Swift 5

从零到精通iOS开发训练营

• 张益珲 编著 •

清華大學出版社
北京

内 容 简 介

本书由专业的 iOS 开发工程师根据新发布的 Swift 5.5 编程语言精心编撰，书中兼备核心语法、编程技巧与应用实践 3 大主题。本书第一部分从 Xcode 开发工具及 Swift 学习环境的搭建开始，重点介绍 Swift 的语言特性和应用场景，提供了大量编程练习，帮助读者尽快掌握 Swift 语言的精髓。第二部分介绍 Swift 开发 iOS 应用的基本技能，包括独立 UI 控件的应用、视图界面逻辑的开发、动画与布局技术、网络与数据处理技术以及新的 SwiftUI 编程技术等，旨在带领读者独立开发一款 iOS 应用程序。第三部分为应用部分，这部分安排了实战项目（简易计算器、生活记事本、中国象棋游戏），项目的安排由简到难，旨在全面锻炼读者的实际开发能力，使用 Swift 进行开发实践。本书还在每一章中插入了模拟面试题，以帮助读者应对 iOS 开发职位的面试。

通过本书的学习，读者可以轻松地掌握使用 Swift 语言开发一款 iOS 软件从理论到实践的全部技术细节。本书适合使用 Swift 开发 iOS 应用的新手，以及有 Objective-C 基础，想学习 Swift 的 iOS 开发人员学习，也适合作为培训机构与大中专院校移动开发课程的教学用书或面试指导书。

图书在版编目（CIP）数据

Swift 5 从零到精通 iOS 开发训练营 / 张益珲编著.—北京：清华大学出版社，2021.8
ISBN 978-7-302-58864-1

Ⅰ. ①S… Ⅱ. ①张… Ⅲ. ①程序语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2021）第 159581 号

责任编辑：王金柱
封面设计：王 翔
责任校对：闫秀华
责任印制：丛怀宇

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 邮 编：100084
社 总 机：010-62770175 邮 购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市天利华印刷装订有限公司
经 销：全国新华书店
开 本：190mm×260mm **印 张：**29.75 **字 数：**762 千字
版 次：2021 年 10 月第 1 版 **印 次：**2021 年 10 月第 1 次印刷
定 价：119.00 元

产品编号：089357-01

前　言

自2014年Apple在全球开发者大会上发布了Swift编程语言，至今已经经历了5个大版本的迭代，随着Swift语言的更新与完善，开发者对其的热情也越来越高，越来越多的公司在开发iOS软件项目时都将Swift作为最先选择的编程语言。

从第一版Swift语言的发布开始，我就一直对这门新兴的编程语言有着浓厚的兴趣，几年前，当我第一次收到清华大学出版社王金柱编辑的邀请，建议写一本帮助新手入门Swift语言iOS开发的工具书时，忐忑的心情至今还记忆犹新。经过再三的考虑与矛盾，我最终决定接下这个任务。当然，不是我对自己的编程技能充满信心，只是觉得把学习过程中遇到的问题、走过的弯路、积累的经验整理成册提供给初学者是一件非常有意义的事情。后来经过半年多的努力，《Swift 3从入门到精通》顺利出版并且得到了不错的回应。尽管在写作的过程中充满了艰辛，但是看到自己的作品可以给读者带来切实的帮助，我也收获到了额外的喜悦与慰藉。后来，在各位读者的帮助下，我对书中出现的错误与不合理之处进行了多次纠正与优化，《Swift 4从入门到精通》得以与读者相见。

截至本书完稿，Swift编程语言已经更新到5.5版本，其间Swift语言改变了很多，也优化了很多，从1.0到3.0版本，Swfit语言经历了质的变化，从3.0到5.0版本，Swift语言也迎来了完善与稳定。Swift语言是少有的在短时间内大版本更迭的编程语言，这也体现了这门语言不拘一格、大胆创新的特点。本书在《Swift 4从入门到精通》的基础上，优化了部分过时的内容，新增了Swift 5.5的新特性与SwiftUI技术的内容，更重要的是，总结了《Swift 4从入门到精通》一书读者的反馈，本书中的内容更加面向应用，并且插入了大量的面试题，并做了试题解析与面试指导。

本书分为三大部分。

第一部（第1~12章）将为读者介绍Swift语言的一些基础语法点，包括数据类型、流程控制语句、运算符、函数与闭包、枚举、结构体、类、属性与方法、对象构造与析构、内存管理、异常处理、扩展与协议以及Swift语言的新特性等。这12章内容将竭力为读者介绍Swift语言的语法特点与应用场景，并且每一章后面都附带有习题，供读者对本章所学的知识进行测试与应用。

第二部分（第13~18章）为iOS开发基础部分，目前Swift语言应用的主要场景为iOS应用的开发。这部分内容将系统地向读者介绍iOS的开发技能，包括独立UI控件的应用、视图界面逻辑的开发、动画与布局技术、网络与数据处理技术等。掌握了这些技能，理论上读者已经具备独立开发一款iOS应用程序的能力。与《Swift 4从入门到精通》一书不同的是，本书中新增了SwiftUI章节，SwiftUI技术将使得软件的界面开发更加容易。

第三部分（第19~21章）为实战部分，学习编程，实战是必经的一关。本书为读者安排了3个实战项目，项目的安排由简入难，并且各个项目的侧重点分布均匀，力图全面锻炼读者的实际开发能力。

除了三大部分循序渐进的技能学习外，在每一章的最后都加入了练习题与模拟面试。练习题可以帮助读者更好地理解和掌握当前章节所学习的内容，模拟面试可以帮助读者增加实战经验，进

而提高应用能力。

本书是一本从基础到实战的Swift编程语言学习教程。如果你符合下面的特点，那么本书就是为你定制的：

（1）对iOS系统软件开发感兴趣，想要从事iOS软件开发的人员。
（2）对编程感兴趣，对Swift编程语言感兴趣的人员。
（3）熟悉Objective-C语言，想要尝试Swift语言的开发者。
（4）需要进行面试指导的Swift求职者。

此外，本书还提供了全部源代码，以方便读者上机演练，读者扫描以下二维码即可下载：

如果你在下载过程中遇到问题，可发送邮件至booksaga@126.com获得帮助，邮件标题为“Swift 5从零到精通iOS开发训练营”。

编程是一门动手性很强的技能，因此在学习本书时，读者首先需要搭建好自己的开发环境（本书第1章有介绍）。在学习书中内容时要对照代码进行实际操作，并且本书的配套资源中也有书中所引用的全部代码，读者在学习时可以进行参考对照。如果读者没有良好的Swift语言基础，在学习本书时，请务必根据章节的顺序安排进行学习，只有有了良好的语言基础，学习后面章节的时候才能得心应手。

本书能够顺利完成，首先要感谢家人对我写作的支持，感谢朋友们的无私帮助。最重要的是感谢清华大学出版社的王金柱编辑，王金柱编辑耐心地纠正了我许多写作中的问题，并且给了我许多非常有价值的建议，指导我完成了本书的编写。没有他的辛勤付出，本书不会出现在读者的面前。最后，感谢所有读者，我们都是编程路途中的学习者，你们的努力和认可让我坚定不移地去做分享知识这件有意义的事，希望我们能够一起努力，一起前进！

编　者

2021年6月27日

目　　录

第一部分　Swift 语言基础语法

第二部分 iOS 开发基础

第三部分 实战

第一部分　Swift 语言基础语法

本书的第一部分将向读者介绍Swift编程语言的基础语法。Swift是一门十分年轻的编程语言，其由苹果公司在2014年的WWDC（苹果开发者大会）上发布。虽然和其他主流语言相比，Swift有些年轻与稚嫩，但其设计思路更加现代化，并且在苹果公司的推动下，其获得了突飞猛进的发展。截至2020年09月，Swift语言发布到了5.3版本。

在Swift语言的发展过程中，Swift 3可谓是一个突破性的版本，其除了移除了一些旧的特性，增加了一些新的特性外，还对许多API接口的命名和结构进行了调整，使其更加契合Swift语言本身。如果读者想要学习Swift语言，又担心其更新变动过大导致学习成本的浪费，现在基本可以放下这个疑虑了。2017年9月，Swift语言版本更新到了4.0，和3.x版本相比，Swift 4.0增强了对内存访问安全的控制，增强了泛型的功能。Swift 4.2又在4.0版本的基础上进行了补充与优化。2019年，Swift 5版本发布，对字符串、函数、枚举、闭包等都做了语法增强，相信Swift语言的生态将会越来越丰富，其与传统的iOS程序开发语言Objective-C相比优势也将越来越大。

和Objective-C语言冗长的函数名相比，Swift语言显得十分简洁，而在功能上，Swift也丝毫不逊色于Objective-C，比较显著的一些特点是Swift语言支持元组类型，支持开发者定义运算符函数，支持简洁的流程控制语句以及强大的闭包技术。这些方面的优势都可以帮助开发者在代码编写中事半功倍。Objective-C语言的设计思路是传统的面向对象语言模式的，而Swift语言的设计思路是面向协议的函数式编程思想，并且Swift语言可以很完美地支持macOS与iOS系统软件的开发，本书第3部分就将以iOS应用软件实战为例介绍Swift语言在实战开发中的应用。

第1章

学习环境的搭建

工欲善其事，必先利其器。

——孔子

做任何事情之前都要将需要使用的工具准备妥当，木匠需要一把好锯，瓦匠需要一把好铲。对于软件开发者，一款强大易用的开发工具是工作中的必备利器。学习编程，首先要学习相应开发环境的搭建和开发工具的使用，并且编程必须在练习中掌握技能，在正式学习之前，安装好开发工具与熟悉开发环境是第一步。本章将向读者介绍Xcode开发集成工具的下载安装及简单使用。

通过本章，你将学习到：

- 申请个人的Apple ID账号。
- 在App Store上下载Xcode开发工具。
- 熟悉Xcode开发工具界面与使用。
- 使用Playground工具进行Swift代码演示与练习。
- 编写第一个Swift程序Hello World。

1.1 申请个人 AppleID 账号

苹果公司在2014年开发者大会上发布了新的编程语言——Swift，同时，苹果公司自家的开发工具Xcode也集成了支持Objective-C与Swift两种编程语言的开发环境。由于Swift语言开源的特性，未来支持Swift语言的开发工具会越来越多，Swift语言的应用场景也会越来越广泛。毋庸置疑的是，目前Xcode依然是最好用的Swift开发工具，本书也将使用Xcode开发工具来进行语言讲解与演示。

Xcode开发工具可以在App Store上免费下载。首先，读者需要有一个个人的Apple ID账号，如果没有，可以在官方网站进行申请（https://appleid.apple.com/account），页面如图1-1所示。

图 1-1　创建一个 Apple ID 账号

在上面的注册网站中，需要使用一个电子邮箱地址作为Apple ID账号，这里读者需要注意，提供的电子邮箱地址务必要准确，Apple ID的激活需要邮箱认证。注册过程中需要填写的密码保护问题也务必认真填写并妥善保存，如果不小心忘记了密码，密码保护问题将成为读者找回密码的一个重要途径。

1.2　下载与安装 Xcode 开发工具

App Store是Apple自家的应用市场软件，其集成了应用程序下载与安装一体化的功能，读者可以十分方便地使用它安装最新版的Xcode开发工具。打开App Store软件，在主界面的搜索栏中填入Xcode，之后按Enter键进行搜索，如图1-2所示。

图 1-2　App Store 主页

搜索结果页中的第一个软件就是Xcode开发工具，单击获取即可进行Xcode工具的下载与安装，如图1-3所示。

图 1-3　获取安装 Xcode 开发工具

需要注意在获取Xcode开发工具时，App Store软件会要求验证开发者账号，读者只需将1.1节中申请到的Apple ID账号和密码正确填入即可。

App Store上获取到的软件默认为最新的正式版软件，截至2020年10月，Xcode最新版本为Xcode 12.3，其包含Swift 5.3相关开发包。如果读者需要旧版本的Xcode开发工具或者需要某些Beta版的Xcode开发工具，可以到苹果开发者中心的工具下载页面进行其他版本的下载（https://developer.apple.com/downloads/）。同样，要进入开发者中心，也需要读者使用Apple ID登录。

提　示
App Store 的服务器有时不太稳定，因此访问起来有时会很慢，读者也可以在https://developer.apple.com/downloads/网站上下载最新的 Xcode 开发工具免安装版，下载完成后直接解压即可使用。

1.3　Xcode 开发工具简介

Xcode开发工具的功能十分强大，可以进行macOS、iOS、tvOS、watchOS平台软件的开发，并且支持使用Objective-C与Swift两种语言环境，同时兼容C、C++语言环境。在下载安装Xcode工具后，其也会打包下载对应模拟器，以iOS开发为例，开发者可以十分方便地使用各种版本的iPhone和iPad模拟器来进行程序调试。

1. Xcode 开发工具的欢迎界面

打开Xcode开发工具，首先会出现软件的欢迎界面，如图1-4所示。

图 1-4　Xcode 开发工具的欢迎界面

各选项的含义说明如下：

- Create a new Xcode project：用于创建一个新的Xcode独立工程，是开发中新建工程常用的一个选项。
- Clone an exiting project：用于从仓库中拉取一个已经存在的项目。本书在语法讲解阶段，大部分会采用Playground来进行代码的演示，在iOS程序开发学习与项目实战阶段，会使用创建工程的方式来进行演示。页面中的Version标注了当前Xcode开发工具的版本，Xcode 12及以上版本都对Swift 5.3语言进行了支持。本书使用Xcode 12.0版本。
- Open a project or file：用来打开一个已经存在的工程或文件。
- Show this window when Xcode launches：用于设置每次启动Xcode开发工具时是否都展示这个欢迎界面。

2．创建一个空的 Xcode 工程

我们先来创建一个空的Xcode工程，用来介绍Xcode编码主界面的构成。

单击Create a new Xcode project选项来创建一个新的Xcode工程，之后会弹出选择工程类型模板的窗口，如图1-5所示。

窗口导航栏为工程运行的平台，iOS应用于iPhone手机与iPad平板电脑软件的开发，watchOS应用于苹果手表软件的开发，tvOS应用于苹果电视软件的开发，OS X应用于Mac计算机软件的开发。这里我们选择OS X平台下的命令行模式，即Command Line Tool，单击Next按钮后，会弹出工程配置窗口，如图1-6所示。

