

The Swift Programming Language in Chinese

<https://github.com/letsswift>

<http://letsswift.com/>

June 15, 2014

目录

1 Swift 中文教程（五）控制流	1
1.1 for 循环	1
1.1.1 for-in 循环	1
1.1.2 For-Condition-Increment 条件循环	3
1.2 while 循环	5
1.2.1 while 循环	5
1.2.2 Do-while 循环	8
1.3 条件语句	8
1.3.1 if 语句	9
1.3.2 switch 语句	10
1.3.3 不会一直执行	11
1.3.4 范围匹配	12
1.3.5 元组	12
1.3.6 数值绑定	13
1.3.7 Where 关键词	16
1.4 控制跳转语句	16
1.4.1 continue	16
1.4.2 break	18
1.4.3 fallthrough	19
1.4.4 标签语句	19

1 Swift 中文教程 (五) 控制流

Swift 提供了所有 C 语言中相似的控制流结构。包括 for 和 while 循环；if 和 switch 条件语句；break 和 continue 跳转语句等。

Swift 还加入了 for-in 循环语句，让编程人员可以在遍历数组，字典，范围，字符串或者其它序列时更加便捷。

相对于 C 语言，Swift 中 switch 语句的 case 语句后，不会自动跳转到下一个语句，这样就避免了 C 语言中因为忘记 break 而造成的错误。另外 case 语句可以匹配多种类型，包括数据范围，元组，或者特定的类型等。switch 语句中已匹配的数值也可以被用在后续的 case 语句体中，where 关键词还能被加入任意的 case 语句中，来增加匹配的方式。

1.1 for 循环

for 循环可以根据设置，重复执行一个代码块多次。Swift 中提供了两种 for 循环方式：

for-in 循环，对于数据范围，序列，集合等中的每一个元素，都执行一次

for-condition-increment，一直执行，知道一个特定的条件满足，每一次循环执行，都会增加一次计数

1.1.1 for-in 循环

下面的例子打印出了 5 的倍数序列的前 5 项

```
for index in 1...5 {  
    println("\(index) times 5 is \(index * 5)")  
}  
// 1 times 5 is 5  
// 2 times 5 is 10  
// 3 times 5 is 15  
// 4 times 5 is 20  
// 5 times 5 is 25
```

迭代的项目是一个数字序列，从 1 到 5 的闭区间，通过使用 (...) 来表示序列。index 被赋值为 1，然后执行循环体中的代码。在这种情况下，循环只有一条语句，也就是打印 5 的 index 倍数。在这条语句执行完毕后，index 的值被更

新为序列中的下一个数值 2, `println` 函数再次被调用, 一次循环直到这个序列的结尾。

在上面的例子中, `index` 在每一次循环开始前都已经被赋值, 因此不需要在每次使用前对它进行定义。每次它都隐式地被定义, 就像是使用了 `let` 关键词一样。注意 `index` 是一个常量。

注意: `index` 只在循环中存在, 在循环完成之后如果需要使用, 需要重新定义才可以。

如果你不需要序列中的每一个值, 可以使用 `_` 来忽略它, 仅仅只是使用循环体本身:

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
println("\(base) to the power of \(power) is \(answer)")
// prints "3 to the power of 10 is 59049"
```

这个例子计算了一个数的特定次方 (在这个例子中是 3 的 10 次方)。连续的乘法从 1 (实际上是 3 的 0 次方) 开始, 依次累乘以 3, 由于使用的是半闭区间, 从 0 开始到 9 的左闭右开区间, 所以是执行 10 次。在循环的时候不需要知道实际执行到第一次了, 而是要保证执行了正确的次数, 因此这里不需要 `index` 的值。

同理我们可以使用 `for-in` 来循环遍历一个数组的元素

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

在遍历字典的时候，可以使用 key-value 对来进行遍历。每一个字典中的元素都是一个 (key, value) 元组，当遍历的时候，可以指定字段的 key 和 value 为一个特定的名称，这样在遍历的时候就可以更好地理解和使用它们，比如下面例子中的 animalName 和 legCount：

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

字典中的元素在遍历的时候一般不需要按照插入的顺序，因此不能保证遍历字典的时候，元素是有序的。更多跟数组和字典相关的内容可以参考：Collection Types

另外在数组和字典中也可以使用类似的遍历方式，如可以使用 for-in 循环来遍历字符串中的每一个字符：

