



SWIFTER

100 个 Swift 必备 Tips
第二版

王巍 (@onevcat)

目录

1. 介绍
2. Swift 新元素
 1. 柯里化 (Currying)
 2. Struct Mutable 的方法
 3. 将 protocol 的方法声明为 mutating
 4. Sequence
 5. tuple
 6. @autoclosure 和 ??
 7. Optional Chaining
 8. 操作符
 9. func 的参数修饰
10. 字面量转换
11. 下标
12. 方法嵌套
13. 命名空间
14. Any 和 AnyObject
15. typealias 和泛型接口
16. 可变参数函数
17. 初始化方法顺序
18. Designated, Convenience 和 Required
19. 初始化返回 nil
20. protocol 组合
21. static 和 class
22. 多类型和容器
23. default 参数
24. 正则表达式
25. 模式匹配
26. ... 和 ..<
27. AnyClass, 元类型和 .self
28. 接口和类方法中的 Self
29. 动态类型和多方法
30. 属性观察
31. final
32. lazy 修饰符和 lazy 方法
33. Reflection 和 Mirror
34. 隐式解包 Optional
35. 多重 Optional
36. Optional Map
37. Protocol Extension
38. where 和模式匹配
39. indirect 和嵌套 enum

3. 从 Objective-C/C 到 Swift

1. Selector
2. 实例方法的动态调用
3. 单例
4. 条件编译
5. 编译标记
6. @UIApplicationMain
7. @objc 和 dynamic
8. 可选接口和接口扩展
9. 内存管理, weak 和 unowned
10. @autoreleasepool
11. 值类型和引用类型
12. String 还是 NSString
13. UnsafePointer
14. C 指针内存管理
15. COpaquePointer 和 C convention
16. GCD 和延时调用
17. 获取对象类型
18. 自省
19. KVO
20. 局部 scope
21. 判等
22. 哈希
23. 类簇
24. Swizzle
25. 调用 C 动态库
26. 输出格式化
27. Options
28. 数组 enumerate
29. 类型编码 @encode
30. C 代码调用和 @asmname
31. sizeof 和 sizeofValue
32. delegate
33. Associated Object
34. Lock
35. Toll-Free Bridging 和 Unmanaged

4. Swift 与开发环境及一些实践

1. Swift 命令行工具
2. 随机数生成
3. print 和 debugPrint
4. 错误和异常处理
5. 断言
6. fatalError
7. 代码组织和 Framework
8. Playground 延时运行

- 9. Playground 可视化
- 10. Playground 与项目协作
- 11. 数学和数字
- 12. JSON
- 13. NSNull
- 14. 文档注释
- 15. 性能考虑
- 16. Log 输出
- 17. 溢出
- 18. 宏定义 define
- 19. 属性访问控制
- 20. Swift 中的测试
- 21. Core Data
- 22. 闭包歧义
- 23. 泛型扩展
- 24. 兼容性
- 25. 枚举 enum 类型
- 26. 尾递归
- 5. 后记
- 6. 版本更新

介绍

虽然我们都希望能尽快开始在 Swift 的世界里遨游，但是我觉得仍然有必要花一些时间将本书的写作目的和适合哪些读者进行必要说明。我不喜欢自吹自擂，也无法承担“骗子”的骂名。在知识这件严肃的事情上，我并不希望对读者产生任何的误导。作为读者，您一定想要找的是一本合适自己的书；而作为作者，我也希望找到自己的伯乐和子期。

为什么要写这本书

中文的科技书籍太少了，内容也太浅了。这是国内市场尴尬的现状：真正有技术的大牛不在少数，但他们很多并不太愿意通过出书的方式来分享他们的知识：一方面是回报率实在太低，另一方面是出版的流程过于繁琐。这就导致了市面上充斥了一些习惯于出版业务，但是却丝毫不顾质量和素质的流氓作者和图书。

特别是对于 Swift 语言来说，这个问题尤其严重。iOS 开发不可谓不火热，每天都有大量的开发者涌入这个平台。而 Swift 的发布更使得原本高温的市场更上一层楼。但是市面上随处可见的都是各种《开发指南》《权威指南》或者《21天学会XXX》系列的中文资料。这些图书大致都是对官方文档的翻译，并没有什么实质的见解，可以说内容单一，索然无味。作为读者，很难理解作者写作的重心和目的(其实说实话，大部分情况下这类书的作者自己都不知道写作的重心和目的是什么)，这样的“为了出版而出版”的图书可以说除了增加世界的熵以外，几乎毫无价值。

如果想要入门 Swift 语言，阅读 Apple 官方教程和文档无论从条理性和权威性来说，都是更好的选择。而中国的 Cocoa 开发者社区也以令人惊叹的速度完成了对文档的高品质翻译，这在其他任何国家都是让人眼红的一件事情。因此，如果您是初学程序设计或者 Swift 语言，相比起那些泯灭良心(抱歉我用了这个词，希望大家不要对号入座)的“入门书籍”，我更推荐您看这份[翻译后的官方文档](#)，这是非常珍惜和宝贵的资源。

说到这里，可以谈谈这本《Swifter - 100 个 Swift 必备 tips》的写作目的了。很多 Swift 的学习者 - 包括新接触 Cocoa/Cocoa Touch 开发的朋友，以及之前就使用 Objective-C 的朋友 -- 所共同面临的一个问题是，入门以后应该如何进一步提高。也许你也有过这样的感受：在阅读完 Apple 的教程后，觉得自己已经学会了 Swift 的语法和使用方式，你满怀信心地打开 Xcode，新建了一个 Swift 项目，想写点什么，却发现实际上满不是那么回事。你需要联想 Optional 应该在什么时候使用，你可能发现本已熟知 API 突然不太确定要怎么表达，你可能遇到怎么也编译不了的问题但却不知如何改正。这些现象都非常正常，因为教程是为了展示某个语法点而写的，而几乎不涉及实际项目中应该如何使用的范例。本书的目的就是为广大已经入门了 Swift 的开发者提供一些参考，以期能迅速提升他们在实践中的能力。因为这部分的中级内容是我自己力所能及，有自信心能写好的；也是现在广大 Swift 学习者所急缺和需要的。

这本书是什么

本书是 Swift 语言的知识点的集合。我自己是赴美参加了 Apple 的 WWDC 14 的，也正是在这届开发者大会上，Swift 横空出世。毫不夸张地说，从 Swift 正式诞生的第一分钟开始，我就在学习

这门语言。虽然天资驽钝，不得其所，但是在这段集中学习和实践的时间里，也还算总结了一些心得，而我把这些总结加以整理和示例，以一个个的小技巧和知识点的形式，编写成了这本书。全书共有 100 节，每一节都是一个相对独立的主题，涵盖了一个中高级开发人员需要知道的 Swift 语言的方方面面。

这本书非常适合用作官方文档的参考和补充，也会是中级开发人员很喜爱的 Swift 进阶读本。具体每个章节的内容，可以参看本书的目录。

这本书不是什么

这本书不是 Swift 的入门教程，也不会通过具体的完整实例引导你用 Swift 开发出一个像是计算器或者记事本这样的 app。这本书的目的十分纯粹，就是探索那些不太被人注意，但是又在每天的开发中可能经常用到的 Swift 特性。这本书并不会系统地介绍 Swift 的语法和特性，因为基于本书的写作目的和内容特点，采用松散的模式和非线性的组织方式会更加适合。

换言之，如果你想找一本 Swift 从零开始的书籍，那这本书不应该是你的选择。你可以在阅读 Apple 文档后再考虑回来看这本书。

组织形式和推荐的阅读方式

100 个 tips 其实不是一个小数目。本书的每个章节的内容是相对独立的，也就是说你没有必要从头开始看，随手翻开到任何一节都是没问题的。当然，按顺序看会是比较理想的阅读方式，因为在写作时我特别注意了让靠前的章节不涉及后面章节的内容；另一方面，位置靠后的章节如果涉及到之前章节内容的话，我添加了跳转到相关章节的链接，这可以帮助迅速复习和回顾之前的内容。我始终坚信不断的重复和巩固，是真正掌握知识的唯一途径。

本书的电子版的目录是可以点击跳转的，您可以通过目录快速地在不同章节之间导航。如果遇到您不感兴趣或者已经熟知的章节，您也完全可以暂时先跳过去，这不会影响您对本书的阅读和理解。

代码运行环境

建议您一边阅读本书时一边开启 Xcode 环境并且对每一章节中的代码进行验证，这有利于您真正理解代码示例想表达的意思，也有利于记忆的形成。随本书所附的 Playground 文件中有大部分章节的示例代码，以供参考。每一段代码示例都不太长，但却是经过精心准备，能很好地说明章节内容的，希望您能在每一章里都能通过代码和我进行心灵上的“对话”。您也可以在已有的基础上进行自己的探索，用来加深对讨论内容的理解。

书中每一章基本都配有代码示例的说明。这些代码一般来说包括 Objective-C 或者 Swift 的代码。理论上来说所有代码都可以在 Swift 2.0 (也就是 Xcode 7) 版本环境下运行。当然因为 Swift 版本变化很快，可能部分代码需要微调或者结合一定的上下文环境才能运行，但我相信这种调整是显而易见的。如果您发现明显的代码错误和无法运行的情况，欢迎到本书的 [issue 页面](#) 上提出，我将尽快修正。

如果没有特别说明，这些代码在 Playground 和项目中都应该可以运行，并拥有同样表现的。但是

也存在一些代码只能在 **Playground** 或者项目文件中才能正确工作的情况，这主要是因为平台限制的因素，如果出现这种情况，我都会在相关章节中特别加以说明。

勘误和反馈

Swift 仍然在高速发展和不断变化中，本书最早版本基于 **Swift 1.0**，当前版本基于 **Swift 2.0**。随着 Swift 的新特性引入以及错误修正，本书难免会存在部分错误，其中包括为对应的更新纰漏或者部分内容过时的情况。虽然我会随着 Swift 的发展继续不断完善和修正这本书，但是这个过程亦需要时间，请您谅解。

另外由于作者水平有限，书中也难免会出现一些错误，如果您在阅读时发现了任何问题，可以到这本书 [issue 页面](#) 进行反馈。我将尽快确认和修正。得益于电子书的优势，本书的读者都可以在本书更新时免费获得所有的新内容。每次更新的变更内容将会写在本书的[更新](#)一节中，您也可以[在更新内容页面](#)上找到同样的列表。

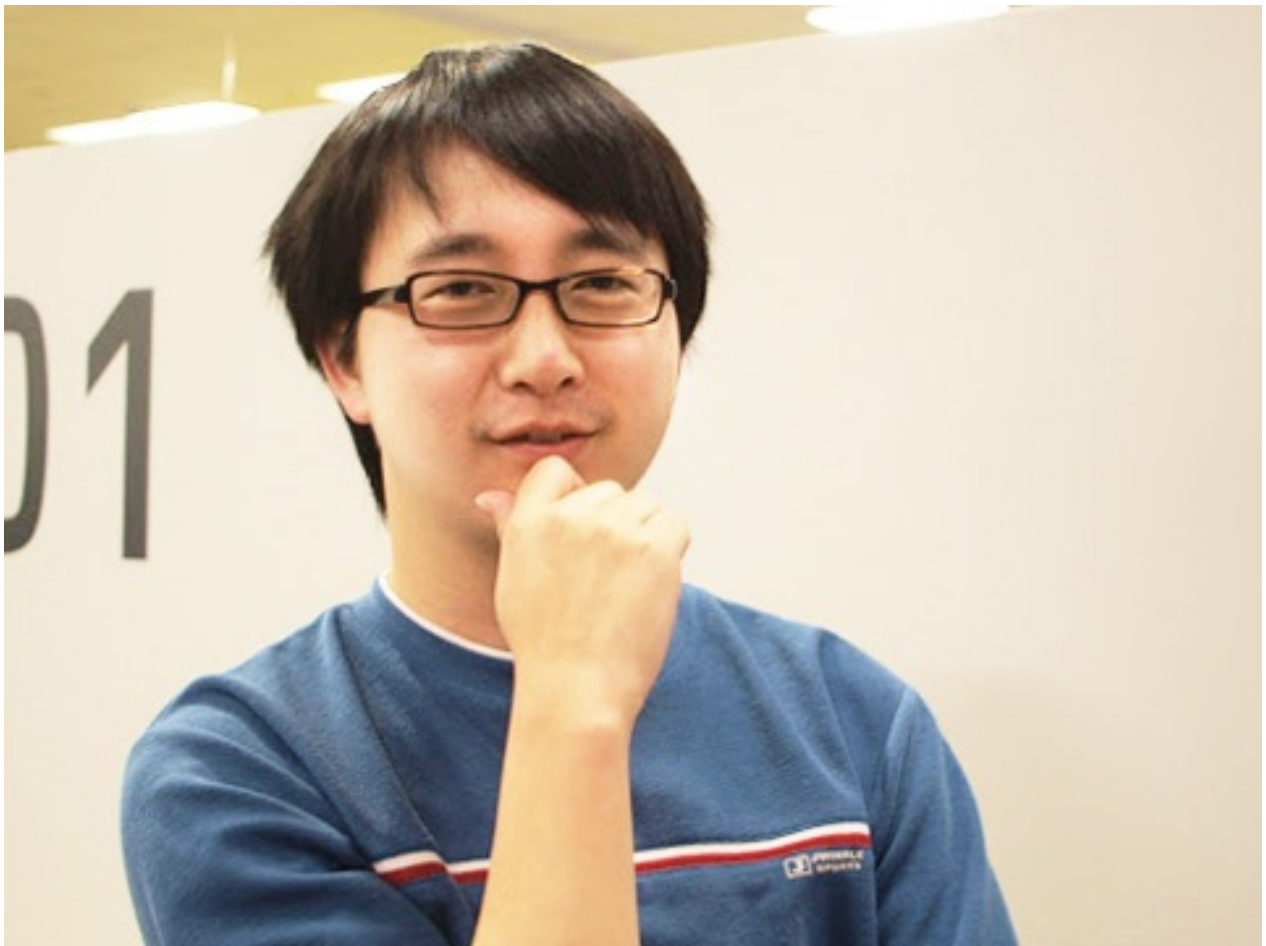
版权问题

为了方便读者使用和良好的阅读体验，本书不包含任何 **DRM 保护**。首先我在此想感谢您购买了这本书，在国内知识产权保护不足的现状下，我自知出版这样一本没有任何保护措施的电子书可能无异于飞蛾扑火。我其实是怀着忐忑的心情写下这些文字的，小心翼翼地希望没有触动到太多人。如果您不是通过 [Gumroad](#)，[Leanpub](#) 或者是 [SelfStore](#) 购买，而拿到这本书的话，您应该是盗版图书的受害者。这本书所提供的知识和之后的服务我想应该是超过它的售价 (大约是一杯星巴克咖啡的价格) 的，在阅读前还请您再三考虑。您的支持将是我继续更新和完善本书的动力，也对我继续前进是很大的鼓励。

另外，这本书也有纸质版本，但是暂时是面向 **Swift 1.2** 的。如果您有意阅读，可以搜索“**Swifter:100个Swift开发必备Tip**”来获取相关信息。

最后一部分是我的个人简介，您可以跳过不看，而直接开始[本书的第一章](#)。

作者简介



王巍 ([onevcats](#)) 是来自中国的一线 iOS 开发者，毕业于清华大学。在校期间就开始进行 iOS 开发，拥有丰富的 Cocoa 和 Objective-C 开发经验，另外他也活跃于使用 C# 的 Unity3D 游戏开发界。曾经开发了《小熊推金币》，《Pomo Do》等一系列优秀的 iOS 游戏和应用。在业余时间，王巍会在 [OneV's Den](#) 撰写博客，分享他在开发中的一些心得和体会。另外，王巍还是翻译项目 [objc 中国](#) 的组织者和管理者，为中国的 Objective-C 社区的发展做出了贡献。同时，他也很喜欢为[开源社区](#)贡献代码，是著名的 Xcode 插件 [VVDocumenter](#) 和开源库 [Kingfisher](#) 的作者。

现在王巍旅居日本，并就职于即时通讯软件公司 Line，从事 iOS 开发工作，致力于为全世界带来更好体验和功能的应用。如果您需要进一步了解作者的话，可以访问他的[资料页面](#)。

1. Swift 新元素

柯里化 (Currying)

Swift 里可以将方法进行柯里化 (Currying)，也就是把接受多个参数的方法变换成接受第一个参数的方法，并且返回接受余下的参数并且返回结果的新方法。举个例子，Swift 中我们可以这样写出多个括号的方法：

```
func addTwoNumbers(a: Int)(num: Int) -> Int {  
    return a + num  
}
```

然后通过只传入第一个括号内的参数进行调用，这样将返回另一个方法：

```
let addToFour = addTwoNumbers(4)    // addToFour 是一个 Int -> Int  
let result = addToFour(num: 6)      // result = 10
```

或者

```
func greaterThan(comparator: Int)(input : Int) -> Bool{  
    return input > comparator;  
}  
  
let greaterThan10 = greaterThan(10);  
  
greaterThan10(input : 13)    // 结果是 true  
greaterThan10(input : 9)    // 结果是 false
```

柯里化是一种量产相似方法的好办法，可以通过柯里化一个方法模板来避免写出很多重复代码，也方便了今后维护。

举一个实际应用时候的例子，在 [Selector](#) 一节中，我们提到了在 Swift 中 `Selector` 只能使用字符串在生成。这面临一个很严重的问题，就是难以重构，并且无法在编译期间进行检查，其实这是十分危险的行为。但是 `target-action` 又是 Cocoa 中如此重要的一种设计模式，无论如何我们都想安全地使用的话，应该怎么办呢？一种可能的解决方式就是利用方法的柯里化。Ole Begemann 在[这篇帖子](#)里提到了一种很好封装，这为我们如何借助柯里化，安全地改造和利用 `target-action` 提供了不少思路。

```
protocol TargetAction {  
    func performAction()  
}  
  
struct TargetActionWrapper<T: AnyObject>:  
    TargetAction {  
    weak var target: T?  
    let action: (T) -> () -> ()
```

```

    func performAction() -> () {
        if let t = target {
            action(t)()
        }
    }
}

enum ControlEvent {
    case TouchUpInside
    case ValueChanged
    // ...
}

class Control {
    var actions = [ControlEvent: TargetAction]()

    func setTarget<T: AnyObject>(target: T,
                                  action: (T) -> () -> (),
                                  controlEvent: ControlEvent) {

        actions[controlEvent] = TargetActionWrapper(
            target: target, action: action)
    }

    func removeTargetForControlEvent(controlEvent: ControlEvent) {
        actions[controlEvent] = nil
    }

    func performActionForControlEvent(controlEvent: ControlEvent) {
        actions[controlEvent]?.performAction()
    }
}

```

Struct Mutable 的方法

在 Swift 中我们基本都是用 `struct` 去定义一个纯数据类型。比如

```
struct User {  
    var age : Int  
    var weight : Int  
    var height : Int  
}
```

而如果我们经常会在变量里添加一些简单的改变变量里内容的方法，比如

```
func gainWeight(newWeight: Int) {  
    weight += newWeight  
}
```

但如果我们直接把这个方法放入 `User` 这个变量中间，你的编译器就会给出一个非常奇怪的错误消息

```
struct User {  
    var age : Int  
    var weight : Int  
    var height : Int  
  
    func gainWeight(newWeight: Int) {  
        weight += newWeight  
    }  
}
```

```
struct User {  
    var age : Int  
    var weight : Int  
    var height : Int  
  
    func gainWeight(newWeight: Int)  
    {  
        weight += newWeight;  
    }  
}
```

❗ Binary operator '+' cannot be applied to two Int operands

错误的消息竟然是 `Binary operator '+' cannot be applied to two int operands`，意思是在2个数字之间没有办法使用 `+=`。但大家都知道 `+=` 在数字变量之间是最常见不过的，为什么不能用？

其实在这个错误代码后面有另一层意思。因为我们忽略了一点，**Struct** 出来的变量是 **Immutable** 的，想要用一个方法去改变变量里面的值的时候必须要加上一个关键词 `mutating`，所以其实这个错误代码的真正含义应该是 因为 `User` 的 `Weight` 是 `Immutable` 的，所以 `+=` 无法在这两个 `Int` 上使用

我们在方法之前加上 `mutating` 之后编译就可以顺利进行了

```
struct User {  
    var age : Int  
    var weight : Int  
    var height : Int  
  
    mutating func gainWeight(newWeight: Int) {  
        weight += newWeight  
    }  
}
```

<pre>var newUser = User(age: 15, weight: 150, height: 170) newUser.gainWeight(100)</pre>	<pre>{age 15, weight 150, height 170} {age 15, weight 250, height 170}</pre>
--	--

将 protocol 的方法声明为 mutating

Swift 的 protocol 不仅可以被 class 类型实现，也适用于 struct 和 enum。因为这个原因，我们在写给别人用的接口时需要多考虑是否使用 mutating 来修饰方法，比如定义为 mutating func myMethod()。Swift 的 mutating 关键字修饰方法是为了能在该方法中修改 struct 或是 enum 的变量，所以如果你没在接口方法里写 mutating 的话，别人如果用 struct 或者 enum 来实现这个接口的话，就不能在方法里改变自己的变量了。比如下面的代码

```
protocol Vehicle
{
    var numberOfWheels: Int {get}
    var color: UIColor {get set}

    mutating func changeColor()
}

struct MyCar: Vehicle {
    let numberOfWheels = 4
    var color = UIColor.blueColor()

    mutating func changeColor() {
        color = UIColor.redColor()
    }
}
```

如果把 protocol 定义中的 mutating 去掉的话，MyCar 就怎么都过不了编译了：保持现有代码不变的话，会报错说没有实现接口；如果去掉 mutating 的话，会报错说不能改变结构体成员。这个接口的使用者的忧伤的眼神，相信你能想象得出。

另外，在使用 class 来实现带有 mutating 的方法的接口时，具体实现的前面是不需要加 mutating 修饰的，因为 class 可以随意更改自己的成员变量。所以说在接口里用 mutating 修饰方法，对于 class 的实现是完全透明，可以当作不存在的。

Sequence

Swift 的 `for...in` 可以用在所有实现了 `SequenceType` 的类型上，而为了实现 `SequenceType` 你首先需要实现一个 `GeneratorType`。比如一个实现了反向的 `generator` 和 `sequence` 可以这么写：

```
// 先定义一个实现了 GeneratorType protocol 的类型
// GeneratorType 需要指定一个 typealias Element
// 以及提供一个返回 Element? 的方法 next()
class ReverseGenerator: GeneratorType {
    typealias Element = Int

    var counter: Element
    init<T>(array: [T]) {
        self.counter = array.count - 1
    }

    init(start: Int) {
        self.counter = start
    }

    func next() -> Element? {
        return self.counter < 0 ? nil : counter--
    }
}

// 然后我们来定义 SequenceType
// 和 GeneratorType 很类似，不过换成指定一个 typealias Generator
// 以及提供一个返回 Generator? 的方法 generate()
struct ReverseSequence<T>: SequenceType {
    var array: [T]

    init (array: [T]) {
        self.array = array
    }

    typealias Generator = ReverseGenerator
    func generate() -> Generator {
        return ReverseGenerator(array: array)
    }
}

let arr = [0,1,2,3,4]

// 对 SequenceType 可以使用 for...in 来循环访问
for i in ReverseSequence(array: arr) {
    print("Index \(i) is \(arr[i])")
}
```

输出为

```
Index 4 is 4
Index 3 is 3
Index 2 is 2
Index 1 is 1
Index 0 is 0
```


如果我们想要深究 `for...in` 这样的方法到底做了什么的话，如果我们将其展开，大概会是下面这个样子：

```
var g = array.generate()
while let obj = g.next() {
    print(obj)
}
```

顺便你可以免费得到的收益是你可以使用像 `map`，`filter` 和 `reduce` 这些方法，因为 `SequenceType` 接口扩展 (protocol extension) 已经实现了它们：

```
extension SequenceType {
    func map<T>(@noescape transform: (Self.Generator.Element) -> T) -> [T]
    func filter(@noescape includeElement: (Self.Generator.Element) -> Bool)
        -> [Self.Generator.Element]

    func reduce<T>(initial: T,
        @noescape combine: (T, Self.Generator.Element) -> T) -> T
}
```

多元组 (Tuple)

多元组是我们的新朋友，多尝试使用这个新特性吧，会让生活轻松不少～

比如交换输入，普通程序员亘古以来可能都是这么写的

```
func swapMe<T>(inout a: T, inout b: T) {  
    let temp = a  
    a = b  
    b = temp  
}
```

但是要是使用多元组的话，我们可以不使用额外空间就完成交换，一下子就达到了文艺程序员的写法：

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

在 **Objective-C** 中有不少需要传递指针的地方，以前的错误处理 `NSError` 是个很好的例子。但是在 **Swift 2.0** 中传入指针的 `NSError` 已经被新加入的异常机制取代的，所以我们在这里举另一个例子来说明多元组的应用。在 **Objective-C** 中 `CGRect` 有一个辅助方法叫做 `CGRectDivide`，它用来将一个 `CGRect` 在一定位置切分成两个区域。具体定义和用法如下：

```
/*  
CGRectDivide(CGRect rect, CGRect *slice, CGRect *remainder,  
             CGFloat amount, CGRectEdge edge)  
*/  
CGRect rect = CGRectMake(0, 0, 100, 100);  
CGRect small;  
CGRect large;  
CGRectDivide(rect, &small, &large, 20, CGRectMinXEdge);
```

上面的代码将 `{0,0,100,100}` 的 `rect` 分割为两部分，分别是 `{0,0,20,100}` 的 `small` 和 `{20,0,80,100}` 的 `large`。由于 C 系语言的单一返回，我们不得不通过传入指针的方式让方法来填充需要的部分，可以说使用起来既不直观，又很麻烦。

而现在在 **Swift** 中，这个方法摇身一变，使用了多元组的方式来同时返回被分割的部分和剩余部分：

```
func rectsByDividing(atDistance: CGFloat, fromEdge: CGRectEdge)  
-> (slice: CGRect, remainder: CGRect)
```

然后使用的时候，对比之前的做法，现在就非常简单并且易于理解了：

```
let rect = CGRectMake(0, 0, 100, 100)
let (small, large) = rect.rectsByDividing(20, fromEdge: .MinXEdge)
```

一个有趣但是不被注意的事实，其实在 Swift 中任何东西都是放在多元组里的

不相信？试试看输出这个吧

```
var num = 42
print(num)
print(num.0.0.0.0.0.0.0.0.0.0)
```

@autoclosure 和 ??

Apple 为了推广和介绍 Swift，破天荒地为此语言开设了一个[博客](#)(当然我觉得是因为 Swift 坑太多需要一个地方来集中解释)。其中[有一篇](#)提到了一个叫做 `@autoclosure` 的关键词。

`@autoclosure` 可以说是 Apple 的一个非常神奇的创造，因为这更多地是像在 “hack” 这门语言。简单说，`@autoclosure` 做的事情就是把一句表达式自动地封装成一个闭包 (closure)。这样有时候在语法上看起来就会非常漂亮。

比如我们有一个方法接受一个闭包，当闭包执行的结果为 `true` 的时候进行打印：

```
func logIfTrue(predicate: () -> Bool) {
    if predicate() {
        print("True")
    }
}
```

在调用的时候，我们需要写这样的代码

```
logIfTrue({return 2 > 1})
```

当然，在 Swift 中对闭包的用法可以进行一些简化，在这种情况下我们可以省略掉 `return`，写成：

```
logIfTrue({2 > 1})
```

还可以更进一步，因为这个闭包是最后一个参数，所以可以使用尾随闭包 (trailing closure) 的方式把大括号拿出来，然后省略括号，变成：

```
logIfTrue{2 > 1}
```

但是不管哪种方式，要么是书写起来十分麻烦，要么是表达上不太清晰，看起来都让人生气。于是 `@autoclosure` 登场了。我们可以改换方法参数，在参数名前面加上 `@autoclosure` 关键字：

```
func logIfTrue(@autoclosure predicate: () -> Bool) {
    if predicate() {
        println("True")
    }
}
```

这时候我们就可以直接写：

```
logIfTrue(2 > 1)
```

来进行调用了，Swift 将会把 `2 > 1` 这个表达式自动转换为 `() -> Bool`。这样我们就得到了一个写法简单，表意清楚的式子。

在 Swift 中，有一个非常有用的操作符，可以用来快速地对 `nil` 进行条件判断，那就是 `??`。这个操作符可以判断输入并在当左侧的值是非 `nil` 的 `Optional` 值时返回其 `value`，当左侧是 `nil` 时返回右侧的值，比如：

```
var level : Int?
var startLevel = 1

var currentLevel = level ?? startLevel
```

在这个例子中我们没有设置过 `level`，因此最后 `startLevel` 被赋值给了 `currentLevel`。如果我们充满好奇心地点进 `??` 的定义，可以看到 `??` 有两种版本：

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

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

在这里我们的输入满足的是后者，虽然表面上看 `startLevel` 只是一个 `Int`，但是其实在使用时它被自动封装成了一个 `() -> Int`，有了这个提示，我们不妨来猜测一下 `??` 的实现吧：

```
func ??<T>(optional: T?, @autoclosure defaultValue: () -> T) -> T {
    switch optional {
        case .Some(let value):
            return value
        case .None:
            return defaultValue()
    }
}
```

可能你会有疑问，为什么这里要使用 `autoclosure`，直接接受 `T` 作为参数并返回不行么？这正是 `autoclosure` 的一个最值得称赞的地方。如果我们直接使用 `T`，那么就意味着在 `??` 操作符真正取值之前，我们就必须准备好一个默认值，这个默认值的准备和计算是会消耗性能的。但是其实要是 `optional` 不是 `nil` 的话，我们是完全不需要这个默认值，而会直接返回 `optional` 解包后的值。这样一来，默认值就白白准备了，这样的开销是完全可以避免的，方法就是将默认值的计算推迟到 `optional` 判定为 `nil` 之后。

就这样，我们可以巧妙地绕过条件判断和强制转换，以很优雅的写法处理对 `Optional` 及默认值的取值了。最后要提一句的是，`@autoclosure` 并不支持带有输入参数的写法，也就是说只有形如 `() -> T` 的参数才能使用这个特性进行简化。另外因为调用者往往很容易忽视 `@autoclosure` 这个特性，所以在写接受 `@autoclosure` 的方法时还请特别小心，如果在容易产生歧义或者误解的时候，还是使用完整的闭包写法会比较好。

在 Swift 1.2 中，`@autoclosure` 的位置发生了变化。现在 `@autoclosure` 需要像本文中这样，写在参数名的前面作为参数修饰，而不是在类型前面作为类型修饰。但是现在标准库中的方法签名还是写在了接受的类型前面，这应该是标准库中的疏漏。在我们自己实现一个 `autoclosure` 时，在类型前修饰的写法在 Swift 1.2 中已经无法编译了。

练习

在 Swift 中，其实 `&&` 和 `||` 这两个操作符里也用到了 `@autoclosure`。作为练习，不妨打开 Playground，试试看怎么实现这两个操作符吧？

Optional Chaining

使用 Optional Chaining 可以让我们摆脱很多不必要的判断和取值，但是在使用的时候需要小心陷阱。

因为 Optional Chaining 是随时都可能提前返回 `nil` 的，所以使用 Optional Chaining 所得到的东西其实都是 Optional 的。比如有下面的一段代码：

```
class Toy {
  let name: String
  init(name: String) {
    self.name = name
  }
}

class Pet {
  var toy: Toy?
}

class Child {
  var pet: Pet?
}
```

在实际使用中，我们想要知道小明的宠物的玩具的名字的时候，可以通过下面的 Optional Chaining 拿到：

```
let toyName = xiaoming.pet?.toy?.name
```

注意虽然我们最后访问的是 `name`，并且在 `Toy` 的定义中 `name` 是被定义为一个确定的 `String` 而非 `String?` 的，但是我们拿到的 `toyName` 其实还是一个 `String?` 的类型。这是由于在 Optional Chaining 中我们在任意一个 `?.` 的时候都可能遇到 `nil` 而提前返回，这个时候当然就只能拿到 `nil` 了。

在实际的使用中，我们大多数情况下可能更希望使用 Optional Binding 来直接取值的这样的代码：

```
if let toyName = xiaoming.pet?.toy?.name {
  // 太好了，小明既有宠物，而且宠物还正好有个玩具
}
```

可能单独拿出来看会很清楚，但是只要稍微和其他特性结合一下，事情就会变得复杂起来。来看看下面的例子：


```
extension Toy {
    func play() {
        //...
    }
}
```

我们为 `Toy` 定义了一个扩展，以及一个玩玩具的 `play()` 方法。还是拿小明举例子，要是玩具的话，就玩之：

```
xiaoming.pet?.toy?.play()
```

除了小明也许我们还有小红小李小张等等..在这种时候我们会想要把这一串调用抽象出来，做一个闭包方便使用。传入一个 `Child` 对象，如果小朋友有宠物并且宠物有玩具的话，就去玩。于是很可能你会写出这样的代码：

这是错误代码

```
let playClosure = {(child: Child) -> () in child.pet?.toy?.play()}
```

这样的代码是没有意义的！

问题在于对于 `play()` 的调用上。定义的时候我们没有写 `play()` 的返回，这表示这个方法返回 `Void` (或者写作一对小括号 `()`，它们是等价的)。但是正如上所说，经过 **Optional Chaining** 以后我们得到的是一个 **Optional** 的结果。也就是说，我们最后得到的应该是这样一个 closure：

```
let playClosure = {(child: Child) -> ()? in child.pet?.toy?.play()}
```

这样调用的返回将是一个 `()?` (或者写成 `Void?` 会更清楚一些)，虽然看起来挺奇怪的，但这就是事实。使用的时候我们可以通过 **Optional Binding** 来判定方法是否调用成功：

```
if let result: () = playClosure(xiaoming) {
    print("好开心~")
} else {
    print("没有玩具可以玩 :(")
}
```

操作符

与 Objective-C 不同，Swift 支持重载操作符这样的特性，最常见的使用方式可能就是定义一些简便的计算了。比如我们需要一个表示二维向量的数据结构：

```
struct Vector2D {  
    var x = 0.0  
    var y = 0.0  
}
```

一个很简单的需求是两个 `Vector2D` 相加：

```
let v1 = Vector2D(x: 2.0, y: 3.0)  
let v2 = Vector2D(x: 1.0, y: 4.0)  
let v3 = Vector2D(x: v1.x + v2.x, y: v1.y + v2.y)  
// v3 为 {x 3.0, y 7.0}
```

如果只做一次的话似乎还好，但是一般情况我们会进行很多这种操作。这样的话，我们可能更愿意定义一个 `Vector2D` 相加的操作，来让代码简化清晰。

对于两个向量相加，我们可以重载加号操作符：

```
func +(left: Vector2D, right: Vector2D) -> Vector2D {  
    return Vector2D(x: left.x + right.x, y: left.y + right.y)  
}
```

这样，上面的 `v3` 以及之后的所有表示两个向量相加的操作就全部可以用加号来表达了：

```
let v4 = v1 + v2  
// v4 为 {x 3.0, y 7.0}
```

类似地，我们还可以为 `Vector2D` 定义像 `-` (减号，两个向量相减)，`-` (负号，单个向量 `x` 和 `y` 同时取负) 等等这样的运算符。这个就作为练习交给大家。

上面定义的加号，减号和负号都是已经存在于 Swift 中的运算符了，我们所做的只是变换它的参数进行重载。如果我们想要定义一个全新的运算符的话，要做的事情会多一件。比如[点积运算](#)就是一个在矢量运算中很常用的运算符，它表示两个向量对应坐标的乘积的和。根据定义，以及参考重载运算符的方法，我们选取 `+` 来表示这个运算的话，不难写出：

```
func +* (left: Vector2D, right: Vector2D) -> Double {
```

```
    return left.x * right.x + left.y * right.y
}
```

但是编译器会给我们一个错误：

Operator implementation without matching operator declaration

这是因为我们没有对这个操作符进行声明。之前可以直接重载像 `+`，`-`，`*` 这样的操作符，是因为 `Swift` 中已经有定义了，如果我们要新加操作符的话，需要先对其进行声明，告诉编译器这个符号其实是一个操作符。添加如下代码：

```
infix operator +* {
    associativity none
    precedence 160
}
```

infix

表示要定义的是一个中位操作符，即前后都是输入；其他的修饰子还包括 `prefix` 和 `postfix`，不再赘述；

associativity

定义了结合律，即如果多个同类的操作符顺序出现的计算顺序。比如常见的加法和减法都是 `left`，就是说多个加法同时出现时按照从左往右的顺序计算（因为加法满足交换律，所以这个顺序无所谓，但是减法的话计算顺序就很重要了）。点乘的结果是一个 `Double`，不再会和其他点乘结合使用，所以这里写成 `none`；

precedence

运算的优先级，越高的话越优先进行运算。`Swift` 中乘法和除法的优先级是 150，加法和减法是 140，这里我们定义点积优先级 160，就是说应该早于普通的乘除进行运算。

有了这些之后，我们就可以很简单地进行向量的点积运算了：

```
let result = v1 +* v2
// 输出为 14.0
```

最后需要多提一点的是，`Swift` 的操作符是不能定义在局部域中的，因为至少会希望在能在全局范围内使用你的操作符，否则操作符也就失去意义了。另外，来自不同 `module` 的操作符是有可能冲突的，这对于库开发者来说是需要特别注意的地方。如果库中的操作符冲突的话，使用者是无法像解决类型名冲突那样通过指定库名字来进行调用的。因此在重载或者自定义操作符时，应当尽量将其作为其他某个方法的“简便写法”，而避免在其中实现大量逻辑或者提供独一无二的功能。这样即使出现了冲突，使用者也还可以通过方法名调用的方式使用你的库。运算符的命名也应当

尽量明了，避免歧义和可能的误解。因为一个不被公认的操作符是存在冲突风险和理解难度的，所以我们不应该滥用这个特性。在使用重载或者自定义操作符时，请先再三权衡斟酌，你或者你的用户是否真的需要这个操作符。

func 的参数修饰

在声明一个 Swift 的方法的时候，我们一般不去指定参数前面的修饰符，而是直接声明参数：

```
func incrementor(variable: Int) -> Int {  
    return variable + 1  
}
```

这个方法接受一个 `Int` 的输入，然后通过将这个输入加 1，返回一个新的比输入大 1 的 `Int`。嘛，就是一个简单的 **+1器**。

有些同学在大学的 C 程序设计里可能学过像 `++` 这样的“自增”运算符，再加上做了不少关于“判断一个数被各种前置 `++` 和后置 `++` 折磨后的输出是什么”的考试题，所以之后写代码时也会不自觉地喜欢带上这种风格。于是同样的功能可能会写出类似这样的方法：

这是错误代码

```
func incrementor(variable: Int) -> Int {  
    return ++variable  
}
```

残念..编译错误。为什么在 Swift 里这样都不行呢？答案是因为 Swift 其实是一门讨厌变化的语言。所有有可能的地方，都被默认认为是不可变的，也就是用 `let` 进行声明的。这样不仅可以确保安全，也能在编译器的性能优化上更有作为。在方法的参数上也是如此，我们不写修饰符的话，默认情况下所有参数都是 `let` 的，上面的代码等效为：

```
func incrementor(let variable: Int) -> Int {  
    return ++variable  
}
```

`let` 的参数，不能重新赋值这是理所当然的。要让这个方法正确编译，我们需要做的改动是将 `let` 改为 `var`：

```
func incrementor(var variable: Int) -> Int {  
    return ++variable  
}
```

现在我们的 **+1器** 又可以正确工作了：

```
var luckyNumber = 7
let newNumber = incrementor(luckyNumber)
// newNumber = 8

print(luckyNumber)
// luckyNumber 还是 7
```

正如上面的例子，我们将参数写作 `var` 后，通过调用返回的值是正确的，而 `luckyNumber` 还是保持了原来的值。这说明 `var` 只是在方法内部作用，而不直接影响输入的值。有些时候我们会希望在方法内部直接修改输入的值，这时候我们可以使用 `inout` 来对参数进行修饰：

```
func incrementor(inout variable: Int) {
    ++variable
}
```

因为在函数内部就更改了值，所以也不需要返回了。调用也要改变为相应的形式，在前面加上 `&` 符号：

```
var luckyNumber = 7
incrementor(&luckyNumber)

println(luckyNumber)
// luckyNumber = 8
```

最后，要注意的是参数的修饰是具有传递限制的，就是说对于跨越层级的调用，我们需要保证同一参数的修饰是统一的。举个例子，比如我们想扩展一下上面的方法，实现一个可以累加任意数字的 **+N器** 的话，可以写成这样：

```
func makeIncrementor(addNumber: Int) -> ((inout Int) -> ()) {
    func incrementor(inout variable: Int) -> () {
        variable += addNumber;
    }
    return incrementor;
}
```

外层的 `makeIncrementor` 的返回里也需要在参数的类型前面明确指出修饰词，以符合内部的定义，否则将无法编译通过。

字面量转换

所谓字面量，就是指像特定的数字，字符串或者是布尔值这样，能够直接了当地指出自己的类型并为变量进行赋值的值。比如下面：

```
let aNumber = 3
let aString = "Hello"
let aBool = true
```

中的 `3`，`Hello` 以及 `true` 就称为字面量。

在 Swift 中，`Array` 和 `Dictionary` 在使用简单的描述赋值的时候，使用的也是字面量，比如：

```
let anArray = [1, 2, 3]
let aDictionary = ["key1": "value1", "key2": "value2"]
```

Swift 为我们提供了一组非常有意思的接口，用来将字面量转换为特定的类型。对于那些实现了字面量转换接口的类型，在提供字面量赋值的时候，就可以简单地按照接口方法中定义的规则“无缝对应”地通过赋值的方式将值转换为对应类型。这些接口包括了各个原生的字面量，在实际开发中我们经常可能用到的有：

- `ArrayLiteralConvertible`
- `BooleanLiteralConvertible`
- `DictionaryLiteralConvertible`
- `FloatLiteralConvertible`
- `NilLiteralConvertible`
- `IntegerLiteralConvertible`
- `StringLiteralConvertible`

所有的字面量转换接口都定义了一个 `typealias` 和对应的 `init` 方法。拿 `BooleanLiteralConvertible` 举个例子：

```
protocol BooleanLiteralConvertible {
    typealias BooleanLiteralType

    /// Create an instance initialized to `value`.
    init(booleanLiteral value: BooleanLiteralType)
}
```

在这个接口中，`BooleanLiteralType` 在 Swift 标准库中已经有定义了：


```
/// The default type for an otherwise-unconstrained boolean literal
typealias BooleanLiteralType = Bool
```

于是在我们需要自己实现一个字面量转换的时候，可以简单地只实现定义的 `init` 方法就行了。举个不太有实际意义的例子，比如我们想实现一个自己的 `Bool` 类型，可以这么做：

```
enum MyBool: Int {
    case myTrue, myFalse
}

extension MyBool: BooleanLiteralConvertible {
    init(booleanLiteral value: Bool) {
        self = value ? myTrue : myFalse
    }
}
```

这样我们就能很容易地直接使用 `Bool` 的 `true` 和 `false` 来对 `MyBool` 类型进行赋值了：

```
let myTrue: MyBool = true
let myFalse: MyBool = false

myTrue.rawValue    // 0
myFalse.rawValue   // 1
```

`BooleanLiteralType` 大概是最简单的形式，如果我们深入一点，就会发现像是 `StringLiteralConvertible` 这样的接口要复杂一些。这个接口不仅类似于上面布尔的情况，定义了 `StringLiteralType` 及接受其的初始化方法，这个接口本身还要求实现下面两个接口：

```
ExtendedGraphemeClusterLiteralConvertible
UnicodeScalarLiteralConvertible
```

这两个接口我们在日常项目中基本上不会使用，它们对应[字符簇](#)和[字符](#)的字面量转换。虽然复杂一些，但是形式上还是一致的，只不过在实现 `StringLiteralConvertible` 时我们需要将这三个 `init` 方法都进行实现。

还是以例子来说明，比如我们有个 `Person` 类，里面有这个人的名字：

```
class Person {
    let name: String
    init(name value: String) {
        self.name = value
    }
}
```

如果想要通过 `String` 赋值来生成 `Person` 对象的话，可以改写这个类：

```

class Person: StringLiteralConvertible {
    let name: String
    init(name value: String) {
        self.name = value
    }

    required init(stringLiteral value: String) {
        self.name = value
    }

    required init(extendedGraphemeClusterLiteral value: String) {
        self.name = value
    }

    required init(unicodeScalarLiteral value: String) {
        self.name = value
    }
}

```

在所有的接口定义的 `init` 前面我们都加上了 `required` 关键字，这是由[初始化方法的完备性需求](#)所决定的，这个类的子类都需要保证能够做类似的字面量转换，以确保类型安全。

在上面的例子里有很多重复的对 `self.name` 赋值的代码，这是我们所不乐见的。一个改善的方式是在这些初始化方法中去调用原来的 `init(name value: String)`，这种情况下我们需要在这些初始化方法前加上 `convenience`：

```

class Person: StringLiteralConvertible {
    let name: String
    init(name value: String) {
        self.name = value
    }

    required convenience init(stringLiteral value: String) {
        self.init(name: value)
    }

    required convenience init(extendedGraphemeClusterLiteral value: String) {
        self.init(name: value)
    }

    required convenience init(unicodeScalarLiteral value: String) {
        self.init(name: value)
    }
}

let p: Person = "xiaoMing"
print(p.name)

// 输出:
// xiaoMing

```

上面的 `Person` 的例子中，我们没有像 `MyBool` 中做的那样，使用一个 `extension` 的方式来扩展类使其可以用字面量赋值，这是因为在 `extension` 中，我们是不能定义 `required` 的初始化方法的。也就是说，我们无法为现有的非 `final` 的 `class` 添加字面量转换 (不过也许这在今后的 `Swift` 版本中能有所改善)。

总结一下，字面量转换是一个很强大的特性，使用得当的话对缩短代码和清晰表意都很有帮助；但是这同时又是一个比较隐蔽的特性：因为你的代码并没有显式的赋值或者初始化，所以可能会给人造成迷惑：比如上面例子中为什么一个字符串能被赋值为 `Person`？你的同事在阅读代码的时候可能不得不去寻找这些负责字面量转换的代码进行查看 (而如果代码库很大的话，这不是一件容易的事情，因为你没有办法对字面量赋值进行 `Cmd + 单击跳转`)。

和其他 **Swift** 的新鲜特性一样，我们究竟如何使用字面量转换，它的最佳实践到底是什么，都还是在研究及讨论中的。因此在使用这样的新特性时，必须力求表意清晰，没有误解，代码才能经受得住历史考验。

下标

下标相信大家都很熟悉了，在绝大多数语言中使用下标来读写类似数组或者是字典这样的数据结构的做法，似乎已经是业界标准。在 **Swift** 中，`Array` 和 `Dictionary` 当然也实现了下标读写：

```
var arr = [1,2,3]
arr[2] // 3
arr[2] = 4 // arr = [1,2,4]

var dic = ["cat":"meow", "goat":"mie"]
dic["cat"] // {Some "meow"}
dic["cat"] = "miao" // dic = ["cat":"miao", "goat":"mie"]
```

数组的话没有什么好多说的，但是字典需要注意，我们通过下标访问得到的结果是一个 `Optional` 的值。这是很容易理解的，因为你不能限制下标访问时的输入值，对于数组来说如果越界了只好直接给你脸色让你崩掉，但是对于字典，查询不到是很正常的一件事情。对此，在 **Swift** 中我们有更好的处理方式，那就是返回 `nil` 告诉你没有要找的东西。

作为一门代表了先进生产力的语言，**Swift** 是允许我们自定义下标的。这不仅包含了对自己写的类型进行下标自定义，也包括了对那些已经支持下标访问的类型（没错就是 `Array` 和 `Dictionary`）进行扩展。我们重点来看看向已有类型添加下标访问的情况吧，比如说 `Array`。很容易就可以在 **Swift** 的定义文件（在 **Xcode** 中通过 **Cmd +** 单击任意一个 **Swift** 内建的类型或者函数就可以访问到）里，找到 `Array` 已经支持的下标访问类型：

```
subscript (index: Int) -> T
subscript (subRange: Range<Int>) -> Slice<T>
```

共有两种，它们分别接受单个 `Int` 类型的序号和一个表明范围的 `Range<Int>`，作为对应，返回值也分别是单个元素和一组对应输入返回的元素。

于是我们发现一个挺郁闷的问题，那就是我们很难一次性取出某几个特定位置的元素，比如在一个数组内，我想取出 `index` 为 0, 2, 3 的元素的时候，现有的体系就会比较吃力。我们很可能会要去枚举数组，然后在循环里判断是否是我们想要的位置。其实这里有更好的做法，比如说可以实现一个接受数组作为下标输入的读取方法：

```
extension Array {
    subscript(input: [Int]) -> ArraySlice<Element> {
        get {
            var result = ArraySlice<Element>()
            for i in input {
                assert(i < self.count, "Index out of range")
                result.append(self[i])
            }
            return result
        }
    }
}
```

```

    set {
        for (index,i) in input.enumerate() {
            assert(i < self.count, "Index out of range")
            self[i] = newValue[index]
        }
    }
}

```

这样，我们的 `Array` 的灵活性就大大增强了：

```

var arr = [1,2,3,4,5]
arr[[0,2,3]] // [1,3,4]
arr[[0,2,3]] = [-1,-3,-4]
arr // [-1,2,-3,-4,5]

