HZ BOOKS
华章科技

摒弃传统入门书讲解方式，项目和任务驱动，逐步引导读者在实战中掌握iOS应用开发的工具、方法和技巧
由浅入深、循序渐进，核心知识点蕴藏在10余个设计精巧的案例中，零基础读者也能开发出高质量应用。

iOS/苹果
技术丛书

Develop iOS App with Swift

跟着项目学iOS应用开发

基于Swift 4

刘铭　陈雪峰　李钢　秦琼　著

机械工业出版社
China Machine Press

跟着项目学iOS应用开发：基于Swift 4

第1章　开始iOS 11和Swift 4编程

大家好，本书的目的是教会大家如何使用iOS 11SDK、Xcode 9和Swift 4编程语言创建iOS应用程序。

不管你是iOS开发的初学者，想通过本书学习如何使用Swift语言编写应用程序；还是之前已经有在iOS 10中开发应用程序的经验，想进一步快速掌握iOS 11的功能和最新版本的Swift 4语言。请放心，本书都可以满足你的需求。

1.1　iOS 11应用程序开发工具

在本节将会向大家介绍开发iOS应用程序需要用到的软件以及相关的硬件。首先，我们必须要拥有一台Mac电脑，不需要是当下最新最快的，但它一定要能运行macOS 10.12.6及以上版本的操作系统。因为苹果的特殊政策，我们只能在macOS上安装iOS应用程序开发工具Xcode。这也就意味着仅仅使用iPad或iPad Pro是不可能完成iOS应用程序开发的任务。如果你拥有一台iMac、MacBook甚至是Mac Mini的话，就足以满足开发的需求。

如果你现在手头确实有些"银子"不足的话，可以考虑购买一台二手的Mac Mini，性价比还是很高的。如果你手头只有PC的话，可以考虑借助Mac in Cloud平台（网址：www.macincloud.com）。在网站上它提供了如同Mac一样的在线云端服务，这样就可以通过现有的PC和互联网实现Mac功能。你只需要在远程系统中下载并安装Xcode就好。如图1-1所示。

另外，还有一种叫作Hackintosh的方式，也就是将macOS操作系统通过非正常的手段安装到自己的PC上，比如通过VMWare、Delphi XE4等方式。但是不管是Mac in Cloud还是用PC安装的Hackintosh，都不能通过这种方式将写好的应用传到iPhone真机上进行测试，唯一的方法就是使用真正的Mac电脑。

虽然不能在真机上运行，但是我们还是可以在Xcode模拟器中运行所编写的iOS项目。而且，即便是在macOS系统上，我们也会在大部分时间利用Xcode模拟器测试项目代码。在模拟器中包含了各种版本的iOS系统，所以可以很好地测试和运行项目。如图1-2所示。

图1-1　MacinCloud网站主页

图1-2　在模拟器中运行并测试iOS项目

开发所用到的软件叫作Xcode，是由苹果公司研发的IDE开发环境。我们可以在Xcode中编写代码、设计界面和调试应用程序，Xcode是完全免费的。

只有在macOS 10.12.6及以上，或者是macOS 10.13及以上环境下才可以下载并安装Xcode 9。强烈建议大家将Mac的操作系统升级到macOS 10.13的最新版本。如何检测你的macOS是否为最新的版本呢？单击屏幕左上角的苹果图标后会弹出一个对话框，在概览标签中就可以看到运行操作系统的版本，如图1-3所示。或者单击对话框右下角的软件更新升级你的操作系统版本。另外，我们还可以在Mac Store中搜索最新的macOS high Sierra（也就是macOS 10.13版本），然后下载安装。

图1-3　在关机本机菜单中查看macOS系统的版本

除了在开发的时候需要安装Xcode以外，最好再安装一款图像编辑软件。比如Adobe的Lightroom、Photoshop，或者是Sketch，如图1-4所示。

图1-4　Lightroom、Photoshop和Sketch软件

在测试应用程序的时候，或是将其上架到App Store之前，你最好有一台iOS物理真机，并进行必要的测试。到底是iPhone还是iPad，这需要根据你的开发目标需求而定。

Xcode模拟器就像一个运行在macOS系统上的虚拟iPhone，我们可以旋转它，并进行简单的手势操作和实现摇晃的功能，可以对其放大或缩小。但是模拟器也会有一定的限制，比如在模拟器中我们无法实现通知、健康或HomeKit功能。

最后需要提示大家的是：在Xcode 7之前，如果要将编写好的程序传到物理真机中，需要向苹果支付99美金的年费。从Xcode 7开始，我们在不需要缴纳年费的情况下也可以进行物理真机测试，你只需要注册一个开发者账号即可。但是，如果想要将应用程序上架到App Store进行销售或推广，则需要缴纳年费。

1.2　下载安装Xcode

接下来，我们需要下载和安装Xcode。Xcode是运行在macOS系统上的一个应用程序，我们会使用它来编写程序代码并创建iOS应用。Xcode是完全免费的，所以我们不用担心会有任何的花销。但是，在我们正式安装Xcode以前，还需要再确认一些事情。

首先要确保我们的Mac有足够的硬盘空间。Xcode安装文件大概是4.5G，所以需要有10G的剩余空间来下载和安装它。要确保这一步非常简单，只需单击桌面左上角的苹果图标，然后找到储存空间，查看硬盘的剩余空间是否够10G，如图1-5所示。

图1-5　查看Mac中的剩余空间

其次，就是需要确定我们的macOS版本是否为最新。检查的方法也非常简单。还是单击屏幕左上角的苹果，然后在概览标签中查看系统的版本是否为10.13或更高。

在确定好前两件事以后，最后一件事，就是确保我们所下载的Xcode版本不是Beta版本。如果下载的Xcode是正式发行版的话，就不用担心它会产生任何问题，而Beta版会包含很多Bug，进而产生很多让你头疼的问题。

在Mac App Store中搜索Xcode，然后单击获取按钮进入Xcode详细页面，这里可以看到当前的Xcode版本是9.2。单击安装按钮，经过一段时间的等待后，Xcode就安装好了，如图1-6所示。

图1-6　在Mac App Store中安装Xcode

1.3　浏览Xcode开发环境

我们在启动Xcode后会看到欢迎界面，这里可以选择以playground开始（Get started with a playground）或者是创建一个新的Xcode项目（Create a new Xcode project），如图1-7所示。

图1-7　Xcode的欢迎界面

实战：快速创建一个全新的Xcode项目。

步骤1：单击Create a new Xcode project向导，在选择项目模板中选择iOS/Application/Single View App，单击Next按钮，如图1-8所示。

图1-8　在选择项目模板中选择Single View App

提示　在模板中还有Game、Master-Detail App、Page-Based App和Tabbed App模板，我们可以根据不同的需求选择不同的模板，除非是创建游戏项目，大部分的开发者都会选择单视图应用程序（Single View App）模板。因为不管是Master-Detail、Page-Based还是Tabbed App模板，都会自动在项目中添加很多代码，而这些代码并不实用。反观Single View App模板，它具有很大的灵活性，可以最大限度地以自定义的方式添加所需要的内容，具体操作方法会在后面详细介绍。

步骤2：在Product Name中需要输入应用程序的名称，这个名字要简单并且最重要的是Cool。这里输入Hello World。Team设置为None。在后面的章节中会讲述如何将App上传到iPhone真机，到时会具体介绍如何设置。

步骤3：Organization Name设置为你公司的名字，如果是个人开发则输入本人的名字即可，比如Liu Ming。

步骤4：Organization Identifier是你的域名的反向，比如你的域名是liuming.cn，这里就需要填写cn.liuming。

步骤5：Language设置为Swift，代表我们使用Swift语言进行项目的开发。

步骤6：在对话框下面还有三个可选框：Core Data是与数据库存储相关；Unit Tests是单元测试相关；UI Tests是用户界面测试相关。在本实例中请不要勾选任何一个选项，如图1-9所示，单击Next按钮。

图1-9　项目设置选项

步骤7：在接下来的对话框中请确定项目保存的位置。这里选择Desktop（桌面），（可以方便我们快速找到它），单击Create按钮。

在项目打开以后，我们就可以看到Xcode所显示的所有不同组件，如图1-10所示。

在界面的顶部是Xcode状态栏，从左侧开始是一个播放（play）按钮，单击它会构建并在模拟器或物理真机上运行项目代码，单击停止（stop）按钮则会终止项目在模拟器或物理真机上的运行。在它们之后的Hello World选项中，我们可以设置在哪个环境里运行应用程序项目。当Mac与iPhone物理真机连接以后，我们就可以在真机上运行，或者手动选择在Xcode模拟器中运行。

图1-10　Xcode的工作界面

当我们选择一种设备版本的模拟器（比如iPhone 7）以后，一旦我们单击Play按钮，就会在macOS上面启动iPhone模拟器，我们可以用鼠标修改它的尺寸。模拟器默认是带曲边的，这意味着可以单击iPhone模拟器左右边缘的仿真按键，实现相应的功能。比如单击Home键可

以让iPhone回到主屏幕，或者单击音量键调整播放声音的大小。建议大家去掉模拟器的曲边显示，在菜单中选择Window，然后取消Show Device Bezels的勾选状态。这样，可以将iPhone屏幕设置得更大一些，方便我们进行调试，如图1-11所示。

图1-11　取消iPhone模拟器的曲边效果

状态栏的中间位置是信息显示窗口，比如在结束运行的时候会显示：Finished running（完成运行）；构建项目的时候会显示Building进度条。当项目出现错误或警告的时候，还会在窗口的右下角出现相应的图标和错误或警告的数量。

位于信息窗口右侧的一组按钮负责切换编辑器的状态，其中前两个按钮的使用频率非常高。第一个是默认的标准编辑器（Standard Editor），它会将Xcode中间部分的区域设置为一个。

当我们单击第二个有两个圆圈图标的按钮时，Xcode会进入辅助编辑器（Assistant Editor）模式，Xcode中间的部分将被分割为两个区域。我们可以将设计的用户界面放在左侧，代码放在右侧，这样方便进行代码与用户界面元素的关联，在后面的章节会对关联有详细介绍。

单击第三个有两个箭头的按钮，会进入版本编辑器（Version Editor）模式，它允许我们可以看到之前代码的版本。比如你在进行了较多代码修改之后，导致项目无法正常运行，就可以通过它回滚到之前的版本，并且可以进行检查和比较。

在顶部状态栏的最右侧还有三个按钮，在单击它们以后，可以分别显示或隐藏Xcode界面中左侧的导航栏、中下部的调试控制台和右侧的工具栏三个面板。

左侧的导航栏面板由9个分项标签组成，其中使用最频繁的是第一项——项目导航，该导航栏中会显示项目中的所有文件，如图1-12所示。

导航栏中的第四项是搜索导航，它包含一个搜索条，并且可以设置对整个项目的搜索还是对某个特定文件夹的搜索，如图1-13所示。

图1-12　导航栏中的项目导航

图1-13　导航栏中的搜索导航

导航栏中的第五项是错误列表，如果项目中出现代码错误或警告的话，通过该列表可以快速找到出现问题的位置。

例如在ViewController.swift文件中随意输入一些字符，Xcode编译器无法解释它们，因此就会高亮显示这行代码，并且在信息窗口、当前文件窗口右上角和当前错误行报出错误的警示图标和原因。单击信息窗口右下角的错误图标以后，导航栏会自动切换到错误列表，在列表中会显示错误的文件名称和内容，如图1-14所示。

图1-14　导航栏中的错误列表

另一种错误类型是警告（Warning），虽然不会造成代码编译错误，但是在运行的时候可能会出现Bug或造成资源的浪费。

例如下面的这段代码，如图1-15所示。

图1-15　一段警告代码

上面这段代码初始化了一个常量number，但是在之后并没有使用它，造成了资源的浪费。因此Xcode的编译器报警（警告用黄色表示）。

第八项是断点（Dreakpoint）导航，可以方便地创建特殊的例外。在代码窗口中单击某一行代码前面的浅槽（行号的位置），就可以创建断点。断点的样子像一个蓝色的箭头，当应用程序在运行的时候遇到了断点就会暂停，我们可以进行调试、观察变量的值、查看运行的状态等。

在设置好断点以后，当再次单击断点后就会变成亮蓝色，代表断点作用暂时被禁止，使用鼠标将其拖曳出浅槽就可以移除断点，如图1-16所示。

图1-16　在Xcode中设置断点

介绍完左侧的导航栏以后，接下来是Xcode底部的Debug区域。该区域被分割为左右两部分。当应用程序运行崩溃的时候，错误信息会显示在右侧的窗口中。这些信息往往会帮助我们找出Bug的原因。另外在代码中往往会通过打印语句输出一些变量的值或状态信息，而这些内容也会显示在该窗口中。当App运行到断点时，可以通过左侧窗口查看当前程序中变量、对象、结构体的值或状态，如图1-17所示。

图1-17　在Xcode中的Debug区域

当我们在编写代码的时候，一般不会用到Xcode右侧的面板。但是当我们需要设计用户界面的时候，就会对它非常依赖。该面板叫作实用工具面板，如图1-18所示。

实战：制作Hello World的用户界面。

步骤1：在项目导航中选择Main.storyboard文件，此时会打开Interface Builder。

步骤2：在实用工具区域的下半部分中找到对象库（Object Library），通过搜索栏找到Label控件，该控件用于显示各种文本信息，如图1-19所示。

图1-18　实用工具面板

图1-19　对象库中的Label控件

步骤3：将Label控件拖曳至ViewController视图上，双击Label将默认内容修改为Hello World。调整好大小，并将其放置到屏幕中央靠上的位置。

步骤4：选择Label下面的背景视图，在实用工具区域的上半部分找到Attribute Inspect标签，将Background设置为蓝色，如图1-20所示。

图1-20　为视图设置蓝色背景

步骤5：选中Label，同样是在Attribute Inspect的Font部分，将Label的文本字号设置为50。此时你会发现当初的Label尺寸不能满足要求了，可以直接使用鼠标调整其大小，设置Label的Color为白色，最后让其居中，如图1-21所示。

构建并运行项目，可以看到我们的第一个项目在模拟器中正常运行了，如图1-22所示。

图1-21　为Label设置颜色和字号

图1-22　模拟器中运行的Hello World应用

如图1-23所示，为大家展示了Xcode的完整工作界面。

图1-23　Xcode的完整工作界面

1.4 初步剖析iOS应用程序

在完成上面这个简单的项目以后，让我们来简单剖析一下：一个iOS应用程序主要由三部分组成。第一部分是视图（View），视图是我们在屏幕上看见的与界面有关的东西，以及将要显示在屏幕上的那些东西。例如按钮、标签或图片这些控件都属于视图。

第二个主要部分是视图控制器（View Controller），它主要通过代码来维护应用程序的运行。比如当用户单击按钮以后程序要做什么，或者是当有数据要显示在屏幕上的时候应该做什么等。

最后一个主要部分就是模型（Model），通过模型我们可以从服务器或本地提取数据，然后通过视图控制器呈现给视图。也可以将用户输入的数据通过视图控制器传递给模型，再由模型进行本地或远程的存储。

以最简单的通信录程序为例，我们通过通信录来管理用户的所有联系人信息。当打开通信录以后，首先会通过视图控制器向模型要数据，比如联系人的电话号码、住址、头像等信息。

模型从数据库或本地获取到这些数据以后，会传回给视图控制器，由视图控制器决定如何用最完美的布局来呈现这些数据。

假如用户想删除一个联系人信息，会通过单击删除按钮，也就是视图的控件，告诉视图控制器。然后视图控制器再将这个请求传递给模型，模型会在数据库中将这个联系人的数据信息从本地或远程数据库中删除，并且将删除状态通知给视图控制器。最后，视图控制器再让视图进行相应数据更新。

刚才我介绍的这些就是MVC设计模式，在iOS开发中这是最常用的一种设计模式。为什么我们要在iOS开发中使用MVC设计模式呢?

因为它非常灵活，方便我们进行管理。比如有一个应用程序，它本身使用的是英文数据库。我们希望它可以使用法文数据库，从而可以将应用程序提供给法国客户使用。因此我们只需要在模型中将原有数据库替换为法文的数据库即可。这样就根本不会涉及视图或视图控制器这两部分，很容易将应用生成一个新的版本。

另一个好处是它们之间相互独立，各自都管理着属于自己的代码。这便于我们调试应用程序里的Bug。比如我们在通信录这个应用程序中看到了错误的布局，那肯定是视图方面的问题。如果发现通信录中联系人名字和头像不匹配，那就可以很快判断出是模型方面出了问题，这样大家都各司其职，使整个项目变得高效灵活。

第2章　Interface Builder介绍

本章我们开始着手创建属于自己的第一个应用程序，并学习如何使用Interface Builder创建简单的用户界面。该程序相当简单，仿照的是2008年上架的一款叫作“我很富有”（I am rich）的应用程序，如图2-1所示。它是由阿明·海因里希（Armin Heinrich）开发，是一款曾在App Store上销售的iOS应用程序。当它被启动以后，在屏幕上只会显示一颗发亮的红宝石，另外还有一个图标，一旦用户单击该图标就会出现下面几行文字：

```
I am rich
I deserv [sic] it
I am good,
healthy &
successful
```

（译文：我富有 我值得 我善良健康又成功。）

图2-1　当时的I Am Rich应用程序

开发者海因里希在该应用程序的描述中说：“这纯粹是一个艺术作品，而且里面完全没有任何隐藏功能，该应用唯一的用意是为了让其他人知道他们足够有钱来买这个应用”。“我很富有”在App Store上的售价分别为999.99美元、799.99欧元及599.99英镑，是App Store中限定应用程序销售价格的最高价。苹果在2008年8月6日，也就是在上架后隔天，未作解释就将应用程序从App Store强制下架。

小知识　在“我很富有”应用下架之前，总共有八位用户购买了该应用程序，且至少有一位用户声称他是不小心买下来的。美国与欧洲分别有6位和2位用户以999.99美元和799.99欧元的价钱买下该应用程序。有5，600至5，880美元分红给阿明·海因里希，苹果则是净赚另外的2，400至2，520美元。

2.1 如何创建Xcode项目

本节我们将创建类似于"我很富有"的iOS应用，整个的操作不是很难，目的是让大家尽快熟悉Xcode的基本操作界面和使用Interface Builder搭建用户界面。

实战：创建I Am Rich应用程序。

步骤1：打开Xcode 9，在欢迎面板中单击Create a new Xcode project，创建一个全新的项目。另外，我们也可以通过Xcode顶部菜单File/New/Project...或者快捷键Shift+Command+N创建项目。

步骤2：在项目模板中选择iOS/Application/Single View App，单击Next按钮。

提示　一般情况下，我们都会从Single View App开始自己的项目，然后再逐步添加其他功能。

步骤3：在项目选项面板中，将Product Name设置为I Am Rich；Organization Name设置为你的名字（例如Liu Ming）；Organization Identifier设置为公司域名的反向；Language设置为Swift。

说明　苹果通过Product Name和Organization Identifier生成一个Bundle Identifier，并通过这个标识在App Store中来区分每一个应用。如果你现在还没有公司或组织域名的话，暂时用cn.你的名字来替代（例如cn.liuming）。

在本项目中不需要存储任何数据，所以不需要勾选Use Core Data。另外，还要确保Include Unit Tests和Include UI Tests两项处于未勾选状态。单击Next按钮。

步骤4：选择好项目的保存位置以后，确保Create Git repository on my Mac处于未勾选状态。这意味着我们不需要对该项目进行代码控制

（Source Control），也就是暂时不需要去管理代码的不同版本。单击Create按钮。

现在项目已经创建完成，可以进行下一步的工作了。

2.2　使用故事板创建用户界面

此时，左侧的导航面板会默认打开项目导航。里面出现的就是目前构成该项目的所有文件。其中一些文件前面是一个雨燕样子的图标，代表是代码文件，里面包含了应用程序的逻辑代码。有些文件前面是一个黄色的图标，代表是界面设计文件，Main.storyboard就是这种类型的文件，我们主要通过它来设计应用程序的用户界面。另外，还有一个LaunchScreen.storyboard界面设计文件，当应用程序启动时，会在屏幕上显示该文件提供的信息，比如公司或你个人的Logo。最后一种类型是Assets.cxassets文件夹，我们可以在该文件夹中放置应用程序会用到的图像、照片或图标资源素材，如图2-2所示。

图2-2　I Am Rich项目中的文件

让我们选中Main.storyboard文件，开始设计I Am Rich项目的用户界面。

实战：设计I Am Rich项目的用户界面。

步骤1：在项目导航选中Main.storyboard文件。此时编辑区域会打开Interface Builder，并显示一个单独的屏幕视图。这是因为我们使用了Single View App模板创建的项目。在这个单独的视图左侧还有一个箭头，它代表的是应用程序启动后在屏幕中所显示的首个视图。如果此时的Interface Builder中有两个视图控制器，我们可以利用这个箭头指定其中一个视图控制器为应用程序启动后首个显示在屏幕上的控制器。

在Interface Builder底部有一个View as：iPhone 8的按钮，如图2-3所示。单击后会打开设备选择面板，可以看到不同屏幕尺寸的iOS设备，分辨率从低到高分别是：iPhone 4s（3.5英寸）、iPhone SE（4英寸）、iPhone 8（4.7英寸）、iPhone X（5.8英寸）、iPhone 8Plus

（5.5英寸）以及iPad 9.7/10.5/12.9英寸三个不同的屏幕尺寸。我们可以随意选择不同屏幕尺寸的设备以及它的方向（横向/纵向）进行设计。

图2-3　Interface Builder中的View as部分

步骤2：在设备选择面板中选择iPhone X，如果视图较大，可以在触控板上通过缩放操作将其调整到合适的大小。

步骤3：打开Xcode右侧的实用工具面板。在对象库中搜索标签（Label）控件，或者使用Command+Option+L快捷键在对象库的搜索栏中输入Label。

实用工具分为上下两个部分，上半部分有6个标签，下半部分有4个标签。这里重点看下半部分的第3个标签——对象库（Object Library），它的图标是圆圈中包含一个正方形，在设计用户界面的时候我们会经常用到它，如图2-4所示。

步骤4：拖曳Label到视图之中，将其放在视图顶部的任意位置即可。在Attributes Inspector中将Label的内容修改为I Am Rich；单击Font右侧的T字标记，将Font设置为Custom，Family设置为Helvetica Neue，Style设置为Thin，Size设置为40，如图2-5所示。

图2-4　对象库中搜索Label控件

图2-5　设置Label的字体和字号

因为修改了字号，所以当前标签中的文本内容根本无法全部呈现在视图中，这时我们需要调整Label的八个控制点，使其全部呈现在视图上。另外，因为Label是iOS开发中最常用的UI控件，所以也可以通过

Command+=快捷键让Interface Builder自动调整Label控件到合适的大小。

小知识　如果拖曳Label到视图的左侧边缘、右侧边缘或视图中央的话，会看到有蓝色的参考线出现，Xcode就是通过这样的方式来帮助设计者快速定位控件的位置。

步骤5：确保选中Label，在Attributes Inspector中将Color设置为White Color。

此时Label中文本的颜色会与设计视图的背景色都为白色。为了能够在视图中快速找到所需要的UI控件，可以通过Document outline快速定位。

在Document Outline中单击View Controller Scene/View Controller/View，如图2-6所示。接着，在Attributes Inspector中将Background设置为RGB 34495E。

技巧　在点开Background以后，可以看到Colors设置面板，它一共包含五个可选方式：色环（Color Wheel）、颜色滑块（Color Sliders）、调色板（Color Palettes）、图像调色板（Image Palettes）、笔（Pencils）。在颜色滑块标签中可以通过RGB方式设置颜色，如图2-7所示。

图2-6　在Document Outline中选View视图

图2-7　Xcode中的颜色设置面板

此时的Label是被我们随意放置的，下面，我们需要精准定位UI控件的位置，也就是一切从客户的角度出发，为客户的需求考虑。不管是标

签（Label）还是按钮（Button），不管是位置还是大小，都需要有精确的设定。

步骤6：再次选中Label，在Size Inspector的View部分中，设置x为108，y为100，width为160，height为50。

此时Label在视图中的大小与位置看着就比较自然、舒服了，如图2-8所示。但是这种方法是最初级、最原始的，在之后的学习中，将会通过自动布局和Sized Classes等特性进行UI控件的完美布局。

图2-8　Label编辑后的最终效果

2.3　如何定位用户界面元素

在上一节我们创建了UILabel控件，并在Size Inspector中设置了它的大小和位置。本节会对设计界面时的定位做一个简单的解释，这样可以便于我们在将来为自己的应用设计用户界面。

可以想象一下，我们将iPhone的屏幕划分为细小的网格，如图2-9所示。而屏幕的左上角就是这张“坐标纸”的原点（0，0），往右是水平方向的正值，往下是垂直方向的正值。在垂直方向上与我们平时使用的真正坐标值正好相反。

图2-9　被划分后的屏幕坐标

对于iPhone 6来说，它水平方向是375个点，垂直方向是667个点。通过苹果的官网我们可以找到所有不同屏幕尺寸iPhone设备的分辨率，如图2-10所示。

当你在屏幕上定位像UILabel这样的用户界面元素的时候，所指定的位置是UILabel控件左上角的那个点，不管是按钮、开关或图像视图都是如此。这样，不管用户界面元素是正方形还是长方形，这个点是可以确定的。

另外，我们还会通过width和height确定界面元素的宽和高。所以要想在视图中确定一个界面元素的位置与大小，我们只需要设置好四个属性即可，即x、y、width和height的值。这与我们之前在Size Inspector中设置的四个属性一致，如图2-11所示。如果你愿意，可以随意修改这四个值来了解每个属性所实现的功能。

图2-10　iPhone手机分辨率指南

图2-11　设置UILabel控件的位置和大小

2.4　导入图像素材到Xcode项目

接下来，我们将会为项目添加一些图片素材，并将一张红宝石图像呈现到应用的视图之中。为了可以在屏幕上显示图像，我们需要添加一个图像视图（Image View）。

实战：在视图中添加一个图像视图。

步骤1：打开Main.storyboard文件，在对象库中找到Image View，通过介绍可以了解到，Image View可以用于显示一个单独的图像或通过Image数组所连成的动画。将Image View拖曳到屏幕中央的位置，如图2-12所示。

图2-12　设置UILabel控件的位置和大小

步骤2：在选中Image View的情况下打开Attributes Inspector，这里面全部都是与Image View相关的属性。其中最重要的一个属性是Image，它用于指定在Image View中显示的图像。在之后的操作中，我们会向大家介绍如何为项目添加图片素材。

步骤3：在选中Image View的情况下打开Size Inspector，将x设置为53，y设置为240，宽和高均设置为270。

步骤4：打开项目中的Assets.xcassets文件，在右侧的列表中有一个AppIcon文件夹，其内部有很多空槽，用于为项目添加各种图标。当我们将项目上传到App Store上时，Xcode会检查并确保所有的空槽都填充了符合要求的图标。

本书中涉及的项目源代码以及素材均可以在GitHub网站中下载，地址为https://github.com/liumingl/ios-11-Swift-4-Tutorial。

在素材文件夹中找到相关资源，然后对照空槽下面的描述将对应的图标拖曳到空槽之中。比如将Icon-App-40x40@2x.png文件拖曳到iPhone Spotlight iOS 7-1140pt的2倍空槽中。因为它只接受40×40点，也就是80×80像素的图像，用于在iOS搜索的时候使用，如图2-13所示。

图2-13　设置应用程序的AppIcon

步骤5：此时，在AppIcon中有很多用于iPad设备的图标空槽，由于本项目只是针对iPhone设备，所以取消勾选工具区域Attributes Inspector的Pad选项即可。

步骤6：在资源文件夹中找到diamond@2x.png文件，并将其直接拖曳到Assets.xcassets文件的列表之中，此时在AppIcon的下面会添加一个新的diamond条目。

这里大家可能已经注意到iOS项目中图片素材文件的命名方式有些奇怪。在一般情况下文件名称被分成2部分，@前面的部分是图片素材的文件名称，而@后面的部分是1x、2x或3x，如果在iPad mini一代设备上显示图片的话，系统会自动调用1x的图像，因为它是标准的屏幕。如果在iPhone SE/5/6/7/8显示图片的话，系统会自动调用2x的图像，因为它们都是Retina显示屏。如果在iPhone Plus机型显示图片的话，系统会自动调用3x的图像。也就是说为了可以让你的应用在所有设备上完美运行，你需要准备3张不同分辨率，但是内容一样的图片。但是，如果在Plus机型上面运行，并且没有找到3x图像的情况下，它会自动使用2x的图像替代，如果没有的话则会再查找1x的图像（它是向下兼容的）。

步骤7：回到Main.storyboard文件，选中刚才添加的Image View，然后将Attributes Inspector中的Image设置为diamond。因为在Assets.xcassets文件中已经添加了diamond的素材，所以该图像会直接显示在Image View之中（如图2-14所示）。

图2-14　设置Image View的Image属性

步骤8：调整Content Mode为Aspect Fit，让Image View中的图像按原图比例调整到合适的大小。

2.5　运行并测试项目

在本章的最后我们将会构建项目，并在模拟器中运行和测试项目。

首先你需要通过Xcode底部的设备选择面板确定你当前项目的用户界面是针对哪款iOS设备开发的，在我们没有学习任何关于自动布局特性的内容之前，暂时还不具备为所有不同屏幕尺寸的iPhone设计完美用户界面布局的能力。

如果你此时选择iPhone SE或者是iPhone 4s的话，界面效果会非常糟糕，如图2-15所示。不过没有关系，一旦我们学习了自动布局特性以及如何为界面元素添加相关约束以后，这个问题就迎刃而解了，因为我们可以仅设计一套用户界面布局，然后让它完美地呈现到所有不同尺寸、不同方向的iOS设备上面。

图2-15　设置Image View的Image属性

但是目前，我们需要一切都保持在iPhone X上面的设计规格，在模拟器的选择上也要设定为iPhone X，如图2-16所示。

一旦我们选择在iPhone X模拟器中运行I Am Rich项目，并通过菜单栏Product/Run或者使用Command+R快捷键运行项目，Xcode顶部状态栏中的信息窗口就会呈现出各种状态和相关进度。在成功构建项目以后，我们就会看到运行应用的模拟器，如图2-17所示。

模拟器实际上是一个运行在macOS系统上的应用程序，它会模拟成iPhone或iPad，并且受到存储和内存的限制。显然，Mac的内存要大于iPhone或iPad的容量，因此在模拟器中运行正常，但是在物理真机上却发生崩溃的情况也是会发生的。

一旦模拟器启动以后，就会自动载入并运行项目中的应用程序。如果在模拟器菜单中选择Hardware/Home或者使用Shift+Command+H快

捷键就可以回到Home屏幕，如图2-18所示，再次单击应用图标还可以回到之前的应用。

图2-16　设置在iPhone X模拟器中运行项目

图2-17　在模拟器中运行的I Am Rich应用

图2-18　在模拟器中回到Home屏幕

第3章　在iPhone真机上安装应用

在Xcode 7之前，如果我们想要将编写好的应用程序安装到iPhone物理真机上进行调试是非常麻烦的。首先要通过开发者账号登录到苹果的开发者管理页面，将用于调试的iPhone的唯一标识码（UDID码）添加到后台，然后更新provisioning profile证书文件，再下载这个文件到Mac电脑，并安装到该电脑上，经过好几个步骤以后，才能进行真机调试。这还不包括会遇到证书过期、添加新机器等问题。而且，更主要的一个问题：你需要为此缴纳每年99美金的年费。

从Xcode 7开始，苹果改变了自己在许可权限上的策略，开发者无须注册开发者账号，仅使用Apple ID就能在物理真机上下载和进行测试体验。不过，如果你打算向App Store提交应用的话，那仍然需要支付费用。

3.1 使用Xcode将项目下载到物理真机

在将项目上传到物理真机之前，请允许我向大家介绍一下“Sideloading”。Sideloading主要是在互联网上使用的一个术语，与“上传”和“下载”类似，被引申为在两个本地设备之间传输文件的过程，特别是在计算机和移动设备，例如手机、iPad或电子阅读器等。

Sideloading通常是指通过USB线、蓝牙、Wi-Fi或通过写入存储卡将媒体文件传输到移动设备中的过程。当涉及iOS应用程序时，Sideloading通常意味着在iOS设备上自行安装自制的应用程序。

接下来，我们要将I Am Rich应用上传到iPhone真机上，但是过程会稍微有点儿曲折。因为Apple的安全需求，所以我们要严格按照下面的步骤操作。

实战：将I Am Rich应用上传到iPhone真机上。

步骤1：在Xcode中打开之前的I Am Rich项目，确定Xcode的版本与iPhone上系统的版本一致。这一步非常重要，因为版本不一致会导致应用程序无法上传到iPhone真机。在Xcode的About菜单选项中查看当前Xcode版本，如图3-1所示，如果当前的版本为9.1或9.2，则iPhone上对应的iOS版本就必须是11.1或11.2。一般来说，高版本的Xcode可以兼容低版本的iOS，但是强烈建议两个版本保持一致。

步骤2：在项目导航中选择顶部的I Am Rich条目（蓝色图标的），在右侧面板中选择TARGETS部分的I Am Rich，并确保选中General标签。你会发现在General标签中有很多设置选项。

步骤3：在Signing部分，确保自动管理签名（Automatically manage signing）处于勾选状态。在该状态下允许Xcode自动创建项目配置文件，设置开发证书和所有的代码签名。除此以外，还有一些工作需要我们手动完成，但是已经比Xcode 7之前的操作简单多了。

步骤4：单击Team右侧的下拉列表，当前的选项是None，如果你之前没有在Xcode中设置过该选项的话，选择添加一个账号（Add an Account...），如图3-2所示。在Accounts面板中添加你自己的Apple ID，该Apple ID可以是你之前用于下载iOS应用的账号，并且不需要将其升级为开发者账号。输入完成以后单击Sign In，如图3-3所示。在面板中单击刚刚添加好的账号，可以看到一些账号相关信息以及当前账号的角色是User，如果是开发者账号的话，登录以后的角色将会是Agent，如图3-4所示。

图3-1　通过Xcode查看版本号

图3-2　添加一个全新的账号

图3-3　利用Apple ID账号登录

图3-4　检查Apple ID账号

步骤5：关闭当前面板并回到General标签，将Team从None修改为新添加的账号。此时，Provisioning Profile和Signing Certificate iPhone Developer也发生了相应的改变，如图3-5所示。

图3-5　在Signing中设置用户账号

步骤6：利用数据线连接iPhone真机与Mac，并确定相互之间已经完全信任。连接成功后，在Xcode菜单中选择Product/Destination，确保iPhone真机的名字出现在Device中并选中它，如图3-6所示。

图3-6　确认Xcode是否认出物理真机

步骤7：在Xcode顶部工具栏的Scheme中再次确认I Am rich项目是运行在iPhone真机后，构建并运行项目。在构建项目的同时，可以看到消息窗口中会显示当前的操作状态，例如准备、安装、运行等。在Xcode上传程序到iPhone的时候，有时会弹出对话框提示用户macOS想要做一些事情，需要用户输入当前登录的用户名和密码，输入完成以后单击Allow按钮。这一步非常重要，因为Xcode会在系统层面做出一些改变和设置，如果单击Deny就会导致后面的操作失败。

步骤8：在Xcode上传应用到iPhone的最后，会弹出一个错误面板，如图3-7所示。这是因为你现在iPhone真机上面还没有信任用于开发的配置文件。目前，Apple ID只是设置在了你的iPhone上，单击OK按钮关闭错误面板。

图3-7　弹出的错误面板信息

步骤9：在iPhone真机上面打开设置→通用→设备管理，单击信任“xxxxx@icloud.com”的连接并确认，如图3-8所示。

图3-8　在iPhone的设置中信任该设备

步骤10：回到Xcode并确保I Am rich项目还是会安装在iPhone真机上面，再次构建并运行项目。在此期间你可能会得到另一个错误信息，告知你需要解锁iPhone以后才能运行，现在只需要解锁你的iPhone即可。

另外，在General标签中还有个Deployment Target选项，它代表当前项目所部署的iOS版本号，如果你选择的是11.1，则会向下兼容11.0或

者是10.3。

挑战　利用之前所掌握的技能，仿照之前的I Am Happy，完成I Am Busy项目。在本书素材中会为大家提供非常Cool的App Icon，以及用于显示在屏幕上的图片。

3.2　通过GitHub下载项目样例代码

完成上面挑战的第一步就是要在GitHub网站下载初始项目文件，很多程序员都会将自己的项目代码放在GitHub上面进行维护，或分享给其他程序员，如图3-9所示。git是一个免费的开源项目，它的创始人就是著名的Linux系统创始人Linus Torvalds。git有很多对于开发者有用的特性，其中最重要的一个就是版本控制（Version Control）。

图3-9　GitHub网站主页面

想象一下，有两位程序员阿刚和雪峰在共同维护一段代码，他们两位应该如何高效地工作呢?

方案一：阿刚先维护这段代码，做完以后再给雪峰继续维护。大家都知道这是效率最低的工作方式，因为同一时间只有一个人在工作。面对这样的工作效率，老板是绝对不会答应的!

方案二：阿刚和雪峰各自复制一份代码独自去维护。但问题在于很难将两份代码合并到一起。

方案三：利用版本控制，将主拷贝存储到核心服务器中，不管是阿刚还是雪峰都可以获取项目代码的主拷贝到本地，各自修改好以后再更新到核心服务器中。例如：阿刚先从服务器获取主代码，在完成修改以后将其合并到主代码库，此时的代码库变成了2.0版本。接着雪峰从服务器获取主代码的时候，得到的就是最新的2.0版本，他可以继续维护代码的其他部分。另外，如果两人在同一时间维护同一段代码应该怎么办呢？阿刚的操作与之前一样，服务器代码将更新为2.0版本。而此时雪峰还是在1.0版本的基础上维护代码，在维护完毕并上传合并的时候，git会检查是否与当前服务器上面的代码有冲突，如果没有则将雪峰的新代码合并到主版本库，此时代码库变成了3.0版本。如果有冲

突，git会列出引起冲突的代码行，由雪峰通过手工的方式在2.0版本的基础上修改相关代码。

git不仅可以对程序代码做版本控制，也可以对PDF、Word、Excel、PPT等文件进行这样的操作。它的好处在于git可以保存每次提交时的状态，这样文档就可以随时回滚到之前所提交的某一个状态。

本书会通过GitHub来维护所有代码，GitHub是一个面向开源及私有软件项目的云端托管平台，因为只支持git作为唯一的版本库格式进行托管，故名GitHub。

本书会涉及很多实战项目，如果每个练习都从头开始，会浪费很多不必要的时间，我们希望大家更多去关注那些重要并且有趣的内容，因此在GitHub中会存储初始项目代码和最终代码，另外还会存储项目所需要的资源素材。

实战：从GitHub下载I Am Busy项目。

步骤1：登录GitHub，并在搜索栏中输入I Am Busy。找到liumingl/I-Am-Busy并单击进入相关页面，如图3-10所示。为了可以快速找到需要的项目，我们可以在搜索的时候指定Language为Swift，以便缩小查找的范围。

图3-10　GitHub网站主页面

步骤2：为了可以拷贝项目代码和素材文件，单击Clone or download按钮，如图3-11所示。

图3-11　下载项目代码到本地

步骤3：将下载的zip文件解压缩，进入I Am Busy-Start文件夹，打开项目文件。

步骤4：在项目导航中选择TARGETS，在General标签将Bundle Identifier中的域名修改为自己的域名，如图3-12所示。

图3-12　打开项目以后修改Bundle Identifier设置

实战：仿照I Am Rich项目，完成I Am Busy项目。

步骤1：从对象库中添加一个UILabel到Main.storyboard的视图控制器的View中。

步骤2：设置UILabel的text属性为I Am Busy。设置Label的字体为Helvetica-Neue-Thin，字号为40。

步骤3：在选中Label的情况下，使用Command+=组合键将UILabel调整到合适的大小，并定位好其位置。

步骤4：再次通过对象库将Image View拖曳到视图控制器的View中，设置Image View显示名为Busy的图像，并且设置Content Mode为Aspect Fit。你可以调整Image View到合适的大小与位置。

步骤5：改变视图控制器中View的背景颜色，并对个别UI控件进行个性化调整。

在Xcode模拟器或iPhone真机上运行该项目，效果如图3-13所示。

图3-13　I Am Busy项目的最终效果

第4章　构建简单的掷骰子游戏

在本章中我们要构建一个简单的掷骰子游戏，在之前的章节中，我们学习了如何使用Interface Builder设计并布局用户界面，进而独立制作了I Am Busy项目。如果你还没能独立完成I Am Busy项目的话，强烈建议你真正自己完成以后再继续下面的内容。

本章的项目非常简单，当启动应用以后会看到一个Logo、两个骰子（拼音：tou zi，作者读shai zi几十年了）和一个按钮，如图4-1所示。当单击按钮以后，两个骰子的面就会发生变化，就好像它们在真正地滚动一样。如果将应用安装到iPhone真机的话，将是一个既简单又非常酷的应用！

图4-1　掷骰子游戏的主界面

在本项目中，我们还是会使用Interface Builder设计界面，将相关图片添加到Xcode Assets中，大部分都会与视图和外观相关。除此以外，我们还会接触一些代码，它们都是最基础的，比如说数组。还有就是如何将代码与界面进行关联，了解IBOutlet和IBAction是如何工作的，以及如何修复一些常见的Bug。最后，我们还会编写生成随机数的方法。

这个应用虽然非常简单，但是其中90%的操作都是一名iOS程序员每天都会涉及的，因此也是非常重要的！

4.1　如何设计掷骰子游戏

首先，让我们先下载相应的素材，在GitHub中搜索IOS 11Swift 4Tutorial就可以定位到该项目。

实战：创建Dicee的用户界面。

步骤1：启动Xcode，在欢迎界面中单击Create a new Xcode project，选择iOS/Application/Single View App，Product Name设置为Dicee，Team保持不变，如果你没有组织名称的话，在Organization Name中填写自己的名字，在Organization Identifier中填写自己全名的域名或者公司的网址。确保Language为Swift，确保最下面的三个复选框处于未勾选状态，单击Next按钮。

步骤2：选择保存项目的本地位置，在确保Source Control处于未勾选状态后，单击Create按钮。

在成功创建好项目以后，接下来我们需要继续在故事板中设计用户界面。

步骤3：打开Main.storyboard文件，在设备选择面板中将设计界面设置为iPhone X屏幕，然后在对象库中将Image View拖曳到视图之中，并将其调整到整个视图大小。

注意　在调整Image View位置的时候，Interface Builder会自动出现参考线帮助我们定位，当Image View在离视图边缘8个点位置的时候，会出现自动停靠的效果，因为Xcode会认为UI控件不适合放置在屏幕的边缘处。大可不必管它，直接将Image View放在视图的最上角，我们需要让Image View中的图像作为整个屏幕的背景图。

虽然项目默认使用iPhone 8的屏幕，但是我们手工将屏幕尺寸修改为iPhone X的5.8英寸。在之后的章节中会向大家介绍如何通过自动布局和约束来进行完美布局。

步骤4：在项目导航中打开Assets.cxassets文件，将本书所提供的素材图片添加到里面。其中包括六套骰子面图片、一套背景图片、一套Logo图片和一套60×60点的App Icon图片，如图4-2所示。

图4-2　Dicee项目中导入到Assets.cxassets文件的素材资源

步骤5：回到Main.stroyboard文件，选中之前作为背景的Image View，然后在Attributes Inspector中将Image设置为newBackground。再从对象库拖曳一个Image View，放置在视图的顶部，将其Image设置为diceeLogo，将Content Mode设置为Aspect Fit，并调整好其位置和大小，如图4-3所示。

步骤6：再从对象库中拖曳两个Image View到视图之中，选择其中一个在Size Inspector中将x设置为40，y设置为300，宽和高均设置为120。将另外一个Image View的x设置为215，y设置为300，宽和高还是120。

注意　本实战中所使用的是以iPhone X屏幕为参照的视图尺寸，所以在实战中请注意屏幕尺寸的选择，在其他屏幕尺寸下使用当前的设置参数并不会有好的结果。

步骤7：确定选中其中一个骰子的Image View，在Attributes Inspector中将Image设置为dice1，再将第二个骰子也做同样的设置，如图4-4所示。

设计用户界面的最后一步是要在视图中添加一个“掷骰子”按钮，当用户单击它以后骰子的面要发生变化。

步骤8：在对象库中将Button拖曳到视图的下半部分，将按钮的标题修改为掷骰子，将Font设置为Helvetica Neue，字号设置为25，最后将按钮调整到合适的大小和位置。设置标题的颜色为白色，按钮的颜色为粉色，如图4-5所示。

图4-3　设置Image View的图像

图4-4　设置骰子的Image View图像

图4-5　设置骰子的Image View图像

现在，我们已经为掷骰子游戏设计好了用户界面，在下面的章节中，会将之前所设计的用户界面元素与程序代码进行关联。这样，当用户单击按钮的时候就可以通过代码让骰子的面发生变化。

4.2　建立代码与界面元素的关联

在设计好用户界面以后，接下来我们要让程序代码了解、掌握一些关键的界面元素，因为这些界面元素在程序运行期间会发生变化，或者是要响应用户的交互操作。

其实我们通过Interface Builder创建的用户界面文件（Main.storyboard）也是代码构成的，在项目导航中的Main.storyboard文件上单击鼠标右键，在弹出的菜单中选择Open As/Source Code，此时就会看到XML格式的Main.storyboard文件的内容。幸运的是，你根本不用担心如何去读懂它，因为在绝大部分的时间里，我们都是在Interface Builder中搞定用户界面。

在本项目中，当我们单击“掷骰子”按钮以后，要以随机图像的形式来呈现骰子的滚动效果，因此需要让程序代码知道在什么时候按钮被单击了，以及是否要改变骰子的外观。在程序代码与用户界面之间，我们需要处理好两种情况：一种是在某种情况下改变某个界面元素的外观，另一种是在某种交互行为发生的时候，通知程序代码。这两种情况的处理会在后面的章节中详细介绍。

首先，需要将Xcode切换到助手编辑器模式，单击顶部工具栏右半部分中画有两个圆圈的按钮。在助手编辑器模式下，Xcode的编辑窗口被分割为左右两部分，你可以同时看到设计部分和代码部分。如果你使用的是13英寸的MacBook Air或是12英寸的MacBook，可能会需要更多的编辑空间，否则在助手编辑器模式下会很难操作。这里建议你暂时关闭左侧的导航区域，右侧的工具区域以及中间部分的Document Outline（我们称之为大纲导览视图），这样便扩大了编辑区域的空间，如图4-6所示。

图4-6　将Xcode切换到助手编辑器模式

让我们选中Main.storyboard文件，开始设计I Am Rich项目的用户界面。

实战：将界面元素与代码建立关联。

步骤1：将Xcode切换到助手编辑器模式，并确保左侧窗口打开的是Main.storyboard故事板文件，右侧窗口打开的是ViewController.swift代码文件。

如果右侧打开的是其他文件，可以按住option键并在项目导航中单击ViewController.swift文件。

步骤2：按住Control键，鼠标拖曳代表左侧骰子的Image View，此时会有一条蓝色细线出现在界面元素与鼠标之间，并且蓝线的端点会跟随鼠标移动，如图4-7所示。

图4-7　为Image View添加IBOutlet关联

步骤3：在下面的代码之间松开鼠标，在弹出的对话框中确认Connection为Outlet，Name设置为diceImageView1，确认Type为UIImageView类型，Storage为默认的Weak类型，最后单击Connect按钮，如图4-8所示。

图4-8　设置Image View的IBOutlet选项

在单击Connect按钮以后，可以发现Xcode为我们自动添加了一行代码。

```
class ViewController: UIViewController {
  @IBOutlet weak var diceImageView1: UIImageView!
```

Xcode创建的IBOutlet关键字是Interface Builder Outlet的缩写，变量diceImageView1是一个指针变量，它指向Main.storyboard中的Image View控件，该变量的类型是与界面元素对应的UIImageView。

在这步操作中有两件事情需要大家注意：一是确认关联的类型一定为Outlet以及记住IBOutlet的名字diceImageView1；另一件事是请暂时忽略weak关键字以及最后的感叹号（！），因为它属于高级一阶的Swift语法内容。

在设置IBOutlet关联的Name属性时，Swift语言对于变量和方法的命名会使用驼峰命名法（CamelCase）。它是编写程序时的一套命名规则，正如它的名称CamelCase所表示的那样，是指混合使用大小写字母来构成变量和函数的名字。即变量名的第一个单词使用小写，从第二个单词开始均要求首字母大写。该命名法在程序设计时非常实用，因为可以很容易地区分每一个单词。

另外，当成功建立IBOutlet关联以后，在其代码行前面的灰色沟槽中，可以看到一个实心的圆圈。并且当鼠标悬停在其上面的时候，Interface Builder中已经建立好关联的界面元素就会被高亮显示。这个功能非常有用，如果它是一个空心圆，则代表IBOutlet变量还没有与故事板中的界面控件建立关联。如果对应到了错误的界面元素，则代表建立了错误的关联。

步骤4：为另一个Image View建立IBOutlet关联，拖曳另一个Image View到之前ViewController.swift文件中@IBOutlet代码的下一行。Connection设置为Outlet，Name设置为diceImageView2，Type和Storage保持默认即可，单击Connect按钮。

```
@IBOutlet weak var diceImageView1: UIImageView!
@IBOutlet weak var diceImageView2: UIImageView!
```

最后，我们还要为“掷骰子”按钮设置一个关联，与之前所建立Outlet关联有些许不同，之前的关联是想要改变某个界面元素的外观。现在要

建立的关联叫作IBAction，它用于允许代码响应用户与应用程序界面的交互行为。

步骤5：与之前建立IBOutlet关联一样，在“掷骰子”按钮上面按住鼠标右键并将其拖曳到ViewController.swift文件中，但是这次需要将其拖曳到文件中最后一个大括号的上方。在弹出的面板中将Connection设置为Action，Name设置为rollButtonPressed，Type设置为UIButton，Event设置为Touch Up Inside。单击Connect按钮后，会出现下面的代码。

```
  @IBAction func rollButtonPressed(_ sender: UIButton) {
  }
}
```

在面板中单击Event后会弹出一个事件列表，其中Touch Up Inside是按钮控件最为常用的一个事件，它代表当用户单击按钮并在按钮区域范围内抬起手指的这个事件。当然，我们也可以选择其他基于按钮的交互事件，例如Touch Drag Outside，代表按住按钮，并将手指移到按钮之外的事件，通常用该事件实现拖曳操作。

此时，我们可以观察下IBOutlet和IBAction之间的不同，IBOutlet用于改变界面元素的外观，而IBAction是在界面元素有人机交互事件时调用设置好的IBAction方法。

二者之间的另外一个不同是：IBOutlet是一个变量的声明，IBAction是类中的一个方法。在rollButtonPressed（）方法中，我们还将添加当用户单击按钮后掷骰子的相关代码。

提示　不知你是否想对Image View添加IBAction方法，如果这样做的话，在Connection中并不会看到Action的选项，因为Xcode默认Image View只能用于呈现各种图像，并不接受用户的互动，而按钮除了可以修改外观以外还可以响应用户的交互。因此Image View只能有IBOutlet，而Button则两者都具备。

4.3 IBOutlets/IBActions调试

上一节，我们学习了如何在界面元素和代码之间建立IBOutlet和IBAction关联，但是对于初学者来说，在建立关联的时候往往会产生一些错误，造成一些Bug。

让我们先观察下面这段代码：

```
class ViewController: UIViewController {
  @IBOutlet weak var diceImageView1: UIImageView!
  @IBOutlet weak var iLoveThisGameImageView: UIImageView!
```

代码中的第一个IBOutlet变量——diceImageView1，非常准确地表达了其代表的界面元素有什么用。但是第二个IBOutlet变量——iLoveThisGameImageView并没有准确地体现出它有什么用，如果现在你想马上修改变量名称的话，错误也就会随之产生。

将iLoveThisGameImageView变量名称修改为diceImageView2，此时你会发现第二个IBOutlet代码行左侧的灰色沟槽中变成了空心圆，代表该IBOutlet变量目前并没有与任何界面元素建立关联。如果此时构建并运行应用程序，会导致应用无法工作而发生崩溃的情况。

当应用程序崩溃的时候，可以在Xcode的代码中看到红色高亮标识，代表有问题发生。调试控制台也会出现并告诉你因为一个未捕获到的异常导致应用程序终止运行。这是一个非常常见的错误，作为一名初学者你可能会在这里看到各种各样的错误信息，发生错误不可怕，但一定要在发生错误的时候仔细了解错误信息的意思。

```
Terminating app due to uncaught exception
'NSUnknownKeyException'
```

现在，错误信息中重要的信息是：该类中没有名为“iLoveThisGameImageView”的变量，很显然我们已经将这个变量的名称修改为diceImageView2了，但是为什么Xcode还会认为项目中会包含iLoveThisGameImageView变量呢?

在项目导航中打开Main.storyboard故事板文件，之前右侧骰子的ImageView是与iLoveThisGameImageView关联的，之后我们破坏了这种关联，虽然代码部分的灰色沟槽中已经变成了空心圆，但是设计界面中并不知道发生了变量名的变化。

为了验证，你可以在项目导航中右击Main.storyboard文件，通过Open As/Source Code查看其界面布局源码，在里面还可以找到iLoveThisGameImageView标记。

如何正确处理修改名称，删除某个IBOutlet或IBAction的情况呢?

首先，要在故事板中断开它们之间的关联，在需要修改的界面元素上右击鼠标，在弹出的浮动面板找到referencing Outlets部分中修改之前的Outlet变量名，本例为iLoveThisGameImageView。单击其右侧的叉子将其删除。此时如果再次查看Main.storyboard的XML格式文档，则不会再出现iLoveThisGameImageView相关的标签。

接下来需要将界面元素与新更名的Outlet变量重新建立关联，还是沿用之前的方法，从Image View到diceImageView2重新为其建立Outlet关联。稍微有些不同的是，按住鼠标右键并拖曳的终点要悬停在代码行中diceImageView2的上面，这时Xcode还会智能高亮显示diceImageView2。

最后可以看到diceImageView2代码行前面灰色沟槽中已经变成了实心圆，可以将鼠标悬浮在其上检查其关联的界面元素是否正确。

不管你是删除IBOutlet还是编辑IBOutlet或IBAction，强烈建议你在界面元素上右击鼠标，在弹出的面板中确认之前的关联是否被移除。

提示　只要在调试控制台中出现类似“this class is not key value coding-compliant for the key....”的字样，你的第一反应就是最近是否在故事板中改变了某些关联。

不知你是否注意到，当我们在模拟器中运行并测试应用程序的时候，调试控制台总是会自动打印出很多调试信息。但是绝大部分的信息对于开发者来说并没有什么实质性的帮助。接下来，我们会通过手动的方式关闭这些系统日志信息，让调试控制台只显示我们需要看到的信息。

实战：关闭控制台中系统日志的自动输出。

步骤1：在Xcode菜单中选择Product/Scheme/Edit Scheme...，在弹出的设置面板的左侧选中Run，在面板的右侧选择Arguments标签。

步骤2：在Environment Variables部分，单击其下方的+号，在Name栏中输入OS_ACTIVITY_MODE，在Value栏中输入disable，最后单击Close按钮，如图4-9所示。

图4-9　关闭控制台中系统日志的自动输出

再次构建并运行应用程序，此时的调试控制台不会再显示之前的系统日志信息。

4.4　使用Swift创建随机数

从本节开始就要进入编写代码的阶段了。首先需要创建变量，我们通过变量存储数据，在类声明的下面声明变量。

```
class ViewController: UIViewController {
  var randomDiceIndex1: Int = 0
```

通过var关键字创建一个叫作randomDiceIndex1的变量，在冒号的后面定义变量的类型为整型（Int），初始值设置为0。注意，变量只是一个数据的容器，让我们可以将数字或字符串放在里面。如果之后要修改变量的数值或文本内容，只需要简单将新值重新赋给该变量即可。

表4-1列出了各种常用的数据类型。

表4-1　常用的数据类型

在表格中整型（Int）用于存储数字，单精度（Float）和双精度（Double）用于存储小数，只是双精度会比单精度存储更多小数位的数。如果你需要用到带20个小数位的PI值，就需要声明一个双精度常量或变量。如果需要处理类似身高、体重的数据，单精度变量就完全可以胜任。Bool是布尔类型，它只能存储true或false两种值。String是字符串类型，除了可以存储文本内容以外，还可以将数值以字符串的形式存储在该类型中，但是字符串类型的数值无法进行计算。

回到Xcode项目，可以自己尝试着添加另外一个变量randomDiceIndex2。

```
class ViewController: UIViewController {

  var randomDiceIndex1: Int = 0
  var randomDiceIndex2: Int = 0
```

该项目为什么需要这两个变量呢？因为需要1到6之间的随机数来代表骰子的六个面。我们将会生成随机数来显示相应的骰子图片，这样用户就会看到骰子在屏幕上的变化了。

修改rollButtonPressed（_sender：UIButton）方法，如下所示：

```
@IBAction func rollButtonPressed(_ sender: UIButton) {
  randomDiceIndex1 = arc4random_uniform(6)
}
```

当用户单击“掷骰子”按钮以后便会执行rollButtonPressed（_sender：UIButton）方法中的代码。首先是通过arc4random_uniform函数生成0到5的随机数。arc4random_uniform函数被定义到Darwin.C.stdlib文件中，是基于C的UNIX函数。在编写代码的时候，我们往往要在代码文件中导入相关的代码库或框架。比如当前的ViewController类中，就导入了UIKit框架。

```
import UIKit

class ViewController: UIViewController {
```

当我们导入了UIKit（用户界面工具，User Interface Kit）框架后，该Swift文件就包含了所有与用户界面、视图、视图控制器相关的类和函数API。如果此时将ViewController.swift文件中的import UIKit代码行删除，Xcode编译器马上就会报出十几个错误，如图4-10所示。

图4-10　注释掉UIKit框架以后编译器报错

arc4random_uniform函数带一个正整型参数6，代表该函数会返回一个0到5之间的随机整数。目前，Xcode编译器会报一个错误，意思是不能将类型为UInt32的值分配给randomDiceIndex1。也就是说arc4random_uniform函数返回的值类型是UInt32，而我们所定义的randomDiceIndex1的类型为Int，两种类型不匹配，所以无法赋值。

UInt32和Int有什么不同呢？UInt32中的U代表无符号（Unsigned），UInt代表前边没有符号的整型，也就是从0开始的整数，UInt32代表的就是无符号32位的整数。而Int类型包括正整数、负整数和0。

解决上面的问题，我们需要将类型进行转换。修改之前的代码如下面这样：

```
@IBAction func rollButtonPressed(_ sender: UIButton) {
randomDiceIndex1 = Int(arc4random_uniform(6))
}
```

通过Int的初始化方法Int（），我们将arc4random_uniform函数返回的UInt32类型的随机数转换为Int类型，这样等号左右两边类型一致，编译器错误也就消失了。

接下来为randomDiceIndex2编写相应代码：

```
@IBAction func rollButtonPressed(_ sender: UIButton) {
  randomDiceIndex1 = Int(arc4random_uniform(6))
  randomDiceIndex2 = Int(arc4random_uniform(6))

  print(randomDiceIndex1)
}
```

为了验证用户在每次单击按钮后randomDiceIndex1变量是否存储了0到5之间的随机数，在方法的最后添加了一行打印语句print（），该函数会将参数值打印到调试控制台中。

构建并运行项目，多次单击“掷骰子”按钮，调试控制台中会打印出每次随机生成的randomDiceIndex1的值，效果如图4-11所示。

图4-11　在控制台中打印出生成的随机数

4.5 数据类型、常量、变量

在上一节中，我们创建了randomDiceIndex1和randomDiceIndex2两个变量，在本节中我们将会深入了解变量、常量以及它们之间的联系。

关闭之前的Dicee项目，在Xcode欢迎界面中单击Get started with a playground，Apple通过Playground帮助开发者以最简单的方式，实现各种想法的测试。Playground大大降低了我们学习Swift的门槛，因为它可以实时执行代码，立即将结果显示出来，并且还有各种交互功能。

在设置面板中将Name设置为Variables，Constants and Data Types，Platform设置为iOS，单击Next按钮，并将其保存到指定的位置。

在打开的Playground中，已经预载入了一个字符串类型的变量str，它的值为Hello，playground。将该行代码删除，然后输入下面的代码。

```
import UIKit

var myAge: Int = 38

myAge  = 39
```

在Swift中声明变量需要先使用var关键字，然后紧跟着的是变量名称，这里输入myAge，因为是驼峰命名，所以my是小写，Age是首字母大写。接下来是冒号跟变量类型，这里输入Int，最后是为myAge设置初始值，所以输入=38。如果要是在之后修改myAge的值，可以直接用等号进行赋值，而不需要再使用关键字var，也就是说var只在变量声明的时候使用一次。

在Playground窗口的左侧可以实时看到每行代码的值，一般情况下是变量的值。如果需要打印一些东西到控制台，也可以使用print

（myAge）将内容打印出来，如图4-12所示。

图4-12　Playground中的代码测试

接下来，我们需要声明另一个变量来存储人名，但是人的名字在一般情况下是不会变化的，所以可以创建一个常量来存储人名。创建常量的关键字是let。如果在声明了常量并对其赋值以后再次修改其值的话，Playground就会显示错误：不能对myName常量进行再赋值，如图4-13所示。

```
let myName: String = "Happy"
```

图4-13　修改常量值的时候编译器报错

实际上，我们可以把变量和常量当作存储数值的容器或者盒子，只不过当作变量的盒子是可以随时被打开，取出之前的东西，再重新放置其他的东西。而当作常量的盒子只能在第一次放好东西以后做封箱处理，不能再被拆开了。

变量和常量的用途不同，变量用于跟踪应用程序的状态变化，比如用户的等级变化，你的实时位置等。常量用于存储永不改变的数据，比如某个第三方应用的API Key或者一个URL链接地址。常量所占用内存的空间要小于变量，考虑到iPhone的优化，建议大家尽量使用常量。

在Playground中添加下面的代码：

```
let myAgeInTenYears = myAge + 10
//等号右边是两个整型数相加，并将结果赋值给左边的myAgeInTenYears常量中
```

上面的代码是两个整型数相加，最后将值赋给一个常量。在声明myAgeInTenYears常量的时候，我们并没有为它指明数据类型，因为Swift编译器会自动进行类型断言，也就是说等号右侧的结果是Int类型，则Swift就自动将左侧的新常量类型设置为Int。另外，两个字符串也可以进行相加运算，但是结果并不是它们相加的和，而是将字符串连接到一起。

```
let myFullName = myName + " Liu"
//myFullName的值为 Happy Liu
```

除了使用+号连接字符串以外，还可以通过反斜线+括号的方式。

```
let myFullName2 = "\(myName) Liu"
//myFullName2的值为 Happy Liu
```

我们使用“”来表示这个值是字符串类型，在字符串中如果包含（变量或常量）的话，则编译器会自动将其替换为相应变量或常量的值。

继续添加下面的代码：

```
let myDetails = "\(myName), \(myAge)"  //结果为: Happy, 39
```

这里，声明的myDetails是一个常量，因为赋值号右边的结果是字符串类型，所以通过类型断言，Swift设置该常量的类型为字符串。在右侧的字符串中，Swift还隐性地将整型变量myAge转换成字符串，并和myName混合在了一起。

关于数据类型还有一点需要大家清楚的是，当我们说到数据类型，其实你可以把它想象为下面这样一个儿童玩具，如图4-14所示。这个玩具的玩法非常简单，就是将几何体穿过孔洞而已，但如果是错误的孔洞你便无法将其填入其中。用它来比作变量和常量是最合适不过的

了，因为它们都是固定的类型。比如有一个字符串变量（玩具中的一个圆形孔洞），我们只能用它存储字符串类型的数据，如果是字符串ABC则没有任何问题。该字符串可以顺利地穿过“孔洞”，但如果将字符串换成了整型数39，则会报错不允许将其穿过孔洞。

图4-14　儿童玩具

所以，每一个变量，每一个常量都暗含着数据类型，该数据类型是在第一次被赋值的时候确定的，或者是在声明的时候被显性确定的。

接下来，我们再看看Swift其他几种基本的数据类型：

```
let wholeNumber: Int = 12  // 可以存储正整数、负整数和0
let text: String = "ABC"   // 字符串值需要使用双引号括起来

let bool: Bool = true      // 只能存储true、false两个值

let floatingPointNumber: Float = 1.3 // 存储小数，受限于位数，精度不高，仅精确到小数点后6位
let double: Double = 3.14159263345    // 用于科学计算，可以存储64位的数字，精准到小数点后15位
```

4.6 解决错误：“The Maximum Number of Provisioning Profiles Reached”

对于那些在iPhone真机上，使用免费的Apple ID账号测试运行应用程序的用户，可能会遇到：“已达到免费开发配置文件的最大数量的应用程序（The Maximum Number of Provisioning Profiles Reached）”的错误。

虽然Apple允许开发者免费在真机上测试应用程序，但是却把可以在iPhone设备上加载的应用数量限制在每周10个。通过一些测试和调查，发现Apple在引入这一新规则的时候会有Bug出现。许多开发者在将3个应用程序加载到他们的iPhone真机之后就会发生错误。Apple肯定会最终解决这个问题，但目前我们需要使用特殊的技巧来解决此问题。

最简单的解决方案是重新创建一个全新的Apple ID，并将这个账号添加到Xcode中，这样你就可以每周将10个应用程序载入到iPhone设备上。

如果上述方案不适合你，或者感觉太麻烦，那么你可以按照下面的步骤删除之前的一些应用程序，并确保当前应用程序可以装载到你的iPhone上。

实战：解决可能发生的“The Maximum Number of Provisioning Profiles Reached”错误。

步骤1：使用USB数据线将iPhone连接到Mac。

步骤2：在Xcode中打开Dicee项目，在菜单中选择Window/Devices and Simulators，如图4-15所示。

图4-15　选择Devices and Simulators

步骤3：在新窗口的边栏中选择你的设备，并删除右边的“I-Am-Happy”和“I-Am-Busy”。如图4-16所示。

图4-16　删除之前的项目

4.7 通过数组改变显示方式

在本章之前的部分中，我们已经创建了两个用于存储随机数的变量randomDiceIndex1和randomDiceIndex2，并编写了几行代码，通过arc4random_uniform（）函数生成随机数。

在本节，我们需要去改变Image View界面元素的属性，另外还会创建数组。

在Dicee项目中，我们通过两个Image View来显示两个骰子的面，并且还将这两个Image View通过IBOutlet的方式与代码建立了关联。虽然我们可以在通用工具区域的Attributes Inspector中，通过修改Image属性这种最直接的方式来改变骰子的面，但是我们真正需要的是在应用程序运行的时候，通过程序代码动态地修改Image View的Image属性。

在rollButtonPressed（_sender：UIButton）方法中的最后，添加下面的代码：

```
@IBAction func rollButtonPressed(_ sender: UIButton) {
  randomDiceIndex1 = Int(arc4random_uniform(6))
  randomDiceIndex2 = Int(arc4random_uniform(6))

  print(randomDiceIndex1)

  diceImageView1.image = UIImage(named: "dice2")
}
```

当用户单击“掷骰子”按钮并得到了两个随机数以后，我们首先修改diceImageView1的image属性值。实际上diceImageView1变量指向的就是故事板中用于显示第一个骰子的Image View界面元素，并且它有很多其他的属性，比如大小的frame属性，背景色的background属性等。

随后，我们将UIImage对象赋值给image属性，UIImage是另一种数据类型，通过它的初始化方法，会将Assets.xcassets中的图像加载进来。在项目导航中单击Assets.xcassets文件夹，可以看到之前导入进来的dice1至dice6图片，虽然这些图片在导入的时候都带有.png的扩展名，但是在代码中可以忽略图片的扩展名，直接使用文件名即可。

构建并运行项目，在模拟器中目前左侧的骰子还是1点的面，但是当单击“掷骰子”按钮以后，应用程序便会调用rollButtonPressed（_sender：UIButton）方法，在该方法中修改diceImageView1的image属性的代码会被执行。

接下来，我们需要让diceImageView1和diceImageView2根据我们的要求显示骰子的1至6的不同面。这6个不同的面是基于文件名dice1至dice6创建的，为了将随机数与图片文件名建立联系，我们要利用数组。添加下面的代码到之前声明变量的下方：

```
class ViewController: UIViewController {

  var randomDiceIndex1: Int = 0
  var randomDiceIndex2: Int = 0

  let diceArray =
["dice1","dice2","dice3","dice4","dice5","dice6"]
```

在编写代码的时候我们会经常看到（）、{}、[]、<>这四种符号，每一种符号都有其特殊的用途，所以千万不要混淆。创建数组需要使用中括号[]，数组中的每一个元素使用逗号分割。其实，数组就像是一个放鸡蛋的盒子，它所包含的东西都是同类型的，也就是说你不能在盒子中又放鸡蛋又放鞋子。类似，在Swift数组中你不能在一个数组中加入两种不同类型的数据，比如字符串和整型数。有关数组的另一件需要牢记的事情是：在现实生活中鸡蛋盒子里面的鸡蛋是从1开始算起的，而在Swift语言中数组的首个元素是从0开始索引的。

如何获取数组中的元素呢？最直接的方式是通过索引值，比如通过diceArray[0]可以获取到数组中第一个元素的值，以此类推。

在Dicee项目中，我们创建的第一个数组要包含的是骰子图像的名称，并且可以利用索引获取相应的图片。而通过arc4random_uniform（）函数生成的随机数也是从0至5，通过数组我们就可以获取到正确的骰子图片了。

修改rollButtonPressed（_sender：UIButton）方法如下面这样：

```
@IBAction func rollButtonPressed(_ sender: UIButton) {
  randomDiceIndex1 = Int(arc4random_uniform(6))
  randomDiceIndex2 = Int(arc4random_uniform(6))

  print(randomDiceIndex1)

  diceImageView1.image = UIImage(named: diceArray[1])
}
```

构建并运行应用程序，效果和之前的一样。在单击“掷骰子”按钮以后，左侧的骰子会显示2点的骰子面。

再次修改rollButtonPressed（_sender：UIButton）方法，构建并运行项目。

```
@IBAction func rollButtonPressed(_ sender: UIButton) {
  randomDiceIndex1 = Int(arc4random_uniform(6))
  randomDiceIndex2 = Int(arc4random_uniform(6))

  diceImageView1.image = UIImage(named:
diceArray[randomDiceIndex1])
  diceImageView2.image = UIImage(named:
diceArray[randomDiceIndex2])
}
```

通过生成两个随机数，我们会生成特定骰子面的图像，并最终通过两个diceImageView的image属性将其显示到屏幕上。

接下来，我们进一步完善Dicee项目，让它更加合理。

当我们打开应用程序的时候，往往希望第一眼看到的两个骰子面是随机出现的，而不是在每次开启后都固定在两个1点的面上。想要做到这点，我们只能在屏幕视图被载入以后通过代码来实现，viewDidLoad（）方法便是实现这一功能的地方。

将rollButtonPressed（_sender：UIButton）方法中的所有代码添加到viewDidLoad（）方法中，代码如下面这样：

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

  randomDiceIndex1 = Int(arc4random_uniform(6))
  randomDiceIndex2 = Int(arc4random_uniform(6))

  diceImageView1.image = UIImage(named:
diceArray[randomDiceIndex1])
  diceImageView2.image = UIImage(named:
diceArray[randomDiceIndex2])
}
```

构建运行项目，应用程序完美运行！

但是，从优化程度来说，这段代码并不是很优雅，因为我们要尽量避免DRY（Don’t Repeat Yourself，不要自我重复）情况出现。如果在项目中发现了重复的代码，你要尽量尝试将其组织到一个方法之中，这样做的好处就是可以尽可能避免出现Bug。

```
func updateDiceImages() {
  randomDiceIndex1 = Int(arc4random_uniform(6))
  randomDiceIndex2 = Int(arc4random_uniform(6))

  diceImageView1.image = UIImage(named:
diceArray[randomDiceIndex1])
  diceImageView2.image = UIImage(named:
diceArray[randomDiceIndex2])
}
```

在ViewController类中创建一个方法，首先使用func关键字，然后是能够实现所需功能的准确的方法名称，如通过updateDiceImages方法名称，我们可以很容易知道它的功能是用于更新骰子的面。在方法名称之后是传递进来的参数列表，因为本方法并不需要传递参数，所以直接使用小括号即可。在小括号的后面便是一对大括号，在大括号中的代码便是该方法所要实现的功能。

现在，我们已经在ViewController中创建了updateDiceImages（）方法，可以在该类中任何需要的地方调用它。

修改viewDidLoad（）方法如下面这样：

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

  updateDiceImages()
}
```

在输入代码的时候，Xcode会通过自动完成特性动态生成一个快捷输入列表，通过回车键可以快速完成输入。这样做的好处在于：一是在输入方法名、变量名的时候会节省输入的时间。二是可以有效避免输入错误的情况。在输入的时候，我们总是要遵循驼峰命名法则，但这样在输入大小写字母的时候就有很大可能会发生输入错误。

修改rollButtonPressed（_sender：UIButton）方法为下面这样：

```
@IBAction func rollButtonPressed(_ sender: UIButton) {
  updateDiceImages()
}
```

在本章中，我们提到了类、方法、函数、对象等词，这些都是和面向对象有关的专属词，在之后的章节中，会详细介绍它们之间的关系和不同。

构建并运行项目，测试运行是否正常。

4.8　为项目添加运动检测功能

我们已经完成了Dicee项目的基本功能，在单击“掷骰子”按钮以后就会改变两个骰子的面。我们还想实现一个非常Cool的特性，即通过iPhone的感应检测器检测用户是否在晃动他们的iPhone，然后再利用代码修改骰子的面。

如何实现感应检测呢？作为程序员不可能记住Swift语言中所有功能实现的方法和函数，因此在某些时候我们可以借助Xcode的帮助文档来解决当前的问题。

在Xcode菜单中选择Help/Developer Documentation，在边栏中选择Swift语言，然后在搜索栏中输入Motion Events。在帮助文档中，可以找到Responding to Motion Events部分，里面包含三个方法，单击每一个方法都可以查到它们的功能，如图4-17所示。

单击motionEnded（_：with：）方法，该方法会在感应检测结束的时候被调用执行。

图4-17　从帮助文档中搜索相关的方法

在ViewController类中创建一个新的方法：

```
override func motionEnded(_ motion: UIEventSubtype, with event:
UIEvent?) {
  updateDiceImages()
}
```

构建并运行该项目，如果是在iPhone真机上面，则可以通过晃动手机的方式改变骰子的面。如果是在模拟器中运行，则可以选择模拟器菜

单中的Hardware/Shake Gesture在模拟器中实现晃动，如图4-18所示。

图4-18　在模拟器中模拟晃动手机的操作

现在，整个Dicee项目已经完成，你可以在GitHub中搜索“liumingl/dicee”关键字来下载该项目源代码。

4.9　挑战：Swift数据类型、变量和数组

在之前的学习中，我们了解了如何将界面元素与代码进行IBOutlet或IBAction关联，如何通过代码修改Image View的image属性值，如何使用数组，如何创建方法等。接下来，你需要独立创建一个项目来巩固所学的知识。该项目涉及的技能都是之前接触过、学习过的，因此不用担心会出现无法完成的情况。

