

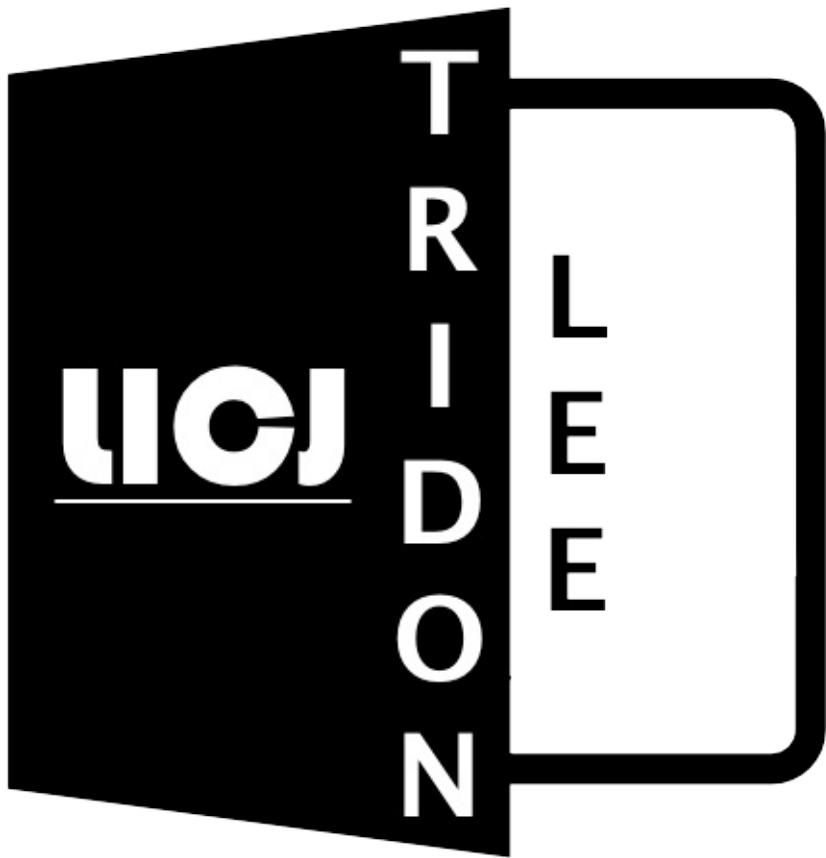
Table of Contents

Introduction	1.1
Swift 开发笔记	1.2
快速体验	1.2.1
playground	1.2.1.1
项目开发体验	1.2.1.2
基本语法	1.2.2
常量和变量	1.2.2.1
Optional	1.2.2.2
控制流	1.2.2.3
循环	1.2.2.4
字符串	1.2.2.5
集合	1.2.2.6
函数	1.2.3
函数	1.2.3.1
闭包	1.2.3.2
闭包的定义	1.2.3.2.1
基本使用	1.2.3.2.2
循环引用	1.2.3.2.3
面向对象	1.2.4
构造函数基础	1.2.4.1
重载构造函数	1.2.4.2
KVC 构造函数	1.2.4.3
便利构造函数	1.2.4.4
懒加载	1.2.4.5
只读属性	1.2.4.6
网络访问	1.2.5
项目演练	1.2.6
准备工作	1.2.6.1
创建模型加载数据	1.2.6.2
明细控制器	1.2.6.3

Swift 开发笔记

编写者：李传军(绰号：军哥)

版本号：1.0.0



内容介绍

本文档是本人在学习 Swift 1.0 ~ Swift 3.1 的过程中遇到语法问题和发现技巧的点点滴滴，仅供参考。

Swift 开发笔记

简介

- Swift 语言由苹果公司在 2014 年推出，用来撰写 OS X 和 iOS 应用程序
- 2014 年，在 Apple WWDC 发布

历史

- 2010 年 7 月，苹果开发者工具部门总监 克里斯·拉特纳 开始着手 Swift 编程语言的设计



- 用一年时间，完成基本架构
- Swift 大约历经 4 年的开发期，2014 年 6 月发布
- 计划在 2015 年底开源

大事件

- 2014 年 6 月苹果在发布 Xcode 6.0 的同时发布了 Swift 1.0
- 2015 年 2 月，苹果同时推出 Xcode 6.2 Beta 5 和 6.3 Beta，在完善 Swift 1.1 的同时，推出了 Swift 1.2 测试版
- 2015 年 6 月，苹果发布了 Xcode 7.0 和 Swift 2.0 测试版，并且宣称在年底开源
- 2015 年 9 月 15 号，正式发布了 Xcode 7.0
- 2016 年 4 月 Xcode 8.0 Beta 中包含 Swift 3 和 Swift 2.3 两个版本的预发布版本
- 2017 年 6 月 Xcode 9.0 Beta 2 中包含 Swift 4 预发布版本

从发布至今，苹果的每一个举措都彰显其大力推广 Swift 的决心，
由于语法更迭太快，最苦逼也是收获最大的莫过于用 Swift 开发框架的程序员们

版本

- 正式版 3.0 Xcode 8.3.3
- 测试版 4.0 Xcode 9.0 beta 2

Swift 特色

- 苹果宣称 Swift 的特点是：快速、现代、安全、互动，而且明显优于 Objective-C 语言
- 可以使用现有的 Cocoa 和 Cocoa Touch 框架
- Swift 取消了 Objective-C 的指针 及其他不安全访问的使用
- 舍弃 Objective-C 早期应用 Smalltalk 的语法，全面改为句点表示法
- 提供了类似 Java 的名字空间（namespace）、泛型（generic）、运算对象重载（operator overloading）
- Swift 被简单的形容为“没有 C 的 Objective-C”（Objective-C without the C）

Swift 现状

- 目前国内有些公司的新项目已经直接采用 Swift 开发
- 目前很多公司都在做 Swift 的人才储备
- 假如换新工作时，会 Swift 开发无疑会增加自身筹码

为什么要学习 Swift?

1. 从4月份开始，苹果提供的资料已经没有 OC 的了，这说明苹果推动 Swift 的决心
2. OC 源自于 Smalltalk-C，迄今已经有 40 多年的历史，虽然 OC 的项目还会在未来持续一段时间，但是更换成 Swift 是未来必然的趋势
3. 现在很多公司都注重人才储备，如果会 Swift，就业会有很大的优势，简历中如果写上会 Swift，虽然面试中虽然不会怎么被问到，但对于薪资提升有很大帮助，同时可以从另外一个侧面证明我们是有自学能力的人，这是所有企业都需要的
4. Swift 里面融合了很多其他面向对象语言的思想，不像OC那么封闭，学会 Swift，再转其他语言会轻松很多
5. Swift 毕竟也是出身自苹果，整体程序开发思路和 OC 是一样的，等 Swift 项目完成后，大家完全可以用同样的思路写出 OC 的来，而且在翻写的过程中，能够对很多原本忽略的 OC 基本功有很大的加强和改善