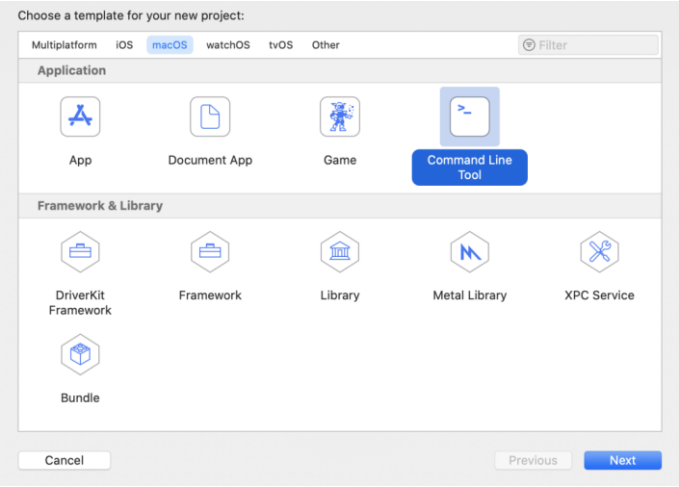

图 1-5　选择工程类型模板

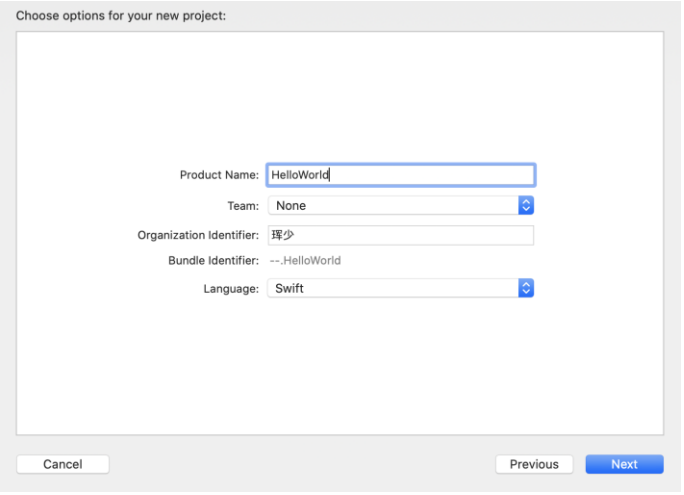

图 1-6　工程配置窗口

图1-6所示的工程配置窗口中各选项的说明如下：

- Product Name：用于填写工程的名称。
- Organization Name：用于填写开发机构组织的名称，一般是软件开发公司的名称。
- Organization Identifier：用于填写机构组织的ID编号。
- Bundle Identifier：工程项目的唯一标识名，Xcode会自动根据组织和工程名称生成，开发者也可以根据需求来自定义这个标识名。Bundle Identifier十分重要，在上线应用生成证书、应用推送功能开发、应用组App Group功能开发时都需要与Bundle Identifier进行关联。
- Language：用于选择开发语言，Xcode工具支持Objective-C、C、C++和Swift这4种语言，iOS开发框架只支持创建Objective-C和Swift这两种语言的工程。这里选择Swift。

单击Next按钮进行工程的创建。之后还会弹出一个工程创建路径设置的窗口，选择工程要存放的路径后，单击Create按钮即可完成工程的创建。

3. Xcode 开发工具的主界面

Xcode开发工具的主界面如图1-7所示。

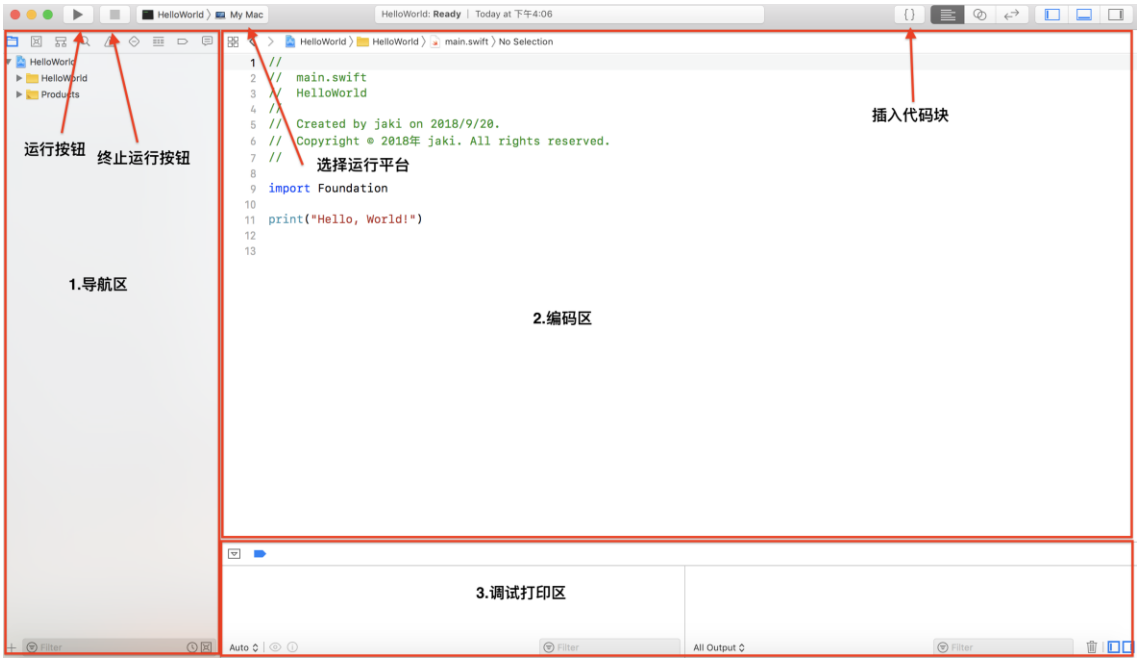

图 1-7　Xcode 开发工具的主界面

Xcode的主界面主要分为3部分，左侧是导航区，其主要作用是展示一些文件与内容的索引，比如文件目录索引、堆栈信息索引、断点信息索引、警告信息索引、搜索信息索引等，通过切换导航区上方的一排按钮可以进行导航内容的切换。右侧上部分为编码区，开发者可在其中进行代码的编写。右侧下部分为调试打印区，开发者可以在其中看到断点处的变量信息以及调试打印信息。Xcode开发工具主界面的左上角有两个功能按钮，其作用是运行工程与停止运行工程，其后边的下拉菜单供开发者根据需要选择不同的运行设备。右边的插入代码块按钮支持开发者进行代码块的自定义，方便快速输入。

当创建完Hello World工程模板后，读者就已经完成了一个简单的入门程序，打印“Hello,World”字符串，单击“运行”按钮运行工程，可以看到调试区中出现的打印信息，如图1-8所示。

```
Hello, World!
Program ended with exit code: 0
```

图 1-8　Xcode 的打印信息

1.4　使用 Playground 进行 Swift 代码演练

Playground文件与工程相比简约许多，其主要使用场景是Swift语法代码的学习与演示。在Xcode的菜单栏中，选择File→New→Playground选项可以创建一个Playground文件，如图1-9所示。

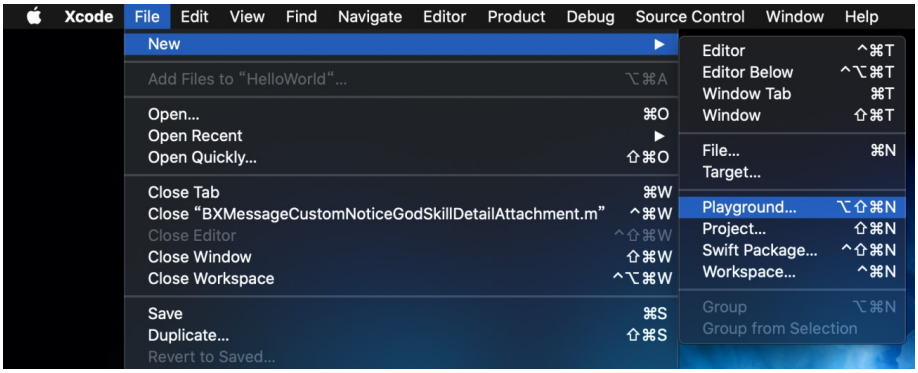

图 1-9　创建 Playground 文件

Playground文件模板有Blank、Game、Map和Single View这4种，我们选择创建一个空的模板，如图1-10所示。

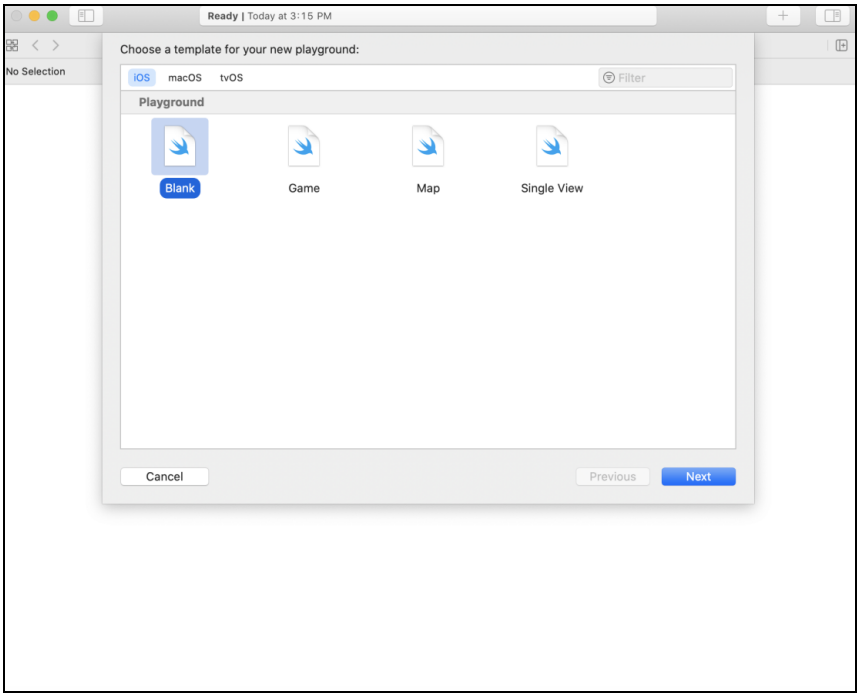

图 1-10　创建空的 Playground 模板

创建完成后，可以看到Playground的界面清爽很多，并且没有了运行与结束运行按钮，当开发者在Playground中编写代码时，可以实时在右侧查看代码的运行情况，如图1-11所示。

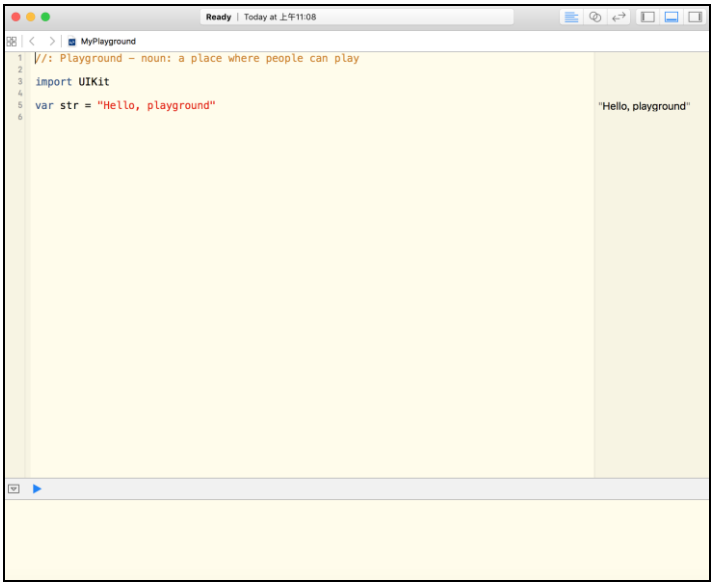

图 1-11　Playground 演练板

通过图1-11可以看到，Playground右侧会将此行代码的相关运行情况反馈给开发者，例如变量计算的值、函数类型、函数运算的结果、打印信息等。对于编程语言的初学者来说，使用Playground可以方便快速地进行上手练习。上面示例的程序是使用Swift语言创建一个字符串变量str，并且对这个变量进行赋值，赋值为“Hello，playground”字符串，右侧展示的为此行代码变量的值。如果代码中有打印操作，除了在右侧展示打印结果外，Playground界面的下侧调试打印区也会对结果进行打印。Playground并非只是一个单文件，我们可以向其中添加其他文件以及资源，总体来说，Playground的设计还是非常强大的。

本章是本书的准备章节，从下一章开始，读者将真正进入Swift语言的学习。在语法学习阶段，读者可能会感到十分枯燥，也可能会感到收效甚微，请坚定信心，在语法阶段打下扎实的基础后，后面iOS的开发学习阶段将事半功倍，再经过本书第3部分的项目实战，读者就可以真正融会贯通，成为一名合格且优秀的Swift开发者！

第2章

量值与基本数据类型

无论数学的任一分支是多么抽象，总有一天会应用在这实际世界上。

—— 尼古巴斯·伊万诺维奇·罗巴切夫斯基

变量一词源于数学，在计算机中，它被用来表示可以改变的值或者计算结果的抽象概念。与变量对应的是常量，它也是一种抽象概念，只是大多数情况下常量表示的值或计算结果是不可改变的。在大多数高级编程语言中，常量和变量的含义往往是广义的，它们可以表示一个具体类型的值、一段代码块、一个内存地址或者一个函数方法，本书中将变量和常量统称为量值。

数据类型则是将具有相同属性的数据进行分类，计算机中所有内容的实质都是数据，计算机的工作原理就是将这些数据存储在内存中的某个位置，并且在需要使用时快速方便地找到它，然后对其进行各种运算操作。不同的数据所占有的内存空间可能会有很大的差异，例如整数数据与浮点数（小数）数据、字符串数据与集合数据等，为了使各类数据能够最优地分配内存，避免不必要的内存消耗，大多数编程语言都定义了一系列的数据类型，Swift也不例外。本章将向读者介绍Swift中支持的基本数据类型，如整型、浮点型、布尔型、元组、可选类型等。

通过本章，你将学习到：

- 常量与变量的意义、声明、命名规范、类型。
- 数学进制与计算机存储原理。
- 整型数据、浮点型数据、布尔型数据的应用。
- Swift语言中的元组类型和可选类型。
- 如何为类型取别名。

2.1 变量与常量

在Swift语言中，let关键字和var关键字分别用来表示常量和变量，无论是let还是var，其作用都是为某个具体量值取了一个名称，在编程中，这种方式叫作量值的声明。在量值的有效作用域内，开发者可以使用这些名称来获取具体的量值。编程中有两个基本的概念：量值和表达式。我们可以简单地将量值理解为结果，例如数字3就是一个整数型量值，字符串“hello”就是一个字符串型量值。而表达式可以理解为一个计算过程，其结果是一个量值，例如1+2就是一个表达式，其结果为量值3；“hello”+“world”也是一个表达式，其结果为量值“hello world”。大多数表达式都是由量值与运算符组成的，这些会在后面具体向读者介绍。

2.1.1 变量与常量的定义和使用

使用Xcode开发工具创建一个名为Swift_Basic的Playground文件，可以看到模板中自动生成了以下两行代码：

```
//引入 UI 开发框架
import UIKit
//定义一个变量，赋值为字符串"Hello, playground"
var str = "Hello, playground"
```

上面的代码中，第1行代码引入了iOS开发框架中的一个UI框架，后面的实战阶段会向读者详细介绍。第2行代码实际上进行了两步操作，首先声明了一个变量str，str就是此变量的名称，之后将“Hello，playground”字符串赋值给这个str变量。我们可以将以上代码分解为如下代码：

```
//1 声明字符串变量 str
var str:String
//2 对字符串变量 str 进行赋值
str = "hello,playground"
```

上面的代码中演示了为量值指定类型的语法，即在常量或者变量名后加冒号，冒号之后写上指定的类型名。Swift是一种类型安全语言，即常量或者变量在声明的时候必须指定明确的类型。看到这里，读者可能会有一些疑问，为何在Xcode生成的模板代码中没有指定str变量的类型，系统依然没有报错，原因要归功于Xcode编译器，Xcode编译器支持对Swift语言的类型自动推断，当声明变量时，如果直接给变量赋初值，则编译器会根据赋值的类型来确定变量的类型，之后变量的类型将不可更改。Swift中可以使用print()函数来进行打印操作，例如打印变量str，示例代码如下：

```
//量值的打印
print(str)
```

在使用常量或者变量时，开发者可以直接通过名称来调用对应的量值，示例代码如下：

```
//更改 str 变量的值
str = "newValue"
```

```
//在 str 字符串变量后边追加 hello
str = str+"hello"
```

Swift语言也支持在同一行语句中声明多个常量或者变量，但是要遵守明确类型的原则，至于具体类型是开发者指定的还是编译器推断的并无关系，例如：

```
//声明定义了 3 个变量：整数类型变量 a、浮点数类型变量 b 和字符串类型变量 c
//编译器推断
var a=1,b=2.9,c="string"
//手动指定
var a2:Int=1,b2:Float=2.9,c2:String="string"
```

