For-In 循环

Summer

> 这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

Swift 提供了多种控制流结构。其中包含 `while` 循环来执行多次任务；`if`、 `guard` 和 `switch` 语句来执行特定条件下不同的代码分支； 还有 `break` 和 `continue` 语句使控制流跳转到你代码中的其他位置。

Swift 还提供了 `for-in` 循环用来更简便的遍历数组（arrays）， 字典（dictionaries），区间（ranges），字符串（strings），和其他序列类型。

Swift 的 `switch` 语句比其他的类 C 语言更加强大。case 可以匹配多种不同的模式，包括间隔匹配（interval matches），元祖（tuples），和转换到特定类型。`switch` 语句的 case 体中匹配的值可以绑定临时常量或变量，在每个 case 中也可以使用 `where` 来实现更复杂的匹配情况。

victoria 翻译于 5个月前

可以使用 `for-in` 循环来遍历序列中的所有元素，例如数组中的所有元素，数字的范围，或者字符串的字符。

下面是使用 `for-in` 循环遍历数组的例子：

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}
```

你也可以通过遍历一个字典来访问它的键值对。遍历字典时其中的每个元

素都会返回成 `(key, value)` 元组（Tuple）的形式， 你也可以在 `for-in` 循环中显式的命名常量来分解 `(key, value)` 元组。 在下面的例子中，字典中的值（Key）被分解为 `animalName` 常量，字典中的值（Value）被分解为 `legCount` 常量。

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
```

`Dictionary` 内部的内容是无序的，所以遍历的顺序不确保和之后重新获取的顺序一致。 特别说明一下，你插入到 `Dictionary` 中的顺序不能决定它真正的顺序。想要了解更多数组（Array）和字典（Dictionary）的相关内容请查看 集合类型.

victoria 翻译于 5个月前

你也可以通过数值范围来进行 `for-in` 循环. 这个例子打印了 `5乘乘法表` 中的前几条记录:

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
```

例子中的遍历区间为使用闭区间操作符 (...) 定义的，包括 1 到 5 在内的数字区间。 index 被定为区间的第一个数字 (1),然后循环内的语句开始执行。在例子中，循环只有一条语句，输出了 5乘乘法表 中当前 index 对应的值。语句执行后，index 更新为区间中的第二个值 (2)，然后函数 print(_:separator:terminator:) 被再次调用。当到达区间结尾，进程结束。

上面的例子中，常数 index 的值在每次循环开始时都会自动赋值。因此，index 不需要在使用前进行声明。只要声明循环时，包含了该常量，就会对其进行隐式声明，不需要使用声明关键词 let 。

如果你不需要使用区间中的所有值，你可以使用 _ 替代变量名来忽略对应的值。

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
print("\(base) to the power of \(power) is \(answer)")
```

allen 翻译于 5个月前

上面的例子计算的是一个数的几次幂（在这个例子中是 3 的 10 次幂）。它从 1 （即 3 的 0 次幂）开始乘 3 ，使用从 1 到 10 的闭区间来确保乘了

十次。对于这个计算每次循环出的值是没有意义的---只是为了循环执行正确的次数。 在循环的变量处使用下划线字符（_）是为了忽略当前值，并且在每次循环的迭代期间不提供对当前值的访问。

在一些情况中你可能不想使用包含两个端点的闭区间。想象在手表表面上画每分钟的刻度标记。你想要从 0 分钟开始画 60 个刻度标记。可以使用半开区间操作符（..<）来包含下界但不包含上界。更多关于区间的内容可以查看 区间运算符.

```
let minutes = 60
for tickMark in 0..<minutes {

}
```

一些用户在他们的界面上可能想要更少的刻度标记。他们可能更喜欢每 5 分钟一个刻度。使用 stride(from:to:by:) 函数可以跳过不需要的标记。

```
let minuteInterval = 5
for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {

}
```

通过 stride(from:through:by:) 使用闭区间也是可以的：

```
let hours = 12
let hourInterval = 3
for tickMark in stride(from: 3, through: hours, by: hourInterval) {

}
```

While 循环

一个 while 循环会一直执行一组语句直到条件变为 false 。这类循环最适合第一次循环前不知道循环的次数的情况。Swift 提供两种类型的 while 循环：

- `while` 在每次循环开始时判断条件。
- `repeat-while` 在每次循环结束时判断条件。

victoria 翻译于 5个月前

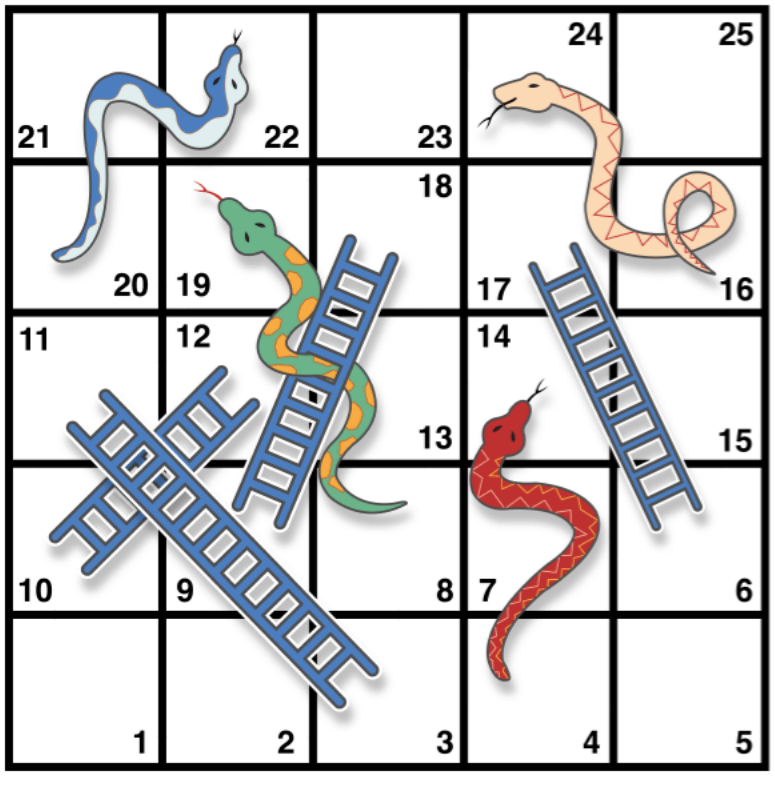
While

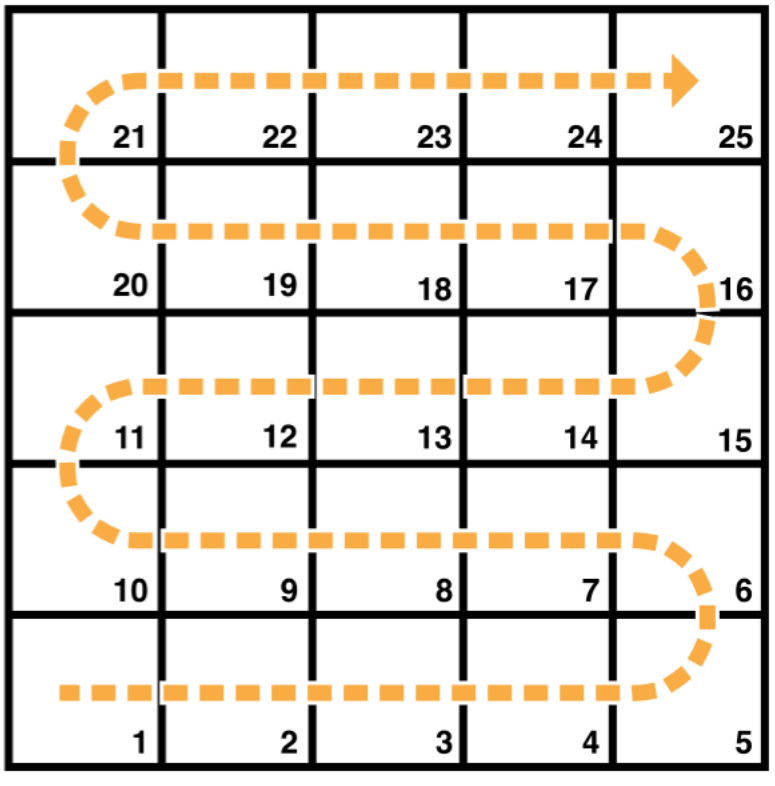
`while` 循环从判断一个单一条件开始。如果条件为 `true`，会重复运行一系列语句，直到条件为 `false` 。

这是 `while` 循环的格式：

```
while condition {
    statements
}
```

下面的例子来玩一个叫 *蛇和梯子* （也叫做 *斜道和梯子* ）的小游戏：





游戏规则如下：

- 棋盘包含 25 个方格，目标是达到或者超过第 25 个方格。
- 玩家起始方格是「0号方格」，它位于棋盘的左下角。
- 每轮你都要通过投掷一枚六面骰来确定那你移动方格的步数，移动的路线遵循上图虚线箭头指示的路径。
- 每轮结束，如果你在梯子底部，就顺着梯子爬上去。
- 每轮结束，如果你在蛇的头部，就顺着蛇身体滑下去。

victoria 翻译于 5个月前

棋盘可以用 `Int` 型数组来表示。数组的大小是基于名为 `finalSquare` 的常量，使用这个常量来初始化数组并且在之后判断是否赢得游戏。因为游戏的起点「0号方格」在棋盘外，所以棋盘用 26 个值为零的 `Int` 型初始化而不是 25 个。

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
```

然后一些方格被设置成更具体的值来表示蛇和梯子。梯子底部的方格带有一个正数使你在棋盘上向上移动，而蛇头部的方格带有一个负数使你在棋盘上向下移动。

