Swift 程式語言

正體中文版

目錄

《正體中文版蘋果 **Swift** 官方教學》

正體中文版蘋果 Swift 官方教學《The Swift Programming Language》

當前階段

簡體中文版已由[numbbbbb]團隊翻譯完成，文字部分已經全部正體中文化，而術語我已經依據 translation.json 裡面的對應完成自動替換成台灣習慣的用法。 第二章以後尚未校稿，歡迎校對，可以隨意提 issue。

譯者記錄

    - 模式 (tommy60703)
    - 泛型參數 (tommy60703)
    - 語法總結 (tommy60703)

貢獻力量

如果想做出貢獻的話，你可以：

- 幫忙校對，挑錯別字、語病等等
- 提出修改建議
- 提出術語翻譯建議

翻譯建議

如果你願意一起校對的話，請仔細閱讀：

- 使用 markdown 進行翻譯，文件名必須使用英文，因為中文的話 gitbook 編譯的時候會出問題
- 引號請使用「」和『』
- fork 過去之後新建一個分支進行翻譯，完成後 pull request 這個分支，沒問題的話我會合併到 master 分支中
- 有其他任何問題都歡迎發 issue，我看到了會盡快回覆

謝謝！

關於術語

翻譯術語的時候請參考這個流程：

- 盡量保證與台灣習慣術語和已翻譯的內容一致
- 盡量先搜尋，一般來說程式語言的大部分術語是一樣的，可以參考這個網站
- 如果以上兩條都沒有找到合適的結果，請自己決定一個合適的翻譯或者直接使用英文原文，後期校對的時候會進行統一
- 校稿時，若有發現沒有被翻譯成台灣術語的大陸術語，可以將它新增到 translation.json 中
- 可以主動提交替換過的文本給我，或是僅提交新增過的 translation.json 也可，我會再進行文本的替換
- 請務必確定提交的翻譯對照組不會造成字串循環替代（ex: 因為「類」->「類別」，造成下次再執行自動翻譯時「類別」又變成「類別別」）

對翻譯有任何意見都歡迎發 issue，我看到了會盡快回覆

參考流程

有些朋友可能不太清楚如何幫忙翻譯，我這裡寫一個簡單的流程，大家可以參考一下：

1. 首先 fork 我的項目
2. 把 fork 過去的項目也就是你的項目 clone 到你的本地
3. 在命令行運行 `git branch develop` 來創建一個新分支
4. 運行 `git checkout develop` 來切換到新分支
5. 運行 `git remote add upstream https://github.com/tommy60703/the-swift-programming-language-in-traditional-chinese` 把我的庫添加為遠端庫
6. 運行 `git remote update` 更新
7. 運行 `git fetch upstream master` 拉取我的庫的更新到本地
8. 運行 `git rebase upstream/master` 將我的更新合並到你的分支

這是一個初始化流程，只需要做一遍就行，之後請一直在 develop 分支進行修改。

如果修改過程中我的庫有了更新，請重復 6、7、8 步。

修改之後，首先 push 到你的庫，然後登錄 GitHub，在你的 repo 的首頁可以看到一個 `pull request` 按鈕，點擊它，填寫一些說明資訊，然後提交即可。

開源協議

基於WTFPL協議開源。

歡迎使用 **Swift**

在本章中你將了解 Swift 的特性和開發歷史，並對 Swift 有一個初步的了解。

> 翻譯：tommy60703 校對：tommy60703, petertom51

關於 Swift

Swift 是一種新的程式語言，用於編寫 iOS 和 OS X 應用程式。Swift 結合了 C 和 Objective-C 的優點並且不受 C 相容性的限制。Swift 採用安全的程式設計模式並添加了很多新特性，這將使程式設計更簡單，更靈活，也更有趣。Swift 是基於成熟而且倍受喜愛的 Cocoa 和 Cocoa Touch 框架（framework），它的降臨將重新定義軟體開發。

Swift 的開發從很久之前就開始了。為了給 Swift 打好基礎，蘋果公司改進了編譯器，除錯器和框架結構。我們使用自動引用計數（Automatic Reference Counting, ARC）來簡化記憶體管理。我們在 Foundation 和 Cocoa 的基礎上構建框架並將其標準化。Objective-C 本身支援區塊、集合語法和模組，所以框架可以輕鬆支援現代程式語言技術。正是得益於這些基礎工作，我們現在才能發佈這樣一個用於未來蘋果軟體開發的新語言。

Objective-C 開發者對 Swift 並不會感到陌生。它採用了 Objective-C 的命名參數以及動態物件模型，可以無縫對接到現有的 Cocoa 框架，並且可以相容 Objective-C 程式碼。在此基礎之上，Swift 還有許多新特性並且支援程序式（procedural）程式設計和物件導向（object-oriented）程式設計。

Swift 對於初學者來說也很友好。它是第一個既滿足工業標準又像腳本語言（scripting language）一樣充滿表現力和趣味的程式語言。它支援程式碼預覽，這個革命性的特性可以允許程式設計師在不編譯和執行應用程式的前提下執行 Swift 程式碼並即時查看結果。

Swift 將現代程式語言的精華和蘋果工程師文化的智慧結合了起來。編譯器對性能進行了優化，程式語言對開發進行了優化，兩者互不干擾，魚與熊掌兼得。Swift 既可以用於開發「hello, world」這樣的小程式，也可以用於開發一套完整的操作系統。所有的這些特性讓 Swift 對於開發者和蘋果來說都是一項值得的投資。

Swift 是編寫 iOS 和 OS X 應用的極佳手段，並將伴隨著新的特性和功能持續演進。我們對 Swift 充滿信心，你還在等什麼！

> 翻譯：tommy60703 校對：tommy60703, jayhsu, ckvir, rsbrian, petertom51

Swift 初見

本頁內容包括：

通常來說，程式語言教學中的第一個程式應該在螢幕上顯示「Hello, world」。在 Swift 中，可以用一行程式碼實作：

```
println("Hello, world")
```

如果你寫過 C 或者 Objective-C 程式碼，那你應該很熟悉這種形式——在 Swift 中，這行程式碼就是一個完整的程式。你不需要為了輸入輸出或者字串處理導入一個單獨的函式庫。全域作用域中的程式碼會被自動當做程式的入口點，所以你也不需要 `main` 函式。你同樣不需要在每個語句結尾寫上分號。

這個教程會通過一系列程式範例來讓你對 Swift 有初步了解，如果你有什麼不理解的地方也不用擔心——任何本章介紹的內容都會在後面的章節中詳細講解。

> 注意： 為了獲得最好的體驗，在 Xcode 當中使用程式碼預覽功能。程式碼預覽功能可以讓你編輯程式碼並即時看到執行結果。 打開Playground

簡單值

使用 `let` 來宣告常數，使用 `var` 來宣告變數。一個常數的值，在編譯的時候，並不需要有明確的值，但是你只能為它賦值一次。也就是說你可以用常數來表示這樣一個值：你只需要決定一次，但是需要使用很多次。

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

常數或者變數的型別必須和你賦給它們的值一樣。然而，宣告時型別不一定要顯式（explicitly）寫明，宣告的同時賦值的話，編譯器會自動推斷型別。在上面的範例中，編譯器推斷出 `myVariable` 是一個整數（integer）因為它的初始值是整數。

如果初始值沒有提供足夠的資訊（或者沒有初始值），那你需要在變數後面宣告型別，用冒號分割。

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

> 練習： 創建一個常數，顯式指定型別為 `Float` 並指定初始值為 4。

值永遠不會被隱式（implicitly）轉換為其他型別。如果你需要把一個值轉換成其他型別，請顯式轉換。

```
let label = "The width is"
let width = 94
let widthLabel = label + String(width)
```

> 練習： 刪除最後一行中的 `String`，錯誤提示是什麼？

有一種更簡單的把值轉換成字串的方法：把值寫到括號中，並且在括號之前寫一個反斜線。例如：

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

> 練習： 使用 `\()` 來把一個浮點計算轉換成字串，並加上某人的名字，和他打個招呼。

使用方括號 `[]` 來創建陣列（array）和字典（dictionary），並使用索引值（index）或鍵（key）來存取元素（elements）。

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"
```

```
var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

要創建一個空陣列或者字典，使用初始化語法。

```
let emptyArray = [String]()
let emptyDictionary = [String: Float]()
```

如果型別資訊可以被推斷出來，你可以用 `[]` 和 `[:]` 來創建空陣列和空字典——就像你宣告變數或者給函式傳參數的時候一樣。

```
shoppingList = []   // 去逛街並買點東西
occupations = [:]
```

控制流程

使用 `if` 和 `switch` 來進行條件操作，使用 `for-in`、 `for`、 `while` 和 `do-while` 來進行迴圈。包裹條件和迴圈變數括號可以省略，但是語句體的大括號是必須的。

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
teamScore
```

在 `if` 語句中，條件必須是一個布林表達式——這意味著像 `if score { ... }` 這樣的程式碼將報錯，而不會隱性地與 0 做對比。

你可以一起使用 `if` 和 `let` 來處理值缺失（missing）的情況。有些變數的值是可選的（optional）。一個可選的值可能是一個具體的值或者是 `nil`，表示值缺失。在型別後面加一個問號來標記這個變數的值是可選的。

```
var optionalString: String? = "Hello"
optionalString == nil

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

> 練習： 把 `optionalName` 改成 `nil`，greeting 會是什麼？添加一個 `else` 語句，當 `optionalName` 是 `nil` 時給 greeting 賦一個不同的值。

如果變數的可選值（optional value）是 `nil`，條件會判斷為 `false`，大括號中的程式碼會被跳過。如果不是 `nil`，會將值賦給 `let` 後面的常數，這樣程式碼區塊中就可以使用這個值了。

`switch` 支援任意型別的資料以及各種比較操作——不僅僅是整數以及測試相等（tests for equality）。

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let x where x.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(x)?"
default:
    let vegetableComment = "Everything tastes good in soup."
}
```

> 練習： 刪除 `default` 語句，看看會有什麼錯誤？

執行 `switch` 中匹配到的子句之後，程式會退出 `switch` 語句，並不會繼續向下執行，所以不需要在每個子句結尾寫 `break`。

你可以使用 `for-in` 來遍歷（iterate）字典，需要兩個變數來表示每個鍵值對（key-value pair）。

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
largest
```

> 練習： 添加另一個變數來記錄哪種型別的數字是最大的。

使用 `while` 來重複執行一段程式碼直到不滿足條件。迴圈條件可以在開頭也可以在結尾。

```
var n = 2
while n < 100 {
    n = n * 2
}
n

var m = 2
do {
    m = m * 2
} while m < 100
m
```

你可以在迴圈中使用 `...` 來表示範圍，也可以使用傳統的寫法，兩者是等價的：

```
var firstForLoop = 0
for i in 0...3 {
    firstForLoop += i
}
firstForLoop

或是

var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
firstForLoop

var secondForLoop = 0
for var i = 0; i < 3; ++i {
    secondForLoop += 1
}
secondForLoop
```

使用 `..<` 創建的範圍不包含上界，如果想包含的話需要使用 `...`。

函式和閉包

使用 `func` 來宣告一個函式，使用名字和參數來呼叫函式。使用 `->` 來指定函式回傳值。

```
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
greet("Bob", "Tuesday")
```

> 練習： 刪除 `day` 參數，添加一個參數來表示今天吃了什麼午飯。

使用一個元組（tuple）來回傳多個值。

```
func getGasPrices() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}
getGasPrices()
```

函式可以帶有可變個數的參數，這些參數在函式內表現為陣列的形式：

```
func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
```

```
        sum += number
    }
    return sum
}
sumOf()
sumOf(42, 597, 12)
```

> 練習： 寫一個計算參數平均值的函式。

函式可以是巢狀的。巢狀的函式可以存取外側函式的變數，你可以使用巢狀函式來重構一個太長或者太複雜的函式。

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

函式是第一等型別（first-class type），這意味著函式可以作為另一個函式的回傳值。

```
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

函式也可以當做參數傳入另一個函式。

```
func hasAnyMatches(list: [Int], condition: Int -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, lessThanTen)
```

函式實際上是一種特殊的閉包（closure），你可以使用 `{}` 來創建一個匿名閉包。使用 `in` 將參數和回傳值型別宣告與閉包函式體進行分離。

```
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

> 練習： 重寫閉包，對所有奇數回傳 0。

有很多種創建閉包的方法。如果一個閉包的型別已知，比如作為一個回呼函式（callback），你可以忽略參數的型別和回傳

值。單個語句閉包會把它語句的值當做結果回傳。

```
numbers.map({ number in 3 * number })
```

你可以通過參數位置而不是參數名字來參考參數——這個方法在非常短的閉包中非常有用。當一個閉包作為最後一個參數傳給一個函式的時候，它可以直接跟在括號後面。

```
sort([1, 5, 3, 12, 2]) { $0 > $1 }
```

物件和類別

使用 `class` 和類別名稱來創建一個類別。類別中屬性的宣告和常數、變數宣告一樣，唯一的區別就是它們的上下文是類別。同樣，方法（method）和函式宣告也一樣。

```
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

> 練習： 使用 `let` 添加一個常數屬性，再添加一個接收一個參數的方法。

要創建一個類別的實體（instance），在類別名後面加上括號。使用點語法（dot syntax）來存取實體的屬性和方法。

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

這個版本的 `Shape` 類別缺少了一些重要的東西：一個建構函式來初始化類別實體。使用 `init` 來創建一個建構函式。

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

注意 `self` 被用來區別實體變數。當你創建實體的時候，像傳入函式參數一樣給類別傳入建構函式的參數。每個屬性都需要賦值——無論是通過宣告（就像 `numberOfSides`）還是通過建構函式（就像 `name`）。

如果你需要在刪除物件實體之前進行一些清理工作，使用 `deinit` 創建一個解構函式。

子類別的定義方法是在它們的類別名稱後面加上父類別的名稱，用冒號分開。創建類別的時候並不需要一個標準的根類別，所以你可以忽略父類別。

子類別如果要重寫父類別的方法的話，需要用 `override` 標記——如果沒有添加 `override` 就重寫父類別方法的話，編譯器會

報錯。編譯器同樣會檢測 `override` 標記的方法是否確實在父類別中。

```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() ->  Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
```

> 練習：創建 `NamedShape` 的另一個子類別 `Circle`，建構函式接收兩個參數，一個是半徑，一個是名稱，實作 `area` 和 `describe` 方法。

屬性可以有 getter 和 setter 。

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }

    var perimeter: Double {
    get {
        return 3.0 * sideLength
    }
    set {
        sideLength = newValue / 3.0
    }
    }

    override func simpleDescription() -> String {
        return "An equilateral triagle with sides of length \(sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
triangle.perimeter
triangle.perimeter = 9.9
triangle.sideLength
```

在 `perimeter` 的 setter 中，新值的名字是 `newValue`。你可以在 `set` 之後顯式的設置一個名字。

注意 `EquilateralTriangle` 類別的建構函式執行了三步：

1. 設置子類別宣告的屬性值
2. 呼叫父類別的建構函式
3. 改變父類別定義的屬性值。其他的工作比如呼叫方法、getters 和 setters 也可以在這個階段完成。

如果你不需要計算屬性，但是仍然需要在設置一個新值之前或之後執行程式碼，使用 `willSet` 和 `didSet`。

比如，下面的類別確保三角形的邊長總是和正方形的邊長相同。

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
    willSet {
        square.sideLength = newValue.sideLength
    }
    }
    var square: Square {
    willSet {
        triangle.sideLength = newValue.sideLength
    }
    }
    init(size: Double, name: String) {
        square = Square(sideLength: size, name: name)
        triangle = EquilateralTriangle(sideLength: size, name: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
triangleAndSquare.square.sideLength
triangleAndSquare.triangle.sideLength
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
triangleAndSquare.triangle.sideLength
```

類別中的方法和一般的函式有一個重要的區別，函式的參數名只在函式內部使用，但是方法的參數名需要在呼叫的時候會被使用（除了第一個參數）。預設情況下，方法的參數名和它在方法內部的名字一樣，不過你也可以定義第二個名字，這個名字被用在方法內部。

```
class Counter {
    var count: Int = 0
    func incrementBy(amount: Int, numberOfTimes times: Int) {
        count += amount * times
    }
}
var counter = Counter()
counter.incrementBy(2, numberOfTimes: 7)
```

處理變數的可選值時，你可以在操作（比如方法、屬性和下標腳本）之前加 `?`。如果 `?` 之前的值是 `nil`，`?` 後面的東西都會被忽略，並且整個表達式回傳 `nil`。否則，`?` 之後的東西都會被執行。在這兩種情況下，整個表達式的值也是一個可選值。

```
let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional square")
let sideLength = optionalSquare?.sideLength
```

列舉和結構

使用 `enum` 來創建一個列舉。就像類別和其他所有命名型別一樣，列舉可以包含方法。

```
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
    func simpleDescription() -> String {
        switch self {
        case .Ace:
            return "ace"
        case .Jack:
            return "jack"
        case .Queen:
            return "queen"
        case .King:
            return "king"
        default:
```

```
            return String(self.rawValue)
        }
    }
}
let ace = Rank.Ace
let aceRawValue = ace.rawValue
```

> 練習：寫一個函式，透過比較它們的原始值來比較兩個 `Rank` 值。

在上面的範例中，列舉原始值的型別是 `Int`，所以你只需要設置第一個原始值，剩下的原始值會按照順序賦值。你也可以使用字串或者浮點數作為列舉的原始值。使用 `rawValue` 屬性來取得列舉值成員的原始值。

使用 `init?(rawValue:)` 這個初始化函式透過原始值來初始化一個列舉值的實體。

```
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

列舉的成員值是實際值，並不是原始值的另一種表達方法。實際上，如果原始值沒有意義，你不需要設置。

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
        case .Spades:
            return "spades"
        case .Hearts:
            return "hearts"
        case .Diamonds:
            return "diamonds"
        case .Clubs:
            return "clubs"
        }
    }

}
let hearts = Suit.Hearts
let heartsDescription = hearts.simpleDescription()
```

> 練習：給 `Suit` 添加一個 `color` 方法，對 `spades` 和 `clubs` 回傳「black」，對 `hearts` 和 `diamonds` 回傳「red」。

注意，有兩種方式可以參考（refer） `Hearts` 成員：給 `hearts` 常數賦值時，列舉成員 `Suit.Hearts` 需要用全名來參考，因為常數沒有顯式指定型別。在 `switch` 裡，列舉成員使用縮寫 `.Hearts` 來參考，因為 `self` 的值已經知道是一個 `Suit`。已知變數型別的情況下你可以使用縮寫。

使用 `struct` 來創建一個結構。結構和類別有很多相同的地方，比如方法和建構函式。它們之間最大的一個區別就是 結構是傳值，類別是傳參考。

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \
        (suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .Three, suit: .Spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

> 練習：給 `Card` 添加一個方法，創建一副完整的撲克牌並把每張牌的 rank 和 suit 對應起來。

一個列舉成員的實例可以有實例值。相同列舉成員的實例可以有不同的值。創建實例的時候傳入值即可。實例值和原始值是不同的：列舉成員的原始值對於所有實例都是相同的，而且你是在定義列舉的時候設置原始值。

例如，考慮從伺服器獲取日出和日落的時間。伺服器會回傳正常結果或者錯誤資訊。

```
enum ServerResponse {
    case Result(String, String)
    case Error(String)
}

let success = ServerResponse.Result("6:00 am", "8:09 pm")
let failure = ServerResponse.Error("Out of cheese.")

switch success {
case let .Result(sunrise, sunset):
    let serverResponse = "Sunrise is at \(sunrise) and sunset is at \(sunset)."
case let .Error(error):
    let serverResponse = "Failure...  \(error)"
}
```

> 練習： 給 `ServerResponse` 和 `switch` 添加第三種情況。

注意如何從 `ServerResponse` 中提取日出和日落時間。

協定和擴展

使用 `protocol` 來宣告一個協定。

```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

類別、列舉和結構都可以實作協定。

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

> 練習： 寫一個實作這個協定的列舉。

注意宣告 `SimpleStructure` 時候 `mutating` 關鍵字用來標記一個會修改結構的方法。 `SimpleClass` 的宣告不需要標記任何方法因為類別中的方法經常會修改類別。

使用 `extension` 來為現有的型別添加功能，比如新的方法和參數。你可以使用擴展來改造定義在別處，甚至是從外部函式庫或者框架引入的一個型別，使得這個型別遵循某個協定。

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
    return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
7.simpleDescription
```

> 練習： 給 `Double` 型別寫一個擴展，添加 `absoluteValue` 功能。

你可以像使用其他命名型別一樣使用協定名稱——例如，創建一個有不同型別但是都實作一個協定的物件集合。當你處理型別是協定的值時，協定外定義的方法不可用。

