Swift 概览

快速检索 [点击收起]

依照传统，使用新语言写的第一个程序都应该是在屏幕上打印“Hello,world!”，使用 Swift 语言，你可以在一行中完成。

```
1 print("Hello, world!")
```

如果你曾使用 C 或者 Objective-C 写代码，那么 Swift 的语法不会让你感到陌生——在 Swift 语言当中，这一行代码就是一个完整的程序！你不需要为每一个功能导入单独的库比如输入输出和字符串处理功能。写在全局范围的代码已被用来作为程序的入口，所以你不再需要 `main()` 函数。同样，你也不再需要在每句代码后边写分号。

通过向你展示各种编程任务，这个概览会给你足够的信息来开始使用 Swift 进行开发。如果觉得这个概览不够详细，不要担心——这个概览所介绍的内容都会在本书的余下章节里进行详细解释。

> 为了更好的阅读体验，我们推荐你使用 Xcode 里的 Playground 打开本章内容，Playground 允许你编辑代码并立即看到代码的运算结果。
>
> 下载本章的 Playground （官网链接）

简单值

使用 `let` 来声明一个常量，用 `var` 来声明一个变量。常量的值在编译时并不要求已知，但是你必须为其赋值一次。这意味着你可以使用常量来给一

个值命名，然后一次定义多次使用。

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

常量或者变量必须拥有和你赋给它们的值相同的类型。不过，你并不需要总是显式地写出类型。在声明一个常量或者变量的时候直接给它们赋值就可以让编译器推断它们的类型。比如上面的例子，编译器就会推断 `myVariable` 是一个整型，因为它的初始值是一个整型。

如果初始值并不能提供足够的信息（或者根本没有提供初始值），就需要在变量的后边写出来了，用冒号分隔。

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

> 实验
>
> 创建一个常量并显式声明类型为 `Float` ，赋值为 4

值绝对不会隐式地转换为其他类型。如果你需要将一个值转换为不同的类型，需要使用对应的类型显示地声明。

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

> 实验
>
> 试试去掉最后一行的转换标记 `String` ，看看会有怎样的报错?

其实还有一种更简单的方法来把值加入字符串：将值写在圆括号里，然后再在圆括号的前边写一个反斜杠 （ \ ） ，举个例子：

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of frui
t."
```

> 实验
>
> 使用 \() 来把一个浮点计算包含进字符串，然后再在一个欢迎语句中插入某人的名字。

为字符串使用三个双引号（ """ ）来一次输入多行内容。只要每一行的缩进与末尾的引号相同，这些缩进都会被移除。比如说：

```
let quotation = """
I said "I have \(apples) apples."
And then I said "I have \(apples + oranges) pieces of fruit."
"""
```

使用方括号（ [] ）来创建数组或者字典，并且使用方括号来按照序号或者键访问它们的元素。

```
var shoppingList = ["catfish", "water", "tulips", "blue pain
t"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

使用初始化器语法来创建一个空的数组或者字典。

```
let emptyArray = [String]()
let emptyDictionary = [String: Float]()
```

如果类型信息能被推断，那么你就可以用[]来表示空数组，用[:]来表示空字典。举个栗子，当你给变量设置新的值或者传参数给函数的时候。

```
shoppingList = []
occupations = [:]
```

控制流

使用 `if` 和 `switch` 来做逻辑判断，使用 `for-in`， `for`， `while`，以及 `repeat-while` 来做循环。使用圆括号把条件或者循环变量括起来不再是强制的了，不过仍旧需要使用花括号来括住代码块。

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

在一个 `if` 语句当中，条件必须是布尔表达式——这意味着比如说 `if score {...}` 将会报错，不再隐式地与零做计算了。

你可以一起使用 `if` 和 `let` 来操作那些可能会丢失的值。这些值使用可选项表示。可选的值包括了一个值或者一个 `nil` 来表示值不存在。在一个值的类型后边使用问号（ ? ）来把某个值标记为可选的。

```
var optionalString: String? = "Hello"
print(optionalString == nil)

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

> 实验
>
> 把 `optionalName` 的值改为 `nil`。你得到的 greeting 是什么内容？添加 `else` 分句，如果 `optionalName` 为 `nil` 就设置不同的内容给 greeting。

如果可选项的值为 `nil`，则条件为 `false` 并且花括号里的代码将会被跳过。否则，可选项的值就会被展开且赋给 `let` 后边声明的常量，这样会让展开的值对花括号内的代码可用。

另一种处理可选值的方法是使用 ?? 运算符提供默认值。如果可选值丢失，默认值就会使用。

