触控科技

Using Swift with Cocoa and Objective-C 完整中文版（CocoaChina 精校）

CocoaChina

2014-6-12

Introducing Swift.

本教程系列由 CocoaChina 整理自 CocoaChina 的 github，禁止商用，特此声明。

Swift 开发语言讨论区：

http://www.cocoachina.com/bbs/thread.php?fid=57

Swift 专题：http://www.cocoachina.com/special/swift/

微博：@CocoaChina

微信：CocoaChina

全文校对：ChildhoodAndy

整理人：齐楠楠

目录

1.开始

1.1.基本设置

本篇译者：Creolophus（github 主页），敬请勘误。

重要事项

这篇文章初步介绍了在开发中用到的 API 或技术。苹果公司提供这些信息来帮助您规划本文所说明的技术和接口以用于苹果的产品上。这些信息会改变，并且根据这篇文章所实现的软件应该在最新的操作系统并根据最新的文档测试。本文档的新版本，可能在未来通过技术和 API 的 seeds 版本来提供

Swift 被设计用来无缝兼容 Cocoa 和 Objective-C 。在 Swift 中，你可以使用 Objective-C 的 API（包括系统框架和你自定义的代码），你也可以在 Objective-C 中 使用 Swift 的 API。这种兼容性使 Swift 变成了一个简单、方便并且强大的工具集成到你的 Cocoa 应用开发工作流程中。

这篇指南包括了三个有关兼容性的重要方面方便你更好地利用来开发 Cocoa 应用：

- **互用性** 使你将 Swift 和 Objective-C 相接合，允许在 Objective-C 中使用 Swift 的 Class 并且当你在写 Swift 代码时利用熟悉的 Cocoa Class、Pattern、Practice。
- **混合和匹配** 允许你创建结合了 Swift 和 Objective-C 文件的混合语言应用，他们能更彼此进行通信。
- **迁移** 由于以上两点，从已经存在的 Objective-C 代码迁移到 Swift 是非常简单的，使得用最新的 Swift 特性代替你的 Objective-C 应用部分内容成为了可能。

在你开始学习这些特性前，你需要对如何建立 Swift 环境来访问 Cocoa 系统框架有个大体了解。

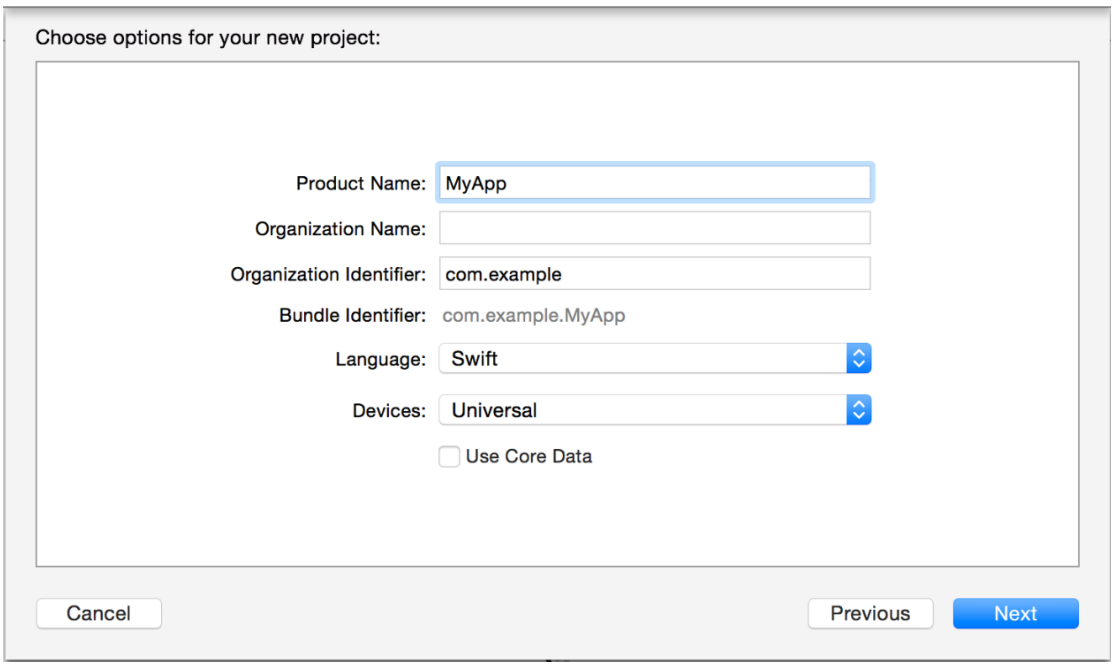
建立你的 Swift 环境

为了开始体验在 Swift 中访问 Cocoa 框架，使用 Xcode 的一个模板来创建一个基于 Swift 应用。

在 Xcode 中创建一个 Swift 项目

1.选择 File > New > Project > (iOS or OS X) > Application > your *template of choice*。

2.点击 Language 弹出菜单并选择 Swift。



Swift 项目的结构几乎和 Objective-C 项目一模一样，只有一个重要的区别：Swift 没有头文件。在实现和接口之间没有显示的划分，所以一个特定类中的所有信息都存储在单独的`.swift`文件中。

现在开始，你可以开始体验在`app delegate`中写 Swift 代码，或者你可以通过选择 File > New > File > (iOS or OS X) > Other > Swift 来创建一个 Swift 类。

理解 Swift 导入过程

在你建立 Xcode 项目后，你可以在 Swift 里导入任意用 Objective-C 来工作的 Cocoa 平台框架。

任意 Objective-C 的框架（或 C 类库）将作为一个 *module*，能直接导入到 Swift 中。这些包括了所有 Objective-C 系统框架-比如 Foundation、UIKit 和 SpriteKit，就像系统支持公共 C 类库。举个例子，想导入 Foundation，只要简单地添加 import 语句到你写的 Swift 文件的顶部。

SWIFT

```
import Foundation
```

这个 import 导入了所有 Foundation 的 API，包括 `NSDate`，`NSURL`，`NSMutableData`，并且他们的所有方法、属性和类别都可以在 Swift 中直接使用。

导入过程是非常简洁的。Objective-C 框架在头文件中申明 API。在 Swift 中，那些头文件被编译成 Objective-C 的 module，接着被导入到 Swift 作为 Swift 的 API。导入决定了 Objective-C 的 Function，Class，Method 和 Type 如何在 Swift 中出现。对于 Function 和 Method，这个过程影响他们的参数和返回值。导入过程可以做下面这些事情：

- 重映射确定的 Objective-C 类型到 Swift 中的同等类型，就像 id 到 `AnyObject`
- 重映射确定的 Objective-C 核心类型到 Swift 中的替代类型， 就像 `NSString` 到 `String`
- 重映射确定的 Objective-C 概念到 Swift 中相匹配的概念，如 `pointers` 到 `optionals`

在互用性章节，你将会了解到更多关于这些映射如何在你的 Swift 代码进行取舍。 导入 Swift 的模型到 Objective-C 和从 Objective-C 导入 Swift 是非常相似的。Swift 申明它的 API，比如一个框架作为 Swift modules。同时这些 Swift modules 会生成 Objective-C 的头文件。这些头文件可以映射回 Objective-C 的 API 中。一些 Swift 的 API 不映射回 Objective-C 因为他们取舍了语言特性发现这些在 Objective-C 中不可用。关于在 Objective-C 中使用 Swift 的更多特性，请参看在同一项目中使用 Swift 和 Objective-C。

注意

你不能直接把 C++ 代码导入 Swift。解决办法是为 C++ 代码创建一个 Objective-C 或者 C 的封装。

2.互用性

2.1. 与 Objective-C API 交互

本篇译者：@Creolophus （git 主页），敬请勘误。

互用性是让 Swift 和 Objective-C 相接合的一种特性，使你能够在一种语言编写的文件中使用另一种语言。当你准备开始把 Swift 融入到你的开发流程中时，你应该懂得如何利用互用性来重新定义并提高你写 Cocoa 应用的方案。

互用性很重要的一点就是允许你在写 Swift 代码时使用 Objective-C 的 API 接口。当你导入一个 Objective-C 框架后，你可以使用原生的 Swift 语法实例化它的 Class 并且与之交互。

初始化(Initialization)

为了使用 Swift 实例化 Objective-C 的 Class，你应该使用 Swift 语法调用它的一个初始化器。当 Objective-C 的 `init` 方法变化到 Swift，他们用 Swift 初始化语法呈现。“init”前缀被截断当作一个关键字，用来表明该方法是初始化方法。那些以“initWith”开头的 `init` 方法，“With”也会被去除。从“init”或者“initWith”中分离出来的这部分方法名首字母变成小写，并且被当做是第一个参数的参数名。其余的每一部分方法名依次变味参数名。这些方法名都在圆括号中被调用。

举个例子，你在使用 Objective-C 时会这样做：

```
//Objective-C
UITableView *myTableView = [[UITableView alloc]
initWithFrame:CGRectZero style:UITableViewStyleGrouped];
```

在 Swift 中，你应该这样做：

```
//Swift
let myTableView: UITableView = UITableView(frame: CGRectZero, style: .Grouped)
```

你不需要调用 alloc，Swift 能替你处理。注意，当使用 Swift 风格的初始化函数的时候，“init”不会出现。

你可以在初始化时显式的声明对象的类型，也可以忽略它，Swift 能够正确判断对象的类型。

```
//Swift
let myTextField = UITextField(frame: CGRect(0.0, 0.0, 200.0, 40.0))
```

这里的 `UITableView` 和 `UITextField` 对象和你在 Objective-C 中使用的具有相同的功能。你可以用一样的方式使用他们，包括访问属性或者调用各自的类中的方法。

为了统一和简易，Objective-C 的工厂方法也在 Swift 中映射为方便的初始化方法。这种映射能够让他们使用同样简洁明了的初始化方法。例如，在 Objective-C 中你可能会像下面这样调用一个工厂方法：

```
//Objective-C
UIColor *color = [UIColor colorWithRed:0.5 green:0.0 blue:0.5 alpha:1.0];
```

