TURING 图灵原创

基于iOS 8正式版全面修订

配有同步练习、同步视频教程、同步实战项目 / 分层架构设计解决 Swift 与 Objective-C 混合搭配问题 / 畅销书《iOS 开发指南》作者关东升最新著作

Swift 开发指南（修订版）

关东升 赵志荣 著

人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：Swift开发指南（修订版）
作者：关东升　赵志荣
ISBN：978-7-115-37333-5

前言

2014年8月1日国内第一本Swift图书——《Swift开发指南》正式上市了，这是我们智捷iOS课堂与图灵教育合作的又一本iOS图书，之前合作的《iOS开发指南：从零基础到App Store上架》承蒙广大读者的厚爱，获得了不错的销售业绩。

由于《Swift开发指南》这本书是基于Xcode 6 beta 4版本编写的， Xcode 6最终版本在Swift语法方面、Xcode的操作界面和工程构建方面都有一些变化，因此我们及时编写了《Swift开发指南》的修订版本。

主要修订的内容包括：Unicode编码表现形式的变化（参见6.1.1节），描述范围的半闭区间表现形式的变化（参见7.1.3节），数组元素追加（参见8.3.2节），访问限定新特性（参见11.7节），析构器示例修改（参见14.3节），构造器继承（参见15.2.3节），Mac OS X工程修改为iOS工程（参见18.2.1节和18.3.1节）和项目实战中iPhone计算器修改（参见21.3节）。

内容和组织结构

本书是我们团队编写的iOS系列图书之一，目的是使从事iOS开发的广大读者通过本书的学习掌握苹果Swift语言，对于原来有Objective-C开发经验的人，能够快速转型到Swift上来开发iOS应用。全书共分为4个部分。

第一部分为基础篇，共10章内容，介绍了Swift的一些基础知识。

第1章介绍了Swift的开发背景以及本书约定。

第2章介绍了如何使用Xcode的Playground编写和运行Swift程序代码，讲述了Swift程序结构以及Playground工具的使用。

第3章介绍了Swift一些基本语法，其中包括标识符和关键字、常量、变量、表达式和注释等内容。

第4章介绍了Swift一些基本运算符，这些运算符包括算术运算符、关系运算符、逻辑运算符、位运算符和其他运算符等。

第5章介绍了Swift一些数据类型，例如UInt8、Int8和Double等，此外还有元组（tuple）等类型。

第6章介绍了Swift中的字符和字符串，以及字符串可变性和字符串的比较等内容。

第7章介绍了Swift语言的控制语句，其中包括分支语句（if和switch）、循环语句（while、do while、for和for in）和跳转语句（break、continue、fallthrough和return）等。

第8章介绍了Swift中提供的两种数据结构的实现：数组和字典。

第9章介绍了Swift中的函数。Swift中的函数可以独立存在，即全局函数；也可以在别的函数中存在，即函数嵌套；也可以在类、结构体和枚举中存在，即方法。

第10章介绍了Swift语言中的闭包，其中包括了闭包的概念、闭包表达式、尾随闭包和捕获值等内容。

第二部分为面向对象篇，共7章，介绍了Swift语言面向对象的相关知识。

第11章首先介绍了现代计算机语言中面向对象的基本特性，然后介绍了Swift语言中面向对象的基本特性，主要包括枚举、结构体和类基本概念及其定义。最后还介绍了Swift面向对象类型嵌套、可选类型和可选链等基本概念。

第12章介绍了Swift中属性和下标的基本概念及其使用规律，主要包括存储属性、计算属性、静态

属性和属性观察者等重要的属性概念。此外，还介绍了下标的概念及使用。

第13章介绍了Swift语言的方法概念、方法的定义以及方法的调用等内容，并讲述了使用实例方法和静态方法声明和调用。

第14章介绍了Swift语言的对象类型的构造过程和析构过程，还介绍了构造器和析构器的使用方法。

第15章讨论了Swift语言的继承性，介绍了Swift中继承只能发生在类类型上，而枚举和结构体不能发生继承，还介绍了Swift中子类继承父类的方法、属性、下标等特征过程，以及如何重写父类的方法、属性、下标等特征。

第16章介绍了Swift中扩展和协议的基本概念及其重要性。具体讲述了如何扩展属性、扩展方法、扩展构造器和扩展下标。在协议部分，介绍了协议如何规定方法和属性，如何把协议当作一种类型使用，以及协议的继承和合成机制。

第17章介绍了Swift中的内存管理机制，讲述了ARC内存管理的原理，以及如何解决对象间强引用循环问题和闭包与引用对象之间强引用循环问题。

第三部分为过渡篇，共两章，主要介绍了如何从Objective-C过渡到Swift，以及它们之间的互相

调用问题。

第18章介绍了如何从Objective-C过渡到Swift，再从Swift调用Objective-C。

第19章介绍了Foundation框架，讲解了如何通过Swift语言使用Foundation框架，还详细介绍了Foundation框架中的数字、字符串、数组、字典等。

第四部分为实战篇，共两章，介绍了iOS应用iPhone计算器的开发过程。

第20章介绍了iOS开发的一些基础知识，包括开发环境Xcode、iOS SDK和iOS API等内容。通过一个基于iPhone的HelloWorld实例项目，介绍iOS应用的运行基本原理。

第21章介绍了iOS应用开发的一般流程，讲述了Objective-C语言与Swift语言混合搭配和调用以及分层架构设计的重要性，并且使用4个（纯Swift、纯Objective-C、Swift调用Objective-C和Objective-C调用Swift）版本实现了iPhone计算器应用。

本书的重点是介绍Swift语言，只是在本书的最后介绍了一些iOS开发的基础知识。有关iOS的更多知识，请大家关注我们智捷iOS课堂的相关图书。

本书网站

为了更好地为广大读者提供服务，我们专门为本书建立了一个服务平台，网址是http://51work6.com/swift.php，大家可以查看相关出版信息，并对书中内容发表评论，提出宝贵意见。

源代码

书中包括了150多个完整的案例项目源代码，大家可以到本书网站下载，或者到图灵社区本书主页（www.iTuring.cn/book/1517）免费注册下载。

同步练习

为了帮助读者消化吸收本书介绍的知识，我们在每一章后面都安排了数量不等的同步练习题。为了能够让广大读者主动思考，同步练习题的参考答案并没有放在书中，而是放在了本书网站上，我们为此专门设立了一个讨论频道。大家也可以到图灵社区本书主页下载和参考。

勘误与支持

我们在本书网站建立了一个勘误专区，可以及时地把书中的问题、失误和纠正反馈给广大读者。如果你发现了任何问题，均可以在网上留言，也可以发送电子邮件到eorient@sina.com，我们会在第一时间回复你。此外，你也可以通过新浪微博与我们联系，我的微博为@tony_关东升。

致谢

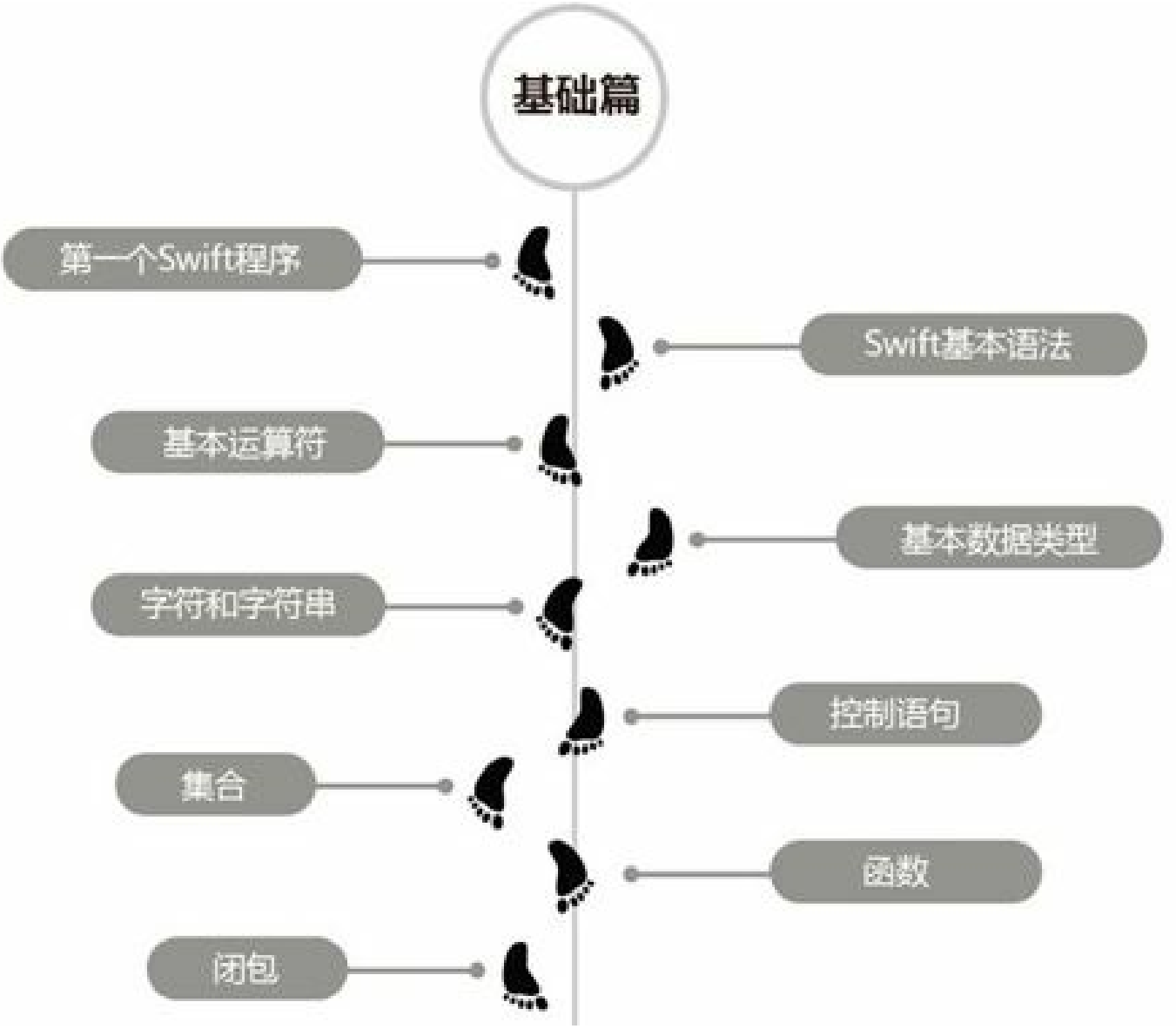
在此感谢图灵的编辑王军花和张霞给我们提供的宝贵意见，感谢智捷iOS课堂团队的贾云龙参与内容的讨论和审核，感谢赵大羽老师手绘了书中全部草图，并从专业的角度修改书中图片，力求更加真实完美地奉献给广大读者。此外，还要感谢我的家人容忍我的忙碌，以及对我的关心和照顾，使我能抽出这么多时间，投入全部精力专心编写此书。

由于时间仓促，书中难免存在不妥之处，请读者原谅。

关东升

2014年10月于北京

第一部分　基础篇


基础篇
第一个Swift程序
Swift基本语法
基本运算符
基本数据类型
字符和字符串
控制语句
集合
函数
闭包

第 1 章　准备开始

当你拿到这本书的时候，我相信你已经下定决心开始学习Swift语言了。那么应该怎么开始呢？这一章我们不讨论技术，而是告诉大家本书的结构、书中的一些约定、开发工具，以及如何使用本书的案例。

1.1　本书学习路线图

本书共分为4篇：基础篇、面向对象篇、过渡篇和实战篇。如图1-1所示，这是Swift的学习路线图，也是本书的内容结构图。

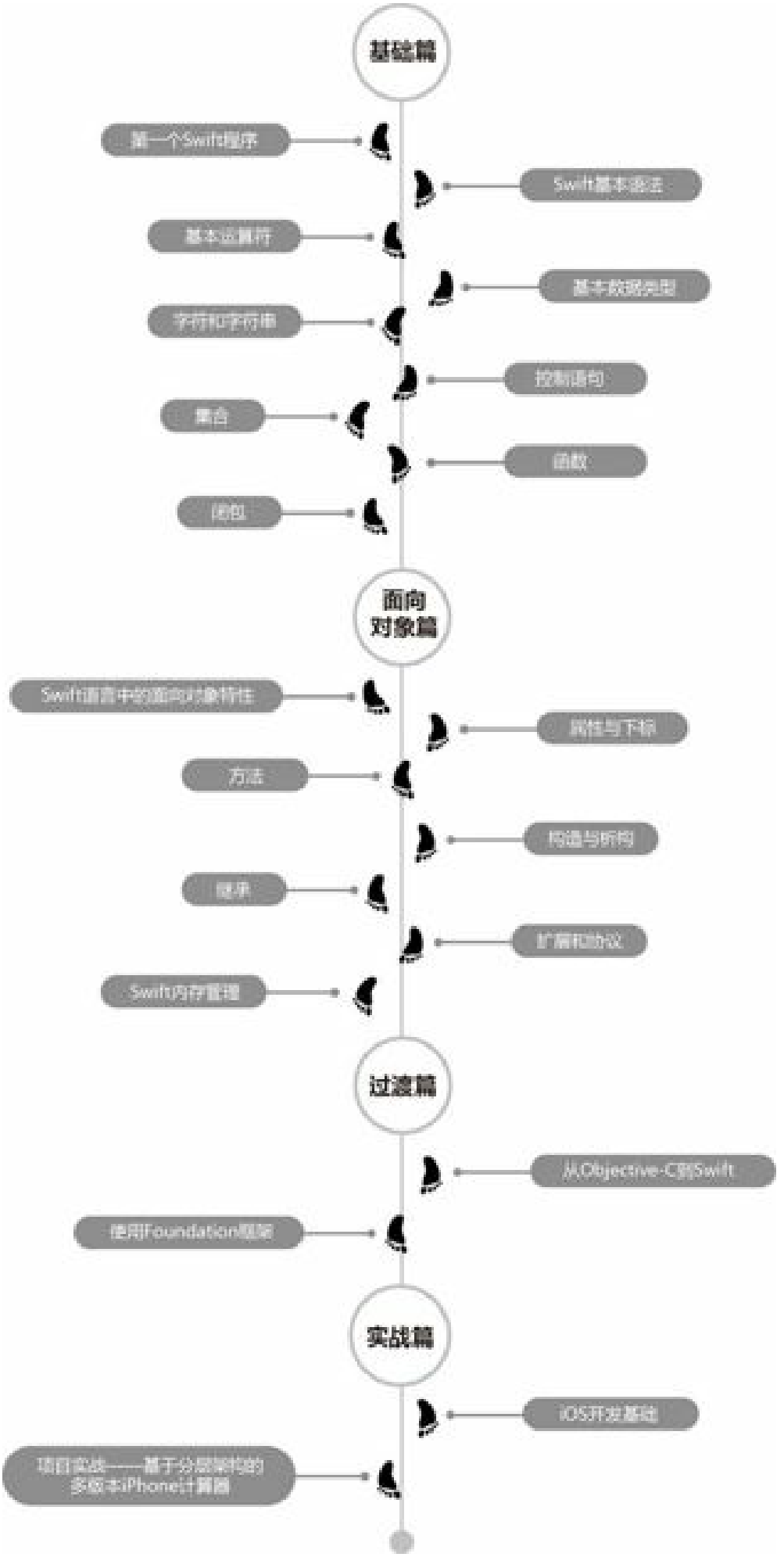

基础篇
第一个Swift程序
Swift基本语法
基本运算符
基本数据类型
字符和字符串
控制语句
集合
函数
闭包
面向对象篇
Swift语言中的面向对象特性
属性与下标
方法
构造与析构
继承
扩展和协议
Swift内存管理
过渡篇
从Objective-C到Swift
使用Foundation框架
实战篇
iOS开发基础
项目实战——基于分层架构的多版本iPhone计算器

图 1-1　Swift学习路线图

1.2　本书中的约定

为了方便大家阅读，本节会介绍一下书中示例代码和图示的相关约定。

1.2.1　示例代码约定

本书包含了大量示例代码，我们可以从图灵网站本书主页免费注册下载，或者从智捷教育提供的本书服务网站（www.51work6.com）下载，解压后会看到如图1-2所示的目录结构。

图 1-2　源代码文件目录

其中，ch2～ch21代表第2章到第21章的示例代码或一些资源文件，其中工程或工作空间的命名有如下几种形式。

- 二级目录标号，如“14.3析构器”说明是14.3节中使用的“析构器”Xcode工程或Playground文件示例。图标为  的是Playground文件，图标为  的是Xcode工程。
- 三级目录标号，如“14.1.1默认构造器”说明是14.1.1节中使用的“默认构造器”。带有“~”的情况，如“2.2.2 ~ 2.2.3 HelloWorld”说明是2.2.2节到2.2.3节共同使用的HelloWorld工程或Playground文件示例。
- ch20目录下只有一个HelloWorld目录，没有标号，说明第20章整章都使用HelloWorld这个示例。

1.2.2 图示的约定

为了更形象有效地说明知识点或描述操作，本书添加了很多图示，下面简要说明图示中一些符号的含义。

- 图中的圈框。有时读者会看到如图1-3所示的圈框，框中的内容是选中的内容或重点

说明的内容。

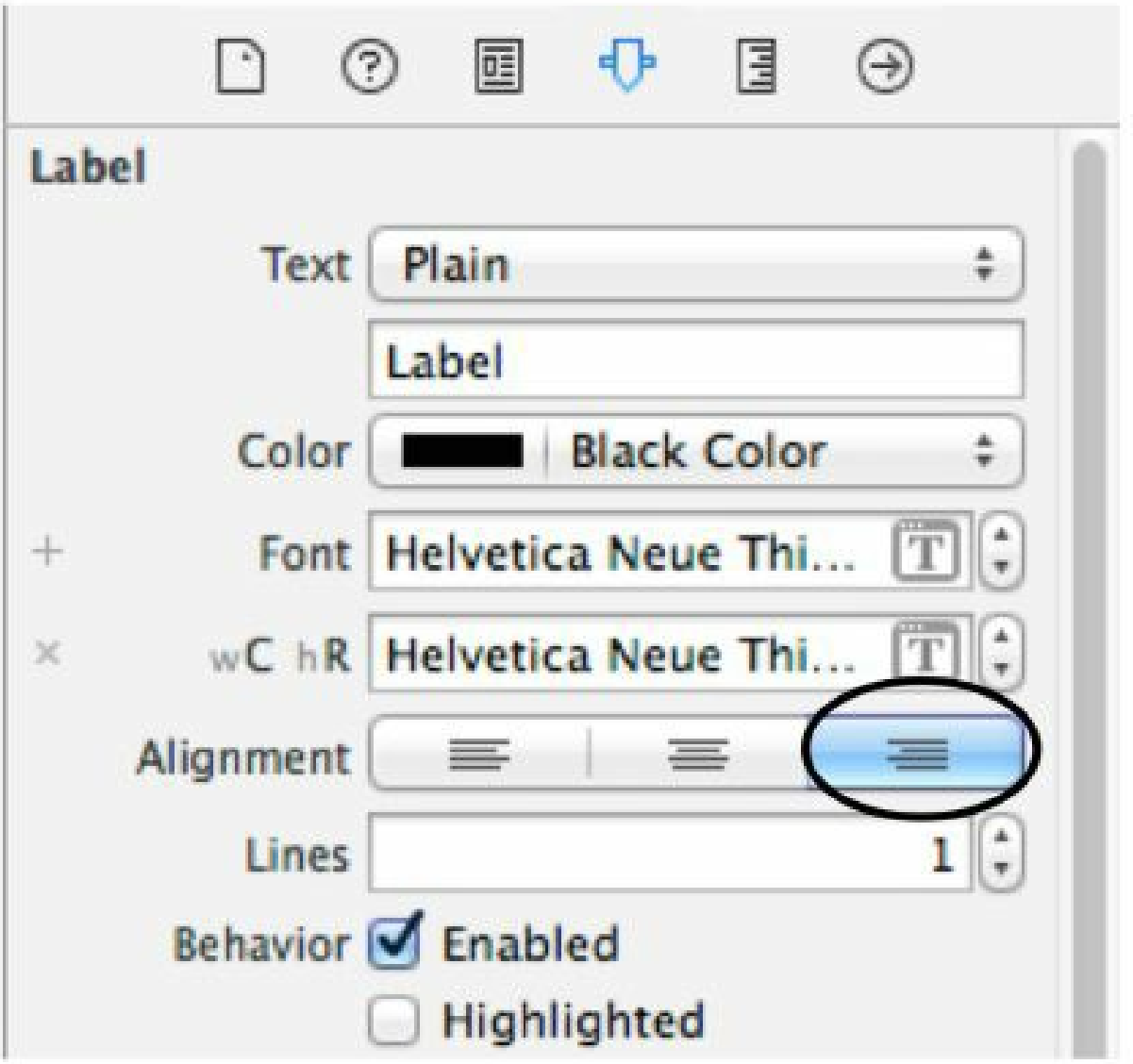


图 1-3　图中圈框

- 图中的箭头。如图1-4所示，实线箭头用于说明用户的动作，一般箭尾是动作开始的地方，箭头指向动作结束的地方。图1-5所示的虚线箭头在书中用得比较多，常用来描述设置控件的属性等操作。

图 1-4　图中实线箭头

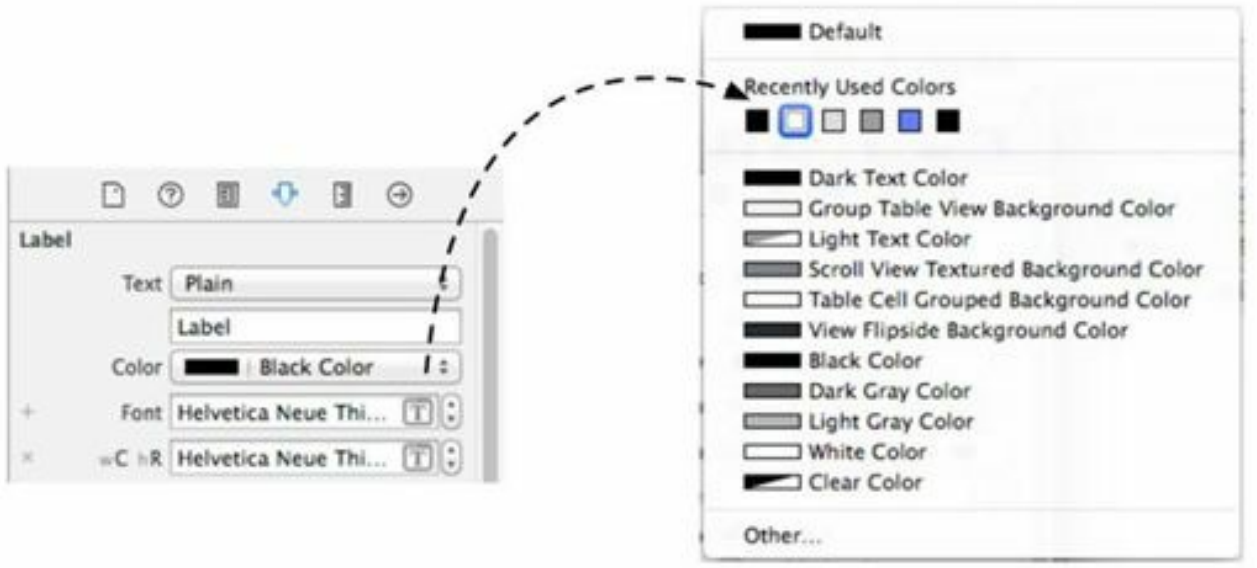


图 1-5　图中虚线箭头

- 图中手势。为了描述操作，我们在图中

放置了  等手势符号，这说明点击了此处的按钮，如图1-6所示。

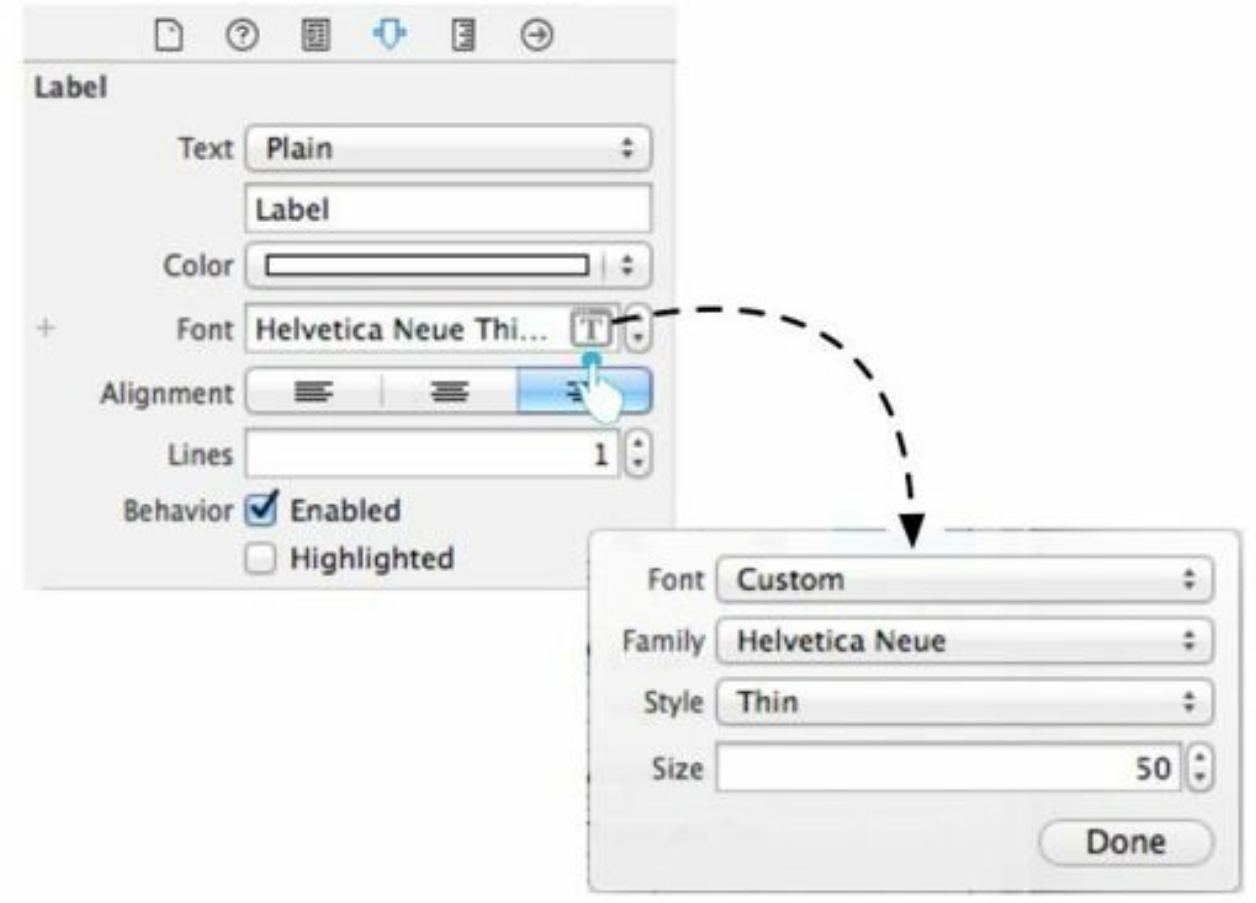

Label
Text
Plain
Label
Color
Font
Helvetica Neue Thi...
Alignment
Lines
1
Behavior
Enabled
Highlighted
Font
Custom
Family
Helvetica Neue
Style
Thin
Size
50
Done

图 1-6　图中手势

1.3 Xcode开发工具

苹果公司于2008年3月6日发布了iPhone和iPod touch的应用程序开发包，其中包括了Xcode开发工具、iPhone SDK和iPhone手机模拟器。第一个Beta版本是iPhone SDK 1.2b1（build 5A147p），发布后可以立即使用，但是同时推出的App Store所需要的固件更新直到2008年7月11日才发布。编写本书时，iOS SDK 8正式版本已经发布。

Mac OS X和iOS开发工具主要是Xcode。自从Xcode 3.1发布以后，Xcode就成为iPhone软件开发工具包的开发环境。Xcode可以开发Mac OS X和iOS应用程序，其版本是与SDK相互对应的。例如，Xcode 3.2.5与iOS SDK 4.2对应，Xcode 4.1与iOS SDK 4.3对应，Xcode 4.2与iOS SDK 5对应，Xcode 4.5和Xcode 4.6与iOS SDK 6对应，Xcode 5与iOS SDK 7对应，Xcode 6与iOS SDK 8对应。

在Xcode 4.1之前，还有一个配套使用的工具Interface Builder，它是Xcode套件的一部分，用来设计窗体和视图，通过它可以“所见即所得”地拖曳控件并定义事件等，其数据以XML的形式存储在xib文件中。在Xcode 4.1之后，Interface

Builder成为了Xcode的一部分，与Xcode集成在一起。

本书介绍的Swift开发语言使用Xcode 6.1工具进行学习和编写，后面在介绍iOS应用开发时也会使用Xcode。

1.3.1 Xcode安装和卸载

Xcode必须安装在Mac OS X系统上，Xcode的版本与Mac OS X系统版本有着严格的对应关系，Xcode 6要求Mac OS X版本在 10.9.2以上。

安装可以通过Mac OS X的Dock启动App Store，如图1-7所示。如果我们需要安装软件或查询软件则需要用户登录，这个用户就是你的App ID，弹出的登录对话框如图1-8所示。如果你没有登录App ID，可以点击“创建App ID”按钮创建。

图 1-7　应用启动App Store界面

图 1-8　App Store用户登录界面

之后，我们可以在右上角的搜索栏中输入要搜索的软件或工具名称“Xcode”关键字，搜索结果如图1-9所示。

图 1-9　搜索Xcode工具

点击“Xcode”进入Xcode信息介绍界面，如图1-10所示，点击“Install”按钮开始安装。

图 1-10　Xcode安装

提示 如果想使用测试版本的Xcode，不能通过App Store下载和安装，App Store所能下载的都是正式发布版本，测试版本的Xcode只有苹果开发者通过开发者账号才能下载。图1-11所示是Xcode测试版下载页面。如果没有开发者账号，大家可以到本书网站（www.51work6.com）下载。

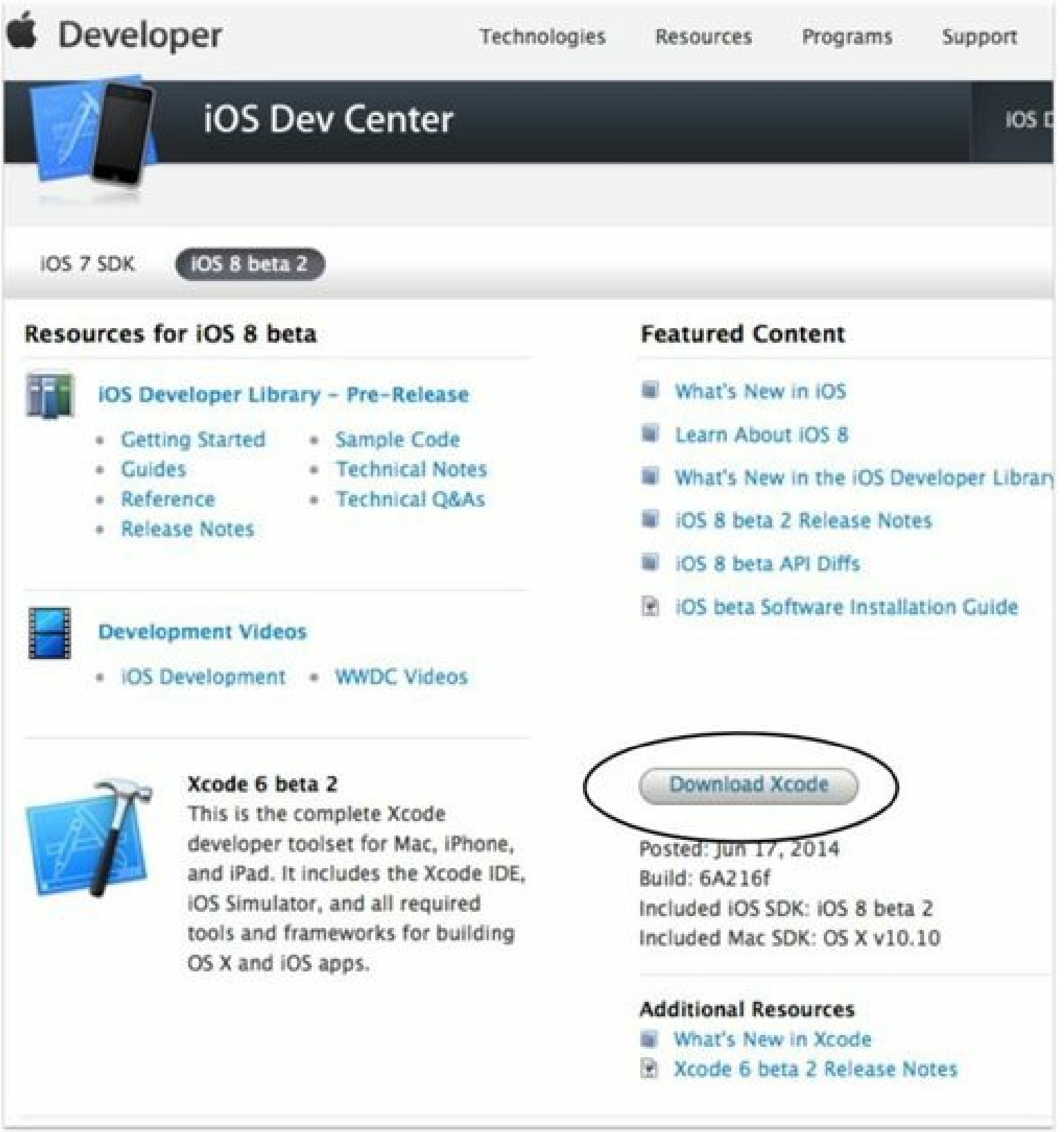


图 1-11　Xcode测试版下载页面

卸载Xcode非常简单，事实上只需要在Mac OS X应用程序中直接删除就可以了。如图1-12所示，

打开应用程序，右键选择“Xcode”弹出菜单，选择“移到废纸篓”删除Xcode应用。如果想彻底删除，只需清空废纸篓就可以了。

图 1-12　Xcode卸载

1.3.2　Xcode界面

打开Xcode 6工具，看到的主界面如图1-13所示。该界面主要分成3个区域，①号区域是工具栏，其中的按钮可以完成大部分工作。②号区域是导航栏，主要是对工作空间中的内容进行导航。③号区域是代码编辑区，我们的编码工作就是在这里完成的。在导航栏上面还有一排按钮，如图1-14所示，默认选中的是“文件”导航面板。关于各个按

钮的具体用法，我们会在以后用到的时候详细介绍。

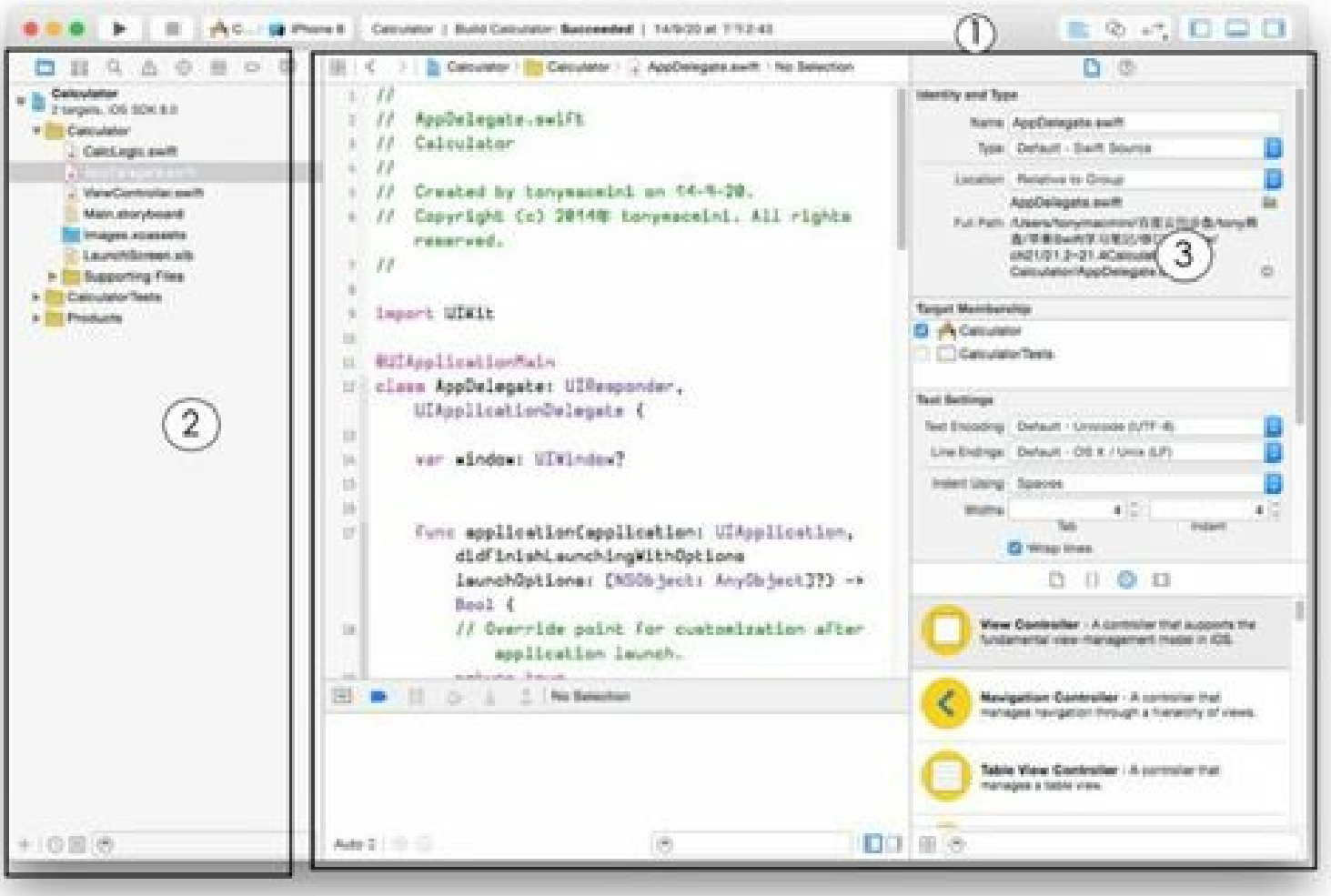


图 1-13　Xcode主界面

图 1-14　Xcode导航面板

在选中导航面板时，导航栏下面也有一排按钮，如图1-15所示。这是辅助按钮，它们的功能都

与该导航面板内容相关。对于不同的导航面板，这些按钮也是不同的。

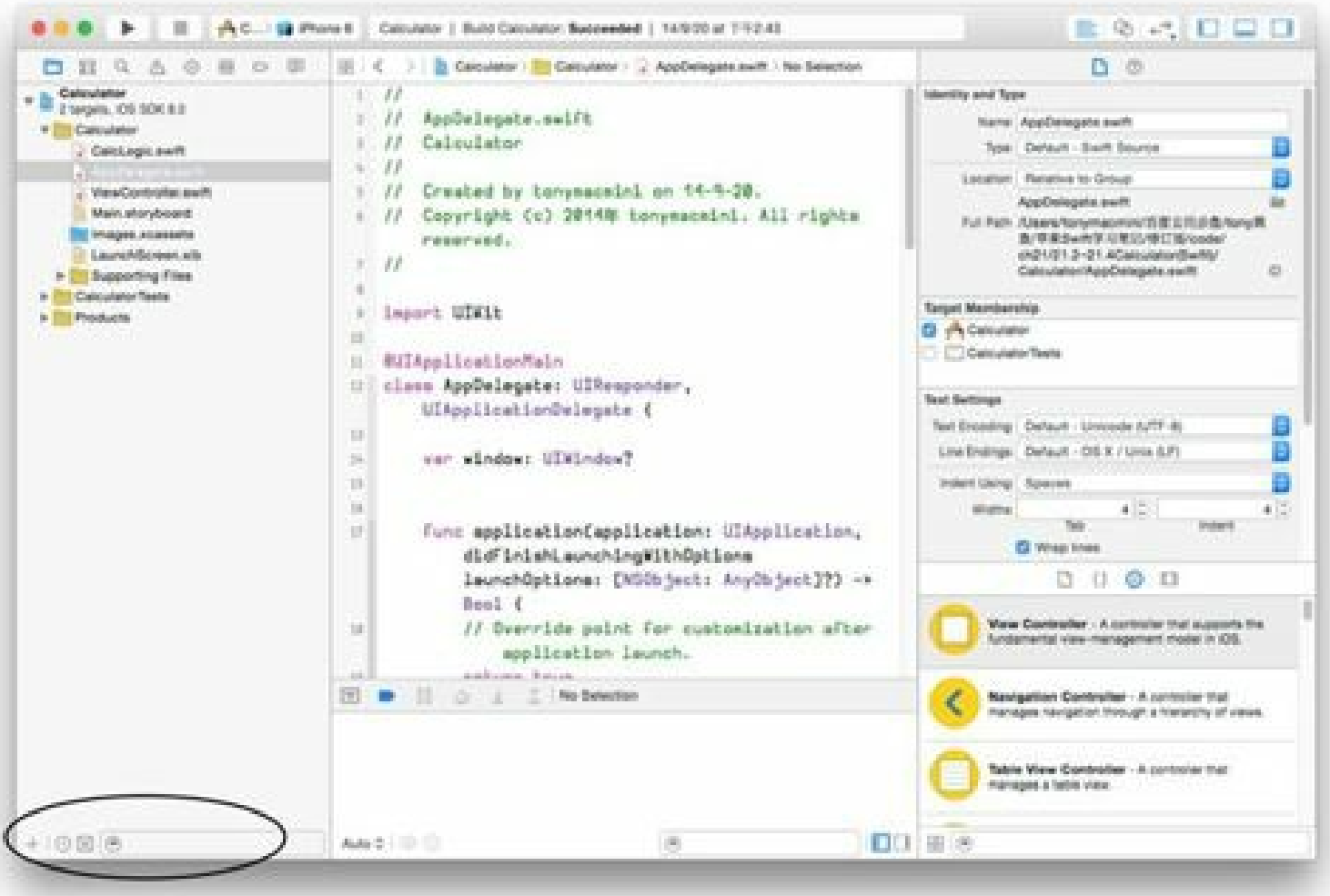


图 1-15　导航面板的辅助按钮

此外，学习Swift还可以使用Playground进行编写和运行，它也是Xcode 6开发工具之一，我们将在第2章介绍。

1.4 如何使用API帮助

对于初学者来说，学会在Xcode中使用API帮助文档是非常重要的。下面我们通过一个例子来介绍API帮助文档的用法。

在编写HelloWorld程序时，可以看到ViewController.swift的代码，具体如下所示：

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after
loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be
recreated.
    }


}
```

如果我们对于`didReceiveMemoryWarning`方法感到困惑，就可以查找帮助文档。如果只是简单查看帮助信息，可以选中该方法，然后选择右边的快捷帮助检查器，如图1-16所示。

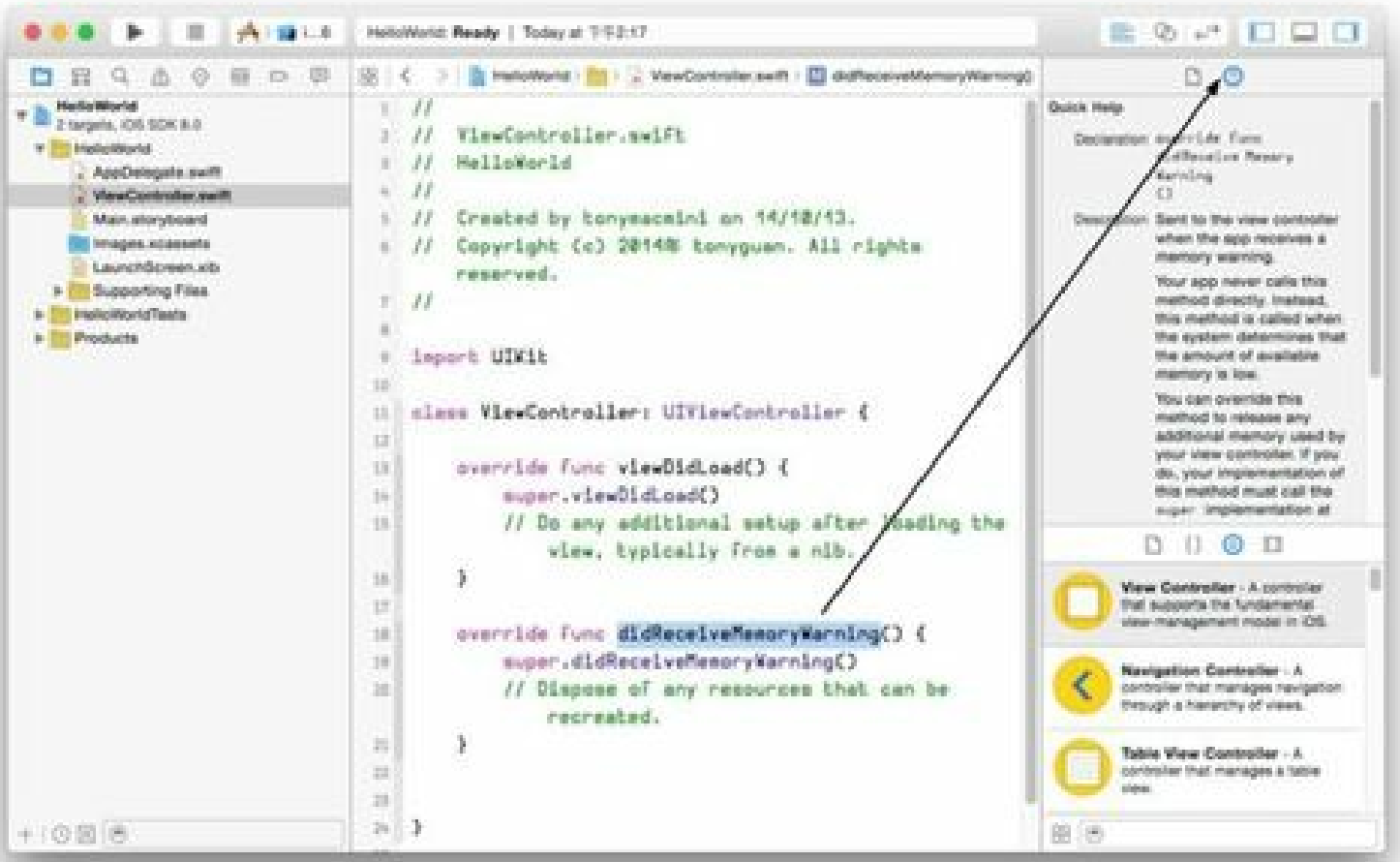


图 1-16　Xcode快捷帮助检查器

在打开的Xcode快捷帮助检查器窗口中，可以看到该方法的描述，其中包括使用的iOS版本、相关主题以及一些相关示例。这里需要说明的是，如果需要查看官方的示例，直接从这里下载即可。

如果想查询比较完整的、全面的帮助文档，可

以按住Alt键双击`didReceiveMemoryWarning`方法名，这样就会打开一个Xcode API帮助搜索结果窗口，如图1-17所示。然后选择感兴趣的主题，进入API帮助界面，如图1-18所示。

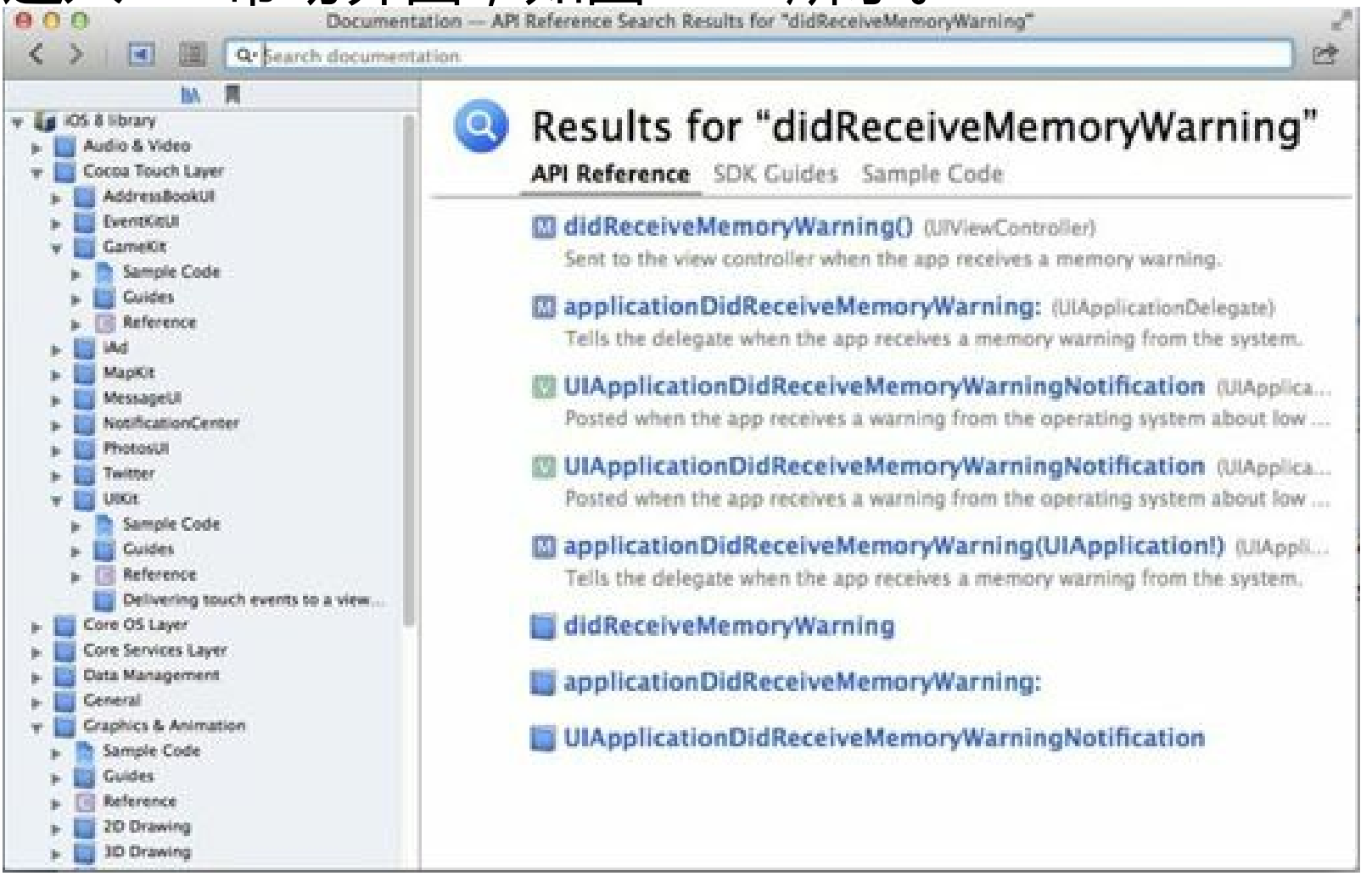


图 1-17　Xcode API帮助搜索结果窗口

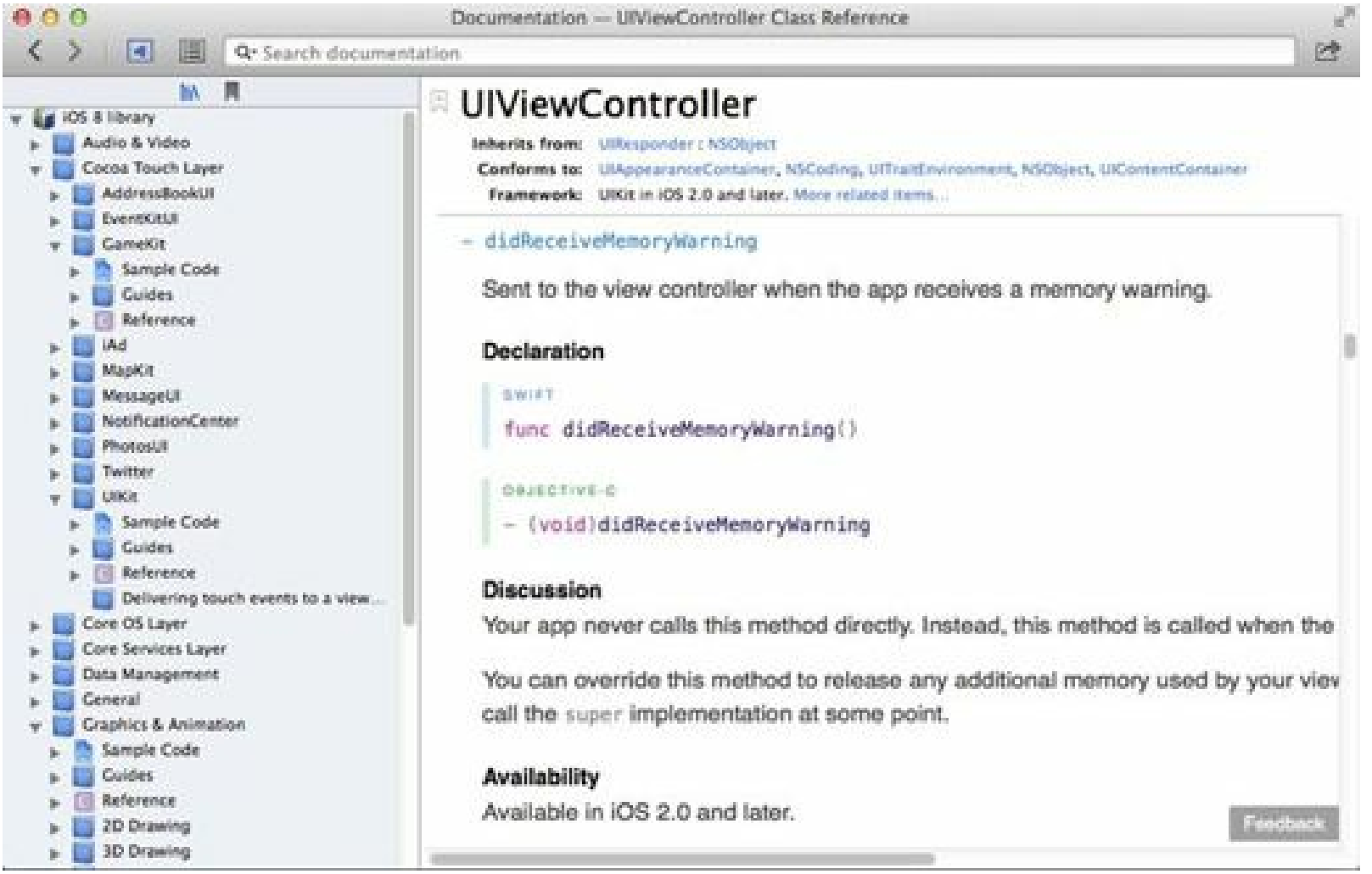


图 1-18　Xcode API帮助界面

API帮助文档还提供给我们一些官方示例，在左边的导航面板中可以找到相关的Sample Code，如图1-19所示。单击展开“Sample Code”找到相关示例工程并单击，在右边内容窗口中可以看到关于该示例的描述，如果单击“Open Project”按钮，就可以打开并下载这个示例工程。

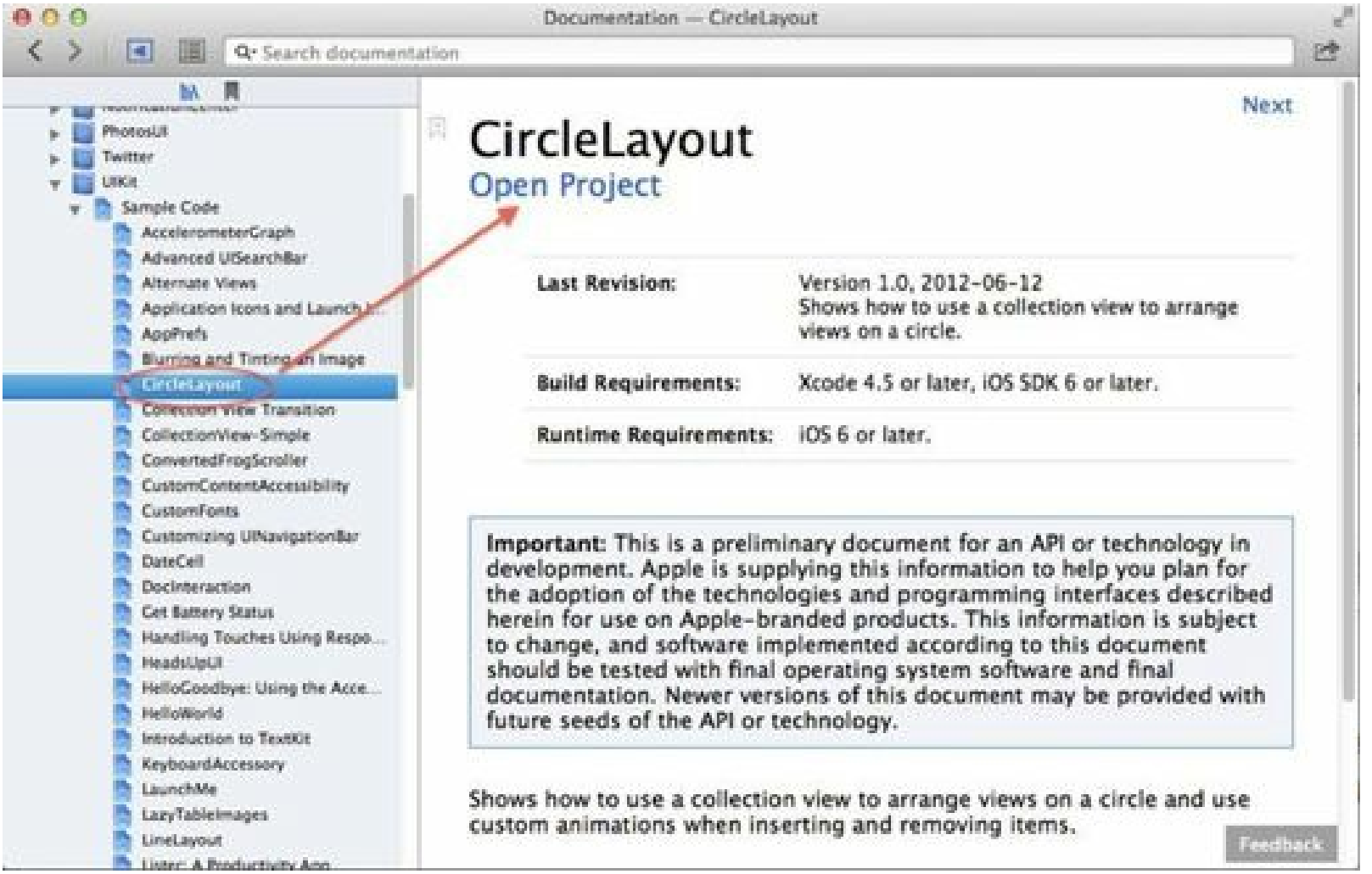

Documentation — CircleLayout
Search documentation
PhotosUI
Twitter
UIKit
Sample Code
AccelerometerGraph
Advanced UISearchBar
Alternate Views
AppPrefs
CircleLayout
CollectionView-Simple
ConvertedFrogScroller
CustomFonts
DateCell
HeadsUpUI
HelloWorld
KeyboardAccessory
LaunchMe
LazyTableImages
LineLayout
Next
CircleLayout
Open Project
Last Revision:
Version 1.0, 2012-06-12
Shows how to use a collection view to arrange views on a circle.
Build Requirements:
Xcode 4.5 or later, iOS SDK 6 or later.
Runtime Requirements:
iOS 6 or later.
Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.
Shows how to use a collection view to arrange views on a circle and use custom animations when inserting and removing items.
Feedback

图 1-19　官方案例

1.5　本章小结

通过对本章内容的学习，我们可以了解到本书的学习路线图，熟悉本书中的一些约定，便于对本书的学习，还可以掌握Xcode开发工具的安装和卸载，并熟练使用API文档。

1.6　同步练习

1. 介绍说明Xcode界面中各个区域的作用。
2. 请使用Xcode中的API帮助文档，找UIViewController关键字的相关帮助信息。

第 2 章　第一个Swift程序

从控制台输出“Hello World”是我学习C语言的第一步，也是我人生中非常重要的一步。多年后的今天，我仍希望以HelloWorld作为第一步，与大家共同开启一个神奇、瑰丽的世界——Swift编程。

本章以HelloWorld作为切入点，向大家系统介绍如何使用Xcode的Playground编写和运行Swift程序代码。

2.1 使用Playground编写

编写和运行Swift程序有多种方式，我们可以通过在Xcode中创建一个iOS或Mac OS X工程来实现，也可以通过使用Xcode6提供的Playground来实现。在学习阶段，我推荐大家使用Playground工具编写和运行Swift程序。

2.1.1 编程利器Playground

Playground离不开Xcode6，它是苹果在Xcde6中添加的新功能。使用Xcode创建工程编写和运行程序，目的是为了使最终的程序编译和发布，而使用Playground的目的是为了学习、测试算法、验证想法和可视化地看到运行结果。

图2-1所示是一个Playground程序运行界面，其中①区域是代码编写区域，②区域是运行结果区域，③区域是时间轴（timeline）区域。时间轴可以查看程序从上到下按照时间运行的结果，不同时间阶段运行的结果可以通过文本、图形和曲线图表等方式展示给开发人员。

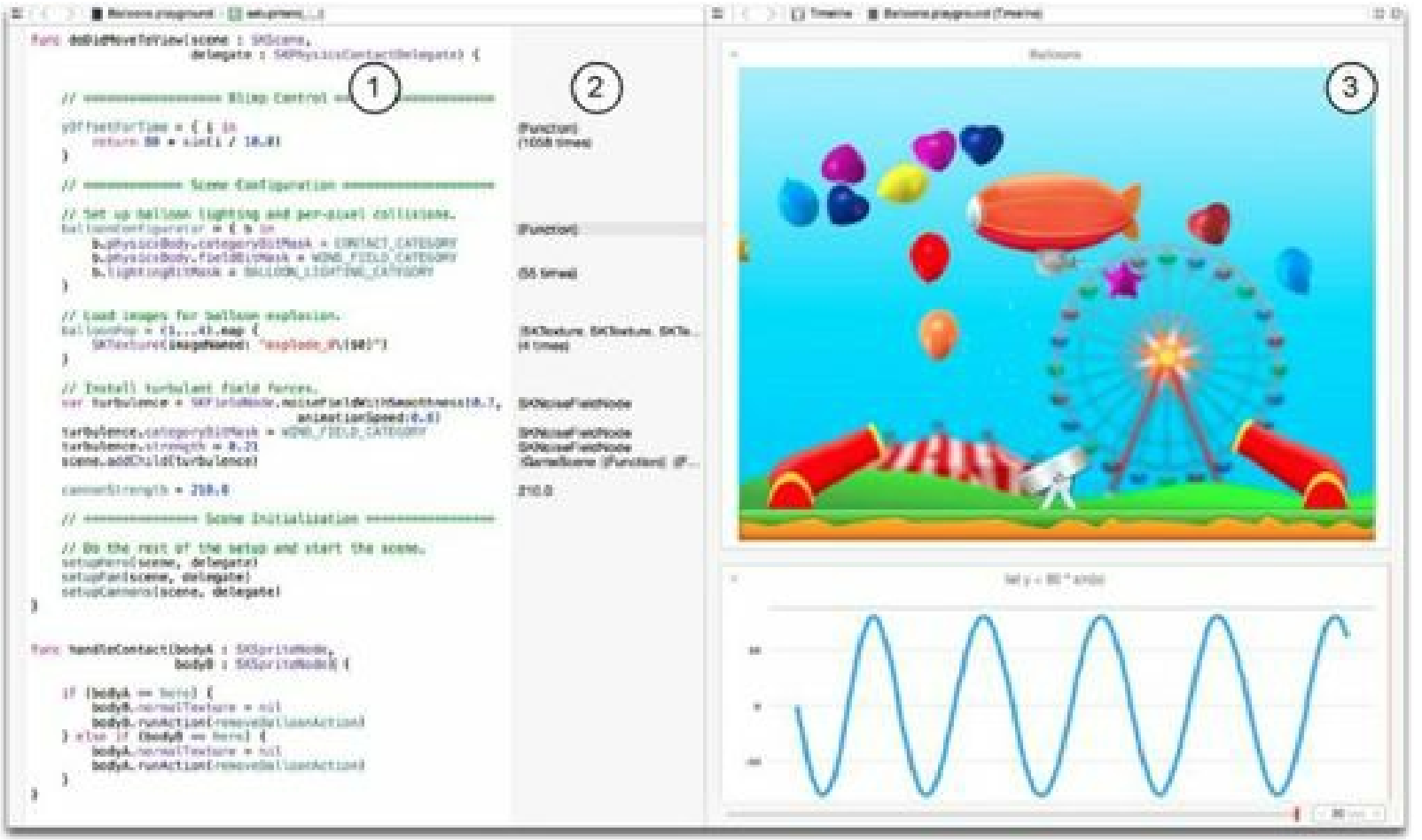


图 2-1　Playground界面

2.1.2　编写HelloWorld程序

下面我们具体介绍如何使用Playground编写HelloWorld程序。首先，打开Xcode6的欢迎界面（如图2-2所示）。一般第一次启动Xcode6就可以看到这个界面，如果没有，可以通过菜单Windows→Welcome to Xcode打开。

图 2-2　Xcode6欢迎界面

在图2-2所示的欢迎界面中，单击“Get started with a playground”弹出如图2-3所示的对话框，在Name中输入文件名“MyPlayground”，这是我们要保存的文件名，在Platform中选择“iOS”，然后单击“Next”按钮，弹出图2-4所示的界面，保存文件对话框。完成之后可以单击“Create”按钮创建Playground，创建成功后界面如图2-5所示。

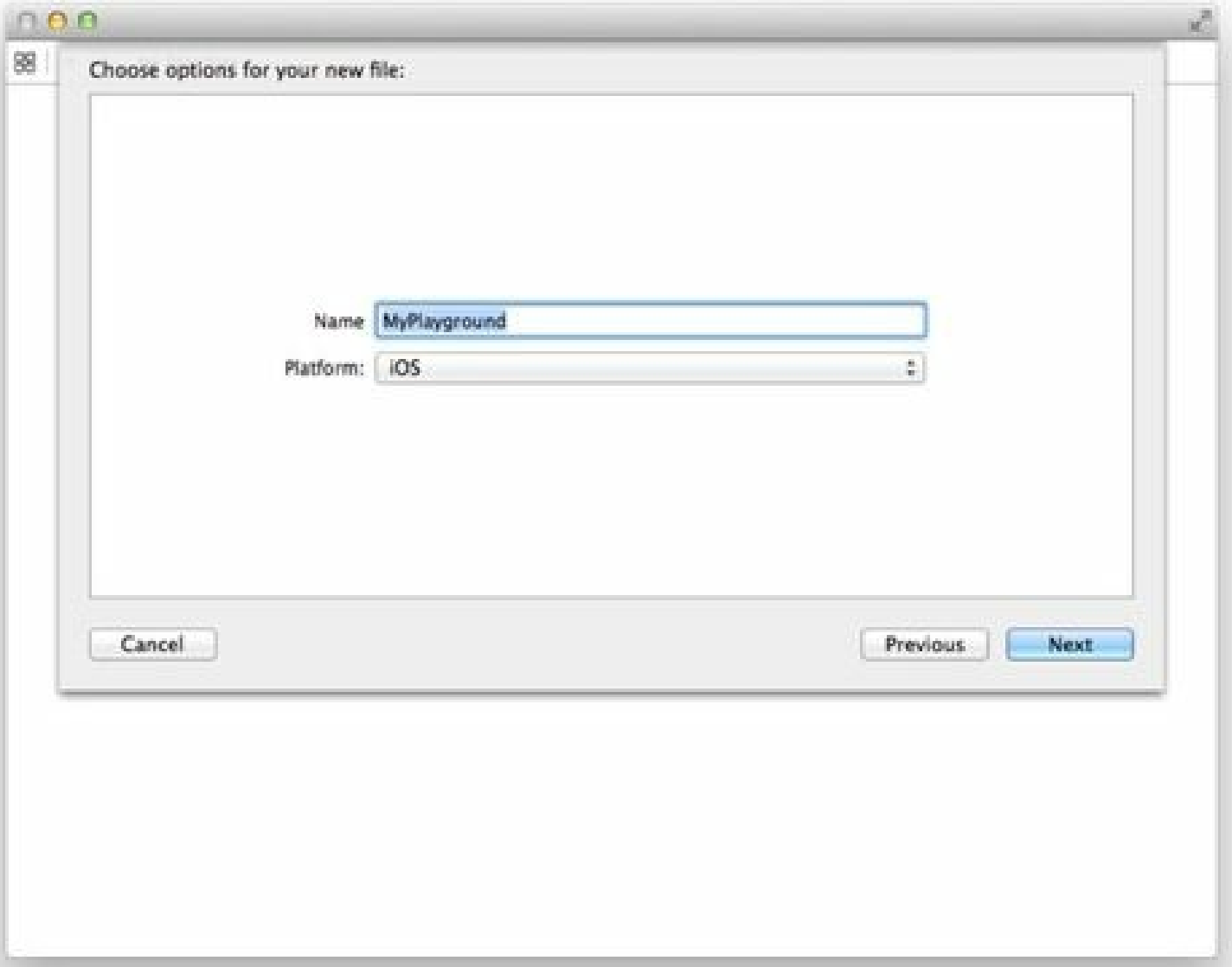

Choose options for your new file:
Name
MyPlayground
Platform:
iOS
Cancel
Previous
Next

图 2-3　输入文件名

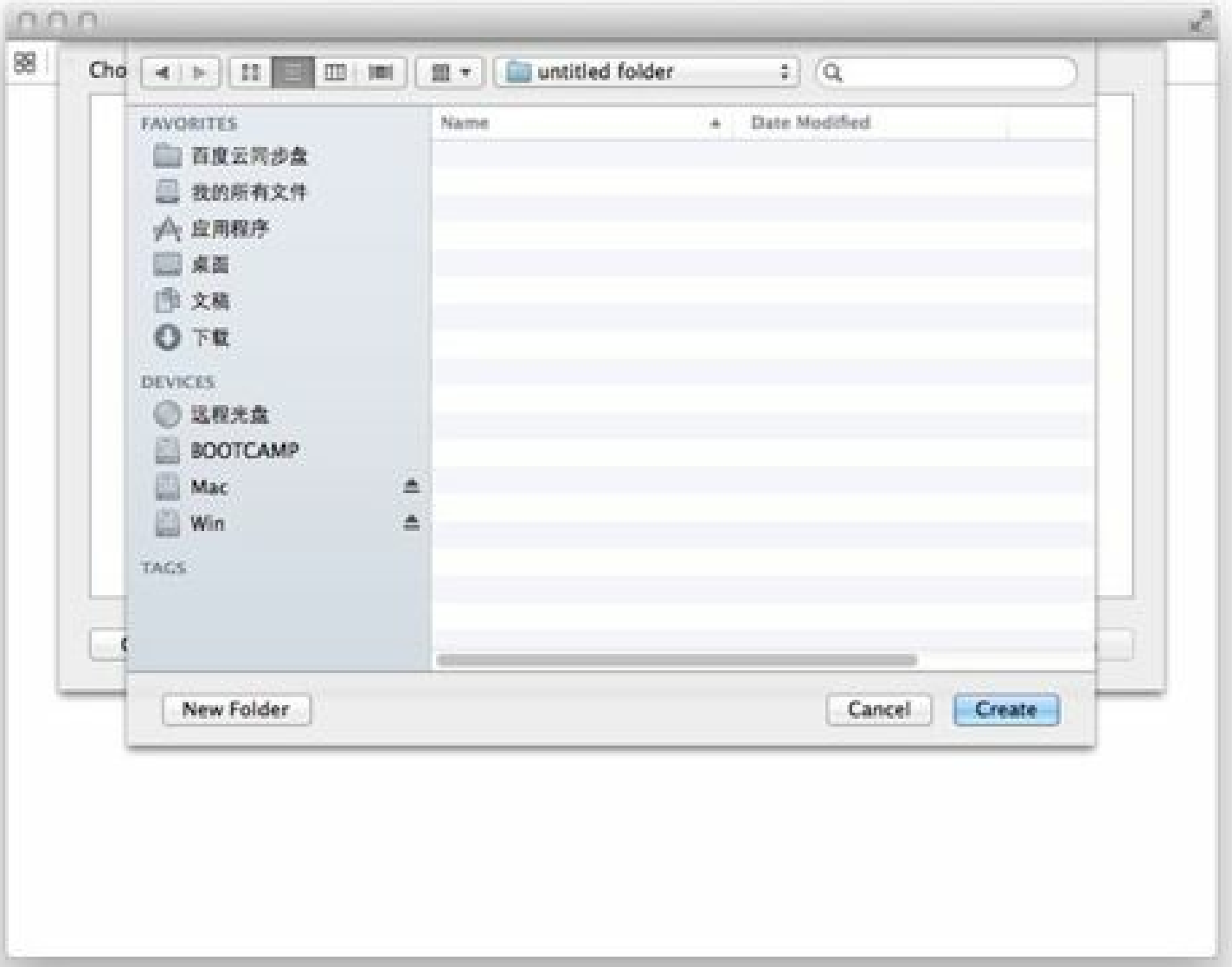


图 2-4　选择保存文件目录

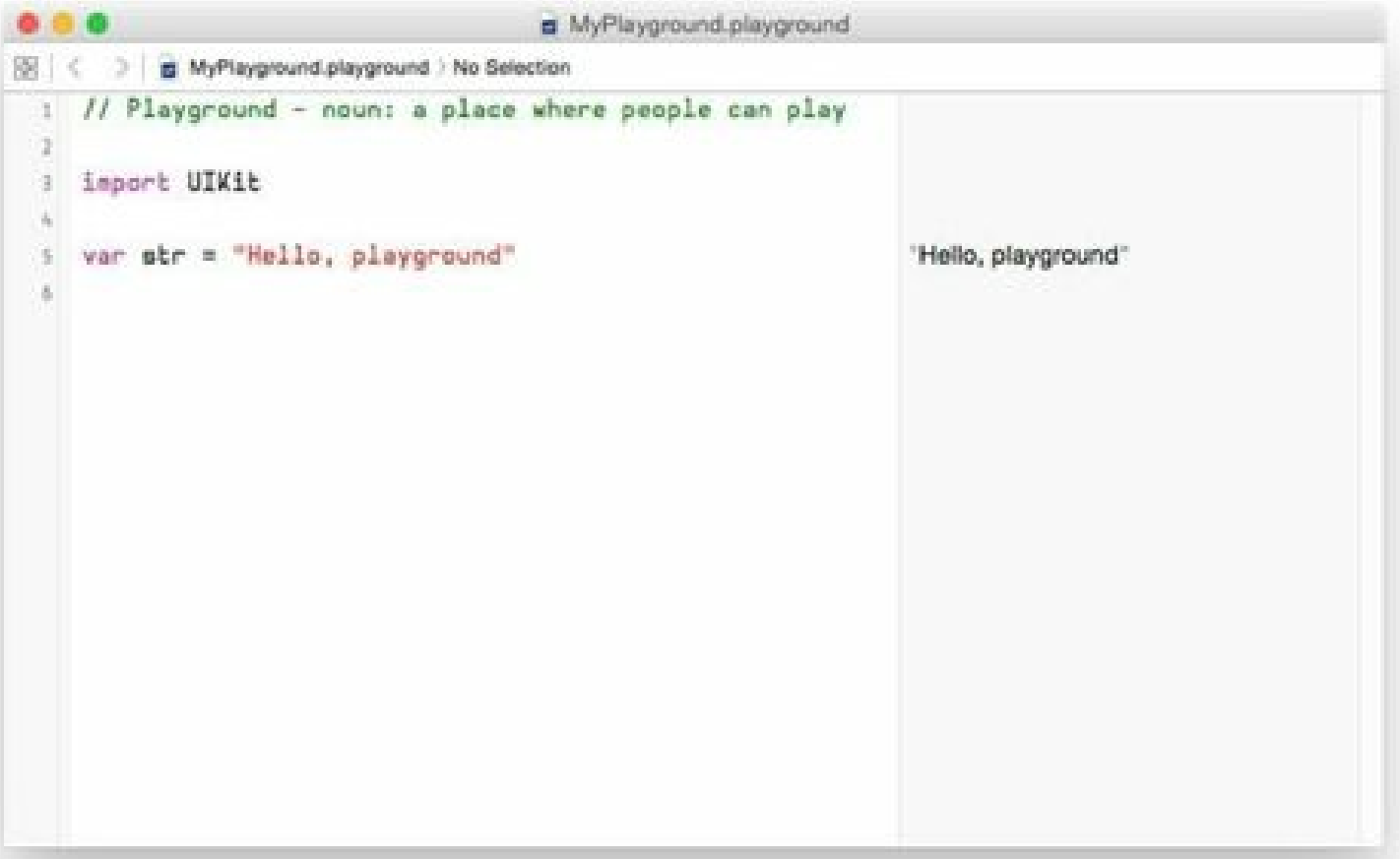


图 2-5　新建Playground界面

我们在图2-5所示的界面就可以编辑了，其中模板已经生成了一些代码，修改代码如下：

```
import UIKit

var str = "Hello World"
println(str)
```

代码修改完成后，马上就会编译运行，但是我

们在右边只能看到str变量情况，不能看到println输出结果，如图2-6所示。此时可以单击“Hello World”后面的“Value History”按钮 ，打开时间轴，如图2-7所示。

图 2-6　修改后的Playground界面

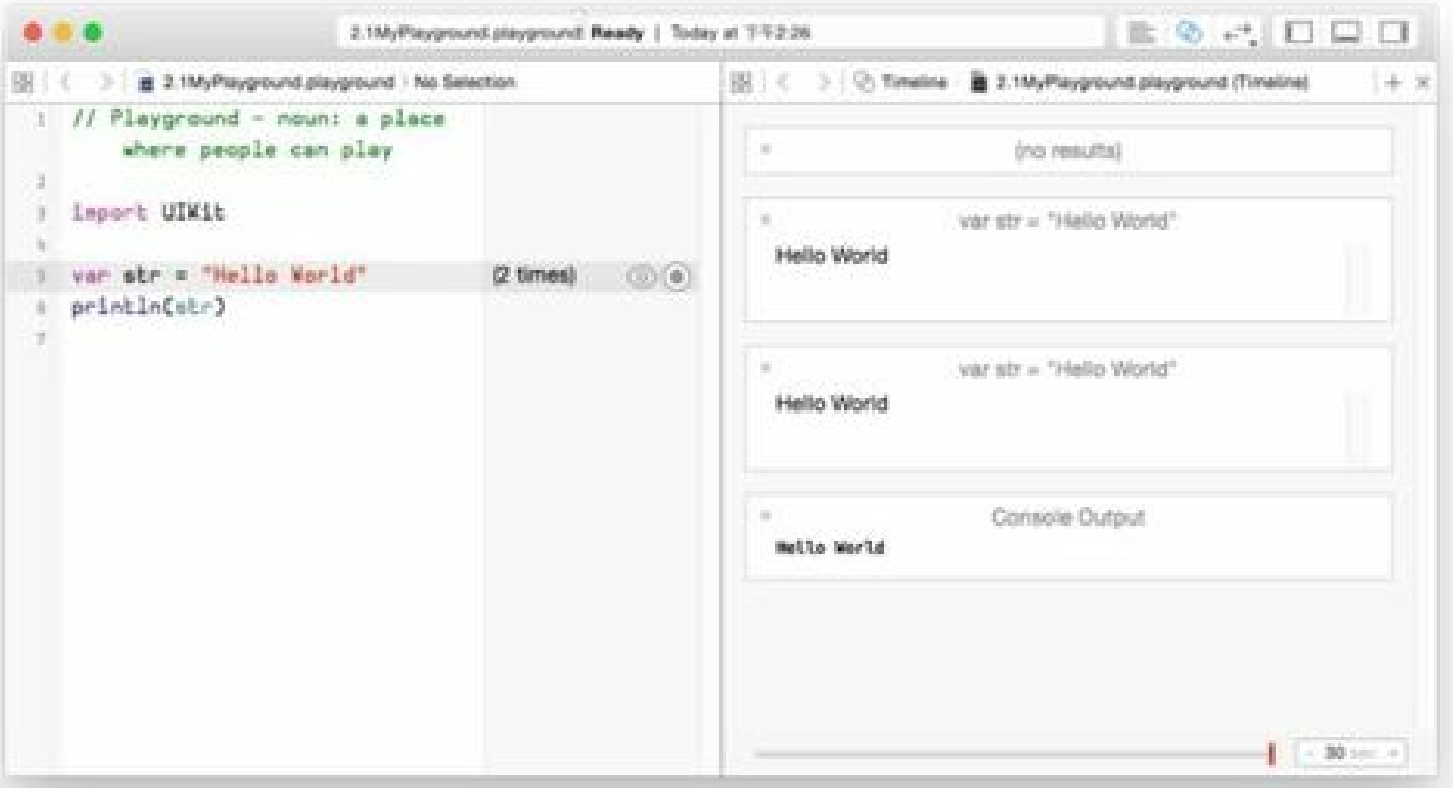


图 2-7　打开Playground时间轴界面

从图2-7所示的时间轴可以看到输出的结果。其中，Console Output是`println`函数的输出结果。

我们还可以通过助手编辑器打开Playground时间轴界面，具体操作过程如图2-8所示。右键单击标题栏，在菜单中选中“Icon and Text”，然后在出现的工具栏中，单击 打开助手编辑器。在助手编辑器中也有时间轴界面。

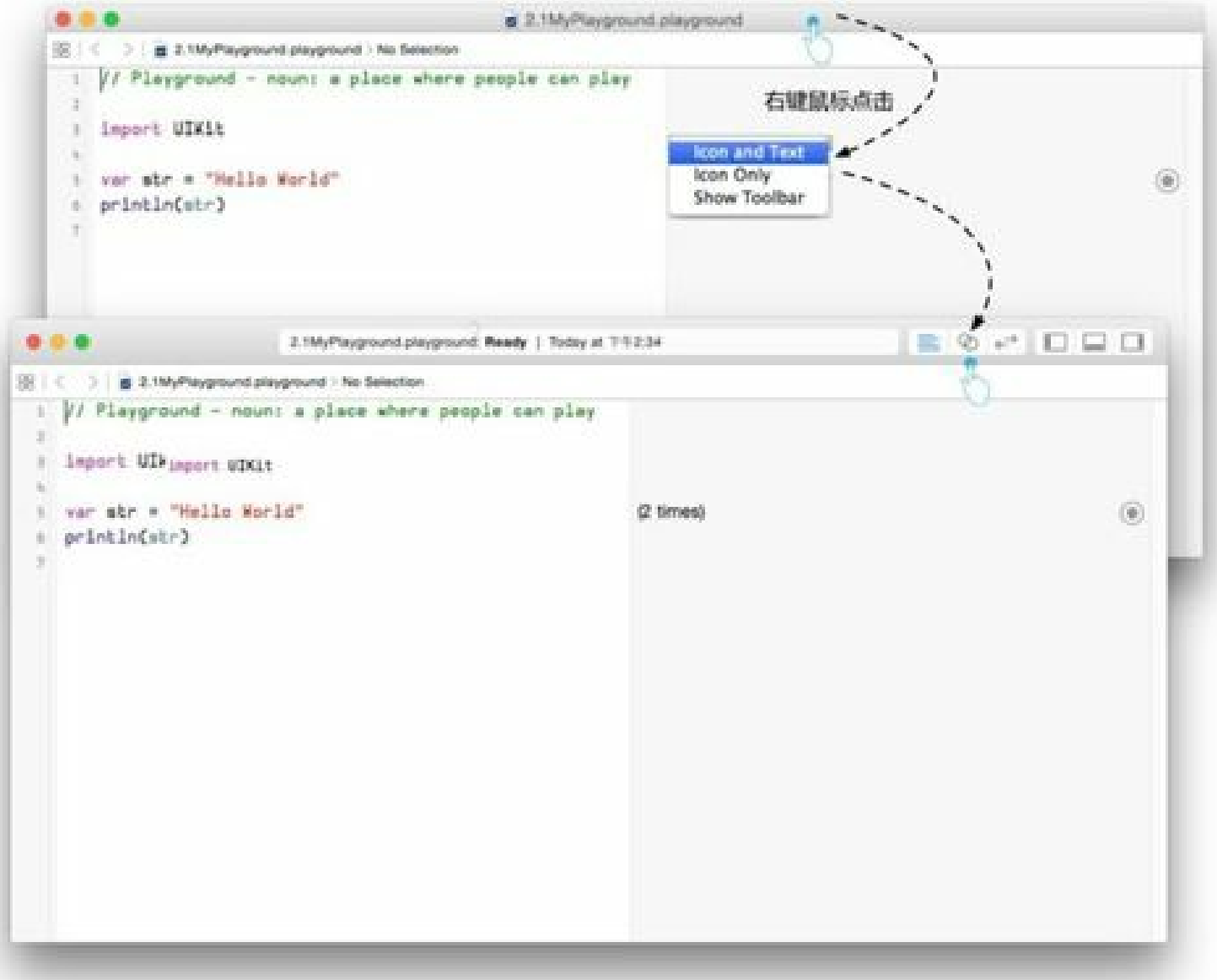


图 2-8　通过助手编辑器打开Playground时间轴界面

2.2 代码解释

Swift实现HelloWorld的方式比C和Objective-C等语言的实现要简单得多，下面我们详细解释一下代码。

1. **import UIKit语句**

import UIKit表示引入UIKit框架，类似于Objective-C中的#import和C中的#include。至于后面引入何种UIKit框架，就需要我们查找API来确定了。就本例而言，我们根本不需要UIKit框架，不过导入也没关系。

2. **var str = "Hello World"**

声明str变量，var表示声明变量。在var中并不能看出变量是什么类型，但Swift可以通过赋值的类型推断出变量的类型。由于我们赋值的是"Hello World"字符串，因此可知str是字符串变量。我们还应该注意到语句结束时没有出现像C和Objective-C等语言结束时的分号（;）。

3. **println(str)**

println是一个函数，能够将变量或量输出到控制台，类似于C中的println函数和Objective-C中的NSLog函数。有关格式化输出的问题我们会在后面再介绍。

这样我们通过短短的3行代码实现了一个HelloWorld输出的功能，事实上我们还可以写得更少。

2.3　本章小结

通过对本章内容的学习，我们可以了解到如何使用Xcode的Playground编写和运行Swift程序代码，了解Swift的程序结构，并熟悉Playground工具的使用。

2.4　同步练习

1. 介绍说明Playground界面中各个区域的作用。

2. 请使用Xcode的Playground编写一个输出`Hello Swift`字符串的Swift程序，并解释代码的含义。

第 3 章　Swift基本语法

本章主要为大家介绍Swift的一些基本语法，其中包括标识符和关键字、常量、变量、表达式和注释等内容。

3.1　标识符和关键字

任何一种计算机语言都离不开标识符和关键字，下面我们将详细介绍Swift标识符和关键字。

3.1.1　标识符

标识符就是给变量、常量、方法、函数、枚举、结构体、类、协议等指定的名字。构成标识符的字母均有一定的规范，Swift语言中标识符的命名规则如下：

- 区分大小写，Myname与myname是两个不同的标识符；
- 标识符首字符可以以下划线（_）或者字母开始，但不能是数字；
- 标识符中其他字符可以是下划线（_）、字母或数字。

例如，identifier、userName、User_Name、_身高等为合法的标识符，而2mail、room#和class为非法的标识符。其中，使用中文“身高”命名的变量是合法的。

注意 Swift中的字母采用的是Unicode编码[1]。Unicode叫做统一编码制，它包含了亚洲文字编码，如中文、日文、韩文等字符，甚至是我们在聊天工具中使用的表情符号，如  等，这些符号事实上也是Unicode，而非图片。这些符号在Swift中都可以使用。

> [1]Unicode是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。Unicode用数字0-0x10FFFF来映射这些字符，最多可以容纳1 114 112个字符，或者说有1 114 112个码位。——百度百科http://baike.baidu.com/view/2602518.htm

如果一定要使用关键字作为标识符，可以在关键字前后添加重音符号（`），例如：

```
let π = 3.14159

let $Hello = "Hello"

let 您好 = "你好世界"

let `class` = "😊😊😊"

//诺亚方舟
let 🌍 = "🐶🐮🐱🐭🐹🐰🐸🐯🐨🐻🐷🐽🐵🐔🐧🐦🐤🐣🐥🐺🐗🐴🦄🐝🐛🐌🐞🐜🕷🐢🐍🐙🐠🐟🐡🐬🐳🐋🐊🐆🐅🐃🐂🐄🐪🐫🐘🐐🐏🐑🐎🐖🐀🐁🐓🦃🕊🐕🐩🐈🐇🐿🐾🐉🐲"
```

其中`class`是关键字，事实上重音符号（`）不是标识符的一部分，它也可以用于其他标识符，如`π`和`π`是等价的。因此使用关键字作为标识符是

一种很不好的编程习惯。

3.1.2 关键字

关键字是类似于标识符的保留字符序列，除非用重音符号（`）将其括起来，否则不能用作标识符。关键字是对编译器具有特殊意义的预定义保留标识符。常见的关键字有以下4种。

- 与声明有关的关键字：class、deinit、enum、extensio
  static、struct、subscript、typeal
  和var。
- 与语句有关的关键字：break、case、continue、defaul
  for、return、switch、where和
  while。
- 表达式和类型关键字：as、dynamicType、is、new、supe
  __FILE__、__FUNCTION__和
  __LINE__。
- 在特定上下文中使用的关键字：associativity、didSet、get、i

nonmutating、operator、override、
unowned、unowned(safe)、unowned(
和willSet。

对于上述关键字，目前我们没有必要全部知道它们的含义，但是要记住：在Swift中，关键字是区分大小写的，因此class和Class是不同的，当然Class不是Swift的关键字。

3.2 常量和变量

我们在上一章中介绍了如何使用Swift编写一个HelloWorld小程序，其中就用到了变量。常量和变量是构成表达式的重要组成部分。

3.2.1 常量

在声明和初始化变量时，在标识符的前面加上关键字let，就可以把该变量指定为一个常量。顾名思义，**常量**是其值在使用过程中不会发生变化的量，实例代码如下：

```
let_Hello = "Hello"
```

_Hello标识符就是常量，只能在初始化的时候被赋值，如果我们再次给_Hello赋值，代码如下：

```
_Hello = "Hello, World"
```

则程序会报错，如图3-1所示，时间轴中显示了错误信息。

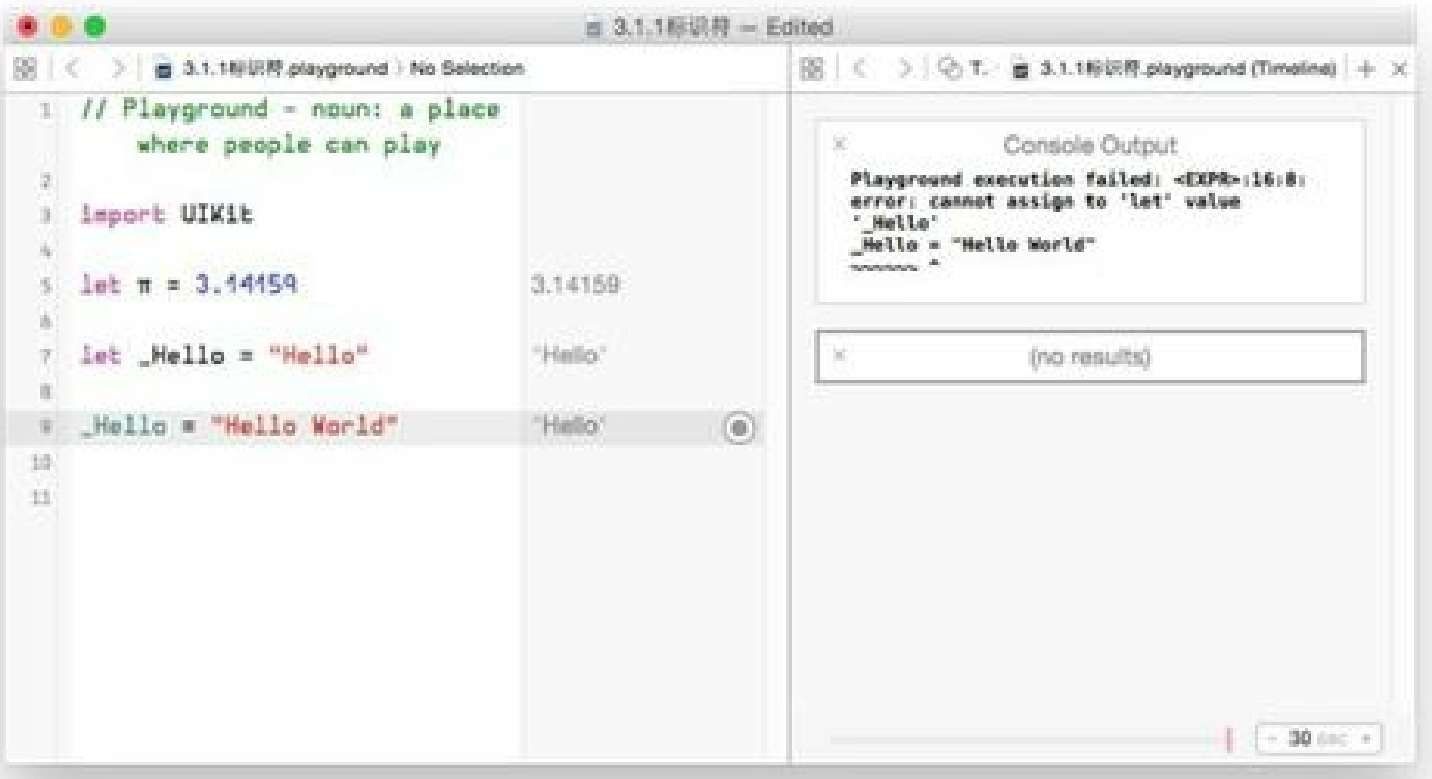


图 3-1　时间轴中错误信息

从错误信息可以获知_Hello是let分配的值，不能被赋值。

3.2.2　变量

在Swift中声明变量，就是在标识符的前面加上关键字var，实例代码如下：

```
var scoreForStudent = 0.0
```

该语句声明Double类型scoreForStudent

变量，并且初始化为0.0。如果在一个语句中声明和初始化了多个变量，那么所有的变量都具有相同的数据类型：

```
var x = 10, y = 20
```

在多个变量的声明中，我们也能指定不同的数据类型：

```
var x = 10, y = true
```

其中x为整型，y为布尔型。

3.2.3 命名规范

在使用常量和变量的时候，要保证它们的命名符合规范，这样程序才具有良好的可读性。这也是一种良好的编程习惯。

1. **常量名**

基本数据类型的常量名全为大写，如果由多个单词构成，则可以用下划线隔开，例如：

```
let YEAR = 60
let WEEK_OF_MONTH = 3
```

2. **变量名**

变量的命名有多种风格，主要以清楚易懂为主。有些程序员为了方便，使用单个字母来作为变量名称，如j和i等，这会为日后程序维护带来困难，变量同名的概率也会增加。单个字母变量一般只用于循环变量，因为它们只作用于循环体内。

在过去，计算机语言对变量名称的长度会有所限制，但现在已经没有这种限制了，因此我们鼓励用清楚的名称来表明变量的作用，通常会以小写字母作为开始，其余单词首字母大写，例如：

```
var maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

这样的名称可以令变量的作用一目了然。

除了常量和变量的命名要规范之外，其他语言对象的命名也要规范。其中类名、协议名、结构体、枚举等类型的命名规范通常是，大写字母作为开始，其余单词首字母大写，例如类

名HelloWorldApp。

函数和方法名往往由多个单词合成，第一个单词通常为动词，以小写字母作为开始，其余单词首字母大写，例如balanceAccount和isButtonPressed。

3.3 注释

Swift程序有两类注释：单行注释（//）和多行注释（/*...*/）。注释方法与C、C++和Objective-C语言都是类似的，下面详细介绍一下。

1. **单行注释**

单行注释可以注释整行或者一行中的一部分，一般不用于连续多行的注释文本。当然，它也可以用来注释连续多行的代码段。以下是两种注释风格的例子：

```
if x > 1 {
    //注释1
} else {
    return false //注释2
}

//if x > 1 {
//    //注释1
//} else {
//    return false //注释2
//}
```

提示 在Xcode中对连续多行的注释文本

可以使用快捷键：选择多行然后按住“command+/”键进行注释。去掉注释也是按住“command+/”键。

2. **块注释**

一般用于连续多行的注释文本，但也可以对单行进行注释。以下是几种注释风格的例子：

```
if x > 1 {
    /* 注释1
} else {
    return false  注释2
}


if x > 1 {
    //注释1
} else {
    return false //注释2
}



if x > 1 {
    /* 注释1
} else {
    return false  注释2 /
```

```
}
/
```

提示　Swift多行注释有一个其他语言没有的优点，就是可以嵌套，上述示例的最后一种情况便实现了多行注释嵌套。

在程序代码中，对容易引起误解的代码进行注释是必要的，但应避免对已清晰表达信息的代码进行注释。需要注意的是，频繁的注释有时反映了代码的低质量。当你觉得被迫要加注释的时候，不妨考虑一下重写代码使其更清晰。

3.4 表达式

表达式是程序代码的重要组成部分，在Swift中，表达式有3种形式。

1. **不指定数据类型**

```
var a1 = 10
let a2 = 20
var a = a1 > a2 ? "a1" : "a2"
```

在上述代码中，我们直接为变量或常量赋值，并没有指定数据类型，因为在Swift中可以自动推断数据类型。

2. **指定数据类型**

```
var a1:Int  = 10
let a2:Int = 20
var a = a1 > a2 ? "a1" : "a2"
```

在上述代码中，`:Int`是为变量和常量指定数据类型。这种写法使程序可读性良好，我们推荐明确指定变量和常量的数据类型。

3. **使用分号**

```
var a1:Int = 10; var a2:Int = 20
var a = a1 > a2 ? "a1" : "a2"
```

在Swift语言中，一条语句结束后可以不加分号也可以添加分号，但是有一种情况必须要用分号，那就是多条语句写在一行的时候，需要通过分号来区别语句。例如：

```
var a1:Int = 10; var a2:Int = 20;
```

3.5　本章小结

通过对本章内容的学习，我们可以了解到Swift语言的基本语法，其中包括标识符和关键字、常量、变量、表达式和注释等内容。

3.6 同步练习

1. 下列是Swift合法标识符的是（　　）。

A. `2variable`　　B. `variable2`　　C. `_whatavariable`　　D. `_3_`

E. `$anothervar`　　F. `#myvar`　　G. 体重　　H. 

I. `class`

2. 下列不是Swift关键字的是（　　）。

A. `if`　　B. `then`　　C. `goto`　　D. `while`

E. `case`　　F. `__COLUMN__`　　G. `where`　　H. `Class`

3. 描述下列代码的运行结果。

```
let Hello1 = "Hello"          ①
Hello1 = "Hello, World"       ②
println(_Hello1)               ③
var Hello2 = "Hello"          ④
Hello2 = "Hello, World"       ⑤
println(_Hello2)               ⑥
```

4. 下列有关Swift注释使用正确的是（　　）。

A.

```
if x > 1 {
      //注释1
 } else {
      return false //注释2
 }
```

B.

```
//let Hello1 = "Hello"
//Hello1 = "Hello, World"
//println(_Hello1)
```

C.

```
/*
let Hello1 = "Hello"
Hello1 = "Hello, World"
println(_Hello1)
*/
```

D.

```
/**
let Hello1 = "Hello"
Hello1 = "Hello, World"
println(_Hello1)
*/
```

5. 下列表达式不正确的是（　　）。

A. `var n1:Int = 10;`

B. `var n1:Int = 10`

C. `var n1 = 10`

D. `var n1:Int = 10; var str:String = 20`

E. `var n1:Int = 10; var str:String = "20"`

F. `var n1:Int = 10; var str:String = '20'`

第 4 章　基本运算符

本章主要为大家介绍一些Swift语言的基本运算符，包括算术运算符、关系运算符、逻辑运算符、位运算符和其他运算符。

4.1　算术运算符

Swift中的算术运算符用来组织整型和浮点型数据的算术运算，按照参加运算的操作数的不同可以分为**一元运算符**和**二元运算符**。

4.1.1　一元运算符

算术一元运算一共有3个，分别是-、++和--。-a是对a取反运算，a++或a--是在表达式运算完后，再给a加一或减一。而++a或--a是先给a加一或减一，然后再进行表达式运算。具体说明参见表4-1。

表4-1　一元算术运算

运算符	名称	说明	例子
-	取反符号	取反运算	b=-a
++	自加一	先取值再加一，或先加一再取值	a++或++a
--	自减一	先取值再减一，或先减一再取值	a--或--a

下面我们来看一个一元算数运算符的示例：

```
var a = 12
println(-a)                    ①

var b = a++                    ②
println(b)

b = ++a                        ③
println(b)
```

输出结果如下：

```
-12
12
14
```

上述代码第①行是-a，是把a变量取反，结果输出是-12。第②行代码是把a++赋值给b变量，先赋值后++，因此输出结果是12。第③行代码是把++a赋值给b变量，先++后赋值，因此输出结果是14。

4.1.2 二元运算符

二元运算符包括：+、 -、 *、 /和%，这些运算符对整型和浮点型数据都有效，具体说明参见表4-2。

表4-2　二元算术运算

运算符	名称	说明	例子
+	加	求a加b的和，还可用于String类型，进行字符串连接操作	a+b
-	减	求a减b的差	a-b
*	乘	求a乘以b的积	a*b
/	除	求a除以b的商	a/b
%	取余	求a除以b的余数	a%b

下面我们来看一个二元算数运算符的示例：

```
//声明一个整型变量
var intResult = 1 + 2
println(intResult)
```

```
intResult = intResult - 1
println(intResult)

intResult = intResult  2
println(intResult)

intResult = intResult  2
println(intResult)

intResult = intResult + 8
intResult = intResult % 7
println(intResult)

println("-------")
/声明一个浮点型变量
var doubleResult = 10.0
println(doubleResult)

doubleResult = doubleResult - 1
println(doubleResult)

doubleResult = doubleResult  2
println(doubleResult)

doubleResult = doubleResult / 2
println(doubleResult)

doubleResult = doubleResult + 8
doubleResult = doubleResult % 7
```

```
println(doubleResult)
```

输出结果如下：

```
3
2
4
2
3
-------
10.0
9.0
18.0
9.0
3.0
```

上述例子中分别对整型和浮点型进行了二元运算，具体语句不再赘述。

4.1.3 算术赋值运算符

算术赋值运算符只是一种简写，一般用于变量自身的变化，具体说明参见表4-3。

表4-3 算术赋值符

运算符	名称	例子
+=	加赋值	a+=b，a+=b+3
-=	减赋值	a-=b
*=	乘赋值	a*=b
/=	除赋值	a/=b
%=	取余赋值	a%=b

下面我们来看一个算术赋值运算符的示例：

```
var a = 1
var b = 2
a += b          // 相当于 a = a + b
println(a)

a += b + 3      // 相当于 a = a + b + 3
println(a)
a -= b          // 相当于 a = a - b
println(a)

a = b          // 相当于 a=ab
println(a)

a = b          / 相当于 a=a/b
```

```
println(a)

a %= b          // 相当于 a=a%b
println(a)
```

输出结果如下：

```
3
8
6
12
6
0
```

上述例子中分别对整型进行了+=、-=、*=、/=和%=运算，具体语句不再赘述。

4.2 关系运算符

关系运算是比较两个表达式大小关系的运算，它的结果是true或false，即布尔型数据。如果表达式成立则结果为true，否则为false。关系运算符有8种：==、!=、>、<、>=、<=、===和!==，具体说明参见表4-4。

表4-4 关系运算符

运算符	名称	说明	例子	可应用的类型
==	等于	a等于b时返回true，否则false。==与=的含义不同	a==b	整型，浮点型，字符型串，布尔型等值类

				型比较
!=	不等于	与==恰恰相反	a!=b	整型，浮点型，字符型串，布尔型等值类型比较
>	大于	a大于b时返回true，否则false	a>b	整型，浮点型，字符型串
				整

<	小于	a小于b时返回true，否则false	a<b	型，浮点型，字符型串
>=	大于等于	a大于等于b时返回true，否则false	a>=b	整型，浮点型，字符型串
<=	小于等于	a小于等于b时返回true，否则false	a<=b	整型，浮点型，字符型串
===	恒等于	a与b同引用同一个实例时返回true，否则false。===与==的含义不同。==是比较两	a===b	主要用于引用数

		个引用的内容是否是同一个实例		据比较
!=	不恒等于	与===恰恰相反	a!==b	主要用于引用数据比较

下面我们来看一个关系运算符的示例：

```
var value1 = 1
var value2 = 2
if value1 == value2 {
    println("value1 == value2")
}

if value1 != value2 {
    println("value1 != value2")
}

if value1 > value2 {
    println("value1 > value2")
}
```

```
if value1 < value2 {
    println("value1 < value2")
}

if value1 <= value2 {
    println("value1 <= value2")
}

let name1 = "world"
let name2 = "world"
if name1 == name2 {                    ①
    println("name1 == name2")
}

let a1 = [1,2]      //数组类型常量
let a2 = [1,2]      //数组类型常量

if a1 == a2 {                          ②
    println("a1 == a2")
}

if a1 === a2 {                         ③
    println("a1 === a2")
}
```

输出结果如下：

```
value1 != value2
value1 < value2
value1 <= value2
name1 == name2
a1 == a2
```

上述例子中，第①行是比较两个字符串内容是否相等，注意字符串String类型不能使用===进行比较，因为String不是引用类型。第②行代码是比较两个数组内容是否相等，我们会发现结果是相等的。第③行代码是比较两个数组是否相等（即是否是同一个实例），我们会发现结果是不相等的。

提示 ===和!==一般不用于引用类型之外的其他类型的比较。由于Swift中引用类型只有类，所以===和!==一般只比较类的实例，而结构体不是引用类型，一般不能比较，但是Array（数组）结构体是例外的一个，它虽然是结构体，但它可以使用===和!==进行比较。所以上述代码第③行并不会有编译错误，如果我们将第②行代码改为===进行比较，则

会发生编译错误。

4.3 逻辑运算符

逻辑运算符是对布尔型变量进行运算，其结果也是布尔型，具体说明参见表4-5。

表4-5 关系运算符

运算符	名称	例子	说明	可应用的类型
!	逻辑反	!a	a为true时，值为false，a为false时，值为true	布尔型
&&	短路与	a&&b	ab全为true时，计算结果为true，否则为false	布尔型
||	短路或	a||b	ab全为false时，计算结果为false，否则为true	布尔型

&&和||都具有短路计算的特点。例如x &&

y，如果 x 为false，则不计算 y（因为不论 y 为何值，“与”操作的结果都为false）。而对于x || y，如果 x 为true，则不计算 y（因为不论 y 为何值，“或”操作的结果都为true）。

所以，我们把&&称为“短路与”、把||称为“短路或”的原因就是，它们在计算过程中就像电路短路一样采用最优化的计算方式，从而提高了效率。

为了进一步理解它们的区别，我们看看下面的例子：

```
var i = 0
var a = 10
var b = 9

if (a > b) || (i++ == 1) {// 换成 | 试一下      ①
    println("或运算为 true")                    ②
} else {
    println("或运算为 false")                   ③
}

println("i = \(i)")                              ④

i = 0;

if (a < b) && (i++ == 1) {// 换成 & 试一下      ⑤
```

```
    println("与运算为 true")                    ⑥
} else {
    println("与运算为 false")                   ⑦
}

println("i = \(i)")                             ⑧
```

上述代码运行输出结果如下：

```
或运算为 true
i = 0
与运算为 false
i = 0
```

其中，第①行代码是进行短路或计算，由于(a > b)是true，后面的表达式(i++ == 1)不再计算，因此结果i不会加一，第④行输出的结果为i = 0。如果我们把第①行短路或换成逻辑或，结果则是i = 1。

类似地，第⑤行代码是进行短路与计算，由于(a < b)是false，后面的表达式(i++ == 1)不再计算，因此结果i不会加一，第⑧行输出的结果为i = 0。如果我们把第⑤行短路与换成逻辑

与，结果则是`i = 1`。

4.4 位运算符

位运算是以二进位（bit）为单位进行运算的，操作数和结果都是整型数据。位运算符有如下几个运算符：&，|，^，~，>>，<<，具体说明参见表4-6。

表4-6 位运算符

运算符	名称	例子	说明
~	位反	~x	将x的值按位取反
&	位与	x&y	x与y位进行位与运算
|	位或	x|y	x与y位进行位或运算
^	位异或	x^y	x与y位进行位异或运算
>>	右移	x>>a	x右移a位，无符号整数高位采用0补位，有符号整数高位采用符号位补位

<<	左移	x<<a	x左移a位，低位位补0

为了进一步理解它们，我们看看下面的例子：

```
let a: UInt8 = 0b10110010
①
let b: UInt8 = 0b01011110
②

println("a | b = \(a | b)") //11111110
③
println("a & b = \(a & b)") //00010010
④
println("a ^ b = \(a ^ b)") //11101100
⑤
println("a = \(a)")         //01001101
⑥

println("a >> 2 = \(a >> 2)")    //00101100
⑦
println("a << 2 = \(a << 2)")    //11001000
⑧

let c:Int8 = -0b1100
⑨
```

```
println("c >> 2 = \(c >> 2)")    //-00000011
⑩
println("c << 2 = \(c << 2)")    //-00110000
⑪
```

输出结果如下：

```
a | b = 254
a & b = 18
a ^ b = 236
~a = 77
a >> 2 = 44
a << 2 = 200
c >> 2 = -3
c << 2 = -48
```

上述代码中，我们在第①行和第②行分别定义了UInt8（无符号8位整数）变量a和b，0b01011110表示二进制整数，前面的0b前缀表示二进制。在第⑨行定义了c为Int8（有符号8位整数），它右位移的时候，高位使用符号位占位的。注意输出结果是十进制的。

代码第③行println("a | b = \(a | b)")是进行位或运算，结果是二进制的

11111110，它的运算过程如图4-1所示，从图中可见，a和b按位进行或计算，只要有一个为1，这一位就为1，否则为0。

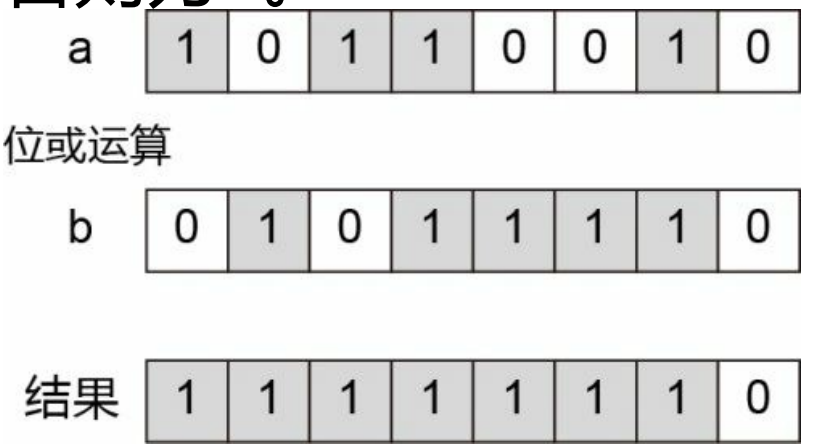


图 4-1　位或运算

代码第④行`println("a & b = \(a & b)")`是进行位与运算，结果是二进制的00010010，它的运算过程如图4-2所示，从图中可见，a和b按位进行与计算，只有两位全部为1，这一位才为1，否则为0。

