控制流

Swift提供了各种控制流程语句。这些包括while循环多次执行任务; if，guard以及switch基于特定条件执行不同分支代码的语句; 以及诸如此类的语句break以及continue将执行流程转移到代码中的另一个点。

迅速还提供了for- in循环，可以很容易地遍历数组，字典，范围，字符串和其它序列。

Swift的switch声明比许多C语言中的声明强大得多。个案可以匹配许多不同的模式，包括区间匹配，元组和特定类型的强制转换。switch案例中的匹配值可以绑定到临时常量或变量以用于案例的正文中，并且复杂的匹配条件可以用where每个案例的子句表示。

For-In循环

您可以使用for- in循环遍历序列，例如数组中的项目，数字范围或字符串中的字符。

此示例使用for- in循环来迭代在阵列中的项目：

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

您也可以迭代字典以访问其键值对。(key, value)当字典迭代时，字典中的每一项都作为元组返回，并且可以将(key, value)元组的成员分解为明确命名的常量，以便在for- in循环体内使用。在下面的代码示例中，字典的键被分解为一个常量animalName，字典的值被分解为一个常量legCount。

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

a的内容Dictionary本质上是无序的，迭代它们并不能保证它们被检索的顺序。特别是，您将项目插入a Dictionary的顺序不会定义它们迭代的顺序。有关数组和字典的更多信息，请参阅集合类型。

您也可以使用for- in数字范围的循环。此示例在五次表中打印前几个条目：

```
for index in 1...5 {
    print("\(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
```

正在迭代的序列是一系列从0 1到0 的数字5，如使用闭范围运算符（...）所示。该值index设置为范围（1）中的第一个数字，并执行循环内的语句。在这种情况下，循环仅包含一条语句，该语句从当前值的五次表中输入条目index。语句执行后，值index更新为包含范围（2）中的第二个值，并print(_:separator:terminator:)再次调用该函数。这个过程一直持续到范围的结束。

在上面的例子中，index是一个常量，其值在循环的每次迭代开始时自动设置。因此，index在使用之前不必声明。它仅仅通过包含在循环声明中而被隐式声明，而不需要let声明关键字。

如果您不需要序列中的每个值，则可以通过使用下划线代替变量名称来忽略这些值。

```
let base = 3
let power = 10
var answer = 1
for _ i
    answer *= base
}
print("\(base) to the power of \(power) is \(answer)")
// Prints "3 to the power of 10 is 59049"
```

上面的例子计算了一个数字与另一个数字的值（在这种情况下，3是幂的10）。它使用以开始和结束的封闭范围乘以1（即，3的幂0）的起始值3十倍。对于这种计算，每次循环中的单个计数器值都是不必要的 - 代码只需执行正确的循环次数。用于代替循环变量的下划线字符（）会导致单个值被忽略，并且在循环的每次迭代过程中都不提供对当前值的访问。 110_

在某些情况下，您可能不想使用封闭范围，其中包括两个端点。考虑在表盘上每刻刻绘制刻度标记。你想60从0分钟开始绘制刻度线。使用半开范围运算符（..<）来包含下界但不包含上界。有关范围的更多信息，请参阅范围操作员。

```
let minutes = 60
for tickMark in 0..<minutes {
    // render the tick mark each minute (60 times)
}
```

有些用户可能需要在他们的用户界面中减少刻度线 他们可能会更喜欢每5分钟一个标记。使用该stride(from:to:by:)功能跳过不需要的标记。

```
let minuteInterval = 5
for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {
    // render the tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55)
}
```

封闭的范围也可以使用，stride(from:through:by:)取而代之：

```
let hours = 12
let hourInterval = 3
for tickMark in stride(from: 3, through: hours, by: hourInterval) {
    // render the tick mark every 3 hours (3, 6, 9, 12)
}
```