如果在同一行代码中声明多个变量并且都没有提供初始值，可以通过指定最后一个变量的类型对整体进行类型指定，例如：

```
//声明 3 个 Int 类型的变量
var one,two,three:Int
```

上面的代码中声明的one、two、three都是Int型变量。

提　示

（1）Swift 语言是一种十分快速简洁的语言，其允许开发者省略分号，自动以换行来分隔语句，同时支持在一行中编写多句代码，此时需要使用分号对语句分隔，例如：

```
var str:String;str = "hello,playground";print(str)
```

（2）对 Swift 语言的类型推断是 Xcode 编译器一个十分优秀的特性，在实际开发中，开发者应该尽量使用这种特性。

（3）如果需要修改变量的值，直接对变量再赋值即可。需要注意的是，所赋值的类型必须和变量的类型保持一致。

2.1.2　变量和常量的命名规范

在Swift语言中，常量和变量的命名规则十分宽泛，可以包括Unicode字符和数字，需要注意的是，不可使用预留关键字来作为常量或者变量的名称，例如let、var这类关键字不可作为量值名来声明。另外，常量和变量的命名不可以数字开头，空格、数学符号、制表符、箭头等符号也不可用在命名中。

可以使用中文进行命名，示例如下：

```
//使用中文进行变量的命名
var 珲少 = "珲少"
```

也可以使用表情符号进行命名，如图2-1所示。

```
//使用表情符号进行命名
var 😄 = "开心"
```

图 2-1　使用表情符号进行变量的命名

也可以使用穿插数字进行命名，注意数字不能作为开头：

```
//含有数字的命名
var pen2 = "钢笔"
```

也可以使用下画线进行命名：

```
//使用下画线进行命名
var _swift_ = "swift"
```

虽然Swift支持的命名方式十分广泛，但在实际开发中，良好的命名风格可以大大提高编码效率与代码的可读性。Swift语言官方文档采用驼峰命名的方式。所谓驼峰命名，是指以单词进行名称的拼接，名称的首字母一般为小写，之后每个单词的首字母大写，其他字母均小写，示例如下：

```
//驼峰命名
var userName = "珲少"
```

提　示

（1）Unicode（统一码、万国码、单一码）是计算机科学领域的一项业界标准，包括字符集、编码方案等。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。Unicode 于 1990 年开始研发，1994 年正式公布。

（2）Swift 中的命名也有一些约定俗成的规则，例如量值属性首字母会小写，类名、枚举名、结构体名首字母会大写。

（3）如果在命名中真的需要使用预留的关键字进行命名，可使用`符号进行包装，但是除非万不得已，开发中尽量不要使用这种方式命名，包装示例如下：

```
//用预留关键字进行命名
var `var` = 2
```

2.2　关于注释

注释是写给开发者自己看的解释性文本，在代码进行编译时，注释语句并不会被编译进工程中，合理地运用注释可以使项目工程的结构更加清晰，团队合作更加顺畅。Swift语言采用和C语言类似的注释方式，使用//符号来注释单行内容，同时也可以使用以/*开头、以*/结尾的方式进行多行注释，示例如下：

```
//单行注释
/*
 多行注释
 注释
 注释
 */
```

Swift语言的注释还有一个十分有趣的特性，即可以进行注释的嵌套，示例如下：

```
//单行注释//注释中的注释
/*
 多行注释
 /*
    注释中的注释
 */
 注释
 注释
 */
```

2.3　初识基本数据类型

本节主要向读者介绍整型、浮点型、布尔型数据在Swift语言中的应用。

2.3.1　数学进制与计算机存储原理

所谓进制，是数学计算中人为规定的一套进位规则。生活中，人们习惯使用十进制进行数据计算，例如到文具店买3支铅笔，到菜市场买菜花费5元3角，等等。在数学与计算机领域，除了十进制之外，二进制、八进制、十六进制的应用也十分广泛，进制的实质是在数据计算时逢几进一（十进制是逢十进一，十六进制是逢十六进一，二进制就是逢二进一，以此类推，x进制就是逢x进位）。

计算机是由逻辑电路组成的，逻辑电路通常只有两个状态，即开关的接通与断开，正好可以表示两种状态（0和1）。对计算机而言，采用二进制不仅能够简化运算法则，提高运算效率，更具有很高的抗干扰能力和可靠性，因此二进制也被称为“机器的语言”。

Swift语言支持开发者使用多种进制进行数据的定义与计算，默认为十进制，如果有特殊需求，可以通过在数据前面加前缀的方式实现，示例如下：

```
var type_10 = 17;        //十进制的 17
var type_2 = 0b10001     //二进制的 17
var type_8 = 0o21        //八进制的 17
var type_16 = 0x11       //十六进制的 17
```

在进一步了解了数据类型的相关知识外，读者首先应该清楚几个概念，计算机内存中最小的数据运算单元是一个二进制位（bit），其只有两种状态：0或者1。字节（B）是最小的数据单元，1字节由8个二进制运算位组成。针对无符号数来说，1字节最大可以表示的数为二进制11111111，即十进制数255。读者如果有一些编程经验，一定会对ASCII码十分熟悉，ASCII码的存储空间即1字节大小，因此其最多可以表示256个字符。在字节之上，还有千字节（KB）、兆字节（MB）、吉字节（GB）、太字节（TB）等，它们之间的换算关系如下：

1B=8bit

1KB=2^10B

1MB=2^10KB
1GB=2^10MB
1TB=2^10GB
1PB=2^10TB

2.3.2 整型数据

Swift语言中的整型数据分为有符号整型数据与无符号整型数据。所谓有符号与无符号，通俗的理解即为分正负号与不分正负号。

对于无符号整型，Swift中提供了5种类型4种存储空间的数据类型，4种存储空间分别占用内存8位、16位、32位和64位。使用Xcode开发工具创建一个新的Playground，命名为BasicDataType，编写如下演示代码：

```
//8 位无符号整型数的最大值 255
var a1 = UInt8.max
//16 位无符号整型数的最大值 65535
var a2 = UInt16.max
//32 位无符号整型数的最大值 4294967295
var a3 = UInt32.max
//64 位无符号整型数的最大值 18446744073709551615
var a4 = UInt64.max
```

上面的代码中创建了4个变量a1、a2、a3、a4。在Swift语言中，整型数据类型实际上是采用结构体的方式实现的，其中max属性可以获取当前类型的最大值。读者可能会有疑问，在实际开发中，到底应该选择哪一种类型来表达无符号整型呢？上面有提到，Swift语言中的无符号整型实际上有5种，还有1种为UInt类型，这种类型编译器会自动适配，在64位的机器上为UInt64，在32位的机器上为UInt32，示例代码如下：

```
//获取数据类型所占位数，在 64 位机器上 UInt 占 8 字节 64 位
var a5 = MemoryLayout<UInt>.size
```

MemoryLayout是Swift标准库中定义的一个枚举，顾名思义其实用于获取内存相关信息，MemoryLayout<UInt>则是一种泛型的用法，调用其size属性可以获取某种数据类型所占内存空间的字节数。

有符号整型数据与无符号整型数据十分类似，只是其首位二进制位为符号位，不纳入数值计算，示例代码如下：

```
var maxInt8 = Int8.max    //127
var mimInt8 = Int8.min   //-128
var maxInt16 = Int16.max  //32767
var minInt16 = Int16.min  //-32768
var maxInt32 = Int32.max  //2147483647
var minInt32 = Int32.min  //-2147483648
var maxInt64 = Int64.max  //9223372036854775807
var minInt64 = Int64.min  //-9223372036854775808
var intSize = sizeof(Int)    //8 位
```

与max属性对应，min属性用于获取整型数据的最小值。

提　示

如果我们可以明确当前场景不会出现负数，则可以使用 UInt 类型，更多时候如果我们不能够保证，最好使用 Int 类型，即我们存储的数据是非负的，对数值统一使用 Int 类型有助于数据间的传递和转换。

2.3.3　浮点型数据

浮点型数据用来表示一些小数，浮点型数据分为单精度浮点型与双精度浮点型，分别用Float与Double表示，示例代码如下：

```
var b = MemoryLayout<Float>.size          //4 字节
var b1 = MemoryLayout<Float32>.size       //4 字节
var b2 = MemoryLayout<Float64>.size       //8 字节
var b3 = MemoryLayout<Float80>.size       //16 字节
var c = MemoryLayout<Double>.size         //8 字节
```

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

```
var sum = 1.25e3 //1.25*(10^3) = 1250
var sun2 = 0x1p3 //1*(2^3) = 8
```

提　示

Double 类型比 Float 类型有着更高的精度，除了某些特殊场景外，我们更推荐使用 Double 类型来定义浮点数。

Swift语言中还有一个十分有意思的特性，无论是整型数据还是浮点型数据，都可以在数字前加任意个0来进行位数填充，也可以在数字中加入下画线进行分隔，进而增加可读性，这些操作并不会影响原始数值，却提高了对开发者编程的友好性，使代码的结构更加清爽，示例如下：

```
var num1 = 001.23           //1.23
var num2 = 1_000            //1000
var num3 = 1_000.1_ 001     //1000.1001
```

2.3.4　布尔型数据

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

创建真与假的布尔值，示例代码如下：

```
var bool1 = true    //创建布尔真变量
var bool2 = false   //创建布尔假变量
```

2.4 两种特殊的基本数据类型

Swift语言还支持两种特殊的基本数据类型，分别是元组类型与可选值类型。元组在实际开发中十分常用，开发者使用元组可以创建出任意数据类型组合的自定义数据类型；而可选值类型是Swift语言的一大特点，通过可选值类型，Swift语言对数值为空进行了严格的把控。

2.4.1 元组

元组是Swift语言中重要的数据类型之一，元组允许一些并不相关的类型自由组合成为新的集合类型。Objective-C语言并不支持元组这样的数据类型，这在很多时候会给开发者带来麻烦。类比生活中的一种情景，元组类型类似于日常生活中的套餐，现在各种服务业都推出了许多有特色的套餐供顾客选择，方便为顾客提供一站式服务。元组提供的就是这样一种编程结构，试想一下，编程中遇到这样一种情形：一个商品有名字和价格，使用元组可以很好地对这种商品类型进行模拟，示例如下：

```
//创建钢笔元组类型，其中有两种类型，即字符串类型的名称和整数类型的价钱
var pen:(name:String,price:Int) = ("钢笔",2)
```

上面的代码在创建元组类型的同时指定了其中参数的名称，即名称参数为name，价格参数为price，开发者可以使用这些参数名称来获取元组中各个参数的值，示例如下：

```
//获取 pen 变量名称
var name = pen.name
//获取 pen 变量价格
var price = pen.price
```

开发者在创建元组时，也可以不指定元组中参数的名称，元组会自动为每个参数分配下标，下标值将从0开始依次递增，示例如下：

```
//不指定参数名称的元组
var car:(String,Int) = ("奔驰",2000000)
//通过下标来取得元组中各个组成元素的值
var carName = car.0
var carPrice = car.1
```

元组实例被创建后，开发者也可以通过指定的变量或者常量来分解它，示例如下：

```
//不指定参数名称的元组
var car:(String,Int) = ("奔驰",2000000)
//进行元组的分解
var (theName,thePrice) = car
//此时 theName 变量被赋值为"奔驰"，thePrice 变量被赋值为 2000000
print(theName,thePrice)
```

上面的代码将元组实例car中的各个组成元素分解到具体变量，有一点读者需要注意，分解后的变量必须与元组中的元素一一对应（个数相等），否则编译器就会报错。代码中使用的print()函数为打印输出函数，print()函数可以接收多个参数，将其以逗号分隔即可。有些时候，开发者可能并不需要获取某个元组实例中所有元素的值，这种情况下，开发者也可以将某些不需要获取的元素使用匿名的方式来接收，示例如下：

```
//不指定参数名称的元组
var car:(String,Int) = ("奔驰",2000000)
//进行元组的分解，将 Int 型参数进行匿名
var (theName,_) = car
//此时 theName 变量被赋值为"奔驰"
print(theName)
```

在Swift语言中，常常使用符号“_”来表示匿名的概念，因此“_”也被称为匿名标识符。上面的代码实际上只分解出了元组car中的第一个元素（String类型）。

提　示

元组虽然使用起来十分方便，但是其只适用于简单数据的组合，对于结构复杂的数据，要采用结构体或者类来实现。

2.4.2　可选值类型

可选值类型（Optional类型）是Swift语言特有的一种类型。首先，Swift语言是一种十分强调类型安全的语言，开发者在使用到某个变量时，编译器会尽最大可能保证此变量的类型和值的明确性，保证减少编程中的不可控因素。然而在实际编程中，任何类型的变量都会遇到值为空的情况，在Objective-C语言中并没有机制来专门监控和管理为空值的变量，程序的运行安全性全部靠开发者手动控制。Swift语言提供了一种包装的方式来对普通类型进行Optional包装，实现对空值情况的监控。

在Swift语言中，如果使用了一个没有赋值的变量，程序会直接报错并停止运行。读者可能会想，如果一个变量在声明的时候没有赋初值，在后面的程序运行中就有可能被赋值，那么对于这种“先声明后赋值”的应用场景，我们应该怎么做呢？在Objective-C中，这个问题很好解决，只需要使用的时候判断一下这个变量是否为nil即可。在Swift中是否也可以这样做呢？使用如下代码来进行试验：

```
var obj:String
if obj==nil {

}
```

编写上面的代码后，可以看到Xcode工具依然抛出了一个错误提示，其实在Swift语言中，未做初始化的普通类型是不允许使用的，哪怕是用来进行判空处理也不被允许，当然也就不可以与nil进行比较运算，这种机制极大地减小了代码的不可控性。因此，开发者在使用前必须保证变量被初始化，代码如下：

```
var obj0:String
```

```
obj0 = "HS"
print(obj0)
```

但是上面的做法在实际开发中并不常用，如果一个变量在逻辑上可能为nil，则开发者需要将其包装为Optional类型，改写上面的代码如下：

```
var obj:String?
if obj==nil {

}
```

此时代码就可以正常运行了。分析上面的代码，在声明obj变量的时候，这里将其声明成了String?类型，在普通类型后面添加符号“?”，即可将普通类型包装为Optional类型。

Optional类型不会独立存在，其总是附着于某个具体的数据类型之上，具体的数据类型可以是基本数据类型，可以是结构体，也可以是类等。Optional类型只有两种值，读者可以将其理解为：

- 如果其附着类型对应的量值有具体的值，则其为具体值的包装。
- 如果其附着类型对应的量值没有具体的值，则其为 nil。

举一个例子，将Int类型的变量a进行Optional包装，此时a的类型为Int?，如果对a进行了赋值，则可以通过拆包的方式获取a的Int类型值，如果没有对a进行赋值，则a为nil。

提　示
Optional 类型中的 nil 读者也可以理解为一种值，其表示空。

Optional类型是对普通类型的一种包装，因此在使用的时候需要对其进行拆包操作，拆包将使用到Swift中的操作符“!”。“?”与“!”这两个操作符是很多Swift语言初学者的噩梦，如果读者理解了Optional类型，那么对这两个操作符的理解和使用将容易许多。首先需要注意，“?”符号可以出现在类型后面，也可以出现在实例后面，如果出现在类型后面，其代表的是此类型对应的Optional类型，如果出现在实例后面，则代表的是可选链的调用，后面章节会详细介绍。“！”符号同样可以出现在类型后面与实例后面，它出现在类型后面代表的是一种隐式解析的语法结构，后面章节会介绍；出现在实例后面代表的是对Optional类型实例的拆包操作。示例如下：

```
//声明 obj 为 String?类型
var obj:String? = "HS"
//进行拆包操作
obj!
```

