翻译：coverxit 校对：numbbbbb

语句

本页包含内容：

在 Swift 中，有两种类型的语句：简单语句和控制流语句。简单语句是最常见的，用于构造表达式和声明。控制流语句则用于控制程序执行的流程，Swift 中有三种类型的控制流语句：循环语句、分支语句和控制传递语句。

循环语句用于重复执行代码块；分支语句用于执行满足特定条件的代码块；控制传递语句则用于修改代码的执行顺序。在稍后的叙述中，将会详细地介绍每一种类型的控制流语句。

是否将分号(;)添加到语句的结尾处是可选的。但若要在同一行内写多条独立语句，请务必使用分号。

GRAMMAR OF A STATEMENT

statement → *expression*; *opt*

statement → *declaration*; *opt*

statement → *loop-statement*; *opt*

statement → *branch-statement*; *opt*

statement → *labeled-statement*

statement → *control-transfer-statement*; *opt*

statement → *statment statements*; *opt*

循环语句

取决于特定的循环条件，循环语句允许重复执行代码块。Swift 提供四种类型的循环语句：`for`语句、`for-in`语句、`while`语句和`do-while`语句。

通过`break`语句和`continue`语句可以改变循环语句的控制流。有关这两条语句，请参考Break 语句和Continue 语句。

GRAMMAR OF A LOOP STATEMENT

loop-statement → *for-statement*

loop-statement → *for-in-statement*

loop-statement → *while-statement*

loop-statement → *do-while-statement*

For 语句

`for`语句允许在重复执行代码块的同时，递增一个计数器。

`for`语句的形式如下：

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

initialzation、*condition*和*increment*之间的分号，以及包围循环体*statements*的大括号都是不可省略的。

`for`语句的执行流程如下：

1. *initialzation*只会被执行一次，通常用于声明和初始化在接下来的循环中需要使用的变量。
2. 计算*condition*表达式： 如果为真(`true`)，*statements*将会被执行，然

后转到第3步。如果为假(`false`)，*statements*和*increment*都不会被执行，`for`至此执行完毕。

3. 计算*increment*表达式，然后转到第2步。

定义在*initialzation*中的变量仅在`for`语句的作用域以内有效。*condition*表达式的值的类型必须遵循`LogicValue`协议。

GRAMMAR OF A FOR STATEMENT

for-statement → **for** *for-init opt* **;** *expression opt* **;** *expression opt code-block*

for-statement → **for** **(** *for-init opt* **;** *expression opt* **;** *expression opt* **)** *code-block*

for-statement → *variable-declaration* | *expression-list*

For-In 语句

`for-in`语句允许在重复执行代码块的同时，迭代集合(或遵循`Sequence`协议的任意类型)中的每一项。

`for-in`语句的形式如下：

```
for `item` in `collection` {
    `statements`
}
```

`for-in`语句在循环开始前会调用*collection*表达式的`generate`方法来获取一个生成器类型（这是一个遵循`Generator`协议的类型）的值。接下来循环开始，调用*collection*表达式的`next`方法。如果其返回值不是`None`，它将会被赋给*item*，然后执行*statements*，执行完毕后回到循环开始处；否则，将不会赋值给*item*也不会执行*statements*，`for-in`至此执行完毕。

GRAMMAR OF A FOR-IN STATEMENT

for-in-statement → **for** *pattern* **in** *expression code-block*

While 语句

`while`语句允许重复执行代码块。

`while`语句的形式如下：

```
while `condition` {
```

```
    `statements`
}
```
`while`语句的执行流程如下：

1. 计算*condition*表达式： 如果为真(`true`)，转到第2步。如果为假(`false`)，`while`至此执行完毕。
2. 执行*statements*，然后转到第1步。

由于*condition*的值在*statements*执行前就已计算出，因此`while`语句中的*statements*可能会被执行若干次，也可能不会被执行。

*condition*表达式的值的类型必须遵循`LogicValue`协议。同时，*condition*表达式也可以使用可选绑定，请参考可选绑定待添加链接。

GRAMMAR OF A WHILE STATEMENT

while-statement → **while** *while-condition code-block*

while-condition → *expression* | *declaration*

Do-While 语句

`do-while`语句允许代码块被执行一次或多次。

`do-while`语句的形式如下：

```
do {
    `statements`
} while `condition`
```
`do-while`语句的执行流程如下：

1. 执行*statements*，然后转到第2步。
2. 计算*condition*表达式： 如果为真(`true`)，转到第1步。如果为假(`false`)，`do-while`至此执行完毕。

由于*condition*表达式的值是在*statements*表达式执行后才计算出，因此*do-while*语句中的*statements*至少会被执行一次。

*condition*表达式的值的类型必须遵循`LogicValue`协议。同时，*condition*表达式也可以使用可选绑定，请参考可选绑定待添加链接。

GRAMMAR OF A DO-WHILE STATEMENT

do-while-statement → **do** *code-block* **while** *while-condition*

分支语句

取决于一个或者多个条件的值，分支语句允许程序执行指定部分的代码。显然，分支语句中条件的值将会决定如何分支以及执行哪一块代码。Swift提供两种类型的分支语句：`if`语句和`switch`语句。

`switch`语句中的控制流可以用`break`语句修改，请参考Break 语句。

GRAMMAR OF A BRANCH STATEMENT

branch-statement → *if-statement*

branch-statement → *switch-statement*

If 语句

取决于一个或多个条件的值，`if`语句将决定执行哪一块代码。

`if`语句有两种标准形式，在这两种形式里都必须有大括号。

第一种形式是当且仅当条件为真时执行代码，像下面这样：

```
if `condition` {
    `statements`
}
```

第二种形式是在第一种形式的基础上添加*else*语句，当只有一个*else*语句时，像下面这样：

```
if `condition` {
    `statements to execute if condition is true`
} else {
    `statements to execute if condition is false`
}
```

同时，*else*语句也可包含`if`语句，从而形成一条链来测试更多的条件，像

下面这样：

```
if `condition 1` {
    `statements to execute if condition 1 is true`
} else if `condition 2` {
    `statements to execute if condition 2 is true`
}
else {
    `statements to execute if both conditions are
false`
}
```

`if`语句中条件的值的类型必须遵循`LogicValue`协议。同时，条件也可以使用可选绑定，请参考可选绑定待添加链接。

GRAMMAR OF AN IF STATEMENT

if-statement → **if** *if-condition code-block else-clause opt*

if-condition → *expression* | *declaration*

else-clause → **else** *code-block* | **else** *if-statement opt*

Switch 语句

取决于`switch`语句的控制表达式(*control expression*)，`switch`语句将决定执行哪一块代码。

`switch`语句的形式如下：

```
switch `control expression` {
    case `pattern 1`:
        `statements`
    case `pattern 2` where `condition`:
        `statements`
    case `pattern 3` where `condition`,
    `pattern 4` where `condition`:
        `statements`
    default:
        `statements`
```

```
}
```
`switch`语句的控制表达式*(control expression)*会首先被计算，然后与每一个*case*的模式(pattern)进行匹配。如果匹配成功，程序将会执行对应的*case*块里的*statements*。另外，每一个*case*块都不能为空，也就是说在每一个*case*块中至少有一条语句。如果你不想在匹配到的*case*块中执行代码，只需在块里写一条`break`语句即可。

可以用作控制表达式的值是十分灵活的，除了标量类型(scalar types，如`Int`、`Character`)外，你可以使用任何类型的值，包括浮点数、字符串、元组、自定义类的实例和可选(optional)类型，甚至是枚举类型中的成员值和指定的范围(range)等。关于在`switch`语句中使用这些类型，请参考控制流一章的Switch。

你可以在模式后面添加一个起保护作用的表达式(guard expression)。起保护作用的表达式是这样构成的：关键字`where`后面跟着一个作为额外测试条件的表达式。因此，当且仅当控制表达式匹配一个*case*的某个模式且起保护作用的表达式为真时，对应*case*块中的*statements*才会被执行。在下面的例子中，控制表达式只会匹配含两个相等元素的元组，如`(1, 1)`：

```
case let (x, y) where x == y:
}
```
正如上面这个例子，也可以在模式中使用`let`（或`var`）语句来绑定常量（或变量）。这些常量（或变量）可以在其对应的起保护作用的表达式和其对应的*case*块里的代码中引用。但是，如果*case*中有多个模式匹配控制表达式，那么这些模式都不能绑定常量（或变量）。

`switch`语句也可以包含默认(`default`)块，只有其它*case*块都无法匹配控制表达式时，默认块中的代码才会被执行。一个`switch`语句只能有一个默认块，而且必须在`switch`语句的最后面。

尽管模式匹配操作实际的执行顺序，特别是模式的计算顺序是不可知的，但是 Swift 规定`switch`语句中的模式匹配的顺序和书写源代码的顺序保持一致。因此，当多个模式含有相同的值且能够匹配控制表达式时，程序只会执行源代码中第一个匹配的*case*块中的代码。

Switch 语句必须是完备的

在 Swift 中，`switch`语句中控制表达式的每一个可能的值都必须至少有一个*case*块与之对应。在某些情况下（例如，表达式的类型是`Int`），你可以使用默认块满足该要求。

不存在隐式的贯穿(fall through)

当匹配的*case*块中的代码执行完毕后，程序会终止`switch`语句，而不会继续执行下一个*case*块。这就意味着，如果你想执行下一个*case*块，需要显式地在你需要的*case*块里使用`fallthrough`语句。关于`fallthrough`语句的更多信息，请参考Fallthrough 语句。

GRAMMAR OF A SWITCH STATEMENT

switch-statement → **switch** *expression* **{** *switch-cases opt* **}**

switch-cases → *switch-case switch-cases opt*

switch-case → *case-label statement* | *default-label statements*

switch-case → *case-label* **;** | *default-label* **;**

case-label → **case** *case-item-list* **:**

case-item-list → *pattern guard-clause opt* | *pattern guard-clause opt*, *case-item-list*

default-label → **default :**

guard-clause → **where** *guard-expression*

guard-expression → *expression*
带标签的语句

你可以在循环语句或`switch`语句前面加上标签，它由标签名和紧随其后的冒号(:)组成。在`break`和`continue`后面跟上标签名可以显式地在循环语句或`switch`语句中更改控制流，把控制权传递给指定标签标记的语句。关于这两条语句用法，请参考Break 语句和Continue 语句。

标签的作用域是该标签所标记的语句之后的所有语句。你可以不使用带标

签的语句，但只要使用它，标签名就必唯一。

关于使用带标签的语句的例子，请参考控制流一章的带标签的语句待添加链接。

GRAMMAR OF A LABELED STATEMENT

labeled-statement → *statement-label loop-statement* | *statement-label switch-statement*

statement-label → *label-name* :

label-name → *identifier*

控制传递语句

通过无条件地把控制权从一片代码传递到另一片代码，控制传递语句能够改变代码执行的顺序。Swift 提供四种类型的控制传递语句：`break`语句、`continue`语句、`fallthrough`语句和`return`语句。

GRAMMAR OF A CONTROL TRANSER STATEMENT

control-transfer-statement → *break-statement*

control-transfer-statement → *continue-statement*

control-transfer-statement → *fallthrough-statement*

control-transfer-statement → *return-statement*

Break 语句

`break`语句用于终止循环或`switch`语句的执行。使用`break`语句时，可以只写`break`这个关键词，也可以在`break`后面跟上标签名(label name)，像下面这样：

```
break
break `label name`
```

当`break`语句后面带标签名时，可用于终止由这个标签标记的循环或

`switch`语句的执行。

而当只写`break`时，则会终止`switch`语句或上下文中包含`break`语句的最内层循环的执行。

在这两种情况下，控制权都会被传递给循环或`switch`语句外面的第一行语句。

关于使用`break`语句的例子，请参考控制流一章的Break待添加链接和带标签的语句待添加链接。

GRAMMAR OF A BREAK STATEMENT

break-statement → **break** *label-name opt*

Continue 语句

`continue`语句用于终止循环中当前迭代的执行，但不会终止该循环的执行。使用`continue`语句时，可以只写`continue`这个关键词，也可以在`continue`后面跟上标签名(label name)，像下面这样：

```
continue
continue `label name`
```

当`continue`语句后面带标签名时，可用于终止由这个标签标记的循环中当前迭代的执行。

而当只写`break`时，可用于终止上下文中包含`continue`语句的最内层循环中当前迭代的执行。

在这两种情况下，控制权都会被传递给循环外面的第一行语句。

在`for`语句中，`continue`语句执行后，*increment*表达式还是会被计算，这是因为每次循环体执行完毕后*increment*表达式都会被计算。

关于使用`continue`语句的例子，请参考控制流一章的Continue待添加链接和带标签的语句待添加链接。

GRAMMAR OF A CONTINUE STATEMENT

continue-statement → **continue** *label-name opt*

Fallthrough 语句

`fallthrough`语句用于在`switch`语句中传递控制权。`fallthrough`语句会把控制权从`switch`语句中的一个*case*传递给下一个*case*。这种传递是无条件的，即使下一个*case*的值与`switch`语句的控制表达式的值不匹配。

`fallthrough`语句可出现在`switch`语句中的任意*case*里，但不能出现在最后一个*case*块中。同时，`fallthrough`语句也不能把控制权传递给使用了可选绑定的*case*块。

关于在`switch`语句中使用`fallthrough`语句的例子，请参考控制流一章的控制传递语句待添加链接。

GRAMMAR OF A FALLTHROUGH STATEMENT

continue-statement → **fallthrough**

Return 语句

`return`语句用于在函数或方法的实现中将控制权传递给调用者，接着程序将会从调用者的位置继续向下执行。

使用`return`语句时，可以只写`return`这个关键词，也可以在`return`后面跟上表达式，像下面这样：

```
return
return `expression`
```

当`return`语句后面带表达式时，表达式的值将会返回给调用者。如果表达式值的类型与调用者期望的类型不匹配，Swift 则会在返回表达式的值之前将表达式值的类型转换为调用者期望的类型。

而当只写`return`时，仅仅是将控制权从该函数或方法传递给调用者，而不返回一个值。（这就是说，该函数或方法的返回类型为`Void`或`()`）

GRAMMAR OF A RETURN STATEMENT

return-statement → **return** *expression opt*