

循环控制



扫码试看/订阅

《Swift核心技术与实战》视频课程

for-in 循环

- 使用 for-in 循环来遍历序列，比如一个范围的数字，数组中的元素或者字符串中的字符。

```
for i in 0...3 {  
    print(i)  
}
```

```
for c in "Hello,World" {  
    print(c)  
}
```

```
let names = ["zhangsan", "lisi", "wangwu", "zhaoliu"]  
for name in names {  
    print(name)  
}
```

for-in 遍历字典

- 当字典遍历时，每一个元素都返回一个 (key, value) 元组，你可以在 for-in 循环体中使用显式命名常量来分解 (key, value) 元组成员。

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

```
for t in numberOfLegs {
    print("\(t.0) has \(t.1) legs")
}
```

for-in 循环

- 如果你不需要序列的每一个值，你可以使用下划线来取代遍历名以忽略值。

```
let base = 3
let power = 5
var answer = 1
for _ in 1...power {
    answer *= base
}
print("\(base) to the power of \(power) is \(answer)")
```

for-in 分段区间

- 使用 `stride(from:to:by:)` 函数来跳过不想要的标记 (开区间)。
- 闭区间也同样适用, 使用 `stride(from:through:by:)` 即可。

```
36 let minuteInterval = 5
37 for tickMark in stride(from: 0, to: 50, by:
    minuteInterval) {
38     print(tickMark)
39 }
```

0
5
10
15
20
25
30
35
40
45

```
36 let minuteInterval = 5
37 for tickMark in stride(from: 0, through: 50,
    by: minuteInterval) {
38     print(tickMark)
39 }
```

0
5
10
15
20
25
30
35
40
45
50

while 循环

- repeat-while 循环 (Objective-C do-while)

```
41  var count = 0
42  repeat {
43      print(count)
44      count += 1
45  } while count < 5
```



```
0
1
2
3
4
```

switch

switch

- switch 语句会将一个值与多个可能的模式匹配。然后基于第一个成功匹配的模式来执行合适的代码块。
- switch 语句一定得是全面的。就是说，给定类型里每一个值都得被考虑到并且匹配到一个 switch 的 case。如果无法提供一个 switch case 所有可能的值，你可以定义一个默认匹配所有的 case 来匹配所有未明确出来的值。这个匹配所有的情况用关键字 default 标记，并且必须在所有 case 的最后出现。

switch

- Objective-C switch 语句如果不全面，仍然可以运行。

```
34 char c = 'z';
35 switch (c) {
36     case 'a':
37         NSLog(@"The first letter of the alphabet");
38         break;
39     case 'z':
40         NSLog(@"The last letter of the alphabet");
41         break;
42 }
```

The last letter of the alphabet

```
49 let someCharacter: Character = "z"
50 switch someCharacter {
51     case "a":
52         print("The first letter of the alphabet")
53     case "z":
54         print("The last letter of the alphabet")
55     }
56
57
```

error: ControlFlow.playground:50:1: error: switch must be exhaustive
switch someCharacter {
^

ControlFlow.playground:50:1: note: do you want to add a default clause?
switch someCharacter {
^

没有隐式贯穿

- 相比 C 和 Objective-C 里的 switch 语句来说，Swift 里的 switch 语句不会默认从匹配 case 的末尾贯穿到下一个 case 里。
- 相反，整个 switch 语句会在匹配到第一个 switch 的 case 执行完毕之后退出，不再需要显式的 break 语句。

```
34 char c = 'z';
35 switch (c) {
36     case 'a':
37         NSLog(@"The first letter of the alphabet");
38     case 'z':
39         NSLog(@"The last letter of the alphabet");
40     default:
41         NSLog(@"Some other character");
42 }
43
```


The last letter of the alphabet
Some other character

```
49 let someCharacter: Character = "z"
50 switch someCharacter {
51     case "a":
52         print("The first letter of the alphabet")
53     case "z":
54         print("The last letter of the alphabet")
55     default:
56         print("Some other character")
57 }
```

