

从OC到Swift

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>

码拉松



实力IT教育 www.520it.com

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  
}
```

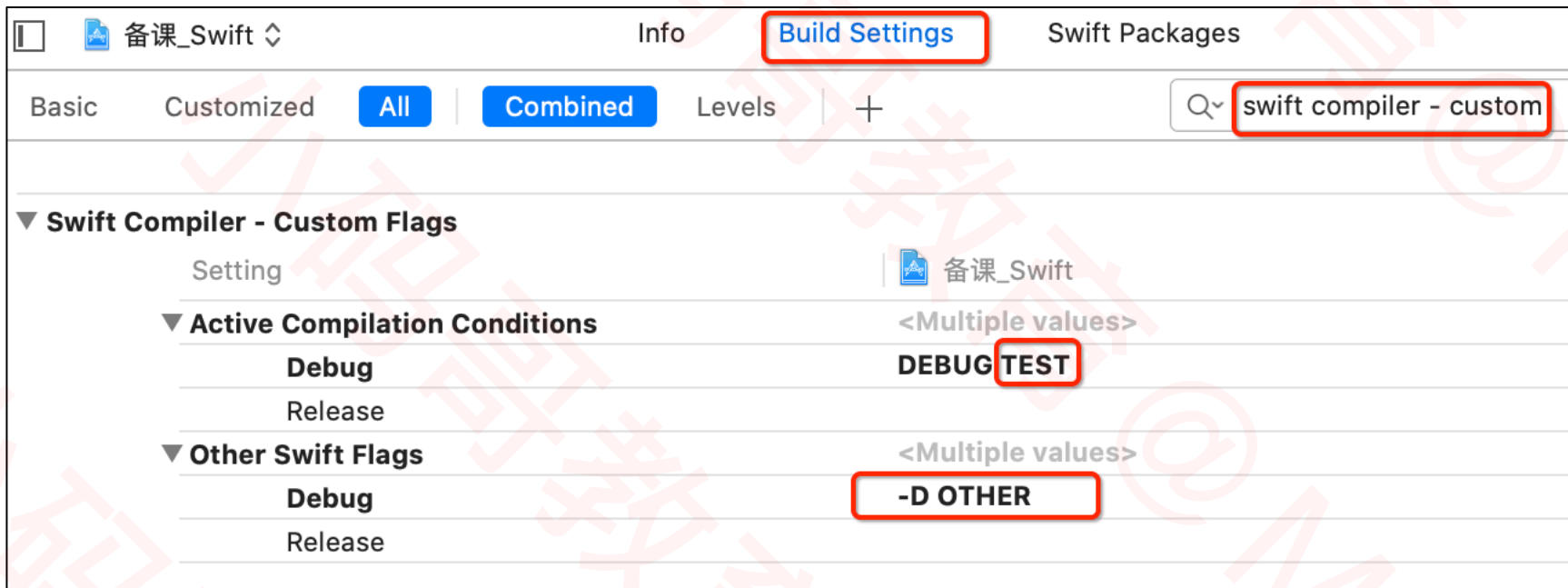


```
public class Person {  
    // MARK: - 属性  
    var age = 0  
    var weight = 0  
    var height = 0  
  
    // MARK: - 私有方法  
    // MARK: 跑步  
    private func run1() {}  
    private func run2() {}  
    // MARK: 走路  
    private func walk1() {}  
    private func walk2() {}  
  
    // MARK: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



```
// 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  
}
```

```
@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))
```


- 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(@"%zd %@ -run", _age, _name); }

+ (void)eat:(NSString *)food other:(NSString *)other { NSLog(@"Person +eat %@ %@", food, other); }
- (void)eat:(NSString *)food other:(NSString *)other { NSLog(@"%zd %@ -eat %@ %@", _age, _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
```

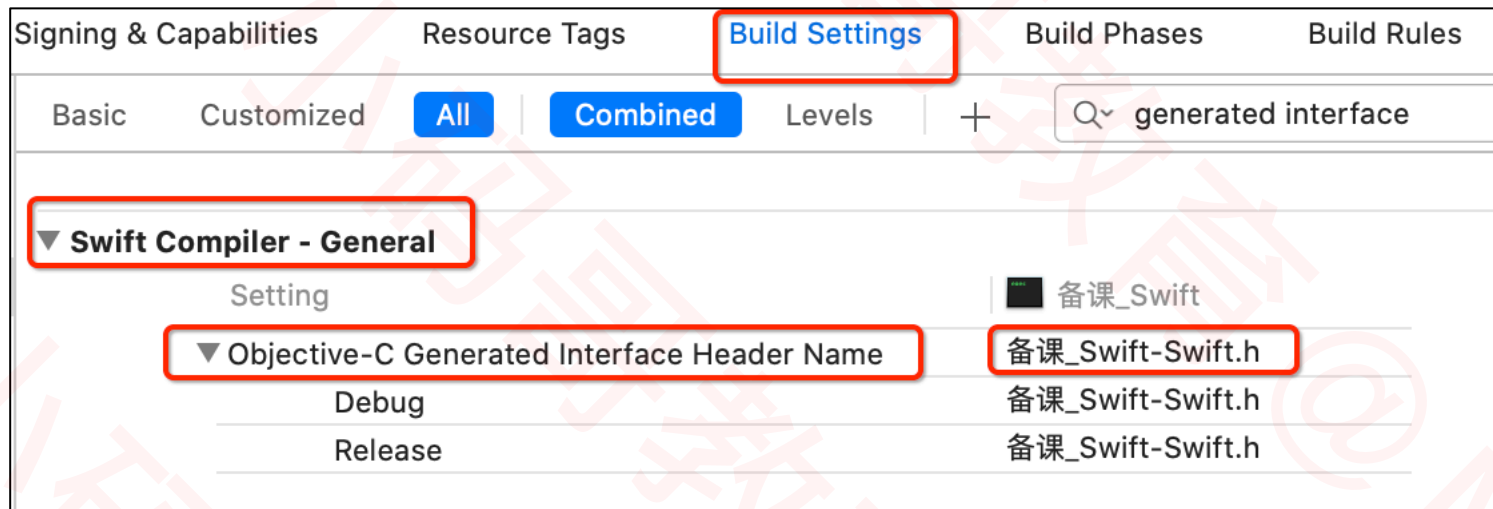
Swift调用OC – @_silgen_name

- 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
```