```
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
```

3 号方格是梯子的底部可以让你向上移动到 11 号方格。为了表示这个，`board[03]` 的值等于 `+08`，也就是整数值 `8`（表示 `3` 到 `11`的差值）。为了对齐这些值和语句，一元正运算符（`+i`）明确的与一元负运算符（`-i`）一起使用，小于 `10` 的数字用 0 填充对齐。（风格上的调整并不是必须的，只是为了让代码更加整洁。）

```
var square = 0
var diceRoll = 0
while square < finalSquare {

    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }

    square += diceRoll
    if square < board.count {

        square += board[square]
    }
}
print("Game over!")
```

victoria 翻译于 5个月前

这个例子中使用了最简单的方式来模拟投掷骰子。没有使用随机数而是使用了从 `0` 开始的变量 `diceRoll` 。每次 `while` 循环 `diceRoll` 的值增加一然后判断它的值是否变得太大。当它的值等于 `7` 时，就超过了骰子的最大值，所以重置为 `1` 。 `diceRoll` 值的顺序将一直会是 `1, 2, 3, 4, 5, 6, 1, 2` 等。

投掷骰子后，玩家移动 `diceRoll` 数量的方格。结果可能让玩家移动到第 25 个方格之外，从而结束游戏。为了应对这种情况代码会判断 `square` 的值小于 `board` 数组的 `count` 属性值。如果 `square` 是有效的，会在 `board[square]` 值上加当前 `square` 的值让玩家顺着梯子爬上去或者顺着蛇滑下去。

> 注意
>
> 如果没有这个判断， `board[square]` 可能会越界访问 `board` 导致运行时错误。

当前 `while` 循环执行结束，再次检查循环条件，判断循环是否应该再次执行。如果玩家移动到或者超出 `25` 号方格，循环条件结果变为 `false` 游戏

结束。

`while` 循环比较适合在这个例子中使用，因为在 `while` 循环前不确认游戏的长度，只能循环到满足特定的条件为止。

victoria 翻译于 5个月前

Repeat-While

另一种 `while` 循环，称作 `repeat-while` 循环，它会在判断循环条件 *之前*，先执行一次循环体。然后重复执行循环体直至条件变为 `false`。

> 注意：
>
> Swift 里的 `repeat-while` 循环类似于其他语言里的 `do-while` 循环。

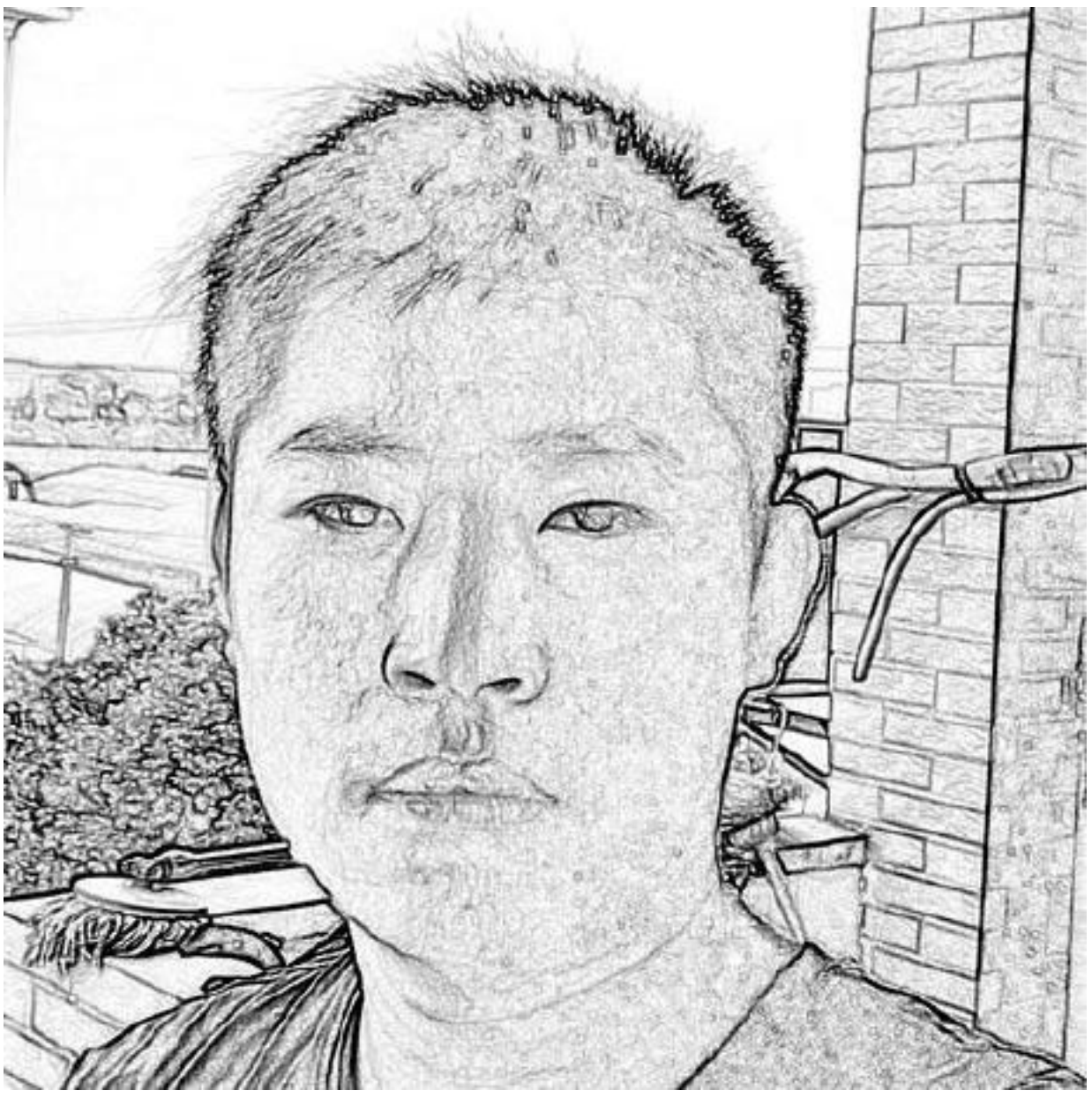
下方是 `repeat-while` 循环的一般形式：

```
repeat {
    statements
} while condition
```

下方是用 repeat-while 循环代替 while 循环，编写的 *蛇和梯子* 有一个例子。 finalSquare，board，square，和 diceRoll 的值初始化方式和 while 循环里的相同。

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
var diceRoll = 0
```

在这个版本的游戏里，循环的 *第一步* 就是检查是梯子还是蛇。没有梯子就让玩家直接上到第 25 方格，所以玩家不能通过移动梯子就赢得游戏。因此，在循环第一步检查是蛇还是梯子是安全的。



suiyun 翻译于 5个月前

游戏开始时，玩家在「0 号方格」。board[0] 一直等于 0，不会有什么影响。

```
repeat {

    square += board[square]

    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }

    square += diceRoll
} while square < finalSquare
print("Game over!")
```

在代码确认过蛇和梯子之后，投掷骰子玩家移动 `diceRoll` 数量的方格。当前循环结束。

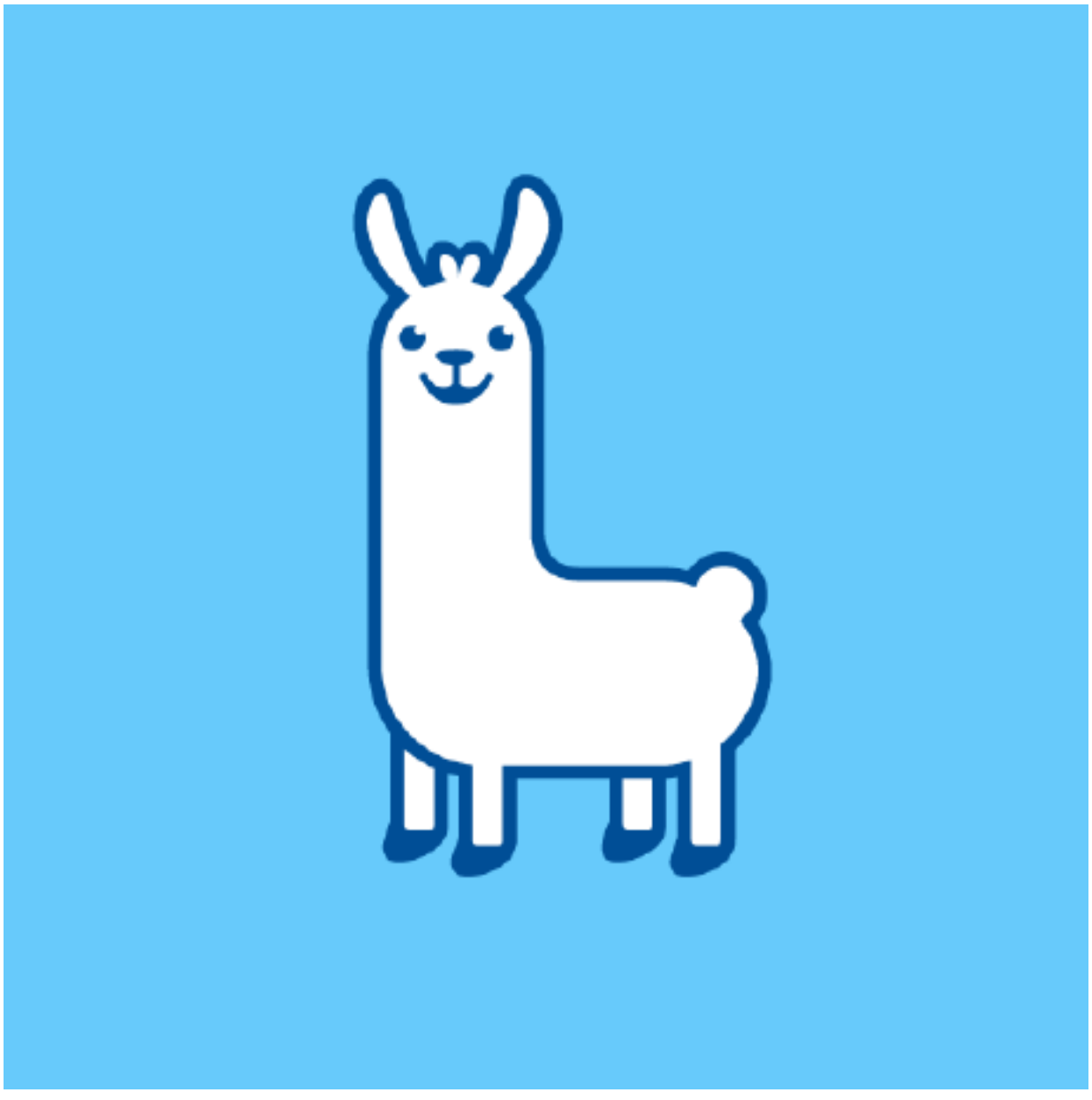
循环条件 (`while square < finalSquare`) 和前面的一样，但这次在第一次循环 *结束* 后才会进行判断。 `repeat-while` 循环结构对于这个游戏比前面 `while` 循环的例子更加合适。在上面的 `repeat-while` 循环中，在 `while` 条件确认 `square` 仍在棋盘上之后会 *立即执行* `square += board[square]` 操作。 这种行为让我们不用像之前 `while` 循环的版本那样判断数组的边界。