```
let nickName: String? = nil
let fullName: String = "John Appleseed"
let informalGreeting = "Hi \(nickName ?? fullName)"
```

Switch 选择语句支持任意类型的数据和各种类型的比较操作——它不再限制于整型和测试相等上。

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

> 实验
>
> 尝试去掉 `default` 选项。会得到什么样的报错?

注意 let 可以用在模式里来指定匹配的值到一个常量当中。

在执行完 switch 语句里匹配到的 case 之后，程序就会从 switch 语句中退出。执行并不会继续跳到下一个 case 里，所以完全没有必要显式地在每一个 case 后都标记 break 。
你可以使用 `for-in` 来遍历字典中的项目，这需要提供一对变量名来储存键值对。字典使用无序集合，所以键值的遍历也是无序的。

```
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
        }
    }
}
print(largest)
```

> 实验
>
> 添加另一个变量来追踪哪一类的数字是最大的，同时那个最大的数是多少。

使用 `while` 来重复代码快直到条件改变。循环的条件可以放在末尾，这样可以保证循环至少运行了一次。

```
var n = 2
while n < 100 {
    n = n * 2
}
print(n)

var m = 2
repeat {
    m = m * 2
} while m < 100
print(m)
```

你可以使用 `..<` 来创造一个序列区间：

```
var total = 0
for i in 0..<4 {
    total += i
}
print(total)
```

使用 `..<` 来创建一个不包含最大值的区间，使用 `...` 来创造一个包含最大值和最小值的区间。

函数和闭包

使用 `func` 来声明一个函数。通过在名字之后在圆括号内添加一系列参数来调用这个方法。使用 `->` 来分隔形式参数名字类型和函数返回的类型。

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

> 实验
>
> 移除 `day` 形式参数。添加一个参数来包含今日午餐吃了什么。然后在欢迎语句里显示出来。

默认情况下，函数使用他们的形式参数名来作为实际参数标签。在形式参数前可以写自定义的实际参数标签，或者使用 `_` 来避免使用实际参数标签。

```
func greet(_ person: String, on day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet("John", on: "Wednesday")
```

使用元组来创建复合值——比如，为了从函数中返回多个值。元组中的元素可以通过名字或者数字调用。

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: I
nt, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}
```

```
let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
print(statistics.sum)
print(statistics.2)
```

函数同样可以接受多个参数，然后把它们存放进数组当中。

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

> 实验
>
> 写一个计算它接受到参数的平均数的函数

函数可以内嵌。内嵌的函数可以访问外部函数里的变量。你可以通过使用内嵌函数来组织代码，以避免某个函数太长或者太过复杂。

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

函数是一等类型，这意味着函数可以把函数作为值来返回。

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

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

函数也可以把另外一个函数作为其自身的参数。

```
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(list: numbers, condition: lessThanTen)
```

函数其实就是闭包的一种特殊形式：一段可以被随后调用的代码块。闭包中的代码可以访问其生效范围内的变量和函数，就算是闭包在它声明的范围之外被执行——你已经在内嵌函数的栗子上感受过了。你可以使用花括号（ `{}` ）括起一个没有名字的闭包。在闭包中使用 `in` 来分隔实际参数和返回类型。

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

> 实验
>
> 重写这个闭包来为所有奇数返回零。

你有更多的选择来把闭包写的更加简洁。当一个闭包的类型已经可知，比如说某个委托的回调，你可以去掉它的参数类型，它的返回类型，或者都去掉。

单语句闭包隐式地返回语句执行的结果。

```
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
```

你可以调用参数通过数字而非名字——这个特性在非常简短的闭包当中尤其有用。当一个闭包作为函数最后一个参数出入时，可以直接跟在圆括号后边。如果闭包是函数的唯一参数，你可以去掉圆括号直接写闭包。

```
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
```

对象和类

通过在 `class` 后接类名称来创建一个类。在类里边声明属性与声明常量或者变量的方法是相同的，唯一的区别的它们在类环境下。同样的，方法和函数的声明也是相同的写法。

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

> 实验
>
> 使用 `let` 添加一个常量属性，添加另一个方法接收一个参数。

通过在类名字后边添加一对圆括号来创建一个类的实例。使用点语法来访问实例里的属性和方法。

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

这个 `Shape` 类的版本缺失了一些重要的东西：一个用在创建实例的时候来设置类的初始化器。使用 `init` 来创建一个初始化器。

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

注意使用 `self` 来区分 `name` 属性还是初始化器里的 `name` 参数。创建类实例的时候给初始化器传参就好像是调用方法一样。每一个属性都需要赋值——要么在声明的时候（比如说 `numberOfSides`），要么就要在初始化器里赋值（比如说 `name`）。

使用 `deinit` 来创建一个反初始化器，如果你需要在释放对象之前执行一些清理工作的话。

声明子类就在它名字后面跟上父类的名字，用冒号分隔。创建类不需要从什么标准根类来继承，所以你可以按需包含或者去掉父类声明。

子类的方法如果要重写父类的实现，则需要使用 `override` ——不使用 `override` 关键字来标记则会导致编译器报错。编译器同样也会检测使用 `override` 的方法是否存在于父类当中。

```
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()
```

