第1章

开发准备

工欲善其事，必先利其器。在学习 iOS 移动开发之前，首先应该将开发环境配置完成并对所需要使用的开发工具进行了解与熟悉。本章首先向读者介绍 iOS 11 系统相比之前系统的一些新特性，使读者对目前主流的 iOS 系统有一个宏观上的了解，然后将一步步演示开发环境的搭建并介绍开发工具 Xcode 的常用功能。

通过本章的学习，读者能够掌握：

- 了解 iOS 11 的新特性和新功能。
- 申请免费的 Apple ID 账号。
- 使用 Xcode 开发工具创建 iOS 工程。
- 使用 Xcode 开发工具编写与调试程序。
- 熟悉 Xcode 工程结构。
- 编写第一个程序 Hello World。
- 使用 Git 工具进行版本管理。
- 使用 GitHub 代码托管平台。

1.1 iOS 11 新特性简述

随着 2017 年 iPhone X 的发布，iOS 系统也迎来了它的又一次重大升级。新版本 iOS 11 在用户交互维度新增了拖放操作，用户在 iOS 系统中可以将元素在不同界面间或不同 APP 间进行传递。在文件管理方面，iOS 11 提供了本地文档与 iCloud 云文档浏览器。除了一些细节上的调整和优化，iOS 11 开发框架中还新增了目前非常受欢迎的机器学习模型和 AR 开发框架。另外，Swift 语言也升级到了 4.0 版本，Swift 语言自从发布以来，就一直褒贬各半，首先其先进的现代编程语言特性的确有很强的安全性和编程效率，另一方面其频繁的更新和接口变动也给开发者的项目维护带来很

大的麻烦。从 Swift 语言的更新内容和趋势来分析，3.0 是一个分界点，Swift 3.0 对语言风格和 API 做了重构性质的升级，4.0 版本则只是完善与补充，并没有比较突出的修改，因此无论是学习还是开发项目，目前的 Swift 语言都是非常适合的。了解 iOS 11 的这些新特性与 Swift 4.0 的相关更新点，相信可以更好地帮助读者学习 iOS 应用程序开发。

1.1.1 新增拖放交互编程接口

在 Mac OS 软件开发时，拖拽交互是一种十分常用的交互方式，在 iOS 11 之前的系统中要实现拖拽交互往往比较困难。iOS 11 系统新引入了拖拽相关的 API，可以帮助开发者快速构建拖拽交互，在 iOS 11 系统中，使用这种 API 进行 APP 的开发为设计提供了一种全新维度的用户交互方式。

拖拽操作在 iPad 上是支持跨应用程序的，用户可以从一个应用中拖取项目，通过 Home 键回到主界面并打开另一个应用程序，然后将被拖拽的项目传递给这个应用程序中。在 iPhone 上，拖拽操作只支持当前应用程序内，用户可以将某个元素从一个界面拖拽到另一个界面，这种维度的操作可以使设计人员在设计产品时，有更大的灵活性。

对于拖拽操作，至少要有两个组件：一个组件作为拖拽源用来提供数据；另一个组件作为拖拽目的用来接收数据。当然，同一个组件既可以是拖拽源也可以是拖拽目的。

任意的 UIView 组件都可以作为拖拽源，让其成为拖拽源其实也十分简单，只需要 3 步：

步骤01 创建一个 UIDragInteraction 行为对象。

步骤02 设置 UIDragInteraction 对象的代理并实现相应方法。

步骤03 将 UIDragInteraction 对象添加到指定的 View 上。

最简单的可拖拽组件的创建示例代码如下：

```
        lazy var dragView = { ()->UIView in
            let view = UIView(frame: CGRect(x: 100, y: 100, width: 100, height: 100))
            view.backgroundColor = UIColor.red
            view.addInteraction(self.dragInteraction)
            return view;
        }()

        lazy var dragInteraction = { ()->UIDragInteraction in
            let dragInteraction = UIDragInteraction(delegate: self)
            dragInteraction.isEnabled = true
            return dragInteraction
        }()

        func dragInteraction(_ interaction: UIDragInteraction, itemsForBeginning session:
UIDragSession) -> [UIDragItem] {
            let provider = NSItemProvider(object: "Hello World" as NSItemProviderWriting)
            let item = UIDragItem(itemProvider: provider)
            return [item]
        }

        override func viewDidLoad() {
            super.viewDidLoad()
            self.view.addSubview(self.dragView)
```

```
        }
```

拖拽源是数据的提供者，放置目的地就是数据的接收者。同样，对于任何自定义的 UIView 视图，我们也可以让其成为放置目的地，需要以下三步完成：

步骤01 创建一个 UIDropInteraction 行为对象。

步骤02 设置 UIDropInteraction 对象的代理并实现协议方法。

步骤03 将其添加到自定义的视图中。

例如，我们将自定义的 UILabel 组件用来显示拖拽的文案，代码如下：

```
    import UIKit
    class ViewController: UIViewController,UIDragInteractionDelegate,UIDropInteractionDelegate
{
        lazy var dragView = { ()->UIView in
            let view = UIView(frame: CGRect(x: 100, y: 100, width: 100, height: 100))
            view.backgroundColor = UIColor.red
            view.addInteraction(self.dragInteraction)
            return view;
        }()
        lazy var dragInteraction = { ()->UIDragInteraction in
            let dragInteraction = UIDragInteraction(delegate: self)
            dragInteraction.isEnabled = true
            return dragInteraction
        }()
        lazy var dropLabel = {()->UILabel in
            let label = UILabel(frame: CGRect(x: 10, y: 300, width: 300, height: 30))
            label.backgroundColor = UIColor.green
            label.isUserInteractionEnabled = true
            label.addInteraction(self.dropInteraction)
            return label
        }()
        lazy var dropInteraction = {()->UIDropInteraction in
            let dropInteraction = UIDropInteraction(delegate: self)
            return dropInteraction
        }()

        func dropInteraction(_ interaction: UIDropInteraction, canHandle session: UIDropSession)
-> Bool {
            return true
        }
        func dropInteraction(_ interaction: UIDropInteraction, sessionDidUpdate session:
UIDropSession) -> UIDropProposal {
                return UIDropProposal(operation: .copy)
        }
        func dropInteraction(_ interaction: UIDropInteraction, performDrop session: UIDropSession)
{
            let _ = session.loadObjects(ofClass: String.self) { (itemArray) in
                self.dropLabel.text = itemArray.first
            }
        }
        func dragInteraction(_ interaction: UIDragInteraction, itemsForBeginning session:
UIDragSession) -> [UIDragItem] {
            let provider = NSItemProvider(object: "Hello World" as NSItemProviderWriting)
```

```
            let item = UIDragItem(itemProvider: provider)
            return [item]
        }
        override func viewDidLoad() {
            super.viewDidLoad()
            self.view.addSubview(self.dragView)
            self.view.addSubview(self.dropLabel)
        }
    }
```

上面的代码将我们自定义的拖拽源提供的 Hello World 拖放进了 UILabel 组件中。

1.1.2 其他新增功能

在 iOS 11 系统中新增了访问本地和 iCloud 文档的功能，开发者可以使用 UIDocumentBrowserViewController 和 UIDocumentBrowserTransitionController 这两个视图控制器来管理文档浏览。

新增了 MusicKit 开发框架，开发者可以在应用中访问完整的 Apple Music 音乐目录，并且提供了更多与 Apple 音乐程序的交互接口。

新增了 ARKit 开发框架，开发者结合摄像头可以更加容易地构建 AR 体验项目。

新增了用于检测人脸识别、条形码等视觉效果的开发框架。

新增了 CoreML 开发框架，开发者更便于将机器学习模型集成到应用中。

1.2 熟悉 iOS 开发环境

Xcode 是进行 iOS 应用开发必备的开发软件。Xcode 开发工具功能十分强大且简单易用，不需要过多的配置，下载并安装后，各种环境和模拟器即关联完毕，对于初学者来说门槛很低。

1.2.1 安装 Xcode 开发工具

由于 iOS 系统的封闭性，开发 iOS 软件的工具环境并不多，Xcode 是 Apple 公司自己开发的一套针对 OS、iOS、watchOS 和 tvOS 的开发环境，使用方便且功能十分强大。用户可以在 App Store 上免费获取 Xcode 开发工具。

首先需要申请个人的 AppleID。AppleID 是 Apple 会员的凭证，也是个人的信息管理凭证。申请个人的 AppleID 是免费的，登录 www.apple.com/cn Apple（中国）官方网站，在屏幕右上角的购物袋按钮中选择“登录”选项，如图 1-1 所示。

图 1-1　登录 Apple（中国）官方网站

在登录界面右下侧选择创建一个 Apple ID，如图 1-2 所示。

图 1-2　创建 Apple ID

然后按照页面中的提示填写相应的信息。需要注意的是，填写的邮箱务必要真实，注册 Apple ID 时会要求进行邮箱验证。

Apple ID 申请成功之后，就可以从 App Store 获取 Xcode 开发工具了。打开 App Store，如图 1-3 所示。

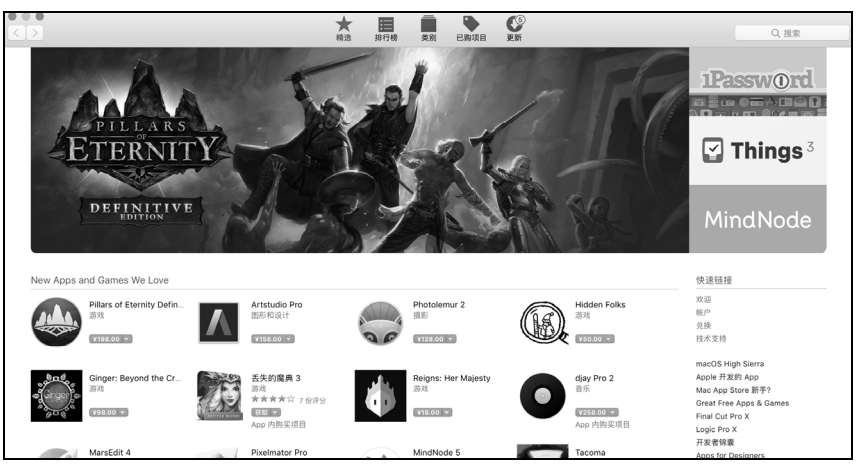

图 1-3　App Store 应用市场

在右上角的搜索框中输入 Xcode，按 return 键进行搜索，会搜索出许多应用，其中第一个就是开发者需要的 Xcode 开发工具，单击“获取”按钮即可安装到我们的电脑中，如图 1-4 所示。

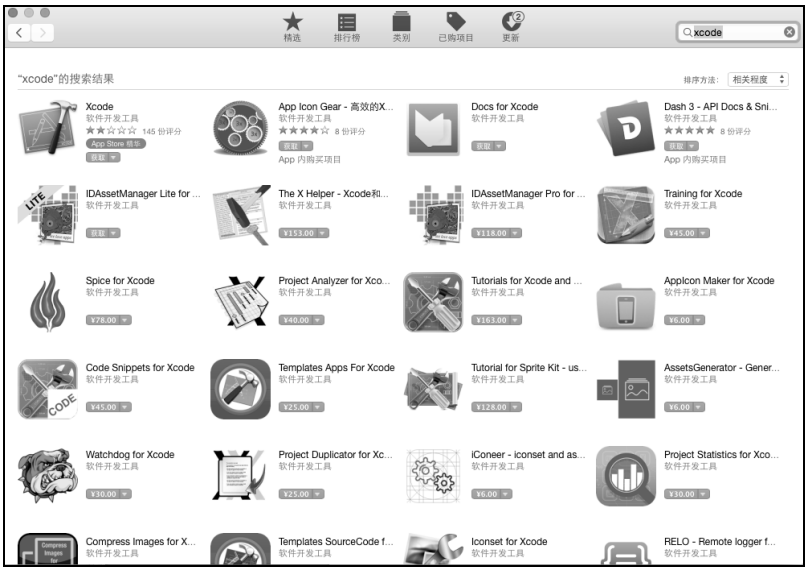

图 1-4　获取 Xcode 开发工具

1.2.2　了解 Xcode 开发工具主界面

打开 Xcode 开发工具进入编码界面，可以发现界面有 4 部分组成，如图 1-5 所示。

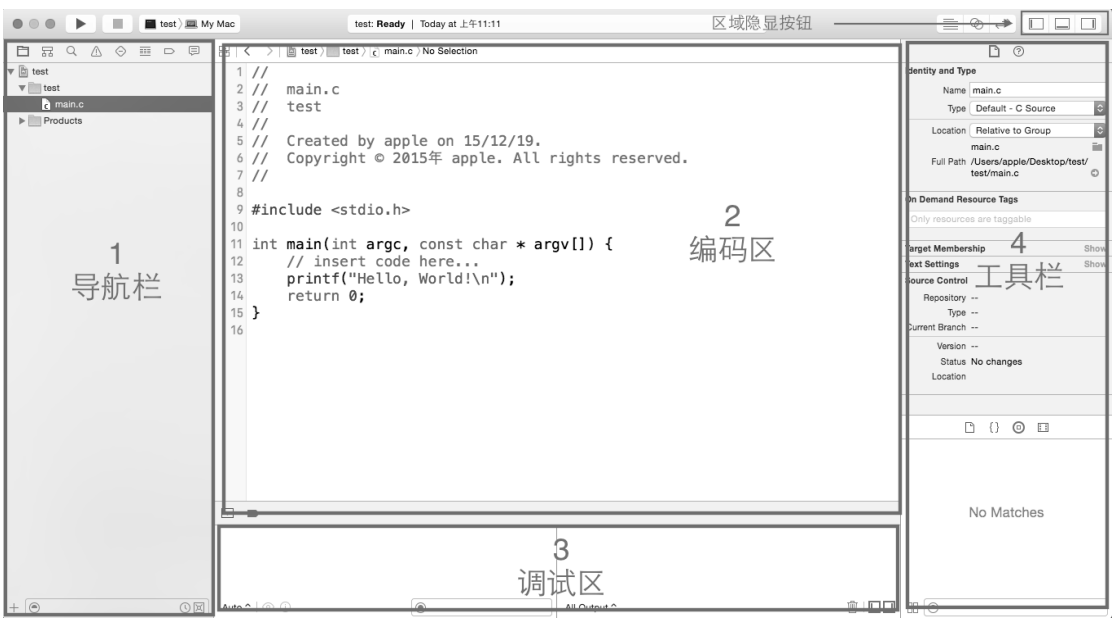

图 1-5　Xcode 开发工具的编码界面

界面最左边是导航栏，其中展示文件目录索引、关键字搜索索引、错误警告索引、断点调试

索引等；中间区域是编码的主要区域，在这个区域中编写相关的程序代码；下边是 Debug 调试区域，代码中的打印信息会展示在这个区域中；最右边是工具栏，用于设置当前编写文件的相关属性。界面的右上角有 3 个按钮，从左向右分别对应导航栏、调试区和工具栏的显隐，在编码时可以将暂时不需要的区域隐藏，扩大编码区域。

图 1-6　程序调试相关功能

在 Xcode 工具主界面的左上角也有一些按钮，如图 1-6 所示。

其中“运行”按钮可以编译并运行项目，“选择项目”按钮可以选择需要运行的项目，“选择运行设备”按钮可以对运行的平台进行选择。

1.2.3　Xcode 开发工具的使用技巧及常用快捷键

熟练使用 Xcode 工具可以使开发变得事半功倍，Xcode 也有许多附加功能可帮助开发者更高效地编写代码。选择 Xcode 标签导航中的 Preferences 选项，如图 1-7 所示，之后会弹出如图 1-8 所示的个人偏好窗口。

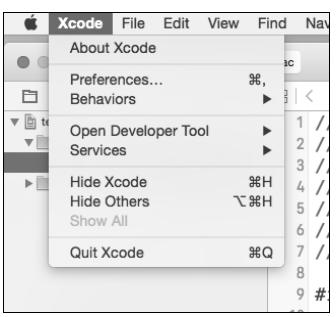

图 1-7　选择 Xcode 菜单选项

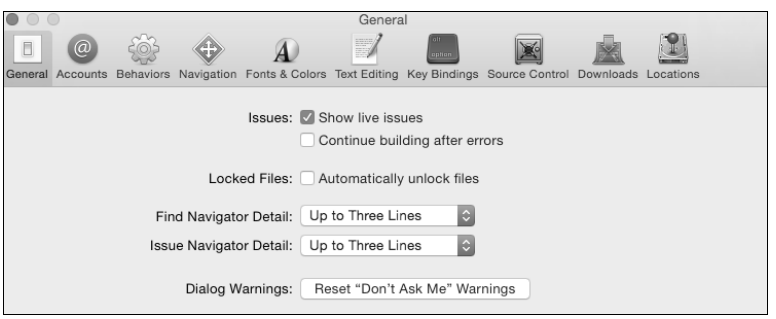

图 1-8　Xcode 个人偏好设置

上面的 10 个标签分别对应 Xcode 的一些偏好设置属性，在 Fonts&Colors 标签中可以进行代码高亮风格和字体大小的设置，如图 1-9 所示。

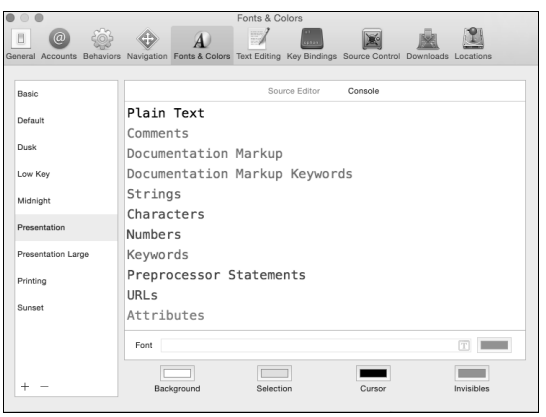

图 1-9　风格字体设置界面

在 Text Editing 标签中可以设置一些编辑代码的属性选项，如图 1-10 所示。

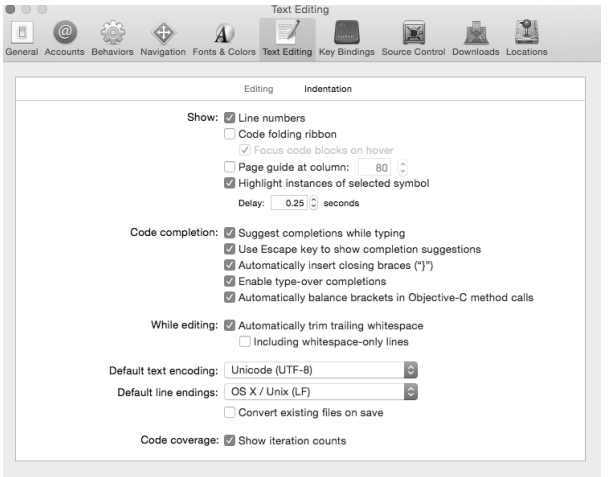

图 1-10　代码编辑选项

其中 Line numbers 可以设置是否显示代码行号，在编写代码时最好选中这一项，对开发者调试代码定位问题十分重要。

除了对 Xcode 进行相关偏好设置可以帮助开发者更便捷地编码外，熟练使用 Xcode 开发工具中的快捷键也可以使开发事半功倍。下面列出了开发中常用的一些 Xcode 快捷键：