这次我们要完成的项目叫作魔力8号球，它是一个占卜类的小游戏。其玩法是：先将窗口朝下，同时向球问是非题，接着轻轻摇晃后再将窗口面转上，之后在球内的二十面体中会有一道答案浮现在窗口，如图4-19所示。

实战：创建魔力8号球游戏

步骤1：创建一个新的Xcode项目，在Xcode菜单中选择File/New/Project。选择Single View Application作为应用程序的模板，Product Name设置为Magic 8Ball。

步骤2：在GitHub中搜索“liumingl/Magic 8Ball”关键字，并从中下载Magic 8Ball Image Assets.zip文件。然后，将下载的zip文件解压缩。

步骤3：在项目导航选中Assets.xcassets文件，然后打开AppIcon，将之前解压缩的Magic 8Ball App Icon Images文件夹中的图片，按照要求拖曳到相应的Icon空槽中。

步骤4：从解压缩的文件夹中将ball相关的图片添加到Assets.xcassets文件夹中，如图4-20所示。

图4-19　魔力8号球玩具

图4-20　在Magic 8Ball项目中添加ball素材

步骤5：在上一个Dicee项目中，我们使用图片作为背景，这次直接修改视图的背景色。

打开Main.storyboard文件，修改控制器视图的Background属性，将颜色设置为RGB#28AAC0。但这不是绝对的，你也可以选择其他颜色作为背景色，如图4-21所示。

步骤6：从对象库中拖曳一个UILabel控件到视图顶部的位置，修改UILabel的内容为“请随意问我问题吧”。

再拖曳一个Image View到视图中央的位置，将其Content Mode属性设置为Aspect Fit，这样可以保证图像不会被拉伸变形。

再拖曳一个UIButton到视图的底部，将其标题修改为“请提问”，并设置合适的字体与字号，如图4-22所示。

图4-21　修改Magic 8Ball视图的背景色

图4-22　设计Magic 8Ball的用户界面

步骤7：将Xcode切换到助手编辑器模式，确保左侧窗口打开的是Main.storyboard，右侧窗口打开的是ViewController.swift文件。

为Image View创建IBOutlet关联，将name设置为imageView。为底部的按钮创建IBAction关联，将IBAction名称设置为askButtonPressed。

步骤8：打开ViewController.swift文件，在class声明的下方创建一个新的数组ballArray，该数组要包含所有15个与ball有关的图片。

在数组的下方再声明一个用于存储随机数的变量randomBallNumber，我们将通过该变量确定ball的图片。

在viewDidLoad（）方法中，利用arc4random_uniform（）函数为randomBallNumber变量赋值，让其为0至X的随机数。

还是在viewDidLoad（）方法中，设置imageView的image属性为UIImage类生成的对象，至于所显示的图像内容，则是依赖于randomBallNumber随机数所对应的数组中的图片文件名。

构建并运行项目，检查Image View中是否正常显示图片，如图4-23所示。

图4-23　测试Magic 8Ball应用的运行效果

步骤9：在步骤8中所添加的两行代码，一行是生成随机数，另一行是设置Image View的image属性。我们需要将它们放在一个全新的方法中。这样，不管是按钮被按下，还是用户摇晃手机时，都可以直接调用该方法。

创建一个叫作newBallImage（）的方法，将其放到askButtonPressed（）方法的下方。将步骤8中的两行代码剪切到新方法中。在askButtonPressed（）方法和viewDidLoad（）方法中调用newBallImage（）。

步骤10：添加motionEnded（_motion：UIEventSubtype，with event：UIEvent？）方法到ViewController类中，并且在该方法中调用newBallImage（）方法。

构建并运行项目，检测在晃动的时候是否会有效果。你可以在GitHub中搜索“liumingl/Magic 8Ball”关键字来下载该项目源代码。

第5章　Swift程序设计基础

通过之前的学习，即便是从来没有编程经验的人，都能够利用已掌握的知识和技能，独立编写代码并生成应用程序，最终将其上传到iPhone真机上运行。

在上一章中，我们了解了常量和变量，以及方法和函数的使用方法，并编写了Magic 8Ball应用程序。在本章中，我们将学习更深层次的有关编程的知识，以便帮助我们走得更远！

5.1　备注、打印语句和调试控制台

学习编程最好的方法就是实践，让我们重新开启Xcode，这时我们需要单击Get started with a playground，而不是之前一直使用的Create a new Xcode project。

Playground是苹果的一个帮助开发者实践想法的简单环境，它与视图、设计或应用无关。可以说它是一个学习以及验证程序员编写代码是否实现了真正想法的地方。

本节我们主要学习注释、打印和控制台，因此在Playground模板中选择iOS/Blank，单击Next按钮，然后将文件名设置为Comments, Printing and the Console，再将其保存到Mac的特定文件夹即可，如图5-1所示。

在自动生成的Playground文件中包含了下面样例代码：

```
//: Playground - noun: a place where people can play
import UIKit

var str = "Hello, playground"
```

图5-1　在Xcode中创建Playground文件

代码中第一行的意思是：Playground是一个可以让人们玩的地方，并且这行代码的颜色是绿色的，在Xcode IDE开发环境中，不同颜色的代码代表不同含义。绿色的代码意味着它是注释语句，Xcode编译器会忽略它，不会将它作为代码来处理。注释总是以两个斜线开始，如果删除两个斜线，则Xcode编译器就会报错，因为其后面的“代码”让编译器根本无法理解。

双斜线只能让一行描述成为注释，如果有多行注释的话，就需要使用/*作为开头，以*/作为结尾，其中的所有行都是注释语句。

```
//: Playground - noun: a place where people can play
/*
    这是一个多行的注释语句
*/
import UIKit

var str = "Hello, playground"
```

接下来是import语句，当前所导入的是用户界面工具（User Interface Kit）框架，这是一个由苹果编写的代码库，它会帮助开发者更快速地创建有用的代码。例如，我们不用自己编写一个如何绘制按钮的代码，或者自己编写一个生成随机数的代码，这些代码都被打包进了UIKit中。只要我们导入了UIKit，就可以使用该框架的所有东西。

接下来的代码是一个名为str的字符串变量，它包含一个字符串：Hello，playground。这里并不需要它，所以我们把该行代码删除。

接下来介绍的是print语句，我们之前使用过它，但是用得不多。先创建一个叫作monsterHealth的变量，如果你设计了一款游戏，里面有恶魔、僵尸和人类，你需要记录恶魔的血量，以及快要接近死亡的血量。

```
import UIKit

var monsterHealth =19
```

如果你在Playground中如代码所示输入19的话，Swift编译器将会报错，如图5-2所示。这是为什么呢？Swift语言是一种非常优雅的语言，你可以把它理解为一种完美的对称。当我们使用操作符的时候，例如=、+、-、/或*号的时候，你必须让它两边都有一个空格的间隔。等号左边有空格，等号的右边就必须要有空格，如果少了一边，

Swift就会对这种非对称格式报错。因此，要不就两边都有空格，要不就两边都不留空格。这里还是强烈建议将操作符两边留出一个空格，既美观又让代码显得非常优雅！

图5-2　为变量monsterHealth变量赋值时编译器报错

当我们将变量monsterHealth的值设置为19以后，Playground右侧窗口中相应的位置会显示变量的值。这非常有用，否则我们很难追踪执行完每一句后变量会发生怎样的变化。

继续在Playground中添加下面的代码：

```
import UIKit

var monsterHealth = 19
monsterHealth = monsterHealth + 31
monsterHealth = monsterHealth / 2
```

这里，我们修改了两次monsterHealth的值，在右侧的窗口中则会依次显示每次修改monsterHealth后的值，如图5-3所示。

图5-3　两次修改变量monsterHealth的值

除了在Playground中通过右侧窗口查看变量的值以外，还可以通过print（）函数打印出常量、变量或表达式的值。在Playground中键入print（）函数：

```
import UIKit

var monsterHealth =19
monsterHealth = monsterHealth + 31
```

```
print(monsterHealth)
```

print（）会将参数的值打印到控制台之中，控制台位于整个窗口的底部，帮助我们调试或找出代码中所出现的问题。如果在窗口底部没有看到控制台的话，可以通过窗口顶部右上角的一组按钮将其切换出来。如图5-4所示。

图5-4　切换出调试控制台

5.2 Swift函数：Part 1-简单函数

在第4章中，我们已经简单了解了什么是常量和变量，以及它们与数据类型之间的关系。在本节我们将学习编程语言中另一个最基础的部分——函数。让我们重新创建一个Playground文件，文件名为Functions，并删除Playground自动生成的所有代码。

```
func nameOfFunction() {

}
```

这里我们只需记住创建函数的语法是使用关键字func，接下来是函数的名称，包含参数的括号，还有一个左大括号。当我们键入回车以后，Xcode会自动为我们补全右大括号，Xcode的自动完成功能可以帮助我们解决绝大部分的输入Bug问题。

函数可以为我们做些什么呢？简单来说，它可以将一大段的指令"打包"到一起，完成一个相对独立的功能。比如我们想将大象放到冰箱里面，就可以利用下面的代码：

```
func 将大象放入冰箱() {
  打开冰箱门()
  将大象放入冰箱()
  关闭冰箱门()
}
```

这样，我们就可以将所有的指令按照顺序放入上面的这个函数之中，当调用该函数的时候，它就会依次执行这些指令。

在Playground中，针对函数的创建，如果我们在一行之中只键入func关键字的话，Xcode编译器会报错：在函数声明时需要一个标识符，如图5-5所示。Swift语言在业界是非常领先的，当我们还未成功创建

函数的时候，它会告诉我们——你的函数不完整。因此，在我们还未键入完整的函数或变量的时候，并不用担心报出的类似的错误。但是，当我们完成整行或整段代码，感觉没有问题的时候，如果出现报错信息，就需要注意了。

图5-5　在函数声明时需要一个标识符

在Playground中完成下面的代码：

```
func getMilk() {
    // 该函数目前不需要实现任何的代码
}
```

对于函数名称，我们要使用驼峰命名法，帮助我们快速识别长函数或变量名称的用途。在函数名称的后面是一对小括号，括号里面用于放置传递进函数需要的参数，如果不需要传递则直接键入（）即可。在括号的后面是大括号，Xcode会自动补全整个大括号，我们可以在其中键入函数所运行的代码。

在之前的项目中，我们已经创建并使用过IBAction函数，在Playground中则可以通过函数名调用函数。在之前代码的下方键入get，此时Xcode会启用自动完成特性，在列表中可以找到getMilk（）函数，选中它并按回车键，该函数会在Playground中自动补全。

```
func getMilk() {
    // 该函数目前不需要实现任何的代码
}

// 调用getMilk()函数
getMilk()
```

目前的代码并没有任何的效果，因为在getMilk（）函数中没有任何代码。在后面的学习中，我们会把这个项目想象为一个家务机器人，让它帮助我们到商店购买一些生活必需品，比如去购买牛奶。在函数里面通过打印语句模拟机器人的活动，继续完成下面的代码：

```
// 创建 getMilk() 函数
func getMilk() {
   print("去门口的小卖店")
   print("买2瓶牛奶")
   print("支付13.20元")
   print("回家")
}

// 调用getMilk()函数
getMilk()
```

在getMilk（）函数中，我们通过四行print语句打印出机器人购买牛奶的过程。在键入每行print语句的时候，都可以发现控制台中有信息的变化。需要说明的是，当我们在函数中键入代码的时候，并不会真正执行这些代码，而只有在最后一行调用函数的时候才会执行函数内部的代码。删除最后一行对getMilk（）的调用，控制台中不会再显示任何信息，或者多次调用getMilk（）函数。可以看到控制台中会显示更多被打印的信息。

5.3 Swift函数：Part 2-函数的输入

接下来，我们要为getMilk（）函数添加参数输入特性。目前，我们是通过硬写入数值的方式要求机器人购买2瓶牛奶。选中之前getMilk（）函数的所有代码，然后通过Command+/快捷方式将其全部注释掉。

注意　当注释掉getMilk（）函数以后，Xcode会报错，这是因为Swift并没有找到getMilk（）函数。

将下面的代码添加到注释代码的下面：

```
func getMilk(howManyMilkCartons: Int) {

}
```

新创建的函数带有一个整型参数，参数名为howManyMilkCartons。虽然创建了新的getMilk（）函数，但是编译器依然报错。只不过这次的错误指在了调用getMilk（）函数行。意思是说：getMilk（）函数丢失了参数howManyMilkCartons。

这里，我们先删除之前的getMilk（）函数，重新输入get，在自动完成列表中则出现了新的带参数的函数，按回车键后，带参数函数便出现在Playground中，并且光标会停留在参数类型Int上面，提示我们要输入一个整型数，这里输入4，代码如下面这样：

```
func getMilk(howManyMilkCartons: Int) {

}

// 调用getMilk()函数
getMilk(howManyMilkCartons: 4)
```

在调用带参数函数的时候，需要先输入函数名称，然后是括号以及括号中的参数。修改代码如下面这样：

```
func getMilk(howManyMilkCartons: Int) {
    print("去门口的小卖店")
    print("买 \(howManyMilkCartons) 瓶牛奶")
    print("支付13.20元")
    print("回家")
}

// 调用getMilk()函数
getMilk(howManyMilkCartons: 4)
```

在上面的代码中，我们将4赋值给参数howManyMilkCartons，在调用函数的时候，其内部就包含一个howManyMilkCartons常量，并且它的值为4。所以在控制台中会显示买4瓶牛奶的信息。

目前这段代码有一个Bug，不管参数设置为多少盒牛奶，支付价格总是13.20元。因此在函数中，需要加入计算牛奶总价的代码。修改代码如下面这样：

```
func getMilk(howManyMilkCartons: Int) {
    print("去门口的小卖店")
    print("买 \(howManyMilkCartons) 瓶牛奶")

    let priceToPay = howManyMilkCartons * 7

    print("支付 \(priceToPay) 元")
    print("回家")
}

// 调用getMilk()函数
getMilk(howManyMilkCartons: 4)
```

当我们创建好priceToPay常量以后，会在状态窗口中看到一个警告（warning），如图5-6所示，单击它以后会在窗口的左侧出现相应的

警告说明：priceToPay常量自从初始化以来，还没有在其他地方使用过。Swift编译器会检测出那些从来没有使用过的常量与变量，提示程序员是否忘记使用了，或者是尽快删除之，以避免浪费宝贵的资源。因为只是警告，所以代码还是可以正常运行。

图5-6　在Playground中出现的编译器警告提示

在打印支付费用的时候，使用"\（）"方式在字符串中呈现常量或变量的值。在控制台中显示的结果为：

```
去门口的小卖店
买 4 瓶牛奶
支付 28 元
回家
```

5.4 Swift函数：Part 3-函数的输出

在本节的学习中，我们会将函数设置为既带参数又带返回值的形式。还是之前的家政机器人项目，选中之前的getMilk（howManyMilkCartons：）函数，通过Command+/快捷键将代码全部注释掉。

然后将刚刚注释的代码再完全复制一遍到其下方，重新选中之前被注释掉的全部代码，再次使用Command+/快捷键取消代码的注释，并将代码修改为下面这样：

```
//func getMilk(howManyMilkCartons: Int) {
//  print("去门口的小卖店")
//  print("买 \(howManyMilkCartons) 瓶牛奶")
//
//  let priceToPay = howManyMilkCartons * 7
//
//  print("支付 \(priceToPay) 元")
//  print("回家")
//}

func getMilk(howManyMilkCartons: Int) -> Int {
  print("去门口的小卖店")
  print("买 \(howManyMilkCartons) 瓶牛奶")

  let priceToPay = howManyMilkCartons * 7

  print("支付 \(priceToPay) 元")
  print("回家")
}


// 调用getMilk()函数
getMilk(howManyMilkCartons: 4)
```

如果我们想创建一个带返回值的函数，需要使用返回标记，也就是在参数小括号的右边添加“->”标记，然后再添加返回值的数据类型。对于该函数，它的返回值类型还是整型。

目前，编译器会显示有一个错误：丢失了函数预期的Int类型的返回值（missing return in a function expected to return'Int'）。这是因为函数中还没有使用return语句将返回值输出到函数之外。

修改之前的代码如下面这样：

```
func getMilk(howManyMilkCartons: Int,
howMuchMoneyRobotWasGiven: Int) -> Int {
  print("去门口的小卖店")
  print("买 \(howManyMilkCartons) 瓶牛奶")

  let priceToPay = howManyMilkCartons * 7

  print("支付 \(priceToPay) 元")
  print("回家")

  let change = howMuchMoneyRobotWasGiven - priceToPay

  return change
}

var amountOfChange = getMilk(howManyMilkCartons: 4,
howMuchMoneyRobotWasGiven: 30)

print("你好主人，这里是找回的 \(amountOfChange) 元钱。")
```

这里首先修改了getMilk（）函数，让它带有2个参数，第一个参数是告诉机器人购买牛奶的数量，第二个参数是给机器人用于购买牛奶的钱数。而函数的返回值便是找回多少钱。需要注意的是，我们在调用getMilk（howManyMilkCartons：Int，howMuchMoneyRobotWas Given：Int）->Int函数的时候，也要相应修改为带有2个参数的形式，为了节省时间你可以直接删除之前的代码，再次输入get后通过Xcode自动完成特性生成新的函数调用代码。其次是创建了新的变量

amountOfChange，并将getMilk（howManyMilkCartons：Int，howMuch MoneyRobotWasGiven：Int）->Int函数的返回值赋值给它。最后通过print语句打印出应该找回的钱数。

5.5 Swift中的条件语句（IF/ELSE）

在现实世界中，我们往往要面对很多选择。在程序设计语言中，我们同样会面对各种选择的问题，只不过在这里我们管它叫作条件语句或选择语句。

让我们用一个简单的姓氏评分的例子来深入了解选择。如图5-7所示，只要我们输入了姓氏和大名以后，便可以得到名字的评分。

图5-7　有趣的姓氏评分应用

打开Playground并新建一个Empty文档，文件名叫作Conditional Statements。修改代码如下面这样：

```
import UIKit

func nameCalculator (yourFirstName : String, yourLastName :
String) -> Int {

  // 生成一个 0 到 100 之间的随机数
  let nameScore = Int(arc4random_uniform(101))

  return nameScore
}

print(nameCalculator(yourFirstName: "铭", yourLastName: "刘"))
```

我们首先创建了一个函数nameCalculator（），该函数包含2个参数和1个返回值。yourFirstName是人的名字，字符串类型；yourLastName是人的姓氏，也是字符串类型。返回值是整型类型，我们会将生成的随机数作为返回值。此时arc4random_uniform（）函数的返回值是UInt32类型，所以需要使用Int（）将其转换为整型再赋值给nameScore常量，并将其作为返回值。

另外，我们可以直接修改函数的返回值类型为UInt32，从而让编译器错误消失。

```
import UIKit

func nameCalculator (yourFirstName : String, yourLastName :
String) -> UInt32 {
  // 生成一个 0 到 100 之间的随机数
  let nameScore = arc4random_uniform(101)

  return nameScore
}
```

这并不是一个真正的姓名测试程序，只是想说明如何使用条件语句。如果评分值在80以上则代表完美，否则就是一般。接下来，我们将使用if语句检查nameScore是否超过80，如果超过则返回字符串"你的名字很完美！"，否则返回字符串"你的名字比较一般"。因为返回的是字符串值，所以将函数的返回值类型修改为String，并删除之前在函数最后的return语句。

```
import UIKit

func nameCalculator (yourFirstName : String, yourLastName :
String) -> String {

  // 生成一个 0 到 100 之间的随机数
  let nameScore = arc4random_uniform(101)

    if nameScore > 80 {
        return "你的名字评分是 \(nameScore)，很完美！"
    }else {
        return "你的名字评分是 \(nameScore)，比较一般。"
    }
}

print(nameCalculator(yourFirstName: "铭", yourLastName: "刘"))
```

在控制台中，可以看到测试的结果，如图5-8所示。

图5-8　在Playground中显示的姓氏测试结果

目前，程序只是针对是否超过80进行判断，我们希望在40到80之间再添加一种新的信息输出，修改之前的代码为下面这样：

```
import UIKit

func nameCalculator (yourFirstName : String, yourLastName :
String) -> String {

  // 生成一个 0 到 100 之间的随机数
  let nameScore = arc4random_uniform(101)

    if nameScore > 80 {
        return "你的名字评分是 \(nameScore)，很完美！"
    }else if nameScore > 40 && nameScore <= 80 {
        return "你的名字评分是 \(nameScore)，还不错！"
    }else {
        return "你的名字评分是 \(nameScore)，比较一般。"
    }
}

print(nameCalculator(yourFirstName: "铭", yourLastName: "刘"))
```

在最后一个else语句的上面添加else if关键字，nameScore的值如果是在40到80之间则会执行其中的代码。&&代表并，||代表或，！代表非，这与其他程序设计语言一致。

单击控制台左上角的蓝色三角图标可以重新执行当前的代码，如图5-9所示。

图5-9　重新测试程序的结果

使用条件语句最重要的一点就是检测机制一定要覆盖全部的情况，如果某些条件被我们忽略掉，就会有Bug出现。另外，在条件分支中程序只要运行到return就会跳出当前的分支，而不管该分支的后面是否还有其他未运行的代码。

5.6　挑战：在Playgrounds中制作人体体重指数计算器

本节将会带领大家制作一个体重指数（The Body Mass Index，BMI）计算器。体重指数是用来量化一个人的体重以及解释他们的身体组成的量度。它被定义为质量（Kg为单位）除以高度（m为单位）的平方。

在这个程序中，我们需要完成两件事情：

·创建一个函数，把人的体重和身高作为参数传递进函数，在函数的最后返回体重指数的值。

·用一些测试值调用函数，并将结果打印到控制台。

体重指数的计算公式为：

对于整个程序的逻辑，我们需要实现下面几点：

·如果体重指数大于25，则使用打印语句告诉用户超重。

·否则，如果体重指数在18.5～25之间，告诉用户体重正常。

·最后，如果他们的体重指数低于18.5，告诉用户他们体重过轻。

你可以先自己尝试着完成这个程序的代码，然后再对照下面的样子对比一下，如图5-10所示。

图5-10　人体体重指数计算器

5.7 Swift中的循环语句

成为程序员的一个特质就是表面上看似极为懒散，但是在遇到代码问题的时候又有非常良好的心态。程序员总是在努力争取如何用不重复的代码来完成现有的工作。在学习本节的内容之前，让我们先创建一个全新的Playground文件，然后删除“var str=”这行代码。让我们先来看下最基础的Swift循环语句是什么样子的。

```
let arrayOfNumbers = [1, 5, 2, 3, 10, 22,32]

for number in arrayOfNumbers {
  print(number)
}
```

在代码中，我们首先创建一个整型数组，其中包含了一些整型数，后面，我们会通过循环算出这些数的总和。在Swift语言中最常用的循环语句是for循环，其中第一个关键字就是for，并且在输入的时候，Xcode会将for显示为不同的颜色。然后就是要创建一个常量，用来循环存储arrayOfNumbers数组中的每一个元素值。之后的in arrayOfNumbers意味着循环数组中的每一个元素，直到读出数组的最后一个元素之后结束，并把元素的值赋给常量number。在本例中，一共要循环执行7次，每次都会在控制台中打印数组元素的值。

除了使用循环获取数组中的每个元素以外，我们还可以通过索引来得到数组中指定元素的值，例如let number=arrayOfNumbers[0]。

接下来，我们要计算出数组所有元素值的和。

```
let arrayOfNumbers = [1, 5, 2, 3, 10, 22,32]

var sum = 0

for number in arrayOfNumbers {
  sum += number // 等同于 sum = sum + number
```

```
  print(sum)
}

print(sum)
```

首先需要初始化一个变量sum，设置它的初始值为0。当进入第一次循环的时候，number的值为1，sum经过加运算以后变成了1。当进入第二次循环的时候，number为5，sum经过加运算以后变成了6。以此类推，在7次循环结束以后，变量sum的值被打印到控制台中。

循环除了可以依次遍历数组的每个元素以外，还可以通过另一种方式进行有序循环，修改for语句为下面这样：

```
for number in 1...10 {
  print(number)
}
```

通过1...10这种书写方式，将会执行10次循环，number的值会从1变为10。另外，如果将其修改为1..<10的话，则会循环9次，number的值将会从1变为9。

再次修改循环语句为下面这样：

```
for number in 1...10 where number % 2 == 0 {
  print(number)
}
```

在循环中我们添加了新的关键字where，意思是只有当number的值能被2整除的时候才会执行循环语句，因此循环体也会执行5次。

5.8 在程序中使用循环

在了解了循环的基本知识以后，让我们来尝试着生成一首歌曲的歌词。

在美国和加拿大有一首传统的倒数歌曲叫作——99瓶啤酒，它是一首20世纪中期的民歌。在长途旅行的时候非常受欢迎，因为它有固定的格式，便于记忆，可以花很长的时间来唱，所以特别适合儿童在长途巴士旅行中歌唱。具体的歌词形式如下：

```
99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the
wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down and pass it around, 97 bottles of beer on the
wall.

97 bottles of beer on the wall, 97 bottles of beer.
Take one down and pass it around, 96 bottles of beer on the
wall.

……

3 bottles of beer on the wall, 3 bottles of beer.
Take one down and pass it around, 2 bottles of beer on the
wall.

2 bottles of beer on the wall, 2 bottles of beer.
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, no more bottles of beer on
the wall.

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the
wall.
```

让我们先来完成循环的基本部分，然后再进行细节方面的修改。

```
import UIKit

func beerSong() -> String {
  var lyrics: String = ""

  for number in 1...5 {
    let newLine: String = "\n\(number) bottle of beer on the
wall, \(number) bottle of beer.
\nTake one down and pass it around, \(number - 1) bottles of
beer on the wall.\n"
    lyrics += newLine
  }

  lyrics += "\nNo more bottles of beer on the wall, no more
bottles of beer.
\nGo to the store and buy some more, 99 bottles of beer on the
wall.\n"

  return lyrics
}

print(beerSong())
```

在上面的代码中，通过for循环我们生成了从1到5的歌词，并且将每一段歌词都添加到了lyrics字符串变量之中，直到循环结束没有啤酒后，再将其打印到控制台。在字符串中，我们使用了转义字符\n，它代表文本内容中的换行，如果Swift编译器在字符串中看到\n就会自动进行一次换行操作。

在真正的歌词中，啤酒瓶的数量是从99向下依次递减的，而在代码中则是增加的。可以通过下面的方式来解决。

```
for number in 1...5 {
  let newLine: String = "\n\(6 - number) bottle of beer on the
wall, \(6 - number) bottle of
beer. \nTake one down and pass it around, \(6 - number - 1)
bottles of beer on the wall.\n"
```

```
  lyrics += newLine
}
```

因为当前是从1到5的循环，所以在newLine中，通过（6-number）的方式生成5到1的字符，解决了当前的问题。但是，这种做法非常不实用。如果将1...5修改为1...40，1...70，1...100的话，则每次调整都会影响到newLine字符串中的（X-number）的算式，这是不现实的。

有人可能会想到能不能通过99...1的方式来解决呢？如果你在Playground中尝试使用这种方法，则不会有任何的输出，因为前面的数字大于后面的数字，编译器会报错。

我们可以通过下面的方法来解决：

```
for number in (1...99).reversed() {
  let newLine: String = "\n\(number) bottle of beer on the
wall, \(number) bottle of beer. \nTake one down and pass it
around, \(number - 1) bottles of beer on the wall.\n"
  lyrics += newLine
}
```

在Swift语言中，（1...99）实际上是一个对象，它的类型为Range。Range类型有一个方法叫作reversed（），通过该方法可以得到1～99的反向范围，也就是99～1，如图5-11所示。通过上面的方式，我们可以随意设置歌曲循环的数量。

图5-11　Range类型值的反向输出

接下来，我们为函数添加一个参数，用于决定啤酒的数量。

```
import UIKit

func beerSong(withThisManyBottles: Int) -> String {
  var lyrics: String = ""
```

```
  for number in (1...withThisManyBottles).reversed() {
    let newLine: String = "\n\(number) bottle of beer on the
wall, \(number) bottle of beer.
\nTake one down and pass it around, \(number - 1) bottles of
beer on the wall.\n"
    lyrics += newLine
  }

  lyrics += "\nNo more bottles of beer on the wall, no more
bottles of beer. \nGo to the storeand buy some more, 99 bottles
of beer on the wall.\n"

  return lyrics
}

print(beerSong(withThisManyBottles: 23))
```

在上面的代码中，为beerSong（withThisManyBottles：Int）->String函数添加了一个Int类型的参数，并且将for语句中的1...99修改为1...withThisManyBottles，这样就会通过传递进来的参数来确定循环的次数。

在函数体内部我们通过参数名来动态设置循环次数，但在Swift语言中，我们还可以通过再定义一个内部参数名来对参数进行扩展。为什么会需要一个内部参数名呢？当我们调用beerSong（withThisManyBottles：）函数的时候，调用者可以清楚地知道该函数的功能以及参数所代表的意义。但是当我们在函数体内部使用参数的时候，并不需要调用者知道内部参数的含义，只要该函数的编写者清楚就可以了。因此，我们可以在函数内部给它起一个不同的名字以区别函数内部和外部的情况。要想做到这点，只要在参数名的后面再添加另一个参数名即可，代码如下所示。

```
import UIKit

func beerSong(withThisManyBottles totalNumberOfBottles: Int) ->
String {
  var lyrics: String = ""
```

```
  for number in (1...totalNumberOfBottles).reversed() {
    let newLine: String = "\n\(number) bottle of beer on the
wall, \(number) bottle of beer. \nTake one down and pass it
around, \(number - 1) bottles of beer on the wall.\n"
    lyrics += newLine
  }

  lyrics += "\nNo more bottles of beer on the wall, no more
bottles of beer. \nGo to the store and buy some more, 99
bottles of beer on the wall.\n"

  return lyrics
}

print(beerSong(withThisManyBottles: 23))
```

在withThisManyBottles参数名的后面添加了totalNumberOfBottles作为函数的内部参数名，因此在函数内部我们一律使用totalNumberOfBottles。你可以把Swift这样的机制理解为withThisManyBottles参数是给调用者看的，让他们知道参数的意思。totalNumberOfBottles是给函数编写者自己用的，使代码更加清晰，可读性更强。

有时你可能会看到外部参数名被下划线（_）替代，这就意味着在调用函数的时候不需要提供参数名称，在函数名称已经足够描述参数的时候，这个方式非常有用。例如下面代码：

```
func beerSong(_ totalNumberOfBottles: Int) -> String {
    // 略去实现代码
}
beerSong(23)
```

为了最终生成完美的歌词，我们将单独处理1瓶啤酒的情况。在最后一次循环的时候，控制台打印出来的信息为：

```
1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, 0 bottles of beer on the
```

```
wall.
```

但实际上最后1瓶的歌词为：

```
1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, no more bottles of beer on
the wall.
```

利用if语句修改之前的代码：

```
import UIKit

func beerSong(withThisManyBottles totalNumberOfBottles: Int) ->
String {
  var lyrics: String = ""

  for number in (1...totalNumberOfBottles).reversed() {
    let newLine: String
    if number == 1 {
      newLine = "\n\(number) bottle of beer on the wall, \
(number) bottle of beer. \nTake one down and pass it around, no
more bottles of beer on the wall.\n"
    }else {
      newLine = "\n\(number) bottle of beer on the wall, \
(number) bottle of beer. \nTake one down and pass it around, \
(number - 1) bottles of beer on the wall.\n"
    }

    lyrics += newLine
  }

  lyrics += "\nNo more bottles of beer on the wall, no more
bottles of beer. \nGo to the store and buy some more, 99
bottles of beer on the wall.\n"

  return lyrics
}

print(beerSong(withThisManyBottles: 23))
```

在for循环中，我们通过if语句来判断number的值是否为1，进而再设置newLine的文字信息。需要注意的是，在之前的for循环里面，我们在声明newLine常量的时候直接将字符串赋值给它，但如果将它放在if语句中，则会因为生存期的问题报错，也就是说在一个大括号中声明的常量或者变量，其生存的范围也就在这个大扩号之内，出了大括号它也就不存在了。

5.9 挑战：脑筋急转弯

在这一节中，我们想通过循环生成一个指定长度的斐波那契数列。斐波那契数列是由0和1开始，之后就是由之前的两数相加而得出。前面几个斐波那契系数是：0，1，1，2，3，5，8，13，21，34。斐波那契数列在现实生活中也是很有用的，如果你搜索有关Fibonacci Patterns的内容，会查到很多与其相关的事物存在，例如向日葵的花盘，如图5-12所示。

图5-12　各种斐波那契数列的应用

修改Playground文件中的代码如下面这样：

```
/*
 FIBONACCI NUMBERS
 0,1,1,2,3,5,8,……
*/

import UIKit

func fibonacci(until n: Int) {
  print(0)
  print(1)

  var num1 = 0
  var num2 = 1

  for iteration in 0...n {
    let num = num1 + num2
    print(num)

    num1 = num2
    num2 = num
  }
}

fibonacci(until: 5)
```

创建的fibonacci（until n：Int）函数带有一个整型参数，代表所生成数列元素的个数。num1和num2分别代表数列中第一个和第二个数，之后就开始通过循环来生成后面的数列元素。常量num代表每次循环所生成的新数，在循环体中我们总是将num1与num2的和赋值给它。最后，为了可以在下一次计算出新的数值，需要将之前的num2赋值给num1，将新计算出来的数（num）赋值给num2，循环体重新开始执行。如果此时将until参数的值修改为20，则输出结果应该是下面这样，如图5-13所示。

图5-13　循环20次的各种斐波那契数列

目前，该项目中还存在着Bug需要解决。参数until的数值代表我们想生成斐波那契数列的元素个数，但是现在的情况并不是这样。因为在函数的开始就已经打印出了0和1两个元素，所以在循环体中我们一共要生成n-2个元素，所以将for语句修改为for iteration in 0...n-3即可。注意n-3是因为循环从0开始。

当前的Swift编译器还有一个警告：常量iteration从来没有使用过，可以考虑将其替换为下划线或者移除它（Immutable value‘iteration’was never used；consider replacing with‘_’or removing it）。因为在循环体中从来就没有使用过常量iteration，可以简单将其替换为下划线（_）即可。

第6章　利用iOS API制作音乐应用

本章我们将会制作一个全新的应用，它叫作xylophone。在这个应用中，会有一个漂亮的图标。单击进入以后会看到七个不同的琴键，利用七种不同的声音，我们可以创建简单的曲子。本章的目的不是去学习更多编程方面的知识，而是通过xylophone应用让大家了解如何使用苹果的帮助文档，Stack Overflow以及专业工具帮助我们找出使用新功能的方法，最终达到让iPhone演奏音乐的目的。

本章涉及的代码可能有些是你不熟悉的，不用担心，因为在后面的学习中会向大家做详细的介绍。

本章旨在教你如何成为一个自力更生的程序员。因为在未来的工作中，你更多的是通过自己的努力来完成属于你自己的项目，其中有很多功能不可能在一本书或几本书中全部涉及，并且随着iOS SDK版本的不断改进和完善，我们更多的是需要通过苹果的官方文档找出解决问题的方法和答案。

6.1　使用故事板中的Tags

在GitHub中下载“Xylophone”项目的源代码，该项目代码压缩包包含了项目的初始代码和完成代码，这里请大家打开初始代码文件夹——Xylophone-Start。

打开Xylophone项目以后，在TARGETS的General中将Bundle Identifier的标识修改为自己的名字，如图6-1所示。

单击项目导航中的Main.storyboard以打开故事板，在视图中可以看到7个不同颜色的琴键，这7个琴键是由7个按钮组成，前文介绍过如何将按钮拖曳到视图中，所以这里不再赘述。但是，为了可以让用户界面完美地呈现在各种尺寸的iOS设备屏幕上，在当前项目中使用了iOS的自动布局特性，相关方面的知识我们会在后面的学习中详细介绍。

图6-1　修改Xylophone项目的Bundle Identifier

将Xcode切换到助手编辑器模式，为了获得更多编辑空间，可以暂时关闭Document Outline窗口，如图6-2所示。

图6-2　Xylophone的初始化项目

与当前视图对应的控制器是ViewController.swift文件中的ViewController类，如果右侧窗口中没有显示该文件代码的话，可以通过该窗口顶部的快捷通道快速定位ViewController.swift文件，如图6-3所示。

图6-3　通过快捷通道快速定位ViewController.swift文件

当用户单击琴键的时候，应用程序要发出相应的声音，所以接下来要为视图中的7个按钮建立IBAction关联。

在第一个红色的琴键上按住鼠标右键，并拖曳其到notePressed（_sender：）方法上面，如图6-4所示。

图6-4　为红色琴键与notePressed（_sender：）方法建立IBAction关联

除了上面的方法以外，我们还可以通过Inspector窗口进行IBAction关联以及查看现有的关联。

选中第一个红色琴键，然后单击Connections Inspector图标，在Sent Events部分中可以看到红色琴键按钮的Touch Up Inside动作对应了ViewController类的notePressed方法。这也就意味着，当用户单击红色琴键的时候会执行notePressed（_sender：）方法，如图6-5所示。

图6-5　在Connections Inspector中查看琴键的IBAction关联

接着选中橘色琴键，可以发现该按钮的Touch Up Inside动作并没有关联任何的IBAction方法，只要按住Touch Up Inside右侧的圆圈，并将其拖曳到notePressed（_sender：）方法上，也可以完成IBAction方法的关联，如图6-6所示。

与之前按住鼠标右键并拖曳的方法不同，使用Connections方法不仅可以建立关联，而且还可以查看当前UI元素的关联信息。

图6-6　通过Connections Inspector建立琴键与代码的IBAction关联

在建立好7个琴键与同一个IBAction方法的关联以后，你应该清楚的是，不管用户按下哪一个琴键，都会执行notePressed（_sender：）方法。这是因为虽然琴键不同，但实现的效果都是让程序出声，只不过是播放的音效不同而已。接下来，我们需要解决的是如何在方法中分辨用户到底按的哪一个琴键。

选中视图中的任一琴键，在Attributes Inspector中找到View部分的Tag属性，通过Tag属性我们就可以分辨出激活IBAction方法的到底是哪一个按钮，如图6-7所示。

图6-7　通过Tag属性分辨用户单击的按钮

单击第一个红色琴键，将其Tag属性设置为1。以此类推，将之后的琴键Tag属性分别设置为2至7。这样，每个琴键都有一个唯一的Tag值。

在notePressed（_sender：）方法中添加下面的代码：

```
@IBAction func notePressed(_ sender: UIButton) {
  print(sender.tag)
}
```

该方法带有一个参数sender，它的类型是UIButton，代表当前用户与之发生交互动作的那个按钮（琴键）。如果单击的是第一个红色琴键，则sender就是那个红色琴键的按钮对象，如果单击的是橘色琴键，则sender就是橘色按钮的对象。然后用点（.）操作符访问sender对象的tag属性。

构建并运行项目，在模拟器中单击不同的琴键，可以看到控制台中打印出相应琴键的tag值。

6.2　学会使用Stack Overflow和Apple Documentation

本节我们的重点不是如何理解程序代码或Swift语法，而是更加宽泛一些的内容。通过本节的学习希望大家可以掌握：在程序开发过程中，当需要实现某些功能的时候，可以通过什么方式去了解、掌握以及真正实现该功能。例如，我们创建的应用程序可能需要用到照相功能，但是在没有学习过任何相关知识的情况下，如何才能实现呢？另外，iOS系统如此强大，也不可能向大家介绍实现所有功能、方法以及API的调用方法，也相信大家不会将所有期望都寄托在某一本书或几本书之上。本节主要是教大家如何通过网络社区，从一名菜鸟变成老炮。

在Xylophone项目中，一个显而易见的需求是当用户单击琴键的时候，需要播放相应的音效。

实战：在网络社区中搜索如何播放音效。

步骤1：在浏览器中输入www.stackoverflow.com网址进入Stack Overflow。它是一个程序设计领域的问答网站。该网站允许注册用户提出或回答问题，还可以对已有问题或答案进行加分、扣分或修改操作，条件是用户达到一定的“声望值”，如图6-8所示。

图6-8　Stack Overflow网站

提示　非常遗憾的是，目前在国内还没有像Stack Overflow这样一个技术类问答网站。因此我们只能在不断提升自身编程水平的同时，还要不断提升自己的英文水平。

步骤2：在Stack Overflow的搜索栏中输入：playing sound in swift，就可以看到与此问题相关的列表。一般情况下，我们在技术开发中所遇到的绝大部分问题都能在这里找到答案。在搜索条目列表的左侧有

一个投票（votes）项，这意味着其他人对该提问的关注程度，从而也反应出这是一个不错的问题。其下方的答案（answers）则代表解决方法的数量，如果答案是白底绿框则代表该问题虽然有人回答，但是还没有获得提问者的最终确认，如图6-9所示。

所以要判定某个提问是否真正得到了解决方案，一定要选择那些高投票数和高答案数的条目。

图6-9　关于提问和解答的数量表示

步骤3：单击搜索列表中的第一个条目（Q：Creating and playing a sound in Swift），如图6-10所示。可以了解到这个提问者是在按下按钮时想要快速创建和播放声音，他知道在Objective-C语言中实现的方法，但不知道如何在Swift语言中实现。

图6-10　关于Creating and playing a sound in swift问题的解答

步骤4：向下滚动浏览器，可以在Answers部分的第一条回答找到解决问题的方法，如图6-11所示。回答者展示了一段游戏相关的代码，在该代码中包含播放音效的代码。

首先是import SpriteKit，在游戏项目中经常会导入该框架库。import AVFoundation才是我们真正需要的框架库，它是Audio和Video的基础库，所以在xylophone项目中需要导入该库。

图6-11　关于Creating and playing a sound in swift的解答

在xylophone项目的ViewController类中导入AVFoundation库。

```
import UIKit
import AVFoundation
```

回到Stack Overflow，复制里面的两行代码并将其粘贴到xylophone项目的ViewController类中。

```
class ViewController: UIViewController{

  // 从项目文件夹中获取声音文件的路径
  var coinSound = NSURL(fileURLWithPath:
NSBundle.mainBundle().pathForResource
("coin", ofType: "wav"))

  // 创建新的音频播放器
  var audioPlayer = AVAudioPlayer()

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

  @IBAction func notePressed(_ sender: UIButton) {
    print(sender.tag)
  }
}
```

步骤5：将代码中的coinSound变量修改为xylophoneSound，将pathForResource（）函数中的第一个参数修改为note1。此时编译器会报错：NSBundle已经被更名为Bundle（NSBundle has been renamed to Bundle），如图6-12所示。因为Swift还是一个非常年轻的语言，从发布到现在也只有三年的时间，并且经历了很多变量名称或方法调用的改变。

当我们看到这样的错误，会有两种不同的解决方式：一是在Xcode编译器中单击Fix链接，由Xcode编译器自动帮你修复所出现的问题；二是可以利用Stack Overflow重新搜索play sound in swift 3，并找到相关问题的答案。

图6-12　通过Xcode编译器修复问题

提示　因为Xcode 9默认使用的是Swift 4版本，我们之前在Stack Overflow上查到的答案则是2.0版本，它们之间的代码并不是完全兼容的，所以直接搜索最新版本的代码显得更容易一些。

虽然当前使用的是Swift 4，但它兼容Swift 3的绝大部分语法和功能，所以在搜索的时候可以使用Swift 3关键字。之所以不使用Swift 4作为关键字，是因为对于刚刚出现的4.0版本，相关的问题和解决问题的答案还不会太多。

在重新搜索的答案列表中，选择根据投票数排列，可以看到目前列在首位的是一个投票数在100的问题，如图6-13所示。

图6-13　通过投票数排序最佳答案

单击“A：How to play a sound using Swift？”进入该提问，我们马上会看到各种Swift语言版本的播放音效的解决方案，如图6-14所示。在开头还会提示你必须使用AVFoundation框架库，可见回答者还真是暖男一位。

图6-14　在不同Swift语言版本中播放音效的方法

步骤6：因为当前使用的是Xcode 9IDE开发环境，所以找到“Swift 4（iOS 11compatible）”部分的代码。将其复制到ViewController.swift中，并进一步修改代码：

```
import UIKit
import AVFoundation

class ViewController: UIViewController{
```

```
  var player: AVAudioPlayer?

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

  @IBAction func notePressed(_ sender: UIButton) {
    guard let url = Bundle.main.url(forResource: "note1",
withExtension: "wav") else { return }

    do {
      try
AVAudioSession.sharedInstance().setCategory(AVAudioSessionCateg
oryPlayback)
      try AVAudioSession.sharedInstance().setActive(true)

      /* 下面一行代码是针对iOS 11的播放器，并且需要修改为与url相应的文件
类型 */
      player = try AVAudioPlayer(contentsOf: url, fileTypeHint:
AVFileType.wav.rawValue)

      guard let player = player else { return }

      player.play()

    } catch let error {
      print(error.localizedDescription)
    }
  }
}
```

在ViewController类中，我们声明了一个AVAudioPlayer类型的变量player，它用于播放url所指定的音效文件。

在notePressed（_sender：UIButton）方法中，首先创建了一个url常量，该常量从项目资源文件夹中指定了note1.wav文件的位置。这里使用了guard关键字，它是做什么用的呢？可以利用Stack Overflow快速找到答案。

实战：在Stack Overflow中查找guard关键字的功能。

步骤1：在Stack Overflow中搜索guard swift。

步骤2：在列表中找到“Q：Swift’s guard keyword”条目，如图6-15所示。

图6-15　搜索guard swift的结果

步骤3：在该条目中，可以从提问人所问的内容中找到需要的答案：苹果从Swift 2引入了guard关键字，可以用它来确保各种数据已经配置完成。同时他也提出了相关的问题：使用guard进行判断是否比使用if语句更加方便和简单？

在下面的答案列表中，可以找到一些线索：与if语句一样，guard基于表达式的布尔值来执行语句。与if不同，guard语句只在条件不满足时才运行。你可以把guard想象成一个断言，可以让你优雅地退出，而不是崩溃。

以xylophone项目为例，在guard语句中，首先根据音效文件名创建了一个url，如果该url存在，则会成功赋值给url常量，代码继续向下运行。如果赋值失败，则直接运行else语句中的内容——优雅地退出，执行return语句。

在url被成功赋值以后，我们通过do/catch语句块进行音效的播放操作，也就是说它是在我们尝试着（try）做（do）某事的时候，捕获（catch）错误的一种方法。如果在执行代码的时候激发了一个错误，该错误就会被catch语句块捕获到，然后将其打印到控制台。

构建并运行项目，当单击琴键的时候就会发出木琴的音效，只不过目前的音效都是同一个声音——note1。很显然，因为在notePressed（_sender：UIButton）方法中，我们只创建了note1.wav的url。本节我们的目的并不是编写代码，而是学习找到解决问题的方法，在后面的章节中会逐步完善相关功能。

在Stack Overflow中，解决问题的办法不仅仅只有一种。如果你仔细寻找甚至可以发现与咱们xylophone项目直接相关又直接可用的代码。

在Stack Overflow中重新使用playing sound in swift关键字搜索问题，在列表中再次单击“Q：Creating and playing a sound in swift”问题，并向下找到包含下面代码的答案：

```
import UIKit
import AVFoundation

class ViewController: UIViewController, AVAudioPlayerDelegate{
  var audioPlayer : AVAudioPlayer!

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

  @IBAction func notePressed(_ sender: UIButton) {

    let soundURL = Bundle.main.url(forResource: "note\
(sender.tag)", withExtension: "wav")

    do {
      audioPlayer = try AVAudioPlayer(contentsOf: soundURL!)
    }
    catch {
      print(error)
    }

    audioPlayer.play()
  }
}
```

如果仿照上面代码的样子，结合之前的项目代码稍加修改，就可以很快实现敲击不同琴键发出不同音效的功能。

修改前：

```
@IBAction func notePressed(_ sender: UIButton) {
  guard let url = Bundle.main.url(forResource: "note1",
withExtension: "wav")
else { return }
```

修改后：

```
@IBAction func notePressed(_ sender: UIButton) {
  guard let url = Bundle.main.url(forResource: "note\
(sender.tag)", withExtension: "wav") else { return }
```

构建并运行项目，测试单击不同的琴键，是否会发出不同的音效。

6.3 利用AVFoundation播放声音

通过之前的实战，我们已经能够通过按钮的tag属性来确定用户单击的是哪个琴键，接下来就是要真正实现音效的播放。在项目导航可以看到Sound Files文件夹中包含了木琴的7个不同的音阶声效文件。单击每一个文件都可以听到音效的声音，你也可以将其替换为自己喜欢的木琴音效。

实战：利用AVFoundation播放声音。

步骤1：回到ViewController.swift文件，删除上节课复制到notePressed（_sender：UIButton）方法中的全部代码。

步骤2：在import UIKit的下方，添加一行代码import AVFoundation。

AVFoundation框架库允许我们在程序中使用音频可视化组件（Audio Visual Component），在这里我们只需要让它来播放音效。

步骤3：将类声明那行代码修改为下面这样：

```
class ViewController: UIViewController, AVAudioPlayerDelegate {
  ……
}
```

步骤4：使用var关键字声明新的变量audioPlayer，类型为AVAudioPlayer，请在声明的最后加上一个叹号。

```
class ViewController: UIViewController, AVAudioPlayerDelegate {

  var audioPlayer: AVAudioPlayer!
  ……
}
```

提示　请不用担心前两步中所添加的AVAudioPlayerDelegate和变量声明中的感叹号（！）是做什么用的，我们将会在后面的章节中做详细介绍。目前我们只需要让这个Xylophone项目发出声音就足够了。

步骤5：在notePressed（_sender：UIButton）方法中添加下面的代码：

```
@IBAction func notePressed(_ sender: UIButton) {

let soundURL = Bundle.main.url(forResource: "note1",
withExtension: "wav")

do {
    audioPlayer = try AVAudioPlayer(contentsOf: soundURL!)
}catch {
    print(error)
}
}
```

这里使用let关键字创建一个常量soundURL，然后将note1.wav文件的URL通过Bundle.main.url（）方法赋值给它。其中参数forResource的值为音效的文件名（note1），with-Extension的值为音效文件的扩展名（wav）。

接下来，我们需要试着（try）载入音效到audioPlayer。当我们键入AVAudioPlayer（contentsOf：URL）方法的时候，可以发现在自动完成面板的列表中，该方法的结尾有throws关键字，代表它能够抛出异常，如图6-16所示。因此，在这里我们需要先使用do关键字创建一个代码块，该块儿中会包含可抛出异常的方法。如果你想让那些可throws的方法在必要时刻抛出异常，必须使用try关键字。

图6-16　带有throws特性的方法

AVAudioPlayer（contentsOf：URL）方法的参数contentsOf代表播放音效的URL，直接将soundURL赋值给它，并注意在soundURL的最后输入一个感叹号（！）。do代码块中仅有一行代码，尝试（try）着让其播放指定URL的音效，如果播放成功则继续向下运行，否则就会执行catch代码块中的指令。

在catch代码块中，只是简单地打印error信息，我们将会在后面章节详细介绍如何使用catch语句。

对于do代码块中的指令，我们只有通过这样的方式来确定问题出现在哪里以及出现了什么样的错误。通过catch中的error参数，可以发现具体的错误描述，并通过print语句将信息打印到控制台，进而找到出现Bug的原因。

步骤6：在notePressed（_sender：UIButton）方法的最后，添加下面一行代码：

```
@IBAction func notePressed(_ sender: UIButton) {

……

  audioPlayer.play()
}
```

通过audioPlayer的play（）方法，我们让App发出声音。

构建并运行项目，单击琴键可以听到单一的木琴声。

6.4　Swift 4中的错误捕获——Do、Catch和Try

若要在notePressed（_sender：UIButton）方法中实现异常捕获，我们会使用do/catch/try关键字，如果你刚刚接触它的话有可能会晕，这节就让我们来梳理一下实现异常捕获的方法。

对于Xylophone项目来说，当我们尝试载入音效文件到audioPlayer播放器的时候有可能会出现问题。这一过程就像是现实生活中，我们往CD机中插入CD光盘一样。

做一个实验，请将"audioPlayer=try AVAudioPlayer（contentsOf：soundURL！）"移到do代码块之外，然后删除try关键字，最后将do/catch代码块注释掉，如图6-17所示。

图6-17　不使用do/catch/try方法处理throws方法

此时编译器会报错："调用了throw"，但是并没有使用try关键字和处理错误（Call can throw，but it is not marked with'try'and the error is not handled），这是什么意思呢?

在Xcode中按住Command键并单击AVAudioPlayer（contentsOf：URL）方法，此时会弹出一个跳转面板，单击其中的Jump to Definition选项，进入AVFoundation框架库之中，如图6-18所示。

图6-18　通过Command-Click进入到类的定义文件

在AVAudioPlayer类的声明部分，可以找到下面的方法声明代码：

```
/* all data must be in the form of an audio file understood by
CoreAudio */
    public init(contentsOf url: URL) throws
```

在方法声明的最后有一个throws关键字，这意味着在我们执行该方法的时候，如果出现了错误，则会抛出一个异常。比如你的note1.wav文件是一个无效的音频文件，则会抛出这个错误告诉你出现了什么问题。

但是，如果我们只是单纯地在audioPlayer前面添加try关键字，Swift并不会捕获到异常。苹果是在Swift 2.0版本的时候引入了do/catch语句，do语句块中包含的是做什么，catch语句块中包含的是如果出现异常，我们要如何处理。

将代码修改为下面的样子：

```
@IBAction func notePressed(_ sender: UIButton) {

 let soundURL = Bundle.main.url(forResource: "note1",
withExtension: "mp3")

  do {
    try audioPlayer = AVAudioPlayer(contentsOf: soundURL!)
  }catch {
    print(error)
  }

  audioPlayer.play()
}
```

为了验证throws的功效，我们在上面的代码中将url（forResource：，withExtension：）方法中的withExtension参数值修改为mp3。然后在项目导航中，将note1文件的扩展名从wav修改为mp3。显然我们错误地定义了音频文件的格式，并且在程序中要求播放这个格式错误的音频文件。

构建并运行项目，程序在audioPlayer.play（）代码行处报错。如图6-19所示。

图6-19　throws方法在运行中出现错误的处理

提示　利用被打印到控制台中的错误代码，可以找出发生问题的原因。在浏览器中进入osstatus.com网站，该网站专门用于提供快速查找Apple API错误的服务，在搜索栏中粘贴之前的错误代码并搜索，如图6-20所示。

图6-20　在osstatus.com中搜索到的错误原因

在错误列表项的描述列中可以发现，引发这个错误的原因一般是文件错误，或者该文件不是指定类型的音频文件实例，再或者文件本身就不是音频文件。

因为我们故意将wav格式的音频文件扩展名修改为mp3格式，所以代码抛出这样的错误也就不足为奇。让我们先将note1.mp3改回到note1.wav，同样需要修改的还有url（forResource：，withExtension：）方法。

另外，如果我们百分之百地确定soundURL所提供的音效文件没有问题，也可以直接抛开do/catch语句，就利用try！关键字执行播放指令：

```
@IBAction func notePressed(_ sender: UIButton) {
  let soundURL = Bundle.main.url(forResource: "note1",
withExtension: "wav")

  try! audioPlayer = AVAudioPlayer(contentsOf: soundURL!)

  audioPlayer.play()
}
```

这样的代码虽然可以实现之前的播放功能，但是前提就是一定要百分百确认参数值没有问题，可以说这是一个大胆的做法，一个没有退路的做法。除非万不得已，一定不要使用这种方式。另外，如果大家在开发的时候见到这样的情况也不至于晕圈。

最后，请让我们将代码还原到之前的do/catch方式。

6.5 创建一个播放声音的方法

在notePressed（_sender：UIButton）方法的下面创建一个新的方法：

```
@IBAction func notePressed(_ sender: UIButton) {
  playSound()
}

func playSound() {

  let soundURL = Bundle.main.url(forResource: "note1",
withExtension: "wav")

  do {
    try audioPlayer = AVAudioPlayer(contentsOf: soundURL!)
  }catch {
    print(error)
  }

  audioPlayer.play()
}
```

这里使用func关键字来创建函数，函数名称要体现出对功能的描述，所以命名为playSound，暂时不带任何参数。将notePressed（_sender：UIButton）方法中的所有代码复制到playSound（）方法里面。

接下来，在notePressed（_sender：UIButton）中，利用sender参数（指向用户单击的那个琴键）的tag属性，分析出用户单击了第几个按钮，通过这个值生成与之相对应的欲载入的音效文件。我们一共有7个不同的音效文件，这里使用数组组织和管理它们。

```
class ViewController: UIViewController, AVAudioPlayerDelegate {
  var audioPlayer: AVAudioPlayer!

  let soundArray =
```

```
["note1","note2","note3","note4","note5","note6","note7"]

  ……
  @IBAction func notePressed(_ sender: UIButton) {

    var selectedSoundFileName: String = soundArray[sender.tag]
    print(selectedSoundFileName)

    playSound()
  }
  ……
```

构建并运行项目，单击1～6号琴键后会在控制台依次打印出note2～note7，当单击最下面的琴键时，应用程序会崩溃终止运行。如图6-21所示。

图6-21　单击最后一个琴键的时候应用崩溃

从图6-21中可以发现，该错误是致命的。我们在之前数组的学习中了解到它是从0开始索引的，所以第一个元素的索引值为0，第二个为1，……，以此类推。当用户单击最下方的琴键时，我们要获取到soundArray[7]的值，但数组中最后一个元素的索引值仅为6，这样就造成了超出数组索引范围的致命错误，使应用程序崩溃。

要想解决这个问题，最直接的方法就是在视图中，将各按钮的tag值从1至7修改为0至6。看似这是一个最简单直接的有效方法，但是其中暗藏着Bug。当我们从对象库中拖曳UI元素到视图上的时候，其默认的tag值都是0。假设新添加一个按钮对象到视图之中，则它的tag值就是0，再将其与notePressed（_sender：UIButton）建立IBAction关联，这样就会造成两个按钮的tag值都是0的情况，大大增加了出现Bug的几率。

另一种解决方案是从索引值入手，修改代码如下面这样：

```
var selectedSoundFileName: String = soundArray[sender.tag - 1]
```

方法简单有效，再也不会遇到之前超出数组索引范围的致命错误了。

目前在ViewController类中还有一个警告：selectedSoundFileName没有发生变化，考虑是否将变量改为常量（Variable'selectedSoundFileName'was never mutated；consider changing to'let'constant）。因为在后面会有对该变量的需求，所以在这里暂时不用理会该警告。

6.6 让App每次播放不同的声音

现在，我们的程序仅能在控制台中打印出与所按琴键相对应的音效文件名称，下一步则是需要播放相应的声音。

在notePressed（_sender：UIButton）方法中，我们声明了selectedSoundFileName变量，并将相应的音效文件名赋值给它。那在playSound（）方法中，我们是否可以直接使用该变量呢?

修改playSound（）方法中的代码为：

```
let soundURL = Bundle.main.url(forResource:
selectedSoundFileName, withExtension: "wav")
```

当我们在Xcode中输入变量或方法的时候，在自动完成窗口中并没有出现selectedSoundFileName变量，这是因为自动完成窗口只会显示当前位置可用的内容。没有selectedSoundFileName的原因在于它不在作用域范围之内。作用域是个很有意思的东西，当我们在一对大括号的内部，例如函数或者方法，声明一个变量，它的可用范围就在这对大括号的内部。超出这个范围，这个变量就不存在了。

如果我们将变量声明在类的大括号里面，则它的作用域就是整个类，也就意味着类中的任何方法都可以使用该变量。而声明在方法中的变量，则只能在该方法中使用。

有关作用域的问题我们下一节会做详细介绍，现在让我们了解一下关于全局变量和局部变量的概念。在notePressed（_sender：UIButton）方法中声明的selectedSoundFileName变量就是局部变量。在ViewController类中声明的audioPlayer就是全局变量，即在类中的任何方法中都可以访问到它。

让我们先在ViewController类的顶部声明一个selectedSoundFileName变量，它现在是全局变量。

```
class ViewController: UIViewController, AVAudioPlayerDelegate {

  var audioPlayer: AVAudioPlayer!
  var selectedSoundFileName: String = ""
  ……
```

在声明全局变量的时候，我们将空字符串（""）赋值给selectedSoundFileName变量，之后在需要修改的地方直接重新赋值即可。修改notePressed（_sender：UIButton）和playSound（）方法。

```
@IBAction func notePressed(_ sender: UIButton) {

  selectedSoundFileName = soundArray[sender.tag - 1]
  ……
}

func playSound() {
  let soundURL = Bundle.main.url(forResource:
selectedSoundFileName, withExtension: "wav")
  ……
}
```

现在selectedSoundFileName的值完全是由与用户交互的琴键决定。构建并运行项目，单击不同的琴键，就可以听到曼妙的琴声了。

让我们回顾一下之前的所有代码，当单击木琴上的第一个（红色）琴键的时候，会调用IBAction方法notePressed（_sender：UIButton），通过该方法的参数sender（用户所单击的那个红色按钮对象）的tag属性得到琴键的标识（当前的值为1，是之前在故事板中设置好的），按钮对象具有很多属性，包括backgroundColor和tag等。利用此tag值和提前定义好的soundArray数组，就可以得到欲播放音效的文件名，因为数组的索引是从0开始的，所以要让tag减去1。此时selectedSoundFileName的值变成了note1。在执行到playSound（）方

法的时候，将url（）方法的forResource参数值设置为selectedSoundFileName，也就是字符串note1。接下来就是生成url和利用该url让audioPlayer播放器载入该文件，当成功载入以后就让audioPlayer播放该音效。

6.7 程序中的“作用域”

这一节让我们来说说有关作用域的事情，请你先想象一幅带有围墙的苹果园的场景，然后再回到我们的Xylophone项目中，打开ViewController.swift文件。

当前selectedSoundFileName变量声明的位置位于ViewController的内部，以及所有方法的外部。为什么要放在这？这背后有哪些意义？

如果把selectedSoundFileName变量的声明放在IBAction方法notePressed（_sender：UIButton）的内部，就代表它是在该方法的内部创建的。这时Swift编译器就会报1个错误和1个警告。其中警告是：变量selectedSoundFileName虽然有写入，但是并没有被读取过；错误是：使用了1个未知的变量selectedSoundFileName，这意味着Xcode不知道你所引用的selectedSoundFileName变量的存在。这就是作用域在起作用，它负责变量的能见度。

如果在函数大括号的内部创建一个变量，它就是局部变量，它只能在函数大括号内部可见，可以在函数内部自由地使用它。但问题是如果我们想要在其他方法中访问该变量，比如playSound（）或viewDidLoad（）方法中，就会出现错误，因为局部变量只能在函数内部局部可见。

将selectedSoundFileName变量的声明放在类的大括号之中，即所有的函数体外部，这就相当于和所有的方法具有同样的层级，它就在所有的方法之中可见。现在Swift编译器的报错已经消失了，我们可以随意地访问和使用它。例如在notePressed（_sender：UIButton）方法中为其赋值，在playSound（）方法中读取它的值。

这种方法虽然可行但并不是一种优化的方法，还记得之前提到的苹果园的事情吗？在你家的果园中有一间房子，另外还有一棵苹果树，如图6-22所示。当苹果树在果园的围墙以内，在它成熟的时候只有你可

以采摘这些苹果。但是当苹果树在围墙外面时，你的邻居、过路人都可以摘苹果，如图6-23所示。

图6-22　仅对你可见的苹果树

图6-23　对其他人都可见的苹果树

把你果园里面的苹果想象为Swift中的局部变量，它仅仅对你可见。但是如果苹果树移到了围墙之外，任何人都可以访问它，其中也包括了你自己。你可以控制你自己，但是你并不能控制你的邻居和过路人。

在playSound（）方法中添加下面一行代码：

```
func playSound() {

  // 强行指定selectedSoundFileName的值为note2
  selectedSoundFileName = "note2"

  let soundURL = Bundle.main.url(forResource:
selectedSoundFileName, withExtension: "wav")

  do {
    try audioPlayer = AVAudioPlayer(contentsOf: soundURL!)
  }catch {
    print(error)
  }

  audioPlayer.play()
}
```

此时构建并运行应用程序，不管按哪个琴键都会发出一样的声音。如果把selectedSoundFileName变量比作苹果树的话，现在它种在了围墙之外，任何人都可以操作它。然而在playSound（）方法的开头却执

行了selectedSoundFileName="note2"代码，这行代码就相当于邻居的采摘行为，我们无法控制围墙外面的苹果树不被别人采摘。

删除之前的全局变量selectedSoundFileName，修改notePressed（_sender：UIButton）和playSound（）方法为下面这样：

```
@IBAction func notePressed(_ sender: UIButton) {

  playSound(soundFileName: soundArray[sender.tag - 1])
}

func playSound(soundFileName: String) {

  let soundURL = Bundle.main.url(forResource: soundFileName,
withExtension: "wav")

  do {
    try audioPlayer = AVAudioPlayer(contentsOf: soundURL!)
  }catch {
    print(error)
  }

  audioPlayer.play()
}
```

我们为playSound（）方法添加了一个参数soundFileName，该参数只在方法内部有效。在notePressed（_sender：UIButton）方法中我们将数组元素的字符串值作为playSound（）方法的参数。整个调用过程中，没有泄露任何变量供其他人来访问和修改。

构建并运行项目，应用程序完美运行！

第7章　使用Model-View-Controller设计模式制作小测验App

本章我们会创建一个稍微复杂点儿的用于测验的应用，并通过构建该应用学习有关Model-View-Controller设计模式（简称MVC设计模式）的相关知识。

当我们启动应用程序以后，将会依次出现13道问题，你可以使用是或否按钮来回答问题。在屏幕的底部还会有一个进度条，用于显示当前答题的数量和所得的分数，在用户单击按钮后会在屏幕上出现相应的确认视图，当回答完所有问题以后，会提示用户是否需要重新作答，如图7-1所示。

图7-1　问答题测试项目

总而言之，我们可以轻而易举地将该应用程序项目变成属于自己的商业应用并上架到App Store上面。

7.1　初始化Quizzler项目

在GitHub中下载本项目的源代码，该项目代码压缩包中包含了项目的初始代码，这里请大家打开初始代码文件夹——Quizzler-Start。

当项目打开以后，第一件事就是在设置面板的General标签中修改Bundle Identifier标识，将.Quizzler之前的部分修改为自己拥有的反向域名。如果计划将这个应用运行在物理真机上，则需要设定Team选项。

另外，我们还需要找到Deployment Info部分的Device Orientation选项，确认只勾选了Portrait，为了让应用具有很好的用户体验，所以让该应用不支持除纵向Home在下的其他方向。如果你要锁定应用支持指定屏幕方向的话，这是最简单的一种方法。

提示　在项目导航中选择顶部左侧为蓝色图标的Quizzler，再单击其右侧的TARGETS/Quizzler。

另外需要注意的是，我们已经将状态栏风格（Status Bar Style）设置为Light，如图7-2所示。状态栏是应用顶部的部分，我们在这里可以检查电池的电量，信号强弱，时间和日期等信息。因为应用的背景色是浅棕色，所以使用默认的黑色状态栏并不会很好看。透明的Light风格可以让界面稍显舒服一些。如果勾选Hide status bar则在Quizzler运行的时候会隐藏状态栏。

图7-2　项目的Status Bar Style设置

除了在General标签中修改项目的各种参数以外，我们还可以直接在Quizzler的配置文件中进行手动设置。在项目导航选中Supporting Files/Info.plist文件，如图7-3所示。

图7-3　Info.plist文件中的项目配置

提示　选择文件的时候请勿执行双击操作，因为在项目导航中双击文件会单独打开一个新的窗口，如果同时开启很多窗口的话会让你顿感晕圈，所以只要保持单击就好。

plist是Property List的缩写，每个Xcode项目都会自动创建这个文件。它会存储应用程序的配置信息，在应用程序运行的时候也是如此。这些信息会以键/值对的形式存储，类似于字典，键是属性名称，值是实际的配置内容。

我们可以通过修改Info.plist文件达到修改项目配置的目的。右侧面板的列表一共包括三项内容：键（Key）、类型（Type）和值（Value）。键就是我们要配置的项目名称，比如Status bar style。值是真正的配置信息，比如Status bar style的值就为UIStatusBarStyleLightContent，另外我们可以通过值右侧的下拉列表框修改它的值。类型实际上指的就是值的类型，常见的类型有String、Boolean、Array和Dictionary。plist配置表中的条目实际上都是键/值配对的，我们管这叫作字典（Dictionary），因为一个键名对应一个特定值，就像实际生活中的字典一样，每个字都有相应的解释。

实际上Info.plist文件中的根（root）条目就是一个字典类型，该字典的内部都是一条条键/值配对信息。要想添加一个新的条目，只需单击根条目Information Property List右侧的加号即可。

单击根条目Information Property List右侧的加号，然后在弹出窗口中找到View controller-based status bar appearance，它的值是Boolean类型，默认值为NO。

在ViewController.swift中我们可以通过代码来修改状态栏的风格，让它成为我们需要的样子，甚至于在一个应用的不同视图控制器中都可以单独设置不同的状态栏风格。但是在有些时候，我们往往要让状态栏在整个应用中有一个固定的风格。当Info.plist中的View controller-

based status bar appearance的值为YES的时候，则视图控制器对状态栏的风格设置优先级高于应用程序的设置；为NO则以应用程序的设置为准，视图控制器对状态栏风格的修改方法无效，是根本不会被调用的。

除了在项目导航中选择Info.plist文件直接进行项目配置的修改以外，在与之前General标签同级的Info标签中也可以做同样的事情。

构建并运行项目，目前应用程序的状态栏就是透明且使用白色作为前景色，看起来自然、漂亮，如图7-4所示。

让我们将注意力集中到项目导航中，与之前的文件组织结构不同，在Quizzler项目中，所有文件都被放到了三个主要文件夹之中进行管理，它们是Controller、Model和View。这正与我们本章所要介绍的Model-View-Controller设计模式有着很紧密的关系。

展开View文件夹并选中Main.storyboard文件，在Interface Builder中看下用户界面布局效果，如图7-5所示。

在视图的上半部是问题的显示区域，它的下面则为“是/否”一绿一红两个按钮，在视图的底部是进度条，随着答题数量的增加，它的宽度也随之变长，还有就是通过左下角的标签来显示“答题数/总题数”，该项目一共有13道，它的总得分是右下角的标签，目前只是给它一个超大的默认分值，这样也就将标签控件的宽度拉得足够大，不至于在显示分值的时候因为数值太大、太长而导致文本内容被截取的情况。

图7-4　Quizzler的初始化项目

图7-5　Main.storyboard文件中的初始用户界面

现在将Xcode切换到助手编辑器模式，确保左边打开的是Main.storyboard，右边打开的是ViewController.swift文件。

在ViewController类中一共有4个IBOutlet变量，将鼠标放在ViewController.swift文件中每个@IBOutlet关键字前面的实心圆点时可以发现，与questionLabel关联的是视图中显示问题的标签控件，与scoreLabel关联的是底部的分值标签控件，与progressBar关联的是底部的答题进度视图，与progressLabel关联的是视图左下角的答题数/总题数标签，如图7-6所示。

图7-6　查看IBOutlet变量与故事板中界面元素的关联

通过前面的学习我们知道这4个IBOutlet变量的名称不可随意修改，因为故事板中的UI控件还是会指向修改之前的变量名，在故事板视图中的UI控件上右击鼠标，在弹出的面板中可以看到相关的IBOutlet和IBAction关联，如图7-7所示。例如，当前视图顶部的标签控件，它的IBOutlet引用就指向了ViewController类里面的questionLabel变量。如果关联出现了问题，例如将变量名称修改为myQuestionLabel，一定要查看UI控件的关联条目是否和新名称匹配。

图7-7　在故事板中检查Label标签与questionLabel变量的关联

在ViewController类中还有一个IBAction方法，它会响应两个按钮的用户交互操作，并且基于不同的tag值来区分用户单击的到底是哪个按钮。

选中标题为“是”的按钮，在Attributes Inspector中找到Tag属性，当前它的值为1，标题为“否”的按钮的Tag值为2。与Xylophone项目一样，我们还是通过sender参数的tag属性判断用户单击的是哪个按钮。

在ViewController类中还有4个提前创建好的方法，在这些方法中暂时没有实现任何代码。

7.2 创建数据模型

本节中我们会创建数据模型（Data Model）。在项目导航中可以看到Quizzler中包含三个文件夹，其中的Controller文件夹包含了AppDelegate.swift和ViewController.swift这2个文件。在View文件夹中包含了Image.xcassets和Main.storyboard这2个文件。但是在Model文件夹中却没有包含任何文件，目前它还是空的，因为我们还没有创建任何的数据模型。创建数据模型需要创建一个新的Swift类型的文件。

到目前为止，我们用到的所有Swift文件都是通过Xcode模板自动创建的，要想在Model文件夹中创建文件，需要在项目导航选中Model文件夹，在Xcode菜单中选择File/New/File...，然后在弹出的文件模板选择面板中确保选中iOS标签，在Source部分里选中Swift File格式的文件，然后单击Next按钮。将文件名设置为Question，并确定Group当前选中的是Model，也就意味着新文件位于Model文件夹中。Targets中的Quizzler一定要处于勾选状态，然后单击Create按钮。

在项目导航中展开Model文件夹，然后选中新创建的Question.swift文件，该文件中除了版权声明和import Foundation导入语句之外没有其他。与之前有所不同，这里导入的是Foundation框架库，而不是UIKit。Swift提供了能够满足各种需求的框架库，就好比商店里出售的各种瑞士军刀一样，它们有大有小，各具不同的功能，从而满足不同用户的需求。在Question.swift中，我们实际上只是需要一个轻量级瑞士军刀的基本功能，比如刀子和瓶起子。所以要根据需求来选择导入的框架库。Foundation框架库比UIKit的量级要“轻”很多。

接下来我们要创建一个新类，第一件事就是使用class关键字声明一个类。

```
import Foundation

class Question {
```

```
}
```

在声明类的时候，一个非常重要的事情就是类名称的命名方式，与变量和方法的驼峰命名法不同，类名称的首字母一定要大写。

该类中一共需要两个属性：一个是questionText常量，字符串类型；另一个是answer常量，布尔类型。

```
class Question {
  let questionText: String
  let answer: Bool
}
```

现在这个Question类将会作为问题的架构，每一个单独的问题都会使用该类来存储问题和答案。我们管在类中声明的常量或变量叫作属性，管类中的函数叫作方法。函数和方法会有细微不同，方法相当于类中的函数。简单来说，在类中实现的函数叫作方法，在类外面就叫作函数。