在 Swift 中，你应该这样做：

```
//Swift
let color = UIColor(red: 0.5, green: 0.0, blue: 0.5, alpha: 1.0)
```

访问属性(Accessing Properties)

在 Swift 中访问和设置 Objective-C 对象的属性时，使用点语法

```
///Swift
myTextField.textColor = UIColor.darkGrayColor()
myTextField.text = "Hello world"
if myTextField.editing {
```

```
    myTextField.editing = false
}
```

当 get 或 set 属性时，直接使用属性名称，不需要附加圆括号。注意，`darkGrayColor`后面附加了一对圆括号，这是因为`darkGrayColor`是`UIColor`的一个类方法，不是一个属性。

在 Objective-C 中，一个有返回值的无参数方法可以被作为一个隐式的访问函数，并且可以与访问器使用同样的方法调用。但在 Swift 中不再能够这样做了，只有使用`@property`关键字声明的属性才会被作为属性引入。

方法(Working with Methods)

在 Swift 中调用 Objective-C 方法时，使用点语法。

当 Objective-C 方法转换到 Swift 时，Objective-C 的`selector`的第一部分将会成为方法名并出现在圆括号的前面，而第一个参数将直接在括号中出现，并且没有参数名，而剩下的参数名与参数则一一对应的填入圆括号中。

举个例子，你在使用 Objective-C 时会这样做：

```
//Objective-C
[myTableView insertSubview:mySubview atIndex:2];
```

在 Swift 中，你应该这样做：

```
//Swift
myTableView.insertSubview(mySubview, atIndex: 2)
```

如果你调用一个无参方法，仍必须在方法名后面加上一对圆括号

```
//Swift
myTableView.layoutIfNeeded()
```

id 兼容性(id Compatibility)

Swift 包含一个叫做 `AnyObject` 的协议类型，表示任意类型的对象，就像 Objective-C 中的 `id` 一样。`AnyObject` 协议允许你编写类型安全的 Swift 代码同时维持无类型对象的灵活性。因为 `AnyObject` 协议保证了这种安全，Swift 将 id 对象导入为 AnyObject。

举个例子，跟 id 一样，你可以为 `AnyObject` 类型的对象分配任何其他类型的对象，你也同样可以为它重新分配其他类型的对象。

```
//Swift
var myObject: AnyObject = UITableViewCell()
myObject = NSDate()
```

你也可以在调用 Objective-C 方法或者访问属性时不将它转换为具体类的类型。这包括了 Objcive-C 中标记为 @objc 的方法。

```
//Swift
let futureDate = myObject.dateByAddingTimeInterval(10)
let timeSinceNow = myObject.timeIntervalSinceNow
```

然而，由于直到运行时才知道 `AnyObject` 的对象类型，所以有可能在不经意间写出不安全代码。另外，与 Objective-C 不同的是，如果你调用方法或者访问的属性 AnyObject 对象没有声明，将会报运行时错误。比如下面的代码在运行时将会报出一个 unrecognized selector error 错误：

```
//Swift
myObject.characterAtIndex(5)
// crash, myObject does't respond to that method
```

但是，你可以通过 Swift 的 optinals 特性来排除这个 Objective-C 中常见的错误，当你用 `AnyObject` 对象调用一个 Objective-C 方法时，这次调用将会变成一次隐式展开 optional（implicitly unwrapped optional）的行为。你可以通过 optional 特性来决定 AnyObject 类型的对象是否调用该方法，同样的，你可以把这种特性应用在属性上。

举个例子，在下面的代码中，第一和第二行代码将不会被执行因为 `length` 属性和 `characterAtIndex:` 方法不存在于 NSDate 对象中。`myLength` 常量会被推测成可选的

`Int`类型并且被赋值为`nil`。同样你可以使用`if-let`声明来有条件的展开这个方法的返回值，从而判断对象是否能执行这个方法。就像第三行做的一样。

```
//Swift
let myLength = myObject.length?
let myChar = myObject.characterAtIndex?(5)
if let fifthCharacter = myObject.characterAtIndex(5) {
    println("Found \(fifthCharacter) at index 5")
}
```

对于 Swift 中的强制类型转换，从 AnyObject 类型的对象转换成明确的类型并不会保证成功，所以它会返回一个可选的值。而你需通过检查该值的类型来确认转换是否成功。

```
//Swift
let userDefaults = NSUserDefaults.standardUserDefaults()
let lastRefreshDate: AnyObject? = userDefaults.objectForKey("LastRefreshDate")
if let date = lastRefreshDate as? NSDate {
    println("\(date.timeIntervalSinceReferenceDate)")
}
```

当然，如果你能确定这个对象的类型（并且确定不是`nil`），你可以添加`as`操作符强制调用。

```
//Swift
let myDate = lastRefreshDate as NSDate
let timeInterval = myDate.timeIntervalSinceReferenceDate
```

使用 nil（Working with nil）

在 Objective-C 中，对象的引用可以是值为`NULL`的原始指针（同样也是 Objective-C 中的`nil`）。而在 Swift 中，所有的值–包括结构体与对象的引用–都被保证为非空。作为替代，你将这个可以为空的值包装为 optional type。当你需要宣告值为空时，你需要使用`nil`。你可以在 Optionals 中了解更多。

因为 Objective-C 不会保证一个对象的值是否非空，Swift 在引入 Objective-C 的 API 的时候，确保了所有函数的返回类型与参数类型都是 optional，在你使用 Objective-C 的 API 之前，你应该检查并保证该值非空。 在某些情况下，你可能绝

对确认某些 Objective-C 方法或者属性永远不应该返回一个 nil 的对象引用。为了让对象在这种情况下更加易用，Swift 使用 implicitly unwrapped optionals 方法引入对象， implicitly unwrapped optionals 包含 optional 类型的所有安全特性。此外，你可以直接访问对象的值而无需检查 nil。当你访问这种类型的变量时，implicitly unwrapped optional 首先检查这个对象的值是否不存在，如果不存在，将会抛出运行时错误。

扩展（Extensions）

Swift 的扩展和 Objective-C 的类别（Category）相似。扩展为原有的类，结构和枚举丰富了功能，包括在 Objective-C 中定义过的。你可以为系统的框架或者你自己的类型增加扩展，只需要导入合适的模块并且保证你在 Objective-C 中使用的类、结构或枚举拥有相同的名字。

举个例子，你可以扩展 `UIBezierPath` 类来为它增加一个等边三角形，这个方法只需提供三角形的边长与起点。

```
//Swift
extension UIBezierPath {
    convenience init(triangleSideLength: Float, origin: CGPoint) {
        self.init()
        let squareRoot = Float(sqrt(3))
        let altitude = (squareRoot * triangleSideLength) / 2
        moveToPoint(origin)
        addLineToPoint(CGPoint(triangleSideLength, origin.x))
        addLineToPoint(CGPoint(triangleSideLength / 2, altitude))
        closePath()
    }
}
```

你也可以使用扩展来增加属性（包括类的属性与静态属性）。然而，这些属性必须是通过计算才能获取的，扩展不会为类，结构体，枚举存储属性。下面这个例子为 `CGRect` 类增加了一个叫 `area` 的属性。

```
//Swift
extension CGRect {
    var area: CGFloat {
    return width * height
    }
```

```
}
let rect = CGRect(x: 0.0, y: 0.0, width: 10.0, height: 50.0)
let area = rect.area
// area: CGFloat = 500.0
```

你同样可以使用扩展来为类添加协议而无需增加它的子类。如果这个协议是在 Swift 中被定义的，你可以添加 comformance 到它的结构或枚举中无论它们在 Objective-C 或在 Swift 中被定义。

你不能使用扩展来覆盖 Objective-C 类型中存在的方法与属性。

闭包（Closures）

Objective-C 中的 `blocks` 会被自动导入为 Swift 中的闭包。例如，下面是一个 Objective-C 中的 block 变量：

```
//Objective-C
void (^completionBlock)(NSData *, NSError *) = ^(NSData *data, NSError *error)
{/* ... */}
```

而它在 Swift 中的形式为

```
//Swift
let completionBlock: (NSData, NSError) -> Void = {data, error in /* ... */}
```

Swift 的闭包与 Objective-C 中的 blocks 能够和睦相处，所以你可以把一个 Swift 闭包传递给一个把 block 作为参数的 Objective-C 函数。Swift 闭包与函数具有互通的类型，所以你甚至可以传递 Swift 函数的名字。

闭包与 blocks 语义上想通但是在一个地方不同：变量是可以直接改变的，而不是像 block 那样会拷贝变量。换句话说，Swift 中变量的默认行为与 Objective-C 中 __block 变量一致。

比较对象（Object Comparison）

当比较两个 Swift 中的对象时，可以使用两种方式。第一种，使用（==），判断两个对象内容是否相同。第二种，使用(===)，判断常量或者变量是否为同一个对象的实例。

Swift 与 Objective-C 一般使用 == 与 === 操作符来做比较。Swift 的 == 操作符为源自 NSObject 的对象提供了默认的实现。在实现 == 操作符时，Swift 调用 NSObject 定义的 isEqual: 方法。

NSObject 类仅仅做了身份的比较，所以你需要在你自己的类中重新实现 isEqual: 方法。因为你可以直接传递 Swift 对象给 Objective-C 的 API，你也应该为这些对象实现自定义的 isEqual: 方法，如果你希望比较两个对象的内容是否相同而不是仅仅比较他们是不是由相同的对象派生。

作为实现比较函数的一部分，确保根据 Object comparison 实现对象的 `hash` 属性。更进一步的说，如果你希望你的类能够作为字典中的键，也需要遵从 `Hashable` 协议以及实现 `hashValues` 属性。

