# 第一章 iOS简介以及环境搭建

如今苹果公司的产品风靡全球，如mac电脑、iPhone手机、iPad等产品，而这同时也要求大量App的加入来满足各种人群的不同需求。有需求就有市场，iOS开发应运而生。当今iOS开发有2种语言：Objective-C和Swift。但随着时间的发展，Swift逐渐成为iOS开发领域的主流，如今的iOS开发大多数使用Swift语言。本章将向读者介绍iOS系统，Swift语言及Xcode开发工具的安装和简单使用。

## 1.1 iOS系统简介

iOS系统是由苹果公司开发的移动操作系统 。该系统由苹果公司于2007年1月9日在Macworld大会上首次公布，原名为iPhone OS，最初的目的是设计给iPhone使用的，后来陆续套用到iPod touch、iPad以及Apple TV等产品上,因此改名为iOS系统。

### 1.1.1 iOS发展历程

2007年1月9日苹果公司在Macworld展览会上公布，随后于同年的6月发布第一版iOS操作系统，最初的名称为“iPhone Runs OS X”。

2008年3月6日，苹果发布了第一个测试版开发包，并且将 “iPhone runs OS X” 改名为 “iPhone OS” 。

2008年9月，苹果公司将iPod touch的系统也换成了”iPhone OS“。

2010年2月27日，苹果公司发布iPad，iPad同样搭载了”iPhone OS”。这年，苹果公司重新设计了“iPhone OS”的系统结构和自带程序。

2010年6月，苹果公司将“iPhone OS”改名为“iOS”，同时还获得了思科iOS的名称授权。

2011年10月4日，苹果公司宣布iOS平台的应用程序已经突破50万个。

2012年6月，苹果公司在WWDC 2012上宣布了iOS 6，提供了超过 200 项新功能。

2013年6月10日，苹果公司在WWDC 2013上发布了iOS 7，几乎重绘了所有的系统APP，去掉了所有的仿实物化，整体设计风格转为扁平化设计。将于2013年秋正式开放下载更新。

2013年9月10日，苹果公司在2013秋季新品发布会上正式提供iOS 7下载更新。

2014年6月3日（西八区时间2014年6月2日），苹果公司在WWDC 2014上发布了iOS 8，并提供了开发者预览版

2014年6月9日，苹果公司在WWDC 2015开发者大会上发布了iOS 9操作系统。

2016年6月13日，苹果开发者大会WWDC在旧金山召开，会议宣布iOS 10的测试版将在2016年夏天推出，正式版将在秋季发布。

2017年6月6日，苹果公司在圣何塞McEnery会议中心召开了WWDC2017全球开发者大会，会上发布了iOS 11系统的测试版，正式版于2017年秋季发布。

iOS 12在2018年6月5日的WWDC2017大会上亮相后开启内测，开发者和用户已经可以开始下载iOS 12开发者预览版Beta1固件进行升级体验。预计在秋季发布会上会发布iOS 12正式版。

### 1.1.2 iOS系统特性

iOS是众人所熟知的一款非常先进的操作系统，清晰易懂的界面、丰富的功能和其极稳定的性能，已经成为iPhone、iPad和iPod touch的强大基础。在这个拥有如此多的强大竞争对手的行业里，iOS系统的稳定、反应迅速和功能众多导致了Apple设备如今遥遥领先的地位。

总结iOS的功能特性，有以下几点：

（1）界面优雅直观，操作简单。苹果公司的产品极其注重用户体验，其设备界面优雅直观，产品操作易上手。这都得益于iOS中极具创新的其专为手指而设计的Multi.Touch界面。

（2）功能丰富。随着iOS系统的版本更新以及硬件的性能增长，其内置辅助功能变得更多、更强大。每一次的版本更新都会推出让人眼前一亮的新功能，用户的体验更加丰富，使用起来更加充满乐趣。App Store中的App也日益丰富，满足了不同用户的不同需求。

（3）安全。iOS 专门设计了低层级硬件和固件功能，用以防止恶意软件和病毒；同时还设计有高层级的 OS 功能，有助于在访问个人信息和企业数据时确保安全性。设备还支持“远程定位”和远程操纵手机擦除信息的功能。即使设备丢失用户也不必太过于担心信息泄露和无法找回设备问题。找回的设备还可以重新恢复至丢失前的状态。

（3）更新方便、免费。iOS系统免费更新，可以在App Store上将其直接下载到iPhone、iPad.

（4）软硬件配合完美、高效。苹果公司的产品的软件和硬件皆由自己设计，因此在设计的同时得以兼顾软硬件之间的配合。因此Apple的硬件和软件系统可以配合的如此天衣无缝。

（5）支持语言众多。这是iOS 设备可在世界各地通用的重要设计。30 多种语言供用户挑选，用户还可以在各种语言之间轻松切换。其键盘有50多种支持特定语言功能的不同版本任用户选择。此外，内置词典支持 50 多种语言。

## 1.2 Swift语言介绍

Swift是苹果于2014年WWDC（苹果开发者大会）发布的新开发语言，可与Objective-C共同运行于Mac OS和iOS平台，用于搭建基于苹果平台的应用程序。其创造者为克里斯·拉特纳（Chris Lattner），[LLVM](https://baike.baidu.com/item/LLVM)项目的主要发起人与作者之一，[Clang](https://baike.baidu.com/item/Clang)[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)的作者。现为[苹果公司](https://baike.baidu.com/item/%E8%8B%B9%E6%9E%9C%E5%85%AC%E5%8F%B8)编译器开发团队的首席架构师。

Swift语言发展时间很短，但它的发展是十分迅速的。来自 RedMonk 编程语言排行榜的公布的 2018 年第一季度编程语言排行榜显示，Swift语言成功挤进全球编译语言前10列，成为增长速度最快的编程语言，并已经超越了Objective-C。

Swift语言发展的迅速，与它本身的强大是分不开的。相比Objective-C，Swift语言语法更为简单，代码简洁，操作方便，也使得应用程序的开发更为简单，快捷，稳定。在安全性和质量上与Objective-C相比也有了提高。如今国内国外对于苹果应用的开发大多使用Swift语言，各大高校关于iOS开发的课程也改为了Swift语言。可以看到，随着时间的流逝，Swift语言必将成为iOS开发的主流。

## 1.3 开发环境的搭建

苹果的软件开发大多在一个功能强大、齐全的开发工具——Xcode上实现。Xcode是一款行在操作系统Mac OS X上的集成开发工具（[IDE](https://baike.baidu.com/item/IDE/8232086)），可以进行macOS，iOS，tvOS，watchOS 平台软件的开发。它具有统一的用户[界面设计](https://baike.baidu.com/item/%E7%95%8C%E9%9D%A2%E8%AE%BE%E8%AE%A1)，[编码](https://baike.baidu.com/item/%E7%BC%96%E7%A0%81/80092)、测试、[调试](https://baike.baidu.com/item/%E8%B0%83%E8%AF%95)都在一个简单的窗口内完成。

搭建开发环境就要下载Xcode，开发者可以在App Store上直接搜索Xcode下载，App Store上的软件为默认最新版，这要求你的系统也是最新的。如果系统不是最新的，开发者可以在苹果官网寻找到适合自己系统版本的Xcode版本来下载。安装Xcode会要求输入你的一个Apple ID，没有Apple ID可以在苹果官网上注册。

Swift开发工具官网地址：[https://developer.apple.com/Xcode/download/](https://developer.apple.com/xcode/download/)。

苹果官网（中国）地址：<https://www.apple.com/cn/。>

### 1.3.1 Xcode欢迎界面介绍

Xcode 安装完成后就可以开始编写 Swift 代码了。打开Xcode，进入欢迎界面，如图1-1所示。

右边显示最近在Xcode中打开的文件。你也可以点击右下方的“open another project...”来打开其他的Xcode文件。左边可以看到有3个选项，它们的含义说明如下：

Get started with a playground:针对Swift语言创建的功能 ，可以实时地将编写的代码进行调试和显示结果于右侧信息栏。

Creat a new Xcode project:创建一个独立的Xcode项目。

Check out an existing project:从仓库中拉取已经存在的项目。

本书主要使用前两个选项。



图1-1 Xcode欢迎界面

### 1.3.2 创建Playground文件

选择 Get started with a playground 选项，在弹出的界面上方选择iOS，Xcode默认为Blank，点击Next，并输入playground 的名称。如图1-2是默认 Swift Playground 的开始窗口。



图1-2 Playground 的开始窗口

Playground文件十分简约，界面简单，主要作用是便于Swift语法代码的学习和演示。开发者编写代码时，每一行的代码结果都会在右边的信息栏中显示。同时playground也提供可以选择让结构显式地在代码下方展示出来的功能。建议初学者在Playground上进行代码练习。

### 1.3.3 创建iOS项目

打开Xcode，选择Create a new Xcode project,如下图创建新项目。点击Next，接着我们输入项目名称（ProductName)，本节创建的项目名称为“HelloWorld”,公司名称（Organization Name),公司标识前缀名（Organization identifier) 还要选择开发语言(Language),选择设备（Dev- -ices)。其中Language选择Swift项了。点击”Next”下一步。选择要存储的文件夹，点击右下角的Create。

项目创建后，默认生成了一个示例文件，如图1-4所示（在界面左侧）。打开main.storyboard,默认看到一个简单的空白的设备应用界面。main.storyboard显示的模拟器大小可以在其正下方界面最底部“View as”看到，点开后会以界面大小顺序显示不同型号模拟器界面大小（如图1-5所示）。Xcode还允许控制main.storyboard中设备界面显示的比例（图1-5右上角部分）。如想设置运行后显示的模拟器型号，则要点击界面标题栏左上位置（如图1-4标记位置）设置模拟器显示的设备型号。

我们为界面添加点内容，在右下方找到Text控件，将它拖入storyboard上，

双击写入文本"Hello World!"（如图1-6）。

运行一下模拟器（command+R 快捷键或在菜单栏中选择 Product => Run或击左上角的运行符号)。模拟器运行结果如图1-7。



图1-3 创建一个Singles View App

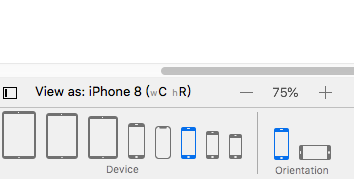
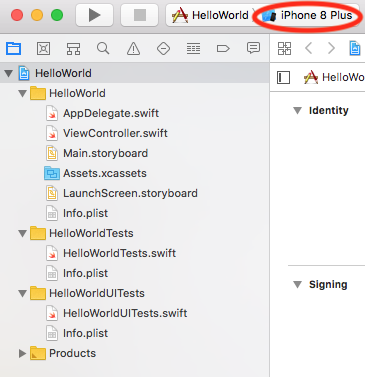


图1-4 示例文件 图1-5 模拟器大小显示



图1-6 添加TextField控件



图1-7 模拟器运行结果

## 1.4 本章小结

本章主要介绍了iOS的发展及其特性，xcode9.0的安装和使用。本章的重点是如何创建swift编译环境和如何使用xcode来编译和创建工程。要求熟练掌握swift编译环境的创建、xcode的安装和使用。

# 第二章 数据类型与运算

程序运算时需要存放数据的载体。本章就来介绍Swift 4中存放数据的载体——常量、变量，还有变量的基本数据类型、以及运算符等基础知识。

## 2.1 变量与常量

在Swift语言中变量值可以分为常量与变量。在程序执行过程中，值一旦设定就不能够发生变化的量称为常量，其值能够发生变化的量称为变量。

### 2.1.1声明常量和变量

常量和变量必须在使用前声明，使用的声明关键字也不同，用let来声明常量，用var来声明变量。

|  |  |
| --- | --- |
|  | let pi:Double=3.1415926 //不可修改  var schoolName:String="HDU" //可以修改 |

可以用print()函数来输出这个常量和变量

|  |  |
| --- | --- |
|  | print(pi)  //输出: 3.1415926  print(schoolName)  //输出：HDU |

声明常量和变量时，需要定义其数据类型。但是不定义数据类型也能够编译成功,不过前提是已经赋值了。这是由于编译器xcode会自动根据用户所赋的值，判断出其所属的类型。

### 2.1.2变量与常量的命名规范

Swift语言中，常量和变量的命名规则十分宽泛，可以包括Unicode字符和数字，但需要注意的是，不可使用语言环境中的关键字来作为常量或者变量的名称，例如let、var不可作为标识符来使用。另外，常量和变量的命名不可以数字开头，空格、数字符号、制表符、箭头等符号也不可用在命名中。

和其他语言类似，Swift也可以用中文进行变量的命名，示例如下：

|  |  |
| --- | --- |
|  | var 杭电 = "杭电" |

可以穿插数字进行命名，注意数字不能作为开头：

|  |  |
| --- | --- |
|  | var name1 = "雨燕" |

可以使用下划线进行命名：

|  |  |
| --- | --- |
|  | var \_swift\_ = "swift" |

虽然Swift的命名方式相对的宽泛，但在实际开发中是多人合作开发，所以良好的代码风格有利于提高编码效率和代码的可读性，所以推荐读者使用驼峰命名方式。所谓驼峰命名是指以单词进行名称的拼接，名称的首字母一般为小写，之后每个单词的首字母大写，其他字母均小写，示例如下：

|  |  |
| --- | --- |
|  | //驼峰命名  var userName = "雨燕" |

## 2.2 关于注释

注释是对代码内容进行说明的解释性文本。在代码进行编译时，注释语句不会被编译进工程中，使用注释能够让别人读懂你的代码。而Swift语言中采用和C语言类似的注释方式，使用//符号来注释单行内容，同时也可以用/\*开头，以\*/结尾的方式进行多行注释，示例如下：

//单行注释

/\*

多行注释

多行注释

多行注释

\*/

Swift语言也可以支持注释的嵌套，示例如下：

//单行注释//注释中的注释

/\*多行注释

/\*

注释中的注释

\*/

多行注释

\*/

## 2.3数据类型

本节主要向读者介绍Swift语言中常见的数据类型，如布尔型、整型、浮点型、字符型以及字符串类型等。

### 2.3.1 布尔型数据

布尔类型很多时候也叫做逻辑类型。对于熟悉Java编程语言的读者可能会了解，在Java语言中，Bool类型其实并非严格意义上的逻辑布尔类型，Java中可以使用零与非零来表达逻辑假与逻辑真。而在Swift语言中则不同，Swift语言的Bool类型十分严格，只有true和false两种值，分别表示真和假。同样，在Swift语言的条件语句以及需要进行逻辑判断的语句中，所使用的条件表达式的值也必须为Bool类型。下面有两个例子：

【例2-1】:

|  |  |
| --- | --- |
|  | var bool1 =true  if bool1{  print("oh,right,boo1 is true!")  }else{  print("oh,no,boo1 not true!")  } |

程序输出结果:

|  |  |
| --- | --- |
|  | oh,right,bool1 is true |

【例2-2】:

|  |  |
| --- | --- |
|  | var bool2 = 1  if bool2{  //这个例子不会通过编译，会报错  } |

### 2.3.2 整数型数据

Swift语言中的整型数据分为有符号整型数据与无符号整型数据。所谓有符号与无符号，通俗的理解即为是否有负数部分。

Swift提供了8，16，32和64位的有符号和无符号整数类型。这些整数类型和C语言的命名方式很像，比如8位无符号整数类型是UInt8，而32位符号整数类型是Int32。你可以访问不同整数类型的min和max属性来获取对应类型的最大值和最小值，实例如下：

|  |
| --- |
| let minValue = UInt8.min  //minValue为0，是无符号8位整数类型的最小值  let minValue = Int8.min  //minValue为-128,是有符号8位类型的最小值  let maxValue = UInt8.max  //maxValue为255，是无符号8位整数类型的最大值  let minValue = Int8.max  //minValue为127,是有符号8位类型的最大值 |

同理可得UInt16,UInt32,UInt64的最大值与最小值，请读者自行验证，在此不再赘述。

### 2.3.3 浮点型数据

浮点型数据用来表示实数。浮点型数据分为单精度浮点型与双精度浮点型，分别用Float与Double表示。

Swift语言中也支持使用科学计数法来表示数字，在十进制中使用e来表示10的n次方，在十六进制中使用p来表示2的n次方。示例代码如下：

|  |  |
| --- | --- |
|  | var num1 = 1.31e4 //1.13\*(10^4) = 13100  var num2 = 0x1p2 //1\*(2^2) = 4 |

### 2.3.4 字符类型

字符类型用来表示单个字符，如数字字符、英文字符、符号字符和中文字符等都可以使用字符类型来表示。

类似于C语言的Char,Swift语言中使用Character来描述字符类型，Character类型占8个字节的内存空间，String类型占24个字节的内存空间。在Swift中可以使用MemoryLayout枚举来获取某个类型所占用的内存空间，其单位为字节，示例如下:

|  |  |
| --- | --- |
|  | print(MemoryLayout<String>.size) |

程序输出结果:

|  |  |
| --- | --- |
|  | 24 |

可知String类型占用24字节内存空间，同理读者可以去验证一下各种类型所占的字节数。

创建字符的两种方法，示例如下：

|  |  |
| --- | --- |
|  | var c1:Character = "a" //直接创建出一个字符  var c2=Character("a") //使用构造方法创建字符类型变量 |

## 2.4字符串

字符串类型是开发中使用最多的一种数据类型，在本节中，会详细的介绍一些字符串的相关操作。

### 2.4.1创建字符串

在Swift语言中，有两种方式去创建字符串，例如：

|  |  |
| --- | --- |
|  | var string1 = "hello world"  print(string1)  var string2 =String("hello world!")  print(string2) |

以上程序执行输出结果：

|  |  |
| --- | --- |
|  | hello world!  hello world! |

### 2.4.2创建空字符串

创建一个空字符串，并且判断它是否为空。

|  |  |
| --- | --- |
|  | var string1=""  if string1.isEmpty{  print("string1是空字符串")  }else{  print("string1不是空字符串")  } |

以上程序执行输出结果为:

|  |  |
| --- | --- |
|  | string1是空字符串 |

### 2.4.3字符串的比较

在实际的开发过程中，经常需要进行字符串的比较。例如，判断用户在表单中输入的数据，与数据库中保存的数据是否一致。代码如下：

|  |  |
| --- | --- |
|  | var inputString = "12345"  var dbString = "123456"  if inputString == dbString{  print("输入正确")  }else{  print("输入错误")  } |

以上程序输出：

|  |  |
| --- | --- |
|  | 输入错误 |

如果只需要简单判断字符串的前部分或后部分，是否与另一个字符串相同，可以使用类型中提供的hasPrefix()和hasSuffix()两个方法，分别进行判断：

|  |  |
| --- | --- |
|  | let firstName = "杭州欢迎您"  let secondName = "北京欢迎您"  print(firstName.hasPrefix("杭州")) //说明是否包含杭州这个前缀  print(secondName.hasSuffix("欢迎您")) //说明是否包含欢迎您这个后缀 |

以上程序输出：

|  |  |
| --- | --- |
|  | true  true |

### 2.4.4字符串的大小写转换与遍历

可以使用uppercased()和lowercased()方法，进行英文字符串的大小写转换

|  |  |
| --- | --- |
|  | let hello="Hello World"  print(hello.uppercased())  print(hello.lowercased()) |

以上程序输出：

|  |  |
| --- | --- |
|  | HELLO WORLD  hello world |

字符串是由一个个字符相互组合而成的，有时需要遍历字符串中的每一个字符，可以用for-in遍历。实例如下：

|  |  |
| --- | --- |
|  | var teleName="135777777"  for i in teleName{      print("\(i) ",terminator:"") //terminator为空，表示输出不换行  } |

以上程序输出：

|  |  |
| --- | --- |
|  | 1 3 5 7 7 7 7 7 7 |

### 2.4.5字符串的截取和移除

在实际的开发过程中，会经常用字符串来查找特定的内容，或者使用字符串来替换另一个字符串。Swift中为开发者提供了很多的相关方法。

|  |  |
| --- | --- |
|  | var vString = "hello world"  var sString=vString[vString.startIndex...vString.index(vString.startInde x,offsetBy:5)]  print(sString)  vString.remove(at:vString.index(vString.startIndex,offsetBy:5))  print(vString) |

以上程序输出：

|  |  |
| --- | --- |
|  | hello  helloworld |

第二行代码的意思是，从vString字符串的开始，到字符串的第五个字符截取下来，然后赋给sString字符串。而第四行的代码意思是，从vString字符串左边开始的第五个字符删除。

## 2.5运算符

运算符也叫做操作符，一般的表达式是由操作数和运算符组成。常见的运算符有赋值、区间、算术、条件等。

### 2.5.1赋值运算符

赋值运算符是在编程中运用频率最高的运算符之一。与其他语言是相似的，使用的是“=”符号，符号的左边是被赋值对象，可以是变量，也可以是表达式，但不能是个常量，而符号的右边可以是个常量。对于初学者来说，时常会混淆相等运算符“==”与赋值运算符“=”的作用。例如:

|  |  |
| --- | --- |
|  | var str:String = "hello world"  //其实这里的“=”就是充当赋值运算的功能 |

### 2.5.2区间运算符

在C语言中，来限定一个变量的取值范围，一般会用这种表达式(i>0 && i<10)。但是Swift中有新的表示方式，例如要输出0->3这四个数，在Swift中是这么做的。

|  |  |
| --- | --- |
|  | for index in 0...3{  print(index)  } |

以上程序输出结果:

|  |  |
| --- | --- |
|  | 0 1 2 3 |

运算符“...”表示的是左闭右闭区间，而 “..<” 表示的是左闭右开区间。

|  |  |
| --- | --- |
|  | for index in 0..<3{  print(index)  } |

以上程序输出结果:

|  |  |
| --- | --- |
|  | 0 1 2 |

注意：区间操作只有这两种，读者切勿自己推导出<..等不存在的运算符。

### 2.5.3算术运算符

在处理一些基础的运算时，算术运算符非常的常见，例如加、减、乘、除、取余等。但是从Swift3.0开始就已经弃用了++、--这两个运算符。

故在此版本中对于自增、自减，利用赋值运算符与算术运算符相结合的复合赋值运算符。示例如下：

|  |  |
| --- | --- |
|  | var tmp=1  //加赋值复合运算  tmp+=3 //tmp = tmp + 3  //减赋值复合运算  tmp-=3 //tmp = tmp-3 |

### 2.5.4条件运算符

条件运算符(又称三目运算符)是一种三元运算符，此运算符需要三个变量或表达式。其能够简便实现代码中的条件选择逻辑。如下例子中为一个简单的条件选择语句示例：

|  |  |
| --- | --- |
|  | var a = 1  var b = 2  if a>b{  print("a>b")  }else{  print("b>=a")  } |

对于以上判断a与b两者的大小，看起来特别的麻烦，如果利用条件运算符，一行代码就能完成。可以简写成如下形式：

print(a>b ? "a>b":"b>=a")

即三目运算符的基本语法格式：条件？成立时执行的语句：不成立时执行语句

## 2.6本章小节

本章主要为读者介绍Swift语言中的常量与变量，以及该语言中一些常用到的数据类型，并且对常见的运算符进行了介绍，读者应该重点关注本语言与其他语言的语法差异，这样学习起来就比较容易。

# 第三章 流程控制

程序中的语句通常是顺序执行的，即从上至下，依次执行，直至结束。顺序结构往往无法满足复杂的现实需求，所以Swift语言为开发者提供了流程控制语句。利用流程控制语句，可以让计算机重复执行某段代码块，或是让程序跳转执行不同的分支等等。在Swift中，流程控制语句主要包含循环语句，条件语句和控制转移语句。

## 3.1循环语句

### 3.1.1 for-in

早先版本的Swift提供两种for循环，第一种是传统三段式for循环，也就是for-conditio n-increment循环，另一种是for-in循环，常用于遍历一个集合里的所有元素。但传统的for循环在Swift3.0中被移除了，所以不做介绍。

Swift中for-in循环的基本语法如下：

|  |  |
| --- | --- |
|  | for 索引in 集合类型{      循环体  } |

用for-in循环打印区间0...5之间的数字。

|  |  |
| --- | --- |
|  | for index in 0...5{      print("当前index为\(index)")  }  //输出:  //当前index为0  //当前index为1  //当前index为2  //当前index为3  //当前index为4  //当前index为5 |

在本例中，index是每次循环遍历开始时被自动赋值的常量。循环开始时，index被赋值为0...5这个区间中的第一个元素0，并且执行打印语句，接下来index被赋值为区间中的下一个元素也就是1，再次打印，以此类推，直至遍历完整个区间。这种情况下，index不需要使用let关键字进行声明，并且在循环体内，不可以修改index的值。

如果不需要index这个索引，比如用for-in循环来计数，就可以使用\_代替index。

|  |  |
| --- | --- |
|  | for \_ in 0..<3{      print("Swift")  }  //输出  //Swift  //Swift  //Swift |

显然打印3次Swift并不需要index，所以用\_代替，这是一种更加优雅的方式。

for-in循环也常常被用来遍历集合类型。用for-in来遍历数组。

|  |  |
| --- | --- |
|  | let strs = ["Swift", "iOS", "HDU"]  for str in strs{      print(str)  }  //输出:  //Swift  //iOS  //HDU |

常量str依次被赋值为数组中的数据项并输出。因为数组是有序的，所以使用for-in循环遍历数组，输出结果和构造数组时的顺序保持一致，而集合和字典内部是无序的，遍历结果可能和构造顺序不一致。

### 3.1.2 while

while循环会一直执行循环体直到判定条件变成假（false）。while循环非常适合使用在第一次迭代前，迭代次数未知的情况下。

Swift提供两种形式的while循环：

while循环，在每次执行循环体之前判断判定条件是否为真（true）。

repeat-while循环，在每次执行循环体之后判断判定条件是否为真（true）。

* while

while循环基本语法：

|  |  |
| --- | --- |
|  | while 条件 {  循环体  } |

while循环先判断条件再执行语句，如果条件为true，会执行循环体，然后再次判断，直到条件变为false。同样是打印0...5之间的数字，这次使用while循环来实现。

|  |  |
| --- | --- |
|  | var index = 0  while index <= 5 {      print("当前index为\(index)")      index += 1  }  //输出:  //当前index为0  //当前index为1  //当前index为2  //当前index为3  //当前index为4  //当前index为5 |

index初始时被赋值为0，第一执行循环体之前，判断index的值是否小于等于5，条件成立，执行print语句，随后执行index+1，index的值变为1。在下一次执行循环体之前，再次判断，之后执行循环体，直到不再满足判定条件，while循环结束。

* repeat-while

repeat-while循环基本语法：

|  |  |
| --- | --- |
|  | repeat{  循环体  }while 条件 |

repeat-while循环不像for-in和while循环在循环体开始执行之前判断条件语句，而是在循环体执行结束之后判断条件是否符合。 使用repeat-while打印0...5之间的数字。

|  |  |
| --- | --- |
|  | var index2 = 0  repeat{      print("当前index2为\(index2)")      index2 += 1  }while(index2<=5)  //当前index2为0  //当前index2为1  //当前index2为2  //当前index2为3  //当前index2为4  //当前index2为5 |

执行完一次循环体之后，如果条件为true，控制流会跳转回上面的repeat，然后重新执行循环体，直到条件变为false为止。

请注意，由于条件表达式出现在循环的尾部，所以循环体在条件被检测之前至少被执行一次。

|  |  |
| --- | --- |
|  | var index3 = 0  repeat {      print("当前index3为\(index3)")  } while (index3 < 0)  //输出:当前index3为0 |

尽管index初始值被设置为0，不满足小于0的条件，但是循环体还是会执行一次，打印出0。

注意：在使用while循环的时候，在循环体中要对判定条件进行一定的修改，使判定条件在有限的循环次数之后变为false，不然会出现死循环。

## 3.2 条件语句

条件语句通过判断条件的真假来控制程序执行不同的分支，在条件为true时执行指定的语句，在条件为 false 时执行另外的语句。Swift 提供两种类型的条件语句：if语句和switch语句。通常，当条件较为简单且可能的情况很少时，使用if语句，而switch语句适用于条件较复杂、有更多排列组合的情况。

### 3.3.1 if-else

if-else基本语法：

|  |  |
| --- | --- |
|  | if 条件 {  语句  } else {  语句  } |

if语句最简单的形式就是只包含一个条件，只有条件为true时，才执行相关代码。

|  |  |
| --- | --- |
|  | var varA:Int = 10  if varA < 20 {      print("varA 小于 20")  }  //输出:varA 小于 20 |

通过配合else从句，实现二选一执行。也就是当条件为false的时候，执行else语句。

|  |  |
| --- | --- |
|  | var varB:Int = 30  if varB < 20 {      print("varB 小于 20")  } else {      print("varB 大于等于 20")  }  //输出:varB 大于等于 20 |

还可以通过多个if-else配合的方式，来实现更复杂的分支。

|  |  |
| --- | --- |
|  | var varC:Int = 20  if varC < 20 {      print("varC 小于 20")  } else if varA == 20{      print("varC 等于 20")  } else {      print("varC大于 20")  }  //输出:varC 大于等于 20 |

### 3.3.2 switch

switch被称作多分支选择语句或是开关选择语句，通过匹配的方式来选择程序执行的代码块。

switch基本语法：

|  |  |
| --- | --- |
|  | switch 表达式 {  case 表达式：  语句  case 表达式:  语句  default:  语句  } |

switch语句会尝试把某个值与若干个模式(pattern)进行匹配，根据第一个匹配成功的模式，switch语句会执行对应的代码。当有可能的情况较多时，通常用switch语句替换if语句以简化代码，提高可读性。

|  |  |
| --- | --- |
|  | let someCharacter : Character = "A"  switch someCharacter {  case "a","e","i","o","u","A","E","I","O","U":      print("\(someCharacter)是一个元音字母")  default:      print("\(someCharacter)不是一个元音字母")  }  //输出:A是一个元音字母 |

与if语句类似，每一个 case 都是代码执行的一条分支。switch语句必须是完备的，也就是说每一个可能的值都必须至少有一个 case 分支与之对应，必要的时候需要使用默认(default)分支来涵盖case分支无法匹配的情况，并且这个默认分支必须在switch语句的最后面。

switch的case还提供了多种功能更加强大的复杂匹配模式。

|  |  |
| --- | --- |
|  | let time = 11  switch time {      case let x where x >= 0 && x <= 7:          print("睡觉时间")      case 8:          print("起床时间")      case 9...12:          print("上课时间")      default:          print("自由活动")  }  //输出:上课时间 |

在swift中，当匹配的case分支中的代码被执行完毕后，会自动终止switch语句，不会继续执行下一个case分支，不需要像其他语言一样，在 case 分支中显式地使用break语句。如果想要实现贯穿的效果，可以使用fallthrough语句，在控制转移语句中会介绍fallthrough语句。

## 3.3 控制转移语句

控制转移语句可以改变代码的执行顺序，比如提前终止循环结构，也可以人为的控制选择结构的跳转，使代码的执行更加的灵活。本节介绍continue、break、fallthrough和guard。

### 3.4.1 continue

continue语句的作用是告诉程序立刻停止本次循环，重新开始下次循环。continue结束的仅仅是一次循环，而不是整个循环过程。

|  |  |
| --- | --- |
|  | var index = 10  repeat{      index = index + 1      if( index == 12 ){ // index 等于 12 时跳过          continue      }      print(index)  }while index < 15  //输出:11 13 14 15 |

当index等于12的时候，print语句被跳过了，而13 14 15还是照常打印，说明循环并未结束。

### 3.4.2 break

break语句的作用是停止本次循环，并结束整个循环过程。

|  |  |
| --- | --- |
|  | var index = 10  repeat{      index = index + 1      if( index == 12 ){ // index 等于 12 时跳过          break      }      print(index)  }while index < 15  //输出:11 |

将continue改为break，程序就仅仅打印了11，这是因为当index等于12的时候，由于break语句整个循环都被终止了，所以12到15都不会被打印。

### 3.4.3 fallthrough

fallthrough 语句可以让 case 之后的语句按顺序继续运行，且不论条件是否满足都会执行。

在大多数语言中的switch结构中，case 要紧跟 break，否则 case 之后的语句会顺序运行，而在 Swift中，默认是不会执行下去的，switch会终止。如果想在 Swift 中让 case 之后的语句会按顺序继续运行，则需要使用 fallthrough 语句。

如果switch中没有fallthrough语句:

|  |  |
| --- | --- |
|  | let index = 10  switch index {  case 100  :      print( "index 的值为 100")  case 10,15  :      print( "index 的值为 10 或 15")  case 5  :      print( "index 的值为 5")  default :      print( "默认 case")  }  //输出:index 的值为 10 或 15 |

在某些case中添加fallthrough语句:

|  |  |
| --- | --- |
|  | let index = 10  switch index {  case 100  :      print( "index 的值为 100")      fallthrough  case 10,15  :      print( "index 的值为 10 或 15")      fallthrough  case 5  :      print( "index 的值为 5")  default :      print( "默认 case")  }  //输出：  //index 的值为 10 或 15  //index 的值为 5 |

从输出中可以看到，当case 10,15对应的语句块被执行以后，case 5对应的语句块也被执行了。

### 3.4.4 guard

guard意为守护。guard语句判断其后的表达式布尔值为false时，才会执行else后的代码，如果为true，则跳过整个guard语句。guard像守卫一样，在代码中设置检查点，如果满足条件，guard就放行，但是当不满足条件时，进入else语句块。示例代码：

|  |  |
| --- | --- |
|  | func check(age : Int){      guard age >= 18 else {        print("未满18岁")        //执行return语句直接退出  return      }      print("年满18岁")  }  check(age: 9)  //输出：未满18岁  check(age: 20)  //输出：年满18岁 |

本例中的guard被设置为检查年龄。当age进入函数体内部，guard对age进行检查，如果age大于等于18就放行，执行print("年满18岁")，如果不满足大于等于18的条件，进入else语句块，执行print("未满18岁")，并且return退出函数。guard就是扮演这样一个守卫的角色，将小于18的输入拦截住。guard尤其适用于层层检查的情况，能避免if里面嵌套if的糟糕写法。

## 3.4 本章小结

本章向读者介绍了swift中的流程控制语句，包括用于遍历集合类型的for-in语句，控制代码重复执行的while循环以及选择代码不同分支执行的条件语句if和switch。随后介绍了控制代码跳转的控制转移语句，包括continue、break、fallthrough、guard。计算机的强大之处就在于它能高效率的做重复的事情，开发者可以利用这些简单的流程控制语句组合出复杂的程序，实现强大的功能。

# 第四章 元组和集合类型

## 4.1 元组

元组(tuple)把多个值组合成一个复合值。如果定义一个学生，包含2个属性：姓名和年龄，可以用元组表示为：

|  |  |
| --- | --- |
|  | let student = ("Swift", 19)  //student 的类型是 (String, Int)，值是 ("Swift", 19) |

示例中，定义了一个类型为(String, Int)的元组，String表示姓名，Int表示年龄。元组中的值可以是任意类型，根据实际场景下的不同需求任意搭配，比如将元组定义成(String, String,String)用来表示姓名，性别，地址或是(String, Double)用来表示姓名和成绩或是其他任何你想要的组合。

在使用元组之前，需要进行元组分解(decompose)，将元组的内容分解成单独的常量或变量，之后就可以单独使用。

|  |  |
| --- | --- |
|  | let (name, age) = student  print("name is \(name)")  //输出: “name is Swift”  print("age is \(age)")  //输出: “age is 19” |

如果只需要元组的一部分值，分解的时候可以用\_代替被忽略的部分。

|  |  |
| --- | --- |
|  | let (onlyName, \_) = student  print("name is \(onlyName)")  //输出: “name is Swift” |

此外，还可以通过下标直接获得元组的单个数据项，下标从零开始。

|  |  |
| --- | --- |
|  | print("name is \(student.0)")  //输出: “name is Swift”  print("age is \(student.1)")  //输出: “age is 19” |

如果，在元组定义的时候给单个数据项命名，之后就可以用名称直接调用。

|  |  |
| --- | --- |
|  | //定义时给数据项命名  let otherStudent = (name:"iOS", age:18)  //直接用.名称调用  print("name is \(otherStudent.name)")  //输出: “name is iOS”  print("age is \(otherStudent.age)")  //输出: “age is 18” |

作为函数返回值时，元组非常有用。例如，一个用来获取网页的函数可能会返回一个(Int, String)元组，分别代表状态码和信息，用来描述网页是否获取成功。和只能返回一个类型的值比较起来，一个包含两个不同类型值的元组可以让函数更好地返回所需要的信息。

虽然元组在临时组织值的时候很有用，但是并不适合创建复杂的数据结构。如果你的数据结构并不是临时使用，请使用类或者结构体而不是元组。

## 4.2 集合类型和集合可变性

Swift 语言提供数组(Array)、集合(Set)和字典(Dictionary)三种基本的集合类型用来存储集合数据。数组（Array）是有序数据的集。集合（Set）是无序无重复数据的集。字典（Dictionary）是无序的键值对的集。Swift 语言中的Arrays、Sets和Dictionary中存储的数据值类型必须明确。这意味着不能把不同类型的数据插入其中，同时从集合中取出的数据项的数据类型也是固定的。

如果创建一个Array、Set或Dictionary并且把它声明成一个变量，这个集合就是可变的。可以在创建之后添加更多或移除已存在的数据项，或者改变集合中的数据项。如果把Array、Set或Dictionary声明成常量，那么它就是不可变的，它的大小和内容都不能被改变。在不需要对集合做出改变的情况，最好声明成常量，因为编译器会对它进行优化。

## 4.3 数组(Array)

### 4.3.1 数组的创建

数组使用有序列表存储同一类型的多个数据项，每个数据项都有一个编号，编号从0开始依次递增至数组长度减一，并且相同的值可以多次出现在一个数组的不同位置中。Swift 数组会强制检测数据项的类型，如果类型不同会报错。Swift中使用Array<Element>这样的形式来定义数组，其中Element是这个数组中唯一允许存在的数据类型，也可以使用[Element]这样的简单语法，两种形式在功能上是一样的。

|  |  |
| --- | --- |
|  | //创建一个空数组  var intArrary = Array<Int>()  var someInts = [Int]()  print("数组中有\(someInts.count)个数据项")  //输出: "数组中有0个数据项" |

Swift 中的Array类型还提供一个构造方法用于创建只包含重复数据的数组。repeating声明想要重复的数据项，count则声明需要重复的次数。

|  |  |
| --- | --- |
|  | //创建一个[Double]数组，等价于[0.0, 0.0, 0.0]  var threeDoubles = Array(repeating: 0.0, count: 3) |

可以使用数组字面量来进行数组的初始化，这是一种用一个或者多个数值构造数组的简单方法。数组字面量是一系列由逗号分割并由方括号包含的数值：[value 1, value 2, value 3]。这种情况下，编译器能自动推导出数组中的数据类型，无需显示声明。

|  |  |
| --- | --- |
|  | //定义包含3个数据项的[String]数组  var shoppingList = ["Eggs", "Milk", "Bread"] |

### 4.3.2 数组的常用属性和方法

Swift中提供了一些常用的属性和方法，这样可以更加方便的使用数组。

|  |  |
| --- | --- |
|  | //返回数组中数据项的数量  let count = shoppingList.count  //判定数组是否为空  let isEmpty = shoppingList.isEmpty  //判定数组中是否存在某个数据项  let contains = shoppingList.contains("Eggs")  //返回特定数据项的下标，如果有多个重复数据项，返回第一个的下标  let index = shoppingList.index(of: "Eggs")  //返回数组的第一个或最后一个数据项  let firstValue = shoppingList.first  let lastValue = shoppingList.last |

### 4.3.3 数组的遍历

可以使用上一章介绍的for-in循环来遍历数组中的数据项。

|  |  |
| --- | --- |
|  | for index in shoppingList.indices{      print(shoppingList[index])  }  //输出: Eggs Milk Bread |

indices是Array类型的属性，这个属性是一个Range范围，代表着数组下标的范围。利用它可以方便的获取下标，访问数组中的数据项。

或是使用下面这种更加简洁的形式遍历数组中的每个数据项。

|  |  |
| --- | --- |
|  | for item in shoppingList{      print(item)  }  //输出: Eggs Milk Bread |

如果同时需要每个数据项的值和下标值，可以使用enumerated()方法来进行数组遍历。enumerated()返回一个由每一个数据项下标值和数据值组成的元组。可以把这个元组分解成临时常量或者变量来进行遍历。

|  |  |
| --- | --- |
|  | //利用元组访问  for (index, item) in shoppingList.enumerated() {      print("item\(index) \(item)")  }  /\*   输出:：   item0 Eggs item1 Milk item2 Bread  \*/ |

### 4.3.4 数组的读取和修改

在Swift中，可以使用append()或者赋值运算符+=在数组的末尾添加数据项。

|  |  |
| --- | --- |
|  | //为shoppingList添加数据项"Tea"  shoppingList.append("Tea")  //+=将["Flowers","Orange"]添加到shoppingList末端  shoppingList += ["Flowers","Orange"]  //当前数组中的元素:Eggs Milk Bread Tea Flowers Orange |

如果需要在数组的指定位置添加新的数据项，可以使用insert()方法，新的数据项被插入数组指定的index位置，该位置原来的数据项以及后面的数据项将依次右移。

|  |  |
| --- | --- |
|  | //在下标为2的位置插入数据项  shoppingList.insert("Pears", at: 2) |

当然，也可以通过下标的方式访问数据项和修改数据项。

|  |  |
| --- | --- |
|  | //输出:下标为0的数据项  print(shoppingList[0])  //输出: Eggs  //把第0个数据项修改为apples,再次输出:  shoppingList[0] = "Apples"  print(shoppingList[0])  //输出: Apples |

Swift数组提供了几种remove方法，用来删除数组中的数据项。

|  |  |
| --- | --- |
|  | //删除指定下标的数据项  shoppingList.remove(at: 1)  //删除第一个数据项  shoppingList.removeFirst()  //删除最后一个数据项  shoppingList.removeLast()  //删除区间内的数据项  shoppingList.removeSubrange(Range(0...1))  //删除所有数据项  shoppingList.removeAll() |