```

练习

虽然我们在这里实现了下标为数组的版本，但是我并不推荐使用这样的形式。如果阅读过[参数列表](#)一节的读者也许会想为什么在这里我们不使用看起来更优雅的参数列表的方式，也就是 `subscript(input: Int...)` 的形式。不论从易用性还是可读性上来说，参数列表的形式会更好。但是存在一个问题，那就是在只有一个输入参数的时候参数列表会导致和现有的定义冲突，有兴趣的读者不妨试试看。当然，我们完全可以使用至少两个参数的参数列表形式来避免这个冲突，即定义形如 `subscript(first: Int, second: Int, others: Int...)` 的下标方法，我想这作为练习留给读者进行尝试会更好。

方法嵌套

方法终于成为了一等公民，也就是说，我们可以将方法当作变量或者参数来使用了。更进一步地，我们甚至可以在一个方法中定义新的方法，这给代码结构层次和访问级别的控制带来的新的选择。

想想看有多少次我们因为一个方法主体内容过长，而不得不将它重构为好几个小的功能块的方法，然后在原来的主体方法中去调用这些小方法。这些具体负责一个个小功能块的方法也许一辈子就被调用这么一次，但是却不得不存在于整个类型的作用域中。虽然我们会将它们标记为私有方法，但是事实上它们所承担的任务往往和这个类型没有直接关系，而只是会在这个类型中的某个方法中被用到。更甚至这些小方法也可能有些复杂，我们还想进一步将它们分成更小的模块，我们很可能也只有将它们放到和其他方法平级的地方。这样一来，本来应该是进深的结构，却被整个展平了，导致之后在对代码的理解和维护上都很成问题。在 Swift 中，我们对于这种情况有了很好的应对，我们可以在方法中定义其他方法，也就是说让方法嵌套起来。

举个例子，我们在写一个网络请求的类 `Request` 时，可能面临着将请求的参数编码到 `url` 里的任务。因为输入的参数可能包括单个的值，字典，或者是数组，因此为了结构漂亮和保持方法短小，我们可能将情况分开，写出这样的代码：

```
func appendQuery(var url: String,
                 key: String,
                 value: AnyObject) -> String {

    if let dictionary = value as? [String: AnyObject] {
        return appendQueryDictionary(url, key, dictionary)
    } else if let array = value as? [AnyObject] {
        return appendQueryArray(url, key, array)
    } else {
        return appendQuerySingle(url, key, value)
    }
}

func appendQueryDictionary(var url: String,
                           key: String,
                           value: [String: AnyObject]) -> String {

    //...
    return result
}

func appendQueryArray(var url: String,
                      key: String,
                      value: [AnyObject]) -> String {

    //...
    return result
}

func appendQuerySingle(var url: String,
                       key: String,
                       value: AnyObject) -> String {

    //...
    return result
}
```

事实上后三个方法都只会在第一个方法中被调用，它们其实和 `Request` 没有直接的关系，所以将它们放到 `appendQuery` 中去会是一个更好的组织形式：

```
func appendQuery(var url: String,
                 key: String,
                 value: AnyObject) -> String {

    func appendQueryDictionary(var url: String,
                               key: String,
                               value: [String: AnyObject]) -> String {

        //...
        return result
    }

    func appendQueryArray(var url: String,
                          key: String,
                          value: [AnyObject]) -> String {

        //...
        return result
    }

    func appendQuerySingle(var url: String,
                           key: String,
                           value: AnyObject) -> String {

        //...
        return result
    }

    if let dictionary = value as? [String: AnyObject] {
        return appendQueryDictionary(url, key, dictionary)
    } else if let array = value as? [AnyObject] {
        return appendQueryArray(url, key, array)
    } else {
        return appendQuerySingle(url, key, value)
    }
}
```

另一个重要的考虑是虽然 `Swift` 提供了 `public`，`internal` 和 `private` 三种访问权限，但是有些方法我们完全不希望其他地方被直接使用。最常见的例子就是在方法的模板中：我们一方面希望灵活地提供一个模板来让使用者可以通过模板定制他们想要的方法，但另一方面又不希望暴露太多实现细节，或者甚至是让使用者可以直接调用到模板。一个最简单的例子就是在[参数修饰](#)一节中提到过的类似这样的代码：

```
func makeIncrementor(addNumber: Int) -> ((inout Int) -> Void) {
    func incrementor(inout variable: Int) -> Void {
        variable += addNumber;
    }
    return incrementor;
}
```


命名空间

Objective-C 一个一直以来令人诟病的地方就是没有命名空间，在应用开发时，所有的代码和引用的静态库最终都会被编译到同一个域和二进制中。这样的后果是一旦我们有重复的类名的话，就会导致编译时的冲突和失败。为了避免这种事情的发生，Objective-C 的类型一般都会加上两到三个字母的前缀，比如 Apple 保留的 `NS` 和 `UI` 前缀，各个系统框架的前缀 `SK` (StoreKit)，`CG` (CoreGraphic) 等。Objective-C 社区的大部分开发者也遵守了这个约定，一般都会将自己名字缩写作为前缀，把类库命名为 `AFNetworking` 或者 `MBProgressHUD` 这样。这种做法可以解决部分问题，至少我们在直接引用不同人的库时冲突的概率大大降低了，但是前缀并不意味着不会冲突，有时候我们确实还是会遇到即使使用前缀也仍然相同的情况。另外一种情况是可能你想使用的两个不同的库，分别在它们里面引用了另一个相同的很流行的第三方库，而又没有更改名字。在你分别使用这两个库中的一个时是没有问题的，但是一旦你将这两个库同时加到你的项目中的话，这个大家共用的第三方库就会和自己发生冲突了。

在 Swift 中，由于可以使用命名空间了，即使是名字相同的类型，只要是来自不同的命名空间的话，都是可以和平共处的。和 C# 这样的显式在文件中指定命名空间的做法不同，Swift 的命名空间是基于 `module` 而不是在代码中显式地指明，每个 `module` 代表了 Swift 中的一个命名空间。也就是说，同一个 `target` 里的类型名称还是不能相同的。在我们进行 app 开发时，默认添加到 app 的主 `target` 的内容都是处于同一个命名空间中的，我们可以通过创建 Cocoa (Touch) Framework 的 `target` 的方法来新建一个 `module`，这样我们就可以在两个不同的 `target` 中添加同样名字的类型了：

```
// MyFramework.swift
// 这个文件存在于 MyFramework.framework 中
public class MyClass {
    public class func hello() {
        print("hello from framework")
    }
}

// MyApp.swift
// 这个文件存在于 app 的主 target 中
class MyClass {
    class func hello() {
        print("hello from app")
    }
}
```

在使用时，如果出现可能冲突的时候，我们需要在类型名称前面加上 `module` 的名字 (也就是 `target` 的名字)：

```
MyClass.hello()
// hello from app

MyFramework.MyClass.hello()
// hello from framework
```

因为是在 `app` 的 `target` 中调用的，所以第一个 `MyClass` 会直接使用 `app` 中的版本，第二个调用我们指定了 `MyFramework` 中的版本。

另一种策略是使用类型嵌套的方法来指定访问的范围。常见做法是将名字重复的类型定义到不同的 `struct` 中，以此避免冲突。这样在不使用多个 `module` 的情况下也能取得隔离同样名字的类型的效果：

```
struct MyClassContainer1 {
    class MyClass {
        class func hello() {
            print("hello from MyClassContainer1")
        }
    }
}

struct MyClassContainer2 {
    class MyClass {
        class func hello() {
            print("hello from MyClassContainer2")
        }
    }
}
```

使用时：

```
MyClassContainer1.MyClass.hello()
MyClassContainer2.MyClass.hello()
```

其实不管哪种方式都和传统意义上的命名空间有所不同，把它叫做命名空间，更多的是一种概念上的宣传。不过在实际使用中只要遵守这套规则的话，还是能避免很多不必要的麻烦的，至少唾手可得的是我们不再需要给类名加上各种奇怪的前缀了。

Any 和 AnyObject

`Any` 和 `AnyObject` 是 `Swift` 中两个妥协的产物，也是很让人迷惑的概念。在 `Swift` 官方编程指南中指出

`AnyObject` 可以代表任何 `class` 类型的实例

`Any` 可以表示任意类型，甚至包括方法 (`func`) 类型

先来说说 `AnyObject` 吧。写过 `Objective-C` 的读者可能会知道在 `Objective-C` 中有一个叫做 `id` 的神奇的东西。编译器不会对向声明为 `id` 的变量进行类型检查，它可以表示任意类的实例这样的概念。在 `Cocoa` 框架中很多地方都使用了 `id` 来进行像参数传递和方法返回这样的工作，这是 `Objective-C` 动态特性的一种表现。现在的 `Swift` 最主要的用途依然是使用 `Cocoa` 框架进行 `app` 开发，因此为了与 `Cocoa` 架构协作，将原来 `id` 的概念使用了一个类似的，可以代表任意 `class` 类型的 `AnyObject` 来进行替代。

但是两者其实是有本质区别的。在 `Swift` 中编译器不仅不会对 `AnyObject` 实例的方法调用做出检查，甚至对于 `AnyObject` 的所有方法调用都会返回 `Optional` 的结果。这虽然是符合 `Objective-C` 中的理念的，但是在 `Swift` 环境下使用起来就非常麻烦，也很危险。应该选择的做法是在使用时先确定 `AnyObject` 真正的类型并进行转换以后再进行调用。

假设原来的某个 `API` 返回的是一个 `id`，那么在 `Swift` 中现在将被映射为 `AnyObject?` (因为 `id` 是可以指向 `nil` 的，所以在这里我们需要一个 `Optional` 的版本)，虽然我们不知道调用来说应该是没问题的，但是我们依然最好这样写：

```
func someMethod() -> AnyObject? {
    // ...

    // 返回一个 AnyObject?, 等价于在 Objective-C 中返回一个 id
    return result
}

let anyObject: AnyObject? = SomeClass.someMethod()
if let someInstance = anyObject as? SomeRealClass {
    // ...
    // 这里我们拿到了具体 SomeRealClass 的实例

    someInstance.funcOfSomeRealClass()
}
```

如果我们注意到 `AnyObject` 的定义，可以发现它其实就是一个接口：

```
protocol AnyObject {
}
```

特别之处在于，所有的 `class` 都隐式地实现了这个接口，这也是 `AnyObject` 只适用于 `class` 类型的原因。而在 `Swift` 中所有的基本类型，包括 `Array` 和 `Dictionary` 这些传统意义上会是 `class` 的东西，统统都是 `struct` 类型，并不能由 `AnyObject` 来表示，于是 `Apple` 提出了一个更为特殊的 `Any`，除了 `class` 以外，它还可以表示包括 `struct` 和 `enum` 在内的所有类型。

为了深入理解，举个很有意思的例子。为了实验 `Any` 和 `AnyObject` 的特性，在 `Playground` 里写下如下代码：

```
import UIKit

let swiftInt: Int = 1
let swiftString: String = "miao"

var array: [AnyObject] = []
array.append(swiftInt)
array.append(swiftString)
```

我们在这里声明了一个 `Int` 和一个 `String`，按理说它们都应该只能被 `Any` 代表，而不能被 `AnyObject` 代表的。但是你会发现这段代码是可以编译运行通过的。那是不是说其实 `Apple` 的编程指南出错了呢？不是这样的，你可以打印一下 `array`，就会发现里面的元素其实已经变成了 `NSNumber` 和 `NSString` 了，这里发生了一个自动的转换。因为我们 `import` 了 `UIKit` (其实这里我们需要的只是 `Foundation`，而在导入 `UIKit` 的时候也会同时将 `Foundation` 导入)，在 `Swift` 和 `Cocoa` 中的这几个对应的类型是可以进行自动转换的。因为我们显式地声明了需要 `AnyObject`，编译器认为我们需要的是 `Cocoa` 类型而非原生类型，而帮我们进行了自动的转换。

在上面的代码中如果我们将 `import UIKit` 去掉的话，就会得到无法适配 `AnyObject` 的编译错误了。我们需要做的是将声明 `array` 时的 `[AnyObject]` 换成 `[Any]`，就一切正确了。

```
let swiftInt: Int = 1
let swiftString: String = "miao"

var array: [Any] = []
array.append(swiftInt)
array.append(swiftString)
array
```

顺便值得一提的是，只使用 `Swift` 类型而不转为 `Cocoa` 类型，对性能的提升是有所帮助的，所以我们应该尽可能地使用原生的类型。

其实说真的，使用 `Any` 和 `AnyObject` 并不是什么令人愉悦的事情，正如开头所说，这都是为妥协而存在的。如果在我们自己的代码里需要大量经常地使用这两者的话，往往意味着代码可能在结构和设计上存在问题，应该及时重新审视。简单来说，我们最好避免依赖和使用这两者，而去尝试明确地指出确定的类型。

typealias 和泛型接口

`typealias` 是用来为已经存在的类型重新定义名字的，通过命名，可以使代码变得更加清晰。使用的语法也很简单，使用 `typealias` 关键字像使用普通的赋值语句一样，可以将某个已经存在的类型赋值为新的名字。比如在计算二维平面上的距离和位置的时候，我们一般使用 `Double` 来表示距离，用 `CGPoint` 来表示位置：

```
func distanceBetweenPoint(point: CGPoint, toPoint: CGPoint) -> Double {
    let dx = Double(toPoint.x - point.x)
    let dy = Double(toPoint.y - point.y)
    return sqrt(dx * dx + dy * dy)
}

let origin: CGPoint = CGPoint(x: 0, y: 0)
let point: CGPoint = CGPoint(x: 1, y: 1)

let distance: Double = distanceBetweenPoint(origin, point)
```

虽然在数学上和最后的程序运行上都没什么问题，但是阅读和维护的时候总是觉得有哪里不对。因为我们没有将数学抽象和实际问题结合起来，使得在阅读代码时我们还需要在大脑中进行一次额外的转换：`CGPoint` 代表一个点，而这个点就是我们在定义的坐标系里的位置；`Double` 是一个数字，它代表两个点之间的距离。

如果我们使用 `typealias`，就可以将这种转换直接写在代码里，从而减轻阅读和维护的负担：

```
import UIKit

typealias Location = CGPoint
typealias Distance = Double

func distanceBetweenPoint(location: Location,
    toLocation: Location) -> Distance {
    let dx = Distance(location.x - toLocation.x)
    let dy = Distance(location.y - toLocation.y)
    return sqrt(dx * dx + dy * dy)
}

let origin: Location = Location(x: 0, y: 0)
let point: Location = Location(x: 1, y: 1)

let distance: Distance = distanceBetweenPoint(origin, toLocation: point)
```

同样的代码，在 `typealias` 的帮助下，读起来就轻松多了。可能单单这个简单例子不会有特别的体会，但是当你遇到更复杂的实际问题时，你就可以不再关心并去思考自己代码里那些成堆的 `Int` 或者 `String` 之类的基本类型到底代表的是什么东西了，这样你应该能省下不少脑细胞。

对于普通类型并没有什么难点，但是在涉及到泛型时，情况就稍微不太一样。首先，`typealias` 是单一的，也就是说你必须指定将某个特定的类型通过 `typealias` 赋值为新名字，而不能将整个泛型

类型进行重命名。下面这样的命名都是无法通过编译的：

这是错误代码

```
class Person<T> {}
typealias Worker = Person
typealias Worker = Person<T>
typealias Worker<T> = Person<T>
```

一旦泛型类型的确定性得到保证后，我们就可以重命名了：

```
class Person<T> {}

typealias WorkId = String
typealias Worker = Person<WorkId>
```

另一个值得一提的是 **Swift** 中是没有泛型接口的，但是使用 **typealias**，我们可以在接口里定义一个必须实现的别名，这在一定范围内也算一种折衷方案。比如在 `GeneratorType` 和 `SequenceType` 这两个接口中，**Swift** 都用到了这个技巧，来为接口确定一个使用的类似泛型的特性：

```
protocol GeneratorType {
    typealias Element
    mutating func next() -> Self.Element?
}

protocol SequenceType {
    typealias Generator : GeneratorType
    func generate() -> Self.Generator
}
```

在实现这些接口时，我们不仅需要实现指定的方法，还要实现对应的 **typealias**，这其实是一种对于接口适用范围的抽象和约束。

可变参数函数

可变参数函数指的是可以接受任意多个参数的函数，我们最熟悉的可能就是 `NSString` 的 `-stringWithFormat:` 方法了。在 `Objective-C` 中，我们使用这个方法生成字符串的写法是这样的：

```
NSString *name = @"Tom";
NSDate *date = [NSDate date];
NSString *string = [NSString stringWithFormat:
    @"Hello %@. Date: %@", name, date];
```

这个方法中的参数是可以任意变化的，参数的第一项是需要格式化的字符串，后面的参数都是向第一个参数中填空。在这里我们不再详细描述 `Objective-C` 中可变参数函数的写法 (毕竟这是一本 `Swift` 的书)，但是我相信绝大多数即使有着几年 `Objective-C` 经验的读者，也很难在不查阅资料的前提下正确写出一个接受可变参数的函数。

但是这一切在 `Swift` 中得到了前所未有的简化。现在，写一个可变参数的函数只需要在声明参数时在类型后面加上 `...` 就可以了。比如下面就声明了一个接受可变参数的 `Int` 累加函数：

```
func sum(input: Int...) -> Int {
    //...
}
```

输入的 `input` 在函数体内部将被作为数组 `[Int]` 来使用，让我们来完成上面的方法吧。当然你可以用传统的 `for...in` 做累加，但是这里我们选择了一种看起来更 `Swift` 的方式：

```
func sum(input: Int...) -> Int {
    return input.reduce(0, combine: +)
}

print(sum(1, 2, 3, 4, 5))
// 输出: 15
```

在使用可变参数时需要注意的是可变参数只能作为方法中的最后一个参数来使用，而不能先声明一个可变参数，然后再声明其他参数。这是很容易理解的，因为编译器将不知道输入的参数应该从哪里截断。另外，在一个方法中，最多只能有一组可变参数。

一个比较恼人的限制是可变参数都必须是同一种类型的，当我们想要同时传入多个类型的参数时就需要做一些变通。比如最开始提到的 `-stringWithFormat:` 方法。可变参数列表的第一个元素是等待格式化的字符串，在 `Swift` 中这会对应一个 `String` 类型，而剩下的参数应该可以是对应格式化标准的任意类型。一种解决方法是使用 `Any` 作为参数类型，然后对接收到的数组的首个元素进行特殊处理。不过因为 `Swift` 提供了使用下划线 `_` 来作为参数的外部标签，来使调用时不再需要加上参数名字。我们可以利用这个特性，在声明方法是就指定第一个参数为一个字符串，然后跟

一个匿名的参数列表，这样在写起来的时候就 "好像" 是所有参数都是在同一个参数列表中进行的处理，会好看很多。比如 Swift 的 `NSString` 格式化的声明就是这样处理的：

```
extension NSString {
    convenience init(format: NSString, _ args: CVarArgType...)
    //...
}
```

调用的时候就和在 Objective-C 时几乎一样了，非常方便：

```
let name = "Tom"
let date = NSDate()
let string = NSString(format: "Hello %@. Date: %@", name, date)
```


初始化方法顺序

与 Objective-C 不同，Swift 的初始化方法需要保证类型的所有属性都被初始化。所以初始化方法的调用顺序就很有讲究。在某个类的子类中，初始化方法里语句的顺序并不是随意的，我们需要保证在当前子类实例的成员初始化完成后才能调用父类的初始化方法：

```
class Cat {
    var name: String
    init() {
        name = "cat"
    }
}

class Tiger: Cat {
    let power: Int
    override init() {
        power = 10
        super.init()
        name = "tiger"
    }
}
```

一般来说，子类的初始化顺序是：

1. 设置子类自己需要初始化的参数， `power = 10`
2. 调用父类的相应的初始化方法， `super.init()`
3. 对父类中的需要改变的成员进行设定， `name = "tiger"`

其中第三步是根据具体情况决定的，如果我们在子类中不需要对父类的成员做出改变的话，就不存在第 3 步。而在这种情况下，Swift 会自动地对父类的对应 `init` 方法进行调用，也就是说，第 2 步的 `super.init()` 也是可以不用写的 (但是实际上还是调用的，只不过是为了简便 Swift 帮我们完成了)。这种情况下的初始化方法看起来就很简单：

```
class Cat {
    var name: String
    init() {
        name = "cat"
    }
}

class Tiger: Cat {
    let power: Int
    override init() {
        power = 10
        // 如果我们不需要打改变 name 的话，
        // 虽然我们没有显式地对 super.init() 进行调用
        // 不过由于这是初始化的最后了，Swift 替我们自动完成了
    }
}
```

Designated, Convenience 和 Required

我们在深入初始化方法之前，不妨先再想想 Swift 中的初始化想要达到一种怎样的目的。

其实就是安全。在 Objective-C 中，`init` 方法是非常不安全的：没有人能保证 `init` 只被调用一次，也没有人保证在初始化方法调用以后实例的各个变量都完成初始化，甚至如果在初始化里使用属性进行设置的话，还可能会造成[各种问题](#)，虽然 Apple 也[明确说明了](#)不应该在 `init` 中使用属性来访问，但是这并不是编译器强制的，因此还是会有很多开发者犯这样的错误。

所以 Swift 有了超级严格的初始化方法。一方面，Swift 强化了 **designated** 初始化方法的地位。Swift 中不加修饰的 `init` 方法都需要在方法中保证所有非 `Optional` 的实例变量被赋值初始化，而在子类中也强制 (显式或者隐式地) 调用 `super` 版本的 **designated** 初始化，所以无论如何走何种路径，被初始化的对象总是可以完成完整的初始化的。

```
class ClassA {
    let numA: Int
    init(num: Int) {
        numA = num
    }
}

class ClassB: ClassA {
    let numB: Int

    override init(num: Int) {
        numB = num + 1
        super.init(num: num)
    }
}
```

在上面的示例代码中，注意在 `init` 里我们可以对 `let` 的实例常量进行赋值，这是初始化方法的重要特点。在 Swift 中 `let` 声明的值是不变量，无法被写入赋值，这对于构建线程安全的 API 十分有用。而因为 Swift 的 `init` 只可能被调用一次，因此在 `init` 中我们可以为不变量进行赋值，而不会引起任何线程安全的问题。

与 **designated** 初始化方法对应的是在 `init` 前加上 `convenience` 关键字的初始化方法。这类方法是 Swift 初始化方法中的“二等公民”，只作为补充和提供使用上的方便。所有的 `convenience` 初始化方法都必须调用同一个类中的 **designated** 初始化完成设置，另外 `convenience` 的初始化方法是不能被子类重写或者是从子类中以 `super` 的方式被调用的。

```
class ClassA {
    let numA: Int
    init(num: Int) {
        numA = num
    }

    convenience init(bigNum: Bool) {
        self.init(num: bigNum ? 10000 : 1)
    }
}
```

```

}

class ClassB: ClassA {
  let numB: Int

  override init(num: Int) {
    numB = num + 1
    super.init(num: num)
  }
}

```

只要在子类中实现重写了父类 `convenience` 方法所需要的 `init` 方法的话，我们在子类中就可以使用父类的 `convenience` 初始化方法了。比如在上面的代码中，我们在 `ClassB` 里实现了 `init(num: Int)` 的重写。这样，即使在 `ClassB` 中没有 `bigNum` 版本的 `convenience init(bigNum: Bool)`，我们仍然还是可以用这个方法来完成子类初始化：

```

let anObj = ClassB(bigNum: true)
// anObj.numA = 10000, anObj.numB = 10001

```

因此进行一下总结，可以看到初始化方法永远遵循以下两个原则：

1. 初始化路径必须保证对象完全初始化，这可以通过调用本类型的 **designated** 初始化方法来得到保证；
2. 子类的 **designated** 初始化方法必须调用父类的 **designated** 方法，以保证父类也完成初始化。

对于某些我们希望子类中一定实现的 **designated** 初始化方法，我们可以通过添加 `required` 关键字进行限制，强制子类对这个方法重写实现。这样做的最大的好处是可以保证依赖于某个 **designated** 初始化方法的 `convenience` 一直可以被使用。一个现成的例子就是上面的 `init(bigNum: Bool)`：如果我们希望这个初始化方法对于子类一定可用，那么应当将 `init(num: Int)` 声明为必须，这样我们在子类中调用 `init(bigNum: Bool)` 时就始终能够找到一条完全初始化的路径了：

```

class ClassA {
  let numA: Int
  required init(num: Int) {
    numA = num
  }

  convenience init(bigNum: Bool) {
    self.init(num: bigNum ? 10000 : 1)
  }
}

class ClassB: ClassA {
  let numB: Int

  required init(num: Int) {
    numB = num + 1
    super.init(num: num)
  }
}

```

另外需要说明的是，其实不仅仅是对 **designated** 初始化方法，对于 **convenience** 的初始化方法，我们也可以加上 `required` 以确保子类对其进行实现。这在要求子类不直接使用父类中的 **convenience** 初始化方法时会非常有帮助。

初始化返回 nil

在 Objective-C 中，`init` 方法除了返回 `self` 以外，其实和一个普通的实例方法并没有太大区别。如果你喜欢的话，甚至可以多次进行调用，这都有限制。一般来说，我们还会在初始化失败 (比如输入不满足要求无法正确初始化) 的时候返回 `nil` 来通知调用者这次初始化没有正确完成。

但是，在 Swift 中默认情况下初始化方法是不能写 `return` 语句来返回值的，也就是说我们没有机会初始化一个 `Optional` 的值。一个很典型的例子就是初始化一个 `url`。在 Objective-C 中，如果我们使用一个错误的字符串来初始化一个 `NSURL` 对象时，返回会是 `nil` 代表初始化失败。所以下面这种 "防止度娘吞链接" 式的字符串 (注意两个 `t` 之间的空格和中文的句号) 的话，也是可以正常编译和运行的，只是结果是个 `nil`：

```
NSURL *url = [[NSURL alloc] initWithString:@"ht tp://swifter. tips"];
NSLog(@"%@", url);
// 输出 (null)
```

但是在 Swift 中情况就不那么乐观了，`-initWithString:` 在 Swift 中对应的是一个 `convenience init` 方法：`init(string URLString: String!)`。上面的 Objective-C 代码在 Swift 中等效为：

```
let url = NSURL(string: "ht tp://swifter. tips")
print(url)
```

`init` 方法在 Swift 1.1 中发生了很大的变化，为了将来龙去脉讲述清楚，我们先来看看在 Swift 1.0 下的表现。

Swift 1.0 及之前

如果在 Swift 1.0 的环境下尝试运行上面代码的话，我们会得到一个 `EXC_BAD_INSTRUCTION`，这说明触发了 Swift 内部的断言，这个初始化方法不接受这样的输入。一个常见的解决方法是使用工厂模式，也就是写一个类方法来生成和返回实例，或者在失败的时候返回 `nil`。Swift 的 `NSURL` 就做了这样的处理：

```
class func URLWithString(URLString: String!) -> Self!
```

使用的时候：

```
let url = NSURL.URLWithString("ht tp://swifter. tips")
print(url)
```

```
// 输出 nil
```

不过虽然可以用这种方式来和原来一样返回 `nil`，但是这也算是一种折衷。在可能的情况下，我们还是应该倾向于尽量减少出现 `Optional` 的可能性，这样更有助于代码的简化。

如果你确实想使用初始化方法而不愿意用工厂函数的话，也可以考虑用一个 `Optional` 量来存储结果

这样你就可以处理初始化失败了，不过相应的代价是代码复杂度的增加

```
let url: NSURL? = NSURL(string: "http://swifter.tips")
// nil
```

Swift 1.1 及之后

虽然在默认情况下不能在 `init` 中返回 `nil`，但是通过上面的例子我们可以看到 Apple 自家的 API 还是有这个能力的。

好消息是在 Swift 1.1 中 Apple 已经为我们加上了初始化方法中返回 `nil` 的能力。我们可以在 `init` 声明时在其后加上一个 `?` 或者 `!` 来表示初始化失败时可能返回 `nil`。比如为 `Int` 添加一个接收 `String` 作为参数的初始化方法。我们希望在方法中对中文和英文的数据进行解析，并输出 `Int` 结果。对其解析并初始化的时候，就可能遇到初始化失败的情况：

```
extension Int {
    init?(fromString: String) {
        self = 0
        var digit = fromString.characters.count - 1
        for c in fromString.characters {
            var number = 0
            if let n = Int(String(c)) {
                number = n
            } else {
                switch c {
                    case "一": number = 1
                    case "二": number = 2
                    case "三": number = 3
                    case "四": number = 4
                    case "五": number = 5
                    case "六": number = 6
                    case "七": number = 7
                    case "八": number = 8
                    case "九": number = 9
                    case "零": number = 0
                    default: return nil
                }
            }

            self = self + number * Int(pow(10, Double(digit)))
            digit = digit - 1
        }
    }
}
```

```
let number1 = Int(fromString: "12")
// {Some 12}

let number2 = Int(fromString: "三二五")
// {Some 325}

let number3 = Int(fromString: "七九八")
// {Some 798}

let number4 = Int(fromString: "吃了么")
// nil

let number5 = Int(fromString: "1a4n")
// nil
```

所有的结果都将是 `Int?` 类型，通过 **Optional Binding**，我们就能知道初始化是否成功，并安全地使用它们了。我们在这类初始化方法中还可以对 `self` 进行赋值，也算是 `init` 方法里的特权之一。

同时像上面例子中的 `NSURL.URLWithString` 这样的工厂方法，在 **Swift 1.1** 中已经不再需要。为了简化 API 和安全，Apple 已经被标记为不可用了并无法编译。而对应地，可能返回 `nil` 的 `init` 方法都加上了 `?` 标记：

```
convenience init?(string urlString: String)
```

在新版本的 **Swift** 中，对于可能初始化失败的情况，我们应该始终使用可返回 `nil` 的初始化方法，而不是类型工厂方法。

protocol 组合

在 Swift 中我们可以使用 `Any` 来表示任意类型 (如果你对此感到模糊或者陌生的话, 可以先看看 Apple 的 Swift 官方教程或者本书的[这篇 tip](#)), 充满好奇心的同学可能已经发现, `Any` 这个类型的定义十分奇怪, 它是一个 `protocol<>` 的同名类型。

`protocol<>` 这样形式的写法在日常 Swift 使用中其实并不多见, 这其实是 Swift 的接口组合的用法。标准的语法形式是下面这样的:

```
protocol<ProtocolA, ProtocolB, ProtocolC>
```

尖括号内是具体接口的名称, 这里表示将名称为 `ProtocolA`, `ProtocolB` 以及 `ProtocolC` 的接口组合在一起的一个新的匿名接口。实现这个匿名接口就意味着要同时实现这三个接口所定义的内容。所以说, 这里的 `protocol` 组合的写法和下面的新声明的 `ProtocolD` 是相同的:

```
protocol ProtocolD: ProtocolA, ProtocolB, ProtocolC {  
  
}
```

那么, 在 `Any` 定义的时候的里面什么都不写的 `protocol<>` 是什么意思呢? 从语意上来说, 这代表一个 "需要实现空接口的接口", 其实就是任意类型的意思了。

除了可以方便地表达空接口这一概念以外, `protocol` 的组合相比于新创建一个接口的最大区别就在于其匿名性。有时候我们可以借助这个特性写出更清晰的代码。因为 Swift 的类型组织是比较松散的, 你的类型可以由不同的 `extension` 来定义实现不同的接口, Swift 也并没有要求它们在一个文件中。这样, 当一个类型实现了很多接口时, 在使用这个类型的时候我们很可能在不查询相关代码的情况下很难知道这个类型所实现的接口。

举个理想化的例子, 比如我们有下面的三个接口, 分别代表了三种动物的叫的方式, 而有一种谜之动物, 同时实现了这三个接口:

```
protocol KittenLike {  
    func meow() -> String  
}  
  
protocol DogLike {  
    func bark() -> String  
}  
  
protocol TigerLike {  
    func aou() -> String  
}  
  
class MysteryAnimal: KittenLike, DogLike, TigerLike {  
    func meow() -> String {
```



```

        return "meow"
    }

    func bark() -> String {
        return "bark"
    }

    func aou() -> String {
        return "aou"
    }
}

```

现在我们要检查某种动物作为宠物的时候的叫声的话，我们可能要重新定义一个叫做 `PetLike` 的接口，表明其实现 `KittenLike` 和 `DogLike`；如果稍后我们又想检查某种动物作为猫科动物的叫声的话，我们也许又要去定义一个叫做 `CatLike` 这样的实现 `KittenLike` 和 `TigerLike` 的接口。最后我们大概会写出这样的东西：

```

protocol PetLike: KittenLike, DogLike {
}

protocol CatLike: KittenLike, TigerLike {
}

struct SoundChecker {
    static func checkPetTalking(pet: PetLike) {
        //...
    }

    static func checkCatTalking(cat: CatLike) {
        //...
    }
}

```

虽然没有引入定义任何新的内容，但是为了实现这个需求，我们还是添加了两个空 `protocol`，这可能会让人困惑，代码的使用者 (也包括一段时间后的你自己) 可能会去猜测 `PetLike` 和 `CatLike` 的作用 -- 其实它们除了标注以外并没有其他作用。借助 `protocol` 组合的特性，我们可以很好的解决这个问题。`protocol` 组合是可以使用 `typealias` 来命名的，于是可以将上面的新定义 `protocol` 的部分换为：

```

typealias PetLike = protocol<KittenLike, DogLike>
typealias CatLike = protocol<KittenLike, TigerLike>

```

这样既保持了可读性，也没有多定义不必要的新类型。

另外，其实如果这两个临时接口我们就只用一次的话，如果上下文里理解起来不会有困难，我们完全可以直接将它们匿名化，变成下面这样：

```

struct SoundChecker {
    static func checkPetTalking(pet: protocol<KittenLike, DogLike>) {

```

```

        //...
    }

    static func checkCatTalking(cat: protocol<KittenLike, TigerLike>) {
        //...
    }
}

```

这样的好处是定义和使用的地方更加接近，这在代码复杂的时候读代码时可以少一些跳转，多一些专注。但是因为使用了匿名的接口组合，所以能表达的信息毕竟少了一些。如果要实际使用这种方法的话，还是需要多多斟酌。

虽然这一节已经够长了，不过我还是想多提一句关于实现多个接口时接口内方法冲突的解决方法。因为在 Swift 的世界中没有人限制或者保证过不同接口的方法不能重名，所以这是有可能出现的情况。比如有 `A` 和 `B` 两个接口，定义如下：

```

protocol A {
    func bar() -> Int
}

protocol B {
    func bar() -> String
}

```

两个接口中 `bar()` 只有返回值的类型不同。我们如果有一个类型 `Class` 同时实现了 `A` 和 `B`，我们要怎样才能避免和解决调用冲突呢？

```

class Class: A, B {
    func bar() -> Int {
        return 1
    }

    func bar() -> String {
        return "Hi"
    }
}

```

这样一来，对于 `bar()`，只要在调用前进行类型转换就可以了：

```

let instance = Class()
let num = (instance as A).bar() // 1
let str = (instance as B).bar() // "Hi"

```

static 和 class

Swift 中表示“类型范围作用域”这一概念有两个不同的关键字，它们分别是 `static` 和 `class`。这两个关键字确实都表达了这个意思，但是在其他一些语言，包括 **Objective-C** 中，我们并不会特别地区分变量/类方法和静态变量/静态函数。但是在 Swift 的早期版本中，这两个关键字却是不能用混的。

在非 `class` 的类型上下文中，我们统一使用 `static` 来描述类型作用域。这包括在 `enum` 和 `struct` 中表述类型方法和类型属性时。在这两个值类型中，我们可以在类型范围内声明并使用存储属性，计算属性和方法。`static` 适用的场景有这些：

```
struct Point {
    let x: Double
    let y: Double

    // 存储属性
    static let zero = Point(x: 0, y: 0)

    // 计算属性
    static var ones: [Point] {
        return [Point(x: 1, y: 1),
                Point(x: -1, y: 1),
                Point(x: 1, y: -1),
                Point(x: -1, y: -1)]
    }

    // 类型方法
    static func add(p1: Point, p2: Point) -> Point {
        return Point(x: p1.x + p2.x, y: p1.y + p2.y)
    }
}
```

`enum` 的情况与这个十分类似，就不再列举了。

`class` 关键字相比起来就明白许多，是专门用在 `class` 类型的上下文中的，可以用来修饰类方法以及类的计算属性。但是有一个例外，`class` 中现在是不能出现 `class` 的存储属性的，我们如果写类似这样的代码的话：

```
class MyClass {
    class var bar: Bar?
}
```

编译时会得到一个错误：

```
class variables not yet supported
```

在 Swift 1.2 及之后，我们可以在 `class` 中使用 `static` 来声明一个类作用域的变量。也即：

```
class MyClass {
    static var bar: Bar?
}
```

的写法是合法的。有了这个特性之后，像单例的写法就可以回归到我们所习惯的方式了。

有一个比较特殊的是 `protocol`。在 Swift 中 `class`，`struct` 和 `enum` 都是可以实现某个 `protocol` 的。那么如果我们想在 `protocol` 里定义一个类型域上的方法或者计算属性的话，应该用哪个关键字呢？答案是使用 `static` 进行定义。在使用的时候，`struct` 或 `enum` 中仍然使用 `static`，而在 `class` 里我们既可以使用 `class` 关键字，也可以用 `static`，它们的结果是相同的：

```
protocol MyProtocol {
    static func foo() -> String
}

struct MyStruct: MyProtocol {
    static func foo() -> String {
        return "MyStruct"
    }
}

enum MyEnum: MyProtocol {
    static func foo() -> String {
        return "MyEnum"
    }
}

class MyClass: MyProtocol {
    // 在 class 中可以使用 class
    class func foo() -> String {
        return "MyClass.foo()"
    }

    // 也可以使用 static
    static func bar() -> String {
        return "MyClass.bar()"
    }
}
```

在 Swift 1.2 之前 `protocol` 中使用的是 `class` 作为关键字，但这确实是不合逻辑的。Swift 1.2 和 2.0 分两次对此进行了改进。现在只需要记住结论，在任何时候使用 `static` 应该都是没有问题的。

多类型和容器

Swift 中有两个原生的容器类型，`Array` 和 `Dictionary`：

```
struct Array<Element> :  
    CollectionType, Indexable, SequenceType,  
    MutableCollectionType, _DestructorSafeContainer {  
  
    //...  
  
}  
  
struct Dictionary<Key : Hashable, Value> :  
    CollectionType, Indexable, SequenceType,  
    DictionaryLiteralConvertible {  
  
    //...  
  
}
```

它们都是泛型的，也就是说我们可以在一个集合中只能放同一个类型的元素。比如：

```
let numbers = [1, 2, 3, 4, 5]  
// numbers 的类型是 [Int]  
  
let strings = ["hello", "world"]  
// strings 的类型是 [String]
```

如果我们要把不相关的类型放到同一个容器类型中的话，一个比较容易想到的是使用 `Any` 或者 `AnyObject`，或者是使用 `NSArray`：

```
import UIKit  
  
let mixed: [Any] = [1, "two", 3]  
  
// 如果不指明类型，由于 UIKit 的存在  
// 将被推断为 [NSObject]  
let objectArray = [1, "two", 3]
```

这样的转换会造成部分信息的损失，我们从容器中取值时只能得到信息完全丢失后的结果，在使用时还需要进行一次类型转换。这其实是在无其他可选方案后的最差选择：因为使用这样的转换的话，编译器就不能再给我们提供警告信息了。我们可以随意地将任意对象添加进容器，也可以将容器中取出的值转换为任意类型，这是一件十分危险的事情：

```
let any = mixed[0] // Any 类型  
let nsObject = objectArray[0] // NSObject 类型
```

其实我们注意到，`Any` 其实是 `protocol`，而不是具体的某个类型。因此就是说其实在容器类型泛型的帮助下，我们不仅可以在容器中添加同一具体类型的对象，也可以添加实现了同一接口的类型的对象。绝大多数情况下，我们想要放入一个容器中的元素或多或少会有某些共同点，这就使得用接口来规定容器类型会很有用。比如上面的例子如果我们希望的是打印出容器内的元素的 `description`，可能我们更倾向于将数组声明为 `[CustomStringConvertible]` 的：

```
import Foundation
let mixed: [CustomStringConvertible] = [1, "two", 3]

for obj in mixed {
    print(obj.description)
}
```

这种方法虽然也损失了一部分类型信息，但是相对于 `Any` 或者 `AnyObject` 还是改善很多，在对于对象中存在某种共同特性的情况下无疑是最方便的。另一种做法是使用 `enum` 可以带有值的特点，将类型信息封装到特定的 `enum` 中。下面的代码封装了 `Int` 或者 `String` 类型：

```
import Foundation
enum IntOrString {
    case IntValue(Int)
    case StringValue(String)
}

let mixed = [IntOrString.IntValue(1),
             IntOrString.StringValue("two"),
             IntOrString.IntValue(3)]

for value in mixed {
    switch value {
    case let .IntValue(i):
        println(i * 2)
    case let .StringValue(s):
        println(s.capitalizedString)
    }
}

// 输出：
// 2
// Two
// 6
```

通过这种方法，我们完整地在编译时保留了不同类型的信息。为了方便，我们甚至可以进一步为 `IntOrString` 使用[字面量转换](#)的方法编写简单的获取方式，但那是另外一个故事了。

default 参数

Swift 的方法是支持默认参数的，也就是说在声明方法时，可以给某个参数指定一个默认使用的值。在调用该方法时要是传入了这个参数，则使用传入的值，如果缺少这个输入参数，那么直接使用设定的默认值进行调用。可以说这是 Objective-C 社区盼了好多年的一个特性了，Objective-C 由于语法的特点几乎无法在不大幅改动的情况下很好地实现默认参数。

和其他很多语言的默认参数相比较，Swift 中的默认参数限制更少，并没有所谓 "默认参数之后不能再出现无默认值的参数" 这样的规则，举个例子，下面两种方法的声明在 Swift 里都是合法可用的：

```
func sayHello1(str1: String = "Hello", str2: String, str3: String) {
    print(str1 + str2 + str3)
}

func sayHello2(str1: String, str2: String, str3: String = "World") {
    print(str1 + str2 + str3)
}
```

其他不少语言只能使用后面一种写法，将默认参数作为方法的最后一个参数。

在调用的时候，我们如果想要使用默认值的话，只要不传入相应的值就可以了。下面这样的调用将得到同样的结果：

```
sayHello1(str2: " ", str3: "World")
sayHello2("Hello", str2: " ")

//输出都是 Hello World
```

这两个调用都省略了带有默认值的参数，sayHello1 中 str1 是默认的 "Hello"，而 sayHello2 中的 str3 是默认的 "World"。

另外如果喜欢 Cmd + 单击点来点去到处看的朋友可能会注意到 NSLocalizedString 这个常用方法的签名现在是：

```
func NSLocalizedString(key: String,
                      tableName: String? = default,
                      bundle: NSBundle = default,
                      value: String = default,
                      comment: String) -> String
```

默认参数写的是 default，这是含有默认参数的方法所生成的 Swift 的调用接口。当我们指定一个编译时就能确定的常量来作为默认参数的取值时，这个取值是隐藏在方法实现内部，而不应该暴

露给其他部分。与 `NSString` 很相似的还有 Swift 中的各类断言：

```
func assert(@autoclosure condition: () -> Bool,
            @autoclosure _ message: () -> String = default,
            file: StaticString = default,
            line: UInt = default)
```


正则表达式

作为一门先进的编程语言，Swift 可以说吸收了众多其他先进语言的优点，但是有一点却是让人略微失望的，就是 Swift 至今为止并没有在语言层面上支持[正则表达式](#)。

大概是因为其实 app 开发并不像 Perl 或者 Ruby 那样的语言需要处理很多文字匹配的问题，Cocoa 开发者确实不是特别依赖正则表达式。但是并不排除有希望使用正则表达式的场景，我们是否能像其他语言一样，使用比如 `==` 这样的符号来进行正则匹配呢？

最容易想到也是最容易实现的当然是自定义 `==` 这个运算符。在 Cocoa 中我们可以使用 `NSRegularExpression` 来做正则匹配，那么其实我们为它写一个包装也并不是什么太困难的事情。因为做的是字符串正则匹配，所以 `==` 左右两边都是字符串。我们可以先写一个接受正则表达式的字符串，以此生成 `NSRegularExpression` 对象。然后使用该对象来匹配输入字符串，并返回结果告诉调用者匹配是否成功。一个最简单的实现可能是下面这样的：

```
struct RegexHelper {
    let regex: NSRegularExpression

    init(_ pattern: String) throws {
        try regex = NSRegularExpression(pattern: pattern,
                                         options: .CaseInsensitive)
    }

    func match(input: String) -> Bool {
        let matches = regex.matchesInString(input,
                                             options: [],
                                             range: NSRange(0, input.characters.count))
        return matches.count > 0
    }
}
```

在使用的时候，比如我们想要匹配一个邮箱地址，我们可以这样来使用：

```
let mailPattern =
    "^([a-z0-9_\\.-]+)@[\\da-z\\.-]+\\.([a-z\\.-]{2,6})$"

let matcher: RegexHelper
do {
    matcher = try RegexHelper(mailPattern)
}

let maybeMailAddress = "onev@onevcat.com"

if matcher.match(maybeMailAddress) {
    print("有效的邮箱地址")
}
// 输出：
// 有效的邮箱地址
```

如果你想问 `mailPattern` 这一大串莫名其妙的匹配表达式是什么意思的话..>嘛..实在抱歉这

里不是正则表达式的课堂，所以关于这个问题我推荐看看这篇很棒的[正则表达式 30 分钟入门教程](#)，如果你连 30 分钟都没有的话，打开 [8 个常用正则表达式](#) 先开始抄吧..