Swift 类型兼容性（Swift Type Compatibility）

当你定义了一个继承自 `NSObject` 或者其他 Objective-C 类的 Swift 类，这些类都能与 Objective-C 无缝连接。所有的步骤都有 Swift 编译器自动完成，如果你从未在 Objective-C 代码中导入 Swift 类，你也不需要担心类型适配问题。另外一种情况，如果你的 Swift 类并不来源自 Objectve-C 类而且你希望能在 Objecive-C 的代码中使用它，你可以使用下面描述的 @objc 属性。

`@objc` 可以让你的 Swift API 在 Objective-C 中使用。换句话说，你可以通过在任何 Swift 方法、类、属性前添加`@objc`，来使得他们可以在 Objective-C 代码中使用。如果你的类继承自 Objective-C,编译器会自动帮助你完成这一步。编译器还会在所有的变量、方法、属性前加 @objc，如果这个类自己前面加上了`@objc` 关键字。当你使用`@IBOutlet`，`@IBAction`，或者是`@NSManaged` 属性时，`@objc` 也会自动加在前面。这个关键字也可以用在 Objetive-C 中的 target-action 设计模式中，例如，`NSTimer` 或者 `UIButton`。

当你在 Objective-C 中使用 Swift API，编译器基本对语句做直接的翻译。例如，`Swift API func playSong(name: String)`会被解释为`- (void)playSong:(NSString *)name`。然而，有一个例外：当在 Objective-C 中使用 Swift 的初始化函数，编译器会在方法前添加“initWith”并且将原初始化函数的第一个参数首字母大写。例如，这个 Swift 初始化函数 `init (songName: String, artist: String` 将被翻译为`- (instancetype)initWithSongName:(NSString *)songName artist:(NSString *)artist`。Swift 同时也提供了一个`@objc`关键字的变体，通过它你可以自定义在 Objectiv-C 中转换的函数名。例如，如果你的 Swift 类的名字包含 Objecytive-C 中不支持的字符，你就可以为 Objective-C 提供一个可供替代的名字。如果你给 Swift 函数提供一个 Objecytive-C 名字，要记得为带参数的函数添加（:）

```
//Swift
@objc(Squirrel)
class Белка {
    @objc(initWithName:)
    init (имя: String) { /*...*/ }
    @objc(hideNuts:inTree:)
    func прячьОрехи(Int, вДереве: Дерево) { /*...*/ }
}
```

当你在 Swift 类中使用`@objc(<#name#>)`关键字，这个类可以不需要命名空间即可在 Objective-C 中使用。这个关键字在你迁徙 Objecive-C 代码到 Swift 时同样也非常有用。由于归档过的对象存贮了类的名字，你应该使用`@objc(<#name#>)`来声明与旧的归档过的类相同的名字，这样旧的类才能被新的 Swift 类解档。

Objective-C 选择器(Selectors)

一个 Objective-C 选择器类型指向一个 Objective-C 的方法名。在 Swift 时代，Objective-C 的选择器被 `Selector` 结构体替代。你可以通过字符串创建一个选择器，比如 `let mySelector: Selector = "tappedButton:"`。因为字符串能够自动转换为选择器，所以你可以把字符串直接传递给接受选择器的方法。

```
//Swift
import UIKit
class MyViewController: UIViewController {
    let myButton = UIButton(frame: CGRect(x: 0, y: 0, width: 100, height: 50))

    init(nibName nibNameOrNil: String!, bundle nibBundleOrNil: NSBundle!) {
```

```
        super.init(nibName: nibName, bundle: nibBundle)
        myButton.targetForAction("tappedButton:", withSender: self)
    }

    func tappedButton(sender: UIButton!) {
        println("tapped button")
    }
}
```

注意： `performSelector:`方法和相关的调用选择器的方法没有导入到 Swift 中因为它们是不安全的。

如果你的 Swift 类继承自 Objective-C 的类，你的所有方法都可以用作 Objective-C 的选择器。另外，如果你的 Swift 类不是继承自 Objective-C，如果你想要当选择器来使用你就需要在前面添加`@objc`关键字，详情请看 Swift 类型兼容性。

2.2. 使用 Objective-C 特性编写 Swift 类

本篇译者：halinuya（git 主页），敬请勘误。

互用性（互操作性）使开发者可以定义融合了 Objective-C 语言特性的 Swift 类。编写 Swift 类时，不仅可以继承 Objective-C 语言编写的父类，采用 Objective-C 的协议，还可以利用 Objective-C 的一些其它功能。这意味着，开发者可以基于 Objective-C 中已有的熟悉、可靠的类、方法和框架来创建 Swift 类，并结合 Swift 提供的现代化和更有效的语言特点对其进行优化。

继承 Objective-C 的类

在 Swift 中，开发者可以定义一个子类，该子类继承自使用 Objective-C 编写的类。创建该子类的方法是，在 Swift 的类名后面加上一个冒号（:），冒号后面跟上 Objective-C 的类名。

```
SWIFT

import UIKit
```

```
class MySwiftViewController: UIViewController {
    // 定义类
}
```

开发者能够从 Objective-C 的父类中继承所有的功能。如果开发者要覆盖父类中的方法，不要忘记使用 `override` 关键字。

采用协议

在 Swift 中，开发者可以采用 Objective-C 中定义好的协议。和 Swift 协议一样，所有 Objective-C 协议都写在一个用逗号隔开的列表中，跟在所在类的父类名后面（如果它有父类的话）。

```
SWIFT

class MySwiftViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource {
    // 定义类
}
```

Objective-C 协议与 Swift 协议使用上是一致的。如果开发者想在 Swift 代码中引用 `UITableViewDelegate` 协议，可以直接使用 `UITableViewDelegate`（跟在 Objective-C 中引用 `id<UITableViewDelegate\>`是等价的）。

编写构造器和析构器

Swift 的编译器确保在初始化时，构造器不允许类里有任何未初始化的属性，这样做能够增加代码的安全性和可预测性。另外，与 Objective-C 语言不同，Swift 不提供单独的内存分配方法供开发者调用。当你使用原生的 Swift 初始化方法时（即使是和 Objective-C 类协作），Swift 会将 Objective-C 的初始化方法转换为 Swift 的初始化方法。关于如何实现开发者自定义构造器的更多信息，请查看构造器。

当开发者希望在类被释放前，执行额外的清理工作时，需要执行一个析构过程来代替 `dealloc` 方法。在实例被释放前，Swift 会自动调用析构器来执行析构过程。Swift 调用完子类的析构器后，会自动调用父类的析构器。当开发者使用

Objective-C 类或者是继承自 Objective-C 类的 Swift 类时，Swift 也会自动为开发者调用这个类的父类里的 `dealloc` 方法。关于如何实现开发者自定义析构器的更多信息，请查看析构器。

集成 Interface Builder

Swift 编译器包含一些属性，使得开发者的 Swift 类集成了 Interface Builder 里的一些特色功能。和 Objective-C 里一样，你能在 Swift 里面使用 OutLets，actions 和实时渲染（live rendering）。

使用 Outlets 和 Action

使用 Outlets 和 Action 可以连接源代码和 Interface Builder 的 UI 对象。在 Swift 里面使用 Outlets 和 Action，需要在属性和方法声明前插入`@IBOutlet`或者`@IBAction`关键字。声明一个 Outlet 集合同样是用`@IBOutlet`属性，即为类型指定一个数组。

当开发者在 Swift 里面声明了一个 Outlet 时，Swift 编译器会自动将该类型转换为弱（weak）、隐式（implicitly）、未包装（unwrapped）的 optional（Object-c 里面对应指针类型）数据类型，并为它分配一个初始化的空值 `nil`。实际上，编译器使用`@IBOutlet weak var name: Type! = nil`来代替 `@IBOutlet var name: Type`。编译器将该类型转换成了弱（weak）、隐式（implicitly）、未包装（unwrapped）的 optional 类型，因此开发者就不需要在构造器中为该类型分配一个初始值了。当开发者从故事板（storyboard）或者 `xib` 文件里面初始化对象 class 后，定义好的 Outlet 和这些对象连接在一起了，所以，这些 Outlet 是隐式的，未包装的。由于创建的 outlets 一般都是弱关系，因此默认 outlets 是弱类型。

例如，下面的 Swift 代码声明了一个拥有 Outlet、Outlets 集合和 Action 的类：

```
SWIFT

class MyViewController: UIViewController {

    @IBOutlet var button: UIButton

    @IBOutlet var textFields: UITextField[]
```

```
    @IBAction func buttonTapped(AnyObject) {
        println("button tapped!")
    }
}
```

在 `buttonTapped` 方法中，消息发送者的信息没有被使用，因此可以省略该方法的参数名。

实时渲染（live rendering）

开发者可以在 Interface Builder 中用`@IBDesignable`和`@IBInspectable`来创建生动、可交互的自定义视图（view）。开发者继承 `UIView` 或者 `NSView` 来自定义一个视图（view）时，可以在类声明前添加`@IBDesignable`属性。当你在 Interface Builder 里添加了自定义的视图后（在监视器面板的自定义视图类中进行设置），Interface Builder 将在画布上渲染你自定义的视图。

注意：只能针对框架里对象进行实时渲染

你也可以将`@IBInspectable`属性添加到和用户定义的运行时属性兼容的类型属性里。这样，当开发者将自定义的视图添加到 Interface Builder 里后，就可以在监视器面板中编辑这些属性。

```
SWIFT

@IBDesignable

class MyCustomView: UIView {
    @IBInspectable var textColor: UIColor
    @IBInspectable var iconHeight: CGFloat
    /* ... */
}
```

指明属性特性

在 Objective-C 中，属性通常都有一组特性（Attributes）说明来指明该属性的一些附加信息。在 Swift 中，开发者可以通过不同的方法来指明属性的这些特性。

强类型和弱类型

Swift 里属性默认都是强类型的。使用 `weak` 关键字修饰一个属性，能指明其对象存储时是一个弱引用。该关键字仅能修饰 optional 对象类型。更多的信息，请查阅特性。

