidden = true  view.backgroundColor = UIColor.black  self.view.addSubview(view) |

这样将不会在屏幕上看到已经设置好的视图。

视图的透明度可根据调节透明度属性alpha的值来实现。

视图的透明度alpha属性是一个浮点数类型，取值范围为0~1.0，从完全透明到完全不透明，默认alpha值为1。修改10.1.1代码的5~10行如下：

|  |  |
| --- | --- |
| 7  8  9  10 | let view = UIView(frame: CGRect(x: 40, y: 100, width: 334, height: 334))  view.alpha = 0.2  view.backgroundColor = UIColor.black  self.view.addSubview(view) |

运行效果如图10-4所示。



图10-4 设置视图透明度为0.2

注：当把alpha值设为0或者isHidden值设为true时，当前的UIView视图及其子视图都会被隐藏，不管子视图的alpha或者isHidden值为多少。

### 10.1.3 UIView的几何属性

在iOS系统中，点的位置是依靠iOS系统的坐标系来确定的，坐标原点位于左上角。iOS包含两个坐标系，其中UIKit框架的坐标系是X轴正方向向右，Y轴正方向向下，而标准的Quartz 2D绘图坐标系为X轴正方向向右，Y轴正方向向上。

一个UIView对象的位置和大小属性，是由两个属性值来组成的：CGPoint和CGSize。其中CGPoint由x、y两个值决定，代表视图在父视图上的相对于原点的位置，CGSize由width和height两个值决定，代表视图的大小。在声明一个UIView的对象时，可以用以下三种属性来定义：frame、bounds和center。

frame决定该视图在父视图的位置及其大小。frame使用方法如下：

|  |  |
| --- | --- |
| 1  2 | UIView.frame(CGRect(x:，y:，width:，height:))  UIView.frame(CGRect(origin:CGPoint，size:CGSize)) |

Bounds决定该视图在其本身的坐标系中的位置和大小，通常用于改变视图的大小。其origin属性默认为（0，0）,即使设置了其他值也无法改变该视图的位置，但其子视图的位置会发生改变。使用方法如下：

|  |  |
| --- | --- |
| 1  2 | UIView.bounds(CGRect(x:，y:，width:，height:))  UIView.bounds(CGRect(origin:CGPoint，size:CGSize)) |

比如想改变某视图的大小，就把x、y设为0或者把CGPoint对象中的x、y设为0：

|  |  |
| --- | --- |
| 1 | UIView.frame(CGRect(x:0，y:0,width:100,height:100)) |

该行代码就是把视图长宽都设为100。注意，在改变该视图大小时，其中心位置不变。

Center决定该视图的中心在其父视图中的位置。使用方法如下：

|  |  |
| --- | --- |
| 1 | UIView.center(CGPoint(x:，y:)) |

下面将用代码来定义视图坐标和尺寸信息。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import UIKit  class ViewController: UIViewController {      override func viewDidLoad() {          super.viewDidLoad()          // Do any additional setup after loading the view, typically from a nib.          //定义并创建视图,设置其背景颜色为红色          let frame = CGRect(x: 40, y: 100, width: 334, height: 334)          let view = UIView(frame: frame)          view.backgroundColor = UIColor.red          //定义并创建另一个视图，设置其背景颜色为黑          let subview = UIView(frame: CGRect(x: 0, y: 0, width: 260, height: 260))          subview.backgroundColor = UIColor.black          //添加第二个视图于第一个视图          view.addSubview(subview)          //添加第一个视图于根视图          self.view.addSubview(view)      }  } |

在17~20行代码中将视图作为子对象添加到另一个视图对象中的方法请查看10.1.4节的介绍。

运行该项目，运行效果如图10-5所示。

接下来设置子视图的bounds属性,在第15行代码下添加一行代码。

|  |  |
| --- | --- |
| 15  16  17 | subview.backgroundColor = UIColor.black  subview.bounds = CGRect(x: 0, y: 0, width: 200, height: 200)  view.addSubview(subview) |

运行项目，运行效果如图10-6所示。可以看到，子视图中心位置未变。

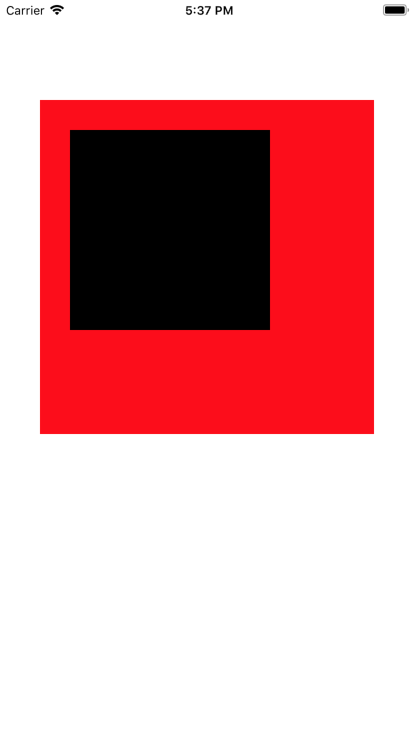
 

图10-5 在一个视图上添加另一个视图 图10-6改变子视图大小

接下来设置父视图的bounds属性,在第8行代码下方添加代码。

|  |  |
| --- | --- |
| 8  9  10 | let view = UIView(frame: frame)  view.bounds = CGRect(x: -100, y: -100, width: 334, height: 334)  view.backgroundColor = UIColor.red |

运行项目，效果如图10-7所示。可以看到subview视图位置改变，这是因为view视图的左上角的坐标在自身的坐标系中变为（-100，-100），subview作为view的子视图要以view的坐标系中的（0，0）为原点。

设置父视图的center属性，在第20行上方添加代码。

|  |  |
| --- | --- |
| 19  20 | view.center = self.view.center  self.view.addSubview(view) |

运行项目，效果如图10-8所示，把view视图的中心移到跟视图的中心。

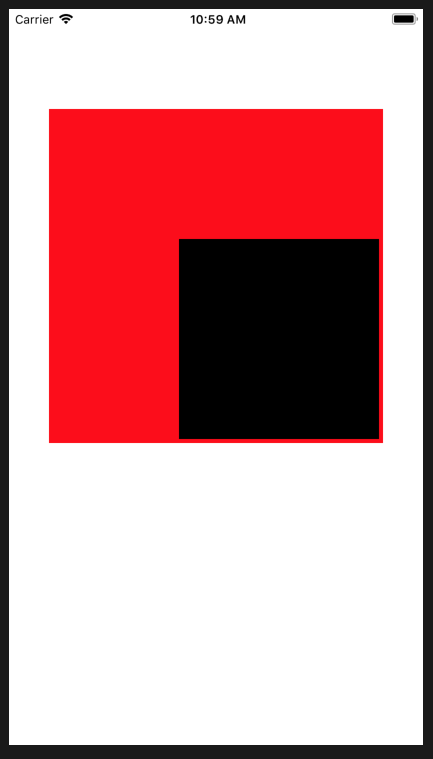
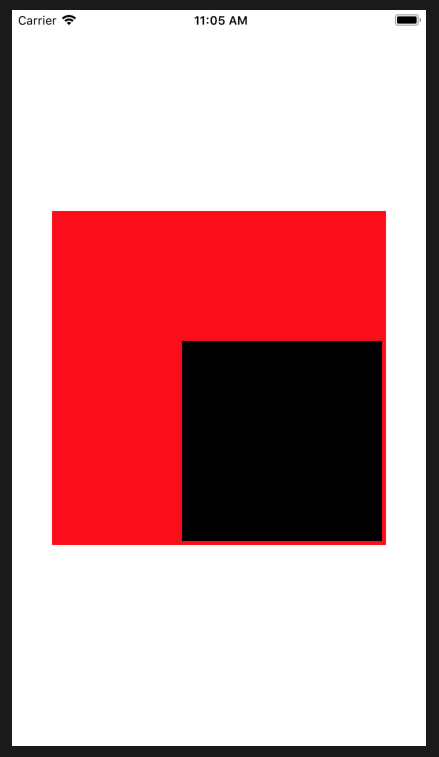
 

图10-7 图10-8

### 10.1.4 UIView的嵌套和层次关系

一个视图中可以嵌套多个子视图。开发者可以根据自身需求通过多级嵌套来形成复杂的图层结构。视图的这种布局方式被称为视图层次，一个视图可以包含任意数量的子视图，通过为子视图添加子视图的方式，可以实现任意深度的嵌套。

对于子视图在屏幕上的显示方式，虽然子视图总是显示在父视图的上方，但是同一层子视图之间可能会发生重叠或者覆盖。这是由视图层次的组织方式来决定的，它还决定了视图对于事件的响应和变化方式，子视图的组织方由它们的父视图负责管理。

UIView对于视图层次的管理方法如下表10-1所示。

表10-1 UIView视图层次管理方法

|  |  |
| --- | --- |
| 方法名称 | 方法说明 |
| addSubview(view:) | 添加视图于父视图层级结构的最上层 |
| insertSubview(view:,at:) | 在指定的位置上插入式图 |
| insertSubview(view:,aboveSubview:) | 将视图添加到指定视图的上方 |
| insertSubview(view:,belowSubview:) | 将视图添加到指定视图的下方 |
| bringSubview(toFront:) | 将指定的子视图移动到最前面 |
| sendSubview(toBack:) | 将指定的子视图移动到最后面 |
| exchangeSubview(at:,withSubviewAt:) | 交换两个指定位置的子视图在父视图的位置 |
| removeFromSuperview | 将子视图从父视图中删除 |

接下来将要用代码创建相同尺寸但拥有不同背景颜色和位置的3个视图，并将它们都添加到根视图上，以便展示视图之间的层次转换。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import UIKit  class ViewController: UIViewController {  override func viewDidLoad() {        super.viewDidLoad()        // Do any additional setup after loading the view, typically from a nib.        //创建第一个视图并添加到根视图中        let view1 = UIView(frame: CGRect(x: 40, y: 100, width: 200, height: 200))        view1.backgroundColor = UIColor.black        self.view.addSubview(view1)        //创建第二个视图并添加到根视图中        let view2 = UIView(frame: CGRect(x: 100, y: 140, width: 200, height: 200))        view2.backgroundColor = UIColor.brown        self.view.addSubview(view2)        //创建第三个视图并添加到根视图中        let view3 = UIView(frame: CGRect(x: 160, y: 180, width: 200, height: 200))        view3.backgroundColor = UIColor.lightGray        self.view.addSubview(view3)      }  } |

运行项目，结果如图10-9所示。可以看到最初的层次是按照添加到根视图的先后来排的，先添加到根视图中的视图在最后一层，最晚添加的在最上层。

接下来调整view1和view2在父视图中的层次。在第17行代码下面添加一行代码：

|  |  |
| --- | --- |
| 17  18 | self.view.addSubview(view3)  self.view.insertSubview(view1, aboveSubview: view2) |

运行项目，结果如图10-10所示。可以看到view1的层次在view2之上。继续添加代码：

|  |  |
| --- | --- |
| 17  18  19 | self.view.addSubview(view3)  self.view.insertSubview(view1, aboveSubview: view2)  self.view.sendSubview(toBack: view3) |

运行项目，结果如图10-11所示。继续添加代码：

|  |  |
| --- | --- |
| 19  20 | self.view.sendSubview(toBack: view3)  view2.removeFromSuperview() |

运行项目，运行效果如图10-12所示。

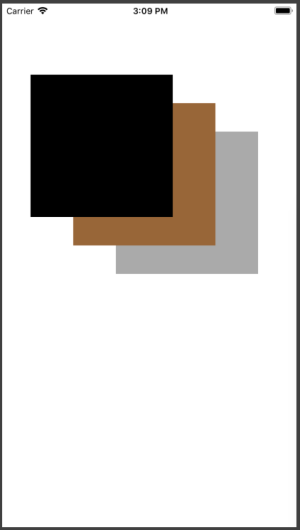
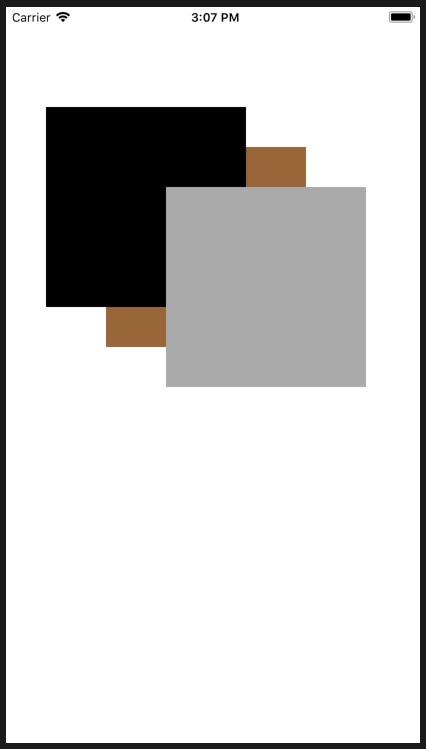
 

图10-9 图10-10

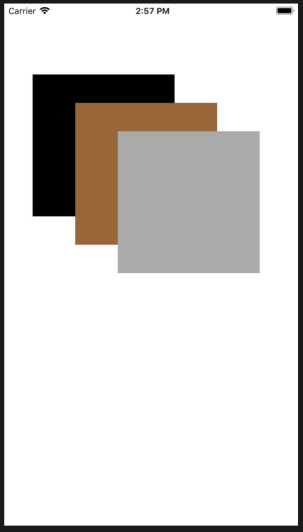
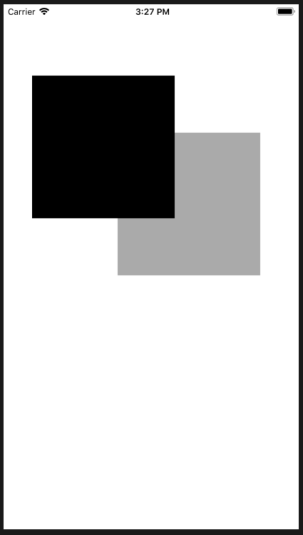
 

图10-11 图10-12

### 10.1.5 UIView的变形操作

UIView的变形操作由CGAffinetransform结构体控制。该结构体代表了一种用于仿射变换的矩阵。结构体的参数指定了从一个坐标系的点转化成另外一个坐标系的点的规则。

仿射变换是一种特殊的映射，保留在一个路径中的平行线，但不一定保留长度或角度。缩放、旋转、平移是最常用的仿射变换。开发者通常不会直接创建一个仿射变换，只需要根据现有的参数，修改现有的仿射变换即可。

常用的几种仿射变换如下表10-2所示。

表10-2 几种常用的仿射变换

|  |  |
| --- | --- |
| 名称 | 说明 |
| translatedBy（x:,y:） | 对已存在的矩阵进行平移 |
| Invert() | 对已存在的矩阵进行反转 |
| scaledBy(x:,y:) | 对已存在的矩阵进行旋转 |
| Rotated(by:) | 对已存在的矩阵进行反转 |
| Concatenating(t2:) | 对仿射效果进行叠加操作 |

接下来会使用上表的方法实现视图的变形。

创建新项目，模板为iOS中的【Single View App】,创建完成后在ViewController.swift中编写代码来创建一个视图并定义其属性。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import UIKit  class ViewController: UIViewController {  override func viewDidLoad() {    super.viewDidLoad()  // Do any additional setup after loading the view, typically from a nib.    //创建一个视图，背景为黑色，并添加到根视图   let view = UIView(frame: CGRect(x: 50, y: 200, width: 314, height: 100))    view.backgroundColor = UIColor.black    self.view.addSubview(view)  }  } |

运行项目，结果如图10-13所示。在第9行代码后添加代码。

|  |  |
| --- | --- |
| 9  10 | self.view.addSubview(view)  view.transform = view.transform.translatedBy(x: 50, y: 0) |

结果如图10-14所示，可以看到视图view向右平移了40。

将第10行代码改为：

|  |  |
| --- | --- |
| 10 | view.transform = view.transform.scaledBy(x: 0.5, y: 0.5) |

运行结果如图10-15所示，可以看到view视图在x水平方向和y垂直方向上各缩小了0.5倍，其中心位置没有变化。

将第10行代码改为：

|  |  |
| --- | --- |
| 10 | view.transform = view.transform.rotated(by: CGFloat(Double.pi/4)) |

运行结果如图10-16所示。可以看到view视图以顺时针方向旋转了45°。Double.pi在swift语言中表示圆周率。

以上对视图的平移、缩放和旋转都是采用已封装好的方法，而对于斜切却没有。开发者只能通过设置参数的值来进行斜切变形。参数说明如下表10-3所示。

表10-3 变形参数

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 说明 | 参数 | 说明 |
| a | 水平方向上的缩放因子 | d | 垂直方向上的缩放因子 |
| b | 水平方向上的斜切因子 | tx | 水平方向上的位移因子 |
| c | 垂直方向上的斜切因子 | ty | 水平方向上的位移因子 |

在第10行代码下方添加代码：

|  |  |
| --- | --- |
| 10  11  12 | view.transform = view.transform.rotated(by: CGFloat(Double.pi/4))  view.transform.b = 0.3  view.transform.c = 0.3 |

运行效果如图10-17所示。

图10-13 图10-14

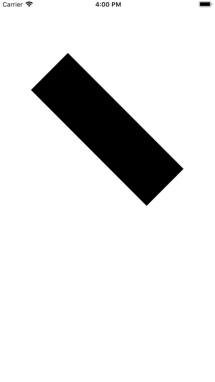
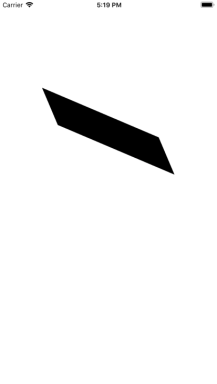
  

图10-15 图10-16 图10-17

## 10.2 CALayer层

在本章开始就已说明，UIView是对视图内容的管理，而UIView真正的绘图部分，是由CALayer（Core Animation Layer）类来管理的。对于创建的视图或者控件，有时不会满足于其只是矩形的视觉效果。通过CALayer可以给视图添加边框、阴影、圆角、颜色渐变等效果。

### 10.2.1 CAlayer边框

创建新项目，模板为iOS中的【Single View App】,创建完成后在ViewController.swift中编写以下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import UIKit  class ViewController: UIViewController {  override func viewDidLoad() {    super.viewDidLoad()    // Do any additional setup after loading the view, typically from a nib.    //创建一个视图，背景为黑色，并添加到根视图      let view = UIView(frame: CGRect(x: 100, y: 200, width: 214, height: 214))      view.backgroundColor = UIColor.black      //设置边框效果，边框宽为20，颜色为灰色      view.layer.borderWidth = 20     view.layer.borderColor = UIColor.gray.cgColor     //把该视图添加到根视图     self.view.addSubview(view)  }  } |

对于第11行，由于CALayer层的颜色使用CGColor类型，所以要使用UIColor中的cgColor属性。运行项目，运行效果如图10-18所示。

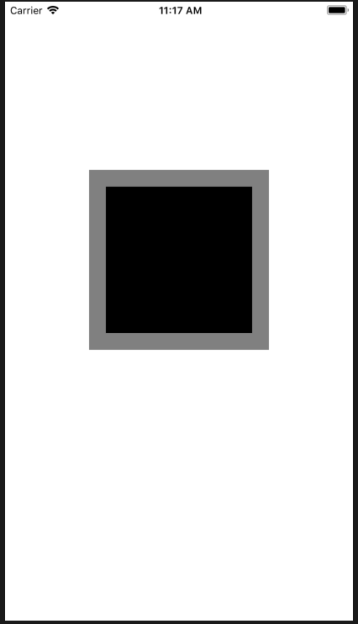
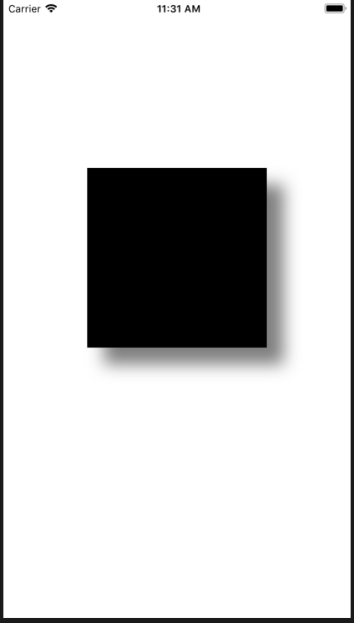
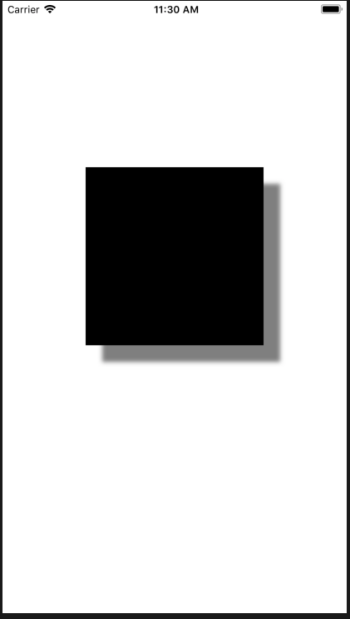
  

