**Learn Xcode for IOS App**

[1. Swift基础语法 2](#_Toc65441899)

[1.1 基础语法 2](#_Toc65441900)

[1.1.1 打印、定义变量 2](#_Toc65441901)

[1.1.2 列表 2](#_Toc65441902)

[1.1.3字典 3](#_Toc65441903)

[1.2 控制流程 3](#_Toc65441904)

[1.2.1 for-in循环 3](#_Toc65441905)

[1.2.2 if判断 4](#_Toc65441906)

[1.2.3 \*while循环 4](#_Toc65441907)

[1.2.4 \*??判断 4](#_Toc65441908)

[1.3 函数和封包 5](#_Toc65441909)

[1.3.1 定义普通函数 5](#_Toc65441910)

[1.3.2 返回元组的函数 5](#_Toc65441911)

[1.3.3 \*函数嵌套 5](#_Toc65441912)

[1.3.4 \*定义返回函数的函数 5](#_Toc65441913)

[1.3.5 \*定义CallBack函数 6](#_Toc65441914)

[1.3.6 \*闭包函数 6](#_Toc65441915)

[1.4 对象和类 6](#_Toc65441916)

[1.4.1 定义类 6](#_Toc65441917)

[1.4.2 继承 6](#_Toc65441918)

[1.4.3 getter和setter 6](#_Toc65441919)

[1.5 枚举和结构 7](#_Toc65441920)

[1.5.1 \*创建枚举 7](#_Toc65441921)

[1.5.2 \*定义结构体 7](#_Toc65441922)

[1.6 \*扩展和协议 7](#_Toc65441923)

[1.7 \*错误处理 7](#_Toc65441924)

# 0. 变量命名风格

基本原则：命名要有描述性，让第一次读代码的人看懂。无所谓变量名长短。

文件命名

全部小写，可以包含下划线，例如cat\_and\_dog\_images

类型命名

每个单词首字母均大写, 不包含下划线: MyViewModel, Cat, Dog

类型实例命名

小写，没有下划线，例如myviewmodel, cat, dog

有多个实例时直接在后面加备注内容，例如myviewmodel1, catblack, dogorange

变量命名

全部小写，可以包含下划线，例如num\_table, size\_box

常量命名

全部大写，可以包含下划线，例如: ITERATION, LENGTH\_MAX

函数命名

每个单词首字母均大写, 不包含下划线: ReadData(), GetLength()

# 1. Swift基础语法

## 1.1 基础语法

### 1.1.1 打印、定义变量

打印

print("Hello World")

定义变量

var age = 25

赋值

age = 28

print("age: ", age)

定义常量

let age = 25

// age = 28 报错

显示制定变量类型

var age:Double = 25.5

print(age)

字符串与数字组合

var player\_name = "Tobias"

var player\_age = 26

print(player\_name + " is " + String(age) + ".") //强制类型转换，字符串可以用"+"拼接

变量嵌入

print("\(player\_name) is \(player\_age).") // 类似于python的格式化输出

定义可以为空的变量

var age:Int? = 25 // 这里有一个问号代表这个变量可以为空，否则下一句会报错

age = nil // nil相当于python里的None

if age == nil

{

print("Age is unknown.")

}

else

{

print("His age is \(age).")

}

### 1.1.2 列表

创建列表

var MyList = ["Tobias", "Haibin", "Mengfan"]

print(MyList)

创建空列表

let mylist = [String]()

打印列表

print(MyList[1]) // "Haibin"

追加元素

Mulist.append("Yunrou")

### 1.1.3字典

定义字典

var MySkill =

[

"Java": 3,

"Python": 4

]

创建空字典

let myskill = [String:Float]()

打印字典

print(MySkill)

print(MySkill["Java"]) // 返回Optional(3)

// Optional( )的意思是这个地方的值可以为空

添加字典内容

MySkill["C++"] = 5

## 1.2 控制流程

### 1.2.1 for-in循环

一般循环

// 两个点

for i in 0..<3

{

print("Step: ", i)

}

// 0, 1, 2

// 三个点

for i in 0..<3

{

print("Step: ", i)

}

// 0, 1, 2, 3

列表循环

var MyList = [1, 2, 3, 4, 5]

for val in MyList

{

print(val)

}

字典循环

var PlayerSocre =

[

"Tobias": 1,

"Haibin": 2

]

for score in PlayerScore

{

if score.value >=2

{

print("\(score.key) is good.")