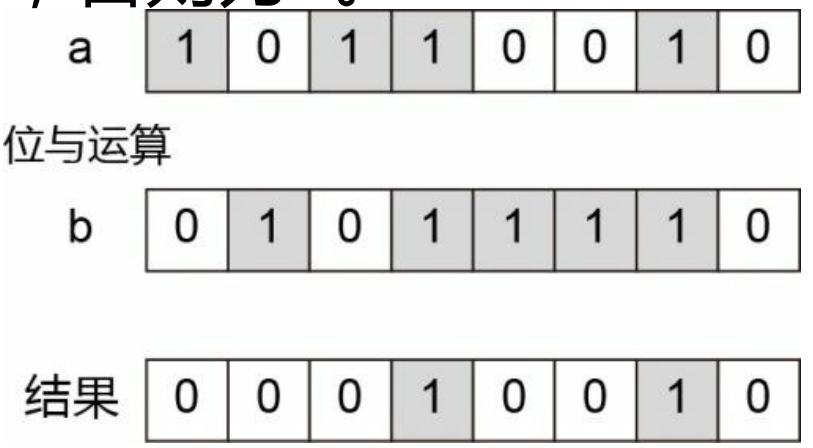


图 4-2　位与运算

代码第⑤行`println("a ^ b = \(a ^ b)")`是进行位异或运算，结果是二进制的

11101100，它的运算过程如图4-3所示，从图中可见，a和b按位进行异或计算，只有两位相反，这一位才为1，否则为0。

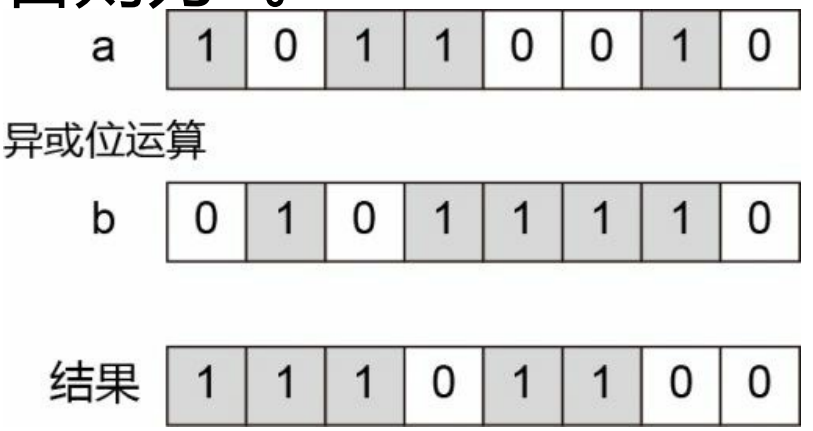


图 4-3　异或位运算

代码第⑦行`println("a >> 2 = \(a >> 2)")`是进行右位移2位运算，结果是二进制的00101100，它的运算过程如图4-4所示，从图中可见，a的低位被移除掉，高位用0补位。

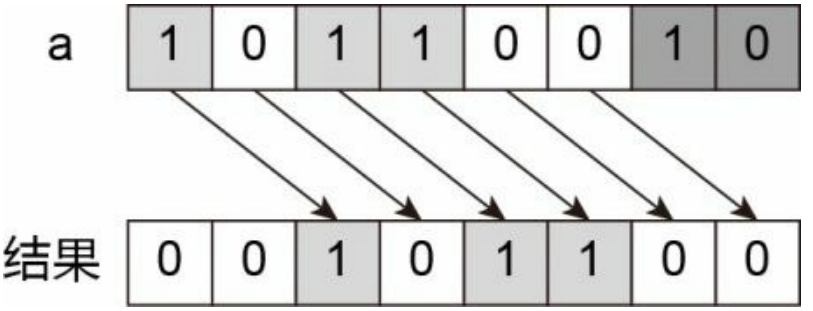


图 4-4　右位移2位运算

代码第⑧行`println("a << 2 = \(a << 2)")`是进行左位移2位运算，结果是二进制的11001000，它的运算过程如图4-5所示，从图中可见，a的高位被移除掉，低位用0补位。

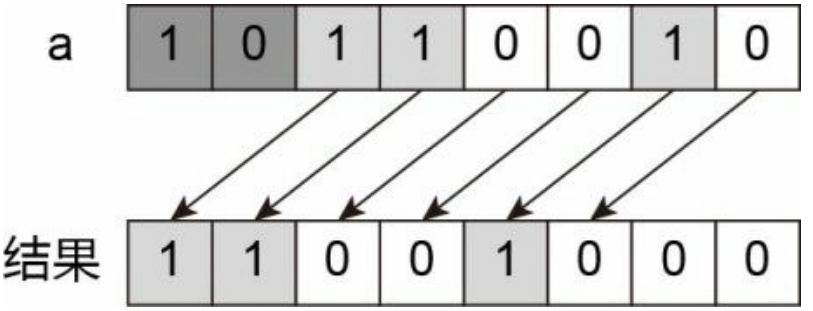


图 4-5　左位移2位运算

通过上面的详细解释，相信大家已经能够理解上述代码的运行结果了，其他的不再赘述。

4.5 其他运算符

除了前面介绍的主要运算符之外，还有一些其他运算符，如下所示。

- 三元运算符（？ ：）：例如x?y:z;，其中x，y和z都为表达式。
- 括号：起到改变表达式运算顺序的作用，它的优先级最高。
- 引用号（.）：实例调用属性、方法等操作符。
- 赋值号（=）：赋值时用等号运算符（=）进行的。
- 下标运算符[]。
- 箭头（->）：说明函数或方法返回值类型。
- 逗号运算符（，）：用于集合分割元素。
- 冒号运算符（：）：用于字典集合分割“键值”对。

示例代码如下：

```
let score: UInt8 = 80
let result = score > 60 ? "及格" : "不及格"
```

```
//三元运算符(? : )
println(result)

var arr = [93, 5, 3, 55, 57]     //使用逗号运算符
(,)
println(arr[2])                  //下标运算符[]

var airports = ["TYO": "Tokyo", "DUB":
"Dublin"] //使用冒号运算符(:)
```

其他运算符将在后面学习的过程中再给大家介绍。

4.6　本章小结

通过对本章内容的学习，我们可以了解到Swift语言的基本运算符，这些算符包括算术运算符、关系运算符、逻辑运算符、位运算符和其他运算符。

4.7 同步练习

1. 下列程序段执行后，t5的结果是（ ）。

```
var t1 = 9, t2 = 11, t3=8
var t4, t5 : Int

t4 = t1 > t2 ? t1 : t2+t1
t5 = t4 > t3 ? t4 : t3
```

A. 8　　B. 20　　C. 11　　D. 9

2. 下列程序段执行后，b，x，y的值分别是（ ）。

```
var x=6,y=8
var b : Bool
b = x > y && x++ == --y
```

A. true，6，8　　B. false，7，7　　C. true，7，7　　D. false，6，8

3. 下列执行程序段后，b，x，y的值分别是（ ）。

```
var x=6,y=8
var b : Bool
b = x++ == --y && x > y
```

A. true, 6, 8　　B. false, 7, 7　　C. true, 7, 7　　D. false, 6, 8

4. 设有定义var x=3.5, y=4.6, z=5.7，则以下的表达式中，值为true的是（　　）。

A. x > y || x > z　　B. x != y　　C. z > (y + x)　　D. x < y && !(x > z)

5. 下列程序段执行后，b3 的结果是（　　）。

```
var b1 = true, b2, b3 : Bool
b3 = b1 ? b1 : b2
```

A. 0　　B. 1　　C. true　　D. false　　E. 无法编译

6. 下列关于使用“<<”和“>>”操作符的结果正确的是（　　）。

A. `1010 0000 0000 0000 >> 4` 的结果是`0000 1010 0000 0000`

B. `1010 0000 0000 0000 >> 4` 的结果是`1111 1010 0000 0000`

C. `0000 1010 0000 0000 << 2` 的结果是`0010 1000 0000 0000`

D. `0000 1010 0000 0000 << 2` 的结果是`0000 1010 0000 0000`

7. 下列表达式中哪两个相对？

A. `16>>2`　　B. `16/2^2`　　C. `16*4`　　D. `16<<2`

8. 下列程序段执行后，输出结果是（　　）。

提示：`String`是结构体，它是值类型。

```
var a :String = "123", b :String = "123"
println(a === b)
a = b
println(a === b)
```

A. false B. false C. true D. true
false
true true false

E. 编译错误

9. 下列程序段执行后，输出结果是（ ）。

提示：NSString是类，它是引用数据类型。

```
var a :NSString = "123", b :NSString = "123"
println(a === b)
a = b
println(a === b)
```

A. false B. false C. true D. true E. 编译错误
false
true true false

10. 下列程序段执行后，输出结果是（ ）。

```
var a1 :String = "123", b1 :String = "123"
var a2 :NSString = "123", b2 :NSString = "123"
println(a1 == b1)
println(a2 !== b2)
```

A. `false`
`false`

B. `false`
`true`

C.
`true`
`false`

D. `true`
`true`

E. 编译错误

第 5 章　基本数据类型

我们在前面的章节中使用到了一些数据类型，例如UInt8、Int8和Double等，本章将详细介绍这些数据类型。Swift中包括了所有C和Objective-C语言中定义的数据类型，并且还有一些独有的数据类型，如元组（tuple）等。

5.1 Swift数据类型

Swift中的数据类型包括：整型、浮点型、布尔型、字符串、元组、集合、枚举、结构体和类等。

这些类型按照参数传递方式的不同可以分为：值类型和引用类型。**值类型**就是在赋值或给函数传递参数时候，创建一个副本，把副本传递过去，这样在函数的调用过程中不会影响原始数据。**引用类型**就是在赋值或给函数传递参数时候，把本身数据传递过去，这样在函数的调用过程中会影响原始数据。

在上述数据类型中，整型、浮点型、布尔型、字符串、元组、集合、枚举和结构体属于值类型，而类属于引用类型。

本章将重点介绍整型、浮点型、布尔型和元组等基本数据类型。

5.2　整型

Swift提供8、16、32、64位形式的有符号及无符号整数。这些整数类型遵循C语言的命名规约，如8位无符号整数的类型为UInt8，32位有符号整数的类型为Int32。我们归纳了Swift中的整型，参见表5-1。

表5-1　整型

数据类型	名称	说明
Int8	有符号8位整型	
Int16	有符号16位整型	
Int32	有符号32位整型	
Int64	有符号64位整型	
Int	平台相关有	在32位平台，Int与Int32宽度一致

	符号整型	在64位平台，Int与Int64宽度一致
UInt8	无符号8位整型	
UInt16	无符号16位整型	
UInt32	无符号32位整型	
UInt64	无符号64位整型	
UInt	平台相关无符号整型	在32位平台，UInt与UInt32宽度一致 在64位平台，UInt与UInt64宽度一致

除非要求固定宽的整型，否则一般我们只使用Int或UInt，这些类型能够与平台保持一致。

下面我们来看一个整型示例：

```
println("UInt8 range: \(UInt8.min) ~ \
```

```
(UInt8.max)")

println("Int8 range: \(Int8.min)   \(Int8.max)")

println("UInt range: \(UInt.min)   \(UInt.max)")

println("UInt64 range: \(UInt64.min)   \
(UInt64.max)")


println("Int64 range: \(Int64.min)   \
(Int64.max)")

println("Int range: \(Int.min) ~ \(Int.max)")
```

输出结果如下：

```
UInt8 range: 0 ~ 255
Int8 range: -128  127
UInt range: 0  18446744073709551615
UInt64 range: 0  18446744073709551615
Int64 range: -9223372036854775808
9223372036854775807
Int range: -9223372036854775808 ~
9223372036854775807
```

上述代码是通过整数的min和max属性计算各个类型的范围。min属性获得当前整数的最小值，max属性获得当前整数的最大值。由于程序运行的电脑是64位的，UInt运行的结果与UInt64相同，Int运行的结果与Int64相同。

我们在前面的学习过程中声明过变量，有时明确指定数据类型，有时则没有指定，例如下面的代码：

```
var ageForStudent = 30

var scoreForStudent:Int = 90
```

变量ageForStudent没有指定任何数据类型，但是我把30赋值给它，30表示Int类型30，因此ageForStudent类型就被确定为Int，这就是Swift提供的类型推断功能。此后我们就不能把非Int数值赋值给ageForStudent。如下代码是有编译错误的：

```
var ageForStudent = 30
```

```
ageForStudent = "20"
```

代码ageForStudent = "20"会发生编译错误，这是因为我们试图将20字符串赋值给Int类型的ageForStudent变量。

5.3 浮点型

浮点型主要用来储存小数数值，也可以用来储存范围较大的整数。它分为**浮点数**（float）和**双精度浮点数**（double）两种，双精度浮点数所使用的内存空间比浮点数多，可表示的数值范围与精确度也比较大。

下面我们归纳Swift中的浮点型，如表5-2所示。

表5-2 浮点型

数据类型	名称	说明
Float	32 位浮点数	不需要很多大的浮点数时候使用
Double	64 位浮点数	默认的浮点数

下面我们来看一个浮点型示例：

```
var myMoney:Float = 300.5;                ①
var yourMoney:Double = 360.5;             ②
```

```
let pi = 3.14159                                    ③
```

上述代码第①行明确指定变量myMoney是Float类型，第②行代码明确指定变量yourMoney是Double类型，第③行pi没有明确数据类型，但是我们给它赋值为3.14159，Swift编译器会自动推断出它是Double类型，注意不是Float类型，这是因为Double是默认浮点型，如果我们一定要使用Float类型，就不能使用自动推断，而是要在声明的时候明确指定Float类型。

5.4 数字表示方式

整型和浮点型都表示数字类型，那么在给这些类型的变量或常量赋值的时候，我们如何表示这些数字的值呢？前面我们曾使用30表示整数Int，使用3.14159表示浮点数Double。在计算机编程的时候，数字是比较丰富的，下面我们介绍一下进制数字表示和指数表示等方式。

5.4.1 进制数字表示

我们为一个整数变量赋值十进制数、二进制数、八进制数、十六进制数。它们的表示方式如下：

- 二进制数，以 0b 为前缀，0是阿拉伯数字，不要误认为是英文字母o，b是英文小写字母，不能大写；
- 八进制数，以0o为前缀，第一个字符是阿拉伯数字0，第二个字符是英文小写字母o，必须小写；
- 十六进制数，以 0x 为前缀，第一个字符是阿拉伯数字0，第二个字符是英文小写字母x，必须小写。

例如下面几条语句都是将整型28赋值给常量：

```
let decimalInt = 28
let binaryInt = 0b11100
let octalInt = 0o34
let hexadecimalInt = 0x1C
```

5.4.2 指数表示

我们在进行数学计算的时候往往会用到指数表示的数值。如果采用十进制表示指数，需要使用e（大写或小写的）表示幂，e2表示10^2。

采用十进制指数表示的浮点数示例如下：

```
var myMoney  = 3.36e2
var interestRate = 1.56e-2
```

其中3.36e2表示的是3.36×10^2，1.56e-2表示的是1.56×10^{-2}。

十六进制表示指数，需要使用p（大写或小写的）表示幂，与十进制不同的是，p2表示2^2。由于十六进制换算起来比较麻烦，因此我们推荐使用十

进制表示。

5.4.3 其他表示

在Swift中，为了阅读的方便，整数和浮点数均可添加多个零或下划线以提高可读性，两种格式均不会影响实际值。

示例代码如下：

```
var interestRate = 000.0156        ①
var myMoney  = 3_360_000           ②
```

代码第①行的000.0156浮点数前面添加了多个零，变量interestRate实际值还是0.0156浮点数。代码第②行的3_360_000中间添加了很多个下划线，变量myMoney实际值还是3360000整数，下划线一般是三位加一个。

5.5 数字类型之间的转换

Swift是一种安全的语言，对于类型的检查非常严格，不同类型之间不能随便转换。本节我们介绍数字类型之间的转换，其他类型之间的转换会在后面相关章节介绍。

5.5.1 整型之间的转换

在C和Objective-C等其他语言中，整型之间有两种转换方法：

- 从小范围数到大范围数转换是自动的；
- 从大范围数到小范围数需要强制类型转换，有可能造成数据精度的丢失。

而在Swift中这两种方法是行不通的，我们需要通过一些函数进行显式地转换，代码如下：

```
let historyScore:UInt8 = 90

let englishScore:UInt16 = 130

let totalScore = historyScore + englishScore
//错误                                  ①

let totalScore = UInt16(historyScore) +
```

```
englishScore    //正确                          ②

let totalScore = historyScore +
UInt8(englishScore)   //正确                    ③
```

上述代码声明和初始化了两个常量historyScore和englishScore，我们把它们相加赋值给totalScore。如果采用第①行代码实现相加，程序就会有编译错误，原因是historyScore是UInt8类型，而englishScore是UInt16类型，它们之间不能转换。

我们有两种转换方法。

一种是把UInt8的historyScore转换为UInt16类型。由于是从小范围数转换为大范围数，这种转换是安全的。代码第②行UInt16(historyScore)就是正确的转换方法。

另外一种是把UInt16的englishScore转换为UInt8类型。由于是从大范围数转换为小范围数，这种转换是不安全的，如果转换的数比较大会造成精度的丢失。代码第③

行UInt8(englishScore)是正确的转换方法。由于本例中englishScore的值是130，这个转换是成功的，如果把这个数修改为1300，虽然程序编译没有问题，但是会在控制台中输出异常信息，这是运行期异常。

上述代码中，UInt16(historyScore)和UInt8(englishScore)事实上是构造器，能够创建并初始化另外一个类型。关于构造器的内容，我们会在第14章详细介绍。

5.5.2 整型与浮点型之间的转换

整型与浮点型之间的转换与整型之间的转换类似，因此我们将上一节的示例修改如下：

```
let historyScore:Float = 90.6
①

let englishScore:UInt16 = 130
②

let totalScore = historyScore + englishScore
//错误                                  ③

let totalScore = historyScore +
Float(englishScore)  //正确，安全                    ④
```

```
let totalScore = UInt16(historyScore) +
englishScore  //正确，小数被截掉        ⑤
```

上述代码经过了一些修改，第①行代码historyScore变量类型是Float类型。第②行代码englishScore变量还是UInt16类型。其中第③行代码直接进行了计算，结果有编译错误。第④行代码是将UInt16类型的englishScore变量转换为Float类型，这种转换是最安全的。第⑤行代码是将Float类型的historyScore变量转换为UInt16类型，这种转换首先会导致小数被截掉，另外如果historyScore变量数很大，会导致运行期异常，这与整型之间的转换是类似的。

5.6 布尔型

布尔型（Bool）只有两个值：true和false。它不能像Objective-C一样使用1替代true，使用0替代false。

实例代码如下：

```
var is🐴 = true
var is🐼 : Bool = false
```

上述代码中，对于变量  和  的命名我们采用了Unicode编码，表现形式上是一个图像，事实上在计算机内部存储的是Unicode编码。与整型和浮点型等其他类型一样，如果没有指定数据类型，Swift可以自动推断类型。

我们可以在if语句中直接使用布尔表达式，代码如下：

```
if (is🐴) {
    println("是的，它是马。")
} else {
    println("不，它是熊猫！")
}
```

5.7 元组类型

元组（tuple）这个词很抽象，它是一种数据结构，在数学中应用广泛。在计算机科学中，元组是关系数据库中的基本概念，元组表中的一条记录，每列就是一个字段。因此在二维表里，元组也称为记录。

元组将多个相互关联组值合为单个值，便于管理和计算。元组内的值可以是任意类型，各字段类型不必相同。元组在作为函数返回多值时尤其有用。

假设我们要管理学生成绩，会定义一个Student元组，它包含学号（id）、姓名（name）、年龄（age）和分数（score）。那么使用Swift语法表示Student元组就是：

```
("1001", "张三", 30, 90)
(id:"1001", name:"张三", age:30, score:90)
```

这两种写法都表示一个叫“张三”的学生的元组，但是第一种写法代码可读性不好，如果不进行说明，或许你能猜出“1001”代表学号，“张三”代表学生姓名，但30和90代表的含义是什么

呢？而第二种写法一目了然，决不会引起困惑，显然代码可读性更好。通过“键-值对”表示更加直观，只不过要稍微多写一些代码。

下面我们来看一个示例：

```
var student1 = ("1001", "张三", 30, 90)
①
println("学生\(student1.1) 学号:\(student1.0) 年龄:\(student1.2)，得分:\(student1.3)")          ②

var student2 = (id:"1002", name:"李四", age:32, score:80)                                        ③
println("学生\(student2.name) 学号:\
(student2.id) 年龄:\(student2.age)，

得分:\(student2.score)")                                                                         ④

let (id1, name1,age1,score1) = student1
⑤

println("学生\(name1) 学号:\(id1) 年龄:\(age1)，得分:\(score1)")                                  ⑥

let (id2, name2,_,_) = student2
⑦

println("学生\(name2) 学号:\(id2)")
⑧
```

输出结果如下：

```
学生张三 学号:1001 年龄:30，得分:90
学生李四 学号:1002 年龄:32，得分:80
学生张三 学号:1001 年龄:30，得分:90
学生李四 学号:1002
```

上述代码第①行声明并初始化了元组类型的student1变量。第②行代码通过字段索引访问字段内容，其中student1.0访问student1的第一个字段，student1.1访问student1的第二个字段，依次访问下去，索引是从0开始的。

第③行代码声明并初始化了元组类型的student2变量，在这一行中采用了“键-值对”表示方式，为给每个字段定义一个名字，访问的时候比较方便。第④行代码通过字段名字访问其中内容，其中student2.name访问student2的name字段，student2.id访问student2的id字段。

第⑤行代码let (id1, name1,age1,score1) = student1事实上是

对student1元组变量的分解，元组变量student1被分解为4个不同的变量中，因此我们在第⑥行代码打印输出"学生张三 学号:1001 年龄:30，得分:90"。有的时候我们并不想分解那么多变量，而是只需要学号和姓名，那么就可以使用第⑦行代码的方式，把不需要的字段使用下划线（_）替代。第⑧行代码输出结果是"学生李四 学号:1002"。

5.8 本章小结

通过对本章内容的学习，我们可以了解到Swift语言的基本数据类型，包括UInt8、Int8和Double等，还有布尔型和元组等。此外，还介绍了数字的表示方式和数字类型之间的转换。

5.9 同步练习

1. 下列数据哪些是值类型？（　　）

A. 元组　　B. 枚举　　C. 结构体　　D. 类

2. 下列数据哪些是引用类型？（　　）

A. 字符串　　B. 枚举　　C. 结构体　　D. 类　　E. 集合

3. 下列说法正确的是（　　）。

A. `Int`是与平台相关有符号整型　　B. `UInt`是与平台相关无符号整型

C. `UInt16`是与平台相关无符号整型　　D. `Int8`是与平台无关的有符号整型

4. 下列表示数字正确的是（　　）。

A. 29　　B. 0X1C　　C. 0x1A　　D. 1.96e-2　　E. 9_600_000

5. 判断正误：Swift中的整数可以从小范围数到大范围数转换是自动的。

6. 判断正误：Swift中的整数从大范围数到小范围数需要强制类型转换，有可能造成数据精度的丢失。

7. 下列语句中能够正常运行的有（　　）。

A.

```
let f:UInt8 = 10.0
let i:UInt16 = 10
let total = UInt16(f) + i
println(total)
```

B.

```
let f:Double = 10.0
let i:UInt16 = 10
let total = UInt16(f) + i
println(total)
```

C.

```
let n:UInt8 = 90
let i:UInt16 = 10
let total = UInt16(n) + i
println(total)
```

D.

```
let n:UInt8 = 90
let i:UInt16 = 10
```

```
let total = UInt8(i) + n
println(total)
```

8. 请描述元组类型，并举例说明。

9. 假设有语句`var 张老师 = ("张三", 30)`，则下列语句有语法错误的是（　　）。

A. `let (name,age) = 张老师`

B. `println("\(张老师.0) \(张老师.1)")`

C. `println("\(张老师.name) \(张老师.age)")`

D. `var (name,age) = 张老师`

10. Swift中的布尔值表示正确的是（　　）。

A. `true`　　B. `false`　　C. `1`　　D. `0`

第 6 章　字符和字符串

由字符组成的一串字符序列，称为**字符串**，我们在前面的章节中也多次用到了字符串，本章将重点进行介绍。

6.1 字符

字符串的组成单位是字符，那么在Swift中什么能够算是字符，这个问题非常重要。

6.1.1 Unicode编码

Swift是一种现代计算机语言，它采用Unicode编码，它的字符几乎涵盖了我们所知道的一切字符。表示一个字符可以使用字符本身，也可以使用它的Unicode编码，特别是无法通过键盘输入的字符，使用编码还是很方便的。但是编码不是很容易记忆，这也是它的问题。

Unicode编码可以有单字节编码、双字节编码和四字节编码，它们表现形式是`\u{n}`，其中`n`为1～8个十六进制数。

下面我们看看示例：

```
let andSign1:Character = "&"
let andSign2 = "\x26"

let lamda1:Character = "λ"
let lamda2 = "\u03bb"

let smile1:Character = "😃"
let smile2 = "\U0001f603"
```

在Swift中，字符类型是Character，与其他类型声明类似，可以指定变量或常量类型为Character，也可以由编译器自动推断。常量andSign1和andSign2保存有&字符，它的Unicode编码是0026，属于单字节编码，使用\u{26}表示。常量lamda1和lamda2保存有λ字符，它是希腊字母莱姆达，它的Unicode编码是03bb，属于双字节编码，使用\u{03bb}表示。常量smile1和smile2保存有笑脸符号，注意不是图片，它的Unicode编码是0001f603，属于双字节编码，使用\u{0001f603}表示。这些编码实在是难以记住，我们可以在http://vazor.com/unicode/网站查询字符与编

码的对应关系。

提示　在C和Objective-C等语言中，字符是放在单引号（'）之间的，然而在Swift语言中，不能使用单引号的方式，必须使用双引号（"）把字符括起来。

6.1.2　转义符

在Swift中，为了表示一些特殊字符，会使用“\”，这称为**字符转义**。常见的转义符的含义参见表6-1。

表6-1　转义符

字符表示	Unicode编码	说明
\t	\u{0009}	水平制表符tab
\n	\u{000a}	换行
\r	\u{000d}	回车
\"	\u{0022}	双引号
\'	\u{0027}	单引号
\	\u{005c}	反斜线

下面我们看看示例：

```
let specialCharTab1 = "Hello\tWorld."
println("specialCharTab1: \(specialCharTab1)")

let specialCharNewLine = "Hello\nWorld."
println("specialCharNewLine: \
(specialCharNewLine)")

let specialCharReturn = "Hello\rWorld."
println("specialCharReturn: \
(specialCharReturn)")

let specialCharQuotationMark =
"Hello\"World\"."
println("specialCharQuotationMark: \
(specialCharQuotationMark)")

let specialCharApostrophe = "Hello\'World\'."
println("specialCharApostrophe: \
(specialCharApostrophe)")

let specialCharReverseSolidus = "Hello\\World."
println("specialCharReverseSolidus: \
(specialCharReverseSolidus)")
```

输出结果如下：

```
specialCharTab1: Hello World.
specialCharNewLine: Hello
World.
specialCharReturn: Hello
World.
specialCharQuotationMark: Hello"World".
specialCharApostrophe: Hello'World'.
specialCharReverseSolidus: Hello\World.
```

上述代码输出了几种特殊符号。

6.2 创建字符串

在Swift 中，字符串的类型是`String`，事实上`String`是一个结构体，有关结构体的知识我们会在11.4节介绍。

要创建一个字符串，可以直接赋值，也可以通过结构体的构造器创建。示例代码如下：

```
let 🌍 = "🐶🐴🐱🐭🐹🐰🐸🐯🐮🐷🐽🐵🙈🙉🙊🐒🐔🐧🐼
🐨🐤🐥🐣🐦🐲🐢🐍🐝🐛🐌🐞🐜🐙🐠🐟🐬🐳🐋🐎🐑
🐂🐃🐄🐪🐫🐘🐐🐓🐕🐖🐁🐀🐈🐉🐇"

println("已经上船的小动物数: \(countElements(🌍))")

let 熊: Character = "🐻"
let 猫: Character = "🐱"
let 🐼 = 熊 + 猫

let emptyString1 = ""
let emptyString2 = String();
```

由于Swift的标识符也都是Unicode编码，所以可以在上述代码中使用汉字（“熊”和“猫”）、  和  等有意思的符号作为常量名。除了直接赋值字符串，还可以通过使用加号（+）把多个字符拼接成一个字符串。`emptyString1`和

emptyString2常量被赋值为空的字符串，其中String()是调用了String的构造器，关于构造器我们会在第14章详细介绍。我们还可以通过countElements函数获得字符串中字符的个数，即字符串的长度。

6.3 字符串可变性

在Objective-C和Java等语言中，字符串有两种：不可变字符串和可变字符串，二者的区别是不可变字符串不能进行拼接、追加等修改，如果进行这种操作就会产生一个新的字符串对象。

Swift语言与此类似，通过为字符串变量声明为let和var实现不可变字符串和可变字符串，let声明的字符串是不可变的，var声明的是可变的。示例代码如下：

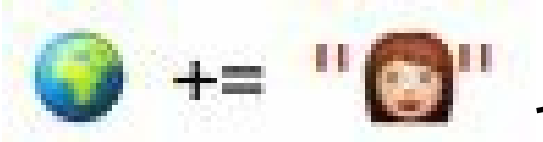
```
var 🌍 = "🐶🐴🐱🐭🐹🐰🐸🦁🐷🐻🐽🐮🐗🐵🐒🐴🐑🐘🐼🐧🐦🐤🐥
🐣🐔🐍🐢🐛🐝🐜🐞🐌🐙🐚🐠🐟🐬🐳🐋🐄🐏🐀🐃🐅🐇🐉🐎🐐🐓🐕🐖
🐁🐂🐲🐡🐊🐫🐪🐆🐈🐩🐾"

🌍 = 🌍 + "👦"
🌍 += "👲"

println("诺亚方舟乘客数: \(countElements(🌍))")


let number = 9
let total = "\(number) 加 10 等于 \(Double(number) + 10)"
```

首先我们声明了var字符串变量🌍，通过语句🌍 = 🌍 + "👦"进行字符串追加。字符串追加也可以使用+=运算符号，所以语句


也可以实现字符串追加。如果被声明为let的字符串常量，则追加操作不被允许。

有的时候需要将其他数据类型如整数、浮点数等数据也与字符串一起拼接，我们可以在字符串中通过\()实现。例如上述代码let total = "\(number) 加 10 等于 \(Double(number) + 10)"就实现了将整数和浮点数与字符串拼接起来。

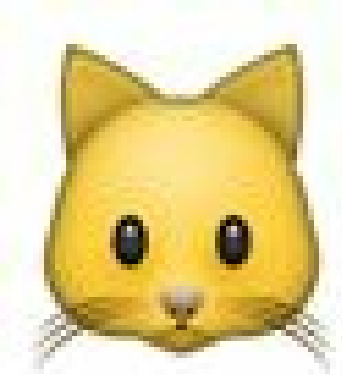
6.4　字符串比较

字符串比较涉及字符串大小和相等比较，以及字符串前缀和后缀的比较。

6.4.1　大小和相等比较

字符串类型与整型和浮点型一样，都可以进行相等以及大小的比较，比较的依据是Unicode编码值大小。例如下面两个字符：

Unicode：1F43C



Unicode：1F431

我们比较一下，由于1F43C要大于1F431，因此在比较时  大于  ，运行以下代码并查看结果。

```
let 熊: String = "🐻"
let 猫: String = "🐱"

if 熊 > 猫 {
    println("🐻 大于 🐱")
} else {
    println("🐻 小于 🐱")
}
```

运行的结果是：  大于  。

当然，比较动物的大小没有太大的实际意义，但是比较ABC等传统字符是有意义的。

提示 上述代码不能使用Character替换String类型，因为Character类型不支持>、<、>=和<=运算符。

与比较大小不同的是，我们需要比较字符串是否相等，需要注意的是，String和Character类型支持==和!=运算符，但是不支持===和!==运算符。让我们看看下面的代码：

```
let 🐼 = 熊 + 猫

if 🐼 == "🐻🐱"
    println("🐼 是 🐻🐱")
} else {
    println("🐼 不是 🐻🐱")
}

let emptyString1 = ""
let emptyString2 = string ()

if emptyString1 == emptystring 2 {
    println("相等")
} else {
    println("不相等")
}
```

在上述代码中，我们比较字符串变量🐼是否等于"🐻🐱"字符串，结果是🐼 是 🐻🐱，这个结果不用过多解释。代码中还比较了通过""和String()创建的两种空字符串是否相等，结果是它们也是相等的。

6.4.2 前缀和后缀比较

在字符串比较中，有时候需要比较前缀或后缀。例如，如果需要判断某个文件夹中特定类型的文件，就要判断它们的扩展名，这就需要判断它的后缀，我们可以使用字符串hasSuffix方法。如果需要判断某个文件夹中特定字符串开头的文件，

就可以使用字符串的hasPrefix方法来判断前缀。

以下代码实现的是文件的查找过程：

```
import UIKit
let docFolder = [
    "java.docx",
    "JavaBean.docx",
    "Objecitve-C.xlsx",
    "Swift.docx"
]
①

var wordDocCount = 0
for doc in docFolder {
②
    if doc.hasSuffix(".docx") {
③
        ++wordDocCount
    }
}
println("文件夹中Word文档个数是： \
(wordDocCount)")

var javaDocCount = 0
for doc in docFolder {

    let lowercaseDoc = doc.lowercaseString
```

```
④
    if lowercaseDoc.hasPrefix("java") {
⑤
        ++javaDocCount
    }
}
println("文件夹中Java相关文档个数是： \
(javaDocCount)")
```

上述代码第①行声明并初始化了数组变量docFolder，关于数组我们会在第7章介绍。第②行代码是使用for in语句遍历数组集合，关于for in语句我们会在7.2.4节详细介绍，这个过程就是从集合docFolder中取出一个元素保存在doc变量中。第③行代码中的doc.hasSuffix(".docx")语句是判断doc字符串的结尾是否是“.docx”，“.docx”是Word文档。第④行代码doc.lowercaseString属性是获得小写字符串，这样我们在判断前缀的时候直接判断是否为“java”就可以了。与lowercaseString属性类似的还有uppercaseString属性。

6.5　本章小结

通过对本章内容的学习，我们可以了解到Swift语言的字符和字符串，以及字符串可变性和字符的比较等内容。

6.6 同步练习

1. 关于Swift中的字符表示方式正确的是（　　）。

A. `"\u{0001f603}"`　　B. "😃"　　C. '\u{0001f603}'　　D. 😃

2. 请说明下面转义符代表的含义。

```
\t \n  \r  \"  \' \\
```

3. 下列表达式正确的是（　　）。

A. `let andSign1:Character = "&"`　　B. `let andSign2 = "\u{26}"`

C. `let lamda1:Character = "λ"`　　D. `var lamda2:String = "\u{03bb}"`

E. `let 熊: String = "🐻"`　　F. `let 猫: Character = "🐱"`

4. 判断正误：Character类型不支持===和!==运算符。String支持===和!==运算符。

5. 判断正误：Character和String类型都支持==和!=运算符。

第 7 章　控制语句

程序设计中的控制语句有3种，即顺序、分支和循环语句。Swift程序通过控制语句来执行程序流，完成一定的任务。程序流是由若干个语句组成的，语句可以是一条单一的语句，也可以是一个用大括号（{}）括起来的复合语句。Swift中的控制语句有以下几类：

- 分支语句：if和switch
- 循环语句：while、do while、for和for in
- 跳转语句：break、continue、fallthrough和return

7.1 分支语句

分支语句提供了一种控制机制，使得程序具有了“判断能力”，能够像人类的大脑一样分析问题。分支语句又称条件语句，条件语句使部分程序可根据某些表达式的值被有选择地执行。Swift编程语言提供了if和switch两种分支语句。

7.1.1 条件语句if

由if语句引导的选择结构有if结构、if else结构和else if结构3种。

1. if结构

如果条件表达式为true就执行语句组，否则就执行if结构后面的语句。与C和Objective-C语言不同的是，即便语句组是单句，也不能省略大括号。语法结构如下：

```
if 条件表达式 {
    语句组
}
```

if结构示例代码如下：

```
var score = 95

if score >= 85 {
    println("您真优秀！")
}

if score < 60 {
    println("您需要加倍努力！")
}

if score >= 60 && score < 85 {
    println("您的成绩还可以，仍需继续努力！")
}
```

程序运行结果如下：

```
"您真优秀！"
```

2. if else结构

所有的语言都有这个结构，而且结构的格式基本相同，语句如下：

```
if 条件表达式 {
    语句组1
} else {
```

```
    语句组2 }
```

当程序执行到if语句时，先判断条件表达式，如果值为true，则执行语句组1，然后跳过else语句及语句组2，继续执行后面的语句。如果条件表达式的值为false，则忽略语句组1而直接执行语句组2，然后继续执行后面的语句。

if else结构示例代码如下：

```
var score = 95

if score < 60 {
    println("不及格")
} else {
    println("及格")
}
```

程序运行结果如下：

```
"及格"
```

3. else if结构

else if结构如下：

```
if 条件表达式1 {
    语句组1
} else if 条件表达式2 {
    语句组2
} else if 条件表达式3 {
    语句组3
...
} else if 条件表达式n {
    语句组n
} else {
    语句组n+1
}
```

可以看出，else if结构实际上是if else结构的多层嵌套，它明显的特点就是在多个分支中只执行一个语句组，而其他分支都不执行，所以这种结构可以用于有多种判断结果的分支中。

else if结构示例代码如下：

```
let testscore = 76
var grade:Character

if testscore >= 90 {
```

```
    grade = "A"
} else if testscore >= 80 {
    grade = "B"
} else if testscore >= 70 {
    grade = "C"
} else if testscore >= 60 {
    grade = "D"
} else {
    grade = "F"
}
println("Grade = \(grade)")
```

输出结果如下：

```
Grade = C
```

其中，var grade:Character是声明字符变量，然后经过判断最后结果是C。

7.1.2 多分支语句switch

switch语句也称开关语句，它提供多分支程序结构。

Swift彻底地颠覆了自C语言以来大家对于

switch的认知，这个颠覆表现在以下两个方面。

一方面，在C、C++、Objective-C、Java甚至是C#语言中，switch语句只能比较离散的单个整数（或可以自动转换为整数）变量，而Swift中的switch语句可以使用整数、浮点数、字符、字符串和元组等类型，而且它的数值可以是离散的也可以是连续的范围。

另一方面，Swift中的switch语句case分支不需要显式地添加break语句，分支执行完成就会跳出switch语句。

下面我们先介绍一下switch语句基本形式的语法结构，如下所示：

```
switch 条件表达式{
    case 值1：
    语句组1
    case 值2，值3：
    语句组2
    case 值3：语句组3
        ...
    case 判断值n：
    语句组n
    default：
    语句组n+1
}
```

每个case后面可以跟一个或多个值，多个值之间用逗号分隔。每个switch必须有一个default语句，它放在所有分支后面。每个case中至少要有一条语句。

当程序执行到switch语句时，先计算条件表达式的值，假设值为A，然后拿A与第1个case语句中的值1进行匹配，如果匹配则执行语句组1，语句组执行完成就跳出switch，不像C语言那样只有遇到break才跳出switch；如果A没有与第1个case语句匹配，则与第2个case语句进行匹配，如果匹配则执行语句组2，以此类推，直到执行语句组n。如果所有的case语句都没有执行，就执行default的语句组n + 1，这时才跳出switch。

switch语句基本形式示例代码如下：

```
let testscore = 86

var grade:Character

switch testscore / 10 {                    ①
```

```
case 9:
    grade = "优"
case 8:
    grade = "良"
case 7,6:                                    ②
    grade = "中"
default:
    grade = "差"
}

println("Grade = \(grade)")
```

输出结果如下：

```
Grade = 良
```

上述代码将100分制转换为：“优”、“良”、“中”、“差”评分制。第①行计算表达式获得0~9分数值。第②行代码中的7,6是将两个值放在一个case。

Swift中的switch不仅可以比较整数类型还可以比较浮点和字符等类型。下面是一个比较浮点数的示例：

```
let value = 1.000

var desc:String

switch value {
case 0.0:
    desc = "最小值"
case 0.5:
    desc = "中值"
case 1.0:
    desc = "最大值"
default:
    desc = "其他值"
}

println("说明 = \(desc)")
```

输出结果如下：

```
说明 = 最大值
```

结果说明1.000是与1.0相等的。
下面是一个字符比较示例：

```
let level = "优"
```

```
var desc:String

switch level {
case "优":
    desc = "90分以上"
case "良":
    desc = "80分~90分"
case "中":
    desc = "70分~80分"
case "差":
    desc = "低于60分"
default:
    desc = "无法判断"
}

println("说明 = \(desc)")
```

输出结果：

```
说明 = 90分以上
```

7.1.3 在switch中使用范围匹配

对于数字类型的比较，switch中的case还可以指定一个范围，如果要比较的值在这个范围内，

则执行这个分支。示例代码如下：

```
let testscore = 80

var grade:Character

switch testscore {
case 90...100:                ①
    grade = "优"
case 80..<90:                 ②
    grade = "良"
case 60..<80:                 ③
    grade = "中"
case 0..<60:                  ④
    grade = "差"
default:                      ⑤
    grade = "无"
}

println("Grade = \(grade)")
```

输出结果如下：

```
说明：良
```

上述代码通过判断成绩范围，给出“优”、“良”、“中”和“差”评分标准，默认值“无”是分数不在上述范围内时给出的。

提示 在定义范围的时候，我们使用了闭区间（...）和半闭区间（..<）运算符。代码第①行90...100范围使用闭区间表示，即90 ≤ grade ≤100的情况，包含两个临界值。而第②、③、④行代码的范围使用了半闭区间，因此80..<90表示 80 ≤ grade < 90，包含下临界值，不包含上临界值。

7.1.4 在switch中比较元组类型

元组作为多个值的表示方式也可以在switch中进行比较。switch使用元组非常灵活，字段可以是普通值，也可以是范围。

示例代码如下：

```
var student = (id:"1002", name:"李四", age:32,
ChineseScore:80, EnglishScore:89)  ①

var desc:String

switch student {
```

```
case (, , _, 90...100, 90...100):  ②
    desc = "优"
case (, , _, 80..<90, 80.. <90):  ③
    desc = "良"
case (, , _, 60..<80, 60..<80):  ④
    desc = "中"
case (, , _, 60..<80, 90...100), (, , _,
90...100, 60..<80):  ⑤
    desc = "偏科"
case (, , _, 0.. <80, 90...100), (, , _,
90...100, 0..<80):  ⑥
    desc = "严重偏科"
default:
    desc = "无"
}

println("说明：\(desc)")
```

输出结果如下：

```
说明：良
```

上述代码第①行声明并初始化学生元组，它有5个字段，其中id为学号，name为姓名，age为年龄，ChineseScore为语文成

绩，EnglishScore为英语成绩。在这个示例中，我们只是比较学生的语文成绩和英语成绩，其他的字段不比较，我们可以在case中使用下划线（_）忽略其中的字段值。第②行代码中(，，_，90...100，90...100)的前三个下划线（_）忽略了id、name和age3个字段，switch不比较它们的值，只比较ChineseScore成绩是否属于90...100范围，比较EnglishScore成绩是否属于90...100范围。代码第③行和第④行也是类似的。

代码第⑤行和第⑥行有些特殊，这里使用了逗号（，）分隔两个元组值，这表示“或”的关系，即(，，_，60..<80，90...100)或(，，_，90...100，60..<80)的情况。

在switch中使用元组还可以使用值绑定和where语句。

1. **值绑定**

使用元组的时候还可以在case分支中将匹配的值绑定到一个临时的常量或变量，这些常量或变量能够在该分支里使用，这被称为**值绑定**。

示例代码如下：

```
var student = (id:"1002", name:"李四", age:32,
ChineseScore:90, EnglishScore:91)

var desc:String

switch student {
case (, , let age, 90...100, 90...100):
①
    if (age > 30) {
②
        desc = "老优"
    } else {
        desc = "小优"
    }
case (, , _, 80..<90, 80..<90):
    desc = "良"
case (, , _, 60..<80, 60..<80):
    desc = "中"
case (, , _, 60..<80, 90...100), (, , _,
90...100, 60..<80):
    desc = "偏科"
case (, , _, 0..<80, 90...100), (, , _,
90...100, 0..<80):
    desc = "严重偏科"
default:
    desc = "无"
}

println("说明：\(desc)")
```

输出结果如下：

```
说明：老优
```

本示例还是关于成绩的问题，其中第①行代码中的`let age`就是值绑定，我们在`case`中声明了一个`age`常量，然后`age`常量就可以在该分支中使用了，我们在第②行代码使用`arge`常量，判断`age > 30`。

2. where语句

在绑定值的情况下，还可以在`case`中使用`where`语句，进行条件的过滤，类似于SQL语句[1]中的`where`子句。

> [1]结构化查询语言（Structured Query Language，SQL），是一种数据库查询和程序设计语言，用于存取数据以及查询、更新和管理关系数据库系统；同时也是数据库脚本文件的扩展名。——百度百科
> http://baike.baidu.com/view/595350.htm

示例代码如下：

```
var student = (id:"1002", name:"李四", age:32,
```

```
ChineseScore:90, EnglishScore:91)

var desc:String

switch student {
case (, , let age, 90...100, 90...100) where
age > 0:  ①
    desc = "优"
case (, , _, 80..<90, 80..<90):
    desc = "良"
case (, , _, 60..<80, 60..<80):
    desc = "中"
case (, , _, 60..<80, 90...100), (, , _,
90...100, 60..<80):
    desc = "偏科"
case (, , _, 0..<80, 90...100), (, , _,
90...100, 0..<80):
    desc = "严重偏科"
default:
    desc = "无"
}

println("说明：\(desc)")
```

输出结果如下：

```
说明：优
```

本示例是对上个示例的修改，代码第①行中的`let age`就是值绑定，然后我们在`case`后面使用了`where age > 0`，过滤掉元组`age`字段小于0的数据。

7.2 循环语句

循环语句能够使程序代码重复执行。Swift编程语言支持4种循环构造类型：while、do while、for和for in。for和while循环是在执行循环体之前测试循环条件，而do while是在执行循环体之后测试循环条件。这就意味着for和while循环可能连一次循环体都未执行，而do while将至少执行一次循环体。for in是for循环的变形，它是专门为集合遍历而设计的。

7.2.1 while语句

while语句是一种先判断的循环结构，格式如下：

```
while 循环条件 {
    语句组
}
```

while循环没有初始化语句，循环次数是不可知的，只要循环条件满足，循环就会一直进行下去。

下面看一个简单的示例，代码如下：

```
var i:Int64 = 0

while i  i < 100000 {
    i++
}

println("i = \(i)")
println("i  i = \(i * i)")
```

输出结果如下：

```
i = 317
i * i = 100489
```

上述程序代码的目的是找到平方数小于100000的最大整数。使用while循环需要注意几点，while循环条件语句中只能写一个表达式，而且是一个布尔型表达式，那么如果循环体中需要循环变量，就必须在while语句之前对循环变量进行初始化。示例中先给i赋值为0，然后在循环体内部必须通过语句更改循环变量的值，否则将会发生死

循环。

提示　死循环对于单线程程序而言是一场灾难，但是在多线程程序中，死循环是必需的，死循环会出现在子线程中。例如游戏设计中对玩家输入装置的轮询，或是动画程序的播放，都是需要死循环的。下面的代码是死循环的一般写法。

```
while true {
    语句组
}
```

提示　循环是比较耗费资源的操作，如何让开发人员测试和评估循环效率呢？Xcode 6提供的Playground工具可以帮助我们实现这个目的。在Playground界面中打开时间轴，具体过程参考2.1节。打开Playground界面后运行代码，如图7-1所示，程序运行过程中会在右边时间轴绘制出一条线段，横轴是经历的时间，纵轴是i值变化，我们在执行完成后拖曳线段，查看运行历史中i值内容。就本例而言，

时间轴中的线段越陡，执行的效率越高。

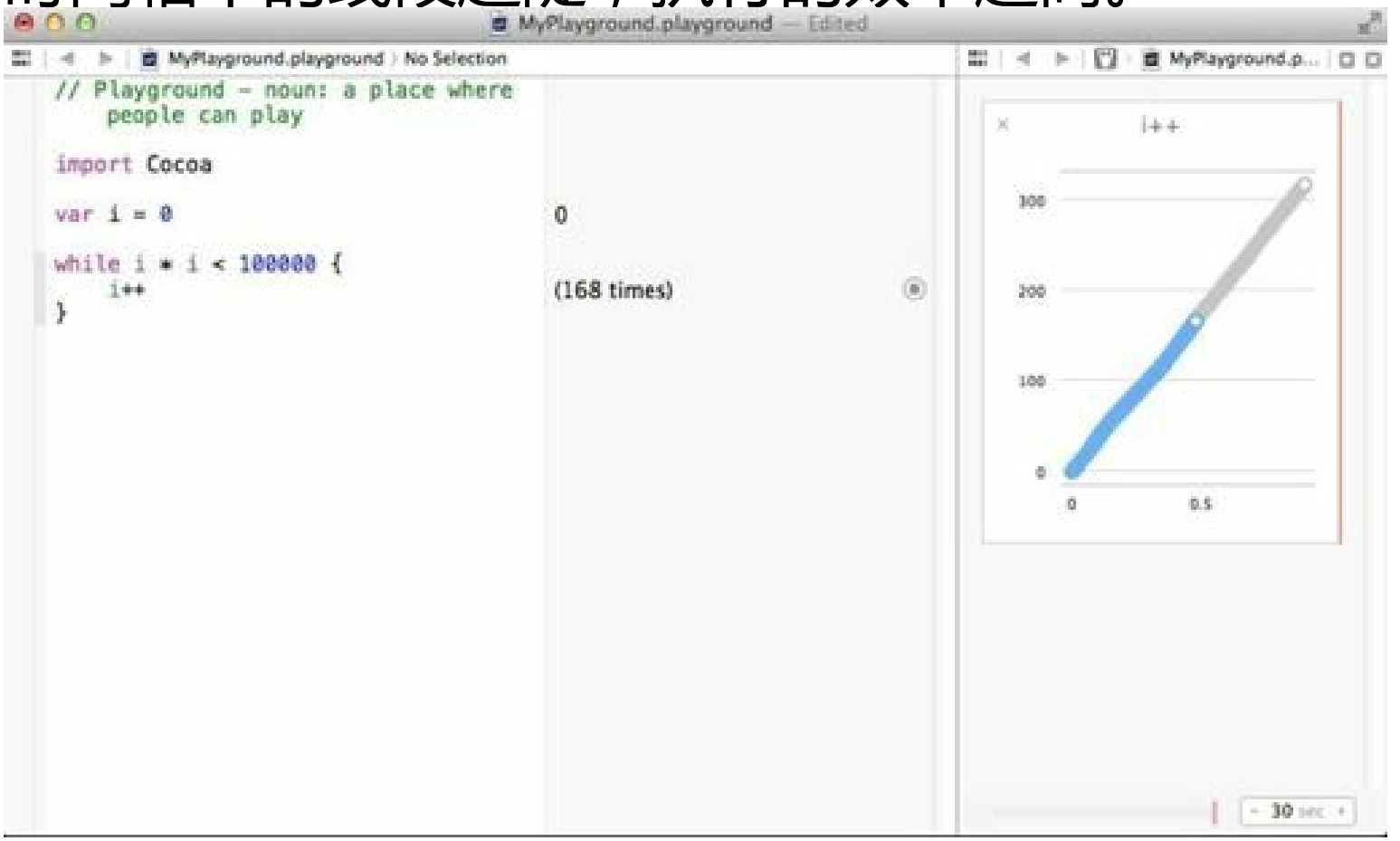


图 7-1　Playground界面

7.2.2　do while语句

do while语句的使用与while语句相似，不过do while语句是事后判断循环条件结构，语句格式如下：

```
do {
    语句组
} while 循环条件
```

do while循环没有初始化语句，循环次数是不可知的，不管循环条件是否满足，都会先执行一次循环体，然后再判断循环条件。如果条件满足则执行循环体，不满足则停止循环。

下面看一个示例代码：

```
var i:Int64 = 0

do{
    i++
} while  i  i < 100000

println("i = \(i)")
println("i  i = \(i * i)")
```

输出结果如下：

```
i = 317
i * i = 100489
```

该示例与上一节的示例是一样的，都是找到平方数小于100 000的最大整数。输出结果也是一样的。

7.2.3 for语句

for语句是应用最广泛、功能最强的一种循环语句。一般格式如下：

```
for 初始化; 循环条件; 迭代 {
    语句组
}
```

当程序执行到for语句时，会先执行初始化语句，它的作用是初始化循环变量和其他变量，然后程序会查看循环条件是否满足，如果满足，则继续执行循环体并计算迭代语句，之后再判断循环条件，如此反复，直到判断循环条件不满足时跳出循环。终止语句一般用来改变循环条件，它可对循环变量和其他变量进行操作。

以下示例代码是计算1~9的平方表程序：

```
println("n   n*n")
println("----------")
for var i = 1; i < 10; i++  {
    println("\(i) x \(i) = \(i * i)")
}
```

输出结果如下：

```
n   n*n
---------
1 x 1 = 1
2 x 2 = 4
3 x 3 = 9
4 x 4 = 16
5 x 5 = 25
6 x 6 = 36
7 x 7 = 49
8 x 8 = 64
9 x 9 = 81
```

在这个程序的循环部分初始时，给循环变量i赋值为1，每次循环都要判断i的值是否小于10，如果为true，则执行循环体，然后给i加1。因此，最后的结果是打印出1~9的平方，不包括9。

初始化、循环条件以及迭代部分都可以为空语句（但分号不能省略），三者均为空的时候，相当于一个无限循环。

```
for ; ;  {
    ......
}
```

在初始化部分和迭代部分，可以使用逗号语句来进行多个操作。逗号语句是用逗号分隔的语句序列，如下程序代码所示：

```
var x:Int32
var y:Int32

for x = 0, y = 10; x < y; x++, y--  {
    println("(x,y) = (\(x),\(y))")
}
```

输出结果如下：

```
(x,y) = (0,10)
(x,y) = (1,9)
(x,y) = (2,8)
(x,y) = (3,7)
(x,y) = (4,6)
```

7.2.4 for in语句

Swift提供了一种专门用于遍历集合的for循环

——for in循环。使用for in循环不必按照for的标准套路编写代码，只需要提供一个集合就可以遍历。

假设有一个数组，原来遍历数组的方式如下：

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

println("----for--------")
for var i = 0; i < countElements(numbers); i++
{
    println("Count is: \(i)")
}
```

输出结果如下：

```
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
```

上述语句`let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`声明并初始化了10个元素数组集合，目前大家只需要知道当初始化数组时，要把相同类型的元素放到[...]中并且用逗号分隔（,）即可，关于数组集合我们会在第10章详细介绍。

采用`for in`循环语句遍历数组的方式如下：

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

println("----for in----")
for item in numbers {
    println("Count is: \(item)")
}
```

输出结果如下：

```
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
```

```
Count is: 8
Count is: 9
```

从上例可以发现，item是循环变量，item之前使用var声明，它是隐式变量声明的。in后面是集合实例，for in循环语句会将后面集合中的元素一一取出来，保存到item中。可见for in语句在遍历集合的时候要简单方便得多。但是对于其他操作，for in循环就不太适合了。

7.3 跳转语句

跳转语句能够改变程序的执行顺序，可以实现程序的跳转。Swift有4种跳转语句：break、continue、fallthrough和return。其中return语句与函数返回有关，我们将在第9章介绍return，本章重点介绍break、continue和fallthrough语句的使用。

7.3.1 break语句

break语句可用于上一节介绍的while、do while、for和for in循环结构，它的作用是强行退出循环结构，不执行循环结构中剩余的语句。

提示 break语句也可用于switch分支语句，但switch默认在每一个分支之后隐式地添加了break，我们一定要显式地添加break才可以使程序运行不受影响。

在循环体中使用break语句有两种方式：可以带有标签，也可以不带标签。语法格式如下：

```
break           //不带标签
break label     //带标签，label是标签名
```

定义标签的时候后面跟一个冒号。不带标签的break语句使程序跳出所在层的循环体，而带标签的break语句使程序跳出标签指示的循环体。

下面看一个示例，代码如下：

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for var i = 0; i < countElements(numbers); i++
{
    if i == 3 {
        break
    }
    println("Count is: \(i)")
}
```

在上述程序代码中，当条件i==3的时候执行break语句，break语句会终止循环，程序运行的结果如下：

```
Count is: 0
```

```
Count is: 1
Count is: 2
```

break还可以配合标签使用，示例代码如下：

```
label1: for var x = 0; x < 5; x++ {
①

    label2: for var y = 5; y > 0; y-- {
②
        if y == x {
            break label1
③
        }
        println("(x,y) = (\(x),\(y))")
    }
}

println("Game Over!")
```

默认情况下，break只会跳出最近的内循环（代码第②行for循环）。如果要跳出代码第①行的外循环，可以为外循环添加一个标签label1:，然后在第③行的break语句后面指定这个标

签label1，这样当条件满足执行break语句的时候，程序就会跳转出外循环了。

程序运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
(x,y) = (1,2)
Game Over!
```

如果break后面没有指定外循环标签，则运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
```

```
(x,y) = (1,2)
(x,y) = (2,5)
(x,y) = (2,4)
(x,y) = (2,3)
(x,y) = (3,5)
(x,y) = (3,4)
(x,y) = (4,5)
Game Over!
```

比较两种运行结果，就会发现给break添加标签的意义，添加标签对于多层嵌套循环是很有必要的，适当使用可以提高程序的执行效率。

7.3.2 continue语句

continue语句用来结束本次循环，跳过循环体中尚未执行的语句，接着进行终止条件的判断，以决定是否继续循环。对于for语句，在进行终止条件的判断前，还要先执行迭代语句。

在循环体中使用continue语句有两种方式可以带有标签，也可以不带标签。语法格式如下：

```
continue           //不带标签
continue label     //带标签，label是标签名
```

下面看一个示例，代码如下：

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for var i = 0; i < countElements(numbers); i++
{
    if i == 3 {
        continue
    }
    println("Count is: \(i)")
}
```

在上述程序代码中，当条件i==3的时候执行continue语句，continue语句会终止本次循环，循环体中continue之后的语句将不再执行，接着进行下次循环，所以输出结果中没有3。程序运行结果如下：

```
Count is: 0
Count is: 1
Count is: 2
Count is: 4
Count is: 5
Count is: 6
Count is: 7
```

```
Count is: 8
Count is: 9
```

带标签的continue语句示例代码如下：

```
label1: for var x = 0; x < 5; x++ {
①

    label2: for var y = 5; y > 0; y-- {
②
        if y == x {
            continue label1
③
        }
        println("(x,y) = (\(x),\(y))")
    }
}

println("Game Over!")
```

默认情况下，continue只会跳出最近的内循环（代码第②行for循环），如果要跳出代码第①行的外循环，可以为外循环添加一个标签label1:，然后在第③行的continue语句后面

指定这个标签label1，这样当条件满足执行continue语句时候，程序就会跳转出外循环了。

程序运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
(x,y) = (1,2)
(x,y) = (2,5)
(x,y) = (2,4)
(x,y) = (2,3)
(x,y) = (3,5)
(x,y) = (3,4)
(x,y) = (4,5)
Game Over!
```

由于跳过了x==y，因此下面的内容没有输出。

```
(x,y) = (1,1)
(x,y) = (2,2)
```

```
(x,y) = (3,3)
(x,y) = (4,4)
```

7.3.3 fallthrough语句

fallthrough是贯通语句，只能使用在switch语句中。为了防止错误的发生，Swift中的switch语句case分支默认不能贯通，即执行完一个case分支就跳出switch语句。但是凡事都有例外，如果你的算法真的需要多个case分支贯通，也可以使用fallthrough语句。

下面是一个没有贯通的示例代码：

```
var j = 1
var x = 4

switch x {
case 1:
    j++
case 2:
    j++
case 3:
    j++
case 4:
    j++
```

```
case 5:
    j++
default:
    j++
}

println("j = \(j)")
```

运行结果如下：

```
j = 2
```

程序流程如图7-2所示，x = 4进入case 4，然后j++，跳出switch语句，所以最后输出j的值为2。

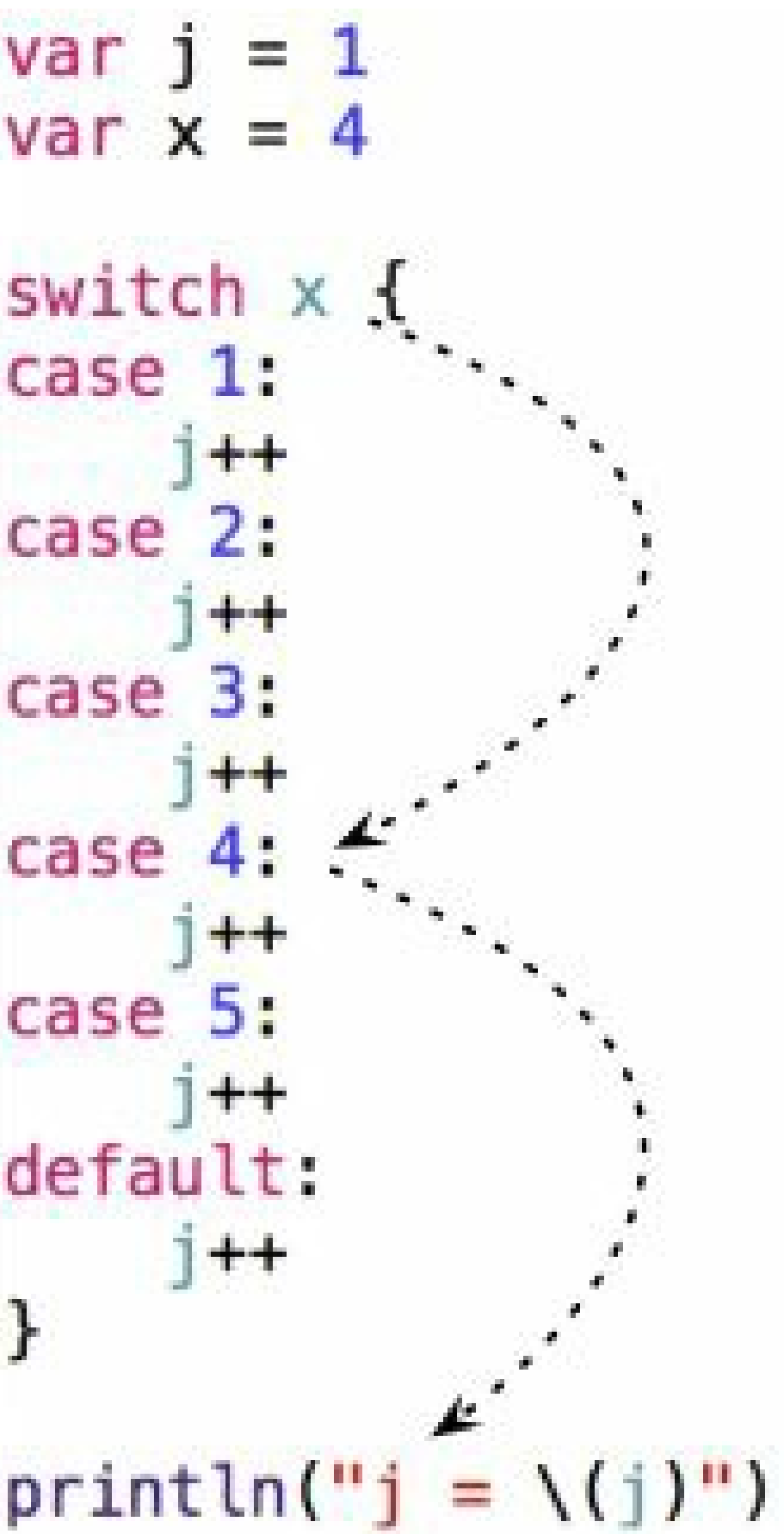


图 7-2　没有贯通的switch语句

我们来修改这个示例代码如下：

```
var j = 1
var x = 4

switch x {
case 1:
    j++
case 2:
    j++
case 3:
    j++
case 4:
    j++
    fallthrough
case 5:
    j++
    fallthrough
default:
    j++
}

println("j = \(j)")
```

运行结果如下：

```
j = 4
```

程序流程如图7-3所示，x = 4进入case 4分支，然后j++。由于fallthrough，程序会进入case 5分支，然后j++。由于还有fallthrough，程序会进入default分支，走完该分支后跳出switch语句，所以最后输出j的值为4。

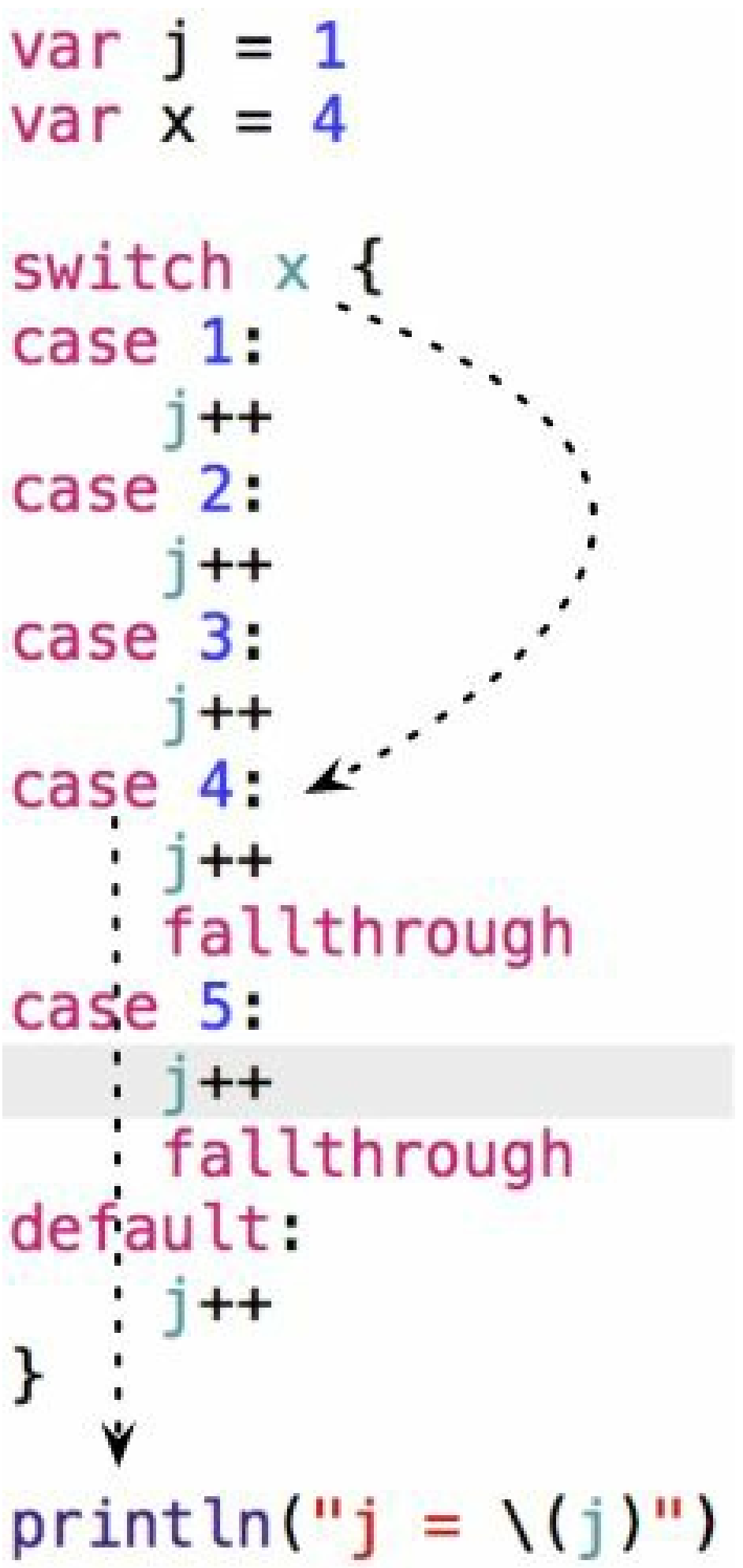

var j = 1
var x = 4
switch x {
case 1:
j++
case 2:
j++
case 3:
j++
case 4:
j++
fallthrough
case 5:
j++
fallthrough
default:
j++
}
println("j = \(j)")

图 7-3　有贯通的switch语句

从以上两个例子可见，fallthrough就是为了贯穿case分支而设的。或许这种语句我们用得很少，但作为一门编程语言，还是要照顾用户的少数特殊需求。

7.4 本章小结

通过对本章内容的学习，我们可以了解到Swift语言的控制语句，其中包括分支语句（if和switch），循环语句（while、do while、for和for in）和跳转语句（break、continue、fallthrough和return）等。

7.5 同步练习

1. 编程题：水仙花数是一个三位数，三位数各位的立方之和等于三位数本身。使用while循环计算水仙花数

2. 编程题：水仙花数是一个三位数，三位数各位的立方之和等于三位数本身。使用do while循环计算水仙花数。

3. 编程题：水仙花数是一个三位数，三位数各位的立方之和等于三位数本身。使用for循环计算水仙花数。

4. 编程题：编写程序输出以下形式的金字塔图案。

```
   **
  *****
*******
```

5. 能从循环语句的循环体中跳出的语句是（　　）。

A. for语句　　B. break语句　　C. while语句　　D. continue语句

6. 若有如下循环语句，则循环体将被执行（　　）。

```
var x=5, y=20
do{
    y-=x
    x++
} while(++x < --y)
```

A. 0次　　B. 1次　　C. 2次　　D. 3次

7. 下列语句序列执行后，i的值是（　　）。

```
var i=16
do {
    i/=2
} while( i > 3 )
```

A. 16　　B. 8　　C. 4　　D. 2

8. 若有以下代码段：

```
let m = ?
switch m
{
case 0:
```

```
    println("case 0")
case 1:
    println("case 1")
case 2:
    println("case 2")
    fallthrough
default:
    println("default")
}
```

则下列m的哪些值将引起"default"的输出？

A. 0　　B. 1　　C. 2　　D. 3

9. 下列语句执行后，x的值是（　　）。

```
var a=3, b=4, x=5
if  ++a < b {
    x=x+1
}
```

A. 5　　B. 3　　C. 4　　D. 6

10. 下列语句序列执行后，k的值是（　　）。

```
var  i=6, j=8, k=10, n=5, m=7
if  i<j || m<n {
```

```
    k++
} else {
    k--
}
```

A. 9　　B. 10　　C. 11　　D. 12

11. 下列语句序列执行后，r的值是（　　）。

```
var ch  = "8"
var r = 10
switch ch {
case "7":
    r = r+3
case "8":
    r = r+5
case "9":
    r = r+6
    break;
default:
    r = r+7
}
```

A. 13　　B. 15　　C. 16　　D. 10

12. 下列语句序列执行后，j的值是（　　）。

```
var j=0,i=3
for ; i>0; i-- {
    j+=i
}
```

A. 5　　B. 6　　C. 7　　D. 8

13. 下列语句序列执行后，i的值是（　　）。

```
var i=10
do {
    i-=2
} while(i>6)
```

A. 10　　B. 8　　C. 6　　D. 4

14. 能构成多分支的语句是（　　）。

A. for语句　　B. while语句　　C. switch语句　　D. do while语句

15. 以下for循环的执行次数是（　　）。

```
for(var x=0,y=0; (y == 0) && (x < 4); x++) {
}
```

A. 无限次　　B. 一次也不执行　　C. 执行4次　　D. 执行3次

16. 以下由do while语句构成的循环执行的次数是（　　）。

```
var k = 0
do {
    ++k
} while ( k < 1 )
```

A. 一次也不执行　　B. 执行1次

C. 无限次　　D. 有语法错，不能执行

17. 下列语句序列执行后，x的值是（　　）。

```
var a=3, b=4, x=5
if ++a == b {
    x = ++a * x
}
```

A. 35　　B. 25　　C. 20　　D. 5

18. 下列语句序列执行后，k的值是（　　）。

```
var  i=6,j=8,k=10,m=7
if  i > j || m < k-- {
    k++
} else {
    k--
}
```

A. 12　　B. 11　　C. 10　　D. 9

19. 下列语句序列执行后，k的值是（　　）。

```
var j=8, k=15
for var i=2; i != j; i++ {
    j -= 2
    k++
}
```

A. 18　　B. 15　　C. 16　　D. 17

20. 下列语句序列执行后，j的值是（　　）。

```
var j=3, i=2
while --i != i/j  {
    j=j+2
}
```

A. 2　　B. 4　　C. 6　　D. 5

21. 下列代码执行的结果是（　　）。

```
var x = 1, y = 6
while y-- == 6 {
    x--
}
println("x= \(x) ,y = \(y)")
```

A. 程序能运行，输出结果：x=0,y=5　　B. 程序能运行，输出结果：x=-1,y=4

C. 程序能运行，输出结果：x=0,y=4　　D. 程序不能编译

22. 下列语句序列执行后，k的值是（　　）。

```
var x=6, y=10, k=5
switch  x % y {
    case 0:
        k = x*y
    case 6:
        k = x/y
        fallthrough
    case 12:
        k = x-y
        fallthrough
```

```
    default:
        k = x*y-x
}
```

A. 60　　B. 5　　C. 0　　D. 54

23. 以下由for语句构成的循环执行的次数是（　　）。

```
for var i = 0; true; i++ {
}
```

A. 有语法错，不能执行　　B. 无限次

C. 执行1次　　D. 一次也不执行

24. 简答题：请例举在switch中使用范围匹配。

25. 简答题：请例举在switch中使用元组类型。

第 8 章　集合

记得大学计算机老师曾告诉我们“程序=数据结构+算法”，记得学过很多数据结构的算法，例如数组（array）、栈（stack）、队列（queue）、链表（linked list）、树（tree）、图（graph）、堆（heap）和哈希表（hash）等结构。这些数据结构的本质是一个集合，可以按照它们的算法对集合中的数据进行添加、删除、排序等集合运算。不同的结构对应于不同的算法，有的考虑节省占用空间，有的考虑提高运行速度，对于程序员而言，它们就像是“熊掌”和“鱼肉”，不可兼得！提高运行速度往往是以牺牲空间为代价的，而节省占用空间往往是以牺牲运行速度为代价的。

Swift中提供了两种数据结构的实现：数组和字典。字典也叫映射或哈希表。这两种结构很有代表性，所以Swift主要提供了这两种结构的实现。

8.1　数组集合

数组是一串有序的由相同类型元素构成的集合。数组中的集合元素是有序的，可以重复出现。图8-1是一个班级集合数组，在这个集合中有一些学生，这些学生是有序的，顺序是它们放到集合中的顺序，我们可以通过下标序号访问它们。这就像老师给进入班级的人分配学号，第一个报到的是“张三”，老师给他分配的是0，第二个报到的是“李四”，老师给他分配的是1，以此类推，最后一个序号应该是“学生人数-1”。

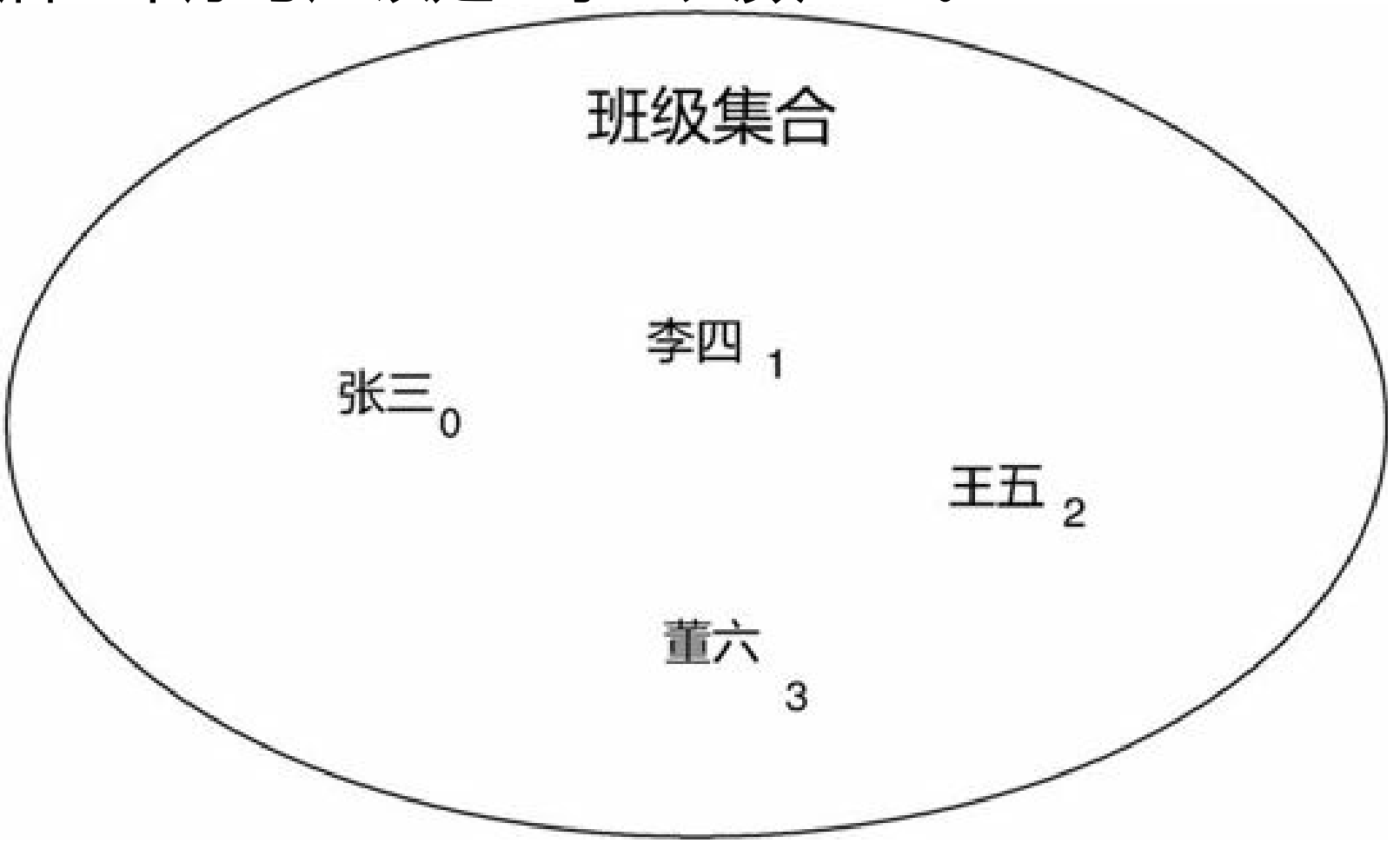


图 8-1　数组集合

提示　数组更关心元素是否有序，而对于是否重复则不关心。请大家记住这个原则。

8.1.1　数组声明与初始化

Swift为数组提供Array结构体类型，在声明一个数组的时候可以使用下面的语句之一。

```
var studentList1: Array<String>

var studentList2: [String]
```

其中变量studentList1明确指定类型为Array<String>，<String>是泛型，说明在这个数组中只能存放字符串类型的数据。studentList2变量也是声明一个只能存放字符串类型的数组。[String]与Array<String>是等价的，[String]是简化写法。

上面声明的集合事实上还不能用，还需要进行初始化，集合类型往往在声明的同时进行初始化。示例代码如下：

```
var studentList1: Array<String> = ["张三","李
```

```
四","王五","董六"]                    ①

var studentList2: [String]  = ["张三","李四","王
五","董六"]                           ②

let studentList3: [String]  = ["张三","李四","王
五","董六"]                           ③

var studentList4 = [String]()
④
```

上述代码都是对数组进行声明和初始化，①~③采用["张三","李四","王五","董六"]的方式进行初始化，这是数组的表示方式，它的语法如图8-2所示。

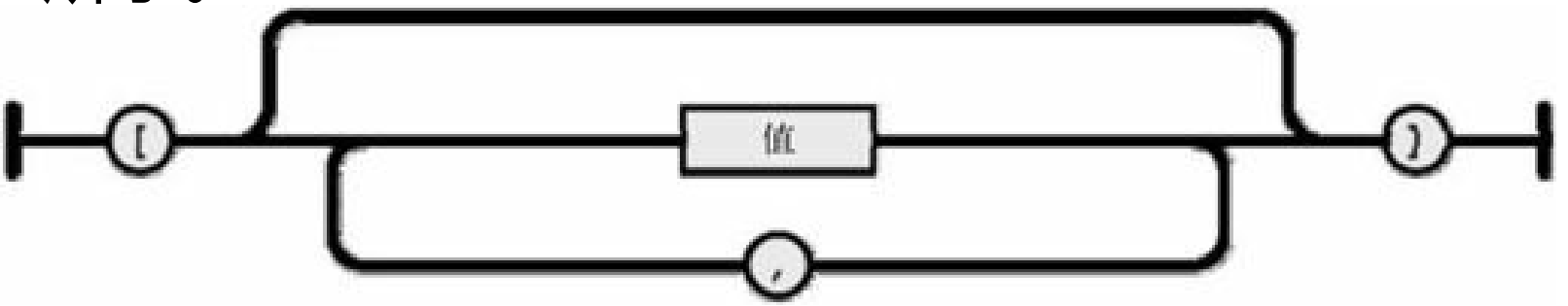


图 8-2　数组表示语法

这个语法类似于JSON[1]中的数组，数组以“[”（左中括号）开始，“]”（右中括号）结束，值之间使用“,”（逗号）进行分隔。

[1]JSON（JavaScript Object Notation）是一种轻量级的数据交换格式。它基于JavaScript（Standard ECMA-262 3rd

> Edition-December 1999）的一个子集。 JSON采用完全独立于语言的文本格式，但是也使用了类似于C语言家族的习惯（包括C、 C++、C#、Java、JavaScript、Perl、Python等）。这些特性使JSON成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成。——百度百科
> http://baike.baidu.com/view/136475.htm

第③行是`let`声明数组，`let`声明的数组是不可变数组，必须在声明的同时进行初始化，一旦被初始化，就不可以被修改。不可变数组在访问效率上比可变数组（`var`声明的数组）要高，可变数组通过牺牲访问效率换取可变。例如，当往可变数组内添加新元素的时候，数组要重新改变它的大小，然后重排它们的索引下标，这些都会影响性能。因此如果你真的确定数组是不需要修改的，那么你应该将它声明为`let`。

代码第④行是初始化一个空的数组，它与`var studentList2: [String]`是有区别的。`var studentList2: [String]`语句声明没有初始化，即没有开辟内存空间，而第④行的`[String]()`进行了初始化，只不过没有任何元素。

8.1.2 数组的修改

我们可以对数组中的元素进行追加、插入、删

除和替换等修改操作。追加元素可以使用数组append方法或+操作符实现，插入元素可以使用数组的insert方法实现，删除元素可以使用数组的removeAtIndex方法实现。

下面我们来看一个示例：

```
var studentList: [String]  = ["张三","李四","王五"]                    ①
println(studentList)
②

studentList.append("董六")
③
println(studentList)
④

studentList += ["刘备", "关羽"]
⑤
println(studentList)
⑥

studentList.insert("张飞",atIndex:studentList.count)                    ⑦
println(studentList)
⑧

let removeStudent =
```

```
studentList.removeAtIndex(0)
⑨
println(studentList)
⑩

studentList[0] = "诸葛亮"
⑪
println(studentList)
⑫
```

输出结果如下：

```
[张三，李四，王五]
[张三，李四，王五，董六]
[张三，李四，王五，董六，刘备，关羽]
[张三，李四，王五，董六，刘备，关羽，张飞]
[李四，王五，董六，刘备，关羽，张飞]
[诸葛亮，王五，董六，刘备，关羽，张飞]
```

上述代码第①行是声明并初始化数组studentList，第②行是打印数组，第③行追加了一个元素“董六”。如果要想追加多个元素，可以采用第⑤行+=操作符，它能够追加多个元素。

第⑦行是使用insert方法插入元

素，atIndex参数是插入的位置。本例中传递的是studentList.count，count是数组的属性，可以获得数组的长度。

第⑨行代码是使用removeAtIndex方法删除元素，参数是删除的位置，方法返回值是删除的元素。本例中传递的是0，表示删除第一个元素。removeAtIndex方法成功执行后，第一个元素会被删除，因此最后studentList输出的结果中没有“张三”，而这个时候removeStudent常量内容是“张三”。

第⑪行代码是替换第一个元素。

8.1.3 数组遍历

数组最常用的操作是**遍历集合**，就是将数组中的每一个元素取出来，进行操作或计算。整个遍历过程与循环分不开，我们可以使用for in循环进行遍历。

下面是遍历数组的示例代码：

```
var studentList: [String]  = ["张三","李四","王五"]

for item in studentList {
```

```
①
    println (item)
}

for (index, value) in enumerate(studentList) {
②
    println("Item \(index + 1 ) : \(value)")
}
```

运行结果如下：

```
张三
李四
王五
Item 1 : 张三
Item 2 : 李四
Item 3 : 王五
```

从上述代码可见，遍历数组有两种方法，它们都采用for in语句。第①行代码的遍历适合于不需要循环变量的情况，for in可以直接从数组studentList中逐一取出元素，然后进行打印输出。第②行代码的遍历适合于需要循环变量的情况，enumerate函数可以取出数组的索引和元

素，其中(index, value)是元组类型。

8.2　字典集合

字典表示一种非常复杂的集合，允许按照某个键来访问元素。字典是由两部分集合构成的，一个是键（key）集合，一个是值（value）集合。键集合是不能有重复元素的，而值集合是可以重复的，键和值是成对出现的。

图8-3所示是字典结构的“学号与学生”集合，学号是键集合，不能重复，学生是值集合，可以重复。

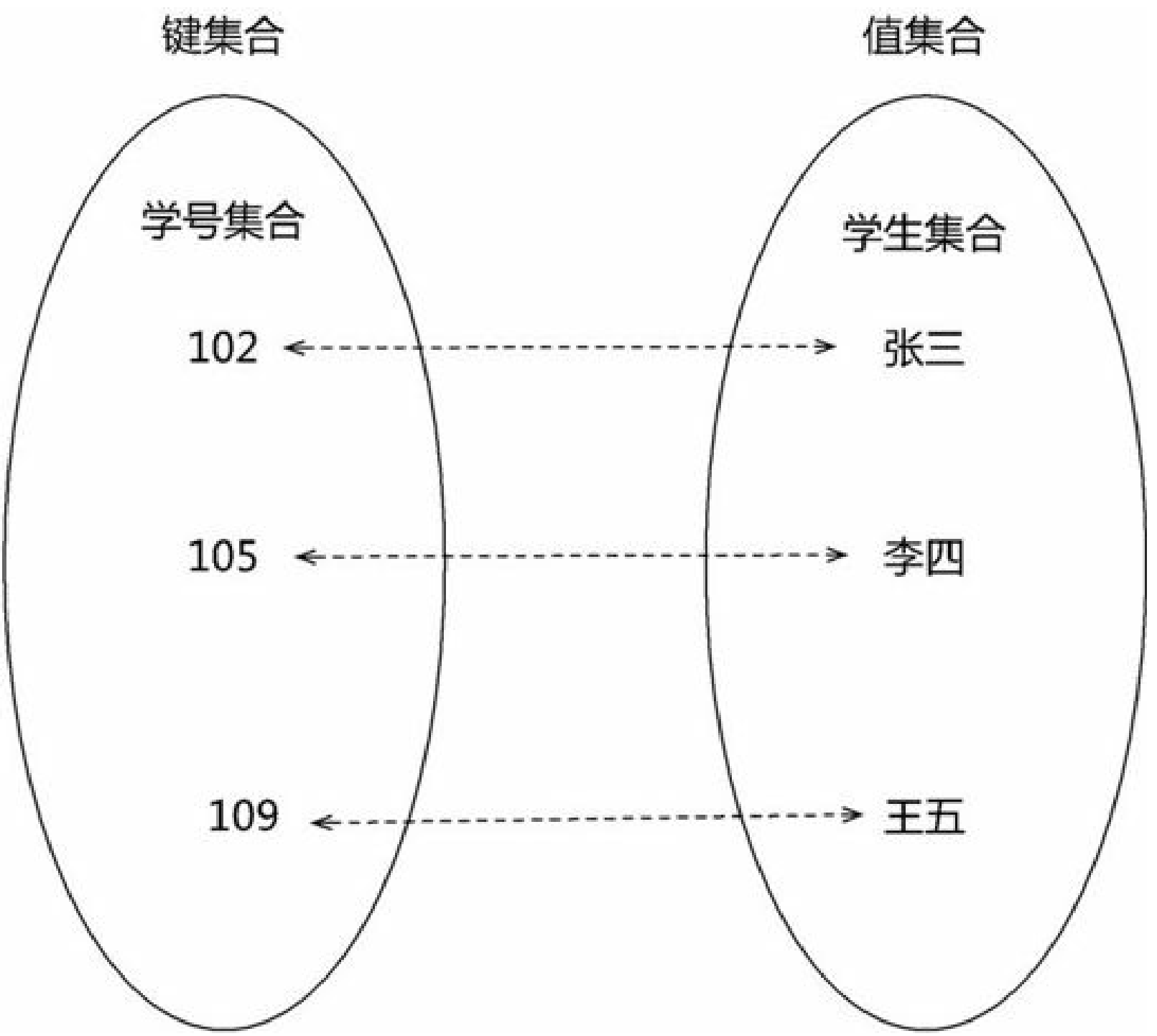


图 8-3　字典集合

提示　字典中键和值的集合是无序的，即便在添加的时候是按照顺序添加的，当取出这些键或值的时候，也会变得无序。字典集合更适合通过键快速访问值，就像查英文字典一

样，键就是要查的英文单词，而值是英文单词的翻译和解释等。有的时候，一个英文单词会对应多个翻译和解释，这也是与字典集合特性对应的。

8.2.1 字典声明与初始化

Swift为字典提供了Dictionary结构体类型，我们在声明一个字典的时候可以使用下面的语句。

```
var studentDictionary: Dictionary<Int, String>
```

其中，变量studentDictionary明确指定类型为Dictionary<Int, String>。其中<Int, String>是泛型，这表明键的集合是Int类型，值的集合是String类型。

上面声明的集合事实上还不能用，还需要进行初始化，集合类型往往是在声明的同时进行初始化的。示例代码如下：

```
var studentDictionary1: Dictionary<Int, String>
= [102 : "张三",105 : "李四", 109 : "王五"]    ①
```

```
var studentDictionary2 = [102 : "张三",105 : "李四", 109 : "王五"]
②

let studentDictionary3 = [102 : "张三",105 : "李四", 109 : "王五"]
③

var studentDictionary4 = Dictionary<Int, String>()
④
```

上述代码都是对字典进行声明和初始化，代码①~③行采用[102 : "张三",105 : "李四", 109 : "王五"]的方式进行初始化，这是字典的表示方式，语法如图8-4所示。

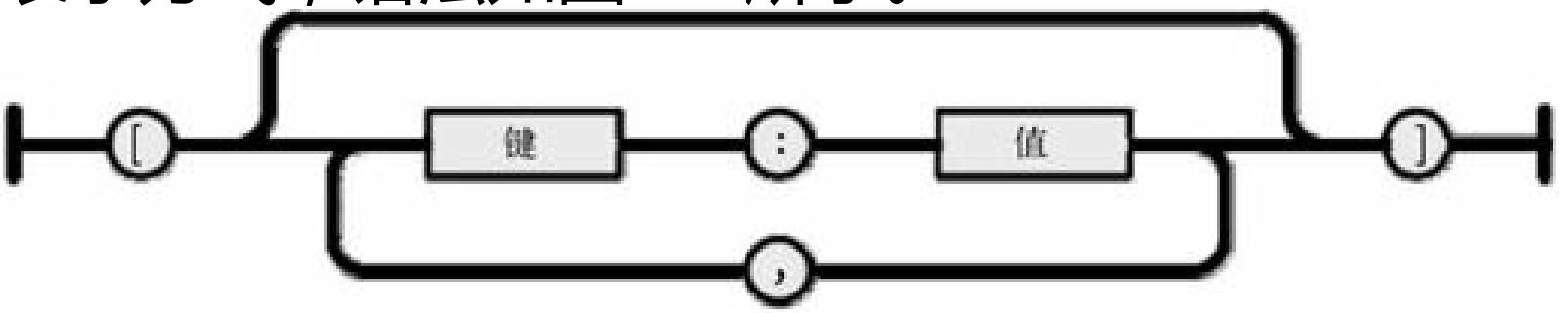


图 8-4　字典表示语法

这个语法类似于JSON中的对象，字典以“[”（左括号）开始，以“]”（右括号）结束。每个键后跟一个“:”（冒号），“键-值”对之间使用“,”（逗号）分隔。

第③行是let声明字典，let声明的字典是不可

变字典，必须在声明的同时初始化，一旦被初始化就不可以被修改。

代码第④行是初始化一个空的字典，键集合为Int类型，值集合为String，初始化后没有任何元素。

8.2.2 字典的修改

我们可以对字典中的元素进行追加、删除和替换等修改操作。字典元素的追加比较简单，只要给一个不存在的键赋一个有效值，就会追加一个“键-值”对元素。

字典元素删除有两种方法，一种是给一个键赋值为nil，就可以删除元素；另一种方法是通过字典的removeValueForKey方法删除元素，方法返回值是要删除的值。

字典元素替换也有两种方法，一种是直接给一个存在的键赋值，这样新值就会替换旧值；另一种方法是通过updateValue(forKey:)方法替换，方法的返回值是要替换的值。

下面我们来看一个示例：

```
var studentDictionary = [102 : "张三",105 : "李四", 109 : "王五"]          ①
```

```
studentDictionary[110] = "董六"
②

println("班级人数：\(studentDictionary.count)")
③

let dismissStudent =
studentDictionary.removeValueForKey(102)
④

println("开除的学生：\(dismissStudent)")
⑤

studentDictionary[105] = nil
⑥

studentDictionary[109] = "张三"
⑦

let replaceStudent =
studentDictionary.updateValue("李四",
forKey:110)    ⑧
println("被替换的学生是：\(replaceStudent)")
⑨
```

输出结果如下：

```
班级人数：4
开除的学生：张三
被替换的学生是：董六
```

上述代码第①行是声明并初始化字典studentDictionary，第②行代码追加键为110、值为“董六”的一个元素，第③行代码是打印班级学生的人数，count是字典的属性，返回字典的长度。

第④行和第⑥行都是删除元素，第④行代码是使用removeValueForKey方法删除元素，dismissStudent是返回值，它保持了被删除的元素。因此我们在第⑤行打印输出dismissStudent是“开除的学生：张三”。第⑥行studentDictionary[105] = nil语句是直接赋值nil也可以删除105对应的元素。

第⑦行和第⑧行都是替换旧元素，如果第⑦行的键不存在，那么结果是在字典中追加一个新的“键-值”对元素。第⑧行是通过updateValue(forKey:)方法替换元素，方法的返回值是“董六”，第⑨行代码是打印“被替换的学生是：董六”。

8.2.3 字典遍历

字典遍历集合也是字典的重要操作。与数组不同，字典有两个集合，因此遍历过程可以只遍历值的集合，也可以只遍历键的集合，也可以同时遍历。这些遍历过程都是通过for in循环实现的。

下面是遍历字典的示例代码：

```
var studentDictionary = [102 : "张三",105 : "李四", 109 : "王五"]

println("---遍历键---")
for studentID in studentDictionary.keys {
①
    println("学号：\(studentID)")
}

println("---遍历值---")
for studentName in studentDictionary.values {
②
    println("学生：\(studentName)")
}

println("---遍历键:值---")
for (studentID, studentName) in
studentDictionary {                          ③
    println ("\(studentID) : \(studentName)")
}
```

运行结果如下：

```
---遍历键---
学号：105
学号：102
学号：109
---遍历值---
学生：李四
学生：张三
学生：王五
---遍历键:值---
105 ： 李四
102 ： 张三
109 ： 王五
```

从上述代码可见，我们有3种方法遍历字典，它们都采用了for in语句。第①行代码遍历了键集合，其中keys是字典属性，可以返回所有键的集合。第②行代码遍历了值的集合，其中values是字典属性，可以返回所有值的集合。第③行代码遍历取出的字典元素，(studentID, studentName)是元组类型，它是由键变量

studentID和值变量studentName组成的。

8.3 集合的复制

集合在赋值或参数传递过程中会发生复制。Swift中的集合都是结构体类型，即值类型。值类型在赋值或参数传递时会发生复制行为，赋予的值或传递的参数是一个副本；而引用类型在赋值或参数传递时不发生复制行为，赋予的值或传递的参数是一个引用（实例本身）。

同样是值类型，集合要比其他值类型（整型、浮点型等）的复制行为更复杂，这是因为集合里面包含了一些数据。这些数据是否发生复制行为呢？这要看这些数据本身是值类型还是引用类型，如果是值类型则发生复制行为，如果是引用类型则不发生复制行为。

下面我们分别讨论字典和数组的复制。

8.3.1 字典复制

在为字典赋值或参数传递的时候，字典总是发生复制行为。而字典中的键和值等数据是否发生复制，要看本身键和值本身的类型。

我们先看一个示例：

```
var students = [102 : "张三",105 : "李四"]
①
var copyStudents = students
```

```
②

copyStudents[102] = "王五"
③
println(students[102])
④
```

上述代码第①行是创建并初始化字典变量students，然后在第②行将字典变量students赋值给copyStudents变量，这个过程中发生了复制行为，students中的内容被复制到copyStudents。由于字典中存放的键和值都是值类型，也发生了复制行为。所以当在第③行修改copyStudents[102]内容的时候，不会影响students[102]内容，在第④行打印的时候还是"张三"。图8-5是这个示例的复制示意图。

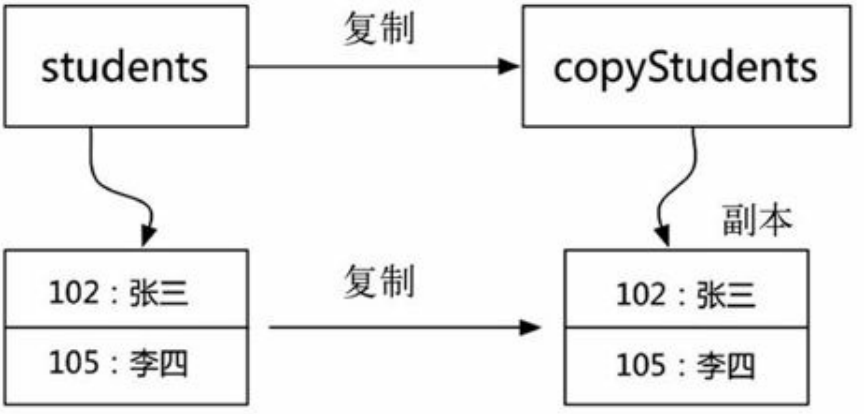


图 8-5　字典值类型复制示意图

下面我们再看一个示例：

```
class Employee {
①
    var name : String          // 姓名
    var salary : Double        // 工资
    init (n : String) {
        name = n
        salary = 0
    }
}

var emps = Dictionary<String, Employee>()
②
let emp1 = Employee(n: "Amy Lee")
③
let emp2 = Employee(n: "Harry Hacker")
④
emps["144-25-5464"] = emp1
⑤
emps["567-24-2546"] = emp2
⑥

//赋值，发生复制
var copyEmps = emps
⑦

let copyEmp : Employee! = copyEmps["567-24-
2546"]
copyEmp.name = "Gary Cooper"
⑧
```

```
let emp : Employee! = emps["567-24-2546"]
println(emp.name)
⑨
```

上述代码第①行是定义一个Employee类，关于类的定义我们会在11.4节详细介绍，在本例中不用关心Employee类的细节问题。

第②行代码声明字典变量，它的键要求是String类型，值要求是Employee类型。第③行和第④行是实例化Employee对象。第⑤行和第⑥行分别将两个对象赋值给字典。

第⑦行代码是将字典变量emps赋值给copyEmps变量，这个过程中发生了复制行为，但是由于emp1和emp2对象是引用类型，不发生复制，因此在第⑧行代码修改copyEmp内容之后，在第⑨行打印的时候输出"Gary Cooper"。

这个示例说明，emps字典中的emp2与copyEmps字典中的copyEmp具有相同的引用。在字典发生复制行为的过程中，引用类型的内容并没有发生复制。图8-6是这个示例的示意图。

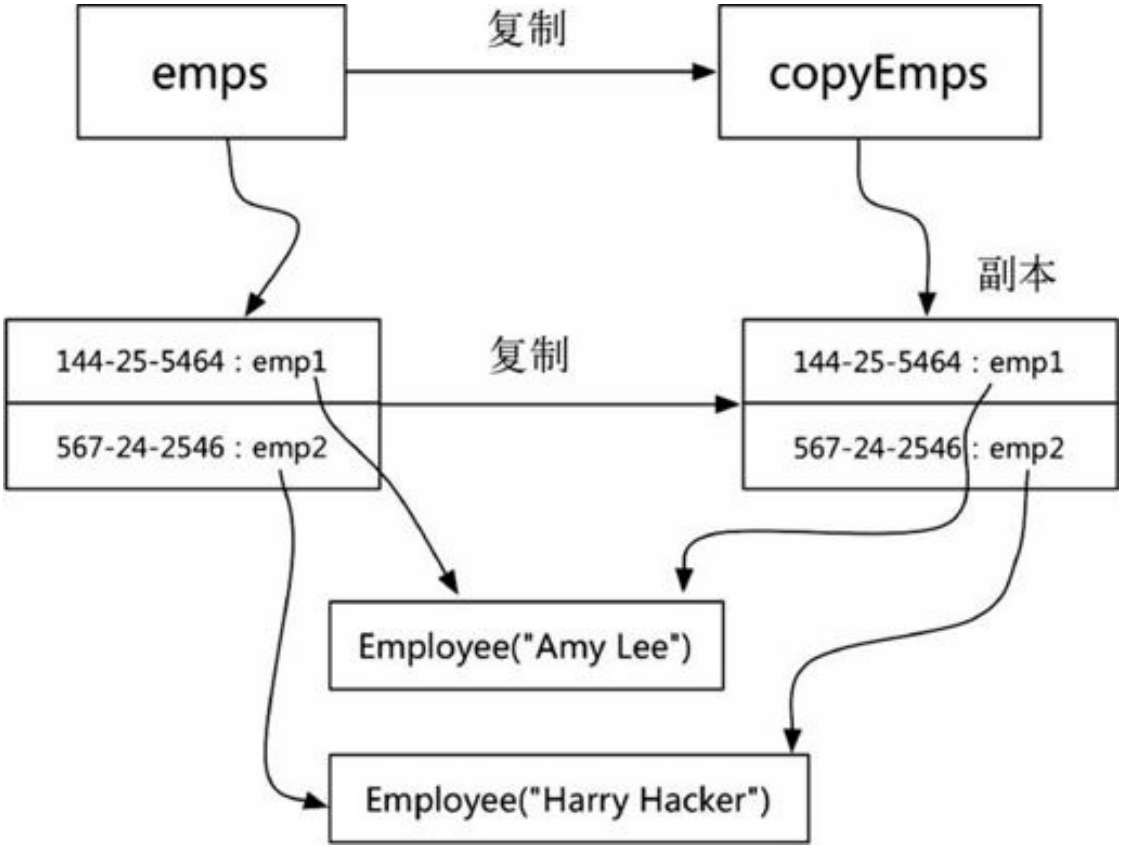


图 8-6　字典引用类型内容示意图

8.3.2　数组复制

Xcde 6 beta2中的数组复制的语法非常复杂，beta3之后变得简单了，与字典的复制类似。在为数组赋值或参数传递的时候，数组总是发生复制行为。而数组中的数据是否也发生复制行为，要看数据本身的类型。

下面我们再看一个示例：

```
var a = ["张三", "李四", "王五"]     ①
var b = a                              ②
var c = a                              ③
```

```
a[0] = "董六"                                    ④

println(a[0])                                    ⑤
println(b[0])                                    ⑥
println(c[0])                                    ⑦
```

上述代码第①行是声明并初始化数组a，在代码第②行和第③行将a赋值给b和c。这时它们发生了复制行为，然后当我们在第④行修改a的第一个元素后，第⑤行输出的是"董六"，而第⑥行和第⑦行输出的是"张三"。

图8-7是这个示例的复制示意图。

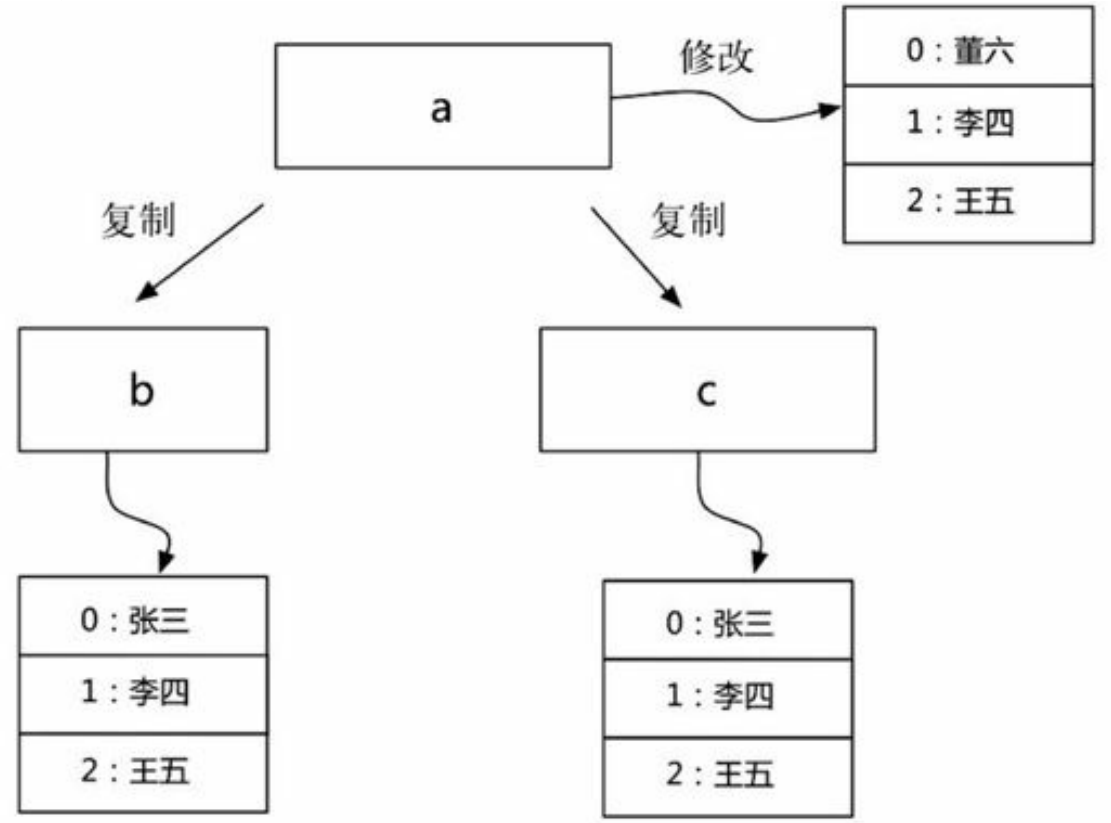


图 8-7　数组值类型复制示意图

下面我们再看一个示例：

```
class Employee {
    var name : String          // 姓名
    var salary : Double        // 工资
    init (n : String) {
        name = n
        salary = 0
    }
}

var emps = Array<Employee>()                          ①
let emp1 = Employee(n: "Amy Lee")                     ②
let emp2 = Employee(n: "Harry Hacker")                ③
emps.append(emp1)                                     ④
emps.append(emp2)                                     ⑤

//赋值，发生复制
var copyEmps = emps                                   ⑥

let copyEmp : Employee! = copyEmps[0]
copyEmp.name = "Gary Cooper"                          ⑦

let emp : Employee! = emps[0]
println(emp.name)                                     ⑧
```

第①行代码声明数组变量，它要求内容数据

是Employee类型。第②行和第③行是实例化Employee对象。第④行和第⑤分别将两个对象添加到数组中。

第⑥行代码是将数组变量emps赋值给copyEmps变量，这个过程中发生了复制行为，但是由于emp1和emp2对象是引用类型，不发生复制，因此在第⑦行代码修改copyEmp内容之后，在第⑧行打印的时候输出"Gary Cooper"。

这个示例说明，emps数组中的emp2与copyEmps数组中的copyEmp具有相同的引用。在数组发生复制行为的过程中，引用类型的内容并没有发生复制。图8-8是这个示例的示意图。

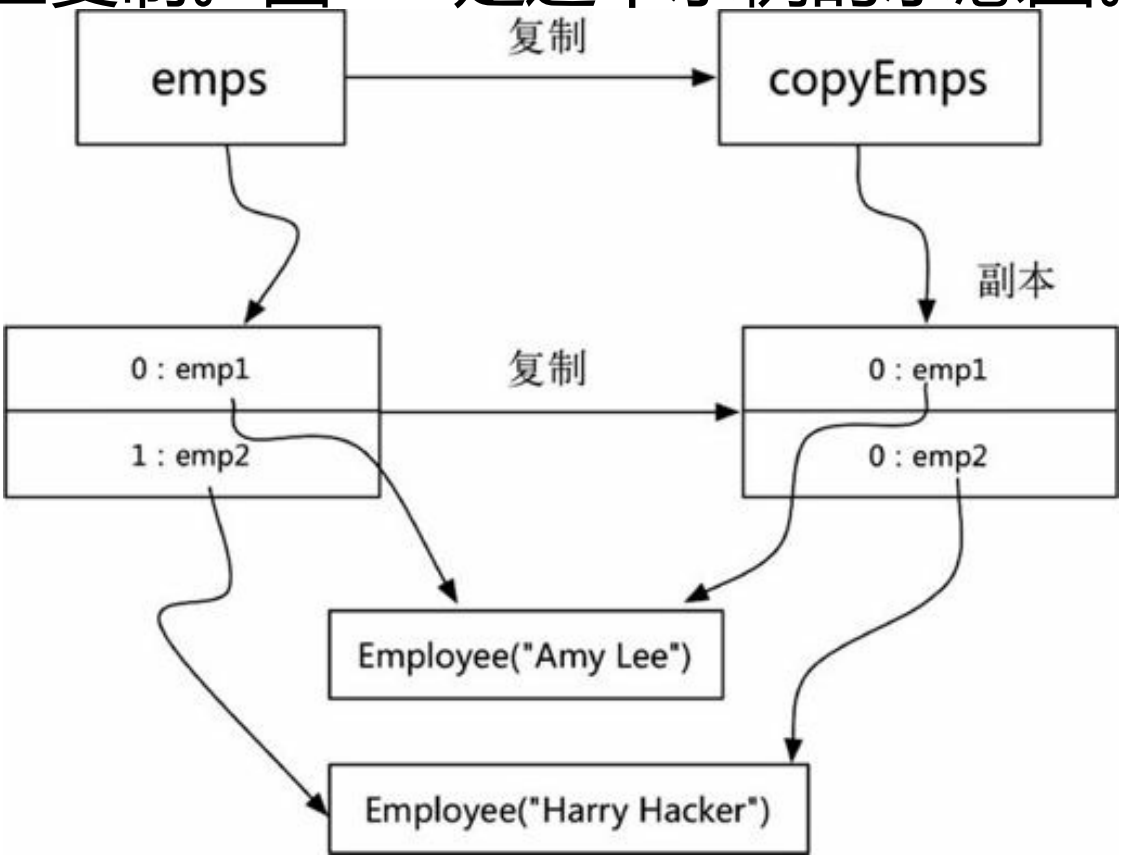


图 8-8　数组引用类型内容示意图

8.4 本章小结

通过对本章内容的学习，我们可以了解到Swift语言的集合，其中包括了数组集合和字典集合。

8.5 同步练习

1. 以下定义数组的语句中，不正确的是（　　）。

A. `let a : Array<Int> = [1,2]`

B. `let a : [Int] = [1,2]`

C. `var b: [String] = ["张三","李四"]`

D. `int Array [] a1,a2`

E. `int a3[]={1,2,3,4,5}`

2. 在一个应用程序中有如下定义：`let a = [1,2,3,4,5,6,7,8,9,10]`。为了打印输出数组a的最后一个元素，下列代码正确的是（　　）。

A. `println(a[10])`　　B. `println(a[9])`

C. `println(a[a.length])`　　D. `println(a(8))`

3. 下列语句序列执行后，打印输出结果是（　　）。

```
var ages = ["张三": 23, "李四": 35, "王五": 65, "董六": 19]
var copiedAges = ages
copiedAges["张三"] = 24
```

```
println(ages["张三"])
```

A. 65　　B. 35　　C. 24　　D. 23

4. 下列语句序列执行后，打印输出结果是（　　）。

```
var n1 = [900, 200, 300]
var n2 = n1
var n3 = n1

n1[0] = 1000
println(n1[0])
println(n2[0])
println(n3[0])
```

A.	B.	C.	D.
900	800	1000	1000
900	900	900	800
900	900	900	900

5. 判断正误：数组的元素是不能重复的。

6. 判断正误：字典由键和值两个集合构成，键集合中的元素不能重复，值集合中的元素可以重复。

7. 编程题：编写一个程序说明Swift数组的使用。

8. 编程题：假设有一个类的定义如下。