建议

- Objective-C & Swift 对比学习能够对苹果底层的很多实现原理有更加深刻的体会
- 分享结束后，建议用 Objective-C 和 Swift 对比写一个实战项目。

Swift 开发快速体验

目标

- playground 快速体验 & 学习资源分享
- 项目开发快速体验，了解 Swift 基本程序结构

学习资源

- 苹果官方博客 <https://developer.apple.com/swift/blog/>
- 苹果官方 Swift 2.0 电子书 <https://itunes.apple.com/us/book/id1002622538>
- 2.0 中文版 <http://wiki.jikexueyuan.com/project/swift/>
- 100个Swift必备tips，作者王巍，建议购买实体书 <http://onevcat.com>

Playground

The screenshot shows a Xcode playground interface. On the left is the code editor with the following Swift code:

```

6 let view = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
7 view.backgroundColor = UIColor.yellowColor()
8
9 let btn = UIButton(type: .ContactAdd)
10 view.addSubview(btn)
11
12 btn.center = view.center
13
14
15 var x: Double = 0
16 for i in 0..<72 {
17     x = sin(M_PI / 18 * Double(i))
18 }
19
20 }

```

The right side displays the results of the code execution. It shows a yellow `view` containing a `UIButton` with a plus sign. Below this, a sine wave is plotted with 72 points.

- Playground 是 Xcode 6 推出的新功能
- 创建工程编写和运行程序，目的是为了编译和发布程序
- 而使用 Playground 的目的是为了：
 - 学习代码
 - 实验代码
 - 测试代码
- 并且能够可视化地看到运行结果
- 另外，使用 Playground 只需要一个文件，而不需要创建一个复杂的工程

快速体验

```

let btn = UIButton(type: UIButtonType.ContactAdd)

let view = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
view.backgroundColor = UIColor.lightGrayColor()

btn.center = view.center
view.addSubview(btn)

print(view)

```

```
print(view.subviews)
```

提示

- 官方提供的一些学习资源是以 `playground` 的形式提供的
- 建立一个属于自己的 `playground` 文件，能够在每次版本升级时，第一时间发现语法的变化

项目开发体验

目标

- 熟悉 Swift 的基本开发环境
- 与 OC 开发做一个简单的对比

代码实现

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // swift 中 () 代替 oc 中的 alloc / init
        let v = UIView(frame: CGRect(x: 0, y: 20, width: 100, height: 100))

        // [UIColor redColor];
        v.backgroundColor = UIColor.redColor()

        // 按钮
        let btn = UIButton(type: .ContactAdd)
        v.addSubview(btn)

        // 监听方法
        btn.addTarget(self, action: "click:", forControlEvents: .TouchUpInside)

        view.addSubview(v)
    }

    func click(btn: UIButton) {
        print("点我了 \(btn)")
    }
}
```

小结

- 在 Swift 中没有了 `main.m`, `@UIApplicationMain` 是程序入口
- 在 Swift 中只有 `.swift` 文件, 没有 `.h/.m` 文件的区别
- 在 Swift 中, 一个类就是用一对 `{}` 括起的, 没有 `@implementation` 和 `@end`
- 每个语句的末尾没有分号, 在其他语言中, 分号是用来区分不同语句的
 - 在 Swift 中, 一般都是一行一句代码, 因此不用使用分号
- 与 OC 的语法快速对比

- 在 OC 中 `alloc / init` 对应 `()`
- 在 OC 中 `alloc / initWithXXX` 对应 `(XXX:)`
- 在 OC 中的类函数调用，在 Swift 中，直接使用 `.`
- 在 Swift 中，绝大多数可以省略 `self.`，建议一般不写，可以提高对语境的理解（闭包时会体会到）
- 在 OC 中的枚举类型使用 `UIButtonTypeContactAdd`，而 Swift 中分开了，操作热键：回车 -> 向右 `-> .`
 - Swift 中，枚举类型的前缀可以省略，如：`.ContactAdd`，但是：很多时候没有智能提示
- 监听方法，直接使用字符串引起
- 在 Swift 中使用 `print()` 替代 OC 中的 `NSLog`

基本语法

目标

- 熟悉 Swift 基本语法
 - 常量 & 变量
 - 可选项
 - 控制流
 - if
 - 三目
 - if let
 - guard
 - switch
 - 字符串
 - 循环
 - 集合
 - 数组
 - 集合

变量和常量

定义

- `let` 定义常量，一经赋值不允许再修改
- `var` 定义变量，赋值之后仍然可以修改

```
//: # 常量
//: 定义常量并且直接设置数值
let x = 20
//: 常量数值一经设置，不能修改，以下代码会报错
// x = 30

//: 使用 `: 类型`，仅仅只定义类型，而没有设置数值
let x1: Int
//: 常量有一次设置数值的机会，以下代码没有问题，因为 x1 还没有被设置数值
x1 = 30
//: 一旦设置了数值之后，则不能再次修改，以下代码会报错，因为 x1 已经被设置了数值
// x1 = 50

//: # 变量
//: 变量设置数值之后，可以继续修改数值
var y = 200
y = 300
```

自动推导

- Swift能够根据右边的代码，推导出变量的准确类型
- 通常在开发时，不需要指定变量的类型
- 如果要指定变量，可以在变量名后使用`:`，然后跟上变量的类型

重要技巧：Option + Click 可以查看变量的类型



没有隐式转换！！！

- Swift 对数据类型要求异常严格
- 任何时候，都不会做隐式转换

如果要对不同类型的数据进行计算，必须要显式的转换

```
let x2 = 100
let y2 = 10.5

let num1 = Double(x2) + y2
let num2 = x2 + Int(y2)
```

let & var 的选择

- 应该尽量先选择常量，只有在必须修改时，才需要修改为 var
- 在 Xcode 7.0 中，如果没有修改变量，Xcode 会提示修改为 let

Optional 可选值

- `Optional` 是 Swift 的一大特色，也是 Swift 初学者最容易困惑的问题
- 定义变量时，如果指定是 `可选的`，表示该变量 可以有一个指定类型的值，也可以是 `nil`
- 定义变量时，在类型后面添加一个 `?`，表示该变量是可选的
- 变量可选项的默认值是 `nil`
- 常量可选项没有默认值，主要用于在构造函数中给常量设置初始数值

```
//: num 可以是一个整数，也可以是 nil，注意如果为 nil，不能参与计算
let num: Int? = 10
```

- 如果 `Optional` 值是 `nil`，不允许参与计算
- 只有 `解包(unwrap)` 后才能参与计算
- 在变量后添加一个 `!`，可以强行解包

注意：必须要确保解包后的值不是 `nil`，否则会报错

```
//: num 可以是一个整数，也可以是 nil，注意如果为 nil，不能参与计算
let num: Int? = 10

//: 如果 num 为 nil，使用 `!` 强行解包会报错
let r1 = num! + 100

//: 使用以下判断，当 num 为 nil 时，if 分支中的代码不会执行
if let n = num {
    let r = n + 10
}
```

常见错误