```
let protocolValue: ExampleProtocol = a
protocolValue.simpleDescription
// protocolValue.anotherProperty  // Uncomment to see the error
```

即使 `protocolValue` 變數執行時的型別是 `simpleClass`，編譯器會把它的型別當做 `ExampleProtocol`。這表示你不能呼叫類別在它實作的協定之外實作的方法或者屬性。

泛型

在角括號裡寫一個名字來創建一個泛型函式或者型別。

```
func repeat<ItemType>(item: ItemType, times: Int) -> ItemType[] {
    var result = ItemType[]()
    for i in 0..times {
        result += item
    }
    return result
}
repeat("knock", 4)
```

你也可以創建泛型類別、列舉和結構。

```
// Reimplement the Swift standard library's optional type
enum OptionalValue<T> {
    case None
    case Some(T)
}
var possibleInteger: OptionalValue<Int> = .None
possibleInteger = .Some(100)
```

在型別名後面使用 `where` 來指定對型別的需求，比如，限定型別實作某一個協定，限定兩個型別是相同的，或者限定某個類別必須有一個特定的父類別

```
func anyCommonElements <T, U where T: Sequence, U: Sequence, T.GeneratorType.Element: Equatable, T.GeneratorType.Elemer
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
```

```
            }
        }
        return false
    }
    anyCommonElements([1, 2, 3], [3])
```

> 練習： 修改 `anyCommonElements` 函式來創建一個函式，回傳一個陣列，內容是兩個序列的共有元素。

簡單起見，你可以忽略 `where`，只在冒號後面寫協定或者類別名稱。 `<T: Equatable>` 和 `<T where T: Equatable>` 是等價的。

Swift 教程

本章介紹了 Swift 的各種特性及其使用方法，是全書的核心部分。

> 翻譯：tommy60703

基礎部分

本頁包含內容：

Swift 是開發 iOS 和 OS X 應用程式的一門新語言。然而，如果你有 C 或者 Objective-C 開發經驗的話，你會發現 Swift 的很多內容都是你熟悉的。

Swift 的型別是在 C 和 Objective-C 的基礎上提出的，`Int` 是整數；`Double` 和 `Float` 是浮點數；`Bool` 是布林值；`String` 是字串。Swift 還有兩個有用的集合型別，`Array` 和 `Dictionary`，請參考集合型別。

就像 C 語言一樣，Swift 使用變數來進行儲存並透過變數名稱來參考值。在 Swift 中，值不可變的變數有著廣泛的應用，它們就是常數，而且比 C 語言的常數更強大。在 Swift 中，如果你要處理的值不需要改變，那使用常數可以讓你的程式碼更加安全並且更好地表達你的意圖。

除了我們熟悉的型別，Swift 還增加了 Objective-C 中沒有的型別比如 tuple。Tuple 可以讓你創建或者傳遞一組資料，比如作為函式的回傳值時，你可以用一個 tuple 回傳多個值。

Swift 還增加了 optional 型別，用於處理值不存在的情況。optional 表示 「那兒有一個值，並且它等於 x 」 或者 「那兒沒有值」。optional 有點像在 Objective-C 中使用 `nil`，但是它可以用在任何型別上，不僅僅是類別。optional 型別比 Objective-C 中的 `nil` 指標更加安全也更具表現力，它是 Swift 許多強大特性的重要組成部分。

Swift 是一個型別安全的語言，optional 就是一個很好的範例。Swift 可以讓你清楚地知道值的型別。如果你的程式碼期望得到一個 `String`，型別安全會阻止你不小心傳入一個 `Int`。你可以在開發階段盡早發現並修正錯誤。

常數和變數

常數和變數把一個名字（比如 `maximumNumberOfLoginAttempts` 或者 `welcomeMessage`）和一個指定型別的值（比如數字 `10` 或者字串 `"Hello"`）關聯起來。常數的值一旦設定就不能改變，而變數的值可以隨意更改。

宣告常數和變數

常數和變數必須在使用前宣告，用 `let` 來宣告常數，用 `var` 來宣告變數。下面的範例展示了如何用常數和變數來記錄使用者嘗試登錄的次數：

```
let maximumNumberOfLoginAttempts = 10
```

```
var currentLoginAttempt = 0
```

這兩行程式碼可以被理解為：

「宣告一個新常數叫 `maximumNumberOfLoginAttempts`，並給它一個值 `10`。然後，宣告一個變數是 `currentLoginAttempt` 並將它的值初始化為 `0`。」

在這個範例中，允許的最大嘗試登錄次數被宣告為一個常數，因為這個值不會改變。目前嘗試登錄次數被宣告為一個變數，因為每次嘗試登錄失敗的時候都需要增加這個值。

你可以在一行中宣告多個常數或者多個變數，用逗號隔開：

```
var x = 0.0, y = 0.0, z = 0.0
```

> 注意：
> 如果你的程式碼中有不需要改變的值，請使用 `let` 關鍵字將它宣告為常數。只將需要改變的值宣告為變數。

型別標注

當你宣告常數或者變數的時候可以加上型別標注（*type annotation*），說明常數或者變數中要儲存的值的型別。如果要添加型別標注，需要在常數或者變數名後面加上一個冒號和空格，然後加上型別名稱。

這個範例給 `welcomeMessage` 變數添加了型別標注，表示這個變數可以儲存 `String` 型別的值：

```
var welcomeMessage: String
```

宣告中的冒號代表著「是...型別」，所以這行程式碼可以被理解為：

「宣告一個型別為 `String`，名字為 `welcomeMessage` 的變數。」

「型別為 `String`」的意思是「可以儲存任意 `String` 型別的值。」 可以把他想成是「這種型別的東西」或是「這種類型的東西」可以被儲存。

`welcomeMessage` 變數現在可以被設置成任意字串：

```
welcomeMessage = "Hello"
```

你可以在同一行定義定義多個相同行別的變數，並用逗號隔開，最後加上型別標注。

```
var red, green, blue: Double
```

> 注意：
> 一般來說你很少需要寫型別標注。如果你在宣告常數或者變數的時候指派了一個初始值，Swift可以推斷出這個常數或者變數的型別，請參考型別安全和型別推斷。在上面的範例中，沒有給 `welcomeMessage` 指派初始值，所以變數 `welcomeMessage` 的型別是透過一個型別標注指定的，而不是透過初始值推斷的。

常數和變數的命名

你可以用任何你喜歡的字元作為常數和變數名，包括 Unicode 字元：

```
let π = 3.14159
let 你好 = "你好世界"
let 🐮 = "dogcow"
```

常數與變數名不能包含數學符號、箭頭、保留的（或者非法的）Unicode 碼位、連線與制表字元（box-drawing characters），也不能以數字開頭，但是可以在常數與變數名的其他地方包含數字。

一旦你將常數或者變數宣告為確定的型別，你就不能使用相同的名字再次進行宣告，或者改變其儲存的值的型別。同時，你也不能將常數與變數進行互轉。

> 注意：
> 如果你需要使用與 Swift 保留關鍵字相同的名稱作為常數或者變數名，你可以使用反引號（`）將關鍵字包圍的方式將其作為名字使用。無論如何，你應當避免使用關鍵字作為常數或變數名，除非你別無選擇。

你可以更改現有的變數值為其他同型別的值，在下面的範例中，`friendlyWelcome` 的值從 `"Hello!"` 改為了 `"Bonjour!"`：

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome 現在是 "Bonjour!"
```

與變數不同，常數的值一旦被確定就不能更改了。嘗試這樣做會導致編譯時報錯：

```
let languageName = "Swift"
languageName = "Swift++"
// 這會報編譯時錯誤 - languageName 不能被改變
```

輸出常數和變數

你可以用 `println` 函式來輸出目前常數或變數的值：

```
println(friendlyWelcome)
// 輸出 "Bonjour!"
```

`println` 是一個用來輸出的全域函式，輸出的內容會在最後換行。如果你用 Xcode，`println` 將會輸出內容到「console」面板上。(另一種函式叫 `print`，唯一區別是在輸出內容最後不會換行。)

`println` 函式輸出傳入的 `String` 值：

```
println("This is a string")
// 輸出 "This is a string"
```

與 Cocoa 裡的 `NSLog` 函式類似的是，`println` 函式可以輸出更複雜的訊息。這些訊息可以包含目前常數和變數的值。

Swift 用字串插值（*string interpolation*）的方式把常數名或者變數名當做占位符（placeholder）加入到長字串中，Swift 會用目前常數或變數的值替換這些占位符。將常數或變數名放入圓括號中，並在開括號前使用反斜線將其跳脫：

```
println("The current value of friendlyWelcome is \(friendlyWelcome)")
// 輸出 "The current value of friendlyWelcome is Bonjour!
```

> 注意：

> 字串插值所有可用的選項，請參考字串插值。

註解

請將你的程式碼中的非執行文字註解成提示或者筆記以方便你將來閱讀。Swift 的編譯器將會在編譯程式碼時自動忽略掉註解部分。

Swift 中的註解與 C 語言的註解非常相似。單行註解以雙正斜線（`//`）作為起始標記:

```
// 這是一個註解
```

你也可以進行多行註解，其起始標記為單個正斜線後跟隨一個星號（`/*`），終止標記為一個星號後跟隨單個正斜線（`*/`）：

```
/* 這是一個,
多行註解 */
```

與 C 語言多行註解不同，Swift 的多行註解可以巢狀地寫在其它的多行註解之中。你可以先寫一個多行註解區塊，然後在這個註解區塊之中再套入第二個多行註解。終止註解時先插入第二個註解區塊的終止標記，然後再插入第一個註解區塊的終止標記：

```
/* 這是第一個多行註解的開頭
/* 這是第二個被套入的多行註解 */
這是第一個多行註解的結尾 */
```

透過運用巢狀多行註解，你可以快速方便的註解掉一大段程式碼，即使這段程式碼之中已經含有了多行註解區塊。

分號

與其他大部分程式語言不同，Swift 並不強制要求你在每條語句的結尾處使用分號（`;`），當然，你也可以按照你自己的習慣添加分號。有一種情況下必須要用分號，即你打算在同一行內寫多條獨立的語句：

```
let cat = "🐱"; println(cat)
// 輸出 "🐱"
```

整數

整數就是沒有小數部分的數字，比如 `42` 和 `-23`。整數可以是 `有號`（正、負、零）或者 `無號`（正、零）。

Swift 提供了 8、16、32 和 64 位元的有號和無號整數型別。這些整數型別和 C 語言的命名方式很像，比如 8 位元無號整數型別是 `UInt8`，32 位元有號整數型別是 `Int32`。就像 Swift 的其他型別一樣，整數型別采用大寫命名法。

整數範圍

你可以存取不同整數型別的 `min` 和 `max` 屬性來獲取對應型別的最大值和最小值：

```
let minValue = UInt8.min  // minValue 為 0, 是 UInt8 型別的最小值
let maxValue = UInt8.max  // maxValue 為 255, 是 UInt8 型別的最大值
```

Int

一般來說，你不需要特別指定整數的長度。Swift 提供了一個特殊的整數型別 `Int`，長度與目前平台的原生字長相同：

- 在 32 位元平台上，`Int` 和 `Int32` 長度相同。
- 在 64 位元平台上，`Int` 和 `Int64` 長度相同。

除非你需要特定長度的整數，一般來說使用 `Int` 就夠了。這可以提高程式碼一致性和可複用性。即使是在 32 位元平台上，`Int` 可以儲存的整數範圍也可以達到 `-2147483648` ~ `2147483647`，大多數時候這已經足夠大了。

UInt

Swift 也提供了一個特殊的無號型別 `UInt`，長度與目前平台的原生字長相同：

- 在 32 位元平台上，`UInt` 和 `UInt32` 長度相同。
- 在 64 位元平台上，`UInt` 和 `UInt64` 長度相同。

> 注意：
>
> 盡量不要使用 `UInt`，除非你真的需要儲存一個和目前平台原生字長相同的無號整數。除了這種情況，最好使用 `Int`，即使你要儲存的值已知是非負的。統一使用 `Int` 可以提高程式碼的可複用性，避免不同型別數字之間的轉換，並且匹配數字的型別推斷，請參考型別安全和型別推斷。

浮點數

浮點數是有小數部分的數字，比如 `3.14159`，`0.1` 和 `-273.15`。

浮點型別比整數型別表示的範圍更大，可以儲存比 `Int` 型別更大或者更小的數字。Swift 提供了兩種有號浮點數型別：

- `Double` 表示 64 位元浮點數。當你需要儲存很大或者很高精度的浮點數時請使用此型別。
- `Float` 表示 32 位元浮點數。精度要求不高的話可以使用此型別。

> 注意：
> `Double` 精確度很高，至少有 15 位數字，而 `Float` 最少只有 6 位數字。選擇哪個型別取決於你的程式碼需要處理的值的範圍。

型別安全和型別推斷

Swift 是一個型別安全（*type safe*）的語言。型別安全的語言可以讓你清楚地知道程式碼要處理的值的型別。如果你的程式碼需要一個 `String`，你絕對不可能不小心傳進去一個 `Int`。

由於 Swift 是型別安全的，所以它會在編譯你的程式碼時進行型別檢查（*type checks*），並把不匹配的型別標記為錯誤。這可以讓你在開發的時候盡早發現並修復錯誤。

當你要處理不同型別的值時，型別檢查可以幫你避免錯誤。然而，這並不是說你每次宣告常數和變數的時候都需要顯式指定型別。如果你沒有顯式指定型別，Swift 會使用型別推斷（*type inference*）來選擇合適的型別。有了型別推斷，編譯器可以在編譯程式碼的時候自動推斷出表達式的型別。原理很簡單，只要檢查你指派的值即可。

因為有型別推斷，和 C 或者 Objective-C 比起來 Swift 很少需要宣告型別。常數和變數雖然需要明確型別，但是大部分工作並不需要你自己來完成。

當你宣告常數或者變數並賦初值的時候型別推斷非常有用。當你在宣告常數或者變數的時候賦給它們一個字面量（*literal value* 或 *literal*）即可觸發型別推斷。（字面量就是會直接出現在你程式碼中的值，比如 `42` 和 `3.14159`。）

例如，如果你給一個新常數指派值為 `42` 並且沒有標明型別，Swift 可以推斷出常數型別是 `Int`，因為你給它指派的初始值看起來像一個整數：

```
let meaningOfLife = 42
// meaningOfLife 會被推測為 Int 型別
```

同理，如果你沒有給浮點字面量標明型別，Swift 會推斷你想要的是 `Double`：

```
let pi = 3.14159
// pi 會被推測為 Double 型別
```

當推斷浮點數的型別時，Swift 總是會選擇 `Double` 而不是 `Float`。

如果表達式中同時出現了整數和浮點數，會被推斷為 `Double` 型別：

```
let anotherPi = 3 + 0.14159
// anotherPi 會被推測為 Double 型別
```

原始值 `3` 沒有顯式宣告型別，而表達式中出現了一個浮點字面量，所以表達式會被推斷為 `Double` 型別。

數值型字面量

整數字面量可以被寫作：

- 一個十進制數，沒有前綴
- 一個二進制數，前綴是 `0b`
- 一個八進制數，前綴是 `0o`
- 一個十六進制數，前綴是 `0x`

下面的所有整數字面量的十進制值都是 `17`：

```
let decimalInteger = 17
let binaryInteger = 0b10001       // 二進制的 17
let octalInteger = 0o21           // 八進制的 17
let hexadecimalInteger = 0x11     // 十六進制的 17
```

浮點字面量可以是十進制（沒有前綴）或者是十六進制（前綴是 `0x`）。小數點兩邊必須有至少一個十進制數字（或者是十六進制的數字）。浮點字面量還有一個 optional 的指數（*exponent*），在十進制浮點數中透過大寫或者小寫的 `e` 來指定，在十六進制浮點數中透過大寫或者小寫的 `p` 來指定。

如果一個十進制數的指數為 `exp`，那這個數相當於基數和 10^exp 的乘積：

- `1.25e2` 表示 1.25 × 10^2，等於 `125.0`。
- `1.25e-2` 表示 1.25 × 10^-2，等於 `0.0125`。

如果一個十六進制數的指數為 `exp`，那這個數相當於基數和 2^exp 的乘積：

- `0xFp2` 表示 15 × 2^2，等於 `60.0`。
- `0xFp-2` 表示 15 × 2^-2，等於 `3.75`。

下面的這些浮點字面量都等於十進制的 `12.1875`：

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

數值型字面量可以包括額外的格式來增強可讀性。整數和浮點數都可以添加額外的零並且包含底線，並不會影響字面量：

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

數值型型別轉換

通常來講，即使程式碼中的整數常數和變數已知非負，也請使用 `Int` 型別。總是使用預設的整數型別可以保證你的整數常數和變數可以直接被複用並且可以匹配整數類別字面量的型別推斷。 只有在必要的時候才使用其他整數型別，比如要處理外部的長度明確的資料或者為了優化性能、記憶體占用等等。使用顯式指定長度的型別可以及時發現溢位並且可以暗示正在處理特殊資料。

整數轉換

不同整數型別的變數和常數可以儲存不同範圍的數字。 `Int8` 型別的常數或者變數可以儲存的數字範圍是 `-128` ~ `127`，而 `UInt8` 型別的常數或者變數能儲存的數字範圍是 `0` ~ `255` 。如果數字超出了常數或者變數可儲存的範圍，編譯的時候會報錯：

```
let cannotBeNegative: UInt8 = -1
// UInt8 型別不能儲存負數，所以會報錯
let tooBig: Int8 = Int8.max + 1
// Int8 型別不能儲存超過最大值的數，所以會報錯
```

由於每種整數型別都可以儲存不同範圍的值，所以你必須根據不同情況選擇性使用數值型型別轉換。這種選擇性使用的方式，可以預防隱式轉換的錯誤並讓你的程式碼中的型別轉換意圖變得清晰。

要將一種數字型別轉換成另一種，你要用目前值來初始化一個期望型別的新數字，這個數字的型別就是你的目標型別。在下面的範例中，常數 `twoThousand` 是 `UInt16` 型別，然而常數 `one` 是 `UInt8` 型別。它們不能直接相加，因為它們型別不同。所以要呼叫 `UInt16(one)` 來創建一個新的 `UInt16` 數字並用 `one` 的值來初始化，然後使用這個新數字來計算：

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

現在兩個數字的型別都是 `UInt16` ，可以進行相加。目標常數 `twoThousandAndOne` 的型別被推斷為 `UInt16` ，因為它是兩個 `UInt16` 值的和。

`SomeType(ofInitialValue)` 是呼叫 Swift 初始化函式並傳入一個初始值的預設方法。在語言內部， `UInt16` 有一個初始化函式，可以接受一個 `UInt8` 型別的值，所以這個初始化函式可以用現有的 `UInt8` 來創建一個新的 `UInt16` 。注意，你並不能傳入任意型別的值，只能傳入 `UInt16` 內部有對應初始化函式的值。不過你可以擴展現有的型別來讓它可以接收其他型別的值（包括自定義型別），請參考擴展。

整數和浮點數轉換

整數和浮點數的轉換必須顯式指定型別：

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi 等於 3.14159，所以被推測為 Double 型別
```

這個範例中，常數 three 的值被用來創建一個 Double 型別的值，所以加號兩邊的數型別須相同。如果不進行轉換，兩者無法相加。

浮點數到整數的同樣可以反向轉換，整數型別可以用 Double 或者 Float 型別來初始化：

```
let integerPi = Int(pi)
// integerPi 等於 3，所以被推測為 Int 型別
```

當用這種方式來初始化一個新的整數值時，浮點值會被截斷。也就是說 4.75 會變成 4，-3.9 會變成 -3。

> 注意：
> 結合數值型常數和變數不同於結合數值型字面量。字面量 3 可以直接和字面量 0.14159 相加，因為數值字面量本身沒有明確的型別。它們的型別只在編譯器需要求值的時候被推測。

型別別名

型別別名（*type aliases*）就是給現有型別定義另一個名字。你可以使用 typealias 關鍵字來定義型別別名。

當你想要給現有型別起一個更有意義的名字時，型別別名非常有用。假設你正在處理特定長度的外部資源的資料：

```
typealias AudioSample = UInt16
```

定義了一個型別別名之後，你可以在任何使用原始名的地方使用別名：

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound 現在是 0
```

本例中，AudioSample 被定義為 UInt16 的一個別名。因為它是別名，AudioSample.min 實際上是 UInt16.min，所以會給 maxAmplitudeFound 賦一個初值 0。

布林值

Swift 有一個基本的布林（*Boolean*）型別，叫做 Bool。布林值指邏輯上的（*logical*），因為它們只能是真或者假。Swift 有兩個布林常數，true 和 false：