图10-18 图10-19 图10-20

### 10.2.2 CALayer阴影

将10.2.1中的代码的9~11行改为：

|  |  |
| --- | --- |
| 9  10  11  12  13  14 | //设置阴影效果，颜色为黑色,阴影偏移量为（20，20）  view.layer.shadowOffset = CGSize(width: 20, height: 20)  view.layer.shadowColor = UIColor.black.cgColor  //设置阴影透明度为0.5，圆角半径为10  view.layer.shadowOpacity = 0.5  view.layer.shadowRadius = 10 |

运行项目，运行效果如图10-19所示。shadowOffset属性为阴影偏移量，类型为CGSize.shadowOpacity默认值为0.0，shadowRadius是为了实现阴影的模糊效果，使阴影更自然平滑,其默认值为3.0。不设置shadowRadius的运行效果如图10-20所示。

### 10.2.3 CALayer圆角

将10.2.1中的代码的9~11行改为：

|  |  |
| --- | --- |
| 9  10 | //设置圆角  view .layer.cornerRadius = 40 |

运行项目，裕兴结果如图10-21所示。可以看到在层的四个角处绘制了指定半径的圆角。

把cornerRadius的值改为正方形视图宽的一半，运行效果如图10-22所示。可以看到视图变为正圆形。继续扩大cornerRadius的值为150，大于正方形宽的一半，运行效果如图10-23所示。

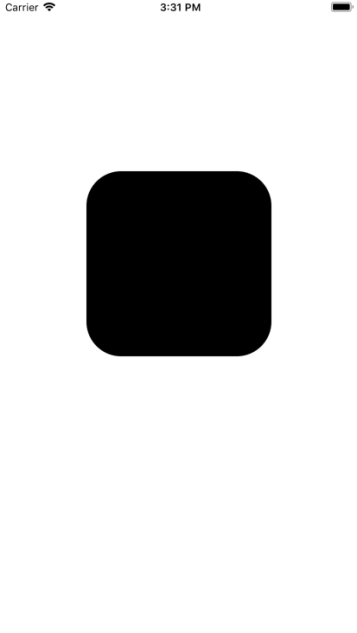
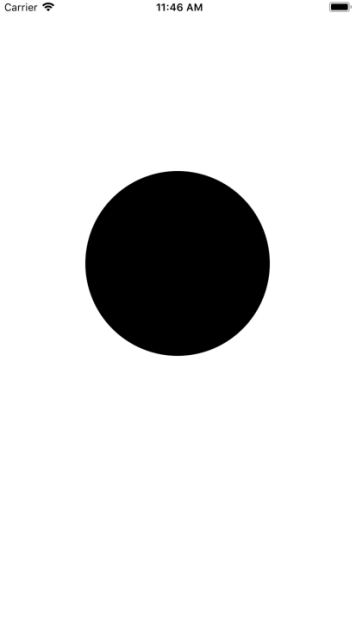
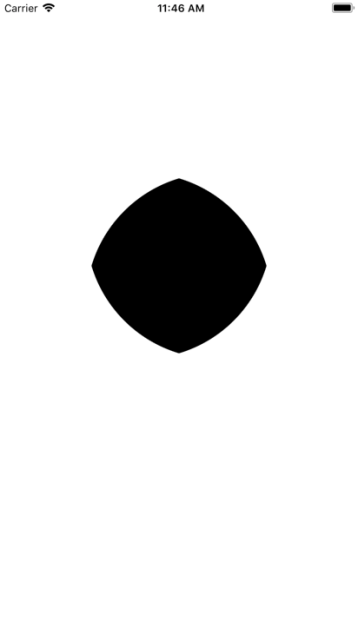
  

图10-21 图10-22 图10-23

### 10.2.4 颜色渐变

CALayer层有一点和UIView很相似，那就是CALayer层也有嵌套和层级关系。通过对各层的绘图控制可以实现多种效果。

CALayer层的颜色渐变是由其CAGradientLayer层控制的。通过该层可以实现多种颜色在某一视图的颜色渐变。

创建新项目，模板为iOS中的【Single View App】,创建完成后在ViewControl- -ler.swift中编写以下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import UIKit  class ViewController: UIViewController {  override func viewDidLoad() {      super.viewDidLoad()     // Do any additional setup after loading the view, typically from a nib.      //创建第一个视图并添加到根视图中     //创建一个视图和一个渐变层，使渐变层位置和大与视图一样大     let rect = CGRect(x: 50, y: 100, width: 214, height: 214)     let view = UIView(frame: rect)      let layer = CAGradientLayer()     layer.frame=view.frame      //设置渐变颜色数组     let firstColor = UIColor.white.cgColor     let secondColor = UIColor.red.cgColor     layer.colors = [firstColor,secondColor,firstColor]      //设置渐变层起点和终点     layer.startPoint = CGPoint(x: 0, y: 0.5)     layer.endPoint = CGPoint(x: 1, y: 0.5)    //设置渐变层各颜色点在颜色线的分布     layer.locations = [0,0.5,1]     //添加视图和层到各自的根中     view.layer.addSublayer(layer)     self.view.addSubview(view)  }  } |

运行项目，运行效果如图10-24所示。



图10-24 颜色渐变

## 10.3 本章小结

本章介绍了UIView视图的基本属性、各属性之间的关系，包括视图的外观、几何、嵌套、层次、变形，简要介绍了UIView视图坐标系的构造。之后还讲述了UIVIew视图与CALayer层之间的关系以及通过对CALayer层的操作实现一定的外观效果。UIView视图是iOS设备应用程序界面的基础，之后的章节如控件、动画等都离不开对UIView的操作。

# 第十一章 基础UI控件

前面一章介绍了UIView的内容。本章介绍常见的基础UI控件，这些UI控件都继承于UIView类，不同控件是对UIView的不同实现。

## 11.1 UILabel

在任意一款应用中，都能见到各种各样的文字标签，Label(标签)是所有控件中最基础也是使用最多的一个控件，其主要作用就是在屏幕中显示一行或者是多行的文字。在UIKit框架中，为开发者提供了很多自定义UILabel属性设置。

### 11.1.1 UILabel的创建

在第十章中，我们已经了解过了如何在Xcode中建立一个iOS项目，并且写出一个可运行的可视化页面，这一小节使用代码在视图中添加一个标签。打开Xcode开发工具，创建一个名为UILabelTest,在其工程目录下，找到名为ViewController.swift的文件，对内部的viewDidLoad方法添加如下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | override func viewDidLoad(){  super.viewDidLoad()  let label = UILabel(frame:CGRect(x:20,y:50,width:300,height:30))      label.text = "hello world"      self.view.addSubview(label)  } |

在上面的代码中，通过UILabel(frame:)构造方法，并且传入一个CGRect类型的结构体，这个结构体的参数设置了在屏幕中需要显示的UILabel控件的位置和尺寸，CGRect确定在iOS的UI系统中绘制的矩形区域，其中的四个参数依次是这个矩形区域的x坐标、Y坐标、宽度和高度。

注意：在UIKit框架中，坐标系是以左上角为坐标原点，向右边为X轴的正方向，向下方是Y轴的正方向。

UILabel中的text属性是用于设置UILabel所要显示的文字，必须要设置为一个String类的字符串值。最后需要调用UIView类的addSubview方法后，将UILabel控件添加到当前的视图上，运行效果如图11-1所示。



图11-1

### 11.1.2 控件的相关属性

在上面的程序中，Label是系统默认的一种样式。UILabel中还为开发者提供了大量的自定义属性，代码如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | override func viewDidLoad(){  super.viewDidLoad()  let label = UILabel(frame:CGRect(x:20,y:50,width:300,height:30))  label.text = "hello world"  label.backgroundColor = UIColor.yellow  label.font = UIFont.systemFont(ofSize:23)  label.textColor = UIColor.red  label.textAlignment = .center  self.view.addSubview(label)  } |

backgroundColor属性设置了标签的背景颜色；font属性设置UILabel控件上的字体相关属性；textColor属性设置UILabel控件上字体的颜色；TextAlignment属性设置UILabel控件上文字的方式。通过添加以上代码，再次运行程序，运行效果如图11-2所示。



图11-2

### 11.1.3 UILabel的多行显示

如果所创建的UILabel中的文字长度超过了其所设的宽度，默认的UILabel控件是不会换行，用省略号代替超出的部分。

|  |  |
| --- | --- |
| 1  2 | let label = UILabel(frame:CGRect(x:20,y:50,width:280,height:40))  label.text = "hello world, hello world , hello world , hello world " |

运行效果如下图11-3所示，超出的文字被省略号代替了。



图11-3

默认的UILabel控件都是单行显示的，可以通过修改一下代码，设置显示行数。

|  |  |
| --- | --- |
| 1 | label.numberOfLines = 0 |

在未修改这个属性值时，默认为1，只能显示一行。如果修改为0，则代表无限换行，直到到底控件最低端。注意UILabel必须要设置的高度要足以容纳这两个每行的文字，否则即使设置了numberOfLines也无法起作用。

## 11.2 UIButton

UIButton也是使用最多的控件之一，可以监听用户在屏幕视图上的多种手势操作，起到一种展示和交互的作用。

### 11.2.1 UIButton的创建

在xcode中创建一个名为UIButtonTest的工程，在ViewController.m的viewDidLoad方法中添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | override func viewDidLoad() {          super.viewDidLoad()          let button = UIButton(type:.system)          button.frame = CGRect(x:40,y:100,width:240,height:30)          button.backgroundColor = UIColor.red          button.setTitle("点击一下",for:.normal)          button.addTarget(self,action:#selector(changeColor),for:.touchUpInside)          self.view.addSubview(button)      }      @objc  func changeColor(){          self.view.backgroundColor = UIColor.yellow      }  } |

首先初始化一个button，控件风格是选择buttonWithType类型的枚举参数进行初始化，buttonWithType有四种枚举类型，代码如下：

|  |
| --- |
| Public enum UIButtonType:Int{  case custom //自定义类型  case system //系统类型  case detailDisclosure //详情按钮类型  case contactAdd //添加按钮类型  } |

UIButton也是继承UIView类，所以控件同样也包括背景色、字体颜色、单击状态等。custom风格是将这些属性设置成默认值，让开发者自定义这些属性风格，system风格是系统内部定义好的一组属性风格，detailDisclosure风格是左边会出现一个详情小图标，contactAdd风格则是在按钮左边显示一个添加小图标。

接下来，介绍button的setTitle方法，setTitle方法有两个参数，第一个是设置了按钮的标题文字，第二个参数则是设置显示此标题文字是的按钮状态。normal为正常状态,即无任何特效。

UIButton控件的核心功能是进行用户交互，可以通过addTarget方法添加触发动作，方法中有三个参数，第一个参数为执行此触发方法的对象，如果是类本身就是self；第二个参数触发动作时要执行的方法；第三个参数为触发方法的条件（必须是UIControlEvents类型数据）。运行效果如图11-4和图11-5所示。

图11-4 图11-5

## 11.3 UITextField

UITextField是一个相对比较复杂的控件，用于处理用户输入的文本，只能单行，不能回车换行。在使用文本框输入文本时，会弹出键盘让用户输入信息，并将输入的信息存储到数据库中，所以本节还要介绍一些代理和键盘相关的知识。

### 11.3.1 UITextField的创建

创建一个名为UITextFieldTest的工程，然后在ViewController.m的viewDidLoad方法中添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | override func viewDidLoad(){  super.viewDidLoad()  let txtField = UITextField(frame: CGRect(x:10,y:100,width:300,height:40))  txtField.borderStyle = .line  txtField.placeholder = "请输入文字"  self.view.addSubview(txtField)  } |

以上代码能够生成一个输入框，UITextField的borderStyle属性用于设置输入框的界面风格，borderStyle有四种枚举类型，代码如下：

|  |
| --- |
| Public enum UITextBorderStyle:Int{  case none //无风格  case line //线性风格  case bezel //bezel风格  case roundedRect //边框风格  } |

UITextField的placeholder属性用来设置提示输入文字，在输入框输入的内容为空时才会显示出来，相反输入框有文字时提示输入文字自动隐藏，运行程序，运行效果如图11-6所示。

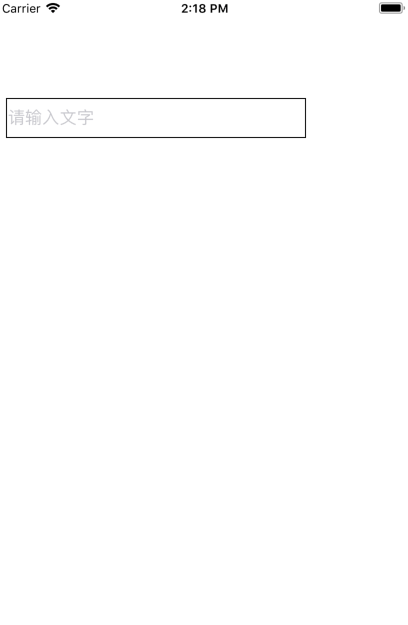


图11-6

### 11.3.2 UITextField的代理方法

UITextField代理其实就是当用户在输入框中输入一个字符时，UITextField这个控件无法判断这个字符是否有效，需要通过对应的代理方法，并在其方法中做逻辑处理。UITextFieldDelegate支持的代理方法如下：

|  |
| --- |
| //输入框将要进入编辑模式时系统自动回调的方法  optional public func textFieldShouldBeginEditing(\_ textField:UITextField)->Bool  //输入框已经进入编辑模式时系统自动回调的方法  optional public func textFieldDidBeginEditing(\_ textField:UITextField)  //输入框将要结束编辑模式时系统自动回调的方法  optional public func textFieldShouldEndEditing(\_ textField:UITextField)->Bool  //输入框已经结束编辑模式时系统自动回调的方法  optional public func textFieldDidEndEditing(\_ textField:UITextField)  //输入框中的内容将要改变时系统自动回调的方法  optional public func textField(\_ textField:UITextField,shouldChangeCharactersIn range:  NSRange,replacementString string:String)->Bool  //输入框中的内容将要被清除时系统自动回调的方法  optional public func textFieldShouldClear(\_ textField:UITextField)->Bool  //用户按键盘上的return键后系统自动回调的方法  optional public func textFieldShouldReturn(\_ textFeild:UITextField)->Bool |

以上方法其上都有对应的注释介绍其功能，可以看到UITextField所有代理方法都是optional属性，即可以在使用中选择性的实现。

有了以上的相关知识，下面实现一个监听文本框程序。首先要遵守相关代理协议，就是让ViewController继承UITextFieldDelegate，代码如下：

|  |  |
| --- | --- |
| 1 | class ViewController: UIViewController,UITextFieldDelegate |

并且在原有的viewDidLoad方法中指定代理在本类中实现，添加代码如下：

|  |  |
| --- | --- |
| 1 | txtField.delegate = self |

最后在ViewController类中实现textField方法，并设置简单的不为空逻辑，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public func textField(\_ textField: UITextField, shouldChangeCharactersIn range: NSRange, replacementString string: String) -> Bool {          if(string.isEmpty){              print("输入框不能为空，请输入文字")          }          return true    } |

当在文本框中输入文字后，又删除了文字，控制台就会打印出“输入框不能为空，请输入文字”。运行效果如图11-7所示。

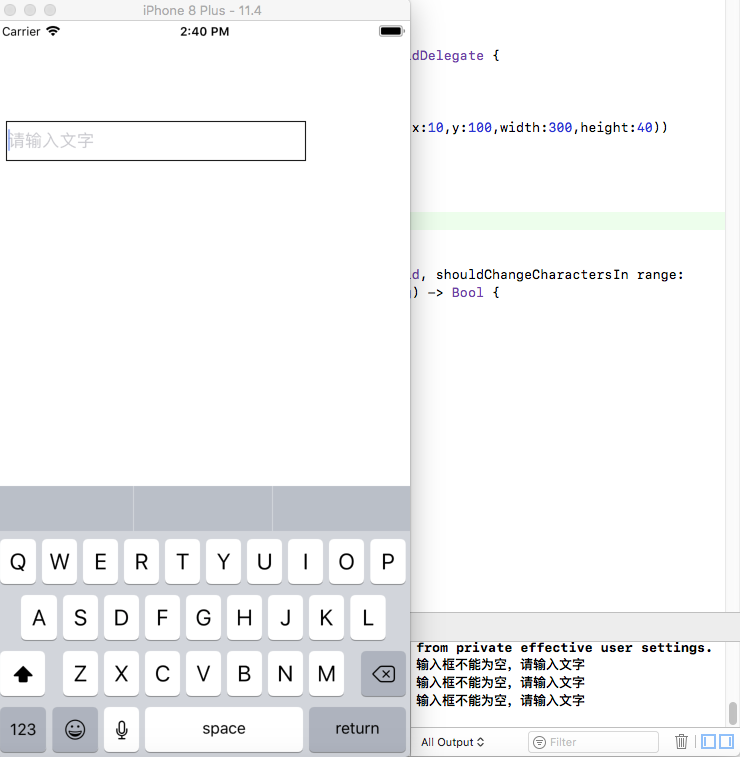


图11-7

读者也可以在这个函数中自行去实现一些更为复杂的效果，这里就不再赘述了。

注意在文本框输入文本时，默认就会弹出虚拟键盘。如果单击文本框没有出现键盘，可能用户调整了设置，使用计算机键盘作为外接键盘，不再弹出虚拟键盘。可以在模拟器的菜单栏找到Hardware->Keyboard，取消勾选Connect Hardware Keyboard即可。

## 11.4 UISwitch

开关控件主要是用来做Bool值的切换，开发中经常需要用到功能的启动或关闭。所以这个控件的使用率也非常高。

### 11.4.1 UISwitch的创建

使用xcode创建一个名为UISwitchTest的工程，在ViewController类中添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | override func viewDidLoad(){  super.viewDidLoad()  let switchView = UISwitch(frame:CGRect(x:20,y:100,width:60,hei ght:30))      switchView.setOn(true,animated:true)      switchView.onTintColor = UIColor.blue      switchView.tintColor = UIColor.red      switchView.thumbTintColor = UIColor.yellow      self.view.addSubview(switchView)  } |

UISwitch功能相对比较简单，所以能设置的属性也对象较少。setOn方法是初始化时设置的开关控件是开还是关，如果是true的话，初始是开，反之为关。onTintColor属性是设置开启状态时控件填充的颜色。tintColor属性是设置关闭状态时控件填充的颜色。thumbTi- -ntColor属性设置开关部分的颜色。运行程序，运行效果如图11-8所示。

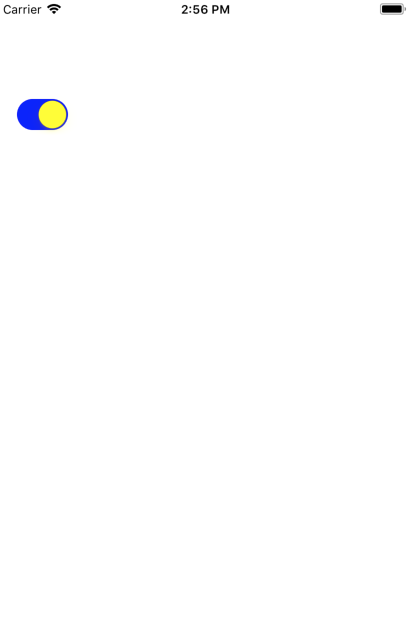


图11-8

### 11.4.2 为控件添加事件

UISwitch同UIButton一样，同属于用户交互控件。可以为其添加交互方法来处理开关逻辑，代码如下：

|  |  |
| --- | --- |
| 1 | switchView.addTarget(self,action:#selector(statePrint),for:.touchUpInside) |