上面那个式子就是我从这里抄来的

现在我们有了方便的封装，接下来就让我们实现 `==` 吧。这里只给出结果了，关于如何实现操作符和重载操作符的内容，可以参考[操作符](#)一节的内容。

```
infix operator == {
    associativity none
    precedence 130
}

func ==(lhs: String, rhs: String) -> Bool {
    do {
        return try RegexHelper(rhs).match(lhs)
    } catch _ {
        return false
    }
}
```

这下我们就可以使用类似于其他语言的正则匹配的方法了：

```
if "onev@onevcats.com" ==
    "^[a-z0-9_\\.-]+@[\\da-z\\.-]+\\.([a-z\\.]{2,6})$" {
    print("有效的邮箱地址")
}
// 输出：
// 有效的邮箱地址
```

Swift 1.0 版本主要会专注于成为一个非常适合制作 app 的语言，而现在看来 Apple 和 Chris 也有野心将 Swift 带到更广阔的平台去。那时候可能会在语言层面加上正则表达式的支持，到时候这篇 tip 可能也就没有意义了，不过我个人还是非常期盼那一天早些到来。

模式匹配

在之前的[正则表达式](#)中，我们实现了 `==` 操作符来完成简单的正则匹配。虽然在 Swift 中没有内置的正则表达式支持，但是一个和正则匹配有些相似的特性其实是内置于 Swift 中的，那就是[模式匹配](#)。

当然，从概念上来说正则匹配只是模式匹配的一个子集，但是在 Swift 里现在的模式匹配还很初级，也很简单，只能支持最简单的相等匹配和范围匹配。在 Swift 中，使用 `~=` 来表示模式匹配的操作符。如果我们看看 API 的话，可以看到这个操作符有下面几种版本：

```
func ~=<T : Equatable>(a: T, b: T) -> Bool

func ~=<T>(lhs: _OptionalNilComparisonType, rhs: T?) -> Bool

func ~=<I : IntervalType>(pattern: I, value: I.Bound) -> Bool
```

从上至下在操作符左右两边分别接收可以判等的类型，可以与 `nil` 比较的类型，以及一个范围输入和某个特定值，返回值很明了，都是是否匹配成功的 `Bool` 值。你是否有想起些什么呢..没错，就是 Swift 中非常强大的 `switch`，我们来看看 `switch` 的几种常见用法吧：

1. 可以判等的类型的判断

```
let password = "akfuv(3"
switch password {
    case "akfuv(3": print("密码通过")
    default:        print("验证失败")
}
```

2. 对 Optional 的判断

```
let num: Int? = nil
switch num {
    case nil: print("没值")
    default: print("\(num!)")
}
```

3. 对范围的判断

```
let x = 0.5
switch x {
    case -1.0...1.0: print("区间内")
    default:        print("区间外")
}
```

这并不是巧合。没错，Swift 的 `switch` 就是使用了 `~=` 操作符进行模式匹配，`case` 指定的模式作为左参数输入，而等待匹配的被 `switch` 的元素作为操作符的右侧参数。只不过这个调用是由 Swift 隐式地完成的。于是我们可以发挥想象的地方就很多了，比如在 `switch` 中做 `case` 判断的时候，我们完全可以使用我们自定义的模式匹配方法来进行判断，有时候这会让代码变得非常简洁，具有条例。我们只需要按照需求重载 `~=` 操作符就行了，接下来我们通过一个使用正则表达式做匹配的例子加以说明。

首先我们要做的是重载 `~=` 操作符，让它接受一个 `NSRegularExpression` 作为模式，去匹配输入的 `String`：

```
func ~= (pattern: NSRegularExpression, input: String) -> Bool {
    return pattern.numberOfMatchesInString(input,
        options: [],
        range: NSRange(location: 0, length: input.characters.count)) > 0
}
```

然后为了简便起见，我们再添加一个将字符串转换为 `NSRegularExpression` 的操作符 (当然也可以使用 `StringLiteralConvertible`，但是它不是这个 tip 的主题，在此就先不使用它了)：

```
prefix operator ~/ {}

prefix func ~/ (pattern: String) -> NSRegularExpression {
    return NSRegularExpression(pattern: pattern, options: nil, error: nil)
}
```

现在，我们在 `case` 语句里使用正则表达式的话，就可以去匹配被 `switch` 的字符串了：

```
let contact = ("http://onevcat.com", "onev@onevcat.com")

let mailRegex: NSRegularExpression
let siteRegex: NSRegularExpression

mailRegex =
    try ~/ "^( [a-z0-9_\\.-]+ )@( [\\da-z\\.-]+ )\\. ( [a-z\\.-]{2,6} ) $"
siteRegex =
    try ~/ "^( https?:\\/\\/ )? ( [\\da-z\\.-]+ )\\. ( [a-z\\.-]{2,6} ) ( [\\w \\.-]* ) *\\/ ? $"

switch contact {
    case (siteRegex, mailRegex): print("同时拥有有效的网站和邮箱")
    case (_, mailRegex): print("只拥有有效的邮箱")
    case (siteRegex, _): print("只拥有有效的网站")
    default: print("嘛都没有")
}

// 输出
// 同时拥有网站和邮箱
```

... 和 ..<

在很多脚本语言中 (比如 Perl 和 Ruby), 都有类似 `0..3` 或者 `0...3` 这样的 Range 操作符, 用来简单地指定一个从 X 开始连续计数到 Y 的范围。这个特性不论在哪个社区, 都是令人爱不释手的写法, Swift 中将其光明正大地 "借用" 过来, 也就不足为奇了。

最基础的用法当然还是在两边指定数字, `0...3` 就表示从 0 开始到 3 为止并包含 3 这个数字的范围, 我们将其称为全闭合的范围操作; 而在某些时候 (比如操作数组的 index 时), 我们更常用的是不包括最后一个数字的范围。这在 Swift 中被用一个看起来有些奇怪, 但是表达的意义很清晰的操作符来定义, 写作 `0..<3` -- 都写了小于号了, 自然是不包含最后的 3 的意思咯。

对于这样得到的数字的范围, 我们可以对它进行 `for...in` 的访问:

```
for i in 0...3 {
    print(i, appendNewline: false)
}

//输出 0123
```

如果你认为 `...` 和 `..<<` 只有这点内容的话, 就大错特错了。我们可以仔细看看 Swift 中对着两个操作符的定义 (为了清晰, 我稍微更改了一下它们的次序):

```
/// Forms a closed range that contains both `minimum` and `maximum`.
func ...<Pos : ForwardIndexType>(minimum: Pos, maximum: Pos)
    -> Range<Pos>

/// Forms a closed range that contains both `start` and `end`.
/// Requires: `start <= end`
func ...<Pos : ForwardIndexType where Pos : Comparable>(start: Pos, end: Pos)
    -> Range<Pos>

/// Forms a half-open range that contains `minimum`, but not
/// `maximum`.
func ..<<Pos : ForwardIndexType>(minimum: Pos, maximum: Pos)
    -> Range<Pos>

/// Forms a half-open range that contains `start`, but not
/// `end`. Requires: `start <= end`
func ..<<Pos : ForwardIndexType where Pos : Comparable>(start: Pos, end: Pos)
    -> Range<Pos>

/// Returns a closed interval from `start` through `end`
func ...<T : Comparable>(start: T, end: T) -> ClosedInterval<T>

/// Returns a half-open interval from `start` to `end`
func ..<<T : Comparable>(start: T, end: T) -> HalfOpenInterval<T>
```

不难发现, 其实这几个方法都是支持泛型的。除了我们常用的输入 `Int` 或者 `Double`, 返回一个

`Range` 以外，这个操作符还有一个接受 `Comparable` 的输入，并返回 `ClosedInterval` 或 `HalfOpenInterval` 的重载。在 `Swift` 中，除了数字以外另一个实现了 `Comparable` 的基本类型就是 `String`。也就是说，我们可以通过 `...` 或者 `..<` 来连接两个字符串。一个常见的使用场景就是检查某个字符是否是合法的字符。比如想确认一个单词里的全部字符都是小写英文字母的话，可以这么做：

```
let test = "heLLo"
let interval = "a..."z"
for c in test.characters {
    if !interval.contains(String(c)) {
        print("\(c) 不是小写字母")
    }
}

// 输出
// L 不是小写字母
```

在日常开发中，我们可能会需要确定某个字符是不是有效的 `ASCII` 字符，和上面的例子很相似，我们可以使用 `\0...~` 这样的 `ClosedInterval` 来进行 (`\0` 和 `~` 分别是 `ASCII` 的第一个和最后一个字符)。

AnyClass，元类型和 .self

在 Swift 中能够表示“任意”这个概念的除了 `Any` 和 `AnyObject` 以外，还有一个 `AnyClass`。 `AnyClass` 在 Swift 中被一个 `typealias` 所定义：

```
typealias AnyClass = AnyObject.Type
```

通过 `AnyObject.Type` 这种方式所得到的是一个元类型 (Meta)。在声明时我们总是在类型的名称后面加上 `.Type`，比如 `A.Type` 代表的是 `A` 这个类型的类型。也就是说，我们可以声明一个元类型来存储 `A` 这个类型本身，而在从 `A` 中取出其类型时，我们需要使用到 `.self`：

```
class A {  
  
}  
  
let typeA: A.Type = A.self
```

其实在 Swift 中，`.self` 可以用在类型后面取得类型本身，也可以用在某个实例后面取得这个实例本身。前一种方法可以用来获得一个表示该类型的值，这在某些时候会很有用；而后者因为拿到的实例本身，所以暂时似乎没有太多需要这么使用的案例。

了解了这个基础之后，我们就明白 `AnyObject.Type`，或者说 `AnyClass` 所表达的东西其实并没有什么奇怪，就是任意类型本身。所以，上面对于 `A` 的类型的取值，我们也可以强制让它是一个 `AnyClass`：

```
class A {  
  
}  
  
let typeA: AnyClass = A.self
```

这样，要是 `A` 中有一个类方法时，我们就可以通过 `typeA` 来对其进行调用了：

```
class A {  
    class func method() {  
        print("Hello")  
    }  
}  
  
let typeA: A.Type = A.self  
typeA.method()  
  
// 或者  
let anyClass: AnyClass = A.self  
(anyClass as! A.Type).method()
```

也许你会问，这样做有什么意义呢，我们难道不是可以直接使用 `A.method()` 来调用么？没错，对于单个独立的类型来说我们完全没有必要关心它的元类型，但是元类型或者元编程的概念可以变得非常灵活和强大，这在我们编写某些框架性的代码时会非常方便。比如我们想要传递一些类型的时候，就不需要不断地去改动代码了。在下面的这个例子中虽然我们是用代码声明的方式获取了 `MusicViewController` 和 `AlbumViewController` 的元类型，但是其实这一步骤完全可以通过读入配置文件之类的方式来进行完成的。而在将这些元类型存入数组并且传递给别的方法来进行配置这一点上，元类型编程就很难被替代了：

```
class MusicViewController: UIViewController {  
  
}  
  
class AlbumViewController: UIViewController {  
  
}  
  
let usingVCTypes: [AnyClass] = [MusicViewController.self,  
    AlbumViewController.self]  
  
func setupViewControllers(vcTypes: [AnyClass]) {  
    for vcType in vcTypes {  
        if vcType is UIViewController.Type {  
            let vc = (vcType as! UIViewController.Type).init()  
            print(vc)  
        }  
    }  
}  
  
setupViewControllers(usingVCTypes)
```

这么一来，我们完全可以搭好框架，然后用 DSL 的方式进行配置，就可以在不触及 Swift 编码的情况下，很简单地完成一系列复杂操作了。

另外，在 Cocoa API 中我们也常遇到需要一个 `AnyClass` 的输入，这时候我们也应该使用 `.self` 的方式来获取所需要的元类型，例如在注册 `tableView` 的 `cell` 的类型的时候：

```
self.tableView.registerClass(  
    UITableViewCell.self, forCellReuseIdentifier: "myCell")
```

`.Type` 表示的是某个类型的元类型，而在 Swift 中，除了 `class`，`struct` 和 `enum` 这三个类型外，我们还可以定义 `protocol`。对于 `protocol` 来说，有时候我们也会想取得接口的元类型。这时我们可以在某个 `protocol` 的名字后面使用 `.Protocol` 来获取，使用的方法和 `.Type` 是类似的。

接口和类方法中的 Self

我们在看一些接口的定义时，可能会注意到出现了首字母大写的 `Self` 出现在类型的位置上：

```
protocol IntervalType {
    //...

    /// Return `rhs` clamped to `self`. The bounds of the result, even
    /// if it is empty, are always within the bounds of `self`
    func clamp(intervalToClamp: Self) -> Self

    //...
}
```

比如上面这个 `IntervalType` 的接口定义了一个方法，接受实现该接口的自身的类型，并返回一个同样的类型。

这么定义是因为接口其实本身是没有自己的上下文类型信息的，在声明接口的时候，我们并不知道最后究竟会是什么样的类型来实现这个接口，Swift 中也不能在接口中定义泛型进行限制。而在声明接口时，我们希望在接口中使用的类型就是实现这个接口本身的类型的话，就需要使用 `Self` 进行指代。

但是在这种情况下，`Self` 不仅指代的是实现该接口的类型本身，也包括了这个类型的子类。从概念上来说，`Self` 十分简单，但是实际实现一个这样的方法却稍微要转个弯。为了说明这个问题，我们假设要实现一个 `Copyable` 的接口，满足这个接口的类型需要返回一个和接受方法调用的实例相同的拷贝。一开始我们可能考虑的接口是这样的：

```
protocol Copyable {
    func copy() -> Self
}
```

这是很直接明了的，它应该做的是创建一个和接受这个方法的对象同样的东西，然后将其返回，返回的类型不应该发生改变，所以写为 `Self`。然后开始尝试实现一个 `MyClass` 来满足这个接口：

```
class MyClass: Copyable {

    var num = 1

    func copy() -> Self {
        // TODO: 返回什么?
        // return
    }
}
```

我们一开始的时候可能回写类似这样的代码：

这是错误代码

```
func copy() -> Self {
    let result = MyClass()
    result.num = num
    return result
}
```

但是显然类型是有问题的，因为该方法要求返回一个抽象的、表示当前类型的 `Self`，但是我们却返回了它的真实类型 `MyClass`，这导致了无法编译。也许你会尝试把方法声明中的 `Self` 改为 `MyClass`，这样声明就和实际返回一致了，但是很快你会发现这样的话，实现的方法又和接口中的定义不一样了，依然不能编译。

为了解决这个问题，我们在这里需要的是通过一个和当前上下文 (也就是和 `MyClass`) 无关的，又能够指代当前类型的方式进行初始化。希望你还能记得我们在[对象类型](#)中所提到的 `dynamicType`，这里我们就可以使用它来做初始化，以保证方法与当前类型上下文无关，这样不论是 `MyClass` 还是它的子类，都可以正确地返回合适的类型满足 `Self` 的要求：

```
func copy() -> Self {
    let result = self.dynamicType.init()
    result.num = num
    return result
}
```

但是很不幸，单单是这样还是无法通过编译，编译器提示我们如果想要构建一个 `Self` 类型的对象的话，需要有 `required` 关键字修饰的初始化方法，这是因为 `Swift` 必须保证当前类和其子类都能响应这个 `init` 方法。在这个例子中，我们添加上一个 `required` 的 `init` 就行了。最后，`MyClass` 类型是这样的：

```
class MyClass: Copyable {

    var num = 1

    func copy() -> Self {
        let result = self.dynamicType.init()
        result.num = num
        return result
    }

    required init() {

    }
}
```

我们可以通过测试来验证一下行为的正确性：

```
let object = MyClass()
object.num = 100
```

```
let newObject = object.copy()
object.num = 1

print(object.num)    // 1
print(newObject.num) // 100
```

而对于 `MyClass` 的子类，`copy()` 方法也能正确地返回子类的经过拷贝的对象了。

另一个可以使用 `self` 的地方是在类方法中，使用起来也十分相似，核心就在于保证子类也能返回恰当的类型。

动态类型和多方法

Swift 中我们虽然可以通过 `dynamicType` 来获取一个对象的动态类型 (也就是运行时的实际类型, 而非代码指定或编译器看到的类型)。但是在使用中, **Swift** 现在却是不支持多方法的, 也就是说, 不能根据对象在动态时的类型进行合适的重载方法调用。

举个例子来说, 在 **Swift** 里我们可以重载同样名字的方法, 而只需要保证参数类型不同:

```
class Pet {}
class Cat: Pet {}
class Dog: Pet {}

func printPet(pet: Pet) {
    print("Pet")
}

func printPet(cat: Cat) {
    print("Meow")
}

func printPet(dog: Dog) {
    print("Bark")
}
```

在对这些方法进行调用时, 编译器将帮助我们找到最精确的匹配:

```
printPet(Cat()) // Meow
printPet(Dog()) // Bark
printPet(Pet()) // Pet
```

对于 `Cat` 或者 `Dog` 的实例, 总是会寻找最合适的方法, 而不会去调用一个通用的父类 `Pet` 的方法。这一切的行为都是发生在编译时的, 如果我们写了下面这样的代码:

```
func printThem(pet: Pet, _ cat: Cat) {
    printPet(pet)
    printPet(cat)
}

printThem(Dog(), Cat())

// 输出:
// Pet
// Meow
```

打印时的 `Dog()` 的类型信息并没有被用来在运行时选择合适的 `printPet(dog: Dog)` 版本的方法, 而是被忽略掉, 并采用了编译期间决定的 `Pet` 版本的方法。因为 **Swift** 默认情况下是不采用动态派发的, 因此方法的调用只能在编译时决定。

要想绕过这个限制，我们可能需要进行通过对输入类型做判断和转换：

```
func printThem(pet: Pet, _ cat: Cat) {  
    if let aCat = pet as? Cat {  
        printPet(aCat)  
    } else if let aDog = pet as? Dog {  
        printPet(aDog)  
    }  
    printPet(cat)  
}  
  
// 输出:  
// Bark  
// Meow
```

属性观察

属性观察 (Property Observers) 是 Swift 中一个很特殊的特性，利用属性观察我们可以在当前类型内监视对于属性的设定，并作出一些响应。Swift 中为我们提供了两个属性观察的方法，它们分别是 `willSet` 和 `didSet`。

使用这两个方法十分简单，我们只要在属性声明的时候添加相应的代码块，就可以对将要设定的值和已经设置的值进行监听了：

```
class MyClass {
    var date: NSDate {
        willSet {
            let d = date
            print("即将将日期从 \(d) 设定至 \(newValue)")
        }

        didSet {
            print("已经将日期从 \(oldValue) 设定至 \(date)")
        }
    }

    init() {
        date = NSDate()
    }
}

let foo = MyClass()
foo.date = foo.date.dateByAddingTimeInterval(10086)

// 输出
// 即将将日期从 2014-08-23 12:47:36 +0000 设定至 2014-08-23 15:35:42 +0000
// 已经将日期从 2014-08-23 12:47:36 +0000 设定至 2014-08-23 15:35:42 +0000
```

在 `willSet` 和 `didSet` 中我们分别可以使用 `newValue` 和 `oldValue` 来获取将要设定的和已经设定的值。属性观察的一个重要用处是作为设置值的验证，比如上面的例子中我们不希望 `date` 超过当前时间的一年以上的的话，我们可以将 `didSet` 修改一下：

```
class MyClass {
    let oneYearInSeconds: NSTimeInterval = 365 * 24 * 60 * 60
    var date: NSDate {

        //...

        didSet {
            if (date.timeIntervalSinceNow > oneYearInSeconds) {
                println("设定的时间太晚了！")
                date = NSDate().dateByAddingTimeInterval(oneYearInSeconds)
            }
            print("已经将日期从 \(oldValue) 设定至 \(date)")
        }
    }

    //...
}
```

更改一下调用，我们就能看到效果：

```
// 365 * 24 * 60 * 60 = 31_536_000
foo.date = foo.date.dateByAddingTimeInterval(100_000_000)

// 输出
// 即将将日期从 2014-08-23 13:24:14 +0000 设定至 2017-10-23 23:10:54 +0000
// 设定的时间太晚了！
// 已经将日期从 2014-08-23 13:24:14 +0000 设定至 2015-08-23 13:24:14 +0000
```

初始化方法对属性的设定，以及在 `willSet` 和 `didSet` 中对属性的再次设定都不会再次触发属性观察的调用，一般来说这会是你所需要的行为，可以放心使用能够。

我们知道，在 **Swift** 中所声明的属性包括存储属性和计算属性两种。其中存储属性将会在内存中实际分配地址对属性进行存储，而计算属性则不包括背后的存储，只是提供 `set` 和 `get` 两种方法。在同一个类型中，属性观察和计算属性是不能同时共存的。也就是说，想在一个属性定义中同时出现 `set` 和 `willSet` 或 `didSet` 是一件办不到的事情。计算属性中我们可以通过改写 `set` 中的内容来达到和 `willSet` 及 `didSet` 同样的属性观察的目的。如果我们无法改动这个类，又想要通过属性观察做一些事情的话，可能就需要子类化这个类，并且重写它的属性了。重写的属性并不知道父类属性的具体实现情况，而只从父类属性中继承名字和类型，因此在子类的重载属性中我们可以对父类的属性任意地添加属性观察的，而不用在意父类中到底是存储属性还是计算属性：

```
class A {
    var number :Int {
        get {
            print("get")
            return 1
        }

        set {print("set")}
    }
}

class B: A {
    override var number: Int {
        willSet {print("willSet")}
        didSet {print("didSet")}
    }
}
```

调用 `number` 的 `set` 方法可以看到工作的顺序

```
let b = B()
b.number = 0

// 输出
// get
// willSet
// set
```

```
// didSet
```

`set` 和对应的属性观察的调用都在我们的预想之中。这里要注意的是 `get` 首先被调用了一次。这是因为我们实现了 `didSet`，`didSet` 中会用到 `oldValue`，而这个值需要在整个 `set` 动作之前进行获取并存储待用，否则将无法确保正确性。如果我们不实现 `didSet` 的话，这次 `get` 操作也将不存在。

final

`final` 关键字可以用在 `class` , `func` 或者 `var` 前面进行修饰, 表示不允许对该内容进行继承或者重写操作。这个关键字的作用和 C# 中的 `sealed` 相同, 而 `sealed` 其实在 C# 算是一个饱受争议的关键字。有一派程序员认为, 类似这样的禁止继承和重写的做法是非常有益的, 它可以更好地对代码进行版本控制, 得到更佳的性能, 以及使代码更安全。因此他们甚至认为语言应当是默认不允许继承的, 只有在显式地指明可以继承的时候才能子类化。

在这里我不打算对这样的想法做出判断或者评价, 虽然上面列举的优点都是事实, 但是另一个事实是不论是 Apple 或者微软, 以及世界上很多其他语言都没有作出默认不让继承和重写的决定。带着“这不是一个可以滥用的特性”的观点, 我们来看看在写 Swift 的时候可能会在什么情况下使用 `final` 。

权限控制

给一段代码加上 `final` 就意味着编译器向你作出保证, 这段代码不会再被修改; 同时, 这也意味着你认为这段代码已经完备并且没有再被进行继承或重写的必要, 因此这往往会是一个需要深思熟虑的决定。在 Cocoa 开发中 app 开发是一块很大的内容, 对于大多数我们自己完成的面向 app 开发代码, 其实不太会提供给别人使用, 这种情况下即使是将所有自己写的代码标记为 `final` 都是一件无可厚非的事情 (但我并不是在鼓励这么做) -- 因为在需要的任何时候你都可以将这个关键字去掉以恢复其可继承性。而在开发给其他开发者使用的库时, 就必须更深入地考虑各种使用场景和需求了。

一般来说, 不希望被继承和重写会有这几种情况:

类或者方法的功能确实已经完备了

对于很多的辅助性质的工具类或者方法, 可能我们会考虑加上 `final` 。这样的类有一个比较大的特点, 是很可能只包含类方法而没有实例方法。比如我们很难想到一种情况需要继承或重写一个负责计算一段字符串的 MD5 或者 AES 加密解密的工具类。这种工具类和方法的算法是经过完备验证和固定的, 使用者只需要调用, 而相对来说不可能有继承和重写的需求。

这种情况很多时候遵循的是以往经验和主观判断, 而单个的开发者的判断其实往往并不可靠。遇到希望把某个自己开发的类或者方法标为 `final` 的时候, 去找几个富有经验的开发者, 问问他们的意见或者看法, 应该是一个比较靠谱的做法。

子类继承和修改是一件危险的事情

在子类继承或重写某些方法后可能做一些破坏性的事情, 导致子类或者父类部分也无法正常工作的情况。举个例子, 在某个公司管理的系统中我们对员工按照一定规则进行编号, 这样通过编号我们能迅速找到任一员工。而假如我们在子类中重写了这个编号方法, 很可能就导致基类中的依赖员工编号的方法失效。在这类情况下, 将编号方法标记为 `final` 以确保稳定, 可能是一种更好

的做法。

为了父类中某些代码一定会被执行

有时候父类中有一些关键代码是在被继承重写后必须执行的 (比如状态配置, 认证等等), 否则将导致运行时候的错误。而在一般的方法中, 如果子类重写了父类方法, 是没有办法强制子类方法一定去调用相同的父类方法的。在 **Objective-C** 的时候我们可以通过指定

`__attribute__((objc_requires_super))` 这样的属性来让编译器在子类没有调用父类方法时抛出警告。在 **Swift** 中对原来的很多 `attribute` 的支持现在还缺失中, 为了达到类似的目的, 我们可以使用一个 `final` 的方法, 在其中进行一些必要的配置, 然后再调用某个需要子类实现的方法, 以确保正常运行:

```
class Parent {  
  
    final func method() {  
        print("开始配置")  
        // ..必要的代码  
  
        methodImpl()  
  
        // ..必要的代码  
        print("结束配置")  
    }  
  
    func methodImpl() {  
        fatalError("子类必须实现这个方法")  
        // 或者也可以给出默认实现  
    }  
  
}  
  
class Child: Parent {  
    override func methodImpl() {  
        // ..子类的业务逻辑  
    }  
}
```

这样, 无论如何我们如何使用 `method`, 都可以保证需要的代码一定被运行过, 而同时又给了子类继承和重写自定义具体实现的机会。

性能考虑

使用 `final` 的另一个重要理由是可能带来的性能改善。因为编译器能够从 `final` 中获取额外的信息, 因此可以对类或者方法调用进行额外的优化处理。但是这个优势在实际表现中可能带来的好处其实就算与 **Objective-C** 的动态派发相比也十分有限, 因此在项目还有其他方面可以优化 (一般来说会是算法或者图形相关的内容导致性能瓶颈) 的情况下, 并不建议使用将类或者方法转为 `final` 的方式来追求性能的提升。

lazy 修饰符和 lazy 方法

延时加载或者说延时初始化是很常用的优化方法，在构建和生成新的对象的时候，内存分配会在运行时耗费不少时间，如果有一些对象的属性和内容非常复杂的话，这个时间更是不可忽略。另外，有些情况下我们并不会立即用到一个对象的所有属性，而默认情况下初始化时，那些在特定环境下不被使用的存储属性，也一样要被初始化和赋值，也是一种浪费。

在其他语言 (包括 Objective-C) 中延时加载的情况是很常见的。我们在第一次访问某个属性时，判断这个属性背后的存储是否已经存在，如果存在则直接返回，如果不存在则说明是首次访问，那么就进行初始化并存储后再返回。这样我们可以把这个属性的初始化时刻推迟，与包含它的对象的初始化时刻分开，以达到提升性能的目的。以 Objective-C 举个例子 (虽然这里既没有费时操作，也不会因为使用延时加载而造成什么性能影响，但是作为一个最简单的例子，可以很好地说明问题)：

```
// ClassA.h
@property (nonatomic, copy) NSString *testString;

// ClassA.m
- (NSString *)testString {
    if (!_testString) {
        _testString = @"Hello";
        NSLog(@"只在首次访问输出");
    }
    return _testString;
}
```

在初始化 ClassA 对象后，`_testString` 是 `nil`。只有当首次访问 `testString` 属性时 `getter` 方法会被调用，并检查如果还没有初始化的话，就进行赋值。为了方便确认，我们还在赋值时打印了一句 `log`。我们之后再多次访问这个属性的话，因为 `_testString` 已经有值，因此将直接返回。

在 Swift 中我们使用在变量属性前加 `lazy` 关键字的方式来简单地指定延时加载。比如上面的的代码我们在 Swift 中重写的话，会是这样：

```
class ClassA {
    lazy var str: String = {
        let str = "Hello"
        print("只在首次访问输出")
        return str
    }()
}
```

我们在使用 `lazy` 作为属性修饰符时，只能声明属性是变量。另外我们需要显式地指定属性类型，并使用一个可以对这个属性进行赋值的语句来在首次访问属性时运行。如果我们多次访问这个实例的 `str` 属性的话，可以看到只有一次输出。

为了简化，我们如果不需要做什么额外工作的话，也可以对这个 `lazy` 的属性直接写赋值语句：

```
lazy var str: String = "Hello"
```

相比起在 Objective-C 中的实现方法，现在的 lazy 使用起来要方便得多。

另外一个不太引起注意的是，在 Swift 的标准库中，我们还有一组 lazy 方法，它们的定义是这样的：

```
func lazy<S : SequenceType>(s: S) -> LazySequence<S>

func lazy<S : CollectionType where S.Index : RandomAccessIndexType>(s: S)
    -> LazyRandomAccessCollection<S>

func lazy<S : CollectionType where S.Index : BidirectionalIndexType>(s: S)
    -> LazyBidirectionalCollection<S>

func lazy<S : CollectionType where S.Index : ForwardIndexType>(s: S)
    -> LazyForwardCollection<S>
```

这些方法可以配合像 map 或是 filter 这类接受闭包并进行运行的方法一起，让整个行为变成延时进行的。在某些情况下这么做也对性能会有不小的帮助。例如，直接使用 map 时：

```
let data = 1...3
let result = data.map {
    (i: Int) -> Int in
    print("正在处理 \(i)")
    return i * 2
}

print("准备访问结果")
for i in result {
    print("操作后结果为 \(i)")
}

print("操作完毕")
```

这么做的输出为：

```
// 正在处理 1
// 正在处理 2
// 正在处理 3
// 准备访问结果
// 操作后结果为 2
// 操作后结果为 4
// 操作后结果为 6
// 操作完毕
```

而如果我们先进行一次 lazy 操作的话，我们就能得到延时运行版本的容器：

```
let data = 1...3
```

```
let result = lazy(data).map {  
    (i: Int) -> Int in  
    print("正在处理 \${i}")  
    return i * 2  
}  
  
print("准备访问结果")  
for i in result {  
    print("操作后结果为 \${i}")  
}  
  
print("操作完毕")
```

此时的运行结果：

```
// 准备访问结果  
// 正在处理 1  
// 操作后结果为 2  
// 正在处理 2  
// 操作后结果为 4  
// 正在处理 3  
// 操作后结果为 6  
// 操作完毕
```

对于那些不需要完全运行，可能提前退出的情况，使用 `lazy` 来进行性能优化效果会非常有效。

Reflection 和 Mirror

熟悉 Java 的读者可能会知道反射 (Reflection)。这是一种在运行时检测、访问或者修改类型的行为的特性。一般的静态语言类型的结构和方法的调用等都需要在编译时决定，开发者能做的很多时候只是使用控制流 (比如 if 或者 switch) 来决定做出怎样的设置或是调用哪个方法。而反射特性可以让我们有机会在运行的时候通过某些条件实时地决定调用的方法，或者甚至向某个类型动态地设置甚至加入属性及方法，是一种非常灵活和强大的语言特性。

Objective-C 中我们不太会经常提及到“反射”这样的词语，因为 Objective-C 的运行时比一般的反射还要灵活和强大。可能很多读者已经习以为常的像是通过字符串生成类或者 selector，并且进而生成对象或者调用方法等，其实都是反射的具体表现。而在 Swift 中其实就算抛开 Objective-C 的运行时的部分，在纯 Swift 范畴内也存在有反射相关的一些内容，只不过相对来说功能要弱得多。

因为这部分内容并没有公开的文档说明，所以随时可能发生变动，或者甚至存在今后被从 Swift 的可调用标准库中去掉的可能 (Apple 已经干过这种事情，最早的时候 Swift 中甚至有隐式的类型转换 __conversion，但因为太过危险，而被彻底去除了。现在隐式转换必须使用[字面量转换](#)的方式进行了)。在实际的项目中，也不建议使用这种没有文档说明的 API，不过有时候如果能稍微知道 Swift 中也存在这样的可能性的话，也许会有帮助 (也说不定哪天 Apple 就扔出一个完整版的反射功能呢)。

Swift 中所有的类型都实现了 _Reflectable，这是一个内部接口，我们可以通过 _reflect 来获取任意对象的一个镜像，这个镜像对象包含类型的基本信息，在 Swift 2.0 之前，这是对某个类型的对象进行探索的一种方法。在 Swift 2.0 中，这些方法已经从公开的标准库中移除了，取而代之，我们可以使用 Mirror 来

```
struct Person {
    let name: String
    let age: Int
}

let xiaoMing = Person(name: "XiaoMing", age: 16)
let r = Mirror(reflecting: xiaoMing) // r 是 MirrorType

print("xiaoMing 是 \(r.displayStyle!)" )

print("属性个数:\(r.children.count)")
for i in r.children.startIndex..
```

通过 Mirror 初始化得到的结果中包含的元素的描述都被集合在 children 属性下，如果你有心可

以到 **Swift** 标准库中查找它的定义，它实际上是一个 `Child` 的集合，而 `Child` 则是一对键值的多元组：

```
public typealias Child = (label: String?, value: Any)
public typealias Children = AnyForwardCollection<Child>
```

`AnyForwardCollection` 是遵守 `CollectionType` 接口的，因此我们可以简单地使用 `count` 来获取元素的个数，而对于具体的代表属性的多元组，则使用下标进行访问。在对于我们的例子中，每个 `Child` 都是具有两个元素的多元组，其中第一个是属性名，第二个是这个属性所存储的值。需要特别注意的是，这个值有可能是多个元素组成嵌套的形式（例如属性值是数组或者字典的话，就是这样的形式）。

如果觉得一个个打印太过于麻烦，我们也可以简单地使用 `dump` 方法来通过获取一个对象的镜像并进行标准输出的方式将其输出出来。比如对上面的对象 `xiaoMing`：

```
dump(xiaoMing)
// 输出:
// ▽ Person
//   - name: XiaoMing
//   - age: 16
```

在这里因为篇幅有限，而且这部分内容很可能随着版本而改变，我们就不再一一介绍 `Mirror` 的更详细的内容了。有兴趣的读者不妨打开 **Swift** 的定义文件并找到这个接口，里面对每个属性和方法的作用有非常详细的注释。

对于一个从对象反射出来的 `Mirror`，它所包含的信息是完备的。也就是说我们可以在运行时通过 `Mirror` 的手段了解一个 **Swift** 类型（当然 `NSObject` 类也可以）的实例的属性信息。该特性最容易想到的应用的特性就是为任意 `model` 对象生成对应的 **JSON** 描述。我们可以对等待处理的对象的 `Mirror` 值进行深度优先的访问，并按照属性的 `valueType` 将它们归类对应到不同的格式化中。

另一个常见的应用场景是类似对 **Swift** 类型的对象做像 **Objective-C** 中 **KVC** 那样的 `valueForKey:` 的取值。通过比较取到的属性的名字和我们想要取得的 `key` 值就行了，非常简单：

```
func valueFrom(object: Any, key: String) -> Any? {
    let mirror = Mirror(reflecting: object)

    for i in mirror.children.startIndex..
```

```
}  
  
// 输出:  
// 通过 key 得到值: XiaoMing
```

在现在的版本中，Swift 的反射特性并不是非常强大，我们只能对属性进行读取，还不能对其设定，不过我们有希望能在将来的版本中获得更为强大的反射特性。另外需要特别注意的是，虽然理论上将反射特性应用在实际的 app 制作中是可行的，但是这一套机制设计的最初目的是用于 REPL 环境和 Playground 中进行输出的。所以我们最好遵守 Apple 的这一设定，只在 REPL 和 Playground 中用它来对一个对象进行深层次的探索，而避免将它用在 app 制作中 -- 因为你永远不知道什么时候它们就会失效或者被大幅改动。

隐式解包 Optional

相对于普通的 `Optional` 值，在 `Swift` 中我们还有一种特殊的 `Optional`，在对它的成员或者方法进行访问时，编译器会帮助我们自动进行解包，这就是 `ImplicitlyUnwrappedOptional`。在声明的时候，我们可以通过在类型后加上一个感叹号 (!) 这个语法糖来告诉编译器我们需要一个可以隐式解包的 `Optional` 值：

```
var maybeObject: MyClass!
```

首先需要明确的是，隐式解包的 `Optional` 本质上与普通的 `Optional` 值并没有任何不同，只是我们在对这类变量的成员或方法进行访问的时候，编译器会自动为我们在后面插入解包符号 !，也就是说，对于一个隐式解包的下面的两种写法是等效的：

```
var maybeObject: MyClass! = MyClass()  
maybeObject!.foo()  
maybeObject.foo()
```

我们知道，如果 `maybeObject` 是 `nil` 的话那么这两种不加检查的写法的调用都会导致程序崩溃。而如果 `maybeObject` 是普通的 `Optional` 的话，我们就只能使用第一种显式地加感叹号的写法，这能提醒我们也许应该使用 `if let` 的 `Optional Binding` 的形式来处理。而对隐式解包来说，后一种写法看起来就好像我们操作的 `maybeObject` 确实是 `MyClass` 类的实例，不需要对其检查就可以使用 (当然实际上这不是真的)。为什么一向以安全著称的 `Swift` 中会存在隐式解包并可以写出让人误认为能直接访问的这种危险写法呢？

一切都是历史的错。因为 `Objective-C` 中 `Cocoa` 的所有类型变量都可以指向 `nil` 的，有一部分 `Cocoa` 的 `API` 中在参数或者返回时即使被声明为具体的类型，但是还是有可能在某些特定情况下是 `nil`，而同时也有另一部分 `API` 永远不会接收或者返回 `nil`。在 `Objective-C` 时，这两种情况并没有被加以区别，因为 `Objective-C` 里向 `nil` 发送消息并不会有什么不良影响。在将 `Cocoa` `API` 从 `Objective-C` 转为 `Swift` 的 `module` 声明的自动化工具里，是无法判定是否存在 `nil` 的可能的，因此也无法决定哪些类型应该是实际的类型，而哪些类型应该声明为 `Optional`。

在这种自动化转换中，最简单粗暴的应对方式是全部转为 `Optional`，然后让使用者通过 `Optional Binding` 来判断并使用。虽然这是最安全的方式，但对使用者来说是一件非常麻烦的事情，我猜不会有人喜欢每次用个 `API` 就在 `Optional` 和普通类型之间转来转去。这时候，隐式解包的 `Optional` 就作为一个妥协方案出现了。使用隐式解包 `Optional` 的最大好处是对于那些我们能确认的 `API` 来说，我们可直接进行属性访问和方法调用，会很方便。但是需要牢记在心的是，隐式解包不意味着“这个变量不会是 `nil`，你可以放心使用”这种暗示，只能说 `Swift` 通过这个特性给了我们一种简便但是危险的使用方式罢了。

另外，其实在 `Apple` 的不断修改 (我相信这是一件消耗大量人月的手工工作) 下，在 `Swift` 的正式版本中，已经没有太多的隐式解包的 `API` 了。最近 `Objective-C` 中又加入了像是 `nonnull` 和

`nullable` 这样的修饰符，这样一来，那些真正有可能为 `nil` 的返回可以被明确定义为普通的 `Optional` 值，而那些不会是 `Optional` 的值，也根据情况转换为了确定的类型。现在比较常见的隐式解包的 `Optional` 就只有使用 `Interface Builder` 时建立的 `IBOutlet` 了：

```
@IBOutlet weak var button: UIButton!
```

如果没有连接 `IB` 的话，对 `button` 的直接访问会导致应用崩溃，这种情况和错误在调试应用时是很容易被发现的问题。在我们的代码的其他部分，还是少用这样的隐式解包的 `Optional` 为好，很多时候多写一个 `Optional Binding` 就可以规避掉不少应用崩溃的风险。

多重 Optional

Optional 可以说是 Swift 的一大特色，它完全解决了“有”和“无”这两个困扰了 Objective-C 许久的哲学概念，也使得代码安全性得到了很大的增加。但是一个陷阱 -- 或者说一个很容易让人迷惑的概念 -- 也随之而来，那就是多重的 Optional。

在深入讨论之前，可以让我们先看看 Optional 是什么。很多读者应该已经知道，我们使用的类型后加上 `?` 的语法只不过是 Optional 类型的语法糖，而实际上这个类型是一个 enum：

```
enum Optional<T> : _Reflectable, NilLiteralConvertible {
    case None
    case Some(T)

    //...
}
```

在这个定义中，对 `T` 没有任何限制，也就是说，我们是可以在 Optional 中装入任意东西的，甚至也包括 Optional 对象自身。打个形象的比方，如果我们将 Optional 比作一个盒子，实际具体的 String 或者 Int 这样的值比作糖果的话，当我们打开一个盒子 (unwrap) 时，可能的结果会有三个 -- 空气，糖果，或者另一个盒子。

空气和糖果都很好理解，也十分直接。但是对于盒子中的盒子，有时候使用时就相当容易出错。特别是在和各种字面量转换混用的时候需要特别注意。

对于下面这种形式的写法：

```
var string: String? = "string"
var anotherString: String?? = string
```

我们可以很明白地知道 anotherString 是 Optional<Optional<String>>。但是除开将一个 Optional 值赋给多重 Optional 以外，我们也可以将直接的字面量值赋给它：

```
var literalOptional: String?? = "string"
```

这种情况还好，根据类型推断我们只能将 Optional<String> 放入到 literalOptional 中，所以可以猜测它与上面提到的 anotherString 是等效的。但是如果我们是将 nil 赋值给它的话，情况就有所不同了。考虑下面的代码：

```
var aNil: String? = nil

var anotherNil: String?? = aNil
```

```
var literalNil: String?? = nil
```

`anotherNil` 和 `literalNil` 是不是等效的呢？答案是否定的。`anotherNil` 是盒子中包了一个盒子，打开内层盒子的时候我们会发现空气；但是 `literalNil` 是盒子中直接是空气。使用中一个最显著的区别在于：

```
if let a = anotherNil {  
    print("anotherNil")  
}  
  
if let b = literalNil {  
    print("literalNil")  
}
```

这样的代码只能输出 `anotherNil`。

另一个值得注意的地方时在Playground 中运行时，或者在用lldb 进行调试时，直接使用 `po` 指令打印 `Optional` 值的话，为了看起来方便，lldb 会将要打印的 `Optional` 进行展开。如果我们直接打印上面的 `anotherNil` 和 `literalNil`，得到的结果都是 `nil`：

```
(lldb) po anotherNil  
nil  
  
(lldb) po literalNil  
nil
```

如果我们遇到了多重 `Optional` 的麻烦的时候，这显然对我们是没有太大帮助的。我们可以使用 `fr` `v -R` 命令来打印出变量的未加工过时的信息，就像这样：

```
(lldb) fr v -R anotherNil  
(Swift.Optional<Swift.Optional<Swift.String>>)  
  anotherNil = Some {  
    ... 中略  
  }  
(lldb) fr v -R literalNil  
(Swift.Optional<Swift.Optional<Swift.String>>)  
  literalNil = None {  
    ... 中略  
  }
```

这样我们就能清晰地分辨出两者的区别了。

Optional Map

我们经常会对 `Array` 类型使用 `map` 方法，这个方法能对数组中的所有元素应用某个规则，然后返回一个新的数组。我们可以在 `CollectionType` 的 `extension` 中找到这个方法的定义：

```
extension CollectionType {
    public func map<T>(@noescape transform:
                      (Self.Generator.Element) -> T) -> [T]

    //...
}
```

举个一个简单的使用例子：

```
let arr = [1,2,3]
let doubled = arr.map{
    $0 * 2
}

print(doubled)
// 输出:
// [2,4,6]
```

这很方便，而且在其他一些语言里 `map` 可以说是很常见也很常用的一个语言特性了。因此当这个特性出现在 `Swift` 中时，也赢得了 `iOS/Mac` 开发者们的欢迎。

现在假设我们有个需求，要将某个 `Int?` 乘 2。一个合理的策略是如果这个 `Int?` 有值的话，就取出值进行乘 2 的操作，如果是 `nil` 的话就直接将 `nil` 赋给结果。依照这个策略，我们可以写出如下代码：

```
let num: Int? = 3

var result: Int?
if let realNum = num {
    result = realNum * 2
} else {
    result = nil
}
```

其实我们有更优雅简洁的方式，那就是使用 `Optional` 的 `map`。对的，不仅在 `Array` 或者说 `CollectionType` 里可以用 `map`，如果我们仔细看过 `Optional` 的声明的话，会发现它也有一个 `map` 方法：

```
public enum Optional<T> :
    _Reflectable, NilLiteralConvertible {
```

```

//...

/// If `self == nil`, returns `nil`. Otherwise, returns `f(self!)`.
public func map<U>(@noescape f: (T) -> U) -> U?

//...
}

```

这个方法能让我们很方便地对一个 **Optional** 值做变化和操作，而不必进行手动的解包工作。输入会被自动用类似 **Optinal Binding** 的方式进行判断，如果有值，则进入 `f` 的闭包进行变换，并返回一个 `U?`；如果输入就是 `nil` 的话，则直接返回值为 `nil` 的 `U?`。

有了这个方法，上面的代码就可以大大简化，而且 `result` 甚至可以使用常量值：

```

let num: Int? = 3
let result = num.map {
    $0 * 2
}

// result 为 {Some 6}

```

Protocol Extension

Swift 2 中引入了一个非常重要的特性，那就是 `protocol extension`。在 Swift 1.x 中，`extension` 仅只能作用在实际的类型上 (也就是 `class`, `struct` 等等)，而不能扩展一个 `protocol`。但是 Swift 的功能实现基本都是基于 `protocol` 来构建的，举个最简单的例子，我们每天使用的 `Array` 就是遵守了 `CollectionType` 这个 `protocol` 的。`CollectionType` 可以说是 Swift 中非常重要的接口，除了 `Array` 以外，像是 `Dictionary` 和 `Set` 也实现了这个接口所定义的内容。

在 `protocol` 不能被扩展的时候，当我们想要为实现了某个接口的类型所有类型添加一些另外的共通的功能时，会非常麻烦。一个很好的例子是 Swift 1.x 时的像是 `map` 或者 `filter` 这样的函数。大体来说，我们有两种思路进行添加：第一种方式是在接口中定义这个方法，然后在所有实现了这个接口的类型中都去实现一遍。每有一个这样的类型，我们就需要写一份类似甚至相同的方法，这显然是很麻烦的，而且完全没有可维护性。另一种方法是在全局范围实现一个接收这个 `protocol` 的方法，相比于前一种方式，我们只需要维护一份代码，显然要好不少，但是缺点在于在全局作用域中引入了只和特定 `protocol` 有关的东西，这也不符合代码设计的美学。作为妥协，Apple 在 Swift 1.x 中采用的是全局方法，如果你尝试寻找的话，可以在标准库的全局 `scope` 中找到像是 `map` 和 `filter` 这样的方法。

在 Swift 2 中这个问题被彻底解决了。现在我们可以对一个已有的 `protocol` 进行扩展，而扩展中实现的方法将作为实现扩展的类型的默认实现。也就是说，假设我们有下面的 `protocol` 声明，以及一个对该接口的扩展：

```
protocol MyProtocol {
    func method()
}

extension MyProtocol {
    func method() {
        print("Called")
    }
}
```

在具体的实现这个接口的类型中，即使我们什么都不写，也可以编译通过。进行调用的话，会直接使用 `extension` 中的实现：

```
struct MyStruct: MyProtocol {

}

MyStruct().method()
// 输出:
// Called in extension
```

当然，如果我们需要在类型中进行其他实现的话，可以像以前那样在具体类型中添加这个方法：

```

struct MyStruct: MyProtocol {
    func method() {
        print("Called in struct")
    }
}

MyStruct().method()
// 输出:
// Called in struct

```

也就是说，`protocol extension` 为 `protocol` 中定义的方法提供了一个默认的实现。`c`有了这个特性以后，之前被放在全局环境中的接受 `CollectionType` 的 `map` 方法，就可以被移动到 `CollectionType` 的接口扩展中去了：

```

extension CollectionType {
    public func map<T>(@noescape transform: (Self.Generator.Element) -> T) -> [T]
    //...
}

```

在日常开发中，另一个可以用到 `protocol extension` 的地方是 `optional` 的接口方法。关于这一点，我们在[可选接口和接口扩展](#)一节中已经讲述过，就不再重复了。

对于 `protocol extension` 来说，有一种会非常让人迷惑的情况，就是在接口的扩展中实现了接口里没有定义的方法时的情况。举个例子，比如我们定义了这样的一个接口和它的一个扩展：

```

protocol A1 {
    func method1() -> String
}

struct B1: A1 {
    func method1() -> String {
        return "hello"
    }
}

```

在使用的时候，无论我们将实例的类型为 `A1` 还是 `B1`，因为实现只有一个，所以没有任何疑问，调用方法时的输出都是“hello”：

```

let b1 = B1() // b1 is B1
b1.method1()
// hello

let a1: A1 = B1()
// a1 is A1
a1.method1()
// hello

```

但是如果在接口里只定义了一个方法，而在接口扩展中实现了额外的方法的话，事情就变得有趣起来了。考虑下面这组接口和它的扩展：


```
protocol A2 {
    func method1() -> String
}

extension A2 {
    func method1() -> String {
        return "hi"
    }

    func method2() -> String {
        return "hi"
    }
}
```

扩展中除了实现接口定义的 `method1` 之外，还定义了一个接口中不存在的方法 `method2`。我们尝试来实现这个接口：

```
struct B2: A2 {
    func method1() -> String {
        return "hello"
    }

    func method2() -> String {
        return "hello"
    }
}
```

`B2` 中实现了 `method1` 和 `method2`。接下来，我们尝试初始化一个 `B2` 对象，然后对这两个方法进行调用：

```
let b2 = B2()

b2.method1() // hello
b2.method2() // hello
```

结果在我们的意料之中，虽然在 `protocol extension` 中已经实现了这两个方法，但是它们只是默认的实现，我们在具体实现接口的类型中可以对默认实现进行覆盖，这非常合理。但是如果我们稍作改变，在上面的代码后面继续添加：

```
let a2 = b2 as A2

a2.method1() // hello
a2.method2() // hi
```

`a2` 和 `b2` 是同一个对象，只不过我们通过 `as` 告诉编译器我们在这里需要的类型是 `A2`。但是这时候在这个同样的对象上调用同样的方法调用却得到了不同的结果，发生了什么？

我们可以看到，对 `a2` 调用 `method2` 实际上是接口扩展中的方法被调用了，而不是 `a2` 实例中的

方法被调用。我们不妨这样来理解：对于 `method1`，因为它在 `protocol` 中被定义了，因此对于一个被声明为遵守接口的类型的实例 (也就是对于 `a2`) 来说，可以确定实例必然实现了 `method1`，我们可以放心大胆地用动态派发的方式使用最终的实现 (不论它是在类型中具体实现的还是在接口扩展中的默认实现)；但是对于 `method3` 来说，我们只是在接口扩展中进行了定义，没有任何规定说它必须在最终的类型中被实现。在使用时，因为 `a2` 只是一个符合 `A2` 的实例，编译器对 `method2` 唯一能确定的只是在接口扩展中有一个默认实现，因此在调用时，因为无法确定安全，也就不会去进行动态派发，而是转而编译期间就确定的默认实现。

也许在这个例子中你会觉得无所谓，因为实际中估计并不会有人将一个已知类型实例转回接口类型。但是要考虑到如果你的一些泛型 **API** 中有类似的直接拿到一个接口类型的结果的时候，调用它的扩展方法时就需要特别小心了：一般来说，如果有这样的需求的话，我们可以考虑将这个接口类型再转回实际的类型，然后进行调用。

整理一下相关的规则的话：

- 如果类型推断得到的是实际类型
 - 那么类型中的实现将被调用；如果类型中没有实现，那么接口扩展中的默认实现被使用
- 如果类型推断得到的是接口
 - 并且方法在接口中进行了定义，那么类型中的实现将被调用；如果类型中没有实现，那么接口扩展中的默认实现被使用
 - 否则 (也就是方法没有在接口中定义)，扩展中的默认实现将被调用

where 和模式匹配

`where` 关键字在 Swift 中非常强大，但是往往容易被忽视。在这一节中，我们就来整理看看 `where` 有哪些使用场合吧。

首先是 `switch` 语句中，我们可以使用 `where` 来限定某些条件 `case`：

```
let name = ["王小二", "张三", "李四", "王二小"]

name.forEach {
    switch $0 {
    case let x where x.hasPrefix("王"):
        print("\(x)是笔者本家")
    default:
        print("你好, \($0)")
    }
}

// 输出:
// 王小二是笔者本家
// 你好, 张三
// 你好, 李四
// 王二小是笔者本家
```