- 新建项目 command+shift+n
- 新建文件 command+n
- 打开 command+o
- 关闭窗口 command+w
- 项目中查找 command+shift+f
- 编译并运行 command+r
- 注释 command+/
- 文件首行 command+上箭头
- 文件末行 command+下箭头
- 生成函数注释 option+command+/
- 行首 command+左箭头
- 行末 command+右箭头
- 上一单词 option+左箭头
- 下一单词 option+右箭头
- 删除此行光标前所有内容 control+delete
- 断点 command+option+b
- 当前行插入断点 command+\
- 查看开发文档 command+option+click

1.3　创建第一个 iOS 项目

很多程序开发者都有 Hello world 情愫，很多优秀的开发者也是通过 Hello World 进入程序世界的。一个复杂的 iOS 工程的起始和最简单的工程 Hello World 有着相同的结构，因此学习 iOS 程序开发时，从 Hello World 工程开始可以快速便捷地了解工程结构。

打开 Xcode 开发工具，在 Welcome 界面选择 Create a new Xcode project 选项新建一个工程，如图 1-11 所示。

图 1-11　使用 Xcode 创建一个新的工程

在选择模板窗口中选择 Single View Application，如图 1-12 所示。

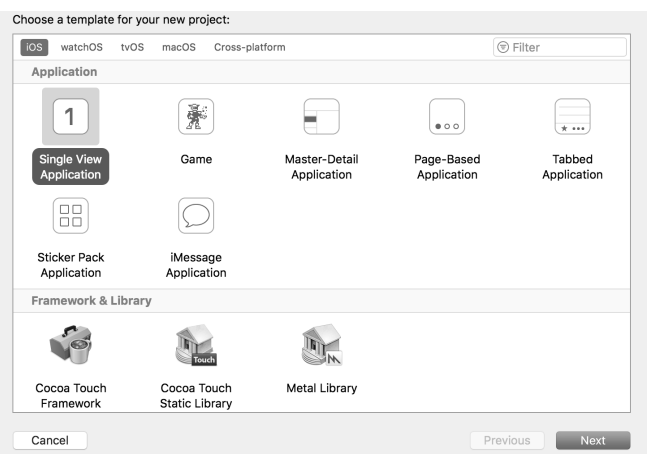

图 1-12　选择工程模板

在模板设置窗口中可以对项目的一些基本属性进行设置，如图 1-13 所示。

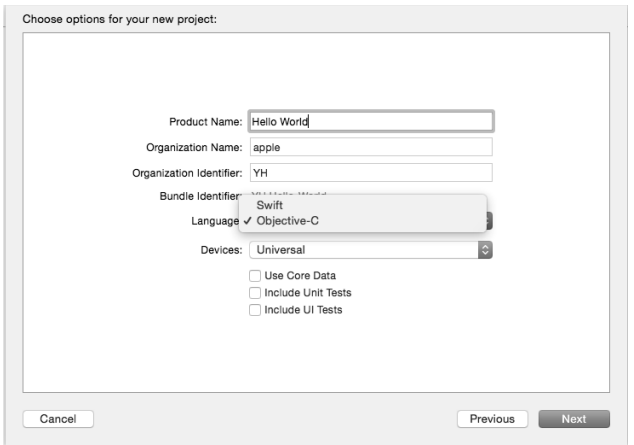

图 1-13　设置工程配置选项

Product Name 用于设置项目的名称；Organization Name 用于填写公司或组织的名称；Organization Identifier 用于填写公司或组织的标识符；Bundle Identifier 是当前项目的标识符（是此应用在 App Store 中的唯一标识符）；Language 可以选择开发项目使用的语言，iOS 项目目前支持 Swift 和 Objective-C 两种语言，本书将采用 Swift 语言进行代码的实战演练；Devices 选项设置支持的设备，可以选择 iPhone、iPad 或 Universal（通用）。

将上面的信息设置好之后，单击 Next 按钮进行工程路径的选择，最后单击 Create 按钮进行工程的创建。

工程创建完成后，就来到熟悉的 Xcode 编码主界面，左侧导航栏中有使用模板帮助开发者创建好的文件层次，工程结构如图 1-14 所示。

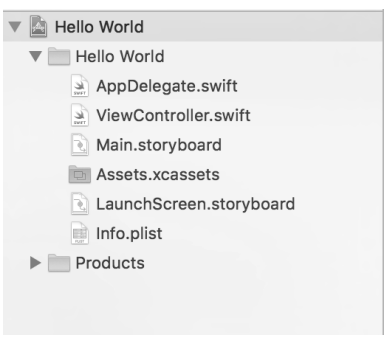

图 1-14　工程目录结构

在 Hello World 工程中有两个主文件夹，即 Hello World 文件夹和 Products 文件夹。Hello World 文件夹存放开发编码文件，Products 文件夹存放编译后的包文件。Hello World 文件夹中的文件是重点需要理解的内容。

在整个工程中，AppDelegate 文件是整个程序的入口，也可以理解为 iOS 程序运行的代理。ViewController 文件是模板自动创建出展示在设备屏幕上的一个视图控制器，与 Main.storyboard 中的视图控制器关联；Main.storyboard 文件是可视化的视图编辑器文件，通过可视化的编辑工具，开发者可以更加快速地对程序界面部分进行开发；Assets.xcassets 文件是图片素材文件管理器，如果项目中需要使用一些图片素材，就可以将图片放入这个管理器中；LaunchScreen.storyboard 是项目初启画面的视图管理器，Info.plist 文件中则保存了项目的一些配置信息。

打开 Main.storyboard 文件，Xcode 的编码区变成了可视化的视图编辑区，取消选中 Use Size Classes，使其只适配 iPhone，如图 1-15 所示。

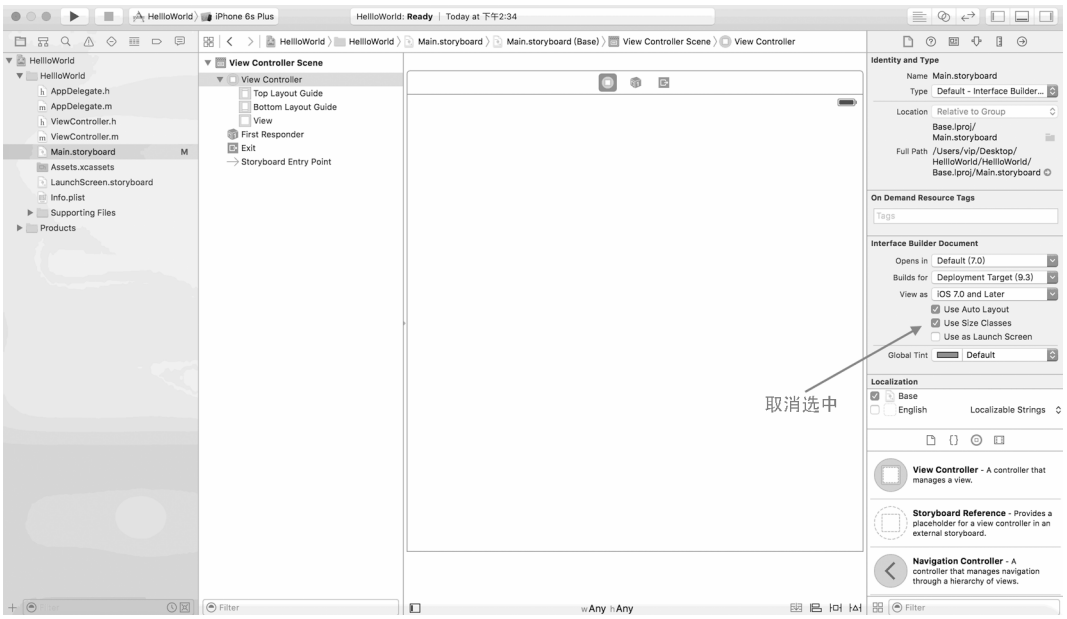

图 1-15　设置适配模式

在编辑器的右下方找到 Label 标签控件，如图 1-16 所示。

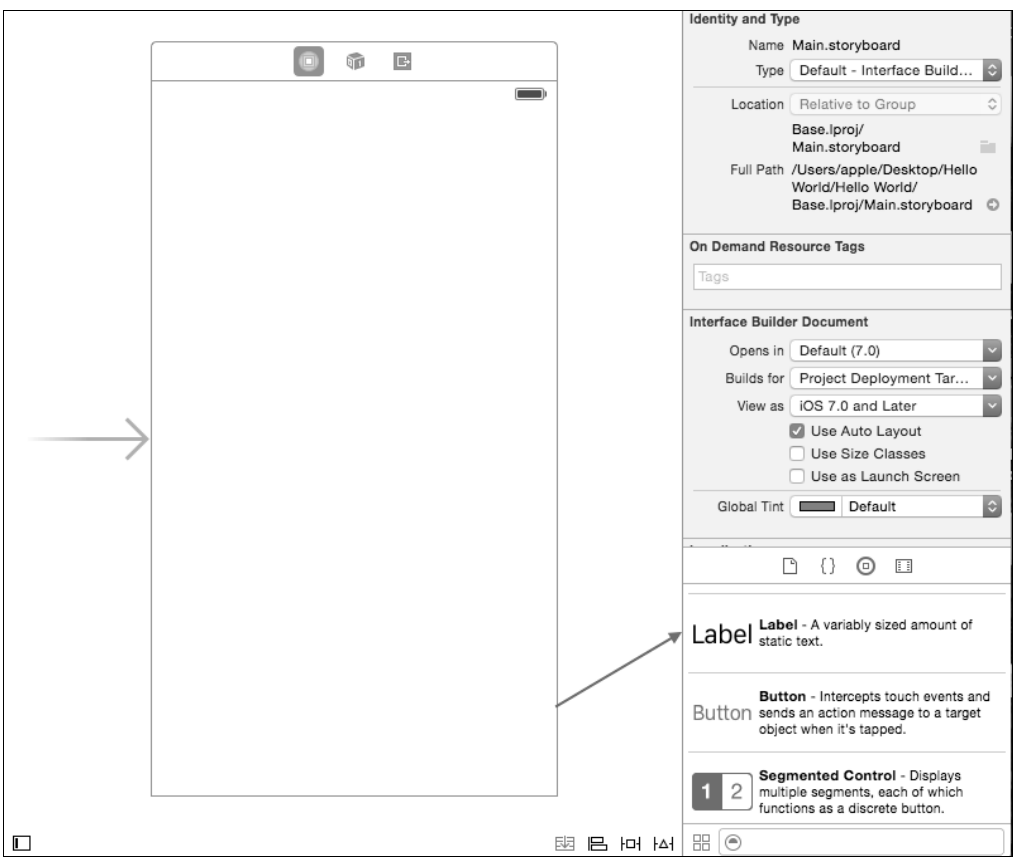

图 1-16　storyboard 文件中的 Label 控件

使用鼠标选中 Label 控件不放，将其拖动到视图控制器的中间。双击视图控制器上的 Label，在其中写入 Hello World 字样，然后单击 Xcode 左上角的“运行”按钮，选择一个模拟器。这时，第一个 iOS 程序就展现在我们面前了，模拟器屏幕上出现了 Hello World 标签，如图 1-17 所示。有没有小激动一下，iOS 程序的世界欢迎你的到来。

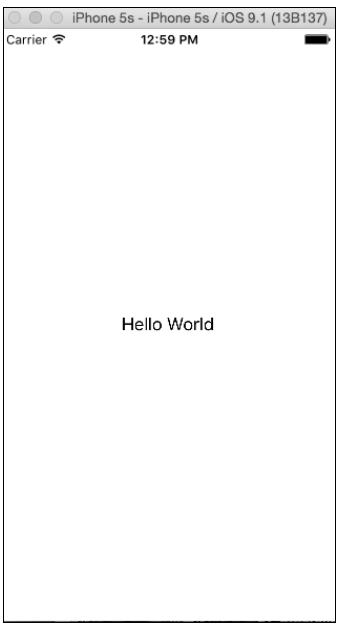

图 1-17　运行 Hello World 工程

1.4　使用 Git 进行项目版本管理

1.4.1　Git 与 Github 简介

在项目开发中使用一个版本控制工具是必不可少的，Git 是一个开源的分布式版本控制系统，它可以协同多人更加高效地协作开发，同时可以帮助开发者根据不同的目的进行项目的分支管理。GitHub 是一个代码托管系统，世界各地的开源项目都可以免费在其上面托管。将一个 Git 管理的仓库托管在 GitHub 上可以实现多个开发者参与、多地点同时协作的开发方式，这将大大提高项目开发的效率。

1.4.2　注册 GitHub 会员

GitHub 免费为开源项目提供代码托管平台，若要使用 GitHub 提供的服务，首先需要注册成为 GitHub 会员。在浏览器中打开 https://github.com，因为 GitHub 服务器部署在国外，打开速度或许有些缓慢。打开后的页面如图 1-18 所示。

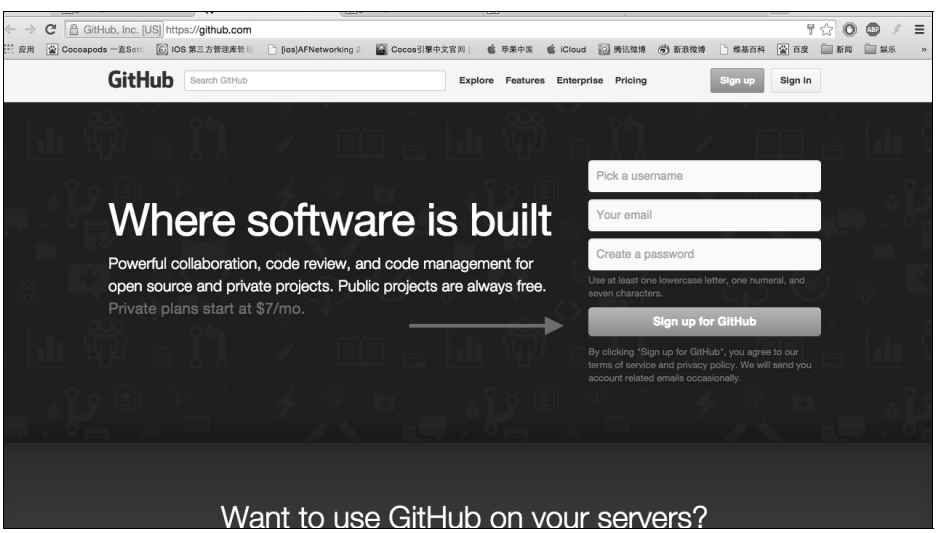

图 1-18　GitHub 主页

单击 Sign up for GitHub 按钮进入注册页面，如图 1-19 所示。在该页面中填写一些基本的信息，这时 github 会对用户是否重复、邮箱是否正确等进行检查，无误后单击 Creat an account 按钮。

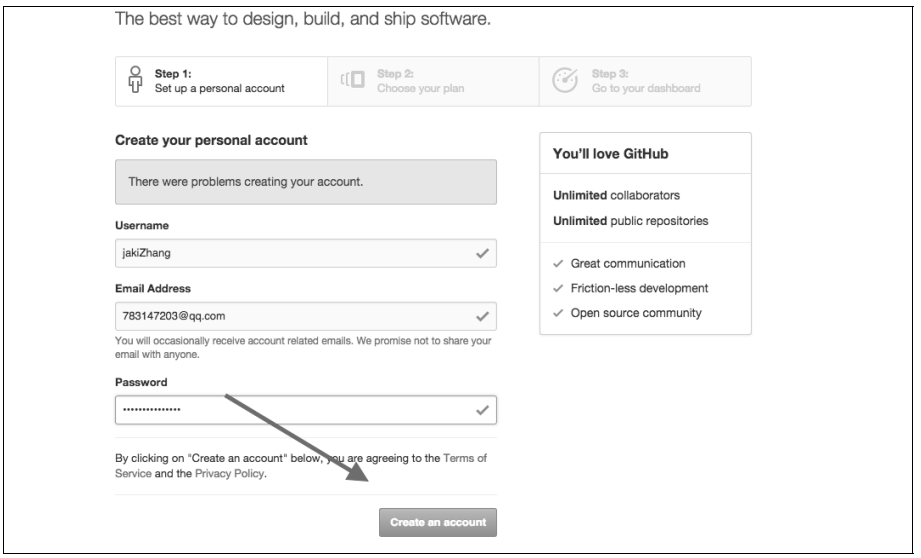

图 1-19　填写注册信息

如果注册成功，GitHub 就会让用户选择服务类型，个人开发者可以选择 free，如图 1-20 所示。单击 Finish sign up 按钮，一个属于用户自己的 GitHub 账号就创建成功了。