在原有的viewDidLoad中添加addTarget触发函数，action告诉程序这个动作由statePrint函数来实现。方法函数如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @objc private func statePrint(\_ switchView:UISwitch){  if(switchView.isOn){  print("开关开启")  }else{  print("开关关闭")  }  } |

由于本节设置的控件为全局变量，所以在addTarget方法中不需要指明传入的控件参数，编译器自动识别。方法中isOn是switchView的一个属性，可以通过这个属性来判断UISwitch控件的开关状态，这里只是输出了一些文字。运行效果如图11-9所示。

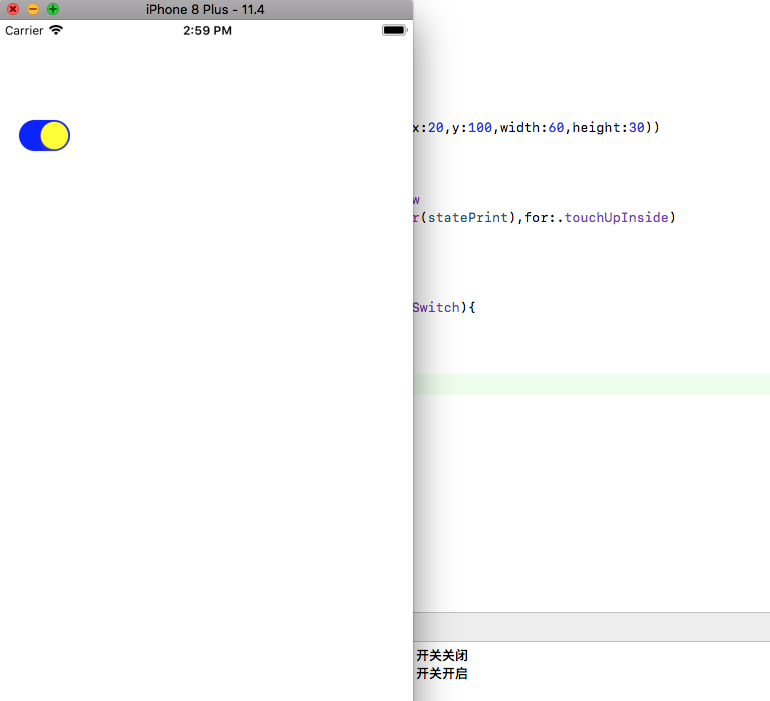


图11-9

## 11.5 UIStepper

UIStepper为步进控件，可以对其设定一个进度，通过“+”、“-”两个按钮来实现属性Value的递增或递减。故在一些电商购物车中的商品数量中经常会使用到这个控件。

使用xcode创建一个名为UIStepperTest的工程，在ViewController类中添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | override func viewDidLoad() {  super.viewDidLoad()  let stepperV = UIStepper(frame: CGRect(x: 20, y: 50, width: 60, height: 30))  stepperV.minimumValue = 0  stepperV.maximumValue = 10  stepperV.stepValue = 1  stepperV.autorepeat = true  stepperV.wraps = true  stepperV.tintColor = UIColor.red  stepperV.addTarget(self, action: #selector(stepper\_Click), for: .valueChanged)  self.view.addSubview(stepperV)  }  @objc private func stepper\_Click(stepperV:UIStepper){  print(Int(stepperV.value))  } |

UIStepper控件mininumValue属性是设置步进控制器的最小值，maximunValue属性是设置步进控制器的最大值。stepValue属性是用来设置步进控制器的步长，即每次按下按钮步进控制器按钮大小改变多少。Autorepeat属性时用来判断如果按住步进控制器中的按钮不放，步进控制器的值是否会一直连续改变。如果为假，要直到用户手指抬起完成单击动作，步进控制器的值才会改变。wraps属性设置步进控制器的值是否循环，如代码所示每次增加到最大值的时候，再次增加一个步进长度就会重新回到最小值。相反，如果把这个属性的布尔值设置成假，那么当步进到达最大值或最小值时，相对应的按钮就会被禁用。tintColor属性是设置控件的颜色。最后通过addTarget方法为步进控制器添加触发事件。触发事件方法中输出stpperV.value的默认值数据类型为Double类型，可以通过Int()来转换成整型。控件及其运行效果如图11-10所示。

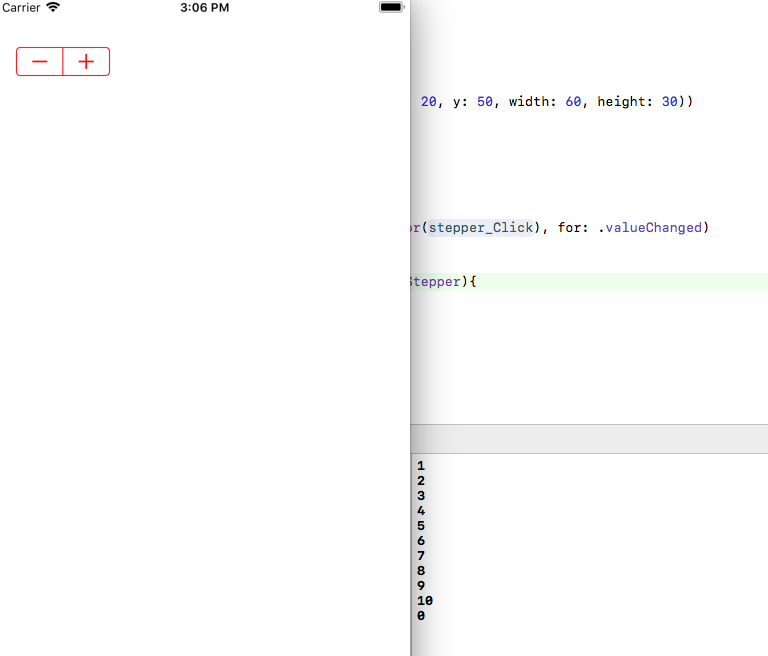


图11-10

## 11.6 UIPageControl

分页视图是一种相当流行的的界面设计模式，常常可以在新手引导页和广告轮播页中使用到。UIPageControl是用于页码管理的一个UI控件，其中间是一行圆点，高亮的一个圆点标记当前所在的页码。

使用xcode创建一个名为UIPageControlTest的工程，在ViewController类中添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | override func viewDidLoad() {  super.viewDidLoad()  let uipage = UIPageControl(frame: CGRect(x: 20, y: 50, width: 300, height: 30))  uipage.pageIndicatorTintColor = UIColor.black  uipage.currentPageIndicatorTintColor = UIColor.yellow  uipage.currentPage = 0  uipage.numberOfPages = 8  uipage.addTarget(self, action: #selector(CurrentPage), for: .valueChanged)  self.view.addSubview(uipage)  }  @objc private func CurrentPage(page:UIPageControl){  print(page.currentPage)  }  } |

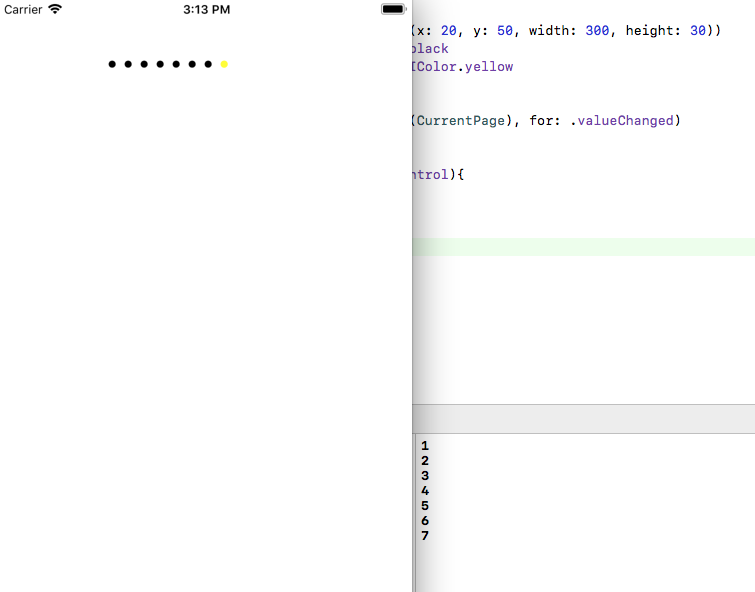
由于UIPageControl控件默认背景圆点是白色的，需要通过设置pageIndicatorTintColor这个属性将所有圆点设置成黑色，这样便于效果演示。currentPageIndicatorTintColor属性是用来设置高亮页点的颜色，currentPage是设置最初的页面停留在哪里。numberOfPages属性用来设置总共有多少页。点击UIPageControl控件的左半边时，页码会向左移动，相反点击右半边，页码会向右移动。在点击的过程中会触发交互方法，会打印出当前的页码。运行效果如图11-11所示。

图11-11

## 11.7 UISegmentedControl

分段控件是由几个segment共同组成，相当于集成了多个button，可以通过点击不同的segment来切换不同的界面。一般用于导航栏的标题视图中，小巧且使用方便。

### 11.7.1 UISegmentedControl的创建

使用xcode创建一个名为UISegmentedControlTest的工程，在ViewController类中添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | override func viewDidLoad() {  super.viewDidLoad()  let IArray:[String] = ["家具","灯饰","建材","装饰"]  let seg = UISegmentedControl(items: IArray)  seg.frame = CGRect(x: 20, y: 100, width: 200, height: 30)  seg.isMomentary = true  seg.tintColor = UIColor.red  seg.selectedSegmentIndex = 1  self.view.addSubview(seg)  }  } |

UISegmentedControl控件的创建方法与其他的控件会有点不同，在初始化此控件时，必须先需要先传入一个数组，来初始化列表字段,然后通过控件的frame属性来设置坐标和大小。isMomentary属性默认为假，设置控件为切换按钮模式，如果设置为真，控件就为触发按钮控件。tintColor属性是用来设置控件所使用的颜色，默认是蓝色。selectedSegmentIndex会默认选中下标为1的标题（UISegmentedControl的下标是从0开始的）。运行效果如图11-12所示。



图11-12

### 11.7.2 UISegmentedControl的增、删、改

UISegmentedControl在初始化后，经常还需要进行动态的增加、删除和修改等操作，UISegmentedControl提供了以下三种方法给开发者使用，在原代码中如下：

|  |  |
| --- | --- |
| 1  2  3 | seg.setTitle("空调", forSegmentAt: 2)  seg.insertSegment(withTitle: "油漆", at: 3, animated: true)  seg.removeSegment(at: 0, animated: true) |

UISegmentedControl中的setTitle方法是重新设置一个按钮标题，即原先第三个按钮的标题会被改成“空调”。insertSegment方法是在某个一索引中插入一个标题，第三个参数是用来是否设置插入动画，由于插入的速度可能太快，肉眼无法观察到，可以通过后面几章的深入学习控件的动画。removeSegment方法是用来删除指定某一个标题。运行效果如图11-13所示。



图11-13

## 11.8 UIDatePicker

UIDatePicker为日期时间选择器，一般可以通过滑动选择日期、时间和持续时长的输入，只需要配置指定风格，而不需要关心底层是如何操作的，在一些旅游网站经常会出现。

使用xcode创建一个名为UIDatePickerTest的工程，在ViewController类中添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | override func viewDidLoad() {     super.viewDidLoad()     let datePicker = UIDatePicker(frame:CGRect(x: 0, y: 0, width:  UIScreen.main.bounds.width, height: 200))     datePicker.datePickerMode = .date     datePicker.backgroundColor = UIColor.gray      datePicker.addTarget(self, action: #selector(ShowTime), for: .valueChanged)      self.view.addSubview(datePicker)    }    @objc private func ShowTime(datePicker:UIDatePicker){     print(datePicker.date)    }  } |

先创建宽度和屏幕设备相同的UIDatePicker控件，屏幕宽度是通过UIScreen类获取的。UIDatePicker控件的datePickerMode属性是用来设置控件的风格。有四个枚举选项：

|  |
| --- |
| Public enum UIDatePickerMode:Int{  case time //显示时间（当地时间）  case date //显示日期（当地日期）  case dateAndTime //显示日期和时间  case countDownTimer //显示小时和分钟  } |

以上对应的控件风格如图11-14到图11-17所示。

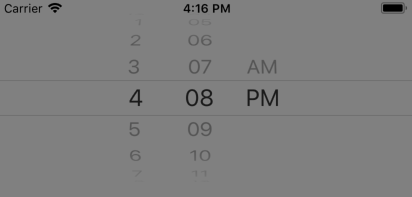
 

图11-14 date模式 图11-15 time模式

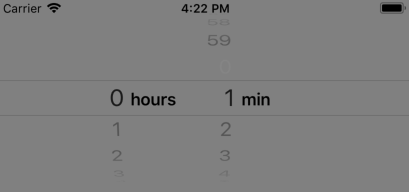
 

图11-16 dateAndTime模式 图11-17 countDownTimer模式

再为其添加的触发方法中，只有每一次选中一个时间不变时，才会触发方法。运行效果如图11-18所示。

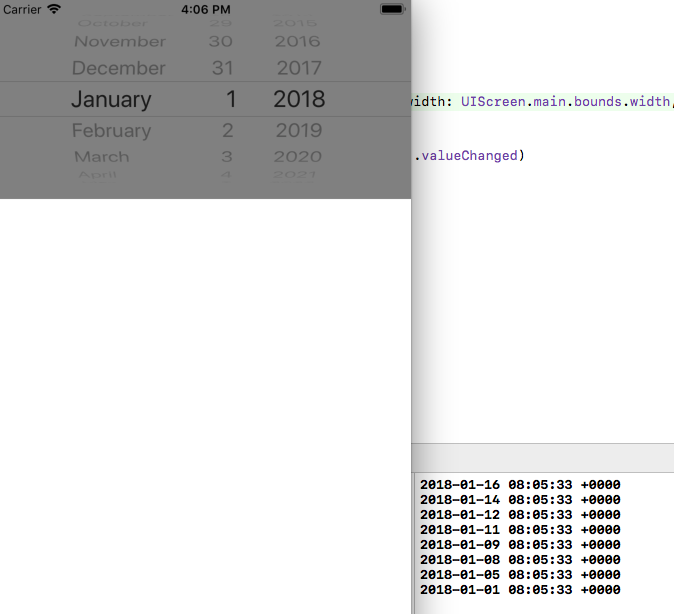


图11-18

## 11.9 UIScrollView

UIScrollView是可滚动视图，允许一个屏幕上显示多个视图内容，没有区域限制，可通过滚动条来查看屏幕外的界面，在新闻类等多种应用中广泛的使用。

使用xcode创建一个名为UIScrollViewTest的工程，在ViewController类中添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | override func viewDidLoad() {  super.viewDidLoad()  let scrollView = UIScrollView(frame: CGRect(x: 40, y: 60, width: 300, height: 400))  scrollView.backgroundColor = UIColor.black  let view1 = UIView(frame: CGRect(x: 0, y: 0, width: 300, height: 400))  view1.backgroundColor = UIColor.red  scrollView.addSubview(view1)  let view2 = UIView(frame: CGRect(x: 0, y: 400, width: 300, height: 400))  view2.backgroundColor = UIColor.green  scrollView.addSubview(view2)  let view3 = UIView(frame: CGRect(x: 0, y: 800, width: 300, height: 400))  view3.backgroundColor = UIColor.blue  scrollView.addSubview(view3)  scrollView.contentSize = CGSize(width:300,height:400\*3)  scrollView.showsVerticalScrollIndicator = true  scrollView.isPagingEnabled = true  scrollView.indicatorStyle = UIScrollViewIndicatorStyle.default  self.view.addSubview(scrollView)  } |

想要滚动视图控件达到滚动的效果，放入的所有视图长或高相加，必须要大于滚动视图控件的长或高。UIScrollView的contentSize属性是用来设置滚动视图所能容纳内容大小的，内容区域决定可滚动的范围。showsVerticalScrollIndicator属性是用来设置滚动时是否显示水平提示条，同理showsVerticalScrollIndicator属性是用来设置滚动时是否显示垂直提示条。isPagingEnabled属性时用来设置是否翻页滚动，默认是滑动多少，滚动多少。indicatorStyle属性是用来设置滚动提示条的风格，系统一共提供了3种风格的滚动条，具体如下：

|  |
| --- |
| Public enum UIScrollViewIndicatorStyle:Int{  case 'default' //默认风格  case black //黑色风格  case white //白色风格  } |

默认风格会在白边界上绘制黑色的滚动条，大多数背景下都适用。黑色风格就是黑色滚动条，用在白色背景下，而白色风格就是白色滚动条，用在黑色背景下。运行效果如图11-19所示。

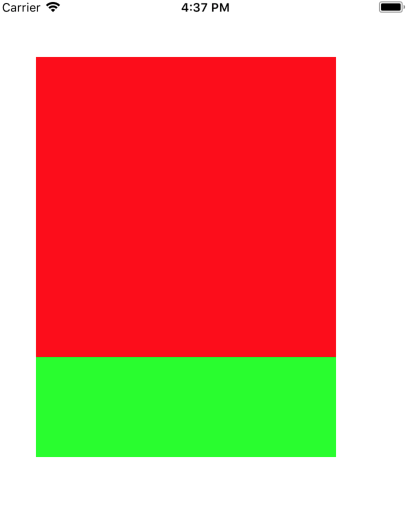


图11-19

## 11.10 本章小结

通过本章的学习，读者需要掌握基础UI控件的使用，以及其相应的属性定义。对于一些可交互性控件，需要掌握如何添加并编写触发方法。并且有几个控件是有前后台数据交互的，需要掌握数据交互时所利用的代理功能。

# 第十二章 视图控制器（UIViewController）

## 12.1 视图控制器（UIViewController）介绍

视图控制器（UIViewController）在iOS开发中至关重要，每个App都要有至少一个视图控制器。UIViewController对内管理与之关联的UIView，对外跟其他UIViewController通信和协调。用户在屏幕上看到的是UIView对象，而在UIView对象的背后是UIViewController在起着管理的作用。

在实际开发过程中，UIViewController的作用主要体现在以下几个方面。

1、视图控制器的属性设置。如背景颜色，适配，视图控制器数组属性等。

2、视图控制器的生命周期的控制。

3、视图控制器间的转场present，或push，以及相对应的dismiss，或pop。

### 12.1.1 视图控制器的生命周期

所谓ViewController的生命周期，其实指的是它控制的视图（View）的生命周期。每当视图的状态发生变化时，视图控制器会自动调用一系列方法来响应变化。  
 通过这些方法，开发者们就可以跟踪到视图的整个生命周期。各个方法按执行顺序排列如下：

|  |  |
| --- | --- |
| 生命周期 | 说明 |
| alloc | 创建一个视图控制器对象，并分配内存空间。 |
| init() | 对视图控制器对象进行初始化。 |
| loadView | 如果从storyboard创建视图，则从storyboard中加载视图。 |
| viewDidLoad | 视图加入完成，可以进行一些自定义操作。 |
| viewWillAppear | 视图即将要展示在屏幕上。 |
| viewDidAppear | 视图已经站在屏幕上显示并完成渲染。 |
| viewWillLayoutSubviews | 视图即将布局其子视图 |
| viewDidLayoutSubviews | 视图已经完成子视图的布局 |
| viewWillDisappear | 视图即将从屏幕中消失 |
| viewDidDisappear | 视图已经从屏幕上消失 |
| dealloc | 视图被销毁 |