这可以说是模式匹配的标准用法，对 `case` 条件进行限定可以让我们更灵活地使用 `switch` 语句。

在 `if let` 或者是 `for` 中我们也可以使用 `where` 来做类似的条件限定：

```
let num: [Int?] = [48, 99, nil]
num.forEach {
    if let score = $0 where score > 60 {
        print("及格啦 - \(score)")
    } else {
        print(":(")
    }
}

// 输出:
// :(
// 及格啦 - 99
// :(

for score in num where score > 60 {
    print("及格啦 - \(score)")
}

// 输出:
// 及格啦 - Optional(99)
```

这两种使用的方式都可以用额外的 `if` 来替代，这里只不过是让我们的代码更加易读了。也有一些场合是只有使用 `where` 才能准确表达的，比如我们在泛型中想要对方法的类型进行限定的时候。比如在标准库里对 `RawRepresentable` 接口定义 `!=` 运算符定义时：

```

/// Returns `true` iff `lhs.rawValue != rhs.rawValue`.
public func !=<T : RawRepresentable
    where T.RawValue : Equatable>(lhs: T, rhs: T) -> Bool

```

这里限定了 `T.RawValue` 必须要遵守 `Equatable` 接口，这样我们才能通过对比 `lhs` 和 `rhs` 的 `rawValue` 是否相等，进而判断这两个 `RawRepresentable` 值是否相等。如果没有 `where` 的保证的话，下面的代码就无法编译。同时，我们也限定了那些 `RawValue` 无法判等的 `RawRepresentable` 类型不能进行判等。

```

/// Returns `true` iff `lhs.rawValue != rhs.rawValue`.
public func !=<T : RawRepresentable
    where T.RawValue : Equatable>(lhs: T, rhs: T) -> Bool {
    return lhs.rawValue != rhs.rawValue
}

```

在 Swift 2.0 中，引入了 [protocol extension](#)。在有些时候，我们会希望一个接口扩展的默认实现只在某些特定的条件下适用，这时我们就可以用 `where` 关键字来进行限定。标准库中的接口扩展大量使用了这个技术来进行限定，比如 `SequenceType` 的 `sort` 方法就被定义在这样一个类型限制的接口扩展中：

```

extension SequenceType where Self.Generator.Element : Comparable {
    public func sort() -> [Self.Generator.Element]
}

```

很自然，如果 `SequenceType` (比如一个 `Array`) 中的元素是不可比较的，那么 `sort` 方法自然也就不能适用了：

```

let sortableArray: [Int] = [3,1,2,4,5]
let unsortableArray: [AnyObject?] = ["Hello", 4, nil]

sortableArray.sort()
// [1,2,3,4,5]

unsortableArray.sort()
// 无法编译
// note: expected an argument list of type
// '(@noescape (Self.Generator.Element, Self.Generator.Element) -> Bool)''
// 这意味着 Swift 尝试使用带有闭包的 `sort` 方法，并期望你输入一种排序方式

```

indirect 和嵌套 enum

在涉及到一些数据结构的经典理论和模型 (没错，就是链表，树和图) 时，我们往往会用到嵌套的类型。比如[链表](#)，在 Swift 中，我们可以这样来定义一个单向链表：

```
class Node<T> {
    let value: T?
    let next: Node<T>?

    init(value: T?, next: Node<T>?) {
        self.value = value
        self.next = next
    }
}

let list = Node(value: 1,
                next: Node(value: 2,
                            next: Node(value: 3,
                                        next: Node(value: 4, next: nil))))
// 单向链表: 1 -> 2 -> 3 -> 4
```

看起来还不错，但是这样的形式在表达空节点的时候并不十分理想。我们不得不借助于 `nil` 来表达空节点，但是事实上空节点和 `nil` 并不是等价的。另外，如果我们想表达一个空链表的话，要么需要把 `list` 设置为 `Optional`，要么把 `Node` 里的 `value` 以及 `next` 都设为 `nil`，这导致描述中存在歧义，我们不得不去做一些人为的约定来表述这样的情况，这在算法描述中是十分致命的。

在 Swift 2 中，我们现在可以使用嵌套的 `enum` 了 -- 而这在 Swift 1.x 里是编译器所不允许的。我们用 `enum` 来重新定义链表结构的话，会是下面这个样子：

```
indirect enum LinkedList<Element: Comparable> {
    case Empty
    case Node(Element, LinkedList<Element>)
}
let linkedList = LinkedList.Node(1, .Node(2, .Node(3, .Node(4, .Empty))))
// 单项链表: 1 -> 2 -> 3 -> 4
```

在 `enum` 的定义中嵌套自身对于 `class` 这样的引用类型来说没有任何问题，但是对于像 `struct` 或者 `enum` 这样的值类型来说，普通的做法是不可行的。我们需要在定义前面加上 `indirect` 来提示编译器不要直接在值类型中直接嵌套。用 `enum` 表达链表的好处在于，我们可以清晰地表示出空节点这一定义，这在像 Swift 这样类型十分严格的语言中是很有帮助的。比如我们可以寥寥数行就轻易地实现链表节点的删除方法，在 `enum` 中添加：

```
func linkedListByRemovingElement(element: Element)
    -> LinkedList<Element> {
    guard case let .Node(value, next) = self else {
        return .Empty
    }
    return value == element ?
```

```
        next : LinkedList.Node(value, next.linkedListByRemovingElement(element))
    }

    let result = linkedList.linkedListByRemovingElement(2)
    print(result)
    // 1 -> 3 -> 4
```

对于上面的算法的分析，就交给读者作为练习吧。:)

2. 从 Objective-C/C 到 Swift

Selector

`@selector` 是 Objective-C 时代的一个关键字，它可以将一个方法转换并赋值给一个 `SEL` 类型，它的表现很类似一个动态的函数指针。在 Objective-C 时 `selector` 非常常用，从设定 `target-action`，到自举询问是否响应某个方法，再到指定接受通知时需要调用的方法等等，都是由 `selector` 来负责的。在 Objective-C 里生成一个 `selector` 的方法一般是这个样子的：

```
-(void) callMe {
    //...
}

-(void) callMeWithParam:(id)obj {
    //...
}

SEL someMethod = @selector(callMe);
SEL anotherMethod = @selector(callMeWithParam:);

// 或者也可以使用 NSStringFromClass
// SEL someMethod = NSStringFromClass(@"callMe");
// SEL anotherMethod = NSStringFromClass(@"callMeWithParam:");
```

一般为了方便，很多人会选择使用 `@selector`，但是如果追求灵活的话，可能会更愿意使用 `NSStringFromClass` 的版本 -- 因为我们可以运行时动态生成字符串，从而通过方法的名字来调用到对应的方法。

在 Swift 中没有 `@selector` 了，我们要生成一个 `selector` 的话现在只能使用字符串。Swift 里对应原来 `SEL` 的类型是一个叫做 `Selector` 的结构体，它提供了一个接受字符串的初始化方法。像上面的两个例子在 Swift 中等效的写法是：

```
func callMe() {
    //...
}

func callMeWithParam(obj: AnyObject!) {
    //...
}

let someMethod = Selector("callMe")
let anotherMethod = Selector("callMeWithParam:")
```

和 Objective-C 时一样，记得在 `callMeWithParam` 后面加上冒号 (:)，这才是完整的方法名字。多个参数的方法名也和原来类似，是这个样子：

```
func turnByAngle(theAngle: Int, speed: Float) {
    //...
}
```



```
let method = Selector("turnByAngle:speed:")
```

另外，因为 `Selector` 类型实现了 `StringLiteralConvertible`，因此我们甚至可以不使用它的初始化方法，而直接用一个字符串进行赋值，就可以完成创建了。

最后需要注意的是，`selector` 其实是 Objective-C runtime 的概念，如果你的 `selector` 对应的方法只在 Swift 中可见的话 (也就是说它是一个 Swift 中的 `private` 方法)，在调用这个 `selector` 时你会遇到一个 `unrecognized selector` 错误：

这是错误代码

```
private func callMe() {  
    //...  
}  
NSTimer.scheduledTimerWithTimeInterval(1, target: self,  
    selector:"callMe", userInfo: nil, repeats: true)
```

正确的做法是在 `private` 前面加上 `@objc` 关键字，这样运行时就能找到对应的方法了。

```
@objc private func callMe() {  
    //...  
}  
  
NSTimer.scheduledTimerWithTimeInterval(1, target: self,  
    selector:"callMe", userInfo: nil, repeats: true)
```

另外，如果方法的第一个参数有外部变量的话，在通过字符串生成 `Selector` 时还有一个约定，那就是在方法名和第一个外部参数之间加上 `with`：

```
func aMethod(external paramName: AnyObject!) { ... }
```

想获取对应的 `Selector`，应该这么写：

```
let s = Selector("aMethodWithExternal:")
```

实例方法的动态调用

在 Swift 中有一类很有意思的写法，可以让我们不直接使用实例来调用这个实例上的方法，而是通过类型取出这个类型的某个实例方法的签名，然后再通过传递实例来拿到实际需要调用的方法。比如我们有这样的定义：

```
class MyClass {  
    func method(number: Int) -> Int {  
        return number + 1  
    }  
}
```

想要调用 `method` 方法的话，最普通的使用方式是生成 `MyClass` 的实例，然后用 `.method` 来调用它：

```
let object = MyClass()  
let result = object.method(1)  
  
// result = 2
```

这就限定了我们只能够在编译的时候就决定 `object` 实例和对应的方法调用。其实我们还可以使用刚才说到的方法，将上面的例子改写为：

```
let f = MyClass.method  
let object = MyClass()  
let result = f(object)(1)
```

这种语法看起来会比较奇怪，但是实际上并不复杂。Swift 中可以直接用 `Type.instanceMethod` 的语法来生成一个可以柯里化的方法。如果我们观察 `f` 的类型 (`Alt + 单击`)，可以知道它是：

```
f: MyClass -> (Int) -> Int
```

其实对于 `Type.instanceMethod` 这样的取值语句，实际上刚才

```
let f = MyClass.method
```

做的事情是类似于下面这样的字面量转换：

```
let f = { (obj: MyClass) in obj.method }
```

这下就不难理解为什么上面的调用方法可以成立了。

这种方法只适用于实例方法，对于属性的 `getter` 或者 `setter` 是不能用类似的写法的。另外，如果我们遇到有类型方法的名字冲突时：

```
class MyClass {  
    func method(number: Int) -> Int {  
        return number + 1  
    }  
  
    class func method(number: Int) -> Int {  
        return number  
    }  
}
```

如果不加改动，`MyClass.method` 将取到的是类型方法，如果我们想要取实例方法的话，可以显式地加上类型声明加以区别。这种方式不仅在这里有效，在其他大多数名字有歧义的情况下，都能很好地解决问题：

```
let f1 = MyClass.method  
// class func method 的版本  
  
let f2: Int -> Int = MyClass.method  
// 和 f1 相同  
  
let f3: MyClass -> Int -> Int = MyClass.method  
// func method 的柯里化版本
```

单例

在 Swift 1.2 后，我们可以使用类变量了，所以 Swift 中的单例也有了比较理想的创建方式，参见本节最后。为了说明和比较，这里保留了 Swift 1.2 之前的一些讨论，以供参考。

单例是一个在 Cocoa 中很常用的模式了。对于一些希望能在全局方便访问的实例，或者在 app 的生命周期中只应该存在一个的对象，我们一般都会使用单例来存储和访问。在 Objective-C 中单例的公认的写法类似下面这样：

```
@implementation MyManager
+ (id)sharedManager {
    static MyManager *staticInstance = nil;
    static dispatch_once_t onceToken;

    dispatch_once(&onceToken, ^{
        staticInstance = [[self alloc] init];
    });
    return staticInstance;
}
@end
```

使用 GCD 中的 `dispatch_once_t` 可以保证里面的代码只被调用一次，以此保证单例在线程上的安全。

因为在 Swift 中可以无缝直接使用 GCD，所以我们可以很方便地把类似方式的单例用 Swift 进行改写：

```
class MyManager {
    class var sharedManager : MyManager {
        struct Static {
            static var onceToken : dispatch_once_t = 0
            static var staticInstance : MyManager? = nil
        }

        dispatch_once(&Static.onceToken) {
            Static.staticInstance = MyManager()
        }

        return Static.staticInstance!
    }
}
```

因为 Swift 1.2 之前并不支持存储类型的类属性，所以我们需要使用一个 `struct` 来存储类型变量。

这样的写法当然没什么问题，但是在 Swift 里我们其实有一个更简单的保证线程安全的方式，那就是 `let`。把上面的写法简化一下，可以变成：

```

class MyManager {
    class var sharedManager : MyManager {
        struct Static {
            static let sharedInstance : MyManager = MyManager()
        }

        return Static.sharedInstance
    }
}

```

还有另一种更受大家欢迎，并被认为是 **Swift 1.2** 之前的最佳实践的做法。由于 **Swift 1.2** 之前 `class` 不支持存储式的 `property`，我们想要使用一个只存在一份的属性时，就只能将其定义在全局的 `scope` 中。值得庆幸的是，在 **Swift** 中是有访问级别的控制的，我们可以在变量定义前面加上 `private` 关键字，使这个变量只在当前文件中可以被访问。这样我们就可以写出一个没有嵌套的，语法上也更简单好看的单例了：

```

private let sharedInstance = MyManager()

class MyManager {
    class var sharedManager : MyManager {
        return sharedInstance
    }
}

```

Swift 1.2 中的改进

Swift 1.2 之前还不支持例如 `static let` 和 `static var` 这样的存储类变量。但是在 **1.2** 中 **Swift** 添加了类变量的支持，因此单例可以进一步简化。

将上面全局的 `sharedInstance` 拿到 `class` 中，这样结构上就更紧凑和合理了。

在 **Swift 1.2** 以及之后，如果没有特别的需求，我们推荐使用下面这样的方式来写一个单例：

```

class MyManager {
    private static let sharedInstance = MyManager()
    class var sharedManager : MyManager {
        return sharedInstance
    }
}

```

条件编译

在 C 系语言中，可以使用 `#if` 或者 `#ifdef` 之类的编译条件分支来控制哪些代码需要编译，而哪些代码不需要。Swift 中没有宏定义的概念，因此我们不能使用 `#ifdef` 的方法来检查某个符号是否经过宏定义。但是为了控制编译流程和内容，Swift 还是为我们提供了几种简单的机制来根据需求定制编译内容的。

首先是 `#if` 这一套编译标记还是存在的，使用的语法也和原来没有区别：

```
#if <condition>

#elseif <condition>

#else

#endif
```

当然，`#elseif` 和 `#else` 是可选的。

但是这几个表达式里的 `condition` 并不是任意的。Swift 内建了几种平台和架构的组合，来帮助我们为不同的平台编译不同的代码，具体地：

方法	可选参数
<code>os()</code>	OSX, iOS
<code>arch()</code>	x86_64, arm, arm64, i386

注意这些方法和参数都是大小写敏感的。举个例子，如果我们统一我们在 iOS 平台和 Mac 平台的关于颜色的 API 的话，一种可能的方法就是配合 `typealias` 进行条件编译：

```
#if os(OSX)
    typealias Color = NSColor
#else
    typealias Color = UIColor
#endif
```

另外对于 `arch()` 的参数需要说明的是 `arm` 和 `arm64` 两项分别对应 32 位 CPU 和 64 位 CPU 的真机情况，而对于模拟器，相应地 32 位设备的模拟器和 64 位设备的模拟器所对应的分别是 `i386` 和 `x86_64`，它们也是需要分开对待的。

另一种方式是对自定义的符号进行条件编译，比如我们需要使用同一个 `target` 完成同一个 app 的收费版和免费版两个版本，并且希望在点击某个按钮时收费版本执行功能，而免费版弹出提示的话，可以使用类似下面的方法：

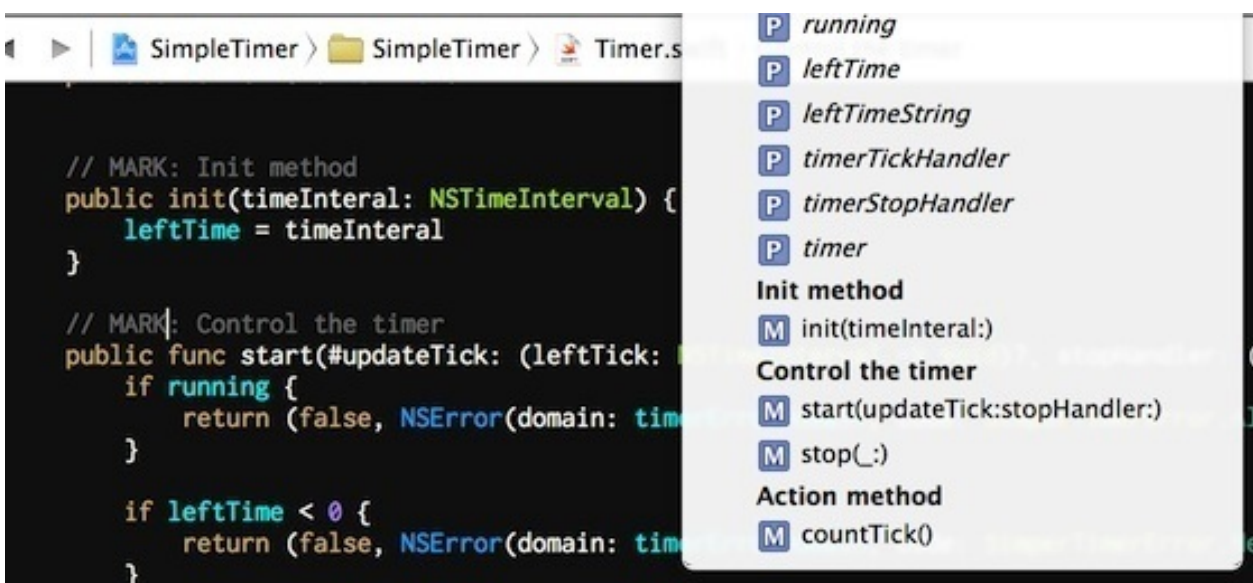
```
@IBAction func someButtonPressed(sender: AnyObject!) {  
    #if FREE_VERSION  
        // 弹出购买提示，导航至商店等  
    #else  
        // 实际功能  
    #endif  
}
```

在这里我们用 `FREE_VERSION` 这个编译符号来代表免费版本。为了使之有效，我们需要在项目的编译选项中进行设置，在项目的 **Build Settings** 中，找到 **Swift Compiler - Custom Flags**，并在其中的 **Other Swift Flags** 加上 `-D FREE_VERSION` 就可以了。

编译标记

在 Objective-C 中，我们经常在代码中插入 `#param` 符号来标记代码的区间，这样在 Xcode 的导航栏中我们就可以看到组织分块后的方法列表。这在单个文件方法较多时进行快速定位非常有用。

在 Swift 中也有类似的方式，我们可以在代码合适的地方添加 `// MARK:` 这样的标记 (注意大写)，并在后面接上名称，Xcode 将在代码中寻找这样的注释，然后以粗体标签的形式将名称显示在导航栏中。比如：



另外我们还可以在冒号的后面加一个横杠 `-`，这样在导航中会在这个位置再多显示一条横线，隔开各个部分，会显得更加清晰。

除了 `// MARK:` 以外，Xcode 还支持另外几种标记，它们分别是 `// TODO:` 和 `// FIXME:`。和 `MARK` 不同的是，另外两个标记在导航栏中不仅会显示后面跟着的名字或者说明，而且它们本身也会被显示出来，用来提示还未完成的工作或者需要修正的地方。这样在阅读源代码时首先看一看导航栏中的标记，就可以对当前文件有个大致的了解了。

以前在 Objective-C 中还有一个很常用的编译标记，那就是 `#warning`，一个 `#warning` 标记可以在 Xcode 的代码编辑器中显示为明显的黄色警告条，非常适合用来提示代码的维护者和使用者需要对某些东西加以关注。这个特性当前的 Swift 版本里还没有对应的方案。希望 Apple 能在接下来的版本中会加入一些类似的标记，像这个样子：

```
// WARNING: Add your API key here
```

很遗憾，暂时没有可以在编译时像 `#warning` 那样生成警告的方法了。

@UIApplicationMain

因为 Cocoa 开发环境已经在新建一个项目时帮助我们进行很多配置，这导致了刚接触 iOS 的开发者都存在基础比较薄弱的问题，其中一个最显著的现象就是很多人无法说清一个 app 启动的流程。程序到底是怎么开始的，AppDelegate 到底是什么，xib 或者 storyboard 是怎么被加载到屏幕上的？这一系列的问题虽然在开发中我们不会每次都去关心和自己配置，但是如果能进行一些了解的话对于程序各个部分的职责的明确会很有帮助。

在 C 系语言中，程序的入口都是 main 函数。对于一个 Objective-C 的 iOS app 项目，在新建项目时，Xcode 将帮我们准备好一个 main.m 文件，其中就有这个 main 函数：

```
int main(int argc, char * argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
                                 NSStringFromClass([AppDelegate class]));
    }
}
```

在这里我们调用了 UIKit 的 UIApplicationMain 方法。这个方法将根据第三个参数初始化一个 UIApplication 或其子类的对象并开始接收事件（在这个例子中传入 nil，意味使用默认的 UIApplication）。最后一个参数指定了 AppDelegate 类作为应用的委托，它被用来接收类似 didFinishLaunching 或者 didEnterBackground 这样的与应用生命周期相关的委托方法。另外，虽然这个方法标明为返回一个 int，但是其实它并不会真正返回。它会一直存在于内存中，直到用户或者系统将其强制终止。

了解了这些后，我们就可以来看看 Swift 的项目中对应的情况了。新建一个 Swift 的 iOS app 项目后，我们会发现所有文件中都没有一个像 Objective-C 时那样的 main 文件，也不存在 main 函数。唯一和 main 有关系的是在默认的 AppDelegate 类的声明上方有一个 @UIApplicationMain 的标签。

不说可能您已经猜到，这个标签做的事情就是将被标注的类作为委托，去创建一个 UIApplication 并启动整个程序。在编译的时候，编译器将寻找这个标记的类，并自动插入像 main 函数这样的模板代码。我们可以试试看把 @UIApplicationMain 去掉会怎么样：

Undefined symbols _main

说明找不到 main 函数了。

在一般情况下，我们并不需要对这个标签做任何修改，但是当我们如果想要使用 UIApplication 的子类而不是它本身的话，我们就需要对这部分内容“做点手脚”了。

刚才说到，其实 Swift 的 app 也是需要 main 函数的，只不过默认情况下是 @UIApplicationMain 帮助我们自动生成了而已。和 C 系语言的 main.c 或者 main.m 文件一样，Swift 项目也可以有一个

名为 `main.swift` 特殊的文件。在这个文件中，我们不需要定义作用域，而可以直接书写代码。这个文件中的代码将作为 `main` 函数来执行。比如我们在删除 `@UIApplicationMain` 后，在项目中添加一个 `main.swift` 文件，然后加上这样的代码：

```
UIApplicationMain(Process.argc, Process.unsafeArgv, nil,
    NSStringFromClass(AppDelegate))
```

现在编译运行，就不会再出现错误了。当然，我们还可以通过将第三个参数替换成自己的 `UIApplication` 子类，这样我们就可以轻易地做一些控制整个应用行为的事情了。比如将 `main.swift` 的内容换成：

```
import UIKit

class MyApplication: UIApplication {
    override func sendEvent(event: UIEvent!) {
        super.sendEvent(event)
        println("Event sent: \(event)");
    }
}

UIApplicationMain(Process.argc, Process.unsafeArgv,
    NSStringFromClass(MyApplication), NSStringFromClass(AppDelegate))
```

这样每次发送事件 (比如点击按钮) 时，我们都可以监听到这个事件了。

@objc 和 dynamic

虽然说 Swift 语言的初衷是希望能摆脱 Objective-C 的沉重的历史包袱和约束，但是不可否认的是经过了二十多年的洗礼，Cocoa 框架早就烙上了不可磨灭的 Objective-C 的印记。无数的第三方库是用 Objective-C 写成的，这些积累无论是谁都不能小觑。因此，在最初的版本中，Swift 不得不考虑与 Objective-C 的兼容。

Apple 采取的做法是允许我们在同一个项目中同时使用 Swift 和 Objective-C 来进行开发。其实一个项目中的 Objective-C 文件和 Swift 文件是处于两个不同世界中的，为了让它们能相互联通，我们需要添加一些桥梁。

首先通过添加 `{product-module-name}-Bridging-Header.h` 文件，并在其中填写想要使用的头文件名称，我们就可以很容易地在 Swift 中使用 Objective-C 代码了。Xcode 为了简化这个设定，甚至在 Swift 项目中第一次导入 Objective-C 文件时会主动弹框进行询问是否要自动创建这个文件，可以说是非常方便。

但是如果想要在 Objective-C 中使用 Swift 的类型的时候，事情就复杂一些。如果是来自外部的框架，那么这个框架与 Objective-C 项目肯定不是处在同一个 target 中的，我们需要对外部的 Swift module 进行导入。这个其实和使用 Objective-C 的原来的 Framework 是一样的，对于一个项目来说，外界框架是由 Swift 写的还是 Objective-C 写的，两者并没有太大区别。我们通过使用 2013 年新引入的 `@import` 来引入 module：

```
@import MySwiftKit;
```

之后就可以正常使用这个 Swift 写的框架了。

如果想要在 Objective-C 里使用的是同一个项目中的 Swift 的源文件的话，可以直接导入自动生成的头文件 `{product-module-name}-Swift.h` 来完成。比如项目的 target 叫做 MyApp 的话，我们就需要在 Objective-C 文件中写

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

但这只是故事的开始。Objective-C 和 Swift 在底层使用的是两套完全不同的机制，Cocoa 中的 Objective-C 对象是基于运行时的，它从骨子里遵循了 KVC (Key-Value Coding，通过类似字典的方式存储对象信息) 以及动态派发 (Dynamic Dispatch，在运行调用时再决定实际调用的具体实现)。而 Swift 为了追求性能，如果没有特殊需要的话，是不会在运行时再来决定这些的。也就是说，Swift 类型的成员或者方法在编译时就已经决定，而运行时便不再需要经过一次查找，而可以直接使用。

显而易见，这带来的问题是如果我们要使用 Objective-C 的代码或者特性来调用纯 Swift 的类型时

候，我们会因为找不到所需要的这些运行时信息，而导致失败。解决起来也很简单，在 **Swift** 类型文件中，我们可以将需要暴露给 **Objective-C** 使用的任何地方 (包括类，属性和方法等) 的声明前面加上 `@objc` 修饰符。注意这个步骤只需要对那些不是继承自 `NSObject` 的类型进行，如果你用 **Swift** 写的 `class` 是继承自 `NSObject` 的话，**Swift** 会默认自动为所有的非 `private` 的类和成员加上 `@objc`。这就是说，对于一个 `NSObject` 的子类，你只需要导入相应的头文件就可以在 **Objective-C** 里使用这个类了。

`@objc` 修饰符的另一个作用是为 **Objective-C** 侧重新声明方法或者变量的名字。虽然绝大部分时候自动转换的方法名已经足够好用 (比如会将 **Swift** 中类似 `init(name: String)` 的方法转换成 `-initWithName:(NSString *)name` 这样)，但是有时候我们还是期望 **Objective-C** 里使用和 **Swift** 中不一样的方法名或者类的名字，比如 **Swift** 里这样的一个类：

```
class 我的类: NSObject {
    func 打招呼(名字: String) {
        print("哈喽, \ \(名字)")
    }
}

我的类().打招呼("小明")
```

Objective-C 的话是无法使用中文来进行调用的，因此我们必须使用 `@objc` 将其转为 **ASCII** 才能在 **Objective-C** 里访问：

```
@objc(MyClass)
class 我的类 {
    @objc(greeting:)
    func 打招呼(名字: String) {
        print("哈喽, \ \(名字)")
    }
}
```

这样，我们在 **Objective-C** 里就能调用 `[[MyClass new] greeting:@"XiaoMing"]` 这样的代码了 (虽然比起原来一点都不好玩了)。另外，正如上面所说的以及在 [Selector](#) 一节中所提到的，即使是 `NSObject` 的子类，**Swift** 也不会在被标记为 `private` 的方法或成员上自动加 `@objc`，以保证尽量不使用动态派发来提高代码执行效率。如果我们确定使用这些内容的动态特性的话，我们需要手动给它们加上 `@objc` 修饰。

但是需要注意的是，添加 `@objc` 修饰符并不意味着这个方法或者属性会变成动态派发，**Swift** 依然可能会将其优化为静态调用。如果你需要和 **Objective-C** 里动态调用时相同的运行时特性的话，你需要使用的修饰符是 `dynamic`。一般情况下在做 **app** 开发时应该用不上，但是在施展一些像动态替换方法或者运行时再决定实现这样的“黑魔法”的时候，我们就需要用到 `dynamic` 修饰符了。在 [KVO](#) 一节中，我们提到了一个关于使用 `dynamic` 的实例。

关于 **Swift** 和 **Objective-C** 混用的一个好消息是，随着 **Swift** 的发展，**Apple** 正在努力改善 **SDK**。在 **Objective-C** 中添加的 `nonnull` 和 `nullable`，以及泛型的数组和字典等，其实上都是为了使 **SDK** 更加适合用 **Swift** 来使用所做的努力，我们还是很有希望在不久的将来能够摆脱掉这些妥协

和束缚的。

可选接口和接口扩展

Objective-C 中的 `protocol` 里存在 `@optional` 关键字，被这个关键字修饰的方法并非必须要被实现。我们可以通过接口定义一系列方法，然后由实现接口的类选择性地实现其中几个方法。在 Cocoa API 中很多情况下接口方法都是可选的，这点和 Swift 中的 `protocol` 的所有方法都必须被实现这一特性完全不同。

那些如果没有实现则接口就无法正常工作的一般是必须的，而相对地像作为事件通知或者对非关键属性进行配置的方法一般都是可选的。最好的例子我想应该是 `UITableViewDataSource` 和 `UITableViewDelegate`。前者中有两个必要方法：

```
-tableView:numberOfRowsInSection:
-tableView:cellForRowAtIndexPath:
```

分别用来计算和准备 `tableView` 的高度以及提供每一个 `cell` 的样式，而其他的像是返回 `section` 个数或者询问 `cell` 是否能被编辑的方法都有默认的行为，都是可选方法；后者（`UITableViewDelegate`）中的所有方法都是详细的配置和事件回传，因此全部都是可选的。

原生的 Swift `protocol` 里没有可选项，所有定义的方法都是必须实现的。如果我们想要像 Objective-C 里那样定义可选的接口方法，就需要将接口本身定义为 Objective-C 的，也即在 `protocol` 定义之前加上 `@objc`。另外和 Objective-C 中的 `@optional` 不同，我们使用没有 `@` 符号的关键字 `optional` 来定义可选方法：

```
@objc protocol OptionalProtocol {
    optional func optionalMethod()
}
```

另外，对于所有的声明，它们的前缀修饰是完全分开的。也就是说你不能像是在 Objective-C 里那样用一个 `@optional` 指定接下来的若干个方法都是可选的了，必须对每一个可选方法添加前缀，对于没有前缀的方法来说，它们是默认必须实现的：

```
@objc protocol OptionalProtocol {
    optional func optionalMethod() // 可选
    func necessaryMethod()        // 必须
    optional func anotherOptionalMethod() // 可选
}
```

一个不可避免的限制是，使用 `@objc` 修饰的 `protocol` 就只能被 `class` 实现了，也就是说，对于 `struct` 和 `enum` 类型，我们是无法令它们所实现的接口中含有可选方法或者属性的。另外，实现它的 `class` 中的方法还必须也被标注为 `@objc`，或者整个类就是继承自 `NSObject`。这对我们写代码来说是一种很让人郁闷的限制。

在 Swift 2.0 中，我们有了另一种选择，那就是使用 **protocol extension**。我们可以在声明一个 **protocol** 之后再用 **extension** 的方式给出部分方法默认的实现。这样这些方法在实际的类中就是可选实现的了。还是举上面的例子，使用接口扩展的话，会是这个样子：

```
protocol OptionalProtocol {
    func optionalMethod()           // 可选
    func necessaryMethod()         // 必须
    func anotherOptionalMethod()   // 可选
}

extension OptionalProtocol {
    func optionalMethod() {
        print("Implemented in extension")
    }

    func anotherOptionalMethod() {
        print("Implemented in extension")
    }
}

class MyClass: OptionalProtocol {
    func necessaryMethod() {
        print("Implemented in Class3")
    }

    func optionalMethod() {
        print("Implemented in Class3")
    }
}

let obj = MyClass()
obj.necessaryMethod() // Implemented in Class3
obj.optionalMethod()  // Implemented in Class3
obj.anotherOptionalMethod() // Implemented in extension
```

内存管理，weak 和 unowned

不管在什么语言里，内存管理的内容都很重要，所以我打算花上比其他 tip 长一些的篇幅仔细地说说这块内容。

Swift 是自动管理内存的，这也就是说，我们不再需要操心内存的申请和分配。当我们通过初始化创建一个对象时，Swift 会替我们管理和分配内存。而释放的原则遵循了自动引用计数 (ARC) 的规则：当一个对象没有引用的时候，其内存将会被自动回收。这套机制从很大程度上简化了我们的编码，我们只需要保证在合适的时候将引用置空 (比如超过作用域，或者手动设为 `nil` 等)，就可以确保内存使用不出现问题。

但是，所有的自动引用计数机制都有一个从理论上无法绕过的限制，那就是循环引用 (retain cycle) 的情况。

什么是循环引用

虽然我觉得循环引用这样的概念介绍不太应该出现在这本书中，但是为了更清晰地解释 Swift 中的循环引用的一般情况，这里还是简单进行说明。假设我们有两个类 A 和 B，它们之中分别有一个存储属性持有对方：

```
class A {
    let b: B
    init() {
        b = B()
        b.a = self
    }

    deinit {
        print("A deinit")
    }
}

class B {
    var a: A? = nil
    deinit {
        print("B deinit")
    }
}
```

在 A 的初始化方法中，我们生成了一个 B 的实例并将其存储在属性中。然后我们又将 A 的实例赋值给了 `b.a`。这样 `a.b` 和 `b.a` 将在初始化的时候形成一个引用循环。现在当有第三方的调用初始化了 A，然后即使立即将其释放，A 和 B 两个类实例的 `deinit` 方法也不会被调用，说明它们并没有被释放。

```
var obj: A? = A()
obj = nil
// 内存没有释放
```


因为即使 `obj` 不再持有 `A` 的这个对象，`b` 中的 `b.a` 依然引用着这个对象，导致它无法释放。而进一步，`a` 中也持有着 `b`，导致 `b` 也无法释放。在将 `obj` 设为 `nil` 之后，我们在代码里再也拿不到对于这个对象的引用了，所以除非是杀掉整个进程，我们已经永远也无法将它释放了。多么悲伤的故事啊..

在 Swift 里防止循环引用

为了防止这种人神共愤的悲剧的发生，我们必须给编译器一点提示，表明我们不希望它们互相持有。一般来说我们习惯希望 "被动" 的一方不要去持有 "主动" 的一方。在这里 `b.a` 里对 `A` 的实例的持有是由 `A` 的方法设定的，我们在之后直接使用的也是 `A` 的实例，因此认为 `b` 是被动的一方。可以将上面的 `class B` 的声明改为：

```
class B {  
    weak var a: A? = nil  
    deinit {  
        print("B deinit")  
    }  
}
```

在 `var a` 前面加上了 `weak`，向编译器说明我们不喜欢持有 `a`。这时，当 `obj` 指向 `nil` 时，整个环境中就没有对 `A` 的这个实例的持有了，于是这个实例可以得到释放。接着，这个被释放的实例上对 `b` 的引用 `a.b` 也随着这次释放结束了作用域，所以 `b` 的引用也将归零，得到释放。添加 `weak` 后的输出：

```
A deinit  
B deinit
```

可能有心的朋友已经注意到，在 Swift 中除了 `weak` 以外，还有另一个冲着编译器叫喊着类似的 "不要引用我" 的标识符，那就是 `unowned`。它们的区别在哪里呢？如果您是一直写 Objective-C 过来的，那么从表面的行为上来说 `unowned` 更像以前的 `unsafe_unretained`，而 `weak` 就是以前的 `weak`。用通俗的话说，就是 `unowned` 设置以后即使它原来引用的内容已经被释放了，它仍然会保持对被已经释放了的对象的一个 "无效的" 引用，它不能是 `Optional` 值，也不会被指向 `nil`。如果你尝试调用这个引用的方法或者访问成员属性的话，程序就会崩溃。而 `weak` 则友好一些，在引用的内容被释放后，标记为 `weak` 的成员将会自动地变成 `nil` (因此被标记为 `@weak` 的变量一定需要是 `Optional` 值)。关于两者使用的选择，Apple 给我们的建议是如果能够确定在访问时不会被释放的话，尽量使用 `unowned`，如果存在被释放的可能，那就选择用 `weak`。

我们结合实际编码中的使用来看看选择吧。日常工作中一般使用弱引用的最常见的场景有两个：

1. 设置 `delegate` 时
2. 在 `self` 属性存储为闭包时，其中拥有对 `self` 引用时

前者是 Cocoa 框架的常见设计模式，比如我们有一个负责网络请求的类，它实现了发送请求以及

接收请求结果的任务，其中这个结果是通过实现请求类的 `protocol` 的方式来实现的，这种时候我们一般设置 `delegate` 为 `weak`：

```
// RequestManager.swift
class RequestManager: RequestHandler {

    @objc func requestFinished() {
        print("请求完成")
    }

    func sendRequest() {
        let req = Request()
        req.delegate = self

        req.send()
    }
}

// Request.swift
@objc protocol RequestHandler {
    optional func requestFinished()
}

class Request {
    weak var delegate: RequestHandler!;

    func send() {
        // 发送请求
        // 一般来说会将 req 的引用传递给网络框架
    }

    func gotResponse() {
        // 请求返回
        delegate?.requestFinished?()
    }
}
```

`req` 中以 `weak` 的方式持有了 `delegate`，因为网络请求是一个异步过程，很可能会遇到用户不愿意等待而选择放弃的情况。这种情况下一般都会将 `RequestManager` 进行清理，所以我们其实是无法保证在拿到返回时作为 `delegate` 的 `RequestManager` 对象是一定存在的。因此我们使用了 `weak` 而非 `owned`，并在调用前进行了判断。

闭包和循环引用

另一种闭包的情况稍微复杂一些：我们首先要知道，闭包中对任何其他元素的引用都是会被闭包自动持有的。如果我们在闭包中写了 `self` 这样的东西的话，那我们其实也就在闭包内持有了当前的对象。这里就出现了一个在实际开发中比较隐蔽的陷阱：如果当前的实例直接或者间接地对这个闭包又有引用的话，就形成了一个 `self -> 闭包 -> self` 的循环引用。最简单的例子是，我们声明了一个闭包用来以特定的形式打印 `self` 中的一个字符串：

```
class Person {
    let name: String
    lazy var printName: ()->() = {
        print("The name is \(self.name)")
    }
}
```

```

    init(personName: String) {
        name = personName
    }

    deinit {
        print("Person deinit \(self.name)")
    }
}

var xiaoMing: Person? = Person(personName: "XiaoMing")
xiaoMing!.printName()
xiaoMing = nil
// 输出:
// The name is XiaoMing, 没有被释放

```

`printName` 是 `self` 的属性，会被 `self` 持有，而它本身又在闭包内持有 `self`，这导致了 `xiaoMing` 的 `deinit` 在自身超过作用域后还是没有被调用，也就是没有被释放。为了解决这种闭包内的循环引用，我们需要在闭包开始的时候添加一个标注，来表示这个闭包内的某些要素应该以何种特定的方式来使用。可以将 `printName` 修改为这样：

```

lazy var printName: ()->() = {
    [weak self] in
    if let strongSelf = self {
        print("The name is \(strongSelf.name)")
    }
}

```

现在内存释放就正确了：

```

// 输出:
// The name is XiaoMing
// Person deinit XiaoMing

```

如果我们可以确定在整个过程中 `self` 不会被释放的话，我们可以将上面的 `weak` 改为 `unowned`，这样就不再需要 `strongSelf` 的判断。但是如果在过程中 `self` 被释放了而 `printName` 这个闭包没有被释放的话（比如生成 `Person` 后，某个外部变量持有了 `printName`，随后这个 `Person` 对象被释放了，但是 `printName` 已然存在并可能被调用），使用 `unowned` 将造成崩溃。在这里我们需要根据实际需求来决定是使用 `weak` 还是 `unowned`。

这种在闭包参数的位置进行标注的语法结构是将要标注的内容放在原来参数的前面，并使用中括号括起来。如果有多个需要标注的元素的话，在同一个中括号内用逗号隔开，举个例子：

```

// 标注前
{ (number: Int) -> Bool in
    //...
    return true
}

// 标注后
{ [unowned self, weak someObject] (number: Int) -> Bool in

```

```
//...  
return true  
}
```

@autoreleasepool

Swift 在内存管理上使用自动引用计数 (ARC) 的一套方法，在 ARC 中虽然不需要手动地调用像是 `retain`，`release` 或者是 `autorelease` 这样的方法来管理引用计数，但是这些方法还是都会被调用的 -- 只不过是编译器在编译时在合适的地方帮我们加入了而已。其中 `retain` 和 `release` 都很直接，就是将对象的引用计数加一或者减一。但是 `autorelease` 就比较特殊一些，它会将接受该消息的对象放到一个预先建立的自动释放池 (auto release pool) 中，并在自动释放池收到 `drain` 消息时将这些对象的引用计数减一，然后将它们从池子中移除 (这一过程形象地称为“抽干池子”)。

在 app 中，整个主线程其实是跑在一个自动释放池里的，并且在每个主 Runloop 结束时进行 `drain` 操作。这是一种必要的延迟释放的方式，因为我们有时候需要确保在方法内部初始化的生成的对象在被返回后别人还能使用，而不是立即被释放掉。

在 Objective-C 中，建立一个自动释放池的语法很简单，使用 `@autoreleasepool` 就行了。如果你新建一个 Objective-C 项目，可以看到 `main.m` 中就有我们刚才说到的整个项目的 `autoreleasepool`：

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        int retVal = UIApplicationMain(
            argc,
            argv,
            nil,
            NSStringFromClass([AppDelegate class]));
        return retVal;
    }
}
```

更进一步，其实 `@autoreleasepool` 在编译时会被展开为 `NSAutoreleasePool`，并附带 `drain` 方法的调用。

而在 Swift 项目中，因为有了 `@UIApplicationMain`，我们不再需要 `main` 文件和 `main` 函数，所以原来的整个程序的自动释放池就不存在了。即使我们使用 `main.swift` 来作为程序的入口时，也是不需要自己再添加自动释放池的。

但是在一种情况下我们还是希望自动释放，那就是在面对在一个方法作用域中要生成大量的 `autorelease` 对象的时候。在 Swift 1.0 时，我们可以写这样的代码：

```
func loadBigData() {
    if let path = NSBundle.mainBundle()
        .pathForResource("big", ofType: "jpg") {

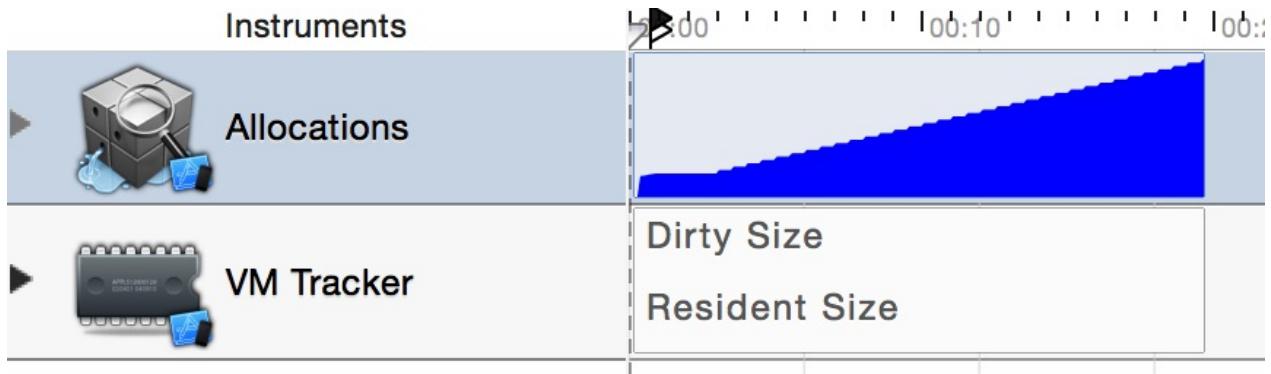
        for i in 1...10000 {
            let data = NSData.dataWithContentsOfFile(
                path, options: nil, error: nil)
        }
    }
}
```

```

        NSThread.sleepForTimeInterval(0.5)
    }
}
}

```

`dataWithContentsOfFile` 返回的是 `autorelease` 的对象，因为我们一直处在循环中，因此它们将一直没有机会被释放。如果数量太多而且数据太大的时候，很容易因为内存不足而崩溃。在 Instruments 下可以看到内存 alloc 的情况：



这显然是一幅很不妙的情景。在面对这种情况的时候，正确的处理方法是在其中加入一个自动释放池，这样我们就可以在循环进行到某个特定的时候施放内存，保证不会因为内存不足而导致应用崩溃。在 Swift 中我们也是能使用 `autoreleasepool` 的 -- 虽然语法上略有不同。相比于原来在 Objective-C 中的关键字，现在它变成了一个接受闭包的方法：

```

func autoreleasepool(code: () -> ())

```

利用尾随闭包的写法，很容易就能在 Swift 中加入一个类似的自动释放池了：

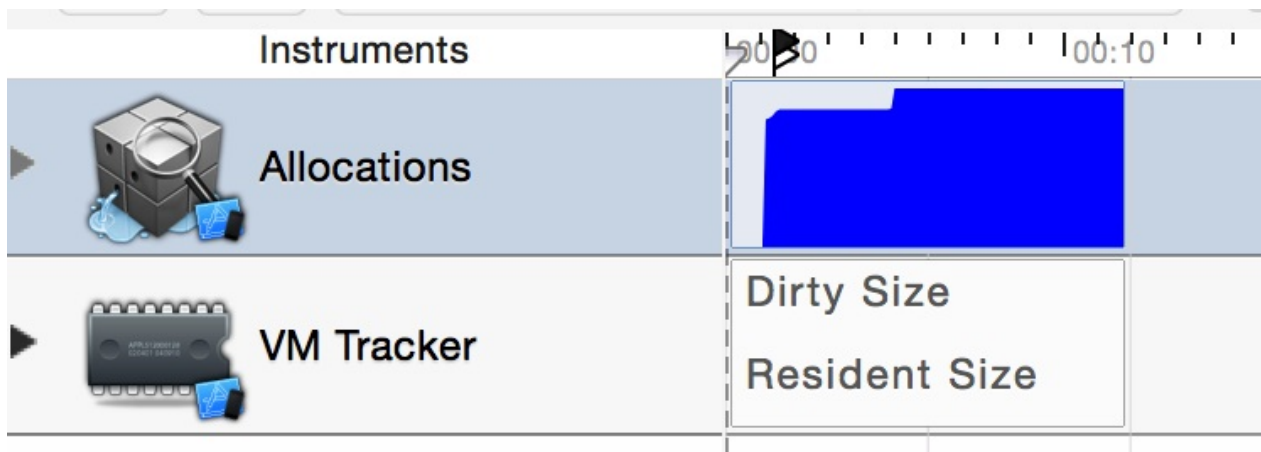
```

func loadBigData() {
    if let path = NSBundle.mainBundle()
        .pathForResource("big", ofType: "jpg") {
        for i in 1...10000 {
            autoreleasepool {
                let data = NSData.dataWithContentsOfFile(
                    path, options: nil, error: nil)

                NSThread.sleepForTimeInterval(0.5)
            }
        }
    }
}

```

这样改动以后，内存分配就没有什么忧虑了：



这里我们每一次循环都生成了一个自动释放池，虽然可以保证内存使用达到最小，但是释放过于频繁也会带来潜在的性能忧虑。一个折衷的方法是将循环分隔开加入自动释放池，比如每 10 次循环对应一次自动释放，这样能减少带来的性能损失。

其实对于这个特定的例子，我们并不一定需要加入自动释放。在 Swift 中更提倡的是用初始化方法而不是用像上面那样的类方法来生成对象，而且从 Swift 1.1 开始，因为加入了可以返回 `nil` 的初始化方法，像上面例子中那样的工厂方法都已经从 API 中删除了。今后我们都应该这样写：