```
class Employee {
    var name : String          // 姓名
    var salary : Double        // 工资
    init (n : String) {
        name = n
        salary = 0
    }
}
```

编写一个程序说明Swift字典的使用。

第 9 章　函数

我们将程序中反复执行的代码封装到一个代码块中，这个代码块模仿了数学中的函数，具有函数名、参数和返回值。

Swift中的函数很灵活，它可以独立存在，即全局函数；也可以存在于别的函数中，即函数嵌套；还可以存在于类、结构体和枚举中，即方法。

9.1 使用函数

使用函数首先需要定义函数，然后在合适的地方调用该函数，函数的语法格式如下：

```
func 函数名(参数列表) -> 返回值类型 {
    语句组
    return 返回值
}
```

在Swift中定义函数时，关键字是func，函数名需要符合标识符命名规范；多个参数列表之间可以用逗号（,）分隔，极端情况下可以没有参数。参数列表语法如图9-1所示。

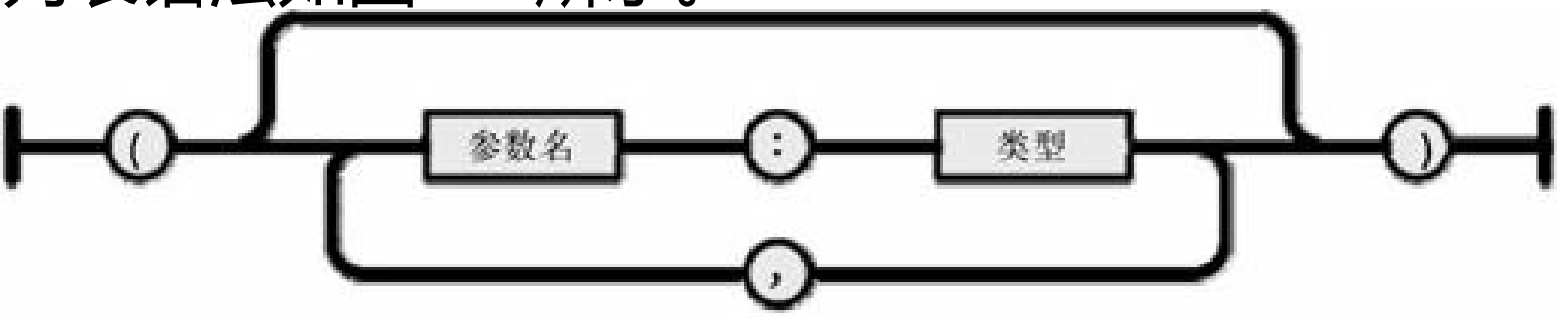


图 9-1 参数列表语法

在参数列表后使用箭头“->”指示返回值类型。返回值有单个值和多个值，多个值返回可以使用元组类型实现。如果函数没有返回值，则“-> 返回值类型”部分可以省略。对应地，如果函数有返回值，就需要在函数体最后使用return语句将

计算的值返回；如果没有返回值，则函数体中可以省略return语句。

函数定义示例代码如下：

```
func rectangleArea(width:Double, height:Double)
-> Double {          ①
    let area = width * height
    return area
②
}

println("320x480的长方形的面积:\
(rectangleArea(320, 480))")          ③
```

上述代码第①行是定义计算长方形的面积rectangleArea，它有两个Double类型的参数，分别是长方形的宽和高。函数的返回类型也是Double。第②行代码是返回函数计算结果。调用函数的过程是通过代码第③行中的rectangleArea(320, 480)语句实现的，调用函数时候需要指定函数名和参数值。

9.2 传递参数

Swift中的函数很灵活，具体体现在传递参数有多种形式。这一节我们介绍几种不同形式的参数。

9.2.1 使用外部参数名

如果我们定义的函数有很多参数，它们又具有相同的数据类型，如果没有清晰的帮助说明，调用者很难知道参数的含义是什么，还记得上一节计算长方形面积的函数rectangleArea吗？我们是这样调用的：

```
println("320x480的长方形的面积:\
(rectangleArea(320, 480))")
```

从调用的代码中，我们很难看出320和480代表的含义。为了提高程序的可读性，我们可以为函数中的参数提供一个外部参数名，首先看看上一节rectangleArea函数的定义，代码如下：

```
func rectangleArea(width:Double, height:Double)
-> Double {
    let area = width * height
    return area
```

```
}
```

其中，参数width和height是函数名的一部分，但是它们只能在函数的内部使用，称为**局部参数名**。我们还可以为每个参数提供一个可以在函数外部使用的名称，称为**外部参数名**，修改rectangleArea函数的定义如下：

```
func rectangleArea(W width:Double, H
height:Double) -> Double {
    let area = width * height
    return area
}
```

我们在局部参数名之前给一个外部参数名，用空格分隔。定义代码中的W和H就是外部参数名。调用代码如下：

```
println("320x480的长方形的面积:\
(rectangleArea(W:320, H:480))")
```

如果我们提供了外部参数名，那么在函数调用

时，必须使用外部参数名，所以W和H不能省略。

外部参数名就像是一个人的“大名”，是让外面人叫的；局部参数名就像是一个人的“小名”，是让家里人叫的。但是也有些人名，在外面和家里是同一个。在rectangleArea函数中，其实局部参数名width和height听起来也不错，含义也很清晰，程序可读性好。我们就可以把它们既作为局部参数名又作为外部参数名使用。修改rectangleArea函数的定义如下：

```
func rectangleArea(#width:Double,
#height:Double) -> Double {
    let area = width * height
    return area
}
```

使用#替代外部参数名，当调用它的时候，可以把局部参数名作为外部参数名使用。调用代码如下：

```
println("320x480的长方形的面积:\
(rectangleArea(width:320, height:480))")
```

其中，width和height是局部参数名，但也可以作为外部参数名使用了。

9.2.2 参数默认值

我们在定义函数的时候可以为参数设置一个默认值，当调用函数的时候可以忽略该参数。看下面的一个示例：

```
func makecoffee(type : String = "卡布奇诺") ->
String {
    return "制作一杯\(type)咖啡。"
}
```

上述代码定义了makecoffee函数，可以帮助我们做一杯香浓的咖啡。由于我喜欢喝卡布奇诺，我就把它设置为默认值。在参数列表中，默认值可以跟在参数的后面，通过等号赋值。

在调用的时候，如果调用者传递了参数，则是其传递过来的值，如果没有传递，则是这个默认值。调用代码如下：

```
let coffee1 = makecoffee(type: "拿铁")
①
```

```
let coffee2 = makecoffee()
②
```

其中第①行代码是传递"拿铁"参数，这种参数要求在前面加上参数名，即type: "拿铁"的形式，注意参数名不能省略。第②行代码没有传递参数，因此使用默认值。最后输出结果如下：

```
制作一杯拿铁咖啡。
制作一杯卡布奇诺咖啡。
```

我们还可以为具有默认值的参数添加外部参数名，也可以使用下划线（_）指定外部参数名，请看下面的示例代码：

```
func CircleArea(R radius: Double = 30, _ pi:
Double = 3.14) -> Double {          ①
    let area = radius  radius  pi
    return area
}

println("圆面积：\(CircleArea(R : 50,
3.1415926))")                              ②
```

其中第①行代码是定义函数，第一个参数有外部参数名R，而第二个参数的外部参数名是下划线（_），这样当我们在第②行代码调用该函数的时候，不需要提供第二个参数的外部参数名。

9.2.3 可变参数

Swift中函数的参数个数可以变化，它可以接受不确定数量的输入类型参数，它们具有相同的类型，有点像是传递一个数组。我们可以通过在参数类型名后面加入（...）的方式来指示这是可变参数。

下面看一个示例：

```
func sum(numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total
}
```

上述代码定义了一个sum函数，用来计算传递给

它的所有参数之和。参数列表numbers：Double...表示这是Double类型的可变参数。在函数体中参数numbers被认为是一个Double数组，使用for in循环遍历numbers数组集合，计算它们的总和，然后返回给调用者。

下面是两次调用sum函数代码：

```
sum(100.0, 20, 30)
sum(30, 80)
```

可以看到每次传递参数的个数是不同的。

9.2.4 参数的传递引用

我们在第5章介绍过，参数传递方式有两种：值类型和引用类型。值类型给函数传递的是参数的一个副本，这样在函数的调用过程中不会影响原始数据。引用类型是把本身数据传递过去，这样在函数的调用过程中会影响原始数据。

在众多数据类型中，只有类是引用类型，其他的数据类型如整型、浮点型、布尔型、字符串、元组、集合、枚举和结构体全部是值类型。

如果一定要将一个值类型参数作为引用传递，

也是可以实现的，Swift提供的inout关键字就可以实现。我们看下面的一个示例：

```
func increment(inout value:Double,
amount:Double = 1.0) {      ①
    value += amount
}

var value : Double = 10.0
②

increment(&value)
③
println(value)

increment(&value, amount:100.0)
④
println(value)
```

第①行代码定义了increment函数，这个函数可以计算一个数值的增长，第一个参数value是需要增长的数值，它被设计为inout类型，inout标识的参数被称为输入输出参数，不能使用var或let标识。第二个参数amount是增长量，它的

默认值是1.0。函数没有声明返回值类型，函数体中不需要return语句，事实上要返回的数据已经通过参数value传递回来，没有必要通过返回值返回了。

第②行代码我们声明并初始化了Double类型变量value，由于在函数调用过程中需要修改它，因此不能声明为常量。

第③行代码increment(&value)是调用函数increment，增长量是默认值，其中&value（在变量前面加&符号）是传递引用方式，它在定义函数时，参数标识与inout是相互对应的。

第④行代码increment(&value, amount:100.0)也是调用函数increment，增长量是100.0。

上述代码输出结果如下：

```
11.0
111.0
```

由于是传递引用方式，输出这个结果就很容易

解释了。

9.3 函数返回值

Swift中函数的返回值也是比较灵活的，形式主要有3种：无返回值、单一返回值和多返回值。这一节我们介绍这3种函数返回值的不同形式。

9.3.1 无返回值函数

有的函数只是为了处理某个过程，或者要返回的数据要通过inout类型参数传递出来，这时可以将函数设置为无返回值。所谓无返回结果，事实上是Void类型，即表示没有数据的类型。

无返回值函数的语法格式有如下3种形式：

```
func 函数名(参数列表) {                                    ①
    语句组
}

func 函数名(参数列表) ->() {                               ②
    语句组
}

func 函数名(参数列表) ->Void {                             ③
    语句组
}
```

无返回值函数不需要“return返回值”语句，第①行语法格式很彻底，参数列表后面没有箭头和类型，第②行语法格式->()，表示返回类型是空的元组。第③行语法格式->Void，表示返回类型是Void类型。

第①行的格式函数定义我们在上一节的increment函数中使用过。increment函数修改为②和③格式的语法如下：

```
func increment(inout value:Double,
amount:Double = 1.0) {
    value += amount
}

func increment(inout value:Double,
amount:Double = 1.0) ->() {
    value += amount
}

func increment(inout value:Double,
amount:Double = 1.0) ->Void {
    value += amount
}
```

9.3.2 多返回值函数

有时我们需要函数返回多个值，这可以通过两种方式来实现。一种是在函数定义的时候，将函数的多个参数声明为引用类型传递，这样当函数调用结束时，这些参数的值就变化了。另一种是将返回定义为元组类型。

本节将介绍通过元组类型返回多值的实现。下面看一个示例：

```
func position(dt: Double, speed:(x:Int, y:Int))
-> (x:Int, y:Int) {                    ①

    var posx:Int = speed.x  Int(dt)
②
    var posy:Int = speed.y  Int(dt)
③

    return (posx, posy)
④
}

let move = position(60.0, (10, -5))
⑤
println("物体位移：\(move.x) , \(move.y)")
⑥
```

这个示例是计算物体在指定时间和速度情况下的位移。第①行代码是定义position函数，其中dt: Double参数是时间，参数speed:(x:Int, y:Int)是元组类型，x和y表示分别表示在x轴和y轴上的速度，速度是矢量，是有方向的，负值表示沿x轴或y轴相反的方向运行。position函数的返回值是(x:Int, y:Int)的元组类型。

函数体中的第②行代码var posx:Int = speed.x * Int(dt)计算x方向的位移，第③行代码var posy:Int = speed.y * Int(dt)计算y方向的位移。最后第④行代码return (posx, posy)将计算后的数据返回。

第⑤行代码调用函数，传递的时间是60.0秒，速度是(10, -5)。第⑥行代码打印输出结果，结果如下：

```
物体位移：600 , -300
```

9.4 函数类型

每个函数都有一个类型，使用函数的类型与使用其他数据类型一样，可以声明变量或常量，也可以作为其他函数的参数或者返回值。

我们有如下3个函数的定义：

```
//定义计算长方形面积函数
func rectangleArea(width:Double, height:Double)
-> Double {                    ①
    let area = width  height
    return area
}

//定义计算三角形面积函数
func triangleArea(bottom:Double, height:Double)
-> Double {                    ②
    let area = 0.5  bottom * height
    return area
}

func sayHello() {
③
    println("Hello, World")
}
```

上述代码中，我们定义了两个函数，其中第①

行rectangleArea(width:Double, height:Double) -> Double函数和第②行triangleArea(bottom:Double, height:Double) -> Double函数，函数类型都是(Double, Double) -> Double。第③行sayHello()函数的函数类型是()->()。

9.4.1 作为函数返回类型使用

我们可以把函数类型作为另一个函数的返回类型使用。下面看一个示例：

```
//定义计算长方形面积函数
func rectangleArea(width:Double, height:Double)
-> Double {
    let area = width * height
    return area
}

//定义计算三角形面积函数
func triangleArea(bottom:Double, height:Double)
-> Double {
    let area = 0.5  bottom  height
    return area
}
```

```
func getArea(type : String) -> (Double, Double)
-> Double  {                                    ①

    var returnFunction:(Double, Double) ->
Double                                          ②

    switch (type) {
    case "rect":   //rect 表示长方形
        returnFunction = rectangleArea
③
    case "tria":  //tria 表示三角形
        returnFunction = triangleArea
④
    default:
        returnFunction = rectangleArea
⑤
    }

    return returnFunction
⑥
}

//获得计算三角形面积函数
var area : (Double, Double)  = getArea("tria")
⑦
println("底10 高13，三角形面积：\(area(10,15))")
⑧

//获得计算长方形面积函数
area = getArea("rect")
```

```
⑨
println("宽10 高15，计算长方形面积：\
(area(10,15))")                                    ⑩
```

上述代码第①行定义函数getArea(type : String) -> (Double, Double) -> Double，其返回类型是(Double, Double) -> Double，这说明返回值是一个函数类型。第②行代码声明returnFunction，它的类型是(Double, Double) -> Double函数类型。第③行代码是在类型type为rect（即长方形）的情况下，把上一节定义的rectangleArea函数名赋值给returnFunction变量，这种赋值之所以能够成功是因为returnFunction类型是rectangleArea函数类型。第④行和第⑤行代码也是如此。第⑥行代码将returnFunction变量返回。

第⑦行和第⑨行代码调用函数getArea，返回值area是函数类型。第⑧行和第⑩行中的area(10,15)调用函数，它的参数列表是(Double, Double)。

上述代码运行结果如下：

```
底10 高13，三角形面积：75.0
宽10 高15，计算长方形面积：150.0
```

9.4.2 作为参数类型使用

我们可以把函数类型作为另一个函数的参数类型使用。下面看一个示例：

```
//定义计算长方形面积函数
func rectangleArea(width:Double, height:Double)
-> Double {
    let area = width * height
    return area
}

//定义计算三角形面积函数
func triangleArea(bottom:Double, height:Double)
-> Double {
    let area = 0.5  bottom  height
    return area
}

func getAreaByFunc(funcName : (Double, Double)
-> Double, a:Double, b:Double) -> Double {
①
```

```
    var area = funcName(a,b)
    return area
}

//获得计算三角形面积函数
var result : Double =
getAreaByFunc(triangleArea, 10, 15)
②
println("底10 高13，三角形面积：\(result)")
③

//获得计算长方形面积函数
result = getAreaByFunc(rectangleArea, 10, 15)
④
println("宽10 高15，计算长方形面积：\(result)")
⑤
```

上述代码第①行定义函数getAreaByFunc，它的第一个参数funcName类型是函数类型(Double, Double) -> Double，第二个和第三个参数都是Double类型。函数的返回值是Double类型，是计算获得的几何图形面积。

第②行是调用函数getAreaByFunc，我们给它传递的第一个参数triangleArea是前文定义的计算三角形面积的函数名，第二个参数是三角形的

底边，第三个参数是三角形的高。函数的返回值result是Double，是计算所得的三角形面积。

第③行也是调用函数getAreaByFunc，我们给它传递的第一个参数rectangleArea是前文定义的计算长方形面积的函数名，第二个参数是长方形的宽，第三个参数是长方形的高。函数的返回值result也是Double，是计算所得的长方形面积。

上述代码的运行结果如下：

```
底10 高13，三角形面积：75.0
宽10 高15，计算长方形面积：150.0
```

综上所述，比较本节与上一节示例，可见它们具有相同的结果，都使用了函数类型(Double, Double) -> Double，通过该函数类型调用triangleArea和rectangleArea函数计算几何图形面积。上一节是把函数类型作为函数返回值类型使用，而本节是把函数类型作为函数的参数类型使用。经过前文的介绍，函数类型也没有什么难的，与其他类型的用法是一样的。

9.5 函数重载

函数重载是指多个函数享有相同的名字但是函数类型必须不同的一组函数，它们互相构成重载关系。

提示 Swift的函数类型包括了参数列表类型和返回值类型，例如(Double, Double) -> Double参数类型是由两个Double类型参数列表和Double类型返回值构成的。也就是说，在Swift中函数名相同、参数列表不同或返回值类型不同的函数都可以构成重载。而在C++和Java等语言中，函数重载只与参数列表不同有关，与函数返回值无关。

下面我们来看一个示例：

```
func receive(i : Int)  {
①
    println("接收一个Int类型数据 i=\(i)")
}

func receive(d :Double) {
②
    println("接收一个Double类型数据 d=\(d)")
}
```

```
func receive(x :Int, y :Int)  {
③
    println("接收两个Int类型数据 x=\(x) y=\(y)")
}

func receive(i :Int) ->Int {
④
    println("接收一个Int类型数据 i=\(i), 返回值类型是Int")
    return i*i
}

let a1:Int = receive(10)
⑤
let a2:() = receive(10)
⑥
let a3:Void = receive(10
⑦
let a4:() = receive(10.0)
⑧
let a5:() = receive(10, 10)
⑨
```

运行输出结果如下：

```
接收一个Int类型数据 i=10, 返回值类型是Int
接收一个Int类型数据 i=10
```

```
接收一个Int类型数据 i=10
接收一个Doubel类型数据 d=10.0
接收两个Int类型数据 x=10 y=10
```

上述代码的第①行~第④行定义了一组重载函数receive。需要注意的是，代码第①行和第④行声明的函数receive，它们的参数列表相同，但是返回值类型不同。

函数调用关系如图9-2所示，第⑤行代码let a1:Int = receive(10)调用的是第④行的函数receive，我们需要明确声明变量或常量类型为Int。第⑥行和第⑦行调用声明常量类型分别是()和Void，它们都表示没有返回值。其他函数调用关系不再赘述。

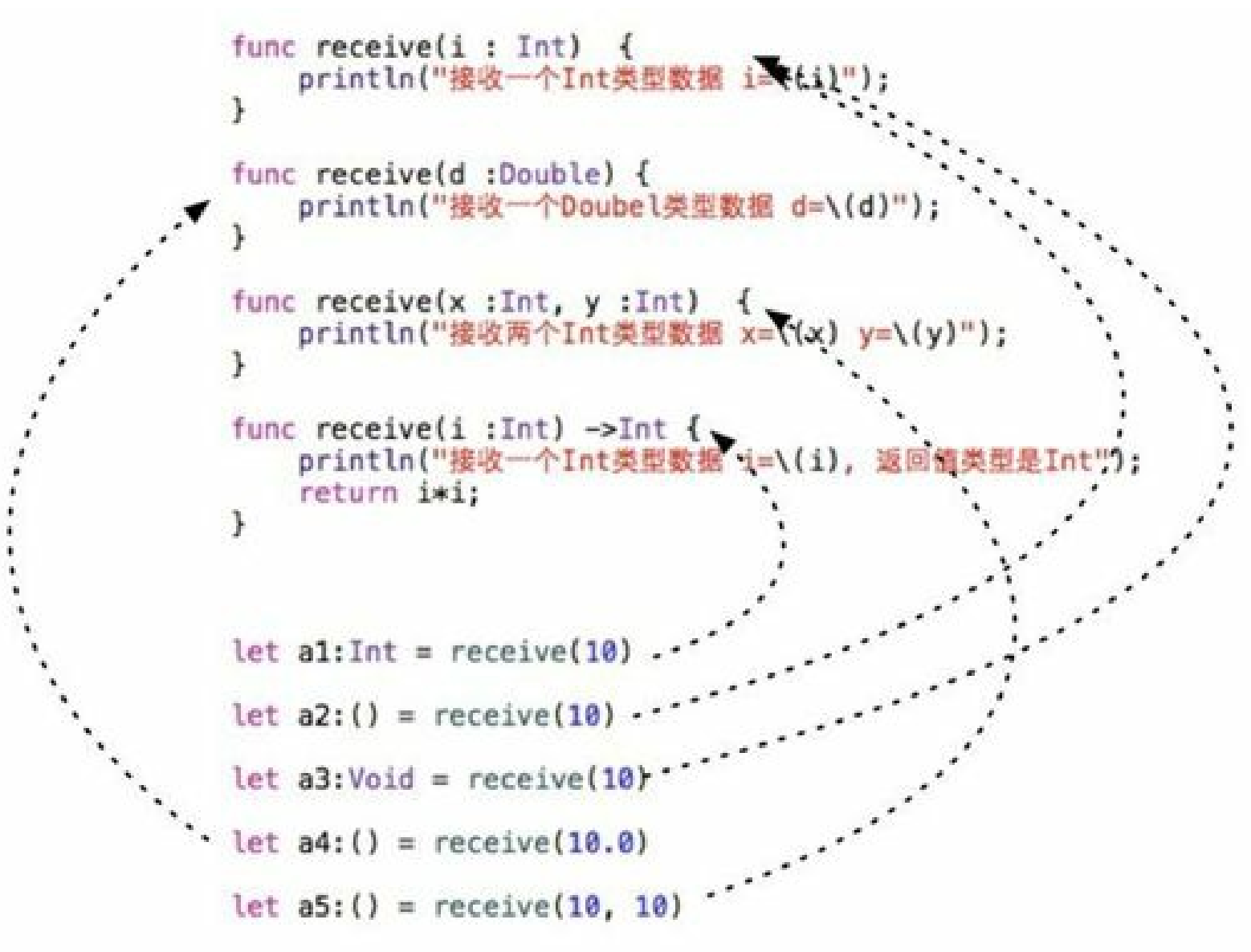

func receive(i : Int) {
println("接收一个Int类型数据 i=\(i)");
}
func receive(d :Double) {
println("接收一个Doubel类型数据 d=\(d)");
}
func receive(x :Int, y :Int) {
println("接收两个Int类型数据 x=\(x) y=\(y)");
}
func receive(i :Int) ->Int {
println("接收一个Int类型数据 i=\(i), 返回值类型是Int");
return i*i;
}
let a1:Int = receive(10)
let a2:() = receive(10)
let a3:Void = receive(10)
let a4:() = receive(10.0)
let a5:() = receive(10, 10)

图 9-2　调用重载函数

9.6 嵌套函数

在此之前我们定义的函数都是全局函数，它们定义在全局作用域中，我们也可以把函数定义在另外的函数体中，称作**嵌套函数**。

下面我们来看一个示例：

```
func calculate(opr :String)-> (Int,Int)-> Int {
①

    //定义+函数
    func add(a:Int, b:Int) -> Int {
②
        return a + b
    }
    //定义-函数
    func sub(a:Int, b:Int) -> Int
③
        return a - b
    }

    var result : (Int,Int)-> Int

    switch (opr) {
    case "+" :
        result = add
④
    case "-" :
        result = sub
```

```
⑤
    default:
        result = add
⑥
    }
    return result
⑦
}

let f1:(Int,Int)-> Int = calculate("+")
⑧
println("10 + 5 = \(f1(10,5))")

let f2:(Int,Int)-> Int = calculate("-")
⑨
println("10 + 5 = \(f2(10,5))")
```

上述代码第①行定义了calculate函数，它的作用是根据运算符进行数学计算，参数opr是运算符，返回值是函数类型(Int,Int)-> Int。在calculate函数体内，第②行定义了嵌套函数add，对两个参数进行加法运算。第③行定义了嵌套函数sub，对两个参数进行减法运算。第④行代码是在运算符为“+”号的情况下将add函数名赋值给函数类型变量result，第⑤行代码是在运

算符为“-”号的情况下将sub函数名赋值给函数类型变量result。第⑥行代码返回函数变量result，第⑦行代码调用calculate函数进行加法运算。第⑧行代码调用calculate函数进行减法运算。

程序运行结果如下：

```
10 + 5 = 15
10 - 5 = 5
```

在函数嵌套中，默认情况下嵌套函数的作用域在外函数体内，但我们可以定义外函数的返回值类型为嵌套函数类型，从而将嵌套函数传递给外函数，被其他调用者使用。

9.7 泛型和泛型函数

泛型（generic）可以使我们在程序代码中定义一些可变的部分，在运行的时候指定。使用泛型可以最大限度地重用代码、保护类型的安全以及提高性能。在Swift集合类中，已经采用了泛型。

9.7.1 一个问题的思考

怎样定义一个函数来判断两个参数是否相等呢？

如果参数是Int类型，则函数定义如下：

```
func isEqualsInt(a:Int, b:Int) -> Bool {
    return (a == b)
}
```

这个函数参数列表是两个Int类型，它只能比较两个Int类型参数是否相等。如果我们想比较两个Double类型是否相等，可以修改上面定义的函数如下：

```
func isEqualsDouble(a:Double, b:Double) -> Bool
{
    return (a == b)
```

```
}
```

这个函数参数列表是两个Double类型，它只能比较两个Double类型参数是否相等。如果我们想比较两个String类型是否相等，可以修改上面定义的函数如下：

```
func isEqualsString(a:String, b:String) -> Bool
{
    return (a == b)
}
```

以上我们分别对3种不同的类型进行了比较，定义了类似的3个函数。那么我们是否可以定义1个函数能够比较3种不同的类型呢？如果isEqualsInt、isEqualsDouble和isEqualsString这3个函数名字后面的Int、Double和String是可变的，那么这些可变部分是与参数类型关联的。

9.7.2 泛型函数

我们可以改造上面的函数，修改内容如下：

```
func isEquals<T>(a: T, b: T) -> Bool {        ①
    return (a == b)
}
```

在函数名isEquals后面添加<T>，参数的类型也被声明为T，T称为占位符，函数在每次调用时传入实际类型才能决定T所代表的类型。如果有多个不同类型，可以使用其他大写字母，一般情况下我们习惯于使用U字母，但是你也可以使用其他的字母。多个占位符用逗号“,”分隔，示例如下：

```
func isEquals<T, U>(a: T, b: U) -> Bool {...}
```

占位符不仅仅可以替代参数类型，还可以替代返回值类型。示例代码如下：

```
func isEquals<T>(a: T, b: T) -> T {...}
```

事实上，上面第①行的函数在编译时会有错误发生，这是因为并不是所有的类型都具有“可比性”，它们必须遵守Comparable协议实现类型。

Comparable协议表示可比较的，在Swift中，基本数据类型以及字符串都是遵守Comparable协议的。

修改代码如下：

```
func isEquals<T: Comparable>(a: T, b: T) ->
Bool {    ②
    return (a == b)
}
```

我们需要在T占位符后面添加冒号和协议类型，这种表示方式被称为**泛型约束**，它能够替换T的类型。在本例中，T的类型必须遵守Comparable协议的具体类。

我们可以通过下列代码测试第②行代码定义的函数：

```
let n1 = 200
let n2 = 100

println(isEquals(n1, n2))

let s1 = "ABC1"
let s2 = "ABC1"
```

```
println(isEquals(s1, s2))
```

分别传递两个Int参数和String参数进行比较，运行结果如下：

```
false
true
```

运行结果无需解释了。泛型在很多计算机语言中都有采用，基本含义都是类似的，但是小的差别还是有的。

9.8　本章小结

通过对本章内容的学习，我们可以了解到Swift语言的函数，其中包括如何使用函数、如何进行参数传递、函数返回值、函数类型、函数重载和嵌套函数等内容。

9.9 同步练习

1. 下列函数定义不正确的是（　　）。

A.

```
func count(string: String) -> (vowels: Int,
consonants: Int, others: Int) {
    return  (1, 2, 3)
}
```

B.

```
func count(string: String) -> () {
}
```

C.

```
func count2(string: String) {
}
```

D.

```
func count3(String string) {
```

```
}
```

2. 下列关于函数参数列表的写法正确的是（　　）。

A.

```
func rectangleArea(W width:Double, H
height:Double) -> Double {
    let area = width * height
    return area
}
```

B.

```
func rectangleArea(W width:Double, H
height:Double) -> Double {
    let area = width * height
    return area
}
```

C.

```
func rectangleArea(#width:Double,
```

```
#height:Double) -> Double {
    let area = width * height
    return area
}
```

D.

```
func rectangleArea(Double width, Double height)
-> Double {
    let area = width * height
    return area
}
```

3. 简答题：请写一个最简单形式的函数。

4. 填空题：请在下列代码横线处填写一些代码，使之能够正确运行。

```
func test1____
{

}
test1("Ravi")
```

5. 有下列函数toLower定义代码：

```
func toLower(#string: String) ->String
{
    return ""
}
```

下列调用语句正确的是（　　）。

A. toLower(string:"Ravi")　　B. toLower(string:"Ravi")

C. toLower(#:"Ravi")　　D. toLower(#:"Ravi")

6. 有下列函数join定义代码：

```
func join(str1:String,str2:String,
with:String="") ->String
{
    return str1+with+str2
}
```

下列调用语句正确的是（　　）。

A. var out1 = join("Hello","World",with:",")

B. `var out2 = join("Hello","World")`

C. `join("Hello","World",with:"-")`

D. `join("Hello","World",with:"#")`

7. 有下列函数sum定义代码：

```
func printNumbers(numbers: Int...)
{
    for number in numbers
    {
        println(number)
    }
}
```

下列调用语句正确的是（　　）。

A. `printNumbers(1,2)`　　B. `printNumbers(1,2,3,4,5,6)`

C. `printNumbers(100.0, 20, 30)`

D. `printNumbers(30.0f)`

8. 有下列函数sum定义代码：

```
func swapNumbers(inout x: Int, inout y: Int)
{
    let temp = x
```

```
    x=y
    y=temp
}

var x: Int = 1
var y: Int = 2
```

下列调用语句正确的是（　　）。

A. `swapNumbers(x,y)`

B. `swapNumbers(&x,&y)`

C. `swapNumbers(inout:&x, inout:&y)`

D. `swapNumbers(inout:x, inout:y)`

9. 填空题：请在下列代码横线处填写一些代码，使之能够正确运行。

```
func addNumber(a:Int,b:Int) ->Int
{
    return a+b
}
var mathFunction: _____
mathFunction = addNumber

var sum = mathFunction(1,2)
```

10. 填空题：请在下列代码横线处填写一些代码，使之能够正确运行。

```
func print____
{

 }

 print(1,"Ravi")
```

11. 填空题：请在下列代码横线出填写一些代码，使之能够正确运行。

```
func addNumber(a:Int,b:Int) -> Int
{
    return a+b
}

func add() -> _____
{
    return addNumber
}

var out = add()(1,2)
```

12. 下列程序的运行结果是（　　）。

```
func addNumber(a:Int,b:Int) -> Int
{
    func print()
    {
        println("a:\(a) b:\(b)")
    }

    print()
    return a+b
}

println(addNumber(10,20))
```

A. a:10 b:20　　B. 30　　C. a:10 b:20　　D. 30

30　　a:10 b:20

13. 编程题：给定一个无序数值，编写一个函数对数组进行排序。

第 10 章　闭包

闭包是一个相对复杂的计算机命题，它的概念很抽象。在本章的开始，我们打算先不马上抛出闭包的概念，而是从一些示例入手，逐渐引入闭包的概念。

10.1 回顾嵌套函数

一门计算语言要支持闭包的前提有两个。

- 支持函数类型，能够将函数作为参数或返回值传递。
- 支持函数嵌套。

这两个前提在Swift中都是满足的，我们先回顾一下9.6节中嵌套函数的示例，通过这个示例，来了解一下闭包的概念以及闭包与函数类型和函数嵌套之间的内在关系。

还记得9.6节中的示例吗？如下所示：

```
func calculate(opr :String)-> (Int,Int)-> Int {

    //定义+函数
    func add(a:Int, b:Int) -> Int {
        return a + b
    }
    //定义-函数
    func sub(a:Int, b:Int) -> Int {
        return a - b
    }

    var result : (Int,Int)-> Int
```

```
    switch (opr) {
    case "+" :
        result = add
    case "-" :
        result = sub
    default:
        result = add
    }
    return result
}
```

该示例定义了calculate函数，并且在calculate函数中定义了嵌套函数add和sub。calculate函数的返回值是(Int,Int)->Int函数类型。

10.2 闭包的概念

在Swift中，可以通过以下代码替代9.6节中的示例代码。

```
func calculate(opr :String)-> (Int,Int)-> Int {

    var result : (Int,Int)-> Int

    switch (opr) {
    case "+" :
        result = {(a:Int, b:Int) -> Int in
①
            return a + b
        }
    default:
        result = {(a:Int, b:Int) -> Int in
②
            return a - b
        }
    }
    return result
}

let f1:(Int,Int)-> Int = calculate("+")
println("10 + 5 = \(f1(10,5))")

let f2:(Int,Int)-> Int = calculate("-")
println("10 + 5 = \(f2(10,5))")
```

原来的嵌套函数add和sub被表达式①和②替代。整理出来就是以下两种形式：

```
{(a:Int, b:Int) -> Int in                    //替代函数add
    return a + b
}

{(a:Int, b:Int) -> Int in                    //替代函数sub
  return a - b
}
```

事实上我们还可以把它们写成一行，如下所示：

```
{(a:Int, b:Int) -> Int in return a + b }
//替代函数add

{(a:Int, b:Int) -> Int in return a - b }
//替代函数sub
```

可以看到代码①和②的表达式就是Swift中的闭包表达式。

通过以上示例的演变，我们可以给Swift 中的闭包一个定义：**闭包**是自包含的匿名函数代码块，可以作为表达式、函数参数和函数返回值，闭包表达式的运算结果是一种函数类型。

Swift中的闭包类似于C和Objective-C 中的代码块、C++和C#中的Lambda表达式、Java中的匿名内部类。

Swift中的闭包可以捕获和存储其所在上下文环境中的常量和变量。这种引用事实上会引起比较麻烦的内存管理问题，好在Swift不需要程序员管理内存。

10.3 使用闭包表达式

Swift中的闭包表达式很灵活，其标准语法格式如下：

```
{ （参数列表） ->返回值类型 in
    语句组
}
```

其中，参数列表与函数中的参数列表形式一样，返回值类型类似于函数中的返回值类型，但不同的是后面有in关键字。

Swift提供了多种闭包简化写法，这一节我们将介绍几种不同的形式。

10.3.1 类型推断简化

类型推断是Swift的强项，Swift可以根据上下文环境推断出参数类型和返回值类型。以下代码是标准形式的闭包：

```
{(a:Int, b:Int) -> Int in
    return a + b
}
```

Swift能推断出参数a和b是Int类型，返回值也是Int类型。简化形式如下：

```
{a, b in return a + b }
```

使用这种简化方式修改后的示例代码如下：

```
func calculate(opr :String)-> (Int,Int)-> Int {

    var result : (Int,Int)-> Int

    switch (opr) {
    case "+" :
        result = {a, b in return a + b }
①
    default:
        result = {a, b in return a - b }
②
    }
    return result
}

let f1:(Int,Int)-> Int = calculate("+")
println("10 + 5 = \(f1(10,5))")

let f2:(Int,Int)-> Int = calculate("-")
```

```
println("10 + 5 = \(f2(10,5))")
```

上述代码第①行和第②行的闭包是上一节示例的简化写法，其中a和b是参数，return后面是返回值。怎么样？很简单吧？

10.3.2 隐藏return关键字

如果在闭包内部语句组只有一条语句，如return a + b等，那么这种语句都是返回语句。前面的关键字return可以省略，省略形式如下：

```
{a, b in a + b }
```

使用这种简化方式修改后的示例代码如下：

```
func calculate(opr :String)-> (Int,Int)-> Int {
    var result : (Int,Int)-> Int

    switch (opr) {
    case "+" :
        result = {a, b in a + b }
①
    default:
```

```
        result = {a, b in a - b }
②
    }
    return result
}
```

上述代码第①行和第②行的闭包return关键字省略了，需要注意的是，省略的前提是闭包中只有一条return语句。下面这样有多条语句是不允许的。

```
{a, b in var c; a + b }
```

10.3.3 缩写参数名称

上一节介绍的闭包表达式已经很简洁了，不过，Swift的闭包还可以再进行简化。Swift提供了参数名称缩写功能，我们可以用$0、$1、$2来表示调用闭包中参数，$0指代第一个参数，$1指代第二个参数，$2指代第三个参数，以此类推$n+1指代第n个参数。

使用参数名称缩写，还可以在闭包中省略参数列表的定义，Swift能够推断出这些缩写参数的类

型。此外，in关键字也可以省略。参数名称缩写之后如下所示：

```
{$0 + $1}
```

使用参数名称缩写修改后的示例代码如下：

```
func calculate(opr :String)-> (Int,Int)-> Int {

    var result : (Int,Int)-> Int

    switch (opr) {
    case "+" :
        result = {$0 + $1}
①
    default:
        result = {$0 - $1}
②
    }
    return result
}

let f1:(Int,Int)-> Int = calculate("+")
println("10 + 5 = \(f1(10,5))")

let f2:(Int,Int)-> Int = calculate("-")
```

```
println("10 + 5 = \(f2(10,5))")
```

上述代码第①行和第②行的闭包采用了参数名称缩写。

10.3.4 使用闭包返回值

闭包表达本质上是函数类型，是有返回值的，我们可以直接在表达式中使用闭包的返回值。重新修改add和sub闭包，示例代码如下：

```
let c1:Int = {(a:Int, b:Int) -> Int in
                return a + b
            }(10,5)
①

println("10 + 5 = \(c1)")

let c2:Int = {(a:Int, b:Int) -> Int in
                    return a - b
                }(10,5)
②

println("10 - 5 = \(c2)")
```

上述代码有两个表达式，第①行代码是给c1赋值，后面是一个闭包表达式。但是闭包表达式不能直接赋值给c1，因为c1是Int类型，需要闭包的返回值。这就需要在闭包结尾的大括号后面接一对小括号(10,5)，通过小括号(10,5)为闭包传递参数。第②行代码也是如此。通过这种方法可以为变量和常量直接赋值，在有些场景下使用非常方便。

10.4 使用尾随闭包

闭包表达式可以作为函数的参数传递，如果闭包表达式很长，就会影响程序的可读性。**尾随闭包**是一个书写在函数括号之后的闭包表达式，函数支持将其作为最后一个参数调用。

下面我们来看一个示例代码：

```
func calculate(opr:String, funN:(Int,Int)->
Int) {                    ①

    switch (opr) {
    case "+" :
        println("10 + 5 = \(funN(10,5))")
    default:
        println("10 - 5 = \(funN(10,5))")
    }
}

calculate("+", {(a:Int, b:Int) -> Int in return
a + b })            ②
calculate("-"){(a:Int, b:Int) -> Int in return
a + b }             ③
```

上述代码第①行是定义calculate函数，其中最后一个参数funN是(Int,Int)-> Int函数类

型，funN可以接收闭包表达式。第②行代码就是调用过程，{(a:Int, b:Int) -> Int in return a + b }是传递的参数。这个参数很长，我们可以通过第③行代码调用，将闭包表达式移到()之外，这种形式就是尾随闭包。

需要注意的是，闭包必须是参数列表的最后一个参数，函数采用如下形式定义：

```
func calculate(funN:(Int,Int)-> Int,
opr:String) {
......
}
```

如果闭包表达式不是最后一个，那么是不能使用尾随闭包写法的。

10.5 捕获上下文中的变量和常量

嵌套函数或闭包可以访问它所在上下文的变量和常量，这个过程称为**捕获值**（capturing value）。即便是定义这些常量和变量的原始作用域已经不存在，嵌套函数或闭包仍然可以在函数体内或闭包体内引用和修改这些值。

下面看一个示例：

```
func makeArray() -> (String)-> [String] {
①

    var ary: [String] = [String]()
②

    func addElement(element:String) ->[String]
{      ③
        ary.append(element)
④
        return ary
⑤
    }
    return addElement
⑥
}

let f1 = makeArray()
⑦
```

```
println("---f1---")
println(f1("张三"))
println(f1("李四"))
println(f1("王五"))

println("---f2---")
⑧
let f2 = makeArray()
println(f2("刘备"))
println(f2("关羽"))
println(f2("张飞"))
```

在上述代码中，第①行定义函数makeArray，它的返回值是(String)->String[]函数类型。第②行声明并初始化了数组变量ary，它的作用域是makeArray函数体。第③行代码定义了嵌套函数addElement，在它的函数体内，第④行代码改变变量ary值，ary变量相对于函数addElement而言，是上下文中的变量。第⑤行代码是从函数体中返回变量ary。第⑥行代码是返回函数类型调用addElement。

这样当在第⑦行调用的时候，f1是嵌套函数addElement的一个实例。需要注意的是，f1每

次调用的时候，变量ary值都能够被保持。第⑧行代码f2也是嵌套函数addElement的一个实例。运行结果如下：

```
---f1---
[张三]
[张三, 李四]
[张三, 李四, 王五]
---f2---
[刘备]
[刘备, 关羽]
[刘备, 关羽, 张飞]
```

f1与f2是嵌套函数addElement的不同实例，它们的运行结果也是独立的。

10.6　本章小结

通过对本章内容的学习，我们可以了解到Swift语言的闭包，其中包括了闭包的概念、闭包表达式、尾随闭包和捕获值等内容。

10.7 同步练习

1. 下列选项中，正确表示闭包定义的是（　　）。

A.

```
{ （参数列表） ->返回值类型
    语句组
}
```

B.

```
{ （参数列表） ->返回值类型 in
    语句组
}
```

C.

```
{ （参数列表） ->返回值类型
    语句组
}
```

D.

```
{ (参数列表)  in
    语句组
}
```

2. 下列选项中，闭包表达式正确的是（　　）。

A.

```
var testEquality1 : (Int, Int) -> Bool = {
    return $0 == $1
}
```

B.

```
var testEquality2 : (Int, Int) -> Bool = {
    $0 == $1
}
```

C.

```
var testEquality3 : (Int, Int) -> Bool = {
    (a : Int, b : Int) -> Bool in
    return a == b
```

```
}
```

D.

```
var testEquality4 : (Int, Int) -> Bool = {
    (a : Int, b : Int) -> Bool
    return a == b
}
```

3. 下列选项中，使用闭包表达式实现两个数相减的是（　　）。

A.

```
var DoMath: (Int, Int) -> Int = {(a:Int, b:Int) -> Int in
    return a-b
}
```

B.

```
var DoMath: (Int, Int) -> Int = {(a, b) return a-b}
```

C.

```
var DoMath: (Int, Int) -> Int = {(a, b) in
return a-b}
```

D.

```
var DoMath: (Int, Int) -> Int = {$0-$1}
```

4. 下面是一个函数的定义：

```
func applyMutliplication(value: Int,
multFunction: Int -> Int) -> Int {
    return multFunction(value)
}
```

能正确地调用applyMutliplication函数的语句是（　　）。

A.

```
applyMutliplication(2, {value in
    value * 3
```

```
    })
```

B.

```
applyMutliplication(2, {value in
    return value * 3
    })
```

C.

```
applyMutliplication(2, {$0 * 3})
```

D.

```
applyMutliplication(2) {$0 * 3}
```

5. 数组类型有一个sort方法可以实现参数数组元素的排序。给出一个数组定义var array = [3,2,4,1]，那么下列排序语句正确的是（　　）。

A. `array.sort({ (item1: Int, item2: Int) -> Bool in return item1 < item2 })`

B. `array.sort({ (item1, item2) -> Bool in return item1 < item2 })`

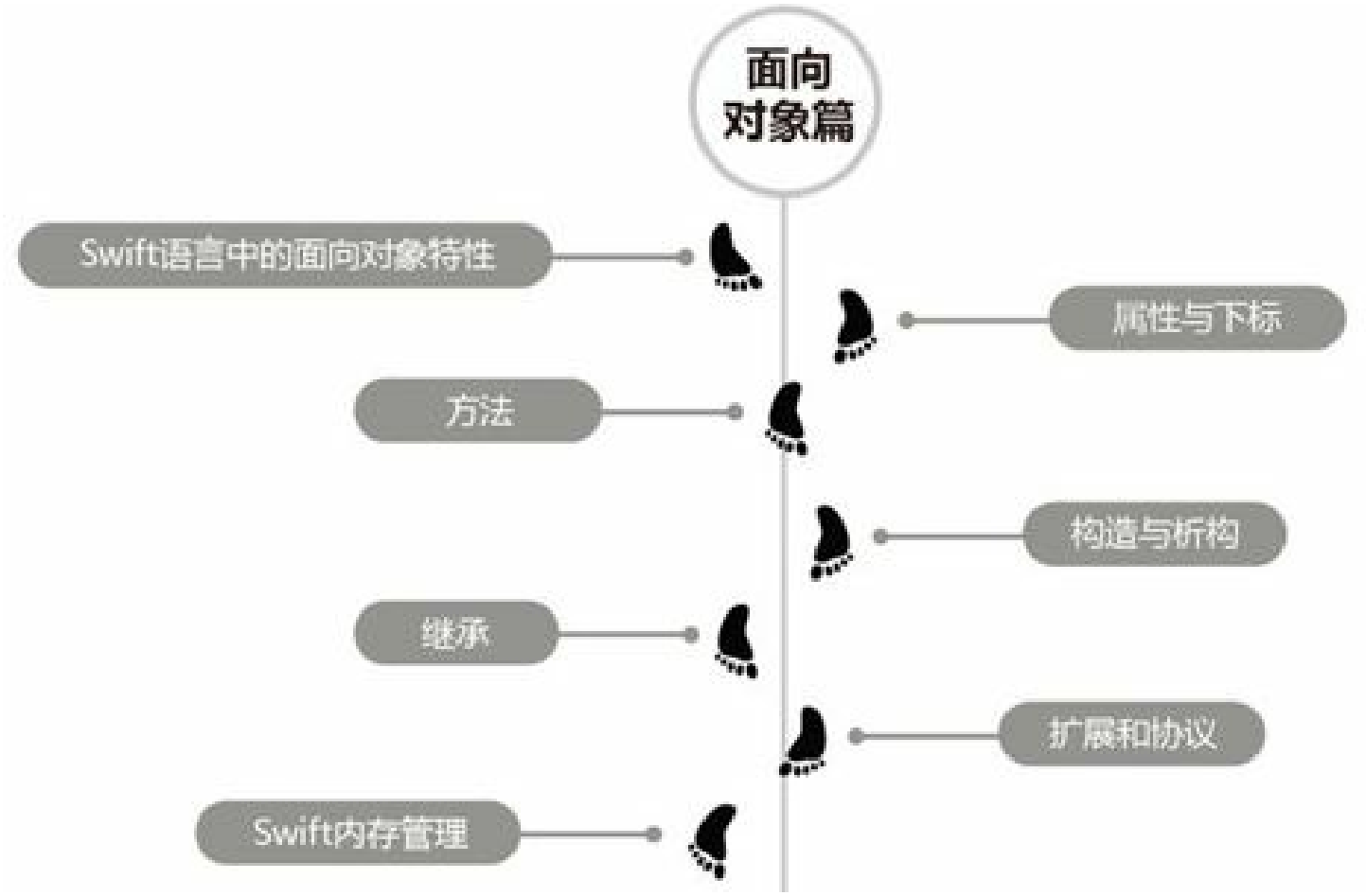
C. `array.sort({ (item1, item2) in return item1 < item2 })`

D. `array.sort { (item1, item2) in return item1 < item2 }`

E. `array.sort { return $0 < $1 }`

F. `array.sort { $0 < $1 }`

第二部分　面向对象篇



第 11 章　Swift语言中的面向对象特性

在现代计算机语言中，面向对象是非常重要的特性，Swift语言也提供了面向对象的支持。而且在Swift语言中，不仅类具有面向对象特性，结构体和枚举也都具有面向对象特性。

11.1 面向对象概念和基本特征

面向对象（OOP）是现代流行的程序设计方法，是一种主流的程序设计规范。其基本思想是使用对象、类、继承、封装、属性、方法等基本概念来进行程序设计。从现实世界中客观存在的事物出发来构造软件系统，并且在系统构造中尽可能运用人类的自然思维方式。

例如，在现实世界的学生管理系统中，张三同学和李四同学是现实世界中客观存在的实体，他们有学号、姓名、班级等属性，他们有学习、问问题以及吃饭走路等动作（或操作）。如果我们要开发一个学生管理软件系统，那么张三同学和李四同学是对象，将他们共同的属性和方法（操作）归纳和总结后，得出一个抽象的描述就是**类**（class），即学生类。

OOP的基本特征包括：封装性、继承性和多态性。

- **封装性** 封装性就是尽可能隐蔽对象的内部细节，对外形成一个边界，只保留有限的对外接口使之与外部发生联系。
- **继承性** 一些特殊类能够具有一般类的全部属性和方法，这称做特殊类对一般类的

继承。例如客轮与轮船，客轮是特殊类，轮船是一般类。通常我们称一般类为**父类**（或基类），特殊类为**子类**（或派生类）。

- **多态性** 对象的多态性是指在父类中定义的属性或方法被子类继承之后，可以使同一个属性或方法在父类及其各个子类中具有不同的含义，这称为**多态性**。例如动物都有吃饭的方法，但是老鼠的吃饭方法和猫的吃饭方法是截然不同的。

11.2 Swift中的面向对象类型

上一节我们介绍了面向对象，在不同的计算机语言中，其具体的体现也是不同的。在C++和Java等语言中通过类实现面向对象，在Swift语言中通过类和结构体（struct）实现面向对象，在Swift语言中，枚举（enum）也具有面向对象特性。结构体和枚举在其他语言中完全没有面向对象特性，Swift语言赋予了它们面向对象生命。

提示 由于OOP中的类在Swift语言中涵盖了枚举、类和结构体。为了防止与OOP中的类发生冲突，在本书中我们把Swift中的这3种类型称为“Swift面向对象类型”。

在面向对象中，将类创建对象的过程称为**实例化**，因此将对象称为**实例**，但是在Swift中，结构体和枚举的实例不称为“对象”，因为结构体和枚举并不是彻底的面向对象类型，而是只包含了一些面向对象的特点。例如，在Swift中继承只发生在类上，结构体和枚举不能继承。

在Swift中，面向对象的概念还有：属性、方法、扩展和协议等，这些概念对于枚举、类和结构体等不同类型有可能不同，我们会在第12章、第

13章和第16章中再明确说明。

11.3 枚举

在C和Objective-C中，枚举用来管理一组相关常量集合，通过使用枚举可以提高程序的可读性，使代码更清晰，更易于维护。而在Swift中，枚举的作用已经不仅仅是定义一组常量、提高程序的可读性了，它还具有了面向对象特性。

我们先来看Swift声明。Swift中也是使用enum关键词声明枚举类型，具体定义放在一对大括号内，枚举的语法格式如下：

```
enum 枚举名
{
        枚举的定义
}
```

“枚举名”是该枚举类型的名称。它首先应该是有效的标识符，其次应该遵守面向对象的命名规范。它应该是一个名称，如果采用英文单词命名，首字母应该大写，尽量用一个英文单词。这个命名规范也适用于类和结构体的命名。“枚举的定义”是枚举的核心，它由一组成员值和一组相关值组成。

11.3.1 成员值

在枚举类型中定义一组成员，与C和Objective-C中枚举的主要作用是一样的，不同的是，在C和Objective-C中成员值是整数类型，因此在C和Objective-C中枚举类型就是整数类型。

而在Swift中，枚举的成员值默认情况下不是整数类型，以下代码是声明枚举示例：

```
enum WeekDays {
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
}
```

上述代码声明了WeekDays枚举，表示一周中的每个工作日，其中定义了5个成员值：Monday、Tuesday、Wednesday、Thursday和Friday，这些成员值并不是整数类型。

在这些成员值前面还要加上case关键字，也可以将多个成员值放在同一行，用逗号隔开，如下所示：

```
enum WeekDays {
    case Monday, Tuesday, Wednesday, Thursday,
Friday
}
```

下面我们看一个示例，代码如下：

```
var day = WeekDays.Friday                          ①
day = WeekDays.Wednesday                           ②
day = .Monday                                      ③

func writeGreeting(day : WeekDays) {               ④

    switch day {                                   ⑤
    case .Monday:
        println("星期一好！")
    case .Tuesday :
        println("星期二好！")
    case .Wednesday :
        println("星期三好！")
    case .Thursday :
        println("星期四好！")
    case .Friday :
        println("星期五好！")
    }                                              ⑥

}
```

```
writeGreeting(day)                                    ⑦
writeGreeting(WeekDays.Friday)                        ⑧
```

上述代码是使用WeekDays枚举的一个示例，其中第①行代码是把WeekDays枚举的成员值Friday赋值给变量day，第②行和第③行代码也是给变量day赋值，我们可以采用完整的“枚举类型名.成员值”的形式，也可以省略枚举类型采用“.成员值”的形式（见代码第③行）。这种省略形式能够访问的前提是，Swift能够根据上下文环境推断类型。因为我们已经在第①行和第②行给day变量赋值，所以即使第③行代码采用缩写，Swift也能够推断出数据类型是WeekDays。

为了方便反复调用，我们在第④行定义了writeGreeting函数。第⑤~⑥行代码使用了switch语句，枚举类型与switch语句能够很好地配合使用。在switch语句中使用枚举类型可以没有default分支，这在使用其他类型时是不允许的。使用default分支的代码如下：

```
func writeGreeting(day : WeekDays) {
```

```
    switch day {
    case .Monday:
        println("星期一好！")
    case .Tuesday :
        println("星期二好！")
    case .Wednesday :
        println("星期三好！")
    case .Thursday :
        println("星期四好！")
    default:
         println("星期五好！")
    }
}
```

需要注意，在switch中使用枚举类型时，switch语句中的case必须全面包含枚举中的所有成员，不能多也不能少，包括使用default的情况下，default也表示某个枚举成员。在上面的示例中，default表示的是Friday枚举成员，在这种情况下，Friday枚举成员的case分支不能再出现了。

上述代码第⑦行和第⑧行是调用函数writeGreeting，传递的参数可以

是WeekDays变量（见代码第⑦行），也可以是WeekDays中的成员值（见代码第⑧行）。

11.3.2 原始值

出于业务上的需要，要为每个成员提供某种具体类型的默认值，我们可以为枚举类型提供原始值（raw values）声明，这些原始值类型可以是：字符、字符串、整数和浮点数等。

原始值枚举的语法格式如下：

```
enum 枚举名 : 数据类型
{
        case 成员名 = 默认值
        ......
}
```

在“枚举名”后面跟“:”和“数据类型”就可以声明原始值枚举的类型，然后在定义case成员的时候需要提供默认值。

以下代码是声明枚举示例：

```
enum WeekDays : Int {
    case Monday              = 0
```

```
    case Tuesday            = 1
    case Wednesday          = 2
    case Thursday           = 3
    case Friday             = 4
}
```

我们声明的WeekDays枚举类型的原始值类型是Int，需要给每个成员赋值，只要是Int类型都可以，但是每个分支不能重复。我们还可以采用如下简便写法，只需要给第一个成员赋值即可，后面的成员值会依次加1。

```
enum WeekDays : Int {
    case Monday = 0, Tuesday, Wednesday,
Thursday, Friday
}
```

以下是完整的示例代码：

```
var day  = WeekDays.Friday

func writeGreeting(day : WeekDays) {

    switch day {
```

```
    case .Monday:
        println("星期一好！")
    case .Tuesday :
        println("星期二好！")
    case .Wednesday :
        println("星期三好！")
    case .Thursday :
        println("星期四好！")
    case .Friday :
        println("星期五好！")
    }

}

let friday = WeekDays.Friday.toRaw()  ①

let thursday = WeekDays.fromRaw(3)  ②

if (WeekDays.Friday.toRaw() == 4) {  ③
    println("今天是星期五")
}

writeGreeting(day)
writeGreeting(WeekDays.Friday)
```

上述代码与上一节的示例非常类似，相同的地方将不再赘述，我们重点看有标号的代码部分。其

中第①行代码是通过WeekDays.Friday的方法toRaw()转换为原始值。虽然在定义的时候Friday被赋值为4，但是并不等于WeekDays.Friday就是整数4了，而是它的原始值为整数4，因此下面的比较是错误的。

```
if (WeekDays.Friday == 4) {
    println("今天是星期五")
}
```

我们需要使用WeekDays.Friday的原始值进行比较，见代码第③行。

toRaw()方法是将成员值转换为原始值，相反fromRaw()方法是将原始值转换为成员值，见代码第②行。

11.3.3 相关值

在Swift中除了可以定义一组成员值，还可以定义一组相关值（associated values），它有点类似于C中的联合类型。下面看一个枚举类型的声明：

```
enum Figure {
```

```
    case Rectangle(Int, Int)
    case Circle(Int)
}
```

枚举类型Figure（图形）有两个相关值：Rectangle（矩形）和Circle（圆形）。Rectangle和Circle是与Figure有关联的相关值，它们都是元组类型，对于一个特定的Figure实例，只能是其中一个相关值。从这一点来看，枚举类型的相关值类似于C中的联合类型。

下面我们看一个示例，代码如下：

```
func printFigure(figure : Figure) {
①

    switch figure {
②
    case .Rectangle(let width, let height):
        println("矩形的宽:\(width) 高:\
(height)")
    case .Circle(let radius):
        println("圆形的半径：\(radius)")
    }
③
}
```

```
var figure = Figure.Rectangle(1024, 768)
④
printFigure(figure)
⑤

figure = .Circle(600)
⑥
printFigure(figure)
⑦
```

上述代码使用前文声明的枚举类型Figure，为了能够反复调用，我们在代码第①行定义了一个函数。其中代码第②~③行使用了switch语句，为了从相关值中提取数据，可以在元组字段前面添加let或var。如果某个相关值元组中字段类型一致，需要全部提取，则可以在相关值前面添加let或var。我们可以使用如下方式修改Rectangle分支：

```
switch figure {
case let .Rectangle( width, height):
    println("矩形的宽:\(width) 高:\(height)")
case .Circle(let radius):
    println("圆形的半径：\(radius)")
```

```
}
```

上述代码第④行`var figure = Figure.Rectangle(1024, 768)`是声明变量`figure`，并初始化为`Rectangle`类型的相关值。第⑤行代码是调用函数`printFigure`打印输出结果：

```
矩形的宽:1024 高:768
```

代码第⑥行`figure = .Circle(600)`初始化为`Circle`类型的相关值。第⑦行代码调用函数`printFigure`并打印输出结果：

```
圆形的半径：600
```

11.4　结构体与类

在面向过程的编程语言（如C语言）中，结构体用得比较多，但是面向对象之后，如在C++和Objective-C中，结构体已经很少使用了。这是因为结构体能够做的事情，类完全可以取而代之。

而Swift语言却非常重视结构体，把结构体作为实现面向对象的重要手段。Swift中的结构体与C++和Objective-C中的结构体有很大的差别，C++和Objective-C中的结构体只能定义一组相关的成员变量，而Swift中的结构体不仅可以定义成员变量（属性），还可以定义成员方法。因此，我们可以把结构体看做是一种轻量级的类。

Swift中的类和结构体非常类似，都具有定义和使用属性、方法、下标和构造器等面向对象特性，但是结构体不具有继承性，也不具备运行时强制类型转换、使用析构器和使用引用计等能力。

11.4.1　类和结构体定义

Swift中的类和结构体定义的语法也是非常相似的。我们可以使用`class`关键词定义类，使用`struct`关键词定义结构体，它们的语法格式如下：

```
class 类名 {
    定义类的成员
}
struct 结构体名 {
    定义结构体的成员
}
```

从语法格式上看，Swift中的类和结构体的定义更类似于Java语法，不需要像C++和Objective-C那样把接口部分和实现部分放到不同的文件中。

类名、结构体名的命名规范与枚举类型的要求是一样的，具体的命名规范请参考11.3节。下面我们来看一个示例：

```
class Employee {                        //定义员工类
    var no : Int = 0                    //定义员工编号属性
    var name : String = ""              //定义员工姓名属性
    var job : String?                   //定义工作属性
    var salary : Double = 0             //定义薪资属性

    var dept : Department?              //定义所在部门属性
}

struct Department {                     //定义部门结构体
    var no : Int = 0                    //定义部门编号属性
    var name : String = ""              //定义部门名称属性
```

```
}
```

Employee是我们定义的类，Department是我们定义的结构体。在Employee和Department中我们只定义了一些属性。关于属性的内容我们将在下一章介绍。

Employee和Department是有关联关系的，Employee所在部门的属性dept与Department关联起来，它们的类图如图11-1所示。

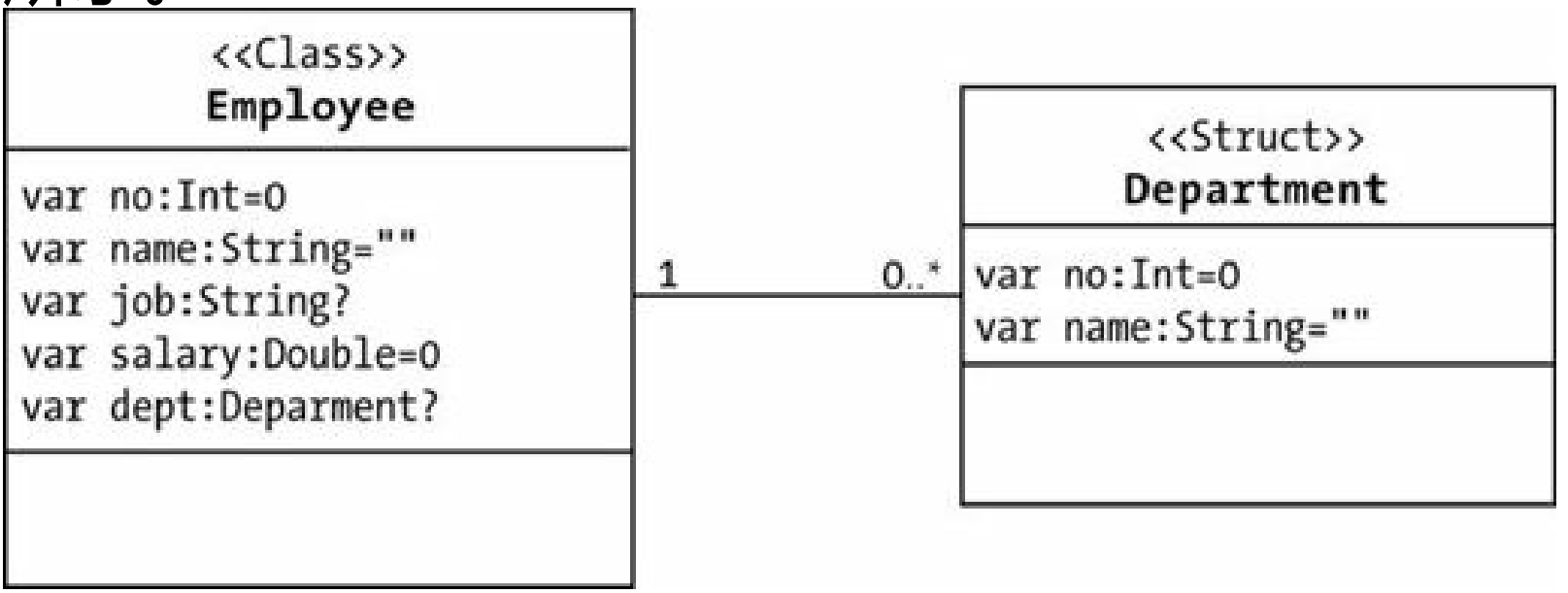


图 11-1　类图

我们可以通过下列语句实例化：

```
var emp = Employee()
var dept = Department()
```

Employee()和Department()是调用它们的构造器实现实例化，关于构造器我们会在14.1节介绍。

提示 实例化之后会开辟内存空间，emp和dept被称为“实例”，但只有类实例化的“实例”才能被称为“对象”。事实上，不仅仅是结构体和类可以实例化，枚举、函数类型和闭包开辟内存空间的过程也可以称为实例化，结果也可以叫“实例”，但不能叫“对象”。

11.4.2 再谈值类型和引用类型

我们在第5章介绍数据类型的时候曾介绍过，数据类型可以分为：值类型和引用类型，这是由赋值或参数传递方式决定的。值类型就是在赋值或给函数传递参数时候，创建一个副本，把副本传递过去，这样在函数的调用过程中不会影响原始数据。引用类型就是在赋值或给函数传递参数的时候，把本身数据传递过去，这样在函数的调用过程中会影响原始数据。

在众多的数据类型中，我们只需记住：只有类是引用类型，其他类型全部是值类型。即便结构体与类非常相似，它也是值类型。值类型还包括整型、浮点型、布尔型、字符串、元组、集合和枚举。

Swift中的引用类型与Java中的引用类型是一样的，Java中的类也是引用类型。如果你没有Java经验，可以把引用类型理解为C、C++和Objective-C语言中的指针类型，只不过不需要在引用类型变量或常量前面加星号（*）。

下面我们看一个示例：

```
var dept = Department()
①
dept.no = 10
dept.name = "Sales"
②

var emp = Employee()
③
emp.no = 1000
emp.name = "Martin"
emp.job = "Salesman"
emp.salary = 1250
emp.dept = dept
④
```

```
func updateDept (dept : Department) {
⑤
    dept.name = "Research"
⑥
}

println("Department更新前:\(dept.name)")
⑦
updateDept(dept)
⑧
println("Department更新后:\(dept.name)")
⑨

func updateEmp (emp : Employee) {
⑩
    emp.job = "Clerk"
⑪
}

println("Employee更新前:\(emp.job)")
⑫
updateEmp(emp)
⑬
println("Employee更新后:\(emp.job)")
⑭
```

上述代码第①~②行创建Department结构体

实例，并设置它的属性。代码第③~④行创建Employee类实例，并设置它的属性。

为了测试结构体是否是值类型，我们在第⑤行代码定义了updateDept函数，它的参数是Department结构体实例。第⑥行代码dept.name = "Research"是改变dept实例。然后在第⑦行打印更新前的部门名称属性，在第⑧行进行更新，在第⑨行打印更新后的部门名称属性。如果更新前和更新后的结果一致，则说明结构体是值类型，反之则为引用类型。事实上第⑥行代码会有编译错误，错误信息如下。

```
Playground execution failed: error:
<REPL>:34:15: error: cannot assign to 'name' in
'dept'
    dept.name = "Research"
    ~~~~~~~~~ ^
```

这个错误提示dept.name = "Research"是不能赋值的，这说明了dept结构体不能修改，因为它是值类型。其实有另外一种办法可以使值类型参数能够以引用类型传递，我们在第9章介绍过使

用inout声明的输入输出类型参数，这里需要修改一下代码：

```
func updateDept (inout dept : Department) {
    dept.name = "Research"
}

println("Department更新前:\(dept.name)")
updateDept(&dept)
println("Department更新后:\(dept.name)")
```

我们不仅要将参数声明为inout，而且要在使用实例前加上&符号。这样修改后输出结果如下：

```
Department更新前:Sales
Department更新后:Research
```

相比之下，第⑩行代码是定义updateEmp函数，它的参数是Employee类的实例，我们不需要将参数声明为inout类型。在第⑪行修改emp没有编译错误，这说明Employee类是引用类型，在调用的时候不用在变量前面添加&符号，见代码第

行。输出结果如下：

```
Employee更新前:Salesman
Employee更新后:Clerk
```

这个结果再次说明了类是引用类。

11.4.3 引用类型的比较

我们在第4章介绍了基本运算符，提到了恒等于（===）和不恒等于（!===）关系运算符。===用于比较两个引用是否为同一个实例，!===则恰恰相反，它只能用于引用类型，也就是类的实例。

下面我们看一个示例：

```
var emp1 = Employee()                    ①
emp1.no = 1000
emp1.name = "Martin"
emp1.job = "Salesman"
emp1.salary = 1250

var emp2 = Employee()                    ②
emp2.no = 1000
emp2.name = "Martin"
emp2.job = "Salesman"
```

```
emp2.salary = 1250


if emp1 === emp2                                    ③
{
    println("emp1 === emp2")
}
if emp1 === emp1                                    ④
{
    println("emp1 === emp1")
}

var dept1 = Department()                            ⑤
dept1.no = 10
dept1.name = "Sales"

var dept2 = Department()                            ⑥
dept2.no = 10
dept2.name = "Sales"

if dept1 == dept2   //编译失败                      ⑦
{
    println("dept1 === dept2")
}
```

上述代码第①行和第②行分别创建了emp1和emp2两个Employee实例。在代码第③行比较

emp1和emp2两个引用是否为一个实例。可以看到，比较结果为False，也就是emp1和emp2两个引用不是一个实例，即便是它们内容完全一样，结果也是False，而第④行的比较结果为True。如果我们采用==比较，结果会如何呢？代码如下：

```
if emp1 == emp2
{
    println("emp1 === emp2")
}
```

答案是有如下编译错误。==比较要求两个实例的类型（类、结构体、枚举等）必须要在该类型中重写==运算符，定义相等规则。同样的错误也会发生在第⑦行代码。

```
Playground execution failed: error:
<REPL>:42:9: error: could not find an overload
for '==' that accepts the supplied arguments
if emp1 == emp2
   ~~~~~^~~~~~~
```

代码第⑤行和第⑥行分别创建了dept1和dept2两个Department实例。在代码第⑦行使用==比较dept1和dept2两个值是否相等，不仅不能比较，而且还会发生编译错误，这在上面已经解释过了。

如果我们采用恒等于===比较dept1和dept2，结果会如何呢？代码如下：

```
if dept1 === dept2
{
    println("dept1 === dept2")
}
```

我们发现会有编译错误。===不能比较值类型，而Department结构体是值类型，因此不能使用===比较。

11.5 类型嵌套

Swift语言中的类、结构体和枚举可以进行嵌套，即在某一类型的{}内部定义类。这种类型嵌套在Java中称为内部类，在C#中称为嵌套类，它们的形式和设计目的都是类似的。

类型嵌套的优点是能够访问它外部的成员（包括方法、属性和其他的嵌套类型），嵌套还可以有多个层次。

下面我们看一个示例：

```
class Employee {
①

    var no : Int = 0
    var name : String = ""
    var job : String = ""
    var salary : Double = 0
    var dept : Department = Department()

    var day : WeekDays = WeekDays.Friday

    struct Department {
②
        var no : Int = 10
        var name : String = "SALES"
    }
```

```
    enum WeekDays {
③
        case Monday
        case Tuesday
        case Wednesday
        case Thursday
        case Friday

         struct Day {
④
             static var message : String =
"Today is..."
         }
    }
}

var emp = Employee()
⑤
println(emp.dept.name)
⑥
println(emp.day)
⑦

let friday = Employee.WeekDays.Friday
⑧
if emp.day == friday {
    println("相等")
}
println(Employee.WeekDays.Day.message)
```

⑨

上述代码第①行定义了Employee类。在Employee类的内部，第②行代码定义了结构体Department，第③行定义了枚举WeekDays。在枚举WeekDays的内部，第④行代码定义了结构体Day。

第⑤行代码实例化Employee返回emp实例，第⑥行代码引用嵌套结构体Department的name属性。第⑦行代码emp.day引用emp实例day属性，它是嵌套枚举类型WeekDays的类型。第⑧行代码Employee.WeekDays.Friday直接引用嵌套枚举类型WeekDays的成员值。第⑨行代码Employee.WeekDays.Day.message引用嵌套结构体Day的静态属性message。关于静态属性的知识，我们将在12.4节详细介绍。

类型嵌套便于我们访问外部类的成员，但它会使程序结构变得不清楚，使程序的可读性变差。

11.6　可选类型与可选链

有时候我们在Swift程序表达式中会看到“?”和“!”等符号，它们代表什么含义呢？这些符号都与可选类型相关，这一节我们就来详细介绍一下。

11.6.1　可选类型

有时候我们使用一个变量或常量，它保存的值可能有也可能没有。例如下列代码：

```
func divide(n1 : Int, n2 : Int) ->Double? {
①
    if n2 == 0 {
        return nil
②
    }
    return Double(n1)/Double(n2)
}
let result : Double? = divide(100, 200)
③
```

上述代码第①行使用了divide函数进行除法运算，在第二个参数n2为零的情况下，函数返回nil，所以第③行代码获得函数返回值要么有值，

要么没有值（等于nil的情况）。为了能够接收这种不确定的返回值，我们需要在类型后面加上问号（？），表示该类型是可选类型，在本例中是Double?。

1. **可选绑定**

可选类型可以用于判断，如下代码所示：

```
if let result2 : Double? = divide(100, 0) {
①
    println("Success.")
} else {
    println("failure.")
}
```

我们在第①行调用函数进行计算，然后把结果直接赋值给变量或常量。如果result2不等于nil，则if语句的逻辑表达式为true。以上代码的判断输出结果如下：

```
failure.
```

这种可选类型在if或while语句中赋值并进行

判断的写法，叫做**可选绑定。**

2. **强制拆封**

如果我们能确定可选类型一定有值，那么在读取它的时候，可以在可选类型的后面加一个感叹号（！）来获取该值。这种感叹号的表示方式称为可选值的**强制拆封**（forced unwrapping）。如下代码所示：

```
let result1 : Double? = divide(100, 200)
println(result1!)
```

`println(result1!)`语句中的`result1!`就进行了强制拆封。

3. **隐式拆封**

为了能够方便地访问可选类型，我们可以将可选类型后面的问号（？）换成感叹号（！），这种可选类型在拆封时变量或常量后面不加感叹号（！）的表示方式称为**隐式拆封**。如下代码所示：

```
let result3 : Double! = divide(100, 200)
println(result3)
```

在变量或常量声明的时候，数据类型Double 后面跟的是感叹号（！）而不是问号（？），在拆封的时候，变量或常量后面不用加感叹号（！），这就是隐式拆封。隐式拆封的变量或常量使用起来就像普通变量或常量一样，你也可以把它看成是普通的变量或常量。

11.6.2　可选链

在介绍可选链之前，我们先看一个类图（见图11-2）。

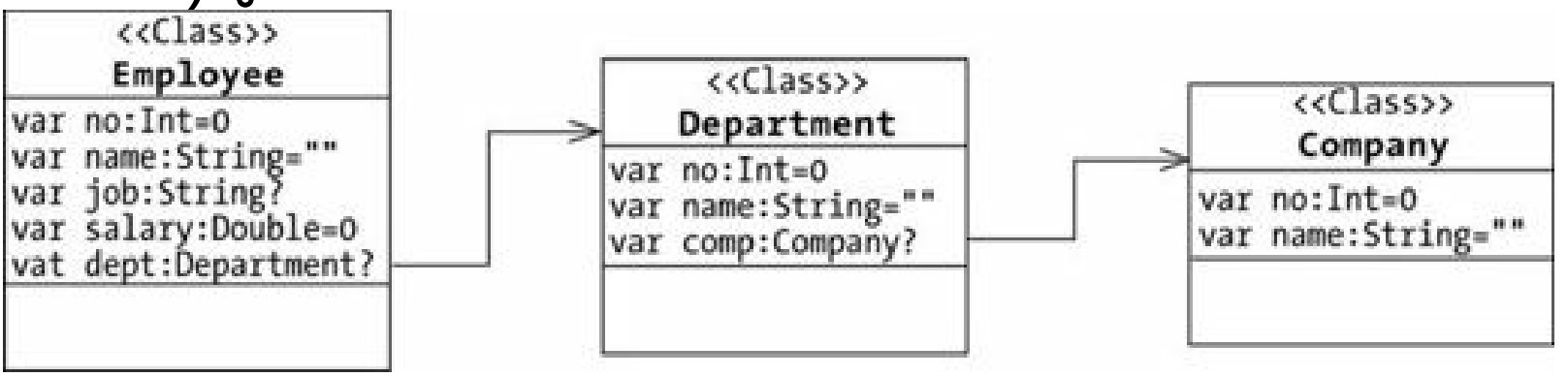


图 11-2　关联关系的类图

这个类图在图11-1类图的基础上增加了公司类（Company），把Department修改为类。它们之间是典型的关联关系类图。这些类一般都是实体类，实体类是系统中的人、事、物。Employee通过dept属性与Department关联，Department通过comp属性与Company关联。

下面看示例代码：

```
class Employee {
①
    var no : Int = 0
    var name : String = "Tony"
    var job : String?
    var salary : Double = 0
    var dept : Department = Department()
②
}

class Department {
③
    var no : Int = 10
    var name : String = "SALES"
    var comp : Company = Company()
④
}

class Company {
⑤
    var no : Int = 1000
    var name : String = "EOrient"
}

var emp = Employee()
⑥
println(emp.dept.comp.name)
⑦
```

上述代码第①行定义了Employee类，第③行定义了Department类，第⑤行定义了Company类。第②行代码var dept : Department = Department()关联到Department类。第④行代码var comp : Company = Company()关联到Company类。

给定一个Employee实例（见第⑥行代码），通过第⑦行代码emp.dept.comp.name可以引用到Company实例，形成一个引用的链条，但是这个“链条”任何一个环节“断裂”（为nil）都无法引用到最后的目标（Company实例）。

事实上，第②行代码是使用Department()构造器实例化dept属性的，这说明给定一个Employee实例，一定会有一个Department与其关联。但是现实世界并非如此，一个新入职的员工未必有部门，这种关联关系有可能有值，也有可能没有值，我们需要使用可选类型（Department?）声明dept属性。第④行代码的comp属性也是类似的。

修改代码如下：

```
class Employee {
    var no : Int = 0
    var name : String = "Tony"
    var job : String?
    var salary : Double = 0
    var dept : Department?                    ①
}

class Department {
    var no : Int = 10
    var name : String = "SALES"
    var comp : Company?                       ②
}
class Company {
    var no : Int = 1000
    var name : String = "EOrient"
}
```

在第①行代码中声明dept为Department?可选类型，第②行代码声明comp为Company? 可选类型。那么原来的引用方式emp.dept.comp.name已经不能应对可选类型了。我们在前面介绍过可选类型的引用，可以使用感叹号（！）进行强制拆封，代码修改如下：

```
println(emp.dept!.comp!.name)
```

但是强制拆封有一个弊端，如果可选链中某个环节为nil，将会导致代码运行时错误。我们可以采用更加“温柔”的引用方式，使用问号（?）来代替原来感叹号（!）的位置，如下所示：

```
println(emp.dept?.comp?.name)
```

问号（?）表示引用的时候，如果某个环节为nil，它不会抛出错误，而是会把nil返回给引用者。这种由问号（?）引用可选类型的方式就是**可选链**。

可选链是一种“温柔”的引用方式，它的引用目标不仅仅是属性，还可以是方法、下标和嵌套类型等。

下面我们看一个具有嵌套类型的示例：

```
class Employee {

    var no : Int = 0
    var name : String = ""
```

```
    var job : String = ""
    var salary : Double = 0
    var dept : Department?                          ①

    struct Department {                             ②
        var no : Int = 10
        var name : String = "SALES"
    }
}

var emp = Employee()

println(emp.dept?.name)                             ③
```

上述代码第①行定义可选类型Department?的属性dept，Department是嵌套结构体类型。在第③行采用可选链方式引用。

输出结果为nil，这是因为emp.dept环节为nil。如果把第①行代码修改一下：

```
var dept : Department? = Department()
```

则输出结果为SALES。这说明可选链可以到达目标name。

至于如何为其他类型加可选链，我们会在后面的学习过程中逐个介绍。

本节在介绍可选类型和可选链时，多次使用了问号（？）和感叹号（！），但是它们的含义是不同的，下面我们详细说明一下。

1. **可选类型中的问号（？）**

声明这个类型是可选类型，访问这种类型的变量或常量时要使用感叹号（！），下列代码是强制拆封：

```
let result1 : Double? = divide(100, 200)
//声明可选类型
println(result1!)
//强制拆封取值
```

2. **可选类型中的感叹号（！）**

声明这个类型也是可选类型，但是访问这种类型的变量或常量时可以不使用感叹号（！），下列代码是隐式拆封：

```
let result3 : Double? = divide(100, 200)
//声明可选类型
println(result3)
```

```
//隐式拆封取值
```

3. **可选链中的感叹号（！）**

多个对象具有关联关系，当从一个对象引用另外对象的方式、属性和下标等成员时就会形成引用链，由于这个“链条”某些环节可能有值，也可能没有值，因此需要采用如下方式访问：

```
emp.dept!.comp!.name
```

4. **可选链中的问号（？）**

在可选链中使用感叹号（！）访问时，一旦“链条”某些环节没有值，程序就会发生异常，于是我们把感叹号（！）改为问号（？），代码如下所示：

```
emp.dept?.comp?.name
```

这样某些环节没有值的时候返回`nil`，程序不会发生异常。

11.7 访问限定

作为一种面向对象的语言封装性是不可缺少的，Swift语言在正式版中增加了访问控制，这样一来Swift语言就可以实现封装特性了。由于在Swift语言中类、结构体和枚举类型都具有面向对象的特性，因此Swift语言的封装就变得比较复杂了。

11.7.1 访问范围

首先，我们需要搞清楚访问范围的界定。访问范围主要有两个：模块和源文件。

模块是指一个应用程序包或一个框架。在 Swift 中，可以用`import`关键字将模块引入到自己的工程中。应用程序包是可执行的，其内部包含了很多Swift文件以及其他文件，应用程序包可以通过Xcode的如图11-3和图11-4所示模板创建。框架也是很多Swift文件及其他文件的集合，但是应用程序包不同的是，它编译的结果是不可以执行文件，框架可以通过Xcode的如图11-5所示的Cocoa Touch Framework模板，以及图11-6所示的Cocoa Framework模板创建。

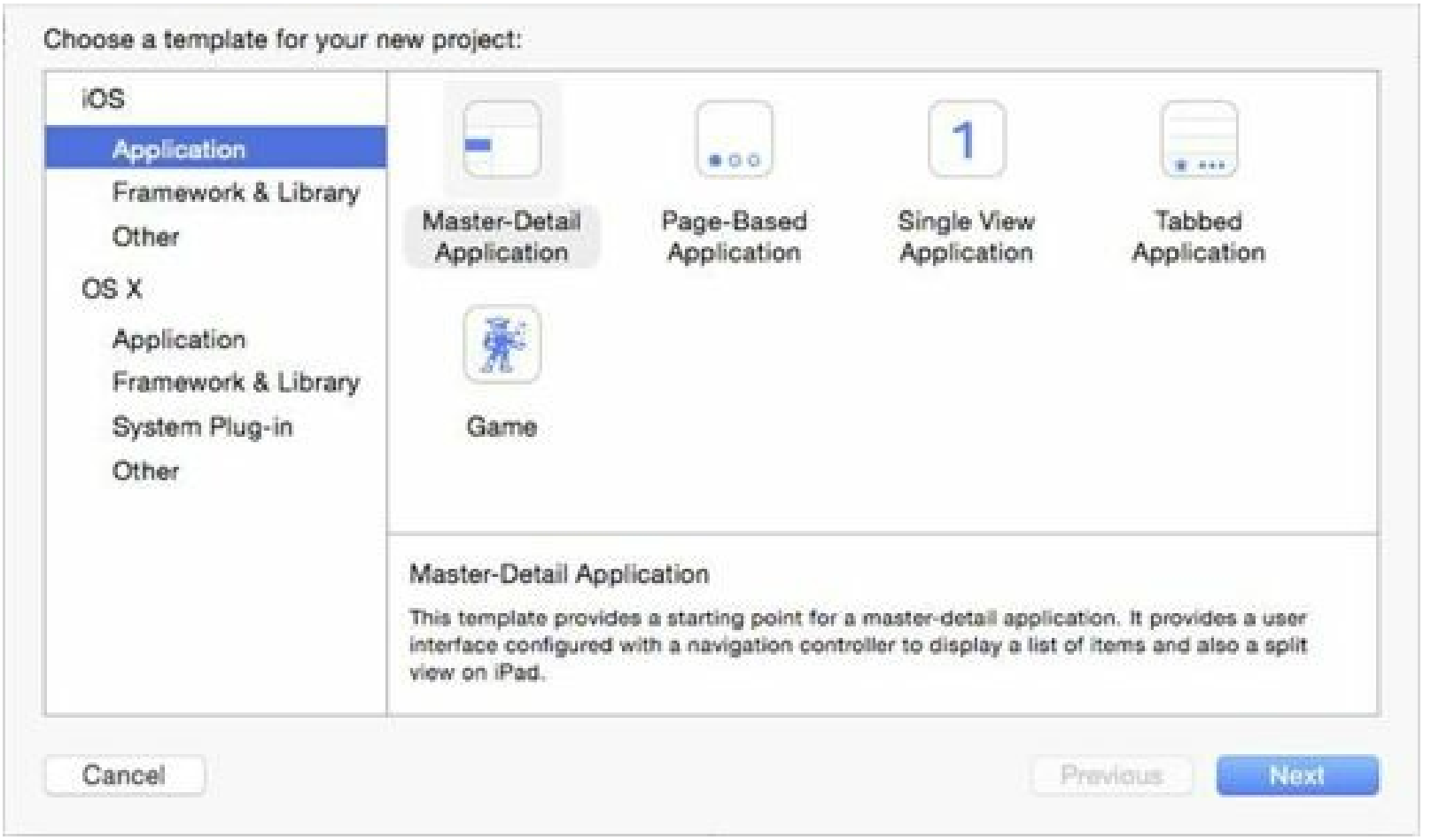


图 11-3　iOS应用程序工程模板

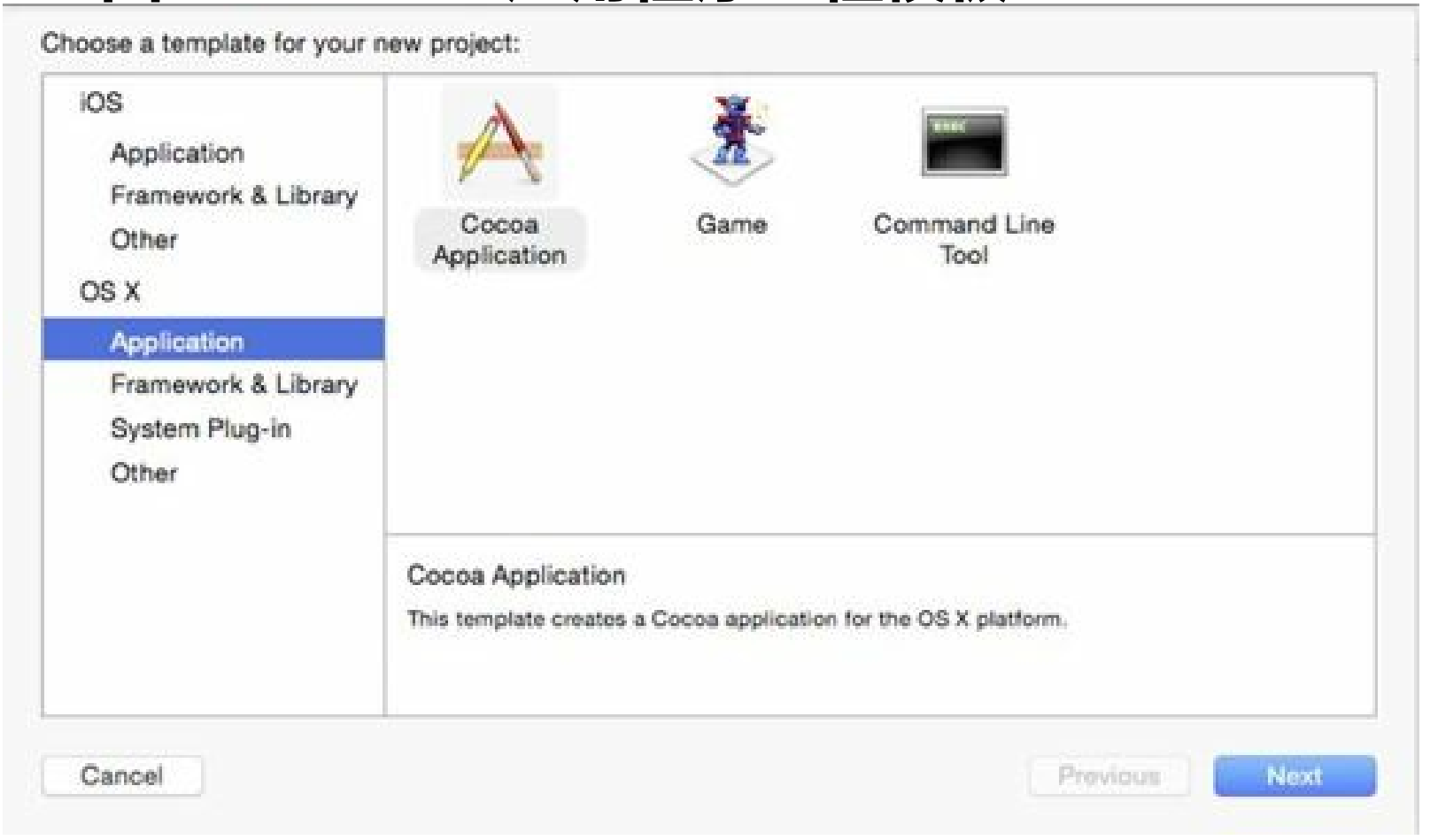


图 11-4　Mac OS X应用程序工程模板

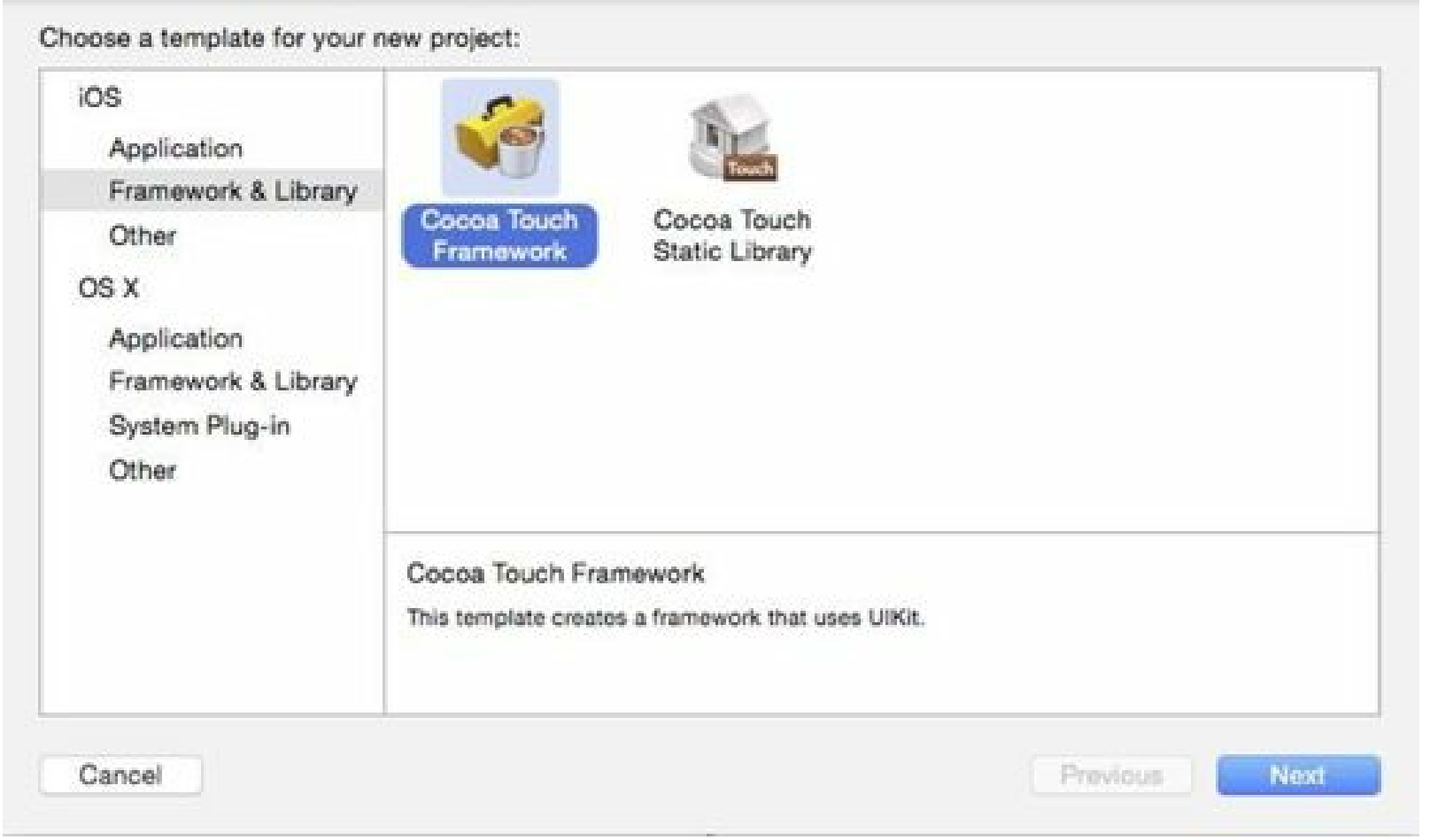


图 11-5　iOS框架和库模板

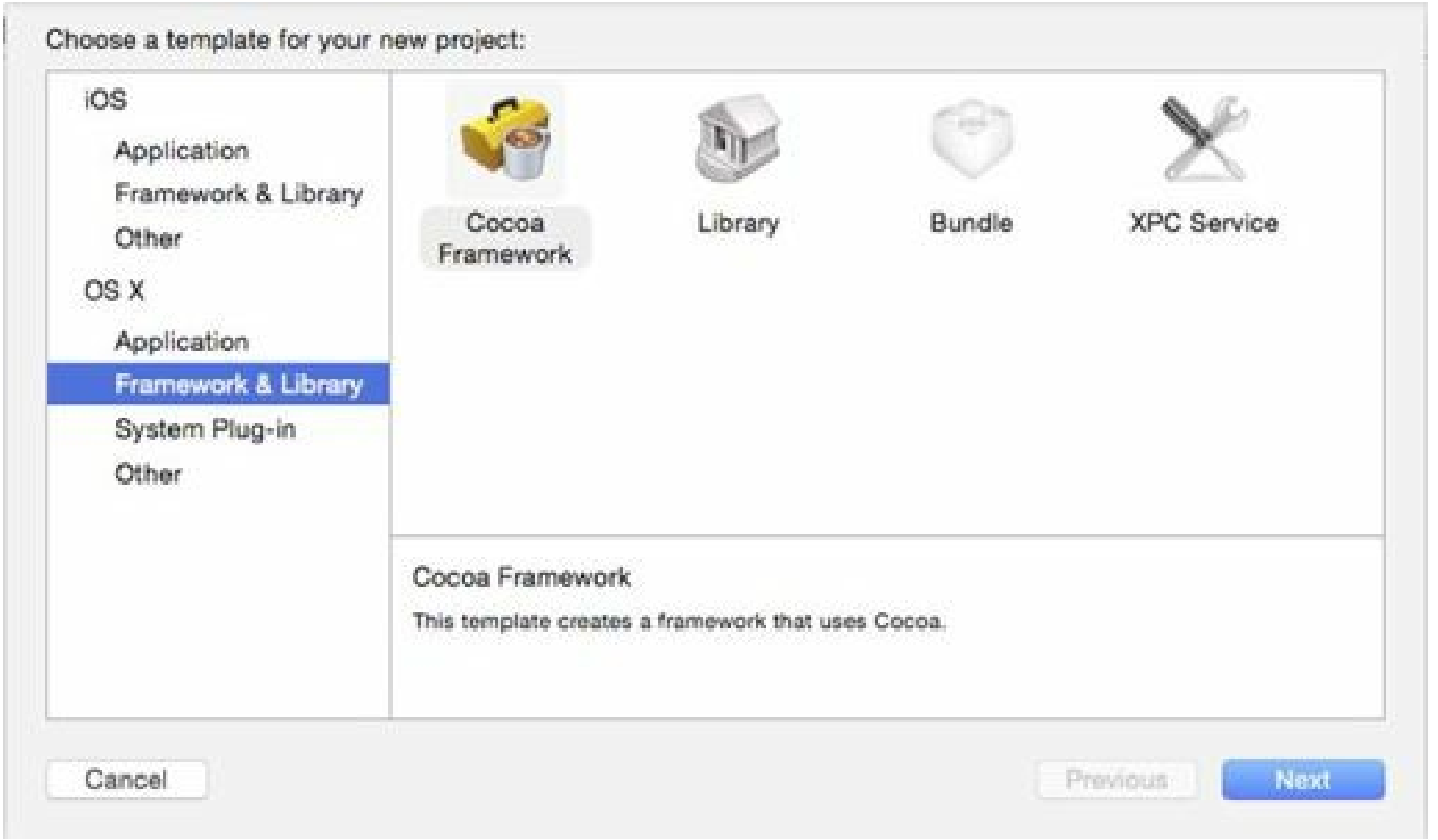


图 11-6　Mac OS X框架和库模板

源文件指的是Swift中的.swift文件，编译之后它被包含在应用程序包或框架中， 通常一个源文件包含一个面向对象类型（类、结构体和枚举），在这些类型中又包含函数、属性等。

11.7.2　访问级别

Swift提供了3种不同访问级别，对应的访问修饰符为：`public`、`internal`和`private`。这些访问修饰符可以修饰类、结构体、枚举等面向对象的类型，还可以修饰变量、常量、下标、元组、函数、属性等内容。

提示 为了便于描述，我们把类、结构体、枚举、变量、常量、下标、元组、函数、属性等内容统一称为“实体”。

- public。可以访问自己模块中的任何public实体。如果使用import语句引入其他模块，我们可以访问其他模块中的public实体。
- internal。只能访问自己模块的任何internal实体，不能访问其他模块中的internal实体。internal可以省略，换句话说，默认访问限定是internal。
- private。只能在当前源文件中使用的实体，称为私有实体。使用private修饰，可以用作隐藏某些功能的实现细节。

使用访问修饰符的示例代码如下：

```
public class PublicClass {}
internal class InternalClass {}
private class PrivateClass {}

public var intPublicVariable = 0
let intInternalConstant = 0  // internal访问级别
```

```
private func intPrivateFunction() {}
```

11.7.3 使用访问级别最佳实践

由于中Swift中访问限定符能够修饰的实体很多，使用起来比较繁琐，下面我们给出一些最佳实践。

1. **统一性原则**

- **原则1**：如果一个类型（类、结构体、枚举）定义为`internal`或`private`，那么类型声明的变量或常量不能使用`public`访问级别。因为`public`的变量或常量可以被任何人访问，而`internal`或`private`的类型不可以。
- **原则2**：函数的访问级别不能高于它的参数和返回类型的访问级别。假设函数声明为`public`级别，而参数或者返回类型声明为`internal`或`private`，就会出现函数可以被任何人访问，而它的参数和返回类型不可以访问的矛盾情况。

我们看看下面的代码：

```
private class Employee {
①
    var no : Int = 0
    var name : String = ""
    var job : String?
    var salary : Double = 0
    var dept : Department?
}

internal struct Department {
②
    var no : Int = 0
    var name : String = ""
}

public let emp = Employee()          //编译错误
③

public var dept = Department()       //编译错误
④
```

上述代码第①行定义了private级别的类Employee，所以当第③行代码创建并声明emp常量时，会发生编译错误。代码第②行定义了internal的结构体Department，所以的当第④行代码创建并声明dept变量时，会发生编译错

误。

我们再看一个使用函数的示例代码：

```
class Employee {
    var no : Int = 0
    var name : String = ""
    var job : String?
    var salary : Double = 0
    var dept : Department?
}

struct Department {
    var no : Int = 0
    var name : String = ""
}

public func getEmpDept(emp : Employee)->
Department? {                ①
    return emp.dept
}
```

上述代码第①行会发生如下编译错误。

```
<EXPR>:22:13:error: function cannot be declared
public because its parameter uses an internal
type
public func getEmpDept(emp : Employee)->
```

```
Department? {
            ^                        ~~~~~~~~
<EXPR>:9:7: note: type declared here
class Employee {
      ^
```

这个错误说明了getEmpDept函数中的Employee类型访问级别（internal）与函数的访问级别（public）不一致。

如果我们修改上述代码如下：

```
public class Employee {
    var no : Int = 0
    var name : String = ""
    var job : String?
    var salary : Double = 0
    var dept : Department?
}

struct Department {
    var no : Int = 0
    var name : String = ""
}

public func getEmpDept(emp : Employee)->
Department? {       ①
```

```
    return emp.dept
}
```

修改后代码第①行还会发生如下编译错误。

```
<EXPR>:22:13: error: function cannot be
declared public because its result uses an
internal type
public func getEmpDept(emp : Employee)->
Department? {
            ^
~~~~~~~~~~
<EXPR>:17:8: note: type declared here
struct Department {
       ^
```

这个错误说明了getEmpDept函数中的Department类型访问级别（internal）与函数的访问级别（public）不一致。

2. **设计原则**

如果我们编写的是应用程序，应用程序包中的所有Swift文件和其中定义的实体，都是供本应用使用的，而不是提供其他模块使用，那么我们就不用设置访问级别了，即使用默认的访问级别。

如果我们开发的是框架，框架编译的文件不能独立运行，因此它天生就是给别人使用的，这种情况下我们要详细设计其中的Swift文件和实体的访问级别，让别人使用的可以设定为public，不想让别人看到的可以设定为internal或private。

3. **元组类型的访问级别**

元组类型的访问级别遵循元组中字段最低级的访问级别，例如下面的代码：

```
private class Employee {
    var no : Int = 0
    var name : String = ""
    var job : String?
    var salary : Double = 0
    var dept : Department?
}

struct Department {
    var no : Int = 0
    var name : String = ""
}

private let emp = Employee()
var dept = Department()

private var student1 = (dept, emp)
①
```

上述代码第①行定义了元组student1，其中的字段dept和emp的最低访问级别是private，所以student1访问级别也是pirvate，这也符合统一性原则。

4. **枚举类型的访问级别**

枚举中成员的访问级别继承自该枚举，因此我们不能为枚举中的成员指定访问级别。示例代码如下：

```
public enum WeekDays {
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
}
```

由于WeekDays枚举类型是public访问级别，因而它的成员也是public级别。

11.8　本章小结

通过对本章内容的学习，我们了解了现代计算机语言中面向对象的基本特性，以及Swift语言中面向对象的基本特性；掌握了枚举、结构体和类等基本概念及其定义。

此外，还了解了Swift面向对象类型嵌套、可选类型和可选链等基本概念。

11.9 同步练习

1. 在Swift中具有面向对象特征的数据类型有（　　）。

A. 枚举　　B. 元组　　C. 结构体　　D. 类

2. 判断正误：在Swift中，类具有面向对象的基本特征，即封装性、继承性和多态性。

3. 判断正误：Swift中的枚举、类和结构体都具有继承性。

4. 有下列枚举类型代码：

```
enum ProductCategory  {
    case Washers , Dryers, Toasters
}
var product = ProductCategory.Toasters
```

枚举类型能够与switch语句结合使用，下列使用switch语句不正确的是（　　）。

A.

```
switch product {
case .Washers:
    println("洗衣机")
case .Dryers :
```

```
    println("烘干机")
default:
    println("烤箱")
}
```

B.

```
switch product {
case .Washers:
    println("洗衣机")
case .Dryers :
    println("烘干机")
case .Toasters :
    println("烤箱")
}
```

C.

```
switch product {
case .Washers:
    println("洗衣机")
case .Dryers :
    println("烘干机")
}
```

D.

```
switch product {
case .Washers:
    println("洗衣机")
default:
    println("烤箱")
}
```

5. 有下列枚举类型代码：

```
enum ProductCategory : String {
    case Washers = "washers", Dryers =
"dryers", Toasters = "toasters"
}
```

下列代码中能够成功输出"烤箱"的是（　　）。

A.

```
if (product.toRaw() == "toasters") {
    println("烤箱")
}
```

B.

```
if (product.toRaw() == .Toasters) {
    println("烤箱")
}
```

C.

```
if (product == .Toasters) {
    println("烤箱")
}
```

D.

```
if (product == "toasters") {
    println("烤箱")
}
```

6. 下列代码是在C语言中定义了联合类型的示例。

```
typedef union{
```

```
    char c;
    int a;
    double b;
} Number;
```

请把它改造成为Swift代码。

7. 判断正误：Swift中枚举是值类型，而类和结构体是引用类型。

8. 判断正误：Swift中结构体有属性、方法、下标、构造器和析构器。

9. 判断正误：由于具有面对对象的特征，所以枚举、类和结构体都可以使用恒等号===进行比较。

10. 下列有关类型嵌套正确的是（　　）。

A.

```
class a {
    class b {
    }
     enum c {
        case c(Character)
    }
    struct d {
    }
}
```

B.

```
enum Number {
    case c(Character)
    case a(Int)
    case b(Double)

    class d {
    }

    struct e {
    }
}
```

C.

```
struct c {
    class b {
    }
}
```

D.

```
struct c1 {
    class b {
        class a {
        }
    }
}
```

11. 运行下列代码的输出结果是（　　）。

```
var cod : String? = "a fish"
var dab : String? = cod

println("cod == \(cod)")
cod = nil
println("cod == \(cod)")
println("dab == \(dab)")
```

A.

```
cod == a fish
cod == nil
dab == nil
```

B.

```
cod == nil
cod == nil
dab == a fish
```

C.

```
cod == a fish
cod == nil
dab == a fish
```

D.

```
cod == nil
cod == nil
dab == nil
```

12. 下列语句能够正确执行的是（　　）。
A.

```
var optionalCod: String
```

```
if optionalCod
{
    println("uppercase optionalCod == \
(optionalCod.uppercaseString)")
}
else
{
    println("optionalCod is nil")
}
```

B.

```
var optionalCod: String?
if optionalCod
{
    println("uppercase optionalCod == \
(optionalCod.uppercaseString)")
}
else
{
    println("optionalCod is nil")
}
```

C.

```
var optionalCod: String?
```

```
if optionalCod
{
    println("uppercase optionalCod == \
(optionalCod!.uppercaseString)")
}
else
{
    println("optionalCod is nil")
}
```

D.