同时系统还会在内存不足的时候调用didReceiveMemoryWarning 方法通知视图控制器，我们可以在这里面进行一些操作，来释放一些额外的资源。

实际上，视图一般不会按照上表的流程依次执行下来，极有可能在不同的状态间相互转换。例如，在视图可见和不可见的状态间相互转换，我们从当前视图切换到另一个视图，当前视图就从可见转换为不可见，当再次切换回来，视图再次可见。

生命周期之间的转换运行效果如图12-1所示。

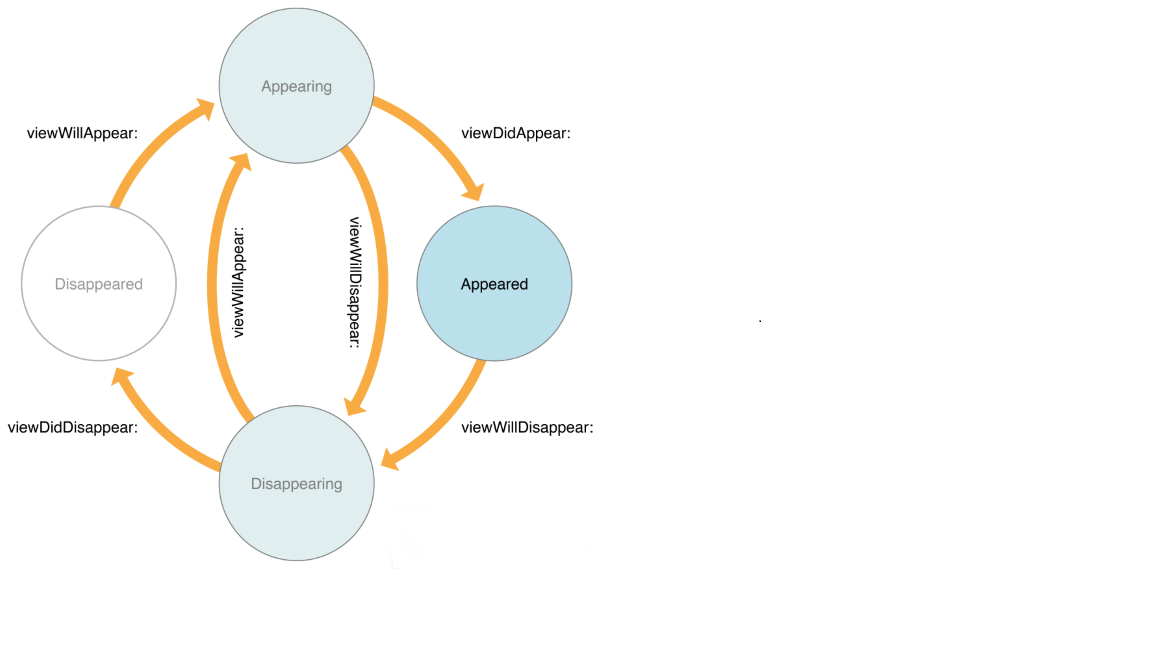


图12-1

### 12.1.2 创建视图控制器

可以使用代码或通过Storyboard创建一个视图控制器。本节通过一个实例【例12-1】，介绍视图控制器的创建和使用。

从模板选择窗口中创建【Single View Application】，取名为【Demo12-1】。

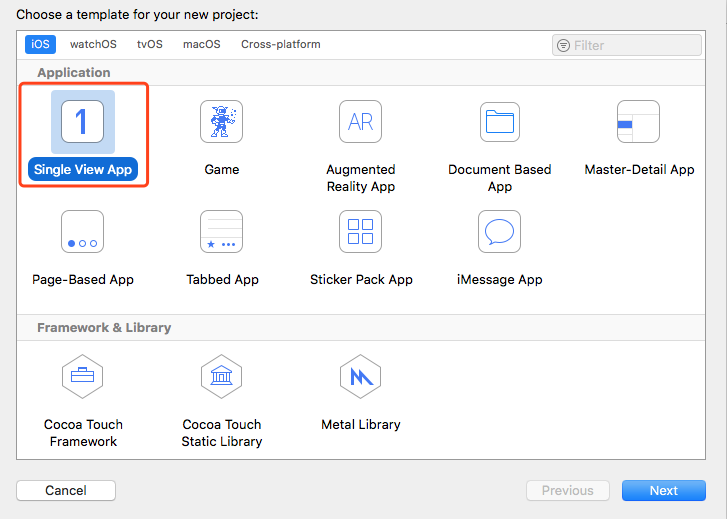


图 12-2

根据模板创建项目时，系统会自动在Storyboard中创建一个视图控制器，默认名称为ViewController，同时创建与之对应的ViewController.swift文件。ViewController默认管理着一个view，被称作根视图。如图12-3所示。

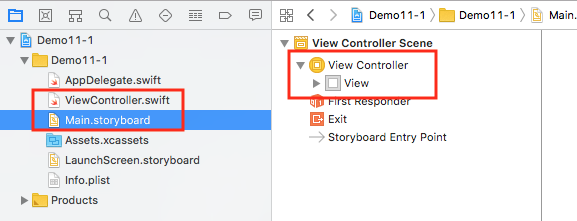


图12-3

1、打开ViewController.swift文件，在viewDidLoad方法中，添加一句代码，对视图控制器的背景颜色做一下修改。如图12-4所示。



图12-4

代码中，self是ViewController本身，通过self.view拿到ViewController管理的根视图，再将backgroundColor设置为棕色。

2、点击【编译运行】按钮或是使用快捷键”command + r”，开启虚拟机，运行程序。如图12-5所示。

屏幕快照 2018-09-15 下午12.48.05

图12-5

运行结果如图12-6所示，背景变成棕色。

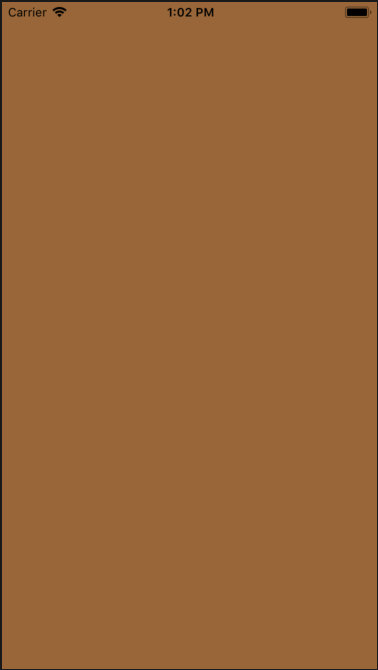


图12-6

这样，就完成了一个最简单的项目。总结起来就是首先创建了一个【Single View Applic- -ation】，然后在默认的ViewController中修改了view的背景颜色，并通过虚拟机运行展示。

在现有的代码基础上，创建一个新的视图控制器。右击项目文件夹，选择【New File】,创建文件。如图12-7所示。

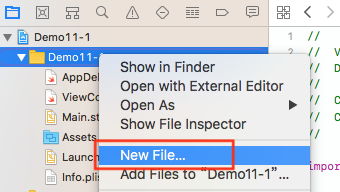


图12-7

在弹出的窗口中，选择【Cocoa Touch Class】选项，如图12-8所示：

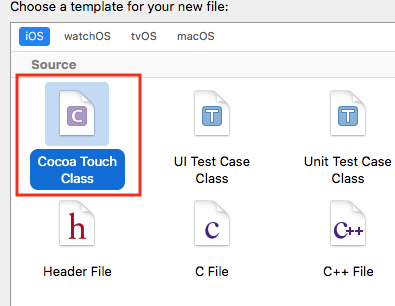


图 12-8

下一步，在创建窗口中的Class输入框中输入MyViewController，表示创建的类的类名为MyViewController，并设置Subclass of为UIViewController，表示MyViewController继承自UIViewController，语言选择swift。如图12-9所示：

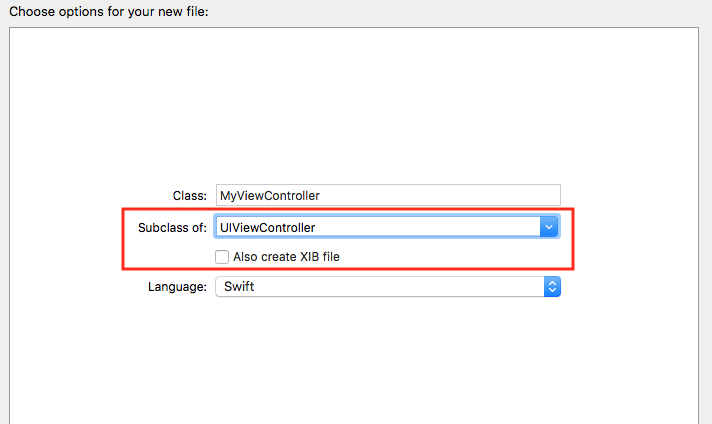


图12-9

同样，在MyViewController.swift文件中，设置背景颜色为绿色，即UIColor.green（这里的代码和设置棕色背景的代码一样不再重复）。

最后一步，将Storyboard中的设计界面和MyViewController相关联。这里有两种方式，第一种方式是在Storyboard中设置关联的类文件，第二种方式是在AppDelegate中设置rootViewController，接下来依次介绍这两种方法。

第一种方式，打开Main.storyboard，选择ViewController，然后点击右侧的【Show the indentity inspector】选项卡，接着修改【Class】选择框内，输入或选择刚刚创建好的MyViewController。如图12-10所示。

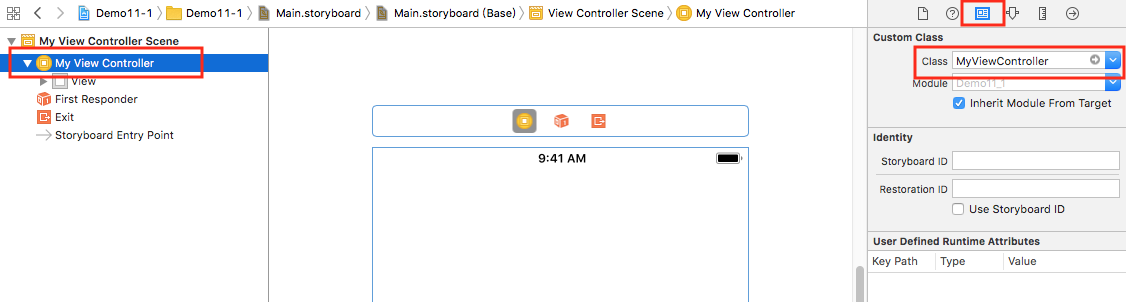


图12-10

完成上述操作后，点击【编译运行】，可以看到效果，页面变成绿色。

更改项目默认视图控制器的第二种方式是通过修改ApoDelegate.swift文件。首先，将Main.storyboard中ViewController的【Class】还原为ViewController，然后从文件列表中打开AppDelegate.swift。如图12-11所示。

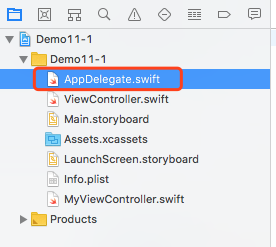


图12-11

在application方法中，添加如下代码。如图12-12所示。



图12-12

再次运行程序能看到和方法一一样的效果，页面背景颜色变成绿色。

### 12.1.3 实现视图控制器间的切换

实际上，应用一般不会只有一个视图，经常会切换视图，例如，在手机上阅读图书时翻页，这时就要用到视图切换。本节中，将通过【例12-2】代码的方式实现一个简单的视图切换，并在视图切换中传递一些数据。

第一步，创建一个【Single View Application】,取名为【Demo12-2】。

第二步，编辑ViewController.swift文件。在该视图控制器控制的视图中，添加一个UILabel用于显示另一个视图控制器返回的数据，再添加一个UIButton控制视图切换。具体代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import UIKit  class ViewController: UIViewController {      var label:UILabel!      override func viewDidLoad() {          super.viewDidLoad()          self.view.backgroundColor = UIColor.darkGray          //创建label          label = UILabel(frame: CGRect(x: 50, y: 100, width: 250, height: 50))        label.text = ""          self.view.addSubview(label          let button = UIButton(frame: CGRect(x: 50, y: 200, width: 250, height: 50))          button.setTitle("跳转", for: UIControlState())          button.backgroundColor = UIColor.orange    button.addTarget(self,action:#selector(ViewController.openSecondCon troller), for: .touchUpInside)          self.view.addSubview(button)      }      @objc func openSecondController(){          let secondViewController = SecondViewController()          secondViewController.viewController = self          secondViewController.text = "传递的参数"          self.present(secondViewController, animated: true, completion: nil)      }      override func didReceiveMemoryWarning() {          super.didReceiveMemoryWarning()      }  } |

第4行，声明UILabel，它将被用于显示另一个视图控制器返回的数据。（行数有变）

第7行，设置了根视图背景颜色为深灰色。

在代码的9-11行中，创建UILabel，默认文字text设置为空，并通过addSubview方法将其添加到ViewController的根视图中。

接着在13-17行中，创建了UIButton按钮对象，通过点击它执行控制器跳转，然后先后设置按钮的标题和背景颜色。在第16行代码中，通过addTarget方法给按钮绑定点击事件，当按钮被点击使，调用openSecondController方法，最后将按钮添加到根视图中去。

代码19-24行，创建了一个名为openSecondController的方法，用来响应按钮的点击事件。第20行，创建了SecondViewController类的实例，这个控制器就是即将跳转的控制器，稍后会有这个类的实现。第21-22行设置secondViewController的viewController属性为self，即当前的视图控制器，设置text为“传递的参数”。最后，调用当前视图控制器的present(newViewController:, animated:, completion:)方法，以模态的方式跳转到新的视图控制器。该方法的第一个参数表示需要打开的新的视图控制器，这里设置为我们刚刚创建的secondViewController，第二个参数表示是否以动画的方式打开视图控制器，第三个参数表示新的视图控制器被打开后的回调方法，本例中不需要所以设置为nil。

接下来通过【New File】创建一个新的视图控制器的子类，取名为SecondViewController。SecondViewController的具体代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import UIKit  class SecondViewController: UIViewController {      var viewController: ViewController?      var text = ""      override func viewDidLoad() {          super.viewDidLoad()          self.view.backgroundColor = UIColor.brown          let label = UILabel(frame: CGRect(x: 50, y: 100, width: 250, height: 50))          label.text = text          self.view.addSubview(label)          let button = UIButton(frame: CGRect(x: 50, y: 200, width: 250, height: 50))          button.setTitle("跳转回去", for: UIControlState())          button.backgroundColor = UIColor.darkGray          button.addTarget(self, action: #selector(self.dismissSelf), for: .touch UpInside)          self.view.addSubview(button)      }      @objc func dismissSelf(){          viewController?.label.text = "返回的参数"          self.dismiss(animated: true, completion: nil)      }  } |

代码4-5行，给SecondViewController声明了两个属性，一个是viewController用于表示弹出当前视图控制器的源视图控制器；另个一字符串text，用于保存从源视图控制器传递过来的参数。

在viewDidLoad方法中，为了区分两个视图控制器，将SecondViewController的背景颜色设置为棕色。

在代码9-17行，首先创建一个UILabel，它用于显示从源试图控制器传来的参数，并添加到根视图。然后创建了一个UIButton，设置标题和背景颜色之后，为其绑定点击事件dismissSelf，当用于点击按钮时，将退出当前的试图控制器，并返回源视图控制器，

最后的20-23行代码中，实现dismissSelf方法。在该方法中，首先设置了源视图控制器中的标签对象的文字内容，然后调用本视图控制器对象的dismiss(animated:,completion:)方法，退出当前视图控制器，返回源视图控制器。该方法的第一个参数设置是否开启动画，第二个参数表示视图控制器退出后的回调方法。

最后点击【编译运行】，可在模拟器上运行程序。实际运行效果如图12-13到12-15所示。

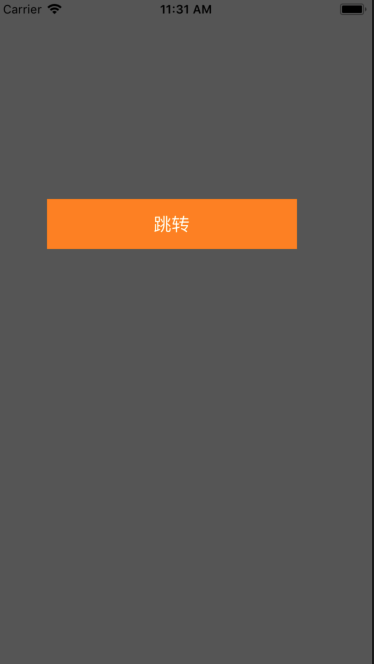
 

图12-13 图12-14

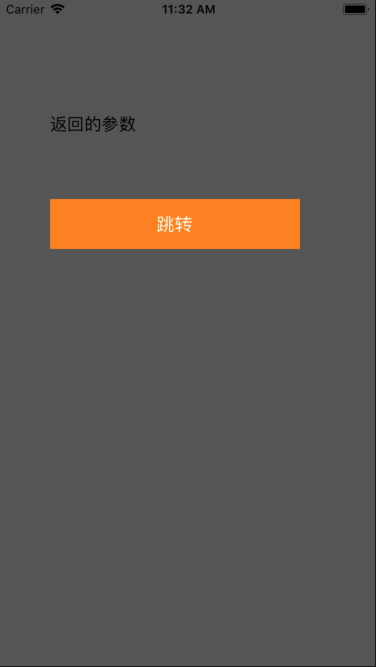


图12-15

进入app，首先看到图12-13，深灰色背景，说明当前是在第一个视图控制器，并且UILabel区域没有文字。点击【跳转】按钮，跳转进入图12-14第二个视图控制器，背景颜色变为棕色，UILable内容显示“传递的参数”，点击【跳转回去】按钮，回到深灰色的源视图控制器，并且UILable区域显示“返回的参数”。这样，我们就完成了一个简单的视图控制器跳转，并且在跳转的过程中传递了一些参数。

## 12.2 标签控制器UITabBarController

当app中有多个视图控制器的时候，就需要对这些视图控制器进行管理。iOS提供了两个特殊的视图控制器——标签栏控制器（UITabBarController）和导航控制器（UINavigationCon troller）去管理其他视图控制器。

UITabBarController是一种常用的视图控制器，例如系统的Health程序，如图12-16所示。

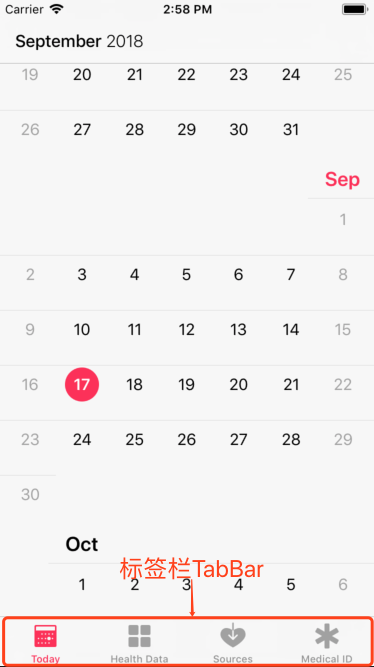


图12-16

UITabBarController通常会被设置为程序的rootViewController，添加到UITarBarController中的只能是别的视图控制器（包括UINavigationController）。UITarBarController主要用来管理用户提供的包含各种内容的子视图控制器，而每一个子视图控制器则负责自己的视图层次关系。

当多个子视图控制器处于相同等级，没有层级关系，那么就应该使用UITabBarViewCon troller。一个UITabBarViewController中可以设置多个Tab，每个Tab对应一个UIViewCon troller，通过点击TabBar上的图标，可以在不同的UIViewController之间切换，显示对应的内容。

### 12.2.1 创建UITabBarController

本小结，通过创建一个包含2个Tab的UITabBarController，来介绍UITabBarController的相关知识。

第一步，创建一个【Single View Application】，命名为【Demo12-3】。然后创建2个视图控制器类，分别命名为FirstViewController和SecondViewController，并编辑FirstViewController.swift文件，在viewDidLoad方法中增加以下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | super.viewDidLoad()  self.view.backgroundColor = UIColor.blue  self.tabBarItem = UITabBarItem(tabBarSystemItem:UITabBarSystemItem.feat ured, tag: 1)  let label = UILabel(frame: CGRect(x: 40, y: 150, width: 250, height: 50))  label.text = "第一页"  self.view.addSubview(label) |

