# 第一章 iOS简介以及环境搭建

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

## 1.1 iOS系统简介

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

### 1.1.1 iOS发展历程

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

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

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

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

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

2011年10月4日，苹果公司宣布iOS平台的应用程序已经突破50万个。

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

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

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

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

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

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

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

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

### 1.1.2 iOS系统特性

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

总结iOS的功能特性，有以下几点：

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

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

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

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

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

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

## 1.2 Swift语言介绍

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

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

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

## 1.3 开发环境的搭建

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

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

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

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

### 1.3.1 Xcode欢迎界面介绍

Xcode 安装完成后就可以开始编写 Swift 代码了。打开Xcode，进入欢迎界面，如图1.1所示。

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

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

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

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

本书主要使用前两个选项。



图1.1 Xcode欢迎界面

### 1.3.2创建Playground文件

选择 Get started with a playground 选项，在弹出的界面上方选择iOS，Xcode默认为Blank，点击Next，并输入playground 的名称。如图1.2是默认 Swift Playground 的开始窗口。



图1.2 Playground 的开始窗口

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

### 1.3.3创建iOS项目

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

项目创建后，默认生成了一个示例文件，如图1.4所示（在界面左侧）。打开main.storyboard,默认看到一个简单的空白的设备应用界面。main.storyboard显示的模拟器大小可以在其正下方界面最底部“View as”看到，点开后会以界面大小顺序显示不同型号模拟器界面大小（如图1.5所示）。Xcode还允许控制main.storyboard中设备界面显示的比例（图1.5右上角部分）。如想设置运行后显示的模拟器型号，则要点击界面标题栏左上位置（如图1.4标记位置）设置模拟器显示的设备型号。

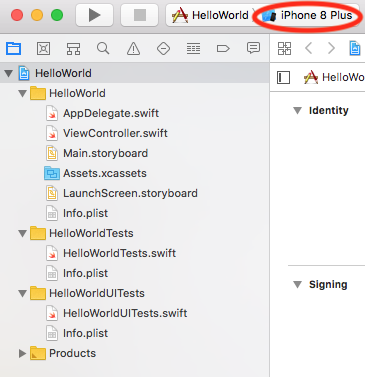
我们为界面添加点内容，在右下方找到Text控件，将它拖入storyboard上，

双击写入文本"Hello World!"（如图1.6）。

运行一下模拟器（command+R 快捷键或在菜单栏中选择 Product => Run或击左上角的运行符号)。模拟器运行结果如图1.7。



图1.3 创建一个Singles View App



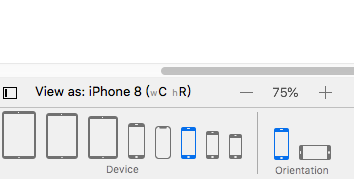


图1.4 示例文件 图1.5 模拟器大小显示



图1.6 添加TextField控件



图1.7 模拟器运行结果

## 1.4 本章小结

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

# 第二章 数据类型与运算

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

## 2.1 变量与常量

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

### 2.1.1声明常量和变量

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

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

可以用print()函数来输出这个常量和变量

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

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

### 2.1.2变量与常量的命名规范

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

和其他语言类似，Swift也可以用中文进行变量的命名，示例如下：

|  |
| --- |
| var 杭电 = "杭电" |

可以穿插数字进行命名，注意数字不能作为开头：

|  |
| --- |
| var name1 = "雨燕" |

可以使用下划线进行命名：

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

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

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

## 2.2 关于注释

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

//单行注释

/\*

多行注释

多行注释

多行注释

\*/

Swift语言也可以支持注释的嵌套，示例如下：

//单行注释//注释中的注释

/\*多行注释

/\*

注释中的注释

\*/

多行注释

\*/

## 2.3数据类型

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

### 2.3.1 布尔型数据

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

【例2-1】:

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

程序输出结果:

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

【例2-2】:

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

### 2.3.2 整数型数据

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

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

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

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

### 2.3.3 浮点型数据

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

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

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

### 2.3.4 字符类型

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

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

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

程序输出结果:

|  |
| --- |
| 24 |

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

创建字符的两种方法，示例如下：

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

## 2.4字符串

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

### 2.4.1创建字符串

在Swift语言中，有两种方式去创建字符串，例如：

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

以上程序执行输出结果：

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

### 2.4.2创建空字符串

创建一个空字符串，并且判断它是否为空。

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

以上程序执行输出结果为:

|  |
| --- |
| string1是空字符串 |

## 2.5运算符

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

### 2.5.1赋值运算符

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

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

### 2.5.2区间运算符

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

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

以上程序输出结果:

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

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

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

以上程序输出结果:

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

注意：区间操作只有这两种，读者切勿自己推导出<..等不存在的运算符。

### 2.5.3算术运算符

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

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

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

### 2.5.4条件运算符

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

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

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

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

即三目运算符的基本语法格式：条件？成立时执行的语句：不成立时执行语句

## 2.6本章小节

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

# 第三章 流程控制

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

## 3.1循环语句

### 3.1.1 for-in

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

Swift中for-in循环的基本语法如下：

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

用for-in循环打印区间0...5之间的数字。

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

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

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

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

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

for-in循环也常常被用来遍历集合类型。用for-in来遍历数组。

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

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

### 3.1.2 while

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

Swift提供两种形式的while循环：

while循环，在每次执行循环体之前判断判定条件是否为真（true）。

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

* while

while循环基本语法：

|  |
| --- |
| while 条件 {  循环体  } |

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

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

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

* repeat-while

repeat-while循环基本语法：

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

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

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

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

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

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

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

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

## 3.3 条件语句

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

### 3.3.1 if-else

if-else基本语法：

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

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

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

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

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

还可以通过多个if-else配合的方式，来实现更复杂的分支。

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

### 3.3.2 switch

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

switch基本语法：

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

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

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

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

switch的case还提供了多种功能更加强大的复杂匹配模式。

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

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

## 3.4 控制转移语句

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

### 3.4.1 continue

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

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

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

### 3.4.2 break

break语句的作用是停止本次循环，并结束整个循环过程。

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

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

### 3.4.3 fallthrough

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

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

如果switch中没有fallthrough语句:

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

在某些case中添加fallthrough语句:

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

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

### 3.4.4 guard

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

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

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

## 3.5本章小结

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

# 第四章 元组和集合类型

## 4.1 元组

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

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

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

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

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

如果只需要元组的一部分值，分解的时候可以用\_代替被忽略的部分。

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

此外，还可以通过下标直接获得元组的单个数据项，下标从零开始。

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

如果，在元组定义的时候给单个数据项命名，之后就可以用名称直接调用。

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

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

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

## 4.2 集合类型和集合可变性

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

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

## 4.3 数组(Array)

### 4.3.1 数组的创建

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

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

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

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

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

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

### 4.3.2 数组的常用属性和方法

Swift中提供了一些常用的属性和方法，这样可以更加方便的使用数组。

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

### 4.3.3 数组的遍历

可以使用上一章介绍的for-in循环来遍历数组中的数据项。

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

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

或是使用下面这种更加简洁的形式遍历数组中的每个数据项。

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

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

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

### 4.3.4 数组的读取和修改

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

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

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

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

当然，也可以通过下标的方式访问数据项和修改数据项。

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

Swift数组提供了几种remove方法，用来删除数组中的数据项。

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

## 4.4 集合(Set)

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

### 4.4.1 集合的创建

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

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

与数组类似，也可以使用数组字面量来构造集合，并且可以使用简化形式写一个或者多个值作为集合数据项。

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

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

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

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

### 4.4.2 集合的遍历

在一个for-in循环中遍历一个Set中的所有值。

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

### 4.4.3 集合的修改

通过调用Set的insert(\_:)方法来添加一个新数据项。

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

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

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

### 4.4.4 集合的重组和组合

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

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

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

union()方法根据两个集合的值创建一个新的集合。

subtracting()方法根据不在该集合中的值创建一个新的集合。

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

## 4.5 字典（Dictionary）

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

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

### 4.5.1 字典的创建

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

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

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

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

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

### 4.5.2 字典的遍历

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

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

### 4.5.3 数组的读取和修改

可以使用下标语法，通过key获取字典中对应的value。

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

通过下标语法，还可以往字典中添加新的数据项。

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

如果下标中的key已经存在于字典中，新设置的value会覆盖旧value。

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

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

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

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

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

## 4.6 本章小结

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

# 第五章 函数与闭包

## 5.1函数

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

### 5.1.1函数的声明和使用

函数的基本语法形式如下：

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

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

（1）无参函数

无参函数就是没有参数列表的函数。无参函数声明定义的一般形式如下：

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

函数调用的形式如下：

|  |
| --- |
| 函数名（） |

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

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

运行结果为：

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

（2） 有参函数

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

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

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

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

其中参数值和参数名的类型一定要一致。

【例 5-2】 以下定义一个sum函数来实现2个数的相加，代码如下：

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

运行结果为：

|  |
| --- |
| 30 |

（3）无返回值的函数

函数是可以没有返回值的，其声明定义的形式如下：

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

函数调用的形式如下：

|  |
| --- |
| 函数名（参数列表） |

（4） 具有一个返回值的函数

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

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

返回的表达式类型必须与函数的返回值类型一致。

【例 5-3】 以下定义一个sum1函数来实现2个数的相加，代码如下：

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

运行结果为：

|  |
| --- |
| 30  40 |

（5）具有多个返回值的函数

函数返回多个返回值，就需要使用到元组类型。这种函数的声明定义形式如下：

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

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

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

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

运行结果为：

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

### 5.1.2 关于函数参数的使用

如果开发者使用的函数为有参函数时，对于参数需要了解以下几点。

（1）关于函数参数名

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

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

调用的一般形式如下：

|  |
| --- |
| 函数名（外部参数名：参数值） |

（2）忽略参数名

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

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

【例 5-5】 以下定义一个sum3函数来实现2个数的相加，代码如下：

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

运行结果为：

|  |
| --- |
| 30 |

### 5.1.3 为参数设置默认值

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

【例 5-6】 以下定义一个sum4函数来实现求和运算，代码如下：

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

运行结果为：

|  |
| --- |
| 40 |

### 5.1.4 可变参数

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

可变参数通过在数据类型名后面加入 ‘...’ 的方式来定义。

【例 5-7】 以下定义一个sum5函数来实现求和运算，代码如下：

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

运行结果为：

|  |
| --- |
| 7.32 |

5.1.5输入输出参数

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

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

注意：使用inout关键字的参数必须是已经声明过的变量。

函数的调用形式如下：

|  |
| --- |
| 函数名（&参数，...） |

【例 5-8】 以下定义一个sum6函数来实现求和运算，代码如下：

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

运行结果为：

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

### 5.1.6函数类型

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

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

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

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

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

或者：

|  |
| --- |
| let/var 常量名/变量名=函数名 |

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

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

运行结果为：

|  |
| --- |
| 50  60 |

函数类型可以作为函数参数来使用。

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

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

运行结果为：

|  |
| --- |
| 80 |

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

函数类型还可以作为返回类型来使用。

【例 5-11】 以下把函数类型作为返回值并使用，代码如下：

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

运行结果为：

|  |
| --- |
| 25 |

### 5.1.7函数的嵌套

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

1.嵌套函数

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

【例 5-12】 以下在函数中声明定义嵌套函数并调用，代码如下：

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

运行结果为：

|  |
| --- |
| 46不可以被3整除。 |

2.嵌套调用

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

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

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

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

【例 5-13】 以下通过递归调用来计算5！的值：

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

运行结果为：

|  |
| --- |
| 120 |

## 5.2 闭包

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

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

### 5.2.1闭包表达式

闭包表达式的基本语法形式如下：

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

定义的语法形式如下：

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

调用形式如下:

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

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

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

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

运行结果为：

|  |
| --- |
| 10  Swift |

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

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

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

运行结果为：

|  |
| --- |
| true |

### 5.2.2闭包表达式的优化

以下都以【例5-13】为例：

**(1)推断类型和**省略return

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

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

（2）简写或省略参数名

开发者可以通过$0,$1,$2来顺序的调用闭包的参数。

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

闭包的函数体很短时可以写成一行，所以写成一行的写法并没有错。

### 5.2.3尾随闭包

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

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

当函数只有闭包这一个参数时，可以把括号也省略掉。

【例 5-16】用sorted函数配合闭包完成排序。

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

运行结果为：

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

### 5.2.4 捕获值

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

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

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

运行结果为：

|  |
| --- |
| 6  12 |

## 5.3 本章小结

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

# 第六章 枚举与结构体

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

## 6.1 枚举类型的创建与应用

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

1、任意枚举类型

enum 枚举名{

}

任意枚举类型在枚举的相关值中用的比较多，一般就是不为枚举类型设置数据类型。

2、指定枚举类型

enum 枚举名:数据类型{

}

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

例如我们定义表示星期的枚举：

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

以上程序执行输出结果为：

|  |
| --- |
| 星期一 |

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

## 6.2枚举的原始值与相关值

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

### 6.2.1枚举的原始值

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

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

通过枚举类型提供的rawValue属性来获取枚举的原始值，示例如下：

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

### 6.2.2枚举的相关值

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

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

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

以上程序执行输出结果为：

|  |
| --- |
| 学生的成绩是：10，10，10 |

## 6.3 Swift结构体

在Swift语言中，结构体与类的使用方法特别的相似。

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

通过关键字struct来定义结构体：

struct nameStruct{

属性

方法

}

实例如下：

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

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

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

## 6.4本章小节

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

# 第七章 类

## 7.1 类的概念

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

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

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

Swift类中的属性包括存储属性和计算属性，方法包括实例方法和类型方法。

### 7.1.1 类的定义

在Swift中，定义类的关键字为class，基本语法：

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

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

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

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

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

程序输出：

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

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

### 7.1.2 构造过程

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

|  |
| --- |
| init(){  //初始化  } |

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

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

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

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

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

这样，Car类就有了两个构造器，使用类名( )的形式可以创建对象。

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

程序输出：

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

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

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

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

本例中，response被声明为可选类型，并且没有初始化。

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

程序输出：

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

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

### 7.1.3 类的引用特征

在swift中，元组、枚举和结构体属于值类型，而类是引用类型。

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

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

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

程序输出：

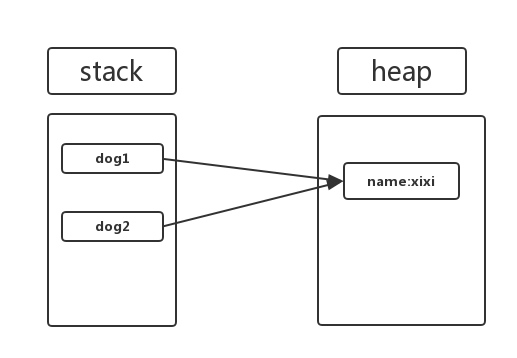
|  |
| --- |
| dog1:haha |

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

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

程序输出：

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

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

### 7.1.4 恒等运算符

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

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

### 7.1.5 析构过程

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

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

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

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

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

在第一行代码中，定义了一个名为FileManager的类。

接着定义了2个实例方法，分别打印对应的一句话，代表完成了某些操作。

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

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

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

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

## 7.2 类的属性

### 7.2.1 存储属性和计算属性

属性可以分为存储属性和计算属性。

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

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

本章之前的属性都属于存储属性，下面用一个例子介绍计算属性。

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

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

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

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

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

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

### 7.2.2 属性观察器

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

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

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

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

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

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

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

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

### 7.2.3 懒加载属性

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

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

## 7.3 类的方法

### 7.3.1 实例方法

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

下面的例子，定义一个简单的Counter类，实现计数的功能。

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

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

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

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

程序输出：

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

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

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