虽然循环

甲while循环执行一组语句，直到条件变为false。当迭代次数在第一次迭代开始之前未知时，最好使用这些类型的循环。Swift提供了两种while循环：

- while 评估每次通过循环开始时的状态。
- repeat- while评估每次通过循环结束时的状态。

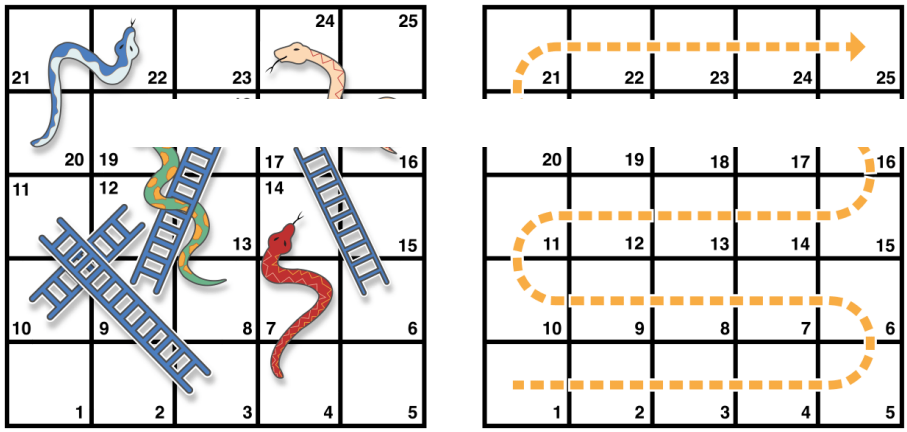
而

一个while循环开始通过评估一个条件。如果条件true成立，则重复一组陈述直到条件变为止false。

以下是while循环的一般形式：

```
而 条件 {
    声明
}
```

这个例子演示了一个简单的*Snakes和Ladders*（也称为*Chutes and Ladders*）游戏：



游戏规则如下：

- 该委员会有25个广场，目的是降落在25平方或以上。
- 玩家的起始方格是“正方形零点”，就在董事会的左下角。
- 每回合，你掷出一个六面骰子，并按照上面虚线箭头指示的水平路径移动该数量的正方形。
- 如果你的回合在梯子的底部结束，那么你在梯子上移动。
- 如果你的回合结束于蛇的头部，那么你会沿着那条蛇移动。

游戏板由一系列Int值表示。其大小基于一个常量finalSquare，用于初始化数组，并在稍后的示例中检查胜利条件。因为玩家从棋盘开始，在“方块零点”上，棋盘以26个零Int值进行初始化，而不是25个。

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

然后一些正方形设置为蛇和梯子的更具体的值。带梯子底座的方块有一个正数，可以将您移动到棋盘上，而带有蛇头的方块有负数可以让您退回棋盘。

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

方3包含移动你到平方11.要表示这样的阶梯的底部，board[03]是等于+08，这相当于一个整数值8（之间的差3和11）。为了对齐值和语句，一元加运算符（+i）与一元减运算符（-i）一起明确使用，低于10用零填充的数字。（风格技术不是绝对必要的，但它们导致整洁的代码。）

```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // roll the dice
    diceRoll += 1
    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]
    }
}
print("Game over!")
```

上面的例子使用了非常简单的方法来掷骰子。不是生成一个随机数，而是从一个diceRoll值开始0。每次while循环时，diceRoll都会加1，然后检查它是否变得过大。只要这个返回值相等7，骰子滚动就会变得太大，并被重置为一个值1。结果是一个序列diceRoll，始终是价值观1，2，3，4，5，6，1，2等。

掷骰子后，玩家前进diceRoll广场。掷骰子有可能将玩家移动到方块25以外，在这种情况下游戏结束。为了处理这种情况，代码检查的结果square小于board数组的count属性。如果square有效，则存储的值将board[square]被添加到当前square值以将播放器向上或向下移动任何梯子或蛇。