读者需要注意，在使用“!”进行Optional值的拆包操作时，必须保证要拆包的值不为nil，否则程序运行会出错。可以在拆包前使用if语句进行安全判断，示例如下：

```
//声明 obj 为 String?类型
var obj:String? = "HS"
if obj != nil {
    obj!
}
```

上面的代码演示的编程结构在实际应用中十分广泛，因此Swift语言还提供了一种if-let语法结

构来进行Optional类型值的绑定操作，可以将上面的结构改写如下：

```
var obj:String? = "HS"
//进行 if-let 绑定
if let tmp = obj {
   print(tmp)
}else{
   obj = "HS"
   print(obj!)
}
```

上面的代码可以这样理解：如果obj有值，则if-let结构将创建一个临时常量tmp来接收obj拆包后的值，并且执行if为真时所对应的代码块，在执行的代码块中，开发者可以直接使用拆包后的obj值tmp。如果obj为nil，则会进入if为假的代码块中，开发者可以在else代码块中将obj重新赋值使用。这种if-let结构实际上完成了判断、拆包、绑定拆包后的值到临时常量3个过程。

if-let结构中也可以同时进行多个Optional类型值的绑定，之间用逗号隔开，示例如下：

```
//if-let 多 Optional 值绑定
var obj1:Int? = 1
var obj2:Int? = 2
if let tmp1 = obj1,let tmp2 = obj2 {
   print(tmp1,tmp2)
}
```

在同时进行多个Optional类型值的绑定时，只有所有Optional值都不为nil，绑定才会成功，代码执行才会进入if为真的代码块中。如果开发者需要在if语句的判断中添加更多业务逻辑，可以通过追加子句的方式来实现，示例如下：

```
//if-let 多 Optional 值绑定
var obj1:Int? = 1
var obj2:Int? = 2
if let tmp1 = obj1,let tmp2 = obj2 ,tmp1<tmp2 {
   print(tmp1,tmp2)
}
```

上面的代码在obj1不为nil、obj2不为nil并且obj1所对应的拆包值小于obj2对应的拆包值的时候才会进入if为真的代码块中，即打印绑定的tmp1与tmp2的值。

你可能发现了，对于一个可选值类型的变量，每次使用时我们都需要为其进行拆包操作，这相对会有些麻烦，其实Swift中还有一种语法：隐式解析。隐式解析适用于这样的场景：当我们明确某个变量初始时为nil，并且在之后使用之前一定会被赋值时，我们可以将其声明为隐式解析的可选值，再对这个变量进行使用，就不需要进行拆包操作了，例如下面的代码会产生运行错误：

```
var obj4:Int?
obj4 = 3
print(obj4 + 1)  //会编译异常，因为obj4没有进行拆包
```

如果将上面的代码做如下的修改，就可以正常运行了：

```
// 声明obj4为隐式解析的变量
```

```
var obj4:Int!
obj4 = 3
// 在使用时，无须再进行拆包操作，Swift会自动帮我们拆包
print(obj4 + 1)
```

Optional值在Swift语言编程中的应用十分灵活，在以后的编程练习中，读者会逐步体会其中的奥妙。

2.5 为类型取别名

在C、C++、Objective-C这些语言中都提供了typedef这样的关键字来为某个类型取一个别名，Swift语言中使用typealias关键字来实现相同的效果，示例如下：

```
//为 Int 类型取一个别名 Price
typealias Price = Int
//使用 Price 代替 Int，效果完全一样
var penPrice:Price = 100
```

上面的代码为Int类型取了别名Price，在后面的使用中，Price类型和Int类型一模一样。在实际开发中，灵活使用typealias为类型取别名可以优化代码的可读性。

2.6 练习及解析

（1）使用两种类型指定方式分别创建Int型变量a=1、b=2，交换a和b的值。

示例解析：

```
var a:Int = 1
var b = 2
//中间变量进行交换
var c = a
a = b
b = c
```

（2）创建4个变量，并分别将十进制数25用二进制、八进制、十进制与十六进制赋值。

示例解析：

```
var count1 = 25             //十进制
var count2 = 0o31           //八进制
var count3 = 0x19           //十六进制
var count4 = 0b00011001     //二进制
```

（3）小文到文具店买文具，其需要购买铅笔、橡皮和文具盒3种文具，3种文具的标价分别为2元、1元和15元，使用元组来模拟这3种文具组成的套装。

示例解析：

```
var bundle:(pencil:Int,eraser:Int,pencilCase:Int) = (2,1,15)
```

（4）编写一个样品质量检测器，当样品的质量大于30单位的时候，输出合格，输入样品可能为空，使用if-let语句来实现。

示例解析：

```
var product:Int? = 100
if let weight = product, weight > 30  {
   print("产品合格")
}
```

2.7　模拟面试

（1）符号“?”和“！”是Swift工程中非常常见的两个符号，请简述你对它们的理解。

回答要点提示：

①首先从类型和实例两个方面理解，“？”出现在类型后表示Optional类型，出现在实例后表示可选链调用。“!”出现在类型后表示默认隐式解析，出现在实例后表示强制拆包。

②这两个符号都与Swift中的Optional类型相关，Optional类型是Swift语言强调安全性的一种方式，某个变量可不可以为空应该是逻辑上决定的，而不是不可预知、不可控的。

③if-let结构与Optional类型值结合使用可以编写出优雅安全的逻辑代码。

核心理解内容：

对Swift中的Optional类型做深入的理解。

（2）十进制、二进制、八进制、十六进制各有什么优势，在哪些场景下使用。

回答要点提示：

①十进制的优势不必多言，日常生活中几乎所有的数学计算使用的都是十进制，钱币的单位、班级的座次、队伍的排序等都是以十进制表示的。

②二进制是计算机最方便理解的进制方式，高低电平状态非常容易表示二进制的0和1，同时也是计算机运行最稳定的存储数据进制方式。

③八进制和十六进制实际上是二进制的聚合方式，在八进制中，每位数字可以表示二进制中的3位，在十六进制中，每位数字可以表示二进制的4位，大大缩短了二进制数的长度，并且便于阅读。常常使用十六进制来表示颜色数据。

核心理解内容：

进制的原理以及转换方法。

（3）Swift语言中是否只有var和let两种数据类型？

回答要点提示：

①这个命题大错特错，在Swift中，var和let并不是数据类型，只是两种用来声明变量的方式。

②Swift是一种强数据类型，和C、C++、Objective-C、Java等语言一样，变量在声明时其数据

类型就已经确定，有时我们没有显式指定，是由于Xcode有自动类型推断功能。

③Swift中的数据类型有基本数据类型和引用数据类型，基本数据类型中又包含整型、浮点型、布尔型、元组等。

核心理解内容：

理解数据类型的意义，理解变量和数据类型之间的关系，明白Xcode的自动类型推断功能。

第3章

字符、字符串与集合类型

单丝不成线，独木不成林。

——中华民谚

在程序开发中，字符串的使用必不可少，其是编程中一种十分重要的数据类型，其实，字符串也是一组字符的集合。有些语言是没有独立的字符串类型的，例如C语言，其往往采用字符数组来作为字符串类型，Objective-C语言中封装了面向对象的字符串类型NSString，并向其中封装了大量的相关方法。而Swift是一种弱化指针的语言，它提供了String类型和Character类型来描述字符串与字符。

集合类型是用于描述一组数据的集合体，例如一组整数组合在一起形成整数集合，一组字符串组合在一起形成字符串集合，等等。在Swift语言中一共提供了3种集合类型，即数组（Array）、集合（Set）和字典（Dictionary），这3种集合类型虽有很多共同点，但在实现上有许多差异，它们分别适用于不同的业务场景。

通过本章，你将学习到：

- 构造字符串、内嵌格式化字符串和分解字符串。
- Swift中的转义字符。
- 字符串相关方法的使用。
- 数组的建立和元素的增、删、改、查。
- 集合的建立与数学运算。
- 字典的建立及数据的操作方法。

3.1 字符串类型

字符串类型顾名思义为一串字符的组合，其在开发中应用甚广，商品的名称、学生的班级、播放音乐的歌词等场景逻辑都需要通过字符串来处理。

3.1.1 进行字符串的构造

读者在使用Xcode开发工具创建第一个Playground模板时，里面的代码实际上就演示了字符串变量的创建，代码如下：

```
var str = "Hello, playground"
```

上面的代码就是一种简单的字符串类型变量的构造方式，即直接通过实体字符串进行赋值，读者可以使用Xcode开发工具创建一个名为String的Playground文件，在其中进行字符串相关代码的演练。

如果需要构造空的字符串，可以使用如下方式：

```
var str = ""
```

这里需要注意，在编写代码时，字符串变量的值为空字符串与字符串变量的值为nil是两个完全不同的概念，如果一个Optional类型变量没有赋值，则其为nil，如果赋值为空字符串，则其并不是nil。判断一个字符串变量的值是否为空字符串有特定的方法，后面会进行介绍。

在Swift语言中，String类型实际上是一个结构体，其实前面章节中学习的整型、浮点型和布尔型也是由结构体实现的。Swift语言中的结构体十分强大，其可以像类一样进行属性和方法的定义，关于结构体的知识，后面章节会专门介绍，这里只需要了解即可。开发者也可以使用String结构体的构造方法来构造String类型的量值，示例如下：

```
//直接赋值
var str:String = "Hello, playground"
//直接赋值为空字符串
str = ""
//通过构造方法来进行 str 变量的构造
str = String()              //构造空字符串 ""
str = String("hello")       //通过字符串构造 "hello"
str = String(666)           //通过整型数据构造 "666"
str = String(6.66)          //通过浮点型数据构造 "6.66"
str = String("a")           //通过字符构造 "a"
str = String(false)         //通过 Bool 值构造 "false"
str = String(describing: (1,1.0,true))  //通过元组构造 "(1,1.0,true)"
str = String(describing: [1, 2, 3])     //通过列表构造 "[1, 2, 3]"
str = String(format:"我是%@","珲少") //通过格式化字符串构造 "我是珲少"
```

Sring类型提供了很多重载的构造方法，开发者可以传入不同类型的参数来构造需要的字符

串。实际上，Swift语言中的String类型提供的构造方式十分宽泛，甚至可以将其他类型通过构造方法转换为字符串，示例如下：

```
str = String(describing: Int.self)       //通过类型来构造字符串 "Int"
```

提　示

整型、浮点型数据可以使用构造方法的方式来实现互相转换，例如：

```
var a = Int(1.05)     //将 1.05 转换成 1
var b = Float(a)      //通过整型数据 a 构造浮点型数据 b
```

3.1.2　字符串的组合

Swift中的String类型对“+”运算符进行了重载实现，即开发者可以直接使用“+”符号将多个字符串组合拼接为新的字符串，示例如下：

```
//字符串的组合
var c1 = "Hello"
var c2 = "World"
var c3 = c1+" "+c2  //"Hello World" //注意中间拼接了一个空格
```

通过加法运算符，开发者可以十分方便地进行字符串变量的组合拼接，有时开发者需要在某个字符串中间插入另一个字符串，除了可以使用格式化的构造方法外，Swift中还提供了一种十分方便的字符串插值方法，示例如下：

```
//使用\()进行字符串插值
var d = "Hello \(123)"      //"Hello 123"
var d2 = "Hello \(c2)"      //"Hello World"
var d3 = "Hello \(1+2)"      //"Hello3"
```

“\()”结构可以将其他数据类型转换为字符串类型并且插入字符串数据的相应位置，也可以进行简单的运算逻辑后将结果插入原字符串中，这种方法可以十分方便地进行字符串的格式化，在开发中应用广泛。

3.2　字符类型

字符类型用来表示单个字符，如数字字符、英文字符、符号字符和中文字符等都可以使用字符类型来表示，也可以通过遍历字符串的方法将字符串中的字符分解出来。

3.2.1　字符类型简介

类似于C语言中的Char，Swift语言中使用Character来描述字符类型，Character类型和String类型都占16字节的内存空间。在Swift中可以使用MemoryLayout枚举来获取某个类型所占用的内存空

间，其单位为字节，示例如下：

```
MemoryLayout<String>.size        //16 个字节 获取 String 类型占用的内存空间
```

Character用来描述一个字符，我们将一组字符组合成为一个数组，用于构造字符串，示例如下：

```
//创建一个字符
var e:Character = "a"
//创建字符数组
var e2 : [Character] = ["H","E","L","L","O"]
//通过字符数组来构造字符串 "HELLO"
var e3 = String(e2)
```

同样，也可以使用构造方法来完成字符类型变量的构造，示例如下：

```
//通过构造方法来创建字符类型变量
var e4 = Character("a")
```

使用for-in遍历可以将字符串中的字符拆解出来，这种方法有时十分好用，for-in遍历是Swift语言中一种重要的代码流程结构。String类型默认实现了迭代器相关协议，直接对其进行遍历可以取出字符串中的每一个字符元素，示例代码如下：

```
//进行 for-in 遍历
let name = "China"
for character in name {
   print(character)
}
```

上面的代码将依次打印C、h、i、n、a。

提　示

for-in 结构是一种重要的循环结构，上面的示例代码中，in 关键字后面需要为一种可迭代的类型，in 关键字前面是每次循环从迭代器中取出的元素，其类型会由 Xcode 编译器自动推断出来，在后面的章节中会有 for-in 结构的详细介绍。

3.2.2 转义字符

Swift语言和C语言类似，除了一些常规的可见字符外，还提供了一些有特殊用途的转义字符，可通过特殊的符号组合来表示特定的意义。示例如下：

- \0：用来表示空白符。
- \\：用来表示反斜杠。
- \t：用来表示制表符。
- \n：用来表示换行符。
- \r：用来表示回车符。
- \'：用来表示单引号。

- \"：用来表示双引号。
- \u{}：用Unicode码来创建字符。

其中，\u{}用来通过Unicode码来创建字符，将Unicode码填入大括号中即可，示例如下:

```
//使用 Unicode 码来创建字符，Unicode 为 21 代表的字符为!
"\u{21}"
```

提　示

在应用开发中，换行符常用来处理多行文本的排版。

3.3　字符串类型中的常用方法

Swift语言的String类型中封装了许多实用的属性和方法，例如字符串的检查，字符的追加、插入、删除操作，字符数的统计等。熟练使用这些属性与方法能够使得开发者在编程中处理数据时游刃有余。

前面介绍过，字符串变量的值为空字符串与字符串变量的值为空是两个不同的概念，String类型的实例通过使用isEmpty方法来判断字符串的值是否为空字符串，示例如下：

```
//判断字符串是否为空
var obj1 = ""
if obj1.isEmpty {
    print("字符串为空字符串")
}
```

还有一种方式也可以用来判断一个字符串变量是否为空字符串，即当字符串变量中的字符数为0时，也可以认定此字符串为空字符串，即通过字符串的count属性判断其中的字符个数是否为0，示例如下：

```
//获取字符串中的字符个数，判断是否为空字符串
if obj1.count == 0 {
   print("字符串为空字符串")
}
```

String类型的实例除了可以使用“+”直接拼接外，还可以使用比较运算符，示例代码如下：

```
var com1 = "30a"
var com2 = "31a"
//比较两个字符串是否相等，只有两个字符串中所有位置的字符都相等时，才为相等的字符串
if com1==com2 {
   print("com1 和 com2 相等")
}
//比较两个字符串的大小
if com1<com2 {
   print("com1 比 com2 小")
}
```

在比较两个字符串的大小时，会逐个对字符的大小进行比较，直至遇到不相等的字符为止。上面的示例代码可以这样理解：先比较com1字符串与com2字符串的第1个字符，若相等，再比较第2个字符，以此类推。由于com2的第4个字符（2）大于com1的第4个字符（1），因此com2字符串大于com1字符串。

开发者可以通过下标的方式来访问字符串中的每一个字符，获取字符串起始下标与结束下标的方法如下：

```
var string = "Hello-Swift"
//获取字符串的起始下标 String.index 类型
var startIndex = string.startIndex
//获取字符串的结束下标 String.index 类型
var endIndex = string.endIndex
```

