

学前须知

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





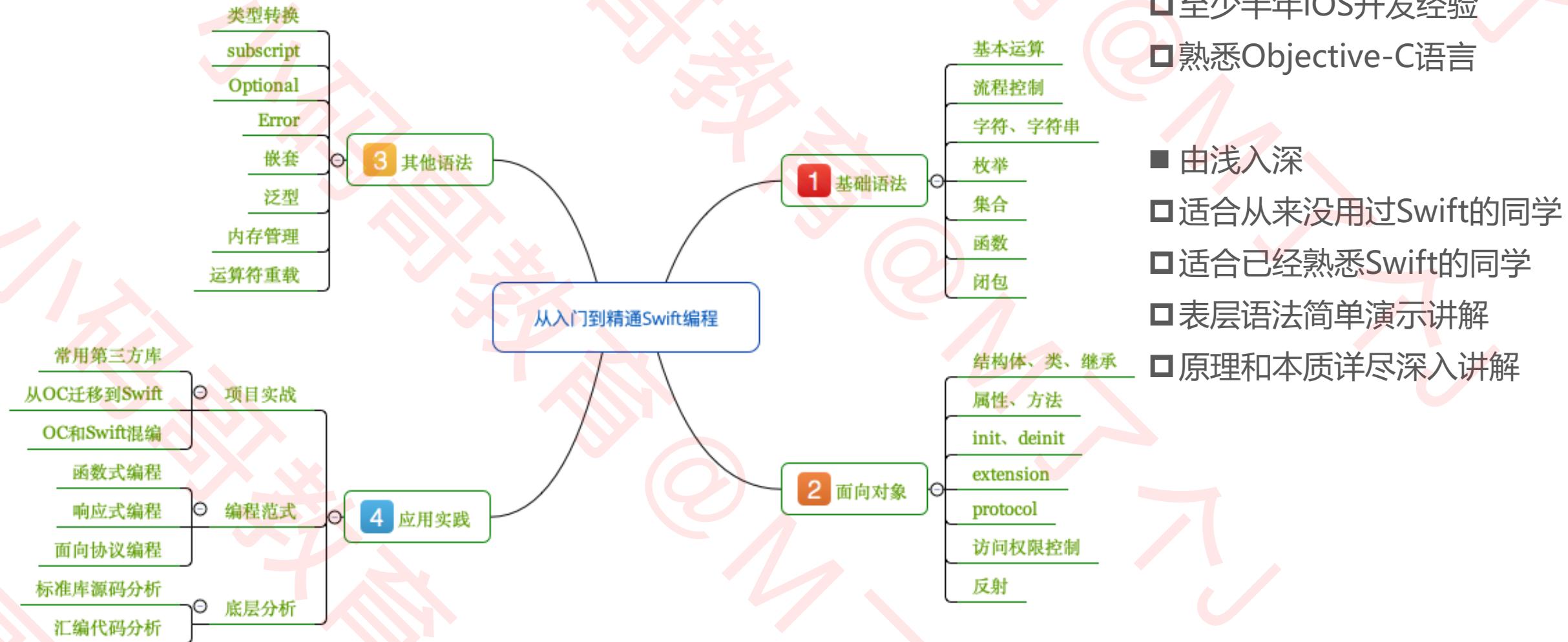
上课环境

- 基于Swift5.1
- 开发工具 : Xcode11
- 操作系统 : macOS 10.14 Mojave、macOS Catalina 10.15
- 下载地址 : <https://developer.apple.com/download/>

- 建议
- 课后用1.5~2倍速度回看视频
- 尊重他人
- 聊天讨论框只会在“适当”的时候展示
- 课程目录后期会更新

课程简介

■ 授课内容如课程大纲所示



- 学习条件
 - 至少半年iOS开发经验
 - 熟悉Objective-C语言
- 由浅入深
 - 适合从来没用过Swift的同学
 - 适合已经熟悉Swift的同学
 - 表层语法简单演示讲解
 - 原理和本质详尽深入讲解

Swift简介

- Swift是Apple在2014年6月WWDC发布的全新编程语言，中文名和LOGO是“雨燕”
- 在Swift刚发布那时，百度\Google一下Swift，出现最多的搜索结果是
 - 美国著名女歌手Taylor Swift，中国歌迷称她为“霉霉”
 - 现在的搜索结果以Swift编程语言相关的内容居多



Taylor Swift

- Swift之父Chris Lattner
- Clang编译器作者、[LLVM](#)项目的主要发起人
- 从Apple离职后，先后跳槽到Tesla、Google
- 目前在Google Brain从事AI研究



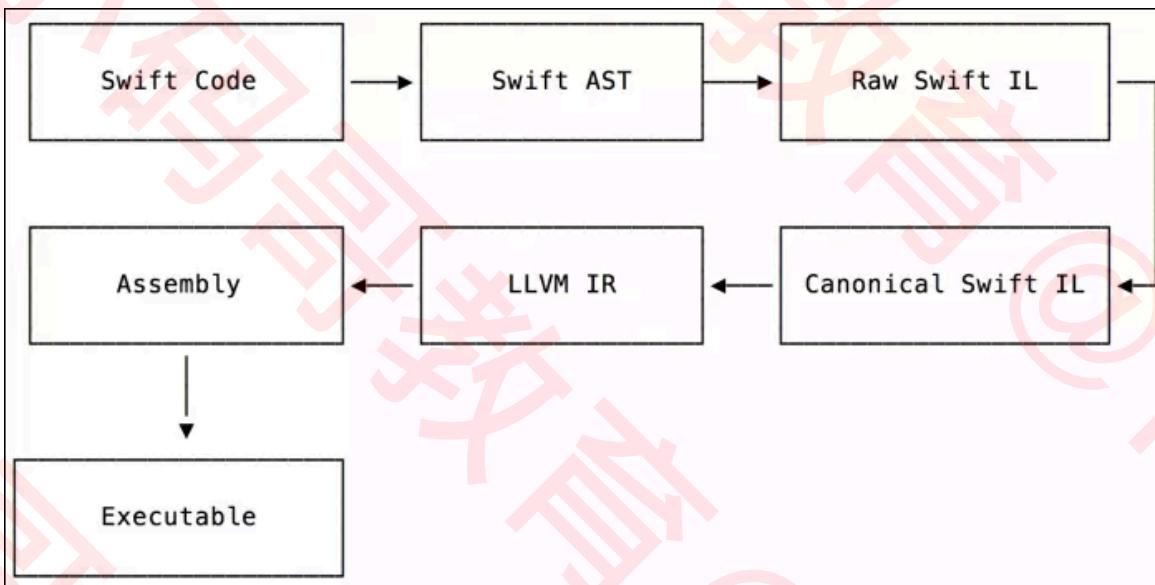
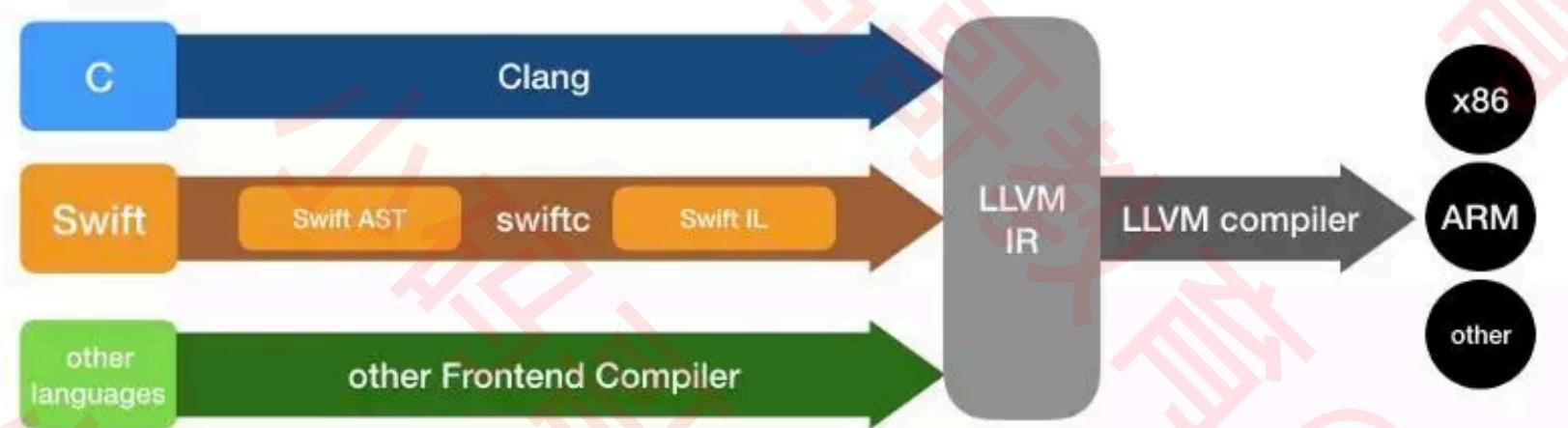
Chris Lattner

Swift版本

- 历时5年发展，从Swift1.x发展到了Swift5.x版本，经历了多次重大改变，ABI终于稳定
- API (Application Programming Interface)：应用程序编程接口
 - ✓ 源代码和库之间的接口
- ABI (Application Binary Interface)：应用程序二进制接口
 - ✓ 应用程序与操作系统之间的底层接口
 - ✓ 涉及的内容有：目标文件格式、数据类型的大小\布局\对齐、函数调用约定等等
- 随着ABI的稳定，Swift语法基本不会再有太大的变动，此时正是学习Swift的最佳时刻
- 截止至2019年6月，目前最新版本：Swift5.1
- Swift完全开源：<https://github.com/apple/swift>，主要采用C++编写



编译流程



■ 参考：<https://swift.org/compiler-stdlib>

swiftc

- swiftc存放在Xcode内部
- Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin

- 一些操作
- 生成语法树： swiftc -dump-ast main.swift
- 生成最简洁的SIL代码： swiftc -emit-sil main.swift
- 生成LLVM IR代码： swiftc -emit-ir main.swift -o main.ll
- 生成汇编代码： swiftc -emit-assembly main.swift -o main.s

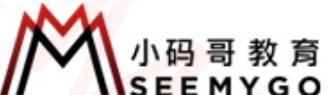
- 对汇编代码进行分析，可以真正掌握编程语言的本质

汇编语言

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



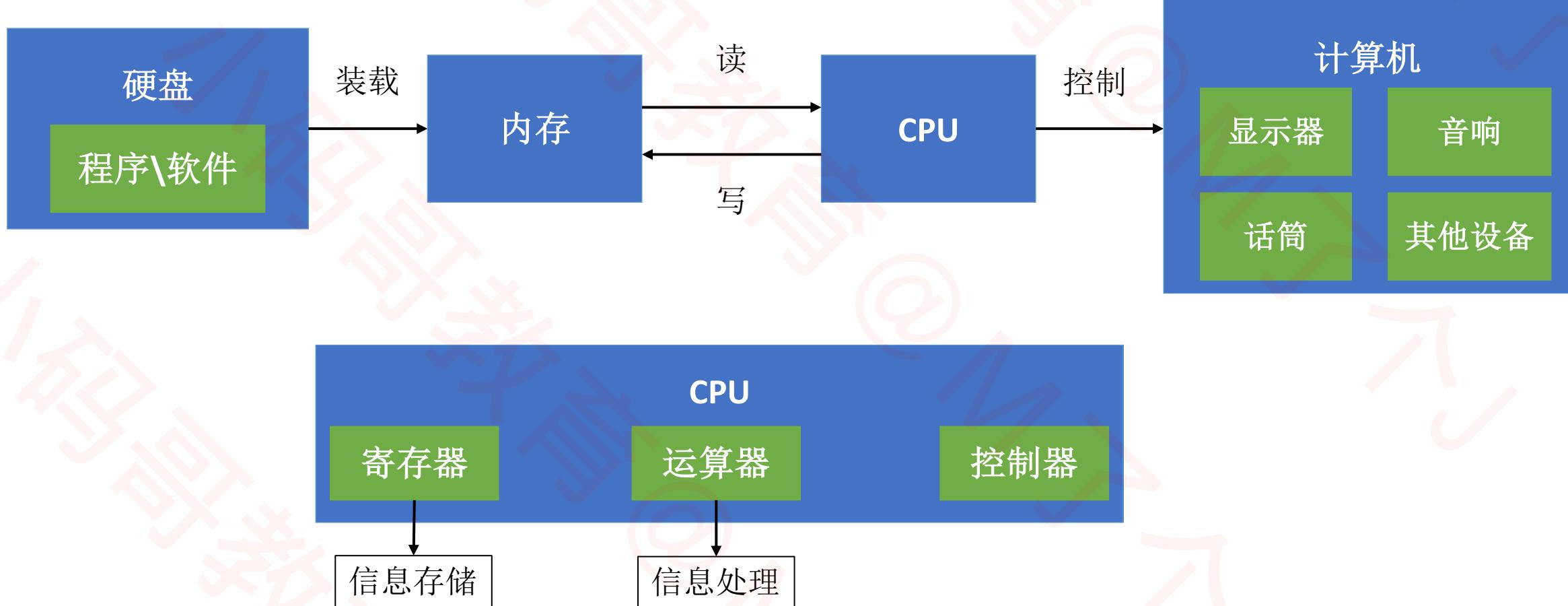
实力IT教育 www.520it.com

码拉松



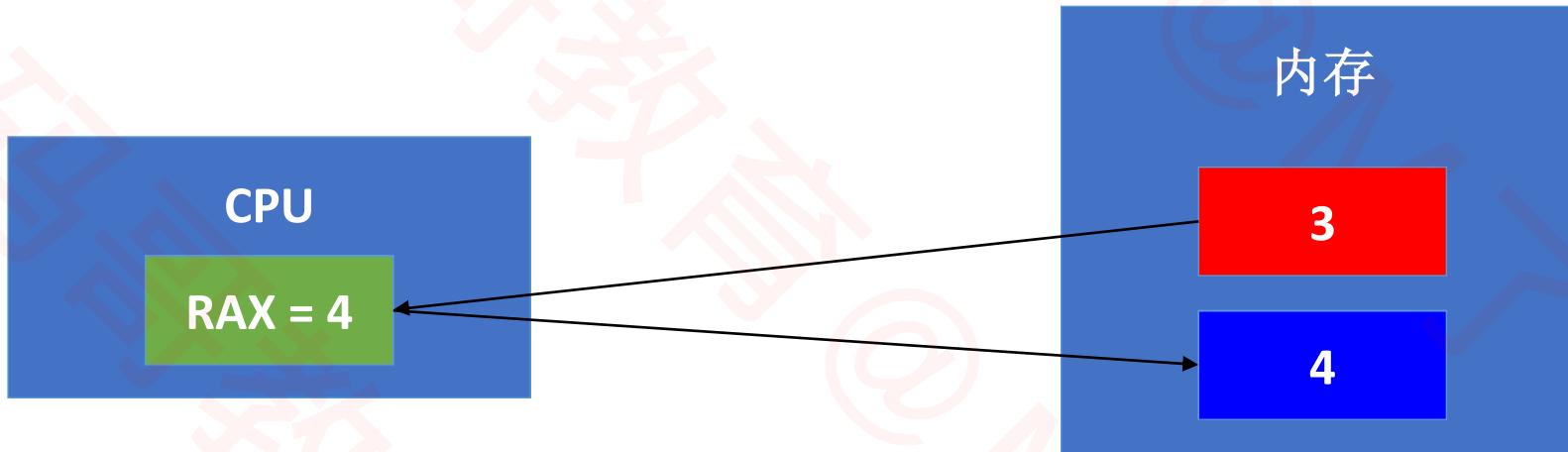
程序的本质

■ 软件\程序的执行过程



寄存器与内存

- 通常，CPU会先将内存中的数据存储到寄存器中，然后再对寄存器中的数据进行运算
- 假设内存中有块红色内存空间的值是3，现在想把它的值加1，并将结果存储到蓝色内存空间
 - CPU首先会将红色内存空间的值放到rax寄存器中：**movq 红色内存空间, %rax**
 - 然后让rax寄存器与1相加：**addq \$0x1, %rax**
 - 最后将值赋值给内存空间：**movq %rax, 蓝色内存空间**





编程语言的发展

■ 机器语言

- 由0和1组成

■ 汇编语言 (Assembly Language)

- 用符号代替了0和1，比机器语言便于阅读和记忆

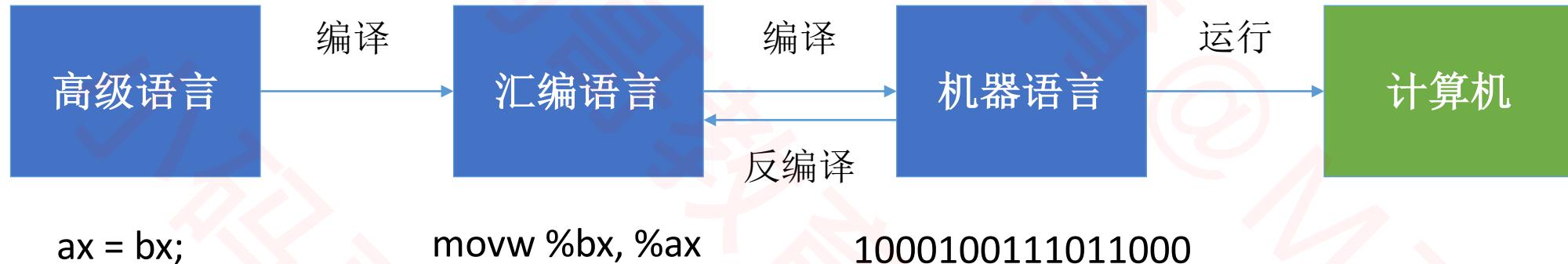
■ 高级语言

- C\C++\Java\JavaScript\Python等，更接近人类自然语言

■ 操作：将寄存器BX的内容送入寄存器AX

- 机器语言：1000100111011000
- 汇编语言：movw %bx, %ax
- 高级语言：ax = bx;

编程语言的发展



- 汇编语言与机器语言一一对应，每一条机器指令都有与之对应的汇编指令
- 汇编语言可以通过编译得到机器语言，机器语言可以通过反汇编得到汇编语言
- 高级语言可以通过编译得到汇编语言\机器语言，但汇编语言\机器语言几乎不可能还原成高级语言



汇编语言的种类

- 汇编语言的种类
 - 8086汇编(16bit)
 - x86汇编(32bit)
 - x64汇编(64bit)
 - ARM汇编(嵌入式、移动设备)
 -
- x86、x64汇编根据编译器的不同，有2种书写格式
 - Intel : Windows派系
 - AT&T : Unix派系
- 作为iOS开发工程师，最主要的汇编语言是
 - AT&T汇编 -> iOS模拟器
 - ARM汇编 -> iOS真机设备

常见汇编指令

项目	AT&T	Intel	说明
寄存器命名	%rax	rax	
操作数顺序	movq %rax, %rdx	mov rdx, rx	将rax的值赋值给rdx
常数\立即数	movq \$3, %rax movq \$0x10, %rax	mov rx, 3 mov rx, 0x10	将3赋值给rax 将0x10赋值给rax
内存赋值	movq \$0xa, 0x1ff7(%rip)	mov qword ptr [rip+0x1ff7], 0xa	将0xa赋值给地址为rip + 0x1ff7的内存空间
取内存地址	leaq -0x18(%rbp), %rax	lea rx, [rbp - 0x18]	将rbp - 0x18这个地址值赋值给rax
jmp指令	jmp *%rdx jmp 0x4001002 jmp *(%rax)	jmp rdx jmp 0x4001002 jmp [rax]	call和jmp写法类似
操作数长度	movl %eax, %edx movb \$0x10, %al leaw 0x10(%dx), %ax	mov edx, eax mov al, 0x10 lea ax, [dx + 0x10]	<p> b = byte (8-bit) s = short (16-bit integer or 32-bit floating point) w = word (16-bit) l = long (32-bit integer or 64-bit floating point) q = quad (64 bit) t = ten bytes (80-bit floating point) </p>

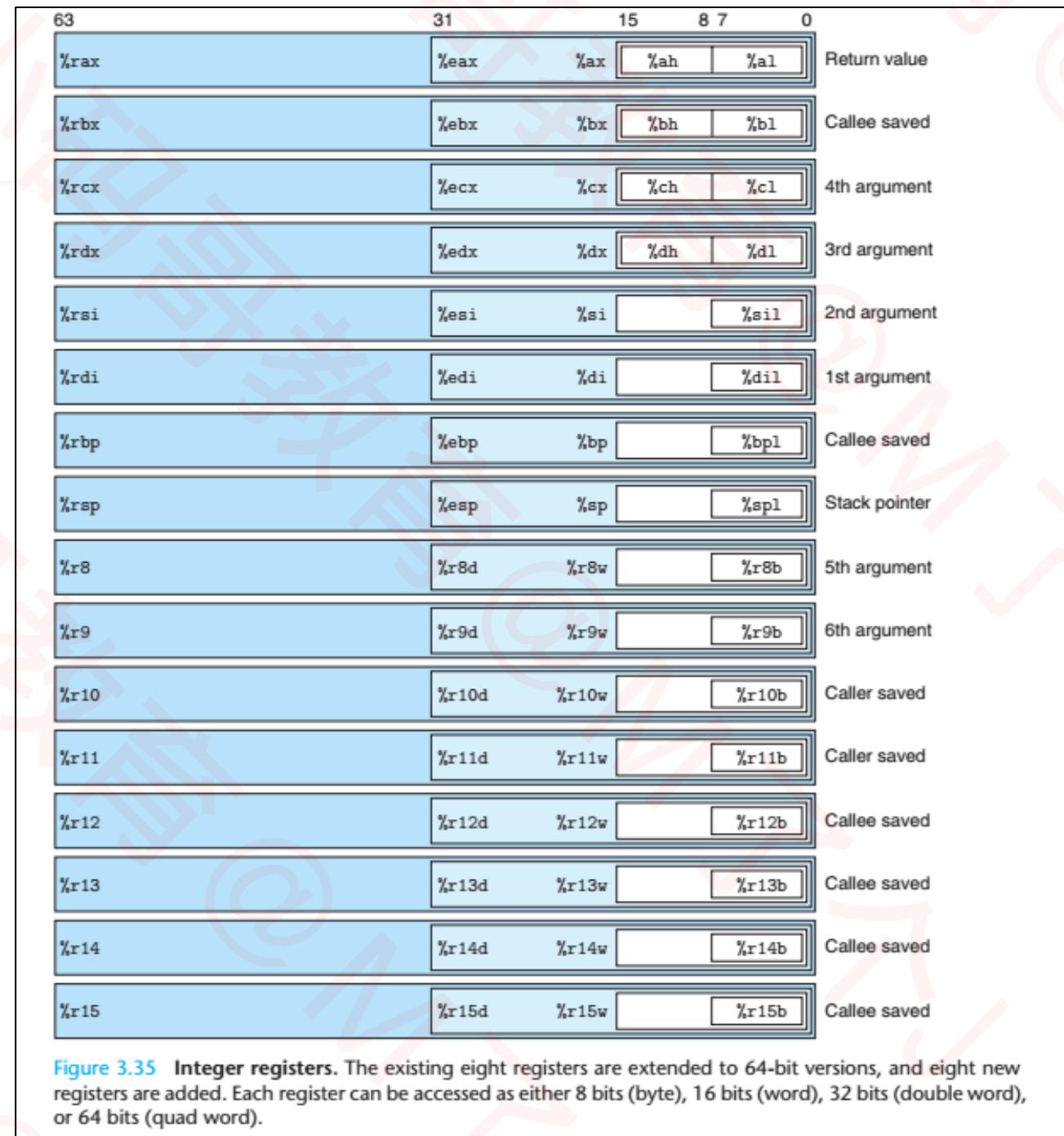


寄存器

- 有16个常用寄存器
 - rax、rbx、rcx、rdx、rsi、rdi、rbp、rsp
 - r8、r9、r10、r11、r12、r13、r14、r15

- 寄存器的具体用途
 - rax、rdx常作为函数返回值使用
 - rdi、rsi、rdx、rcx、r8、r9等寄存器常用于存放函数参数
 - rsp、rbp用于栈操作
 - rip作为指令指针
 - ✓ 存储着CPU下一条要执行的指令的地址
 - ✓ 一旦CPU读取一条指令，rip会自动指向一条指令（存储下一条指令的地址）

寄存器





lldb常用指令

■ 读取寄存器的值

□ register read/格式

□ register read/x

■ 修改寄存器的值

□ register write 寄存器名称 数值

□ register write rax 0

■ 读取内存中的值

□ x/数量-格式-字节大小 内存地址

□ x/3xw 0x00000010

■ 修改内存中的值

□ memory write 内存地址 数值

□ memory write 0x00000010 10

■ 格式

□ x是16进制，f是浮点，d是十进制

■ 字节大小

□ b – byte 1字节

□ h – half word 2字节

□ w – word 4字节

□ g – giant word 8字节

■ expression 表达式

□ 可以简写 : expr 表达式

□ expression \$rax

□ expression \$rax = 1

■ po 表达式

■ print 表达式

□ po/x \$rax

□ po (int)\$rax



lldb常用指令

■ `thread step-over`、`next`、`n`

□ 单步运行，把子函数当做整体一步执行（源码级别）

■ `thread step-in`、`step`、`s`

□ 单步运行，遇到子函数会进入子函数（源码级别）

■ `thread step-inst-over`、`nexti`、`ni`

□ 单步运行，把子函数当做整体一步执行（汇编级别）

■ `thread step-inst`、`stepi`、`si`

□ 单步运行，遇到子函数会进入子函数（汇编级别）

■ `thread step-out`、`finish`

□ 直接执行完当前函数的所有代码，返回到上一个函数（遇到断点会卡住）



规律

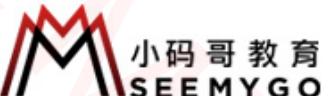
- 内存地址格式为：`0x4bdc(%rip)`，一般是全局变量，全局区（数据段）
- 内存地址格式为：`-0x78(%rbp)`，一般是局部变量，栈空间
- 内存地址格式为：`0x10(%rax)`，一般是堆空间

基础语法

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





Hello World

```
print("Hello World!")
```

- 不用编写main函数，Swift将全局范围内的首句可执行代码作为程序入口
- 一句代码尾部可以省略分号（;），多句代码写到同一行时必须用分号（;）隔开
- 用**var**定义变量，**let**定义常量，编译器能自动推断出变量\常量的类型

1	let	a	=	10	10
2	let	b	=	20	20
3	var	c	=	a + b	30
4	c	+=	30		60

- Playground可以快速预览代码效果，是学习语法的好帮手
- Command + Shift + Enter：运行整个Playground
- Shift + Enter：运行截止到某一行代码



Playground - View

```
1 import UIKit
2 import PlaygroundSupport
3
4 let view = UIView()
5 view.frame = CGRect(x: 0, y: 0, width: 100, height: 100)
6 view.backgroundColor = UIColor.blue
7 PlaygroundPage.current.liveView = view
```



A screenshot of the Xcode playground interface. On the left, the file browser shows a project structure with a 'Sources' folder and a 'Resources' folder containing a file named 'logo.png'. The main editor area contains the following Swift code:

```
1 import UIKit
2 import PlaygroundSupport
3
4 let imageView = UIImageView(image: UIImage(named: "logo"))
5 PlaygroundPage.current.liveView = imageView
```

The bottom right corner of the screen features the orange and white Apple Swift logo.

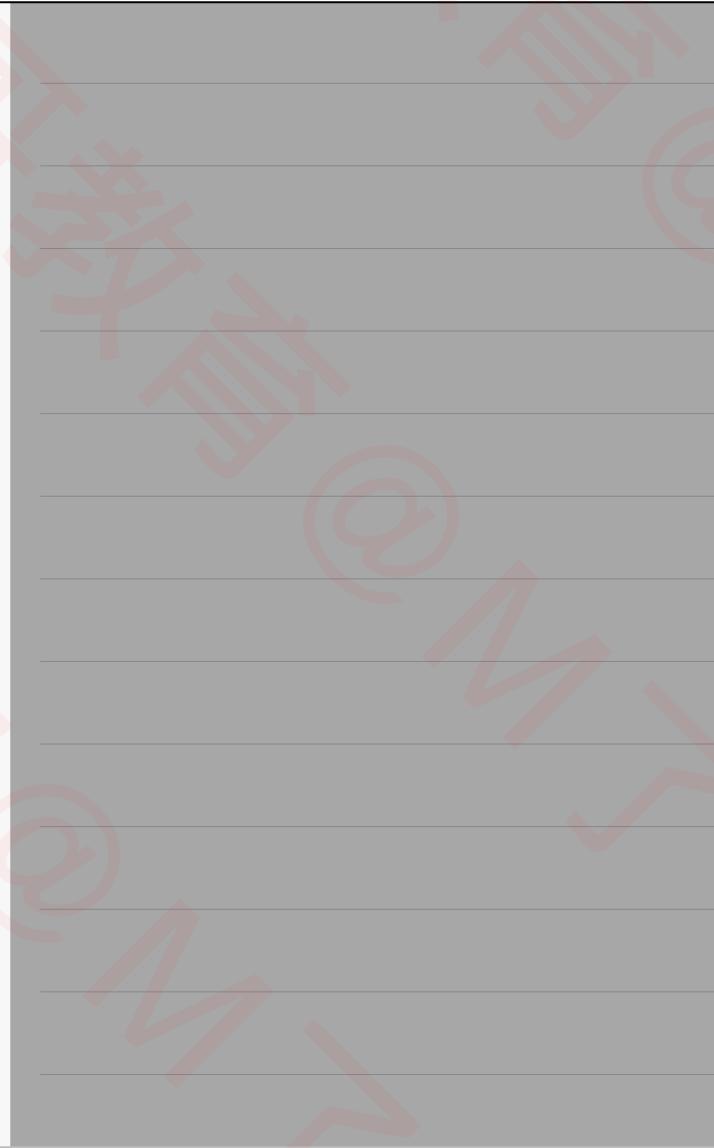


Playground - ViewController

```
1 import UIKit
2 import PlaygroundSupport
3
4 let vc = UITableViewController()
5 vc.view.backgroundColor = UIColor.lightGray
6 PlaygroundPage.current.liveView = vc
```

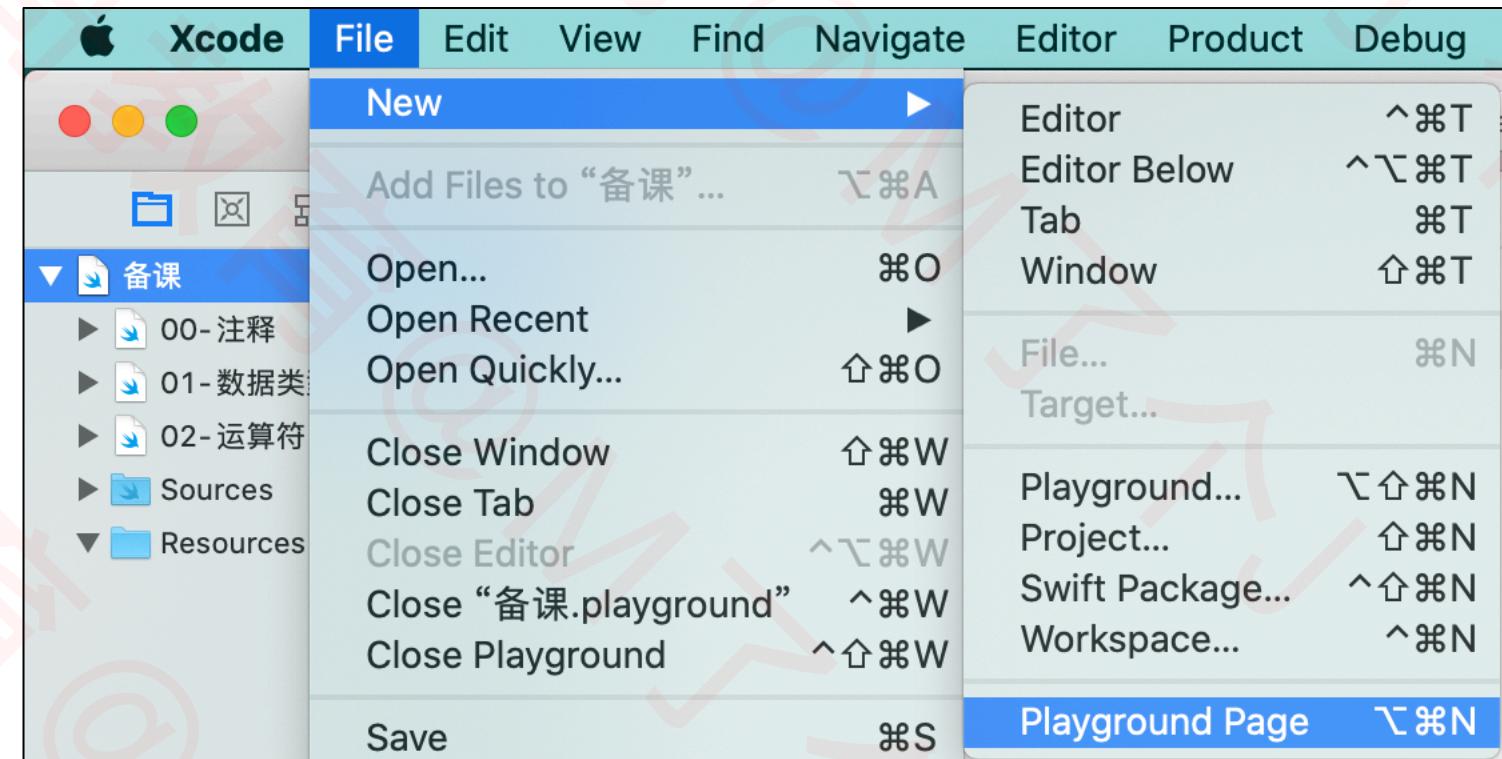
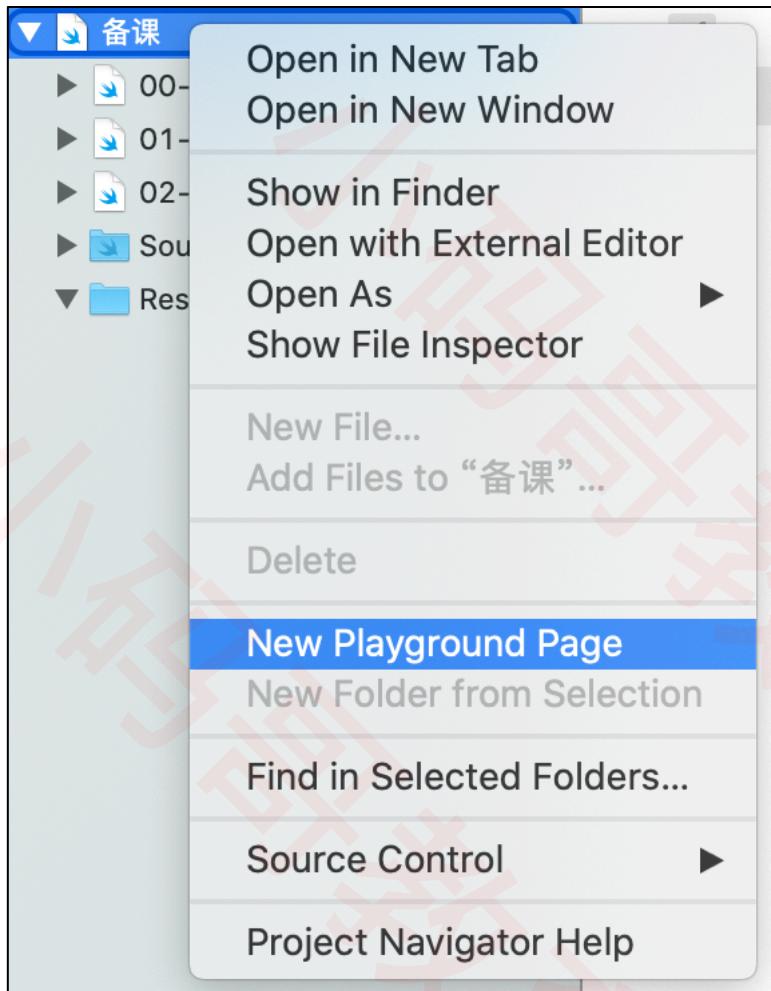
```
7
8
9
```

```
▶
```





Playground – 多Page



注释

```
// 单行注释
```

```
/*  
多行注释  
*/  
  
/*  
1  
/* 多行注释的嵌套 */  
2  
*/
```

- Playground的注释支持markup语法（与markdown相似）

```
//: 开始markup  
  
/*:  
开始markup  
*/
```

- 开启markup渲染效果：Editor -> Show Rendered Markup
- 注意：Markup只在Playground中有效

Markup语法

```
//: [上一页](@previous)  
//: [下一页](@next)
```

[上一页](#) [下一页](#)

```
/*:  
# 一级标题  
  
## 无序列表  
- First Item  
- Secound Item  
  
## 有序列表  
1. First Item  
2. Secound Item  
  
## 笔记  
> This is a note  
---  
  
## 图片  
![Logo](logo.png "Local image")  
  
## 链接  
* [小码哥教育](<a href="https://520it.com">https://520it.com</a>)  
  
## 粗体/斜体  
这是**Bold**, 这是*Italic*  
*/
```

一级标题
无序列表

- First Item
- Secound Item

有序列表

1. First Item
2. Secound Item

笔记

Note
This is a note

图片



链接

- 小码哥教育

粗体/斜体

这是Bold, 这是Italic

常量

- 只能赋值1次
- 它的值不要求在编译时期确定，但使用之前必须赋值1次

```
let age1 = 10

let age2: Int
age2 = 20

func getAge() -> Int {
    return 30
}
let age3 = getAge()
```

- 常量、变量在初始化之前，都不能使用

```
let age: Int
var height: Int
print(age)
print(height)
```



Constant 'age' used before being initialized



Variable 'height' used before being initialized

- 下面代码是错误的

```
let age
age = 20
```

⚠ Found an unexpected second identifier in constant declaration; is there an accidental break?