victoria 翻译于 5个月前

条件语句

很多时候我们需要根据不同条件来执行不同代码段。比如，当发生错误时我们想要运行一段额外的代码，或者当某个值太大或太小时我们想展示一段信息。要实现这些，就需要使用 *条件语句*。

Swift 提供两种条件语句：`if` 语句和 `switch` 语句。通常，使用 `if` 语句来执行结果可能性较少的简单条件；`switch` 语句则更适合于有较多组合的更复杂的条件，而且，当需要使用模式匹配来判断执行合适的代码段时，`switch` 语句会更有用。



lukeee 翻译于 5个月前

If

`if` 语句最简单的形式只有一个 `if` 条件，而且只有当这个条件为 `true` 时才会执行对应的代码。

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
}
```

上面的例子判断温度是否小于或等于 32 华氏度（即水的冰点）。 假如低于 32 华氏度，就会打印一条信息；否则不会打印任何消息，并且会继续执行 `if` 语句块之后（即 `if` 语句右大括号之后）的代码。

当 `if` 条件为 `false` 时，`if` 语句提供了一个可选分支语句，叫做 *else 分支语句*。*else 分支语句* 使用 `else` 关键字进行声明。

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
```

这两个分支中肯定有一个会被执行。由于温度高达 40 华氏度，并没有冷到需要建议人们戴围巾，所以 `else` 分支会被触发。

lukeee 翻译于 5个月前

你可以将 `if` 语句串联起来，组成一系列分支语句。

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
```

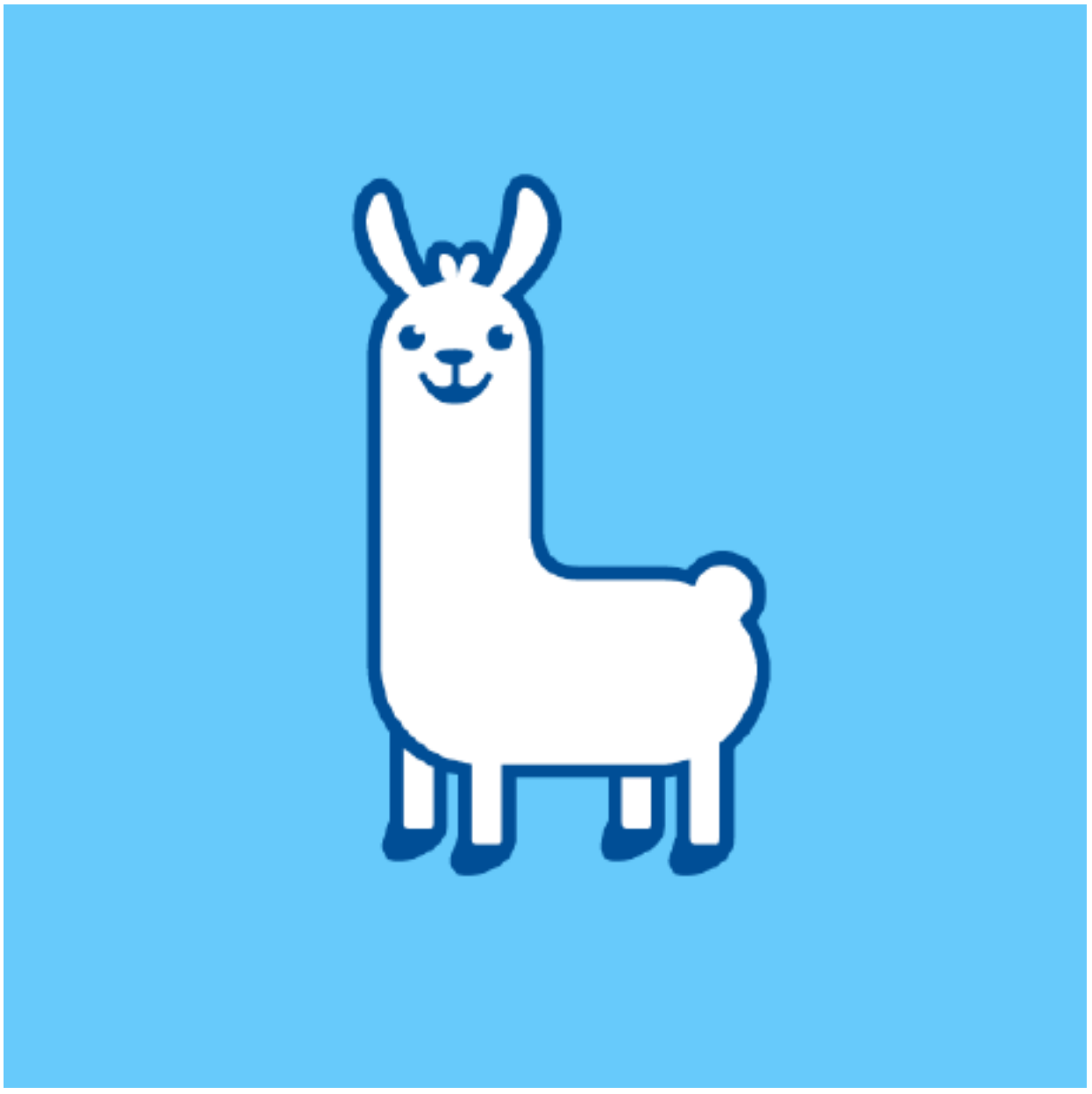
这样，另一个 `if` 语句就被添加好了，它用来对气温较高的情况做出反应。最后的 `else` 分支语句仍然存在，它在气温不太高也不太低时打印消息。

最后的 `else` 分支语句是可选的，在条件不需要十分完备时可以省略。