此时的Question.swift会报错，如图7-8所示。说明此时的Question类还没有初始化方法。同时，类中的两个属性也还没有被赋初始值。

图7-8　创建用于存储问题的Question类

接下来让我们为这两个属性赋初始值，代码如下：

```
class Question {
  let questionText: String = "你还是你吗？"
  let answer: Bool = true
}
```

当赋值完成以后，错误马上消失。这是因为类中的属性都已经被赋值，没有不确定因素的隐患存在。但是，修改后的代码并没有实际意义，因为常量已经被赋值，而这个值并不是我们真正需要的。

因此我们要使用init关键字来创建初始化方法。

```
class Question {
  let questionText: String
  let answer: Bool

  init(text: String, correctAnswer: Bool) {
    questionText = text
    answer = correctAnswer
  }
}
```

当我们创建一个新的Question类型对象的时候，可以通过初始化方法来确定还需要做些什么。初始化方法的第一个参数text是字符串类型的问题描述，第二个参数是布尔类型的正确答案。在init（）方法中，通过两行代码为属性questionText和answer赋值。

在本节中我们接触到了Foundation框架库，它为应用程序提供了最基础层面的功能，包括数据存储和持久性连接、文本处理、日期和时间计算、排序和过滤以及网络连接。我们可以在https://developer.apple.com/reference/foundation里查阅到更多关于Foundation框架库的信息。

UIKit框架为iOS或tvOS应用程序提供了所需的基础架构。它提供实现用户界面的窗口和视图体系结构，用于向应用程序提供多点触控和其他类型输入的事件处理基础结构，以及管理用户、系统和应用程序之间的交互所需的主运行循环。该框架还提供了包括动画支持、文档支持、绘图和打印支持，关于当前设备的信息，文本管理和显示，搜索支持，可访问性支持，应用程序扩展支持和资源管理相关功能。在https://developer.apple.com/reference/uikit里可以查阅到更多相关信息。

7.3 面向对象

本节我们会学习程序开发中一个非常重要的概念——面向对象（Object Oriented Programming，OOP）。之前，你经常会看到类（Class）和对象（Object）的概念，在本节我们将会深入理解它们之间的不同以及如何使用它们创建复杂的应用程序项目。

要想理解面向对象，让我们先回到最初的编程时代。我们都知道计算机只认识0和1两个数字以及0和1组成的代码，我们管它叫作机器语言。其实，你所看到的计算机就是由百亿、千亿个切换器构成，切换器的状态非0即1。所以作为程序员，我们可以让计算机去根据需求进行计算，或者是通过0、1的数字串格式给它下达指令。要想使用机器语言，你需要键入成千上万的0和1，如果你决心使用机器语言来开发应用程序的话，那将是一个非常疯狂和不切实际的想法。因此，有人开始使用近似于英文语法的编程语言，去编写可以让计算机翻译成0/1格式的机器语言的代码。

在1970年的时候，丹尼斯·里奇以B语言为基础，在贝尔实验室设计、开发出了C语言。它其实就是现代编程语言的鼻祖，在其之后推出的C#、C++和Objective-C都是以C语言为基础的。实际上，还有很多C语言特性都留存在其他语言中，例如PHP、Java、Scala等。

C语言属于过程式编程语言的范畴，过程式编程就是按照指令列表依次执行的程序编码方式。就好比开始做什么，然后做什么，再做什么，再做什么……一行一行地执行代码。举个例子，你的餐厅仅有一位职员，需要让她完成所有的事情。从一早开始要打开餐厅的门，开灯，根据顾客的订单将菜品放到餐桌上，给用完餐的顾客结账。总之，你总是给职员一个很长很长的任务单，让她从开始一步一步做下去。当然，这样的形式并不灵活，同时指令列表会很长很长也非常难于调试。

如果我们将餐厅的雇员从一位增加到三位，一位是服务员，一位是糕点师，一位是厨师。现在这三个人都有自己的角色，做着自己所负责

的工作。他们每个人都有自己擅长的技能，并且如果哪一位的工作做得不理想，可以随时更换一位新的雇员，而不会影响到另外两个人。

通过这种现代的方式，有助于帮助我们以更加简单的方式进行调试。因为出现问题的地方会被限制在某个独立的对象中，修改类中的代码并不会影响其他方面的代码。这就是面向对象开发的概念。

我们可以发送消息给某个对象，例如餐厅中需要一些蛋糕，则需要给糕点师发送消息，糕点师在收到消息以后就开始制作更多的蛋糕。在这样的环节中，信息的传递并不需要经过服务员，她并不关心制作蛋糕的事情。

在之前的很长一段时间，苹果使用面向对象的C语言——Objective-C作为基于macOS系统的程序设计语言。就像是Android平台使用Java语言开发一样。而Swift语言则是苹果近几年自行研发的用于iOS、macOS、tvOS和watchOS平台的程序设计语言。

在简单了解了面向对象开发的概念以后，接下来就要看看类和对象之间有什么不同。类就好比一张蓝图，它包含了创建一个对象的指令集。如图7-9所示，这是一个汽车类而不是真正的汽车，因此它不是一个汽车对象。它仅仅是一个如何创建汽车的指令集，比如指定有多少个座椅，有多少个门，车是什么颜色的。

图7-9　制造汽车的蓝图

可以想象一下，我们会创建一个Car类，car对象则是通过Car类创建的，它包含三个主要内容。第一个是属性（property），例如在上一节的Question类中声明了两个属性。第二个是对象中可以执行的方法（action），它是对象可以实现的功能，例如汽车如何前进，如何刹车，如何转向。第三个是事件（event），一旦某些事情发生的时候，需要对象要做出何种反应，例如当汽车启动以后要做什么，或者是当

雨点掉在挡风玻璃上的时候要做什么。在创建类的时候，我们要定义好这三件事，也就是创建好这张“蓝图”。

在之前的Question类中，我们定义了两个属性，使用常量定义了questionText和answer。我们使用init初始化方法定义了一个事件，当从Question类创建对象的时候会执行该方法。在执行初始化方法的时候，要传递两个参数，一个是问题，一个是正确答案。最后，是一个能够被类中其他方法随时调用的someFunction（）方法，如图7-10所示。

图7-10　Question类的属性、事件与方法

在面向对象开发的过程中，为了能够让程序员清楚地描述类和对象的相关内容。让我们尝试着说明一些词的意思。当我们说的是在类的范围内创建的常量和变量的时候，它指的就是属性。如果是在类的范围之外所声明的就只能称为常量或变量。我们在类中所声明的函数就叫作方法，如果是在类外声明的就叫作函数。

如果你是一个苹果粉，可能会对史蒂夫·乔布斯（Steve Jobs）有关面向对象编程（Object-Oriented Programming）的话题感兴趣。下面是1994年滚石访谈的摘录，史蒂夫（并不是纯粹的程序员）简单地解释了OOP。

史蒂夫·乔布斯说：对象就像人一样。他们真实地存在，呼吸着，并富有知识内涵。他们知道如何做事情，有记忆，可以回忆之前的事情。程序员不是在低级别层面上与它进行交互，而是在非常高的抽象层次上进行交互，就像我们做的这样。

这里有一个例子：如果我是你的洗衣对象，你可以把脏衣服给我，并发给我一条消息，说：“你能洗我的衣服吗？。”我碰巧知道旧金山最好的洗衣店在哪里。我说的是英语，口袋里正好有钱。所以我出门打辆出租车，告诉司机带我到旧金山的那个地方。我去洗你的衣服，打

出租车再回到这里。我给你干净的衣服，并说：“这是给你洗干净的衣服。”

你不知道我是怎么做到的。你不知道洗衣的地方。也许你说法语，甚至不会叫出租车。你无法支付，因为你口袋里没有美元。我知道如何做到这一切，而你不必了解这些。所有这些复杂的事情都被我解决了，而且能够在非常高的抽象层次上进行交互，这就是对象。它们封装了复杂性，并且复杂性的接口是高层次的。

7.4 创建答题库类

这一节我们将会创建一个新类。

在项目导航选中Model文件夹，Control-click文件夹后，在快捷菜单中选择New File...，与之前一样，在文件模板选择面板中选中iOS标签中的Swift File，文件名设置为QuestionBank，确保Group为Model，Targets中勾选Quizzler，单击Create按钮。此时在Model文件夹中新增加了QuestionBank.swift文件。

选中新创建的QuestionBank.swift，当前的文件中只导入了Foundation框架库。

QuestionBank类的目的在于存储所有题目，并为Quizzler项目使用。该题库中的每一道题都是QuestionBank类型的对象。

通过下面的代码创建QuestionBank类。

```
class QuestionBank {
}
```

在创建好QuestionBank类以后，需要在其内部声明一个Question类型的数组。var代表它是变量，list是类的属性名，在等号的右侧是[Question]，代表数组中所存储元素的数据类型是Question，最后使用一对小括号结束数组的声明。在类中创建的变量或常量就叫作属性，我们通过这个属性来管理应用中的数据。

提示　声明一个数组有很多方式，这里所使用的方法会得到一个初始化好的空数组，只不过这个数组中还没有任何的元素。

为了能够向数组中填充需要的数据，需要创建一个初始化方法。

```
class QuestionBank {

  var list = [Question]()

  init() {
    // 创建一个问题，再将它添加到list数组中。
    let item = Question(text: "吃烧烤不能喝啤酒。", correctAnswer:
true)

    // 添加Question对象到list数组中
    list.append(item)

    // 跳过第一步，直接通过数组的append()方法将Question对象添加到list
    list.append(Question(text: "喝白酒时最好喝茶水。",
correctAnswer: false))

    list.append(Question(text: "空腹最好不吃柿子。",
correctAnswer: true))

    list.append(Question(text: "在野外遇到雷雨天气时，感觉自己的头发
竖起来了，皮肤发热，要立刻就地卧倒，不可继续站立。", correctAnswer:
true))

    list.append(Question(text: "参加运动会，临赛前一定要吃饱，这样可
以增加体能取得好成绩。",
correctAnswer: false))

    list.append(Question(text: "面膜做的时间越久越好。",
correctAnswer: false))

    list.append(Question(text: "晕船时应尽量将头部固定，不要让头部来
回晃动。",
correctAnswer: true))

    list.append(Question(text: "被鱼刺卡住以后应该猛吃食物，迅速咽
下。", correctAnswer: false))

    list.append(Question(text: "红眼病病人是可以与他人共用生活用品和
学习用品的。", correctAnswer: false))

    list.append(Question(text: "身上着火后，应迅速用灭火器灭火。",
correctAnswer: false))
```

```
    list.append(Question(text: "创伤伤口内有玻璃碎片等大块异物时，到医院救治前，可自行取出。"
, correctAnswer: false))

    list.append(Question(text: "发生煤气中毒时，首先应将门、窗打开通风换气。", correctAnswer: true))

    list.append(Question(text: "进行人工呼吸前，应先清除患者口腔内的痰、血块和其他杂物等，以保证呼吸道通畅。", correctAnswer: true))

  }
}
```

我们使用init关键字创建初始化方法，与之前Question类带有2个参数的初始化方法不同，这里的方法不需要任何参数。在其内部，我们要设置所有13道题目的信息。如果你不想输入上面这些信息的话，可以回到GitHub网站的Quizzler项目文件夹，在README.md中找到相关代码，再将其复制到init（）方法里面。

在init（）方法中，我们首先声明了一个常量item，并通过Question类的初始化方法将生成的对象赋值给它。然后使用数组的append（）方法将Question对象添加到数组的末尾。此时list数组中已经包含了一个Question类型的对象。

接下来，我们使用append（）方法，直接将Question类的初始化方法所生成的对象添加到list数组中，这里一共添加了13道题目。

7.5 Model View Controller（MVC）设计模式

通过本章的Quizzler实战练习，我们要更加深入地了解MVC设计模式。为什么设计模式这么有用，我们应该如何使用设计模式呢？

首先需要搞清楚设计模式是什么？它是开发者对普遍问题的最简单最实用的解决方案。用一个实际生活中的例子来解释：很早很早以前，在人类面对又黑又冷的环境时，是如何御寒呢？早期的人类通过建造草屋来解决这个问题，并且在草屋中还可以躲避其他动物的攻击。但是后来他们认识到火可以阻挡动物的攻击。但是如果有一些凶猛的野生动物潜伏在你家的周围时，家人可能随时会有危险。所以人类改进了设计模式——使用泥巴建造房屋，这样动物在向人类发起攻击的时候就不会那么轻易得手了。后来人类又改进了设计模式——使用木材建造房屋，这样的房子不仅保暖而且更加安全。大多数人都会认为这是一个非常不错的设计模式。为了能够让木屋看起来不像是一间茅草屋，在建造的时候就需要设计模式（蓝图）。设计模式规定了房子的墙建在哪里，房顶建在哪里，卫生间要挨着卧室，每个房间要有窗户等，依据这个设计模式就可以建造出一个具有基本功能的房子，至于它的装修、个性化设置可以在之后慢慢实施。

如果你有一个几万行代码的复杂应用程序，设计模式可以帮助你做什么呢？当我们使用MVC设计模式以后，这个应用就不再像是一碗“炸酱面”，面、酱、黄豆、蒜、肉丁、豆芽、萝卜丁……都混在一起，而是通过组件的形式将它们搭建在一起。

首先让我们来看看组件，MVC设计模式中的M代表数据模型（Model），它负责构建和管理数据。如果项目使用了数据库的话，则它会负责与数据库的协调和沟通，进而读取或删除数据库中的数据。实际上，我们使用它来构建数据，并将数据通过一定形式传递给控制器（View Controller）组件去进行下一步的处理。

接下来是视图（View），它其实就是我们在屏幕上面看到的东西。

在这两个组件中间的就是控制器（Controller），它负责数据模型与视图之间的沟通。如果你运行了iOS的通信录程序，并且查找里面叫作李钢的手机号码，则需要向控制器提出请求，控制器将这个请求做一些处理后向数据模型发送一条消息，询问是否有李钢的手机号码数据。数据模型就像是仓库管理员，它从数据库中获取数据并格式化为一个控制器能看懂的结构（类的对象），然后返回给控制器，最后控制器再将相关数据赋值给视图控件，所需的信息最终便呈现到了iPhone屏幕上面。

MVC设计模式对于信息的传递有一些抽象，我们用一个现实生活的例子来说明。在餐厅中，数据模型就是这里的食材，例如鸡蛋、牛奶、面粉、蔬菜和肉。数据模型被厨师加工成一道道菜肴。这些菜肴会传递给服务员，这里的服务员就是控制器。服务员会将做好的菜肴传递给视图，也就是食客们的餐桌上。

这里，控制器（服务员）会先得到一个请求——1号桌需要一盘拍黄瓜。服务员会将需求告诉厨房的厨师，由厨师将食材准备好以后按照要求做一盘拍黄瓜。然后服务员再将拍黄瓜送到顾客的餐桌上。

让我们回到之前的Quizzler项目代码，在项目导航中可以看到此时的项目已经被分为Model、View、Controller三个部分，通过文件名可知ViewController.swift是一个控制器组件，它可以让视图显示信息，也可以向数据模型索要信息。如果餐厅中的顾客点了一张披萨，作为控制器的服务员就会在得到这个需求以后，马上将信息反馈给顾客：对不起，这是一家中餐厅，我们无法制作披萨。

Model文件夹中的Question.swift的功能就是将数据以题目的方式组织起来，便于将它们在屏幕上显示出来。QuestionBank类的作用就是组织所有的Question对象。

Controller文件夹中的ViewController.swift会将数据显示在视图或屏幕上，让用户可以看到最终的结果，并与其进行交互。

为什么要使用MVC设计模式？它有什么优点呢？一是它提供了完美的代码架构。当你编写复杂代码的时候，无法真正完全掌控它会发生什么。现在，将项目设计为Model-View-Controller设计模式，你可以发现这些关系在代码中会实现得非常好。二是作为编程界最常用的设计模式，它非常容易被其他程序员识别出来。所以，在两个以上程序员负责的项目中，使用MVC设计模式可以很好地做到无缝连接，不会或很少发生混乱的问题。第三，利用MVC可以复用代码，你会注意到在MVC设计模式中，数据模型与视图永远不会直接发生联系。也就相当于在餐厅中，顾客永远不会和厨师发生任何联系一样。例如，如果Quizzler项目在法国运行，我们可以轻松地将数据从英文换成法文，而不用考虑其他的代码问题。最后，使用MVC设计模式允许我们使用多任务特性，这与你编写项目的代码多少有关。例如有三个程序员共同开发一个项目，你可能会让一位同事更多地关注视图部分，他应该是前端设计师。另一位同事是后端设计师，更多的是维护和校验数据等。第三位同事则是更多地在控制器中编写代码，将后端数据呈现到前端视图之中。这样三位程序员就有合作有分工，提高了开发效率。

有很多iOS开发初学者可能不太喜欢使用MVC设计模式，因为它看起来添加了一些多余的工作，但是到了项目开发的后期，到了开始维护该项目并进行功能升级的时候，它的优势就会立即显现出来。

7.6 初始化第一个题目

在之前的实战练习中我们已经在Model文件夹中创建了QuestionBank类。在QuestionBank类中声明了一个数组list用于存储Question对象，每个Question对象都包含了问题与答案。之后，我们又创建了13个Question对象，并将其添加到list数组之中。因为这些代码都是在QuestionBank的初始化方法中执行的，所以在创建QuestionBank对象的时候，13道题目就会被添加到list属性中。

在ViewController类中，我们将会创建一个QuestionBank类型的对象。

```
class ViewController: UIViewController {

  let allQuestions = QuestionBank()
  ……
```

接下来，要将QuestionBank对象中的第一道题显示到屏幕上。在viewDidLoad（）方法中添加下面的代码：

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

  let firstQuestion = allQuestions.list[0]
  questionLabel.text = firstQuestion.questionText
}
```

在viewDidLoad（）方法中创建了常量firstQuestion，并将allQuestions对象中的数组list里面的第一个元素赋值给它，因为list数组中的元素为Question类型，所以通过Swift的类型断言特性，自动将firstQuestion设定为Question类型。

因为数组的第一个元素索引值为0，所以使用[0]来获取该元素对象。第一个元素的题目应该是：吃烧烤不能喝啤酒。以此类推，[1]是第二

题，题目内容应该是：喝白酒时最好喝茶水。

然后将firstQuestion的questionText属性值赋值给IBOutlet变量questionLabel的text属性。因为questionLabel指向的是故事板里面的题目标签，所以当代码执行到这里的时候会将题目显示到屏幕上面。

通过刚刚在viewDidLoad（）方法中新添加的两行代码我们可以了解到，当我们需要访问或设置对象里面的某个属性的时候，需要使用点（.）操作符。

构建并运行项目，当应用启动以后就会看到第一道题目显示到了屏幕上的QuestionLabel位置上，如图7-11所示。目前两个按钮还不能工作，因为我们还没有对相关方法编写程序代码。

图7-11　在Quizzler项目中显示第一道问题

当用户单击是/否两个按钮的时候，将会激活IBAction方法answerPressed（_sender：UIButton）。为了能够判断用户单击的按钮是否为题目的正确答案，需要在ViewController类中再添加一个属性pickedAnswer，该变量用于存储用户选择的答案，变量的类型为布尔型，并且将初始值设置为false。

```
class ViewController: UIViewController {

  let allQuestions = QuestionBank()
  var pickedAnswer: Bool = false
  ……
```

接下来在answerPressed（_sender：UIButton）方法中添加下面的代码：

```
@IBAction func answerPressed(_ sender: UIButton) {
  if sender.tag == 1 {
```

```
    pickedAnswer = true
  }else if sender.tag == 2 {
    pickedAnswer = false
  }

  checkAnswer()
}
```

因为在故事板中，已经提前将是（第一个）按钮的tag设置为1，否（第二个）按钮的tag设置为2。所以在answerPressed（_sender：UIButton）方法中通过参数sender.tag来确定用户单击的是哪一个按钮。如果tag值为1，则代表用户单击了是按钮，否则代表用户单击了否按钮。当用户单击按钮做出回答以后，要判断回答是否正确。这里我们会调用类中的checkAnswer（）方法来检查对错，只不过目前该方法中还没有任何的代码。

提示　我们使用==操作符判断左右两边表达式的值是否相等，使用!=操作符判断左右两边表达式的值是否不相等。而单独的等于号在Swift语言中是赋值号，即将右边表达式的值赋值给左边的常量或变量。

在checkAnswer（）方法中添加下面的代码：

```
func checkAnswer() {
  let correctAnswer = allQuestions.list[0].answer

  if correctAnswer == pickedAnswer {
    print("回答正确！")
  }else {
    print("错误！")
  }
}
```

在上面的代码中，通过allQuestions的list[0]获取到数组中的第一个元素对象，再直接通过点（.）操作符获取该对象的answer属性值，并将

其赋值给常量correctAnswer。接下来就是使用if语句判断用户回答是否正确，并在控制台输出相应的信息。

构建并运行项目，单击题目下方的两个按钮，控制台中会显示相应的结果，但是当前我们只会停留在第一题中。

7.7 处理后续题目

在上一节中，我们对应用程序中所显示的第一道题进行了处理，在本节中将会继续编写代码来处理后面的题目。

首先为类再添加一个属性（变量）var questionNumber：Int=0，该变量用于跟踪答题的状态，也就是用户当前回答了多少道题。每回答完一道题，就让该变量的值加1。

```
class ViewController: UIViewController {

  let allQuestions = QuestionBank()
  var pickedAnswer: Bool = false
  var questionNumber: Int = 0

  ……

  @IBAction func answerPressed(_ sender: UIButton) {
    if sender.tag == 1 {
      pickedAnswer = true
    }else if sender.tag == 2 {
      pickedAnswer = false
    }

    checkAnswer()
    questionNumber += 1
    questionLabel.text =
allQuestions.list[questionNumber].questionText
  }
  ……
```

当应用程序开始运行的时候，会先执行viewDidLoad（）方法，第一道题也就相应地呈现到屏幕上，直到用户单击是/否按钮以后，我们应该做些什么呢？每个问题只有一个正确答案，所以当用户单击其中一个按钮以后，会通过checkAnswer（）方法检查对错，接下来就需要在屏幕上呈现第二道题，因此需要在调用checkAnswer（）方法之后让questionNumber的值加1。

在将questionNumber的值加1以后，为questionLabel的text属性重新赋值，这个值就是list数组中的第questionNumber个元素的questionText属性值，只要用户单击一次按钮，questionNumber就会加1，然后会显示一道新的题目。

构建并运行项目，可以依次作答完成所有的问题，直到在完成最后一道题目的选择后程序崩溃，如图7-12所示。崩溃的原因是出现了致命错误——超出了数组索引值范围，我们会在之后解决这个问题。

图7-12　做完12道题目以后应用程序崩溃

目前首要解决的是用户回答正确与否的问题，在控制台显示的信息与题目的正确答案不一致。问题出现在checkAnswer（）方法中，当前我们总是将list数组中第一道题的答案与用户的选择做比较，这肯定是不行的。修改代码如下面这样：

```
let correctAnswer = allQuestions.list[questionNumber].answer
```

再次构建并运行程序，该问题解决。

7.8 使用Xcode调试控制台

本节我们会继续深入学习如何利用控制台让整个项目代码的调试变得更加容易。在上一节中所出现的致命错误，是因为questionNumber变量导致的，这一点在崩溃的时候我们就已经猜想到了，但是如何证明这一点呢？我们需要通过调试控制台打印出当前questionNumber的值。

在控制台中键入print命令：print questionNumber，当我们输入questionNumber的时候还会出现和代码编辑器一样的自动完成窗口，选择需要的内容直接按回车键即可，如图7-13所示。你会发现这个print命令与Swift语言稍微有些不同，print之后并没有小括号，只有需要呈现结果的变量名，按回车键以后你会看到输出的结果。

图7-13　在调试控制台中打印questionNumber的值

```
(lldb) print questionNumber
(Int) $R0 = 13
```

这里可以看到questionNumber的值为Int类型的13。其实就是在全部问题都呈现到屏幕上以后，questionNumber又再次加1，而list数组中已经没有索引值为13的元素对象了。

其实，我们可以随时在控制台中输出当前类中的变量值，比如在控制台中键入print allQuestions.list，就会看到下面的输出结果：

```
(lldb) print allQuestions.list
([Quizzler.Question]) $R1 = 13 values {
  [0] = 0x000060c00045e5a0 (questionText = "吃烧烤不能喝啤酒。",
answer = true)
  [1] = 0x0000604000257f40 (questionText = "喝白酒时最好喝茶水。",
```

```
answer = false)
  [2] = 0x0000604000257d60 (questionText = "空腹最好不吃柿子。", answer = true)
  [3] = 0x0000604000258060 (questionText = "在野外遇到雷雨天气时，感觉自己的头发竖起来了，皮肤发热，要立刻就地卧倒，不可继续站立。", answer = true)
  [4] = 0x0000604000257fa0 (questionText = "参加运动会，临赛前一定要吃饱，这样可以增加体能取得好成绩。", answer = false)
  [5] = 0x0000604000257f70 (questionText = "面膜做的时间越久越好。", answer = false)
  [6] = 0x0000604000258090 (questionText = "晕船时应尽量将头部固定，不要让头部来回晃动。", answer = true)
  [7] = 0x0000604000258030 (questionText = "被鱼刺卡住以后应该猛吃食物，迅速咽下。", answer = false)
  [8] = 0x0000604000258000 (questionText = "红眼病病人是可以与他人共用生活用品和学习用品的。", answer = false)
  [9] = 0x00006040002580c0 (questionText = "身上着火后，应迅速用灭火器灭火。", answer = false)
  [10] = 0x00006040002580f0 (questionText = "创伤伤口内有玻璃碎片等大块异物时，到医院救治前，可自行取出。", answer = false)
  [11] = 0x0000604000258120 (questionText = "发生煤气中毒时，首先应将门、窗打开通风换气。", answer = true)
  [12] = 0x0000604000258150 (questionText = "进行人工呼吸前，应先清除患者口腔内的痰、血块和其他杂物等，以保证呼吸道通畅。", answer = true)
}
```

因为list是一个数组，所以在控制台可以看到数组中所有的13个值，其中索引值是从0到12。如果我们硬是要访问list数组索引值为13的元素，肯定会发生致命错误，因为根本就没有这个值。

如果我们不想使用print语句来查看变量信息的话，还可以通过控制台左侧的变量查看窗口解决，如图7-14所示。展开self条目可以发现，allQuestions里面有list数组，其中一共有13个元素，最后一个元素的索引值为12。而当前questionNumber的值为13，根本没有该索引值的元素存在。

图7-14　查看当前运行类中变量的值

目前，已经介绍了两种使用控制台进行调试的方法，并找到了问题所在，下面就是要解决这个问题。停止应用程序的运行，修改answerPressed（_sender：UIButton）方法和nextQuestion（）方法。

```
@IBAction func answerPressed(_ sender: UIButton) {
  if sender.tag == 1 {
    pickedAnswer = true
  }else if sender.tag == 2 {
    pickedAnswer = false
  }

  checkAnswer()
  questionNumber += 1

  nextQuestion()
}

func nextQuestion() {
  if questionNumber <= 12 {
    questionLabel.text =
allQuestions.list[questionNumber].questionText
  }else {
    print("全部答完！")
    questionNumber = 0
  }
}
```

在answerPressed（_sender：UIButton）方法中直接调用nextQuestion（）方法，将之前的questionLabel.text=allQuestions.list[questionNumber].questionText移动到nextQuestion（）方法中。然后通过if语句进行条件判断，如果questionNumber的值小于等于12则执行该代码。如果超过12则在控制台打印信息，并将questionNumber的值重置为0，这样当用户回答完第13道题目以后再单击按钮就会回到第一题。

构建并运行项目，查看该问题是否被解决。但是目前该项目的用户体验还不是很好，需要在后面的章节中进一步改进。

7.9　如何实现UIAlertController以及弹出窗口给用户

在上一节中我们学习了如何通过检测机制来防止应用程序在运行到最后的时候出现崩溃情况。并且在questionNumber的值大于12的时候还会在控制台中输出一条信息，但是这条信息仅限于程序员在控制台中可以看到，本节将解决如何让用户在回答完13道题目以后得到消息通知，并询问是否需要重做这些题目的问题。

在本节我们会使用一个全新的用户界面控件UIAlertController，它有一个标题和一个文本信息，还有一个取消和一个确定按钮。通过用户的选择来决定应用程序下一步要做什么，如图7-15所示。

有关UIAlertController的详细介绍可以查看苹果开发手册的相关内容，链接地址为：
https://developer.apple.com/documentation/uikit/uialertcontroller。

在概述（Overview）部分有对该UI控件的说明和使用方法，以及使用的样例代码。在主题（Topics）部分中，有创建警告控件的初始化方法，该方法会创建并返回一个可以显示给用户的警告视图控制器。通过它的参数可以指定对话框的标题（title）、信息（message）以及警告控件的样式（preferred style），目前它包含的样式有action sheet和modal alert两种。modal alert会在屏幕的中央呈现一个对话框，在对话框中出现几个可选项，如图7-15所示。action sheet会从屏幕底部滑出一个界面，它也有类似的标题和信息，并让用户做出选择，如图7-16所示。

图7-15　UIAlertController控件的外观

图7-16　UIAlertController控件的action sheet风格

在当前项目中，我们会使用modal alert警告方式。修改nextQuestion（）方法中的代码：

```
func nextQuestion() {
  if questionNumber <= 12 {
    questionLabel.text =
allQuestions.list[questionNumber].questionText
  }else {
    let alert = UIAlertController(title: "了不起！", message: "你
已经完成了所有的题目，是否想重新开始呢？", preferredStyle: .alert)
  }
}
```

除了在对话框中显示标题和信息以外，还需要有两个按钮让用户选择是否重新做题。这需要通过UIAlertAction类实现动作按钮。通过苹果的帮助文档https://developer.apple.com/documentation/uikit/uialertaction可以找到UIAlertAction的相关说明。它的初始化方法如下：

```
init(title: String?, style: UIAlertActionStyle, handler:
((UIAlertAction) -> Void)? = nil)
```

其中title是按钮的标题，style是按钮的风格，handler则代表用户单击按钮以后要做什么。至于按钮的风格一共有三种，它是UIAlertActionStyle枚举类型，包括：默认（default）、取消（cancel）和不可逆（destructive）。

继续修改nextQuestion（）方法：

```
func nextQuestion() {
  if questionNumber <= 12 {
    questionLabel.text =
allQuestions.list[questionNumber].questionText
```

```
  }else {
    let alert = UIAlertController(title: "了不起！", message: "你已经完成了所有的题目，是否想重新开始呢？", preferredStyle: .alert)

    let restartAction = UIAlertAction(title: "重新开始", style: .default, handler:
{ (alertAction) in self.startOver() })

    alert.addAction(restartAction)
    present(alert, animated: true, completion: nil)
  }
}
```

首先，我们通过UIAlertAction类的初始化方法创建一个按钮，该按钮的标题为重新开始，风格是默认类型，handler参数用于设定当用户单击该按钮以后所执行的代码。如果你之前没有接触过Swift语言的话，对这样的语法可能会感到有些奇怪。当前我们提供给handler参数的内容并不是一个值或一个对象，而是一段代码，这段代码被一对大括号包围起来（handler：{}），因此管这样的方式叫作闭包。在当前的handler闭包中需要执行类中的startOver（）方法，因此需要调用self.startOver（）。因为闭包有其独立的生存期，它是独立在类之外的特殊代码块，即使代码块是在类中也是如此。所以在闭包之中需要使用self关键字指明要执行当前类中的startOver（）方法。如果删除闭包中的self关键字，Xcode编译器则会报错。现在你只需要清楚的是，利用self关键字可以在整个Swift文件中搜索startOver（）方法。有关闭包和self的相关知识我们会在之后的章节中详细介绍。

接下来，需要利用addAction（）方法将restartAction对象添加到alert控制器中。

最后利用present（）方法，让alert控制器呈现到iPhone屏幕上，其中第一个参数是alert控制器对象，animated代表是否启用动画效果，completion与handler一样是个闭包，代表在alert控制器呈现到屏幕上以后要执行什么代码，这里使用nil关键字代表没有任何操作。

构建并运行项目，在完成13道题目以后，会弹出一个警告窗口，此时可以单击重新开始按钮来重新作答，如图7-17所示。

图7-17　做完所有题目以后可以重新作答

目前，当我们单击重新开始按钮以后并没有任何的事情发生，仅仅是关闭了警告窗口而已。因为在代码中，当用户单击重新开始按钮以后会执行类中的startOver（）方法，而目前该方法中并没有任何执行代码。在startOver（）方法中添加下面的代码：

```
func startOver() {
  questionNumber = 0
}
```

在该方法中，我们会重置questionNumber的值为0，让用户可以重新做题。

再次构建并运行项目，在单击重新开始按钮以后，用户可以重新作答。但是这里还有一个Bug需要修复。

当用户单击重新开始按钮以后，会执行startOver（）方法，此时questionNumber的值会重置为0。而当前屏幕上的题目还停留在第一轮的最后一道，没有进行更新，当用户单击按钮以后才会更新题目。在调用answerPressed（_sender：UIButton）方法的时候，会先执行questionNumber+=1代码，此时questionNumber的值变为了1，再执行nextQuestion（），这就意味着从第2轮开始，重新开始以后永远无法显示题库中的第2道题。

修复这个Bug非常简单，只要在startOver（）方法中questionNumber=0代码的后面添加一行nextQuestion（）即可。

7.10 高级别的重写

这一节我们将会整理之前的代码文件，以提高代码的可读性，便于后期的维护。

在Quizzler项目启动以后，首先会运行ViewController类中的viewDidLoad（）方法，在该方法中，创建了firstQuestion对象，并且将list数组中的第一个元素赋值给它。随后，将题目内容显示到屏幕的questionLabel控件上。

根据用户的答题选择，会执行类中的answerPressed（_sender：UIButton）方法，如果sender参数的tag属性值是1，则代表用户选择了是。如果是2则代表选择了否。在该方法中，通过调用checkAnswer（）方法检查回答是否正确。在检查完成以后会让questionNumber的值增加1，再调用nextQuestion（）方法。

在nextQuestion（）方法中，首先会判断questionNumber的值是否小于等于12，如果为真则更新题目内容。如果为假则代表13道题目全部作答完毕，这时会弹出一个警告对话框，询问用户是否要重新开始。当用户单击重新开始按钮以后，则会将questionNumber重置为0，并调用nextQuestion（）方法更新屏幕上的题目。

实际上在viewDidLoad（）方法中的两行程序代码与nextQuestion（）方法实现的功能相同，所以将viewDidLoad（）方法修改为下面这样也可以：

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

  nextQuestion()
}
```

7.11 统计分数

可能你已经注意到了，在目前项目中还没有进度条的变化和分数值的显示，用户还不知道已经做了几道题，当前得了多少分。本节我们将解决这些问题。

我们需要一个变量来跟踪所得到的分数，就如同使用questionNumber跟踪当前做到第几题一样。在ViewController类中添加一个变量。

```
var questionNumber: Int = 0
var score: Int = 0
```

当用户答对一道题以后就需要增加score的值，所以修改checkAnswer（）方法如下面这样：

```
if correctAnswer == pickedAnswer {
  print("回答正确！")
  score = score + 1
}else {
  print("错误！")
}
```

当分数有所变化以后，需要更新scoreLabel来显示最新的分数值，我们需要在哪里完成scoreLabel的更新呢？可能你会想到的位置是在score=score+1代码行的下面，如果运行的话没有任何问题，可是仔细想想，更新Label控件的操作放在checkAnswer（）方法中似乎不太合理，该方法应该是负责判断用户答题是否正确。所以关于更新UI的操作，放在一个独立的方法中更显合适。

在ViewController类中找到updateUI（）方法，添加下面的代码：

```
func updateUI() {
  scoreLabel.text = score
}
```

将score分数值赋值给scoreLabel的text属性，这样操作的意图是正确的，但是Swift编译器会报语法错误——不能将Int类型的值赋值给String类型（Cannot assign value of type'Int'to type'String？'）。

Swift是一种非常严格的程序设计语言，在赋值方面，我们只能把Int类型的值赋给Int类型的变量，把字符串赋值给字符串类型的变量。在上面的代码中，text是字符串类型，score是整型值，类型不一致就会导致语法错误。将之前的代码修改为scoreLabel.text="\（score）"，在双引号中使用\（）转意符将变量替换为字符串值，而且表达式的值也是字符串类型。

为了能够在需要的时候更新UI界面，要在用户切换到下一题的时候调用updateUI（）方法。

```
func nextQuestion() {
  if questionNumber <= 12 {
    questionLabel.text =
allQuestions.list[questionNumber].questionText

    updateUI()
  }else {
  ……
}
```

构建并运行项目，当答对一道题后分数会增加1。为了让答题分数显得更加人性化，可以在分数的前面加上文字说明。将代码修改为scoreLabel.text="分数：\（score）"。

接下来，我们需要解决的是为用户显示当前答题的进度情况。主要是通过更新progressLabel来实现。在updateUI（）方法中，添加下面的代码：

```
func updateUI() {
  scoreLabel.text = "分数: \(score)"
  progressLabel.text = "\(questionNumber + 1) / 13"
}
```

因为questionNumber的值是从0开始依次递增的，所以需要将其加1，以便显示其真正的题目序号。构建并运行项目，如图7-18所示，可以看到progressLabel显示了相关信息。

图7-18　progressLabel显示的相关信息

最后我们需要处理的是progressBar，也就是屏幕底部的黄色进度条，我们希望通过这个进度条可以图形化地显示出当前所答题目的进度。当然这还是与questionNumber有关，而且还需要知道屏幕的宽度值，并且将其13等分。每次进度条的长度变化都应该是questionNumber与其的乘积。

修改updateUI（）如下面这样：

```
func updateUI() {
  scoreLabel.text = "分数: \(score)"
  progressLabel.text = "\(questionNumber + 1) / 13"

  progressBar.frame.size.width = (view.frame.size.width / 13) *
questionNumber
}
```

首先通过progressBar的frame属性设置进度条在其父视图（也就是屏幕视图）中的大小和位置。frame中包含两个重要的属性：一个是size，用于确定进度条的宽度及高度；另一个是origin，通过其x和y属性来确定进度条在父视图中的位置。

此时的代码行会报错，如图7-19所示。意思是在乘号（*）的两侧不能一边是单精度值，一边却是整型值。因为view.frame.size.width的类型为CGFloat，所以乘号左边是单精度值，而questionNumber则是整型值，所以Swift编译器报错。

图7-19　单精度*整型数值报错

修复这个Bug非常简单，将代码修改为progressBar.frame.size.width=（view.frame.size.width/13）*CGFloat（questionNumber+1）即可。通过CGFloat（）函数将整型值转换为单精度值，乘号两边类型一致，报错消失。

构建并运行项目，进度条会根据当前回答题目的数量发生变化。当完成全部13道题目以后黄色进度条的宽度与屏幕宽度一致。当单击重新开始按钮以后，进度条又回到最初的宽度。

7.12　合并Objective-C代码到Swift

在本节中，我们会将ProgressHUD开源库合并到Quizzler项目之中。在整个过程中，你可能会遇到一些与这个库有关的黄色叹号警告。请不要担心这些警告，它们并不会影响应用程序的功能。但是，如果你想去除这些警告，可以在本节的最后找到答案。

首先在startOver（）方法中，每当用户重新开始做题的时候，要将score的值重置为0。

```
func startOver() {
  score = 0
  questionNumber = 0
  nextQuestion()
}
```

接下来我们需要做的一件事是，当用户每做完一道题的时候有一个反馈。目前的情况是在回答完一道题以后，Xcode的控制台会显示正确与否，这些信息只适用于程序员调试和捕获问题，但最终用户却无法看到。所以需要实现针对用户选择的反馈功能。

我们通过HUD（Head-Up Display，平视显示器）来实现该功能，如图7-20所示。HUD最早运用在航空器上，作为飞行的辅助仪器。平视的意思是指飞行员不需要低头就能够看到他需要的重要信息。平视显示器最早出现在军用飞机上，以降低飞行员需要低头查看仪表的频率，避免注意力中断以及丧失对状态意识的掌握。因为HUD的方便性以及能够提高飞行安全，民航机也纷纷跟进安装，之后汽车也开始安装该设备。

图7-20　飞机中的HUD应用

在Quizzler项目，我们希望在用户单击是/否按钮以后，会出现类似HUD方式的反馈信息，告诉用户是否回答正确。如果我们自己编写代码来实现该功能的话，可能需要一周的时间，为了提高效率，我们会使用代码库。这是一个第三方库，很多人在上面写了实现各种功能的成熟开源代码。这也就意味着他们编写的代码对于其他人来说都是可见的，都允许直接将这些代码集成到自己的项目之中。

如果有编程经验的程序员看到这里应该会知道，我所说的这个第三方库指的是GitHub。在GitHub网站上直接搜索relatedcode/ProgressHUD关键字，它是一个轻量级的HUD，如图7-21所示。

在ProgressHUD的详细页面中，向下滚动到OVERVIEW所显示的内容，我们可以看到这个轻量级的HUD是如何工作的，如图7-22所示。从左至右分别显示了通过ProgressHUD进行后台载入的状态，操作正确（成功）的状态和操作错误（失败）的状态。接下来我们将这段代码导入Quizzler项目之中。

图7-21　GitHub中的ProgressHUD项目

图7-22　ProgressHUD项目的介绍页面

实战：通过手动方式将ProgressHUD添加到项目中。

步骤1：单击GitHub里面的Clone or download连接，并解压缩下载后的文档，如图7-23所示。

图7-23　ProgressHUD项目解压缩后的文件夹

步骤2：打开ProgressHUD文件夹，其中还有一个ProgressHUD子文件夹，里面有ProgressHUD.bundle、ProgressHUD.h和ProgressHUD.m

三个文件，选择这三个文件，然后将其拖曳到Xcode的Quizzler项目之中，如图7-24所示。

图7-24　将三个文件拖曳到项目之中

步骤3：在弹出的选项面板中，确保Copy items if needed和Add to targets：Quizzler处于勾选状态。单击Finish按钮。

步骤4：此时Xcode会弹出标题——“是否希望配置一个Objective-C桥文件？（Would you like to configure an Objective-C bridging header？）”的对话框，如图7-25所示。这是因为ProgressHUD代码使用Objective-C语言编写，而我们的项目则是用Swift语言编写。所以Xcode通过桥文件允许我们在Swift项目中使用Objective-C代码。单击Create Bridging Header按钮。

图7-25　是否希望配置一个Objective-C桥文件?

接下来，我们需要告诉Swift项目哪个文件是Objective-C的代码。

在项目导航中选择Quizzler-Bridging-Header.h文件，在编辑窗口中键入一行Objective-C代码：#import"ProgressHUD.h"。这一步非常重要，否则Xcode不识别该Objective-C代码文件。

修改ViewController类中的checkAnswer（）方法如下面这样：

```
func checkAnswer() {
  let correctAnswer = allQuestions.list[questionNumber].answer

  if correctAnswer == pickedAnswer {

    ProgressHUD.showSuccess("正确")

    score = score + 1
```

```
  }else {
    ProgressHUD.showError("错误！")
  }
}
```

构建并运行项目，用户在每次作答的时候都会看到正确与否的反馈。如图7-26所示。

图7-26　导入ProgressHUD后的运行效果

利用第三方开源代码库，我们轻松地在项目中实现了反馈功能。

7.13　挑战：制作情商测试应用

在这一节中，我会抛砖引玉，引导大家制作一款企业招聘时的情商测试应用，用以帮助员工了解自己的情商状况。这个应用共有29道题，测试时间20分钟，最大EQ为154分。如果你已经准备就绪，请开始吧！

步骤1：下载初始化项目。

在GitHub中下载EQTest项目的源代码，如图7-27所示。

图7-27　EQTest项目源代码

在该项目中，故事板的用户界面布局已经设置完成。视图中的三个按钮与ViewController类建立了三个IBOutlet关联和一个IBAction关联。

之所以要为按钮建立IBOutlet关联，是因为在该项目中每道题目的作答内容并不一样，我们需要根据Question对象提供的选项内容来动态修改按钮的标题。

下面是ViewController.swift文件中的代码：

```
class ViewController: UIViewController {

  @IBOutlet weak var answerOneButton: UIButton!
  @IBOutlet weak var answerTwoButton: UIButton!
  @IBOutlet weak var answerThreeButton: UIButton!

  override func viewDidLoad() {
    super.viewDidLoad()

    // 通过下面的三行代码让按钮的外观变成圆角矩形
    answerOneButton.layer.cornerRadius = 25
    answerTwoButton.layer.cornerRadius = 25
    answerThreeButton.layer.cornerRadius = 25
```

```
    }

    // 用户单击按钮以后执行的方法
    @IBAction func answerPressed(_ sender: UIButton) {
    }
}
```

步骤2：制作一个Tag属性记录。

在Main.storyboard故事板中，将三个按钮的tag属性值从上到下依次设置为1、2、3。在之后的分值统计时，需要根据tag值计算每道题的得分，如图7-28所示。

图7-28　分别设置三个按钮的Tag值

Question类中的属性与Quizzler有所不同，当前类中一共有三个属性：字符串类型的questionText，用于存储题目；字符串数组类型的questionOption，用于存储题目选项，在测试题中既有三选一的情况，也有二选一的情况，我们在后面会根据情况进行处理；整型数组类型的questionScore，用于存储每种选项的分值。所以，questionOption和questionScore这两个数组的元素个数必须一致，否则会超出数组范围，导致应用程序崩溃。

```
class Question {
    let questionText: String
    let questionOption: [String]
    let questionScore: [Int]

    init(text: String, option: [String], score: [Int]) {
        questionText = text
        questionOption = option
        questionScore = score
    }
}
```

在QuestionBank类的初始化方法中你可以看到所有的题目都添加到了list数组之中。

```
class QuestionBank {
  var list = [Question]()

  init() {
    // 第1题
    var item = Question(text: "我有能力克服各种困难。", option:
["是的", "不一定",
"不是"], score: [6, 3, 0])
    list.append(item)

    // 第2题
    item = Question(text: "如果我能到一个新的环境，我要把生活安排
得：", option: ["和从前相仿", "不一定", "和从前不一样"], score: [6,
3, 0])
    list.append(item)

    ……
  }
}
```

步骤3：为题目的按钮选项设置标题。

在该项目中，nextQuestion（）方法的代码修改变动比较大。

```
func nextQuestion() {
  if questionNumber <= 28 {
    questionLabel.text =
allQuestions.list[questionNumber].questionText

    answerOneButton.isHidden = true
    answerTwoButton.isHidden = true
    answerThreeButton.isHidden = true

    for (index, option) in
allQuestions.list[questionNumber].questionOption.enumerated() {
      if index == 0 {
        answerOneButton.isHidden = false
```

```
        answerOneButton.setTitle(option, for:
UIControlState.normal)
      }else if index == 1 {
        answerTwoButton.isHidden = false
        answerTwoButton.setTitle(option, for:
UIControlState.normal)
      }else if index == 2 {
        answerThreeButton.isHidden = false
        answerThreeButton.setTitle(option, for:
UIControlState.normal)
      }
    }

    updateUI()
  }else {
  ……
```

因为在29道题中有一部分是二选一，还有一部分是三选一。所以，每次在屏幕上呈现新题的时候，先让三个选项按钮控件隐藏。然后通过for循环迭代出由questionNumber索引的特定题目选项。因为是数组类型的对象，所以可以通过数组类的enumerated（）方法得到每个元素的索引（index）和元素值（option）。

在循环中，如果questionOption数组有3个元素，则会让三个按钮都显示在屏幕上，并且使用setTitle（）方法设置每一个按钮的标题。它包含两个参数，第一个是文本信息，第二个state，即按钮的状态。其中state最常用的状态就是UIControlState.normal，代表按钮的正常状态。除此以外还有highlighted（高亮）、disabled（禁用）、selected（选中）、focused（焦点）等几种状态。

如果questionOption数组中只有两个元素，则在一开始的时候三个按钮均被设置为隐藏，然后通过两次循环只显示第一个和第二个按钮选项，而第三个按钮始终会处于隐藏状态。

步骤4：计算答题分值。

当用户单击题目选项的时候会执行answerPressed（）方法，通过按钮的tag属性来确定用户选择的是哪个选项。

```
@IBAction func answerPressed(_ sender: UIButton) {
  pickedAnswer = sender.tag - 1
  checkAnswer()
  questionNumber += 1
  nextQuestion()
}
```

在确定选项以后，接下来则是计算分值，让questionNumber加1，在屏幕上显示下一道题目。我们先来看看checkAnswer（）方法：

```
func checkAnswer() {
  score = score +
allQuestions.list[questionNumber].questionScore[pickedAnswer]
}
```

在checkAnswer（）方法中，我们会针对用户回答的选项的位置来获取分值，然后将其累加。

步骤5：更新UI。

在每切换一道新的题目的时候就会执行updateUI（）方法，在该方法中我们只需将之前的13修改为29即可。

```
func updateUI() {
  progressLabel.text = "\(questionNumber + 1) / 29"

  progressBar.frame.size.width = (view.frame.size.width / 29) *
CGFloat(questionNumber + 1)
}
```

步骤6：呈现EQ测试结果。

呈现EQ测试结果的代码是在nextQuestion（）方法中，如果questionNumber的数值超过28，则需要根据score变量的值来呈现测试结果。

```
func nextQuestion() {
  if questionNumber <= 28 {
    ……
  }else {
    var title = ""
    var message = ""
    if score < 70 {
      title = "你的EQ较低"
      message = "你常常不能控制自己，你极易被自己的情绪所影响。很多时候，你轻易被击怒、动火、发脾气，这是非常危险的信号 —— 你的事业可能会毁于你的暴躁。对此最好的解决办法是能够给不好的东西一个好的解释，保持头脑冷静使自己心情开朗。"
    }else if score >= 70 && score < 109 {
      title = "你的EQ一般"
      message = "对于一件事，你不同时候的表现可能不一，这与你的意识有关，你比前者更具有EQ意识，但这种意识不是常常都有，因此需要你多加注意、时时提醒自己。"
    }else if score >= 110 && score < 129 {
      title = "你的EQ较高"
      message = "你是一个快乐的人，不易恐惊担忧，对于工作你热情投入、敢于负责，你为人更是正义正直、同情关怀，这是你的长处，应该努力保持。"
    }else if score >= 130 {
      title = "你就是个EQ高手"
      message = "你的情商高超不但是你事业的助手，更是你事业有成的一个重要前提条件。"
    }

    let alert = UIAlertController(title: title, message:
message, preferredStyle: .alert)

    let restartAction = UIAlertAction(title: "重新开始", style:
.default, handler:
{ (alertAction) in self.startOver() })

    alert.addAction(restartAction)
    present(alert, animated: true, completion: nil)
  }
}
```

构建并运行项目，运行效果如图7-29所示。

图7-29　EQTest项目的运行效果

第8章　iOS的自动布局和设置约束

这一章我们主要学习如何在故事板中布局UI控件和设置约束，以便让我们仅仅通过一种设计布局，就能在各种尺寸的iOS设备屏幕和方向上显示出完美的界面布局。

还记得之前我们在Dicee项目中的界面布局吗？纵向可以完美显示，而转换到了横向则相当糟糕，如图8-1所示。这是因为在横向方向，每个UI控件的大小和位置还是沿用之前纵向的设置。所以当你在创建一个项目的时候，一定要考虑是否为其设计横向界面布局。

另外，如果运行在不同设备上，如图8-2所示，界面布局也是相当糟糕。

图8-1　纵向和横向显示的Dicee应用的界面效果

图8-2　不同尺寸屏幕上Dicee的显示效果

正如你知道的，iPhone设备具有不同的屏幕尺寸：

·对于iPhone 5/5s/SE，纵向模式的屏幕由水平320点（或640像素）和垂直568点（或1136像素）组成。

·对于iPhone 6/6s/7/8，屏幕由水平375点（或750像素）和垂直667点（或1334像素）组成。

·对于iPhone 6/6s/7/8Plus，屏幕由水平414点（或1242像素）和垂直736点（或2208像素）组成。

·对于全新的iPhone X，屏幕由水平375点（或1125像素）和垂直812点（或2436像素）组成。

·对于iPhone 4s，屏幕由320个点（或640个像素）和480个点（或960个像素）组成。

这里所提供的屏幕分辨率单位为什么是点而不是像素呢?

早在2007年，苹果就推出了3.5英寸屏幕，分辨率为320×480，即水平320像素和垂直480像素的初代iPhone。之后在iPhone 3G和iPhone 3GS上保留了这个屏幕分辨率。显然，如果你当时正在构建一个应用程序，一个点就是对应一个像素。后来，苹果推出了带有视网膜显示屏的iPhone 4。屏幕分辨率翻倍至640×960像素。所以一个点对应视网膜显示屏的两个像素。

以点为单位的坐标系统使程序员的开发变得轻松。无论屏幕分辨率如何变化（例如，分辨率再次翻倍至1280×1920像素），我们仍然处理的是基于像素的点数（即iPhone 4/4s的320×480或iPhone 5/5s/SE的320×568）。点和像素之间的转换自动由iOS处理。

如果不使用自动布局，则在故事板中放置按钮的位置是固定的。也就是说，我们硬性指定按钮的origin属性。在Dicee项目中，“掷骰子”按钮的frame.origin被设置为（120，501）。因此，无论你使用的是4英寸、4.7英寸还是5.5英寸的模拟器，iOS都会在指定的位置绘制按钮。图8-3显示了不同设备上frame的起始位置。这就解释了为什么“掷骰子”按钮只适合在iPhone 6/7/8上显示，而在其他iOS设备上的效果就非常糟糕。

图8-3　不同屏幕尺寸的屏幕上显示固定位置的按钮

很显然，我们希望应用程序在所有iPhone机型上都有完美的布局，并且可以纵向和横向显示，这就是我们学习自动布局的原因。目前，针对布局的问题有两种解决方案：一是通过代码的方式来确定按钮（Button）、标签（Label）、图像（ImageView）等界面元素在屏幕上的大小与位置；二是通过自动布局（Auto Layout）和设置约束的方式，比如设置一个按钮总是定位在屏幕的正中央位置，而不管屏幕的大小与方向。

8.1 通过代码定位UI元素

为了能够更好地说明如何使用代码来确定UI控件的大小与位置，需要使用Single View Application模板创建一个新的项目，在项目导航中选择ViewController.swift文件。

让我们将注意力集中到ViewController类的viewDidLoad（）方法，在应用程序启动，各种相关UI控件被载入设备的内存之后，控制器就会调用该方法，我们可以在该方法中通过代码的方式设置指定UI控件的大小与位置，以及设置某些数据的初始值。

修改viewDidLoad（）方法如下面这样：

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

  let square = UIView(frame: CGRect(x: 0, y: 0, width: 50,
height: 50))
  square.backgroundColor = UIColor.red
  self.view.addSubview(square)
}
```

通过UIView的初始化方法创建了一个视图对象，该初始化方法带有一个参数frame，frame用于指定一个矩形在屏幕上的大小与位置。为了确定这个frame，我们使用CGRect（）函数生成一个矩形，要想确定一个矩形我们需要四个方面的信息：矩形的左上角在其父视图的x和y的位置，以及矩形的宽度与高度值。

新创建的矩形在其父视图（0，0）点的位置，请记住：父视图的坐标体系也是从左上角开始的。所以这个视图的位置是手机屏幕左上角开始的长和宽均为50的矩形。为了在模拟器中可以看到这个视图区域，通过设置square的backgroundColor属性将其背景色设置为红色。

最后，我们使用view的addSubview（）方法将square添加到当前控制器的视图之中，这也就意味着square的父视图就是控制器中的View。还记得每个控制器默认都有一个视图（View）吗？它就是控制器中的根（顶级）视图，所有的UI控件或子视图都被添加到其内部，如图8-4所示。

构建并运行项目，可以发现屏幕的左上角有一个红色的矩形，如图8-5所示。

为了可以更好地理解frame的四个属性，你可以随意修改x、y、width、height的值，从中体会不同数值带来的不同效果。例如将上面的代码修改为let square=UIView（frame：CGRect（x：50，y：50，width：100，height：100））。

图8-4　控制器中的根视图

图8-5　项目的运行效果

除了UIView以外，按钮（UIButton）、图像（UIImageView）、标签（UILabel）等控件都可以用这种方式创建，只需要将UIView关键字替换为相应的UI控件类即可。

但是，当这段代码在iPhone SE、iPhone 8或iPhone 8Plus模拟器中运行的话，这个square在屏幕上的显示效果不尽相同。虽然矩形的大小与位置还是代码中的数值，但是在不同分辨率的屏幕上所显示的位置就有了一些出入。如果这个矩形是应用程序的Logo，或者是非常重要的选项按钮，则会在不同的iPhone设备显示在不同的位置上，用户体验会非常糟糕。

读到这里你是否意识到，如果我们确定UI控件的大小与位置越是精确，对于处理不同屏幕尺寸的布局就越是麻烦。要如何解决这个问题

呢？可以通过square与其父视图的位置关系来确定布局。

例如我们想将square定位到屏幕的中央，而不管是什么型号的iPhone。对于frame的四个属性，我们可以将它们设置为下面这样：

```
x = self.view.frame.width / 2 - square.frame.width / 2
y = self.view.frame.height / 2 - square.frame.height / 2
width = 100
height = 100
```

这里，我们让square的x属性等于其父视图（屏幕视图）宽度的一半，再减去自己宽度的一半，相当于向左移动50个点，这样square就会水平居中。y属性等于父视图高度的一半，再减去自己高度的一半，相当于向上移动50个点，这样square就会垂直居中。但是真正的代码应该像下面这样：

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

  let square = UIView(frame: CGRect(x: 0, y: 0, width: 100,
height: 100))

  let x = self.view.frame.width / 2 - square.frame.width / 2
  let y = self.view.frame.height / 2 - square.frame.width / 2

  square.frame.origin = CGPoint(x: x, y: y)

  square.backgroundColor = UIColor.red
  self.view.addSubview(square)
}
```

构建并运行项目，效果如图8-6所示。

图8-6　项目的运行效果

8.2 自动布局

这一节将会向大家介绍如何在Interface Builder中使用自动布局（Auto Layout）特性，以及通过可视化方式创建约束。

自动布局是基于约束的布局系统，它允许程序员创建一个自适应的用户界面，用以在各种屏幕尺寸和方向上完美显示用户界面布局。作为初学者可能你会觉得非常难学，甚至某些开发者宁愿编写大量的程序代码去设置用户界面控件的大小与位置，而故意回避去使用它。但是，请相信我所说的：如果你现在还不学习使用它的话，最终你将会被淘汰。

在十年之前iPhone第一代首次发布的时候，只有一个屏幕尺寸——3.5英寸。后来有了4英寸的iPhone 4。在2014年9月，苹果推出了4.7英寸的iPhone 6和5.5英寸的6Plus。现在，iPhone的屏幕尺寸包括：3.5英寸、4英寸、4.7英寸、5.5英寸和5.8英寸的屏幕。当你在设计应用程序界面的时候，必须支持所有这些屏幕尺寸。如果应用程序要同时支持iPhone和iPad（也称为通用应用程序），则需要确保该应用程序适合更多的屏幕尺寸，包括7.9英寸、9.7英寸、10.5英寸和12.9英寸。如果不使用自动布局，那么创建支持所有屏幕分辨率的应用程序将会非常困难。

什么叫作"基于约束的布局"呢?

请考虑之前的square视图，如果要将其置于视图的中心，应该如何准确描述其位置呢？你可能会用这样的方法描述：无论屏幕的分辨率和方向如何，square应该水平和垂直居中。

这里实际上定义了两个约束：

·垂直居中

·水平居中

这些约束表达了界面中视图的布局规则。

自动布局是通过各种约束实现的。虽然我们用文字描述了约束条件，但是自动布局中的约束条件是以数学形式表示的。例如，如果要定义square的位置，你可能想要说“square的左边缘应该是它父视图的左边缘30点”。这将转换为square.left=（fatherView.left+30）。

幸运的是，我们并不需要通过代码的方式来描述这些约束条件，而是可以直接使用Interface Builder来创建所描述的约束。

现在让我们来看看如何在Interface Builder中定义布局约束来居中square视图。

8.2.1　在界面生成器中实时预览布局效果

继续编辑项目中的Main.storyboard文件。在将约束添加到用户界面之前，让我先介绍一个在Xcode中非常方便和实用的功能。

你可以在模拟器中测试应用程序的用户界面，以便查看它在不同屏幕尺寸下的布局。但是，Xcode在Interface Builder中为开发人员提供了一个配置栏（configuration bar）来预览用户界面的布局效果。

在默认情况下，Interface Builder被设置为在iPhone 8（4.7英寸）上预览用户界面。要查看应用程序在其他iPhone设备上的显示效果，需要单击View as：iPhone 8按钮以显示配置栏，然后选择要预览的iPhone/iPad设备进行测试，如图8-7所示。你还可以改变设备的方向，查看最终的显示效果。

图8-7　Xcode的配置栏

配置栏是从Xcode 8开始被引入的一个很棒的功能。

8.2.2　使用自动布局将square居中

现在让我们继续讨论自动布局，Xcode提供了两种方法来定义自动布局的约束：

·自动布局栏

·控制拖动

我们将在这里演示这两种方法，让我们从自动布局栏开始。在Interface Builder编辑器的右下角，你可以从自动布局栏找到5个按钮，使用这些按钮来定义各种类型的布局约束并解决布局的问题，如图8-8所示。

图8-8　自动布局栏中的5个按钮

自动布局栏中的每个按钮都有特定的功能：

·对齐（Align）：创建对齐约束，例如对齐两个视图的左边缘。

·添加新的约束（Add new constraints）：创建空间约束，例如定义UI控件的宽度。

·解决自动布局问题（Resolve auto layout issues）：解决布局问题。

·堆栈（Stack）：将视图嵌入堆栈视图（stack view）中。堆栈视图是从Xcode 7引入的新功能。

·更新帧（Update frames）：参照给定的布局约束更新frames的位置和大小。

正如前面所说的，为了居中square视图，必须定义两个约束：水平居中和垂直居中。这两个约束都是关于当前控制器视图的。

为了创建约束，我们使用Align按钮。首先在Interface Builder中选择square视图，然后在布局栏中单击Align图标。在弹出式菜单中，同时选中“Horizontally in Container”和“Vertically in Container”选项，然后单击“Add 2Constraints”按钮，如图8-9所示。

提示　你可以通过快捷键command+0隐藏项目导航栏，这样会释放更多的屏幕空间，便于我们将注意力放在用户界面的设计上。

你现在会看到一组红色的约束线，如图8-10所示。如果在文档大纲视图中展开“Constraints”选项，则会发现该视图的两个新约束。这两个约束会确保按钮始终位于View的中心。或者，你可以在“Size Inspector”检查器中查看这些约束。

红色约束线代表square视图的约束设置还不完整，Swift无法根据当前仅有的这两个约束来确定square视图的大小和位置。确定选中square视图，单击添加新的约束按钮，并在弹出菜单中勾选Width和Height，然后单击Add 2Constraints按钮。此时square的约束线均变成了蓝色，如图8-11所示。

图8-9　为Square创建水平和垂直约束

图8-10　查看square的两个约束

图8-11　为square添加了四个约束

提示　当用户界面控件的约束布局配置正确且不存在歧义时，它的约束线将呈现蓝色。

构建并运行项目，无论屏幕的大小和方向如何，square都在屏幕上居中显示。

8.2.3　解决布局约束的问题

目前，我们创建的布局约束是完美的，但有些时候情况并非总是如此。在用户界面控件数量较多的视图中，我们往往会忽略掉某些元素的约束，从而导致整个界面布局的混乱。然而，Xcode足够智能，可以帮助我们检测各种约束问题。

现在，尝试将square视图拖曳到屏幕的左下角。Interface Builder立即检测到一些布局问题，相应的约束线变成橙色，表示用户界面控件放错了位置，如图8-12所示。

图8-12　设置的约束与用户界面元素实际位置不匹配

当你创建了不明确或有冲突的约束时，会出现自动布局问题。比如我们要让square在视图中垂直和水平居中，但现在却位于视图的左下角。因此Interface Builder使用橙色线条来指明布局问题。虚线框则表示square的预期位置。

当有任何布局问题时，“文档大纲”视图会显示一个指示箭头（红色或者橙色的）。现在，单击指示箭头查看问题列表。对于这样的布局问题，Interface Builder足够智能，可以帮我们相应解决。

单击问题旁边的指示器图标，在弹出的窗口上会显示一些解决方案。在这种情况下，选择“更新帧（Update frames）”选项并单击“修复错位（Fix Misplacement）”按钮。square将被移动到视图的中心位置，如图8-13所示。

图8-13　利用Xcode修复约束的问题

或者，你可以简单地单击布局栏的“更新Frames”按钮来解决当前的问题。

上面的这个布局问题是手动触发的，目的只是想演示如何找到问题并修复它们。若你在后面章节中进行练习时遇到类似的布局问题，你就可以知道如何快速解决该类布局问题了。

8.2.4　另一种预览故事板的方式

虽然可以使用配置栏来预览应用程序用户界面，但Xcode还为程序员提供了备用预览功能，可以同时在不同设备上预览用户界面。

在Interface Builder中，打开“助手”弹出式菜单的Preview（1）。按住option键，然后单击Main.storyboard（Preview），如图8-14所示。

图8-14　打开故事板的预览功能

Xcode将在助理编辑器模式中显示应用程序的用户界面的预览效果。默认情况下，它会显示iPhone 8屏幕的预览效果。你可以单击助手编辑器左下角的“+”按钮来添加其他iOS设备（例如iPhone SE/8Plus）以进行预览。如果你想要了解横向屏幕的外观，只需单击旋转按钮即可。预览功能对于设计应用程序的用户界面非常有用。你可以对故事板进行更改（例如，向视图添加一个按钮），并查看用户界面控件在各种设备上的显示效果，如图8-15所示。

图8-15　在Interface Builder中查看不同尺寸屏幕的界面布局

提示　当你在预览助手窗口中添加了很多设备以后，Xcode可能无法同时将所有设备尺寸预览放入屏幕。这时可以使用触控板，用两根手指向左或向右滑动来浏览预览效果。如果你用的是滚轮鼠标，只需按住shift键水平滚动。

8.2.5　添加一个标签

现在你已经对自动布局和预览功能有了一些了解，接下来让我们在视图的右下角添加一个标签，看看如何定义标签的布局约束。iOS中的标签通常用于显示简单的文本和消息。

在Interface Builder编辑器中，从对象库中拖出一个标签并将其放置在视图的右下角附近。双击标签并将其更改为“欢迎使用自动布局”或任何你想要显示的文本信息。然后按command+=自动调整标签到合适的大小，如图8-16所示。

图8-16　在视图中添加一个新的Label

如果再次打开预览助手，你应该会看到用户界面有所改变。因为没有为标签定义任何布局约束，所以在不同设备上会看到这个标签呈现在不同位置上，如图8-17所示。

图8-17　在视图中添加一个新的Label

怎么解决这个问题？显然，我们需要为标签设置一些约束。问题是：我们应该添加什么样的约束？

我们试着用文字来描述标签的要求。你可能像这样描述它：标签应该放在视图的右下角。

虽然描述正确但还不够精确。描述标签位置的更精确的方法是这样的：标签位于距视图右边缘0点，距离视图底部20点的位置。

当你准确地描述一个控件的位置时，可以很容易想出布局约束。在这里，标签的约束应该是：

·标签距离视图的右边缘0点。

·标签距离视图底部20点。

在自动布局中，我们将这种约束称为间距约束。要创建这些间距约束，可以使用布局按钮的“添加约束”按钮。但是这次我们将使用按住鼠标右键并拖曳的方法来创建自动布局约束。在Interface Builder中，你可以通过该方法，将某个用户界面控件拖曳到自身或者拖曳到另一个界面控件上，用于创建全新的约束。

要添加第一个间距约束，请在标签上按住鼠标右键并向右拖动鼠标，直到视图变为蓝色突出显示。然后释放鼠标按钮，你会看到一个弹出式菜单，显示约束选项列表。选择“Trailing Space to Safe Area”，这样就从标签右边缘到视图的右边缘添加了一个间距约束，如图8-18所示。

图8-18　为Label创建间距约束

在文档大纲视图中，你应该会看到新的约束。Interface Builder现在以红色显示约束线，表示存在一些缺失的约束。这很正常，因为我们还要继续创建第二个约束。

现在，使用同样的方法从标签上将鼠标拖曳到视图的底部。松开鼠标并在快捷菜单中选择“Bottom Space to Safe Area”。这会从标签底部到视图底部布局指导线创建一个间距约束，如图8-19所示。

图8-19　继续为Label创建间距约束

一旦你添加了两个约束，所有的约束行都应该是纯蓝色的。在预览用户界面或在模拟器中运行应用程序时，标签应在所有屏幕尺寸上都正确显示，甚至在横向模式下也能正常显示，如图8-20所示。

图8-20　为Label创建两个间距约束以后的效果

至此，你已经正确定义了约束。但是，你可能会注意到文档大纲中的黄色指示符。如果单击该指示符便会发现与本地化有关的布局警告。

这是为什么呢？在Xcode 9中的Interface Builder有一个新的功能。当前的程序项目只支持英语。我们定义的布局约束完美地适用于英语。但是如果这个项目需要支持其他的语言呢？目前的布局约束是否能够适用于从右到左的语言（例如阿拉伯语）？

在Xcode 9中Interface Builder将检查你的布局约束，并检查它们是否适合所有的语言。如果发现有问题，则会发出定位警告。要解决这个问题，你可以选择第二个选项来添加前导约束，如图8-21所示。

图8-21　在大纲导览视图中修复问题约束

8.2.6　安全区域

在文档大纲中，你是否注意到一个名为“安全区域（Safe Areas）”的条目？不知道你还记不记得我们之前定义的间距约束也与安全区域有关。我们定义过两个间距约束：

·将空间拖到安全区域（Trailing space to Safe Area）

·底部空间到安全区域（Bottom space to Safe Area）

那么，安全区又是什么呢？首先，苹果从Xcode 9开始引入了安全区域的概念，以取代之前Xcode中使用的顶部和底部的参考线。

在文档大纲中选择安全区域条目，蓝色区域便是安全区域。安全区域实际上是一个布局参考，代表控制器中视图的一部分，不被状态栏和其他内容所遮掩。如图8-22所示，安全区域是除状态栏之外的整个视图。

图8-22　阴影部分为安全区域

安全区域布局参考可以帮助程序员更轻松地处理布局约束，因为安全区域会在导航栏或其他内容覆盖视图时自动更新。

我们将之前的square视图的水平和垂直的中央对齐约束修改为与Safe Areas相关的，如图8-23所示，如果视图中没有导航栏或标签栏，则安全区域是除状态栏之外的整个视图。

图8-23　不同界面中的安全区域大小不同

如果视图中包含导航栏，则无论是使用iOS 11中的标准标题还是大标题，安全区域都会自动调整。square会放置在导航栏下方居中的位

置。因此，UI对象只会相对于安全区域受到约束限制，即使将导航或标签栏添加到用户界面之中，你的布局也是正确的。

8.2.7　编辑约束

“欢迎使用自动布局”标签现在距安全区的拖尾锚点有16个点。如果你想增加标签和视图右侧之间的距离怎么办？Interface Builder提供了一种编辑约束常量的简便方法。

你可以在文档大纲视图中选择约束或直接在设计区域选择约束。在Attributes Inspector检查器中可以找到此约束的属性，包括关系、常数和优先级。常数现在设置为16。你可以将其更改为30以增加一些额外的空间，如图8-24所示。

图8-24　修改Label的尾部间隔约束值

另外，你也可以双击约束线，在弹出的约束设置面板中编辑其属性，如图8-25所示。

图8-25　通过双击约束线的方式修改约束值

8.3　自动布局实战——设置约束

还记得之前所做的Dicee项目吗？当时我们并没有对其进行自动布局或添加约束的处理。因此在不同iPhone设备或屏幕方向上都显示得不尽完美。本节我们将会利用所学的自动布局技能使其完美呈现在各种屏幕尺寸和方向的设备上。

你可以直接从GitHub中下载Dicee项目，或者使用自己之前做好的Dicee项目。打开项目以后，在项目导航中选择Main.storyboard文件，并确定在Interface Builder配置栏中的默认设备是iPhone 8（wC hR）。

目前的界面布局在4.7英寸的屏幕上面会完美显示，但是在其他iPhone设备上的显示效果就非常糟糕。因此在本节中我们要让这个界面布局既能够适应大尺寸的屏幕，又能够适应一些小尺寸的屏幕。

整个Dicee用户界面可以分成上中下三个部分，上面的部分是应用程序的Logo图标，中间的部分是两个骰子，下面的部分则是“掷骰子”按钮。

我们先来搞定中间的部分，为了可以更好地布局这2个骰子，我们将其放到一个容器（子视图）之中。

实战：设置Middle Container的约束。

步骤1：从对象库中拖曳一个View到控制器视图之中，然后将2个骰子拖曳到其内部，使2个Image View成为刚刚新添加View的子视图。在Size Inspector中将新添加视图的width属性设置为295，height属性设置为120。

提示　可以在文档大纲视图中，使用鼠标直接拖动相关条目到目标视图的内部。如图8-26所示。

图8-26　将按钮拖曳到新建的View中

步骤2：在文档大纲视图中将新添加的视图的名字修改为Middle Container。然后将2个骰子的位置调整在Middle Container容器的两边。如图8-27所示。

图8-27　调整骰子在View中的位置

步骤3：为Middle Container添加4个约束，分别是容器的width、height以及容器在屏幕的水平和垂直方向居中对齐。确保选中该容器，单击Interface Builder右下角布局栏的Add New Constraints按钮，勾选Width（295点）和Height（120点）两个约束，然后单击Add 2Constraints按钮以固定视图容器的尺寸，如图8-28所示。单击布局栏中的Align按钮，勾选Horizontally in Container和Vertically in Container两个约束，然后单击Add 2Constraints按钮以固定视图容器的位置，如图8-29所示。

图8-28　调整新View的宽度和高度

提示　此时不管我们选择哪种屏幕尺寸的iPhone，Middle Container视图都会以相同的大小呈现到屏幕的中央位置。

图8-29　调整新View的水平和垂直约束

实战：为背景图设置约束。

虽然Middle Container的位置和大小已经布局完成，但是红色背景图在不同尺寸的屏幕上显示不尽完美，尤其是在iPhone 8Plus上面会有白色空白出现。我们需要将Image View的上下左右边缘与屏幕视图的上下左右边缘重合，也就是将它们相应的边缘间距设置为0。

选中红色背景的Image View控件，单击布局栏的Add New Constraints按钮，在约束面板的上半部分，有代表四个方向的工字形线段，当前它们的颜色都是灰色，单击使其变为红色。当工形线变为红色以后，代表当前所选中的Image View（呈现背景图的Image View），与其距离最近的界面控件（当前项目中是其父视图）在上下左右四个方向的距离为0，即边缘与屏幕边缘重合。单击Add 4Constraints按钮让四个约束生效，如图8-30所示。