```
var optionalCod: String!
if optionalCod
{
    println("uppercase optionalCod == \
(optionalCod.uppercaseString)")
}
else
{
    println("optionalCod is nil")
}
```

13. 若有以下多个有关联关系类的定义：

```
class Person {
```

```
    var residence: Residence?
}

class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
    return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        println("The number of rooms is \
(numberOfRooms)")
    }
    var address: Address?
}

class Room {
    let name: String
    init(name: String) { self.name = name }
}

class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if buildingName {
            return buildingName
```

```
        } else if buildingNumber {
            return buildingNumber
        } else {
            return nil
        }
    }
}
```

以下是类的访问代码，执行如下代码，说法正确的是（　　）。

```
let john = Person()
let johnsStreet1 =
john.residence?.address?.street          ①
let johnsStreet2 =
john.residence!.address!.street          ②
```

A. 程序有编译错误
B. 没有编译错误，但有运行错误
C. 代码第①行能够执行
D. 代码第②行不能执行

第 12 章　属性与下标

在面向对象分析与设计方法学（OOAD）中，类是由属性和方法组成的，属性一般是访问数据成员。在Objective-C中，属性是为了访问封装后的数据成员（成员变量）而设计的，属性本身并不存储数据，数据是由数据成员存储的。而Swift中的属性分为存储属性和计算属性，存储属性就是Objective-C中的数据成员，计算属性不存储数据，但可以通过计算其他属性返回数据。

对于集合类型中的元素，还可以通过下标访问。下标在Java语言中称为索引属性，Swift的下标也具有属性特性，因此本章后面将会介绍下标的使用。

12.1　存储属性

存储属性可以存储数据，分为**常量属性**（用关键字let定义）和**变量属性**（用关键字var定义）。存储属性适用于类和结构体两种Swift面向对象类型。

12.1.1　存储属性概念

我们在前面的章节中曾用到过属性，例如11.4节的员工类（Employee）和部门类（Department）。它们的类图如图12-1所示，Employee 的部门属性dept与Department之间进行了关联。

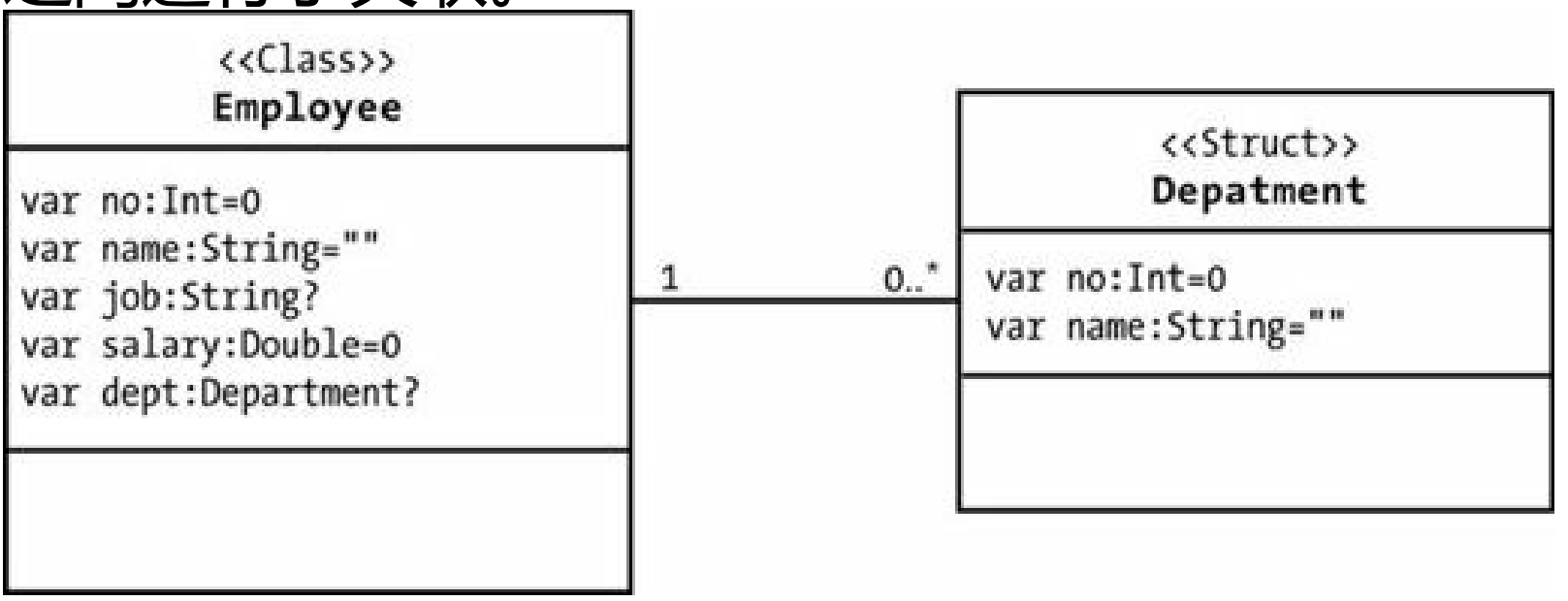


图 12-1　类图

我们可以在定义存储属性时指定默认值，示例代码如下：

```
class Employee {
    let no : Int = 0
    var name : String = ""
    var job : String?
    var salary : Double = 0
    var dept : Department?
}

struct Department {
    let no : Int = 0
    var name : String = ""
}

var emp = Employee()
emp.no = 100                    //编译错误
①

let dept = Department()
dept.name = 30                  //编译错误
②

let emp1 = Employee()
emp1.name =  "Tony"
③
```

实例通过点（.）运算符调用属性，代码第①行试图修改常量属性，程序会发生编译错误。第②行

代码也会发生编译错误，因为实例dept本身是常量，即便它的属性name是变量属性，也不能修改。但是代码第③行emp1.name = "Tony"却可以编译通过，emp1实例也是常量，name是变量属性。这是因为emp1是类实例，是引用类型，dept是结构体实例，是值类型。引用类型相当于指针，其常量也可以修改，但值类型的常量是不能修改的。

12.1.2 延迟存储属性

由于Employee和Department有关联关系，Employee类中的dept属性关联到了Department结构体。这种关联关系体现为：一个员工必然隶属于一个部门，而一个部门有很多员工。一个员工实例对应于一个部门实例。

下面看以下代码实现：

```
class Employee {
    var no : Int = 0
    var name : String = ""
    var job : String?
    var salary : Double = 0
    var dept : Department = Department()
```

```
①
}

struct Department {
    let no : Int = 0
    var name : String = ""
}

let emp = Employee()
②
```

在代码第②行创建Employee实例的时候，也会同时在第①行实例化dept（部门）属性。然而程序或许不关心他隶属于哪个部门，只关心他的no（编号）和name（姓名）。虽然不使用dept实例，但是仍然会占用内存。在Java中，有一种数据持久化技术叫Hibernate[1]。为了应对这种情况，Hibernate有一种延时加载技术，Swift也采用了延迟加载技术。修改代码如下：

[1]Hibernate是一种Java语言下的对象关系映射解决方案。它是使用GNU宽通用公共许可证发行的自由、开源的软件。它为面向对象的领域模型到传统的关系型数据库的映射，提供了一个使用方便的框架。——引自于维基百科 http://zh.wikipedia.org/wiki/Hibernate

```
class Employee {
    var no : Int = 0
    var name : String = ""
    var job : String?
    var salary : Double = 0
    lazy var dept : Department = Department()
①
}
struct Department {
    let no : Int = 0
    var name : String = ""
}

let emp = Employee()
②
```

我们在dept属性前面添加了关键字lazy声明，这样dept属性就是延时加载。**延时加载**，顾名思义，就是dept属性只有在第一次访问它的时候才加载，如果永远不访问，它就不会创建，这样就可以减少内存占用。

12.1.3 属性观察者

为了监听属性的变化，可以使用Swift提供的以下属性观察者。

- `willSet`在设置新的值之前调用。
- `didSet`在新值被设置之后马上调用。

我们可以根据需要，使用部分或全部的属性观察者。它们不能应用于延迟存储属性，但能够应用于一般的存储属性和计算属性。计算属性所需的属性观察者将在第15章介绍。

12.2 计算属性

计算属性本身不存储数据，而是从其他存储属性中计算得到数据。与存储属性不同，类、结构体和枚举都可以定义计算属性。

12.2.1 计算属性概念

计算属性提供了一个getter（取值访问器）来获取值，以及一个可选的setter（设置访问器）来间接设置其他属性或变量的值。计算属性的语法格式如下：

```
面向对象类型 类型名 {
①
    存储属性
②
    ......
    var 计算属性名 : 属性数据类型 {
③
    get {
④
        return 计算后属性值
⑤
    }
⑥
    set (新属性值) {
⑦
```

```
            ......
        }
⑧
        }
⑨
}
```

计算属性的语法格式比较混乱，这里我们解释一下。第①行的“面向对象类型”包括类、结构体和枚举3种。第②行的“存储属性”表示有很多存储属性。

事实上，第③~⑨行代码才是定义计算属性，变量必须采用var声明。第④~⑥行代码是getter访问器，它其实是一个方法，在访问器中对属性进行计算。最后在第⑤行代码必须使用return语句将计算结果返回。

第⑦~⑧行代码是setter访问器，其中第⑦行代码中，“新属性值”是要赋值给属性值。

定义计算属性比较麻烦，要注意后面的几个大括号的对齐关系。

我们先看一个示例：

```
import UIKit
①
```

```
class Employee {
    var no : Int = 0
    var firstName : String = "Tony"
②
    var lastName : String = "Guan"
③
    var job : String?
    var salary : Double = 0
    lazy var dept : Department = Department()

    var fullName : String {
④
    get {
        return firstName + "." + lastName
⑤
    }
    set (newFullName) {
⑥
        var name =
newFullName.componentsSeparatedByString(".")  ⑦
        firstName = name[0]
        lastName = name[1]
    }
    }
}

struct Department {
    let no : Int = 0
    var name : String = ""
```

```
}

var emp = Employee()
println(emp.fullName)
⑧

emp.fullName = "Tom.Guan"
⑨
println(emp.fullName)
⑩
```

上述代码第①行是引入UIKit框架，第②行代码是定义员工的firstName存储属性，第③行代码是定义员工的lastName存储属性。为了获得fullName（全名），fullName = firstName . lastName，不需要定义一个fullName存储属性，而是可以通过firstName和lastName拼接（计算）而成。

第④行代码直接定义fullName计算属性。第⑤行是返回拼接的结果。第⑥行代码中的newFullName是要存储传递进来的参数值，set(newFullName)可以省略如下形式，使用Swift默认名称newValue替换newFullName。

```
set {
      var name =
newValue.componentsSeparatedByString(".")
      firstName = name[0]
      lastName = name[1]
}
```

代码第⑦行是使用String的字符串分割方法componentsSeparatedByString，指定逗号为字符串的分割符号，分割方法返回的是String数组。

第⑧行代码println(emp.fullName)是调用属性的getter访问器，取出属性值。第⑨行代码emp.fullName = "Tom.Guan"是调用属性的setter访问器，给属性赋值。

12.2.2 只读计算属性

计算属性可以只有getter访问器，没有setter访问器，这就是**只读计算属性**。指定计算属性不仅不用写setter访问器，而且get{}代码也可以省略。与上一节相比，代码将大大减少。修改上一节示例为只读计算属性，代码如下：

```
class Employee {
    var no : Int = 0
    var firstName : String = "Tony"
    var lastName : String = "Guan"
    var job : String?
    var salary : Double = 0
    lazy var dept : Department = Department()

    var fullName : String {
①
        return firstName + "." + lastName
    }
}

struct Department {
    let no : Int = 0
    var name : String = ""
}

var emp = Employee()
println(emp.fullName)
```

只读计算属性经过简化后，第①行代码是更加简洁的setter访问器。只读计算属性不能够赋值，下列语句是错误的。

```
emp.fullName = "Tom.Guan"
```

12.2.3 结构体和枚举中的计算属性

前面介绍的示例都是类的计算属性，本节将介绍一些结构体和枚举中的计算属性，从而比较它们的差异。

示例代码如下：

```
struct Department {
①
    let no : Int = 0
    var name : String = "SALES"

    var fullName : String {
②
        return "Swift." + name + ".D"
    }
}

var dept = Department()
println(dept.fullName)
③

enum WeekDays  : String {
④
    case Monday      =    "Mon."
    case Tuesday     =    "Tue."
```

```
    case Wednesday  =    "Wed."
    case Thursday   =    "Thu."
    case Friday     =    "Fri."

    var message : String {
⑤
        return "Today is " + self.toRaw()
⑥
    }
}

var day = WeekDays.Monday
println(day.message)
⑦
```

上述代码第①行定义了结构体Department，第②行定义了只读属性fullName，第③行是读取fullName属性。

第④行定义了枚举类型WeekDays，第⑤行定义了只读属性message，第⑥行代码中使用了self.toRaw()语句将当前实例值转换为原始值，其中self代表当前实例，toRaw()方法是转换为原始值，否则不能进行字符串拼接。代码第⑦行是读取message属性。

12.3 属性观察者

为了监听属性的变化，Swift提供了属性观察者。属性观察者能够监听存储属性的变化，即便变化前后的值相同，它们也能监听到。但它们不能监听延迟存储属性和常量存储属性的变化。

Swift中的属性观察者主要有以下两个。

- willSet：观察者在修改之前调用。
- didSet：观察者在修改之后立刻调用。

属性观察者的语法格式如下：

```
面向对象类型 类型名 {
①
    ......
    var 存储属性 : 属性数据类型 = 初始化值 {          ②
    willSet(新值) {
③
        ......
    }
④
    didSet(旧值) {
⑤
        ......
    }
⑥
```

```
    }
⑦
}
```

属性观察者的语法格式比计算属性还要混乱，下面我们解释一下。第①行的“面向对象类型”包括类和结构体，不包括枚举，因为枚举不支持存储属性。

代码第②~⑥行是定义存储属性。第②行的“存储属性”是我们定义的存储属性名。

代码第③~④行是定义willSet观察者。第③行代码中的“新值”是传递给willSet观察者的参数，它保存了将要替换原来属性的新值。参数的声明可以省略，系统会分配一个默认的参数newValue。

代码第⑤~⑥行是定义didSet观察者。第⑤行代码中的“旧值”是传递给didSet观察者的参数，它保存了被新属性替换的旧值。参数的声明也可以省略，系统会分配一个默认的参数oldValue。

示例代码如下：

```
class Employee {
```

```
①
    var no : Int = 0
    var name : String = "Tony" {
②
    willSet(newNameValue) {
③
        println("员工name新值：\(newNameValue)")
④
    }
    didSet(oldNameValue) {
⑤
        println("员工name旧值：\(oldNameValue)")
⑥
    }
    }
    var job : String?
    var salary : Double = 0
    var dept : Department?
}

struct Department {
⑦
    var no : Int = 10 {
⑧
    willSet {
⑨
        println("部门编号新值：\(newValue)")
⑩
    }
    didSet {
```

```
⑪
        println("部门编号旧值：\(oldValue)")
⑫
    }
    }
    var name : String = "RESEARCH"
}

var emp = Employee()
emp.no = 100
emp.name = "Smith"
⑬

var dept = Department()
dept.no = 30
⑭
```

上述代码第①行定义了Employee类，第②行是定义name属性，第③行是定义name属性的willSet观察者，newNameValue是由我们分配的传递新值的参数名，第④行是willSet观察者内部处理代码，其中使用了参数newNameValue。第⑤行是定义name属性的didSet观察者，oldNameValue是由我们分配的传递旧值的参数名，第⑥行是didSet观察者内部处理代码，其

中使用了参数oldNameValue。

第⑦行定义了Department类，第⑧行是定义no属性，第⑨行是定义no属性的willSet观察者，注意这里没有声明参数，但是我们可以在观察者内部使用newValue，见代码第⑩行，newValue是由系统分配的参数名。第⑪行是定义no属性的didSet观察者，注意这里也没有声明参数，但是我们可以在观察者内部使用oldValue，见代码第⑫行，oldValue是由系统分配的参数名。

上述代码运行结果如下：

```
员工name新值：Smith
员工name旧值：Tony
部门编号新值：30
部门编号旧值：10
```

对于这两个属性观察者，我们可以根据自己的需要来使用。它们常常应用于后台处理，以及需要更新界面的业务需求。

12.4 静态属性

在介绍静态属性之前，我们先来看一个类的设计，有一个Account（银行账户）类，假设它有3个属性：amount（账户金额）、interestRate（利率）和owner（账户名）。在这3个属性中，amount和owner会因人而异，不同的账户这些内容是不同的，而所有账户的interestRate都是相同的。

amount和owner属性与账户个体有关，称为**实例属性**。interestRate属性与个体无关，或者说是所有账户个体共享的，这种属性称为**静态属性**或**类型属性**。

3种面向对象类型（结构体、枚举和类）都可以定义静态属性，它们的语法格式分别如下所示：

```
struct 结构体名 {
①
    static var(或let) 存储属性   = "xxx"
②
    ......
    static var 计算属性名 : 属性数据类型 {
③
    get {
        return 计算后属性值
```

```
    }
    set (新属性值) {
        ......
    }
    }
}

enum 枚举名 {
④
    static var(或let) 存储属性   = "xxx"
⑤
    ......
    static var 计算属性名 : 属性数据类型 {
⑥
    get {
        return 计算后属性值
    }
    set (新属性值) {
        ......
    }
    }
}

class 类名 {
⑦
    ......
    class var 计算属性名 : 属性数据类型 {
⑧
    get {
        return 计算后属性值
```

```
    }
    set (新属性值) {
        ......
    }
    }
}
```

上述代码中，第①行是定义结构体，结构体中可以定义静态存储属性和计算属性。第②代码是定义静态存储属性，声明关键字是static，这个属性可以是变量属性，也可以是常量属性。第③行代码是定义静态计算属性，声明使用的关键字是static，计算属性不能为常量，这里只能是变量。结构体静态计算属性也可以是只读的，语法如下：

```
static var 计算属性名 : 属性数据类型 {
    return 计算后属性值
}
```

第④行是定义枚举，枚举中不可以定义实例存储属性，但可以定义静态存储属性，也可以定义静态计算属性。定义枚举静态属性与定义结构体静态

属性的语法完全一样，这里就不再赘述了。

第⑦行是定义类，类中可以定义实例存储属性，但不可以定义静态存储属性。类中可以定义静态计算属性。声明使用的关键字是`class`，这与结构体和枚举的声明不同。

我们对上述说明进行了归纳，见表12-1。

表12-1　面向对象类型属性

面向对象类型	实例存储属性	静态存储属性	实例计算属性	静态计算属性
类	支持	不支持	支持	支持
结构体	支持	支持	支持	支持
枚举	不支持	支持	支持	支持

提示　在静态计算属性中不能访问实例属性（包括存储属性和计算属性），但可以访问其他静态属性。在实例计算属性中能访问实例属性，也能访问静态属性。

12.4.1　结构体静态属性

下面我们先看一个Account结构体静态属性示例：

```
struct Account {

    var amount : Double = 0.0
//账户金额
    var owner : String = ""
//账户名

    static var interestRate : Double = 0.668
//利率          ①

    static var staticProp : Double {
②
        return interestRate  1_000_000
    }

    var instanceProp : Double {
③
        return Account.interestRate  amount
    }
}

//访问静态属性
println(Account.staticProp)
④
```

```
var myAccount = Account()
//访问实例属性
myAccount.amount = 1_000_000
⑤
//访问静态属性
println(myAccount.instanceProp)
⑥
```

上述代码定义了Account结构体，其中第①行代码定义了静态存储属性interestRate，第②行代码定义了静态计算属性staticProp，在其属性体中可以访问interestRate等静态属性。第③行代码定义了实例计算属性instanceProp，在其属性体中能访问静态属性interestRate，访问方式为“类型名.静态属性”，如Account.interestRate。第④行代码也是访问静态属性，访问方式也是“类型名.静态属性”。

第⑤行和第⑥行代码是访问实例属性，访问方式是“实例.实例属性”。

12.4.2 枚举静态属性

下面我们先看一个Account枚举静态属性示例：

```
enum Account {

    case 中国银行
①
    case 中国工商银行
    case 中国建设银行
    case 中国农业银行
②

    static var interestRate : Double = 0.668 //
利率      ③

    static var staticProp : Double {
④
        return interestRate  1_000_000
    }

    var instanceProp : Double {
⑤

        switch (self) {
⑥
            case 中国银行:
                Account.interestRate = 0.667
            case 中国工商银行:
                Account.interestRate = 0.669
            case 中国建设银行:
                Account.interestRate = 0.666
            case 中国农业银行:
```

```
                    Account.interestRate = 0.668
            }
⑦
            return Account.interestRate  1_000_000
⑧
        }
}

//访问静态属性
println(Account.staticProp)
⑨

var myAccount = Account.中国工商银行
//访问实例属性
println(myAccount.instanceProp)
⑩
```

上述代码定义了Account枚举类型，其中第①~②行代码定义了枚举的4个成员。第③行代码定义了静态存储属性interestRate，第④行代码定义了静态计算属性staticProp，在其属性体中可以访问interestRate等静态属性。第⑤行代码定义了实例计算属性instanceProp，其中第⑥~⑦行代码使用switch语句判断当前实例的值，获得不同的利息，第⑥行代码中使用了self，它指代

当前实例本身。第⑧行代码是返回计算的结果。

第⑨行代码是访问静态属性。第⑩行代码是访问实例属性。

示例运行结果如下：

```
668000.0
669000.0
```

12.4.3 类静态属性

下面我们先看一个Account类静态属性示例：

```
class Account {
①

    var amount : Double = 0.0                    //
账户金额
    var owner : String = ""                      //
账户名

    var interestRate : Double = 0.668            //
利率                ②

    class var staticProp : Double {
③
        return 0.668  1_000_000
```

```
    }

    var instanceProp : Double {
④
        return self.interestRate  self.amount
⑤
    }
}

//访问静态属性
println(Account.staticProp)
⑥

var myAccount = Account()
//访问实例属性
myAccount.amount = 1_000_000
//访问静态属性
println(myAccount.instanceProp)
⑦
```

上述代码第①行定义了Account类，第②行代码定义了存储属性interestRate，注意在类中不能定义静态存储属性。第③行代码定义了静态计算属性staticProp，关键字是class。第④行代码定义了实例计算属性instanceProp，在第⑤行代码访问实例属性interestRate和amount，访问

当前对象的实例属性可以在属性前
加“self.”，self指代当前实例本身。第⑥行代码也是访问静态属性。第⑦行代码是访问实例属性。

12.5 使用下标

还记得数组和字典吗？下面的示例代码我们曾在第8章中使用过。

```
var studentList: String[]  = ["张三","李四","王五"]
studentList[0] = "诸葛亮"

var studentDictionary = [102 : "张三",105 : "李四", 109 : "王五"]
studentDictionary[110] = "董六"
```

在访问数组和字典的时候，可以采用下标访问。其中数组的下标是整数类型索引，字典的下标是它的“键”。

12.5.1 下标概念

在Swift中，我们可以定义一些集合类型，它们可能会有一些集合类型的存储属性，这些属性中的元素可以通过下标访问。Swift中的下标相当于Java中的索引属性和C#中的索引器。

下标访问的语法格式如下：

```
面向对象类型 类型名 {
```

```
①
        其他属性
    ......
        subscript(参数: 参数数据类型) -> 返回值数据
类型 {          ②
        get {
③
            return 返回值
        }
④

        set(新属性值) {
⑤
            ......
        }
⑥
        }
⑦
}
```

上述定义中，第①行的“面向对象类型”包括类、结构体和枚举3种。第③~⑦行代码定义了下标，下标采用subscript关键字声明。下标也有类似于计算属性的getter和setter访问器。

第③~④行代码是getter访问器，getter访问器其实是一个方法，在最后使用return语句将计算

结果返回。

第⑤~⑥行代码是setter访问器，其中第⑤行代码“新属性值”是要赋值给属性值。参数的声明可以省略，系统会分配一个默认的参数newValue。

12.5.2 示例：二维数组

在Swift中没有提供二维数组，只有一维数组Array。我们可以自定义一个二维数组类型，然后通过两个下标参数访问它的元素，形式上类似于C语言的二维数组。

采用下标的二维数组示例代码如下：

```
struct DoubleDimensionalArray {
①

    let rows: Int, columns: Int
②
    var grid: [Int]

    init(rows: Int, columns: Int) {
③
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows  columns,
repeatedValue: 0)       ④
    }
```

```
    subscript(row: Int, col: Int) -> Int {
⑤

        get {
            return grid[(row  columns) + col]
⑥
        }

        set (newValue1){
            grid[(row  columns) + col] =
newValue1                    ⑦
        }
    }

}

var ary2 = DoubleDimensionalArray(rows: 10,
columns: 10)                ⑧

//初始化二维数组
for var i = 0; i < 10; i++ {
    for var j = 0; j < 10; j++ {
        ary2[i,j] = i  j
⑨

    }
}

//打印输出二维数组
```

```
for var i = 0; i < 10; i++ {
    for var j = 0; j < 10; j++ {
        print("\t \(ary2[i,j])")
    }
    print("\n")
}
```

上述代码第①行定义了二维数组结构体DoubleDimensionalArray，第②行代码声明了存储属性rows和columns，分别使用了存储二维数组的最大行数和最大列数。第③行代码是声明构造器，它使用了初始化实例。有关构造器的详细内容请参见第14章。第④行代码grid = Array(count: rows * columns, repeatedValue: 0)是初始化存储属性grid，grid是一维数组，二维数组中的数据事实上保存在grid属性中。repeatedValue参数表示数组中所有元素全部赋值为0。

第⑤行代码定义下标，其中的参数有两个。第⑥行代码是getter访问器返回，数据是从一维数组grid返回的，(row * columns) + col是它的下标表达式。第⑦行代码是setter访问器返回，把一个新值参数newValue1赋值给一维数组grid，

它的下标表达式也是(row * columns) + col。注意新值参数的声明可以省略，使用系统提供的newValue参数，setter访问器可以修改为如下形式：

```
set{
    grid[(row * columns) + col] = newValue
}
```

第⑧行代码是创建并初始化10×10大小的二维数组。访问二维数组使用第⑨行代码ary2[i,j]属性。最后输出结果如下：

```
0   0   0   0   0   0   0   0   0   0
0   1   2   3   4   5   6   7   8   9
0   2   4   6   8   10   12   14   16   18
0   3   6   9   12   15   18   21   24   27
0   4   8   12   16   20   24   28   32   36
0   5   10   15   20   25   30   35   40   45
0   6   12   18   24   30   36   42   48   54
0   7   14   21   28   35   42   49   56   63
0   8   16   24   32   40   48   56   64   72
0   9   18   27   36   45   54   63   72   81
```

这两个双循环非常影响性能。我们可以通过Playground的时间轴查看它的运行情况，如图12-2所示。

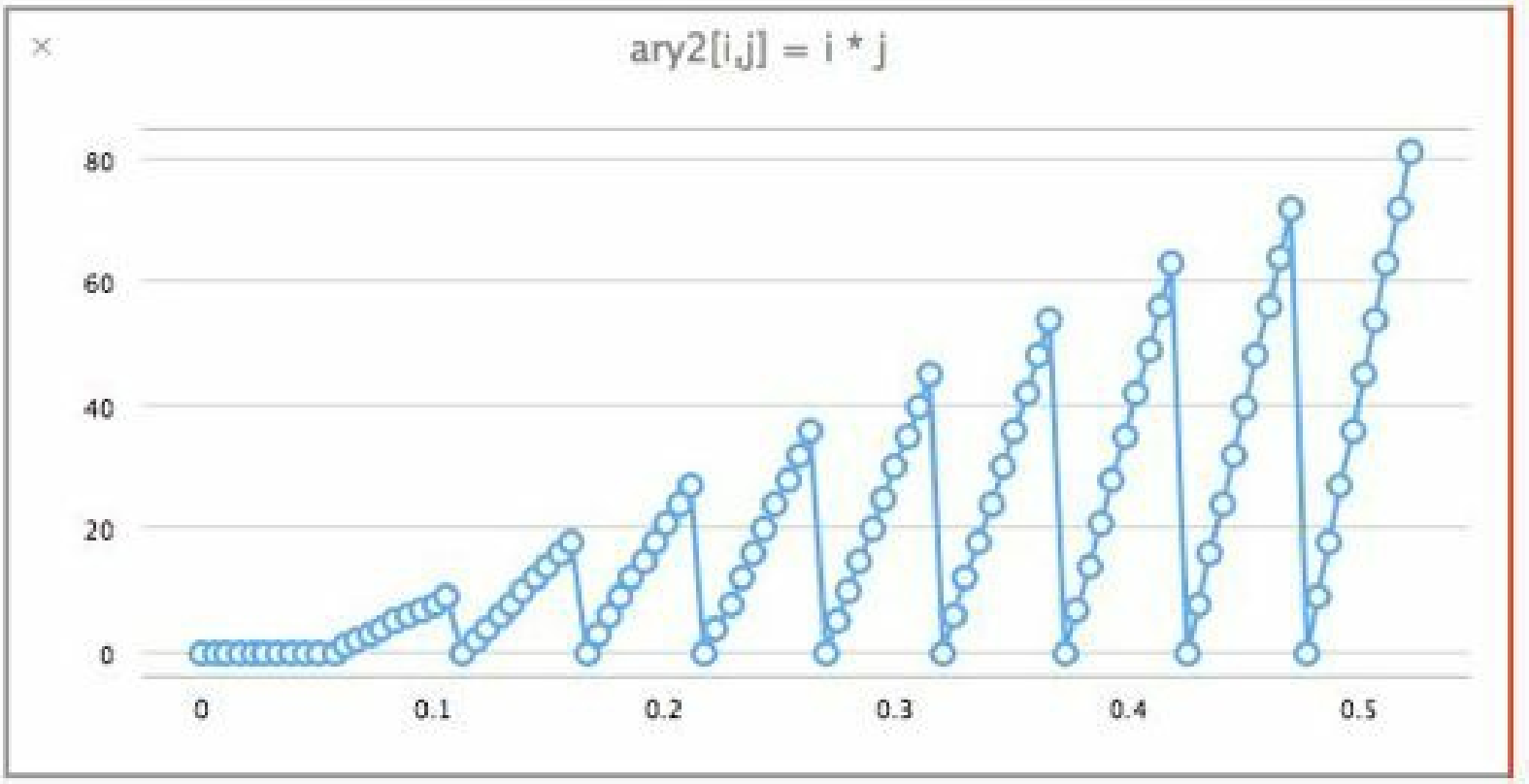


图 12-2　Playground时间轴

下标参数的个数是没有限制的，可以有多个，但是访问的时候会很麻烦，因此尽量少用。

12.6　本章小结

通过对本章内容的学习，我们了解了Swift中属性和下标的基本概念，掌握了它们的使用规律。主要是理解了存储属性、计算属性、静态属性和属性观察者等重要的属性概念以及下标的概念，掌握了下标的使用方法。

12.7 同步练习

1. 判断正误：枚举、类和结构体都支持存储属性。

2. 判断正误：枚举、类和结构体都支持属性观察者。

3. 判断正误：枚举、类和结构体都支持计算属性。

4. 判断正误：枚举、类和结构体都可以定义静态属性。

5. 判断正误：计算属性可以`var`或`let`声明。

6. 判断正误：存储属性可以`var`或`let`声明。

7. 判断正误：枚举中不可以定义实例存储属性，但可以定义静态存储属性，也可以定义静态计算属性。

8. 判断正误：定义枚举静态属性与定义结构体静态属性语法完全相同。

9. 判断正误：类中可以定义实例存储属性，但不可以定义静态存储属性。

10. 判断正误：类中可以定义静态计算属性。

11. 判断正误：在静态计算属性中能访问实例属性（包括存储属性和计算属性），也可以访问其他静态属性。

12. 判断正误：在实例计算属性中能访问实例属

性，也能访问静态属性。

13. 判断正误：实例属性访问方式是“实例.实例属性”。

14. 判断正误：静态属性访问方式“类型名.静态属性”。

15. 判断正误：类、结构体和枚举都支持下标。

16. 编程题：使用脚本编写一个电话号码本的程序。

17. 编程题：使用脚本编写一个英语字典程序。

第 13 章　方法

在面向对象分析与设计方法学（OOAD）中，类是由属性和方法组成的，方法用于完成某些操作，完成计算数据等任务。

在Swift中方法是在枚举、结构体或类中定义的函数，因此我们之前介绍的函数知识都适用于方法。方法是具有面向对象的特点，与属性类似，方法可以分为：实例方法和静态方法。

13.1 实例方法

实例方法与实例属性类似，都隶属于枚举、结构体或类的个体，即实例。通过实例化这些类型，创建实例，使用实例调用的方法。

我们上一章介绍了一个Account（银行账户）结构体，下面我们重新定义它为类，代码如下：

```
class Account {

    var amount : Double = 10_000.00
// 账户金额
    var owner : String = "Tony"
// 账户名
    //计算利息
    func interestWithRate(rate : Double) ->
Double {            ①
        return rate * amount
    }
}

var myAccount = Account()
②
//调用实例方法
println(myAccount.interestWithRate(0.88))
③
```

上述代码第①行定义了方法interestWithRate用来计算利息，从形式上看，方法与函数非常相似。第②行代码是实例化Account，myAccount是实例。第③行代码是调用方法，方法的调用前面要有主体，而函数不需要，例如myAccount.interestWithRate(0.88)是通过myAccount实例调用interestWithRate方法，调用操作符是“.”，与属性调用一样。

13.1.1 使用规范的命名

在Swift中，方法和函数的主要区别有以下3个。

- 方法的调用前面要有主体，而函数不需要。
- 方法是在枚举、结构体或类内部定义的。
- 方法命名规范与函数不同。

这一节我们主要讨论方法的命名规范问题。在很多人看来，这或许并不是一个技术问题，而是为了增强代码可读性所需要做的工作，因此很多人并

不重视命名规范。然而在Swift中，方法命名规范却不仅仅是为了增强代码的可读性，更多的是出于与Objective-C混合编程的需要。Swift要求使用规范的命名是有历史原因的，目前苹果为iOS和Mac OS X应用开发提供的开发语言是Objective-C和Swift，同一个API两种语言共存。图13-1所示是iOS中的表视图数据源协议UITableViewDataSource。从图中可见，有两种语言（Swift和Objective-C）的API。

图 13-1　API比较

我们比较同一个方法的两个不同语言的描述：

```
- (UITableViewCell *)tableView:(UITableView
)tableView
        cellForRowAtIndexPath:(NSIndexPath
)indexPath {}                          //Objective-
C


func tableView(tableView: UITableView,
        cellForRowAtIndexPath indexPath:
NSIndexPath) -> UITableViewCell {}    //Swift
```

从这两种方法的命名和参数来看，它们非常相似，按照这样的规范命名后，很多Objective-C语言程序员能够很快地转到Swift语言上来。在我看来这就是苹果的良苦用心！

下面我们就详细介绍一下这些规范。首先Swift中的方法和Objective-C中的方法应该是极其相似的。Objective-C中的方法命名遵循了SmallTalk[1]语法风格，它将一个方法名分成几个部分，称为**多重参数**。假设定义一个按照索引插入对象（或实例）到集合里的一个方法，图13-2是Objective-C中的方法定义，图13-3是Swift中的方法定义。

[1]一种面向对象的语言。

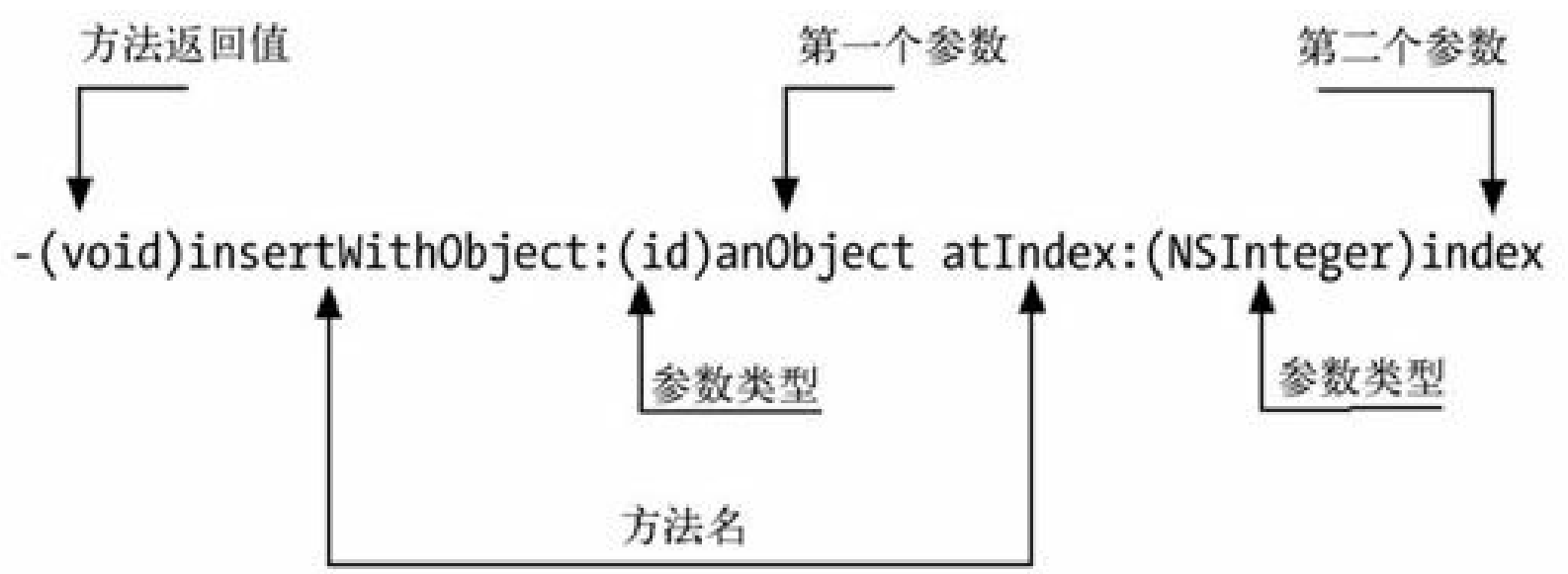


图 13-2　Objective-C方法定义

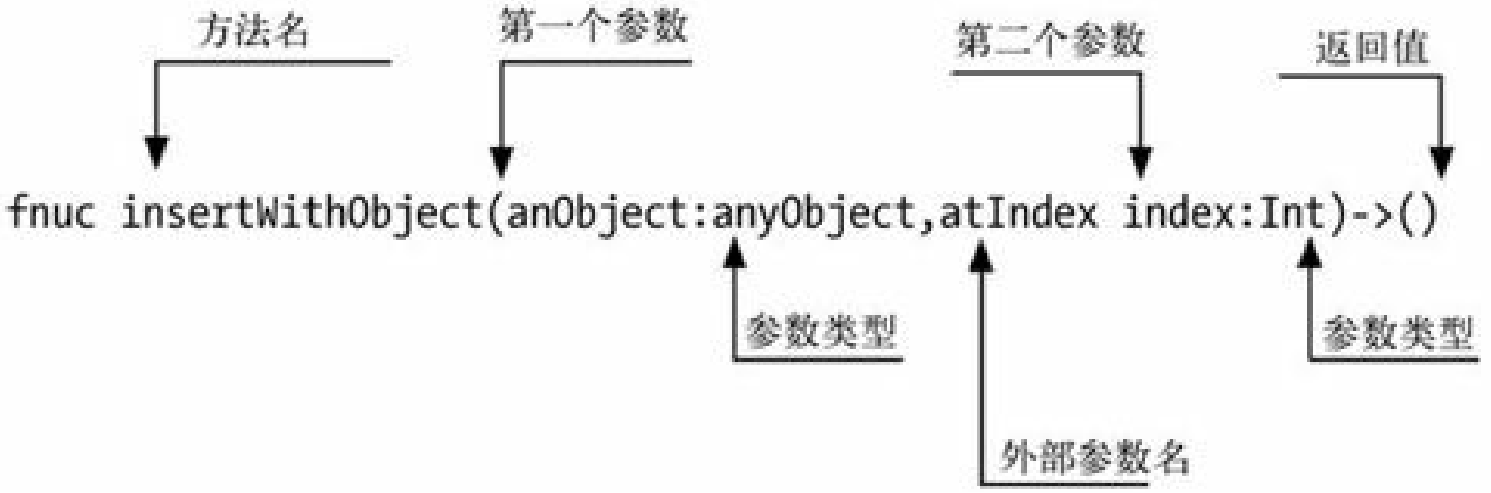


图 13-3　Swift方法定义

方法中有两个参数，第一个参数是anObject，Swift中的anyObject相当于Objective-C中的id类型。第二个参数是index，Swift中的Int相当于Objectire-C中的NSInteger类型。Swift中方法的名称通常用一个介词（比如with，for，by）指向方法的第一个参数，如图13-3中的insertWithObject命名，从第二个参数后，可以指定外部参数名（atIndex）。如果没

有指定，如图13-4所示，会将本地参数名（index）作为外部参数名，在函数中需要在本地参数名前加“#”，而方法不需要。

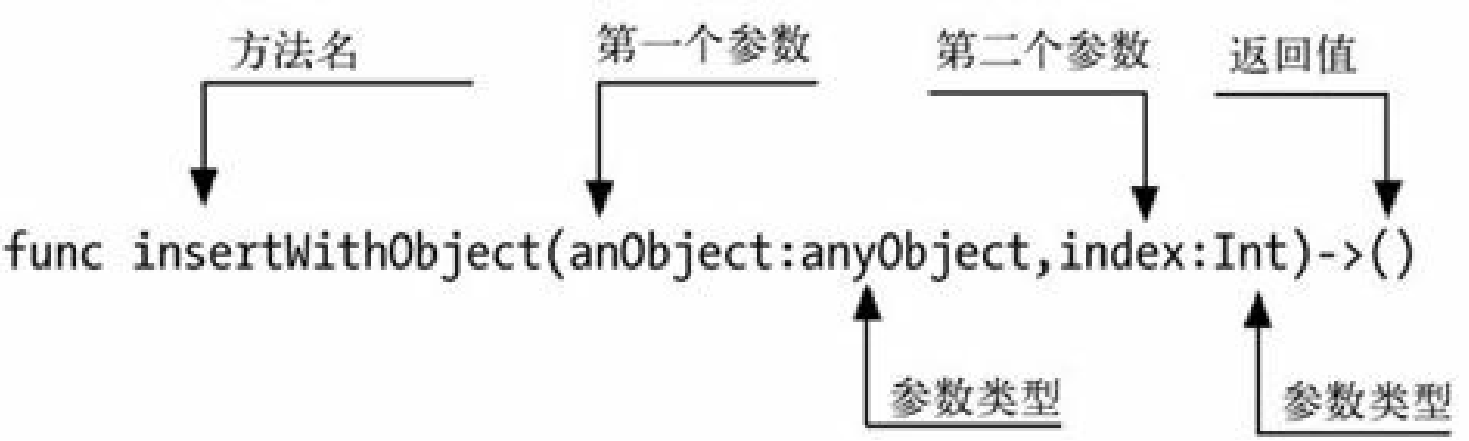


图 13-4　Swift方法定义

外部参数名是在方法外访问时使用的。我们可以采用如下代码调用该方法：

```
实例.insertWithObject("元素", atIndex : 1)
//对应图13-3定义方法的调用
实例.insertWithObject("元素", index : 1)
//对应图13-4定义方法的调用
```

其中“元素”是要插入的数据，atIndex 是外部参数名，index是默认的外部参数名。

示例代码如下：

```
class Employee {
    var no : Int = 0
```

```
    var name : String = ""
    var job : String?
    var salary : Double = 0
    var dept : Department?
}

class Department {
    var no : Int = 0
    var name : String = ""
    var employees : [Employee] = [Employee]()
①

    func insertWithObject(anObject : AnyObject
, atIndex  index : Int)->() {       ②

        let emp = anObject as Employee
③
        employees.insert(emp, atIndex:index)
④

    }
}

var dept = Department()
⑤

var emp1 = Employee()
dept.insertWithObject(emp1, atIndex: 0)

var emp2 = Employee()
```

```
dept.insertWithObject(emp2, atIndex: 0)

var emp3 = Employee()
dept.insertWithObject(emp3, atIndex: 0)
⑥

println(dept.employees.count)
⑦
```

Employee与Department之间是一对多的关系，代码第①行说明一个部门包含多个员工，所以employees是一个Employee的数组。第②行代码是定义insertWithObject方法。在方法体中第③行是将类型为AnyObject的参数anObject强制转换为Employee类型。第④行按照索引插入数据。

代码第⑤~⑥行是实例化Department，并初始化插入3个员工实例。最后我们在第⑦行打印employees属性的长度。

如果在方法中不指定外部参数名，而是以本地参数作为外部参数名，则修改示例代码如下：

```
......
func insertWithObject(anObject : AnyObject ,
```

```
index : Int)->() {

    let emp = anObject as Employee
    employees.insert(emp, atIndex:index)

}
......
dept.insertWithObject(emp1, index: 0)
```

提示 如果insertWithObject是函数，index参数前面是要加“#”号的，代码如下：

```
func insertWithObject(anObject : AnyObject
, #index : Int)->() {......}
```

13.1.2 结构体和枚举方法变异

结构体和枚举中的方法默认情况下是不能修改属性的。我们将上一节的Department定义改成结构体，代码如下：

```
class Employee {
    var no : Int = 0
    var name : String = ""
```

```
    var job : String?
    var salary : Double = 0
    var dept : Department?
}

struct Department {
    var no : Int = 0
    var name : String = ""

    var employees : [Employee] = [Employee]()

    func insertWithObject(anObject : AnyObject
, index : Int)->() {

        let emp = anObject as Employee
        employees.insert(emp, atIndex:index)  ①

    }
}

var dept = Department()

var emp1 = Employee()
dept.insertWithObject(emp1, index: 0)

var emp2 = Employee()
dept.insertWithObject(emp2, index: 0)

var emp3 = Employee()
dept.insertWithObject(emp3, index: 0)
```

```
println(dept.employees.count)
```

上述程序代码第①行会发生编译错误，错误信息如下：

```
Playground execution failed: error:
<REPL>:22:8: error: immutable value of type
'Employee[]' only has mutating members named
'insert'
        employees.insert(emp, atIndex:index)
        ^         ~~~~~~
```

错误提示employees属性不可以修改。如果要修改，就要将方法声明为变异的（mutating）。修改方法声明如下：

```
......
      mutating func insertWithObject(anObject :
AnyObject , index : Int)->() {

      let emp = anObject as Employee
      employees.insert(emp, atIndex:index)

  }
```

```
.....
```

我们在枚举和结构体方法前面添加关键字mutating，将方法声明为变异方法，变异方法能够修改变量属性，但不能修改常量属性。

13.2 静态方法

与静态属性类似，Swift中还定义了静态方法，也称为**类型方法**，所谓“类型”是指枚举、结构体和类。静态方法定义的方法也是与静态属性类似的，枚举和结构体的静态方法使用的关键字是static，类的静态方法使用的关键字是class。

13.2.1 结构体中静态方法

下面我们先看一个结构体静态方法的示例，代码如下：

```
struct Account {

    var owner : String = "Tony"
//账户名          ①
    static var interestRate : Double = 0.668
//利率            ②

    static func interestBy(amount : Double) ->
Double {          ③

        return interestRate * amount
    }

    func messageWith (amount : Double) ->
```

```
String {                    ④

        var interest =
Account.interestBy(amount)
        return "\(self.owner) 的利息是\
(interest)"

    }
}

//调用静态方法
println(Account.interestBy(10_000.00))
⑤

var myAccount = Account()
⑥
//调用实例方法
println(myAccount.messageWith (10_000.00))
⑦
```

上述代码是定义Account结构体，第①行代码声明了实例属性owner。第②行代码声明了静态属性interestRate。第③行代码是定义静态方法interestBy，静态方法与静态计算属性类似，它不能访问实例属性或实例方法。

第④行是定义实例方法messageWith，实例方

法能访问实例属性和方法，也能访问静态属性和方法。在该方法中我们使用self.owner语句，其中self是一个隐藏属性，指代当前类型实例，一般情况下我们不要使用它，除非属性名与变量或常量名发生冲突。

提示 Swift的静态方法中也能使用self，这在其他面向对象的计算机语言中是不允许的。此时self表示当前数据类型，不代表枚举、结构体或类的实例。

13.2.2 枚举中静态方法

下面我们再看一个枚举静态方法的示例，代码如下：

```
enum Account {

    case 中国银行
    case 中国工商银行
    case 中国建设银行
    case 中国农业银行

    static var interestRate : Double = 0.668 //
利率                    ①
```

```
    static func interestBy(amount : Double) ->
Double {          ②
        return interestRate * amount
    }
}

//调用静态方法
println(Account.interestBy(10_000.00 ))
③
```

上述代码是定义Account枚举，第①行代码声明了静态属性interestRate。第②行代码是定义静态方法interestBy，静态方法与静态计算属性类似，它不能访问实例属性或实例方法。第③行代码是调用静态方法。

从示例可以看出，结构体和枚举的静态方法使用定义没有区别。

13.2.3 类中静态方法

下面我们再看一个类方法的示例，代码如下：

```
class Account {

    var owner : String = "Tony"
//账户名
```

```
    class func interestBy(amount : Double) ->
Double {              ①
        return 0.8886 * amount
    }
}

//调用静态方法
println(Account.interestBy(10_000.00 ))
②
```

上述代码是定义Account类，第①行代码是使用关键字class定义静态方法interestBy，静态方法与静态计算属性类似，它不能访问实例属性或实例方法。第②行代码是调用静态方法。

13.3　本章小结

通过对本章内容的学习，我们了解了Swift语言的方法概念、方法的定义以及方法的调用等内容，熟悉了实例方法和静态方法的声明和调用。

13.4 同步练习

1. 判断正误：枚举、结构体和类都可以定义实例方法。

2. 判断正误：枚举、结构体和类都可以定义静态方法。

3. 判断正误：在声明静态方法时使用的关键字是`class`。

4. 判断正误：在声明静态方法时使用的关键字是`static`。

5. 判断正误：类、结构体和枚举中的方法能修改属性。

6. 判断正误：类、结构体和枚举中的方法都可以声明为变异。

7. 判断正误：结构体和枚举可以将方法声明为变异方法，变异方法能够修改变量属性，但不能修常量属性。

8. 判断正误：枚举和结构体的静态方法使用的关键字是`static`，类的静态方法使用的关键字是`class`。

9. 判断正误：实例方法能访问实例属性和方法，也能访问静态属性和方法。

10. **判断正误：静态方法与静态计算属性类似，它不能访问实例属性或实例方法。**

11. **在下列代码横线处填写一个选项使之能够正确运行。**

```
struct Point {
    var x = 0.0, y = 0.0
    _____ func moveByX(deltaX: Double, y deltaY:
Double) {
        self = Point(x: x + deltaX, y: y +
deltaY)
    }
}
```

A. mutating　　B. static　　C. class　　D. 无

第 14 章　构造与析构

结构体和类在创建实例的过程中需要进行一些初始化工作，这个过程称为**构造过程**。相反，在这些实例最后被释放的时候需要进行一些清除资源的工作，这个过程称为**析构过程**。

本章我们将重点介绍构造器和析构器的使用方法。构造器的情况比较复杂，还会涉及继承的相关问题，有关继承部分的构造我们会在第15章再介绍。

14.1 构造器

结构体和类的实例在构造过程中会调用一种特殊的方法init，称为**构造器**。构造器init没有返回值，可以重载。在多个构造器重载的情况下，运行环境可以根据它的外部参数名或参数列表调用合适的构造器。

类似的方法在Objective-C中也称为构造器，在C++中称为构造函数。不同的是，Objective-C中的构造器有返回值，而C++中的构造函数名必须跟类名相同，没有返回值。

14.1.1 默认构造器

结构体和类在构造过程中会调用一个构造器，即便是没有编写任何构造器，它也是在里面的。下面看示例代码：

```
class Rectangle {
①
    var width : Double  = 0.0
②
    var height : Double = 0.0
③
}

var rect = Rectangle()
```

```
④
rect.width = 320.0
⑤
rect.height = 480.0
⑥

println("长方形:\(rect.width) x \(rect.height)")
```

我们在上述代码第①行定义了Rectangle类，存储属性直接进行了初始化，在类中没有任何init的定义。第④行代码是创建实例的过程，这种代码在前面的学习过程中应该很常见了，那么我们有没有问过自己，为什么在类型后面要加一对小括号呢？小括号代表着方法的调用，Rectangle()表示调用了某个方法，这个方法就是构造器init()。

事实上，在Rectangle的定义过程中省略了构造器，相当于如下代码：

```
class Rectangle {
    var width : Double  = 0.0
    var height : Double = 0.0

    init() {

    }
}
```

```
①

    }
}
```

如果Rectangle是结构体，则它的定义如下：

```
struct Rectangle {
    var width : Double = 0.0
    var height : Double = 0.0
}
```

而结构体Rectangle的默认构造器与类Rectangle的默认构造器是不同的，相当于如下代码：

```
struct Rectangle {
    var width : Double = 0.0
    var height : Double = 0.0

    init() {
①

    }
```

```
    init(width : Double, height : Double) {
②
        self.width    = width
        self.height   = height
    }
}
```

结构体Rectangle省略了一些构造器，除了第①行的无参数名的构造器init()之外，还有第②行的有参数名的构造器，该构造器是与存储属性相对应的，关于这种构造器我们还会在14.1.3节详细介绍。

从以上示例可以看出，类和结构体的默认构造也是有所不同的。要调用哪个构造器是根据传递的参数类型和参数名决定的。

14.1.2 构造器与存储属性初始化

构造器的主要作用就是初始化存储属性，我们在init()构造器中初始化存储属性width和height后，那么在定义它们时就不需要初始化了。

修改Rectangle代码如下：

```
calss Rectangle {
    var width : Double
    var height : Double

    init() {
        width   = 0.0
        height  = 0.0
    }
}
```

如果存储属性在构造器中没有初始化，在定义的时候也没有初始化，那么就会发生编译错误。

构造器还可以初始化常量存储属性，下面我们看示例代码：

```
class Employee {                    ①
    let no : Int                    ②
    var name : String?              ③
    var job : String?               ④
    var salary : Double             ⑤
    var dept : Department?          ⑥

    init() {                        ⑦
        no = 0                      ⑧
        salary = 0.0                ⑨
    }
```

```
}

struct Department {                    ⑩
    let no : Int                       ⑪
    var name : String                  ⑫

    init() {                           ⑬
        no = 10                        ⑭
        name = "SALES"                 ⑮
    }
}

let dept = Department()
var emp = Employee()
```

上述代码第①行和第⑩行分别定义了Employee类和Department结构体。其中，Employee的no属性（见第②行）和Department的no属性（见第②行）都是常量类型属性。在我们学习常量的时候，曾讲过常量只能在定义的同时赋值，而在构造器中，常量属性可以不遵守这个规则，它们可以在构造器中赋值，参见代码第⑦行和第⑭行，这种赋值不能放在普通方法中。

另外，存储属性一般在定义的时候初始化。如

果不能确定初始值，可以采用可选类型属性，见第③行、第④行和第⑥行代码。或者也可以在构造器中初始化，见代码第⑮行。

14.1.3 使用外部参数名

为了增强程序的可读性，Swift中的方法和函数可以使用外部参数名。在构造器中也可以使用外部参数名。构造器中的外部参数名要比一般的方法和函数更有意义，由于构造器命名都是init，如果一个对象类型中有多个构造器，我们就可以通过不同的外部参数名区分不同的构造器。

下面看示例代码：

```
class RectangleA {
    var width : Double
    var height : Double

    init(W width : Double,H height : Double) {
①
        self.width   = width
②
        self.height  = height
③
    }
}
```

```
var recta = RectangleA(W : 320, H : 480)
④
println("长方形A:\(recta.width) x \
(recta.height)")
```

上述代码第①行是定义构造器init(W width : Double,H height : Double)，这个构造器有两个参数width和height，并且我们为参数提供了外部参数名W和H。

代码第②行和第③行是参数赋值给属性，其中使用了self关键字，表示当前实例，self.width表示当前实例的width属性，在参数命名与属性命名发生冲突时使用self，参数的作用域是构造器体，在参数与属性发生命名冲突时，参数屏蔽了属性，这种情况下引用属性前面要加self。

第④行代码是创建RectangleA实例，这里使用了外部参数名。

提示 上述示例中，虽然我们定义的是类，但也完全适用于结构体。

外部参数名可以简化，在函数中可以在参数前加#，使得局部参数名变成外部参数名。但在构造器中就不用那么麻烦，构造器中的局部参数名可以直接作为外部参数名使用。

下面看示例代码：

```
class RectangleB {
    var width : Double
    var height : Double

    init(width : Double, height : Double) {
①
        self.width   = width
        self.height  = height
    }
}

var rectb = RectangleB(width : 320, height :
480)          ②
println("长方形B:\(rectb.width) x \
(rectb.height)")
```

上述代码第①行定义构造器init(width : Double, height : Double)，其中没有声明外部参数名。在第②行代码调用构造器时，我们使

用了外部参数名width和height，这些外部参数名就是局部参数名。

前面介绍的几个示例适用于类和结构体，但以下写法只适用于结构体类型，如果在结构体中使用默认的构造器，则示例代码如下：

```
struct RectangleD {
    var width : Double = 0.0
    var height : Double = 0.0
}
```

代码中使用了默认的构造器，调用它的时候可以声明外部参数名，结构体类型可以按照从上到下的顺序，把属性名作为外部参数名，依次提供参数。构造器调用代码如下：

```
var rectc = RectangleD(width : 320, height:
480)
```

width和height是属性名，参数顺序是属性的定义顺序。

提示　这种写法是一种默认构造器，但只适用于结构体，在类中不能使用。

14.2 构造器重载

我们在第9章介绍过函数重载，与函数一样，方法也存在重载，其重载的方式与函数一致。那么作为构造器的特殊方法，是否也存在重载呢？答案是肯定的。

14.2.1 构造器重载概念

Swift中函数重载的条件也适用于构造器，条件如下：

- 函数有相同的名字；
- 参数列表不同或返回值类型不同，或外部参数名不同。

Swift中的构造器可以满足以下两个条件，代码如下：

```
class Rectangle {

    var width : Double
    var height : Double

    init(width : Double, height : Double) {
①
        self.width   = width
```

```
        self.height  = height
    }

    init(W width : Double,H height : Double) {
②
        self.width   = width
        self.height  = height
    }

    init(length : Double) {
③
        self.width   = length
        self.height  = length
    }

    init() {
④
        self.width   = 640.0
        self.height  = 940.0
    }

}

var rectc1 = Rectangle(width : 320.0, height :
480.0)      ⑤
println("长方形:\(rectc1.width) x \
(rectc1.height)")

var rectc2 = Rectangle(W : 320.0, H : 480.0)
⑥
```

```
println("长方形:\(rectc2.width) x \
(rectc2.height)")

var rectc3 = Rectangle(length: 500.0)
⑦
println("长方形3:\(rectc3.width) x \
(rectc3.height)")

var rectc4 = Rectangle()
⑧
println("长方形4:\(rectc4.width) x \
(rectc4.height)")
```

上述代码第①~④行定义了4个构造器，其他是重载关系。从参数个数和参数类型上看，第①行和第②行的构造器是一样的，但是它们的外部参数名不同，所以在第⑤行调用的是第①行的构造器，第⑥行调用的是第②行的构造器。

第③行和第④行的构造器参数个数与第①行不同，所以在第⑦行调用的是第③行的构造器，第④行调用的是第⑧行的构造器。

14.2.2 值类型构造器代理

为了减少多个构造器间的代码重复，在定义构造器时，可以通过调用其他构造器来完成实例的部

分构造过程，这个过程称为**构造器代理**。构造器代理在值类型和引用类型中使用方式不同，本节我们先介绍值类型构造器代理。

将上一节的示例修改如下：

```
struct Rectangle {

    var width : Double
    var height : Double

    init(width : Double, height : Double) {
①
        self.width   = width
        self.height  = height
    }

    init(W width : Double,H height : Double) {
②
        self.width   = width
        self.height  = height
    }

    init(length : Double) {
③
        self.init(W : length, H : length)
    }

    init() {
```

```
④
        self.init(width: 640.0, height: 940.0)
    }

}

var rectc1 = Rectangle(width : 320.0, height :
480.0)      ⑤
println("长方形:\(rectc1.width) x \
(rectc1.height)")

var rectc2 = Rectangle(W : 320.0, H : 480.0)
⑥
println("长方形:\(rectc2.width) x \
(rectc2.height)")

var rectc3 = Rectangle(length: 500.0)
⑦
println("长方形3:\(rectc3.width) x \
(rectc3.height)")

var rectc4 = Rectangle()
⑧
println("长方形4:\(rectc4.width) x \
(rectc4.height)")
```

将Rectangle声明为结构体类型，其中也有4

个构造器重载。在第③行和第④行的构造器中使用了self.init语句，self指示当前实例本身，init是本身的构造器，第③行的self.init(W : length, H : length)语句是在调用第②行定义的构造器，第④行的self.init(width: 640.0, height: 940.0)语句是在调用第①行定义的构造器。

这种在同一个类型中通过self.init语句进行调用就是我们说的构造器代理。

14.2.3 引用类型构造器横向代理

引用类型构造器代理就是类构造器代理。由于类有继承关系，类构造器代理比较复杂，分为横向代理和向上代理。

- **横向代理**类似于值类型构造器代理，发生在同一类内部，这种构造器称为**便利构造器**（convenience initializers）。
- **向上代理**发生在继承情况下，在子类构造过程中要先调用父类构造器，初始化父类的存储属性，这种构造器称为**指定构造器**（designated initializers）。

由于我们还没有介绍继承，因此本章只介绍横向代理。

将上一节的示例修改如下：

```
class Rectangle {

    var width : Double
    var height : Double

    init(width : Double, height : Double) {
①
        self.width   = width
        self.height  = height
    }

    init(W width : Double,H height : Double) {
②
        self.width   = width
        self.height  = height
    }

    convenience init(length : Double) {
③
        self.init(W : length, H : length)
    }

    convenience init() {
④
```

```
        self.init(width: 640.0, height: 940.0)
    }

}

var rectc1 = Rectangle(width : 320.0, height :
480.0)          ⑤
println("长方形:\(rectc1.width) x \
(rectc1.height)")

var rectc2 = Rectangle(W : 320.0, H : 480.0)
⑥
println("长方形:\(rectc2.width) x \
(rectc2.height)")

var rectc3 = Rectangle(length: 500.0)
⑦
println("长方形3:\(rectc3.width) x \
(rectc3.height)")

var rectc4 = Rectangle()
⑧
println("长方形4:\(rectc4.width) x \
(rectc4.height)")
```

将Rectangle声明为类，其中也有4个构造器重载。在第③行和第④行的构造器中使用了

self.init语句，并且在构造器前面加上了convenience关键字，convenience表示便利构造器，这说明我们定义构造器是横向代理调用其他构造器。

第③行的self.init(W : length, H : length)语句是在横向调用第②行定义的构造器代理，第④行的self.init(width: 640.0, height: 940.0)语句是在横向调用第①行定义的构造器代理。

14.3 析构器

与构造过程相反，实例最后释放的时候，需要清除一些资源，这个过程就是析构过程。在析构过程中也会调用一种特殊的方法deinit，称为**析构器**。析构器deinit没有返回值，也没有参数，所以不能重载。析构函数只适用于类类型，不能应用于枚举和结构体。

类似的方法在C++中称为**析构函数**，不同的是，C++中的析构函数常常用来释放不再需要的实例内存资源。而在Swift 中，内存管理采用自动引用计数（ARC），不需要在析构器释放不需要的实例内存资源，但是还是有一些清除工作需要在这里完成，如关闭文件等处理。

下面看示例代码：

```
class Rectangle {

    var width : Double
    var height : Double

    init(width : Double, height : Double) {
        self.width   = width
        self.height  = height
    }
```

```
    init(W width : Double,H height : Double) {
        self.width   = width
        self.height  = height
    }

    deinit {
①
        println("调用析构器...")
        self.width = 0.0
        self.height  = 0.0
    }
}

var rectc1 : Rectangle? = Rectangle(width :
320, height : 480)        ②
println("长方形:\(rectc1!.width) x \
(rectc1!.height)")
rectc1 = nil
③

var rectc2 : Rectangle? = Rectangle(W : 320, H
: 480)                    ④
println("长方形:\(rectc2!.width) x \
(rectc2!.height)")
rectc2 = nil
⑤
```

上述代码第①行使用deinit关键字定义了析构

器，在这个析构器中重新设置了存储属性值。第②行代码创建了Rectangle实例rectc1。注意rectc1类型是Rectangle?，这说明rectc1是可选类型，可以被赋值为nil，类似地，第④行代码也创建了Rectangle可选类型实例rectc2。可选类型在使用时后面要加“!”，如rectc1!.width。

那么什么时候调用析构器呢？实例被赋值为nil，表示实例需要释放内存，在释放之前先调用析构器，然后再释放。代码第③行和第④行是触发调用析构器的条件。

运行结果如下：

```
长方形:320.0 x 480.0
调用析构器...
长方形:320.0 x 480.0
调用析构器...
```

在Playground环境下测试和运行上述代码，是不会调用析构器的，还需要使用Xcode创建一个iOS应用程序，然后在应用程序中运行这段代码，才会调用析构器。

具体过程是：启动Xcode 6，然后单击File→New→Project菜单，在打开的Choose a template for your new project界面中选择“Single View Application”工程模板（如图14-1所示）。

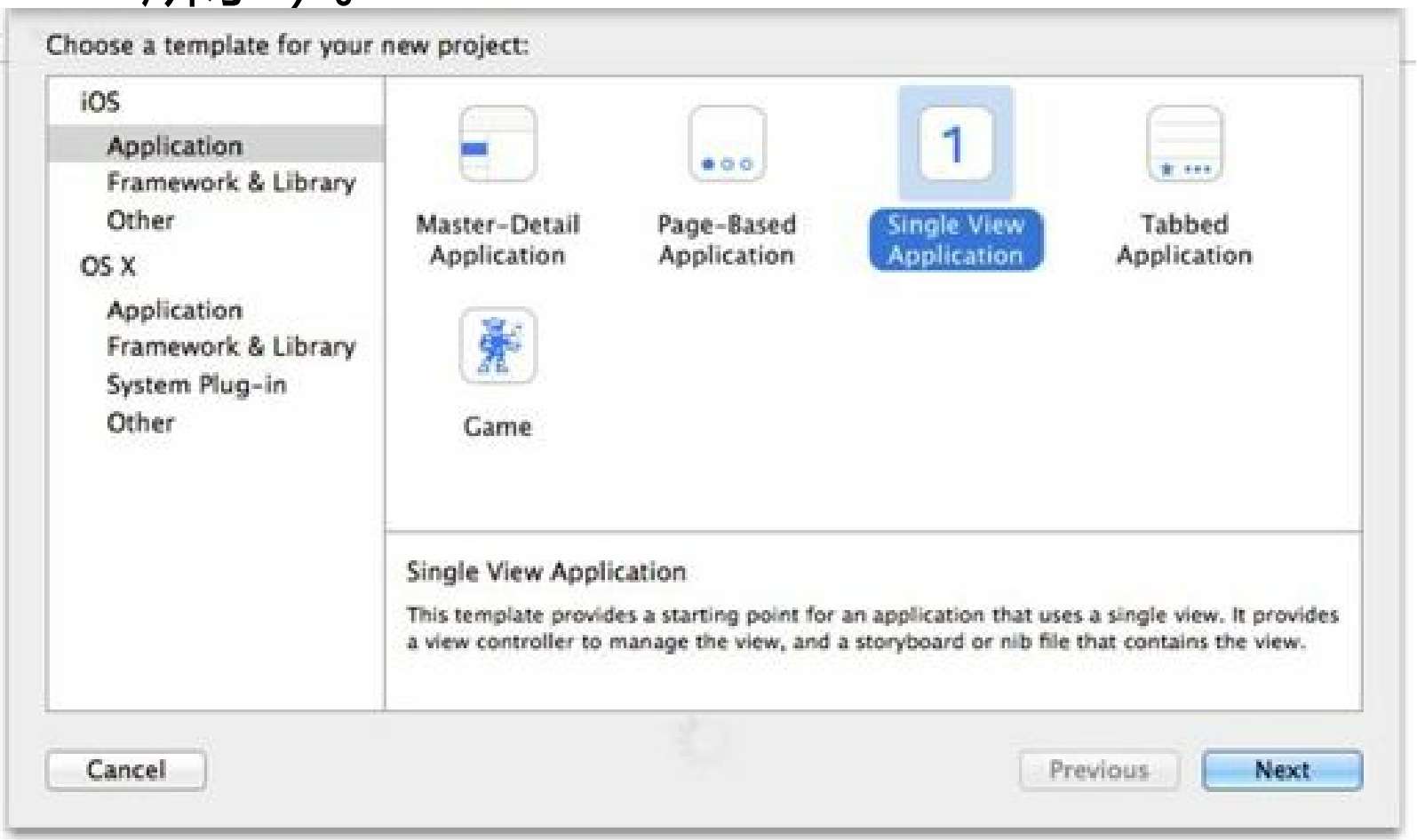


图 14-1　选择工程模板

接着单击“Next”按钮，随即出现如图14-2所示的界面。

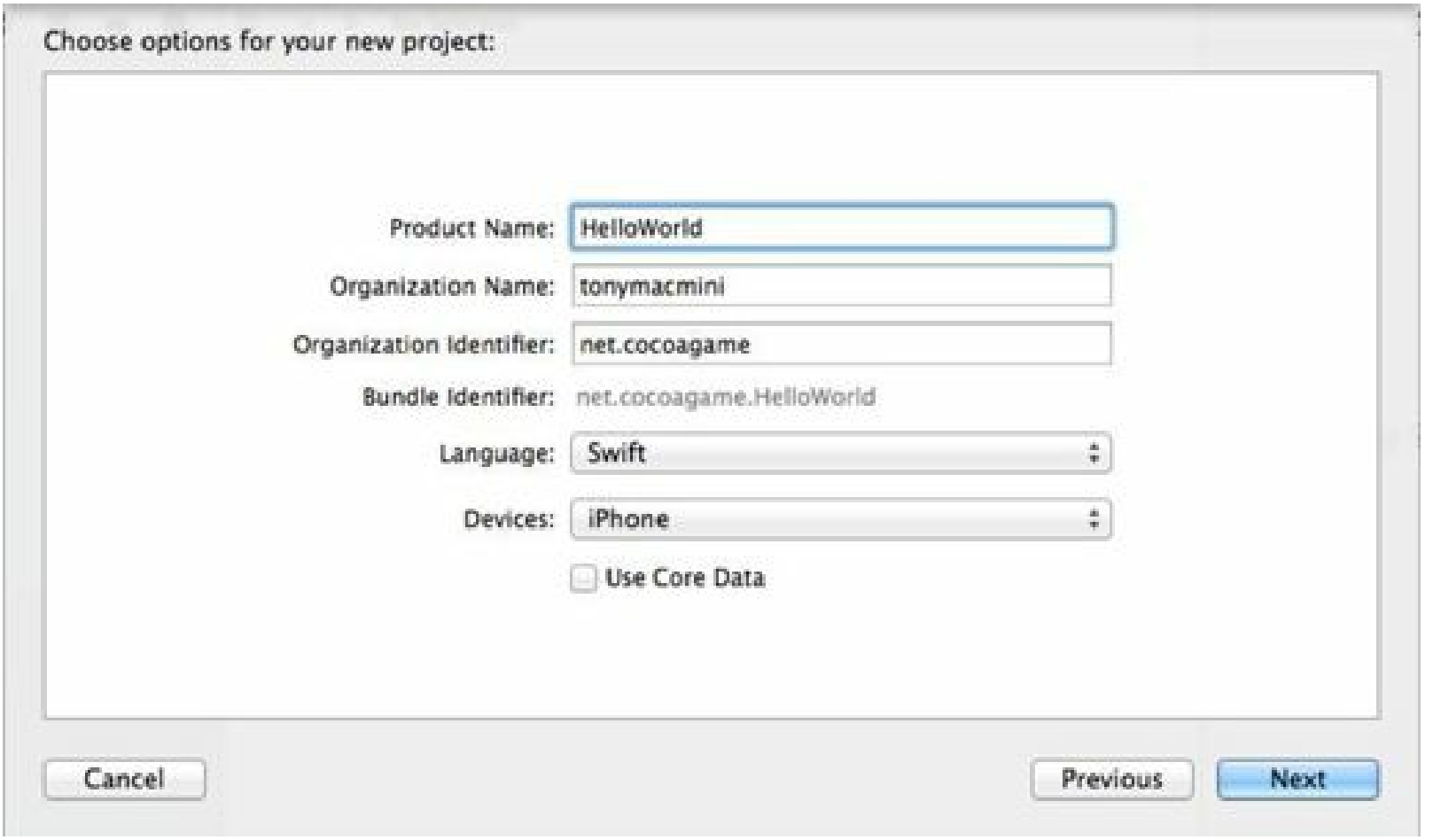


图 14-2　新工程中的选项

在图14-2所示的对话框的Product Name中，输入一个工程名称，其他输入项目读者可以参考图14-2，这些输入项目我们会在第20章详细解释，这里不再赘述。设置完相关的工程选项后，单击“Next”按钮，进入下一级界面。根据提示选择存放文件的位置，然后单击“Create”按钮，将出现如图14-3所示的界面。

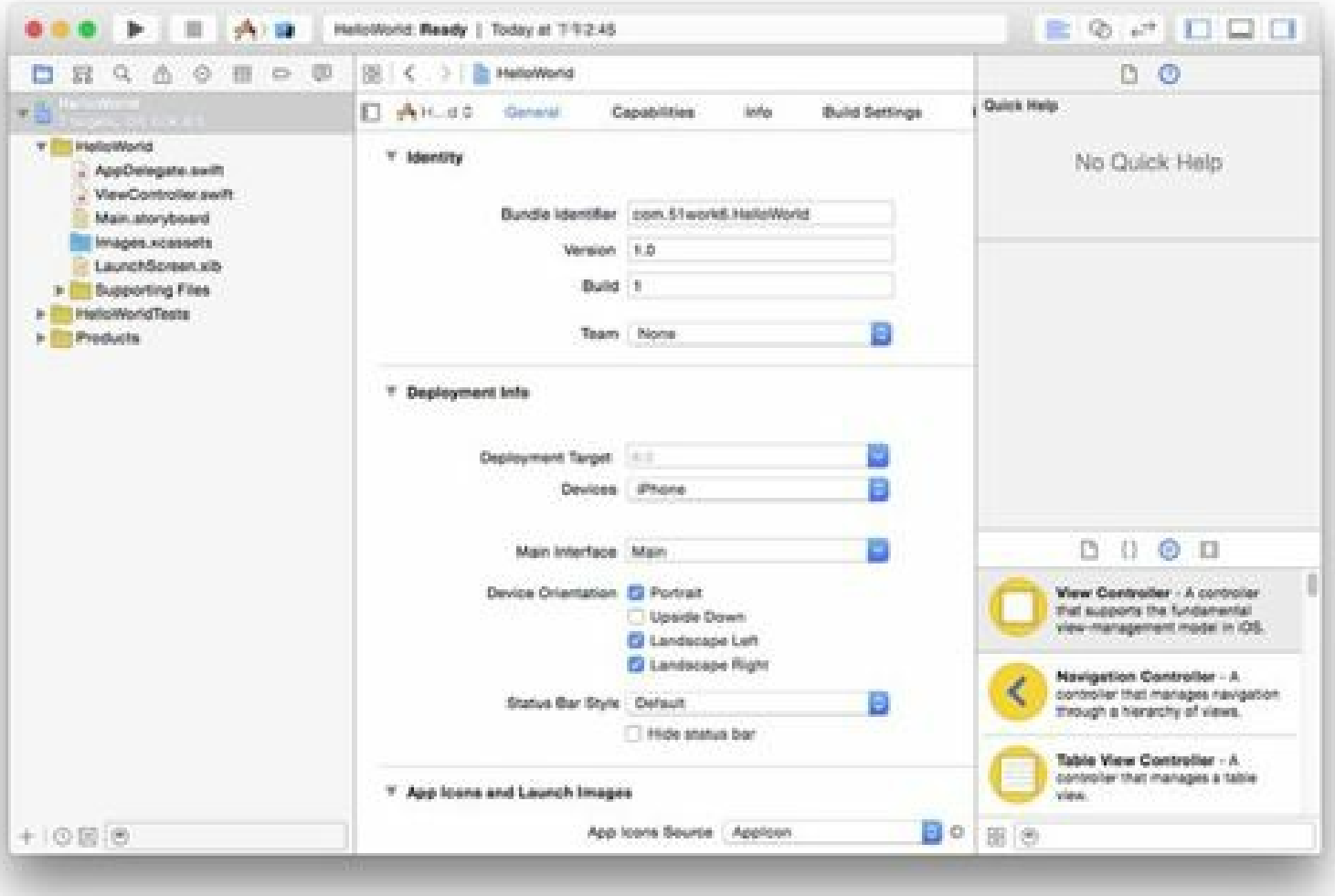


图 14-3　新创建的工程

由于我们并不需要运行iOS应用中的那些UI界面，因此在工程中选中AppDelegate.swift、ViewController.swift和Main.storyboard这3个文件，在右键菜单中选择Delete删除这3个文件，之后的Xcode工程界面如图14-4所示。

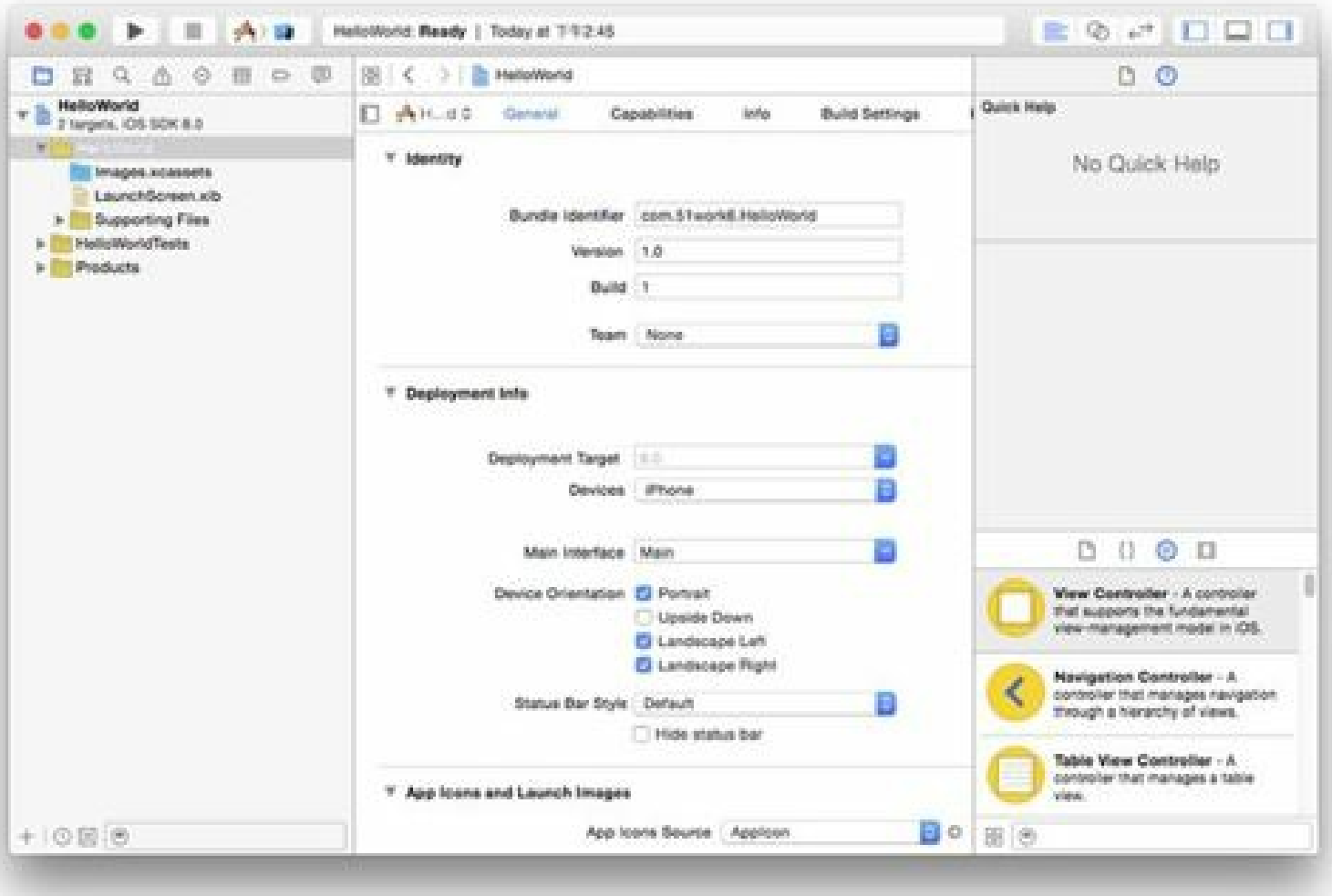


图 14-4　删除文件

接着在工程中添加我们测试使用的Swift文件，选择图14-4所示的HelloWorld工程，在右键菜单中选择New File...，弹出如图14-5所示的对话框，选择iOS→Source→Swift File，然后单击“Next”按钮，进入下一级界面，如图14-6所示，在“Save AS”中输入文件名为main，选择存放文件的位置，然后单击“Create”按钮创建Swift文件。

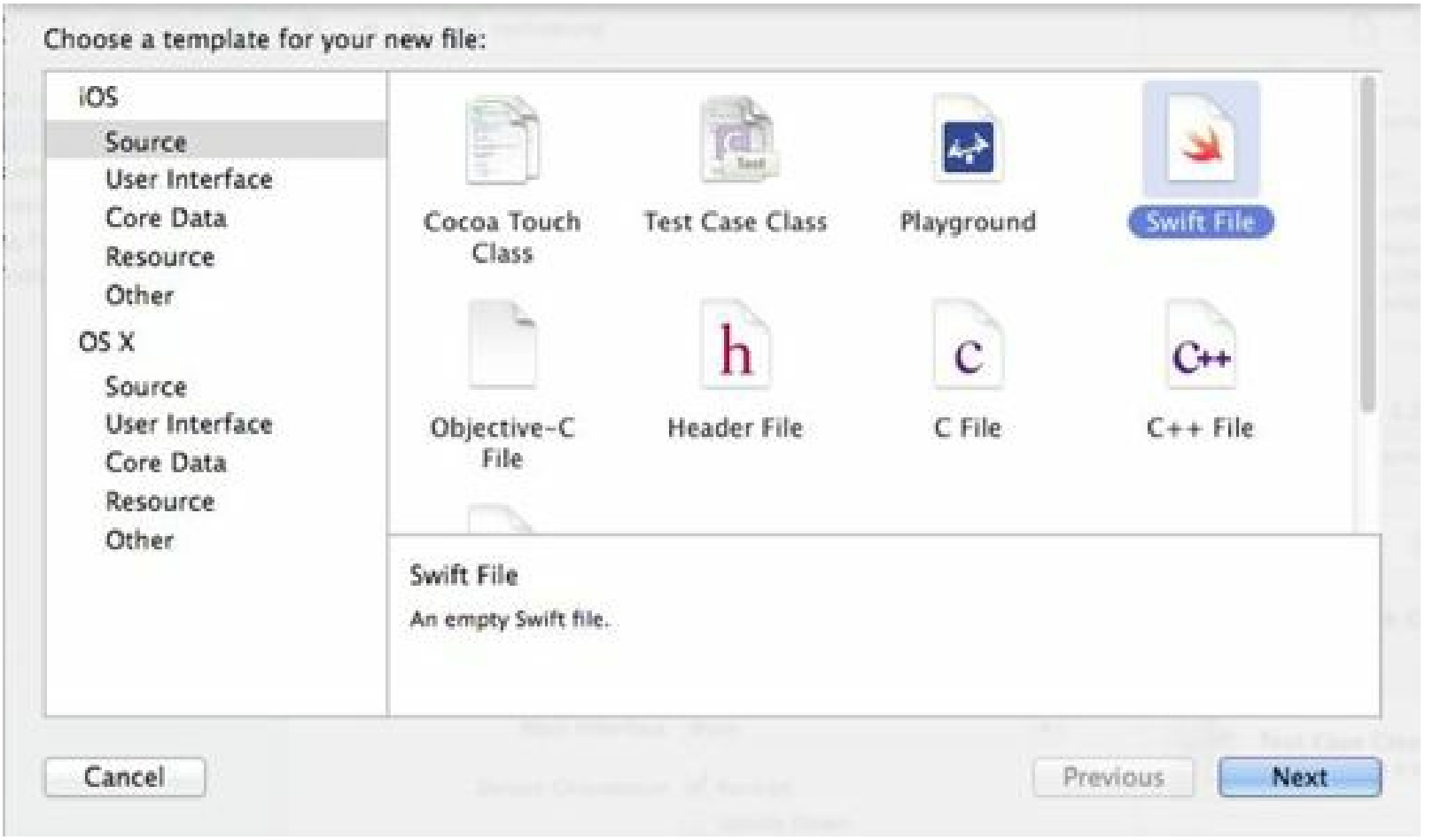


图 14-5　在工程中添加Swift文件

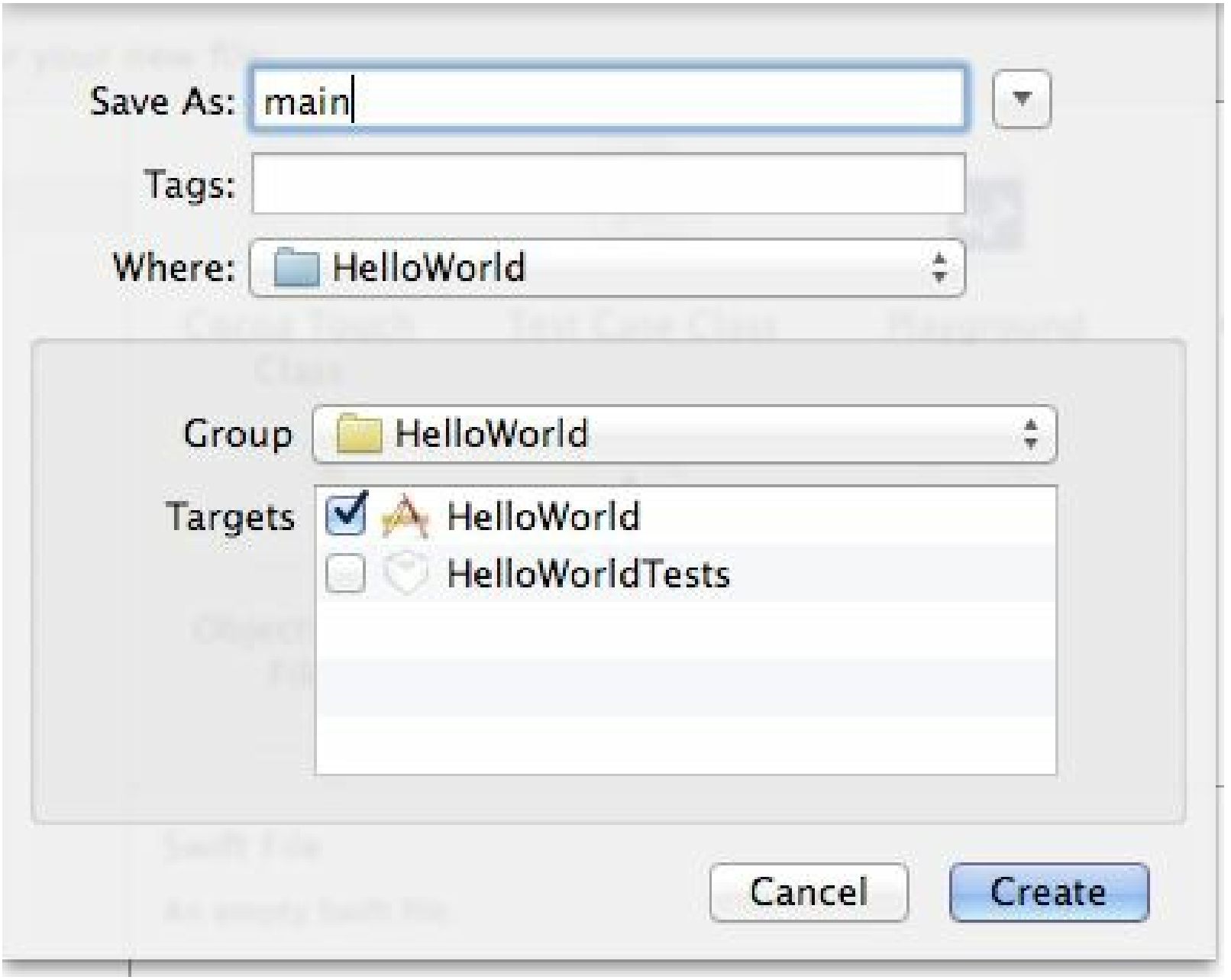


图 14-6　保存文件

Swift文件创建成功之后，在main.swift中编写前面介绍的构造器代码，编写完成后我们就可以运行测试了。具有的运行过程是：如图14-7所示，在工具栏中选择运行的模拟器，然后单击左上角的运行按钮  ，iOS应用就会运行，并且会启动一个模拟器，我们不用关心这个模拟器界面，只需要图

14-8所示的日志信息就可以了。

图 14-7　运行应用

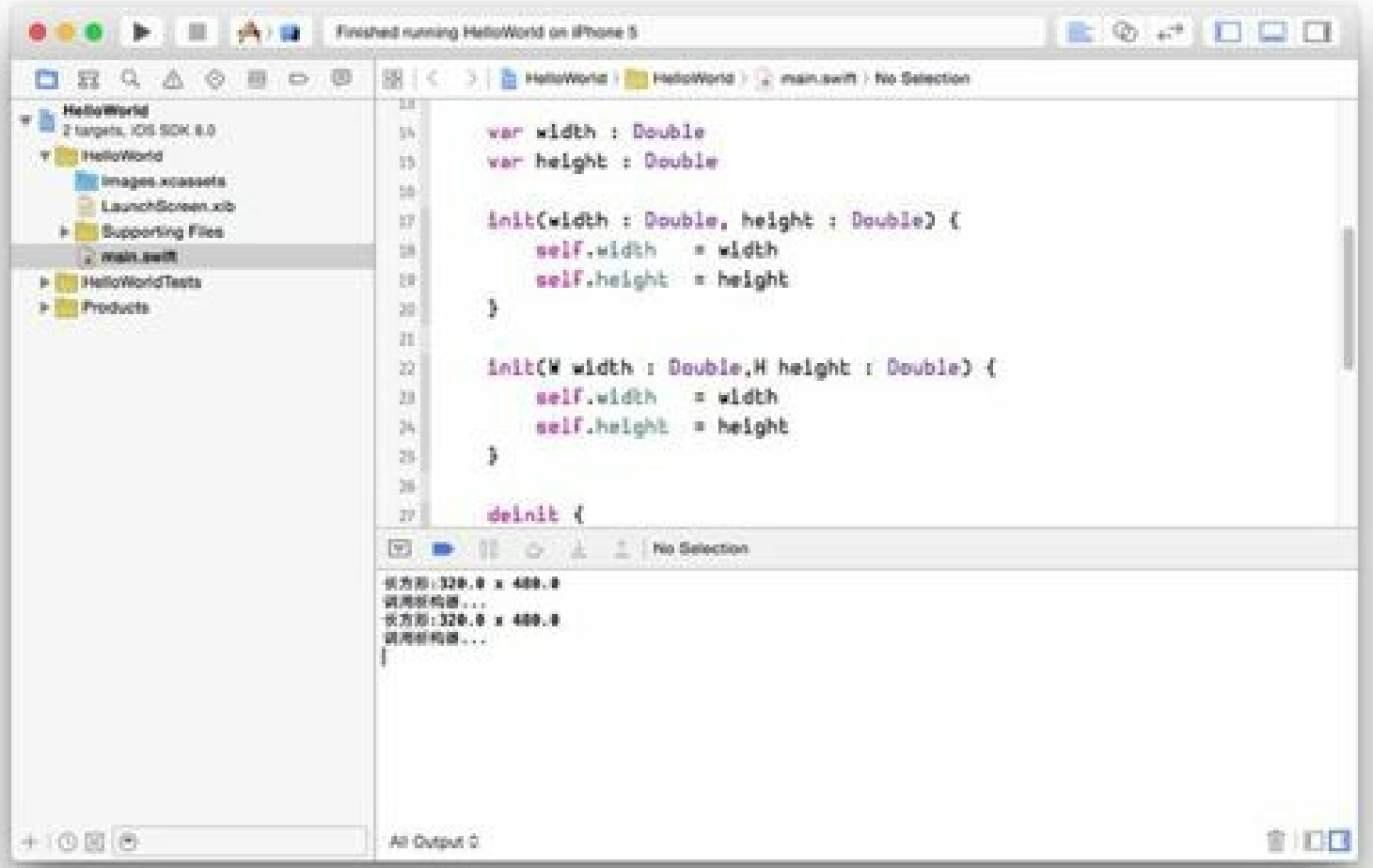


图 14-8　运行应用的日志信息

关于iOS的其他知识我们先不用考虑，只要能够通过上述步骤创建一个iOS测试工程，并且能够运行测试就可以了，其他深层次的内容我们会在第20章再详细解释。

14.4　本章小结

通过对本章内容的学习，我们了解了Swift对象类型的构造过程和析构过程，掌握了构造器和析构器的使用方法。

14.5 同步练习

1. 判断正误：枚举、结构体和类的实例在构造过程中会调用构造器。

2. 判断正误：枚举、结构体和类的实例在析构过程中也会调用析构器。

3. 判断正误：析构器deinit没有返回值，也没有参数，所以不能重载。

4. 判断正误：构造器init没有返回值，可以重载，多个构造器重载情况下，运行环境可以根据它的外部参数名或参数列表调用合适构造器。

5. 判断正误：构造器的主要作用就是初始化存储属性。

6. 判断正误：构造器的主要作用就是初始化计算属性。

7. 判断正误：存储属性要么在定义的时候初始化，要么在构造器中初始化。

8. 判断正误：关键字convenience可以修改引用类型的构造器，这种构造器能够用于横向代理。

9. 下列选项中会发生编译错误的是（　　）。

A.

```
struct Circle {
```

```
    var R : Double
    init() {
        R = 0
    }
}
```

B.

```
struct Circle {
    var R : Double
    init() {
    }
}
```

C.

```
struct Circle {
    var R : Double
}
```

D.

```
struct Circle {
    var R : Double = 0.0
```

```
}
```

10. 下列选项中会发生编译错误的是（　　）。

A.

```
class Circle {
    var R : Double
}
```

B.

```
class Circle {
    var R : Double = 0.0
}
```

C.

```
class Circle {
    var R : Double
    init() {
    }
}
```

D.

```
class Circle {
    var R : Double
    init() {
        R = 0.0
    }
}
```

11. 下列选项中会发生编译错误的是（　　）。

A.

```
class Dog {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "未命名")
    }
}
```

B.

```
class Dog {
    var name: String
    init(name: String) {
        self.name = name
    }
    init() {
        self.init(name: "未命名")
    }
}
```

C.

```
class Dog {
    var name: String
    init(name: String) {
        self.name = name
    }
    init() {
        self.name = "未命名"
    }
}
```

D.

```
class Dog {
```

```
    var name: String = "未命名"
    init(name: String) {
        self.name = name
    }
}
```

12. 关于析构器，定义正确的是（　　）。

A. `deinit`　　B. `deinit()`　　C. `Deinit`　　D. `Deinit()`

第 15 章　继承

继承性是面向对象的重要特征之一。Swift中的继承只能发生在类上，不能发生在枚举和结构体上。在Swift中，一个类可以继承另一个类的方法、属性、下标等特征，当一个类继承其他类时，继承类叫子类，被继承类叫父类（或超类）。子类继承父类后，可以重写父类的方法、属性、下标等特征。

15.1 从一个示例开始

为了了解继承性，我们先看这样一个场景：一位面向对象的程序员小赵，在编程过程中需要描述和处理个人信息，于是他定义了类Person，如下所示：

```
class Person {
    var name : String
    var age : Int

    func description() -> String {
        return "\(name) 年龄是: \(age)"
    }
    init () {
        name = ""
        age  = 1
    }
}
```

一周以后，小赵又遇到了新的需求，需要描述和处理学生信息，于是他又定义了一个新的类Student，如下所示：

```
class Student {
    var name : String
```

```
    var age : Int

    var school : String

    func description() -> String {
        return "\(name) 年龄是: \(age)"
    }
    init() {
        school = ""
        name = ""
        age  = 8
    }
}
```

很多人会认为小赵的做法能够理解并相信这是可行的，但是问题在于Student和Person两个类的结构太接近了，后者只比前者多了一个属性school，却要重复定义其他所有的内容，实在让人“不甘心”。Swift提供了解决类似问题的机制，那就是**类的继承**，代码如下所示：

```
class Student : Person {
    var school : String
    override init() {
        school = ""
        super.init()
```

```
        age  = 8
    }
}
```

Student类继承了Person类中的所有特征，“:”之后的Person类是父类。Swift中的类可以没有父类，例如Person类，定义的时候后面没有“:”，这种没有父类的就是**基类**。Swift中的继承与在Objective-C等面向对象的语言不同，在Objective-C中，所有类的基类都是NSObject，在Swift中没有规定这样的一个类。此外override init()是子类重写父类构造器。

提示 Swift中也没有像Objective-C等面向对象的语言有访问限定符，如private（私有的）、public（公有的）和protected（保护的）。由于没有公有、私有和保护之分，父类的所有特征（方法、属性、下标）都可以被子类继承。

一般情况下，一个子类只能继承一个父类，这称为**单继承**，但有的情况下一个子类可以有多个不

同的父类，这称为**多重继承**。在Swift中，类的继承只能是单继承。多重继承可以通过协议实现。也就是说，在Swift中，一个类只能继承一个父类，但是可以遵守多个协议。

15.2 构造器继承

我们在第14章介绍过构造与析构，在一个实例的构造过程中会调用构造器这样一个特殊的方法。在构造器中可以使用构造器代理帮助完成部分构造工作。类构造器代理分为横向代理和向上代理，横向代理只能在发生在同一类内部，这种构造器称为便利构造器。向上代理发生在继承的情况下，在子类构造过程中，要先调用父类构造器初始化父类的存储属性，这种构造器称为指定构造器。

第14章只介绍了便利构造器调用，本节将介绍向上代理和指定构造器调用。

15.2.1 构造器调用规则

我们先看看上一节的Person和Student类示例，修改代码如下：

```
class Person {
    var name : String
    var age : Int

    func description() -> String {
        return "\(name) 年龄是: \(age)"
    }
    convenience init () {
①
```

```
        self.init(name: "Tony")
        self.age = 18
    }
    convenience init (name : String) {
②
        self.init(name: name, age: 18)
    }
    init (name : String, age : Int) {
③
        self.name = name
        self.age  = age
    }
}

class Student : Person {
    var school : String
    init (name : String, age : Int, school :
String) {                    ④
        self.school = school
        super.init(name : name, age : age)
    }
    convenience override init (name : String,
age : Int) {                ⑤
        self.init(name : name, age : age,
school : "清华大学")
    }
}

let student = Student()
println("学生: \(student.description())")
```

在Person类中定义了3个构造器，见代码第①~③行，其中，第①行和第②行是便利构造器，第③行是指定构造器。在Student类中，定义了2个构造器，见代码第④~⑤行，其中第④行是指定构造器，第⑤行是便利构造器。

构造器之间的调用形成了构造器链，如图15-1所示，我们为这个构造器添加了标号。

Swift限制构造器之间的代理调用的规则有3条，如下所示。

- **指定构造器必须调用其直接父类的的指定构造器。**从图15-1可见，Student中的④号指定构造器调用Person中的③号指定构造器。
- **便利构造器必须调用同一类中定义的其他构造器。**从图15-1可见，Student中的⑤号便利构造器调用同一类中的④号便利构造器，Person中的①号便利构造器调用同一类中的②号便利构造器。
- **便利构造器必须最终以调用一个指定构造器结束。**从图15-1可见，Student中的

⑤号便利构造器调用同一类中的④号指定构造器，Person中的②号便利构造器调用同一类中的③号指定构造器。

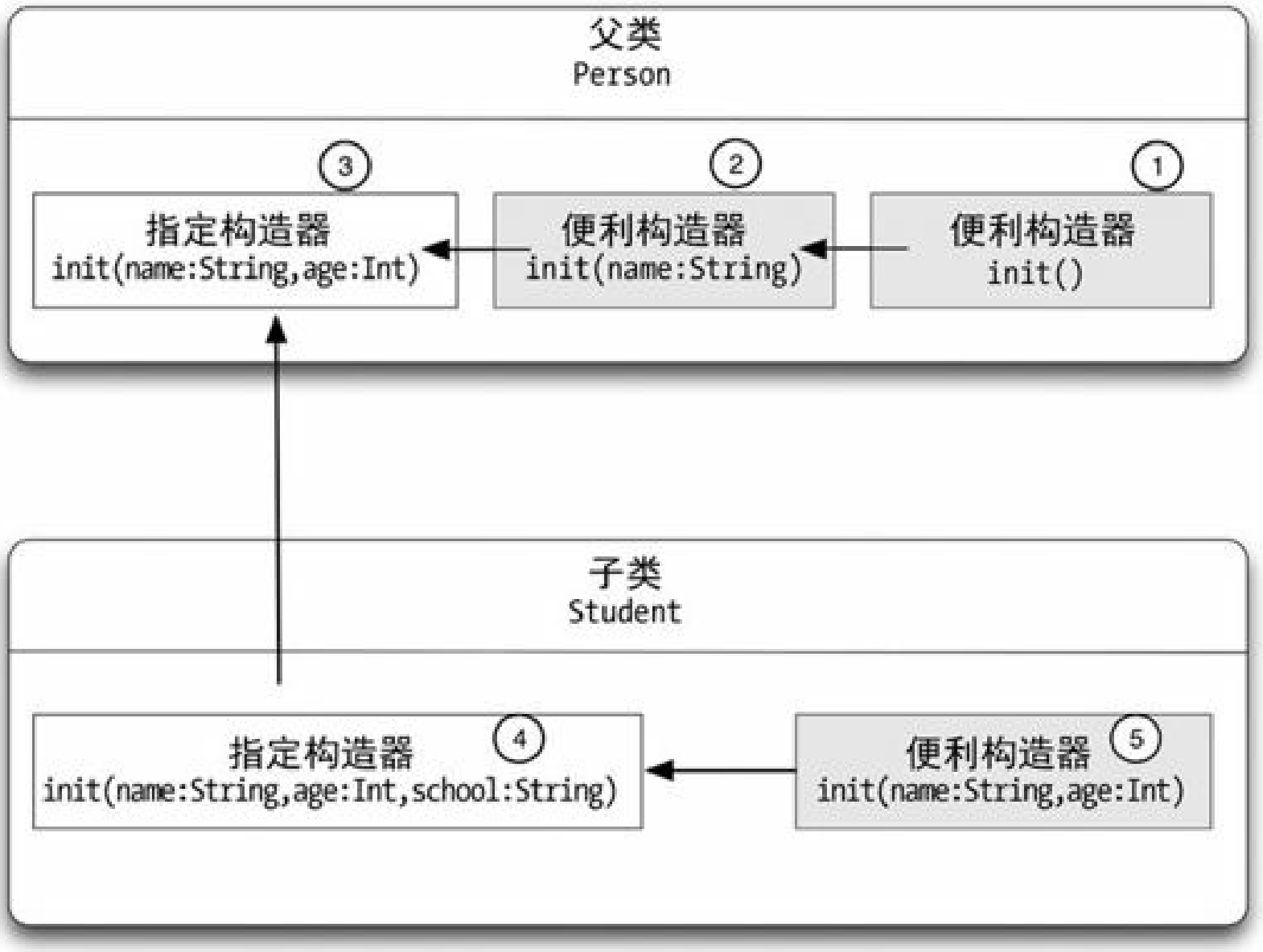


图 15-1　构造器链

15.2.2　构造过程安全检查

在Swift中，类的构造过程包含两个阶段，如图15-2所示。

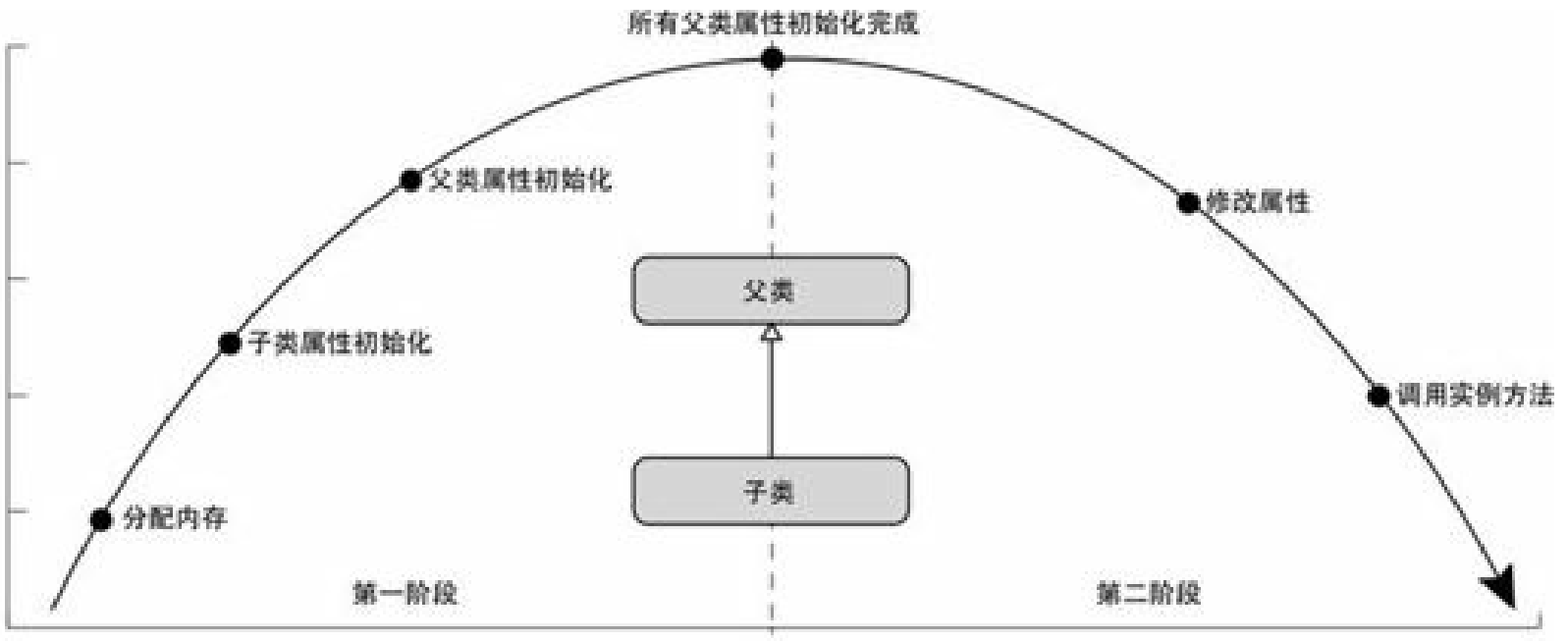


图 15-2　构造过程的两个阶段

第一阶段，首先分配内存，初始化子类存储属性，沿构造器链向上初始化父类存储属性，到达构造器链顶部，初始化全部的父类存储属性。

第二阶段，从顶部构造器链往下，可以对每个类进行进一步修改存储属性、调用实例方法等处理。

Swift编译器在构造过程中可以进行一些安全检查工作，这些工作可以有效地防止属性在初始化之前被访问，也可以防止属性被另外一个构造器意外地赋予不同的值。

为确保构造过程顺利完成，Swift提供了4种安全检查。

1. **安全检查1**

指定构造器必须保证其所在类的所有存储属性

都初始化完成，之后才能向上调用父类构造器代理。

示例代码如下：

```
class Student : Person {
    var school : String

    init (name : String, age : Int, school : String) {  ①
        self.school = school  ②
        super.init(name : name, age : age)  ③
    }

    convenience override init (name : String, age : Int) {
        self.init(name : name, age : age, school : "清华大学")
    }
}
```

上述代码第①行和第④行是定义构造器，其中第②行代码是初始化school属性，它一定要在第③行super.init(name : name, age : age)语句之前调用，super.init(name : name, age : age)是向上调用父类构造器代

理。如果我们把第②行和第③行的代码互换一下，会出现如下编译错误：

```
error: property 'self.school' not initialized
at super.init call
        super.init(name : name, age : age)
        ^
```

使用两段式构造过程分析上述代码，语句①~③都还属于第一阶段。

2. **安全检查2**

指定构造器必须先向上调用父类构造器代理，然后再为继承的属性设置新值，否则指定构造器赋予的新值将被父类中的构造器所覆盖。

示例代码如下：

```
class Student : Person {
    var school : String
    init (name : String, age : Int, school :
String) {          ①
        self.school = school
②
        super.init(name : name, age : age)
③
        self.name = "Tom"
```

```
④
        self.age = 28
⑤
    }

    convenience override init (name : String,
age : Int) {
        self.init(name : name, age : age,
school : "清华大学")
    }
}
```

上述代码第①行定义指定构造器init (name : String, age : Int, school : String)，第②行代码初始化school属性，这条语句必须要放在向上调用父类构造器代理之前（见第③行代码super.init(name : name, age : age)）。安全检查2关键是检查第④行和第⑤行代码，这两行代码是为从父类继承下来的name和age属性赋值，这要在向上调用父类构造器代理之后进行。

使用两段式构造过程分析上述init (name : String, age : Int, school : String)构造器，语句②和③属于第一阶段，语句④和⑤属

于第二阶段。

3. **安全检查3**

便利构造器必须先调用同一类中的其他构造器代理，然后再为任意属性赋新值，否则便利构造器赋予的新值将被同一类中其他指定构造器所覆盖。

示例代码如下：

```
class Student : Person {
    var school : String
    ......

        convenience override init (name :
String, age : Int) {       ①
        self.init(name : name, age : age,
school : "清华大学")       ②
        self.name = "Tom"
③

  }
}
```

上述代码第①行定义了便利构造器`init (name : String, age : Int)`，其中第②行代码是调用同一类中的其他构造器代理，首先调用语句`self.init(name : name, age : age,`

school : "清华大学"），然后再调用第③行的self.name = "Tom"语句为school属性赋值。

4. **安全检查4**

构造器在第一阶段构造完成之前，不能调用实例方法，也不能读取实例属性。

15.2.3 构造器继承

Swift中的子类构造器的来源有两种：自己编写和从父类继承。并不是父类的所有的构造器都能自动继承下来，能够从父类自动继承下来的构造器是有条件的，如下所示。 条件1：如果子类没有定义任何指定构造器，它将自动继承所有父类的指定构造器。 条件2：如果子类提供了所有父类指定构造器的实现，无论是通过条件1继承过来的，还是通过自己编写实现的，它都将自动继承所有父类的便利构造器。

下面看示例代码：

```
class Person {
①
    var name : String
    var age : Int

    func description() -> String {
```

```
        return "\(name) 年龄是: \(age)"
    }
    convenience init () {
②
        self.init(name: "Tony")
        self.age = 18
    }
    convenience init (name : String) {
③
        self.init(name: name, age: 18)
    }
    init (name : String, age : Int) {
④
        self.name = name
        self.age = age
    }
}

class Student : Person {
⑤
    var school : String
    init (name : String, age : Int, school :
String) {                ⑥
        self.school = school
        super.init(name : name, age : age)
    }
    convenience override init (name : String,
age : Int) {             ⑦
        self.init(name : name, age : age,
school : "清华大学")
```

```
    }
}

class Graduate : Student {
⑧
    var special : String = ""
}
```

上述代码第①行、第⑤行和第⑧行分别定义了Person、Student和Graduate，它们是继承关系。

我们先看看符合条件1的继承，Graduate继承Student，Graduate类没有定义任何指定构造器，它将自动继承所有Student的指定构造器，如图15-3所示，符合条件1后，Graduate从Student继承了如下指定构造器：

```
init (name : String, age : Int, school : String)
```

我们再看符合条件2的继承，由于Graduate实现了Student的所有指定构造器，Graduate将自

动继承所有Student的便利构造器。如图15-3所示，符合条件2后，Graduate从Student继承了如下3个便利构造器：

```
init (name : String, age : Int)
init (name : String)
init ()
```

如图15-3所示，Student继承Person后有4个构造器。

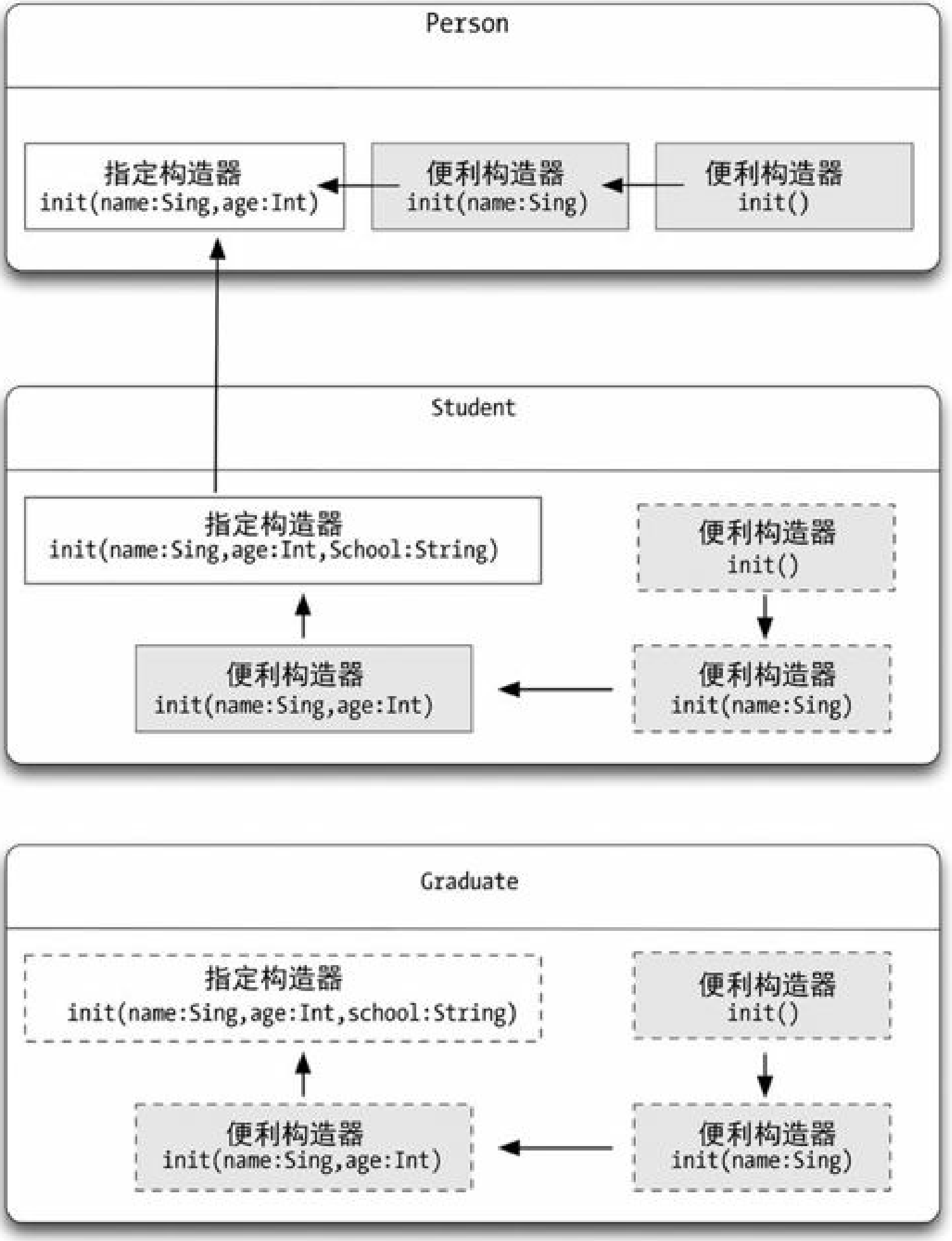

Person
指定构造器
init(name:Sing,age:Int)
便利构造器
init(name:Sing)
便利构造器
init()
Student
指定构造器
init(name:Sing,age:Int,School:String)
便利构造器
init()
便利构造器
init(name:Sing,age:Int)
便利构造器
init(name:Sing)
Graduate
指定构造器
init(name:Sing,age:Int,school:String)
便利构造器
init()
便利构造器
init(name:Sing,age:Int)
便利构造器
init(name:Sing)

图 15-3　构造器自动继承（虚线表示继承过来的构造器）

条件1对Student不起作用，因为它有指定构造器，Student类中的便利构造器init (name : String, age : Int)满足了条件2（见代码第⑦行），它实现了父类指定构造器init (name : String, age : Int)（见代码第④行），由于子类构造器与父类构造器参数相同，需要使用override关键字，表示子类构造器重写（overriding）了父类构造器。

由于Student类实现了父类指定构造器，因此也继承了父类的另外两个便利构造器（见代码第②行和第③行）。

15.3 重写

一个类继承另一个类的属性、方法、下标等特征后，子类可以重写（overriding）这些特征，overriding也有人翻译为“覆盖”，为了统一名称本书全部翻译为“重写”。下面我们就逐一介绍这些特征的重写。

15.3.1 属性重写

我们可以在子类中重写从父类继承来的属性，属性的重写一方面可以重写getter和setter访问器，另一方面可以重写属性观察者。

通过对第12章属性的学习，我们知道，计算类型属性需要使用getter和setter访问器，而存储属性不需要。子类在继承父类后，也可以通过getter和setter访问器重写父类的存储属性和计算属性。

下面看一个示例：

```
class Person {

    var name : String
①
    var age : Int
②

    func description() -> String {
```

```
        return "\(name) 年龄是: \(age)"
    }

    init (name : String, age : Int) {
        self.name = name
        self.age  = age
    }
}

class Student : Person {

    var school : String
③

    override var age : Int {
④
    get {
        return super.age
⑤
    }
    set {
        super.age = newValue < 8 ? 8 : newValue
⑥
    }
    }
⑦

    convenience init() {
        self.init(name : "Tony", age : 18,
school : "清华大学")
```