读 / 写和只读

在 Swift 中，没有 `readwrite` 和 `readonly` 特性。当声明一个存储型属性时，使用 `let` 修饰其为只读；使用 `var` 修饰其为可读 / 写。当声明一个计算型属性时，为其提供一个 getter 方法，使其成为只读的；提供 getter 方法和 setter 方法，使其成为可读 / 写的。更多信息，请查阅属性。

拷贝语义

在 Swift 中，Objective-C 的 `copy` 特性被转换为`@NSCopying`属性。这一类的属性必须遵守 `NSCopying` 协议。更多信息，请查阅特性。

实现 Core Data Managed Object Subclasses

Core Data 提供了基本存储和实现 `NSManagedObject` 子类的一组属性。在 Core Data 模型中，与管理对象子类相关的特性或者关系的每个属性定义之前，将 `@NSmanaged` 特性加入。与 Objective-C 里面的 `@dynamic` 特性类似，`@NSManaged` 特性告知 Swift 编译器，这个属性的存储和实现将在运行时完成。但是，与`@dynamic` 不同的是，`@NSManaged` 特性仅在 Core Data 支持中可用。

2.3. Cocoa 数据类型

本篇译者：wongzigii（Github 主页），敬请勘误。

作为对 Objective-C 互用性（互操作性）的一部分，Swift 提供快捷高效的方式来处理 Cocoa 数据类型。

Swift 会自动将一些 Objective-C 类型转换为 Swift 类型，以及将 Swift 类型转换为 Objective-C 类型。在 Objective-C 和 Swift 中也有一些具有互用性的数据类型。那些可转换的数据类型或者具有互用性的数据类型被称为 *bridged* 数据类型。举个例子，在 Swift 中，您可以将一个 `Array` 值传递给一个要求为 `NSArray` 对象的方法。你也可以转换一个 bridged 类型和它的副本。当你使用 `as` 转换 bridged 类型或者那些由常量和变量所提供的类型时，Swift 会桥接它们的数据类型。

Swift 也提供一种简单便捷的覆盖方法来连接 Foundation 的数据类型，在后面的 Swift 语言中，你能在它的句法中感受到自然和统一。

字符串

wift 会在 `String` 类型和 `NSString` 类型中自动转换。这意味着在可以使用 `NSString` 对象的地方，您可以使用一个属于 Swift 的 `String` 类型代替它，这样做会同时拥有它们数据类型的特点，`String` 类型的插值，基于 Swift 设计的 APIs 以及 `NSString` 类更广的适用范围。因此，您几乎不必再在你的代码中使用 `NSString` 类。事实上，当 Swift 接入 Objective-C APIs 时，它将把所有 `NSString` 类型替换为 `String` 类型。当您在您的 Objective-C 代码中使用 Swift 类时，接入的 API 会将所有 `String` 类型替换成 `NSString` 类型。
为了允许字符串转换，只需接入 Foundation。举个例子，您在 Swift 的一个字符串中调用了 `capitalizedString`--一个 `NSString` 类的方法，此后 Swift 会自动将 `String` 转换为一个 `NSString` 对象并调用方法。这个方法甚至会返回一个 Swift 的 `String` 类型，因为它在接入的时候被替换了。

```
1.  import Foundation
2.  let greeting = "hello, world!"
3.  let capitalizedGreeting = greeting.capitalizedString
4.  // capitalizedGreeting: String = Hello, World!
```

如果您确实需要用到一个 `NSString` 对象，您可以用一个 Swift 的 `String` 值并转换它。`String` 类型总是可以从一个 `NSString` 对象转换为一个 Swift 的 `String` 的值，因此，再没有必要去使用一个可选的类型转换器`()as?)`。您也可以再一个字符串中通过定义常量和变量来创建一个 `NSString` 对象。

```
1.  import Foundation
```

```
2.  let myString: NSString = "123"
3.  if let integerValue = (myString as String).toInt()){
4.      println("\(myString) is the integer \(integerValue)")
5.  }
```

本地化

在 Objective-C 中，常用 `NSLocalizedString` 类的宏来定位一个字符串。这集合的宏包括 `NSLocalizedStringFromTableInBundle` 和 `NSLocalizedStringWithDefaultValue`。而在 Swift 中，只用一个函数就可以实现跟整个 `NSLocalizedString` 集一样的功能，即 `NSLocalizedString(key:tableName:bundle:value:comment:)`。这个 `NSLocalizedString` 函数分别为 `tableName`，`bundle` 和 `value` 参数提供了一个默认值。你可以用它来替换宏。

数字

Swift 会自动将已确定的数字类型 `Int` 和 `Float` 转换为 `NSNumber`。这样的转换允许你基于其中一种类型创建一个 `NSNumber`：

```
let n = 42
let m: NSNumber = n
```

你也能传递一个 `Int` 类型的值，比如传递给一个要求为 `NSNumber` 类型的参数。同时需要注意的是，`NSNumber` 可以包含多种不同的类型，因此您不能把它传递给单一的一个 `Int` 值。

下面所列出的类型都会自动转换为 `NSNumber`：

- `Int`
- `UInt`
- `Float`
- `Double`
- `Bool`

类集合

Swift 会自动将 `NSArray` 和 `NSDictionary` 类转换为 Swift 里等价的类。这意味着你将

受益于 Swift 强大的算法和得天独厚的语法来处理集合--可互相转换的 Foundation 和 Swift 集合类型。

数组

Swift 会在 `Array` 类型和 `NSArray` 类型中自动转换。当你从一个 Swift 数组转换到一个 `NSArray` 对象，转换后的数组是一个 `AnyObject[]` 类型的数组。如果某个对象是 Objective-C 或者 Swift 类的实例，或者这个对象可以转换成另一种类型，那么这个对象则属于 `AnyObject` 类型的对象。你可以将任一 `NSArray` 对象转换成一个 Swift 数组，因为所有 Objective-C 的对象都是 `AnyObject` 类型的。正因如此，Swift 的编译器会在接入 Objective-C APIs 的时候将 `NSArray` 类替换成 `AnyObject[]`。
当你将一个 `NSArray` 对象转换成一个 Swift 数组后，你也可以将数组强制类型转换成一个特定的类型。与从 `NSArray` 类转换到 `AnyObject[]` 不同的是，从 `AnyObject` 类型的对象转换成明确的类型并不会保证成功。由于直到运行时编译器才知道 `AnyObject` 的对象能否被强制转换为特定的类型，因此，从 `AnyObject[]` 转换为 `SomeType[]` 会返回一个 optional 的值。举个例子，如果你知道一个 Swift 数组只包含 `UIView` 类的实例(或者一个 `UIView` 类的子类)，你可以将 `AnyObject` 类型的数组元素强制转换为 `UIView` 对象。如果 Swift 数组中得元素在运行时不是 `UIView` 类型的对象，那么转换则会返回 `nil`。

```
let swiftyArray = foundationArray as AnyObject[]
if let downcastedSwiftArray = swiftArray as? UIView[] {
    // downcastedSwiftArray contains only UIView objects
}
```

你也可以在 for 循环中将 NSArray 对象定向地强制转换为特定类型的 Swift 数组:

```
for aView: UIView! in foundationArray {
     // aView is of type UIView
}
```

注意：这种转换是强制转换，如果转换不成功则会在运行时产生错误信息。
当你从 Swift 数组转换为 `NSArray` 对象时，Swift 数组里的元素必须是属于 `AnyObject` 的。例如，一个 `Int[]` 类型的 Swift 数组包含 `Int` 结构的元素。`Int` 类型并不是一个类的实例，但由于 `Int` 类型转换成了 `NSNumber` 类，`Int` 类型属于 `AnyObject` 类型的。因此，你可以将一个 `Int[]` 类型的 Swift 数组转换为 `NSArray` 对

象。如果 Swift 数组里的一个元素不属于 `AnyObject` 类型，那么在运行时就会产生错误。

你也可以从 Swift 数组中创建一个 `NSArray` 对象。当你将一个常量或变量定义为一个 `NSArray` 对象并分配一个数组给它作为实例变量时，Swift 将会创建一个 `NSArray` 对象，而不是一个 Swift 数组。

```
let schoolSupplies: NSArray = ["Pencil", "Eraser", "Notebkko"]
// schoolSupplies is an NSArray object containing NSString objects
```

上面的例子中，Swift 数组包含包含三个 `String` 字符串。由于从 `String` 类型转换为 `NSString` 类，数组字面量被转换成一个 `NSArray` 对象，并成功分配给 `schoolSupplies` 变量。

当您在 Objective-C 代码中使用 Swift 类或者协议时，接入的 API 会将全部所有类型的 Swift 数组代替为 `NSArray`。若您将一个 `NSArray` 对象传递给 Swift 的 API 并要求数组元素为一个新的类型，运行时就会产生错误。如果 Swift API 返回一个不能被转换为 `NSArray` 类型的 Swift 数组，错误也会产生。

字典

敬请期待

Foundation 数据类型

Swift 也提供一种简单便捷的覆盖方法来连接定义在 Foundation 框架中的数据类型。在 `NSSize` 和 `NSPoint` 中使用覆盖方法，在剩下的 Swift 语言中，你能在它的句法中感受到自然和统一。比如，你可以使用如下语法创建一个 `NSSize` 类型的结构:

```
let size = NSSize(width: 20, height: 40)
```

覆盖方法也允许你以一种自然的方式调用 Foundation 的结构函数。

```
let rect = NSRect(x: 50, y: 50, width: 100, height: 100)
let width = rect.width    // equivalent of NSWidth(rect)
let maxX = rect.maxY      // equivalent of NSMaxY(rect)
```

Swift 可以将 `NSUInteger` 和 `NSInteger` 转换为 `Int` 类型。这些类型都会在 Foundation APIs 中变为 `Int` 类型。在 Swift 中 `Int` 常被尽可能地用以连贯性，同时当你要求一个无符号整数类型时，`UInt` 类型总是可使用的。