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

orangesAreOrange 和 turnipsAreDelicious 的型別會被推斷為 Bool，因為它們的初值是布林字面量。就像之前提到的 Int 和 Double 一樣，如果你創建變數的時候給它們指派 true 或者 false，那你不需要將常數或者變數宣告為 Bool 型別。初始化常數或者變數的時候如果所指派的值的型別已知，就可以觸發型別推斷，這讓 Swift 程式碼更加簡潔並且可讀性更高。

當你使用條件語句比如 if 語句的時候，布林值非常有用：

```
if turnipsAreDelicious {
    println("Mmm, tasty turnips!")
} else {
    println("Eww, turnips are horrible.")
}
// 輸出 "Eww, turnips are horrible."
```

條件語句，例如 `if`，請參考控制流程。

如果你在需要使用 `Bool` 型別的地方使用了非布林值，Swift 的型別安全機制會報錯。下面的範例會報告一個編譯時錯誤：

```
let i = 1
if i {
    // 這個範例不會透過編譯，會報錯
}
```

然而，下面的範例是合法的：

```
let i = 1
if i == 1 {
    // 這個範例會編譯成功
}
```

`i == 1` 的比較結果是 `Bool` 型別，所以第二個範例可以透過型別檢查。類似 `i == 1` 這樣的比較，請參考基本運算子。

和 Swift 中的其他型別安全的範例一樣，這個方法可以避免錯誤並保證這塊程式碼的意圖總是清晰的。

Tuples

*Tuples*把多個值組合成一個複合值。tuple 內的值可以使任意型別，並不要求是相同型別。

下面這個範例中，`(404, "Not Found")` 是一個描述 *HTTP* 狀態碼（*HTTP status code*）的 tuple。HTTP 狀態碼是當你請求網頁的時候 web 伺服器回傳的一個特殊值。如果你請求的網頁不存在就會回傳一個 `404 Not Found` 狀態碼。

```
let http404Error = (404, "Not Found")
// http404Error 的型別是 (Int, String)，值是 (404, "Not Found")
```

`(404, "Not Found")` tuple 把一個 `Int` 值和一個 `String` 值組合起來表示 HTTP 狀態碼的兩個部分：一個數字和一個人類別可讀的描述。這個 tuple 可以被描述為「一個型別為 `(Int, String)` 的 tuple」。

你可以把任意順序的型別組合成一個 tuple，這個 tuple 可以包含所有型別。只要你想，你可以創建一個型別為 `(Int, Int, Int)` 或者 `(String, Bool)` 或者其他任何你想要的組合的 tuple。

你可以將一個 tuple 的內容分解（*decompose*）成單獨的常數和變數，然後你就可以正常使用它們了：

```
let (statusCode, statusMessage) = http404Error
println("The status code is \(statusCode)")
// 輸出 "The status code is 404"
println("The status message is \(statusMessage)")
// 輸出 "The status message is Not Found"
```

如果你只需要一部分 tuple 值，分解的時候可以把要忽略的部分用底線（`_`）標記：

```
let (justTheStatusCode, _) = http404Error
println("The status code is \(justTheStatusCode)")
// 輸出 "The status code is 404"
```

此外，你還可以透過索引值來存取 tuple 中的單個元素，索引值從零開始：

```
println("The status code is \(http404Error.0)")
// 輸出 "The status code is 404"
println("The status message is \(http404Error.1)")
// 輸出 "The status message is Not Found"
```

你可以在定義 tuple 的時候給單個元素命名：

```
let http200Status = (statusCode: 200, description: "OK")
```

給 tuple 中的元素命名後，你可以透過名字來獲取這些元素的值：

```
println("The status code is \(http200Status.statusCode)")
// 輸出 "The status code is 200"
println("The status message is \(http200Status.description)")
// 輸出 "The status message is OK"
```

作為函式回傳值時，tuple 非常有用。一個用來獲取網頁的函式可能會回傳一個 `(Int, String)` tuple 來描述是否獲取成功。和只能回傳一個型別的值比較起來，一個包含兩個不同型別的值的 tuple 可以讓函式的回傳訊息更有用。請參考函式參數與回傳值。

> 注意：
> tuple 在臨時組織值的時候很有用，但是並不適合創建複雜的資料結構。如果你的資料結構並不是臨時使用，請使用類別別或者結構而不是 tuple。請參考類別別和結構。

Optionals

使用*optionals*來處理值可能不存在的情況。optionals 型別表示：

- 有值，等於 x

或者

- 沒有值

> 注意：
>
> C 和 Objective-C 中並沒有 optionals 型別這個概念。最接近的是 Objective-C 中的一個特性，一個方法要不回傳一個物件，要不回傳 `nil`，`nil` 表示「缺少一個合法的物件」。然而，這只對物件起作用——對於結構，基本的 C 型別或者列舉型別不起作用。對於這些型別，Objective-C 方法一般會回傳一個特殊值（比如 `NSNotFound`）來暗示值不存在。這種方法假設方法的呼叫者知道並記得對特殊值進行判斷。然而，Swift 的 optionals 可以讓你暗示任意型別的值不存在，並不需要一個特殊值。

來看一個範例。Swift 的 `String` 型別有一個叫做 `toInt` 的方法，作用是將一個 `String` 值轉換成一個 `Int` 值。然而，並不是所有的字串都可以轉換成一個整數。字串 `"123"` 可以被轉換成數字 `123`，但是字串 `"hello, world"` 不行。

下面的範例使用 `toInt` 方法來嘗試將一個 `String` 轉換成 `Int`：

```
let possibleNumber = "123"
let convertedNumber = possibleNumber.toInt()
// convertedNumber 被推測為型別 "Int?", 也就是 "optional Int"
```

因為 `toInt` 方法可能會失敗，所以它回傳一個*optional* `Int`，而不是一個 `Int`。一個 optional `Int` 被寫作 `Int?` 而不是 `Int`。問號暗示包含的值是 optional 型別，也就是說可能包含 `Int` 值也可能不包含值。（不能包含其他任何值比如 `Bool` 值或者 `String` 值。只能是 `Int` 或者什麼都沒有。）

if 語句以及強制解析

你可以使用 `if` 語句透過對比nil的方式來判斷一個 optional 是否包含值。使用「等於」運算子（`==`）或是「不等於」運算子(`!=`)來執行這樣的比較。

如果一個 optional 有值，就會被認為是「不等於」`nil`。

```
if convertedNumber != nil {
    println("convertedNumber contains some integer value.")
}
// 輸出 "convertedNumber contains some integer value."
```

當你確定 optional 確實包含值之後，你可以在 optional 的名字後面加一個感嘆號（`!`）來獲取值。這個驚嘆號表示「我知道這個 optional 有值，請使用它。」這被稱為 optional 值的強制解析（*forced unwrapping*）：

```
if convertedNumber != nil {
    println("convertedNumber has an integer value of \(convertedNumber!).")
}
// 輸出 "123 has an integer value of 123"
```

更多關於 `if` 語句的內容，請參考控制流程程。

> 注意：
> 使用 `!` 來獲取一個不存在的 optional 值會導致執行時錯誤。使用 `!` 來強制解析值之前，一定要確定 optional 包含一個非 `nil` 的值。

Optional 綁定

使用*optional* 綁定（*optional binding*）來判斷 optional 是否包含值，如果包含就把值指派給一個臨時常數或者變數。Optional 綁定可以用在 `if` 和 `while` 語句中來對 optional 的值進行判斷並把值指派給一個常數或者變數。`if` 和 `while` 語句，請參考控制流程程。

像下面這樣在 `if` 語句中寫一個 optional 綁定：

```
if let constantName = someOptional {
    statements
}
```

你可以像上面這樣使用 optional 綁定來重寫 `possibleNumber` 這個範例：

```
if let actualNumber = possibleNumber.toInt() {
    println("\(possibleNumber) has an integer value of \(actualNumber)")
} else {
    println("\(possibleNumber) could not be converted to an integer")
}
```

```
// 輸出 "123 has an integer value of 123"
```

這段程式碼可以被理解為：

「如果 `possibleNumber.toInt` 回傳的 optional `Int` 包含一個值，創建一個叫做 `actualNumber` 的新常數並將optional 包含的值指派給它。」

如果轉換成功， `actualNumber` 常數可以在 `if` 語句的第一個分支中使用。它已經被 optional 包含的值初始化過，所以不需要再使用 `!` 後綴來獲取它的值。在這個範例中， `actualNumber` 只被用來輸出轉換結果。

你可以在 optional 綁定中使用常數和變數。如果你想在 `if` 語句的第一個分支中操作 `actualNumber` 的值，你可以改成 `if var actualNumber`，這樣 optional 包含的值就會被指派給一個變數而不是常數。

nil

你可以給 optional 變數指派為 `nil` 來表示它沒有值：

```
var serverResponseCode: Int? = 404
// serverResponseCode 包含一個optionals的 Int 值 404
serverResponseCode = nil
// serverResponseCode 現在不包含值
```

> 注意：
> `nil` 不能用於非 optional 的常數和變數。如果你的程式碼中有常數或者變數需要處理值不存在的情況，請把它們宣告成對應的 optional 型別。

如果你宣告一個 optional 常數或者變數但是沒有指派，它們會自動被設置為 `nil`：

```
var surveyAnswer: String?
// surveyAnswer 被自動設置為 nil
```

> 注意：
>
> Swift 的 `nil` 和 Objective-C 中的 `nil` 並不一樣。在 Objective-C 中， `nil` 是一個指向不存在物件的指標。在 Swift 中， `nil` 不是指標——它是一個確定的值，用來表示值不存在。任何型別的 optional 狀態都可以被設置為 `nil`，不只是物件型別。

隱式解析 Optionals

如上所述，optionals 暗示了常數或者變數可以「沒有值」。optionals 可以透過 `if` 語句來判斷是否有值，如果有值的話可以透過 optional 綁定來解析值。

有時候在程式架構中，第一次被指派之後，可以確定一個 optional 總會有值。在這種情況下，每次都要判斷和解析 optional 值是非常低效的，因為可以確定它總會有值。

這種型別的 optionals 狀態被定義為隱式解析 *optionals*（*implicitly unwrapped optionals*）。把想要用作 optionals 的型別的後面的問號（ `String?` ）改成感嘆號（ `String!` ）來宣告一個隱式解析 optionals。

當 optionals 被第一次指派之後就可以確定之後一直有值的時候，隱式解析 optionals 非常有用。隱式解析 optionals 主要被用在 Swift 中類別別的初始化中，請參考類別別實例之間的迴圈強參考。

一個隱式解析 optionals 其實就是一個普通的 optionals，但是可以被當做非 optionals 來使用，並不需要每次都使用解析來獲取 optionals 值。下面的範例展示了 optionals `String` 和隱式解析 optional `String` 之間的區別：

```
let possibleString: String? = "An optional string."
println(possibleString!) // 需要驚嘆號來獲取值
// 輸出 "An optional string."
```

```
let assumedString: String! = "An implicitly unwrapped optional string."
println(assumedString)  // 不需要感嘆號
// 輸出 "An implicitly unwrapped optional string."
```

你可以把隱式解析 optional 當做一個可以自動解析的 optional。你要做的只是宣告的時候把感嘆號放到型別的結尾，而不是每次取值的 optional 名字的結尾。

> 注意：
>
> 如果你在隱式解析 optional 沒有值的時候嘗試取值，會觸發執行時錯誤。和你在沒有值的普通 optional 後面加一個驚嘆號一樣。

你仍然可以把隱式解析 optional 當做普通 optional 來判斷它是否包含值：

```
if assumedString != nil {
    println(assumedString)
}
// 輸出 "An implicitly unwrapped optional string."
```

你也可以在 optional 綁定中使用隱式解析 optional 來檢查並解析它的值：

```
if let definiteString = assumedString {
    println(definiteString)
}
// 輸出 "An implicitly unwrapped optional string."
```

> 注意：
>
> 如果一個變數之後可能變成 `nil` 的話請不要使用隱式解析 optionals。如果你需要在變數的生命周期中判斷是否是 `nil` 的話，請使用普通的 optionals。

Assertions

Optionals 可以讓你判斷值是否存在，你可以在程式碼中優雅地處理值不存在的情況。然而，在某些情況下，如果值不存在或者值並不滿足特定的條件，你的程式碼可能並不需要繼續執行。這時，你可以在你的程式碼中觸發一個*assertion*來結束程式碼的執行並透過除錯來找到值不存在的原因。

使用 Assertions 進行除錯

一個 assertion 會在執行時判斷一個邏輯條件是否為 `true`。從字面意思來說，一個 assertion「斷言」一個條件是否為真。你可以使用 assertion 來保證在執行其他程式碼之前，某些重要的條件已經被滿足。如果條件判斷為 `true`，程式碼執行會繼續進行；如果條件判斷為 `false`，程式碼執行停止，你的應用程式被終止。

如果你的程式碼在除錯環境下觸發了一個 assertion，比如你在 Xcode 中構建並執行一個應用程式，你可以清楚地看到不合法的狀態發生在哪裡並檢查 assertion 被觸發時你的應用程式的狀態。此外，assertion 允許你附加一條除錯訊息。

你可以使用全域 `assert` 函式來寫一個 assertion。向 `assert` 函式傳入一個結果為 `true` 或者 `false` 的表達式以及一條訊息，當表達式為 `false` 的時候這條訊息會被顯示：

```
let age = -3
assert(age >= 0, "A person's age cannot be less than zero")
// 因為 age < 0，所以assertion會觸發
```

在這個範例中，只有 `age >= 0` 為 `true` 的時候程式碼執行才會繼續，也就是說，當 `age` 的值非負的時候。如果 `age` 的值是負數，就像程式碼中那樣，`age >= 0` 為 `false`，assertion 被觸發，結束應用程式。

assertion 訊息不能使用字串插值。assertion 訊息可以省略，就像這樣：

```
assert(age >= 0)
```

何時使用 **Assertions**

當條件可能為 `false` 時使用 assertion，但是最終一定要保證條件為 `true`，這樣你的程式碼才能繼續執行。assertion 的適用情境：

- 整數型別的 subscript 索引值被傳到一個自定義 subscript 實作，但是 subscript 索引值可能太小或者太大。
- 需要給函式傳入一個值，但是非法的值可能導致函式不能正常執行。
- 一個 optional 值現在是 `nil`，但是後面的程式碼執行需要一個非 `nil` 值。

請參考Subscripts和函式。

> 注意：
> Assertion 可能導致你的應用程式終止執行，所以你應當仔細設計你的程式碼來讓非法條件不會出現。然而，在你的應用程式發佈之前，有時候非法條件可能出現，這時使用 assertion 可以快速發現問題。

> 翻譯：tommy60703

基本運算子

本頁包含內容：

運算子是檢查、改變、合並值的特殊符號或短語。例如，加號 `+` 將兩個數相加（如 `let i = 1 + 2`）。複雜些的運算例如邏輯 AND 運算子 `&&`（如 `if enteredDoorCode && passedRetinaScan`），又或直接讓 i 值加 1 的累加運算子 `++i` 等。

Swift 支援大部分標準 C 語言的運算子，且改進許多特性來減少常見的編碼錯誤。如，指派運算子（`=`）不回傳值，以防止把想要判斷相等運算子（`==`）的地方寫成指派運算子導致的錯誤。數值運算子（`+`，`-`，`*`，`/`，`%` 等）會檢測並不允許溢位，以此來避免保存變數時由於變數大於或小於其型別所能儲存的範圍時導致的異常結果。當然允許你使用 Swift 的溢位運算子來實作溢位。詳情參見溢位運算子。

有別於 C 語言，在 Swift 中你可以對浮點數進行餘數運算（`%`），Swift 還提供了 C 語言所沒有的，用來表達兩數之間的值的區間運算子，（`a..<b` 和 `a...b`），這方便我們表達一個區間內的數值。

本章節只描述了 Swift 中的基本運算子，進階運算子包含了進階運算子、如何自定義運算子，以及如何進行自定義型別的運算子多載。

術語

運算子有一元，二元和三元運算子。

- 一元運算子對單一操作物件操作（如 `-a`）。一元運算子分前綴運算子和後綴運算子，前綴運算子需要緊跟在操作物件之前（如 `!b`），後綴運算子需要緊跟在操作物件之後（如 `i++`）。
- 二元運算子操作兩個操作物件（如 `2 + 3`），是中綴的，因為它們出現在兩個操作物件之間。
- 三元運算子操作三個操作物件，和 C 語言一樣，Swift 只有一個三元運算子，就是三元條件運算子（`a ? b : c`）。

受運算子影響的值叫運算元，在表達式 `1 + 2` 中，加號 `+` 是二元運算子，它的兩個運算元是值 `1` 和 `2`。

指派運算子

指派運算（`a = b`），表示用 `b` 的值來初始化或更新 `a` 的值：

```
let b = 10
var a = 5
a = b
// a 現在等於 10
```

如果指派的右邊是一個 tuple，它的元素可以馬上被分解多個常數或變數：

```
let (x, y) = (1, 2)
// 現在 x 等於 1, y 等於 2
```

與 C 語言和 Objective-C 不同，Swift 的指派運算子並不回傳任何值。所以以下程式碼是錯誤的：

```
if x = y {
    // 此句錯誤，因為 x = y 並不回傳任何值
}
```

這個特性保護你不會把（`==`）錯寫成（`=`）了，由於`if x = y`是錯誤程式碼，Swift 從底層幫你避免了這些程式碼錯誤。

數值運算子

Swift 支援所有數值型別基本的四則運算：

- 加法（`+`）
- 減法（`-`）
- 乘法（`*`）
- 除法（`/`）

```
1 + 2       // 等於 3
5 - 3       // 等於 2
2 * 3       // 等於 6
10.0 / 2.5  // 等於 4.0
```

與 C 語言和 Objective-C 不同的是，Swift 預設不允許在數值運算中出現溢位情況。但你可以使用 Swift 的溢位運算子來達到你有目的的溢位（如`a &+ b`）。詳情參見溢位運算子。

加法運算子也用於`String`的拼接：

```
"hello, " + "world"  // 等於 "hello, world"
```

兩個`Character`值或一個`String`和一個`Character`值，相加會生成一個新的`String`值：

```
let dog: Character = ""
let cow: Character = ""
let dogCow = dog + cow
// dogCow 現在是 " "
```

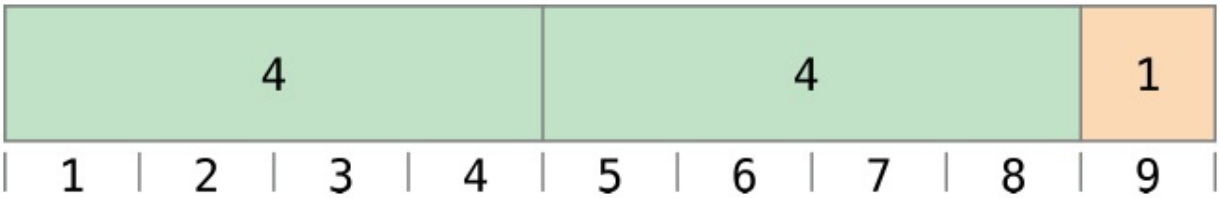
詳情參見字元和字串的拼接。

餘數運算

餘數運算（`a % b`）是計算`b`的多少倍剛剛好可以容入`a`，回傳多出來的那部分（餘數）。

> 注意：
> 餘數運算（`%`）在其他語言也叫取模運算。然而嚴格說來，我們看該運算子對負數的操作結果，"餘數"比"取模"更合適些。

我們來談談取餘數是怎麼回事，計算 `9 % 4`，你先計算出 `4` 的多少倍會剛好可以容入 `9` 中：



2 倍，非常好，那餘數是 1（用橙色標出）

在 Swift 中這麼來表達：

```
9 % 4    // 等於 1
```

為了得到 `a % b` 的結果，`%` 計算了以下等式，並輸出 `餘數` 作為結果：

a = (*b* × 倍數) + 餘數

當 `倍數` 取最大值的時候，就會剛好可以容入 `a` 中。

把 `9` 和 `4` 代入等式中，我們得 `1`：

```
9 = (4 × 2) + 1
```

同樣的方法，我來們計算 `-9 % 4`：

```
-9 % 4   // 等於 -1
```

把 `-9` 和 `4` 代入等式，`-2` 是取到的最大整數：

```
-9 = (4 × -2) + -1
```

餘數是 `-1`。

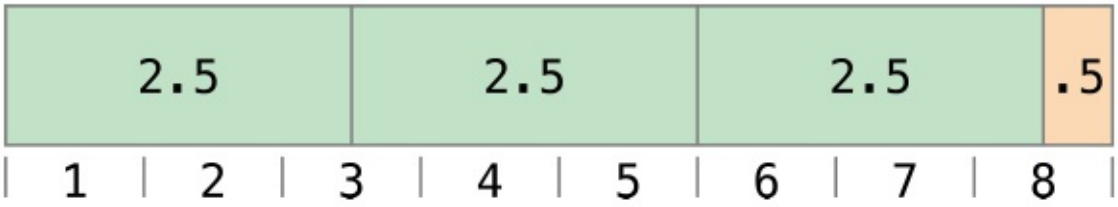
在對負數 `b` 餘數時，`b` 的符號會被忽略。這意味著 `a % b` 和 `a % -b` 的結果是相同的。