> 注意

> 如果未执行此检查，则board[square]可能尝试访问board数组边界外的值，这会触发运行时错误。

然后当前while循
25，则回路的状

一个while循环是在这种情况下，适当的，因为游戏的长度是不是在开始明确while循环。相反，循环被执行直到满足特定的条件。

重复，虽然

while循环 的另一种变体（称为repeat- while循环）*在*考虑循环条件*之前*首先执行一次循环块。然后它继续重复循环直到条件成立false。

> 注意
>
> 本repeat- while环斯威夫特类似于一个do- while循环其他语言。

这里有一个的一般形式repeat- while循环：

```
重复 {
    声明
} 而 条件
```

这里的*蛇和梯子*例子再次，写成repeat- while循环而不是一个while循环。的值finalSquare，board，square，和diceRoll以完全相同的方式初始化为一个while循环。

```
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
```

在这个版本的游戏中，循环中的*第一个*动作是检查梯子或蛇。棋盘上没有任何梯子将球员直接拉到25平方，所以不可能通过上梯来赢得比赛。因此，检查一条蛇或梯子是循环中的第一个动作是安全的。

在比赛开始时，玩家处于“平方零点”。board[0]总是平等0并且没有效果。

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

在代码检查蛇和梯子之后，掷骰子并且玩家被diceRoll方块向前移动。当前循环执行结束。

循环的条件（while square < finalSquare）与以前相同，但是这次它不会被计算，直到循环的第一次运行*结束*。repeat- while循环的结构比while上例中的循环更适合这个游戏。在repeat- while上面的循环，square += board[square]始终执行*后立即*循环的while条件证实，square仍然是在黑板上。这种行为消除了在while前面描述的游戏的循环版本中看到的数组边界检查的需要。

条件声明

根据特定条件执行不同的代码通常很有用。发生错误时可能需要运行一段代码，或者当值变得过高或过低时显示消息。要做到这一点，你需要*有条件的*部分代码。

Swift提供了两种方法将条件分支添加到代码中：if语句和switch语句。通常情况下，您if只用几个可能的结果就可以使用该陈述来评估简单的条件。该switch语句更适合于具有多种可能排列的更复杂的条件，并且在模式匹配可帮助选择适当的代码分支来执行的情况下非常有用。

如果

最简单的形式是if声明只有一个if条件。它只有在条件满足时才执行一组语句true。

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

上面的例子检查温度是否小于或等于华氏32度（水的冰点）。如果是，则打印一条消息。否则，不会打印任何消息，并且在if语句的大括号之后继续执行代码。

该if语句可以提供一组替代语句，称为*else子句*，用于if条件满足的情况false。这些陈述由else关键字指示。

```
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.")
}
// Prints "It's not that cold. Wear a t-shirt."
```

这两个分支之一总是被执行。由于温度已经升高到40华氏度，所以不再建议穿围巾，所以else分支被触发。

您可以将多个if语句链接在一起考虑附加的子句。

```
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.")
}
// Prints "It's really warm. Don't forget to wear sunscreen."
```

在这里，增加了一个额外的if声明以应对特别温暖的气温。最后的else条款仍然存在，并且它对任何既不太热也不太冷的温度打印响应。

else然而， 最终条款是可选的，如果条件集合不需要完整，则可以排除该条款。

```
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.")
}
```

因为温度不太冷太热以触发if或else if条件下，印刷的任何消息。

开关

一个switch声明会考虑一个值并将其与几种可能的匹配模式进行比较。然后根据成功匹配的第一个模式执行适当的代码块。一个switch语句提供的另一种if用于应对多种潜在状态的语句。

以最简单的形式，一个switch语句将一个值与一个或多个相同类型的值进行比较。

```
切换 一些价值来考虑 {
案例 值1：
    回应价值1
案例 值2，
    值3：
    回应价值2或3
默认值：
    否则，做别的事情
}
```

每个switch陈述包含多个可能的*案例*，每个*案例*都以case关键字开头。除了与特定值进行比较外，Swift还为每种情况提供了几种方式来指定更复杂的匹配模式。这些选项将在本章稍后介绍。