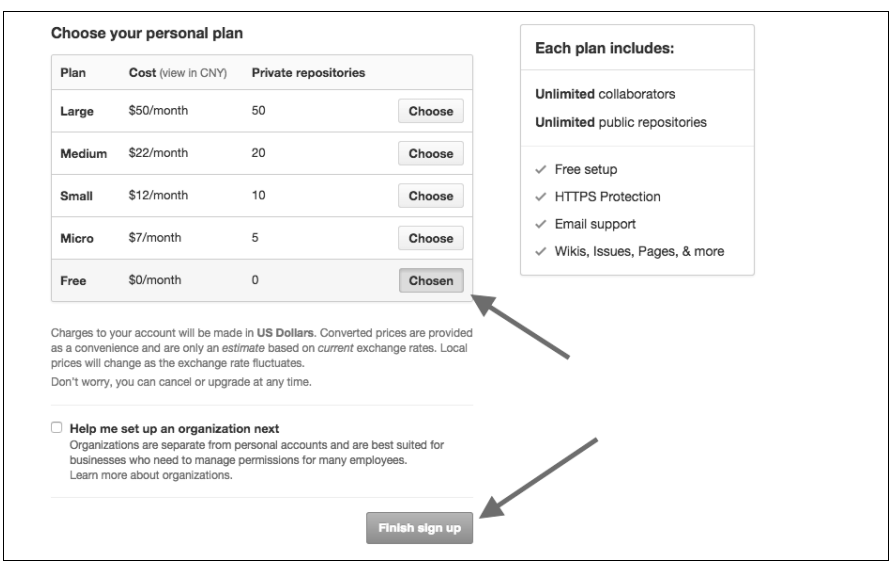

图 1-20　选择服务类型

1.4.3　使用 Xcode 创建 Git 仓库

Xcode 是系统一体化性很强的 iOS 开发工具，在安装 Xcode 时默认安装了 Git 工具，我们在创建工程的时候可以选择创建本地 Git 仓库，如图 1-21 所示。

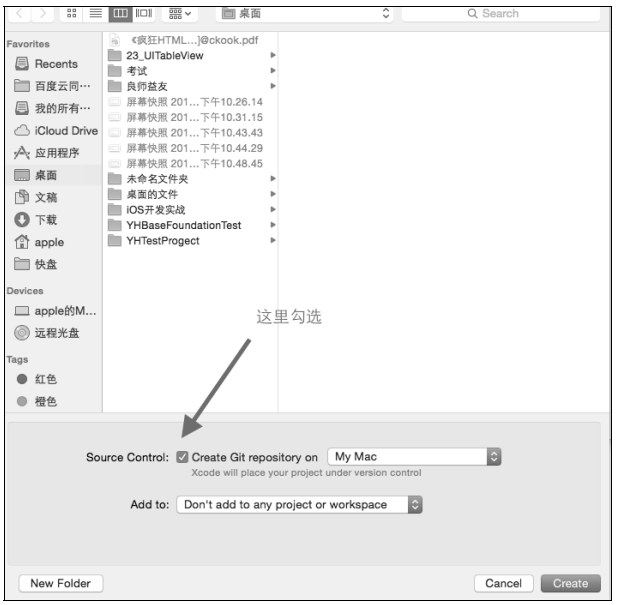

图 1-21　创建 Git 仓库

之后在 Xcode 工具导航中的 Source Control 标签中可以看到当前项目的仓库，如图 1-22 所示。在开发中，这里面代码版本管理的功能将大有用处。

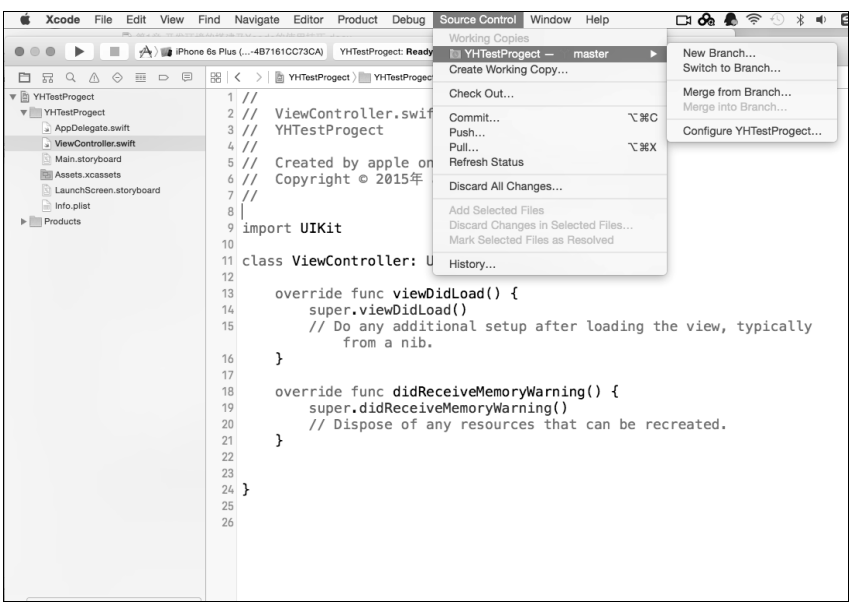

图 1-22　Xcode 的代码管理功能

New Branch 可以创建一个新的分支，创建的分支是原工程代码一个副本，但是可以在不影响其他分支的情况下独立开发新的扩展。举个例子，项目的初始版本是 1.0，现在需要开发 2.0 版本，开发者完全可以在 1.0 版本的基础上拉出一个 2.0 分支，在 2.0 分支上做的开发工作都不会影响 1.0 版本。

Switch to Branch 提供切换分支功能，开发者可以在多个分支之间自行切换，灵活开发。

Merge from Branch 可以进行分支的合并，在开发中这也是一个很强大的功能。例如，开发者需要在当前工程中添加一个风险较大的模块，这时可以拉出一个新的分支，在新的分支上进行开发，开发完成并且测试没有问题之后，可以在原分支上使用 Merge from Branch 进行代码合并。

Check Out 可以从远端检测出默认分支，这个功能使用时要特别注意，如果本地分支中文件有改动，将会被覆盖。

Commit 可以将改动的代码提交到本地，提交时会有提交用户记录和备注操作。

Push 功能将本地的改动推送到远端服务器，如推送到 GitHub 平台进行托管。

Pull 功能与 Push 对应是从远端服务器拉取有更改的代码。

1.4.4　用 Xcode 建立本地 Git 仓库与 GitHub 代码托管平台的关联

前面已经创建了 GitHub 代码托管平台账号和本地 Git 仓库。Git 仓库用于代码版本的本地控制，GitHub 平台帮助多地、多人合作开发，两者结合才能最高效地进行项目的开发。首先，需要在 GitHub 平台上创建一个远程 repository（仓库），利用申请好的账号登录 GitHub 进入主页，单击 New repository 按钮，如图 1-23 所示。注意，如果是新创建的 GitHub 账号，就需要先进行邮箱验证。

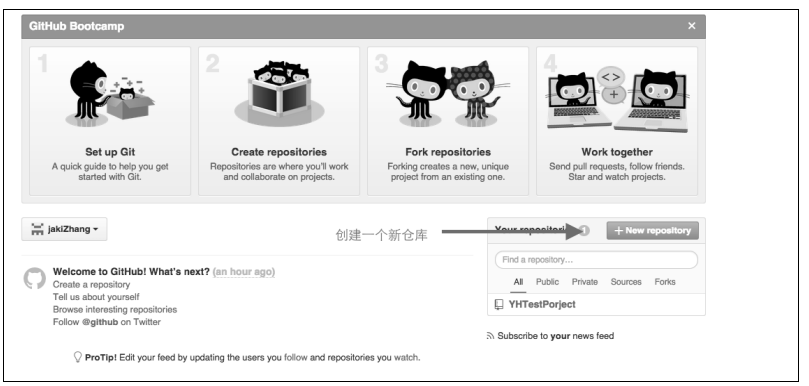

图 1-23　新建代码仓库

这样一个远程仓库就创建好了，只是目前空空如也，如图 1-24 所示。GitHub 为这个仓库分配了一个远程的地址，通过这个地址，开发者可以将其与本地的仓库关联，进行多人远程协作。

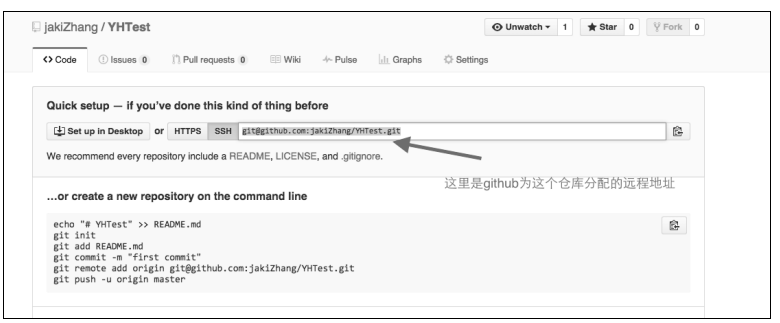

图 1-24　代码仓库地址

再回到 Xcode 工具导航，在 Source Control 标签中选择项目的本地仓库，选择 Configure

YHTestProgect，如图 1-25 所示。

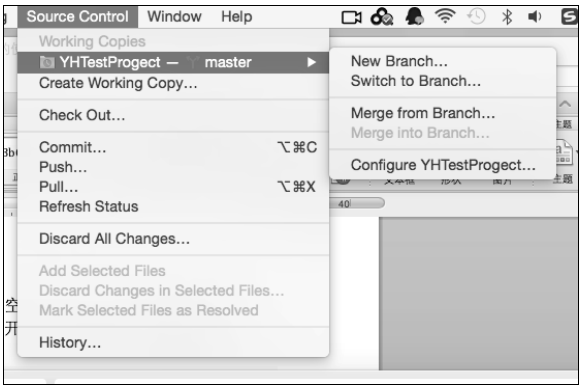

图 1-25　进行代码仓库设置

在弹出的设置菜单中的 Remotes 标签里单击加号，选择 Add Remote，如图 1-26 所示。

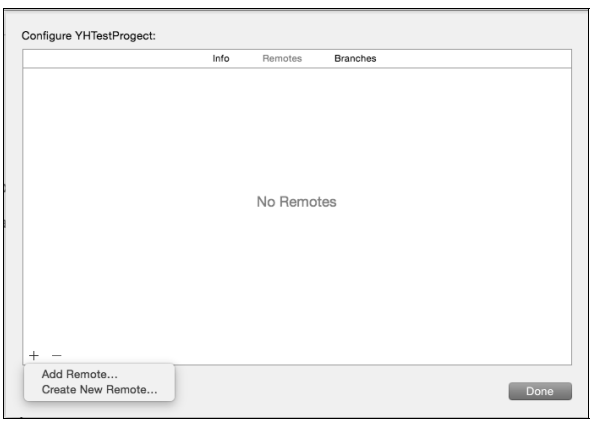

图 1-26　关联一个远程仓库

如图 1-27 所示，输入 GitHub 远程仓库的地址，并单击 Add Remote 按钮。

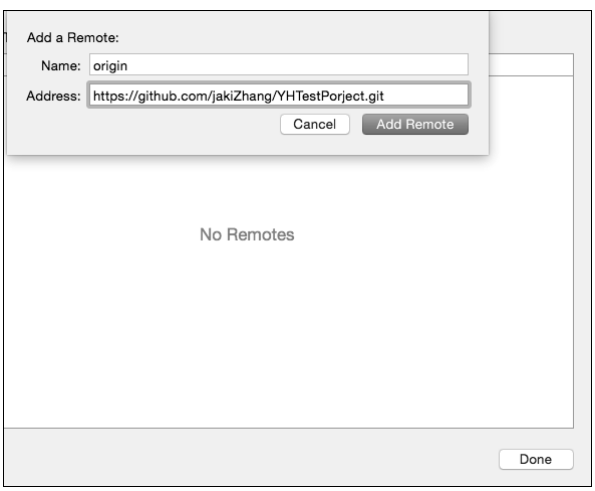

图 1-27　关联远程仓库地址

然后使用 Push 功能将本地的代码 Push 到 GitHub 上。第一次使用时，Push 需要输入用户名和密码，使用 GitHub 账号的用户名和密码即可。注意，这里的用户名不是邮箱，是 GitHub 会员用户名，如图 1-28 所示。如果 Push 成功，本地的 Git 仓库就和托管在 GitHub 上的仓库进行了关联，我们可以随时随地更新代码到 GitHub 上，也可以将 GitHub 上更新的代码拉到本地来。

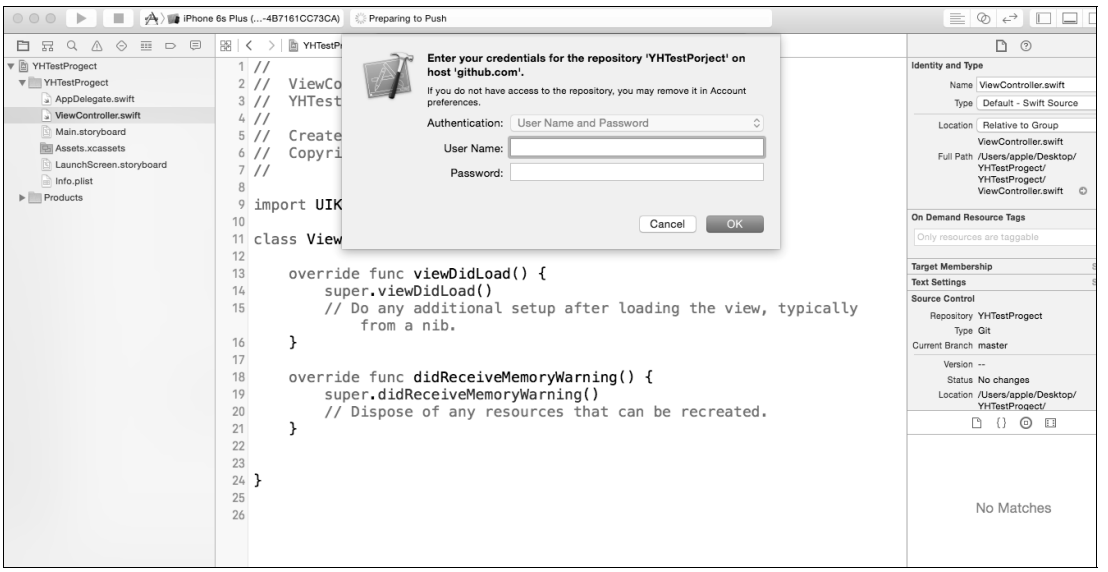

图 1-28　进行代码管理操作

第 2 章

基础 UI 组件

UI 界面是连接用户与应用程序的桥梁，通过美观易用的图形界面，用户可以很快熟悉应用程序的功能和操作方法，大大节约学习使用软件的时间成本。同时，成功的 UI 设计不仅使用户赏心悦目，在许多情况下也会引导用户向正确的方向操作。iOS 系统的 UI 框架十分简约美观，并且易于开发者进行自定义，本章将向读者介绍 iOS 系统的 UI 体系架构和 iOS 应用开发中常用的一些独立 UI 控件。

通过本章的学习，读者能够掌握：

- 基础 UI 控件的使用和其属性的自定义
- 通过可交互的控件实现与用户操作进行逻辑交互
- 了解代理设计模式
- 通过控件代理方法监听用户操作
- 灵活组合使用控件进行 UI 开发
- 实战中项目的搭建步骤与界面间的跳转

2.1 iOS 系统 UI 框架的介绍

在 iOS 系统的开发框架中，UIKit 是专门负责界面渲染的一个框架，其中不仅封装了许多开发中常用的 UI 控件（如 UILabel（标签）控件、UIButton（按钮）控件等），还集成了建立视图联系的视图控制器和交互用户操作的用户触摸事件等。UIKit 框架的结构十分复杂，学习 UIKit 框架时，首先需要从大局着眼，熟悉框架的构成以及与其中模块间的联系，然后从细节入手，从常用的控件类学起，多多积累，时常练习，由局部到整体逐渐掌握 iOS 开发中 UI 的相关部分。

如图 2-1 所示为 UIKit 框架中类的结构与简易关系。UIKit 框架中大致分为三部分：

（1）独立的 UI 视图控件。

（2）充当视图控件载体的控制器类。

（3）管理触摸事件、手势操作、键盘操作等交互的管理类。

独立于 UIKit 框架外，代理和通知中心的设计模式为开发者提供了 UI 属性变化的回调接口。由于各个模块是独立的，因此开发者可以十分方便地进行 UI 上的自定义与扩展操作，同时各个模块又各有接口关联，在数据传递和逻辑交互上也井井有条毫不混乱。