首先在第2行代码，设置了当前视图控制器所管理的view的背景颜色，这里设置为蓝色。

第3行代码，设置tabBarItem，也就是设置每个视图控制器在TabBar中对应的图标和文字。此处，设置为名为featured的系统自带图标，tag参数是一个整型，用于表示这个视图控制器。当然，tabBarItem也能使用自定义的图片和文字，稍后的章节会介绍，本例主要是介绍UITabBarController的用法。

第5-6行代码在当前的view下添加了一个UILabel，添加文字，以便和第二个视图控制器区分开来。

最后，将label添加到视图中。

第二步，在SecondViewController的viewDidLoad方法中，添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | super.viewDidLoad()  self.view.backgroundColor = UIColor.brown  self.tabBarItem = UITabBarItem(tabBarSystemItem:UITabBarSystemItem.book marks, tag: 1)  let label = UILabel(frame: CGRect(x: 40, y: 150, width: 250, height: 50))  label.text = "第二页"  self.view.addSubview(label) |

这段代码和步骤一中的代码基本相同，不同的是第2行，设置背景为棕色，第3行tabBarItem设置为bookmarks样式，以及第6行，label的文字内容是“第二页”，以上的修改都是为了和第一个视图控制器区别开来，读者可以设置自己喜欢的颜色或文字。

第三步，实现UITabBarController。完成自视图控制器的属性设置，我们就可以进入关键的创建UITabBarController的步骤了。首先打开AppDelegate.swift文件，然后在application方法中，添加如下代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let firstViewController = FirstViewController()  let secondViewController = SecondViewController()  let tabViewController = UITabBarController()  tabViewController.viewControllers = [firstViewController, secondViewController]  self.window?.rootViewController = tabViewController  return true |

第1，2行，分别创建之前定义好的两种视图控制器的实例，它们将作为UITabBarCon troller的子视图控制器。

第4行，创建UITabBarController的实例。

第5行，将创建好的视图控制器放在一个数组中，并赋值给tabViewController的viewController属性，这样这两个视图控制器就成为了tabViewController控制的子视图控制器了。

最后第6行，将tabViewController设置为rootViewController。

完成以上三个步骤，点击【编译运行】按钮，运行虚拟机，就可以看到如图12-17和图12-18的效果。点击TabBar上的item实现不同控制器之间的切换。

图 12-17 图12-18

### 12.2.2 改变UITabBarController的索引

在【Demo12-3】中，可以通过点击TabBar中的item图标进行页面的跳转，但是实际使用中经常会遇到通过点击页面上的一个按钮，然后进入相关页面。这种需求可以通过设置UITabBarController实例中的selectedIndex属性来实现。接下来通过【Demo12-4】来实现这个功能。

第一步，创建一个【Single View Application】，命名为【Demo12-4】。然后创建2个视图控制器类，分别命名为FirstViewController，在viewDidLoad方法中添加代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | super.viewDidLoad()  self.view.backgroundColor = UIColor.blue  self.tabBarItem = UITabBarItem(tabBarSystemItem:UITabBarSystemIte m.featured, tag: 0)  let label = UILabel(frame: CGRect(x: 40, y: 150, width: 250, height: 50))  label.text = "第一页"  self.view.addSubview(label)  let button = UIButton(frame: CGRect(x: 40, y: 220, width: 240, height: 44))  button.setTitle("跳转第二页", for: UIControlState())  button.backgroundColor = UIColor.black  button.addTarget(self, action: #selector(FirstViewController.enterPage2), for: .touchUpInside)  self.view.addSubview(button) |

从第1行到第7行的代码和【Demo11-3】中相同，完成设置tabBarItem和label两项工作。

在第9行，创建了一个按钮button，设置大小和位置，第10，11行设置标题和颜色。

第11行，为button绑定点击事件，当按钮被点击调用enterPage2方法。

最后第12行，将button添加到视图中去。

第二步，在FirstViewController中新增一个enterPage2函数。代码如下：

|  |  |
| --- | --- |
| 1  2  3 | @objc func enterPage2(){      self.tabBarController?.selectedIndex=1   } |

这段代码中，将selectedIndex设置为1，从0开始计算，1是第二个视图控制器，所以这代码的作用是控制跳转到第二个视图控制器。

第三步。创建SecondViewController和修改AppDelegate文件，其中代码和【Demo12-3】完全相同，不再重复。

最后四步，点击编译运行按钮启动项目。运行结果如图12-19所示。



图12-19

当点击【跳转第二页】按钮，跳转到第二页，运行效果如图12-20所示。



图12-20

这样就实现了通过在view中点击按钮在UITabBarController中的页面跳转。

### 12.2.3 添加角标

在使用微信，qq等通讯软件的时候，常常会看到TabBar的图标上有红色的数字，代表着未读信息数目。通过设置tabBarItem的badgeValue属性即可达到这种效果。在【Demo11-4】的FirstViewController.swift的viewDidLoad方法中添加一句代码。

|  |  |
| --- | --- |
| 1 | self.tabBarItem.badgeValue = "8" |

注意，这里的“8“不是数字，而是字符串。

点击【编译运行】即可看到如图11-21所示：

11.21  
图12-21

## 12.3 导航控制器UINavigationController

一节，我们学习了UITabBarController的基本使用。之前介绍过UITabBarController所管理的视图控制器之间处于相同等级。但如果需要管理有一些层级关系的视图控制器，我们就需要用到UINavigationController了。UINavigationController使用栈实现，栈具有先进后出的特点，因此UINavigationController适合用于处理和显示分层的数据。

UINavigationController在iOS系统中十分常见。最典型的应用就是【设置】。如图12-22，在【设置】中有很多标签。比如【general】（【通用】），当点击【general】，就会进入下一级菜单，如图12-23所示，当然我们还可以点击按钮进入下一层。这就是典型的分层结构，使用的就是接下来要介绍的UINavigationController。

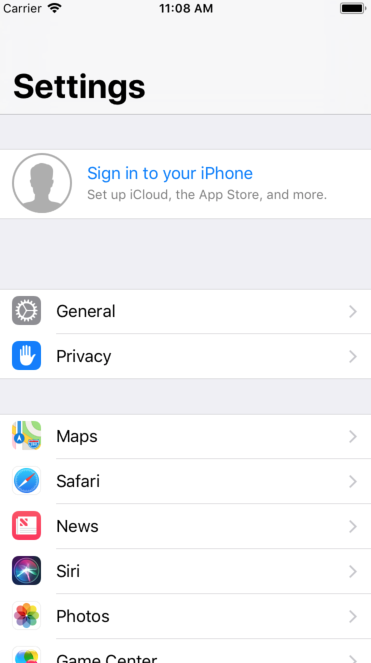
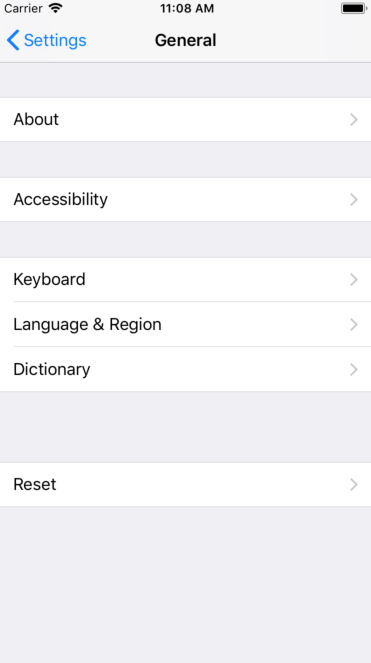
 

图 12-22 图12-23

### 12.3.1 UINavigationController视图嵌套

NavigationController是一个容器视图控制器，它将其他视图控制器的内容嵌入到自身内部。可以从导航控制器的视图属性访问其视图。该视图包含导航栏、可选工具栏和与最顶层视图控制器对应的内容视图。图11-24显示了如何组装这些视图来显示整个导航界面。(在此图中，导航界面进一步嵌入到选项卡栏界面中。)尽管导航栏和工具栏视图的内容发生了变化，但视图本身不会发生变化。实际更改的唯一视图是导航堆栈上最顶层视图控制器提供的自定义内容视图。

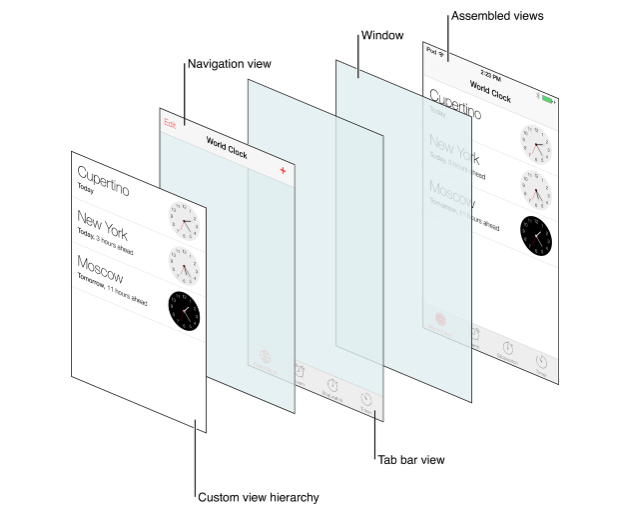


图12-24

### 12.3.2 创建UINavigationController

本节将通过一个实例【Demo12-5】，介绍UINavigationController的创建和使用。

首先，创建一个【Single View Application】，命名为【Demo12-5】。然后创建2个视图控制器类，分别命名为FirstViewController和SecondViewController。这两个视图控制器将作为UINavigationController导航栏控制器的子视图控制器。

接下来编辑FirstViewController.swift和SecondViewController.swift两个文件。在FirstViewController的viewDidLoad方法中添加如下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5 | super.viewDidLoad()  self.title = "第一页"  self.view.backgroundColor = UIColor.blue  self.navigationItem.rightBarButtonItem = UIBarButtonItem(title: "下一页", style: .plain, target: self, action: #selector(FirstViewController.nextPage)) |

第2行设置视图控制器的title，这个标题将显示在导航栏上。第3行设置了view的背景颜色为蓝色。

在第4行中，通过设置当前视图控制器navigationItem的rightBarButtonItem的样式，对顶部导航栏右侧的按钮进行自定义操作。本例中设置为点击按钮调用nextPage方法，跳转至第二页。

接下来编写nextPage方法。在viewDidLoad方法下面添加nextPage方法，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4 | @objc func nextPage(){      let viewController = SecondViewController()     self.navigationController?.pushViewController(viewController, animated: true)   } |

在nextPage方法中，先创建了第二个视图控制器SecondViewController的实例，然后通过self.navigatioonController获取当前视图的导航栏控制器，再通过pushViewController将创建好的SecondViewController的实例压入栈中，第一个参数是压入栈的视图控制器，true表示显示动画。

第二步，编辑SecondViewController,swift，在viewDidLoad方法中设置标题和给view添加背景颜色以便和第一页区分开。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | self.title = "第二页"  self.view.backgroundColor = UIColor.brown  第三步，修改AppDelegate.swift，设置启动的根视图控制器。代码如下：  func application(\_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {  let firstViewController = FirstViewController()  let navgationController = UINavigationController(rootViewController: firstViewController)  self.window?.rootViewController = navgationController  return true  } |

第2行，创建FirstViewController的实例，第3行，创建UINavigationController的实例，同时将第2行创建的FirstViewController的实例作为navgationController 的根视图控制器。最后将navgationController设置为应用程序窗口的根视图控制器，这样app启动，默认调用navgationController。

点击【编译运行】，可看到如图12-25的结果，点击右上角【下一页】按钮，进入第二页，如图12-26所示，此时点击左上角【第一页】，便可返回第一页。

图12-25 图12-26

### 12.3.3 导航栏和工具栏的显示和隐藏

默认状态下，底部的工具栏Tab bar view是隐藏的，我们可以在视图控制器的viewWillAppear方法中，在即将显示时，更改工具栏和导航栏的可见性。

在【Demo12-5】的基础上稍作修改。打开FirstViewController.swift，在该文件中重写viewWillAppear方法。

|  |  |
| --- | --- |
| 1  2  3  4  5 | override func viewWillAppear(\_ animated: Bool) {  super.viewWillAppear(animated)  self.navigationController?.setToolbarHidden(false, animated: false)     self.navigationController?.setNavigationBarHidden(true, animated: true)  } |

第4行设置ToolbarHidden属性为false，表示不隐藏Toolbar。第5行设置Navigation BarHidden属性为true，表示隐藏NavigationBar。

点击【编译运行】可以看到顶部的NavigationBar被隐藏了，而之前没有显示的Toolbar现在显示出来了。如图12-27所示。



图 12-27

### 12.4 本章小结

本章介绍了视图控制器（UIViewController）的一些理论知识，包括基本概念，生命周期，随后用实例的方式介绍了视图控制器的创建和视图控制器间的切换。

随后讲解了常用的UITabBarController标签控制器和UINavigationController导航控制器的使用，鉴于应用程序一般都是多界面的，所以掌握这两个常用的控制器是很有必要的。

# 动画、音频和视频

随着移动互联网的发展，手机越来越偏向娱乐化，播放音乐、视频，用手机拍照都是现代手机不可或缺的功能。而iOS设备对于多媒体的支持非常强大，拥有多套API来支持音频视频的播放、录制，还有对麦克风和摄像头的操作。

本章将要介绍iOS开发中动画的制作，音频以及视频的播放。

## 13.1 动画的制作

动画效果是提高用户体验的关键因素。在iOS系统中，其用户界面有很多细微精美的动画，这使得应用的体验感更具有吸引力，更有动态性。适当的动画效果会是程序在传达状态，提供反馈方面更有体验感，让人们感到其操作更加可视化。

Core Animation和UIKit是iOS中实现动画效果的主要技术。它们提供了丰富、简单的API，使得开发者可以用很简单的代码就实现酷炫的动画效果。UIKit可以看做对CoreAnima tion的封闭。接下来将会对UIKit中的动画进行一些讲解，对Core Animation不再讲解。

### 13.1.1 UIView的动画

在UIKit中，很多API都可以看到animated参数，表示是否动画显示，其实这是UIKit封装CoreAnimation后的结果。UIKit主要API散落在UIView+UIViewAnimationWithBlocks和UIView+UIViewKeyframeAnimations两个分类..使用UIView动画可以通过修改以下UIView视图来产生动画效果：

a、大小变化(frame)  
 b、拉伸变化(bounds)  
 c、中心位置(center)  
 d、旋转(transform)  
 e、透明度(alpha)  
 f、背景颜色(backgroundColor)  
 g、拉伸内容(contentStretch)

UIKit可以将动画集成到UIView类中，实现简单动画的创建过程。UIView类中定义了几个内在支持动画的属性声明，视图根据属性变化提供内建的动画支持。也就是说，开发者可以通过改变上述动画属性和定义一些相关属性，来设置动画效果。

UIView中有都实现动画效果的方式：UIView自带的动画、UIView的block动画、关键帧动画。

UIView的block动画的设置是相对来说比较简单的一种。创建一个新项目，模板为【Single View App】，在其ViewController.swift中编写代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import UIKit  class ViewController: UIViewController {     override func viewDidLoad() {          super.viewDidLoad()          //创建视图，颜色为黑色，并添加该视图到根视图          let rect = CGRect(x: 50, y: 80, width: 314, height: 240)          let view = UIView(frame: rect)          view.backgroundColor = UIColor.black          self.view.addSubview(view)          //设置执行动画的函数viewChange方法          func viewChange(){              view.bounds = CGRect(x: 0, y: 0, width: 114, height: 100)         view.alpha = 0.2              view.transform = view.transform.rotated(by: CGFloat(Double.pi))          }          //设置动画播放速度曲线类型          UIView.setAnimationCurve(.easeIn)          //设置动画时长，动画效果按照viewChange中的来执行       UIView.animate(withDuration: 5,delay: 1, options:.curveEaseIn, animations: viewChange)      }  } |

其中第13~16行中设置视图大小的缩小，透明度的变换和旋转。

第22行使用了UIView.animate()方法来设置动画。对于该函数，其参数及说明如下表：

表13-1 UIView.animate参数

|  |  |
| --- | --- |
| UIView.animate()参数 | 说明 |
| withduration: TimeInterval | 动画执行时间 |
| dealy: TimeInterval | 动画延迟执行时间 |
| uisngSpringWithDamping: CGfloat | 弹簧阻力，取值范围为 0.0~1.0，数值越小“弹簧”振动效果越明显。 |
| initialSpringVelocity: CGFloat | 动画初始速度（pt/s）,数值越大初始速度越大。但要注意初始速度太大而动画时间太小时，会发生反弹情况。 |
| options: UIViewAnimationOptions | 动画播放速度曲线 |
| animations:()-> Void | 执行动画的函数，是本动画的核心 |
| completion((Bool) -> Void)? | 动画之行结束的回调，可选性，可设为nil |

其中对动画最简单的初始化方式如下：

UIView.animate(withDuration: Int, animations: viewChange)

对于动画播放速度曲线，其类型见表13-2：

表13-2 UIView的block动画曲线类型

|  |  |
| --- | --- |
| 动画曲线类型 | 说明 |
| curveEaseInOut | 动画播放速度为开始和结尾慢中间快 |
| curveEaseIn | 动画播放速度为开始慢，之后逐渐加速 |
| curveEaseOut | 动画播放速度为开始快，之后逐渐减速 |
| curveLinear | 动画播放速度不变 |

运行项目，运行效果如图13-1所示。可以看到，动画的执行结果和viewChange()中设置的视图属性一样：旋转360°，尺寸逐渐收缩和透明度逐渐减小。

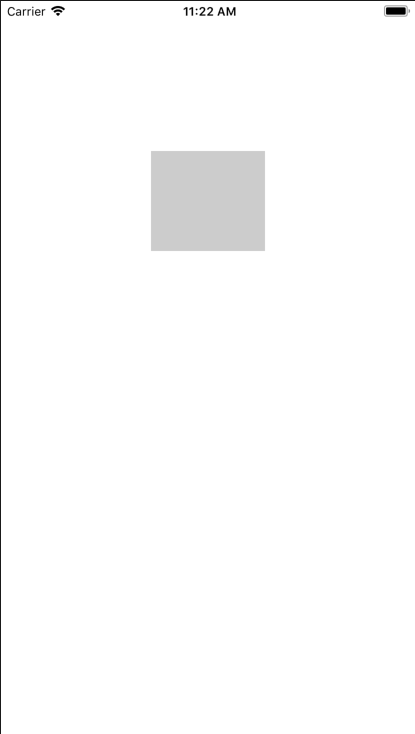
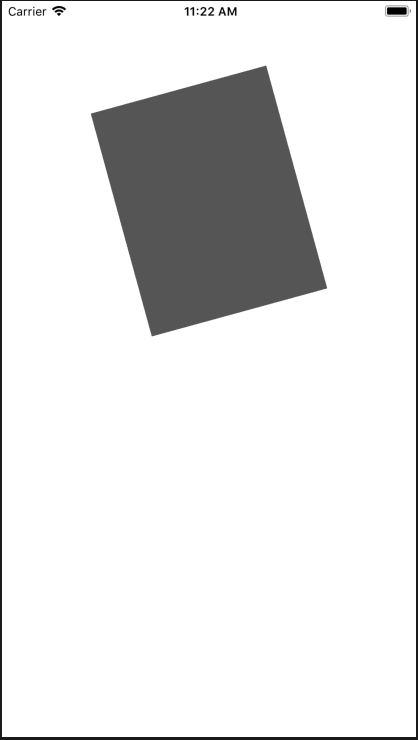
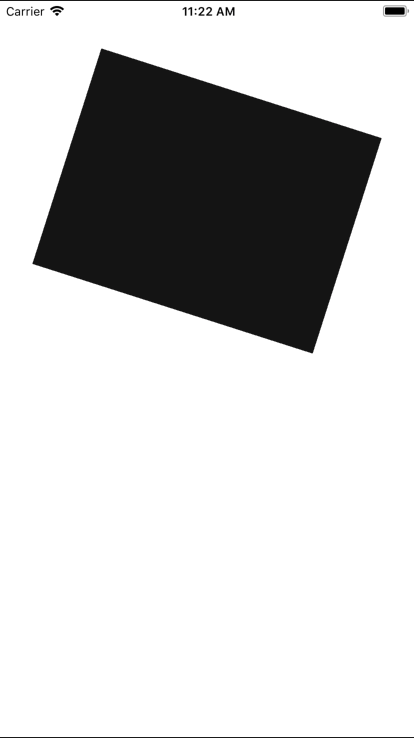
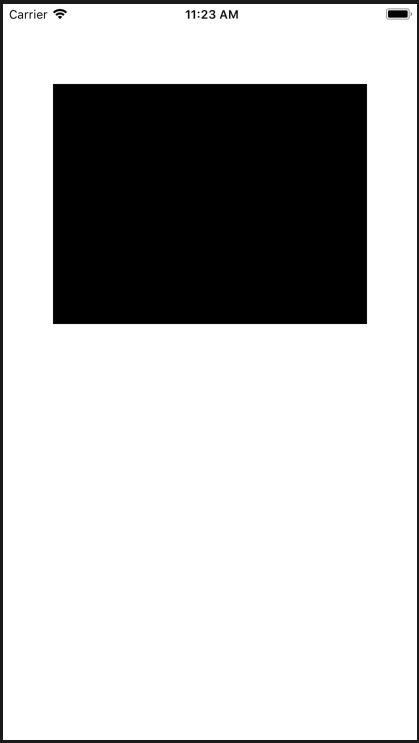