与if声明的主体一样，每个声明case都是代码执行的独立分支。该switch声明确定应该选择哪个分支。此过程称为*打开*正在考虑的值。

每个switch陈述都必须是*详尽的*。也就是说，所考虑类型的每个可能值都必须与其中一种switch情况相匹配。如果不适合为每个可能的值提供一个案例，则可以定义一个默认案例来覆盖任何未明确解决的值。此默认情况由default关键字指示，并且必须始终显示最后一个。

本示例使用switch语句来考虑一个名为的小写字符someCharacter：

```
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")
}
// Prints "The last letter of the alphabet"
```

该switch声明的第一个案例与英文字母的第一个字母匹配a，并且第二个案例与最后一个字母匹配z。因为switch每个可能的字符都必须有一个字符，而不仅仅是每个字母字符，所以该switch语句使用一个default大小写来匹配除aand 之外的所有字符z。该条款确保该switch声明是详尽无遗的。

没有隐含的穿透

与switchC和Objective-C中的switch语句相比，Swift中的语句不会落在每个案例的底部，默认情况下是下一个。相反，switch只要第一个匹配switch案例完成，整个语句就会结束执行，而不需要明确的break语句。这使得switch声明比C中的声明更安全和更易于使用，并避免switch错误地执行多个案例。

> 注意
>
> 虽然break在Swift中不是必需的，但您可以使用break语句来匹配和忽略特定的案例，或者在案例已经完成执行之前突破匹配的案例。有关详细信息，请参阅切换语句中的中断。

每个案例的主体*必须*包含至少一个可执行语句。写下面的代码是无效的，因为第一种情况是空的：

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a": // Invalid, the case has an empty body
case "A":
    print("The letter A")
default:
    print("Not the letter A")
}
// This will report a compile-time error.
```

与switchC中的陈述不同，这个switch陈述不符合两者"a"和"A"。相反，它会报告case "a":不包含任何可执行语句的编译时错误。这种方法避免了从一种情况到另一种情况的意外损失，并且使得安全代码的意图更加清

晰。

为了使switch与匹配二者的单一情况下"a"和"A"，这两个值组合成的化合物的情况下，用逗号分隔的值。

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

为了便于阅读，复合案例也可以写在多行上。有关复合个案的更多信息，请参阅复合个案。

> 注意
> 要在特定switch情况结束时明确贯彻，请使用fallthrough关键字，如Fallthrough中所述。

间隔匹配

switch可以检查案件中的 值是否包含在间隔中。此示例使用数字间隔为任何大小的数字提供自然语言计数：

```
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).")
// Prints "There are dozens of moons orbiting Saturn."
```

在上面的例子中，approximateCount在switch声明中进行评估。每个case将该值与一个数字或区间进行比较。因为approximateCount落在12和100之间naturalCount的值被分配了值"dozens of"，并且执行从switch语句中转出。

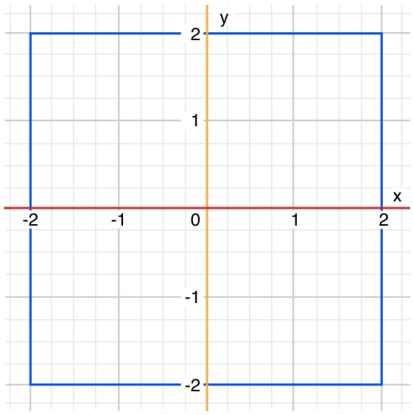
元组

您可以使用元组在同一个switch语句中测试多个值。元组中的每个元素都可以针对不同值或值的间隔进行测试。或者，使用下划线字符（_）（也称为通配符模式）来匹配任何可能的值。

下面的例子用一个（x，y）点表示，表示为一个简单的类型元组(Int, Int)，并将它分类到示例后面的图上。

