> 本教程基于 Swift 官网翻译，目前版本为 5.1
>
> 官网教程链接
>
> 2020-02-12

Swift 教程（Swift Tour）

使用新语言的第一个程序一般都会建议在屏幕上打印“ Hello，world！”字样。而在Swift中，这可以用一行代码完成：

```
print("Hello, world!")
// 打印 Hello, world!
```

如果您用过C或Objective-C，则此语法看起来很熟悉。而在Swift中，这行代码就是完整的程序。您无需导入单独的库即可实现输入/输出或字符串处理等功能。将在全局范围内编写的代码作为程序的入口点，因此不需要 `main()` 函数。也不需要在每个语句的末尾写分号。

本教程通过展示如何完成各种编程任务，来给开始在Swift中编写代码提供了足够的信息。如果您不了解某些内容，请不要担心，本教程的其余部分将详细介绍本教程中介绍的所有内容。

> **注意**
>
> 为了获得最佳体验，请在Xcode的playground中打开本章。在playground中，您可以编辑代码清单并立即查看结果。
>
> 下载 playground

简单值

使用 `let` 声明一个常数， `var` 声明一个变量。常量的值在编译时不需要知道，但是您必须为它赋值一次。这样就能用常量来命名那些只赋值一次但在许多地方都使用的值。

```
var myVariable = 42
myVariable = 50
let myContant = 30
```

常量或变量的类型必须与要分配给它的值的类型相同。但是，也不必总是显式地写明类型。在创建常量或变量时提供一个值可让编译器推断其类型。在上面的示例中，编译器将其推断 `myVariable` 为整数，因为其初始值为整数。

如果初始值不能提供足够的信息（或者没有初始值），可以通过在变量后写一个类型（用冒号分隔）来指定类型。

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

> 实验
>
> 创建一个显式类型为 `Float` 且值为的常数 `4` 。

```
let a : Double = 4
```

值永远不会隐式转换为另一种类型。如果需要将值转换为其他类型，请显式创建所需类型的实例。

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

> 实验
>
> 尝试 `String` 从最后一行删除转换。你得到什么错误?
>
> `Binary operator '+' cannot be applied to operands of type 'String' and 'Int'`

```
Binary operator '+' cannot be applied to operands of type 'String' and 'Int'
```

有一种甚至更简单的方法可以向字符串中添加值：在括号中写值，并在括号 `()` 前写反斜杠 `\` 。例如：

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

> 实验
>
> 用于 `\()` 在字符串中包含浮点计算，并在问候语中包含某人的姓名。

对于占用多行的字符串，请使用三个双引号（ `"""` ）。只要每个引用行的缩进都与右引号的缩进匹配，就将其删除。例如：

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

```
I said "I have 3 apples."
And then I said "I have 8 pieces of fruit."
```

使用方括号（`[]`）创建数组和字典，并通过在方括号中写入索引或键来访问元素。最后一个元素后允许使用逗号。

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

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

数组随着添加元素而自动增长。

```
shoppingList.append("blue paint")
print(shoppingList)
```

要创建一个空数组或字典，请使用构造函数语法。

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

如果可以推断类型信息，则可以将一个空数组编写为`[]`，将空字典编写为，`[:]`例如，当您为变量设置新值或将参数传递给函数时。

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

控制流

`if`和`switch`用于条件语句，`for-in`，`while`和`repeat-while用于循环。条件或循环变量的括号是可选的。`其中的程序要用`{}`括起来。

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
 if score > 50 {
   teamScore += 3
 } else {
   teamScore += 1
 }
}
print(teamScore)
// Prints "11"
```

在一条`if`语句中，条件必须是布尔表达式——这意味着类似`if score { ... }`的代码是错误，因为不能与零进行隐式比较。

可以使用`if`和`let`一起用于可能缺少的值。这些值表示为可选值。可选值包含一个值或包含`nil`——表示一个指示值不存在。在值的类型后写一个问号（`?`）表示将该值标记为可选。