标识符

- 标识符（比如常量名、变量名、函数名）几乎可以使用任何字符
- 标识符不能以数字开头，不能包含空白字符、制表符、箭头等特殊字符

```
func 🐂🍺() {  
    print("666")  
}  
🍺()  
  
let 👽 = "ET"  
var 🥛 = "milk"
```

常见数据类型

值类型 (value type)	枚举 (enum)	Optional
	结构体 (struct)	Bool、Int、Float、Double、Character String、Array、Dictionary、Set
引用类型 (reference type)	类 (class)	

- 整数类型 : Int8、Int16、Int32、Int64、UInt8、UInt16、UInt32、UInt64
- 在32bit平台，Int等价于Int32；在64bit平台，Int等价于Int64
- 整数的最值 : UInt8.max、Int16.min
- 一般情况下，都是直接使用Int即可
- 浮点类型 : Float，32位，精度只有6位；Double，64位，精度至少15位

```
let letFloat: Float = 30.0
let letDouble = 30.0
```

字面量

```
// 布尔  
let bool = true // 取反是false
```

```
// 字符串  
let string = "小码哥"
```

```
// 字符 (可存储ASCII字符、Unicode字符)  
let character: Character = "🐶"
```

```
// 整数  
let intDecimal = 17 // 十进制  
let intBinary = 0b10001 // 二进制  
let intOctal = 0o21 // 八进制  
let intHexadecimal = 0x11 // 十六进制
```

- 整数和浮点数可以添加额外的零或者添加下划线来增强可读性
- 100_0000、1_000_000.000_000_1、000123.456

```
// 浮点数  
let doubleDecimal = 125.0 // 十进制, 等价于1.25e2, 0.0125等价于1.25e-2  
let doubleHexadecimal1 = 0xFp2 // 十六进制, 意味着15x2^2, 相当于十进制的60.0  
let doubleHexadecimal2 = 0xFp-2 // 十六进制, 意味着15x2^-2, 相当于十进制的3.75  
/* 以下都是表示12.1875  
    十进制: 12.1875、1.21875e1  
    十六进制: 0xC.3p0 */
```

```
// 数组  
let array = [1, 3, 5, 7, 9]
```

```
// 字典  
let dictionary = ["age" : 18, "height" : 168, "weight" : 120]
```



类型转换

```
// 整数转换  
let int1: UInt16 = 2_000  
let int2: UInt8 = 1  
let int3 = int1 + UInt16(int2)
```

```
// 整数、浮点数转换  
let int = 3  
let double = 0.14159  
let pi = Double(int) + double  
let intPi = Int(pi)
```

```
// 字面量可以直接相加，因为数字字面量本身没有明确的类型  
let result = 3 + 0.14159
```



元组 (Tuple)

```
let http404Error = (404, "Not Found")
print("The status code is \(http404Error.0)")
```

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
```

```
let (justTheStatusCode, _) = http404Error
```

```
let http200Status = (statusCode: 200, description: "OK")
print("The status code is \(http200Status.statusCode)")
```

流程控制

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



if-else

```
let age = 4
if age >= 22 {
    print("Get married")
} else if age >= 18 {
    print("Being a adult")
} else if age >= 7 {
    print("Go to school")
} else {
    print("Just a child")
}
```

- if后面的条件只能是Bool类型

```
if age { ! 'Int' is not convertible to 'Bool'
}
```

- if后面的条件可以省略小括号
- 条件后面的大括号不可以省略



while

```
var num = 5
while num > 0 {
    print("num is \(num)")
    num -= 1
} // 打印了5次
```

```
var num = -1
repeat {
    print("num is \(num)")
} while num > 0 // 打印了1次
```

- repeat-while相当于C语言中的do-while
- 这里不用num--，是因为
- 从Swift3开始，去除了自增（++）、自减（--）运算符



for

■ 闭区间运算符 : a...b , a <= 取值 <= b

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for i in 0...3 {
    print(names[i])
} // Anna Alex Brian Jack
```

```
let range = 1...3
for i in range {
    print(names[i])
} // Alex Brian Jack
```

```
let a = 1
var b = 2
for i in a...b {
    print(names[i])
} // Alex Brian
```

```
// i默认是let, 有需要时可以声明为var
for var i in 1...3 {
    i += 5
    print(i)
} // 6 7 8
```

```
for _ in 1...3 {
    print("for")
} // 打印了3次
```

```
for i in a...3 {
    print(names[i])
} // Alex Brian Jack
```

■ 半开区间运算符 : a.. **, a <= 取值 < b**

```
for i in 1..<5 {
    print(i)
} // 1 2 3 4
```

for - 区间运算符用在数组上

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names[0...3] {
    print(name)
} // Anna Alex Brian Jack
```

■ 单侧区间：让区间朝一个方向尽可能的远

```
for name in names[2...] {
    print(name)
} // Brian Jack

for name in names[...2] {
    print(name)
} // Anna Alex Brian

for name in names[...<2] {
    print(name)
} // Anna Alex
```

```
let range = ...5
range.contains(7) // false
range.contains(4) // true
range.contains(-3) // true
```



区间类型

```
let range1: ClosedRange<Int> = 1...3  
let range2: Range<Int> = 1..  
let range3: PartialRangeThrough<Int> = ...5
```

- 字符、字符串也能使用区间运算符，但默认不能用在for-in中

```
let stringRange1 = "cc"..."ff" // ClosedRange<String>  
stringRange1.contains("cb") // false  
stringRange1.contains("dz") // true  
stringRange1.contains("fg") // false  
  
let stringRange2 = "a"..."f"  
stringRange2.contains("d") // true  
stringRange2.contains("h") // false
```

```
// \0到~囊括了所有可能要用到的ASCII字符  
let characterRange: ClosedRange<Character> = "\0"..."~"  
characterRange.contains("G") // true
```

带间隔的区间值

```
let hours = 11
let hourInterval = 2
// tickMark的取值: 从4开始, 累加2, 不超过11
for tickMark in stride(from: 4, through: hours, by: hourInterval) {
    print(tickMark)
} // 4 6 8 10
```



switch

```
var number = 1
switch number {
case 1:
    print("number is 1")
    break
case 2:
    print("number is 2")
    break
default:
    print("number is other")
    break
} // number is 1
```

■ case、default后面不能写大括号{}

```
var number = 1
switch number {
case 1:
    print("number is 1")
case 2:
    print("number is 2")
default:
    print("number is other")
} // number is 1
```

■ 默认可以不写break，并不会贯穿到后面的条件



fallthrough

- 使用 `fallthrough` 可以实现贯穿效果

```
var number = 1
switch number {
case 1:
    print("number is 1")
    fallthrough
case 2:
    print("number is 2")
default:
    print("number is other")
}
// number is 1
// number is 2
```

switch注意点

- switch必须要保证能处理所有情况

```
var number = 1
switch number {    • Switch must be exhaustive
case 1:
    print("number is 1")
case 2:
    print("number is 2")
}
```

- case、default后面至少要有一条语句
- 如果不想做任何事，加个break即可

```
var number = 1
switch number {
case 1:
    print("number is 1")
case 2:
    print("number is 2")
default:
    break
}
```

switch注意点

- 如果能保证已处理所有情况，也可以不必使用default

```
enum Answer { case right, wrong }
let answer = Answer.right
switch answer {
case Answer.right:
    print("right")
case Answer.wrong:
    print("wrong")
}
```

```
// 由于已确定answer是Answer类型，因此可以省略Answer
switch answer {
case .right:
    print("right")
case .wrong:
    print("wrong")
}
```



复合条件

■ switch也支持Character、String类型

```
let string = "Jack"
switch string {
case "Jack":
    fallthrough
case "Rose":
    print("Right person")
default:
    break
} // Right person
```

```
switch string {
case "Jack", "Rose":
    print("Right person")
default:
    break
} // Right person
```

```
let character: Character = "a"
switch character {
case "a", "A":
    print("The letter A")
default:
    print("Not the letter A")
} // The letter A
```



区间匹配、元组匹配

```
let count = 62
switch count {
case 0:
    print("none")
case 1..<5:
    print("a few")
case 5..<12:
    print("several")
case 12..<100:
    print("dozens of")
case 100..<1000:
    print("hundreds of")
default:
    print("many")
} // dozens of
```

```
let point = (1, 1)
switch point {
case (0, 0):
    print("the origin")
case (_, 0):
    print("on the x-axis")
case (0, _):
    print("on the y-axis")
case (-2..<=2, -2..<=2):
    print("inside the box")
default:
    print("outside of the box")
} // inside the box
```

- 可以使用下划线 `_` 忽略某个值
- 关于 `case` 匹配问题，属于模式匹配（Pattern Matching）的范畴，以后会再次详细展开讲解



值绑定

```
let point = (2, 0)
switch point {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
} // on the x-axis with an x value of 2
```

- 必要时 `let` 也可以改为 `var`



where

```
let point = (1, -1)
switch point {
case let (x, y) where x == y:
    print("on the line x == y")
case let (x, y) where x == -y:
    print("on the line x == -y")
case let (x, y):
    print("(x, y) is just some arbitrary point")
} // on the line x == -y
```

```
// 将所有正数加起来
var numbers = [10, 20, -10, -20, 30, -30]
var sum = 0
for num in numbers where num > 0 { // 使用where来过滤num
    sum += num
}
print(sum) // 60
```



标签语句

```
outer: for i in 1...4 {  
    for k in 1...4 {  
        if k == 3 {  
            continue outer  
        }  
        if i == 3 {  
            break outer  
        }  
        print("i == \$(i), k == \$(k)")  
    }  
}
```

函数

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



函数的定义

```
func pi() -> Double {  
    return 3.14  
}  
  
func sum(v1: Int, v2: Int) -> Int {  
    return v1 + v2  
}  
  
sum(v1: 10, v2: 20)
```

■ 形参默认是 `let`，也只能是 `let`

■ 无返回值

```
func sayHello() -> Void {  
    print("Hello")  
}
```

```
func sayHello() -> () {  
    print("Hello")  
}
```

```
func sayHello() {  
    print("Hello")  
}
```



隐式返回 (Implicit Return)

- 如果整个函数体是一个单一表达式，那么函数会隐式返回这个表达式

```
func sum(v1: Int, v2: Int) -> Int {  
    v1 + v2  
}  
sum(v1: 10, v2: 20) // 30
```



返回元组：实现多返回值

```
func calculate(v1: Int, v2: Int) -> (sum: Int, difference: Int, average: Int) {  
    let sum = v1 + v2  
    return (sum, v1 - v2, sum >> 1)  
}  
let result = calculate(v1: 20, v2: 10)  
result.sum // 30  
result.difference // 10  
result.average // 15
```



函数的文档注释

```
/// 求和【概述】  
///  
/// 将2个整数相加【更详细的描述】  
///  
/// - Parameter v1: 第1个整数  
/// - Parameter v2: 第2个整数  
/// - Returns: 2个整数的和  
///  
/// - Note: 传入2个整数即可【批注】  
///  
func sum(v1: Int, v2: Int) -> Int {  
    v1 + v2  
}
```

■ 参考：<https://swift.org/documentation/api-design-guidelines/>

Summary

求和【概述】

Declaration

```
func sum(v1: Int, v2: Int) -> Int
```

Discussion

将2个整数相加【更详细的描述】

Note

传入2个整数即可【批注】

Parameters

v1 第1个整数

v2 第2个整数

Returns

2个整数的和

Declared In

03-函数.xcplaygroundpage

```
95 func sum(v1: Int, v2: Int) -> Int {  
96     v1 + v2  
97 }
```



参数标签 (Argument Label)

■ 可以修改参数标签

```
func goToWork(at time: String) {  
    print("this time is \(time)")  
}  
goToWork(at: "08:00")  
// this time is 08:00
```

■ 可以使用下划线 _ 省略参数标签

```
func sum(_ v1: Int, _ v2: Int) -> Int {  
    v1 + v2  
}  
sum(10, 20)
```

默认参数值 (Default Parameter Value)

- 参数可以有默认值

```
func check(name: String = "nobody", age: Int, job: String = "none") {  
    print("name=\(name), age=\(age), job=\(job)")  
}  
  
check(name: "Jack", age: 20, job: "Doctor") // name=Jack, age=20, job=Doctor  
check(name: "Rose", age: 18) // name=Rose, age=18, job=none  
check(age: 10, job: "Batman") // name=nobody, age=10, job=Batman  
check(age: 15) // name=nobody, age=15, job=none
```

- C++的默认参数值有个限制：必须从右往左设置。由于Swift拥有参数标签，因此并没有此类限制
- 但是在省略参数标签时，需要特别注意，避免出错

```
// 这里的middle不可以省略参数标签  
func test(_ first: Int = 10, middle: Int, _ last: Int = 30) { }  
test(middle: 20)
```



可变参数 (Variadic Parameter)

```
func sum(_ numbers: Int...) -> Int {  
    var total = 0  
    for number in numbers {  
        total += number  
    }  
    return total  
}  
sum(10, 20, 30, 40) // 100
```

- 一个函数最多只能有1个可变参数
- 紧跟在可变参数后面的参数不能省略参数标签

```
// 参数string不能省略标签  
func test(_ numbers: Int..., string: String, _ other: String) { }  
test(10, 20, 30, string: "Jack", "Rose")
```



Swift自带的print函数

```
/// - Parameters:  
///   - items: Zero or more items to print.  
///   - separator: A string to print between each item. The default is a single space (` " `).  
///   - terminator: The string to print after all items have been printed. The  
///     default is a newline (` "\n" `).  
public func print(_ items: Any..., separator: String = " ", terminator: String = "\n")
```

```
print(1, 2, 3, 4, 5) // 1 2 3 4 5
```

```
print(1, 2, 3, 4, 5, separator: "_") // 1_2_3_4_5
```

```
print("My name is Jake.", terminator: "")  
print("My age is 18.")  
// My name is Jake.My age is 18.
```



输入输出参数 (In-Out Parameter)

- 可以用 **inout** 定义一个输入输出参数：可以在函数内部修改外部实参的值

```
func swapValues(_ v1: inout Int, _ v2: inout Int) {  
    let tmp = v1  
    v1 = v2  
    v2 = tmp  
}  
  
var num1 = 10  
var num2 = 20  
swapValues(&num1, &num2)
```

- 可变参数不能标记为 **inout**
- **inout** 参数不能有默认值
- **inout** 参数只能传入可以被多次赋值的
- **inout** 参数的本质是地址传递（引用传递）

```
func swapValues(_ v1: inout Int, _ v2: inout Int) {  
    (v1, v2) = (v2, v1)  
}
```



函数重载 (Function Overload)

- 规则
- 函数名相同
- 参数个数不同 || 参数类型不同 || 参数标签不同

```
sum(v1: 10, v2: 20) // 30
sum(v1: 10, v2: 20, v3: 30) // 60
sum(v1: 10, v2: 20.0) // 30.0
sum(v1: 10.0, v2: 20) // 30.0
sum(10, 20) // 30
sum(a: 10, b: 20) // 30
```

```
func sum(v1: Int, v2: Int) -> Int {
    v1 + v2
}
```

```
func sum(v1: Int, v2: Int, v3: Int) -> Int {
    v1 + v2 + v3
} // 参数个数不同
```

```
func sum(v1: Int, v2: Double) -> Double {
    Double(v1) + v2
} // 参数类型不同
func sum(v1: Double, v2: Int) -> Double {
    v1 + Double(v2)
} // 参数类型不同
```

```
func sum(_ v1: Int, _ v2: Int) -> Int {
    v1 + v2
} // 参数标签不同
func sum(a: Int, b: Int) -> Int {
    a + b
} // 参数标签不同
```



函数重载注意点

- 返回值类型与函数重载无关

```
func sum(v1: Int, v2: Int) -> Int { v1 + v2 }
func sum(v1: Int, v2: Int) { }
sum(v1: 10, v2: 20)      ! Ambiguous use of 'sum(v1:v2:)'
```

- 默认参数值和函数重载一起使用产生二义性时，编译器并不会报错（在C++中会报错）

```
func sum(v1: Int, v2: Int) -> Int {
    v1 + v2
}
func sum(v1: Int, v2: Int, v3: Int = 10) -> Int {
    v1 + v2 + v3
}
// 会调用sum(v1: Int, v2: Int)
sum(v1: 10, v2: 20)
```

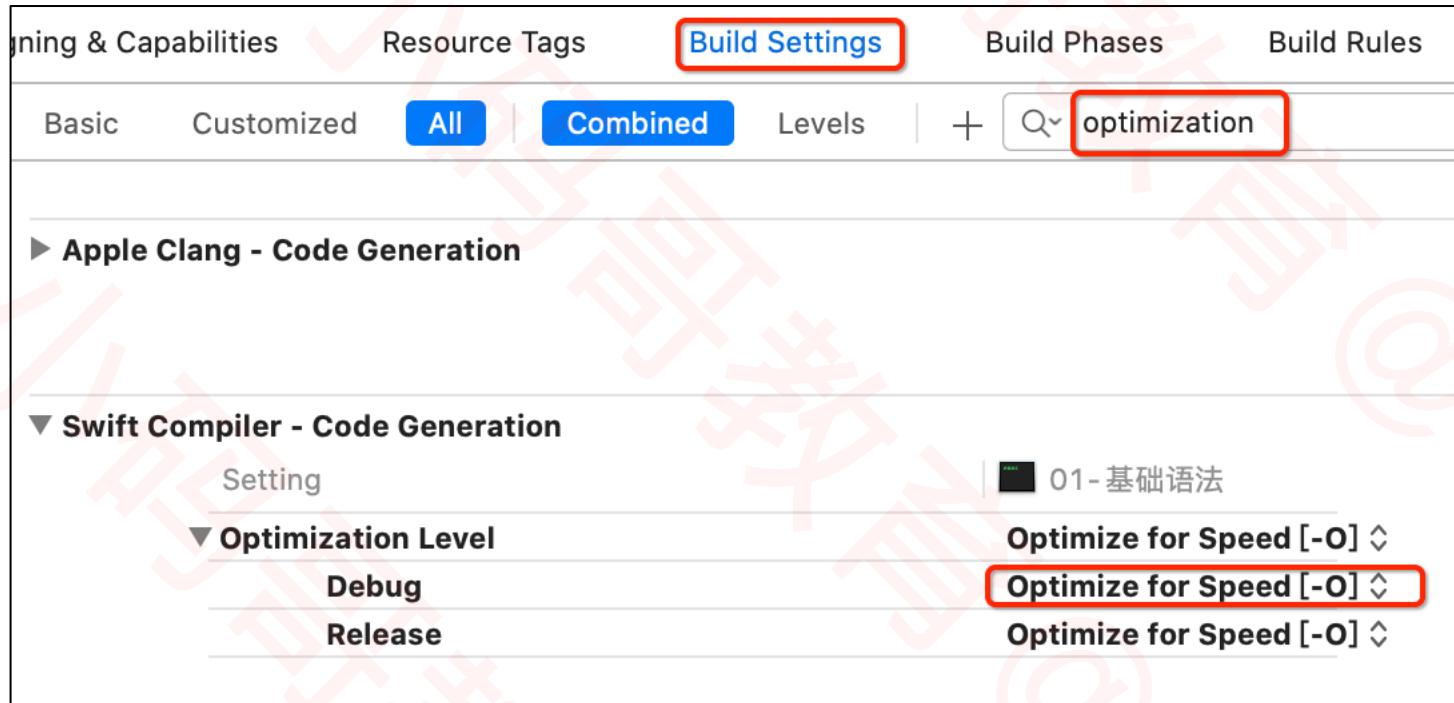
函数重载注意点

- 可变参数、省略参数标签、函数重载一起使用产生二义性时，编译器有可能会报错

```
func sum(v1: Int, v2: Int) -> Int {  
    v1 + v2  
}  
func sum(_ v1: Int, _ v2: Int) -> Int {  
    v1 + v2  
}  
func sum(_ numbers: Int...) -> Int {  
    var total = 0  
    for number in numbers {  
        total += number  
    }  
    return total  
}  
// error: ambiguous use of 'sum'  
sum(10, 20)
```

内联函数 (Inline Function)

- 如果开启了编译器优化 (Release模式默认会开启优化)，编译器会自动将某些函数变成内联函数
- 将函数调用展开成函数体



- 哪些函数不会被自动内联？
- 函数体比较长
- 包含递归调用
- 包含动态派发
-

```
// 永远不会被内联（即使开启了编译器优化）  
@inline(never) func test() {  
    print("test")  
}
```

```
// 开启编译器优化后，即使代码很长，也会被内联（递归调用函数、动态派发的函数除外）  
@inline(__always) func test() {  
    print("test")  
}
```

- 在Release模式下，编译器已经开启优化，会自动决定哪些函数需要内联，因此没必要使用**@inline**



函数类型 (Function Type)

- 每一个函数都是有类型的，函数类型由形式参数类型、返回值类型组成

```
func test() { } // () -> Void 或者 () -> ()
```

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

```
// 定义变量  
var fn: (Int, Int) -> Int = sum  
fn(2, 3) // 5, 调用时不需要参数标签
```

函数类型作为函数参数

```
func sum(v1: Int, v2: Int) -> Int {  
    v1 + v2  
}  
func difference(v1: Int, v2: Int) -> Int {  
    v1 - v2  
}  
func printResult(_ mathFn: (Int, Int) -> Int, _ a: Int, _ b: Int) {  
    print("Result: \(mathFn(a, b))")  
}  
printResult(sum, 5, 2) // Result: 7  
printResult(difference, 5, 2) // Result: 3
```



函数类型作为函数返回值

```
func next(_ input: Int) -> Int {  
    input + 1  
}  
  
func previous(_ input: Int) -> Int {  
    input - 1  
}  
  
func forward(_ forward: Bool) -> (Int) -> Int {  
    forward ? next : previous  
}  
  
forward(true)(3) // 4  
forward(false)(3) // 2
```

- 返回值是函数类型的函数，叫做高阶函数 (Higher-Order Function)



typealias

■ typealias用来给类型起别名

```
typealias Byte = Int8
typealias Short = Int16
typealias Long = Int64
```

```
typealias Date = (year: Int, month: Int, day: Int)
func test(_ date: Date) {
    print(date.0)
    print(date.year)
}
test((2011, 9, 10))
```

```
typealias IntFn = (Int, Int) -> Int

func difference(v1: Int, v2: Int) -> Int {
    v1 - v2
}

let fn: IntFn = difference
fn(20, 10) // 10

func setFn(_ fn: IntFn) { }
setFn(difference)

func getFn() -> IntFn { difference }
```

■ 按照Swift标准库的定义，Void就是空元组()

```
public typealias Void = ()
```



嵌套函数 (Nested Function)

- 将函数定义在函数内部

```
func forward(_ forward: Bool) -> (Int) -> Int {  
    func next(_ input: Int) -> Int {  
        input + 1  
    }  
    func previous(_ input: Int) -> Int {  
        input - 1  
    }  
    return forward ? next : previous  
}  
forward(true)(3) // 4  
forward(false)(3) // 2
```

枚举

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





枚举的基本用法

```
enum Direction {  
    case north  
    case south  
    case east  
    case west  
}
```

```
var dir = Direction.west  
dir = Direction.east  
dir = .north  
print(dir) // north
```

```
enum Direction {  
    case north, south, east, west  
}
```

```
switch dir {  
case .north:  
    print("north")  
case .south:  
    print("south")  
case .east:  
    print("east")  
case .west:  
    print("west")  
}
```



关联值 (Associated Values)

- 有时将枚举的成员值跟其他类型的值关联存储在一起，会非常有用

```
enum Score {  
    case points(Int)  
    case grade(Character)  
}
```

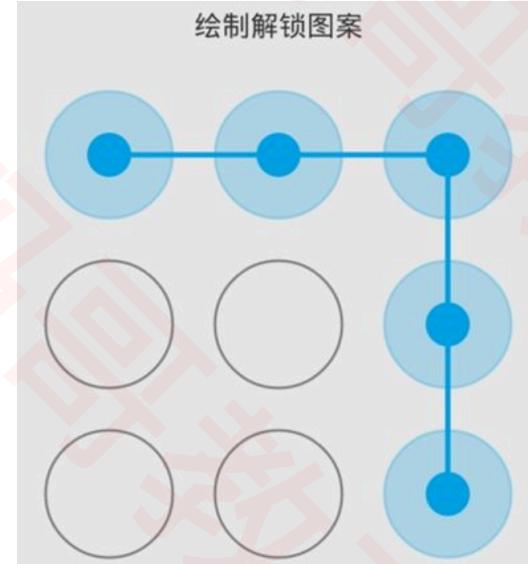
```
var score = Score.points(96)  
score = .grade("A")
```

```
switch score {  
case let .points(i):  
    print(i, "points")  
case let .grade(i):  
    print("grade", i)  
} // grade A
```

```
enum Date {  
    case digit(year: Int, month: Int, day: Int)  
    case string(String)  
}  
  
var date = Date.digit(year: 2011, month: 9, day: 10)  
date = .string("2011-09-10")  
  
switch date {  
case .digit(let year, let month, let day):  
    print(year, month, day)  
case .string(value):  
    print(value)  
}
```

- 必要时 `let` 也可以改为 `var`

关联值举例



```
enum Password {  
    case number(Int, Int, Int, Int)  
    case gesture(String)  
}
```

```
var pwd = Password.number(3, 5, 7, 8)  
pwd = .gesture("12369")
```

```
switch pwd {  
case let .number(n1, n2, n3, n4):  
    print("number is ", n1, n2, n3, n4)  
case let .gesture(str):  
    print("gesture is", str)  
}
```



原始值 (Raw Values)

- 枚举成员可以使用相同类型的默认值预先对应，这个默认值叫做：原始值

```
enum PokerSuit : Character {  
    case spade = "\u2660" // ♠  
    case heart = "\u2665" // ♥  
    case diamond = "\u2666" // ♦  
    case club = "\u2663" // ♣  
}
```

```
var suit = PokerSuit.spade  
print(suit) // spade  
print(suit.rawValue) // ♠  
print(PokerSuit.club.rawValue) // ♣
```

```
enum Grade : String {  
    case perfect = "A"  
    case great = "B"  
    case good = "C"  
    case bad = "D"  
}  
  
print(Grade.perfect.rawValue) // A  
print(Grade.great.rawValue) // B  
print(Grade.good.rawValue) // C  
print(Grade.bad.rawValue) // D
```

- 注意：原始值不占用枚举变量的内存



隐式原始值 (Implicitly Assigned Raw Values)

- 如果枚举的原始值类型是 Int、String , Swift会自动分配原始值

```
enum Direction : String {  
    case north = "north"  
    case south = "south"  
    case east = "east"  
    case west = "west"  
}
```

```
// 等价于  
enum Direction : String {  
    case north, south, east, west  
}  
  
print(Direction.north) // north  
print(Direction.north.rawValue) // north
```

```
enum Season : Int {  
    case spring, summer, autumn, winter  
}  
  
print(Season.spring.rawValue) // 0  
print(Season.summer.rawValue) // 1  
print(Season.autumn.rawValue) // 2  
print(Season.winter.rawValue) // 3
```

```
enum Season : Int {  
    case spring = 1, summer, autumn = 4, winter  
}  
  
print(Season.spring.rawValue) // 1  
print(Season.summer.rawValue) // 2  
print(Season.autumn.rawValue) // 4  
print(Season.winter.rawValue) // 5
```



递归枚举 (Recursive Enumeration)

```
indirect enum ArithExpr {  
    case number(Int)  
    case sum(ArithExpr, ArithExpr)  
    case difference(ArithExpr, ArithExpr)  
}
```

```
enum ArithExpr {  
    case number(Int)  
    indirect case sum(ArithExpr, ArithExpr)  
    indirect case difference(ArithExpr, ArithExpr)  
}
```

```
let five = ArithExpr.number(5)  
let four = ArithExpr.number(4)  
let two = ArithExpr.number(2)  
let sum = ArithExpr.sum(five, four)  
let difference = ArithExpr.difference(sum, two)
```

```
func calculate(_ expr: ArithExpr) -> Int {  
    switch expr {  
        case let .number(value):  
            return value  
        case let .sum(left, right):  
            return calculate(left) + calculate(right)  
        case let .difference(left, right):  
            return calculate(left) - calculate(right)  
    }  
}  
  
calculate(difference)
```

MemoryLayout

- 可以使用MemoryLayout获取数据类型占用的内存大小

```
enum Password {  
    case number(Int, Int, Int, Int)  
    case other  
}
```

MemoryLayout<Password>.stride // 40, 分配占用的空间大小

MemoryLayout<Password>.size // 33, 实际用到的空间大小

MemoryLayout<Password>.alignment // 8, 对齐参数

```
var pwd = Password.number(9, 8, 6, 4)  
pwd = .other  
MemoryLayout.stride(ofValue: pwd) // 40  
MemoryLayout.size(ofValue: pwd) // 33  
MemoryLayout.alignment(ofValue: pwd) // 8
```



思考下面枚举变量的内存布局