```
unexpectedly found nil while unwrapping an Optional value
```

翻译

在[解包]一个可选值时发现 `nil`

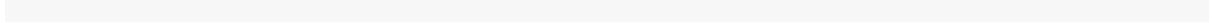
?? 运算符

- `??` 运算符可以用于判断 变量/常量 的数值是否是 `nil`，如果是则使用后面的值替代
- 在使用 Swift 开发时，`??` 能够简化代码的编写

```
let num: Int? = nil

let r1 = (num ?? 0) + 10
print(r1)
```

Optional



控制流

if

- Swift 中没有 C 语言中的 非零即真 概念
- 在逻辑判断时必须显示地指明具体的判断条件 `true / false`
- `if` 语句条件的 `()` 可以省略
- 但是 `{}` 不能省略

```
let num = 200
if num < 10 {
    print("比 10 小")
} else if num > 100 {
    print("比 100 大")
} else {
    print("10 ~ 100 之间的数字")
}
```

三目运算

- Swift 中的 三目 运算保持了和 OC 一致的风格

```
var a = 10
var b = 20
let c = a > b ? a : b
print(c)
```

适当地运用三目，能够让代码写得更加简洁

可选项判断

- 由于可选项的内容可能为 `nil`，而一旦为 `nil` 则不允许参与计算
- 因此在实际开发中，经常需要判断可选项的内容是否为 `nil`

单个可选项判断

```
let url = NSURL(string: "http://www.baidu.com")

//: 方法1: 强行解包 - 缺陷, 如果 url 为空, 运行时会崩溃
let request = NSURLRequest(URL: url!)

//: 方法2: 首先判断 - 代码中仍然需要使用 `!` 强行解包
if url != nil {
    let request = NSURLRequest(URL: url!)
```

```

}

//: 方法3: 使用 `if let`，这种方式，表明一旦进入 if 分支，u 就不再是可选项
if let u = url where u.host == "www.baidu.com" {
    let request = URLRequest(URL: u)
}

```

可选项条件判断

```

//: 1> 初学 swift 一不小心就会让 if 的嵌套层次很深，让代码变得很丑陋
if let u = url {
    if u.host == "www.baidu.com" {
        let request = URLRequest(URL: u)
    }
}

//: 2> 使用 where 关键字,
if let u = url where u.host == "www.baidu.com" {
    let request = URLRequest(URL: u)
}

```

- 小结
 - `if let` 不能与使用 `&&`、`||` 等条件判断
 - 如果要增加条件，可以使用 `where` 子句
 - 注意：`where` 子句没有智能提示

多个可选项判断

```

//: 3> 可以使用 `，` 同时判断多个可选项是否为空
let oName: String? = "张三"
let oNo: Int? = 100

if let name = oName {
    if let no = oNo {
        print("姓名：" + name + " 学号：" + String(no))
    }
}

if let name = oName, let no = oNo {
    print("姓名：" + name + " 学号：" + String(no))
}

```

判断之后对变量需要修改

```

let oName: String? = "张三"
let oNum: Int? = 18

```

```

if var name = oName, num = oNum {
    name = "李四"
    num = 1

    print(name, num)
}

```

guard

- `guard` 是与 `if let` 相反的语法，Swift 2.0 推出的

```

let oName: String? = "张三"
let oNum: Int? = 18

guard let name = oName else {
    print("name 为空")
    return
}

guard let num = oNum else {
    print("num 为空")
    return
}

// 代码执行至此，name & num 都是有值的
print(name)
print(num)

```

- 在程序编写时，条件检测之后的代码相对是比较复杂的
- 使用 `guard` 的好处
 - 能够判断每一个值
 - 在真正的代码逻辑部分，省略了一层嵌套

switch

- `switch` 不再局限于整数
- `switch` 可以针对 任意数据类型 进行判断
- 不再需要 `break`
- 每一个 `case` 后面必须有可以执行的语句
- 要保证处理所有可能的情况，不然编译器直接报错，不处理的条件可以放在 `default` 分支中
- 每一个 `case` 中定义的变量仅在当前 `case` 中有效，而 OC 中需要使用 `{}`

```

let score = "优"

switch score {

```

```

case "优":
    let name = "学生"
    print(name + "80~100分")
case "良": print("70~80分")
case "中": print("60~70分")
case "差": print("不及格")
default: break
}

```

- switch 中同样能够赋值和使用 where 子句

```

let point = CGPoint(x: 10, y: 10)
switch point {
case let p where p.x == 0 && p.y == 0:
    print("中心点")
case let p where p.x == 0:
    print("Y轴")
case let p where p.y == 0:
    print("X轴")
case let p where abs(p.x) == abs(p.y):
    print("对角线")
default:
    print("其他")
}

```

- 如果只希望进行条件判断，赋值部分可以省略

```

switch score {
case _ where score > 80: print("优")
case _ where score > 60: print("及格")
default: print("其他")
}

```

for 循环

- OC 风格的循环

```
var sum = 0
for var i = 0; i < 10; i++ {
    sum += i
}
print(sum)
```

- for-in , 0..<10 表示从0到9

```
sum = 0
for i in 0..<10 {
    sum += i
}
print(sum)
```

- 范围 0...10 表示从0到10

```
sum = 0
for i in 0...10 {
    sum += i
}
print(sum)
```

- 省略下标
 - _ 能够匹配任意类型
 - _ 表示忽略对应位置的值

```
for _ in 0...10 {
    print("hello")
}
```

字符串

在 Swift 中绝大多数的情况下，推荐使用 `String` 类型

- `String` 是一个结构体，性能更高
 - `String` 目前具有了绝大多数 `NSString` 的功能
 - `String` 支持直接遍历
- `NSString` 是一个 OC 对象，性能略差
- Swift 提供了 `String` 和 `NSString` 之间的无缝转换

字符串演练

- 遍历字符串中的字符

```
for s in str.characters {
    print(s)
}
```

- 字符串长度

```
// 返回以字节为单位的字符串长度，一个中文占 3 个字节
let len1 = str.lengthOfBytesUsingEncoding(NSUTF8StringEncoding)
// 返回实际字符的个数
let len2 = str.characters.count
// 返回 utf8 编码长度
let len3 = str.utf8.count
```

- 字符串拼接
 - 直接在 "" 中使用 \(变量名) 的方式可以快速拼接字符串

```
let str1 = "Hello"
let str2 = "World"
let i = 32
str = "\(i) 个 " + str1 + " " + str2
```

终于再也不用考虑 `stringWithFormat` 了 :D

- 可选项的拼接
 - 如果变量是可选项，拼接的结果中会有 `Optional`
 - 为了应对强行解包存在的风险，苹果提供了 `??` 操作符
 - `??` 操作符用于检测可选项是否为 `nil`
 - 如果不是 `nil`，使用当前值
 - 如果是 `nil`，使用后面的值替代

```
let str1 = "Hello"
let str2 = "World"
let i: Int? = 32
str = "\((i ?? 0) 个 " + str1 + " " + str2
```

- 格式化字符串

- 在实际开发中，如果需要指定字符串格式，可以使用 `String(format:....)` 的方式