这里需要注意，startIndex和endIndex获取到的值为Index类型，并不是整数类型，它们不能直接进行加减运算，需要使用相应的方法进行下标的移动操作，示例如下：

```
//获取某个下标后一个下标对应的字符 char="e"
var char = string[string.index(after: startIndex)]
//获取某个下标前一个下标对应的字符 char2 = "t"
var char2 = string[string.index(before: string.endIndex)]
```

上面的代码中，index(after:)方法用来获取当前下标的后一位下标，index(before:)方法用来获取当前下标的前一位下标。也可以通过传入下标范围的方式来截取字符串中的某个子串，示例如下：

```
//通过范围获取字符串中的一个子串 Hello
var subString = string[startIndex...string.index(startIndex, offsetBy: 4)]
var subString2 = string[string.index(endIndex, offsetBy: -5)..<endIndex]
```

上面的示例代码中，“…”为范围运算符，在后面的章节中会详细介绍，offsetBy参数传入的是下标移动的位数，若向其中传入正数，则下标向后移动相应位数，若向其中传入负数，下标向前移动相应位数。使用这种方式来截取字符串十分方便。String类型中还封装了一些方法，可以帮助开发者便捷地对字符串进行追加、插入、替换、删除等操作，示例如下：

```
//获取某个子串在父串中的范围
var range = string.range(of: "Hello")
//追加一个字符，此时 string = "Hello-Swfit!"
string.append(Character("!"))
//追加字符串操作，此时 string = "Hello-Swift! Hello-World"
string.append(" Hello-World")
//在指定位置插入一个字符，此时 string = "Hello-Swift!~ Hello-World"
string.insert("~", at: string.index(string.startIndex, offsetBy: 12))
//在指定位置插入一组字符，此时 string = "Hello-Swift!~~~~ Hello-World"
string.insert(contentsOf: ["~","~","~"], at: string.index(string.startIndex,
offsetBy: 12))
//在指定范围替换一个字符串，此时 string = "Hi-Swift!~~~~ Hello-World"
string.replaceSubrange(string.startIndex...string.index(string.startIndex,
offsetBy: 4), with: "Hi")
//在指定位置删除一个字符，此时 string = "Hi-Swift!~~~~ Hello-Worl"
string.remove(at: string.index(before:string.endIndex))
//删除指定范围的字符，此时 string = "Swift!~~~~ Hello-Worl"
```

```
    string.removeSubrange(string.startIndex...string.index(string.startIndex,
offsetBy: 2))
    //删除所有字符，此时 string = ""
    string.removeAll()
```

下面的方法可以方便地完成字符串的大小写转换：

```
var string2 = "My name is Jaki"
//全部转换为大写
string2 = string2.uppercased() //结果为"MY NAME IS JAKI"
//全部转换为小写
string2 = string2.lowercased() //结果为"my name is jaki"
```

下面的方法可以用来检查字符串的前缀和后缀：

```
//检查字符串是否有 My 前缀
string2.hasPrefix("My")
//检查字符串是否有 jaki 后缀
string2.hasSuffix("jaki")
```

温馨提示：本章介绍了许多String类型中封装的方法，熟练运用这些方法可以极大地提高开发者的编程效率，本章后面会为读者准备丰富的练习题，尽可能多地实践练习是掌握一门编程语言语法的不二法门。

3.4　集合类型

在Swift语言中一共提供了3种集合类型：数组（Array）、集合（Set）和字典（Dictionary）。数组类型是一种有序集合，放入其中的数据都有一个编号，且编号从0开始依次递增。通过编号，开发者可以找到Array数组中对应的值。集合是一组无序的数据，其中存入的数据没有编号，开发者可以使用遍历的方法获取其中所有的数据。集合是一种键值映射结构，其中每存入一个值都要对应一个特定的键，且键不能重复，开发者通过键可以直接获取到对应的值。Swift官方开发文档中的一张示例图片可以十分清晰地描述这3种集合类型的异同，如图3-1所示。

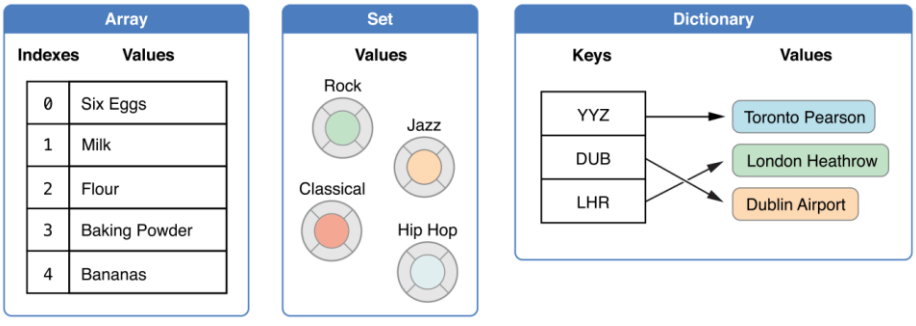

图 3-1　三种集合类型的异同

本节将介绍这三种集合类型的特点及操作数据的方法。

3.4.1 数组（Array）类型

数组中能够存放的元素并非只是数字，它可以存放任意类型的数据，但是所有数据的类型必须统一。在实际开发中，数组中元素的类型决定了数组的类型，例如，一个存放整型数据的数组被称为整型数组，一个存放字符串型数据的数组被称为字符串型数组。在创建数组实例的时候，必须明确指定其中所存放元素的类型。使用Xcode开发工具创建一个名为CollectType的Playground，在其中编写如下示例代码：

```
//Int 型数组
var array1:[Int]
var array2:Array<Int>
```

上面两行代码都声明了一个Int类型的数组实例，数组的创建可以使用两种方式，一种是使用数组的构造方法来创建，另一种是使用中括号来快捷创建，示例如下：

```
//创建空数组
array1 = []
array2 = Array()
//创建整型数组
array1 = [1,2,3]
//通过一组元素创建数组
array2 = Array(arrayLiteral: 1,2,3)
```

提　示

和 String 类型类似，空数组的含义并非是变量为 nil，而是数组中的元素为空，Swift 中只有 Optional 类型的变量可以为 nil。

在Swift语言中，数组采用结构体来实现，对于大量重复元素的数组，开发者可以直接使用快捷方法来创建，示例如下：

```
//创建大量相同元素的数组
//创建有 10 个 String 类型元素的数组，并且每个元素都为字符串"Hello"
var array3 = [String](repeating: "Hello", count: 10)
//创建有 10 个 Int 类型元素的数组，且每个元素都为 1
var array4 = Array(repeating: 1, count: 10)
```

读者需要注意，数组在声明时必须要明确其类型，但是开发者并不一定需要显式地指定类型，如果数组在声明时也设置了初始值，则编译器会根据赋值类型自动推断出数组的类型。数组数组中对加法运算符也进行了重载，开发者可以使用“+”进行两个数组的相加，相加的结果即将第2个数组中的元素拼接到第1个数组后面。需要注意，相加的数组类型必须相同，示例如下：

```
//数组相加 array5 = [1,2,3,4,5,6]
var array5 = [1,2,3]+[4,5,6]
```

数组中提供了许多方法供开发者来获取数组实例的相关信息或者对数组进行增、删、改、查的操作。示例如下：

```
var array = [1,2,3,4,5,6,7,8,9]
//获取数组中元素的个数 9
array.count
//检查数组是否为空数组
if array.isEmpty {
    print("array 为空数组")
}
//通过下标获取数组中的元素 1
var a = array[0]
//获取区间元素组成的新数组 [1,2,3,4]
var subArray = array[0...3]
//获取数组的第 1 个元素
var b = array.first
//获取数组的最后一个元素
var c = array.last
//修改数组中某个位置的元素
array[0] = 0
//修改数组中区间范围的元素
array[0...3] = [1,2,3,4]
//向数组中追加一个元素
array.append(10)
//向数组中追加一组元素
array.append(contentsOf: [11,12,13])
//向数组中的某个位置插入一个元素
array.insert(0, at: 0)
//向数组中的某个位置插入一组元素
array.insert(contentsOf: [-2,-1], at: 0)
//移除数组中某个位置的元素
array.remove(at: 1)
//移除数组中首个位置的元素
array.removeFirst()
//移除最后一个位置的元素
array.removeLast()
//移除前几位元素 参数为要移除元素的个数
array.removeFirst(2)
//移除后几位元素 参数为要移除元素的个数
array.removeLast(2)
//移除一个范围内的元素
array.removeSubrange(0...2)
//替换一个范围内的元素
array.replaceSubrange(0...2, with: [0,1])
//移除所有元素
array.removeAll()
//判断数组中是否包含某个元素
if array.contains(1){
    print(true)
}
```

这里需要注意，只有当数组实例为变量时，才可以使用增、删、改等方法，常量数组不能进行与修改相关的操作。

开发者也可以使用for-in遍历来获取数组中的元素，示例如下：

```
//Int 型数组
let arrayLet = [0,1,2,3,4]
//(Int,Int)型数组
let arrayLet2 = [(1,2),(2,3),(3,4)]
//直接遍历数组
for item in arrayLet {
   print(item)
}
//进行数组枚举遍历，将输出 (0,0) (1,1) (2,2) (3,3) (4,4)
for item in arrayLet.enumerated(){
   print(item)
}
//进行数组角标遍历
for index in arrayLet2.indices{
  print(arrayLet2[index], separator:"")
}
```

可以直接对数组实例进行遍历，Swift中的for-in结构和Objective-C中的for-in结构还是有一些区别的，Swift中的for-in结构在遍历数组时会按照顺序进行遍历。数组实例中还有一个enumerated()方法，这个方法会返回一个元组集合，将数组的下标和对应元素返回。开发者也可以通过遍历数组的下标来获取数组中的元素，和String类型不同的是，数组中的下标可以是Int类型，而String中的下标是严格的Index类型，这里需要注意，不要混淆。

提　示

数组类型中有一个 indices 属性，这个属性将返回一个范围（Range），此范围就是数组下标的范围。

数组类型中还提供了一个排序函数，如果数组中的元素为整型数据，则可以使用系统提供的sorted(by:)方法来进行排序操作，如果是一些自定义的类型，开发者也可以对sorted(by:)方法传入闭包参数实现新的排序规则，这部分内容会在后面章节中详细介绍。进行数组排序的方法示例代码如下：

```
var arraySort = [1,3,5,6,7]
//从大到小排序
arraySort = arraySort.sorted(by: >)
//从小到大排序
arraySort = arraySort.sorted(by: <)
```

下列方法可以获取数组中的最大值与最小值：

```
var arraySort = [1,3,5,6,7]
//获取数组中的最大值
arraySort.max()
//获取数组中的最小值
arraySort.min()
```

3.4.2　集合（Set）类型

集合类型不关注元素的顺序，但是其中的元素不可以重复，读者也可以将其理解为一个无序的集合。与数组一样，集合在声明时必须指定其类型，或者对其赋初值，使得编译器可以自行推断出集合的类型。声明与创建集合的示例代码如下：

```
//创建 set
var set1:Set<Int> = [1,2,3,4]
var set2 = Set(arrayLiteral: 1,2,3,4)
```

由于集合并不关注其中元素的顺序，因此通过下标的方式来取值对集合来说不太有意义，但是集合类型依然支持通过下标来获取其中的元素，示例如下：

```
//获取集合首个元素（顺序不定）
set1[set1.startIndex]
//进行下标的移动
//获取某个下标后一个元素
set1[set1.index(after: set1.startIndex)]
//获取某个下标后几个元素
set1[set1.index(set1.startIndex, offsetBy: 3)]
```

需要注意，集合的下标操作为不可逆的操作，只能向后移动，不能向前移动。

下面这个方法可以获取集合实例中的一些信息：

```
//获取元素个数
set1.count
//判断集合是否为空集合
if set1.isEmpty {
   print("集合为空")
}
//判断集合中是否包含某个元素
if set1.contains(1){
   print("集合包含")
}
//获取集合中的最大值
set1.max()
//获取集合中的最小值
set1.min()
```

集合同样支持进行增、删、改、查操作，示例如下：

```
//向集合中插入一个元素
set1.insert(5)
//移除集合中的某个元素
set1.remove(1)
//移除集合中的第一个元素
set1.removeFirst()
//移除集合中某个位置的元素
set1.remove(at: set1.firstIndex(of: 3)!)
```

```
//移除集合中所有的元素
set1.removeAll()
```

在使用remove(at:)方法删除集合某个位置的元素时，需要传入一个集合元素的下标值，通过集合实例的firstIndex(of:)方法可以获取具体某个元素的下标值。需要注意，这个方法将会返回一个Optional类型的可选值，因为要寻找的元素可能不存在，在使用时，开发者需要对其进行拆包操作。

集合与数组除了有序和无序的区别外，集合还有一个独有的特点：可以进行数学运算，例如交集运算、并集运算、补集运算等。Swift官方开发文档中的一张图片示意了集合进行数学运算时的场景，如图3-2所示。

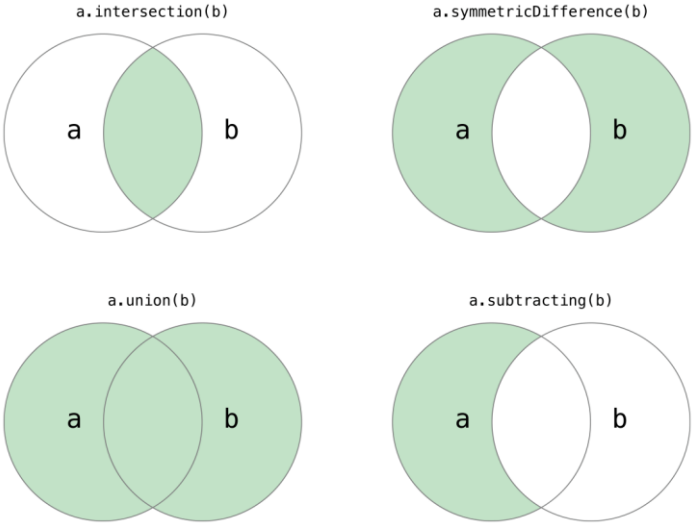

图 3-2　集合进行数学运算示意图

从图3-2中可以看出，集合支持4类数学运算，分别为intersection（交集）运算、symmetricDifference（交集的补集）运算、union（并集）运算和subtracting（补集）运算。交集运算的结果为两个集合的交集，交集的补集运算的结果为a集合与b集合的并集除去a集合与b集合的交集，并集运算的结果为两个集合的并集，补集运算的结果为a集合除去a集合与b集合的交集。上述4种运算的示例代码如下：

```
var set3:Set<Int> = [1,2,3,4]
var set4:Set<Int> = [1,2,5,6]
//返回交集 {1，2}
var setInter = set3.intersection(set4)
//返回交集的补集{3，4，5，6}
var setEx = set3.symmetricDifference(set4)
//返回并集{1，2，3，4，5，6}
var setUni = set3.union(set4)
//返回第二个集合的补集{3，4}
var setSub = set3.subtracting(set4)
```

使用比较运算符“==”可以比较两个集合是否相等，当两个集合中的所有元素都相等时，两个集合才相等。集合中还提供了一些方法用于判断集合间的关系，示例代码如下：

```
var set5:Set = [1,2]
var set6:Set = [2,3]
var set7:Set = [1,2,3]
var set8:Set = [1,2,3]
//判断是否是某个集合的子集，set5 是 set7 的子集，返回 ture
set5.isSubset(of: set7)
//判断是否是某个集合的超集，set7 是 set5 的超集，返回 ture
set7.isSuperset(of: set5)
//判断是否是某个集合的真子集，set5 是 set7 的真子集，返回 ture
set5.isStrictSubset(of: set7)
//判断是否是某个集合的真超集，set7 不是 set8 的真超集，返回 false
set7.isStrictSuperset(of: set8)
```