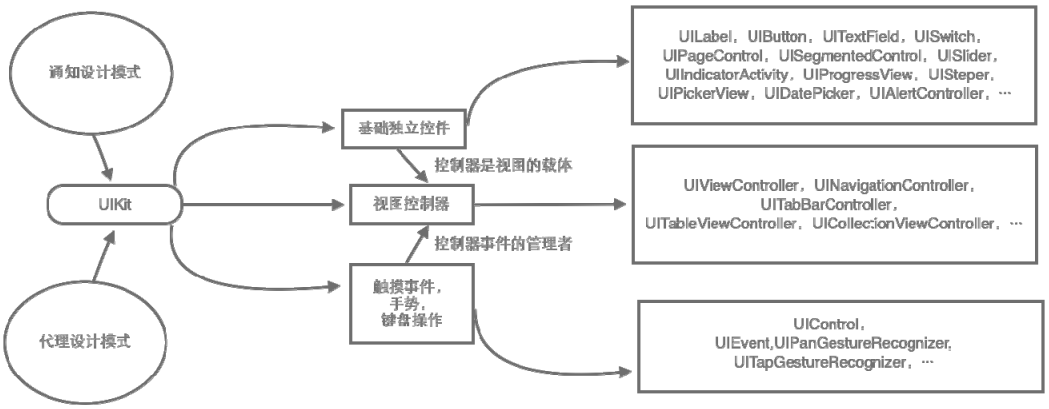

图 2-1　UIKit 框架的类结构与简易关系图

2.1.1　MVC 设计模式

MVC 模式是一种传统的软件设计模式，全称为 Model（模型）-View（视图）-Controller（控制器）。在这种模式中，重新设计 UI 界面时不需要重复编写数据和逻辑相关的代码。在 MVC 模式中，数据由数据模型 Model 层管理，UI 视图由 View 层管理，将 View 与数据进行关联，处理逻辑的相关操作交由控制器 Controller 层进行管理，View 层与 Controller 层的分离使得许多 View 可以进行复用，在一些开源的开发者社区也可以看到大量自定义的 View 层控件，在项目中几乎不需要修改就可以直接使用。Model 层与 View 层的分离使得 Model 层不再依赖 View 层，在数据模型没有改变的情况下，View 层可以方便地重新进行设计。

在 UIKit 框架中，基础的独立控件就充当 View 层，控制器部分充当 Controller 层，数据层一般由开发者自行进行模型的设计和编写，在 Controller 中通过控件的功能接口将数据与控件关联。

2.1.2　代理设计模式

代理设计模式是软件编程中进行数据传递的一种重要方式，这种模式十分抽象，但在 UIKit 框架中却随处可见。举一个例子来说明，一个盲人背着一个瘸子走路，瘸子看到前面有一个水坑，他告诉盲人让他绕过去，盲人听从瘸子的指挥，两人平安到达目的地。在这组行为中，瘸子看到前面有障碍，可是瘸子不能走路；另一方面，盲人可以走路，但是他看不到，他不知道这个障碍是否存在。于是，语言沟通就充当了瘸子与盲人之间的代理，瘸子看到障碍后通过语言沟通将这个信息告诉盲人，盲人获取信息，绕过障碍。类比编程开发中，系统 UI 控件充当着故事中的瘸子的角色，

开发者充当着盲人的角色，当系统 UI 控件接收到用户的一些动作行为时，它并不知道应该怎么处理这些行为逻辑，于是通过代理方法将用户的这一动作告诉开发者，开发者在代理方法中处理相关的程序逻辑。

提　示

代理这种设计模式比较抽象，初学者可能很难理解，不过读者不用担心，在后面具体讲解控件时会使用代理方法来处理交互逻辑，到时候再结合本小节内容进行理解，就会容易很多。

2.2　视图控制器——UIViewController

UIViewController 是 UIKit 框架中控制器部分的基础，一些复杂的 Controller 类也是基于 UIViewController 继承出来的。在开发应用程序时，所有界面也都是基于 UIViewController 搭建出来的。

2.2.1　UIViewController 的生命周期

生命周期是指一个对象从创建出来到其被释放销毁的整个过程。Swift 是一种面向对象的高级语言，为了保持内存的平衡与程序运行的高效，当需要一个对象时，它会被创建并分配内存空间；同样，当它不再被需要时，也应该被系统释放回收。在一个 UIViewController 对象从创建到释放的过程中，会依次调用许多生命周期函数，了解这些函数的调用时机和功能是 iOS 开发者的必修课。打开 Xcode 开发工具，创建一个名为 UIViewControllerTest 的工程，工程创建出来后，系统的模板自动生成一个 ViewController 类，这个类继承于 UIViewController 并与 Main.storyboard 中的初始视图控制器关联。简单来说，这个类创建了一个视图控制器，作为工程的根视图控制器，我们可以在这个类中编写相关代码来对 UIViewController 的生命周期进行研究。

UIViewController 中与生命周期相关的函数有很多，列举如下。

```
//从 xib 文件进行初始化
override init(nibName nibNameOrNil: String?, bundle nibBundleOrNil: Bundle?)
//从归档进行初始化
required init?(coder aDecoder: NSCoder)
//从 xib 加载
override func awakeFromNib()
//加载视图
override func loadView()
//加载视图完成
override func viewDidLoad()
//将要布局子视图
override func viewWillLayoutSubviews()
//完成布局子视图
override func viewDidLayoutSubviews()
//收到内存警告
```

```
    override func didReceiveMemoryWarning()
    //视图将要展示
    override func viewWillAppear(_ animated: Bool)
    //视图完成展示
    override func viewDidAppear(_ animated: Bool)
    //视图将要消失
    override func viewWillDisappear(_ animated: Bool)
    //视图已经消失
    override func viewDidDisappear(_ animated: Bool)
    //析构方法
    deinit
```

实践是最好的老师，在编程的学习中亦是如此。要了解上面方法的执行次序，跟踪程序的运行是最快的方法，先在 ViewController.swift 文件中实现这些方法，代码如下。

```
import UIKit
var tip = 0
class ViewController: UIViewController {
    //从 xib 文件进行初始化
    override init(nibName nibNameOrNil: String?, bundle nibBundleOrNil: Bundle?) {
        super.init(nibName: nibNameOrNil, bundle: nibBundleOrNil)
        tip+=1
        print("init:\(tip)")
    }
    //从归档进行初始化
    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        tip+=1
        print("init?:\(tip)")
    }
    //从 xib 加载
    override func awakeFromNib() {
        super.awakeFromNib()
        tip+=1
        print("awakeFromNib:\(tip)")
    }
    //加载视图
    override func loadView() {
        super.loadView()
        tip+=1
        print("loadView:\(tip)")
    }
    //加载视图完成
    override func viewDidLoad() {
        super.viewDidLoad()
        tip+=1
        print("viewDidLoad:\(tip)")
    }
    //将要布局子视图
    override func viewWillLayoutSubviews() {
        super.viewWillLayoutSubviews()
        tip+=1
        print("viewWillLayoutSubviews:\(tip)")
    }
    //完成布局子视图
```

```
    override func viewDidLayoutSubviews() {
        super.viewDidLayoutSubviews()
        tip+=1
        print("viewDidLayoutSubviews:\(tip)")
    }
    //收到内存警告
    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        tip+=1
        print("didReceiveMemoryWarning:\(tip)")
    }
    //视图将要展示
    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        tip+=1
        print("viewWillAppear:\(tip)")
    }
    //视图完成展示
    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        tip+=1
        print("viewDidAppear:\(tip)")
    }
    //视图将要消失
    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        tip+=1
        print("viewWillDisappear:\(tip)")
    }
    //视图已经消失
    override func viewDidDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        tip+=1
        print("viewDidDisappear:\(tip)")
    }
    //析构方法
    deinit {
        tip+=1
        print("deinit:\(tip)")
    }
}
```

上面的代码中创建一个全局变量 tip，使用这个变量对程序的运行过程进行标记。ViewController 是 UIViewController 的一个子类，在子类中覆写父类的方法时，需要先调用父类的此方法。self 和 super 这两个关键字一直是初学者的噩梦，甚至在许多时候，拥有一定开发经验的开发者也不能完全理解其意义。在这里读者只需要记住，在实例方法中，self 指调用这个方法的实例，即当前类的一个实例对象，如果用 super 调用方法，就代表从父类中找这个方法实现；而在类方法或静态方法中 self 指当前类，如果用 super 调用方法，代码就从父类中找当前类方法或静态方法的实现。

提　示

deinit 方法是唯一一个不需要并且也不能在实现里调用父类方法的函数，这个函数在 ARC（自动引用计数）环境中不再被开发者需要，但是开发者依然可以重写这个函数来监测内存的释放情况。

运行工程，在 Xcode 的调试区会打印出如图 2-2 所示的信息。

```
init?:1
awakeFromNib:2
loadView:3
viewDidLoad:4
viewWillAppear:5
viewWillLayoutSubviews:6
viewDidLayoutSubviews:7
viewDidAppear:8
```

图 2-2　跟踪 Swift 程序运行的打印信息

从打印信息中可以清晰地看到 UIViewController 被创建的过程中依次调用的方法，但是这些信息并不全面，并没有体现出 UIViewController 销毁时的过程，也没有展现从不同来源初始化的 UIVewController 生命周期的不同。

init 和 initWithCoder 方法作用相似，都是对对象做初始化工作，如果从代码进行初始化，就会调用 init 方法；如果从归档文件进行初始化，就会调用 initWihCoder 方法。awakeFromNib 方法会在从 xib 或 storyboard 中加载的 UIViewController 将要被激活时调用。

loadView 方法是开始加载 UI 视图的初始方法，这个方法除非开发者手动调用，否则在 UIViewController 的生命周期中只会被调用一次。

viewDidLoad 方法在视图已经加载完成后会被调用，因为这个函数调用的时候，视图控制器的基本系统功能已经完成初始化，开发者一般会将一些 Controller 额外定义功能的初始化工作放在这个函数中。

- viewWillAppear 方法在视图即将显示的时候调用。
- viewWillLayoutSubviews 方法在视图将要布局其子视图时被调用。
- viewDidLayoutSubviews 方法在视图布局完成其子视图时被调用。
- viewDidAppear 方法在视图已经显示后被调用。

上面这些方法是 UIViewController 从创建到展现出来之间调用的生命周期函数，这些只是一半，UIViewController 被释放和销毁的过程由如下方法完成：

- viewWillDisappear 方法在视图将要消失时调用，开发者可以在其中做一些数据清理的操作。
- viewDidDisappear 方法在视图已经消失时被调用。
- deinit 方法是对象的销毁方法，在对象被释放时调用，开发者可以通过在其中打印信息的方式检查一个类是否存在内存泄露等问题。

2.2.2 UIViewController 的视图层级结构

UIViewController 自带一个 UIView 类型的 view 视图，这个 view 平铺在屏幕上，是视图控制器的根视图，在视图控制器中添加其他 UI 控件都是添加在这个 view 上。UIView 类通过 addSubview 方法添加子 view 视图，子 view 视图也可以继续使用 addSubview 方法添加自己的子视图。在第 1 章中，Hello World 标签实际上就是添加在视图控制器的根视图上的一个子 Label 标签视图，iOS 系统通过这样的层级结构管理整个 UI 体系。

2.3 文本控件——UILabel

在任意一款应用中，随处可见各种各样的文字标签，Label（标签）是应用程序 UI 体系中最基础也是最简单的一个控件。顾名思义，其主要作用是在屏幕视图上显示一行或多行文本，类似于生活中的便条标签。在 UIKit 框架中，UILabel 有很多可定制属性提供给开发者进行控件的自定义设置。

2.3.1 使用 UILabel 在屏幕上创建一个标签控件

在第 1 章中，通过 storyboard 文件很轻松地在视图上创建了显示 Hello World 的标签，这一小节我们使用代码实现同样的效果。打开 Xcode 开发工具，创建一个名为 HelloWorldText 的工程，在 ViewController 中的 viewDidLoad 方法中添加如下代码。

```
override func viewDidLoad() {
    super.viewDidLoad()
    let label = UILabel(frame: CGRect(x: 20, y: 100, width: 280, height: 30))
    label.text = "Hello World"
    self.view.addSubview(label)
}
```

在上面的代码中，UILabel（frame:）构造方法是 UILabel 类中的一个初始化方法，该初始化方法中需要传入一个 CGRect 类型的结构体，这个结构体参数决定了在屏幕上创建 UILabel 控件的位置和尺寸，CGRect 确定在 iOS 的 UI 系统中绘制的一个矩形区域，其中的 4 个参数依次设置这个矩形区域的 x 坐标、y 坐标、宽度和高度。

提　示

在 UIKit 框架中，UI 坐标系与生活中的数学坐标系略有不同，数学坐标系中横向为 x 轴，向右增大，纵向为 y 轴，向上增大，在 UI 坐标系中，横向为 x 轴，向右增大，纵向为 y 轴，向下增大，即原点在左上角。

UILabel 类的 text 属性用于设置标签上的文字，必须设置为一个 String 类型的字符串值。在调

用 UIView 类的 addSubview 方法后，将 UILabel 控件添加在当前的视图上，运行工程会看到如图 2-3 所示的效果。

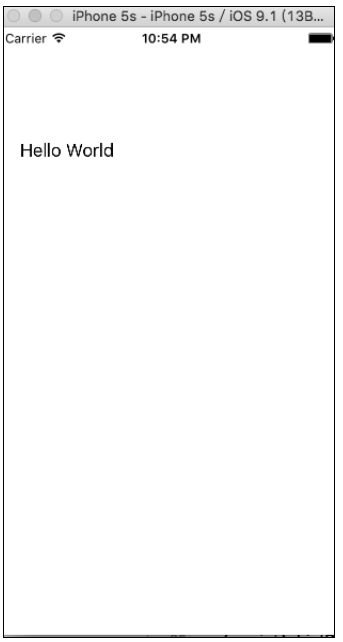

图 2-3　代码创建的 UILabel 控件

2.3.2　自定义标签控件的相关属性

在上面创建的 HelloWorldText 工程中，标签控件的样式是系统默认的，有时开发者需要更多元化的标签（如各种颜色、字体等），UILabel 中有大量的属性提供给开发者进行自定义，代码示例如下。

```
        override func viewDidLoad() {
            super.viewDidLoad()
            let label = UILabel(frame: CGRect(x: 20, y: 100, width:
280, height: 30))
            label.text = "Hello World"
            //设置背景颜色
            label.backgroundColor = UIColor.red
            //设置字体和字号
            label.font = UIFont.systemFont(ofSize: 23)
            //设置字体颜色
            label.textColor = UIColor.white
            //设置对齐模式
            label.textAlignment = .center
            //设置引用颜色
            label.shadowColor = UIColor.green
            //设置阴影偏移量
            label.shadowOffset = CGSize(width: 10, height: 10)
            //设置断行模式
            label.lineBreakMode = .byWordWrapping
            self.view.addSubview(label)
        }
```

backgroundColor 属性设置了标签的背景颜色，实际上 backgroundColor 并非是 UILabel 特有的属性，很多继承 UIView 的子类都有这个属性。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 控件上文字的对齐模式，默认是居中对齐。下面列举设置对齐模式使用的枚举值。

```
    public enum NSTextAlignment : Int {
        case left //居左对齐
        case center // 居中对齐
        case right // 居右对齐
    }
```

图 2-4　自定义属性的 UILabel

shadowColor 属性设置文字的阴影颜色，shadowOffset 属性设置阴影的偏移量，即阴影与本体之间的偏移距离，这个属性要设置一个 CGSize 类型的结构体，CGSize 中的两个参数分别代表横向偏移量和纵向偏移量。通过添加上面的代码，再次运行工程，效果如图 2-4 所示。

2.3.3 多行显示的 UILabel 控件与换行模式

通过 UILabel（frame:）初始化方法创建的 UILabel 控件会有一个宽度，如果文字长度超过了 UILabel 控件的宽度，默认的 UILabel 控件并不会换行，而是用省略号代替超出的部分。例如，将 UILabel 控件高度和 text 属性改如下：

```
let label = UILabel(frame: CGRect(x: 20, y: 100, width: 280, height: 30))
label.text = "Hello World,It is a good idea, So,what do you want to konw?"
```

运行工程，会看到多出的文字被截断了，UILabel 并没有换行，如图 2-5 所示。

图 2-5 文字被截断的 UILabel 控件

默认的 UILabel 都是单行显示的，可以通过如下属性设置显示的行数：

```
label.numberOfLines = 0;
```

将 numberOfLines 设置为一个整数值，代表支持多少行显示，如果设置为 0，就代表无限换行，直到文字结束或到达 UILabel 控件的最底端为止。

UILabel 中还有一个 lineBreakMode 属性，这个属性可以设置换行和截断模式，这个属性设置的值为 NSLineBreakMode 枚举，意义如下。

```
public enum NSLineBreakMode : Int {
    case byWordWrapping // 以单词为标准进行换行
    case byCharWrapping // 以字符为标准进行换行

    case byTruncatingHead // 头部截断
    case byTruncatingTail // 尾部截断
    case byTruncatingMiddle // 中间截断
}
```

上面这个枚举值的设置效果如图 2-6~图 2-10 所示。

图 2-6 NSLineBreakByWordWrapping

图 2-7 NSLineBreadByCharWrapping

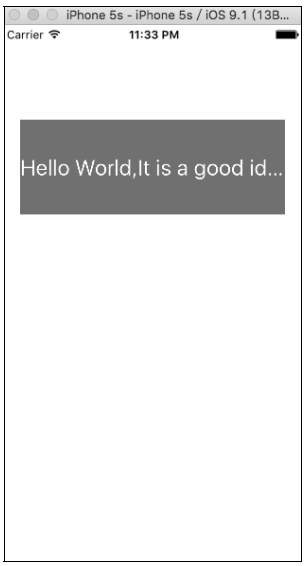

图2-8　NSLineBreakByTruncatingHead　图2-9　NSLineBreakByTruncatingMiddle　图2-10　NSLineBreakByTruncatingTail

2.4　按钮控件——UIButton

对于一个应用类软件来说，展示和交互就是生命。UILabel 控件可以说是 UIKit 框架中最简单基础的显示控件，与之对应的 UIButton 控件是 UIKit 框架中最简单基础的交互控件。UIButton 通常又被称为按钮控件，它的确也发挥着按钮的功能，可以监听用户在屏幕视图上的多种手势操作。