```
let data = NSData(contentsOfFile: path)
```

使用初始化方法的话，我们就不需要面临自动释放的问题了，每次在超过作用域后，自动内存管理都将为我们处理好内存相关的事情。

值类型和引用类型

Swift 的类型分为值类型和引用类型两种，值类型在传递和赋值时将进行复制，而引用类型则只会使用引用对象的一个 "指向"。Swift 中的 `struct` 和 `enum` 定义的类型是值类型，使用 `class` 定义的为引用类型。很有意思的是，Swift 中的所有的内建类型都是值类型，不仅包括了传统意义像 `Int`，`Bool` 这些，甚至连 `String`，`Array` 以及 `Dictionary` 都是值类型的。这在程序设计上绝对算得上一个震撼的改动，因为据我所知现在流行的编程语言中，像数组和字典这样的类型，几乎清一色都是引用类型。

那么使用值类型有什么好处呢？相较于传统的引用类型来说，一个很显而易见的优势就是减少了堆上内存分配和回收的次数。首先我们需要知道，Swift 的值类型，特别是数组和字典这样的容器，在内存管理上经过了精心的设计。值类型的一个特点是在传递和赋值时进行复制，每次复制肯定会产生额外开销，但是在 Swift 中这个消耗被控制在了最小范围内，在没有必要复制的时候，值类型的复制都是不会发生的。也就是说，简单的赋值，参数的传递等等普通操作，虽然我们可能用不同的名字来回设置和传递值类型，但是在内存上它们都是同一块内容。比如下面这样的代码：

```
func test(arr: [Int]) {
    for i in arr {
        print(i)
    }
}

var a = [1,2,3]
var b = a
let c = b
test(a)
```

这么折腾一圈下来，其实我们只在第一句 `a` 初始化赋值时发生了内存分配，而之后的 `b`，`c` 甚至传递到 `test` 方法内的 `arr`，和最开始的 `a` 在物理内存上都是同一个东西。而且这个 `a` 还只在栈空间上，于是这个过程对于数组来说，只发生了指针移动，而完全没有堆内存的分配和释放的问题，这样的运行效率可以说极高。

值类型被复制的时机是值类型的内容发生改变时，比如下面在 `b` 中又加入了一个数，此时值复制就是必须的了：

```
var a = [1,2,3]
var b = a
b.append(5)
// 此时 a 和 b 的内存地址不再相同
```

值类型在复制时，会将存储其中的值类型一并进行复制，而对于其中的引用类型的话，则只复制一份引用。这是合理的行为，因为我们不会希望引用类型莫名其妙地引用到了我们设定以外其他对象：


```

class MyObject {
    var num = 0
}

var myObject = MyObject()
var a = [myObject]
var b = a

b.append(myObject)

myObject.num = 100
println(b[0].num)    //100
println(b[1].num)    //100

// myObject 的改动同时影响了 b[0] 和 b[1]

```

虽然将数组和字典设计为值类型最大的考虑是为了线程安全，但是这样的设计在存储的元素或条目数量较少时，给我们带来了另一个优点，那就是非常高效，因为 "一旦赋值就不太会变化" 这种使用情景在 Cocoa 框架中是占有绝大多数的，这有效减少了内存的分配和回收。但是在少数情况下，我们显然也可能在数组或者字典中存储非常多的东西，并且还要对其中的内容进行添加或者删除。在这时，Swift 内建的值类型的容器类型在每次操作时都需要复制一遍，即使是存储的都是引用类型，在复制时我们还是需要存储大量的引用，这个开销就变得不容忽视了。幸好我们还有 Cocoa 中的引用类型的容器类来对应这种情况，那就是 `NSMutableArray` 和 `NSMutableDictionary`。

所以，在使用数组合字典时的最佳实践应该是，按照具体的数据规模和操作特点来决定到时是使用值类型的容器还是引用类型的容器：在需要处理大量数据并且频繁操作 (增减) 其中元素时，选择 `NSMutableArray` 和 `NSMutableDictionary` 会更好，而对于容器内条目小而容器本身数目多的情况，应该使用 Swift 语言内建的 `Array` 和 `Dictionary`。

String 还是 NSString

既然像 `String` 这样的 Swift 的类型和 Foundation 的对应的类是可以无缝转换的，那么我们在使用和选择的时候，有没有什么需要特别注意的呢？

简单来说，没有特别需要注意的，但是尽可能的话还是使用原生的 `String` 类型。

原因有三。

首先虽然 `String` 和 `NSString` 有着良好的互相转换的特性，但是现在 Cocoa 所有的 API 都接受和返回 `String` 类型。我们没有必要也不必给自己凭空添加麻烦去把框架中返回的字符串做一遍转换，既然 Cocoa 鼓励使用 `String`，并且为我们提供了足够的操作 `String` 的方法，那为什么不直接使用呢？

其次，因为在 Swift 中 `String` 是 `struct`，相比起 `NSObject` 的 `NSString` 类来说，更切合字符串的“不变”这一特性。通过配合常量赋值 (`let`)，这种不变性在多线程编程时就非常重要了，它从原理上将程序员从内存访问和操作顺序的担忧中解放出来。另外，在不触及 `NSString` 特有操作和动态特性的时候，使用 `String` 的方法，在性能上也会有所提升。

最后，因为 `String` 里的 `String.CharacterView` 实现了像 `CollectionType` 这样的接口，因此有些 Swift 的语法特性只有 `String` 才能使用，而 `NSString` 是没有的。一个典型就是 `for...in` 的枚举，我们可以写：

```
let levels = "ABCDE"
for i in levels.characters {
    print(i)
}

// 输出：
// ABCDE
```

而如果转换为 `NSString` 的话，是无法使用 `characters` 并且进行枚举的。

不过也有例外的情况。有一些 `NSString` 的方法在 `String` 中并没有实现，一个很有用的就是在 iOS 8 中新加的 `containsString`。我们想使用这个 API 来简单地确定某个字符串包括一个子字符串时，只能先将其转为 `NSString`：

```
if (levels as NSString).containsString("BC") {
    println("包含字符串")
}

// 输出：
// 包含字符串
```

Swift 的 `String` 没有 `containsString` 是一件很奇怪的事情，理论上应该不存在实现的难度，希望只是 Apple 一时忘了这个新加的 API 吧。当然你也可以自行用扩展的方式在自己的代码库为 `String` 添加这个方法。当然，还有一些其他的像 `length` 和 `characterAtIndex`：这样的 API 也没有 `String` 的版本，这主要是因为 `String` 和 `NSString` 在处理编码上的差异导致的。

使用 `String` 唯一的一个比较麻烦的地方在于它和 `Range` 的配合。在 `NSString` 中，我们在匹配字符串的时候通常使用 `NSRange` 来表征结果或者作为输入。而在使用 `String` 的对应的 API 时，`NSRange` 也会被映射成它在 Swift 中且对应 `String` 的特殊版本：`Range<String.Index>`。这有时候会让人非常讨厌：

```
let levels = "ABCDE"

let nsRange = NSRange(1, 4)
// 编译错误
// Cannot invoke `stringByReplacingCharactersInRange`
// with an argument list of type '(NSRange, withString: String)'
levels.stringByReplacingCharactersInRange(nsRange, withString: "AAAA")

let indexPositionOne = levels.startIndex.successor()
let swiftRange = indexPositionOne..advance(indexPositionOne, 4)
levels.stringByReplacingCharactersInRange(swiftRange, withString: "AAAA")
// 输出:
// AAAAA
```

一般来说，我们可能更愿意和基于 `Int` 的 `NSRange` 一起工作，而不喜欢使用麻烦的 `Range<String.Index>`。这种情况下，将 `String` 转为 `NSString` 也许是个不错的选择：

```
let nsRange = NSRange(1, 4)
(levels as NSString).stringByReplacingCharactersInRange(
    nsRange, withString: "AAAA")
```

UnsafePointer

Swift 本身从设计上来说是一门非常安全的语言，在 Swift 的思想中，所有的引用或者变量的类型都是确定并且正确对应它们的实际类型的，你应当无法进行任意的类型转换，也不能直接通过指针做出一些出格的事情。这种安全性在日常的程序开发中对于避免不必要的 bug，以及迅速而且稳定地找出代码错误是非常有帮助的。但是凡事都有两面性，在高安全的同时，Swift 也相应地丧失了部分的灵活性。

现阶段想要完全抛弃 C 的一套东西还是相当困难的，特别是在很多上古级别的 C API 框架还在使用 (或者被间接使用)。开发者，尤其是偏向较底层的框架的开发者不得不面临着与 C API 打交道的时候，还是无法绕开指针的概念，而指针在 Swift 中其实并不被鼓励，语言标准中也是完全没有与指针完全等同的概念的。为了与庞大的 C 系帝国进行合作，Swift 定义了一套对 C 语言指针的访问和转换方法，那就是 `UnsafePointer` 和它的一系列变体。对于使用 C API 时如果遇到接受内存地址作为参数，或者返回是内存地址的情况，在 Swift 里会将它们转为 `UnsafePointer<Type>` 的类型，比如说如果某个 API 在 C 中是这样的话：

```
void method(const int *num) {
    printf("%d", *num);
}
```

其对应的 Swift 方法应该是：

```
func method(num: UnsafePointer<CInt>) {
    print(num.memory)
}
```

我们这个 tip 所说的 `UnsafePointer`，就是 Swift 中专门针对指针的转换。对于其他的 C 中基础类型，在 Swift 中对应的类型都遵循统一的命名规则：在前面加上一个字母 `c` 并将原来的第一个字母大写：比如 `int`，`bool` 和 `char` 的对应类型分别是 `CInt`，`CBool` 和 `CChar`。在上面的 C 方法中，我们接受一个 `int` 的指针，转换到 Swift 里所对应的就是一个 `CInt` 的 `UnsafePointer` 类型。这里原来的 C API 中已经指明了输入的 `num` 指针的不可变的 (`const`)，因此在 Swift 中我们与之对应的是 `UnsafePointer` 这个不可变版本。如果只是一个普通的可变指针的话，我们可以使用 `UnsafeMutablePointer` 来对应：

C API	Swift API
<code>const Type *</code>	<code>UnsafePointer</code>
<code>Type *</code>	<code>UnsafeMutablePointer</code>

在 C 中，对某个指针进行取值使用的是 `*`，而在 Swift 中我们可以使用 `memory` 属性来读取相应内存中存储的内容。通过传入指针地址进行方法调用的时候就都比较相似了，都是在前面加上 `&`

符号，C 的版本和 Swift 的版本只在申明变量的时候有所区别：

```
// C
int a = 123;
method(&a);    // 输出 123

// Swift
var a: CInt = 123
method(&a)     // 输出 123
```

遵守这些原则，使用 `UnsafePointer` 在 Swift 中进行 C API 的调用应该就不会有很大问题了。

另外一个重要的课题是如何在指针的内容和实际的值之间进行转换。比如我们如果由于某种原因需要涉及到直接使用 `CFArray` 的方法来获取数组中元素的时候，我们会用到这个方法：

```
func CFArrayGetValueAtIndex(theArray: CFArray!, idx: CFIndex)
    -> UnsafePointer<Void>
```

因为 `CFArray` 中是可以存放任意对象的，所以这里的返回是一个任意对象的指针，相当于 C 中的 `void *`。这显然不是我们想要的东西。Swift 中为我们提供了一个强制转换的方法 `unsafeBitCast`，通过下面的代码，我们可以看到应当如何使用类似这样的 API，将一个指针强制按位转成所需类型的对象：

```
let arr = NSArray(object: "meow")
let str = unsafeBitCast(CFArrayGetValueAtIndex(arr, 0), CFString.self)
// str = "meow"
```

`unsafeBitCast` 会将第一个参数的内容按照第二个参数的类型进行转换，而不去关心实际是不是可行，这也正是 `UnsafePointer` 的不安全所在，因为我们不必遵守类型转换的检查，而拥有了在指针层面直接操作内存的机会。

其实说了这么多，Apple 将直接的指针访问冠以 `Unsafe` 的前缀，就是提醒我们：这些东西不安全，亲们能不用就别用了吧（作为 Apple，另一个重要的考虑是如果避免指针的话可以减少很多系统漏洞）！在日常开发中，我们确实不太需要经常和这些东西打交道（除了传入 `NSError` 指针这个历史遗留问题以外，而且在 Swift 2.0 中也已经使用异常机制替代了 `NSError`）。总之，尽可能地在高抽象层级编写代码，会是高效和正确的有力保证。无数先辈已经用血淋淋的教训告诉我们，要避免去做这样的不安全的操作，除非你确实知道你在做的是什。

C 指针内存管理

C 指针在 Swift 中被冠名以 `unsafe` 的另一个原因是无法对其进行自动的内存管理。和 `Unsafe` 类的指针工作的时候，我们需要像 ARC 时代之前那样手动地来申请和释放内存，以保证程序不会出现泄露或是因为访问已释放内存而造成崩溃。

我们如果想要声明，初始化，然后使用一个指针的话，完整的做法是使用 `alloc` 和 `initialize` 来创建。如果一不小心，就很容易写成下面这样：

这是错误代码

```
class MyClass {
    var a = 1
    deinit {
        print("deinit")
    }
}

var pointer: UnsafeMutablePointer<MyClass>!

pointer = UnsafeMutablePointer<MyClass>.alloc(1)
pointer.initialize(MyClass())

println(pointer.memory.a) // 1

pointer = nil
```

虽然我们最后将 `pointer` 值为 `nil`，但是由于 `UnsafeMutablePointer` 并不会自动进行内存管理，因此其实 `pointer` 所指向的内存是没有被释放和回收的（这可以从 `MyClass` 的 `deinit` 没有被调用来加以证实；这造成了内存泄露。正确的做法是为 `pointer` 加入 `destroy` 和 `dealloc`，它们分别会释放指针指向的内存的对象以及指针自己本身：

```
var pointer: UnsafeMutablePointer<MyClass>!

pointer = UnsafeMutablePointer<MyClass>.alloc(1)
pointer.initialize(MyClass())

println(pointer.memory.a)
pointer.destroy()
pointer.dealloc(1)
pointer = nil

// 输出：
// 1
// deinit
```

如果我们在 `dealloc` 之后再去访问 `pointer` 或者再次调用 `dealloc` 的话，迎接我们的自然是崩溃。这并不出意料之外，相信有过手动管理经验的读者都会对这种场景非常熟悉了。

在手动处理这类指针的内存管理时，我们需要遵循的一个基本原则就是谁创建谁释放。`destroy` 与 `dealloc` 应该要与 `alloc` 成对出现，如果不是你创建的指针，那么一般来说你就不需要去释放它。一种最常见的例子就是如果我们是通过调用了某个方法得到的指针，那么除非文档或者负责这个方法的开发者明确告诉你应该由使用者进行释放，否则都不应该去试图管理它的内存状态：

```
var x:UnsafeMutablePointer<tm>!  
var t = time_t()  
time(&t)  
x = localtime(&t)  
x = nil
```

最后，虽然在本节的例子中使用的都是 `alloc` 和 `dealloc` 的情况，但是指针的内存申请也可以使用 `malloc` 或者 `calloc` 来完成，这种情况下在释放时我们需要对应使用 `free` 而不是 `dealloc`。

大概就这么多，祝你好运！

COpaquePointer 和 C convention

在 C 中有一类指针，你在头文件中无法找到具体的定义，只能拿到类型的名字，而所有的实现细节都是隐藏的。这类指针在 C 或 C++ 中被叫做不透明指针 (Opaque Pointer)，顾名思义，它的实现和表意对使用者来说是不透明的。

我们在这里不想过多讨论 C 中不透明指针的应用场景和特性，毕竟这是一本关于 Swift 的书。在 Swift 中对应这类不透明指针的类型是 `COpaquePointer`，它用来表示那些在 Swift 中无法进行类型描述的 C 指针。那些能够确定类型的指针所表示的是其指向的内存是可以用某个 Swift 中的类型来描述的，因此都使用更准确的 `UnsafePointer<T>` 来存储。而对于另外那些 Swift 无法表述的指针，就统一写为 `COpaquePointer`，以作补充。

在 Swift 早期 beta 的时候，曾经有不少 API 返回或者接受的是 `COpaquePointer` 类型。但是随着 Swift 的逐渐完善，大部分涉及到指针的 API 里的 `COpaquePointer` 都被正确地归类到了合适的 `Unsafe` 指针中，因此现在在开发中可能很少能再看到 `COpaquePointer` 了。最多的使用场景可能就是 `COpaquePointer` 和某个特定的 `Unsafe` 之间的转换了，我们可以分别使用这两个类型的初始化方法将一个指针转换从某个类型强制地转为另一个类型：

```
public struct UnsafeMutablePointer<Memory> :  
    Equatable, Hashable ... {  
  
    //..  
  
    init(_ other: COpaquePointer)  
  
    //..  
}  
  
public struct COpaquePointer:  
    Equatable,  
    Hashable,  
    NilLiteralConvertible {  
  
    //..  
  
    init<T>(_ source: UnsafePointer<T>)  
  
    //..  
}
```

`COpaquePointer` 在 Swift 中扮演的是指针转换的中间人的角色，我们可以通过这个类型在不同指针类型间进行转换。当然了，这些转换都是不安全的，除非你知道自己在干什么，以及有十足的把握，否则不要这么做！

另外一种重要的指针形式是指向函数的指针，在 C 中这种情况也并不少见，即一块存储了某个函数实际所在的位置的内存空间。从 Swift 2.0 开始，与这类指针可以被转化为闭包，不过和其他普通闭包不同，我们需要为它添加上 `@convention` 标注。

举个例子，如果我们在 C 中有这样一个函数：


```
int cFunction(int (callback)(int x, int y)) {
    return callback(1, 2);
}
```

这个函数接受一个 `callback`，这个 `callback` 有两个 `int` 类型的参数，`cFunction` 本身返回这个 `callback` 的结果。如果我们想在 Swift 中使用这个 C 函数的话，应该这样写：

```
let callback: @convention(c) (Int32, Int32) -> Int32 = {
    (x, y) -> Int32 in
    return x + y
}

let result = cFunction(callback)
print(result)
// 输出:
// 3
```

在没有歧义的情况下，我们甚至可以省掉这个标注，而直接将它以一个 Swift 闭包的形式传递给 C：

```
let result = cFunction {
    (x, y) -> Int32 in
    return x + y
}
print(result)
// 输出:
// 3
```

完美，你甚至感觉不到自己是在和 C 打交道！

GCD 和延时调用

因为 Playground 不进行特别配置的话是无法在线程中进行调度的，因此本节中的示例代码需要在 Xcode 项目环境中运行。在 Playground 中可能无法得到正确的结果。

GCD 是一种非常方便的使用多线程的方式。通过使用 GCD，我们可以在确保尽量简单的语法的前提下进行灵活的多线程编程。在“复杂必死”的多线程编程中，保持简单就是避免错误的金科玉律。好消息是在 Swift 中是可以无缝使用 GCD 的 API 的，而且得益于闭包特性的加入，使用起来比之前在 Objective-C 中更加简单方便。在这里我不打算花费很多时间介绍 GCD 的语法和要素，如果这么做的话就可以专门为 GCD 写上一节了。在下面我给出了一个日常里最通常会使用到的例子 (说这个例子能覆盖到日常的 GCD 使用的 50% 以上也不为过)，来展示一下 Swift 里的 GCD 调用会是什么样子：

```
// 创建目标队列
let workingQueue = dispatch_queue_create("my_queue", nil)

// 派发到刚创建的队列中，GCD 会负责进行线程调度
dispatch_async(workingQueue) {
    // 在 workingQueue 中异步进行
    print("努力工作")
    NSThread.sleepForTimeInterval(2) // 模拟两秒的执行时间

    dispatch_async(dispatch_get_main_queue()) {
        // 返回到主线程更新 UI
        print("结束工作，更新 UI")
    }
}
```

因为 UIKit 是只能在主线程工作的，如果我们在主线程进行繁重的工作的话，就会导致 app 出现“卡死”的现象：UI 不能更新，用户输入无法响应等等，是非常糟糕的用户体验。为了避免这种情况的出现，对于繁重 (如图像加滤镜等) 或会很长时间才能完成的 (如从网络下载图片) 处理，我们应该把它们放到后台线程进行，这样在用户看来 UI 还是可以交互的，也不会出现卡顿。在工作进行完成后，我们需要更新 UI 的话，必须回到主线程进行 (牢记 UI 相关的工作都需要在主线程执行，否则可能发生不可预知的错误)。

在日常的开发工作中，我们经常会遇到这样的需求：在 xx 秒后执行某个方法。比如切换界面 2 秒后开始播一段动画，或者提示框出现 3 秒后自动消失等等。以前在 Objective-C 中，我们可以使用一个 NSObject 的实例方法，`-performSelector:withObject:afterDelay:` 来指定在若干时间后执行某个 selector。在 Swift 2 之前，如果你新建一个 Swift 的项目，并且试图使用这个方法 (或者这个方法的其他一切变形) 的话，会发现这个方法并不存在。在 Swift 2 中虽然这一系列 `performSelector` 的方法被加回了标准库，但是由于 Swift 中创建一个 selector 并不是一件安全的事情 (你需要通过字符串来创建，这在之后代码改动时会很危险)，所以最好尽可能的话避免使用这个方法。另外，原来的 `performSelector:` 这套东西在 ARC 下并不是安全的。ARC 为了确保参数在方法运行期间的存在，在无法准确确定参数内存情况的时候，会将输入参数在方法开始时先进行 `retain`，然后在最后 `release`。而对于 `performSelector:` 这个方法我们并没有机会为被调用的方法指定参数，于是被调用的 selector 的输入有可能会是指向未知的垃圾内存地址，然

后...HOHO，要命的是这种崩溃还不能每次重现，想调试？见鬼去吧..

但是如果不论如何，我们都还想继续做延时调用的话应该怎么办呢？最容易想到的是使用 `NSTimer` 来创建一个若干秒后调用一次的计时器。但是这么做我们需要创建新的对象，和一个本来并不相干的 `NSTimer` 类扯上关系，同时也会用到 **Objective-C** 的运行时特性去查找方法等等，总觉得有点笨重。其实 **GCD** 里有一个很好用的延时调用我们可以加以利用写出很漂亮的方法来，那就是 `dispatch_after`。最简单的使用方法看起来是这样的：

```
let time: NSTimeInterval = 2.0
let delay = dispatch_time(DISPATCH_TIME_NOW,
                          Int64(time * Double(NSEC_PER_SEC)))
dispatch_after(delay, dispatch_get_main_queue()) {
    print("2 秒后输出")
}
```

代码非常简单，并没什么值得详细说明的。只是每次写这么多的话也挺累的，在这里我们可以稍微将它封装的好用一些，最好再加上取消的功能。在 **iOS 8** 中 **GCD** 得到了惊人的进化，现在我们可以通过将一个 `dispatch_block_t` 对象传递给 `dispatch_block_cancel`，来取消一个正在等待执行的 `block`。取消一个任务这样的特性，这在以前是 `NSOperation` 的专利，但是现在我们使用 **GCD** 也能达到同样的目的了。这里我们将类似地来尝试实现 `delay call` 的取消，整个封装也许有点长，但我还是推荐一读。大家也可以把它当作练习材料检验一下自己的 **Swift** 基础语法的掌握和理解的情况：

```
import Foundation

typealias Task = (cancel : Bool) -> Void

func delay(time:NSTimeInterval, task:()->()) -> Task? {

    func dispatch_later(block:()->()) {
        dispatch_after(
            dispatch_time(
                DISPATCH_TIME_NOW,
                Int64(time * Double(NSEC_PER_SEC))),
            dispatch_get_main_queue(),
            block)
    }

    var closure: dispatch_block_t? = task
    var result: Task?

    let delayedClosure: Task = {
        cancel in
        if let internalClosure = closure {
            if (cancel == false) {
                dispatch_async(dispatch_get_main_queue(), internalClosure);
            }
        }
        closure = nil
        result = nil
    }

    result = delayedClosure

    dispatch_later {
        if let delayedClosure = result {
```

```

        delayedClosure(cancel: false)
    }
}

return result;
}

func cancel(task: Task?) {
    task?.cancel()
}

```

使用的时候就很简单了，我们想在 2 秒以后干点儿什么的话：

```

delay(2) { println("2 秒后输出") }

```

想要取消的话，我们可以先保留一个对 `Task` 的引用，然后调用 `cancel`：

```

let task = delay(5) { print("拨打 110") }

// 仔细想一想..
// 还是取消为妙..
cancel(task)

```

获取对象类型

我们一再强调，如果遵循规则的话，Swift 会是一门相当安全的语言：不会存在类型的疑惑，绝大多数的内容应该能在编译期间就唯一确定。但是不论是 Objective-C 里很多开发者早已习惯的灵活性，还是在程序世界里总是千变万化的需求，都不可能保证一尘不变。我们有时候也需要引入一定的动态特性。而其中最为基本但却是最为有用的技巧是获取任意一个实例类型。

在 Objective-C 中我们可以轻而易举地做到这件事，使用 `-class` 方法就可以拿到对象的类，我们甚至可以用 `NSStringFromClass` 将它转换为一个能够打印出来的字符串：

```
NSDate *date = [NSDate date];
NSLog(@"%@", NSStringFromClass([date class]));
// 输出:
// __NSDate
```

在 Swift 中，我们会发现不管是纯 Swift 的 `class` 还是 `NSObject` 的子类，都没有像原来那样的 `class()` 方法来获取类型了。对于 `NSObject` 的子类，因为其实类的信息的存储方式并没有发生什么大的变化，因此我们可以求助于 Objective-C 的运行时，来获取类并按照原来的方式转换：

```
let date = NSDate()
let name: AnyClass! = object_getClass(date)
println(name)
// 输出:
// __NSDate
```

其中 `object_getClass` 是一个定义在 ObjectiveC 的 runtime 中的方法，它可以接受任意的 `AnyObject!` 并返回它的类型 `AnyClass!` (注意这里的叹号，它表明我们甚至可以输入 `nil`，并期待其返回一个 `nil`)。在 Swift 中其实为了获取一个 `NSObject` 或其子类的对象的实际类型，对这个调用其实有一个好看一些的写法，那就是 `dynamicType`。上面的代码用一种 "更 Swift" 一些的语言转换一下，会是这个样子：

```
let date = NSDate()
let name = date.dynamicType
print(name)
// 输出:
// __NSDate
```

很好，似乎我们的问题能解决了。但是仔细想想，我们上面用的都是 Objective-C 的动态特性，要是换成一个 Swift 内建类型的话，会怎么样呢？比如原生的 `String`，

```
let string = "Hello"
let name = string.dynamicType
```

```
print(name)
// 输出:
// String
```

可以看到对于 Swift 的原生类型，这种方式也是可行的。(值得指出的是，其实这里的真正的类型名字还带有 `module` 前缀，也就是 `Swift.String`。直接 `print` 只是调用了 `CustomStringConvertible` 中的相关方法而已，你可以使用 `debugPrint` 来进行确认。关于更多地关于 `print` 和 `debugPrint` 的细节，可以参考 [print 和 debugPrint](#) 一节的内容。)

在 Swift 1.2 之前，直接对 Swift 内建的非 `AnyObject` 类型使用 `.dynamicType` 可能会导致编译错误或者无法得到正确结果。但是随着 Swift 不断完善和改进，现在我们已经可以统一地使用 `.dynamicType` 来获取一个对象的类型了。

自省

程序设计和人类哲学所面临的同一个很重大的课题就是解决 "我是谁" 这个问题。在哲学里，这个问题属于自我认知的范畴，而在程序设计时，这个问题涉及到自省 (Introspection)。

向一个对象发出询问，以确定它是不是属于某个类，这种操作就称为自省。在 Objective-C 中因为 `id` 这样的可以指向任意对象的指针的存在 (其实严格来说 Objective-C 的指针的类型都是可以任意指向和转换的，它们只不过是帮助编译器理解你的代码而已)，我们经常需要向一个对象询问它是不是属于某个类。常用的方法有下面两类：

```
[obj1 isKindOfClass:[ClassA class]];
[obj2 isKindOfClass:[ClassB class]];
```

`-isKindOfClass:` 判断 `obj1` 是否是 `ClassA` 或者其子类的实例对象；而 `isKindOfClass:` 则对 `obj2` 做出判断，当且仅当 `obj2` 的类型为 `ClassB` 时返回为真。

这两个方法是 `NSObject` 的方法，所以我们在 Swift 中如果写的是 `NSObject` 的子类的话，直接使用这两个方法是没有任何问题的：

```
class ClassA: NSObject { }
class ClassB: ClassA { }

let obj1: NSObject = ClassB()
let obj2: NSObject = ClassB()

obj1.isKindOfClass(ClassA.self) // true
obj2.isKindOfClass(ClassA.self) // false
```

关于 `.self` 的用法，我们在 [.self 和 AnyClass](#) 一节里已经有所提及，这里就不再重复了。

在 Objective-C 中我们几乎所有的类都会是 `NSObject` 的子类，而在 Swift 的世界中，处于性能考虑，只要有可能，我们应该更倾向于选择那些非 `NSObject` 子类的 Swift 原生类型。对于那些不是 `NSObject` 的类，我们应该怎么确定其类型呢？

首先需要明确的一点是，我们为什么需要在运行时去确定类型。因为有泛型支持，Swift 对类型的推断和记录是完备的。因此在绝大多数情况下，我们使用的 Swift 类型都应该是在编译期间就确定的。如果你写的代码中经常需要检查和确定 `AnyObject` 到底是什么类的话，几乎就意味着你的代码设计出了问题 (或者你正在写一些充满各种 "小技巧" 的代码)。虽然没有太多的意义，但是我们还是可以做这件事情：

```
class ClassA { }
class ClassB: ClassA { }
```

```
let obj1: AnyObject = ClassB()
let obj2: AnyObject = ClassB()

obj1.isKindOfClass(ClassA.self) // true
obj2.isKindOfClass(ClassA.self) // false
```

在 Swift 中对于 `AnyObject` 使用最多的地方应该就是原来那些返回 `id` 的 Cocoa API 了。

为了快速确定类型，Swift 提供了一个简洁的写法：对于一个不确定的类型，我们现在可以使用 `is` 来进行判断。`is` 在功能上相当于原来的 `isKindOfClass`，可以检查一个对象是否属于某类型或其子类型。`is` 和原来的区别主要在于亮点，首先它不仅可以用于 `class` 类型上，也可以对 Swift 的其他像是 `struct` 或 `enum` 类型进行判断。使用起来是这个样子的：

```
class ClassA { }
class ClassB: ClassA { }

let obj: AnyObject = ClassB()

if (obj is ClassA) {
    print("属于 ClassA")
}

if (obj is ClassB) {
    print("属于 ClassB")
}
```

另外，编译器将对这种检查进行必要性的判断：如果编译器能够唯一确定类型，那么 `is` 的判断就没有必要，编译器将会抛出一个警告，来提示你并没有转换的必要。

```
let string = "String"
if string is String {
    // Do something
}

// 'is' test is always true
```


KVO

KVO (Key-Value Observing) 是 Cocoa 中公认的最强大的特性之一，但是同时它也以烂到家的 API 和极其难用著称。和[属性观察](#)不同，KVO 的目的并不是为当前类的属性提供一个钩子方法，而是为了其他不同实例对当前的某个属性 (严格来说是 `keypath`) 进行监听时使用的。其他实例可以充当一个订阅者的角色，当被监听的属性发生变化时，订阅者将得到通知。

这是一个很强大的属性，通过 KVO 我们可以实现很多松耦合的结构，使代码更加灵活和强大：像通过监听 `model` 的值来自动更新 UI 的绑定这样的工作，基本都是基于 KVO 来完成的。

在 Swift 中我们也是可以使用 KVO 的，但是仅限于在 `NSObject` 的子类中。这是可以理解的，因为 KVO 是基于 KVC (Key-Value Coding) 以及动态派发技术实现的，而这些东西都是 Objective-C 运行时的概念。另外由于 Swift 为了效率，默认禁用了动态派发，因此想用 Swift 来实现 KVO，我们还需要做额外的工作，那就是将想要观测的对象标记为 `dynamic`。

在 Swift 中，为一个 `NSObject` 的子类实现 KVO 的最简单的例子看起来是这样的：

```
class MyClass: NSObject {
    dynamic var date = NSDate()
}

private var myContext = 0

class Class: NSObject {

    var myObject: MyClass!

    override init() {
        super.init()
        myObject = MyClass()
        print("初始化 MyClass, 当前日期: \(myObject.date)")
        myObject.addObserver(self,
            forKeyPath: "date",
            options: .New,
            context: &myContext)

        delay(3) {
            self.myObject.date = NSDate()
        }
    }

    override func observeValueForKeyPath(keyPath: String?,
        ofObject object: AnyObject?,
        change: [String : AnyObject]?,
        context: UnsafeMutablePointer<Void>)
    {
        if let change = change where context == &myContext {
            let a = change[NSKeyValueChangeNewKey]
            print("日期发生变化 \(a)")
        }
    }
}

let obj = Class()
```

这段代码中用到了一个叫做 `delay` 的方法，这不是 `Swift` 的方法，而是本书在[延时调用](#)一节中实现的一个方法。这里您只需要理解我们是过了三秒以后在主线程将 `myObject` 中的时间更新到了当前时间即可。

我们标明了 `MyClass` 的 `date` 为 `dynamic`，然后在一个 `Class` 的 `init` 中将自己添加为该实例的观察者。接下来等待了三秒钟之后改变了这个对象的被观察属性，这时我们的观察方法就将被调用。运行这段代码，输出应该类似于：

```
初始化 MyClass, 当前日期: 2014-08-23 16:37:20 +0000
日期发生变化 Optional(2014-08-23 16:37:23 +0000)
```

别忘了，新的值是从字典中取出的。虽然我们能够确定 (其实是 `Cocoa` 向我们保证) 这个字典中会有相应的键值，但是在实际使用的时候我们最好还是进行一下判断或者 `Optional Binding` 后再加以使用，毕竟世事难料。

在 `Swift` 中使用 `KVO` 有两个显而易见的问题。

首先是 `Swift` 的 `KVO` 需要依赖的东西比原来多。在 `Objective-C` 中我们几乎可以没有限制地对所有满足 `KVC` 的属性进行监听，而现在我们需要属性有 `dynamic` 进行修饰。大多数情况下，我们想要观察的类不一定是 `dynamic` 修饰的 (除非这个类的开发者有意为之，否则一般也不会有人愿意多花功夫在属性前加上 `dynamic`，因为这毕竟要损失一部分性能)，并且有时候我们很可能也无法修改想要观察的类的源码。遇到这样的情况的话，一个可能可行的方案是继承这个类并且将需要观察的属性使用 `dynamic` 进行重写。比如刚才我们的 `MyClass` 中如果 `date` 没有 `dynamic` 的话，我们可能就需要一个新的 `MyChildClass` 了：

```
class MyClass: NSObject {
    var date = NSDate()
}

class MyChildClass: MyClass {
    dynamic override var date: NSDate {
        get { return super.date }
        set { super.date = newValue }
    }
}
```

对于这种重载，我们没有必要改变什么逻辑，所以在子类中简单地用 `super` 去调用父类里相关的属性就可以了。

另一个大问题是对于那些非 `NSObject` 的 `Swift` 类型怎么办。因为 `Swift` 类型并没有通过 `KVC` 实现，所以更不用谈什么对属性进行 `KVO` 了。对于 `Swift` 类型，语言中现在暂时还没有原生的类似 `KVO` 的观察机制。我们可能只能通过[属性观察](#)来实现一套自己的类似替代了。结合泛型和闭包这些 `Swift` 的先进特性 (当然是相对于 `Objective-C` 来说的先进特性)，把 `API` 做得比原来的 `KVO` 更优雅其实不是一件难事。[Observable-Swift](#) 就利用了这个思路实现了一套对 `Swift` 类型进行观察的机制，如果您也有类似的需求，不妨可以参考看看。

局部 scope

C 系语言中在方法内部我们是任意添加成对的大括号 `{ }` 来限定代码的作用范围的。这么做一般来说有两个好处，首先是超过作用域后里面的临时变量就将失效，这不仅可以使方法内的命名更加容易，也使得那些不被需要的引用的回收提前进行了，可以稍微提高一些代码的效率；另外，在合适的位置插入括号也利于方法的梳理，对于那些不太方便提取为一个单独方法，但是又应该和当前方法内的其他部分进行一些区分的代码，使用大括号可以将这样的结构进行一个相对自然的划分。

举一个不失一般性的例子，虽然我个人不太喜欢使用代码手写 UI，但钟情于这么做的人还是不在少数。如果我们要在 Objective-C 中用代码构建 UI 的话，我们一般会选择在 `-loadView` 里写一些类似这样的代码：

```
-(void)loadView {
    UIView *view = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320, 480)];

    UILabel *titleLabel = [[UILabel alloc]
        initWithFrame:CGRectMake(150, 30, 20, 40)];
    titleLabel.textColor = [UIColor redColor];
    titleLabel.text = @"Title";
    [view addSubview:titleLabel];

    UILabel *textLabel = [[UILabel alloc]
        initWithFrame:CGRectMake(150, 80, 20, 40)];
    textLabel.textColor = [UIColor redColor];
    textLabel.text = @"Text";
    [view addSubview:textLabel];

    self.view = view;
}
```

在这里只添加了两个 `view`，就已经够让人心烦的了。真实的界面当然会比这个复杂很多，想想看如果有十来个 `view` 的话，这段代码会变成什么样子吧。我们需要考虑对各个子 `view` 的命名，以确保它们的意义明确。如果我们在上面的代码中把某个配置 `textLabel` 的代码写错成了 `titleLabel` 的话，编译器也不会给我们任何警告。这种 `bug` 是非常难以发现的，因此在类似这种一大堆代码但是又不太可能进行重用的时候，我更推荐使用局部 `scope` 将它们分隔开来。比如上面的代码建议加上括号重写为以下形式，这样至少编译器会提醒我们一些低级错误，我们也可能更专注于每个代码块：

```
-(void)loadView {
    UIView *view = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320, 480)];

    {
        UILabel *titleLabel = [[UILabel alloc]
            initWithFrame:CGRectMake(150, 30, 20, 40)];
        titleLabel.textColor = [UIColor redColor];
        titleLabel.text = @"Title";
        [view addSubview:titleLabel];
    }
}
```

```

{
    UILabel *textLabel = [[UILabel alloc]
        initWithFrame:CGRectMake(150, 80, 20, 40)];
    textLabel.textColor = [UIColor redColor];
    textLabel.text = @"Text";
    [view addSubview:textLabel];
}

self.view = view;
}

```

在 Swift 中，直接使用大括号的写法是不支持的，因为这和闭包的定义产生了冲突。如果我们想类似地使用局部 **scope** 来分隔代码的话，一个不错的选择是定义一个接受 `()->()` 作为函数的全局方法，然后执行它：

```

func local(closure: ()->()) {
    closure()
}

```

在使用时，可以利用尾随闭包的特性模拟局部 **scope**：

```

override func loadView() {
    let view = UIView(frame: CGRectMake(0, 0, 320, 480))

    local {
        let titleLabel = UILabel(frame: CGRectMake(150, 30, 20, 40))
        titleLabel.textColor = UIColor.redColor()
        titleLabel.text = "Title"
        view.addSubview(titleLabel)
    }

    local {
        let textLabel = UILabel(frame: CGRectMake(150, 80, 20, 40))
        textLabel.textColor = UIColor.redColor()
        textLabel.text = "Text"
        view.addSubview(textLabel)
    }

    self.view = view
}

```

不过在 Swift 2.0 中，为了处理异常，Apple 加入了 `do` 这个关键字来作为捕获异常的作用域。这一功能恰好为我们提供了一个完美的局部作用域，现在我们可以简单地使用 `do` 来分隔代码了：

```

do {
    let titleLabel = UILabel(frame: CGRectMake(150, 80, 20, 40))
    //...
}

```

在 Objective-C 中还有一个很棒的技巧是使用 GNU C 的**声明扩展**来在限制局部作用域的时候同时进行赋值，运用得当的话，可以使代码更加紧凑和整洁。比如上面的 `titleLabel` 如果我们需要保

留一个引用的话，在 Objective-C 中可以写为：

```
self.titleLabel = ({
    UILabel *label = [[UILabel alloc]
        initWithFrame:CGRectMake(150, 30, 20, 40)];
    label.textColor = [UIColor redColor];
    label.text = @"Title";
    [view addSubview:label];
    label;
});
```

Swift 里当然没有 GNU C 的扩展，但是使用匿名的闭包的话，写出类似的代码也不是难事：

```
titleLabel = {
    let label = UILabel(frame: CGRectMake(150, 30, 20, 40))
    label.textColor = UIColor.redColor()
    label.text = "Title"
    self.view.addSubview(label)
    return label
}()
```

这也是一种隔离代码的很好的方式。

判等

我们在 Objective-C 时代，通常使用 `-isEqualToString:` 来在已经能确定比较对象和待比较对象都是 `NSString` 的时候进行字符串判等。Swift 中的 `String` 类型中是没有 `-isEqualToString:` 或者 `-isEqual:` 这样的方法的，因为这些毕竟是 `NSObject` 的东西。在 Swift 的字符串内容判等，我们简单地使用 `==` 操作符来进行：

```
let str1 = "快乐的字符串"
let str2 = "快乐的字符串"
let str3 = "开心的字符串"

str1 == str2 // true
str1 == str3 // false
```

在判等上 Swift 的行为和 Objective-C 有着巨大的差别。在 Objective-C 中 `==` 这个符号的意思是判断两个对象是否指向同一块内存地址。其实很多时候这并不是我们经常所期望的判等，我们更关心的往往还是对象的内容相同，而这种意义的相等即使两个对象引用的不是同一块内存地址时，也是可以做到的。Objective-C 中我们通常通过对 `-isEqual:` 进行重写，或者更进一步去实现类似 `-isEqualToString:` 这样的 `-isEqualToClass:` 的带有类型信息的方法来进行内容判等。如果我们没有在任意子类重写 `-isEqual:` 的话，在调用这个方法时会直接使用 `NSObject` 中的版本，去直接进行 Objective-C 的 `==` 判断。

在 Swift 中情况大不一样，Swift 里的 `==` 是一个操作符的声明，在 `Equatable` 里声明了这个操作符的接口方法：

```
protocol Equatable {
    func ==(lhs: Self, rhs: Self) -> Bool
}
```

实现这个接口的类型需要定义适合自己类型的 `==` 操作符，如果我们认为两个输入有相等关系的话，就应该返回 `true`。实现了 `Equatable` 的类型就可以使用 `==` 以及 `!=` 操作符来进行相等判定了 (在实现时我们只需要实现 `==`，`!=` 的话由标准库自动取反实现)。这和原来 Objective-C 的 `isEqual:` 的行为十分相似。比如我们在一个待办事项应用中，从数据库中取得带有使用 `uuid` 进行编号的待办条目，在实践中我们一般考虑就使用这个 `uuid` 来判定两个条目对象是不是同一条目。让这个表示条目的 `TodoItem` 类实现 `Equatable` 接口：

```
class TodoItem {
    let uuid: String
    var title: String

    init(uuid: String, title: String) {
        self.uuid = uuid
        self.title = title
    }
}
```

```
extension TodoItem: Equatable {

}

func ==(lhs: TodoItem, rhs: TodoItem) -> Bool {
    return lhs.uuid == rhs.uuid
}
```

对于 `==` 的实现我们并没有像实现其他一些接口一样将其放在对应的 `extension` 里，而是放在了全局的 `scope` 中。这是合理做法，因为你需要在全局范围内都能使用 `==`。事实上，Swift 的操作符都是全局的，关于操作符的更多信息，可以参看[操作符](#)一节。

Swift 的基本类型都重载了自己对应版本的 `==`，而对于 `NSObject` 的子类来说，如果我们使用 `==` 并且没有对于这个子类的重载的话，将转为调用这个类的 `-isEqual:` 方法。这样如果这个 `NSObject` 子类原来就实现了 `-isEqual:` 的话，直接使用 `==` 并不会造成它和 Swift 类型的行为差异；但是如果无法找到合适的重写的话，这个方法就将回滚到最初的 `NSObject` 里的实现，对引用对象地址进行直接比较。因此对于 `NSObject` 子类的判等你有两种选择，要么重载 `==`，要么重写 `-isEqual:`。如果你只在 Swift 中使用你的类的话，两种方式等效的；但是如果你还需要在 Objective-C 中使用这个类的话，因为 Objective-C 不接受操作符重载，只能使用 `-isEqual:`，这时你应该考虑使用第二种方式。

对于原来 Objective-C 中使用 `==` 进行的对象指针的判定，在 Swift 中提供的是另一个操作符 `===`。在 Swift 中 `===` 只有一种重载：

```
func ===(lhs: AnyObject?, rhs: AnyObject?) -> Bool
```

它用来判断两个 `AnyObject` 是否是同一个引用。

对于判等，和它紧密相关的一个话题就是[哈希](#)，因为哈希是一个稍微复杂的话题，所以我将它分成了一个单节。但是如果在实际项目中你需要重载 `==` 或者重写 `-isEqual:` 来进行判等的话，很可能你也会想看看有关[哈希](#)的内容，重载了判等的话，我们还需要提供一个可靠的哈希算法使得判等的对象在字典中作为 `key` 时不会发生奇怪的事情。

哈希

哈希表或者说散列表是程序世界中的一种基础数据结构，鉴于有太多的各类教程和资料已经将这个问题翻来覆去讲了无数遍，作为一个非科班出身的半路出家码农就不在数据结构理论或者哈希算法这方面班门弄斧了。简单说，我们需要为判等结果为相同的对象提供相同的哈希值，以保证在被用作字典的 **key** 时的确定性和性能。在这里，我们主要说说在 **Swift** 里对于哈希的使用。

在**判等**中我们提到，**Swift** 中对 `NSObject` 子类对象使用 `==` 时要是该子类没有实现这个操作符重载的话将回滚到 `-isEqual:` 方法。对于哈希计算，**Swift** 也采用了类似的策略。**Swift** 类型中提供了一个叫做 `Hashable` 的接口，实现这个接口即可为该类型提供哈希支持：

```
protocol Hashable : Equatable {
    var hashCode: Int { get }
}
```

Swift 的原生 `Dictionary` 中，**key** 一定是要实现了的 `Hashable` 接口的类型。像 `Int` 或者 `String` 这些 **Swift** 基础类型，已经实现了这个接口，因此可以用来作为 **key** 来使用。比如 `Int` 的 `hashCode` 就是它本身：

```
let num = 19
print(num.hashCode) // 19
```

对 **Objective-C** 熟悉的读者可能知道 `NSObject` 中有一个 `-hash` 方法。当我们对一个 `NSObject` 的子类的 `-isEqual:` 进行重写的时候，我们一般也需要将 `-hash` 方法重写，已提供一个判等为真时返回同样哈希值的方法。在 **Swift** 中，`NSObject` 也默认就实现了 `Hashable`，而且和判等的时候情况类似，`NSObject` 对象的 `hashCode` 属性的访问将返回其对应的 `-hash` 的值。

所以在重写哈希方法时候所采用的策略，与判等的时候是类似的：对于非 `NSObject` 的类，我们需要遵守 `Hashable` 并根据 `==` 操作符的内容给出哈希算法；而对于 `NSObject` 子类，需要根据是否需要在 **Objective-C** 中访问而选择合适的重写方式，去实现 `Hashable` 的 `hashCode` 或者直接重写 `NSObject` 的 `-hash` 方法。

也就是说，在 **Objective-C** 中，对于 `NSObject` 的子类来说，其实 `NSDictionary` 的安全性是通过人为来保障的。对于那些重写了判等但是没有重写对应的哈希方法的子类，编译器并不能给出实质性的帮助。而在 **Swift** 中，如果你使用非 `NSObject` 的类型和原生的 `Dictionary`，并试图将这个类型作为字典的 **key** 的话，编译器将直接抛出错误。从这方面来说，如果我们尽量使用 **Swift** 的话，安全性将得到大大增加。

类簇

虽然可能不太被重视，但类簇 (class cluster) 确实是 Cocoa 框架中广泛使用的设计模式之一。简单来说类簇就是使用一个统一的公共的类来订制定一的接口，然后在表面之下对应若干个私有类进行实现的方式。这么做最大的好处是避免的公开很多子类造成混乱，一个最典型的例子是

`NSNumber`：我们有一系列的不同的方法可以从整数，浮点数或者是布尔值来生成一个 `NSNumber` 对象，而实际上它们可能会是不同的私有子类对象：

```
NSNumber * num1 = [[NSNumber alloc] initWithInt:1];
// __NSCFNumber

NSNumber * num2 = [[NSNumber alloc] initWithFloat:1.0];
// __NSCFNumber

NSNumber * num3 = [[NSNumber alloc] initWithBool:YES];
// __NSCFBoolean
```

类簇在子类种类繁多，但是行为相对统一的时候对于简化接口非常有帮助。

在 Objective-C 中，`init` 开头的初始化方法虽然打着初始化的名号，但是实际做的事情和其他方法并没有太多不同之处。类簇在 Objective-C 中实现起来也很自然，在所谓的“初始化方法”中将 `self` 进行替换，根据调用的方式或者输入的类型，返回合适的私有子类对象就可以了。

但是 Swift 中的情况有所不同。因为 Swift 拥有真正的初始化方法，在初始化的时候我们只能得到当前类的实例，并且要完成所有的配置。也就是说对于一个公共类来说，是不可能在初始化方法中返回其子类的信息的。对于 Swift 中的类簇构建，一种有效的方法是使用工厂方法来进行。例如下面的代码通过 `Drinking` 的工厂方法将可乐和啤酒两个私有类进行了类簇化：

```
class Drinking {
    typealias LiquidColor = UIColor
    var color: LiquidColor {
        return LiquidColor.clearColor()
    }

    class func drinking(name: String) -> Drinking {
        var drinking: Drinking
        switch name {
        case "Coke":
            drinking = Coke()
        case "Beer":
            drinking = Beer()
        default:
            drinking = Drinking()
        }

        return drinking
    }
}

class Coke: Drinking {
    override var color: LiquidColor {
```

```

        return LiquidColor.blackColor()
    }
}

class Beer: Drinking {
    override var color: LiquidColor {
        return LiquidColor.yellowColor()
    }
}

let coke = Drinking.drinking("Coke")
coke.color // Black

let beer = Drinking.drinking("Beer")
beer.color // Yellow

```

通过[获取对象类型](#)中提到的方法，我们也可以确认 `coke` 和 `beer` 各自的动态类型分别是 `Coke` 和 `Beer`。