与数组类似，集合也可以通过for-in遍历的方式来获取所有集合中的数据，可以通过3种方法来进行遍历：遍历元素、遍历集合的枚举与遍历集合的下标。集合枚举会返回一个元组，元组中将集合下标和其对应的值一同返回，示例代码如下：

```
//遍历元素
for item in set7 {
    print(item)
}
//遍历集合的枚举
for item in set7.enumerated() {
    print(item)
}
//遍历集合的下标
for index in set7.indices {
    print(set7[index])
}
```

集合虽然不强调元素顺序，但是在遍历时，开发者可以对其进行排序后再遍历，示例如下：

```
//从大到小排序再遍历集合
for item  in set7.sorted(by: >) {
    print(item)
}
```

3.4.3　字典（Dictionary）类型

字典是生活中常用的学习工具，字典在使用时是由一个索引找到一个结果。例如，英汉词典通过英文单词可以找到其对应的汉语解释，成语词典通过成语可以找到其对应的意义解释，等等。这种数据的存储模式被称为键值映射模式，即通过一个确定的键可以找到一个确定的值。类比上面的例子，在英汉词典中，英文单词就是键，汉语释义就是值；在成语词典中，成语就是键，意义解释就是值。在Swift语言中也有这样的一种Dictionary集合，即字典集合类型。

Swift中的任何类型在声明时都必须明确其类型，通过对Array和Set的学习，读者应该知道，对于集合类型，在声明时务必明确其内部元素的类型，字典也不例外，由于字典中的一个元素实际上是由键和值两部分组成的，因此在声明字典时，也需要明确其键和值的类型。有两种方式可以进行

字典的声明或创建，示例代码如下：

```
//声明字典[param1:param2]，这种结构用于表示字典类型，param1 为键类型，param2 为值类型
var dic1:[Int:String]
//这种方式和[:]效果一样，dic2 与 dic1 为相同的类型
var dic2:Dictionary<Int,String>
//字典创建与赋值
dic1 = [1:"1",2:"2",3:"3"]
dic2 = Dictionary(dictionaryLiteral: (1,"1"),(2,"2"),(3,"3"))
//在创建字典时，也可以不显式声明字典的类型，可以通过赋初值的方式来使编译器自动推断
var dic3 = ["1":"one"]
//创建空字典
var dic4:[Int:Int] = [:]
var dic5:Dictionary<Int,Int> = Dictionary()
```

需要注意，字典通过键来找到特定的值，在字典中值可以重复，但是键必须唯一。这样才能保证一个确定的键能找到一个确定的值，并且如果开发者在字典中创建重复的键，编译器也会报出错误。

字典类型也支持使用isEmpty与count来判断是否为空并获取元素个数，示例代码如下：

```
//获取字典中的元素个数
dic1.count
//判断字典是否为空
if dic4.isEmpty{
   print("字典为空")
}
```

通过具体键可以获取与修改对应的值，示例如下：

```
//通过键操作值
//获取值
dic1[2]
//修改值
dic1[1]="0"
//添加一对新的键值
dic1[4] = "4"
```

上面代码中的dic1[1]="0"与dic1[4]="4"实际上完成了相同的操作，可以这样理解：在对某个键进行赋值时，如果这个键存在，则会进行值的更新，如果这个键不存在，则会添加一对新的键值。然而在开发中，很多情况下需要对一个存在的键进行更新操作，如果这个键不存在，则不添加新键值对，要实现这种效果，可以使用字典的更新键值方法，示例代码如下：

```
//对键值进行更新
dic1.updateValue("1", forKey: 1)
```

updateValue(value:forkey:)方法用于更新一个已经存在的键值对，其中第1个参数为新值，第2个参数为要更新的键。这个方法在执行时会返回一个Optional类型的值，如果字典中此键存在，则会更新成功，并将键的旧值包装成Optional值返回，如果此键不存在，则会返回nil。在开发中，常常使用if-let结构来处理，示例如下：

```
//使用 if let 处理 updateValue 的返回值
```

```
if let oldValue = dic1.updateValue("One", forKey: 1) {
   print("Old Value is \(oldValue)")
}
```

其实在通过键来获取字典中的值时，也会返回一个Optional类型的值，如果键不存在，则此Optional值为nil，因此也可以使用if-let结构来保证程序的安全性，示例如下：

```
//通过键获取的数据也将返回 Optional 类型的值，也可以使用 if let
if let value = dic2[1] {
   print("The Value is \(value)")
}
```

下面的方法可以实现对字典中键值对的删除操作：

```
//通过键删除某个键值对
dic1.removeValue(forKey: 1)
//删除所有键值对
dic1.removeAll()
```

在对字典进行遍历操作时，可以遍历字典中所有键组成的集合，也可以遍历字典中所有值组成的集合，通过字典实例的keys属性与values属性分别可以获取字典的所有键与所有值，示例代码如下：

```
//通过键来遍历字典
for item in dic2.keys {
   print(item)
}
//通过值来遍历字典
for item in dic2.values {
   print(item)
}
//直接遍历字典
for item in dic2 {
   print(item)
}
for (key,value) in dic2 {
   print("\(key):\(value)")
}
```

如以上代码所示，也可以直接对字典实例进行遍历，遍历中会返回一个元组类型包装字典的键和值。

在进行字典键或者值的遍历时，也支持对其进行排序遍历，实例如下：

```
for item in dic2.keys.dic2.keys.sorted(by: >) {
   print(dic2[item]!)
}
```

3.5 练习及解析

（1）分别创建字符串变量China和MyLove，将这两个变量拼接成为一句话并且对拼接后的新字符串变量进行遍历，并检查其中是否有L字符，有则进行打印操作。

解析：

```
var str1 = "China"
var str2 = String("MyLove")
var str3 = str1+str2!
for chara in str3 {
    if chara == "L" {
        print(chara)
    }
}
```

（2）删除下面字符串中的所有“!”和“?”符号。

```
swsvr!vrfe?123321!!你好!世界?
```

解析：

```
var stringOri2 =  "swsvr!vrfe?123321!!你好?世界!"
//创建一个空字符串用于进行接收
var stringRes2 = String()
for index in stringOri2.indices {
    if stringOri2[index] != "?" && stringOri2[index] != "!" {
        stringRes2.append(stringOri2[index])
    }
}
```

（3）将字符串abcdefg进行倒序排列，并打印。

解析：

```
var stringOri3 = "abcdefg"
var index3 = stringOri3.endIndex
var stringRes3 = String()
while index3>stringOri3.startIndex {
    index3 = stringOri3.index(before: index3)
    stringRes3.append(stringOri3[index3])
}
print(stringRes3)
```

（4）将“*”符号逐个插入下面字符串的字符中间，并打印。

```
我爱你中国
```

解析：

```
var stringOri4 = "我爱你中国"
```

```
var stringRes4 = String()
for index in stringOri4.indices {
   stringRes4.append(stringOri4[index])
   if index<stringOri4.index(before: stringOri4.endIndex) {
      stringRes4.append(Character("*"))
   }
}
print(stringRes4)
```

（5）将下面字符串中所有的abc替换成Hello，并打印。

```
abc 中国 abc 美国 abc 英国~德国 abc 法国 abc
```

解析：

```
var stringOri5 = "abc 中国 abc 美国 abc 英国~德国 abc 法国 abc"
var range5 = stringOri5.range(of:"abc")
while range5 != nil {
   stringOri5.replaceSubrange(range5!, with: "Hello")
   range5 = stringOri5.range(of:"abc")
}
print(stringOri5)
```

（6）进行正负号翻转，并打印。

①将 -123 转换为+123。

②将 +456 转换为-456。

解析：

```
var stringOri6 = "-123"
var stringOri_6 = "+456"
if stringOri6.hasPrefix("-"){
   stringOri6.replaceSubrange(stringOri6.startIndex..<stringOri6.index
(after: startIndex), with: "+")
}
if stringOri_6.hasPrefix("+"){
   stringOri_6.replaceSubrange(stringOri_6.startIndex..<stringOri_6.index
(after: startIndex), with: "-")
}
print(stringOri6,stringOri_6)
```

（7）将下列数组中的0去掉，返回新的数组，并打印输出。

```
[1,13,45,5,0,0,16,6,0,25,4,17,6,7,0,15]
```

解析：

```
var arrayOri1 = [1,13,45,5,0,0,16,6,0,25,4,17,6,7,0,15]
var arrayRes1 = Array<Int>()
for index in arrayOri1.indices {
   if arrayOri1[index] == 0 {
     continue
   }
   arrayRes1.append(arrayOri1[index])
}
```

```
print(arrayRes1)
```

（8）定义一个包含10个元素的数组，对其进行赋值，使每个元素的值等于其下标，然后输出，最后将数组倒置后输出。

解析：

```
var arrayOri2 = Array<Int>()
for index in 0...9{
   arrayOri2.append(index)
}
print(arrayOri2)
//进行倒置排序
arrayOri2.sort(by: { (a, b) -> Bool in
   return a>b
})
print(arrayOri2)
```

（9）工程测量到两组数据，分别为2、4、3、5与3、4、7、1。对两组数据进行整合，使其合成一组数据，重复的数据只算一次，使用代码描述此过程。

解析：

```
var setOri3:Set<Int> = [2,4,3,5]
var setOri32:Set<Int> = [3,4,7,1]
var setRes3 = setOri3.union(setOri32)
```

（10）期末考试中，王晓成绩为98，邹明成绩为86，李小伟成绩为93，用字典结构来对三人的成绩进行存储，并以从高到低的排序输出。

解析：

```
var dicOri4 = ["王晓":98,"邹明":86,"李小伟":93]
for item in dicOri4.sorted(by: { (student1, student2) -> Bool in
   return student1.value > student2.value
}){
   print(item)
}
```

本题在解析时使用到了排序闭包，后面章节会对闭包的语法进行详细讲解。

3.6 模拟面试

（1）简述Array、Set和Dictionary的异同点，并说明各自的应用场景。

回答要点提示：

①首先Array、Set和Dictionary都是Swift的集合类型，所谓集合类型，是指一组数据集合，Swift是一种强类型语言，集合中的元素必须保持一致。

②Array和Set最大的区别是Array有序，Set无序。由于Array的有序性，因此在存储时，Array中的每一个元素都会被分配一个下标，我们可以通过下标来获取具体位置的数据，因此Array的存

储灵活性和查询速度相比Set会略差。如果在开发中，我们需要的仅仅是一个数据池，并不特别在意数据的顺序，可以选择Set类型，否则可以选择Array类型。

③Array和Dictionary最大的区别在于Array是通过递增的整数索引来关联元素的，而Dictionary则是使用任意数据类型作为索引来关联元素的。Dictionary要比Array更加灵活，同样其对“顺序”的描述能力没有Array强。

核心理解内容：

理解Swift语言中3种常用的集合类型的特点，熟练使用Array、Set和Dictionary的相关操作方法，牢记集合对象中的数据类型必须保持一致。

（2）开发中的字符串解析是指什么，有什么用？

回答要点提示：

①字符串解析是指使用相关函数对字符串进行处理，比如截取、拼接、替换、部分删除、分解等。在Swift中提供了丰富的原生函数来对字符串进行处理。

②字符串解析在实际开发过程中应用非常广，比如音乐类软件对歌词（LRC）文件的解析，实际上就是使用字符串解析技术来从LRC歌词文件中解析出歌曲名称、歌手名、时间等信息。字符串解析技术也常常可以用来进行文本的格式整理，比如去掉多余的空格和换行符等。

③关于字符串解析，重中之重是JSON数据处理，在移动端，几乎所有和网络相关的数据交换都是采用JSON数据格式，JSON解析就是一种基础的字符串解析技术。

核心理解内容：

字符串解析实际上就是对字符串进行处理，通俗一点，即对字符串进行增（拼接、插入）、删（截取、移除）、改（替换）、查（检索）等操作。学习Swift语言必须要熟练掌握String类型中封装的相关函数，多写多练。

第 4 章

基本运算符与程序流程控制

上帝创造了整数，所有其余的数都是人造的。

——利奥波德·克罗内克

世界上所有的运算无外乎都是由计算过程与结果两部分组成的，无论这个结果是否符合预期目标。在编程中，运算由表达式表示，而量值和运算符共同构成了表达式。Swift语言对运算符的支持可谓强大，其除了支持C语言与Objective-C语言中常用的运算符之外，还提供了一些十分有特点的运算符，例如空合并运算符、区间运算符等。除此之外，Swift语言还支持对运算符进行重载与自定义操作，开发者可以根据自己的需要为系统的运算符提供新的运算方法，甚至自定义自己的运算符。

程序存在的意义就是帮助人们实现解题思路和进行重复性的计算，然而任何复杂问题的解决过程都不是从上到下线性完成的，对程序流程的控制能力是编程语言强大的关键所在。Swift语言中提供了强大的程序流程控制语句，无论是循环结构、选择结构还是跳转结构，开发者都可以十分方便地运用，并且Swift语言的语句设计也更加简洁与优美。通过本章的学习，读者将会更深刻地体会到这一点。

通过本章，你将学习到：

- 各种运算符的应用。
- 运算符的优先级与结合性。
- 使用for-in结构进行循环遍历。
- 使用while与repeat-while结构进行条件循环。
- 使用if与if-else结构进行选择判断。
- 使用switch-case结构进行多分支选择。
- 使用跳转语句灵活控制程序流程。

4.1　初识运算符

运算符的作用在于将量值或者表达式结合在一起进行计算。Swift语言中的运算符按照其操作数的个数可以分为3类，分别为：

- 一元运算符：一元运算符作用于一个操作数，其可以出现在操作数的前面，例如正负运算符“+”“-”，逻辑非运算符“!”。
- 二元运算符：二元运算符作用于两个操作数之间，例如加减运算符“+”“-”等。
- 三元运算符：三元运算符作用于三个操作数之间，经典的三元运算符有问号、冒号运算符，其可以方便地实现简单的判断选择结构。

4.1.1　赋值运算符

赋值运算符应该是在编程中出现频率最高的运算符之一，在对任何量值进行赋值时，都需要使用到赋值运算符“=”。需要注意，“=”在Swift语言中是赋值运算符，并不是相等运算符，对于一些编程初学者，很容易将相等运算符“==”与赋值运算符“=”混淆使用。使用Xcode开发工具创建一个名为Operational的Playground文件，赋值运算符示例如下：

```
//字符串赋值
var str = "Hello, playground"
//整型赋值
var count = 5
//元组赋值
var group = (1,2,"12")
//Bool 赋值
var bol = true
```

如果要将一个元组中的值依次赋给指定的量值，Swift中还支持使用解构赋值的语法来进行赋值，示例如下：

```
// 进行解构赋值
var (n1, n2, n3) = (1, 2, 3)
print(n1, n2 ,n3) // 1, 2, 3
```

提　示

赋值运算符用于值的传递，其结果是量值被赋了具体的值。相等运算符则用于比较操作，其会返回一个 Bool 类型的逻辑值。

4.1.2　基本算术运算符

基本算术运算符用于进行一些基本的数学运算，例如加、减、乘、除等。需要注意，Swift语

言从2.2版本之后，删除了自增运算符“++”与自减运算符“--”，目前版本的Swift语言中不可以使用这两个运算符。Swift中支持的基本算术运算符示例如下：

```
//相加运算
1+2
//相减运算
2-1
//相乘运算
2*2
//相除运算
4/2
//取余运算
4%3
```