}

else

{

print("\(score.key) is not good.")

}

}

### 1.2.2 if判断

var answer = 1

var perfect\_solution = 1

var good\_solution = 2

if answer == perfect\_solution

{

print("Your answer is perfect!")

}

else if answer == good\_solution

{

print("Your answer is good!")

}

else

{

print("Your answer is incorrect.")

}

### 1.2.3 \*while循环

一般写法

var i = 1

while i <= 5

{

print("Step: ", i)

i += 1

}

repeat写法

var j = 10

repeat

{

print("Step: ", j)

j += 1

} while j <= 5

### 1.2.4 \*??判断

??运算符判断前者是否为空值，如果是空，就选择后者

var choice\_1:String? = "KIT"

var choice\_2 = "TUM"

print("I would like to go the \(choice\_1 ?? choice\_2) for study!")

// "I would like to go the KIT for study!"

choice\_1 = nil

print("I would like to go the \(choice\_1 ?? choice\_2) for study!")

// "I would like to go the TUM for study!"

## 1.3 函数和封包

### 1.3.1 定义普通函数

func FunctionName(Parameter:Tpye) -> ReturnTpye

{

return ReturnValue

}

举例：

func power(variable:Int) -> Int

{

return a \* a

}

print("The power of 3 is \(power(variable:3)).")

### 1.3.2 返回元组的函数

func edge\_length(x1:Int, x2:Int, y1:Int, y2:Int) -> (Length:Int, Width:Int)

{

return (x2 - x1, y2 - y1)

}

直接打印

print(edge\_length(x1:0, x2:10, y1:1, y2:5))

// (Length: 10, Width: 4)

赋值后使用

let SizeOfBox = edge\_length(x1:0, x2:10, y1:1, y2:5)

print(SizeOfBox.Length) // 10

print(SizeOfBox.Width) // 4

### 1.3.3 \*函数嵌套

func add\_2\_number(x:Int, y:Int) -> Int

{

func add() -> Int

{

return x + y

}

return add()

}

add()里面可以直接调用add\_2\_number()的变量

### 1.3.4 \*定义返回函数的函数

func FindMethod() -> ((Int, Int)-> Int) // 返回的是一个输入2输出1的函数，也就是下面的add()

{

func add(x:Int, y:Int) -> Int

{

return x + y

}

return add

}

调用

var MyMethod = FindMethod()

print(MyMethod(10, 20)) // 30

### 1.3.5 \*定义CallBack函数

func showScreen(threshold:Int, cb: (String) -> Void)

{

pring("showScreen: ", threshold)

cb("Hello, World!")

}

func screen\_cb(screen\_data:String) -> Void

{

pring("screen\_data is ", screen\_data)

}

showScreen(threshold: 100, cb: screen\_cb)

// showScreen: 100

// screen\_data is Hello, World!

### 1.3.6 \*闭包函数

var numbers = [1, 2, 3, 4, 5]

var MyList = numbers.map({(number:Int) -> Int in

return number \* number

})

print(MyList) // [1, 4, 9, 16, 25]

## 1.4 对象和类

### 1.4.1 定义类

class Player

{

var score = 0

init(score:Int) // 和Python一样

{

self.score = score

}

func PlayerLevel() -> String

{

if self.score >= 30

{

return "MVP"

}

else

{

return "Good"

}

}

}

使用类

var player1 = Player(score: 25)

print(player1.PlayerLevel())

### 1.4.2 继承和多态

class NBAPlayer: Player // 继承

{

override func PlayerLever() -> String // 多态

{

if self.score >= 15

{

return "MVP"