```
var optionalString: String? = "Hello"
print(optionalString == nil)
// Prints "false"
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
 greeting = "Hello, \(name)"
}
```

> 实验
>
> 更改 `optionalName` 为 `nil` 。你会得到什么问候？添加一个 `else` 子句，如果 `optionalName` 是 `nil` 则设置不同的问候语。

如果可选值为 `nil` ，则条件为 `false` 并且花括号中的代码将被跳过。否则，通过 `let` 将可选值解包分配给常量，以便在花括号内使用。

处理可选值的另一种方法是使用 `??` 运算符提供默认值。如果缺少可选值，则使用默认值。

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

Switch 语句支持任何类型的数据和各种比较操作而非限于整数和相等性测试。

```
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.")
}
// Prints "Is it a spicy red pepper?"
```

> 实验
>
> 尝试删除默认情况。你得到什么错误？
>
> `Switch must be exhaustive`

注意上面代码中第7行中 `let` 的用法，用来将与模式匹配的值分配给常量。

当某个 case 匹配后会执行case中的代码，然后程序将从switch语句退出。而不会继续检查下一个 case，因此不需要在每个 case的代码末尾显式使用 break 退出 switch。

`for - in` 通过为每个键值对提供一对名称来遍历字典中的项目。字典是无序集合，因此其键和值以任意顺序进行迭代。

```
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)
// Prints "25"
```

实验

类似求所有值中的最大值，添加另一个变量用来找出各个类别的最大值。

代码如下：

```
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 {
 var largestOfKind = 0
 for number in numbers {
  if number > largest {
   largest = number
  }
  if number > largestOfKind {
   largestOfKind = number
  }
 }
 print("The largest number in \(kind) is \(largestOfKind)")
}
print("The largest number is \(largest)")
```

运行结果：（可能顺序有差异）

```
The largest number in Square is 25
The largest number in Fibonacci is 8
The largest number in Prime is 13
The largest number is 25
```

使用 `while` 重复执行一段代码，直到条件发生变化。循环的条件也可以在末尾，以确保循环至少运行一次。

```
var m = 2
repeat {
    m *= 2
} while m < 100
print(m)
// Prints "128"
```

您可以通过使用 `..<` 为循环创建一个包含在两个值之间的索引范围。

```
var total = 0
for i in 0..<4 {
    total += i
}
print(total)
// Prints "6"
```

使用 `..<` 省略了上限值，使用 `...` 则包含上限值。

函数和闭包

使用 `func` 声明函数。通过在函数名称后加上括号中的参数列表来调用函数。使用 `->` 将参数名称和类型与函数的返回类型分开。

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

> 实验
>
> 删除 `day` 参数。添加一个参数以在问候语中包括今天的特色午餐。

默认情况下，函数使用其参数名称作为其参数的标签。可以在参数名称之前写一个自定义参数标签，或只写 `_` 为不使用任何参数标签。

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

greet("John", on: "Wednesday")
```

使用元组（tuple）生成复合值，例如，从函数返回多个值。元组的元素可以通过名称或数字来引用。

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, 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)
// Prints "120"
print(statistics.0)
// Prints "3"
```

函数可以嵌套。嵌套函数可以访问在外部函数中声明的变量。在代码很长或者很复杂时可以使用嵌套函数。

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

returnFifteen()
```

函数也是一种类型。就是说一个函数可以作为值返回给另一个函数。

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

var increment = makeIncrementer()
increment(7) // 原文就写到这里，但是直接调试时，由于没有后文的使用会有警告

print("the result increment(7) is \(increment(7))")
// Print "the result increment(7) is 8"
```

一个函数还可以是另一个函数的参数。

```
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)
```

函数实际上是闭包的一种特殊情况。所谓**闭包**是指可以稍后调用的代码块。闭包中的代码可以访问在创建闭包的作用域中可用的变量和函数等内容，即使闭包在执行时位于不同的作用域中。在嵌套函数的示例中已经可以看到类似的用法。不带名称的闭包只需要用花括号（`{}`）括起来。使用 `in` 将参数和返回类型与程序体分开。

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

> 实验
>
> 重写闭包，让所有的奇数都返回 0。
>
> ```
> numbers.map({ (number: Int) -> Int in
>   let result = number % 2 == 0 ? result : 0
>   return result
> })
> ```

您有几种选择可以更简洁地编写闭包。当已知闭包的类型（例如委托的回调）时，可以省略其参数的类型，返回类型或两者。单个语句的闭包隐式返回该语句的值。