也有某些情况下，self不可以省略，比如第一节包含参数的构造函数的例子：

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

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

### 7.3.2 类型方法

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

下面定义一个Math类，包含一个求绝对值的方法。

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

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

通过类名.(dot)方法名来调用类型方法。

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

程序输出：

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

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

## 7.4 类的继承

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

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

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

首先，创建一个基类，也就是没有继承其他类的类。

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

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

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

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

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

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

在定义好子类以后，就可以定义子类的实例了。

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

程序输出：

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

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

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

## 7.5 本章小结

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

# 第八章 扩展与协议

## 8.1 扩展

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

扩展声明使用关键字 extension：

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

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

### 8.1.1属性

扩展可以向已有类型添加计算属性和类型属性,但是不能添加存储属性。

【例 8-1】创建一个Num类并向这个类型扩展属性，代码如下：

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

运行结果为：

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

### 8.1.2 类型方法和实例方法

在扩展中可以向已有类型添加新的类型方法和实例方法。

【例 8-2】使用扩展向类Student添加新的类型方法和实例方法：

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

运行结果为：

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

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

【例 8-3】使用扩展向值类型添加新的方法来改变实例的值，代码如下：

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

运行结果为：

|  |
| --- |
| 15 |

### 8.1.3 构造器

扩展可以向类型添加新的构造器。

【例 8-4】使用扩展向结构体Newstruct添加新的构造器：

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

运行结果为：

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

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

【例 8-5】使用扩展向Ms类添加新的构造器：

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

运行结果为：

|  |
| --- |
| jarry  micheal |

### 8.1.4下标脚本

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

【例8-6】使用扩展向Ss类型添加新的下标脚本：

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

运行结果为：

|  |
| --- |
| false |

## 8.2 协议

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

协议的定义形式如下：

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

对于遵守该协议的类型，以类为例，形式如下：

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

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

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

下面将会介绍对于协议中不同种类成员的具体声明和其实现。

### 8.2.1属性

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

对协议中属性的声明如下：

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

【例8-7】声明协议Stu并定义类Studentclass来遵守该协议：

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

运行结果为：

|  |
| --- |
| Jerry  10 |

### 8.2.2方法

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

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

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

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

运行结果为：

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

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

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

【例8-9】声明协议Change并定义枚举number来遵守该协议：

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

运行结果为：

|  |
| --- |
| 3.3 |

### 8.2.3可选协议

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

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

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

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

运行结果为：

|  |
| --- |
| 10.648 |

### 8.2.4协议类型

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

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

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

运行结果为：

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

假如某数组的类型为协议类型，如：

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

这表示数组里的所有元素都要遵守协议。

### 8.2.5协议的继承

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

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

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

### 8.2.6协议的其他功能

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

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

使用 ‘&’ 把各协议隔开就可实现协议的合成。

如果想让某协议只能被类遵守，可以用关键字class来说明，如：

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

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

（1）is操作符用来检查实例是否遵循了某个协议。

（2）as?是当实例遵循某协议时，返回该协议类型;否则返回nil。

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

## 8.3 本章小结

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

# 第九章 异常处理与类型转换

## 9.1异常处理

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

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

### 9.1.1普通的异常处理

对于没有异常捕获的程序中，如何来修改和处理程序中的错误? 实例如下:

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

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

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

以上程序输出:

|  |
| --- |
| 除数不能为零  nil |

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

### 9.1.2异常的捕获处理

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

第一步:定义异常类型

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

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

通过以上的方法，可以高度自定义自己的异常类型。

第二步：使用throws来声明异常，用throw来抛出异常

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

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

第三步：异常的捕获与处理

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

do{

try 可能抛出异常的函数

语句  
}catch 异常类型{

语句

}

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

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

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

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

### 9.1.3异常处理统一出口

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

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

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

以上程序输出：

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

## 9.2类型转换

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

### 9.2.1类型检查

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

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

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

### 9.2.2类型转换

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

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

以上程序输出:

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

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

### 9.3本章小节

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