}

else

{

return "Good"

}

}

}

### 1.4.3 getter和setter

class Circle

{

var radius = 0.0

init(radius: Float)

{

self.radius = radius

}

var area:Float

{

get // 每次获取，就执行这个

{

return 3.14 \* power(self.radius, 2)

}

set // 每次设置，就执行这个

{

self.radius = power(newValue / 3.14, 0.5)

}

}

}

使用

var circle = Circle(radius: 10)

print(circle.area) // 314

circle.area = 100

print(circle.radius) // 5.64333

## 1.5 枚举和结构

### 1.5.1 \*创建枚举

enum Language

{

case java

case python

case c

case matlab

case cpp

}

### 1.5.2 \*定义结构体

struct LoginForm

{

var uid:Int

var pwd:String

var authorized\_code:String

}

let user1 = LoginForm(uid: 1001, pwd: "helloworld", authorized\_code: "12345")

## 1.6 \*扩展和协议

## 1.7 \*错误处理

# 2. Xcode + SwiftUI入门

右边的框点一下Resume就可以现实UI界面

command + A 全选， control + i 自动排版

## 2.1 布局

### 2.1.1 基本布局

最基本的横向纵向排列：在ContentView.swift里面

struct ContentView: View {

var body: some View {

Text("Hello, world!")

}

}

按住command键左键点击Text( )，选择Embed in H/V Stack然后就会有

struct ContentView: View {

var body: some View {

VStack {

Text("Hello, world!")

}

}

}

然后可以在Text("Hello, world!")下面增加东西了，比如再增加一个Text("Hello, world!")

在Vstack后面可以加入选项

VStack(alignment: .trailing, spacing:0 ) {

Text("My first APP")

Text("Haibin")

}

.trailing, .center, .leading分别是右对齐，居中对齐和左对齐。

spacing是两个文本直接的间隔

**右上角的“+”可以拖拽到代码或者UI里面**

### 2.1.2 图片

添加图片：点击Assets.xcassets，左下角加号，点击Image Set，改名字，然后把图片拖进来即可

放置图片：

Image("AirCase") // 插入图片

.resizable() // 大小可变

.aspectRatio(contentMode:.fit) // 图片大小的适应方式，还有.fill

点击图片的动作onTapGasture

.onTapGesture {

withAnimation{

self.zoomed.toggle() // zoom.toggle()就是true变false, false变true

}

}

上面意味着点击图片则zoomed在true和false之间变换

把.aspectRatio(contentMode:.fit)改写成

.aspectRatio(contentMode: self.zoomed ? .fit : .fill)

则图片的填充形式也会随着点击图片而变化

### 2.1.3 Button

Button(action: { // action是点击之后的动作

self.title = self.textInput // 先不用管这个

}, label: {

Text("Confirm") // 显示的内容

.font(.title)

.padding()

.background(Color.blue) // 背景颜色

.foregroundColor(.white) // 文字颜色

.cornerRadius(10) // 按钮圆角

})

### 2.1.4 TextField

@State private var textInput:String = "" // 一个状态变量，绑定数据

TextField("Type Here", text: self.$textInput) // Hypt Here是Hint。输入绑定在textInput上

.font(.title)

.frame(width: UIScreen.main.bounds.width - 50, height: 100, alignment:.center)

.padding(10)

.textFieldStyle(RoundedBorderTextFieldStyle())

## 2.2 数据绑定

struct ContentView: View {

@State private var textInput:String = "" // 建立2个状态变量绑定Text和TextField

@State private var title:String = "Hello World!"

var body: some View {

VStack (alignment:.center, spacing: 10) {

Text(self.title) // 把Text文本框绑定self.title

.font(.title)

TextField("Type Here", text: self.$textInput)

.font(.title)

.frame(width: UIScreen.main.bounds.width - 50, height: 100, alignment:.center)

.padding(10)

.textFieldStyle(RoundedBorderTextFieldStyle())

Button(action: {

self.title = self.textInput // 按钮按下则把文本框内容赋给Text

}, label: {

Text("Confirm")