2.4.1　创建一个按钮改变屏幕颜色

通过 Xcode 开发工具创建一个名为 UIButtonTest 的工程，在 ViewController.swift 的 viewDidLoad 方法中添加如下代码。

```
    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("点我一下", for: .normal)
        button.addTarget(self, action: #selector(changeColor), for: .touchUpInside)
        self.view.addSubview(button)
    }
```

一般通过 UIButton 控件的类方法 buttonWithType 进行按钮控件的初始化，这里传入一个 UIButtonType 类型的枚举参数来确定创建的 UIButton 控件的风格，可用枚举值及意义如下。

```
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
    }
```

不同的风格枚举创建出来的按钮样式如图 2-11~图 2-14 所示。

图 2-11　UIButtonTypeCustom

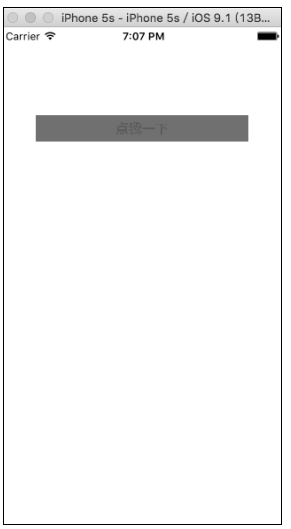

图 2-12　UIButtonTypeSystem

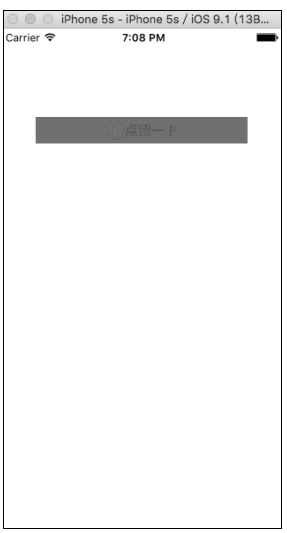

图 2-13　UIButtonTypeDetailDisclosure

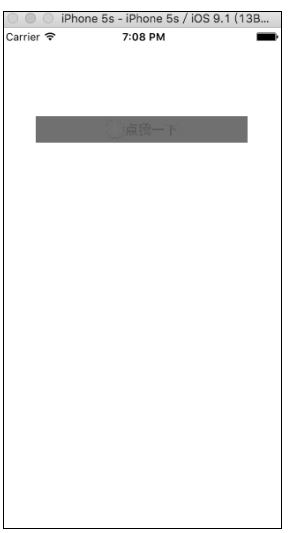

图 2-14　UIButtonTypeContactAdd

UIButton 控件包括背景色、字体颜色、字体、单击状态等属性。Custom 风格是将这些属性全部采用默认值，需要开发者自行进行设置；System 风格是系统定义好的一组属性设置的风格；DetailDisclosure 风格会在左边显示一个详情小图标；ContactAdd 风格是在按钮左边显示一个添加小图标。

回到上面的代码，setTitle 方法有两个参数，第一个设置按钮的标题文字，第二个设置显示此标题文字时的按钮状态。UIControlState 中定义了许多用于交互控件的状态定义，常用枚举值如下。

```
    public struct UIControlState : OptionSet {
```

```
        public static var normal: UIControlState { get }  //正常状态
        public static var highlighted: UIControlState { get } //高亮状态
        public static var disabled: UIControlState { get }  //不可用状态
        public static var selected: UIControlState { get } // 选中状态
    }
```

在上面的枚举中，Normal 是按钮的初始状态，此时并没有进行任何交互操作；Highlighted 为高亮状态，即当用户手指单击到按钮但并没有抬起时的状态；Disabled 为不可用状态，此时用户的单击操作将无效；Selected 为选中状态，用于一些充当切换开关的按钮。

UIButton 控件的核心功能是进行用户交互，通过 addTarget 方法进行触发方法的添加。这个方法需要 3 个参数：第 1 个参数为执行此触发方法的对象，一般会填写当前视图控制器类对象本身 self；第 2 个参数为对应的方法；第 3 个参数为触发方法的条件。这里需要传入一个 UIControlEvents 类型数据，其中常用值的意义如下。

```
    public struct UIControlEvents : OptionSet {
        public static var touchDown: UIControlEvents { get } //用户手指按下时触发方法
        public static var touchDownRepeat: UIControlEvents { get } // 用户多次重复按下时触发
        public static var touchDragInside: UIControlEvents { get }//用户在控件范围内拖滑移动时触发
        public static var touchDragOutside: UIControlEvents { get }//用户在控件范围内按下，并拖拽到
控件范围外抬起时触发
        public static var touchDragEnter: UIControlEvents { get }//用户手指拖滑进控件范围内触发
        public static var touchDragExit: UIControlEvents { get }//用户手指拖滑结束时触发
        public static var touchUpInside: UIControlEvents { get }//用户在控件范围内按下并在控件范围内
抬起时触发，即单击
        public static var touchUpOutside: UIControlEvents { get }//用户在控件范围内按下并在范围外抬
起触发
        public static var touchCancel: UIControlEvents { get }//触摸事件被取消时触发
        public static var valueChanged: UIControlEvents { get } // 控件的 value 值改变时触发
    }
```

上面定义的选项决定了触发交互的用户操作行为。

下面是实现 changeColor 触发方法的示例。

```
        func changeColor()  {
           self.view.backgroundColor = UIColor(red: CGFloat(arc4random()%255)/255.0, green:
CGFloat(arc4random()%255)/255.0, blue: CGFloat(arc4random()%255)/255.0, alpha: 1)
        }
```

changeColor 方法中使用通过 RGBA 方式来创建颜色对象，前 3 个参数分别设置了颜色红、绿、蓝 3 色值，第 4 个参数设置了颜色的透明度值，每个参数的取值均为 0~1 的浮点值。

运行工程，单击视图上的按钮，可以看到视图颜色随机切换的霓虹效果。

2.4.2　更加多彩的 UIButton 控件

2.4.1 节的例子只是创建了一个未加任何修饰的按钮控件，可以通过 UIButton 的一些属性为其添加背景或内容图片来创建更多彩的按钮控件。

首先，向工程项目中添加一张图片，在工程的导航窗口部分中单击 Assets.xcassets 包文件，选择一张图片并将其拖入素材区，如图 2-15 所示。

图 2-15　向工程中添加图片素材

通过下面的代码来对按钮控件进行相关设置。

```
        let button = UIButton(type: .custom)
        button.setBackgroundImage(UIImage(named: "image"), for: .normal)
```

运行工程，会看到如图 2-16 所示的效果，按钮被添加了图片背景。

可以发现，背景图片的效果是当图片作为背景时，按钮标题显示在图片层之上。UIButon 中还有一个方法用于设置图片为内容图片，这种情况下图片将和标题并列显示，代码如下。

```
    button .setImage(UIImage(named: "image"), for: .normal)
```

效果如图 2-17 所示。

图 2-16　按钮添加背景图片

图 2-17　按钮添加内容图片

在图 2-17 中可以看到，图片和文字是左右并排排列且共同居中显示的。在很多时候，开发者会需要上下排列或以其他形式进行图片和文字的排列，UIButton 类中也提供了接口供开发者进行内容、图片和文字的位置设置，示例代码如下。

```
button.contentEdgeInsets = UIEdgeInsetsMake(0, 0, 0, 0)
button.imageEdgeInsets = UIEdgeInsetsMake(0, 0, 0, 0)
button.titleEdgeInsets = UIEdgeInsetsMake(0, 0, 0, 0)
```

setContentEdgeInsets 方法用于设置整体内容的区域偏移量，UIEdgeInsetsMake()方法中的 4 个参数分别代表上、左、下、右 4 个方向的偏移量，读者可以对这 4 个值进行修改并观察效果。setImageEdgeInsets 方法只设置内容图片的位置偏移量，setTitleEdgeInsets 方法只设置内容标题的位置偏移量。

2.5　文本输入框控件——UITextField

相比于 UILabel 和 UIButton 控件，UITextField 控件要复杂得多。UITextField 是 iOS 系统中进行文本输入操作的一种 UI 控件，用户通过键盘将输入操作传递给 UITextField 控件，UITextField 控件采用代理的设计模式再将用户的一些操作行为以回调方式传递给开发者，最后由开发者进行具体的逻辑处理。

2.5.1　在屏幕上创建一个输入框

打开 Xcode 开发工具，创建一个名为 UITextFIeldTest 的工程，在 ViewController 类的 viewDidLoad 方法中添加如下代码。

```
override func viewDidLoad() {
    super.viewDidLoad()
    let textField = UITextField(frame: CGRect(x: 20, y: 100, width: 280, height: 30))
    textField.borderStyle = .roundedRect
    textField.placeholder = "请输入文字"
    self.view.addSubview(textField)
}
```

上面的代码创建了一个输入框，UITextField 的 placeholder 属性用于设置提示文字，这些文字在输入框中有输入时会自动隐藏，在输入框输入的内容为空时才会显示出来，其作用主要是提示用户此输入框的作用，如登录界面的用户名输入框通常会这样提示：“请输入您的用户名”。运行工程，会看到如图 2-18 和图 2-19 所示的效果。

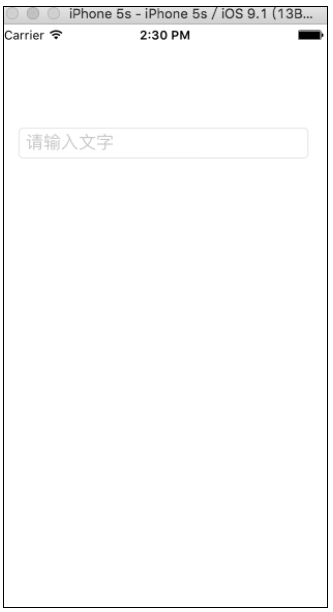

图 2-18　未输入文字的输入框

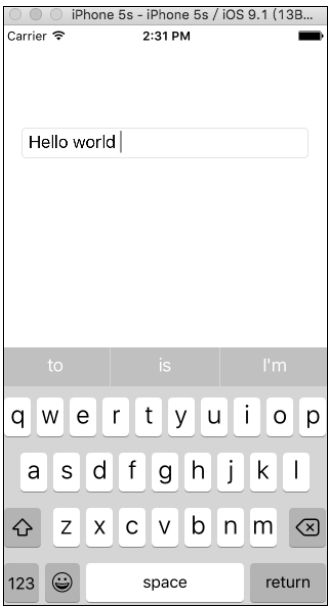

图 2-19　正在输入的输入框

UITextField 的 borderStyle 属性用于设置输入框的界面风格，UITextBorderStyle 枚举值的意义如下。

```
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
}
```

边框风格的界面效果如图 2-18 所示，其他风格的输入框界面效果如图 2-20~图 2-22 所示。

图 2-20　UITextBorderStyleNone

图 2-21　UITextBorderStyleLine

图 2-22　UITextBorderStyleBezel

2.5.2　UITextField 的常用属性介绍

UITextField 中也有相关属性用于输入框中文字属性的设置，代码示例如下。

```
        textField.textColor = UIColor.red
        textField.font = UIFont.systemFont(ofSize: 14)
        textField.textAlignment = .center
```

开发者除了对输入框中文字属性进行设置外，UITextField 还支持自定义左视图和右视图。UITextField 的左视图应用十分广泛，例如很多密码输入框的左边都会有一个小钥匙的图片，用来提示用户输入框的作用。设置左视图的代码示例如下。

```
        let imageView = UIImageView(image: UIImage(named:
"image"))
        textField.leftView = imageView
        textField.leftViewMode = .always
```

UITextField 的 leftView 属性需要传入一个 UIView 或其子类的对象，示例代码中使用了 UIImageView，leftViewMode 属性设置了显示左视图的显示模式，枚举意义如下。

```
public enum UITextFieldViewMode : Int {
    case never         //从不显示
    case whileEditing    //编辑时显示
    case unlessEditing   //非编辑时显示
    case always        //总是显示
}
```

图 2-23　显示左视图的 UITextField

此时运行工程，可以看到如图 2-23 所示的界面效果。

提　示

UIImageView 是专门用来展示图片的视图类。这个类不仅可以通过设置 image 属性进行图片的渲染展示，还可以通过一些设置实现帧动画的播放。

2.5.3　UITextField 的代理方法

在本章的开头部分介绍了代理这种设计模式，UITextField 的一些回调就是通过代理方法来实现的。例如，很多网站的会员账号是采用手机号码来注册的，这时对于用户名输入框来说，只能允许用户输入不超过 11 位的数字，如果用户输入非数字字符或输入超限，应用会进行处理，使用户的这次输入无效。其实这个过程就是一个代理回调处理逻辑的过程，首先用户输入一个字符，字符被传进 UITextField 中，UITextField 无法判断这个字符是否有效，它将字符通过代理方法再传递给开发者，开发者来做逻辑处理。UITextFieldDelegate 中支持以下代理方法。

```
    //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(_ textField: UITextField) -> Bool
```

textFieldShouldBeginEditing 方法是当用户在屏幕上单击输入框，键盘将要弹出来时会调用。这个函数有一个 BOOL 类型的返回值，如果开发者在实现这个函数时返回 NO，键盘就不会弹出来，UITextFiled 控件也不能进入编辑状态。textFiedShouldEndEditing 方法与 textFieldShouldBeginEditing 方法类似，只是它对应的是结束编辑状态。textFieldDidBeginEditing 方法和 textFieldDidEndEditing 方法分别是在 UITextField 已经开始和已经结束编辑状态时触发的方法。shouldChangeCharactersInRange: replacementString 方法在输入框中内容将要改变时会调用，这时会传进两个参数给开发者使用，range 是将要改变的字符范围，string 是将要替换成的字符串，同时这个函数还需要返回一个 BOOL 类型的返回值，如果返回 NO，这次字符改变行为就不成功，判断用户的输入是否合法的操作，一般会放在这个代理方法中进行。textFieldShouldClear 方法在单击清除按钮后会被调用，这里的返回值如果返回 NO，这次清除操作就会无效。textFieldShouldReturn 方法在用户按键盘上的 return 键后进行调用。

提　示

所谓 UITextField 的编辑状态，就是光标出现在输入框中并且闪烁，键盘弹出等待用户输入的状态。

2.5.4　实现一个监听输入信息的用户名输入框

上面介绍了 UITextFieldDelegate 中的相关方法，有了这些知识，已经可以实现一个实时监听的输入框了。使用代理方法需要以下 3 步：

（1）遵守相应协议。
（2）设置代理。
（3）实现代理方法。

首先，在类的声明部分添加要遵守的代理，如下所示。

```
    class ViewController: UIViewController,UITextFieldDelegate
```

在 viewDidLoad 方法中添加如下一行代码进行代理的设置。

```
    textField.delegate = self;
```

在 ViewController 类中实现如下代理方法。

```
        func textField(_ textField: UITextField, shouldChangeCharactersIn range: NSRange,
replacementString string: String) -> Bool {
            if string.characters.count>0 {
                let charas:[Character] = ["0","1","2","3","4","5","6","7","8","9"]
                let char = string.characters.first!
                if !charas.contains(char) {
                    print("请输入数字")
                    return false
                }
                if textField.text!.characters.count+string.characters.count>11 {
                    print("超过 11 位啦")
                    return false
                }
            }
            return true
        }
```

在实现的 textField:shouldChangeCharactersInRange:replacementString 方法中先进行了是否是数字的逻辑判断。如果不是数字，就会打印提示信息，并且使本次输入无效，之后判断数字是否超过 11 位，这个判断条件中取的字符长度是输入框上原有文字的长度加上本次输入字符的长度，如果超过 11 位，就打印信息并使本次输入无效。这样，一个只能输入数字且不可超过 11 位的输入框就编写完成了，如图 2-24 所示。

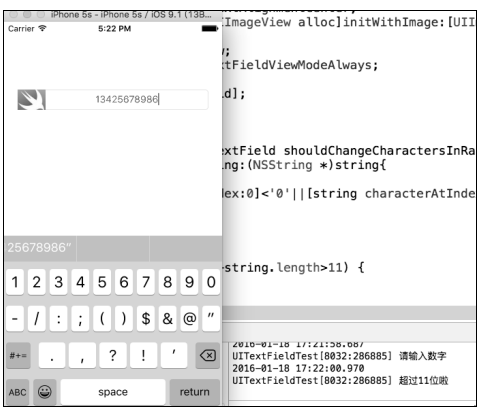

图 2-24　监听用户输入操作的 UITextField

2.6　开关控件——UISwitch

UISwitch 是 UIKit 框架中的一个十分小巧简洁的控件，用于一些简单的切换功能逻辑中，在很多 Apple 自行开发的应用中，这个控件的使用率也非常高。

2.6.1　创建一个开关控件

使用 Xcode 开发工具创建一个名为 UISwitchTest 的工程，在 ViewController 类的 viewDidLoad

方法中添加如下代码。

```
override func viewDidLoad() {
    super.viewDidLoad()
    let swi = UISwitch(frame: CGRect(x: 100, y: 100, width: 100, height: 40))
    swi.onTintColor = UIColor.green
    swi.tintColor = UIColor.red
    swi.thumbTintColor = UIColor.orange
    self.view.addSubview(swi)
}
```

由于 UISwitch 的功能十分简单，因此可设置的属性十分有限。onTintColor 属性用于设置控件开启状态的填充色；tintColor 属性设置控件关闭状态的边界色；thumbTintColor 属性设置开关按钮的颜色。运行工程，效果如图 2-25 和图 2-26 所示。