## 4.4 集合(Set)

集合(Set)用来存储相同类型并且没有顺序的值。集合中不能够存储重复的数据项。

### 4.4.1 集合的创建

Swift 中的Set类型被定义为Set<Element>，这里的Element表示Set中允许存储的类型，和数组不同的是，集合没有等价的简化形式。

|  |  |
| --- | --- |
|  | var letters = Set<Character>()  print("集合中有\(letters.count)个数据项")  //输出: 集合中有0个数据项 |

与数组类似，也可以使用数组字面量来构造集合。

|  |  |
| --- | --- |
|  | // favoriteGenres 被构造成含有三个初始值的集合  var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"] |

Set类型不能从数组字面量中被单独推断出来，因此Set类型必须显式声明。由于Swift 的类型推断功能，如果使用一个数组字面量构造一个Set，那么无需写出Set中的数据项的具体类型。上面例子中的favoriteGenres的构造形式可以采用简化的方式代替。

|  |  |
| --- | --- |
|  | var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"] |

由于数组字面量中的所有数据项类型相同，Swift 可以推断出Set<String>作为favoriteGenres变量的正确类型。

### 4.4.2 集合的遍历

在一个for-in循环中遍历一个Set中的所有值。

|  |  |
| --- | --- |
|  | //for-in遍历favoriteGenres  for genre in favoriteGenres {      print(genre)  }  //输出: Rock Classical Hip hop |

### 4.4.3 集合的修改

通过调用Set的insert(\_:)方法来添加一个新数据项。

|  |  |
| --- | --- |
|  | favoriteGenres.insert("Jazz")  //favoriteGenres 现在包含4个数据项 Rock Classical Hip hop Jazz |

可以通过调用Set的remove()方法去删除一个数据项，如果该值是该Set的一个数据项则删除该数据项并且返回被删除的数据项值，如果该Set不包含该值，则返回nil。另外，Set中的所有数据项可以通过它的removeAll()方法删除。

|  |  |
| --- | --- |
|  | if let removedGenre = favoriteGenres.remove("Rock") {      print("成功删除\(removedGenre)")  } else      print("集合中不包含该数据项")  }  //输出: 成功删除Rock |

### 4.4.4 集合的重组和组合

有些方法能够高效地完成Set的一些基本操作，比如把两个集合组合到一起，判断两个集合共有数据项，或者判断两个集合是否全包含，部分包含或者不相交。常用方法：

intersection()方法根据两个集合中都包含的值创建的一个新的集合。

symmetricDifference()方法根据在一个集合中但不在两个集合中的值创建一个新的集合。

union()方法根据两个集合的值创建一个新的集合。

subtracting()方法根据不在该集合中的值创建一个新的集合。

|  |  |
| --- | --- |
|  | let oddDigits: Set = [1, 3, 5, 7, 9]  let evenDigits: Set = [0, 2, 4, 6, 8]  let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]  oddDigits.union(evenDigits).sorted()  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  oddDigits. intersection(evenDigits).sorted()  // []  oddDigits.subtracting(singleDigitPrimeNumbers).sorted()  // [1, 9]  oddDigits. symmetricDifference(singleDigitPrimeNumbers).sorted()  // [1, 2, 9] |

## 4.5 字典（Dictionary）

在查阅英汉字典的时候，通过英文单词可以找到对应的中文解释。这种通过键（英文单词）查找值（中文解释）的存储模式称为键值对映射，Swift中的字典(Dictionary)就是利用这一模式。

字典(Dictionary)存储键值对，字典中的值(Value)可以有重复，而键(Key)必须是唯一的，每个值都有一个对应的键，在使用字典时是通过键确定值，只有键唯一才能确保找到一个确定的值，当然如果开发者尝试设置重复的键，编译器会报错。

### 4.5.1 字典的创建

Swift 的字典使用Dictionary<Key, Value>来定义，其中Key是字典中键的数据类型，不可重复，Value是字典中对应于这些键所存储值的数据类型。也可以用[Key: Value]这样简化的形式去创建一个字典类型。

|  |  |
| --- | --- |
|  | var diallingCode = [String: String]()  // diallingCode 是一个空的 [String: String] 字典 |

还可以使用字典字面量来构造字典，这和刚才介绍过的数组字面量拥有相似语法。字典字面量是一种将一个或多个键值对写作Dictionary集合的快捷方式。

一个键值对是一个key和一个value的结合体。在字典字面量中，每一个键值对的键和值都由冒号(:)分割。这些键值对构成一个列表，被方括号包含、由逗号分割。

|  |  |
| --- | --- |
|  | var diallingCodes = ["010":"北京", "021":"上海", "0571":"杭州"] |

### 4.5.2 字典的遍历

可以使用for-in循环，来遍历某个字典的键或值，还可以通过元组同时遍历字典中的键值对。

|  |  |
| --- | --- |
|  | //遍历diallingCodes的keys  for key in diallingCodes.keys {      print(key)  }  //输出: 010 021 0571  //遍历diallingCodes的values  for value in diallingCodes.values {      print(value)  }  //输出: 北京 上海 杭州  for (key,value) in diallingCodes {      print("\(key) : \(value)")  }   //输出: 021 : 上海 010 : 北京 0571 : 杭州 |

### 4.5.3 数组的读取和修改

可以使用下标语法，通过key获取字典中对应的value。

|  |  |
| --- | --- |
|  | //输出: key "010" 对应的value  let cityName = diallingCodes["010"]  print(cityName!)  //输出: 北京 |

通过下标语法，还可以往字典中添加新的数据项。

|  |  |
| --- | --- |
|  | //向字典中添加 key="020", value="广东"  diallingCodes["020"] = "广东"  print(diallingCodes["020"]!)  //输出: 广东 |

如果下标中的key已经存在于字典中，新设置的value会覆盖旧value。

|  |  |
| --- | --- |
|  | //再次设置key="020"的value  diallingCodes["020"] = "深圳"  print(diallingCodes["020"]!)  //输出: 深圳 |

也可以使用字典的updateValue()方法，修改指定key对应的value。

|  |  |
| --- | --- |
|  | //利用updateValue更新key="020"对应的value  diallingCodes.updateValue("武汉", forKey:"020" )  print(diallingCodes["020"]!)  //输出: 武汉 |

当需要删除字典的某对键值时，可以使用removeValue()。将某个key对应的value设置成nil，也能起到相同的作用。以下两行代码，都能在字典中删除key为“010“的键值对。

|  |  |
| --- | --- |
|  | //某个值设置为nil  diallingCodes["010"] = nil  //调用removeValue  diallingCodes.removeValue(forKey: "010")  Swift还提供了remoeAll()方法，用于一次性删除字典中的所有键值对。  diallingCodes.removeAll()  print("字典中还剩\(diallingCodes.count)数据项")  //输出: 字典中还剩0数据项 |

## 4.6 本章小结

本章向读者介绍了元组，它作为临时的数据结构是个不错的选择。接下来介绍了Swift中的3种集合类型，分别是数组(Array)、集合(Set)和字典(Dictionary)，重点是介绍这些数据类型的定义、遍历、修改，它们作为开发中的常用工具，非常重要，需要熟练掌握。

# 第五章 函数与闭包

## 5.1函数

函数是人们为了减少代码编写量和时间，以及使代码结构鲜明、清晰易懂而提出的概念。使用函数可以把有特定功能的代码封装起来，然后在多处使用，这样方便快捷且代码简洁。本节将介绍使用swift语言如何来编写及调用函数。

### 5.1.1函数的声明和使用

函数的基本语法形式如下：

|  |  |
| --- | --- |
|  | func 函数名（参数列表...）->返回值类型...{      语句...  } //其中...表示其数量可以为任意非负整数 |

函数名是一个标识符，一般表示此函数的功能；参数列表由参数名和数据类型组成，使用参数是为了接受调用函数时传递的参数；’->’后面跟着的是函数的返回值类型，函数可以有返回值，也可以没有返回值。函数如果以参数的角度分类可分为无参函数和有参函数；如果以返回值的角度来分类的话可分为无返回值的函数和有返回值的函数。下面会依次讨论不同种类函数的声明和使用。

（1）无参函数

无参函数就是没有参数列表的函数。无参函数声明定义的一般形式如下：

|  |  |
| --- | --- |
|  | func 函数名（）->返回值类型{      语句  } |

函数调用的形式如下：

|  |  |
| --- | --- |
|  | 函数名（） |

【例 5-1】以下定义一个无参函数printHelloSwift（）并调用它，此函数用来输出字符串“ Hello Swift!”,代码如下：

|  |  |
| --- | --- |
|  | func printHelloSwift(){              //声明定义函数  print(“Hello Swift!”)  }  printHelloSwift()            //调用函数 |

运行结果为：

|  |  |
| --- | --- |
|  | Hello Swift! |

（2） 有参函数

所谓有参函数，就是函数列表不为空的函数，参数数量可以是一个或多个。有参函数声明定义的一般形式如下：

|  |  |
| --- | --- |
|  | func 函数名（参数名1：数据类型，参数名2：数据类型…）->返回值类{      语句  } |

其中参数名和数据类型用‘：’号来隔开，不同的参数用‘，’号来隔开。其调用的形式如下：

|  |  |
| --- | --- |
|  | 函数名（参数名1：参数值1，参数名2：参数值2…） |

其中参数值和参数名的类型一定要一致。

【例 5-2】 以下定义一个sum函数来实现2个数的相加，代码如下：

|  |  |
| --- | --- |
|  | func sum(value1:Int,value2:Int){      print(value1+value2)  }  sum(value1:20,value2:10) |

运行结果为：

|  |  |
| --- | --- |
|  | 30 |

（3）无返回值的函数

函数是可以没有返回值的，其声明定义的形式如下：

|  |  |
| --- | --- |
|  | func 函数名（参数列表...）{      语句...  } |

函数调用的形式如下：

|  |  |
| --- | --- |
|  | 函数名（参数列表） |

（4） 具有一个返回值的函数

在函数中返回一个值是最常见的。函数返回值的类型必须在函数声明定义时为函数设定，并用return语句进行返回。这种函数的声明定义形式如下：

|  |  |
| --- | --- |
|  | func 函数名（参数表）->返回值类型{  语句  return 表达式  } |

返回的表达式类型必须与函数的返回值类型一致。

【例 5-3】 以下定义一个sum1函数来实现2个数的相加，代码如下：

|  |  |
| --- | --- |
|  | func sum1(value1:Int,value2:Int)->Int{      return value1+value2  }  print(sum1(value1:20,value2:10))  //直接调用  let d=sum1(value1:20,value2:20)   //赋值  print(d) |

运行结果为：

|  |  |
| --- | --- |
|  | 30  40 |

（5）具有多个返回值的函数

函数返回多个返回值，就需要使用到元组类型。这种函数的声明定义形式如下：

|  |  |
| --- | --- |
|  | Func5 函数名（参数列表）->(数据类型1，数据类型2，数据类型3...){  语句  return （表达式1，表达式2，表达式3...）  } |

数据类型之间用‘，’号来隔开，return中的表达式的类型要与返回值的数据类型一一对应。

【例 5-4】 以下定义一个sum2函数来实现2个数和2个字符串的的相加，代码如下：

|  |  |
| --- | --- |
|  | func sum2(value1:Int,value2:Int,str1:String,str2:String) -> (Int,String){  //2个数和2个字符串分别相加      let k=value1+value2      let l=str1+str2      return (k,l)  }  print(sum2(value1: 5, value2: 1, str1: "hello", str2: "swift")) |

运行结果为：

|  |  |
| --- | --- |
|  | (6, "helloswift") |

### 5.1.2 关于函数参数的使用

如果开发者使用的函数为有参函数时，对于参数需要了解以下几点。

（1）关于函数参数名

有时参数的参数名会有两个，这两个函数参数名分为外部参数名和本地参数名。本地参数名用于声明函数时在函数内部使用，外部参数名用于调用该参数时使用。外部参数名是为了让函数中的参数明确和便于理解。定义函数时，外部参数名需写在本地参数名之前，并用空格来隔开。声明定义的一般形式如下：

|  |  |
| --- | --- |
|  | func 函数名（外部参数名 本地参数名：数据类型）->返回值类型{  ……  } |

调用的一般形式如下：

|  |  |
| --- | --- |
|  | 函数名（外部参数名：参数值） |

（2）忽略参数名

如果开发者想在调用函数时忽略参数名，可以使用下划线 ’\_’ 代替外部参数名来实现。它的一般形式如下：

|  |  |
| --- | --- |
|  | func 函数名（\_ 参数名1：数据类型，\_ 参数名2：数据类型…）->返回值类型{  语句  } |

【例 5-5】 以下定义一个sum3函数来实现2个数的相加，代码如下：

|  |  |
| --- | --- |
|  | func sum3(\_ value1:Int,\_ value2:Int)->Int{      return value1+value2  }  print(sum3(10,20)) //省略参数名 |

运行结果为：

|  |  |
| --- | --- |
|  | 30 |

### 5.1.3 为参数设置默认值

开发者可以在函数体中为每个参数定义默认值。当默认值定义后，调用这个函数时可以忽略这个参数，系统会把默认值传给此参数。

【例 5-6】 以下定义一个sum4函数来实现求和运算，代码如下：

|  |  |
| --- | --- |
|  | func sum4(value1:Int=20,value2:Int)->Int{ //参数有默认值      return value1+value2  }  print(sum4(value2:20)) |

运行结果为：

|  |  |
| --- | --- |
|  | 40 |

### 5.1.4 可变参数

可变参数可以接受零个或多个指定类型的值。函数调用时，可以用可变参数来指定函数参数，其数量是不确定的。

可变参数通过在数据类型名后面加入 ‘...’ 的方式来定义。

【例 5-7】 以下定义一个sum5函数来实现求和运算，代码如下：

|  |  |
| --- | --- |
|  | func sum5(numbers:Double...){       //可变参数      var total:Double=0.0      //求和      for number in numbers{          total+=number      }      print(total)  }  sum5(numbers: 1.1,2.22,4) |

运行结果为：

|  |  |
| --- | --- |
|  | 7.32 |

### 5.1.5输入输出参数

一般默认在函数中定义的参数都是常量参数，也就是这个参数只可以查询使用，不能改变它的值。 如果想要改变参数的值，且参数在函数调用后其改变仍有效，可以在参数名后加 ‘inout’ 关键字，这样就可以改变这个参数的值了。它的一般形式如下：

|  |  |
| --- | --- |
|  | func 函数名（参数名：inout 数据类型，...）->返回值类型{      语句  } |

注意：使用inout关键字的参数必须是已经声明过的变量。

函数的调用形式如下：

|  |  |
| --- | --- |
|  | 函数名（&参数，...） |

【例 5-8】 以下定义一个sum6函数来实现求和运算，代码如下：

|  |  |
| --- | --- |
|  | func sum6(value1:inout Int,\_ value2:Int){  //输入输出参数      value1 = value2+value1  }  var s = 20  print("没加任何值的s为\(s)")  sum6(value1: &s, 20)  print("加了值后的s为\(s)") |

运行结果为：

|  |  |
| --- | --- |
|  | 没加任何值的s为20  加了值后的s为40 |

### 5.1.6函数类型

在swift中，每一个函数都有一个特定的类型，这种类型称为函数类型。函数类型由函数的参数类型和返回值类型组成。例如【例 5-5】中的函数：

|  |  |
| --- | --- |
|  | func sum3(\_ value1:Int,\_ value2:Int)->Int{      return value1+value2  } |

此函数的类型就是（int,int）->int。对于没有返回值的无参函数来说，它的类型为()->()。

Swift语言中可以使用函数类型来定义其他常量/变量，就像使用其他类型一样。函数类型使用的一般形式如下：

|  |  |
| --- | --- |
|  | let/var 常量名/变量名：函数类型=函数名 |

或者：

|  |  |
| --- | --- |
|  | let/var 常量名/变量名=函数名 |

【例 5-9】 以下把sum7函数的类型赋值给变量来实现求和运算，代码如下：

|  |  |
| --- | --- |
|  | func sum7(value1:Int,value2:Int)->Int{      return value1+value2  }  let add:(Int,Int)->Int=sum7  print(add(20, 30))  print(sum7(value1: 40, value2:20)) |

运行结果为：

|  |  |
| --- | --- |
|  | 50  60 |

函数类型可以作为函数参数来使用。

【例 5-10】 以下把上例sum7函数的类型作为函数参数来使用，代码如下：

|  |  |
| --- | --- |
|  | func sum8(value1:Int,value2:Int)->Int{      return value1+value2  }  func adder(fun:(Int,Int)->Int,a:Int,b:Int)->Int{      let x=fun(a,b)      return x  }  print(adder(fun: sum8, a: 20, b: 60)) |

运行结果为：

|  |  |
| --- | --- |
|  | 80 |

其中adder函数中的fun只是一个参数名，其类型为(Int,Int)->Int，所有这个类型的函数都能成为adder函数的参数。从这个例子可以看出，函数类型作为函数参数与数据类型作为函数参数是一样的。

函数类型还可以作为返回类型来使用。

【例 5-11】 以下把函数类型作为返回值并使用，代码如下：

|  |  |
| --- | --- |
|  | func mult(a:Int,b:Int)->Int{      return a\*b  }  func sum9(value1:Int,value2:Int)->Int{      return value1+value2  }  func numchange(c:Int,d:Int)->(Int,Int)->Int{      return c>d ? sum9:mult  }  let num=numchange(c: 40, d: 50)  print(num(5,5)) |

运行结果为：

|  |  |
| --- | --- |
|  | 25 |

### 5.1.7函数的嵌套

函数的嵌套分为嵌套函数和函数的嵌套调用2种。嵌套函数指的是函数内定义的一个新的函数，定义它的函数可以调用该函数。嵌套调用的形式是在一个函数中调用其他函数，或者调用自身函数。调用自身的嵌套调用我们称之为递归调用。

1.嵌套函数

默认情况下，嵌套函数只能在父函数中调用，在父函数外部无法被调用。定义嵌套函数的父函数也可以返回它的某一个嵌套函数，使得这个函数可以在其他域中被使用。

【例 5-12】 以下在函数中声明定义嵌套函数并调用，代码如下：

|  |  |
| --- | --- |
|  | func isDivisible (\_ a:Int,\_ b:Int)->()->String{  func divisibled()->String{return "\(b)可以被\(a)整除。"}  func undivisibled()->String{return "\(b)不可以被\(a)整除。"}      return b%a==0 ? divisibled:undivisibled  }  let x=isDivisible(3, 46) //x是一个函数  let result=x() //调用x函数  print(result) |

运行结果为：

|  |  |
| --- | --- |
|  | 46不可以被3整除。 |

2.嵌套调用

嵌套调用是一个函数调用一个或多个函数（不是在自己内部定义的）。其基本形式如下：

|  |  |
| --- | --- |
|  | func 函数名0（参数列表）->返回值数据类型{  ......  }  func 函数名1（参数列表）->返回值数据类型{  ......  调用函数0  ......  }  func 函数名2（参数列表）->返回值数据类型{  ......  调用函数1  ......  } |

递归调用属于嵌套调用的一种特殊方式。它在调用函数时调用了该函数本身。其形式如下：

|  |  |
| --- | --- |
|  | func 函数名1（参数列表）->返回值数据类型{  ......  调用函数1  ......  } |

【例 5-13】 以下通过递归调用来计算5！的值：

|  |  |
| --- | --- |
|  | func fac(value:Int)->Int{       //递归调用      if(value==1){          return value      }else{          return value\*fac(value: value-1)      }  }  print(fac(value: 5)) |

运行结果为：

|  |  |
| --- | --- |
|  | 120 |

## 5.2 闭包

闭包可以看做无参数名的函数，但它不是函数。函数实际上是特殊的闭包。闭包能够像函数一样根据语句实现某种特定功能，也可以捕获和存储上下文中变量的引用。

本章主要讲解swift语言中闭包的声明定义和使用、尾随闭包和捕获值等的相关知识。

### 5.2.1闭包表达式

闭包表达式的基本语法形式如下：

|  |  |
| --- | --- |
|  | {(参数列表) -> 返回值类型 in  语句  } |

定义的语法形式如下：

|  |  |
| --- | --- |
|  | let/var 闭包表达式常量名称/闭包表达式常量名称=具有参数的闭包表达式 |

调用形式如下:

|  |  |
| --- | --- |
|  | 闭包表达式常量名称/闭包表达式变量名称（参数值1，参数值2...） |

其中参数列表和函数的一样可以是常量、变量、输入输出函数，但是不能有默认值。参数列表和返回值类型和函数一样可有可无，多个参数和返回值类型的声明定义的形式也和函数一样。关键字 ‘in’ 后面跟着的是闭包表达式的语句，相当于函数的语句部分。闭包表达式声明定义的同时一定要赋值给一个常量或变量，这就相当于给了一个函数名。其调用形式就如函数一样了。

【例 5-14】 以下实现两个整数的相除和输出字符串“Swift”，代码如下：

|  |  |
| --- | --- |
|  | //有参有返回值闭包  let divide = {(val1: Int, val2: Int) -> Int in      return val1 / val2  }  let res = divide(200, 20)  print (res)  //无参无返回值闭包  let printSwift={() in      print("Swift")  }  printSwift() |