Foundation 函数

在 Swift 中，`NSLog` 可在系统控制台输出信息。您可以像在 Objective-C 中使用过的语法格式那样使用此函数。

```
NSLog("%.7f", pi)        // Logs "3.1415927" to the console
```

同时，Swift 也提供像 `print` 和 `println` 那样的输出函数。这些函数简单，粗暴，多效，多归于 Swift 的字符插入法。这些函数不会在系统控制台输出信息，但在需要的时候却是存在可用的。
Swift 中不再存在 `NSAssert` 函数，取而代之的是 `assert` 函数。

Core Foundation

Swift 中的 Core Foundation 类型是一个成熟的类。当出现内存管理注释时，Swift 会自动地管理 Core Foundation 对象的内存，这其中包括你实例化了的 Core Foundation 对象。在 Swift 中，你可以自由变换 Fundation 和 Core Foundation 类型。如果你想先转换为桥接 Foundation 类型时，你也可以桥接一些 toll-free bridged Core Foundation 类型到 Swift 标准库类型。

重定义类型

当 Swift 导入 Core Foundation 类型时，编译器会重映射导入的类型名字。编译器会从每个类型名字的末端移除 *Ref*，这是因为所有的 Swift 类都属于引用类型，因此后缀是多余的。

Core Foundation 中的 `CFTypeRef` 类型会对 `Anyobject` 类型重映射。所以你以前使用的 `CFTypeRef`，现在该换成 `AnyObject` 了

内存管理对象

在 Swift 中，从 annotated APIs 返回的 Core Foundation 对象能够自动进行内存管理--你不再需要调用自身的 `CFRetain`，`CFRelease`，或者 `CFAutorelease` 函数。如果你从自身的 C 函数和 Objective-C 方法中返回一个 Core Foundation 对象，你需要用 `CF_RETURNS_RETAINED` 或者 `CF_RETURNS_NOT_RETAINED` 注释这个对象。当 Swift 代码中包含这些 APIs 时，编译器会在编译时自动调用内存管理。如果你只调用那些不会间接返回 Core Foundation 对象的 annotated APIs，那么现在你可以跳过本节的剩余部分了。否则，让我们继续学习那些难管理的 Core Foundation 对象吧。

非托管对象

当 Swift 导入还尚未被注释的 APIs 时，编译器将不会自动地对返回的 Core Foundation 对象进行内存管理。Swift 将这些返回的 Core Foundation 对象封闭在一个 `Unmanaged<T>`结构中。那些间接返回 Core Foundation 的对象也是难以管理的。举个例子，这里有一个 unannotated 的 C 函数:

```
CFStringRef StringByAddingTwoStrings(CFStringRef string1, CFStringRef string2)
```

这里说明了 Swift 是怎么导入的:

```
func StringByAddingTwoStrings(CFString!, CFString!) -> Unmanaged<CFString>!
```

假设您从 unannotated APIs 接收了一个难以管理的对象，在使用它之前，你必须要将它转换为一个能够内存管理的对象。在这方面，Swift 可以帮你进行内存管理而不用自己动手。同时，`Unmanaged<T>`结构也提供了两个方法来把一个难以管理的对象转换为一个可内存管理的对象--`takeUnretainedValue()`方法和`takeRetainedValue()`方法。这两个方法会返回原始的，开放的对象类型。您可以根据您实际调用的 APIs 返回的 unretained 或 retained 的对象，来选择哪一方法更合适。

比如，假设这里有一个 C 函数，这个函数在返回值前不会释放 `CFString` 对象。在使用这个对象前，您使用 `takeUnretainedValue()`函数，以将它转换为一个能够内存管理的对象。

```
let memoryManagedResult = StringByAddingTwoStrings(str1,
str2).takeUnretainedValue()
```

```
// memoryManagedResult is a memory managed CFString
```

您也可以在一个非托管的对象中使用 `retain()`， `release()`和 `autorelease()`方法，但是这种做法并不值得推荐。

2.4. 采用 Cocoa 设计模式

本篇译者：@JaceFu（git 主页），敬请勘误。

使用 Cocoa 现有的一些设计模式，是帮助开发者开发一款拥有合理设计思路、稳定的性能、良好的可扩展性应用的有效方法之一。这些模式都依赖于在 Objective-C 中定义的类。因为 Swift 与 Objective-C 的互用性，所以你依然可以在 Swift 代码中使用这些设计模式。在一些情况下，你甚至可以使用 Swift 语言的特性扩展或简化这些 Cocoa 设计模式，使这些设计模式更强大、更易于使用。

委托

在 Swift 和 Objective-C 中，委托通常由一个定义交互方法和遵循规范的委托属性的协议表示。与 Objective-C 相比，当你在 Swift 中继承一个委托时，虽然继承模式不变，但是内部的实现已经改变了。就像在 Objective-C 中，在你向委托发送消息之前，不管它是不是 `nil` 你都会去查看，如果定义的方法是非必须实现的方法，不管委托有没有实现这个方法，你也都会去查看。而在 Swift 中，通过保持类型安全的特性，可以有效的消除这些繁琐、不必要的行为问题。
下面列出的代码可以说明这个过程：

1. 检查 `myDelegate` 不为 `nil`。
2. 检查 `myDelegate` 是否实现了继承的 `window:willUseFullScreenContentSize:` 方法。
3. 如果 `myDelegate` 不为 `nil` 并且实现了 `window:willUseFullScreenContentSize:` 方法，那么调用该方法，将该方法的返回值分配给名为 `fullScreenSize` 的属性。
4. 将该方法的返回值输出在控制台。

```
// @inteface MyObject : NSObject
// @property (nonatomic, weak) id<NSWindowDelegate> delegate;
// @end
if let fullScreenSize = myDelegate?.window?(myWindow, willUseFullScreenContentSize:
mySize) {
```

```
    println(NSStringFromSize(fullScreenSize))
}
```

注意： 在一个完全使用 Swift 编写的 app 中，在定义 `delegate` 属性时，它作为一个不定值的 `NSWindowDelegate` 对象，并将初始值设为 `nil`。

延迟初始化

你可以在 Lazy Stored Properties 中了解到更多关于延迟初始化的信息。

错误报告

Swift 中的错误报告模式沿用了 Objective-C 的模式，但 Swift 中不定值返回值的新特性给我们带来了额外的好处。举个很简单的例子，你用 `Bool` 值作为一个函数的返回值，用于标识该函数是否执行成功，当你需要输出错误信息时，你可以在函数中添加一个 `NSErrorPointer` 类型的输出参数 `NSError`。这个类型类似 Objective-C 中的 `NSError **`，并增加了内存安全性和非强制性的传参。你可以使用 `&` 运算符作为前缀引用一个不定值 `NSError` 类型作为 `NSErrorPointer` 对象传递错误信息。如下面的代码所示：

```
1.  var writeError : NSError?
2.  let written = myString.writeToFile(path, atomically: false,
3.      encoding: NSUTF8StringEncoding,
4.      error: &writeError)
5.  if !written {
6.      if let error = writeError {
7.          println("write failure: \(error.localizedDescription)")
8.      }
9.  }
```

当你实现自己的方法时，你需要配置一个 `NSErrorPointer` 对象，并将 `NSErrorPointer` 对象的 `memory` 属性设为你创建的 `NSError` 对象。首先检查调用者传递的参数，确保它是一个非 `nil` 的 `NSError` 对象。

```
1.  func contentsForType(typeName: String! error: NSErrorPointer) -> AnyObject!
    {
2.      if cannotProduceContentsForType(typeName) {
3.          if error {
```

```
4.             error.memory = NSError(domain: domain, code: code, userInfo: [:])
5.         }
6.         return nil
7.     }
8.     // ...
9. }
```

键值观察

敬请期待。

Target-Action 模式

当有特定事件发生，需要一个对象向另一个对象发送消息时，我们通常采用 Cocoa 的 Target-Action 设计模式。Swift 和 Objective-C 中的 Target-Action 模型基本类似。在 Swift 中，你可以使用 `Selector` 类型达到 Objective-C 中 selectors 的效果。请在 Objective-C Selectors 中查看在 Swift 中使用 Target-Action 设计模式的示例。

类型匹配与统一规范

在 Objective-C 中，你可以使用 `isKindOfClass:` 方法检查某个对象是否是指定类型，可以使用 `conformsToProtocol:` 方法检查某个对象是否遵循特定协议的规范。在 Swift 中，你可以使用 `is` 运算符完成上述的功能，或者也可以使用 `as?` 向下匹配指定类型。

你可以使用 `is` 运算符检查一个实例是否是指定的子类。如果该实例是指定的子类，那么 `is` 运算结果为 `true`，反之为 `false`。

```
if object is UIButton {
    // object is of type UIButton
} else {
    // object is not of type UIButton
}
```

你也可以使用 `as?` 运算符尝试向下匹配子类型，`as?` 运算符返回不定值，结合 `if-let` 语句使用。

```
if let button = object as? UIButton {
    // object is successfully cast to type UIButton and bound to button
} else {
    // object could not be cast to type UIButton
}
```

请在 Type Casting 中查看更多信息。

检查匹配协议的语法与检查匹配类的语法是一样的，下面是使用 `as?` 检查匹配协议的示例：

```
if let dataSource = object as? UITableViewDataSource {
    // object conforms to UITableViewDataSource and is bound to dataSource
} else {
    // object not conform to UITableViewDataSource
}
```

注意，当做完匹配之后，`dataSource` 会转换为 `UITableViewDataSource` 类型，所以你只能访问和调用 `UITableViewDataSource` 协议定义的属性和方法。当你想进行其他操作时，必须将其转换为其他的类型。

可以在 Protocols 查看更多相关信息。

2.5. 与 C 语言交互编程

本篇译者：shockinglee（git 主页），敬请勘误。

作为一种可与 Objective-C 相互调用的语言，Swift 也具有一些与 C 语言的类型和特性，如果你的代码有需要，Swift 也提供了和常见的 C 代码结构混合编程的编程方式。

基本数据类型

Swift 提供了一些和 C 语言的基本类型如 `char`，int，float，double`等价的 Swift 基本数据类型。然而，这些 Swift 的核心基本类型之间并不能隐式的相互转换，如 Int。因此，只有你的代码明确要求它们时再使用这些类型，而 Int 可以在任何你想使用它的时候使用。

C 类型	Swift 类型
bool	CBool
char, signed char	CChar
unsigned char	CUnsignedChar
short	CShort
unsigned short	CUnsignedShort
int	CInt
unsigned int	CUnsignedInt
long	CLong
unsigned long	CUnsignedLong
long long	CLongLong
unsigned long long	CUnsignedLongLong
wchar_t	CWideChar
char16_t	CChar16
char32_t	CChar32
float	CFloat
double	CDouble

枚举

Swift 引进了用宏 `NS_ENUM` 来标记的任何 C 风格的枚举类型。这意味着无论枚举值是在系统框架还是在自定义的代码中定义的，当他们导入到 Swift 时，他们的前缀名称将被截断。例如，看这个 Objective-C 枚举：

```
//Objective-C
typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
    UITableViewCellStyleSubtitle
};
```

在 Swift 中这样来实现：

```
//Swift
enum UITableViewCellStyle: Int {
    case Default
    case Value1
    case Value2
    case Subtitle
    }
```

当您需要指向一个枚举值时，使用以点（.）开头的枚举名称：

```
//Swift
let cellStyle: UITableViewCellStyle = .Default
```

Swift 也引进了标有 `NS_OPTIONS` 宏选项。而选项的行为类似于引进的枚举，选项还可以支持一些位操作，如 &，| 和 ～。在 Objective-C 中，你用一个空的选项设置标示恒为零（0）。在 Swift 中，使用 `nil` 代表没有任何选项。

指针

Swift 尽可能避免让您直接访问指针。然而，当您需要直接操作内存的时候，Swift 也为您提供了多种指针类型。下面的表使用 Type 作为占位符类型名称来表示语法的映射。

对于参数，使用以下映射：

C 句法	Swift 句法
const void *	CConstVoidPointer
void *	CMutableVoidPointer
const Type *	CConstPointer<Type>
Type *	CMutablePointer<Type>

对于返回类型，变量和参数类型的多层次指针，使用以下映射：

C 句法	Swift 句法
void *	COpaquePointer
Type *	UnsafePointer<Type>

对于类（class）类型，使用以下映射：

C 句法	Swift 句法
Type * const *	CConstPointer<Type>
Type * __strong *	CMutablePointer<Type>
Type **	AutoreleasingUnsafePointer<Type>

C 可变指针

当一个函数被声明为接受 `CMutablePointer\<Type\>` 参数时，这个函数可以接受下列任何一个类型作为参数：

- `nil`，作为空指针传入
- 一个 `CMutablePointer\<Type\>` 类型的值
- 一个操作数是 Type 类型的左值的输入输出表达式，作为这个左值的内存地址传入

- 一个输入输出 Type[] 值，作为一个数组的起始指针传入，并且它的生命周期将在这个调用期间被延长

  如果您像这样声明了一个函数：

```
//Swift
func takesAMutablePointer(x: CMutablePointer<Float>) { /*...*/ }
```

  那么您可以使用以下任何一种方式来调用这个函数：

```
//Swift
var x: Float = 0.0
var p: CMutablePointer<Float> = nil
var a: Float[] = [1.0, 2.0, 3.0]

takesAMutablePointer(nil)
takesAMutablePointer(p)
takesAMutablePointer(&x)
takesAMutablePointer(&a)
```

  当函数被声明使用一个 `CMutableVoidPointer` 参数，那么这个函数接受任何和 `CMutablePointer\<Type\>` 相似类型的 Type 操作数。

  如果您这样定义了一个函数：

```
//Swift
func takesAMutableVoidPointer(x: CMutableVoidPointer) { /* ... */ }
```

  那么您可以使用以下任何一种方式来调用这个函数：

```
//Swift
var x: Float = 0.0, y: Int = 0
var p: CMutablePointer<Float> = nil, q: CMutablePointer<Int> = nil
var a: Float[] = [1.0, 2.0, 3.0], b: Int = [1, 2, 3]

takesAMutableVoidPointer(nil)
takesAMutableVoidPointer(p)
takesAMutableVoidPointer(q)
takesAMutableVoidPointer(&x)
```

```
takesAMutableVoidPointer(&y)
takesAMutableVoidPointer(&a)
takesAMutableVoidPointer(&b)
```

C 常指针

当一个函数被声明为接受 `CConstPointer\<Type\>`参数时，这个函数可以接受下列任何一个类型作为参数：

- `nil`，作为空指针传入
- 一个 `CMutablePointer\<Type\>`, `CMutableVoidPointer`, `CConstPointer\<Type\>`, `CConstVoidPointer`，或者在必要情况下转换成 `CConstPointer\<Type\>`的 `AutoreleasingUnsafePointer\<Type\>`值
- 一个操作数是 Type 类型的左值的输入输出表达式，作为这个左值的内存地址传入
- 一个 `Type[]`数组值，作为一个数组的起始指针传入，并且它的生命周期将在这个调用期间被延长

如果您这样定义了一个函数：

```
//Swift
func takesAConstPointer(x: CConstPointer<Float>) { /*...*/ }
```

那么您可以使用以下任何一种方式来调用这个函数：

```
//Swift
var x: Float = 0.0
var p: CConstPointer<Float> = nil

takesAConstPointer(nil)
takesAConstPointer(p)
takesAConstPointer(&x)
takesAConstPointer([1.0, 2.0, 3.0])
```

当函数被声明使用一个 `CConstVoidPointer` 参数，那么这个函数接受任何和 `CConstPointer\<Type\>` 相似类型的 `Type` 操作数。 如果您这样定义了一个函数：

```
//Swift
func takesAConstVoidPointer(x: CConstVoidPointer) { /* ... */ }
```

那么您可以使用以下任何一种方式来调用这个函数：

```
//Swift
var x: Float = 0.0, y: Int = 0
var p: CConstPointer<Float> = nil, q: CConstPointer<Int> = nil

takesAConstVoidPointer(nil)
takesAConstVoidPointer(p)
takesAConstVoidPointer(q)
takesAConstVoidPointer(&x)
takesAConstVoidPointer(&y)
takesAConstVoidPointer([1.0, 2.0, 3.0])
takesAConstVoidPointer([1, 2, 3])
```

自动释放不安全指针

当一个函数被声明为接受 `AutoreleasingUnsafePointer\<Type\>` 参数时，这个函数可以接受下列任何一个类型作为参数：

- `nil`，作为空指针传入
- 一个 `AutoreleasingUnsafePointer\<Type\>` 值
- 其操作数是原始的，复制到一个临时的没有所有者的缓冲区的一个输入输出表达式，该缓冲区的地址传递给调用，并返回时，缓冲区中的值加载，保存，并重新分配到操作数。

注意：这个列表没有包含数组。

如果您这样定义了一个函数：

```
//Swift
```

```
func takesAnAutoreleasingPointer(x: AutoreleasingUnsafePointer<NSDate?>) { /* ...
*/ }
```

那么您可以使用以下任何一种方式来调用这个函数：

```
//Swift
var x: NSDate? = nil
var p: AutoreleasingUnsafePointer<NSDate?> = nil

takesAnAutoreleasingPointer(nil)
takesAnAutoreleasingPointer(p)
takesAnAutoreleasingPointer(&x)
```

注意：C 语言函数指针没有被 Swift 引进。

全局常量

在 C 和 Objective-C 语言源文件中定义的全局常量会自动地被 Swift 编译引进并做为 Swift 的全局常量。

预处理指令

Swift 编译器不包含预处理器。取而代之的是，它充分利用了编译时属性，生成配置，和语言特性来完成相同的功能。因此，Swift 没有引进预处理指令。

简单宏

在 C 和 Objective-C，您通常使用的#define 指令定义的一个宏常数，在 Swift，您可以使用全局常量来代替。例如：一个全局定义`#define FADE_ANIMATION_DURATION 0.35`，在 Swift 可以使用 `let FADE_ANIMATION_DURATION = 0.35` 来更好的表述。由于简单的用于定义常量的宏会被直接被映射成 Swift 全局量，Swift 编译器会自动引进在 C 或 Objective-C 源文件中定义的简单宏。

复杂宏

在 C 和 Objective-C 中使用的复杂宏在 Swift 中并没有与之对应的定义。复杂宏是那些不用来定义常量的宏，而是用来定义包含小括号（），函数的宏。您在 C 和 Objective-C 使用复杂的宏是用来避免类型检查的限制和相同代码的重复劳动。然而，宏也会产生 Bug 和重构的困难。在 Swift 中你可以直接使用函数和泛型来达到同样的效果。因此，在 C 和 Objective-C 源文件中定义的复杂宏在 Swift 是不能使用的。

编译配置

Swift 代码和 Objective-C 代码以不同的方式进行条件编译。Swift 代码可以根据生成配置的评价配进行有条件的编译。生成配置包括 true 和 false 字面值，命令行标志，和下表中的平台测试函数。您可以使用-D <#Flag#>指定命令行标志。

函数	有效参数
os()	OSX, iOS
arch()	x86_64, arm, arm64, i386

注意

arch(arm) 的生成配置不会为 64 位 arm 设备返回 `true`，当代码运行在为 32 位的 ios 模拟器器时，arch(i386) 的生成配置返回 `true`。

一个简单的条件编译需要以下代码格式：

```
#if build configuration
    statements
#else
    statements
#endif
```

一个由零个或多个有效的 Swift 语句声明的 `statements`，可以包括表达式，语句和控制流语句。您可以添加额外的构建配置要求，条件编译说明用 && 和 || 操作符以及 ！ 操作符，添加条件控制块用 #elseif：

```
#if build configuration && !build configuration
    statements
#elseif build configuration
    statements
#else
    statements
#endif
```