```
let h = 8
let m = 23
let s = 9
let timeString = String(format: "%02d:%02d:%02d", arguments: [h, m, s])
let timeStr = String(format: "%02d:%02d:%02d", h, m, s)
```

String & Range 的结合

- 在 Swift 中，`String` 和 `Range` 连用时，语法结构比较复杂
- 如果不习惯 Swift 的语法，可以将字符串转换成 `NSString` 再处理

```
let helloString = "我们一起去飞"
(helloString as NSString).substringWithRange(NSMakeRange(2, 3))
```

- 使用 `Range <Index>` 的写法

```
let startIndex = helloString.startIndex.advancedBy(0)
let endIndex = helloString.endIndex.advancedBy(-1)

helloString.substringWithRange(startIndex..

```

集合

数组

- 数组使用 `[]` 定义，这一点与 OC 相同

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

- 遍历

```
for num in numbers {
    print(num)
}
```

- 通过下标获取指定项内容

```
let num1 = numbers[0]
let num2 = numbers[1]
```

- 可变&不可变
 - `let` 定义不可变数组
 - `var` 定义可变数组

```
let array = ["zhangsan", "lisi"]
//: 不能向不可变数组中追加内容
//array.append("wangwu")
var array1 = ["zhangsan", "lisi"]

//: 向可变数组中追加内容
array1.append("wangwu")
```

- 数组的类型

- 如果初始化时，所有内容类型一致，那么数组中保存的是该类型的内容
- 如果初始化时，所有内容类型不一致，那么数组中保存的是 `NSObject`

```
//: array1 仅允许追加 String 类型的值
//array1.append(18)

var array2 = ["zhangsan", 18]
//: 在 Swift 中，数字可以直接添加到集合，不需要再转换成 `NSNumber`  

array2.append(100)
//: 在 Swift 中，如果将结构体对象添加到集合，仍然需要转换成 `NSValue`  

array2.append(NSValue(CGPoint: CGPoint(x: 10, y: 10)))
```

- 数组的定义和实例化
 - 使用 `[]` 可以只定义数组的类型
 - 实例化之前不允许添加值
 - 使用 `[类型]()` 可以实例化一个空的数组

```
var array3: [String]
//: 实例化之前不允许添加值
//array3.append("laowang")
//: 实例化一个空的数组
array3 = [String]()
array3.append("laowang")
```

- 数组的合并
 - 必须是相同类型的数组才能够合并
 - 开发中，通常数组中保存的对象类型都是一样的！

```
array3 += array1

//: 必须是相同类型的数组才能够合并，以下两句代码都是不允许的
//array3 += array2
//array2 += array3
```

- 数组的删除

```
//: 删除指定位置的元素
array3.removeAtIndex(3)
//: 清空数组
array3.removeAll()
```

- 内存分配
 - 如果向数组中追加元素，超过了容量，会直接在现有容量基础上 * 2

```
var list = [Int]()

for i in 0...16 {
    list.append(i)
    print("添加 \(i) 容量 \(list.capacity)")
}
```

字典

- 定义
 - 同样使用 `[]` 定义字典
 - `let` 不可变字典

- `var` 可变字典
- `[String : NSObject]` 是最常用的字典类型

```
//: [String : NSObject] 是最常用的字典类型
var dict = ["name": "zhangsan", "age": 18]
```

- 赋值

- 赋值直接使用 `dict[key] = value` 格式
- 如果 `key` 不存在，会设置新值
- 如果 `key` 存在，会覆盖现有值

```
//: * 如果 key 不存在，会设置新值
dict["title"] = "boss"
//: * 如果 key 存在，会覆盖现有值
dict["name"] = "lisi"
dict
```

- 遍历

- `k`, `v` 可以随便写
- 前面的是 `key`
- 后面的是 `value`

```
//: 遍历
for (k, v) in dict {
    print("\(k) ~~~ \(v)")
}
```

- 合并字典

- 如果 `key` 不存在，会建立新值，否则会覆盖现有值

```
//: 合并字典
var dict1 = [String: NSObject]()
dict1["nickname"] = "大老虎"
dict1["age"] = 100

//: 如果 key 不存在，会建立新值，否则会覆盖现有值
for (k, v) in dict1 {
    dict[k] = v
}
print(dict)
```

函数

目标

- 函数
 - 定义格式
 - 外部参数
 - 无返回值的三种情况
- 闭包
 - 闭包的定义
 - 尾随闭包
 - 循环引用
 - OC Block复习

函数

目标

- 掌握函数的定义
- 掌握外部参数的用处
- 掌握无返回类型的三种函数定义方式

代码实现

- 函数的定义
 - 格式 `func 函数名(形参列表) -> 返回值 {代码实现}`
 - 调用 有以下三种版本：
 - Swift 1.0 `let result = 函数名(值1, 值2...)` 所有的形参都会省略，其他的程序员非常喜欢！
 - Swift 2.0 `let result = 函数名(值1, 参数2: 值2...)` 第一个形参的名称省略
 - Swift 3.0 `let result = 函数名(参数1: 值1, 参数2: 值2...)` 调用的方式 -> OC的程序员非常喜欢

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

let result = sum(a: 10, b: 20)
```

- 没有返回值的函数，一共有三种写法
 - 省略
 - `()`
 - `Void`

```
func demo(str: String) -> Void {
    print(str)
}

func demo1(str: String) -> () {
    print(str)
}

func demo2(str: String) {
    print(str)
}

demo("hello")
demo1("hello world")
```

```
demo2("olleh")
```

- 外部参数

- 在形参名前再增加一个外部参数名，能够方便调用人员更好地理解函数的语义
- 格式： func 函数名(外部参数名 形式参数名: 形式参数类型) -> 返回值类型 { // 代码实现 }
- Swift 2.0 中，默认第一个参数名省略
- Swift 3.0 中不默认第一个参数名省略

```
func sum1(num1 a: Int, num2 b: Int) -> Int {  
    return a + b  
}  
  
sum1(num1: 10, num2: 20)
```

闭包

与 OC 中的 Block 类似，闭包主要用于异步操作执行完成后的代码回调，网络访问结果以参数的形式传递给调用方

目标

- 掌握闭包的定义
- 掌握闭包的概念和用法
- 了解尾随闭包的写法
- 掌握解除循环引用的方法

OC 中 Block 概念回顾

- 闭包类似于 OC 中的 Block
 - 预先定义好的代码
 - 在需要时执行
 - 可以当作参数传递
 - 可以有返回值
 - 包含 self 时需要注意循环引用

闭包的定义

- 定义一个函数

```
//: 定义一个 sum 函数
func sum(num1 num1: Int, num2: Int) -> Int {
    return num1 + num2
}
sum(num1: 10, num2: 30)

//: 在 Swift 中函数本身就可以当作参数被定义和传递
let mySum = sum
let result = mySum(num1: 20, num2: 30)
```