```

let cokeClass = NSStringFromClass(coke.dynamicType) //Coke
let beerClass = NSStringFromClass(beer.dynamicType) //Beer

```

Swizzle

Swizzle 是 Objective-C 运行时的黑魔法之一。我们可以通过 Swizzle 的手段，在运行时对某些方法的实现进行替换，这是 Objective-C 甚至说 Cocoa 开发中最为华丽，同时也是最为危险的技巧之一。

因为 Objective-C 在方法调用时是通过类的 dispatch table 来用 selector 对实现进行查找的，因此我们在运行时如果能够替换掉某个 selector 对应的实现，那么我们就能够在运行时“重新定义”这个方法的行为。如果你不太理解的话，可以想象成某个类能响应的方法是存放在一个类似字典的结构中的，键为方法的名字 (也就是 selector)，而值就是方法真正做的事情。执行某个方法时我们告诉 Objective-C 运行时想要执行的方法的名字，然后使用这个名字从这个“字典”中取值并执行。通过替换这里的值，我们就可以在不改变原来代码结构的情况下偷天换日了。

一般来说可能不太用得到这样的技术，但是在某些情况下会非常有用，特别是当我们需要触及到一些系统框架的东西的时候。比如我们已经有一个庞大的项目，并使用了很多 UIButton 来让用户交互。某一天，产品汪突然说我们需要统计一下整个 app 中用户点击所有按钮的次数。对于完全不懂技术的选手来说，在他们眼中这似乎不应该是什难事 -- 只要弄个计数器然后在每次点按钮的时候加一就可以了嘛。但是对于每一个以代码为生的人来说，面临的一个严峻的问题是，这要怎么办。

我们当然可以寻遍项目里的所有按钮点击后的事件代码，然后建立一个全局计数器来计数，但是，之后的维护怎么办，寻找的时候发生了遗漏怎么办，新加入的人不知道这茬怎么办？显然这是最糟糕的一条路。另一个方法是创建一个 UIButton 的子类，然后重写它的点击事件的方法。这种策略虽然好些，但是我们需要找遍项目中的按钮，并改变它们的继承关系，上面的那些问题也依然存在，而且要是我们已经在项目中使用了其他 UIButton 的子类的话，我们就不得不再去为那些子类创建新的子类，费时费力。

这种时候就该轮到 Swizzle 大显身手了。我们在全局范围内将所有的 UIButton 的发送事件的方法换掉，就可以一劳永逸地解决这个问题 -- 没有一段段代码的替换查找，不会遗漏任何按钮，之后开发中也不需要对这个计数的功能特别地注意什么。

在 Swift 中，我们也可以利用 Objective-C 运行时来进行 Swizzle。比如上面的例子，我们就可以使用这样的扩展来完成：

```
extension UIButton {
    class func xxx_swizzleSendAction() {
        struct xxx_swizzleToken {
            static var onceToken : dispatch_once_t = 0
        }
        dispatch_once(&xxx_swizzleToken.onceToken) {
            let cls: AnyClass! = UIButton.self

            let originalSelector = Selector("sendAction:to:forEvent:")
            let swizzledSelector = Selector("xxx_sendAction:to:forEvent:")

            let originalMethod =
                class_getInstanceMethod(cls, originalSelector)
```

```

        let swizzledMethod =
            class_getInstanceMethod(cls, swizzledSelector)

            method_exchangeImplementations(originalMethod, swizzledMethod)
    }
}

public func xxx_sendAction(action: Selector,
                           to: AnyObject!,
                           forEvent: UIEvent!)
{
    struct xxx_buttonTapCounter {
        static var count: Int = 0
    }

    xxx_buttonTapCounter.count += 1
    print(xxx_buttonTapCounter.count)
    xxx_sendAction(action, to: to, forEvent: forEvent)
}
}

```

在 `xxx_swizzleSendAction` 方法 (因为是向一个常用类中添加方法, 最好还是加上前缀以防万一) 中, 我们先获取将被替换的方法 (`sendAction:to:forEvent:`) 和用来替换它的方法 (`xxx_sendAction:to:forEvent:`) 的 `selector`, 然后通过运行时对这两个方法的具体实现进行了交换。在 `xxx_sendAction:to:forEvent:` 的实现中, 我们先将计数器进行加一, 然后输出。最后我们看起来是在这个方法中调用了自己, 似乎会形成一个死循环。但是因为我们实际上已经交换了 `sendAction:to:forEvent:` 和 `xxx_sendAction:to:forEvent:` 的实现, 所以在做这个调用时恰好调用到的是原来的那个方法的实现。同理, 在外部使用 `sendAction:to:forEvent:` 的时候 (也就是点击按钮的时候), 实际调用的实现会是我们在这一章定义的带有计数器累加的实现。

最后我们需要在 app 启动时调用这个 `xxx_swizzleSendAction` 方法。在 Objective-C 中我们一般在 `category` 的 `+load` 中完成, 但是 Swift 的 `extension` 和 Objective-C 的 `category` 略有不同, `extension` 并不是运行时加载的, 因此也没有加载时候就会被调用的类似 `load` 的方法。另外, `extension` 中也不应该做方法重写去覆盖 `load` (其实重写也是无效的)。事实上, Swift 实现的 `load` 并不是在 app 运行开始就被调用的。基于这些理由, 我们使用另一个类初始化时会被调用的方法来进行交换:

```

extension UIButton {
    override public class func initialize() {
        if self != UIButton.self {
            return
        }
        UIButton.xxx_swizzleSendAction()
    }
}

```

和 `+load` 不同的是, `+initialize` 会在当前类以及它的子类被初始化时调用。在这里我们对当前类的类型进行了判断, 来保证安全性。另外, 在 `xxx_swizzleSendAction` 中, 也使用一个 `once_token` 来保证交换代码仅会被执行一次。

现在, 我们所有的按钮事件都会走我们替换进去的方法了, 每点一次实际发送了事件的按钮, 你都能在控制台看到当前点击数的输出了。

这种方式的 Swizzle 使用了 Objective-C 的动态派发，对于 `NSObject` 的子类是可以直接使用的，但是对于 Swift 的类，因为默认并没有使用 Objective-C 运行时，因此也没有动态派发的方法列表，所以如果要 Swizzle 的是 Swift 类型的方法的话，我们需要将原方法和替换方法都加上 `dynamic` 标记，以指明它们需要使用动态派发机制。关于这方面的知识，可以参看 [@objc](#) 和 [dynamic](#) 的内容。

我们有另一种方法，甚至可以完全不借助 Objective-C 运行时，而是直接替换 Swift 调用时使用的封装过的类似“函数指针”，来达到对 Swift 类型进行“Swizzle”的目的。但是这个话题和背后的原理超出了本书的范围，如果你对此感兴趣，可以尝试看看 [SWRoute](#) 这个项目以及它背后的[原理](#)。

调用 C 动态库

C 是程序世界的宝库，在我们面向的设备系统中，也内置了大量的 C 动态库帮助我们完成各种任务。比如涉及到压缩的话我们很可能会借助于 `libz.dylib`，而像 xml 的解析的话一般链接 `libxml.dylib` 就会方便一些。

因为 Objective-C 是 C 的超集，因此在以前我们可以无缝地访问 C 的内容，只需要指定依赖并且导入头文件就可以了。但是骄傲的 Swift 的目的之一就是甩开 C 的历史包袱，所以现在在 Swift 中直接使用 C 代码或者 C 的库是不可能的。举个例子，计算某个字符串的 MD5 这样简单地需求，在以前我们直接使用 `CommonCrypto` 中的 `CC_MD5` 就可以了，但是现在因为我们在 Swift 中无法直接写 `#import <CommonCrypto/CommonCrypto.h>` 这样的代码，这些动态库暂时也没有 module 化，因此快捷的方法就只有借助于通过 Objective-C 来进行调用了。因为 Swift 是可以通过 `{product-module-name}-Bridging-Header.h` 来调用 Objective-C 代码的，于是 C 作为 Objective-C 的子集，自然也一并被解决了。比如对于上面提到的 MD5 的例子，我们就可以通过头文件导入以及添加 `extension` 来解决：

```
// TargetName-Bridging-Header.h
#import <CommonCrypto/CommonCrypto.h>

// StringMD5.swift
extension String {
    var MD5: String {
        let cString = self.cStringUsingEncoding(NSUTF8StringEncoding)
        let length = CUnsignedInt(
            self.lengthOfBytesUsingEncoding(NSUTF8StringEncoding)
        )
        let result = UnsafeMutablePointer<CUnsignedChar>.alloc(
            Int(CC_MD5_DIGEST_LENGTH)
        )

        CC_MD5(cString!, length, result)

        return String(format:
            "%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x",
            result[0], result[1], result[2], result[3],
            result[4], result[5], result[6], result[7],
            result[8], result[9], result[10], result[11],
            result[12], result[13], result[14], result[15])
    }
}

// 测试
print("swifter.tips".MD5)

// 输出
// dff88de99ff03d109de22fed4f71a273
```

当然，那些有强迫症的处女座读者可能不会希望在代码中沾上哪怕一点点 C 的东西，而更愿意面对纯纯的 Swift 代码，这样的话，也不妨重新制作 Swift 版本的轮子。比如对于 `CommonCrypto` 里的功能，已经可以在[这里](#)找到完整的 Swift 实现了，如果你只是需要 MD5 的话，[这里](#)也有一个实现。不过如果可能的话，暂时还是建议尽量使用现有的经过无数时间考验的 C 库。一方面现在

Swift 还很年轻，各种第三方库的引入和依赖机制还并不是很成熟；另外，使用动态库毕竟至少可以减少一些 app 尺寸，不是么？

输出格式化

C 系语言在字符串格式化输出上，需要通过类似 `%d`，`%f` 或者在 Objective-C 中的 `%@"` 这样的格式在指定的位置设定占位符，然后通过参数的方式将实际要输出的内容补充完整。例如 Objective-C 中常用的向控制台输出的 `NSLog` 方法就使用了这种格式化方法：

```
int a = 3;
float b = 1.234567;
NSString *c = @"Hello";
NSLog(@"int:%d float:%f string:%@", a, b, c);
// 输出:
// int:3 float:1.234567 string:Hello
```

在 Swift 里，我们在输出时一般使用的 `println` 中是支持字符串插值的，而字符串插值时将直接使用类型的 `Streamable`，`Printable` 或者 `DebugPrintable` 接口 (按照先后次序，前面的没有实现的话则使用后面的) 中的方法返回的字符串并进行打印。这样，我们就可以不借助于占位符，也不用再去记忆类型所对应的字符表示，就能很简单地输出各种类型的字符串描述了。比如上面的代码在 Swift 中可以等效写为：

```
let a = 3;
let b = 1.234567 // 我们在这里不去区分 float 和 Double 了
let c = "Hello"
print("int:\(a) double:\(b) string:\(c)")
// 输出:
// int:3 double:1.234567 string:Hello
```

不需要记忆麻烦的类型指代字符是很赞的事情，这大概也算摆脱了 C 留下的一个包袱吧。但是类 C 的这种字符串格式化也并非一无是处，在需要以一定格式输出的时候传统的方式就显得很有用，比如我们打算只输出上面的 `b` 中的小数点后两位的话，在 Objective-C 中使用 `NSLog` 时可以写成下面这样：

```
NSLog(@"float:%.2f", b);
// 输出:
// float:1.23
```

而到了 Swift 的 `print` 中，就没有这么幸运了，这个方法并不支持在字符串插值时使用像小数点限定这样的格式化方法。因此，我们可能不得不往回求助于使用类似原来那样的字符串格式化方法。`String` 的格式化初始方法可以帮助我们利用格式化的字符串：

```
let format = String(format: "%.2f", b)
print("double:\(format)")
// 输出:
// double:1.23
```


当然，每次这么写的话也很麻烦。如果我们需要大量使用类似的字符串格式化功能的话，我们最好为 `Double` 写一个扩展：

```
extension Double {  
    func format(f: String) -> String {  
        return String(format: "%\n(f)f", self)  
    }  
}
```

这样，在使用字符串插值和 `print` 的时候就能方便一些了：

```
let f = ".2"  
print("double:\n(b.format(f))")
```

Options

不要误会，我们谈的是 Options，不是 Optional。后者已经被谈论太多了，我不想再在上面再多补充什么了。

我们来说说 Options，或者在 Objective-C 中的 `NS_OPTIONS`，在 Swift 中是怎样的形式吧。

在 Objective-C 中，我们有很多需要提供某些选项的 API，它们一般用来控制 API 的具体行为配置等。举个例子，常用的 `UIView` 动画的 API 在使用时就可以进行选项指定：

```
[UIView animateWithDuration:0.3
                        delay:0.0
                        options:UIViewAnimationOptionCurveEaseIn |
                              UIViewAnimationOptionAllowUserInteraction
                        animations:^(
// ...
    } completion:nil];
```

我们可以使用 `|` 或者 `&` 这样的按位逻辑符对这些选项进行操作，这是因为一般来说在 Objective-C 中的 Options 的定义都是类似这样的按位错开的：

```
typedef NS_OPTIONS(NSUInteger, UIViewAnimationOptions) {
    UIViewAnimationOptionLayoutSubviews      = 1 << 0,
    UIViewAnimationOptionAllowUserInteraction = 1 << 1,
    UIViewAnimationOptionBeginFromCurrentState = 1 << 2,

    //...

    UIViewAnimationOptionTransitionFlipFromBottom = 7 << 20,
}
```

通过一个 `typedef` 的定义，我们可以使用 `NS_OPTIONS` 来把 `UIViewAnimationOptions` 映射为每一位都不同的一组 `NSUInteger`。不仅是这个动画的选项如此，其他的 `Option` 值也都遵循着相同的规范映射到整数上。如果我们不需要特殊的什么选项的话，可以使用 `kNilOptions` 作为输入，它被定义为数字 0。

```
enum {
    kNilOptions = 0
};
```

在 Swift 中，对于原来的枚举类型 `NS_ENUM` 我们有新的 `enum` 类型来对应。但是原来的 `NS_OPTIONS` 在 Swift 里显然没有枚举类型那样重要，并没有直接的源生类型来进行定义。原来的 `Option` 值现在被映射为了满足 `OptionSetType` 接口的 `struct` 类型，以及一组静态的 `get` 属性：

```
public struct UIViewAnimationOptions : OptionSetType {
    public init(rawValue: UInt)
    static var LayoutSubviews: UIViewAnimationOptions { get }
    static var AllowUserInteraction: UIViewAnimationOptions { get }

    //...

    static var TransitionFlipFromBottom: UIViewAnimationOptions { get }
}
```

这样一来，我们就可以用和原来类似的方式为方法指定选项了。用 **Swift** 重写上面的 `UIView` 动画的代码的话，我们可以使用这个新的 `struct` 的值。在使用时，可以用生成集合的方法来制定符合“或”逻辑多个选项：

```
UIView.animateWithDuration(0.3,
    delay: 0.0,
    options: [.CurveEaseIn, .AllowUserInteraction],
    animations: {},
    completion: nil)
```

`OptionSetType` 是实现了 `SetAlgebraType` 的，因此我们可以对两个集合进行各种集合运算，包括并集 (union)、交集 (intersect) 等等。另外，对于不需要选项输入的情况，也就是对应原来的 `kNilOptions`，现在我们直接使用一个空的集合 `[]` 来表示：

```
UIView.animateWithDuration(0.3,
    delay: 0.0,
    options: [],
    animations: {},
    completion: nil)
```

要实现一个 `Options` 的 `struct` 的话，可以参照已有的写法建立类并实现 `OptionSetType`。因为基本上所有的 `Options` 都是很相似的，所以最好是准备一个 `snippet` 以快速重用：

```
struct YourOption: OptionSetType {
    let rawValue: UInt
    static let None = YourOption(rawValue: 0)
    static let Option1 = YourOption(rawValue: 1)
    static let Option2 = YourOption(rawValue: 1 << 1)
    //...
}
```

数组 enumerate

使用 `NSArray` 时一个很常遇见的需求是在枚举数组内元素的同时也想使用对应的下标索引，在 `Objective-C` 中最方便的方式是使用 `NSArray` 的 `enumerateObjectsUsingBlock:` 方法。因为通过这个方法可以显式地同时得到元素和下标索引，这会有最好的可读性，并且 `block` 也意味着可以方便地在不同的类之间传递和复用这些代码。

比如我们想要对某个数组内的前三个数字进行累加 (请原谅我，因为这只是为这一节内容生造出来的例子，实际情况下我们就算有这样的需求可能也不太会这么处理)：

```
NSArray *arr = @[1, 2, 3, 4, 5];
__block NSInteger result = 0;
[arr enumerateObjectsUsingBlock:^(NSNumber *num, NSUInteger idx, BOOL *stop) {
    result += [num integerValue];
    if (idx == 2) {
        *stop = YES;
    }
}];

NSLog(@"%ld", result);
// 输出: 6
```

这里我们需要用到 `*stop` 这个停止标记的指针，并且直接设置它对应的值为 `YES` 来打断并跳出循环。而在 `Swift` 中，这个 `API` 的 `*stop` 被转换为了对应的 `UnsafeMutablePointer<ObjCBool>`。如果不明白 `Swift` 的指针的表示形式的话，一开始可能会被吓一跳，但是一旦当我们明白 `Unsafe` 开头的这些指针类型的用法之后，就会知道我们需要对应做的事情就是将这个指向 `ObjCBool` 的指针指向的内存的内容设置为 `true` 而已：

```
let arr: NSArray = [1, 2, 3, 4, 5]
var result = 0
arr.enumerateObjectsUsingBlock { (num, idx, stop) -> Void in
    result += num as! Int
    if idx == 2 {
        stop.memory = true
    }
}
print(result)
// 输出: 6
```

虽然说使用 `enumerateObjectsUsingBlock:` 非常方便，但是其实从性能上来说这个方法并不理想 ([这里](#)有一篇四年前的星期五问答阐述了这个问题，而且一直以来情况都没什么变化)。另外这个方法要求作用在 `NSArray` 上，这显然已经不符合 `Swift` 的编码方式了。在 `Swift` 中，我们在遇到这样的需求的时候，有一个效率，安全性和可读性都很好的替代，那就是快速枚举某个数组的 `EnumerateGenerator`，它的元素是同时包含了元素下标索引以及元素本身的多元组：

```
var result = 0
```

```
for (idx, num) in [1,2,3,4,5].enumerate() {  
    result += num  
    if idx == 2 {  
        break  
    }  
}  
print(result)
```

基本上来说，是时候可以和 `enumerateObjectsUsingBlock:` 说再见了。

类型编码 @encode

Objective-C 中有一些很冷僻但是如果知道的话在特定情况下会很有用的关键字，比如说通过类型获取对应编码的 `@encode` 就是其中之一。

在 Objective-C 中 `@encode` 使用起来很简单，通过传入一个类型，我们就可以获取代表这个类型的编码 C 字符串：

```
char *typeChar1 = @encode(int32_t);
char *typeChar2 = @encode(NSArray);
// typeChar1 = "i", typeChar2 = "{NSArray=#}"
```

我们可以对任意的类型进行这样的操作。这个关键字最常用的地方是在 Objective-C 运行时的消息发送机制中，在传递参数时，由于类型信息的缺失，需要类型编码进行辅助以保证类型信息也能够被传递。在实际的应用开发中，其实使用案例比较少：某些 API 中 Apple 建议使用 `NSValue` 的 `valueWithBytes:objCType:` 来获取值 (比如 `CIAffineClamp` 的[文档](#)里)，这时的 `objCType` 就需要类型的编码值；另外就是在类型信息丢失时我们可能需要用到这个特性，我们稍后会举一个这方面的例子。

Swift 使用了自己的 `Metatype` 来处理类型，并且在运行时保留了这些类型的信息，所以 Swift 并没有必要保留这个关键字。我们现在不能获取任意类型的类型编码了，但是在 Cocoa 中我们还是可以通过 `NSValue` 的 `objCType` 属性来获取对应值的类型指针：

```
class NSValue : NSObject, NSCopying, NSSecureCoding, NSCoding {
    //...
    var objCType: UnsafePointer<Int8> { get }

    //...
}
```

比如我们如果想要获取某个 Swift 类型的“等效的”类型编码的话，我们需要现将它转换为 `NSNumber` (`NSNumber` 是 `NSValue` 的子类)，然后获取类型：

```
let int: Int = 0
let float: Float = 0.0
let double: Double = 0.0

let intNumber: NSNumber = int
let floatNumber: NSNumber = float
let doubleNumber: NSNumber = double

String.fromCString(intNumber.objCType)
String.fromCString(floatNumber.objCType)
String.fromCString(doubleNumber.objCType)

// 结果分别为：
```

```
// {Some "q"}  
// {Some "f"}  
// {Some "d"}  
// 注意, fromCString 返回的是 `String?`
```

对于像其他一些可以转换为 `NSValue` 的类型, 我们也可以通过同样的方式获取类型编码, 一般来说这些类型会是某些 `struct`, 因为 `NSValue` 设计的初衷就是被作为那些不能直接放入 `NSArray` 的值的容器来使用的:

```
let p = NSValue(CGPoint: CGPointMake(3, 3))  
String.fromCString(p.objCType)  
// {Some "{CGPoint=dd}"}  
  
let t = NSValue(CGAffineTransform: CGAffineTransformIdentity)  
String.fromCString(t.objCType)  
// {Some "{CGAffineTransform=dddddd}"}  

```

有了这些信息之后, 我们就能够在这种类型信息可能损失的时候构建起准确的类型转换和还原了。

举例来说, 我们如果想要在 `NSUserDefaults` 中存储一些不同类型的数字, 然后读取时需要准确地还原为之前的类型的话, 最容易想到的应该是使用[类簇](#)来获取这些数字转为 `NSNumber` 后真正的类型, 然后存储。但是 `NSNumber` 的类簇子类都是私有的, 我们如果想要藉此判定的话, 就不得不使用私有 API, 这是不可接受的。变通的方法就是在存储时使用 `objCType` 获取类型, 然后将数字本身和类型的字符串一起存储。在读取时就可以通过匹配类型字符串和类型的编码, 确定数字本来所属的类型, 从而直接得到像 `Int` 或者 `Double` 这样的类型明确的量。

C 代码调用和 @asmname

如果我们导入了 `Darwin` 的 C 库的话，我们就可以在 `Swift` 中无缝地使用 `Darwin` 中定义的 C 函数了。它们涵盖了绝大多数 [C 标准库](#) 中的内容，可以说为程序设计提供了丰富的工具和基础。导入 `Darwin` 十分简单，只需要加上 `import Darwin` 即可。但事实上，`Foundation` 框架中包含了 `Darwin` 的导入，而我们在开发 app 时肯定会使用 `UIKit` 或者 `Cocoa` 这样的框架，它们又导入了 `Foundation`，因此我们在平时开发时并不需要特别做什么，就可以使用这些标准的 C 函数了。很让人开心的一件事情是 `Swift` 在导入时为我们将 `Darwin` 也进行了类型的自动转换对应，比如对于三角函数的计算输入和返回都是 `Swift` 的 `Double` 类型，而非 C 的类型：

```
func sin(x: Double) -> Double
```

使用起来也很简单，因为这些函数都是定义在全局的，所以只需要直接调用就可以了：

```
sin(M_PI_2)
// 输出: 1.0
```

而对于第三方的 C 代码，`Swift` 也提供了协同使用的方法。我们知道，`Swift` 中调用 `Objective-C` 代码非常简单，只需要将合适的头文件暴露在 `{product-module-name}-Bridging-Header.h` 文件中就行了。而如果我们想要调用非标准库的 C 代码的话，可以遵循同样的方式，将 C 代码的头文件在桥接的头文件中进行导入：

```
//test.h
int test(int a);

//test.c
int test(int a) {
    return a + 1;
}

//Module-Bridging-Header.h
#import "test.h"

//File.swift
func testSwift(input: Int32) {
    let result = test(input)
    print(result)
}

testSwift(1)
// 输出: 2
```

另外，我们甚至还有一种不需要借助头文件和 `Bridging-Header` 来导入 C 函数的方法，那就是使用 `Swift` 中的一个隐藏的符号 `@asmname`。 `@asmname` 可以通过方法名字将某个 C 函数直接映射为 `Swift` 中的函数。比如上面的例子，我们可以将 `test.h` 和 `Module-Bridging-Header.h` 都删掉，然后

将 swift 文件中改为下面这样，也是可以正常进行使用的：

```
//File.swift
//将 C 的 test 方法映射为 Swift 的 c_test 方法
@asmname("test") func c_test(a: Int32) -> Int32

func testSwift(input: Int32) {
    let result = c_test(input)
    print(result)
}

testSwift(1)
// 输出: 2
```

这种导入在第三方 C 方法与系统库重名导致调用发生命名冲突时，可以用来为其中之一的函数重新命名以解决问题。当然我们也可以利用 Module 名字 + 方法名字的方式来解决这个问题。

除了作为非头文件方式的导入之外，@asmname 还承担着和 @objc 的“重命名 Swift 中类和方法名字”类似的任务，这可以将 C 中不认可的 Swift 程序元素字符重命名为 ascii 码，以便在 C 中使用。

sizeof 和 sizeofValue

喜欢写 C 的读者可能会经常和 `sizeof` 打交道，不论是分配内存，I/O 操作，还是计算数组大小的时候基本都会用到。这个在 C 中定义的运算符可以作用于类型或者某个实际的变量，并返回其在内存中的尺寸 `size_t` (这是和平台无关的一个整数类型)。在 Cocoa 中，我们也有一部分 API 需要涉及到类型或者实例的内存尺寸，这时候就可以使用 `sizeof` 来计算。一个常见的用例是在从一个数组生成 `NSData` 的时候需要传入数据长度。因为在 Objective-C 中 `sizeof` 这个 C 运算符被保留了，因此我们可以直接这么做：

```
char bytes[] = {1, 2, 3};
NSData *data = [NSData dataWithBytes:&bytes length:sizeof(bytes)];
// <010203>
```

C 中的 `sizeof` 有两个版本，既可以接受类型，也可以接受某个具体的值：

```
sizeof(int)
sizeof(a)
```

与传统的 C 或者 Objective-C 里的运算符的存在形式不同，在 Swift 中，为了保证类型安全，`sizeof` 经过了一层包装。在 Swift 中 `sizeof` 不再是运算符，而是一个只能接受类型的方法。我们还可以找到一个接受具体值，并返回其尺寸的方法：`sizeofValue`。另外，这样个方法返回的都是看起来比较友好和直接的 `Int`，而不再是 `size_t`：

```
func sizeof<T>(_: T.Type) -> Int
func sizeofValue<T>(_: T) -> Int
```

虽然 `sizeofValue` 接受的是具体值，但是它和 C 时代的接受具体值的版本的 `sizeof` 行为并不相同。Swift 的 `sizeofValue` 所返回的是这个值实际的大小，而并非其内容的大小。具体来说，如果我们在 Swift 中想表示上面的 `bytes` 的话，我们会将其类型写为 `[CChar]`。在 C 或者 Objective-C 中，对 `bytes` 做 `sizeof` 返回的是整个数组内容在内存中占据的尺寸，每个 `char` 为 1，而数组元素为 3，因此这个值为 3。而在 Swift 中，我们如果直接对 `bytes` 做 `sizeofValue` 操作的话，将返回 8，这其实是在 64 位系统上一个引用的长度：

```
// C
char bytes[] = {1, 2, 3};
sizeof(bytes);
// 3

// Swift
var bytes: [CChar] = [1, 2, 3]
sizeofValue(bytes)
```

所以，我们不能简单地用 `sizeofValue` 来获取长度，而需要进行一些计算。上面的生成 `NSData` 的方法在 **Swift** 中书写的话，等效的代码应该是下面这样的：

```
var bytes: [CChar] = [1,2,3]
let data = NSData(bytes: &bytes, length:sizeof(CChar) * bytes.count)
```

作为练习，尝试一下指出下面的代码的结果分别是什么，并尝试解释一下

```
enum MyEnum: UInt16 {
    case A = 0
    case B = 1
    case C = 65535
}

sizeof(UInt16)
sizeof(MyEnum)
sizeofValue(MyEnum.A)
sizeofValue(MyEnum.A.rawValue)
```

delegate

Cocoa 开发中接口-委托 (protocol-delegate) 模式是一种常用的设计模式，它贯穿于整个 Cocoa 框架中，为代码之间的关系清理和解耦合做出了不可磨灭的贡献。

在 ARC 中，对于一般的 delegate，我们会在声明中将其指定为 weak，在这个 delegate 实际的对象被释放的时候，会被重置回 nil。这可以保证即使 delegate 已经不存在时，我们也不会由于访问到已被回收的内存而导致崩溃。ARC 的这个特性杜绝了 Cocoa 开发中一种非常常见的崩溃错误，说是救万千程序员于水火之中也毫不为过。

在 Swift 中我们当然也会希望这么做。但是当我们尝试书写这样的代码的时候，编译器不会让我们通过：

```
protocol MyClassDelegate {
    func method()
}

class MyClass {
    weak var delegate: MyClassDelegate?
}

class ViewController: UIViewController, MyClassDelegate {
    // ...
    var someInstance: MyClass!

    override func viewDidLoad() {
        super.viewDidLoad()

        someInstance = MyClass()
        someInstance.delegate = self
    }

    func method() {
        print("Do something")
    }

    //...
}

// weak var delegate: MyClassDelegate? 编译错误
// 'weak' cannot be applied to non-class type 'MyClassDelegate'
```

这是因为 Swift 的 protocol 是可以被除了 class 以外的其他类型遵守的，而对于像 struct 或是 enum 这样的类型，本身就不通过引用计数来管理内存，所以也不可能用 weak 这样的 ARC 的概念来进行修饰。

想要在 Swift 中使用 weak delegate，我们就需要将 protocol 限制在 class 内。一种做法是将 protocol 声明为 Objective-C 的，这可以通过在 protocol 前面加上 @objc 关键字来达到，Objective-C 的 protocol 都只有类能实现，因此使用 weak 来修饰就合理了：

```
@objc protocol MyClassDelegate {  
    func method()  
}
```

另一种可能更好的办法是在 `protocol` 声明的名字后面加上 `class`，这可以为编译器显式地指明这个 `protocol` 只能由 `class` 来实现。

```
protocol MyClassDelegate: class {  
    func method()  
}
```

相比起添加 `@objc`，后一种方法更能表现出问题的实质，同时也避免了过多的不必要的 Objective-C 兼容，可以说是一种更好的解决方式。

Associated Object

不知道是从什么时候开始，“是否能够通过 `Category` 给已有的类添加成员变量”就成为了一道 Objective-C 面试中的常见题目。不幸的消息是这个面试题目在 Swift 中可能依旧会存在。

得益于 Objective-C 的运行时和 Key-Value Coding 的特性，我们可以在运行时向一个对象添加值存储。而在使用 `Category` 扩展现有的类的功能的时候，直接添加实例变量这种行为是不被允许的，这时候一般就使用 `property` 配合 `Associated Object` 的方式，将一个对象“关联”到已有的要扩展的对象上。进行关联后，在对这个目标对象访问的时候，从外界看来，就似乎是直接在通过属性访问对象的实例变量一样，可以非常方便。

在 Swift 中这样的方法依旧有效，只不过在写法上可能有些不同。两个对应的运行时的 `get` 和 `set` `Associated Object` 的 API 是这样的：

```
func objc_getAssociatedObject(object: AnyObject!,
                             key: UnsafePointer<Void>
) -> AnyObject!

func objc_setAssociatedObject(object: AnyObject!,
                             key: UnsafePointer<Void>,
                             value: AnyObject!,
                             policy: objc_AssociationPolicy)
```

这两个 API 所接受的参数也都 Swift 化了，并且因为 Swift 的安全性，在类型检查上严格了不少，因此我们有必要也进行一些调整。在 Swift 中向某个 `extension` 里使用 `Associated Object` 的方式将对象进行关联的写法是：

```
// MyClass.swift
class MyClass {
}

// MyClassExtension.swift
private var key: Void?

extension MyClass {
    var title: String? {
        get {
            return objc_getAssociatedObject(self, &key) as? String
        }

        set {
            objc_setAssociatedObject(self,
                                    &key, newValue,
                                    .OBJC_ASSOCIATION_RETAIN_NONATOMIC)
        }
    }
}

// 测试
func printTitle(input: MyClass) {
    if let title = input.title {
```

```
        print("Title: \(title)")
    } else {
        print("没有设置")
    }
}

let a = MyClass()
printTitle(a)
a.title = "Swifter.tips"
printTitle(a)

// 输出:
// 没有设置
// Title: Swifter.tips
```

`key` 的类型在这里声明为了 `Void?`，并且通过 `&` 操作符取地址并作为 `UnsafePointer<Void>` 类型被传入。这在 Swift 与 C 协作和指针操作时是一种很常见的用法。关于 C 的指针操作和这些 `unsafe` 开头的类型的用法，可以参看 [UnsafePointer](#) 一节的内容。

Lock

无并发，不编码。而只要一说到多线程或者并发的代码，我们可能就很难绕开对于锁的讨论。简单来说，为了在不同线程中安全地访问同一个资源，我们需要这些访问顺序进行。Cocoa 和 Objective-C 中加锁的方式有很多，但是其中在日常开发中最常用的应该是 `@synchronized`，这个关键字可以用来修饰一个变量，并为其自动加上和解除互斥锁。这样，可以保证变量在作用范围内不会被其他线程改变。举个例子，如果我们有一个方法接受参数，需要这个方法是线程安全的话，就需要在参数上加锁：

```
- (void)myMethod:(id)anObj {
    @synchronized(anObj) {
        // 在括号内 anObj 不会被其他线程改变
    }
}
```

如果没有锁的话，一旦 `anObj` 的内容被其他线程修改的话，这个方法的行为很可能就无法预测了。

但是加锁和解锁都是要消耗一定性能的，因此我们不太可能为所有的方法都加上锁。另外其实在一个 app 中可能会涉及到多线程的部分是有限的，我们也没有必要为所有东西加上锁。过多的锁不仅没有意义，而且对于多线程编程来说，可能会产生很多像死锁这样的陷阱，也难以调试。因此在使用多线程时，我们应该尽量将保持简单作为第一要务。

扯远了，我们回到 `@synchronized` 上来。虽然这个方法很简单好用，但是很不幸的是在 Swift 中它已经 (或者是暂时) 不存在了。其实 `@synchronized` 在幕后做的事情是调用了 `objc_sync` 中的 `objc_sync_enter` 和 `objc_sync_exit` 方法，并且加入了一些异常判断。因此，在 Swift 中，如果我们忽略掉那些异常的话，我们想要 lock 一个变量的话，可以这样写：

```
func myMethod(anObj: AnyObject!) {
    objc_sync_enter(anObj)

    // 在 enter 和 exit 之间 anObj 不会被其他线程改变

    objc_sync_exit(anObj)
}
```

更进一步，如果我们喜欢以前的那种形式，甚至可以写一个全局的方法，并接受一个闭包，来将 `objc_sync_enter` 和 `objc_sync_exit` 封装起来：

```
func synchronized(lock: AnyObject, closure: () -> ()) {
    objc_sync_enter(lock)
    closure()
    objc_sync_exit(lock)
}
```


再结合 Swift 的尾随闭包的语言特性，这样，使用起来的时候就和 Objective-C 中很像了：

```
func myMethodLocked(anObj: AnyObject!) {  
    synchronized(anObj) {  
        // 在括号内 anObj 不会被其他线程改变  
    }  
}
```

Toll-Free Bridging 和 Unmanaged

有经验的读者看到这章的标题就能知道我们要谈论的是 Core Foundation。在 Swift 中对于 Core Foundation (以及其他一系列 Core 开头的框架) 在内存管理进行了一系列简化, 大大降低了与这些 Core Foundation (以下简称 CF) API 打交道的复杂程度。

首先值得一提的是对于 Cocoa 中 Toll-Free Bridging 的处理。Cocoa 框架中的大部分 NS 开头的类其实在 CF 中都有对应的类型存在, 可以说 NS 只是对 CF 在更高层面的一个封装。比如 NSURL 和它在 CF 中的 CFURLRef 内存结构其实是同样的, 而 NSString 则对应着 CFStringRef。

因为在 Objective-C 中 ARC 负责的只是 NSObject 的自动引用计数, 因此对于 CF 对象无法进行内存管理。我们在把对象在 NS 和 CF 之间进行转换时, 需要向编译器说明是否需要转移内存的管理权。对于不涉及到内存管理转换的情况, 在 Objective-C 中我们就直接在转换的时候加上 __bridge 来进行说明, 表示内存管理权不变。例如有一个 API 需要 CFURLRef, 而我们有一个 ARC 管理的 NSURL 对象的话, 这样来完成类型转换:

```
NSURL *fileURL = [NSURL URLWithString:@"SomeURL"];
SystemSoundID theSoundID;
//OSStatus AudioServicesCreateSystemSoundID(CFURLRef inFileURL,
//                                           SystemSoundID *outSystemSoundID);
OSStatus error = AudioServicesCreateSystemSoundID(
    (__bridge CFURLRef)fileURL,
    &theSoundID);
```

而在 Swift 中, 这样的转换可以直接省掉了, 上面的代码可以写为下面的形式, 简单了许多:

```
import AudioToolbox

let fileURL = NSURL(string: "SomeURL")
var theSoundID: SystemSoundID = 0

//AudioServicesCreateSystemSoundID(inFileURL: CFURL,
// _ outSystemSoundID: UnsafeMutablePointer<SystemSoundID>) -> OSStatus
AudioServicesCreateSystemSoundID(fileURL!, &theSoundID)
```

细心的读者可能会发现在 Objective-C 中类型的名字是 CFURLRef, 而到了 Swift 里成了 CFURL。CFURLRef 在 Swift 中是被 typealias 到 CFURL 上的, 其实不仅是 URL, 其他的各类 CF 类型都进行了类似的处理。这主要是为了减少 API 的迷惑: 现在这些 CF 类型的行为更接近于 ARC 管理下的对象, 因此去掉 Ref 更能表现出这一特性。

另外在 Objective-C 时代 ARC 不能处理的一个问题是 CF 类型的创建和释放。虽然不能自动化, 但是遵循命名规则来处理的话还是比较简单的: 对于 CF 系的 API, 如果 API 的名字中含有 Create, Copy 或者 Retain 的话, 在使用完成后, 我们需要调用 CFRelease 来进行释放。

不过 Swift 中这条规则已成明日黄花。既然我们有了明确的规则, 那为什么还要一次一次不厌其烦

地手动去写 `Release` 呢？基于这种想法，`Swift` 中我们不再需要显式地去释放带有这些关键字的内容了（事实上，含有 `CFRelease` 的代码甚至无法通过编译）。也就是说，`CF` 现在也在 `ARC` 的管辖范围之内了。其实背后的机理一点都不复杂，只不过在合适的地方加上了像 `CF_RETURNS_RETAINED` 和 `CF_RETURNS_NOT_RETAINED` 这样的标注。

但是有一点例外，那就是对于非系统的 `CF API`（比如你自己写的或者是第三方的），因为并没有强制机制要求它们一定遵照 `Cocoa` 的命名规范，所以贸然进行自动内存管理是不可行的。如果你没有明确地使用上面的标注来指明内存管理的方式的话，将这些返回 `CF` 对象的 `API` 导入 `Swift` 时，它们的类型会被对对应为 `Unmanaged<T>`。

这意味着在使用时我们需要手动进行内存管理，一般来说会使用得到的 `Unmanaged` 对象的 `takeUnretainedValue` 或者 `takeRetainedValue` 从中取出需要的 `CF` 对象，并同时处理引用计数。`takeUnretainedValue` 将保持原来的引用计数不变，在你明白你没有义务去释放原来的内存时，应该使用这个方法。而如果你需要释放得到的 `CF` 的对象的内存时，应该使用 `takeRetainedValue` 来让引用计数加一，然后在使用完后对原来的 `Unmanaged` 进行手动释放。为了能手动操作 `Unmanaged` 的引用计数，`Unmanaged` 中还提供了 `retain`，`release` 和 `autorelease` 这样的“老朋友”供我们使用。一般来说使用起来是这样的（当然这些 `API` 都是我虚构的）：

```
// CFGetSomething() -> Unmanaged<Something>
// CFCreateSomething() -> Unmanaged<Something>
// 两者都没有进行标注，Create 中进行了创建

let unmanaged = CFGetSomething()
let something = unmanaged.takeUnretainedValue()
// something 的类型是 Something，直接使用就可以了

let unmanaged = CFCreateSomething()
let something = unmanaged.takeRetainedValue()

// 使用 something

// 因为在取值时 retain 了，使用完成后进行 release
unmanaged.release()
```

切记，这些只有在没有标注的极少数情况下才会用到，如果你只是调用系统的 `CF API`，而不会去写自己的 `CF API` 的话，是没有必要关心这些的。

3. Swift 与开发环境及一些实践

Swift 命令行工具

Swift 的 REPL (Read-Eval-Print Loop) 环境可以让我们使用 Swift 进行简单的交互式编程。也就是说每输入一句语句就立即执行和输出。这在很多解释型的语言中是很常见的，非常适合用来对语言的特性进行学习。

要启动 REPL 环境，就要使用 Swift 的命令行工具，它是以 `xcrun` 命令的参数形式存在的。首先我们需要确认使用的 Xcode 版本是否是 6.1 或者以上，如果不是的话，可以在 Xcode 设置里 Locations 中的 Command Line Tools 一项中进行选择。然后我们就可以在命令行中输入 `xcrun swift` 来启动 REPL 环境了。

要指出的是，REPL 环境只是表现得像是即时的解释执行，但是其实质还是每次输入代码后进行编译再运行。这就限制了我们不太可能在 REPL 环境中做很复杂的事情。

另一个用法是直接将一个 `.swift` 文件作为命令行工具的输入，这样里面的代码也会被自动地编译和执行。我们甚至还可以在 `.swift` 文件最上面加上命令行工具的路径，然后将文件权限改为可执行，之后就可以直接执行这个 `.swift` 文件了：

```
#!/usr/bin/env swift
print("hello")
```

```
// Terminal
> chmod 755 hello.swift
> ./hello.swift

// 输出:
hello
```

这些特性与其他的解释性语言表现得完全一样。

相对于直接用 `swift` 命令执行，Swift 命令行工具的另一个常用的地方是直接脱离 Xcode 环境进行编译和生成可执行的二进制文件。我们可以使用 `swiftc` 来进行编译，比如下面的例子：

```
// MyClass.swift
class MyClass {
    let name = "XiaoMing"
    func hello() {
        println("Hello \(name)")
    }
}

// main.swift
let object = MyClass()
object.hello()
```

```
> swiftc MyClass.swift main.swift
```

将生成一个名叫 `main` 的可执行文件。运行之：

```
> ./main
// 输出:
// Hello XiaoMing
```

利用这个方法，我们就可以用 **Swift** 写出一些命令行的程序了。

最后想说明的一个 **Swift** 命令行工具的使用案例是生成汇编级别的代码。有时候我们想要确认经过优化后的汇编代码实际上做了些什么。`swiftc` 提供了参数来生成 `asm` 级别的汇编代码：

```
swiftc -O hello.swift -o hello.asm
```

再结合像是 [Hopper](#) 这样的工具，我们就能够了解编译器具体做了什么工作。

Swift 的命令行工具还有不少强大的功能，对此感兴趣的读者不妨使用 `swift --help` 以及 `swiftc --help` 来查看具体还有哪些参数可以使用。

随机数生成

随机数生成一直是程序员要面临的大问题之一，在高中电脑课堂上我们就知道，由 CPU 时钟，进程和线程所构建出的世界中，是没有真正的随机的。在给定一个随机种子后，使用某些神奇的算法我们可以得到一组伪随机的序列。

`arc4random` 是一个非常优秀的随机数算法，并且在 Swift 中也可以使用。它会返回给我们一个任意整数，我们想要在某个范围里的数的话，可以做模运算 (%) 取余数就行了。但是有个陷阱..

这是错误代码

```
let diceFaceCount = 6
let randomRoll = Int(arc4random()) % diceFaceCount + 1
print(randomRoll)
```

其实在 iPhone 5s 上完全没有问题，但是在 iPhone 5 或者以下的设备中，有时候程序会崩溃...请注意这个“有时候”..

最让程序员郁闷的事情莫过于程序有时候会崩溃而有时候又能良好运行。还好这里的情况比较简单，聪明的我们马上就能指出原因。其实 Swift 的 `Int` 是和 CPU 架构有关的：在 32 位的 CPU 上（也就是 iPhone 5 和前任们），实际上它是 `Int32`，而在 64 位 CPU (iPhone 5s 及以后的机型) 上是 `Int64`。 `arc4random` 所返回的值不论在什么平台上都是一个 `UInt32`，于是在 32 位的平台上就有一半几率在进行 `Int` 转换时越界，时不时的崩溃也就不足为奇了。

这种情况下，一种相对安全的做法是使用一个 `arc4random` 的改良版本：

```
func arc4random_uniform(_: UInt32) -> UInt32
```

这个改良版本接受一个 `UInt32` 的数字 `n` 作为输入，将结果归一化到 0 到 `n - 1` 之间。只要我们的输入不超过 `Int` 的范围，就可以避免危险的转换：

```
let diceFaceCount: UInt32 = 6
let randomRoll = Int(arc4random_uniform(diceFaceCount)) + 1
println(randomRoll)
```

最佳实践当然是为创建一个 `Range` 的随机数的方法，这样我们就能在之后很容易地复用，甚至设计类似与 `Randomable` 这样的接口了：

```
func randomInRange(range: Range<Int>) -> Int {
    let count = UInt32(range.endIndex - range.startIndex)
```

```
        return Int(arc4random_uniform(count)) + range.startIndex
    }

    for _ in 0...100 {
        print(randomInRange(1...6))
    }
}
```


print 和 debugPrint

在定义和实现一个类型的时候，Swift 中的一种非常常见，也是非常先进的做法是先定义最简单的类型结构，然后再通过扩展 (extension) 的方式来实现为数众多的接口和各种各样的功能。这种按照特性进行分离的设计理念对于功能的可扩展性的提升很有帮助。虽然在 Objective-C 中我们也可以通过类似的 protocol + category 的形式完成类似的事情，但 Swift 相比于原来的方式更加简单快捷。

CustomStringConvertible 和 CustomDebugStringConvertible 这两个接口就是很好的例子。对于一个普通的对象，我们在调用 print 对其进行打印时只能打印出它的类型：

```
class MyClass {
    var num: Int
    init() {
        num = 1
    }
}

let obj = MyClass()
print(obj)
// MyClass
```

对于 struct 来说，情况好一些。打印一个 struct 实例的话，会列举出它所有成员的名字和值：比如我们有一个日历应用存储了一些会议预约，model 类型包括会议的地点，位置和参与者的名字：

```
struct Meeting {
    var date: NSDate
    var place: String
    var attendeeName: String
}

let meeting = Meeting(date: NSDate(timeIntervalSinceNow: 86400),
    place: "会议室B1",
    attendeeName: "小明")
print(meeting)
// 输出:
// Meeting(date: 2015-08-10 03:15:55 +0000,
//      place: "会议室B1", attendeeName: "小明")
```

直接这样进行输出对了解对象的信息很有帮助，但也会存在问题。首先如果实例很复杂，我们将很难在其中找到想要的结果；其次，对于 class 的对象来说，只能得到类型名字，可以说是毫无帮助。我们可以对输出进行一些修饰，让它看起来好一些，比如使用格式化输出的方式：

```
print("于 \(meeting.date) 在 \(meeting.place) 与 \(meeting.attendeeName) 进行会议")
// 输出:
// 于 2014-08-25 11:05:28 +0000 在 会议室B1 与 小明 进行会议
```

但是如果每次输出的时候，我们都去写这么一大串东西的话，显然是不可接受的。正确的做法应该是使用 `CustomStringConvertible` 接口，这个接口定义了将该类型实例输出时所用的字符串。相对于直接在原来的类型定义中进行更改，我们更应该倾向于使用一个 `extension`，这样不会使原来的核心部分的代码变乱变脏，是一种很好的代码组织的形式：

```
extension Meeting: CustomStringConvertible {
    var description: String {
        return "于 \(self.date) 在 \(self.place) 与 \(self.attendeeName) 进行会议"
    }
}
```

这样，再当我们使用 `print` 时，就不再需要去做格式化，而是简单地将实例进行打印就可以了：

```
print(meeting)
// 输出：
// 于 2015-08-10 03:33:34 +0000 在 会议室B1 与 小明 进行会议
```

`CustomDebugStringConvertible` 与 `CustomStringConvertible` 的作用很类似，但是仅发生在调试中使用 `debugger` 来进行打印的时候的输出。对于实现了 `CustomDebugStringConvertible` 接口的类型，我们可以在给 `meeting` 赋值后设置断点并在控制台使用类似 `po meeting` 的命令进行打印，控制台输出将为 `CustomDebugStringConvertible` 中定义的 `debugDescription` 返回的字符串。

错误和异常处理

在开始这一节的内容之前，我想先阐明两个在很多时候被混淆的概念，那就是异常 (exception) 和错误 (error)。

在 Objective-C 开发中，异常往往是由程序员的错误导致的 app 无法继续运行，比如我们向一个无法响应某个消息的 `NSObject` 对象发送了这个消息，会得到 `NSInvalidArgumentException` 的异常，并告诉我们 "unrecognized selector sent to instance"; 比如我们使用一个超过数组元素数量的下标来试图访问 `NSArray` 的元素时，会得到 `NSRangeException`。类似由于这样所导致的程序无法运行的问题应该在开发阶段就被全部解决，而不应当出现在实际的产品中。相对来说，由 `NSError` 代表的错误更多地是指那些“合理的”，在用户使用 app 中可能遇到的情况：比如登陆时用户名密码验证不匹配，或者试图从某个文件中读取数据生成 `NSData` 对象时发生了问题 (比如文件被意外修改了) 等等。

但是 `NSError` 的使用方式其实变相在鼓励开发者忽略错误。想一想在使用一个带有错误指针的 API 时我们做的事情吧。我们会在 API 调用中产生和传递 `NSError`，并藉此判断调用是否失败。作为某个可能产生错误的方法的使用者，我们用传入 `NSErrorPointer` 指针的方式来存储错误信息，然后在调用完毕后去读取内容，并确认是否发生了错误。比如在 Objective-C 中，我们会写类似这样的代码：

```
NSError *error;
BOOL success = [data writeToFile: path options: options error: &error];
if(error) {
    // 发生了错误
}
```

这非常棒，但是有一个问题：在绝大多数情况下，这个方法并不会发生什么错误，而很多工程师也为了省事和简单，会将输入的 `error` 设为 `nil`，也就是不关心错误 (因为可能他们从没见过这个 API 返回错误，也不知要如何处理)。于是调用就变成了这样：

```
[data writeToFile: path options: options error: nil];
```

但是事实上这个 API 调用是会出错的，比如设备的磁盘空间满了的时候，写入将会失败。但是当这个错误出现并让你的 app 陷入难堪境地的时候，你几乎无从下手进行调试 -- 因为系统曾经尝试过通知你出现了错误，但是你却选择视而不见。

在 Swift 2.0 中，Apple 为这么语言引入了异常机制。现在，这类带有 `NSError` 指针作为参数的 API 都被改为了可以抛出异常的形式。比如上面的 `writeToFile:options:error:`，在 Swift 中变成了：

```
public func writeToFile(path: String,
```

```
options writeOptionsMask: NSDataWritingOptions) throws
```

我们在使用这个 API 的时候，不再像之前那样传入一个 `error` 指针去等待方法填充，而是变为使用 `try catch` 语句：

```
do {
    try d.writeToFile("Hello", options: [])
} catch let error as NSError {
    print ("Error: \(error.domain)")
}
```

如果你不使用 `try` 的话，是无法调用 `writeToFile:` 方法的，它会产生一个编译错误，这让我们无法有意无意地忽视掉这些错误。在上面的示例中 `catch` 将抛出的异常 (这里就是个 `NSError`) 用 `let` 进行了类型转换，这其实主要是针对 Cocoa 现有的 API 的，是对历史的一种妥协。对于我们新写的可抛出异常的 API，我们应当抛出一个实现了 `ErrorType` 的类型，`enum` 就非常合适，举个例子：

```
enum LoginError: ErrorType {
    case UserNotFound, UserPasswordNotMatch
}

func login(user: String, password: String) throws {
    //users 是 [String: String]，存储[用户名:密码]

    if !users.keys.contains(user) {
        throw LoginError.UserNotFound
    }

    if users[user] != password {
        throw LoginError.UserPasswordNotMatch
    }

    print("Login successfully.")
}
```

这样的 `ErrorType` 可以非常明确地指出问题所在。在调用时，`catch` 语句实质上是在进行模式匹配：

```
do {
    try login("onevcats", password: "123")
} catch LoginError.UserNotFound {
    print("UserNotFound")
} catch LoginError.UserPasswordNotMatch {
    print("UserPasswordNotMatch")
}

// Do something with login user
```

可以看出，在 Swift 中，我们虽然把这块内容叫做“异常”，但是实质上它更多的还是“错误”而非真正意义上的异常。

如果你之前写过 **Java** 或者 **C#** 的话，会发现 **Swift** 中的 `try catch` 块和它们中的有些不同。在那些语言里，我们会把可能抛出异常的代码都放在一个 `try` 里，而 **Swift** 中则是将它们放在 `do` 中，并只在可能发生异常的语句前添加 `try`。相比于 **Java** 或者 **C#** 的方式，**Swift** 里我们可以更清楚地知道是哪一个调用可能抛出异常，而不必逐句查阅文档。

当然，**Swift** 现在的异常机制也并不是十全十美的。最大的问题是类型安全，不借助于文档的话，我们现在是无法从代码中直接得知所抛出的异常的类型。比如上面的 `login` 方法，光看方法定义我们并不知道 `LoginError` 会被抛出。一个理想中的异常 **API** 可能应该是这样的：

```
func login(user: String, password: String) throws LoginError
```

很大程度上，这是由于要与以前的 `NSError` 兼容所导致的妥协，对于之前的使用 `NSError` 来表达错误的 **API**，我们所得到的错误对象本身就是用像 `domain` 或者 `error number` 这样的属性来进行区分和定义的，这与 **Swift 2.0** 中的异常机制所抛出的直接使用类型来描述错误的思想暂时是无法兼容的。不过有理由相信随着 **Swift** 的迭代更新，这个问题会在不久的将来得到解决。

另一个限制是对于非同步的 **API** 来说，抛出异常是不可用的 -- 异常只是一个同步方法专用的处理机制。**Cocoa** 框架里对于异步 **API** 出错时，保留了原来的 `NSError` 机制，比如很常用的

`NSURLSession` 中的 `dataTask` **API**：

```
func dataTaskWithURL(_ url: NSURL,
                    completionHandler completionHandler: ((NSData!,
                                                            NSURLResponse!,
                                                            NSError!) -> Void)?) -> NSURLSessionDataTask
```

对于异步 **API**，虽然不能使用异常机制，但是因为这类 **API** 一般涉及到网络或者耗时操作，它所产生的错误的可能性要高得多，所以开发者们其实无法忽视这样的错误。但是像上面这样的 **API** 其实我们在日常开发中往往并不会去直接使用，而会选择进行一些封装，以求更方便地调用和维护。一种现在比较常用的方式就是借助于 `enum`。作为 **Swift** 的一个重要特性，枚举 (`enum`) 类型现在是可以与其他的实例进行绑定的，我们还可以让方法返回枚举类型，然后在枚举中定义成功和错误的状态，并分别将合适的对象与枚举值进行关联：

```
enum Result {
    case Success(String)
    case Error(NSError)
}

func doSomethingParam(param:AnyObject) -> Result {
    //...做某些操作，成功结果放在 success 中
    if success {
        return Result.Success("成功完成")
    } else {
        let error = NSError(domain: "errorDomain", code: 1, userInfo: nil)
        return Result.Error(error)
    }
}
```

在使用时，利用 `switch` 中的 `let` 来从枚举值中将结果取出即可：

```
let result = doSomethingParam(path)

switch result {
    case let .Success(ok):
        let serverResponse = ok
    case let .Error(error):
        let serverResponse = error.description
}
```

在 `Swift 2.0` 中，我们甚至可以在 `enum` 中指定泛型，这样就使结果统一化了。

```
enum Result<T> {
    case Success(T)
    case Failure(NSError)
}
```

我们只需要在返回结果时指明 `T` 的类型，就可以使用同样的 `Result` 枚举来代表不同的返回结果了。这么做可以减少代码复杂度和可能的状态，同时不是优雅地解决了类型安全的问题，可谓一举两得。

因此，在 `Swift 2` 时代中的错误处理，现在一般的最佳实践是对于同步 `API` 使用异常机制，对于异步 `API` 使用泛型枚举。

断言

断言 (assertion) 在 Cocoa 开发里一般用来在检查输入参数是否满足一定条件，并对其进行“论断”。这是一个编码世界中的哲学问题，我们代码的使用者 (有可能是别的程序员，也有可能是未来的自己) 很难做到在不知道实现细节的情况下去对自己的输入进行限制。大多数时候编译器可以帮助我们进行输入类型的检查，但是如果代码需要在特定的输入条件下才能正确运行的话，这种更细致的条件就难以控制了。在超过边界条件的输入的情况下，我们的代码可能无法正确工作，这就需要在代码实现中进行一些额外工作。

一种容易想到的做法是在方法内部使用 `if` 这样的条件控制来检测输入，如果遇到无法继续的情况，就提前返回或者抛出错误。但是这样的做法无疑增加了 API 使用的复杂度，也导致了很多运行时的额外开销。对于像判定输入是否满足某种条件的运用情景，我们有更好的选择，那就是断言。

Swift 为我们提供了一系列的 `assert` 方法来使用断言，其中最常用的一个是：

```
func assert(@autoclosure condition: () -> Bool,
            @autoclosure _ message: () -> String = default,
            file: StaticString = default,
            line: UInt = default)
```

如果您对参数的默认值的 `default` 感兴趣的话，可以看看[默认参数](#)一节的内容，有简单介绍。

在使用时，最常见的情况是给定条件和一个简单的说明。举一个在温度转换时候的例子，我们想要把摄氏温度转为开尔文温度的时候，因为[绝对零度](#)永远不能达到，所以我们不可能接受一个小于 -273.15 摄氏度的温度作为输入：

```
func convertToKelvin(# celsius: Double) -> Double {
    assert(celsius > absoluteZeroInCelsius, "输入的摄氏温度不能低于绝对零度")
    return celsius - absoluteZeroInCelsius
}

let roomTemperature = convertToKelvin(celsius: 27)
// roomTemperature = 300.15

let tooCold = convertToKelvin(celsius: -300)
// 运行时错误:
// assertion failed:
// 输入的摄氏温度不能低于绝对零度 : file {YOUR_FILE_PATH}, line {LINE_NUMBER}
```

在遇到无法处理的输入时，运行会产生错误，保留堆栈，并抛出我们预设的信息，用来提醒调用这段代码的用户。

断言的另一个优点是它是一个开发时的特性，只有在 `Debug` 编译的时候有效，而在运行时是不被

编译执行的，因此断言并不会消耗运行时的性能。这些特点使得断言成为面向程序员的在调试开发阶段非常合适的调试判断，而在代码发布的时候，我们也不需要刻意去将这些断言手动清理掉，非常方便。

虽然默认情况下只在 **Release** 的情况下断言才会被禁用，但是有时候我们可能出于某些目的希望断言在调试开发时也暂时停止工作，或者是在发布版本中也继续有效。我们可以通过显式地添加编译标记达到这个目的。在对应 **target** 的 **Build Settings** 中，我们在 **Swift Compiler - Custom Flags** 中的 **Other Swift Flags** 中添加 `-assert-config Debug` 来强制启用断言，或者 `-assert-config Release` 来强制禁用断言。当然，除非有充足的理由，否则并不建议做这样的改动。如果我们需要在 **Release** 发布时在无法继续时将程序强行终止的话，应该选择使用 `fatalError`。

原来在 Objective-C 中使用的断言函数 `NSAssert` 在 Swift 中已经被彻底移除，和我们永远地说再见了。

fatalError

细心的读者可能会发现，在我们调试一些纯 Swift 类型出现类似数组越界这样的情况时，我们在控制台得到的报错信息会和传统调试 `NSObject` 子类时不太一样，比如在使用 `NSArray` 时：

```
let array: NSArray = [1,2,3]
array[100]
// 输出:
// *** Terminating app due to uncaught exception 'NSRangeException',
// reason: '*** -[__NSArrayI objectAtIndex:]:
// index 100 beyond bounds [0 .. 2]'
```

而如果我们使用 Swift 类型的话：

```
let array = [1,2,3]
array[100]
// 输出:
// fatal error: Array index out of range
```

在调试时我们可以使用断言来排除类似这样的问题，但是断言只会在 `Debug` 环境中有效，而在 `Release` 编译中所有的断言都将被禁用。在遇到确实因为输入的错误无法使程序继续运行的时候，我们一般考虑以产生致命错误 (`fatalError`) 的方式来终止程序。

`fatalError` 的使用非常简单，它的 API 和断言的比较类似

```
@noreturn func fatalError(@autoclosure message: () -> String = default,
                           file: StaticString = default,
                           line: UInt = default)
```

关于语法，唯一需要解释的是 `@noreturn`，这表示调用这个方法的话可以不再需要返回值，因为程序整个都将终止。这可以帮助编译器进行一些检查，比如在某些需要返回值的 `switch` 语句中，我们只希望被 `switch` 的内容在某些范围内，那么我们在可以在不属于这些范围的 `default` 块里直接写 `fatalError` 而不再需要指定返回值：

```
enum MyEnum {
    case Value1, Value2, Value3
}

func check(someValue: MyEnum) -> String {
    switch someValue {
    case .Value1:
        return "OK"
    case .Value2:
        return "Maybe OK"
    default:
        fatalError()
    }
}
```

```

default:
    // 这个分支没有返回 String, 也能编译通过
    fatalError("Should not show!")
}
}

```

在我们实际自己编码的时候，经常会有不想让别人调用某个方法，但又不得不将其暴露出来的时候。一个最常见并且合理的需求就是“[抽象类型或者抽象函数](#)”。在很多语言中都有这样的特性：父类定义了某个方法，但是自己并不给出具体实现，而是要求继承它的子类去实现这个方法，而在 Objective-C 和 Swift 中都没有直接的这样的抽象函数语法支持，虽然在 Cocoa 中对于这类需求我们有时候会转为依赖接口和委托的设计模式来变通地实现，但是其实 Apple 自己在 Cocoa 中也有很多类似抽象函数的设计。比如 `UIActivity` 的子类必须要实现一大堆指定的方法，而正因为缺少抽象函数机制，这些方法都必须在文档中写明。

在面对这种情况时，为了确保子类实现这些方法，而父类中的方法不被错误地调用，我们就可以利用 `fatalError` 来在父类中强制抛出错误，以保证使用这些代码的开发者留意到他们必须在自己的子类中实现相关方法：

```

class MyClass {
    func methodMustBeImplementedInSubclass() {
        fatalError("这个方法必须在子类中被重写")
    }
}

class YourClass: MyClass {
    override func methodMustBeImplementedInSubclass() {
        print("YourClass 实现了该方法")
    }
}

class TheirClass: MyClass {
    func someOtherMethod() {

    }
}

YourClass().methodMustBeImplementedInSubclass()
// YourClass 实现了该方法

TheirClass().methodMustBeImplementedInSubclass()
// 这个方法必须在子类中被重写

```

不过一个好消息是 Apple [意识到了](#)抽象函数这个特性的缺失，所以很可能在今后的 Swift 版本中我们能看到这个语法特性的加入。

不仅仅是对于类似抽象函数的使用中可以选择 `fatalError`，对于其他一切我们不希望别人随意调用，但是又不得不去实现的方法，我们都应该使用 `fatalError` 来避免任何可能的误会。比如父类标明了某个 `init` 方法是 `required` 的，但是你的子类永远不会使用这个方法来自初始化时，就可以采用类似的方式，被广泛使用 (以及被广泛讨厌的) `init(coder: NSCoder)` 就是一个例子。在子类中，我们往往会写：

```

required init(coder: NSCoder) {

```

```
fatalError("NSCoding not supported")  
}
```

来避免编译错误。

代码组织和 Framework

Apple 为了 iOS 平台的安全性考虑，是不允许动态链接非系统的框架的。因此在 app 开发中我们所使用的第三方框架如果是库文件的方式提供的话，一定都是需要链接并打包进最后的二进制可执行文件中的静态库。最初级和原始的静态库是以 `.a` 的二进制文件加上一些 `.h` 的头文件进行定义的形式提供的，这样的静态库在使用时比较麻烦，我们除了将其添加到项目和配置链接外，还需要进行指明头文件位置等工作。这样造成的结果不仅是添加起来比较麻烦，而且因为头文件的路径可能在不同环境下会存在不一样的情况，而造成项目在换一个开发环境后就因为配置问题造成无法编译。有过这种经历的开发人员都知道，调配开发环境是一件非常让人讨厌和耗费时间的事情。

而 Apple 自己的框架都是 `.framework` 为后缀的动态框架，是集成在操作系统中的，我们使用这些框架的时候只需要在 `target` 配置时进行指明就可以，非常方便。

因为 `framework` 的易用性，因此很多开发者都很喜欢类似的“即拖即用，无需配置”的体验。一些框架和库的开发者为了使用体验一般会用一些[第三方提供的方法](#)来模拟地生成行为类似的框架，比如 [Dropbox](#) 或者 [Facebook](#) 的 iOS SDK 都是基于这种技术完成的。

但是要特别指出，虽然和 Apple 的框架的后缀名一样是 `.framework`，使用方式也类似，但是这些第三方框架都是实实在在的静态库，每个 app 需要在编译的时候进行独立地链接。

从 Xcode 6 开始 Apple 官方提供了单独制作类似的 `framework` 的方法，这种便利性可能会使代码的组织方式发生重大变化。我们现在可以在一个 app 项目中添加新的类型为 Cocoa Touch Framework 的 `target`，并在同一个项目中通过 `import` 这个 `target` 的 `module` 名字 (一般和这个 `target` 的名字是一样的，除非使用了一些像中杠 `-` 这样在 `module` 名中的非法字符)，来引入并进行使用。

这类框架在运行时是嵌入在 app 的 `bundle` 中进行动态链接的，我们的 app 本体以及各个 `extension` 都可以使用。这么做的一个明显的好处是我们可以不同 `target` 之间很简单地重用代码，因为我们总会有一些在 app 本身和扩展中重复的东西，这时候将它们用框架的形式组织起来会是一个很好的选择。另一方面，就算你没有计划开发扩展，尝试将一部分代码分离到框架中也是有助于我们梳理项目的架构的。比如将所有的模型层组织为一个框架，如果你在这个过程中发现有困难的话，这很可能就是你需要重新考虑和重构项目架构的信号了。

但是这样制作的框架只能嵌入在自己的 app 中，如果我们希望将自己制作的框架提供给别人使用的话，我们可以新建一个专门生成框架的项目。接下来我们会通过一个简单的例子来告诉你应该怎么做，但是在这之前，需要先说明，使用框架项目并单独导出 `framework` 文件这种做法，是为 Objective-C 准备的。因为 Swift 暂时并不是稳定版本，而 Swift 的运行时也是经常改变，并且没有集成到操作系统中，所以官方并不推荐单独为 Swift 制作框架。我们虽然可以使用纯 Swift 制作可用的第三方库 (接下来你会看到要怎么做)，但是并不能保证它在所有的运行环境中都能良好工作。关于 Swift 代码的兼容性，可以参看[相关章节](#)的内容。

在当前 Swift 中，我们使用 Swift 框架的最佳实践是将整个框架项目 (包括其中源代码) 以项

目依赖的方式添加到自己的项目中，并一起进行编译使用。本节所要讲述的是制作单独的编译好的框架文件供别人使用，虽然暂时还不建议将这种方法用在实际项目之中，但是这里着重想展现的是使用 Swift 制作框架文件的可能性。

当然你也可以使用 Objective-C 来制作框架，这样就没有这些限制了，因为本来这个特性现在暂时也只是为 Objective-C 准备的。使用 Objective-C 制作框架的过程和 Swift 大同小异，因为我们这是一本关于 Swift 的书籍，所以就只使用 Swift 来进行介绍了。

首先通过新建菜单的 Framework & Library 创建一个 Cocoa Touch Framework 项目，命名为 HelloKit，然后添加一个 Swift 文件以及随便一些什么内容，比如：

```
public class Hello {  
    public class func sayHello() {  
        print("Hello Kit")  
    }  
}
```

注意我们在这里添加了 public 声明，这是因为我们的目的是在当前 module 之外使用这些代码。将运行目标选择为任一 iOS 模拟器，然后使用 Shift + Cmd + I 进行 Profiling 编译。我们可以在项目的生成的数据文件夹中 (使用 Window 菜单的 Organizer 可以找到对应项目的该文件夹位置) 的 /Build/Products/Release-iphonesimulator 里找到 HelloKit.framework。

如果直接使用 Cmd + B 进行编译的话我们得到的会是一个 Debug 版本的结果，在绝大多数情况下这应该不是我们想要的，除非你是需要用来进行调试。

然后新建一个项目来看看如何使用这个框架吧。建立新的 Xcode 项目，语言当然是选择为 Swift，然后将刚才的 HelloKit.framework 拖到 Xcode 项目中就可以了。我们最好勾选上 Copy items if needed，这样原来的框架的改动就不会影响到我们的项目了。

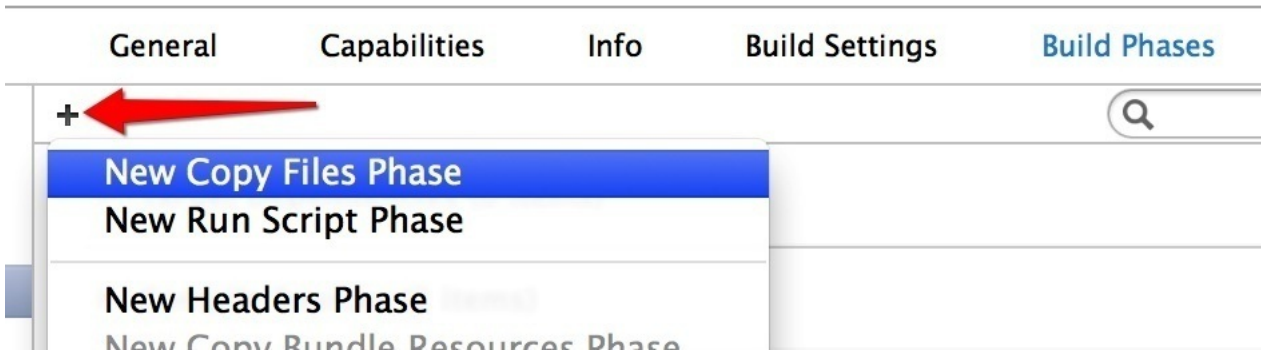
接下来，我们在需要使用这个框架的地方加上对框架的导入和调用。为了简单，我们就在 AppDelegate.swift 的 didFinishLaunching 方法中对 sayHello 进行一次调用：

```
func application(application: UIApplication!,  
    didFinishLaunchingWithOptions launchOptions: NSDictionary!) -> Bool {  
    // Override point for customization after application launch.  
  
    Hello.sayHello()  
  
    return true  
}
```

当然，别忘记在顶部加上 import HelloKit 来导入框架。

和其他的只做链接的添加框架的方式略有不同，最后一步我们还需要在编译的时候将这个框架复制到项目包中。在 Build Phases 选项卡里添加一个 Copy File 的阶段 (如图)，然后将目标设定为 Frameworks，将我们的 HelloKit.framework 添加到新建的阶段里，来指定 IDE 在编译时进行复

制。



我想就这么多了，现在使用模拟器运行这个项目，我们应该可以在控制台中看到我们的输出了：Hello Kit。

但是故事还没有最终结束。我们刚才编译的时候只做了模拟器的版本，如果你尝试一下在 app 项目中将目标切换为真机的话，会发现根本无法编译，这是由于模拟器和实际设备所使用的架构不同导致的。我们需要回到框架项目中，将编译目标切换为 iOS Device，然后再次使用 Shift + Cmd + I 进行编译。这时我们可以在 Release-iphoneros 文件夹下得到真实设备可以使用的框架。最后我们通过 lipo 命令将适用于多个架构的二进制文件进行合并，以得到可以在模拟器和实际设备上通用的二进制文件：

```
lipo -create -output HelloKit \
    Release-iphoneros/HelloKit.framework/HelloKit \
    Release-iphonesimulator/HelloKit.framework/HelloKit
```

然后将得到的包含各架构的新的 HelloKit 文件复制到刚才的模拟器版本的 HelloKit.framework 中(没错其实它是个文件夹)，覆盖原来的版本。最后再将 Release-iphoneros 中的框架文件里的 /Modules/HelloKit.swiftmodule 下的 arm.swiftmodule 和 arm64.swiftmodule 两个文件复制到模拟器版本的对应的文件夹下(这个文件夹下最终应该会有 i386, x86_64, arm 和 arm64 四个版本的 module 文件)。我们现在就得到了一个通吃模拟器和实际设备的框架了，用这个框架替换掉刚才我们复制到 app 项目中的那个，app 应该就可以同时在模拟器和设备上使用这个自制框架了。

再次提醒，本文所述的用 Swift 构建框架项目，然后在其他项目中使用这个框架的做法并不是推荐做法。对于 Objective-C 来说这个做法没有什么太大问题，但是对于 Swift 的框架来说，因为现在 Swift 的解释和运行环境还没有非常稳定，因此在项目中使用非同项目 target 的框架的时候，很有可能项目和框架的 Swift 运行环境有所差异。有时候这会导致不必要的问题和麻烦。

Playground 延时运行

从 WWDC 14 的 Keynote 上 Chris 的演示就能看出 Playground 异常强大，但是从本质来说 Playground 的想法其实非常简单，就是提供一个可以即时编辑的类似 REPL 的环境。

Playground 为我们提供了一个顺序执行的环境，在每次更改其中代码后整个文件会被重新编译，并清空原来的状态并运行。这个行为与测试时的单个测试用例有一些相似，因此有些时候在测试时我们会遇到的问题我们在 Playground 中也会遇到。

其中最基础的一个就是异步代码的执行，比如这样的 `NSTimer` 在默认的 Playground 中是不会执行的：

```
class MyClass {
    @objc func callMe() {
        print("Hi")
    }
}

let object = MyClass()

NSTimer.scheduledTimerWithTimeInterval(1, target: object,
                                       selector: "callMe", userInfo: nil, repeats: true)
```

关于 selector 的使用 和 `@objc` 标记可以分别参见 [Selector](#) 以及 [@objc](#) 和 [dynamic](#)。

在执行完 `NSTimer` 语句之后，整个 Playground 将停止掉，`Hi` 永远不会被打印出来。放心，这种异步的操作没有生效并不是因为你写错了什么，而是 Playground 在执行完了所有语句，然后正常退出了而已。

为了使 Playground 具有延时运行的本领，我们需要引入 Playground 的“扩展包”`XCPlayground` 框架。现在这个框架中包含了几个与 Playground 的行为交互以及控制 Playground 特性的 API，其中就包括使 Playground 能延时执行的黑魔法，`XCPlayground.setExecutionShouldContinueIndefinitely`。

我们只需要在刚才的代码上面加上：

```
import XCPlayground
XCPlayground.setExecutionShouldContinueIndefinitely(true)
```

就可以看到 `Hi` 以每秒一次的频率被打印出来了。

在实际使用和开发中，我们最经常面临的异步需求可能就是网络请求了，如果我们想要在 Playground 里验证某个 API 是否正确工作的话，使用 `XCPlayground` 的这个方法开启延时执行也是必要的：


```
let url = NSURL(string: "http://httpbin.org/get")!

let getTask = NSURLSession.sharedSession().dataTaskWithURL(url) {
    (data, response, error) -> Void in
    let dictionary = try! NSJSONSerialization.JSONObjectWithData(data!, options: [])

    print(dictionary)
}

getTask.resume()
```

延时运行也是有限度的。如果你足够有耐心，会发现在第一个例子中的 `NSTimer` 每秒打印一次的 Hi 的计数最终会停留在 30 次。这是因为即使在开启了持续执行的情况下，Playground 也不会永远运行下去，默认情况下它会在顶层代码最后一句运行后 30 秒的时候停止执行。这个时间长度对于绝大多数的需求场景来说都是足够的了，但是如果你想改变这个时间的话，可以通过 `Alt + Cmd + 回车` 来打开辅助编辑器。在这里你会看到控制台输出和时间轴，将右下角的 30 改成你想要的数字，就可以对延时运行的最长时间进行设定了。

之前的像是 [GCD 和延时调用](#) 这样的章节中也涉及到了延时运行，你可以将这里的技巧应用到之前的示例代码上，这样你就可以在 Playground 中得到正确的结果了。

Playground 可视化

在程序界，很多小伙伴都会对研究排序算法情有独钟，并且试图将排序执行的过程可视化，以便让大家更清晰直观地了解算法步骤。有人把可视化排序做得很[正统明了](#)，也有人把它做到了[艺术层次](#)。

想在 Cocoa 中做一个可视化的排序算法演示可不是一件容易的事情，很可能你会需要一套绘制图形的框架，并且考虑如何在屏幕上呈现每一步的过程。但是在 Playground 中事情就变得简单多了：我们可以使用 `XCPlayground` 框架的 `XCPCaptureValue` 方法来将一组数据轻而易举地绘制到时间轴上，从而让我们能看到每一步的结果。这不仅对我们直观且及时地了解算法内部的变化很有帮助，也会是教学或者演示时候的神兵利器。

`XCPCaptureValue` 的使用方法很简单，在 `import XCPlayground` 导入框架后，可以找到该方法的定义：

```
func XCPCaptureValue<T>(identifier: String, value: T)
```

我们可以多次调用该方法来做图，相同的 `identifier` 的数据将会出现在同一张图上，而 `value` 将根据输入的次序进行排列。举一个完整的例子来说明会比较快，比如下面的代码实现了简单的冒泡排序，我们在每一轮排序完成后使用 `plot` 方法将当前的数组状态用 `XCPCaptureValue` 的方式进行了输出。通过在时间轴 (通过“`Alt+Cmd+回车`”打开 Assistant Editor) 的输出图，我们就可以非常清楚地了解到整个算法的执行过程了。

```
import XCPlayground

var arr = [14, 11, 20, 1, 3, 9, 4, 15, 6, 19,
           2, 8, 7, 17, 12, 5, 10, 13, 18, 16]

func plot<T>(title: String, array: [T]) {
    for value in array {
        XCPCaptureValue(title, value: value)
    }
}

plot("起始", array: arr)

func swap(inout x: Int, inout y: Int) {
    (x, y) = (y, x)
}

func bubbleSort<T: Comparable>(inout input: [T]) {
    for var i = input.count; i > 1; i-- {
        var didSwap = false
        for var j = 0; j < i - 1; j++ {
            if input[j] > input[j + 1] {
                didSwap = true
                swap(&input[j], &input[j + 1])
            }
        }
        if !didSwap {

```

```

        break
    }
    plot("第 \((input.count - (i - 1))\) 次迭代", array: input)
}
plot("结果", array: input)
}

bubbleSort(&arr)

```

因为 `xCPCaptureValue` 的数据输入是任意类型的，所以不论是传什么进去都是可以表示的。它们将以 `QuickLook` 预览的方式被表现出来，一些像 `UIImage`，`UIColor` 或者 `UIBezierPath` 这样的类型已经实现了 `QuickLook`。当然对于那些没有实现快速预览的 `NSObject` 子类，也可以通过重写

```

func debugQuickLookObject() -> AnyObject?

```

来提供一个预览输出。在上面的冒泡排序方法中，我们可以接收任意满足 `Comparable` 的数组，而绘图方法也可以接受任意类型的输入。作为练习，可以试试看把 `arr` 的全部数字都换成一些随机的字符串看看时间轴的输出是什么样子吧。

Playground 与项目协作

我们提到过使用 Framework 的方式来组织和分离代码。除了能够得到更清晰的架构层次和方便的代码重用外，我们还能通过这个方式得到一个额外的好处，那就是在项目的 Playground 中使用这些代码。

一般来说，最主要的使用 Playground 的方式可能是建立单独的 Playground，然后在其中实验一些小的代码片段和 API。但是在实际开发中，我们面临的更多的是针对具体项目的问题。如果我们想在单独的 Playground 中使用我们已经写的类或者方法的话，我们只能将这些类和方法的代码复制到 Playground 中，然后再进行依赖于它们的实验。这样的做法非常麻烦：迅速地确定所有代码的依赖关系对于人来说本身就不是一件容易的事情，很可能你需要多次复制和检查才能最终建立起一套可用的环境；另一个问题是你需要时刻记住，Ctrl + C 和 V 在绝大多数情况下都是恶魔，想想如果你的项目代码之后发生了改变，你要怎么样才能让 Playground 里的内容和它们同步呢？从头开始再来一遍？显然这么做会是个悲剧。

Playground 其实是可以用在项目里的，通过配置，我们是可以做到让 Playground 使用项目中已有的代码的。直接说结论的话，我们需要满足以下的一些条件：

1. Playground 必须加入到项目之中，单独的 Playground 是不能使用项目中的已有代码的；

最简单的方式是在项目中使用 File -> New -> File... 然后在里面选择 Playground。注意不要直接选择 File -> New -> Playground...，否则的话你还需要将新建的 Playground 拖到项目中来。

2. 想要使用的代码必须是通过 Cocoa (Touch) Framework 以一个单独的 target 的方式进行组织的；
3. 编译结果的位置需要保持默认位置，即在 Xcode 设置中的 Locations 里 Derived Data 保持默认值；
4. 如果是 iOS 应用，这个框架必须已经针对 iPhone 5s Simulator 这样的 64 位的模拟器作为目标进行过编译；

iOS 的 Playground 其实是运行在 64 位模拟器上的，因此为了能找到对应的符号和执行文件，框架代码的位置和编译架构都是有所要求的。

在满足这些条件后，你就可以在 Playground 中通过 `import` 你的框架 module 名字来导入代码，然后进行使用了。

数学和数字

Darwin 里的 `math.h` 定义了很多和数学相关的内容，它在 Swift 中也被进行了 `module` 映射，因此在 Swift 中我们是可以直接使用的。有了这个保证，我们就不需要担心在进行数学计算的时候会和标准有什么差距。比如，我们可以轻易地使用圆周率来计算周长，也可以使用各种三角函数来完成需要的屏幕位置计算等等：

```
func circlePerimeter(radius: Double) -> Double {
    return 2 * M_PI * radius
}

func yPosition(dy: Double, angle: Double) -> Double {
    return dy * tan(angle)
}
```

Swift 除了导入了 `math.h` 的内容以外，也在标准库中对极限情况的数字做了一些约定，比如我们可以使用 `Int.max` 和 `Int.min` 来取得对应平台的 `Int` 的最大和最小值。另外在 `Double` 中，我们还有两个很特殊的值，`infinity` 和 `NaN`。

`infinity` 代表无穷，它是类似 `1.0 / 0.0` 这样的数学表达式的结果，代表数学意义上的无限大。在这里我们强调了数学意义，是因为受限于计算机系统，其实是没有真正意义的无穷大的，毕竟这是我们讨论的平台。一个 64 位的系统中，Swift 的 `Double` 能代表的最大的数字大约是 `1.797693134862315e+308`，而超过这个数字的 `Double` 虽然在数学意义上并不是无穷大，但是也会在判断的时候被认为是无穷：

```
1.797693134862315e+308 < Double.infinity // true
1.797693134862316e+308 < Double.infinity // false
```

当然一般来说和无穷大比较大小是没有意义的，虽然在绝大多数情况下我们不会在这个上面栽跟头，但是谁又知道会不会真的遇到这样的情况呢？

另一个有趣的東西是 `NaN`，它是“Not a Number”的简写，可以用来表示某些未被定义的或者出现了错误的运算，比如下面的操作都会产生 `NaN`：

```
let a = 0.0 / 0.0
let b = sqrt(-1.0)
let c = 0.0 * Double.infinity
```

与 `NaN` 再进行的运算结果也都将是 `NaN`。Swift 的 `Double` 中也为我们提供了直接获取一个 `NaN` 的方法，我们可以通过使用 `Double.NaN` 来直接获取一个 `NaN`。在某些边界条件下，我们可能会希望判断一个数值是不是 `NaN`。和其他数字 (包括无穷大) 相比，`NaN` 在这点上非常特殊。你不能将 `NaN` 用来做相等判断或者大小比较，因为它本身就不是数字，因此这类比较就没有意义了。比如

对于一个理论上的恒等式 `num == num`，在 `NaN` 的情况下就有所不同：

```
let num = Double.NaN
if num == num {
    print("Num is \ \(num)")
} else {
    print("NaN")
}

// 输出:
// NaN
```

用判定自身是否与自己相等的方式就可以判定一个量是不是 `NaN` 了。当然，一个更加容易读懂和简洁的方式使用 `Double` 的 `isNaN` 或者 `Darwin` 中的 `isnan` 来判断：

```
let num = Double.NaN
if num.isNaN {
    print("NaN")
}

if isnan(num) {
    print("NaN")
}

// 输出:
// NaN
// NaN
```

JSON

如果 app 需要有网络功能并且有一个后端服务器处理和返回数据的话，那么现在基本上要和 JSON 打交道是没跑儿了的。在 Swift 里处理 JSON 其实是一件挺棘手的事情，因为 Swift 对于类型的要求非常严格，所以在解析完 JSON 之后想要从结果的 `AnyObject` 中获取某个键值是一件非常麻烦的事情。举个例子，我们使用 `NSJSONSerialization` 解析完一个 JSON 字符串后，得到的是 `AnyObject?`：

```
// jsonString
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

```
let jsonString = //...

let json: AnyObject = try! NSJSONSerialization.JSONObjectWithData(
    jsonString.dataUsingEncoding(NSUTF8StringEncoding, allowLossyConversion: true)!,
    options: [])
```

我们如果想要访问 menu 里的 popup 中 第一个 menuitem 的 value 值的话，最正规的情况下，需要写这样的代码：

```
if let jsonDic = json as? NSDictionary {
    if let menu = jsonDic["menu"] as? [String: AnyObject] {
        if let popup = menu["popup"] {
            if let popupDic = popup as? [String: AnyObject] {
                if let menuItems = popupDic["menuitem"] {
                    if let menuItemsArr = menuItems as? [AnyObject] {
                        if let item0 = menuItemsArr[0]
                           as? [String: AnyObject] {
                            if let value: AnyObject = item0["value"] {
                                print(value)
                            }
                        }
                    }
                }
            }
        }
    }
}
// 输出 New
```

什么？你难道把这段代码看完了？我都不忍心写下去了...如果你真的想要坚持这么做的话，我只能说呵呵，并且祝你好运了。

当然，在 Swift 1.2 中，我们可以在同一个 `if let` 语句中进行 `unwrap`，这样会比原来稍好一些，但是依旧十分麻烦：

```
if let jsonDic = json as? NSDictionary,
    menu = jsonDic["menu"] as? [String: AnyObject],
    popup = menu["popup"],
    popupDic = popup as? [String: AnyObject],
    menuItems = popupDic["menuitem"],
    menuItemsArr = menuItems as? [AnyObject],
    item0 = menuItemsArr[0] as? [String: AnyObject],
    value = item0["value"]
{
    print(value)
}
```

那么，我们应该怎么做呢？在上面的代码中，最大的问题在于我们为了保证类型的正确性，做了太多的转换和判断。我们并没有利用一个有效的 JSON 容器总应该是字典或者数组这个有用的特性，而导致每次使用下标取得的值都是需要转换的 `AnyObject`。如果我们能够重载下标的话，就可以通过下标的取值配合 `Array` 和 `Dictionary` 的 `Optional Binding` 来简单地在 JSON 中取值。鉴于篇幅，我们在这里不给出具体的实现。感兴趣的读者可以移步看看 [json-swift](#) 或者 [SwiftyJSON](#) 这样的项目，它就使用了重载下标访问的方式简化了 JSON 操作。使用这个工具，上面的访问可以简化为下面的类型安全的样子：

```
// 使用 SwiftyJSON
if let value = JSON(json)["menu"]["popup"]["menuitem"][0]["value"].string {
    print(value)
}
```

这样就简单多了。

NSNull

NSNull 出场最多的时候就是在 JSON 解析了。

在 Objective-C 中，因为 NSDictionary 和 NSArray 只能存储对象，对于像 JSON 中可能存在的 null 值，NSDictionary 和 NSArray 中就只能用 NSNull 对象来表示。Objective-C 中的 nil 实在是太方便了，我们向 nil 发送任何消息时都将返回默认值，因此很多时候我们过于依赖这个特性，而不再去进行检查就直接使用对象。大部分时候这么做没有问题，但是在处理 JSON 时，NSNull 却无法使用像 nil 那样的对所有方法都响应的特性。而又因为 Objective-C 是没有强制的类型检查的，我们可以任意向一个对象发送任何消息，这就导致如果 JSON 对象中存在 null 时 (不论这是有意为之还是服务器方面出现了某种问题) 的话，对其映射为的 NSNull 直接发送消息时，app 将发生崩溃。相信有过一定和后端协作的开发经验的读者，可能都遇到过这样的问题：

```
NSInteger voteCount = [jsonDic objectForKey:@"voteCount"] integerValue];
// 如果在 JSON 中 voteCount 对应的是 null 的话
// [NSNull intValue]: unrecognized selector sent to instance 崩溃
```

在 Objective-C 中，我们一般通过严密的判断来解决这个问题：即在每次发送消息的时候都进行类型检查，以确保将要接收消息的对象不是 NSNull 的对象。另一种方法是添加 NSNull 的 category，让它响应各种常见的方法 (比如 integerValue 等)，并返回默认值。两种方式都不是非常完美，前一种过于麻烦，后一种难免有疏漏。

而在 Swift 中，这个问题被语言的特性彻底解决了。因为 Swift 所强调的就是类型安全，无论怎么说都需要一层转换。因此除非我们故意犯二不去将 AnyObject 转换为我们需要的类型，否则我们绝对不会错误地向一个 NSNull 发送消息。NSNull 会默默地被通过 Optional Binding 被转换为 nil，从而避免被执行：

```
// 假设 jsonValue 是从一个 JSON 中取出的 NSNull
let jsonValue: AnyObject = NSNull()

if let string = jsonValue as? String {
    print(string.hasPrefix("a"))
} else {
    print("不能解析")
}

// 输出：
// 不能解析
```


文档注释

文档的重要性是毋庸置疑的，使用别人的代码时我们一般都会去查阅文档了解 API 的使用方法，与别人合作时我们也都要为各自的代码负责，一个最简单的方式就是撰写易读可用的文档。关于文档的种种好处就不再赘述了，在这一节中我们着重看看怎么为 Swift 的代码编写文档。

对于程序设计的文档，业界的标准做法都是自动生成。一般我们会将文档嵌入地以某种规范的格式以注释的形式写在实际代码的上方，这样文档的自动生成器就可以扫描源代码并读取这些符合格式的注释，最后生成漂亮的文档了。对于 Objective-C 来说，这方面的自动生成工具有 Apple 标准的 [HeaderDoc](#)，以及第三方的 [appledoc](#) 或者 [Doxygen](#) 等。

从 Xcode 5 开始，IDE 默认集成了提取文档注释并显示为 Quick Help 的功能，从那以后，能被 Xcode 识别的文档注释就成为了事实上的行业标准。在 Objective-C 时代，传统的 [Javadoc](#) 格式的注释是被接受的，而到了 Swift，Apple 采用了一套新的文档注释方法，对于一个简单的方法，我们的文档注释看起来应该是这样的：

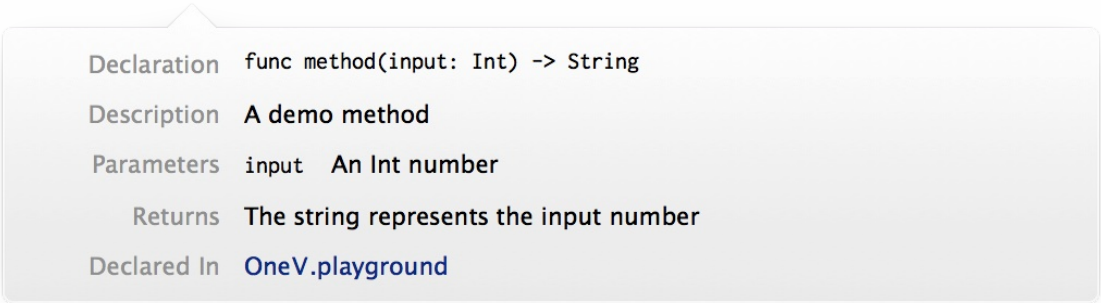
```
/**
 A demo method

 - parameter input: An Int number

 - returns: The string represents the input number
 */
func method(input: Int) -> String {
    return String(input)
}
```

在文档注释的块中 (在这里是被 `/**...*/` 包围的注释)，我们需要使用 `- parameter` 紧接输入参数名的形式来表达对输入参数的说明。如果有多个参数的情况下，我们会需要对应着写多组 `- parameter` 语句。如果返回值不是 `Void` 的话，我们还需要写 `- returns:` 来对返回进行说明。

这时，我们如果使用 `Alt + 单击` 的方式点选 `method` 的话，就可以看到由 Xcode 格式化后的 Quick Help 对话框：



The Quick Help dialog box displays the following information for the `method` function:

- Declaration** `func method(input: Int) -> String`
- Description** `A demo method`
- Parameters** `input` `An Int number`
- Returns** `The string represents the input number`
- Declared In** `OneV.playground`

在调用这个方法时，同样的提示也会出现，非常方便。

对于像属性这样的简单的声明，我们直接使用 `///` 就可以了：

```
struct Person {  
    /// name of the person  
    var name: String  
}
```

另外还有一些其他的关键字，你可以在[这里](#)找到一个相对完整的列表。

现在暂时除了 Xcode 自身的渲染之外，其他传统的文档自动生成工具还不能很好地读取 Swift 的文档注释。不过相信很快像 HeaderDoc 或者 appledoc 这样的工具就会进行更新并提供支持，这并没有太大的实现难度。另外，有一个叫做 jazzy 的新项目在这方面已经做出了一些成果。现在 jazzy 已经被 CocoaPods 用作提取 Swift 文档的工具，成为事实上的标准了。

最后，如果你觉得在 Xcode 中手写 `-parameter` 或者 `-returns` 这样的东西非常浪费时间的话，可以尝试使用一款叫做 VVDocumenter 的 Xcode 插件，它能够帮助你快速并且自动地生成符合格式的文档注释模板，你需要做的只是填上你需要的描述。

作为利益相关的说明，VVDocumenter 的开发者就是我本人，欢迎大家使用。:)

性能考虑

在 WWDC 14 的 Keynote 上，Swift 相比于其他语言的速度优势被特别进行了强调。但是这种速度优势是有条件的，虽然由于编译器的进步我们可能可以在不了解语言特性的时候随便写也能得到性能上的改善，但是如果能够稍微理解背后的机制的话，我们就能投“编译器所好”，写出更高效的代码。

相比于 Objective-C，Swift 最大的改变就在于方法调用上的优化。在 Objective-C 中，所有的对于 NSObject 的方法调用在编译时会被转为 objc_msgSend 方法。这个方法运用 Objective-C 的运行时特性，使用派发的方式在运行时对方法进行查找。因为 Objective-C 的类型并不是编译时确定的，我们在代码中所写的类型不过只是向编译器的一种“建议”，不论对于怎样的方法，这种查找的代价基本都是同样的。

这个过程的等效的表述可能类似这样 (注意这只是一种表述，与实际的代码和工作方式无关)：

```
methodToCall = findMethodInClass(class, selector);  
// 这个查找一般需要遍历类的方法表，需要花费一定时间  
  
methodToCall(); // 调用
```

Objective-C 运行时十分高效，相比于 I/O 这样的操作来说，单次的方法派发和查找并不会花费太多的时间，但实事求是地说，这确实也是 Objective-C 性能上可以改进的地方，这种改善在短时间内大量方法调用时会比较明显。

Swift 因为使用了更安全和严格的类型，如果我们在编写代码中指明了某个实际的类型的话 (注意，需要的是实际具体的类型，而不是像 Any 这样的抽象的接口)，我们就可以向编译器保证在运行时该对象一定属于被声明的类型。这对编译器进行代码优化来说是非常有帮助的，因为有了更多更明确的类型信息，编译器就可以在类型中处理多态时建立虚函数表 (vtable)，这是一个带有索引的保存了方法所在位置的数组。在方法调用时，与原来动态派发和查找方法不同，现在只需要通过索引就可以直接拿到方法并进行调用了，这是实实在在的性能提升。这个过程大概相当于：

```
methodToCall = class.vtable[methodIndex]  
// 直接使用 methodIndex 获取实现  
  
methodToCall(); // 调用
```

更进一步，在确定的情况下，编译器对 Swift 的优化甚至可以做到将某些方法调用优化为 inline 的形式。比如在某个方法被 final 标记时，由于不存在被重写的可能，vtable 中该方法的实现就完全固定了。对于这样的方法，编译器在合适的情况下可以在生成代码的阶段就将方法内容提取到调用的地方，从而完全避免调用。

通过 Benchmark 我们可以看出，在一些基本的算法上，Swift 的速度是要远胜过 Objective-C 的，

而就算相较于世界上无可匹敌的 C，也[没有逊色太多](#)。

所以对于性能方面，我们应该注意的地方就很明显了。如果遇到性能敏感和关键的代码部分，我们最好避免使用 Objective-C 和 NSObject 的子类。在以前我们可能会选择使用混编一些 C 或者 C++ 代码来处理这些关键部分，而现在我们多了 Swift 这个选项。相比起 C 或者 C++，Swift 的语言特性上要先进得多，而使用 Swift 类型和标准库进行编码和构建的难度，比起使用 C 或 C++ 来要简单太多。另外，即使不是性能关键部分，我们也应该尽量考虑在必要时减少使用 NSObject 和它的子类。如果没有动态特性的需求的话，保持在 Swift 基本类型中会让我们得到更多的性能提升。

Log 输出

Log 输出是程序开发中很重要的组成部分，虽然它并不是直接的业务代码，但是却可以忠实地反映我们的程序是如何工作的，以及记录程序运行的过程中发生了什么。

在 Swift 中，最简单的输出方法就是使用 `print`，在我们关心的地方输出字符串和值。但是这并不够，试想一下当程序变得非常复杂的时候，我们可能会输出很多内容，而想在其中寻找到我们希望的输出其实并不容易。我们往往需要更好更精确的输出，这包括输出这个 `log` 的文件，调用的行号以及所处的方法名字等等。

我们当然可以在 `print` 的时候将当前的文件名字和那些必要的信息作为参数同我们的消息一起进行打印：

```
// Test.swift
func method() {
    //...
    print("文件名:Test.swift, 方法名:method, 这是一条输出")
    //...
}
```

但是这显然非常麻烦，每次输入文件名和方法名不说，随着代码的改变，这些 Log 的位置也可能发生改变，这时我们可能还需要不断地去维护这些输出，代价实在太大了。

在 Swift 中，编译器为我们准备了几个很有用的编译符号，用来处理类似这样的需求，它们分别是：

符号	类型	描述
FILE	String	包含这个符号的文件的完整路径
LINE	Int	符号出现处的行号
COLUMN	Int	符号出现处的列
FUNCTION	String	包含这个符号的方法名字

因此，我们可以通过使用这些符号来写一个好一些的 Log 输出方法：

```
func printLog<T>(message: T,
                 file: String = __FILE__,
                 method: String = __FUNCTION__,
                 line: Int = __LINE__)
{
    print("\(file as NSString).lastPathComponent)[\line], \method): \message)")
}
```

这样，在进行 log 的时候我们只需要使用这个方法就能完成文件名，行号以及方法名的输出了。最棒的是，我们不再需要对这样的输出进行维护，无论在哪里它都能正确地输出各个参数：

```
// Test.swift
func method() {
    //...
    printLog("这是一条输出")
    //...
}

// 输出:
// Test.swift[62], method(): 这是一条输出
```

另外，对于 log 输出更多地其实是用在程序开发和调试的过程中的，过多的输出有可能对运行的性能造成影响。在 **Release** 版本中关闭掉向控制台的输出也是软件开发中一种常见的做法。如果我们在开发中就注意使用了统一的 log 输出的话，这就变得非常简单了。使用[条件编译](#)的方法，我们可以添加条件，并设置合适的编译配置，使 `printLog` 的内容在 **Release** 时被去掉，从而成为一个空方法：

```
func printLog<T>(message: T,
                file: String = __FILE__,
                method: String = __FUNCTION__,
                line: Int = __LINE__)
{
    #if DEBUG
    print("\(file as NSString).lastPathComponent)[\(line)], \(method): \(message)")
    #endif
}
```

新版本的 LLVM 编译器在遇到这个空方法时，甚至会直接将这个方法整个去掉，完全不去调用它，从而实现零成本。

溢出

对于 Mac 开发，我们早已步入了 64 位时代，而对 iOS 来说，64 位的乐章才刚刚开始。在今后一段时间内，我们都需要面临同时为 32 位和 64 位的设备进行开发的局面。由于这个条件所导致的最直接的一个结果就是数字类型的区别。

最简单的例子，在 Swift 中我们一般简单地使用 `Int` 来表示整数，在 iPhone 5 和以下的设备中，这个类型其实等同于 `Int32`，而在 64 位设备中表示的是 `Int64`（这点和 Objective-C 中的 `NSInteger` 表现是完全一样的，事实上，在 Swift 中 `NSInteger` 只是一个 `Int` 的 [typealias](#)。这就意味着，我们在开发的时候必须考虑同样的代码在不同平台上的表现差异，比如下面的这段计算在 32 位设备上和 64 位设备上的表现就完全不同：

```
class MyClass {
    var a: Int = 1
    func method() {
        a = a * 100000
        a = a * 100000
        a = a * 100000
        print(a)
    }
}

MyClass().method()

// 64 位环境 (iPhone 5s 及以上)
// 1,000,000,000,000,000

// 32 位环境 (iPhone 5c 及以下)
// 崩溃
```

因为 32 位的 `Int` 的最大值为 2,147,483,647，这个方法的计算已经超过了 `Int32` 的最大值。和其他一些编程语言的处理不同的是，Swift 在溢出的时候选择了让程序直接崩溃而不是截掉超出的部分，这也是一种安全性的表现。

另外，编译器其实已经足够智能，可以帮助我们在编译的时候就发现某些必然的错误。比如：

```
func method() {
    var max = Int.max
    max = max + 1
}
```

这种常量运算在编译时就进行了，发现计算溢出后编译无法通过。

在存在溢出可能性的地方，第一选择当然是使用更大空间的类型来表示，比如将原来的 `Int32` 显式地声明为 `Int64`。如果 64 位整数还无法满足需求的话，我们也可以考虑使用两个 `Int64` 来软件实现 `Int128`（据我所知现在还没有面向消费领域的 128 位的电子设备）的行为。

最后，如果我们想要其他编程语言那样的对溢出处理温柔一些，不是让程序崩溃，而是简单地从高位截断的话，可以使用溢出处理的运算符，在 Swift 中，我们可以使用以下这五个带有 `&` 的运算符，这样 Swift 就会忽略掉溢出的错误：

- 溢出加法 (`&+`)
- 溢出减法 (`&-`)
- 溢出乘法 (`&*`)
- 溢出除法 (`&/`)
- 溢出求模 (`&+`)

这样处理的结果：

```
var max = Int.max
max = max &+ 1

// 64 位系统下
// max = -9,223,372,036,854,775,808
```


宏定义 `define`

Swift 中没有宏定义了。

宏定义确实是一个让人又爱又恨的特性，合理利用的话，可以让我们写出很多简洁漂亮的代码，但是同时，不可否认的是宏定义无法受益于 IDE 工具，难以重构和维护，很可能隐藏很多 bug，这对于开发其实并不是一件好事。

Swift 中将宏定义彻底从语言中拿掉了，并且 Apple 给了我们一些替代的建议：

- 使用合适作用范围的 `let` 或者 `get` 属性来替代原来的宏定义值，例如很多 Darwin 中的 C 的 `define` 值就是这么做的：

```
var M_PI: Double { get } /* pi */
```

- 对于宏定义的方法，类似地在同样作用域写为 Swift 方法。一个最典型的例子是 `NSLocalizedString` 的转变：

```
// objc
#define NSLocalizedString(key, comment) \
[[NSBundle mainBundle] localizedStringForKey:(key) value:@"" table:nil]
```

```
// Swift
func NSLocalizedString(
    key: String,
    tableName: String? = default,
    bundle: NSBundle = default,
    value: String = default,
    comment: String) -> String
```

- 随着 `#define` 的消失，像 `#ifdef` 这样通过宏定义是否存在来进行条件判断并决定某些代码是否参与编译的方式也消失了。但是我们仍然可以使用 `#if` 并配合编译的配置来完成条件编译，具体的方法可以参看[条件编译](#)一节的内容。

`define` 在编译时实际做的事情类似查找替换，因此往往可以用来做一些很强大的事情，比如只替换掉某部分内容。举个例子，如果 Swift 中有 `define` 的话，我们可能可以写出这样的宏定义：

```
#define PUBLIC_CLASS_START(x) public class x {
#define PUBLIC_CLASS_END }
```

然后在文件中这样使用：

```
PUBLIC_CLASS_START(MyClass)

var myVar: Int = 1

PUBLIC_CLASS_END
```

但事实上这在 **Swift** 中是无法做到的。

虽然这只是一个没什么实际用处的例子，但是这展现了我们完全改变代码表现结构的可能性。在自动代码生成或者统一配置修改时有的情况下会很好用。而现在暂时在 **Swift** 中无法对应这样的用法，所以在 **Swift** 中可能短期内我们可能很难看到类似 [Kiwi](#) 这样的严重依赖宏定义来改变语法结构的有趣的项目了。

属性访问控制

Swift 中由低至高提供了 `private`，`internal` 和 `public` 三种访问控制的权限。默认的 `internal` 在绝大部分时候是适用的，另外由于它是 Swift 中的默认的控制级，因此它也是最为方便的。

但是对于一个严格的项目来说，精确的最小化访问控制级别对于代码的维护来说还是相当重要的。对于方法来说比较直接，我们想让同一 `module` (或者说是 `target`) 中的其他代码访问的话，保持默认的 `internal` 就可以了。如果我们在为其他开发者开发库的话，可能会希望用一些 `public`，因为在 `target` 外只能调用到 `public` 的代码。而那些只希望在本文件内访问的方法，我们应该用 `private` 加以限制，以防止暴露给项目的其他部分。要特别说明的一点是，Swift 中的 `private` 和其他大部分语言不太一样，它的限制范围是按文件，而不是按照类型来的。就是说，即使是两个毫不相关的类型，只要被写在同一个文件中，那么这个文件里的 `private` 就可以被相互访问到。

以上是方法和类型的访问控制的情况。而对于属性来说，访问控制还多了一层需要注意的地方。在类型中的属性默认情况下：

```
class MyClass {  
    var name: String?  
}
```

因为没有任何的修饰，所以我们可以同一 `module` 中随意地读取或者设置这个变量。从类型外部读取一个实例成员变量是很普通的需求，而对其进行设定的话就需要小心一些了。当然实际我们在构建一个类时是会有需要设置的情况的，一般来说会是在这个类型外的地方，对这个类型对象的某些特性进行配置。

对于那些我们只希望在当前文件中使用的属性来说，当然我们可以在声明前面加上 `private` 使其变为私有：

```
class MyClass {  
    private var name: String?  
}
```

但是在开发中所面临的更多的情况是我们希望在类型之外也能够读取到这个类型，同时为了保证类型的封装和安全，只能在类型内部对其进行改变和设置。这时，我们可以通过下面的写法将读取和设置的控制权限分开：

```
class MyClass {  
    private(set) var name: String?  
}
```

因为 `set` 被限制为了 `private`，我们就可以保证 `name` 只会在当前文件被更改。这为之后更改或者调试代码提供了很好的范围控制，可以让我们确定只需要在当前文件中来寻找问题。

这种写法没有对读取做限制，相当于使用了默认的 `internal` 权限。如果我们希望在别的 `module` 中也能访问这个属性，同时又保持只在当前文件可以设置的话，我们需要将 `get` 的访问权限提高为 `public`。属性的访问控制可以通过两次的访问权限指定来实现，具体来说，将刚才的声明变为：

```
public class MyClass {  
    public private(set) var name: String?  
}
```

这时我们就可以在 `module` 之外也访问到 `MyClass` 的 `name` 了。

我们在 `MyClass` 前面也添加的 `public`，这是编译器所要求的。因为如果只为 `name` 的 `get` 添加 `public` 而不管 `MyClass` 的话，`module` 外就连 `MyClass` 都访问不到了，属性的访问控制级别也就没有任何意义了。

Swift 中的测试

在软件开发中，测试的重要性不言而喻。Xcode 中集成了 XCTest 作为测试框架，Swift 代码的测试默认也使用这个框架进行。

关于 XCTest 的使用方法，比如像 `setUp`，`tearDown` 以及 `testxxx` 等在 Swift 下和以前也并没有什么不同，作为一本介绍 tip 的书籍，我不打算在此重复这些。如果对测试的理论基础和实践方法感兴趣的话，不妨看看 Objective-C 中国上相关的[话题文章](#)。

XCTest 中测试和待测试的 app 是分别独立存在于两个不同的 target 里的。这在 Swift 2.0 之前使测试 Swift 代码时面临了由访问权限带来的巨大困境。在 Objective-C 时代，测试的 target 通过依赖应用 target 并导入头文件来获取 app 的 API 并对其进行测试。而在 Swift 中因为 module 模块的管理方法和访问控制权限的设计，使得这个过程出现了问题：一般对于 app，我们都不会将方法标记为 `public`，而会遵循访问权限最小的原则，使用默认的 `internal` 或者是 `private`。对于有些 `internal` 的方法，其实我们是需要去进行测试的。但是由于测试的 target 和 app 的 target 是不同的，因此在测试中导入 app 的 module 后我们是访问不到那些默认 `internal` 的待测试方法的，这就使得测试变得不可能了。

如果我们正在开发的是一个类库的话，为了别人能导入和使用我们的库，我们需要把对外的方法和成员都标记成 `public`，这样我们就能直接在测试 target 中导入类库 target 并访问到待测试 API 了：

```
// 位于框架 target 的业务代码
public func methodToTest() {

}

// 测试
import MyFramework

//...
func testMethodToTest() {

    // 配置测试

    someObj.methodToTest()

    // 断言结果
}
```

对于类库来说，这种做法是没什么问题的 -- 那些被标记为 `public` 的东西恰好就是需要被测试的代码接口。但是对于 app 开发时的测试来说，我们需要尽可能地控制访问权限：我们没有理由为一些理论上不存在外部调用可能的代码赋予 `public` 这样高级的权限，这违背了最小权限的设计原则。对 app 的测试在 Swift 1.x 的时代中一直是一个很麻烦的问题。而在 Swift 2.0 中，Apple 为 app 的测试开了“后门”。现在我们可以通过在测试代码中导入 app 的 target 时，在之前追加 `@testable`，就可以访问到 app target 中 `internal` 的内容了。

```
// 位于 app target 的业务代码
func methodToTest() {

}

// 测试
@testable import MyApp

//...
func testMethodToTest() {

    // 配置测试

    someObj.methodToTest()

    // 断言结果
}
```

Core Data

相信大多数开发者第一次接触到 Objective-C 的 `@dynamic` 都是在和 Core Data 打交道的时候。Objective-C 中的 `@dynamic` 和 Swift 中的 `dynamic` 关键字完全是两回事。在 Objective-C 中，如果我们将某个属性实现为 `@dynamic` 的话，就意味着告诉编译器我们不会在编译时就确定这个属性的行为实现，因此不需要在编译期间对这个属性的 `getter` 或/及 `setter` 做检查和关心。这是我们向编译器做出的庄严承诺，表示我们将在运行时来提供这个属性的存取方法 (当然相应地，如果在运行时你没有履行这个承诺的话，应用就会挂给你看)。

所有的 Core Data Model 类都是 `NSManagedObject` 的子类，它为我们实现了一整套的机制，可以利用我们定义的 Core Data 数据图和关系在运行时动态生成合适的 `getter` 和 `setter` 方法。在绝大多数情况下，我们只需要使用 Xcode 的工具自动生成 `NSManagedObject` 的子类并使用就行了。在 Objective-C 中一个典型的 `NSManagedObject` 子类的样子是这样的：

```
// MyModel.h
@interface MyModel : NSManagedObject

@property (nonatomic, copy) NSString * title;

@end

// MyModel.m
#import "MyModel.h"
@implementation MyModel

@dynamic title;

@end
```

很遗憾，Swift 里是没有 `@dynamic` 关键字的，因为 Swift 并不保证一切都走动态派发，因此从语言层面上提供这种动态转发的语法也并没有太大意义。在 Swift 中严格来说是没有原来的 `@dynamic` 的完整的替代品的，但是如果我们将范围限定在 Core Data 的话就有所不同。

Core Data 是 Cocoa 的一个重要组成部分，也是非常依赖 `@dynamic` 特性的部分。Apple 在 Swift 中专门为 Core Data 加入了一个特殊的标注来处理动态代码，那就是 `@NSManaged`。我们只需要在 `NSManagedObject` 的子类的成员的字段上加上 `@NSManaged` 就可以了：

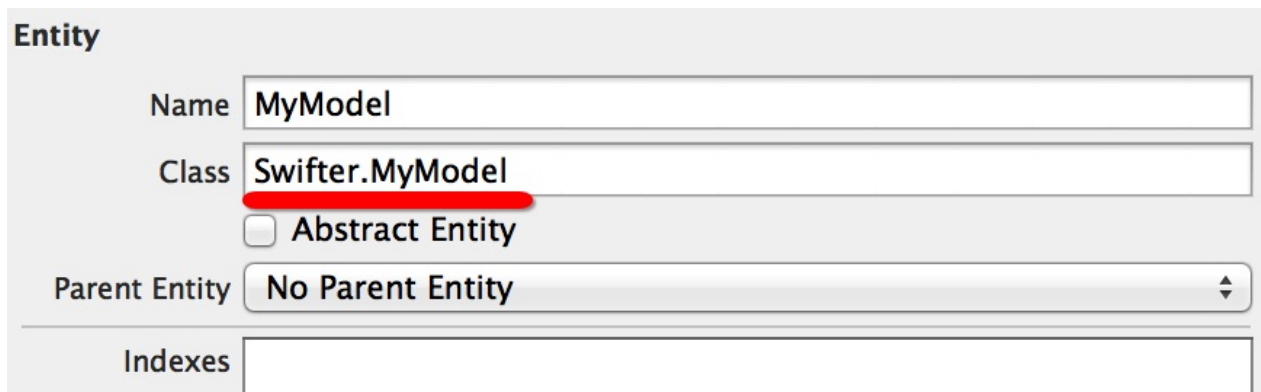
```
class MyModel: NSManagedObject {

    @NSManaged var title: String

}
```

这时编译器便不再会纠结于没有初始化方法实现 `title` 的初始化，而在运行时对于 `MyModel` 的读写也都将能利用数据图完成恰当的操作了。

另外，在通过数据模型图创建 Entity 时要特别注意在 Class 中指定类型名时必须加上 app 的 module 名字，才能保证在代码中做类型转换时不发生错误。



The screenshot shows the 'Entity' configuration panel in Xcode's Data Model Editor. It contains the following fields and options:

- Name:** A text field containing 'MyModel'.
- Class:** A text field containing 'Swifter.MyModel', which is highlighted with a red underline.
- Abstract Entity:** A checkbox that is currently unchecked.
- Parent Entity:** A dropdown menu showing 'No Parent Entity'.
- Indexes:** An empty table with one column header.

最后要指出一点，Apple 在文档中指出 `@NSManaged` 是专门用来解决 Core Data 中动态代码的问题的，因此我们最好是遵守这个规则，只在 `NSManagedObject` 的子类中使用它。但是如果你将 `@NSManaged` 写到其他的类中，也是能够编译通过的。在这种情况下，被标记的属性的访问将会回滚到 Objective-C 的 `getter` 和 `setter` 方法。也即，对于一个叫做 `title` 的属性，在运行时会调用 `title` 和 `setTitle:` 方法。行为上来说和以前的 `@dynamic` 关键字是一样的，我们当然也可以使用 Objective-C 运行时来提供这两个方法，但是要注意的是这么做的话我们就必须对涉及到的类和方法标记为 `@objc`。我并不推荐这样做，因为你无法知道这样的代码在下一个版本中是否还能工作。

闭包歧义

Swift 的闭包写法很多，但是最正规的应该是完整地将闭包的输入和输出都写上，然后用 `in` 关键字隔离参数和实现。比如我们想实现一个 `Int` 的 `extension`，使其可以执行闭包若干次，并同时将其次数传递到闭包中：

```
extension Int {
    func times(f: Int -> ()) {
        print("Int")
        for i in 1...self {
            f(i)
        }
    }
}

3.times { (i: Int) -> () in
    print(i)
}

// 输出:
// Int
// 1
// 2
// 3
```

这里闭包接受 `Int` 输入且没有返回，在这种情况下，我们可以将这个闭包的调用进行简化，成为下面这样：

```
3.times { i in
    print(i)
}
```

这是我们很常见的写法了，也是比较推荐的写法。但是比如某一天，我们觉得这种传入参数的 `times` 有些麻烦，很多时候我们并不需要当前的次数，而只是想简单地将一个闭包重复若干次的话，可能我们会写出 `Int` 的另一个闭包无参数的扩展方法：

```
extension Int {
    func times(f: Void -> Void) {
        print("Void")
        for i in 1...self {
            f()
        }
    }
}
```

你也许会这么解读这段代码：`Int` 有一个扩展方法 `times`，它接受一个叫做 `f` 的闭包，这个闭包不接受参数也没有返回；`times` 的作用是按照这个 `Int` 本身的次数来执行 `f` 闭包若干次。

在早期的 Swift 版本中，这里存在一个歧义调用。虽然在 Swift 1.2 之后的新版本中这个歧义调用问题已经由编译器解决了，但是在修订这个章节时，我认为保留之前的一些讨论可能会对理解整个问题有所帮助。

如果我们在 Swift 1.2 之前的版本中运行这段代码时，输出将发生改变：

```
// 输出：
// Void
//
//
//
```

现在的输出变成了 Void 后面接了三行空格。一个以 i 为参数原来正常工作的方法，在加入了一个“不接受参数”的新的方法情况下，却实际上调用了这个新的方法。我们在没有改变原来的代码的情况下，仅仅是加入了新的方法就让原来的代码失效了，这到底是为什么，又发生了什么？

很明显，现在被调用的是 Void 版本的扩展方法。在继续之前，我们需要明确 Swift 中的 Void 到底是什么。在 Swift 的 module 定义中，Void 只是一个 typealias 而已，没什么特别：

```
typealias Void = ()
```

那么，() 又是什么呢？在[多元组](#)的最后我们指出了，其实 Swift 中任何东西都是放在多元组里的。(42, 42) 是含有两个 Int 类型元素的多元组，(42) 是含有一个 Int 的多元组，那么 () 是什么？没错，这是一个不含有任何元素的多元组。所以其实我们在 extension 里声明的 func times(f: Void -> Void) 根本不是“不接受参数”的闭包，而是一个接受没有任何元素的多元组的闭包。这也不奇怪为什么我们的方法会调用错误了。

当然，在实际使用中这种情况基本是不会发生的。之所以调用到了 Void 版本的方法，是因为我们并没有在调用的时候为编译器提供足够的类型推断信息，因此 Swift 为我们选择了代价最小的 Void 版本来执行。如果我们将调用的代码改为：

```
3.times { i in
    println(i + 1)
}
```

可以看到，这回的输出是：

```
// 输出：
// Int
// 2
// 3
// 4
```

毫无疑问，因为 `Void` 是没有实现 `+ 1` 的，所以类型推断判定一定会调用到 `Int` 类型的版本。

其实不止是 `Void`，像在使用元组时也会有这样的疑惑。比如我们又加入了一个这样看起来是“接受两个参数”的闭包的版本：

```
extension Int {
    func times(f: (Int, Int) -> ()) {
        print("Tuple")
        for i in 1...self {
            f(i, i)
        }
    }
}
```

如果我们先注释掉其他的歧义版本，我们可以看到 `i in` 这种接受一个参数的调用仍然可以编译和运行，它的输出会是：

```
// Tuple
// (1, 1)
// (2, 2)
// (3, 3)
```

道理和 `Void` 是一样的，因此就不再赘述了。

在 **Swift 1.2** 中，类似上面的有歧义的调用会导致编译器报错，并提醒我们发生歧义的方法。得益于新的编译环境，我们现在可以写出更安全和更有保证的代码。

但无论如何，在使用可能存在歧义的闭包时，过度依赖于类型推断其实是一种比较危险的行为，可读性也很差 -- 除非你自己清楚地知道输入类型，否则很难判断调用的到底是哪个方法。为了增强可读性和安全性，最直接是在调用时尽量指明闭包参数的类型。虽然在写的时候会觉得要多写一些内容，但是在 IDE 的帮助下默认实现也是带有全部参数类型的，所以这并不是问题。相信在以后进行扩展和阅读时我们都会感谢当初将类型写全的决定。

```
3.times { (i: Int)->>() in
    print(i)
}

3.times { (i: Void)->>() in
    print(i)
}

3.times { (i: (Int,Int))->>() in
    print(i)
}
```

泛型扩展

Swift 对于泛型的支持使得我们可以避免为类似的功能多次书写重复的代码，这是一种很好的简化。而对于泛型类型，我们也可以使用 `extension` 为泛型类型添加新的方法。

与为普通的类型添加扩展不同的是，泛型类型在类型定义时就引入了类型标志，我们可以直接使用。例如 Swift 的 `Array` 类型的定义是：

```
public struct Array<Element> : CollectionType, Indexable, ... {
    //...
}
```

在这个定义中，已经声明了 `Element` 为泛型类型。在为类似这样的泛型类型写扩展的时候，我们不需要在 `extension` 关键字后的声明中重复地去写 `<Element>` 这样的泛型类型名字 (其实编译器也不允许我们这么做)，在扩展中可以使用和原来所定义一样的符号即可指代类型本体声明的泛型。比如我们想在扩展中实现一个 `random` 方法来随机地取出 `Array` 中的一个元素：

```
extension Array {
    var random: Element? {
        return self.count != 0 ?
            self[Int(arc4random_uniform(UInt32(self.count)))] :
            nil
    }
}

let languages = ["Swift", "ObjC", "C++", "Java"]
languages.random!
// 随机输出是这四个字符串中的某个

let ranks = [1, 2, 3, 4]
ranks.random!
// 随机输出是这四个数字中的某个
```

在扩展中是不能添加整个类型可用的新泛型符号的，但是对于某个特定的方法来说，我们可以添加 `T` 以外的其他泛型符号。比如在刚才的扩展中加上：

```
func appendRandomDescription
<U: CustomStringConvertible>(input: U) -> String {

    if let element = self.random {
        return "\(element) " + input.description
    } else {
        return "empty array"
    }
}
```

我们限定了只接受实现了 `CustomStringConvertible` 的参数作为参数，然后将这个内容附加到自身

的某个随机元素的描述上。因为参数 `input` 实现了 `CustomStringConvertible`，所以在方法中我们可以使用 `description` 来获取描述字符串。

```
let languages = ["Swift", "ObjC", "C++", "Java"]
languages.random!

let ranks = [1, 2, 3, 4]
ranks.random!

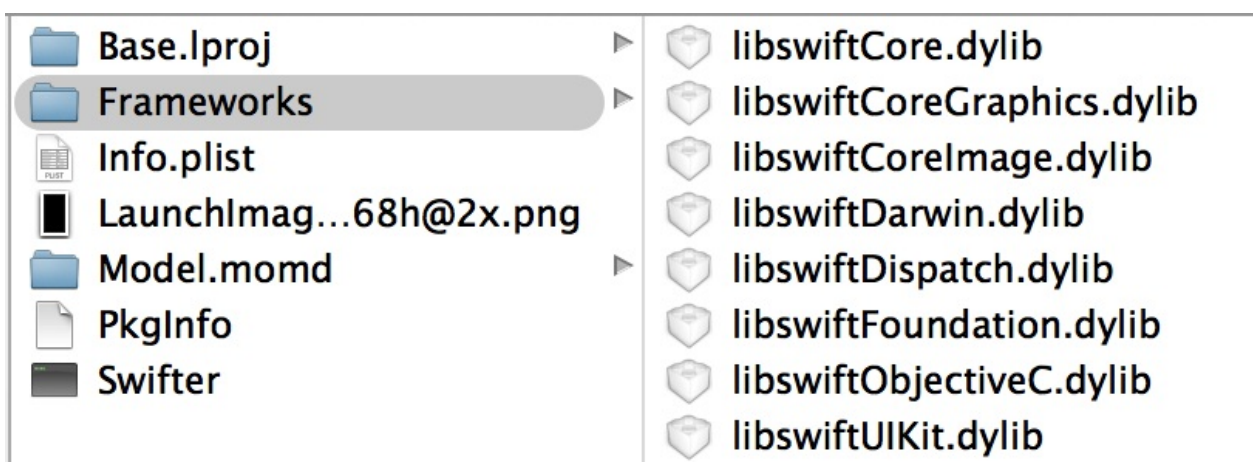
languages.appendRandomDescription(ranks.random!)
// 随机组合 languages 和 ranks 中的各一个元素，然后输出
```

虽然这是个生造的需求，但是能说明泛型在扩展里的使用方式。简单说就是我们不能通过扩展来重新定义当前已有的泛型符号，但是可以对其进行使用；在扩展中也不能为这个类型添加泛型符号；但只要名字不冲突，我们是可以在新声明的方法中定义和使用新的泛型符号的。

兼容性

作为一门新兴语言，Swift 必然会经常性地发生变化。可以预见到在未来的一到两年内 Swift 必然会迎来很多的修正和特性强化。现在的 Swift 作为一门 app 开发语言，最终的运行环境会是各式各样的电子设备。在语言版本不断更新变化的同时，如何尽可能地使更多的设备和系统可以使用这门语言开发的 app，是最大的问题之一。

Apple 通过将一个最小化的运行库集成打包到 app 中这样的方式来解决兼容性的问题。使用了 Swift 语言的项目在编译时会在 app 包中带有这一套运行时环境，并在启动时加载这些 `dylib` 包作为 Swift 代码的运行环境。这些库文件位于打包好的 app 的 `Frameworks` 文件夹中：



这样带来的好处有两点。首先是虽然 Swift 语言在不断变化，但是你的 app 不论在什么样的系统版本上都可以保持与开发编译时的行为一致，因为你依赖的 Swift 运行时是和 app 绑定的。这对于确保 Swift 升级后新版本的 app 在原有的设备和系统上运行正常是必要的。

另一个好处是向下兼容。虽然 Swift 是和 iOS 8 及 OSX 10.10 一同推出的，但是通过加载 Swift 的动态库，Apple 允许 Swift 开发的 app 在 iOS 7 和 OSX 10.9 上也能运行，这对 Swift 的尽快推广和使用也是十分关键的。

但是这样的做法的缺点也很明显，那就是更大的 app 尺寸和内存占用。在 Swift 1.0 版本下，通过 Release 打包后同样的 Swift 空工程的 ipa 文件要比 Objective-C 空工程的尺寸大上 4~5 MB，在设备上运行时也会有额外的 2~3 MB 的内存空间开销。如果制作的 app 对于磁盘空间占用很敏感的话，现在的 Swift 的这个不足是难以绕开的。

Xcode 会在编译 app 时判断在当前项目中是否含有 Swift 文件，如果存在的话，将自动为我们把运行时的 dylib 复制到 app 包中。而在 iOS 8 中，我们可以为系统开发像是动作扩展，照片编辑或者通知中心窗体等扩展组件。这些扩展是以 target 的形式存在于主 app 项目中的。因此存在一种可能性，那就是主项目没有用到 Swift，但是在扩展中用到了 Swift。这种情况下，我们需要手动将项目 app target 的编译设置中 Build Options 下的 Embedded Content Contains Swift Code 设置为 Yes，以确保 Swift 的运行库被打包进 app 中。

另外还需一提的是对于第三方框架的使用。虽然我们在 Framework 一节中提到了使用 Swift 构建

框架并提供使用，但是现在直接使用编译好的 **Swift** 框架并不是一件明智的事情。对于第三方 **Swift** 代码的正确使用方式，要么是直接将源代码添加到项目中进行编译，要么是将生成 **framework** 的项目作为依赖添加到自己的项目中一起编译。总之，我们最好是取得源代码并确保让其与我们的项目共用同一套运行环境，任何已编译好的二进制包在运行使用时都是要承担 **Swift** 版本升级所带来的兼容性风险的。

这个打包进 **app** 的运行环境可以说是到目前为止使用 **Swift** 开发的最大的限制。关于这个限制，**Apple** 承诺将在一两年内 **Swift** 持续改进并且拥有一个相对稳定的运行时 **API** 后，将其添加到系统中进行固定。届时这篇文章中的所有限制都将不再存在。但是在此之前，如果我们想用 **Swift** 进行开发的话，就必须面对和承受这些不足。

列举 enum 类型

设想我们有这样一个需求，通过对于一副扑克牌的花色和牌面大小的 `enum` 类型，凑出一套不含大小王的扑克牌的数组。

扑克牌花色和牌面大小分别由下面两个 `enum` 来定义：

```
enum Suit: String {
    case Spades = "黑桃"
    case Hearts = "红桃"
    case Clubs = "草花"
    case Diamonds = "方片"
}

enum Rank: Int, Printable {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
    var description: String {
        switch self {
            case .Ace:
                return "A"
            case .Jack:
                return "J"
            case .Queen:
                return "Q"
            case .King:
                return "K"
            default:
                return String(self.rawValue)
        }
    }
}
```

最容易想到的方式当然不外乎对两个 `enum` 进行两次循环，先循环取出 `Suit` 中的四种花色，然后在其中循环 `Rank` 类型取出数字后，就可以配合得到 52 张牌了。

在其他很多语言中，我们可以对 `enum` 类型或者其某个类似 `values` 的属性直接进行枚举，写出来的话，可能会是类似这样的代码：

```
for suit in Suit.values {
    for rank in Rank.values {
        // ...
        // 处理数据
    }
}
```

但是在 **Swift** 中，由于在 `enum` 中的某一个 `case` 中我们是可以添加具体值的 (就是 `case Some(T)` 这样的情况)，因此直接使用 `for...in` 的方式在语义上是无法表达出所有情况的。不过因为在我们这个特定的情况中并没有带有参数的枚举类型，所以我们可以利用 `static` 的属性来获取一个可

以进行循环的数据结构：

```
protocol EnumeratableEnumType {
    static var allValues: [Self] {get}
}

extension Suit: EnumeratableEnumType {
    static var allValues: [Suit] {
        return [.Spades, .Hearts, .Clubs, .Diamonds]
    }
}

extension Rank: EnumeratableEnumType {
    static var allValues: [Rank] {
        return [.Ace, .Two, .Three,
                .Four, .Five, .Six,
                .Seven, .Eight, .Nine,
                .Ten, .Jack, .Queen, .King]
    }
}
```

在这里我们使用了一个接口来更好地定义适用的接口。关于其中的 `class` 和 `static` 的使用情景，可以参看[这一篇总结](#)。在实现了 `allValues` 后，我们就可以按照上面的思路写出：

```
for suit in Suit.allValues {
    for rank in Rank.allValues {
        print("\(suit.rawValue)\(rank)")
    }
}

// 输出：
// 黑桃A
// 黑桃2
// 黑桃3
// ...
// 方片K
```

尾递归

递归在程序设计中是一种很有用的方法，它可以将复杂的过程用易于理解的方式转化和描述。举个例子，比如我们想要写一个从 0 累加到 n 的函数，如果我们不知道等差数列求和公示的话，就可以用递归的方式来做：

```
func sum(n: UInt) -> UInt {
    if n == 0 {
        return 0
    }
    return n + sum(n - 1)
}

sum(4) // 10
sum(100) // 5050
```

看起来没问题。但是我们如果用一个大一点的数的话，运行时就会出现错误，比如

```
sum(1000000)
// EXC_BAD_ACCESS (code=2, address=...)
```

这是因为每次对于 `sum` 的递归调用都需要在调用栈上保存当前状态，否则我们就无法计算最后的 `n + sum(n - 1)`。当 `n` 足够大，调用栈足够深的时候，栈空间将被耗尽而导致错误，也就是我们常说的栈溢出了。

一般对于递归，解决栈溢出的一个好方法是采用尾递归的写法。顾名思义，尾递归就是让函数里的最后一个动作是一个函数调用的形式，这个调用的返回值将直接被当前函数返回，从而避免在栈上保存状态。这样一来程序就可以更新最后的栈帧，而不是新建一个，来避免栈溢出的发生。在 Swift 2.0 中，编译器现在支持嵌套方法的递归调用了 (Swift 1.x 中如果你尝试递归调用一个嵌套函数的话会出现编译错误)，因此 `sum` 函数的尾递归版本可以写为：

```
func tailSum(n: UInt) -> UInt {
    func sumInternal(n: UInt, current: UInt) -> UInt {
        if n == 0 {
            return current
        } else {
            return sumInternal(n - 1, current: current + n)
        }
    }

    return sumInternal(n, current: 0)
}

tailSum(1000000)
```

但是如果你在项目中直接尝试运行这段代码的话还是会报错，因为在 Debug 模式下 Swift 编译器

并不会对尾递归进行优化。我们可以在 `scheme` 设置中将 `Run` 的配置从 `Debug` 改为 `Release`，这段代码就能正确运行了。

后记及致谢

其实写这本书纯属心血来潮。从产生想法到做出决定花了一炷香的时间，而从开始下笔到这篇后记花了一个月的时间。

这么点儿时间，确实是不够写出一本好书的

这是我到现在，所得到的第一个教训。

虽然在博客上已经坚持写了两三年，但是写书来说，这还是自己的第一次。从刚下笔时的诚惶诚恐，到中途的渐入佳境，挥挥洒洒，再到最后绞尽脑汁，也算是在这一个月里把种种酸甜苦辣尝了个遍。诚然，不会有哪一本书能完美，大家也不必指望能得到什么武林秘籍帮你一夜功成。知识的积累从来都只能依靠日常点滴，而我也尽了自己的努力，尝试将我的积累分享出来，仅此而已。

所谓靡不有初，鲜克有终，我看过太多的雄心壮志，也见过许多的半途而废。如果您看到了这个后记，那么大概您真的是耐着性子把这本有些枯燥书都看完了。我很感谢您的坚持和对我的忍耐，并希望这些积累能够在您自己的道路上起到一些帮助。当然也可能您是喜欢“直接翻看后记”的剧透党，但我依然想要进行感谢，这个世界总会因为感谢而温暖和谐。

这本书在写作过程中参考了许多资料，包括但不限于 [Apple 关于 Swift 的官方文档](#)，[Apple 开发者论坛](#) 上关于 Swift 的讨论，[Stackoverflow](#) 的 Swift 标签的所有问题，[NSHipster](#)，[NSBlog](#) 的[周五问答](#)，[Airspeed Velocity](#)，[猫·仁波切](#) 以及其他一些由于篇幅限制而没有列出参考博客。在此对这些社区的贡献者们表示衷心感谢。

在我写作的过程中，国内的许多开发者朋友们忍受了我的各种莫名其妙的低级疑问，他们的热心和细致的解答，对这本书的深入和准确性起到了很大的帮助。而本书的预售工作也在大家的捧场和宣传下顺利地进行并完成，在这里我想一并向他们表示感谢，正是你们的坚持和努力，让国内的开发者社区如此充满活力。

最后，感谢我的家人在这一个月时间内对我的照顾，让我可以不用承担和思考太多写书以外的事情。随着这本书暂时告一段落，我想是时候回归到每天洗碗和拖地的日常劳动中去了。

我爱这个世界，愿程序让这个世界变得更美好！

版本更新

2.0.0 (2015 年 8 月 24 日)

2.0.0 版本是本书的一个重大更新。根据 Swift 2.0 的内容重新修订了本书，包括对新内容的扩展和过时内容的删除。新版本中对原来的 tips 进行了归类整理，将全书大致分为三个部分，以方便读者阅读查阅。另外，为了阅读效果，对全书的排版和字体等进行了大幅调整。

这一年来随着 Swift 的逐渐变化，书中有不少示例代码的用法已经和现在版本的 Swift 有所区别，因此在这次修订中对所有的代码都在 Swift 2.0 环境下进行了再次的验证和修改。现在本书的所有代码例子已经附在 Swifter.playground 文件中，一并提供给读者进行参考。

新增条目

- [《Protocol Extension》](#)
- [《where 和模式匹配》](#)
- [《错误和异常处理》](#)：代替了原来的错误处理一节。因为 Swift 2 中引入了异常处理的机制，因此现在对于发生错误后如何获取错误信息以及从错误中恢复有了新的方式。原来的内容已经不再适合新版本，因此用新的一节来替代。
- [《indirect 和嵌套 enum》](#)
- [《尾递归》](#)

修改和删除

- 为了表意明确，有特别的理由的个例除外，将其他所有返回 () 的闭包的改写为了返回 Void。
- [《@autoclosure 和 ??》](#) 修正了一处笔误。
- [《操作符》](#) 有几处 Vector2 应为 Vector2D，修正用词错误。
- [《typealias 和泛型接口》](#) 修正一处表述问题，使得句子读起来更通顺。
- [《Sequence》](#) 中 Sequence 相关的全局方法现在已经被写为接口扩展，因此对说明也进行了相应地更改。
- [《多元组》](#) 中有关错误处理的代码已经被异常机制替代，因此选取了一个新的例子来说明如何使用多元组。
- [《方法参数名称省略》](#) Swift 2 中已经统一了各 scope 中的方法名称中的参数规则，因此本节已经没有存在的必要，故删去。
- [《Swift 命令行工具》](#) 编译器的命令行工具在输出文件时的参数发生了变化，对此进行修正。另外在运行命令时省去了已经不必要的 xcrun。
- [《下标》](#) Array 现在的泛型类型占位符变为了 Element，另外原来的 Slice 被 ArraySlice 取代了。更新了代码使其能在新版本下正常工作。
- [《初始化返回 nil》](#) 因为 Int 已经有了内建的从 String 进行初始化的方法，因此改变了本节的例程。现在使用一个中文到数字的转换初始化方法来进行说明。
- [《protocol 组合》](#) 修正了示例代码中的错误。

- 《**static 和 class**》Swift 2 中已经可以用 `static` 作为通用修饰，因此修改了一些过时的内容。
- 《**可选接口**》中加入了关于使用接口扩展来实现接口方法可选的技巧。因为加入了其他内容，这一节也更名为《**可选接口和接口扩展**》。
- 《**内存管理, weak 和 unowned**》新版本中 Playground 也能正确地反应内存状况以及与 ARC 协同工作了，因此去除了必须在项目中运行的条件，另外修改了代码使它们能在 Playground 中正常工作。
- 《**default 参数**》中与参数 `#` 修饰符相关的内容已经过时，删除。`NSLocalizedString` 的补全现在也已经改进，所以不再需要说明。
- 《**正则表达式**》`NSRegularExpression` 的初始化方法现在有可能直接抛出异常，使用异常机制重写了本节的示例代码。
- 《**模式匹配**》同《**正则表达式**》，使用异常机制重写了示例代码。
- 《**COpaquePointer 和 C convention**》`CFunctionPointer` 在 Swift 2.0 中被删除，现在 C 方法指针可以直接由 Swift 闭包进行无缝转换。重写了该部分内容，添加了关于 `@convention` 标注的说明。
- 《**Foundation 框架**》在 Swift 2.0 中 `String` 和 `NSString` 的转换已经有了明确的界限，因此本节内容已经过时，故删去。
- 《**GCD 和延时调用**》重新说明了 iOS 8 中对 block 的改进。另外由于 Swift 2 中重新引入了 `performSelector`，对相关内容进行了小幅调整。
- 《**获取对象类型**》`Swift.Type` 现在对于 `print` 和 `debugPrint` 中有了新的实现，进行了补充说明和代码修正。
- 《**类型转换**》因为 Objective-C 中对 collection 加入了泛型的支持，现在在 Swift 中使用 Cocoa API 时基本已经不太需要类型转换，故删去。
- 《**局部 scope**》添加了关于 `do` 的说明。Swift 2.0 中加入了 `do` 关键字，可以作为局部作用域来使用。
- 《**print 和 debugPrint**》现在 `Printable` 和 `DebugPrintable` 接口的名称分别改为了 `CustomStringConvertible` 和 `CustomDebugStringConvertible`。
- 《**Playground 限制**》随着 Apple 对 Playground 的改进和修复，原来的一些限制 (特别是内存管理上的限制) 现在已经不复存在。这一节已经没有太大意义，故删去。
- 《**Swizzle**》`+load` 方法在 Swift 1.2 中已经不能被覆盖使用，另外使用 Swift 实现的 `+load` 方法在运行时也不再被调用，因此需要换为使用 `+initialize` 来实现方法的交换。改写了代码以使其正常工作，另外加入了关于交换方法选择的说明。
- 《**find**》因为引入了 `protocol extension`，像类似 `find` 一类作用在 collection 上的全局方法都已经使用 `protocol extension` 实现了，因此本节移除。
- 《**Reflection 和 Mirror**》`reflect` 方法和 `MirrorType` 类型现在已经变为 Swift 标准库的私有类型，现在我们需要使用 `Mirror` 来获取和使用对象的反射。重写了本节的内容以符合 Swift 2.0 中的反射特性和使用方式。另外，为了避免误导，对反射的使用场合也进行了说明。
- 《**文档注释**》Swift 2.0 中文档注释的格式发生了变化，因此对本节内容进行了修改已符合新版本的格式要求。
- 《**Options**》原来的 `RawOptionSetType` 在 Swift 2.0 中已经被新的 `OptionSetType` 替代，现在 Options 有了更简洁的表示方法和运算逻辑。另外加入了 Options 集合运算的内容，以及更新了生成 Options 的代码片段。
- 《**Associated Object**》作为 Key 值的变量需要是 Optional 类型，因此对原来不正确的示例代码进行了修改。
- 《**Swift 中的测试**》Swift 2 中导入了 `@testable`，可以让测试 target 访问到导入的 target 的 internal 代码，因此本节的一些讨论过时了。根据 Swift 2.0 的测试方式重写了本节内容。

- [其他]: 修正了一些用词上的不妥和错别字。

1.2.1 (2015 年 2 月 25 日)

- 《@autoclosure 和 ??》，以及其他出现 @autoclosure 的章节中，将 @autoclosure 的位置进行了调整。现在 @autoclosure 作为参数名的修饰，而非参数类型的修饰。
- 《闭包歧义》Swift 1.2 中闭包歧义的使用将由编译器给出错误。在保留 Swift 1.1 及之前的讨论的前提下，补充说明了 Swift 1.2 版本以后的闭包歧义的处理和避免策略。
- 《获取对象类型》删除了过时内容和已经无效的黑科技，补充了 dynamicType 对内建 Swift 类型的用法说明。
- 《单例》现在可以直接使用类常量/变量，因此更新了推荐的单例写法。
- 《static 和 class》中更新了类常/变量的用法。
- 《@UIApplicationMain》中 C_ARGC 和 C_ARGV 分别被 Process.argc 和 Process.unsafeArgv 替代。
- 《COpaquePointer 和 CFunctionPointer》更新了一处 API 的参数名。
- 《类型转换》使用意义明确的 Swift 1.2 版本的 as! 进行强制转换。
- 《数学和数字》作为补充，添加了 Darwin 中判定 NAN 的方法 isnan。
- 《Swift 命令行工具》中新增了新版本中不需要 xcrun 的说明。
- 《方法参数名称省略》中的一处 API 的 unwrap 更新。
- 《C 代码调用和 @asmname》修正了一个头文件引入的错误。

1.2.0 (2015 年 2 月 10 日)

- 《static 和 class》一节针对 Swift 1.2 进行了更新。Swift 1.2 中 protocol 中定义的“类方法”需要使用 static 而非 class。
- 《多类型和容器》中的错别字，“不知名”应该为“不指明”。
- 《内存管理，weak 和 unowned》中标注例子中标注错误，标注中的逗号应该是冒号。
- 《Reflection 和 MirrorType》一节代码中遗漏了一个引号。
- 《Any 和 AnyObject》修正了一处赋值时的代码警告。

1.1.2 (2014 年 12 月 2 日)

- 《default 参数》中的错别字，“常亮”应该为“常量”。
- 《Selector》中示例代码 func aMethod(external paramName: String) { ... } 中 String 应为 AnyObject!，否则会导致程序崩溃（因为 Swift 的原生 String 并没有实现 NSCopying）。
- 《获取对象类型》中由于 API 变更，使用 objc_getClass 和 objc runtime 获取对象类型的方法已经不再有效。新的 API 需要与 UnsafePointer 有关，已经超出章节内容，故将该部分内容删除。
- 《多类型和容器》中由于 Swift 类型推断和字面量转换的改善，原来的陷阱基本都已经消除，因此本节进行了一些简化，去掉了过时的内容。

1.1.1 (2014 年 11 月 7 日)

- 《闭包歧义》中的内容在 Swift 1.1 中已经发生了变化，因此重写了这一节。

- 将代码示例中的 `toRaw()` 和 `fromRaw()` 按照 Swift 1.1 的语法改为了 `rawValue` 和对应的 `init` 方法。

1.1.0 (2014 年 10 月 21 日)

- 《属性访问控制》中“Swift 中的 `swift` 和其他大部分语言不太一样”应为“Swift 中的 `private` 和其他大部分语言不太一样”
- 《代码组织和 Framework》中“开发中我们所使以的第三方框”改为“开发中我们所使用的第三方框”
- 《方法名称参数省略》中“使用自动覆盖的方式”应为“使用原子写入的方式”
- 《内存管理, `weak` 和 `unowned`》中对 `unowned` 的表述不当, 进行了修正
- 《字面量转换》的内容完全重写, 更新为适应 Swift 1.1 版本。
- 《Any 和 AnyObject》一节作出修正, Any 现在可以支持方法类型了。
- 《初始化返回 nil》一节作出修正, 在本书 1.0.1 版本的基础上, 进一步更改了部分内容的表述, 使其更适应 Swift 1.1 中关于可失败的初始化方法的修改。

1.0.1 (2014 年 9 月 25 日)

- 《多类型和容器》中在 `let mixed: [Printable]` 处有个字面量转换的小陷阱, 在是否导入 Foundation 时存在一个有趣的差别, 对这部分进行了一定补充说明;
- 《将 protocol 的方法声明为 mutating》中 `blueColor()! -> blueColor()`。 `blueColor()` 返回的已经是 `UIColor` 而不是 `UIColor?`;
- 《typealias 和泛型接口》中有误字“wei”, 删除;
- 《Optional Chaining 文字错误》它们的等价的 -> 它们是等价的;
- 《Swift 命令行工具》中示例代码 `xcrun swift MyClass.swift main.swift` 应当为 `xcrun swiftc MyClass.swift main.swift`;
- 《内存管理, `weak`和`unowned`》笔误, “变量一定需要时 Optional 值”中“时”应当为“是”。

1.0.0 (2014 年 9 月 19 日)

《Swifter - 100 个 Swift 必备 tips》首发