运行结果为：

|  |  |
| --- | --- |
|  | 10  Swift |

闭包表达式可以用于其他函数中，可以在函数中先声明其类型，在调用该函数时再定义闭包。

【例 5-15】 以下将闭包表达式作为函数的一部分，实现判断在数组中是否存在某个值的公倍数。代码如下：

|  |  |
| --- | --- |
|  | func hasMultiple(arr:[Int],\_ value:Int,\_ cm:(\_ num:Int,\_ value:Int) ->Bool)->Bool{      //闭包作为参数      for item in arr{          if(cm(item,value)){              return true              }          }      return false  }  let ray=[20,80,60,100]  var v=hasMultiple(arr: ray,16,{(num,value)->Bool in return num%value==0  })  print(v) |

运行结果为：

|  |  |
| --- | --- |
|  | true |

### 5.2.2闭包表达式的优化

以下都以【例5-13】为例：

(1)推断类型和省略return

hasMultiple()函数的第三个参数是闭包表达式，它的类型为 ’(Int,Int) ->Bool’。闭包可以根据上下文推断其参数和返回值的类型，所以->和数值类型以及返回值类型都可以省略；如果闭包表达式的语句的代码只有一行，那么也可以省略return，如以下代码所示：

|  |  |
| --- | --- |
|  | var v=hasMultiple(arr: ray,16,{(num,value)  in num%value ==0}) |

（2）简写或省略参数名

开发者可以通过$0,$1,$2来顺序的调用闭包的参数。

|  |  |
| --- | --- |
|  | var v=hasMultiple(arr: ray,16,{$0%$1==0}) |

闭包的函数体很短时可以写成一行，所以写成一行的写法并没有错。

### 5.2.3尾随闭包

当闭包作为一个函数的参数，且是最后一个参数时，可以使用尾随闭包。尾随闭包（又称Trailing闭包）是一个书写在函数括号之后的闭包表达式，这是因为有的闭包表达式很长，用尾随闭包可以增强函数的可读性。以【例 5-13】为例，其函数的调用可写成：

|  |  |
| --- | --- |
|  | var v=hasMultiple(arr: ray, 16){$0%$1==0} |

当函数只有闭包这一个参数时，可以把括号也省略掉。

【例 5-16】用sorted函数配合闭包完成排序。

|  |  |
| --- | --- |
|  | let number=[1,55,26,8,46,20]  var reverse = number.sorted() { $0 > $1 }  var reverse1 = number.sorted { $0 > $1 }  print(reverse)  print(reverse1) |

运行结果为：

|  |  |
| --- | --- |
|  | [55, 46, 26, 20, 8, 1]  [55, 46, 26, 20, 8, 1] |

### 5.2.4 捕获值

闭包可以捕获其被定义的上下文的常量或变量的值，实际上相当于使用闭包把这些常量和变量存起来。在常量或变量的原作用域不存在的情况下，仍然能够在闭包函数体内修改这些值。闭包捕获值最简单的应用形式就是嵌套函数。嵌套函数可以捕获其父函数定义的所有参数、常量和变量。

【例 5-17】以下定义一个嵌套函数，该函数将会捕获其父函数的参数和变量，代码如下：

|  |  |
| --- | --- |
|  | func numberOfMultiply(multiplier num:Int) -> () -> Int {      var result = 3      func multiply() -> Int {          result \*= num          return result      }      return multiply  }  let mul = numberOfMultiply(multiplier: 2)  print(mul())  print(mul()) |

运行结果为：

|  |  |
| --- | --- |
|  | 6  12 |

## 5.3 本章小结

本章主要介绍了函数和闭包的有关内容，重点介绍了函数的生声明和使用以及闭包的生命和使用。本章需要重点掌握函数的声明和调用方式，闭包表达式的声明和调用，熟练应用函数与闭包的各种功能。

# 第六章 枚举与结构体

枚举也是一种数据类型，一般枚举的作用是将事物的选择限定到一个范围中。如一星期只有七天，在编程中就可以用枚举来限定星期数。

## 6.1 枚举类型的创建与应用

swift中使用枚举类型，主要分为两种，一种是创建任意类型的枚举类型，另外一种是创建指定数据类型的枚举类型。以下是这两种枚举类型的定义：

1、任意枚举类型

enum 枚举名{

}

任意枚举类型在枚举的相关值中用的比较多，一般就是不为枚举类型设置数据类型。

2、指定枚举类型

enum 枚举名:数据类型{

}

指定枚举类型常见于枚举的原始值，一般就是为枚举类型设置数据类型，来限定枚举中枚举成员的数据类型。

例如我们定义表示星期的枚举：

|  |  |
| --- | --- |
|  | enum week{  case SUNDAY  case MONDAY  case TUESDAY  case WEDNESDAY  case THURSDAY  case FRIDAY  case SATURDAY  }  var weekDay = week.SUNDAY  weekDay = .MONDAY  switch weekDay{  case .SUNDAY: print("星期天")  case .MONDAY: print("星期一")  case .TUESDAY: print("星期二")  case .WEDNESDAY: print("星期三")  case .THURSDAY: print("星期四")  case .FRIDAY: print("星期五")  case .SATURDAY: print("星期六")  default: print("没有匹配值")  } |

以上程序执行输出结果为：

|  |  |
| --- | --- |
|  | 星期一 |

在示例中，需要注意到几个点，如果一个变量的类型已经确认为某个枚举类型，那么开发者在进行变量赋值的时候，是可以直接将枚举名省略掉，直接使用点语法获取枚举值即可。

## 6.2枚举的原始值与相关值

枚举的原始值就是给定义的枚举成员再赋上一个值。对于枚举类型是整数型，默认的原始值是从零开始逐渐递增，而枚举类型是其他类型的，需要开发者指定初始值。

### 6.2.1枚举的原始值

前面创建的枚举中，并没有为枚举类型初始化一个值，而枚举的原始值就是在初始化枚举类型时，为其附上一个初始值。示例如下：

|  |  |
| --- | --- |
|  | //为枚举类型指定一个原始值  enum original:Int{  case a = 1  case b = 2  case c = 3  case d = 40  } |

通过枚举类型提供的rawValue属性来获取枚举的原始值，示例如下：

|  |  |
| --- | --- |
|  | //创建枚举变量  var num = original.a  //获取num枚举变量的原始值“a”  var rawValue = num.rawValue |

### 6.2.2枚举的相关值

在Swift语言中，枚举的相关值在枚举中可以声明为不同的数据类型，且值不是事先设定的。相关值是当你在创建一个基于枚举成员的新常量或变量时才会被设置，即每次值都可以不同。

以下实例中定义一个名为Student的枚举类型，它包含名为Name的一个相关值 (String),和名为Score的一个相关值(Int,Int,Int)。示例如下：

|  |  |
| --- | --- |
|  | enum Student{  case Name(String)  case Score(Int,Int,Int)  }  var stuName = Student.Name("HDU")  var studScore = Student.Score(10,10,10)  switch studScore{  case .Name(let studName):  print("学生的名字:\(studName)")  case .Score(let Chinese1,let Math2,let English3):  print("学生的成绩:\(Chinese1),\(Math2),\(English3)")  } |

以上程序执行输出结果为：

|  |  |
| --- | --- |
|  | 学生的成绩是：10，10，10 |

## 6.3 Swift结构体

在Swift语言中，结构体与类的使用方法特别的相似。

与其他的语言不同的是，Swift语言的结构体不仅仅只是可以定义属性，而且还可以添加方法，与类的功能特别的相似。

通过关键字struct来定义结构体：

struct nameStruct{

属性

方法

}

实例如下：

|  |  |
| --- | --- |
|  | struct dog{  var name:String  var color:String  func bark(){  print("汪汪")  }  } |

从以上的实例可以看出，Swift中的结构体与类的用法特别的相似，但是结构体和类还是有本质区别的。

在结构体中，数据的传递是以复制的方式进行传递的。虽然结构体中能够声明方法，但是结构体没有继承和多态的性质。

## 6.4本章小节

本章主要给读者介绍了Swift语言中的枚举和结构体，相对与其他语言中的枚举类型，Swift中的枚举显得比较灵活。而对于结构体来说是比较大的变化，结构体中能够声明方法，使其许多的应用上和类非常的相似。

# 第七章 类

## 7.1 类的概念

Swift类是构建代码所用的一种通用且灵活的构造体。可以为类定义属性和方法。

现实世界中的一切事物都可以被看作对象，而类则是对象的抽象。对象是具体的，而类是抽象的。可以把类看作对象的模板，根据类创建的对象被称作该类的一个实例。

举个例子，如果要买一台电脑，你会如何考虑？你可能会考虑品牌、CPU、显卡、内存、价格等因素，此时你心中的电脑就是类，它是抽象的。随后你买了一台真实的电脑，它的CPU是i7，显卡是GTX1070，内存16G。这里的电脑就是对象，它是具体的，是根据“电脑”这个抽象概念制造出来的。除了电脑你可能还买了鼠标，此时你桌上的鼠标和电脑都被可以称作对象，不过桌上的鼠标是“鼠标类”的一个实例，而桌上的电脑是“电脑类”的一个实例。电脑的CPU型号，显卡型号，内存大小被称作属性，用电脑浏览网页，看电影，玩游戏被称为动作，也就是方法。

Swift类中的属性包括存储属性和计算属性，方法包括实例方法和类型方法。

### 7.1.1 类的定义

在Swift中，定义类的关键字为class，基本语法：

|  |  |
| --- | --- |
|  | class ClassName {  //定义属性和方法  } |

在定义类的时候，类名应使用UpperCamelCase（首字母大写的驼峰命名法，如SomeClass）来命名，这是一种规范，Swift中的类都采用这种命名方式。而属性名，方法名应采用lowerCamelCase（首字母小写的驼峰命名法，如someAttribute）来命名，以便和类名更好的区分。

|  |  |
| --- | --- |
|  | class Student{      var name:String = "Tom"        func sayHi(){          print("hi! I'm \(name)")      }  } |

本例中，创建了Student类，并定义了一个属性和一个方法。属性name类型为String，默认值设置为Tom，swift中的存储属性，必须要进行初始化，这里采用赋初值的方式。随后定义了sayHi方法，打印一句话。

|  |  |
| --- | --- |
|  | //通过类创建对象  var stu = Student()  stu.sayHi()  stu.name = "Jack"  stu.sayHi() |

程序输出：

|  |  |
| --- | --- |
|  | hello! I'm Tom  hello! I'm Jack |

首先创建了一个对象stu，这个对象被称作Student类的实例，然后调用了sayHi方法，由于Student默认的name是Tom，所以打印出hello! I'm Tom。修改stu的name属性为Jack，再次调用sayHi，打印出hello! I'm Jack。

### 7.1.2 构造过程

除了设置默认值的方式外，还可以采用构造器对类的存储属性进行初始化。可以将构造器看作创建特定类型的实例的特殊方法。基本语法如下：

|  |  |
| --- | --- |
|  | init(){  //初始化  } |

构造器在创建类的实例的时候被调用，在构造器中，完成对存储属性的初始化。类中自定义的构造器会覆盖系统默认的构造器。

|  |  |
| --- | --- |
|  | class Car{      var brand:String      var speed:Int      init(){          brand = "benz"          speed = 60      }  } |

本例中，为Car类定义了两个存储属性brand和speed，并定义了一个无参的构造器init，在其中对存储属性进行初始化。

当然也可以在定义构造器init()时提供构造参数，为Car类继续添加一个包含参数的构造器。

|  |  |
| --- | --- |
|  | class Car{      var brand:String      var speed:Int      init(){          brand = "benz"          speed = 60      }      init(brand:String, speed:Int){          self.brand = brand          self.speed = speed      }      func drive() {          print("brand is \(brand), speed is \(speed)")      }  } |

这样，Car类就有了两个构造器，使用类名( )的形式可以创建对象。

|  |  |
| --- | --- |
|  | var car = Car()  car.drive()  var car2 = Car(brand: "BMW", speed: 70)  car2.drive() |

程序输出：

|  |  |
| --- | --- |
|  | brand is benz, speed is 60  brand is BMW, speed is 70 |

调用包含参数的构造器创建car2，直接将brand设置为BMW，将speed设置为70。

如果存储属性在逻辑上可以为空，就需要将它定义为可选类型optional，当存储属性被声明为可选类型时，将自动被设置成为nil。

|  |  |
| --- | --- |
|  | class SurveyQuestion {      var text: String      var response: String?      init(text: String) {          self.text = text      }      func ask() {  print(text)  print("response:\(response)")      }  } |

本例中，response被声明为可选类型，并且没有初始化。

|  |  |
| --- | --- |
|  | var sq = SurveyQuestion(text: "Are you ok?")  sq.ask() |

程序输出：

|  |  |
| --- | --- |
|  | text:Are you ok?  response:nil |

在没有初始化response的情况下，response被默认设置成了nil。

### 7.1.3 类的引用特征

在swift中，元组、枚举和结构体属于值类型，而类是引用类型。

值类型最显著的特征就是深拷贝(Deep Copy)，也就是复制的时候，传递的是整个对象，每次复制都会在栈内存中开辟一块新的独立的空间，存储拷贝后的副本。

与值类型不同的是，引用类型，在复制的时候，传递的是栈内存中的引用，可以将其看作指针，原对象和副本都指向堆内存的同一块区域，如果通过其中之一对堆内存进行修改，两者都会受影响。通过一个例子，展示一下引用类型的特征。定义一个简单的类，并打印。

|  |  |
| --- | --- |
|  | class Dog{      var name = "haha"  }  var dog1 = Dog()  print("dog1:\(dog1.name)") |

程序输出：

|  |  |
| --- | --- |
|  | dog1:haha |

此时，正确打印dog1的名字haha。如果创建dog2，并将dog1的值赋给dog2，然后修改dog2的name为xixi，分别打印dog1和dog2的name。

|  |  |
| --- | --- |
|  | var dog2 = dog1  dog2.name = "xixi"  print("dog1:\(dog1.name)")  print("dog2:\(dog2.name)") |

程序输出：

|  |  |
| --- | --- |
|  | dog1:xixi  dog2:xixi |

在修改了dog2的name的时候，dog1的name也被修改了。这说明实际上dog1和dog2指向的是同一个对象，换话说，dog2只是dog1的别名。它们的关系可以表示成下图7-1所示。

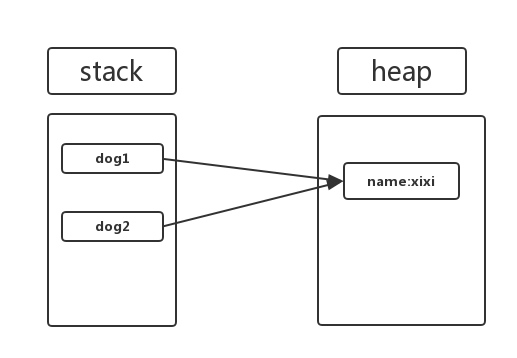


图7-1

### 7.1.4 恒等运算符

上节介绍了类是引用类型，必然出现多个变量或常量引用同一个实例的情况，为了判断两个常量或变量是否引用同一个实例，Swift提供了两个运算符。

|  |  |
| --- | --- |
| 恒等运算符 | 不恒等运算符 |
| ===（三个等号） | ！==（！+两个等号） |
| 如果两个常量或变量引用同一个实例，返回true，否则返回false。 | 如果两个常量或变量不是引用同一个实例，返回true，否则返回false。 |

### 7.1.5 析构过程

Swift通过自动引用计数器(ARC)处理实例的内存管理，Swift会自动释放没有被引用的实例以回收资源，一般情况下不需要手动清理。但是，如果在代码中，引用了外部资源，如读文件，那就需要用到析构方法，在实例被释放之前，做一些额外的工作。

析构方法的基本语法和构造方法类似，但是析构方法不能带任何参数，析构方法由系统调用。

|  |  |
| --- | --- |
|  | Deinit {  //执行析构过程  } |

例如，如果创建了一个自定义的类来打开一个文件，并写入若干数据，就需要在类的实例被释放前关闭该文件。

|  |  |
| --- | --- |
|  | class FileManager {      func loadFile(){          print("Load file from the resource")      }      func writeFile(){          print("Write the data into the file")      }      deinit {          print("Delete the fileManager")      }  } |

在第一行代码中，定义了一个名为FileManager的类。

接着定义了2个实例方法，分别打印对应的一句话，代表完成了某些操作。

最后添加了一个deinit析构函数。当FileManager类的实例的引用计数器为0时，系统会自动调用该析构函数。

|  |  |
| --- | --- |
|  | var fileManager:FileManager? = FileManager()  fileManager!.loadFile()  fileManager!.writeFile()  fileManager = nil |

这里，创建了一个FileManager类的实例，然后依次调用loadFile和writeFile实例方法，最后将fileManager设置为nil，将该实例的引用计数器设置为0，从而使系统调用析构函数，最终的打印结果为：

|  |  |
| --- | --- |
|  | Load file from the resource  Write the data into the file  Delete the fileManager |

## 7.2 类的属性

### 7.2.1 存储属性和计算属性

属性可以分为存储属性和计算属性。

存储属性的作用是存储，简单来说，一个存储属性就是存储在特定类或结构体的实例里的一个变量或常量，变量存储属性就用var定义，常量存储属性用let定义。

计算属性计算一个值，不能直接存储值，而是提供一个getter来取值，一个可选的setter来间接设置其他属性或变量的值。

本章之前的属性都属于存储属性，下面用一个例子介绍计算属性。

|  |  |
| --- | --- |
|  | class Square {      var side:Double        var area:Double {          get{              return self.side \* self.side          }          set(newArea){              self.side = sqrt(newArea)          }      }        init(side: Double) {          self.side = side      }  } |

在这段代码中，定义了一个Square类。它有两个属性，边长和面积。其中，边长（side）是存储属性，而面积（area）是计算属性。在get方法中，将area的值赋值为side \* side，在set方法中，同时设置side的值为sqrt(area)，也就是area开二次方。

|  |  |
| --- | --- |
|  | var square = Square(side: 10)  print(square.area)  //打印 100.0 |

第1行，定义了一个边长为10的正方形，接着打印正方形的面积，自动调用了area的get方法，正确打印出100.0，如果修改一下square的area的值。

|  |  |
| --- | --- |
|  | square.area = 25  print(square.area)  //打印 25.0  print(square.side)  //打印 5.0 |

area被设置为25时，会自动调用area的set方法，在设置area的同时，把side的值设置为sqrt(25)，为5.0。

### 7.2.2 属性观察器

通过给属性添加willSet和didSet方法可以给类添加属性观察器。属性观察器监控和响应属性值的变化，每次属性被设置值的时候都会调用属性观察器，即使新值和当前值相同的时候也不例外。

willSet是在设置属性值之前被调用，而didSet方法则在设置属性值之后被调用。

当实现willSet时，新的属性值作为常量参数被传递。可以为这个参数起一个别名，如果不命名，则这个参数就默认被命名为newValue，在didSet中原先的值被命名成oldValue。

|  |  |
| --- | --- |
|  | class Tower {      var height : Int = 10 {          willSet {              print("willSet：即将设置新值\(newValue)")          }            didSet{              print("didSet：成功将\(oldValue)设置成\(height)")          }      }  } |

在Tower类中，设置了一个height属性，表示高度，初始化设置成为10。在height的willSet观察器中，打印出height即将被设置的值，而didSet观察器中，打印出之前的oldValue和当前的height。编写一个简单的测试代码。

|  |  |
| --- | --- |
|  | var tower = Tower()  tower.height = 20 |

测试代码中，首先实例化一个tower，此时height是初始化的10，之后，修改height属性为20，程序输出：

|  |  |
| --- | --- |
|  | willSet：即将设置新值20  didSet：成功将10设置成20 |

### 7.2.3 懒加载属性

懒加载属性又称延迟加载属性，指的是当第一次被调用的时候才会计算其初始值的属性。

因为被声明成懒加载的属性，可能在实例构造完成之前无法得到值。而常量属性在构造过程完成之前必须有初始化的值，所以懒加载属性必须声明为变量，并使用lazy关键字标记，以后的开发中，会经常用到懒加载。

## 7.3 类的方法

### 7.3.1 实例方法

实例方法是属于类或结构体实例的方法。可以通过实例方法设置和修改实例的属性值，或是实现实例的某些功能。实例方法和函数类似，使用func声明，可以定义参数和返回，具体逻辑书写在一对大括号中。在实例方法中可以调用所属类型的属性和其他实例方法。

下面的例子，定义一个简单的Counter类，实现计数的功能。

|  |  |
| --- | --- |
|  | class Counter {      var count = 0      func increment() {          count += 1      }      func increment(by amount: Int) {          count += amount      }      func reset() {          count = 0      }  } |

Counter包含一个存储属性和3个实例方法。count用于计数，无参的increment()方法实现计数器加一，而increment(by：Int)方法让计数器一次增加指定数量。reset则可以让计数器清零。

要调用实例方法，必须先创建类的实例，然后才能通过类的实例名.(dot)方法名的方式调用。

|  |  |
| --- | --- |
|  | let counter = Counter()  print("count:\(counter.count)")  counter.increment()  print("count:\(counter.count)")  counter.increment(by: 3)  print("count:\(counter.count)")  counter.reset()  print("count:\(counter.count)") |