The last letter of the alphabet

没有隐式贯穿

- 每一个 case 的函数体必须包含至少一个可执行的语句。
- 在一个 switch 的 case 中匹配多个值可以用逗号分隔，并且可以写成多行。

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
  case "a":  'case' label in a 'switch' should have at least one executable statement
  case "A":
    print("The letter A")
  default:
    print("Not the letter A")
}
```

```
59
60 let anotherCharacter: Character = "a"
61 switch anotherCharacter {
62   case "a", "A":
63     print("The letter A")
64   default:
65     print("Not the letter A")
66 }
```

The letter A

区间匹配

- switch 的 case 的值可以在一个区间中匹配

```
70 let approximateCount = 62
71 let countedThings = "moons orbiting Saturn"
72 var naturalCount: String
73 switch approximateCount {
74 case 0:
75     naturalCount = "no"
76 case 1..<5:
77     naturalCount = "a few"
78 case 5..<12:
79     naturalCount = "several"
80 case 12..<100:
81     naturalCount = "dozens of"
82 case 100..<1000:
83     naturalCount = "hundreds of"
84 default:
85     naturalCount = "many"
86 }
87 print("There are \(naturalCount) \(countedThings).")
```



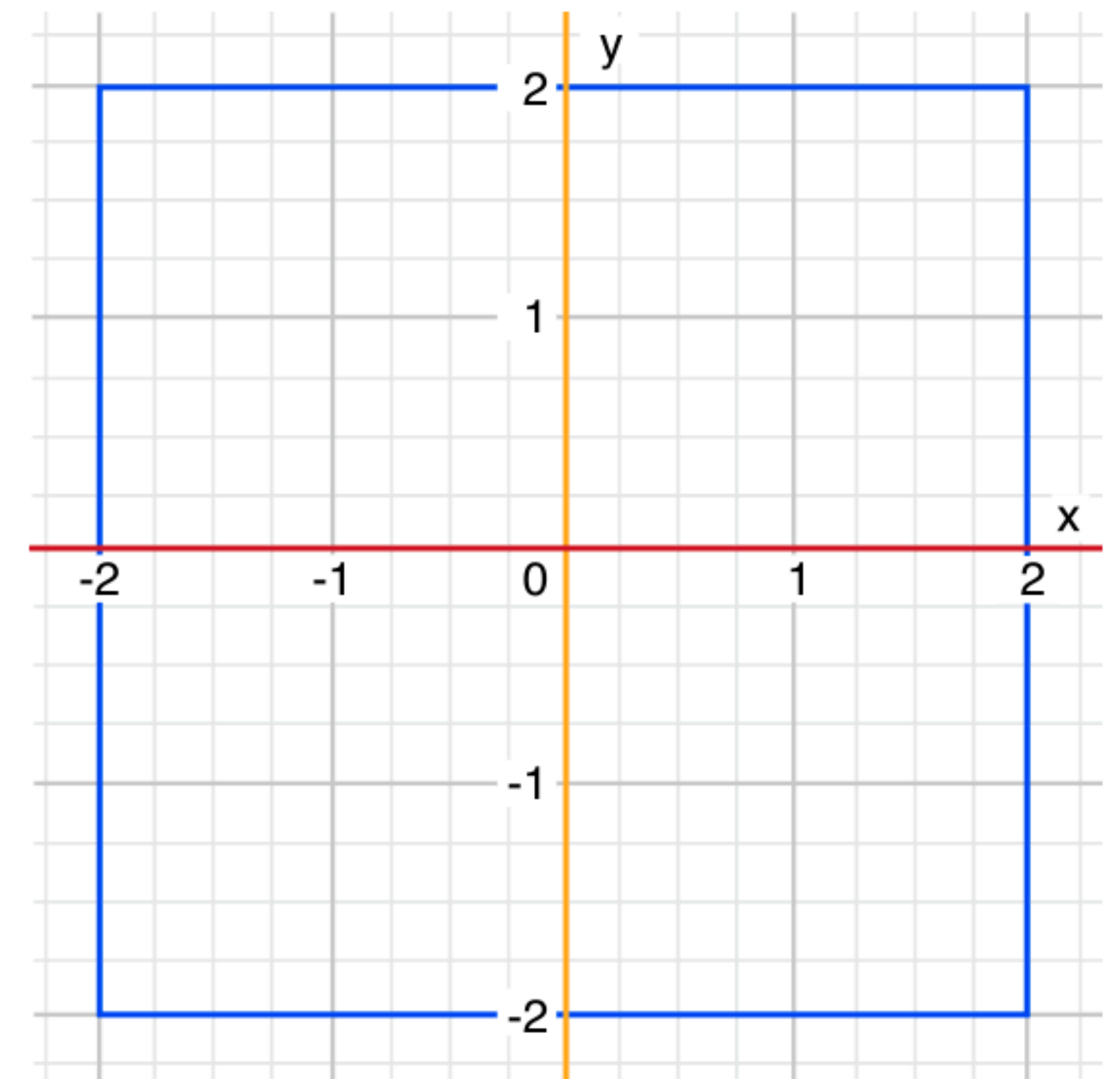
There are dozens of moons orbiting Saturn.

元组匹配

- 你可以使用元组来在一个 switch 语句中测试多个值
- 使用下划线 (_) 来表明匹配所有可能的值