图8-30　调整新View的水平和垂直约束

现在，红色背景图在任何尺寸的屏幕上都会全屏显示。

实战：为骰子设置约束。

在Middle Container中的2个骰子也需要添加相应的约束，选中左侧的骰子，在Add New Constraints面板中勾选Width和Height，以及将其顶部和左侧边缘工形线点亮，这样便可以确定左侧骰子的大小与位置。同样，选中右侧的骰子，在Add New Constraints面板中勾选Width和Height，以及将其顶部和右侧边缘工形线点亮。现在Middle Container及其内部的2个骰子的约束均添加完成，如图8-31所示。

图8-31　为骰子添加相应的约束

在配置栏中将屏幕方向修改为横向，骰子部分依然会完美显示，如图8-32所示。

图8-32　横屏模式下的骰子显示效果

实战：为Logo和“掷骰子”按钮添加约束。

实际上，我们需要让Logo和掷骰子按钮分别在所属部分的居中位置，因此需要先将这两个用户界面控件分别放到容器之中。

步骤1：从对象库中拖曳一个View到故事板中，让其左上角与屏幕的左上角对齐，右边缘与屏幕的右边缘重合，底部靠近Middle Container的顶部。

步骤2：在文档大纲视图中将新添加的View名称修改为Top Container，然后将View中的diceeLogo调整到该容器内部。

步骤3：单击布局栏中的Add New Constraints按钮，在面板的上半部分中将四个方向的工形线全部按亮。然后分别单击四个常数值右侧的下三角确认Top Container的顶部与View顶部的间距为0，其左侧边缘与屏幕Safe Area的左边缘间距为0，其右侧边缘与屏幕Safe Area的右边缘间距为0，其底部与Middle Container的顶部间距为0，如图8-33所示。

图8-33　为Top Container添加约束

步骤4：从对象库再拖曳一个View到故事板中，让其左下角与屏幕的左下角对齐，右边缘与屏幕的右边缘重合，顶部靠近Middle Container的底部。

步骤5：在文档大纲视图中将新添加的View名称修改为Bottom Container，然后将View中的“掷骰子”按钮调整到该容器中。

步骤6：确定选中Bottom Container容器，单击Add New Constraints按钮，在面板的上半部分中将四个方向的工形线全部按亮。然后确认Bottom Container的顶部与Middle Container底部的间距为0，其左侧边缘与屏幕Safe Area的左边缘间距为0，其右侧边缘与屏幕Safe Area的右边缘间距为0，其底部与View的底部间距为0。

步骤7：为了可以清晰地分辨上中下三部分区域，将所添加的三个视图的背景色修改为不同的颜色。

在Interface Builder的配置栏中，我们可以随意选择不同屏幕尺寸的iPhone设备，此时这三个部分会被均匀布局到屏幕上。

步骤8：选中Logo图标，为其添加Width和Height两个约束，并添加“Horizontally in Container”和“Vertically in Container”两个约束。对“掷骰子”按钮也做同样的操作。最后将三个容器视图的背景色设置为无色。

步骤9：在配置栏中选择不同的iPhone设备，可以看到布局根据所设置的约束完美显示在屏幕上，如图8-34所示。

图8-34　为Top Container添加约束

8.4　挑战自动布局

如果你自认为已经可以熟练运用自动布局特性为用户界面控件添加合适的约束的话，接下来就将进入自我挑战时间。在为读者提供的初始化项目中，故事板里没有任何的UI控件，仅仅是在Assets.xcassets文件中为读者提供了两个图像素材文件：applePad和appleWatch。

通过自动布局特性，我们想要达到的最终布局效果为：不管是纵向还是横向屏幕方向，applePad和appleWatch图像分别占据在控制器视图上下两个部分的容器之中，并且无论屏幕尺寸如何变化，这两个容器的高度均相等，两张图像也始终位于容器的中央位置。

最终的运行效果如图8-35所示。

图8-35　不同屏幕尺寸的界面布局效果

实战：挑战提示。

步骤1：在GitHub网站上搜索“liumingl/Auto Layout Practice iOS11”关键字，下载初始项目代码，解压缩项目到目标文件夹，并打开项目。

步骤2：在项目导航中选择Main.storyboard文件，从对象库拖曳2个Image View到屏幕之中，将其中一个Image view的Image属性设置为applePad，将另一个设置为Apple Watch。并将这2个Image View的Content Mode设置为Aspect Fit。

步骤3：从对象库拖曳2个UIView对象到视图上，一个充满屏幕的上半部分，并将名称修改为Top View，另一个充满屏幕的下半部分，并将名称修改为Bottom View。为这2个容器分别创建约束，其中Top View要与控制器的View创建顶部、左侧和右侧的约束，其底部要与Bottom View的顶部创建间距为0的约束。Bottom View要与控制器的View创建底部、左侧和右侧的约束。

步骤4：此时会有红色约束线出现，因为Interface Builder目前还无法计算出每个容器的具体高度是多少。同时选中上下两个容器，然后在Add New Constraints面板中勾选Equal Heights，并单击Add 1Constraint按钮，如图8-36所示。

图8-36　通过约束让两个视图的高度相等

步骤5：分别在两个容器中为两个Image View添加相应的约束，固定它们的高度与宽度值，并设置水平和垂直方向居中。最终效果如图8-37所示。

图8-37　项目的最终运行效果

8.5　在自动布局中使用堆叠视图

在本节中我们将利用堆叠视图（Stack View）再结合约束，对多个用户界面控件进行定位与对齐。在不同屏幕方向的情况下，利用堆叠视图可以简化对用户界面的布局设置。

在之前版本的Xcode中，我们只能通过设置容器，并为容器设置相关约束来进行多用户界面控件的布局。在最新的Xcode中，苹果通过堆叠视图让程序员的设计流程更加方便、简单。

在设计应用程序界面的时候，我们往往需要让多个界面元素或水平/垂直平均分布在某个容器之中。例如macOS系统中的计算器，在整个视图中一共有19个按钮和1个标签控件用于显示输入的内容，如图8-38所示。

图8-38　macOS中的计算器应用程序界面

如果分别为这20个界面元素创建约束的话，大致需要80个左右。光是这些约束就会让你在Interface Builder中眼花缭乱。接下来，我们将会使用堆叠视图来快速搭建计算器的用户界面。

实战：创建计算器的用户界面。

步骤1：创建一个全新的Xcode项目，选择Single View App，Product Name设置为Auto Layout Calculator。

注意　本节主要是通过堆叠视图创建计算器应用程序的用户界面，有兴趣的话可以在完成界面布局以后，自己尝试添加相关代码。

步骤2：选中Main.storyboard文件，从对象库拖曳1个按钮对象到视图之中，在Size Inspector中将按钮的Width和Height属性均设置为50。在Attributes Inspector中将Background属性设置为蓝色，Text Color属

性设置为白色，Font属性设置为Bold 30，最后将按钮的Title修改为1。

为该按钮对象创建与ViewController类的IBAction关联，方法名称设置为buttonPressed（）。

接下来，我们将会复制这个设置好的按钮，并将它们充满屏幕的大部分空间。

步骤3：选中当前的按钮，按住Option键后拖曳该按钮，这样便复制了1个新的按钮。重复这样的操作再复制2个，此时视图中一共有4个按钮，如图8-39所示。

提示　被复制出来的新按钮不仅保留原有按钮的所有属性设置，而且之前设置的IBAction方法也会被保留。

步骤4：同时选中这4个按钮，重复前几步操作，再添加16个这样的按钮，如图8-40所示。

图8-39　通过复制创建四个计算器按钮

图8-40　通过复制创建二十个计算器按钮

目前的用户界面中一共有四列五行的计算器按键，在没有设置任何约束的情况下，在各种设备和方向的屏幕上的显示效果并不理想。

步骤5：选中第一行的四个按键，单击布局栏中的Embed in Stack按钮，或者在菜单栏中选择Editor/Embed In/Stack View，如图8-41所示。

此时的4个按键已经横向水平合并到了一起，但它们之间并没有间隔，而且还会有一小部分的重叠。接下来，我们就来修复这个问题。

步骤6：在故事板中选中刚刚创建好的Stack View，在Attributes Inspector中将Distribution设置为Fill Equally，这样可以保证堆叠视图中的每个界面元素在水平方向上宽度相等。继续保持Stack View的选中状态，单击Add New Constraints按钮，将左右两侧的工形线点亮，让Stack View的左右边缘与Safe Area的边缘有10个点的间隔，如图8-42所示。

图8-41　为第一行的4个按钮建立堆叠

图8-42　为第一行的Stack View设置约束

提示　如果在故事板中选择某个视图比较困难的话，可以直接在文档大纲视图中选取该视图。

步骤7：在Attributes Inspector中，将Spacing属性设置为10。此时，4个按键会等宽等距地布局在水平堆叠视图之中，每个按键之间会有10个点的间隔，如图8-43所示。

接下来，我们可以继续按照第一行的操作方法，将下面的按键以行为单位内嵌到四个独立的堆叠视图之中。不过，既然已经做好的第一行，我们就可以直接复制它。

步骤8：删除除第一行以外的所有按键，然后将第一行的堆叠视图重新复制4遍。再选中所有的堆叠视图，单击布局栏中的Embed in Stack按钮。因为所有的堆叠视图呈垂直方向排列，所以Xcode智能地将最后创建的堆叠视图的Axis属性设置为垂直方向，如图8-44所示。

图8-43　对第一行的Stack View设置相关的属性

图8-44　Xcode智能的将堆叠视图设置为垂直方向

步骤9：为最外层垂直方向的堆叠视图创建布局约束，设置其上下左右边缘与Safe Area边缘的间距都为0，并确保勾选Constraint to margins。在Attributes Inspector中确保Alignment设置为Fill，Distribution设置为Fill Equally，Spacing设置为10。在配置栏中修改屏幕的尺寸和方向，效果如图8-45所示。

图8-45　为最外层的堆叠视图创建约束

接下来，我们还需要在所有按键的上方为计算器添加1个标签控件，通过该控件来显示用户输入的数字以及计算的结果。要想实现这个布局，我们并不需要去破坏现有堆叠视图，直接将标签控件添加到相应的堆叠视图中即可。

实战：添加标签控件到堆叠视图中。

步骤1：从对象库中拖曳一个标签控件到垂直堆叠视图的顶部，使其成为最外层堆叠视图中最顶端的子视图。将内容修改为任意数字（比如34585.23），将Color设置为白色，将Background设置为Light Gray Color，将字号设置为50，对齐方式为右对齐。

注意　任何控件都可以被随意添加到堆叠视图之中，所以在放置控件到目标位置之前，一定要确认好位置，否则就会出现布局混乱的情况。

在Mac的计算器应用中，Label中的数字是不能紧贴屏幕右侧边缘的，所以我们需要利用一个容器，为标签设置一些约束，从而解决这个问题。

步骤2：从对象库中拖曳一个视图（UIView）到最外层的垂直堆叠视图之中，再将之前的标签控件拖曳到视图里面。因为视图本身在垂直堆叠视图的内部，所以不需要设置任何的约束。

步骤3：选中视图中的标签控件，为其上下左右与它的父视图边缘创建0、0、10、10间距的约束，如图8-46所示。

步骤4：将Label的Background修改为无填充色，将其父视图的Background设置为Light Gray Color。

接下来，我们需要仿照Mac版本的计算器修改所有的按键标题。但是这里出现一个问题：计算器中最下面的一行只有3个按键，而且最左侧的按键0占据了2个按键的宽度。如果此时你强行删除1个按钮的话，该水平堆叠视图中的三个按键会等宽平均布局，如图8-47所示，用户体验会不尽如人意。

图8-46　为Label的容器添加约束

图8-47　为Label的容器添加约束

实战：设置约束调整按键宽度。

步骤1：同时选中小数点（.）和等号（=）两个按键，在Add New Constraints面板中勾选Equal Widths。此时这两个按键的宽度被设置为相同。

步骤2：同时选中0和小数点（.）按键，在Add New Constraints面板中勾选Equal Widths，让这两个按键的宽度也相等。

实际上我们希望按键0的宽度是小数点按键宽度的2倍，但是在添加约束的时候我们并不能实现这样的操作，需要通过单独编辑约束的方式来解决它。

步骤3：选中包含按键0的水平堆叠视图，在Attributes Inspector中将Distribution设置为Fill Proportionally，让堆叠视图按照约束比例布

局。

步骤4：选中按键0，然后在Size Inspector中找到与它相关的约束，这里只有1条，单击其右侧的Edit，在弹出的面板中将Multiplier设置为1：2，代表根据比例按键0的宽度是2倍，如图8-48所示。

图8-48　将按键0的宽度设置为小数点按键的2倍

提示　如果你仔细看的话，会发现按键0的边缘与其他行的边缘并没有完全对齐，这是因为该行的间隔只有2个10点，而其他行都是3个10点。可以重新调整按键0的Multiplier属性值为1：2.1。

步骤5：最后将C、+/-、%三个键的背景色修改为Dark Gray Color，将÷、×、-、+四个键的背景颜色修改为橘红色，如图8-49所示。

虽然界面的整体感觉不错，但是最外层的堆叠视图最好还是与屏幕边缘有一定的间隔距离。选中最外层的堆叠视图，在Size Inspector中找到之前设置好的4个约束，依次单击这些约束的Edit按钮，在弹出的面板中将Constant属性值设置为20，如图8-50所示。

图8-49　将最右侧的按键设置为橘红色

图8-50　设置按键的边缘有20点的间隔

对于堆叠视图还有一点需要说明的是，在大多数情况下我们总是希望其内部的所有子视图都按照等宽（水平堆叠）或等高（垂直堆叠）排列。但在特殊的时候却希望按照内容的多少以比例的方式进行布局。

实战：在堆叠视图中按照比例布局。

将按键8的内容修改为3.14159，此时该按键的内容由于太长被自动截取为3...9。

选中包含该按键的水平堆叠视图，然后在Attributes Inspector中将Distribution属性修改为Fill Proportionally，效果如图8-51所示。

图8-51　设置按键的Distribution属性

第9章　Swift 4中阶知识

本章我们会更加深入地学习Swift的相关知识与概念，主要包括：类（class）、对象（object）、属性（property）和继承（inheritance），还有就是override关键字是用来做什么的，初始化和可选是什么。

9.1　类和对象

类就像是决定各种属性的蓝图，如图9-1所示。通过一张汽车设计蓝图，我们可以确定一辆汽车包含几个座位，发动机是多大的，有几个排气孔等。一旦你根据这张蓝图制造了一辆汽车，那么这辆真实存在的汽车便是对象。

图9-1　汽车蓝图

这辆汽车有很多属性，也就是对象中的变量与常量。例如汽车的颜色，使用代码可以表示为：let color=red。汽车的座位数，使用代码let numberOfSeats=5表示。在程序代码中，这些都被称为属性（property）。

对象也包含动作（action），也就是我们常说的方法（method）。比如我们想让汽车开动前行，则可以让该对象执行func drive（）方法。

最后，如果我们想要让某个事件发生或某个特殊时间点做某件事情的话，可以利用事件（event）。例如当视图控制器中的视图被载入内存以后会执行viewDidLoad（）方法。

9.2　创建全新的类

首先让我们创建一个全新的Xcode项目。

实战：使用新的模板创建项目。

步骤1：Xcode中创建一个新的项目，在模板选择面板中选择macOS/Application/Command Line Tool，如图9-2所示。

为什么我们要创建Command Line Tool项目而不是Single View App呢？主要是因为不想让大家被其他花里胡哨的东西所影响，而且本身这个项目也不需要Assets.xcassets等素材文件夹的支持。

图9-2　选择Command Line Tool模板

本章中创建的项目非常简单，其目的仅仅是帮助读者了解那些代码都是做什么用的，而不用关注UI或视图控制器方面的事情。

步骤2：在项目选项面板中，将Product Name设置为Classes and Objects，并确认Language为Swift，然后单击Next按钮。在选择好保存位置以后，单击Create按钮完成项目的创建。

在项目创建好以后，你会发现它的配置选项比之前iOS/Single View App要少很多。而且在项目导航栏中，只有一个swift文件——main.swift。这里就是我们编写并执行代码的地方，从头到尾，始终都在这里。目前该文件中的代码只有下面两行：

```
import Foundation

print("Hello, World!")
```

另外，我们将会在后面的学习中为项目添加一些附属文件，用于存储我们的数据模型等。

构建并运行项目，你可能已经预感到在Xcode底部的控制台中会打印出“Hello，World！”字符串，除此以外并无其他。与之前我们创建的项目有些不同的是，之前总是有一个可视化界面，总是有一个视图可能呈现相关的内容，而这里并没有这样的视图，我们只会看到main.swift文件中的类和对象。

接下来，我们需要在该项目中创建一个新类，继而在main.swift文件中调用该类。

步骤3：在项目导航中的Classes and Objects文件夹（黄色图标）上面单击鼠标右键，在弹出的快捷菜单中选择New File...，在文件模板面板中选择macOS/Swift File，单击Next按钮。

设置新类的文件名为Car，单击Create按钮。

目前的Car.swift文件中除了import Foundation一行代码以外并无其他，我们需要在这里手动创建Car类。

提示　在Swift文件中需要继续保留Foundation框架，因为它包含了Swift语言中很多基本的功能，例如基本的数据类型、集合和操作系统服务、数学函数、随机函数等。

在Car.swift文件中添加下面的代码：

```
class Car {

}
```

这里使用class关键字创建类，然后添加类的名称，在Swift中命名类的名称时需要其所有单词的首字母都为大写，这是与常量、变量和方法命名不一样的地方。在类名称的后面使用大括号完成类的创建。

目前Car类还是一个完全空白的类，因为在大括号中并没有任何的代码。它就像是一张空白的蓝图，为了可以让这张蓝图起到一定的作用，接下来我们需要在该类中添加一些代码。

在Car类中添加下面的属性：

```
class Car {

  var colour = "Black"
  var numberOfSeats: Int = 5

}
```

如果把属性放到类的外面，其实它就是变量和常量。在Car类中，我们设置了一个colour属性，并且让它的值为字符串类型的Black。也就意味着，现在所有从生产线上生产的汽车默认都是黑颜色的。第二个属性是汽车里面的座椅数量（numberOfSeats），因为代表的是数量，所以我们指明该变量的类型为整型。

目前，我们已经为汽车蓝图添加了两个属性，如果现在决定使用该蓝图创建一个汽车对象的话，那它就会是一辆黑色五座汽车。

9.3 创建枚举

在9.2节中，我们创建了全新的Car类，它包含两个属性——colour和numberOfSeats。

接下来，我们会继续添加一个新的属性——carType来标识汽车的类型，比如普通轿车（Sedan）、双门轿车（Coupe）、两厢轿车（Hatchback）等。但是对于carType属性，我们该如何定义它的类型呢？也许你会用整型值0代表普通轿车，用1代表迷你型轿车，用2代表两厢轿车。但是这样做的问题在于还需要一个文档或备注来说明每个数值代表什么类型的汽车。

又或者出于某些原因，其他人负责了你的项目代码，那当他们看到这些数值的时候会不清楚0、1、2代表的是什么。因此为了能够尽可能清楚地表达汽车的类型，我们需要使用枚举类型（enumeration，简称enum）。

实际上，创建一个枚举等同于创建一个新的数据类型。例如整型类型代表所有的整数值，字符串类型代表由单个字符组成的字符串。我们可以创建一个CarType类型，并且让它有几个不同的选项。

在Car类定义代码的上面，创建一个枚举类型：

```
import Foundation

enum CarType {
  case Sedan
  case Coupe
  case Hatchback
}

class Car {
……
```

这里使用关键字enum声明一个枚举类型，之后添加枚举的名字，与类的命名方式一样，单词的首字母要求大写。最后是一对大括号。在大括号之中，我们使用了另一个关键字——case，每个case代表一种汽车类型。

接下来，我们就可以在Car类中创建CarType类型的变量了：

```
class Car {

  var colour = "Black"
  var numberOfSeats: Int = 5
  var typeOfCar: CarType = .Coupe

}
```

这里我们创建了一个新的属性typeOfCar，它是我们自定义的数据类型CarType，这个属性的值是.Coupe。利用点（.）标记可以访问CarType枚举中的所有情况，其实如果你仅仅输入点（.），然后再耐心等待1秒钟左右，Xcode将会列出所有的case值，以供我们选择。

9.4　根据类创建一个对象

在本节中，我们将会向大家介绍如何使用自定义的枚举属性，这样我们就再也不需要使用0、1、2这些数值来代替所有的情况，大大减少了出现Bug的情况。

首先我们需要先将“汽车蓝图”（Car类）实例化为真正的汽车对象（Car对象）。可能你会发现，目前在项目导航中的Car.swift文件，其头部的雨燕图标都是灰色的，代表该文件有所修改但是还没有保存，这个时候请先使用Command+S快捷键将文件保存上，如图9-3所示。

图9-3　保存修改后的Car.swift文件

保存修改后的文件看似是一个非常简单的操作，但却非常重要。因为有些时候，当我们试图去引用一些还没有被保存的、在其他文件中的代码的时候，往往会出现很多问题。

接下来，我们需要在main.swift文件中实例化一个Car对象，因为在该项目中程序代码会从这个文件开始运行。

在main.swift文件中添加下面的代码：

```
import Foundation

let myCar = Car()
```

使用let关键字创建一个全新的对象——myCar，我们使用Car类作为它的“蓝图”。

现在我们已经成功创建了myCar对象，但是还没有看到任何的汽车出现在屏幕上。下面让我们通过myCar对象的属性来看看这辆车是什么

样子。

```
let myCar = Car()

print(myCar.colour)
print(myCar.numberOfSeats)
print(myCar.typeOfCar)
```

构建并运行项目，程序会先创建一个Car类型的对象——myCar，然后会在控制台中显示对象的各种属性。

```
Black
5
Coupe
Program ended with exit code: 0
```

9.5 类的初始化

在9.4节中我们创建的Car类型对象，在实例化的时候并没有携带任何参数，因此在控制台中打印出来的对象属性都是"蓝图"中的默认值。如果我们想要一辆个性化的汽车该怎么办呢？我们可以在类中创建类初始化方法。

我们希望汽车是可以自定义颜色的，虽然colour的默认颜色为黑色，但通过下面的代码我们可以将其修改为红色。

```
let myCar = Car()

print(myCar.colour)
print(myCar.numberOfSeats)
print(myCar.typeOfCar)

myCar.colour = "Red"

print(myCar.colour)
```

控制台打印内容为：

```
Black
5
Coupe
Red
Program ended with exit code: 0
```

如果将上面的代码转化为现实生活中的场景，就相当于我们在拿到新车以后，再返厂将车的颜色喷涂为红色，作为一般的客户来说，这种做法是不现实的。其实我们并不想重新为车喷涂颜色，而是汽车在生产线上装配的时候就是红色。

其实，在将类实例化的时候我们希望有一个自定义的设置将汽车的颜色喷涂为红色，让汽车生产厂商直接按照客户的需求来生产汽车。在Swift语言中，需要在类中创建一个自定义初始化方法。

在main.swift中删除之前修改汽车颜色的代码，再在Car类中添加一个初始化方法：

```
class Car {

  var colour = "Black"
  var numberOfSeats: Int = 5
  var typeOfCar: CarType = .Coupe

  init(customerChosenColour: String) {
    colour = customerChosenColour
  }
}
```

这里使用init关键字声明一个初始化方法，括号中的参数代表着客户想要的汽车颜色。在初始化方法中，将这个颜色赋值给Car类的colour属性。

回到main.swift文件，此时编译器会报错：初始化方法丢失参数customerChosen-Colour。如图9-4所示。这是因为在之前定义Car类的时候，我们并没有定义该类的初始化方法。而现在的init（）方法中有一个必须实现的参数。

图9-4　之前的实例化方法调用报错

修改代码如下：

```
let myCar = Car(customerChosenColour: "Red")

print(myCar.colour)
```

```
print(myCar.numberOfSeats)
print(myCar.typeOfCar)
```

尽管在Car类中colour属性的默认值为黑色，但是我们利用初始化方法，将对象的属性值设置为了自定义的红色。也就意味着在制造汽车的时候，汽车的颜色已经从默认的黑色修改为了红色。

构建并运行项目，在控制台显示的信息如下：

```
Red
5
Coupe
Program ended with exit code: 0
```

因此，通过类的初始化方法可以重写类中属性的默认值。如果你还记得之前介绍的有关类的事件（Event）的内容，那么初始化方法就属于这种情况。在实例化对象的时候，程序会自动调用初始化方法，只不过目前我们仅仅是在该方法中设置colour属性的值。

9.6 Designated和Convenience初始化方法

现在，创建的Car类还有一个问题，如果我们试图再实例化一个汽车对象，就必须在初始化的时候指定汽车的颜色，即使Car类中的colour属性默认是黑色也是如此，否则编译器就会报错。

其实我们更需要一个用于实例化标准属性的汽车对象的初始化方法。在Swift语言中，我们可以使用designated和convenience两种类型的初始化方法来解决这个问题。

回到之前的Car.swift文件，当我们使用init关键字创建初始化方法的时候，这个方法就叫作Designated初始化方法，你可以在这个方法中强制设置某些参数，让类在实例化的时候必须通过参数被赋值。例如可以将Designated初始化方法修改为下面这样：

```
init(customerChosenColour: String, numberOfSeats: Int,
typeOfCar: CarType) {
  colour = customerChosenColour
  // 因为参数名称与类中属性名称一样，所以这里使用self.属性名称来代表将参数的值赋给类中的属性。
  self.numberOfSeats = numberOfSeats
  self.typeOfCar = typeOfCar
}
```

但是在这样设置了Designated初始化方法以后，每次在实例化对象的时候就会非常麻烦，而且类中已经为每个属性设置了默认值，很多客户可能只是需要个性化其中的一种属性，并且随着类要实现的功能越来越多，有可能在Car类中需要定义十几甚至几十个属性，通过这样的方式肯定是不现实的。

要想解决这个问题，我们需要通过Convenience初始化方法。在Car类中添加下面的代码：

```
class Car {

  var colour = "Black"
  var numberOfSeats: Int = 5
  var typeOfCar: CarType = .Coupe

  init() {
  }

  convenience init(customerChosenColour: String) {
    self.init()
    colour = customerChosenColour
  }
}
```

这里使用convenience init关键字创建convenience初始化方法。当我们使用convenience初始化方法的时候，首先要调用Designated初始化方法来实例化一个对象，然后再去设置这个对象的colour属性。

回到main.swift文件，将之前的代码修改为let myCar=Car（），这意味着myCar对象使用的是Car类的Designated初始化方法实例化的对象。

在其下面添加新的代码：

```
let myCar = Car()

let myFriendCar = Car(customerChosenColour: "Gold")

print(myCar.colour)
print(myCar.numberOfSeats)
print(myCar.typeOfCar)

print(myFriendCar.colour)
print(myFriendCar.numberOfSeats)
print(myFriendCar.typeOfCar)
```

此时在自动完成列表中，我们既可以使用Car类的Designated方法实例化一个全部为默认值的对象，也可以使用Convenience方法实例化一

个个性化颜色的汽车对象，如图9-5所示。

图9-5　通过自动完成特性显示的初始化方法

控制台显示信息如下：

```
Black    // myCar属性
5
Coupe
Gold     // myFriendCar属性
5
Coupe
Program ended with exit code: 0
```

在本节的最后希望大家一定要记住的是：用Designated初始化方法必须要保证类中的所有属性都被赋值，而Convenience初始化方法实际上是将Designated方法实例化好的对象的个别属性值进行重写。有了这样的经验，你还可以再定义几个不同的Convenience方法来实例化不同需求的对象。

9.7　创建一个方法

目前，我们已经创建了一个Car类，并且通过该类实例化了2个对象。在类中除了定义3个属性和2个事件方法，本节我们要在类中添加动作（方法）让汽车可以开动起来。在Convenience方法的下面添加一个新的方法。

```
func drive() {
  print("汽车已经开动")
}
```

在该方法中使用func关键字定义一个方法，方法名称为drive，该方法没有任何参数。在方法中，我们使用print语句表示汽车已经开动。

在main.swift文件中删除之前所有的print代码，在最后添加一行新的代码：myCar.drive（）。我们可以通过点（.）操作符访问对象的属性或执行对象中的方法。

构建并运行项目，在控制台中可以看到打印的信息，代表drive（）方法被执行。

```
汽车已经开动
Program ended with exit code: 0
```

提示　在类外面定义的叫作函数，在类内部定义的函数则叫作方法。之前用于生成随机数的arc4random_uniform（）就是函数，因为它没有定义在任何的类中，因此不需要用点操作符来调用它。

9.8 类的继承

在本节中我们将会学习继承（Inheritance）的相关知识。现在所有的Car类型的对象都具备了开动（drive）的能力，然而随着时代的发展和社会的进步，特斯拉品牌的汽车已经具备了自动驾驶的能力，我们将会在类中添加可以自动驾驶的汽车。

但是自动驾驶汽车不完全等同于普通的汽车，为了创建它，我们需要一个新的"蓝图"，继续在项目中创建一个新的Swift文件——SelfDrivingCar。

在Classes and Objects文件夹（黄色图标）中添加一个新的Swift文件。文件名称设置为SelfDrivingCar，确认Group为Classes and Objects，并勾选Targets中的Classes and Objects选项。

在SelfDrivingCar.swift文件中的import Foundation下面添加类定义的代码：

```
class SelfDrivingCar {

}
```

在selfDrivingCar类中需要很多的属性，包括普通汽车的colour、numberOfSeats和typeOfCar。还记得之前我说过的优秀的程序员都是非常懒的，作为其中的一员，我们并不想手动建立这些属性，复制/粘贴的方式也不优雅，所以我们通过让SelfDrivingCar类继承Car类来处理这个问题。

修改SelfDrivingCar类的代码。

```
import Foundation
class SelfDrivingCar : Car {
}
```

在SelfDrivingCar名称的后面添加冒号（：），接着是Car，这样SelfDrivingCar类就继承了Car类的所有属性和方法。我们可以说，SelfDrivingCar现在是Car的子类，Car是SelfDrivingCar的父类。

在main.swift文件中实例化一个新的SelfDrivingCar对象。

```
let mySelfDrivingCar = SelfDrivingCar()
mySelfDrivingCar.drive()
print(mySelfDrivingCar.colour)
```

构建并运行项目，在控制台中显示如下信息：

```
汽车已经开动   // 调用myCar.drive()
汽车已经开动   // 调用mySelfDrivingCar.drive()
Black        // 调用mySelfDrivingCar.colour
```

Car中的任何事情在SelfDrivingCar类中都可以有。

在程序语言中，我们可以用动物族谱来更好地说明继承，如图9-6所示。因为所有的动物都需要呼吸，所以在Animals类中定义了breathe（）方法。而不管是鸟类（Birds）还是哺乳动物（Mammals）类都会从Animals继承breathe（）方法。另外Birds类还有属于自己的fly（）方法，Mammals类有属于自己的hasHair属性。

图9-6　有关动物的继承

在图谱中，人类（Humans）是Mammals的子类，因此它既拥有来自于Animals类的breathe（）方法，又拥有来自于Mammals类的hasHair属性。

9.9 重写一个继承的方法

在之前的例子中，如果我们再创建一个鱼（Fish）的类，它也是需要继承Animals类，但是Fish是不能够呼吸空气的，它有自己独特的呼吸方式——在水中通过腮呼吸。这就需要我们在Fish类中重写（override）breathe（）方法。

```
class Fish : Animals {
  override func breathe() {
    super.breathe()
    // 在水中呼吸的代码
  }
}
```

在Fish类中我们重新定义breathe（）方法，但要在func breathe（）的前面使用override关键字，这样当我们调用Fish类的breathe（）时，会先执行父类（Animals）的breathe（）方法，接着再执行自己在水中呼吸的代码。

如果你查看之前任何项目中的ViewController.swift文件的代码，会发现很多方法都被重写了。

```
class ViewController: UIViewController {

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

  updateUI()
}
```

实际上，ViewController类都是继承于UIViewController类，在UIViewController类中也有viewDidLoad（）方法，用于处理其他视图对象的显示情况。通过重写viewDidLoad（）方法，我们可以知道在

父类中定义了viewDidLoad（）方法，而且还重写了该方法，添加了属于自己的类的特定方法。

super关键字用于指定ViewController类的父类，也就是UIViewController类，并执行父类的viewDidLoad（）方法，我们并不关心它会去做什么，只要做好自身类的事情就好。

接下来，我们会利用所学的继承的相关知识，完善SelfDrivingCar类。

在SelfDrivingCar类中添加一个新的属性——Destination，因为作为普通汽车，目的地是由司机决定的，但是在没有司机的自动驾驶汽车上，我们会需要这个属性来确定汽车行驶的目的地。

```
class SelfDrivingCar : Car {
  var destination: String = "幸福巷4号"
}
```

destination是SelfDrivingCar类唯一的属性。

接下来我们需要重写drive（）方法。如果只是使用func关键字，编译器则会报错，如图9-7所示。这意味着在父类Car中已经定义了drive（）方法，子类中要重写drive（）方法则必须使用override关键字。另外，如果在编写代码的时候遇到同样的报错，就可以判断父类中定义了同样的方法。

图9-7　没有使用override关键字报错

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

  print("驾驶的目的地为：" + destination)
}
```

在该方法中，首先还是执行父类的drive（）方法，然后再执行自身的扩展功能代码，打印自动驾驶的目的地。

修改main.swift代码。

```
import Foundation

let mySelfDrivingCar = SelfDrivingCar()
mySelfDrivingCar.drive()
```

构建并运行项目，在控制台中会显示下面两条信息。

汽车已经开动　// 来自于Car类的drive()方法
驾驶的目的地为：幸福巷4号　　// 来自于SelfDrivingCar类的drive()方法

9.10　Swift语言中的可选

如果你之前仔细观察过项目代码的话，不难发现在swift语言中有很多地方都用到了问号（？）和感叹号（！），如图9-8所示。本节我们就向大家解读这两个符号的作用。

图9-8　项目代码中的？和！标识

让我们回到SelfDrivingCar类，在该类中声明了一个属性叫作destination，用来指引自动驾驶汽车驶向到哪里。但是，我们给了这个属性一个默认值——“幸福巷4号”，这样的代码似乎有点“不近人情”，为什么要让每个刚下线的新车的目的地都是幸福巷4号呢？

或者你想到可以不为destination赋值，也就是让destination为nil。在Swift语言中，nil代表没有值，它与纯粹的空字符串（""）完全不同。当前的destination的类型是String，所以我们只能将它的初始值设置为""，但这并不代表它没有被赋值，而是被赋值为空字符串。在当前情况下，我们想先让destination属性在用户赋值之前一直为nil。

如果在声明destination属性的时候将nil赋值给它，编译器将会报错，如图9-9所示。nil是一种非常特殊的数据类型，我们不能直接将nil赋值给String类型的变量。

图9-9　不能直接将nil赋值给String类型的变量

正确的做法是将属性的声明修改为var destination：String？，也就是在变量声明的最后添加一个问号，这样也就代表该变量包含了nil值。因此，在Swift编程里面，当我们想要创建一个属性时，建议在变量声明的最后添加问号。

在修改完成以后，print语句会报错：使用了一个“未拆包”（unwrapped）的可选字符串变量，需要在其结尾添加感叹号（！），如图9-10所示。

图9-10　使用了一个“未拆包”的可选字符串变量

让我们再重新梳理一下destination的问题，为了可以让它包含nil值，需要在声明变量的时候在其末尾添加问号，我们管这种形式的变量叫作可选（optional）变量。该类型的变量不仅可以包含字符串类型的值，还可以包含nil。

但是，为了在之后可以访问或修改该变量，就不得不将其做“拆包”处理。其中一种方式叫作强制拆包，当使用变量的时候，在其结尾处添加一个感叹号。这也就相当于告诉编译器：我确认这个变量包含有真正的类型值，而不是nil。

将代码修改为print（“驾驶的目的地为：”+destination！），构建并运行项目，编译器会报错——发生致命错误：在拆包一个可选值的时候，意外发现nil，如图9-11所示。

图9-11　在拆包一个可选值的时候，意外发现nil

使用感叹号代表你告诉编译器destination中已经包含了确切的值，并且绝对不是nil，直接使用它就好。但实际上，我们并没有在任何地方为destination赋予真正的字符串类型值。编译器并不会检测它是否为nil，所以在打印的时候会报致命错误，因为print语句是不能打印nil值的。

所以使用感叹号为可选变量进行强制拆包是一种非常危险的方式，这种方式完全依赖于程序员百分百确认该变量有真正的值存在，否则应用程序就会崩溃。

在main.swift文件中添加下面的代码：

```
import Foundation

let mySelfDrivingCar = SelfDrivingCar()
mySelfDrivingCar.destination = "幸福巷4号"
mySelfDrivingCar.drive()
```

构建并运行项目，可以在控制台看到如下的信息，程序运行状态良好。

```
汽车已经开动
驾驶的目的地为：幸福巷4号
Program ended with exit code: 0
```

虽然我们通过为destination赋值的方式解决了之前的问题，但是如果下次忘记了为destination赋值，应用程序依旧还会崩溃。作为用户，谁也不希望在App运行的时候发生崩溃，并跳回到Home主屏幕。这样不可避免地会得到用户1星的评价和评论。

为了避免出现崩溃的情况，我们需要一个更安全的机制，让这种情况永远不会发生。修改SelfDrivingCar类如下：

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

  if destination != nil {
    print("驾驶的目的地为：" + destination!)
  }
}
```

我们需要通过if语句来检查可选变量是否安全，因此使用if destination！=nil语句，这样如果destination有真正值会正常打印到控制台，否则就根本不会执行print语句。

在main.swift文件中删除对destination的赋值语句，构建并运行项目，应用程序便不会再发生崩溃的情况，这样项目的代码就非常安全了。

接下来，再说说“拆包”的第二个方式：可选绑定（optional binding）。使用可选绑定可以在你访问该变量之前，帮助你检测并确保可选变量有真正的值存在。将drive（）方法中的代码修改如下：

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

  if let userSetDestination = destination {
    print("驾驶的目的地为：" + userSetDestination)
  }
}
```

构建并运行项目，运行效果和之前一样，不会发生崩溃的情况。这就是可选绑定的语法，在if语句中，使用let创建一个常量，并将可选变量赋值给这个常量。当destination的值不为nil时，就会执行if语句中的代码。并且可选变量的值就存储在userSetDestination常量中，在if语句内部可以随时使用它。

使用可选绑定的好处在于，避免了由于强制拆包发生崩溃的情况，让代码更加安全。

可选是Swift语言中非常精彩的一个特性，让我们再用生活中一个具体的例子来描述一下它的概念。

我们把之前声明的变量想象为一个盒子，并且把一只叫作Tommy的猫放进这个盒子里面。另外，我们又在盒子中放了一瓶毒药（oh my god），而且这瓶毒药非常容易被弄开，并且随时都可以毒死Tommy猫。

此时，我们把盒子封上，并且去了其他的地方度过了一个不短的假期，因此我们听不到盒子的任何声音。问题就是，在我们回来以后，并不知道Tommy猫是死是活。

这里的盒子就好比是可选变量，它里面的Tommy猫可能活着，也可能因为喝了毒药死掉了。这就相当于可选变量可能包含真正的值或者是nil，但是只有在拆开盒子以后才能够得到确认。

在Swift语言中，如果创建了一个带有问号的变量就意味着我们创建了一个可选。它就像是一个密封好的盒子，可能有值也可能没有。当我们试图去使用变量时，需要打开盒子才能发现答案。如果盒子中是nil，则应用程序运行会崩溃。如果盒子中包含真正的值，则应用程序会继续平滑运行。

如果在使用变量的时候，在其后面添加一个感叹号，就叫作强制拆包。就好比不管盒子里面是什么情况，我们都会打开盒子，直接使用它。

另一种选择叫作可选绑定，例如在这种情况下，我们先摇一摇盒子，看里面是否有什么动静。如果盒子没有动静则代表里面已经没有活物了，直接忽略它以防止发生崩溃的情况。如果盒子里面有动静的话，则继续执行相应的代码。

声明可选变量的语法只是简单地在数据类型的后面加上问号，如下面代码行所声明的两个变量的类型，第一行变量的类型是字符串，第二行变量的类型则是可选字符串。

```
var destination1: String
var destination2: String?
```

为了使用该可选变量，我们可以直接在它的后面放一个感叹号。例如下面的代码：

```
print("驾驶的目的地为：" + destination!)
```

强制拆包的效果相当于将可选字符串类型强制转换为字符串类型，有经验的程序员很少会使用这种方式访问可选变量，因为如果变量的值为nil就会导致应用程序崩溃。

比较安全的方式是通过if语句判断可选变量的值是否为nil，如果不是则执行if语句的相关代码，防止应用程序崩溃。

```
if destination != nil {
  print("驾驶的目的地为: " + destination!)
}
```

Swift推荐的方式是在if语句中使用let关键字创建一个新的常量，并且让这个常量等于可选变量的值，如果可选变量有真正的值，则会执行if语句的相关代码，这就是可选绑定。

```
if let userSetDestination = destination {
  print("驾驶的目的地为: " + userSetDestination)
}
```

在可选绑定代码中，我们不会强制拆包任何变量，而是先检测destination是否有值，如果有则将其赋值给常量userSetDestination，并在括号中使用这个新常量。如果destination的值为nil，则直接跳过if语句中的代码。

第10章　利用Cocoapods、GPS、APIS、REST制作天气应用

在本章中，将会让大家的开发技能提高到一个新的层次。我们将会了解CocoaPods和开源代码库的相关知识。实际上，我们可以利用一些小巧而精致的第三方代码库，将其整合到我们自己的项目之中，这样可以节省大量的开发时间。另外，我们还会学习应用程序编程接口（Application Program Interface，API）的使用方法，并通过相关API抓取网站中的数据，再应用到自己的应用之中。

本章我们将会创建一个天气相关的应用，当它启动以后首先会显示基于本地的天气情况，这需要抓取当前手机的GPS数据，并将其经纬度值数据发送给远程的天气服务并得到当前位置的天气数据。

有两类信息需要显示在应用之中，当前的温度和当前的气象情况。比如单击切换按钮，可以进入另一个控制器视图，这也是本章我们将要学习的内容。在一个应用中包含多个控制器和视图。这里我们会实现在输入城市名称并单击获取气象信息按钮后使应用显示指定城市的气象信息。

运行效果如图10-1所示。

图10-1　Weather项目的运行效果

10.1　设置项目

首先，我们需要先在GitHub中下载Weather项目的初始代码。在初始项目中已经包含了设计好的用户界面并添加了相关约束，还有就是项目会用到的所有图片素材。

打开Weather项目，在项目导航中单击Main.storyboard文件。可以看到当前故事板中包含了多个控制器视图。在之前的所有项目中，都只有一个单独的视图控制器。这是我们第一次管理多个控制器视图，而且将来也会如此，在之后所创建的项目中还会添加更多的控制器。

目前所有的UI控件都已经设置好了约束，所以在不同屏幕尺寸的iPhone中，它们都会完美显示，如图10-2所示。

图10-2　在不同尺寸屏幕上的运行效果

在故事板中选择WeatherViewController视图控制器，然后将Xcode切换到助手编辑器模式，如图10-3所示。其中temperatureLabel关联的是故事板中显示温度的标签控件，weatherIcon关联的是显示天气情况的图像控件，cityLabel关联的是显示城市名称的标签控件。

图10-3　WeatherViewController中关联的IBOutlet和IBAction

在界面的右上角还有切换城市气象信息的按钮，如图10-4所示。但是它并没有关联任何的IBAction方法，这里使用的是Segue方式。当用户单击按钮以后就会触发标识为changeCityName的Segue，它会将屏幕切换到ChangeCityViewController控制器的视图。你可以在故事板中“点亮”Segue连线，然后在Attributes Inspector中看到有关标识（Identifier）的相关信息。

图10-4　2个控制器之间的Segue

在ChangeCityViewController控制器视图中有一个文本框控件，它与changeCityTextField属性建立了IBOutlet关联。当用户在文本框中输入了城市的名称，然后再单击其下方的Get Weather按钮就可以得到该城市的气象信息，GetWeather按钮关联了getWeatherPressed（_sender：AnyObject）这个IBAction方法。

这里有一个小小的设计缺陷：Get Weather并不像一个真正的按钮，有的用户可能并不认为这是一个可交互的控件。你也可以随意去修改这个按钮的风格，但是苹果的扁平化设计理念是让任何东西都去除冗余、厚重和繁杂的装饰效果，这样可以让"信息"本身重新作为核心被凸显出来。

最后，在视图的左上角还有一个返回按钮，它关联的IBAction方法为backButton Pressed（_sender：AnyObject）。在该方法中只有一条最基本的代码，用于销毁当前的视图控制器，并将用户带回到之前的WeatherViewController控制器。

```
@IBAction func backButtonPressed(_ sender: AnyObject) {
  self.dismiss(animated: true, completion: nil)
}
```

另外，在项目的Images.xcassets文件中，可以发现里面有很多预设好的气象图标，我们将会根据不同的气象条件让它们呈现到屏幕上。

10.2　注册免费的API Key

在该项目中，我们会在WeatherViewController类中看到一个叫作APP_ID的属性，它是字符串类型的常量，是在www.openweathermap.org网站注册后得到的API Key。只有通过这个API Key才能获取指定城市的气象信息。

```
class WeatherViewController: UIViewController {

  //Constants
  let WEATHER_URL =
"http://api.openweathermap.org/data/2.5/weather"
  let APP_ID = "1d505359c7db3fcf2502ffd40525ddc8"
  ……
}
```

提示　强烈建议读者使用自己注册后的API Key来构建当前的项目，因为当应用程序在每分钟有超过60次APP_ID访问的时候就会收取费用。

实战：在OpenWeatherMap上建立你自己的免费账号，并获得一个免费的API Key。

步骤1：在www.openweathermap.org网站中创建自己的账号。

步骤2：在用户控制面板中单击API Keys链接，复制Key中的字符串，如图10-5所示。

图10-5　在openweathermap上创建账号并获取API Keys

另外，你可以通过Name部分的按钮修改API服务的默认名称，方便我们将来区分多个API服务。但是，不管你创建了多少个API服务，作为免费用户来说，我们每分钟访问API的数量不能超过60次。

步骤3：将之前复制的Key字符串值替换到WeatherViewController类中声明APP_ID属性的代码部分。

10.3　为什么需要Cocoapods?

本节中，我们将要在你的Mac电脑里面安装一个CocoaPods。CocoaPods是什么呢?

当我们开发iOS应用程序的时候，经常会用到很多第三方开源类库，比如JSONKit、AFNetWorking等。可能某个类库又用到其他类库，所以为了使用它，必须还得额外下载其他类库，而其他类库又可能会用到其他类库——“子子孙孙无穷尽也”，这也许是一种比较特殊的情况。总而言之，一个个手动去下载所需的类库十分麻烦。

另外一种常见情况是，如果在项目中用到的类库有版本更新，我们就必须重新下载新版本的代码，然后再将其重新加入到项目之中，十分麻烦。如果能有什么工具可以解决这两个烦人的问题，那将“善莫大焉”。CocoaPods就是为此而存在的。

CocoaPods是iOS最常用、最著名的类库管理工具，上述两个烦人的问题，通过CocoaPods，只需要一行命令就可以完全解决，当然前提是你必须正确设置它。重要的是，绝大部分著名的开源类库都支持CocoaPods。所以，作为iOS程序员，掌握CocoaPods的使用方法是必不可少的基本技能。目前，CocoaPods拥有超过4.2万个库，用于超过300万个应用程序上面。

CocoaPods的官方网址是CocoaPods.org，在这里我们可以看到有关CocoaPods的相关介绍。另外，通过主页上的搜索栏可以直接找到相关的第三方开源类库。例如，我们想做一个进度条，你可以花费一周的时间自己去搞定它，也可以在搜索栏中输入“Progress Bar”找到很多关于进度条的项目代码，如图10-6所示。

图10-6　CocoaPods中关于进度条的代码

如果想要查看某个开源类库的详细信息，可以单击Expand按钮，这里我们展开YLProgressBar，如图10-7所示。

图10-7　YLProgressBar的相关介绍页面

在展开页面中，我们可以看到YLProgressBar开源库类的版本号、GitHub作者信息、效果预览、使用和安装方法、联系方式和许可证书。另外，在页面的右侧还可以查阅到代码的语言、发布时间、下载次数、安装次数、在GitHub中的状态，以及安装指南等。

接下来，在我们将开源类库整合到项目之前，先学习如何安装和设置CocoaPods。

10.3.1　在你的Mac上安装和设置Cocoapods

要想使用CocoaPods，我们需要在macOS上打开终端应用程序，通过命令行的方式先安装CocoaPods。

在终端中输入下面的命令：

```
liumingdeMacBook-Pro:~ liuming$ sudo gem install cocoapods
```

我们使用gem来安装CocoaPods，如果你使用过Ruby on Rails的话，对于gems就会比较熟悉。如果你没有使用过的话也没有关系，我们只是通过gem方式将CocoaPods安装到你的系统之中。其实，gems是RubyGems的简称，它是一个用于对Ruby组件进行打包的Ruby打包系统。

sudo命令代表我们以管理员权限执行后面的命令，在经过不长时间的等待以后就可以使用CocoaPods命令。如图10-8所示。

图10-8　通过终端安装CocoaPods

接下来，我们需要在macOS中设置CocoaPods。在终端中继续输入下面的命令：

```
pod setup --verbose
```

在命令行中有一个参数--verbose，它可以让我们看到CocoaPods每一步的设置进程和当前的状态。当设置完成以后，会在最后显示Setup completed信息。如图10-9所示。这意味着我们已经在macOS系统中安装并设置好CocoaPods，现在就可以使用它，而且以后再也不用执行这个操作了。

图10-9　设置CocoaPods

10.3.2　在你的Xcode项目中安装Pods

接下来，我们要将一些pods添加到Xcode项目中。首先，关闭之前的Xcode应用程序项目，因为CocoaPods会在后台对Xcode进行一些改变。

其次，就是要清楚项目的存储位置。因为项目的初始化代码是从GitHub下载的，在对其解压缩以后，需要进入到该文件夹中，如图10-10所示。当前的项目文件夹中包含一个Weather文件夹和一个Weather.xcodeproj文件，接下来执行pod init命令，当再次检查该文件夹时，会发现多了一个Podfile文件。

图10-10　通过pod init命令创建Podfile文件

此时你需要编辑Podfile文件，双击将其打开，文件的内容如下：

```
# Uncomment the next line to define a global platform for your
project
# platform :ios, '9.0'

target 'Weather' do
  # Comment the next line if you're not using Swift and don't
want to use dynamic frameworks
  use_frameworks!

  # Pods for Weather

end
```

一般的文本编辑器不如Xcode代码编辑器好用，所以在终端中输入open-a Xcode Podfile命令，打开当前文件夹中的Podfile文件。

将Podfile文件中的内容修改如下：

```
platform :ios, '9.0'

target 'Weather' do
  # Comment the next line if you're not using Swift and don't
want to use dynamic frameworks
  use_frameworks!

  # Pods for Weather
  pod 'SwiftyJSON'
  pod 'Alamofire'
  pod 'SVProgressHUD'

end
```

CocoaPods与Swift的语法虽然不尽相同，但是语义还是可以理解的。在Swift中我们使用//进行注释，在CocoaPods中则使用#号。

删除Podfile中的#号注释，激活platform：ios，'9.0'代码，表示我们的项目是基于iOS 9以上的SDK版本。

在Swift中，我们使用{}界定一段完整的代码块。在CocoaPods中则使用do...end作为标识，do就相当于“{”，而end相当于“}”。

在代码块中，使用pod命令添加开源库类，在Weather项目中我们需要：SwiftyJSON、Alamofire和SVProgressHUD三个开源库类。

如果想查看这些开源库的用途，可以利用cocoapods.org主页的搜索栏找到相关的内容，如图10-11所示。

图10-11　在CocoaPods.org中查看SVProgressHUD的用途

接下来回到macOS的终端应用程序，将当前目录切换到Weather文件夹。首先通过pod--version命令检查CocoaPods的版本，当前最新的版本应该是1.4.0。只要版本号大于1.1.0就没有问题。如果CocoaPods低于1.1.0版本的话，通过百度搜索一下如何升级CocoaPods即可。

确认终端应用程序中的当前目录为Weather，然后输入pod install命令。这样就会安装前面所提到的三个开源库类，如图10-12所示。

图10-12　使用pod install命令在当前项目中安装第三方库

如果将来某个开源库类发布了新版本，就可以直接使用pod update进行升级，非常方便。

10.4　设置Location Manager并从iPhone获取GPS数据

通过CocoaPods安装好开源库类以后，必须通过双击Weather.xcworkspace（白色图标的）文件才能正常打开项目，因为该文件中包含了CocoaPods的相关信息。

在项目导航中你可以发现Weather和Pods两个蓝色项目图标，展开Weather项目及其内部的Weather文件夹，你可以看到所有的文件被组织成Model、View和Controller形式，每个文件夹中都包含着相关的文件，如图10-13所示。

图10-13　通过CocoaPods将第三方库整合到Weather项目

首先，在项目导航中打开Controller文件夹中的Weather-ViewController.swift文件，该类中有2个预设好的常量属性，它是我们从openweathermap.org网站获取气象数据的“开门砖”。WEATHER_URL是从网站获取数据的url链接，APP_ID是我们自己在网站注册后得到的App Key。设置App Key的原因是网站需要跟踪哪种类型的用户需要访问什么内容的数据。

在openweathermap.org网站的API链接中，你可以看到各种类型的Web Service服务，如图10-14所示。比如当前天气数据、五天每3小时一次的预报、气象站数据管理、空气污染数据等。

图10-14　openweathermap.org提供的各种类型的Web Service服务

在WeatherViewController类中，会发现有很多预置好的Mark注释语句，每个Mark都代表需要通过方法实现的一部分功能。比如MARK：-

Networking代表需要实现网络连接功能，MARK：-JSON Parsing代表需要实现JSON格式数据的解析功能，MARK：-Location Manager Delegate Methods代表需要实现定位功能等。我们需要在这些标注的位置下面添加相关的方法。另外，我们还可以利用代码编辑窗口顶部的分隔条快速定位到标记的位置。如图10-15所示。

图10-15　快速定位到标记的位置

第一个标记是Networking，我们会利用HTTP从openweathermap网站请求数据。第二个标记部分是JSON Parsing，通过编写代码将获取到的数据解析为项目可以使用的格式。第三部分是UI Updates，通过代码更新气象条件的图标和相关的标签。Location Manager Delegate Methods部分是获取当前的位置，即iPhone手机当前的经纬度坐标。最后一个部分Change City Delegate methods是通过代码将一个控制器的数据传递到另一个控制器。

实战：获取GPS数据。

步骤1：在WeatherViewController.swift文件中导入GPS相关类库import CoreLocation。

通过该类库，苹果允许我们使用手机的GPS定位功能。按住option键并单击CoreLocation类库可以查看CoreLocation的相关描述。

步骤2：在WeatherViewController类声明的最后，修改代码如下。

```
class WeatherViewController: UIViewController,
CLLocationManagerDelegate {
```

CLLocationManagerDelegate是Swift语言中处理位置数据的协议（Protocol），也就是说，WeatherViewController类既是

UIViewController的子类，又符合CLLocationManagerDelegate的规则。

步骤3：在TODO：Declare instance variables here的下面声明实例变量。

```
//TODO: Declare instance variables here
let locationManager = CLLocationManager()
```

这里声明了一个CLLocationManager类型的常量，在其初始化方法中没有任何参数。

步骤4：修改viewDidLoad（）方法如下。

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

//TODO:Set up the location manager here.
locationManager.delegate = self

}
```

这里设置locationManager的delegate属性等于self，这个self相当于当前类——WeatherViewController。这里引入了一个全新的概念——委托（delegate）。

至于如何确定位置，这涉及了信号塔的GPS三角定位。另外，如果蜂窝信号很弱则可以通过Wi-Fi尝试去查找位置。

在CoreLocation类库中有很多的类和方法来帮助我们进行定位，locationManager对象的用处就是帮助我们找出iPhone当前的位置，而WeatherViewController类就充当了locationManager的委托（delegate）对象。这也就意味着每当locationManager找到具体位置的时候，该控制器就自愿充当处理位置数据的角色。

为了让WeatherViewController类能够成为locationManager委托对象的角色，首先要让其符合CLLocationManagerDelegate协议，然后在viewDidLoad（）方法中将自己（self）赋值给locationManager的delegate属性。协议和委托是Swift语言中较高层次的概念，我们会在后面的章节中进行详细介绍。

目前，我们只要清楚，为了通过locationManager获取GPS数据，需要让Weather-ViewController类作为locationManager的delegate。让locationManager处理所有的定位功能，只要有位置数据更新，locationManager知道会向谁去报告。

步骤5：在之前代码的下面添加下面的代码。

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

  //TODO:Set up the location manager here.
  locationManager.delegate = self
  locationManager.desiredAccuracy =
kCLLocationAccuracyHundredMeters
}
```

这里设置了locationManager的另一个属性来确定定位的精准度，在输入kCLLocationAccuracy的时候，自动完成列表中会列出多达6种不同的精确度设置。其中Best是最高级别的精确度，除此以外还有接近10米（NearestTenMeters）、100米（HundredMeters）、1千米（Kilometer）、3千米（ThreeKilometers）和适合导航（BestForNavigation），如图10-16所示。如果在项目中使用到定位功能的话，请先想好它需要什么样的精准度，才能满足应用程序的需求。

图10-16　GPS定位的6种不同精度

例如我们的应用程序是一个导航类应用，则需要适合导航的精准度。如果是天气类应用程序，则可以设置为100米精准度。

步骤6：继续添加下面的代码。

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

  //TODO:Set up the location manager here.
  locationManager.delegate = self
  locationManager.desiredAccuracy =
kCLLocationAccuracyHundredMeters
  locationManager.requestWhenInUseAuthorization()
}
```

最后，我们需要让应用程序询问用户是否打开获取位置数据的权限。如果你安装了一个需要定位的应用程序（例如美团），在第一次运行的时候都会询问是否打开定位的权限，上面的代码就是实现这个功能。

这里一共有2个可选方法，requestAlwaysAuthorization（）方法是在应用程序进入后台以后都可以进行定位，就像是一位永远不知疲倦的间谍。而requestWhenInUseAuthorization（）方法则只会在当前使用应用程序的时候才打开定位功能。

10.5　定位权限

如果你现在构建并运行应用程序的话，并不会在屏幕上看到权限提示框，原因是还需要添加一个描述信息。为了做到这一点我们需要编辑property list文件。

当我们创建一个全新的Xcode项目的时候，默认会在Supporting Files文件夹中创建一个info.plist文件。打开它以后会看到有很多不同的配置属性以及与其相对应的值，如图10-17所示。

图10-17　项目中的Info.plist文件

接下来，我们需要在info.plist中创建2个新的属性。单击Information Property List一行右侧的⊕按钮，此时其下方会添加一行新的属性值，如图10-18所示。

图10-18　在Info.plist中添加一行配置

此时新添加的默认属性名称为Application Category，将其修改为Privacy-Location When In Use Usage Description，再添加一个属性，名称为Privacy-Location Usage Description。

提示　在添加属性名称的时候，我们可以借助Xcode的自动完成列表，可以有效防止输入错误而产生的Bug。

在我们添加这两个属性以后，当程序执行到viewDidLoad（）方法中的locationManager.requestWhenInUseAuthorization（）一行，会去查找Info.plist文件中的两个属性值，并将它们显示到用户的屏幕上。

现在，我们需要给这两个属性赋值，这样在弹出权限提示框的时候会显示相应的信息。将这两个属性值都设置为：为了获取气象信息需要定位你当前的位置。

另一种修改属性值的方式是通过代码，在项目导航里面右击Info.plist文件，在弹出的快捷菜单中选择Open As/Source Code，你会发现Info.plist文件内容实际上是标准的XML格式。

注意　在Source Code状态下编辑项目属性会大大增加Bug出现的几率，所以强烈建议还是使用Property List方式进行属性的设置。

在GitHub网站提供的该项目的描述部分，你会发现修复应用程序数据传输安全（Fix for App Transport Security Override）的方法，将这部分XML格式的代码复制到Info.plist文件中新添加的两个属性的下方代码如下所示，其效果图如图10-19所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>NSLocationUsageDescription</key>
    <string>为了获取气象信息需要定位你当前的位置。</string>
    <key>NSLocationWhenInUseUsageDescription</key>
    <string>为了获取气象信息需要定位你当前的位置。</string>
    <key>NSAppTransportSecurity</key>
    <dict>
        <key>NSExceptionDomains</key>
        <dict>
            <key>openweathermap.org</key>
            <dict>
                <key>NSIncludesSubdomains</key>
                <true/>

<key>NSTemporaryExceptionAllowsInsecureHTTPLoads</key>
                <true/>
            </dict>
        </dict>
    </dict>
    ……
```

图10-19　GitHub中的应用程序数据传输安全配置

添加这些属性的原因是苹果只允许iOS设备通过HTTPS方式与远程API进行沟通，这是一种安全的数据传输协议，例如GitHub网站使用的就是HTTPS传输协议。

现在的问题是：只有付费用户才能使用openweathermap网站提供的HTTPS数据传输协议，而免费用户只能使用HTTP这种不安全的数据传输协议。因此，需要将之前的XML代码复制到Info.plist文件中，才可以调用非HTTPS的URL链接。

构建并运行应用程序，在程序启动以后你就会看到一个消息框，如图10-20所示。单击允许按钮，这意味着应用程序可以使用GPS传感器。如果你不小心单击了不允许按钮，也不用担心，可以在模拟器上通过主屏幕中的设置/隐私/定位服务/Weather，重新开启Weather应用程序的定位服务。

图10-20　项目运行后弹出的定位请求

10.6 在WeatherViewController中获取GPS数据

让我们回到WeatherViewController.swift文件，在viewDidLoad（）方法的最后再添加一行代码。

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

  //TODO:Set up the location manager here.
  locationManager.delegate = self
  locationManager.desiredAccuracy =
kCLLocationAccuracyHundredMeters
  locationManager.requestWhenInUseAuthorization()
  locationManager.startUpdatingLocation()
}
```

startUpdatingLocation（）方法会启动iPhone的定位服务，需要记住的是，它是一个异步方法（Asynchronous Method），也就意味着该方法会在后台抓取GPS位置坐标。

假如它运行在前台的话，也就代表它是在系统的主线程中运行，这样会导致整个应用程序发生“冻结”（假死）的情况。因为在iPhone查找到GPS位置的时候，根据之前所设置的定位精确度，会需要大概几秒钟甚至是十几秒的时间，此时应用程序根本无法响应用户的交互操作。这是非常糟糕的用户体验，因为用户不知道应用程序到底发生了什么。

startUpdatingLocation（）方法是在应用程序后台进行GPS定位，那它到底做了什么呢？如果查找到足够精确的位置坐标，它就会发送信息给WeatherViewController类。因为我们设置了locationManager的delegate属性，它就相当于locationManager对象的“老板”。为了让WeatherViewController类可以获取到位置信息，需要包含一个名叫

didUpdateLocations（）方法。在WeatherViewController.swift文件的下半部分可以找到它的注释。

在"//Write the didUpdateLocations method here："注释的下方，添加一个方法：

```
//Write the didUpdateLocations method here:
func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation]) {
}
```

在代码编辑面板中可以直接输入didUpdateLocations，在自动完成面板中可以看到3个相关的方法，如图10-21所示。我们要用的是第一个locationManager（_manager：CLLocationManager，didUpdateLocations locations：[CLLocation]）方法，并且在下方的方法描述中说明了在iPhone获取到有效位置数据后会告诉委托对象——WeatherViewController。

图10-21　在控制器中添加locationManager（）方法

注意　在输入方法的时候由于方法名称太长，可能会导致Bug的出现，强烈建议大家通过自动完成特性输入方法名称。

一旦locationManager发现新的位置，就会激活didUpdateLocations方法，我们会在后面添加代码来完善它的功能。现在，我们需要添加来自于locationManager的另一个委托方法locationManager（_manager：CLLocationManager，didFailWithError error：Error），在locationManager获取位置数据失败时告诉委托对象——WeatherViewController。失败的原因有多种的情况，可能因为用户将iPhone设置为飞行模式或因为用户是在电梯、隧道或地下停车场里面导致没有蜂窝信号等，这些情况都会导致该方法被激活。在该方法中

我们需要将错误信息打印到控制台，并且要在故事板的Label中显示定位失败的信息。

将didFailWithError方法修改为下面这样：

```
//Write the didFailWithError method here:
func locationManager(_ manager: CLLocationManager,
didFailWithError error: Error) {
  print(error)
  cityLabel.text = "定位失败"
}
```

在didUpdateLocations方法中有一个参数是locations，它是CLLocation数组类型。按住Option键并单击CLLocation可以看到相关的描述。

CLLocation对象是由CLLocationManager对象生成的位置数据，在该对象中包含了iPhone设备根据之前所设置的测量精准度、生成的地理位置坐标（包括经度值和纬度值）和高度值。

在调用startUpdatingLocation（）方法以后，locationManager开始获取当前设备的位置信息，每次位置的更新都会将这个新位置（CLLocation类型的对象）添加到didUpdateLocations方法的参数——location的数组之中。所以，在location数组中会包含一大堆的位置数据对象，不过我们最关心的是数组中最新的数据。因为被添加到数组中的首个位置数据的精度是比较粗略的，之后会越来越精准，所以最新的一个才是我们最想要的。

在didUpdateLocations方法中添加下面的代码：

```
func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation]) {
  let location = locations[locations.count - 1]

  if location.horizontalAccuracy > 0 {
    locationManager.stopUpdatingLocation()
    print("经度 = \(location.coordinate.longitude), 纬度 = \
```

```
(location.coordinate.latitude)")
  }
}
```

在didUpdateLocations方法中创建一个常量location，为了获取locations数组中最新的值，通过locations.count-1得到数组中最后一个元素的索引值，其中count属性值代表数组中元素的个数，因为数组索引是从0开始的，所以再减1就得到最后一个元素的索引值，然后利用该索引值得到最新的元素对象。

接下来我们需要判断位置信息的有效性，使用if语句判断location的horizontalAccuracy属性值是否大于零。location的经纬度坐标代表世界地图中的一个具体位置，如图10-22所示。horizontalAccuracy就相当于以该位置为圆心的半径，也就是用户可能的位置。

从图10-22中可以发现，horizontalAccuracy的值越高代表用户可能在的位置范围就越大。但是当这个值为负数（小于零）的时候就代表获取到了一个无效的数据。获取到一个有效数据时，就通过stopUpdatingLocation（）方法让locationManager停止定位，因为locationManager在通过GPS定位的时候是非常耗电的，除非你想故意为电池放电，否则在获取到有效数据之后，就要马上停止iPhone定位功能。

图10-22　经纬度坐标及精确度

之后，通过print语句将有效的经纬度值打印到控制台。CLLocation类中包含一个coordinate属性，该属性中包含longitude（经度）和latitude（纬度）属性。

构建并在模拟器中运行项目，如果在控制台中看到如图10-23所示的信息，这是因为Mac硬件中并没有GPS模块，所以在模拟器中运行就会定位失败。如果你是在iPhone真机上测试就不会出现这样的问题。

图10-23　模拟器没有GPS模块导致定位失败

解决这个问题我们需要在模拟器菜单中选择Debug/Location/Apple，这会让模拟器模拟一个GPS模块，而且设定当前iPhone的位置为苹果总部。控制台会显示如下的信息。

```
经度 = -122.03031802, 纬度 = 37.33259552
```

此时证明locationManager已经起作用了。为了验证经纬度坐标的真实性，可以通过浏览器访问www.latlong.net，在主页顶部单击Lat Long to Address链接，然后输入经度值和纬度值，就可以查找详细信息，如图10-24所示。

图10-24　latlong.net提供的定位服务

另外，你也可以通过地址查找经纬度坐标，单击Address to Lat Long链接，然后输入地址信息（例如北京市天坛）就可以，如图10-25所示。

图10-25　通过城市名称确定经纬度坐标

接下来，我们需要将经纬度坐标值作为参数发送回openweathermap的API。在didUpdateLocations（）方法中继续添加代码：

```
func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation]) {
  let location = locations[locations.count - 1]

  if location.horizontalAccuracy > 0 {
```

```
    locationManager.stopUpdatingLocation()

    print("经度 = \(location.coordinate.longitude), 纬度 = \
(location.coordinate.latitude)")

    let latitude = location.coordinate.latitude
    let longitude = location.coordinate.longitude

    let params: [String: String] = ["lat": latitude, "lon":
longitude, "appid": APP_ID]
  }
}
```

这里创建了latitude和longitude两个常量，分别来自location对象的coordinate属性。params的类型是[String：String]，它是字典（Dictionary）类型，也就相当于具有键/值元素的数组，对于字典的使用会在之后章节进行具体的介绍。其中字典的键的类型为String，值的类型也为String。

其中，字典params中第一个值的键（Key）为"lat"，它是字符串类型，它的值是常量latitude。另一个参数的键为"lon"，也是字符串类型，它的值是常量longitude。最后一个参数appid是我们在openweathermap注册后得到的APP Key。有了这三个参数我们便可以知道所在的城市名称。这三个参数的名称是特定的，不可以随便修改，在http://www.openweathermap.org/current页面中的By geographic coordinates部分可以看到代码样例，如图10-26所示。

图10-26　OpenWeatherMap提供的API格式

其中，API的调用方式为api.openweathermap.org/data/2.5/weather？lat={lat}&lon={lon}，{lat}和{lon}就是在代码中定义的参数。我们可以将样例API调用复制到浏览器中，此时会返回401错误，提示为：无效的API Key，可以访问http://openweathermap.org/faq#error401查看相关信息，如图10-27所示。

图10-27　通过浏览器调用openweathermap的API

通过该页面可知，从2015年10月9日开始，API请求就需要一个有效的APPID了。这也是为什么在params字典中还要定义appid键。

此时编译器会报错，如图10-28所示。意思是latitude和longitude当前是Double类型，而字典中的值类型为String类型。

图10-28　latitude和longitude的类型为Double引发的错误

此时可以如下修改之前的代码：

```
let latitude = String(location.coordinate.latitude)
let longitude = String(location.coordinate.longitude)
```

虽然现在会出现1个警告信息，但是不用担心，这只是目前还没有使用过params而已。

10.7　委托、字典和API

10.7.1　委托

在本节，我们会了解什么是委托（delegate），为什么要为locationManager设置delegate属性。

假如我们要在两个类（Class A和Class B）之间传递数据，最简单的方式是在Class A中实例化一个Class B对象，取名叫B1。假设在Class B中有一个属性叫作data，那么在Class A中我们可以很方便地设置实例化好的B1对象中data的属性值，例如将它设置为123，如图10-29所示。这是最简单最基本的在类之间传递数据的方法。

图10-29　Class A中实例化一个Class B对象

我们可以把Class A和Class B想象为两个视图控制器，它们之间需要传递数据。在大部分情况下，两个视图控制器都是我们自己定义的，所以会知道它们内部所有的属性和方法名称，数据之间的传递没有什么问题。

在本项目中，当locationManager找到iPhone当前位置以后，它会发送这个城市的名称，例如beijing。但是问题在于locationManager对象并不确定将这个值发给WeatherViewController的哪个属性，因为locationManager类是苹果工程师编写的，CoreLocation根本不可能知道我们编写的WeatherViewController类中会包含什么属性和方法。

要想解决这个问题，就需要用到委托。当locationManager定位成功以后，会发送位置信息给delegate，如果此时的delegate值为nil则不会发生任何事情。但是如果设置了delegate，比如当前的项目中就是WeatherViewController，那它就会接收来自locationManager对象发送

来的数据。一旦控制器接收到数据，就会进一步使用该地点去获取气象数据。

概括来说，哪个控制器类用到CoreLocation的定位功能，就需要将该控制器作为CLLocationManager类的delegate属性值，这样才能接收到相关数据。

10.7.2 字典

在之前的代码中我们使用到了字典（Dictionary），本节我们就要弄清楚什么是字典。

在现实生活中的字典包含一个词条和其相关的解释。

SWIFT：Swift，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macOS和iOS平台，用于搭建基于苹果平台的应用程序。

在Swift语言中，我们使用字典将键（Key）/值（Value）组成一对，每个值都对应唯一的键。当创建一个字典的时候，我们往往会创建多个键/值配对，但是它们必须都是相同的数据结构。比如键都是字符串类型，值也都是字符串类型；或者键是字符串类型，值都是整型类型。

```
var dict1 = ["中国": "北京", "英国": "伦敦"]
var dict2 = ["刘铭": 39, "李钢": 34]
```

在上面的代码中，Swift通过类型断言明确了dict1的数据类型为[String：String]，dict2的数据类型为[String：Int]。在定义字典的时候也可以写成下面这样：

```
var dict1: [String: String] = ["lat": "1233432", "lon": "4325",
"appid": "984732"]
var dict2: [String: Int] = ["刘铭": 39, "李钢": 34, "陈雪峰": 35]
```

这样的语法声明形式似乎和之前的数组有相似之处，没错，数组和字典都是Swift语言中的集合类型，通过不同的方式来组织和管理数据。数组管理的是有序数据，通过索引来访问数组中的指定元素。字典管理的是键/值配对数据，通过键来访问字典中的指定元素的值。

在Playground环境中，我们可以尝试获取字典中值。

```
var dict1: [String: String] = ["lat": "1233432", "lon": "4325",
"appid": "984732"]

let latitude = params["lat"]
```

10.7.3 API

之前我们一直在说使用API，但是API到底是什么呢？它是应用编程接口（Application Program Interface）的缩写，实际上它只是一个约定。为了能够从Web服务器获取到所需的信息，这个约定是你的应用程序必须遵守的规则，并且这个规则是预先设定好并写在了API文档中。

对于openweathermap的数据API，我们需要提供给它经度值、纬度值和App ID，那么它会给我们该位置的相关气息数据。如图10-30所示。我们会根据API文档提供的规则，让app生成一个请求（request）并发送给Web服务器，然后Web服务器会返回一个包含数据的响应。

图10-30　Class A中实例化一个Class B对象

10.8 使用Alamofire

在本节中，我们会通过相关参数生成HTTP请求，进而获取气息信息。如何生成HTTP请求以及如何发送这个请求，需要借助第三方类库Alamofire。利用该库可以节省很多的时间，例如在处理HTTP请求、接收相关信息和处理服务器异常等方面上。

首先在didUpdateLocations方法的最后添加如下代码：

```
func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation]) {
  let location = locations[locations.count - 1]

  if location.horizontalAccuracy > 0 {
    ……
    let params: [String: String] = ["lat": latitude, "lon":
longitude, "appid": APP_ID]
    getWeatherData(url: WEATHER_URL, parameters: params)
  }
}
```

我们通过getWeatherData（url：String，parameters：[String：String]）方法从openweathermap获取气象信息，当前Xcode编译器会报错：Use of unresolved identifier'getWeatherData'，这是因为我们还没有实现该方法。

接下来在Write the getWeatherData method here：注释行的下面创建getWeatherData（url：String，parameters：[String：String]）方法。

```
//Write the getWeatherData method here:
func getWeatherData(url: String, parameters: [String: String])
{
}
```

为了可以在getWeatherData（url：String，parameters：[String：String]）方法中使用Alamo-fire，需要先导入它。在WeatherViewController.swift文件中添加如下代码：

```
import UIKit
import CoreLocation
import Alamofire
import SwiftyJSON
```

提示　此时Xcode编译器可能会报错——No such module‘Alamofire’，尽管我们已经通过CocoaPods方式在项目安装了Alamofire，但是Xcode还是没有发现它。不用担心，这只不过是Xcode的Bug，可以先使用快捷键Shift+Command+K清理（Clean）一下项目，再使用Command+B重新构建（Build）一下项目，报错就会消失。

要想使用好Alamofire，最好先阅读一下它的说明文档。在浏览器中访问https://github.com/Alamofire/Alamofire，里面包含了Alamofire的特性说明、安装方法、使用样例等。

在getWeatherData（url：String，parameters：[String：String]）方法中，添加下面的代码：

```
func getWeatherData(url: String, parameters: [String: String])
{
  Alamofire.request(url, method: .get, parameters:
parameters).responseJSON {
    response in
    if response.result.isSuccess {
      print("成功获取气象数据")
    }else {
      print("错误 \(String(describing: response.result.error))")
      self.cityLabel.text = "连接问题"
    }
  }
}
```

新输入的这段代码是Alamofire的标准格式代码，request（）方法用于生产一个HTTP请求，它带有3个参数，url参数来自于项目之前定义好的WEATHER_URL链接，method参数是HTTP请求的方式，这里是get方式，除了get方式以外还有head、delete、post等其他方式。第3个参数就是API请求所需要的参数，也就是我们之前定义好的字典常量。

当我们将request发送到openweathermap以后就会收到回应的一些数据，这时需要检测response中result的isSuccess是否为真。如果isSuccess的值为假，则需要告诉用户网络连接发生了问题，这里需要将错误信息打印到控制台，并在cityLabel上显示相关信息。如果为真则需要去处理返回的数据。

这部分的代码与我们之前见过的有点不同，原因是该方法采用了异步方式。这也就意味着Alamofire是在后台与openweathermap服务器进行数据沟通的，这样就不会让用户在交互的时候有“假死”的感觉。

当我们从服务器得到响应信息以后就会运行response in里面的代码，也就意味着后台进程处理完成，数据已经从Web服务器获取到。

10.9 JSON以及如何解析JSON

本节我们需要处理在发送请求以后，成功响应的情况。在之前的代码中，我们只是简单地在控制台中打印"成功获取气象数据"的信息。

我们知道在成功响应以后，下一步需要格式化从openweathermap传回的数据，然后再将其显示到屏幕上。

```
if response.result.isSuccess {
  print("成功获取气象数据")
  let weatherJSON: JSON = response.result.value
}else {
```

首先，我们创建了常量weatherJSON，它的数据类型是JSON。JSON是JavaScript Object Notation的缩写，在很多程序语言和应用程序中都会用到JSON格式的数据。因为这是一种非常简单的大批量数据整理格式，并且易于在互联网上进行数据传输。

从openweathermap服务器获取的JSON格式数据就存储在response.result.value中，我们直接将它赋值给weatherJSON。

此时编译器会报错，因为response中result的value是可选的，所以需要在value后面添加一个感叹号将其强制拆包。因为在if语句中已经确认result.isSuccess的值为真，所以value一定是有真值存在的，这里才会使用强制拆包的方式。

编译器报的另一个错误是因为value的类型为Any？（Any的可选类型），虽然对其进行了强制拆包，但是我们不能将其赋值给JSON类型的常量。我们需要现将其转换为JSON类型。将之前的代码修改为：let weatherJSON：JSON=JSON（response.result.value！）就可以了。

实际上这里的JSON（）来自于第三方类库SwiftyJSON，JSON则是该类库中的一个结构体。它可以帮助我们更加方便地使用JSON格式的数

据。

在我们得到JSON格式的数据以后，可以将其打印到控制台，添加如下代码：

```
if response.result.isSuccess {
  print("成功获取气象数据")
  let weatherJSON: JSON = JSON(response.result.value!)
  print(weatherJSON)
}else {
```

构建并运行项目，虽然在模拟器中没有发现任何变化，但是在控制台中可以看到一大堆JSON格式的数据。如图10-31所示。

图10-31　程序运行后在控制台打印的位置信息

在控制台中，从最外面的一对大括号开始就是JSON格式的数据。另外，由于之前的代码设置，一旦在我们看到数据以后，就会让locationManager停止更新位置的操作。除此以外，我们还可以将locationManager的delegate属性设置为nil，这样做会更加彻底一些，因为目前我们不再需要locationManager了。

在didUpdateLocations方法中添加如下代码：

```
func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation]) {
  let location = locations[locations.count - 1]

  if location.horizontalAccuracy > 0 {
    locationManager.stopUpdatingLocation()
    locationManager.delegate = nil  //将delegate设置为nil
    ……
```

为了可以更加清楚地查看JSON数据的内容，我们可以借助jsoneditoronline.org网站，该网站可以根据提供的JSON数据生成便于查看的结构列表，增加了数据的可读性，如图10-32所示。

图10-32　jsoneditoronline.org所生成的JSON数据结构

从结构列表中我们可以发现，当前的JSON数据一共包含了12个对象，其中main包含了5个与温湿度相关的信息，name是所定位的城市名称，id代表的是城市的ID，coordinate是定位的经纬度坐标，weather是天气情况等。

回到WeatherViewController.swift文件，在Write the updateWeatherData method here：注释的下面添加一个新的方法：

```
//Write the getWeatherData method here:
func getWeatherData(url: String, parameters: [String: String])
{
  Alamofire.request(url, method: .get, parameters:
parameters).responseJSON {
    response in
    if response.result.isSuccess {
      print("成功获取气象数据")
      let weatherJSON: JSON = JSON(response.result.value!)
      // 删除之前的print语句，添加对updateWeatherData()方法的调用。这里一定要使用`self.`，否则编译器会报错。
      self.updateWeatherData(json: weatherJSON)
    }else {
      print("错误 \(String(describing: response.result.error))")
      self.cityLabel.text = "连接问题"
    }
  }
}

//MARK: - JSON Parsing
/***************************************************************
*/

//Write the updateWeatherData method here:
```

```
func updateWeatherData(json: JSON) {
}
```

---

updateWeatherData（）带有一个JSON类型的参数，我们会从getWeatherData（）方法传递weatherJSON常量到该方法。在调用updateWeatherData（）方法的时候，一定要通过self.方式。因为平常Swift会智能地在文件内部通过方法名来搜索并调用指定的方法，而Alamofire方法的执行代码中包含了闭包，它是一种在函数内部实现的简单函数。我们会在之后有详细的讲解。

总而言之，只要你在调用方法的时候看到了in关键字，就可以确定其在使用闭包，也就是正在函数中使用函数。如果此时在闭包（函数里面的函数）里面调用方法或者函数的话，编译器就会产生混淆，不知道调用的是在哪里声明的函数。所以在闭包中调用方法的规则是：只要看到in关键字，就需要在调用方法的前面使用self。

继续在updateWeatherData（）方法中添加下面的代码：

```
func updateWeatherData(json: JSON) {
  let tempResult = json["main"]["temp"]
}
```

如果仔细查看在控制台中打印的JSON数据，会发现城市的温度值是json字典中的main键里面的temp键的值。为了可以导航到该值，我们需要通过json["main"]获取到main的值，这个值还是一个字典类型，所以再通过json["main"]["temp"]获取到temp的值。

10.10　创建气象数据模型

通过前面的学习，我们已经了解了气象数据的结构，以及如何获取到需要的信息的方法。接下来，我们要将这些信息呈现到屏幕上。为了让信息更加方便维护，我们要创建一个数据模型。它会包含那些我们需要使用的属性，比如气象信息、温度等。

在项目导航中打开Model文件夹中的WeatherDataModel.swift文件。在该类中需要创建4个属性，如表10-1所示。

表10-1　需要创建的气象属性说明

其中，第一个变量temperature是城市的温度，Int类型，初始值为0。第二个变量是condition，对应JSON数据中的weather-id，也是Int类型。city是城市名称，字符串类型。weatherIconName是相应天气的图标文件名，也是字符串类型。

在WeatherDataModel类中添加4个属性：

```
class WeatherDataModel {

  //Declare your model variables here
  var temperature: Int = 0
  var condition: Int = 0
  var city: String = ""
  var weatherIconName: String = ""
  ……
```

回到WeatherViewController.swift文件中，在类中声明一个WeatherDataModel类型的对象weatherDataModel。

```
//TODO: Declare instance variables here
let locationManager = CLLocationManager()
let weatherDataModel = WeatherDataModel()
```

继续修改updateWeatherData()方法中的代码：

```
func updateWeatherData(json: JSON) {
  let tempResult = json["main"]["temp"].double

  weatherDataModel.temperature = Int(tempResult! - 273.15)
}
```

首先，因为json["main"]["temp"]是JSON类型，所以需要使用.double将其转换为双精度，但是此时的tempResult是可选，所以使用时要将其强制拆包。又因为open-weathermap提供的温度值是国际上的绝对温度值，所以要减去273.15得到摄氏温度值。又因为2个double值相减得到的是double类型值，所以需要借助Int（）方法将其转换为整型值，最后再将其赋值给WeatherDataModel的temperature属性。

继续为weatherDataModel对象的其他属性赋值：

```
func updateWeatherData(json: JSON) {
  let tempResult = json["main"]["temp"].double

  weatherDataModel.temperature = Int(tempResult! - 273.15)
  weatherDataModel.city = json["name"].stringValue
  weatherDataModel.condition = json["weather"]["id"].intValue
}
```

通过json["name"].stringValue可以获取到城市信息。通过json["weather"][0]["id"].intValue可以获取到该城市的气象情况，只不过该情况是用整型值表示的。通过openweathermap.org/weather-conditions链接可以查阅代码的含义，以及相应的图标，如图10-33所示。

图10-33　openweathermap提供的各种气象信息代码

在WeatherDataModel类中可以看到预先定义好的updateWeatherIcon（）方法，去除其全部的注释语句。在Switch语句部分，已经按照气象代码为我们建立好了所有代码。该方法通过condition参数接收传递进来的气象代码，然后会返回相应的字符串类型的气象名称，这个名称与项目中Images.xcassets文件里面的Icon名称一一对应。

例如，传递进的参数为701，则会执行case 701...771：中的代码，也就是return"fog"，方法会返回字符串fog，因为参数值从701至771都会执行这段代码。通过之后的代码，我们会将Images.xcassets文件里面的fog图标呈现到屏幕上，如图10-34所示。

图10-34　与气象信息对应的图片素材图标

在实现updateWeatherIcon（）方法以后，就可以在updateWeatherData（）方法中添加下面的代码：

```
func updateWeatherData(json: JSON) {
  ……
  weatherDataModel.condition = json["weather"]["id"].intValue
  weatherDataModel.weatherIconName =
weatherDataModel.updateWeatherIcon(condition:
weatherDataModel.condition)
}
```

通过上面的代码，我们就可以为WeatherDataModel对象中的4个属性赋值。

在目前的项目中还有一个问题需要解决，如果在项目中错误设置了APP_ID的值，应用程序在运行的时候就会崩溃，如图10-35所示。如果打印JSON数据的话，可以发现错误是因为无效的API key所致。

图10-35　输入了错误的API Key导致的应用运行崩溃情况

通过打印到控制台的信息我们可以发现，openweathermap返回的是无效API key（因为我们修改了APP_ID）的JSON信息，所以并没有我们需要的气象信息，这也就意味着tempResult的值为nil，在我们对tempResult强制拆包的时候，应用程序发生崩溃。

为了解决这个问题，我们使用可选绑定的方式来为WeatherDataModel对象赋值。修改之前的代码如下：

```
func updateWeatherData(json: JSON) {
  if let tempResult = json["main"]["temp"].double {
    weatherDataModel.temperature = Int(tempResult - 273.15)
    weatherDataModel.city = json["name"].stringValue
    weatherDataModel.condition = json["weather"]["id"].intValue
    weatherDataModel.weatherIconName =
weatherDataModel.updateWeatherIcon(condition:
weatherDataModel.condition)
  }else {
    cityLabel.text = "气象信息不可用"
  }
}
```

此时，如果tempResult的值为nil的话，则不会执行if语句内部的代码，也就不会发生崩溃的情况，而且在else语句中会将信息显示到屏幕上，如图10-36所示。

接下来，我们会实现更新用户界面的相关代码。

在WeatherViewController类中注释语句Write the updateUIWithWeatherData method here：的下面添加一个方法：

```
func updateUIWithWeatherData() {
  cityLabel.text = weatherDataModel.city
  temperatureLabel.text = String(weatherDataModel.temperature)
```

```
  weatherIcon.image = UIImage(named:
weatherDataModel.weatherIconName)
}
```

在该方法中不需要任何参数，我们直接使用weatherDataModel常量即可。最后在updateWeatherData（）方法中添加对该方法的调用。

```
func updateWeatherData(json: JSON) {
  if let tempResult = json["main"]["temp"].double {
    weatherDataModel.temperature = Int(tempResult - 273.15)
    weatherDataModel.city = json["name"].stringValue
    weatherDataModel.condition = json["weather"]["id"].intValue
    weatherDataModel.weatherIconName =
weatherDataModel.updateWeatherIcon(condition:
weatherDataModel.condition)
    // 更新控制器中的UI控件
    updateUIWithWeatherData()
  }
  ……
```

构建并运行项目，可以看到在模拟器中正常显示的气象信息。另外，还可以通过之前介绍的https://www.latlong.net/网站查找到你所在的城市经纬度，然后在模拟器菜单中选择Debug/Location/Custom Location...，手动设置经纬度值，如图10-37所示。

图10-36　处理获取气象信息

图10-37　在模拟器中通过指定经纬度

不可用的情况值模拟特定城市

10.11　Segues的相关介绍

本节我们会了解什么是Segues，以及如何使用Segues。请先打开项目中的故事板，可以看到里面有两个视图控制器。

Segues是Interface Builder里面功能最强大的东西，它允许我们从一个控制器连接到另一个，或者是从控制器中的一个UI控件（比如按钮）连接到另一个控制器。

在当前的项目中，为了可以从WeatherViewController切换到CityViewController，我们创建了一个Segue，点亮Segue让其变成蓝色，然后就可以发现引发这个Segue的原始控件就是Weather控制器视图右上角的切换城市按钮，如图10-38所示。也就是说当我们单击这个切换按钮以后，Segue会自动导航到City控制器视图。幸运的是，这一切已经在初始化项目中设置好了。

接下来，我们主要来了解一下如何在故事板中创建Segue，以及如何在两个控制器之间传递数据。

实战：创建一个Segue。

步骤1：创建一个全新的Single View App项目，Project Name设置为Segues。此时在故事板中只有一个控制器视图。

步骤2：从对象库中拖曳一个新的视图控制器到故事板里面，此时故事板中变成了2个控制器视图。Interface Builder会报错，因为第二个视图不可达，也就是说它没有任何进入点或者标识，无法将该控制器呈现到屏幕上。

步骤3：从对象库拖曳一个Button到第一个控制器视图中，然后按住鼠标右键并将它拖曳到第二个视图控制器。此时会建立一条Segue，Segue的原点是按钮，终点是第二个视图控制器。在弹出的快捷菜单中，会发现有很多不同类型的选项。如图10-39所示。

图10-38　故事板中的Segue

图10-39　在故事板中创建Segue

其中，show是最简单的连接方式，它会以滑动的方式在屏幕上呈现第二个控制器视图。当Segue创建好以后，在控制器之间会有一个箭头。

如果此时构建并运行项目，在单击按钮以后会进入第二个视图控制器。

需要清楚的是，在项目导航中的ViewController.swift文件与故事板中的第一个视图控制器有关联。在故事板选中第一个控制器，在Identifier Inspector中的Custom Class部分中可以看到Class被设置为ViewController，代表当前的控制器指向ViewController.swift文件。

如果选中第二个视图控制器，你会发现它并没有关联的任何类，并且在助手编辑器模式中，也没有与之相匹配的实体类文件存在。接下来，我们就要创建与之关联的类文件，并将其关联到故事板中的第二个控制器。

步骤4：在项目导航中选中Segues文件夹（黄色图标），使用快捷键Command+N创建一个新的iOS/Cocoa Touch Class类型的文件，单击Next按钮。将Class设置为SecondViewController，确保Subclass of设置为UIViewController，Language为Swift。

步骤5：在文件创建好以后，接下来我们需要将该代码文件与故事板中的第二个控制器关联起来。在故事板中选中第二个控制器，在Identifier Inspector中将Class设置为SecondViewController。现在，在助手编辑器模式中就会发现第二个控制器已经与SecondViewController关联起来。

在故事板中单击建立好的Segue连接，通过高亮提示可以清楚地知道当用户单击按钮以后就会切换到Second控制器。除此之外还有第二种方法，先选中Segue并按Delete键将其删除。然后在故事板中从第一个控制器顶部的黄色图标按住鼠标右键并将其拖曳到第二个控制器，如图10-40所示，最后在快捷菜单中选择show，点亮Segue连接，可以发现它的原点现在是第一个控制器。

图10-40　通过控制器顶部的图标创建Segue

这时如何实现单击按钮以后进入到第二个控制器呢?

首先，选中故事板中的Segue，然后在Identifier Inspector中将其Identifier设置为goToSecondScreen。这里我们为这个Segue设置了一个标识，便于在代码中访问该Segue。

然后，为第一个控制器的按钮控件建立IBAction方法，名称定义为buttonPressed。

```
@IBAction func buttonPressed(_ sender: UIButton) {
  performSegue(withIdentifier: "goToSecondScreen", sender:
self)
}
```

performSegue（）方法用于执行一个Segue连接，其中第一个参数代表要执行的Segue标识，也就是之前在故事板中设置的Identifier。第二个参数是这个Segue的发起者，因为是第一个控制器发起的，所以这里使用self。

使用这种方式的好处在于，如果App需要用户在单击某个按钮以后，根据不同的情况呈现不同控制器到屏幕上，则需要创建多个Segue，再根据情况执行特定的Segue。

构建并运行项目，单击按钮以后在屏幕上依然会呈现第二控制器。

接下来，我们需要在切换控制器的时候从第一控制器传递一些数据到第二控制器。

步骤1：在第一个控制器的视图中添加一个TextField，在第二控制器的视图中添加一个Label控件如图10-41所示。这样做的目的是让用户在Text Field中输入一些文字，单击按钮以后，在第二控制器的Label中显示这些内容。

图10-41　设置两个控制器视图的用户界面

步骤2：在第一控制器中为Text Field控件创建IBOutlet关联，名称叫作textField。在第二控制器为Label控件创建IBOutlet关联，名称叫作label。

步骤3：在SecondViewController类中声明一个属性var textPassedOver：String？，用于接收从ViewController传递过来的字符串。

步骤4：在ViewController类中添加一个新的方法。

```
override func prepare(for segue: UIStoryboardSegue, sender:
Any?){
  if segue.identifier == "goToSecondScreen" {
    let destinationVC = segue.destination as!
SecondViewController
    destinationVC.textPassedOver = textField.text!
  }
}
```

每个控制器类都包含prepareForSegue（）方法，当控制器发生切换的时候，原始控制器类就会在发生切换之前先执行该方法。这里我们要使用override关键字重写该方法，因为在父类中也定义了此方法。

在方法中根据参数segue的identifier属性判断当前的视图切换是否为goToSecond-Screen，然后利用segue的destination属性获取到目标控制器对象，因为只有我们自己知道该控制器对象是SecondViewController类型，但是编译器并不知道，因此需要通过强制转换代码as！SecondViewController将其转换为SecondViewController类型。只有这样，才能在下一行为destination对象的textPassedOver属性赋值。

步骤5：在SecondViewController类的viewDidLoad（）方法中添加一行代码。

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

  label.text = textPassedOver
}
```

构建并运行项目，在Text Field输入的信息会显示到Label中。

10.12　在项目中使用委托和协议

目前为止，我们的项目可以完美地告诉用户当前位置的气象信息，如果运行在iPhone真机上的话，就可以告诉你真正位置的气象数据。如果你特别想在模拟器中显示当前位置的气象数据，可以在模拟器菜单中选择Debug/Location/Custom Location...，然后设置Latitude和Longitude参数即可。

如果我们需要获取其他指定城市的气象数据，则需要借助项目中的CityViewController控制器了。因此在故事板中我们将切换按钮与City控制器建立Segue关联。在City控制器视图中有一个Text Field，允许我们输入一个城市的名称，比如guangzhou，然后单击Get Weather按钮获取该城市的气象数据。

在项目导航中打开CityViewController.swift文件，可以看到在IBAction方法get-WeatherPressed（）里面有下面这些注释代码，我们需要在后面实现这些代码。另外，该类还有一个与Text Field关联的IBOutlet变量changeCityTextField。

```
//1 通过Text Field得到城市名称

//2 如果有一个delegate设置，则调用userEnteredANewCityName()方法

//3 销毁CityViewController并返回到WeatherViewController
```

通过上面的注释语句我们可以了解getWeatherPressed（）方法的功能：当用户输入城市名称并单击Get Weather按钮以后，需要将数据回传给WeatherViewController。

为了可以在两个完全不同的控制器之间传递数据，我们需要学习委托（Delegate）和协议（Protocol）的相关知识。委托和协议是swift语言

中比较高级的主题，如果读者理解有困难的话可以反复多看几遍这部分内容。

当我们根据CLLocationManager创建该类型的一个对象后，将WeatherViewController作为它的委托对象（通过locationManager.delegate=self）。也就是说，当从CLLocationManager对象发送相关信息以后，WeatherViewController便可以响应并接收这些信息。

因为CoreLocation框架没有开源，所以无法看到其内部的执行代码。但是通过开发文档我们可以知道，一旦locationManager找到了确定位置，就会通过其delegate属性确定要把位置数据报告给谁（当前项目是WeatherViewController对象）。

在我们设置了WeatherViewController作为CLLocationManager的Delegate以后，我们会通过didUpdateLocations和didFailWithError方法来确定，当接收到CLLocationManager对象发来的数据以后要做什么。换句话说就是当定位成功时会通过didUpdateLocations方法接收数据，当定位失败时会通过didFailWithError方法获取错误信息。

所以，在我们使用苹果自己的非开源框架代码的时候，可以通过文档了解其委托的实现方法，进而实现自己需要的功能。

但是，当前项目中需要实现的功能是完成两个控制器之间的切换和数据的传递，而且这两个控制器代码都是可见的，我们需要创建属于自己的协议和委托。

实战：创建自定义的协议和委托。

步骤1：在CityViewController.swift文件的顶部，CityViewController类的外部创建一个协议声明。

```
//Write the protocol declaration here:
protocol ChangeCityDelegate {
```

```
  func userEnteredANewCityName(city: String)
}
```

使用protocol关键字创建协议，后面是协议的名称ChangeCityDelegate。在该协议的内部仅包含一个方法userEnteredANewCityName（city：String），它的参数用于传递城市的名称。我们想要在用户单击Get Weather按钮以后激活一个发送到WeatherViewController的消息，这个消息包含了用户在Text Field中输入的城市名称。

实际上你可以把协议看成是一个约定，现在我们只是在起草这个约定。要想让一个类成为另一个类的delegate，就需要让它实现userEnteredANewCityName（city：String）方法。也就是从ChangeCityViewController类向WeatherViewController发送userEnteredANewCityName消息并携带城市名称参数的时候，在WeatherViewController类中必须要有相应的处理方法。

在起草了约定以后，还需要在ChangeCityViewController类中声明一个新属性delegate。

步骤2：在ChangeCityViewController类中添加一个新的属性。

```
var delegate: ChangeCityDelegate?
```

delegate变量的类型为可选ChangeCityDelegate，意味着我们只能将符合ChangeCity-Delegate协议的视图控制器对象赋值给它，或者在没有赋值的情况下它的值为nil。

步骤3：项目导航中打开WeatherViewController.swift文件，让WeatherViewController类遵守新添加的约定。

```
class WeatherViewController: UIViewController,
CLLocationManagerDelegate, ChangeCityDelegate {
```

此时编译器报错：WeatherViewController不符合ChangeCityDelegate协议。虽然我们要在类声明中让WeatherViewController类遵守ChangeCityDelegate协议，但是目前还没有在类中实现该协议中的userEnteredANewCityName（city：String）方法。另外，在点开错误叹号以后Xcode会自动帮助我们添加这个协议方法，如图10-42所示。

图10-42　WeatherViewController中还没有实现委托方法

由此可见，协议中的方法是双方约定好的，执行对象在必要的时候会向委托对象发送消息（也就是协议方法名称），因此委托对象就必须要定义这样的协议方法来处理相应的情况。

步骤4：打开WeatherViewController.swift文件，在Write the userEnteredANewCityName Delegate method here：注释语句的下面添加新的方法。

```
func userEnteredANewCityName(city: String) {
  print(city)
}
```

ChangeCityViewController在需要发送消息的时候，会通过delegate属性调用该方法，例如delegate.userEnteredANewCityName（"beijing"）。因此，在从WeatherViewController切换到ChangeCityViewController的时候，需要将WeatherViewController对象赋值给Change-CityViewController的delegate属性。

步骤5：在WeatherViewController的Write the PrepareForSegue Method here注释的下面添加新的方法。

```
override func prepare(for segue: UIStoryboardSegue, sender:
Any?) {
```

```
  if segue.identifier == "changeCityName" {
    let destinationVC = segue.destination as!
ChangeCityViewController
    destinationVC.delegate = self
  }
}
```

与上一节的方法类似，当两个控制器切换的时候，我们通过prepare（）方法的segue参数获取到目标视图控制器，其中changeCityName是这个Segue的Identifier的属性值，我们可以通过故事板中的Identifier Inspector中查到。

其次，该Segue的目标控制器是ChangeCityViewController，所以通过Segue的destination属性获取该控制器对象，因为destination是UIViewController类型，所以需要使用as！向下强制将其转换为ChangeCityViewController类型。

最后，将WeatherViewController对象自身赋值给ChangeCityViewController对象的delegate属性，这样我们就可以在ChangeCityViewController中向WeatherViewController发送消息了。

步骤6：在ChangeCityViewController类中，添加下面的代码。

```
@IBAction func getWeatherPressed(_ sender: AnyObject) {

  //1 通过Text Field得到城市名称
  let cityName = changeCityTextField.text!

  //2 如果有一个delegate设置，则调用userEnteredANewCityName()方法
  delegate?.userEnteredANewCityName(city: cityName)

  //3 销毁CityViewController并返回到WeatherViewController
  self.dismiss(animated: true, completion: nil)
}
```

在该方法中，先从Text Field获取到用户输入的城市名称，然后通过delegate属性调用WeatherViewController类的

userEnteredANewCityName方法，并将城市名称作为参数传递过去。然后通过dismiss（）方法销毁当前视图控制器，它有两个参数，animated代表以动画的方式销毁，completion代表控制器在销毁以后还运行什么代码，这里使用nil代表不运行任何代码。

为什么在delegate后面会有一个问号（？）呢？这涉及可选链（Optional Chaining）的概念，因为delegate是可选类型，所以它可能有值或是nil。delegate？.userEnteredANewCityName（city：cityName）代表如果有值则会继续执行方法，如果为nil则自动忽略。这也是可选绑定了另一个用途。

构建并运行项目，在Text Field中输入一个城市名称，单击按钮以后在控制台中会打印出这个名称。

10.13　如何在视图控制器间传递数据

在我们自己创建的应用程序中，会发现经常要在视图控制器间传递数据，这就是委托和协议存在的原因。本节的主要任务是让读者再次熟悉控制器之间数据传递的具体过程，因此本节的项目非常简单，重点在于实现的代码上。

在GitHub上面搜索“liumingl/PassDataBetweenVC”关键字，然后下载并解压缩PassDataBetweenVC项目到本地磁盘中，在Xcode中打开该项目的Main.storyboard文件，如图10-43所示。

本项目共包含两个控制器——First控制器和Second控制器，它们两个之间由一个Segue连接，其Identifier为sendDataForwards。First和Second控制器各有label和text-Field两个IBOutlet关联，First控制器中与按钮关联的IBAction方法是send-Button-Pressed（），Second控制器与按钮关联的IBAction方法是sendDataBack（）。

图10-43　PassDataBetweenVC项目中的两个控制器界面

应用的运行流程是：First控制器是启动控制器，用户在文本框中输入一些文字以后单击按钮，屏幕会切换到Second控制器视图，Label2会显示用户在文本框中输入的信息。用户此时在Second控制器的文本框中输入一些文字，单击按钮以后返回到First控制器，并且在Label1中会显示用户之前输入的信息。

实战：在两个控制器之间传递数据。

步骤1：在项目导航中打开FirstViewController.swift文件，重写prepare（）方法。

```
override func prepare(for segue: UIStoryboardSegue, sender:
Any?) {
  if segue.identifier == "sendDataForwards" {
    let secondVC = segue.destination as! SecondViewController
    secondVC.data = textField.text!
  }
}
```

当从First控制器切换到Second控制器的时候，会调用该方法。如果执行的是标识为是sendDataForwards的Segue，则通过Segue的destination属性获取到目标控制器对象。然后将文本框中的字符串赋值给Second控制器的data属性。

需要说明的是，这里我们是通过Segue的destination属性获取到Second控制器对象。如果用let secondVC=SecondViewController（）虽然也可以创建新的Second控制器对象。但是，通过这种方式创建的对象并不是故事板中Segue所指向的已经实例化好的目标对象。

步骤2：在sendButtonPressed（）方法中添加下面的代码。

```
@IBAction func sendButtonPressed(_ sender: UIButton) {
  performSegue(withIdentifier: "sendDataForwards", sender:
self)
}
```

当用户单击“给我来点刺激的”按钮以后，会执行标识为sendDataForwards的Segue，控制器切换启动，激活执行prepare（）方法。

步骤3：编辑SecondViewController.swift文件，在viewDidLoad（）方法中添加对label的赋值代码。

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

```
  label.text = data
}
```

构建并运行项目，在First控制器输入文字，单击按钮以后会呈现到Second控制器的Label上，如图10-44所示。

图10-44　从First控制器传递字符串数据到Second控制器

接下来你可能会想到使用同样的方式，从Second控制器发送数据到First控制器。如果你想试试的话，咱们就一起来付诸实践，因为实践是最好的老师！并且我还要向你证明为什么这样做不行。

步骤4：在故事板中从Second控制器到First控制器创建一个新的Segue，设置它的Identifier为sendDataBack。

步骤5：在FirstViewController类中添加一个新的属性：var dataPassedBack=""。

步骤6：在SecondViewController类中添加prepare（）方法。

```
override func prepare(for segue: UIStoryboardSegue, sender:
Any?) {
  if segue.identifier == "sendDataBack" {
    let firstVC = segue.destination as! FirstViewController
    firstVC.dataPassedBack = textField.text!
  }
}
```

步骤7：在sendDataBack（）方法中，激活sendDataBack Segue。

```
@IBAction func sendDataBack(_ sender: UIButton) {
  performSegue(withIdentifier: "sendDataBack", sender: nil)
}
```

步骤8：在FirstViewController的viewDidLoad（）方法中添加如下代码。

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

  label.text = dataPassedBack
}
```

构建并运行项目，确实可以将Second控制器中的文字信息传回给First控制器，但是这里面存在一个非常严重的问题：控制器在每次进行切换的时候，都会产生一个新的控制器对象，循环往复，无穷尽也！随着切换次数的不断更加，App所占据内存的空间也不断增加，直到将系统的内存全部“吃光”。

为了验证这一点，修改First控制器中的sendButtonPressed（）方法，添加如下代码。

```
@IBAction func sendButtonPressed(_ sender: UIButton) {
  view.backgroundColor = UIColor.blue
  performSegue(withIdentifier: "sendDataForwards", sender:
self)
}
```

当用户单击按钮以后，会将First控制器视图的背景色从粉红色改为蓝色。

构建并运行项目，当单击First控制器按钮的时候请仔细观察背景色的变化，在其闪变为蓝色后，会进入到Second控制器。在Second控制器中单击按钮以后，按照我们的想法应该会回到之前蓝色背景的First控制器。但结果并非这样，此时又会进入到一个粉红色背景的First控制器。重复这样的操作依旧如此，屡试不爽！

实际上，我们通过这种方式在进行控制器切换的时候，都在创建新的控制器拷贝，就好像是A（First对象）传给B（Second对象），B传给C（First对象2），C传给D（Second对象2）……然后在这些控制器之间传递着数据。你以为是在两个控制器之间来回切换，其实是在不断地创建新的控制器。

实战：控制器回调的正确方法。

步骤1：删除FirstViewController类中的dataPassedBack属性，以及viewDidLoad（）方法中的label.text=dataPassedBack代码。删除SecondViewController类中的prepare（）方法，以及sendDataBack（）方法中的代码。在故事板中删除Identifier为sendDataBack的Segue。

如何在First控制器中获取到Second控制器发过来的文字信息呢？答案是需要使用委托和协议。

步骤2：在SecondViewController.swift文件中创建一个协议。

```
protocol CanReceive {
  func dataReceived(data: String)
}
```

当前定义的协议名称为CanReceive，它仅包含一个required方法——dataReceived（），data参数用于回传文字信息。所谓required方法就是委托对象必须实现的方法。

其实，协议相当于类级别代码，我们完全可以将其定义到一个单独的文件里面，但是为了方便使用，就直接将其写到SecondViewController.swift里面。

注意　绝对不能将协议写到SecondViewController类的内部，它们是独立的两个部分。

就像是球队的教练，协议本身不会实现任何的功能，就像教练不需要在赛场上扣篮和运球一样，它仅仅是约定的规则而已。

步骤3：接下来，我们要让FirstViewController符合这个协议。

```
class FirstViewController: UIViewController, CanReceive {
```

在Xcode中标记为父类的高亮颜色和协议的高亮颜色是不一样的，我们可以利用这一点来区别父类和协议。

此时编译器报错：FirstViewController不符合CanReceive协议。这是因为还没有实现协议方法。

步骤4：在FirstViewController类中实现协议方法。

```
func dataReceived(data: String) {
  label.text = data
}
```

现在，我们已经创建好了协议，并让FirstViewController类符合该协议。接下来需要让FirstViewController类成为委托对象，并且在用户单击按钮的时候激活协议方法。

步骤5：在SecondViewController类中添加一个delegate属性。

```
var delegate: CanReceive?
```

delegate属性的类型是协议名称，并且它是可选类型，因为delegate可能是nil。如果程序不需要Second控制器返回数据给First控制器，则不需要设置delegate。

步骤6：我们需要在用户单击按钮以后发送文字返回First控制器，所以要在sendDataBack（）方法中添加delegate方法的调用。

```
@IBAction func sendDataBack(_ sender: UIButton) {
  delegate?.dataReceived(data: textField.text!)
  dismiss(animated: true, completion: nil)
}
```

步骤7：在FirstViewController类的prepare（）方法中，需要让自身成为SecondView-Controller类的delegate。

```
override func prepare(for segue: UIStoryboardSegue, sender:
Any?) {
  if segue.identifier == "sendDataForwards" {
    let secondVC = segue.destination as! SecondViewController
    secondVC.data = textField.text!
    secondVC.delegate = self
  }
}
```

注意　该方法中的secondVC对象是在初始化Segue的时候就创建好了。

现在，我们已经成功完成了协议和委托。让我们重新捋一捋思路。

首先，创建一个协议，该协议包含一个required方法。然后，让接收数据的控制器符合这个协议，并且还要实现required委托方法。接下来，创建一个delegate属性，类型与协议名称一致，并且是可选。如果delegate有值存在的话，在特定的时候可以通过它执行delegate指向的类里面的委托方法。如果需要销毁当前控制器还需要执行dismiss（）方法。最后，将接收对象设置为delegate即可。

构建并运行项目，单击Second控制器按钮以后，会返回到之前的First控制器，因为它的背景色为蓝色。

10.14　基于城市名称的天气数据请求

现在的Weather项目离完成已经只差最后一步了，在WeatherViewController类中的userEnteredANewCityName（）已经接收了更改的城市名称，目前我们只是将这个城市名称打印到控制台，本节我们需要将城市名称发送到openweathermap服务器，并得到它的气象数据。所有的工作流程与之前获取气象数据差不多。

修改userEnteredANewCityName（）方法。

```
func userEnteredANewCityName(city: String) {
  let params: [String: String] = ["q": city, "appid": APP_ID]
  getWeatherData(url: WEATHER_URL, parameters: params)
}
```

在该方法中，我们重新定义了参数字典，字典包含两个Key，第一个q代表城市的名称，第二个Key代表openweathermap服务的注册ID。为什么第一个Key是q呢？在open-weathermap网站的API文档中，你可以找到答案，如图10-45所示。

另外，为了让temperatureLabel中的摄氏温度显示得更好看，可以这样修改：

```
func updateUIWithWeatherData() {
  cityLabel.text = weatherDataModel.city
  temperatureLabel.text = String(weatherDataModel.temperature)
+ "°"
  weatherIcon.image = UIImage(named:
weatherDataModel.weatherIconName)
}
```

其中这个温度符号，可以通过macOS系统的图标输入法调出，使用快捷键Control+Command+空格键调出表情与符号对话框，然后在搜索

中输入degree就可以找到需要的符号，如图10-46所示。

构建并运行项目，选择一个你比较关注的城市（比如guangzhou、shenzhen、harbin等），然后单击Get Weather按钮，就可以看到实时的气象信息，如图10-47所示。

图10-45　openweathermap网站提供的API参数说明

图10-46　输入标准的摄氏温度符号

图10-47　切换到指定的城市

10.15　挑战：利用Cocoapods、REST和APIs构建比特币价格跟踪应用

在本书最开始的部分，我们创建了一个非常简单的应用程序——I am Rich，它只包含一个图像和一个标签。在本节我们会制作一个真正的I am Rich应用程序。利用之前我们所学过的CocoaPods，并利用API进行网络连接，然后从网站抓取数据。

这个项目是一个非常大的挑战，但是却非常有意思。它是一个比特币价格跟踪的应用。在启动它以后，你会在屏幕上看到三个UI元素，如图10-48所示。

图中的顶部是一个静态图片，是一个比特币的Logo。它的下面是一个Price标签，用于显示一个比特币在某个特定货币上的价值，最下面是使用UI Picker控件让我们滚动选择当前的货币，之后会在Price标签中更新比特币的价格。

实战：制作比特币价格跟踪应用。

步骤1：在GitHub上面下载Bitcoin-Ticker项目的初始骨架，将其解压缩到桌面。

在项目导航中打开Main.storyboard故事板文件，查看其中的UI元素。这里面包含一个Image View控件用于显示比特币的Logo，用户不会去单击或修改它，因此它并没有IBOutlet关联。黄色的Price标签在控制器中有一个IBOutlet关联叫作bitcoin-Price-Label，底部的滚动转盘（UIPicker控件）在控制器中也有一个IBOutlet关联，叫作currency-Picker，如图10-49所示。

图10-48　比特币价格跟踪应用

图10-49　故事板中的用户界面

步骤2：设置和使用UIPicker类。

故事板中的UIPicker其实是一个滚动轮盘，我们用它来显示可以查询比特币价格的所有可用货币。在移动设备开发中，这是一种最简单的选项选择方式，让程序员无须使用大量的按钮，从而节省了宝贵的屏幕空间。而且它的设置非常简单，只需简单的几个步骤即可完成。

首先，打开ViewController.swift文件，在ViewController类声明的顶部，添加UIP-icker-ViewDataSource和UIPickerViewDelegate协议，如图10-50所示。

接下来，在viewDidLoad（）方法中，将UIPickerView的delegate和dataSource属性均设置为self。

图10-50　添加协议

当前，Xcode将会有一些报错呈现给你。这是因为还需要将一些必需的委托方法添加到代码之中，以符合UIPickerViewDelegate协议。

首先，添加numberOfComponents（）方法以确定我们在滚动转盘中需要多少列。

```
func numberOfComponents(in pickerView: UIPickerView) -> Int {
  return 1
}
```

其次，使用pickerView（numberOfRowsInComponent：）方法告诉Xcode这个滚动转盘有多少行，可以使用数组的count方法来获取这个数据。

```
func pickerView(_ pickerView: UIPickerView,
numberOfRowsInComponent component: Int) -> Int {
  return currencyArray.count
}
```

最后，使用pickerView：titleForRow：方法将currencyArray数组中的字符串填充到滚动转盘的标题上。

```
func pickerView(_ pickerView: UIPickerView, titleForRow row:
Int, forComponent component: Int) -> String? {
  return currencyArray[row]
}
```

构建并运行项目，验证滚动转盘是否实现了我们的意图。接下来我们需要响应用户与转盘之间的交互。

将pickerView：didSelectRow：委托方法放在刚刚创建的其他方法的下面，以告诉滚动转盘用户选择了特定的行。

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

当用户在转盘选择完成以后，就会调用该方法，这里暂时先将用户选择的行号打印到控制台。

构建并运行项目，查看控制台日志是否为你选择后的期望值。其实，在控制台中显示行号并不直观。我们可以更改该打印语句以打印用户所选择的货币。

```
func pickerView(_ pickerView: UIPickerView, didSelectRow row:
Int, inComponent component: Int) {
```

```
  print(currencyArray[row])
}
```

步骤3：构建API URL格式。

我们将通过访问bitcoinaverage.com网站来获取比特币的货币价格值。利用该网站提供的比特币API，以特定货币获取比特币的当前价值的网址格式为：https://apiv2.bitcoinaverage.com/indices/global/ticker/BTC<货币>，例如：https://apiv2.bitcoinaverage.com/indices/global/ticker/BTCCNY获取人民币的比特币价格。

我们已经将所有的货币名称存储在currencyArray数组之中。通过这个数组和用户选择的行号来组成API调用的URL链接。这里已经创建了一个baseURL变量来存储API调用链接，即在货币代码之前的URL中的所有内容。

修改pickerView：didSelectRow：方法如下。

```
func pickerView(_ pickerView: UIPickerView, didSelectRow row:
Int, inComponent component: Int) {
  finalURL = baseURL + currencyArray[row]
  print(finalURL)
}
```

构建并运行项目，当选择好币种以后，确保在控制台中可以看到正确的URL链接。

步骤4：在项目中设置Cocoapods。

与Weather项目类似，我们需要在此项目中使用Alamofire和SwiftyJSON库。尝试着自己将它们整合到这个BitcoinTicker项目中。

·打开macOS的终端（Terminal）应用程序，将目录更改为包含Bitcoin Ticker项目的文件夹。

·使用pod init命令，初始化一个新的Podfile。

·在Xcode中打开Podfile文件。

·添加两个库（SwiftyJSON和Alamofire）。

·确保删除platform：ios，'9.0'前面的注释标记。

·在终端中运行pod install。

·打开.xcworkspace文件。

步骤5：Networking调用。

如果你向下滚动ViewController.swift文件，会看到我们复制并粘贴了Weather项目中所有与网络连接的相关代码。

在ViewController.swift中导入Alamofire和SwiftyJSON库。

提示　在键入import语句之前请先使用快捷键Command+B构建一下项目，再导入SwiftyJSON和Alamofire库，防止编译器报错。

取消与网络连接相关的所有代码注释。

更新getWeatherData（）方法以使其适用于我们当前的项目。

```
func getBitcoinData(url: String) {

  Alamofire.request(url, method: .get)
    .responseJSON { response in
      if response.result.isSuccess {

        print("成功！已经获取到比特币数据")
        let bitcoinJSON : JSON = JSON(response.result.value!)

        self.updateBitcoinData(json: bitcoinJSON)

      } else {
        print("Error: \(String(describing:
```

```
response.result.error))")
        self.bitcoinPriceLabel.text = "Connection Issues"
      }
  }
}
```

步骤6：解析JSON数据。

就在网络连接代码的下面，我们已经包含了解析Weather项目复制的JSON代码。修改此代码，以使它适用于我们当前的项目。

```
func updateBitcoinData(json: JSON) {
  if let bitcoinResult = json["last"].rawString() {
    bitcoinPriceLabel.text = bitcoinResult
  }
}
```

提示　你可以在浏览器中键入下面的链接查看JSON格式的数据。https://apiv2.bitcoinaverage.com/indices/global/ticker/BTCCNY

步骤7：进一步修改用户界面。

在Weather项目中，我们有许多标签和图像视图需要更新，这就是为什么我们要将UI更新独立到一个方法中。在这个比特币应用程序中，我们需要更新的唯一控件就是bitcoinPriceLabel标签。我们在上一步已经实现了UI更新代码。

步骤8：完成最后的步骤。

我们已经编写了进行网络调用并解析JSON结果的代码。但是如果你仔细观察，会发现还没有在代码中实现触发网络请求的代码。

启动API调用最合理的地方是用户滚动UIPickerView改变币种的时候，在pickerView：didSelectRow：方法中，调用你的网络方法并传入查找比特币价格所需的URL。

```
func pickerView(_ pickerView: UIPickerView, didSelectRow row:
Int, inComponent component: Int) {
  finalURL = baseURL + currencyArray[row]
}
```

构建并运行项目，查看是否运行正常。

最后的挑战：其实我们可以让应用更加完美，在价格数据的前面加上货币符号不是会更好吗?

我们已经按货币符号出现在currencyArray数组中的顺序输入了所有货币符号，请使用它为每个价格结果提供相应的货币符号。

```
["$", "R$", "$", "¥", "€", "£", "$", "Rp", "?", "", "¥", "$",
"kr", "$", "z?", "lei", "", "kr", "$", "$", "R"]
```

第11章　利用云端数据库、iOS动画和高级Swift特性构建聊天应用

在本章，我们将会构建一个比较成熟的应用——Happy Chat，它会实现类似于聊天室的功能。在创建Happy Chat应用的过程中，我们将会学习关于如何存储数据，如何使用Bmob提供的云服务进行远端数据存储，并且还会深入了解iOS中最常用的表格视图控制器（Table View Controller）组件。在邮件、iMessage、联系人等应用中你会看到表格视图控制器，可以说它是iOS应用程序开发中使用频率最高的组件。

在本章的学习中，我们还会利用CocoaPods整合第三方开源库到项目之中，并且通过在模拟器中生成两个不同的iOS的设备，进行文字信息形式的聊天。除此以外，我们还会借助Bmob云平台服务实现应用的用户注册、登录，并将数据存储到云端。

11.1　关于Bmob

在搭建Happy Chat聊天项目的时候，有一个非常重要的部分就是要借助Bmob云平台。因此本节我们需要了解有关Bmob的知识。

什么是Bmob？它是全方位、一体化的后端服务平台。我们无须再分心去建造后端服务，便可以轻松地拥有开发中所需的各种后端功能。比如最重要的一个特性就是云端与本地的实时数据同步。Bmob可以为我们提供实时数据与文件的存储功能，轻松实现云端与本地的数据连通。而且，数据存储能力除了对常规的文本信息进行存储，还可以存储图片、视频、音频，甚至是地理位置等信息。除此以外，Bmob服务还内置了用户系统，包括用户的注册、登录，还有即时通信、权限控制等，开发者只需使用几行简单的代码就可以实现，如图11-1所示。

图11-1　Bmob的基本功能

最主要的是，我们无须支付任何费用就能够使用Bmob云平台服务的很多基本功能，从而进行各种测试项目的开发。

其实，在现实生活中，有很多是三五个人组成的iOS开发团队，每个人都承担着很多繁杂的任务。如果单独聘请一位服务器端开发和维护人员，将会大大增加开发和运营维护的成本。使用Bmob，我们就不用担心如何去维护服务器端的代码了。因为Bmob会自动帮助我们处理这些事情。

Bmob有一大堆的优秀特性，其中包括数据的存储、身份的认证、数据分析以及通知等。在本章的学习中，我们将会接触它的两个核心特性。第一个就是数据存储特性，我们将会学习如何存储数据到Bmob的云端，并且在从多台iOS设备上获取这些信息。又因为Bmob是跨平台的云端服务，这也就意味着你在iOS设备上将信息发送到Bmob云端，如果相应的安卓系统的应用也在使用该数据的话，那么就可以轻松实

现跨平台聊天功能。Bmob的另一个特性就是允许我们进行身份验证。我们会在Happy Chat中搭建用户注册和登录视图，然后要求用户必须要通过邮件和密码进行账户的注册，然后通过Bmob的认证功能，让用户登录到应用程序，这样他们就可以开始愉快地聊天了。

本节我们只是简单介绍一下Bmob平台最重要的特性与功能，随着后面的学习，我们会逐步进行更深入的了解。

11.1.1　在LeanCloud上注册账户

在GitHub上下载Happy Chat初始化项目。

在初始项目下载完成以后，我们要做的第一件事就是登录Bmob.cn，注册一个用户账号。在创建了Bmob账号以后，我们可以进入到控制台，控制台里面单击创建应用按钮。在创建应用面板中，将应用名称设置为Happy Chat，然后设置应用类型为应用–社交通信，确认勾选开发版（免费）选项，因为其余的选择都会让我们支付一定的费用，最后单击创建应用按钮，如图11-2所示。

图11-2　在Bmob网站注册用户后创建应用项目

11.1.2 设置Bmob

在成功注册了Bmob账号以及下载了初始项目后，现在我们需要在项目中整合一些第三方类库。

实战：整合第三方类库。

步骤1：打开macOS中的终端应用，导航到项目的目录，并执行Pod init命令。打开Pod为我们创建的Podfile文件，添加下面几个所需的第三方类库。

```
# Pods for Happy Chat
pod 'BmobSDK'
pod 'SVProgressHUD'
pod 'ChameleonFramework'
```

其中，BmobSDK是Bmob平台的iOS软件开发工具（Software Development Kit，SDK）。另外两个则是项目中将会用到的两个特性的支持类。

步骤2：执行Pod install命令安装三个开源类库。

```
MacBook-Pro:Happy Chat-Finished liuming$ pod install
Analyzing dependencies
Downloading dependencies
Installing BmobSDK (2.2.8)
Installing ChameleonFramework (2.1.0)
Installing SVProgressHUD (2.2.3)
Generating Pods project
Integrating client project

[!] Please close any current Xcode sessions and use `Happy
Chat.xcworkspace` for this project from now on.
Sending stats
Pod installation complete! There are 3 dependencies from the
Podfile and 3 total pods installed.
```

终端应用会在后台下载所有的库源代码，一旦看到Pod installation complete！信息，并且在其下面没有显示任何的错误信息，则代表安装成功。

步骤3：单击Happy Chat.xcworkspace打开Happy Chat项目，使用快捷键Command+B构建项目。

此时你会发现Xcode编译器会显示若干数量的警告信息。这些信息均与ChameleonFramework相关。这是因为被整合到项目中的ChameleonFramework代码相对于当前的Swift 4语言版本有点老旧，通过一些兼容机制，这些老旧的代码还是可以运行的。虽然对于运行没有任何影响，但是我们还是希望消除这些警告信息。

步骤4：在项目导航中打开Podfile文件，在文件的最后添加下面的语句，保存并退出以后，在终端应用中导航到Happy Chat目录，然后执行Pod update命令更新Podfile的配置。

```
#取消其他第三方库的警告信息
post_install do |installer|
  installer.pods_project.targets.each do |target|
    target.build_configurations.each do |config|

config.build_settings['CLANG_WARN_DOCUMENTATION_COMMENTS'] =
'NO'
    end
  end
end
```

步骤5：重新打开Happy Chat项目，再次构建项目，你会发现Xcode编译器没有显示任何警告。

11.2　保存数据到Bmob

本节我们需要将Bmob整合到Happy Chat项目之中。登录Bmob.cn网站并进入控制台页面。在控制台中单击左侧的应用，然后在应用列表里面找到我们刚创建的Happy Chat应用。单击进入该应用，此时在应用程序的左侧列表里单击设置链接，复制其中的Application ID字符串内容，如图11-3所示。

图11-3　在Bmob中查找Application ID

11.2.1　创建桥接头文件

当前的Bmob SDK是由Objective-C编写的，所以想要在Swift项目中使用Objective-C的类和方法的话，需要创建一个.h头文件，并且把你想在Swift中使用的Objective-C的头文件都包含进来。

创建桥接头文件的方法有两种：可以自己手动创建一个桥接头文件并在项目配置项里面进行设置，也可以使用更快捷的方式，在你的项目里创建一个无用的Objective-C类文件（如：ViewController.m），Xcode将自动询问你是否要创建一个桥接头文件，如图11-4所示。在创建完头文件后，你就可以直接删除ViewController.m文件了，然后在Happy Chat-Bridging-Header.h文件中引入一行代码：

```
#import <BmobSDK/Bmob.h>
```

图11-4　创建桥接头文件

11.2.2 测试云端数据库的读写

实战：编写代码测试应用是否可以读写云端数据库。

步骤1：在项目导航中打开AppDelegate.swift文件，在application：didFinishLaunchingWithOptions：方法中添加如下代码：

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {

Bmob.register(withAppKey: "088e9ca802449a3117c23e1066585629")

return true
}
```

register（）方法用于向Bmob注册当前使用的应用，其中的字符串参数就是之前在Bmob网站上复制的Application ID。应用程序一旦启动就会调用application：didFinishLaunchingWithOptions：方法，所以它会在任何东西呈现在屏幕上面之前被调用，因此这里是配置Bmob的最合适的地方。

步骤2：为了验证连接云端数据库的读写操作是否正常，在AppDelegate类中添加一个新的方法。

```
func save(){
  let gamescore:BmobObject = BmobObject(className: "GameScore")
  gamescore.setObject("刘怀羽", forKey: "playerName")
  gamescore.setObject(90, forKey: "score")
  gamescore.saveInBackground { (isSuccessful, error) in

    if error != nil{
      print("error is \(error!.localizedDescription)")
    }else{
      print("存储成功")
    }
```

```
    }
}
```

在该方法中，我们首先初始化一个BmobObject类型的对象，该对象会查找云端Happy Chat应用中的GameScore数据表，如果数据表不存在的话就直接创建它。然后将在数据表中添加一行数据，将playerName字段设置为乐乐，将score字段设置为90。接下来，通过saveInBackground（）方法将新添加的数据存储到云端的表中。该方法带有一个闭包，如果存储过程结束，则会执行闭包中的代码，打印错误描述或者打印存储成功。

注意　如果云端的GameScore表中不包含playerName和score字段，则Bmob会自动为我们创建它。

步骤3：在application：didFinishLaunchingWithOptions：方法的return true上面一行，添加对save（）方法的调用。

构建并运行项目，当看到控制台显示存储成功信息以后，在浏览器中登录Bmob控制台，可以看到在GameScore表中所添加的信息，如图11-5所示。

图11-5　通过代码在云端数据库添加一行数据

步骤4：如果测试成功，请注释掉之前对save（）方法的调用。

11.2.3　在应用上注册一些用户

在成功将一些信息传输到云端数据表以后，接下来我们要完成在Bmob平台的认证功能，也就是在Happy Chat应用中注册一些新用户。

Bmob的用户注册功能，需要用户提供用户名和密码，另外电子邮件是可选项。在实现注册功能之前，先让我们熟悉一下当前的这个项目的大体设置。

在项目中打开Main.storyboard文件，在故事板里面我们可以发现其中包含了很多控制器视图，如图11-6所示。

图11-6　Happy Chat故事板中的用户界面

首先是欢迎视图，在用户启动应用程序后，最先看到的就是这个控制器的视图。通过该视图，用户可以选择登录或注册操作。当用户单击注册按钮以后，屏幕会切换到注册控制器的视图。在注册视图中，用户只要输入用户名和密码就可以完成注册功能。在登录视图上，用户只需要输入自己之前注册成功的用户名和密码，就可以完成登录的操作。

一旦用户注册或者登录成功的话，就会进入到聊天视图控制器。在聊天视图控制器中我们使用了表格视图。表格视图里面将会包含很多的单元格，我们会把用户聊天内容显示到这些单元格之中。在聊天视图中，还有一个退出按钮，当我们想退出聊天室的时候可以单击它。在聊天控制器的下方还有一个用于输入文字信息的文本框和一个发送按钮。

在故事板中从欢迎视图到注册视图，以及欢迎视图到登录视图都是通过Segue连接的。

实战：实现用户的注册功能。

步骤1：在故事板选中注册控制器视图，将Xcode切换到助手编辑器模式。

此时在右侧的代码编辑窗口中会显示RegisterViewController.swift文件的内容。其中该类中有两个IBOutlet属性，nameTextfield对应的是视图中上面的文本框，passwordTextfield对应的则是下面的。最下面的注册按钮则对应类中的registerPressed（）方法。当用户单击该按钮以后，会通过Bmob创建一个新的用户。

步骤2：在registerPressed（）方法中添加下面的代码。

```
@IBAction func registerPressed(_ sender: AnyObject) {

  //TODO: Set up a new user on our Bomb database
  let user = BmobUser()
  user.username = nameTextfield.text!
  user.password = passwordTextfield.text!
  user.signUpInBackground { (isSuccessful, error) in
    if isSuccessful {
      print("注册成功")
    }else{
      print("注册失败，错误原因：\(error!.localizedDescription)")
    }
  }
}
```

在该方法中，我们首先创建了一个BmobUser类型的对象，然后将用户设置的用户名和密码信息赋值给它的username和password属性，接下来执行BmobUser的用户注册方法，在注册过程结束以后会执行signUpInBackground（）方法中的闭包代码，它带有两个参数：isSuccessful代表注册是否成功，如果注册失败则可以通过error参数获取到错误信息。这里的signUpInBackground（）方法是后台执行方法，否则就会引起用户界面“假死”的情况。

构建并运行项目，在注册页面中输入用户名和密码，然后单击注册按钮，如果成功则会在控制台打印出相应的信息，如图11-7所示。

图11-7　注册Happy Chat新用户

步骤3：在故事板中，选中注册视图中的密码文本框，在Attributes Inspector里面，勾选Text Input Traits部分中的Secure Text Entry选项，此时该文本框在故事中会显示一个小黑点。再选中登录视图中的密码文本框，也做同样的设置。

接下来，我们希望在用户注册成功以后直接进入到聊天视图，而不是再经过登录视图的登录操作，因为这样会带来很差的用户体验。

实战：用户注册成功后直接进入到聊天控制器。

步骤1：在故事板中选中从注册视图到聊天视图的Segue，查看其Identifier为goToChat。

这里，我们希望在注册成功以后直接执行performSegue（）方法。

步骤2：在Register控制器的registerPressed（）方法中，添加如下代码：

```
if isSuccessful {
  print("注册成功")
  // 执行从注册控制器到聊天控制器的Segue
  self.performSegue(withIdentifier: "goToChat", sender: self)
}else{
```

构建并运行项目，当成功注册新用户以后，App会跳转到聊天视图，如图11-8所示。

图11-8　注册功能后跳转到聊天视图

11.3 Swift闭包

本节我们将会了解有关闭包的知识。闭包实际上是一个匿名的函数，或者说该函数不具有函数名称。它实际上是一个密闭的代码包裹，我们可以直接使用它的功能，并且将它作为参数或返回值进行传递。

到目前为止，我们在实战练习中创建了很多函数，在创建函数的时候，总是使用func关键字声明一个函数，然后定义一个函数名称，在函数名称后面的括号中包含需要的参数，参数包含参数名和参数类型。之后我们可以使用减号+箭头（->）来说明这个函数有返回值，并且在后边必须跟返回值的类型。在之后的大括号中我们需要写入该函数的执行代码，如果该函数有返回值的话，还需要return这个返回值。

```
func functionName(parameter: parameterType) -> returnType {
  // 做一些事情
  return output
}
```

其实，函数就像我们实际生活中的烤面包机，未烘烤的面包片就像是参数，我们通过烤面包机（函数）输出一个烤好的面包片（返回值）。反过来，在程序设计中我们可以把一些实现某个功能的代码打包在一起，然后给它起一个名字。在需要的时候直接通过这个名字调用该函数或者方法。

目前我们一共见过三种不同类型的函数：第一种是不带参数和返回值，只是简单地实现某个功能；第二种是带有参数或返回值的函数；第三种函数比较有意思，它以函数作为参数，输出的时候，也可能将函数作为返回值。

让我们在Playground中编写一些代码来体会这样的函数。

实战：创建calculator函数。

步骤1：创建一个calculator函数，实现简单的两个参数相加的功能。

```
func calculator(n1: Int, n2: Int) -> Int {
  return n1 + n2
}

calculator(n1: 3, n2: 6)
```

此时，如果我们想计算两个数的乘积，就需要修改calculator函数体内部的代码，将加号修改为乘号。这并不是我们想要的效果。所以，接下来我们为数学计算定义单独的函数。

步骤2：添加add（）方法。

```
func add(num1: Int, num2: Int) -> Int {
  return num1 + num2
}
```

add（）是实现两个数相加功能的，带有2个参数和1个返回值。所以这个方法的类型就为：（Int，Int）->Int，也就是函数所有的参数类型和返回值类型。

步骤3：我们希望将add（）方法作为第三个参数传递给calculator（）方法，因此需要将calculator（）方法做如下修改：

```
func calculator(n1: Int, n2: Int, operation: (Int, Int)->Int) -
> Int {
  return operation(n1, n2)
}
```

此时的calculator（）方法带有3个参数，第一个和第二个参数是两个整型数，第三个operation参数，它是一个函数类型，与之前的Int类型、String类型不同，它带有2个整型参数和1个整型返回值。

在函数中，我们需要调用operation（）函数，它具有2个整型参数，所以这里将n1和n2作为它的参数，它会返回一个整型值，也就是两个数相加的结果。

修改最下面的calculator（）函数的调用，如calculator（n1：4，n2：7，operation：add）。

在calculator（）函数中的第三个参数，我们将add（）函数作为参数带入到calculator（）函数中，在函数体内部执行add（）函数，并将n1和n2作为operation（相当于add（）的函数）的参数进行计算，并把operation（）的返回值作为calculator（）的返回值进行输出。

注意　calculator（n1：4，n2：7，operation：add）代码一定要放到两个函数定义的后面，否则会出现找不到函数的错误。

让我们在Playground中再添加一个multiply（）函数，然后将calculator（）的第三个参数修改为multiply，这样就可以看到两个数相乘的结果。

```
func multiply(num1: Int, num2: Int) -> Int {
  return num1 * num2
}

calculator(n1: 4, n2: 7, operation: multiply)
```

相信大家到这里已经了解了将函数作为参数的操作方法。接下来，我们来看看如何将函数作为另一个函数的返回值。

目前的程序代码还是比较冗长的，为了减少代码数量，我们可以使用闭包（Closure）。闭包就是匿名的函数，或者你可以把函数看成是有名字的闭包。实际上，我们只是将能够实现某一功能的代码打包在一起，然后自由地去传递它而已。下面的函数是我们非常熟悉的结构：

```
func sum(firstNumber: Int, secondNumber: Int) -> Int {
  return firstNumber + secondNumber
}
```

该函数包含func关键字、函数名称、2个参数和一个返回值类型。为了将该函数转换为闭包，我们先移除func关键字和函数名称，然后再将左大括号移到参数括号的前面，在它的位置放置一个in关键字，现在就创建好了一个闭包。请大家一定牢记这个闭包的转换方法，在之后的实战中会经常用到。

```
{ (firstNumber: Int, secondNumber: Int) -> Int in
  return firstNumber + secondNumber
}
```

我们可以将这个闭包随意传送到其他函数中，或者作为变量直接使用。

实战：使用闭包来替代之前的multiply（）函数。

步骤1：修改之前的代码如下：

```
func calculator(n1: Int, n2: Int, operation: (Int, Int)->Int) -
> Int {
  return operation(n1, n2)
}

calculator(n1: 4, n2: 7, operation: { (num1: Int, num2: Int) ->
Int in
  return num1 * num2
})
```

虽然使用了闭包的形式，但是运行结果并没有发生改变。

在Swift语言中，编译器可以基于值断言数据类型的能力。如果声明一个变量var a=2，则编译器会自动将a设置为整型。所以，我们可以继

续简化calculator（）函数的调用。使用calculator（n1：4，n2：7，operation：{（num1，num2）->Int in，编译器会自动将num1和num2设置为整型，因为在calculator（）函数中，通过return operation（n1，n2）代码，我们将n1的值赋值给num1，n2的值赋值给num2，它们都是整型，所以munX也是整型。

步骤2：通过类型断言，编译器可以判断出闭包的返回类型为Int，所以可以直接删除它。此时的代码为calculator（n1：4，n2：7，operation：{（num1，num2）in。

步骤3：删除闭包中的return关键字，因为在闭包中只有一行代码，编译器会判断你想要的返回值就是该行代码的值，此时的代码为calculator（n1：4，n2：7，operation：{（num1，num2）in num1*num2}）。

步骤4：你可能会觉得目前的代码已经是最简化的了，其实不然。在闭包中，编译器会将参数标识为0、1、2，…，其中0代表第一个参数值，以此类推。所以可以将代码修改为calculator（n1：4，n2：7，operation：{$0*$1}）。

步骤5：修改代码如下，查看运算的结果依然正确。

```
func calculator(n1: Int, n2: Int, operation: (Int, Int)->Int) -
> Int {
  return operation(n1, n2)
}

let result = calculator(n1: 4, n2: 7, operation: { (num1, num2)
-> Int in num1 * num2 })

print(result)
```

不管你是否相信，在Swift语言中还有这样一条规则，如果函数的最后一个参数是闭包，则可以先删除operation参数名称，再将闭包代码移到函数右边小括号的外面。代码如下：

```
let result = calculator(n1: 4, n2: 7){ $0 * $1 }
```

虽然通过闭包可以将程序代码简化到变态的程度，但是我建议大家还是保留基本的信息，因为这样会大大提高程序的可读性同时降低你的维护成本。当然，如果你经历了几年的开发历程，已经非常熟悉Swift语言了，你可能会真正偏好于使用超级简化的代码。

闭包的另一个非常高级的特性就是可以不通过循环语句去挨个获取集合对象中的每一个元素。

实战：通过闭包操作数组中的元素。

步骤1：在Playground中创建一个整型数组array，let array=[2，5，3，7，23，54]。我们需要让数组中的每一个整型数都加1，再生成一个新的数组。

以前我们可能会通过循环语句来实现这个功能，现在我们可以使用Map函数来实现。

步骤2：这里我们需要定义一个数组转换规则。

```
let array = [2,5,3,7,23,54]

func addOne (n1: Int) -> Int {
  return n1 + 1
}

array.map(addOne)
```

通过Array类的map函数，会遍历数组中的每一个元素，然后对每个元素都执行addOne（）函数，并且将结果放在一个新的数组里面。

步骤3：如果使用闭包的形式，则需要删除addOne的函数名称，将大括号前移，并在返回值类型后面加上in关键字。然后，可以删除闭包

中参数的类型、返回值类型以及return关键字。

```
let array = [2,5,3,7,23,54]

array.map { (n1) in n1 + 1 }
```

如果你愿意的话，还可以简化成array.map{$0+1}。

在本节的最后让我们再次梳理一下闭包的语法结构，闭包的结构如下所示：

```
{ (parameters) -> return type in
  statements
}
```

在这个没有名字的被大括号封闭的代码块中，首先是参数列表，然后是返回值类型，接下来是in关键字，在in的下面则是闭包中的执行代码。

11.4 事件驱动、应用程序生存期

11.4.1 事件驱动——应用运行的本质

在简单了解什么是闭包以后，接下来让我们回到之前的Register控制器类，然后在registerPressed（）方法里边找到创建用户的这段代码。

```
user.signUpInBackground { (isSuccessful, error) in
  if isSuccessful {
    print("注册成功")
    self.performSegue(withIdentifier: "goToChat", sender: self)
  }else{
    print("注册失败，错误原因：\(error!.localizedDescription)")
  }
}
```

在BmobUser类的signUpInBackground（）方法中，包含了一个函数类型参数。一旦用户被创建完成，就会调用该闭包中的代码。我们称这种方式叫作回调方法（callback method）。要想了解什么是回调方法，我们需要先来了解一下iOS应用程序的工作原理。

在之前，我们学习Model、View、Controller设计模式的时候，剖析过应用程序的结构。Model处理的是数据，View处理的是用户在屏幕上看到的东西，而Controller则负责处理Model与View之间的沟通。本节我们要了解的，主要是在它们背后的东西。也就是当用户单击了按钮以后，会发生什么事情?

当用户单击按钮以后，实际上按钮会发送一个消息给视图控制器。视图控制器在收到消息以后会清楚自己现在要去更新一些东西。比如，现在你的iOS应用程序里面包含很多的功能，可能有一个按钮需要用户去单击，可能有一个视频让用户去播放，也有可能是从云端数据库接收一些JSON格式的数据。

视图控制器会通过发送或接收消息的方式，处理所有的这些事件。也就是当用户单击按钮以后，视图控制器就会收到一个消息——现在按钮被用户单击了。当有电话打来的时候，iOS系统就会给当前的控制器发送一个消息，告诉它现在有一个电话打进来了。当云端数据库有数据更新的时候，控制器也会收到一个消息，此时你便可以去更新UI中的数据了。

实际上视图控制器会接收很多这样的消息。因此，我们在iOS上的编程实际上都是将事先预测到的事件，有针对性地去编写相应的代码。

在UIViewController类中定义了viewDidLoad（）方法，当控制器视图被显示到屏幕以后，操作系统就会发送viewDidLoad消息到当前的控制器，而控制器就会调用viewDidLoad（）方法来进行相关操作。

11.4.2　应用程序的生存期

就像你的移动电话、MP3播放器或笔记本电脑都具有不同大小的内存空间，以及不同级别的电池寿命一样，像iPhone这样的智能手机，续航时间很大程度上取决于你打开应用程序的数量。另外，应用程序的内存占用情况也会被操作系统监控。

如果你现在从一个笔记类软件切换到一个3D射击类游戏，当前的游戏就会使用大量的内存空间。因此，你的应用会有一个生存周期。

首先，应用程序在启动以后它会在屏幕上面可见，当你单击了Home键或者是接听电话时，应用程序就会进入后台。实际上，当你的应用程序在屏幕上消失，它就会进入后台。直到操作系统将它“杀死”之前，应用程序还是会占用一些内存空间，如图11-9所示。

图11-9　应用程序完整的生存期示意图

11.4.3 什么是完成处理?

本节我们来说说完成处理（completion handler）的相关操作。让我们在Playground中模拟一下用户注册的流程。

步骤1：创建BmobUser和MyApp两个类，仿照用户注册的流程，我们为这两个类添加相关的方法与属性。

```
class BmobUser {
  var username: String?  // 用户名称
  var password: String?  // 密码

  func signUpInBackground() {
    // 进行用户注册的相关操作

  }
}

class MyApp {
  // 模拟用户单击注册按钮
  func registerButtonPressed() {
    let user = BmobUser()

    user.username = "lele"
    user.password = "123456"

    user.signUpInBackground()
  }
}
```

如果在项目中我们这样来执行用户注册的话，应用的用户界面会出现短时的假死，所以Bmob SDK会在后台运行signUpInBackground（）方法，在注册成功以后，我们需要进入聊天控制器视图，这就需要一种方法，一旦后台注册完成就可以通知到主线程的当前控制器类。也就是说在BmobUser类的signUpInBackground（）方法中，当完成注册过程以后可以给registerButtonPressed（）发送一个消息，告知新

用户是否注册成功，或者发送注册失败的原因。这就需要我们进行完成处理。

步骤2：在MyApp类中创建一个completed（）方法。

```
func completed(isSuccessful: Bool, error: Error?) {
}
```

该方法带有2个参数，第一个参数代表新用户注册是否成功，第二个参数是当注册发生错误的时候，包含错误信息。

步骤3：修改signUpInBackground（）方法的定义如下：

```
func signUpInBackground(completed: (Bool, Error? )->Void) {
  // 进行用户注册的相关操作
}
```

此时的signUpInBackground（）方法带有1个参数，该参数是函数类型，符合MyApp的completed（）方法的类型。没错，我们就是要将completed（）方法作为参数传递到signUpInBackground（）方法。

步骤4：修改signUpInBackground（）方法，模拟新用户注册成功。

```
func signUpInBackground(completion: (Bool, Error?)->Void) {
  // 进行用户注册的相关操作

  let isSuccessful = true

  completion(isSuccessful, nil)
}
```

通过代码可知，我们让新用户注册成功，并在方法的最后会调用completion所指向的函数，并传递true和nil两个参数值。

步骤5：修改MyApp类的代码。

```
class MyApp {

  func registerButtonPressed() {
    let user = BmobUser()

    user.username = "lele"
    user.password = "123456"

    user.signUpInBackground(completion: completed)
  }

  func completed(isSuccessful: Bool, error: Error?) {
    print("新用户注册：\(isSuccessful)")
  }
}
```

在调用signUpInBackground（）方法的时候，将completed（）方法作为参数。这样在signUpInBackground（）方法中，用户注册完成以后就会调用这个方法，也就相当于MyApp类的completed（）方法会被调用。

步骤6：在两个类的外面，添加如下代码。

```
let myApp = MyApp()
myApp.registerButtonPressed()
```

这里我们激活registerButtonPressed（）方法，创建BmobUser类型的对象，在设置好其username和password属性后，执行signUpInBackground（）方法，并将MyApp类的completed（）方法作为参数。

在signUpInBackground（）方法中，一旦我们创建用户的过程完成，就会调用completion参数所指向的方法并带有两个参数。

在completed（）方法中，我们只是简单打印出新用户是否注册成功。

这里我们为大家演示了完成处理的整个流程，在Happy Chat项目中，我们只会看到例子中的MyApp部分，而且在Happy Chat中我们使用的是闭包形式。

步骤7：用之前介绍的方法，将MyApp类中的completed（）方法转换为闭包形式。

```
user.signUpInBackground(){(isSuccessful, error) in
  print("新用户注册：\(isSuccessful)")
}
```

11.5　导航控制器是如何工作的?

在完成用户注册功能以后，接下来我们需要让用户可以登录到应用之中。

在目前的代码中，当我们注册成功以后会自动切换到ChatViewController控制器。为了可以返回到登录界面，需要先实现退出功能。

实战：在Chat控制器中实现用户的退出功能。

步骤1：在故事板中选中ChatViewController视图，将Xcode切换到助手编辑器模式，在IBAction方法logOutPressed（）中，添加如下代码。

```
@IBAction func logOutPressed(_ sender: AnyObject) {

  //TODO: 退出并回到 WelcomeViewController
  BmobUser.logout()
}
```

这里通过BmobUser类的logout（）方法注销登录账号并删除本地账号。

让我们回到故事板，仔细观察其中的几个控制器视图，应用程序是从导航控制器（Navigation Controller）开始的，首先呈现的是欢迎控制器，如果用户单击注册按钮的话，导航控制器会推出注册控制器视图，注册成功以后导航控制器会进一步推出聊天控制器视图。如图11-10所示。

图11-10　Happy Chat应用的故事板

接下来，当用户单击聊天控制器中右上角的退出按钮，要回到之前的欢迎控制器视图。一切都是从导航控制器开始，其他的控制器在导航控制器中都是嵌套呈现的，也就相当于导航控制器在管理着其他一切控制器的视图。

凡是加入到导航控制器的控制器视图，在顶部都会呈现一个导航栏，这样我们就可以确定某个控制器是否加入到了导航控制器之中。还记得之前我们在创建Segue的时候在快捷菜单中选择show模式吗？如果是在导航控制器中，加入一个新的控制器就需要选择push。

步骤2：在logOutPressed（）方法中添加下面的代码，在用户成功退出以后回到欢迎控制器。

```
@IBAction func logOutPressed(_ sender: AnyObject) {

  //TODO: 退出并回到 WelcomeViewController
  BmobUser.logout()
  navigationController?.popToRootViewController(animated: true)
}
```

构建并运行项目，在注册成功并进入聊天控制器，再单击退出按钮以后，导航控制器会退回到位于根部的WelcomeViewController。

11.6　编写登录屏幕代码

在成功创建了用户注册功能以后，接下来我们要实现用户的登录功能。

实战：实现用户登录的功能。

步骤1：在项目导航中打开LoginViewController.swift文件。在logInPressed（）方法中添加下面的代码：

```
@IBAction func logInPressed(_ sender: AnyObject) {
  BmobUser.loginWithUsername(inBackground: nameTextfield.text!,
password: passwordTextfield.text!) { (user, error) in

  }
}
```

通过BmobUser类的loginWithUsername（）方法实现用户的登录功能，它带有3个参数，分别是username、password和闭包。也就是在后台线程中完成登录过程后，会执行闭包中的代码。该闭包带2个参数，BmobUser类型的用户对象信息和如果发生错误的错误信息。

步骤2：在闭包中添加下面的代码。

```
BmobUser.loginWithUsername(inBackground: nameTextfield.text!,
password: passwordTextfield.text!) { (user, error) in
  if error != nil {
    print(error!.localizedDescription)
  }else {
    print("登录成功")
    self.performSegue(withIdentifier: "goToChat", sender: self)
  }
}
```

如果error不为nil则代表登录出现了问题，这里会打印错误信息，否则打印登录成功到Xcode控制台，并通过Segue推出到聊天控制器。

构建并运行项目，在登录成功以后会推出到ChatViewController视图。如图11-11所示。

图11-11　用户登录成功进入到聊天控制器

11.7 表格视图

通过前面几节的实战练习，我们已经为Happy Chat项目实现了用户注册、登录以及退出的功能。本节我们将会开始逐步实现最后一个视图控制器，也就是聊天控制器的功能。这是我们Happy Chat应用中最重要的一个控制器。在这个控制器中，我们要使用一个非常重要的组件——表格视图（Table View）。

实际上，表格视图就是一个增强版的视图，它能够显示一定数量的单元格，并且可以进行纵向滚动。我们称它为表格视图，其实有点儿用词不当，因为一个真正的表格，它可以包含行和列。但是在iOS中的表格视图，它只能有一列。

在很多应用程序中，我们都会看到表格视图，比如说电子邮件、iMessage，甚至是iPhone的设置等。在表格视图中，我们将几个单元格组成一组，表格视图就会被分成几个部分。每一个部分（Section）都会完成相同的功能。在我们的Happy Chat应用中，将会使用表格视图显示所有人发送的文本消息。为了实现这一功能，我们需要将一个表格视图内嵌到ChatViewController的视图之中。

请仔细观察故事板中的聊天控制器视图，它已经被分割成了几部分。在视图的顶部是状态栏，这里会显示手机的信号强弱、电量、时间等。接下来是导航栏，代表当前的ChatViewController已经被添加到了导航控制器的堆栈之中。在其下方就是本节的重点——表格视图。它占据了我们控制器视图绝大部分的空间。但并不是所有，因为在视图的底部还有一个用于输入信息的文本框和一个发送按钮。

实际上，在聊天控制器视图中的表格视图已经添加了相应的自动布局约束。如果你想创建带有表格的控制器，最简单的方式是直接从对象库中拖曳一个表格视图控制器（Table View Controller）到故事板中，该视图的全部空间会被表格视图占据。如果在控制器视图中只有一部分是用于呈现表格视图的话，你可以从对象库中拖曳一个表格视图（Table View）到控制器视图里边。因此，表格视图控制器和表格视

图是两个不同的对象，前者是包含表格视图的控制器，后者则是单纯的表格视图。

在ChatViewController类中，我们采用的是后者，该类中包含一个IBOutlet属性messageTableView，它与视图中的表格视图关联。

实战：让ChatViewController类符合两个协议。

步骤1：在ChatViewController类的声明部分，添加两个表格操作相关的协议。

```
class ChatViewController: UIViewController,
UITableViewDelegate, UITableViewDataSource {
```

它代表当前的ChatViewController类是表格视图的委托对象，当在表格视图中发生特定事件的时候，比如用户单击某个单元格，在某个单元格上面滑动等，ChatViewController就会得到通知，并可以去处理这些事件。

第二个协议UITableViewDataSource代表当前的ChatViewController要负责提供在表格视图中显示的数据。

步骤2：在viewDidLoad（）方法中添加下面的代码。

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

  //TODO: Set yourself as the delegate and datasource here:
  messageTableView.delegate = self
  messageTableView.dataSource = self
  ……
```

此时Xcode编译器会报错：ChatViewController不符合UITableViewDataSource协议。（Type'ChatViewController'does not

conform to protocol'UITableViewDataSource'）这是因为UITableViewDataSource协议有2个required方法我们还没有实现。

步骤3：在TODO：Declare cellForRowAtIndexPath here：注释行的下面添加一个方法。

```
//TODO: Declare cellForRowAtIndexPath here:
func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {

}
```

当表格视图中要显示一些东西的时候会调用该方法。在故事板中的表格视图里面，我们可以看到一个Prototype Cell，如图11-12所示。

表格视图虽然可以显示成百上千的单元格数据，但是它却非常高效。实际上，它的工作就是载入那些只在当前会呈现到屏幕上的单元格。如图11-13所示，在下面的消息列表视图中，一共有7个单元格。

当我们决定要向上滚动视图并查看下一个单元格的内容时，当前表格视图中的所有单元格都会上移。此时，顶部的单元格会从屏幕顶端移出，出于执行效率的考虑，系统会为那个被移出的单元格填充需要的数据，然后接到底部第7个单元格的下面。这样，它就变成了第8个单元格。所以，不管用户向上或向下滚动出多少个单元格内容，实际上该表格视图只使用了不超过10个单元格对象。每个被滑出的单元格对象都通过复用的方式被重新填充了新的数据。

图11-12　聊天控制器视图中的Prototype Cell

图11-13　聊天应用中的消息界面

tableView：cellForRowAt：方法就是要提供将会显示在表格视图上的单元格对象。但是我们不会使用Xcode提供的默认单元格对象，因为默认的格式只是白色背景、黑色文本和灰色分割线，这是很低级的设计。我们希望用户看到的信息更加优雅，为了这点我们将会创建自定义的单元格。

步骤4：在项目导航的View/Custom Cell里面，你可以看到项目提供的设计文件MessageCell.xib。文件中有一个单元格，它里面包含一个Image View用于显示个人头像。在右侧的视图中包含上下两个Label，上面的用于显示用户名，下面的用于显示信息。

在Attributes Inspector中可以看到当前MessageCell的Identifier属性被设置为customMessageCell，如图11-14所示。这个设置非常关键，否则我们无法在Chat-ViewController类中引用这个单元格对象。

图11-14　单元格的Identifier属性

在Identifier Inspector中可以看到当前的MessageCell与CustomMessageCell类相关联，如图11-15所示。

图11-15　单元格的Class属性

在项目导航中还有一个Message.swift文件，该类管理着显示在单元格中的信息。它里面需要包含三个属性，分别对应MessageCell的头像、用户名和文字信息。

步骤5：在ChatViewController类的tableView：cellForRowAt：方法中添加下面的代码。

```
func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
```

```
  let cell = tableView.dequeueReusableCell(withIdentifier:
"customMessageCell", for: indexPath) as! CustomMessageCell
}
```

这里我们使用tableView的dequeueReusableCell（）方法创建可以复用的单元格对象，将identifier参数设置为在故事板中查看到的MessageCell的identifier属性，也就是CustomMessageCell。indexPath就是从tableView：cellForRowAt：传递过来的参数，它代表我们要创建的单元格所呈现到表格视图中的位置。它包含两个属性：row（行）和section（部分）。目前我们创建的单元格都是在一个section之中，并且默认的section为0，所以只关注row即可。

因为在应用中我们使用了一个自定义的单元格，所以需要在最后添加一个强制转换。这样编译器就会知道单元格对象是CustomMessageCell类型，我们可以直接访问其内部的三个IBOutlet属性，例如avatarImageView、messageBody和senderUsername。

当我们载入表格视图的时候，TableView就会通过delegate调用该方法所要单元格对象。ChatViewController就会拆分集合中的数据，并将它们分别显示到单元格里面。至于应该创建的是表格视图中的第几个单元格，我们会通过cellForRowAt参数得到顺序。基本上我们的单元格顺序与indexPath的row属性值一致。

创建单元格的代码类也非常简单，只需要创建一个新的Cocoa Touch Class文件，将Subclass of设置为UITableViewCell，并且勾选Also create XIB file，然后定义Class的名称即可，如图11-16所示。

在项目导航中，我们可以看到新添加了两个文件——NewTableViewCell.swift和NewTableViewCell.xib。其中NewTableViewCell.xib是单元格的设计文件。我们可以将这两个文件通过IBOutlet连接到代码文件中并创建单元格的拷贝，如图11-17所示。

图11-16　创建新的Cocoa Touch Class文件

图11-17　新创建的NewTableViewCell类

因为在Happy Chat项目中已经创建好了CustomMessageCell类，所以我们先将刚刚创建的NewTableViewCell类的两个文件删除。

步骤6：为了可以在ChatViewController中使用CustomMessageCell类，需要先在viewDidLoad（）方法中进行注册。

```
//TODO: Register your MessageCell.xib file here:
messageTableView.register(UINib(nibName: "MessageCell", bundle:
nil), forCellReuseIdentifier: "customMessageCell")
```

该方法会注册一个Nib对象，Nib文件就是早期开发中的Xib文件，它们都同属于设计文件。这里的Nib文件包含供表格视图使用的单元格设计文件，但是该文件并没有设置它的标识。register（）方法的第一个参数是UINib类型的对象，其中nibName是项目中单元格的xib文件，而forCellReuseIdentifier参数是xib文件中单元格对象的identifier属性值。

步骤7：在tableView：cellForRowAt：方法中添加一个临时数组。

```
//TODO: Declare cellForRowAtIndexPath here:
func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(withIdentifier:
"customMessageCell", for: indexPath) as! CustomMessageCell

  let messageArray = ["第一条信息" "第二条信息" "第三条信息"]
  cell.messageBody.text = messageArray[indexPath.row]
  return cell
}
```

messageArray数组只是在测试阶段显示临时信息的数组，之后我们会通过indexPath.row得到与当前单元格匹配的文字信息，并将其赋值给cell.messageBody的text属性。还记得cell是CustomMessageCell类型，该类型中包含几个IBOutlet属性，其中一个就是messageBody，它与MessageCell.xib中单元格里面的Label关联。

如果应用程序运行到聊天控制器，当呈现表格视图的时候，在第一次调用tableView：cellForRowAt：方法时，indexPath.row的值为0，因此"第一条信息"会被赋值到第一个单元格的messageBody属性，进而显示在Label上。

我们调用的tableView：cellForRowAt：方法需要返回一个UITableViewCell类型的对象，所以在该方法中，我们首先设置了预先设计好的CustomMessageCell类，然后设置了单元格的messageBody属性，最后将这个单元格对象返回到表格视图里面。

接下来，我们还需要指定表格视图显示多少个单元格。

步骤8：在TODO：Declare numberOfRowsInSection here：注释的下面添加一个方法。

```
//TODO: Declare numberOfRowsInSection here:
func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
  return 3
}
```

该方法的返回值为Int类型，也就是要告诉表格视图每个section中有多少个单元格。其中section会通过numberOfRowsInSection参数告诉我们，我们会根据这个参数来设置每个section的单元格数。

目前，我们会在表格视图中显示messageArray数组中的内容，所以将返回值设置为3。

如果我们想在表格视图中创建多个section，那还需要实现UITableViewDataSource协议中的可选方法numberOfSections（），来指定表格视图中有多少个section。例如：

```
func numberOfSections(in tableView: UITableView) -> Int {
  return 1
}
```

构建并运行项目，在登录成功以后，你可以在表格视图中看到3个单元格。但是目前的显示出现了一点儿Bug，如图11-18所示。这是因为表格视图中的单元格默认高度均为44个点，但是在聊天应用中，我们可能会输入很长的信息，因此不能将单元格的高度限制为44个点。

图11-18　在聊天控制器中呈现的三个单元格

为了解决这个问题，我们已经在CustomMessageCell中设置了约束，为了激活这些约束，也就相当于告诉表格视图所创建的单元格要基于其呈现内容的多少。如果message-Body中的内容很多，则要调整单元格的尺寸。我们需要创建一个新的方法。

实战：根据内容动态调整单元格的高度。

步骤1：在ChatViewController类的TODO：Declare configureTableView here：注释的下面，添加一个新的方法。

```
func configureTableView() {
  messageTableView.rowHeight = UITableViewAutomaticDimension
  messageTableView.estimatedRowHeight = 120.0
}
```

UITableViewAutomaticDimension用于指定表格视图使用一个给定的尺寸作为单元格高度的默认值。estimatedRowHeight属性指定一个预估

的高度，这里设置为120.0个点。这个高度值差不多是大部分信息足够的高度值。通过UITableViewAutomaticDimension的设置，如果你预估的高度不正确，表格视图会利用约束自动调整单元格尺寸。

步骤2：在viewDidLoad（）方法的最后，添加对configure-TableView（）的调用。

为了验证是否解决了Bug，我们可以将messageArray中的字符串元素写得更长些，构建并运行项目，如图11-19所示。

图11-19　按照内容动态设置单元格高度

在本节的前面，我们临时创建了一个messageArray数组来显示聊天信息。但实际上，我们需要通过Model来呈现真实的聊天信息。

在项目导航选中Model文件夹中的Message.swift文件，当前该类中并没有任何的属性和方法。我们需要在这里指定聊天信息的所有属性。

在Message类中添加两个属性。

```
class Message {

  //TODO: Messages need a messageBody and a sender variable
  var sender: String = ""
  var messageBody: String = ""
}
```

现在我们已经完成了数据模型类的创建，接下来将使用它来存储和呈现聊天信息。

11.8　了解UI动画

作为一名iOS程序员，我们多少要有一些编写代码动画的技能。这次我们将会学习关于渐变动画的相关知识。渐变动画涉及一个图像或视图的开始和结束的位置，通过设置一个过渡时间来生成这个动画。计算机会自动绘制开始和结束之间的所有的动画效果。

在Happy Chat项目中，我们同样需要让视图实现动画效果。在聊天控制器中，当用户单击视图底部的文本框输入文字信息的时候，会弹出虚拟键盘。这就需要控制器在弹出虚拟键盘的时候腾出一部分空间。这也就是为什么在ChatViewController类中会定义一个NSLayoutConstraint类型的heightConstraintIBOutlet属性。

heightConstraint关联的既不是视图也不是按钮，而是一个约束，是针对屏幕底部视图的高度约束。如图11-20所示。当用户在文本框中输入信息的时候，需要通过代码将高度约束值变得大些，这会让文本框向上提升一个高度，以至于不会遮挡从下方滑出的虚拟键盘。

图11-20　在故事板中的高度约束

整个过程的第一步需要检测用户是否将焦点定位到文本框里面。

实战：动态改变约束值。

步骤1：为ChatViewController类添加UITextFieldDelegate协议。

```
class ChatViewController: UIViewController,
UITableViewDelegate, UITableViewDataSource, UITextFieldDelegate
{
```

通过该协议，我们可以在控制器类中接收到用户与文本框互动的消息。

步骤2：在viewDidLoad（）方法//Set yourself as the delegate of the text field here：注释语句的下面，设置messageTextfield的delegate属性。

```
//TODO: Set yourself as the delegate of the text field here:
messageTextfield.delegate = self
```

如果你仔细回忆的话，可能会记得在之前添加其他协议的时候，编译器总是会报错让我们创建必须实现的委托方法。此时编译器并没有报错，这是因为UITextFieldDelegate协议中都是可选方法。即我们可以不必在委托对象的类中定义这些方法，如果定义了这些方法，会在相关事件发生的时候调用它；而如果没有定义，则会在事件发生时跳过，此时控制器不会有任何操作。

步骤3：在TODO：Declare textFieldDidBeginEditing here：注释代码的下面创建委托方法。

```
//TODO: Declare textFieldDidBeginEditing here:
func textFieldDidBeginEditing(_ textField: UITextField) {
  heightConstraint.constant = 308
  view.layoutIfNeeded()
}
```

当用户开始在文本框中编辑文字的时候会激活该方法，至于是哪个文本框，则是通过之前在viewDidLoad（）方法中的messageTextfield.delegate=self代码决定。

一般来说，虚拟键盘的高度是290个点，所以不想键盘被遮挡的话，需要将高度约束从50调整为340以上，这里我们将高度约束的constan属性修改为348。在修改完约束值以后，我们还需要调用UIView的

layoutIfNeed（）方法让所有屏幕上的视图重新绘制，包括更新后的约束。

构建并运行项目，在单击进入文本框以后，底部视图的高度立即发生了变化。如果此时在模拟器中使用快捷键Command+K则会从屏幕底部滑出虚拟键盘，如图11-21所示。

图11-21　在模拟器中弹出虚拟键盘后的效果

在真实的设备上，每次我们单击文本框都会弹出虚拟键盘，但是现在视图高度的变化效果非常糟糕，我们希望以动画的方式来呈现视图高度的变化。

实战：以动画的方式来呈现虚拟键盘。

步骤1：修改textFieldDidBeginEditing（）方法中的代码。

```
func textFieldDidBeginEditing(_ textField: UITextField) {
  UIView.animate(withDuration: 0.5, animations: {
    self.heightConstraint.constant = 348
    self.view.layoutIfNeeded()
  })
}
```

其中，withDuration参数代表动画的时长。animations是闭包，代表我们要实现的动画，这里我们需要更新约束值，因为是在闭包之中，所以还要加上self.。

构建并运行项目，可以看到视图高度变化的动画效果，比之前的效果平滑了很多。

步骤2：在TODO：Declare textFieldDidEndEditing here：注释代码的下面创建委托方法。

```
//TODO: Declare textFieldDidEndEditing here:
func textFieldDidEndEditing(_ textField: UITextField) {
  UIView.animate(withDuration: 0.5) {
    self.heightConstraint.constant = 50
    self.view.layoutIfNeeded()
  }
}
```

这里通过动画的方式，将视图的高度调整回50。

如果此时构建并运行项目的话，虽然可以顺利滑出虚拟键盘，但是目前还没有某个事件可以结束文本框的编辑状态。接下来，我们需要实现的效果是当用户单击文本框以外的任何区域，比如表格视图，就可以结束文本框的编辑状态，从而让虚拟键盘消失，底部视图回到之前的高度约束值。

为了实现这个效果，我们需要创建一个自定义手势识别，当用户用手单击表格视图以后激活文本框编辑结束事件。

步骤3：在viewDidLoad（）方法的TODO：Set the tapGesture here：注释语句的下面，添加这样的代码。

```
//TODO: Set the tapGesture here:
let tapGesture = UITapGestureRecognizer(target: self, action:
#selector(tableViewTapped))
messageTableView.addGestureRecognizer(tapGesture)
```

首先，通过UITapGestureRecognizer类初始化一个单击手势识别对象，当有符合该手势的交互操作时，就会调用self.tableViewTapped（）方法。第二行是将该手势识别添加到表格视图上，也就意味着只有在表格视图中才会识别该手势并执行相应的方法。

此时编译器会报错，因为我们还没有创建tableViewTapped（）方法。

步骤4：在TODO：Declare tableViewTapped here：注释语句的下面，创建一个新的方法。

```
//TODO: Declare tableViewTapped here:
func tableViewTapped() {
  messageTextfield.endEditing(true)
}
```

当用户单击表格视图以后，就会结束messageTextfield的编辑状态。

此时编译器又会报另外一个错误：Argument of'#selector'refers to instance method'tableViewTapped（）'that is not exposed to Objective-C。如果发生这种情况，只需单击Fix按钮，Xcode会为我们修复这个问题。编译器会将tableViewTapped（）方法明确标记为@objc，如同@objc func tableViewTapped（）。

在这里我们接触到了一个新的关键字——selector，selector并不复杂，但它是很久之前iOS开发所使用的Objective-C语言遗留下来的。现在，Objective-C语言还有很多非常好的东西，不过也有一些不适合现代开发语言的东西。在Swift语言中使用的selector就来自于Objective-C，它会执行类中的方法，但是在应用程序运行之前，编译器并不知道这个方法。只是在运行过程中，我们才会使用selector去决定执行哪个方法。

构建并运行项目，在滑出虚拟键盘的情况下，单击表格视图，虚拟键盘消失，高度约束变为50，如图11-22所示。

图11-22　虚拟键盘出现和消失后的界面效果

11.9 发送消息

在本节中，我们将实现发送消息到Bmob云端数据库的功能。为了实现这个功能，我们必须在IBAction方法sendPressed（）中添加一些代码。当用户在单击发送按钮以后，应用就会将文本框中的内容发送到云端数据库。

首先我们要做的就是在用户单击发送按钮以后，强制让文本框处于结束编辑状态，这样虚拟键盘便会消失。

步骤1：在ChatViewController类的sendPressed（）方法中添加一行代码。

```
@IBAction func sendPressed(_ sender: AnyObject) {
  messageTextfield.endEditing(true)

  //TODO: Send the message to Bmob and save it in our database
  messageTextfield.isEnabled = false
  sendButton.isEnabled = false

}
```

在用户单击发送按钮以后，我们暂时让文本框和按钮处于禁止状态，因为我们不想在数据传输到云端的过程中，允许用户再次提交新的聊天信息。

步骤2：继续在sendPressed（）方法中添加代码。

```
@IBAction func sendPressed(_ sender: AnyObject) {
  messageTextfield.endEditing(true)

  //TODO: Send the message to Bmob and save it in our database
  messageTextfield.isEnabled = false
  sendButton.isEnabled = false

  let user = BmobUser.current()
```

```
  let chatMessage = BmobObject(className: "Message")
  chatMessage?.setObject(user?.username, forKey: "Sender")
  chatMessage?.setObject(messageTextfield.text, forKey:
"MessageBody")

}
```

这里通过BmobUser的current（）方法获取到当前登录的用户信息，然后使用BmobObject类实例化云端的Message数据表，如果云端不存在该表的话则会直接创建。

接下来，我们在表中添加了一行记录，Sender字段为发送消息的人名，MessageBody字段则是消息内容。

步骤3：继续在sendPressed（）方法的最后添加下面的代码。

```
@IBAction func sendPressed(_ sender: AnyObject) {
  messageTextfield.endEditing(true)
  ……
  chatMessage?.saveInBackground{ (isSuccessed, error) in
    if error != nil {
      //发生错误
      print("error is \(error!.localizedDescription)")
    }else{
      print("聊天信息存储云端成功")
      self.messageTextfield.isEnabled = true
      self.sendButton.isEnabled = true
      self.messageTextfield.text = ""
    }
  }
}
```

通过BmobObject类的saveInBackground（）方法，我们将聊天记录添加到云端的Message数据表中。如果存储成功，我们还需要恢复之前禁用的两个控件的功能，并清空文本框中已经被提交的信息。

构建并运行项目，在文本框中输入一些文字，然后单击发送按钮。如果一切正常，可以在控制台中看到聊天信息存储云端成功的日志信息。

为了验证代码是否正确，我们可以登录Bmob.cn，在控制台中查看Message数据表中是否包含所提交的信息，如图11-23所示。

图11-23　在Bmob网站的控制台中查看提交的消息

11.10　通过Bmob监听数据表的变化

在成功将聊天信息添加到云端数据表以后，接下来让我们先把之前用于测试的垃圾信息清除掉。

实战：整理之前用于测试的垃圾信息。

步骤1：在ChatViewController类中创建一个新的属性。

```
// Declare instance variables here
var messageArray: [Message] = [Message]()
```

这里创建了一个Message类型的数组，使用[类名称]（）的方式初始化了一个新的没有包含任何元素的数组对象。

步骤2：在TODO：Create the listen method here：注释语句的下面，创建一个方法用于监听云端数据表的变化。

```
//TODO: Setup the listen method here:
func listen() {
  let event = BmobEvent.default()
  if let event = event {
    event.delegate = self
    event.start()
  }
}
```

该方法会创建一个默认的BmobEvent对象，我们通过它来监听云端数据表的变化。如果event创建成功，则继续设置event的delegate属性，并让其开始监听。

技巧　这里使用了if let event=event代码来将event可选绑定。其中后面的event是通过其上一行代码声明的常量，而前者的event则是在当

前if语句中生成的新常量。后者的event常量的生存期仅限于if语句的执行体内部。整个if语句的意思是：如果上一行的常量event为nil，则跳过整个if语句。如果不为nil，则会将值赋给if语句内部的event常量，在if语句体内部调用的都是这个event常量。

此时，编译器会报错：Cannot assign value of type'ChatViewController'to type'BmobEventDelegate！'，这是因为虽然将当前的ChatViewController设置为BmobEvent的委托对象，但是在类声明的时候，并没有让ChatViewController符合BmobEventDelegate协议。

步骤3：修改ChatViewController类的声明。class ChatViewController：UIViewController，UITableViewDelegate，UITableViewDataSource，UITextFieldDelegate，BmobEventDelegate{。

接下来，我们需要实现BmobEventDelegate协议中的5个委托方法。

步骤4：在MARK：-BmobEvent Delegate Methods注释语句的下面，添加这5个委托方法。

```
// 告知控制器监听功能已经连接上云端Bmob服务器
func bmobEventDidConnect(_ event: BmobEvent!) {
  print(event.description)
}

// 告知控制器监听可以订阅或者取消订阅
func bmobEventCanStartListen(_ event: BmobEvent!) {
  event.listenTableChange(BmobActionTypeUpdateTable, tableName:
"Message")
}

// 在订阅监听时，接收信息
func bmobEvent(_ event: BmobEvent!, didReceiveMessage message:
String!) {
  print("didReceiveMessage \(message)")
}
```

```
// BmobEvent发生错误时
func bmobEvent(_ event: BmobEvent!, error: Error!) {
  print(error.localizedDescription)
}

// 告知控制器监听功能连接不了云端Bmob服务器
func bmobEventDidDisConnect(_ event: BmobEvent!, error: Error!)
{
}
```

当event.start（）被执行以后，event就会调用delegate（也就是ChatViewController类）对象的bmobEventDidConnect（）方法确认监听功能是否连接上云端的Bmob服务器。如果失败则会调用bmobEvent：error：方法传递错误信息。

在成功连接以后，BmobEvent会调用bmobEventCanStartListen（）方法询问监听哪个数据表的哪个动作，涉及的动作有：表更新（BmobActionTypeUpdateTable）、行更新（Bmob-ActionTypeUpdateRow）、表删除（BmobActionTypeDeleteTable）和行删除（Bmob-ActionTypeDeleteRow）。

构建并运行项目，此时控制台会打印错误信息：The operation couldn’t be completed.（SocketIOError error-6.），这还是由于苹果的网络安全策略导致的。仿照之前Weather项目的操作，在Info.plist文件中添加两个配置信息。

```
<key>NSAppTransportSecurity</key>
  <dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
  </dict>
```

再次构建并运行项目，可以发现运行一切正常。虽然目前还没有从bmobEvent：didReceiveMessage：方法中得到任何消息，但是一旦我们在文本框中输入一些聊天信息，然后单击发送按钮，控制台就会显示出BmobEvent反馈的信息。

```
didReceiveMessage Optional("
{\"appKey\":\"088e9ca802449a3117c23e1066585629\",\"tableName\":
\"Message\",\"objectId\":\"\",\"action\":\"updateTable\",\"data
\":{\"MessageBody\":\"乐乐你好
□\",\"Sender\":\"lele\",\"createdAt\":\"2018-02-19
12:36:05\",\"objectId\":\"cee3fb59ae\",\"updatedAt\":\"2018-02-
19 12:36:05\"}}")
```

实战：从Bmob云端获取消息信息。

步骤1：我们要利用之前的SwiftyJSON解析这些JSON格式的数据。修改Podfile文件，添加Pod'SwiftyJSON'的第三方类库，然后在终端中执行pod update命令。

步骤2：在安装好SwiftyJSON以后，需要在Xcode中按Command+B组合键重新编译项目，然后在ChatViewController中导入SwiftyJSON。

步骤3：修改bmobEvent：didReceiveMessage：方法。

```
func bmobEvent(_ event: BmobEvent!, didReceiveMessage message:
String!) {
  let messageJSON: JSON = JSON(message!)
  print(messageJSON)
}
```

这里将从云端传递过来的JSON格式数据转换为可以直接使用的对象。在jsoneditoronline.org网站中可以解析出相应的数据结构。这里我们需要的是data中的MessageBody和Sender两个信息，如图11-24所示。

图11-24　分析从Bmob云端数据库中传回的数据

步骤4：继续修改bmobEvent:didReceiveMessage：方法。

```
func bmobEvent(_ event: BmobEvent!, didReceiveMessage message:
String!) {
  let messageJSON: JSON = JSON(parseJSON: message)
  print(messageJSON)

  let text = messageJSON["data"]["MessageBody"]
  let sender = messageJSON["data"]["Sender"]

  let message: Message = Message()
  message.messageBody = text.stringValue
  message.sender = sender.stringValue

  self.messageArray.append(message)
  self.configureTableView()
  self.messageTableView.reloadData()
}
```

这里通过SwiftyJSON类将获取到的MessageBody和Sender信息赋值给Message类型的对象。然后，我们把新生成的Message对象添加到messageArray数组之中。在新加入聊天信息以后我们需要重置表格视图的高度，所以要执行configureTableView（）方法，最后再通过表格视图的reloadData（）方法重新刷新表格视图。

步骤5：在tableView：cellForRowAt：方法中，删除之前测试用的数组，然后修改相关代码。

```
func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(withIdentifier:
"customMessageCell", for: indexPath) as! CustomMessageCell

  cell.messageBody.text =
messageArray[indexPath.row].messageBody
  cell.senderUsername.text = messageArray[indexPath.row].sender
  cell.avatarImageView.image = UIImage(named: "egg")

  return cell
}
```

步骤6：修改tableView：numberOfRowsInSection：方法中的代码，返回messageArray数组的元素个数。

```
func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
  return messageArray.count
}
```

构建并运行项目，可以看到聊天信息被添加到表格视图之中，如图11-25所示。

图11-25　聊天信息被添加到云端数据库中

或者你还可以打开两个模拟器，然后用不同的账号登录，互相发送消息，效果如图11-26所示。

图11-26　从Bmob云端监听数据到本地

11.11　进一步完善用户体验和用户界面

虽然我们的Happy Chat的项目制作已经接近了尾声，但是在代码中还有很多print语句，print语句是将一些关键信息打印到控制台，便于程序员随时查看当前的运行状态。但是对于用户来说，我们需要通过一种机制让用户知道这个时候我们正在与云端服务器进行数据交换或进行验证。

11.11.1 利用Progress Spinner改善用户体验

在用户登录的时候，我们可以在进行网络数据传输与验证的时候，显示一个进度指示器，这样当用户单击按钮的时候，会通过指示器告诉他们当前正在进行着一项网络操作，从而改善用户体验。

还记得之前我们通过CocoaPods方式安装的SVProgressHUD吗？接下来，我们就使用它来制作这个效果。

步骤1：在LoginViewController类中导入该类库，import SVProgressHUD。

步骤2：一旦用户单击登录按钮，就激活了该方法。

```
@IBAction func logInPressed(_ sender: AnyObject) {
  SVProgressHUD.show()
  ……
```

SVProgressHUD类的show（）方法会显示一个旋转的圆圈，告诉用户该应用当前正在处理一些事情，请耐心等待。

步骤3：在用户认证成功以后，执行Segue跳转之前，我们让其消失。

```
if error != nil {
  print(error!.localizedDescription)
}else {
  print("登录成功")
  SVProgressHUD.dismiss()
  self.performSegue(withIdentifier: "goToChat", sender: self)
}
```

步骤4：在RegisterViewController中的用户注册部分添加同样的效果。

```
@IBAction func registerPressed(_ sender: AnyObject) {

  SVProgressHUD.show()
  ……
  user.signUpInBackground() { (isSuccessful, error) in
    if isSuccessful {
    print("注册成功")
      SVProgressHUD.dismiss()
      self.performSegue(withIdentifier: "goToChat", sender:
self)
    }else{
      print("注册失败，错误原因：\(error!.localizedDescription)")
    }
  }
}
```

构建并运行项目，在用户登录和注册的时候可以看到SVProgressHUD的效果，如图11-27所示。

图11-27　SVProgressHUD的效果

11.11.2　区别不同的用户

本项目的最后一件事情就是改善应用程序的外观，虽然读这本书的大都是程序员，不会天天去做设计的工作，但是某些基本的地方还是要我们亲自去处理的。

当前不管是哪位用户的聊天消息，在表格视图中都是同样的风格，接下来我们需要通过ChameleonFramework区分它们的风格。

实战：改善应用程序的外观。

步骤1：在ChatViewController类中导入框架ChameleonFramework。

步骤2：在ChatViewController的viewDidLoad（）方法的最后，添加一行代码。

```
override func viewDidLoad() {
  super.viewDidLoad()
  ……
  // 设置表格视图中没有单元格之间的分割线
  messageTableView.separatorStyle = .none
}
```

接下来我们需要借助Chameleon为不同的用户设置不同的风格。Chameleon（变色龙）是一款轻量级但功能强大的iOS平板颜色框架。它可以让我们的应用轻而易举地保持美丽的界面。

变色龙是市面上第一个也是唯一一个专注于"平面色彩（flat color）"的颜色框架。使用变色龙，可以不用记忆那些UIColor RGB值，节省数个小时去计算出你App所使用的正确颜色组合。

步骤3：在tableView：cellForRowAt：方法中，添加下面的代码。

```
func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
  ……
  if cell.senderUsername.text == BmobUser.current().username {
    cell.avatarImageView.backgroundColor = UIColor.flatMint()
    cell.messageBackground.backgroundColor =
UIColor.flatSkyBlue()
  }else {
    cell.avatarImageView.backgroundColor =
UIColor.flatWatermelon()
    cell.messageBackground.backgroundColor = UIColor.flatGray()
  }
  return cell
}
```

这里我们需要先判断当前的消息是来自自己还是其他人，如果是自己则让单元格的头像背景色为薄荷色，消息的背景色为天空蓝。如果是其他人的话则是西瓜红和灰色。

构建并运行项目，同时打开两个模拟器，然后用两个用户的身份发送消息，可以看到最终的效果，如图11-28所示。

图11-28　不同用户通过颜色来区分

第12章　Git、GitHub和版本控制

本章我们将重点学习如何使用Git和GitHub来帮助我们更加方便地维护项目。在之前的学习中，我们仅仅是跟随本书所提供的链接，从GitHub网站下载初始的骨架项目。本章则会正式向大家介绍Git是如何工作的，以及如何使用它完成各种任务。也将介绍如何使用Xcode及命令行进行版本控制、克隆仓库和做分支等。

12.1 版本控制和Git

假设我们创建了一个全新的Swift文件，并在其中写了几行代码，那么现在就可以通过Git将其推送到版本控制系统之中，我们称这次的保存为保存点一，它代表这是第一次保存。

之后我又写了几行代码，并将其再次推送到版本控制系统，我们管它叫保存点二。再之后的两次版本修改，依次叫保存点三、保存点四。但是，在最后一次修改代码的时候，我故意搞乱了很多代码，导致代码文件根本无法正常运行，并且也无法修复，因此我只能把最后这个版本的代码文件删除并销毁。

但是作为开发者，我们肯定会清楚地知道，在销毁了一个文件以后，项目的运行可能会发生问题，这是因为代码文件之间都是相互联系、相互引用的，所以我们急需回滚到之前的一个正常版本。这就是Git的作用。

当然我们也可以使用其他工具来完成上述事情，但是Git是现今最流行的工具。我们可以通过Git比对当前的这个混乱版本代码与之前的正常版本代码的不同，或者使用最简单的方式回滚到之前的保存点四，甚至于回滚到保存点三等更早期的版本。

12.2　使用Git和命令行进行版本控制

我们会通过一个实例向大家展示如何通过Git命令行来进行版本控制。

步骤1：打开MacOS系统的终端应用程序——Terminal，然后在终端导航到当前用户的桌面（Desktop）目录。再创建一个新的目录Story，并进入Story目录。

```
cd ~/Desktop
mkdir Story
cd Story
```

步骤2：在Story目录中创建一个新的chapter1.txt文件，并在该文件中输入一些文字信息，保存并退出。

```
touch chapter1.txt
open chapter1.txt   // 或者使用vim chapter1.txt命令编辑该文件
```

步骤3：为Story目录创建一个本地仓库，并跟踪该目录中所有文件的改变，我们需要在Story目录中键入git init命令。

```
git init
```

此时命令行会提示“初始化了一个空的Git仓库在Story/.git/”，如图12-1所示。

图12-1　执行git init命令

此时通过Finder在Story目录中你看不到任何的变化，但是在终端中执行ls-a命令，你就会发现Story目录中多出一个.git的隐藏目录，该目录会用于跟踪你提交的所有改变，从而实现版本控制。

当前的Story目录也叫工作目录，我们使用这个目录来学习版本控制。目前，我们只是开始跟踪文件的改变，比如chapter1.txt文件。然后，我需要添加这个文件到暂存区（Staging Area）。暂存区基本上是一个中间区域，你在这里可以选取目录中需要提交的文件。

步骤4：使用git status命令查看当前暂存区的状况，红色代表它目前还是未被跟踪的文件，比如chapter1.txt。它当前只是存在于工作目录之中，但是并没有进入暂存区。

```
liumingdeMacBook-Pro:Story liuming$ git status
On branch master

No commits yet

Untracked files:
(use "git add <file>..." to include in what will be committed)

  chapter1.txt

nothing added to commit but untracked files present (use "git
add" to track)
```

步骤5：使用git add chapter1.txt命令，将文件添加在暂存区中。再次执行git status命令即可发现文件变成了绿色，如图12-2所示。现在暂存区中的文件就具备了被提交的条件。

图12-2　添加文件到暂存区

步骤6：使用git commit-m“完成Chapter 1”命令提交，如图12-3所示。命令参数-m之后的双引号中代表的是提交信息，这个参数非常重要，它可以帮助我们跟踪提交过程中都做了哪些改变。

图12-3　将暂存区的修改提交到Git仓库

当我们创建保存点的时候，你需要尽可能标记从前一次提交到当前版本都做了哪些改变。提交信息可以完全根据你自己的需求来定，比如可以是将“初始化提交”作为初始信息；或者你想更加有针对性，可以是“完成Chapter 1”。

步骤7：通过git log命令可以查看之前提交的信息，如图12-4所示。

图12-4　查看目前仓库的状态

终端所列出的信息包括提交的时间（Date）、提交的作者（Author）及哈希数（类似于a88a2dbae9a1a393a8571069c1fee3d83e50f79e）。其中，这个哈希数是本次提交的唯一标识。信息的最后是关于本次提交的文本信息。

步骤8：使用touch命令，再创建chapter2.txt和chapter3.txt两个文件，并使用open命令为文件添加文字内容。

步骤9：通过git status命令可以知道chapter2.txt和chapter3.txt两个文件目前没有被添加到暂存区。你可以通过类似于之前的命令将两个文件一个一个添加到暂存区，或者直接使用git add.命令将目录中所有的文件都添加到暂存区，如图12-5所示。

图12-5　使用git add.命令添加多个文件到暂存区

步骤10：使用git commit-m"完成Chapter 2和Chapter 3"命令完成提交操作，再利用git log命令查看提交信息，如图12-6所示。

图12-6　查看第二次提交的信息

现在可以在列表中看到两个提交信息，它们的哈希数值并不相同。

让我们重新梳理一下整个Git流程。在工作区Story目录中我们创建了一个文件，这里我们使用git init初始化这个工作区并创建本地仓库。之后，我们使用git add将文件添加到暂存区中。为什么要有暂存区的存在呢？有时候你可能不想要所有的文件被跟踪和提交，通过暂存区可以筛出你想跟踪的文件。一旦暂存区中的文件有了改动，我们可以使用git commit将其提交到本地仓库（Local Repository）。

现在我们的文件已经进入本地仓库中，这也就意味着即使弄坏了文件，我们仍然可以使用git checkout命令将文件回滚到之前提交的一个版本。

实战练习：回滚操作。

步骤1：打开chapter3.txt文件，修改其中的内容，然后保存。至于混乱的文件内容，我们大可不必担心，我们会回滚到之前的版本。

步骤2：使用git status命令查看目前有过改动的文件为chapter3.txt。然后使用git diff chapter3.txt命令查看两个版本之间的不同，如图12-7所示。

图12-7　对比文件修改前后的不同

其中红色部分是之前的文本，而绿色部分是修改以后的文本。

步骤3：使用git checkout chapter3.txt命令，将chapter3.txt文件回滚到之前的一个版本，如图12-8所示。

图12-8　将chapter3.txt文件回滚到之前的版本

12.3　GitHub和远程仓库

在上一节中，我们学习了如何在本地实现Git和版本控制。本节我们将了解如何在GitHub网站中创建远程仓库。在之前的学习中我们已经用到了在GitHub上下载的骨架项目。如果你目前在GitHub中还没有账号，则可先创建一个账号。

实战：在GitHub中创建远程仓库。

步骤1：在成功登录以后，单击个人主页面的右上角的+号链接，创建一个新的仓库。将Repository name设置为Story，将Description设置为我的主场，选中Public，并确认未勾选Initialize this repository with a README，然后单击页面中的Creating repository按钮，如图12-9所示。

图12-9　在GitHub中创建仓库

作为免费用户，GitHub只允许我们将仓库设置为公开，这也就意味着任何人都可以看到该仓库的内容，但是我们可以设置谁可以提交内容到远程仓库。

步骤2：接下来我们会看到两种设置仓库的方式：一种是通过GitHub For Mac应用程序客户端设置仓库，另一种则是使用命令行指令设置仓库。我们将会推送本地现存的仓库到远程仓库中。

步骤3：复制顶部HTTPS中的链接 https://github.com/liumingl/Story.git，再使用图12-10中标注的两行代码，推送本地仓库到远程仓库。在终端进入Story目录，通过git log命令查看之前的提交是否正常。

步骤4：使用git remote add origin https://github.com/liumingl/Story.git命令，其中origin代表创建

的远程名称，理论上可以给它起任何名字。只不过绝大多数的程序员都起这个名字，已经习惯了。

现在，远程连接origin已经创建，我们可以推送本地仓库到远程仓库了。

图12-10　GitHub中生成的推送远程仓库到本地的命令行代码

步骤5：使用git push-u origin master命令进行推送，其中u选项代表连接你的远程和本地仓库，之后是推送origin，也就是之前定义的远程名称。推送的目标是Master，它是分支的名称，如图12-11所示。

提示　master分支是GitHub默认的所有提交的主分支。

图12-11　推送本地仓库内容到GitHub

上传成功后，刷新浏览器可以看到上传的文件列表，如图12-12所示。

图12-12　在GitHub看出推送后的内容

12.4 Gitignore

在本节，我们将学习有关Git Ignore及如何设置规则防止提交某些本地文件到远程仓库的相关知识。

步骤1：打开终端应用程序，导航到Desktop，创建一个新的Project目录。然后在Project目录中创建4个文本文件。

```
MacBook-Pro:Story liuming$ cd ~/Desktop/
MacBook-Pro:Desktop liuming$ mkdir Project
MacBook-Pro:Desktop liuming$ cd Project/
MacBook-Pro:Project liuming$ touch file1.txt
MacBook-Pro:Project liuming$ touch file2.txt
MacBook-Pro:Project liuming$ touch file3.txt
MacBook-Pro:Project liuming$ touch secrets.txt
```

在secrets.txt文件中，我们可能存储了安全密码或是API Key，在与GitHub远程同步的时候，我们绝对不希望将该文件推送到具有公开权限的远端GitHub服务器。

我们想添加到忽略文件的另一种类型就是本地设置或用户的偏好设置，在项目中肯定有很多类似这样的文件，我们绝对不想让其他用户下载这些文件并复制到他们的项目中。一个最常见的例子就是要忽略.DS_Store的文件。.DS_Store文件是一个设置文件，用于保存图标、进行文件排序等。也就是说该文件仅服务于创建者自己对文件夹的偏好设置。

.DS_Store文件还是一个隐藏文件，我们不可能在finder里面看到它。但是在命令行中我们可以使用ls-a看到所有的隐藏文件。

步骤2：在Project目录中，使用touch.gitignore命令创建一个隐藏文件。虽然在创建以后看不见它，但是使用ls-a可以找到它。

如果使用open.gitignore命令会使用默认编辑器打开.gitignore文件，我们可以在其中添加希望忽略的文件，但是目前我们先直接将该文件关闭。

步骤3：使用git init命令初始化仓库，然后使用git add.将Project目录中所有文件添加到暂存区。利用git status可以查看所有文件的状态，如图12-13所示。

图12-13　创建git忽略文件

通过git status命令可以发现所有的文件都被添加到了暂存区，如果现在提交的话会事与愿违，因为我们不希望提交.DS_Store和secrets.txt文件到仓库。现在需要先移除添加到暂存区的所有文件。

步骤4：使用git rm--cached-r.命令，将已经添加到暂存区的所有文件移出暂存区。其中rm代表移除（remove），--cached代表缓存，-r代表循环到所有的文件，而最后的.代表所有暂存区的文件。

```
MacBook-Pro:Project liuming$ git rm --cached -r .
rm '.gitignore'
rm 'file1.txt'
rm 'file2.txt'
rm 'file3.txt'
rm 'secrets.txt'
```

此时使用git status可以发现所以的文件都处于未跟踪状态。

```
MacBook-Pro:Project liuming$ git status
On branch master
No commits yet
Untracked files:
(use "git add <file>..." to include in what will be committed)

  .gitignore
  file1.txt
```

```
  file2.txt
  file3.txt
  secrets.txt

nothing added to commit but untracked files present (use "git
add" to track)
```

步骤5：在编辑器中打开.gitignore文件，添加两个需要忽略的文件——.DS_Store和secrets.txt。在忽略文件中，我们还可以利用通配符来筛选特定的文件，比如*.txt、*.swift等。

步骤6：再次执行git add.命令，通过git status命令可以发现添加到暂存区的文件并没有.DS_Store和secrets.txt。

```
MacBook-Pro:Project liuming$ git add .
MacBook-Pro:Project liuming$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   .gitignore
new file:   file1.txt
new file:   file2.txt
new file:   file3.txt
```

步骤7：执行git commit-m"初始化完全提交"。

```
MacBook-Pro:Project liuming$ git commit -m "初始化完全提交"
[master (root-commit) 33f2500] 初始化完全提交
 4 files changed, 2 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 file1.txt
 create mode 100644 file2.txt
 create mode 100644 file3.txt
```

下面将打开Xcode并创建一个全新的Xcode项目，这里将会展示如何在项目中添加一个新文件。

步骤1：创建一个全新的Single View App项目，将Project Name设置为Test，在保存项目之前一定要确保勾选了Create Git repository on my Mac。接下来，我们对故事板做出一些改变，在视图中添加一个Image View，保存并退出Xcode。

注意　在创建项目的时候已经通过Xcode创建了仓库，所以项目中的所有文件已经被添加到暂存区中，任何的修改都会被标记。

步骤2：在终端中导航到Test项目，创建.gitignore文件，并编辑该文件忽略所有Swift项目可以忽略的文件类型。

为了可以查阅出Swift项目可以忽略的文件类型，在GitHub网站中搜索gitignore关键字，并找到github/gitignore链接。在该仓库中找到Swift.gitignore文件，并复制文件中所有内容到本地Test项目的.gitignore文件中。如图12-14所示。另外，强烈建议在文件的顶端添加对.DS_Store文件的忽略。

图12-14　所有可以被忽略的文件格式

步骤3：在Test目录中执行git init、git add.和git status命令，可以发现在暂存区中新添加了.gitignore文件以及修改了Main.storyboard文件。

```
(use "git reset HEAD <file>..." to unstage)

new file:   .gitignore
modified:   Test/Base.lproj/Main.storyboard
```

步骤4：执行git commit-m"初始化提交"命令，将改动提交到仓库。

```
MacBook-Pro:Test liuming$ git commit -m "初始化提交"
[master dd17cf0] 初始化提交
```

```
2 files changed, 83 insertions(+), 3 deletions(-)
create mode 100644 .gitignore
```

12.5 克隆

本节我们将了解如何将远端仓库通过克隆（Cloning）拉回到本地的机制。

在GitHub中搜索swift-2048，可以找到austinzheng/swift-2048项目。接下来，我们就将它克隆到本地。

步骤1：进入swift-2048主页面，并单击右上角的Clone or download按钮。单击弹出面板中链接右侧的图标，将链接复制到剪贴板中。

提示　在之前的实战练习中，因为还没有系统学习过如何使用Git，所以那个时候我们只是简单地单击Download ZIP链接，直接下载骨架项目，这种方式其实并不是一个好的选择。

步骤2：打开Mac系统的终端应用程序，导航到桌面或者是你希望项目存储的位置，使用git clone{在GitHub中拷贝的URL链接}命令，将项目克隆到本地。其中{在GitHub中拷贝的URL链接}需要替换为相关链接。

```
MacBook-Pro:Desktop liuming$ git clone
https://github.com/austinzheng/swift-2048.git
Cloning into 'swift-2048'...
remote: Counting objects: 287, done.
remote: Total 287 (delta 0), reused 0 (delta 0), pack-reused
287
Receiving objects: 100% (287/287), 91.23 KiB | 149.00 KiB/s,
done.
Resolving deltas: 100% (155/155), done.
```

现在，在桌面位置上出现了swift-2048目录，并且该项目已经从gitHub克隆到了本地。

步骤3：在Xcode中打开swift-2048项目，并修改项目设置中的Bundle Identifier，以及Team设置，如图12-15所示。

图12-15　修改克隆项目的配置信息

步骤4：在终端中进入swift-2048目录，然后通过git log命令查看项目之前所提交的信息，如图12-16所示。

图12-16　查看克隆项目的git状态

12.6　分支和迁移

这里先通过一个简单的例子说明分支的作用。假设我们在本地仓库中有1和2两个版本的Commit。此时此刻，团队中的一个程序员想尝试着在项目中做出一些非常牛的新特性。这是一个非常新奇的点子，但是不保证能够制作成功。所以，我们可以在非主分支中做这些事情，也就是非master分支。

只要我们愿意，可以创建很多很多的分支。就拿现在的这个情况来说，我们在第二次提交之后，会创建一个新的分支——Experimental，并且开始在这个分支上添加一些新的特性，写入一些新的代码，并且在最后提交这些修改。

与此同时，我们还会在主分支上继续工作，提交一些常规的更新代码来维护我们的主项目。这时，还可以继续维护，并在Experimental分支上进行代码的编写，尝试做出一些新的东西，并且提交这些想法到这个分支上。

目前的仓库中已经有两个分支，它们是并行存在的，我们可以同时进行维护和开发。如果在未来的某个时刻，我们觉得分支中的代码非常酷，而且非常实用，就可以将它迁移回主分支上。在两个分支合并以后，我们还可以继续在主分支提交修改，或者制作更多的分支。

有些时候，开发者需要尝试实现一些新的想法，或者是需要修复一些旧的bug，然而这些事情有可能会破坏项目，所以不能在master分支上直接修改。此时，可以利用其他分支，一旦修改没有产生其他的问题，就可以把它推送回到master分支上，如图12-17所示。

图12-17　GitHub中分支示意图

实战：实现分支。

步骤1：打开终端应用程序，导航到之前在Desktop上创建的Story目录中。该目录存储这之前的三个文件：chapter1.txt、chapter2.txt和chapter3.txt。使用git log命令可以查看到之前的提交信息。

步骤2：使用git branch alien-plot命令创建一个名叫alien-plot（外星人阴谋）的分支。通过git branch可以查看当前仓库的分支情况。

```
MacBook-Pro:Story liuming$ git branch alien-plot
MacBook-Pro:Story liuming$ git branch
  alien-plot
* master
```

你可以看到有一个分支叫alien-plot，另一个叫master。Master前面的星号代表当前的分支，上述代码表示我们当前是在master分支上面。

步骤3：使用git checkout alien-plot命令切换分支到alien-plot上面。

```
MacBook-Pro:Story liuming$ git checkout alien-plot
Switched to branch 'alien-plot'
```

在该分支上对chapter1.txt和chapter2.txt文件进行简单修改。

步骤4：使用git add.命令将修改添加到暂存区之中。然后再使用git commit-m"在alien-plot分支上，修改chapter1和2"。

```
MacBook-Pro:Story liuming$ git add .
Book-Pro:Story liuming$ git commit -m "在alien-plot分支上，修改
chapter1和2"
[alien-plot 88f59f5] 在alien-plot分支上，修改chapter1和2
 2 files changed, 6 insertions(+), 2 deletions(-)
```

步骤5：使用git log命令查看提交信息，最新的提交是作用在alien-plot分支上。

```
MacBook-Pro:Story liuming$ git log
commit 88f59f5000cde5c4e0ff8b29ea877e21f8435229 (HEAD -> alien-
plot)
Author: Liuming <liuming_cn@qq.com>
Date:   Thu Feb 15 07:36:23 2018 +0800
```

在alien-plot分支上，修改chapter1和2

```
commit 0c7ec43735f03dbfb9450e9894c5fd28e8cb8b9b (origin/master,
master)
Author: Liuming <liuming_cn@qq.com>
Date:   Wed Feb 14 10:54:54 2018 +0800
```

完成Chapter2和Chapter 3

步骤6：使用git branch查看当前的分支，并使用git checkout master切回到主分支。

```
MacBook-Pro:Story liuming$ git branch
* alien-plot
master
MacBook-Pro:Story liuming$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

此时再次查看Story目录中的chapter1和chapter2文件，你会发现它们的内容还是停留在之前主分支的状态，是不是很神奇呢?

步骤7：在主分支中创建一个新的文件——chapter4.txt，并在其中输入一些文字内容，将该文件添加到仓库中。

```
MacBook-Pro:Story liuming$ touch chapter4.txt
// 打开chapter4文件，添加一些文字内容
MacBook-Pro:Story liuming$ open chapter4.txt
MacBook-Pro:Story liuming$ git add .
```

```
MacBook-Pro:Story liuming$ git commit -m "添加 chapter4"
[master 1469eae] 添加 chapter4
 1 file changed, 1 insertion(+)
 create mode 100644 chapter4.txt
```

步骤8：使用git log命令查看当前的仓库日志，你可以发现此时的列表中并没有之前的alien-plot分支信息，而是全部与master分支相关。

```
MacBook-Pro:Story liuming$ git log
commit 1469eae2e8c715108bcc5f6b34d9dd253360e108 (HEAD ->
master)
Author: Liuming <liuming_cn@qq.com>
Date:   Thu Feb 15 07:46:15 2018 +0800

添加 chapter4

commit 0c7ec43735f03dbfb9450e9894c5fd28e8cb8b9b (origin/master)
Author: Liuming <liuming_cn@qq.com>
Date:   Wed Feb 14 10:54:54 2018 +0800

完成 Chapter2和Chapter 3

commit a88a2dbae9a1a393a8571069c1fee3d83e50f79e
Author: Liuming <liuming_cn@qq.com>
Date:   Wed Feb 14 10:27:56 2018 +0800

完成 Chapter 1
```

如果你愿意，完全可以使用git checkout alien-plot命令切回到alien-plot分支，该分支目前还是只要3个文件。

接下来，我们需要实现将alien-plot分支中的改变合并到master分支。

实战：将alien-plot分支合并到master分支。

步骤1：确保仓库当前是在master分支。为了合并alien-plot分支内部的改变，可使用git merge alien-plot命令。

此时终端会打开一个文本编辑器，并且允许我们添加一些合并信息。退出文本编辑器以后，马上就会看到“Merge branch'alien-plot'”的信息。

```
commit 183599dcdf94955dcaeca81581d671cdea6d2f3a (HEAD ->
master)
Merge: 1469eae 88f59f5
Author: Liuming <liuming_cn@qq.com>
Date:   Thu Feb 15 08:00:11 2018 +0800

Merge branch 'alien-plot'
```

步骤2：使用git push origin master-u命令，将改动提交到GitHub远端服务器上面。在提交成功以后，通过浏览器可以直接在GitHub网站中查看所有的提交和分支，如图12-18所示。

图12-18　在GitHub网站中查看所有的提交和分支

```
MacBook-Pro:Story liuming$ git push origin master -u
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 903 bytes | 903.00 KiB/s, done.
Total 9 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 2 local
objects.
To https://github.com/liumingl/Story.git
 0c7ec43..183599d  master -> master
Branch master set up to track remote branch master from origin.
```

在Insights页面中的Network标签中，可以看到分支的图例，如图12-19所示。

图12-19　分支的图例

对于上面的这些操作，我们完全可以在GitHub网站上完成，而且操作非常方便。

实战：在GitHub中操作。

步骤1：在GitHub中创建一个新的仓库Story2，勾选Initialize this repository with a README，单击Creating repository按钮。

步骤2：在Story2页面中单击Create new file按钮，创建chapter1.txt文件，并为该文件添加一些文字内容，如图12-20所示。在页面底部填写提交信息并单击Commit new file按钮。

此时的仓库仅有一个分支——master，分支中包含了刚刚创建的chapter1.txt文件。

步骤3：单击Story2页面的Branch：master链接，并创建一个名为experimental的分支，如图12-21所示。现在，experimental分支中直接包含master分支的所有拷贝。

图12-20　在Story2中创建文件

图12-21　创建experimental分支

步骤4：在experimental分支中修该chapter1.txt文件的内容，然后将其提交，如图12-22所示。

图12-22　提交新的修改

在默认状态下，chapter1.txt文件的修改会提交到experimental分支。

步骤5：在Story2主页面中将分支切换到master，查看chapter1.txt文件，发现其还是之前的状态。

此时，你会发现在Story2的主页面上，有一个淡黄色的提示条，如图12-23所示。

图12-23　回到master分支

通过其右侧的Compare&pull request按钮，可以把experimental分支的修改合并到master中。

步骤6：确认当前是在master分支中，创建一个全新的chapter2.txt文件。通过Insight/Network可以发现，该项目一共有2个分支，黑色的是master分支，下面蓝色的是experimental分支，如图12-24所示。

图12-24　分支图例

步骤7：单击主页面中的New pull request按钮，然后将base设置为master，将compare设置为experimental。单击Create pull request按钮，如图12-25所示。

在当前页面的底部可以看到分支中内容不同的部分，如图12-26所示。

图12-25　合并分支

图12-26　查看分支的不同内容

步骤8：在确认页面中单击Merge pull request按钮，再单击Comfirm merge按钮即可。在Insights/Network中，你可以看到图谱已经显示两个分支又合并到一起了，如图12-27所示。

图12-27　分支图例

12.7　在Xcode 9中使用Git和GitHub

目前，我们已经通过命令行方式实现了很多的版本控制功能，有很多的程序员都喜欢这样的操作方式。在全新的Xcode 9中，也实现了强大的版本控制功能，并且好于之前的任何一个Xcode版本。

在本节中，我们会尝试使用Xcode 9来替代所有的命令行指令。但是，最后选择哪一种方式，这完全取决于你的喜好。如果你希望在不触碰任何按键的情况下去实现之前所学的所有版本控制功能，那么Xcode 9将是一个非常好的选择。

实战：使用Xcode内置的GitHub。

步骤1：创建Xcode项目，选择iOS/Single View App模板，将Product Name设置为DiDi（这绝不是模仿滴滴的项目），并确认勾选了Create Git repository on my Mac。

提示　为了让项目具有版本控制能力，我们需要使用Xcode在Mac上面创建Git仓库，这相当于在终端里面执行git init指令。

在创建项目以后，所有的文件都会被添加到暂存区之中，任何的添加、删除、修改操作都会在项目导航中看到相应的提示。

步骤2：在故事板的视图里面添加2个Label，让它们看起来像图12-28所示的样子。

图12-28　DiDi项目的用户界面

另外，在项目的Assets.xcassets文件中添加两张素材图片：Milk和cow。

项目在模板初始化以后便完成了第一次提交，我们可以通过源代码控制导航（Source Control Navigation）窗口查看当前的分支，如图12-29所示。

由此可见，当项目在创建后完成了第一次提交。接下来，让我们进行第二次提交。

步骤3：在菜单Source Control中选择Commit...，在弹出的代码控制窗口的左侧列表中，我们可以看到所有发生改变的文件，从中可以发现项目中添加了两种图片，以及Main.storyboard文件中的代码发生了怎么样的改变。在填写了信息说明以后，便可以提交项目中所涉及的7个相关文件到本地仓库，我们可以从Xcode左侧的列表中查到它们的详细修改信息，如图12-30所示。

图12-29　提交初始化项目到GitHub

此时，在Branches/master中会看到两个提交信息。

步骤4：在提交成功以后，在Main.storyboard视图中随意添加几种不同的UI控件，甚至出现混乱的布局也没有关系。在ViewController.swift文件中随意添加一些文字内容，即使导致编译器报错都没有关系，最后再删除AppDelegate.swift文件。

图12-30　进行第二次提交

步骤5：在Source Control/Discard All Changes...，此时项目会回滚到之前的最后一次提交的状态。

如果我们现在需要创建一个分支的话，也是非常的简单。

实战：在Xcode中创建分支。

步骤1：在Branches/master中，选择第二次提交的条目——右击该条目，并在快捷菜单中选择Branch From XXXX。在面板中将Branch设置为milk-design，如图12-31所示。

图12-31　为DiDi项目创建分支

此时，在Branches中出现了两个分支：master和milk-design。

步骤2：虽然目前有2个分支，但是当前的分支为master。右击milk-design，在快捷菜单中选择Checkout...，将当前的分支设置为milk-design，如图12-32所示。

图12-32　确认当前分支

步骤3：在故事板的视图中添加一个Image View，将图像设置为Milk。在菜单中选择Source Control/Commit，填写提交信息为“修改ViewController的用户界面”，然后单击Commit 1File按钮。

此时在Branches/milk-design中出现了3个提交条目。

步骤4：再次选中Branches中的master，在提交信息为“添加了Label和Images。”的提交条目上单击鼠标右键，通过Branch From XXXX命令创建另一个分支——cow-design。

步骤5：通过Checkout命令，将cow-design切换到当前分支。因为是基于master的分支，所以此时的故事板视图中还是只有两个Label控件。添加Image View到视图之中，如图12-33所示。接下来，提交当前的修改到cow-design分支，并设置提交信息为“修改用户界面–添加cow图像”。

在成功创建了另外两个分支以后，接下来我们需要将milk-design合并到master分支中。

实战：将两个分支项目合并到master分支中。

步骤1：在Source Control导航的Branches中，将milk-design设置为当前分支，右击master分支，在弹出的快捷菜单中选择Merge"milk-design"into"master"...。

此时master分支中的故事板视图里面已经包含了之前在milk-design中的设计布局，如图12-34所示。

图12-33　提交修改到cow-design分支

图12-34　将milk-design与master分支合并

步骤2：在master分支上打开ViewController.swift文件，在该类中重写viewWillAppear（）方法。提交该修改，并设置提交信息为：添加view will appear。

```
override func viewWillAppear(_ animated: Bool) {
}
```

接下来，我们需要将本地的这些改动同步到GitHub网站上。

实战：将修改提交到GitHub。

步骤1：在Xcode的偏好设置中，选择Accounts标签，单击+号以添加自己的GitHub账号，如图12-35所示。

图12-35　在Xcode中添加GitHub账号

步骤2：在Source Control导航中，右击顶部的DiDi条目（蓝色图标），在快捷菜单中选择Crate"DiDi"Remote on GitHub...，或者选择Add Existing Remote...将当前项目添加到现存的仓库中。这里让我们创建一个新的仓库。

步骤3：在创建“DIDI”远程仓库的面板中设置仓库名称为DiDi，单击Create按钮。

步骤4：在GitHub网站中，你可以看到已经上传的DiDi项目。

步骤5：在Xcode中打开ViewController.swift文件，删除其中的didReceiveMemoryWarning（）方法。在提交修改的时候，勾选面板下方的Push to remote选项，可以将最新的改动推送到GitHub上。或者直接使用菜单Source Control/Push命令将修改推送到GitHub上，如图12-36所示。

图12-36　将修改推送到GitHub上

步骤6：在GitHub上面为项目创建一个新的README.md文件，并在其中输入一些对项目的描述内容。

如果此时我们从本地推送修改到GitHub上面，会提示本地仓库过期，如图12-37所示。这是因为远端服务器中有文件被修改，所以在推送之前需要先从远端拉回所有的改动和变化。

图12-37　本地仓库的过期提示

步骤7：先执行Source Control/Commit...命令，提交本地的所有改动。然后执行Pull命令将远端修改拉回到本地，此时在DiDi目录中出现

了README.md文件。此时我们再执行Push命令就没有任何问题了。

第13章　使用Core Data、User Defaults学习本地数据存储

本章带领大家制作一款类TODO List的App。它的灵感来源于我们经常使用的TODO类应用程序，如图13-1所示。

该应用允许我们创建一个新的事务列表，比如可以创建一个购物清单的任务列表，然后通过一个特殊的颜色区别于其他的事务列表。你可以选择其中的一个列表，然后在这个指定的列表中添加要做的事项条目。在添加了一些事项以后，你会发现这个事项列表会呈现出一个非常漂亮的从浅至深的渐变色，如图13-2所示。

图13-1　TODO应用的运行界面

图13-2　事项列表中的渐变色

图13-2所示就是我们所添加的事项条目，你可以勾选这些条目，代表该任务已经完成，也可以通过向左滑动将它们删除。除此以外，你还可以在事务列表中搜索关键字，比如我们可以搜索“苹果”关键字，然后单击搜索，这样就可以看到在列表中会列出包含“苹果”的相关条目。最神奇的地方在于，应用中所有的数据都被保存到了模拟器本地，或者是iPhone的物理真机上面。这也就意味着如果你在设备上关闭了应用，或者是升级了该应用，或者是升级了iOS的版本，或者是换了一部新手机，所有的数据还会储存在你的设备上。如果你单击购物清单事务的话，仍然可以看到里边存储的条目。这就是本章我们要完成的任务。

本章我们会使用不同的方式去处理本地的数据，包括使用default存储少量的数据。另外，还可以使用Core Data存储大批量的数据。Core Data就像我们所使用的数据库。在本应用中，我们将会创建一个关系型数据库——Realm，并以该数据库作为后台。最后，我们还会编写一些前端代码，让我们的应用程序看起来更加漂亮。

13.1 创建UITableViewController的子类

首先，我们需要创建一个Single View App，将Product Name设置为TODO，这里确保Use Core Data处于未勾选的状态，当我们在后面需要使用到Core Data的时候会手动添加该功能。因为在项目之初，所以我们还是要尽可能保持项目架构的精简、整洁和清晰。

另外，在保存项目之前请确保勾选Create Git repository on my Mac，因为在本章我们将会使用不同的方法去连接本地数据。有的时候，我们会在实现了数据连接功能以后再回滚到之前的版本，去对比两种方法的不同。

在创建好新的项目以后，首先我们要在项目设置中取消Deveice Orientation中的Landscape Left和Landscape Right勾选状态，因为我们只想让应用纵向显示。

接下来，我们就要在项目中通过一种全新的方式来创建表格视图控制器。

实战：在项目中创建表格视图控制器。

步骤1：在故事板中，从对象库里面拖曳一个新的表格视图控制器（Table View Controller）。迄今为止，我们一直在使用标准的视图控制器（View Controller），这两个控制器从外观上来说还是有很多不同的。如图13-3所示，表格视图控制器自带有一个表格视图、一个Prototype单元格及所有的委托协议。

步骤2：在故事板中将之前指向视图控制器的箭头拖曳到表格视图控制器。箭头指向的控制器代表初始视图控制器（initial view controller），也就意味着一旦应用启动，该控制器的视图就会呈现到屏幕上，如图13-4所示。

在调整好初始控制器以后，删除之前项目模板所生成的视图控制器。

如果在删除前忘记调整初始箭头，则可以选中表格视图控制器，然后在Attributes Inspector中勾选View Controller部分中的Is Initial View Controller，如图13-5所示。

图13-3　对象库中的表格视图控制器

图13-4　将表格视图控制器设置为初始控制器

图13-5　在Attributes Inspector中设置初始控制器

此时你会发现，目前在故事板中的表格视图控制器并没有连接到项目中的ViewController.swift文件，因为该文件是与之前故事板中被删除的控制器关联的。接下来，我们要将该文件修改为符合表格视图控制器的类文件。

步骤3：在项目导航中打开ViewController.swift文件，修改ViewController类的声明。为了更加明确，先将类名称修改为TodoListViewController。需要注意的是，这里将其父类修改为UITableViewController，代表该类的父类是UITableViewController。

```
class TodoListViewController: UITableViewController {
```

然后，在项目导航中，将ViewController.swift文件名修改为TodoListViewController.swift。

步骤4：回到故事板中，选中表格视图控制器后在Identifier Inspector中将Class设置为TodoListViewController，此时故事板中的用户界面与TodoListViewController代码类建立关联。

此时Xcode有一个警告错误：Prototype table cells must have reuse identifiers。我们需要为表格视图单元格指定一个标识。

步骤5：选中Prototype Cell，在Attributes Inspector中将Identifier设置为ToDoItemCell，警告消失。

提示　如果在故事板中不方便选中Prototype Cell对象，则可以借助大纲导览视图（Document Outline）中的列表项选取表格视图中的单元格。

接下来，让我们对用户界面做一些修改。

实战：修改用户界面。

步骤1：故事板中选中表格控制器视图，菜单中选择Editor/Embed In/Navigation Controller，让其成为导航控制器中的根控制器。

步骤2：选中表格视图控制器顶部的导航栏，在Attributes Inspector中将其Title设置为TODO，如图13-6所示。

图13-6　修改导航控制器的Title属性

步骤3：选中导航控制器视图顶部的导航栏，在Attributes Inspector中将Bar Tint颜色修改为蓝色。再将Title Color设置为白色，如图13-7所示。

图13-7　修改导航栏的Bar Tint属性

接下来，我们要设置与表格视图控制器相关的代码。因为在故事板中我们从对象库直接创建了表格视图控制器，在代码类中直接设置ToDoListViewController为UITableViewController的子类，所以我们就不用像之前那样单独声明UITableViewDelegate和

UITableViewDataSource协议，以及建立与表格视图的IBOutlet关联了。

实战：设置TodoListViewController类中的代码。

步骤1：在TodoListViewController中创建itemArray数组。

```
let itemArray = [" 购买水杯", " 吃药", " 修改密码"]
```

显然，我们利用该数组临时呈现一些事务列表项。

步骤2：为表格视图创建两个Table View DataSource方法，一个用于返回要显示的单元格对象，另一个则用于显示表格视图有多少行。

```
//MARK: - Table View DataSource methods
override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(withIdentifier:
"ToDoItemCell", for: indexPath)
  cell.textLabel?.text = itemArray[indexPath.row]

  return cell
}

override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
  return itemArray.count
}
```

在cellForRowAt（）方法中，我们首先从表格视图中获取一个可复用的单元格对象，这个单元格的标识为ToDoItemCell，其是和之前在故事板中为Prototype Cell设置的标识一样的单元格。之后设置textLabel的text为数组中相应的元素内容，textLabel是每个单元格对象都会有的内置Label。

在numberOfRowsInSection（）方法中，直接返回itemArray数组的元素个数作为单元格的数量。

构建并运行项目，效果如图13-8所示。

图13-8　在单元格中显示自定义好的事项

接下来，我们将要实现的是：当用户单击单元格以后，要在调试控制台打印出该单元格的信息，并且当用户单击单元格的时候还可以呈现一个勾选标记。这些功能需要我们实现UITableViewDelegate协议中的方法。

步骤1：在ToDoListViewController类中实现下面的方法。

```
//MARK: - Table View Delegate methods
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
  print(indexPath.row)
}
```

该方法用于告诉控制器用户单击了表格视图中的哪个单元格，我们通过indexPath参数得到该信息。

构建并运行项目，如果单击了第一个单元格，则控制台会显示0。如果想要打印单元格中的内容，因为它与itemArray数组中的元素一致，所以只需要将打印语句修改为print（itemArray[indexPath.row]）即可。

目前，当用户单击单元格以后，被选中的单元格就会呈现灰色的高亮状态。我们需要换一种呈现方式。

步骤2：在tableView：didSelectRowAt：方法中添加下面的代码。

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
  print(itemArray[indexPath.row])

  tableView.deselectRow(at: indexPath, animated: true)
}
```

构建并运行项目，在用户单击单元格以后灰色高亮会逐渐变淡消失，看起来是一个非常不错的用户体验。

接下来，我们要实现的是在单元格中呈现勾选标记，这需要使用一个名为accessory的属性。

步骤3：在故事板中，通过大纲导览视图选中ToDoItemCell，然后在Attributes Inspector中将Accessory设置为Checkmark，此时你会发现单元格的右侧会出现一个勾选标记。我们还是将它设置为默认状态None，如图13-9所示。

图13-9　设置表格视图单元格的Accessory属性

步骤4：继续修改tableView：didSelectRowAt：方法中的代码。

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {

  tableView.cellForRow(at: indexPath)?.accessoryType =
.checkmark

  tableView.deselectRow(at: indexPath, animated: true)
}
```

其中，cellForRow（at indexPath：IndexPath）方法会通过indexPath参数获取到表格视图中指定单元格对象。然后再通过该单元格对象的accessoryType属性设置其属性值为.checkmark。

如果此时构建并运行项目，当用户单击单元格后确实会出现勾选标记，但是当再次单击的时候却不会有任何变化。所以我们需要借助if语句，进行勾选状态的切换。

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {

  if tableView.cellForRow(at: indexPath)?.accessoryType ==
.checkmark {
    tableView.cellForRow(at: indexPath)?.accessoryType = .none
  }else {
    tableView.cellForRow(at: indexPath)?.accessoryType =
.checkmark
  }

  tableView.deselectRow(at: indexPath, animated: true)
}
```

构建并运行项目，我们可以任意切换单元格的选中状态，如图13-10所示。

图13-10　完成单元格的勾选效果

13.2　在UIAlert中使用文本框创建新的条目

在我们进行接下来的实战练习之前，最好先提交当前的项目到远程仓库里面。

实战：提交项目到GitHub。

步骤1：在代码控制导航中，在顶部的TODO条目单击鼠标右键，在快捷菜单中选择Create“TODO”Remote on GitHub。在弹出的面板中设置Repository Name为TODO，设置Remote Name为origin，最后单击Create按钮，如图13-11所示。

步骤2：在菜单中选择Source Control/Commit...，可以发现此时我们修改了项目中不少的文件。将提交信息设置为“TODOListViewController完成datasource和delegate方法”，并勾选Push to remote：origin/master，单击Commit 4Files按钮，如图13-12所示。

此时，在Branches的master分支中，我们可以看到两个提交：Initial Commit和刚才的一次提交。我们在之后的操作中可以回滚到这两个提交时候的状态。

图13-11　在GitHub账号中创建一个新的仓库

图13-12　将修改提交到GitHub上面

现在让我们回到TodoListViewController.swift文件，此时需要为它添加一些功能。

步骤3：为了可以在表格中继续添加条目，首先需要为应用添加一个按钮。最简单的方式就是从对象库拖曳一个Bar Button Item，再将它放置到ToDoListViewController视图中导航栏的右侧。

步骤4：选中Bar Button Item，然后在Attributes Inspector中将System Item设置为Add，

此时该按钮会变成+号。再将Tint属性修改为白色，让其颜色与Title保持一致。

步骤5：为该按钮创建一个IBAction连接，方法名称为addButtonPressed。

```
//MARK: - Add New Items
@IBAction func addButtonPressed(_ sender: UIBarButtonItem) {
  let alert = UIAlertController(title: "添加一个新的ToDo项目",
message: "", preferredStyle: .alert)

  let action = UIAlertAction(title: "添加项目", style: .default)
{ (action) in
    // 用户单击添加项目按钮以后要执行的代码
    print("成功！")
  }

  alert.addAction(action)
  present(alert, animated: true, completion: nil)
}
```

当用户单击+按钮以后，会执行addButtonPressed（_sender：UIBarButtonItem）方法。在该方法中，我们会创建一个UIAlertController类型的对象，并设置警告对话框的标题为"添加一个新的ToDo项目”，风格为.alert类型，如图13-13所示。UIAlertController警告对话框一共有两种风格：Alert和ActionSheet。第一种风格会出现在屏幕的中央位置，第二种则会从屏幕底部滑出。

在addButtonPressed（_sender：UIBarButtonItem）方法中，我们接着创建了UIAlertAction类型的对象，它会在对话框中呈现一个用户可以单击的按钮，一旦用户填写了新的条目信息，就可以单击该按钮。这里设置按钮的风格为default，在单击按钮以后会执行方法中的handler闭包，这里带有一个参数，就是用户单击的这个UIAlertAction对象。在闭包中我们先简单打印一个“成功！”信息到控制台。

后面的代码会将所创建的UIAlertAction对象添加到UIAlertController对话框之中，最后通过present（）方法将警告对话框显示到屏幕上。

构建并运行项目，在单击+号以后，可以看到一个警告对话框出现在屏幕上面。当单击添加项目按钮以后，控制台会显示“成功！”信息，如图13-14所示。

图13-13　警告对话框的两种风格

图13-14　警告对话框的运行效果

接下来，我们需要让警告对话框中呈现一个文本框，这样用户才能够输入新的事务项目。当然在用户单击添加项目按钮以后，会在控制台打印事务项目信息。

实战：在警告对话框中添加文本框。

步骤1：在addButtonPressed（_sender：UIBarButtonItem）方法中添加下面的代码。

```
@IBAction func addButtonPressed(_ sender: UIBarButtonItem) {

  ……
  alert.addTextField { (alertTextField) in
    alertTextField.placeholder = "创建一个新项目..."
```

```
    print(alertTextField.text!)
  }

  alert.addAction(action)
  present(alert, animated: true, completion: nil)
}
```

我们通过UIAlertController的addTextField（）方法在对话框中添加了一个文本框，完成闭包的参数代表所创建的文本框对象，在闭包中设置了文本框的placeholder属性，并且在用户单击按钮以后，还会将用户所输入的事务名称打印到控制台。

构建并运行项目，经过测试我们发现，所填写的事务信息并没有打印到控制台。这是因为在用户单击按钮以后只调用了UIAlertAction的闭包，因此就不会再执行addTextField（）的闭包了。该闭包的代码在之前向对话框中添加文本框的时候已经被执行了。

这时，我们需要在方法中声明一个变量，用于存储alertTextField对象，这样在UIAlertAction闭包中就可以随时访问它了。

步骤2：修改addButtonPressed（_sender：UIBarButtonItem）方法，通过在方法内部声明一个变量，在UIAlertAction的闭包中就可以访问对话框中的文本框对象。

```
@IBAction func addButtonPressed(_ sender: UIBarButtonItem) {

  // 声明一个新的变量，生存期在方法的内部
  var textField = UITextField()

  let alert = UIAlertController(title: "添加一个新的ToDo项目",
message: "", preferredStyle: .alert)

  let action = UIAlertAction(title: "添加项目", style: .default)
{ (action) in
    // 当用户单击添加项目按钮以后要执行的代码
    print(textField.text!)
  }
```

```
  alert.addTextField { (alertTextField) in
    alertTextField.placeholder = "创建一个新项目..."
    // 让textField指向alertTextField，因为出了闭包，alertTextField
不存在
    textField = alertTextField
  }

  alert.addAction(action)
  present(alert, animated: true, completion: nil)
}
```

构建并运行项目，在单击按钮以后，文本框内用户所输入信息被打印到控制台中，如图13-15所示。

接下来，我们需要将文本框中的数据添加到itemArray数组之中。

步骤3：为了可以将数据添加到itemArray数组之中，将itemArray常量修改为变量var itemArray=[“购买水杯”，“吃药”，“修改密码”]。

步骤4：在UIAlertAction的闭包中，修改代码如下：

```
let action = UIAlertAction(title: "添加项目", style: .default) {
(action) in
  // 当用户单击添加项目按钮以后要执行的代码
  self.itemArray.append(textField.text!)
  self.tableView.reloadData()
}
```

这里，我们将文本框中的数据添加到itemArray数组之中。除此以外，还要通过表格视图的reloadData（）方法让其重新载入数据来更新表格视图中所显示的数据，如图13-16所示。

图13-15　通过闭包实现对文本框的操作

图13-16　将新添加的事务添加到数组之中

在完成了添加新项目到表格视图以后，我们再次提交项目到远程仓库中。设置提交信息为“添加项目功能完成！”，注意一定要勾选Push to remote：origin/master，如图13-17所示。

图13-17　提交修改到GitHub

13.3 持续本地数据存储

13.3.1 为什么需要持续的本地数据存储

在之前的实战练习中，我们实现了添加事务项目的功能。但是当我们的应用在退出以后，就会出现一个Bug。

我们先来看一下AppDelegate.swift文件，当应用在运行中出现系统级事件的时候就会调用AppDelegate类中的委托方法。

首先找到didFinishLaunchingWithOptions（）方法，当应用启动的时候会调用该方法。它的调用级别要高于初始（Initial）视图控制器的viewDidLoad（）方法。

在didFinishLaunchingWithOptions（）方法中添加一行打印语句：print（"didFinishLaunchingWithOptions"）。

当应用在前台运行的时候，如果有电话打进来就会调用applicationWillResignActive（）方法。在用户选择接听电话后，我们可以在该方法中执行相关指令防止用户数据丢失。比如用户正在应用中填写表单的时候有电话打进来，我们可以在该方法中将数据保存到本地。

在应用的界面从屏幕上消失的时候就会调用applicationDidEnterBackground（）方法。比如当用户按Home键，或者是打开了另一个不同的应用，这也就意味着我们的应用进入了后台。

在applicationDidEnterBackground（）方法中添加一行打印语句：print（"applicationDidEnterBackground"）。

还有一个非常重要的方法是applicationWillTerminate（），当应用被用户或系统终止运行的时候就会调用该方法。在该方法中添加print

（"applicationWillTerminate"）语句。

让我们再次运行项目，观察AppDelegate中各种委托方法的执行顺序。当应用启动以后，在控制台首先会看到didFinishLaunchingWithOptions，该方法会在应用启动后的第一时间运行。当用户单击Home键回到主屏幕以后，在控制台会看到applicationDidEnterBackground。另外，当我们切换到另一个应用的时候也会看到该信息。最后，当系统需要回收宝贵的内存资源，或者是被用户强制退出的时候才会执行applicationWillTerminate（）方法。双击Home键，在iOS应用程序切换选择界面中将TODO项目向上划出屏幕以后，会在控制台看到applicationWillTerminate信息。

每一个应用都有其自己独特的生存期，从应用启动开始，它会出现在屏幕上，然后它可能会退到后台，直到最后资源回收，就像是我们人类的出生——生活——死亡的过程一样。

在我们清楚了上面这些委托方法都是做什么的以后，接下来看一下Bug是如何产生的。在模拟器中启动我们的TODO项目，在添加了一个新的事务项目以后，再将应用终止，你可以想到当再次回到应用的表格视图中时，之前所添加的事务就会消失，因为我们根本没有保存它。所以，这也是我们需要持续本地数据存储（persistent local data storage）的原因。

让应用终止运行的方法有很多，可以在应用切换界面中向上划出应用程序，如图13-18所示。另外，在更新应用或更新iOS系统的时候，在系统需要回收内存资源的时候都会终止应用程序的运行。

图13-18　在模拟器中终止应用程序的运行

13.3.2　使用UserDefaults实现持续本地数据存储的功能

在iOS系统中，我们的应用都存在独立的沙箱（sandbox）之中，如图13-19所示。原因是苹果为了确保设备的使用安全。这样可以防止恶意应用获取其他应用所存储的数据（比如网银数据），或者是试图去执行一些非法操作（安卓中的Root）。

图13-19　应用程序都存在于自己的安全沙箱之中

对于你的应用来说，沙箱就像是一个小型的“监狱”。每个应用都拥有存储文件和文档的文件夹，它们可以随意读取自己的文件夹，但是绝对无法读取其他应用的文件或文档文件夹。

每次在你将iPhone的应用备份到Mac或者是iCloud云端的时候，应用中的Document文件夹总是会被备份。如果你购买了一个全新的iPhone，所存储到Document文件夹中的数据都不会被删除，这样就可以保证恢复到新iPhone的应用还保留有之前的数据。换句话说，如果将数据存储到应用的Document文件夹之中，不管是更换了iPhone，还是升级了iOS系统，还是升级了应用程序版本，你的私有数据还会安全的存在于iCloud云端或是本地的Mac电脑中。

沙箱不仅仅是让各个应用相互独立，也会让应用与iOS系统之间相互分离。我们不能恶意获取操作系统的安全数据，比如说用户的指纹数据或联系人信息等，这也是为什么iPhone比其他类型的手机更加安全。

当我们存储数据以及通过各种方法持续使用数据的时候，这些数据将会被保存到应用程序的容器内部，这样才能保证不会被其他的应用程序访问到，你也同样无法访问到其他应用的数据。

实战：使用UserDefaults存储本地数据。

步骤1：在ToDoListViewController类中创建一个全新的UserDefaults类型的对象，它使用键/值配对的方式，在整个应用运行期间维持各种数据。

```
class TodoListViewController: UITableViewController {

  let defaults = UserDefaults.standard
  ……
```

因为UserDefaults是单例模式，所以需要通过类方法standard获取该类的实例。

UserDefaults适合存储轻量级的本地客户端数据，它也有其局限性，但它非常易于使用，并且非常适合简单存储诸如字符串和数字之类的东西。比如记住密码功能，要保存一个系统的用户名、密码，UserDefaults是首选。下次再登录的时候就可以直接从UserDefaults里面读取用户上次登录的信息。

一般来说，本地存储数据还可以使用SQlite数据库，或者使用自己建立的plist文件等来存储，但是我们还需要亲自编写创建文件及读取文件的代码，比较麻烦。而使用UserDefaults则不用管这些东西，就像读字符串一样，直接读取就可以了。

UserDefaults支持的数据格式很多，有Int、Float、Double、BOOL、Array和Dictionary，甚至还包括Any类型。

步骤2：在UIAlertAction类的闭包中添加一行代码。

```
let action = UIAlertAction(title: "添加项目", style: .default) {
(action) in
  // 当用户单击添加项目按钮以后要执行的代码
  self.itemArray.append(textField.text!)
  self.defaults.set(self.itemArray, forKey: "ToDoListArray")
  self.tableView.reloadData()
}
```

因为是在闭包之中，所以必须要使用self.表示调用的变量和方法都是在类中声明或创建的。通过set（）方法，将itemArray数组存储到UserDefaults中，与其对应的键名为ToDoListArray。

如果此时构建并运行项目，在添加一个新的事务以后，终止应用再重新开启它，表格中依然只会看到之前的三个事务。这是因为目前在代码中还没有让UserDefaults对象执行保存命令。只有在执行了保存命令以后，通过set（）方法所设置的键/值配对数据才会保存到一个plist格式的文件中。接下来，让我们找到这个文件的位置并看看它的存储格式。

步骤3：在AppDelegate类的didFinishLaunchingWithOptions（）方法中，通过下面的代码可以找出该应用在Mac操作系统中的实际位置。

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {

  print(NSSearchPathForDirectoriesInDomains(.documentDirectory,
.userDomainMask, true).last! as String)

  return true
}
```

iOS系统会为每一个应用程序生成一个私有目录，这个目录位于MacOS系统下iPhone模拟器文件夹内部，并会随机生成一个字符串作为目录名。在每一次应用程序启动时，这个字母数字串都不同于上一次。

我们可以通过上面的代码找到应用程序项目所使用的Documents目录，这个目录通常会作为数据持久化保存的位置。

因为应用是在沙箱（sandbox）中的，在文件读写权限上会受到限制，因此只能在下面几个目录下读写文件：

·Documents：应用中用户数据可以放在这里，iTunes备份和恢复的时候会包括此目录。

·tmp：存放临时文件，iTunes不会备份和恢复此目录，此目录下的文件可能会在应用退出后删除。

·Library/Caches：存放缓存文件，iTunes不会备份此目录，此目录下的文件不会在应用退出后被删除。

构建并运行项目，此时可以在控制台中看到类似下面的信息：

```
/Users/liuming/Library/Developer/CoreSimulator/Devices/7A08E65B-5457-4CB5-AEA6-064CDB120F6A/data/Containers/Data/Application/D2722567-82A1-457C-B57E-7D1D9B9A008F/Documents
```

此时，请再次添加一次拯救世界的事务项目，然后终止应用的运行。

步骤4：在Finder中通过菜单中的前往/前往文件夹...选项直接打开Documents文件夹，然后向上返回一级，也就是进入应用程序所在的文件夹，这里是D2722567-82A1-457C-B57E-7D1D9B9A008F。此时我们会看到四个文件夹：Documents、Library、SystemData和tmp，如图13-20所示。

图13-20　应用程序在自己沙箱中的四个文件夹

因为通过UserDefaults类存储的数据都会保存到Library/Preferences之中，所以我们可以进入该文件夹，你会发现里面有一个类似cn.liuming.TODO.plist的文件。双击打开它以后会发现里面存储着四个事务项目。其中ToDoListArray就是在set（）方法中定义的键名，其内部包含了四个元素，如图13-21所示。

图13-21　应用程序在自己沙箱中的四个文件夹

为什么保存好的四个事务项目并没有出现在表格视图中？这是因为我们在应用启动以后还没有去主动获取UserDefaults的数据。

步骤5：在TodoListViewController类的viewDidLoad（）方法中添加下面的代码。

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

  if let items = defaults.array(forKey: "ToDoListArray") as?
[String] {
    itemArray = items
  }
}
```

如果从UserDefaults对象获取的数据是字符串数组，则通过可选绑定的方式赋值给items，然后在if语句内部将items赋值给itemArray数组。

构建并运行项目，此时我们会看到表格视图中已经出现了四个事务项目，如果你愿意可以再添加一个项目，终止应用程序的运行以后再次将其打开，效果依旧。

现在，我们需要再次提交修改后的项目到仓库。提交信息可以设置为“使用UserDefaults方法将数据保存到本地”。记住勾选Push to remote：origin/master。

13.3.3 UserDefaults说明

作为一名iOS开发者，你可能在项目中经常会用到UserDefaults，因为它使用起来非常简单、灵活，也不用写太多复杂的代码。接下来，让我们来看看UserDefaults还可以做什么。

在Playground中编写下面的这些代码。

```
let defaults = UserDefaults.standard

defaults.set(0.24, forKey: "Volume")

let volume = defaults.float(forKey: "Volume")
```

这里通过UserDefaults存储键名为Volume的单精度型的值0.24，最后再将其从UserDefaults中取出，并赋值给volume变量。

除了可以定义单精度型的值外，我们还可以在UserDefaults中定义布尔型、字符串型和日期型的值，如图13-22所示。

图13-22　在UserDefaults中存储不同基础类型的数据

请注意，我们将一个日期型对象Date（）也存储到了UserDefaults之中，在获取它的时候，必须使用object（）方法，因为UserDefaults在存储它的时候是将它作为Any类型存储的。Any类型实际代表的就是任意的类型。

接下来，让我们看看集合类型的操作。

```
let array = [1,2,3]
defaults.set(array, forKey: "myArray")
let dictionary = ["name": "Happy"]
defaults.set(dictionary, forKey: "myDictionary")
```

```
let myArray = defaults.array(forKey: "myArray")
let myDictionary = defaults.dictionary(forKey: "myDictionary")
```

其中，UserDefaults针对数组和字典有自己单独的方法array（）和dictionary（）。

UserDefaults方式的存储非常灵活、简单，但是它只能存储几千字节的内容和简单的类型，对于大量的数据无法很好地管理，因为它毕竟不是数据库，也不能将它作为数据库来用，它只是一个简单的键/值配对的数据集，并以plist格式存储文件。

如果你想要从UserDefaults中读取一条相关数据，则它会读取整个plist文件，再提供给你指定键的值。如果你的plist文件特别大，那么会花费太多的资源和时间。

13.3.4　Swift中的单例模式

在整个UserDefaults的使用过程中，它的实例化非常特别，是通过UserDefaults.standard实例化一个UserDefaults对象实现的。类似的实例化方法还有URLSession.shared。通过这种方式进行实例化类的方式，我们称之为单例模式。

什么是单例模式？它有什么特别的地方？单例模式的类，在整个应用程序的运行过程中只有一个实例对象，并且它可以通过共享的方式用在不同的类和对象之中。

让我们先创建一个Car类。

```
class Car {
  var colour = "Red"
}

let myCar = Car()
myCar.colour = "Black"

let yourCar = Car()
print(yourCar.colour)
```

在上面的代码中，Car类中有一个属性colour，它的默认值为Red。接下来我们创建了两个Car对象，不管我们如何修改myCar的colour属性值，yourCar的colour属性值都是Red，因为这是两个相互独立的对象。

图13-23　部分单元格在勾选以后发生了错位的情况

接下来让我们在另一个Playground文件中创建一个Car类。

```
class Car {
  var colour = "Red"

  static let singletonCar = Car()
}

let myCar = Car.singletonCar
myCar.colour = "Black"

let yourCar = Car.singletonCar
print(yourCar.colour)
```

从控制台中可以看到所打印的yourCar的colour值为Black，而且不管你通过Car.singletonCar创建多少个常量和变量，都会是同一个拷贝。也就是说，假如我们在某个类中修改了Car的colour属性值，那么其他类中的Car类型对象也会发生变化。

实际上，UserDefaults达到的就是这样的效果，它本身就是一个单例。我们总是通过UserDefaults.standard创建一个实例对象，因此不管是在哪个类中通过该方法存储和读取数据，都会源自同一个plist文件，并且不会发生数据冲突的情况。

13.3.5　创建自定义数据模型

目前我们的应用程序看起来非常漂亮，但是这里面存在着一系列的Bug，只不过你还没有发现。

让我们在itemArray数组里面添加多个项目，例如var itemArray=[“购买水杯”，“吃药”，“修改密码”，“a”，“b”，“c”，“d”，“e”，“f”，“g”，“h”，“i”，“j”，“k”，“l”，“m”，“n”，“o”，“p”]，然后再将viewDidLoad（）方法中读取UserDefaults的数据到itemArray数组的代码注释掉。

此时构建并运行项目，在模拟器中我们可以看到，所列出的项目已经超出了当前屏幕的范围，我们可以通过上下滚动表格视图浏览所有的项目。但是，当我们单击第一个单元格时，在它的右侧会出现一个勾选状态。如果再上下移动表格视图的话，你就会发现之前的勾选发生了错位。不管我们如何调整，总是有错位的情况出现，如图13-23所示。

产生上述Bug的原因在于表格视图单元格的复用。当我们通过dequeueReusableCell（）方法获取一个可复用的单元格对象的时候，表格视图会去查找标识为ToDoItemCell的可以复用的单元格对象并启用它们。这也就意味着，当位于表格视图中第一个位置的单元格被向上滑出表格以后，它就不再可见，而且会被马上移到表格视图的底部，作为一个可以复用的单元格时刻准备从底部再次出现。当它再次出现的时候，之前的勾选状态并未被重置，所以带着之前那个单元格对象的状态又出现在了表格的底部。我们要如何避免这样的情况发生呢?

这就需要我们在复用每一个单元格的时候，针对当前的数据检查它的勾选状态。当前的数据都是通过一个简单的数组提供的，从现在开始显然不能满足我们的需求了，我们需要创建一个全新的数据模型。

实战：创建数据模型。

步骤1：在TODO文件夹中创建一个新的Group，名称为Data Model。在该组中创建一个新的swift文件，名称为Item.swift。

提示　为了很好地区分Model、View和Controller，我们可以再创建一个Controllers、Views和Supporting Files，然后将相关文件拖曳到各组的内部，如图13-24所示。

步骤2：在Item.swift文件中创建一个Item类。

```
import Foundation

class Item {
  var title = ""
  var done = false
}
```

在Item类中，title是字符串类型用于存储事务的名称。done是布尔类型，用于指明是否完成了该事务，这里将它的初始值设置为false，代表该事务没有完成。

图13-24　调整项目的分组结构

步骤3：回到TodoListViewController.swift文件，首先将itemArray属性修改为var itemArray=[Item]（），然后修改viewDidLoad（）方法中的代码，修改后如下面的样子。

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

  let newItem =Item()
  newItem.title = "购买水杯"
  itemArray.append(newItem)
}
```

在该方法中，我们首先创建了一个Item对象，并将title设置为之前的第一个事务，然后将该Item对象添加到itemArray数组之中，现在的itemArray是Item类型的数组。

步骤4：复制viewDidLoad（）方法中的代码，再添加两个事务。

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

  let newItem =Item()
  newItem.title = "购买水杯"
  itemArray.append(newItem)

  let newItem2 =Item()
  newItem2.title = "吃药"
  itemArray.append(newItem2)

  let newItem3 =Item()
  newItem3.title = "修改密码"
  itemArray.append(newItem3)
}
```

因为我们将itemArray从字符串数组修改为itemArray数组，所以接下来有很多的地方需要修改。

步骤5：在cellForRowAt（）方法中将cell.textLabel？.text=itemArray[indexPath.row]修改为cell.textLabel？.text=itemArray[indexPath.row].title，因为通过itemArray[indexPath.row]代码只能获取到Item对象，所以需要借助.title获取事务名称。

步骤6：在addButtonPressed（_sender：UIBarButtonItem）方法中，修改UIAlertAction闭包中的代码。

```
let action = UIAlertAction(title: "添加项目", style: .default) {
(action) in
  // 用户单击添加项目按钮以后要执行的代码
```

```
  let newItem = Item() // 创建Item类型对象
  newItem.title = textField.text!    // 设置title属性

  self.itemArray.append(newItem)     // 将newItem添加到itemArray数组之中
  self.defaults.set(self.itemArray, forKey: "ToDoListArray")
  self.tableView.reloadData()
}
```

步骤7：在didSelectRowAt（）方法中，将用户每一次的操作记录到相应的Item对象的done属性之中，并重新刷新选中的单元格。

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {

  if itemArray[indexPath.row].done == false {
    itemArray[indexPath.row].done = true
  }else {
    itemArray[indexPath.row].done = false
  }

  tableView.beginUpdates()
  tableView.reloadRows(at: [indexPath], with:
UITableViewRowAnimation.none)
  tableView.endUpdates()

  tableView.deselectRow(at: indexPath, animated: true)
}
```

通过indexPath参数，我们可以知道用户单击了哪个单元格，进而设置与单元格位置对应的itemArray数组中的Item对象的done属性。

然后我们通过UITableView类的beginUpdates（）方法告诉表格视图我们想要马上更新某些单元格对象的界面了。endUpdates（）方法则用于告诉表格视图更新单元格的操作结束。在这两个方法之间，我们需要通过UITableView的reloadRows（）方法告诉表格视图需要马上更新的单元格有哪些，更新的时候是否需要动画效果。这里需要更新的单元格是通过IndexPath类型的数组指定的。

步骤8：在cellForRowAt（）方法中，在return cell语句的上面添加下面的代码。

```
if itemArray[indexPath.row].done == false {
  cell.accessoryType = .none
}else {
  cell.accessoryType = .checkmark
}

return cell
```

当表格视图中的单元格需要刷新的时候，根据Item对象的done属性值来设置单元格的勾选状态。

提示　如果你愿意，可以在该方法中添加一个print语句：print（“更新第：\（indexPath.row）行”），我们可以在控制台查看单元格的更新状态。

构建并运行项目，单击单元格以后可以看到修改后的效果。

为了更好地测试多个Item的效果，我们在viewDidLoad（）方法中添加更多的事务。

```
let newItem3 = Item()
newItem3.title = "修改密码"
itemArray.append(newItem3)
// 再向itemArray数组中添加117个newItem
for index in 4...120 {
  let newItem = Item()
  newItem.title = "第\(index)件事务"
  itemArray.append(newItem)
}
```

构建并运行项目，随意单击单元格都不会出现任何的问题，如图13-25所示。

图13-25　由代码生成的事务项目

在之前的cellForRowAt（）方法中，我们使用if语句，根据单元格的accessoryType的属性值设置勾选状态。下面我们使用一种简单的方法来实现该功能。

```
  let item = itemArray[indexPath.row]

  cell.accessoryType = item.done == true ? .checkmark : .none

//    if item.done == false {
//      cell.accessoryType = .none
//    }else {
//      cell.accessoryType = .checkmark
//    }

  return cell
}
```

这里，如果item.done等于true，则会将.checkmark赋值给cell的accessoryType属性，否则会将.none赋值给它。这比起上面注释掉的五行if语句要简单得多，并且更具可读性。

技巧　如果再简化一些的话，可以将item.done==true修改为item.done。也就是说问号前面的值为真则执行冒号前面的值，为假则执行冒号后面的值。

13.3.6 UserDefaults的弊端

接下来，我们需要尝试着使用UserDefaults将Item对象保存到本地磁盘之中。在项目之中，我们在UIAlertAction的闭包中使用self.defaults.set（self.itemArray，forKey："ToDoListArray"）代码将itemArray存储到UserDefaults中。

当你构建并运行项目的时候会发现，当我们添加完一个事务以后，应用程序发生了崩溃。通过控制台打印的日志我们可以发现，User Defaults在试图设置非property-list的对象。

```
2018-02-27 22:44:55.902210+0800 TODO[24678:2928629] [User
Defaults] Attempt to set a non-property-list object (
    "TODO.Item",
    "TODO.Item",
    "TODO.Item",
    "TODO.Item",
```

这也就意味着，我们无法在UserDefaults中存储任意类型的对象。从现在开始我们就需要考虑使用另外一种方法来替代UserDefaults在本地存储数据。因为UserDefaults只适用于小量数据，并且数据类型的限制非常严格。

13.4 认识NSCoder

在之前的学习中，我们使用UserDefaults来存储简单类型的数据。一旦我们创建了Item类型的对象，不仅要存储事务名称，还要记录是否完成该事务。只有通过Item对象，我们才能正确地在表格视图中显示事务的完成状态。

但是问题在于，当我们使用UserDefaults存储自定义类型的对象时，它并不支持这样的存储。本节我们就要使用另外一种方法来解决在磁盘中存储数据的问题。

13.4.1 使用NSCoder编码对象数组

步骤1：需要删除didFinishLaunchingWithOptions（）方法中的print语句。

步骤2：在TodoListViewController类的viewDidLoad（）方法中，创建一个常量存储应用的Document的路径。

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

  let dataFilePath = FileManager.default.urls(for:
.documentDirectory, in: .userDomainMask).first
  print(dataFilePath)
```

其中，FileManager类用于管理应用中的文件系统，并通过default属性获取该类的实例。由此可见，它是一个单例类。在urls（）方法中，我们需要得到document的路径位置，所以这里使用.documentDirectory，注意在自动完成的列表中还有一个.documentation-Directory的枚举值，一定不要选它，这两个文件夹位置是完全不同的。通过urls（）方法我们会得到一个数组，其中第一个元素就是Document的位置。

构建并运行项目，在控制台中会打印类似下面的信息。

```
Optional(file:///Users/liumingl/Library/Developer/CoreSimulator
/Devices/EE243D9-8088-8FB-04E-
564773D5D88/data/Containers/Data/Application/CAA88251-FF23-
4362-02C-E7A4886FEC1/Documents/)
```

在上面的信息中，因为没有拆包可选的操作，所以会显示为Optional（）的形式。在finder中直接导航到Documents的文件夹。

步骤3：删除TodoListViewController类中的UserDefaults变量的声明，然后修改之前的let dataFilePath代码为let dataFilePath=FileManager.default.urls（for：.documentDirectory，in：.userDomainMask）.first？.appendingPathComponent（"Items.plist"）。

通过这样的修改，相当于在URL地址的后面添加了一个文件名，最终地址类似于……902C-4E7A4886FEC1/Documents/Items.plist。如果此时运行项目的话，在Documents文件夹中并不会存在该文件，目前只是生成一个地址而已。

为了可以在类中直接使用dataFilePath地址，我们将dataFilePath调整为ToDoList-ViewController类的一个属性。

```
class TodoListViewController: UITableViewController {

  var itemArray = [Item]()

  let dataFilePath = FileManager.default.urls(for:
.documentDirectory, in:
.userDomainMask).first?.appendingPathComponent("Items.plist")
```

步骤4：在UIAlertAction的闭包中，我们需要借助PropertyListEncoder类对itemArray数组进行编码。

```
let action = UIAlertAction(title: "添加项目", style: .default) {
(action) in
  // 用户单击添加项目按钮以后要执行的代码
  let newItem = Item()
  newItem.title = textField.text!

  self.itemArray.append(newItem)

  let encoder = PropertyListEncoder()

  let data = encoder.encode(self.itemArray)
```

```
  self.tableView.reloadData()
}
```

这里需要创建一个PropertyListEncoder类的实例，然后通过它的encode（）方法将Item类型数组编码为plist格式。此时编译器会报几个错误，让我们依次解决它。

步骤5：因为encode（）方法具有throw功能，所以需要使用do...catch语句。

```
do {
  let data = try encoder.encode(self.itemArray)
  try data.write(to: self.dataFilePath!)
}catch {
  print("编码错误：\(error)")
}
```

在上面的代码中，我们通过write（）方法，将数据存储到指定的路径。

步骤6：为了可以对Item类型的对象编码，还需要让Item类符合Encodable协议。也就是说，要让Item类型能够编码为plist格式或者JSON格式。如果你自定义一个类，它的所有属性必须是标准数据类型，比如字符串、布尔、数组、字典等类型。

```
class Item: Encodable {
```

构建并运行项目，单击+号添加一个新的事务，在表格视图中可以看到新添加的条目。此时可以通过Finder导航到应用的Documents文件夹，可以发现里面出现了Items.plist文件，如图13-26所示。

图13-26　新添加的数据存储到Items.plist文件中

如果你用Items.plsit和之前的Userdefaults.plist文件对比，就会发现UserDefaults文件只能存储极为有限的数据类型，并且第一个根的类型值为Dictionary。

对于事务状态的修改还存在一个Bug：当用户单击单元格以后，勾选状态还没有被存储到Items.plist文件中。我们需要将之前的存储代码拷贝到didSelectRowAt（）方法中。但是，更优雅的方式是添加一个新的saveItems（）方法。

步骤7：在TodoListViewController类的底部，添加saveItems（）方法。

```
func saveItems() {
  let encoder = PropertyListEncoder()

  do {
    let data = try encoder.encode(itemArray)
    try data.write(to: dataFilePath!)
  }catch {
    print("编码错误：\(error)")
  }
}
```

因为不是在闭包中，所以可以删除方法中的self.语句。

步骤8：在UIAlertAction闭包和didSelectRowAt（）方法中调用该方法。

```
let action = UIAlertAction(title: "添加项目", style: .default) {
(action) in
  // 用户单击添加项目按钮以后要执行的代码

  let newItem = Item()
  newItem.title = textField.text!

  self.itemArray.append(newItem)

  self.saveItems()
```

```
  self.tableView.reloadData()
}
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {

  itemArray[indexPath.row].done =
!itemArray[indexPath.row].done

  saveItems()

  tableView.deselectRow(at: indexPath, animated: true)
}
```

13.4.2 使用NSCoder解码

在TodoListViewController类的viewDidLoad（）方法中，我们依然使用着三个测试数据来填充itemArray数组。接下来，我们要实现从磁盘上的Items.plist文件读取之前保存的Item类型的数据。

步骤1：在TodoListViewController类中添加一个新的方法。

```
func loadItems() {
  if let data = try? Data(contentsOf: dataFilePath!) {
    let decoder = PropertyListDecoder()
    do {
      itemArray = try decoder.decode([Item].self, from: data)
    }catch {
      print("解码item错误！")
    }
  }
}
```

通过Data类，我们从Documents文件夹下的Items.plist文件中读取数据。因为Data的初始化方法是throw类型，所以需要使用try命令。又因为其生成的对象是可选类型，所以这里又使用可选绑定将其拆包。如果从Items.plist读出了数据，则会执行if语句体中的代码。

在if语句体中，我们先定义了一个用于解码的PropertyListDecoder对象，然后通过它的decode（）方法将plist格式数据解码为Item数组对象。该方法的第一个参数就是用于指定解码后的数据类型，第二个参数提供解码的数据。

步骤2：在viewDidLoad（）方法中，删除之前手动添加的三个newItem类型的测试数据对象，并调用loadItems（）方法。

```
override func viewDidLoad() {
  super.viewDidLoad()
  print(dataFilePath!)
```

```
  loadItems()
}
```

步骤3：我们还需要让Item类符合Decodable协议，因此将Item的类声明部分修改为：class Item：Encodable，Decodable{。只要类中包含的都是标准数据类型，就可以将其从plist或JSON格式解码为实际的类型。在Swift 4中，我们可以直接将Encodable，Decodable修改为Codable，它代表既符合Encodable，又符合Decodable协议。

构建并运行项目，在随意添加几个事务项目以后退出应用程序，然后重新启动运行，你可以发现此时的TODO记住了之前所有的修改内容，如图13-27所示。

图13-27　通过NSCode存取本地数据

13.5　在应用中使用数据库

到目前为止，不管我们利用UserDefaults还是通过Encoder/Decoder方式，都只是存储和读取简单的键/值配对的数据。从本节开始，我们将学习如何通过关系型数据库来进行复杂的数据存储及搜索。

通过表13-1所示我们先来浏览一下不同的本地数据存储方式。我们已经使用了UserDefaults和Codable方式，这两种方式都是针对微小型数据。而表中的其他三种方式则都利用了数据库或数据库解决方案，它们更适合复杂应用程序。

表13-1　本地数据存储方式

现在，非常多的iPhone应用程序在后台通过数据库在本地存储数据。SQLite是一种简易型数据库，如果你熟悉关系型数据库和SQL语句，就可以使用它来帮你处理大型的数据以及对数据的查询。

Core Data是苹果开发的操作数据的框架，它可以工作在关系数据库之上，并可以将数据库中独立的数据表转换为对象，并使用Swift代码来维护数据库中的数据。这样会比直接使用数据库的原生SQL语句效果更有优势。

最后一个是Realm，它是一个开源框架，是快速、简单的数据库解决方案，而且非常流行。

通过上面的介绍可以清晰地知道，如果你要存储少量的基础数据，可以使用UserDefaults；如果要存储少量自定义对象，可以使用Codable将数据编码为plist格式；如果是大型数据，而你又非常熟悉SQL语言，可以使用SQLite；如果你从一开始就通过Core Data设置数据库，则Core Data是一个非常好的解决方案；如果你需要更快、更简单、更有效果的数据存储解决方案，则可以使用Realm。

13.5.1 设置和配置Core Data

在之前的项目中，我们通过Codable协议将数据存储到plist文件中，并且能够从该文件中获取数据和添加新的项目。我们先将之前的修改提交到远程仓库中。

在接下来的几节中，我们将会使用Core Data实现数据库的CRUD操作，即创建（Create）、读取（Read）、更新（Update）和销毁（Destroy）。

首先，我们需要了解如何在项目中设置和配置Core Data的数据模型。之前在创建应用程序项目的时候，我们并没有勾选Use Core Data选项。因为在实际开发过程中，随着项目的推进可能会存在很多的变数。比如在一开始的时候，项目并不需要Core Data来存储数据，你可能打算使用Codable来处理。但是随着项目的推进，你需要将数据进行排序，这就需要使用Core Data的相关功能。

为了让大家体验更多的设置方式，我们并没有在项目之初就设置Core Data。接下来，将会向大家展示如何为项目添加Core Data。

实战：为TODO项目启用Core Data功能。

步骤1：在Xcode中创建一个新的Single View App项目，将Product Name设置为CoreDataTest，并且勾选Use Core Data选项。

步骤2：在项目导航中打开AppDelegate.swift文件，可以看到文件底部有两个新的方法。

```
// MARK: - Core Data stack

lazy var persistentContainer: NSPersistentContainer = {

  let container = NSPersistentContainer(name: "CoreDataTest")
  container.loadPersistentStores(completionHandler: {
(storeDescription, error) in
```

```
    if let error = error as NSError? {
      fatalError("Unresolved error \(error), \
(error.userInfo)")
    }
  })
  return container
}()

// MARK: - Core Data Saving support
func saveContext () {
  let context = persistentContainer.viewContext
  if context.hasChanges {
    do {
      try context.save()
    } catch {
      let nserror = error as NSError
      fatalError("Unresolved error \(nserror), \
(nserror.userInfo)")
    }
  }
}
```

这两个方法都与Core Data有关，前者是Persistent容器，后者则用于将数据存储到数据库。

步骤3：回到TODO项目，在Xcode菜单中选择File/New/File...，在新文件模板选择面板中选择iOS/Core Data/Data Model类型的文件，如图13-28所示。将新文件的名称设置为DataModel。将新创建的DataModel文件放置在Data Model文件夹中。

步骤4：在项目导航中打开AppDelegate.swift文件，在文件的底部将之前CoreDataTest项目中AppDelegate.swift文件底部的两个Core Data方法拷贝过来，并且将NSPersistent Container（name："CoreDataTest"）修改为NSPersistentContainer（name："DataModel"），其中DataModel就是我们刚刚添加的Core Data数据模型。

图13-28　创建新的Data Model文件

现在我们的TODO项目就如同勾选了Use Core Data选项一样，也具备了Core Data功能。

步骤5：在项目导航中打开DataModel.xcdatamodeld文件。单击编辑区域底部的Add Entity按钮添加一个实体（Entity），如图13-29所示。实体在一定程度上就相当于类，每个实体都会包含一些属性。实体中的属性就相当于类中的属性。你也可以将实体想象为一个表格的数据，一个Excel工作簿中的表。

图13-29　在DataModel中创建一个实体

当我们选中新创建的Entity以后，在右侧的工具区域中可以看到一个新的Data Model Inspector，我们可以对实体中的属性进行详细设置，如图13-30所示。

步骤6：在Data Model Inspector中将Name修改为Item。因为该实体仅会存储之前的Item对象。在Item实体中，我们增加两个属性。单击Attributes部分的+号，设置属性名称为title，类型为String。在选中title属性的情况下，发现Attribute部分中的Optional是勾选状态，我们要取消其勾选状态，让title成为必填项。

步骤7：添加另一个属性，名称为done，类型为Boolean，同样取消Optional勾选。

图13-30　在实体中进行属性设置

步骤8：当前我们已经通过Core Data创建了Item实体，因此项目中就无须使用Item.swift来定义数据模型了。在项目导航中将Item.swift文件删除。

步骤9：回到DataModel，在Item的Data Model Inspector中将Class部分中的Module设置为Current Product Module，如图13-31所示。虽然这一步对项目没有太多实质性影响，但是随着项目的不断推进，你可能会创建很多的实体，如果不设置的话将来可能会出现一些问题。

图13-31　将Item实体的Module属性设置为Current Product Module

另外，在Module属性下方的Codegen属性中，一共有三个选项：Manual/None、Class Definition和Category/Extension，默认的是Class Definition，它会将实体、数据和属性转换为类的形式，我们可以通过类和类的属性来维护它们。神奇的地方在于，Xcode会自动产生这个类，并且不会出现在项目导航中，这种方式是我们使用最多的一种方式。

如果想要查看该实体的情况，在Finder中进入当前用户目录，并找到Library目录。如果没有发现该目录的话，可以在终端中键入chflags nohidden~/Library/命令让其显示到Finder之中。

在类似于/Users/{当前macOS的用户名}/Library/Developer/Xcode/DerivedData/TODO-cnbguupbasafoubovkijzzzcwhxb/Build/Intermediates.noindex/TODO.build/Debug-iphonesimulator/TODO.build/DerivedSources/CoreDataGenerated/DataModel这个位置，我们可以看到有三个文件，如图13-32所示。其中，Item+CoreDataClass和Item+CoreDataProperties这两个文件非常重要。当我们修改实体名称或删除实体的时候会影响到Class文件。当我们修改属性的时候会影响到Properties文件。一般我们不会手动修改这些代码。

图13-32　Item实体的文件位置

如果你选择了Category/Extension，则需要创建一个与实体名称相同的类，Xcode会自动连接它以允许你使用Core Data。

如果你选择了Manual/None，则不会生成相关的类和代码。

作为Core Data的初学者，你会更多地使用Class Definition方式，因为它实现非常的方便。但是也有很多的开发者愿意使用Category/Extension，因为他们可以在类中自定义代码。

打开Item+CoreDataClass.swift文件，发现Item继承于NSManagedObject类。Managed Object是Core Data模型对象，它类似但不是一个标准类，但是我们可以对它进行子类化以管理Core Data数据。如果你创建了自己的自定义类，并选中了Category/Extension，则你的类也需要继承自NSManagedObject类。

```
import Foundation
import CoreData

public class Item: NSManagedObject {
}
```

回到AppDelegate.swift文件中，我们之前在这里粘贴了两个方法，其中，persistentContainer是一个全局变量，saveContext（）是一个方法。

首先在applicationWillTerminate（）方法中调用saveContext（）方法，这样在应用程序退出时可以保存数据库中有改变的数据。

对于persistentContainer变量，它使用了一个我们从未见过的关键字lazy，它是做什么用的呢？当我们以lazy方式声明变量的时候，编译器不会马上创建该变量的实例，而是只有在需要用到它的时候才会去创建。也就相当于当我们试图使用persistentContainer变量时，才会去执行其内部的代码，创建该变量，占用需要的内存空间。

这里我们创建的是NSPersistentContainer类型的变量，它是我们存储所有数据的基础，相当于SQLite数据库。通过NSPersistentContainer类，我们可以使用不同类型的数据库，比如用XML、SQLite。

在声明变量的时候，我们会创建一个NSPersistentContainer类型的常量，并指定之前创建的Core Data模型——DataModel作为它的参数。这样，所有的相关信息都会被载入container常量之中。当使用loadPersistentStores（）方法载入模型后，可以通过完成闭包判定是否成功载入。如果成功，则返回该常量值给persistentContainer这个lazy变量。

对于saveContext（）方法，它提供了存储数据方面的支持，我们只是先在应用终止运行的时候调用它。在该方法中我们定义了一个context，在后面我们会经常看到context，它实际上是一个区域，直到你将临时区域中的数据保存到context之前，我们可以在这个区域里修改和更新数据，也可以执行撤销和重做操作。对比之前的GitHub内容，Context很像是GitHub的临时区域，我们可以在这里修改、更新任何事情，直到Git将修改的内容提交到仓库之中。

对于代码，我们需要了解两件重要的事情：一是创建了persistentContainer变量，它与SQLite数据库一样；二是context，它就是一个临时区域，我们可以在这里修改、删除数据。

13.5.2 如何使用Core Data存储数据

在本节我们将使用Core Data创建和保存事务数据。

在UIAlertAction闭包中，我们通过let newItem=Item（）代码创建了Item对象，但是在启用了Core Data特性后，需要另外一种不同的方法。

```
let action = UIAlertAction(title: "添加项目", style: .default) {
(action) in
  // 用户单击添加项目按钮以后要执行的代码

  let context = (UIApplication.shared.delegate as!
AppDelegate).persistentContainer.viewContext

  let newItem = Item(context: context)

  newItem.title = textField.text!
  self.itemArray.append(newItem)
  self.saveItems()
}
```

此时的Item类是由Core Data自动生成的，所以需要通过Item（context：）初始化方法将类实例化，这里需要Core Data的context值作为参数。

提示　在Item上单击鼠标右键，可以看到此时的Item属于NSManagedObject的子类，是由Core Data负责管理的。

我们在AppDelegate类的saveContext（）方法中见过context，它是persistentContainer中的一个属性。

我们并不需要在TodoListViewController类中创建context属性，通过AppDelegate.persistentContainer.viewContext便可以获取到它。但是AppDelegate只是一个类，并不是对象，此时我们需要的是AppDelegate对象。

再通过UIApplication.shared可以获取到当前正在运行的应用实例，由此可见，UIApplication类也是单例模式。该对象中的delegate属性就是实例化的AppDelegate类型的对象，因为目前它的类型为UIApplicationDelegate，所以还需要再使用（UIApplication.shared.delegate as！AppDelegate）语句将其转换为我们的AppDelegate类的实例。

在saveItems（）方法中，我们需要调用context的save（）方法来存储数据，所以将代码修改为：

```
func saveItems() {
  do {
    let context = (UIApplication.shared.delegate as!
AppDelegate).persistentContainer.viewContext
    try context.save()
  }catch {
    print("保存context错误：\(error)")
  }

  tableView.reloadData()
}
```

因为我们在两个地方都用到了context，所以可以将其设置为类的属性，进而修改两个地方对它的调用。

```
import CoreData

class TodoListViewController: UITableViewController {

  let context = (UIApplication.shared.delegate as!
AppDelegate).persistentContainer.viewContext

  ……
  //MARK: - Add New Items
  @IBAction func addButtonPressed(_ sender: UIBarButtonItem) {
    ……
    let action = UIAlertAction(title: "添加项目", style:
.default) { (action) in
      // 用户单击添加项目按钮以后要执行的代码
```

```
      let newItem = Item(context: self.context)

      newItem.title = textField.text!
      self.itemArray.append(newItem)
      self.saveItems()
    }
    ……
  }

  func saveItems() {
    do {
      try context.save()
    }catch {
      print("保存context错误：\(error)")
    }

    tableView.reloadData()
  }
}
```

构建并运行项目，添加一个新的事务，在控制台中可以看到相关的数据信息。

```
保存context错误：Error Domain=NSCocoaErrorDomain Code=1570 "The
operation couldn’t be completed. (Cocoa error 1570.)"
UserInfo={NSValidationErrorObject=<TODO.Item: 0x61c000097480>
(entity: Item; id: 0x61c000227ec0
<x-coredata:///Item/tD83CE0B3-262C-404B-A94A-DB0D8EE27A8D2> ;
data: {
    done = nil;
    title = "save the world!";
}), NSValidationErrorKey=done, NSLocalizedDescription=The
operation couldn’t be completed. (Cocoa error 1570.)}
```

通过上面的日志我们可以知道，Core Data在保存的时候出现了错误。保存的操作不能正常完成，错误代码为1570，发生错误的键名为done。原来导致保存失败的原因是Item对象的done值为nil。

还记得之前在创建实体的时候，在Data Model Inspector中我们将done属性设置为非可选了吗？这也就代表done属性必须有值存在。所

以在UIAlertAction闭包中再添加一行代码：

```
newItem.title = textField.text!
newItem.done = false  // 让done属性的默认值为false
```

再次构建并运行项目，添加一个新的事务，控制台不再报错。

13.5.3　查看SQLite后端数据库

在默认情况下，Core Data使用SQLite作为后端数据库。这一节我们就来找出它的位置。

在viewDidLoad（）方法中添加一条打印语句：print（FileManager.default.urls（for：.document-Directory，in：.userDomainMask））。

构建并运行项目，找到该应用的Library目录位置，再进入Application Support目录就可以看到DataModel.sqlite文件了，如图13-33所示。接下来就可以利用各种SQLite查看软件将其打开了。

这里推荐大家从Mac App Store上面下载免费版本的Datum来查看数据库。

使用Datum打开SQLite文件以后可以看到类似图13-34所示的这些信息。

图13-33　在Finder中查找DataModel.sqlite文件

图13-34　在Mac App Store上面下载应用

13.5.4　Core Data基础

在实际操作中，我们已经接触了很多的新词，但它们都描述的是同一个概念，如表13-2所示。

表13-2　不同的新词

在程序开发领域中，我们所说的Class对应的就是Core Data中的Entity以及数据库中的Table。在程序中的属性就是数据库中的字段。

对于一张普通的表格来说，这张表就是Core Data中的实体，每一列的名称，比如部门、编码、价格都是表中的字段，在Core Data中就是属性。表中的每一行就相当于一个NSManagedObject对象。虽然本章出现了很多新名词，但是只要记住实体说的就是类或表，属性就是表中的字段，Core Data中的NSManagedObject对象，就是表中单独的一行记录即可。

假设我们的应用一共有Buyers、Products和Orders三个实体，它们都存储在一个永久存储区。这个区域就是persistent container。这个容器中包含了类似SQLite的数据库，以及表与表之间的关系。

在编写程序代码的时候，我们不能直接与persistent容器交互，必须要通过一个中间件，也就是我们之前接触的context。这个context就是一个临时区域，我们可以在这个区域中创建欲添加到实体中的新记录、欲修改的数据或者是想要删除的数据。这也就是之前说的创建（Create）、读取（Read）、修改（Update）和销毁（Destroy）。

需要记住的一点是，所有的CRUD操作都要在context中进行，不能直接操作persistent container。另外，你还可以在context中执行取消（Undo）和重做（Redo）操作。最后在提交的时候我们只需要调用context的save（）方法即可，整个过程与GitHub极为相似，如图13-35所示。

图13-35　应用、Context和Persistent Container之间的关系

就目前的项目来说，为了可以在控制器类中使用Core Data，我们定义了context属性。每个iOS应用程序都有一个UIApplication类型的对象，通过该对象的delegate属性，便可以获取到AppDelegate的实例，接下来再通过AppDelegate实例获取到该类中的persistentContainer属性。注意，persistentContainer是一个lazy变量，这意味着只有在用到该变量的时候，程序才会为我们创建它的实例。在初始化persistentContainer的时候，我们通过参数指定包含实体数据的数据模型。项目中的实体名称为Item，它包含两个属性title和done，然后通过loadPersistentStores（）方法载入DataModel。

在控制器中通过persistentContainer获取viewContext属性以后，就可以操作这个临时区域了。为了可以将一个新的数据添加到实体，我们创建了一个新的Item类型对象let newItem=Item（context：self.context）。Item类是在Data Model编辑器中创建实体的时候，由Core Data自动生成的。通过Item类，我们可以直接访问数据对象的属性，比如title和done。Item对象是NSManagedObject类型，NSManagedObject对象实际上就是实体表中一行独立的记录。

在完成了Item对象的赋值以后，需要调用context的save（）方法将临时区域中的数据存储到persistentContainer中。

13.5.5 从Core Data读取、修改和删除数据

目前我们的项目还存在很多的问题，当应用启动以后表格中并没有任何事务信息，但是在SQLite文件中已经写入了记录。

实战：从Core Data中读取数据。

步骤1：修改TodoListViewController类的loadItems（）方法。

```
func loadItems() {
  let request: NSFetchRequest<Item> = Item.fetchRequest()
}
```

首先创建一个NSFetchRequest类型的常量request，我们通过它获取Item格式的搜索结果。<Item>代表获取到的结果类型是Item类型。Swift在很少的情况下需要程序员指定数据类型，但是在指定了类型以后，会帮助程序员或团队中的其他人理解代码的意思。但是在关键的地方，我们还是必须明确指出某个结果的数据类型。

在声明request的时候，我们必须明确给出实体的数据类型，这代表该请求会得到一批Item类型的对象。

步骤2：继续修改loadItems（）方法。

```
func loadItems() {
  let request: NSFetchRequest<Item> = Item.fetchRequest()

  do {
    itemArray = try context.fetch(request)
  }catch {
    print("从context获取数据错误：\(error)")
  }
}
```

通过context的fetch（）方法，执行上面定义的搜索请求。

步骤3：在viewDidLoad（）方法的最后，添加对loadItems（）方法的调用。

构建并运行项目，在应用启动以后可以看到数据呈现到表格视图之中，如图13-36所示。

图13-36　读取Core Data中的数据

接下来，我们要实现修改Core Data中数据的操作。本项目中修改数据最理想的地方是在didSelectRowAt（）方法里面。

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {

  itemArray[indexPath.row].done =
!itemArray[indexPath.row].done

  let title = itemArray[indexPath.row].title
  itemArray[indexPath.row].setValue(title + " - (已完成)",
forKey: "title")

  saveItems()

  tableView.beginUpdates()
  tableView.reloadRows(at: [indexPath], with:
UITableViewRowAnimation.none)
  tableView.endUpdates()

  tableView.deselectRow(at: indexPath, animated: true)
}
```

当用户单击某个事项以后，会在该事项title的结尾加上-（已完成）字符串。相关的改动只会影响到context区域，直到调用save（）指令前，所有的修改都不会影响到persistentContainer。

目前的代码只是让大家了解如何通过Core Data修改数据，故现在应将新添加的代码注释掉。

另外，如果要删除实体中的某个对象，我们可以利用content的delete（）方法。

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {

  context.delete(itemArray[indexPath.row])
  itemArray.remove(at: indexPath.row)
  ……
}
```

当用户单击事项以后，可以通过delete（）方法直接删除该NSManagedObject对象，然后再从itemArray数组中移除该对象。同样，在执行save（）方法之前，所有的删除操作都是在临时区域实现的。

为了继续后面的学习，我们还是先将和删除相关的两行代码注释掉。

13.6　借助Core Data的查询功能实现搜索

本节我们将利用Core Data搜索指定的Item对象。首先我们需要在表格视图中添加一个搜索栏。

实战：在用户界面中添加搜索栏。

步骤1：打开Main.storyboard文件，从对象库中拖曳Search Bar到表格控制器视图导航栏的下面。从大纲视图中可以确认此时的Search Bar应该位于表格视图的内部、ToDoItemCell的上方，如图13-37所示。

图13-37　在故事板中添加搜索栏

步骤2：修改TodoListViewController类的声明，使其符合UISearchBarDelegate协议。

```
class TodoListViewController: UITableViewController,
UISearchBarDelegate {
```

还记得之前我们往往会在viewDidLoad（）方法中添加对delegate属性赋值的语句吗？这里我们可以使用类似searchBar.delegate=self语句来设置delegate的值。但是还有另外一种方法可以实现对delegate属性的设置。

步骤3：在Main.storyboard中选中searchBar，按住鼠标右键，并拖曳其到上方的黄色图标，该图标代表当前的视图控制器。在弹出的Outlets快捷菜单中选择delegate，这样就相当于将searchBar的delegate属性值设置为当前的控制器对象，如图13-38所示。

图13-38　在故事板中设置搜索栏的delegate属性

同样，我们可以选中表格视图，按住鼠标右键，并拖曳到黄色图标，此时的快捷菜单中会显示DataSource和Delegate均被选中的状态。

另外，通过拖曳Document Outline中的Search Bar到黄色图标，也会有相同的效果。

当前的TodoListViewController类中只是符合UISearchBarDelegate协议，如果该类还要实现照片获取器的功能，则还需要添加UIImagePickerControllerDelegate协议和UINavigationDelegate协议，如果有文本框的话，可能还需要UITextFieldDelegate协议。面对如此多的协议，将会产生很多的委托方法，因此我们可以通过Extension的方式将类按照功能分割一下。

实战：为TodoListViewController类设置Extension。

步骤1：删除之前TodoListViewController类的UISearchBarDelegate协议。

步骤2：在TodoListViewController类的下方，添加一个扩展（extension），然后在扩展中添加一个委托方法。

```
extension TodoListViewController: UISearchBarDelegate {
  func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
  }
}
```

使用扩展的好处在于我们在一个类文件中可以创建多个扩展，每个扩展都与一种协议相关，这样所涉及的委托方法都会相对独立，增加代码的可读性并且便于维护。

步骤3：在searchBarSearchButtonClicked（）方法中添加下面的代码。

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
  let request: NSFetchRequest<Item> = Item.fetchRequest()

  print(searchBar.text!)
}
```

当用户在搜索栏中输入信息并单击虚拟键盘的搜索按键以后，就会调用该方法。这里我们先生成一个对于Item实体的搜索请求，然后再打印搜索栏中输入的文本信息，如图13-39所示。

图13-39　为搜索栏实现搜索请求

步骤4：为了可以搜索到指定内容的Item对象，我们需要添加下面的代码。

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
  let request: NSFetchRequest<Item> = Item.fetchRequest()
  let predicate = NSPredicate(format: "title CONTAINS[c] %@",
searchBar.text!)
  request.predicate = predicate
}
```

这里创建一个NSPredicate类型的对象，format参数代表查询的谓词，即搜索条件。这里会搜索Item实体中title里面包含（CONTAINS）搜索栏里面的字符的记录，[c]代表不区分大小写。其中%@是通配符，它会被第二个参数的值替代。如果搜索栏中的内容是“拯救”，则format参数的字符串就为“title CONTAINS[c]拯救”。最后将过滤谓词添加到request搜索请求之中。

如果大家对于谓词过滤语句还不是很熟悉的话，在GitHub的相关资源中为大家提供了一个谓词相关的文档，大家可以轻松查到符合自己需要的查询语句。

步骤5：继续在方法中添加相关代码。

```
let predicate = NSPredicate(format: "title CONTAINS %@",
searchBar.text!)
request.predicate = predicate

let sortDescriptor = NSSortDescriptor(key: "title", ascending:
true)
request.sortDescriptors = [sortDescriptor]
```

这里会对搜索到的Item对象按照title属性增量排序。

步骤6：最后在方法中添加对Item实体的搜索指令，可以直接复制loadItems（）方法中的代码。

```
request.sortDescriptors = [sortDescriptor]

do {
  itemArray = try context.fetch(request)
}catch {
  print("从context获取数据错误：\(error)")
}

tableView.reloadData()
```

构建并运行项目，使用+号增加足够测试数量的事项，然后通过搜索栏搜索指定的字符串，可以看到需要的结果，如图13-40所示。

图13-40　初步实现搜索栏的功能

步骤7：继续简化之前searchBarSearchButtonClicked（）方法中的代码。

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
  let request: NSFetchRequest<Item> = Item.fetchRequest()

  request.predicate = NSPredicate(format: "title CONTAINS[c]
```

```
%@", searchBar.text!)

  request.sortDescriptors = [NSSortDescriptor(key: "title",
ascending: true)]

  loadItems(with: request)
}
```

同时修改loadItems（）方法为下面这样。

```
func loadItems(with request: NSFetchRequest<Item>) {
  do {
    itemArray = try context.fetch(request)
  }catch {
    print("从context获取数据错误：\(error)")
  }

  tableView.reloadData()
}
```

在定义loadItems（）方法的时候，参数有两个名称，第一个是对外部所显示的名称with，第二个是方法内部调用的时候所使用的名称。这么做的目的是使代码更加优雅和美观。

步骤8：修改viewDidLoad（）方法。

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

print(FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask))

let request: NSFetchRequest = Item.fetchRequest()
loadItems(with: request)
}
```

技巧　为了方便调用，我们还可以为loadItems（）方法的参数添加一个默认值。修改方法的定义为func loadItems（with request：

NSFetchRequest<Item>=Item.fetchRequest（））{。如果在调用的时候不输入参数，该方法就会将搜索请求参数设置为获取全部的Item实体的记录。

通过这样的修改，我们就可以将viewDidLoad（）方法简化为：

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

  print(FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask))

  loadItems()
}
```

构建并运行项目，测试搜索功能是否正常。

接下来，我们需要实现的是当用户搜索事项完成以后，还原到最初的原始事项列表。比如当用户单击搜索栏右侧的叉号按钮时，在清理搜索栏中文字的同时，列出所有的Item事项。需要在TodoListViewController的扩展类中实现该功能。

实战：还原之前的所有事项。

步骤1：在UISearchBarDelegate扩展类中添加新的委托方法。

```
func searchBar(_ searchBar: UISearchBar, textDidChange
searchText: String) {
  if searchBar.text?.count == 0 {
    loadItems()
  }
}
```

一旦搜索栏中的文字内容发生了变化就会调用该方法。在该方法中，会判断搜索栏中的文字数量是否为0，如果为0则代表搜索栏中的文字被用户清空，或者是单击了右侧的叉号按钮后由系统直接清空。

如果搜索文字被清空，就让控制器直接调用loadItems（）方法显示所有的事项。

构建并运行项目，在搜索结束以后单击叉号按钮，让表格视图显示所有的Item实体中的记录，如图13-41所示。

图13-41　还原所有的事项

接下来我们再解决一个小问题，当用户单击叉号按钮以后，希望虚拟键盘可以自动消失。因为此时虚拟键盘若还留在表格视图中已没有任何意义了。要想实现这个功能，我们需要让searchBar停止first responder。

步骤2：在刚才的if语句中再添加一行代码：

一旦用户单击搜索栏以后，searchBar就会成为屏幕上的首要响应对象（First Responder）。如果该对象是带有输入功能的控件，虚拟键盘就会自动从屏幕下方滑出。如果我们取消它的首要响应状态，虚拟键盘会自动消失。

构建并运行项目，在单击叉号按钮以后，虚拟键盘并没有滑出消失，这是为什么呢?

在应用程序运行的时候，通过loadItems（）获取所有Item对象是在后台线程运行的，所以我们不能在这里执行任何与前端用户界面相关的代码。

在模拟器中运行TODO应用的时候，单击Debug控制台的暂停按钮，如图13-42所示。

图13-42　中断应用程序在模拟器中的运行

在Debug导航区域中，我们发现有应用运行了多个线程，它们中的一个是主线程，也就是Thread 1。例如，如果我们通过网络在主线程调用云端bomb数据库，则在Internet上面获取数据的时候，你的应用会处于"冻结"的状态，直到获取数据的操作完成。因此，我们需要将这个任务放在后台线程，也就是其他线程中去处理，如图13-43所示。

图13-43　应用程序运行时候的主线程与后台线程

一旦在后台完成任务，我们可能需要用这些数据来更新主线程中的用户界面。因此在后台线程中需要先获取主线程，然后才能让虚拟键盘消失。

步骤3：修改searchBar（_searchBar：UISearchBar，textDidChange searchText：String）方法为如这样。

```
if searchBar.text?.count == 0 {
  loadItems()

  DispatchQueue.main.async {
    searchBar.resignFirstResponder()
  }
}
```

其中DispatchQueue.main用于获取主线程，async则用于指明主线程和后台线程一起并行执行任务，也就相当于在后台查询记录的同时让搜索栏失去首要响应。

构建并运行项目，单击搜索栏中叉号按钮以后，虚拟键盘消失。

13.7　借助Core Data创建关系图

很多事项处理的应用程序都带有分类，以帮助我们将众多的事项分类成组。要想在TODO项目中实现该功能，需要通过Core Data设置另一个实体，以及在两个实体之间建立关系。

实战：修改用户界面。

步骤1：在Main.storyboard中拖曳一个新的表格视图控制器到导航控制器与Todo-ListViewController之间。删除之前导航控制器与Todo控制器之间的Segue。现在，我们应该让新添加的表格控制器成为导航堆栈中的根控制器。

在导航控制器顶部的黄色图标单击鼠标右键并拖曳到新添加的表格控制器，在弹出的快捷菜单中选择Relationship/root view controller，如图13-44所示。

图13-44　将新添加的表格视图控制器设置为导航的根控制器

新添加的控制器用于在应用启动以后呈现一个事务分类列表，它会带着用户进入该分类中的所有事项。

步骤2：在新的控制器的黄色图标单击鼠标右键并拖曳到Todo控制器，在弹出的快捷菜单中选择Show Segue，将该Segue的Identifier设置为goToItems。

步骤3：在项目导航的Controllers组中添加一个新的Cocoa Touch Class，将Subclass of设置为UITableViewController，将Class设置为CategoryViewController。在保存的时候确认Group选为Controllers，Targets勾选了TODO，如图13-45所示。

图13-45　在项目中创建新的表格视图控制器

步骤4：在Main.storyboard中将新表格控制器的Class设置为CategoryViewController，再选中其单元格，在Attributes Inspector中将Identifier设置为CategoryCell。

步骤5：从对象库拖曳一个Bar Button Item控件到Category控制器的导航栏中，在Attributes Inspector中将System Item设置为Add，将Tine设置为白色。

步骤6：为Add按钮设置IBAction关联，将Name设置为addButtonPressed。

步骤7：单击Category控制器的导航，在Attributes Inspector中将Title设置为TODO，再将Todo控制器的导航标题修改为事项。

因为CategoryViewController是系统生成的，里面包含了很多暂时不需要的委托方法和注释，我们将其整理为下面的样子。

```
import UIKit

class CategoryViewController: UITableViewController {
  override func viewDidLoad() {
    super.viewDidLoad()

  }

  @IBAction func addButtonPressed(_ sender: UIBarButtonItem) {
  }
}
```

接下来，我们需要在DataModel.xcdatamodeld文件中修改数据模型。

实战：修改DataModel.xcdatamodeld的数据模型。

步骤1：在DataModel.xcdatamodeld中除了可以使用表格式编辑界面外，还可以使用图谱式风格，单击界面右下角的图标即可，如图13-

46所示。图谱风格对于经常操作数据库的开发者而言非常的熟悉。基于这种风格，我们将添加一个新的实体。

图13-46　DataModel的两种编辑风格

步骤2：单击Add Entity按钮添加一个新的实体，然后在Data Model Inspector中将Name修改为Category，然后单击Add Attribute按钮为实体添加属性，如图13-47所示。

图13-47　为实体添加属性

步骤3：将第一个属性的Name设置为name，该属性用于存储分类的名称。然后取消Optional的勾选，代表该属性是必填项。将Attribute Type设置为String，代表该属性值只能为字符串类型，如图13-48所示。

步骤4：因为Category中的分类要包含Item中的记录，所以右击Category实体并拖曳到Item实体。当松开鼠标后你会看到13-49所示情形。

图13-48　设置name的属性

图13-49　将Category与Item建立关联

此时在两个实体中会出现关系（Relationships）部分，并且关系名称均为newRelationship，这个名称对于我们来说没有意义。因为每一个Category的记录都会指向多个Item的记录，所以这里我们将Category实体中的关系名称修改为items。另外，因为是指向多个Item，所以在

选中Items关系的情况下，在Data Model Inspector中将Type设置为To Many，如图13-50所示。

图13-50　设置实体关系的属性

步骤5：将Item实体的newRelationship修改为parentCategory，我们会通过它指定Item对象属于哪个分类。因为每个Item对象仅属于一个Category，所以该关系的Type就是默认的To One。

利用Core Data的图谱风格，我们可以方便地创建多个实体，并且可以快速建立实体之间的联系。

现在让我们回到CategoryViewController，首先使用import CoreData导入Core Data框架。

实战：在CategoryViewController类中实现Table View Delegate、Table View Data Source和数据维护方法。

步骤1：在CategoryViewController类中添加下面的属性。

```
class CategoryViewController: UITableViewController {

  var categories = [Category]()

  let context = (UIApplication.shared.delegate as!
AppDelegate).persistentContainer.viewContext
  ……
}
```

在该类中我们创建Category类型的数组categories，为了可以实现Core Data的CRUD功能，从AppDelegate中获取viewContext。

步骤2：在CategoryViewController类中添加TableView的DataSource委托方法。

```
//MARK: - Table View Data Source 方法
override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
  return categories.count
}

override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(withIdentifier:
"CategoryCell", for: indexPath)
  cell.textLabel?.text = categories[indexPath.row].name
  return cell
}
```

步骤3：实现添加Category对象的功能。

```
@IBAction func addButtonPressed(_ sender: UIBarButtonItem) {
  var textField = UITextField()

  let alert = UIAlertController(title: "添加新的类别", message:
"", preferredStyle: .alert)
  let action = UIAlertAction(title: "添加", style: .default) {
(action) in
    let newCategory = Category(context: self.context)
    newCategory.name = textField.text!

    self.categories.append(newCategory)
    self.saveCategories()
  }

  alert.addAction(action)
  alert.addTextField { (field) in
    textField = field
    textField.placeholder = "添加一个新的类别"
  }

  present(alert, animated: true, completion: nil)
}
```

方法中的代码与TodoListViewController类中的极为相似，当用户单击Add按钮以后会在屏幕上呈现一个添加类别对话框，在输入完类别名

称并单击添加按钮以后会将创建的Category对象添加到categories数组中，并将结果保存到persistentContainer容器中。

步骤4：在TodoListViewController类中创建saveCategories（）方法。

```
//MARK: - Table View 数据维护方法

func saveCategories() {
  do {
    try context.save()
  }catch {
    print("保存Category错误: \(error)")
  }
  tableView.reloadData()
}
```

步骤5：继续添加loadCategories（）方法。

```
func loadCategories() {
  let request: NSFetchRequest<Category> =
Category.fetchRequest()
  do {
    categories = try context.fetch(request)
  }catch {
    print("载入Category错误: \(error)")
  }

  tableView.reloadData()
}
```

该方法中的request的类型是NSFetchRequest，代表想要获取的对象都与Category类型相关。

步骤6：在viewDidLoad（）方法的最后添加对loadCategories（）方法的调用。

构建并运行项目，应用启动以后会看到类别列表，只不过目前还没有创建任何的类别。仿照下图的样子，创建四个以上的类别名称。为了测试数据是否被写入persistentContainer，将应用关闭再重新启动，查看是否还会显示所添加的类别，如图13-51所示。

图13-51　添加事务分类到实体中

接下来我们需要实现的是当用户单击购物清单事务以后，屏幕会呈现TodoList控制器的表格视图，并且在表格中列出该类别的所有Item对象。

实战：呈现选中类别的所有事项。

步骤1：在CategoryViewController类中，在MARK：-Table View Delegate注释代码的下方添加didSelectRowAt（）方法。

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
  performSegue(withIdentifier: "goToItems", sender: self)
}
```

当用户单击单元格以后就会调用该方法，你需要借助Segue方法，从Category控制器切换到Todo控制器，Segue的标识为goToItems。

在调用performSegue（）方法之前，我们还需要做一些准备工作，因为现在Todo-ListViewController需要载入的并不是所有的Item对象，而是用户指定类别的Item对象。

步骤2：在didSelectRowAt（）方法的下面添加prepare（）委托方法。

```
override func prepare(for segue: UIStoryboardSegue, sender:
Any?) {
```

```
  let destinationVC = segue.destination as!
TodoListViewController
  if segue.identifier == "goToItems" {

  }
}
```

当调用performSegue（）方法并确定执行标识为goToItems的Segue以后，接下来就是执行该方法了。这里首先获取Segue的目标控制器，使用if语句是防止从Category控制器到Todo控制器有多个Segue，只有执行标识为goToItems的Segue的时候才会执行下面的代码。

步骤3：继续在prepare（）方法中添加下面的代码。

```
if segue.identifier == "goToItems" {
  if let indexPath = tableView.indexPathForSelectedRow {
    destinationVC.selectedCategory = categories[indexPath.row]
  }
}
```

通过tableView的indexPathForSelectedRow我们可以获取用户当前选择的单元格位置，因为它的值是可选的，所以这里需要使用可选绑定将其拆包。然后再将相应的Category对象赋值给TodoListViewController类的selectedCategory属性，只不过目前我们还没有定义该属性。

步骤4：在TodoListViewController类中添加一个属性var selectedCategory：Category？，它的类型是可选的。

需要注意的是，现在在获取Item对象的时候，必须通过selectedCategory属性查找指定类别的事项。

在Swift语言中，我们可以在声明变量的后面添加一对大括号，然后在内部使用didSet{……}关键字加大括号的方式，定义在为

selectedCategory赋值的时候需要做什么事情，在这里我们可以调用loadItems（）方法。

```
var selectedCategory: Category? {
  didSet {
    loadItems()
  }
}
```

因为有了这样的设置，我们可以删除viewDidLoad（）方法中对loadItems（）方法的调用。

步骤5：在UIAlertAction的闭包中，当创建Item对象的时候，我们此时还要为Item的parentCategory赋值，只有这样新创建的Item对象才可能有一个具体的类别。

```
newItem.title = textField.text!
newItem.done = false
// 将selectedCategory的值赋给Item对象的parentCategory关系属性
newItem.parentCategory = self.selectedCategory
```

构建并运行项目，单击进入某个类别以后，你会发现在TodoList控制器中会显示当前所有的事项。实际上我们当前所创建的所有Item对象都没有parentCategory关联，在TodoListViewController的表格视图中，所有的数据都来自于itemArray数组，itemArray数组的数据则来自于loadItems（）方法。在该方法中我们只是简单地通过fetch（）方法获取Item实体中所有的记录。因此，我们需要在查询数据库的时候进行数据过滤。

步骤6：在loadItems（）方法中添加下面的代码。

```
func loadItems(with request: NSFetchRequest<Item> =
Item.fetchRequest()) {
  let predicate = NSPredicate(format: "parentCategory.name
```

```
MATCHES %@", selectedCategory!.name!)
  request.predicate = predicate
  do {
    itemArray = try context.fetch(request)
  }catch {
    print("从context获取数据错误: \(error)")
  }

  tableView.reloadData()
}
```

我们通过NSPredicate创建了一个只获取parentCategory.name完全等于selectedCategory.name的记录，也就是与Item关联的Category对象的name要等于从Category控制器传递过来的数据。

如果此时构建并运行项目的话，在TodoList控制器中，我们根据类别输入几个事项，但是在搜索的时候我们会发现该功能失效了，搜索栏并不能按照我们提供的文字内容去搜索，而只是重新进行了排序。这是因为在searchBarSearchButtonClicked（）方法中，我们通过NSSortDescriptor（）方法对结果进行了排序。

为了可以在loadItems（）方法中只针对selectedCategory提供的类别进行搜索，我们需要在loadItems（）方法中添加第二个参数。

```
func loadItems(with request: NSFetchRequest<Item> =
Item.fetchRequest(), predicate: NSPredicate) {
  ……
```

我们可以创建request请求，还可以创建谓词，这样在用户进行搜索的时候除了当前的request请求以外，还可以设定其他的条件。

步骤7：继续完善loadItems（）方法。

```
func loadItems(with request: NSFetchRequest<Item> =
Item.fetchRequest(), predicate: NSPredicate) {
  let categoryPredicate = NSPredicate(format:
```

```
"parentCategory.name MATCHES %@", selectedCategory!.name!)

  let compoundPredicate =
NSCompoundPredicate(andPredicateWithSubpredicates:
[categoryPredicate, predicate])

  request.predicate = compoundPredicate
  ……
```

在该方法中，我们使用NSCompoundPredicate类的初始化方法将两个甚至多个谓词组合到一起。在当前代码中，我们使用AND逻辑将两个谓词（Predicate）进行连接，也就是筛选出所有谓词都要符合的记录。所以compoundPredicate代表的是在Item实体中找出类别和搜索内容都符合的记录。

此时编译器会报错：TodoList控制器的两个调用loadItems（）的地方缺少predicate参数。因为当前我们为loadItems（）方法的第一个参数设置了默认值，而第二个参数并没有默认值，现在我们进一步完善该方法。

```
func loadItems(with request: NSFetchRequest<Item> =
Item.fetchRequest(), predicate: NSPredicate? = nil) {
  let categoryPredicate = NSPredicate(format:
"parentCategory.name MATCHES %@", selectedCategory!.name!)

  if let addtionalPredicate = predicate {
    request.predicate =
NSCompoundPredicate(andPredicateWithSubpredicates:
[categoryPredicate, addtionalPredicate])
  }else {
    request.predicate = categoryPredicate
  }

  do {
    itemArray = try context.fetch(request)
  }catch {
    print("从context获取数据错误：\(error)")
  }

  tableView.reloadData()
}
```

在该方法中，我们先来判断传递进loadItems（）方法的predicate是否有值，如果有则使用NSCompoundPredicate将两个谓词混合到一起。如果没有则获取指定Category的搜索记录。

构建并运行项目，在某个类别中搜索指定内容，结果正常，如图13-52所示。

图13-52　实现搜索功能

第14章　使用Realm进行本地数据存储

接着上一章的项目代码，我们将使用Realm来替代Core Data对本地数据进行存储。在正式开始之前，我们先提交当前项目代码到GitHub。

如果你之前对Realm还不太了解的话，可以直接访问realm.io，通过主页中相关介绍来学习如何使用Realm。在Realm的官网中提供了很多的说明文档以及代码样例。与SQLite和Core Data有所不同，它是开源项目，所以可以深入了解它是如何实现核心功能的。

我们将要使用的是Realm的Database功能，在https://realm.io/products/realm-database页面中，我们可以选择项目开发的语言，如图14-1所示。当选择了Swift语言以后，Realm就会带我们到Swift开发文档页面。我们可以在该页面中下载Realm for Swift，或者直接从GitHub下载源码。

图14-1　Realm Database的主页

14.1　在项目中集成Realm

实战：通过CocoaPods方式安装Realm。

步骤1：打开终端应用程序，通过命令行方式导航到TODO项目的目录，然后执行下面的命令。

```
pod init
open Podfile -a Xcode
```

步骤2：将Podfile的文件内容修改为：

```
platform :ios, '9.0'

target 'TODO' do
  use_frameworks!

  # Pods for TODO 最新的beta版本为3.2.0-beta.3
  pod 'RealmSwift', '3.2.0-beta.3'

end
```

步骤3：在终端中执行pod install命令。经过一段时间的等待，Realm已经集成到TODO项目之中。打开TODO并编译该项目，你可能会发现有很多的黄色警告图标，而这些警告均出自RealmSwift。重新打开Podfile文件，在最底下一行添加inhibit_all_warnings！，在终端中执行Pod update命令，再重新编译项目以后警告消失。

接下来，我们需要在项目中使用Realm。

步骤4：在AppDelegate中导入Realm。

```
import RealmSwift
```

步骤5：在didFinishLaunchingWithOptions（）方法中，创建一个全新的Realm对象，我们可以把Realm想象为另一个persistentContainer。

```
do {
  let realm = try Realm()
} catch {
  print("初始化Realm发生错误。")
}
```

Realm允许我们通过面向对象的方式来管理本地数据，接下来我们需要创建新的Swift类——Data。

步骤6：在Data Model组中，添加一个新的Swift类型的文件，名称为Data。确保Group为Data Model，Targets为TODO。

步骤7：修改Data.swift文件中的内容如下。

```
import Foundation
import RealmSwift

class Data: Object {
  var name: String = ""
  var age: Int = 0
}
```

对于一般的数据类来说，这已经足够了。但是作为Realm的数据类，仅仅这样是不行的，我们需要在声明变量的前面加上dynamic关键字。简单点说，即dynamic的意思是该变量是动态的。复杂点说，即dynamic是声明修饰符。大家都知道在几年以前，不管是iOS开发还是MacOS开发所使用的语言都是Objective-C。

Objective-C和Swift在底层使用的是两套完全不同的机制，Objective-C对象是基于运行时的，它从骨子里遵循了KVC（Key-Value Coding，通过类似字典的方式存储对象信息）以及动态派发（Dynamic

Dispatch，在运行调用时再决定实际调用的具体实现）。而Swift为了追求性能，使用的是静态派发（Static Dispatch），如果没有特殊需要的话，是不会在运行时再来决定这些的。也就是说，Swift类型的对象或方法在编译时就已经决定，而运行时便不再需要经过一次查找，可以直接使用。

将变量标识为dynamic，就可以允许我们在运行应用的时候监控和改变它。比如，用户可以在应用运行的时候动态改变name这个属性名称，因为Realm在运行的时候可能会根据需要修改数据库结构。

因为dynamic是Objective-C的API，所以在dynamic关键字的前面还要加上@objc关键字。总而言之，加上@objc dynamic两个关键字以后，就可以在应用运行的情况下，监控和动态修改属性的名称了。

步骤8：修改Data类的属性为下面这样。

```
class Data: Object {
  @objc dynamic var name: String = ""
  @objc dynamic var age: Int = 0
}
```

接下来我们需要测试Realm是否可以正常运行。

步骤9：在AppDelegate类的didFinishLaunchingWithOptions（）方法中继续添加代码。

```
let data = Data()
data.name = "乐乐"
data.age = 10

do {
  let realm = try Realm()
  try realm.write {
    realm.add(data)
  }
} catch {
```

```
  print("初始化Realm发生错误。")
}
```

这里我们利用Data类创建了一个Realm对象，然后设置该对象的两个属性值。在创建Realm成功以后，会通过write（）方法将数据对象添加到Realm数据库中。

构建并运行项目，利用Finder导航到应用的Documents目录中，你可以看到Realm生成的default.realm文件，如图14-2所示。

图14-2　应用程序中的realm文件

另外，我们也可以通过Realm.Configuration.defaultConfiguration.fileURL了解Realm文件的具体存储位置。

为了可以查看Realm数据库中的数据，我们可以通过Mac App Store在MacOS中安装Realm Browser，该软件的特点是免费。

在Realm Browser中，我们可以在左侧的Model中找到Data，右侧是Data的两个属性和一条记录，这条记录是我们在启动应用以后被创建的，如图14-3所示。

图14-3　在Realm Browser中查看realm文件中的数据

14.2　使用Realm保存数据

在介绍如何使用Realm保存数据之前，我们先整理一下之前的didFinishLaunching-With-Options（）方法。

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {

  do {
    let realm = try Realm()
  } catch {
    print("初始化Realm发生错误。")
  }

  return true
}
```

在项目导航中删除之前的Data.swift文件，并创建两个新的swift文件：Item.swift和Category.swift，这两个文件的用途是关联Realm数据库中的两个实体。但是，在DataModel.xcdatamodeld文件中，我们已经为Core Data的两个实体创建了内置的Item和Category类。一旦我们在Item.swift文件中声明Item类，编译器就会报错：不能重复声明Item类，如图14-4所示。

图14-4　Xcode中Item的报错

解决的方法非常简单，在DataModel.xcdatamodeld文件中，分别将两个实体的Codegen都设置为Manual/None即可，或者直接删除该文件。

实战：为Realm设置两个实体类。

步骤1：将Item.swift类修改为下面这样。

```
import Foundation
import RealmSwift

class Item: Object {
  @objc dynamic var title: String = ""
  @objc dynamic var done: Bool = false
}
```

步骤2：将Category.swift类修改为下面这样。

```
import Foundation
import RealmSwift

class Category: Object {
  @objc dynamic var name: String = ""
}
```

接下来，我们需要为这两个实体建立关系，就如同之前在Core Data中创建的一样。首先，设置Category指向所拥有的Item对象，然后设置item对象指向Category。

步骤3：修改Category类中的代码。

```
class Category: Object {
  @objc dynamic var name: String = ""

  let items = List<Item>()
}
```

List是RealmSwift中定义的集合类，其就像是数组或字典。使用List将会让Realm的数据库操作更加简单。一旦我们定义了List就需要为List中的数据指定类型，当前我们需要让List包含Item类型的对象。请注意List<Item>只是定义了一种数据类型，代表包含的是Item类型的

List，但是这里并没有将其实例化为对象，所以最后还需要用（）将类型实例化为对象，代表当前初始化的List对象中暂时没有任何的Item对象。

现在，Category类中包含了一个items属性，用于指向当前类别中所包含的所有Item对象的列表。但是在Realm中，所有的反向关系还需要我们手动定义。所以需要在Item类中继续定义与Category类的关系。

步骤4：修改Item类中的代码。

```
class Item: Object {
  @objc dynamic var title: String = ""
  @objc dynamic var done: Bool = false

  var parentCategory = LinkingObjects(fromType: Category.self,
property: "items")
}
```

这里使用LinkingObjects类将Item对象关联到0个、1个或者多个Category类别上，formType参数用于指定关联的Realm类，我们通过Category.self将Item指向Category类。property参数用于指定我们之前在Category中所设置的关系。

接下来，我们需要修复用Realm替代Core Data后所产生的各种报错。

实战：修复Xcode中产生的问题。

步骤1：在Category控制器类中，定义一个Realm对象。

```
import RealmSwift

class CategoryViewController: UITableViewController {

  let realm = try! Realm()
  ……
```

在这里我们使用try！来强制实例化Realm，虽然代码得到了简化，但是危险性极高，强烈建议大家在编写自己项目的时候一定要使用do{……}catch{……}的方法，否则可能会出现一些莫名其妙的问题。

步骤2：在Category控制器类的addButtonPressed（）方法中，修改创建类别的代码。

```
let action = UIAlertAction(title: "添加", style: .default) {
(action) in
  let newCategory = Category()
  newCategory.name = textField.text!

  self.categories.append(newCategory)
  self.saveCategories()
}
```

这里将原来的Category（context：self.context）修改为Category（），因为现在调用的是RealmSwift框架中的类。

步骤3：将saveCategories（）方法修改每一次在保存的时候，我们都需要传递Category对象到save（）方法。

```
func save(category: Category) {
  do {
    try realm.write {
      realm.add(category)
    }
  }catch {
    print("保存Category错误：\(error)")
  }
  tableView.reloadData()
}
```

步骤4：再回到addButtonPressed（）方法中，将self.saveCategories（）修改为self.save（category：newCategory）。

目前为止，我们已经将创建Category的代码修复完成，为了可以测试创建功能是否正常，我们先注释掉其他所有有问题的代码。

构建并运行项目，在应用启动以后添加同之前一样的几个类别，然后在Realm Browser中打开Realm文件，可以看到图14-5所示的信息。

图14-5　查看Realm文件中的Category信息

通过Realm Browser可以发现：当前Category实体中有两个属性，即name是类别的名称，items用于指向每个类别所拥有的item数量，只不过目前还没有任何相关的Item对象。

14.3 使用Realm读取数据

在本节我们将会通过Realm读取需要的数据。

步骤1：修改Category控制器类的loadCategories（）方法为下面这样。

```
func loadCategories() {
  categories = realm.objects(Category.self)

  tableView.reloadData()
}
```

这里我们使用objects（）方法获取指定实体的所有记录。因为它的返回值是Results，所以接下来，我们需要修改在类中声明的categories的类型。

步骤2：将var categories=[Category]（）修改为var categories：Results<Category>？，这代表categories是Results类型，该类型中的结果是Category的对象。注意，categories是可选类型，因为你在第一次运行应用的时候，类别的数量肯定是0。

在这次修改以后，numberOfRowsInSection（）方法会报错，因为此时的categories是可选的，我们需要对其做拆包处理。这里我们使用一个全新的方式，将之前的代码修改为return categories？.count？？1。我们管？？叫空合运算符，简单来说就是当？？前面的值为nil的时候，表达式的值为？？后面的值。如果前面的值不为nil，表达式的值就是前面的值。另外？？前面必须是一个可选值。

对于return categories？.count？？1来说，如果categories为nil，则返回1；不为nil，则返回.count的值。

步骤3：在cellForRowAt（）方法中，将cell.textLabel？.text=categories[indexPath.row].name修改为cell.textLabel？.text=categories？[indexPath.row].name？？“没有任何的类别”。因为考虑到在没有类别的时候，表格视图中指定单元格的数量为1，所以在这种情况下需要设置一个特别的单元格。

步骤4：在prepare（）方法中，修改destinationVC.selectedCategory=categories[indexPath.row]为destinationVC.selectedCategory=categories？[indexPath.row]，该行的报错消失。

步骤5：在addButtonPressed（）方法中，注释掉self.categories.append（newCategory）。

步骤6：最后删除没用的let context=（UIApplication.shared.delegate as！AppDelegate）.persistentContainer.viewContext的属性声明。

为了可以测试在没有类别的时候会显示指定的信息，注释掉viewDidLoad（）方法中的loadCategories（）。

构建并运行项目，可以看到表格中显示的指定信息，尽管当前的categories中没有任何的类别对象，应用也不会发生崩溃。测试完成以后请取消viewDidLoad（）方法中的loadCategories（）的注释，如图14-6所示。

图14-6　测试没有事务的情况

接下来，让我们继续修复TodoList控制器中的问题。

实战：修复TodoList控制器的问题。

步骤1：将import CoreData修改为import RealmSwift。

步骤2：取消var selectedCategory中调用loadItems（）方法的注释。

步骤3：删除类中对context变量的声明。

步骤4：添加let realm=try！Realm（）属性的声明。

步骤5：修改loadItems（）方法为下面这样。

```
func loadItems() {

  itemArray = selectedCategory?.items.sorted(byKeyPath:
"title", ascending: true)

  tableView.reloadData()
}
```

通过selectedCategory的items关系属性可以得到关联的所有Item对象，然后通过sorted（）方法将其按title属性排序。

此时该行会报错：不能将'Results？'类型的值赋给'[Item]'类型。回到类中声明itemArray的地方，将声明修改为var itemArray：Results<Item>？。

接下来为大家介绍Xcode编辑器的一个非常亲民的特性——批量替换变量名称。因为现在itemArray存储的不是数组类型，所以需要将它的名称修改为todoItems。

实战：将itemArray修改为todoItems。

步骤1：选中itemArray然后右击，在快捷菜单中选择Refactor/Rename，如图14-7所示。

图14-7　在Xcode中启用Rename功能

步骤2：在当前位置修改itemArray为todoItems，相关的所有变量名称均会被显式修改，如图14-8所示。

图14-8　在Xcode中显示欲更名的变量

步骤3：如果需要修改注释中的变量，则可以单击名称右边的+号，如图14-9所示。

图14-9　通过Xcode修改变量名称

步骤4：修改numberOfRowsInSection（）方法中的代码为return todoItems？.count？？1。

步骤5：修改cellForRowAt（）方法为下面的样子

```
override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {

  let cell = tableView.dequeueReusableCell(withIdentifier:
"ToDoItemCell", for: indexPath)

  if let item = todoItems?[indexPath.row] {
    cell.textLabel?.text = item.title
    cell.accessoryType = item.done == true ? .checkmark : .none
  }else {
    cell.textLabel?.text = "没有事项"
  }

  return cell
}
```

步骤6：注释掉didSelectRowAt（）方法中的所有代码，我们将会在下一节进行修复。

步骤7：修改UIAlertAction（）闭包中的代码。

```
let action = UIAlertAction(title: "添加项目", style: .default) {
(action) in
  // 用户单击添加项目按钮以后要执行的代码

  if let currentCategory = self.selectedCategory {
    do {
      try self.realm.write {
        let newItem = Item()
        newItem.title = textField.text!
        currentCategory.items.append(newItem)
      }
    }catch {
      print("保存Item发生错误：\(error)")
    }
  }
  self.tableView.reloadData()
}
```

这里先拆包selectedCategory对象，如果有值存在，则创建新的Item对象，并设置它的title属性，以及通过关系属性items将新Item对象添加到关系中。

构建并运行项目，在事务列表中添加几个事项。另外，在Realm Browser中可以看到与购物清单关联的Item一共有3个，如图14-10所示。

图14-10　在添加事项以后查看realm文件

14.4　使用Realm修改和移除数据

本节我们会修改事项的完成状态。

修改TodoList控制器类的didSelectRowAt（）方法。

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {

  if let item = todoItems?[indexPath.row] {
    do {
      try realm.write {
        item.done = !item.done
      }
    }catch {
      print("保存完成状态失败：\(error)")
    }
  }

  tableView.beginUpdates()
  tableView.reloadRows(at: [indexPath], with:
UITableViewRowAnimation.none)
  tableView.endUpdates()

  tableView.deselectRow(at: indexPath, animated: true)
}
```

构建并运行项目，切换事项的完成状态，在Realm Browser中可以看到所发生的变化。如果你仔细观看就会发现一旦切换了状态，Realm马上就会发生变化，速度非常快，如图14-11所示。

图14-11　在应用中修改事项状态以后realm文件变发生了变化

作为测试，如果想要删除Realm中的某个记录的话，可以直接使用Realm类的delete（）方法。例如TodoList控制器类的didSelectRowAt（）方法，在realm.write的闭包中修改代码为下面的样子。

```
try realm.write {
  realm.delete(item)
  //item.done = !item.done
}
```

如果此时构建并运行项目的话，单击某个事项就会删除它。但是此时编译器会崩溃并终止运行，因为我们是针对修改状态编写的代码，后面的reloadRows（）方法就无法更新了，从而导致应用程序运行崩溃。

14.5 使用Realm检索数据

在前面的章节中已经向大家介绍了如何通过Realm来创建、读取、修改和删除（CRUD）数据。本节我们将会了解如何使用Realm检索数据。

步骤1：解除TodoList控制器类的UISearchBarDelegate委托方法的注释。

步骤2：修改searchBarSearchButtonClicked（）方法为下面这样。

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
  todoItems = todoItems?.filter("title CONTAINS[c] %@",
searchBar.text!).sorted(byKeyPath: "title", ascending: false)
}
```

在该方法中，我们将todoItems过滤为title属性只包含搜索栏中输入的文字，并且对检索的数据进行排序。

对于textDidChange（）方法，我们保留原来的代码即可。

步骤3：对于搜索来说，我们更希望将检索到的数据按照创建时间排序，因此需要在Item类中添加一个新的属性。

```
class Item: Object {
  @objc dynamic var title: String = ""
  @objc dynamic var done: Bool = false
  @objc dynamic var dateCreated: Date?  // 用于保存Item对象的创建时间

  var parentCategory = LinkingObjects(fromType: Category.self,
property: "items")
}
```

步骤4：在TodoList控制器类的addButtonPressed（）方法的UIAlertAction的闭包中，添加对dateCreated属性的赋值。

```
try self.realm.write {
  let newItem = Item()
  newItem.title = textField.text!
  newItem.dateCreated = Date() // Date()会返回当前时间
  currentCategory.items.append(newItem)
}
```

步骤5：修改searchBarSearchButtonClicked（）方法中排序的代码，并调用tableView的reloadData（）方法。

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
  todoItems = todoItems?.filter("title CONTAINS[c] %@",
searchBar.text!).sorted(byKeyPath: "dateCreated", ascending:
false)

  tableView.reloadData()
}
```

构建并运行项目，应用程序运行的时候可能会发生崩溃。单击绿色列表图标，可以查看列出的相关解释信息，如图14-12所示。在应用运行之后，我们调整了Item类，添加了dateCreated属性，而此时Xcode不知道在应用运行的时候要如何处理该属性。

图14-12　TODO在应用时报错

最简单的方式是通过Finder导航到Realm文件所在的文件夹，删除与Realm相关的两个文件及一个文件夹。重新构建并运行项目即可。此时的应用中不会有任何的类别与事项，我们需要重新添加相关数据。这时，通过Realm Browser可以在Item类发现新添加的dateCreated属性，如图14-13所示。

图14-13　删除realm文件后再次运行项目

14.6　回顾Realm的操作流程

在本节，我们会一起回顾和梳理在项目中操作Realm的全部流程。在此之前，我们需要先做一下整理工作。

步骤1：在AppDelegate.swift文件中删除对Core Data框架的导入，删除saveContext（）方法和persistentContainer属性。

步骤2：删除applicationWillTerminate（）方法中对saveContext（）方法的调用。

步骤3：在didFinishLaunchingWithOptions（）方法中，我们初始化了一个Realm对象，主要是确保在应用中Realm可以正常运行，但是在AppDelegate类中我们并没有实际用到它，因此可以使用下划线来代替变量名。另外，注释掉之前的print语句。

```
//print(Realm.Configuration.defaultConfiguration.fileURL!)

do {
  _ = try Realm()
} catch {
  print("初始化Realm发生错误。")
}
```

对于两个Realm的实体类来说，Category是我们首个创建的Realm对象类，它是Object的子类，我们能够通过它将数据保存到Realm数据库中。在该类中我们定义了name属性，代表类别的名称。dynamic代表可以在应用运行的时候监控属性中值的变化。另外一个是关系属性items，它指定了每个Category都可以有多个Item对象，它的类型是List。

Item类同样继承于Object，它包含三个属性，同时我们还指定了一个反转关系parentCategory，通过它将每个Item对象与其所属类别连接

起来。在创建反转关系的时候，需要指明连接目标的类以及连接目标（Category类）中的关系属性名称（items）。

在Category控制器类中，我们创建了Realm类型的对象，将categories属性从数组变成了新的集合类型——Results，并且出于安全的考虑将其设置为可选。当我们在viewDidLoad（）方法中载入所有类别的时候，利用了Realm的objects（）方法。在刷新表格视图的时候，如果categories没有值，则让表格只显示一个单元格，否则会依据具体数量显示单元格。我们通过Realm的write（）方法来保存新的类别。

当用户单击事务单元格以后，会执行didSelectRowAt（）委托方法，通过performSegue（）方法执行指定的Segue。在进行控制器切换之前，还会利用prepare（）方法获取目标控制器对象，并将用户选中的事务信息传递到目标控制器。在将category对象赋值给TodoList控制器的selectedCategory属性以后，会执行loadItems（）方法。与loadCategories（）方法类似，这里会获取当前类别中的所有Item对象的连接，只不过这里是按照title属性排序的。

在TodoList控制器类中，todoItems也是Results类型，在与表格相关的委托方法中，我们还是按部就班地执行相关的代码。在添加新事项的时候，我们会在UIAlertAction闭包中创建新的Item对象，为Item对象的属性赋值，让新Item加入Category之中，而这一切都需要在Realm的write（）方法的闭包中完成。

如果用户单击了某个单元格事项，则会切换Item对象的done属性值。

最后是搜索栏的处理，通过searchBarSearchButtonClicked（）方法来搜索特定文字内容的Item对象。这里通过filter（）方法，并使用类似于SQL语句的字符串参数，搜索title属性中包含指定字符串的对象，并对结果按创建时间排序。当用户结束搜索以后，通过textDidChange（）方法，我们在结束搜索以后重置todoItems对象。

14.7　让单元格可以滑动

在进行接下来的操作之前，让我们先提交代码到GitHub，可以设置提交信息为：使用Realm存储本地数据。

在此之前，我们用了很长的时间去解决如何在后台处理我们的数据。从本节开始，我们将会逐步完善前端设计，不断增强应用的用户体验。因为目前的应用看起来还是相当幼稚的，第一个需要实现的功能就是删除一个不需要的事务。

删除事务的操作可以仿照邮件应用，如果我们在某一封邮件的单元格上面向左滑动手指，就可以看到相应的选项。如果再继续往左滑动，就可以直接将邮件删除。要想实现这一功能，可以借助CocoaPods。

在CocoaPods主页中搜索SwipeCellKit，通过该项目的GitHub连接，我们可以看到它包含了各种风格的过渡效果，如图14-14所示。

图14-14　SwipeCellKit介绍

实战：在项目中安装SwipeCellKit。

步骤1：关闭现有的TODO项目，在终端应用中导航到TODO项目目录，通过open Podfile-a Xcode命令编辑Podfile文件。

步骤2：将文件修改为下面这样。

```
platform :ios, '10.0'

target 'TODO' do
  use_frameworks!

  # Pods for TODO
  pod 'RealmSwift', '3.2.0-beta.3'
```

```
  pod 'SwipeCellKit'
end

inhibit_all_warnings!
```

步骤3：在终端中执行pod install命令。

接下来，我们会实现滑动单元格的功能。

实战：实现用户的滑动操作交互功能。

步骤1：在Category控制器类中导入SwipeCellKit框架import SwipeCellKit。

步骤2：在cellForRowAt（）方法中，将获取的可复用单元格转换为SwipeTableViewCell类型，并将cell的delegate属性设置为self。

```
let cell = tableView.dequeueReusableCell(withIdentifier:
"CategoryCell", for: indexPath) as! SwipeTableViewCell
cell.delegate = self
```

步骤3：在Category控制器类的最后添加一个extension。

```
// MARK: - Swipe Cell Delegate Methods

extension CategoryViewController: SwipeTableViewCellDelegate {
  func tableView(_ tableView: UITableView, editActionsForRowAt
indexPath: IndexPath, for orientation: SwipeActionsOrientation)
-> [SwipeAction]? {
    guard orientation == .right else { return nil }

    let deleteAction = SwipeAction(style: .destructive, title:
"删除") { action, indexPath in
      // handle action by updating model with deletion
      print("类别被删除")
    }

    // customize the action appearance
```

```
    deleteAction.image = UIImage(named: "delete")

    return [deleteAction]
  }
}
```

你可以直接从GitHub中复制editActionsForRowAt（）方法的所有代码到当前的扩展中，当用户用手指在单元格上滑动的时候会触发该方法。

在方法的内部，只接受手指从右侧滑动的操作，否则会返回nil。如果用户单击了title为删除的按钮，则会执行闭包中的代码，目前闭包中只有一行注释语句。另外，在方法中还定义了删除按钮的图标为delete，但是现在在该项目中还没有该图标。

通过提供的资料文件找到Trash-Icon.png文件，并将其添加到项目的Assets.xcassets文件之中，然后修改deleteAction.image=UIImage（named："delete"）代码为deleteAction.image=UIImage（named："Trash-Icon"）。

如果此时构建并运行项目的话，应用会崩溃，控制台会打印类似下面的信息。

```
Could not cast value of type 'UITableViewCell' (0x1063f7038) to
'SwipeCellKit.SwipeTableViewCell' (0x103d5e650).
```

之所以出现上述情况，是因为Xcode不能将UITableViewCell类型的对象转换为SwipeTableViewCell类型。

步骤4：在Main.storyboard中选中Category控制器表格视图里面的单元格，在Identifier Inspector中，将Class设置为SwipeTableViewCell，将Module设置为SwipeCellKit，因为该单元格类来自于SwipeCellKit框架，如图14-15所示。

图14-15　设置事务单元格的属性

构建并运行项目，在Category控制器视图中从右向左拖动鼠标，会出现图14-16所示的效果。

当前的问题在于单元格的高度不够，“删除”图标显示不完全。

步骤5：在viewDidLoad（）方法中添加一行代码tableView.rowHeight=80.0。

再次构建并运行项目，可以看到完整的图标，如图14-17所示，如果单击删除按钮的话，在控制台会看到通过print语句打印的相关信息。

图14-16　测试SwipeCellKit的执行效果

图14-17　正确的运行效果

接下来，我们需要在editActionsForRowAt（）方法里面的SwipeAction（）的闭包中实现类别删除的代码。

```
let deleteAction = SwipeAction(style: .destructive, title: "删除") { action, indexPath in
  if let categoryForDeletion = self.categories?[indexPath.row]
{
    do {
        try self.realm.write {
        self.realm.delete(categoryForDeletion)
      }
    }catch {
      print("删除类别失败：\(error)")
    }

    tableView.reloadData()
  }
}
```

在闭包中，先拆包categories中的指定事务，如果该值存在则将其删除，最后再刷新表格视图。

构建并运行项目，可以再创建一个新的事务，然后再将其删除，之后运行正常。

在邮件程序中，我们可以通过从右向左滑动直接删除邮件，但是在目前的TODO项目中，我们只能先滑动，再单击删除按钮来完成相同的任务。接下来，我们就在TODO项目中实现该功能。

在GitHub中SwipeCellKit的主页面上找到Usage部分中editActions-Options-For-Row-At（）方法的定义。这是一个可选方法，用于实现对单元格的一些自定义的操作。

将下面的方法复制到editActionsForRowAt（）方法的下面，将方法中的transitionStyle一行代码注释掉。

```
func tableView(_ tableView: UITableView,
editActionsOptionsForRowAt indexPath: IndexPath, for
orientation: SwipeActionsOrientation) -> SwipeTableOptions {
    var options = SwipeTableOptions()
    options.expansionStyle = .destructive
    // options.transitionStyle = .border
    return options
}
```

在方法中可以为options设置扩展风格（Expansion Style）和过渡风格（Transition Styles），如图14-18所示。这里我们将会使用扩展的Destructive风格。

图14-18　SwipeCellKit中的扩展风格

最后，我们直接将editActionsForRowAt（）方法中的tableView.reloadData（）代码删除就可以了，因为删除单元格的操作已经由editActionsOptionsForRowAt（）方法替我们完成了。

如果对于TodoList控制器类来说，要想实现删除Item对象的功能，同样需要上面的操作。你可以自己对比Category控制器类先尝试着为TodoList控制器类添加滑动单元格的功能，然后再阅读下面的操作步骤。

实战：为TodoList控制器添加单元格的滑动删除功能。

步骤1：在TodoList控制器类中导入import SwipeCellKit框架。

步骤2：在viewDidLoad（）方法中设置单元格的高度tableView.rowHeight=80.0。

步骤3：在cellForRowAt（）方法中将cell强制转换为SwipeTableViewCell类型，并且设置其delegate属性值。

```
let cell = tableView.dequeueReusableCell(withIdentifier:
"ToDoItemCell", for: indexPath) as! SwipeTableViewCell
cell.delegate = self
```

步骤4：将Category控制器类关于SwipeTableViewCellDelegate协议的扩展代码全部复制到TodoList控制器中，并修改为下面的样子。

```
// MARK: - Swipe Cell Delegate Methods
extension TodoListViewController: SwipeTableViewCellDelegate {
  func tableView(_ tableView: UITableView, editActionsForRowAt
indexPath: IndexPath, for orientation: SwipeActionsOrientation)
-> [SwipeAction]? {
    guard orientation == .right else { return nil }

    let deleteAction = SwipeAction(style: .destructive, title:
"删除") { action, indexPath in
      // 通过todoItems获取到用户将会删除的Item对象
```

```
      if let itemForDeletion = self.todoItems?[indexPath.row] {
        do {
          try self.realm.write {
            self.realm.delete(itemForDeletion) // 删除Item对象
          }
        }catch {
          print("删除事项失败：\(error)")
        }
      }
    }

    // 自定义单元格在用户滑动后所呈现的外观
    deleteAction.image = UIImage(named: "Trash-Icon")

    return [deleteAction]
  }

  func tableView(_ tableView: UITableView,
editActionsOptionsForRowAt indexPath: IndexPath, for
orientation: SwipeActionsOrientation) -> SwipeTableOptions {
    var options = SwipeTableOptions()
    options.expansionStyle = .destructive
    return options
  }
}
```

步骤5：在Main.storyboard文件中，将TodoList控制器视图中的单元格的Class设置为SwipeTableViewCell，将Module设置为SwipeCellKit。

构建并运行项目，如果愿意的话还可以打开Realm Browser，观察是否可以成功删除指定的事项。

14.8　让App的界面更加丰富多彩

目前，TODO的颜色还是比较单调的，没有人会喜欢这样一款颜色朴素的应用程序。接下来我们会借助第三方开源类，为项目添加丰富多彩的颜色，让它显得更加生动活泼。

在之前的项目中我们曾使用过Chameleon框架。通过该框架，我们为应用程序界面设置了漂亮的颜色。在本节中我们将会使用该框架的一些其他的功能。

充分利用这些第三方类库，可以帮助我们快速完成项目的开发，而不用所有的代码都自己来亲自编写，这样可以使开发更有效率。很多程序员也会做出非常漂亮、实用的东西，并乐于将它们作为第三方类库分享到GitHub。而我们只需要通过非常少的代码去直接调用它们，何乐而不为呢?

在Chameleon主页面上找到Swift 3的CocoaPods的安装代码，并将其复制到项目的Podfile文件中。

```
  # Pods for TODO
  pod 'RealmSwift', '3.2.0-beta.3'
  pod 'SwipeCellKit'
  pod 'ChameleonFramework/Swift', :git =>
'https://github.com/ViccAlexander/ Chameleon.git'
```

在终端应用程序中执行pod install命令。

在安装完成以后，打开项目并使用Control+B重新构建项目，确保没有任何报错。

在chameleon框架的文档中找到UIColor Methods的链接，在里面可以看到如何设置一个随机颜色。

实战：为表格设置随机颜色。

步骤1：在Category控制器类中导入框架import Chameleon-Framework。

步骤2：在cellForRowAt（）方法中，添加一行代码：cell.backgroundColor=UIColor.randomFlat。

构建并运行项目，可以看到图14-19所示的效果。

图14-19　为TODO添加颜色

在目前的表格视图中，每个单元格之间都会有一条分割线，在有单元格背景颜色的情况下，我们可以将其取消。

步骤3：在Category控制器类的viewDidLoad（）方法中，添加一行tableView.separatorStyle=.none代码。

目前，在Category控制器中显示的类别单元格的背景色都是随机分配的，这也就意味着一旦我们关闭TODO再重新打开它的时候，颜色就会发生变化。如何固定住每个事务单元格的背景颜色呢？我们将会把与事务关联的颜色作为Category实体的属性。

可能你会想到在Category类中添加一个新的colour属性，属性的类型为UIColor，如下面这样。

```
class Category: Object {
  @objc dynamic var name: String = ""
  @objc dynamic var colour: UIColor = UIColor()

  let items = List<Item>()
}
```

在该类中我们添加一个colour属性，并将其类型设置为UIColor。这样的话，在运行的时候会报如下的错误：

```
*** Terminating app due to uncaught exception 'RLMException',
reason: 'Property 'colour' is declared as 'UIColor', which is
not a supported RLMObject property type. All properties must be
primitives, NSString, NSDate, NSData, NSNumber, RLMArray,
RLMLinkingObjects, or subclasses of RLMObject. See
https://realm.io/docs/objc/latest/api/Classes/RLMObject.html
for more information.'
```

上述错误的意思是：Realm数据库只能保存基本类型、NSString、NSDate、NSData、NSNumber、RLMArray、RLMLinkingObjects或者是RLMObject的子类对象。UIColor不能作为RLMObject的属性来进行存储。因此，我们只能使用String类型来存储颜色值。

实战：使用Realm存储文字的颜色信息。

步骤1：在Category类中添加一个新的colour属性，属性的类型为String。

```
class Category: Object {
  @objc dynamic var name: String = ""
  @objc dynamic var colour: String = ""

  let items = List<Item>()
}
```

步骤2：在Category控制器类创建Category对象的UIAlertAction闭包中，修改代码为下面的样子。

```
let action = UIAlertAction(title: "添加", style: .default) {
(action) in
  let newCategory = Category()
  newCategory.name = textField.text!
  newCategory.colour = UIColor.randomFlat.hexValue()
```

```
  self.save(category: newCategory)
}
```

这里通过hexValue（）方法获取颜色的hex值，它是字符串类型的值。

步骤3：在cellForRowAt（）方法中，添加一行代码：cell.backgroundColor=UIColor（hexString：categories?[indexPath.row].colour？？"1D9BF6"）。

这样设置单元格的背景色会从Realm的数据表中读取，如果无法获取到Category对象，则会直接设置背景色为1D9BF6。

构建并运行项目，然后退出再重新进入，类别的颜色不会发生变化。

对于TodoList控制器来说，我们希望在表格中创建单元格的背景色渐变的效果。用蓝色来举例，第一个单元格的颜色为浅浅的蓝色，第二个为浅蓝色，第三个为蓝色，第四个为深一点儿的蓝色……以此类推。通过Chameleon可以为我们实现这个效果。

在ChameleonFramework中包含了24种不同颜色的亮暗素材，我们可以在GitHub主页查询到，如图14-20所示。

图14-20　Chameleon提供的24种渐变色

实战：为事项单元格设置渐变色。

步骤1：在TodoList控制器类的cellForRowAt（）方法中添加一行代码。

```
if let item = todoItems?[indexPath.row] {
  cell.textLabel?.text = item.title
  cell.accessoryType = item.done == true ? .checkmark : .none
```

```
  // 设置单元格的背景色
  cell.backgroundColor = FlatSkyBlue().darken(byPercentage:
CGFloat(indexPath.row / todoItems?.count))

}else {
  cell.textLabel?.text = "没有事项"
}
```

通过FlatSkyBlue我们可以获取天空蓝的颜色，再通过它的darken（）方法可以自定义这个天空蓝的阴暗程度，0为最浅，1为最深。所以在byPercentage参数中，我们通过indexPath.row/todoItems？.count表达式来计算出阴暗度，以5个事项为例，第一个阴暗度为0，然后是0.2、0.4、0.6和0.8。

但是此时编译器会报错，因为todoItems是可选类型，所以todoItems？.count的结果就是可选整型。整个的计算结果就是可选单精度。这里需要利用可选绑定进行修正。

步骤2：将上面的方法进一步修改为下面这样。

```
if let colour = FlatSkyBlue().darken(byPercentage:
CGFloat(indexPath.row / todoItems!.count)) {
  cell.backgroundColor = colour
}
```

修改后使用todoItems！.count将其强制拆包，这样做很安全，因为是通过if语句对其进行了拆包的。

步骤3：在viewDidLoad（）方法中，添加一行tableView.separatorStyle=.none代码，删除单元格之间的分割线。

构建并运行项目，在TodoList表格中添加一些事项，效果如图14-21所示。

目前，事项中所有的单元格都是同一种颜色，并没有按照我们想象的效果呈现。问题出现在CGFloat（indexPath.row/todoItems！.count）一句。虽然我们将计算的结果强制转换为单精度，但是在Swift语言中，整型值（indexPath.row的值）除以整型值（todoItems！.count的值）的结果还是整型值，这也就意味着1/5、2/5、3/5……的值都是0，Swift会将结果小数点后面的值去掉，返回一个整型，即便最后将其转换为单精度，结果也是0.0。

因此，我们需要将byPercentage参数的代码修改为：byPercentage：CGFloat（indexPath.row）/CGFloat（todoItems！.count），这样生成的才是有效的参数值。

构建并运行项目，可以看到效果如图14-22所示。

图14-21　单元格中生成的颜色

图14-22　单元格中生成的渐变色

这里还有一个问题，越来越深的天空蓝色使得我们根本无法看到下面几个单元格的文字内容，其实ChameleonFramework已经为我们提供了相应的解决方案。在GitHub中ChameleonFramework的主页面里找到对比文本（Contrasting Text）部分，如图14-23所示。

图14-23　Chameleon的对比文本

简单来说，借助Chameleon我们可以让文字从背景颜色中脱颖而出。

根据Chameleon算法，利用对比颜色的特征我们可以得到最好的对比色，从而让文字在背景色中突出出来。我们只需要提供背景色的颜色，就可以得到适合的文本颜色。

步骤4：在上面的设置单元格的代码中添加下面的一行代码。

```
if let colour = FlatSkyBlue().darken(byPercentage:
CGFloat(indexPath.row) / CGFloat(todoItems!.count)) {
  cell.backgroundColor = colour
  cell.textLabel?.textColor = ContrastColorOf(colour,
returnFlat: true)
}
```

通过ContrastColorOf（）方法，我们可以得到以colour为背景色的最为合适的文本颜色或者是对比色，returnFlat参数用于确定是否为平涂。

构建并运行项目，可以看到图14-24所示的效果。

在当前情况下，所有的文字颜色都是白色，如果我们的背景色是黑白渐变的话，文字颜色也会随着变化。如果你愿意，可以将let colour=FlatSkyBlue（）修改为let colour=FlatWhite（），运行效果如图14-25所示。

为了和之前Category控制器中事务单元格的颜色一致，接下来我们将单元格的颜色修改为与事务单元格一致的颜色。

将let colour=FlatSkyBlue（）代码修改为从Category控制器类传递过来的Category对象的颜色值：let colour=UIColor（hexString：selectedCategory！.colour）？。因为selectedCategory是可选的，所以这里使用！将其强制拆包。又因为之前通过if语句将item拆包，所以可以确保selectedCategory值不会为nil。最后的问号则会通过当前的if语句进行拆包，所以这句代码不会发生问题。

图14-24　最终的对比文本效果

图14-25　不同背景的不同对比文本效果

构建并运行项目，可以看到效果如图14-26所示。

图14-26　TODO项目的运行效果

因为购物清单单元格的背景色为红色，所以在事项表格中的单元格背景色就是对应的红色由亮到暗。

14.9　调整导航栏的UI

我们之前做了大量的工作来美化应用的用户界面，在本章的最后一节，我们将会修整一些东西，让TODO看起来更像是一款可以上架到App Store上的真正应用。

首先是导航栏的界面，我们希望它能够大一些，在Main.storyboard文件中，选中Navigation Bar，在Attributes Inspector中勾选Navigation Bar部分中的Perfers Large Titles选项。另外，还需要将Large Title Text Attributes部分中的Title Color设置为白色。

构建并运行项目，可以看到图14-27所示的效果。

图14-27　TODO的导航栏设置效果

接下来，我们将TodoList控制器的导航栏调整为和Category单元格一样的颜色。

修改TodoList控制器类的viewDidLoad（）方法为下面的样子。

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

  if let colourHex = selectedCategory?.colour {

    guard let navBar = navigationController?.navigationBar else
{
      fatalError("导航栏不存在！")
    }

    navigationController?.navigationBar.barTintColor =
UIColor(hexString: colourHex)
  }

  tableView.rowHeight = 80.0
```

```
  tableView.separatorStyle = .none
}
```

通过navigationBar.barTintColor我们可以修改当前控制器的导航栏颜色。另外，因为navigationController是可选的，所以我们先利用guard语句判断navigationController是否存在。

构建并运行项目，当进入事项界面后应用程序崩溃，如图14-28所示。

图14-28　导航栏不存在崩溃

问题的原因就在于导航栏不存在。

对于初学者来说，应用程序在运行时发生崩溃是件非常可怕的事情，但同时它也是一件好事，我们可以发现代码中潜在的Bug，所以千万不要着急和烦恼，根据提示解决问题就好。

对于控制器类的viewDidLoad（）方法来说，它会在当前控制器内部的UI控件全部被载入以后被调用。但是，我们的TodoList控制器是在导航控制器的控制器堆栈之中，导航栏并不属于TodoList控制器的UI控件，所以这个时候navigationController还是nil。

要想解决这个问题，我们需要在控制器类中添加另外一个方法——viewWillAppear（），该方法会在所有视图控件完全都准备好，即将呈现到屏幕上的时候被调用。此时的navigationController的值就不会是nil了。

将之前的代码剪切到viewWillAppear（）方法中。

```
override func viewWillAppear(_ animated: Bool) {
  if let colourHex = selectedCategory?.colour {
    title = selectedCategory!.name
```

```
    guard let navBar = navigationController?.navigationBar else
{
      fatalError("导航栏不存在！")
    }

    navBar.barTintColor = UIColor(hexString: colourHex)
  }
}
```

这里，我们还设置了title属性为类别的名称，title属性代表的是导航栏中的标题，之前默认值为事项，现在让它显示事务的名称。因为之前已经对selectedCategory拆包，所以直接使用强制拆包即可。

构建并运行项目，运行效果如图14-29所示。

图14-29　根据事务颜色设置导航栏颜色

接下来，我们还需要调整搜索栏的颜色。

首先为搜索栏与TodoList控制器类建立IBOutlet关联，将name设置为searchBar。

在viewWillAppear（）方法中添加对搜索栏颜色设置的代码。

```
override func viewWillAppear(_ animated: Bool) {
  if let colourHex = selectedCategory?.colour {
    title = selectedCategory!.name
    guard let navBar = navigationController?.navigationBar else
{
      fatalError("导航栏不存在！")
    }

    if let navBarColor = UIColor(hexString: colourHex) {
      navBar.barTintColor = navBarColor
      navBar.tintColor = ContrastColorOf(navBarColor,
returnFlat: true)
      searchBar.barTintColor = navBarColor
    }
```

```
    }
}
```

因为UIColor（hexString：colourHex）生成的是可选值，所以这里将其拆包，再针对导航栏的颜色以及搜索栏的颜色进行赋值。另外，我们还通过navBar.tintColor设置了导航栏中按钮文字的颜色，这里使用对比色。

构建并运行项目，运行效果如图14-30所示。

接下来，我们还要修改导航栏中标题的颜色，因为在故事板中将导航栏设置为Large，所以需要通过largeTitleTextAttributes属性进行设置。继续在viewWillAppear（）方法中添加代码。

```
if let navBarColor = UIColor(hexString: colourHex) {
  navBar.barTintColor = navBarColor
  navBar.tintColor = ContrastColorOf(navBarColor, returnFlat:
true)
  // largeTitleTextAttributes是字典类型
  navBar.largeTitleTextAttributes =
[NSAttributedStringKey.foregroundColor:
ContrastColorOf(navBarColor, returnFlat: true)]

  searchBar.barTintColor = navBarColor
}
```

因为largeTitleTextAttributes是字典类型，所以需要通过某个键名来设置其属性。iOS系统中所有用于字符串外观风格的键名都整理到NSAttributedStringKey结构体之中，这里我们需要设置的是字符前景色（foregroundColor），并将其颜色设置为导航栏颜色的对比色。

最后，我们需要在故事板中将之前针对导航栏中的+号按钮所设置的颜色，从白色修改为default，这样在控制器类中才会通过代码来修改其颜色。

构建并运行项目，运行效果如图14-31所示。

图14-30　设置搜索栏的颜色

图14-31　设置导航栏各个元素的颜色

在viewWillAppear（）方法中，我们既使用了if语句又使用了guard语句，那么到底在什么情况下使用if或guard呢？其实并没有什么明确的划分，if语句包含为真时候的语句体、为假时候的语句体，以及其他情况的语句体。而guard则只有在为假的时候才执行其语句体，否则会向下继续执行。

根据笔者的经验，一般情况下还是用if语句比较好。如果在判断的时候，条件为假后会严重影响到应用的运行，则使用guard抛出致命错误的提示。

目前，一旦进入到TodoList再返回到Category控制器，导航栏的颜色风格还是保持着修改后的样子。我们需要让它回到默认的颜色风格，所以要在TodoList控制器中添加viewWillDisappear（）方法，该方法会在控制器视图即将从屏幕上消失时被调用。

```
override func viewWillDisappear(_ animated: Bool) {
  guard let originalcolour = UIColor(hexString: "1D9BF6" ) else
{ fatalError() }

  navigationController?.navigationBar.barTintColor =
originalcolour
  navigationController?.navigationBar.tintColor = FlatWhite()
  navigationController?.navigationBar.largeTitleTextAttributes
= [NSAttributedStringKey.foregroundColor: FlatWhite()]
}
```

最后，我们还要设置Category控制器表格中文字的对比色，修改cellForRowAt（）方法。

```
override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(withIdentifier:
"CategoryCell", for: indexPath) as! SwipeTableViewCell
  cell.delegate = self
  cell.textLabel?.text = categories?[indexPath.row].name ?? "没
有任何的类别"

  guard let categoryColour = UIColor(hexString: categories?
[indexPath.row].colour ?? "1D9BF6") else { fatalError() }

  cell.textLabel?.textColor = ContrastColorOf(categoryColour,
returnFlat: true)
  cell.backgroundColor = categoryColour

  return cell
}
```

构建并运行项目，运行效果如图14-32所示。

图14-32　TODO应用的最终运行效果

第15章　机器学习和Core-ML

本章我们将学习一些令人激动的东西，这就是机器学习——Core-ML。通过苹果最新的机器学习框架，我们可以在应用程序开发中让我们的应用更加智能。在2017年6月WWDC开发者大会上，苹果发布了机器学习。那时我们非常兴奋，因为我们可以在很多的应用中使用Core-ML来实现很多有趣和实用的事情。本章先会向大家介绍什么是机器学习（Machine Learning）。如果你之前从来没有听说过机器学习，或者只是在某些场合听到过一两次"机器学习"这个名词，那么现在了解什么是机器学习是非常必要的；然后会向大家介绍当前可以使用的不同类型的机器学习，或者说是在真实应用中可以使用的机器学习；最后我们会通过实战练习会教给大家如何使用Core-ML在iOS中用Swift语言去实现可视化的识别。

15.1　介绍机器学习

15.1.1　机器学习

什么是机器学习呢？简单来说呢，它就是通过一大堆的学习，让计算机可以通过没有明确代码的方式进行学习。本书自始至终都是教大家如何使用Swift语言进行程序开发，执行指令的设备不是计算机而是iPhone或者iPad。我们用一些明确的指令告诉它们要做什么，比如我们之前所做的Quizzler项目的应用，只有在用户单击了正确答案以后，iPhone才会显示一个对勾，告诉用户他答对了。这里我们使用了条件判断语句。如果反过来，我们让计算机去回答这些问题，然后我们就像教自己的孩子一样，教会计算机这些知识哪些是对的，哪些是错的，通过不断地体验来积累它们的知识技能，这就是机器学习。

我们再换一种说法，来解释什么是机器学习。就像是《星球大战》中的BB 8，我们想让它从起始的位置最终走到标有旗子的终点，如图15-1所示。

图15-1　BB 8达到旗子终点示意图

我们可以编写程序，让它先往前走，当遇到障碍物无法向前走的时候让它向右转，然后直达标旗目标。这是一个非常简单的程序代码脚本，而且效果非常好。但是，如果遮挡物移动到了其他位置，我们的程序代码就没有任何意义了，因为BB 8还是会按照原来的程序流程进行移动。当然你也可以让程序代码更加复杂一些，比如说你可以先确定终点的坐标，然后根据一些路由算法去找出最短路线，并最终到达终点；或者是当BB 8遇到障碍物的时候，让它向左或向右躲闪，最终到达终点，这也是完全可以的。

但是利用机器学习，我们可以只是简单地告诉BB 8你需要到达标旗终点就可以了。在这一过程中，它可能会走一些弯路，也有可能会走出

不同的线路。一旦它到达终点，我们就会告诉它：All right，你走到了目的地。随着时间的流逝，如果我们让BB 8保持这种不断的训练，它就会通过学习，来减少遇到遮挡物的次数，并找出最短的路由到达终点。而整个过程我们都不需要编写明确的代码，这就是机器学习的本质。

所以在这里我们为机器学习所下的定义就是：使计算机能够在没有明确编程的情况下进行学习。说这句话的人叫亚瑟·塞缪尔，他是机器学习的先驱之一，他第一个编写了能够称之为机器学习的算法。他在不给计算机编写任何明确代码的情况下，让机器自己去尝试和学习如何下国际象棋，然后再不断地重复游戏并提高自己的下棋水平。

机器学习通常被分为两个类别：一个是监督式学习，另一个是非监督式学习，这与我们要如何训练机器模型相关。

15.1.2　监督式学习

监督式学习（Supervised Learning），相当于你要手把手教计算机要学习的东西。一个最著名的例子就是教计算机如何识别一只猫。通过各种的猫的图片，如图15-2所示，我们会告诉计算机，哪个是一只猫。当然对于人类来说，识别一只猫是非常简单的事情，并且我们还会区分出图片上的动物若不是猫则是什么动物。

理解如何区分不同的动物的过程正是机器学习做的事情。我们为计算机提供了非常多的猫的图片，并且在提供图片的同时，还为它提供了一个标签（Label），告诉它这张图片里的动物是猫。通过标签和图像的混合数据，机器便知道了这是一只猫。这也就是为什么在监督式机器学习中，所有训练样本都是带标签的。

通过这样不断地学习，让机器了解到很多有关猫的特征，直到有一次我们只提供给计算机猫的一部分的图片，如图15-3所示，即使在没有标签的情况下，机器也能够识别出它是一只猫，因为这张图片包含了很多猫的特性，并与之前所提供猫的图片非常类似。

图15-2　监督式学习各种各样的猫

图15-3　只有部分特征的猫

在训练机器学习模型（Machine Learning Model）的时候，我们实际上会提供很多的图片，而且每张图片都会带有一个非常明确的标签。例如我们将一大堆狗的图片、一大堆猫的图片和一大堆奶牛的图片，都灌进机器学习模型之中。这几种动物都包含了不同的饲养方式、不同的大小和重量等苛刻的条件。机器通过之前的学习经验，开始去分类这些图片，把它们都放在独立的分组中，如图15-4所示。

图15-4　机器模型学习示意图

在这种情况下，模型就是机器中与学习相关的事情。你提供的图片就是训练样本。一旦我们完成了训练，再给机器提供一张训练样本中没有的图片，机器可以基于之前的学习，识别出这是一张狗的图片，并且最终拼写出答案——这是一只狗。在这里，我们管这张机器从来没有见过的图片叫测试数据，我们管输出的内容叫输出（output）。输出可以有各种形式，比如文字、在棋盘上所走的一步棋等。输出的结果依赖于我们如何训练模型，以及我们想要它做什么。这就会涉及监督式学习中一个最基本的概念——分类。

对于分类，你可以想象这样一个场景：你想让计算机学习关于苹果和梨之间的不同。这对于人类来说是一件非常容易的事情。但是在程序中，我们必须要告诉计算机，在苹果和梨之间都有哪些不同。其实，这是一个非常复杂的问题，因为你需要让计算机将这两张图放大到可以看到每一个像素，一般来说，苹果图片趋向于更多的红色，而梨趋向于更多的绿色。但是如果我们又提供给计算机一个绿色的苹果，那么计算机可能就会基于之前的学习。把它标识为梨。所以说，我们需要用更多的代码去设置苹果的特性，比如我们会说苹果比梨更圆一些、更红一些等。你可以尝试着给计算机呈现一些它之前没有见过的东西，比如说桃，这相当于一种异常事件，因为它既不属于苹果，也不属于梨。计算机会使用我们之前所设定的规则，尝试着将它分类。

这只是一个演示，即便是人，我们也很难准确分辨出所有事物。在人类整个学习过程中，就是要将苹果的唯一特征与其他水果区分开来。通过程序的规则列表去分类和识别这些图像，是非常困难且耗时较长的，比如让计算机去识别苹果和桃就更加的困难。

有关机器学习模型有一件非常好的事情，就是可以复用它。我们可以创建一个泛型分类，将手写数字识别成整型数，也就是相当于OCR。同时你可以使用相同的范型分类，再次进行更深层次的训练。比如我

们可以通过机器学习去识别邮件中的内容，从而判断这个邮件是垃圾邮件还是正常邮件。现在让我们看一下机器是如何做到的。

我们可以创建这样一个图表。图表中有一条线代表临界值，如果电子邮件达到了临界值以上，该邮件就会进入垃圾邮件，否则会进入收件箱。这个临界值是如何生成的呢？我们会有一个垃圾的智能过滤程序，我们在训练这个模型的时候使用了大批的邮件，并且为这些邮件都提供了垃圾或者是非垃圾的标签。

比如说，我们判断一封邮件是否为垃圾邮件，就会查看其内容的链接是否过多。如果这封邮件里面包含了上百个链接，那么就可以判定它是垃圾邮件。如果我们按照这个规则绘制图表的话，它就应该如图15-5所示这个样子。

图15-5　机器学习判断垃圾邮件

机器学习模型的工作就是尝试着去绘制这样一条线，通过所提供的训练样本，找出链接是多少的这个临界点。比如说，这个邮件中的链接数少于5个，那它可能就是正常的邮件，我们将它放入收件箱。如果超过5个，那它有可能就是垃圾邮件，我们会把它放到垃圾箱。这只是分辨垃圾邮件时涉及的众多特性中的一个，我们还可以通过邮件内容中的图片数量或者是邮件中“销售”“购买”等关键字的数量来确定。这样的学习会持续很长的时间。在邮箱中，每一次被标记为垃圾的邮件，都相当于给了机器学习一次机会。机器还会根据新的数据或者新的特性，来增加判断的准确性。

15.1.3 非监督式学习

非监督式学习与监督学习的不同之处在于，事先没有任何的训练样本，而需要直接对数据进行建模。这听起来似乎有点不可思议，但是在我们自身认识世界的过程中有很多地方都用到了非监督式学习。

比如我们去参观一个画展，我们对艺术一无所知，但是欣赏完多幅作品之后，我们也能把它们分成不同的派别，比如哪些画更朦胧一点，哪些画更写实一些。即使我们不知道什么叫朦胧派，什么叫写实派，但是至少我们能把它们分为两类。非监督式学习里典型的例子就是聚类（Clustering）了。聚类的目的在于把相似的东西聚在一起，而我们并不关心这一类是什么。因此，一个聚类算法通常只需要知道如何计算相似度就可以了。

那么，什么时候应该采用监督式学习，什么时候应该采用非监督式学习呢？一种非常简单的回答就是从定义入手。如果我们在分类的过程中有训练样本（Training Data），则可以考虑用监督式学习方法；如果没有训练样本，那就不可能用监督式学习了。但是事实上，我们在针对一个现实问题进行解答的过程中，即使没有现成的训练样本，我们也能够凭借自己的双眼，从待分类的数据中人工标注一些样本，并把它们作为训练样本，这样的话就可以用监督式学习的方法来做了。

15.1.4　强化学习

本节我们来说说强化学习。

如果我们在使用烤箱烘焙食物的时候，用手指触碰了里面非常热的东西，这个时候我们就会被灼伤，在未来的一段时间，我们就不会再触碰它了。因为疼痛的灼伤感，强化了我们对这个操作的记忆。但是这种记忆方式是比较残酷的，这可能导致我们自身受到很多的伤害，因此像这种负能量的学习是不可取的，如图15-6所示。

我们使用更多的强化学习方式是正向奖赏，例如我们训练一只狗，让它按照我们的要求做出某些动作，当我们在给狗下达“坐下”指令以后，它按照要求坐下，我们就会给它一定的奖赏，让它知道，自己做了一件正确的事情。这就是强化学习。

在强化学习中，一个最具代表性的应用就是下棋。如图15-7所示，假设棋盘左方的选手是使用了强化学习算法的机器。它会持续地去计算下棋期间赢的可能性。如果它实际走一步棋，并且这一步棋增加了赢的可能性，那么这就是正向强化。如果对手此时进行了有效还击，并减少了机器赢的可能性，这就是负向强化。机器通过与人下非常多的棋，并通过非常多的训练周期，就能够学到如何下每一步棋，从而让机器赢的可能性增加。现在世界上最著名的强化学习的应用程序就是谷歌的AlphaGo。

图15-6　不可取的强化学习

图15-7　最著名的强化学习应用程序——AlphaGO

15.2 Core-ML——整合机器学习到iOS应用中

为了完成后面的实战练习，我们需要Xcode 9和安装了iOS 11的物理真机。

15.2.1 什么是Core-ML?

现在我们来说说如何使用机器学习。该是Core-ML大显身手的时候了。首先我们要弄清楚什么是Core-ML，实际上它允许我们做两件事情，从而在iOS项目中，可以很容易地整合机器学习。

第一件事情就是允许我们载入一个预先训练好的机器学习模型，它包含一个简单的可转换我们之前训练模型的方式。我们可以将这个模型转换为一个类，让这个类在Xcode中使用，如图15-8所示。

图15-8 Core-ML示意图

我们将会把预先训练好的数据模型转换到.mlmodel文件之中，它是一个完全的开放的文件格式，并且包含了所有的输入输出。

第二件事情就是，Core-ML允许你去做一个断言（Predication），这样的话就可以将模型载入本地的iOS设备上。一旦用户下载并使用了你的应用程序，它就能使用模型去做断言，比如图像识别、语音识别或者是其他任何训练模型。

通过浏览器可以访问https://developer.apple.com/machine-learning/网址，以了解苹果关于机器学习的内容。它包括概述、如何在iOS中使用机器学习。网页中最有趣的部分是模型的下载，这里面提供了已经训练好的"即插即用"的模型，如图15-9所示。

图15-9 Apple提供的机器学习模型

我们在项目中将会使用谷歌的Inception v3模型。它能够检测出一千种物体，比如树木、动物、食物、车辆、人员等，它是通过大量的图片训练出来的模型。我们会使用它来识别在App中通过手机的摄像头所

拍摄出来的图片。另外，我们还会检测一个图片，看图片中的东西是否为热狗。

在WWDC大会上，苹果宣布了通过Core-ML能够做很多的事情。它能够尽可能详细地进行分类。另外，在绝大多数情况下，我们所实现的机器学习，仅仅是使用它来分类。

但我们应明白Core-ML不能做什么。我们不能使用自己的应用生成的那些数据样本来训练机器。当我们安装了一个提前训练好的模型时，它就相当于一个静态模型。在大多数情况下，这个模型会被放置在一个工作区里边，它只是放在那里。目前，还没有任何的方法能够让用户使用自己在App中生成的数据去训练数据模型。还有就是Core-ML不加密，如果你使用了一个专利数据模型，或者是模型中包含了敏感的数据，你就要知道这些都是不加密的。虽然Core-ML有上述的这些限制，但是它所做的事情还是足够令我们兴奋，我们甚至可以在不了解什么是机器学习的情况下，使用一些简单的代码做出令人不可思议的事情。

目前在App Store中，典型的与机器学习相关的应用有Garden Answers Plant Id，其可通过为植物的花和叶子拍照识别出该植物，如图15-10所示。

图15-10　使用到Core-ML的应用

15.2.2　Core-ML能做什么

本节我们会制作一个识别热狗的应用程序，在应用中我们会调用摄像头来拍摄照片，然后通过Core-ML来将其分类，并判断它是否为热狗。

实战：搭建图像识别应用的构架。

步骤1：在Xcode中创建一个新的Single View App项目，将Product Name设置为SeeFood。

步骤2：在之前提到的苹果网站中下载Model到本地，选择Inception V3，因为它的识别分类效果是目前最好的。将下载好的Inceptionv3.mlmodel文件拖曳到Xcode项目之中，确认勾选了Copy items if needed，单击Finish按钮。

一旦我们将Inceptionv3.mlmodel文件添加到项目之中，在项目导航中单击该文件以后，在编辑窗口中我们就可以看到Xcode已经为该模型创建了一个Model Class，如图15-11所示。

图15-11　将Inceptionv3.mlmodel文件添加到项目中

单击Model Class中Inceptionv3右侧的箭头，就可以看到类的相关代码。接下来，我们需要编写一些代码将模型整合到视图控制器中。

步骤3：在ViewController.swift文件中，导入两个框架。

```
import CoreML
import Vision
```

vision框架可以帮助我们更简单地处理图像，其允许我们让图像在Core-ML中工作，并且免去编写一大堆的代码的麻烦。

实战：通过照片获取器在应用中拍照。

使用UIImagePicker类可以让我们通过摄像头拍照并选择照片进行识别，整个过程非常简单。

步骤1：在ViewController类的声明中添加UIImagePickerControllerDelegate协议。

```
class ViewController: UIViewController,
UIImagePickerControllerDelegate, UINavigationControllerDelegate
{
```

在使用UIImagePickerControllerDelegate协议的时候，我们必须实现UINavigation-ControllerDelegate协议，它们是依赖关系。

步骤2：在故事板中选择ViewController视图，然后在菜单中选择Editor/Embed In/Navigation Controller，让当前的控制器内置于一个导航控制器之中，如图15-12所示。

图15-12　内嵌导航控制器

此时会有一个导航栏位于视图的顶部，这样就可以通过导航来控制控制器之间的切换了。

步骤3：在对象库中找到Bar Button Item对象，将其添加到ViewController视图导航栏的右侧。在Attributes Inspector中将System Item设置为Camera，如图15-13所示。

图15-13　设置Bar Button Item

步骤4：在对象库中将Image View添加到ViewController视图，该控件用于显示摄像头所拍摄的，或者是从iPhone照片库选择的图像。设置其大小占满控制器整个视图空间（除导航栏），并设置它的约束在四个方向上均为0（代表紧密贴合屏幕的三个边和上边的导航栏），如图15-14所示。

图15-14　创建必要的约束

步骤5：将之前的拍照按钮与ViewController类建立IBAction关联。将Image View与ViewController类建立IBOutlet关联。

```
@IBOutlet weak var imageView: UIImageView!

@IBAction func cameraTapped(_ sender: UIBarButtonItem) {
}
```

步骤6：在ViewController类中添加一个新的属性。

```
let imagePicker = UIImagePickerController()

override func viewDidLoad() {
  super.viewDidLoad()

  imagePicker.delegate = self
  imagePicker.sourceType = .camera
  imagePicker.allowsEditing = false
}
```

这里创建的imagePicker对象就是iOS中的照片获取器。为了可以在用户选择照片以后通知到ViewController类，在viewDidLoad（）方法中，需要设置其delegate属性指向当前类。

sourceType属性用于指定照片获取器通过什么渠道获取照片，这里设置为摄像头。allowsEditing属性是一个布尔型值，用于指定是否允许用户编辑选中的照片或视频。你可能想开启它，因为这样可以让用户裁剪照片，让机器学习更少的区域，从而更有针对性地进行识别。这里我们暂时将它设置为false，即不激活该功能。

接下来，我们需要在用户单击导航栏的摄像头图标的时候调出照片获取器。

步骤7：在cameraTapped（）方法中添加下面的代码。

```
@IBAction func cameraTapped(_ sender: UIBarButtonItem) {
  present(imagePicker, animated: true, completion: nil)
}
```

通过present（）方法，将UIImagePickerController对象以动画的方式呈现到屏幕上，在呈现完成以后不会执行任何代码。

下面，我们要通过照片获取器来得到照片，并通过机器学习进行识别。

实战：获取照片以后通过机器学习识别。

步骤1：在ViewController类中添加下面的方法。

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {
  let userPickedImage =
info[UIImagePickerControllerOriginalImage]
  imageView.image = userPickedImage
}
```

当用户从照片获取器中成功取图像以后会调用该方法，它带有2个参数：picker指明的是该方法来自于哪个UIImagePickerController对象的

调用，当前是来自于imagePicker对象；参数info是字典类型格式，在该字典中，包含了用户所选择的图像。

在方法内部，首先通过info字典获取用户选择的图像，因为info是字典，所以需要通过键获取UIImage类型的图像，这个键名就是iOS SDK预定义好的UIImagePickerControllerOriginalImage，代表用户所选择的图像原始图。

因为在方法中info字典的类型为[String：Any]，所以从info字典得到的值的类型为Any？，在将userPickedImage赋值给imageView的image属性的时候，编译器会报错Cannot assign value of type'Any？'to type'UIImage？'，即不能将Any？类型赋值给UIImage？类型。

步骤2：将之前方法中的代码修改为下面这样。

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {
  if let userPickedImage =
info[UIImagePickerControllerOriginalImage] as? UIImage {
    imageView.image = userPickedImage
  }
  imagePicker.dismiss(animated: true, completion: nil)
}
```

这里，使用可选绑定方式，如果info[UIImagePickerControllerOriginalImage]的值存在，则将其转换为可选UIImage类型，然后赋值给userPickedImage常量。如果有值，则再将userPickedImage赋值给imageView的image属性。这样既增加了代码的可读性，又使代码更加安全，所有的错误都消失了。

在方法的最后，我们使用dismiss（）方法销毁照片获取器。

因为在项目中使用了摄像头，所以需要运行在真机上面。但是在构建并运行项目的时候，应用程序会发生崩溃，从控制台可以看出是这因

为reason：'Source type 1not available'。因为我们所运行的项目试图访问一个敏感数据而没有使用描述。

步骤3：打开info.plist文件，添加新的属性Privacy-Camera Usage Description，然后将值设置为：当前的应用程序需要使用你的摄像头。

提示　如果你想让应用程序可以访问用户的照片库，则需要设置Privacy-Photo Library Usage Description属性。

再次构建并运行项目，可以看到在使用摄像头的时候会让用户选择是否允许使用摄像头或访问照片库。

技巧　如果将imagePicker的sourceType属性修改为.photoLibrary，则可以在模拟器中打开该项目，然后通过照片库来载入图像。

15.2.3　如何识别图像并反馈结果

本节我们开始在项目中使用模型，将Inception V3机器学习模型整合到项目之中，并对相关的操作做出一定的解释。

实战：为项目添加图像识别功能。

步骤1：将从imagePicker中获取的UIImage类型的图像转换为CIImage类型。

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {
  if let userPickedImage =
info[UIImagePickerControllerOriginalImage] as? UIImage {
    imageView.image = userPickedImage

    guard let ciimage = CIImage(image: userPickedImage) else {
      fatalError("无法转换图像到CIImage")
    }

    detect(image: ciimage)
  }
  imagePicker.dismiss(animated: true, completion: nil)
}
```

这里我们需要将从照片获取器得到的UIImage对象转换为CIImage对象，因为它是标准的Core Image图像，而且在使用Vision和Core-ML框架中的方法时，必须要用这种特定类型。

如果我们只是通过let ciimage=CIImage（image：userPickedImage）语句赋值一个ciimage常量，虽然运行没有任何问题，但是我们希望可以为代码添加一些安全特性，让代码的运行更加安全。

通过guard……else{……}语句，如果guard后面的代码运行失败，则会执行else中的代码。在当前的情况下，如果不能将userPickedImage转

换为CIImage类型的对象，则会激活一个致命错误，并提示：无法转换图像到CIImage，程序终止运行。

接下来，我们需要处理这个CIImage对象，并获取图像的解读信息或分类。

步骤2：在ViewController类中添加一个方法。

```
func detect(image: CIImage) {
  guard let model = try? VNCoreMLModel(for:
Inceptionv3().model) else {
    fatalError("载入CoreML模型失败")
  }

  let request = VNCoreMLRequest(model: model) { (request,
error) in
    guard let results = request.results as?
[VNClassificationObservation] else {
      fatalError("模型处理图像失败")
    }

  print(results)
}
```

detect（）方法带有一个CIImage参数，在方法体中我们将使用Inception V3模型。我们通过VNCoreMLModel类创建一个model常量，并将Inception V3作为该对象的机器学习模型。之后，我们会使用Inception V3对图像进行分类。

如果在代码中我们按let model=VNCoreMLModel（for：Inceptionv3（）.model）这样编写，编译器会报错：VNCoreMLModel（）方法有一个抛出（throw），但是还没有标记try关键字和相应的错误处理代码。

因为VNCoreMLModel（）方法会在初始化失败的时候抛出带有错误描述的异常，所以我们需要使用try关键字去尝试执行后面的代码。如果

代码执行成功，则将其封装到一个可选常量里面。如果执行失败就会抛出错误，然后model就会被赋值为nil。

在这种情况下，我们再次使用guard关键字，在model的值为nil的情况下，就会运行else中的代码。

接下来，我们需要实现一个图形分析请求，这样才能通过Core-ML模型去处理图像。通过VNCoreMLRequest（）创建一个请求，其第一个参数model代表我们所使用的模型，第二个是完成处理的闭包参数，代表请求执行完毕后要做的事情。在闭包内部，通过request的results属性，我们可以得到图像的分析处理结果，该结果是VNClassificationObservation类型的数组。如果results获取失败则会执行fatalError（）方法。

在方法的最后，我们直接打印结果到控制台。

步骤3：在let request=语句的下面，添加下面两行代码。

```
let request = VNCoreMLRequest(model: model) { (request, error)
in
  ……
}

let handler = VNImageRequestHandler(ciImage: image)
try! handler.perform([request])
```

将之前转换好的CIImage对象作为VNImageRequestHandler对象的参数输入到模型中。最后，在handler中执行之前生成的请求。注意，请求是以数组的形式作为参数的，这代表我们可以同时执行多个请求。这里直接使用try！代表handler的perform（）方法不会出现任何的问题，强制执行这个方法。

步骤4：在imagePickerController（_picker：UIImagePickerController，didFinishPicking-Media WithInfo info：

[String：Any]）方法中添加对detect（image：CIImage）方法的调用。

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {
  if let userPickedImage =
info[UIImagePickerControllerOriginalImage] as? UIImage {
    imageView.image = userPickedImage

    guard let ciimage = CIImage(image: userPickedImage) else {
      fatalError("无法转换图像到CIImage")
    }
    // 调用图像识别
    detect(image: ciimage)
  }
  imagePicker.dismiss(animated: true, completion: nil)
}
```

构建并运行项目，在真机上面使用该应用拍照后，可以在控制台中看到大量的输出信息，如图15-15所示。其中排在输出第一位的就是相似度最高的名称。在当前的照片中，与猕猴（macaque）有97.9313%相似度，接下来的其他名称相似度就少得可怜了。是不是觉得很神奇呢?

15.2.4　判断图片中的食物

本节我们将会实现通过照片获取器判断图像中的内容是否为“热狗”，或者是其他任何指定的东西。

在detect（image：CIImage）方法中，我们首先导入Inception V3模型并创建请求（request），然后要求模型去识别和分类任何数据。在处理数据完成以后，执行VNCoreMLRequest的回调闭包。目前我们只是将分类信息打印到控制台。

图15-15　机器学习分析后的结果

接下来我们需要做的就是从结果中判断图像中是否为热狗，也就是相似度最高的是否为热狗。通过之前在控制台打印出的信息，我们可以看到数组形式的VNClassi-ficationObservation对象。我们需要获取其中的第一个元素。

实战：识别图像是否为热狗。

修改detect（image：CIImage）方法中的闭包代码。

```
let request = VNCoreMLRequest(model: model) { (request, error)
in
  guard let results = request.results as?
[VNClassificationObservation] else {
    fatalError("模型处理图像失败")
  }

  //print(results)
  if let firstResult = results.first {
    if firstResult.identifier.contains("hotdog") {
      self.navigationItem.title = "热狗！"
    }else {
      self.navigationItem.title = "不是热狗！"
    }
```

```
    }
}
```

这里先注释掉之前的打印语句，然后获取数组中的第一个元素，它是VNClassification-Observation类型的对象。注意，第一个元素所提供的信息是相似度最高的。通过contains（）方法，我们可以判断VNClassificationObservation的identifier属性中是否包含dumplings字符串。如果包含，则让导航栏的标题部分显示热狗，否则显示不是热狗。

构建并运行项目，通过照片获取器取得一张照片，代码会自动分析该照片是否为热狗，并将结果显示到导航栏中，如图15-16所示。

图15-16　机器学习分析图片是否为热狗