- 定义一个闭包

- 闭包 = { (行参) -> 返回值 in // 代码实现 }
- in 用于区分函数定义和代码实现

```
//: 闭包 = { (行参) -> 返回值 in // 代码实现 }
let sumFunc = { (num1 x: Int, num2 y: Int) -> Int in
    return x + y
}
sumFunc(num1: 10, num2: 20)
```

- 最简单的闭包，如果没有参数/返回值，则 参数/返回值/in 统统都可以省略
 - { 代码实现 }

```
let demoFunc = {
    print("hello")
}
```

基本使用

GCD 异步

- 模拟在后台线程加载数据

```
func loadData() {
    DispatchQueue.global().async { () -> Void in
        print("耗时操作 \(NSThread.currentThread())")
    })
}
```

- 尾随闭包，如果闭包是最后一个参数，可以用以下写法
- 注意上下两段代码，`}` 的位置

```
func loadData() {
    DispatchQueue.global().async { () -> Void in
        print("耗时操作 \(NSThread.currentThread())")
    }
}
```

- 闭包的简写，如果闭包中没有参数和返回值，可以省略

```
func loadData() {
    DispatchQueue.global().async {
        print("耗时操作 \(NSThread.currentThread())")
    }
}
```

自定义闭包参数，实现主线程回调

- 添加没有参数，没有返回值的闭包

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

    loadData {
        print("完成回调")
    }
}

// MARK: - 自定义闭包参数
func loadData(finished: ()->()) {

    DispatchQueue.global().async {
```

```

print("耗时操作 \u00d7(NSThread.currentThread())")

DispatchQueue.main.async(execute: {
    print("主线程回调 \u00d7(NSThread.currentThread())")

    // 执行回调
    finished()
})

}

}

```

- 添加回调参数
- 嵌套的 GCD Xcode 不会改成尾随闭包

```

override func viewDidLoad() {
    super.viewDidLoad()

    loadData4 { (html) -> () in
        print(html)
    }
}

/// 加载数据
/// 完成回调 - 传入回调闭包，接收异步执行的结果
func loadData4(finished: @escaping (_ html: String) -> ()) {

    DispatchQueue.global().async {
        print("加载数据 \u00d7(NSThread.currentThread())")

        DispatchQueue.main.async(execute: {
            print("完成回调 \u00d7(NSThread.currentThread())")

            finished(html: "<h1>hello world</h1>")
        })
    }
}

```

循环引用

- 建立 NetworkTools 对象

```
class NetworkTools: NSObject {

    /// 加载数据
    ///
    /// - parameter finished: 完成回调
    func loadData(finished: @escaping ()->()) {
        print("开始加载数据...")

        // ...
        finished()
    }

    deinit {
        print("网络工具 88")
    }
}
```

- 实例化 NetworkTools 并且加载数据

```
class ViewController: UIViewController {

    var tools: NetworkTools?

    override func viewDidLoad() {
        super.viewDidLoad()

        tools = NetworkTools()
        tools?.loadData() {
            print("come here \(self.view)")
        }
    }

    /// 与 OC 中的 dealloc 类似，注意此函数没有()
    deinit {
        print("控制器 88 了!")
    }
}
```

运行不会形成循环引用，因为 loadData 执行完毕后，就会释放对 self 的引用

- 修改 NetworkTools，定义回调闭包属性

```
/// 完成回调属性
```

```

var finishedCallBack: ((()->())?

/// 加载数据
///
/// - parameter finished: 完成回调
func loadData(finished: @escaping () -> () -> ()) {

    self.finishedCallBack = finished

    print("开始加载数据...")

    // ...
    working()
}

func working() {
    finishedCallBack?()
}

deinit {
    print("网络工具 88")
}

```

运行测试，会出现循环引用

解除循环引用

- 与 OC 类似的方法

```

/// 类似于 OC 的解除引用
func demo() {
    weak var weakSelf = self
    tools?.loadData() {
        print("\(weakSelf?.view)")
    }
}

```

- Swift 推荐的方法

```

loadData { [weak self] in
    print("\(self?.view)")
}

```

- 还可以

```

loadData { [unowned self] in
    print("\(self.view)")
}

```

闭包(Block) 的循环引用小结

- Swift

- [weak self]
 - self 是可选项，如果self已经被释放，则为 nil
- [unowned self]
 - self 不是可选项，如果self已经被释放，则出现 野指针访问

- Objc

- __weak typeof(self) weakSelf;
 - 如果 self 已经被释放，则为 nil
- __unsafe_unretained typeof(self) weakSelf;
 - 如果 self 已经被释放，则出现野指针访问

面向对象

目标

- 构造函数
 - 构造函数的基本概念
 - 构造函数的执行顺序
 - KVC 在构造函数中的使用及原理
 - 便利构造函数
 - 析构函数
 - 区分 重载 和 重写
- 懒加载
- 只读属性（计算型属性）
- 设置模型数据（`didSet`）

构造函数基础

构造函数 是一种特殊的函数，主要用来在创建对象时初始化对象，为对象 成员变量 设置初始值，在 OC 中的构造函数是 `initWithXXX`，在 Swift 中由于支持函数重载，所有的构造函数都是 `init`

构造函数的作用

- 分配空间 `alloc`
- 设置初始值 `init`

必选属性

- 自定义 `Person` 对象

```
class Person: NSObject {

    /// 姓名
    var name: String
    /// 年龄
    var age: Int
}
```

提示错误 `Class 'Person' has no initializers -> 'Person'` 类没有实例化器

原因：如果一个类中定义了必选属性，必须通过构造函数为这些必选属性分配空间并且设置初始值

- 重写 父类的构造函数

```
/// `重写`父类的构造函数
override init() {

}
```

提示错误 `Property 'self.name' not initialized at implicitly generated super.init call -> 属性 'self.name' 没有在隐式生成的 super.init 调用前被初始化`

- 手动添加 `super.init()` 调用

```
/// `重写`父类的构造函数
override init() {
    super.init()
}
```

提示错误 `Property 'self.name' not initialized at super.init call -> 属性 'self.name' 没有在 super.init 调用前被初始化`

- 为必选属性设置初始值

```
// `重写`父类的构造函数
override init() {
    name = "张三"
    age = 18

    super.init()
}
```

小结

- 非 Optional 属性，都必须在构造函数中设置初始值，从而保证对象在被实例化的时候，属性都被正确初始化
- 在调用父类构造函数之前，必须保证本类的属性都已经完成初始化
- Swift 中的构造函数不用写 func

子类的构造函数

- 自定义子类时，需要在构造函数中，首先为本类定义的属性设置初始值
- 然后再调用父类的构造函数，初始化父类中定义的属性

```
// 学生类
class Student: Person {

    /// 学号
    var no: String

    override init() {
        no = "001"

        super.init()
    }
}
```

小结

- 先调用本类的构造函数初始化本类的属性
- 然后调用父类的构造函数初始化父类的属性
- Xcode 7 beta 5之后，父类的构造函数会被自动调用，强烈建议写 super.init()，保持代码执行线索的可读性
- super.init() 必须放在本类属性初始化的后面，保证本类属性全部初始化完成

Optional 属性

- 将对象属性类型设置为 `Optional`