与 C 语言编译器的条件编译相反，Swift 条件编译语句必须完全是自包含和语法有效的代码块。这是因为 Swift 代码即使没有被编译，也要全部进行语法检查。

3.Mix and Match

3.1. 在同一工程中使用 Swift 和 Objective-C

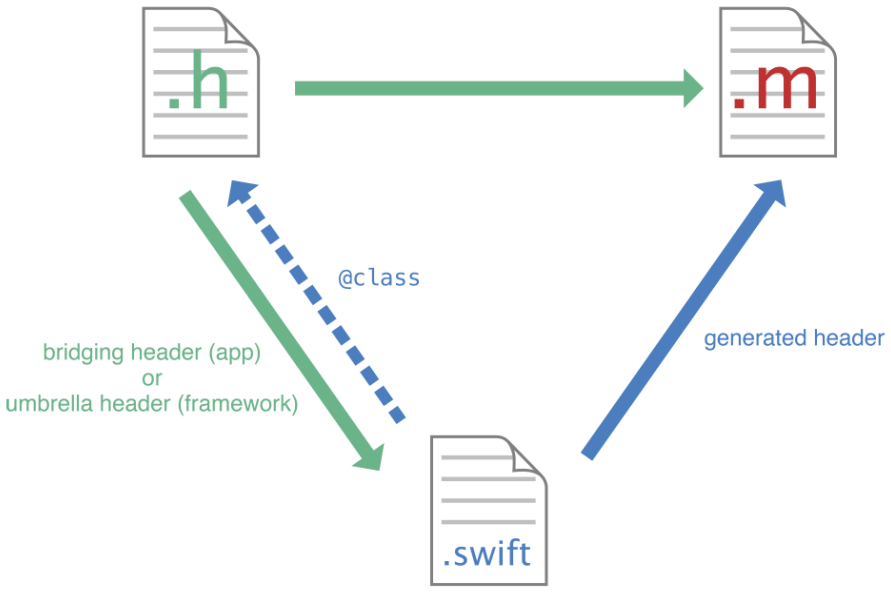
本篇译者：haolloyin（github 主页），敬请勘误。

Swift 与 Objective-C 的兼容能力使你可以在同一个工程中同时使用两种语言。你可以用这种叫做 `mix and match` 的特性来开发基于混合语言的应用，可以用 Swfit 的最新特性实现应用的一部分功能，并无缝地并入已有的 Objective-C 的代码中。

Mix and Match 概述

Objective-C 和 Swift 文件可以在一个工程中并存，不管这个工程原本是基于 Objective-C 还是 Swift。你可以直接往现有工程中简单地添加另一种语言的源文件。这种自然的工作流使得创建混合语言的应用或框架 target，与用单独一种语言时一样简单。

混合语言的工作流程只有一点点区别，这取决于你是在写应用还是写框架。下面描述了普通的用两种语言在一个 target 中导入模型的情况，后续章节会有更多细节。

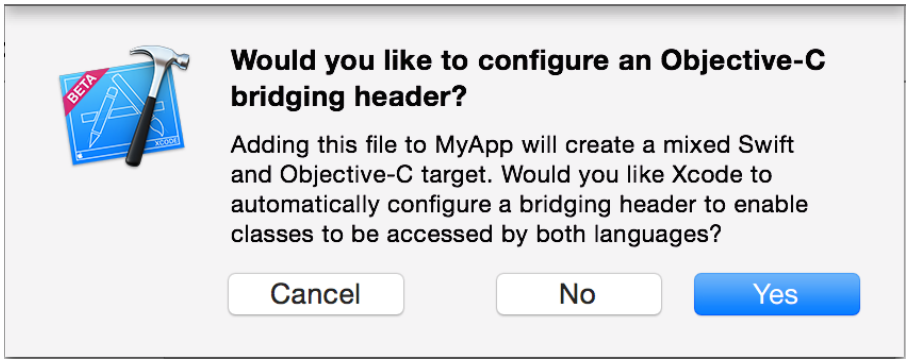


在同一个 App Target 中进行代码导入

如果你在写混合语言的应用，可能需要用 Swift 代码访问 Objective-C 代码，或者反之。下面的流程描述了在非框架 target 中的应用。

将 Objective-C 导入 Swift

在一个应用的 target 中导入一些 Objective-C 文件供 Swift 代码使用时，你需要依赖与 Objective-C 的桥接头文件（`bridging header`）来暴露给 Swift。当你添加 Swift 文件到现有的 Objective-C 应用（或反之）时，Xcode 会自动创建这些头文件。



如果你同意，Xcode 会在源文件创建的同时生成头文件，并用 product 的模块名加上 `-Bridging-Header.h` 命名。关于 product 的模块名，详见 Naming Your Product Module。

你应该编辑这个头文件来对 Swift 暴露出 Objective-C 代码。

在同一 target 中将 Objective-C 代码导入到 Swift 中

1. 在 Objective-C 桥接头文件中，import 任何你想暴露给 Swift 的头文件，例如：

```
// OBJECTIVE-C

#import "XYZCustomCell.h"
#import "XYZCustomView.h"
#import "XYZCustomViewController.h"
```

1. 确保在 `Build Settings` 中 Objective-C 桥接头文件的 `build setting` 是基于 Swfit 编译器，即 `Code Generation` 含有头文件的路径。这个路径必须是头文件自身的路径，而不是它所在的目录。

   这个路径应该是你工程的相对路径，类似 `Info.plist` 在 `Build Settings` 中指定的路径。在大多数情况下，你不需要修改这个设置。

   在这个桥接头文件中列出的所有 public 的 Objective-C 头文件都会对 Swift 可见。之后当前 target 的所有 Swift 文件都可以使用这些头文件中的方法，不需要任何 import 语句。用 Swift 语法使用这些 Objective-C 代码，就像使用系统自带的 Swift 类一样。

```
// SWIFT

let myCell = XYZCustomCell()
myCell.subtitle = "A custom cell"
```

将 Swift 导入 Objective-C

向 Objective-C 中导入 Swift 代码时，你依赖 Xcode 生成的头文件来向 Objective-C 暴露 Swift 代码。这是自动生成 Objective-C 头文件，它包含了你的 target 中所有 Swift 代码中定义的接口。可以把这个 Objective-C 头文件看作 Swift 代码的 `umbrella header`。它以 product 模块名加 `-Swift.h` 来命名。关于 product 的模块名，详见 Naming Your Product Module。

你不需要做任何事情来生成这个头文件，只需要将它导入到你的 Objective-C 代码来使用它。注意这个头文件中的 Swift 接口包含了它所使用到的所有 Objective-C 类型。如果你在 Swift 代码中使用你自己的 Objective-C 类型，确保先将对应的 Objective-C 头文件导入到你的 Swift 代码中，然后才将 Swift 自动生成的头文件导入到 Objective-C .m 源文件中来访问 Swift 代码。

在同一 target 中将 Swift 代码导入到 Objective-C 中

- 在相同 target 的 Objective-C .m 源文件中，用下面的语法来导入 Swift 代码：

```
// OBJECTIVE-C

#import "ProductModuleName-Swift.h"
```

target 中任何 Swift 文件将会对 Objective-C .m 源文件可见，包括这个 import 语句。关于在 Objective-C 代码中使用 Swift 代码，详见 Using Swift from Objective-C。

	导入到 Swift	导入到 Objc
Swift 代码	不需要import语句	#import
Objc 代码	不需要import语句；需要 Objc `umbrella头文件`	#import "Header.h"

在同个 Framework 的 Target 中导入代码

如果你在写一个混合语言的框架，可能会从 Swift 代码访问 Objective-C 代码，或者反之。

将 Objc 导入 Swift

要将一些 Objective-C 文件导入到同个框架 target 的 Swift 代码中去，你需要将这些文件导入到 Objective-C 的 `umbrella header` 来供框架使用。

在同一 framework 中将 Objective-C 代码导入到 Swift 中

确保将框架 target 的 `Build Settings > Packaging > Defines Module` 设置为 `Yes`。然后在你的 `umbrella header` 头文件中导入你想暴露给 Swift 访问的 Objective-C 头文件，例如：

```
1. // OBJECTIVE-C
2. #import <XYZ/XYZCustomCell.h>
3. #import <XYZ/XYZCustomView.h>
4. #import <XYZ/XYZCustomViewController.h>
```

Swift 将会看到所有你在 `umbrella header` 中公开暴露出来的头文件，框架 target 中的所有 Swift 文件都可以访问你 Objective-C 文件的内容，不需要任何 import 语句。

```
1. // SWIFT
2.
3. let myCell = XYZCustomCell()
4. myCell.subtitle = "A custom cell"
```

将 Swift 导入 Objc

要将一些 Swift 文件导入到同个框架的 target 的 Objective-C 代码去，你不需要导入任何东西到 `umbrella header` 文件，而是将 Xcode 为你的 Swift 代码自动生成的头文件导入到你的 Obj .m 源文件去，以便在 Objective-C 代码中访问 Swift 代码。

在同一 framework 中将 Swift 代码导入到 Objective-C 中

确保将框架 target 的 `Build Settings > Packaging` 中的 `Defines Module` 设置为 `Yes`。用下面的语法将 Swift 代码导入到同个框架 target 下的 Objective-C .m 源文件去。

```
// OBJECTIVE-C
#import <ProductName/ProductModuleName-Swift.h>
```

这个 import 语句所包含的 Swift 文件都可以被同个框架 target 下的 Objective-C .m 源文件访问。关于在 Objective-C 代码中使用 Swift 代码，详见 Using Swift from Objective-C。

	导入到 Swift	导入到 Objc
Swift 代码	不需要import语句	#import
Objc 代码	不需要import语句；需要 Objc `umbrella头文件`	#import "Header.h"

导入外部 Framework

你可以导入外部框架，不管这个框架是纯 Objective-C，纯 Swift，还是混合语言的。import 外部框架的流程都是一样的，不管这个框架是用一种语言写的，还是包含两种语言。当你导入外部框架时，确保 `Build Setting > Pakaging > Defines Module` 设置为 `Yes`。

用下面的语法将框架导入到不同 target 的 Swift 文件中：

```
1. // SWIFT
2.
3. import FrameworkName
```

用下面的语法将框架导入到不同 target 的 Objective-C .m 文件中：

```
1. // OBJECTIVE-C
2.
3. @import FrameworkName;
```

	导入到 Swift	导入到 Objc
任意语言框架	import FrameworkName	@import FrameworkName;

在 Objective-C 中使用 Swift

当你将 Swift 代码导入 Objective-C 文件之后，用普通的 Objective-C 语法使用 Swift 类。

```
// OBJECTIVE-C

MySwiftClass *swiftObject = [[MySwiftClass alloc] init];
[swiftObject swiftMethod];
```

Swift 的类或协议必须用 `@Objective-C attribute` 来标记，以便在 Objective-C 中可访问。这个 attribute 告诉编译器这个 Swift 代码可以从 Objective-C 代码中访问。如果你的 Swift 类是 Objective-C 类的子类，编译器会自动为你添加 `@Objective-C attribute`。详见 Swift Type Compatibility。
你可以访问 Swift 类或协议中用 `@Objective-C attribute` 标记过东西，只要它和 Objective-C 兼容。不包括一下这些 Swift 独有的特性：

- Generics - 范型
- Tuples - 元组

- Enumerations defined in Swift - Swift 中定义的枚举
- Structures defined in Swift - Swift 中定义的结构体
- Top-level functions defined in Swift - Swift Swift 中定义的顶层函数
- Global variables defined in Swift - Swift 中定义的全局变量
- Typealiases defined in Swift - Swift 中定义的类型别名
- Swift-style variadics - Swift 风格可变参数
- Nested types - 嵌套类型
- Curried functions - 柯里化后的函数

例如带有范型类型作为参数，或者返回元组的方法不能在 Objective-C 中使用。

为了避免循环引用，不要将 Swift 代码导入到 Objective-C 头文件中。但是你可以在 Objective-C 头文件中前向声明（`forward declare`）一个 Swift 类来使用它，然而，注意**不能在 Objective-C 中继承一个 Swift 类**。

在 Objective-C 头文件中引用 Swift 类

这样前向声明 Swift 类：

```
1. // OBJECTIVE-C
2. // MyObjective-CClass.h
3.
4. @class MySwiftClass;
5.
6. @interface MyObjective-CClass : NSObject
7. - (MySwiftClass *)returnSwiftObject;
8. /* ... */
9. @end
```

Product Module 模块命名

Xcode 为 Swift 代码生成的头文件的名称，以及 Xcode 创建的 Objective-C 桥接头文件名称，都是从你的 product 模块名生成的。默认你的 product 模块名和 product 名一样。然而，如果你的 product 名有特殊字符（nonalphanumeric，非数字、字母的字符），例如点号，那么它们会被下划线（_）替换之后作为你的 product 模块名。如果 product 名以数字开头，那么第一个数字会用下划线替换掉。

你可以给 product 模块名提供一个自定义的名称，Xcode 会用这个名称来命名桥接的和自动生成的头文件。你只需要在修改在 `build setting` 中的 `Product Module Name` 即可。

问题解决提示

- 把 Swift 和 Objective-C 文件看作相同的代码集合，并注意命名冲突；
- 如果你用框架，确保 `Build Setting > Pakaging > Defines Module` 设置为 `Yes`；
- 如果你使用 Objective-C 桥接头文件，确保在 `Build Settings` 中 Objective-C 桥接头文件的 `build setting` 是基于 Swfit 编译器，即 `Code Generation` 含有头文件的路径。这个路径必须是头文件自身的路径，而不是它所在的目录；
- Xcode 使用你的 product 模块名，而不是 target 名来命名 Objective-C 桥接头文件和为 Swift 自动生成的头文件。详见 Naming Your Product Module；
- 为了在 Objective-C 中可用， Swift 类必须是 Objective-C 类的子类，或者用 `@Objective-C` 标记；
- 当你将 Swift 导入到 Objective-C 中时，记住 Objective-C 不会将 Swift 独有的特性翻译成 Objective-C 对应的特性。详见列表 Using Swift from Objective-C；
- 如果你在 Swift 代码中使用你自己的 Objective-C 类型，确保先将对应的 Objective-C 头文件导入到你的 Swift 代码中，然后才将 Swift 自动生成的头文件 import 到 Objective-C .m 源文件中来访问 Swift 代码。

4.迁移

4.1. 将 Objective-C 代码迁至 Swift

本篇译者：@xudeheng （git 主页），敬请勘误。

迁移工作正好提供了一个重新审视现有 Objective-C 应用程序的机会，也可以通过 Swift 代码来更好的优化应用程序的体系架构，逻辑以及性能。直接的说，你将用先前学到的 mix and match 以及这两个语言间的互操作性来进行增量迁移工作。Mix-and-match 功能使得选择哪些特性和功能来用 Swift 来实现，哪些依然用 Objective-C 来实现变得简单。Swift 和 Objective-C 的互用性又使得将这些功能集成到 Objective-C 变得并不困难。通过这些工具可以开放 Swift 的扩展功能并集成到现有的 Objective-C 项目中而完全不必立刻使用 Swift 重写整个项目。

为你的 Objective-C 代码迁移做好准备

在开始迁移你的代码之前，请确保你的 Objective-C 和 Swift 代码间有着最佳兼容性。这意味着整理并使用 Objective-C 的现代化特性来优化你的现有项目。为了和 Swift 进行更容易的无缝交互，你的现有代码需要遵循现代编码实践。这里有个简短的适配练习列表，参看 Adopting Mordern Objective-C。

迁移过程

最有效迁移代码的方式是基于逐个文件的方式，即一次完成一个类。由于你不能在 Objective-C 中继承 Swift 类， 最好选择一个没有子类的（译者：从类的继承角度来看，应该先从类族树的叶子节点开始，自底向上的进行迁移操作）。你就可以用单个`.swift`文件来代替对应的`.m`和`.h`文件了。你所有的实现代码和接口将直接放进单个 Swift 文件。你不用再创建头文件了；Xcode 会在你需要引用的时候自动生成头文件。(译者：当然这实在 xcode 内部机制完成的了，对开发者是透明的)

准备工作

- 在 Xcode 中:File>New>File>(iOS 或者 OS X) > Other > Swift 为对应的 Objective-C `.m` 和 `.h` 文件创建一个 Swift 类。
- 导入相关系统框架。
- 如果你希望在 Swift 文件中访问 Objective-C 代码的话，可以填入一个 Objective-C 桥接头。具体的操作步骤，请看 Importing Code from Within the Same App Target。
- 为使你的 Swift 类能在 Objective-C 中访问使用，可以继承 Objective-C 类，或者标记上`@objc`属性。为类指定特殊的名称，以在 Objective-C 中使用，标记上`@objc(#name#)`, `<#name#>`就是在 Objective-C 中引用的 Swift 类名。 更多信息，请看 Swift Type Compatibility。

开始工作

- 你可以通过继承 Objective-C 类，适配 Objective-C 协议，或者更多的方式，来让 Swift 类集成 Objective-C 行为。更多信息，请看 Writing Swift Classes with Objective-C Behavior
- 当你使用 Objective-C APIs 的时候，你需要知道 Swift 是怎样来翻译某些 Objective-C 特性的。更多信息，请看 Interacting with Objective-C APIs
- 当用 Swift 编写用到 Cocoa 框架的代码时，记住某些类型是被桥接的，意味着你可以使用某些 Swift 类型来替代 Objective-C 类型。更多信息，请看 Working with Cocoa Data Types
- 当你在 Swift 中运用 Cocoa 设计模式得时候，请看 Adopting Cocoa Design Patterns 获取更多的通用设计模式的转换信息。
- 对于打算将项目从 Objective-C 转换到 Swfit 的人，请看 Propeties。
- 在必要的时候，请为 Swift 的属性或方法，通过`@objc(<#name#>)`属性来提供 Objective-C 名称，就像这样：

```
var enabled: Bool {
    @objc(isEnabled) get {
        /* ... */
    }
```

```
}
```

- 分别用 `func` 和 `class func` 来表示 instance(-) 和 class(+) 方法。
- 声明简单的宏来作为常量，将复杂的宏转换为函数。

大功告成

- 在你的 Objective-C 代码中更新 import 语句为 #import "模块名-Swift.h"，在 Importing Code from Within the Same App Target 中曾有提到。
- 在 Target 的成员选择框中去掉勾选框来移除原始的 Objective-C `.m` 文件.不要立刻删除 `.m` 和 `.h` 文件，以备解决问题用。
- 如果你给 Swift 类起了一个不同的名字，请使用 Swift 类名代替 Objective-C 名。

问题解决提示

对于不同的项目，迁移的经历是不尽相同的。无论怎样，都有一些通用的步骤和工具能帮你解决代码迁移时碰到的问题：

- 记住：你不能在 Objective-C 中继承 Swift 类。因此，被你迁移的类不能有任何的 Objective-C 子类存在于你的应用中。
- 当你迁移一个类到 Swift 的时候，你必须从 target 中移除相关的 `.m` 文件，以避免编译时提示出现重复的符号等编译错误。
- 为了在 Objective-C 中可以访问并使用，Swift 类必须是一个 Objective-C 类的子类，或者被标记为 `@objc`。
- 当你在 Objective-C 中使用 Swift 代码的时候，记住 Objective-C 不能理解那些 Swift 的某些特性，请看 Using Swift from Objective-C。
- 可以通过 `Commond + 点击` 一个 Swift 类名来查看他生成的头文件。
- 可以通过 `Option + 点击` 一个符号来查看更详细的信息，比如它的类型，属性以及文档注释等。