```
enum TestEnum {  
    case test1, test2, test3  
}  
  
var t = TestEnum.test1  
t = .test2  
t = .test3
```

```
enum TestEnum : Int {  
    case test1 = 1, test2 = 2, test3 = 3  
}  
  
var t = TestEnum.test1  
t = .test2  
t = .test3
```

```
enum TestEnum {  
    case test  
}  
  
var t = TestEnum.test
```

```
enum TestEnum {  
    case test(Int)  
}  
  
var t = TestEnum.test(10)
```

```
enum TestEnum {  
    case test1(Int, Int, Int)  
    case test2(Int, Int)  
    case test3(Int)  
    case test4(Bool)  
    case test5  
}  
  
var e = TestEnum.test1(1, 2, 3)  
e = .test2(4, 5)  
e = .test3(6)  
e = .test4(true)  
e = .test5
```

■ 它们的switch语句底层又是如何实现的？

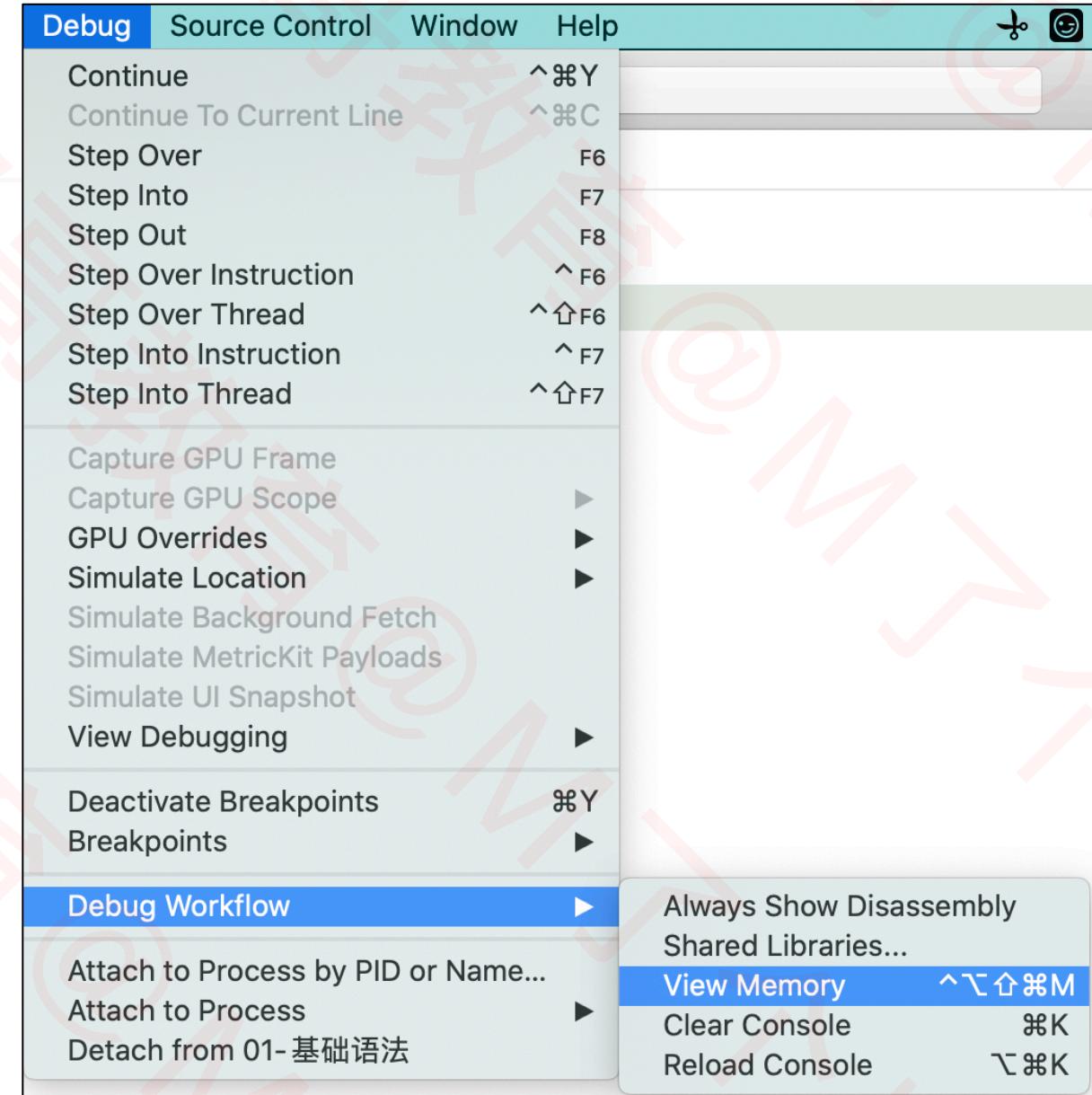


窥探内存

```
2 var num = 10
3 var n = num + 1
4
```

V n = (Int) 0
V num = (Int) 10

- Print Description of "num"
- Copy
- View Value As
- ▶
- Edit Value...
- Edit Summary Format...
- ▶
- Add Expression...
- Delete Expression
- ▶
- Watch "num"
- View Memory of "num"**



■ 窥探内存细节的小工具：<https://github.com/CoderMJLee/Mems>

进一步观察下面枚举的内存布局

```
enum TestEnum {  
    case test0  
    case test1  
    case test2  
    case test4(Int)  
    case test5(Int, Int)  
    case test6(Int, Int, Int, Bool)  
}
```

```
enum TestEnum {  
    case test0  
    case test1  
    case test2  
    case test4(Int)  
    case test5(Int, Int)  
    case test6(Int, Int, Bool, Int)  
}
```

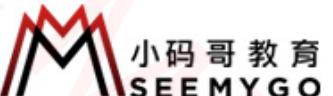
```
enum TestEnum {  
    case test0  
    case test1  
    case test2  
    case test4(Int)  
    case test5(Int, Int)  
    case test6(Int, Bool, Int)  
}
```

可选项

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





可选项（Optional）

- 可选项，一般也叫可选类型，它允许将值设置为 nil
- 在类型名称后面加个问号 ? 来定义一个可选项

```
var name: String? = "Jack"  
name = nil
```

```
var age: Int? // 默认就是nil  
age = 10  
age = nil
```

```
var array = [1, 15, 40, 29]  
func get(_ index: Int) -> Int? {  
    if index < 0 || index >= array.count {  
        return nil  
    }  
    return array[index]  
}
```

```
print(get(1)) // Optional(15)  
print(get(-1)) // nil  
print(get(4)) // nil
```

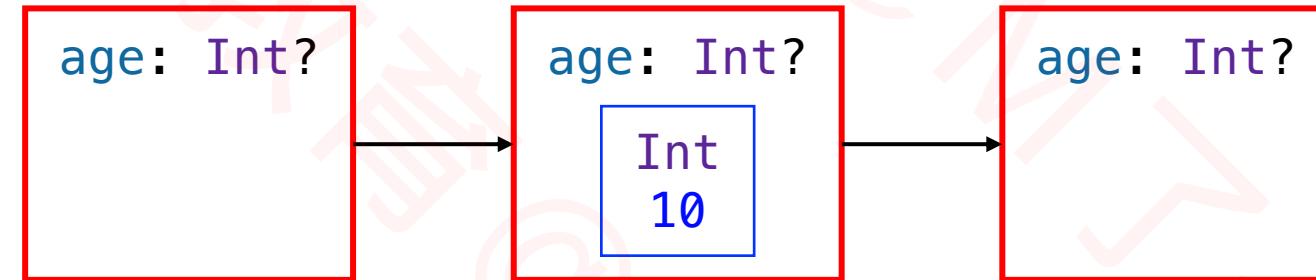
强制解包 (Forced Unwrapping)

■ 可选项是对其它类型的一层包装，可以将它理解为一个盒子

□ 如果为 `nil`，那么它是个空盒子

□ 如果不为 `nil`，那么盒子里装的是：被包装类型的数据

```
var age: Int? // 默认就是nil  
age = 10  
age = nil
```



■ 如果要从可选项中取出被包装的数据（将盒子里装的东西取出来），需要使用 **感叹号！** 进行强制解包

```
var age: Int? = 10  
let ageInt: Int = age!  
ageInt += 10
```

■ 如果对值为 `nil` 的可选项（空盒子）进行强制解包，将会产生运行时错误

```
var age: Int?  
age!
```

Fatal error: Unexpectedly found nil while unwrapping an Optional value



判断可选项是否包含值

```
let number = Int("123")
if number != nil {
    print("字符串转换整数成功: \(number!)")
} else {
    print("字符串转换整数失败")
}
// 字符串转换整数成功: 123
```



可选项绑定 (Optional Binding)

■ 可以使用可选项绑定来判断可选项是否包含值

□ 如果包含就自动解包，把值赋给一个临时的常量(let)或者变量(var)，并返回true，否则返回false

```
if let number = Int("123") {  
    print("字符串转换整数成功: \(number)")  
    // number是强制解包之后的Int值  
    // number作用域仅限于这个大括号  
} else {  
    print("字符串转换整数失败")  
}  
// 字符串转换整数成功: 123
```

```
enum Season : Int {  
    case spring = 1, summer, autumn, winter  
}  
if let season = Season(rawValue: 6) {  
    switch season {  
        case .spring:  
            print("the season is spring")  
        default:  
            print("the season is other")  
    }  
} else {  
    print("no such season")  
}  
// no such season
```



等价写法

```
if let first = Int("4") {  
    if let second = Int("42") {  
        if first < second && second < 100 {  
            print("\(first) < \(second) < 100")  
        }  
    }  
}  
// 4 < 42 < 100
```

```
if let first = Int("4"),  
    let second = Int("42"),  
    first < second && second < 100 {  
    print("\(second) < \(second) < 100")  
}  
// 4 < 42 < 100
```



while循环中使用可选项绑定

```
// 遍历数组，将遇到的正数都加起来，如果遇到负数或者非数字，停止遍历
var strs = ["10", "20", "abc", "-20", "30"]
```

```
var index = 0
var sum = 0
while let num = Int(strs[index]), num > 0 {
    sum += num
    index += 1
}
print(sum)
```



空合并运算符 ?? (Nil-Coalescing Operator)

```
public func ?? <T>(optional: T?, defaultValue: @autoclosure () throws -> T?) rethrows -> T?
```

```
public func ?? <T>(optional: T?, defaultValue: @autoclosure () throws -> T) rethrows -> T
```

■ a ?? b

□ a 是可选项

□ b 是可选项 或者 不是可选项

□ b 跟 a 的存储类型必须相同

□ 如果 a 不为nil , 就返回 a

□ 如果 a 为nil , 就返回 b

□ 如果 b 不是可选项 , 返回 a 时会自动解包



空合并运算符 ?? (Nil-Coalescing Operator)

```
let a: Int? = 1  
let b: Int? = 2  
let c = a ?? b // c是Int? , Optional(1)
```

```
let a: Int? = nil  
let b: Int? = 2  
let c = a ?? b // c是Int? , Optional(2)
```

```
let a: Int? = nil  
let b: Int? = nil  
let c = a ?? b // c是Int? , nil
```

```
let a: Int? = 1  
let b: Int = 2  
let c = a ?? b // c是Int , 1
```

```
let a: Int? = nil  
let b: Int = 2  
let c = a ?? b // c是Int , 2
```

```
let a: Int? = nil  
let b: Int = 2  
// 如果不使用??运算符  
let c: Int  
if let tmp = a {  
    c = tmp  
} else {  
    c = b  
}
```



多个 ?? 一起使用

```
let a: Int? = 1
let b: Int? = 2
let c = a ?? b ?? 3 // c是Int , 1
```

```
let a: Int? = nil
let b: Int? = 2
let c = a ?? b ?? 3 // c是Int , 2
```

```
let a: Int? = nil
let b: Int? = nil
let c = a ?? b ?? 3 // c是Int , 3
```



??跟if let配合使用

```
let a: Int? = nil
let b: Int? = 2
if let c = a ?? b {
    print(c)
}
// 类似于if a != nil || b != nil
```

```
if let c = a, let d = b {
    print(c)
    print(d)
}
// 类似于if a != nil && b != nil
```



if语句实现登陆

```
func login(_ info: [String : String]) {  
    let username: String  
    if let tmp = info["username"] {  
        username = tmp  
    } else {  
        print("请输入用户名")  
        return  
    }  
    let password: String  
    if let tmp = info["password"] {  
        password = tmp  
    } else {  
        print("请输入密码")  
        return  
    }  
    // if username ....  
    // if password ....  
    print("用户名: \(username)", "密码: \(password)", "登陆ing")  
}
```

```
login(["username" : "jack", "password" : "123456"]) // 用户名: jack 密码: 123456 登陆ing  
login(["password" : "123456"]) // 请输入密码  
login(["username" : "jack"]) // 请输入用户名
```



guard语句

```
guard 条件 else {  
    // do something....  
    // 退出当前作用域  
    // return、break、continue、throw error  
}
```

- 当guard语句的条件为**false**时，就会执行大括号里面的代码
- 当guard语句的条件为**true**时，就会跳过guard语句
- guard语句特别适合用来“提前退出”

■ 当使用guard语句进行可选项绑定时，绑定的常量(**let**)、变量(**var**)也能在外层作用域中使用

```
func login(_ info: [String : String]) {  
    guard let username = info["username"] else {  
        print("请输入用户名")  
        return  
    }  
    guard let password = info["password"] else {  
        print("请输入密码")  
        return  
    }  
    // if username ....  
    // if password ....  
    print("用户名: \(username)", "密码: \(password)", "登陆ing")  
}
```



隐式解包 (Implicitly Unwrapped Optional)

- 在某些情况下，可选项一旦被设定值之后，就会一直拥有值
- 在这种情况下，可以去掉检查，也不必每次访问的时候都进行解包，因为它能确定每次访问的时候都有值
- 可以在类型后面加个感叹号！定义一个隐式解包的可选项

```
let num1: Int! = 10
let num2: Int = num1
if num1 != nil {
    print(num1 + 6) // 16
}
if let num3 = num1 {
    print(num3)
}
```

```
let num1: Int! = nil
// Fatal error: Unexpectedly found nil while implicitly unwrapping an Optional value
let num2: Int = num1
```



字符串插值

- 可选项在字符串插值或者直接打印时，编译器会发出警告

```
var age: Int? = 10  
print("My age is \(age)")
```

- 至少有3种方法消除警告

```
print("My age is \(age!)")  
// My age is 10
```

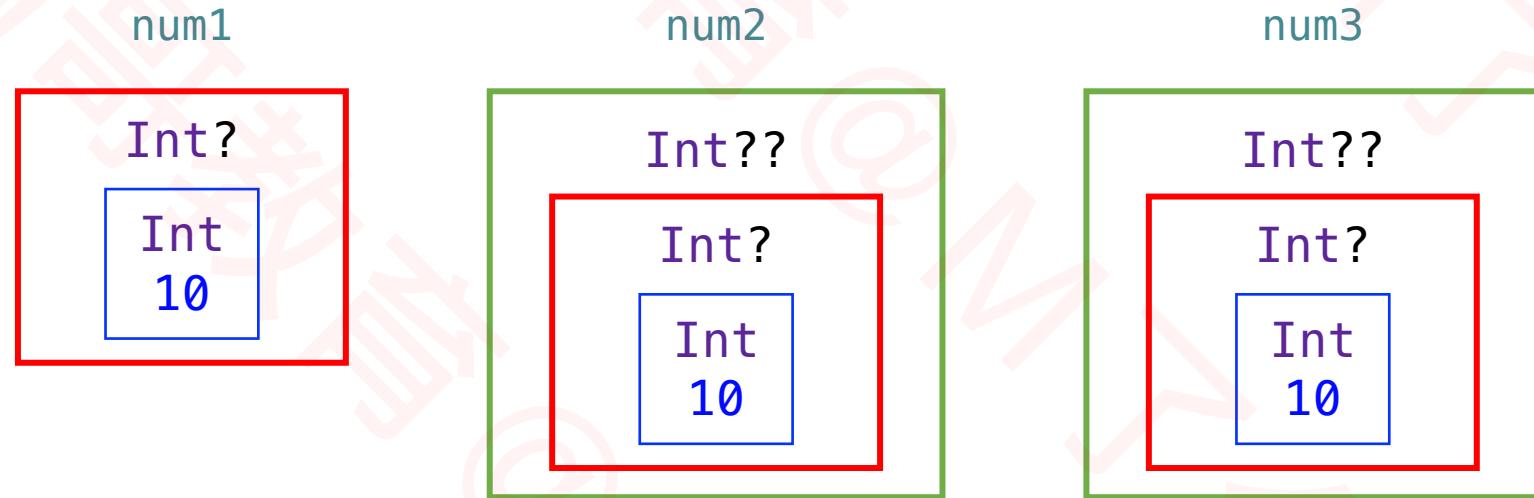
```
print("My age is \(String(describing: age))")  
// My age is Optional(10)
```

```
print("My age is \(age ?? 0)")  
// My age is 10
```

多重可选项

```
var num1: Int? = 10  
var num2: Int?? = num1  
var num3: Int?? = 10
```

```
print(num2 == num3) // true
```



- 可以使用lldb指令 **frame variable -R** 或者 **fr v -R** 查看区别



多重可选项

```
var num1: Int? = nil  
var num2: Int?? = num1  
var num3: Int?? = nil
```

```
print(num2 == num3) // false
```

```
(num2 ?? 1) ?? 2 // 2  
(num3 ?? 1) ?? 2 // 1
```

num1

Int?

num2

Int??

Int?

num3

Int??

结构体和类

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>

码拉松



实力IT教育 www.520it.com

结构体

- 在 Swift 标准库中，绝大多数的公开类型都是结构体，而枚举和类只占很小一部分
- 比如 Bool、Int、Double、String、Array、Dictionary 等常见类型都是结构体

```
① struct Date {  
②     var year: Int  
③     var month: Int  
④     var day: Int  
⑤ }  
⑥ var date = Date(year: 2019, month: 6, day: 23)
```

- 所有的结构体都有一个编译器自动生成的初始化器（ initializer，初始化方法、构造器、构造方法）
- 在第⑥行调用的，可以传入所有成员值，用以初始化所有成员（存储属性，Stored Property）

结构体的初始化器

■ 编译器会根据情况，可能会为结构体生成多个初始化器，宗旨是：保证所有成员都有初始值

```
struct Point {  
    var x: Int  
    var y: Int  
}  
  
var p1 = Point(x: 10, y: 10)  
var p2 = Point(y: 10) 2 ● Missing argument for parameter 'x' in call  
var p3 = Point(x: 10) 2 ● Missing argument for parameter 'y' in call  
var p4 = Point()      2 ● Missing argument for parameter 'x' in call
```

```
struct Point {  
    var x: Int = 0  
    var y: Int  
}  
  
var p1 = Point(x: 10, y: 10)  
var p2 = Point(y: 10)  
var p3 = Point(x: 10)  
var p4 = Point()
```

```
struct Point {  
    var x: Int  
    var y: Int = 0  
}  
  
var p1 = Point(x: 10, y: 10)  
var p2 = Point(y: 10)  
var p3 = Point(x: 10)  
var p4 = Point()
```

```
struct Point {  
    var x: Int = 0  
    var y: Int = 0  
}  
  
var p1 = Point(x: 10, y: 10)  
var p2 = Point(y: 10)  
var p3 = Point(x: 10)  
var p4 = Point()
```



思考：下面代码能编译通过么？

```
struct Point {  
    var x: Int?  
    var y: Int?  
}  
var p1 = Point(x: 10, y: 10)  
var p2 = Point(y: 10)  
var p3 = Point(x: 10)  
var p4 = Point()
```

- 可选项都有个默认值nil
- 因此可以编译通过



自定义初始化器

- 一旦在定义结构体时自定义了初始化器，编译器就不会再帮它自动生成其他初始化器

```
struct Point {  
    var x: Int = 0  
    var y: Int = 0  
    init(x: Int, y: Int) {  
        self.x = x  
        self.y = y  
    }  
    var p1 = Point(x: 10, y: 10)  
    var p2 = Point(y: 10)  
    var p3 = Point(x: 10)  
    var p4 = Point()
```



窥探初始化器的本质

■ 以下2段代码完全等效

```
struct Point {  
    var x: Int = 0  
    var y: Int = 0  
}  
var p = Point()
```

```
struct Point {  
    var x: Int  
    var y: Int  
    init() {  
        x = 0  
        y = 0  
    }  
}  
var p = Point()
```



结构体内存结构

```
struct Point {  
    var x: Int = 0  
    var y: Int = 0  
    var origin: Bool = false  
}  
print(MemoryLayout<Point>.size) // 17  
print(MemoryLayout<Point>.stride) // 24  
print(MemoryLayout<Point>.alignment) // 8
```

类

■ 类的定义和结构体类似，但编译器并没有为类自动生成可以传入成员值的初始化器

```
class Point {  
    var x: Int = 0  
    var y: Int = 0  
}  
  
let p1 = Point()  
let p2 = Point(x: 10, y: 20)  
let p3 = Point(x: 10)  
let p4 = Point(y: 20)
```

```
struct Point {  
    var x: Int = 0  
    var y: Int = 0  
}  
  
let p1 = Point()  
let p2 = Point(x: 10, y: 20)  
let p3 = Point(x: 10)  
let p4 = Point(y: 20)
```

```
class Point {  
    var x: Int  
    var y: Int  
}  
  
let p1 = Point()
```

! Class 'Point' has no initializers

! 'Point' cannot be constructed because it has no accessible initializers



类的初始化器

- 如果类的所有成员都在定义的时候指定了初始值，编译器会为类生成无参的初始化器
- 成员的初始化是在这个初始化器中完成的

```
class Point {  
    var x: Int = 10  
    var y: Int = 20  
}  
let p1 = Point()
```

```
class Point {  
    var x: Int  
    var y: Int  
    init() {  
        x = 10  
        y = 20  
    }  
}  
let p1 = Point()
```

- 上面2段代码是完全等效的

结构体与类的本质区别

- 结构体是值类型（枚举也是值类型），类是引用类型（指针类型）

```
class Size {
    var width = 1
    var height = 2
}
```

```
struct Point {
    var x = 3
    var y = 4
}
```

```
func test() {
    var size = Size()
    var point = Point()
```

栈空间

内存地址	内存数据		说明
0x10000	3	point	point.x
0x10008	4		point.y
0x10010	0x90000	size	Size对象的内存地址

堆空间

内存地址	内存数据	说明
0x90000	0xe41a8	指向类型信息
0x90008	0x20002	
0x90010	1	
0x90018	2	
		Size对象

- 上图都是针对64bit环境

值类型

- 值类型赋值给 `var`、`let` 或者给函数传参，是直接将所有内容拷贝一份
- 类似于对文件进行 `copy`、`paste` 操作，产生了全新的文件副本。属于深拷贝（deep copy）

```
struct Point {  
    var x: Int  
    var y: Int  
}
```

```
func test() {  
    var p1 = Point(x: 10, y: 20)  
    var p2 = p1  
}
```

栈空间

内存地址	内存数据		说明
0x10000	10	p2	p2.x
0x10008	20		p2.y
0x10010	10	p1	p1.x
0x10018	20		p1.y

```
p2.x = 11  
p2.y = 22  
// 请问 p1.x 和 p1.y 是多少?
```



值类型的赋值操作

```
var s1 = "Jack"
var s2 = s1
s2.append("_Rose")
print(s1) // Jack
print(s2) // Jack_Rose
```

```
var a1 = [1, 2, 3]
var a2 = a1
a2.append(4)
a1[0] = 2
print(a1) // [2, 2, 3]
print(a2) // [1, 2, 3, 4]
```

```
var d1 = ["max" : 10, "min" : 2]
var d2 = d1
d1["other"] = 7
d2["max"] = 12
print(d1) // ["other": 7, "max": 10, "min": 2]
print(d2) // ["max": 12, "min": 2]
```

- 在Swift标准库中，为了提升性能，**String、Array、Dictionary、Set**采取了Copy On Write的技术
- 比如仅当有“写”操作时，才会真正执行拷贝操作
- 对于标准库值类型的赋值操作，Swift能确保最佳性能，所有没必要为了保证最佳性能来避免赋值
- 建议：不需要修改的，尽量定义成**let**

值类型的赋值操作

```
struct Point {  
    var x: Int  
    var y: Int  
}  
  
var p1 = Point(x: 10, y: 20)  
p1 = Point(x: 11, y: 22)
```

栈空间

内存地址	内存数据	说明	
0x10010	10	p1	p1.x
0x10018	20		p1.y

栈空间

内存地址	内存数据	说明	
0x10010	11	p1	p1.x
0x10018	22		p1.y

引用类型

- 引用赋值给 `var`、`let` 或者给函数传参，是将内存地址拷贝一份
- 类似于制作一个文件的替身（快捷方式、链接），指向的是同一个文件。属于浅拷贝（ shallow copy ）

```
class Size {
    var width: Int
    var height: Int
    init(width: Int, height: Int) {
        self.width = width
        self.height = height
    }
}
```

```
func test() {
    var s1 = Size(width: 10, height: 20)
    var s2 = s1
}
```

```
s2.width = 11
s2.height = 22
// 请问s1.width和s1.height是多少?
```

堆空间

栈空间

内存地址	内存数据	
0x10000	0x90000	s2
0x10008	0x90000	s1

内存地址	内存数据	说明
0x90000	0xe41a8	指向类型信息
0x90008	0x20002	引用计数
0x90010	10	width
0x90018	20	height

对象的堆空间申请过程

- 在Swift中，创建类的实例对象，要向堆空间申请内存，大概流程如下

□ `Class.__allocating_init()`

□ `libswiftCore.dylib : _swift_allocObject_`

□ `libswiftCore.dylib : swift_slowAlloc`

□ `libsystem_malloc.dylib : malloc`

- 在Mac、iOS中的`malloc`函数分配的内存大小总是16的倍数

- 通过`class_getInstanceSize`可以得知：类的对象至少需要占用多少内存

```
class Point {  
    var x = 11  
    var test = true  
    var y = 22  
}  
  
var p = Point()  
class_getInstanceSize(type(of: p)) // 40  
class_getInstanceSize(Point.self) // 40
```

引用类型的赋值操作

```
class Size {
    var width: Int
    var height: Int
    init(width: Int, height: Int) {
        self.width = width
        self.height = height
    }
    var s1 = Size(width: 10, height: 20)
    s1 = Size(width: 11, height: 22)
}
```

栈空间

内存地址	内存数据
0x10000	0x90000
0x10000	0x80000

堆空间

内存地址	内存数据	说明
0x90000	0xe41a8	Size对象
0x90008	0x20002	
0x90010	10	
0x90018	20	

内存地址	内存数据	说明
0x80000	0xe41a8	Size对象
0x80008	0x20002	
0x80010	11	
0x80018	22	

值类型、引用类型的let

```
struct Point {  
    var x: Int  
    var y: Int  
}  
  
class Size {  
    var width: Int  
    var height: Int  
    init(width: Int, height: Int) {  
        self.width = width  
        self.height = height  
    }  
}
```

```
let p = Point(x: 10, y: 20)  
p = Point(x: 11, y: 22)  
p.x = 33  
p.y = 44  
  
let s = Size(width: 10, height: 20)  
s = Size(width: 11, height: 22)  
s.width = 33  
s.height = 44
```

```
let str = "Jack"  
str.append("_Rose")  
  
let arr = [1, 2, 3]  
arr[0] = 11  
arr.append(4)
```



嵌套类型

```
struct Poker {  
    enum Suit : Character {  
        case spades = "♠", hearts = "♥", diamonds = "♦", clubs = "♣"  
    }  
    enum Rank : Int {  
        case two = 2, three, four, five, six, seven, eight, nine, ten  
        case jack, queen, king, ace  
    }  
}
```

```
print(Poker.Suit.hearts.rawValue)
```

```
var suit = Poker.Suit.spades  
suit = .diamonds
```

```
var rank = Poker.Rank.five  
rank = .king
```

枚举、结构体、类都可以定义方法

- 一般把定义在枚举、结构体、类内部的函数，叫做方法

```
class Size {  
    var width = 10  
    var height = 10  
    func show() {  
        print("width=\(width), height=\(height)")  
    }  
}  
  
let s = Size()  
s.show() // width=10, height=10
```

```
struct Point {  
    var x = 10  
    var y = 10  
    func show() {  
        print("x=\(x), y=\(y)")  
    }  
}  
  
let p = Point()  
p.show() // x=10, y=10
```

```
enum PokerFace : Character {  
    case spades = "♠", hearts = "♥", diamonds = "♦", clubs = "♣"  
    func show() {  
        print("face is \(rawValue)")  
    }  
}  
  
let pf = PokerFace.hearts  
pf.show() // face is ♥
```

- 方法占用对象的内存么？
 - 不占用
 - 方法的本质就是函数
 - 方法、函数都存放在代码段

作业

■ 思考以下结构体、类对象的内存结构是怎样的？

```
struct Point {  
    var x: Int  
    var b1: Bool  
    var b2: Bool  
    var y: Int  
}  
  
var p = Point(x: 10, b1: true, b2: true, y: 20)
```

```
class Size {  
    var width: Int  
    var b1: Bool  
    var b2: Bool  
    var height: Int  
    init(width: Int, b1: Bool, b2: Bool, height: Int) {  
        self.width = width  
        self.b1 = b1  
        self.b2 = b2  
        self.height = height  
    }  
}  
  
var s = Size(width: 10, b1: true, b2: true, height: 20)
```

闭包

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





闭包表达式 (Closure Expression)

- 在Swift中，可以通过`func`定义一个函数，也可以通过闭包表达式定义一个函数

```
func sum(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }
```

```
var fn = {
    (v1: Int, v2: Int) -> Int in
    return v1 + v2
}
fn(10, 20)
```

```
{
    (v1: Int, v2: Int) -> Int in
    return v1 + v2
}(10, 20)
```

```
{
    (参数列表) -> 返回值类型 in
    函数体代码
}
```



闭包表达式的简写

```
func exec(v1: Int, v2: Int, fn: (Int, Int) -> Int) {  
    print(fn(v1, v2))  
}
```

```
exec(v1: 10, v2: 20, fn: {  
    (v1: Int, v2: Int) -> Int in  
    return v1 + v2  
})
```

```
exec(v1: 10, v2: 20, fn: {  
    v1, v2 in return v1 + v2  
})
```

```
exec(v1: 10, v2: 20, fn: {  
    v1, v2 in v1 + v2  
})
```

```
exec(v1: 10, v2: 20, fn: { $0 + $1 })
```

```
exec(v1: 10, v2: 20, fn: +)
```

尾随闭包

- 如果将一个很长的闭包表达式作为函数的最后一个实参，使用尾随闭包可以增强函数的可读性
- 尾随闭包是一个被书写在函数调用括号外面（后面）的闭包表达式

```
func exec(v1: Int, v2: Int, fn: (Int, Int) -> Int) {  
    print(fn(v1, v2))  
}
```

```
exec(v1: 10, v2: 20) {  
    $0 + $1  
}
```

- 如果闭包表达式是函数的唯一实参，而且使用了尾随闭包的语法，那就不需要在函数名后边写圆括号

```
func exec(fn: (Int, Int) -> Int) {  
    print(fn(1, 2))  
}
```

```
exec(fn: { $0 + $1 })  
exec() { $0 + $1 }  
exec { $0 + $1 }
```



示例 – 数组的排序

```
func sort(by areInIncreasingOrder: (Element, Element) -> Bool)
```

```
/// 返回true: i1排在i2前面
/// 返回false: i1排在i2后面
func cmp(i1: Int, i2: Int) -> Bool {
    // 大的排在前面
    return i1 > i2
}
```

```
var nums = [11, 2, 18, 6, 5, 68, 45]
nums.sort(by: cmp)
// [68, 45, 18, 11, 6, 5, 2]
```

```
nums.sort(by: {
    (i1: Int, i2: Int) -> Bool in
    return i1 < i2
})
nums.sort(by: { i1, i2 in return i1 < i2 })
nums.sort(by: { i1, i2 in i1 < i2 })
nums.sort(by: { $0 < $1 })
nums.sort(by: <)
nums.sort() { $0 < $1 }
nums.sort { $0 < $1 }
// [2, 5, 6, 11, 18, 45, 68]
```



忽略参数

```
func exec(fn: (Int, Int) -> Int) {  
    print(fn(1, 2))  
}  
exec { _,_ in 10 } // 10
```

闭包 (Closure)

- 网上有各种关于闭包的定义，个人觉得比较严谨的定义是
- 一个函数和它所捕获的变量\常量环境组合起来，称为闭包
- ✓ 一般指定义在函数内部的函数
- ✓ 一般它捕获的是外层函数的局部变量\常量

```
typealias Fn = (Int) -> Int
func getFn() -> Fn {
    var num = 0
    func plus(_ i: Int) -> Int {
        num += i
        return num
    }
    return plus
} // 返回的plus和num形成了闭包
```

```
func getFn() -> Fn {
    var num = 0
    return {
        num += $0
        return num
    }
}
```

```
var fn1 = getFn()
var fn2 = getFn()
fn1(1) // 1
fn2(2) // 2
fn1(3) // 4
fn2(4) // 6
fn1(5) // 9
fn2(6) // 12
```

■ 思考：如果num是全局变量呢？

- 可以把闭包想象成是一个类的实例对象
- 内存在堆空间
- 捕获的局部变量\常量就是对象的成员（存储属性）
- 组成闭包的函数就是类内部定义的方法

```
class Closure {
    var num = 0
    func plus(_ i: Int) -> Int {
        num += i
        return num
    }
}
var cs1 = Closure()
var cs2 = Closure()
cs1.plus(1) // 1
cs2.plus(2) // 2
cs1.plus(3) // 4
cs2.plus(4) // 6
cs1.plus(5) // 9
cs2.plus(6) // 12
```

练习

```
typealias Fn = (Int) -> (Int, Int)
func getFns() -> (Fn, Fn) {
    var num1 = 0
    var num2 = 0
    func plus(_ i: Int) -> (Int, Int) {
        num1 += i
        num2 += i << 1
        return (num1, num2)
    }
    func minus(_ i: Int) -> (Int, Int) {
        num1 -= i
        num2 -= i << 1
        return (num1, num2)
    }
    return (plus, minus)
}
```

```
let (p, m) = getFns()
p(5) // (5, 10)
m(4) // (1, 2)
p(3) // (4, 8)
m(2) // (2, 4)
```

```
class Closure {
    var num1 = 0
    var num2 = 0
    func plus(_ i: Int) -> (Int, Int) {
        num1 += i
        num2 += i << 1
        return (num1, num2)
    }
    func minus(_ i: Int) -> (Int, Int) {
        num1 -= i
        num2 -= i << 1
        return (num1, num2)
    }
}
```

```
var cs = Closure()
cs.plus(5) // (5, 10)
cs.minus(4) // (1, 2)
cs.plus(3) // (4, 8)
cs.minus(2) // (2, 4)
```

练习

```
var functions: [() -> Int] = []
for i in 1...3 {
    functions.append { i }
}
for f in functions {
    print(f())
}
```

```
// 1
// 2
// 3
```

```
class Closure {
    var i: Int
    init(_ i: Int) {
        self.i = i
    }
    func get() -> Int {
        return i
    }
}
var cses: [Closure] = []
for i in 1...3 {
    cses.append(Closure(i))
}
for cls in cses {
    print(cls.get())
}
```



注意

- 如果返回值是函数类型，那么参数的修饰要保持统一

```
func add(_ num: Int) -> (inout Int) -> Void {  
    func plus(v: inout Int) {  
        v += num  
    }  
    return plus  
}  
var num = 5  
add(20)(&num)  
print(num)
```

自动闭包

```
// 如果第1个数大于0，返回第一个数。否则返回第2个数
func getFirstPositive(_ v1: Int, _ v2: Int) -> Int {
    return v1 > 0 ? v1 : v2
}
getFirstPositive(10, 20) // 10
getFirstPositive(-2, 20) // 20
getFirstPositive(0, -4) // -4
```

```
// 改成函数类型的参数，可以让v2延迟加载
func getFirstPositive(_ v1: Int, _ v2: () -> Int) -> Int? {
    return v1 > 0 ? v1 : v2()
}
getFirstPositive(-4) { 20 }
```