程序输出：

|  |  |
| --- | --- |
|  | count:0  count:1  count:4  count:0 |

实例方法都包含一个隐含的参数self，self代指这个实例本身，多数情况下self是可以省略的，比如本节计数器的例子，实际上increment可以写作：

|  |  |
| --- | --- |
|  | func increment() {  self.count += 1  } |

也有某些情况下，self不可以省略，比如第一节包含参数的构造函数的例子：

|  |  |
| --- | --- |
|  | init(brand:String, speed:Int){          self.brand = brand          self.speed = speed  } |

这里的方法参数brand和类的属性brand是重名的，所以此时，必须要用self.来区分两者，消除歧义。如果写作brand = brand，Swift会认为brand指的都是函数的参数brand。

### 7.3.2 类型方法

类的另一种方法叫做类型方法或者静态方法。如果某个操作不依赖具体实例，比如求某个数的绝对值，就可以将其定义为类型方法，实例方法为实例所有，而静态方法为类所有，需要通过类来使用。在为结构体和枚举定义静态方法时，使用static关键字。在为类定义类型方法时，使用class关键字。

下面定义一个Math类，包含一个求绝对值的方法。

|  |  |
| --- | --- |
|  | class Math {      class func abs(\_ num:Int)->Int{          return num >= 0 ? num : -num      }  } |

本例中，利用三元运算符，返回num的绝对值，等价于num大于等于0返回本身，否则返回num的相反数。实现求绝对值这个功能是不依赖于实例的，所以定义成类型方法。事实上，类似Math这种工具类都包含大量类型方法。

通过类名.(dot)方法名来调用类型方法。

|  |  |
| --- | --- |
|  | let A = -20  let absA = Math.abs(A)  print("absA:\(absA)") |

程序输出：

|  |  |
| --- | --- |
|  | absA:20 |

如果一个方法不依赖于实例，建议将它定义为静态方法，可以省去对类进行实例化的步骤。

## 7.4 类的继承

一个类可以继承另一个类的方法，属性和其它特性。当一个类继承其它类时，继承类叫子类，被继承类叫超类（或父类）。在 Swift 中，继承是区分“类”与其它类型的一个基本特征。

在 Swift 中，类可以调用和访问父类的方法，属性和下标，并且可以重写这些方法，属性和下标来优化或修改它们的行为。Swift 会检查重写定义在父类中是否有匹配的定义，以此确保重写行为是正确的。

类的继承是面向对象编程的重要特性之一，具有契合人类认知规律，构建层次关系，减少代码冗余，增加系统可维护性等优点。

首先，创建一个基类，也就是没有继承其他类的类。

|  |  |
| --- | --- |
|  | class Animal {      func say(){          print("I'm an animal")      }        func run() {          print("I'm running")      }  } |

这里定义了一个名为Animal的基类，它拥有两个方法say和run，接着定义一个继承自Animal的子类Cat。

|  |  |
| --- | --- |
|  | class Cat:Animal{      var name: String = "hehe"        override func say(){          print("I'm cat,my name is \(name)")      }  } |

上面的代码中，创建了Cat的类，用：表示这个类继承自Animal，所以Animal是Cat的父类，此时Cat就继承了Animal的say方法和run方法。

在子类中可以添加父类没有的属性和方法，这里添加了一个名为name的属性，并且初始化为 “hehe”。

接着，在子类中重写了父类的say方法。在重写一个父类的方法时，需要在重写的方法名称前面加上override关键字，表示这是对父类方法的重写。

在定义好子类以后，就可以定义子类的实例了。

|  |  |
| --- | --- |
|  | var cat = Cat()  cat.run()  cat.say() |

程序输出：

|  |  |
| --- | --- |
|  | I'm running  I'm cat,my name is hehe |

虽然在Cat中没有定义run方法，但是依旧可以调用run方法，是因为Cat继承了Animal的run方法。由于对于Animal的say方法进行了重写，所以在Cat中调用say方法是输出的Cat中定义的内容。

如果不希望父类的属性、下标、方法被重写，可以在属性或方法之前，添加final关键字修饰。例如 final var、final func、final class func。当子类试图重写final修饰的方法或属性时，编译时都会报错。

## 7.5 本章小结

类是面向对象编程的核心，在面向对象的世界中万物都是对象，类则是对象的抽象。本章介绍了Swift类，包括类的基本概念，类的属性和方法，以及类的创建、析构和使用，重点在概念和语法的掌握，更多关于面向对象编程的理念需要在实践学习领会。

# 第八章 扩展与协议

## 8.1 扩展

扩展可以用来向类，结构体和枚举添加新功能，但是已有的功能不能够在扩展中重写。也就是说，扩展只添加，不修改。

扩展声明使用关键字 extension：

|  |  |
| --- | --- |
|  | extension 已定义类型名 {      // 添加的的新功能  } |

使用扩展可以实现以下操作：添加计算型属性和类型属性；定义实例方法和类型方法；提供新的构造器；定义下标；定义和使用新的嵌套类型；使一个已有类型符合某个协议。接下来将会详细介绍如何利用扩展进行操作。

### 8.1.1属性

扩展可以向已有类型添加计算属性和类型属性,但是不能添加存储属性。

【例 8-1】创建一个Num类并向这个类型扩展属性，代码如下：

|  |  |
| --- | --- |
|  | class Num {      var a:Int=2      var stt = "hello"  }  extension Num{ //属性的扩展  var b:Double{return 2.1} //计算属性  // var b:Double=2.1 //报错，因为是存储属性      class var PrintHelloWorld:String{ //类型属性          return "hello world!!"      }  }  var number=Num()  print(number.a,number.b,Num.PrintHelloWorld,number.stt) |

运行结果为：

|  |  |
| --- | --- |
|  | 2 2.1 hello world!! hello |

### 8.1.2 类型方法和实例方法

在扩展中可以向已有类型添加新的类型方法和实例方法。

【例 8-2】使用扩展向类Student添加新的类型方法和实例方法：

|  |  |
| --- | --- |
|  | class  Student {      var name:String=""      var id:String=""      var age:Int=0  }  //扩展方法  extension Student{      func printInformation(name:String,id:String,age:Int) {          print("\(name)'s id is \(id).")          print("\(name) is \(age).")      }      class var isStudent:String{          return "He/She is a student."      }  }  print(Student.isStudent)  var st=Student()  st.printInformation(name: "Jane", id: "10015", age: 16) |

运行结果为：

|  |  |
| --- | --- |
|  | He/She is a student.  Jane's id is 10015.  Jane is 16. |

对值类型进行扩展时，可以用关键字mutating 修饰某方法，使该方法可以在其内部直接改变实例本身的值。

【例 8-3】使用扩展向值类型添加新的方法来改变实例的值，代码如下：

|  |  |
| --- | --- |
|  | extension Int{      mutating func add(){          self=self+10      }  }  var h=5  h.add()  print(h) |

运行结果为：

|  |  |
| --- | --- |
|  | 15 |

### 8.1.3 构造器

扩展可以向类型添加新的构造器。

【例 8-4】使用扩展向结构体Newstruct添加新的构造器：

|  |  |
| --- | --- |
|  | struct Newstruct{      var num:Int      var str:String="nothing"      init(num0:Int) { //原构造器          num=num0      }  }  extension Newstruct { //扩展构造器      init(num1:Int,str1:String){          num=num1          str=str1      }  }  let n=Newstruct(num0: 55)  let m=Newstruct(num1: 66, str1: "Hello")  print(n.num,n.str)  print(m.num,m.str) |

运行结果为：

|  |  |
| --- | --- |
|  | 55 nothing  66 Hello |

对于类来说，扩展只能添加新的便利构造器 init()，但是不能向类中添加新的指定构造器或析构函数 deinit() 。向类添加构造器时，要在init之前添加关键字convenience。

【例 8-5】使用扩展向Ms类添加新的构造器：

|  |  |
| --- | --- |
|  | class Ms {      var f:Int      var str:String      init() {          f=10          str="jarry"      }  }  extension Ms{      convenience init(f:Int,nr:String){          self.init()          self.str=nr          self.f=f      }  }  var obj=Ms()  print(obj.str)  var obj1=Ms(f: 66, nr: "micheal")  print(obj1.str) |

运行结果为：

|  |  |
| --- | --- |
|  | jarry  micheal |

### 8.1.4下标脚本

使用扩展可以向已有类型添加新的下标脚本。创建以subscript为函数名的函数，可以使用下标脚本来运行该函数。

【例8-6】使用扩展向Ss类型添加新的下标脚本：

|  |  |
| --- | --- |
|  | struct Ss {      var nee:Int  }  extension Ss{      subscript(x:Int,y:Int)->Bool{          return x\*y/nee == 0 ? true:false      }  }  let ss=Ss(nee: 3)  print(ss[8,22]) |

运行结果为：

|  |  |
| --- | --- |
|  | false |

## 8.2 协议

协议规定了一系列属性和方法，但又不在协议中具体实现它。这和对结构体中的属性只声明不定义很相像。对于能够按协议要求对协议规定的属性和方法进行实现的类型，被称为遵守该协议。这里的类型包括类，结构体和枚举。

协议的定义形式如下：

|  |  |
| --- | --- |
|  | protocol 协议名称 {      // 协议内容  } |

对于遵守该协议的类型，以类为例，形式如下：

|  |  |
| --- | --- |
|  | class 类名: 协议一, 协议二，...{      // 类内容，包括对协议内容的实现  } |

类型名后面跟冒号 ‘：’ 后跟遵守的协议。类型可以遵守多个协议，各协议之间用逗号 ‘，’ 隔开。对于类来说，如果它要继承某个类，那么要把继承的父类名放在所有协议之前，形式如下:

|  |  |
| --- | --- |
|  | class 类名: 父类, 协议一, 协议二，...{  // 类内容，包括对协议内容的实现  } |

下面将会介绍对于协议中不同种类成员的具体声明和其实现。

### 8.2.1属性

协议中可以声明属性。属性类型可以是实例属性或类型属性，可以是存储属性或计算属性。协议中通常用 var 来声明变量属性，而且必须要说明该属性是只读属性还是可读写属性。

对协议中属性的声明如下：

|  |  |
| --- | --- |
|  | protocol 协议名 {      var 属性名：数据类型 { get set } //可读写属性  var 属性名：数据类型 { get } //只读属性  static var 属性名：数据类型 { get } //类型属性  } |

【例8-7】声明协议Stu并定义类Studentclass来遵守该协议：

|  |  |
| --- | --- |
|  | protocol Stu{      static var age:Int { get }      var name:String { get }  }  class Studentclass: Stu{      var name:String {          return "Jerry"      }      class var age:Int {          return 10      }  }  var ncc = Studentclass()  print(ncc.name)  print(Studentclass.age) |

运行结果为：

|  |  |
| --- | --- |
|  | Jerry  10 |

### 8.2.2方法

协议中可以声明方法。在协议中声明的方法不声明方法内容，只声明该方法的参数及其类型和返回值类型。协议中方法的声明形式如下：

|  |
| --- |
| protocol 协议名{  //类型方法的声明  static func 方法名(参数名1：数据类型，参数名2：数据类型...)->返回值数据类型  //实例方法的声明  func 方法名(参数名1：数据类型，参数名2：数据类型...)->返回值数据类型  } |

【例8-8】声明协议PrintHello并定义类SayHello来遵守该协议：

|  |  |
| --- | --- |
|  | protocol PrintHello {      func printHello()  }  class SayHello:PrintHello{      var name:String="Jane"      func printHello(){          print("Hello \(name)!!")      }  }  let ls=SayHello()  ls.printHello() |

运行结果为：

|  |  |
| --- | --- |
|  | Hello Jane!! |

当实例为值类型时，如果想利用协议中的方法改变实例中成员的值，必须要用到关键字‘mutating’，语法形式如下：

|  |
| --- |
| protocol 协议名{  mutating func 方法名(参数名1：数据类型，参数名2：数据类型...)  } |

【例8-9】声明协议Change并定义枚举number来遵守该协议：

|  |  |
| --- | --- |
|  | protocol Change {      mutating func changeNum()  }  enum numberChoice:Double,Change{      case a=1.1      case b=2.2      case c=3.3      mutating func changeNum() { //改变枚举中属性的值          switch self{          case .a:              self = .c          case .b:              break          case .c:              self = .a          }      }  }  var nn=numberChoice.a  nn.changeNum()  print(nn.rawValue) |

运行结果为：

|  |  |
| --- | --- |
|  | 3.3 |

### 8.2.3可选协议

所谓可选协议，就是遵守该协议的类型可以选择实现或不实现协议成员，可以实现全部成员或部分成员。这让协议的灵活性大大增强。可选协议名前要加上关键字 @objc，协议中使用关键字 @objc optional 作为前缀来定义可选成员。其一般形式如下：

|  |
| --- |
| @objc protocol 协议名 {      @objc optional func 方法名(参数名1：数据类型，参数名2：数据类型...)-> 数据类型      @objc optional var 属性名: 数据类型 { get }  } |

【例8-10】声明可选协议DataChange并定义类Data来遵守该协议：

|  |  |
| --- | --- |
|  | @objc protocol DataMul{      @objc optional var value:Int{get}      @objc optional func dataMultiplied(data:Double)->Double  }  //以下类没有实现协议中所有成员  class Data:DataMul{      var n=1.0      var value1 = 3      @objc func dataMultiplied(data: Double) -> Double {          for \_ in 1...value1{               n=n\*data          }      return n      }  }  let data\_x=Data()  print(data\_x.dataMultiplied(data: 2.2)) |

运行结果为：

|  |  |
| --- | --- |
|  | 10.648 |

### 8.2.4协议类型

协议可以作为类型被使用。如某协议作为某函数参数的类型，那么这个参数可以是任意遵守了此协议的数据类型。

【例8-11】声明协议Naaa,使用该协议类型作为方法cak中参数的参数类型，并定义结构体Sd来遵守该协议：

|  |  |
| --- | --- |
|  | protocol Naaa {                            //协议      var h:Double{get}      var style:String{get}      var r:Double{get}  }  func cake(pa:Naaa){                        //参数为协议类型      let h=pa.h      let area=pa.r\*pa.r      let name=pa.style      print(h,area,name)  }  struct CakeStyle:Naaa{                    //遵守协议的结构体      var h:Double      var style: String      var r: Double  }  //结构体实例化  let s = CakeStyle(h: 0.2, style: "pudding", r: 0.6)  cake(pa:s)                                //结构体类型作为参数 |

运行结果为：

|  |  |
| --- | --- |
|  | 0.2 0.36 pudding |

假如某数组的类型为协议类型，如：

|  |  |
| --- | --- |
|  | var array:Array<OneProtocol> |

这表示数组里的所有元素都要遵守协议。

### 8.2.5协议的继承

协议也可以像类一样被其他协议继承。对于继承了某协议的协议，相当于拥有父类协议声明的所有成员。而遵守了该协议的数据类型，要实现该协议本身和其父协议声明的所有成员。

【例8-12】声明协议FirstProtocol，声明协议SecondProtocol继承FirstProtocol并定义结构体Mystruct来遵守SecondProtocol协议：

|  |  |
| --- | --- |
|  | protocol FirstProtocol {      var a:Int{get}      var b:Double{get}  }  protocol SecondProtocol:FirstProtocol { //继承协议      var s:String{get}  }  struct Mystruct:SecondProtocol { //结构体遵守协议      var a=5      var b=1.26      var s="hello Swift"  } |

### 8.2.6协议的其他功能

有的类型要同时遵循好几个协议，Swift协议可以使用关键字 ‘&’ 进行合成，这样可以在某些情况下简化代码的书写。一般形式如下：

|  |
| --- |
| protocol case {}  protocol case1 {}  protocol case2 {}  class Saa:case&case1&case2{} |

使用 ‘&’ 把各协议隔开就可实现协议的合成。

如果想让某协议只能被类遵守，可以用关键字class来说明，如：

|  |
| --- |
| protocol Myprotocol:class {} |

有的实例遵守多个协议或者遵守的协议继承其他协议，当想要检查某实例是否遵循某协议或想把该实例转化为某协议类型时，就要用到关键字 ’is’、’as!’、’as?’ 。这几个关键字的功能分别是：

（1）is操作符用来检查实例是否遵循了某个协议。

（2）as?是当实例遵循某协议时，返回该协议类型;否则返回nil。

（3）as！用以强制向下转型。使用时一定要确定要转型的协议是所需要的。如果强转失败，会引起运行时错误。

## 8.3 本章小结

本章主要介绍了扩展与协议的相关知识，如基本形式、调用方法、自身特性和自身其他功能。本章的重点在于掌握扩展与协议的基本用法，了解扩展与协议的全部用法，熟练应用扩展与协议的声明定义和调用方法。

# 第九章 异常处理与类型转换

## 9.1异常处理

开发者通常希望程序在运行过程中，即使出现了未知异常，还能够不中断的运行下去。Swift语言提供了异常处理机制来解决这个问题。

程序中的问题分为两种，一种是错误，另外一种是异常。在编译时就能够检测出来是否有错误，如果有错不能通过编译。而异常是能够通过编译，只是在运行的过程中发生了错误，这种错误称为异常。

### 9.1.1普通的异常处理

对于没有异常捕获的程序中，如何来修改和处理程序中的错误? 实例如下:

|  |  |
| --- | --- |
|  | var x:Int = 6  var y:Int = 0  var z = x/y  print(z) |

程序必然会抛出异常，虽然程序没有语法错误，能够通过编译，但是运行的过程中，发现y为零，导致程序抛出异常，由于没有对异常处理的代码，导致程序终止运行。如果想让程序继续运行下去，可以这样修改程序，实例如下：

|  |  |
| --- | --- |
|  | var x:Int = 6  var y:Int = 0  var z:Int?  if y==0{  print("除数不能为零")  }else{  z = x/y  }  print(z) |

以上程序输出:

|  |  |
| --- | --- |
|  | 除数不能为零  nil |

通过以上的两个例子，可以看出对于程序运行中出现的异常，程序只有终止运行，让开发者自己手动的去修改程序。下面来介绍一种方法，在程序运行中出现异常后，程序还能继续运行。

### 9.1.2异常的捕获处理

如果程序中出现异常，导致程序终止运行。这种情况非常的不利于开发者编写程序，Swift中为开发者提供了异常的处理方法，一般可以分为4步:

第一步:定义异常类型

在swift中需要定义自己的异常类型，但是必须要遵从语言中提供的Error协议，通过enum关键字枚举自定义出自己的异常类型。实例如下：

|  |
| --- |
| enum TestError:Error{  case HighError  case MidError  case lowError  } |

通过以上的方法，可以高度自定义自己的异常类型。

第二步：使用throws来声明异常，用throw来抛出异常

在函数中添加throws关键字，表明这个函数是能够抛出异常的函数。而要抛出什么样的异常，是要根据函数中的具体情况，并通过throw来抛出异常。实例如下：

|  |  |
| --- | --- |
|  | func TestFunc(param:bool)throws->Void{  if param{  print("ok")  }else{  throw TestError.lowError  }  } |

第三步：异常的捕获与处理

在swift语言中，是使用do-catch语句来处理所抛出的异常。下面是do-catch语句的一般形式：

do{

try 可能抛出异常的函数

语句  
}catch 异常类型{

语句

}

do-catch异常结构处理方法，需要把有可能抛出异常的代码语句放入到do结构块当中。如果此语句出现了异常，会判断所属异常类型，执行相匹配异常代码。如果此语句没有抛出异常，则程序将继续执行都后面的语句。示例如下：

|  |  |
| --- | --- |
|  | do{  try TestFunc(param:false)  }catch TestError.lowError{  print("this is lowError")  } catch TestError.MidError{  print("this is MidError")  }catch TestError.HighError{  print("this is HighError")  } |

do语句块中存放了一个能抛出异常的函数体，当函数抛出异常时，会与catch的异常名进行比对，如果对比成功，即捕获到异常，程序就会运行相对应异常下的语句。

在swift中异常处理中，try关键字是告诉编译器当try紧跟的函数抛出异常时，将其送往对应的catch中处理。还有两种不同的方法，第一种是把try改成try!,这种用法是告诉系统一定没有异常，也就是说不用do-catch来处理，开发中不建议用，一旦有异常，程序崩溃。而另一种是把try改成try?，这种用法是告诉系统可能有异常，也可能没有异常。如果没有异常，系统会自动将结果包装成一个可选类型。如果有异常，系统会返回nil。

### 9.1.3异常处理统一出口

通常情况下，异常处理需要释放函数中所使用的一些资源和关闭文件操作。但是由于异常处理存在一种跳转的情况，所以需要程序的异常处理的统一出口。

Swift中是利用defer代码块来实现最后的延时执行，即此段代码所放何处，程序最后都会执行这段代码。实例如下：

|  |  |
| --- | --- |
|  | func testFunc2(){  defer{  print("程序结束时调用......")  }  print("程序开始......")  print("程序运行中......")  }  testFunc2() |

以上程序输出：

|  |  |
| --- | --- |
|  | 程序开始......  程序运行中......  程序结束时调用...... |

## 9.2类型转换

对于Swift中的类型转换，一般分为两步，第一步是对其进行类型检查，判断某个实例是否属于具体类型。第二步就是对其类型的实例进行类型的转换。

9.2.1类型检查

Swift语言中要判断某个实例是否属于某个具体类型，可以使用is关键字，组成一个判断语句来返回一个布尔值，可以根据布尔值的真假来判断是否同属一个类型。实例如下:

|  |  |
| --- | --- |
|  | var str = "HDU"  if str is String{  print("str 的类型是String")  } |