提　示

取余运算符必须在整数间进行运算时使用。

将赋值运算符与基本算术运算符结合使用可以组合成复合赋值运算符，复合赋值运算符可以在一个表达式中完成一项基本运算与赋值的复合操作，示例如下：

```
var tmp=1
//加赋值复合运算
tmp+=3  //tmp = tmp +3
//减赋值复合运算
tmp-=3  //tmp = tmp -3
//乘赋值复合运算
tmp*=3  //tmp = tmp *3
//除赋值复合运算
tmp/=3  //tmp = tmp /3
//取余赋值复合运算
tmp%=3  //tmp = tmp %3
```

除了上面提到的运算符外，正负运算符和数学中的正负号作用类似，负运算符会改变数据的正负性，正运算符会保持数据的正负性。示例如下：

```
var a = 1
var b = -2
+b //-2
-a //-1
```

提　示

自增与自减运算符在 Swift 2.2 及之前版本可以使用，Swift 2.2 版本后，基于代码的可读性与减少歧义的考虑，移除了这两个运算符。

4.1.3　基本逻辑运算符

基本算术运算符进行数学上的算术操作，基本逻辑运算符进行逻辑运算操作。可以简单理解

为，逻辑运算就是生活中所定义的真与假。系统定义的基本逻辑运算符会返回一个Bool类型的逻辑值，因此，基本逻辑运算符组成的逻辑表达式在if判断语句中经常会用到。

Swift中支持的基本逻辑运算符有：逻辑与运算符“&&”、逻辑或运算符“||”、逻辑非运算符“!”三种，逻辑运算只在逻辑值（Bool类型值）之间进行，与、或、非三种运算中前两者为二元运算符，需要有两个Bool类型的操作数，非运算符为一元运算符，需要有一个Bool类型的操作数。这三种运算符有如下特点：

- 与：两个操作数都为真，结果才为真，有一个操作数为假，则结果为假。
- 或：两个操作数有一个为真，则结果为真，两个操作数都为假，则结果为假。
- 非：操作数为真，则结果为假，操作数为假，则结果为真。

示例如下：

```
var p1 = true
var p2 = false
//与运算 false
p1&&p2
//或运算 true
p1||p2
//非运算 false
!p1
```

提　示

与 Objective-C 语言不同，Swift 语言中逻辑运算的操作数必须为严格的 Bool 类型。

4.1.4　比较运算符

Swift中的比较运算符用于两个操作数之间的比较运算，其会返回一个Bool类型的逻辑值。基本的比较运算符有：等于比较运算符“==”、小于比较运算符“<”、大于比较运算符“>”、不等于比较运算符“!=”、小于等于比较运算符“<=”以及大于等于比较运算符“>=”。示例如下：

```
1==2 //等于比较，返回 false
1<2  //小于比较，返回 true
1>2  //大于比较，返回 false
1 != 2 //不等于比较，返回 true
1<=2  //小于等于比较，返回 true
1>=2  //大于等于比较，返回 false
```

Swift中对于元组的比较操作读者需要注意：首先要比较的元组中的元素个数和对应位置的元素类型必须相同，其次元组中每一个元素必须支持比较运算操作，示例代码如下：

```
var tp1 = (3,4,"5")
var tp2 = (2,6,"9")
var tp3 = ("1",4,5)
tp1<tp2           //将返回 false
```

上面的代码中，元组实例tp1与元组实例tp2中的元素个数和对应类型相同，且所有元素都支持

比较运算操作，所以tp1与tp2可以进行比较运算。tp1与tp3虽然元素个数相同，但是tp1的第1个元素为整型，tp3的第一个元素为字符串类型，类型不同，所以不能进行比较运算。Swift在进行元组间的比较运算的时候，会遵守这样一个原则：从第1个元素开始比较，如果比较出了结果，则不再进行后面元素的比较运算，直接返回结果Bool值，如果没有比较出结果，那么继续依次比较后面的元素，直到比较出结果为止。

4.1.5 条件运算符

条件运算符（三目运算符）是一种三元运算符，其可以简便实现代码中的条件选择逻辑。例如如下代码为一个简单的条件选择语句示例：

```
var m = 3
var n = 6
if m>n {
   print("m>n")
}else{
   print("m<=n")
}
```

上面的代码对变量m和n进行了比较，并将比较结果进行打印，如果使用条件运算符，上面的逻辑可以简写成如下模样：

```
print(m>n ? "m>n":"m<=n")
```

上面的代码在语法上可以简化为这样的格式：条件?成立的代码:不成立的代码。

可以看到，条件运算符（三目运算符）只使用了一句代码就完成了if-else结构需要多句代码才能完成的工作，其编写效率很高。我们来分解一下条件运算符的组成结构，首先条件运算符需要有3个操作数，其中第1个操作数必须为一个条件语句或者一个Bool类型的值，第2个和第3个操作数可以是任意类型的值或者一个有确定值的表达式，3个操作数由问号“？”和冒号“：”进行分割，当问号前的操作数值为真时，条件运算符运算的结果为冒号前的操作数的值，当问号前的操作数值为假时，条件运算符运算的结果为冒号后的操作数的值。

4.2 Swift 语言中两种特殊的运算符

前边介绍过，Optional可选（值）类型是Swift语言的一大特点，Swift语言中的空合并运算符也是专门为Optional值类型所设计的。除了空合并运算符外，Swift语言中的区间运算符也十分强大易用，熟练使用这些运算符将极大地提高开发效率。

4.2.1 空合并运算符

可选值类型是Swift语言的一个独特之处，空合并运算符就是针对可选值类型而设计的运算符。

首先来看一段示例代码：

```
var q:Int? = 8
var value:Int
if q != nil {
   value = q!
}else{
   value = 0
}
```

上面的示例就是一个简单的if-else的选择结构，利用4.1.5小节介绍的条件运算符（三目运算符）可以将上面的代码简写如下：

```
var q:Int? = 8
var value:Int
value = (q != nil) ? (q!) : 0
```

使用条件运算符改写后的代码简单很多，Swift语言中还提供了空合并运算符来更加简洁地处理这种Optional类型值的条件选择结构，空合并运算符由“??”表示，上面的代码可以改写成如下形式：

```
//空合并运算符
var q:Int? = 8
var value:Int
value = q ?? 0
```

空合并运算符“??”是一个二元运算符，使用空合并运算符改写后的代码更加简洁。其需要两个操作数，第一个操作数必须为一个Optional值，如果此Optional值不为nil，则将其进行拆包操作，并作为空合并运算的运算结果。如果此Optional值为nil，则会将第二个操作数作为空合并操作运算的结果返回。使用空合并操作符来处理有关Optional值的选择逻辑将十分方便。

4.2.2　区间运算符

在C语言中，关于范围概念的描述常常会使用一个表达式来表示，例如：

```
//表示大于 0 小于 10 的范围
index>0 && index<10
```

在Objective-C语言中提供了NSRange这样一个结构体来描述范围，虽然直观了许多，但开发者在使用时需要构造NSRange实例，使用起来就略显烦琐。Swift中除了支持Range结构体来描述范围外，还提供了一个区间运算符来快捷直观地表示范围区间。示例如下：

```
//创建范围 >=0 且<=10 的闭区间
var range1 = 0...10
//创建范围>=0 且<10 的半开区间
var range2 = 0..<10
```

也可以通过“~=”运算符来检查某个数字是否包含于范围中，示例如下：

```
//8 是否在 range1 中
```

```
print(range1 ~= 8)  //输出 true
```

区间运算符常见于for-in循环结构中，开发者常常会使用区间运算符来定义循环次数，示例如下：

```
//a...b 为闭区间写法
for index in 0...3 {
    print(index)
}
//a..<b 为左闭右开区间
for index in 0..<3 {
    print(index)
}
```

提　示

在 for-in 循环结构中，如果 in 关键字后面是一个集合，则变量 index 会自动获取集合中的元素；如果 in 关键字后面是一个范围，则 index 获取到的是从左向右依次遍历到的范围索引数。

4.3　循环结构

在程序编写中，常常会有大量而重复的操作，这也是计算机计算的优势所在，对于开发者来说，循环结构就是为执行大量而重复的代码块诞生的。Swift语言主要提供了for-in遍历、while与repeat-while条件循环3种循环结构。

4.3.1　for-in 循环结构

读者对于for-in结构并不陌生，在前面章节中介绍的很多内容都使用了for-in结构进行演示。如果读者了解C/Objective-C语言，这里就要注意了，在C/Objective-C语言中也支持for-in循环结构，但是其被称为快速遍历，用它进行的循环操作是无序的。Swift语言中的for-in结构则强大很多，其可以进行无序的循环遍历，也可以进行有序的循环遍历。

在Swift 2.2及以上版本中，在循环结构中做的最大更改就是删除了经典的for()循环结构，许多C/Objective-C语言的开发者可能会不太习惯，事实上，Swift语言中的for-in结构已经完全可以代替曾经的for()循环，并且比for()循环的实现更加简洁优美，同for()循环一同移除的还有“++”和“--”运算符。

使用Xcode开发工具创建一个名为ControlFlow的Playground，进行循环代码的测试。要使用for-in结构进行有序的循环遍历，需要配合区间运算符，并且指定一个循环次数变量，示例如下：

```
//将打印 1，2，3，4，5
for index in 1...5 {
    print(index)
}
```

for-in结构中需要两个参数，第2个参数可以是一个集合类型的实例，也可以是一个范围区间，第1个参数为捕获参数，每次从第2个参数中遍历出的元素便会赋值给它，开发者在循环结构中可以直接使用。

在进行for-in循环遍历的时候，开发者并不需要捕获到遍历出的值，可以使用匿名参数来接收，Swift中使用“_”符号来表示匿名参数，示例如下：

```
//如果不需要获取循环中的循环次序，可以使用如下方式
var sum=0;
for _ in 1...3 {
    sum += 1
}
```

对集合的遍历是for-in循环常用的场景之一，这些在前面讲解集合类型的章节中已经详细介绍过了，简单的示例代码如下：

```
//遍历集合类型
var collection1:Array = [1,2,3,4]
var collection2:Dictionary = [1:1,2:2,3:4,4:4]
var collection3:Set = [1,2,3,4]
for obj in collection1 {
    print(obj)
}
for (key , value) in collection2 {
    print(key,value)
}
for obj in collection3 {
    print(obj)
}
```

4.3.2　while 与 repeat-while 条件循环结构

while与repeat-while结构在C/Objective-C语言中也支持，并且功能基本一致，只是Swift语言将do-while结构修改为repeat-while。

在开发中，经常有条件循环的需求，例如模拟水池蓄水的过程，每次蓄水1/10，当蓄满水后停止蓄水。while循环结构可以十分方便地创建这类循环代码，示例如下：

```
var i=0
//当 i 不小于 10 时跳出循环
while i<10 {
    print("while",i)
    //进行 i 的自增加
    i+=1
}
```

在while循环结构中，while关键字后面需要填写一个逻辑值或者以逻辑值为结果的表达式作为循环条件，如果逻辑值为真，则程序会进入while循环体。执行完循环体的代码后进行循环条件的判断，如果循环条件依然为真，则会再次进入循环体，否则循环结束。由于while循环是根据循环

条件来判断是否进入循环体的，如果循环条件一直成立，则会无限循环，因此开发者在使用while循环的时候，注意要在循环体中对循环条件进行修改，且修改的结果是循环条件不成立，否则会出现死循环。

上面演示的while结构会先进行条件判断，再进行循环体的执行。repeat-while结构则会先执行一次循环体，再进行循环条件的判断，图4-1中列出了两种结构的异同。

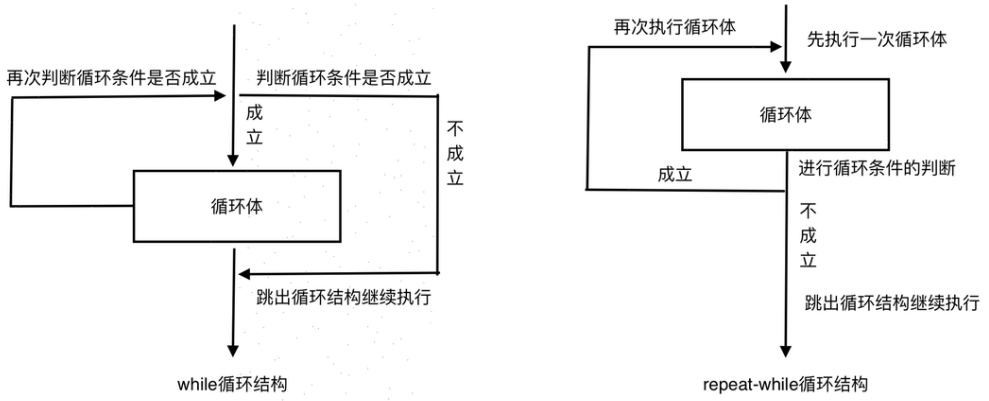

图 4-1　while 结构与 repeat-while 结构的异同

repeat-while循环结构示例代码如下：

```
var j=0
//先执行一次循环体，再判断循环条件是否成立
repeat {
    print("repeat while")
    j+=1
} while j<10
```

4.4　条件选择与多分支选择结构

在开发中有两种重要的流程结构，即循环结构与分支结构，循环结构用于处理大量的重复操作，分支结构则用于处理由条件差异而产生的代码分支路径。Swift语言中提供的分支结构有if结构、if-else结构与switch-case结构。

4.4.1　if 与 if-else 条件选择结构

if与if-else结构语句是Swift语言中基础的分支结构语句，开发者可以使用单个if语句进行单条件分支，也可以使用if与else组合来实现多条件分支，示例如下：

```
var c = 10
//进行 if 条件判断
```

```
if c<10 {
    print(c)
}
//进行 if-else 组合
if c>10 {
    c-=10
    print(c)
}else{
    print(c)
}
//进行 if-else 多分支组合
if c>0&&c<10 {
    print(c)
}else if c<=0 {
    c = -c
    print(c)
}else if c<=10&&c<20{
    c-=10
    print(c)
}else{
    print("bigger")
}
```

提　示

（1）需要注意，if 关键字后面跟的条件必须为严格意义上的逻辑值或者结果为逻辑值的表达式。这点 Swift 语言和 C/Objective-C 语言有一定差异。

（2）if-else 组合结构每个分支是互斥的，只能有一个分支的代码被执行，条件的判断顺序会从上到下进行，直到找到一个判断条件为真的分支或者最后一个 else 语句。同时，开发者也可以不加单独的 else 语句，这种情况下，如果没有条件成立的分支，则任何分支都不会被执行，程序会继续向后执行。

4.4.2　switch-case 多分支选择结构

switch语句也被称为开关选择语句，它通过匹配的方式来选择要执行的代码块，Swift语言中的switch语句更加强大，不像C/Objective-C中的switch语句只能进行int类型值的匹配，Swift语言中的switch语句可以进行任意数据类型的匹配，并且case子句的语法和扩展都更加灵活。使用switch结构进行字符分支匹配的示例如下：

```
//使用 switch 语句进行字符分支匹配
switch charac {
case "a":
    print("chara is a")
case "b":
    print("chara is b")
case "c":
    print("chara is c")
```

```
default ://default 用于处理其他额外情况
   print("no charac")
}
```

如以上代码所示，switch关键字后面需要填写要进行分支匹配的元素，在switch结构中通过子句case的列举进行元素值的匹配，匹配成功后，会执行相应case子句中的代码，如以上代码将打印字符串“chara is b”。default为switch语句中的默认匹配语句，即如果前面所有的case子句都没有匹配成功，则会执行default中的代码，开发者也可以将default子句省略，这时如果所有case子句都没有匹配上，则会跳过switch结构，直接执行后面的代码。