```
var numbers = [20, 19, 7, 12]
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
// Prints "[60, 57, 21, 36]"
```

> 上面这段代码值得好好学习，很优雅。

可以通过数字而不是名称来引用参数——这种方法在很短的闭包中特别有用。作为最后一个参数传递给函数的闭包可以在括号（注：谁的括号？）后立即显示。如果闭包是函数的唯一参数，则可以完全省略括号。

```
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
// Prints "[20, 19, 12, 7]"
```

对象和类

使用 `class` 关键字加上类的名称来创建一个类。类中的属性声明与常量或变量声明的编写方式相同，只不过它是在类的上下文中编写的。同样，方法和函数声明的编写方式相同。

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

> **实验**
>
> 用 `let` 添加一个常量属性，并添加另一个带有参数的方法。

通过在类名称后加上括号来创建类的实例。使用点语法访问实例的属性和方法。

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

这个 `Shape` 类还缺少一个重要的内容：创建实例时用于设置类的构造函数（注：构造函数）。可以使用 `init` 创建一个。

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

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

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

注意区分构造函数中的 `self.name` 所代表的属性和 `name` 所代表的参数。创建类的实例时，构造函数的参数像函数调用一样传递。每个属性都需要在其声明（如 `numberOfSides`）或构造函数（如 `name`）中分配一个值。

如果需要在释放对象之前执行一些清理，请使用 `deinit` 来创建一个析构函数。

子类可以在类名之后加上冒号（`:`）后再加上父类的名字。所有的标准根类（standard root class）不需要继承任何类，因此可以根据需要包含或忽略父类。

子类中覆写父类的方法需要使用 `override` 进行标记，不带 `override`，编译器会报错。编译器还会检测那些带有 `override` 但实际上没有覆写父类中任何方法的方法。

```
class Square: NamedShape {
```

```
2
3     var sideLength: Double
4
5     init(sideLength: Double, name: String) {
6       self.sideLength = sideLength
7       super.init(name: name)
8       numberOfSides = 4
9     }
10
11    func area() -> Double {
12      return sideLength * sideLength
13    }
14
15    override func simpleDescription() -> String {
16      return "A square with sides of length \(sideLength)."
17    }
18  }
19
20  let test = Square(sideLength: 5.2, name: "my test square")
21  test.area()
22  test.simpleDescription()
23
```

> **实验**
>
> 制作另一个 `NamedShape` 的子类 `Circle` 。它以半径和名称作为其初始值设定项的参数。在类中实现 `area()` 和 `simpleDescription()` 方法。

除了存储的简单属性外，属性还可以具有 *getter* 和 *setter* 。

```
1   class EquilateralTriangle: NamedShape {
2
3     var sideLength: Double = 0.0
4
5     init(sideLength: Double, name: String) {
6       self.sideLength = sideLength
7       super.init(name: name)
8       numberOfSides = 3
9     }
10
11    var perimeter: Double {
12      get {
13        return 3.0 * sideLength
14      }
15      set {
16        sideLength = newValue / 3.0
17      }
18    }
19
20    override func simpleDescription() -> String {
21      return "An equilateral triangle with sides of length \(sideLength)."
```

```
    }
}

var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
print(triangle.perimeter)
// Prints "9.3"

triangle.perimeter = 9.9
print(triangle.sideLength)
// Prints "3.3000000000000003"
```

在 `perimeter` 的 *setter* 中，新值具有隐式名称 `newValue` 。您可以在 *set* 后面的花括号中提供一个明确的名称。

请注意， `EquilateralTriangle` 类的构造函数有三个不同的步骤：

1. 设置子类中声明的属性的值。
2. 调用父类的构造函数。
3. 更改父类定义的属性的值。此时还可以完成对方法、 *getter* 或 *setter* 等其他设置工作。

如果您不需要对属性进行计算，但仍然需要提供在设置新值之前和之后运行的代码，可以使用 `willSet` 和 `didSet` 。当构造函数外属性值发生改变时就会运行这些代码。例如，下面的类确保其三角形的边长始终与其正方形的边长相同。