类型检查主要是用在有继承关系的类上，即子类实例进行父类类型检查时为真，父类实例进行子类类型的检查时为假。

### 9.2.2类型转换

Swift语言中的类型转换是使用as关键字。同类型检查相似，类型转换也是主要用在有继承关系的类上，即一个父类类型的集合可以接受子类类型的实例，在父类类型也可以直接转换成子类类型。实例如下：

|  |  |
| --- | --- |
|  | class Base{  }  class MyClass:Base{  var name:String?  }  class SubOne:MyClass{  var nameOne:String?  }  class SubTwo:MyClass{  var nameTwo:String?  }  var obj1 = MyClass()  obj1.name = "HD"  var obj2 = SubOne()  obj2.nameOne = "nameone"  var obj3 = SubTwo()  obj3.nameTwo = "nametwo"  var array:[MyClass] = [obj1,obj2,obj3]  for i in 0..<array.count{  var obj = array[i]  if obj is SubOne{  print((obj as! SubOne).nameOne!)  continue  }  if obj is SubTwo{  print((obj as! SubTwo).nameTwo!)  continue  }  if obj is MyClass{  print(obj.name!)  }  } |

以上程序输出:

|  |  |
| --- | --- |
|  | HD  nameone  nametwo |

注：在使用类型转换时，需要使用as?和as!方式。as?是一种比较安全的转换方式，其会将类型转换后的结果映射为Optional值，如果类型转换成功，则值为原实例，如果类型转换失败，则会放回Optional值nil。而as!是一种强制转换方式，其默认此次转换一定成功，如果转换失败，则会产生运行时错误，程序会崩溃。开发者在使用as!进行类型转换时，必须保证实例的真实类型和要转化的类型一致。

## 9.3本章小节

本章中介绍了异常以及异常的处理，在Swift中所需要抛出的异常类型，需要开发者自己来定义相对应的异常，这样在编写过程中可以高度自定义所要抛出的异常。类型转换主要是指子类与父类之间的转换，在转换过程中必须要对类之间的继承关系要比较清楚。本章的内容是难点也是重点，在实际的开发过程中应用的非常频繁，希望读者认真掌握。

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。

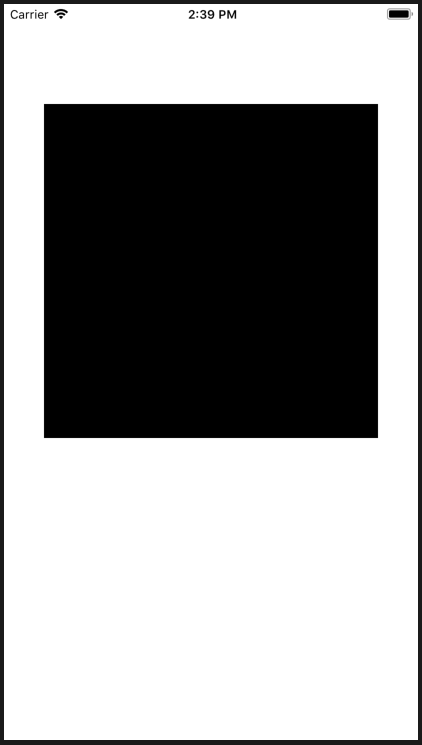


图10-2 使用UIView创建正方形视图

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。

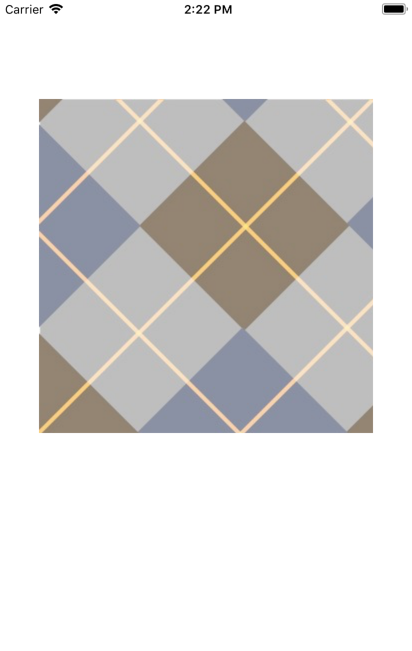


图10-3 设置格子图片为视图背景

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。

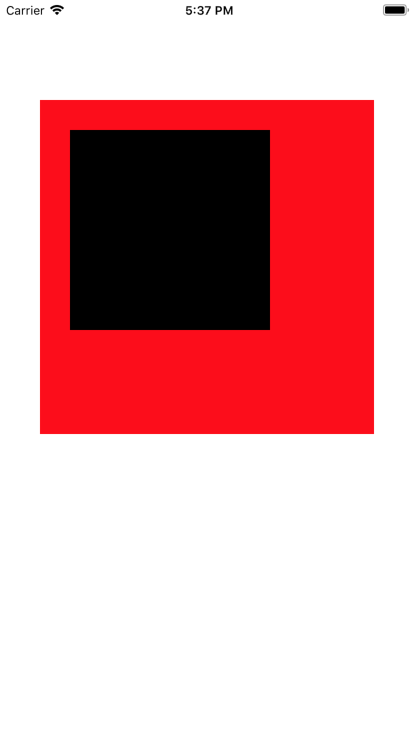
 

图10-5 在一个视图上添加另一个视图 图10-6改变子视图大小

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。

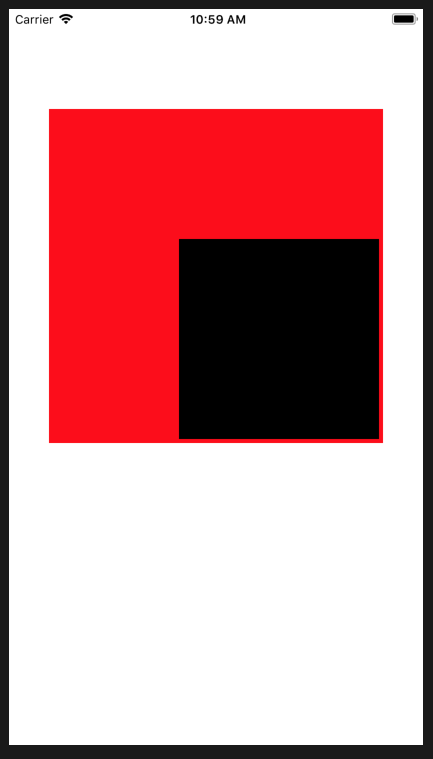
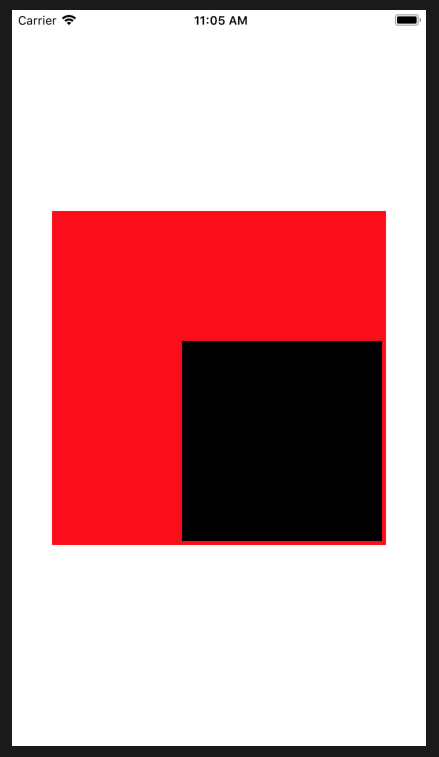
 

图10-7 图10-8

### 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。

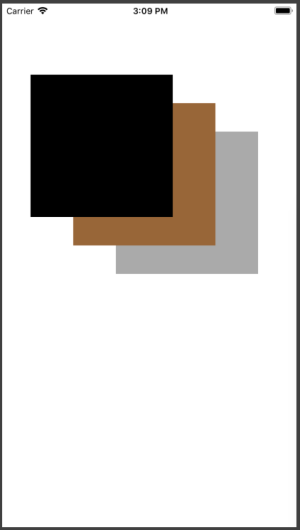
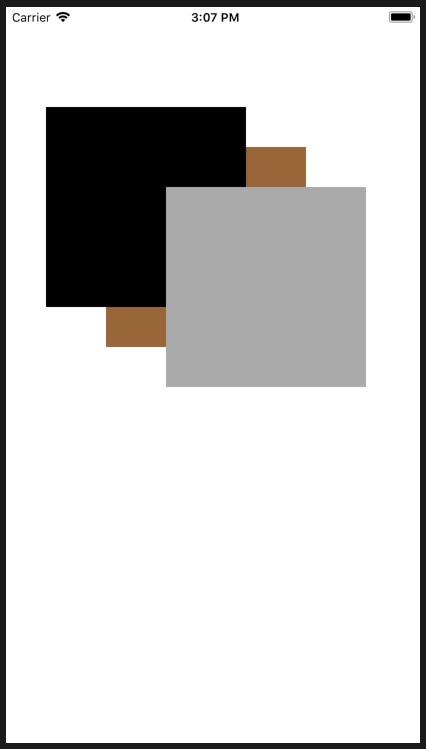
 

图10-9 图10-10

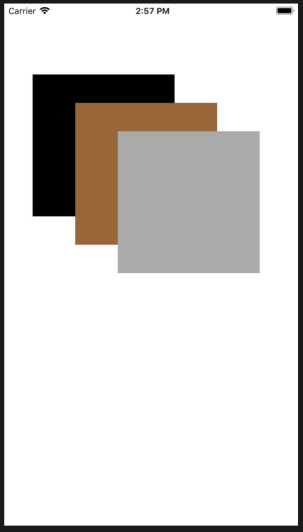
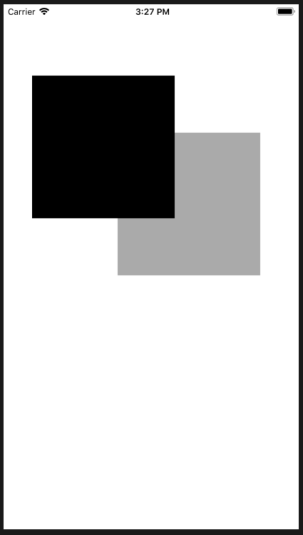
 

图10-11 图10-12

### 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

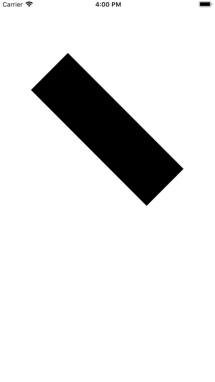
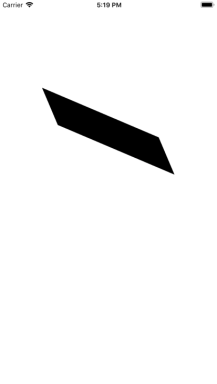
  

图10-15 图10-16 图10-17

## 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。

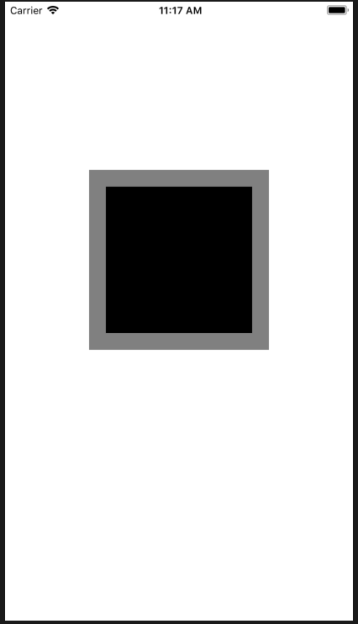
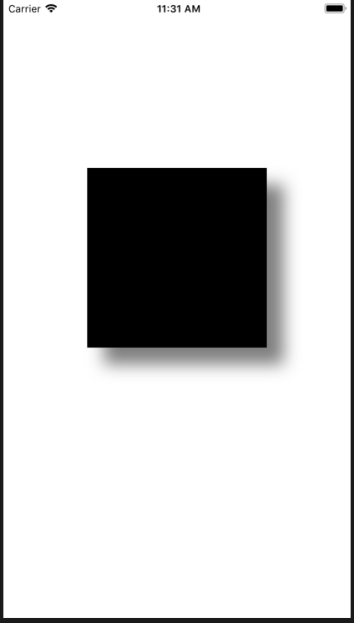
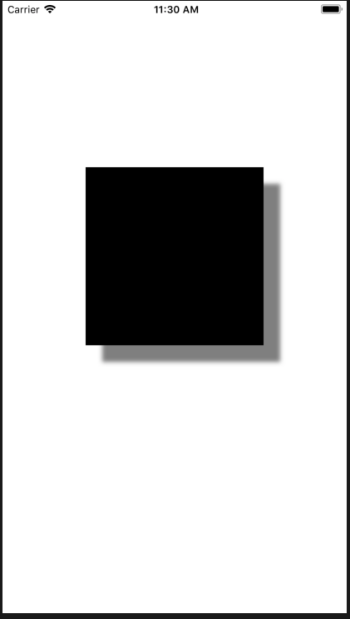
  

图10-18 图10-19 图10-20

### 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。

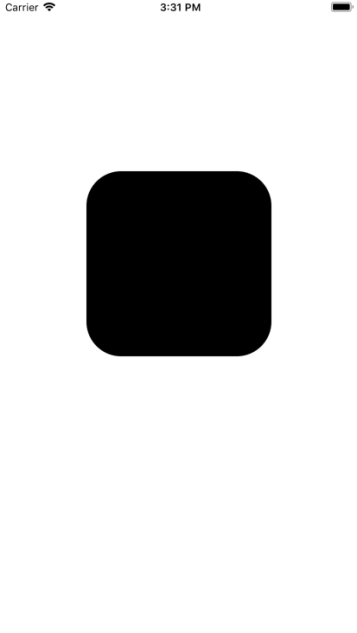
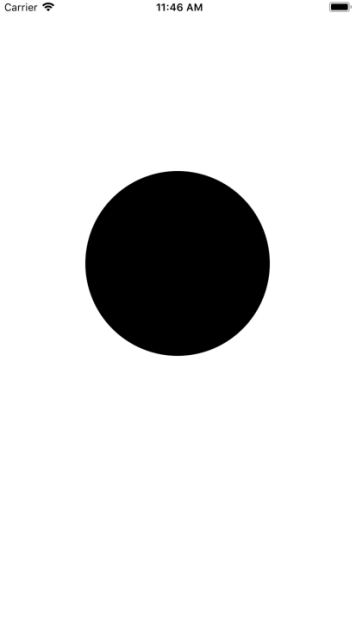
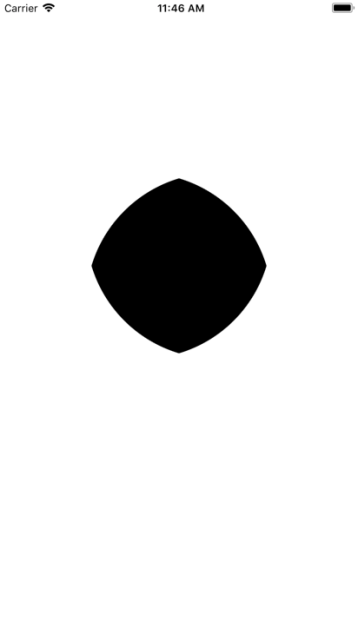
  