浮點數餘數計算

不同於 C 語言和 Objective-C，Swift 是可以對浮點數取餘數的。

```
8 % 2.5 // 等於 0.5
```

這個範例中，`8` 除於 `2.5` 等於 `3` 余 `0.5`，所以結果是一個 `Double` 值 `0.5`。



累加和累減運算

和 C 語言一樣，Swift 也提供了方便對變數本身加 1 或減 1 的累加（`++`）和累減（`--`）運算子。其操作物件可以是整數和浮點數。

```
var i = 0
++i      // 現在 i = 1
```

每呼叫一次`++i`，`i`的值就會加 1。實際上，`++i`是`i = i + 1`的簡寫，而`--i`是`i = i - 1`的簡寫。

`++`和`--`既是前綴又是後綴運算子。`++i`，`i++`，`--i`和`i--`都是有效的寫法。

我們需要注意的是這些運算子修改了`i`後有一個回傳值。如果你只想修改`i`的值，那你就可以忽略這個回傳值。但如果你想使用回傳值，你就需要留意前綴和後綴操作的回傳值是不同的。

- 當`++`前綴的時候，先自增再回傳。
- 當`++`後綴的時候，先回傳再累加。

例如：

```
var a = 0
let b = ++a // a 和 b 現在都是 1
let c = a++ // a 現在 2，但 c 是 a 累加前的值 1
```

上述範例，`let b = ++a`先把`a`加 1 了再回傳`a`的值。所以`a`和`b`都是新值`1`。

而`let c = a++`，是先回傳了`a`的值，然後`a`才加 1。所以`c`得到了`a`的舊值 1，而`a`加 1 後變成 2。

除非你需要使用`i++`的特性，不然推薦你使用`++i`和`--i`，因為先修改後回傳這樣的行為更符合我們的邏輯。

一元負號

數值的正負號可以使用前綴`-`（即一元負號）來切換：

```
let three = 3
let minusThree = -three       // minusThree 等於 -3
let plusThree = -minusThree   // plusThree 等於 3，或 "負負3"
```

一元負號（`-`）寫在運算元之前，中間沒有空格。

一元正號

一元正號（`+`）不做任何改變地回傳運算元的值。

```
let minusSix = -6
let alsoMinusSix = +minusSix  // alsoMinusSix 等於 -6
```

雖然一元 `+` 做無用功，但當你在使用一元負號來表達負數時，你可以使用一元正號來表達正數，如此你的程式碼會具有對稱之美。

複合指派（Compound Assignment Operators）

如同強大的 C 語言，Swift 也提供把其他運算子和指派運算（`=`）組合起來的複合指派運算子，加賦運算（`+=`）是其中一個範例：

```
var a = 1
a += 2 // a 現在是 3
```

表達式 `a += 2` 是 `a = a + 2` 的簡寫，一個加賦運算就把加法和指派兩件事完成了。

> 注意：
> 複合指派運算沒有回傳值，`let b = a += 2` 這類別程式碼是錯誤。這不同於上面提到的累加和累減運算子。

在表達式章節裡有複合運算子的完整列表。

比較運算

所有標準 C 語言中的比較運算都可以在 Swift 中使用。

- 等於（`a == b`）
- 不等於（`a != b`）
- 大於（`a > b`）
- 小於（`a < b`）
- 大於等於（`a >= b`）
- 小於等於（`a <= b`）

> 注意：
> Swift 也提供恆等 `===` 和不恆等 `!==` 這兩個比較符來判斷兩個物件是否參考同一個物件實例。更多細節在類別別與結構。

每個比較運算都回傳了一個顯示表達式是否成立的布林值：

```
1 == 1   // true, 因為 1 等於 1
2 != 1   // true, 因為 2 不等於 1
2 > 1    // true, 因為 2 大於 1
1 < 2    // true, 因為 1 小於2
1 >= 1   // true, 因為 1 大於等於 1
2 <= 1   // false, 因為 2 並不小於等於 1
```

比較運算多用於條件語句，如 `if` 條件：

```
let name = "world"
if name == "world" {
    println("hello, world")
} else {
    println("I'm sorry \(name), but I don't recognize you")
```

```
}
// 輸出 "hello, world", 因為 `name` 就是等於 "world"
```

關於 `if` 語句，請看控制流程程。

三元條件運算(Ternary Conditional Operator)

三元條件運算的特殊在於它是有三個運算元的運算子，它的原型是 `問題 ? 答案1 : 答案2`。它簡潔地表達根據 `問題` 成立與否作出二選一的操作。如果 `問題` 成立，回傳 `答案1` 的結果; 如果不成立，回傳 `答案2` 的結果。

使用三元條件運算簡化了以下程式碼：

```
if question {
  answer1
} else {
  answer2
}
```

這裡有個計算表格行高的範例。如果有表頭，那行高應比內容高度要高出 50 像素; 如果沒有表頭，只需高出 20 像素。

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight 現在是 90
```

這樣寫會比下面的程式碼簡潔：

```
let contentHeight = 40
let hasHeader = true
var rowHeight = contentHeight
if hasHeader {
    rowHeight = rowHeight + 50
} else {
    rowHeight = rowHeight + 20
}
// rowHeight 現在是 90
```

第一段程式碼範例使用了三元條件運算，所以一行程式碼就能讓我們得到正確答案。這比第二段程式碼簡潔得多，無需將 `rowHeight` 定義成變數，因為它的值無需在 `if` 語句中改變。

三元條件運算提供有效率且便捷的方式來表達二選一的選擇。需要注意的事，過度使用三元條件運算就會由簡潔的程式碼變成難懂的程式碼。我們應避免在一個組合語句使用多個三元條件運算子。

空值聚合運算子

空值聚合（nil coalescing）運算子（ `a ?? b` ）如果 `optional a` 有值的話就解析，但若 `a` 是空值，就回傳預設值 `b`。 這運算元 `a` 永遠是個optional型別。而運算子 `b` 必須得式 `a` 的型別。

空值聚合運算子是以下表達式的簡碼。

```
a != nil ? a! : b
```

上述程式碼使用三元條件運算子，並且當 a 不是空值時，強制解析（ a! ）來存取 a，若 a 是空值則回傳 b 。空值聚合運算子題空一個更優雅的方式，並以一個明確可讀的形式來封裝類似的條件確認及解析。

> 注意： 若a是非空值，那就不會運算b。這就是所謂的最小化求值。

下面這個例子使用空值聚合運算子來選擇預設顏色值或是一個optional使用者定義的顏色值。

```
let defaultColorName = "red"
var userDefinedColorName: String?   // 預設為空值

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName 是空值，所以 colorNameToUse 就被設成預設的 「red」。
```

變數 userDefinedColorName 是定義成optional字串型別，且其預設值為空值。因為 userDefinedColorName 變數是個optional型別，你可以使用空值聚合運算子來決定他的值。在上面的例子當中，這運算子被用來定義一個Sting型別 colorNameToUse 變數的初始值。因為 userDefinedColorName 是空值，這表達式 userDefinedColorName ?? defaultColorName 回傳 defaultColorName ，也就是「red」。

如果你把 userDefinedColorName 設成非空值，並且執行空值聚合運算子來確認，則 userDefinedColorName 就會被解析，而不是使用預設值：

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName非空值，所以colorNameToUse被設成「green」。
```

區間運算子

Swift 提供了兩個方便表達一個區間的值的運算子。

閉區間運算子

閉區間運算子（ a...b ）定義一個包含從 a 到 b (包括 a 和 b)的所有值的區間。 a 的值必不大於 b 。 閉區間運算子在迭代一個區間的所有值時是非常有用的，如在 for-in 迴圈中：

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

關於 for-in ，請看控制流程程。

半閉區間運算子

半閉區間運算子（ a..<b ）定義一個從 a 到 b 但不包括 b 的區間。 之所以稱為半閉區間，是因為該區間包含第一個值而不包括最後的值。就像閉區間運算子一樣， a 的值必不大於 b 。如果 a 跟 b 相等，那結果的區間就會是空。

半閉區間運算子的實用性在於當你使用一個 0 始的列表(如陣列)時，非常方便地從 0 數到列表的長度。

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
```

```
for i in 0..count {
    println("第 \(i + 1) 個人叫 \(names[i])")
}
// 第 1 個人叫 Anna
// 第 2 個人叫 Alex
// 第 3 個人叫 Brian
// 第 4 個人叫 Jack
```

陣列有 4 個元素，但 `0..<count` 只數到 3(最後一個元素的索引)，因為它是半閉區間。關於陣列，請查閱陣列。

邏輯運算

邏輯運算的操作物件是邏輯布林值。Swift 支援基於 C 語言的三個標準邏輯運算。

- 邏輯非（ `!a` ）
- 邏輯且（ `a && b` ）
- 邏輯或（ `a || b` ）

邏輯非

邏輯非運算子（ `!a` ）對一個布林值取反，使得 `true` 變成 `false`， `false` 變成 `true` 。

它是一個前綴運算子，需出現在運算元之前，且不加空格。讀作 `非 a` ，然後我們看以下範例：

```
let allowedEntry = false
if !allowedEntry {
    println("ACCESS DENIED")
}
// 輸出 "ACCESS DENIED"
```

`if !allowedEntry` 語句可以讀作「如果非 alowed entry」，接下一行程式碼只有在如果「非 allow entry」為 `true` ，即 `allowEntry` 為 `false` 時被執行。

在範例程式碼中，小心地選擇布林常數或變數有助於程式碼的可讀性，並且避免使用雙重邏輯非運算，或混亂的邏輯語句。

邏輯且

邏輯且運算子（ `a && b` ）表達了只有 `a` 和 `b` 的值都為 `true` 時，整個表達式的值才會是 `true` 。

只要任意一個值為 `false` ，整個表達式的值就為 `false` 。事實上，如果第一個值為 `false` ，那麼是不去計算第二個值的，因為它已經不可能影響整個表達式的結果了。這被稱做「捷徑計算（short-circuit evaluation）」。

以下範例，只有兩個 `Bool` 值都為 `true` 值的時候才允許進入：

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// 輸出 "ACCESS DENIED"
```

邏輯或

邏輯或運算子（ `a || b` ）是一個由兩個連續的 `|` 組成的中綴運算子。它表示了兩個邏輯表達式的其中一個為 `true` ，整個表

達式就為 `true`。

和邏輯且運算子類似，邏輯或也使用「捷徑計算」，當左端的表達式為 `true` 時，將不計算右邊的表達式了，因為它不可能改變整個表達式的值了。

以下範例程式碼中，第一個布林值（`hasDoorKey`）為 `false`，但第二個值（`knowsOverridePassword`）為 `true`，所以整個表達是 `true`，於是允許進入：

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// 輸出 "Welcome!"
```

組合邏輯

我們可以組合多個邏輯運算來表達一個複合邏輯：

```
if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// 輸出 "Welcome!"
```

這個範例使用了含多個 `&&` 和 `||` 的複合邏輯。但無論怎樣，`&&` 和 `||` 始終只能操作兩個值。所以這實際是三個簡單邏輯連續操作的結果。我們來解讀一下：

如果我們輸入了正確的密碼並通過了視網膜掃描; 或者我們有一把有效的鑰匙; 又或者我們知道緊急情況下重置的密碼，我們就能把門打開進入。

前兩種情況，我們都不滿足，所以前兩個簡單邏輯的結果是 `false`，但是我們是知道緊急情況下重置的密碼的，所以整個複雜表達式的值還是 `true`。

使用括號來表明優先級

為了一個複雜表達式更容易讀懂，在合適的地方使用括號來表明優先級是很有效的，雖然它並非必要的。在上個關於門的權限的範例中，我們給第一個部分加個括號，使用它看起來邏輯更明確：

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// 輸出 "Welcome!"
```

這括號使得前兩個值被看成整個邏輯表達中獨立的一個部分。雖然有括號和沒括號的輸出結果是一樣的，但對於讀程式碼的人來說有括號的程式碼更清晰。可讀性比簡潔性更重要，請在可以讓你程式碼變清晰地地方加個括號吧！

> 翻譯：tommy60703

字串和字元（Strings and Characters）

本頁包含內容：

`String` 是例如 "hello, world"、"albatross" 這樣的有序的 `Character`（字元）型別的值的集合，透過 `String` 型別來表示。

Swift 的 `String` 和 `Character` 型別提供了一個快速的、相容 Unicode 的方式來處理程式碼中的文字訊息。 創建和操作字串的語法與 C 語言中字串操作相似，輕量並且易讀。 字串連接操作只需要簡單地透過 `+` 號將兩個字串相連即可。 與 Swift 中其他值一樣，能否更改字串的值，取決於其被定義為常數還是變數。

儘管語法簡單，但 `String` 型別是一種快速、現代化的字串實作。 每一個字串都是由獨立編碼的 Unicode 字元組成，並支援以不同 Unicode 表示（representations）來存取這些字元。

Swift 可以在常數、變數、字面量和表達式中進行字串插值操作，可以輕鬆創建用於顯示、儲存和列印的自定義字串。

> 注意：
> Swift 的 `String` 型別與 Foundation `NSString` 類別進行了無縫橋接。如果你利用 Cocoa 或 Cocoa Touch 中的 Foundation Framework 進行工作。所有 `NSString` API 都可以呼叫你創建的任意 `String` 型別的值。除此之外，還可以使用本章介紹的 `String` 特性。你也可以在任意要求傳入 `NSString` 實體作為參數的 API 中使用 `String` 型別的值來替代。 更多關於在 Foundation 和 Cocoa 中使用 `String` 的資訊請查看 Using Swift with Cocoa and Objective-C。

字串字面量（String Literals）

你可以在你的程式碼中包含一段預先定義的字串值作為字串字面量。 字串字面量是由雙引號 ("") 包著的具有固定順序的文字字元集。

字串字面量可以用於為常數和變數提供初始值。

```
let someString = "Some string literal value"
```

> 注意：
> `someString` 變數透過字串字面量進行初始化，Swift 因此推斷該變數為 `String` 型別。

字串字面量可以包含以下特殊字元：

- 跳脫字元 `\0` (空字元)、 `\\` (反斜線)、 `\t` (水平 tab)、 `\n` (換行)、 `\r` (回車)、 `\"` (雙引號)、 `\'` (單引號)。

- 單位元組 Unicode 純量，寫成 `\xnn`，其中 `nn` 為兩位十六進制數。
- 雙位元組 Unicode 純量，寫成 `\unnnn`，其中 `nnnn` 為四位十六進制數。
- 四位元組 Unicode 純量，寫成 `\Unnnnnnnn`，其中 `nnnnnnnn` 為八位十六進制數。

下面的程式碼為各種特殊字元的使用範例。 `wiseWords` 常數包含了兩個跳脫字元 (雙括號)；`dollarSign`、 `blackHeart` 和 `sparklingHeart` 常數展示了三種不同格式的 Unicode 純量：

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein
// "\"Imagination is more important than knowledge\" - Einstein
let dollarSign = "\x24"             // $,  Unicode 純量 U+0024
let blackHeart = "\u2665"           // ♥,  Unicode 純量 U+2665
let sparklingHeart = "\U0001F496"  // , Unicode  純量 U+1F496
```

初始化空字串 (Initializing an Empty String)

為了建構一個很長的字串，可以創建一個空字串作為初始值。 可以將空的字串字面量指派給變數，也可以初始化一個新的 `String` 實體：

```
var emptyString = ""               // 空字串字面量
var anotherEmptyString = String()  // 初始化 String 實體
// 兩個字串均為空並等價。
```

你可以透過檢查其 `Boolean` 型別的 `isEmpty` 屬性來判斷該字串是否為空：

```
if emptyString.isEmpty {
    println("Nothing to see here")
}
// prints "Nothing to see here"
```

字串可變性 (String Mutability)

你可以透過將一個特定字串分配給一個變數來對其進行修改，或者分配給一個常數來保證其不會被修改：

```
var variableString = "Horse"
variableString += " and carriage"
// variableString 現在為 "Horse and carriage"
let constantString = "Highlander"
constantString += " and another Highlander"
// 這會回報一個編譯錯誤 (compile-time error) - 常數不可以被修改。
```

> 注意：
> 在 Objective-C 和 Cocoa 中，你透過選擇兩個不同的類別別(`NSString` 和 `NSMutableString`)來指定該字串是否可以被修改，Swift 中的字串是否可以修改僅透過定義的是變數還是常數來決定。

字串是實值型別（Strings Are Value Types）

Swift 的 `String` 型別是實值型別。 如果你創建了一個新的字串，那麼當其進行常數、變數指派操作或傳遞給函式/方法時，字串值是被複製一份過去的。 任何情況下，都會對現存字串的值創建新副本，並對該新副本進行傳遞或指派操作。 實值型別在結構和列舉是實值型別 中進行了說明。

> 注意：

> 與 Cocoa 中的 `NSString` 不同，當你在 Cocoa 中創建了一個 `NSString` 實體，並將其傳遞給一個函式/方法，或者指派給一個變數，你傳遞或指派的是該 `NSString` 實體的參考，除非你特別要求，否則字串不會生成新的副本來進行指派操作。

Swift 預設字串複製的方式保證了在函式/方法中傳遞的是字串的值。 很明顯無論該值來自於哪裡，都是你獨自擁有的。 你傳遞的字串本身不會被更改，除非你主動更改它。

在實際編譯時，Swift 編譯器會最佳化字串的使用，使實際的複製只發生在絕對必要的情況下，這意味著你將字串作為實值型別的同時可以獲得極高的效能。

使用字元（Working with Characters）

Swift 的 `String` 型別表示特定序列的 `Character` （字元） 型別的值的集合。 每一個字元值代表一個 Unicode 字元。 你可利用 `for-in` 迴圈來遍歷字串中的每一個字元：

```
for character in "Dog!"  {
    println(character)
}
// D
// o
// g
// !
//
```

for-in 迴圈在 For 迴圈 中進行了詳細介紹。

另外，透過標明一個 `Character` 型別註解並透過字元字面量進行指派，可以建立一個獨立的字元常數或變數：

```
let yenSign: Character = "¥"
```

計算字元數量 (Counting Characters)

透過呼叫全域 `countElements` 函式，並將字串作為參數傳入，可以獲取該字串的字元數量。

```
let unusualMenagerie = "Koala , Snail , Penguin , Dromedary "
println("unusualMenagerie has \(countElements(unusualMenagerie)) characters")
// prints "unusualMenagerie has 40 characters"
```

> 注意：
> 不同的 Unicode 字元以及相同 Unicode 字元的不同表示方式可能需要不同數量的記憶體空間來儲存。所以 Swift 中的字元在一個字串中並不一定占用相同的記憶體空間。因此字串的長度不得不透過遍歷字串中每一個字元的長度來進行計算。如果你正在處理一個長字串，需要注意 `countElements` 函式必須遍歷字串中的字元以精準計算字串的長度。
>
> 另外需要注意的是透過 `countElements` 回傳的字元數量並不總是與包含相同字元的 `NSString` 的 `length` 屬性相同。 `NSString` 的 `length` 屬性是基於利用 UTF-16 表示的十六位代碼單元數字，而不是基於 Unicode 字元。為了解決這個問題， `NSString` 的 `length` 屬性在被 Swift 的 `String` 存取時會成為 `utf16count` 。

連接字串和字元 (Concatenating Strings and Characters)

字串和字元的值可以透過加法運算子（ `+` ）相加在一起並創建一個新的字串值：

```
let string1 = "hello"
let string2 = " there"
let character1: Character = "!"
let character2: Character = "?"

let stringPlusCharacter = string1 + character1        // 等於 "hello!"
let stringPlusString = string1 + string2              // 等於 "hello there"
let characterPlusString = character1 + string1        // 等於 "!hello"
let characterPlusCharacter = character1 + character2  // 等於 "!?"
```

你也可以透過加法指派運算子 (`+=`) 將一個字串或者字元添加到一個已經存在字串變數上：

```
var instruction = "look over"
instruction += string2
// instruction 現在等於 "look over there"

var welcome = "good morning"
welcome += character1
// welcome 現在等於 "good morning!"
```

> 注意：
> 你不能將一個字串或者字元添加到一個已經存在的字元變數上，因為字元變數只能包含一個字元。

字串插值 (String Interpolation)

字串插值是一種構建新字串的方式，可以在其中包含常數、變數、字面量和表達式。 你插入的字串字面量的每一項都被包裹在以反斜線為前綴的圓括號中：