```
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, name: name)
 }
}

var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
print(triangleAndSquare.square.sideLength)
// Prints "10.0"
print(triangleAndSquare.triangle.sideLength)
// Prints "10.0"
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
print(triangleAndSquare.triangle.sideLength)
// Prints "50.0"
```

使用可选值时，可以在对方法、属性和下标之类的操作之前编写 `?` 。如果之前的 `?` 值为 `nil` ，则 `?` 后面的一切都会被忽略，整个表达式的值为 `nil` 。否则，可选值解包，并且 `?` 之后的所有内容都会起作用。在这两种情况下，整个表达式的值都是一个可选值。

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

枚举和结构体

使用 `enum` 创建一个枚举。像类和所有其他命名类型一样，枚举可以具有与之关联的方法。

```
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.rawValue)
  }
 }
}

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

> 实验
>
> 编写一个通过比较两个原始值来比较两个值的 `Rank` 的函数。

默认情况下，Swift会分配从零开始的原始值，并每次递增一，但是您可以通过显式指定值来更改此行为。在上面的示例中， `Ace` 显式指定了的原始值 `1` ，其余原始值按顺序分配。您还可以使用字符串或浮点数作为枚举的原始类型。使用 `rawValue` 属性访问枚举中各种情况（case）的原始值。

使用 `init?(rawValue:)` 构造函数从原始值创建枚举实例。它返回与原始值匹配的枚举情况，或者在没有匹配的 `Rank` 时返回 `nil` 。

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

枚举的case值是实际值，而不仅仅是原始值的另一种方法。实际上，若不存在有意义的原始值，就不必提供原始值。

```
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()
```

> **实验**
>
> 给 `Suit` 添加一个 `color()` 方法，对于黑桃和草花返回“ *black* ”，对于红心和方块返回“ *red* ”。

请注意 `hearts` 上面应用中的两种方式：为常量 `hearts` 分配值时，使用 `Suit.hearts` 完整引用，因为常量没有指定显式类型。在 *switch* 内部，枚举用缩写形式 `.hearts` 表示，因为 `self` 已经明确为 `Suit` 了。只要知道值的类型，就可以使用缩写形式。

如果枚举具有原始值，则将这些值确定为声明的一部分，这意味着特定枚举用例的每个实例始终具有相同的原始值。另一种枚举用例的选择方法是让值与用例关联——这些值是在创建实例时确定的，并且对于枚举用例的每个实例而言都可以不同。可以把关联值的行为看做类似于枚举案例实例的存储属性。例如，考虑从服务器请求日出和日落时间的情况。服务器要么以请求的信息作为响应，要么以错误的描述作为响应。

```
enum ServerResponse {
 // 括号里面必须指定关联值类型，不能没有
 case result(String, String)
 case failure(String)
}
//把一个常量赋值为枚举中的某个值，如：success就指明为 result
let success = ServerResponse.result("6:00 am", "8:09 pm")
let failure = ServerResponse.failure("Out of cheese.")

// switch 后面的常量要和上面 let 中的一致，
// case 即是判断这个常量究竟是枚举中的哪个值，例子中的是 result

switch success {
case let .result(sunrise, sunset):
```

```
    // 匹配后，可以自动析取两个参数的值
    print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
  case let .failure(message):
    print("Failure... \(message)")
  }

  // Prints "Sunrise is at 6:00 am and sunset is at 8:09 pm."
```

> 实验
>
> 在 `ServerResponse` 的switch中添加第三种情况。

请注意，如何从 `ServerResponse` 值中提取日出和日落时间，以作为与switch-case中匹配的值的一部分。

使用 `struct` 创建结构体。结构体支持许多与类相同的行为，包括方法和构造函数。结构和类之间最重要的区别之一是，结构在代码中传递时始终会被复制，而类是通过引用传递的。

```
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
>
> 编写一个函数，该函数返回一个包含一整套纸牌（共52张）的数组，使用rank表示牌点，使用suit表示花色。

协议和扩展

使用 `protocol` 关键字来声明协议。

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

类、枚举和结构体都可以采用协议。

```
class SimpleClass: ExampleProtocol {
  var simpleDescription: String = "A very simple class."
  var anotherProperty: Int = 69105
```