.font(.title)

.padding()

.background(Color.blue)

.foregroundColor(.white)

.cornerRadius(10)

})

}

}

}

## 2.3 获得系统数据

struct ContentView: View {

@Environment(\.colorScheme) var mode // 获取系统数据并附给mode

var body: some View {

Text("System color scheme is: \(mode == .dark ? "Dark" : "Light").") // 根据数据调整内容

}

}}

## 2.4 \*观察者数据绑定

在建立项目的时候选择APP生命周期为UIKit APP Delegate

command + N 建立一个新的Swift文件（我给它起名ViewModel）

在ViewModel.swift里建立一个类，这个类可以在APP生命周期储存所有数据

import Foundation

import SwiftUI

class myviewmodel: ObservableObject { // 继承于ObservableObject

@Published var AppTitle:String = "Learn MVVM" // 比方说我建立2个字符串

@Published var Author:String = "Haibin Zhao" // Published是公用的

}

在AppDelegate.swift里面：

在class里面创建实例

import UIKit

@main

class AppDelegate: UIResponder, UIApplicationDelegate {

var MyViewModel: myviewmodel! // 创建实例，别忘了加 "!"

在application函数里创建self这个实例

func application(...) -> Bool {

self.MyViewModel = myviewmodel() // 这里

return true

}

... // 其他代码省略

}

在SceneDelegate.swift的scene函数里加2行并改1行:

func scene(...) {

// 加下面这2行，从AppDelegate.swift里面拿到实例MyViewModel

let app = UIApplication.shared

let delegate = app.delegate as! AppDelegate

// 改这1行，将MyViewModel传入ContentView页面

let contentView = ContentView(MyViewModel: delegate.MyViewModel)

if let windowScene = scene as? UIWindowScene { ... }

}

最后在ContentView里面使用ViewModel里的变量：

import SwiftUI

struct ContentView: View {

@ObservedObject var MyViewModel: myviewmodel // 获得这个实例

var body: some View {

Text(MyViewModel.AppTitle) // 就是之前定义的"Learn MVVM"了

}

}

struct ContentView\_Previews: PreviewProvider {

static var previews: some View {

ContentView(MyViewModel: myviewmodel()) // 这里也改一下，为了看Preview

}

}

# 8. 访问环境对象

观察者绑定过于繁琐，因为要一层一层传递。可以将变量直接设置为环境变量，从各个地方都能读取。

在建立项目的时候选择APP生命周期为UIKit APP Delegate

command + N 建立一个新的Swift文件（我给它起名ViewModel）

在ViewModel.swift里建立一个类，这个类可以在APP生命周期储存所有数据

import Foundation

import SwiftUI

class myviewmodel: ObservableObject { // 继承于ObservableObject

@Published var AppTitle:String = "Learn MVVM" // 比方说我建立2个字符串

@Published var Author:String = "Haibin Zhao" // Published是公用的

}

在AppDelegate.swift里面：

在class里面创建实例

import UIKit

@main

class AppDelegate: UIResponder, UIApplicationDelegate {

var MyViewModel: myviewmodel! // 创建实例，别忘了加 "!"

在application函数里创建self这个实例

func application(...) -> Bool {

self.MyViewModel = myviewmodel() // 这里

return true

}

... // 其他代码省略

}

在SceneDelegate.swift的scene函数里加3行:

func scene(...) {

// 加下面这2行，从AppDelegate.swift里面拿到实例MyViewModel

let app = UIApplication.shared

let delegate = app.delegate as! AppDelegate

let contentView = ContentView()

.environmentObject(delegate.MyViewModel) // 加这1行，将MyViewModel变成环境变量

if let windowScene = scene as? UIWindowScene { ... }

}

最后在ContentView里面使用ViewModel里的变量：

import SwiftUI

struct ContentView: View {

@EnvironmentObject var MyViewModel: myviewmodel // 获得这个实例

var body: some View {

Text(MyViewModel.AppTitle) // 就是之前定义的"Learn MVVM"了

}

}

struct ContentView\_Previews: PreviewProvider {

static var previews: some View {

ContentView()