```
let multiplier = 3
let message = "\(multiplier) 乘以 2.5 是 \(Double(multiplier) * 2.5)"
// message 是 "3 乘以 2.5 是 7.5"
```

在上面的範例中， `multiplier` 作為 `\(multiplier)` 被插入到一個字串字面量中。 當創建字串執行插值計算時此占位符（placeholder）會被替換為 `multiplier` 實際的值。

`multiplier` 的值也作為字串中後面表達式的一部分。 該表達式計算 `Double(multiplier) * 2.5` 的值並將結果 (7.5) 插入到字串中。 在這個範例中，表達式寫為 `\(Double(multiplier) * 2.5)` 並包含在字串字面量中。

> 注意：
> 插值字串中寫在括號中的表達式不能包含非跳脫雙引號 (`"`) 和反斜杠 (`\`)，並且不能包含回車 (`\r`)或換行 (`\n`) 符號。

比較字串 (Comparing Strings)

Swift 提供了三種方式來比較字串的值：字串相等、前綴相等和後綴相等。

字串相等 (String Equality)

如果兩個字串以同一順序包含完全相同的字元，則認為兩者字串相等：

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if quotation == sameQuotation {
    println("These two strings are considered equal")
}
```

```
// prints "These two strings are considered equal"
```

前綴/後綴相等 (Prefix and Suffix Equality)

透過呼叫字串的 `hasPrefix` / `hasSuffix` 方法來檢查字串是否擁有特定前綴/後綴。 兩個方法均需要以字串作為參數傳入並回傳布林值。 兩個方法均執行基本字串和前綴/後綴字串之間逐個字元的比較。

下面的範例以一個字串陣列表示莎士比亞話劇《羅密歐與朱麗葉》中前兩場的場景位置：

```
let romeoAndJuliet = [
    "Act 1 Scene 1: Verona, A public place",
    "Act 1 Scene 2: Capulet's mansion",
    "Act 1 Scene 3: A room in Capulet's mansion",
    "Act 1 Scene 4: A street outside Capulet's mansion",
    "Act 1 Scene 5: The Great Hall in Capulet's mansion",
    "Act 2 Scene 1: Outside Capulet's mansion",
    "Act 2 Scene 2: Capulet's orchard",
    "Act 2 Scene 3: Outside Friar Lawrence's cell",
    "Act 2 Scene 4: A street in Verona",
    "Act 2 Scene 5: Capulet's mansion",
    "Act 2 Scene 6: Friar Lawrence's cell"
]
```

你可以利用 `hasPrefix` 方法來計算話劇中第一幕的場景數：

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
    if scene.hasPrefix("Act 1 ") {
        ++act1SceneCount
    }
}
println("There are \(act1SceneCount) scenes in Act 1")
// prints "There are 5 scenes in Act 1"
```

類似地，你可以用 `hasSuffix` 方法來計算發生在不同地方的場景數：

```
var mansionCount = 0
var cellCount = 0
for scene in romeoAndJuliet {
    if scene.hasSuffix("Capulet's mansion") {
        ++mansionCount
    } else if scene.hasSuffix("Friar Lawrence's cell") {
        ++cellCount
    }
}
println("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
// prints "6 mansion scenes; 2 cell scenes」
```

大寫和小寫字串（Uppercase and Lowercase Strings）

你可以透過字串的 `uppercaseString` 和 `lowercaseString` 屬性來存取大寫/小寫版本的字串。

```
let normal = "Could you help me, please?"
let shouty = normal.uppercaseString
// shouty 值為 "COULD YOU HELP ME, PLEASE?"
let whispered = normal.lowercaseString
// whispered 值為 "could you help me, please?"
```

Unicode

Unicode 是一個國際標準，用於文字的編碼和表示。 它使你可以用標準格式表示來自任意語言幾乎所有的字元，並能夠對文字文件或網頁這樣的外部資源中的字元進行讀寫。

Swift 的字串和字元型別是完全相容 Unicode 標準的，它支援如下所述的一系列不同的 Unicode 編碼。

Unicode 術語（Unicode Terminology）

Unicode 中每一個字元都可以被解釋為一個或多個 unicode 純量。 字元的 unicode 純量是一個唯一的 21 位元數字 (和名稱)，例如 `U+0061` 表示小寫的拉丁字母 A ("a")， `U+1F425` 表示小雞表情 ("")。

當 Unicode 字串被寫進程式本文檔或其他儲存結構當中，這些 unicode 純量將會按照 Unicode 定義的格式之一進行編碼。其包括 `UTF-8`（以 8 位元代碼單元進行編碼） 和 `UTF-16`（以 16 位元代碼單元進行編碼）。

字串的 Unicode 表示（Unicode Representations of Strings）

Swift 提供了幾種不同的方式來存取字串的 Unicode 表示。

你可以利用 `for-in` 來對字串進行遍歷，從而以 Unicode 字元的方式存取每一個字元值。 該過程在 使用字元 中進行了介紹。

另外，能夠以其他三種 Unicode 相容的方式存取字串的值：

- UTF-8 代碼單元集合 (利用字串的 `utf8` 屬性進行存取)
- UTF-16 代碼單元集合 (利用字串的 `utf16` 屬性進行存取)
- 21 位元的 Unicode 純量值集合 (利用字串的 `unicodeScalars` 屬性進行存取)

下面由 `D``o``g``!` 和 （`DOG FACE`，Unicode 純量為 `U+1F436`)組成的字串中的每一個字元代表著一種不同的表示：

```
let dogString = "Dog!"
```

UTF-8

你可以透過遍歷字串的 `utf8` 屬性來存取它的 `UTF-8` 表示。 其為 `UTF8View` 型別的屬性， `UTF8View` 是無號 8 位元 (`UInt8`) 值的集合，每一個 `UInt8` 值都是一個字元的 UTF-8 表示：

```
for codeUnit in dogString.utf8 {
    print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 240 159 144 182
```

上面的範例中，前四個 10 進制代碼單元值 (68, 111, 103, 33) 代表了字元 `D` `o` `g` 和 `!`，它們的 UTF-8 表示與 ASCII 表示相同。 後四個代碼單元值 (240, 159, 144, 182) 是 `DOG FACE` 的 4 位元組 UTF-8 表示。

UTF-16

你可以透過遍歷字串的 `utf16` 屬性來存取它的 `UTF-16` 表示。 其為 `UTF16View` 型別的屬性， `UTF16View` 是無號 16 位元 (`UInt16`) 值的集合，每一個 `UInt16` 都是一個字元的 UTF-16 表示：

```
for codeUnit in dogString.utf16 {
    print("\(codeUnit) ")
}
```

```
print("\n")
// 68 111 103 33 55357 56374
```

同樣，前四個代碼單元值 (68, 111, 103, 33) 代表了字元 `D` `o` `g` 和 `!`，它們的 UTF-16 代碼單元和 UTF-8 完全相同。

第五和第六個代碼單元值 (55357 和 56374) 是 `DOG FACE` 字元的 UTF-16 表示。 第一個值為 `U+D83D` (十進制值為 55357)，第二個值為 `U+DC36` (十進制值為 56374)。

Unicode 純量 (Unicode Scalars)

你可以透過遍歷字串的 `unicodeScalars` 屬性來存取它的 Unicode 純量表示。 其為 `UnicodeScalarView` 型別的屬性，`UnicodeScalarView` 是 `UnicodeScalar` 的集合。 `UnicodeScalar` 是 21 位元的 Unicode 程式碼點。

每一個 `UnicodeScalar` 擁有一個值屬性，可以回傳對應的 21 位元數值，用 `UInt32` 來表示。

```
for scalar in dogString.unicodeScalars {
    print("\(scalar.value) ")
}
print("\n")
// 68 111 103 33 128054
```

同樣，前四個代碼單元值 (68, 111, 103, 33) 代表了字元 `D` `o` `g` 和 `!`。 第五位數值，128054，是一個十六進制 1F436 的十進制表示。 其等同於 `DOG FACE` 的 Unicode 純量 U+1F436。

作為查詢字元值屬性的一種替代方法，每個 `UnicodeScalar` 值也可以用來構建一個新的字串值，比如在字串插值中使用：

```
for scalar in dogString.unicodeScalars {
    println("\(scalar) ")
}
// D
// o
// g
// !
//
```

> 翻譯：zqp 校對：shinyzhu, stanzhai

集合型別 (Collection Types)

本頁包含內容：

- 陣列（Arrays）
- 字典（Dictionaries）
- 集合的可變性（Mutability of Collections）

Swift 語言提供經典的陣列和字典兩種集合型別來儲存集合資料。陣列用來按順序儲存相同型別的資料。字典雖然無序儲存相同型別資料值但是需要由獨有的識別符號參考和尋址（就是鍵值對）。

Swift 語言裡的陣列和字典中儲存的資料值型別必須明確。 這意味著我們不能把不正確的資料型別插入其中。 同時這也說明我們完全可以對獲取出的值型別非常自信。 Swift 對顯式型別集合的使用確保了我們的程式碼對工作所需要的型別非常清楚，也讓我們在開發中可以早早地找到任何的型別不匹配錯誤。

> 注意：
> Swift 的陣列結構在被宣告成常數和變數或者被傳入函式與方法中時會相對於其他型別展現出不同的特性。 獲取更多資訊請參見集合的可變性與集合在賦值和複製中的行為章節。

陣列

陣列使用有序列表儲存同一型別的多個值。相同的值可以多次出現在一個陣列的不同位置中。

Swift 陣列特定於它所儲存元素的型別。這與 Objective-C 的 NSArray 和 NSMutableArray 不同，這兩個類別可以儲存任意型別的物件，並且不提供所回傳物件的任何特別資訊。在 Swift 中，資料值在被儲存進入某個陣列之前型別必須明確，方法是通過顯式的型別標注或型別推斷，而且不是必須是 `class` 型別。例如： 如果我們創建了一個 `Int` 值型別的陣列，我們不能往其中插入任何不是 `Int` 型別的資料。 Swift 中的陣列是型別安全的，並且它們中包含的型別必須明確。

陣列的簡單語法

寫 Swift 陣列應該遵循像 `Array<SomeType>` 這樣的形式，其中 `SomeType` 是這個陣列中唯一允許存在的資料型別。 我們也可以使用像 `SomeType[]` 這樣的簡單語法。 儘管兩種形式在功能上是一樣的，但是推薦較短的那種，而且在本文中都會使用這種形式來使用陣列。

陣列建構語法

我們可以使用字面量來進行陣列建構，這是一種用一個或者多個數值建構陣列的簡單方法。字面量是一系列由逗號分割並由方括號包含的數值。 `[value 1, value 2, value 3]`。

下面這個範例創建了一個叫做 `shoppingList` 並且儲存字串的陣列：

```
var shoppingList: String[] = ["Eggs", "Milk"]
// shoppingList 已經被建構並且擁有兩個初始項。
```

`shoppingList` 變數被宣告為「字串值型別的陣列」，記作 `String[]`。 因為這個陣列被規定只有 `String` 一種資料結構，所以只有 `String` 型別可以在其中被存取。 在這裡， `shoppinglist` 陣列由兩個 `String` 值（ `"Eggs"` 和 `"Milk"` ）建構，並且由字面量定義。

> 注意：
> `Shoppinglist`陣列被宣告為變數（`var`關鍵字創建）而不是常數（`let`創建）是因為以後可能會有更多的資料項被插入其中。

在這個範例中，字面量僅僅包含兩個`String`值。匹配了該陣列的變數宣告（只能包含`String`的陣列），所以這個字面量的分配過程就是允許用兩個初始項來建構`shoppinglist`。

由於 Swift 的型別推斷機制，當我們用字面量建構只擁有相同型別值陣列的時候，我們不必把陣列的型別定義清楚。`shoppinglist`的建構也可以這樣寫：

```
var shoppingList = ["Eggs", "Milk"]
```

因為所有字面量中的值都是相同的型別，Swift 可以推斷出`String[]`是`shoppinglist`中變數的正確型別。

存取和修改陣列

我們可以通過陣列的方法和屬性來存取和修改陣列，或者下標語法。 還可以使用陣列的唯讀屬性`count`來獲取陣列中的資料項數量。

```
println("The shopping list contains \(shoppingList.count) items.")
// 輸出"The shopping list contains 2 items."（這個陣列有2個項）
```

使用布林項`isEmpty`來作為檢查`count`屬性的值是否為 0 的捷徑。

```
if shoppingList.isEmpty {
    println("The shopping list is empty.")
} else {
    println("The shopping list is not empty.")
}
// 列印 "The shopping list is not empty."（shoppinglist不是空的）
```

也可以使用`append`方法在陣列後面添加新的資料項：

```
shoppingList.append("Flour")
// shoppingList 現在有3個資料項，有人在攤煎餅
```

除此之外，使用加法賦值運算子（`+=`）也可以直接在陣列後面添加資料項：

```
shoppingList += "Baking Powder"
// shoppingList 現在有四項了
```

我們也可以使用加法賦值運算子（`+=`）直接添加擁有相同型別資料的陣列。

```
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList 現在有7項了
```

可以直接使用下標語法來獲取陣列中的資料項，把我們需要的資料項的索引值放在直接放在陣列名稱的方括號中：

```
var firstItem = shoppingList[0]
// 第一項是 "Eggs"
```

注意第一項在陣列中的索引值是 `0` 而不是 `1`。 Swift 中的陣列索引總是從零開始。

我們也可以用下標來改變某個已有索引值對應的資料值：

```
shoppingList[0] = "Six eggs"
// 其中的第一項現在是 "Six eggs" 而不是 "Eggs"
```

還可以利用下標來一次改變一系列資料值，即使新資料和原有資料的數量是不一樣的。下面的範例把 `"Chocolate Spread"`， `"Cheese"`，和 `"Butter"` 替換為 `"Bananas"` 和 `"Apples"`：

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList 現在有六項
```

> 注意：
> 我們不能使用下標語法在陣列尾部添加新項。如果我們試著用這種方法對索引越界的資料進行檢索或者設置新值的操作，我們會引發一個執行期錯誤。我們可以使用索引值和陣列的 `count` 屬性進行比較來在使用某個索引之前先檢驗是否有效。除了當 `count` 等於 0 時（說明這是個空陣列），最大索引值一直是 `count - 1`，因為陣列都是零起索引。

呼叫陣列的 `insert(atIndex:)` 方法來在某個具體索引值之前添加資料項：

```
shoppingList.insert("Maple Syrup", atIndex: 0)
// shoppingList 現在有7項
// "Maple Syrup" 現在是這個列表中的第一項
```

這次 `insert` 函式呼叫把值為 `"Maple Syrup"` 的新資料項插入列表的最開始位置，並且使用 `0` 作為索引值。

類似的我們可以使用 `removeAtIndex` 方法來移除陣列中的某一項。這個方法把陣列在特定索引值中儲存的資料項移除並且回傳這個被移除的資料項（我們不需要的時候就可以無視它）：

```
let mapleSyrup = shoppingList.removeAtIndex(0)
// 索引值為0的資料項被移除
// shoppingList 現在只有6項，而且不包括Maple Syrup
// mapleSyrup常數的值等於被移除資料項的值 "Maple Syrup"
```

資料項被移除後陣列中的空出項會被自動填補，所以現在索引值為 `0` 的資料項的值再次等於 `"Six eggs"`：

```
firstItem = shoppingList[0]
// firstItem 現在等於 "Six eggs"
```

如果我們只想把陣列中的最後一項移除，可以使用 `removeLast` 方法而不是 `removeAtIndex` 方法來避免我們需要獲取陣列的 `count` 屬性。就像後者一樣，前者也會回傳被移除的資料項：

```
let apples = shoppingList.removeLast()
// 陣列的最後一項被移除了
// shoppingList現在只有5項，不包括cheese
// apples 常數的值現在等於"Apples" 字串
```

陣列的遍歷

我們可以使用 `for-in` 迴圈來遍歷所有陣列中的資料項：

```
for item in shoppingList {
    println(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

如果我們同時需要每個資料項的值和索引值，可以使用全域 `enumerate` 函式來進行陣列遍歷。 `enumerate` 回傳一個由每一個資料項索引值和資料值組成的元組。我們可以把這個元組分解成臨時常數或者變數來進行遍歷：

```
for (index, value) in enumerate(shoppingList) {
    println("Item \(index + 1): \(value)")
}
// Item 1: Six eggs
// Item 2: Milk
// Item 3: Flour
// Item 4: Baking Powder
// Item 5: Bananas
```

更多關於 `for-in` 迴圈的介紹請參見for 迴圈。

創建並且建構一個陣列

我們可以使用建構語法來創建一個由特定資料型別構成的空陣列：

```
var someInts = Int[]()
println("someInts is of type Int[] with \(someInts.count) items。")
// 列印 "someInts is of type Int[] with 0 items。"（someInts是0資料項的Int[]陣列）
```

注意 `someInts` 被設置為一個 `Int[]` 建構函式的輸出所以它的變數型別被定義為 `Int[]` 。

除此之外，如果程式碼上下文中提供了型別資訊， 例如一個函式參數或者一個已經定義好型別的常數或者變數，我們可以使用空陣列語句創建一個空陣列，它的寫法很簡單： `[]` （一對空方括號）：

```
someInts.append(3)
// someInts 現在包含一個INT值
someInts = []
// someInts 現在是空陣列，但是仍然是Int[]型別的。
```

Swift 中的 `Array` 型別還提供一個可以創建特定大小並且所有資料都被預設的建構方法。我們可以把準備加入新陣列的資料項數量（ `count` ）和適當型別的初始值（ `repeatedValue` ）傳入陣列建構函式：

```
var threeDoubles = Double[](count: 3, repeatedValue:0.0)
// threeDoubles 是一種 Double[]陣列，等於 [0.0, 0.0, 0.0]
```

因為型別推斷的存在，我們使用這種建構方法的時候不需要特別指定陣列中儲存的資料型別，因為型別可以從預設值推斷出來：

```
var anotherThreeDoubles = Array(count: 3, repeatedValue: 2.5)
// anotherThreeDoubles is inferred as Double[], and equals [2.5, 2.5, 2.5]
```

最後，我們可以使用加法運算子（`+`）來組合兩種已存在的相同型別陣列。新陣列的資料型別會被從兩個陣列的資料型別中推斷出來：

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles 被推斷為 Double[], 等於 [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

字典

字典是一種儲存多個相同型別的值的容器。每個值（value）都關聯唯一的鍵（key），鍵作為字典中的這個值資料的識別符號。和陣列中的資料項不同，字典中的資料項並沒有具體順序。我們在需要通過識別符號（鍵）存取資料的時候使用字典，這種方法很大程度上和我們在現實世界中使用字典查字義的方法一樣。

Swift 的字典使用時需要具體規定可以儲存鍵和值型別。不同於 Objective-C 的 `NSDictionary` 和 `NSMutableDictionary` 類別可以使用任何型別的物件來作鍵和值並且不提供任何關於這些物件的本質資訊。在 Swift 中，在某個特定字典中可以儲存的鍵和值必須提前定義清楚，方法是通過顯性型別標注或者型別推斷。

Swift 的字典使用 `Dictionary<KeyType, ValueType>` 定義,其中 `KeyType` 是字典中鍵的資料型別， `ValueType` 是字典中對應於這些鍵所儲存值的資料型別。

`KeyType` 的唯一限制就是可雜湊的，這樣可以保證它是獨一無二的，所有的 Swift 基本型別（例如 `String`， `Int`， `Double` 和 `Bool`）都是預設可雜湊的，並且所有這些型別都可以在字典中當做鍵使用。未關聯值的列舉成員（參見列舉）也是預設可雜湊的。

字典字面量

我們可以使用字典字面量來建構字典，它們和我們剛才介紹過的陣列字面量擁有相似語法。一個字典字面量是一個定義擁有一個或者多個鍵值對的字典集合的簡單語句。

一個鍵值對是一個 `key` 和一個 `value` 的結合體。在字典字面量中，每一個鍵值對的鍵和值都由冒號分割。這些鍵值對構成一個列表，其中這些鍵值對由方括號包含並且由逗號分割：

```
[key 1: value 1, key 2: value 2, key 3: value 3]
```

下面的範例創建了一個儲存國際機場名稱的字典。在這個字典中鍵是三個字母的國際航空運輸相關程式碼，值是機場名稱：

```
var airports: Dictionary<String, String> = ["TYO": "Tokyo", "DUB": "Dublin"]
```

`airports` 字典被定義為一種 `Dictionary<String, String>`,它意味著這個字典的鍵和值都是 `String` 型別。

> 注意：
> `airports` 字典被宣告為變數（用 `var` 關鍵字）而不是常數（ `let` 關鍵字）因為後來更多的機場資訊會被添加到這個示例字典中。

`airports` 字典使用字典字面量初始化，包含兩個鍵值對。第一對的鍵是 `TYO`， 值是 `Tokyo`。第二對的鍵是 `DUB`， 值是 `Dublin`。

這個字典語句包含了兩個 `String: String` 型別的鍵值對。它們對應 `airports` 變數宣告的型別（一個只有 `String` 鍵和 `String` 值的字典）所以這個字典字面量是建構兩個初始資料項的 `airport` 字典。

和陣列一樣，如果我們使用字面量建構字典就不用把型別定義清楚。`airports` 的也可以用這種方法簡短定義：