```
temperatureInFahrenheit = 72
```

```
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
}
```

因为气温不高不低，不会触发上面例子中的 `if` 和 `else if` 语句，所以不会打印任何消息。



lukeee 翻译于 5个月前

Switch

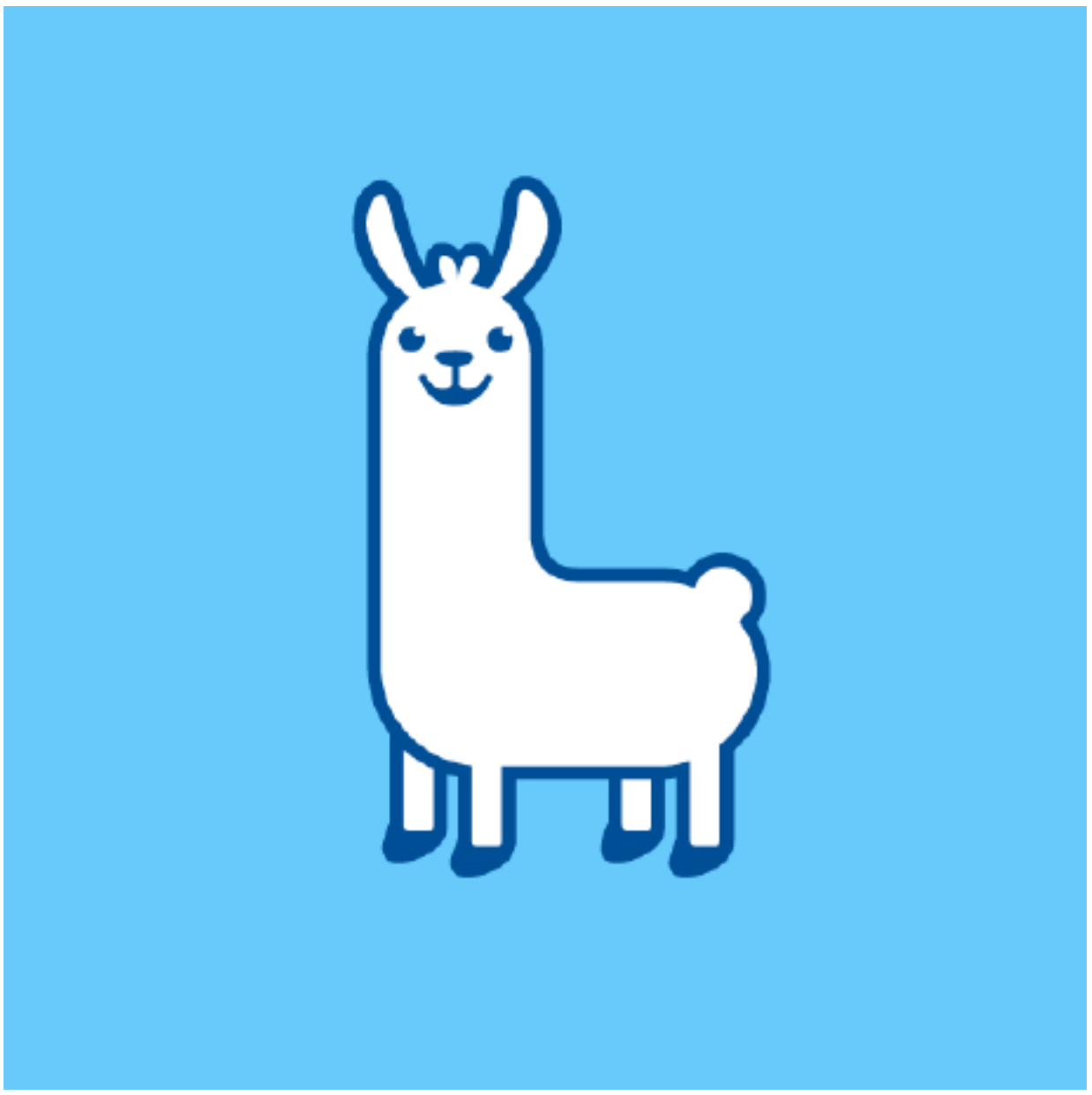
`switch` 语句会将某一个值与其它几种可能匹配的模式进行比较，然后它会执行第一个匹配成功的模式下对应的代码块。当可能的情形非常多时，应该使用 `switch` 语句替代 `if` 语句。

`switch` 语句最简单的形式是将一个值和另外一个或几个同类型的值进行比较。

```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,
     value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

每个 switch 语句都由多个可能的 *情况* 组成，每种 *情况* 都以关键字 case 开头。为了对比某些特定的值，Swift 提供了几种不同的方法来指定更加复杂的匹配模式。这些将在本章节的稍后部分提及。

像 if 语句一样，在 switch 语句中每一个 case 都是一个单独的代码执行分支，而 switch 语句会根据具体情况决定哪个分支被执行。这个过程会根据给定值进行条件 *匹配* 。



每个 switch 语句必须是 *可穷尽的*。 也就是说，判断的类型的每个可能的值都要有一个 switch 的分支（case）与之对应。为每个可能的值创建一个分支是不合理的，你可以定义一个默认分支来覆盖没有单独处理的其他所有值。这个默认分支使用 default 关键字声明，并且必须放在最后。

下面例子使用 switch 语句匹配名为 someCharacter 的单个小写字符：

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
    print("The first letter of the alphabet")
case "z":
    print("The last letter of the alphabet")
default:
    print("Some other character")
}
```

这个 switch 语句的第一个分支匹配英文字母表的第一个字母 a ，第二个分支匹配最后一个字母 z 。因为 switch 必须有一个分支覆盖所有可能的字符，不止是字母表中的字符，所以这个 switch 语句使用 default 分支匹配除了 a 和 z 以外的所有字符。这样就保证了这个 switch 语句的穷尽性。

victoria 翻译于 5个月前

不存在隐式的贯穿

与 C 语言和 Objective-C 中的 `switch` 语句相反，Swift 中的 `switch` 语句在执行完一个分支后不会「贯穿」到下一个分支。相反，整个 `switch` 语句一旦完成第一个匹配的 `switch` 分支就会结束，而不需要明确的 `break` 语句。这使得 Swift 中的 `switch` 语句比 C 语言中的更加安全、易用，并且避免了错误地执行多个 `switch` 分支的情况。

> 注意
>
> 虽然在 Swift 中 `break` 不是必须的，你可以使用 `break` 语句来匹配和忽略特定的分支或者或者在分支全部执行前跳出。更多细节，查看 Switch 语句中的 Break.

每一个分支中 *必须* 包含至少一个可执行的语句。以下代码是无效的，因为第一个分支是空的：

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a":
case "A":
    print("The letter A")
default:
    print("Not the letter A")
}
```

victoria 翻译于 5个月前

这个 `switch` 语句不会像 C 语言一样同时匹配 `"a"` 和 `"A"`。相反的，它在编译时时会报错：`case "a":` 不包含任何可执行语句。这种方法可以避免一个分支意外贯穿到另一个分支，也使代码更安全，意图更清晰。

用逗号将两个值分开，组合成一个复合分支语句，可以使 `switch` 语句中单个 `case` 分支同时匹配 `"a"` 和 `"A"`。

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a", "A":
    print("The letter A")
default:
    print("Not the letter A")
}
```

为了可读性， 复合分支语句可以写成多行，更多关于复合分支语句的信息，参考 复合分支语句。

> 注意
>
> 使用 `fallthrough` 关键字，可以显式的贯穿特定的 `case` 分支，参考 贯

穿。

allen 翻译于 5个月前

区间匹配

switch 中分支匹配的值也可以是一个区间。这个例子使用数字区间来匹配任意数字对应的自然语言格式：

```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
let naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")
```

上述例子中， approximateCount 在 switch 语句中被估值。它与每一个 case 分支中的数字区间进行匹配。因为 approximateCount 落在 12 到 100 的区间， 所以 naturalCount 被赋值为 「 dozens of 」， 然后 switch 语句跳出执行。

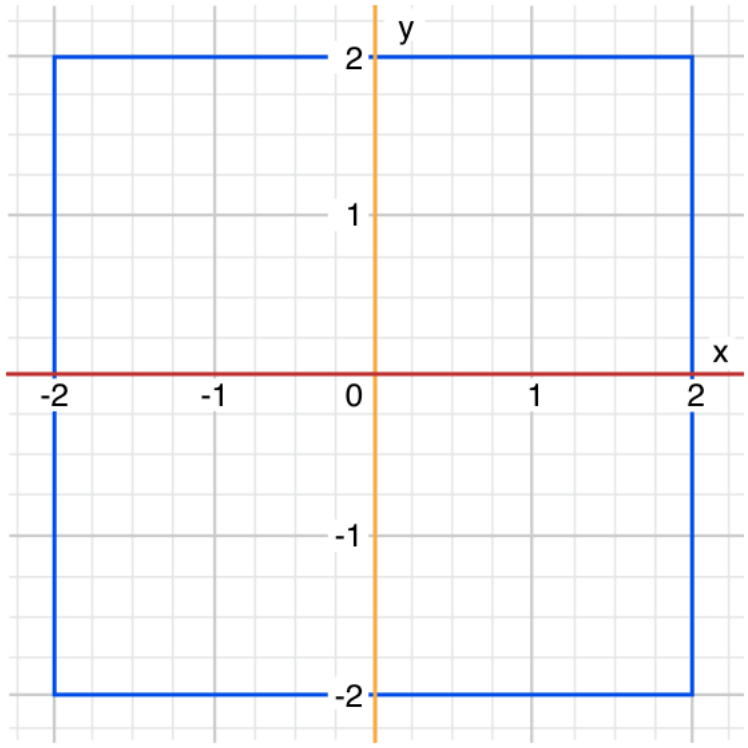
allen 翻译于 5个月前

元组

你可以使用元组在同一个 `switch` 语句中测试多个值。可以针对不同的值或值的间隔来测试元组的每个元素。或者使用下划线（_）来匹配任何可能的值，这也被称为通配符模式。

下面的示例声明了一个 (x, y) 点，该变量是类型为 `(Int, Int)` 的元组，并将其显示在示例后面的图上。

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
```



`switch` 语句确定该点是在原点 (0, 0)，红色 x 轴上，橙色 y 轴上，蓝色 4 × 4 的格子内，还是在格子外面。

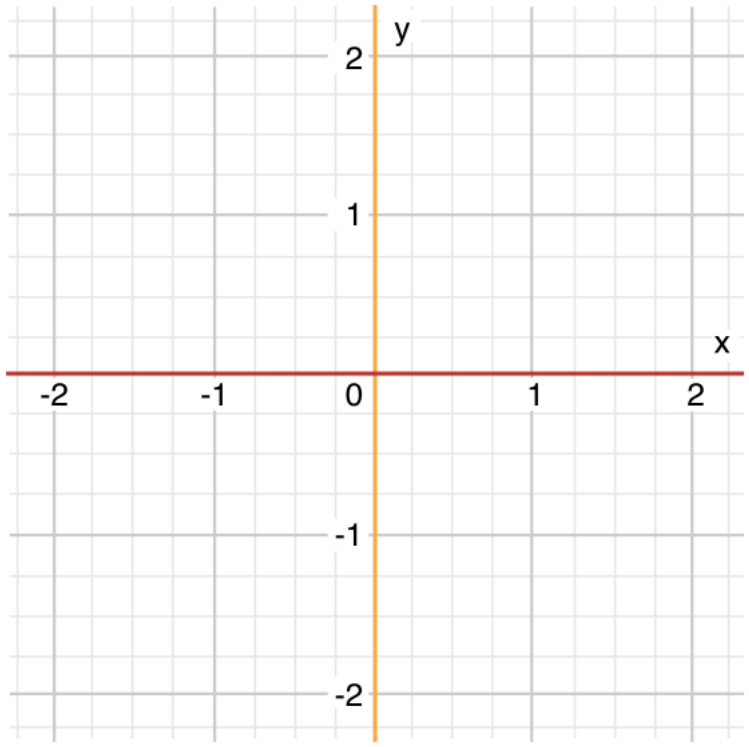
与 C 语言不同，Swift 允许同一个值符合多个 `switch` 分支。实际上，在这个例子中，点 (0, 0) 匹配所有*四个*分支。但是，如果匹配多个分支，则始终使用第一个匹配的分支。点 (0, 0) 首先匹配 `case (0, 0)`，因此所有其他的匹配分支都被忽略。

值绑定

switch 分支可以将其匹配的一个值或多个值赋值给临时的常量或变量，常量或变量可以在 case 主体中使用。这个行为被称为*值绑定*，因为值在 case 主体中被绑定给临时的常量或变量。

下面的示例声明了一个 (x, y) 点，其类型为 (Int, Int) 的元组，并且该点展示在示例后面的图上：

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
```



switch 语句用来判断该点在红色 x 轴上，橙色 y 轴上，还是除了轴之外的其他地方。

三个 switch 分支声明了占位符常量 x 和 y，暂时从 anotherPoint 中获取一个或多个元组值。第一个分支 case (let x, 0) 匹配任何 y 值为 0 的点，并把 x 的值赋值给临时常量 x。同样地，第二个分支 case (0, let y) 匹配任何 x 值为 0 的点，并把 y 的值赋值给临时常量

y。

在声明临时常量之后，可以在 case 代码块中使用该常量。这里，它们用来打印点的分类。

这个 switch 语句没有 default 分支。在最后一个分支 case let (x, y) 中，声明了一个可以匹配任何值的有两个占位符常量的元组。因为 anotherPoint 是有两个值的元组，这个分支可以匹配剩余的任何值，并不需要 default 分支来使 switch 语句穷举。

wzshare 翻译于 5个月前

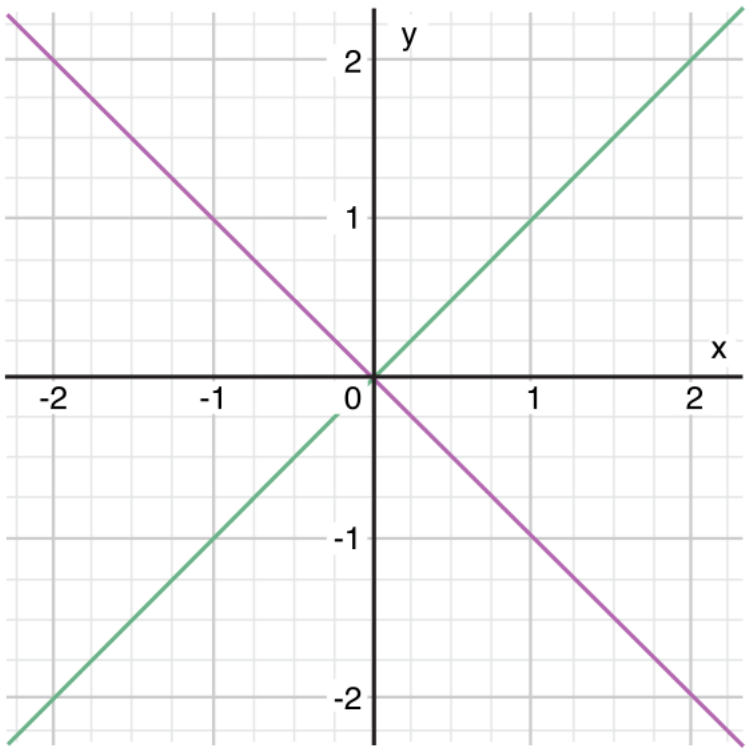
Where

switch 分支中可以使用 where 子句来检测额外的条件。

下面的示例展示把 (x, y) 点划分到随后的图上：

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    print("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
```

```
    print("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    print("(\(x), \(y)) is just some arbitrary point")
}
```



switch 语句用来判断该点在绿色对角线 x == y 上，还是紫色对角线 x == -y 上。

三个 switch 分支声明了占位符常量 x 和 y，它们从 yetAnotherPoint 中获取两个元组值。这些常量用作 where 子句的一部分，用来创建一个动态分类器。只有当 where 子句满足计算值为 true 时，switch 分支才匹配当前的 point。

与前一个示例一样，最后一个 case 匹配所有剩余的值，所以不需要 default 分支来使 switch 语句穷举。

wzshare 翻译于 5个月前

复合分支

在 case 后面写多个模式可以把多个分支共享在同一个主体中，每个模式用逗号隔开。如果任何一个模式匹配，那么这个分支就是匹配的。如果模式太多，可以把模式写为多行。比如：

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
     "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) is not a vowel or a consonant")
}
```

switch 语句的第一个分支匹配英语中的所有五个小写元音。同样，第二个分支匹配所有的小写辅音。最后，default 分支匹配其余字符。

复合分支也可以包含值绑定。复合分支的所有模式必须包含在同一组值绑

定中，并且每个绑定必须从复合分支的所有模式中获取相同类型的值。这样确保无论复合分支中哪个部分匹配，分支主体的代码总是可以访问绑定的值，并且确保值总是有相同的类型。

```
let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
case (let distance, 0), (0, let distance):
    print("On an axis, \(distance) from the origin")
default:
    print("Not on an axis")
}
```

上面的 case 中有两个模式：(let distance, 0) 匹配 x 轴上的点，(0, let distance) 匹配 y 轴上的点。两种模式都包含 distance 的绑定，distance 在两种模式中是一个整数，这意味着 case 主体中的代码总是可以访问 distance 的值。

wzshare 翻译于 5个月前

控制转移语句

*控制转移语句*通过将控制从一段代码转移到另一段代码来改变代码的执行顺序。Swift 中有五个控制转移语句：

- `continue`
- `break`
- `fallthrough`
- `return`
- `throw`

`continue`、 `break` 和 `fallthrough` 语句在下面有详细描述。 `return` 语句在 函数 中描述，`throw` 语句在 使用抛出函数传递错误 中描述。

Continue

`continue` 语句告诉循环停止正在做的事情，并在循环的下一次迭代开始时再启动。它仿佛在说「我完成了当前的循环迭代」而没有完全离开循环。

下面的示例从小写字符串中删除所有的元音和空格，并创建一个神秘的谜语：

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
let charactersToRemove: [Character] = ["a", "e", "i", "o", "u", " "]
for character in puzzleInput {
    if charactersToRemove.contains(character) {
        continue
    } else {
        puzzleOutput.append(character)
    }
}
print(puzzleOutput)
```

上面的代码只要匹配到元音或空格时就调用 `continue` 关键字，使循环的本次迭代立即结束并跳到下一次迭代的开始。

wzshare 翻译于 5个月前

Break

`break` 语句立即结束整个控制流语句的执行，当你想在 `switch` 或循环中提前结束时，可以在 `switch` 或循环中使用 `break` 语句。

在循环语句中使用 Break

在循环语句中使用时，`break` 立即结束循环的执行，并把控制转移到循环右括号（`}`）后面的代码上。不执行来自当前循环迭代的下一步代码，并且不再开始循环的迭代。

在 Switch 语句中使用 Break

在 `switch` 语句中使用时，`break` 会使 `switch` 语句立即结束执行，并把控制转移到 `switch` 语句的右括号（`}`）后面的代码上。

此行为可用于匹配和忽略 `switch` 语句中的一个或多个分支。 因为 Swift 的 `switch` 语句是穷举的并且不允许空分支，所以有时需要故意匹配并忽略一个分支使你的意图明确。 你可以将 `break` 语句作为要忽略的分支的

整个主体来使用。当该分支与 switch 语句匹配时，分支中的 break 语句使 switch 语句立即结束执行。

> 注意
>
> 如果 switch 分支只包含注释会报编译时错误。 注释不是语句，不会使 switch 分支被忽略。总是使用 break 语句来忽略 switch 分支。

wzshare 翻译于 5个月前

下面这个例子对一个 Character 类型的值进行分支，并判断其是否为四种语言中某个代表数字的符号。 简洁起见，每个 switch 语句包含了多个值。

```
let numberSymbol: Character = "三"
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "๑":
    possibleIntegerValue = 1
case "2", "٢", "二", "๒":
    possibleIntegerValue = 2
case "3", "٣", "三", "๓":
    possibleIntegerValue = 3
case "4", "٤", "四", "๔":
```

```
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
    print("The integer value of \(numberSymbol) is \(integerValue).")
} else {
    print("An integer value could not be found for \(numberSymbol).")
}
```

这个例子判断了变量 numberSymbol 是否为拉丁语、阿拉伯语、中文或泰语中代表数字 1 到 4 的某个符号。如果匹配成功， switch 语句中的一个分支将会把一个叫做 possibleIntegerValue 的 Int? 可选型变量设为相应的值。

在 switch 语句执行完成后，以上例子可以通过可选绑定来判断匹配是否成功。作为一个可选型变量， possibleIntegerValue 的初值为 nil ，只有当 possibleIntegerValue 被 switch 的前四个分支设为一个实际的值的时候，可选绑定才会成功。

因为无法将所有可能的 Character 值在以上例子中枚举出来， 一个 default 分支可以用来处理所有未被列举的情况。这个 default 分支不用做任何事，所以它的内容可以是一个简单的 break 语句。当匹配到 default 分支时，这 break 语句将会结束 switch 结构的执行，代码将会继续从 if let 语句开始执行。

owenlyn 翻译于 5个月前

贯穿

在 Swift 中， switch 语句的每个分支在判断结束后不会「贯穿」到下一个分支。即，整个 switch 语句会在第一个匹配的分支语句执行完成后终止。相反地，C 语言明确要求在每个 switch 分支结束时手动添加 break 语句来防止贯穿。相对而言，默认没有贯穿使得 Swift 中的 switch 语句更加简洁，可读性更强，并可以因此避免错误地执行多个 switch 分支。

如果需要像 C 语言中那样的贯穿行为，你可以在分支中逐个添加 fallthrough 关键字。 下面这个例子就利用了贯穿 fallthrough 来给数字添加描述。

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
```

owenlyn 翻译于 5个月前

查看其他 1 个版本

这个例子声明了一个新的名为 description 的字符串变量并初始化。接下来这个函数通过一个 switch 语句来给 integerToDescribe 变量赋值。如果 integerToDescribe 的值是一个被枚举出的质数，这个函数会在 description 末尾添加一段文字来标明这是一个质数。然后它利

用 fallthrough 关键字来「贯穿」 default 分支。这个 default 分支在 swift 语句结束前会在 description 的末尾再次添加一段文字。

除非 integerToDescribe 在被枚举出的数字中，否则它不会被 switch 语句匹配成功。因为没有其它的分支语句， default 分支将给 integerToDescribe 赋值。

在 switch 语句执行结束后， print(_:separator:terminator:) 函数会打印出该数字的描述。在这个例子里，数字 5 被正确地标记为质数。

> 注意
>
> fallthrough 关键字不会检查 switch 语句中下一个分支的条件，它只是让代码在执行的过程中直接进入下一个分支 (或 default 分支) 中的语句, 就像 C 语言中 switch 语句的标准行为。

owenlyn 翻译于 5个月前

带标签语句

在 Swift 中，你可以在循环体和条件语句中嵌套循环体和条件语句来创造复杂的控制流结构。并且，循环体和条件语句都可以使用 break 语句来提前结束整个代码块。因此，显式地指明 break 语句想要终止的是哪个循环体或者条件语句，会很有用。类似地，如果你有许多嵌套的循环体，显式指明 continue 语句想要影响哪一个循环体也会非常有用。

为了实现这个目的，你可以使用标签（ statement label ）来标记一个循环体或者条件语句，对于一个条件语句，你可以使用 break 加标签的方

式，来结束这个被标记的语句。对于一个循环语句，你可以使用 `break` 或者 `continue` 加标签，来结束或者继续这条被标记语句的执行。

声明一个带标签的语句是通过在该语句的关键词的同一行前面放置一个标签，作为这个语句的前导关键字（`introducor keyword`），并且该标签后面跟随一个冒号。下面是一个针对 `while` 循环体的标签语法，同样的规则适用于所有的循环体和条件语句。

```
label name: while condition {
    statements
}
```

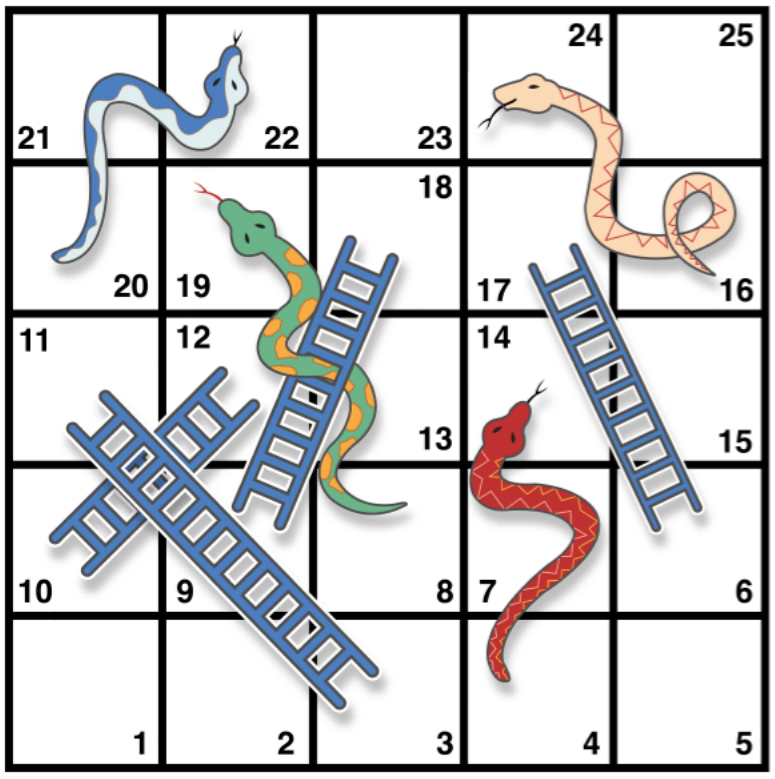
Kevin 翻译于 5个月前

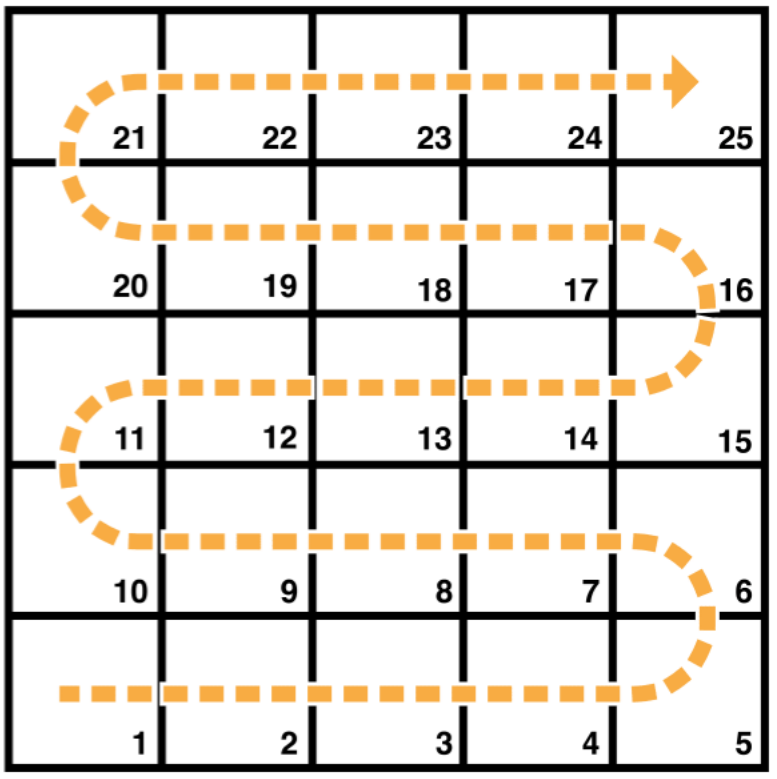
下面的例子是前面章节中蛇和梯子的适配版本，在此版本中，我们将使用一个带有标签的 while 循环体中调用 `break` 和 `continue` 语句。这次，游戏增加了一条额外的规则：

- 为了获胜，你必须刚好落在第 25 个方块中。

如果某次掷骰子使你的移动超出第 25 个方块，你必须重新掷骰子，直到你掷出的骰子数刚好使你能落在第 25 个方块中。

游戏的棋盘和之前一样：





finalSquare, board, square, 和 diceRoll 值被和之前一样的方式初始化：

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
var diceRoll = 0
```

这个版本的游戏使用 while 循环和 switch 语句来实现游戏的逻辑。 while 循环有一个标签名 gameLoop ， 来表明它是游戏的主循环。

该 while 循环体的条件判断语句是 while square != finalSquare ， 这表明你必须刚好落在方格25中。

```
gameLoop: while square != finalSquare {
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:

        break gameLoop
    case let newSquare where newSquare > finalSquare:

        continue gameLoop
    default:

        square += diceRoll
```

```
        square += board[square]
    }
}
print("Game over!")
```

Kevin 翻译于 5个月前

每次循环迭代开始时掷骰子。与之前玩家掷完骰子就立即移动不同，这里使用了 `switch` 语句来考虑每次移动可能产生的结果，从而决定玩家本次是否能够移动：

- 如果骰子数刚好使玩家移动到最终的方格里，游戏结束。 `break gameLoop` 语句跳转控制去执行 `while` 循环体后的第一行代码，意味着游戏结束。
- 如果骰子数将会使玩家的移动超出最后的方格，那么这种移动是无效的，玩家需要重新掷骰子。 `continue gameLoop` 语句结束本次 `while` 循环，开始下一次循环。
- 在剩余的所有情况中，骰子数产生的都是有效的移动。玩家向前移动 `diceRoll` 个方格，然后游戏逻辑再处理玩家当前是否处于蛇头或者梯子的底部。接着本次循环结束，控制跳转到 `while` 循环体的条件判断语句处，再决定是否需要继续执行下次循环。

> 注意
>
> 如果上述的 `break` 语句没有使用 `gameLoop` 标签，那么它将会中断 `switch` 语句而不是 `while` 循环。 `gameLoop` 标签清晰的表明了 `break` 想要中断的是哪个代码块。
>
> 同时请注意，当调用 `continue gameLoop` 去跳转到下一次循环迭代时，这里使用 `gameLoop` 标签并不是严格必须的。因为在这个游戏中，

> 只有一个循环体，所以 continue 语句会影响到哪个循环体是没有歧义的。然而， continue 语句使用 gameLoop 标签也是没有危害的。这样做符合标签的使用规则，同时参照旁边的 break gameLoop ，能够使游戏的逻辑更加清晰和易于理解。