```
func getFirstPositive(_ v1: Int, _ v2: @autoclosure () -> Int) -> Int? {
    return v1 > 0 ? v1 : v2()
}
getFirstPositive(-4, 20)
```

- `@autoclosure` 会自动将 20 封装成闭包 `{ 20 }`
- `@autoclosure` 只支持 `() -> T` 格式的参数
- `@autoclosure` 并非只支持最后一个参数
- 空合并运算符 ?? 使用了 `@autoclosure` 技术
- 有`@autoclosure`、无`@autoclosure`，构成了函数重载

- 为了避免与期望冲突，使用了`@autoclosure`的地方最好明确注释清楚：这个值会被推迟执行

属性

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



属性

■ Swift中跟实例相关的属性可以分为2大类

口存储属性 (Stored Property)

- ✓ 类似于成员变量这个概念
- ✓ 存储在实例的内存中
- ✓ 结构体、类可以定义存储属性
- ✓ 枚举不可以定义存储属性

口计算属性 (Computed Property)

- ✓ 本质就是方法 (函数)
- ✓ 不占用实例的内存
- ✓ 枚举、结构体、类都可以定义计算属性

```
print(MemoryLayout<Circle>.stride) // 8
```

```
struct Circle {  
    // 存储属性  
    var radius: Double  
    // 计算属性  
    var diameter: Double {  
        set {  
            radius = newValue / 2  
        }  
        get {  
            radius * 2  
        }  
    }  
}
```

```
var circle = Circle(radius: 5)  
print(circle.radius) // 5.0  
print(circle.diameter) // 10.0
```

```
circle.diameter = 12  
print(circle.radius) // 6.0  
print(circle.diameter) // 12.0
```



存储属性

- 关于存储属性，Swift有个明确的规定
- 在创建类 或 结构体的实例时，必须为所有的存储属性设置一个合适的初始值
- ✓ 可以在初始化器里为存储属性设置一个初始值
- ✓ 可以分配一个默认的属性值作为属性定义的一部分

计算属性

- **set**传入的新值默认叫做**newValue**，也可以自定义

```
struct Circle {  
    var radius: Double  
    var diameter: Double {  
        set(newDiameter) {  
            radius = newDiameter / 2  
        }  
        get {  
            return radius * 2  
        }  
    }  
}
```

- 只读计算属性：只有**get**，没有**set**

```
struct Circle {  
    var radius: Double  
    var diameter: Double {  
        get {  
            return radius * 2  
        }  
    }  
}
```

```
struct Circle {  
    var radius: Double  
    var diameter: Double {  
        get {  
            return radius * 2  
        }  
    }  
}
```

- 定义计算属性只能用**var**，不能用**let**

□ **let**代表常量：值是一成不变的

□ 计算属性的值是可能发生变化的（即使是只读计算属性）



枚举rawValue原理

- 枚举原始值rawValue的本质是：只读计算属性

```
enum TestEnum : Int {  
    case test1 = 1, test2 = 2, test3 = 3  
    var rawValue: Int {  
        switch self {  
            case .test1:  
                return 10  
            case .test2:  
                return 11  
            case .test3:  
                return 12  
        }  
    }  
}
```

```
print(TestEnum.test3.rawValue) // 12
```



延迟存储属性 (Lazy Stored Property)

- 使用 `lazy` 可以定义一个延迟存储属性，在第一次用到属性的时候才会进行初始化

```
class Car {  
    init() {  
        print("Car init!")  
    }  
    func run() {  
        print("Car is running!")  
    }  
}
```

```
class Person {  
    lazy var car = Car()  
    init() {  
        print("Person init!")  
    }  
    func goOut() {  
        car.run()  
    }  
}
```

```
let p = Person()  
print("-----")  
p.goOut()
```

```
Person init!  
-----  
Car init!  
Car is running!
```

```
class PhotoView {  
    lazy var image: Image = {  
        let url = "https://www.520it.com/xx.png"  
        let data = Data(url: url)  
        return Image(data: data)  
    }()  
}
```

- `lazy` 属性必须是 `var`，不能是 `let`
- `let` 必须在实例的初始化方法完成之前就拥有值
- 如果多条线程同时第一次访问 `lazy` 属性
- 无法保证属性只被初始化1次

延迟存储属性注意点

- 当结构体包含一个延迟存储属性时，只有`var`才能访问延迟存储属性
- 因为延迟属性初始化时需要改变结构体的内存

```
struct Point {  
    var x = 0  
    var y = 0  
    lazy var z = 0  
}  
let p = Point()  
print(p.z)  ⚡ Cannot use mutating getter on immutable value: 'p' is a 'let' constant
```



属性观察器 (Property Observer)

- 可以为非`lazy`的`var`存储属性设置属性观察器

```
struct Circle {  
    var radius: Double {  
        willSet {  
            print("willSet", newValue)  
        }  
        didSet {  
            print("didSet", oldValue, radius)  
        }  
    }  
    init() {  
        self.radius = 1.0  
        print("Circle init!")  
    }  
}
```

```
// Circle init!  
var circle = Circle()  
  
// willSet 10.5  
// didSet 1.0 10.5  
circle.radius = 10.5  
  
// 10.5  
print(circle.radius)
```

- `willSet`会传递新值，默认叫`newValue`
- `didSet`会传递旧值，默认叫`oldValue`
- 在初始化器中设置属性值不会触发`willSet`和`didSet`
- 在属性定义时设置初始值也不会触发`willSet`和`didSet`



全局变量、局部变量

- 属性观察器、计算属性的功能，同样可以应用在全局变量、局部变量身上

```
var num: Int {  
    get {  
        return 10  
    }  
    set {  
        print("setNum", newValue)  
    }  
}  
  
num = 11 // setNum 11  
print(num) // 10
```

```
func test() {  
    var age = 10 {  
        willSet {  
            print("willSet", newValue)  
        }  
        didSet {  
            print("didSet", oldValue, age)  
        }  
    }  
    age = 11  
    // willSet 11  
    // didSet 10 11  
}  
test()
```



inout的再次研究

```
struct Shape {  
    var width: Int  
    var side: Int {  
        willSet {  
            print("willSetSide", newValue)  
        }  
        didSet {  
            print("didSetSide", oldValue, side)  
        }  
    }  
    var girth: Int {  
        set {  
            width = newValue / side  
            print("setGirth", newValue)  
        }  
        get {  
            print("getGirth")  
            return width * side  
        }  
    }  
    func show() {  
        print("width=\(width), side=\(side), girth=\(girth)")  
    }  
}
```

```
func test(_ num: inout Int) {  
    num = 20  
}  
  
var s = Shape(width: 10, side: 4)  
test(&s.width)  
s.show()  
print("-----")  
test(&s.side)  
s.show()  
print("-----")  
test(&s.girth)  
s.show()
```

```
getGirth  
width=20, side=4, girth=80  
-----  
willSetSide 20  
didSetSide 4 20  
getGirth  
width=20, side=20, girth=400  
-----  
getGirth  
setGirth 20  
getGirth  
width=1, side=20, girth=20
```



inout的本质总结

- 如果实参有物理内存地址，且没有设置属性观察器
 - 直接将实参的内存地址传入函数（实参进行引用传递）
- 如果实参是计算属性 或者 设置了属性观察器
 - 采取了Copy In Copy Out的做法
 - ✓ 调用该函数时，先复制实参的值，产生副本【get】
 - ✓ 将副本的内存地址传入函数（副本进行引用传递），在函数内部可以修改副本的值
 - ✓ 函数返回后，再将副本的值覆盖实参的值【set】
- 总结：inout的本质就是引用传递（地址传递）

类型属性 (Type Property)

■ 严格来说，属性可以分为

□ 实例属性 (Instance Property)：只能通过实例去访问

✓ 存储实例属性 (Stored Instance Property)：存储在实例的内存中，每个实例都有1份

✓ 计算实例属性 (Computed Instance Property)

□ 类型属性 (Type Property)：只能通过类型去访问

✓ 存储类型属性 (Stored Type Property)：整个程序运行过程中，就只有1份内存（类似于全局变量）

✓ 计算类型属性 (Computed Type Property)

■ 可以通过 **static** 定义类型属性

□ 如果是类，也可以用关键字 **class**

```
struct Car {  
    static var count: Int = 0  
    init() {  
        Car.count += 1  
    }  
}
```

```
let c1 = Car()  
let c2 = Car()  
let c3 = Car()  
print(Car.count) // 3
```



类型属性细节

- 不同于存储实例属性，你必须给存储类型属性设定初始值
 - 因为类型没有像实例那样的 `init` 初始化器来初始化存储属性
- 存储类型属性默认就是 `lazy`，会在第一次使用的时候才初始化
 - 就算被多个线程同时访问，保证只会初始化一次
 - 存储类型属性可以是 `let`
- 枚举类型也可以定义类型属性（存储类型属性、计算类型属性）



单例模式

```
public class FileManager {  
    public static let shared = FileManager()  
    private init() { }  
}
```

```
public class FileManager {  
    public static let shared = {  
        // ....  
        // ....  
        return FileManager()  
    }()  
    private init() { }  
}
```

方法

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



方法 (Method)

- 枚举、结构体、类都可以定义实例方法、类型方法
- 实例方法 (Instance Method) : 通过实例对象调用
- 类型方法 (Type Method) : 通过类型调用, 用 **static** 或者 **class** 关键字定义

```
class Car {  
    static var cout = 0  
    init() {  
        Car.cout += 1  
    }  
    static func getCount() -> Int { cout }  
}  
  
let c0 = Car()  
let c1 = Car()  
let c2 = Car()  
print(Car.getCount()) // 3
```

- **self**

- 在实例方法中代表实例对象
- 在类型方法中代表类型

- 在类型方法 **static func getCount** 中

- **cout** 等价于 **self.cout**、**Car.self.cout**、**Car.cout**

mutating

■ 结构体和枚举是值类型，默认情况下，值类型的属性不能被自身的实例方法修改

□ 在 `func` 关键字前加 `mutating` 可以允许这种修改行为

```
struct Point {  
    var x = 0.0, y = 0.0  
    mutating func moveBy(deltaX: Double, deltaY: Double) {  
        x += deltaX  
        y += deltaY  
        // self = Point(x: x + deltaX, y: y + deltaY)  
    }  
}
```

```
enum StateSwitch {  
    case low, middle, high  
    mutating func next() {  
        switch self {  
            case .low:  
                self = .middle  
            case .middle:  
                self = .high  
            case .high:  
                self = .low  
        }  
    }  
}
```



@discardableResult

■ 在func前面加个`@discardableResult`，可以消除：函数调用后返回值未被使用的警告⚠

```
struct Point {  
    var x = 0.0, y = 0.0  
    @discardableResult mutating  
    func moveX(deltaX: Double) -> Double {  
        x += deltaX  
        return x  
    }  
}  
var p = Point()  
p.moveX(deltaX: 10)
```

```
@discardableResult  
func get() -> Int {  
    return 10  
}  
get()
```

下标

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



下标 (subscript)

- 使用 **subscript** 可以给任意类型（枚举、结构体、类）增加下标功能，有些地方也翻译为：下标脚本
- **subscript** 的语法类似于实例方法、计算属性，本质就是方法（函数）

```
class Point {  
    var x = 0.0, y = 0.0  
    subscript(index: Int) -> Double {  
        set {  
            if index == 0 {  
                x = newValue  
            } else if index == 1 {  
                y = newValue  
            }  
        }  
        get {  
            if index == 0 {  
                return x  
            } else if index == 1 {  
                return y  
            }  
            return 0  
        }  
    }  
}
```

```
var p = Point()  
p[0] = 11.1  
p[1] = 22.2  
print(p.x) // 11.1  
print(p.y) // 22.2  
print(p[0]) // 11.1  
print(p[1]) // 22.2
```

- **subscript** 中定义的返回值类型决定了
- **get** 方法的返回值类型
- **set** 方法中 **newValue** 的类型
- **subscript** 可以接受多个参数，并且类型任意

下标的细节

- subscript可以没有set方法，但必须要有get方法

```
class Point {  
    var x = 0.0, y = 0.0  
    subscript(index: Int) -> Double {  
        get {  
            if index == 0 {  
                return x  
            } else if index == 1 {  
                return y  
            }  
            return 0  
        }  
    }  
}
```

- 如果只有get方法，可以省略get

```
class Point {  
    var x = 0.0, y = 0.0  
    subscript(index: Int) -> Double {  
        if index == 0 {  
            return x  
        } else if index == 1 {  
            return y  
        }  
        return 0  
    }  
}
```

下标的细节

■ 可以设置参数标签

```
class Point {  
    var x = 0.0, y = 0.0  
    subscript(index i: Int) -> Double {  
        if i == 0 {  
            return x  
        } else if i == 1 {  
            return y  
        }  
        return 0  
    }  
}
```

```
var p = Point()  
p.y = 22.2  
print(p[index: 1]) // 22.2
```

■ 下标可以是类型方法

```
class Sum {  
    static subscript(v1: Int, v2: Int) -> Int {  
        return v1 + v2  
    }  
}  
print(Sum[10, 20]) // 30
```



结构体、类作为返回值对比

```
class Point {  
    var x = 0, y = 0  
}  
  
class PointManager {  
    var point = Point()  
    subscript(index: Int) -> Point {  
        get { point }  
    }  
}
```

```
var pm = PointManager()  
pm[0].x = 11  
pm[0].y = 22  
// Point(x: 11, y: 22)  
print(pm[0])  
// Point(x: 11, y: 22)  
print(pm.point)
```

```
struct Point {  
    var x = 0, y = 0  
}  
  
class PointManager {  
    var point = Point()  
    subscript(index: Int) -> Point {  
        set { point = newValue }  
        get { point }  
    }  
}
```



接收多个参数的下标

```
class Grid {  
    var data = [  
        [0, 1, 2],  
        [3, 4, 5],  
        [6, 7, 8]  
    ]  
    subscript(row: Int, column: Int) -> Int {  
        set {  
            guard row >= 0 && row < 3 && column >= 0 && column < 3 else {  
                return  
            }  
            data[row][column] = newValue  
        }  
        get {  
            guard row >= 0 && row < 3 && column >= 0 && column < 3 else {  
                return 0  
            }  
            return data[row][column]  
        }  
    }  
}
```

```
var grid = Grid()  
grid[0, 1] = 77  
grid[1, 2] = 88  
grid[2, 0] = 99  
print(grid.data)
```

继承

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





继承 (Inheritance)

- 值类型 (枚举、结构体) 不支持继承，只有类支持继承
- 没有父类的类，称为：基类
- Swift 并没有像 OC、Java 那样的规定：任何类最终都要继承自某个基类

NOTE

Swift classes do not inherit from a universal base class. Classes you define without specifying a superclass automatically become base classes for you to build upon.

- 子类可以重写父类的下标、方法、属性，重写必须加上 `override` 关键字



内存结构

```
class Animal {  
    var age = 0  
}  
  
class Dog : Animal {  
    var weight = 0  
}  
  
class ErHa : Dog {  
    var iq = 0  
}
```

```
let a = Animal()  
a.age = 10  
// 32  
print(Mems.size(ofRef: a))  
/*  
0x0000001000073e0  
0x0000000000000002  
0x000000000000000a  
0x0000000000000000  
*/  
print(Mems.memStr(ofRef: a))
```

```
let d = Dog()  
d.age = 10  
d.weight = 20  
// 32  
print(Mems.size(ofRef: d))  
/*  
0x000000100007490  
0x0000000000000002  
0x000000000000000a  
0x0000000000000014  
*/  
print(Mems.memStr(ofRef: d))
```

```
let e = ErHa()  
e.age = 10  
e.weight = 20  
e.iq = 30  
// 48  
print(Mems.size(ofRef: e))  
/*  
0x000000100007560  
0x0000000000000002  
0x000000000000000a  
0x0000000000000014  
0x0000000000000001e  
0x0000000000000000  
*/  
print(Mems.memStr(ofRef: e))
```



重写实例方法、下标

```
class Animal {  
    func speak() {  
        print("Animal speak")  
    }  
    subscript(index: Int) -> Int {  
        return index  
    }  
}
```

```
class Cat : Animal {  
    override func speak() {  
        super.speak()  
        print("Cat speak")  
    }  
    override subscript(index: Int) -> Int {  
        return super[index] + 1  
    }  
}
```

```
var anim: Animal  
anim = Animal()  
// Animal speak  
anim.speak()  
// 6  
print(anim[6])
```

```
anim = Cat()  
// Animal speak  
// Cat speak  
anim.speak()  
// 7  
print(anim[6])
```



重写类型方法、下标

- 被`class`修饰的类型方法、下标，允许被子类重写
- 被`static`修饰的类型方法、下标，不允许被子类重写

```
class Animal {  
    class func speak() {  
        print("Animal speak")  
    }  
    class subscript(index: Int) -> Int {  
        return index  
    }  
}  
  
// Animal speak  
Animal.speak()  
// 6  
print(Animal[6])
```

```
class Cat : Animal {  
    override class func speak() {  
        super.speak()  
        print("Cat speak")  
    }  
    override class subscript(index: Int) -> Int {  
        return super[index] + 1  
    }  
}  
  
// Animal speak  
// Cat speak  
Cat.speak()  
// 7  
print(Cat[6])
```



重写属性

- 子类可以将父类的属性（存储、计算）重写为计算属性
- 子类不可以将父类属性重写为存储属性
- 只能重写`var`属性，不能重写`let`属性
- 重写时，属性名、类型要一致
- 子类重写后的属性权限 不能小于 父类属性的权限
 - 如果父类属性是只读的，那么子类重写后的属性可以是只读的、也可以是可读写的
 - 如果父类属性是可读写的，那么子类重写后的属性也必须是可读写的



重写实例属性

```
class Circle {  
    var radius: Int = 0  
    var diameter: Int {  
        set {  
            print("Circle setDiameter")  
            radius = newValue / 2  
        }  
        get {  
            print("Circle getDiameter")  
            return radius * 2  
        }  
    }  
}
```

```
var circle: Circle  
circle = Circle()  
circle.radius = 6  
// Circle getDiameter  
// 12  
print(circle.diameter)  
// Circle setDiameter  
circle.diameter = 20  
// 10  
print(circle.radius)
```



重写实例属性

```
class SubCircle : Circle {  
    override var radius: Int {  
        set {  
            print("SubCircle setRadius")  
            super.radius = newValue > 0 ? newValue : 0  
        }  
        get {  
            print("SubCircle getRadius")  
            return super.radius  
        }  
    }  
    override var diameter: Int {  
        set {  
            print("SubCircle setDiameter")  
            super.diameter = newValue > 0 ? newValue : 0  
        }  
        get {  
            print("SubCircle getDiameter")  
            return super.diameter  
        }  
    }  
}
```

```
circle = SubCircle()  
  
// SubCircle setRadius  
circle.radius = 6  
  
// SubCircle getDiameter  
// Circle getDiameter  
// SubCircle getRadius  
// 12  
print(circle.diameter)  
  
// SubCircle setDiameter  
// Circle setDiameter  
// SubCircle setRadius  
circle.diameter = 20  
  
// SubCircle getRadius  
// 10  
print(circle.radius)
```



重写类型属性

- 被 `class` 修饰的计算类型属性，可以被子类重写
- 被 `static` 修饰的类型属性（存储、计算），不可以被子类重写

```
class Circle {  
    static var radius: Int = 0  
    class var diameter: Int {  
        set {  
            print("Circle setDiameter")  
            radius = newValue / 2  
        }  
        get {  
            print("Circle getDiameter")  
            return radius * 2  
        }  
    }  
}
```

```
class SubCircle : Circle {  
    override static var diameter: Int {  
        set {  
            print("SubCircle setDiameter")  
            super.diameter = newValue > 0 ? newValue : 0  
        }  
        get {  
            print("SubCircle getDiameter")  
            return super.diameter  
        }  
    }  
}
```

```
Circle.radius = 6  
// Circle getDiameter  
// 12  
print(Circle.diameter)  
// Circle setDiameter  
Circle.diameter = 20  
// 10  
print(Circle.radius)
```

```
SubCircle.radius = 6  
// SubCircle getDiameter  
// Circle getDiameter  
// 12  
print(SubCircle.diameter)  
// SubCircle setDiameter  
// Circle setDiameter  
SubCircle.diameter = 20  
// 10  
print(SubCircle.radius)
```

属性观察器

- 可以在子类中为父类属性（除了只读计算属性、`let`属性）增加属性观察器

```
class Circle {  
    var radius: Int = 1  
}  
  
class SubCircle : Circle {  
    override var radius: Int {  
        willSet {  
            print("SubCircle willSetRadius", newValue)  
        }  
        didSet {  
            print("SubCircle didSetRadius", oldValue, radius)  
        }  
    }  
}  
  
var circle = SubCircle()  
// SubCircle willSetRadius 10  
// SubCircle didSetRadius 1 10  
circle.radius = 10
```



属性观察器

```
class Circle {  
    var radius: Int = 1 {  
        willSet {  
            print("Circle willSetRadius", newValue)  
        }  
        didSet {  
            print("Circle didSetRadius", oldValue, radius)  
        }  
    }  
}  
  
class SubCircle : Circle {  
    override var radius: Int {  
        willSet {  
            print("SubCircle willSetRadius", newValue)  
        }  
        didSet {  
            print("SubCircle didSetRadius", oldValue, radius)  
        }  
    }  
}
```

```
var circle = SubCircle()  
// SubCircle willSetRadius 10  
// Circle willSetRadius 10  
// Circle didSetRadius 1 10  
// SubCircle didSetRadius 1 10  
circle.radius = 10
```



属性观察器

```
class Circle {  
    var radius: Int {  
        set {  
            print("Circle setRadius", newValue)  
        }  
        get {  
            print("Circle getRadius")  
            return 20  
        }  
    }  
}  
  
class SubCircle : Circle {  
    override var radius: Int {  
        willSet {  
            print("SubCircle willSetRadius", newValue)  
        }  
        didSet {  
            print("SubCircle didSetRadius", oldValue, radius)  
        }  
    }  
}
```

```
var circle = SubCircle()  
// Circle getRadius  
// SubCircle willSetRadius 10  
// Circle setRadius 10  
// Circle getRadius  
// SubCircle didSetRadius 20 20  
circle.radius = 10
```



属性观察器

```
class Circle {  
    class var radius: Int {  
        set {  
            print("Circle setRadius", newValue)  
        }  
        get {  
            print("Circle getRadius")  
            return 20  
        }  
    }  
}  
  
class SubCircle : Circle {  
    override static var radius: Int {  
        willSet {  
            print("SubCircle willSetRadius", newValue)  
        }  
        didSet {  
            print("SubCircle didSetRadius", oldValue, radius)  
        }  
    }  
}
```

```
// Circle getRadius  
// SubCircle willSetRadius 10  
// Circle setRadius 10  
// Circle getRadius  
// SubCircle didSetRadius 20 20  
SubCircle.radius = 10
```



final

- 被**final**修饰的方法、下标、属性，禁止被重写
- 被**final**修饰的类，禁止被继承

初始化

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



初始化器

■ 类、结构体、枚举都可以定义初始化器

■ **类**有2种初始化器：指定初始化器（designated initializer）、便捷初始化器（convenience initializer）

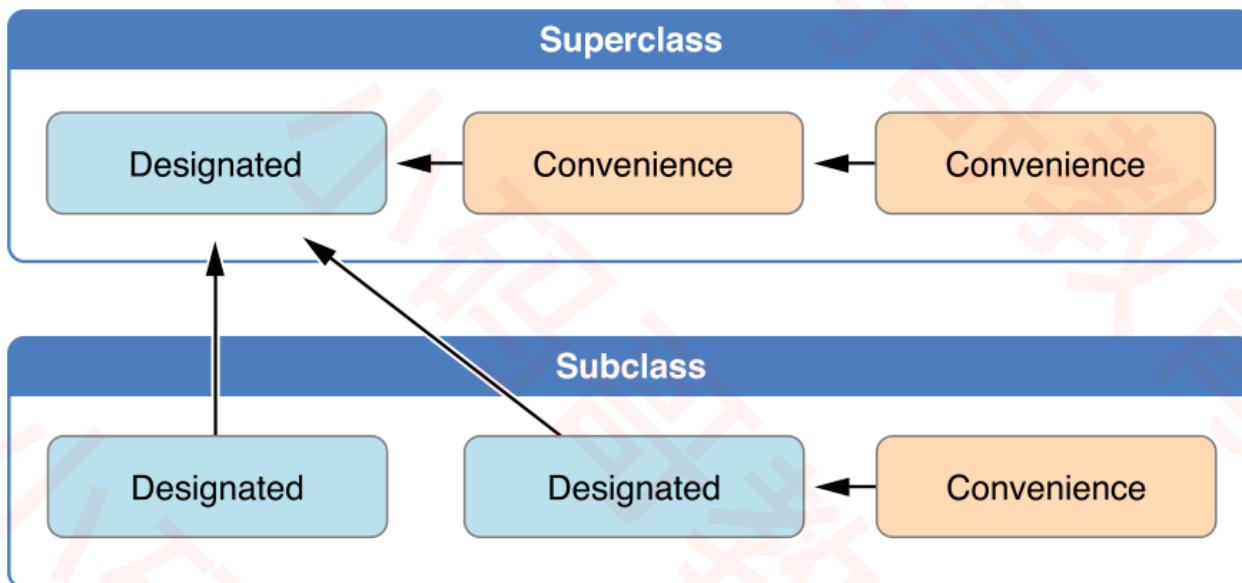
```
// 指定初始化器
init(parameters) {
    statements
}
```

```
// 便捷初始化器
convenience init(parameters) {
    statements
}
```

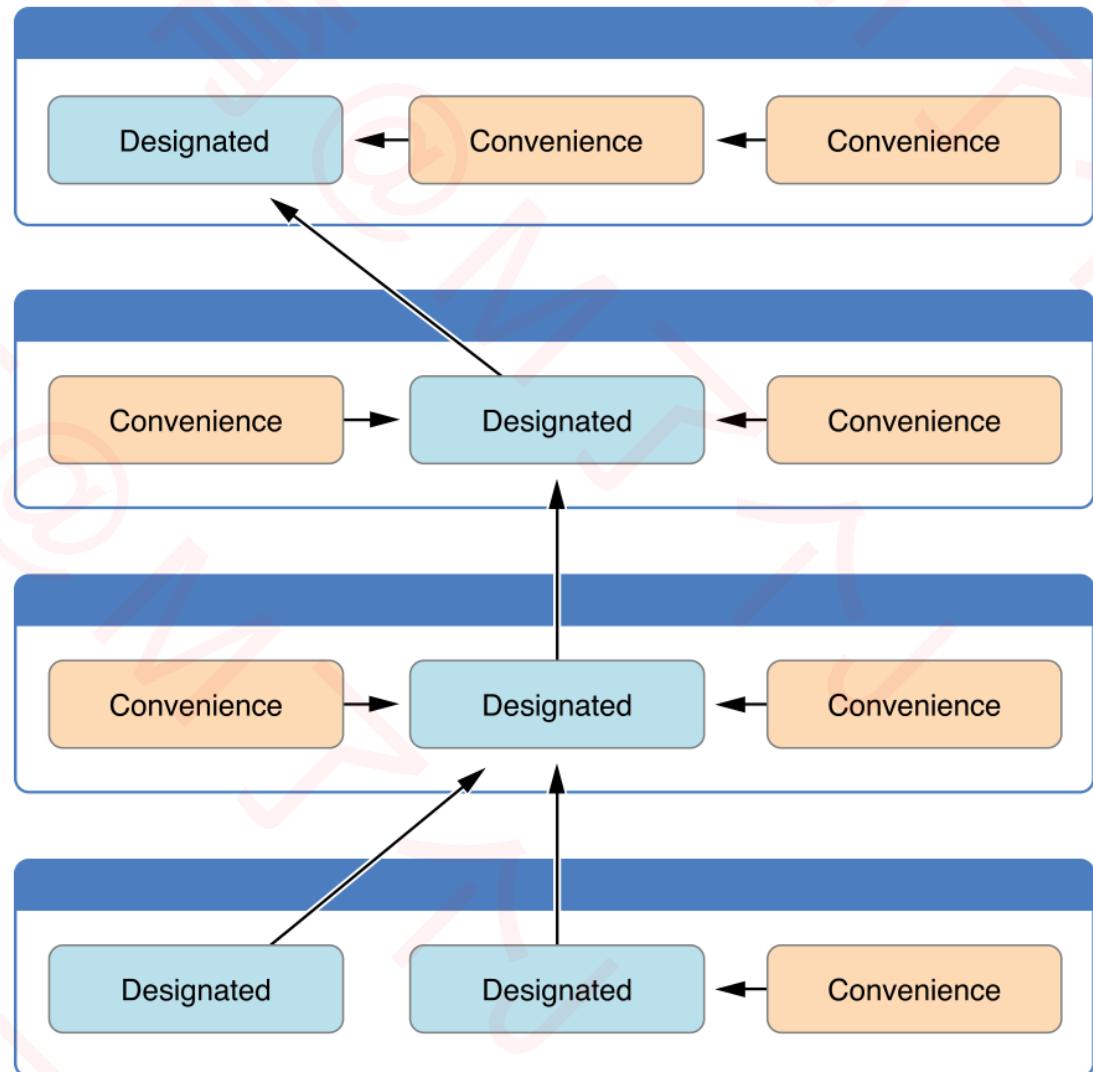
- 每个类至少有一个指定初始化器，指定初始化器是类的主要初始化器
- 默认初始化器总是类的指定初始化器
- 类偏向于少量指定初始化器，一个类通常只有一个指定初始化器

- 初始化器的相互调用规则
 - 指定初始化器必须从它的直系父类调用指定初始化器
 - 便捷初始化器必须从相同的类里调用另一个初始化器
 - 便捷初始化器最终必须调用一个指定初始化器

初始化器的相互调用



- 这一套规则保证了
- 使用任意初始化器，都可以完整地初始化实例





两段式初始化

■ Swift在编码安全方面是煞费苦心，为了保证初始化过程的安全，设定了**两段式初始化**、**安全检查**

■ 两段式初始化

□ 第1阶段：初始化所有存储属性

- ① 外层调用指定\便捷初始化器
- ② 分配内存给实例，但未初始化
- ③ 指定初始化器确保当前类定义的存储属性都初始化
- ④ 指定初始化器调用父类的初始化器，不断向上调用，形成初始化器链

□ 第2阶段：设置新的存储属性值

- ① 从顶部初始化器往下，链中的每一个指定初始化器都有机会进一步定制实例
- ② 初始化器现在能够使用`self`（访问、修改它的属性，调用它的实例方法等等）
- ③ 最终，链中任何便捷初始化器都有机会定制实例以及使用`self`



安全检查

- 指定初始化器必须保证在调用父类初始化器之前，其所在类定义的所有存储属性都要初始化完成
- 指定初始化器必须先调用父类初始化器，然后才能为继承的属性设置新值
- 便捷初始化器必须先调用同类中的其它初始化器，然后再为任意属性设置新值
- 初始化器在第1阶段初始化完成之前，不能调用任何实例方法、不能读取任何实例属性的值，也不能引用 `self`
- 直到第1阶段结束，实例才算完全合法



重写

- 当重写父类的指定初始化器时，必须加上**override**（即使子类的实现是便捷初始化器）
- 如果子类写了一个匹配父类便捷初始化器的初始化器，不用加上**override**
 - 因为父类的便捷初始化器永远不会通过子类直接调用，因此，严格来说，子类无法重写父类的便捷初始化器



自动继承

- ① 如果子类没有自定义任何指定初始化器，它会自动继承父类所有的指定初始化器
- ② 如果子类提供了父类所有指定初始化器的实现（要么通过方式①继承，要么重写）
 - 子类自动继承所有的父类便捷初始化器
- ③ 就算子类添加了更多的便捷初始化器，这些规则仍然适用
- ④ 子类以便捷初始化器的形式重写父类的指定初始化器，也可以作为满足规则②的一部分



required

- 用 **required** 修饰指定初始化器，表明其所有子类都必须实现该初始化器（通过继承或者重写实现）
- 如果子类重写了 **required** 初始化器，也必须加上 **required**，不用加 **override**

```
class Person {  
    required init() { }  
    init(age: Int) { }  
}  
  
class Student : Person {  
    required init() {  
        super.init()  
    }  
}
```



属性观察器

- 父类的属性在它自己的初始化器中赋值不会触发属性观察器，但在子类的初始化器中赋值会触发属性观察器

```
class Person {  
    var age: Int {  
        willSet {  
            print("willSet", newValue)  
        }  
        didSet {  
            print("didSet", oldValue, age)  
        }  
    }  
    init() {  
        self.age = 0  
    }  
}
```

```
class Student : Person {  
    override init() {  
        super.init()  
        self.age = 1  
    }  
}  
  
// willSet 1  
// didSet 0 1  
var stu = Student()
```

可失败初始化器

- 类、结构体、枚举都可以使用 `init?` 定义可失败初始化器

```
class Person {  
    var name: String  
    init?(name: String) {  
        if name.isEmpty {  
            return nil  
        }  
        self.name = name  
    }  
}
```

- 之前接触过的可失败初始化器

```
var num = Int("123")  
public init?(_ description: String)
```

```
enum Answer : Int {  
    case wrong, right  
}  
var an = Answer(rawValue: 1)
```

- 不允许同时定义参数标签、参数个数、参数类型相同的可失败初始化器和非可失败初始化器
- 可以用 `init!` 定义隐式解包的可失败初始化器
- 可失败初始化器可以调用非可失败初始化器，非可失败初始化器调用可失败初始化器需要进行解包
- 如果初始化器调用一个可失败初始化器导致初始化失败，那么整个初始化过程都失败，并且之后的代码都停止执行
- 可以用一个非可失败初始化器重写一个可失败初始化器，但反过来是不行的



反初始化器 (deinit)

- **deinit**叫做反初始化器，类似于C++的析构函数、OC中的dealloc方法
- 当类的实例对象被释放内存时，就会调用实例对象的**deinit**方法

```
class Person {  
    deinit {  
        print("Person对象销毁了")  
    }  
}
```

- **deinit**不接受任何参数，不能写小括号，不能自行调用
- 父类的**deinit**能被子类继承
- 子类的**deinit**实现执行完毕后会调用父类的**deinit**

可选链

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



可选链 (Optional Chaining)

```
class Car { var price = 0 }
class Dog { var weight = 0 }
class Person {
    var name: String = ""
    var dog: Dog = Dog()
    var car: Car? = Car()
    func age() -> Int { 18 }
    func eat() { print("Person eat") }
    subscript(index: Int) -> Int { index }
}
```

```
var person: Person? = Person()
var age1 = person!.age() // Int
var age2 = person?.age() // Int?
var name = person?.name // String?
var index = person?[6] // Int?
```

```
func getName() -> String { "jack" }
// 如果person是nil，不会调用getName()
person?.name = getName()
```

- 如果可选项为`nil`，调用方法、下标、属性失败，结果为`nil`
- 如果可选项不为`nil`，调用方法、下标、属性成功，结果会被包装成可选项
- 如果结果本来就是可选项，不会进行再次包装