```
4
5       func adjust() {
6         simpleDescription += "  Now 100% adjusted."
7       }
8     }
9
10    var a = SimpleClass()
11    a.adjust()
12    let aDescription = a.simpleDescription
13
14    struct SimpleStructure: ExampleProtocol {
15      var simpleDescription: String = "A simple structure"
16      mutating func adjust() {
17        simpleDescription += " (adjusted)"
18      }
19    }
20
21    var b = SimpleStructure()
22    b.adjust()
23    let bDescription = b.simpleDescription
```

> 实验
>
> 向 `ExampleProtocol` 中添加新的需求（注：如新的成员变量或者方法）。如何修改 `SimpleClass` 和 `SimpleStructure` 使其仍然符合协议?

请注意，在 `SimpleStructure` 的声明中使用了关键字 `mutating` 来标记用于修改结构的方法。而 `SimpleClass` 则不需要，因为类上的方法始终可以修改该类。

使用 `extension` 关键字向现有类型添加功能，例如新方法和计算属性。您可以使用扩展将协议一致添加到在其他地方声明的类型，甚至添加到从库或框架导入的类型。

```
1     extension Int: ExampleProtocol {
2       var simpleDescription: String {
3         return "The number \(self)"
4       }
5
6       mutating func adjust() {
7         self += 42
8       }
9     }
10
11    print(7.simpleDescription)
12
13    // Prints "The number 7"
```

> 实验
>
> 为 `Double` 类型添加 `absoluteValue` （绝对值）属性编写扩展。

可以像使用任何其他命名类型一样使用协议名称，例如，创建具有不同类型但都符合一个协议的对象的集合。当使用类型为协议类型的值时，协议定义之外的方法不可用。

```
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription)
// Prints "A very simple class.  Now 100% adjusted."
// print(protocolValue.anotherProperty)  // Uncomment to see the error
```

即使变量 `protocolValue` 在运行时类型为 `SimpleClass` ，编译器也将其视为给定类型 `ExampleProtocol` 。这意味着除了协议一致性之外，您不能意外访问该类实现的方法或属性。

错误处理

可以使用适用 `Error` 协议的任何类型来表示错误。

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

使用 `throw` 抛出一个错误，使用 `throws` 标记一个函数具有抛出错误的功能。如果在函数中抛出错误，该函数将立即返回，并且调用该函数的代码将处理该错误。

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

有几种处理错误的方法。一种方法是使用 `do - catch` 。其中，可以通过在前面写入 `try` 来标记可能引发错误的代码。在 `catch` 块内，除非为错误指定其他名称，错误名称会自动指定为 `error` 。

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

// Prints "Job sent"
```

> 实验
>
> 将打印机名称更改为 `"Never Has Toner"` ，让函数 `send(job:toPrinter:)` 引发错误。

您可以为处理特定错误提供多个 `catch` 块。编写方式与switch中的case语句的编写方式一样。

```
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)
}

// Prints "Job sent"
```

> **实验**
>
> 在 `do` 块内添加引发错误的代码。您需要引发哪种错误，来让第一个 `catch` 块处理？那第二块和第三块呢？

处理错误的另一种方法是 `try?` 用于将结果转换为可选的。如果函数抛出错误，则特定的错误将被丢弃，结果为 `nil` 。否则，结果是一个可选值，其中包含函数返回的值。

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

用 `defer` 写的是在函数中的所有其它代码后执行代码块，只是在函数返回之前。无论函数是否引发错误，都将执行代码。 `defer` 即使需要在不同的时间执行设置和清除代码，也可以使用它们彼此相邻。

```
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)
// Prints "false"
```

泛型

在一对尖括号内写一个名称，以构成函数或类型的泛型。

```swift
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)
```

泛型可用于函数、方法、类、枚举和结构体。

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

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

可以在程序体前使用 `where` 关键字来指定需求列表，例如，要求类型实现协议，要求两个类型相同或要求一个类具有特定的父类。

```swift
func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> Bool
  where T.Element: Equatable, T.Element == U.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(_:_:)` 函数使其返回任何两个序列共有的元素的数组。

`<T: Equatable>` 的写法与 `<T> ... where T: Equatable` 的写法是一样的。