```
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")
}
// Prints "(1, 1) is inside the box
```



该switch语句确定点是位于原点（0,0），红色x轴上，橙色y轴上，位于原点中心的蓝色4x4框内还是框外。

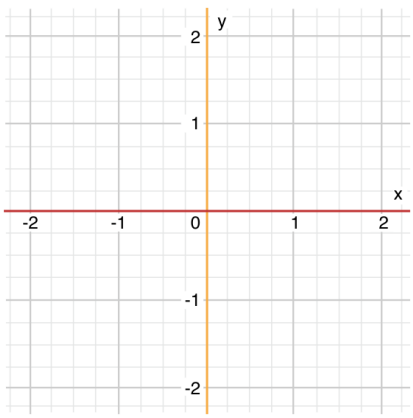
与C不同，Swift允许多个switch案例考虑相同的值或值。事实上，在这个例子中，点（0,0）可以匹配全部*四种*情况。但是，如果可能有多个匹配项，则始终使用第一个匹配的案例。点（0,0）将case (0, 0)首先匹配，因此所有其他匹配的情况都将被忽略。

价值绑定

一个switch案例可以将它匹配的值或值命名为临时常量或变量，以便在案例正文中使用。这种行为被称为*值绑定*，因为值被绑定到案例正文中的临时常量或变量。

下面的例子将一个（x，y）点表示为一个类型的元组(Int, Int)，并将它分类到下面的图表中：

```
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))")
}
// Prints "on the x-axis with an x value of 2"
```



该switch语句确定该点是在红色的x轴上，橙色的y轴上还是其他地方（在两个轴上）。

这三种switch情况下，声明占位符常量x和y，暂时采取在一个或两个元组值的anotherPoint。第一种情况，case (let x, 0)匹配任何一个y值为0的点，并将该点的x值赋给临时常量x。类似地，第二种情况，case (0, let y)匹配x值为的任意点，0并将该点的y值分配给临时常量y。

临时常量声明后

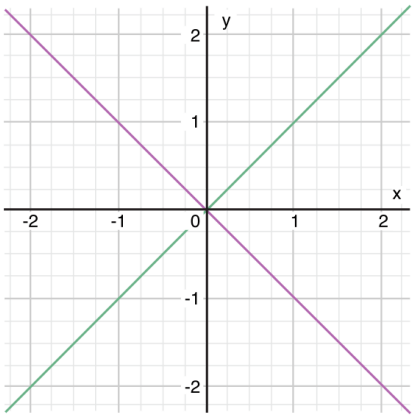
这个switch声明没有default案件。最后一种情况是case let (x, y)，声明一个可以匹配任何值的两个占位符常量的元组。因为anotherPoint它总是一个包含两个值的元组，所以这种情况会匹配所有可能的剩余值，并且default不需要一个例子来使switch语句穷举。

哪里

一个switch案例可以使用一个where条款来检查附加条件。

下面的示例在下面的图中对（x，y）点进行了分类：

```
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")
}
// Prints "(1, -1) is on the line x == -y"
```



该switch声明确定该点是否在绿色对角线上x == y，其中，在紫色对角线上x == -y，还是两者都不是。

这三种switch情况下，声明占位符常量x和y，暂时采取从两元组值yetAnotherPoint。这些常量用作where子句的一部分，以创建动态过滤器。只有当条款的条件评估为该switch值时，该案例才与当前值匹配。
pointwheretrue

和前面的例子一样，最后一个案例匹配所有可能的剩余值，因此default不需要一个案例来使switch语句穷举。

复合案例

共享相同主体的多个开关案例可以通过case在各个模式之间以逗号之间写入多个模式来组合。如果任何模式匹配，则认为该情况匹配。如果列表很长，模式可以写入多行。例如：

```
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")
}
```

```
// Prints "e is a vowel"
```

该switch语句的第一种情况相匹配的英语所有五个元音字母小写。同样，它的第二个案例匹配所有小写英语辅音。最后，该de

复合个案也可以包含值绑定。复合大小写的所有模式都必须包含相同的一组数值绑定，并且每个绑定必须从复合大小写中的所有模式中获取相同类型的值。这可以确保：无论案件的哪个部分匹配，案件正文中的代码都可以访问绑定的值，并且该值始终具有相同的类型。

```
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")
}
// Prints "On an axis, 9 from the origin"
```

在case上面有两个模式：(let distance, 0)在x轴的匹配点和(0, let distance)在y轴一致点。两种模式都包含一个绑定，distance并且distance在这两种模式中都是一个整数 - 这意味着case可以总是访问某个值的代码distance。

控制转移语句

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

- continue
- break
- fallthrough
- return
- throw

的continue，break和fallthrough语句描述如下。该return语句在函数中进行了throw描述，声明在使用投掷函数传播错误中进行了描述。

继续

该continue语句告诉循环停止它正在做的事情，并在循环的下一次迭代开始时重新开始。它说“我已经完成了当前的循环迭代”而不会完全离开循环。

以下示例从小写字符串中删除所有元音和空格以创建一个神秘的拼图短语：

```
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)
// Prints "grtmndsthnklk"
```

上面的代码continue在匹配元音或空格时调用关键字，导致循环的当前迭代立即结束并直接跳转到下一次迭代的开始。

打破

该break语句立即结束整个控制流程语句的执行。当您想要比其他情况下更早地终止或循环语句的执行时，该break语句可以在switch循环语句switch或循环语句中使用。

循环陈述中断

在循环语句中使用时，break立即结束循环的执行，并在控制循环的大括号（}）后将控制权转交给代码。不会执行循环当前迭代的其他代码，也不会开始循环的进一步迭代。

断开开关语句

在switch语句内部使用时，break会导致switch语句立即结束其执行，并在控制switch语句的大括号（}）结束后将控制权转交给代码。

此行为可用于匹配和忽略switch语句中的一个或多个个案。因为斯威夫特的switch陈述是详尽无遗的，并且不允许出现空案，所以有时有必要故意匹配和忽略案件，以便明确表达你的意图。您可以通过将该break陈述写为您想要忽略的整个案例来做到这一点。当该switch语句与该语句匹配时，该break语句内部的switch语句立即结束该语句的执行。

> 注意
>
> 一个switch仅包含注释的情况下被报告为编译时错误。评论不是陈述，也不会导致switch案件被忽略。始终使用break语句来忽略一个switch案例。

以下示例将打开一个Character值并确定它是否以四种语言之一表示数字符号。为简洁起见，在一个switch案例中涵盖了多个值。

```
let numberSymbol: Character = "三"  // Chinese symbol for the number 3
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).")
}
// Prints "The integer value of 三 is 3."
```

本示例检查numberSymbol以确定它是否是数字拉丁，阿拉伯语，中国，泰国或符号1来4。如果找到匹配项，则switch声明的其中一个案例会设置一个可选Int?变量，并将其称为possibleIntegerValue适当的整数值。

After the `switch` statement completes its execution, the example uses optional binding to determine whether a value was found. The `possibleIntegerValue` variable has an implicit initial value of `nil` by virtue of being an optional type, and so the optional binding will succeed only if `possibleIntegerValue` was set to an actual value by one of the `switch` statement's first four cases.

因为Character在上面的例子中列出每个可能的值是不现实的，所以一个defaultcase处理任何不匹配的字符。这种default情况不需要执行任何操作，因此它是以单一break语句作为其主体编写的。只要default匹配，该break语句就会结束switch语句的执行，代码继续执行if let。

下通

在Swift中，switch陈述不会落在每个案例的底部，并进入下一个案例。也就是说，switch只要第一个匹配的案例完成，整个语句就会完成它的执行。相比之下，C要求您break在每个switch案例的末尾插入一个明确的陈述

以防止漏洞。避免默认的延迟意味着Swift switch语句比C语言中的对应语言更加简洁和可预测，因此避免switch了错误地执行多个事件。

如果您需要C风格的贯穿行为，您可以根据具体情况选择加入fallthrough关键字的行为。下面的例子fallthrough用于

```
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)
// Prints "The number 5 is a prime number, and also an integer."
```

这个例子声明了一个新的String变量，description并为它分配一个初始值。该函数然后考虑integerToDescribe使用switch语句的值。如果值integerToDescribe是列表中的一个素数，则该函数将文本附加到末尾description，以注意该数字是素数。然后它使用fallthrough关键字“陷入”default情况。该default案例在描述的末尾添加了一些额外的文本，并且switch声明已完成。

除非integerToDescribe是已知素数列表中的值，否则根本不匹配第一种switch情况。因为没有其他特定的情况，integerToDescribe所以与default案例相匹配。

在之后switch的语句执行完毕，数量的描述是使用打印print(_:separator:terminator:)功能。在这个例子中，这个数字5被正确识别为一个素数。

> 注意
>
> 该fallthrough关键字不检查案件的switch情况下，它会导致执行陷入。的fallthrough关键字简单地使代码执行直接移动到下一个的情况下（或内的语句default的情况下）嵌段，在C的标准switch语句的行为。

标记语句

在Swift中，可以在其他循环和条件语句中嵌套循环和条件语句，以创建复杂的控制流结构。但是，循环和条件语句都可以使用该break语句过早结束它们的执行。因此，明确您希望break语句终止的循环或条件语句有时很有用。同样，如果您有多个嵌套循环，明确continue说明语句应影响哪个循环会很有用。

为了实现这些目标，您可以用*语句标签*标记循环语句或条件语句。使用条件语句，您可以在语句中使用语句标签break来结束标记语句的执行。使用循环语句，可以使用带有breakor continue语句的语句标签来结束或继续执行带标签的语句。

带标签的语句通过将标签放置在与语句的介绍人关键字相同的行上，后跟冒号。下面是while循环语法的一个例子，虽然所有循环和switch语句的原理是相同的：

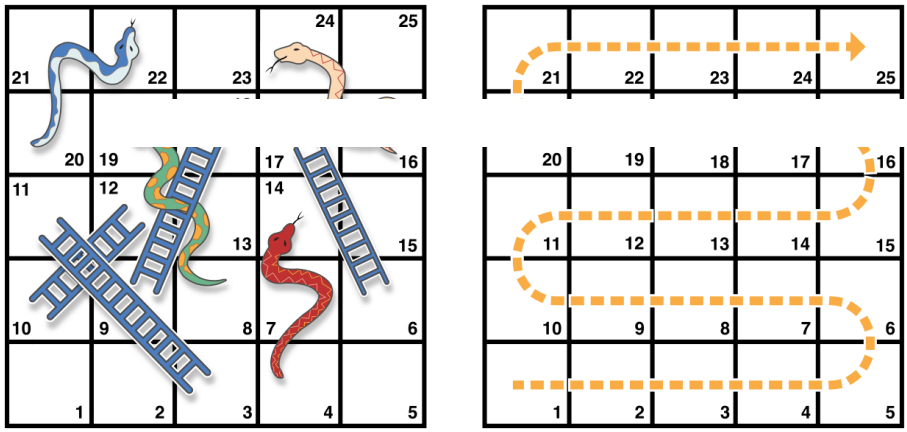
```
标签名称 : while 条件 {
    声明
}
```

下面的示例使用break，并continue声明与标记while的改编版环*蛇和梯子*，你在本章前面看到的比赛。这一次，游戏有一个额外的规则：

- 要赢得比赛，你必须*完全*登上25号广场。

如果一个特定的掷骰子会使你超过25平方，你必须再次掷出，直到你掷出25号方所需的确切数量。

游戏板和以前一样。



的值`finalSquare`，`board`，`square`，并`diceRoll`以同样的方式为前初始化：

```
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
```

该版本的游戏使用`while`循环和`switch`声明来实现游戏的逻辑。该`while`循环有一个声明标签`gameLoop`，表明它是Snakes and Ladders游戏的主要游戏循环。

该`while`循环的条件为`while square != finalSquare`，以反映您必须准确降落在广场25。

```
gameLoop: while square != finalSquare {
    diceRoll += 1
    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]
    }
}
print("Game over!")
```

骰子在每个循环开始时滚动。循环不是立即移动播放器，而是使用`switch`声明来考虑移动的结果并确定移动是否被允许：

- 如果骰子掷出将玩家移动到最后一个方格中，则游戏结束。该`break gameLoop`语句将控制转移到`while`循环外部的第一行代码，从而结束游戏。
- 如果骰子掷出将移动玩家*超越*最后的广场，此举是无效的，玩家需要再次滚动。该`continue gameLoop`语句结束当前`while`循环迭代并开始循环的下一次迭代。
- 在所有其他情况下，掷骰子是一个有效的举措。玩家通过`diceRoll`方块向前移动，游戏逻辑检查任何蛇和梯子。循环结束，并且控制返回到该`while`条件以决定是否需要另一转。

> 注意
>
> 如果`break`上面的陈述没有使用`gameLoop`标签，它将打破`switch`陈述，而不是`while`陈述。使用`gameLoop`标签可以清楚地知道应该终止哪个控制语句。
>
> `gameLoop`调用`continue gameLoop`跳转到循环的下一次迭代时，使用标签并不是必须的。游戏中只有一个循环，因此`continue`语句将影响到哪个循环并不含糊。但是，`gameLoop`在`continue`声明中使用标签没有任

> 何伤害。这样做与标签在break声明旁边的使用保持一致，并有助于使游戏的逻辑更加清晰，便于阅读和理解。

提前退出

甲guard语句，像一个if语句，执行根据表达式的布尔值的语句。您使用guard语句来要求条件必须为真，以便guard执行语句后的代码。与if声明不同，guard声明总是有一个else子句 - else如果条件不正确，子句中的代码将被执行。

```
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"])
// Prints "Hello John!"
// Prints "I hope the weather is nice near you."
greet(person: ["name": "Jane", "location": "Cupertino"])
// Prints "Hello Jane!"
// Prints "I hope the weather is nice in Cupertino."
```

如果符合guard语句的条件，则在guard语句的大括号之后继续执行代码。在guard语句出现的代码块的其余部分，可以使用任何使用可选绑定作为条件一部分赋值的变量或常量。

如果不满足该条件，else则执行分支内的代码。该分支必须传递控制以退出guard语句出现的代码块。它可以做到这一点与控制权转移的语句，如return，break，continue，或者throw，也可以调用一个函数或方法不返回，如fatalError(_:file:line:)。

与使用guard语句进行相同的检查相比，使用需求语句可以提高代码的可读性if。它可以让你编写通常执行的代码，而不用将其包装在一个else块中，并且可以让代码处理违反要求的代码。

检查API可用性

Swift内置了对API可用性检查的支持，确保您不会意外使用给定部署目标上不可用的API。

编译器使用SDK中的可用性信息来验证代码中使用的所有API在项目指定的部署目标上是否可用。如果您尝试使用不可用的API，Swift会在编译时报告错误。

在本页

您使用的*可靠性条件*中的一个if或guard语句来有条件地执行代码块，这取决于你想使用的API是否在运行时可用。编译器在验证该代码块中的API可用时使用可用性条件中的信息。

```
if #available(iOS 10, macOS 10.12, *) {
    // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS
} else {
    // Fall back to earlier iOS and macOS APIs
}
```

上面的可用性条件指定在iOS中，if语句的主体仅在iOS 10及更高版本中执行; 在macOS中，仅在macOS 10.12及更高版本中。最后一个参数*是必需的，并且指定在任何其他平台上，if由目标指定的最小部署目标上的执行主体。

通用形式中，可用性条件采用平台名称和版本的列表。您可以使用平台的名称，如`iOS`，`macOS`，`watchOS`，和`tvOS`-对于完整列表，请参阅声明属性。除了指定主要版本号（如iOS 8或MacOS 10.10）之外，您还可以指定次要版本号，如iOS 11.2.6和macOS 10.13.3。

```
如果 #ava
    语句在API可用时执行
} else {
    如果API不可用，则执行回退语句
}
```