```
var airports = ["TYO": "Tokyo", "DUB": "Dublin"]
```

因為這個語句中所有的鍵和值都分別是相同的資料型別，Swift 可以推斷出 `Dictionary<String, String>` 是 `airports` 字典的正確型別。

讀取和修改字典

我們可以通過字典的方法和屬性來讀取和修改字典，或者使用下標語法。和陣列一樣，我們可以通過字典的唯讀屬性 `count` 來獲取某個字典的資料項數量：

```
println("The dictionary of airports contains \(airports.count) items.")
// 列印 "The dictionary of airports contains 2 items."（這個字典有兩個資料項）
```

我們也可以在字典中使用下標語法來添加新的資料項。可以使用一個合適型別的 key 作為下標索引，並且分配新的合適型別的值：

```
airports["LHR"] = "London"
// airports 字典現在有三個資料項
```

我們也可以使用下標語法來改變特定鍵對應的值：

```
airports["LHR"] = "London Heathrow"
// "LHR"對應的值 被改為 "London Heathrow
```

作為另一種下標方法，字典的 `updateValue(forKey:)` 方法可以設置或者更新特定鍵對應的值。就像上面所示的示例，`updateValue(forKey:)` 方法在這個鍵不存在對應值的時候設置值或者在存在時更新已存在的值。和上面的下標方法不一樣，這個方法回傳更新值之前的原值。這樣方便我們檢查更新是否成功。

`updateValue(forKey:)` 函式會回傳包含一個字典值型別的可選值。舉例來說：對於儲存 `String` 值的字典，這個函式會回傳一個 `String?` 或者「可選 `String` 」型別的值。如果值存在，則這個可選值值等於被替換的值，否則將會是 `nil`。

```
if let oldValue = airports.updateValue("Dublin Internation", forKey: "DUB") {
    println("The old value for DUB was \(oldValue).")
}
// 輸出 "The old value for DUB was Dublin."（DUB原值是dublin）
```

我們也可以使用下標語法來在字典中檢索特定鍵對應的值。由於使用一個沒有值的鍵這種情況是有可能發生的，可選型別回傳這個鍵存在的相關值，否則就回傳 `nil`：

```
if let airportName = airports["DUB"] {
    println("The name of the airport is \(airportName).")
} else {
    println("That airport is not in the airports dictionary.")
}
// 列印 "The name of the airport is Dublin Internation."（機場的名字是都柏林國際）
```

我們還可以使用下標語法來通過給某個鍵的對應值賦值為 `nil` 來從字典裡移除一個鍵值對：

```
airports["APL"] = "Apple Internation"
// "Apple Internation"不是真的 APL機場, 刪除它
airports["APL"] = nil
// APL現在被移除了
```

另外，`removeValueForKey`方法也可以用來在字典中移除鍵值對。這個方法在鍵值對存在的情況下會移除該鍵值對並且回傳被移除的value或者在沒有值的情況下回傳`nil`：

```
if let removedValue = airports.removeValueForKey("DUB") {
    println("The removed airport's name is \(removedValue).")
} else {
    println("The airports dictionary does not contain a value for DUB.")
}
// prints "The removed airport's name is Dublin International."
```

字典遍歷

我們可以使用`for-in`迴圈來遍歷某個字典中的鍵值對。每一個字典中的資料項都由`(key, value)`元組形式回傳，並且我們可以使用臨時常數或者變數來分解這些元組：

```
for (airportCode, airportName) in airports {
    println("\(airportCode): \(airportName)")
}
// TYO: Tokyo
// LHR: London Heathrow
```

`for-in`迴圈請參見For 迴圈。

我們也可以通過存取它的`keys`或者`values`屬性（都是可遍歷集合）檢索一個字典的鍵或者值：

```
for airportCode in airports.keys {
    println("Airport code: \(airportCode)")
}
// Airport code: TYO
// Airport code: LHR

for airportName in airports.values {
    println("Airport name: \(airportName)")
}
// Airport name: Tokyo
// Airport name: London Heathrow
```

如果我們只是需要使用某個字典的鍵集合或者值集合來作為某個接受`Array`實例 API 的參數，可以直接使用`keys`或者`values`屬性直接建構一個新陣列：

```
let airportCodes = Array(airports.keys)
// airportCodes is ["TYO", "LHR"]

let airportNames = Array(airports.values)
// airportNames is ["Tokyo", "London Heathrow"]
```

> 注意：
> Swift 的字典型別是無序集合型別。其中字典鍵，值，鍵值對在遍歷的時候會重新排列，而且其中順序是不固定的。

創建一個空字典

我們可以像陣列一樣使用建構語法創建一個空字典：

```
var namesOfIntegers = Dictionary<Int, String>()
// namesOfIntegers 是一個空的 Dictionary<Int, String>
```

這個範例創建了一個 `Int, String` 型別的空字典來儲存英語對整數的命名。它的鍵是 `Int` 型，值是 `String` 型。

如果上下文已經提供了資訊型別，我們可以使用空字典字面量來創建一個空字典，記作 `[:]`（中括號中放一個冒號）：

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers 現在包含一個鍵值對
namesOfIntegers = [:]
// namesOfIntegers 又成為了一個 Int, String型別的空字典
```

> 注意：
> 在後台，Swift 的陣列和字典都是由泛型集合來實作的，想了解更多泛型和集合資訊請參見泛型。

集合的可變性

陣列和字典都是在單個集合中儲存可變值。如果我們創建一個陣列或者字典並且把它分配成一個變數，這個集合將會是可變的。這意味著我們可以在創建之後添加更多或移除已存在的資料項來改變這個集合的大小。與此相反，如果我們把陣列或字典分配成常數，那麼它就是不可變的，它的大小不能被改變。

對字典來說，不可變性也意味著我們不能替換其中任何現有鍵所對應的值。不可變字典的內容在被首次設定之後不能更改。不可變性對陣列來說有一點不同，當然我們不能試著改變任何不可變陣列的大小，但是我們可以重新設定相對現存索引所對應的值。這使得 Swift 陣列在大小被固定的時候依然可以做的很棒。

Swift 陣列的可變性行為同時影響了陣列實例如何被分配和修改，想獲取更多資訊，請參見集合在賦值和複製中的行為。

> 注意：
> 在我們不需要改變陣列大小的時候創建不可變陣列是很好的習慣。如此 Swift 編譯器可以優化我們創建的集合。

> 翻譯：vclwei, coverxit, NicePiao 校對：coverxit, stanzhai

控制流程

本頁包含內容：

Swift提供了類似 C 語言的流程控制結構，包括可以多次執行任務的 `for` 和 `while` 迴圈，基於特定條件選擇執行不同程式碼分支的 `if` 和 `switch` 語句，還有控制流程程跳轉到其他程式碼的 `break` 和 `continue` 語句。

除了 C 語言裡面傳統的 for 條件遞增（ `for-condition-increment` ）迴圈，Swift 還增加了 `for-in` 迴圈，用來更簡單地遍歷陣列（array），字典（dictionary），區間（range），字串（string）和其他序列型別。

Swift 的 `switch` 語句比 C 語言中更加強大。在 C 語言中，如果某個 case 不小心漏寫了 `break` ，這個 case 就會貫穿（fallthrough）至下一個 case，Swift 無需寫 `break` ，所以不會發生這種貫穿（fallthrough）的情況。case 還可以匹配更多的型別模式，包括區間匹配（range matching），元組（tuple）和特定型別的描述。 `switch` 的 case 語句中匹配的值可以是由 case 體內部臨時的常數或者變數決定，也可以由 `where` 分句描述更複雜的匹配條件。

For 迴圈

`for` 迴圈用來按照指定的次數多次執行一系列語句。Swift 提供兩種 `for` 迴圈形式：

- `for-in` 用來遍歷一個區間（range），序列（sequence），集合（collection），系列（progression）裡面所有的元素執行一系列語句。
- for條件遞增（ `for-condition-increment` ）語句，用來重複執行一系列語句直到達成特定條件達成，一般通過在每次迴圈完成後增加計數器的值來實作。

For-In

你可以使用 `for-in` 迴圈來遍歷一個集合裡面的所有元素，例如由數字表示的區間、陣列中的元素、字串中的字元。

下面的範例用來輸出乘 5 乘法表前面一部分內容：

```
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
```

範例中用來進行遍歷的元素是一組使用閉區間運算子（ `...` ）表示的從 `1` 到 `5` 的數字。 `index` 被賦值為閉區間中的第一個數字（ `1` ），然後迴圈中的語句被執行一次。在本例中，這個迴圈只包含一個語句，用來輸出當前 `index` 值所對應的乘 5 乘法表結果。該語句執行後， `index` 的值被更新為閉區間中的第二個數字（ `2` ），之後 `println` 方法會再執行一次。整個過程會進行到閉區間結尾為止。

上面的範例中， `index` 是一個每次迴圈遍歷開始時被自動賦值的常數。這種情況下， `index` 在使用前不需要宣告，只需要將

它包含在迴圈的宣告中，就可以對其進行隱式宣告，而無需使用 `let` 關鍵字宣告。

> 注意：
> `index` 常數只存在於迴圈的生命周期裡。如果你想在迴圈完成後存取 `index` 的值，又或者想讓 `index` 成為一個變數而不是常數，你必須在迴圈之前自己進行宣告。

如果你不需要知道區間內每一項的值，你可以使用底線（ `_` ）替代變數名來忽略對值的存取：

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

這個範例計算 base 這個數的 power 次冪（本例中，是 `3` 的 `10` 次冪），從 `1` （ `3` 的 `0` 次冪）開始做 `3` 的乘法， 進行 `10` 次，使用 `1` 到 `10` 的閉區間迴圈。這個計算並不需要知道每一次迴圈中計數器具體的值，只需要執行了正確的迴圈次數即可。底線符號 `_` （替代迴圈中的變數）能夠忽略具體的值，並且不提供迴圈遍歷時對值的存取。

使用 `for-in` 遍歷一個陣列所有元素：

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

你也可以通過遍歷一個字典來存取它的鍵值對（key-value pairs）。遍歷字典時，字典的每項元素會以 `(key, value)` 元組的形式回傳，你可以在 `for-in` 迴圈中使用顯式的常數名稱來解讀 `(key, value)` 元組。下面的範例中，字典的鍵（key）解讀為常數 `animalName` ，字典的值會被解讀為常數 `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
```

字典元素的遍歷順序和插入順序可能不同，字典的內容在內部是無序的，所以遍歷元素時不能保證順序。關於陣列和字典，詳情參見集合型別。

除了陣列和字典，你也可以使用 `for-in` 迴圈來遍歷字串中的字元（ `Character` ）：

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

For條件遞增（for-condition-increment）

除了 `for-in` 迴圈，Swift 提供使用條件判斷和遞增方法的標準 C 樣式 `for` 迴圈:

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

下面是一般情況下這種迴圈方式的格式：

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

和 C 語言中一樣，分號將迴圈的定義分為 3 個部分，不同的是，Swift 不需要使用圓括號將「initialization; condition; increment」包括起來。

這個迴圈執行流程如下：

1. 迴圈首次啟動時，初始化表達式（*initialization expression*）被呼叫一次，用來初始化迴圈所需的所有常數和變數。
2. 條件表達式（*condition expression*）被呼叫，如果表達式呼叫結果為 `false`，迴圈結束，繼續執行 `for` 迴圈關閉大括號（`}`）之後的程式碼。如果表達式呼叫結果為 `true`，則會執行大括號內部的程式碼（*statements*）。
3. 執行所有語句（*statements*）之後，執行遞增表達式（*increment expression*）。通常會增加或減少計數器的值，或者根據語句（*statements*）輸出來修改某一個初始化的變數。當遞增表達式執行完成後，重複執行第 2 步，條件表達式會再次執行。

上述描述和迴圈格式等同於：

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

在初始化表達式中宣告的常數和變數（比如 `var index = 0`）只在 `for` 迴圈的生命周期裡有效。如果想在迴圈結束後存取 `index` 的值，你必須要在迴圈生命周期開始前宣告 `index`。

```
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")
// 輸出 "The loop statements were executed 3 times
```

注意 `index` 在迴圈結束後最終的值是 `3` 而不是 `2`。最後一次呼叫遞增表達式 `++index` 會將 `index` 設置為 `3`，從而導致 `index < 3` 條件為 `false`，並終止迴圈。

While 迴圈

`while` 迴圈執行一系列語句直到條件變成 `false`。這類別迴圈適合使用在第一次迭代前迭代次數未知的情況下。Swift 提供兩

種 `while` 迴圈形式：

- `while` 迴圈，每次在迴圈開始時計算條件是否符合；
- `do-while` 迴圈，每次在迴圈結束時計算條件是否符合。

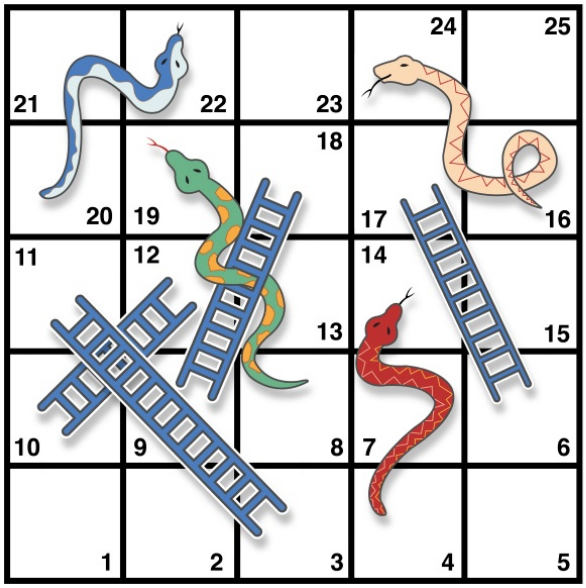
While

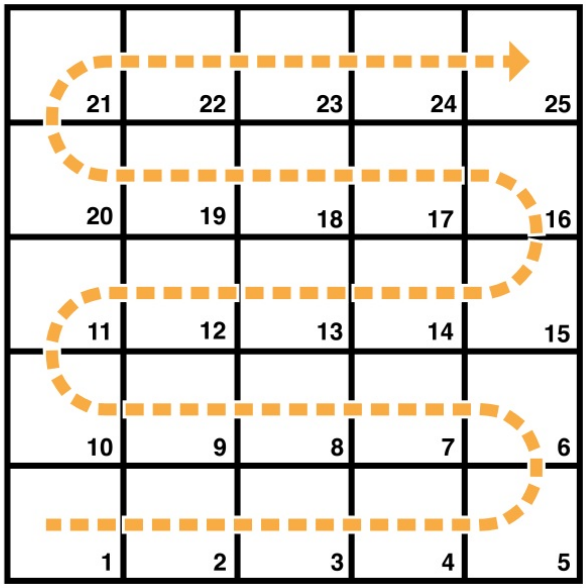
`while` 迴圈從計算單一條件開始。如果條件為 `true`，會重複執行一系列語句，直到條件變為 `false`。

下面是一般情況下 `while` 迴圈格式：

```
while condition {
statements
}
```

下面的範例來玩一個叫做蛇和梯子（*Snakes and Ladders*）的小遊戲，也叫做滑道和梯子（*Chutes and Ladders*）：





遊戲的規則如下：

- 遊戲盤面包括 25 個方格，遊戲目標是達到或者超過第 25 個方格；
- 每一輪，你通過擲一個 6 邊的骰子來確定你移動方塊的步數，移動的路線由上圖中橫向的虛線所示；
- 如果在某輪結束，你移動到了梯子的底部，可以順著梯子爬上去；
- 如果在某輪結束，你移動到了蛇的頭部，你會順著蛇的身體滑下去。

遊戲盤面可以使用一個 `Int` 陣列來表達。陣列的長度由一個 `finalSquare` 常數儲存，用來初始化陣列和檢測最終勝利條件。遊戲盤面由 26 個 `Int` 0 值初始化，而不是 25 個（由 `0` 到 `25`，一共 26 個）：

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

一些方塊被設置成有蛇或者梯子的指定值。梯子底部的方塊是一個正值，使你可以向上移動，蛇頭處的方塊是一個負值，會讓你向下移動：

```
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 號方塊是梯子的底部，會讓你向上移動到 11 號方格，我們使用 `board[03]` 等於 `+08` （來表示 `11` 和 `3` 之間的差值）。使用一元加運算子（ `+i` ）是為了和一元減運算子（ `-i` ）對稱，為了讓盤面程式碼整齊，小於 10 的數字都使用 0 補齊（這些風格上的調整都不是必須的，只是為了讓程式碼看起來更加整潔）。

玩家由左下角編號為 0 的方格開始遊戲。一般來說玩家第一次擲骰子後才會進入遊戲盤面：

```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // 擲骰子
    if ++diceRoll == 7 { diceRoll = 1 }
    // 根據點數移動
    square += diceRoll
    if square < board.count {
        // 如果玩家還在棋盤上，順著梯子爬上去或者順著蛇滑下去
        square += board[square]
    }
}
println("Game over!")
```

本例中使用了最簡單的方法來模擬擲骰子。 `diceRoll` 的值並不是一個隨機數，而是以 `0` 為初始值，之後每一次 `while` 迴圈， `diceRoll` 的值使用前綴累加運算子(`++i`)來累加 1，然後檢測是否超出了最大值。 `++diceRoll` 呼叫完成後，回傳值等於 `diceRoll` 累加後的值。任何時候如果 `diceRoll` 的值等於7時，就超過了骰子的最大值，會被重置為 `1` 。所以 `diceRoll` 的取值順序會一直是 `1` ， `2` ， `3` ， `4` ， `5` ， `6` ， `1` ， `2` 。

擲完骰子後，玩家向前移動 `diceRoll` 個方格，如果玩家移動超過了第 25 個方格，這個時候遊戲結束，相應地，程式碼會在 `square` 增加 `board[square]` 的值向前或向後移動（遇到了梯子或者蛇）之前，檢測 `square` 的值是否小於 `board` 的 `count` 屬性。

如果沒有這個檢測（ `square < board.count` ）， `board[square]` 可能會越界存取 `board` 陣列，導致錯誤。例如如果 `square` 等於 `26` ， 程式碼會去嘗試存取 `board[26]` ，超過陣列的長度。

當本輪 `while` 迴圈執行完畢，會再檢測迴圈條件是否需要再執行一次迴圈。如果玩家移動到或者超過第 25 個方格，迴圈條件結果為 `false` ，此時遊戲結束。

`while` 迴圈比較適合本例中的這種情況，因為在 `while` 迴圈開始時，我們並不知道遊戲的長度或者迴圈的次數，只有在達成指定條件時迴圈才會結束。

Do-While

`while` 迴圈的另外一種形式是 `do-while` ，它和 `while` 的區別是在判斷迴圈條件之前，先執行一次迴圈的程式碼區塊，然後重複迴圈直到條件為 `false` 。

下面是一般情況下 `do-while` 迴圈的格式：

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

還是蛇和梯子的遊戲，使用 `do-while` 迴圈來替代 `while` 迴圈。 `finalSquare` 、 `board` 、 `square` 和 `diceRoll` 的值初始化同 `while` 迴圈一樣：

```
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
```

`do-while` 的迴圈版本，迴圈中第一步就需要去檢測是否在梯子或者蛇的方塊上。沒有梯子會讓玩家直接上到第 25 個方格，所以玩家不會通過梯子直接贏得遊戲。這樣在迴圈開始時先檢測是否踩在梯子或者蛇上是安全的。

遊戲開始時，玩家在第 0 個方格上，`board[0]` 一直等於 0， 不會有什麼影響：

```
do {
    // 順著梯子爬上去或者順著蛇滑下去
    square += board[square]
    // 擲骰子
    if ++diceRoll == 7 { diceRoll = 1 }
    // 根據點數移動
    square += diceRoll
} while square < finalSquare
println("Game over!")
```

檢測完玩家是否踩在梯子或者蛇上之後，開始擲骰子，然後玩家向前移動 `diceRoll` 個方格，本輪迴圈結束。

迴圈條件（`while square < finalSquare`）和 `while` 方式相同，但是只會在迴圈結束後進行計算。在這個遊戲中，`do-while` 表現得比 `while` 迴圈更好。`do-while` 方式會在條件判斷 `square` 沒有超出後直接執行 `square += board[square]`，這種方式可以去掉 `while` 版本中的陣列越界判斷。

條件語句

根據特定的條件執行特定的程式碼通常是十分有用的，例如：當錯誤發生時，你可能想執行額外的程式碼；或者，當輸入的值太大或太小時，向使用者顯示一條訊息等。要實作這些功能，你就需要使用條件語句。