```
if let _ = person?.eat() { // ()?
    print("eat调用成功")
} else {
    print("eat调用失败")
}
```

```
var dog = person?.dog // Dog?
var weight = person?.dog.weight // Int?
var price = person?.car?.price // Int?
```

- 多个`?`可以链接在一起
- 如果链中任何一个节点是`nil`，那么整个链就会调用失败

可选链

```
var scores = ["Jack": [86, 82, 84], "Rose": [79, 94, 81]]  
scores["Jack"]?[0] = 100  
scores["Rose"]?[2] += 10  
scores["Kate"]?[0] = 88
```

```
var num1: Int? = 5  
num1? = 10 // Optional(10)  
  
var num2: Int? = nil  
num2? = 10 // nil
```

```
var dict: [String : (Int, Int) -> Int] = [  
    "sum" : (+),  
    "difference" : (-)  
]  
var result = dict["sum"]?(10, 20) // Optional(30), Int?
```

协议

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





协议 (Protocol)

- 协议可以用来定义方法、属性、下标的声明，协议可以被枚举、结构体、类遵守（多个协议之间用逗号隔开）

```
protocol Drawable {  
    func draw()  
    var x: Int { get set }  
    var y: Int { get }  
    subscript(index: Int) -> Int { get set }  
}
```

```
protocol Test1 {}  
protocol Test2 {}  
protocol Test3 {}  
class TestClass : Test1, Test2, Test3 {}
```

- 协议中定义方法时不能有默认参数值
- 默认情况下，协议中定义的内容必须全部都实现
- 也有办法办到只实现部分内容，以后的课程会讲到



协议中的属性

```
protocol Drawable {  
    func draw()  
    var x: Int { get set }  
    var y: Int { get }  
    subscript(index: Int) -> Int { get set }  
}
```

- 协议中定义属性时必须用**var**关键字
- 实现协议时的属性权限要不小于协议中定义的属性权限
- 协议定义**get**、**set**，用**var**存储属性或**get**、**set**计算属性去实现
- 协议定义**get**，用任何属性都可以实现

```
class Person : Drawable {  
    var x: Int = 0  
    let y: Int = 0  
    func draw() {  
        print("Person draw")  
    }  
    subscript(index: Int) -> Int {  
        set {}  
        get { index }  
    }  
}
```

```
class Person : Drawable {  
    var x: Int {  
        get { 0 }  
        set {}  
    }  
    var y: Int { 0 }  
    func draw() { print("Person draw") }  
    subscript(index: Int) -> Int {  
        set {}  
        get { index }  
    }  
}
```



static、class

- 为了保证通用，协议中必须用 `static` 定义类型方法、类型属性、类型下标

```
protocol Drawable {  
    static func draw()  
}  
  
class Person1 : Drawable {  
    class func draw() {  
        print("Person1 draw")  
    }  
}  
  
class Person2 : Drawable {  
    static func draw() {  
        print("Person2 draw")  
    }  
}
```



mutating

- 只有将协议中的实例方法标记为mutating
- 才允许结构体、枚举的具体实现修改自身内存
- 类在实现方法时不用加mutating，枚举、结构体才需要加mutating

```
protocol Drawable {
    mutating func draw()
}

class Size : Drawable {
    var width: Int = 0
    func draw() {
        width = 10
    }
}

struct Point : Drawable {
    var x: Int = 0
    mutating func draw() {
        x = 10
    }
}
```

init

- 协议中还可以定义初始化器 `init`
- 非 `final` 类实现时必须加上 `required`

```
protocol Drawable {  
    init(x: Int, y: Int)  
}  
  
class Point : Drawable {  
    required init(x: Int, y: Int) {}  
}  
  
final class Size : Drawable {  
    init(x: Int, y: Int) {}  
}
```

- 如果从协议实现的初始化器，刚好是重写了父类的指定初始化器
- 那么这个初始化必须同时加 `required`、`override`

```
protocol Livable {  
    init(age: Int)  
}  
  
class Person {  
    init(age: Int) {}  
}  
  
class Student : Person, Livable {  
    required override init(age: Int) {  
        super.init(age: age)  
    }  
}
```



init、init?、init!

- 协议中定义的 init?、init!，可以用 init、init?、init! 去实现
- 协议中定义的 init，可以用 init、init! 去实现

```
protocol Livable {  
    init()  
    init?(age: Int)  
    init!(no: Int)  
}
```

```
class Person : Livable {  
    required init() {}  
    // required init!() {}  
  
    required init?(age: Int) {}  
    // required init!(age: Int) {}  
    // required init(age: Int) {}  
  
    required init!(no: Int) {}  
    // required init?(no: Int) {}  
    // required init(no: Int) {}  
}
```



协议的继承

- 一个协议可以继承其他协议

```
protocol Runnable {  
    func run()  
}  
  
protocol Livable : Runnable {  
    func breath()  
}  
  
class Person : Livable {  
    func breath() {}  
    func run() {}  
}
```



协议组合

```
protocol Livable {}  
protocol Runnable {}  
class Person {}
```

■ 协议组合，可以包含1个类类型（最多1个）

```
// 接收Person或者其子类的实例  
func fn0(obj: Person) {}  
// 接收遵守Livable协议的实例  
func fn1(obj: Livable) {}  
// 接收同时遵守Livable、Runnable协议的实例  
func fn2(obj: Livable & Runnable) {}  
// 接收同时遵守Livable、Runnable协议、并且是Person或者其子类的实例  
func fn3(obj: Person & Livable & Runnable) {}
```

```
typealias RealPerson = Person & Livable & Runnable  
// 接收同时遵守Livable、Runnable协议、并且是Person或者其子类的实例  
func fn4(obj: RealPerson) {}
```



CaseIterable

- 让枚举遵守 CaseIterable 协议，可以实现遍历枚举值

```
enum Season : CaseIterable {
    case spring, summer, autumn, winter
}
let seasons = Season.allCases
print(seasons.count) // 4
for season in seasons {
    print(season)
} // spring summer autumn winter
```



CustomStringConvertible

- 遵守CustomStringConvertible、CustomDebugStringConvertible协议，都可以自定义实例的打印字符串

```
class Person : CustomStringConvertible, CustomDebugStringConvertible {
    var age = 0
    var description: String { "person_\(\age)" }
    var debugDescription: String { "debug_person_\(\age)" }
}
var person = Person()
print(person) // person_0
debugPrint(person) // debug_person_0
```

- print调用的是CustomStringConvertible协议的description
- debugPrint、po调用的是CustomDebugStringConvertible协议的debugDescription

The screenshot shows an Xcode interface with the following details:

- Code editor:
 - Line 7: `var person = Person()`
 - Line 8: `print(person) // person_0` (highlighted in green)
 - Line 9: `debugPrint(person) // debug_person_0`
- Toolbar: Shows various Xcode icons for navigation and selection.
- Bottom bar:
 - (lldb) `po person`
 - Output: `debug_person_0`

Any、AnyObject

■ Swift提供了2种特殊的类型：Any、 AnyObject

□ Any：可以代表任意类型（枚举、结构体、类，也包括函数类型）

□ AnyObject：可以代表任意类类型（在协议后面写上： AnyObject代表只有类能遵守这个协议）

✓ 在协议后面写上： class也代表只有类能遵守这个协议

```
var stu: Any = 10
stu = "Jack"
stu = Student()
```

```
// 创建1个能存放任意类型的数组
// var data = Array<Any>()
var data = [Any]()
data.append(1)
data.append(3.14)
data.append(Student())
data.append("Jack")
data.append({ 10 })
```



is、as?、as!、as

■ **is**用来判断是否为某种类型，**as**用来做强制类型转换

```
protocol Runnable { func run() }
```

```
class Person {}
```

```
class Student : Person, Runnable {
```

```
    func run() {
```

```
        print("Student run")
```

```
    }
```

```
    func study() {
```

```
        print("Student study")
```

```
    }
```

```
}
```

```
var stu: Any = 10
```

```
print(stu is Int) // true
```

```
stu = "Jack"
```

```
print(stu is String) // true
```

```
stu = Student()
```

```
print(stu is Person) // true
```

```
print(stu is Student) // true
```

```
print(stu is Runnable) // true
```

```
var stu: Any = 10
```

```
(stu as? Student)?.study() // 没有调用study
```

```
stu = Student()
```

```
(stu as? Student)?.study() // Student study
```

```
(stu as! Student).study() // Student study
```

```
(stu as? Runnable)?.run() // Student run
```

```
var data = [Any]()
```

```
data.append(Int("123") as Any)
```

```
var d = 10 as Double
```

```
print(d) // 10.0
```



X.self、X.Type、AnyClass

- X.self是一个元类型 (metadata) 的指针 , metadata存放着类型相关信息
- X.self属于X.Type类型

```
class Person {}
class Student : Person {}
var perType: Person.Type = Person.self
var stuType: Student.Type = Student.self
perType = Student.self
```

```
var anyType: AnyObject.Type = Person.self
anyType = Student.self

public typealias AnyClass = AnyObject.Type
var anyType2: AnyClass = Person.self
anyType2 = Student.self
```

```
var per = Person()
var perType = type(of: per) // Person.self
print(Person.self == type(of: per)) // true
```



元类型的应用

```
class Animal { required init() {} }

class Cat : Animal {}
class Dog : Animal {}
class Pig : Animal {}

func create(_ clses: [Animal.Type]) -> [Animal] {
    var arr = [Animal]()
    for cls in clses {
        arr.append(cls.init())
    }
    return arr
}

print(create([Cat.self, Dog.self, Pig.self]))
```



元类型的应用

```
import Foundation
class Person {
    var age: Int = 0
}
class Student : Person {
    var no: Int = 0
}
print(class_getInstanceSize(Student.self)) // 32
print(class_getSuperclass(Student.self)!) // Person
print(class_getSuperclass(Person.self)!) // Swift._SwiftObject
```

- 从结果可以看得出来，Swift还有个隐藏的基类：**Swift._SwiftObject**
- 可以参考Swift源码：<https://github.com/apple/swift/blob/master/stdlib/public/runtime/SwiftObject.h>

Self

■ Self代表当前类型

```
class Person {  
    var age = 1  
    static var count = 2  
    func run() {  
        print(self.age) // 1  
        print(Self.count) // 2  
    }  
}
```

■ Self一般用作返回值类型，限定返回值跟方法调用者必须是同一类型（也可以作为参数类型）

```
protocol Runnable {  
    func test() -> Self  
}  
class Person : Runnable {  
    required init() {}  
    func test() -> Self { type(of: self).init() }  
}  
class Student : Person {}
```

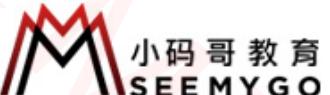
```
var p = Person()  
// Person  
print(p.test())  
  
var stu = Student()  
// Student  
print(stu.test())
```

错误处理

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





错误类型

- 开发过程常见的错误
- 语法错误（编译报错）
- 逻辑错误
- 运行时错误（可能会导致闪退，一般也叫做异常）
-



自定义错误

- Swift中可以通过Error协议自定义运行时的错误信息

```
enum SomeError : Error {  
    case illegalArg(String)  
    case outOfBounds(Int, Int)  
    case outOfMemory  
}
```

- 函数内部通过throw抛出自定义Error，可能会抛出Error的函数必须加上throws声明

```
func divide(_ num1: Int, _ num2: Int) throws -> Int {  
    if num2 == 0 {  
        throw SomeError.illegalArg("0不能作为除数")  
    }  
    return num1 / num2  
}
```

- 需要使用try调用可能会抛出Error的函数

```
var result = try divide(20, 10)
```



do-catch

- 可以使用do-catch捕捉Error

```
func test() {  
    print("1")  
    do {  
        print("2")  
        print(try divide(20, 0))  
        print("3")  
    } catch let SomeError.integerDivideByZero(msg) {  
        print("参数异常:", msg)  
    } catch let SomeError.outOfBounds(size, index) {  
        print("下标越界:", "size=\(size)", "index=\(index)")  
    } catch SomeError.outOfMemory {  
        print("内存溢出")  
    } catch {  
        print("其他错误")  
    }  
    print("4")  
}
```

```
test()  
// 1  
// 2  
// 参数异常： 0不能作为除数  
// 4
```

```
do {  
    try divide(20, 0)  
} catch let error {  
    switch error {  
    case let SomeError.integerDivideByZero(msg):  
        print("参数错误: ", msg)  
    default:  
        print("其他错误")  
    }  
}
```

- 抛出Error后，try下一句直到作用域结束的代码都将停止运行



处理Error

■ 处理Error的2种方式

① 通过do-catch捕捉Error

② 不捕捉Error，在当前函数增加throws声明，Error将自动抛给上层函数

✓ 如果最顶层函数（main函数）依然没有捕捉Error，那么程序将终止

```
func test() throws {
    print("1")
    print(try divide(20, 0))
    print("2")
}
try test()
// 1
// Fatal error: Error raised at top level
```

```
do {
    print(try divide(20, 0))
} catch is SomeError {
    print("SomeError")
}
```

```
func test() throws {
    print("1")
    do {
        print("2")
        print(try divide(20, 0))
        print("3")
    } catch let error as SomeError {
        print(error)
    }
    print("4")
}
try test()
// 1
// 2
// illegalArg("0不能作为除数")
// 4
```

try?、try!

- 可以使用try?、try!调用可能会抛出Error的函数，这样就不用去处理Error

```
func test() {  
    print("1")  
    var result1 = try? divide(20, 10) // Optional(2), Int?  
    var result2 = try? divide(20, 0) // nil  
    var result3 = try! divide(20, 10) // 2, Int  
    print("2")  
}  
test()
```

- a、b是等价的

```
var a = try? divide(20, 0)  
var b: Int?  
do {  
    b = try divide(20, 0)  
} catch { b = nil }
```



rethrows

- **rethrows**表明：函数本身不会抛出错误，但调用闭包参数抛出错误，那么它会将错误向上抛

```
func exec(_ fn: (Int, Int) throws -> Int, _ num1: Int, _ num2: Int) rethrows {  
    print(try fn(num1, num2))  
}  
// Fatal error: Error raised at top level  
try exec(divide, 20, 0)
```

defer

■ defer语句：用来定义以任何方式（抛错误、return等）离开代码块前必须要执行的代码

□ defer语句将延迟至当前作用域结束之前执行

```
func open(_ filename: String) -> Int {  
    print("open")  
    return 0  
}  
  
func close(_ file: Int) {  
    print("close")  
}
```

■ defer语句的执行顺序与定义顺序相反

```
func fn1() { print("fn1") }  
func fn2() { print("fn2") }  
  
func test() {  
    defer { fn1() }  
    defer { fn2() }  
}  
  
test()  
// fn2  
// fn1
```

```
func processFile(_ filename: String) throws {  
    let file = open(filename)  
    defer {  
        close(file)  
    }  
    // 使用file  
    // ....  
    try divide(20, 0)  
  
    // close将会在这里调用  
}  
try processFile("test.txt")  
// open  
// close  
// Fatal error: Error raised at top level
```

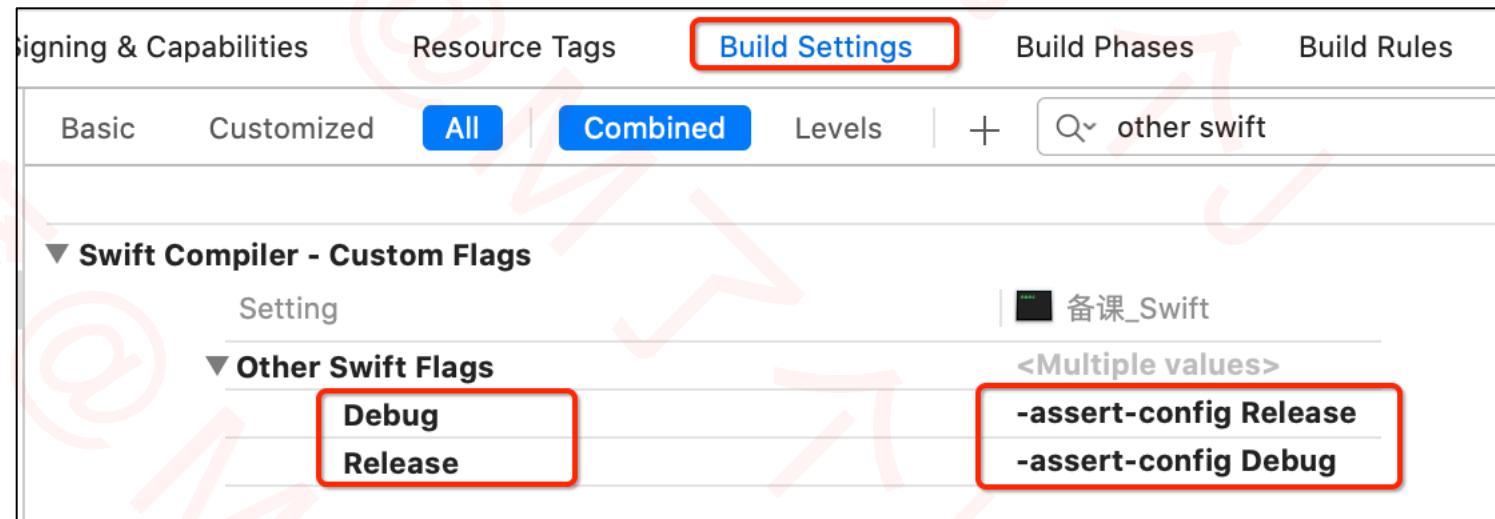
assert (断言)

- 很多编程语言都有断言机制：不符合指定条件就抛出运行时错误，常用于调试（Debug）阶段的条件判断
- 默认情况下，Swift的断言只会在Debug模式下生效，Release模式下会忽略

```
func divide(_ v1: Int, _ v2: Int) -> Int {  
    assert(v2 != 0, "除数不能为0")  
    return v1 / v2  
}  
print(divide(20, 0))
```

- 增加**Swift Flags**修改断言的默认行为

- assert-config Release**：强制关闭断言
- assert-config Debug**：强制开启断言





fatalError

- 如果遇到严重问题，希望结束程序运行时，可以直接使用`fatalError`函数抛出错误（这是无法通过`do-catch`捕捉的错误）
- 使用了`fatalError`函数，就不需要再写`return`

```
func test(_ num: Int) -> Int {  
    if num >= 0 {  
        return 1  
    }  
    fatalError("num不能小于0")  
}
```

- 在某些不得不实现、但不希望别人调用的方法，可以考虑内部使用`fatalError`函数

```
class Person { required init() {} }  
class Student : Person {  
    required init() { fatalError("don't call Student.init") }  
    init(score: Int) {}  
}  
var stu1 = Student(score: 98)  
var stu2 = Student()
```



局部作用域

- 可以使用 do 实现局部作用域

```
do {  
    let dog1 = Dog()  
    dog1.age = 10  
    dog1.run()  
}  
  
do {  
    let dog2 = Dog()  
    dog2.age = 10  
    dog2.run()  
}
```

泛型

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





泛型 (Generics)

- 泛型可以将类型参数化，提高代码复用率，减少代码量

```
func swapValues<T>(_ a: inout T, _ b: inout T) {  
    (a, b) = (b, a)  
}
```

```
var i1 = 10  
var i2 = 20  
swapValues(&i1, &i2)  
  
var d1 = 10.0  
var d2 = 20.0  
swapValues(&d1, &d2)  
  
struct Date {  
    var year = 0, month = 0, day = 0  
}  
var dd1 = Date(year: 2011, month: 9, day: 10)  
var dd2 = Date(year: 2012, month: 10, day: 11)  
swapValues(&dd1, &dd2)
```

- 泛型函数赋值给变量

```
func test<T1, T2>(_ t1: T1, _ t2: T2) {}  
var fn: (Int, Double) -> () = test
```



泛型

```
class Stack<E> {  
    var elements = [E]()  
    func push(_ element: E) { elements.append(element) }  
    func pop() -> E { elements.removeLast() }  
    func top() -> E { elements.last! }  
    func size() -> Int { elements.count }  
}
```

```
class SubStack<E> : Stack<E> {}
```

```
struct Stack<E> {  
    var elements = [E]()  
    mutating func push(_ element: E) { elements.append(element) }  
    mutating func pop() -> E { elements.removeLast() }  
    func top() -> E { elements.last! }  
    func size() -> Int { elements.count }  
}
```

```
var stack = Stack<Int>()  
stack.push(11)  
stack.push(22)  
stack.push(33)  
print(stack.top()) // 33  
print(stack.pop()) // 33  
print(stack.pop()) // 22  
print(stack.pop()) // 11  
print(stack.size()) // 0
```

```
enum Score<T> {  
    case point(T)  
    case grade(String)  
}  
let score0 = Score<Int>.point(100)  
let score1 = Score.point(99)  
let score2 = Score.point(99.5)  
let score3 = Score<Int>.grade("A")
```

关联类型 (Associated Type)

■ 关联类型的作用：给协议中用到的类型定义一个占位名称

■ 协议中可以拥有多个关联类型

```
protocol Stackable {  
    associatedtype Element // 关联类型  
    mutating func push(_ element: Element)  
    mutating func pop() -> Element  
    func top() -> Element  
    func size() -> Int  
}
```

```
class Stack<E> : Stackable {  
    // typealias Element = E  
    var elements = [E]()  
    func push(_ element: E) {  
        elements.append(element)  
    }  
    func pop() -> E { elements.removeLast() }  
    func top() -> E { elements.last! }  
    func size() -> Int { elements.count }  
}
```

```
class StringStack : Stackable {  
    // 给关联类型设定真实类型  
    // typealias Element = String  
    var elements = [String]()  
    func push(_ element: String) { elements.append(element) }  
    func pop() -> String { elements.removeLast() }  
    func top() -> String { elements.last! }  
    func size() -> Int { elements.count }  
}  
var ss = StringStack()  
ss.push("Jack")  
ss.push("Rose")
```



类型约束

```
protocol Runnable { }
class Person { }
func swapValues<T : Person & Runnable>(_ a: inout T, _ b: inout T) {
    (a, b) = (b, a)
}
```

```
protocol Stackable {
    associatedtype Element: Equatable
}
class Stack<E : Equatable> : Stackable { typealias Element = E }
```

```
func equal<S1: Stackable, S2: Stackable>(_ s1: S1, _ s2: S2) -> Bool
    where S1.Element == S2.Element, S1.Element : Hashable {
    return false
}
```

```
var stack1 = Stack<Int>()
var stack2 = Stack<String>()
// error: requires the types 'Int' and 'String' be equivalent
equal(stack1, stack2)
```

协议类型的注意点

```
protocol Runnable {}  
class Person : Runnable {}  
class Car : Runnable {}  
  
func get(_ type: Int) -> Runnable {  
    if type == 0 {  
        return Person()  
    }  
    return Car()  
}  
  
var r1 = get(0)  
var r2 = get(1)
```

■ 如果协议中有associatedtype

```
protocol Runnable {  
    associatedtype Speed  
    var speed: Speed { get }  
}  
class Person : Runnable {  
    var speed: Double { 0.0 }  
}  
class Car : Runnable {  
    var speed: Int { 0 }  
}
```

```
func get(_ type: Int) -> Runnable {  
    if type == 0 {  
        return Person()  
    }  
    return Car()  
}
```

! Protocol 'Runnable' can only be used as a generic constraint because it has Self or associated type requirements

! Protocol 'Runnable' can only be used as a generic constraint because it has Self or associated type requirements

```
func get(_ run: Runnable) {  
}
```

! Protocol 'Runnable' can only be used as a generic constraint because it has Self or associated type requirements



泛型解决

■ 解决方案①：使用泛型

```
func get<T : Runnable>(_ type: Int) -> T {  
    if type == 0 {  
        return Person() as! T  
    }  
    return Car() as! T  
}  
var r1: Person = get(0)  
var r2: Car = get(1)
```

不透明类型 (Opaque Type)

- 解决方案②：使用some关键字声明一个不透明类型

```
func get(_ type: Int) -> some Runnable { Car() }
var r1 = get(0)
var r2 = get(1)
```

- some限制只能返回一种类型

```
func get(_ type: Int) -> some Runnable {
    if type == 0 {
        return Person()
    }
    return Car()
}
```

2 ! Function declares an opaque



some

- some除了用在返回值类型上，一般还可以用在属性类型上

```
protocol Runnable { associatedtype Speed }
class Dog : Runnable { typealias Speed = Double }
class Person {
    var pet: some Runnable {
        return Dog()
    }
}
```



可选项的本质

■ 可选项的本质是 enum 类型

```
public enum Optional<Wrapped> : ExpressibleByNilLiteral {  
    case none  
    case some(Wrapped)  
    public init(_ some: Wrapped)  
}
```

```
var age: Int? = .none  
age = 10  
age = .some(20)  
age = nil
```

```
var age: Int? = 10  
var age0: Optional<Int> = Optional<Int>.some(10)  
var age1: Optional = .some(10)  
var age2 = Optional.some(10)  
var age3 = Optional(10)  
age = nil  
age3 = .none
```

```
switch age {  
case let v?:  
    print("some", v)  
case nil:  
    print("none")  
  
}  
  
switch age {  
case let .some(v):  
    print("some", v)  
case .none:  
    print("none")  
}
```

```
var age: Int? = nil  
var age0 = Optional<Int>.none  
var age1: Optional<Int> = .none
```



可选项的本质

```
var age_: Int? = 10
var age: Int?? = age_
age = nil

var age0 = Optional.some(Optional.some(10))
age0 = .none
var age1: Optional<Optional> = .some(.some(10))
age1 = .none
```

```
var age: Int?? = 10
var age0: Optional<Optional> = 10
```

String与Array

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



关于String的思考

- 1个String变量占用多少内存？
- 下面2个String变量，底层存储有什么不同？

```
var str1 = "0123456789"  
var str2 = "0123456789ABCDEF"
```

- 如果对String进行拼接操作，String变量的存储会发生什么变化？

```
str1.append("ABCDE")  
str1.append("F")  
  
str2.append("G")
```

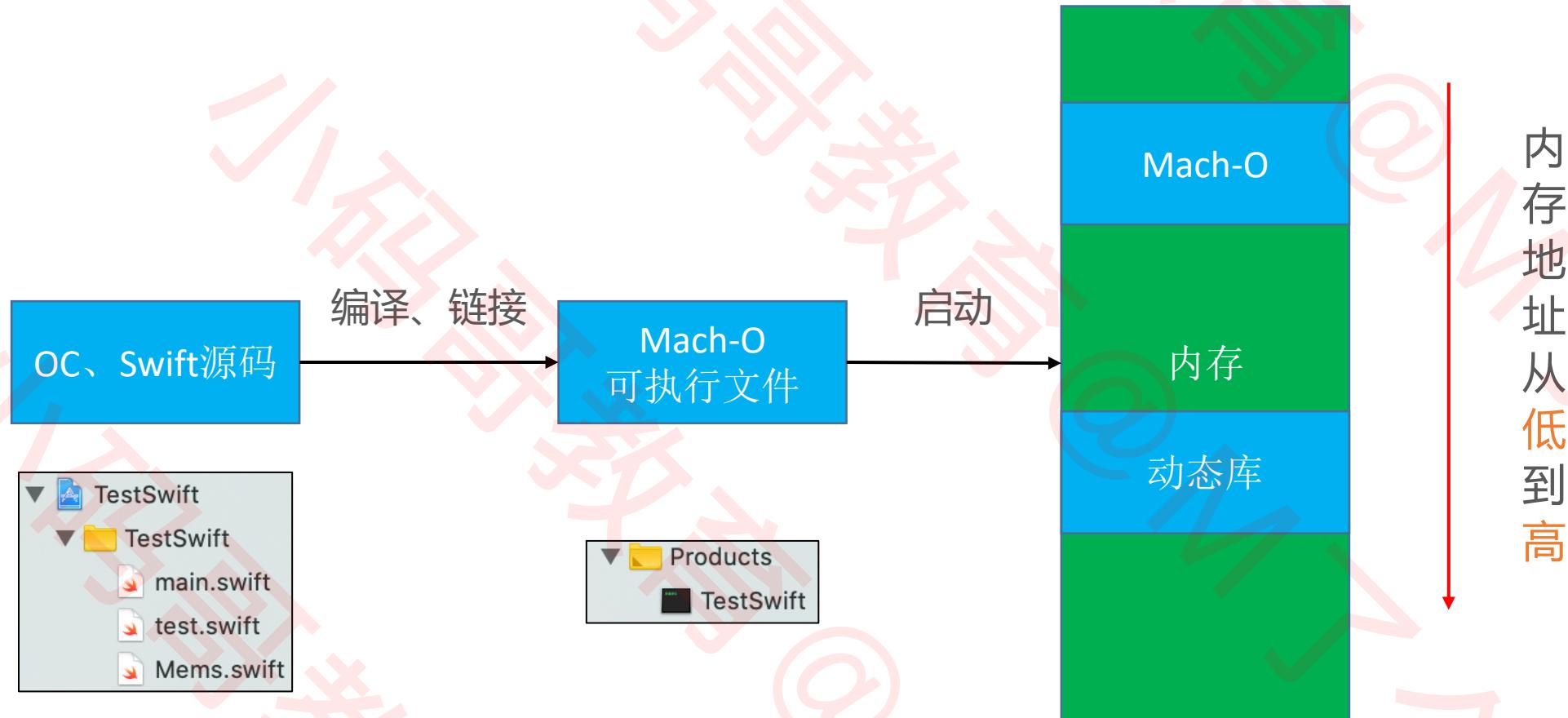
- ASCII码表：<https://www.ascii-code.com/>

内存地址从低到高





从编码到启动APP



dyld_stub_binder

■ 符号的延迟绑定通过dyld_stub_binder完成

■ jmpq *0xb31(%rip)格式的汇编指令

□ 占用6个字节

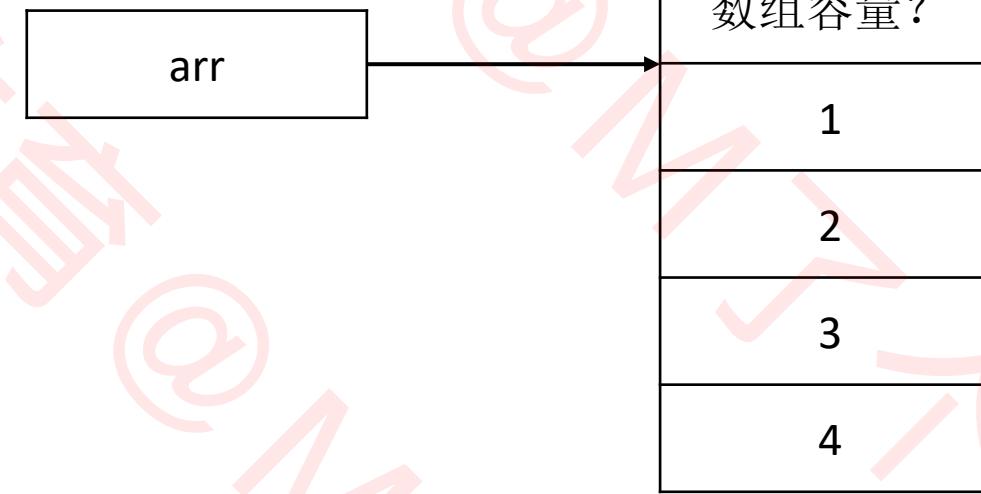
	Offset	Data	Description	Value
▼ Executable (X86_64)				
*Mach64 Header	00006000	0000000100004676	Indirect Pointer	[0x100006000->__\$SKsE4last7ElementQzSg
► Load Commands	00006008	00000001000047AC	Indirect Pointer	[0x100006008->__\$SS10FoundationE6forma
► Section64 (_TEXT,_text)	00006010	0000000100004680	Indirect Pointer	[0x100006010->__\$SS19stringInterpolati
► Section64 (_TEXT,_stubs)	00006018	000000010000468A	Indirect Pointer	[0x100006018->__\$SS21_builtinStringLit
► Section64 (_TEXT,_stub_helper)	00006020	0000000100004694	Indirect Pointer	[0x100006020->__\$SS6appendyySSF]
► Section64 (_TEXT,_cstring)	00006028	000000010000469E	Indirect Pointer	[0x100006028->__\$SYsSHRzSH8RawValueSYR
► Section64 (_TEXT,_swift5_typeref)	00006030	00000001000046A8	Indirect Pointer	[0x100006030->__\$SYsSHRzSH8RawValueSYR
► Section64 (_TEXT,_const)	00006038	00000001000046B2	Indirect Pointer	[0x100006038->__\$SYsSHRzSH8RawValueSYR
► Section64 (_TEXT,_swift5_reflstr)	00006040	000000010000463A	Indirect Pointer	[0x100006040->__\$Sa12arrayLiteralSayG
► Section64 (_TEXT,_swift5_fieldmd)	00006048	0000000100004644	Indirect Pointer	[0x100006048->__\$Sa6appendyyxnFs5UInt8
► Section64 (_TEXT,_swift5_assocty)	00006050	00000001000046BC	Indirect Pointer	[0x100006050->__\$SaMa]
► Section64 (_TEXT,_swift5_proto)	00006058	000000010000464E	Indirect Pointer	[0x100006058->__\$SayxSicigs5UInt8V_Tg5
► Section64 (_TEXT,_swift5_types)	00006060	0000000100004658	Indirect Pointer	[0x100006060->__\$Slss16IndexingIterato
► Section64 (_TEXT,_ unwind_info)	00006068	0000000100004662	Indirect Pointer	[0x100006068->__\$Sn15uncheckedBoundsSn
► Section64 (_TEXT,_eh_frame)	00006070	000000010000466C	Indirect Pointer	[0x100006070->__\$s16IndexingIteratorV4
► Section64 (_DATA_CONST,_got)	00006078	00000001000046C6	Indirect Pointer	[0x100006078->__\$s17_assertionFailure
► Section64 (_DATA_CONST,_const)	00006080	00000001000046D0	Indirect Pointer	[0x100006080->__\$s17withUnsafePointer2
► Section64 (_DATA_CONST,_objc_imageinfo)	00006088	00000001000046DA	Indirect Pointer	[0x100006088->__\$s18_fatalErrorMessage
▼ Section64 (_DATA,_la_symbol_ptr)	00006090	00000001000046E4	Indirect Pointer	[0x100006090->__\$s26DefaultStringInter
*Lazy Symbol Pointers	00006098	00000001000046EE	Indirect Pointer	[0x100006098->__\$s26DefaultStringInter
► Section64 (_DATA,_data)	000060A0	00000001000046F8	Indirect Pointer	[0x1000060A0->__\$s26DefaultStringInter
► Dynamic Loader Info	000060A8	0000000100004702	Indirect Pointer	[0x1000060A8->__\$s27_allocateUninitial
► Function Starts	000060B0	000000010000470C	Indirect Pointer	[0x1000060B0->__\$s27_bridgeAnythingTo0
► Data in Code Entries	000060B8	0000000100004716	Indirect Pointer	[0x1000060B8->__\$s2eeoiySbx_xtSYRzS08R
► Symbol Table	000060C0	000000010000461C	Indirect Pointer	[0x1000060C0->_malloc_size]
► Dynamic Symbol Table	000060C8	0000000100004626	Indirect Pointer	[0x1000060C8->_memcpy]
► String Table	000060D0	0000000100004630	Indirect Pointer	[0x1000060D0->_memset]
► Code Signature	000060D8	0000000100004720	Indirect Pointer	[0x1000060D8->_swift_allocateGenericVa

	Offset	Data	Description	Value
▼ Executable (X86_64)				
*Mach64 Header	00004685	E982FFFFFF	jmp	0x10000460c
► Load Commands	0000468A	6893000000	push	0x93
► Section64 (_TEXT,_text)	0000468F	E978FFFFFF	jmp	0x10000460c
► Section64 (_TEXT,_stubs)	00004694	68E0000000	push	0xe0
▼ Section64 (_TEXT,_stub_helper)	00004699	E96EFFFFFF	jmp	0x10000460c
*Assembly	0000469E	68F8000000	push	0xf8
► Section64 (_TEXT,_cstring)	000046A3	E964FFFFFF	jmp	0x10000460c
► Section64 (_TEXT,_swift5_typeref)	000046A8	6828010000	push	0x128

关于Array的思考