> 实验
>
> 创建另一个 NamedShape 的子类，名为 Circle ，它接收半径和名称作为其初始化器的参数。并在 Circle 类里实现一个 area() 和一个 simpleDescription() 方法。

除了存储属性，你也可以拥有带有 getter 和 setter 的计算属性。

```
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 triangle with sides of length
\(sideLength)."
    }
}
```

```
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a
triangle")
print(triangle.perimeter)
triangle.perimeter = 9.9
print(triangle.sideLength)
```

在 `perimeter` 的 setter 中，新值被隐式地命名为 `newValue` 。你可以提供一个显式的名字放在 set 后边的圆括号里。

注意 `EquilateralTriangle` 类的初始化器有三个不同的步骤：

1. 设定子类声明的属性的值；
2. 调用父类的初始化器；
3. 改变父类定义的属性中的值，以及其他任何使用方法，getter 或者 setter 等需要在这时候完成的内容。

如果你不需要计算属性但仍然需要在设置一个新值的前后执行代码，使用 `willSet` 和 `didSet` 。比如说，下面的类确保三角形的边长始终和正方形的边长相同。

```
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, nam
e: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "a
nother test shape")
print(triangleAndSquare.square.sideLength)
print(triangleAndSquare.triangle.sideLength)
```

```
triangleAndSquare.square = Square(sideLength: 50, name: "lar
ger square")
print(triangleAndSquare.triangle.sideLength)
```

当你操作可选项的值的时候，你可以在可选项前边使用 ？比如方法，属性和下标脚本。如果 ？前的值是 `nil` ，那 ？后的所有内容都会被忽略并且整个表达式的值都是 `nil` 。否则，可选项的值将被展开，然后 ？后边的代码根据展开的值执行。在这两种情况当中，表达式的值是一个可选的值。

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

枚举和结构体

使用 `enum` 来创建枚举，类似于类和其他所有的命名类型，枚举也能够包含方法。

```
enum Rank: Int {
    case ace = 1
    case two, three, four, five, six, seven, eight, nine, te
n
    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.rawValue)
        }
    }
}
let ace = Rank.ace
```

```
let aceRawValue = ace.rawValue
```

> 实验
>
> 写一个函数通过对比它们的原始值来对比两个 Rank 值

默认情况下，Swift 从零开始给原始值赋值后边递增，但你可以通过指定特定的值来改变这一行为。在上边的栗子当中，原始值的枚举类型是 `Int`，所以你只需要确定第一个原始值。剩下的原始值是按照顺序指定的。你同样可以使用字符串或者浮点数作为枚举的原始值。使用 `rawValue` 属性来访问枚举成员的原始值。

使用 `init?(rawValue:)` 初始化器来从一个原始值创建枚举的实例。

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

枚举成员的值是实际的值，不是原始值的另一种写法。事实上，在这种情况下没有一个有意义的原始值，你根本没有必要提供一个。

```
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()
```

> 实验

> 添加一个 `color()` 方法到 `Suit`，为黑桃和梅花返回“black”，为红桃和方片返回“red”。

注意有两种方法可以调用枚举的 `hearts` 成员：当给 `hearts` 指定一个常量时，枚举成员 `Suit.hearts` 会被以全名的方式调用因为常量并没有显式地指定类型。在 Switch 语句当中，枚举成员可以通过缩写的方式 `.hearts` 被调用，因为 `self` 已经明确了是 suit。你可以在任何值的类型已经明确的场景下使用使用缩写。

如果枚举拥有原始值，这些值在声明时确定，就是说每一个这个枚举的实例都将拥有相同的原始值。另一个选择是让case与值关联——这些值在你初始化实例的时候确定，这样它们就可以在每个实例中不同了。比如说，考虑在服务器上请求日出和日落时间的case，服务器要么返回请求的信息，要么返回错误信息。

```
enum ServerResponse {
    case result(String, String)
    case failure(String)
}

let success = ServerResponse.result("6:00 am", "8:09 pm")
let failure = ServerResponse.failure("Out of cheese.")

switch success {
case let .result(sunrise, sunset):
    print("Sunrise is at \(sunrise) and sunset is at \(sunse
t).")
case let .failure(message):
    print("Failure...  \(message)")
}
```

> 实验
>
> 添加第三个 case 到 `ServerResponse` 和 switch。

注意现在日出和日落时间是从 `ServerResponse` 值中以switch case 匹配的形式取出的。

使用 `struct` 来创建结构体。结构体提供很多类似与类的行为，包括方法和初始化器。其中最重要的一点区别就是结构体总是会在传递的时候拷贝其自身，而类则会传递引用。

```
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()
```