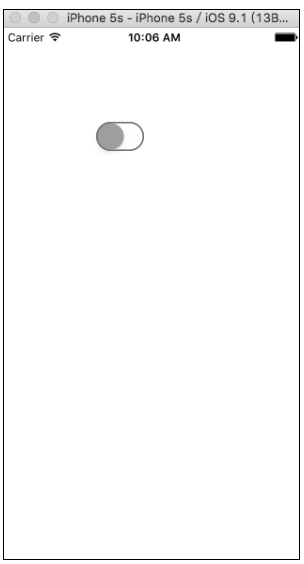

图 2-25　关闭状态的 UISwitch 控件

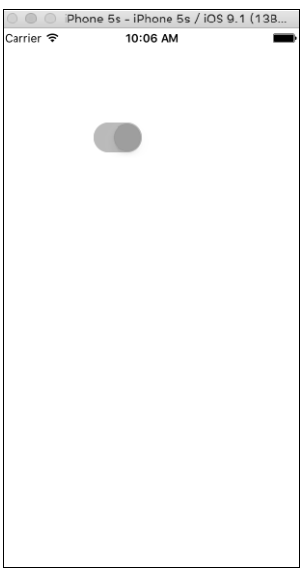

图 2-26　开启状态的 UISwitch 控件

2.6.2　为 UISiwtch 控件添加触发方法

UISwitch 也属于用户交互控件，可以为其添加交互方法来处理某些开与关的逻辑。UISwitch 继承于 UIControl，继承于 UIControl 的类都可以通过 addTarget 方法来进行触发方法的添加，代码如下。

```
swi.addTarget(self, action: #selector(changeColor), for: .valueChanged)
```

这里实现的触发方法使用带一个参数值的函数，系统传入的参数为 UISwitch 对象本身，方法实现代码如下。

```
func changeColor(swi:UISwitch) {
    if (swi.isOn) {
        self.view.backgroundColor = UIColor.red;
    }else{
        self.view.backgroundColor = UIColor.white;
```

```
        }
    }
```

UISwitch 的 isOn 属性是一个布尔值，通过这个值可以判断 UISwitch 控件的开关状态，然后分别进行相应的操作即可，这里在切换 UISwitch 控件的开关状态时进行了当前视图背景颜色的转换。

2.7　分页控制器——UIPageControl

分页视图是一种十分流行的界面设计模式，使用的场景也非常多，如新手引导页和广告轮播页等。UIPageControl 是用于页码管理的一个 UI 控件，其表现形式是一行圆点，其中高亮的一个圆点标记当前所在的页码。使用 Xcode 开发工具创建一个名为 UIPageControlTest 的工程，在 ViewController 类的 viewDidLoad 方法中添加如下代码。

```
    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = UIColor.black
        let page = UIPageControl(frame: CGRect(x: 20, y: 100,
width: 280, height: 30))
        page.currentPageIndicatorTintColor = UIColor.red
        page.addTarget(self, action: #selector(changeNum),
for: .valueChanged)
        page.numberOfPages = 8
        self.view.addSubview(page)
    }
```

将当前视图的背景颜色设置为黑色便于进行效果的演示，UIPageControl 的 currentPageIndicatorTintColor 属性用于设置高亮页码点的颜色，numberOfPages 属性用于设置页码数量。运行工程，效果如图 2-27 所示。

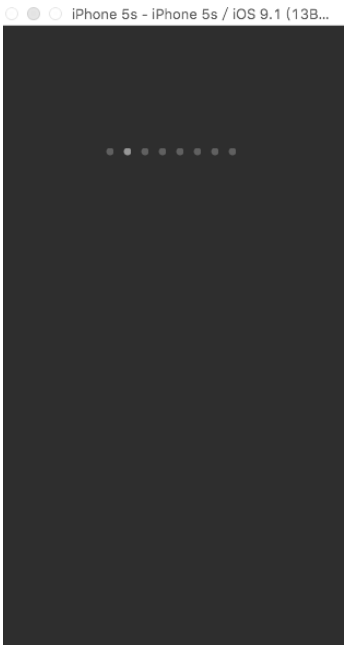

图 2-27　UIPageControl

在单击 UIPageControl 控件的左半边时，页码点会向左移动，单击 UIPageControl 的右半边时，页码点会向右移动。在单击 UIPageControl 控件的同时也会触发交互方法，交互方法的实现如下所示，这里打印了当前的页码数（从 0 开始）。

```
    func changeNum(page:UIPageControl)  {
        print(page.currentPage)
    }
```

2.8　分段控制器——UISegmentedControl

UISegmentedControl 用于管理和实现一组内容的切换逻辑，如几个并列关系的界面之间相互切换。UISegmentedControl 常见于导航栏的标题视图中，因其小巧的外表和简洁的接口风格，在原生

和第三方应用中都十分常见。

2.8.1 UISegmentedControl 基本属性的应用

使用 Xcode 开发工具创建一个名为 UISegmentedControlTest 的工程，在 ViewController 类的 viewDidLoad 方法中添加如下代码。

```
        override func viewDidLoad() {
            super.viewDidLoad()
            let seg = UISegmentedControl(items:
["one","","three","four"])
            seg.frame = CGRect(x: 20, y: 100, width: 280, height: 30)
            seg.setImage(UIImage(named:
"image")?.withRenderingMode(.alwaysOriginal), forSegmentAt: 1)
            seg.setContentOffset(CGSize(width: 10, height: 10),
forSegmentAt: 1)
            seg.isMomentary = true
            self.view.addSubview(seg)
        }
```

代码中使用带 item 数组参数的构造方法来初始化 UISegmentedControl 对象，这个方法中需要传入一个标题数组，数组中字符串的个数和内容决定了 UISegmentedControl 控件中按钮的个数和标题。setImage 方法用于设置某个按钮的图案，其中按钮的编号 Index 从 0 开始计。setContentOffset 方法设置其中某个按钮内容的位置偏移。UISegmentedControl 的 isMomentary 属性默认为布尔值假，控件为切换按钮模式，如果设置为布尔值真，控件就为触发按钮模式。运行工程，效果如图 2-28 所示。

图 2-28 UISegmentedControl

提 示

UISegmenedControl 在切换按钮模式时，当用户点击一个按钮后，此按钮会一直保持选中状态直到用户切换为另一个按钮为止，而在触发按钮模式中，用户手指离开屏幕后，按钮将不会继续保持选中状态。

2.8.2 对 UISegmentedControl 中的按钮进行增、删、改操作

UIsegmentedControl 对象中的按钮除了在初始化时可以进行创建外，在程序运行过程中，也可以进行动态的添加、删除和修改操作。UISegmentedControl 中有如下方法可供开发者使用。

```
            seg.insertSegment(withTitle: "new", at: 2, animated: true)
            seg.removeSegment(at: 1, animated: true)
            seg.setTitle("replace", forSegmentAt: 1)
            seg.removeAllSegments()
```

insertSegment 方法用于在 UISegmentedControl 当前按钮组中插入一个新的标题按钮，第 1 个参数是设置按钮的标题文字，第 2 个参数是设置插入的位置，第 3 个参数是设置是否带动画效果，

与这个方法对应的还有一个插入方法用于插入一个图片按钮；removeSegment 方法可以在已有的按钮中移除一个；setTitle 方法可以重新设置一个按钮的标题，与之对应的 setImage 方法可以重新设置一个按钮的图片；removeAllSegments 方法将移除所有的按钮。

2.8.3　UISegmentedControl 中按钮宽度的自适应

UISegmentedControl 控件中的按钮宽度默认是平均分的，如果某个按钮的标题长度超出了宽度的范围，将会被自动截断，如图 2-29 所示。

开发者可以手动对 UISegmentedControl 中的每个按钮的宽度进行设置，以便设置按钮宽度与其文字相适应，示例代码如下，效果如图 2-30 所示。

```
seg.setWidth(130, forSegmentAt: 3)
```

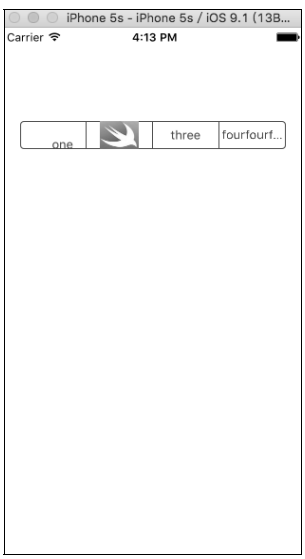

图 2-29　UISegmentedControl 按钮标题文字被截断

图 2-30　自定义 UISegmentedContrl 按钮宽度

上面的方法可以对 UISegmentedControl 中按钮宽度进行设置，但是有一个致命的缺点，开发者可能并不知道这个按钮上面标题文字所占的宽度，如果使用强制计算的方法，就会徒增许多代码量。幸运的是，UISegmentedControl 中还提供了一个属性，可以让 UISegmentedControl 自己计算其中按钮需要的宽度，让其进行宽度的自适应，代码如下。

```
seg.apportionsSegmentWidthsByContent = true
```

将 apportionsSegmentWidthsByContent 属性设置为布尔值真，UISegmentedControl 控件中的按钮宽度将进行自适应。

提　示

UISegmentedControl 添加触发方法也是通过 addTarget 方法来设置的，需要监听的触发动作和 UIPageControl 控件一致，为 valueChanged。

2.9 滑块控件——UISlider

前几节所介绍的控件都有一个共同的特点，即状态的变化或值的变化都是离散的，如 UIButton 的正常、高亮、选中，UISwitch 的开和关，UISegmentedControl 按钮值的切换等。UISlider 控件与上述控件最大的区别在于其值的变化可以是连续的，由于这种特性，UISlider 控件可以处理一些连续变化量的交互逻辑。

2.9.1 UISlider 的创建与常规设置

使用 Xcode 开发工具创建一个名为 UISliderTest 的工程，在 ViewController 类的 viewDidLoad 方法中添加如下代码。

```
override func viewDidLoad() {
    super.viewDidLoad()
    let slider = UISlider(frame: CGRect(x: 20, y: 100, width: 280, height: 30))
    slider.isContinuous = true
    slider.minimumValue = 0
    slider.maximumValue = 10
    slider.minimumTrackTintColor = UIColor.red
    slider.maximumTrackTintColor = UIColor.green
    slider.thumbTintColor = UIColor.blue
    slider.addTarget(self, action: #selector(changeValue), for: .valueChanged)
    self.view.addSubview(slider)
}
```

continuous 属性用于设置 UISlider 控件的触发方法是否连续触发，设置为布尔值真，用户在滑动滑块时，触发方法就会多次执行；如果设置为布尔值假，只有当用户滑动操作结束时，方法才会触发。minimunValue 设置 UISlider 控件的最小值，即滑块在最左端时控件的值；maximumValue 设置 UISlider 控件的最大值，即滑块在最右端时控件的值；minimumTrackTintColor 属性设置滑块在左中轴的颜色。maximumTrackColor 设置滑块在右中轴的颜色；thumTintColor 设置滑块本身的颜色。运行工程，效果如图 2-31 所示。

使用 addTarget:action:forControlEvents:进行触发方法的添加，在 UISlider 类型的参数中可以获取控件的当前值并进行逻辑处理，示例如下。

```
func changeValue(slider:UISlider) {
    print(slider.value)
}
```

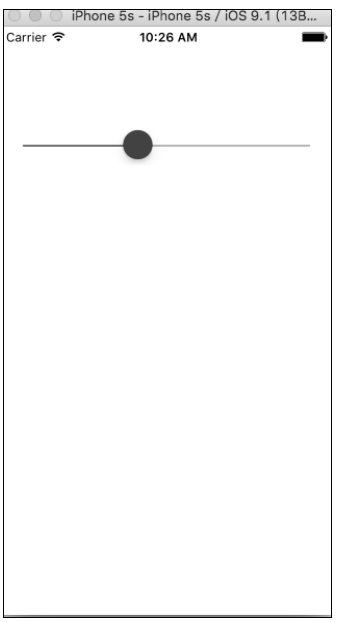

图 2-31 UISlider 控件

2.9.2　对 UISlider 添加图片修饰

UISlider 类提供了一些方法对 UISlider 控件进行图片修饰，使用如下代码。

```
            slider.minimumValueImage = UIImage(named:
"image")
            slider.maximumValueImage = UIImage(named:"
image")
            slider.setThumbImage(UIImage(named: "image"),
for: .normal)
```

minimumValueImage 属性设置左视图图片，maximumValueImage 属性设置右视图图片，setThumbImage:forState:方法设置控件在某个状态下的滑块图片。这时运行工程，效果如图 2-32 所示。

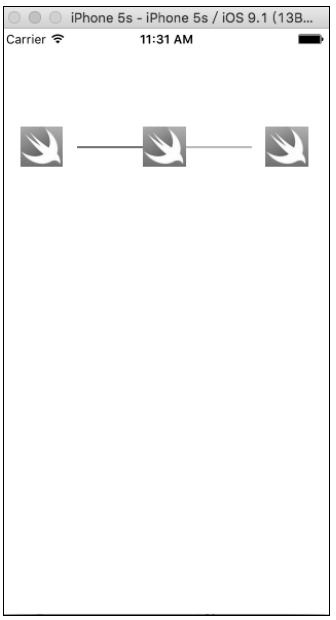

图 2-32　添加了图片修饰的 UISlider 控件

2.10　活动指示器控件——UIActivityIndicatorView

UIActivityIndicatorView 控件通常被称为风火轮控件，在某些加载复杂数据视图或下载数据的场景中经常可以看到它的身影。UIActivityIndicatorView 控件中主要作用是在加载等待的时间中给用户一些界面活动的提示，不至于使用户感觉到界面卡死的假象。

使用 Xcode 开发工具创建一个名为 UIActivityIndicatorViewTest 的工程，在 ViewController 类的 viewDidLoad 方法中添加如下代码。

```
        override func viewDidLoad() {
            super.viewDidLoad()
            self.view.backgroundColor = UIColor.red
            let indicator = UIActivityIndicatorView(activityIndicatorStyle: .gray)
            indicator.center = CGPoint(x: self.view.frame.size.width/2, y:
self.view.frame.size.height/2)
            indicator.color = UIColor.black
            self.view.addSubview(indicator)
            indicator.startAnimating()
        }
```

UIActivityIndicatorView(activityIndicatorStyle: .gray)方法通过一个风格枚举来对控件进行初始化。UIActivityIndicatorViewStyle 中枚举的值意义如下。

```
    public enum UIActivityIndicatorViewStyle : Int {
        case whiteLarge    //大尺寸白色风格
        case white         //白色风格
        case gray          //灰色风格
    }
```

其中，各个风格的效果如图 2-33~图 2-35 所示。UIActivityIndicatorView 的 color 属性可以设置活动指示器的颜色。

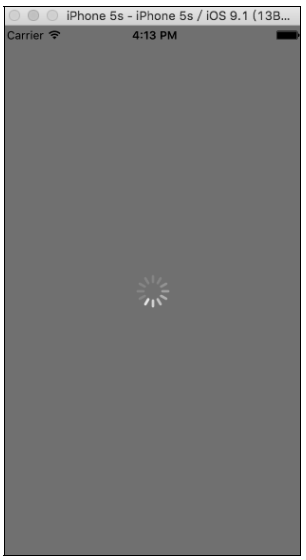

图 2-33 UIActivityIndicatorViewStyleWhiteLarge

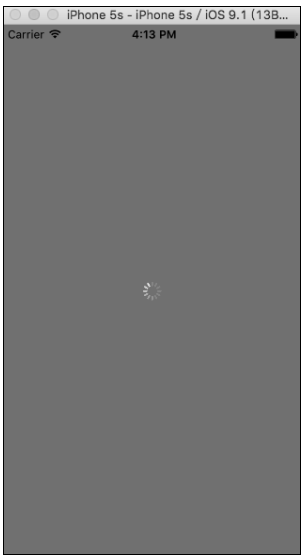

图 2-34 UIActivityIndicatorViewStyleWhite

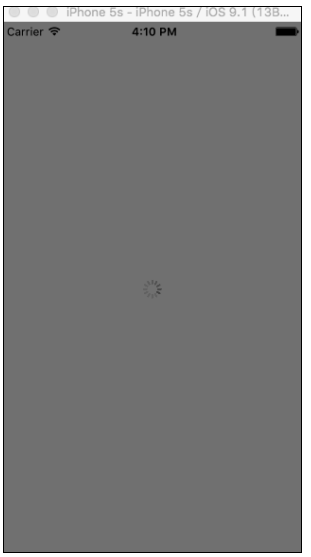

图 2-35 UIActivityIndicatorViewStyleGray

将活动指示器控件添加到视图上之后，需要调用 startAnimating 方法使指示器开始转动，与其对应，调用 stopAnimating 方法使指示器停止转动。

2.11 进度条控件——UIProgressView

UIKit 框架中的 UIProgressView 控件可以创建一个进度条，这个控件在播放器类软件中较为常见，使用 Xcode 开发工具创建一个名为 UIProgressViewTest 的工程，在 ViewController 类的 viewDidLoad 方法中添加如下代码。

```
        override func viewDidLoad() {
            super.viewDidLoad()
            let progressView = UIProgressView(frame: CGRect(x:
20, y: 100, width: 280, height: 30))
            progressView.progressTintColor = UIColor.red
            progressView.trackTintColor = UIColor.blue
            self.view.addSubview(progressView)
            progressView.progress = 0.5
        }
```

progressTintColor 属性设置已走过进度的颜色；trackTintColor 属性设置未走过进度的颜色；progress 属设置进度条当前的进度，取值为 0~1 的浮点数。运行上面的代码后，效果如图 2-36 所示。