.environmentObject(myviewmodel()) // 这里也改一下，为了看Preview

}

}

# 9. MVVM模式

建立SwiftUI APP生命周期

\*建立文件夹将View, Model, ViewModel分类储存

View: UI界面

Model: 数学模型，函数

ViewModel: 控制器(Model转换成View的工具)

右键单击项目文件夹，New Group，命名即可。

在Model文件夹里建立swift文件（比如Book.swift）存放数据结构：

import SwiftUI

struct Book {

var Title:String

var Author:String

}

在View文件夹里建立**SwiftUI View**文件（比如MyHomeView.swift）存放View

import SwiftUI

struct MyHomeView: View {

@StateObject private var viewmodel = MyHomeViewModel() // 和ObservedObject类似

var body: some View {

Text(self.viewmodel.AppTitle)

}

}

struct MyHomeView\_Previews: PreviewProvider {

static var previews: some View {

MyHomeView()

}

}

在ViewModel里面建立MyHomeViewModel.swift来交互View和Model

import SwiftUI

class MyHomeViewModel: ObservableObject {

@Published var bookinfo:Book

init()

{

self.bookinfo = Book(Title: "My Book", Author: "Haibin")

}

var AppTitle:String

{

return self.bookinfo.Title + " (Author: \(self.bookinfo.Author))"

}

}

最后在Project.swift里面改变App启动时的View

import SwiftUI

@main

struct learn\_xcode\_5App: App {

var body: some Scene {

WindowGroup {

MyHomeView() // 这里

}

}

}

# 10. 页面导航

## 10.1 导航页面的基本设置

在var body: some View {}用NavigationView{}把内容圈起来

var body: some View {

NavigationView

{

VStack

{

Text("Hello World!")

.foregroundColor(self.setColor ? Color.red : Color.blue)

Spacer()

}

// 然后设置navigation页面的东西，可以设置页面题目，左上和右上角的东西

.navigationBarTitle("Home Screen", displayMode: .inline) // 标题

.navigationBarItems(

leading: Button(action: { // 左边的东西，这里是个按钮

self.setBold.toggle()

}, label: {

Text("Change the font")

}),

trailing: Button(action: { // 右边的东西，也是个按钮

self.setColor.toggle()

}, label: {

Text("Change the color")

}))

}

}

## 10.2 导航链接

建立View文件夹，把主页面拖进去，或者自己建立一个。

import SwiftUI

struct ContentView: View {

@State private var activeLink: Bool = false // 控制是否切换页面

var body: some View {

NavigationView

{

VStack(alignment: .center, spacing: 0)

{

Text("Hello World!")

NavigationLink(

// 进入DetailView要同时传递activeLink参数。是否进入取决于activeLink的真假

destination: DetailView(active: self.$activeLink), isActive: self.$activeLink,

label:

{

Button(action: {

self.activeLink = true // 点击按钮之后就进入了

}, label: {

Text("Detail Page")

})

})

}

.padding()

.navigationBarTitle("Main Page", displayMode: .inline)

}

}

}

struct ContentView\_Previews: PreviewProvider {

static var previews: some View {

ContentView()

}

}

再建立一个DetailView的SwiftUI文件

import SwiftUI

struct DetailView: View {

@Binding var active: Bool // 传进来的activeLink绑定active

var body: some View

{

Text("I love you!")

Button(action: {

self.active = false // 点击按钮，则activeLink变成false，退回主页面

}, label: {

Text("Back")

})

}

}

struct detailView\_Previews: PreviewProvider {

// @Binding static var active = false

static var previews: some View {

// 为了看效果，这里也要传入一个true，要这样写才行

DetailView(active: .constant(true))

}

}

# 11. Tab标签组件

# 12. 数据读取

# 13. 弹出Sheet视图

# 14. 弹出框

# 15. 警告框

# 16. 获取手机方向

# 17. 父容器几何读取器