图10-21 图10-22 图10-23

### 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。

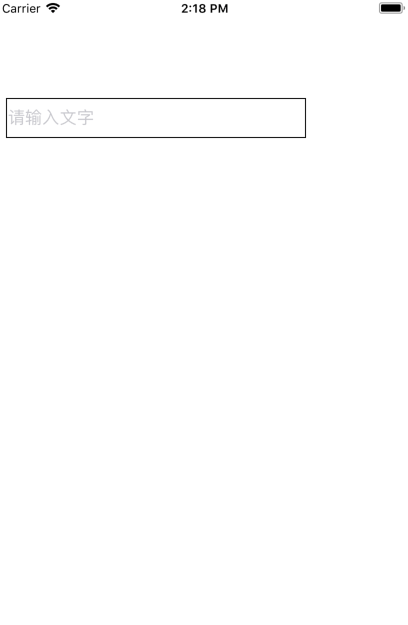


图11-6

### 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。

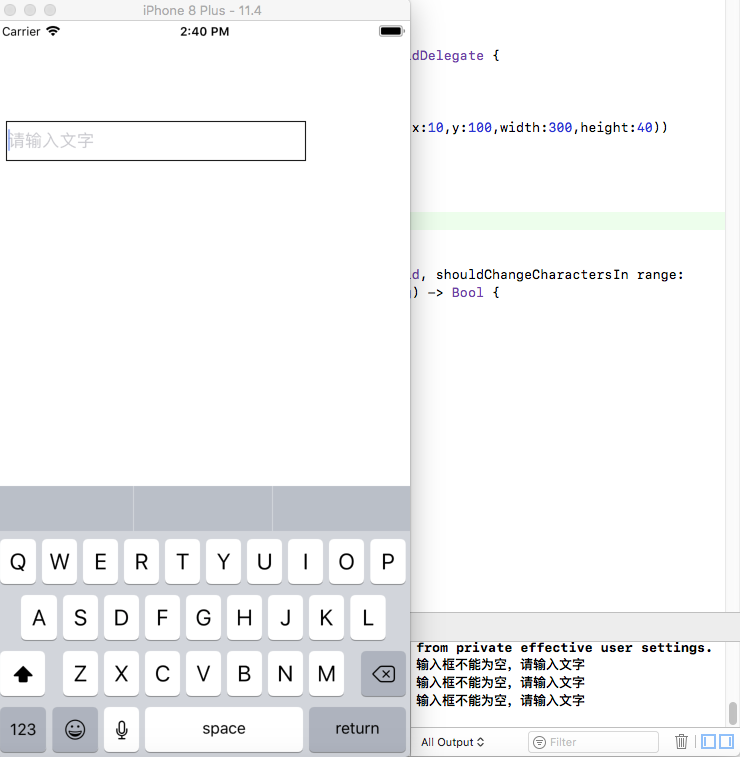


图11-7

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。

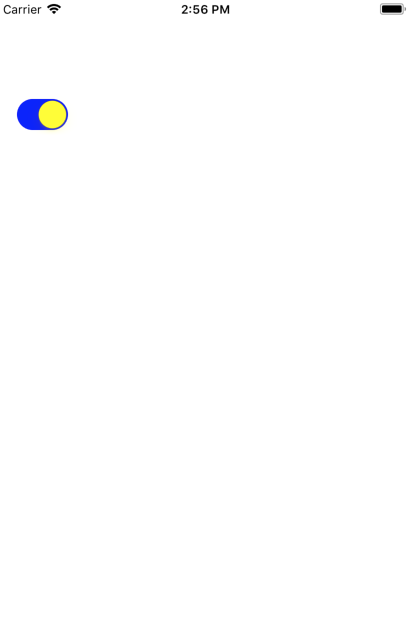


图11-8

### 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。

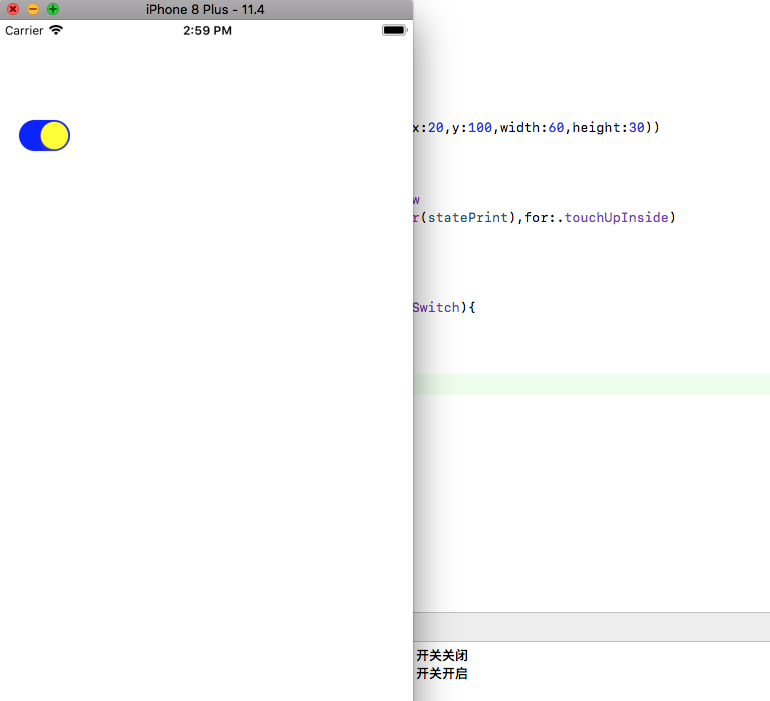


图11-9

## 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。

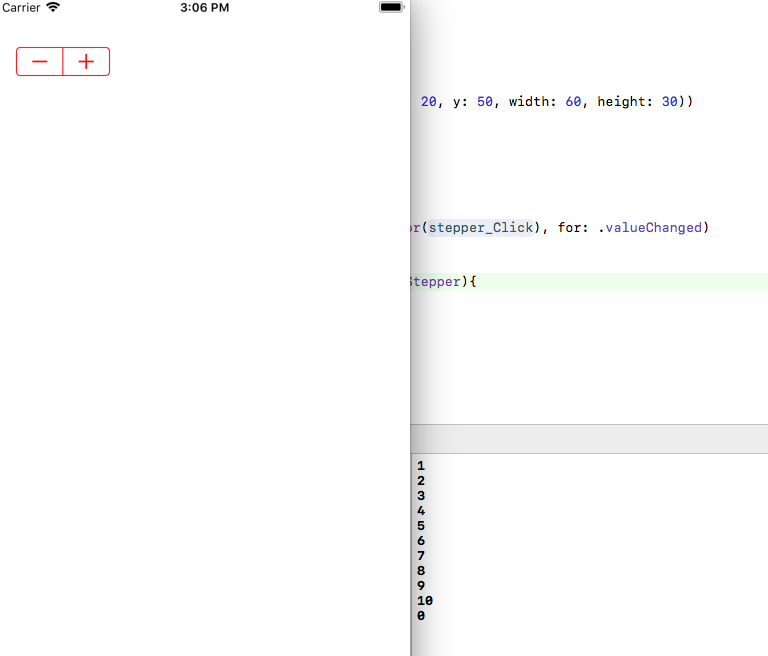


图11-10

## 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)  }  } |

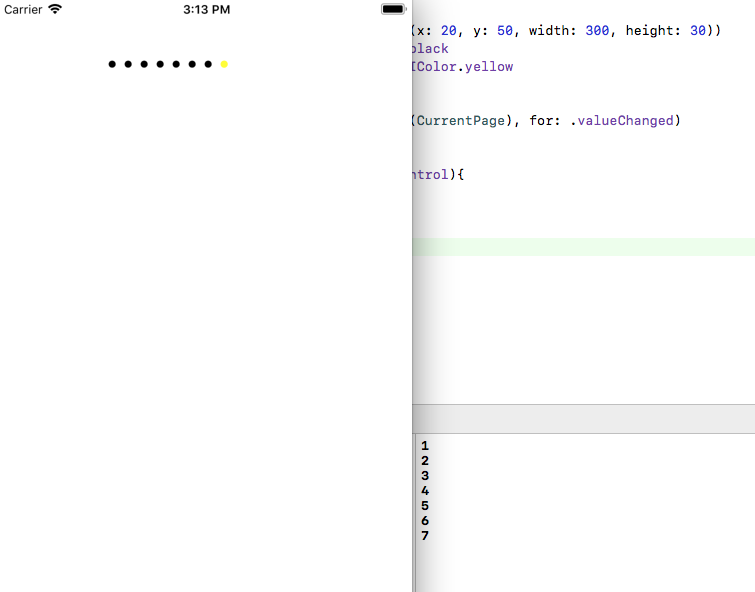
由于UIPageControl控件默认背景圆点是白色的，需要通过设置pageIndicatorTintColor这个属性将所有圆点设置成黑色，这样便于效果演示。currentPageIndicatorTintColor属性是用来设置高亮页点的颜色，currentPage是设置最初的页面停留在哪里。numberOfPages属性用来设置总共有多少页。点击UIPageControl控件的左半边时，页码会向左移动，相反点击右半边，页码会向右移动。在点击的过程中会触发交互方法，会打印出当前的页码。运行效果如图11-11所示。

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。

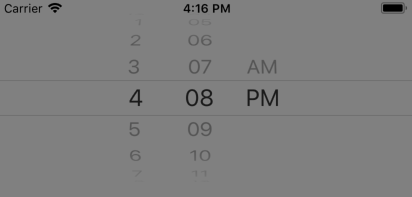
 

图11-14 date模式 图11-15 time模式

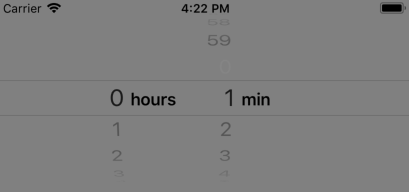
 

图11-16 dateAndTime模式 图11-17 countDownTimer模式

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。

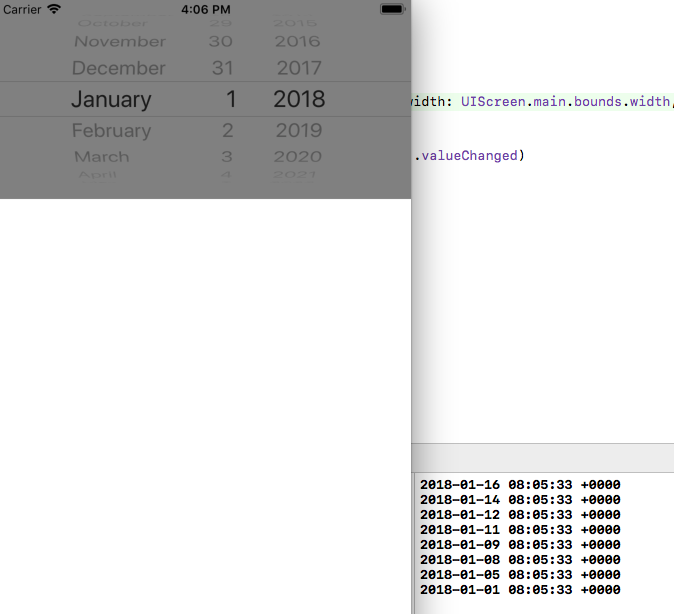


图11-18

## 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。

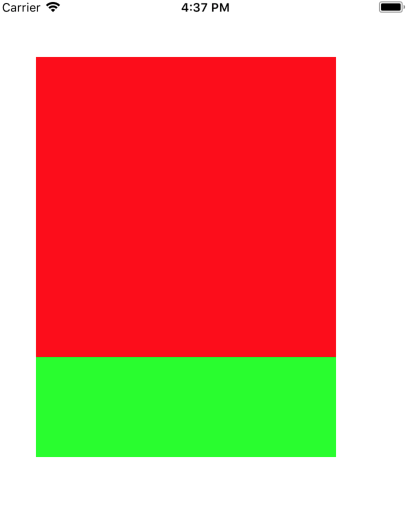


图11-19

## 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。

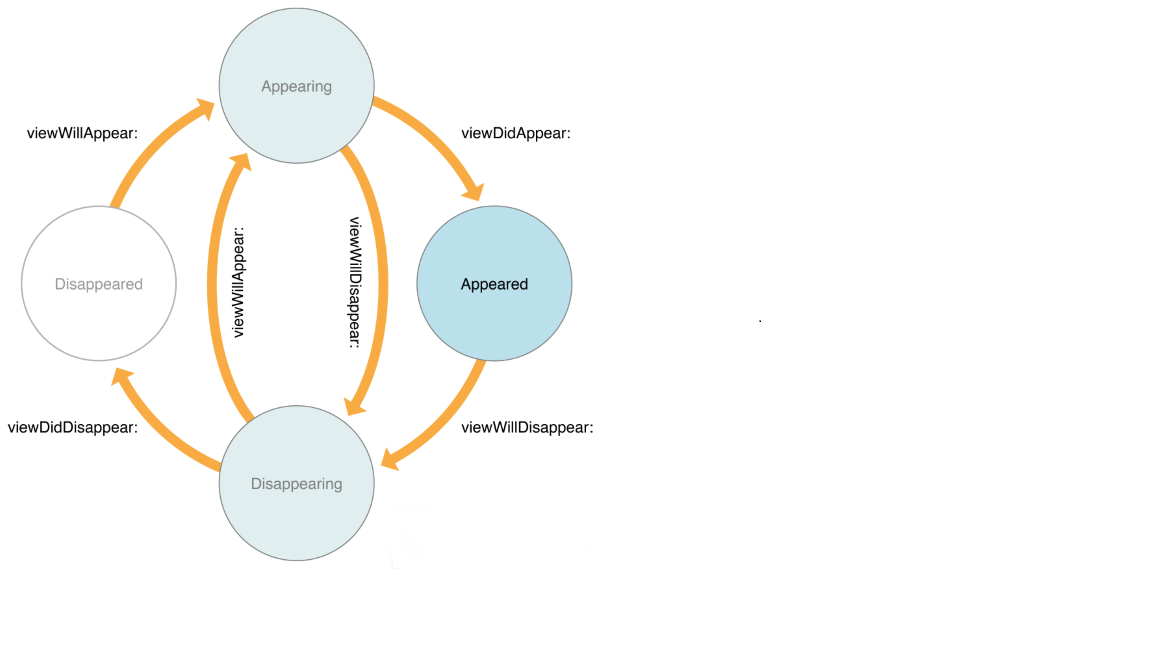


图12-1

### 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】。

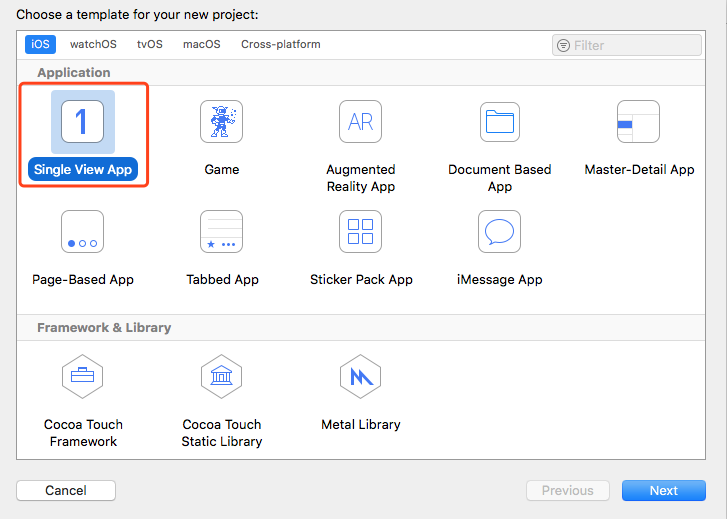


图 12-2

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。

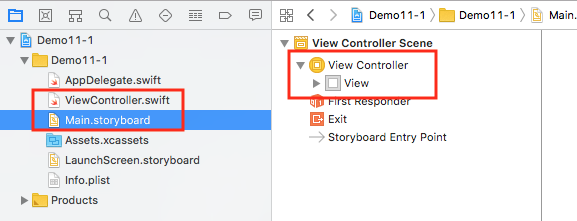


图12-3

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。

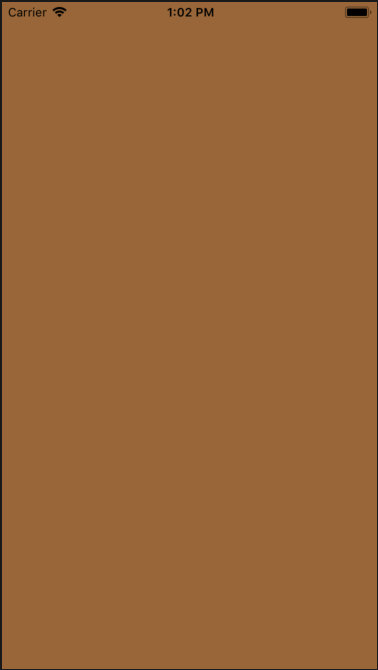


图12-6

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。

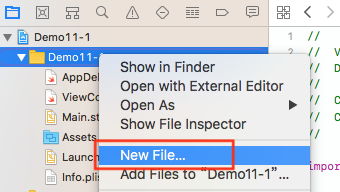


图12-7

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

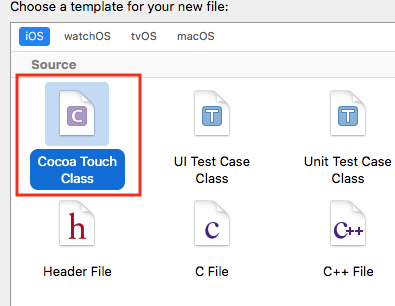


图 12-8

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：

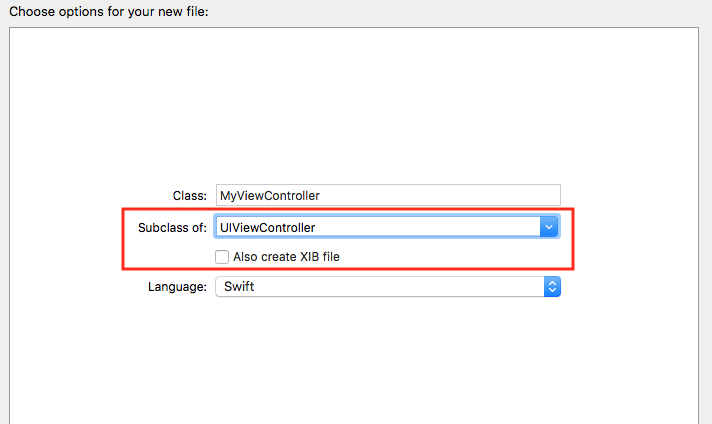


图12-9

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。

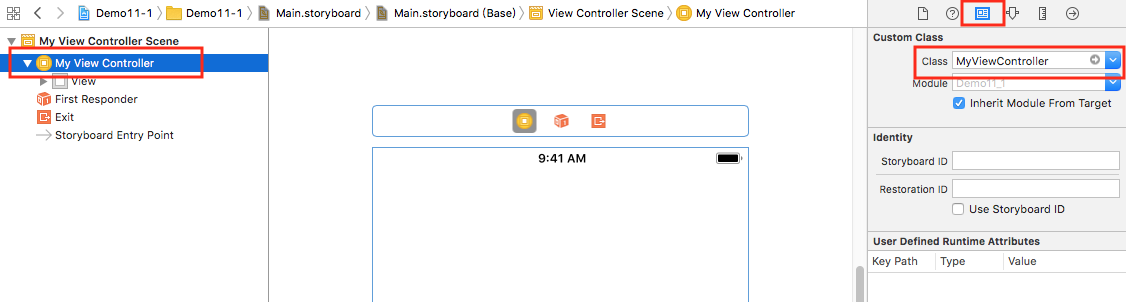


图12-10

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。

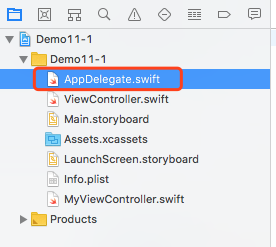


图12-11

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。

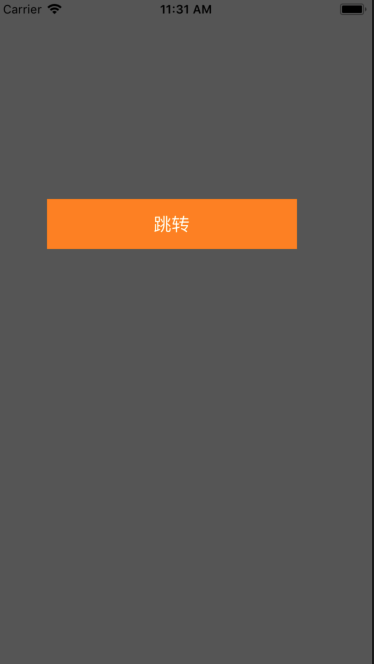
 

图12-13 图12-14

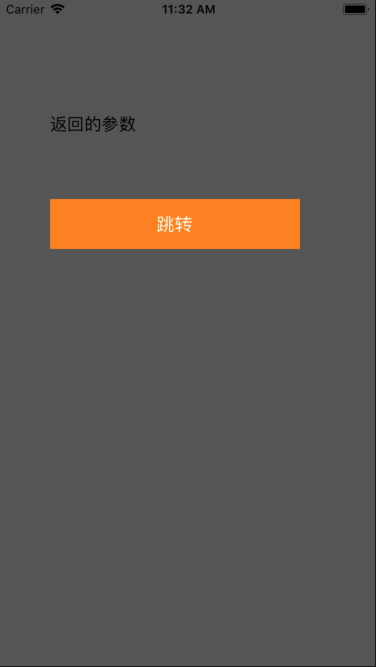


图12-15

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。

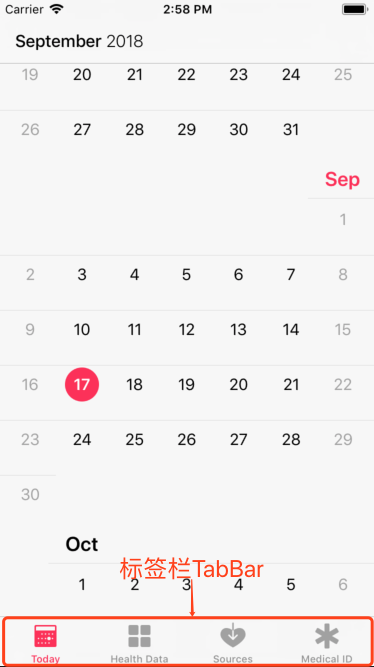


图12-16

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。

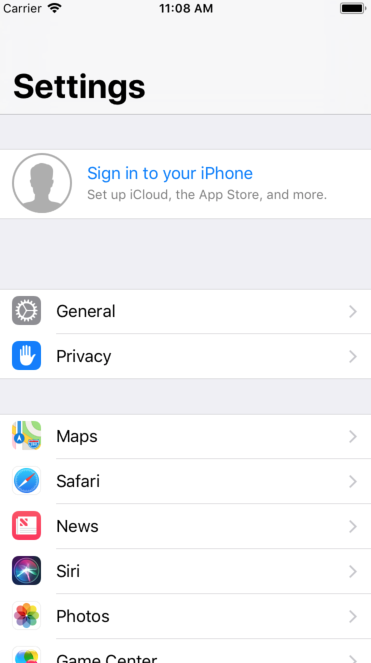
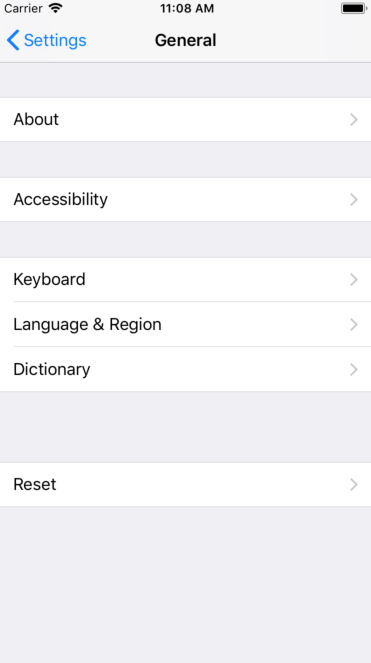
 

图 12-22 图12-23

### 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。

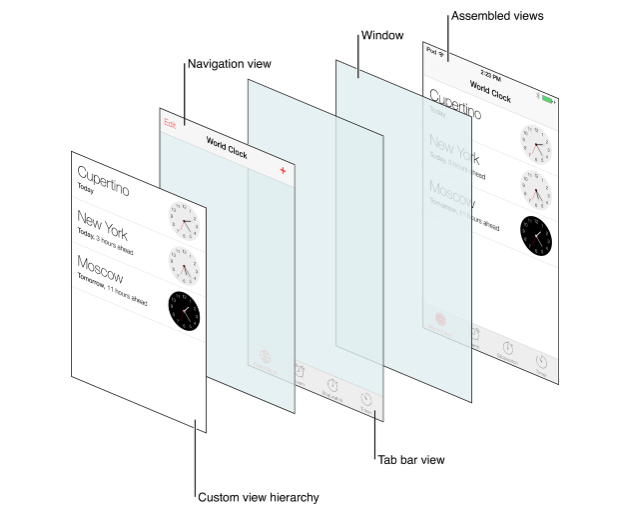


图12-24

### 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。

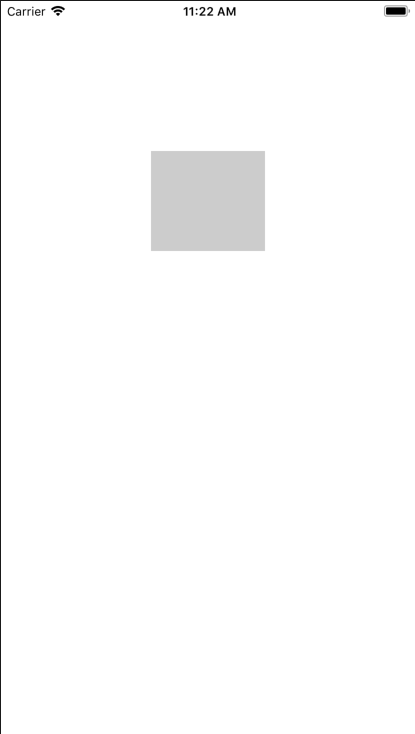
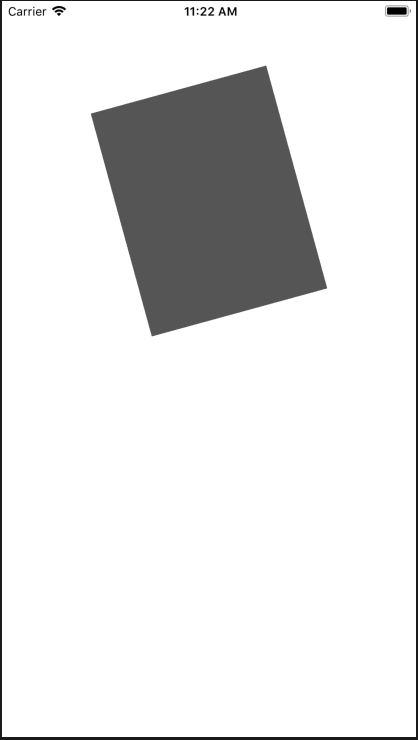
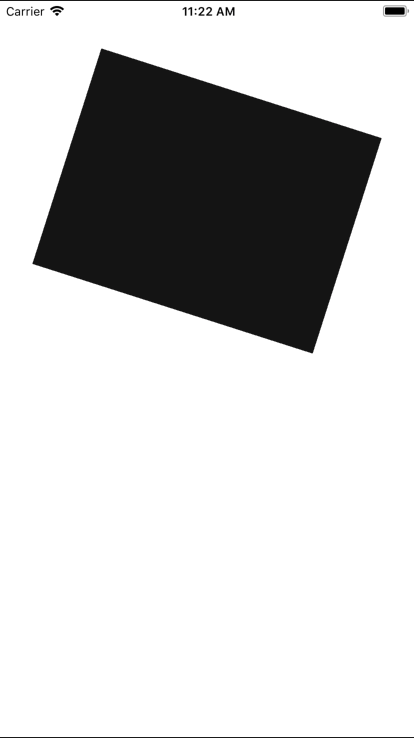
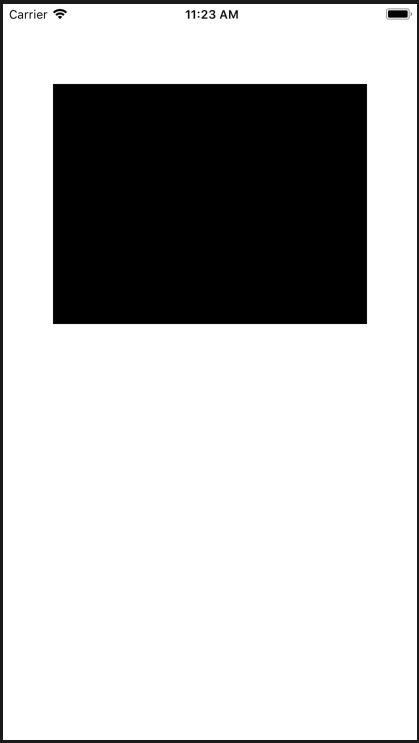