```
public struct Array<Element>
var arr = [1, 2, 3, 4]
```

- 1个Array变量占用多少内存？
- 数组中的数据存放在哪里？



高级运算符

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松

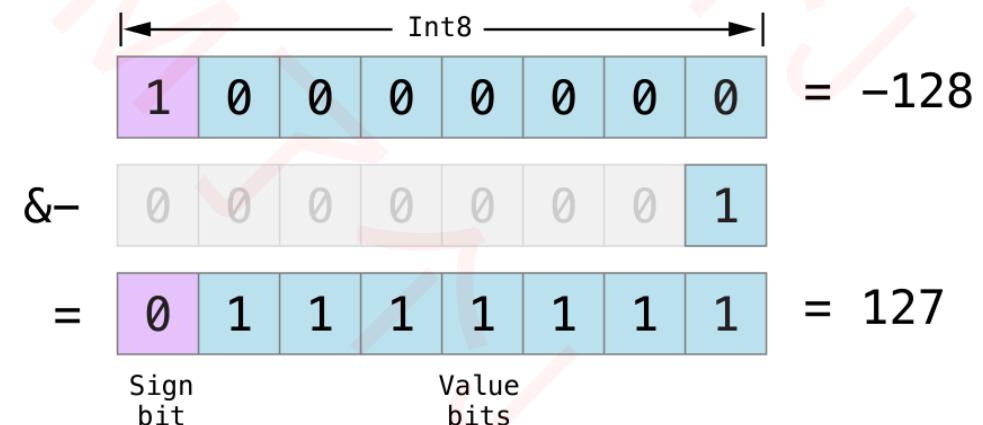
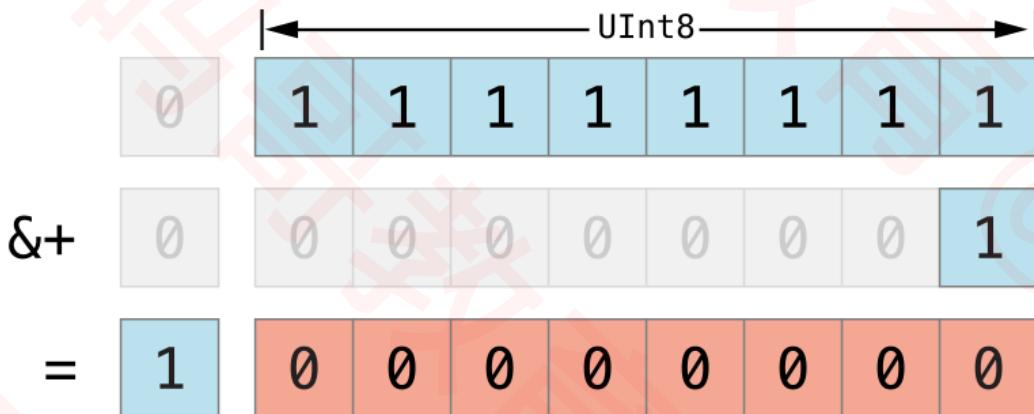
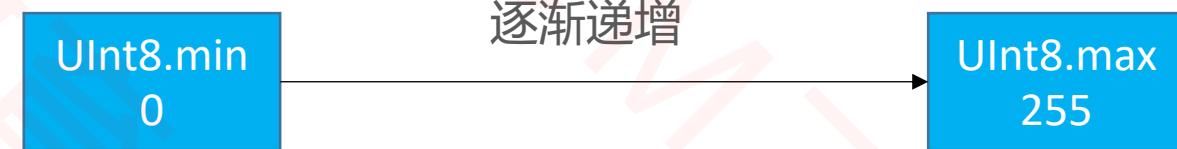


溢出运算符 (Overflow Operator)

- Swift的算数运算符出现溢出时会抛出运行时错误
- Swift有溢出运算符 (&+、&-、&*) , 用来支持溢出运算

```
var min = UInt8.min
print(min &- 1) // 255, Int8.max

var max = UInt8.max
print(max &+ 1) // 0, Int8.min
print(max &* 2) // 254, 等价于 max &+ max
```





运算符重载 (Operator Overload)

- 类、结构体、枚举可以为现有的运算符提供自定义的实现，这个操作叫做：运算符重载

```
struct Point {  
    var x: Int, y: Int  
}  
  
func + (p1: Point, p2: Point) -> Point {  
    Point(x: p1.x + p2.x, y: p1.y + p2.y)  
}  
  
let p = Point(x: 10, y: 20) + Point(x: 11, y: 22)  
print(p) // Point(x: 21, y: 42)
```

```
struct Point {  
    var x: Int, y: Int  
    static func + (p1: Point, p2: Point) -> Point {  
        Point(x: p1.x + p2.x, y: p1.y + p2.y)  
    }  
}
```



运算符重载

```
static func + (p1: Point, p2: Point) -> Point {  
    Point(x: p1.x + p2.x, y: p1.y + p2.y)  
}  
  
static func - (p1: Point, p2: Point) -> Point {  
    Point(x: p1.x - p2.x, y: p1.y - p2.y)  
}  
  
static prefix func - (p: Point) -> Point {  
    Point(x: -p.x, y: -p.y)  
}  
  
static func += (p1: inout Point, p2: Point) {  
    p1 = p1 + p2  
}
```

```
static prefix func ++ (p: inout Point) -> Point {  
    p += Point(x: 1, y: 1)  
    return p  
}  
  
static postfix func ++ (p: inout Point) -> Point {  
    let tmp = p  
    p += Point(x: 1, y: 1)  
    return tmp  
}  
  
static func == (p1: Point, p2: Point) -> Bool {  
    (p1.x == p2.x) && (p1.y == p2.y)  
}
```



Equatable

- 要想得知2个实例是否等价，一般做法是遵守 **Equatable** 协议，重载 == 运算符
- 与此同时，等价于重载了 != 运算符

```
struct Point : Equatable {  
    var x: Int, y: Int  
}  
  
var p1 = Point(x: 10, y: 20)  
var p2 = Point(x: 11, y: 22)  
print(p1 == p2) // false  
print(p1 != p2) // true
```

- Swift为以下类型提供默认的 **Equatable** 实现
- 没有关联类型的枚举
- 只拥有遵守 **Equatable** 协议关联类型的枚举
- 只拥有遵守 **Equatable** 协议存储属性的结构体

- 引用类型比较存储的地址值是否相等（是否引用着同一个对象），使用恒等运算符 === 、 !==

Comparable

```
// score大的比较大，若score相等，age小的比较大
struct Student : Comparable {
    var age: Int
    var score: Int
    init(score: Int, age: Int) {
        self.score = score
        self.age = age
    }
    static func < (lhs: Student, rhs: Student) -> Bool {
        (lhs.score < rhs.score)
        || (lhs.score == rhs.score && lhs.age > rhs.age)
    }
    static func > (lhs: Student, rhs: Student) -> Bool {
        (lhs.score > rhs.score)
        || (lhs.score == rhs.score && lhs.age < rhs.age)
    }
    static func <= (lhs: Student, rhs: Student) -> Bool {
        !(lhs > rhs)
    }
    static func >= (lhs: Student, rhs: Student) -> Bool {
        !(lhs < rhs)
    }
}
```

- 要想比较2个实例的大小，一般做法是：
 - 遵守 Comparable 协议
 - 重载相应的运算符

```
var stu1 = Student(score: 100, age: 20)
var stu2 = Student(score: 98, age: 18)
var stu3 = Student(score: 100, age: 20)
print(stu1 > stu2) // true
print(stu1 >= stu2) // true
print(stu1 >= stu3) // true
print(stu1 <= stu3) // true
print(stu2 < stu1) // true
print(stu2 <= stu1) // true
```



自定义运算符 (Custom Operator)

- 可以自定义新的运算符：在全局作用域使用 **operator** 进行声明

```
prefix operator 前缀运算符
```

```
postfix operator 后缀运算符
```

```
infix operator 中缀运算符 : 优先级组
```

```
precedencegroup 优先级组 {  
    associativity: 结合性(left\right\none)  
    higherThan: 比谁的优先级高  
    lowerThan: 比谁的优先级低  
    assignment: true 代表在可选链操作中拥有跟赋值运算符一样的优先级  
}
```

```
prefix operator +++  
infix operator +- : PlusMinusPrecedence  
precedencegroup PlusMinusPrecedence {  
    associativity: none  
    higherThan: AdditionPrecedence  
    lowerThan: MultiplicationPrecedence  
    assignment: true  
}
```

- Apple 文档参考：

- https://developer.apple.com/documentation/swift/swift_standard_library/operator_declarations
- <https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID380>



自定义运算符

```
struct Point {  
    var x: Int, y: Int  
    static prefix func +++ (point: inout Point) -> Point {  
        point = Point(x: point.x + point.x, y: point.y + point.y)  
        return point  
    }  
    static func +- (left: Point, right: Point) -> Point {  
        return Point(x: left.x + right.x, y: left.y - right.y)  
    }  
    static func +- (left: Point?, right: Point) -> Point {  
        print("+-")  
        return Point(x: left?.x ?? 0 + right.x, y: left?.y ?? 0 - right.y)  
    }  
}
```

```
struct Person {  
    var point: Point  
}  
var person: Person? = nil  
person?.point +- Point(x: 10, y: 20)
```

扩展

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码马拉松





扩展 (Extension)

- Swift中的扩展，有点类似于OC中的分类 (Category)
- 扩展可以为枚举、结构体、类、协议添加新功能
 - 可以添加方法、计算属性、下标、（便捷）初始化器、嵌套类型、协议等等
- 扩展不能办到的事情
 - 不能覆盖原有的功能
 - 不能添加存储属性，不能向已有的属性添加属性观察器
 - 不能添加父类
 - 不能添加指定初始化器，不能添加反初始化器
 - ...



计算属性、下标、方法、嵌套类型

```
extension Double {  
    var km: Double { self * 1_000.0 }  
    var m: Double { self }  
    var dm: Double { self / 10.0 }  
    var cm: Double { self / 100.0 }  
    var mm: Double { self / 1_000.0 }  
}
```

```
extension Array {  
    subscript(nullable idx: Int) -> Element? {  
        if (startIndex..            return self[idx]  
        }  
        return nil  
    }  
}
```

```
extension Int {  
    func repetitions(task: () -> Void) {  
        for _ in 0..<self { task() }  
    }  
    mutating func square() -> Int {  
        self = self * self  
        return self  
    }  
    enum Kind { case negative, zero, positive }  
    var kind: Kind {  
        switch self {  
            case 0: return .zero  
            case let x where x > 0: return .positive  
            default: return .negative  
        }  
    }  
    subscript(digitIndex: Int) -> Int {  
        var decimalBase = 1  
        for _ in 0..<digitIndex { decimalBase *= 10 }  
        return (self / decimalBase) % 10  
    }  
}
```



协议、初始化器

```
class Person {  
    var age: Int  
    var name: String  
    init(age: Int, name: String) {  
        self.age = age  
        self.name = name  
    }  
}  
  
extension Person : Equatable {  
    static func == (left: Person, right: Person) -> Bool {  
        left.age == right.age && left.name == right.name  
    }  
    convenience init() {  
        self.init(age: 0, name: "")  
    }  
}
```

```
struct Point {  
    var x: Int = 0  
    var y: Int = 0  
}  
extension Point {  
    init(_ point: Point) {  
        self.init(x: point.x, y: point.y)  
    }  
}  
var p1 = Point()  
var p2 = Point(x: 10)  
var p3 = Point(y: 20)  
var p4 = Point(x: 10, y: 20)  
var p5 = Point(p4)
```

- 如果希望自定义初始化器的同时，编译器也能够生成默认初始化器
- 可以在扩展中编写自定义初始化器
- **required**初始化器也不能写在扩展中

协议

- 如果一个类型已经实现了协议的所有要求，但是还没有声明它遵守了这个协议
- 可以通过扩展来让它遵守这个协议

```
protocol TestProtocol {  
    func test()  
}  
  
class TestClass {  
    func test() {  
        print("test")  
    }  
}  
  
extension TestClass : TestProtocol {}
```

- 编写一个函数，判断一个整数是否为奇数？

```
func isOdd<T: BinaryInteger>(_ i: T) -> Bool {  
    i % 2 != 0  
}
```

```
extension BinaryInteger {  
    func isOdd() -> Bool { self % 2 != 0 }  
}
```

协议

- 扩展可以给协议提供默认实现，也间接实现『可选协议』的效果
- 扩展可以给协议扩充『协议中从未声明过的方法』

```
protocol TestProtocol {  
    func test1()  
}  
  
extension TestProtocol {  
    func test1() {  
        print("TestProtocol test1")  
    }  
    func test2() {  
        print("TestProtocol test2")  
    }  
}
```

```
class TestClass : TestProtocol {}  
var cls = TestClass()  
cls.test1() // TestProtocol test1  
cls.test2() // TestProtocol test2  
var cls2: TestProtocol = TestClass()  
cls2.test1() // TestProtocol test1  
cls2.test2() // TestProtocol test2
```

```
class TestClass : TestProtocol {  
    func test1() { print("TestClass test1") }  
    func test2() { print("TestClass test2") }  
}  
  
var cls = TestClass()  
cls.test1() // TestClass test1  
cls.test2() // TestClass test2  
var cls2: TestProtocol = TestClass()  
cls2.test1() // TestClass test1  
cls2.test2() // TestProtocol test2
```

泛型

```
class Stack<E> {
    var elements = [E]()
    func push(_ element: E) {
        elements.append(element)
    }
    func pop() -> E { elements.removeLast() }
    func size() -> Int { elements.count }
}

// 扩展中依然可以使用原类型中的泛型类型
extension Stack {
    func top() -> E { elements.last! }
}

// 符合条件才扩展
extension Stack : Equatable where E : Equatable {
    static func == (left: Stack, right: Stack) -> Bool {
        left.elements == right.elements
    }
}
```

访问控制

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





访问控制 (Access Control)

- 在访问权限控制这块，Swift提供了5个不同的访问级别（以下是从高到低排列，实体指被访问级别修饰的内容）
 - **open**：允许在定义实体的模块、其他模块中访问，允许其他模块进行继承、重写（**open**只能用在类、类成员上）
 - **public**：允许在定义实体的模块、其他模块中访问，不允许其他模块进行继承、重写
 - **internal**：只允许在定义实体的模块中访问，不允许在其他模块中访问
 - **fileprivate**：只允许在定义实体的源文件中访问
 - **private**：只允许在定义实体的封闭声明中访问
- 绝大部分实体默认都是 **internal** 级别



访问级别的使用准则

■ 一个实体不可以被更低访问级别的实体定义，比如

- 变量\常量类型 \geq 变量\常量
- 参数类型、返回值类型 \geq 函数
- 父类 \geq 子类
- 父协议 \geq 子协议
- 原类型 \geq typealias
- 原始值类型、关联值类型 \geq 枚举类型
- 定义类型A时用到的其他类型 \geq 类型A
-



元组类型

- 元组类型的访问级别是所有成员类型最低的那个

```
internal struct Dog {}  
fileprivate class Person {}  
  
// (Dog, Person) 的访问级别是 fileprivate  
fileprivate var data1: (Dog, Person)  
private var data2: (Dog, Person)
```



泛型类型

- 泛型类型的访问级别是 **类型的访问级别** 以及 **所有泛型类型参数的访问级别** 中最低的那个

```
internal class Car {}  
fileprivate class Dog {}  
public class Person<T1, T2> {}  
  
// Person<Car, Dog>的访问级别是fileprivate  
fileprivate var p = Person<Car, Dog>()
```

成员、嵌套类型

- 类型的访问级别会影响成员（属性、方法、初始化器、下标）、嵌套类型的默认访问级别
- 一般情况下，类型为 `private` 或 `fileprivate`，那么成员\嵌套类型默认也是 `private` 或 `fileprivate`
- 一般情况下，类型为 `internal` 或 `public`，那么成员\嵌套类型默认是 `internal`

```
public class PublicClass {  
    public var p1 = 0 // public  
    var p2 = 0 // internal  
    fileprivate func f1() {} // fileprivate  
    private func f2() {} // private  
}  
  
class InternalClass { // internal  
    var p = 0 // internal  
    fileprivate func f1() {} // fileprivate  
    private func f2() {} // private  
}
```

```
fileprivate class FilePrivateClass { // fileprivate  
    func f1() {} // fileprivate  
    private func f2() {} // private  
}  
  
private class PrivateClass { // private  
    func f() {} // private  
}
```

成员的重写

- 子类重写成员的访问级别必须 \geq 子类的访问级别，或者 \geq 父类被重写成员的访问级别
- 父类的成员不能被成员作用域外定义的子类重写

```
public class Person {  
    private var age: Int = 0  
}  
  
public class Student : Person {  
    override var age: Int {  
        set {}  
        get {10}  
    }  
}
```

```
public class Person {  
    private var age: Int = 0  
}  
  
public class Student : Person {  
    override var age: Int {  
        set {}  
        get {10}  
    }  
}
```



下面代码能否编译通过？

```
private class Person {}  
fileprivate class Student : Person {}
```

```
private struct Dog {  
    var age: Int = 0  
    func run() {}  
}
```

```
fileprivate struct Person {  
    var dog: Dog = Dog()  
    mutating func walk() {  
        dog.run()  
        dog.age = 1  
    }  
}
```

```
private struct Dog {  
    private var age: Int = 0  
    private func run() {}  
}
```

```
fileprivate struct Person {  
    var dog: Dog = Dog()  
    mutating func walk() {  
        dog.run()  
        dog.age = 1  
    }  
}
```

- 直接在全局作用域下定义的**private**等价于**fileprivate**



getter、setter

- getter、setter默认自动接收它们所属环境的访问级别
- 可以给setter单独设置一个比getter更低的访问级别，用以限制写的权限

```
fileprivate(set) public var num = 10
class Person {
    private(set) var age = 0
    fileprivate(set) public var weight: Int {
        set {}
        get { 10 }
    }
    internal(set) public subscript(index: Int) -> Int {
        set {}
        get { index }
    }
}
```



初始化器

- 如果一个public类想在另一个模块调用编译生成的默认无参初始化器，必须显式提供public的无参初始化器
 - 因为public类的默认初始化器是internal级别
- required初始化器 \geq 它的默认访问级别
- 如果结构体有private\fileprivate的存储实例属性，那么它的成员初始化器也是private\fileprivate
 - 否则默认就是internal



枚举类型的case

- 不能给enum的每个case单独设置访问级别
- 每个case自动接收enum的访问级别
- public enum定义的case也是public

协议

- 协议中定义的要求自动接收协议的访问级别，不能单独设置访问级别
- **public**协议定义的要求也是**public**
- 协议实现的访问级别必须 \geq 类型的访问级别，或者 \geq 协议的访问级别
- 下面代码能编译通过么？

```
public protocol Runnable {  
    func run()  
}  
  
public class Person : Runnable {  
    func run() {}  
}
```



扩展

- 如果有显式设置扩展的访问级别，扩展添加的成员自动接收扩展的访问级别
- 如果没有显式设置扩展的访问级别，扩展添加的成员的默认访问级别，跟直接在类型中定义的成员一样
- 可以单独给扩展添加的成员设置访问级别
- 不能给用于遵守协议的扩展显式设置扩展的访问级别

扩展

- 在同一文件中的扩展，可以写成类似多个部分的类型声明
- 在原本的声明中声明一个私有成员，可以在同一文件的扩展中访问它
- 在扩展中声明一个私有成员，可以在同一文件的其他扩展中、原本声明中访问它

```
public class Person {  
    private func run0() {}  
    private func eat0() {  
        run1()  
    }  
}
```

```
extension Person {  
    private func run1() {}  
    private func eat1() {  
        run0()  
    }  
}
```

```
extension Person {  
    private func eat2() {  
        run1()  
    }  
}
```



将方法赋值给var\let

- 方法也可以像函数那样，赋值给一个let或者var

```
struct Person {  
    var age: Int  
    func run(_ v: Int) { print("func run", age, v) }  
    static func run(_ v: Int) { print("static func run", v) }  
}
```

```
let fn1 = Person.run  
fn1(10) // static func run 10
```

```
let fn2: (Int) -> () = Person.run  
fn2(20) // static func run 20
```

```
let fn3: (Person) -> ((Int) -> ()) = Person.run  
fn3(Person(age: 18))(30) // func run 18 30
```

内存管理

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



内存管理

- 跟OC一样，Swift也是采取基于引用计数的ARC内存管理方案（针对堆空间）
- Swift的ARC中有3种引用
 - 强引用（ strong reference ）：默认情况下，引用都是强引用
 - 弱引用（ weak reference ）：通过`weak`定义弱引用
 - ✓ 必须是可选类型的`var`，因为实例销毁后，ARC会自动将弱引用设置为`nil`
 - ✓ ARC自动给弱引用设置`nil`时，不会触发属性观察器
 - 无主引用（ unowned reference ）：通过`unowned`定义无主引用
 - ✓ 不会产生强引用，实例销毁后仍然存储着实例的内存地址（类似于OC中的`unsafe_unretained`）
 - ✓ 试图在实例销毁后访问无主引用，会产生运行时错误（野指针）
 - **Fatal error: Attempted to read an unowned reference but object 0x0 was already deallocated**



weak、unowned的使用限制

- weak、unowned只能用在类实例上面

```
protocol Livable : AnyObject {}  
class Person {}  
  
weak var p0: Person?  
weak var p1: AnyObject?  
weak var p2: Livable?  
  
unowned var p10: Person?  
unowned var p11: AnyObject?  
unowned var p12: Livable?
```



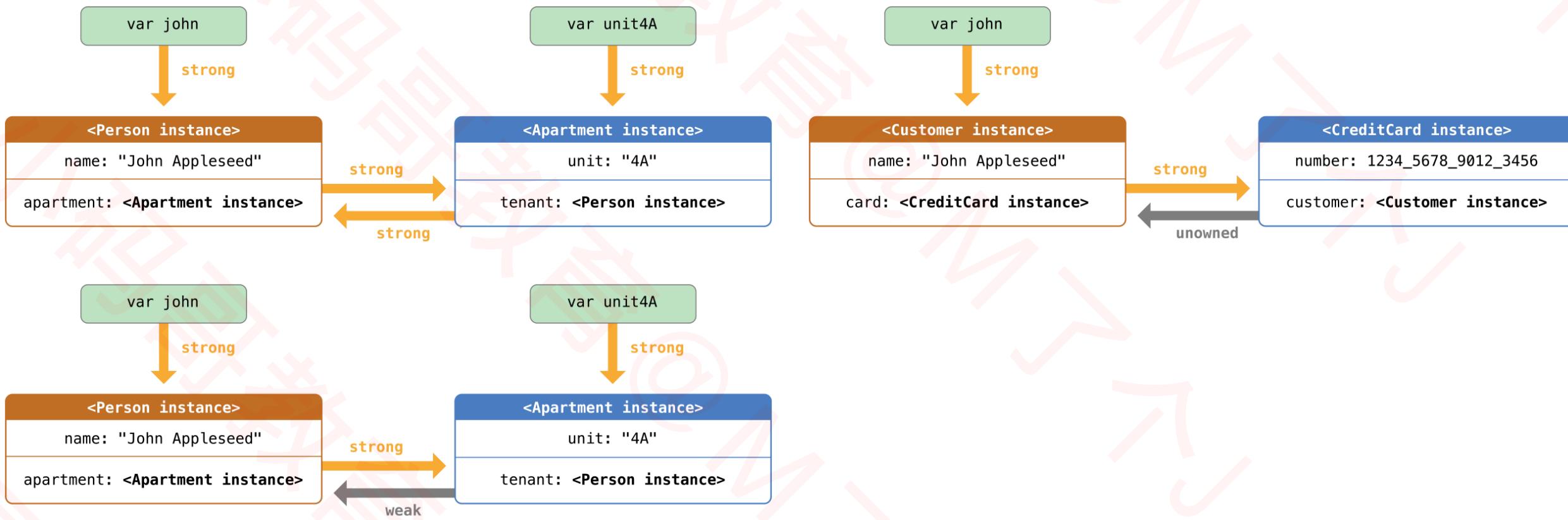
Autoreleasepool

```
public func autoreleasepool<Result>(invoking body: () throws -> Result) rethrows -> Result
```

```
autoreleasepool {
    let p = MJPerson(age: 20, name: "Jack")
    p.run()
}
```

循环引用 (Reference Cycle)

- weak、unowned 都能解决循环引用的问题，unowned 要比 weak 少一些性能消耗
- 在生命周期中可能会变为 nil 的使用 weak
- 初始化赋值后再也不会变为 nil 的使用 unowned



闭包的循环引用

- 闭包表达式默认会对用到的外层对象产生额外的强引用（对外层对象进行了retain操作）
- 下面代码会产生循环引用，导致Person对象无法释放（看不到Person的deinit被调用）

```
class Person {  
    var fn: (() -> ())?  
    func run() { print("run") }  
    deinit { print("deinit") }  
}  
func test() {  
    let p = Person()  
    p.fn = { p.run() }  
}  
test()
```

- 在闭包表达式的捕获列表声明weak或unowned引用，解决循环引用问题

```
p.fn = {  
    [weak p] in  
    p?.run()  
}
```

```
p.fn = {  
    [unowned p] in  
    p.run()  
}
```

```
p.fn = {  
    [weak wp = p, unowned up = p, a = 10 + 20] in  
    wp?.run()  
}
```



闭包的循环引用

- 如果想在定义闭包属性的同时引用 `self`，这个闭包必须是 `lazy` 的（因为在实例初始化完毕之后才能引用 `self`）

```
class Person {  
    lazy var fn: () -> () = {  
        [weak self] in  
        self?.run()  
    }  
    func run() { print("run") }  
    deinit { print("deinit") }  
}
```

- 左边的闭包 `fn` 内部如果用到了实例成员（属性、方法）
 - 编译器会强制要求明确写出 `self`

- 如果 `lazy` 属性是闭包调用的结果，那么不用考虑循环引用的问题（因为闭包调用后，闭包的生命周期就结束了）

```
class Person {  
    var age: Int = 0  
    lazy var getAge: Int = {  
        self.age  
    }()  
    deinit { print("deinit") }  
}
```



@escaping

- 非逃逸闭包、逃逸闭包，一般都是当做参数传递给函数
- 非逃逸闭包：闭包调用发生在函数结束前，闭包调用在函数作用域内
- 逃逸闭包：闭包有可能在函数结束后调用，闭包调用逃离了函数的作用域，需要通过`@escaping`声明

```
import Dispatch  
typealias Fn = () -> ()
```

```
// fn是非逃逸闭包  
func test1(_ fn: Fn) { fn() }
```

```
// fn是逃逸闭包  
var gFn: Fn?  
func test2(_ fn: @escaping Fn) { gFn = fn }
```

```
// fn是逃逸闭包  
func test3(_ fn: @escaping Fn) {  
    DispatchQueue.global().async {  
        fn()  
    }  
}
```

```
class Person {  
    var fn: Fn  
    // fn是逃逸闭包  
    init(fn: @escaping Fn) {  
        self.fn = fn  
    }  
    func run() {  
        // DispatchQueue.global().async也是一个逃逸闭包  
        // 它用到了实例成员（属性、方法），编译器会强制要求明确写出self  
        DispatchQueue.global().async {  
            self.fn()  
        }  
    }  
}
```



逃逸闭包的注意点

■ 逃逸闭包不可以捕获inout参数

```
typealias Fn = () -> ()
func other1(_ fn: Fn) { fn() }
func other2(_ fn: @escaping Fn) { fn() }
func test(value: inout Int) -> Fn {
    other1 { value += 1 }

    // error: 逃逸闭包不能捕获inout参数
    other2 { value += 1 }

    func plus() { value += 1 }
    // error: 逃逸闭包不能捕获inout参数
    return plus
}
```

内存访问冲突 (Conflicting Access to Memory)

- 内存访问冲突会在两个访问满足下列条件时发生：
 - 至少一个是写入操作
 - 它们访问的是同一块内存
 - 它们的访问时间重叠（比如在同一个函数内）

```
// 不存在内存访问冲突
func plus(_ num: inout Int) -> Int { num + 1 }
var number = 1
number = plus(&number)
```

```
// 存在内存访问冲突
// Simultaneous accesses to 0x0, but modification requires exclusive access
var step = 1
func increment(_ num: inout Int) { num += step }
increment(&step)
```

```
// 解决内存访问冲突
var copyOfStep = step
increment(&copyOfStep)
step = copyOfStep
```



内存访问冲突

```
func balance(_ x: inout Int, _ y: inout Int) {  
    let sum = x + y  
    x = sum / 2  
    y = sum - x  
}  
  
var num1 = 42  
var num2 = 30  
balance(&num1, &num2) // OK  
balance(&num1, &num1) // Error
```

```
struct Player {  
    var name: String  
    var health: Int  
    var energy: Int  
    mutating func shareHealth(with teammate: inout Player) {  
        balance(&teammate.health, &health)  
    }  
}  
  
var oscar = Player(name: "Oscar", health: 10, energy: 10)  
var maria = Player(name: "Maria", health: 5, energy: 10)  
oscar.shareHealth(with: &maria) // OK  
oscar.shareHealth(with: &oscar) // Error
```

```
var tulpe = (health: 10, energy: 20)  
// Error  
balance(&tulpe.health, &tulpe.energy)  
  
var holly = Player(name: "Holly", health: 10, energy: 10)  
// Error  
balance(&holly.health, &holly.energy)
```



内存访问冲突

- 如果下面的条件可以满足，就说明重叠访问结构体的属性是安全的
 - 你只访问实例存储属性，不是计算属性或者类属性
 - 结构体是局部变量而非全局变量
 - 结构体要么没有被闭包捕获要么只被非逃逸闭包捕获

```
// Ok
func test() {
    var tulpe = (health: 10, energy: 20)
    balance(&tulpe.health, &tulpe.energy)

    var holly = Player(name: "Holly", health: 10, energy: 10)
    balance(&holly.health, &holly.energy)
}
test()
```

指针

■ Swift中也有专门的指针类型，这些都被定性为“Unsafe”（不安全的），常见的有以下4种类型

- `UnsafePointer<Pointee>` 类似于 `const Pointee *`
- `UnsafeMutablePointer<Pointee>` 类似于 `Pointee *`
- `UnsafeRawPointer` 类似于 `const void *`
- `UnsafeMutableRawPointer` 类似于 `void *`

```
var age = 10
func test1(_ ptr: UnsafeMutablePointer<Int>) {
    ptr.pointee += 10
}
func test2(_ ptr: UnsafePointer<Int>) {
    print(ptr.pointee)
}
test1(&age)
test2(&age) // 20
print(age) // 20
```

```
var age = 10
func test3(_ ptr: UnsafeMutableRawPointer) {
    ptr.storeBytes(of: 20, as: Int.self)
}
func test4(_ ptr: UnsafeRawPointer) {
    print(ptr.load(as: Int.self))
}
test3(&age)
test4(&age) // 20
print(age) // 20
```



指针的应用示例

```
var arr = NSArray(objects: 11, 22, 33, 44)
arr.enumerateObjects { (obj, idx, stop) in
    print(idx, obj)
    if idx == 2 { // 下标为2就停止遍历
        stop.pointee = true
    }
}
```

```
var arr = NSArray(objects: 11, 22, 33, 44)
for (idx, obj) in arr.enumerated() {
    print(idx, obj)
    if idx == 2 {
        break
    }
}
```



获得指向某个变量的指针

```
var age = 11
var ptr1 = withUnsafeMutablePointer(to: &age) { $0 }
var ptr2 = withUnsafePointer(to: &age) { $0 }
ptr1.pointee = 22
print(ptr2.pointee) // 22
print(age) // 22

var ptr3 = withUnsafeMutablePointer(to: &age) { UnsafeMutableRawPointer($0) }
var ptr4 = withUnsafePointer(to: &age) { UnsafeRawPointer($0) }
ptr3.storeBytes(of: 33, as: Int.self)
print(ptr4.load(as: Int.self)) // 33
print(age) // 33
```



获得指向堆空间实例的指针

```
class Person {}  
var person = Person()  
var ptr = withUnsafePointer(to: &person) { UnsafeRawPointer($0) }  
var heapPtr = UnsafeRawPointer(bitPattern: ptr.load(as: UInt.self))  
print(heapPtr!)
```



创建指针

```
var ptr = UnsafeRawPointer(bitPattern: 0x100001234)
```

```
// 创建
var ptr = malloc(16)
// 存
ptr?.storeBytes(of: 11, as: Int.self)
ptr?.storeBytes(of: 22, toByteOffset: 8, as: Int.self)
// 取
print((ptr?.load(as: Int.self))!) // 11
print((ptr?.load(fromByteOffset: 8, as: Int.self))!) // 22
// 销毁
free(ptr)
```

```
var ptr = UnsafeMutableRawPointer.allocate(byteCount: 16, alignment: 1)
ptr.storeBytes(of: 11, as: Int.self)
ptr.advanced(by: 8).storeBytes(of: 22, as: Int.self)
print(ptr.load(as: Int.self)) // 11
print(ptr.advanced(by: 8).load(as: Int.self)) // 22
ptr.deallocate()
```

创建指针