```
--
89 let somePoint = (1, 1)
90 switch somePoint {
91 case (0, 0):
92     print("(0, 0) is at the origin")
93 case (_, 0):
94     print("\(somePoint.0), 0) is on the x-axis")
95 case (0, _):
96     print("0, \(somePoint.1)) is on the y-axis")
97 case (-2...2, -2...2):
98     print("\(somePoint.0), \(somePoint.1)) is inside the box")
99 default:
100     print("\(somePoint.0), \(somePoint.1)) is outside of the
        box")
101 }
```

(1, 1) is inside the box



值绑定

- switch 的 case 可以将匹配到的值临时绑定为一个常量或者变量，来给 case 的函数体使用。
- 如果使用 var 关键字，临时的变量就会以合适的值来创建并初始化。对这个变量的任何改变都只会在 case 的函数体内有效。

```
...
104 let anotherPoint = (2, 0)
105 switch anotherPoint {
106 case (let x, 0):
107     print("on the x-axis with an x value of \(x)")
108 case (0, let y):
109     print("on the y-axis with a y value of \(y)")
110 case let (x, y):
111     print("somewhere else at (\(x), \(y))")
112 }
```



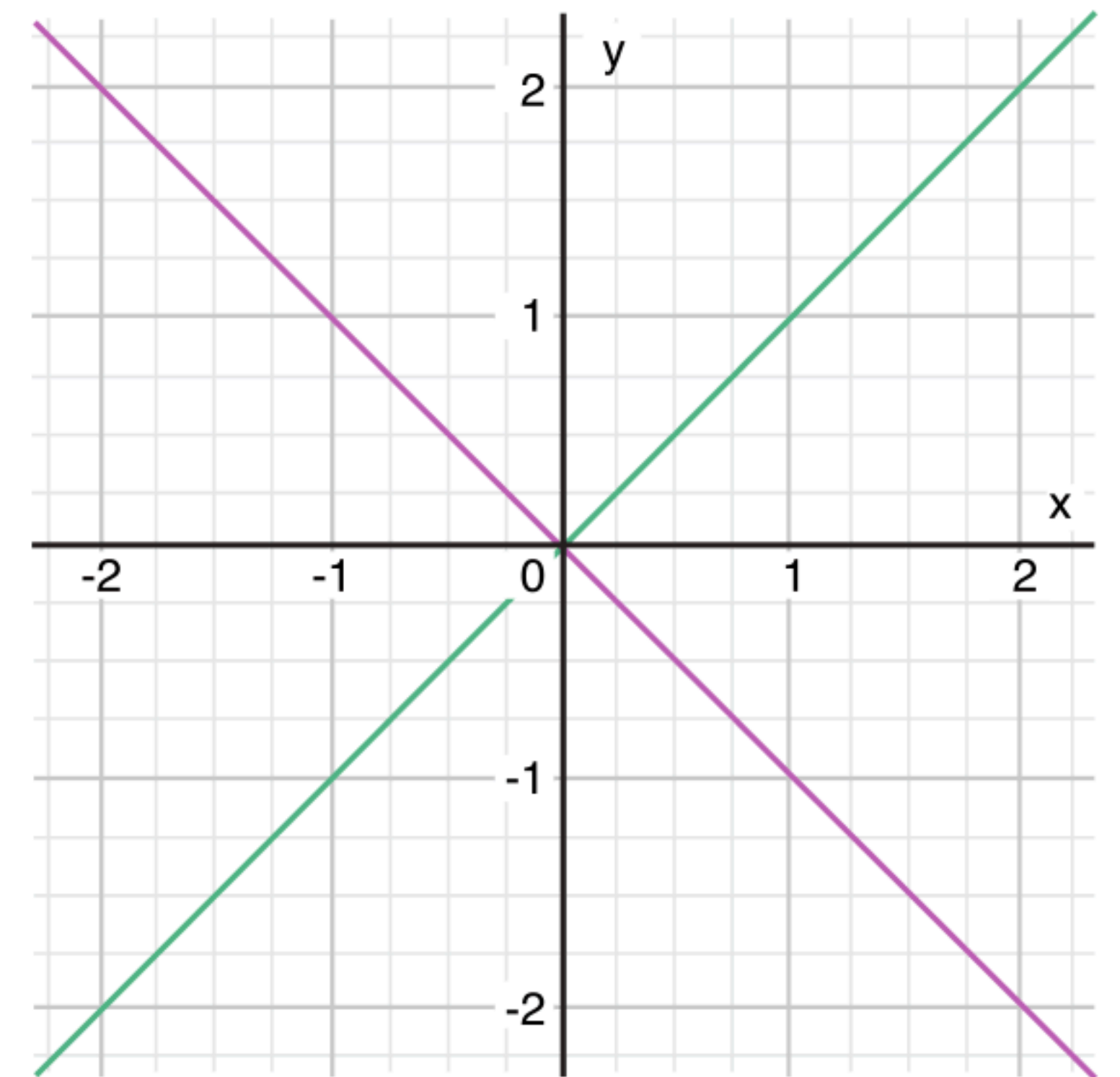
on the x-axis with an x value of 2

where 字句

- switch case 可以使用 where 分句来检查是否符合特定的约束

```
114
115 let yetAnotherPoint = (1, -1)
116 switch yetAnotherPoint {
117   case let (x, y) where x == y:
118     print("\(x), \(y)) is on the line x == y")
119   case let (x, y) where x == -y:
120     print("\(x), \(y)) is on the line x == -y")
121   case let (x, y):
122     print("\(x), \(y)) is just some arbitrary point")
123 }
```

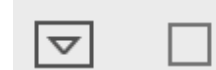
(1, -1) is on the line $x == -y$



复合匹配

- 多种情形共享同一个函数体的多个情况可以在 case 后写多个模式来复合，在每个模式之间用逗号分隔。如果任何一个模式匹配了，那么这个情况都会被认为是匹配的。如果模式太长，可以把它们写成多行。