```
for character in "Hello" {
    println(character)
}
// H
// e
// l
// l
// o
```

1.1.2 For-Condition-Increment 条件循环

Swift 同样支持 C 语言样式的 for 循环，它也包括了一个条件语句和一个增量语句：

```
for var index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
```

```
// index is 1
// index is 2
```

下面是这种 for 循环的一般结构:

```
for initialization; condition; increment {
    statements
}
```

分号在这里用来分隔 for 循环的三个结构, 和 C 语言一样, 但是不需要用括号来包裹它们。

这种 for 循环的执行方式是:

1. 当进入循环的时候, 初始化语句首先被执行, 设定好循环需要的变量或常量
2. 测试条件语句, 看是否满足继续循环的条件, 只有在条件语句是 true 的时候才会继续执行, 如果是 false 则会停止循环。
3. 在所有的循环体语句执行完毕后, 增量语句执行, 可能是对计数器的增加或者是减少, 或者是其它的一些语句。然后返回步骤 2 继续执行。

这种循环方式还可以被描述为下面的形式:

```
initialization
while condition {
    statements
    increment
}
```

在初始化语句中被定义 (比如 `var index = 0`) 的常量和变量, 只在 for 循环语句范围内有效。如果想要在循环执行之后继续使用, 需要在循环开始之前就定义好:

```
var index: Int
for index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
```

```
println("The loop statements were executed \(index) times")
// prints "The loop statements were executed 3 times"
```

需要注意的是，在循环执行完毕之后，`index` 的值是 3，而不是 2。因为是在 `index` 增 1 之后，条件语句 `index < 3` 返回 `false`，循环才终止，而这时，`index` 已经为 3 了。

1.2 while 循环

`while` 循环执行一系列代码块，直到某个条件为 `false` 为止。这种循环最长用于循环的次数不确定的情况。Swift 提供了两种 `while` 循环方式：

`while` 循环，在每次循环开始前测试循环条件是否成立

`do-while` 循环，在每次循环之后测试循环条件是否成立

1.2.1 while 循环

`while` 循环由一个条件语句开始，如果条件语句为 `true`，一直执行，直到条件语句变为 `false`。下面是一个 `while` 循环的一般形式：

```
while condition {
    statements
}
```

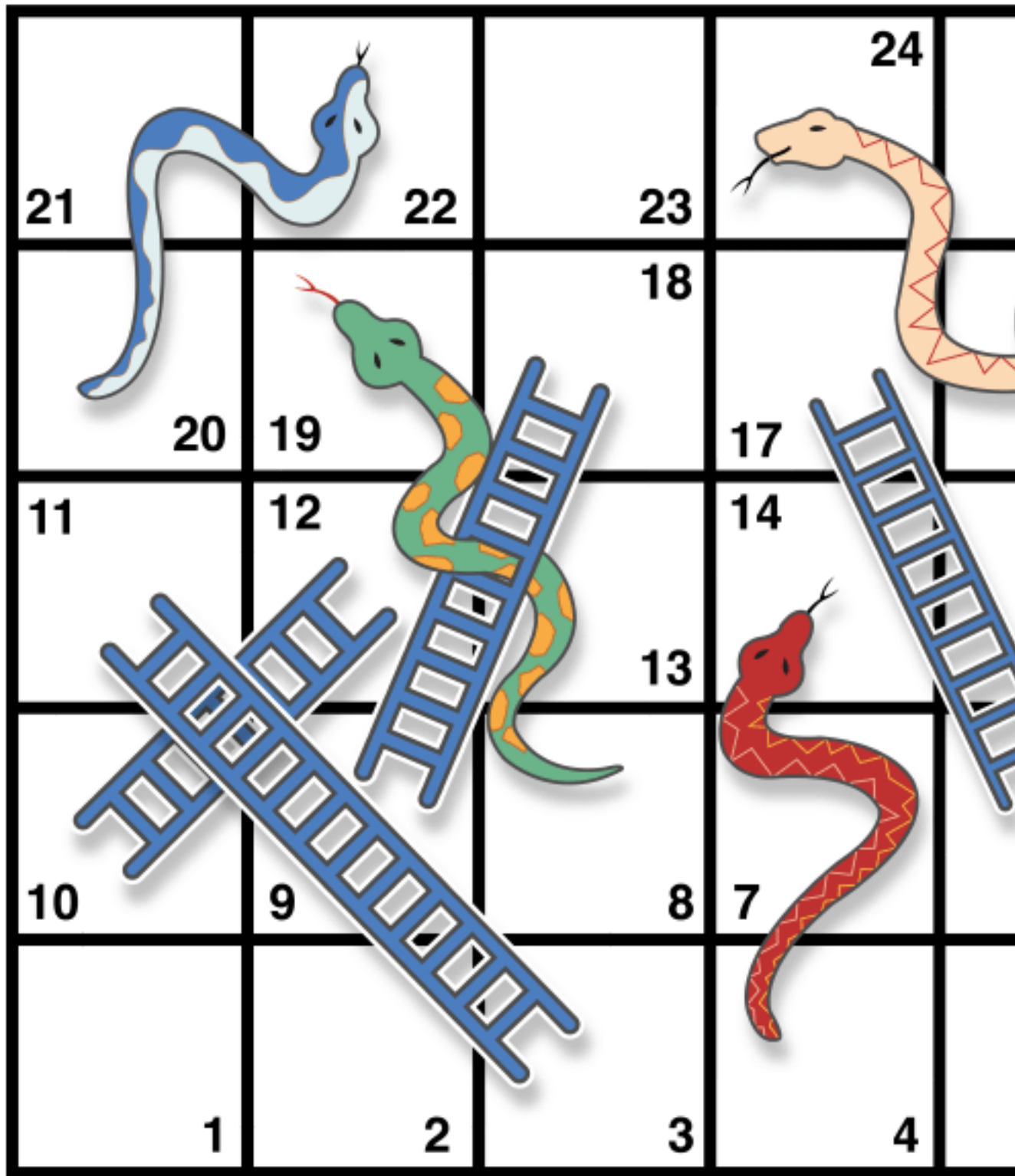
下面的例子是一个简单的游戏，Snakes and Ladders，蛇和梯子

游戏的规则是这样的：

- 游戏面板上有 25 个格子，游戏的目标是到达第 25 个格子；
- 每个回合通过一个 6 面的骰子来决定行走的步数，行走的路线按右图所示；
- 如果落在梯子的底部，那么就爬上那个梯子到达另外一个格子；
- 如果落到蛇的头部，就会滑到蛇尾部所在的格子。

游戏面板由一个 `Int` 数组组成，大小由一个常量设置 `finalSquare`，同时用来检测是否到达了胜利的格子。游戏面板由 26 个 `Int` 数字 0 初始化（不是 25 个，因为从 0 到 25 有 26 个数字）

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



其中一些格子被设置为一些特定的值用来表示蛇或者梯子。有梯子的地方是整数，而有蛇的地方是负数：

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

第三个格子是一个梯子的底部，表示玩家可以通过梯子到达第 11 格，因此设置 board[3] 为 +08，表示前进 8 步。同理蛇的位置设置为负数，表示后退 i 步。

玩家从为 0 的格子开始游戏。

```
var square = 0  
var diceRoll = 0  
while square < finalSquare {  
    // roll the dice  
    if ++diceRoll == 7 { diceRoll = 1 }  
    // move by the rolled amount  
    square += diceRoll  
    if square < board.count {  
        // if we're still on the board, move up or down for a snake or a ladder  
        square += board[square]  
    }  
}  
println("Game over!")
```

这个例子用到了一个非常简单的掷骰子的方式，就是每次加 1，而不是使用一个随机数。diceRoll 用来表示每次行走的步数，需要注意的是，每次执行前，++diceRoll 都会先执行加 1，然后再与 7 比较，如果等于 7 的话，就设置为 1，因此可以看出 diceRoll 的变化是 1,2,3,4,5,6,1.....

在掷骰子之后，玩家移动 diceRoll 指示的步数，这时可能已经超过了 finalSquare，因此需要进行 if 判断，如果为 true 的话，执行该格子上的事件：如果是普通格子就不动，如果是梯子或者蛇就移动相应的步数，这里只需要直接使用 square += board[square] 就可以了。

在 while 循环执行完毕之后，重新检查条件 square < finalSquare 是否成立，继续游戏直到游戏结束。

1.2.2 Do-while 循环

另一种 while 循环是 do-while 循环。在这种循环中，循环体中的语句会先被执行一次，然后才开始检测循环条件是否满足，下面是 do-while 循环的一般形式：

```
do {  
    statements  
} while condition
```

上面的蛇与梯子的游戏使用 do-while 循环来写可以这样完成。初始化语句和 while 循环的类似：

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

在这种循环中，第一个动作就是检测是否落在梯子或者蛇上，因为没有梯子或者蛇可以让玩家直接到达第 25 格，所以游戏不会直接结束，接下来的过程就和上面的 while 循环类似了，循环的条件语句还是检测是否已经达到最终格子。

```
do {  
    // move up or down for a snake or ladder  
    square += board[square]  
    // roll the dice  
    if ++diceRoll == 7 { diceRoll = 1 }  
    // move by the rolled amount  
    square += diceRoll  
} while square < finalSquare  
println("Game over!")
```

1.3 条件语句

通常情况下我们都需要根据不同条件来执行不同语句。比如当错误发生的时候，执行一些错误信息的语句，告诉编程人员这个值是太大了还是太小了等等。这里就需要用到条件语句。

Swift 提供了两种条件分支语句的方式，if 语句和 switch 语句。一般 if 语句比较常用，但是只能检测少量的条件情况。switch 语句用于大量的条件可能发生时的条件语句。

1.3.1 if 语句

在最基本的 if 语句中，条件语句只有一个，如果条件为 true 时，执行 if 语句块中的语句：

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

上面这个例子检测温度是不是比 32 华氏度（32 华氏度是水的冰点，和摄氏度不一样）低，如果低的话就会输出一行语句。如果不低，则不会输出。if 语句块是用大括号包含的部分。

当条件语句有多种可能时，就会用到 else 语句，当 if 为 false 时，else 语句开始执行：

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

在这种情况下，两个分支的其中一个一定会被执行。

同样也可以有多个分支，使用多次 if 和 else

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

```
} else {  
    println("It's not that cold. Wear a t-shirt.")  
}  
// prints "It's really warm. Don't forget to wear sunscreen."
```

上面这个例子中有多个 if 出现，用来判断温度是太低还是太高，最后一个 else 表示的是温度不高不低的时候。

当然 else 也可以被省掉

```
temperatureInFahrenheit = 72  
if temperatureInFahrenheit <= 32 {  
    println("It's very cold. Consider wearing a scarf.")  
} else if temperatureInFahrenheit >= 86 {  
    println("It's really warm. Don't forget to wear sunscreen.")  
}
```

在这个例子中，温度不高不低的时候不会输入任何信息。

1.3.2 switch 语句

switch 语句考察一个值的多种可能性，将它与多个 case 相比较，从而决定执行哪一个分支的代码。switch 语句和 if 语句不同的是，它还可以提供多种情况同时匹配时，执行多个语句块。

switch 语句的一般结构是：

```
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 语句包含有多个 case 语句块，除了直接比较值以外，Swift 还提供了多种更加复杂的模式匹配的方式来选择语句执行的分支，这在后续的小节会继续介绍。

在 switch 中，每一个 case 分支都会被匹配和检测到，所有 case 没有提到的情况都必须使用 default 关键词。注意 default 关键词必须在所有 case 的最后。

下面的例子用 switch 语句来判断一个字符的类型：

```
let someCharacter: Character = "e"
switch someCharacter {
    case "a", "e", "i", "o", "u":
        println("\(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":
        println("\(someCharacter) is a consonant")
    default:
        println("\(someCharacter) is not a vowel or a consonant")
}
// prints "e is a vowel"
```

在这个例子中，首先看这个字符是不是元音字母，再检测是不是辅音字母。其它的情况都用 default 来匹配即可。

1.3.3 不会一直执行

跟 C 和 Objective-C 不同，Swift 中的 switch 语句不会因为在 case 语句的结尾没有 break 就跳转到下一个 case 语句执行。switch 语句只会执行匹配上的 case 里的语句，然后就会直接停止。这样可以让 switch 语句更加安全，因为很多时候编程人员都会忘记写 break。

每一个 case 中都需要有可以执行的语句，下面的例子就是不正确的：

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
    case "a":
    case "A":
        println("The letter A")
    default:
        println("Not the letter A")
}
// this will report a compile-time error
```

跟 C 不同, switch 语句不会同时匹配 a 和 A, 它会直接报错。一个 case 中可以有多个条件, 用逗号, 分隔即可:

```
switch some value to consider {
    case value 1,
         value 2:
        statements
}
```

1.3.4 范围匹配

switch 语句的 case 中可以匹配一个数值范围, 比如:

```
let count = 3_000_000_000_000
let countedThings = "stars in the Milky Way"
var naturalCount: String
switch count {
    case 0:
        naturalCount = "no"
    case 1...3:
        naturalCount = "a few"
    case 4...9:
        naturalCount = "several"
    case 10...99:
        naturalCount = "tens of"
    case 100...999:
        naturalCount = "hundreds of"
    case 1000...999_999:
        naturalCount = "thousands of"
    default:
        naturalCount = "millions and millions of"
}
println("There are \(naturalCount) \(countedThings).")
// prints "There are millions and millions of stars in the Milky Way."
```

1.3.5 元组

case 中还可以直接测试元组是否符合相应的条件, `_` 可以匹配任意值。

下面的例子是判断 (x,y) 是否在矩形中, 元组类型是 (Int,Int)

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

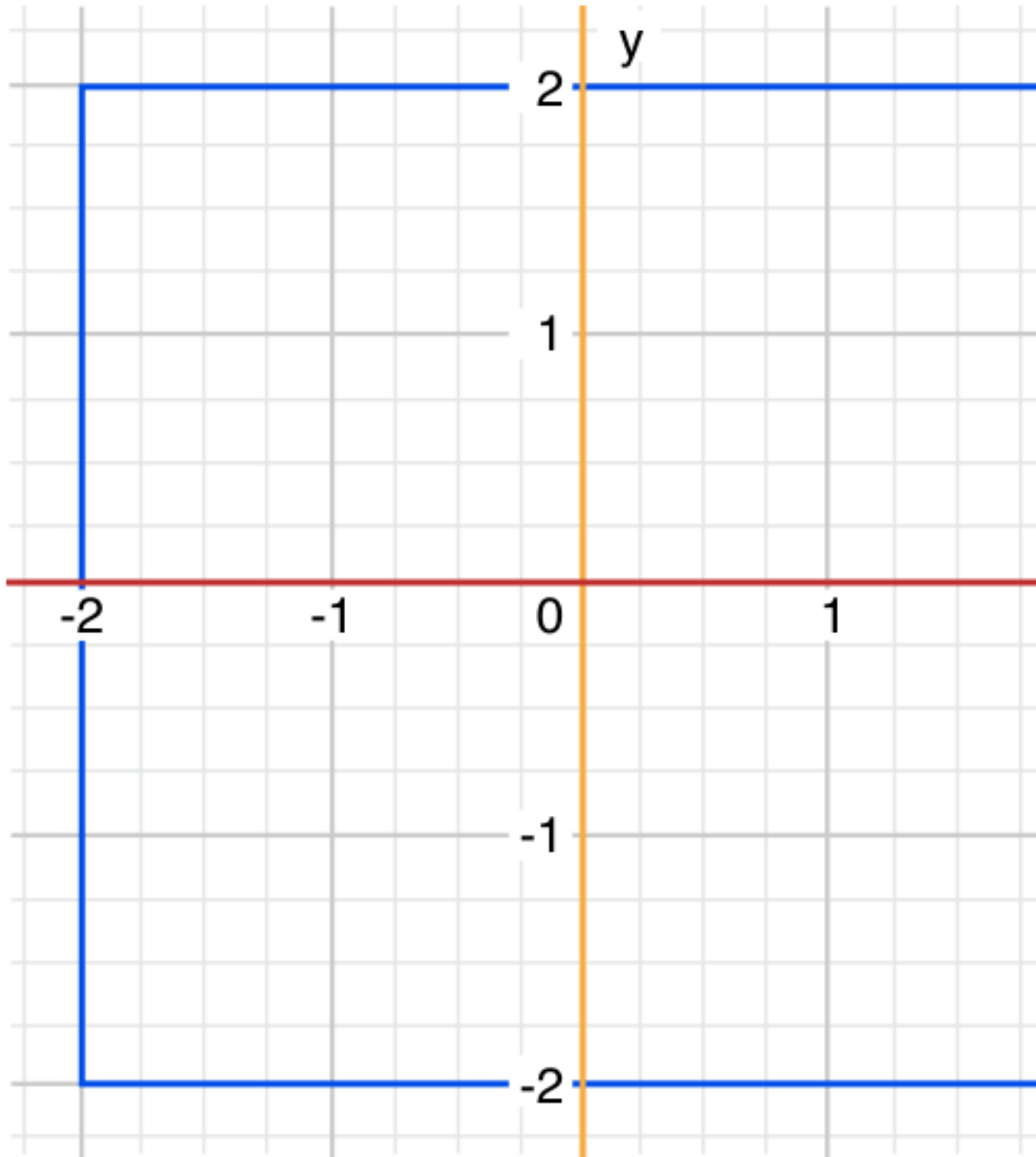
和 C 语言不同, Swift 可以判断元组是否符合条件。

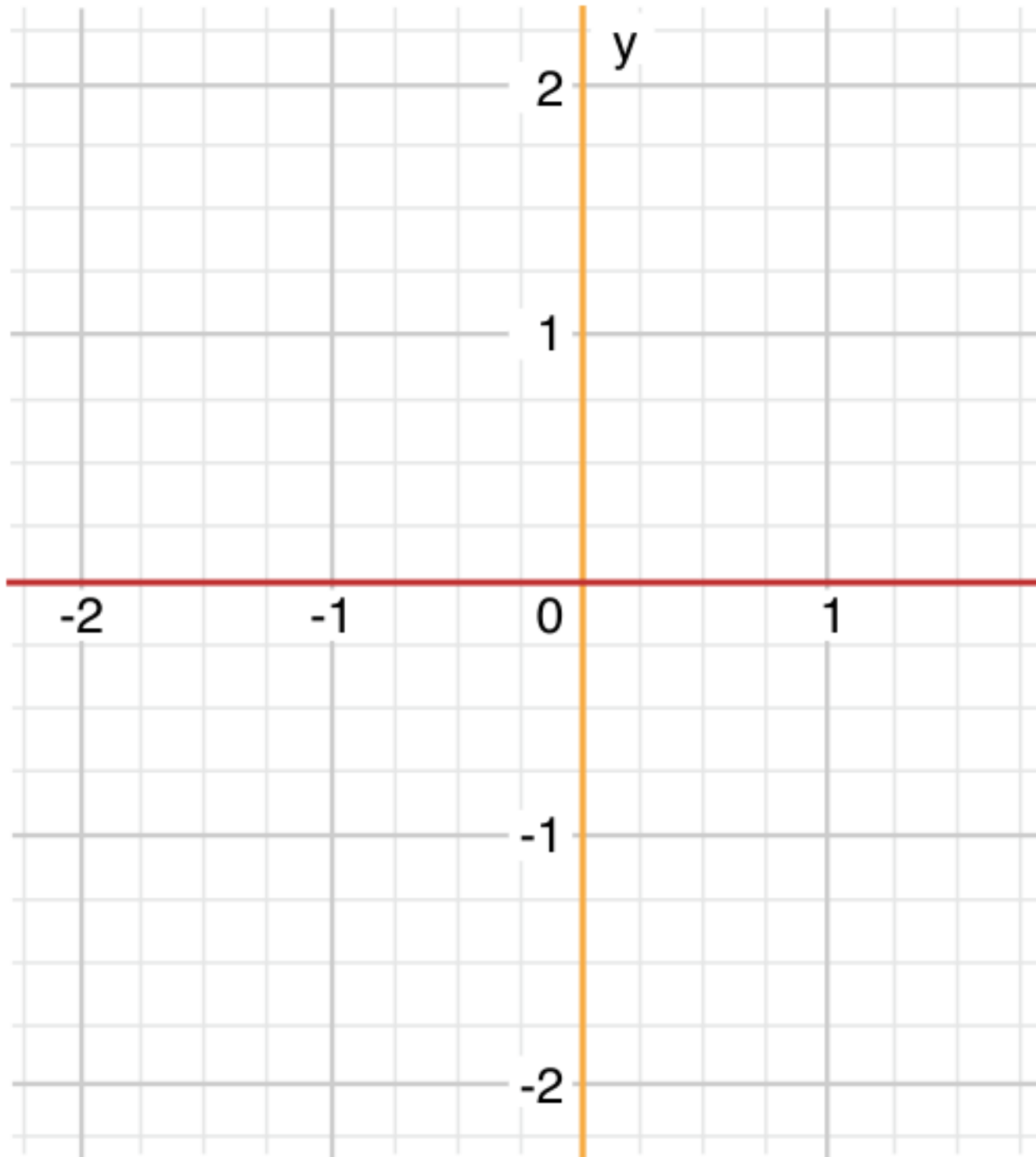
1.3.6 数值绑定

在 case 匹配的同时, 可以将 switch 语句中的值绑定给一个特定的常量或者变量, 以便在 case 的语句中使用。比如:

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

switch 语句判断一个点是在 x 轴上还是 y 轴上, 或者在其他地方。这里用到了匹配和数值绑定。第一种情况, 如果点是 (x,0) 模式的, 将值绑定到 x 上, 这样在 case 语句中可以输出该值。同理如果在 y 轴上, 就输出 y 的值。





1.3.7 Where 关键词

switch 语句可以使用 where 关键词来增加判断的条件，在下面的例子中：

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
    case let (x, y) where x == y:
        println("\(x), \(y)) is on the line x == y")
    case let (x, y) where x == -y:
        println("\(x), \(y)) is on the line x == -y")
    case let (x, y):
        println("\(x), \(y)) is just some arbitrary point")
}
// prints "(1, -1) is on the line x == -y"
```

每个 case 都因为有 where 而不同，第一个 case 就是判断 x 是否与 y 相等，表示点在斜线 $y=x$ 上。

1.4 控制跳转语句

在 Swift 中控制跳转语句有 4 种，让编程人员更好地控制代码的流转，包括：

- continue
- break
- fallthrough
- return

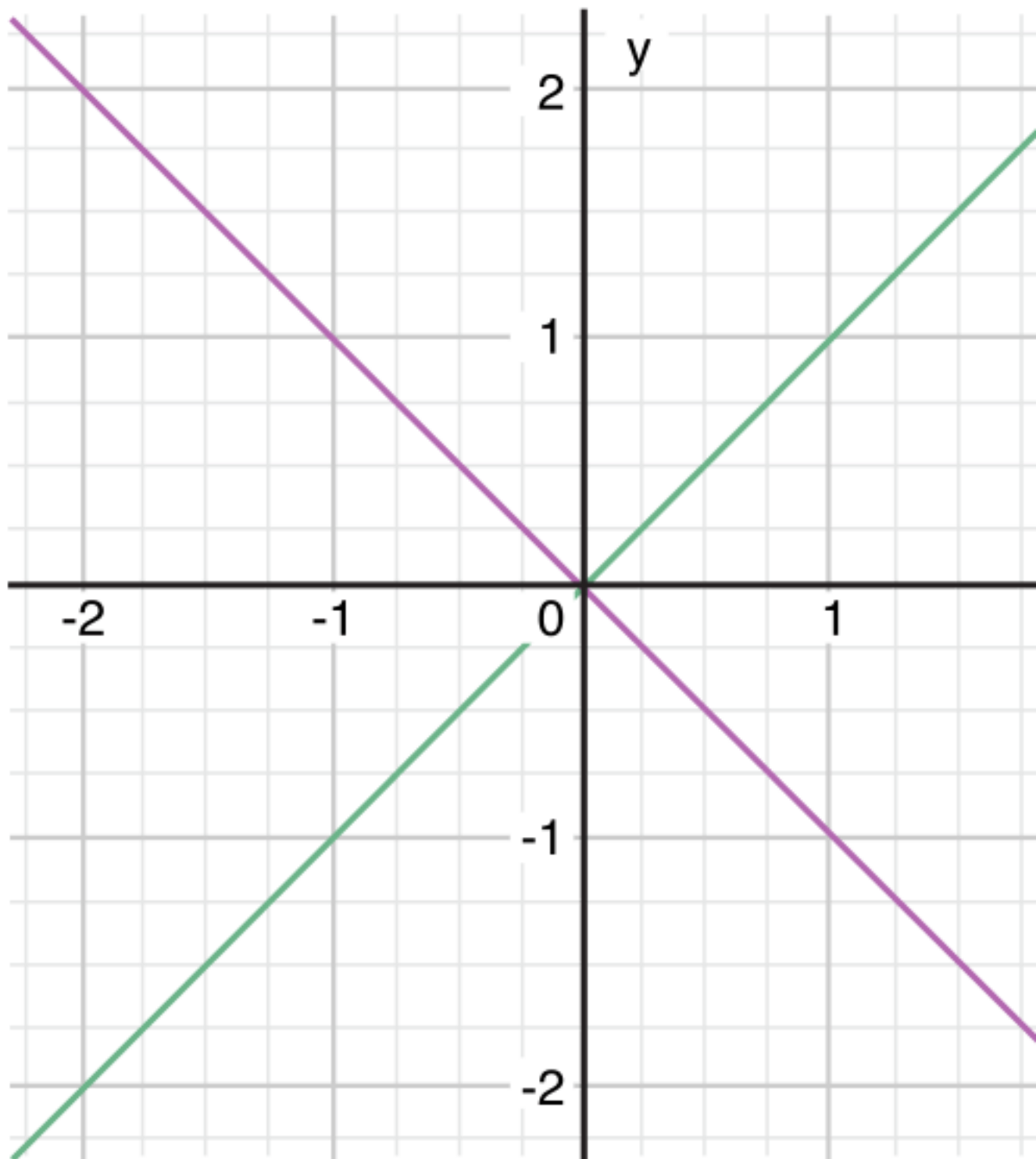
其中 continue, break 和 fallthrough 在下面详细介绍，return 语句将在函数一章介绍。

1.4.1 continue

continue 语句告诉一个循环停止现在在执行的语句，开始下一次循环。

注意：在 for-condition-increment 循环中，increment 增量语句依然执行，只是略过了一次循环体。

下面的例子实现的是去除一个字符串中的空格和元音字母，从而组成一个字谜：



```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput {
    switch character {
        case "a", "e", "i", "o", "u", " ":
            continue
        default:
            puzzleOutput += character
    }
}
println(puzzleOutput)
// prints "grtmndsthnlk"
```

遍历字符串的每一个字符，当遇到元音字母或者空格时就忽略，进行下一次循环，从而得到了最终的字谜。

1.4.2 break

break 语句将终止整个循环的执行，可以用在循环语句中，也可以用在 switch 语句中。

```
let numberSymbol: Character = "三" // Simplified Chinese for the number 3
var possibleIntegerValue: Int?
switch numberSymbol {
    case "1", "一", "壹", " ":
        possibleIntegerValue = 1
    case "2", "二", "贰", " ":
        possibleIntegerValue = 2
    case "3", "三", "叁", " ":
        possibleIntegerValue = 3
    case "4", "四", "肆", " ":
        possibleIntegerValue = 4
    default:
        break
}

if let integerValue = possibleIntegerValue {
    println("The integer value of \(numberSymbol) is \(integerValue).")
}
```

```
} else {  
    println("An integer value could not be found for \(numberSymbol).")  
}  
// prints "The integer value of 3 is 3."
```

上面的例子首先检查 `numberSymbol` 是不是一个数字，阿拉伯数字，汉字，拉丁文或者泰文都可以。如果匹配完成，则将 `possibleIntegerValue` 赋值。最后通过 `if` 语句检测是否已被赋值，并绑定到 `integerValue` 常量上，最后输出。`default` 语句用来迎接未能被上述 `case` 匹配的情况，但是不需要做任何事情，因此直接使用 `break` 终止即可。

1.4.3 fallthrough

由于 Swift 中的 `switch` 语句不会自动的因为没有 `break` 而跳转到下一个 `case`，因此如果需要像 C 语言中那样，依次执行每个 `case` 的时候，就需要用到 `fallthrough` 关键词。

像下面这个例子一样，`default` 分支最终都会被执行：

```
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."  
}  
println(description)  
// prints "The number 5 is a prime number, and also an integer."
```

1.4.4 标签语句

`switch` 和循环可以互相嵌套，循环之间也可以互相嵌套，因此在使用 `break` 或者 `continue` 的时候，需要知道到底是对哪个语句起作用。这就需要用到标签语句。标签语句的一般形式如下：

```
label name: while condition {
```

```
    statements
}
```

下面的例子演示了如何使用标签语句以及嵌套的循环和 switch。

依然采用之前的那个梯子与蛇的游戏，第一步依然是设置初始值：

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

然后，使用一个 while 循环与 switch 的嵌套来完成游戏

```
gameLoop: while square != finalSquare {
    if ++diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
        case finalSquare:
            // diceRoll will move us to the final square, so the game is over
            break gameLoop
        case let newSquare where newSquare > finalSquare:
            // diceRoll will move us beyond the final square, so roll again
            continue gameLoop
        default:
            // this is a valid move, so find out its effect
            square += diceRoll
            square += board[square]
    }
}
println("Game over!")
```

在这个代码中，将游戏的循环命名为 gameLoop，然后在每一步移动格子时，判断当前是否到达了游戏终点，在 break 的时候，需要将整个游戏循环终止掉，而不是终止 switch，因此用到了 break gameLoop。同样的，在第二个分支中，continue gameLoop 也指明了需要 continue 的是整个游戏，而不是 switch 语句本身。

感谢翻译小组成员：李起攀 ([微博](#))、若晨 ([微博](#))、YAO、粽子、山有木兮木有枝、渺 -Bessie、墨离、Tiger 大顾 ([微博](#))，校对：CXH_ME([微博](#)),Joshua 孟思拓 ([微博](#))

本文由翻译小组成员原创发布，个人转载请注明出处和原始链接，商业转载请联系我们 感谢您对我们工作的支持