```
var ptr = UnsafeMutablePointer<Int>.allocate(capacity: 3)
ptr.initialize(to: 11)
ptr.successor().initialize(to: 22)
ptr.successor().successor().initialize(to: 33)

print(ptr.pointee) // 11
print((ptr + 1).pointee) // 22
print((ptr + 2).pointee) // 33

print(ptr[0]) // 11
print(ptr[1]) // 22
print(ptr[2]) // 33

ptr.deinitialize(count: 3)
ptr.deallocate()
```



创建指针

```
class Person {  
    var age: Int  
    var name: String  
    init(age: Int, name: String) {  
        self.age = age  
        self.name = name  
    }  
    deinit { print(name, "deinit") }  
}
```

```
var ptr = UnsafeMutablePointer<Person>.allocate(capacity: 3)  
ptr.initialize(to: Person(age: 10, name: "Jack"))  
(ptr + 1).initialize(to: Person(age: 11, name: "Rose"))  
(ptr + 2).initialize(to: Person(age: 12, name: "Kate"))  
// Jack_deinit  
// Rose_deinit  
// Kate_deinit  
ptr.deinitialize(count: 3)  
ptr.deallocate()
```



指针之间的转换

```
var ptr = UnsafeMutableRawPointer.allocate(byteCount: 16, alignment: 1)

ptr.assumingMemoryBound(to: Int.self).pointee = 11
(ptr + 8).assumingMemoryBound(to: Double.self).pointee = 22.0

print(unsafeBitCast(ptr, to: UnsafePointer<Int>.self).pointee) // 11
print(unsafeBitCast(ptr + 8, to: UnsafePointer<Double>.self).pointee) // 22.0

ptr.deallocate()
```

- unsafeBitCast是忽略数据类型的强制转换，不会因为数据类型的变化而改变原来的内存数据
- 类似于C++中的reinterpret_cast

```
class Person {}

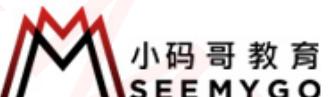
var person = Person()
var ptr = unsafeBitCast(person, to: UnsafeRawPointer.self)
print(ptr)
```

字面量

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



字面量 (Literal)

```
var age = 10
var isRed = false
var name = "Jack"
```

- 上面代码中的`10`、`false`、`"Jack"`就是字面量

- 常见字面量的默认类型

- `public typealias IntegerLiteralType = Int`
- `public typealias FloatLiteralType = Double`
- `public typealias BooleanLiteralType = Bool`
- `public typealias StringLiteralType = String`

- Swift自带的绝大部分类型，都支持直接通过字面量进行初始化

- `Bool`、`Int`、`Float`、`Double`、`String`、`Array`、`Dictionary`、`Set`、`Optional`等

```
// 可以通过typealias修改字面量的默认类型
typealias FloatLiteralType = Float
typealias IntegerLiteralType = UInt8
var age = 10 // UInt8
var height = 1.68 // Float
```



字面量协议

- Swift自带类型之所以能够通过字面量初始化，是因为它们遵守了对应的协议
- Bool : ExpressibleByBooleanLiteral
- Int : ExpressibleByIntegerLiteral
- Float、Double : ExpressibleByIntegerLiteral、ExpressibleByFloatLiteral
- Dictionary : ExpressibleByDictionaryLiteral
- String : ExpressibleByStringLiteral
- Array、Set : ExpressibleByArrayLiteral
- Optional : ExpressibleByNilLiteral

```
var b: Bool = false // ExpressibleByBooleanLiteral
var i: Int = 10 // ExpressibleByIntegerLiteral
var f0: Float = 10 // ExpressibleByIntegerLiteral
var f1: Float = 10.0 // ExpressibleByFloatLiteral
var d0: Double = 10 // ExpressibleByIntegerLiteral
var d1: Double = 10.0 // ExpressibleByFloatLiteral
var s: String = "jack" // ExpressibleByStringLiteral
var arr: Array = [1, 2, 3] // ExpressibleByArrayLiteral
var set: Set = [1, 2, 3] // ExpressibleByArrayLiteral
var dict: Dictionary = ["jack" : 60] // ExpressibleByDictionaryLiteral
var o: Optional<Int> = nil // ExpressibleByNilLiteral
```



字面量协议应用

```
extension Int : ExpressibleByBooleanLiteral {
    public init(booleanLiteral value: Bool) { self = value ? 1 : 0 }
}
var num: Int = true
print(num) // 1
```

■ 有点类似于C++中的转换构造函数

```
class Student : ExpressibleByIntegerLiteral, ExpressibleByFloatLiteral, ExpressibleByStringLiteral,
CustomStringConvertible {
    var name: String = ""
    var score: Double = 0
    required init(floatLiteral value: Double) { self.score = value }
    required init(integerLiteral value: Int) { self.score = Double(value) }
    required init(stringLiteral value: String) { self.name = value }
    required init(unicodeScalarLiteral value: String) { self.name = value }
    required init(extendedGraphemeClusterLiteral value: String) { self.name = value }
    var description: String { "name=\(name),score=\(score)" }
}
var stu: Student = 90
print(stu) // name=,score=90.0
stu = 98.5
print(stu) // name=,score=98.5
stu = "Jack"
print(stu) // name=Jack, score=0.0
```

字面量协议应用

```
struct Point {  
    var x = 0.0, y = 0.0  
}  
  
extension Point : ExpressibleByArrayLiteral, ExpressibleByDictionaryLiteral {  
    init(arrayLiteral elements: Double...) {  
        guard elements.count > 0 else { return }  
        self.x = elements[0]  
        guard elements.count > 1 else { return }  
        self.y = elements[1]  
    }  
    init(dictionaryLiteral elements: (String, Double)...) {  
        for (k, v) in elements {  
            if k == "x" { self.x = v }  
            else if k == "y" { self.y = v }  
        }  
    }  
}  
  
var p: Point = [10.5, 20.5]  
print(p) // Point(x: 10.5, y: 20.5)  
p = ["x" : 11, "y" : 22]  
print(p) // Point(x: 11.0, y: 22.0)
```

模式匹配

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





模式 (Pattern)

- 什么是模式 ?
 - 模式是用于匹配的规则 , 比如 switch 的 case 、捕捉错误的 catch 、 if\guard\while\for 语句的条件等

- Swift 中的模式有
 - 通配符模式 (Wildcard Pattern)
 - 标识符模式 (Identifier Pattern)
 - 值绑定模式 (Value-Binding Pattern)
 - 元组模式 (Tuple Pattern)
 - 枚举 Case 模式 (Enumeration Case Pattern)
 - 可选模式 (Optional Pattern)
 - 类型转换模式 (Type-Casting Pattern)
 - 表达式模式 (Expression Pattern)



通配符模式 (Wildcard Pattern)

- `_` 匹配任何值
- `_?` 匹配非`nil`值

```
enum Life {  
    case human(name: String, age: Int?)  
    case animal(name: String, age: Int?)  
}
```

```
func check(_ life: Life) {  
    switch life {  
        case .human(let name, _):  
            print("human", name)  
        case .animal(let name, _?):  
            print("animal", name)  
        default:  
            print("other")  
    }  
}
```

```
check(.human(name: "Rose", age: 20)) // human Rose  
check(.human(name: "Jack", age: nil)) // human Jack  
check(.animal(name: "Dog", age: 5)) // animal Dog  
check(.animal(name: "Cat", age: nil)) // other
```



标识符模式 (Identifier Pattern)

- 给对应的变量、常量名赋值

```
var age = 10  
let name = "jack"
```



值绑定模式 (Value-Binding Pattern)

```
let point = (3, 2)
switch point {
case let (x, y):
    print("The point is at (\(x), \(y)).")
}
```



元组模式 (Tuple Pattern)

```
let points = [(0, 0), (1, 0), (2, 0)]
for (x, _) in points {
    print(x)
}
```

```
let name: String? = "jack"
let age = 18
let info: Any = [1, 2]
switch (name, age, info) {
case (_?, _, _ as String):
    print("case")
default:
    print("default")
} // default
```

```
var scores = ["jack" : 98, "rose" : 100, "kate" : 86]
for (name, score) in scores {
    print(name, score)
}
```



枚举Case模式 (Enumeration Case Pattern)

- if case语句等价于只有1个case的switch语句

```
let age = 2
// 原来的写法
if age >= 0 && age <= 9 {
    print("[0, 9]")
}

// 枚举Case模式
if case 0...9 = age {
    print("[0, 9]")
}
guard case 0...9 = age else { return }
print("[0, 9]")
```

```
switch age {
case 0...9: print("[0, 9]")
default: break
}
```

```
let ages: [Int?] = [2, 3, nil, 5]
for case nil in ages {
    print("有nil值")
    break
} // 有nil值
```

```
let points = [(1, 0), (2, 1), (3, 0)]
for case let (x, 0) in points {
    print(x)
} // 1 3
```



可选模式 (Optional Pattern)

```
let age: Int? = 42
if case .some(let x) = age { print(x) }
if case let x? = age { print(x) }
```

```
let ages: [Int?] = [nil, 2, 3, nil, 5]
for case let age? in ages {
    print(age)
} // 2 3 5
```

```
let ages: [Int?] = [nil, 2, 3, nil, 5]
for item in ages {
    if let age = item {
        print(age)
    }
} // 跟上面的for, 效果是等价的
```

```
func check(_ num: Int?) {
    switch num {
        case 2?: print("2")
        case 4?: print("4")
        case 6?: print("6")
        case _: print("other")
        case _: print("nil")
    }
}
check(4) // 4
check(8) // other
check(nil) // nil
```



类型转换模式 (Type-Casting Pattern)

```
let num: Any = 6
switch num {
case is Int:
    // 编译器依然认为num是Any类型
    print("is Int", num)
//case let n as Int:
//    print("as Int", n + 1)
default:
    break
}
```

```
class Animal { func eat() { print(type(of: self), "eat") } }
class Dog : Animal { func run() { print(type(of: self), "run") } }
class Cat : Animal { func jump() { print(type(of: self), "jump") } }
func check(_ animal: Animal) {
    switch animal {
    case let dog as Dog:
        dog.eat()
        dog.run()
    case is Cat:
        animal.jump()
    default: break
    }
}
// Dog eat
// Dog run
check(Dog())
// Cat eat
check(Cat())
```



表达式模式 (Expression Pattern)

- 表达式模式用在case中

```
let point = (1, 2)
switch point {
case (0, 0):
    print("(0, 0) is at the origin.")
case (-2...2, -2...2):
    print("\(point.0), \(point.1)) is near the origin.")
default:
    print("The point is at (\(point.0), \(point.1)).")
} // (1, 2) is near the origin.
```



自定义表达式模式

- 可以通过重载运算符，自定义表达式模式的匹配规则

```
struct Student {  
    var score = 0, name = ""  
    static func ~= (pattern: Int, value: Student) -> Bool { value.score >= pattern }  
    static func ~= (pattern: ClosedRange<Int>, value: Student) -> Bool { pattern.contains(value.score) }  
    static func ~= (pattern: Range<Int>, value: Student) -> Bool { pattern.contains(value.score) }  
}
```

```
var stu = Student(score: 75, name: "Jack")  
switch stu {  
case 100: print(">= 100")  
case 90: print(">= 90")  
case 80..  
    <90: print("[80, 90)")  
case 60...79: print("[60, 79]")  
case 0: print(">= 0")  
default: break  
} // [60, 79]
```

```
if case 60 = stu {  
    print(">= 60")  
} // >= 60
```

```
var info = (Student(score: 70, name: "Jack"), "及格")  
switch info {  
case let (60, text): print(text)  
default: break  
} // 及格
```



自定义表达式模式

```
extension String {  
    static func ~= (pattern: (String) -> Bool, value: String) -> Bool {  
        pattern(value)  
    }  
}  
  
func hasPrefix(_ prefix: String) -> ((String) -> Bool) { { $0.hasPrefix(prefix) } }  
func hasSuffix(_ suffix: String) -> ((String) -> Bool) { { $0.hasSuffix(suffix) } }  
  
var str = "jack"  
switch str {  
case hasPrefix("j"), hasSuffix("k"):  
    print("以j开头, 以k结尾")  
default: break  
} // 以j开头, 以k结尾
```



自定义表达式模式

```
func isEven(_ i: Int) -> Bool { i % 2 == 0 }
func isOdd(_ i: Int) -> Bool { i % 2 != 0 }

extension Int {
    static func ~= (pattern: (Int) -> Bool, value: Int) -> Bool {
        pattern(value)
    }
}
```

```
var age = 9
switch age {
case isEven:
    print("偶数")
case isOdd:
    print("奇数")
default:
    print("其他")
}
```

```
prefix operator ~>
prefix operator ~>=
prefix operator ~<
prefix operator ~<=
prefix func ~> (_ i: Int) -> ((Int) -> Bool) { { $0 > i } }
prefix func ~>= (_ i: Int) -> ((Int) -> Bool) { { $0 >= i } }
prefix func ~< (_ i: Int) -> ((Int) -> Bool) { { $0 < i } }
prefix func ~<= (_ i: Int) -> ((Int) -> Bool) { { $0 <= i } }
```

```
var age = 9
switch age {
case ~>=0:
    print("1")
case ~>10:
    print("2")
default: break
} // [0, 10]
```

where

- 可以使用 `where` 为模式匹配增加匹配条件

```
var data = (10, "Jack")
switch data {
case let (age, _) where age > 10:
    print(data.1, "age>10")
case let (age, _) where age > 0:
    print(data.1, "age>0")
default: break
}
```

```
var ages = [10, 20, 44, 23, 55]
for age in ages where age > 30 {
    print(age)
} // 44 55
```

```
protocol Stackable { associatedtype Element }
protocol Container {
    associatedtype Stack : Stackable where Stack.Element : Equatable
}
```

```
func equal<S1: Stackable, S2: Stackable>(_ s1: S1, _ s2: S2) -> Bool
    where S1.Element == S2.Element, S1.Element : Hashable {
        return false
}
```

```
extension Container where Self.Stack.Element : Hashable { }
```

从OC到Swift

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码马拉松



MARK、TODO、FIXME

- // MARK: 类似于OC中的 #pragma mark
- // MARK: - 类似于OC中的 #pragma mark -
- // TODO: 用于标记未完成的任务
- // FIXME: 用于标记待修复的问题

```
func test() {  
    // TODO: 未完成  
}  
  
func test2() {  
    var age = 10  
    // FIXME: 有待修复  
    age += 20  
}
```



```
public class Person {  
    // MARK: - 属性  
    var age = 0  
    var weight = 0  
    var height = 0  
  
    // MARK: - 私有方法  
    // MARK: 跑步  
    private func run1() {}  
    private func run2() {}  
    // MARK: 走路  
    private func walk1() {}  
    private func walk2() {}  
  
    // MARK: - 公共方法  
    public func eat1() {}  
    public func eat2() {}  
}
```



条件编译

```
// 操作系统: macOS\iOS\tvOS\watchOS\Linux\Android\Windows\FreeBSD
#if os(macOS) || os(iOS)
// CPU架构: i386\x86_64\arm\arm64
#elseif arch(x86_64) || arch(arm64)
// swift版本
#elseif swift(<5) && swift(>=3)
// 模拟器
#elseif targetEnvironment(simulator)
// 可以导入某模块
#elseif canImport(Foundation)
#else
#endif
```



条件编译

The screenshot shows the Xcode build settings interface for a project named "备课_Swift". The "Build Settings" tab is selected. In the search bar, the query "swift compiler - custom" is entered. Under the "Swift Compiler - Custom Flags" section, the "Active Compilation Conditions" are set to "Debug" and "TEST". The "Other Swift Flags" section contains the flag "-D OTHER".

// debug模式

#if DEBUG

// release模式

#else

#endif

```
#if TEST
```

```
print("test")
```

```
#endif
```

```
#if OTHER
```

```
print("other")
```

```
#endif
```

```
func log<T>(_ msg: T,  
             file: NSString = #file,  
             line: Int = #line,  
             fn: String = #function) {  
    #if DEBUG  
    let prefix = "\(file.lastPathComponent)_\(line)_\(fn):"  
    print(prefix, msg)  
    #endif  
}
```



系统版本检测

```
if #available(iOS 10, macOS 10.12, *) {  
    // 对于iOS平台, 只在iOS10及以上版本执行  
    // 对于macOS平台, 只在macOS 10.12及以上版本执行  
    // 最后的*表示在其他所有平台都执行  
}
```



API可用性说明

```
@available(iOS 10, macOS 10.15, *)
class Person {}

struct Student {
    @available(*, unavailable, renamed: "study")
    func study_() {}
    func study() {}

    @available(iOS, deprecated: 11)
    @available(macOS, deprecated: 10.12)
    func run() {}
}
```

- 更多用法参考：<https://docs.swift.org/swift-book/ReferenceManual/Attributes.html>



iOS程序的入口

- 在AppDelegate上面默认有个@UIApplicationMain标记，这表示
- 编译器自动生成入口代码（main函数代码），自动设置AppDelegate为APP的代理
- 也可以删掉@UIApplicationMain，自定义入口代码：新建一个main.swift文件

```
//  
//  main.swift  
//  TestiOS  
//  
//  Created by MJ Lee on 2019/7/22.  
//  Copyright © 2019 MJ Lee. All rights reserved.  
  
  
import UIKit  
  
class MJApplication : UIApplication {}  
  
UIApplicationMain(CommandLine.argc,  
                  CommandLine.unsafeArgv,  
                  NSStringFromClass(MJApplication.self),  
                  NSStringFromClass(AppDelegate.self))
```

Swift调用OC

- 新建1个桥接头文件，文件名格式默认为：**{targetName}-Bridging-Header.h**



- 在 **{targetName}-Bridging-Header.h** 文件中 `#import` OC需要暴露给Swift的内容

```
#import "MJPerson.h"
```



Swift调用OC – MJPerson.h

```
int sum(int a, int b);

@interface MJPerson : NSObject
@property (nonatomic, assign) NSInteger age;
@property (nonatomic, copy) NSString *name;

- (instancetype)initWithAge:(NSInteger)age name:(NSString *)name;
+ (instancetype)personWithAge:(NSInteger)age name:(NSString *)name;

- (void)run;
+ (void)run;

- (void)eat:(NSString *)food other:(NSString *)other;
+ (void)eat:(NSString *)food other:(NSString *)other;
@end
```



Swift调用OC – MJPerson.m

```
@implementation MJPerson
- (instancetype)initWithAge:(NSInteger)age name:(NSString *)name {
    if (self = [super init]) {
        self.age = age;
        self.name = name;
    }
    return self;
}
+ (instancetype)personWithAge:(NSInteger)age name:(NSString *)name {
    return [[self alloc] initWithAge:age name:name];
}
+ (void)run { NSLog(@"Person +run"); }
- (void)run { NSLog(@"%@", _name); }

+ (void)eat:(NSString *)food other:(NSString *)other { NSLog(@"Person +eat %@ %@", food, other); }
- (void)eat:(NSString *)food other:(NSString *)other { NSLog(@"%@", _name, food, other); }
@end

int sum(int a, int b) { return a + b; }
```



Swift调用OC – Swift代码

```
var p = MJPerson(age: 10, name: "Jack")
p.age = 18
p.name = "Rose"
p.run() // 18 Rose -run
p.eat("Apple", other: "Water") // 18 Rose -eat Apple Water

MJPerson.run() // Person +run
MJPerson.eat("Pizza", other: "Banana") // Person +eat Pizza Banana

print(sum(10, 20)) // 30
```



Swift调用OC – Swift代码

```
var p = MJPerson(age: 10, name: "Jack")
p.age = 18
p.name = "Rose"
p.run() // 18 Rose -run
p.eat("Apple", other: "Water") // 18 Rose -eat Apple Water

MJPerson.run() // Person +run
MJPerson.eat("Pizza", other: "Banana") // Person +eat Pizza Banana

print(sum(10, 20)) // 30
```



Swift调用OC – @_silgen_name

- 如果C语言暴露给Swift的函数名跟Swift中的其他函数名冲突了
- 可以在Swift中使用 `@_silgen_name` 修改C函数名

```
// C语言
int sum(int a, int b) {
    return a + b;
}
```

```
// Swift
@_silgen_name("sum") func swift_sum(_ v1: Int32, _ v2: Int32) -> Int32
print(swift_sum(10, 20)) // 30
print(sum(10, 20)) // 30
```



OC调用Swift

- Xcode已经默认生成一个用于OC调用Swift的头文件，文件名格式是： **{targetName}-Swift.h**

The screenshot shows the Xcode Build Settings interface. The 'Build Settings' tab is selected. In the 'Swift Compiler - General' section, under 'Objective-C Generated Interface Header Name', the 'Debug' and 'Release' configurations both have the value '备课_Swift-Swift.h'.

Setting	备课_Swift
Debug	备课_Swift-Swift.h
Release	备课_Swift-Swift.h



OC调用Swift – Car.swift

```
import Foundation

@objcMembers class Car: NSObject {
    var price: Double
    var band: String
    init(price: Double, band: String) {
        self.price = price
        self.band = band
    }
    func run() { print(price, band, "run") }
    static func run() { print("Car run") }
}

extension Car {
    func test() { print(price, band, "test") }
}
```

- Swift暴露给OC的类最终继承自**NSObject**
- 使用**@objc**修饰需要暴露给OC的成员
- 使用**@objcMembers**修饰类
 - 代表默认所有成员都会暴露给OC（包括扩展中定义的成员）
 - 最终是否成功暴露，还需要考虑成员自身的访问级别



OC调用Swift – {targetName}-Swift.h

- Xcode会根据Swift代码生成对应的OC声明，写入 **{targetName}-Swift.h** 文件

```
@interface Car : NSObject
@property (nonatomic) double price;
@property (nonatomic, copy) NSString * _Nonnull band;
- (nonnullinstancetype)initWithPrice:(double)price band:(NSString * _Nonnull)band OBJC_DESIGNATED_INITIALIZER;
- (void)run;
+ (void)run;
- (nonnullinstancetype)init SWIFT_UNAVAILABLE;
+ (nonnullinstancetype)new SWIFT_UNAVAILABLE_MSG("-init is unavailable");
@end

@interface Car (SWIFT_EXTENSION(备课_Swift))
- (void)test;
@end
```



OC调用Swift – OC代码

```
#import "备课_Swift-Swift.h"
int sum(int a, int b) {
    Car *c = [[Car alloc] initWithPrice:10.5 band:@"BMW"];
    c.band = @"Bently";
    c.price = 108.5;
    [c run]; // 108.5 Bently run
    [c test]; // 108.5 Bently test
    [Car run]; // Car run
    return a + b;
}
```

OC调用Swift – @objc

- 可以通过 `@objc` 重命名Swift暴露给OC的符号名（类名、属性名、函数名等）

```
@objc(MJCar)
@objcMembers class Car: NSObject {
    var price: Double
    @objc(name)
    var band: String
    init(price: Double, band: String) {
        self.price = price
        self.band = band
    }
    @objc(drive)
    func run() { print(price, band, "run") }
    static func run() { print("Car run") }
}
extension Car {
    @objc(exec:v2:)
    func test() { print(price, band, "test") }
}
```

```
MJCar *c = [[MJCar alloc] initWithPrice:10.5 band:@"BMW"];
c.name = @"Bently";
c.price = 108.5;
[c drive]; // 108.5 Bently run
[c exec:10 v2:20]; // 108.5 Bently test
[MJCar run]; // Car run
```



选择器 (Selector)

- Swift中依然可以使用选择器，使用#selector(name)定义一个选择器
- 必须是被@objcMembers或@objc修饰的方法才可以定义选择器

```
@objcMembers class Person: NSObject {  
    func test1(v1: Int) { print("test1") }  
    func test2(v1: Int, v2: Int) { print("test2(v1:v2:)") }  
    func test2(_ v1: Double, _ v2: Double) { print("test2(_:_:)") }  
    func run() {  
        perform(#selector(test1))  
        perform(#selector(test1(v1:)))  
        perform(#selector(test2(v1:v2:)))  
        perform(#selector(test2(_:_:)))  
        perform(#selector(test2 as (Double, Double) -> Void))  
    }  
}
```

String

- Swift的字符串类型String，跟OC的NSString，在API设计上还是有较大差异

```
// 空字符串
var emptyStr1 = ""
var emptyStr2 = String()
```

```
var str = "123456"
print(str.hasPrefix("123")) // true
print(str.hasSuffix("456")) // true
```

```
var str: String = "1"
// 拼接, jack_rose
str.append("_2")
// 重载运算符 +
str = str + "_3"
// 重载运算符 +=
str += "_4"
// \()插值
str = "\(str)_5"
// 长度, 9, 1_2_3_4_5
print(str.count)
```



String的插入和删除

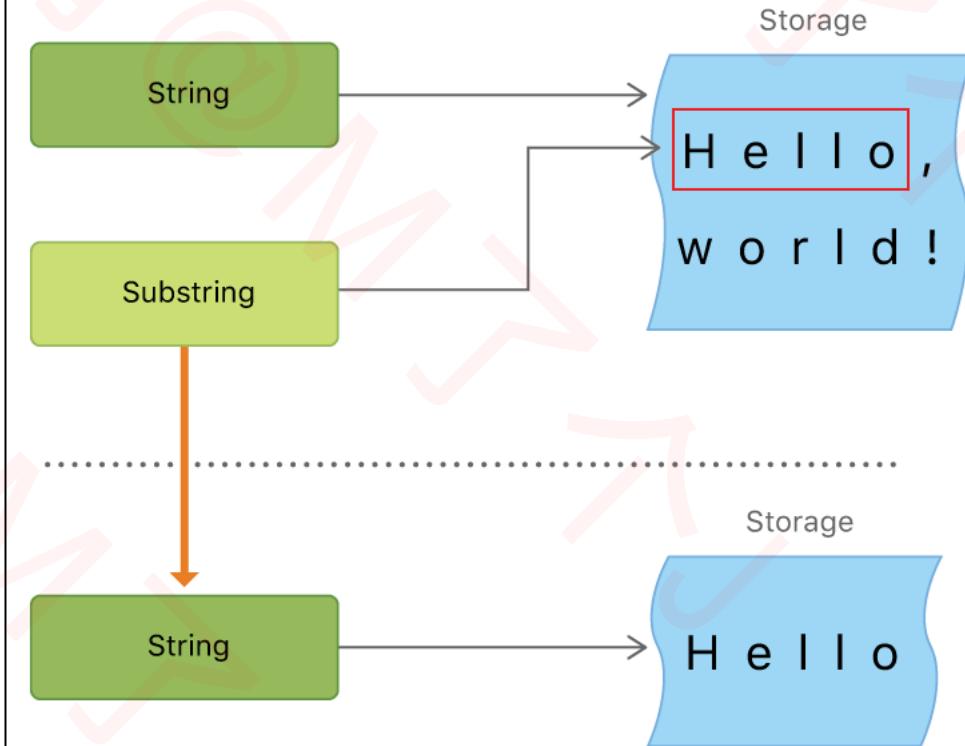
```
var str = "1_2"
// 1_2_
str.insert("_", at: str.endIndex)
// 1_2_3_4
str.insert(contentsOf: "3_4", at: str.endIndex)
// 1666_2_3_4
str.insert(contentsOf: "666", at: str.index(after: str.startIndex))
// 1666_2_3_8884
str.insert(contentsOf: "888", at: str.index(before: str.endIndex))
// 1666hello_2_3_8884
str.insert(contentsOf: "hello", at: str.index(str.startIndex, offsetBy: 4))
```

```
// 666hello_2_3_8884
str.remove(at: str.firstIndex(of: "1")!)
// hello_2_3_8884
str.removeAll { $0 == "6" }
var range = str.index(str.endIndex, offsetBy: -4)..
```

Substring

- `String`可以通过下标、`prefix`、`suffix`等截取子串，子串类型不是`String`，而是`Substring`

```
var str = "1_2_3_4_5"
// 1_2
var substr1 = str.prefix(3)
// 4_5
var substr2 = str.suffix(3)
// 1_2
var range = str.startIndex..
```



- `Substring`和它的`base`，共享字符串数据
- `Substring`发生修改 或者 转为`String`时，会分配新的内存存储字符串数据



String 与 Character

```
for c in "jack" { // c是Character类型
    print(c)
}

var str = "jack"
// c是Character类型
var c = str[str.startIndex]
```



String相关的协议

- BidirectionalCollection 协议包含的部分内容
 - startIndex、endIndex 属性、index 方法
 - String、Array 都遵守了这个协议
- RangeReplaceableCollection 协议包含的部分内容
 - append、insert、remove 方法
 - String、Array 都遵守了这个协议
- Dictionary、Set 也有实现上述协议中声明的一些方法，只是并没有遵守上述协议



多行String

```
let str = """"  
1  
    "2"  
3  
    '4'  
....
```

```
// 如果要显示3引号，至少转义1个引号  
let str = """"  
Escaping the first quote \""""  
Escaping two quotes \"\""  
Escaping all three quotes \"\"\"  
....
```

```
1  
    "2"  
3""""  
    '4'
```

```
Escaping the first quote """  
Escaping two quotes """  
Escaping all three quotes """
```

```
// 以下2个字符串是等价的  
let str1 = "These are the same."  
let str2 = """  
These are the same.  
....
```

```
// 缩进以结尾的3引号为对齐线  
let str = """"  
    1  
        2  
    3  
        4  
....
```

```
1  
    2  
3  
    4
```



String 与 NSString

- String 与 NSString 之间可以随时随地桥接转换
- 如果你觉得String的API过于复杂难用，可以考虑将String转为NSString

```
var str1: String = "jack"
var str2: NSString = "rose"

var str3 = str1 as NSString
var str4 = str2 as String

// ja
var str5 = str3.substring(with: NSRange(location: 0, length: 2))
print(str5)
```

- 比较字符串内容是否等价
- String使用 == 运算符
- NSString使用isEqual方法，也可以使用 == 运算符（本质还是调用了isEqual方法）



Swift、OC桥接转换表

String	\cong	NSString
String	\leftarrow	NSMutableString
Array	\cong	NSArray
Array	\leftarrow	NSMutableArray
Dictionary	\cong	NSDictionary
Dictionary	\leftarrow	NSMutableDictionary
Set	\cong	NSSet
Set	\leftarrow	NSMutableSet



只能被class继承的协议

```
protocol Runnable1: AnyObject {}
protocol Runnable2: class {}
@objc protocol Runnable3 { }
```

- 被 `@objc` 修饰的协议，还可以暴露给OC去遵守实现

可选协议

- 可以通过 `@objc` 定义可选协议，这种协议只能被 `class` 遵守

```
@objc protocol Runnable {
    func run1()
    @objc optional func run2()
    func run3()
}

class Dog: Runnable {
    func run3() { print("Dog run3") }
    func run1() { print("Dog run1") }
}
var d = Dog()
d.run1() // Dog run1
d.run3() // Dog run3
```

dynamic

- 被 @objc dynamic 修饰的内容会具有动态性，比如调用方法会走runtime那一套流程

```
class Dog: NSObject {
    @objc dynamic func test1() {}
    func test2() {}
}
var d = Dog()
d.test1()
d.test2()
```

```
movq -0x70(%rbp), %rcx
movq (%rcx), %rdx
andq (%rax), %rdx
movq %rcx, %r13
callq *0x50(%rdx)
```

test2

```
movq 0x8fb4(%rip), %rsi ; "test1"
movq -0x60(%rbp), %rax
movq %rax, %rdi
callq 0x100007c5e ; symbol stub for: objc_msgSend
```

- Swift 支持 KVC \ KVO 的条件
- 属性所在的类、监听器最终继承自 `NSObject`
- 用 `@objc dynamic` 修饰对应的属性

```
class Observer: NSObject {  
    override func observeValue(forKeyPath keyPath: String?,  
                             of object: Any?,  
                             change: [NSKeyValueChangeKey : Any]?,  
                             context: UnsafeMutableRawPointer?) {  
        print("observeValue", change? [.newKey] as Any)  
    }  
}
```

```
class Person: NSObject {  
    @objc dynamic var age: Int = 0  
    var observer: Observer = Observer()  
    override init() {  
        super.init()  
        self.addObserver(observer,  
                         forKeyPath: "age",  
                         options: .new,  
                         context: nil)  
    }  
    deinit {  
        self.removeObserver(observer,  
                            forKeyPath: "age")  
    }  
}  
var p = Person()  
// observeValue Optional(20)  
p.age = 20  
// observeValue Optional(25)  
p.setValue(25, forKey: "age")
```



block方式的KVO

```
class Person: NSObject {
    @objc dynamic var age: Int = 0
    var observation: NSKeyValueObservation?
    override init() {
        super.init()
        observation = observe(\Person.age, options: .new) {
            (person, change) in
            print(change.newValue as Any)
        }
    }
    var p = Person()
    // Optional(20)
    p.age = 20
    // Optional(25)
    p.setValue(25, forKey: "age")
```

关联对象 (Associated Object)

- 在Swift中，`class`依然可以使用关联对象
- 默认情况，`extension`不可以增加存储属性
- 借助关联对象，可以实现类似`extension`为`class`增加存储属性的效果

```
class Person {}  
extension Person {  
    private static var AGE_KEY: Void?  
    var age: Int {  
        get {  
            (objc_getAssociatedObject(self, &Self.AGE_KEY) as? Int) ?? 0  
        }  
        set {  
            objc_setAssociatedObject(self,  
                                    &Self.AGE_KEY,  
                                    newValue,  
                                    .OBJC_ASSOCIATION_ASSIGN)  
        }  
    }  
}
```

```
var p = Person()  
print(p.age) // 0  
p.age = 10  
print(p.age) // 10
```



资源名管理

```
let img = UIImage(named: "logo")

let btn = UIButton(type: .custom)
btn.setTitle("添加", for: .normal)

performSegue(withIdentifier: "login_main", sender: self)
```

```
let img = UIImage(R.image.logo)

let btn = UIButton(type: .custom)
btn.setTitle(R.string.add, for: .normal)

performSegue(withIdentifier: R.segue.login_main, sender: self)
```

```
enum R {
    enum string: String {
        case add = "添加"
    }
    enum image: String {
        case logo
    }
    enum segue: String {
        case login_main
    }
}
```

- 这种做法实际上是参考了Android的资源名管理方式

资源名管理