```
class Person: NSObject {  
    /// 姓名  
    var name: String?  
    /// 年龄  
    var age: Int?  
}
```

- 可选属性 不需要设置初始值，默认初始值都是 `nil`
- 可选属性 是在设置数值的时候才分配空间的，是延迟分配空间的，更加符合移动开发中延迟创建的原则

重载构造函数

- Swift 中支持函数重载，同样的函数名，不一样的参数类型

```
/// `重载`构造函数
///
/// - parameter name: 姓名
/// - parameter age: 年龄
///
/// - returns: Person 对象
init(name: String, age: Int) {
    self.name = name
    self.age = age

    super.init()
}
```

注意事项

- 如果重载了构造函数，但是没有实现默认的构造函数 `init()`，则系统不再提供默认的构造函数
- 原因，在实例化对象时，必须通过构造函数为对象属性分配空间和设置初始值，对于存在必选参数的类而言，默认的 `init()` 无法完成分配空间和设置初始值的工作

调整子类的构造函数

- 重写 父类的构造函数

```
/// `重写`父类构造函数
///
/// - parameter name: 姓名
/// - parameter age: 年龄
///
/// - returns: Student 对象
override init(name: String, age: Int) {
    no = "002"

    super.init(name: name, age: age)
}
```

- 重载 构造函数

```
/// `重载`构造函数
///
/// - parameter name: 姓名
/// - parameter age: 年龄
```

```
/// - parameter no: 学号
///
/// - returns: Student 对象
init(name: String, age: Int, no: String) {
    self.no = no

    super.init(name: name, age: age)
}
```

注意：如果是重载的构造函数，必须 `super` 以完成父类属性的初始化工作

重载 和 重写

- 重载，函数名相同，参数名 / 参数类型 / 参数个数不同
 - 重载函数并不仅仅局限于 构造函数
 - 函数重载是面向对象程序设计语言的重要标志
 - 函数重载能够简化程序员的记忆
 - OC 不支持函数重载，OC 的替代方式是 `withXXX...`
- 重写，子类需要在父类拥有方法的基础上进行扩展，需要 `override` 关键字

KVC 字典转模型构造函数

```
/// `重写`构造函数
///
/// - parameter dict: 字典
///
/// - returns: Person 对象
init(dict: [String: AnyObject]) {
    setValuesForKeys(dict)
}
```

- 以上代码编译就会报错！
- 原因：
 - KVC 是 OC 特有的，KVC 本质上是在 `运行时`，动态向对象发送 `setValueForKey:` 方法，为对象的属性设置数值
 - 因此，在使用 KVC 方法之前，需要确保对象已经被正确 实例化
- 添加 `super.init()` 同样会报错
- 原因：
 - 必选属性 必须在调用父类构造函数之前完成初始化分配工作
- 将必选参数修改为可选参数，调整后的代码如下：

```
/// 个人模型
class Person: NSObject {

    /// 姓名
    var name: String?
    /// 年龄
    var age: Int?

    /// `重写`构造函数
    ///
    /// - parameter dict: 字典
    ///
    /// - returns: Person 对象
    init(dict: [String: AnyObject]) {
        super.init()

        setValuesForKeys(dict)
    }
}
```

运行测试，仍然会报错

错误信息: `this class is not key value coding-compliant for the key age.` -> 这个类的键值 `age` 与 键值编码不兼容

- 原因:
 - 在 Swift 中, 如果属性是可选的, 在初始化时, 不会为该属性分配空间
 - 而 OC 中基本数据类型就是保存一个数值, 不存在 可选 的概念
- 解决办法: 给基本数据类型设置初始值
- 修改后的代码如下:

```
/// 姓名
var name: String?
/// 年龄
var age: Int? = 0

/// `重写`构造函数
///
/// - parameter dict: 字典
///
/// - returns: Person 对象
init(dict: [String: AnyObject]) {
    super.init()

    setValuesForKeys(dict)
}
```

提示: 在定义类时, 基本数据类型属性一定要设置初始值, 否则无法正常使用 KVC 设置数值

KVC 函数调用顺序

```
init(dict: [String: AnyObject]) {
    super.init()

    setValuesForKeys(dict)
}

override func setValue(value: AnyObject?, forKey key: String) {
    print("Key \(key) \(value)")

    super.setValue(value, forKey: key)
}

// `NSObject` 默认在发现没有定义的键值时, 会抛出 `NSUndefinedKeyException` 异常
override func setValue(value: AnyObject?, forUndefinedKey key: String) {
    print("UndefinedKey \(key) \(value)")
}
```

- `setValuesForKeys` 会按照字典中的 `key` 重复调用 `setValue(forKey:)` 函数

- 如果没有实现 `forUndefinedKey` 函数，程序会直接崩溃
 - `NSObject` 默认在发现没有定义的键值时，会抛出 `NSUndefinedKeyException` 异常
- 如果实现了 `forUndefinedKey`，会保证 `setValuesForKeys` 继续遍历后续的 `key`
- 如果父类实现了 `forUndefinedKey`，子类可以不必再实现此函数

子类的 KVC 函数

```
/// 学生类
class Student: Person {

    /// 学号
    var no: String?
}
```

- 如果父类中已经实现了父类的相关方法，子类中不用再实现相关方法

convenience 便利构造函数

- 默认情况下，所有的构造方法都是指定构造函数 Designated
- `convenience` 关键字修饰的构造方法就是便利构造函数
- 便利构造函数具有以下特点：
 - 可以返回 `nil`
 - 只有便利构造函数中可以调用 `self.init()`
 - 便利构造函数不能被重写 或者 `super`

```
/// `便利构造函数`  
///  
/// - parameter name: 姓名  
/// - parameter age: 年龄  
///  
/// - returns: Person 对象, 如果年龄过小或者过大, 返回 nil  
convenience init?(name: String, age: Int) {  
    if age < 20 || age > 100 {  
        return nil  
    }  
  