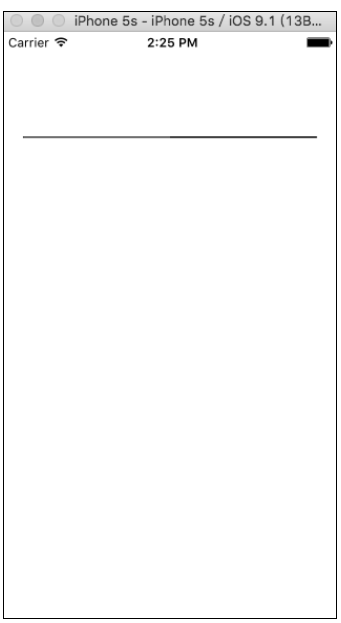

图 2-36 UIProgressView 控件

2.12　步进控制器——UIStepper

步进控制器从名字上大致可以了解其功能，是进行离散式数据调节的常用视图控件。

2.12.1　步进控制器的基本属性使用

使用 Xcode 开发工具创建一个名为 UIStepperTest 的工程，在 ViewController 类的 viewDodLoad 方法中添加如下代码。

```
override func viewDidLoad() {
    super.viewDidLoad()
    let stepper = UIStepper()
    stepper.center = CGPoint(x: 100, y: 100)
    stepper.isContinuous = true
    stepper.autorepeat = true
    stepper.wraps = true
    stepper.minimumValue = 1
    stepper.maximumValue = 10
    stepper.stepValue = 1
    stepper.tintColor = UIColor.red
    self.view.addSubview(stepper)
    stepper.addTarget(self, action: #selector(click), for: .valueChanged)
}
```

UIKit 框架中所有的视图控件都有 center 这个属性，center 属性用于设置控件的中心位置坐标，上面的代码将步进控制器位置设置在坐标为（100,100）的位置。continuous 属性设置触发方法是否连续执行，当用户按住步进控制器上的某个按钮不放时，如果 continuous 属性设置为布尔值真，添加的触发方法就会一直连续执行，步进控制器的值每变化一次方法就会执行一次。autorepeat 属性从字面理解为自动重复，当 autorepeat 属性设置为布尔值真时，用户如果按住步进控制器中的按钮不放，步进控制器的值就会一直连续改变，如果 autorepeat 设置为布尔值假，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，触发方法才会执行。warps 属性设置步进控制器的值是否循环，如果设置为布尔值真，当值增加到最大时，用户继续单击增加按钮，值就会从最小值重新开始增加，反之亦然，如果这个属性设置为布尔值假，当步进控制器到达极值时，相应的按钮将会被禁用。minimumValue 属性设置步进控制器的最小值，maximumValue 属性设置步进控制器的最大值。stepValue 属性用于设置步进控制器的步长，即每次按下按钮后步进控制器的值改变的大小。tintColor 属性设置控件的颜色。步进控制器也是通过 addTarget 方法来添加触发事件的，在触发方法中将会传入 UIStepper 对象本身，开发者通过获取其值来做出相应的逻辑处理。上面代码中的 click 方法实现如下，这里打印了 UIStepper 控件的值。

```
func click(stepper:UIStepper)  {
    print(stepper.value)
}
```

运行上面的程序代码，会看到如图 2-37 所示的效果。

2.12.2 自定义 UIStepper 按钮图片

在图 2-37 中可以看到，系统的 UIStepper 控件默认是显示一个减号和一个加号，单击加号值增加，单击减号值减小，开发者也可以通过以下方法自定义两个按钮的图片。

```
            stepper.setDecrementImage(UIImage(named:
"image")?.withRenderingMode(.alwaysOriginal), for: .normal)
            stepper.setIncrementImage(UIImage(named:
"image")?.withRenderingMode(.alwaysOriginal), for: .normal)
```

setDecrementImage 方法设置减按钮的图片，setIncrementImage 方法设置加按钮的图片，效果如图 2-38 所示。

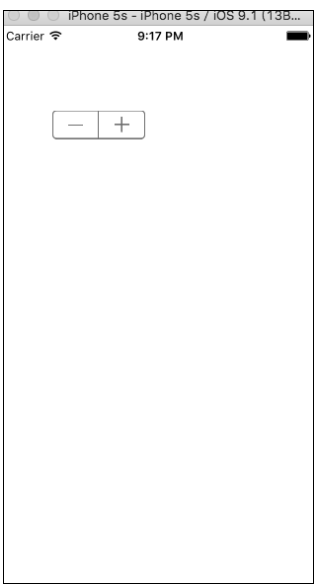

图 2-37 UIStepper 控件

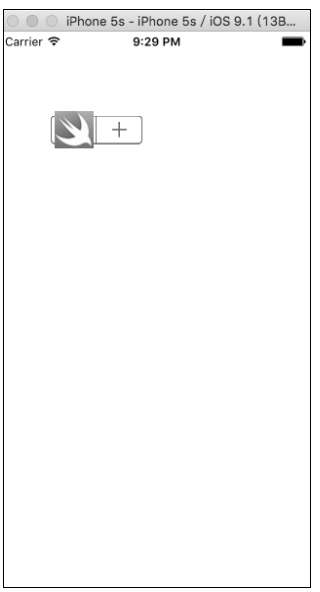

图 2-38 自定义图片的 UIStepper

2.13 选择器控件——UIPickerView

UIPickerView 是一个简易的列表控件，用于提供有限个数的选项供用户选择。UIPickerView 的 UI 设计十分漂亮，是 iOS 系统特有的 UI 模式。在实际应用中，UIPickerView 的应用也十分广泛，如省市县选择列表、时间选择、日期选择等都可以通过 UIPickerView 设计。

2.13.1 创建一个 UIPickerView 控件

UIPickerView 与之前章节中的 UI 控件有着很大的不同，UIPickerView 更加复杂一些，它是通

过代理和数据源的方法对其进行设置和数据源的填充，这种控件的设计模式也是代理模式的应用之一。使用 Xcode 开发工具创建一个名为 UIPickerViewTest 的工程，在 ViewController 类的声明部分添加遵守相应的协议，这里需要遵守的两个协议分别是 UIPickerViewDelegate 和 UIPickerViewDataSource。

```
class ViewController: UIViewController,UIPickerViewDataSource,UIPickerViewDelegate
```

在 viewDidLoad 方法中添加如下代码。

```
    override func viewDidLoad() {
        super.viewDidLoad()
        let picker = UIPickerView(frame: CGRect(x: 20, y: 100, width: 280, height: 150))
        picker.delegate = self
        picker.dataSource = self
        self.view.addSubview(picker)
    }
```

上面的代码中除了对 UIPickerView 进行创建和初始化工作外，还设置了当前类对象为其数据源和代理。在 ViewController 类中实现如下代理方法。

```
    func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return 2
    }

    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int
{
        return 10
    }
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component:
Int) -> String? {
        return "\(component)分区\(row)行数据"
    }
    func pickerView(_ pickerView: UIPickerView, rowHeightForComponent component: Int) ->
CGFloat {
        return 44
    }
   func pickerView(_ pickerView: UIPickerView, widthForComponent component: Int) -> CGFloat
{
        return 144
    }
```

- numberOfComponents 方法返回一个整型数据，用于设置 UIPickerView 视图的分区数，也可以理解为选择列表的列数。
- pickerView: numberOfRowsInComponent 方法的返回值设置 UIPickerView 每个分区的行数，参数 component 用于判断具体的分区。
- titleForRow 方法的返回值设置列表中每一行的数据，这个方法中的两个参数 row 和 component 分别用于区分行与列。
- rowHeightForComponent 方法的返回值设置具体行的行高。
- widthForComponent 方法的返回值设置分区的宽度，即列的宽度。

运行工程，效果如图 2-39 所示。

图 2-39 UIPickerView 控件

2.13.2 UIPickerView 选中数据时的回调代理

UIPickerView 总是会展现几列数据帮助用户进行快速选择，当用户上下滑动 UIPickerView 列表时，列表中的数据会进行上下滑动移动；当移动动作停止时，悬停在 UIPickerView 列表中间的数据即为用户选中的数据，并且此时系统也会调用 UIPickerView 的如下代理方法通知开发者用户的选择。

```
        func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component:
Int) {
            print("\(component)分区\(row)行")
        }
```

在调用 didSelectRow 方法时，row 参数和 component 参数会将用户选择的行和分区的信息传递给开发者。

2.14 通过 CALayer 对视图进行修饰

CALayer 已经不属于 UIKit 框架中的内容，事实上任何一个 UIView 的子类中都包含一个 CALayer 的属性，Layer 是视图中专门用来渲染 UI 的一个层级，而 View 层除了 UI 的展现外，还封装了与用户交互的相关功能，并且 View 层的 UI 展现也是通过 Layer 渲染的。因此，在 iOS 开发中，很多动画的效果都是通过 CALayer 实现的，这些在后面会专门讲解，本节将通过操作 Layer 层的一些简单属性对基本系统控件的 UI 表现进行渲染。

2.14.1　创建圆角的控件

UIKit 中的大多数控件创建时的尺寸都是规则矩形。在实际项目中，开发者可能会需要使用圆角的控件，以 UIButton 控件为例，使用 Xcode 开发工具创建一个名为 CALayerTest 的工程，在 ViewController 类的 viewDidLiad 方法中添加如下代码来设置 UIButton 控件的圆角。

```
override func viewDidLoad() {
    super.viewDidLoad()
    let btn = UIButton(type: .custom)
    btn.frame=CGRect(x:100, y: 100, width: 100, height: 100)
    btn.backgroundColor = UIColor.red
    btn.layer.masksToBounds = true
    btn.layer.cornerRadius = 10
    self.view.addSubview(btn)
}
```

CALayer 对象的 masksToBounds 属性设置为 YES，对视图的边界进行修饰效果才会显现。cornerRadius 属性设置圆角的半径，如果控件的形状为矩形，当这个值设置为控件边长的一半时，控件的形状就会变成圆形。运行工程，效果如图 2-40 所示。

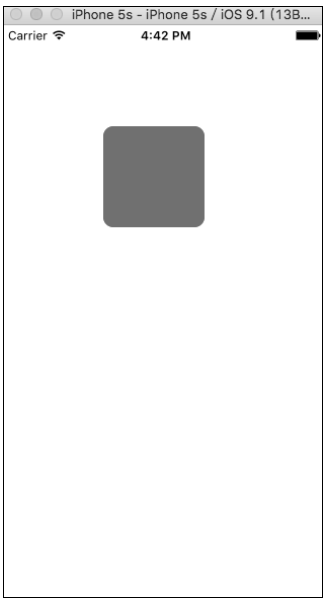

图 2-40　圆角控件

2.14.2　创建带边框的控件

在 iOS 7 系统之前，系统风格的 UIButton 控件支持一种带边框的风格，在 iOS 7 之后，系统不再支持创建出带边框的 UIButton 控件了，但是开发者可以根据需要在 Layer 层做相关修饰来使 UIButton 控件带边框。

使用如下代码来创建带边框的 UIButton 控件。

```
override func viewDidLoad() {
    super.viewDidLoad()
    let btn = UIButton(type: .custom)
    btn.frame=CGRect(x: 100, y: 100, width: 100, height: 100)
    btn.backgroundColor = UIColor.red
    btn.layer.borderColor = UIColor.green.cgColor
    btn.layer.borderWidth = 5
    self.view.addSubview(btn)
}
```

borderColor 属性设置边框的颜色，这个属性需要设置为一个 CGColor 类型的对象，UIColor 对象可以通过调用 cgColor 方法转换成 CGColor 对象。borderWidth 属性设置边框的宽度。运行工程，效果如图 2-41 所示。

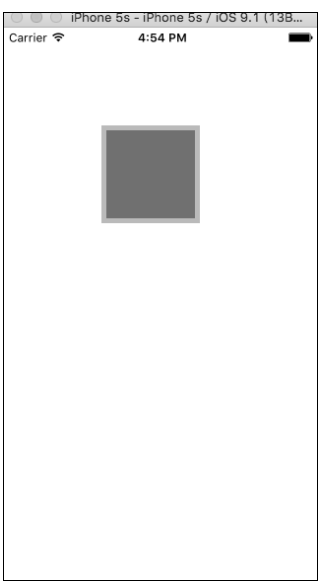

图 2-41　带边框的控件

2.14.3 为控件添加阴影效果

通过 CALayer 的属性还可以为控件添加一个立体的阴影效果，使控件的展示有一定的 3D 视觉效果，使用如下代码来为控件添加阴影。

```
        override func viewDidLoad() {
            super.viewDidLoad()
            let btn = UIButton(type: .custom)
            btn.frame = CGRect(x: 100, y: 100, width: 100, height:
100)
            btn.backgroundColor = UIColor.red
            btn.layer.shadowColor = UIColor.gray.cgColor
            btn.layer.shadowOffset = CGSize(width: 10, height: 10)
            btn.layer.shadowOpacity = 1
            self.view.addSubview(btn)
        }
```

shadowColor 属性设置阴影的颜色必须为 cgColor 对象；shadowOffset 属性设置阴影的位置与原控件位置间的相对偏移；shadowOpacity 属性设置阴影的透明度，如果不设置，就默认为全透明，在界面上将看不到任何效果。运行工程，效果如图 2-42 所示。

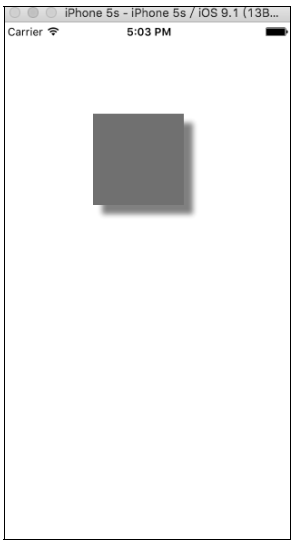

图 2-42 带阴影效果的控件

2.15 警告控制器——UIAlertController

在 iOS 8 系统之前，UIKit 框架中有两个独立的视图控件用于在界面上弹出一个警告视图，分别是 UIAlertView（警告框控件）和 UIActionSheet（活动列表控件）。iOS 8 系统之后，已将 UIAlertView 和 UIActionSheet 进行了规范与统一，将这两个控件合并成新的视图控制器——UIAlertController。关于 UIAlertView 和 UIActionSheet 的使用在本章的扩展节中会介绍，UIAlertController 相比于前面两个控件有很大的优势：一是方法进行了统一，便于开发者使用；二是 UIAlertController 提供了更强的扩展性接口；三是关于回调方法的处理。UIAlertController 使用代码块的形式代替了原先 UIAlertView 和 UIActionSheet 的代理方法，变得更加直观和简洁。

2.15.1 UIAlertController 的警告框

在项目中经常会使用警告框对用户的某些敏感操作提出警告提示。例如，当用户单击某个删除动作的按键时，一般会弹出一个警告框，警告用户是否确定删除，当用户单击删除后，才真正的执行删除操作。

使用 Xcode 开发工具创建一个名为 UIAlertController_AlertView 的工程，由于警告视图一般会在用户做某个操作之后弹出，因此我们可以在 ViewController 类中实现一个当用户单击屏幕时会触发的方法来做演示，代码如下。

```
        override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
            let alertView = UIAlertController(title: "标题", message: "警告的内容",
preferredStyle: .alert)
            let action = UIAlertAction(title: "按钮", style: .default, handler:{ (UIAlertAction)
-> Void in
                print("click")
            })
            let action2 = UIAlertAction(title: "取消", style: .cancel, handler:{ (UIAlertAction)
-> Void in
                print("取消")
            })
            let action3 = UIAlertAction(title: "注意", style: .destructive, handler:{ (UIAlertAction)
-> Void in
                print("注意")
            })
            alertView.addAction(action)
            alertView.addAction(action2)
            alertView.addAction(action3)
            alertView.addTextField { (textfield) in
                textfield.placeholder = "place"
            }
            self.present(alertView, animated: true, completion: nil)
        }
```

touchesBegan 方法是在用户单击屏幕时被系统自动调用，在这个方法中实现对警告视图的创建和设置。

alertController(title:message:preferredStyle:) 方法用于创建 UIAlertController 对象，第 1 个参数为警告视图的标题；第 2 个参数为警告视图的内容；第 3 个参数为警告视图的风格，有两种风格可以选择，分别是 UIAlertControllerStyleAlert（警告框风格）和 UIAlertControllerStyleActionSheet（活动列表风格）。

UIAlertAction 可以理解为是一个封装了触发方法的选项按钮，其 action With Title:style:handler 方法用于创建一个 UIAlertAction 的对象，第 1 个参数为按钮的标题；第 2 个参数为按钮的风格，有 3 种风格可以选择，分别是 UIAlertActionStyleDefault（默认风格），UIAlertActionStyleCancel（取消风格）和 UIAlertActionStyleDestructive（消极风格）。

UIAlertController 的 addAction 方法将向警告视图内添加一个选项按钮。addTextFieldWith：ConfigurationHandler 方法将向警告框中添加一个输入框，开发者可以在代码块中对输入框 TextField 进行一些设置。运行工程，效果如图 2-43 和图 2-44 所示。

提　示

只有当警告控制器的风格为 UIAlertControllerStyleAlert 时才可以使用 addTextFieldWithConfigurationHandler 方法进行输入框的添加，否则会出现错误。

因为 UIAlertController 是一种视图控制器，所以要使用 present ViewController:animated: completion 方法进行跳转。