还有一点读者需要注意，在C/Objective-C语言中，case语句不会因匹配成功而中断，如果不进行手动控制，switch结构中的case子句会依次进行匹配执行。举一个例子，如果第1个case子句匹配成功，第2个case子句也匹配成功，则第1个case子句和第2个case子句中的代码都会执行，因此C/Objective-C程序开发中，开发者一般会在每个case子句后面添加break关键字进行手动中断。Swift语句优化了这一点，一个case语句匹配成功后，会自动跳出switch结构，如果不进行特殊处理，switch结构中的分支只会被执行一个或者一个也不执行。

switch-case结构也支持开发者在一个case子句中编写多个匹配条件，程序在执行到这个case子句时，只要有一个条件匹配成功，就会执行此case下的代码，示例如下：

```
//同一个 case 中可以包含多个分支
switch charac {
case "a","b","c" :
   print("chara is word")
case "1","2","3" :
   print("chara is num")
default :
   print("no charac")
}
```

case子句的匹配条件也可以是一个区间范围，当要匹配的参数在这个区间范围内时，就会执行此case下的代码，示例如下：

```
//在 case 中也可以使用一个范围
var num = 3
switch num {
case 1...3 :
   print("1<=num<=3")
case 4 :
   print("chara is num")
default :
   print("no charac")
}
```

从上面的示例中可以了解，Swift语言中的switch-case结构十分灵活强大，如果将switch-case结构和元组结合使用，开发者在编写代码时将更加灵活多变。首先，对于元组类型参数的匹配，case子句可以进行选择匹配和优化匹配，示例如下：

```
//使用 Switch 语句进行元组的匹配
var tuple = (0,0)
```

```
switch tuple {
    //进行完全匹配
case (0,1):
    print("Sure")
    //进行选择性匹配
case (_,1):
    print("Sim")
    //进行元组元素的范围匹配
case(0...3,0...3):
    print("SIM")
default:
    print("")
}
```

如以上代码所示，在进行元组的匹配时，有3种方式可以选择：第1种方式是完全匹配，即元组中所有元素都必须完全相等，才算匹配成功；第2种方式是选择匹配，即开发者只需要指定元组中的一些元素进行匹配，不需要关心的元素可以使用匿名参数标识符来代替，这种方式下，只要指定的参数都相等，就算匹配成功；第三种方式是范围匹配，即相应位置指定的范围包含需匹配元组相应位置的值，就算匹配成功。其中第2种匹配方式可以和第3种匹配方式组合使用。

Swift语言中的case子句中还可以捕获switch元组的参数，在相应的case代码块中可以直接使用捕获到的参数，这在开发中可以简化所编写的代码，示例如下：

```
var tuple = (0,0)
//进行数据绑定
switch tuple {
    //对元组中的第一个元素进行捕获
case (let a,1):
    print(a)
case (let b,0):
    print(b)
    //捕获元组中的两个元素，let(a,b) 与 (let a,let b)意义相同
case let(a,b):
    print(a,b)
default:
    print("")
}
```

这里读者需要注意，要捕获的元素并不能起到匹配的作用，例如元组tuple中有两个元素，如果case条件为(let a,1)，则在进行匹配时会匹配tuple中第2个参数，如果匹配成功，则会将tuple元组的第1个参数的值传递给a常量，并且执行此case中的代码块，在这个代码块中，开发者可以直接使用常量a。因此，要捕获的元素在匹配时实际上充当着匿名标识符的作用，如以上代码中的第3个case子句，其条件为let(a,b)，实际上这个条件始终会被匹配成功。并且，如果开发者对元组中的所有元素都进行了捕获，在代码表现上，可以写作(let a,let b)，也可以直接捕获整个元组，写作let(a,b)，这两种方式只是写法上有差异，在使用时并无差别。

switch-case结构的参数捕获语法在使用起来为开发者带来了不少的便利。然而其也有一个问题，它将所有要捕获的元素都作为匿名参数来进行匹配，有时候并不是开发者想要的结果。例如上面的tuple元组，开发者需要捕获元组中的第1个元素，同时又需要与第1个元素相关的条件成立时再

使case子句匹配成功，针对这种情况，Swift中也提供了相应的办法来处理，可通过在case语句中追加where条件的方式来实现上述需求，示例如下：

```
//对于进行了数据捕获的 Switch-case 结构，可以使用 where 关键字来进行条件判断
switch tuple {
case (let a,1):
   print(a)
   //当元组中的两个元素都等于 0 时才匹配成功，并且捕获第一个元素的值
case (let b,0) where b==0:
   print(b)
//当元组中的两个元素相同时，才会进入下面的 case
case let(a,b) where a==b:
   print(a,b)
default:
   print("")
}
```

4.5 Swift 语言中的流程跳转语句

跳转语句可以提前中断循环结构，也可以人为控制选择结构的跳转，使代码的执行更加灵活多变。Swift中提供了大量的流程跳转语句供开发者使用，熟悉这些语句的结构与特点可以使开发效率大大提高。Swift中提供的流程跳转语句主要有continue、break、fallthrough、return、throw、guard。

continue语句用于循环结构中，其作用是跳过本次循环，直接开始下次循环。这里需要注意，continue的作用并不是跳出循环结构，而是跳过本次循环，直接执行下一个循环周期，示例如下：

```
for index in 0...9 {
   if  index == 6 {
      continue
   }
   print("第\(index)次循环")
}
```

上面的示例代码将跳过index等于6时的代码块，在打印信息中会缺少index等于6时的打印输出。需要注意的是，continue语句默认的操作范围直接包含它的这一层循环结构，如果代码中嵌套了多层循环结构，continue语句会跳过本次循环。那么，如果想要实现不跳过本次循环，而是直接跳至开发者指定的那一层循环结构，该如何写呢？示例如下：

```
MyLabel:for indexI in 0...2 {
   for indexJ in 0...2 {
      if indexI == 1 {
         continue MyLabel
      }
      print("第\(indexI)\(indexJ)次循环")
   }
}
```

以上代码创建了两层循环结构，在内层循环中使用了continue语句进行跳转，MyLabel是外层循环的标签，因此这里的continue跳转将会跳出indexI等于1时的外层循环，直接开始indexI等于2的循环操作。

break语句是中断语句，其也可以用于循环结构中，和continue语句不同的是，break语句会直接中断直接包含它的循环结构，即当循环结构为一层时，如果循环并没有执行完成，则后面所有的循环都将被跳过。如果有多层循环结构，程序会直接中断直接包含它的循环结构，继续执行该循环结构外层的循环结构，示例如下：

```
for index in 0...9 {
    if  index == 6 {
        break
    }
    print("第\(index)次循环")
}
```

上面的代码在index等于6时使用了break语句进行中断，第5次循环后的所有打印信息都将被跳过。break语句默认将中断直接包含它的循环结构，同样也可以使用指定标签的方式来中断指定的循环结构，示例如下：

```
MyLabel:for indexI in 0...2 {
    for indexJ in 0...2 {
        if indexI == 1 {
            break MyLabel
        }
        print("第\(indexI)\(indexJ)次循环")
    }
}
```

break语句也可以用于switch结构中。在switch结构中，break语句将直接中断后面所有的匹配过程，直接跳出switch结构。在Swift语言中，switch-case选择匹配结构默认就是break操作，故开发者不必手动添加break代码。

fallthrough语句是Swift中特有的一种流程控制语句，前面提到过，当Swift语言中的switch-case结构匹配到一个case后，会自动中断后面所有case的匹配操作，如果在实际开发中需要switch-case结构不自动进行中断操作，可以使用fallthrough语句，示例如下：

```
var tuple = (0,0)
switch tuple {
case (0,0):
    print("Sure")
    //fallthrough 会继续执行下面的 case
    fallthrough
case (_,0):
    print("Sim")
    fallthrough
case(0...3,0...3):
    print("SIM")
default:
    print("")
}
```

以上示例代码将会打印Sure、Sim和SIM。

return语句对于读者来说应该十分熟悉，其在函数中用于返回结果值，也可以用于提前结束无返回值类型的函数。当然，return语句的应用场景不只局限于函数中，在闭包中也可以使用return进行返回。函数的相关知识会在后面的章节详细介绍，这里只做简单演示，示例如下：

```
//有返回值函数的返回
func myFunc()->Int{
    return 0
}
//无返回值函数的返回
func myFunc(){
    return
}
```

throw语句用于异常的抛出，throw语句抛出的异常如果不进行捕获处理，也会使程序中断。Swift语言中有抛出异常和处理异常的代码结构，在后面的章节中会详细介绍，这里只做简单演示。在函数中抛出异常的示例代码如下：

```
//定义异常类型
enum MyError:Error{
    case errorOne
    case errorTwo
}
func  newFunc() throws{
    //抛出异常
    throw MyError.errorOne
}
```

guard-else结构语句是Swift 2.0之后新加入的一种语法结构，Swift团队创造它的目的在于使代码的结构和逻辑更加清晰。在实际开发中，尤其是在函数的编写中，经常会遇到这样的场景：当参数符合某个条件时，函数才能正常执行，否则直接通过return来终止函数的执行，如果不使用guard-else结构，示例代码如下：

```
func myFuncTwo(param:Int)  {
    if param <= 0  {
        return
    }
    print("其他操作")
}
```

上面的代码结构在逻辑上并不那么优美，开发者的原意是当param参数大于0时才执行函数中的操作，在2.0之前却使用了相反的逻辑来中断函数，当然，开发者也可以将函数实现如下：

```
func myFuncTwo(param:Int)  {
    if param > 0  {
         print("其他操作")
    }
}
```

经过修改后，代码逻辑清晰了许多，然而还是有一些问题，如果这个函数中需要做的操作很

多，那么所有条件判断的代码都将写在if语句块中，代码结构就显得杂乱无章，guard-else语句就是为了优化这种情况而产生的。guard-else语句也被称为守护语句，顾名思义，其作用就是确保某个条件成立才允许其后的代码执行，示例如下：

```
func myFuncTwo(param:Int)  {
   guard param>0 else {
      return
   }
   print("其他操作")
}
```

4.6　练习及解析

（1）将下列描述翻译成Swift表达式。

小李买了5支铅笔、1块橡皮、3本作业本和11个书签。每支铅笔2元，每块橡皮3元，每本作业本2.5元，每个书签0.5元，计算小李共花了多少钱。

解析：

```
// 共 26 元
// 将不同类型的数据拆开计算，方便 Swift 进行类型推断
var tip =  11 * 0.5
var book = 3 * 2.5
var sum = 5 * 2 + 1 * 3 + tip + book
```

（2）设计一个表达式来生成1~7的随机数。

解析：

```
// arc4random()为 Swift 标准函数库中的随机数生成函数
var rand = arc4random()%7+1
```

（3）语文、数学、英语3门科目进行测试，当3门科目的成绩都大于60且总分不小于200分时，成绩才为合格，使用Swift表达式来描述上述逻辑。

解析：

```
var Language=60
var Math=65
var English=70
if Language>60 && Math>60 && English>60 && (Language+Math+English)>200 {
   print("合格")
}
```

（4）编写闰年判断的表达式。

闰年：①能够被400整除。

②能够被4整除，但是不能够被100整除。

解析：

```
var year = 2016
```

```
if year%400==0 || ((year%4==0) && (year%100 != 0)) {
   print("闰年")
}
```

（5）学校乒乓球比赛需要每班出一名主选手和一名辅助选手参赛，比赛分为上下两场，上半场主选手得分超过30分则下半场需要辅助选手进行比赛，否则下半场依然由主选手进行比赛，使用条件运算符（三目运算符）描述下半场比赛出赛的选手。

解析：

```
var mark = 40
var people = mark>30 ? "主选手" : "辅助选手"
```

（6）打印如下图案：

```
**********
*????????*
*????????*
**********
```

解析：

```
for indexH in 1...4 {
   //每行有 10 列符号
   print("")
   for indexV in 1...10 {
      //第一行和最后一行为*
      if indexH==1 || indexH==4 {
         print("*", separator: "", terminator: "")
      }else{
         //第一列和最后一列为*
         if indexV==1 || indexV==10 {
            print("*", separator: "", terminator: "")
         }else{
            //其余为?
            print("?",separator: "",terminator: "")
         }
      }
   }
}
```

print()函数会自动在打印末尾添加换行符，使用带三个参数的print()函数，并且将后两个参数设置为空字符串可以屏蔽print函数的自动换行功能。

（7）打印出所有的“水仙花数”。所谓“水仙花数”，是指一个三位数，其各位数字的立方和等于该数本身。

解析：

```
for item in 100...999 {
   //获取个位数字
   var dig = item%10
   //获取十位数字
   var tens = item/10%10
```

```
    //获取百位数字
    var hundred = item/100
    //获取结果，这里可以考虑用 pow(Double,Double) 函数代替 dig*dig*dig
    var sum = dig*dig*dig + tens*tens*tens + hundred*hundred*hundred

    if sum == item {
        print(item)
    }
}
```

（8）猴子吃桃问题：猴子第一天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个，第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第10天早上想再吃时，见只剩下一个桃子了。求第一天共摘了多少。

解析：

```
var count = 1
for day in 1...9 {
    count = (count+1)*2
}
print(count)
```

（9）两个乒乓球队进行比赛，每队各出三人。甲队为p1、p2、p3三人，乙队为q1、q2、q3三人。抽签决定了比赛名单后，有人向队员打听比赛的名单。p1说他不和q1比，p3说他不和q1、q3比，请编写程序列出三队赛手的名单。

解析：

```
//标识甲队
var p1 = 1
var p2 = 2
var p3 = 3
//标识乙队
var q1 = 0
var q2 = 0
var q3 = 0
for indexI in 1...3 {
    q1 = indexI
    for indexJ in 1...3 {
        q2 = indexJ
        for indexK in 1...3 {
            q3 = indexK
            if indexI != indexJ && indexI != indexK && indexJ != indexK {
                if q1 != p1 && p3 != q1 && p3 != q3 {
                    print(q1,q2,q3)
                }
            }
        }
    }
}
//输出 2,3,1
```

（10）求1+2!+3!+...+20!的和。

解析：

```
var sumC = 0
for var index in 1...20 {
    var tmp = 1
    while index > 0{
        tmp *= index
        index -= 1
    }
    sumC+=tmp
}
print(sumC)
//输出 2561327494111820313
```

（11）打印倒金字塔

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

解析：

```
    for indexJ in 1...7 {
        if indexJ < indexI{
            //先打印左侧空格
            print(" ", separator: "", terminator: "")
        }else if indexJ+(indexI-1)<=7 {
            //再打印*
            print("*",separator: "",terminator: "")
        }
    }
    //换行
    print("")
}
```

4.7 模拟面试

（1）编程中的流程控制结构有哪几种，分别用于什么场景？

回答要点提示：

①编程中主要的流程结构有顺序结构、分支结构、循环结构、跳转与中断结构。

②在编写代码时，我们的核心思路和代码的主流程都是线性的，代码是一行一行向下执行的，这就是我们最常用的顺序结构。分支结构是程序逻辑的重要描述方式，输入不同，不同的运行场景都会对程序执行的结果产生影响，这时我们需要使用分支结构来处理。循环结构用来处理大量重复的工作。跳转和中断结构使得分支和循环结构更加灵活可控。

核心理解内容：

理解各种程序流程控制的方法，能够在开发中根据实际场景灵活使用各种流程控制结构。

（2）运算符是一门编程语言的基础，Swift中有哪些特殊的运算符？

回答要点提示：

①Swift是一门非常强大的语言，在Swift语言中，开发者可以根据需要对运算符进行重载，也可以进行运算符的自定义。

②由于Swift语言中存在Optional类型值，因此Swift语言中提供了空合并运算符来对Optional值进行快捷的条件运算。

③在Swift语言中，区间运算符也是一种十分有特点的运算符，使用它可以方便地创建区间与范围，在集合遍历、字符串和数组的截取中都十分有用。

核心理解内容：

熟悉Swift中的运算符重载和自定义的方法，熟练使用Swift原生定义的各种运算符。