```
    }

    init (name : String, age : Int, school :
String) {
        self.school = school
        super.init(name : name, age : age)
    }
}

let student1 = Student()
println("学生年龄：\(student1.age)")
Student1.age = 6
println("学生年龄：\(student1.age)")
```

上述代码第①行在Person类中定义存储name属性，第②行定义存储age属性。然后在Person的子类Student中重写age属性，其中第④~⑦行是重写代码，重写属性前面要添加override关键字，见代码第④行。在getter方法器中，第⑤行代码返回super.age，super指代Person类实例，super.age是直接访问父类的age属性。在setter访问器中，第⑥行代码super.age = newValue < 8 ? 8 : newValue，是比较新值是否小于8岁（8岁为上学年龄），如果小于8

岁，把8赋值给父类的age属性，否则把新值赋值给父类的age属性。

从属性重写可见，子类本身并不存储数据，数据是存储在父类的存储属性中的。

以上示例是重写属性getter和setter访问器，我们还可以重写属性观察者，代码如下：

```
class Person {

    var name : String
    var age : Int

    func description() -> String {
        return "\(name) 年龄是: \(age)"
    }

    init (name : String, age : Int) {
        self.name = name
        self.age  = age
    }
}

class Student : Person {

    var school : String

    override var age : Int {
```

```
①
    willSet {
②
        println("学生年龄新值：\(newValue)")
③

    }
    didSet{
④
        println("学生年龄旧值：\(oldValue)")
⑤
    }
    }
⑥

    convenience init() {
        self.init(name : "Tony", age : 18,
school : "清华大学")
    }

    init (name : String, age : Int, school :
String) {
        self.school = school
        super.init(name : name, age : age)
    }
}

let student1 = Student()
println("学生年龄：\(student1.age)")
Student1.age = 6
```

```
⑦
println("学生年龄：\(student1.age)")
```

上述代码第①~⑥行重写了age属性观察者。重写属性前面要添加override关键字，见代码第①行。如果只关注修改之前的调用，可以只重写willSet观察者；如果只关注修改之后的调用，可以只重写didSet观察者，总之是比较灵活的。在观察者中，还可以使用系统分配默认参数newValue和oldValue。

代码第⑦行修改了age属性，修改前后的输出结果如下：

```
学生年龄新值：6
学生年龄旧值：18
```

提示 一个属性重写了观察者后，就不能同时对getter和setter访问器重写。另外，常量属性和只读计算属性也都不能重写属性观察者。

15.3.2 方法重写

我们可以在子类中重写从父类继承来的实例方法和静态方法（又称为类方法）。

下面看一个示例：

```
class Person {

    var name : String
    var age : Int

    func description() -> String {
①
        return "\(name) 年龄是: \(age)"
    }

    class func printlnClass() ->() {
②
        println( "Person 打印...")
    }

    init (name : String, age : Int) {
        self.name = name
        self.age  = age
    }
}

class Student : Person {
```

```
    var school : String

    convenience init() {
        self.init(name : "Tony", age : 18,
school : "清华大学")
    }

    init (name : String, age : Int, school :
String) {
        self.school = school
        super.init(name : name, age : age)
    }

    override func description() -> String {
③
         println("父类打印 \
(super.description())")                    ④
         return "\(name) 年龄是: \(age), 所在学
校: \(school)。"
    }

    override class func printlnClass() ->() {
⑤
        println( "Student 打印...")
    }
}

let student1 = student()
println("学生1：\(student1.description())")
⑥
```

```
Person.printlnClass()
⑦
Student.printlnClass()
⑧
```

在Person类中，第①行代码是定义实例方法description，第②行代码是定义静态方法printlnClass，然后在Person类的子类Student类中重写description和printlnClass方法，代码第③行是重写实例方法description，重写的方法前面要添加关键字override。第④行代码使用super.description()语句调用父类的description方法，其中super指代父类实例。

第⑤行代码是重写静态方法printlnClass，在静态方法中不能访问实例属性。

最后第⑥行调用了description方法。由于在子类中重写了该方法，所以调用的是子类中的description方法。输出结果是：

```
父类打印 Tony 年龄是: 18
```

```
学生1：Tony 年龄是：18，所在学校：清华大学。
```

为了测试静态方法重写，第⑦行调用了Person.printlnClass()语言，它是调用父类的printlnClass静态方法，输出结果是：

```
Person 打印...
```

第⑧行调用了Student.printlnClass()语言，它是调用子类的printlnClass静态方法，输出结果是：

```
Student 打印...
```

15.3.3 下标重写

下标是一种特殊属性。子类属性重写是重写属性的getter和setter访问器，对下标的重写也是重写下标的getter和setter访问器。

下面看一个示例：

```
class DoubleDimensionalArray {
①

    let rows: Int, columns: Int
    var grid: [Int]

    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns,
repeatedValue: 0)
    }

    subscript(row: Int, col: Int) -> Int {
②

        get {
            return grid[(row  columns) + col]
        }

        set {
            grid[(row  columns) + col] =
newValue
        }
    }
③

}
```

```
class SquareMatrix : DoubleDimensionalArray {
④

    override subscript(row: Int, col: Int) ->
Int {                          ⑤

        get {
⑥
            return super.grid[(row  columns) +
col]                           ⑦
        }

        set {
⑧
            super.grid[(row  columns) + col] =
newValue * newValue            ⑨
        }
⑩
    }
}

var ary2 = SquareMatrix(rows: 5, columns: 5)

for var i = 0; i < 5; i++ {
    for var j = 0; j < 5; j++ {
        ary2[i,j] = i + j
    }
}

for var i = 0; i < 5; i++ {
```

```
    for var j = 0; j < 5; j++ {
        print("\t\t \(ary2[i,j])")
    }
    print("\n")
}
```

上述代码第①行定义了类DoubleDimensionalArray，它在代码第②行和第③行定义了下标。第④行代码定义了类SquareMatrix，它继承了DoubleDimensionalArray类，并且在第④~⑩行重写了父类的下标。与其他类的重写类似，前面需要添加关键字override，见代码第⑤行。第⑥行是重写getter访问器，其中的第⑦行super.grid[(row * columns) + col]语句中使用super调用父类的grid属性。第⑧行代码是重写setter访问器，其中的第⑨行super.grid[(row *columns) + col] = newValue* newValue语句是给父类的grid属性赋值。

15.3.4 使用final关键字

我们可以在类的定义中使用final关键字声明类、属性、方法和下标。final声明的类不能被继承，final声明的属性、方法和下标不能被重写。

下面看一个示例：

```
final class Person {
①

    var name : String

    final var age : Int
②

    final func description() -> String {
③
        return "\(name) 年龄是: \(age)"
    }

    final class func printlnClass() ->() {
④
        println( "Person 打印...")
    }

    init (name : String, age : Int) {
        self.name = name
        self.age  = age
    }
}
```

```
class Student : Person {
//编译错误          ⑤

    var school : String

    convenience init() {
        self.init(name : "Tony", age : 18,
school : "清华大学")
    }

    init (name : String, age : Int, school :
String) {
        self.school = school
        super.init(name : name, age : age)
    }

    override func description() -> String {
//编译错误          ⑥
        println("父类打印 \
(super.description())")
        return "\(name) 年龄是: \(age), 所在学校:
\(school)。"
    }

    override class func printlnClass() ->() {
//编译错误          ⑦
        println( "Student 打印...")
    }
```

```
    override var age : Int {
//编译错误          ⑧
    get {
        return super.age
    }
    set {
        super.age = newValue < 8 ? 8 : newValue
    }
    }
}
```

上述代码第①行定义Person类，它被声明为final，说明它是不能被继承的，因此代码第⑤行定义Student类，并声明为Person子类时，会报如下编译错误：

```
Inheritance from a final class 'Person'
```

第②行定义的age属性也是final，那么在代码第⑧行试图重写age属性时，会报如下编译错误：

```
Var overrides a 'final' var
```

第③行定义description实例方法，它被声明为final，那么在代码第⑥行试图重写description实例方法时，会报如下编译错误：

```
Instance method overrides a 'final' instance method
```

第④行定义printlnClass静态方法，它被声明为final，那么在代码第⑦行试图重写printlnClass静态方法时，会报如下编译错误：

```
Class method overrides a 'final' class method
```

使用final可以控制我们的类被有限地继承，特别是在开发一些商业软件时，适当地添加final限制是非常有必要的。

15.4 类型检查与转换

继承会发生在子类和父类中，如图15-4所示，是一系列类的继承关系类图，Person是类层次结构中的根类，Student是Person的直接子类，Worker是Person的直接子类。

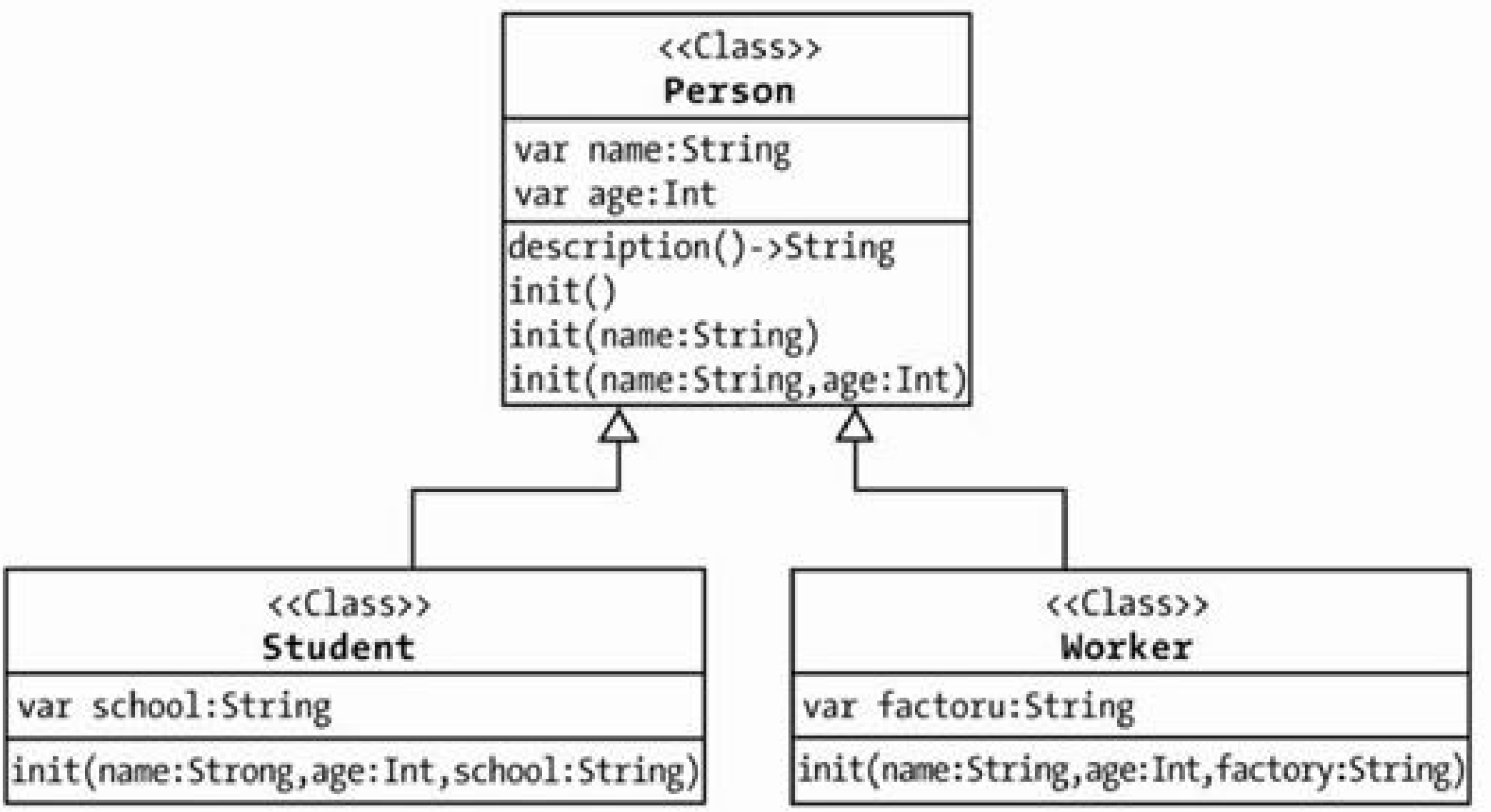


图 15-4 继承关系类图

这个继承关系类图的具体实现代码如下：

```
class Person {
    var name : String
    var age : Int

    func description() -> String {
```

```
        return "\(name) 年龄是: \(age)"
    }
    convenience init () {
        self.init(name: "Tony")
        self.age = 18
    }
    convenience init (name : String) {
        self.init(name: name, age: 18)
    }
    init (name : String, age : Int) {
        self.name = name
        self.age  = age
    }
}

class Student : Person {
    var school : String
    init (name : String, age : Int, school :
String) {
        self.school = school
        super.init(name : name, age : age)
    }
}

class Worker : Person {
    var factory : String
    init (name : String, age : Int, factory :
String) {
        self.factory = factory
        super.init(name : name, age : age)
```

```
    }
}
```

下面我们将以此为例，介绍Swift类的类型检查与转换，其中包括is操作符、as操作符以及Any和AnyObject类型等。

15.4.1 使用is操作符

is操作符可以判断一个实例是否是某个类的类型。如果实例是目标类型，结果返回true，否则为false。

下面看一个示例：

```
let student1 = Student(name : "Tom", age : 18,
school : "清华大学")        ①
let student2 = Student(name : "Ben", age : 28,
school : "北京大学")
let student3 = Student(name : "Tony", age : 38,
school : "香港大学")        ②

let worker1 = Worker(name : "Tom", age : 18,
factory : "钢厂")                ③
let worker2 = Worker(name : "Ben", age : 20,
factory : "电厂")                ④
```

```
let people = [student1, student2, student3,
worker1, worker2]                    ⑤

var studentCount = 0
var workerCount = 0

for item in people {
⑥
    if item is Worker {
⑦
        ++workerCount
    } else if item is Student {
⑧
        ++studentCount
    }
}

println("工人人数：\(workerCount) ，学生人数：\
(studentCount) 。")
```

上述代码第①行和第②行创建了3个Student实例，第③行和第④行创建了两个Worker实例，然后把这5个实例放入people数组集合中。

在第⑥行使用for in遍历people数组集合。在循环体中，第⑦行item is Worker表达式是

判断集合中的元素是否是Worker类的实例。类似地，第⑧行item is Student表达式是判断集合中的元素是否是Student类的实例。

输出结果如下：

```
工人人数：2，学生人数：3。
```

15.4.2 使用as操作符

在介绍as操作符之前，我们先了解一下对象的类型转换，并不是所有的类型都能互相转换。下面先看如下语句：

```
let p1 : Person = Student(name : "Tom", age :
20, school : "清华大学")
let p2 : Person = Worker(name : "Tom", age :
18, factory : "钢厂")
let p3 : Person = Person(name : "Tom", age :
28)
```

我们创建了3个实例p1、p2、p3，类型都是Person。p1是Student实例，p2是Worker实

例，p3是Person实例。首先，对象类型转换一定发生在继承的前提下，p1和p2都声明为Person类型，而实例是由Person子类型实例化的。

表15-1归纳了p1、p2和p3这3个实例与Worker、Student和Person这3种类型之间的转换关系。

表15-1　类型转换

对象	Person类型	Worker类型	Student类型	说明
p1	支持	不支持	支持（向下转型）	类型：Perso 实例：Stude
p2	支持	支持（向下转型）	不支持	类型：Perso 实例：Worke
p3	支持	不支持	不支持	类型：Perso 实

				例：Perso

作为这段程序的编写者，我们知道p1本质上是Student实例，但是表面上看是Person类型，编译器也无法推断p1的实例是Person、Student还是Worker。我们可以使用is操作符来判断它是哪一类的实例。然后在转换时可以使用as操作符将其转换为子类类型，即把Person类型的p1转为Student子类类型，这种转换被称为**向下转型**。这种转换是有风险的，如果p1不是目标类型，转换就会失败。为了不发生异常，我们可以使用as?将其转换为目标类型的可选类型，能够成功则转换，不成功则返回nil。

从表15-1可见，p1到Student类型转换是向下转型，能够成功，p1到Worker类型转换是向下转型但会失败，p1到Person类型转换不需要向下转型就能够成功赋值，因为p1本身就是Person类型。p2与p1类似。

p3与p1和p2有很大的不同，因为p3本质上是Person实例，不能向下转型。

下面看一个示例：

```
let student1 = Student(name : "Tom", age : 18,
school : "清华大学")          ①
let student2 = Student(name : "Ben", age : 28,
school : "北京大学")
let student3 = Student(name : "Tony", age : 38,
school : "香港大学")          ②

let worker1 = Worker(name : "Tom", age : 18,
factory : "钢厂")                ③
let worker2 = Worker(name : "Ben", age : 20,
factory : "电厂")                ④

let people = [student1, student2, student3,
worker1, worker2]                    ⑤

for item in people {
⑥

    if let student = item as? Student {
⑦
        println("Student school: \
(Student.school)")                          ⑧
    } else if let worker = item as? Worker {
⑨
        println("Worker factory: \
(Worker.factory)")                          ⑩
    }
```

```
}
```

上述代码第①行和第②行创建了3个Student实例，第③行和第④行创建了两个Worker实例。然后把这5个实例放入people数组集合中。

在第⑥行使用for in遍历people数组集合。在循环体中，第⑦行let student = item as? Student语句使用as?操作符将元素转换为Student类型。如果转换成功，则把元素赋值给Student变量，否则将nil赋值给Student变量，转换成功执行第⑧行代码。第⑨行代码与第⑦行代码类似，不再赘述。

最后输出结果如下：

```
Student school: 清华大学
Student school: 北京大学
Student school: 香港大学
Worker factory: 钢厂
Worker factory: 电厂
```

15.4.3 使用Any和AnyObject类型

在Swift中还提供了两种类型表示不确定类型：AnyObject和Any。AnyObject可以表示任何类的实例，而Any可以表示任何类型，包括类和其他数据类型，也包括Int和Double的基本数据类型。

下面将上一节的示例修改如下：

```
let student1 = Student(name : "Tom", age : 18,
school : "清华大学")
let student2 = Student(name : "Ben", age : 28,
school : "北京大学")
let student3 = Student(name : "Tony", age : 38,
school : "香港大学")

let worker1 = Worker(name : "Tom", age : 18,
factory : "钢厂")
let worker2 = Worker(name : "Ben", age : 20,
factory : "电厂")

let people1: [Person] = [student1, student2,
student3, worker1, worker2]          ①
let people2: [AnyObject] = [student1, student2,
student3, worker1, worker2]       ②
let people3: [Any] = [student1, student2,
student3, worker1, worker2]              ③
```

```
for item in people3 {
④

    if let Student = item as? Student {
        println("Student school: \
(Student.school)")
    } else if let Worker = item as? Worker {
        println("Worker factory: \
(Worker.factory)")
    }

}
```

上述代码第①行是将5个实例放入Person数组中，第②行代码是将5个实例放入AnyObject数组中，第③行代码是将5个实例放入Any数组中。

这3种类型的数组都可以成功放入5个实例，而且可以在第④行使用for Int循环遍历出来，其他的类型代码不再解释。

15.5　本章小结

通过对本章内容的学习，我们掌握了Swift语言的继承性，了解了Swift中的继承只能发生在类类型上，而枚举和结构体不能发生继承，熟悉了子类继承父类的方法、属性、下标等特征的过程，还学习了子类如何重写父类的方法、属性、下标等特征。

15.6 同步练习

1. 请描述两段式构造过程以及下列代码B类的构造过程。

```
class A {
    var x: Int
    init(x: Int) {
        self.x = x
    }
}

class B: A {
    var y: Int
    init(x: Int, y: Int) {
        self.y = y
        super.init(x: x)
        self.x = self.generate()
    }

    func generate() -> Int {
        /* return some value */
        return 0
    }
}
```

2. 简述Swift在构造过程中的4种安全检查。

3. 判断正误：一个属性重写了观察者后，就不能同时对getter和setter访问器重写。

4. 判断正误：常量属性和只读计算属性都不能重写属性观察者。

5. 判断正误：我们可以重写父类的实例方法和静态方法（或称为类方法）。

6. 判断正误：final关键字声明类、属性、方法和下标。final声明的类不能被继承，final声明的属性、方法和下标不能被重写。

7. 判断正误：is操作符可以判断一个实例是否是某个类的类型。如果实例是目标类型结果返回true，否则为false。

8. 判断正误：使用as操作符转换为子类类型，这种转换被称为向下转型。

9. 判断正误：AnyObject可以表示任何类的实例，而Any可以表示任何类型。

10. 给定如下基类A：

```
class A {
    var x: Int
    init(x: Int) {
        self.x = x
    }
```

```
}
```

则关于子类B的定义正确的是（　　）。

A.

```
class B: A {
    var y: Int
    init(x: Int, y: Int) {
        self.y = y
        super.init(x: x)
        self.x = self.generate()
    }

    func generate() -> Int {
        /* return some value */
        return 0
    }
}
```

B.

```
class B: A {
    var y: Int
    init(x: Int, y: Int) {
        super.init(x: x)
```

```
        self.y = y
        self.x = self.generate()
    }

    func generate() -> Int {
        /* return some value */
        return 0
    }
}
```

C.

```
class B: A {
    var y: Int
    init(x: Int, y: Int) {
        self.y = y
        super.init(x: x)
    }
}
```

D.

```
class B: A {
    var y: Int
    init(x: Int, y: Int) {
        super.init(x: x)
```

```
        self.y = y
    }
}
```

第 16 章　扩展和协议

在Swift中，扩展和协议是两个非常重要的概念，它们相对独立，又互相关联。本章将对这两个概念进行详细介绍。

16.1 扩展

在面向对象分析与设计方法学（OOAD）中，为了增强一个类的新功能，我们可以通过继承机制从父类继承下来一些成员，然后再根据自己的需要在子类中添加一些成员，这样我们就可以得到增强功能的新类了，但是这种方式受到了一些限制，继承过程比较繁琐，类继承性可能被禁止，有些功能也可能无法继承。

在Swift中可以使用一种扩展机制，在原有类型（类、结构体和枚举）的基础上添加新功能。扩展是一种“轻量级”的继承机制，即使原有类型被限制继承，我们仍然可以通过扩展机制“继承”原有类型的功能。

扩展机制还有另外一个优势：它扩展的类型可以是类、结构体和枚举，而继承只能是类，不能是结构体和枚举。

提示 对于扩展这种“轻量级”继承机制，只有Objective-C中的分类机制与此类似，其他面向对象的语言中均没有，因此很多Java程序员在使用Swift语言时，不擅长使用扩展机制，而是保守地使用继承机制。在设计基于Swift语言程序时，我们要优先考虑使用扩展

机制是否能够满足我们的需求，如果不能再考虑使用继承机制。

16.1.1 声明扩展

声明扩展的语法格式如下：

```
extension 类型名 {
    //添加新功能
}
```

声明扩展的关键字是extension，“类型名”是Swift中已有的类型，包括类、结构体和枚举，但是我们仍然可以扩展整型、浮点型、布尔型、字符串等基本数据类型，这是因为这些类型本质上也是结构体类型。打开Int的定义如下：

```
struct Int : SignedInteger {
    init()
    init(_ value: Int)
    static func
convertFromIntegerLiteral(value: Int) -> Int
    typealias ArrayBoundType = Int
    func getArrayBoundValue() -> Int
    static var max: Int { get }
```

```
    static var min: Int { get }
}
```

从定义可见Int是结构体类型。不仅是Int类型，我们熟悉的整型、浮点型、布尔型、字符串等数据类型本质上都是结构体类型。

具体而言，Swift中的扩展机制可以在原类型中添加的新功能包括：

- 实例计算属性和静态计算属性
- 实例方法和静态方法
- 构造器
- 下标

此外，还有嵌套类型等内容也可以扩展。下面我们将重点介绍扩展计算属性、扩展方法、扩展构造器和扩展下标。

16.1.2 扩展计算属性

我们可以在原类型上扩展计算属性，包括实例计算属性和静态计算属性。这些添加计算属性的定义，与普通的计算属性的定义是一样的。

下面先看一个实例计算属性示例。我们在网络

编程的时候，为了减少流量，从服务器端返回的不是错误信息描述，而是错误编码，然后在本地再将错误编码转换为错误描述信息。为此我们定义了如下Int类型扩展：

```
extension Int {
①
    var errorMessage : String {
②
        var errorStr = ""
        switch (self) {
③
        case -7:
            errorStr = "没有数据。"
        case -6:
            errorStr = "日期没有输入。"
        case -5:
            errorStr = "内容没有输入。"
        case -4:
            errorStr = "ID没有输入。"
        case -3:
            errorStr = "据访问失败。"
        case -2:
            errorStr = "您的账号最多能插入10条数据。"
        case -1:
            errorStr = "用户不存在，请到http://iosbook3.com注册。"
```

```
        default:
            errorStr = ""
        }
④
        return errorStr
    }
}
let message = (-7).errorMessage
⑤
println("Error Code : -7 , Error Message : \
(message)")                          ⑥
```

上述代码第①行定义Int类型的扩展，第②行代码定义只读计算属性errorMessage，第③行和第④行代码是switch分支语言，switch表达式是self，即当前实例，然后通过switch的case判断是哪个分支，并返回错误描述信息。我们在扩展中经常使用self获得当前实例。

第⑤行代码(-7).errorMessage是获得-7编码对应的错误描述信息。注意整个-7包括负号是一个完整的实例，因此调用它的属性时需要将-7作为一个整体用小括号括起来。然而，如果是7则不需要括号。

下面再看一个静态属性的示例：

```
struct Account {
①
    var amount : Double = 0.0                    //
账户金额
    var owner : String = ""                      //
账户名
}

extension Account {
②
    static var interestRate : Double {           //
利率                                ③
        return 0.668
    }
}

println(Account.interestRate)
④
```

上述代码第①行是定义Account结构体，第②行代码是定义Account结构体的扩展类型，其中第③行代码是定义静态只读计算属性interestRate。interestRate是利率，对于所有账户都是一样的，所以它被定义为静态属性。

第④行代码是打印输出interestRate属性，访问方式与其他的静态计算属性一样，通过“类型名”加“.”来访问静态计算属性。

此外，在扩展中不仅可以定义只读计算属性，还可以定义读写计算属性、实例计算属性和静态计算属性。它们的定义方式与在原类型中定义的是一样的，这里不再赘述。

16.1.3 扩展方法

我们可以在原类型上扩展方法，包括实例方法和静态方法。这些添加方法的定义与普通方法的定义是一样的。

下面先看一个示例：

```
extension Double {
    static var interestRate : Double = 0.668 //
利率
    func interestBy1() -> Double {
①
        return self * Double.interestRate
②
    }
    mutating func interestBy2() {
③
        self = self  Double.interestRate
```

```
④
    }
    static func interestBy3(amount : Double) ->
Double {        ⑤
        return interestRate  amount
⑥
    }
}

let interest1 = (10_000.00).interestBy1()
⑦
println("利息1 : \(interest1)")

var interest2 = 10_000.00
⑧
interest2.interestBy2()
⑨
println("利息2 : \(interest2)")

var interest3 = Double.interestBy3(10_000.00)
⑩
println("利息3 : \(interest3)")
```

上述代码定义Double类型的扩展，其中第①行代码是定义实例方法interestBy1，该方法第②行代码self * Double.interestRate是计算利息，其中self是当前实

例，Double.interestRate是静态属性利率。

第③行代码是定义实例方法interestBy2，它也可以计算利息，但是没有返回值，而是通过第④行代码self = self * Double.interestRate，把计算结果直接赋值给当前实例self。在结构体和枚举类型中给self赋值会有编译错误，需要在方法前面加上mutating关键字，表明这是变异方法。

第⑤行代码是定义静态方法interestBy3，它也可以计算利息，参数有返回值，参数是计算利息的金额，第⑥行代码是返回值是计算利息结果。

这3个方法在调用时是不同的，第⑦行代码是调用interestBy1方法计算利息，调用它的实例10_000.00，它的返回值被赋值给interest1常量，这是很常见的调用过程。

第⑧行代码是调用interestBy2方法计算利息，我们不能使用10_000.00实例调用，而是需要一个Double类型的变量interest2。interestBy2是变异方法，它会直接改变变量interest2的值，因此第⑨行的interest2.interestBy2()语句调用完成后，

变量interest2的值就改变了。

第⑩行代码是调用interestBy3方法计算利息，它是静态方法，调用它需要以“类型名.”的方式即“Double.”的方式调用。

16.1.4 扩展构造器

扩展类型的时候，也可以添加新的构造器。值类型与引用类型扩展有所区别。值类型包括了除类以外的其他类型，主要是枚举类型和结构体类型。

下列代码是扩展结构体类型中定义构造器的示例：

```
struct Rectangle {
①

    var width : Double
    var height : Double

    init(width : Double, height : Double) {
        self.width   = width
        self.height  = height
    }

}
extension Rectangle {
```

```
②
    init(length : Double) {
③
        self.init(width : length, height :
length)          ④
    }
}

var rect = Rectangle(width : 320.0, height :
480.0)          ⑤
println("长方形:\(rect.width) x \(rect.height)")

var square = Rectangle(length: 500.0)
⑥
println("正方形:\(square.width) x \
(square.height)")
```

上述代码第①行是定义结构体Rectangle，然后在第②行定义了Rectangle的扩展类型。其中第③行定义构造器init(length : Double)，只有一个参数。然后在第④行调用self.init(width : length, height : length)语句，self.init是调用了原类型的两个参数的构造器。

第⑤行代码调用两个参数的构造器创建

Rectangle实例，这个构造器是原类型提供的，这时候的Rectangle类型已经是第②行定义的扩展类型了。

第⑥行代码调用一个参数的构造器创建Rectangle实例，这个构造器是扩展类型提供的。

下面我们讨论一下引用类型扩展中定义构造器。引用类型只包含一个类型，即类类型。在类中，由于考虑到继承问题，类中构造器分为指定构造器和便利构造器。扩展类的时候能向类中添加新的便利构造器，但不能添加新的指定构造器或析构器。指定构造器和析构器只能由原类型提供。

下列代码是扩展类中定义构造器的示例：

```
class Person {
①
    var name : String
    var age : Int
    func description() -> String {
        return "\(name) 年龄是: \(age)"
    }
    init (name : String, age : Int) {
②
        self.name = name
        self.age  = age
```

```
    }
}

extension Person {
③
    convenience init (name : String) {
④
        self.init(name : name, age : 8)
⑤
    }
}

let p1 = Person(name : "Mary")
⑥
println("Person1 : \(p1.description())")
let p2 = Person(name : "Tony", age : 28)
⑦
println("Person2 : \(p2.description())")
```

第①行代码是定义类Person，其中代码第②行提供了两个参数的构造器。第③行代码是定义Person类的扩展类型，其中代码第④行提供了一个参数的构造器，它是便利构造器。在这个构造器中，第⑤行代码self.init(name : name, age : 8)调用指定构造器代理部分构造任务。

第⑥行代码调用两个参数的构造器创建Person

实例，这个构造器是原类型提供的，这时候的Person类型已经是第②行定义的扩展类型了。

第⑦行代码调用一个参数的构造器创建Person实例，这个构造器是扩展类型提供的。

16.1.5 扩展下标

我们可以把下标认为是特殊的属性，可以实现索引访问属性。我们可以在原有类型的基础上扩展下标功能。

字符串本身没有提供按照下标访问字符的功能。下面我们扩展字符串，实现下标访问字符功能，代码如下：

```
extension String {

    subscript(index : Int) ->String {
①

        if  index > countElements(self) {
②
            return ""
        }
        var c : String = ""
        var i = 0
```

```
        for character in self {
③
            if (i == index) {
④
                c = String(character)
⑤
                break
⑥
            }
            i++
        }
        return c
    }
}

let s = "The quick brown fox jumps over the
lazy dog"       ⑦
println(s[0])
⑧

println("ABC"[2])
⑨
```

上述代码是扩展字符串String类型，添加一个下标。第①行定义下标，Int类型参数是下标索引，返回值是String类型，是要访问的字符。第②行代码是判断下标是否越界，如果越界则返回空

字符串。第③行代码是使用for in循环遍历字符串，第④行代码判断当前循环变量i是否等于index参数，如果相等，则通过第⑤行代码c = String(character)将字符character赋值字符串c变量。直接采用c = character语句赋值会发生错误，这是因为c是字符串类型，character是字符类型。

第⑦行代码声明并初始化字符串常量s，使用经典英语全字母句"The quick brown fox jumps over the lazy dog"[1]来初始化s常量。第⑧行代码s[0]是通过下标访问，结果输出为“T”。第⑨行代码"ABC"[2]是通过下标访问"ABC"字符串的内容，结果输出为“C”。

[1] “The quick brown fox jumps over the lazy dog”（中译为“敏捷的棕毛狐狸从懒狗身上跃过”）是一个著名的英语全字母句，常被用于测试字体的显示效果和键盘有没有故障。此句也常以“quick brown fox”作为指代简称。——引自于维基百科（http://zh.wikipedia.org/wiki/The_quick_brown_fox_jumps_o

16.2 协议

在面向对象分析与设计方法学（OOAD）中，可能会有这样的经历：一些类的方法所执行的内容是无法确定的，只能等到它的子类中才能确定下来。例如，几何图形类可以有绘制图形的方法，但是绘制图形方法的具体内容无法确定，这是因为我们不知道绘制的是什么样的几何图形。如图16-1所示，子类矩形有自己的绘制方法，子类圆形也有自己的绘制方法。

几何图形这种类在面向对象分析与设计方法学中称为**抽象类**，方法称为**抽象方法**。矩形和圆形是几何图形的子类，它们实现了几何图形的绘制图形的抽象方法。

如果几何图形类中所有的方法都是抽象的，那么在Swift和Objective-C中称为协议（protocol），在Java语言中称为接口，在C++中是纯虚类。

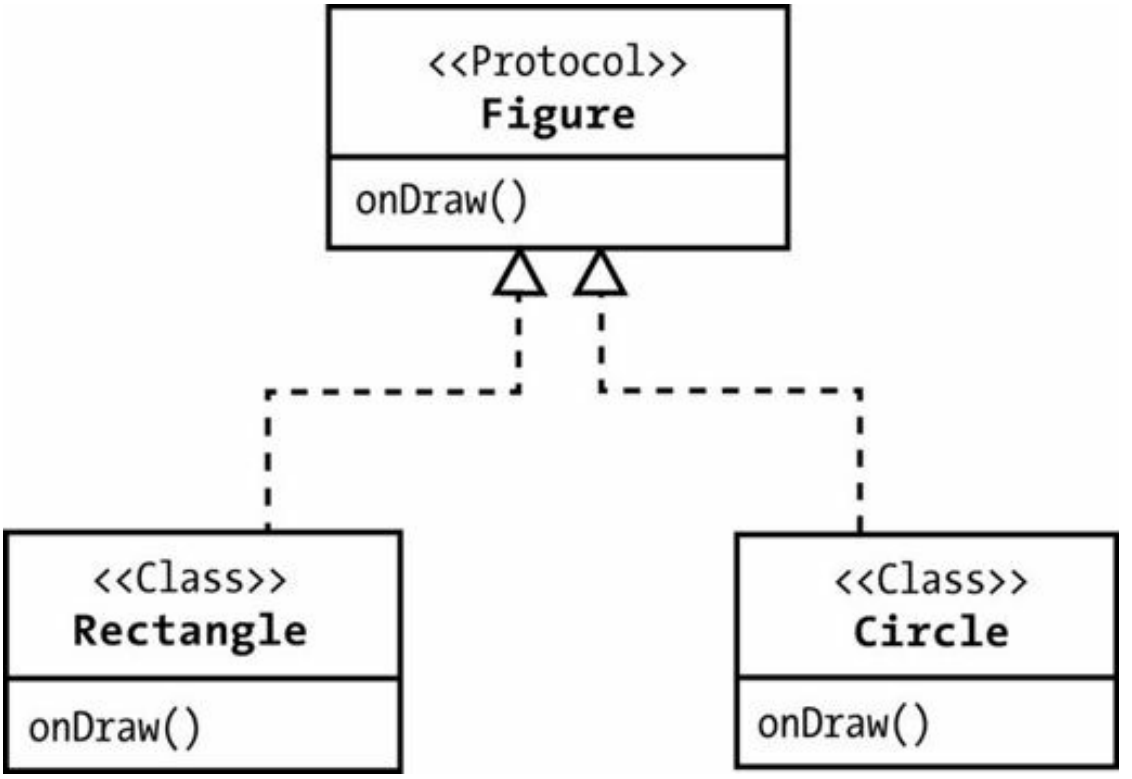


图 16-1　几何图形类图

也就是说，协议是高度抽象的，它只规定绘制图形的抽象方法名（`onDraw`）、参数列表和返回值等信息，不给出具体的实现。这种抽象方法由遵守该协议的“遵守者”具体实现的过程在Swift和Objective-C中称为**遵守协议**，在Java中称为**实现接口**。

16.2.1　声明和遵守协议

在Swift中，类、结构体和枚举类型可以声明遵守某个协议，并提供该协议所要求的属性和方法。

协议定义语法如下所示：

```
protocol 协议名 {
```

```
    // 协议内容
}
```

在声明遵守协议时，语法如下所示：

```
类型 类型名 ： 协议1, 协议2 {
    // 遵守协议内容
}
```

其中类型包括class、struct和enmu，类型名是我们自己定义的，冒号“:”后是需要遵守的协议。当要遵守多个协议时，各协议之间用逗号“,”隔开。

如果一个类继承父类的同时也要遵守协议，应当把父类放在所有的协议之前，如下所示：

```
class 类名 : 父类, 协议1, 协议2 {
    // 遵守协议内容
}
```

只有类的定义会有父类和协议混合声明，结构体和枚举是没有父类型的。

具有而言，协议可以要求其遵守者提供实例属性、静态属性、实例方法和静态方法等内容的实现。下面我们重点介绍一下对方法和属性的要求。

16.2.2 协议方法

协议可以要求其遵守者实现某些指定方法，包括实例方法和静态方法。这些方法在协议中被定义，协议方法与普通方法类似，但不支持变长参数和默认值参数，也不需要大括号和方法体。

1. **实例协议方法**

下面先看看实例协议方法定义与实现。以下是示例代码：

```
protocol Figure {
①
    func onDraw() //定义抽象绘制几何图形
②
}
class Rectangle : Figure {
③
    func onDraw() {
        println("绘制矩形...")
    }
}
class Circle : Figure {
④
```

```
    func onDraw() {
        println("绘制圆形...")
    }
}

let rect : Figure = Rectangle()
⑤
rect.onDraw()
⑥

let circle : Figure = Circle()
⑦
circle.onDraw()
⑧
```

上述代码第①行定义了协议Figure，其中代码第②行定义抽象绘制几何图形onDraw方法。从代码中可见，只有方法的声明没有具体实现（没有大括号和方法体）。第③行是定义类Rectangle，它是Figure协议的遵守者，它具体实现了Figure协议规定的onDraw方法，当然它的实现也很简单，只是打印一个字符串"绘制矩形..."。

第④行是定义类Circle，它也是Figure协议的遵守者，它也具体实现了Figure协议规定的

onDraw方法，当然它的实现也很简单，只是打印一个字符串"绘制圆形..."。

第⑤行代码创建Rectangle实例，但是声明类型为Figure，我们可以把协议作为类型使用，rect即便是Figure类型，本质上还是Rectangle实例，所以在第⑥行调用onDraw方法的时候，输出结果是"绘制矩形..."。类似地，代码第⑦行是创建的Circle实例，第⑧行调用onDraw方法，输出"绘制圆形..."。

2. **静态协议方法**

在协议中定义静态方法与在类中定义静态方法类似，方法前面要添加class关键字。那么遵守该协议的时候，遵守者静态方法前的关键字是class还是static呢？这与遵守者类型是有关系的：如果是类，关键字就是class；如果是结构体或枚举，关键字就是static。

以下是示例代码：

```
protocol Account  {
①
    class func interestBy(amount : Double) ->
Double          ②
```

```
}

class ClassImp : Account {
③
    class func interestBy(amount : Double) ->
Double {          ④
        return 0.668  amount
    }
}

struct StructImp : Account {
⑤
    static func interestBy(amount : Double) ->
Double {          ⑥
        return 0.668  amount
    }
}

enum EnumImp : Account {
⑦
    static func interestBy(amount : Double) ->
Double {          ⑧
        return 0.668 * amount
    }
}
```

上述代码第①行是定义协议Account，第②行

是声明协议静态方法interestBy，注意需要在方法前面添加关键字class。

第③行代码是定义类ClassImp，它要求遵守Account协议，第④行具体实现静态协议方法interestBy，注意方法前面的关键字只能是class。

第⑤行代码是定义结构体StructImp，它要求遵守Account协议，第⑥行具体实现静态协议方法interestBy，注意方法前面的关键字只能是static。

第⑦行代码是定义枚举EnumImp，它要求遵守Account协议，第⑧行具体实现静态协议方法interestBy，注意方法前面的关键字只能是static。

静态协议方法定义和声明都比较麻烦，要与具体的类型有关，使用的时候需要注意。

3. 变异方法

在结构体和枚举类型中可以定义变异方法，而在类中没有这种方法。原因是结构体和枚举类型中的属性是不可以修改的，通过定义变异方法，可以在变异方法中修改这些属性。而类是引用类型，不

需要变异方法就可以修改自己的属性。

在协议定义变异方法时，方法前面要添加mutating关键字。类、结构体和枚举类型都可以实现变异方法，类实现的变异方法时，前面不需要关键字mutating；而结构体和枚举实现变异方法时，前面需要关键字mutating。

以下是示例代码：

```
protocol Editable  {
①
    mutating func edit()
②
}

class ClassImp : Editable {
③
    var name = "ClassImp"
    func edit() {
④
        println("编辑ClassImp...")
        self.name = "编辑ClassImp..."
⑤
    }
}

struct StructImp : Editable {
⑥
```

```
    var name = "StructImp"
    mutating func edit() {
⑦
        println("编辑StructImp...")
        self.name = "编辑StructImp..."
⑧
    }
}

enum EnumImp : Editable {
⑨
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday

    mutating func edit() {
⑩
        println("编辑EnumImp...")
        self = .Friday
⑪
    }
}

var classInstance : Editable = ClassImp()
classInstance.edit()

var structInstance : Editable = StructImp()
structInstance.edit()
```

```
var enumInstance : Editable = EnumImp.Monday
enumInstance.edit()
```

上述代码第①行是定义协议Editable，第②行是声明协议变异方法edit，注意方法前面添加关键字mutating。

第③行代码是定义类ClassImp，它要求遵守Editable协议。第④行具体实现变异方法edit，由于是类遵守该协议，方法前不需要添加关键字mutating。第⑤行是修改当前实例的name属性。在类中，这种修改是允许的。

第⑥行代码是定义结构体StructImp，它要求遵守Editable协议。第⑦行具体实现变异方法edit，方法前需要添加关键字mutating。第⑧行是修改当前实例的name属性，在结构体中修改属性的方法必须是变异的，我们可以尝试将关键字mutating去掉，会发生编译错误。

第⑨行代码是定义枚举EnumImp，它要求遵守Editable协议。第⑩行具体实现变异方法edit，方法前需要添加关键字mutating。第⑪行是修改

当前实例的name属性，在结构体中修改属性的方法必须是变异的，否则会发生编译错误。

最后的输出结果如下：

```
编辑ClassImp...
编辑StructImp...
编辑EnumImp...
```

16.2.3 协议属性

协议可以要求其遵守者实现某些指定属性，包括实例属性和静态属性，在具体定义的时候，每一种属性都可以有只读和读写之分。

对于遵守者而言，实现属性是非常灵活的。无论是存储属性还是计算属性，只要能满足协议属性的要求，就可以通过编译。甚至是协议中只规定了只读属性，而遵守者提供了对该属性的读写实现，这也是被允许的，因为遵守者满足了协议的只读属性要求。协议只规定了遵守者必须要做的事情，但没有规定不能做的事情。

1. **实例协议属性**

下面先看看实例协议属性定义与实现。示例代码如下：

```
protocol Person {
①
    var firstName : String { get set }
②
    var lastName : String { get set }
③
    var fullName : String { get }
④
}

class Employee : Person {
⑤
    var no : Int = 0
    var job : String?
    var salary : Double = 0

    var firstName : String = "Tony"
⑥
    var lastName : String = "Guan"
⑦

    var fullName : String {
⑧
    get {
        return self.firstName + "." +
self.lastName
    }
    set (newFullName) {
        var name =
newFullName.componentsSeparatedByString(".")
```

```
            self.firstName = name[0]
            self.lastName = name[1]
        }
        }
}
```

上述代码第①行是定义协议Person，在该协议中声明了3个属性，其中第②行和第③行属性都是可以读写的，声明时使用get和set关键字说明它是可读写的，与普通计算属性相比，getter和setter访问器没有大括号，没有具体实现。代码第④行的fullName属性是只读属性，声明时使用get关键字说明它是只读的。

第⑤行代码定义Employee类，它被要求遵守Person协议，因此需要实现Person协议所规定的3个属性。其中第⑥行代码是实现firstName属性，从定义上看，firstName是存储属性，它事实上实现了Person协议中的var firstName : String { get set }属性规定，否则我们是不能为firstName属性赋值的，也无法获得firstName属性值。第⑦行代码中的lastName属性也是类似的。

第⑧行代码的fullName属性是计算属性，它实现了Person协议中的var fullName : String { get }属性规定。计算属性fullName除了要通过定义getter访问器，实现Person协议只读属性规定外，还定义了setter访问器。Person协议对此没有规定。

2. 静态协议属性

在协议中定义静态属性与在协议中定义静态属性类似，属性前面要添加class关键字。那么在遵守协议时，遵守者静态属性前面的关键字是class还是static呢？这与遵守者类型是有关系的：如果是类，关键字就是class；如果是结构体或枚举，关键字就是static。

以下是示例代码：

```
protocol Account  {
①
    class var interestRate : Double {get}  //利率
                                        ②
    class func interestBy(amount : Double) -> Double
}

class ClassImp : Account {
```

```
③

    class var interestRate : Double  {
④
        return 0.668
    }
    class func interestBy(amount : Double) ->
Double {
        return ClassImp.interestRate * amount
    }
}

struct StructImp : Account {
⑤

    static var interestRate : Double = 0.668
⑥

    static func interestBy(amount : Double) ->
Double {
        return  StructImp.interestRate  amount
    }
}

enum EnumImp : Account {
⑦

    static var interestRate : Double = 0.668
⑧
    static func interestBy(amount : Double) ->
```

```
Double {
        return  EnumImp.interestRate  amount
    }
}
```

上述代码第①行是定义协议Account，第②行是声明协议静态属性interestRate，注意需要在属性前面添加关键字class。

第③行代码是定义类ClassImp，它要求遵守Account协议，第④行具体实现静态协议属性interestRate，注意属性前面的关键字只能是class。

第⑤行代码是定义结构体StructImp，它要求遵守Account协议，第⑥行具体实现静态协议属性interestRate，注意属性前面的关键字只能是static。

第⑦行代码是定义枚举EnumImp，它要求遵守Account协议，第⑧行具体实现静态协议属性interestRate，注意属性前面的关键字只能是static。

静态协议属性定义和声明都比较麻烦，要与具

体的类型有关，使用的时候需要注意。

16.2.4 把协议作为类型使用

虽然协议没有具体的实现代码，不能被实例化，但它的存在就是为了规范其他类型遵守它实现。在很多人看来，协议并没有什么用途，但事实上协议是非常重要的，它是面向接口编程必不可少的机制，面向接口编程系统的定义与实现应该分离。协议作为数据类型暴露给使用者，使其不用关心具体的实现细节，从而提供系统的可扩展性和可复用性。

在Swift中，协议是作为数据类型使用的，它可以出现在任意允许其他数据类型出现的地方。具体情况请看下面的示例：

```
protocol Person {
①

    var firstName : String { get set }
②
    var lastName : String { get set }
③
    var fullName : String { get }
④
```

```
    func description() -> String
⑤

}

class Student : Person {
⑥

    var school : String
    var firstName : String
    var lastName : String

    var fullName : String {
        return self.firstName + "." +
self.lastName
    }

    func description() -> String {
        return "firstName: \(firstName)
lastName: \(lastName) school: \(school)"
    }

    init (firstName : String, lastName :
String, school : String) {
        self.firstName = firstName
        self.lastName = lastName
        self.school = school
    }
}
```

```
class Worker : Person {
⑦

    var factory : String
    var firstName : String
    var lastName : String

    var fullName : String {
        return self.firstName + "." +
self.lastName
    }

    func description() -> String {
        return "firstName: \(firstName)
lastName: \(lastName) factory: \(factory)"
    }

    init (firstName : String, lastName :
String, factory : String) {
        self.firstName = firstName
        self.lastName = lastName
        self.factory = factory
    }
}

let student1 : Person = Student(firstName :
"Tom", lastName : "Guan", school : "清华大学")
⑧
let student2 : Person = Student(firstName :
"Ben", lastName : "Guan", school : "北京大学")
```

```
let student3 : Person = Student(firstName :
"Tony", lastName : "Guan", school : "香港大学")
⑨

let worker1 : Person = Worker(firstName :
"Tom", lastName : "Zhao", factory : "钢厂")
⑩
let worker2 : Person = Worker(firstName :
"Ben", lastName : "Zhao", factory : "电厂")
⑪

let people : [Person] = [student1, student2,
student3, worker1, worker2]
⑫

for item : Person in people {
⑬

    if let student = item as? Student {
⑭
        println("Student school: \
(student.school)")
        println("Student fullName: \
(student.fullName)")
        println("Student description: \
(student.description())")
    } else if let worker = item as? Worker {
⑮
        println("Worker factory: \
(worker.factory)")
```

```
        println("Worker fullName: \
(worker.fullName)")
        println("Worker description: \
(worker.description())")
    }

}
```

上述代码第①行定义了协议Person，其中第②行和第③行声明了可读写属性，第④行声明了只读属性，第⑤行声明了方法。

第⑥行定义了Student类，它遵守Person协议。类似地，第⑦行定义了Worker类，它也遵守Person协议。

代码第⑧行和第⑨行创建了3个Student实例，它们的类型是Person协议。代码第⑩行和第⑪行创建了两个Worker实例，它们的类型也是Person协议。然后在第⑫行将这5个实例放入集合people中，people是可以保存Person协议类型的数组。

第⑬行遍历数组people，然后第⑭行使用as操作符进行类型转换，将Person类型转换

为Student类型。类似地，第⑮行使用as操作符将Person类型转换为Worker类型。

协议作为类型使用，与其他类型没有区别，不仅可以使用as操作符进行类型转换，还可以使用is操作符进行类型检查。除了不能实例化之外，协议可以像其他类型一样使用。

16.2.5 协议的继承

一个协议继承其他协议就像是类继承一样，图16-2所示是一个继承关系的类。Person和Student都是协议，Student协议继承了Person协议，而Graduate类遵守Student协议。

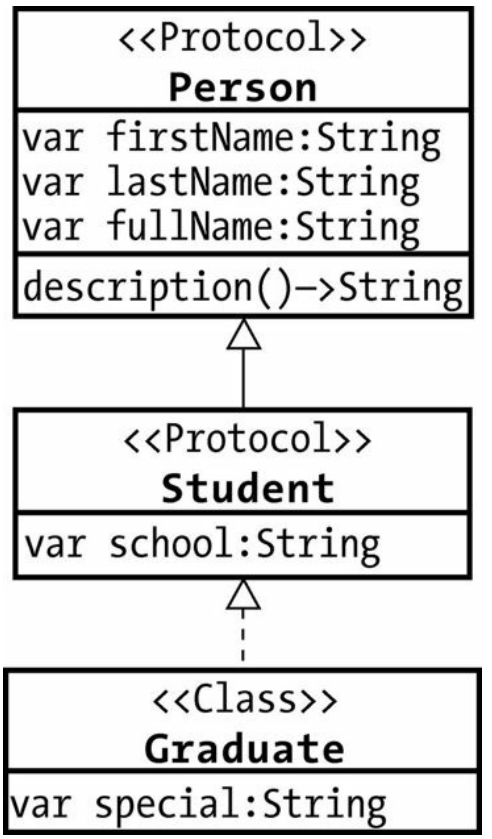


图 16-2　协议的继承

下面看具体的示例：

```
protocol Person {
①
    var firstName : String { get set }
    var lastName : String { get set }
    var fullName : String { get }
    func description() -> String
}

protocol Student : Person {
②
    var school : String { get set }
}

class Graduate : Student {
③

    var special : String

    var firstName : String
    var lastName : String
    var school : String

    var fullName : String {
        return self.firstName + "." +
self.lastName
    }
```

```
    func description() -> String {
        return " firstName: \(firstName)\n
lastName: \(lastName)\n
        School: \(school)\n Special: \
(special)"
    }

    init (firstName : String, lastName :
String, school : String, special : String) {
        self.firstName = firstName
        self.lastName = lastName
        self.school = school
        self.special = special
    }
}

let gStudent = Graduate(firstName : "Tom",
lastName : "Guan",
    school : "清华大学", special : "计算机")
④

println(gStudent.description())
```

上述代码第①行定义了Person协议，第②行定义了Student协议，Student协议继承Person协议，继承的声明与类继承声明一样使用冒

号“:”。如果有多个协议需要继承，可以用逗号“,”分隔各个协议。

第③行定义了Graduate类，它遵守Student协议，同时遵守Person协议。代码第④行实例化了Graduate。

16.2.6 协议的合成

多个协议可以临时合成一个整体，作为一个类型使用。首先要有一个类型在声明时遵守多个协议。如图16-3所示的轮船协议Ship和武器协议Weapon，它们都声明了一个可读性属性，军舰类WarShip同时遵守了这两个协议。

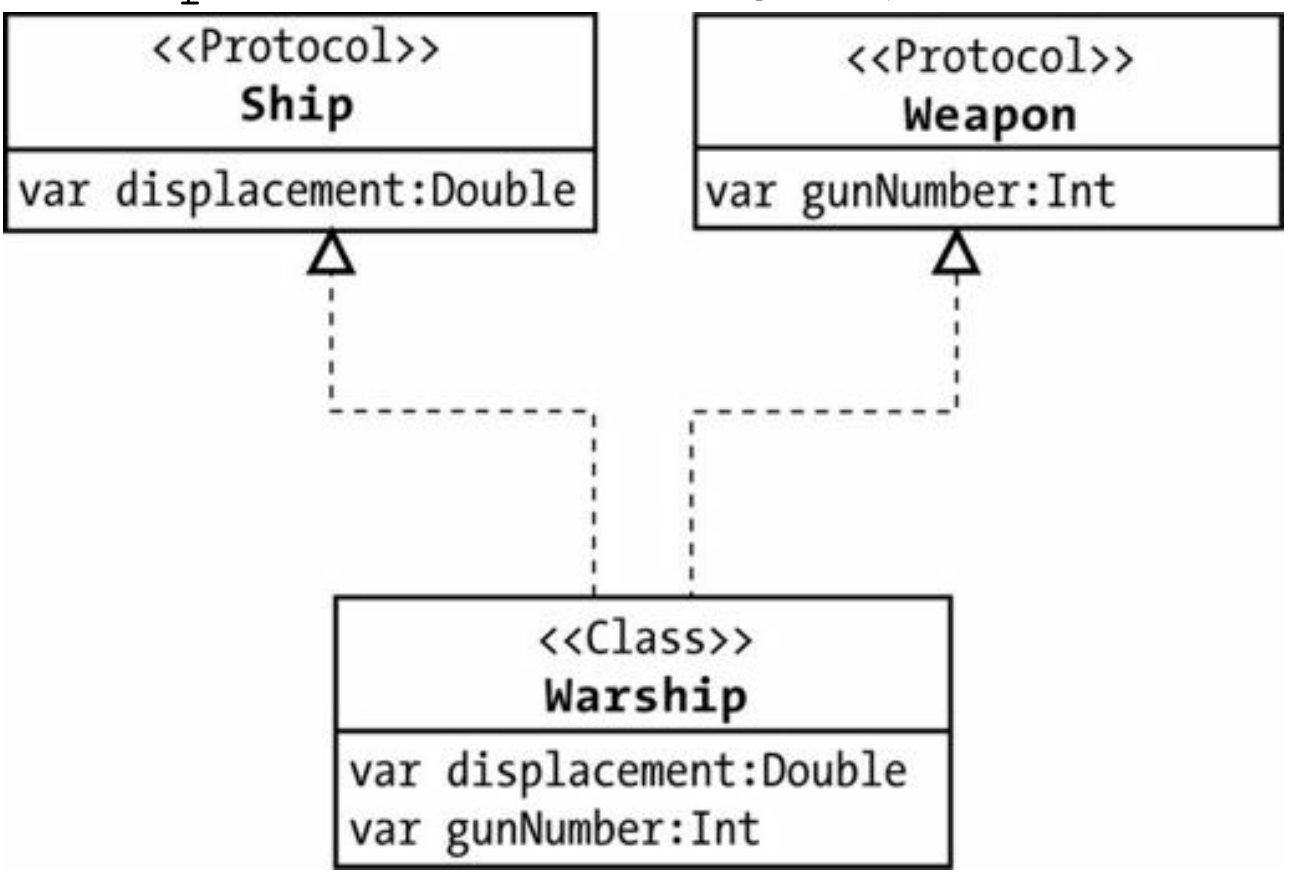


图 16-3　遵守多个协议

下面看具体的示例：

```
//定义轮船协议
protocol Ship {
①
    //排水量
    var displacement : Double { get set }
}

//定义武器协议
protocol Weapon {
②
    //火炮门数
    var gunNumber : Int { get set }
}

//定义军舰类
class WarShip : Ship, Weapon {
③
    //排水量
    var displacement  = 1000_000.00
    //火炮门数
    var gunNumber = 10
}

func showWarResource(resource: protocol<Ship,
Weapon>) {                                  ④
    println("Ship \(resource.displacement) -
```

```
Weapon \(resource.gunNumber)")      ⑤
}

let ship = WarShip()
showWarResource(ship)
⑥
```

上述代码第①行是定义轮船协议Ship，代码第②行是定义武器协议Weapon，代码第③行是定义军舰类，它遵守Ship和Weapon。

代码第④行定义函数showWarResource，其中参数为protocol<Ship, Weapon>类型，这种类型的参数要同时遵守Ship和Weapon协议。这种类型就是**协议合成**，它是一种临时的类型，当作用域结束时，这个类型就不会存在了。代码第⑤行中的参数resource可以访问displacement和gunNumber属性。

showWarResource函数是在代码第⑥行调用的，它的参数是WarShip类的实例，它能够满足protocol<Ship, Weapon>类型的要求。

16.3 扩展中声明协议

我们在16.1节介绍了扩展，在扩展中也可以声明遵守某个协议，语法如下所示：

```
extension 类型名 : 协议1, 协议2 {
    // 协议内容
}
```

下面我们看看示例代码：

```
protocol Editable  {
①
    mutating func edit()
}

struct Account {
②
    var amount : Double = 10.0                  //账户金额
    var owner : String = ""                     //账户名
}

extension Account : Editable {
③
    mutating func edit() {
```

```
④
        self.amount *= 100
        self.owner = "Tony"
    }
}

var account  = Account()
⑤
account.edit()
⑥
println("\(account.owner) - \(account.amount)")
⑦
```

上述代码第①行定义了Editable协议，第②行代码定义了Account结构体，第③行定义了Account结构体扩展，同时声明遵守Editable协议。第④行定义的方法是实现Editable协议中规定的方法，在方法中修改属性amount和owner。

第⑤行代码是创建Account实例，第⑥行是调用edit方法修改属性，最后代码第⑦行是打印修改之后的属性值。

16.4　本章小结

通过对本章内容的学习，我们理解了Swift中扩展和协议的重要性，掌握了基本概念，熟悉了如何扩展属性、扩展方法、扩展构造器和扩展下标。在协议部分，掌握了协议如何规定方法和属性，如何把协议当做一种类型使用，还了解了协议的继承和合成机制。

16.5 同步练习

1. 判断正误：整型、浮点型、布尔型、字符串等基本数据类型也可以有扩展机制。

2. Swift中扩展机制可以在原类型中添加新功能的内容包括（　　）。

A. 实例计算属性和静态计算属性

B. 实例方法和静态方法

C. 构造器

D. 下标

3. 判断正误：扩展机制可以扩展实例计算属性和静态计算属性。

4. 判断正误：扩展类的时候能向类中添加新的便利构造器，但不能添加新的指定构造器或析构器，指定构造器和析构器只能由原类型提供。

5. 判断正误：类、结构体和枚举，都可以遵守多个协议时，各协议之间用逗号“,”隔开。

6. 判断正误：如果一个类继承父类的同时也要求遵守协议，应当把父类放在所有的协议之前。

7. 判断正误：在结构体和枚举类型中可以定义变异方法，而在类中没有这种方法。原因是结构体和枚举类型中的属性是不可修改的，通过定义变异方法，可以在变异方法中修改这些属性。

8. 判断正误：虽然协议没有具体的实现代码，

不能被实例化，它的存在就是为了规范其他的类型遵守它实现。

9. 有以下协议定义：

```
protocol Speaker {
    class func speak()
}
```

则下列实现协议的遵守者定义正确的是（　　）。

A.

```
class Dog : Speaker {
    static func speak() {
        println("Wang Wang!")
    }
}
```

B.

```
class Dog : Speaker {
    class func speak() {
        println("Wang Wang!")
    }
```

```
}
```

C.

```
struct Cat : Speaker {
    static func speak() {
        println("Miao Miao!")
    }
}
```

D.

```
struct Cat : Speaker {
    class func speak() {
        println("Miao Miao!")
    }
}
```

10. 有以下协议定义：

```
protocol Speaker {
    mutating func speak()
}
```

则下列实现协议的遵守者定义正确的是（　　）。

A.

```
class Dog : Speaker {
    mutating func speak() {
        println("Wang Wang!")
    }
}
```

B.

```
struct Cat : Speaker {
    mutating func speak() {
        println("Miao Miao!")
    }
}
```

C.

```
class Dog : Speaker {
    func speak() {
        println("Wang Wang!")
```

```
    }
}
```

D.

```
struct Cat : Speaker {
    func speak() {
        println("Miao Miao!")
    }
}
```

11. 编程题：编写一个Array扩展，使其能够计算集合的元素之和。

12. 编程题：编写程序包含协议、基类和子类等内容。

13. 编程题：编写程序包含协议、基类、子类和扩展等内容。

第 17 章　Swift内存管理

很多计算机语言中的内存管理常常令人谈之色变。比如，以C++和C为代表的手动内存管理模式，使用起来非常麻烦，经常导致内存泄漏和过度释放等问题。再如，以Java和C#为代表的内存垃圾回收机制（Garbage Collection，GC），程序员不用关心内存释放的问题，这种方式在后台有一个线程，负责检查已经不再使用的对象，然后将其释放。由于后台有一个线程一直运行，因此会严重影响性能。

而Objective-C的内存管理经历过两个阶段：手动引用计数内存管理（Manual Reference Counting，MRC）和自动引用计数内存管理（Automatic Reference Counting，ARC）。MRC就是由程序员自己负责管理对象生命周期，负责对象的创建和销毁。ARC就是程序员不用关心对象释放的问题，编译器在编译的时候在合适的位置插入对象内存释放代码。

Swift在内存管理方面吸收了Objective-C的先进思想，采用了ARC（自动引用计数）内存管理模式。

17.1 Swift内存管理概述

具体而言，Swift中的ARC内存管理是对引用类型的管理，即对类所创建的对象采用ARC管理。而对于值类型，如整型、浮点型、布尔型、字符串、元组、集合、枚举和结构体等，是由处理器自动管理的，程序员不需要管理它们的内存。

提示 ARC内存管理和值类型内存管理有一定的区别。虽然两者都不需要程序员管理，但本质上是有区别的，ARC和MRC一样都是针对引用类型的管理，引用类型与Objective-C中的对象指针类型一样，它们的内存分配区域是在“堆”上的，需要人为管理。而值类型内存分配区域是在“栈”上的，由处理器管理，不需要人为管理。

17.1.1 引用计数

每个Swift类创建的对象都有一个内部计数器，这个计数器跟踪对象的引用次数，称为引用**计数**（Reference Count，简称RC）。当对象被创建的时候，引用计数为1，每次对象被引用的时候会使其引用计数加1，如果不需要的时候，对象引用断开（赋值为`nil`），其引用计数减1。当对象的引

用计数为0的时候，对象的内存才被释放。

图17-1是内存引用计数原理示意图。图中的房间就好比是对象的内存，一个人进入房间打开灯，就是创建一个对象，这时候对象的引用计数是1。有人进入房间，引用计数加1；有人离开房间，引用计数减1。最后一个人离开房间，引用计数为0，房间灯关闭，对象内存才被释放。

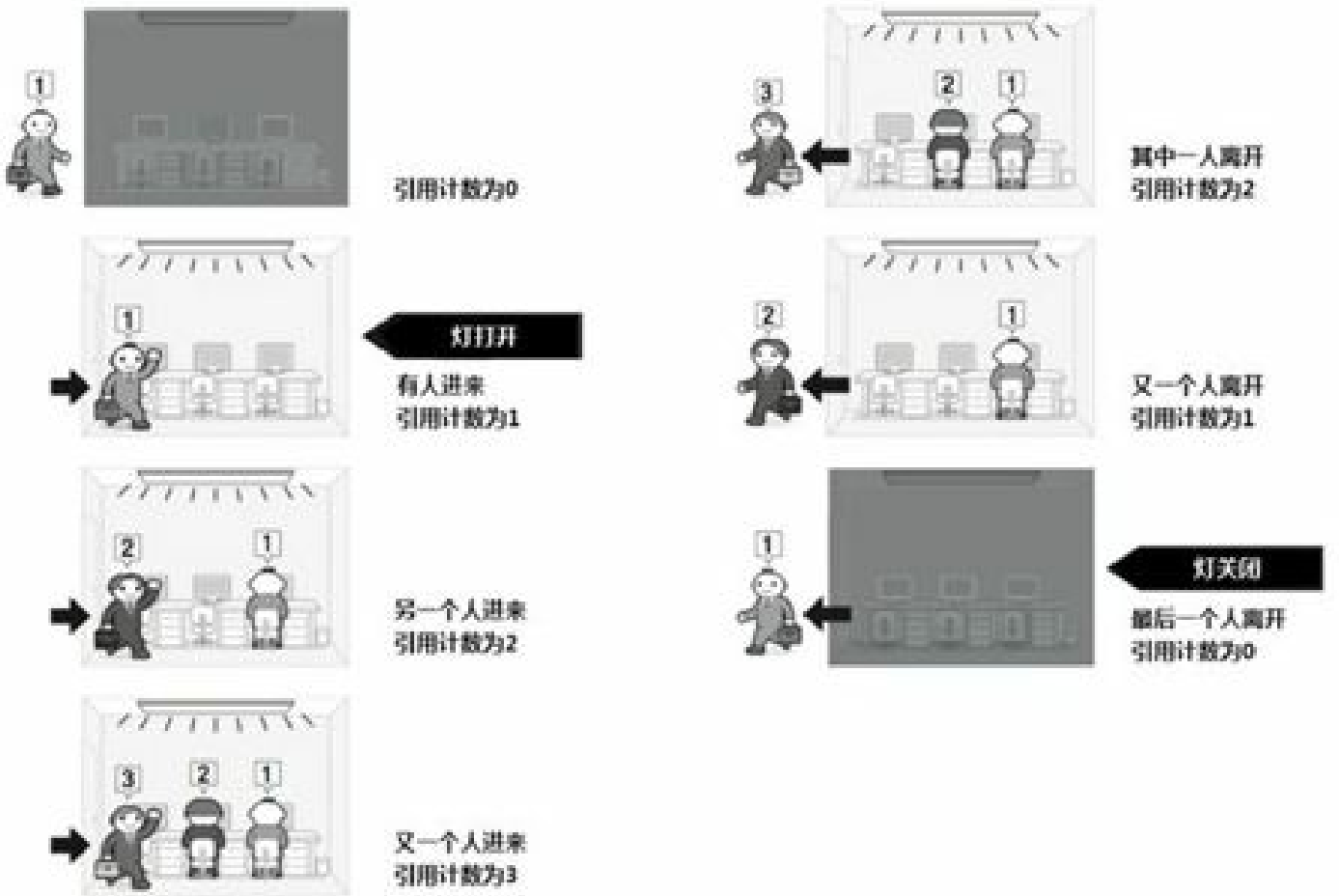


图 17-1　内存引用计数原理示意图

17.1.2　示例：Swift自动引用计数

下面我们通过一个示例了解一下Swift中的自动

引用计数原理。图17-2是Employee类创建的对象的生命周期，该图描述了对象被赋值给3个变量，以及它们的释放过程。

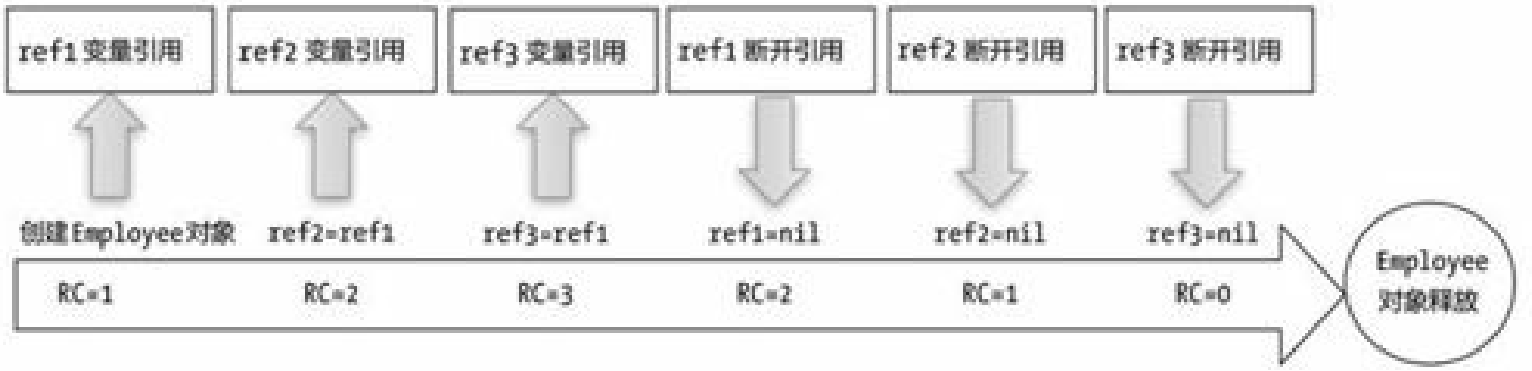


图 17-2　Employee对象生命周期

示例代码如下：

```
class Employee {
①
    var no : Int
    var name : String
    var job : String
    var salary : Double

    init(no : Int, name: String, job : String,
salary : Double) {                    ②
        self.no = no
        self.name = name
        self.job = job
        self.salary = salary
        println("员工\(name) 已经构造成功。")
③
```

```
    }
    deinit {
④
        println("员工\(name) 已经析构成功。")
⑤
    }
}

var ref1: Employee?
⑥
var ref2: Employee?
⑦
var ref3: Employee?
⑧

ref1 = Employee(no: 7698, name: "Blake", job
:"Salesman", salary : 1600)     ⑨

ref2 = ref1
⑩
ref3 = ref1
⑪

ref1 = nil
⑫
ref2 = nil
⑬
ref3 = nil
⑭
```

上述代码第①行声明了Employee类，第②行代码是定义构造器，在构造器中初始化存储属性，并且在代码第③行输出构造成功信息。第④行代码是定义析构器，并在代码第⑤行输出析构成功信息。

代码第⑥~⑧行是声明3个Employee类型变量，这个时候还没有创建Employee对象分配内存空间。代码第⑨行是真正创建Employee对象分配内存空间，并把对象的引用分配给ref1变量，ref1与对象建立“强引用”关系，“强引用”关系能够保证对象在内存中不被释放，这时候它的引用计数是1。第⑩行代码ref2 = ref1是将对象的引用分配给ref2，ref2也与对象建立“强引用”关系，这时候它的引用计数是2。第⑪行代码ref3 = ref1是将对象的引用分配给ref3，ref3也与对象建立“强引用”关系，这时候它的引用计数是3。

然后在代码第⑫行通过ref1 = nil语句断开ref1对Employee对象的引用，这时候它的引用计数是2。以此类推，ref2 = nil时它的引用计数是1，ref3 = nil时它的引用计数是0，当引用

计数为0的时候Employee对象被释放。

我们可以测试一下看看效果，如果设置断点单步调试，会发现代码运行完第⑨行后控制台输出：

```
员工Blake 已经构造成功。
```

析构器输出的内容直到运行完第⑭行代码才输出：

```
员工Blake 已经析构成功。
```

这说明只有在引用计数为0的情况下才调用析构器，释放对象。

提示 在Playground环境下测试和运行上述代码，是不会调用析构器的，也不能调试代码，我们需要使用Xcode创建一个iOS应用程序，创建过程请参考14.3节，然后在应用程序中运行这段代码，运行过程请参考14.3节。

17.2 强引用循环

当两个对象的存储属性互相引用对方的时候，一个对象释放的前提是对方先释放，另一对象释放的前提也是对方先释放，这样就会导致类似于“死锁”的状态，最后谁都不能释放，导致内存泄漏。这种现象就是**强引用循环**。

假设我们开发一个人力资源管理系统，其中Employee（员工）与Department（部门）的关联关系如图17-3所示，Employee的dept属性关联到Department，Department的manager（部门领导）属性关联到Employee。

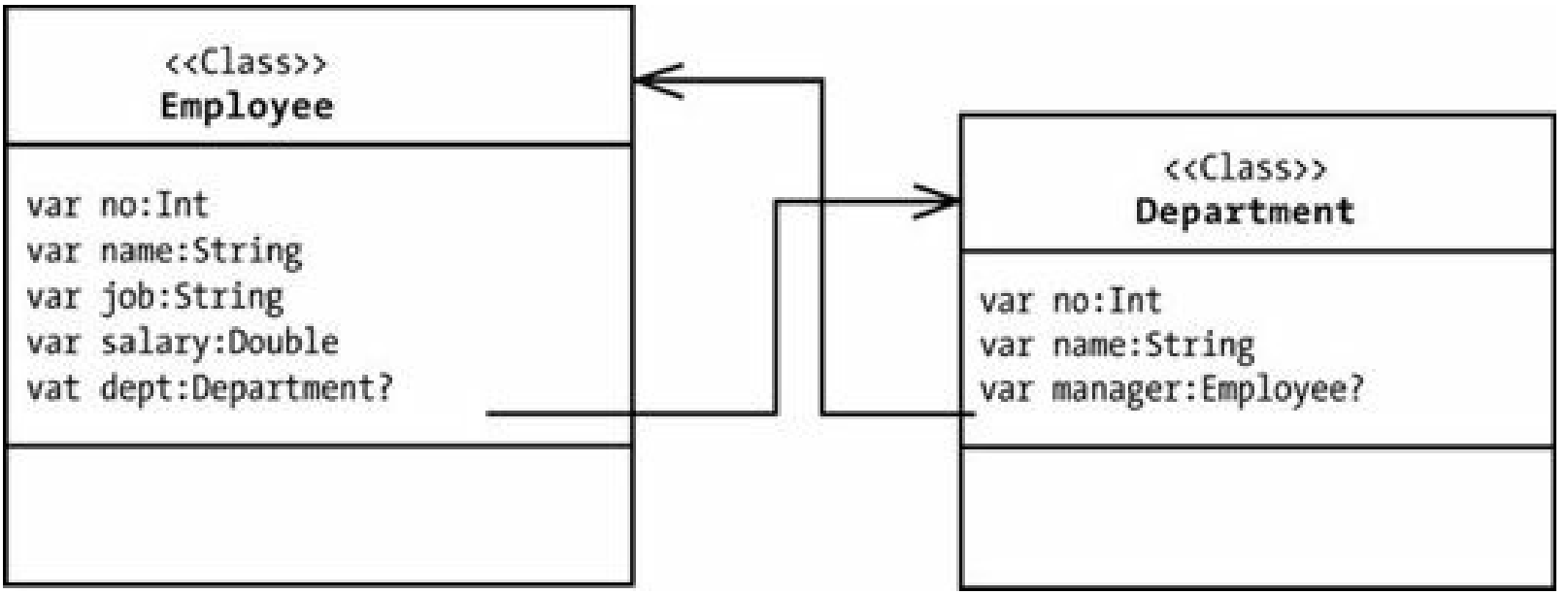


图 17-3 Employee与Department的关联关系

示例代码如下：

```
class Employee {
①
    var no : Int
    var name : String
    var job : String
    var salary : Double
    var dept : Department?
②

    init(no : Int, name: String, job : String,
salary : Double) {
        self.no = no
        self.name = name
        self.job = job
        self.salary = salary
        println("员工\(name) 已经构造成功。")
    }
    deinit {
        println("员工\(name) 已经析构成功。")
    }

}

class Department {
③
    var no : Int = 0
    var name : String = ""
    var manager : Employee?
④
```

```
    init(no : Int, name: String) {
        self.no = no
        self.name = name
        println("部门\(name) 已经构造成功。")
    }
    deinit {
        println("部门\(name) 已经析构成功。")
    }
}

var emp: Employee?
⑤
var dept: Department?
⑥

emp = Employee(no: 7698, name: "Blake", job
:"Salesman", salary : 1600)    ⑦
dept = Department(no : 30, name: "Sales")
⑧

emp!.dept = dept
⑨
dept!.manager = emp
⑩



emp = nil
⑪
dept = nil
```

⑫

上述代码第①行定义了员工类Employee，第②行代码var dept : Department?声明所在部门的属性，它的类型是Department可选类型。第③行代码定义了部门类Department，第④行代码 var manager : Employee?声明部门领导的属性，它的类型是Employee可选类型。

第⑤行代码var emp: Employee?声明Employee引用类型变量emp，第⑥行代码var dept: Department?声明Department引用类型变量dept。

第⑦行代码创建Employee对象并赋值给emp，emp与Employee对象建立强引用关系。第⑧行代码创建Department对象并赋值给dept，dept与Department对象建立强引用关系。但是此时，emp和dept两个对象之间并没有建立关系，它们之间的关系如图17-4所示。

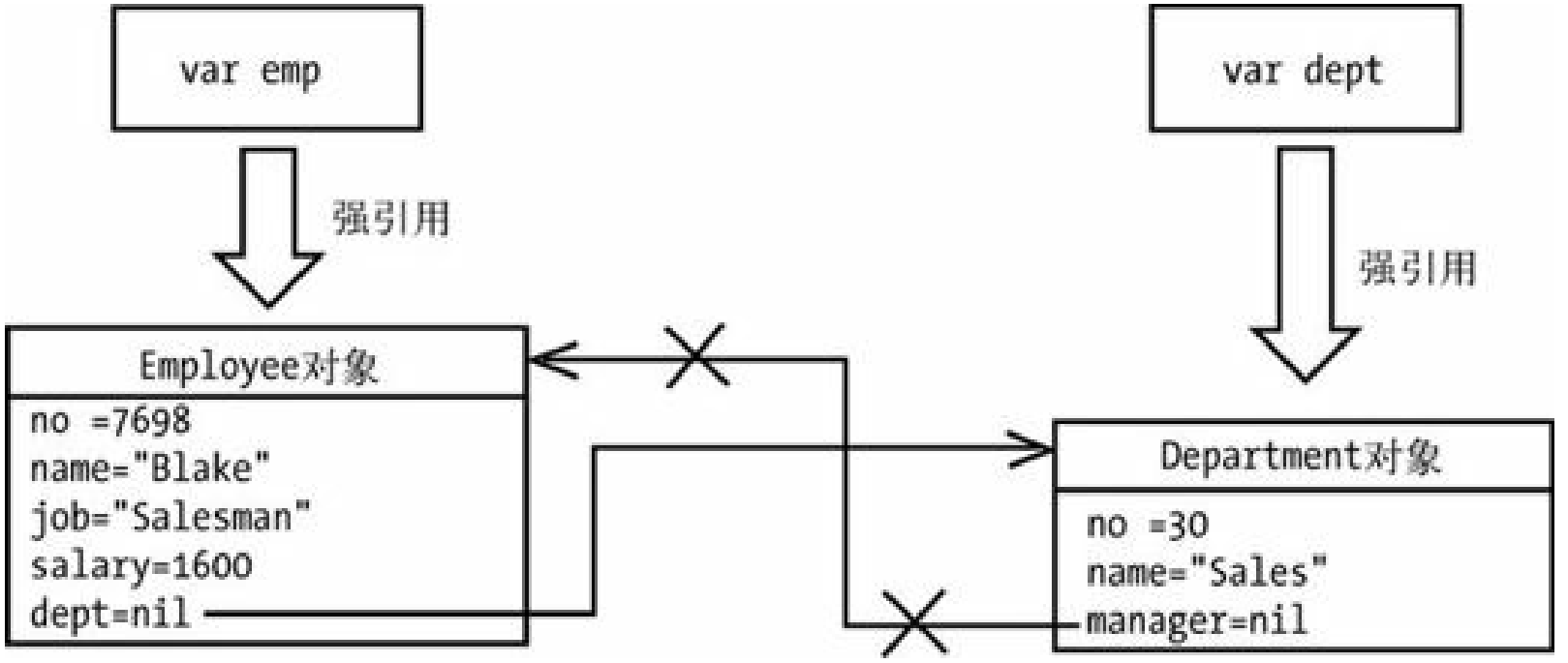


图 17-4　emp与dept对象之间没有建立关系

代码第⑨行emp!.dept = dept将引用变量dept赋值给Employee的dept属性，代码第⑩行dept!.manager = emp将引用变量emp赋值给Department的manager属性，此时emp和dept两个对象就建立了关系。它们之间的关系如图17-5所示。

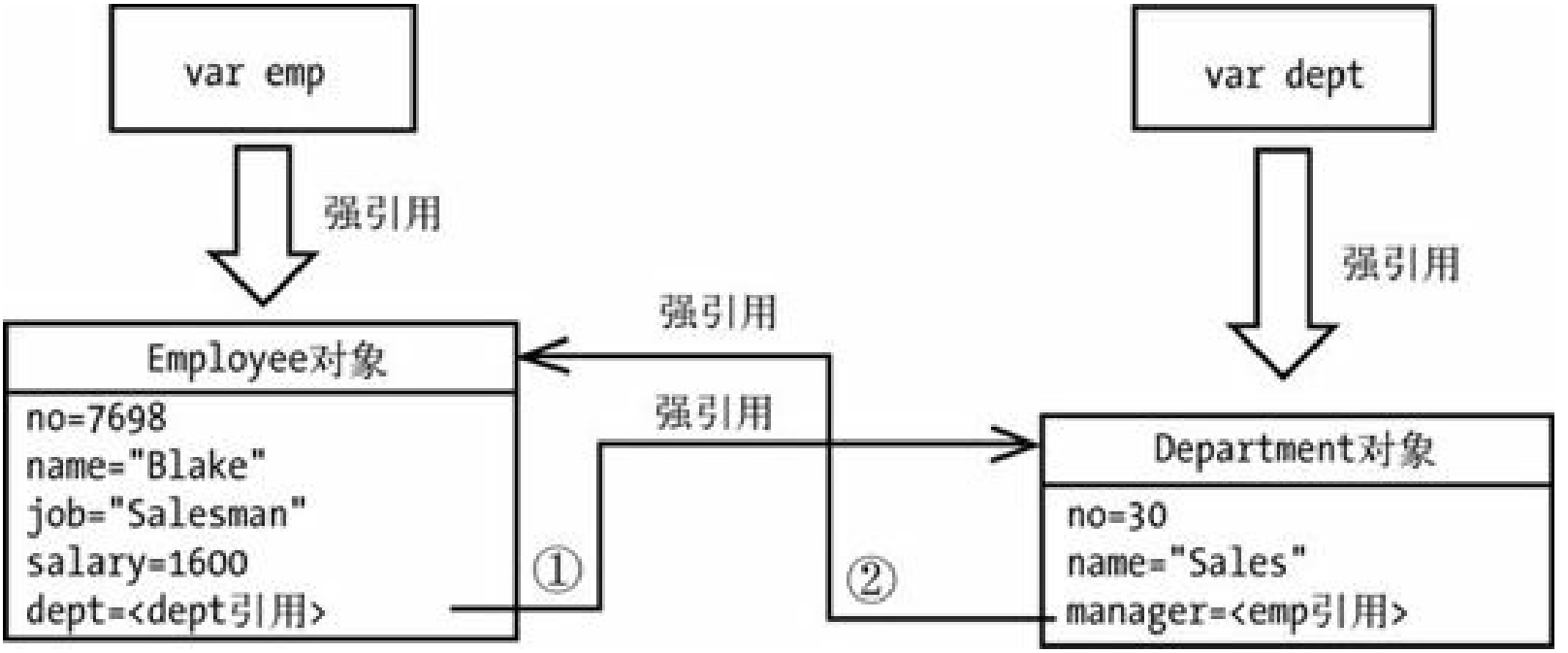


图 17-5　emp与dept对象之间建立关系

如果我们通过第⑪行代码`emp = nil`和第⑫行代码`dept = nil`断开引用关系，如图17-6所示，但是`Employee`对象和`Department`对象并没有被释放。这是因为①号引用关系（`Employee`对象`dept`属性引用`Department`对象）保持`Department`对象不被释放。而②号引用关系（`Department`对象`manager`属性引用`Employee`对象）保持`Employee`对象不被释放。

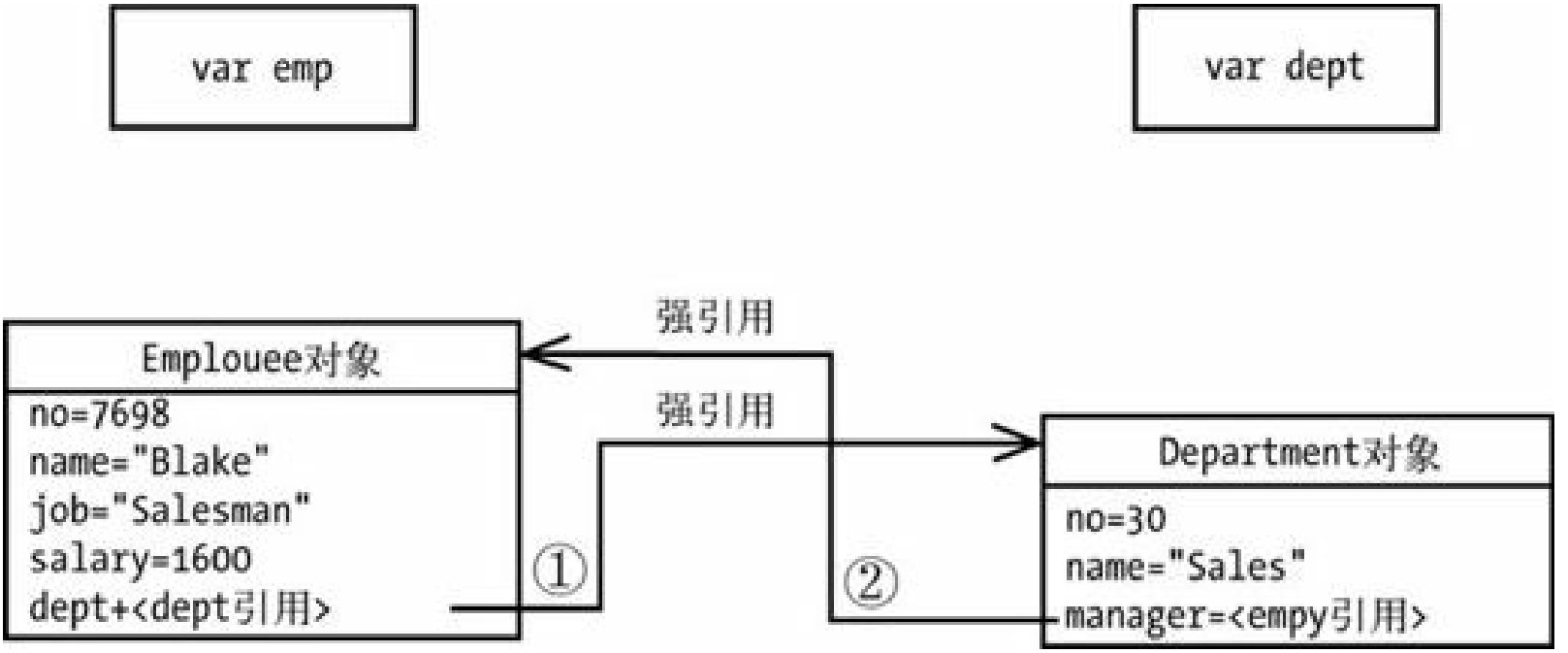


图 17-6　emp和dept强引用断开

最后Employee对象和Department对象都没有被释放，这就是强引用循环，会导致内存泄漏。

17.3 打破强引用循环

打一个比方，强引用循环就像是两个人在吵架，每个人都顾及面子，都不肯示弱，都希望对方服个软，于是这场吵架就永远不会停止。如果想让吵架停下来，就需要让一个人主动示弱，服个软，吵架就会停止了。

打破强引用循环方法与停止吵架是类似的，我们在声明一个对象的属性时，让它具有能够“主动示弱”的能力，当遇到强引用循环问题的时候，不保持强引用。

Swift 提供了两种办法来解决强引用循环问题：弱引用（weak reference）和无主引用（unowned reference）。

17.3.1 弱引用

弱引用允许循环引用中的一个对象不采用强引用方式引用另外一个对象，这样就不会引起强引用循环问题。弱引用适合于引用对象可以没有值的情况，因为弱引用可以没有值，我们必须将每一个弱引用声明为可选类型，使用关键字`Weak`声明为弱引用。

例如17.2节中介绍的人力资源管理系统，`Employee`的`dept`属性关联

到Department，Department的manager（部门领导）属性关联到Employee。如果我们规定的业务需求是一个员工可以没有部门（刚刚入职），即Employee的dept（所在部门）属性可以为nil。一个部门也可以暂时没有领导，即Department的manager（部门领导）属性也可以为nil。

示例代码如下：

```
class Employee {
①
    var no : Int
    var name : String
    var job : String
    var salary : Double
    var dept : Department?
②

    init(no : Int, name: String, job : String,
salary : Double) {
        self.no = no
        self.name = name
        self.job = job
        self.salary = salary
        println("员工\(name) 已经构造成功。")
    }
```

```
    deinit {
        println("员工\(name) 已经析构成功。")
    }

}

class Department {
③
    var no : Int = 0
    var name : String = ""
    weak var manager : Employee?
④

    init(no : Int, name: String) {
        self.no = no
        self.name = name
        println("部门\(name) 已经构造成功。")
    }
    deinit {
        println("部门\(name) 已经析构成功。")
    }
}

var emp: Employee?
var dept: Department?

emp = Employee(no: 7698, name: "Blake", job
:"Salesman", salary : 1600)
dept = Department(no : 30, name: "Sales")
```

```
emp!.dept = dept
⑤
dept!.manager = emp
⑥

emp = nil
⑦
dept = nil
⑧
```

上述代码第①行定义了员工类Employee，第②行代码var dept : Department?声明所在部门的属性，它的类型是Department可选类型。第③行代码定义了部门类Department，第④行代码weak var manager : Employee?声明部门领导的属性，它的类型是Employee可选类型，使用关键字weak声明为弱引用。

第⑤行代码emp!.dept = dept建立强引用关系，第⑥行代码dept!.manager = emp建立弱引用关系。如图17-7所示，其中的关系②是弱引用关系。

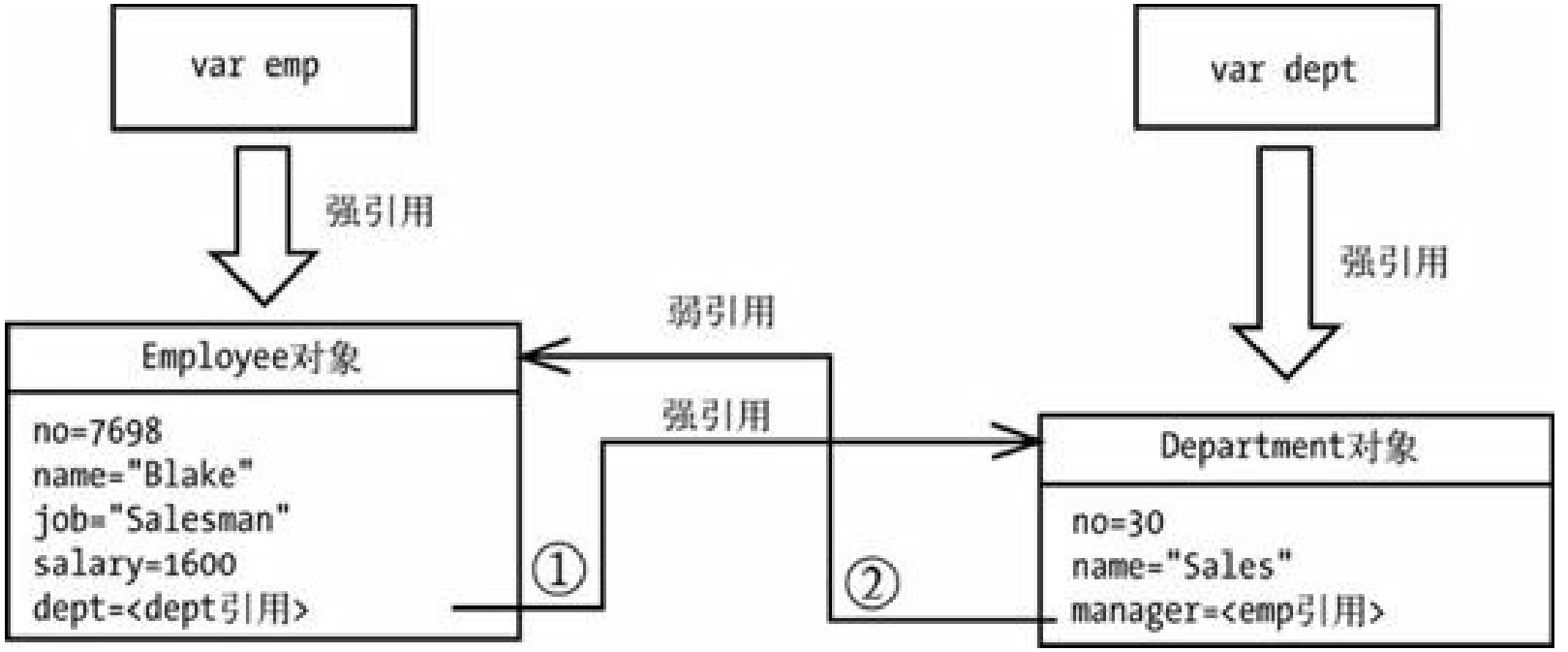


图 17-7　emp与dept对象之间建立关系

如果我们通过第⑦行代码`emp = nil`和第⑧行代码`dept = nil`断开引用关系，如图17-8所示，由于没有指向`Employee`对象的强引用，所以`Employee`对象会被释放。①号强引用关系也会被打破，`Department`对象被释放。

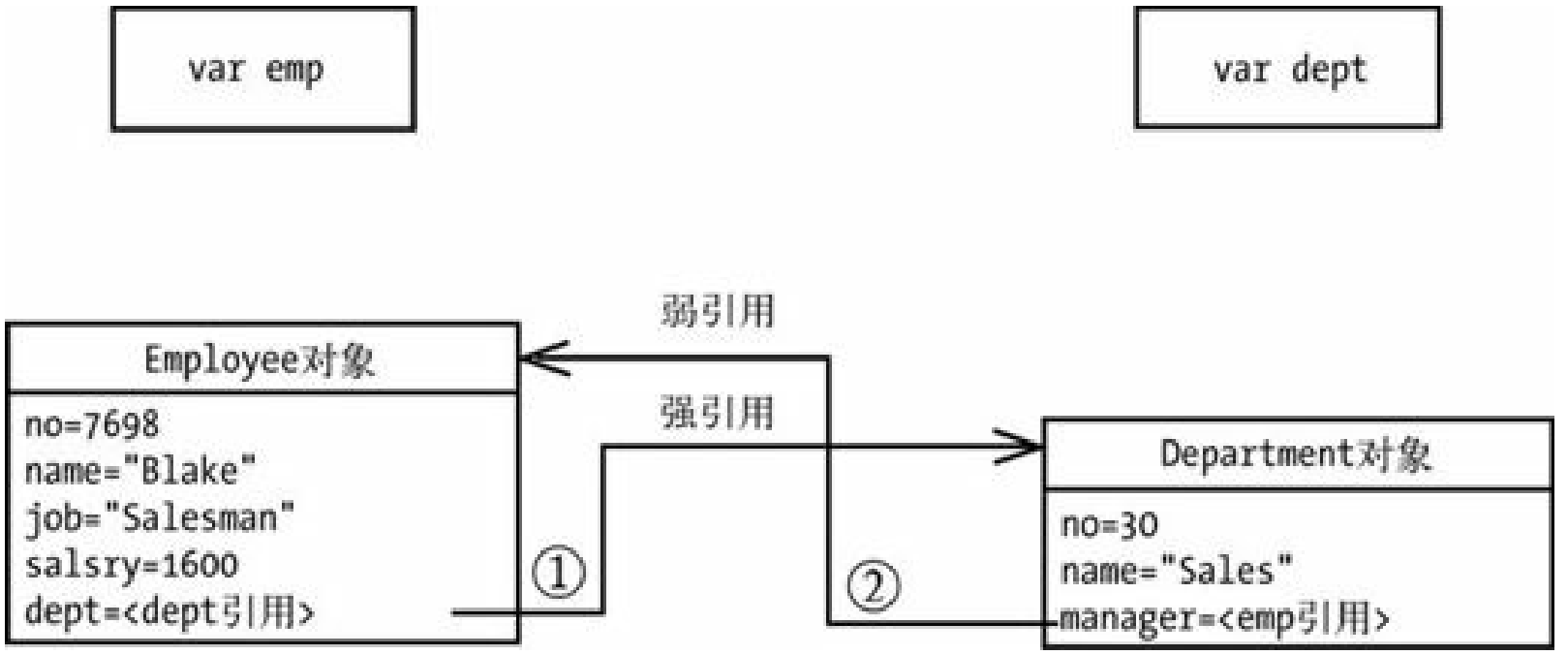


图 17-8　emp和dept弱引用断开

17.3.2 无主引用

无主引用与弱引用一样，允许循环引用中的一个对象不采用强引用方式引用另外一个对象，这样就不会引起强引用循环问题。无主引用适用于引用对象永远有值的情况，它总是被定义为非可选类型，使用关键字unowned表示这是一个无主引用。

例如17.2节中介绍的人力资源管理系统，Employee的dept属性关联到Department，Department的manager（部门领导）属性关联到Employee。如果我们规定的业务需求是一个员工可以没有部分（刚刚入职），即Employee的dept（所在部门）属性可以为nil，一个部分不能没有领导，Department的manager（部门领导）属性是不能为nil的。如图17-9所示，其中Employee的dept声明中有问号“?”，说明是可选类型，也就是可以为nil。而Department的manager声明中没有问号“?”，说明是非可选类型，不可以为nil。

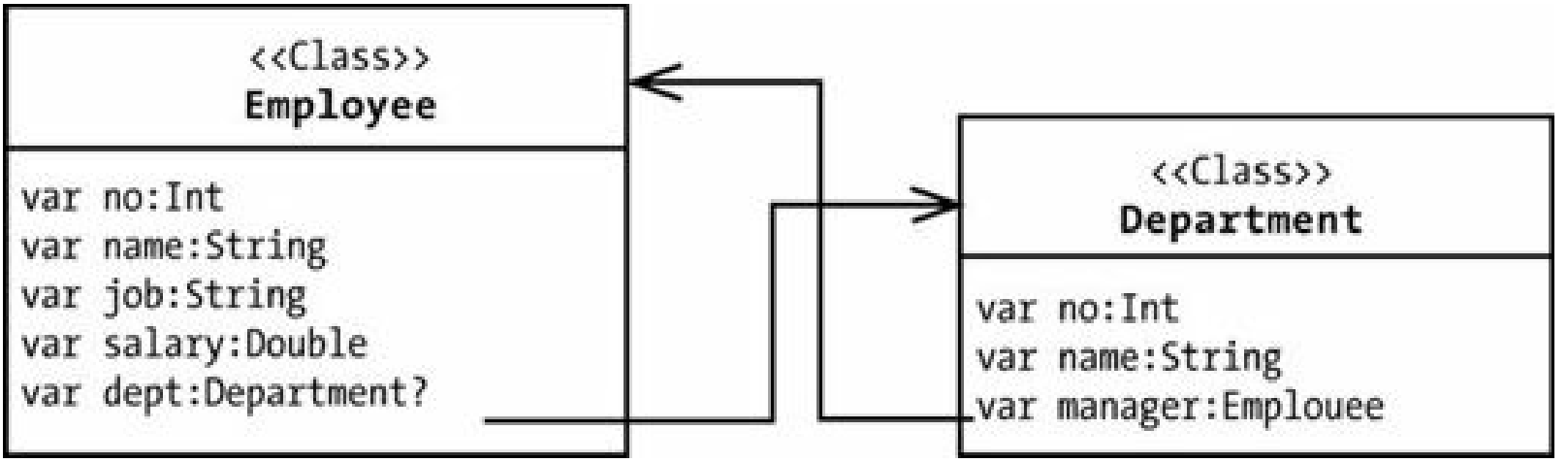


图 17-9　Employee与Department关联关系

示例代码如下：

```
class Employee {
①
    var no : Int
    var name : String
    var job : String
    var salary : Double
    var dept : Department?
②

    init(no : Int, name: String, job : String,
salary : Double) {
        self.no = no
        self.name = name
        self.job = job
        self.salary = salary
        println("员工\(name) 已经构造成功。")
    }
    deinit {
```

```
        println("员工\(name) 已经析构成功。")
    }

}

class Department {
③
    var no : Int = 0
    var name : String = ""
    unowned var manager : Employee
④

    init(no : Int, name: String, manager :
Employee) {
        self.no = no
        self.name = name
        self.manager = manager
        println("部门\(name) 已经构造成功。")
    }
    deinit {
        println("部门\(name) 已经析构成功。")
    }
}

var emp: Employee?

emp = Employee(no: 7698, name: "Blake", job
:"Salesman", salary : 1600)    ⑤
emp!.dept = Department(no : 30, name: "Sales",
manager : emp!)                ⑥
```

```
emp = nil
⑦
```

上述代码第①行定义了员工类Employee，第②行代码var dept : Department?声明所在部门的属性，它的类型是Department可选类型。第③行代码定义了部门类Department，第④行代码unowned var manager : Employee声明部门领导的属性，它的类型是Employee类型，不能为nil，使用unowned关键字声明为无主引用。

第⑤行代码创建Employee对象，第⑥行代码给Employee对象的dept属性赋值，如图17-10所示，这里实例化Department对象并没有像上一节一样把Department对象的引用赋值给一个变量，而是直接把Department对象赋值给Employee对象的dept属性。由于没有Department对象的引用变量，我们不能通过给引用变量赋值为nil的方式来释放对象，它的释放依赖于Employee对象的释放，这就是无主引用的特点。

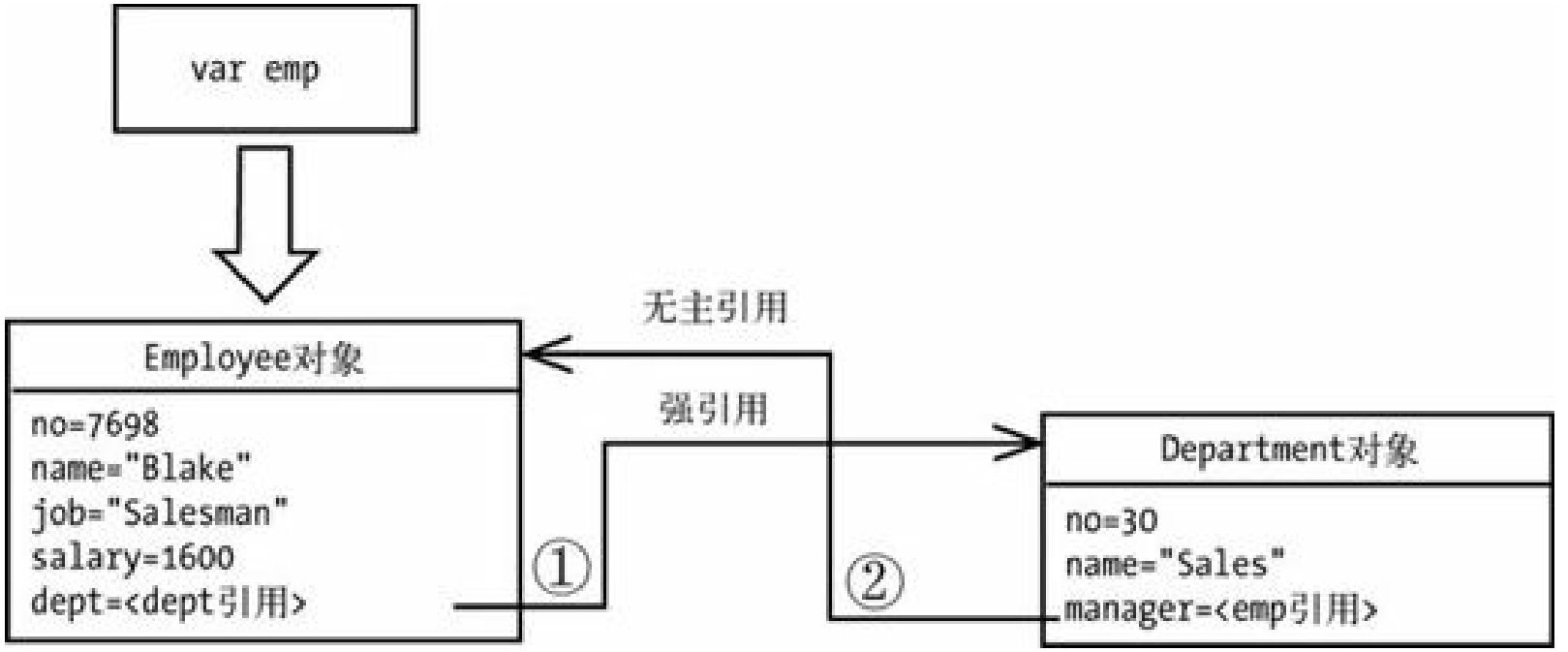


图 17-10　emp与dept对象之间建立关系

第⑦行代码通过`emp = nil`语句断开强引用关系。如图17-11所示，由于没有指向`Employee`对象的强引用，所以`Employee`对象会被释放。然后①号强引用关系也被打破，`Department`对象被释放。

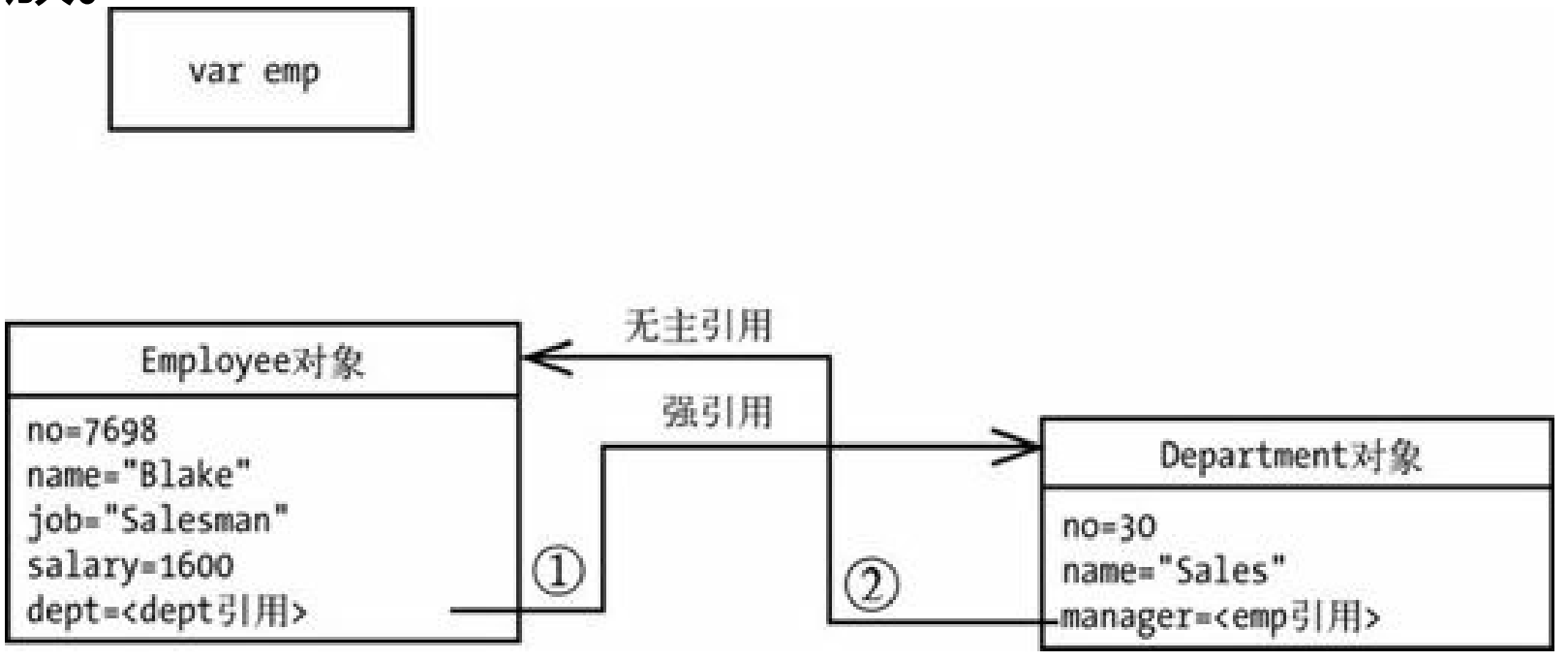


图 17-11　emp和dept无主引用断开

另外，上述代码第②行的可选类型可以声明为如下形式：

```
var dept : Department!
```

这样当我们引用该对象时候可以采用隐式拆封。隐式拆封表达式是emp!.dept.name，但如果不使用隐式拆封，表达式形式是emp!.dept!.name。使用可选类型隐式拆封可以使代码变得更加简洁，引用它的时候省略了感叹号（！）。

17.4 闭包中的强引用循环

由于闭包本质上也是引用类型，因此也可能在闭包和上下文捕获变量（或常量）之间出现强引用循环问题。

并不是所有的捕获变量（或常量）都会发生强引用循环问题，只有将一个闭包赋值给对象的某个属性，并且这个闭包体使用了该对象，才会产生闭包强引用循环。

17.4.1 一个闭包中的强引用循环示例

我们通过一个示例来了解一下闭包中的强引用循环。示例代码如下：

```
class Employee {
①
    var no : Int = 0
    var firstName : String
    var lastName : String
    var job : String
    var salary : Double

    init(no : Int, firstName : String, lastName
: String, job : String, salary : Double) {
        self.no = no
        self.firstName = firstName
        self.lastName = lastName
```

```
        self.job = job
        self.salary = salary
        println("员工\(firstName) 已经构造成功。")
    }
    deinit {
        println("员工\(firstName) 已经析构成功。")
    }

    lazy var fullName : ()-> String = {
②
        return self.firstName + "." +
self.lastName
③
    }
}

var emp : Employee? = Employee(no: 7698,
firstName: "Tony", lastName : "Guan",
job :"Salesman", salary : 1600)

println(emp!.fullName())
④
emp = nil
⑤
```

上述代码第①行定义了类Employee，第②行代码定义了计算属性fullName，这个属性值的计

算是通过一个闭包实现的，闭包的返回值类型是()-> String。属性fullName被声明为lazy，说明该属性是延迟加载的，由于在第③行代码中捕获了self，self能够在闭包体中使用，那么属性必须声明为lazy，即所有属性初始化完成后，self表示的对象才能被创建。

第④行代码emp!.fullName()调用闭包返回fullName属性值。代码第⑤行emp = nil断开强引用，释放对象。程序输出的结果是：

```
员工Tony 已经构造成功。
Tony.Guan
```

从结果可见，析构器并没有被调用，也就是说对象没有被释放。导致这个问题的原因是闭包与捕获对象之间发生了强引用循环。

17.4.2 解决闭包强引用循环

解决闭包强引用循环问题有两种方法：弱引用和无主引用。到底应该采用弱引用还是无主引用，与两个对象之间的选择条件是一样的。如果闭包和捕获的对象总是互相引用并且总是同时销毁，则将

闭包内的捕获声明为无主引用。当捕获的对象有时可能为nil时，则将闭包内的捕获声明为弱引用。如果捕获的对象绝对不会为nil，那么应该采用无主引用。

Swift在闭包中定义了捕获列表来解决强引用循环问题，基本语法如下：

```
lazy var 闭包: <闭包参数列表> -><返回值类型> = {
①
    [unowned 捕获对象] <闭包参数列表> -><返回值类型
> in                ②
  或 [weak 捕获对象]   <闭包参数列表> -><返回值类型
> in                ③
    //闭包体
}
```

或

```
lazy var 闭包: () -> <返回值类型> = {
④
    [unowned 捕获对象] in
⑤
  或 [weak 捕获对象] in
⑥
    //闭包体
```