图 2-43　选项按钮超过两个的警告框

图 2-44　选项按钮为两个的警告框

对于 UIAlertControllerStyleAlert 风格而言，当选项按钮不超过两个时，按钮会横向并列排列，超过两个则会纵向排列。

2.15.2　UIAlertController 之活动列表

活动列表的作用与警告框类似，只是 UI 展现形式不同，使用 Xcode 开发工具创建一个名为 UIAlertController_ActionSheet 的工程，在 ViewController 类中添加如下代码。

```
        override func touchesBegan(_ touches: Set<UITouch>, with
event: UIEvent?) {
            let actionSheet = UIAlertController(title: "标题",
message: "内容", preferredStyle: .actionSheet)
            let action = UIAlertAction(title: "one",
style: .destructive, handler: {(UIAlertAction) -> Void in
                print("one")
            })
            let action2 = UIAlertAction(title:"two",style:.default,
handler: {(UIAlertAction) -> Void in
                print("two")
            })
            let action3 = UIAlertAction(title: "three",
style: .cancel, handler: {(UIAlertAction) -> Void in
                print("three")
            })
            actionSheet.addAction(action)
            actionSheet.addAction(action2)
            actionSheet.addAction(action3)
            self.present(actionSheet,animated:true,completion:nil)
        }
```

图 2-45　活动列表

运行工程，会看到如图 2-45 所示的效果。

2.16　基础 UI 控件扩展篇

本节将介绍前面尚未提到的一些在 iOS 开发中至关重要的 UI 控件，这些控件有从基本控件扩展而来的 UISearchBar 和 UIDatePicker，也有已经被弃用的 UIAlertView 和 UIActionSheet。

2.16.1　搜索栏控件——UISearchBar

UISearchBar 是 UITextField 与 UISegmentedControl 的组合与扩展，UISearchBar 的应用情景更加专一，专门用来创建搜索栏。使用 Xcode 开发工具创建一个名为 UISearchBarTest 的工程，在 ViewController 类的 viewDidLoad 方法中添加如下代码。

```
    override func viewDidLoad() {
        super.viewDidLoad()
        let searchBar = UISearchBar(frame: CGRect(x: 20, y: 100, width: 280, height: 30))
        searchBar.tintColor = UIColor.red
        searchBar.placeholder = "请输入要搜索的内容"
        searchBar.showsScopeBar = true
        searchBar.showsCancelButton = true
        searchBar.showsBookmarkButton = true
//        searchBar.showsSearchResultsButton = true  //设置显示搜索结果按钮
        searchBar.scopeButtonTitles = ["one","two","three"]
        self.view.addSubview(searchBar)
    }
```

UISearchBar 的 tintColor 属性用于设置光标和扩展栏的颜色；placeholder 属性用于设置搜索栏中的提示文字；showsScopeBar 属性用于设置是否显示扩展栏，扩展栏实际上是一个 UISegmentedControl 控件。showsCancelButton 和 showsBookmarkButton 属性分别设置是否显示取消按钮和图书按钮，这里需要注意，这两个按钮只能显示一个，后设置的会覆盖先设置的。setScopeButtonTitles 方法需要传入一个数组，用于设置扩展栏上所有按钮的标题，传入的数组中元素的个数对应扩展栏上按钮的个数。运行工程，效果如图 2-46 和图 2-47 所示。

图 2-46　显示图书按钮的搜索栏

图 2-47　显示搜索结果按钮的搜索栏

UISeatchBar 中单击按钮的触发逻辑和用户输入要搜索的字符时的相关监听都是通过 UISearchBarDelegate 协议中的方法回调，在实现代理方法之前，不要遗漏遵守 UISearchBarDelegat 代理并且将 UISearchBar 实例的 delegate 属性设置为当前视图控制器本身。常用的代理方法的意义如下。

```
        //单击切换扩展栏上按钮时触发的方法
        func searchBar(_ searchBar: UISearchBar, selectedScopeButtonIndexDidChange selectedScope:
Int) {
        }
        //搜索框中字符将要改变时触发的方法
        func searchBar(_ searchBar: UISearchBar, shouldChangeTextIn range: NSRange,
replacementText text: String) -> Bool {
           return  true
        }
        //搜索框中字符已经改变后触发的方法
        func searchBar(_ searchBar: UISearchBar, textDidChange searchText: String) {
        }
        //点击图书按钮触发的方法
        func searchBarBookmarkButtonClicked(_ searchBar: UISearchBar) {
        }
        //点击取消按钮触发的方法
        func searchBarCancelButtonClicked(_ searchBar: UISearchBar) {
        }
        //点击搜索结果按钮触发的方法
        func searchBarResultsListButtonClicked(_ searchBar: UISearchBar) {
        }
        //按键盘上的搜索键触发的方法
        func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
        }
        //搜索框将要开始编辑时触发的方法
        func searchBarShouldBeginEditing(_ searchBar: UISearchBar) -> Bool {
           return true
        }
        //搜索框将要结束编辑时触发的方法
        func searchBarShouldEndEditing(_ searchBar: UISearchBar) -> Bool {
           return true
        }
        //搜索框已经开始编辑时触发的方法
        func searchBarTextDidBeginEditing(_ searchBar: UISearchBar) {
        }
        //搜索框已经结束编辑时触发的方法
        func searchBarTextDidEndEditing(_ searchBar: UISearchBar) {
        }
```

2.16.2 日期时间选择器——UIDatePicker

UIDatePicker 在 UI 展现方面和 UIPickerView 十分相似，但是 UIDataPicker 并非是继承于 UIPickerView 的子类，它是继承于 UIControl 的一个独立控件，因此，在实现逻辑上，UIDatePicker 不采用代理回调的模式。

使用 Xcode 开发工具创建一个名为 UIDatePickerTest 的工程，在 ViewController 类的 viewDidLoad 方法中添加如下代码。

```
    override func viewDidLoad() {
        super.viewDidLoad()
        let datePicker = UIDatePicker(frame: CGRect(x: 20, y: 100, width: 280, height: 150))
        datePicker.datePickerMode = .time
        datePicker.addTarget(self, action: #selector(selector), for: .valueChanged)
        self.view.addSubview(datePicker)
    }
```

UIDatePicker 控件的 datePickerMode 属性设置控件的风格，可选的枚举意义如下。

```
public enum UIDatePickerMode : Int {
    case time // 时间模式
    case date //日期模式
    case dateAndTime //日期和时间模式
    case countDownTimer // 计时模式
}
```

各种风格模式的效果如图 2-48~图 2-51 所示。UIDatePicker 控件使用 addTarget 方法添加触发事件，在触发函数中可以获取 UIDatePicker 中当前的日期时间信息，如下所示。

图 2-48　UIDatePickerModeTime

图 2-49　UIDatePickerModeDate

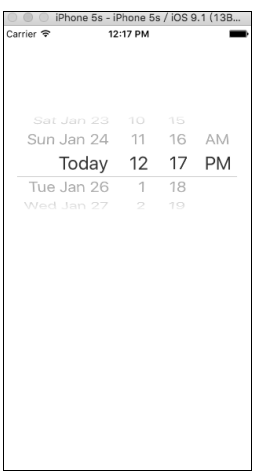

图 2-50　UIDatePickerModeDateAndTime

图 2-51　UIDatePickerModeCountDownTimer

```
        func selector(datePicker:UIDatePicker)  {
           print(datePicker.date)
        }
```

2.16.3 警告视图——UIAlertView

前面提到过，在 iOS 8 之后，系统采用统一的 UIAlertController 代替了 UIAlertView 和 UIActionSheet，但是在目前所有 iOS 各系统用户的占比中，iOS 7 系统仍然存在。既然存在，开发者在编写代码时就要考虑到对 iOS 7 系统的兼容性，如果在 iOS 7 系统上运行使用 UIAlertController 方法的程序，程序就会直接崩溃。

使用 Xcode 开发工具创建一个名为 UIAlertViewTest 的工程，在 ViewController 类中添加如下函数。

```
        override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
           let alert = UIAlertView(title: "标题", message: "内容", delegate: self, cancelButtonTitle:
"取消", otherButtonTitles: "确定")
           alert.show()
        }
```

当用户单击屏幕时，touchesBegan 方法会被调用，UIAlertView 的 initWithTitle 方法创建 UIAlertView 的对象，这个方法中第 1 个参数为警告框的标题，第 2 个参数为警告框的内容，第 3 个参数设置代理，第 4 个参数设置取消按钮的标题，第 5 个参数设置其他按钮的标题，在第 5 个参数之后，可以继续添加标题参数，以逗号进行分隔。调用 show 方法对警告框进行展现。

UIAlertView 的按钮触发方法是通过代理回调的，首先需要在 ViewController.m 文件中遵守如下协议。

```
    class ViewController: UIViewController,UIAlertViewDelegate
```

实现下面的代理方法来监听用户的单击按钮操作。

```
        func alertView(_ alertView: UIAlertView, clickedButtonAt buttonIndex: Int) {
           print("click")
        }
```

alertView:clickedButtonAtIndex 代理方法中的 buttonIndex 参数会传递用户单击的按钮标号，按钮的排号从 0 开始，依次递增。

2.16.4 活动列表——UIActionSheet

UIActionSheet 与 UIAlertView 的用法十分相似，使用 Xcode 开发工具创建一个名为 UIActionSheetTest 的工程，在 ViewController 类中添加遵守协议和创建活动列表的相关代码。

```
    //遵守协议
    class ViewController: UIViewController,UIActionSheetDelegate
    //实现方法
    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
          let actionSheet = UIActionSheet(title: "标题", delegate: self, cancelButtonTitle: "
取消", destructiveButtonTitle: "删除",otherButtonTitles:"确定")
```

```
        actionSheet.show(in: self.view)
    }
    func actionSheet(_ actionSheet: UIActionSheet, clickedButtonAt buttonIndex: Int) {
        print("click")
    }
```

其中，需要注意的地方在于 UIActionSheet 展现时调用 showInView 方法。

提　示

iOS 系统是向下兼容的，如在 iOS 9 系统中使用被弃用的 iOS 8 之前的方法，Xcode 会给出警告，而程序依然可以很好地工作，但是如果在低 iOS 版本中使用了高版本才有的方法，程序就会直接崩溃。

2.17　实战：登录注册界面的搭建

本节将实现一个简易登录注册的界面，目的是练习综合使用本章中介绍的这些独立的 UI 控件，所有复杂控件都是由简单控件组合与扩展而来的，所有复杂的界面也都是将独立的控件进行组合和使用的。

使用 Xcode 开发工具创建一个名为 LoginView 的工程。分析一下需求，我们需要两个界面，一个作为登录界面，一个作为注册界面。将框架中创建好的 ViewController 作为登录界面，再新建一个文件作为注册界面。在 Xocde 的文件导航区右击，选择 NewFile 选项，如图 2-52 所示。

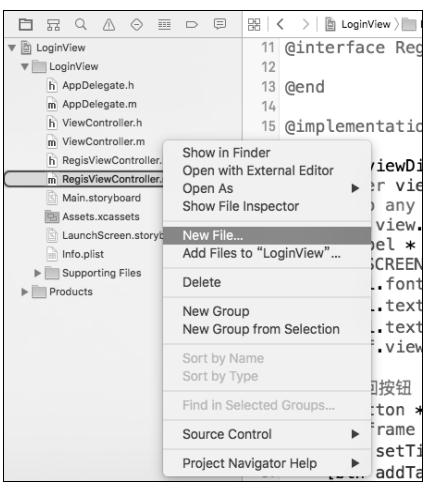

图 2-52　新建一个文件

在弹出的窗口中选择 Cocoa Touch Class 选项，再单击 Next 按钮，如图 2-53 所示。

在弹出的选项窗口中继承的父类一栏选择 UIViewController，类名取为 RegisController，然后单击 Next 按钮，如图 2-54 所示。在弹出的选择创建路径的窗口中直接单击 Create 按钮，这时会看到 Xcode 的文件导航栏中多了 RsgisController 类文件。

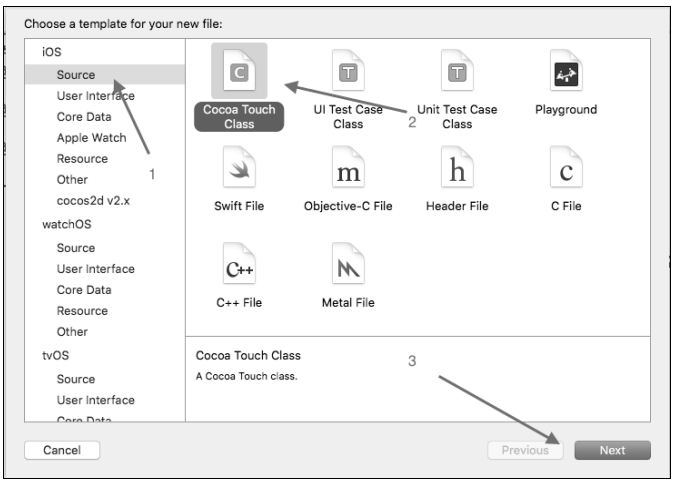

图 2-53　创建 iOS 类文件

图 2-54　设置创建类的相关参数

在 iOS 开发中，有一些规则是开发者们心照不宣的，命名规则就是这样，虽然语法中没有严格规定，但一般在对变量和对象取名时会采用首字母小写的驼峰命名法（如 oneTitle），而类名的首字母一般要大写（如 OneClass）。

创建了注册界面的文件，再回到 ViewController 类中。先来编写登录界面的代码，这里将用户名输入框和密码输入框声明为成员变量，以便于在其他函数中使用这些对象，代码如下。

```
    var loginText:UITextField?
    var passwdText:UITextField?
```

因为创建控件中需要设置控件的位置，所以可以声明一个全局的常量定义屏幕的尺寸，示例如下。

```
let SCREEN_SIZE = UIScreen.main.bounds.size
```

在 ViewDidLoad 方法中添加如下代码。

```
    override func viewDidLoad() {
        super.viewDidLoad()
```

```
            loginText = UITextField(frame: CGRect(x: 20, y: 80, width: SCREEN_SIZE.width-40, height:
30))
            loginText?.borderStyle = .roundedRect
            loginText?.placeholder = "请输入用户名"
            let loginImage = UIImageView(frame: CGRect(x: 0, y: 0, width: 20, height: 20))
            loginImage.image = UIImage(named: "login_user")
            loginText?.leftView = loginImage
            loginText?.leftViewMode = .always

            passwdText = UITextField(frame: CGRect(x: 20, y: 130, width: SCREEN_SIZE.width-40,
height: 30))
            passwdText?.borderStyle = .roundedRect
            passwdText?.placeholder = "请输入密码"
            let passwdImage = UIImageView(frame: CGRect(x: 0, y: 0, width: 20, height: 20))
            passwdImage.image = UIImage(named: "login_pwdico")
            passwdText?.leftView = passwdImage
            passwdText?.leftViewMode = .always
            self.view.addSubview(loginText!)
            self.view.addSubview(passwdText!)
            //创建登录按钮和注册按钮
            let btn = UIButton(type: .system)
            btn.frame = CGRect(x: SCREEN_SIZE.width/4-50, y: 180, width: 100, height: 30)
            btn.setTitle("登录", for: .normal)
            btn.layer.masksToBounds = true
            btn.layer.cornerRadius = 10
            btn.backgroundColor = UIColor.cyan
            btn.addTarget(self, action: #selector(login), for: .touchUpInside)

            let btn2 = UIButton(type: .system)
            btn2.frame = CGRect(x: SCREEN_SIZE.width/4*3-50, y: 180, width: 100, height: 30)
            btn2.setTitle("注册", for: .normal)
            btn2.layer.masksToBounds = true
            btn2.layer.cornerRadius = 10
            btn2.backgroundColor = UIColor.cyan
            btn2.addTarget(self, action: #selector(regis), for: .touchUpInside)
            self.view.addSubview(btn)
            self.view.addSubview(btn2)
        }
```

上面的代码中创建了两个输入框和两个按钮，两个按钮的触发方法实现代码如下。

```
        func regis() {
            let regisController = Regisontroller()
            self.present(regisController, animated: true, completion: nil)
        }
        func login()  {
            if loginText!.text!.isEmpty {
                let alertCon = UIAlertController(title: "温馨提示", message: "请输入用户名",
preferredStyle: .alert)
                let action = UIAlertAction(title: "好的", style: .default, handler: nil)
                alertCon.addAction(action)
                self.present(alertCon, animated: true, completion: nil)
                return
            }
            if passwdText!.text!.isEmpty {
```

```
            let alertCon = UIAlertController(title: "温馨提示", message: "请输入密码",
preferredStyle: .alert)
            let action = UIAlertAction(title: "好的", style: .default, handler: nil)
            alertCon.addAction(action)
            self.present(alertCon, animated: true, completion: nil)
            return
        }
        let alertCon = UIAlertController(title: "温馨提示", message: "登录成功",
preferredStyle: .alert)
        let action = UIAlertAction(title: "好的", style: .default, handler: nil)
        alertCon.addAction(action)
        self.present(alertCon, animated: true, completion: nil)
        return
    }
```

在注册方法 regis 中创建注册界面并进行跳转。在登录按钮的触发方法 login 中先进行用户名框是否为空的判断，如果为空，就会弹出警告框提示用户；然后进行密码框是否为空的判断，如果为空，就会弹出警告框提示用户，如果用户名框和密码框都不为空，就会弹出登录成功的提示。

在注册界面 RigisController 类中添加一个返回按钮和其触发方法，代码如下。

```
    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = UIColor.white
        let label = UILabel(frame: CGRect(x: 20, y: 100, width:
SCREEN_SIZE.width-40, height: 60))
        label.font = UIFont.systemFont(ofSize: 23)
        label.text = "注册界面"
        label.textAlignment = .center
        self.view.addSubview(label)
        //返回按钮
        let btn = UIButton(type: .system)
        btn.frame = CGRect(x: SCREEN_SIZE.width/2-50, y: 220,
width: 100, height: 30)
        btn.setTitle("返回", for: .normal)
        btn.addTarget(self, action: #selector(retu),
for: .touchUpInside)
        self.view.addSubview(btn)
    }
    func retu()  {
        self.dismiss(animated: true, completion: nil)
    }
```

图 2-55　登录注册界面

一个简易的登录注册界面就搭建完成了，效果如图 2-55 所示。