Swift 提供兩種型別的條件語句：`if` 語句和 `switch` 語句。通常，當條件較為簡單且可能的情況很少時，使用 `if` 語句。而 `switch` 語句更適用於條件較複雜、可能情況較多且需要用到模式匹配（pattern-matching）的情境。

If

`if` 語句最簡單的形式就是只包含一個條件，當且僅當該條件為 `true` 時，才執行相關程式碼：

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

上面的範例會判斷溫度是否小於等於 32 華氏度（水的冰點）。如果是，則列印一條訊息；否則，不列印任何訊息，繼續執行 `if` 塊後面的程式碼。

當然，`if` 語句允許二選一，也就是當條件為 `false` 時，執行 *else* 語句：

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

顯然，這兩條分支中總有一條會被執行。由於溫度已升至 40 華氏度，不算太冷，沒必要再圍圍巾——因此，`else` 分支就被觸發了。

你可以把多個 `if` 語句鏈接在一起，像下面這樣：

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

在上面的範例中，額外的 `if` 語句用於判斷是不是特別熱。而最後的 `else` 語句被保留了下來，用於列印既不冷也不熱時的訊息。

實際上，最後的 `else` 語句是可選的：

```
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.")
}
```

在這個範例中，由於既不冷也不熱，所以不會觸發 `if` 或 `else if` 分支，也就不會列印任何訊息。

Switch

`switch` 語句會嘗試把某個值與若干個模式（pattern）進行匹配。根據第一個匹配成功的模式， `switch` 語句會執行對應的程式碼。當有可能的情況較多時，通常用 `switch` 語句替換 `if` 語句。

`switch` 語句最簡單的形式就是把某個值與一個或若干個相同型別的值作比較：

> 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`
> }

`switch` 語句都由多個 *case* 構成。為了匹配某些更特定的值，Swift 提供了幾種更複雜的匹配模式，這些模式將在本節的稍後部分提到。

每一個 case 都是程式碼執行的一條分支，這與 `if` 語句類似。與之不同的是， `switch` 語句會決定哪一條分支應該被執行。

`switch` 語句必須是完備的。這就是說，每一個可能的值都必須至少有一個 case 分支與之對應。在某些不可能涵蓋所有值的情況下，你可以使用預設（ `default` ）分支滿足該要求，這個預設分支必須在 `switch` 語句的最後面。

下面的範例使用 `switch` 語句來匹配一個名為 `someCharacter` 的小寫字元：

```
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")
}
// 輸出 "e is a vowel"
```

在這個範例中，第一個 case 分支用於匹配五個母音，第二個 case 分支用於匹配所有的子音。

由於為其它可能的字元寫 case 分支沒有實際的意義，因此在這個範例中使用了預設分支來處理剩下的既不是母音也不是子音的字元——這就保證了 `switch` 語句的完備性。

不存在隱式的貫穿（No Implicit Fallthrough）

與 C 語言和 Objective-C 中的 `switch` 語句不同，在 Swift 中，當匹配的 case 分支中的程式碼執行完畢後，程式會終止 `switch` 語句，而不會繼續執行下一個 case 分支。這也就是說，不需要在 case 分支中顯式地使用 `break` 語句。這使得 `switch` 語句更安全、更易用，也避免了因忘記寫 `break` 語句而產生的錯誤。

> 注意：
> 你依然可以在 case 分支中的程式碼執行完畢前跳出，詳情請參考Switch 語句中的 break。

每一個 case 分支都必須包含至少一條語句。像下面這樣書寫程式碼是無效的，因為第一個 case 分支是空的：

```
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
```

不像 C 語言裡的 `switch` 語句，在 Swift 中，`switch` 語句不會同時匹配 `"a"` 和 `"A"`。相反的，上面的程式碼會引起編譯期錯誤：`case "a": does not contain any executable statements`——這就避免了意外地從一個 case 分支貫穿到另外一個，使得程式碼更安全、也更直觀。

一個 case 也可以包含多個模式，用逗號把它們分開（如果太長了也可以分行寫）：

> switch `some value to consider` {
> case `value 1`,
> `value 2`:
> `statements`
> }
>
> 注意：
> 如果想要貫穿至特定的 case 分支中，請使用 `fallthrough` 語句，詳情請參考貫穿（Fallthrough）。

區間匹配（Range Matching）

case 分支的模式也可以是一個值的區間。下面的範例展示了如何使用區間匹配來輸出任意數字對應的自然語言格式：

```
let count = 3_000_000_000_000
let countedThings = "stars in the Milky Way"
var naturalCount: String
switch count {
case 0:
    naturalCount = "no"
```

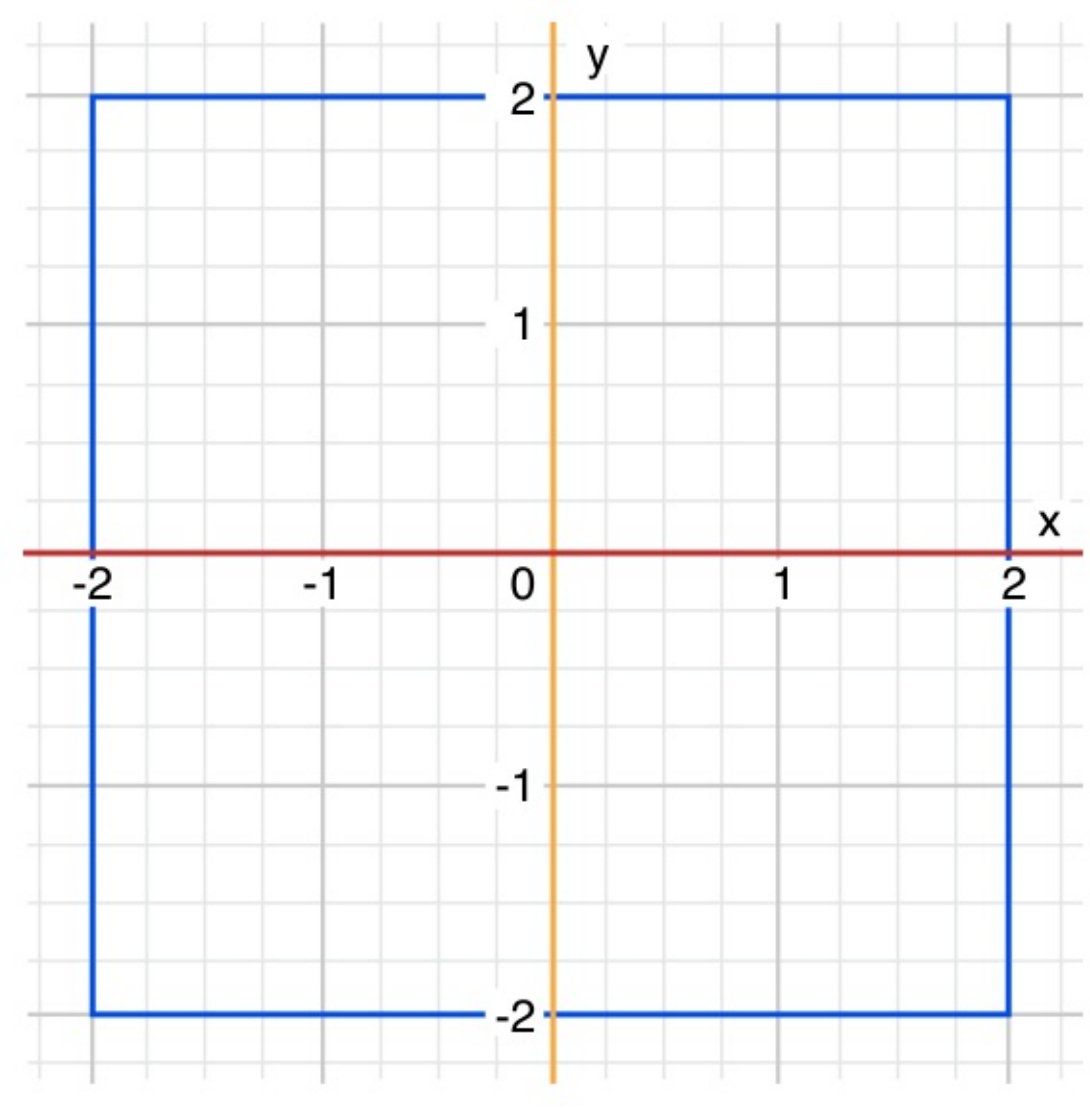
```
case 1...3:
    naturalCount = "a few"
case 4...9:
    naturalCount = "several"
case 10...99:
    naturalCount = "tens of"
case 100...999:
    naturalCount = "hundreds of"
case 1000...999_999:
    naturalCount = "thousands of"
default:
    naturalCount = "millions and millions of"
}
println("There are \(naturalCount) \(countedThings).")
// 輸出 "There are millions and millions of stars in the Milky Way."
```

元組（Tuple）

你可以使用元組在同一個 `switch` 語句中測試多個值。元組中的元素可以是值，也可以是區間。另外，使用底線（`_`）來匹配所有可能的值。

下面的範例展示了如何使用一個 `(Int, Int)` 型別的元組來分類別下圖中的點(x, y)：

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



在上面的範例中，`switch` 語句會判斷某個點是否是原點(0, 0)，是否在紅色的x軸上，是否在黃色y軸上，是否在一個以原點為中心的4x4的矩形裡，或者在這個矩形外面。

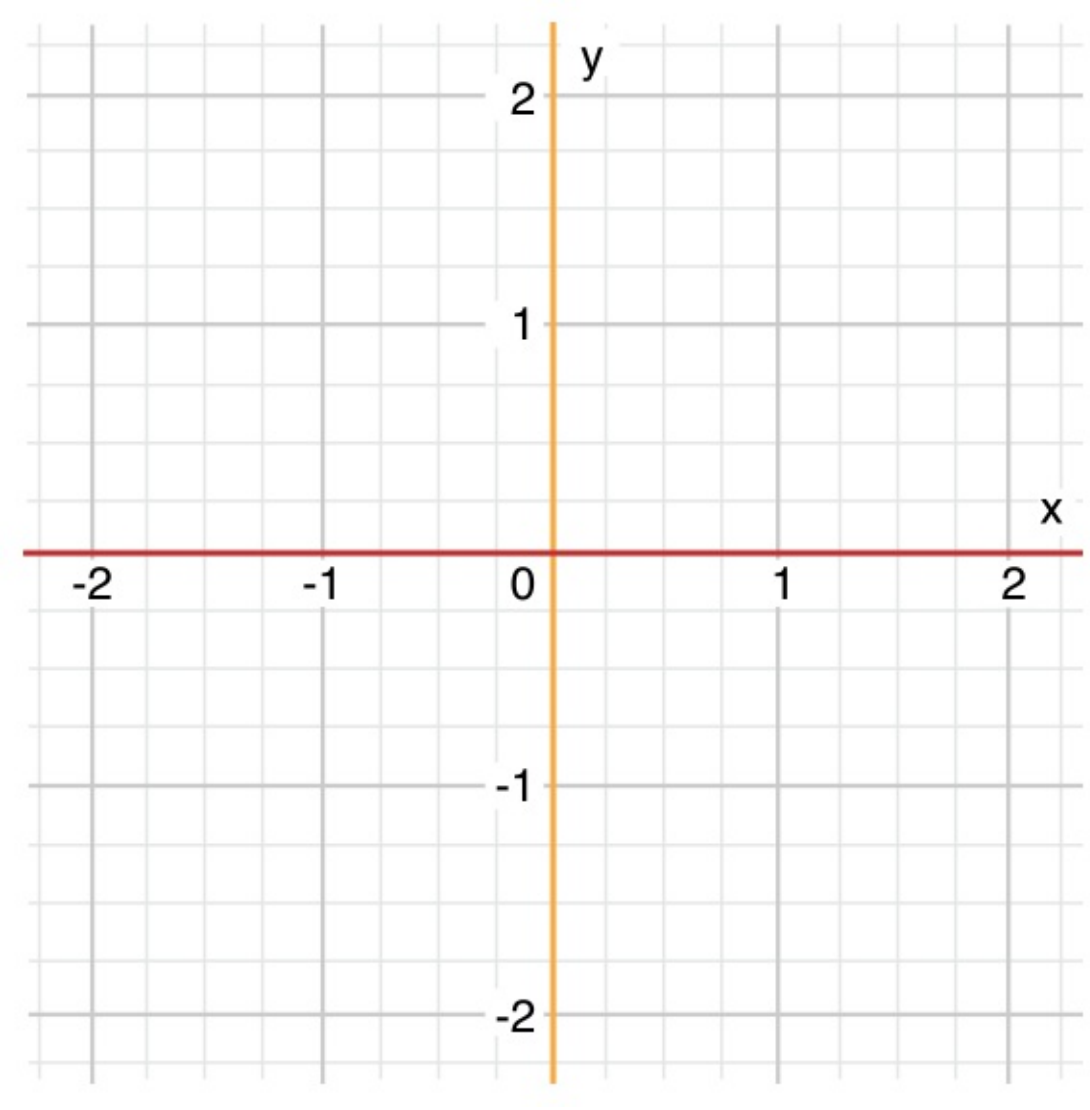
不像 C 語言，Swift 允許多個 case 匹配同一個值。實際上，在這個範例中，點(0, 0)可以匹配所有四個 *case*。但是，如果存在多個匹配，那麼只會執行第一個被匹配到的 case 分支。考慮點(0, 0)會首先匹配 `case (0, 0)`，因此剩下的能夠匹配(0, 0)的 case 分支都會被忽視掉。

值綁定（Value Bindings）

case 分支的模式允許將匹配的值綁定到一個臨時的常數或變數，這些常數或變數在該 case 分支裡就可以被參考了——這種行為被稱為值綁定（value binding）。

下面的範例展示了如何在一個 `(Int, Int)` 型別的元組中使用值綁定來分類別下圖中的點(x, y)：

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



在上面的範例中， `switch` 語句會判斷某個點是否在紅色的x軸上，是否在黃色y軸上，或者不在坐標軸上。

這三個 case 都宣告了常數 `x` 和 `y` 的占位符，用於臨時獲取元組 `anotherPoint` 的一個或兩個值。第一個 case —— `case (let x, 0)` 將匹配一個縱坐標為 `0` 的點，並把這個點的橫坐標賦給臨時的常數 `x` 。類似的，第二個 case —— `case (0, let y)` 將匹配一個橫坐標為 `0` 的點，並把這個點的縱坐標賦給臨時的常數 `y` 。

一旦宣告了這些臨時的常數，它們就可以在其對應的 case 分支裡參考。在這個範例中，它們用於簡化 `println` 的書寫。

請注意，這個 `switch` 語句不包含預設分支。這是因為最後一個 case —— `case let(x, y)` 宣告了一個可以匹配余下所有值的元組。這使得 `switch` 語句已經完備了，因此不需要再書寫預設分支。

在上面的範例中， `x` 和 `y` 是常數，這是因為沒有必要在其對應的 case 分支中修改它們的值。然而，它們也可以是變數——程式將會創建臨時變數，並用相應的值初始化它。修改這些變數只會影響其對應的 case 分支。

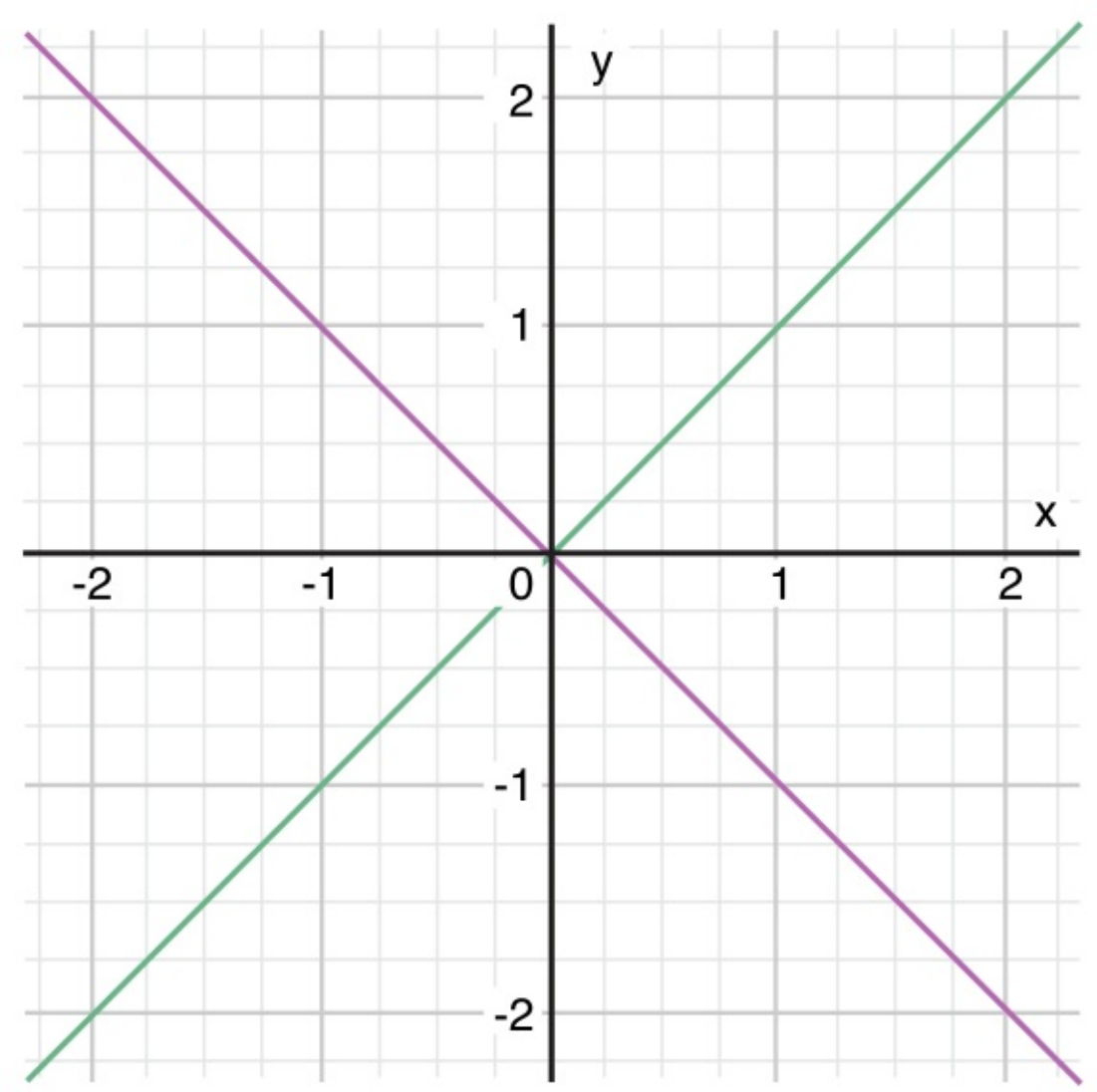
Where

case 分支的模式可以使用 `where` 語句來判斷額外的條件。

下面的範例把下圖中的點(x, y)進行了分類別：

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



在上面的範例中，`switch` 語句會判斷某個點是否在綠色的對角線 `x == y` 上，是否在紫色的對角線 `x == -y` 上，或者不在對角線上。

這三個 case 都宣告了常數 `x` 和 `y` 的占位符，用於臨時獲取元組 `yetAnotherPoint` 的兩個值。這些常數被用作 `where` 語句的一部分，從而創建一個動態的過濾器(filter)。當且僅當 `where` 語句的條件為 `true` 時，匹配到的 case 分支才會被執行。

就像是值綁定中的範例，由於最後一個 case 分支匹配了余下所有可能的值，`switch` 語句就已經完備了，因此不需要再書寫預設分支。

控制轉移語句（Control Transfer Statements）

控制轉移語句改變你程式碼的執行順序，通過它你可以實作程式碼的跳轉。Swift有四種控制轉移語句。

- continue
- break

- fallthrough
- return

我們將會在下面討論 `continue`、 `break` 和 `fallthrough` 語句。 `return` 語句將會在函式章節討論。

Continue

`continue` 語句告訴一個迴圈立刻停止本次迴圈迭代，重新開始下次迴圈迭代。就好像在說「本次迴圈迭代我已經執行完了」，但是並不會離開整個迴圈。

> 注意：
> 在一個for條件遞增（ `for-condition-increment` ）迴圈中，在呼叫 `continue` 語句後，迭代增量仍然會被計算求值。迴圈繼續像往常一樣工作，僅僅只是迴圈中的執行程式碼會被跳過。

下面的範例把一個小寫字串中的母音字母和空格字元移除，生成了一個含義模糊的短句：

```
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)
    // 輸出 "grtmndsthnklk"
```