```
}
```

上述语法格式可以定义两种闭包捕获列表，其中第①行代码的语法格式是最为普通的格式，其中<闭包参数列表> ->><返回值类型>与第②行和第③行的<闭包参数列表> ->><返回值类型>要对应上。示例如下：

```
lazy var fullName : (String, String) -> String
= {
    [weakself]  (firstName : String, lastName :
String) -> String in
    //闭包体
}
```

第④行的语法格式是无参数情况下的捕获列表，可以省略参数类别，只保留in，Swift编译器会通过上下文推断出参数列表和返回值类型。示例如下：

```
lazy var fullName : () -> String = {
    [unownedself]  in
    //闭包体
```

```
}
```

下面看示例代码：

```
class Employee {
    var no : Int = 0
    var firstName : String
    var lastName : String
    var job : String
    var salary : Double

    init(no : Int, firstName : String, lastName
: String, job : String, salary : Double) {
        self.no = no
        self.firstName = firstName
        self.lastName = lastName
        self.job = job
        self.salary = salary
        println("员工\(firstName) 已经构造成功。")
    }
    deinit {
        println("员工\(firstName) 已经析构成功。")
    }

    lazy var fullName : ()-> String = {
        [weak self]  ()-> String in
①
        letfn = self!.firstName
```

```
②
        letln = self!.lastName
③
        returnfn + "." + ln
    }
}

var emp : Employee? = Employee(no: 7698,
firstName: "Tony", lastName : "Guan",
job :"Salesman", salary : 1600)

println(emp!.fullName())

emp = nil
```

我们将第①行代码修改为[weak self] ()-> String in，该捕获列表是弱引用，捕获对象是self。由于是弱引用，在引用self的时候，代码第②行和第③行需要在后面加感叹号“!”，表明是强制拆封。

程序输出的结果是：

```
员工Tony 已经构造成功。
Tony.Guan
员工Tony 已经析构成功。
```

从结果可见析构器被调用了。

我们可以将fullName属性定义为无主引用的捕获列表形式，代码如下：

```
lazy var fullName : ()-> String = {
    [unownedself]  in
    letfn = self.firstName
    letln = self.lastName
    returnfn + "." + ln
}
```

采用哪一种引用方式，可以根据用户需求而定，这里不再赘述。此外，我们还可以根据需要为闭包提供参数。

17.5　本章小结

通过对本章内容的学习，我们了解了Swift的内存管理机制和ARC内存管理的原理，学会了如何解决对象间的强引用循环问题和闭包与引用对象之间的强引用循环问题。

17.6 同步练习

1. 关于Swift内存管理，下列选项中说明正确的是（　　）。

A. 采用垃圾回收机制　　B. ARC

C. MRC　　D. GC

2. 请简单介绍Swift的ARC内存管理机制。

3. 判断正误：强引用关系能够保证对象在内存中不被释放。

4. 程序分析题：

```
class Dog {
    var name: String = "未命名"
    init(name: String) {
        self.name = name
    }
}

var reference1: Dog?
var reference2: Dog?
var reference3: Dog?

reference1 = Dog(name: "泰迪")          ①
reference2 = reference1                  ②
reference3 = reference1                  ③

reference1 = nil                         ④
reference3 = nil                         ⑤
```

运行上述程序片段，Dog对象的引用计数为多少？

5. 请简单介绍强引用循环。

6. 解决强引用循环方法有哪些？（　　）

A. 弱引用　　B. 无主引用　　C. 强引用　　D. 闭包强引用

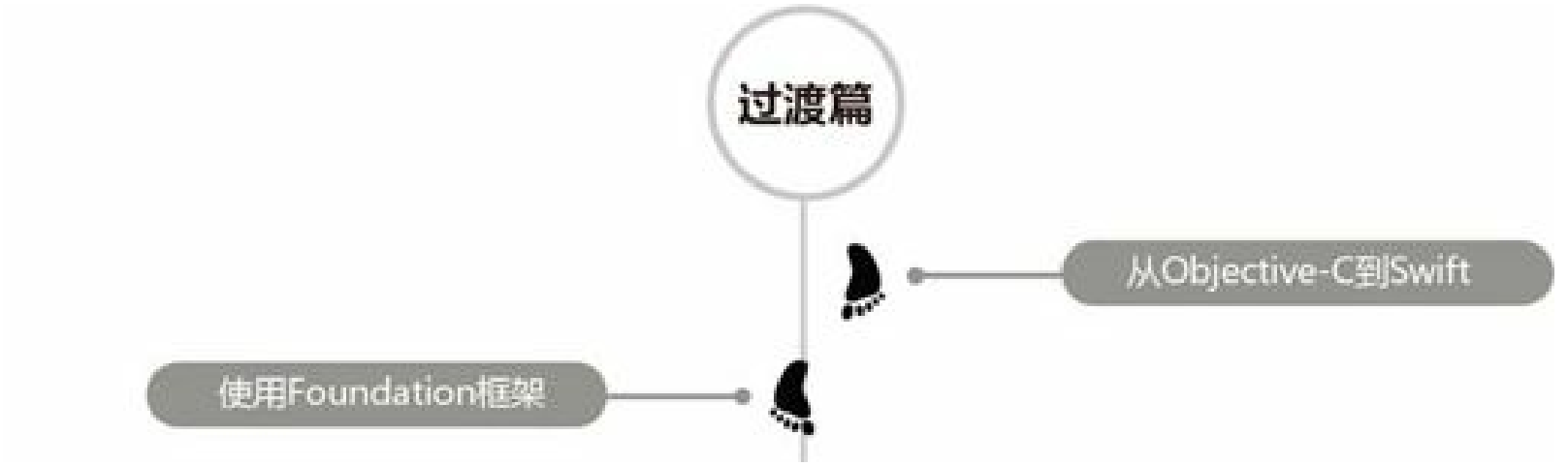
7. 判断正误：由于闭包本质上也是引用类型，因此也可能在闭包和上下文捕获变量（或常量）之间出现强引用循环问题。

8. 编程题：编写一个程序体现通过弱引用打破强引用循环示例。

9. 编程题：编写一个程序体现通过无主引用打破强引用循环示例。

10. 编程题：编写一个程序体现解决闭包强引用循环示例。

第三部分　过渡篇



第 18 章　从Objective-C到Swift

或许，你现在就是一个iOS程序员，你对Objective-C很熟悉，对iOS开发也很熟悉，然而，苹果公司在iOS 8之后推出了Swift语言。那么，如何才能快速地从Objective-C过渡到Swift呢？

本章我们将重点讲解如何从Objective-C过渡到Swift，如何利用现有的Objective-C工程调用Swift代码，以及如何利用Swift工程调用以前写好的Objective-C代码。

18.1 选择语言

在苹果公司的Swift语言出现之前，开发iOS或Mac OS X应用主要使用Objective-C语言，此外还可以使用C和C++语言，但是UI部分只能使用Objective-C语言。

Swift语言出现后，iOS程序员有了更多的选择。在苹果社区里，有很多人在讨论Swift语言以及Objective-C语言的未来，人们关注的重点是Swift语言是否能够完全取代Objective-C语言。然而在我看来，苹果公司为了给程序员提供更多的选择，会让这两种语言并存。既然是并存，我们就有4种方式可以选择：

- 采用纯Swift的改革派方式；
- 采用纯Objective-C的保守派方式；
- 采用Swift调用Objective-C的左倾改良派方式；
- 采用Objective-C调用Swift的右倾改良派方式。

本书一直在介绍纯Swift的方式，而纯Objective-C的方式超出了本书的范围，后两种方式是本章的重点，无论是Swift调用Objective-C，

还是Objective-C调用Swift，我们都需要做一些工作。

18.2 Swift调用Objective-C

Swift调用Objective-C需要一个名为“<工程名>-Bridging-Header.h”的桥接头文件，如图18-1所示。桥接头文件的作用是为Swift调用Objective-C对象搭建一个桥，它的命名必须是“<工程名>-Bridging-Header.h”，我们需要在桥接头文件中引入Objective-C头文件，而且桥接头文件是需要管理和维护的。

图 18-1 Swift调用Objective-C与桥接头文件

18.2.1 创建Swift的iOS工程

为了能够更好地介绍混合搭配调用，我们首先创建一个基于Swift的iOS工程。具体参考14.3节创建工程，删除AppDelegate.swift、ViewController.swift和Main.storyboard这3个文件。

参考14.3节创建一个main.swift文件，注意在文

件创建过程中，Xcode会弹出一个是否创建桥接头文件对话框，如图18-2所示，这里要选择Yes，这样就会在创建swift文件的同时也创建一个桥接头文件，如果我们选择No也可以在以后创建，就是有些麻烦。

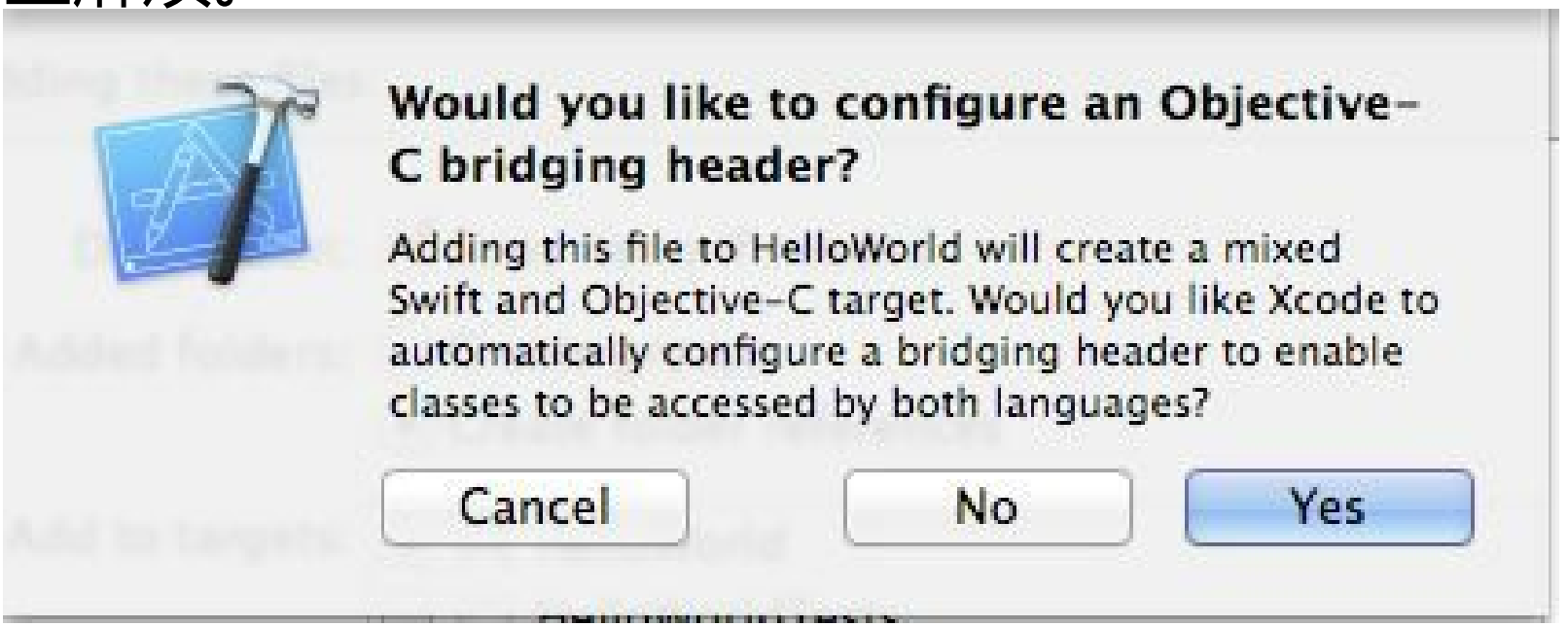


图 18-2　创建桥接头文件

完成上述操作的Xcode界面如图18-3所示。

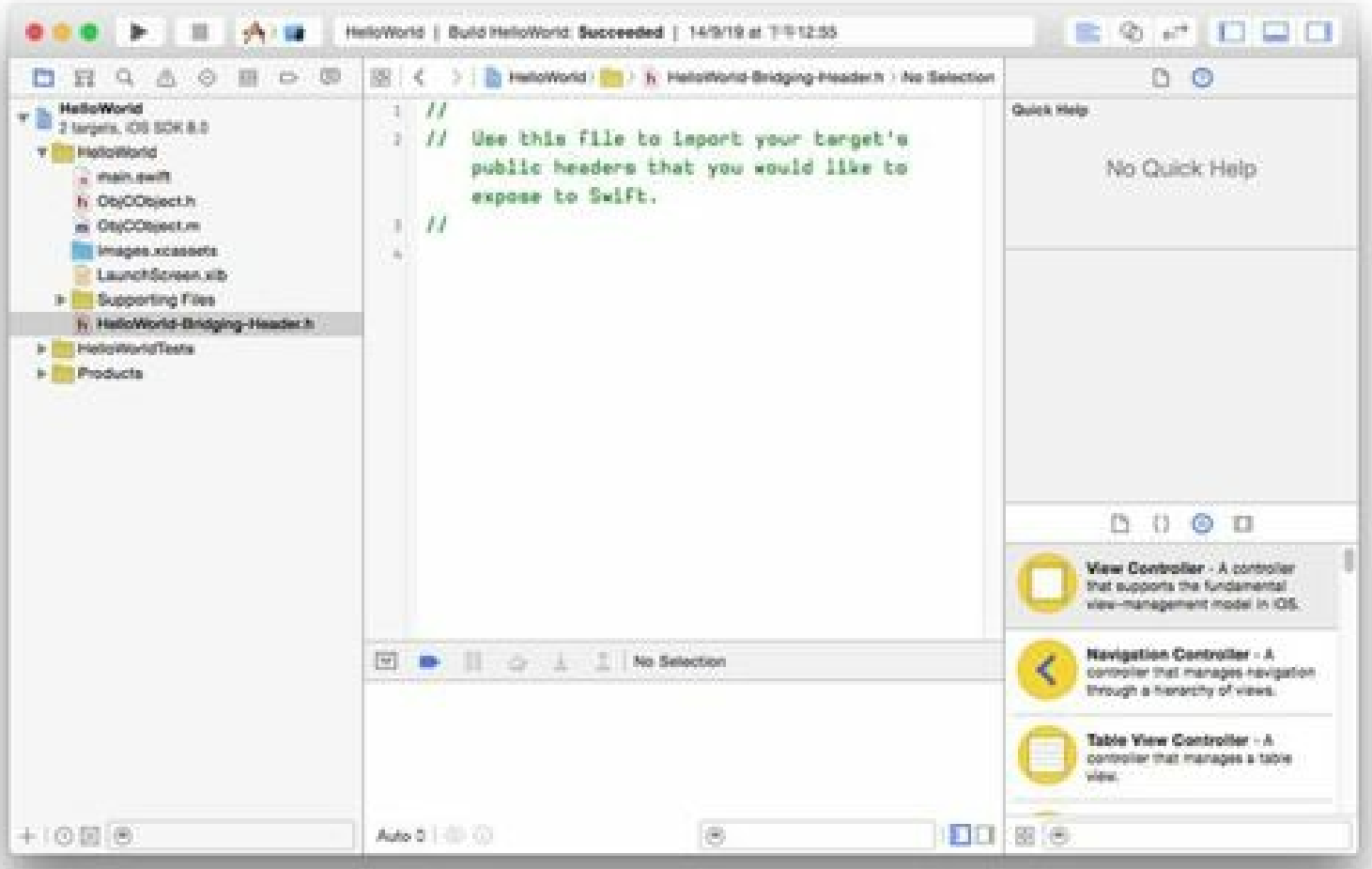


图 18-3　Xcode界面

18.2.2　在Swift工程中添加Objective-C类

我们刚刚创建了Swift的工程，还需要调用其他Objective-C类来实现某些功能，需要添加Objective-C类到Swift工程中。具体过程是，右键选择HelloWorld组，然后选择菜单中的“New File...”弹出新建文件模板对话框，如图18-4所示，选择“iOS　→Source→Cocoa Touch Class”。

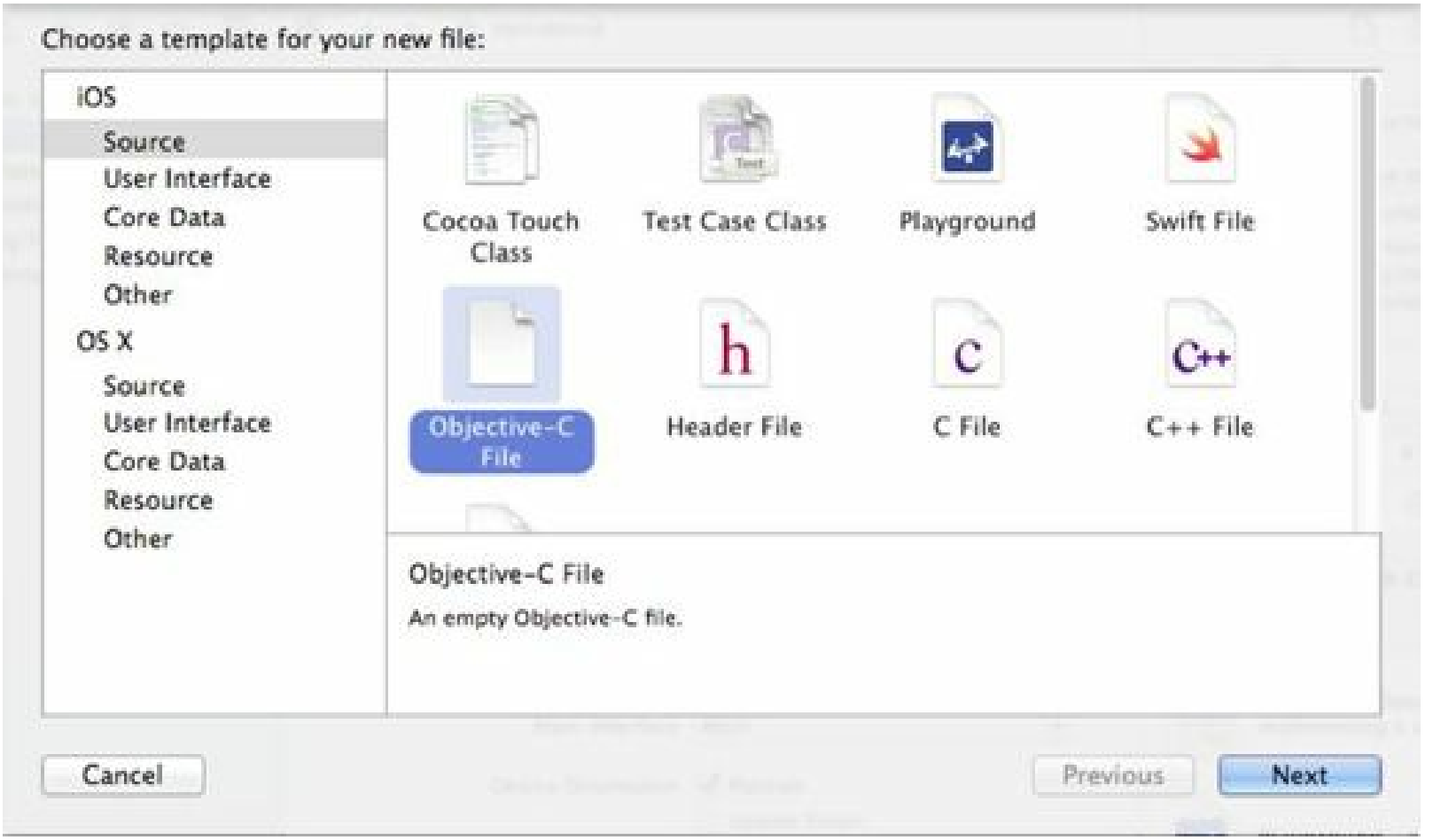


图 18-4　新建文件模板

接着单击“Next”按钮，随即出现如图18-5所示的界面。在Class中输入“ObjCObject”，在Language中选择Objective-C，其他的选项保持默认值就可以了。

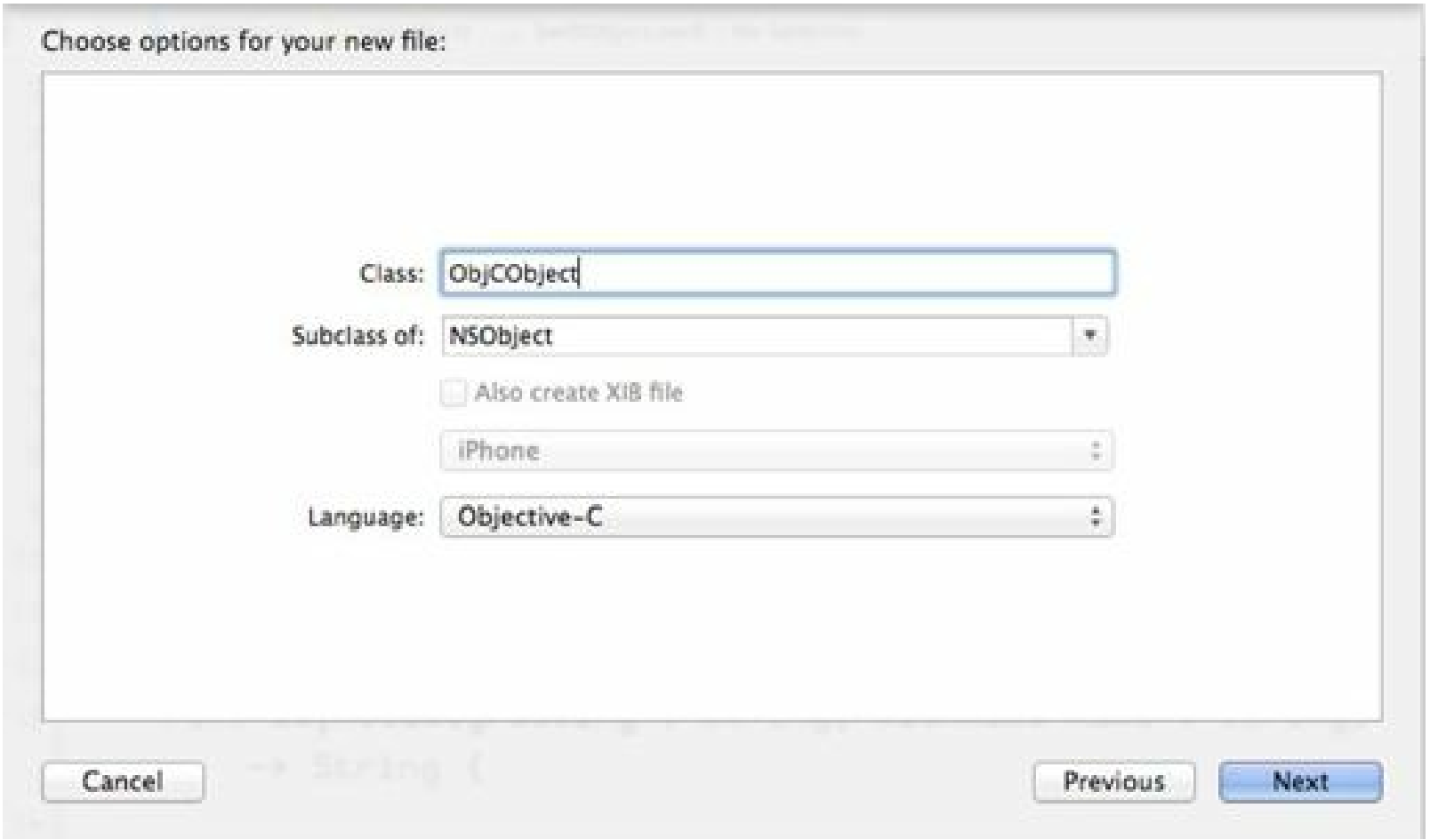


图 18-5　新建Objective-C类

相关选项设置完成后，单击“Next”按钮，进入保存文件界面，根据提示选择存放文件的位置，然后单击“Create”按钮创建Objective-C类。

18.2.3　调用代码

Objective-C的ObjCObject创建完成后，我们会在Xcode工程中看到新增加的两个文件ObjCObject.h和ObjCObject.m。本书并不打算过多地介绍Objective-C语言，但为了更好地理解Swift与Objective-C的互相调用，我们还是简单地解释一下Objective-C代码。

ObjCObject.h代码如下：

```
#import <Foundation/Foundation.h>
①

@interface ObjCObject : NSObject
②

-(NSString*)sayHello:(NSString*)greeting
withName: (NSString*)name;        ③

@end
```

ObjCObject.h文件是Objective-C的头文件，我们在这里定义类，声明类的成员变量和方法。第①行代码引入Foundation框架的头文件。第②行代码定义类ObjCObject，它继承自NSObject父类。NSObject类是所有Objective-C的根类。第③行代码声明了实例方法sayHello:withName:，它有两个参数greeting和name。

ObjCObject.m代码如下：

```
#import "ObjCObject.h"
①
```

```
@implementation ObjCObject

-(NSString*)sayHello:(NSString*)greeting
withName: (NSString*)name          ②
{
    NSString *string =
[NSStringstringWithFormat:@"Hi,%@
%@.",name,greeting];
    return string;
}

@end
```

上述代码第①行引入ObjCObject.h头文件，第②行代码定义sayHello: withName:方法。

下面我们再来看看Swift调用文件main.swift的代码：

```
import Foundation
①

var obj : ObjCObject = ObjCObject()
②
var hello = obj.sayHello("Good morning",
withName:"Tony")                    ③
```

```
println(hello)
④
```

上述代码第①行的`import Foundation`语句是引入Foundation框架，类似于Objective-C的`#import <Foundation/Foundation.h>`语句，关于Foundation框架我们会在下一章介绍。

第②行代码声明并实例化`ObjCObject`类的实例`obj`。`ObjCObject`就是Objective-C里定义的`ObjCObject`。

第③行代码调用`ObjCObject`类的`sayHello:withName:`方法。要注意Swift调用时的方法名和参数与Objective-C中该方法的方法名和参数的对应关系，如图18-6所示。

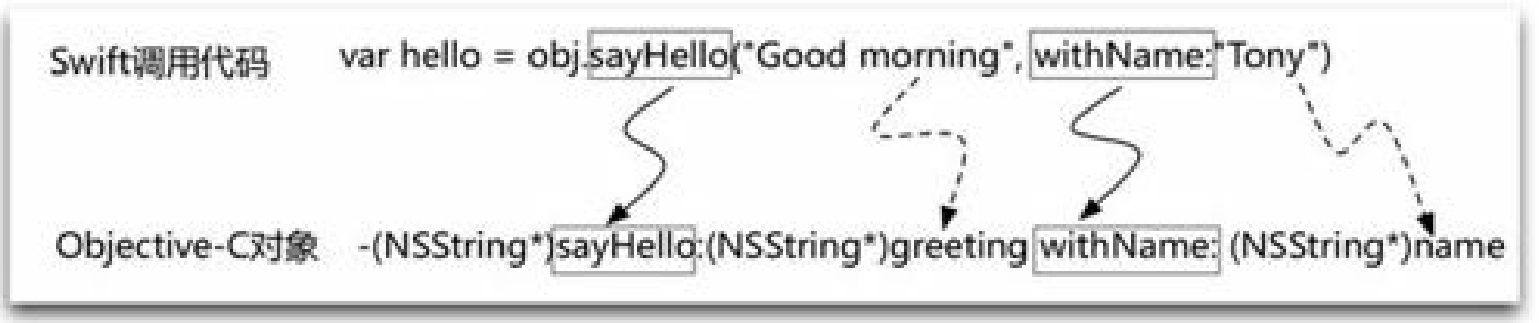


图 18-6　Objective-C与Swift调用方法和参数对应关系

第④行代码打印输出结果，输出结果如下：

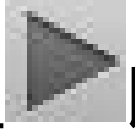
```
Hi,Tony Good morning.
```

要使Swift能够调用Objective-C，还必须在桥接头文件中引入Objective-C头文件。HelloWorld-Bridging-Header.h代码如下：

```
#import "ObjCObject.h"
```

很简单，只有一行代码，如果还有其他的Objective-C头文件，都需要在此引入。

这样就实现了在Swift中调用Objective-C代码，我们可以借助于这样的调用充分地利用已有的Objective-C文件，减少重复编写代码，提供工作效率。

至此，整个工程创建完毕。点击左上角的“运行”按钮  即可查看运行结果，运行结果输出到日志窗口。运行和查看日志窗口的操作细节可以参考14.3节，这里不再赘述。

18.3 Objective-C调用Swift

如果已经有了一个老的iOS应用，它是使用Objective-C编写的，而它的一些新功能需要采用Swift来编写，这时就可以从Objective-C调用Swift。

Objective-C调用Swift时不需要桥接头文件，而是需要Xcode生成的头文件。这种文件由Xcode生成，不需要我们维护，对于开发人员也是不可见的。如图18-7所示，它能够将Swift中的类暴露给Objective-C，它的命名是：<工程名>-swift.h。我们需要将该头文件引入到Objective-C文件中，而且Swift中的类需要声明为`@objc`。

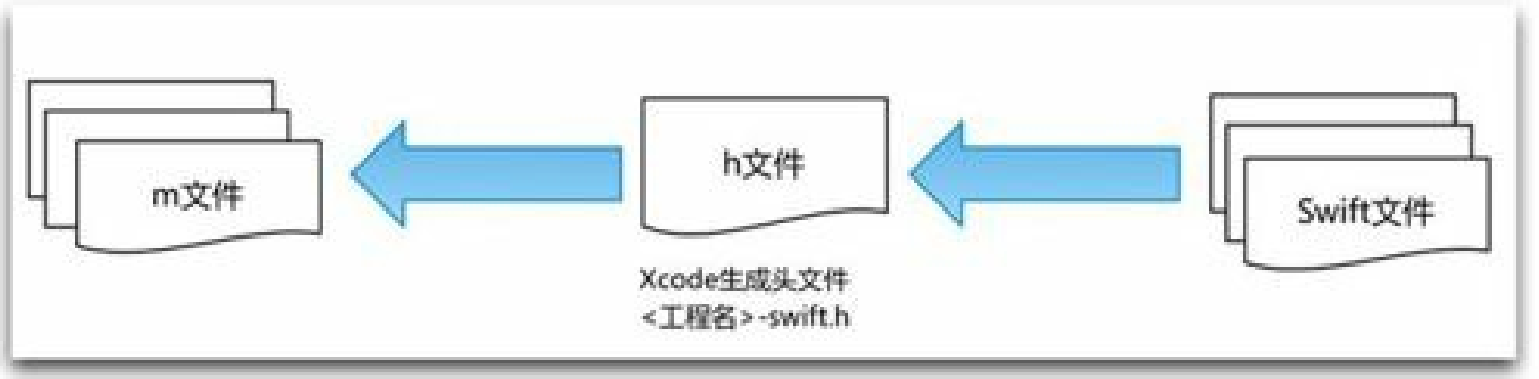


图 18-7 Objective-C调用Swift与Xcode生成头文件

18.3.1 创建Objective-C的iOS工程

为了能够更好地介绍混合搭配调用，我们首先

创建一个Objective-C的iOS工程。

启动Xcode 6，然后单击File→New→Project菜单，在打开的Choose a template for your new project界面中选择“iOS→Application→Single View Application”工程模板（如图18-8所示）。

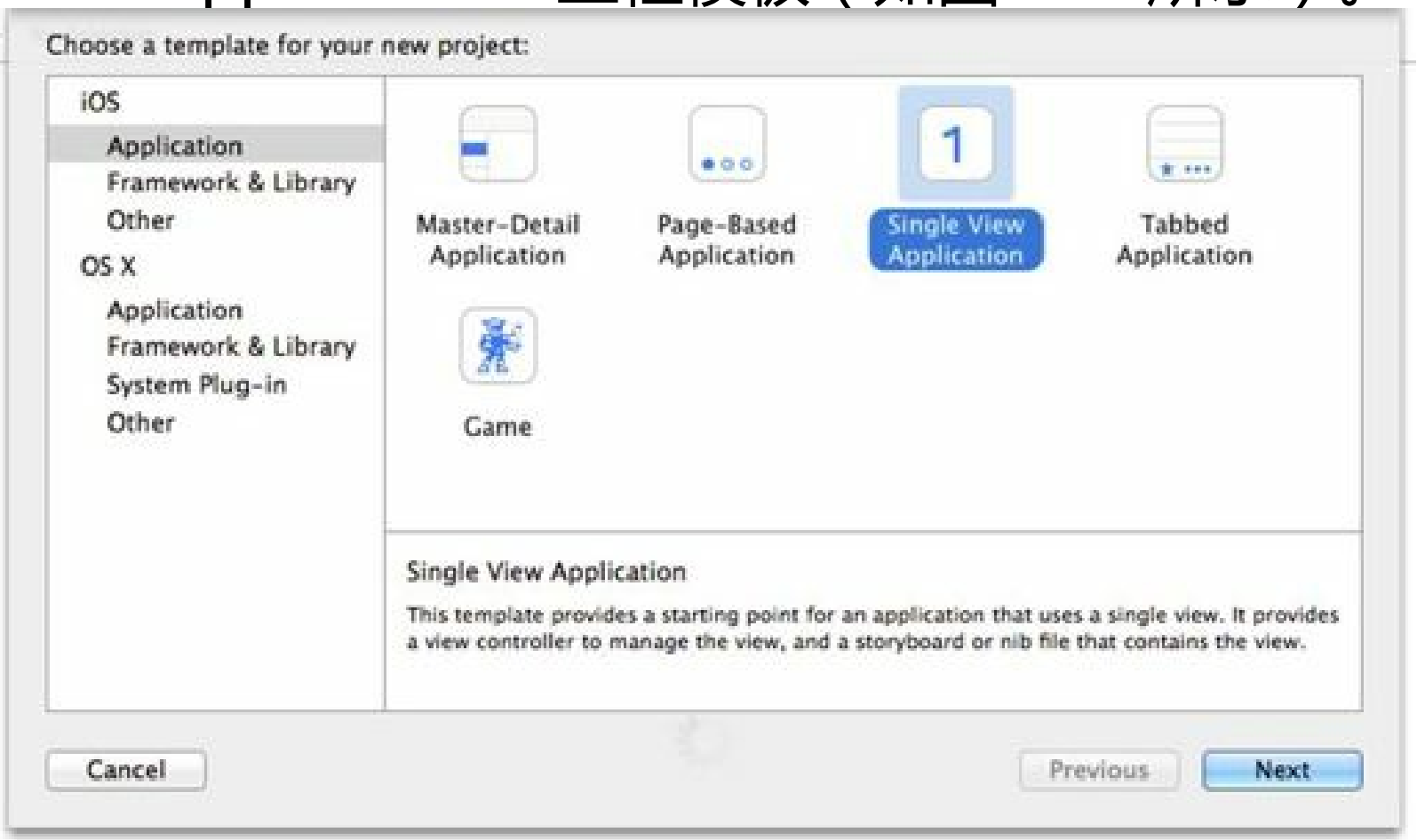


图 18-8　选择工程模板

接着单击“Next”按钮，随即出现如图18-9所示的界面。

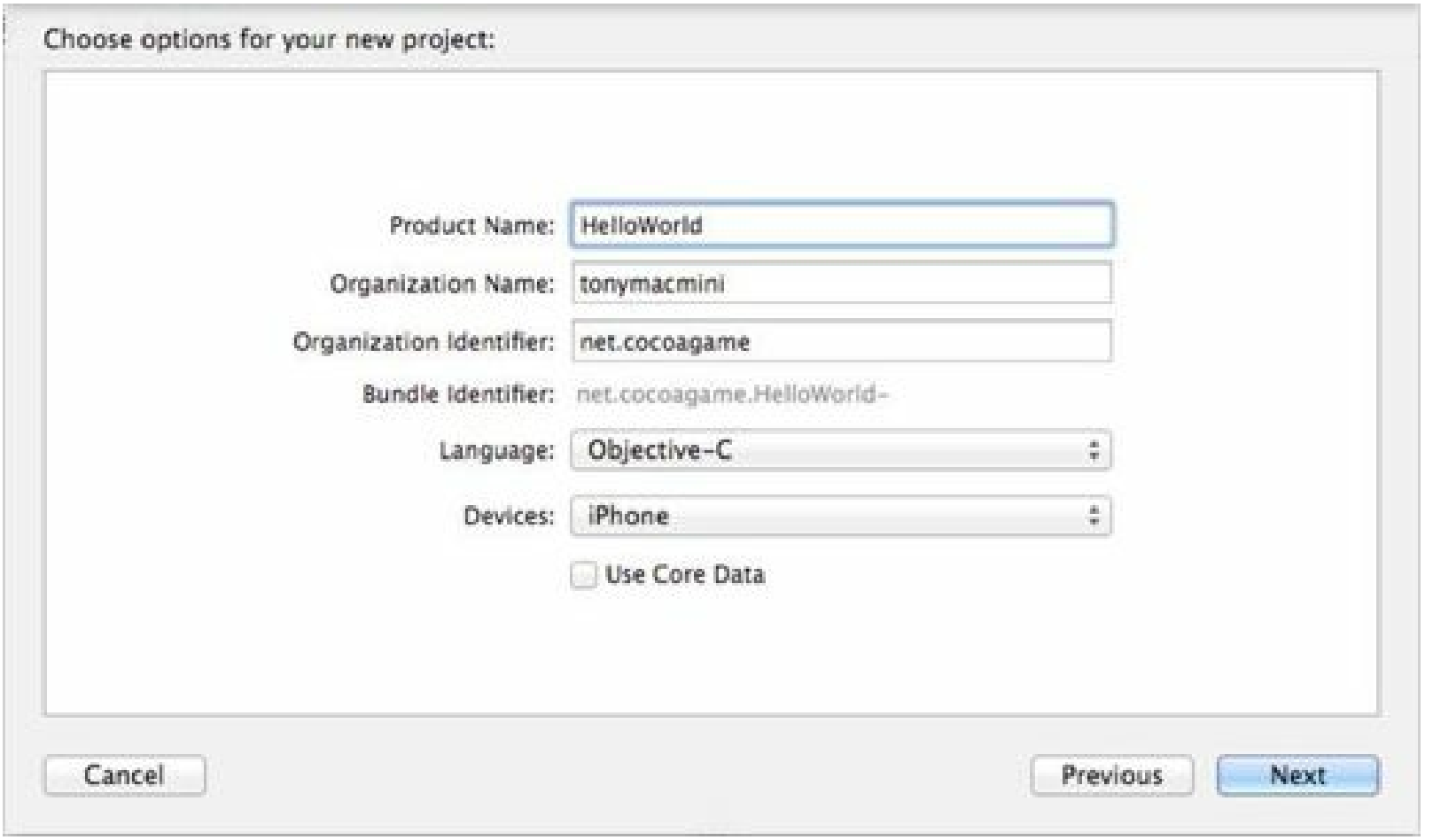


图 18-9　新工程中的选项

在Product Name中输入“HelloWorld”，在Language中选择Objective-C，其他的项目可以结合自己的实际情况输入内容。相关的工程选项设置完成后，单击“Next”按钮，后面的步骤与18.2.1节类似，在此不再赘述。

创建成功后的界面如图18-10所示，然后选择图18-10中的5个文件删除，文件删除后工程中的main.m文件会有编译错误，如图18-11所示，这个错误我们先不用考虑它，我们会在下一节修改main.m文件。

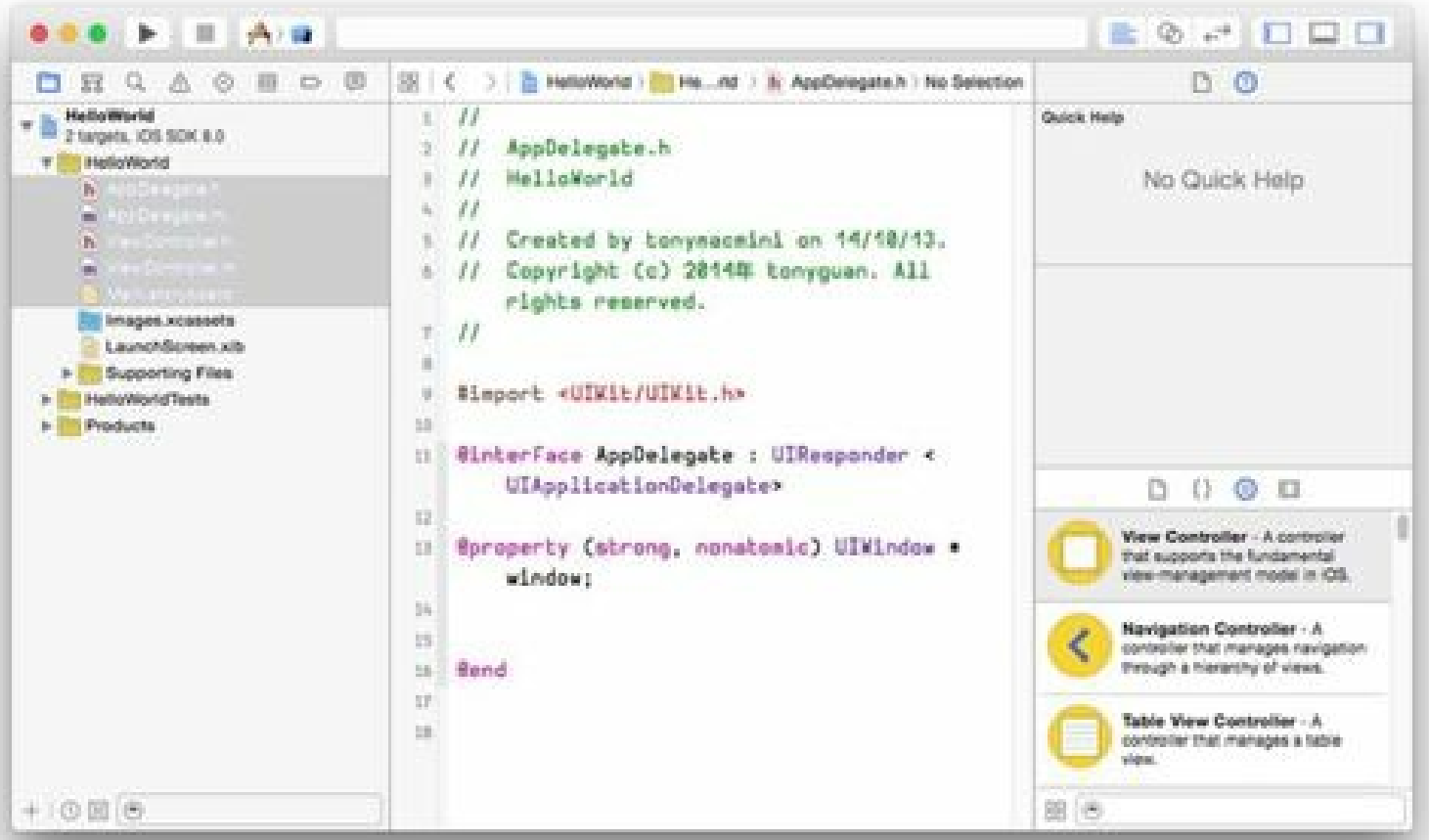


图 18-10　新建的Objective-C工程

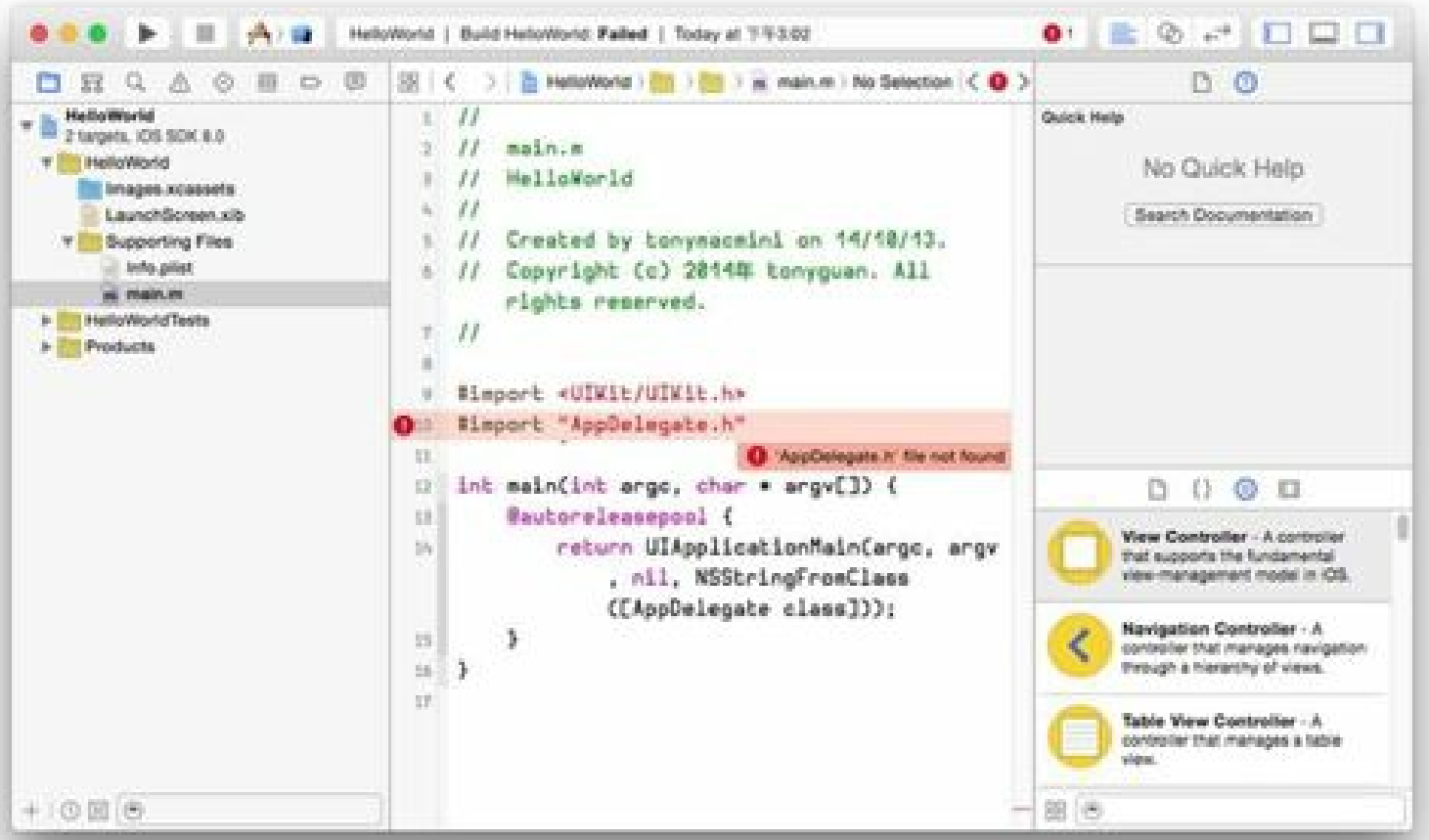


图 18-11　main.m编译错误

18.3.2　在Objective-C工程中添加Swift类

我们刚刚创建了Objective-C的工程，需要添加Swift类到工程中。具体过程是，右键选择HelloWorld组，选择菜单中的“New File...”弹出新建文件模板对话框。如图18-12所示，选择iOS X→Source→Cocoa Touch Class。

注意　这里我们并没有选择Swift File，是因为需要创建的Swift类是基于Cocoa Touch

框架的。

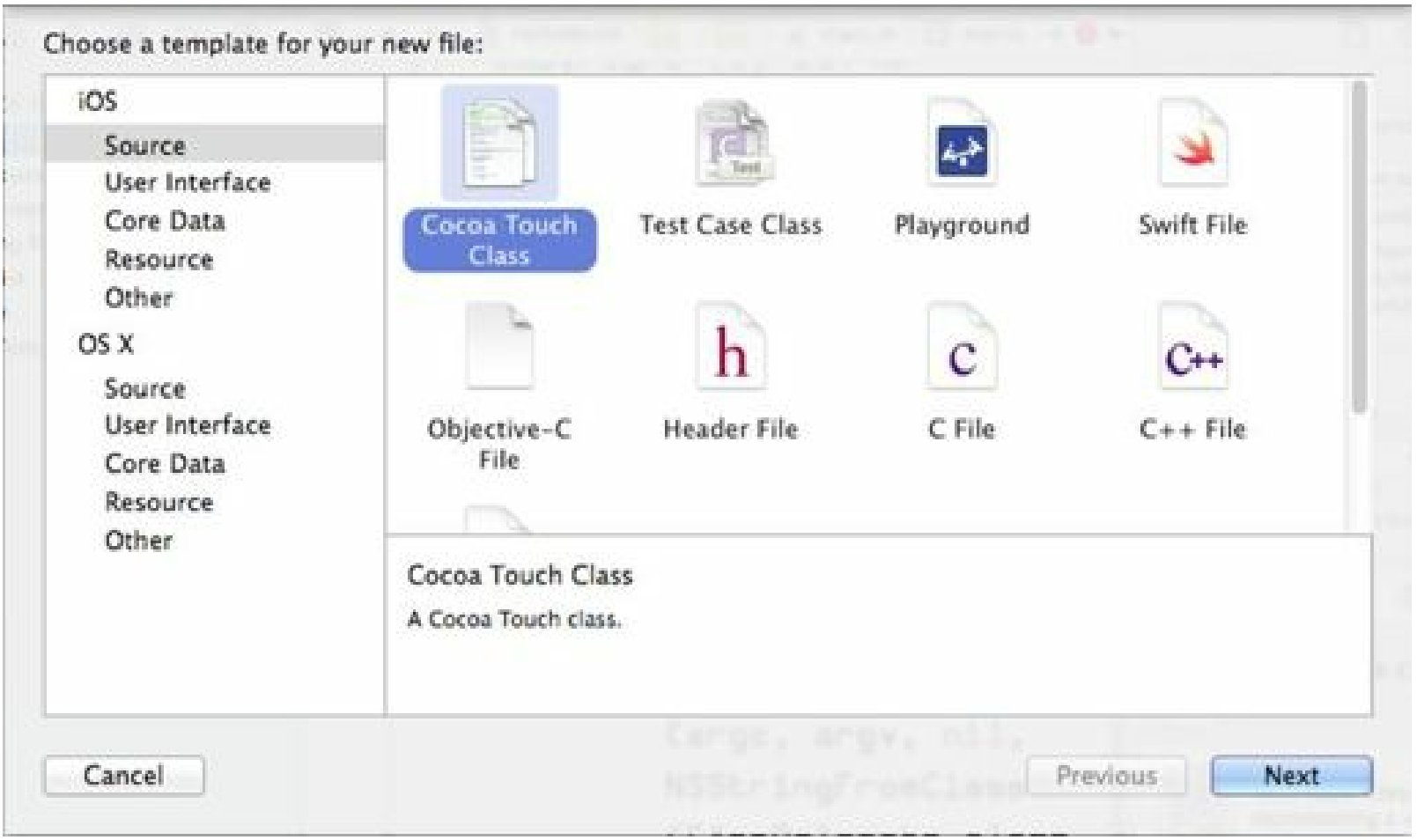


图 18-12　新建文件模板

接着单击“Next”按钮，随即出现如图18-13所示的界面。在Class中输入“SwiftObject”，在Subclass of中选择NSObject，这个选项可以让生成的Swift类继承于NSObject。在Language中选择Swift。

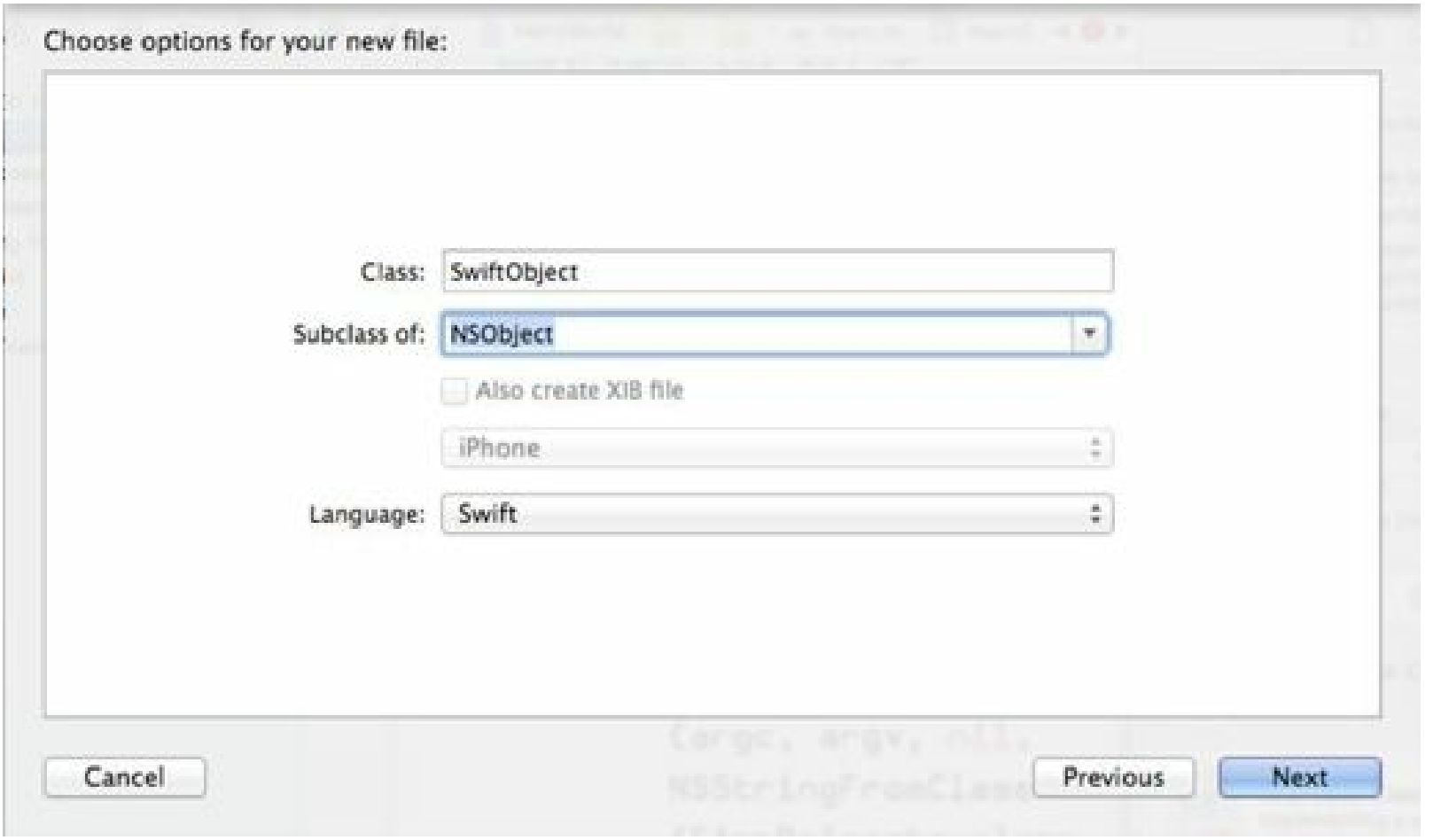


图 18-13　新建Swift类

相关选项设置完成后，单击“Next”按钮，进入保存文件界面，根据提示选择存放文件的位置，然后单击“Create”按钮创建Swift类。如果工程中没有桥接头文件，在创建过程中，Xcode也会提示我们是否添加桥接头文件，我们需要选择添加。

以上操作成功后在Xcode工程中生成了SwiftObject.swift文件，Xcode界面如图18-14所示。

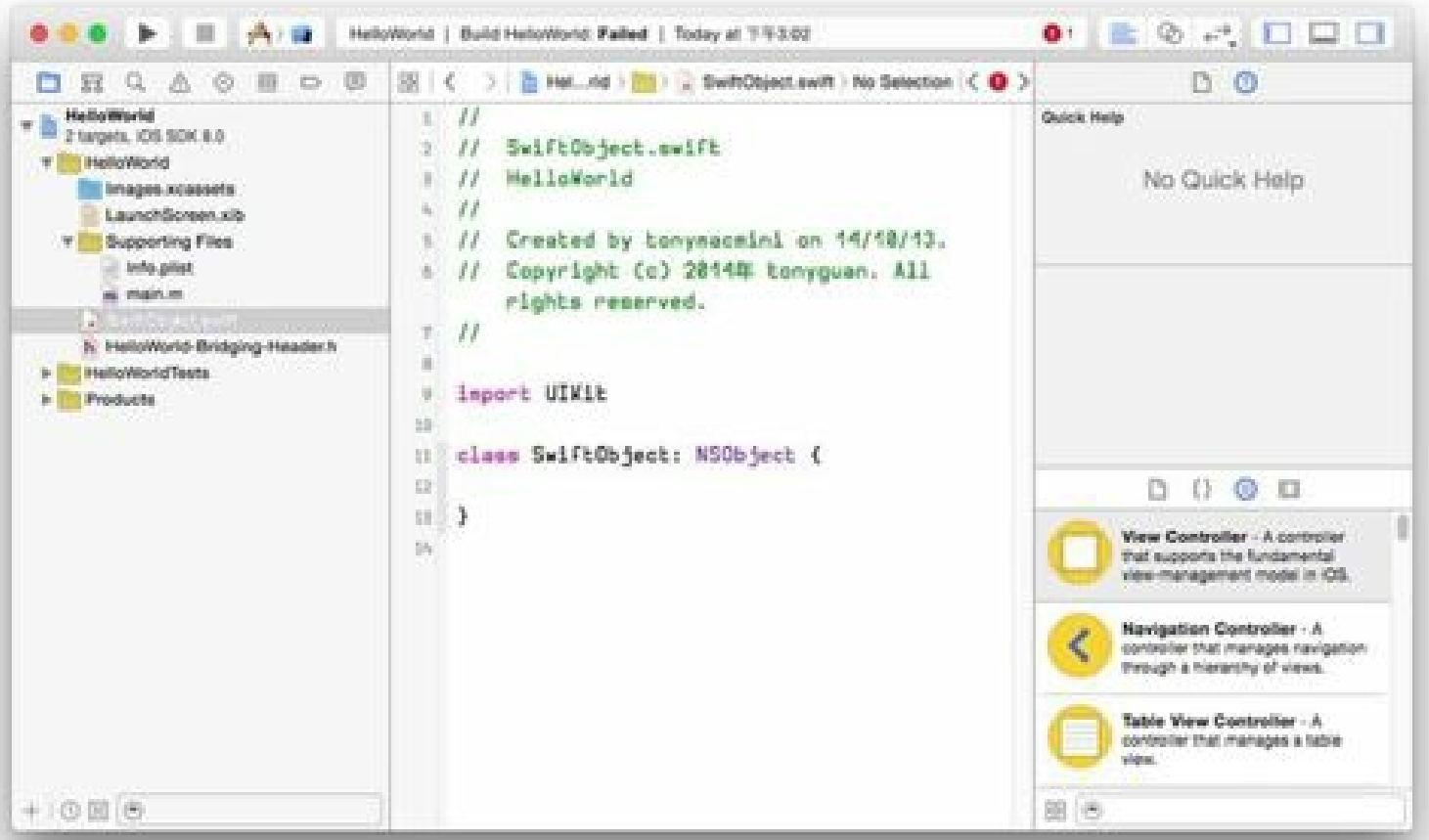


图 18-14　Xcode界面

18.3.3　调用代码

Swift的SwiftObject创建完成后，我们会在Xcode工程中看到新增加的SwiftObject.swift文件。我们在SwiftObject.swift中编写如下代码：

```
import Foundation
①

@objc class SwiftObject: NSObject {
②
```

```
    func sayHello(greeting : String, withName
name : String) -> String {  ③

        var string = "Hi," + name
        string += greeting

        return string
    }
}
```

上述代码第①行引入了Foundation框架的头文件。第②行代码定义SwiftObject类，SwiftObject类继承自NSObject类。另外，我们在类前面声明为@objc，@objc所声明的类能够被Objective-C访问，@objc还可以修饰属性。

第③行代码定义了sayHello方法，它有两个参数，第一个参数不需要指定外部参数名，第二个参数（除了第一个以后所有的参数）需要指定外部参数名，例如withName是name参数的外部参数名。这是为了方便在Objective-C中调用。

下面看Objective-C端的代码，main.m文件代码如下：

```
#import <Foundation/Foundation.h>
#import "HelloWorld-swift.h"
①

int main(intargc, const char * argv[]) {

    SwiftObject sobj = [[SwiftObject alloc]
init];                              ②
    NSString hello = [sobj sayHello:@"Good
morning" withName:@"Tony"];  ③

    NSLog(@"%@",hello);
④

    return 0;
}
```

上述代码第①行引入头文件HelloWorld-swift.h，它是Objective-C调用Swift对象所必需的，它的命名规则是“<工程名>-swift.h”。

第②行代码实例化SwiftObject对象，SwiftObject是Swift中定义的类。第③行代码调用SwiftObject的sayHello方法，它在Objective-C中被调用时的方法和参数命名与SwiftObject的方法和参数之间的对应关系如

图18-15所示。

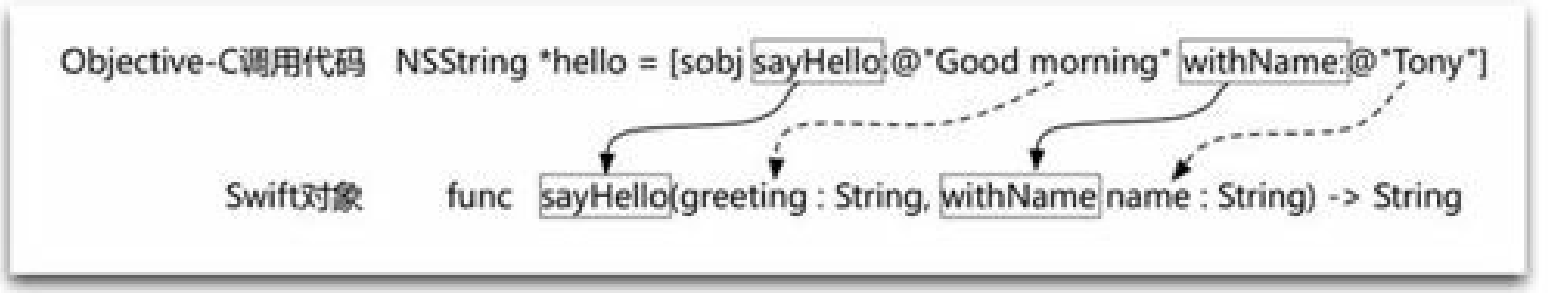


图 18-15　Swift与Objective-C调用方法和参数对应关系

第④行代码`NSLog(@"%@",hello)`用于输出结果，输出结果如下：

```
2014-07-05 14:25:43.879 HelloWorld[3266:303]
Hi,TonyGood morning
```

这样就实现了在Objective-C中调用Swift代码，我们可以借助于这样的调用充分利用已有的Swift文件，减少重复编写代码，提高工作效率。

18.4　本章小结

通过对本章内容的学习，我们了解了如何从Objective-C过渡到Swift，以及如何利用Swift调用Objective-C。

18.5 同步练习

1. 下列说法正确的是（　　）。

A. Swift调用Objective-C时需要桥接头文件<工程名>-Bridging-Header.h。

B. Objective-C调用Swift时需要桥接头文件<工程名>-Bridging-Header.h。

C. Swift调用Objective-C时需要Xcode生成头文件<工程名>-swift.h。

D. Objective-C调用Swift时需要Xcode生成头文件<工程名>-swift.h。

2. 下列说法正确的是（　　）。

A. 桥接头文件命名规则是：<工程名>-Bridging-Header.h

B. 桥接头文件命名规则是：<工程名>-swift.h

C. Xcode生成头文件命名规则是：<工程名>-Bridging-Header.h

D. Xcode生成头文件命名规则是：<工程名>-swift.h

3. 下列说法正确的是（　　）。

A. 桥接头文件需要程序员管理和维护。

B. Xcode生成头文件需要程序员管理和维护。

C. 桥接头文件是由Xcode管理生成和维护的，对程序员是不可见的。

D. Xcode生成头文件是由Xcode管理生成和维护的，对程序员是不可见的。

4. 参考书中案例，编写一个由Swift调用Objective-C的案例。

5. 参考书中案例，编写一个由Objective-C调用Swift的案例。

第 19 章　使用Foundation框架

学习过Objective-C语言的读者都应该知道Foundation框架[1]，它是开发Mac OS X或iOS应用时都会使用的最基本的框架。

[1]框架就是一些库，类似于C语言中的标准库。

对于Mac OS X开发，会使用Cocoa框架，它是一种支持应用程序提供丰富用户体验的框架，它实际上由Foundation和Application Kit（AppKit）框架组成；对于iOS开发，会使用Cocoa Touch框架，它实际上由Foundation和UIKit框架组成。

AppKit和UIKit框架都是与窗口、按钮、列表等相关的类。Foundation是Mac OS X和iOS应用程序开发的基础框架，它包括了一些基本的类，如数字、字符串、数组、字典等。

我们上面所说的框架是否可以在Swift中使用呢？答案是肯定的，苹果搭建一个“桥”使得Swift只要引入这些框架，就可以使用它们，名称调用规范遵守第18章介绍的`sayHello`方法。

使用这些框架有很多优势，有些Swift语言中没有解决的问题，可以通过这些框架提供的类来解决。事实上，大家以后学习iOS都会用到UIKit框架中的一些类，这些类都是通过Swift语言调用

Objective-C对象实现的。Foundation框架是最基础的框架，我们很多地方都会用到它，因此本章重点介绍使用Foundation框架。

19.1 数字类NSNumber

在Objective-C语言中有一些基本数据类型[2]：int、char、float和double，但是它们都不是类，不具有方法、成员变量和属性以及面向对象的特征。为了实现“一切都是对象”的承诺，因此在Foundation框架中使用NSNumber类来封装这些数字类型。这样数字就具有面向对象的基本特征了。

[2]基本数据类型int、char、float和double事实上是由C语言提供的，它们可以在C、C++和Objective-C中使用。

19.1.1 获得NSNumber实例

不仅是NSNumber类，Foundation框架中几乎所有的类，都有两种获得实例的方式：一种是通过构造器创建，另一种是通过工厂设计模式创建。

如图19-1所示，打开NSNumber的API帮助文档，单击菜单栏中的“显示目录”按钮，在页面的左边显示目录。其中以number开头的方法是静态工厂创建方法，它通过工厂设计模式创建NSNumber对象获得实例；而以init开头的方法构造器，则通过构造器创建NSNumber对象并初始化成员获得实例。

图 19-1　NSNumber API帮助文档

提示　静态工厂创建方法命名是有规律的，即把类名去掉NS前缀，小写后作为方法的开头，例如NSString的静态工厂创建方法`class func stringWithUTF8String(_ bytes: CString) -> Self!`。该方法是通过一个UTF8字符串创建NSString字符串对象，它采用的命名方法是使用string开头。

下面看示例代码：

```
import Foundation
①

var intSwift = 80

var intNumber1  = NSNumber(integer: intSwift)
②
var intNumber2  = NSNumber(int: CInt(intSwift))
③

var floatNumber1 = NSNumber(double: 80.00)
④

let myint = intNumber1.intValue
⑤
let myfloat = floatNumber1.floatValue
⑥
```

使用Foundation框架需要通过第①行的import Foundation语句引入框架。代码第②行、第③行和第④行使用NSNumber构造器创建对象，也就是NSNumber API中的init开头的方法构造器，例如我们打开NSNumber API中-

initWithInteger:构造器，如图19-2所示，在Foundation框架中以init开头的方法构造器就是这样的，不同的语言语法表现形式不同。init(integer value: Int)构造器调用时就需要写成NSNumber(integer: intSwift)的形式，integer是外部参数名。

图 19-2 NSNumber API中-initWithInteger:构造器

代码第③行、第⑥行和第⑦行使用静态工厂创建对象。第⑧行和第⑨代码是调用属性获得基本数据类型数值。

第④行和第⑦行分别使用了CInt()和

CDouble()函数，它们将数据类型转换为CInt和CDouble类型。在Swift中有一些数据类型与Objective-C类型中的基本数据类型相对应。表19-1是Objective-C基本数据类型和Swift类型对应关系表。

表19-1　类型对应关系

Objective-C类型	Swift类型
bool	CBool
char, signed char	CChar
unsigned char	CUnsignedChar
short	CShort
unsigned short	CUnsignedShort
int	CInt
unsigned int	CUnsignedInt
long	CLong
unsigned long	CUnsignedLong
long long	CLongLong
unsigned long long	CUnsignedLongLong
wchar_t	CWideChar
char16_t	CChar16

char32_t	CChar32
float	CFloat
double	CDouble

提示 Swift中的转换函数就是该类型加上括号，如CInt类型的转换函数就是CInt()。

19.1.2 NSNumber对象的比较

数字的比较是非常常见的操作，那么在比较两个NSNumber对象大小的时候，可以转化成为基本数据类型比较，当然可以使用NSNumber的方法比较，这就是对象的优势了。

与比较相关方法有isEqualToNumber和compare，这两个方法的定义如下：

```
func isEqualToNumber(_ aNumber: NSNumber!) -
>Bool
func compare(_ aNumber: NSNumber!) -
>NSComparisonResult
```

isEqualToNumber只是比较是否相等，

而compare方法可以比较是否相等、大于和小于，它的返回值是NSComparisonResult枚举类型。NSComparisonResult枚举类型定义如下：

```
enum NSComparisonResult : Int {
    case OrderedAscending          //升序，前一个数比后一个数小
    case OrderedSame               //两个数相等
    case OrderedDescending         //降序，前一个数比后一个数大
}
```

下面看一个示例：

```
import Foundation

var intSwift = 89
var intNumber  = NSNumber(integer: intSwift)
var floatNumber = NSNumber(double: 80.00)

if intNumber.isEqualToNumber(floatNumber) == true  {
    NSLog("相等")
} else {
    NSLog("不相等")
}
```

```
switch intNumber.compare(floatNumber) {
    case.OrderedAscending:
        NSLog("第一个数小于第二个数")
    case .OrderedDescending:
        NSLog("第一个数大于第二个数")
    case .OrderedSame:
        NSLog("第一个数等于第二个数")
}
```

输出结果如下：

```
2014-07-05 21:08:10.089
PlaygroundStub_OSX[7052:303] 不相等
2014-07-05 21:08:10.092
PlaygroundStub_OSX[7052:303] 第一个数大于第二个数
```

上述代码比较简单，基本不需要解释，但是需要注意的是NSLog函数是Foundation框架中的函数，也可以在Swift中使用，它的用途是日志输出，与Swift的println类似。

19.2 字符串类

在Foundation框架中，字符串类有两种：不可变字符串类NSString和可变字符串类NSMutableString。NSString是定义固定大小的字符串，NSMutableString可对字符串做追加、删除、修改、插入和拼接等操作，而不会产生新的对象。

19.2.1 NSString类

NSString是不可变字符串类，如果对字符串进行拼接等操作会产生新的对象。创建NSString对象的方法与NSNumber类似，有两种方式，这里不再赘述。

NSString有很多方法，下面总结的是常用属性和方法。

- +stringWithString: 静态工厂创建方法创建NSString对象，NSString构造方法还有很多。
- -length 返回Unicode字符的长度属性。
- -stringByAppendingString: 实

现了字符串的拼接，这个方法会产生下一个新的对象。

- -isEqualToString: 较两个字符串是否相等。
- -compare: 比较两个字符串大小。
- -uppercaseString 换成为大写属性。
- -lowercaseString 换成为小写属性。
- -substringToIndex: 可以获得字符串的前x个字符串。
- -substringFromIndex: 可以截取x索引位置到尾部字符串。
- -rangeOfString: 字符串查找。

提示 由于上述方法或属性是按照Objective-C语言命名的，其中方法或属性前面的“-”表示实例方法或属性，而“+”表示静态方法或属性。“:”结尾说明是方法名，而且后面有参数。

下面看NSString字符串示例代码：

```
var str1 : NSString = "aBcDeFgHiJk"
var str2 : NSString = "12345"
var res : NSString
var compareResult : NSComparisonResult
var subRange : NSRange
①

//字符个数
NSLog("字符串str1长度: %i",str1.length)
//输出：11                          ②

//复制字符串到res
res = NSString.stringWithString(str1)
③
NSLog("复制: %@", res)
//输出：aBcDeFgHiJk                 ④

//复制字符串到str1尾部
str2 = str1.stringByAppendingString(str2)
⑤
NSLog("连接字符串: %@", str2)
//输出：aBcDeFgHiJk12345

//测试字符串相等
if str1.isEqualToString(res) == true  {
    NSLog("str1 == res")
} else {
    NSLog("str1 != res")
}
```

```
//输出：str1 == res

//测试字符串<> ==
compareResult = str1.compare(str2)
switch compareResult {
    case .OrderedAscending:
        NSLog("str1 < str2")
    case .OrderedSame:
        NSLog("str1 == str2")
    Default:
        NSLog("str1 > str2")
}
//输出：str1 < str2

res = str1.uppercaseString
NSLog("大写字符串：%@", res)
//输出：ABCDEFGHIJK

res = str1.lowercaseString
NSLog("小写字符串：%@", res)
//输出：abcdefghijk
NSLog("原始字符串： %@", str1)
//输出：aBcDeFgHiJk

//获得前三个数
res = str1.substringToIndex(3)
⑥
NSLog("字符串str1的前三个字符: %@",res)//输出：aBc
res = str1.substringFromIndex(4)
NSLog("截取字符串，从第索引4到尾部: %@",res)
```

```
⑦
//输出：截取字符串，从第索引4到尾部: eFgHiJk

var temp:NSString = str1.substringFromIndex(3)
    res = temp.substringToIndex(2)
⑧
NSLog("截取字符串，从第索引3到5: %@",res)
//截取字符串，从第索引3到5: De

//字符串查找
subRange = str2.rangeOfString("34")
⑨
if subRange.location == NSNotFound {
⑩
    NSLog("字符串没有找到")
} else {
    NSLog("找到的字符串索引 %i长度是 %i",
subRange.location, subRange.length)
⑪
}
//输出：找到的字符串索引 13 长度是 2
```

上述代码也比较简单，我们择要解释一下。其中代码第①行是声明NSRange的变量subRange。NSRange是一个结构体，描述一个范围，它有两个成员location（位置）和

length（长度）。

第②行代码是通过NSLog函数输出字符串str1的长度。在NSLog函数中可以格式化输出信息，如果要想输出整数，使用%i指定格式形式。

第③行代码是复制字符串到res变量，虽然NSString字符串类是不可变的，它可以重写构建其他的字符串对象，stringWithString方法是静态工厂方法。第④行代码是通过NSLog函数格式化字符串信息，使用%@指定格式形式。

第⑤行代码是复制字符串到str1尾部，stringByAppendingString方法可以拼接str1和str2产生一个新的对象，然后把新对象的引用又赋值给str2。

第⑥行代码substringToIndex(index)方法是从前往后截取字符串，截取前index个字符串。第⑦行代码substringFromIndex方法是从index位置截取字符串到尾部。如果想截取中间的字符串怎么办呢？第⑧行代码回答了这个问题，其实很简单，就是两个方法一起用。

字符串查找在字符串计算中也是经常使用的，其中代码第⑨行的subRange = str2.rangeOf

String("34")方法，可以查找"34"字符串在str2字符串中的位置，其返回值subRange是NSRange类型。关于NSRange类型我们在前文已经解释了，其中第⑩行代码是判断是否找到了目标字符串，如果location成员值等于NSNotFound常量说明没有找到。NSNotFound是Foundation框架中定义的常量。

19.2.2 NSMutableString类

NSMutableString是NSString的子类，有很多方法，下面总结常用的属性和方法。

- +stringWithCapacity: 静态工厂创建方法NSMutableString对象。
- -initWithCapacity: 构造器，通过指定容量构造NSMutableString对象，NSMutableString构造方法还有很多。
- -insertString:atIndex: 插入字符串，不会创建新的对象。
- -appendString: 追加字符串，不会

创建新的对象。

- -deleteCharactersInRange: 在一个范围内删除字符串，不会创建新的对象。
- -replaceCharactersInRange:withStr替换字符串，不会创建新的对象。

下面看NSMutableString字符串示例代码：

```
import Foundation

var str1 : NSString = "Objective C"
var search, replace : NSString

var mstr : NSMutableString
var substr : NSRange

//从不可变的字符创建可变字符串对象
mstr = NSMutableString.stringWithString(str1)
NSLog(" %@", mstr)
//输出： Objective C

//插入字符串
mstr.insertString(" Java", atIndex : 9)
NSLog(" %@", mstr)
```

```
//输出： Objective Java C

//具有连接效果的插入字符串
mstr.insertString(" and C++", atIndex :
mstr.length)
NSLog(" %@", mstr)
//输出： Objective Java C and C++

//字符串连接方法
mstr.appendString(" and C")
NSLog(" %@", mstr)
//输出： Objective Java C and C++ and C

//使用NSRange删除字符串
mstr.deleteCharactersInRange(NSMakeRange(16,
13))                                    ①
NSLog(" %@", mstr)
//输出： Objective Java CC

//查找字符串位置
substr = mstr.rangeOfString("string B and")
if substr.location != NSNotFound {
    mstr.deleteCharactersInRange(substr)
    NSLog(" %@", mstr)
}

//直接设置可变字符串
mstr.setString("This is string A ")
②
NSLog(" %@", mstr)
```

```
//输出： This is string A

mstr.replaceCharactersInRange(NSMakeRange(8,
8), withString: "a mutable string ")
NSLog(" %@", mstr)
//输出： This is a mutable string

//查找和替换
search = "This is "
replace = "An example of "

substr = mstr.rangeOfString(search)

if substr.location != NSNotFound {
    mstr.replaceCharactersInRange(substr,
withString: replace)
    NSLog(" %@", mstr)
    //输出： An example of a mutable string
}

//查找和替换所有的情况
search = "a"
replace = "X"
substr = mstr.rangeOfString(search)

while substr.location != NSNotFound {
    mstr.replaceCharactersInRange(substr,
withString: replace)
    substr = mstr.rangeOfString(search)
}
```

```
NSLog(" %@", mstr)
//输出： An exXmple of X mutXble string
```

上述代码第①行中使用NSMakeRange(16, 13)，NSMakeRange是一个函数，可以创建NSRange结构体，第一个参数是位置，第二个参数是长度。第②行代码mstr.setString("This is string A ")是重新设置字符串。

19.2.3 NSString与String之间的关系

在Swift中，使用字符串有可能会使用Foundation中的NSString和Swift中的String。下面我们介绍一下它们之间的关系。Swift在底层能够将String与NSString无缝地桥接起来，String可调用 NSString的全部API。

因为可以在String中使用NSString，很多String API不具有的功能可以通过调用NSString API实现，但是有些时候类型转换是必要的。

下面我们看一个使用String和NSString的示例代码：

```
import Foundation
①

let foundationString : NSString = "alpha bravo
charlie delta echo"                    ②

let swiftString : String = foundationString
③
let foundationString2 : NSString = swiftString
④

let swiftArray : Array =
swiftString.componentsSeparatedByString(" ")
⑤

let intString : NSString = "456"
⑥
let intValue = (intString as String).toInt()
⑦
```

要想使用NSString，需要引入Foundation或Cocoa。代码第①行是引入Foundation，第②行代码声明并初始化NSString字符串，第③行代码是将NSString字符串赋值给String字符串变量swiftString。在这个这个过程中，事实上发生了类型转换，但是我们不需要做任何事情。类似

地，第④行代码是将String字符串赋值给NSString字符串，这个过程中也发生了类型转换。

第⑤行代码调用了NSString的componentsSeparatedByString方法，该方法可以使用指定的字符分割字符串返回数组，本来返回的数组是Foundation框架中的NSArray，但是本例中是Array，这个过程中发生了从NSArray到Array的类型转换，这个问题我们将在19.3节详细讨论。

第⑥行代码声明并初始化NSString字符串，它是由数字组成的字符串，这种字符串可以转换为数字类型。第⑦行代码使用String的toInt方法进行转换，但是调用该方法需要String，因此需要使用as运算符将NSString强制类型转换为String。在大多数情况下，NSString和String之间不需要进行这种强制类型的转换。

19.3 数组类

在Foundation框架中，数组被封装成为类，数组有两种：NSArray不可变数组类和NSMutableArray可变数组类。

19.3.1 NSArray类

NSArray有很多方法，下面总结常用的属性和方法。

- -initWithArray: 构造器，通过指定Array参数构造NSArray对象。
- -count 返回当前数组的长度。
- -objectAtIndex: 按照索引返回数组中的元素。
- -contaiNSObject: 是否包含某一元素。

下面看一个NSArray数组的示例代码：

```
import Foundation

let weeksArray = ["星期一","星期二", "星期三", "星期四", "星期五", "星期六","星期日"]    ①
```

```
var weeksNames:NSArray = NSArray(Array:
weeksArray)
②

NSLog("星期名字")
NSLog("====   ====")

for vari = 0; i<weeksNames.count; i++ {
③
    var obj : AnyObject =
weeksNames.objectAtIndex(i)
④
    var day = obj as NSString
⑤
    NSLog("%i     %@", i, day)
}

for item : AnyObject in weeksNames {
⑥
    var day = item as NSString
⑦
    NSLog("%@", day)
}
```

上述代码第①行是声明并初始化Swift的Array数组类型常量weeksArray，第②行代码是通过

weeksArray常量创建并初始化NSArray数组类型对象weeksNames。

第③行代码是采用的for循环遍历weeksNames数组，第④行代码使用objectAtIndex(i)方法按照索引从数组中取出元素，取值的元素是AnyObject类型。第⑤行代码将AnyObject类型的obj变量转换为NSString，因为本来weeksNames数组中的元素就是NSString类型的。

第⑥行代码是采用for in循环遍历weeksNames数组，第⑦行代码var day = item as NSString是将元素转换为NSString类型。

输出结果如下：

```
2014-07-06 18:04:16.129
PlaygroundStub_OSX[2477:303] 星期名字
2014-07-06 18:04:16.132
PlaygroundStub_OSX[2477:303] ====   ====
2014-07-06 18:04:16.139
PlaygroundStub_OSX[2477:303] 0      星期一
2014-07-06 18:04:16.144
PlaygroundStub_OSX[2477:303] 1      星期二
2014-07-06 18:04:16.151
```

```
PlaygroundStub_OSX[2477:303] 2      星期三
2014-07-06 18:04:16.156
PlaygroundStub_OSX[2477:303] 3      星期四
2014-07-06 18:04:16.162
PlaygroundStub_OSX[2477:303] 4      星期五
2014-07-06 18:04:16.167
PlaygroundStub_OSX[2477:303] 5      星期六
2014-07-06 18:04:16.172
PlaygroundStub_OSX[2477:303] 6      星期日
2014-07-06 18:04:16.178
PlaygroundStub_OSX[2477:303] 星期一
2014-07-06 18:04:16.181
PlaygroundStub_OSX[2477:303] 星期二
2014-07-06 18:04:16.184
PlaygroundStub_OSX[2477:303] 星期三
2014-07-06 18:04:16.188
PlaygroundStub_OSX[2477:303] 星期四
2014-07-06 18:04:16.192
PlaygroundStub_OSX[2477:303] 星期五
2014-07-06 18:04:16.195
PlaygroundStub_OSX[2477:303] 星期六
2014-07-06 18:04:16.199
PlaygroundStub_OSX[2477:303] 星期日
```

19.3.2 NSMutableArray类

NSMutableArray是NSArray的子类，它有很多方法和属性，下面总结其常用的方法和属性。

- -addObject: 在数组的尾部追加一个元素。
- -insertObject:atIndex: 按照索引插入一个元素。
- -removeObjectAtIndex: 移除特定索引的元素。
- -removeObject: 移除特定元素。
- -initWithCapacity: 实例构造方法。

下面看NSArray数组的示例代码：

```
import Foundation

var weeksNames : NSMutableArray =
NSMutableArray(capacity: 3)
①

weeksNames.addObject("星期一")
②
weeksNames.addObject("星期二")
weeksNames.addObject("星期三")
weeksNames.addObject("星期四")
weeksNames.addObject("星期五")
weeksNames.addObject("星期六")
```

```
weeksNames.addObject("星期日")
③

NSLog("星期名字")
NSLog("====   ====")

for vari = 0; i<weeksNames.count; i++ {
    var obj : AnyObject =
weeksNames.objectAtIndex(i)
    var day = obj as NSString
    NSLog("%i     %@", i, day)
}

for item : AnyObject in weeksNames {
    var day = item as NSString
    NSLog("%@", day)
}
```

第①行代码是通过构造器-initWithCapacity:实例化NSMutableArray，其中capacity是外部参数名，capacity为容器大小，也就是数组中初始的单元。第②行~第③行是添加元素到weeksNames数组，如果超过了容量会自动追加的。

19.3.3 NSArray与Array之间的关系

NSArray与Array之间的关系如同NSString与String之间的关系，Swift在底层能够将它们自动地桥接起来，一个NSArray对象桥接之后的结果是[AnyObject]数组（保存AnyObject元素的Array数组）。

下面我们看一个使用Array和NSArray的示例：

```
import Foundation
①

let foundationString : NSString = "alpha bravo
charlie delta echo"                          ②
let foundationArray : NSArray =
foundationString.componentsSeparatedByString("
")        ③
let swiftArray = foundationArray
④

for item in foundationArray  {
⑤
    println(item) //输出类型是NSString
⑥
}
```

```
for item in foundationArray as [String] {
⑦
    printin(item) //输出类型是String
⑧
}

for item in swiftArray  {
    printin(item) //输出类型是AnyObject
⑨
}

for item in swiftArray as [String] {
    printin(item) //输出类型是String
⑩
}
```

代码第①行是引入Foundation。第②行代码声明并初始化NSString字符串，第③行代码使用NSString的componentsSeparatedByString方法，该方法可以使用指定的字符分割字符串，返回NSArray数组。

第④行代码是将NSArray数组赋值给Array数组，这个过程也发生了类型转换，不仅是NSArray到Array的转换，而且它们的内部元素也从

NSString转换为AnyObject。

第⑤行代码是遍历foundationArray集合，第⑥行代码输出的是NSString数据。

第⑦行代码是将数组foundationArray集合转换为[String]数组，然后遍历集合，第⑧行代码输出的是String数据。

第⑨行代码输出的是AnyObject数据。第⑩代码输出的是String数据。

19.4 字典类

在Foundation框架中提供一种字典集合，它是由“键-值”对构成的集合。键集合不能重复，值集合没有特殊要求。键和值集合中的元素可以是任何对象，但是不能是nil。Foundation框架字典类也分为NSDictionary不可变字典和NSMutableDictionary可变字典。

19.4.1 NSDictionary类

NSDictionary有很多方法和属性，下面总结其常用的方法和属性。

- -initWithDictionary: 构造器，通过Swift的Dictionary创建NSDictionary对象。
- -initWithObjects:forKeys: 构造器，通过键集合和值集合创建NSDictionary对象。
- -count 字典集合的长度。
- -objectForKey: 通过键获得值对象。
- -allKeys: 返回所有键集合。

下面看NSDictionary数组的示例代码：

```
import Foundation

let keyString : NSString = "one two three four
five"                                    ①
var keys : NSArray =
keyString.componentsSeparatedByString(" ")
②

let valuestring : NSString = "alpha bravo
charlie delta echo"                          ③
var values : NSArray =
valuestring.componentsSeparatedByString(" ")
④

var dict : NSDictionary = NSDictionary(objects:
keys, forKeys:values)                    ⑤

NSLog("%@", dict.description)
⑥

var value:NSString = dict.objectForKey("three")
as NSString                              ⑦
NSLog("three = %@", value)

var kys = dict.allKeys
⑧
for item : AnyObject in kys {
```

```
⑨
    var key = item as NSString
    NSLog("%@ - %@", key,
dict.objectForKey(key) as NSString)
}
```

上述代码第①行和第③行代码是NSString字符串，字符串由单词和空格组成。第②行和第④行代码使用空格分割字符串，返回类型是NSArray数组。

第⑤行代码实例化NSDictionary对象，objects参数是值数组values，forKeys参数是。第⑥行代码description属性是获得字典的内容。第⑦行代码是通过objectForKey方法读取键对应的值，并且转换为NSString类型。

第⑧行代码dict.allKeys是获得所有的键集合kys，第⑨行代码是遍历键集合kys。

输出结果如下：

```
2014-07-06 20:19:07.274
PlaygroundStub_OSX[4110:303] {
five = echo;
four = delta;
```

```
one = alpha;
three = charlie;
two = bravo;
}
2014-07-06 20:19:07.281
PlaygroundStub_OSX[4110:303] three = charlie
2014-07-06 20:19:07.296
PlaygroundStub_OSX[4110:303] one - alpha
2014-07-06 20:19:07.300
PlaygroundStub_OSX[4110:303] five - echo
2014-07-06 20:19:07.305
PlaygroundStub_OSX[4110:303] three - charlie
2014-07-06 20:19:07.308
PlaygroundStub_OSX[4110:303] two - bravo
2014-07-06 20:19:07.313
PlaygroundStub_OSX[4110:303] four - delta
```

19.4.2 NSMutableDictionary类

NSMutableDictionary是NSDictionary的子类，它有很多方法和属性，下面总结其常用的方法和属性。

- -setObject:forKey: 通过键和值。
- -removeObjectForKey: 按照键移

除值。

下面看NSDictionary数组的示例代码：

```
import Foundation

var mutable : NSMutableDictionary =
NSMutableDictionary()                          ①
// add objects
mutable.setObject("Tom", forKey:
"tom@jones.com")                                ②
mutable.setObject("Bob", forKey:
"bob@dole.com")

NSLog("%@", mutable.description)

var keys = mutable.allKeys
for item : AnyObject in keys {
    var key = item as NSString
    NSLog("%@ - %@", key,
mutable.objectForKey(key) as NSString)
}
```

上述代码第①行是实例化NSMutableDictionary，第②行代码是通过setObject方法添加键和值。

输出结果如下：

```
2014-07-06 20:42:11.596
PlaygroundStub_OSX[4332:303] {
    "bob@dole.com" = Bob;
    "tom@jones.com" = Tom;
}
2014-07-06 20:42:11.605
PlaygroundStub_OSX[4332:303] bob@dole.com - Bob
2014-07-06 20:42:11.608
PlaygroundStub_OSX[4332:303] tom@jones.com -
Tom
```

19.4.3 NSDictionary与Dictionary之间的关系

NSDictionary与Dictionary之间的关系如同NSArray与Array之间的关系，Swift在底层能够将它们自动地桥接起来，一个NSDictionary对象桥接之后的结果是[NSObject : AnyObject]字典（值为NSObject 类型，键为AnyObject 类型的Dictionary字典）。

下面我们看一个使用Dictionary和NSDictionary的示例：

```
import Foundation
①

let keyString : NSString = "one two three four
five"
let keys : NSArray =
keyString.componentsSeparatedByString(" ")

let valueString : NSString = "alpha bravo
charlie delta echo"
let values : NSArray =
valueString.componentsSeparatedByString(" ")

let foundationDict : NSDictionary =
NSDictionary(objects:values, forKeys:keys)
②

let swiftDict : Dictionary = foundationDict
③
println(swiftDict.description)

let value: AnyObject? = swiftDict["three"]
④
println("three = \(value)")