- 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**



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;
- (nonnull instancetype)initWithPrice:(double)price band:(NSString * _Nonnull)band OBJC_DESIGNATED_INITIALIZER;
- (void)run;
+ (void)run;
- (nonnull instancetype)init SWIFT_UNAVAILABLE;
+ (nonnull instancetype)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

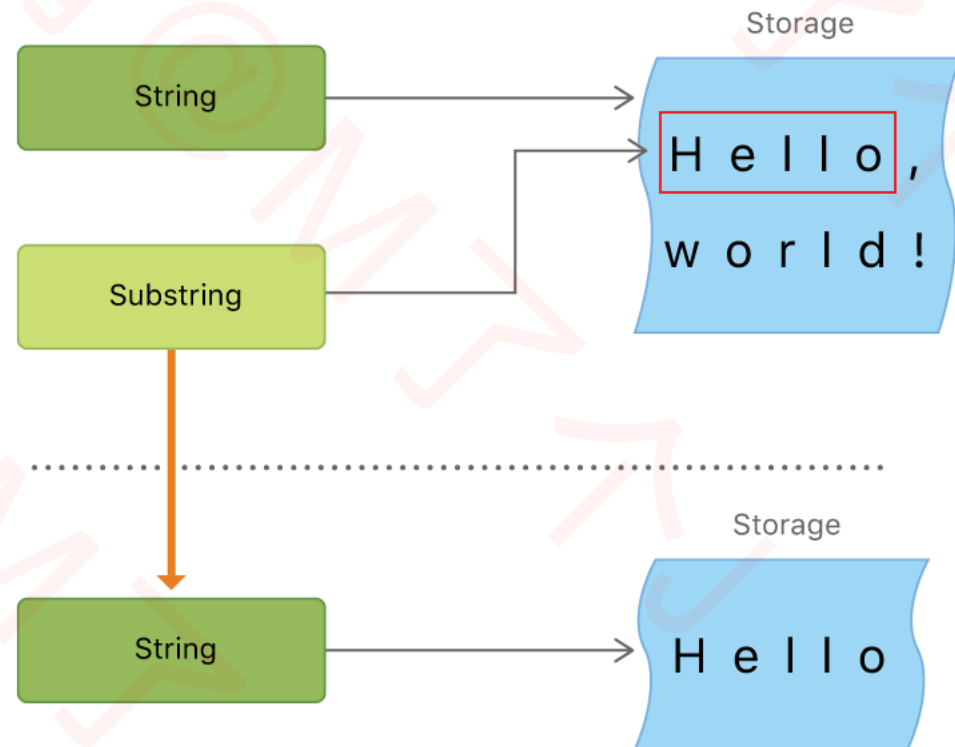
```
var str = "1_2"  
// 1_2_  
str.insert("_", at: str.endIndex)  
// 1_2_3_4  
str.insert(contentsOf: "3_4", at: str.endIndex)  
// 1666_2_3_4  
str.insert(contentsOf: "666", at: str.index(after: str.startIndex))  
// 1666_2_3_8884  
str.insert(contentsOf: "888", at: str.index(before: str.endIndex))  
// 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)..  
str.index(before: str.endIndex)  
// hello_2_3_4  
str.removeSubrange(range)
```

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..  
var substr3 = str[range]  
  
// 最初的String, 1_2_3_4_5  
print(substr3.base)  
  
// Substring -> String  
var str2 = String(substr3)
```



- `Substring`和它的`base`，共享字符串数据
- `Substring`转为`String`时，会重新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'
""
```

```
1
    "2"
3""
    '4'
```

```
// 以下2个字符串是等价的
let str1 = "These are the same."
let str2 = ""
These are the same.
""
```

```
// 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 \"\"\"
""
```

```
Escaping the first quote ""
Escaping two quotes ""
Escaping all three quotes ""
```

```
// 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	⇒	NSString
String	←	NSMutableString
Array	⇒	NSArray
Array	←	NSMutableArray
Dictionary	⇒	NSDictionary
Dictionary	←	NSMutableDictionary
Set	⇒	NSSet
Set	←	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) {  
            (_, 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
```

```
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

```
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>

```
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 - 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

```
let age1: Int = {
    print(666)
    return 10
}()

class ViewController: UIViewController {
    static let age2: Int = {
        print(888)
        return 20
    }()
    override func viewDidLoad() {
        super.viewDidLoad()
        print(age1)
        print(age1)
        // 666 10 10

        print(ViewController.age2)
        print(ViewController.age2)
        // 888 20 20
    }
}
```