    self.init(dict: ["name": name, "age": age])  
}
```

注意：在 Xcode 中，输入 `self.init` 时没有智能提示

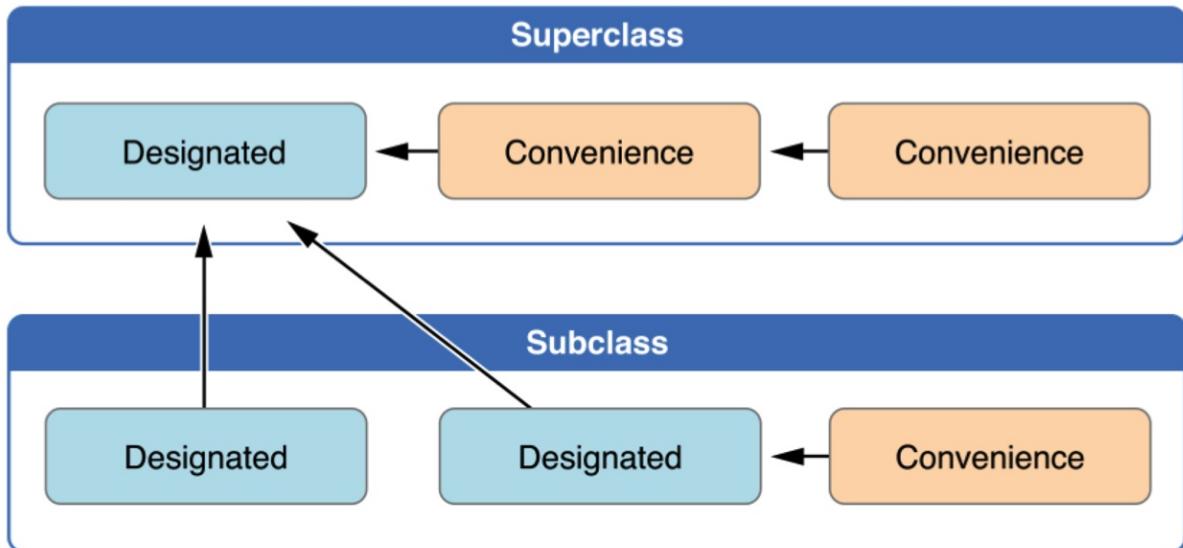
```
/// 学生类  
class Student: Person {  
  
    /// 学号  
    var no: String?  
  
    convenience init?(name: String, age: Int, no: String) {  
        self.init(name: name, age: age)  
  
        self.no = no  
    }  
}
```

便利构造函数应用场景

- 根据给定参数判断是否创建对象，而不像指定构造函数那样必须要实例化一个对象出来
- 在实际开发中，可以对已有类的构造函数进行扩展，利用便利构造函数，简化对象的创建

构造函数小结

- 指定构造函数必须调用其直接父类的指定构造函数（除非没有父类）
- 便利构造函数必须调用同一类中定义的其他 指定构造函数 或者用 `self.` 的方式调用 父类的便利构造函数
- 便利构造函数可以返回 `nil`
- 便利构造函数不能被继承



懒加载

在 iOS 开发中，懒加载是无处不在的

- 懒加载的格式如下：

```
lazy var person: Person = {
    print("懒加载")
    return Person()
}()
```

- 懒加载本质上是一个闭包
- 完整写法如下：供参考
 - { } 包装代码
 - () 执行代码

日常开发：

1. 闭包中的智能提示不好
2. 闭包中如果出现 self. 还需要注意循环引用

```
lazy var label = { () -> UILabel in
    let l = UILabel()
    // 设置 label 的属性...
    return l
}()
```

- 以上 Person 代码可以改写为以下格式

```
let personFunc = { () -> Person in
    print("懒加载")
    return Person()
}
lazy var demoPerson: Person = self.personFunc()
```

- 懒加载的简单写法

```
lazy var demoPerson: Person = Person()
```

只读属性(计算型属性)

getter & setter

- 在 Swift 中 `getter & setter` 很少用，以下代码仅供了解

```
private var _name: String?  
var name: String? {  
    get {  
        return _name  
    }  
    set {  
        _name = newValue  
    }  
}
```

存储型属性 & 计算型属性

- 存储型属性 - 需要开辟空间，以存储数据
- 计算型属性 - 执行函数返回其他内存地址

```
var title: String {  
    get {  
        return "Mr " + (name ?? "")  
    }  
}
```

- 只实现 `getter` 方法的属性被称为计算型属性，等同于 OC 中的 `ReadOnly` 属性
- 计算型属性本身不占用内存空间
- 不可以给计算型属性设置数值
- 计算型属性可以使用以下代码简写

```
var title: String {  
    return "Mr " + (name ?? "")  
}
```

计算型属性与懒加载的对比

- 计算型属性
 - 不分配独立的存储空间保存计算结果
 - 每次调用时都会被执行
 - 更像一个函数，不过不能接收参数，同时必须有返回值

```
var title2: String {  
    return "Mr" + (name ?? "")  
}
```

- 懒加载属性
 - 在第一次调用时，执行闭包并且分配空间存储闭包返回的数值
 - 会分配独立的存储空间
 - 与 OC 不同的是，lazy 属性即使被设置为 nil 也不会被再次调用

```
lazy var title: String = {  
    return "Mr " + (self.name ?? "")  
}()
```

Xcode 7 中的网络请求

ATS 应用传输安全

App Transport Security (ATS) lets an app add a declaration to its Info.plist file that specifies the domains with which it needs secure communication. ATS prevents accidental disclosure, provides secure default behavior, and is easy to adopt. You should adopt ATS as soon as possible, regardless of whether you're creating a new app or updating an existing one. If you're developing a new app, you should use HTTPS exclusively. If you have an existing app, you should use HTTPS as much as you can right now, and create a plan for migrating the rest of your app as soon as possible.

强制访问

```
<key>NSAppTransportSecurity</key>
<dict>
    <!--Include to allow all connections (DANGER)-->
    <key>NSAllowsArbitraryLoads</key>
        <true/>
</dict>
```

设置白名单

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
    <dict>
        <key>localhost</key>
        <dict>
            <key>NSTemporaryExceptionAllowsInsecureHTTPLoads</key>
            <true/>
        </dict>
    </dict>
</dict>
```

- 网络访问代码

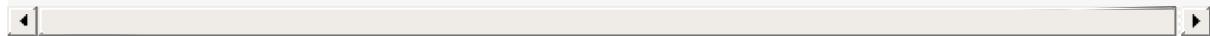
```
let url = NSURL(string: "http://www.weather.com.cn/data/sk/101010100.html")!

NSURLSession.sharedSession().dataTaskWithURL(url) { (data, _, error) in
    if error != nil {
        print(error)
        return
    }
}
```

```
}.resume()
```

- 运行提示：App Transport Security has blocked a cleartext HTTP (http://) resource load since it is insecure. Temporary exceptions can be configured via your app's Info.plist file.
- JSON 序列化

```
let dict = try! NSJSONSerialization.JSONObjectWithData(data!, options: NSJSONReadingOptions(rawValue: 0))
print(dict)
```



- try catch

```
do {
    let tmpData = "{\"name\": \"zhangsan\"}".dataUsingEncoding(NSUTF8StringEncoding)
    let dict = try NSJSONSerialization.JSONObjectWithData(tmpData!, options: NSJSONReadingOptions[])
    print(dict)
} catch {
    print(error)
}
```

项目演练

目标

- 创建一个 iOS App 熟悉 Swift 基础语法的应用
- 体会 Swift 和 OC 开发的对比

明确开发步骤

1. 创建 `TLListTableViewController`
2. 绑定数据，用闭包回调，模拟网络加载延时操作
3. 自定义 `UITableViewCell Identifier` 为 `listCellId`
4. 跳转到 `TLDetailTableViewController`
5. 传递数值，设置 `TLDetailTableViewController` 的 UI
6. 保存，使用 闭包 回调

准备工作

- 修改 `TLListTableViewController` 的类为 `UITableViewController`
- 删除 `Main.storyboard` 中的默认控制器，新增为 `UITableViewController`
- 设置为初始控制器
- 设置 Cell 的可重用标识符 `listCellId`
- 指定 `UITableViewController` 的类为 `TLListTableViewController`

```
class TLListTableViewController: UITableViewController {

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

创建模型加载数据

知识点

- `extension` 可以让代码结构更加清晰
- 闭包 (block)
 - 应用场景：异步执行完成后，通过参数传递异步执行的结果
 - 定义技巧
 - 引用当前对象必须使用 `self.`
- 重写 `description` 属性可以方便开发调试

加载数据 — `extension`

- 新建 `TLPerson.swift`

```
/// 个人信息
class TLPerson: NSObject {
    var name: String?
    var age: Int = 0

    init(dict: [String: AnyObject]) {
        super.init()

        setValuesForKeys(dict)
    }
}
```

- 编写 `loadData` 函数实现数据异步加载

```
// MARK: - 数据处理
extension TLListTableViewController {
    /// 加载数据
    private func loadData() {

        DispatchQueue.global().async {

            print("后台加载数据...")

            var array = [TLPerson]()

            // 填充数据
            for i in 0..<50 {
                let name = "张三 \(i)"
                let age = random() % 20 + 10
            }
        }
    }
}
```

```

        array.append(TLPerson(dict: ["name": name, "age": age]))
    }

    print(array)
}
}
}

```

小结

- `extension` 类似于 `oc` 的 `Category`，不过可以按照函数类型区分代码
- 能够让代码结构具有更好的可读性
- 重写 `description` 属性，便于调试

对象描述信息

```

override var description: String {
    let keys = ["name", "age"]
    return dictionaryWithValues(forKey: keys).description
}

```

加载数据 —— 完成回调

```

/// 加载数据
private func loadData(finished: (array: [TLPerson]) -> ()) {

    DispatchQueue.global().async {

        print("后台加载数据...")

        var array = [Person]()

        // 填充数据
        for i in 0..<50 {
            let name = "张三 \(i)"
            let age = random() % 20 + 10

            array.append(TLPerson(dict: ["name": name, "age": age]))
        }

        DispatchQueue.main.async(execute: {
            finished(array: array)
        })
    }
}

```

- 闭包参数的定义技巧

- 定义参数 `finished: ()->()`
- 根据需要添加闭包的参数 `finished: (array: [TLPerson])->()`
- 调用闭包，需要附带 外部参数

在 iOS 开发中，闭包(block)最常用的应用场景就是异步执行完成后，通过参数传递异步执行的结果

- 定义个人数据数组

```
/// 个人数据数组
private var persons: [TLPerson]?
```

- 调整调用 `loadData` 函数

```
loadData { (array) -> () in
    print(array)
}
```

绑定表格

- 实现数据源方法

```
// MARK: - 表格数据源方法
extension TLListTableViewController {
    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return persons?.count ?? 0
    }

    override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "listCellId", forIndexPath: indexPath)

        cell.textLabel?.text = persons![indexPath.row].name

        return cell
    }
}
```

- 调整 `loadData` 函数

```
loadData { (array) -> () in
    // 1. 记录异步结果
    self.persons = array
```

```
// 2. 刷新表格  
self.tableView.reloadData()  
}
```

- 提示

- `array` 没有智能提示
- 闭包由于是提前准备好的代码，因此在使用到当前对象属性时，一定要指明当前对象 `self.`

明细控制器

知识点

- `IBOutlet` 和 `IBAction` 连线都会连到同一个 `swift` 文件中，因此在连线时一定注意选择类型

布局

- 新建 `TLDetailTableViewController` 继承自 `UITableViewController`
- 在 `Main.storyboard` 中给 `TLDetailTableViewController` 嵌入导航控制器
- 新增 `UITableViewController` 并指定类型为 `TLDetailTableViewController`
- 从 `Cell` 连线到 `TLDetailTableViewController` 并选择 `Show`
- 新增 `Navigation Item`，并且设置标题
- 新增 `Bar Button Item` 并且设置标题为 `保存`，`Enabled` 设置为 `false`
- 新增 `姓名 & 年龄` 文本框

连线

- 连接 `IBOutlet`

```
@IBOutlet weak var nameText: UITextField!
@IBOutlet weak var ageText: UITextField!
```

- 连线 `IBAction`，将两个文本框的 `Editing Changed` 事件连线到 `textDidChange` 函数

```
/// 文本变化
@IBAction func textChange(_ sender: Any) {
    navigationItem.rightBarButtonItem?.enabled = nameText.hasText() && ageText.hasText()
}
```

- 连线保存按钮

```
/// 保存
@IBAction func savePerson(_ sender: Any) {
}
```

传递数值

- 定义模型属性，准备接收数值

```
/// 个人模型
```

```
var person: TLPerson?
```

- 在 `TLListTableViewController` 中实现 `prepare` 函数，传递参数

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // 判断目标控制器
    guard let vc = segue.destination as? TLDetailTableViewController else {
        return
    }

    // 取出当前选中行
    guard let indexPath = tableView.indexPathForSelectedRow else {
        return
    }

    // 传递数值
    vc.person = persons![indexPath.row]
}
```

- 在 `viewDidLoad` 中设置控件内容

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

    nameText.text = person?.name
    ageText.text = String(person?.age ?? 0)
}
```

- 抽取函数，设置导航按钮的激活状态

```
private func setupNavigation() {
    navigationItem.rightBarButtonItem?.enabled = nameText.hasText() && ageText.hasText()
}
```

- 分别修改 `viewDidLoad` 和 `textChange` 的函数调用
- 实现 `savePerson` 函数

```
/// 保存
@IBAction func save(sender: AnyObject) {
    person?.name = nameText.text
    person?.age = Int(ageText.text!) ?? 0

    print(person)
}
```

闭包回调

- 定义闭包回调属性

```
/// 保存个人信息回调
var savePersonCallBack: (()->())?
```

- 完成回调

```
// 保存
@IBAction func savePerson(sender: AnyObject) {
    person?.name = nameText.text
    person?.age = Int(ageText.text!) ?? 0

    print(person)

    // 完成回调
    savePersonCallBack?()

    // 控制器弹栈
    navigationController?.popViewControllerAnimated(true)
}
```

- 设置完成回调

```
override func prepare(segue: UIStoryboardSegue, sender: AnyObject?) {
    // 判断目标控制器
    guard let vc = segue.destinationViewController as? DetailViewController else {
        return
    }

    // 取出当前选中行
    guard let indexPath = tableView.indexPathForSelectedRow else {
        return
    }

    // 传递数值
    vc.person = persons![indexPath.row]
    // 完成回调
    vc.savePersonCallBack = {
        self.tableView.reloadData()
    }
}
```

- 简化完成回调

```
vc.savePersonCallBack = self.tableView.reloadData
```