图13-1 动画效果

使用UIView自带的动画实现同样的效果。把第12~22行代码改为如下所示。

|  |  |
| --- | --- |
| 12  13  14  15  16  17  18  19  20  21  22 | //配置动画参数  UIView.beginAnimations(nil, context: nil)  UIView.setAnimationDuration(3)  UIView.setAnimationDelay(1)  UIView.setAnimationCurve(.easeIn)  UIView.setAnimationBeginsFromCurrentState(true)  //设置视图相关参数  view.bounds = CGRect(x: 0, y: 0, width: 114, height: 100)  view.alpha = 0.2  view.transform = view.transform.rotated(by: CGFloat(Double.pi)   UIView.commitAnimations() |

这里通过UIView的类方法beginAnimations来开始配置动画参数。此方法会启动动画，但不会立即执行动画，直到调用UIView中的类方法commitAnimations。视图对象执行介于beginAnimations和commitAnimations方法之间的操作。commitAnimation方法执行后，动画才会开始播放。

第16行中设置动画播放速度曲线与block动画表示方式有所不同，但效果相同，UIView动画中播放速度曲线类型包括：easeIn、easeOut、easeInOut、linear。第17行设置动画开始播放时的状态为当前状态。运行后的效果如图13-1所示。

### 13.1.2 使用UIImageView播放帧动画

帧动画即一张一张播放的动画。使用UIImageView可以存储一个UIImage数据类型的图片数组，然后根据设置好的图片序列来播放图片制作成帧动画。但是用这种方法播放动画请注意以下两点：

图片序列中的图片需要有相同的尺寸，否则会产生意料之外的动画效果。

每个图片的scale属性的值（即缩放比例）要相同。

创建一个新项目，模板为【Single View App】，在Xcode界面左侧的项目导航区打开,导入要生成动画的图片序列，如图13-2所示。

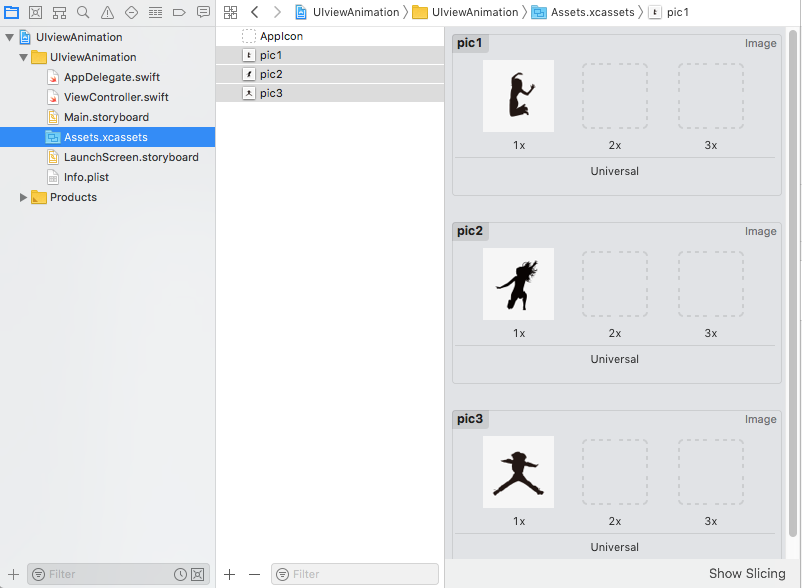


图13-2向Assets.xcassets导入图片序列

打开ViewController.swift文件，在其中编写代码如下。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  22  23  24  25 | import UIKit  class ViewController: UIViewController {    override func viewDidLoad() {          super.viewDidLoad()          // Do any additional setup after loading the view, typically from a nib.          //创建UIImage类型数组，并把图片按序列添加到数组中          var theImage = [UIImage]()          for i in 1...3{              theImage.append(UIImage(named: "pic\(i)")!)          }          //创建图片视图          let theImageView = UIImageView(frame: CGRect(x: 50, y: 150, width: 314, height: 314))          //设置图片视图对象为theImage图像数组          theImageView.animationImages = theImage          //设置动画时间          theImageView.animationDuration = 3          //设置重复播放s次数，0表示不限次数          theImageView.animationRepeatCount = 0        //开始动画播放          theImageView.startAnimating()          //将图片视图添加到根视图中          self.view.addSubview(theImageView)      }  } |

运行该项目，图片按以下图片顺序循环出现。

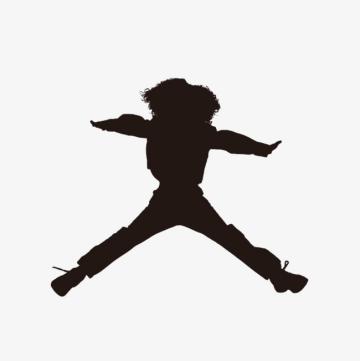
  

图13-3 帧动画中的图片

## 13.2 音频的播放

音频播放从形式上可分为音效播放和音乐播放。音效主要指通常作为点缀的音频，一般比较短暂。这类音频不需要进度控制和循环控制等。音乐指的是较长的音频，对于这类音频需要对其在音量、进度等方面有精确地控制。

在iOS中对音频的播放方式有4种：System Sound Services 、AVAudioPlayer、Audio Queue Services和Open AL技术。这四种音频播放方式各有其特点：

System Sound Services是最底层、最简单的音频播放服务，其特点为：1、限制：用于播放不超过30秒的声音。2、支持格式：CAF、AIF和使用PCM或IMA/ADPCM数据的WAV文件。3、其他：没有提供操纵声音和控制音量的功能。功能有：

a、声音：播放声音文件。如果手机被设置为静音，用户什么也听不到。

b、提醒：播放一个声音文件且震动，如果手机被设置为静音或震动，将通过震动提醒用户。

c、震动：手机震动。

AVAudioPlayer，其功能类似于System Sound Service，但是比System Sound Service强大的多，它可以实现：

a、播放任意时长的音频文件。

b、播放文件中或者内存缓冲区的音频文件。但它只能播放一个指定路径的音频，如果想要播放多个路径的音频就要创建多个AVAudioPlayer。

c、可控制音频播放的音量、进度，可以多个音频同时播放、支持循环播放。

Audio Queue Service可以实现对声音的完全控制。开发者可以将音频数据从文件中读取到内存缓冲区，并对声音数据进行特殊处理，如改变音频播放的速率、改变其声音的音色等。

Open AL是一个跨平台的开源音频处理接口，其设计给多通道三维位置音效的特效表现，使得它特别适合具有复杂音频使用场景的游戏开发。

接下来将会针对System Sound Services和AVAudioPlayer进行简单的讲解。

### 13.2.1 System Sound Services的使用

在设置一个自定义的警告音或者消息提示音时，System Sound Services无疑是最好的选择。它比其他方法更节省资源。

System Sound Services本身无法进行音频的循环播放，但是它可以通过其AudioServices- -AddSystemComplection方法对音频进行监听，在音频播放完后可以重新让音频播放，以此来实现音频的循环播放。接下来将使用System Sound Services重复播放一个短暂的铃声。

使用xcode新创建一个名为“shortVideo“的项目，项目模板为Single View App，创建完成后在项目名称文件夹上单机鼠标右键，选择右键菜单中的Add Files To，向项目中导入一个名为Hint、格式为mp3的音频文件。

在左侧项目导航区打开ViewController.swift文件，在其中编辑代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import UIKit  //使用System Sound Services要导入AudioToolbox框架  import AudioToolbox  class ViewController: UIViewController {      override func viewDidLoad() {          super.viewDidLoad()          // Do any additional setup after loading the view, typically from a nib.          var \_soundId:SystemSoundID = 0          //获得音频文件路径          let path = Bundle.main.path(forResource: "Hint", ofType: "mp3")          //将字符格式的文件路径转换为URL路径          let soundURL = URL(fileURLWithPath: path!)          //加载指定路径的音频文件，并创建一个System sound对象          AudioServicesCreateSystemSoundID(soundURL as CFURL, &\_soun dId)          //监听音频播放状态，实现c循环播放          AudioServicesAddSystemSoundCompletion(\_soundId, nil, nil, {(sound ID,clientData)->Void in              print("Audio restart.")              AudioServicesPlaySystemSound(soundID)          }, nil)          //开始播放音频          AudioServicesPlaySystemSound(\_soundId)      }  } |

运行该项目会循环播放音频，并且每次音频重新播放，都会在日志区输出一条语句，如图13-4所示。

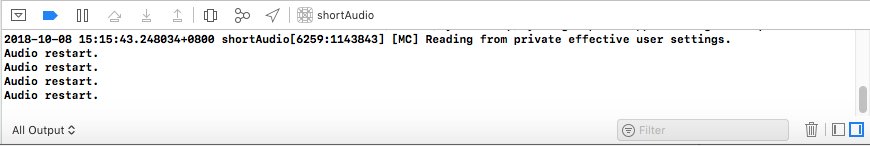


图13-4日志输出

### 13.2.2 使用AVAudioPlayer播放音乐

AVAudioPlayer支持对音频播放进度、音量大小和播放速率等的控制，通过遵循AVAudioPlayerDelegate协议实现对音频播放状态的监测。如音频文件是否处于播放状态、音频播放速率大小。其支持的音频文件格式如下。

a、AAC

b、AMR (Adaptive multi-Rate，一种语音格式)

c、ALAC (Apple lossless Audio Codec)

d、iLBC (internet Low Bitrate Codec，另一种语音格式)

e、IMA4 (IMA/ADPCM)

f、linearPCM (uncompressed)

g、u-law 和 a-law

h、MP3 (MPEG-Laudio Layer 3)

AVAudioPlayer常用属性说明如表13-3所示。

表13-3 AVAudioPlayer常用属性

|  |  |
| --- | --- |
| 属性名称 | 属性说明 |
| volume | 播放器的音频音量，值：0.0～1.0 |
| pan | 立体声设置 设为 －1.0 则左边播放 设为 0.0 则中央播放 设为 1.0 则右边播放 |
| isPlayer | 布尔类型，表示audio player是否处于播放状态 |
| rate | audio player播放视频的速度，范围在0.5~2.0之间，其中1.0表示正常播放速率 |
| numberOfLoops | audio player循环播放的次数，-1表示无限循环播放 |
| numberOfChannles | audio player的声道数 |
| duration | audio player音频的总长度，单位为秒 |
| currentTime | audio player当前播放位置的时间点，单位为秒 |

AVAudioPlayer常用方法说明如表13-4所示。

表13-4 AVAudioPlayer常用方法列表

|  |  |
| --- | --- |
| init(contentsOf:URL) | 从指定路径上加载并播放音频文件 |
| player() | 以异步的方式播放一条音频 |
| palyer(atTime:TimeInterval) | 从指定的位置开始音频文件的播放 |
| Pause() | 暂停音频的播放 |
| Stop() | 停止音频的播放 |
| prepareToPlay | 将音频文件加载到缓冲区，为其播放做准备 |
| Init(data：Data) | 从内存缓冲区加载并播放音频文件 |
| averagePower(forChannle：Int) | 获得正在播放的音频指定声道的平均功率，单位分贝 |
| peakPower(forChannle：Int) | 获得正在播放音频指定声道的峰值功率，单位分贝 |
| updateMeters() | 更新正在播放音频的所有声道的平均和峰值功率数值 |

使用Xcode新创建一个名为“musicOpen“的项目，项目模板为Single View App，创建完成后在项目名称文件夹上单机鼠标右键，选择右键菜单中的Add Files To，向项目中导入2个格式为mp3的音频文件。

在左侧项目导航区打开ViewController.swift文件，在其中编辑代码如下。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | import UIKit  //y使用AVAudioPlayer要引入AVFoundation框架  import AVFoundation  //给当前类文件添加AVAudioPlayerDelegate协议来实现对音频播放事件的监听  class ViewController: UIViewController ,AVAudioPlayerDelegate{      var audioPlayer:AVAudioPlayer = AVAudioPlayer()      var audioPlayer1:AVAudioPlayer = AVAudioPlayer()      override func viewDidLoad() {          super.viewDidLoad()          // Do any additional setup after loading the view, typically from a nib.          let width = UIScreen.main.bounds.width       //获取这2个音频文件的路径，将字符格式的文件路径转换为URL路径          let path1 = Bundle.main.path(forResource: "Hotei-Battle Without Honour", ofType: "mp3")          let path2 = Bundle.main.path(forResource: "Tobu[7obu]&Itro-Sunburst", ofType: "mp3")          let soundUrl = URL(fileURLWithPath: path1!)          let soundUrl1 = URL(fileURLWithPath: path2!)          //实现对这两个音频文件的初始化操作，并加载音频文件          do{            try audioPlayer = AVAudioPlayer(contentsOf: soundUrl)              audioPlayer.volume = 0.5              audioPlayer.numberOfLoops = -1              audioPlayer.delegate = self          }          catch{              print(error)          }          do{              try audioPlayer1 = AVAudioPlayer(contentsOf: soundUrl1)            audioPlayer1.volume = 0.5              audioPlayer1.numberOfLoops = -1              audioPlayer1.delegate = self          }          catch{              print(error)          }          //创建一个按钮，实现音频播放的暂停操作          let stopMusic = UIButton(frame: CGRect(x: 40, y: 80, width: width - 80, height: 44))          stopMusic.backgroundColor = UIColor.purple          stopMusic.setTitle("stop music", for: UIControlState.init(rawValue: 0))        stopMusic.addTarget(self, action: #selector(ViewController.pauseMusic), for: .touchUpInside)          self.view.addSubview(stopMusic)          //创建一个按钮，实现音频的切换播放          let changeMusic = UIButton(frame: CGRect(x: 40, y: 200, width: width - 80, height: 44))          changeMusic.backgroundColor = UIColor.purple         changeMusic.setTitle("change music", for: UIControlState.init(rawValue: 0))    changeMusic.addTarget(self, action: #selector(ViewController.changeMusic), for: .touchUpInside)          self.view.addSubview(changeMusic)          //创建一个按钮，实现开始音频播放和让暂停的音频播放的操作          let startMusic = UIButton(frame: CGRect(x: 40, y: 320, width: width - 80, height: 44))          startMusic.backgroundColor = UIColor.purple          startMusic.setTitle("start music", for: UIControlState.init(rawValue: 0))          startMusic.addTarget(self, action: #selector(ViewController.startMusic), for: .touchUpInside)        self.view.addSubview(startMusic)      }      var player1IsPause = false      var playerIsPause = false      //以下三个函数是对以上三个按钮操作的具体实现      @objc func pauseMusic(){          if self.audioPlayer.isPlaying || self.audioPlayer1.isPlaying{              if audioPlayer1.isPlaying{                  self.audioPlayer1.pause()                  player1IsPause = true              }else{                  self.audioPlayer.pause()                  playerIsPause = true              }          }      }      @objc func changeMusic(){          if self.audioPlayer.isPlaying || playerIsPause{              playerIsPause = false              self.audioPlayer.stop()              self.audioPlayer.currentTime = 0              self.audioPlayer1.play()          }          else{              player1IsPause = false              self.audioPlayer.play()              self.audioPlayer1.currentTime = 0              self.audioPlayer1.stop()          }      }      @objc func startMusic(){          if playerIsPause || player1IsPause{              if player1IsPause{                  self.audioPlayer1.play()                player1IsPause = false              }else{                  self.audioPlayer.play()                  playerIsPause = false              }          }          else{              self.audioPlayer.play()          }      }  } |

运行项目，模拟器界面如图13-5所示。点击start music按钮开始播放音频，点击stop music按钮暂停播放音频，再次点击start music按钮暂停的音频继续播放，点击change music按钮切换音频并且播放它。

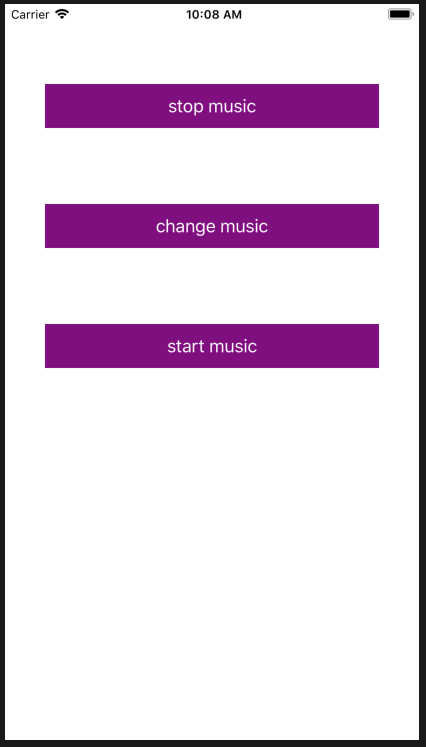


图13-5 音频播放模拟器界面

## 13.3 视频的播放

iOS开发中播放视频一般以以下两种方式播放：MPMoviePlayerController和AVPlayer。MPMoviePlayerController是iOS9.0版本之前的方式，这种视频播放方式比AVPlayer更简单，但是灵活上缺失，而AVPlayer可以高度自定义。所以如今大多数的视频播放都是用AVPlayer。接下来将会对AVPlayer的使用进行讲解，MPMoviePlayerController不再讲解说明，感兴趣的可以在网上查看。

### 13.3.1 使用AVPlayer播放视频

使用AVPlayer需要导入AVFoundation框架。AVPlayer是用来播放基于时间的视听媒体的控制器，但只是对播放和资源时间相关信息的管理，具体的用户界面需要开发者自定义。AVPlayer本身是不可见的，如果想让其可见，需要使用AVPlayerLayer类。简单的视频播放只需创建AVPlayer视频操作对象，将该对象添加到一个AVPlayerLayer类实例上，然后将该AVPlayerLayer类实例添加到当前视图控制器的根视图的子层中。

使用Xcode新创建一个名为“movieBegin“的项目，项目模板为Single View App，创建完成后在项目名称文件夹上单机鼠标右键，选择右键菜单中的Add Files To，向项目中导入1个格式为mp4的音频文件。

在左侧项目导航区打开ViewController.swift文件，在其中编辑代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import UIKit  //需要导入AVFoundation框架  import  AVFoundation  class ViewController: UIViewController {      override func viewDidLoad() {          super.viewDidLoad()          // Do any additional setup after loading the view, typically from a nib.          //得到文件的URL          let moviePath = Bundle.main.path(forResource: "01", ofType: "m4v")          let movieURL = URL(fileURLWithPath: moviePath!)          //创建AVPlayer对象          let avPlayer = AVPlayer(url: movieURL as URL)          //将AVPlayer对象加入到AVPlayerLayer层中          let avPlayerLayer = AVPlayerLayer(player:avPlayer)          //设置层范围大小和视频播放尺寸          avPlayerLayer.frame = self.view.bounds          avPlayerLayer.videoGravity = AVLayerVideoGravity.resizeAspect          //加入到当前视图控制器根视图层中          self.view.layer.addSublayer(avPlayerLayer)          //播放          avPlayer.play()      }  } |