for (key, value) in swiftDict {
⑤
    println ("\(key) - \(value)")
}
```

代码第①行是引入Foundation。第②行代码声明并初始化NSDictionary字典，第③行代码是将NSDictionary字典赋值给Dictionary字典，这个过程也发生了类型转换，不仅是NSDictionary到Dictionary的转换，而且它们的内部元素也发生了转换。

第④行代码是从Dictionary字典取three键对应的值，它的类型是可选的AnyObject类型，这是因为有可能取不到这个值。第⑤行代码是遍历Dictionary字典键和值集合。

19.5　本章小结

通过对本章内容的学习，我们了解了什么是Foundation框架，以及如何通过Swift语言使用Foundation框架。此外，还了解了Foundation框架中的数字、字符串、数组、字典等。

19.6 同步练习

1. 下列数组定义正确的是（　　）。

A. `let a : NSArray = [1,2]`

B. `let a : [Int] = [1,2]`

C. `var b: NSArray = NSArray(array: ["张三","李四"])`

D. `var b: NSArray = NSArray(objects: ["张三","李四"])`

2. 给定语句`let s : NSString = "Testing 1 2 3"`，那么表达式`s.substringToIndex(5)`的值是（　　）。

A. `Test`　　B. `Testi`　　C. `Testin`　　D. `undefined`

3. 给定语句`let f : NSNumber = NSNumber.numberWithFloat(100.0)`下面取值语句正确的是（　　）。

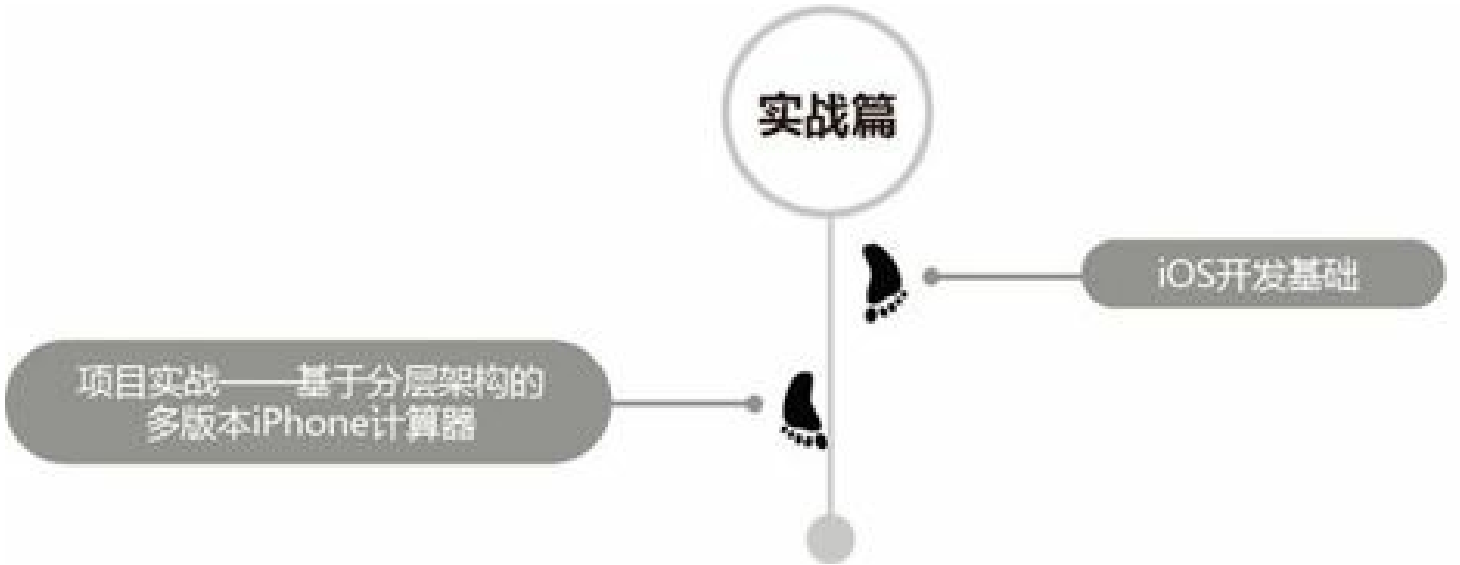
A. `NSNumber.floatValue()`　　B. `f.floatValue`

C. `f.floatValue()`　　D. `f.numberWithFloat`

4. 编程题：编写一个程序说明Swift使用Foundation框架中的数组。

5. 编程题：编写一个程序说明Swift使用Foundation框架中的字典。

第四部分　实战篇



第 20 章　iOS开发基础

我们在开始编写iOS项目之前有必要了解一些iOS基础知识，但本书不是系统地介绍iOS开发的知识，而是侧重介绍Swift语言，能够通过编写一个iPhone项目了解iOS开发基础。

20.1 iOS介绍

iOS的系统架构分为4层——Cocoa Touch层、Media层、Core Services层和Core OS层，相关内容可参见20.3节。

下面我们简要介绍一下iOS的一些功能，具体如下所述。

- **多点触摸和手势**。触摸功能在iOS设备之前就已经采用，但基本都是单点触摸，即只能用一个手指，而iOS设备能够感应多个手指的触摸。为了配合这种多点触摸，iOS上的触摸分为多种手势：触击、双击、滑动、长期间触击、轻拂、刷屏和手指合拢张开等。
- **统一的屏幕尺寸**。到目前为止，iOS屏幕尺寸有6套：iPhone和iPod touch是3.5英寸，iPhone 5、iPhone 5s、iPhone 5c和第5代iPod touch是4英寸，iPhone 6是4.7英寸，iPhone 6 Plus是5.5英寸，iPad是9.7英寸，iPad mini是7.9英寸。统一的屏幕尺寸给应用软件开发带来很多好处，开发人员可以不用关心屏幕尺寸适配的问题，从而把精力集中在其他方面。

- **高分辨率**。iPhone 4S的屏幕分辨率是960×640（像素），iPhone 5、iPhone 5s、iPhone 5c和第5代iPod touch的屏幕分辨率是1136×640（像素），iPhone 6屏幕分辨率1334×750（像素），iPhone 6 Plus屏幕分辨率是1920×1080（像素），第1代、第2代iPad的屏幕分辨率是1024×768（像素），第3代iPad的屏幕分辨率是2048×1536（像素），而iPad mini的屏幕分辨率是1024×768（像素）。
- **重力加速计**。iOS内置了重力加速计。有了重力加速计，用户能够玩很多有意思的游戏（如极品飞车，它可以把iPhone作为方向盘，通过重力加速计感应方向的变化）。此外，还有很多与重力加速计有关的应用软件，如水平尺应用等。
- **指南针**。iOS内置了指南针设备。很多应用基于指南针，例如导航软件和地图应用软件等。
- **蓝牙和Wi-Fi连接**。iOS内置了蓝牙和Wi-Fi通信模块。iOS设备之间可以采用Wi-Fi互相连接，也可以采用蓝牙进行连接，很多基于局域网的游戏就是通过这个功能实现

的。当然，也可以通过Wi-Fi上网，这可以节约用户的上网费用。此外，iOS还可以与计算机连接。

20.2 第一个iOS应用HelloWorld

在学习之初，我们有必要对使用Xcode创建iOS工程做一个概览，这里我们通过创建一个基于Swift语言的HelloWorld的iPhone工程来详述其中涉及的知识点。

实现HelloWorld应用后，会在界面上展示字符串Hello World（效果如图20-1所示），其中主要包含Label（标签）控件和ImageView（图片视图）。

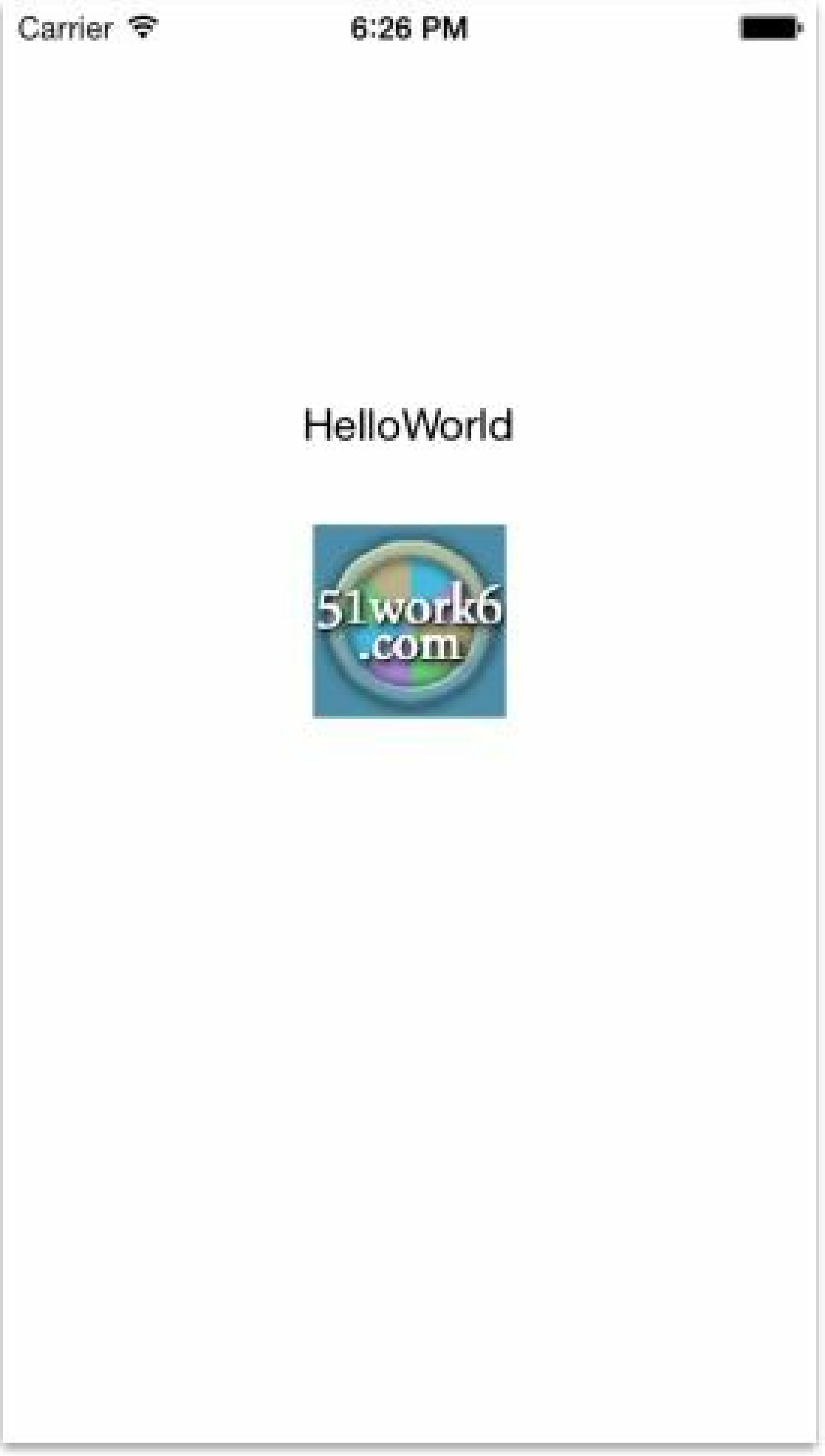

Carrier
6:26 PM
HelloWorld
51work6
.com

图 20-1　HelloWorld的iPhone界面

20.2.1　创建工程

我们在前面的章节介绍过创建iOS工程，但并没有详细解释，为了完整地介绍HelloWorld应用开发过程，我们还是从创建工程开始介绍。

启动Xcode 6，然后单击File→New→Project菜单，在打开的Choose a template for your new project界面中选择“Single View Application”工程模板（如图20-2所示）。

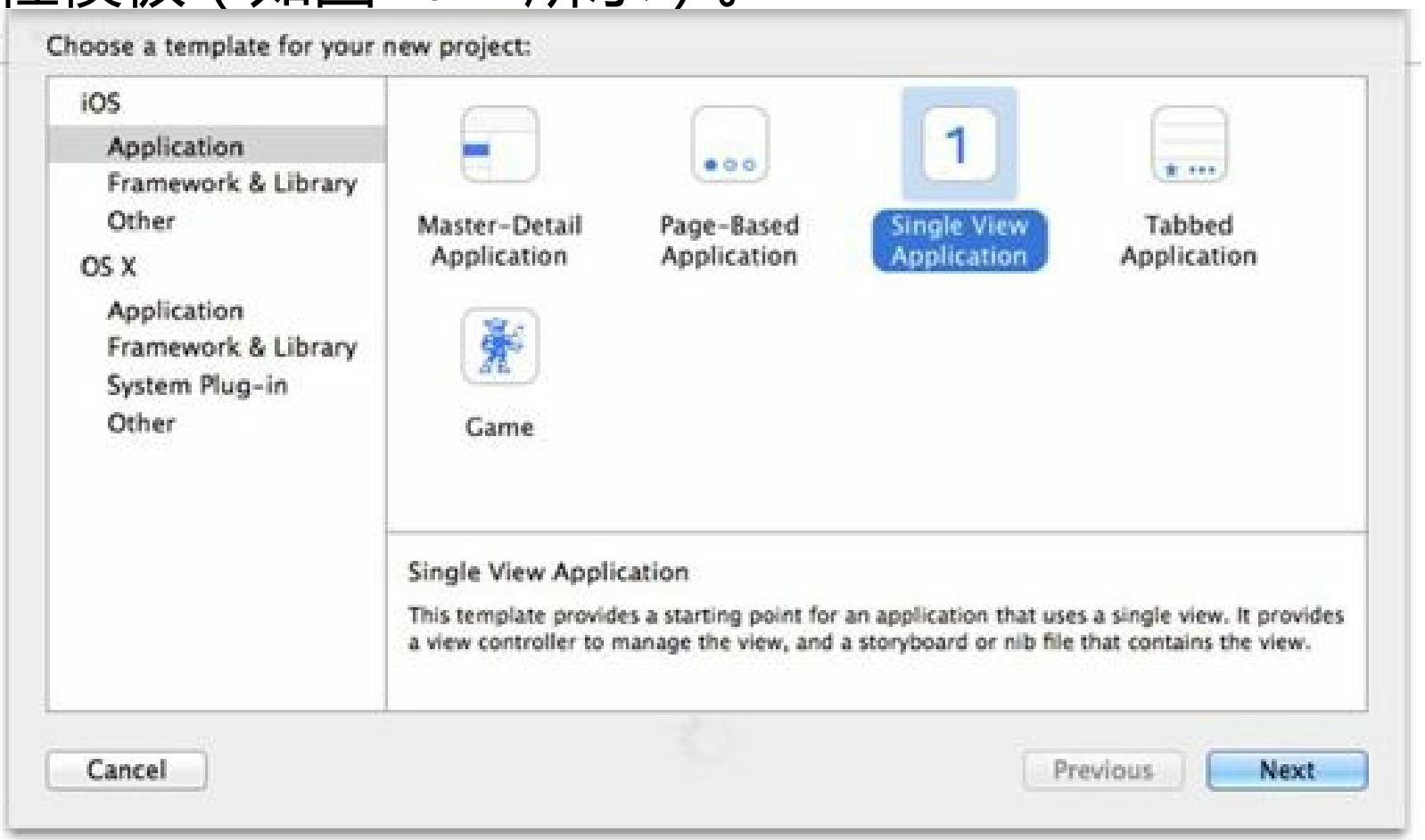


图 20-2　选择工程模板

接着单击“Next”按钮，随即出现如图20-3所

示的界面。

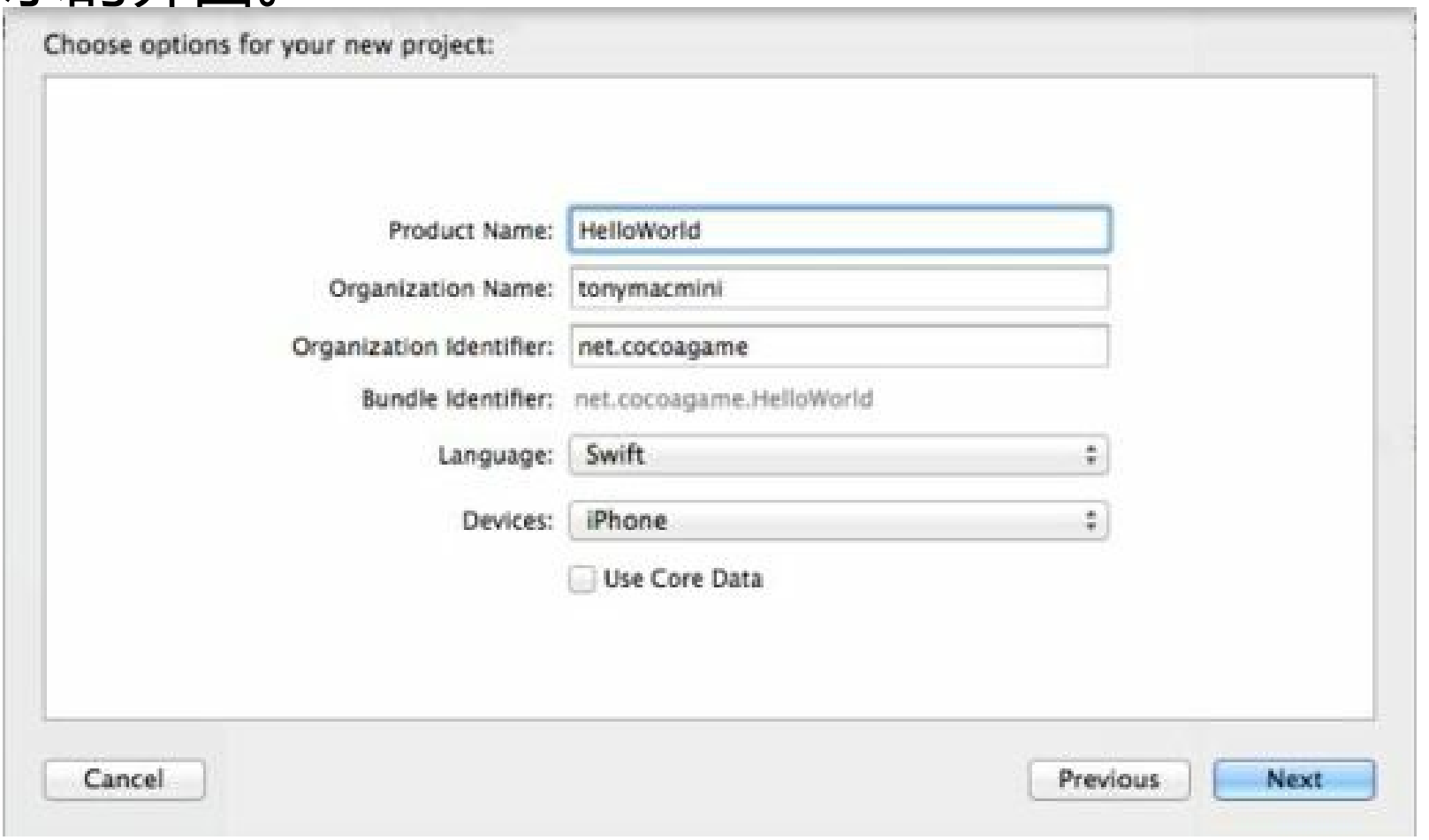


图 20-3　新工程中的选项

图20-3所示的内容是创建新工程所需的基本信息，其中选项说明如下。

- Product Name。工程名字。
- Organization Name。组织名字。
- Organization Identifier。组织标识（很重要）。一般情况下，这里输入的是组织名或公司的域名（如net.cocoagame），类似于Java中的包命名。
- Bundle Identifier。捆绑标识符（很重

要）。该标识符由Product Name+ Organization Identifier构成。因为在App Store发布应用的时候会用到它，所以它的命名不可重复。

- Language。开发语言选择。我们在这里可以选择开发应用所使用的语言，Xcode 6中可以选择Swift和Objective-C。
- Devices选项。这是选择应用运行的设备，可以构建基于iPhone或iPad的工程，也可以构建通用工程。通用工程是指在iPhone和iPad上都可以正常运行的工程。

我们可以根据自己的实际情况和需要输入相关内容。设置完相关的工程选项后，单击“Next”按钮，进入下一级界面。根据提示选择存放文件的位置，然后单击“Create”按钮，将出现如图20-4所示的界面。

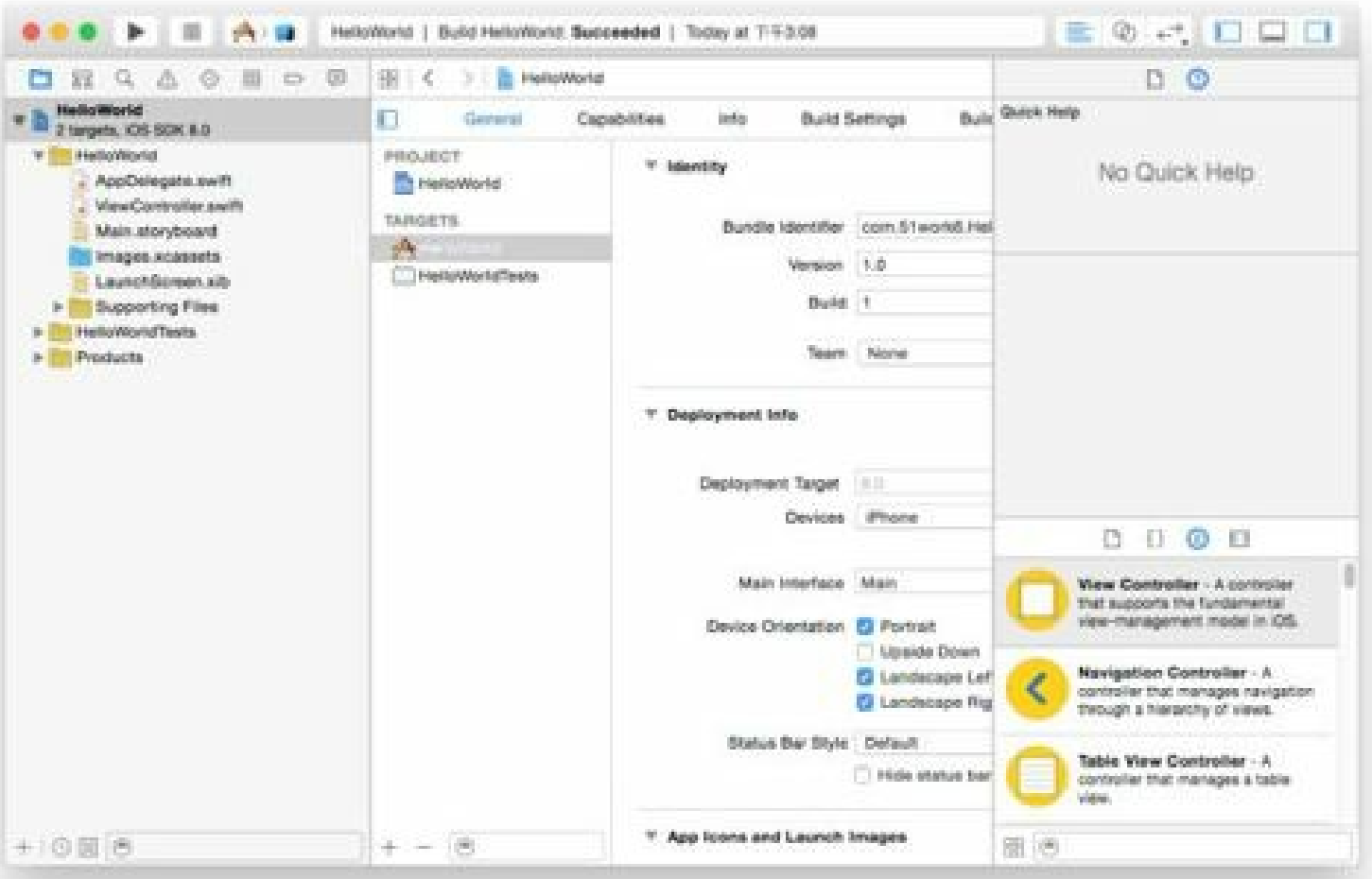


图 20-4　新创建的工程

在左边的导航栏中双击打开Main.storyboard文件，看到如图20-5所示的设计界面，这个设计界面是使用Interface Builder（IB）工具打开的。Xcode 4.1之后，Interface Builder被集成到了Xcode开发工具中。

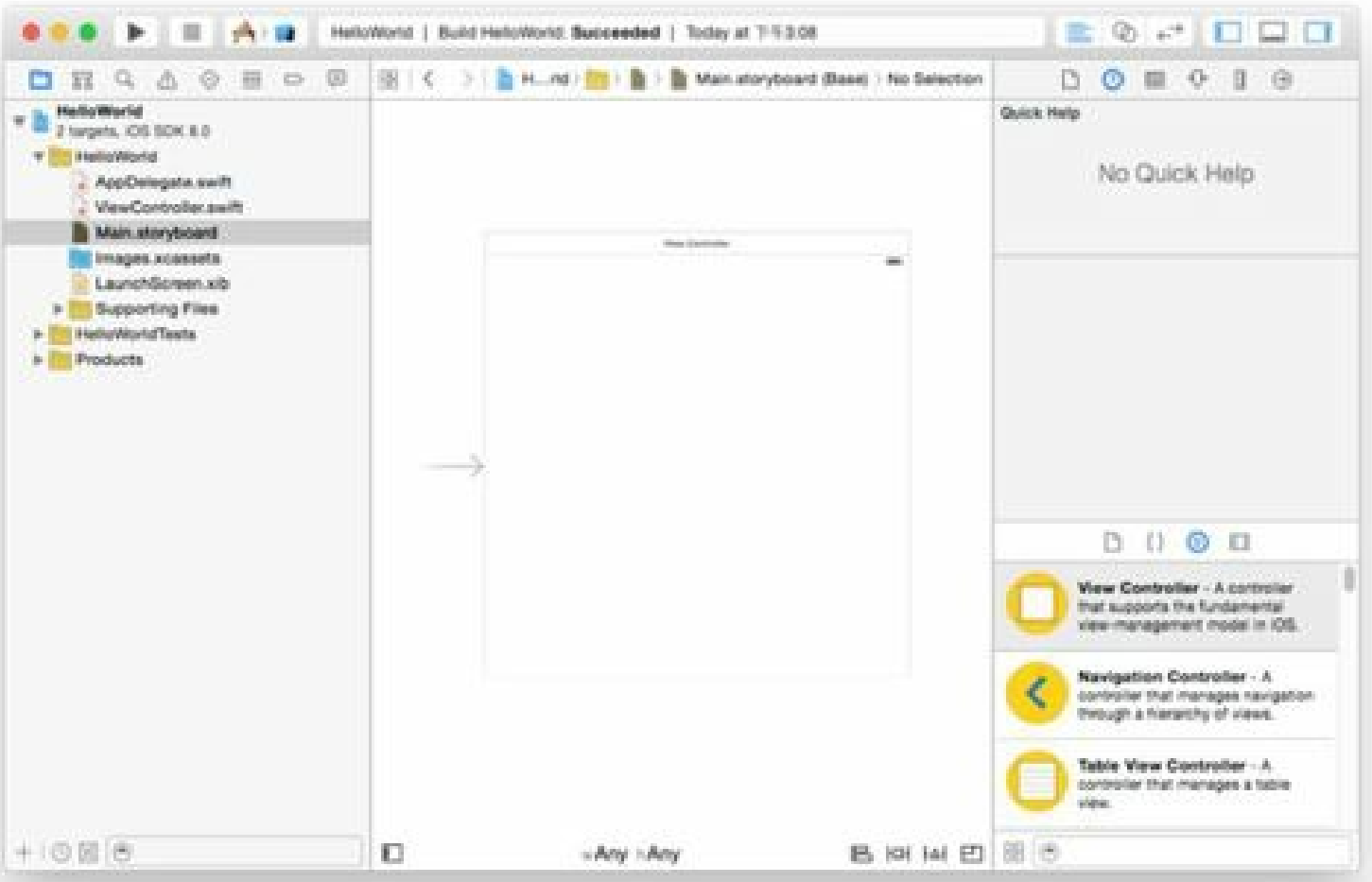


图 20-5　Interface Builder设计界面

默认情况下，设计界面尺寸是通用大小，我们需要根据应用的实际情况，改变设计界面的大小。由于我们的HelloWorld是为iPhone5以上4英寸屏幕设备开发的应用，所以需要改变设计界面的大小，单击图20-6中设计下面的“wAny hAny”（Size Classes[1]）按钮，在弹出的预览框中，拖动鼠标选择区域。改变之后的设计界面如图20-7所示。

[1]Size Classes（尺寸分类），注意它不是Size类的意思，而是iOS 8推出的解决各种不同屏幕尺寸适配问题的技术。

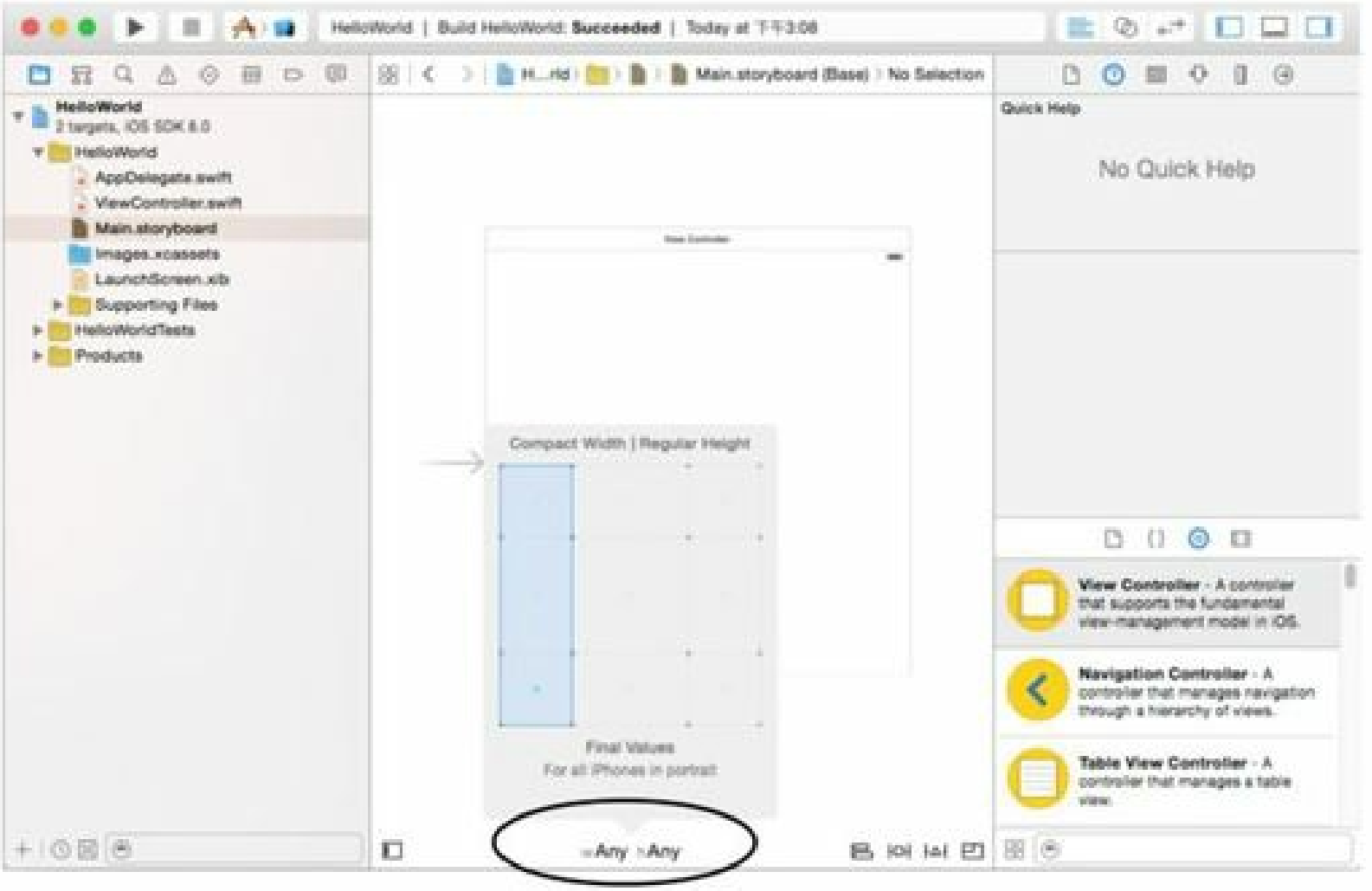


图 20-6　改变设计界面的大小

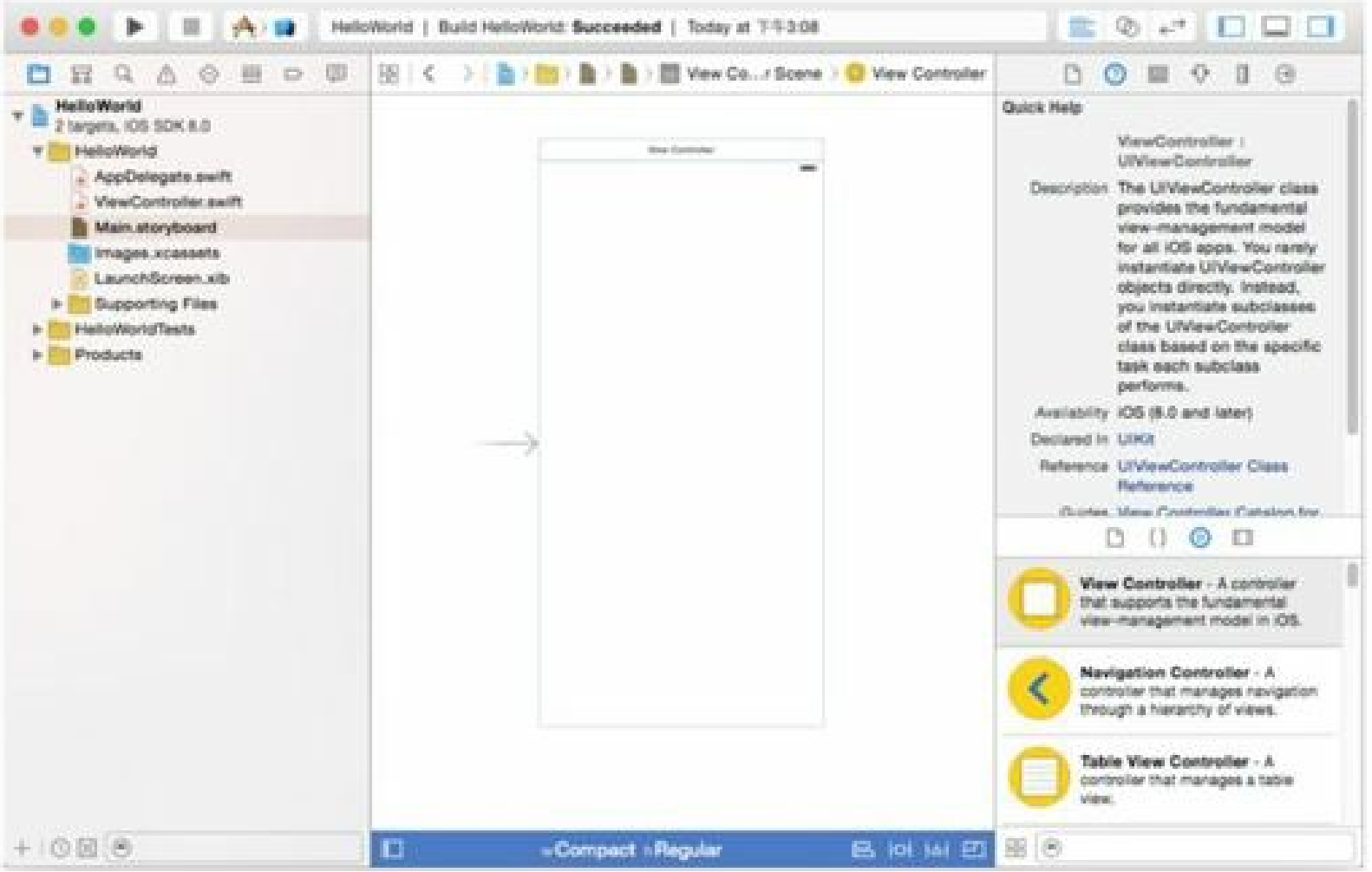


图 20-7　改变尺寸后的设计界面

在右下角的对象库中选择Label控件，将其拖曳到View设计界面上并调整其位置，如图20-8所示。双击Label控件，使其处于编辑状态（也可以通过控件的属性来设置），在其中输入“Hello World”。

提示　右下角的窗口默认情况下显示的并非是对象库，我们需要按图20-8所示点击显示对象库按钮 使之显示。还有，左边的场景窗口能够查看当前界面中有哪些视图控制器、

视图和控件，关闭和打开场景窗口，可以通过图20-8所示按钮实现。

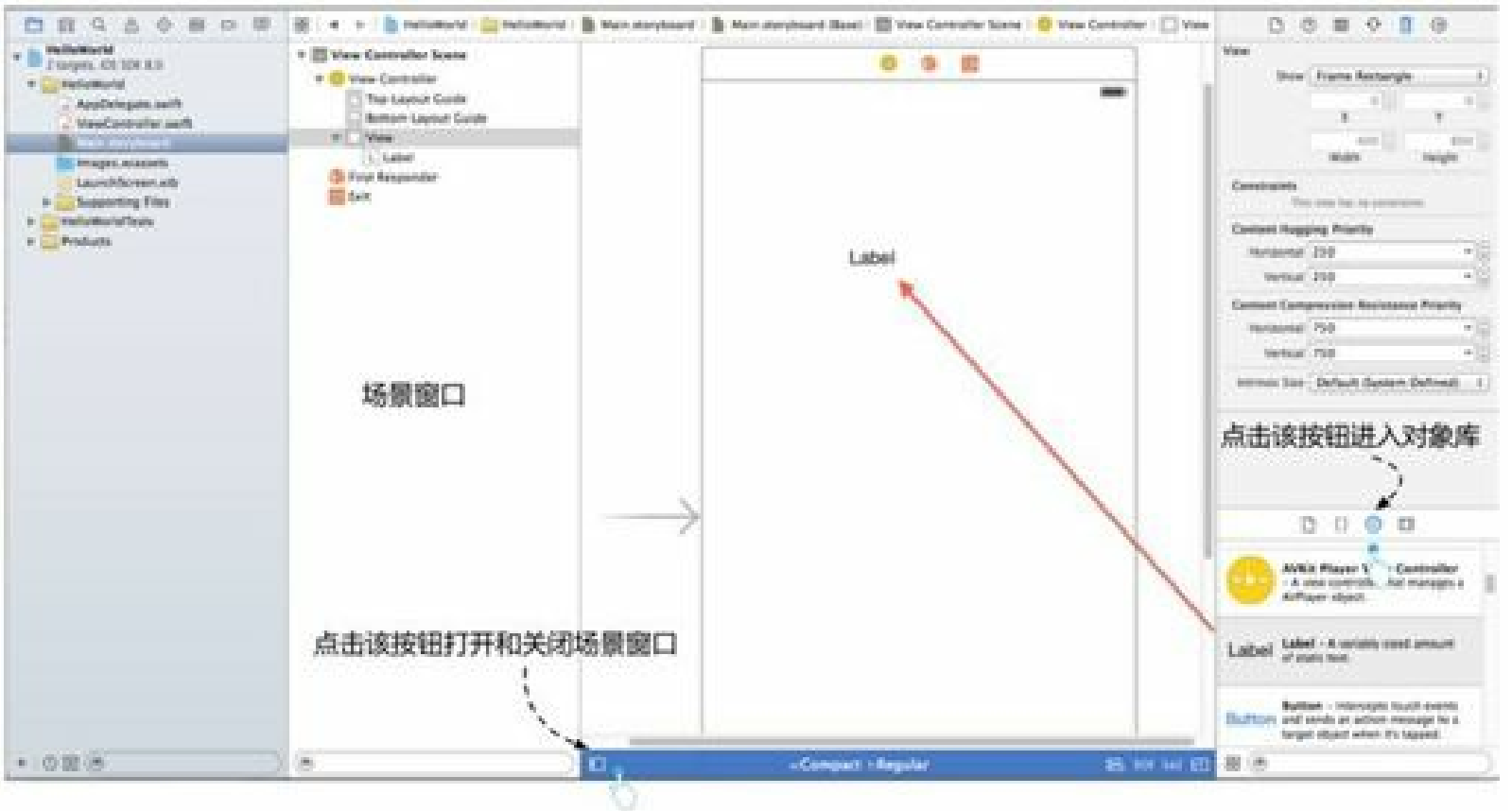


图 20-8　添加Label控件

接下来添加ImageView（图片视图）到View的设计界面，从右下角的对象库中选择ImageView控件，将其拖曳到View上，如图20-9所示。ImageView是图片的容器，它的大小应该跟原始图片是一样的，否则都会比率失调或者失真，因此需要设置它的尺寸。在View设计视图中选择ImageVew，然后再选中右边的尺寸检查器，按照如图20-10所示的尺寸设置，其中X和Y代表坐

标，Width和Height代表宽和高，将Width和Height都设置为76点[2]，这是因为要放置的图片大小是152×152（像素），对于iOS Retina显示屏而言，1点=2倍像素。修改完Width和Height属性后，重新摆放ImageView控件使其居中。

[2] iPhone 6 Plus显示屏中，1点 = 3倍像素。iPhone 6、iPhone 5/5s/5c（iPod touch 5）、iPhone 4s、iPad Air、iPad 3、iPad mini 2显示屏中，1点 = 2倍像素。iPad 2、iPad mini、iPhone 4之前的设备显示屏中，1点 = 1倍像素。

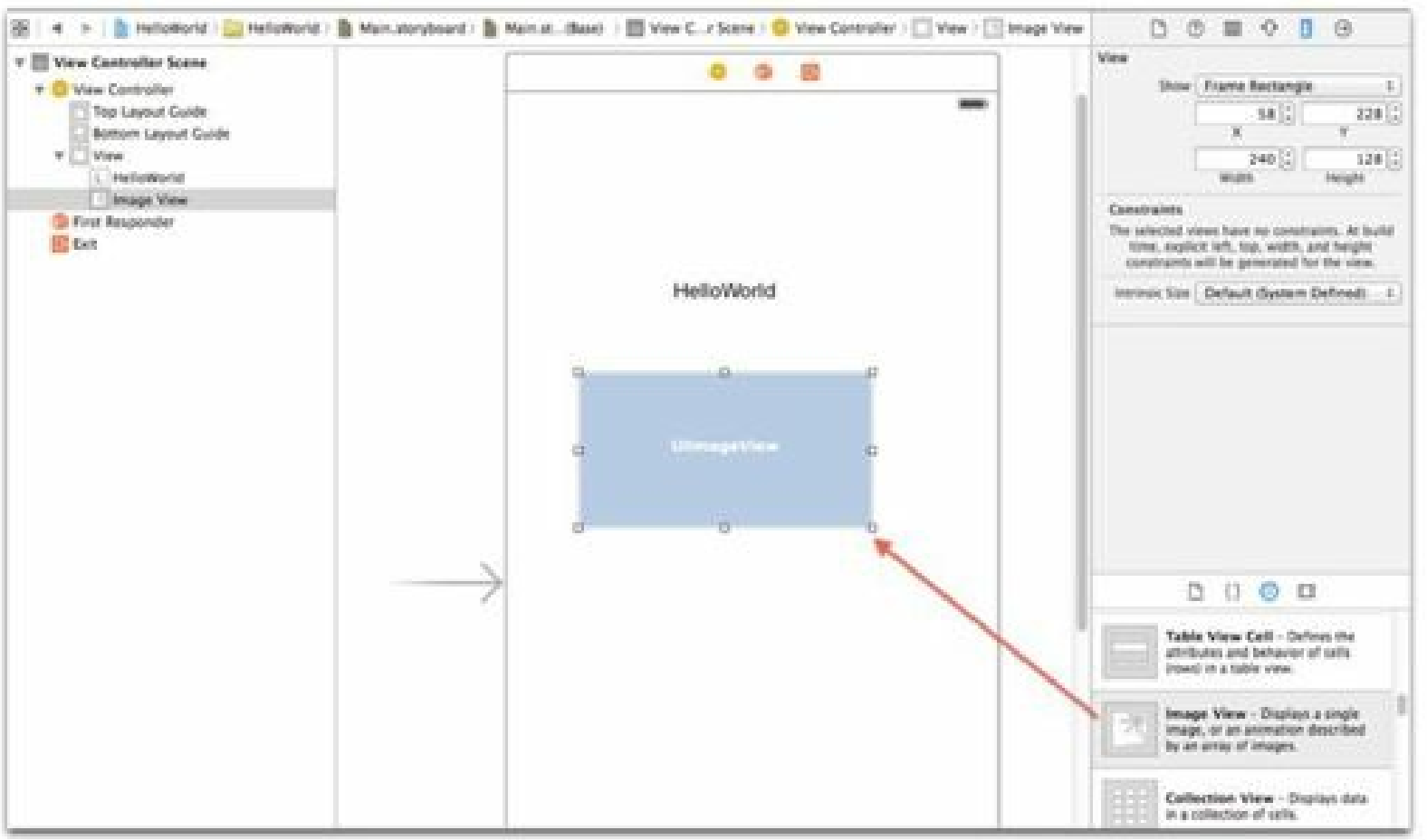


图 20-9　添加ImageView

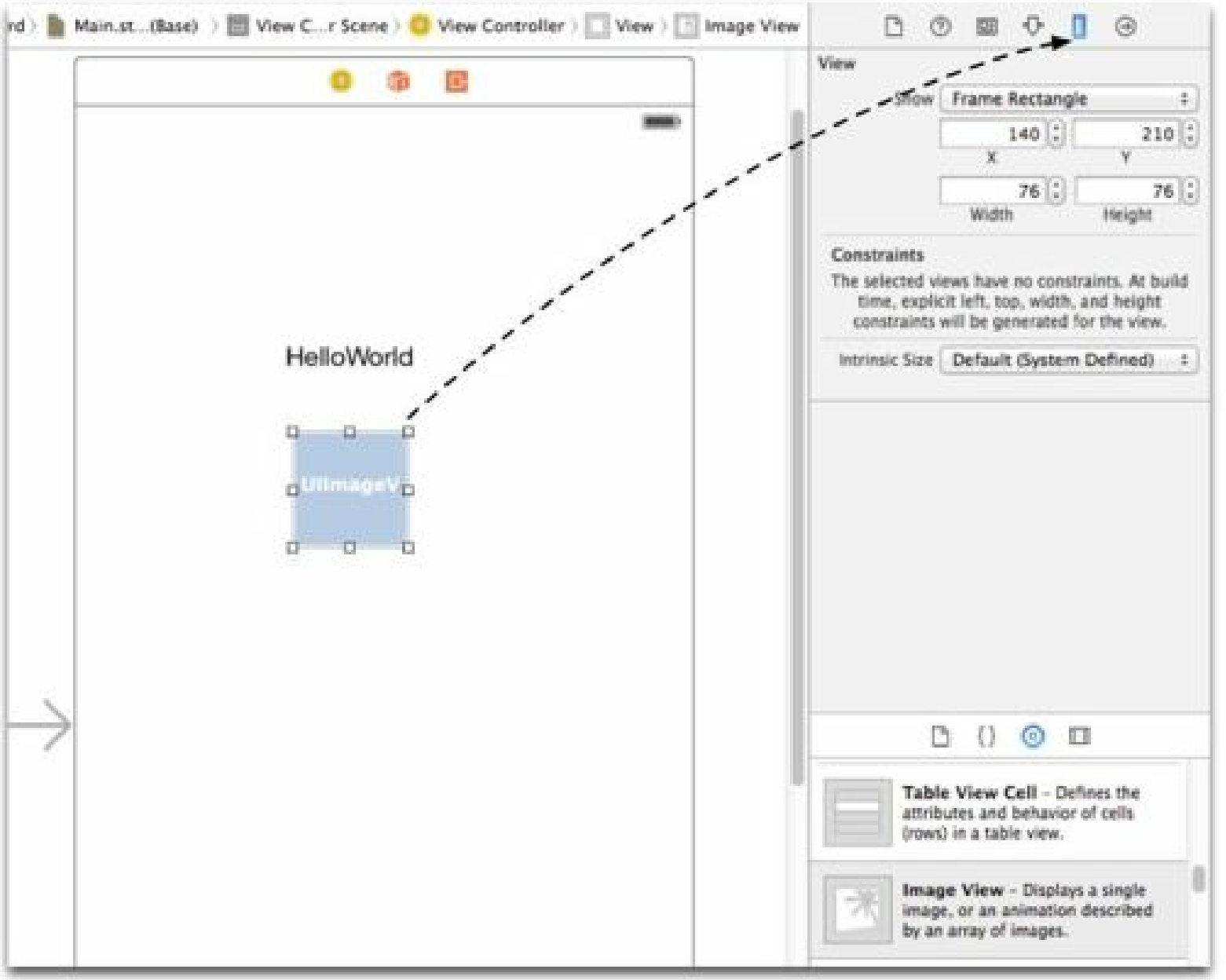


图 20-10　设置ImageView尺寸

设置好ImageView尺寸后，本例可以设置它所关联的图片，但是现在你的工程中是没有图片的，需要通过Xcode将图片添加到工程中。具体步骤是，如图20-11所示，右键选择Supporting Files组，选择菜单中的Add Files to “HelloWorld”...弹出选择文件对话框。如图20-12所示，在Destination中选中Copy items if needed，这样可

以将文件复制到我们的工程目录中，在Add to targets中选择HelloWorld。

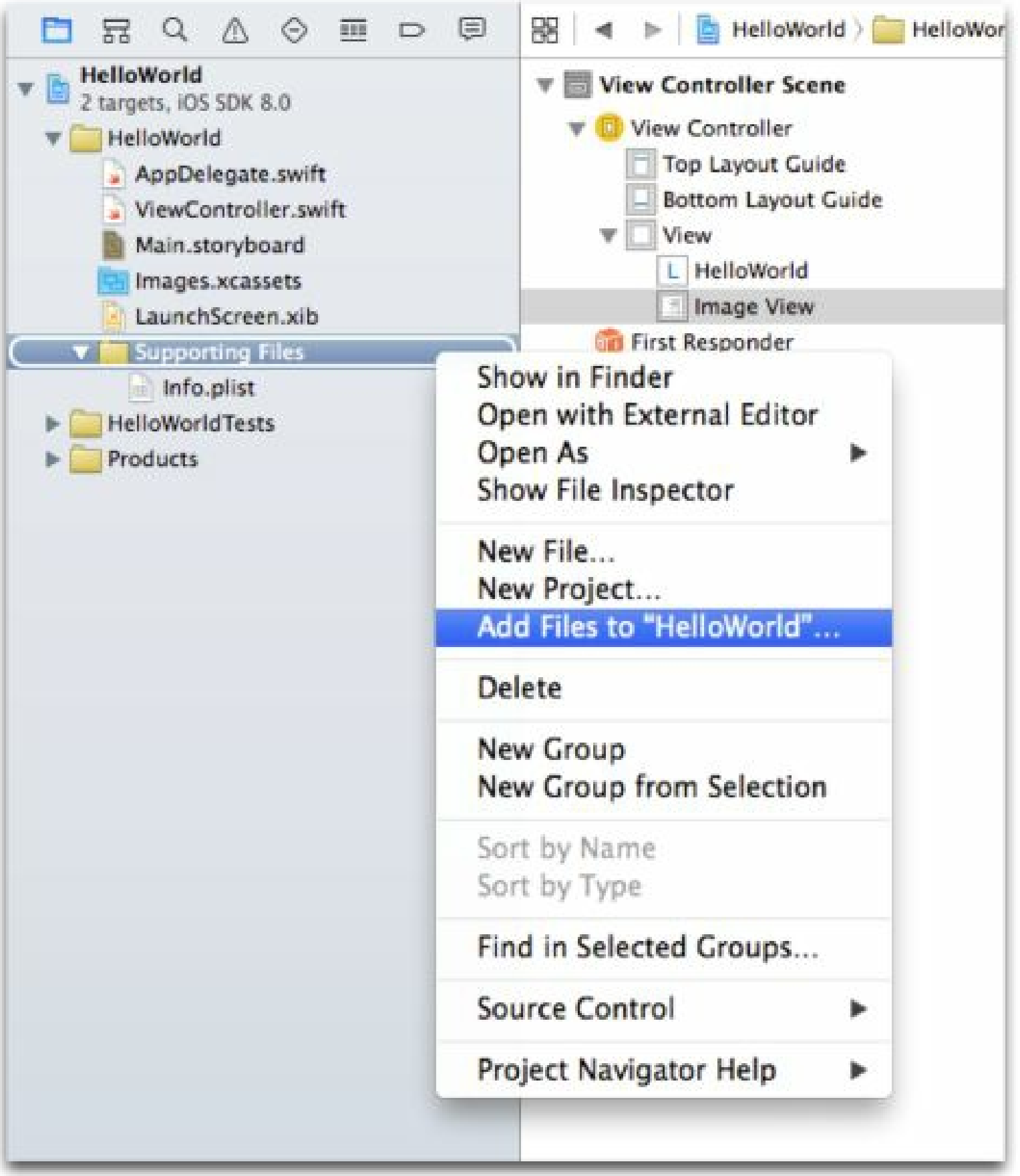

HelloWorld
2 targets, iOS SDK 8.0
HelloWorld
AppDelegate.swift
ViewController.swift
Main.storyboard
Images.xcassets
LaunchScreen.xib
Supporting Files
Info.plist
HelloWorldTests
Products
HelloWorld
View Controller Scene
View Controller
Top Layout Guide
Bottom Layout Guide
View
HelloWorld
Image View
First Responder
Show in Finder
Open with External Editor
Open As
Show File Inspector
New File...
New Project...
Add Files to "HelloWorld"...
Delete
New Group
New Group from Selection
Sort by Name
Sort by Type
Find in Selected Groups...
Source Control
Project Navigator Help

图 20-11　添加文件

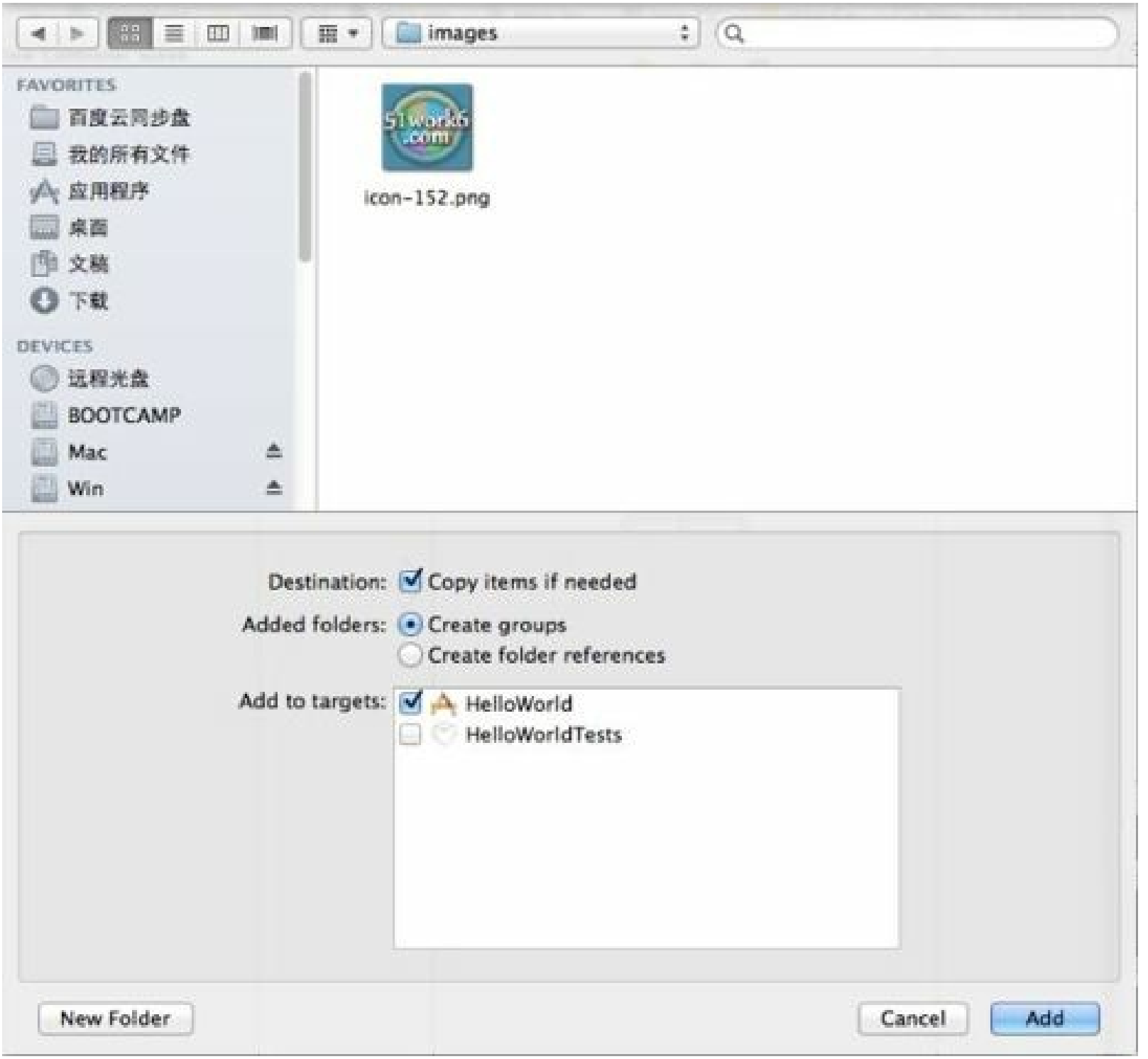


图 20-12　选择文件对话框

把图片文件添加到工程后，我们再选择ImageView，然后打开右边的属性检查器 ，在Image View→Image下的列表中选择图片文件（icon-152.png），如图20-13所示。

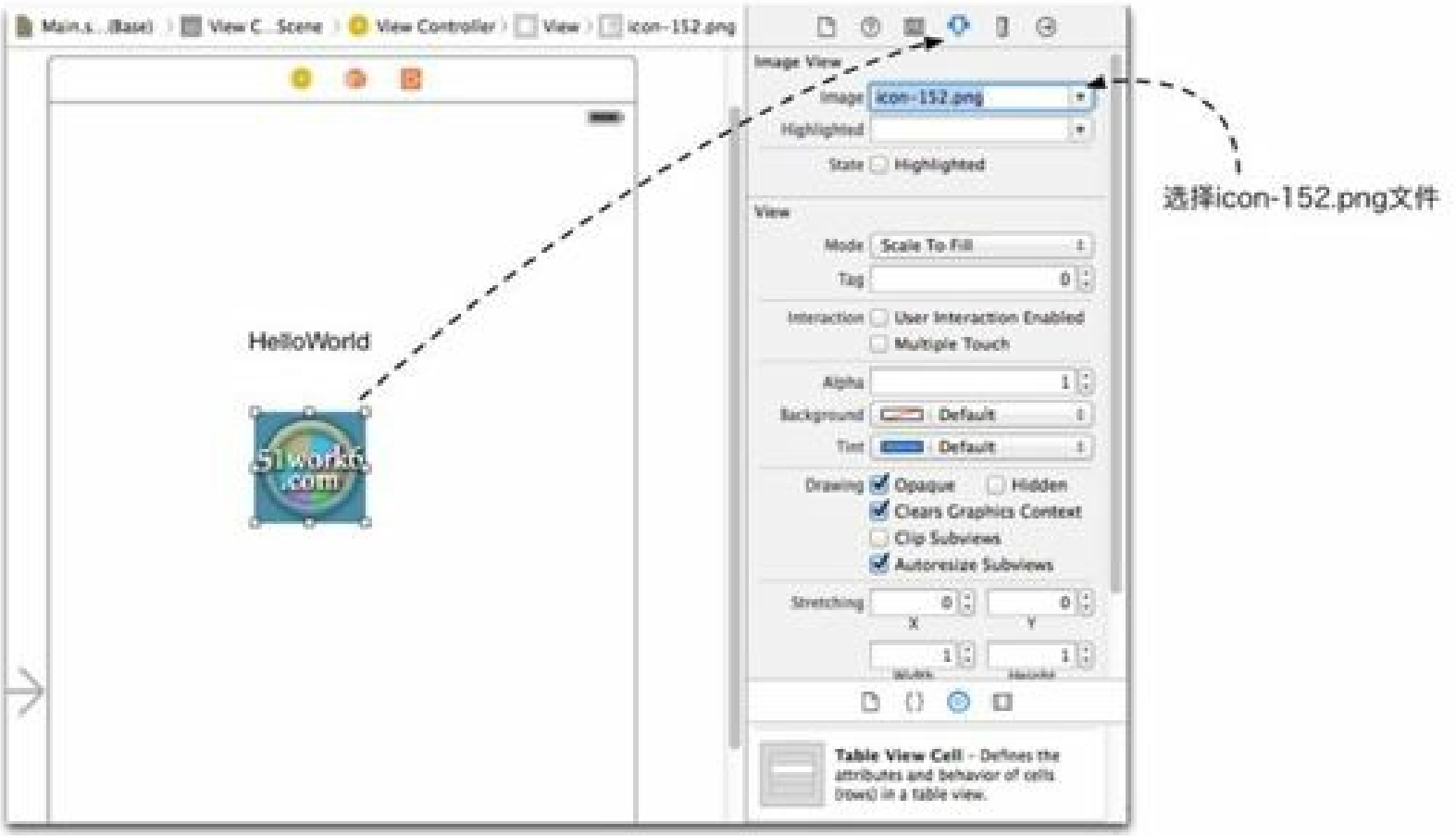


图 20-13　设置ImageView属性

至此，整个工程创建完毕。如图20-14所示，选择运行的模拟器或设备，然后单击左上角的运行按钮  ，即可看到图20-15所示的运行结果。


选择运行模拟器
HelloWorld
iPhone 5s
2 targets, iOS SDK 8.0
HelloWorld
AppDelegate.swift
ViewController.swift
Main.storyboard
Images.xcassets
运行按钮
iOS Device
iOS Simulator
iPad 2
iPad Air
iPad Retina
iPhone 4s
iPhone 5
iPhone 5s
iPhone 6 Plus
iPhone 6
Resizable iPad
Resizable iPhone

图 20-14　运行应用


iOS Simulator – iPhone 5s – iPhone 5s / iOS 8...
Carrier
10:56 AM
HelloWorld
51work6
.com

图 20-15　运行结果

我们在没有输入任何代码的情况下，就已经利用Xcode工具的Single View Application模板创建了一个工程并成功运行，Xcode之强大可见一斑。

20.2.2　Xcode中的iOS工程模板

从图20-16中可以看出，iOS工程模板分为3类——Appliction、Framework & Library和Other，下面将分别详细介绍这3类模板。

1. **Application类型**

大部分开发工作都是从使用Application类型模板创建iOS程序开始的。该类型共包含5个模板，具体如下所示。

- Master-Detail Application。可以构建树形结构导航模式应用，生成的代码中包含了导航控制器和表视图控制器等。
- Page-Based Application。可以构建类似于电子书效果的应用，这是一种平铺导航。
- Single View Application。可以构建简单的单个视图应用。
- Tabbed Application。可以构建标签导

航模式的应用，生成的代码中包含了标签控制器和标签栏等。

- Game。可以帮助开发iOS游戏应用，其中包括几个苹果自己的游戏引擎，以及直接使用Open GL开发iOS 3D游戏。

2. **Framework & Library类型**

Framework & Library类型的模板如图20-16所示，它可以构建基于Cocoa Touch框架和Cocoa Touch的静态库。

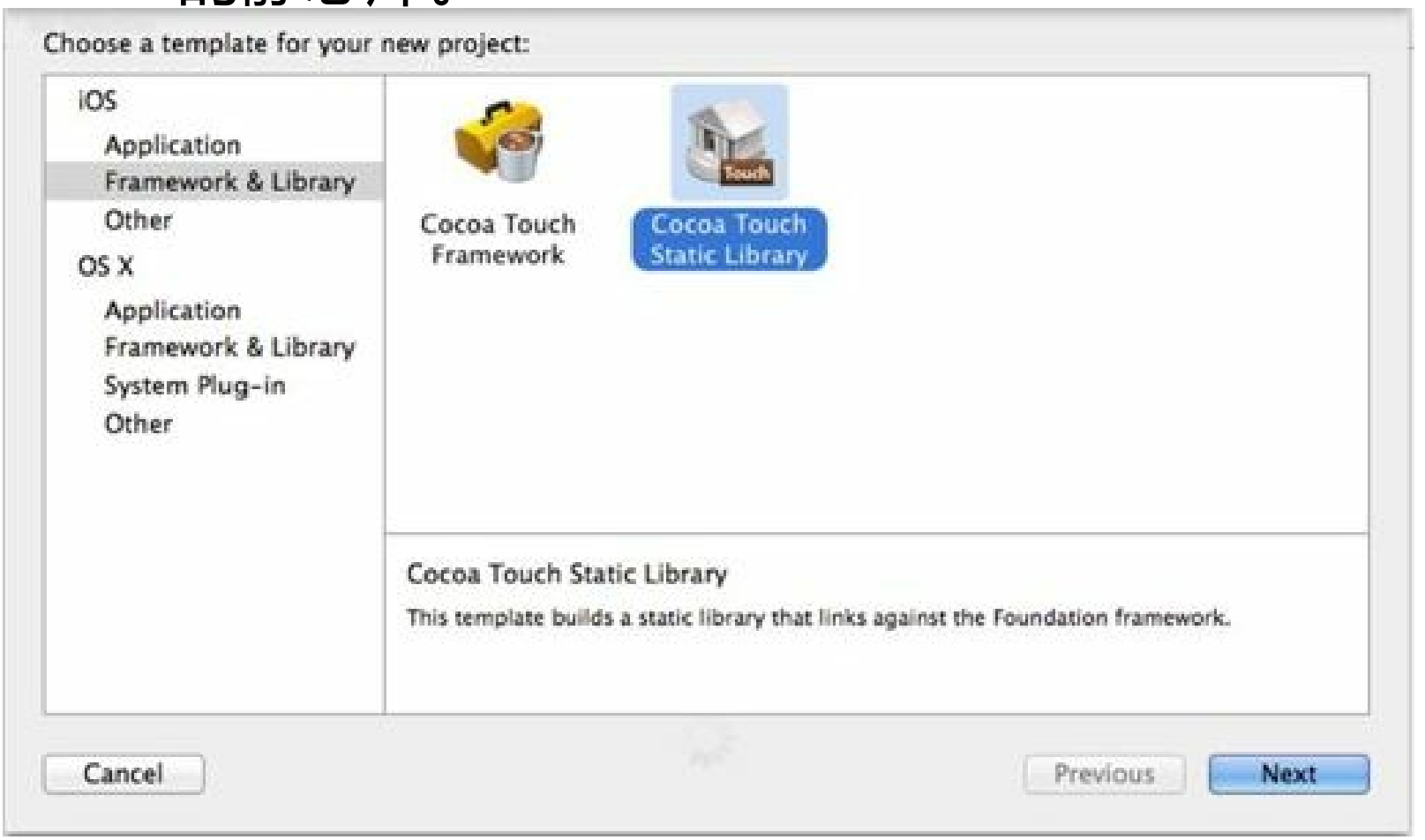


图 20-16　Framework & Library类型模板

3. **Other类型**

利用该类型，我们可以构建应用内购买内容包（In-App Purchase）和空工程，如图20-17所示。使用应用内购买内容包，可以帮助我们构建具有内置收费功能的应用。

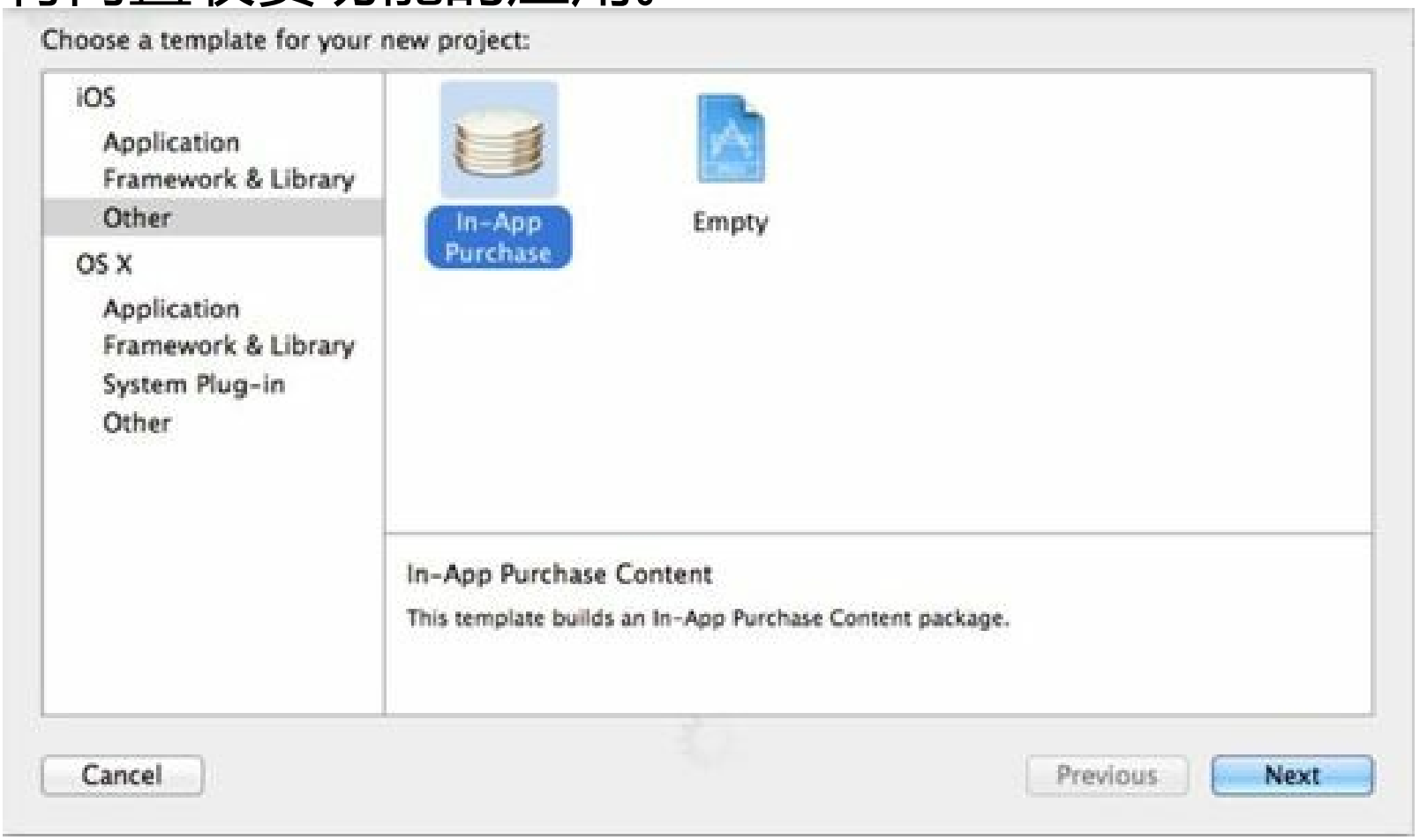


图 20-17　Other类型模板

我们可以根据需要选用不同的工程模板，这可以大大减少我们的工作量。

20.2.3　程序剖析

在创建HelloWorld的过程中，生成了很多文件（展开Xcode左边的项目导航视图可以看到，如图20-18所示），它们各自的作用是什么？彼此间又

是怎样的一种关系呢？

如图20-18所示，导航视图下有HelloWorld、HelloWorldTests和Products这3个组。其中，HelloWorld组中放置HelloWorld工程的主要代码，而HelloWorldTests组中放置的是HelloWorld程序的单元测试代码。Frameworks放置HelloWorld代码所依赖的框架或库，Products组放置了编译后的工程。下面我们重点介绍HelloWorld组中的内容。

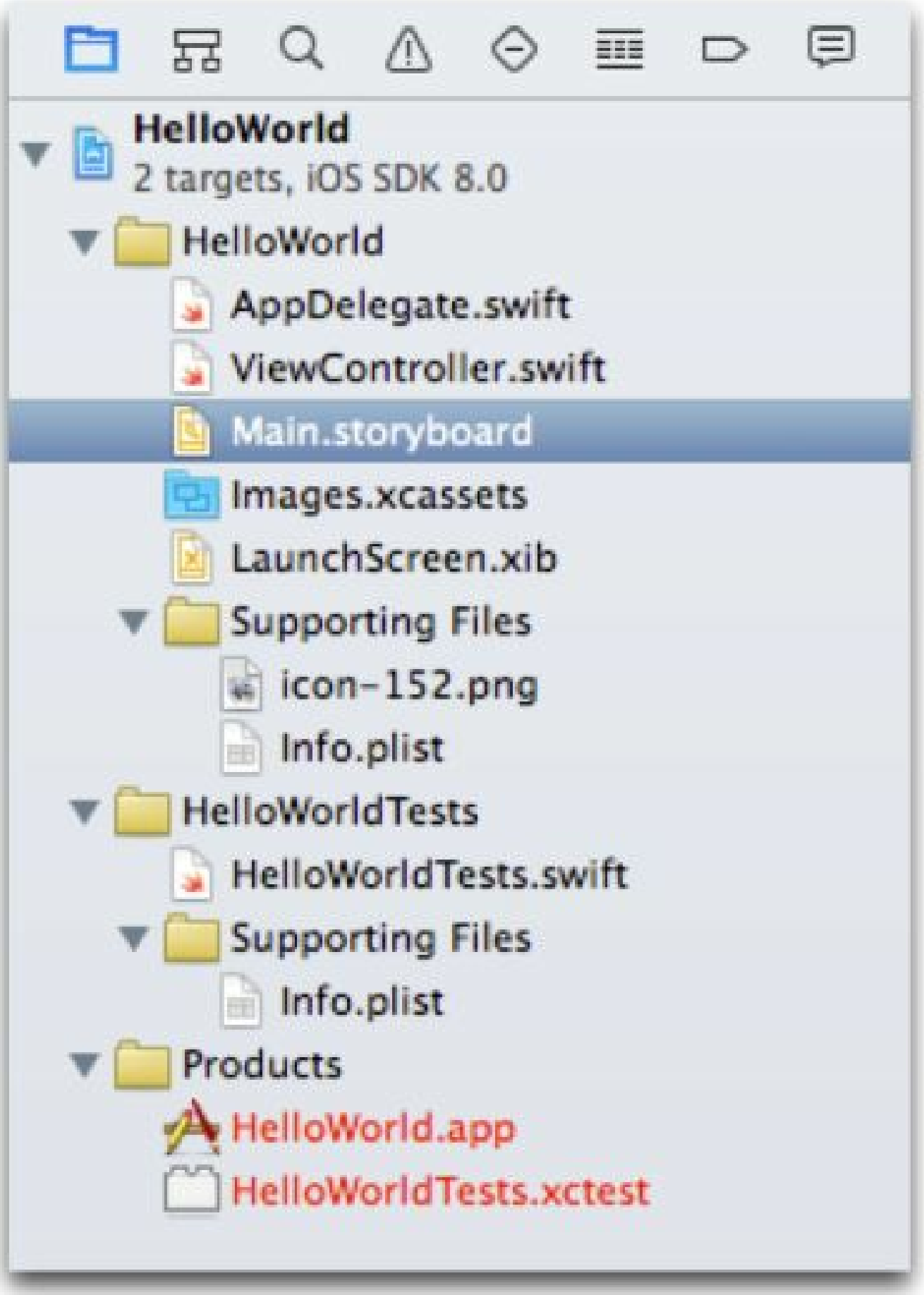

HelloWorld
2 targets, iOS SDK 8.0
HelloWorld
AppDelegate.swift
ViewController.swift
Main.storyboard
Images.xcassets
LaunchScreen.xib
Supporting Files
icon-152.png
Info.plist
HelloWorldTests
HelloWorldTests.swift
Supporting Files
Info.plist
Products
HelloWorld.app
HelloWorldTests.xctest

图 20-18　HelloWorld工程导航面板

在HelloWorld组中共有两个Swift文件：AppDelegate.swift和ViewController.swift，以及一个组Supporting Files。我们主要的编码工作就是在AppDelegate.swift和ViewController.swift这两个Swift文件中进行的，在这两个文件中分别定义了两个类AppDelegate和ViewController，它们的类图如图20-19所示。

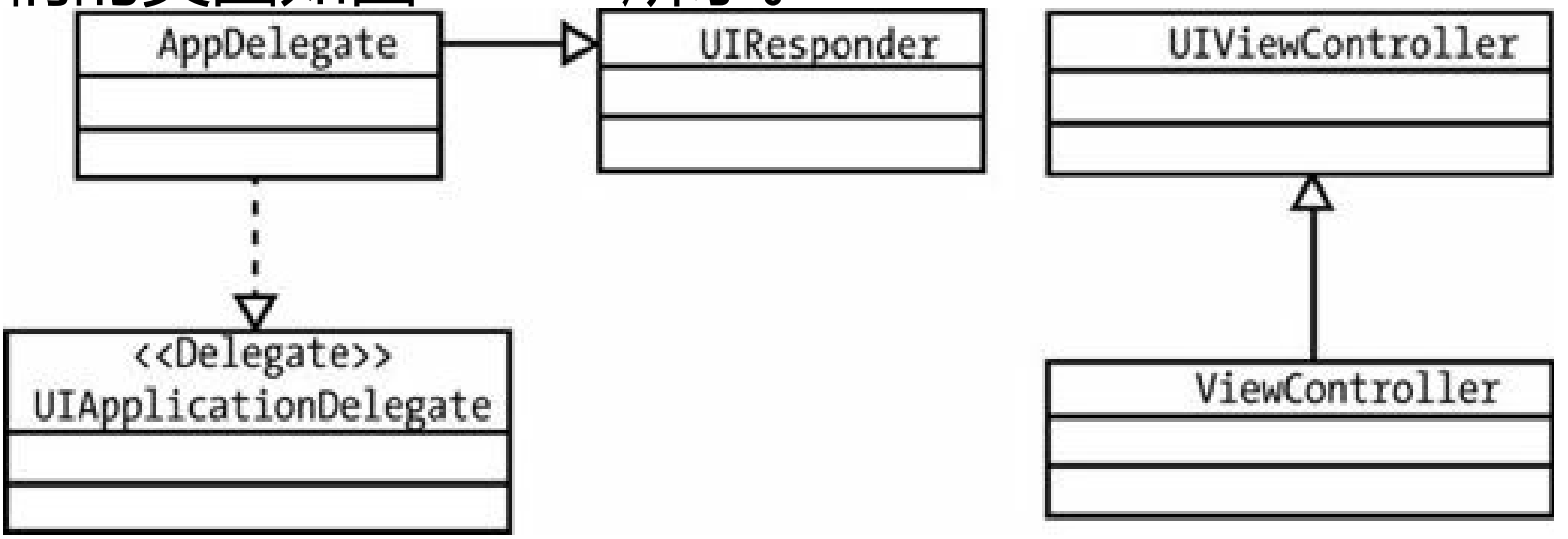


图 20-19　HelloWorld工程类图

AppDelegate是应用程序委托对象，它继承了UIResponder类，并遵守了UIApplicationDelegate委托协议。UIResponder类可以使子类AppDelegate具有处理相应事件的能力，而UIApplicationDelegate委托协议使AppDelegate能够成为应用程序委托对象，这

种对象能够响应应用程序的生命周期。相应地，AppDelegate的子类也可以实现这两个功能。

ViewController类继承自UIViewController类，它是视图控制器类，在工程中扮演着根视图和用户事件控制类的角色。

在HelloWorld启动过程中，操作系统首先实例化AppDelegate类，这是因为AppDelegate类被标注为@UIApplicationMain的，说明AppDelegate是iOS应用程序的入口，这个类还必须实现UIApplicationDelegate委托协议。AppDelegate类相关代码如下所示：

```
importUIKit

@UIApplicationMain
classAppDelegate: UIResponder,
UIApplicationDelegate {
        var window: UIWindow?
        ......
}
```

AppDelegate类是应用程序委托对象，这个类

中继承的一系列方法在应用生命周期的不同阶段会被回调。启动HelloWorld时，首先会调用application:didFinishLaunchingWithC方法，该方法的代码如下：

```
    func application(application: UIApplication,
didFinishLaunchingWithOptions
                                launchOptions:
[NSObject: AnyObject]?) -> Bool {
        // Override point for customization
after application launch.
        return true
    }
```

在HelloWorld组中还有Images.xcassets文件夹（灰色的是文件夹，黄色的是组），它可以放置工程中的图片。

提示 “文件夹”与“组”的区别是，在访问资源文件时，文件夹中的资源在访问路径里需要这个路径。如果一个icon.png文件放在image文件夹下，访问它的路径是“image/icon.png”，如果image是组，则访问它的路径是“icon.png”。

默认情况下Supporting Files组只有一个文件Info.plist，Info.plist是当前工程属性描述文件，它的默认名是Info.plist，如果想使用其他的名字，需要在编译参数（build settings）中重新设置。

20.3 iOS API简介

苹果的iOS API在不同版本间有很多变化，但是iOS API的整体架构没有什么变化，如图20-20所示，分为4层——Cocoa Touch层、Media层、Core Services层和Core OS层。

图 20-20 iOS API整体架构图

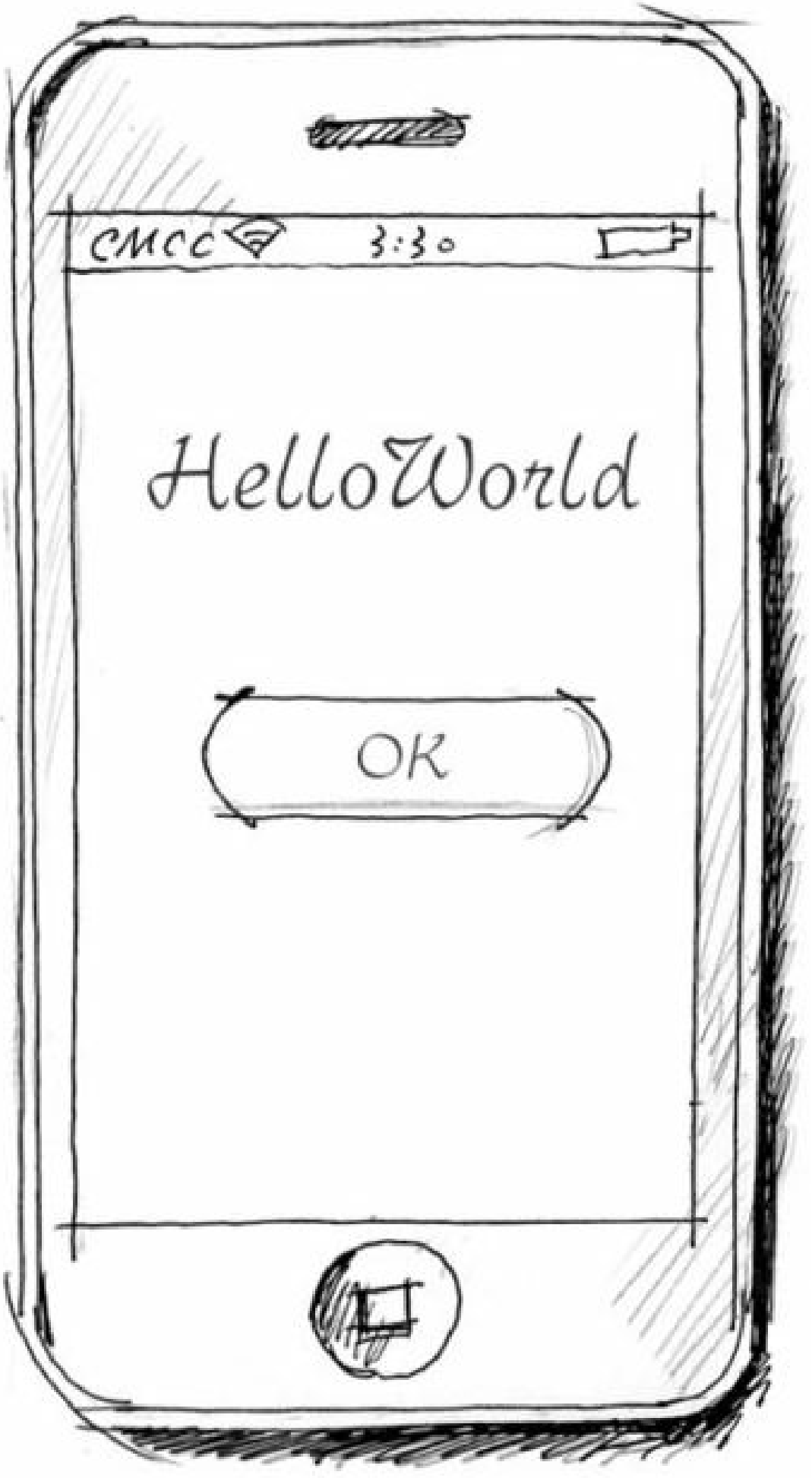
- Cocoa Touch层。该层提供了构建iOS应用的一些基本系统服务（如多任务、触摸输入和推送通知等）和关键框架。
- Media层。Media层提供了图形、音频、视频和AirPlay技术。
- Core Services层。该层提供了iCloud、应用内购买、SQLite数据库和XML支持等技术。
- Core OS层。该层提供了一些低级功能，开发中一般不直接使用它。

20.4　本章小结

通过对本章内容的学习，我们了解到了iOS开发的一些基础知识，包括开发环境Xcode、iOS SDK和iOS API等内容。通过一个基于iPhone的HelloWorld实例项目了解了iOS应用的运行基本原理。

20.5　同步练习

编程题：请参考下面的界面原型草图，编写一个iPhone程序。


CMCC
3:30
HelloWorld
OK

第 21 章　项目实战——基于分层架构的多版本iPhone计算器

这是本书的最后一章，也是本书的画龙点睛之笔。我想通过一个实际的iPhone计算器应用，使读者能够将本书前面讲过的知识点串联起来，熟悉Swift语言的特点，了解iOS应用开发的一般流程，掌握Objective-C语言与Swift语言混合搭配和调用，了解分层架构设计的重要性。

21.1 应用分析与设计

本节将从开发这个应用的分析和设计开始讲起，其中设计过程包括需求分析、原型设计、架构设计和应用设计。

21.1.1 应用概述

为了学习Swift语言，我们需要编写一个简单而又完整的应用，还要考虑到与UI相关的知识，因此我们选择了计算器应用。几乎每个平台都有计算器，有的计算器很简单，只有加、减、乘、除运算；有的可以进行复杂的数学计算，这是为工程计算准备的；还有的可以进行二进制、八进制和十六进制数的计算，是为程序员准备的。或许你不是程序员，也不是工程师，只是卖菜的小贩、杂货铺的小老板、整理报销差旅费的业务员，你可能只需要可以进行加、减、乘、除简单计算的计算器就足够了。

21.1.2 需求分析

由客户群决定了我们开发的计算器应用是主要实现简单的加、减、乘、除运算的计算器，而且考虑到iPhone与iPad应用场景的不同，计算器应用只考虑iPhone版本就可以了，同时考虑到单手操作，

我们只需设计和开发iPhone竖屏布局的计算器。

根据上面的功能描述，确定需求如下：

- 加运算
- 减运算
- 乘运算
- 除运算

这里我们采用用例分析的方法来描述用例图，如图21-1所示。

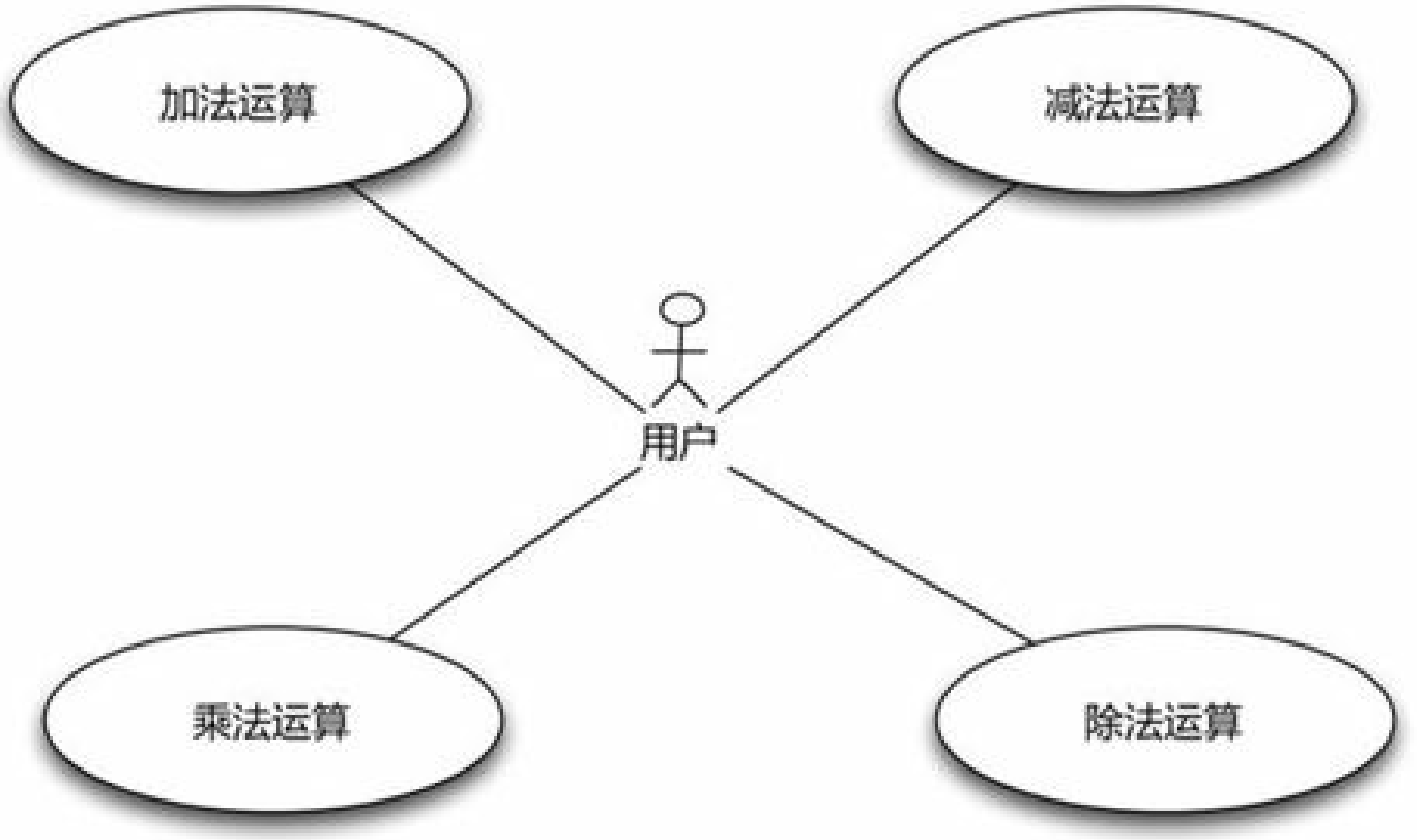


图 21-1　计算器应用用例图

21.1.3　原型设计

原型设计草图对于应用设计人员、开发人员、测试人员、UI设计人员以及用户都是非常重要的，该应用的原型如图21-2所示。

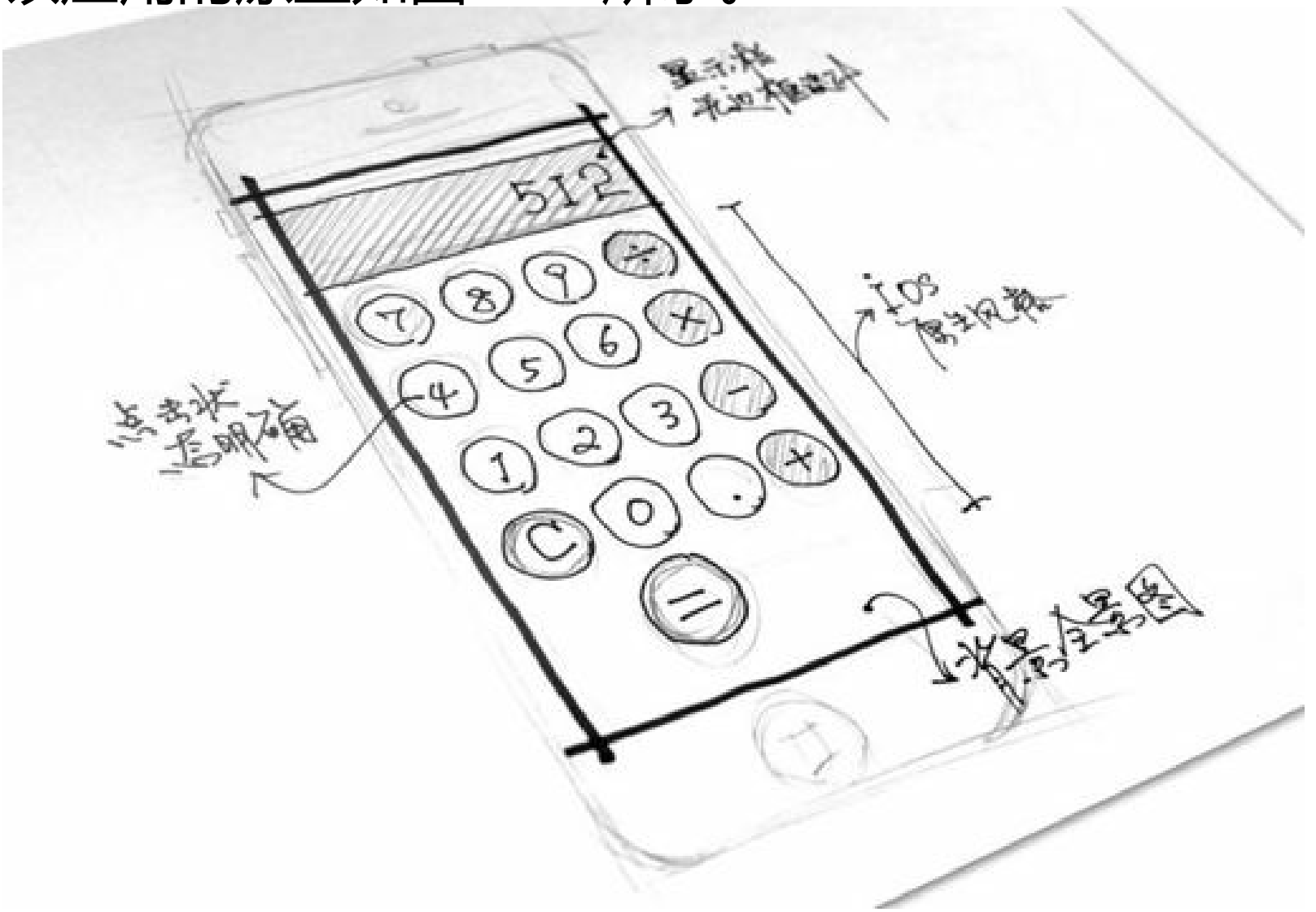


图 21-2　原型设计图

21.1.4　分层架构设计

为应用设计架构也是必需的，而且分层架构设计还可以提高项目管理的水平，科学化任务分配以及实施敏捷开发。架构设计就是应用的“骨架”，

搭建好“骨架”，再往里添砖加瓦。

首先，我们来了解一下应用程序的架构设计。软件设计的原则是提高软件系统的可复用性和可扩展性，架构设计采用层次划分的方式，这些层次之间是松耦合的，层次内部是高内聚的。图21-3是通用的低耦合应用架构图。

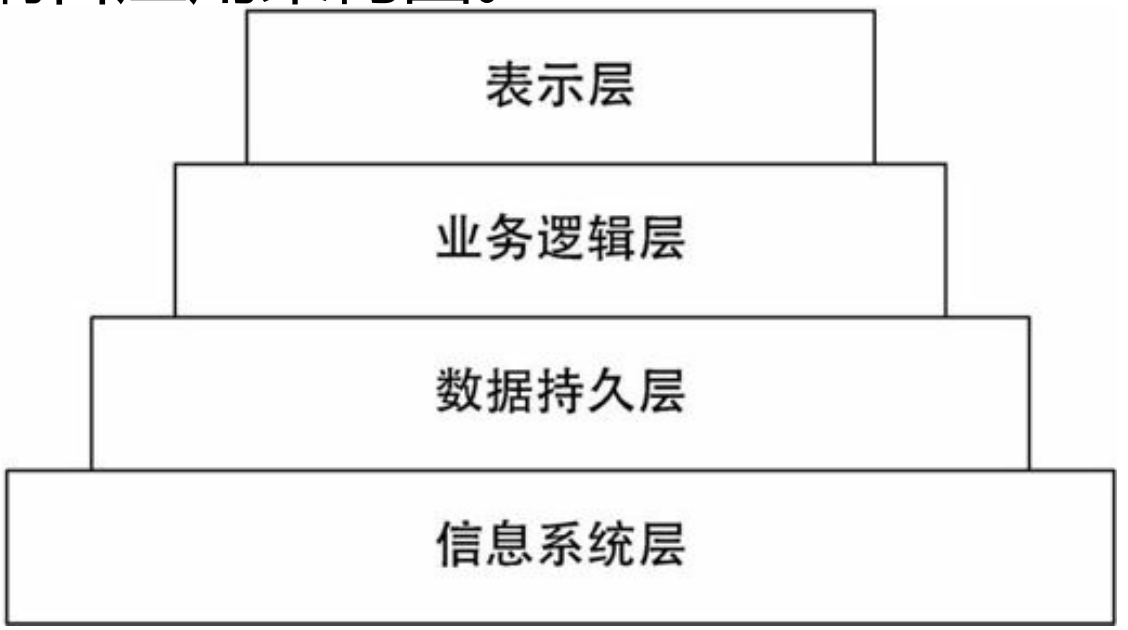


图 21-3　通用低耦合应用架构图

- **表示层**。用户与系统交互的组件集合。用户通过这一层向系统提交请求或发出指令，系统通过这一层接收用户请求或指令，待指令消化吸收后再调用下一层，接着将调用结果展现到这一层。表示层应该是轻薄的，不应该具有业务逻辑。
- **业务逻辑层**。系统的核心业务处理层。负责接收表示层的指令和数据，待指令和数

据消化吸收后，再进行组织业务逻辑的处理，并将结果返回给表示层。

- **数据持久层**。数据持久层用于访问信息系统层，即访问数据库或文件操作的代码应该只能放到数据持久层中，而不能出现在其他层中。
- **信息系统层**。系统的数据来源，可以是数据库、文件、遗留系统或者网络数据。

图21-3看起来像一个多层“蛋糕”，蛋糕师们在制作多层蛋糕的时候，先做下层再做上层，最后做顶层。没有下层就没有上层，这叫做“上层依赖于下层”。图21-3说明了，信息系统层是最底层，是所有层的基础，没有信息系统层就没有其他层。其次是数据持久层，没有数据持久层就没有业务逻辑层和表示层。再次是逻辑层，没有逻辑层就没有表示层，最后是表示层。也就是说，我们开发一个应用的顺序应该是，先是信息层，其次是数据持久层，再次是业务逻辑层，最后是表示层。

但是凡事都有例外，有时候应用不需要存储数据，也就不需要信息系统层，没有信息系统层，数据持久层也就没有必要了，数据持久层是为了访问数据信息而设计的。图21-3所示的架构会变成图

21-4所示的两层架构。

图 21-4　两层架构图

我们需要开发的计算器应用没有数据存储，因此可以采用图21-4所示的两层架构设计。

21.1.5　应用设计

架构设计定下来后，我们就可以对计算器应用进行设计，如图21-5所示，计算器应用表示层需要一个视图控制器类`ViewController`，一般是一个界面视图控制器类。图21-6所示为详细的类图。

图 21-5　计算器应用架构设计

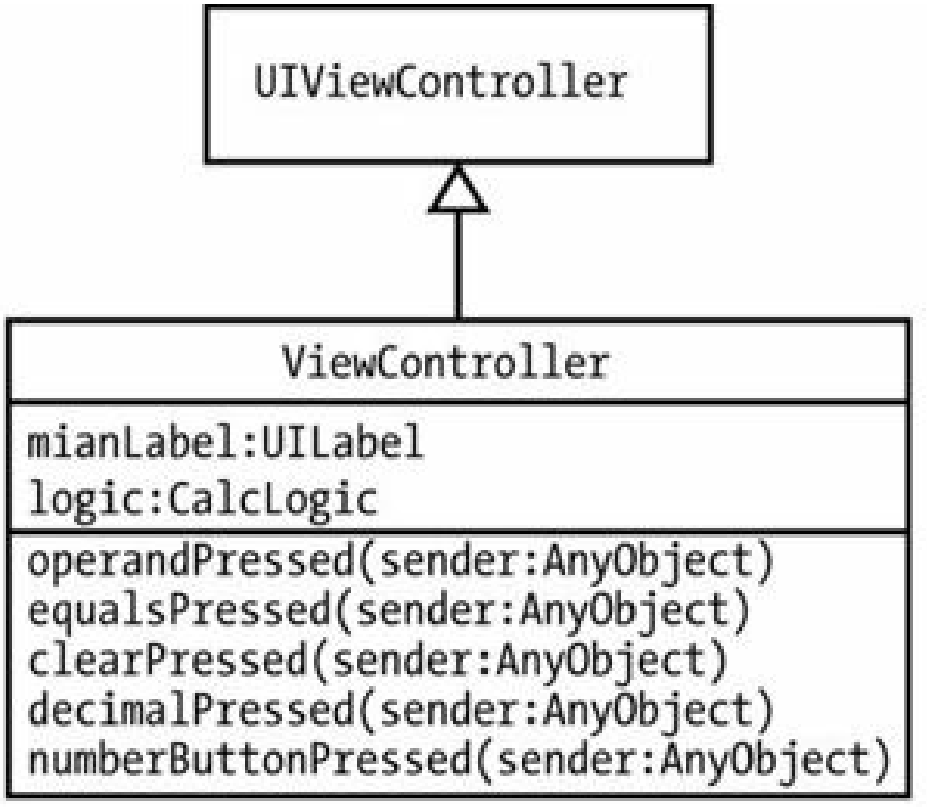


图 21-6　计算器应用类图

ViewController主界面的视图控制器和ViewController类中的成员如表21-1所示。

表21-1　ViewController类

类成员	成员类型	成员类型/返回值类型
operandPressed(sender: AnyObject)	方法	void
equalsPressed(sender: AnyObject)	方法	void
clearPressed(sender: AnyObject)	方法	void
decimalPressed(sender: AnyObject)	方法	void
numberButtonPressed(sender: AnyObject)	方法	void
mainLabel	属性	UILabel
logic	属性	CalcLogic

CalcLogic是计算业务逻辑处理类，CalcLogic类中的成员如表21-2所示。

表21-2 CalcLogic类述

类成员	成员类型	成员类返回值类
init ()	方法	无
updateMainLabelStringByNumberTag (tag : Int, mainLabelString : String)-> String	方法	Strin
doesStringContainDecimal(string : String)-> Bool	方法	Bool
calculateByTag(tag : Int, mainLabelString : String)->String	方法	void

clear()	方法	void
deinit	方法	无
lastRetainValue	属性	Doubl
opr	属性	Opera
isMainLabelTextTemporary	属性	Bool

21.2 创建工程

我们首先需要创建一个工程。启动Xcode 6，然后单击File→New→Project菜单，在打开的Choose a template for your new project界面中选择“Single View Application”工程模板（如图21-7所示）。

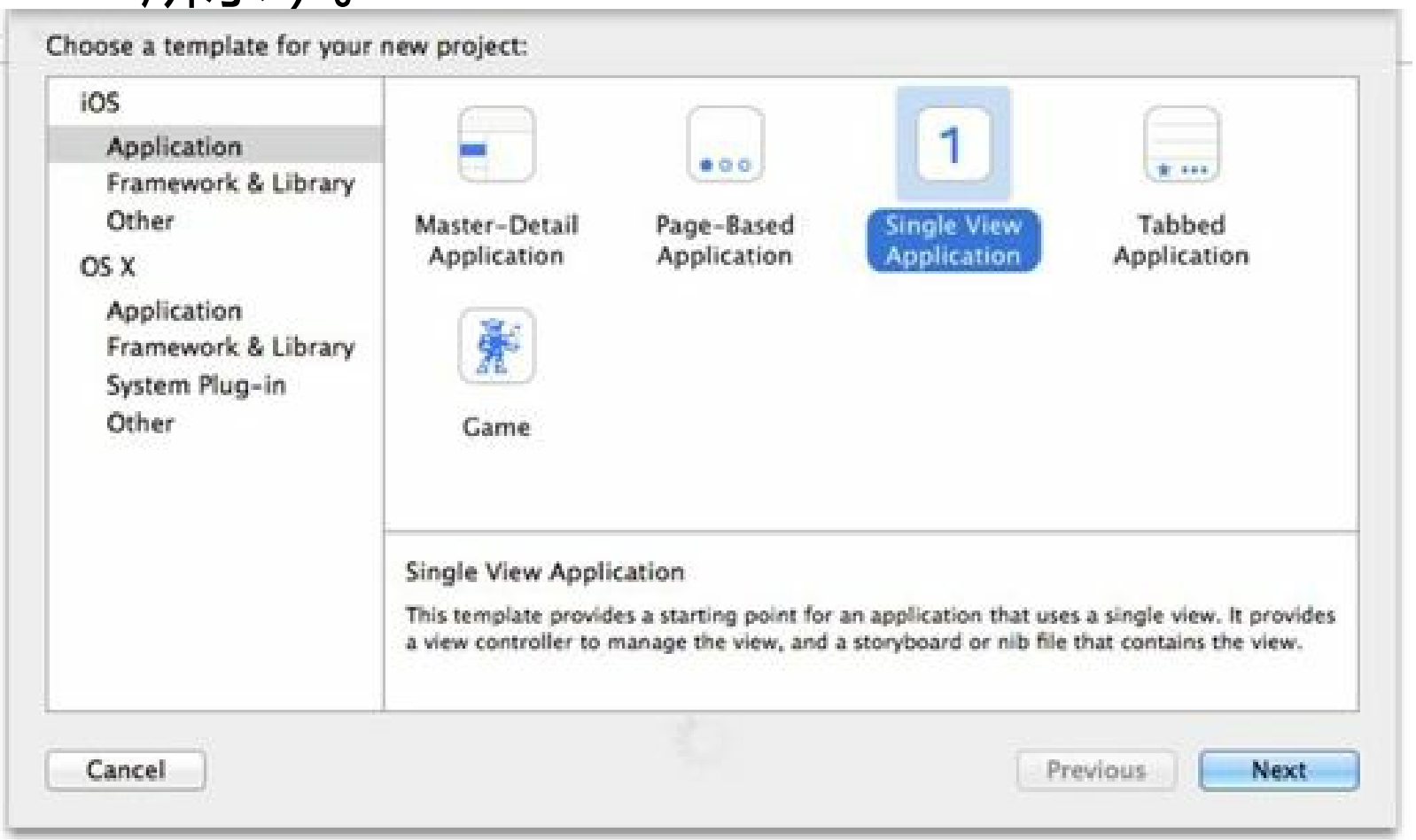


图 21-7 选择工程模板

接着单击“Next”按钮，随即出现如图21-8所示的界面。在Product Name中输入“Calculator”，在Language中选择Swift，在Devices中选择iPhone。其他的内容可以根据自己的情况填写。

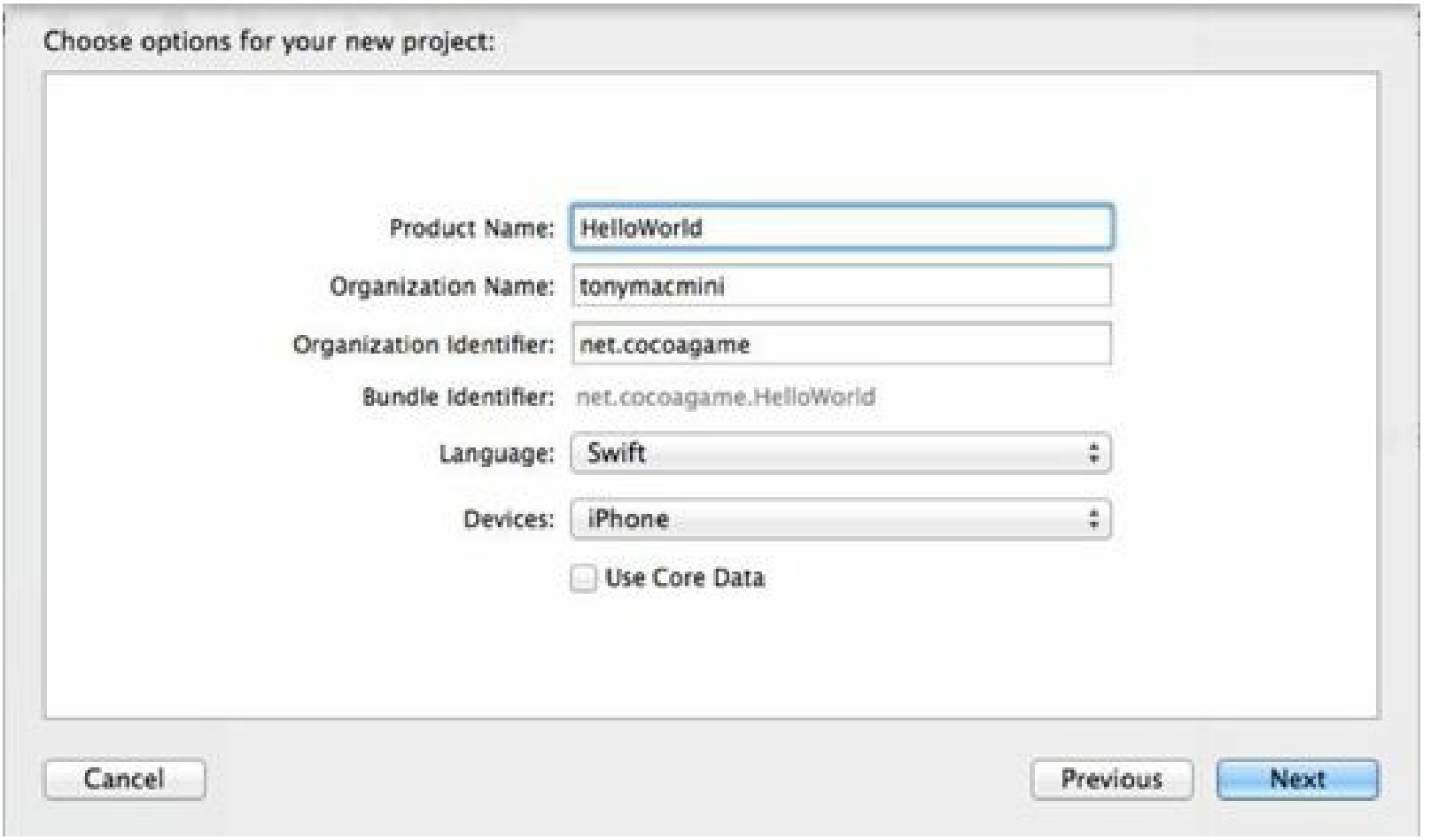


图 21-8　新工程中的选项

设置完相关的工程选项后，单击“Next”按钮，进入下一级界面。根据提示选择存放文件的位置，然后单击“Create”按钮，将出现如图21-9所示的界面。

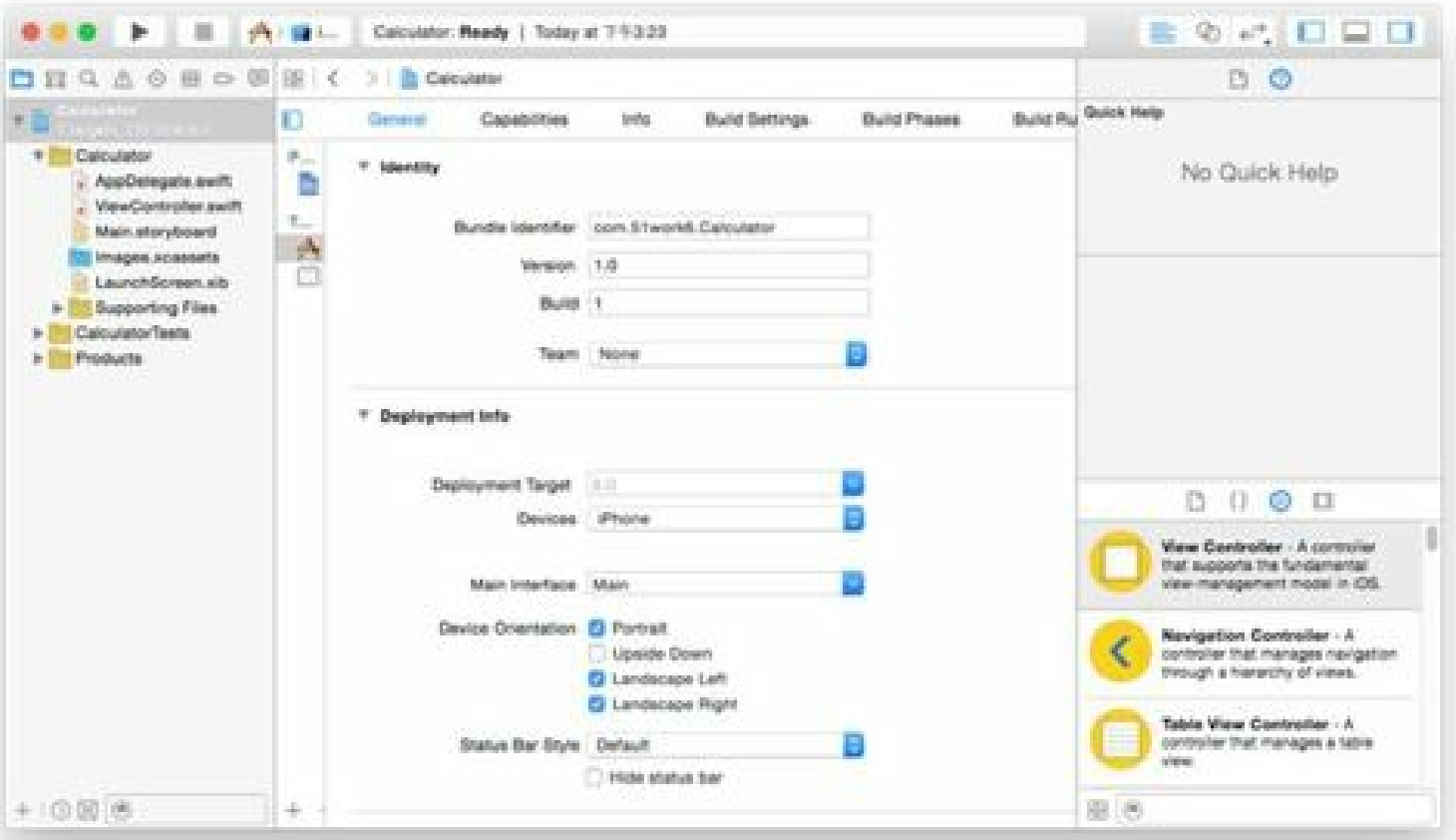


图 21-9　新创建的工程

工程创建完成后，就可以进行业务逻辑层的开发了。

21.3 业务逻辑层开发

表示层依赖于业务逻辑层，因此我们最好先开发业务逻辑层。由于我们采用纯Swift编写应用，因此业务逻辑层也要采用Swift语言编写。

21.3.1 创建CalcLogic.swift文件

首先需要创建一个Swift源代码文件。具体操作方法为：右击工程名，在弹出的快捷菜单中选择“New File...”，然后再打开如图21-10所示的Choose a template for your new file对话框。

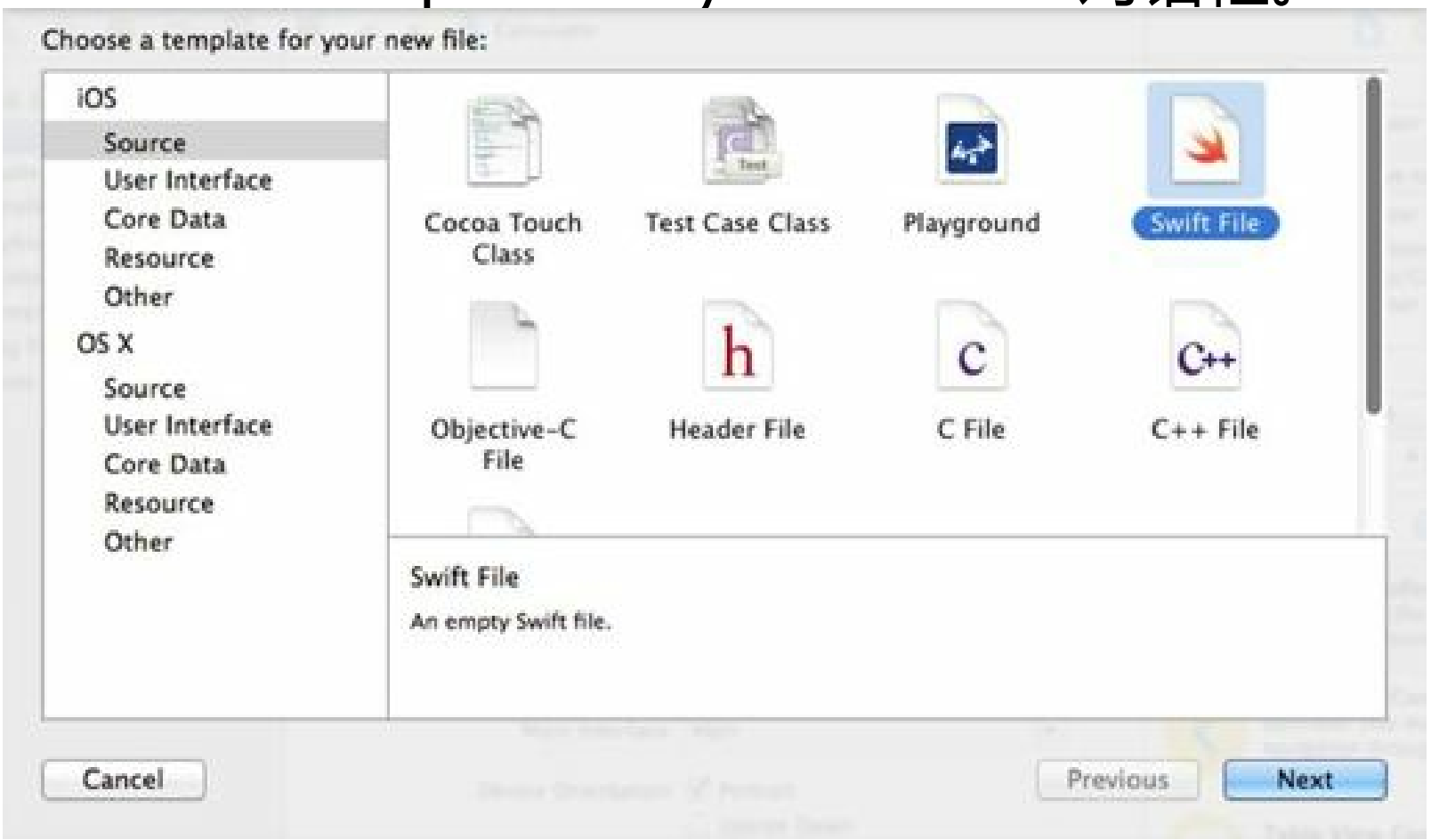


图 21-10 选择新创建文件模板

选择iOS→Source→Swif File文件模板，单

击“Next”按钮，弹出如图21-11所示的对话框，在Save As中输入文件名“CalcLogic.swift”。在这里我们还可以选择文件保存目录，选择好之后单击“Create”按钮创建文件。如果成功创建文件，就会看到如图21-12所示的界面，一个空的Swift文件就创建好了。

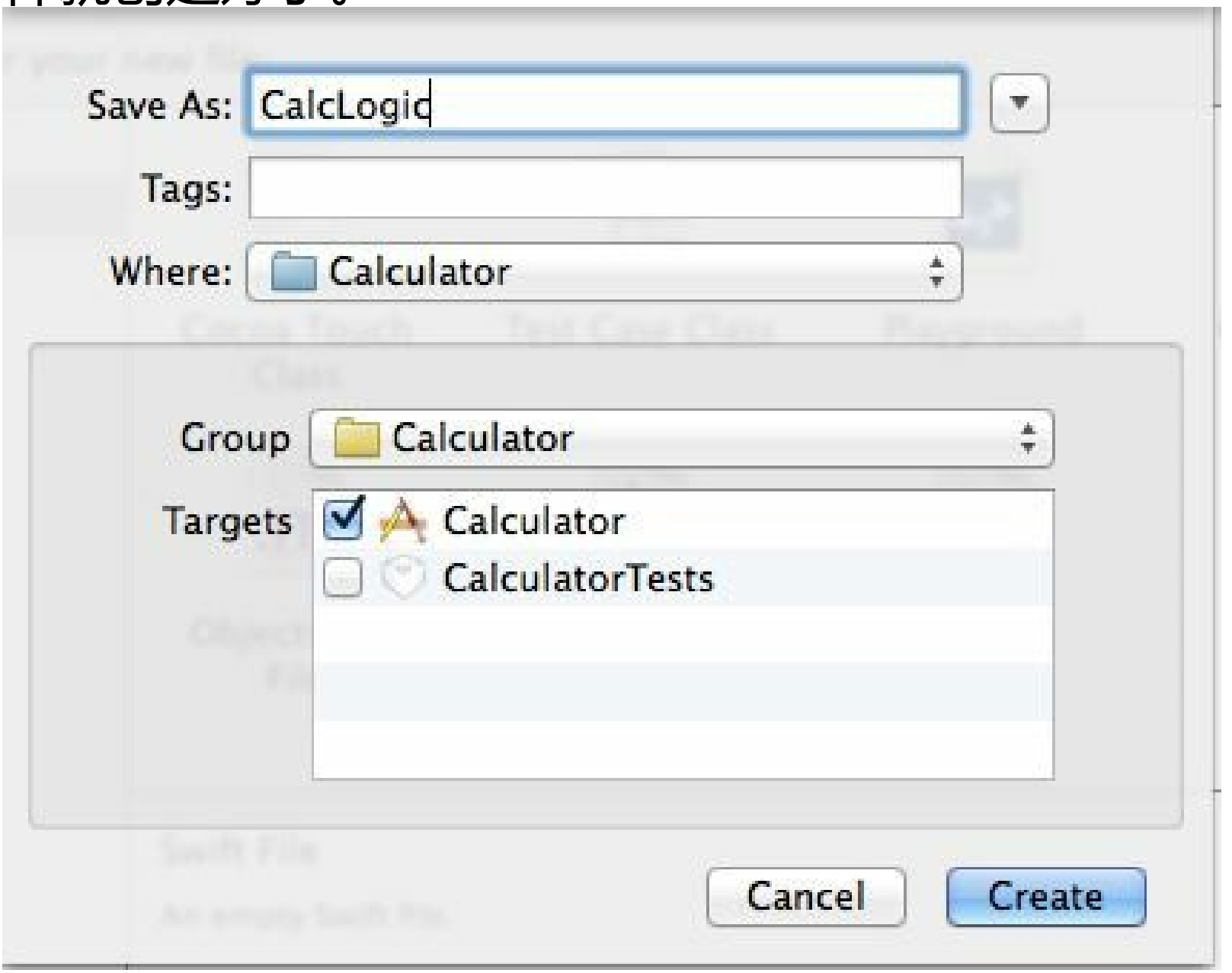


图 21-11　创建文件对话框

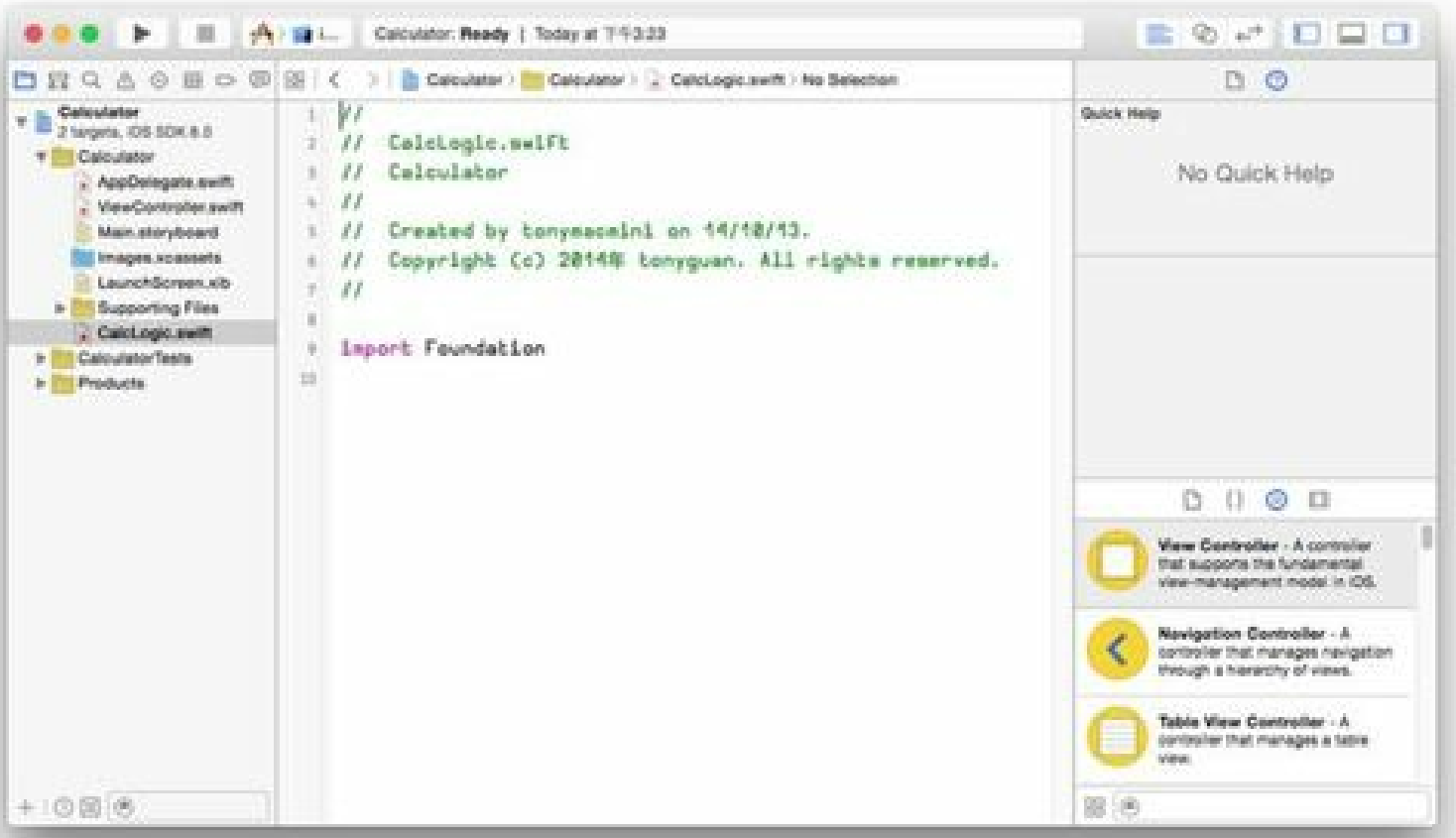


图 21-12　创建Swift文件成功

21.3.2　枚举类型Operator

枚举类型Operator中包含了加、减、乘、除等运算符成员值，用来表示用户所点击的运算符。

Operator代码如下：

```
enum Operator : Int {
    case Plus = 200, Minus, Multiply, Divide
    case Default = 0
}
```

Operator枚举类型中定义了5个原始值，均是Int类型。Plus表示加运算符，值为200；Minus表示减运算符，值为201；Multiply表示乘运算符，值为202；Divide表示除运算符，值为203；Default表示默认值，值为0。这些原始值与在Interface Builder设计器中加、减、乘、除运算符按钮的tag属性值相同，这样的设计便于计算和判断。

Default成员需要注意，在没有点击任何运算符按钮之前，或者是在用户按了等号按钮计算完成之后，运算符的值都是Default，因此将Default的值设为0。

21.3.3 CalcLogic类中属性

下面我们参考图21-6以及表21-1编写CalcLogic类中属性。在CalcLogic类中添加如下代码：

```
class CalcLogic {

    //保存上一次的值
```

```
    var lastRetainValue : Double
    //最近一次选择的操作符（加、减、乘、除）
    var opr : Operator
    //临时保存MainLabel内容，为true时，输入数字
MainLabel内容被清为0
    var isMainLabelTextTemporary : Bool
    ......
}
```

这几个属性都是存储属性，没有设置默认值，所以我们要在构造器中设置它们的默认值。各个属性的含义已在表21-1中说明了，这里不再赘述。

21.3.4 CalcLogic类中构造器和析构器

CalcLogic类中构造器可以初始化存储属性，由于CalcLogic类中存储属性初始化比较简单，也不需要参数初始化，所以我们设计了无参数构造器init()，代码如下：

```
init () {
    println("CalcLogic init")
    lastRetainValue = 0.0
    isMainLabelTextTemporary  = false
    opr = .Default
}
```

我们在无参数构造器init()中初始化了3个存储属性，如果不在构造器中初始化属性，也可以直接在属性声明时初始化。

与构造器对应的是析构器，析构器代码如下：

```
deinit {
        println("CalcLogic deinit")
   }
```

在析构器中也没有什么资源需要释放，因此在本例中我们只是简单地打印输出字符串。

21.3.5 CalcLogic类中更新主标签方法

CalcLogic类中定义了更新主标签方法，当用户点击数字按钮后，需要更新主标签内容，表示层给该方法传递的参数是数字按钮的tag属性和当前主标签内容。

这个过程的计算是比较复杂的：一方面，需要判断当前主标签内容是否是临时的，由于是进行数学计算，当前主标签内容就是计算的结果，当再次

输入数字的时候，该数字应该替换主标签内容。另一方面，需要进行小数点的判断，如果已经有小数点了，就不能将小数点追加到主标签之后。在上述两种情况之外，就是正常的将数字追加到主标签之后。

该方法的代码如下：

```
func updateMainLabelStringByNumberTag(tag : Int,
    withMainLabelString mainLabelString : String)->String {                    ①

    var string = mainLabelString                                                ②

    if (isMainLabelTextTemporary)  {                                            ③
        string = “0”                                                            ④
        isMainLabelTextTemporary = false                                        ⑤
    }

    let optNumber = tag - 100                                                   ⑥
    //把String转为double
    var mainLabelDouble = (string as NSString).doubleValue                      ⑦
```

```
    if mainLabelDouble == 0 &&
doesStringContainDecimal(string) == false {
⑧
        return String(optNumber)
⑨
    }
    let resultString = string +
String(optNumber)
⑩

    return resultString
}
```

上述代码第①行定义方法，其中第一个参数是tag，它是按钮的标签属性。第二个参数是mainLabelString主标签内容，withMainLabelString是外部参数名。第②行代码是将参数mainLabelString赋值给字符串变量string。第③行代码是判断当前主标签内容是否是临时的，如果是临时的，就设置字符串变量string = “0”，并且设置isMainLabelTextTemporary = false。

第⑥行代码let optNumber = tag - 100

是通过按钮tag属性计算操作数，我们在Interface Builder设计器中设置0~9数字按钮的tag属性是100~109，因此tag -100计算要进行计算的操作数，如果我们点击的是数字按钮8，它的tag属性是108，要计算的操作数就是8。

第⑦行代码是将字符串string转换为Double类型。字符串string类型是Swift的String类型，不能直接转换为Double类型，需要借助于Foundation框架中的NSString类进行转换，Foundation框架是用Objective-C语言编写的。在Swift中，只要遵守一定的规范，Swift和Objective-C之间可以互相调用。因此有些功能在Swift中实现很困难，我们可以通过Objective-C的一些类实现该功能，然后再把结果返回给Swift调用程序。

虽然是在Swift中调用Objective-C语言，但是语法还是Swift的语法。第⑦行的(string as NSString).doubleValue语句是Swift的语法，首先是通过string as NSString语句将Swift的字符串String转换为Objective-C的字符串NString。doubleValue是NSString类的实例属性，能够将字符串转换为Double类型。

上述代码第⑧行是判断主标签内容转换为Double类型后是否等于0，而且通过doesStringContainDecimal方法判断是否有小数点。第⑨行代码是将Double类型转换为String类型。第⑩行代码是将字符串拼接起来，并返回给调用程序。

21.3.6 CalcLogic类中判断是否包含小数点方法

CalcLogic类中定义了判断字符串中是否包含小数点方法，在很多情况下都需要判断一个字符串中是否已经包含了小数点，如果包含则用户不能再输入小数点。总之这个方法是暴露给表示层，至于表示层判断它的目的是什么CalcLogic类中并不关心。

该方法的代码如下：

```
func doesStringContainDecimal(string : String)-
>Bool {
    for ch in string {
        if ch == "." {
            return true
        }
```

```
    }
    return false
}
```

Swift中的字符串String类没有提供直接判断字符串中是否包含某个特定字符的方法。我们可以使用循环遍历字符串，判断如果有字符为小数点“.”，则返回true，否则返回false。

21.3.7 CalcLogic类中计算方法

CalcLogic类中定义了计算方法，该方法的代码如下：

```
func calculateByTag(tag : Int,
withMainLabelString mainLabelString : String)-
>String {              ①

    //把String转为为double
    var currentValue = (mainLabelString as
NSString).doubleValue

    switch Operator {
②
    case .Plus:
        lastRetainValue += currentValue
```

```
    case .Minus:
        lastRetainValue -= currentValue
    case .Multiply:
        lastRetainValue *= currentValue
    case .Divide:
        if currentValue != 0 {
            lastRetainValue = currentValue
        } else {
            opr = .Default
            isMainLabelTextTemporary = true
            return "错误"
        }
    Default:
        lastRetainValue = currentValue
③
    }