```
extension UIImage {
    convenience init?(name: R.image) {
        self.init(named: name.rawValue)
    }
}

extension UIViewController {
    func performSegue(withIdentifier identifier: R.segue, sender: Any?) {
        performSegue(withIdentifier: identifier.rawValue, sender: sender)
    }
}

extension UIButton {
    func setTitle(_ title: R.string, for state: UIControl.State) {
        setTitle(title.rawValue, for: state)
    }
}
```



资源名管理的其他思路

```
let img = UIImage(named: "logo")  
  
let font = UIFont(name: "Arial", size: 14)
```

```
let img = R.image.logo  
  
let font = R.font.arial(14)
```

```
enum R {  
    enum image {  
        static var logo = UIImage(named: "logo")  
    }  
    enum font {  
        static func arial(_ size: CGFloat) -> UIFont? {  
            UIFont(name: "Arial", size: size)  
        }  
    }  
}
```

■ 更多优秀的思路参考

- <https://github.com/mac-cain13/R.swift>
- <https://github.com/SwiftGen/SwiftGen>

多线程开发 - 异步

```
public typealias Task = () -> Void

public static func async(_ task: @escaping Task) {
    _async(task)
}

public static func async(_ task: @escaping Task,
                        _ mainTask: @escaping Task) {
    _async(task, mainTask)
}

private static func _async(_ task: @escaping Task,
                         _ mainTask: Task? = nil) {
    let item = DispatchWorkItem(block: task)
    DispatchQueue.global().async(execute: item)
    if let main = mainTask {
        item.notify(queue: DispatchQueue.main, execute: main)
    }
}
```



多线程开发 – 延迟

多线程开发 – 异步延迟

```
@discardableResult
public static func asyncDelay(_ seconds: Double,
                               _ task: @escaping Task) -> DispatchWorkItem {
    return _asyncDelay(seconds, task)
}

@discardableResult
public static func asyncDelay(_ seconds: Double,
                               _ task: @escaping Task,
                               _ mainTask: @escaping Task) -> DispatchWorkItem {
    return _asyncDelay(seconds, task, mainTask)
}

private static func _asyncDelay(_ seconds: Double,
                               _ task: @escaping Task,
                               _ mainTask: Task? = nil) -> DispatchWorkItem {
    let item = DispatchWorkItem(block: task)
    DispatchQueue.global().asyncAfter(deadline: DispatchTime.now() + seconds,
                                      execute: item)
    if let main = mainTask {
        item.notify(queue: DispatchQueue.main, execute: main)
    }
    return item
}
```



多线程开发 – once

- `dispatch_once`在Swift中已被废弃，取而代之
- 可以用**类型属性**或者**全局变量\常量**
- 默认自带 `lazy + dispatch_once` 效果

```
fileprivate let initTask2: Void = {
    print("initTask2-----")
}()

class ViewController: UIViewController {
    static let initTask1: Void = {
        print("initTask1-----")
    }()
}

override func viewDidLoad() {
    super.viewDidLoad()

    let _ = Self.initTask1

    let _ = initTask2
}
```



多线程开发 – 加锁

■ gcd信号量

```
class Cache {  
    private static var data = [String: Any]()  
    private static var lock = DispatchSemaphore(value: 1)  
    static func set(_ key: String, _ value: Any) {  
        lock.wait()  
        defer { lock.signal() }  
  
        data[key] = value  
    }  
}
```

■ Foundation

```
private static var lock = NSLock()  
static func set(_ key: String, _ value: Any) {  
    lock.lock()  
    defer { lock.unlock() }  
}
```

```
private static var lock = NSRecursiveLock()  
static func set(_ key: String, _ value: Any) {  
    lock.lock()  
    defer { lock.unlock() }  
}
```

函数式编程

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



Array的常见操作

```
var arr = [1, 2, 3, 4]
// [2, 4, 6, 8]
var arr2 = arr.map { $0 * 2 }
// [2, 4]
var arr3 = arr.filter { $0 % 2 == 0 }
// 10
var arr4 = arr.reduce(0) { $0 + $1 }
// 10
var arr5 = arr.reduce(0, +)
```

```
func double(_ i: Int) -> Int { i * 2 }
var arr = [1, 2, 3, 4]
// [2, 4, 6, 8]
print(arr.map(double))
```

```
var arr = [1, 2, 3]
// [[1], [2, 2], [3, 3, 3]]
var arr2 = arr.map { Array.init(repeating: $0, count: $0) }
// [1, 2, 2, 3, 3, 3]
var arr3 = arr.flatMap { Array.init(repeating: $0, count: $0) }
```

```
var arr = ["123", "test", "jack", "-30"]
// [Optional(123), nil, nil, Optional(-30)]
var arr2 = arr.map { Int($0) }
// [123, -30]
var arr3 = arr.compactMap { Int($0) }
```

```
// 使用reduce实现map、filter的功能
var arr = [1, 2, 3, 4]
// [2, 4, 6, 8]
print(arr.map { $0 * 2 })
print(arr.reduce([]) { $0 + [$1 * 2] })

// [2, 4]
print(arr.filter { $0 % 2 == 0 })
print(arr.reduce([]) { $1 % 2 == 0 ? $0 + [$1] : $0 })
```



lazy的优化

```
let arr = [1, 2, 3]
let result = arr.lazy.map {
    (i: Int) -> Int in
    print("mapping \(i)")
    return i * 2
}
print("begin----")
print("mapped", result[0])
print("mapped", result[1])
print("mapped", result[2])
print("end----")
```

```
begin-----
mapping 1
mapped 2
mapping 2
mapped 4
mapping 3
mapped 6
end-----
```



Optional的map和flatMap

```
var num1: Int? = 10
// Optional(20)
var num2 = num1.map { $0 * 2 }

var num3: Int? = nil
// nil
var num4 = num3.map { $0 * 2 }
```

```
var fmt = DateFormatter()
fmt.dateFormat = "yyyy-MM-dd"
var str: String? = "2011-09-10"
// old
var date1 = str != nil ? fmt.date(from: str!) : nil
// new
var date2 = str.flatMap(fmt.date)
```

```
var num1: Int? = 10
// Optional(Optional(20))
var num2 = num1.map { Optional.some($0 * 2) }
// Optional(20)
var num3 = num1.flatMap { Optional.some($0 * 2) }
```

```
var score: Int? = 98
// old
var str1 = score != nil ? "score is \(score!)" : "No score"
// new
var str2 = score.map { "score is \($0)" } ?? "No score"
```

```
var num1: Int? = 10
var num2 = (num1 != nil) ? (num1! + 10) : nil
var num3 = num1.map { $0 + 10 }
// num2、num3是等价的
```



Optional的map和flatMap

```
struct Person {  
    var name: String  
    var age: Int  
}  
  
var items = [  
    Person(name: "jack", age: 20),  
    Person(name: "rose", age: 21),  
    Person(name: "kate", age: 22)  
]  
  
// old  
func getPerson1(_ name: String) -> Person? {  
    let index = items.firstIndex { $0.name == name }  
    return index != nil ? items[index!] : nil  
}  
  
// new  
func getPerson2(_ name: String) -> Person? {  
    return items.firstIndex { $0.name == name }.map { items[$0] }  
}
```



Optional的map和flatMap

```
struct Person {  
    var name: String  
    var age: Int  
    init?(json: [String : Any]) {  
        guard let name = json["name"] as? String,  
              let age = json["age"] as? Int else {  
            return nil  
        }  
        self.name = name  
        self.age = age  
    }  
}  
  
var json: Dictionary? = ["name" : "Jack", "age" : 10]  
// old  
var p1 = json != nil ? Person(json!) : nil  
// new  
var p2 = json.flatMap(Person.init)
```



函数式编程 (Functional Programming)

■ 函数式编程 (Functional Programming , 简称FP) 是一种编程范式，也就是如何编写程序的方法论

□ 主要思想：把计算过程尽量分解成一系列可复用函数的调用

□ 主要特征：函数是“第一等公民”

✓ 函数与其他数据类型一样的地位，可以赋值给其他变量，也可以作为函数参数、函数返回值

■ 函数式编程最早出现在LISP语言，绝大部分的现代编程语言也对函数式编程做了不同程度的支持，比如

□ Haskell、JavaScript、Python、**Swift**、Kotlin、Scala等

■ 函数式编程中几个常用的概念

□ Higher-Order Function、Function Currying

□ Functor、Applicative Functor、Monad

■ 参考资料

□ http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

□ <http://www.mokacoding.com/blog/functor-applicative-monads-in-pictures>



FP实践 – 传统写法

```
// 假设要实现以下功能: [(num + 3) * 5 - 1] % 10 / 2
var num = 1
```

```
func add(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }
func sub(_ v1: Int, _ v2: Int) -> Int { v1 - v2 }
func multiple(_ v1: Int, _ v2: Int) -> Int { v1 * v2 }
func divide(_ v1: Int, _ v2: Int) -> Int { v1 / v2 }
func mod(_ v1: Int, _ v2: Int) -> Int { v1 % v2 }
```

```
divide(mod(sub(multiple(add(num, 3), 5), 1), 10), 2)
```



FP实践 – 函数式写法

```
func add(_ v: Int) -> (Int) -> Int { { $0 + v } }
```

```
func sub(_ v: Int) -> (Int) -> Int { { $0 - v } }
```

```
func multiple(_ v: Int) -> (Int) -> Int { { $0 * v } }
```

```
func divide(_ v: Int) -> (Int) -> Int { { $0 / v } }
```

```
func mod(_ v: Int) -> (Int) -> Int { { $0 % v } }
```

```
infix operator >>> : AdditionPrecedence
```

```
func >>><A, B, C>(_ f1: @escaping (A) -> B,  
                      _ f2: @escaping (B) -> C) -> (A) -> C { { f2(f1($0)) } }
```

```
var fn = add(3) >>> multiple(5) >>> sub(1) >>> mod(10) >>> divide(2)  
fn(num)
```



高阶函数 (Higher-Order Function)

- 高阶函数是至少满足下列一个条件的函数:
 - 接受一个或多个函数作为输入 (map、filter、reduce等)
 - 返回一个函数
- FP中到处都是高阶函数

```
func add(_ v: Int) -> (Int) -> Int { { $0 + v } }
```

柯里化 (Currying)

- 什么是柯里化？
- 将一个接受多参数的函数变换为一系列只接受单个参数的函数

```
func add(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }  
add(10, 20)
```

↓ 柯里化

```
func add(_ v: Int) -> (Int) -> Int { { $0 + v } }  
add(10)(20)
```

- `Array`、`Optional`的`map`方法接收的参数就是一个柯里化函数



柯里化 (Currying)

```
func add1(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }
func add2(_ v1: Int, _ v2: Int, _ v3: Int) -> Int { v1 + v2 + v3 }
```

```
func currying<A, B, C>(_ fn: @escaping (A, B) -> C)
    -> (B) -> (A) -> C {
    { b in { a in fn(a, b) } }
}
```

```
func currying<A, B, C, D>(_ fn: @escaping (A, B, C) -> D)
    -> (C) -> (B) -> (A) -> D {
    { c in { b in { a in fn(a, b, c) } } }
```

```
let curriedAdd1 = currying(add1)
print(curriedAdd1(10)(20))
let curriedAdd2 = currying(add2)
print(curriedAdd2(10)(20)(30))
```



柯里化 (Currying)

```
func add(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }
func sub(_ v1: Int, _ v2: Int) -> Int { v1 - v2 }
func multiple(_ v1: Int, _ v2: Int) -> Int { v1 * v2 }
func divide(_ v1: Int, _ v2: Int) -> Int { v1 / v2 }
func mod(_ v1: Int, _ v2: Int) -> Int { v1 % v2 }
```

```
prefix func ~<A, B, C>(_ fn: @escaping (A, B) -> C)
-> (B) -> (A) -> C { { b in { a in fn(a, b) } } }
```

```
infix operator >>> : AdditionPrecedence
func >>><A, B, C>(_ f1: @escaping (A) -> B,
                     _ f2: @escaping (B) -> C) -> (A) -> C { { f2(f1($0)) } }
```

```
var num = 1
var fn = (~add)(3) >>> (~multiple)(5) >>> (~sub)(1) >>> (~mod)(10) >>> (~divide)(2)
fn(num)
```



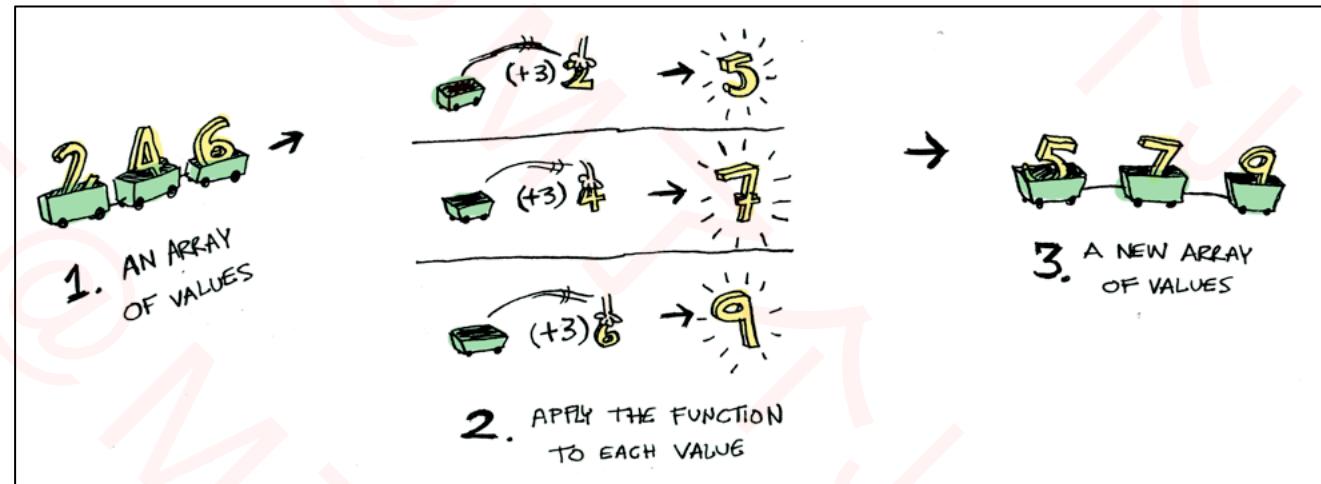
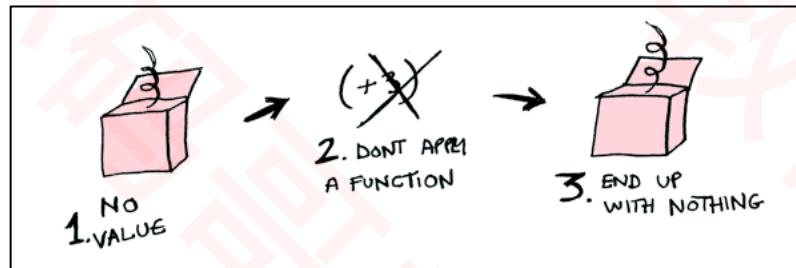
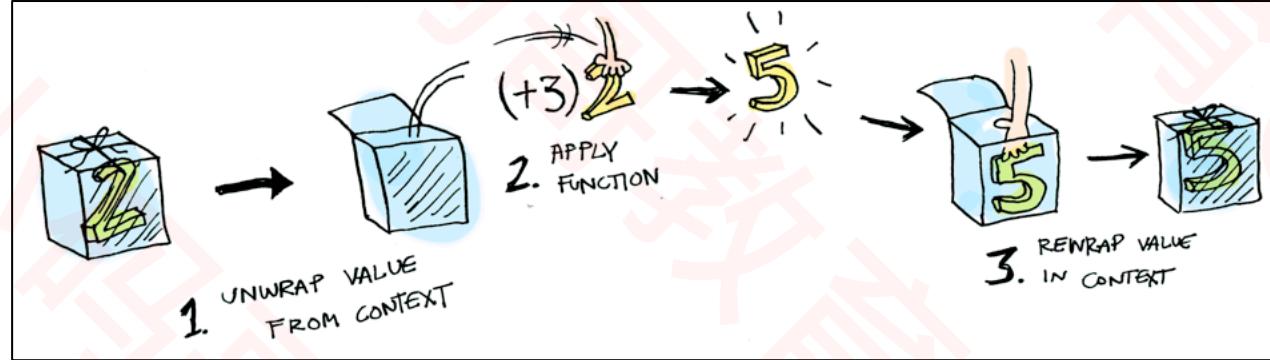
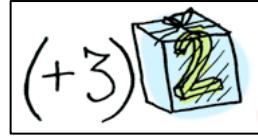
函子 (Functor)

- 像Array、Optional这样支持map运算的类型，称为函子 (Functor)

```
// Array<Element>
public func map<T>(_ transform: (Element) -> T) -> Array<T>
```

```
// Optional<Wrapped>
public func map<U>(_ transform: (Wrapped) -> U) -> Optional<U>
```

函子 (Functor)



适用函子 (Applicative Functor)

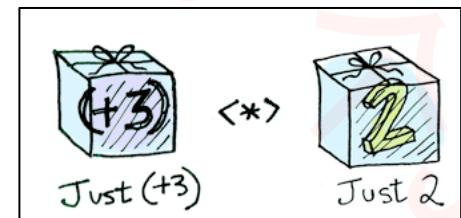
- 对任意一个函子 F，如果能支持以下运算，该函子就是一个适用函子

```
func pure<A>(_ value: A) -> F<A>
func <*><A, B>(fn: F<(A) -> B>, value: F<A>) -> F<B>
```

- `Optional`可以成为适用函子

```
func pure<A>(_ value: A) -> A? { value }
infix operator <*> : AdditionPrecedence
func <*><A, B>(fn: ((A) -> B)?, value: A?) -> B? {
    guard let f = fn, let v = value else { return nil }
    return f(v)
}
```

```
var value: Int? = 10
var fn: ((Int) -> Int)? = { $0 * 2}
// Optional(20)
print(fn <*> value as Any)
```





适用函子 (Applicative Functor)

■ Array可以成为适用函子

```
func pure<A>(_ value: A) -> [A] { [value] }
func <*><A, B>(fn: [(A) -> B], value: [A]) -> [B] {
    var arr: [B] = []
    if fn.count == value.count {
        for i in fn.startIndex..<fn.endIndex {
            arr.append(fn[i](value[i]))
        }
    }
    return arr
}
```

```
// [10]
print(pure(10))

var arr = [{ $0 * 2}, { $0 + 10 }, { $0 - 5 }] <*> [1, 2, 3]
// [2, 12, -2]
print(arr)
```



单子 (Monad)

- 对任意一个类型 F ，如果能支持以下运算，那么就可以称为是一个单子 (Monad)

```
func pure<A>(_ value: A) -> F<A>
func flatMap<A, B>(_ value: F<A>, _ fn: (A) -> F<B>) -> F<B>
```

- 很显然，`Array`、`Optional`都是单子

面向协议编程

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松



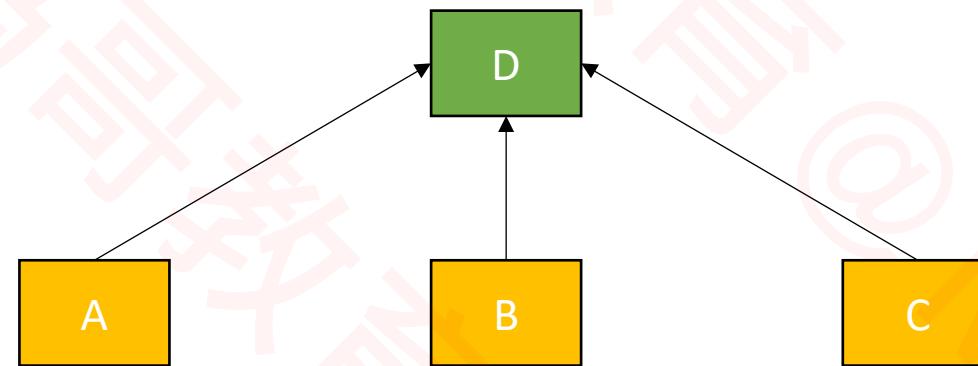


面向协议编程

- 面向协议编程 (Protocol Oriented Programming , 简称POP)
 - 是Swift的一种编程范式 , Apple于2015年WWDC提出
 - 在Swift的标准库中 , 能见到大量POP的影子
- 同时 , Swift也是一门面向对象的编程语言 (Object Oriented Programming , 简称OOP)
 - 在Swift开发中 , OOP和POP是相辅相成的 , 任何一方并不能取代另一方
- POP能弥补OOP一些设计上的不足

回顾OOP

- OOP的三大特性：封装、**继承**、多态
- 继承的经典使用场合
- 当多个类（比如A、B、C类）具有很多共性时，可以将这些共性抽取到一个父类中（比如D类），最后A、B、C类继承D类



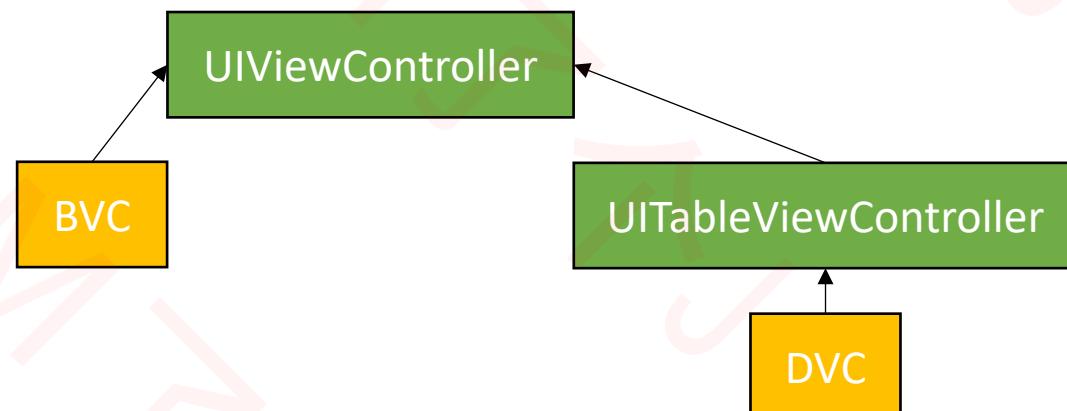
OOP的不足

- 但有些问题，使用OOP并不能很好解决，比如
- 如何将 BVC、DVC 的公共方法 `run` 抽取出来？

```
class BVC: UIViewController {  
    func run() {  
        print("run")  
    }  
}
```

```
class DVC: UITableViewController {  
    func run() {  
        print("run")  
    }  
}
```

- 基于OOP想到的一些解决方案？
 1. 将 `run` 方法放到另一个对象A中，然后BVC、DVC拥有对象A属性
 - 多了一些额外的依赖关系
 2. 将 `run` 方法增加到 `UIViewController` 分类中
 - `UIViewController` 会越来越臃肿，而且会影响它的其他所有子类
 3. 将 `run` 方法抽取到新的父类，采用多继承？(C++ 支持多继承)
 - 会增加程序设计复杂度，产生菱形继承等问题，需要开发者额外解决



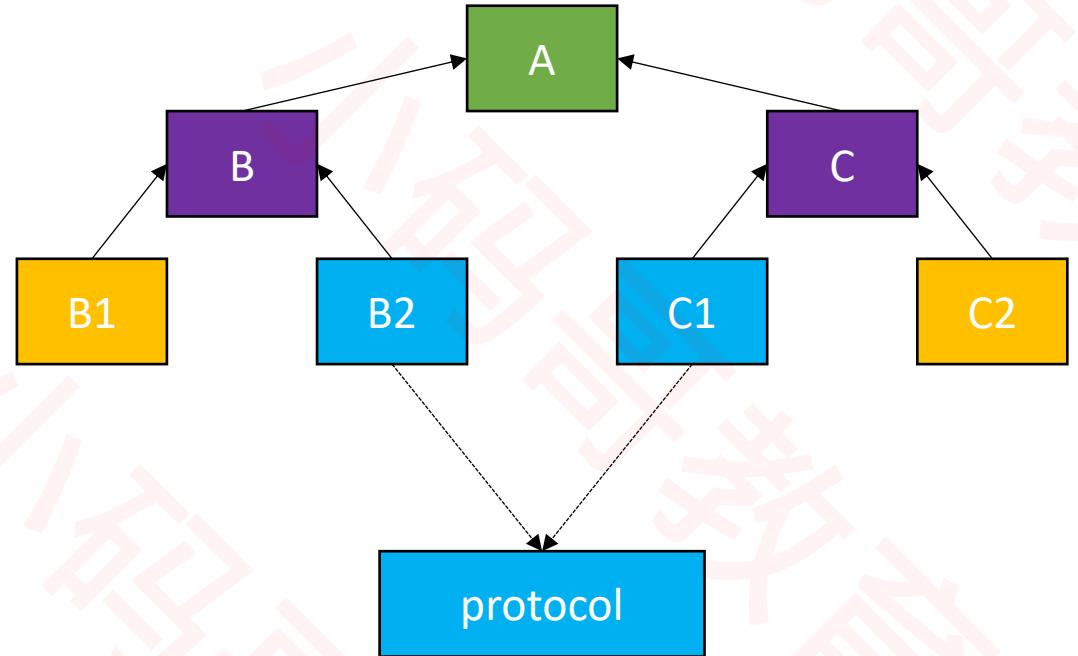


POP的解决方案

```
protocol Runnable {  
    func run()  
}  
  
extension Runnable {  
    func run() {  
        print("run")  
    }  
}
```

```
class BVC: UIViewController, Runnable {}  
class DVC: UITableViewController, Runnable {}
```

再举例





POP的注意点

- 优先考虑创建协议，而不是父类（基类）
- 优先考虑值类型（ struct、enum ），而不是引用类型（ class ）
- 巧用协议的扩展功能
- 不要为了面向协议而使用协议



利用协议实现前缀效果

```
var string = "123fdsf434"  
print(string.mj.numberCount())
```

```
struct MJ<Base> {  
    let base: Base  
    init(_ base: Base) {  
        self.base = base  
    }  
}
```

```
protocol MJCompatible {}  
extension MJCompatible {  
    static var mj: MJ<Self>.Type {  
        get { MJ<Self>.self }  
        set {}  
    }  
    var mj: MJ<Self> {  
        get { MJ(self) }  
        set {}  
    }  
}
```

```
extension String: MJCompatible {}  
extension MJ where Base == String {  
    func numberCount() -> Int {  
        var count = 0  
        for c in base where ("0"..."9").contains(c) {  
            count += 1  
        }  
        return count  
    }  
}
```



Base: 类

```
class Person {}
class Student: Person {}

extension Person: MJCompatible {}
extension MJ where Base: Person {
    func run() {}
    static func test() {}
}
```

```
Person.mj.test()
Student.mj.test()

let p = Person()
p.mj.run()

let s = Student()
s.mj.run()
```



Base: 协议

```
var s1: String = "123fd434"
var s2: NSString = "123fd434"
var s3: NSMutableString = "123fd434"
print(s1.mj.numberCount())
print(s2.mj.numberCount())
print(s3.mj.numberCount())
```

```
extension String: MJCompatible {}
extension NSString: MJCompatible {}
extension MJ where Base: ExpressibleByStringLiteral {
    func numberCount() -> Int {
        let string = base as! String
        var count = 0
        for c in string where ("0"..."9").contains(c) {
            count += 1
        }
        return count
    }
}
```



利用协议实现类型判断

```
func isArray(_ value: Any) -> Bool { value is [Any] }
isArray( [1, 2] )
isArray( ["1", 2] )
isArray( NSArray() )
isArray( NSMutableArray() )
```

```
protocol ArrayType {}
extension Array: ArrayType {}
extension NSArray: ArrayType {}
func isArrayType(_ type: Any.Type) -> Bool { type is ArrayType.Type }
isArrayType([Int].self)
isArrayType([Any].self)
isArrayType(NSArray.self)
isArrayType(NSMutableArray.self)
```

响应式编程

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>



实力IT教育 www.520it.com

码拉松





响应式编程

- 响应式编程 (Reactive Programming , 简称RP)
 - 也是一种编程范式，于1997年提出，可以简化异步编程，提供更优雅的数据绑定
 - 一般与函数式融合在一起，所以也会叫做：函数响应式编程 (Functional Reactive Programming , 简称FRP)
- 比较著名的、成熟的响应式框架
 - ReactiveCocoa
 - ✓ 简称RAC，有Objective-C、Swift版本
 - ✓ 官网：<http://reactivecocoa.io/>
 - ✓ github：<https://github.com/ReactiveCocoa>
 - ReactiveX
 - ✓ 简称Rx，有众多编程语言的版本，比如RxJava、RxKotlin、RxJS、RxCpp、RxPHP、RxGo、**RxSwift**等等
 - ✓ 官网：<http://reactivex.io/>
 - ✓ github：<https://github.com/ReactiveX>



RxSwift

- RxSwift (ReactiveX for Swift) , ReactiveX的Swift版本
- 源码 : <https://github.com/ReactiveX/RxSwift>
- 中文文档 : <https://beeth0ven.github.io/RxSwift-Chinese-Documentation/>
- RxSwift的github上已经有详细的安装教程，这里只演示CocoaPods方式的安装

① Podfile

```
use_frameworks!

target 'target_name' do
    pod 'RxSwift', '~> 5'
    pod 'RxCocoa', '~> 5'
end
```

② 命令行

- pod repo update
- pod install

③ 导入模块

```
import RxSwift
import RxCocoa
```

■ 模块说明

- RxSwift : Rx标准API的Swift实现，不包括任何iOS相关的内容
- RxCocoa : 基于RxSwift，给iOS UI控件扩展了很多Rx特性

RxSwift的核心角色

- Observable：负责发送事件（Event）
- Observer：负责订阅Observable，监听Observable发送的事件（Event）



```
public enum Event<Element> {  
    /// Next element is produced.  
    case next(Element)  
  
    /// Sequence terminated with an error.  
    case error(Swift.Error)  
  
    /// Sequence completed successfully.  
    case completed  
}
```

- Event有3种
 - next：携带具体数据
 - error：携带错误信息，表明Observable终止，不会再发出事件
 - completed：表明Observable终止，不会再发出事件



创建、订阅Observable1

```
var observable = Observable<Int>.create { observer in
    observer.onNext(1)
    observer.onCompleted()
    return Disposables.create()
}

// 等价于
observable = Observable.just(1)
observable = Observable.of(1)
observable = Observable.from([1])
```

```
var observable = Observable<Int>.create { observer in
    observer.onNext(1)
    observer.onNext(2)
    observer.onNext(3)
    observer.onCompleted()
    return Disposables.create()
}

// 等价于
observable = Observable.of(1, 2, 3)
observable = Observable.from([1, 2, 3])
```

```
observable.subscribe { event in
    print(event)
}.dispose()
```

```
observable.subscribe(onNext: {
    print("next", $0)
}, onError: {
    print("error", $0)
}, onCompleted: {
    print("completed")
}, onDisposed: {
    print("dispose")
}).dispose()
```



创建、订阅Observable2

```
let observable = Observable<Int>.timer(.seconds(3),  
                                      period: .seconds(1),  
                                      scheduler: MainScheduler.instance)  
  
observable.map { "数值是 $($0)" }  
    .bind(to: label.rx.text)  
    .disposed(by: bag)
```



创建Observer

```
let observer = AnyObserver<Int>.init { event in
    switch event {
        case .next(let data):
            print(data)
        case .completed:
            print("completed")
        case .error(let error):
            print("error", error)
    }
}
Observable.just(1).subscribe(observer).dispose()
```

```
let binder = Binder<String>(label) { label, text in
    label.text = text
}
Observable.just(1).map { "数值是\($0)" }.subscribe(binder).dispose()
Observable.just(1).map { "数值是\($0)" }.bind(to: binder).dispose()
```



扩展Binder属性

```
extension Reactive where Base: UIView {  
    var hidden: Binder<Bool> {  
        Binder<Bool>(base) { view, value in  
            view.isHidden = value  
        }  
    }  
}
```

```
let observable = Observable<Int>.interval(.seconds(1),  
                                         scheduler: MainScheduler.instance)  
observable.map { $0 % 2 == 0 }.bind(to: button.rx.hidden).disposed(by: bag)
```



传统状态监听

- 在开发中经常要对各种状态进行监听，传统的常见监听方案有
 - KVO
 - Target-Action
 - Notification
 - Delegate
 - Block Callback
- 传统方案经常会出现错综复杂的依赖关系、耦合性较高，还需要编写重复的非业务代码



RxSwift的状态监听1

```
button.rx.tap.subscribe(onNext: {
    print("按钮被点击了1")
}).disposed(by: bag)
```

```
let data = Observable.just([
    Person(name: "Jack", age: 10),
    Person(name: "Rose", age: 20)
])
data.bind(to: tableView.rx.items(cellIdentifier: "cell")) { row, person, cell in
    cell.textLabel?.text = person.name
    cell.detailTextLabel?.text = "\(person.age)"
}.disposed(by: bag)

tableView.rx.modelSelected(Person.self)
    .subscribe(onNext: { person in
        print("点击了", person.name)
    }).disposed(by: bag)
```



RxSwift的状态监听2

```
class Dog: NSObject {
    @objc dynamic var name: String?
}

dog.rx.observe(String.self, "name")
    .subscribe(onNext: { name in
        print("name is", name ?? "nil")
    }).disposed(by: bag)
dog.name = "larry"
dog.name = "wangwang"
```

```
NotificationCenter.default.rx
    .notification(UIApplication.didEnterBackgroundNotification)
    .subscribe(onNext: { notification in
        print("APP进入后台", notification)
    }).disposed(by: bag)
```



既是Observable，又是Observer

```
Observable.just(0.8).bind(to: slider.rx.value).dispose()
```

```
slider.rx.value.map {  
    "当前数值是: \"\$0\""  
}.bind(to: textField.rx.text).disposed(by: bag)
```

```
textField.rx.text  
.subscribe(onNext: { text in  
    print("text is", text ?? "nil")  
}).disposed(by: bag)
```

- 诸如`UISlider.rx.value`、`UITextField.rx.text`这类属性值，既是Observable，又是Observer
- 它们是`RxCocoa.ControlProperty`类型



Disposable

- 每当Observable被订阅时，都会返回一个Disposable实例，当调用Disposable的dispose，就相当于取消订阅
- 在不需要再接收事件时，建议取消订阅，释放资源。有3种常见方式取消订阅

```
// 立即取消订阅（一次性订阅）
observable.subscribe { event in
    print(event)
}.dispose()
```

```
// 当bag销毁（deinit）时，会自动调用Disposable实例的dispose
observable.subscribe { event in
    print(event)
}.disposed(by: bag)
```

```
// self销毁时（deinit）时，会自动调用Disposable实例的dispose
let _ = observable.takeUntil(self.rx.deallocated).subscribe { event in
    print(event)
}
```