```
127 let someCharacter: Character = "e"
128 switch someCharacter {
129     case "a", "e", "i", "o", "u":
130         print("\(someCharacter) is a vowel")
131     case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
132         "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
133         print("\(someCharacter) is a consonant")
134     default:
135         print("\(someCharacter) is not a vowel or a consonant")
136 }
```

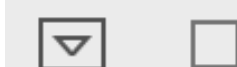


e is a vowel

复合匹配 - 值绑定

- 复合匹配同样可以包含值绑定。所有复合匹配的模式都必须包含相同的值绑定集合，并且复合情况中的每一个绑定都得有相同的类型格式。这才能确保无论复合匹配的那部分命中了，接下来的函数体中的代码都能访问到绑定的值并且值的类型也都相同。

```
137  
138 let stillAnotherPoint = (9, 0)  
139 switch stillAnotherPoint {  
140 case (let distance, 0), (0, let distance):  
141     print("On an axis, \ (distance) from the origin")  
142 default:  
143     print("Not on an axis")  
144 }
```



On an axis, 9 from the origin

控制转移

控制转移

- continue
- break
- fallthrough
- return
- throw

continue

- `continue` 语句告诉循环停止正在做的事情并且再次从头开始循环的下一次遍历。它是说“我不再继续当前的循环遍历了”而不是离开整个的循环。

break

- break 语句会立即结束整个控制流语句。当你想要提前结束 switch 或者循环语句或者其他情况时可以在 switch 语句或者循环语句中使用 break 语句。
- 当在循环语句中使用时，break 会立即结束循环的执行，并且转移控制到循环结束花括号（ } ）后的第一行代码上。当前遍历循环里的其他代码都不会被执行，并且余下的遍历循环也不会开始了。
- 当在 switch 语句里使用时，break 导致 switch 语句立即结束它的执行，并且转移控制到 switch 语句结束花括号（ } ）之后的第一行代码上。

fallthrough

- 如果你确实需要 C 或者 Objective-C 风格的贯穿行为，你可以选择在 switch 每个 case 末尾使用 fallthrough 关键字。

```
146 let integerToDescribe = 5
147 var description = "The number \(integerToDescribe) is"
148 switch integerToDescribe {
149     case 2, 3, 5, 7, 11, 13, 17, 19:
150         description += " a prime number, and also"
151         fallthrough
152     default:
153         description += " an integer."
154 }
155 print(description)
```



```
⌵ □
The number 5 is a prime number, and also an integer.
|
```

语句标签

- 可以用语句标签来给循环语句或者条件语句做标记。在一个条件语句中，你可以使用一个语句标签配合 `break` 语句来结束被标记的语句。在循环语句中，你可以使用语句标签来配合 `break` 或者 `continue` 语句来结束或者继续执行被标记的语句。

guard 和检查 API 可用性

guard

- guard 语句，类似于 if 语句，基于布尔值表达式来执行语句。使用 guard 语句来要求一个条件必须是真才能执行 guard 之后的语句。与 if 语句不同，guard 语句总是有一个 else 分句——else 分句里的代码会在条件不为真的时候执行。

检查 API 的可用性

- Swift 拥有内置的对 API 可用性的检查功能，它能够确保你不会悲剧地使用了对部属目标不可用的 API。
- 你可以在 if 或者 guard 语句中使用一个可用性条件来有条件地执行代码，基于在运行时你想用的哪个 API 是可用的。

```
if #available(platform name version, ..., *) {  
    statements to execute if the APIs are available  
} else {  
    fallback statements to execute if the APIs are unavailable  
}
```

```
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  
}
```

模式和模式匹配

模式

- 模式代表单个值或者复合值的结构。
- 例如，元组 (1, 2) 的结构是由逗号分隔的，包含两个元素的列表。因为模式代表一种值的结构，而不是特定的某个值，你可以利用模式来匹配各种各样的值。比如，(x, y) 可以匹配元组 (1, 2)，以及任何含两个元素的元组。除了利用模式匹配一个值以外，你可以从复合值中提取出部分或全部值，然后分别把各个部分的值和一个常量或变量绑定起来。

模式分类

- Swift 中的模式分为两类：一种能成功匹配任何类型的值，另一种在运行时匹配某个特定值时可能会失败。
 - 第一类模式用于解构简单变量、常量和可选绑定中的值。此类模式包括通配符模式、标识符模式，以及包含前两种模式的值绑定模式和元组模式。你可以为这类模式指定一个类型标注，从而限制它们只能匹配某种特定类型的值。
 - 第二类模式用于全模式匹配，这种情况下你试图匹配的值在运行时可能不存在。此类模式包括枚举用例模式、可选模式、表达式模式和类型转换模式。你在 switch 语句的 case 标签中，do 语句的 catch 子句中，或者在 if、while、guard 和 for-in 语句的 case 条件句中使用这类模式。

模式分类

- 通配符模式 (Wildcard Pattern)
- 标识符模式 (Identifier Pattern)
- 值绑定模式 (Value-Binding Pattern)
- 元组模式 (Tuple Pattern)
- 枚举用例模式 (Enumeration Case Pattern)
- 可选项模式 (Optional Pattern)
- 类型转换模式 (Type-Casting Pattern)
- 表达式模式 (Expression Pattern)

通配符模式 (Wildcard Pattern)

- 通配符模式由一个下划线 (_) 构成, 用于匹配并忽略任何值。当你想忽略被匹配的值时可以使用该模式。

```
for _ in 1...3 {  
    // ...  
}
```

标识符模式 (Identifier Pattern)

- 标识符模式匹配任何值，并将匹配的值和一个变量或常量绑定起来。

```
let someValue = 42|
```

值绑定模式 (Value-Binding Pattern)

- 值绑定模式把匹配到的值绑定给一个变量或常量。把匹配到的值绑定给常量时，用关键字 `let`，绑定给变量时，用关键字 `var`。

```
let point = (3, 2)
switch point {
// 将 point 中的元素绑定到 x 和 y
case let (x, y):
    print("The point is at \(x), \(y).")
}
```

元组模式 (Tuple Pattern)

- 元组模式是由逗号分隔的，具有零个或多个模式的列表，并由一对圆括号括起来。元组模式匹配相应元组类型的值。
- 你可以使用类型标注去限制一个元组模式能匹配哪种元组类型。例如，在常量声明 `let (x, y): (Int, Int) = (1, 2)` 中的元组模式 `(x, y): (Int, Int)` 只匹配两个元素都是 `Int` 类型的元组。
- 当元组模式被用于 `for-in` 语句或者变量和常量声明时，它仅可以包含通配符模式、标识符模式、可选模式或者其他包含这些模式的元组模式。

```
164
165 let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]
166 for (x, 0) in points {
167     /* ... */
168 }
169
```

error: ControlFlow.playground:167:9: error: expected pattern
for (x, 0) in points {
 ^

```
104
165 let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]
166 for (x, y) in points where y == 0 {
167     print("\(x) and \(y)")
168 }
169
```

0 and 0
1 and 0
2 and 0

枚举用例模式（Enumeration Case Pattern）

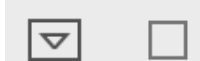
- 枚举用例模式匹配现有的某个枚举类型的某个用例。枚举用例模式出现在 switch 语句中的 case 标签中，以及 if、while、guard 和 for-in 语句的 case 条件中。

可选项目模式 (Optional Pattern)

- 可选项模式匹配 `Optional<Wrapped>` 枚举在 `some(Wrapped)` 中包装的值。
- 可选项目模式为 `for-in` 语句提供了一种迭代数组的简便方式，只为数组中非 `nil` 的元素执行循环体。

```
170 let someOptional: Int? = 42
171 // Match using an enumeration case pattern.
172 if case .some(let x) = someOptional {
173     print(x)
174 }
175
176 // Match using an optional pattern.
177 if case let x? = someOptional {
178     print(x)
179 }
```

```
let arrayOfOptionalInts: [Int?] = [nil, 2, 3, nil, 5]
// Match only non-nil values.
for case let number? in arrayOfOptionalInts {
    print("Found a \(number)")
}
```



42

42

类型转换模式 (Type-Casting Pattern)

- 有两种类型转换模式，is 模式和 as 模式。is 模式只出现在 switch 语句中的 case 标签中。is 模式和 as 模式形式如下：
 - is 类型
 - 模式 as 类型

类型转换模式 (Type-Casting Pattern)

- is 模式仅当一个值的类型在运行时和 is 模式右边的指定类型一致，或者是其子类的情况下，才会匹配这个值。is 模式和 is 运算符有相似表现，它们都进行类型转换，但是 is 模式没有返回类型。
- as 模式仅当一个值的类型在运行时和 as 模式右边的指定类型一致，或者是其子类的情况下，才会匹配这个值。如果匹配成功，被匹配的值的类型被转换成 as 模式右边指定的类型。

类型转换模式 (Type-Casting Pattern)

```
protocol Animal {
    var name: String { get }
}

struct Dog: Animal {
    var name: String {
        return "dog"
    }
    var runSpeed: Int
}

struct Bird: Animal {
    var name: String {
        return "bird"
    }

    var flightHeight: Int
}

struct Fish: Animal {
    var name: String {
        return "fish"
    }
    var depth: Int
}
```

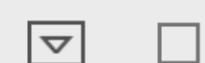
```
213
214 let animals:[Any] = [Dog(runSpeed: 55), Bird(flightHeight:
215     2000), Fish(depth: 100)]
216
217 for animal in animals {
218     switch animal {
219     case let dog as Dog:
220         print("\(dog.name) can run \(dog.runSpeed)")
221     case let fish as Fish:
222         print("\(fish.name) can dive depth \(fish.depth)")
223     case is Bird:
224         print("bird can fly!")
225     default:
226         print("unknown animal!")
227     }
228 }
```

```
dog can run 55
bird can fly!
fish can dive depth 100
```

表达式模式 (Expression Pattern)

- 表达式模式代表表达式的值。表达式模式只出现在 switch 语句中的 case 标签中。
- 表达式模式代表的表达式会使用 Swift 标准库中的 `~=` 运算符与输入表达式的值进行比较。如果 `~=` 运算符返回 `true`，则匹配成功。默认情况下，`~=` 运算符使用 `==` 运算符来比较两个相同类型的值。它也可以将一个整型数值与一个 Range 实例中的一段整数区间做匹配。

```
229 let point = (1, 2)|
230 switch point {
231     case (0, 0):
232         print("(0, 0) is at the origin.")
233     case (-2...2, -2...2):
234         print("\(point.0), \(point.1) is near the origin.")
235     default:
236         print("The point is at \(point.0), \(point.1).")
237 }
```



(1, 2) is near the origin.

表达式模式 (Expression Pattern)

- 可以重载 `~=` 运算符来提供自定义的表达式匹配行为

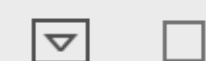
```
239 func ~= (pattern: String, value: Int) -> Bool {  
240     return pattern == "\(value)"  
241 }  
242 switch point {  
243 case ("0", "0"):  
244     print("(0, 0) is at the origin.")  
245 default:  
246     print("The point is at \(point.0), \(point.1).")  
247 }
```

The point is at (1, 2).

表达式模式 (Expression Pattern)

- 自定义类型默认也是无法进行表达式模式匹配的，也需要重载 `~=` 运算符。

```
---
249 struct Employee {
250     var salary: Float
251 }
252 let e = Employee(salary: 9999)
253 func ~= (lhs: Range<Float>, rhs: Employee) -> Bool {
254     return lhs.contains(rhs.salary)
255 }
256 switch e {
257 case 0.0..<1000:
258     print("艰难生活")
259 case 1000..<5000:
260     print("小康社会")
261 case 5000..<10000:
262     print("活得很滋润")
263 default:
264     break
265 }
```



活得很滋润



扫码试看/订阅

《Swift核心技术与实战》视频课程