    /记录当前操作符，下次计算时使用
    opr = Operator(rawValue:tag)!
④

    let resultString = NSString(format: "%g",
lastRetainValue)
⑤

    isMainLabelTextTemporary = true
⑥

    return resultString
}
```

上述代码第①行是定义方法，其中第一个参数是tag，它是按钮的标签属性。第二个参数是mainLabelString主标签内容。

第②行代码是switch 多分支语句，通过switch语句判断用户点击了哪个运算符，然后进行计算，计算的结果被保存到存储属性lastRetainValue中。注意当分支为.Divide（除法运算）时，如果除数为0，计算发生错误，则将字符串“错误”返回。

第③行代码是默认分支，把当前值currentValue赋值给lastRetainValue 属性。

第④行代码是将枚举的原始值转换为成员值，其中的tag是按钮tag属性值。表达式后面的“!”表示对其进行强力拆封，当tag无法转换时则把nil赋值给Operator。

第⑤行代码是将Double类型的lastRetainValue格式化为字符串。格式化过程使用Objective-C的NSString类协助完成，format:参数是指定的格式，%g是用于格式

化浮点类型数值。与%f和%e不同的是，%g可以防止出现.00现象，.00现象就是，当输入的是一个整数，例如10，经过格式化后变成了10.00，在计算器应用中是不希望出现.00现象的。

第⑥行代码是给属性isMainLabelTextTemporary赋值为true，这是因为计算结束后，用户再输入数字时，原来主标签内容将被清除。通过布尔变量isMainLabelTextTemporary判断是否已清除主标签内容。

21.3.8 CalcLogic类中清除方法

当用户点击C（清除）按钮时，需要将CalcLogic类恢复到初始状态。我们可以通过重新实例化CalcLogic类来达到这一目的，但是重新实例化会耗费更多系统资源。如果不重新实例化，我们可以编写一个方法类似于构造器初始化存储属性，该方法代码如下：

```
func clear() {
    lastRetainValue = 0.0
    isMainLabelTextTemporary  = false
    opr = .Default
```

```
}
```

该方法所做之事与init构造器是一样的，都是初始化3个存储属性。表示层通过调用该方法将CalcLogic类重新初始化。

21.4　表示层开发

从客观上讲，表示层开发的工作量是很大的，工作要做得很细致。

21.4.1　添加图片资源

首先需要把应用中要用的图片等资源添加到工程中。具体步骤如图21-13所示，右键选择Supporting Files组，选择菜单中的Add Files to “Calculator”...弹出选择文件对话框。如图21-14所示，选择我们提供的图片文件夹（images），并在Destination中选中Copy items if needed，这样可以将文件或文件夹复制到我们的工程目录中。在Add to folders中选择Create groups，可以在Xcode中创建images组。然后在Add to targets中选择Calculator。

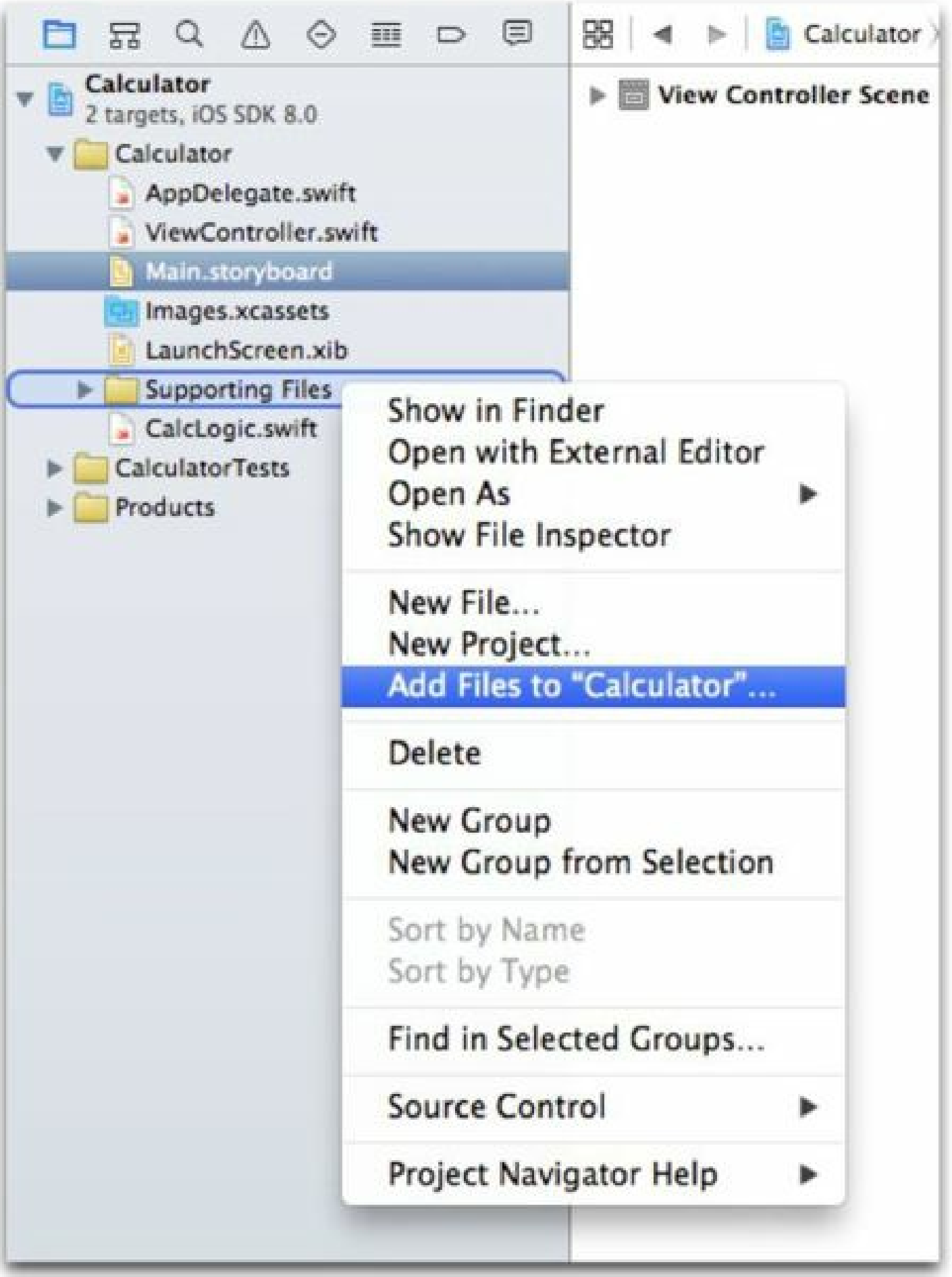

Calculator
Calculator
2 targets, iOS SDK 8.0
Calculator
AppDelegate.swift
ViewController.swift
Main.storyboard
Images.xcassets
LaunchScreen.xib
Supporting Files
CalcLogic.swift
CalculatorTests
Products
View Controller Scene
Show in Finder
Open with External Editor
Open As
Show File Inspector
New File...
New Project...
Add Files to "Calculator"...
Delete
New Group
New Group from Selection
Sort by Name
Sort by Type
Find in Selected Groups...
Source Control
Project Navigator Help

图 21-13　添加图片资源文件

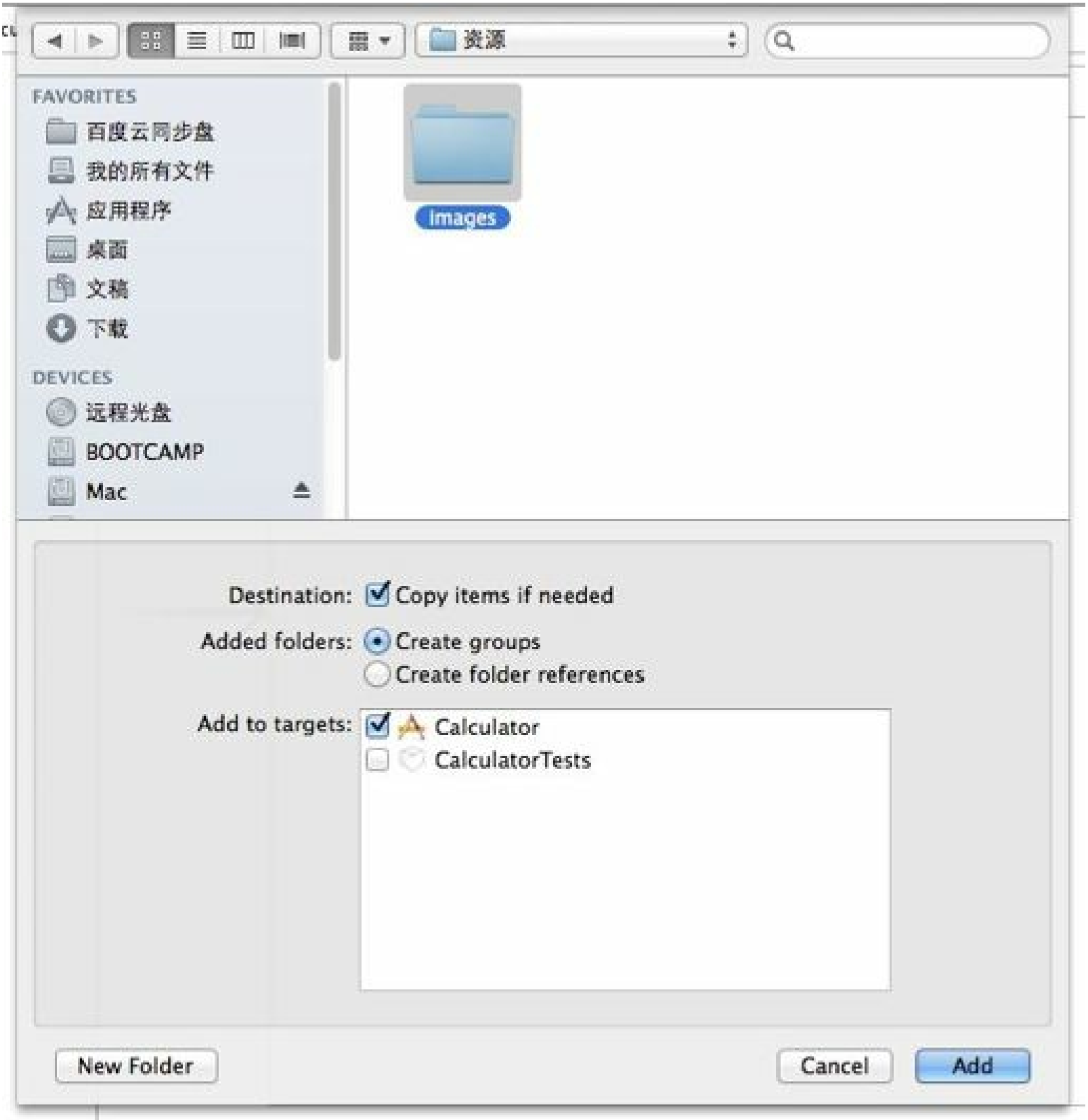


图 21-14　添加文件或文件夹对话框

由于图片资源文件很多，下面我们通过表21-3来解释一下它们的含义。

表21-3　图片资源文件

文件名	说明
0.png~9.png	0~9数字按钮默认状态显示的图片
0-down.png~9-down.png	0~9数字按钮高亮状态显示的图片
c.png	清除按钮默认状态显示的图片
c-down.png	清除按钮高亮状态显示的图片
Decimal.png	小数点按钮默认状态显示的图片
Decimal-down.png	小数点按钮高亮状态显示的图片
Plus.png	加按钮默认状态显示的图片
Plus-down.png	加按钮高亮状态显示的图片
Minus.png	减按钮默认状态显示的图片
Minus-down.png	减按钮高亮状态显示的图片

Multiply.png	乘按钮默认状态显示的图片
Multiply -down.png	乘按钮高亮状态显示的图片
Divide.png	除按钮默认状态显示的图片
Divide-down.png	除按钮高亮状态显示的图片
Equals.png	等号按钮默认状态显示的图片
Equals-down.png	等号按钮高亮状态显示的图片
icon.png	计算器应用图标
Cal.png	主界面背景

21.4.2 改变设计界面大小

在开始界面设计之前，需要改变设计界面的大小。我们预计应用会在iPhone 5及之后的设备上发布，而且是竖屏，因此参考20.2节，通过图21-15所示的步骤改变设计界面大小。

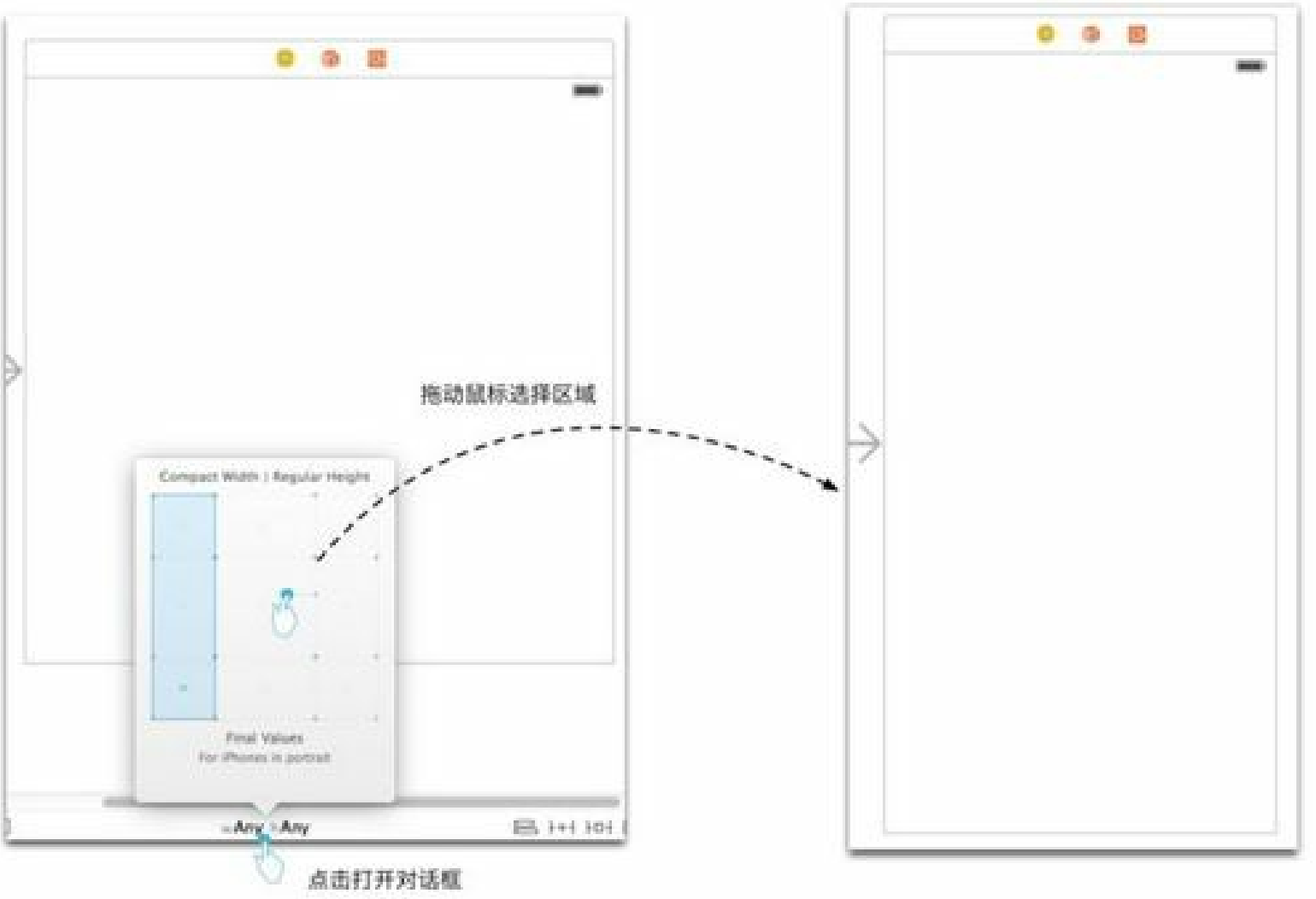
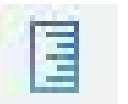


图 21-15　改变设计界面大小

21.4.3　添加计算器背景

主界面中的计算器背景是一张图片，把图片放到界面中，需要将UIImageView控件添加到设计界面，具体操作可以参考20.2节。拖曳控件UIImageView到设计界面，然后选择属性检查器 ，选择Image View→Image为Cal.png。

为了保证UIImageView控件大小及高宽比与背

景图片Cal.png保持一致，需要修改UIImage View尺寸等属性。选中右边的尺寸检查器  ，如图21-16所示的数值，设置坐标X和Y的属性，设置宽和高属性Width和Height。

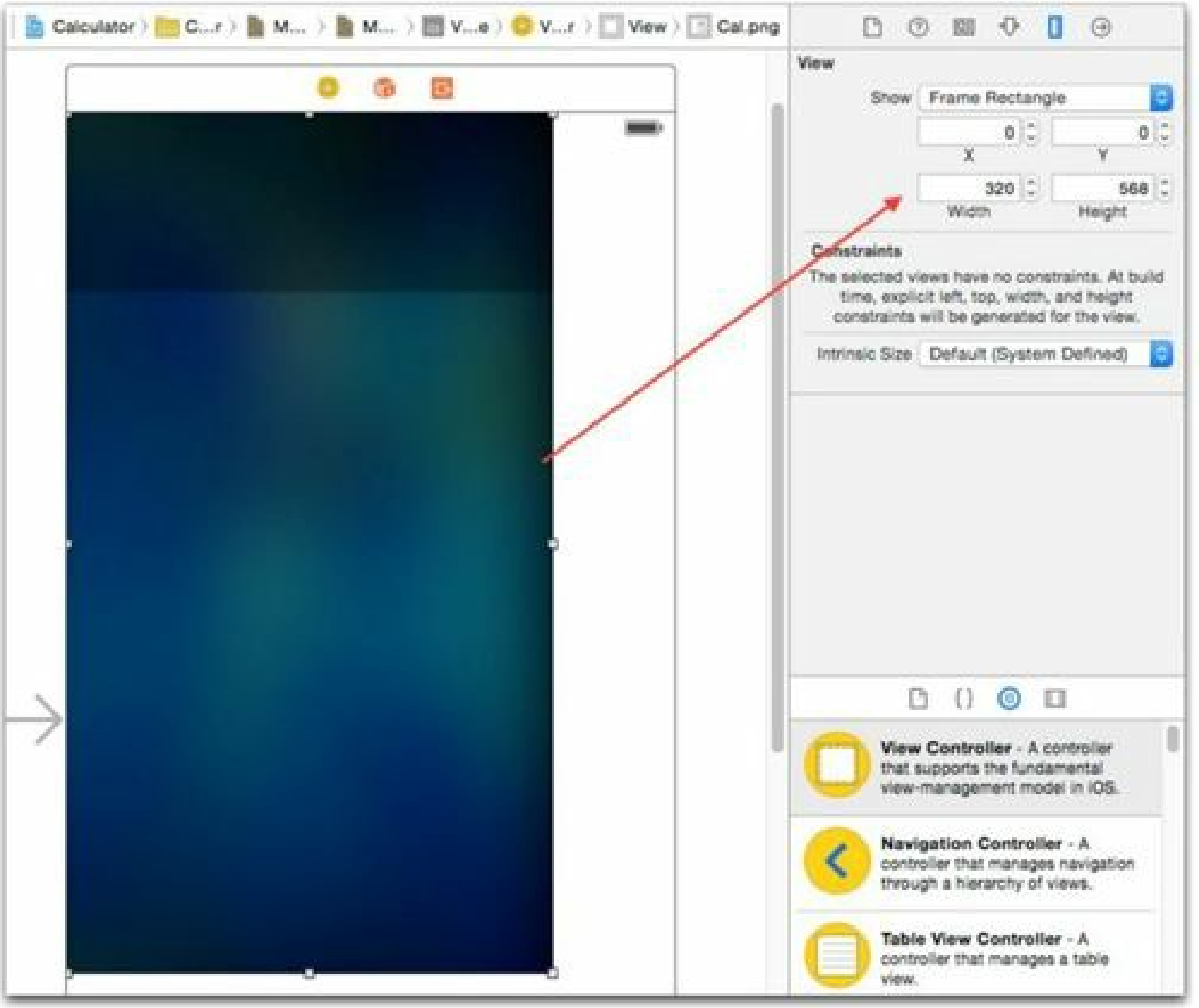


图 21-16　修改UIImageView尺寸等属性

提示　原始背景图片大小是640×1136（像素），放到Retina显示屏幕中UIImageView控件大小就是320×568（像素），即高和宽减少了一半。这样设置后我们会发现设计界面下边还有空白（见图21-

16），这是因为我们选择的设计界面高度是640点（1280像素），这没有关系，在iPhone 5设备上，下边多出的空白不会显示在屏幕上。

21.4.4　在设计界面中添加主标签

主界面中的主标签是用来显示输入数字和计算结果的控件，我们可以使用UILable控件。我们需要从控件库中拖曳一个Label到设计界面，并拖动它摆放到图21-17所示的位置和大小。如果你觉得拖曳不准确，可以设置它的尺寸属性，尺寸属性值X=0、Y=0、Width=320和Height=119。

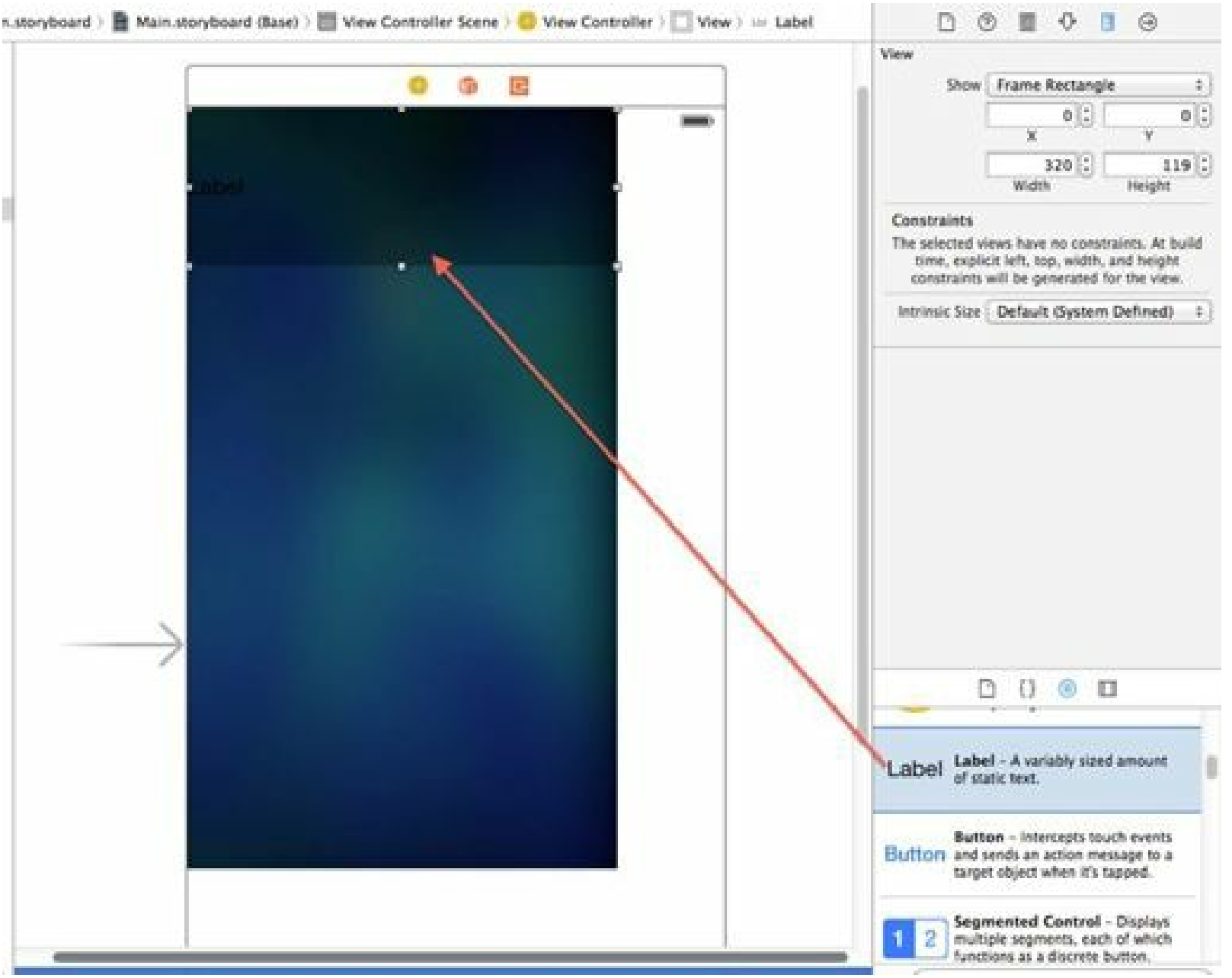


图 21-17　添加Label

摆放好主标签控件的位置后，就需要设置它的文字相关属性。首先是颜色，选择主标签控件，打开属性检查器 ，设置Label→Color属性为白色，如图21-18所示。再设置Label→Alignment属性为右对齐，如图21-19所示。

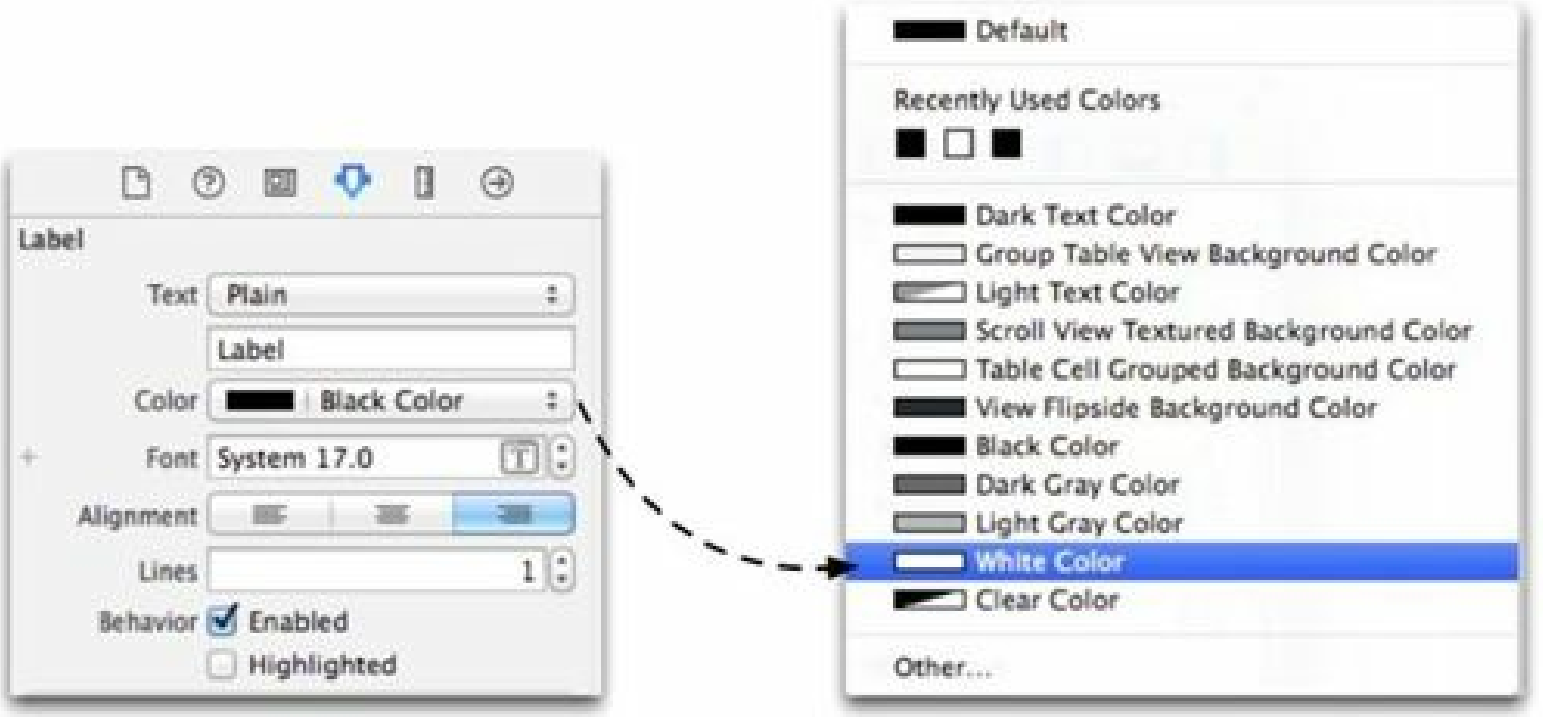


图 21-18　设置主标签文字颜色

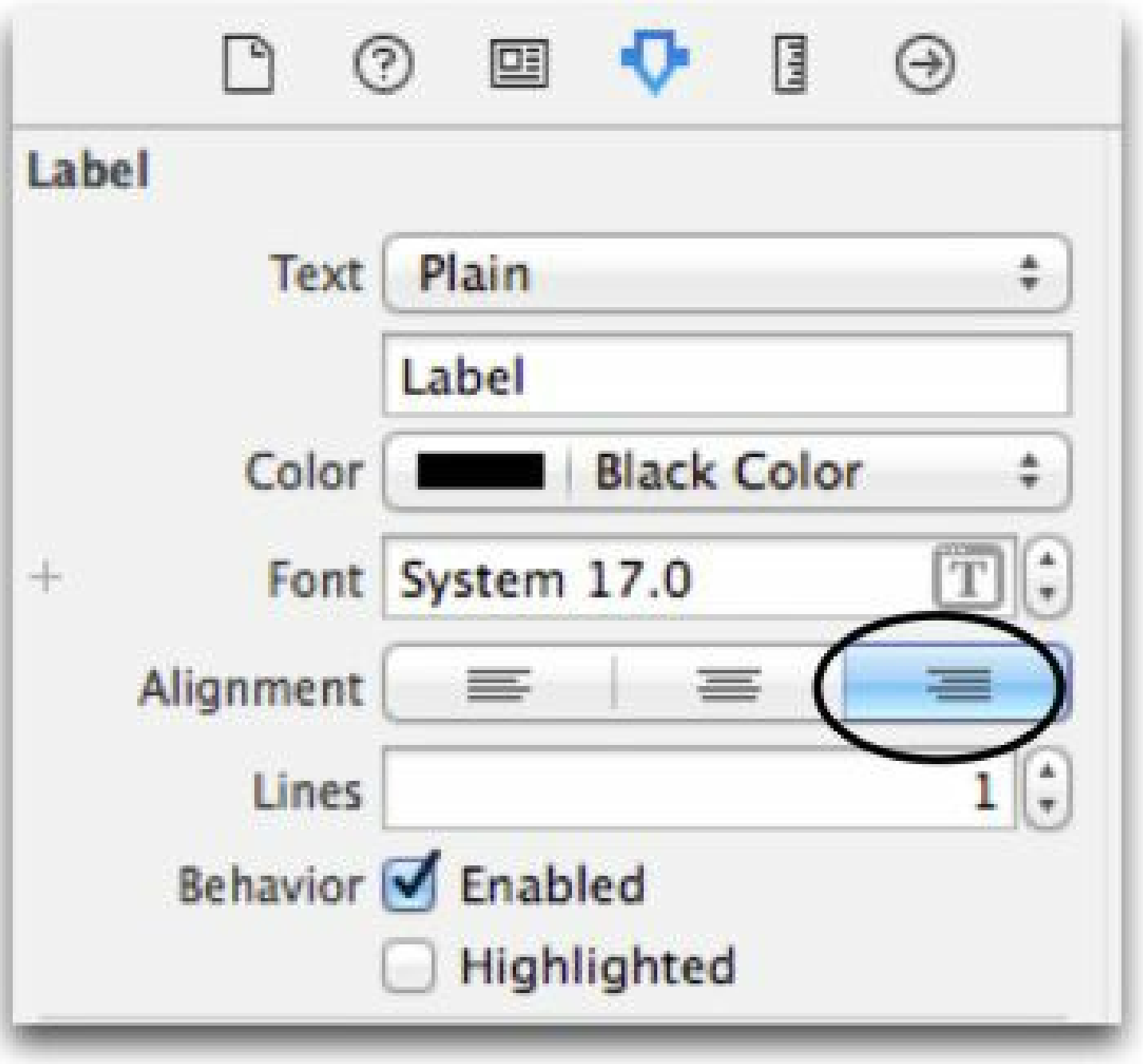


图 21-19　设置主标签文字对齐

然后设置文字字体，选择Label→Font属性，如图21-20所示，在弹出的对话框中设置Font为Custom（自定义），Family为Helvetica Neue，Style为Thin，Size为50。

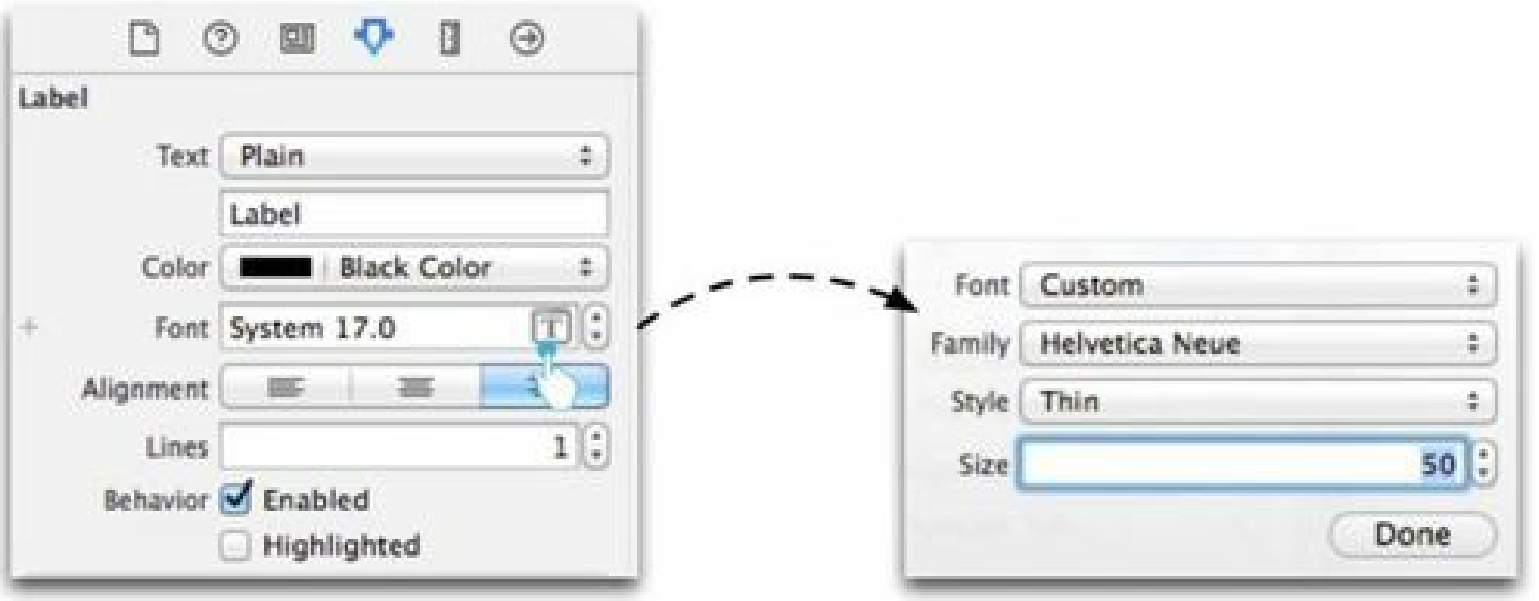


图 21-20　设置主标签字体

当设置完成这些属性之后，我们需要选择Label标签，拖曳Label标签轮廓填充屏幕的宽度，效果如图21-21所示。

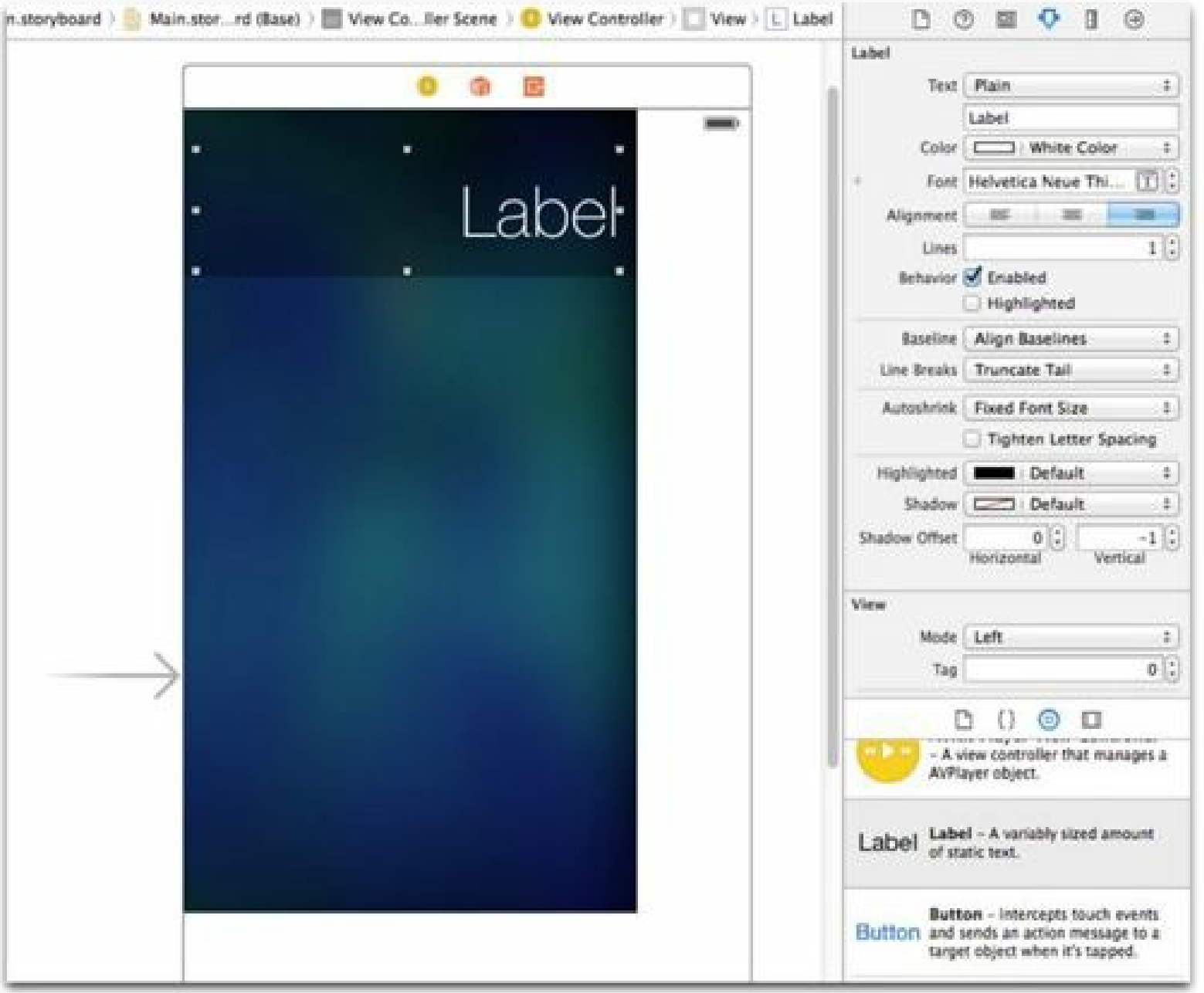


图 21-21　效果比较

21.4.5　在设计界面中添加按钮

从原型上看，主界面有17个按钮，我们先详细介绍其中一个按钮的设计过程，其他的以此类推。左上角的按钮是7，我们先来设计按钮7。从控件库中拖曳Button控件到设计界面，如图21-22所示。

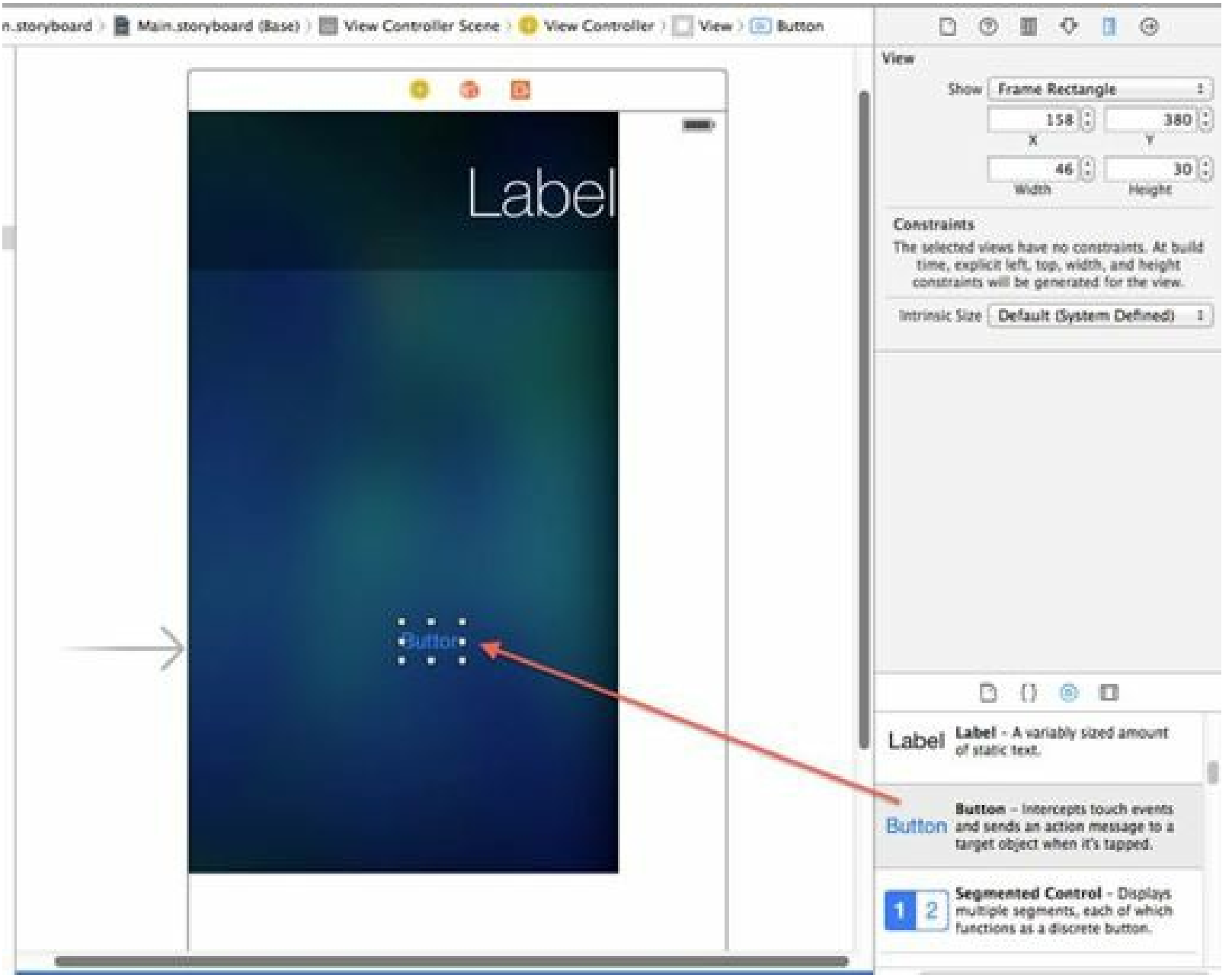


图 21-22　添加Button控件

然后我们需要设置按钮的图片，由于按钮有4种状态，分别是Default（默认）状态、Highlighted（高亮）状态、Selected（选择）状态和Disabled（不可用）状态。每种状态都对应一套按钮风格，包括颜色、背景颜色、图片、背景图片以及字体等。状态切换如图21-23所示。

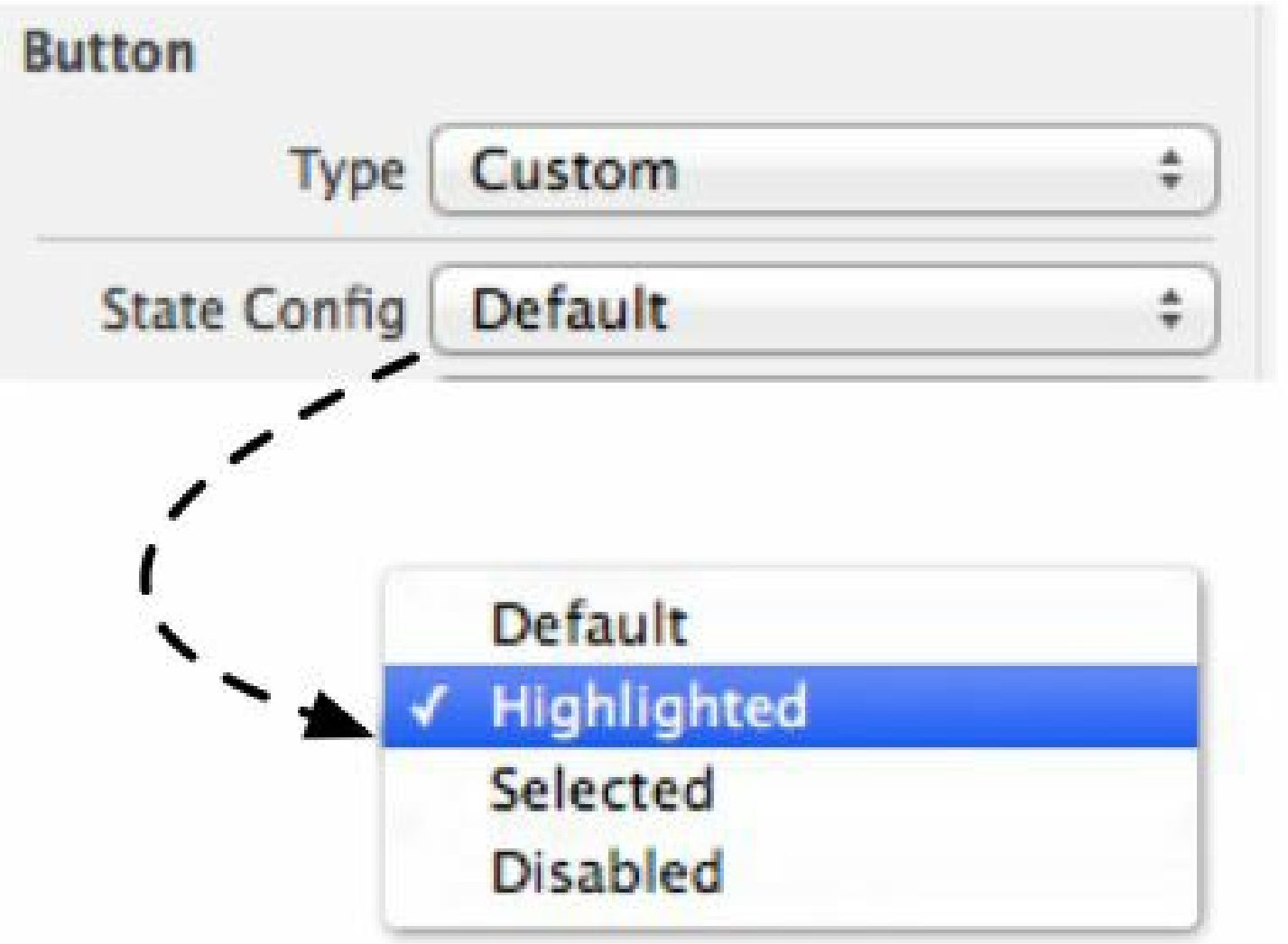


图 21-23　切换Button状态

在本例中，我们只设置默认状态下Image属性为7-down.png，高亮状态下Image属性为7-down.png。如图21-24所示，选择高亮状态，然后在下面的Image属性下拉列表中选择7-down.png。

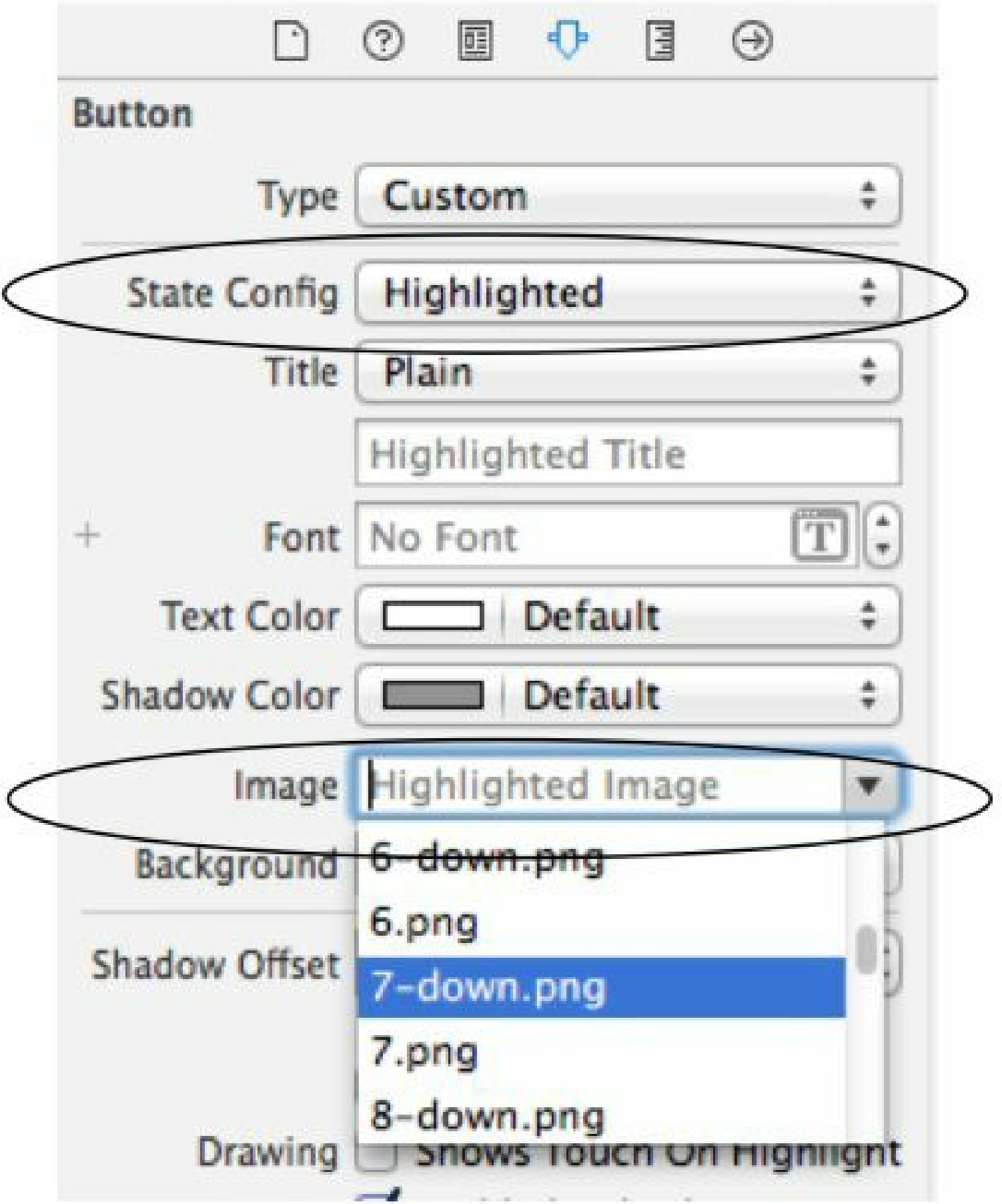

Button
Type
Custom
State Config
Highlighted
Title
Plain
Highlighted Title
Font
No Font
Text Color
Default
Shadow Color
Default
Image
Highlighted Image
Background
6-down.png
6.png
Shadow Offset
7-down.png
7.png
8-down.png
Drawing

图 21-24　选择特定状态下图片

接下来还需要设置按钮的大小，打开尺寸检查器  ，如图21-25所示，宽和高属性Width和Height设置为80。X和Y属性不需要在此设置。我们需要手动拖曳到左上角，具体位置参考原型设计图21-2。

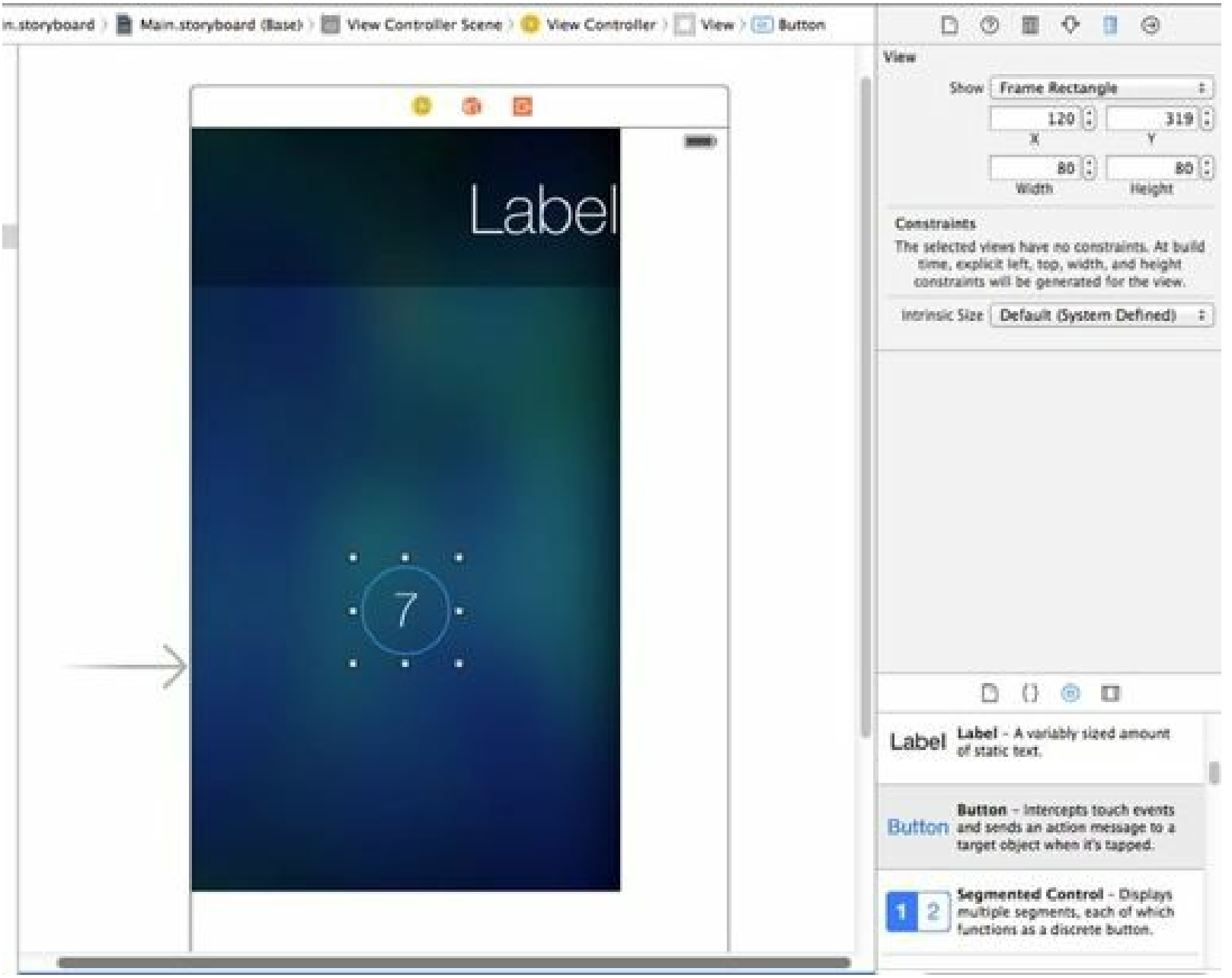


图 21-25　设置按钮大小

依次按照按钮7的设计过程，设计其他16个按钮。全部设计完成后，界面如图21-26所示。

图 21-26　设计完成

为方便程序的计算，我们为主界面中的每一个按钮添加Tag属性。具体过程是选中控件，打开右边的属性检测器，选择View→Tag属性，在这里输入该控件的Tag值，Tag属性默认为0。图21-27所示是设置“0”按钮的Tag属性。按照此方法，参考表21-4，依次设置各个按钮的Tag值。

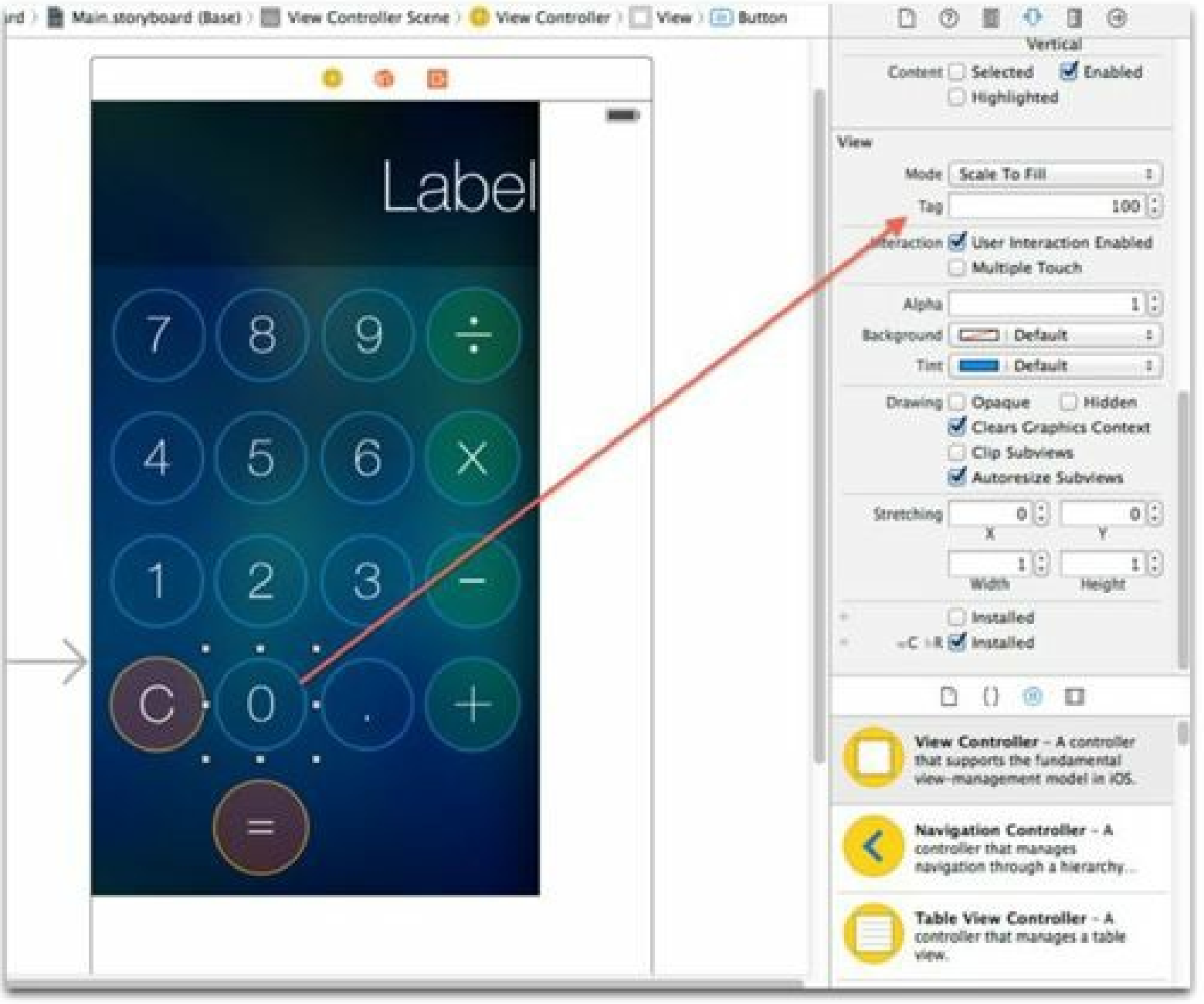


图 21-27　设置Tag属性

表21-4　按钮的Tag属性值

按钮	Tag属性值
0~9数字按钮	100~109
小数点按钮	0

清除按钮	0
等于按钮	0
加运算符按钮	200
减运算符按钮	201
乘运算符按钮	202
除运算符按钮	203

21.4.6 控件的输出口和动作

为了将访问控件状态、事件和控件联系到一起，我们引入了输出口和动作的概念。

1. **输出口**

为了能够访问标签等控件，我们需要给标签定义并连接输出口。单击左上角第一组按钮中的“打开辅助编辑器”按钮  ，打开如图21-28所示的界面。

图 21-28　辅助编辑器

选中标签，同时按住control键，将标签主标签拖曳到图21-29所示的位置。

图 21-29　拖曳标签Label

释放鼠标，会弹出一个对话框。在Connection栏中选择Outlet，将输出口命名为mainLabel，如图21-30所示。

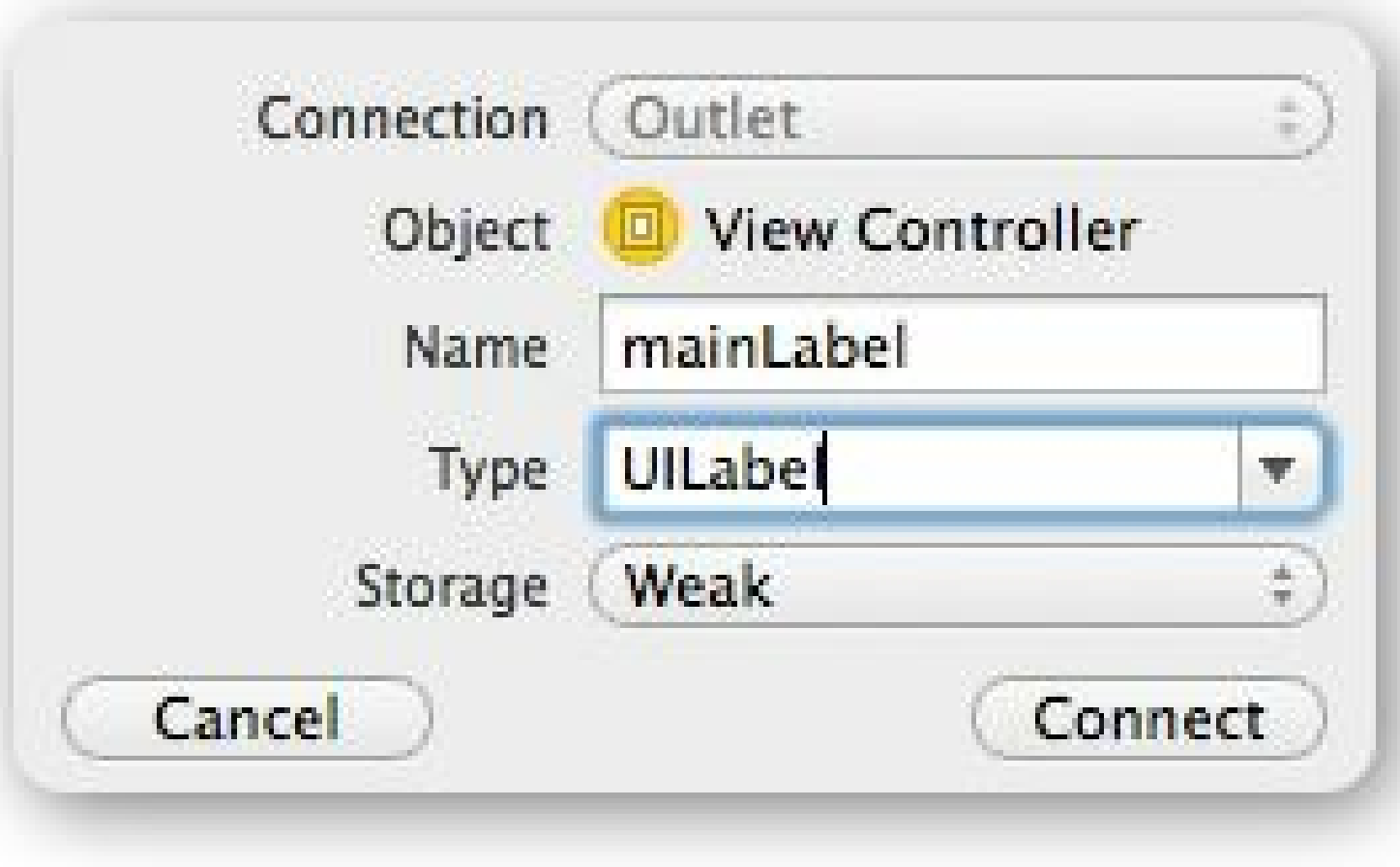


图 21-30　设置输出口

单击“Connect”按钮，右边的编辑界面将自动添加如下代码：

```
@IBOutlet var mainLabel: UILabel！
```

mainLabel被声明为@IBOutlet，这样就为主标签控件定义了输出口变量mainLabel，就可以在程序中访问mainLabel了，mainLabel就是

主标签控件的实例。

2. **动作**

为了响应按钮的事件，要把按钮事件与视图控制器中的方法管理起来。为此，需要将响应事件的方法声明为@IBAction。此处列举一个动作的代码：

```
@IBAction func numberButtonPressed(sender:
AnyObject) {
}
```

参数sender是事件源，也就是发生动作事件的控件，AnyObject是它的类型。

添加动作与输出口类似，打开辅助编辑器界面。选中按钮控件同时按住control键，将按钮控件拖曳到图21-31所示的位置。

图 21-31　拖曳按钮控件

释放鼠标，会弹出一个如图21-32所示的对话框，在Connection栏中选择Action，Name中输入“numberButtonPressed”，其他选项用默认

值就可以。

图 21-32　定义动作

单击“Connect”按钮，右边的编辑界面将自动添加如下一行代码：

```
@IBAction func numberButtonPressed(sender:
AnyObject) {

}
```

如果希望多个按钮共用一个方法，可以在拖曳第二个控件时候，直接把它拖曳到辅助编辑器的动

作方法上。如图21-33所示，选中按钮控件，同时按住control键，将按钮控件拖曳到辅助编辑器的动作方法上，然后释放鼠标键，这样就可以为多个控件的动作定义一个方法了。类似输出口可以有多个控件对应一个输出口属性，设置方法与动作类似。

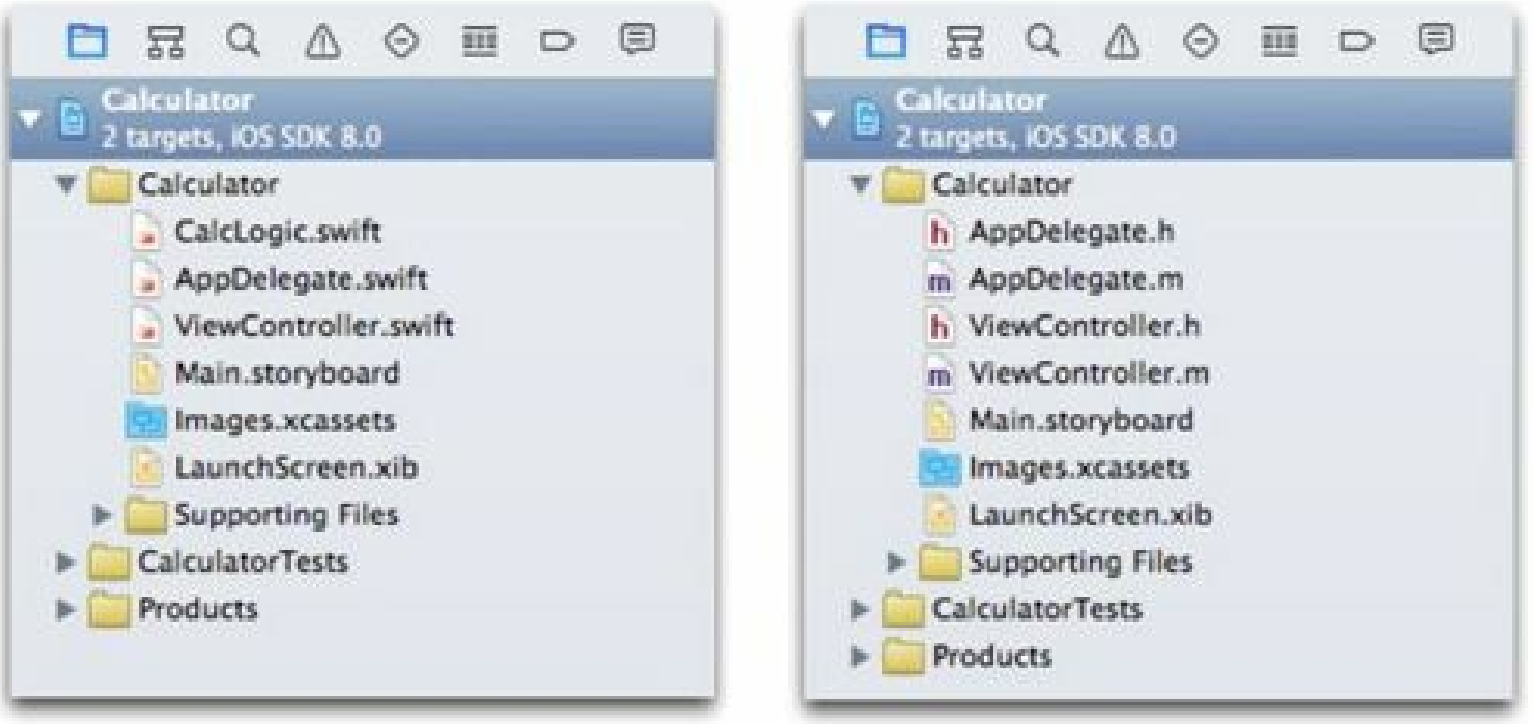


图 21-33　拖曳其他控件到动作方法

按照上面的方法依次为其他16个按钮添加动作，事件触发的方法请参考表21-5，其他的按钮设计这里不再赘述。

表21-5　按钮动作方法

按钮	触发的方法
	numberButtonPressed

0~9数字按钮	
小数点按钮	decimalPressed
清除按钮	clearPressed
等于按钮	equalsPressed
运算符按钮	operandPressed

21.4.7 视图控制器

下面我们看看视图控制器ViewController.swift的代码，代码如下：

```
class ViewController: UIViewController  {

    @IBOutlet var mainLabel: UILabel!

    var logic : CalcLogic!
①

    override func viewDidLoad() {
②
        super.viewDidLoad()
        mainLabel.text = "0"
        logic = CalcLogic()
    }
```

```
    override func didReceiveMemoryWarning() {
③
        super.didReceiveMemoryWarning()
        logic = nil
    }


    @IBAction func operandPressed(sender:
AnyObject) {
④
        var btn : UIButton = sender as UIButton
⑤
        mainLabel.text =
logic.calculateByTag(btn.tag,
withMainLabelString: mainLabel.text)    ⑥
    }

    @IBAction func equalsPressed(sender:
AnyObject) {
        var btn : UIButton = sender as UIButton
        mainLabel.text =
logic.calculateByTag(btn.tag,
withMainLabelString: mainLabel.text)
    }

    @IBAction func clearPressed(sender:
AnyObject) {
        mainLabel.text = "0"
        logic.clear()
```

```
    }

    @IBAction func decimalPressed(sender:
AnyObject) {

        if
logic.doesStringContainDecimal(mainLabel.text)
== false {                                  ⑦
            let string = mainLabel.text + "."
            mainLabel.text = string
        }
    }


    @IBAction func numberButtonPressed(sender:
AnyObject) {

        var btn : UIButton = sender as UIButton
        mainLabel.text =
logic.updateMainLabelStringByNumberTag(btn.tag,
        withMainLabelString: mainLabel.text!)
⑧

    }
}
```

上述代码第①行是声明CalcLogic可变类型变

量logic。第②行代码重写UIViewController父类的viewDidLoad()方法。该方法在视图控制器加载时调用，我们在该方法中可以进行一些初始化。类似地，第③行的didReceiveMemoryWarning()方法将在视图控制器卸载或内存报警时调用，我们在该方法中应该释放一些资源。

第④行代码operandPressed方法是在用户点击运算符按钮时调用，第⑤行代码var btn: UIButton = sender as UIButton是将参数sender转换为UIButton类型。转换为UIButton类型后，才能通过访问UIButton的属性和方法，例如tag属性。第⑥行代码是调用业务逻辑层的calculateByTag方法进行计算，返回结果用来更新mainLabel内容。

第⑦行代码在用户点击小数点时调用，在该方法中判断mainLabel标签中是否已经有小数点，如果没有则在mainLabel后面拼接一个小数点。

第⑧行代码是在用户点击了数字按钮时调用的语句，通过调用业务逻辑层的updateMainLabelStringByNumberTag方

法，将更新之后的字符串重新设置到控件mainLabel内容里。

21.5 Objective-C版本的计算器

前面的小节中我们介绍了纯Swift实现的iPhone计算器，这一节我们介绍Objective-C版本的iPhone计算器。由于本书并不是介绍Objective-C语言的书，因此本章我们假定广大读者对Objective-C语言是熟悉的。介绍Objective-C版本的主要目的是比较对于相同的一个应用，使用两种不同的语言（Objective-C和Swift）开发实现有哪些差别。Objective-C版本的iPhone计算器源代码随书一起发布给大家，这里不再详细介绍实现过程。

21.5.1 Xcode工程文件结构比较

我们使用Xcode打开两个工程，Xcode工程文件结构如图21-34所示，左图为Swift版本，右图为Objective-C版本。

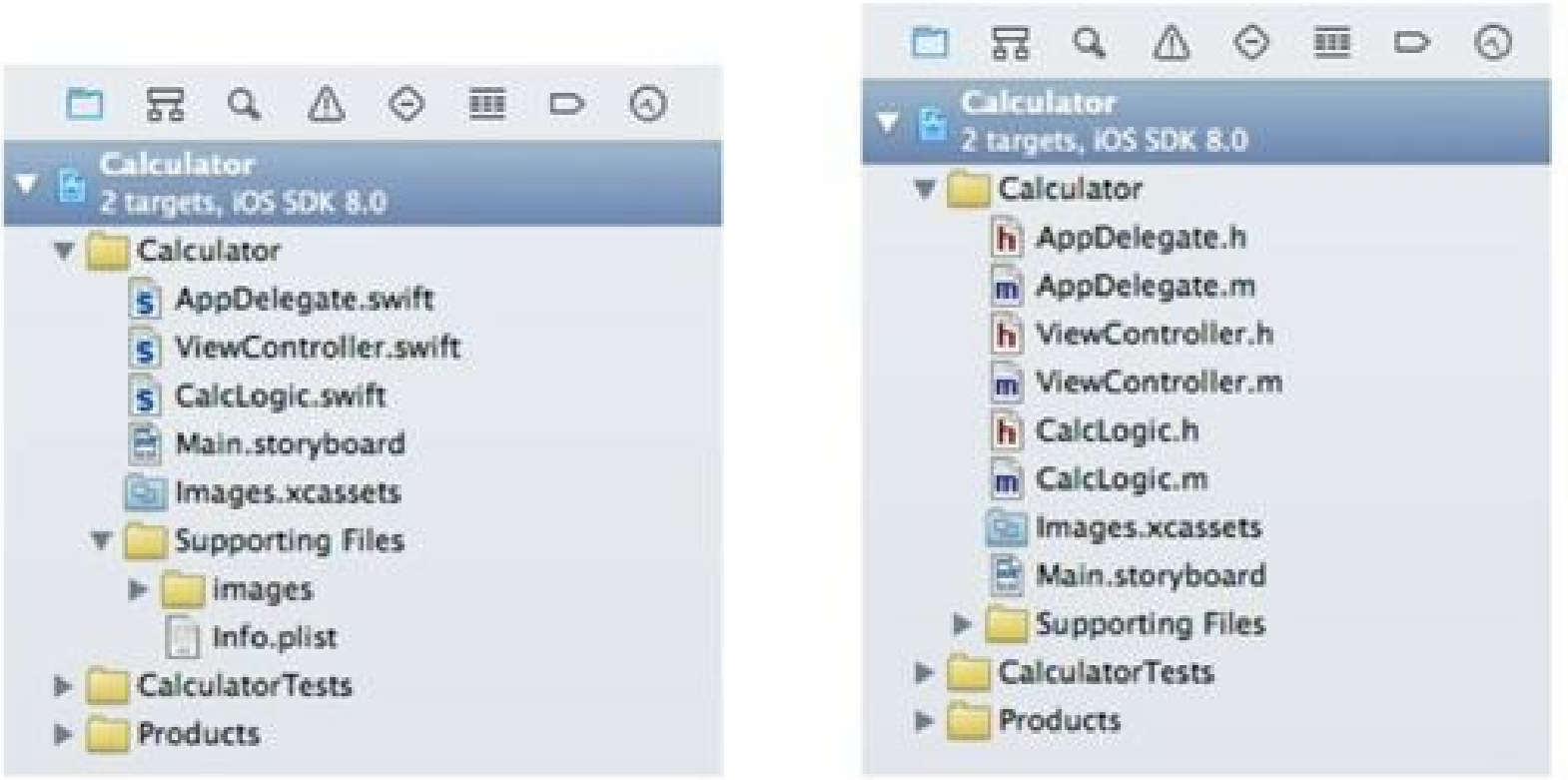


图 21-34　工程文件结构

从图中可见，Objective-C中有.h文件和.m（或.mm）文件，而Swift中只需要一个文件.swift，不需要h文件。至于其他的文件，如资源文件、配置文件和故事板文件（Main.storyboard）都没有区别，甚至在打开故事板Main.storyboard进行界面设计时也都是一样的。

21.5.2　表示层比较

表示层主要是视图控制器`ViewController`，Objective-C版本中的ViewController.h文件代码如下：

```
#import <UIKit/UIKit.h>
#import "CalcLogic.h"

@interface ViewController : UIViewController
{
    CalcLogic *logic;
①
}

@property (weak, nonatomic) IBOutlet UILabel
*mainLabel;        ②

- (IBAction)operandPressed:(id)sender;
③
- (IBAction)equalsPressed:(id)sender;
- (IBAction)clearPressed:(id)sender;
- (IBAction)decimalPressed:(id)sender;
- (IBAction)numberButtonPressed:(id)sender;

@end
```

Objective-C中的h文件用来声明类和类中成员，其中成员变量是在{}之间定义的，代码第①行定义成员变量logic。第②行代码是定义属性，Objective-C属性类似于Swift中的计算属性。第③行代码是声明成员方法，这些方法具体的属性要在

m或mm文件中实现。

Objective-C版本中ViewController.m文件代码如下：

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    self.mainLabel.text = @"0";
    logic = [[CalcLogic alloc] init];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (IBAction)operandPressed:(id)sender {
    UIButton* btn = (UIButton*)sender;
    self.mainLabel.text =  [logic
calculateByTag:btn.tag
withMainLabelString:self.mainLabel.text];
}
```

```
- (IBAction)equalsPressed:(id)sender {
    UIButton* btn = (UIButton*)sender;
    self.mainLabel.text =  [logic
calculateByTag:btn.tag
withMainLabelString:self.mainLabel.text];
}

- (IBAction)clearPressed:(id)sender {
    self.mainLabel.text = @"0";
    [logic clear];
}

- (IBAction)decimalPressed:(id)sender {
    if ( [logic
doesStringContainDecimal:self.mainLabel.text]
== FALSE) {
        NSString s =  [self.mainLabel.text
stringByAppendingString:@"."];
        self.mainLabel.text  =  s;
    }
}

- (IBAction)numberButtonPressed:(id)sender {
    UIButton btn = (UIButton*)sender;
    self.mainLabel.text = [logic
updateMainLabelStringByNumberTag:btn.tag

withMainLabelString:self.mainLabel.text];
```

```
}

@end
```

我们可以比较一下上述代码与Swift中代码的区别，这里不再赘述。

21.5.3 业务逻辑层比较

Objective-C版本中的CalcLogic.h文件代码如下：

```
#import <Foundation/Foundation.h>

typedef enum {
①
    Plus = 200, Minus, Multiply, Divide,
    Default = 0
} Operator;
②

@interface CalcLogic : NSObject
③
{
    //保存上一次的值
    double lastRetainValue;
    //最近一次选择的操作符（加、减、乘、除）
```

```
    Opertor opr;
    //临时保存MainLabel内容，为true时，输入数字
MainLabel内容被清为0
    BOOL isMainLabelTextTemporary;
}
//构造方法
-(id) init;
④
-(void)clear;

-(NSString*)updateMainLabelStringByNumberTag:
(int) tag
                          withMainLabelString:
(NSString*)mainLabelString;

-(BOOL)doesStringContainDecimal:
(NSString*)string;


-(NSString*)calculateByTag:(int) tag
                          withMainLabelString:
(NSString*)mainLabelString;


@end
```

我们在h文件中除了声明CalcLogic类之外，

还定义了枚举Operand，代码第①~②行是定义枚举Operand类型，Objective-C中的枚举Operand与Swift枚举有很大的差别。Objective-C中的枚举成员值只能是整数类型。

第③行代码是声明CalcLogic类，它继承了NSObject类，在Objective-C中所有类的根类都是NSObject类。第④行代码是声明构造函数init()，它的作用相当于Swift中的构造器。它的命名一般都是以init开头，返回值是本身类型的指针，而Swift中的构造器没有返回值。

Objective-C版本中的CalcLogic.m文件代码如下：

```
#import "CalcLogic.h"

@implementation CalcLogic

//构造方法
-(id) init
{
    NSLog(@"CalcLogic init");
    self = [super init];
①
    if (self) {
②
```

```
        lastRetainValue = 0.0;
        isMainLabelTextTemporary  = FALSE;
        opr = Default;
    }
    return self;
③
}

-(void)clear
{
    lastRetainValue = 0.0;
    isMainLabelTextTemporary  = FALSE;
    opr = Default;
}

-(NSString*)updateMainLabelStringByNumberTag:
(int) tag
                          withMainLabelString:
(NSString*)mainLabelString
{
    NSString* string = mainLabelString;

    if (isMainLabelTextTemporary)  {
        string = @"0";
        isMainLabelTextTemporary = FALSE;
    }

    int optNumber = tag - 100;
    //把String转为double
    double mainLabelDouble =
```

```
string.doubleValue;

    if (mainLabelDouble == 0
        && [self doesStringContainDecimal:
string] == false) {
        NSString strOptNumber = [NSString
stringWithFormat:@"%i", optNumber];
        return strOptNumber;
    }

    NSString resultString = [string
stringByAppendingFormat:@"%i", optNumber];

    return resultString;

}

-(BOOL)doesStringContainDecimal:
(NSString*)string
{
    NSString searchForDecimal = @".";
    NSRange range = [string
rangeOfString:searchForDecimal];

    if (range.location != NSNotFound)
        return YES;
    return NO;
}
```

```
-(NSString)calculateByTag:(int) tag
       withMainLabelString:
(NSString*)mainLabelString
{

    //把String转为为double
    double currentValue =
mainLabelString.doubleValue;

    switch (opr) {
④
    case Plus:
            lastRetainValue += currentValue;
            break;
⑤
    case Minus:
            lastRetainValue -= currentValue;
            break;
    case Multiply:
            lastRetainValue = currentValue;
            break;
    case Divide:
        if (currentValue != 0) {
            lastRetainValue = currentValue;
        } else {
            opr = Default;
            isMainLabelTextTemporary = TRUE ;
            return @"错误";
        }
            break;
```

```
    Default:
            lastRetainValue = currentValue;
    }

    /记录当前操作符，下次计算时使用
    opr = tag;
    NSString resultString = [NSString
stringWithFormat:@"%g", lastRetainValue];
    isMainLabelTextTemporary = TRUE ;

    return resultString;

}

@end
```

上述代码程序结构上与Swift版一样，但在语法上有很大区别，代码第①行`self = [super init]`语句是先构造父类实例，然后在第②行判断是否成功，如果成功再初始化成员变量。第③行返回自身对象的指针，这就是Objective-C语言的构造过程，要比Swift繁琐。

还有代码第④行使用`switch`语句。对于`switch`语句，Objective-C与Swift有很大的差别，Objective-C的`switch`语句的每一个分支都需

要加break，否则就会贯穿；而Swift默认每个分支都有break。

21.6 Swift调用Objective-C实现的计算器

除了使用纯Swift和纯Objective-C开发计算器应用之外，我们还可以使用Swift和Objective-C两种语言混合搭配来实现计算器应用。混合搭配可以有两种模式，一种是以Swift语言为主，以Objective-C语言为辅；另一种是以Objective-C语言为主，以Swift语言为辅。使用这两种模式的前提是你的应用采用了分层设计架构。

应用程序的运行过程是先启动表示层的视图，然后由表示层响应用户事件，调用业务逻辑层进行计算，计算结果再返回表示层展示出来。因此所谓的“主”就是表示层由谁来担当，所谓的“辅”就是业务逻辑层由谁来担当。

基于分层架构设计下的两种混合搭配模式分别是：

- 表示层使用Swift，业务逻辑层使用Objective-C，Swift调用Objective-C。
- 表示层使用Objective-C，业务逻辑层使用Swift，Objective-C调用Swift。

本节我们先介绍Swift调用Objective-C的情

况。

21.6.1　在Swift工程中添加Objective-C类

如果我们已经有一个使用Swift语言编写的工程，还需要调用其他Objective-C类实现某些功能，那就可以充分地利用以前使用Objective-C编写的代码。

下面我们具体在Swift版计算器工程中添加Objective-C的`CalcLogic`类。右键选择Calculator组，选择菜单中的“New File...”弹出新建文件模板对话框。如图21-35所示，选择iOS→Source→Cocoa Touch Class。

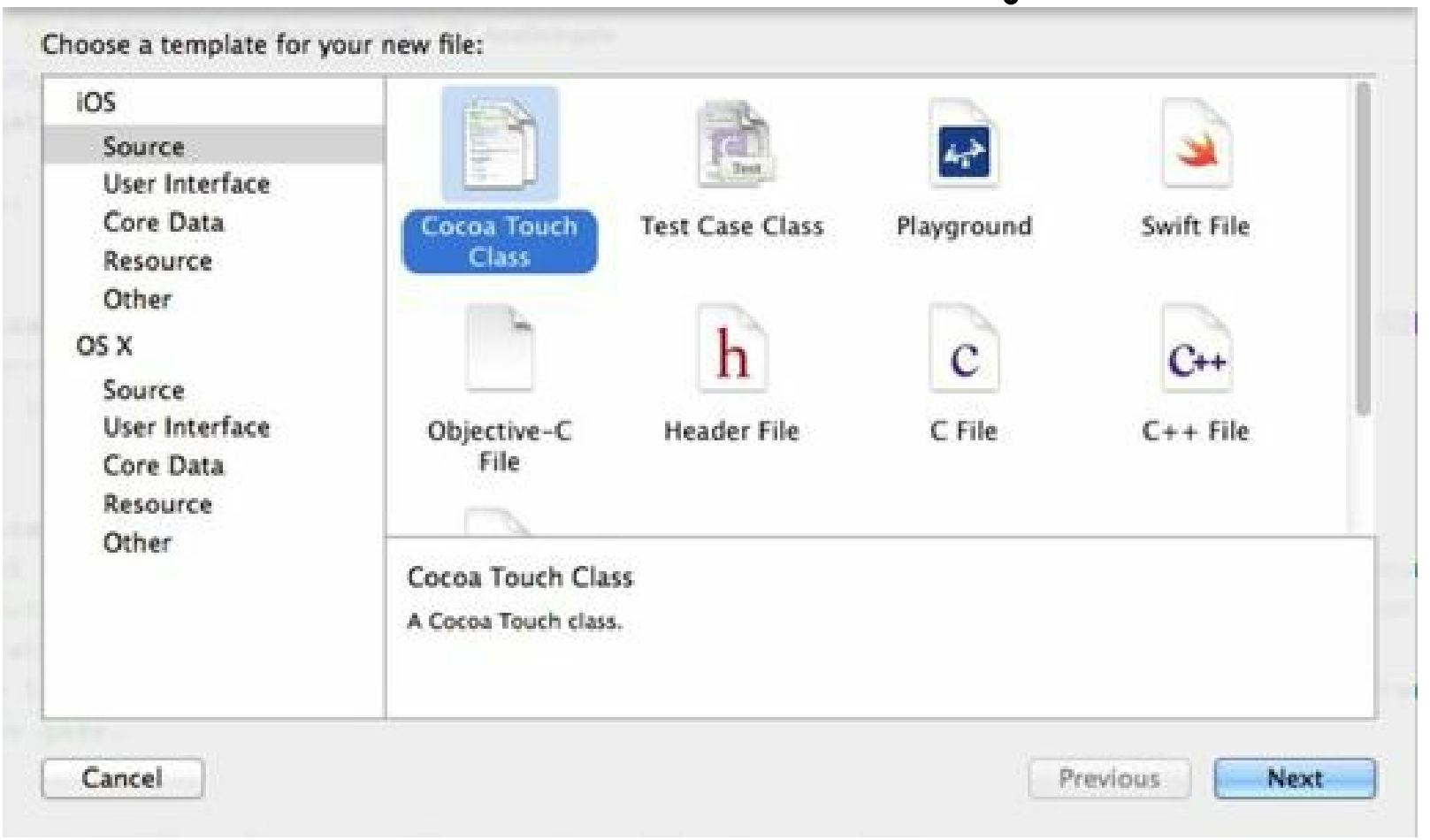


图 21-35　选择文件模板

接着单击“Next”按钮，随即出现图21-36所示的界面。在Class中输入“CalcLogic”，在Language中选择Objective-C，其他的项目保持默认值就可以了。

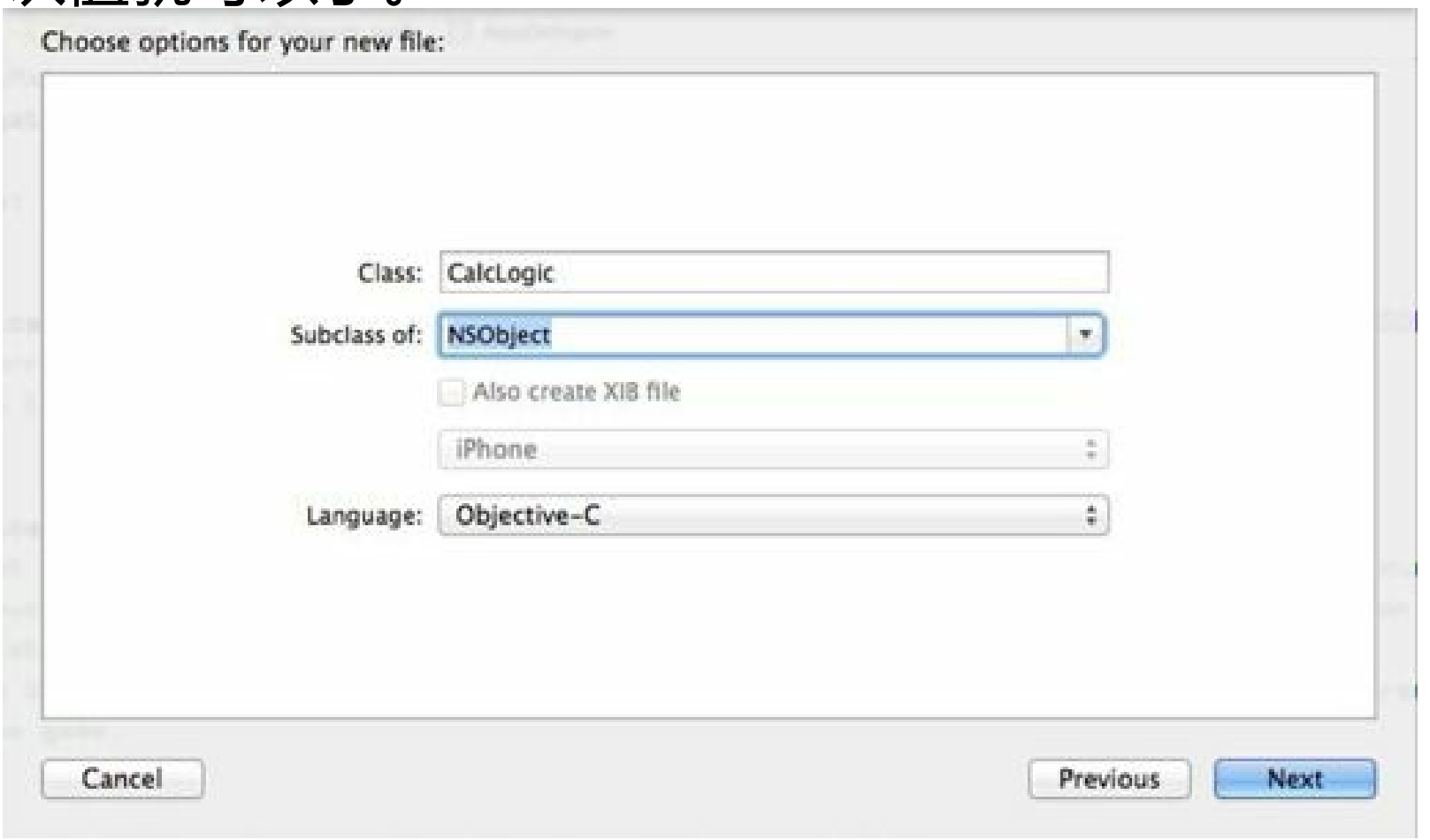


图 21-36　新建Objective-C类

设置完相关选项后，单击“Next”按钮，进入保存文件界面，根据提示选择存放文件的位置，然后单击“Create”按钮创建Objective-C类。

如果工程中没有桥接头文件，那么在添加Objective-C类的时候，Xcode会提示我们是否添加，弹出如图21-37所示的对话框，我们应该选

择“Yes”，就会在Calculator工程中创建桥接头文件Calculator-Bridging-Header.h。

图 21-37　创建桥接头文件

然后将被Swift调用的所有Objective-C类的头文件在桥接头文件中引入。Calculator-Bridging-Header.h代码如下：

```
//
//  Use this file to import your target's
public headers that you would like to expose to
Swift.
//
```

```
#import "CalcLogic.h"
```

在本例中，只引入了CalcLogic.h一个头文件。

21.6.2 调用代码

事实上，如果Swift和Objective-C两端的方法命名很规范，我们几乎不需要修改什么就可以调用了。这里主要需要修改Swift端的ViewController.swift代码：

```
import UIKit

class ViewController: UIViewController  {

    @IBOutlet var mainLabel: UILabel!

    var logic : CalcLogic!

    override func viewDidLoad() {
        super.viewDidLoad()
mainLabel.text = "0"
        logic = CalcLogic()
    }

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

```
        logic = nil
    }


    @IBAction func operandPressed(sender:
AnyObject) {
        var btn : UIButton = sender as UIButton
        mainLabel.text =
logic.calculateByTag(CInt(btn.tag),
        withMainLabelString: mainLabel.text)
①
    }

    @IBAction func equalsPressed(sender:
AnyObject) {
        var btn : UIButton = sender as UIButton
        mainLabel.text =
logic.calculateByTag(CInt(btn.tag),
        withMainLabelString: mainLabel.text)
②
    }

    @IBAction func clearPressed(sender:
AnyObject) {
        mainLabel.text = "0";
        logic.clear()
    }

    @IBAction func decimalPressed(sender:
AnyObject) {
```

```
        if
logic.doesStringContainDecimal(mainLabel.text)
== false {                    ③
            let string = mainLabel.text + "."
            mainLabel.text = string
        }
    }


    @IBAction func numberButtonPressed(sender:
AnyObject) {

        var btn : UIButton = sender as UIButton
        mainLabel.text =
logic.updateMainLabelStringByNumberTag(CInt(btn.

        withMainLabelString: mainLabel.text)
④

    }
}
```

上述代码第①~④行调用了Objective-C类CalcLogic，其中第①行代码是调用CalcLogic的如下方法：

```
-(NSString*)calculateByTag:(int)
tagwithMainLabelString:
(NSString*)mainLabelString
```

要注意Objective-C中方法的命名，以及在Swift中调用的方法名和参数要对应，它们的对应关系如图21-38所示。

图 21-38　方法命名对应关系

Objective-C类`CalcLogic`中方法命名是采用多重参数的，方法名的第一部分名`calculateByTag`就是在Swift调用时的方法名，`CalcLogic`中其他部分名，如`withMainLabelString`是在Swift调用时的外部参数名。

代码编写完成后，我们可以运行一下看看效果。

21.7　Objective-C调用Swift实现的计算器

在上一节我们实现了Swift调用Objective-C，现在我们再来介绍一下Objective-C调用Swift实现。

21.7.1　在Objective-C工程中添加Swift类

如果我们已经有一个使用Objective-C语言编写的工程，还需要调用其他Swift类实现某些功能，那就可以充分利用Swift编写的代码。

下面我们具体在Objective-C版计算器工程中添加Swift的`CalcLogic`类。右键选择Calculator组，选择菜单中的“New File...”弹出新建文件模板对话框。如图21-39所示，选择iOS→Source→Cocoa Touch Class。

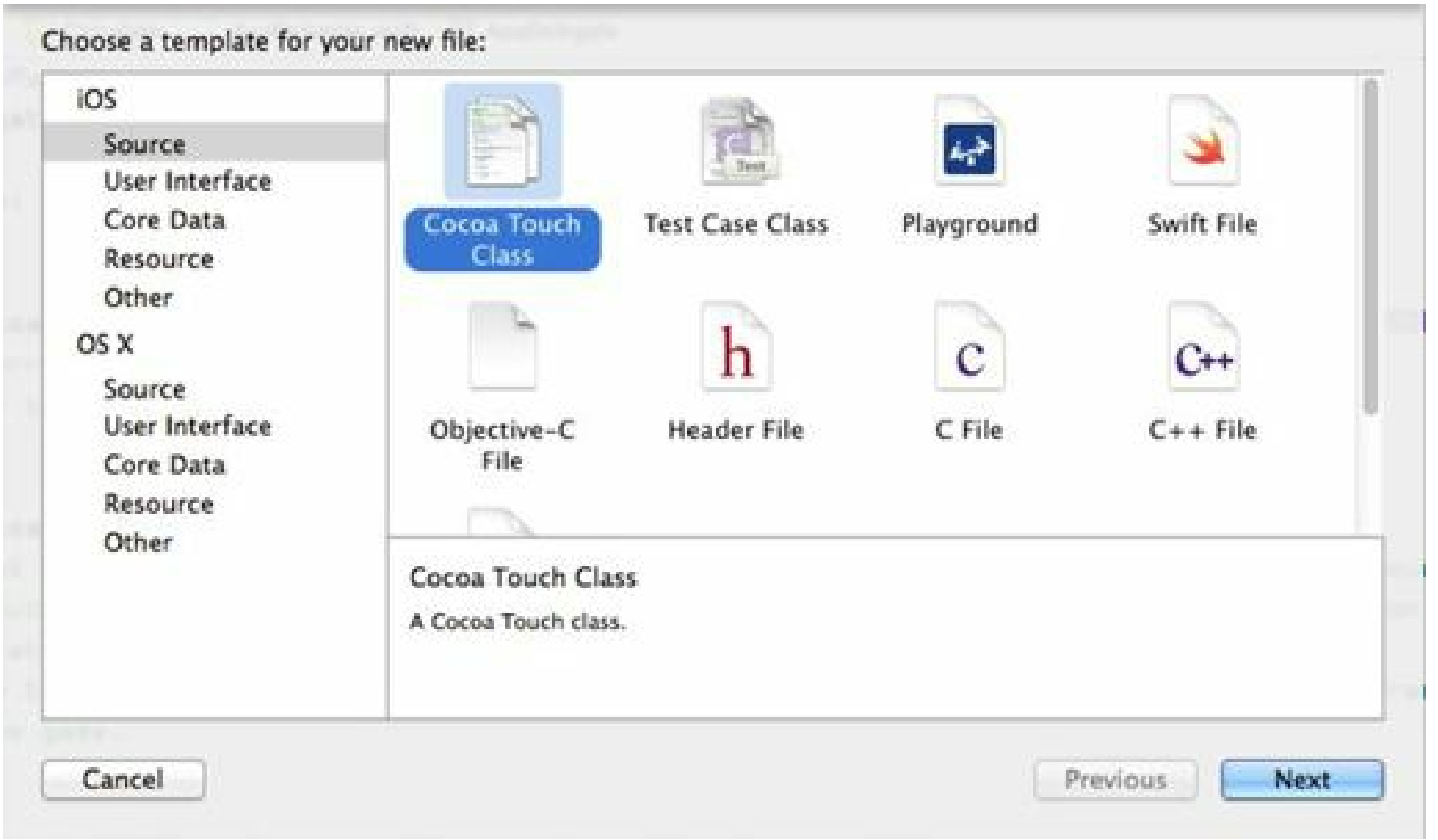


图 21-39　选择文件模板

接着单击“Next”按钮，随即出现如图21-40所示的界面。在Class中输入“CalcLogic”，Subclass of（继承自）项目选择NSObject，在Language中选择Swift，其他的项目保持默认值就可以了。

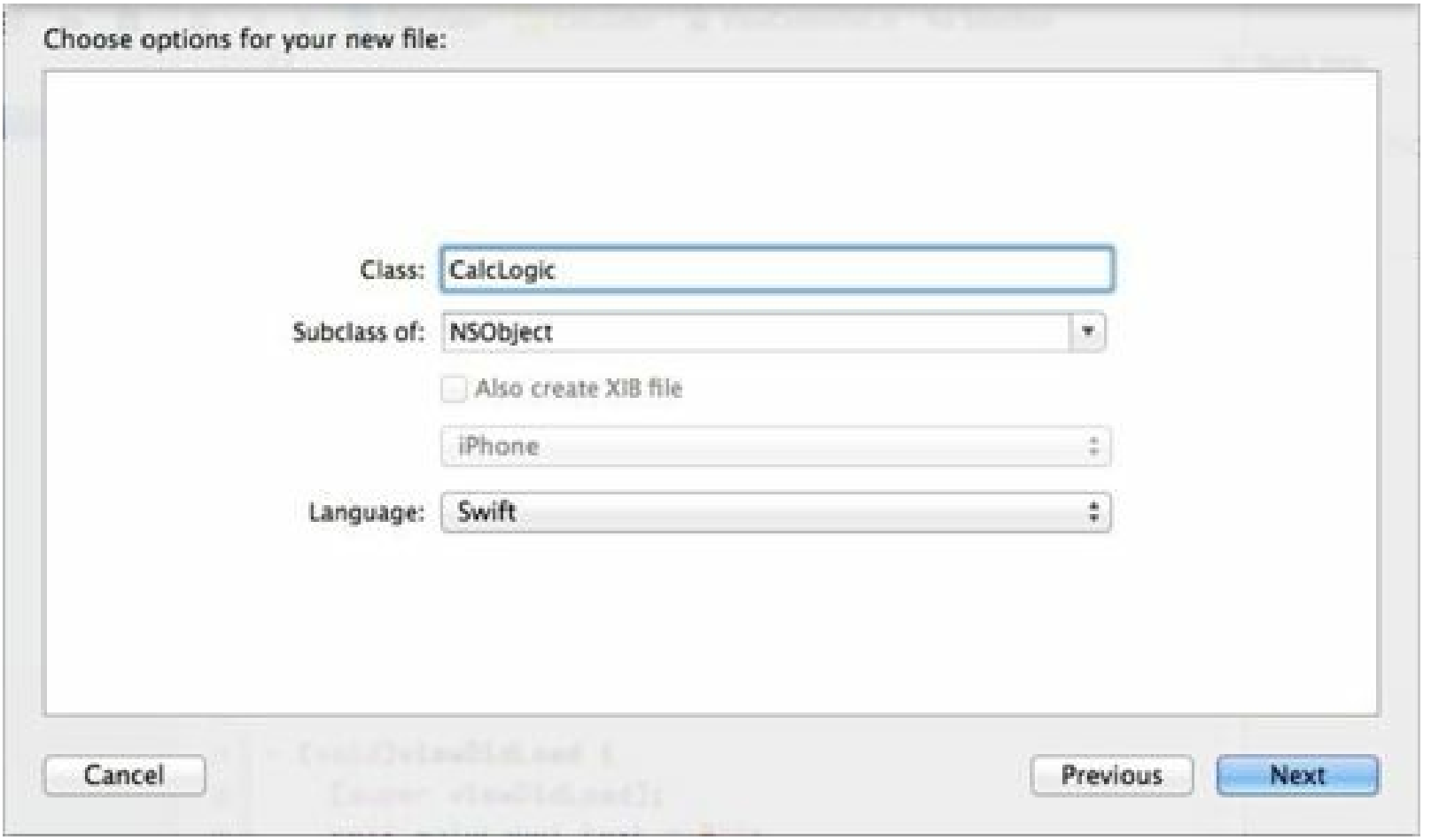


图 21-40　新建Swift类

设置完相关选项后，单击“Next”按钮，进入保存文件界面，根据提示选择存放文件的位置，然后单击“Create”按钮创建Swift类。

21.7.2　调用代码

从Objective-C调用Swift不需要桥接头文件。在Objective-C端，视图控制器`ViewController`的代码基本上不需要修改，但是要想访问Swift对象还是需要引入一个头文件，这个文件的命名是 <工程名>-Swift.h，那么就需要在ViewController.h文件中引入头文件，ViewController.h代码如下：

```
#import <UIKit/UIKit.h>
#import "Calculator-Swift.h"
①

@interface ViewController : UIViewController
{
    CalcLogic logic;
}

@property (weak, nonatomic) IBOutlet UILabel
mainLabel;

- (IBAction)operandPressed:(id)sender;
- (IBAction)equalsPressed:(id)sender;
- (IBAction)clearPressed:(id)sender;
- (IBAction)decimalPressed:(id)sender;
- (IBAction)numberButtonPressed:(id)sender;

@end
```

上述代码第①行引入了头文件Calculator-Swift.h，这样Objective-C就可以访问工程中的Swift类了。

此外，还需要修改一下Swift端的代码，Swift端的CalcLogic.swift代码如下：

```
import Foundation

enum Operator : Int {
    case Plus = 200, Minus, Multiply, Divide
    case Default = 0
}

@objc class CalcLogic : NSObject {
①

    //保存上一次的值
    var lastRetainValue : Double
    //最近一次选择的操作符（加、减、乘、除）
    var opr : Operator
    //临时保存MainLabel内容，为true时，输入数字MainLabel内容被清为0
    var isMainLabelTextTemporary : Bool

    /*
    * 构造器

    override init () {
②
        println("CalcLogic init")
        lastRetainValue = 0.0
        isMainLabelTextTemporary  = false
        opr = .Default
    }
```

```
    * 析构器

    deinit {
        println("CalcLogic deinit")
    }


     判断字符串中是否包含.
    /
    func updateMainLabelStringByNumberTag(tag :
Int, withMainLabelString
    mainLabelString : String)->String {

        var string = mainLabelString

        if (isMainLabelTextTemporary)  {
            string = “0”
            isMainLabelTextTemporary = false
        }

        let optNumber = tag - 100
        //把String转为double
        var mainLabelDouble = (string as
NSString).doubleValue

        if mainLabelDouble == 0 &&
doesStringContainDecimal(string) == false {
            return String(optNumber)
        }
```

```
        let resultString = string +
String(optNumber)

        return resultString
    }


    /*
    * 判断字符串中是否包含小数点

    func doesStringContainDecimal(string :
String)->Bool {
        for ch in string {
            if ch == "." {
                return true
            }
        }
        return false
    }


     点击操作符时的计算
    /
    func calculateByTag(tag : Int,
withMainLabelString mainLabelString : String)-
>String {

        //把String转为double
        var currentValue = (mainLabelString as
NSString).doubleValue
```

```
        switch Operator {
        case .Plus:
            lastRetainValue += currentValue
        case .Minus:
            lastRetainValue -= currentValue
        case .Multiply:
            lastRetainValue *= currentValue
        case .Divide:
            if currentValue != 0 {
                lastRetainValue = currentValue
            } else {
                opr = .Default
                isMainLabelTextTemporary = true
                return "错误"
            }
        Default:
            lastRetainValue = currentValue
        }

        /记录当前操作符，下次计算时使用
        opr = Operator(rawValue:tag)!

        let resultString = NSString(format:
"%g", lastRetainValue)

        isMainLabelTextTemporary = true

        return resultString
    }
```

```
    func clear() {
        lastRetainValue = 0.0
        isMainLabelTextTemporary  = false
        opr = .Default
    }
}
```

上述代码与纯Swift版本变化不是很大。但是需要注意代码第①行，CalcLogic类继承了NSObject类，而纯Swift版本中的CalcLogic类是没有继承任何父类的。再有我们在类前面声明为@objc，这也是与纯Swift版本的区别，@objc所声明的类能够被Objective-C访问，@objc还可以修饰属性。代码第②行override init()是重写NSObject父类构造器init()。

代码编写完成后，我们可以运行一下看看效果。

21.8　本章小结

通过对本章内容的学习，我们了解了iOS应用开发的一般流程，掌握了Objective-C语言与Swift语言混合搭配和调用，了解了分层架构设计的重要性。

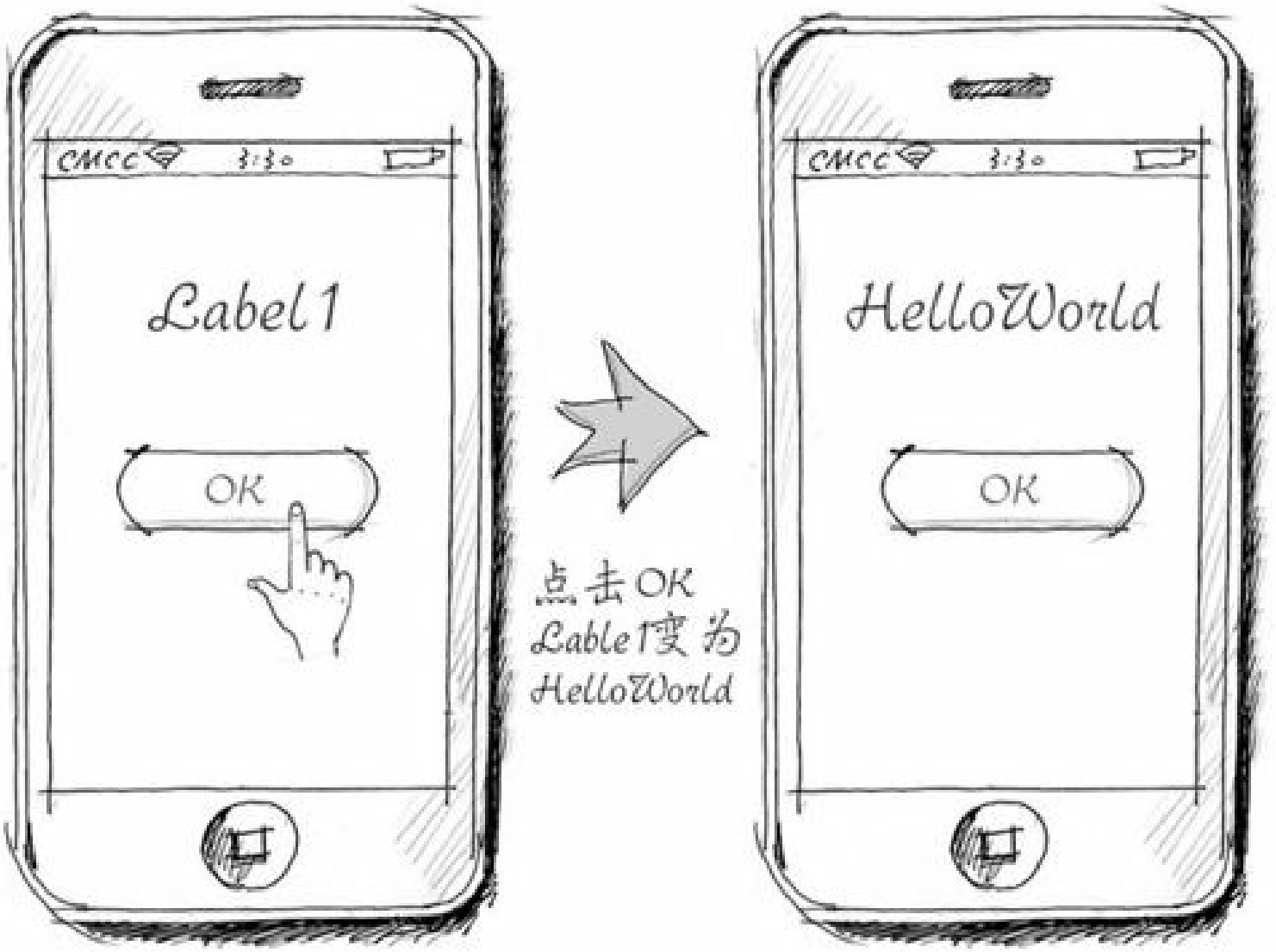
21.9　同步练习

1. 编程题：将本章的iPhone计算器应用，自己重新编写，熟悉每一行代码。

2. 编程题：将第18章的同步练习第4题，重新编写成iPhone版本。

3. 编程题：将第18章的同步练习第5题，重新编写成iPhone版本。

4. 编程题：请参考下面的界面原型草图，编写一个iPhone程序。


CMCC 3:30
Label1
OK
点击OK
Lable1变为
HelloWorld
CMCC 3:30
HelloWorld
OK

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks
