这篇文章简要介绍了苹果于 WWDC 2014 发布的编程语言——Swift。

**Swift 是什么？**

Swift 是苹果于 WWDC 2014 发布的编程语言，这里引用 **The Swift Programming Language** 的原话：

Swift is a new programming language for iOS and OS X apps that builds on the best of C and Objective-C, without the constraints of C compatibility.

Swift adopts safe programming patterns and adds modern features to make programming easier, more flexible and more fun.

Swift’ s clean slate, backed by the mature and much-loved Cocoa and Cocoa Touch frameworks, is an opportunity to imagine how software development works.

Swift is the first industrial-quality systems programming language that is as expressive and enjoyable as a scripting language.

简单的说：

Swift 用来写 iOS 和 OS X 程序。（估计也不会支持其它屌丝系统）

Swift 吸取了 C 和 Objective-C 的优点，且更加强大易用。

Swift 可以使用现有的 Cocoa 和 Cocoa Touch 框架。

Swift 兼具编译语言的高性能（Performance）和脚本语言的交互性（Interactive）。

**Swift 语言概览**

**基本概念**

注：这一节的代码源自 The Swift Programming Language 中的 A Swift Tour。

**Hello, world**

类似于脚本语言，下面的代码即是一个完整的 Swift 程序。

- println("Hello, world") 变量与常量

Swift 使用 var 声明变量，let 声明常量。

- var myVariable = 42
- myVariable = 50
- let myConstant = 42

**类型推导**

Swift 支持类型推导（Type Inference），所以上面的代码不需指定类型，如果需要指定类型：

- let explicitDouble : Double = 70

Swift 不支持隐式类型转换（Implicitly casting），所以下面的代码需要显式类型转换（Explicitly casting）：

- let label = "The width is "
- let width = 94
- let width = label + String(width)

字符串格式化

Swift 使用\(item)的形式进行字符串格式化：

- let apples = 3
- let oranges = 5
- let appleSummary = "I have \(apples) apples."
- let appleSummary = "I have \(apples + oranges) pieces of fruit."

数组和字典

Swift 使用[]操作符声明数组（array）和字典（dictionary）：

- var shoppingList = ["catfish", "water", "tulips", "blue paint"]
- shoppingList[1] = "bottle of water"
- var occupations = [
- "Malcolm": "Captain",
- "Kaylee": "Mechanic",
- ]
- occupations["Jayne"] = "Public Relations"

一般使用初始化器（initializer）语法创建空数组和空字典：

- let emptyArray = String[]()
- let emptyDictionary = Dictionary<String, Float>()

如果类型信息已知，则可以使用[]声明空数组，使用[:]声明空字典。

**控制流**

**概览**

Swift 的条件语句包含 if 和 switch，循环语句包含 for-in、for、while 和 do-while，循环/判断条件不需要括号，但循环/判断体（body）必需括号：

- let individualScores = [75, 43, 103, 87, 12]
  var teamScore = 0
  for score in individualScores {
  if score > 50 {
  teamScore += 3
  } else {
  teamScore += 1
  }
  }

**可空类型**

结合 if 和 let，可以方便的处理可空变量（nullable variable）。对于空值，需要在类型声明后添加?显式标明该类型可空。

- var optionalString: String? = "Hello"
  optionalString == nil

  var optionalName: String? = "John Appleseed"
  var gretting = "Hello!"
  if let name = optionalName {
  gretting = "Hello, \(name)"
  }

**灵活的 switch**

Swift 中的 switch 支持各种各样的比较操作：

- let vegetable = "red pepper"
- switch vegetable {
- case "celery":
- let vegetableComment = "Add some raisins and make ants on a log."
- case "cucumber", "watercress":
- let vegetableComment = "That would make a good tea sandwich."
- case let x where x.hasSuffix("pepper"):
- let vegetableComment = "Is it a spicy \(x)?"
- default:
- let vegetableComment = "Everything tastes good in soup."
- }

**其它循环**

for-in 除了遍历数组也可以用来遍历字典：

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
largest
```

while 循环和 do-while 循环：

```
var n = 2
while n < 100 {
        n = n * 2
}
n
var m = 2
do {
        m = m * 2
} while m < 100
m
```

Swift 支持传统的 for 循环，此外也可以通过结合..（生成一个区间）和 for-in 实现同样的逻辑。

```
var firstForLoop = 0
for i in 0..3 {
    firstForLoop += i
}
firstForLoop
var secondForLoop = 0
for var i = 0; i < 3; ++i {
    secondForLoop += 1
}
```

注意：Swift 除了..还有...：..生成前闭后开的区间，而...生成前闭后闭的区间。

**函数和闭包**

**函数**

Swift 使用 func 关键字声明函数：

```
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
greet("Bob", "Tuesday")
```

通过元组（Tuple）返回多个值：

```
func getGasPrices() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}
getGasPrices()
```

支持带有变长参数的函数：

```
func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}
sumOf()
sumOf(42, 597, 12)
```

函数也可以嵌套函数：

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

作为头等对象，函数既可以作为返回值，也可以作为参数传递：

```
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
```

```
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)

func hasAnyMatches(list: Int[], condition: Int -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, lessThanTen)
```

**闭包**

本质来说，函数是特殊的闭包，Swift 中可以利用{}声明匿名闭包：

```
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
    })
```

当闭包的类型已知时，可以使用下面的简化写法：

```
numbers.map({ number in 3 * number })
```

此外还可以通过参数的位置来使用参数，当函数最后一个参数是闭包时，可以使用下面的语法：

```
sort([1, 5, 3, 12, 2]) { $0 > $1 } 类和对象 创建和使用类
```

Swift 使用 class 创建一个类，类可以包含字段和方法：

```
class Shape {
var numberOfSides = 0
func simpleDescription() -> String {
return "A shape with \(numberOfSides) sides."
}
```

```
}
```

创建 Shape 类的实例，并调用其字段和方法。

```
var shape = Shape()
shape.numberOfSides = 7
varshapeDescription = shape.simpleDescription()
```

通过 init 构建对象，既可以使用 self 显式引用成员字段（name），也可以隐式引用（numberOfSides）。

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String
    init(name: String) {
        self.name = name
    }
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

使用 deinit 进行清理工作。

**继承和多态**

Swift 支持继承和多态（override 父类方法）：

```
class Square: NamedShape {
    var sideLength: Double
    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }
    func area() -> Double {
        return sideLength * sideLength
    }
    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
```

注意：如果这里的 simpleDescription 方法没有被标识为 override，则会引发编译错误。

**属性**

为了简化代码，Swift 引入了属性（property），见下面的 perimeter 字段：

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0
    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }
    var perimeter: Double {
    get {
        return 3.0 * sideLength
    }
    set {
        sideLength = newValue / 3.0
    }
    }
    override func simpleDescription() -> String {
        return "An equilateral triagle with sides of length \(sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
triangle.perimeter
triangle.perimeter = 9.9
triangle.sideLength
```

注意：赋值器（setter）中，接收的值被自动命名为 newValue。

**willSet 和 didSet**

EquilateralTriangle 的构造器进行了如下操作：

为子类型的属性赋值。 调用父类型的构造器。 修改父类型的属性。

如果不需要计算属性的值，但需要在赋值前后进行一些操作的话，使用 willSet 和 didSet：

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
    willSet {
        square.sideLength = newValue.sideLength
    }
```

```
        }
        var square: Square {
        willSet {
            triangle.sideLength = newValue.sideLength
        }
        }
        init(size: Double, name: String) {
            square = Square(sideLength: size, name: name)
            triangle = EquilateralTriangle(sideLength: size, name: name)
        }
    }
    var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
    triangleAndSquare.square.sideLength
    triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
    triangleAndSquare.triangle.sideLength
```

从而保证 triangle 和 square 拥有相等的 sideLength。

**调用方法**

Swift 中，函数的参数名称只能在函数内部使用，但方法的参数名称除了在内部使用外还可以在外部使用（第一个参数除外），例如：

```
    class Counter {
        var count: Int = 0
        func incrementBy(amount: Int, numberOfTimes times: Int) {
            count += amount * times
        }
    }
    var counter = Counter()
    counter.incrementBy(2, numberOfTimes: 7)
```

注意 Swift 支持为方法参数取别名：在上面的代码里，numberOfTimes 面向外部，times 面向内部。

**?的另一种用途**

使用可空值时，?可以出现在方法、属性或下标前面。如果?前的值为 nil，那么?后面的表达式会被忽略，而原表达式直接返回 nil，例如：

```
    let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional
    square")
    let sideLength = optionalSquare?.sideLength
```

当 optionalSquare 为 nil 时，sideLength 属性调用会被忽略。

**枚举和结构**

**枚举**

使用 enum 创建枚举——注意 Swift 的枚举可以关联方法：

```
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
        func simpleDescription() -> String {
        switch self {
            case .Ace:
                return "ace"
            case .Jack:
                return "jack"
            case .Queen:
                return "queen"
            case .King:
                return "king"
            default:
                return String(self.toRaw())
        }
    }
}
let ace = Rank.Ace
let aceRawValue = ace.toRaw()
```

使用 toRaw 和 fromRaw 在原始（raw）数值和枚举值之间进行转换：

```
if let convertedRank = Rank.fromRaw(3) {
let threeDescription = convertedRank.simpleDescription()
}
```

注意枚举中的成员值（member value）是实际的值（actual value），和原始值（raw value）没有必然关联。

一些情况下枚举不存在有意义的原始值，这时可以直接忽略原始值：

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
        func simpleDescription() -> String {
        switch self {
            case .Spades:
                return "spades"
            case .Hearts:
```

```
            return "hearts"
        case .Diamonds:
            return "diamonds"
        case .Clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.Hearts
let heartsDescription = hearts.simpleDescription()
```

除了可以关联方法，枚举还支持在其成员上关联值，同一枚举的不同成员可以有不同的关联的值：

```
enum ServerResponse {
    case Result(String, String)
    case Error(String)
}
let success = ServerResponse.Result("6:00 am", "8:09 pm")
let failure = ServerResponse.Error("Out of cheese.")
switch success {
    case let .Result(sunrise, sunset):
        let serverResponse = "Sunrise is at \(sunrise) and sunset is at \(sunset)."
    case let .Error(error):
        let serverResponse = "Failure... \(error)"
}
```

**结构**

Swift 使用 struct 关键字创建结构。结构支持构造器和方法这些类的特性。结构和类的最大区别在于：结构的实例按值传递（passed by value），而类的实例按引用传递（passed by reference）。

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .Three, suit: .Spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

**协议（protocol）和扩展（extension） 协议**

Swift 使用 protocol 定义协议：

```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

类型、枚举和结构都可以实现（adopt）协议：

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += " Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription
struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

**扩展**

扩展用于在已有的类型上增加新的功能（比如新的方法或属性），Swift 使用 extension 声明扩展：

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
7.simpleDescription
```

**泛型（generics）**

Swift 使用<>来声明泛型函数或泛型类型：

```
func repeat<ItemType>(item: ItemType, times: Int) -> ItemType[] {
    var result = ItemType[]()
    for i in 0..times {
        result += item
    }
    return result
}
repeat("knock", 4)
```

Swift 也支持在类、枚举和结构中使用泛型：

```
// Reimplement the Swift standard library's optional type
enum OptionalValue<T> {
    case None
    case Some(T)
}
var possibleInteger: OptionalValue<Int> = .None
possibleInteger = .Some(100)
```

有时需要对泛型做一些需求（requirements），比如需求某个泛型类型实现某个接口或继承自某个特定类型、两个泛型类型属于同一个类型等等，Swift 通过 where 描述这些需求：

```
func anyCommonElements <T, U where T: Sequence,
U: Sequence, T.GeneratorType.Element: Equatable,
T.GeneratorType.Element == U.GeneratorType.Element> (lhs: T, rhs: U) -> Bool {
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
        }
    }
    return false
}
anyCommonElements([1, 2, 3], [3])
```

Swift 语言概览就到这里，有兴趣的朋友请进一步阅读 The Swift Programming Language。

接下来聊聊个人对 Swift 的一些感受。

**个人感受**

**注意**：下面的感受纯属个人意见，仅供参考。

**大杂烩**

尽管我接触 Swift 不足两小时，但很容易看出 Swift 吸收了大量其它编程语言中的元素，这些元素包括但不限于：

属性（Property）、可空值（Nullable type）语法和泛型（Generic Type）语法源自 C#。 格式风格与 Go 相仿（没有句末的分号，判断条件不需要括号）。 Python 风格的当前实例引用语法（使用 self）和列表字典声明语法。 Haskell 风格的区间声明语法（比如 1..3，1...3）。 协议和扩展源自 Objective-C（自家产品随便用）。 枚举类型很像 Java（可以拥有成员或方法）。 class 和 struct 的概念和 C#极其相似。

注意这里不是说 Swift 是抄袭——实际上编程语言能玩的花样基本就这些，况且 Swift 选的都是在我看来相当不错的特性。

而且，这个大杂烩有一个好处——就是任何其它编程语言的开发者都不会觉得 Swift 很陌生——这一点很重要。

**拒绝隐式（Refuse implicity）**

Swift 去除了一些隐式操作，比如隐式类型转换和隐式方法重载这两个坑，干的漂亮。

Swift 的应用方向

我认为 Swift 主要有下面这两个应用方向：

**教育**

我指的是编程教育。现有编程语言最大的问题就是交互性奇差，从而导致学习曲线陡峭。相信 Swift 及其交互性极强的编程环境能够打破这个局面，让更多的人——尤其是青少年，学会编程。

这里有必要再次提到 Brec Victor 的 Inventing on Principle，看了这个视频你就会明白一个交互性强的编程环境能够带来什么。

**应用开发**

现有的 iOS 和 OS X 应用开发均使用 Objective-C，而 Objective-C 是一门及其繁琐（verbose）且学习曲线比较陡峭的语言，如果 Swift 能够提供一个同现有 Obj-C 框架的简易互操作接口，我相信会有大量的程序员转投 Swift；与此同时，Swift 简易的语法也会带来相当数量的其它平台开发者。

总之，上一次某家大公司大张旗鼓的推出一门编程语言及其编程平台还是在 2000 年（微软推出 C#），将近 15 年之后，苹果推出 Swift——作为开发者，我很高兴能够见证一门编程语言的诞生