图13-1 动画效果

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。

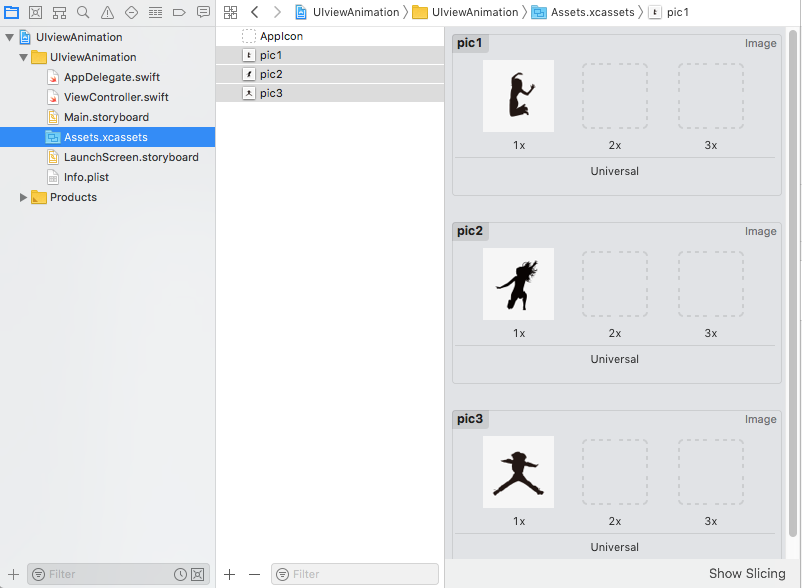


图13-2向Assets.xcassets导入图片序列

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。

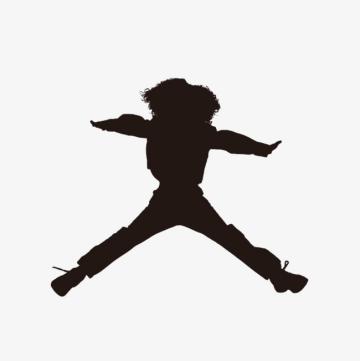
  

图13-3 帧动画中的图片

## 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。

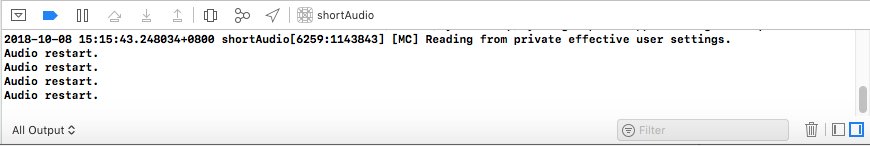


图13-4日志输出

### 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。

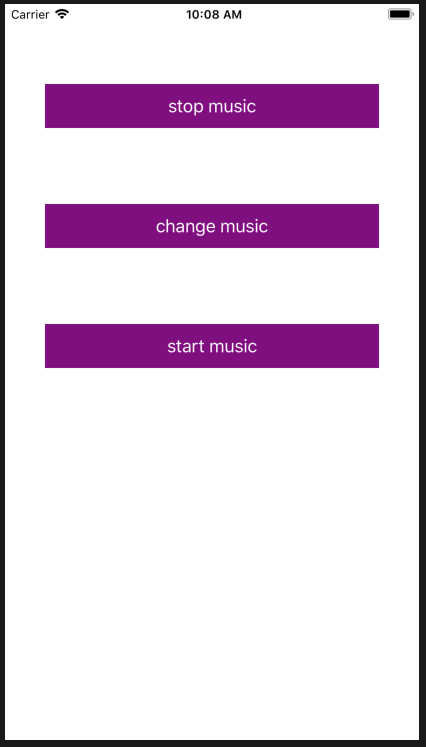


图13-5 音频播放模拟器界面

## 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。

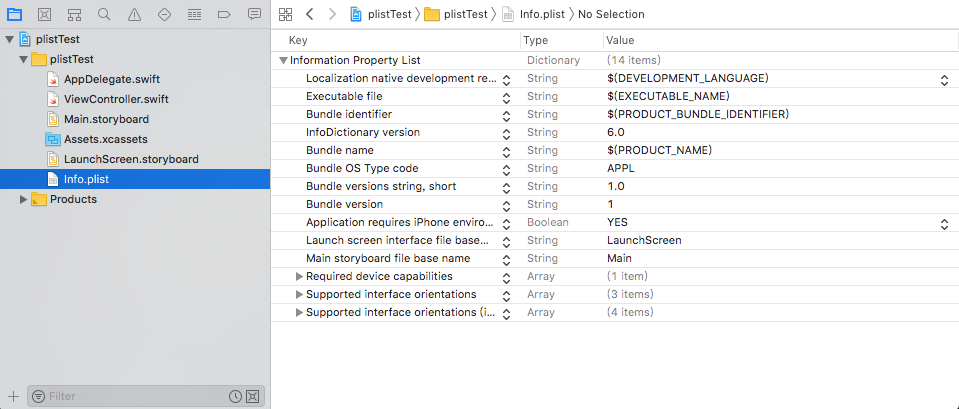


图14-1 plist文件

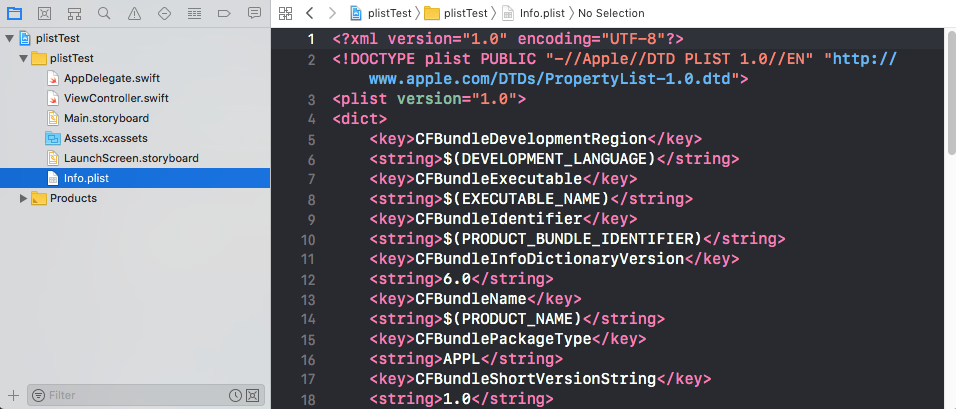


图14-2 plist文件xml源代码

### 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。

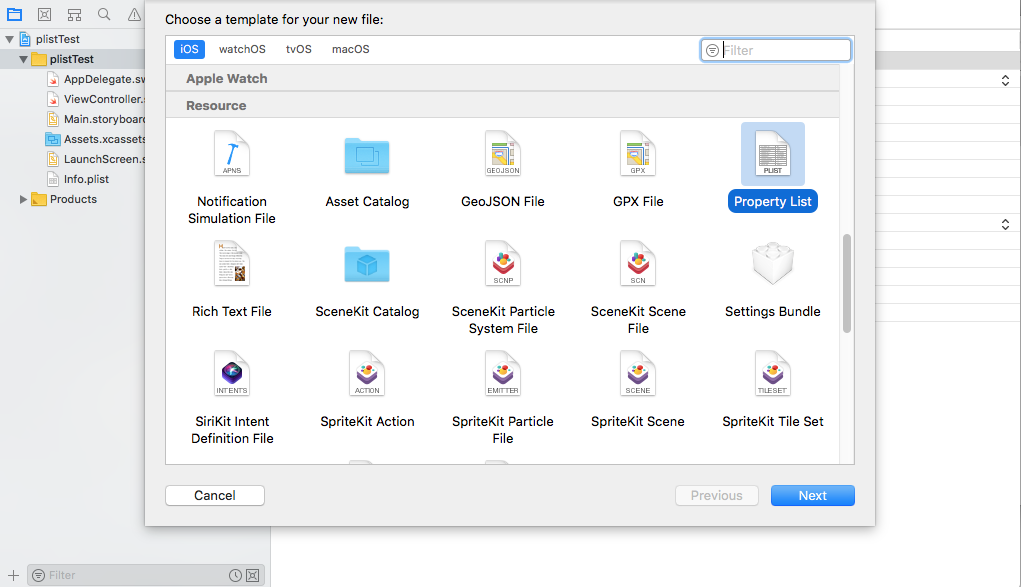


图14-3

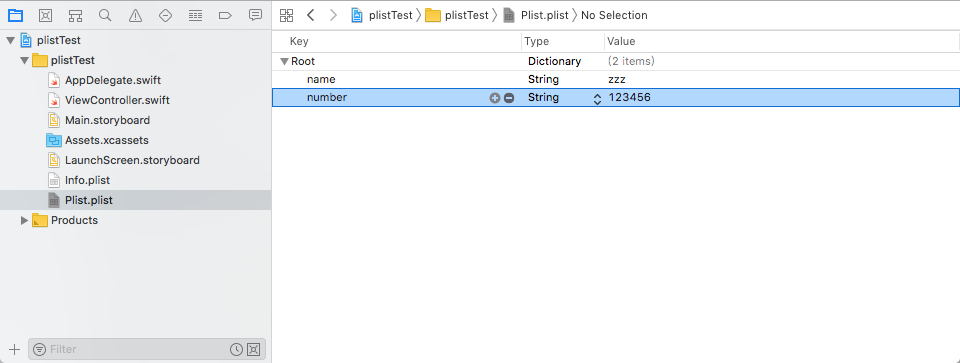


图14-4

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。

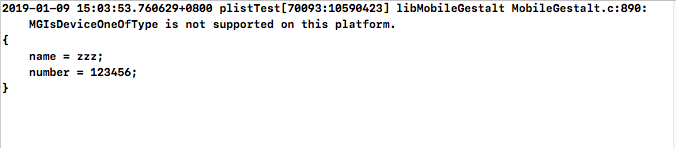


图14-5 日志区输出结果

### 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

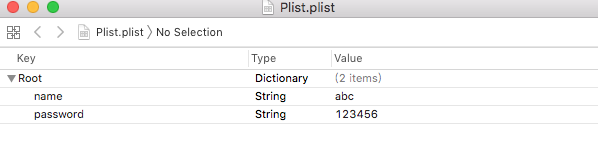


图14-7 沙箱内的plist文件

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。

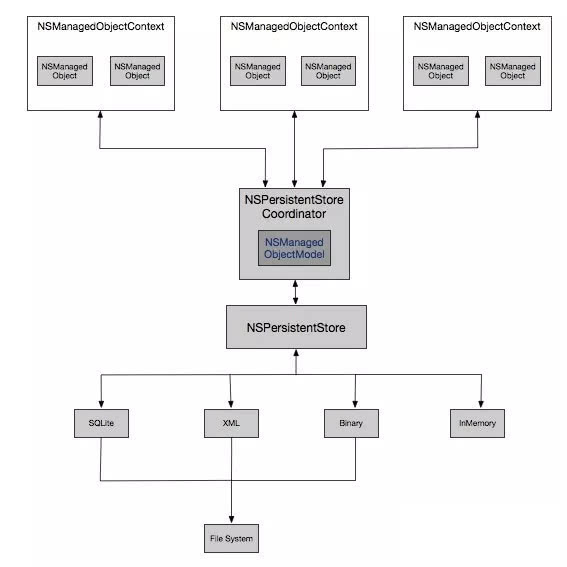


图14-8

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.”。

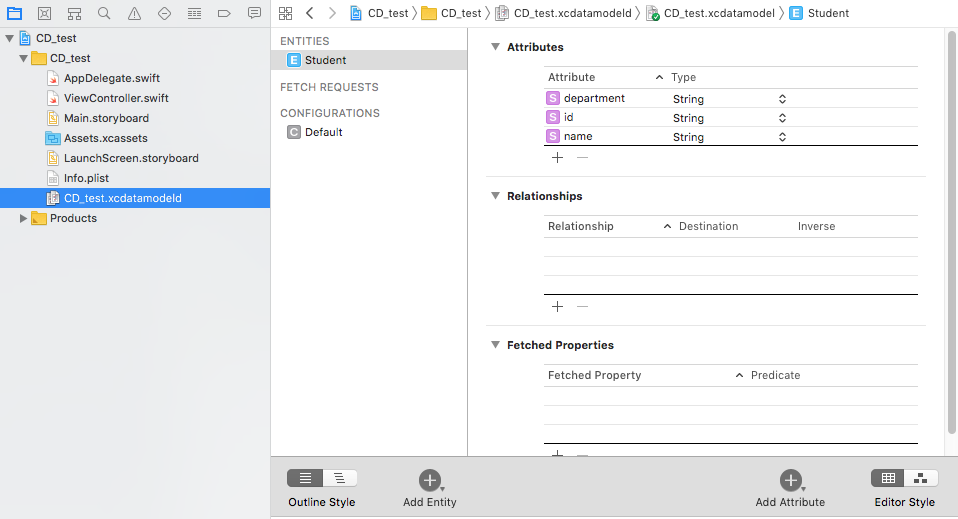


图14-9 CoreDate实体属性

### 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。

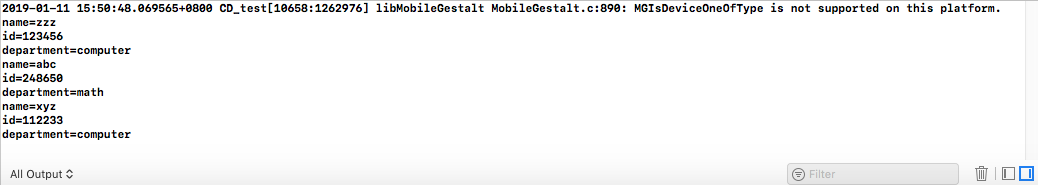


图14-11

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。

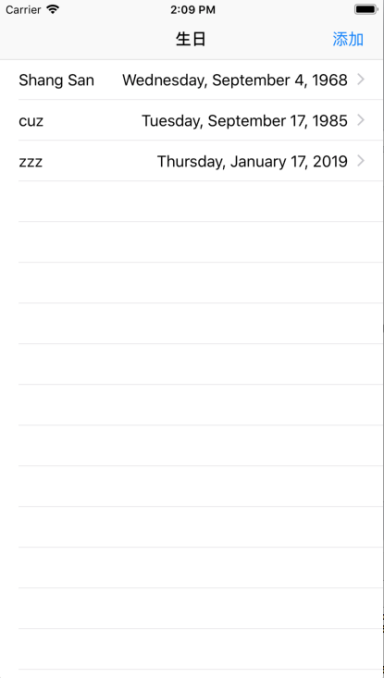


图15-1 生日记录薄App界面

## 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。

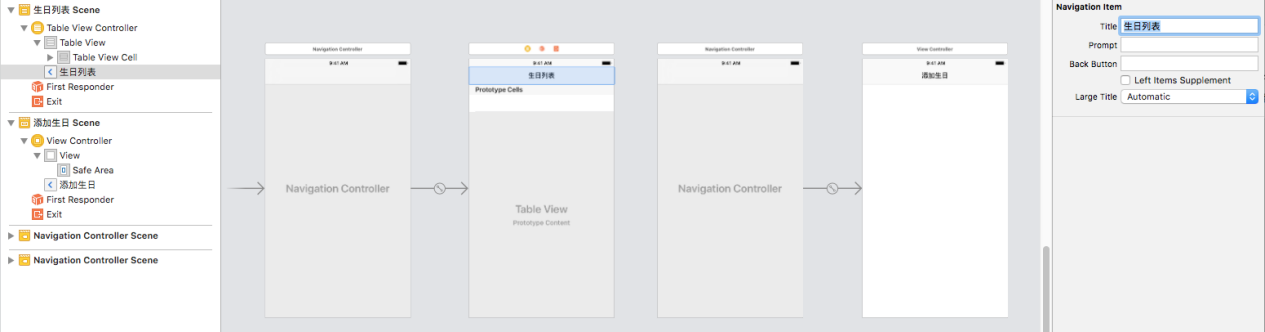


图15-2 storyboard显示结果

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。

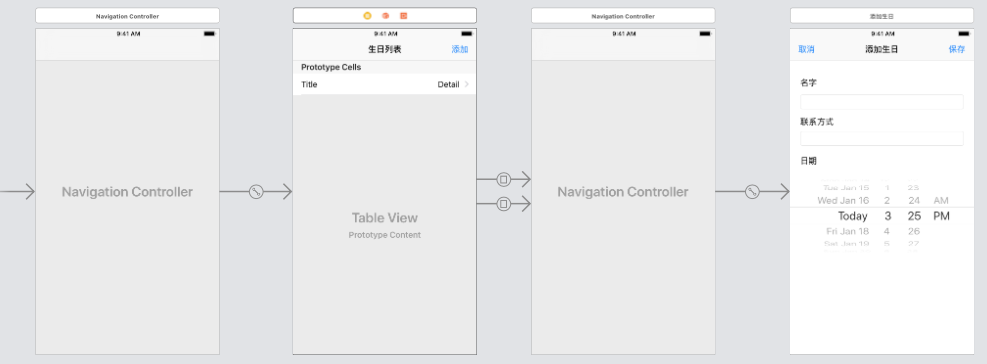


图15-3 界面布置结果

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。

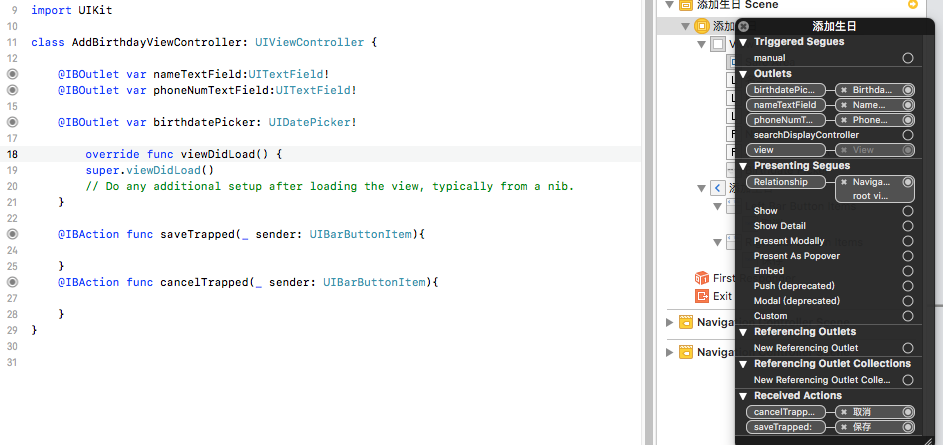


图15-4 代码与控件连接

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。

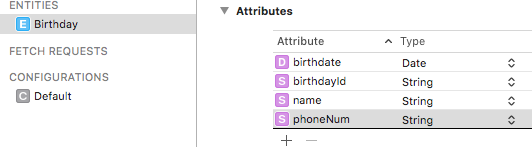


图15-5 CoreData实体及其属性

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。