Kevin 翻译于 5个月前

提前退出

guard 语句和 if 语句一样，根据表达式的布尔值执行语句。 使用 guard 语句要求条件必须为真才能执行 guard 语句之后的代码。 和 if 语句不同，guard 语句总是有一个 else 分支 --- 如果条件不为真，则执行 else 分支中的代码。

```
func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }

    print("Hello \(name)!")

    guard let location = person["location"] else {
        print("I hope the weather is nice near you.")
        return
    }

    print("I hope the weather is nice in \(location).")
}

greet(person: ["name": "John"])
```

```
greet(person: ["name": "Jane", "location": "Cupertino"])
```

如果满足 guard 语句的条件，则在 guard 声明的结束括号后继续执行代码。 当任何变量或常量在使用可选绑定作为条件被赋值后，它的值都可用于 guard 语句后的其余代码块。

如果不满足该条件，则执行 else 分支内的代码。 该分支必须转移控制以退出 guard 语句后的代码块。 它可以通过控制转移语句来执行此操作，例如 return， break， continue 或 throw，也可以调用一个无返回值的函数或方法，例如 fatalError(_:file:line:) 。

相比于使用 if 语句进行判断，使用 guard 语句可以提高代码的可读性。它可以让你编写出连贯执行的代码，而不必将其包装在 else 块中，并且让你更加从容地处理异常代码。

jihongboo 翻译于 5个月前

检测 API 可用性

Swift 内置支持检查 API 可用性，这可以确保我们不会在当前部署机器上，不小心地使用了不可用的 API。

编译器使用 SDK 中的可用信息来验证我们的代码中使用的所有 API 在项目指定的部署目标上是否可用。如果我们尝试使用一个不可用的 API，Swift 会在编译时报错。

我们在 `if` 或 `guard` 语句中使用 可用性条件（*availability condition*)去有条件的执行一段代码，来在运行时判断调用的 API 是否可用。编译器使用从可用性条件语句中获取的信息去验证，在这个代码块中调用的 API 是否可用。

```
if #available(iOS 10, macOS 10.12, *) {

} else {

}
```

以上可用性条件指定， `if` 语句的代码块仅仅在 iOS 10 或 macOS 10.12 及更高版本才运行。最后一个参数，`*`，是必须的，用于指定在所有其它平台中，如果版本号高于你的设备指定的最低版本， `if` 语句的代码块将会运行。

在它一般的形式中，可用性条件使用了一个平台名字和版本的列表。平台名字可以是 `iOS`，`macOS`，`watchOS`，和 `tvOS` ---请访问声明属性来获取完整列表。 请参阅 Declaration Attributes. 除了指定像 iOS 8 或 macOS 10.10 的大版本号，也可以指定像 iOS 11.2.6 以及 macOS 10.13.3 的小版本号。

```
if #available(platform name version, ..., *) {
     APIs 可用，语句将执行
} else {
    APIs 不可用，语句将不执行
}
```

Kevin 翻译于 5个月前

> 本文中的所有译文仅用于学习和交流目的，转载请务必注明文章译者、出处、和本文链接
> 我们的翻译工作遵照 CC 协议，如果我们的工作有侵犯到您的权益，请及时联系我们。

> 这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

函数 是执行一个具体任务的一段独立代码块，你可以通过为函数命名来标识其任务功能，当需要执行这个任务时，函数名就可以用来「调用」该函数。

Swift 的统一的函数语法非常灵活，以致于从一个简单、无参数名的 C 风格函数，到一个复杂、多参数名的 Objective-C 风格方法都可以灵活表达。参数可以通过设置一个默认值，以简化函数的调用。 也可以传递可修改参数，一旦函数完成执行，传递的参数值就会被修改。

Swift 中的函数类型由参数值类型和返回值类型共同组成。你可以像其它类型一样来使用这个类型，这样把一个函数做为参数传递给另一个函数就会非常容易，并且可以从其它函数来返回函数。另外，一个封装了具体功能的函数能直接嵌套在另一个函数的代码块中。

函数的定义和调用

在定义一个函数时，你可以可选地提供一个或多个输入值作为参数。当函

数执行完成时，你也能可选地提供一个值作为返回值。

每一个函数的 函数名 描述了这个这个函数需要做的事情。你可以通过这个函数名去调用它并为它提供符合参数类型的参数值。 函数的实参值必须按形参的参数列表顺序依次传入。

下面我们定义一个 `greet(person:)` 方法，它表示向一个人打招呼， 它接受一个 `String` 类型的值做为输入并返回一个 `String` 类型的值 。

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

函数名的定义应该清晰地指明这个函数的功能，函数的函数名前面必须要加 `func` 关键字。函数 `->`（向右的箭头）后面是返回值，`->` 后面跟返回值的类型。

上面方法的方法名描述了这个方法要做的事情、需要的参数值和当执行完成时返回的值。在其他地方调用时，这个方法清晰的表达了它的作用

```
print(greet(person: "Anna"))

print(greet(person: "Brian"))
```

调用 `greet(person:)` 时，传递一个 `String` 类型的人名作为值，例如这样 `greet(person: "Anna")` 。因为这个方法有一个返回值，所以像上面这样放在 `print()` 方法中，可以直接打印出执行完的结果。

> 注意：
>
> `print(_:separator:terminator:)` 方法在调用时除第一个参数外，其他值可以不传，因为其他参数有默认值。函数语法变化的讨论在这里
> Function Argument Labels and Parameter Names
> Default Parameter Values

`greet(person:)` 方法定义了一个 greeting 变量，并设置了一条简单的打

招呼信息。当函数执行完时，这个值通过函数的 return 关键字返回出去。

你可以多次调用 greet(person:) 方法通过传递不同值，上面例子展示了调用时分别传递 "Anna" 和 "Brian" 的情况。

为了简化函数体书写，我们可以把消息的创建和返回合并在一条语句中:

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
```

函数的参数和返回值

在 Swift 中，函数的参数和返回值是非常灵活的。你能定义任何事 无论是一个单一参数的简单函数 还是有着多个参数和不同参数选项的复杂函数。

无参函数

函数可以没有参数。 下面是一个没有参数的函数，当调用时，它总是返回同一个 String 类型的值：

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
```

函数在定义时，即使函数没有任何参数，函数名后也需要跟 () 圆括号。调用时，函数名后面跟一对空的 () 圆括号。

多个参数

函数也可以有多个参数，这些参数写在函数名后面的 () 内，参数之间通过，逗号分隔。

下面方法接受一个 `String` 类型的人名值和是否已经打过招呼的 `Bool` 值作为输入，返回一个给这个人打招呼的信息：

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
```

你能调用 `greet(person:alreadyGreeted:)` 函数 通过传递一个 `String` 类型的人名值 和一个 `Bool` 类型的值。用来和之前的 `greet(person:)` 函数做区分。 尽管他们函数名都是 greet ，但一个接受一个参数，而另一个接受两个。

无返回值函数

函数也可以没有返回值。这个 `greet(person:)` 函数版本就没有返回值，而是将结果直接打印出来：

```
func greet(person: String) {
    print("Hello, \(person)!")
}
greet(person: "Dave")
```

因为它不需要返回一个值，所以函数的定义可以不写 `->` 箭头及后面的返回值类型。

> 注意
>
> 严格意义来说，`greet(person:)` 函数 *仍然* 返回一个值，只是这个返回值没有被定义。函数返回值没有定义时，默认是返回 `Void` 类型。它是一个简单的空元祖，可以被写做 `()` 。

调用函数时，它的返回值可以被忽略：

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")

printWithoutCounting(string: "hello, world")
```

第一个函数 `printAndCount(string:)` 打印一个字符串，然后返回这个字符串的字符集数量。 第二个函数 `printWithoutCounting(string:)` 调用第一个函数，忽略了第一个函数的返回值。 所以当调用第二个函数时， 信息仍然被第一个函数打印了，但第一个函数的返回值确没被使用。

> 注意
>
> 返回值可以被忽略， 但函数的返回值还是需要接收。 一个有返回值的函数的返回值不允许直接丢弃不接收，如果你尝试这样做，编译器将给你抛出错误。

多返回值函数

你可以用一个元祖类型包装多个值来作为一个函数的返回值。

下面这个 `minMax(array:)` 函数，它找出参数数组中的最大整数和最小整数：

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

这个 `minMax(array:)` 函数返回一个包含了两个整型 `Int` 的元祖。 它们的键名分别是 `Max` 和 `Min`，所以你可以通过这个键名来访问这两个值。

`minMax(array:)` 方法先定义了两个变量 `currentMin` 和 `currentMax` ， 分别存储这数组中第一个元素。 然后迭代数组，检查每一个值是否比最小值小或比最大值大，如果是则分别记录这个值。 最后，找出的最小值和最大值被包装在一个元祖中返回。

因为这个元祖的成员值被作为函数返回值的一部分，最大值和最小值能通过点语法来直接被访问。

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
```

注意这个从函数返回的元祖的成员不需要在被指定键名，因为它们的键名已经被作为函数返回类型的一部分而指定。

可选元祖返回类型

如果从函数返回的元祖类型有可能为空， 你可以用一个可选元祖来指示这个返回值可能为 `nil` 。你可以在元祖返回类型后面加上一个 `?` 问号来表示返回值可能为空， 例如 `(Int, Int)?` 或 `(String, Int, Bool)?`。

> 注意
>
> 一个可选的元祖类型例如 `(Int, Int)?` 和元祖值可选例如 `(Int?, Int?)` 是不同的。 对于前者是整个元祖可能为空，而后者则是元祖内每一个独立的元素可能为空。

上面这个 `minMax(array:)` 函数返回了一个包含两个 `Int` 值的元祖。然而，这个函数没有对传递进来的数组进行任何安全性的检查。 如果这个数组为空， 这个 `minMax(array:)` 函数在尝试访问数组第一个元素时，将在运行时触发一个数组越界的错误。

为了处理空数组这种情况，将 `minMax(array:)` 方法的返回值标记为可选

类型。如果数组为空，将返回 `nil`：

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

你也能用可选值绑定的方法来检查 `minMax(array:)` 方法是否返回一个有效值：

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
```

函数的参数标签和参数名

每一个参数都由一个 *参数标签* 和一个 *参数名* 构成。 参数标签被用在调这个方法时； 每一个参数标签写在参数的前面。参数名被用在函数的具体实现中。 默认参数的参数标签可以不写，用参数名来作为参数标签。

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {

}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

所有的参数必须有一个唯一的名称。 尽管多个参数可以有相同的参数标签， 但唯一的参数标签将使你的代码可读性更好。

明确参数标签

参数标签在参数名前面， 通过一个空格 来分隔：

```
func someFunction(argumentLabel parameterName: Int) {

}
```

这个修改后的 `greet(person:)` 函数接收一个人的姓名和家乡名称，并返回一个 `String` 类型的 greeting：

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
```

多使用参数标签可以使函数在被调用时表达更清晰，可读性更好。

参数标签省略

如果一个参数不需要参数标签，可以用下划线 _ 来代替之前的参数标签。

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {

}
someFunction(1, secondParameterName: 2)
```

如果一个参数有一个明确的参数标签，在调用时这个标签 *必须* 被明确写明。

参数默认值

你可以给函数的任何参数提供一个默认值，通过写在参数类型后面。 如果提供了默认值，你就可以在调用时省略给这个参数传值。

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int =

}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6)
someFunction(parameterWithoutDefault: 4)
```

把没有默认值的参数放在有默认值的参数前面。 没有默认值的参数对函数更重要 --- 当调用时，把它们写在前面更容易区分具有部分相同参数的函数，无论默认参数是否被忽略。

可变参数

可变参数 接受 0 个或多个具体相同类型的值。调用时，你可以用一个可变参数来代表这些有着不确定数量的多个参数。在参数类型后面跟上 3 个点 `...` 来表示参数的数量可变。

传入函数体内的可变参数可以被当做一个数组类型来使用。 下面这个例子中，变量名 `numbers` 表示的一系列可变参数（每一个的类型为 `Double`）被合并成更合适的数组类型 `[Double]` 。

函数 `arithmeticMean()` 为传入的一系列数字计算出平均值：

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)

arithmeticMean(3, 8.25, 18.75)
```

> 注意
>
> 一个函数至多只能有一个可变参数。

传入传出参数

函数参数默认是常量，不能直接修改其值。编译器会报错如果你尝试在函数体内修改传入参数的值。 但如果你执意要修改这个参数值， 并希望在函数执行完成后修改的值仍然有效， 那么用 *传入传出参数* 来代替普通参数。

传入传出参数通过在参数类型前加上 inout 关键字来定义。传入传出参数可以有一个初始值， 传入函数后值将被修改，在函数执行完传出后，这个变量的初始值就会被替换完成。 更多有关传出传出参数的行为和编译器优化的详细讨论，请移步
In-Out Parameters.

传入传出参数只支持变量。常量或字面量将不被允许做为参数传递，因为它们都不能被修改。 传值时，在参数名前面加上 & 符号，来表示它能在函数体内被修改。

> 注意
>
> 传入传出参数不能有默认值，并且可变参数也不能被标记 inout 。

下面这个 swapTwoInts(_:_:) 函数，有两个参数名分别为 a 和 b 的传入传出参数：

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

swapTwoInts(_:_:) 函数简单地交换两个传入参数 a 和 b 的值。 首先将参数 a 的值存储于临时变量 temporaryA 中，然后将 b 的值赋值给 a ， 最后将临时变量 temporaryA 的值再赋值给 b。

你可以通过传递两个 Int 类型的参数来调用 swapTwoInts(_:_:) 函数来交换彼此的值。 需要注意的是， 在调用 swapTwoInts(_:_:) 方法时，变量 someInt 和 anotherInt 需要加上 & 符号：

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
```

上面这个例子中，即便变量 someInt 和 anotherInt 被定义在函数体外

部，但通过参数传递，`swapTwoInts(_:_:)` 函数还是修改了彼此的初始值。

> 注意
>
> 在同一个函数中，传入传出参数和返回值不一定要同时存在。上面这个例子中， `swapTwoInts` 函数没有返回值， 但变量 `someInt` 和 `anotherInt` 的初始值仍然被修改了。 传入传出参数为函数影响函数体外部的作用域提供了一种可选的方式。

函数类型

每个函数的具体 *函数类型* 由它的参数类型和返回类型共同决定。

举个例子：

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

上面分别定义了两个叫 `addTwoInts` 和 `multiplyTwoInts` 的运算函数。 分别接受两个 `Int` 类型的参数， 经过各自合适地运算再返回一个 `Int` 值。

这两个函数的类型都是 `(Int, Int) -> Int` 。你可以理解为：

「这个函数接收两个都是 `Int` 的值，返回一个 `Int` 类型的值。」

再例如， 下面这个函数没有参数也没有返回值：

```
func printHelloWorld() {
    print("hello, world")
}
```

那么它的函数类型就是 `() -> Void` ， 你可以理解为没有参数返回值是空

Void 。

使用函数类型

Swift 中，函数类型的使用和其他类型一样。 举个例子， 你可以定义 一个函数类型的常量或变量，并给其赋一个函数类型的值：

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

你可以理解为：

『 定义了一个函数类型的 mathFunction 变量， 「 它接收两个 Int 类型的参数，并返回一个 Int 值。」并把 addTwoInts 函数关联给这个变量。』

mathFunction 的类型和 addTwoInts(_:_:) 函数的类型相同，所以Swift 的类型检查器将允许这样的赋值。

现在，你就可以用 mathFunction 变量来调用 addTwoInts(_:_:) 函数了：

```
print("Result: \(mathFunction(2, 3))")
```

有着相同类型的不同方法可以赋值给同一个变量：

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
```

像其他类型一样，当然你给一个变量或常量赋一个函数类型的值时，Swift 将帮你自动推导出值的真实类型：

```
let anotherMathFunction = addTwoInts
```

函数类型作为参数

你可以把 (Int, Int) -> Int 类型的函数作为一个参数传递给另一个函数。 当这个函数被调用时，这使得具体实现逻辑被当做一个函数传递给

了这个函数的调用者。

下面这个例子，打印 math functions 函数相加后的结果：

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int)
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
```

这个例子中， 定义了一个 printMathResult(_:_:_:) 的函数，它接收三个参数。第一个参数是一个叫 mathFunction的函数，其类型是 (Int, Int) -> Int。 你可以为第一个参数传递一个类型是 (Int, Int) -> Int 的函数作为参数。 第二个和第三个参数 a 和 b 都是 Int 类型。这两个变量被当做第一个函数参数的输入值传入给了第一个参数。

当调用 printMathResult(_:_:_:) 时，分别传递 addTwoInts(_:_:) 函数和另外两个值 3 和 5。 3 和 5 被当做第一个函数的参数传递给它做了相加，最终打印结果 8。

函数 printMathResult(_:_:_:) 的作用是打印 addTwoInts(_:_:) 函数的返回值。 它不关心传入函数的具体实现 — 只关心传入函数的正确类型。 这使得 printMathResult(_:_:_:) 函数把一些具体的功能逻辑实现推给了它的调用者。

返回类型为函数类型

你可以把一个函数类型作为另一个函数的返回类型。在这个返回箭头后面 (->) 跟上你要返回的具体函数类型。

下面这个例子分别定义了两个 stepForward(_:) 和 stepBackward(_:) 简单的函数。 stepForward(_:) 函数返回一个在其输入值上 +1 后的值， stepBackward(_:) 方法返回一个在其输入值上 -1 后的值，两个方法的函数类型都是 (Int) -> Int：

```
func stepForward(_ input: Int) -> Int {
```

```
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

chooseStepFunction(backward:) 函数的返回类型是 (Int) -> Int 。 该函数根据一个 Bool 类型值来判断是返回 stepForward(_:) 还是 stepBackward(_:) 函数：

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

现在你可以通过调用 chooseStepFunction(backward:) 并为其输入一个 Bool 类型的值来获得一个递增或递减的函数：

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
```

在上面这个例子中，最终返回递增还是递减函数由 currentValue 变量的值来决定。 变量 currentValue 的初始值是 3 ， currentValue > 0 比较结果就为 true，所以调用 chooseStepFunction(backward:) 后返回 stepBackward(_:) 函数。 常量 moveNearerToZero 存储着该函数的返回函数。

现在 moveNearerToZero 表示这个递减函数，从输入值递减至 0：

```
print("Counting to zero:")

while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
```

嵌套函数

到目前为止，在本章中你遇到的这些函数例子都是 *全局函数* ， 它们被定义在全局作用域上。 当然，你也可以在函数体内定义一个函数，来做为该函数的 *嵌套函数* 。

虽然嵌套函数默认对函数体外部是透明的，但仍然可以被该函数调用。函数也可以通过返回其内部的嵌套函数来使这个被嵌套的函数在外部作用域可以被使用。

你可以重写上面的 `chooseStepFunction(backward:)` 函数，来返回和使用其内部的嵌套函数：

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)

while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
```