运行项目，模拟器界面如图13-6所示。



图13-6 视频播放模拟器界面

其中代码的17行设置了VideoGravity属性，它有三个选项，对应的作用如下：

表13-5 AVPlayerlayer的VideoGravity属性类型

|  |  |
| --- | --- |
| 方法名称 | 说明 |
| AvPlayerVideoGravityResizeAspectFill | 保留视频的宽高比，对视频进行缩放，将其填满层的区域范围 |
| AvPlayerVideoGravityResizeAspect | 保留视频的宽高比，对视频进行缩放，在视频层完整显示 |
| AvPlayerVideoGravityResize | 将视频进行拉伸来匹配层的显示区域 |

在ViewControoler中有一个AVPlayerViewController子类，不仅可以实现视频播放，还提供了额外的视频播放控制界面。使用它需要引入AVKit框架。这里对其不再讲解。

## 13.2 本章小结

本章介绍了UIView的自带动画、UIView的block动画、通过UIImageView实现动画效果。UIView自带的动画和block动画能够实现相同的效果，只是实现方式不同；UIImageView是通过播放一系列的图片实现动画效果。然后又讲述了如何实现音视频的简单播放，主要是对当今流行的一些音视频框架的大致使用进行了介绍。

# 第十四章 数据的持久化

数据持久化就是将内存中的数据模型转换为存储模型，以及将存储模型转换为内存中的数据模型的统称。应用进程的许多数据需要存储在硬盘中进行持久化存储，如游戏进度存档、用户历史信息等。

使用数据持久化的好处：

1、持久化技术封装了数据访问细节，为大部分业务逻辑提供[面向对象](http://www.so.com/s?q=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&ie=utf-8&src=internal_wenda_recommend_text)的API。

2、通过持久化技术可以减少访问数据库数据次数，增加应用程序执行速度。

3、代码重用性高，能够完成大部分数据库操作。

4、松散耦合，使持久化不依赖于底层数据库和上层业务逻辑实现，更换数据库时只需修改配置文件而不用修改代码。

iOS中使用的数据持久化方式有：1、property list（属性列表）2、Preference（偏好设置）3、NSKeyedArchiver（归档）4、SQLite3/FMDB(嵌入式数据库) 5、CoreData(面向对象的嵌入式数据库)

本章主要介绍对property list的创建、读取、添加、修改、删除和写入以及如何对CoreData进行数据的存储、查找、编辑和删除操作。

## 14.1 plist文件

plist文件全名为property list，即属性列表文件。它以类似于键值对(key-value)的形式来存储项目的一些属性或配置信息。在新建一个项目时，项目文件夹中会自动包含一个名为[info.plist]文件，该文件存储了该项目的各项配置信息，包括该项目版本号、语言环境、主故事版等信息的配置。plist文件的存储格式为xml格式。plist文件内容及其xml源代码如图14-1和图14-2所示。

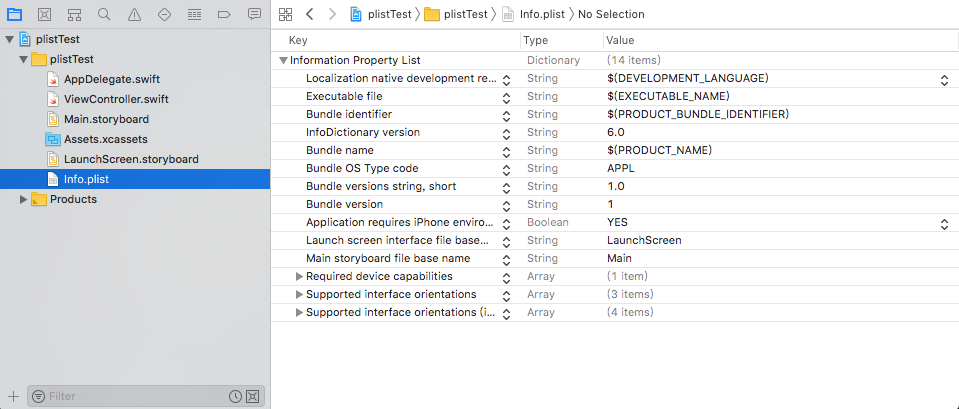


图14-1 plist文件

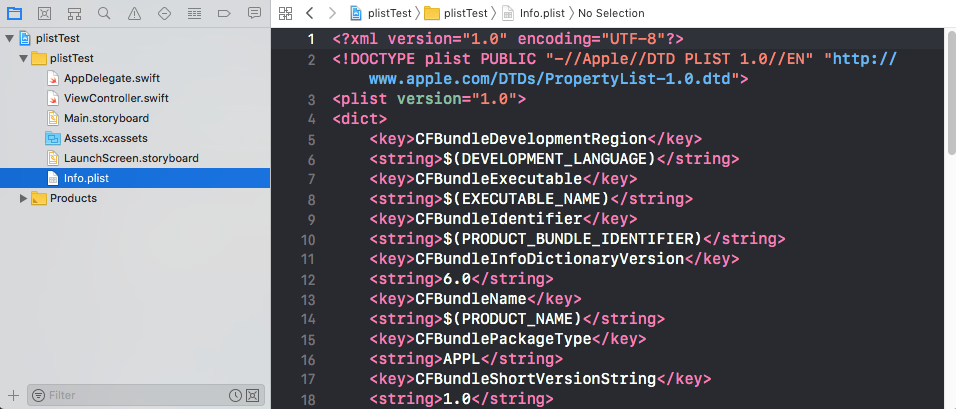


图14-2 plist文件xml源代码

### 14.1.1 plist文件创建和解析

创建一个基于【Single View Application】模板的空项目，创建完毕后在项目名称文件夹上点击鼠标右键，在弹出的菜单栏中选择【New File】，接着在弹出的模板窗口中在iOS栏选择【Resource】模板下的【Property List】，创建plist文件，如图14-3所示。

创建的plist文件在项目文件夹中可以看到。点击该文件，会在右侧窗口显示该plist文件内容。在右侧窗口处点击右键，在弹出的菜单栏选择【Add Row】，这样就可以在plist文件中添加一个新的键值对，如图14-4所示。

双击键对相应的Key和Value位置可对其进行编写，对选中的键对点击Key栏的‘+’号可以添加一行数据，点击‘-’号可以删除该行数据。点击键值对的Type栏可以在弹出的菜单栏选择该行数据的的数据类型。plist文件的数据类型有Array、Dictionary、Boolean、Date、Data、String、Number。其中前两个为集合类型。plist文件的根数据类型即解析plist文件得到的数据类型，只有两种NSArray或者NSDictionary。

如图14-4所示在plist文件中添加两行数据。

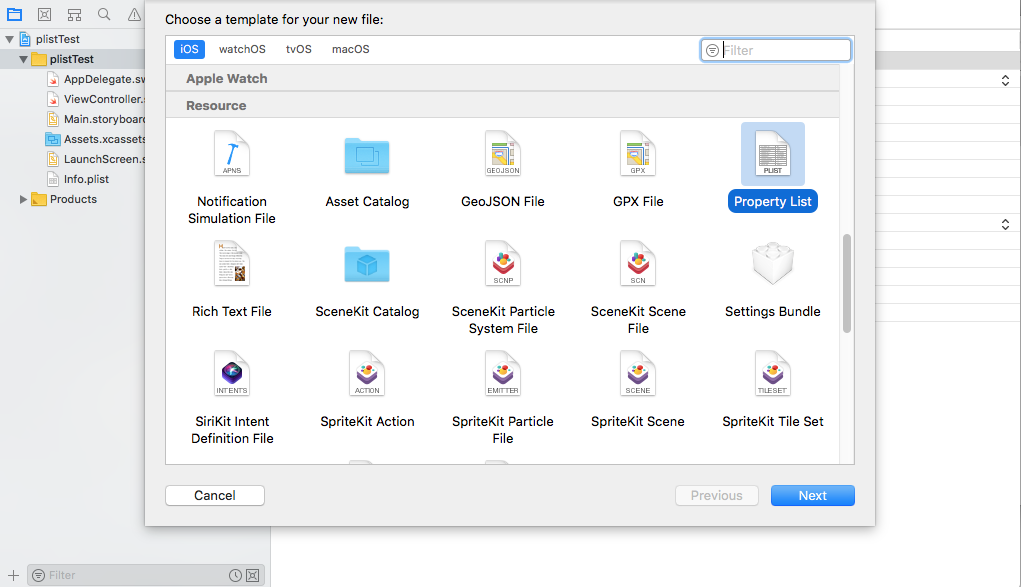


图14-3

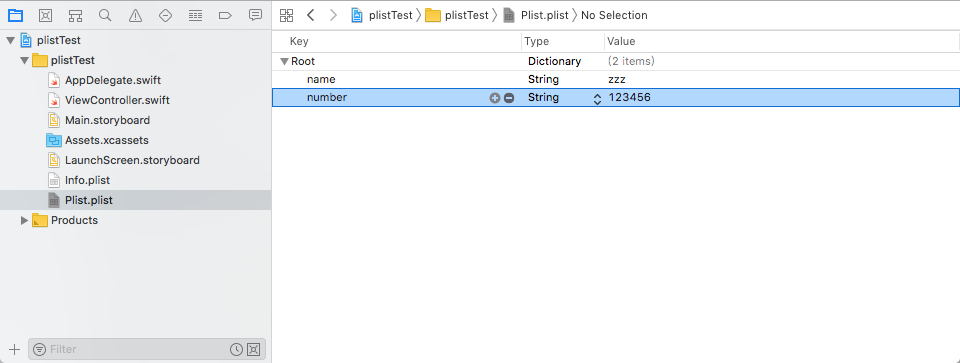


图14-4

编辑【ViewController.swift】文件，输入以下代码，实现解析plist文件并在日志区输出plist文件内容的操作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import UIKit  class ViewController: UIViewController {   override func viewDidLoad() {        super.viewDidLoad()        // Do any additional setup after loading the view, typically from a nib.        //获得plist文件在沙箱中的路径        let path=Bundle.main.path(forResource: "Plist", ofType: "plist")        //根据该路径加载plist文件,并将其转换为可变字典类型对象        let plistDate:NSMutableDictionary=NSMutableDictionary.init(contentsOf File: path!)!          //获得字典对象的描述信息          let info=plistDate.description          print(info)      }  } |

日志区输出结果如图14-5所示。

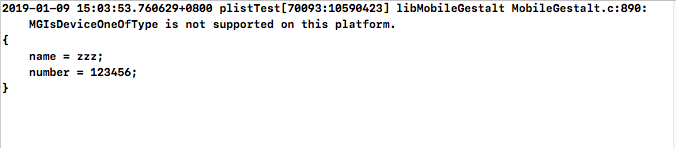


图14-5 日志区输出结果

### 14.1.2 plist文件的修改、删除和写入

编辑【ViewController.swift】文件，输入以下代码，实现plist文件的修改、删除和写入操作并在日志区输出修改后的plist文件内容。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import UIKit  class ViewController: UIViewController {      override func viewDidLoad() {          super.viewDidLoad()          // Do any additional setup after loading the view, typically from a nib.          //获得plist文件在沙箱中的路径          let path=Bundle.main.path(forResource: "Plist", ofType: "plist")          //根据该路径加载plist文件,并将其转换为可变字典类型对象   let plistDate:NSMutableDictionary=NSMutableDictionary.init(contents  OfFile: path!)!          //对Key为name的键值对进行修改，以下两个操作结果相同          //plistDate.setObject("changed", forKey: "name" as NSCopying)          plistDate["name"]="abc"          //添加一行数据          plistDate.setObject("123456", forKey: "password" as NSCopying)          //删除一行数据          plistDate.removeObject(forKey: "number" as NSCopying)          //将修改后的内容写入文件          plistDate.write(toFile: path!, atomically: true)          //获得字典对象的描述信息并输出          let info=plistDate.description          print(info)          //输出plist文件路径          print(path!)      }  } |

日志区输出结果如图14-6所示。

打开Finder，使用快捷键command+shift+g，复制日志输出的路径来找到plist文件所在文件夹，打开该文件，文件内容如图14-7所示。



图14-6 日志输出结果

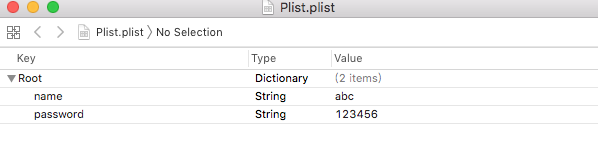


图14-7 沙箱内的plist文件

这里需要注意的是，在plist文件写入时，会先清空整个plist文件，然后重新将修改后的内容写进去。如果执行以下操作。

|  |  |
| --- | --- |
| 1  2  3 | let data:NSMutableDictionary=NSMutableDictionary()  data.setObject("ccc", forKey: "name" as NSCopying)  data.write(toFile: path!, atomically: true) |

即创建一个新的可变字典类型对象并写入plist文件，那么plist文件的内容只有data的内容。所以要修改plist文件内容要先将其内容赋值给一个对象，然后对该对象进行修改操作，再写入plist文件。

## 14.2 CoreData

CoreData是Apple官方为iOS提供的一个数据持久化方案，其本质是一个通过封装底层数据操作，让程序员以面向对象的方式存储和管理数据的ORM框架（Object-Relational Mapping：对象-关系映射，简称ORM）。虽然底层支持SQLite、二进制数据、xml等多种文件存储，但是主要还是用来操作SQLite数据库。

开发者不需要学习或者使用SQL语句，只需要使用CoreData框架提供的对象和接口以及图形化工具，即可完成SQLite数据库的创建、表关系、增删改查等一系列操作，在一定程度上降低了开发人员的学习成本并增加了代码的统一性和可阅读性。

CoreData主要包含以下几个类：

a、NSManagedObjectModel：托管对象模型，映射实体类和数据库数据的关系，本质是一个XML文件

b、NSManagedObject：托管对象，对应数据库数据的实体

c、NSManagedObjectContext：托管对象上下文，管理托管对象

d、NSPersistentStoreCoordinator：持久化存储调度器，用来处理磁盘持久化数据和实体类对象的相互转化

e、NSPersistentStore：持久化存储器，负责磁盘持久化数据存取

CoreData的总体结构框架如图14-8所示。

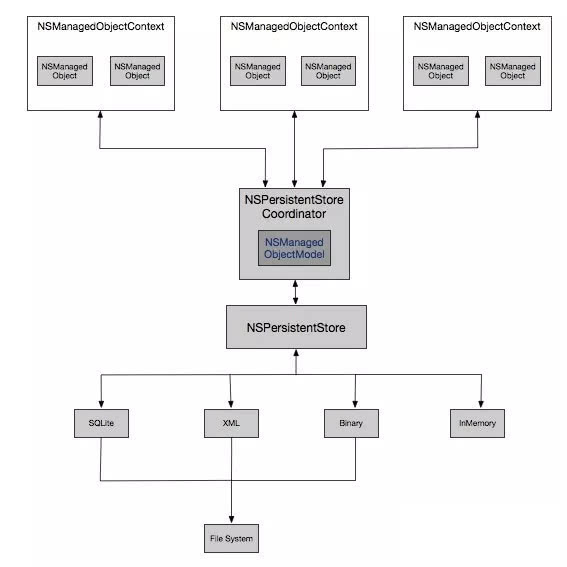


图14-8

本节将要介绍的是CoreData的创建、插入、查找、修改和删除操作。

### 14.2.1 CoreData实体的创建和数据的插入

若想在项目中使用CoreData，需要引入CoreData框架，最简单的方法是在创建项目时勾选【Use Core Data】，然后创建好的项目会自动生成一个后缀名为.xcdatamodeld的文件。该文件可以使用图形化的方式编辑数据类型。打开AppDelegate.swift文件会发现在文件的末尾多出了关于CoreData的内容。

点击底部的【Add Entity】，添加一个新的实体，将实体名称改为Student。在【Attribute】设置区，点击+号图标为实体添加3个属性并设置其数据类型，如图14-9所示。注意，实体名称的首字母必须大写，而属性名称首字母必须小写。

编辑【ViewController.swift】文件，输入以下代码，实现对CoreData实体插入数据的操作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import UIKit  //引入CoreData框架  import CoreData  class ViewController: UIViewontroller {   override func viewDidLoad() {         super.viewDidLoad()         // Do any additional setup after loading the view, typically from a nib.        //获得当前项目的AppDelegate对象，进而取出其中的托管对象上下文       let appDel:AppDelegate = UIApplication.shared.delegate as! AppDelegate        let context=appDel.persistentContainer.viewContext        //在托管对象上下文中插入一个Student实体对象        let stu=NSEntityDescription.insertNewObject(forEntityName: "Student", into: context) as! Student        //给实体对象各属性赋值        stu.name="zzz"       stu.id="123456"        stu.department="computer"        //将实体对象保存到托管对象上下文中        do {              try context.save()            print("save data successfully.")          } catch  {             print("save data failed.")          }      }  } |

运行该项目，若保存成功会在日志区输出“save data successfully.”。

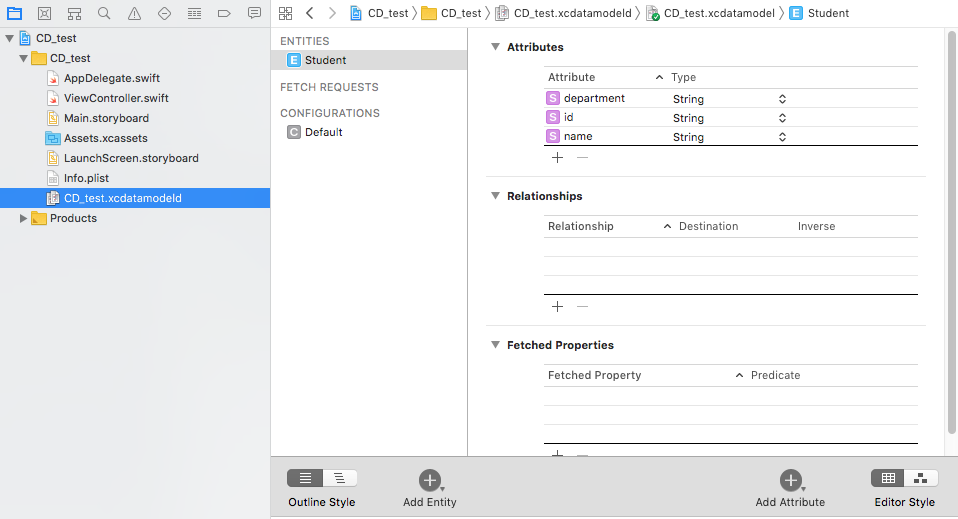


图14-9 CoreDate实体属性

### 14.2.2 CoreData的查找操作

要查找CoreData的数据需要使用NSPredicate谓词和NSFetchRequest类。使用托管对象上下文来查找数据时，要先创建请求对象，再编写查询条件，请求对象根据查询条件在托管上下文中进行数据查找。

在上一节的代码上进行修改，此时的托管对象上下文已经存储了上一节中的实体。修改11行以后的代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | //创建Student类型实体   let entity=NSEntityDescription.entity(forEntityName: "Student", in: context)  //创建请求对象和查询谓词，设置请求对象的实体类型、查询谓词、偏移量和查询结果数量   let req=NSFetchRequest<Student>(entityName: "Student")  req.entity=entity   //创建查询谓词，设置请求对象的查询谓词  let pred=NSPredicate.init(format: "id='123456'", "")  req.predicate=pred  //设置请求对象的查询偏移量和查询数量  req.fetchLimit=20  req.fetchOffset=0  //查询与谓词相对应的实体数据并输出  do {     let datas:[Student]?=try context.fetch(req)     for stu in datas!{        print("name=\(stu.name!)")        print("id=\(stu.id!)")        print("department=\(stu.department!)")         }    } catch  {       print("fetch failed.")  } |

运行项目，结果如图14-10所示。

图14-10

对于18行，如果req.predicate=nil,那么以上代码会输出在该托管对象上下文中Student的全部实体。

### 14.2.3 CoreData修改和删除操作

如果要对CoreData中的数据进行修改和删除，首先要查找数据库中所需要修改的对应实体，通过托管对象上下文对实体对应的托管对象进行操作，然后将更改后的结果保存到持久化存储中。

假设在CoreData中已经存储了3个Student实体，内容如下：

Entity1：name=zzz，id=123456，department=computer

Entity2：name=abc，id=248650，department=math

Entity3：name=xyz，id=112233，department=computer

编辑【ViewController.swift】文件，输入以下代码，实现对CoreData实体修改数据的操作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import UIKit  import CoreData  class ViewController: UIViewController {      override func viewDidLoad() {          super.viewDidLoad()          // Do any additional setup after loading the view, typically from a nib.       let appDel:AppDelegate = UIApplication.shared.delegate as! AppDelegate          let context=appDel.persistentContainer.viewContext         let entity=NSEntityDescription.entity(forEntityName: "Student", in: context)          let req=NSFetchRequest<Student>(entityName: "Student")          req.entity=entity          let pred=NSPredicate.init(format: "id='123456'", "")          req.predicate=pred          req.fetchLimit=20          req.fetchOffset=0          do {             //查找到要修改的实体，通过托管对象上下文进行修改，然后保存              let changedDatas:[Student]?=try context.fetch(req)              for stu in changedDatas!{                  stu.department="computer"              }              try context.save()              //输出CoreData中的所有Student实体              req.predicate=nil              let datas:[Student]?=try context.fetch(req)              for stu in datas!{                  print("name=\(stu.name!)")                  print("id=\(stu.id!)")                  print("department=\(stu.department!)")              }          } catch  {              print("fetch failed.")          }      }  } |

运行项目，日志区输出结果如图14-11所示。

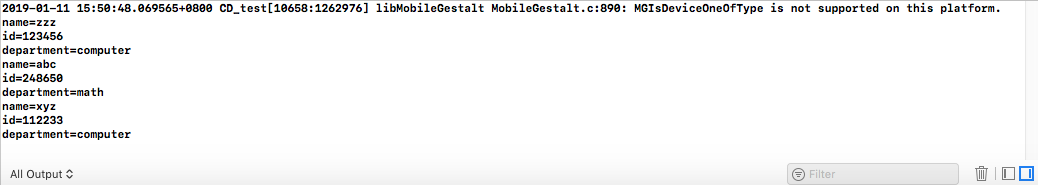


图14-11

删除CoreData的数据使用delete(\_ object: [NSManagedObject](apple-reference-documentation://hszrY8Gi1t))方法。将上述代码的22行修改为。

|  |  |
| --- | --- |
| 1  2  3  4 | for stu in changedDatas!{    context.delete(stu)  }  try context.save() |

就成功删除了谓词所对应的实体。运行修改后的项目，日志区输出结果如图14-12所示。

图14-12

## 14.3本章小结

本章介绍了对plist文件的创建、解析、修改和删除以及通过Core Date实现对数据的创建、读取、查找、修改和删除。plist文件主要用于记录键值对类型的配置信息，而Core Date是App存储数据的主要方式，在iOS开发中的作用比较大，是本章的重点。

第

三

部

分

项

目

篇

# 第十五章 生日记录簿

本章将会应用前面学习的内容来完成一个名为“生日记录薄”的App。该App实现记录姓名、联系方式和生日。该App完成后的界面如图15-1所示。

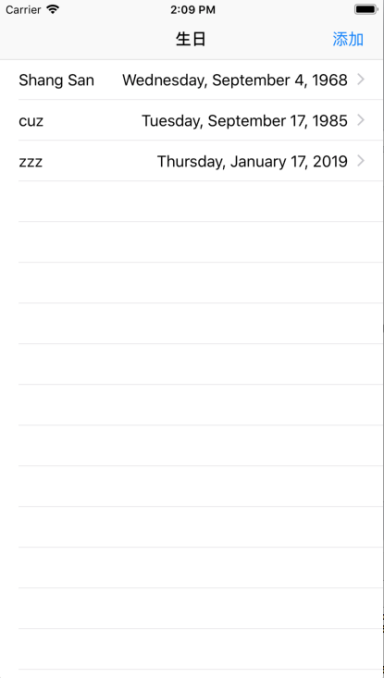


图15-1 生日记录薄App界面

## 15.1 设置App界面

创建新项目，命名为birthdayBook。注意在创建的时候选中Use CoreData一项。创建完成后点击项目文件夹的Main.storyboard。在故事板的library中找到Table View Controller，将其拖拽到故事板上。把原本指向View Controller的箭头移动，使其指向Table View Controller。给这两个视图控制器分别添加一个Navigation Controller，具体方法是：选中视图控制器，在屏幕左上角的菜单栏点击【Editor】-->【Embled In】-->【Navigation Controller】。

展示出故事板的Document Outline，找到Table View Controller Scene栏中的Navigation Item，在右侧的属性检查器例输入屏幕的标题，这样导航条会显示该标题。将Table View Controller的标题设为“生日列表”，View Controller的标题设为“添加生日”。完成后结果如图15-2所示。

接着在视图控制器中添加按钮、标签等控件。在Table View Controller的导航条右侧添加一个Bar Button Item，设置标题为“添加”。点击Phototype Cells下方空白处，在属性查看器的Table View Cell栏的Style设置单元格显示内容的方式为Right Detail，Identifierce设置为birthday，Accessory设置为Disclosure Indicator。

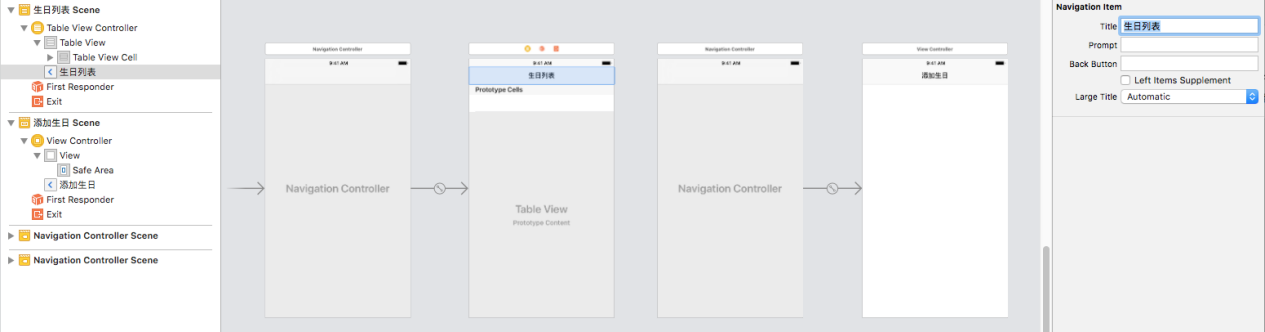


图15-2 storyboard显示结果

在View Controller的导航条左右侧各添加一个Bar Button Item，标题分别为“取消”和“保存”。添加3个Label，标题分别为“名字”、“联系方式”、“日期”。再添加2个Text Field和一个Date Picker。将它们放置到合适的位置，如图15-3所示。

点击“保存”按钮，将其拖拽到View Controller的导航控制器上，形成一个Segue在弹出的菜单栏中选择Present Modally。可以实现点击该按钮转到“添加生日”视图中。对Phototype Cells做同样操作。为了把这两个Segue区分开，设置Phototype Cells的Segue为update。这样基本的界面就布置完毕了，效果如图15-3所示。

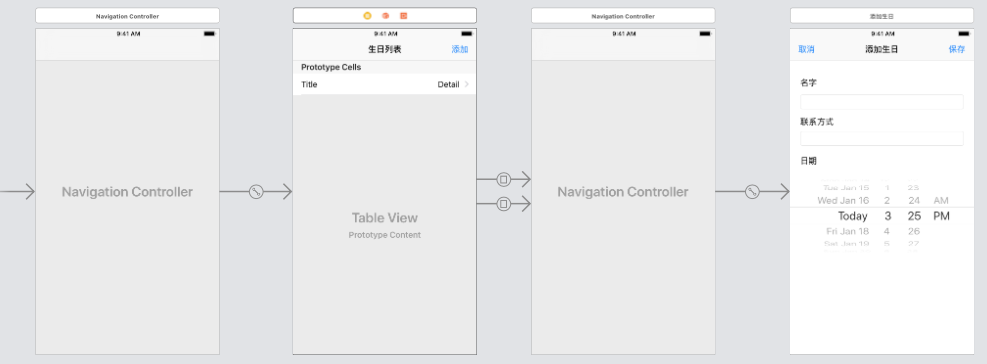


图15-3 界面布置结果

为了使App在任何设备上都有完美外观，需要让App自己调整所有东西的位置和大小来适配设备。选中View Controller视图控制器，点击编辑区右下角4个图标的最右边的图标，在弹出的菜单栏选中Resolve Auto Layout Issue，自动设置布局约束来适配设备。

## 15.2 代码和故事板的连接

为了方便识别文件所指向的视图控制器，修改Viewcontroller.swift文件名为AddBirthdayViewcontroller.swift，并将代码中的类名改为AddBirthdayViewcontr- -oller。右击项目文件夹，点击菜单栏中的【New File...】，添加【Cocoa Touch Class】

文件，确定【Subclass of:】栏为UITableViewController，将类名改为birthdayTableV- -iewController。将故事板中的Table View Controller的类改为birthdayTableV- -iewController，View Controller的类改为AddBirthdayViewcontroller。利用编辑助理将编辑区分为2部分，左边为AddBirthdayViewcontroller.swift，右边为故事板。将AddBirthdayViewcontroller视图控制器的2个Bar Button Item、2个Text Field和一个Date Picker都拖拽到AddBirthdayView cont -roller.swift中并命名。其中2个Bar Button Item为Action，其他为Outlet。如图15-4所示。

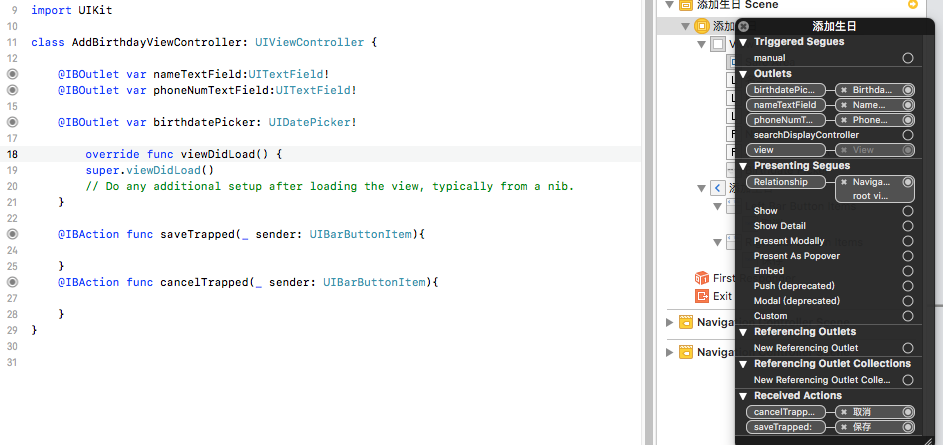


图15-4 代码与控件连接

接着设置Phototype Cells的identifier为birthdayCellIdentifier。接下来设置CoreData的实体及其属性。设置结果如图15-5所示，其中的birthdayId是为了之后对数据进行修改和删除而做的唯一标识符。

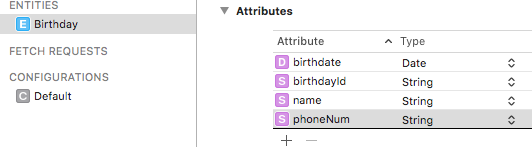


图15-5 CoreData实体及其属性

打开BirthdayTableViewcontroller.swift，引入CoreData框架，在viewDidLoad函数中和其上方添加以下代码：

|  |  |
| --- | --- |
|  | class birthdayTableViewController: UITableViewController {  //创建实体数组和日期格式器  var birthdays=[Birthday]()  let dateFormatter=DateFormatter()  override func viewDidLoad() {  super.viewDidLoad()  //使该格式器只显示日期  dateFormatter.dateStyle = .full  dateFormatter.timeStyle = .none  } |

使 numberOfSections(in：）返回值为1，表示表格视图只有一个章节。然后使tableView (\_:numberOfRowsInSection:)返回值为birthdays.count来设置表格视图行数。找到tableView (\_:cellForRowAt:)函数，取消其注释，在其中修改代码为：

|  |  |
| --- | --- |
|  | //以下函数本被注释掉，找到并修改，其作用是说明单元格的内容  override func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  let cell = tableView.dequeueReusableCell(withIdentifier: "birthday CellIdentifier", for: indexPath)  let birthday=birthdays[indexPath.row]  let name=birthday.name  cell.textLabel?.text=name  if let date=birthday.birthdate as Date?{  cell.detailTextLabel?.text=dateFormatter.string(from: date)  }  else{  cell.detailTextLabel?.text=""  }  return cell  } |

这样就使表格视图的每一行显示出对应的内容了。

## 15.3 实现数据的添加、删除和修改

对于数据的添加，在App中的实际操作是：点击“添加”按钮进入“添加生日”界面，在文本框输入相应内容、使用日志选择器选好日期后点击“保存”按钮。数据的删除操作为：左滑单元格会在该单元格右边出现“Delete”，点击它删除数据。数据的修改操作为：点击单元格，进入“添加生日”界面，该界面的文本框和日志选择器此时保存着对应的单元格的数据，修改文本框内容和日期后点击“保存”。由于添加和修改操作都要进入同一个视图控制器，且修改操作涉及传值操作，因此在AddBirthdayViewController.swift文件的viewDidLoad()方法上方添加以下代码：

|  |  |
| --- | --- |
|  | //通过以下变量进行页面转变时的传值  var name:String!  var phoneNumber:String!  var date:Date!  var id:String!  //判断是修改数据还是添加数据  var isChange:Bool = false |

打开BirthdayTableViewController.swift文件，在其类中重写prepare（for:sender:）函数：

|  |  |
| --- | --- |
|  | //判断是要修改还是添加  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  // Get the new view controller using segue.destination.  // Pass the selected object to the new view controller.  if segue.identifier=="update"{  let navigationController = segue.destination as! UINavigation Controller  let addBirthdayViewController = navigationCon troller.topView Controller as! AddBirthdayViewController  addBirthdayViewController.navigationItem.title="修改生日"  let indexPath = self.tableView.indexPathForSelectedRow  let birthday = birthdays[indexPath!.row]  addBirthdayViewController.id=birthday.birthdayId  addBirthdayViewController.isUpdate=true  addBirthdayViewController.name = birthday.name  addBirthdayViewController.phoneNumber = birthday.phoneNum  addBirthdayViewController.date = birthday.birthdate!  }  else{  let navigationController = segue.destination as! UINavigation Controller  let addBirthdayViewController = navigationController.topView Controller as! AddBirthdayViewController  addBirthdayViewController.isUpdate=false  } |

该函数的作用是：判断点击的是“添加”按钮还是单元格，如果是添加按钮，使addBirthdayViewController中的isUpdate为false，否则isUpdate为true，并进行传值操作。由于是修改数据，在单击单元格时修改了“添加生日”界面的Navigation Item的标题为“修改生日”。

由于生日不会超过当前日期，打开addBirthdayViewController.swift文件，在viewDidLoad()函数中添加一行代码：

|  |  |
| --- | --- |
|  | birthdatePicker.maximumDate=Date() |

在之前添加的saveTapped(\_:)函数中添加以下代码：

|  |  |
| --- | --- |
|  | @IBAction func saveTapped(\_ sender: Any) {  let appDelegate = UIApplication.shared.delegate as! AppDelegate  let context = appDelegate.persistentContainer.viewContext  let name = NameTextField.text!  let phoneNum = phoneNumTextField.text!  let birthdate = birthdatePicker.date  if isUpdate==false {  //添加数据到数据库中  let newBirthday = Birthday(context: context)  newBirthday.name = name  newBirthday.phoneNum = phoneNum  newBirthday.birthdate = birthdate  newBirthday.birthdayId = UUID().uuidString  if let uniqueId = newBirthday.birthdayId{  print("birthday是\(uniqueId)")  }  do {  try context.save()  } catch {  print("save failed.")  }  }  if isUpdate==true{  //找到与点击的单元格对应的数据并修改它  Let entity:NSEntityDescription?=NSEntityDescription.entity(for EntityName: "Birthday", in: context)  let request = NSFetchRequest<Birthday>(entityName: "Birth day")  request.fetchOffset=0  request.entity=entity  let predicate = NSPredicate(format: "birthdayId=%@", id)  request.predicate=predicate  do{  let result:[AnyObject]=try context.fetch(request)  for birthday:Birthday in result as! [Birthday]{  birthday.name=NameTextField.text  birthday.phoneNum=phoneNumTextField.text  birthday.birthdate=birthdatePicker.date  }  try context.save()  }  catch {  print("failed.")  }  }  dismiss(animated: true, completion: nil)  } |

点击取消按钮时，只需退出当前页面即可，在CancelTapped(\_:)函数中添加一行：

|  |  |
| --- | --- |
|  | dismiss(animated: true, completion: nil) |

对于点击单元格进行页面转换的传值操作，在前面内容提到过已经把单元格中的数据传给addBirthdayViewController.swift中的几个变量。重写viewWillAppear(\_:)函数：

|  |  |
| --- | --- |
|  | override func viewWillAppear(\_ animated: Bool) {  if isUpdate==true{  NameTextField.text=name  phoneNumTextField.text=phoneNumber  birthdatePicker.date=date  }  } |

对于数据的删除操作，打开BirthdayTableViewController.swift，找到tableView(\_:cellEditRowAt:)和tableView(\_:commit:)函数取消注释，将

tableView(\_:cellEditRowAt:)返回值设为true，使得单元格可编辑，然后将ableView(\_:com mit:) 函数修改为以下代码。

|  |  |
| --- | --- |
|  | // Override to support editing the table view.  override func tableView(\_ tableView: UITableView, commit editingStyle: UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {  if birthdays.count > indexPath.row {  let birthday = birthdays[indexPath.row]  let appDelegate = UIApplication.shared.delegate as! AppDele gate  let context = appDelegate.persistentContainer.viewContext  context.delete(birthday)  birthdays.remove(at: indexPath.row)  do{  try context.save()  }catch{  print("failed")  }  tableView.deleteRows(at:[indexPath], with: .fade)  }  } |

至此，“生日记录薄”App基本完成了。想要修改App在模拟器中显示“生日记录薄”这个名字和其图标，向项目文件夹添加图片，打开Info.plist文件，添加2项分别为Bundle display name和Icon file，分别输入“生日记录薄”和图片名，再运行模拟器，就会发现对应的App名字和图标都变成想要的了（如图15-6所示）。



图15-6 App在模拟器中

为了实现生日列表按照名字排序显示，重写BirthdayTableViewController.swift中的viewWillAppear(\_:)函数：

|  |  |
| --- | --- |
|  | override func viewWillAppear(\_ animated: Bool) {  super.viewWillAppear(animated)  let appDelegate = UIApplication.shared.delegate as! AppDelegate  let context = appDelegate.persistentContainer.viewContext  let fetchRequest = NSFetchRequest<Birthday>(entityName: "Birth day")  let sortDescriptor = NSSortDescriptor(key: "name", ascending: false)  fetchRequest.sortDescriptors = [sortDescriptor]  do {  birthdays = try context.fetch(fetchRequest)  } catch{  print("failed")  }  tableView.reloadData()  } |

运行项目，试试App是否像预期的那样正常运行。

## 15.4 本章小结

本章通过项目实例，简单描述了基本的iOS设备App的开发过程，也是对之前所学内容的一个简单实现。通过本章能够将本书所教内容有效的结合起来。该项目相对简单，逻辑清晰易懂，对iOS开发感兴趣的可以在网上寻找较复杂的项目，相信会对开发能力的提升有很大帮助。