> 实验
>
> 给 `Card` 添加一个方法来创建一整副扑克牌，并且把每张牌的 rank 和 suit 对应起来。

协议和扩展

使用 `protocol` 来声明协议。

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

类，枚举以及结构体都兼容协议。

```
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
```

> 实验
>
> 给 ExampleProtocol 添加另外一个要求。如果要让 SimpleClass 和 SimpleStructure 依旧遵循这个协议，你需要做什么？

注意使用 mutating 关键字来声明在 SimpleStructure 中使方法可以修改结构体。在 SimpleClass 中则不需要这样声明，因为类里的方法总是可以修改其自身属性的。

使用 extension 来给现存的类型增加功能，比如说新的方法和计算属性。你可以使用扩展来使协议来别处定义的类型，或者你导入的其他库或框架。

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

> 实验
>
> 给 Double 类型写一个扩展，添加 absoluteValue 属性。

你可以使用协议名称就像其他命名类型一样——比如说，创建一个拥有不同类型但是都遵循同一个协议的对象的集合。当你操作类型是协议类型的值的时候，协议外定义的方法是不可用的。

```
let protocolValue: ExampleProtocol = a
```

```
print(protocolValue.simpleDescription)
// print(protocolValue.anotherProperty) // Uncomment to see t
he error
```

尽管变量 `protocolValue` 有 `SimpleClass` 的运行时类型，但编译器还是把它看做 `ExampleProtocol` 。这意味着你不能访问类在这个协议中扩展的方法或者属性。

错误处理

你可以用任何遵循 `Error` 协议的类型来表示错误。

```
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}
```

使用 `throw` 来抛出一个错误并且用 `throws` 来标记一个可以抛出错误的函数。如果你在函数里抛出一个错误，函数会立即返回并且调用函数的代码会处理错误。

```
func send(job: Int, toPrinter printerName: String) throws ->
String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

有好几种方法来处理错误。一种是使用 `do-catch` 。在 `do` 代码块里，你用 `try` 来在能抛出错误的函数前标记。在 `catch` 代码块，错误会自动赋予名字 `error` ，如果你不给定其他名字的话。

```
do {
    let printerResponse = try send(job: 1040, toPrinter: "Bi
 Sheng")
    print(printerResponse)
```

```
} catch {
    print(error)
}
```

> 实验
>
> 改变 printer 的名字为 "Never Has Toner"，好让 send(job:toPrinter:) 函数抛出一个错误。

你可以提供多个 catch 代码块来处理特定的错误。你可以在 catch 后写一个模式，用法和 switch 语句里的 case 一样。

```
do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
```

> 实验
>
> 添加一些代码来在 do 代码块里抛出一个错误。你要抛出什么样的错误才能让第一个 catch 代码块处理到？第二个，第三个呢？

另一种处理错误的方法是使用 try? 来转换结果为可选项。如果函数抛出了错误，那么错误被忽略并且结果为 nil 。否则，结果是一个包含了函数返回值的可选项。

```
let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")
let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner")
```

1.

使用 defer 来写在函数返回后也会被执行的代码块，无论是否错误被

抛出。你甚至可以在没有错误处理的时候使用 `defer` ，来简化需要在多处地方返回的函数。

```
var fridgeIsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]

func fridgeContains(_ food: String) -> Bool {
    fridgeIsOpen = true
    defer {
        fridgeIsOpen = false
    }

    let result = fridgeContent.contains(food)
    return result
}
fridgeContains("banana")
print(fridgeIsOpen)
```

泛型

把名字写在尖括号里来创建一个泛型方法或者类型。

```
func makeArray<Item>(repeating item: Item, numberOfTimes: Int
) -> [Item] {
    var result = [Item]()
    for _ in 0..<numberOfTimes {
        result.append(item)
    }
    return result
}
makeArray(repeating: "knock", numberOfTimes:4)
```

你可以从函数和方法同时还有类，枚举以及结构体创建泛型。

```
// Reimplement the Swift standard library's optional type
enum OptionalValue<Wrapped> {
    case none
    case some(Wrapped)
```

```
}
var possibleInteger: OptionalValue<Int> = .none
possibleInteger = .some(100)
```

在类型名称后紧接 `where` 来明确一系列需求——比如说，来要求类型实现一个协议，要求两个类型必须相同，或者要求类必须继承自特定的父类。

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

> 实验
>
> 修改 `anyCommonElements(_:_:)` 函数来返回一个 两个数组中共有元素 的数组。

写 `<T: Equatable>` 和 `<T> ... where T: Equatable` 是完全相同的。

《Swift 概览》上有5条评论

Pingback: スウィフトパーソナルスタディノート - プログラマーのフロントライン

Pingback: Swift 个人学习笔记 – 01: A Swift Tour – x

Pingback: Swift 個人學習筆記 - 01: A Swift Tour - 程序員的後花園