在上面的程式碼中，只要匹配到母音字母或者空格字元，就呼叫 `continue` 語句，使本次迴圈迭代結束，從新開始下次迴圈迭代。這種行為使 `switch` 匹配到母音字母和空格字元時不做處理，而不是讓每一個匹配到的字元都被列印。

Break

`break` 語句會立刻結束整個控制流程的執行。當你想要更早的結束一個 `switch` 程式碼區塊或者一個迴圈時，你都可以使用 `break` 語句。

迴圈語句中的 break

當在一個迴圈中使用 `break` 時，會立刻中斷該迴圈的執行，然後跳轉到表示迴圈結束的大括號(`}`)後的第一行程式碼。不會再有本次迴圈迭代的程式碼被執行，也不會再有下次的迴圈迭代產生。

Switch 語句中的 break

當在一個 `switch` 程式碼區塊中使用 `break` 時，會立即中斷該 `switch` 程式碼區塊的執行，並且跳轉到表示 `switch` 程式碼區塊結束的大括號(`}`)後的第一行程式碼。

這種特性可以被用來匹配或者忽略一個或多個分支。因為 Swift 的 `switch` 需要包含所有的分支而且不允許有為空的分支，有時為了使你的意圖更明顯，需要特意匹配或者忽略某個分支。那麼當你想忽略某個分支時，可以在該分支內寫上 `break` 語句。當那個分支被匹配到時，分支內的 `break` 語句立即結束 `switch` 程式碼區塊。

> 注意：
> 當一個 `switch` 分支僅僅包含注釋時，會被報編譯時錯誤。注釋不是程式碼語句而且也不能讓 `switch` 分支達到被忽略的效果。你總是可以使用 `break` 來忽略某個分支。

下面的範例通過 `switch` 來判斷一個 `Character` 值是否代表下面四種語言之一。為了簡潔，多個值被包含在了同一個分支情況

中。

```
let numberSymbol: Character = "三"  // 簡體中文裡的數字 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 {
    println("The integer value of \(numberSymbol) is \(integerValue).")
} else {
    println("An integer value could not be found for \(numberSymbol).")
}
// 輸出 "The integer value of 三 is 3."
```

這個範例檢查 `numberSymbol` 是否是拉丁，阿拉伯，中文或者泰語中的 `1` 到 `4` 之一。如果被匹配到，該 `switch` 分支語句給 `Int?` 型別變數 `possibleIntegerValue` 設置一個整數值。

當 `switch` 程式碼區塊執行完後，接下來的程式碼通過使用可選綁定來判斷 `possibleIntegerValue` 是否曾經被設置過值。因為是可選型別的緣故， `possibleIntegerValue` 有一個隱式的初始值 `nil` ，所以僅僅當 `possibleIntegerValue` 曾被 `switch` 程式碼區塊的前四個分支中的某個設置過一個值時，可選的綁定將會被判定為成功。

在上面的範例中，想要把 `Character` 所有的的可能性都列舉出來是不現實的，所以使用 `default` 分支來包含所有上面沒有匹配到字元的情況。由於這個 `default` 分支不需要執行任何動作，所以它只寫了一條 `break` 語句。一旦落入到 `default` 分支中後， `break` 語句就完成了該分支的所有程式碼操作，程式碼繼續向下，開始執行 `if let` 語句。

貫穿（Fallthrough）

Swift 中的 `switch` 不會從上一個 case 分支落入到下一個 case 分支中。相反，只要第一個匹配到的 case 分支完成了它需要執行的語句，整個 `switch` 程式碼區塊完成了它的執行。相比之下，C 語言要求你顯示的插入 `break` 語句到每個 `switch` 分支的末尾來阻止自動落入到下一個 case 分支中。Swift 的這種避免預設落入到下一個分支中的特性意味著它的 `switch` 功能要比 C 語言的更加清晰和可預測，可以避免無意識地執行多個 case 分支從而引發的錯誤。

如果你確實需要 C 風格的貫穿（fallthrough）的特性，你可以在每個需要該特性的 case 分支中使用 `fallthrough` 關鍵字。下面的範例使用 `fallthrough` 來創建一個數字的描述語句。

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

這個範例定義了一個 `String` 型別的變數 `description` 並且給它設置了一個初始值。函式使用 `switch` 邏輯來判斷 `integerToDescribe` 變數的值。當 `integerToDescribe` 的值屬於列表中的質數之一時，該函式添加一段文字在 `description` 後，來表明這個是數字是一個質數。然後它使用 `fallthrough` 關鍵字來「貫穿」到 `default` 分支中。 `default` 分支添加一段額外的文字在 `description` 的最後，至此 `switch` 程式碼區塊執行完了。

如果 `integerToDescribe` 的值不屬於列表中的任何質數，那麼它不會匹配到第一個 `switch` 分支。而這裡沒有其他特別的分支情況，所以 `integerToDescribe` 匹配到包含所有的 `default` 分支中。

當 `switch` 程式碼區塊執行完後，使用 `println` 函式列印該數字的描述。在這個範例中，數字 `5` 被準確的識別為了一個質數。

> 注意：
> `fallthrough` 關鍵字不會檢查它下一個將會落入執行的 case 中的匹配條件。 `fallthrough` 簡單地使程式碼執行繼續連接到下一個 case 中的執行程式碼，這和 C 語言標準中的 `switch` 語句特性是一樣的。

帶標簽的語句（Labeled Statements）

在 Swift 中，你可以在迴圈和 `switch` 程式碼區塊中嵌套迴圈和 `switch` 程式碼區塊來創造複雜的控制流程結構。然而，迴圈和 `switch` 程式碼區塊兩者都可以使用 `break` 語句來提前結束整個方法體。因此，顯示地指明 `break` 語句想要終止的是哪個迴圈或者 `switch` 程式碼區塊，會很有用。類似地，如果你有許多嵌套的迴圈，顯示指明 `continue` 語句想要影響哪一個迴圈也會非常有用。

為了實作這個目的，你可以使用標簽來標記一個迴圈或者 `switch` 程式碼區塊，當使用 `break` 或者 `continue` 時，帶上這個標簽，可以控制該標簽代表物件的中斷或者執行。

產生一個帶標簽的語句是通過在該語句的關鍵詞的同一行前面放置一個標簽，並且該標簽後面還需帶著一個冒號。下面是一個 `while` 迴圈的語法，同樣的規則適用於所有的迴圈和 `switch` 程式碼區塊。

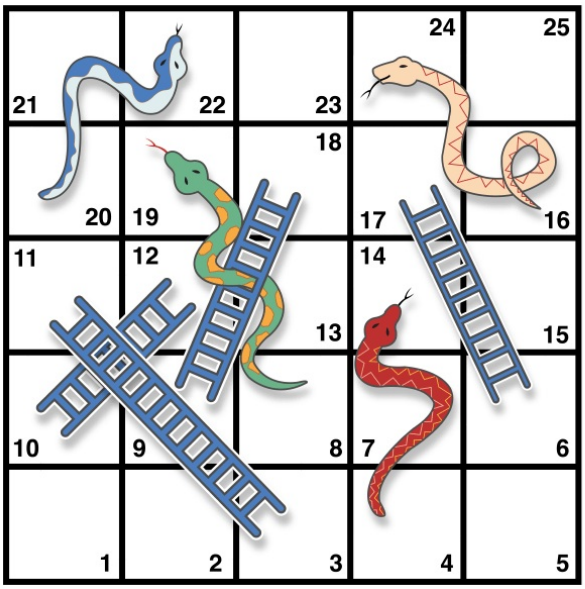
```
label name: while condition {
statements
}
```

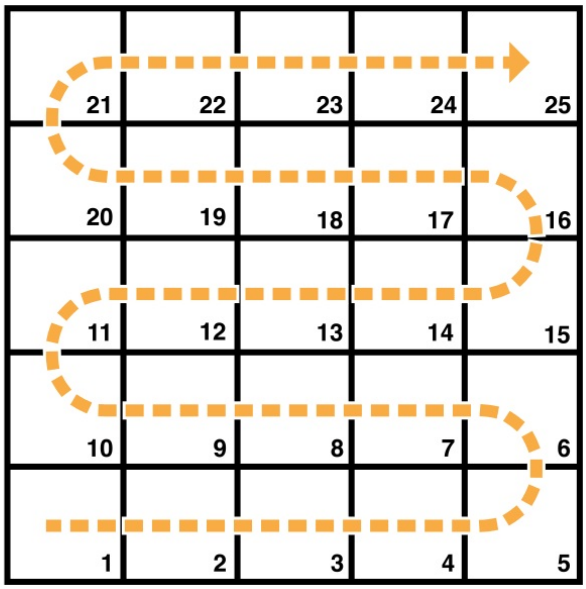
下面的範例是在一個帶有標簽的 `while` 迴圈中呼叫 `break` 和 `continue` 語句，該迴圈是前面章節中蛇和梯子的改編版本。這次，遊戲增加了一條額外的規則：

- 為了獲勝，你必須剛好落在第 25 個方塊中。

如果某次擲骰子使你的移動超出第 25 個方塊，你必須重新擲骰子，直到你擲出的骰子數剛好使你能落在第 25 個方塊中。

遊戲的棋盤和之前一樣：





值 `finalSquare`、 `board`、 `square` 和 `diceRoll` 的初始化也和之前一樣：

```
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
```

這個版本的遊戲使用 `while` 迴圈和 `switch` 方法塊來實作遊戲的邏輯。 `while` 迴圈有一個標簽名 `gameLoop`，來表明它是蛇與梯子的主迴圈。

該 `while` 迴圈的條件判斷語句是 `while square !=finalSquare`，這表明你必須剛好落在方格25中。

```
gameLoop: while square != finalSquare {
    if ++diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        // 到達最後一個方塊，遊戲結束
        break gameLoop
    case let newSquare where newSquare > finalSquare:
        // 超出最後一個方塊，再擲一次骰子
        continue gameLoop
    default:
        // 本次移動有效
        square += diceRoll
        square += board[square]
    }
}
println("Game over!")
```

每次迴圈迭代開始時擲骰子。與之前玩家擲完骰子就立即移動不同，這裡使用了 `switch` 來考慮每次移動可能產生的結果，從而決定玩家本次是否能夠移動。

- 如果骰子數剛好使玩家移動到最終的方格裡，遊戲結束。 `break gameLoop` 語句跳轉控制去執行 `while` 迴圈後的第一行程式碼，遊戲結束。
- 如果骰子數將會使玩家的移動超出最後的方格，那麼這種移動是不合法的，玩家需要重新擲骰子。 `continue gameLoop` 語句結束本次 `while` 迴圈的迭代，開始下一次迴圈迭代。
- 在剩余的所有情況中，骰子數產生的都是合法的移動。玩家向前移動骰子數個方格，然後遊戲邏輯再處理玩家當前是否處於蛇頭或者梯子的底部。本次迴圈迭代結束，控制跳轉到 `while` 迴圈的條件判斷語句處，再決定是否能夠繼續執行下次迴圈迭代。

> 注意：
> 如果上述的 `break` 語句沒有使用 `gameLoop` 標簽，那麼它將會中斷 `switch` 程式碼區塊而不是 `while` 迴圈。使用 `gameLoop` 標簽清晰的表明了 `break` 想要中斷的是哪個程式碼區塊。 同時請注意，當呼叫 `continue gameLoop` 去跳轉到下一次迴圈迭代時，這裡使用 `gameLoop` 標簽並不是嚴格必須的。因為在這個遊戲中，只有一個迴圈，所以 `continue` 語句會影響到哪個迴圈是沒有歧義的。然而， `continue` 語句使用 `gameLoop` 標簽也是沒有危害的。這樣做符合標簽的使用規則，同時參照旁邊的 `break gameLoop`，能夠使遊戲的邏輯更加清晰和易於理解。

> 翻譯：honghaoz 校對：LunaticM

函式（Functions）

本頁包含內容：

函式是用來完成特定任務的獨立的程式碼區塊。你給一個函式起一個合適的名字，用來標示函式做什麼，並且當函式需要執行的時候，這個名字會被「呼叫」。

Swift 統一的函式語法足夠靈活，可以用來表示任何函式，包括從最簡單的沒有參數名字的 C 風格函式，到複雜的帶局部和外部參數名的 Objective-C 風格函式。參數可以提供預設值，以簡化函式呼叫。參數也可以既當做傳入參數，也當做傳出參數，也就是說，一旦函式執行結束，傳入的參數值可以被修改。

在 Swift 中，每個函式都有一種型別，包括函式的參數值型別和回傳值型別。你可以把函式型別當做任何其他普通變數型別一樣處理，這樣就可以更簡單地把函式當做別的函式的參數，也可以從其他函式中回傳函式。函式的定義可以寫在在其他函式定義中，這樣可以在嵌套函式範圍內實作功能封裝。

函式的定義與呼叫（Defining and Calling Functions）

當你定義一個函式時，你可以定義一個或多個有名字和型別的值，作為函式的輸入（稱為參數，parameters），也可以定義某種型別的值作為函式執行結束的輸出（稱為回傳型別）。

每個函式有個函式名，用來描述函式執行的任務。要使用一個函式時，你用函式名「呼叫」，並傳給它匹配的輸入值（稱作實參，arguments）。一個函式的實參必須與函式參數表裡參數的順序一致。

在下面範例中的函式叫做 `"greetingForPerson"`，之所以叫這個名字是因為這個函式用一個人的名字當做輸入，並回傳給這個人的問候語。為了完成這個任務，你定義一個輸入參數-一個叫做 `personName` 的 `String` 值，和一個包含給這個人問候語的 `String` 型別的回傳值：

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

所有的這些資訊彙總起來成為函式的定義，並以 `func` 作為前綴。指定函式回傳型別時，用回傳箭頭 `->`（一個連字元後跟一個右角括號）後跟回傳型別的名稱的方式來表示。

該定義描述了函式做什麼，它期望接收什麼和執行結束時它回傳的結果是什麼。這樣的定義使的函式可以在別的地方以一種清晰的方式被呼叫：

```
println(sayHello("Anna"))
// prints "Hello, Anna!"
println(sayHello("Brian"))
// prints "Hello, Brian!
```

呼叫 `sayHello` 函式時，在圓括號中傳給它一個 `String` 型別的實參。因為這個函式回傳一個 `String` 型別的值， `sayHello` 可以被包含在 `println` 的呼叫中，用來輸出這個函式的回傳值，正如上面所示。

在 `sayHello` 的函式體中，先定義了一個新的名為 `greeting` 的 `String` 常數，同時賦值了給 `personName` 的一個簡單問候訊息。然後用 `return` 關鍵字把這個問候回傳出去。一旦 `return greeting` 被呼叫，該函式結束它的執行並回傳 `greeting` 的當前值。

你可以用不同的輸入值多次呼叫 `sayHello` 。上面的範例展示的是用 `"Anna"` 和 `"Brian"` 呼叫的結果，該函式分別回傳了不同的結果。

為了簡化這個函式的定義，可以將問候訊息的創建和回傳寫成一句：

```
func sayHelloAgain(personName: String) -> String {
    return "Hello again, " + personName + "!"
}
println(sayHelloAgain("Anna"))
// prints "Hello again, Anna!
```

函式參數與回傳值（Function Parameters and Return Values）

函式參數與回傳值在Swift中極為靈活。你可以定義任何型別的函式，包括從只帶一個未名參數的簡單函式到複雜的帶有表達性參數名和不同參數選項的複雜函式。

多重輸入參數（Multiple Input Parameters）

函式可以有多個輸入參數，寫在圓括號中，用逗號分隔。

下面這個函式用一個半開區間的開始點和結束點，計算出這個範圍內包含多少數字：

```
func halfOpenRangeLength(start: Int, end: Int) -> Int {
    return end - start
}
println(halfOpenRangeLength(1, 10))
// prints "9
```

無參函式（Functions Without Parameters）

函式可以沒有參數。下面這個函式就是一個無參函式，當被呼叫時，它回傳固定的 `String` 訊息：

```
func sayHelloWorld() -> String {
    return "hello, world"
}
println(sayHelloWorld())
// prints "hello, world
```

儘管這個函式沒有參數，但是定義中在函式名後還是需要一對圓括號。當被呼叫時，也需要在函式名後寫一對圓括號。

無回傳值函式（Functions Without Return Values）

函式可以沒有回傳值。下面是 `sayHello` 函式的另一個版本，叫 `waveGoodbye` ，這個函式直接輸出 `String` 值，而不是回傳它：

```
func sayGoodbye(personName: String) {
```

```
    println("Goodbye, \(personName)!")
}
sayGoodbye("Dave")
// prints "Goodbye, Dave!
```

因為這個函式不需要回傳值，所以這個函式的定義中沒有回傳箭頭（->）和回傳型別。

> 注意：
> 嚴格上來說，雖然沒有回傳值被定義，`sayGoodbye` 函式依然回傳了值。沒有定義回傳型別的函式會回傳特殊的值，叫 `Void`。它其實是一個空的元組（tuple），沒有任何元素，可以寫成 `()`。

被呼叫時，一個函式的回傳值可以被忽略：

```
func printAndCount(stringToPrint: String) -> Int {
    println(stringToPrint)
    return countElements(stringToPrint)
}
func printWithoutCounting(stringToPrint: String) {
    printAndCount(stringToPrint)
}
printAndCount("hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting("hello, world")
// prints "hello, world" but does not return a value
```

第一個函式 `printAndCount`，輸出一個字串並回傳 `Int` 型別的字元數。第二個函式 `printWithoutCounting` 呼叫了第一個函式，但是忽略了它的回傳值。當第二個函式被呼叫時，訊息依然會由第一個函式輸出，但是回傳值不會被用到。

> 注意：
> 回傳值可以被忽略，但定義了有回傳值的函式必須回傳一個值，如果在函式定義底部沒有回傳任何值，這叫導致編譯錯誤（compile-time error）。

多重回傳值函式（Functions with Multiple Return Values）

你可以用元組（tuple）型別讓多個值作為一個複合值從函式中回傳。

下面的這個範例中，`count` 函式用來計算一個字串中母音，子音和其他字母的個數（基於美式英語的標準）。

```
func count(string: String) -> (vowels: Int, consonants: Int, others: Int) {
    var vowels = 0, consonants = 0, others = 0
    for character in string {
        switch String(character).lowercaseString {
        case "a", "e", "i", "o", "u":
            ++vowels
        case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
          "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
            ++consonants
        default:
            ++others
        }
    }
    return (vowels, consonants, others)
}
```

你可以用 `count` 函式來處理任何一個字串，回傳的值將是一個包含三個 `Int` 型值的元組（tuple）：

```
let total = count("some arbitrary string!")
println("\(total.vowels) vowels and \(total.consonants) consonants")
// prints "6 vowels and 13 consonants
```

需要注意的是，元組的成員不需要在函式中回傳時命名，因為它們的名字已經在函式回傳型別有有了定義。

函式參數名稱（Function Parameter Names）

以上所有的函式都給它們的參數定義了 `參數名（parameter name）`：

```
func someFunction(parameterName: Int) {
    // function body goes here, and can use parameterName
    // to refer to the argument value for that parameter
}
```

但是，這些參數名僅在函式體中使用，不能在函式呼叫時使用。這種型別的參數名被稱作 `局部參數名（local parameter name）`，因為它們只能在函式體中使用。

外部參數名（External Parameter Names）

有時候，呼叫函式時，給每個參數命名是非常有用的，因為這些參數名可以指出各個實參的用途是什麼。

如果你希望函式的使用者在呼叫函式時提供參數名字，那就需要給每個參數除了局部參數名外再定義一個 `外部參數名`。外部參數名寫在局部參數名之前，用空格分隔。

```
func someFunction(externalParameterName localParameterName: Int) {
    // function body goes here, and can use localParameterName
    // to refer to the argument value for that parameter
}
```

> 注意：
> 如果你提供了外部參數名，那麼函式在被呼叫時，必須使用外部參數名。

以下是個範例，這個函式使用一個 `結合者（joiner）` 把兩個字串聯在一起：

```
func join(s1: String, s2: String, joiner: String) -> String {
    return s1 + joiner + s2
}
```

當你呼叫這個函式時，這三個字串的用途是不清楚的：

```
join("hello", "world", ", ")
// returns "hello, world
```

為了讓這些字串的用途更為明顯，我們為 `join` 函式添加外部參數名：

```
func join(string s1: String, toString s2: String, withJoiner joiner: String) -> String {
    return s1 + joiner + s2
}
```

在這個版本的 `join` 函式中，第一個參數有一個叫 `string` 的外部參數名和 `s1` 的局部參數名，第二個參數有一個叫 `toString` 的外部參數名和 `s2` 的局部參數名，第三個參數有一個叫 `withJoiner` 的外部參數名和 `joiner` 的局部參數名。

現在，你可以使用這些外部參數名以一種清晰地方式來呼叫函式了：

```
join(string: "hello", toString: "world", withJoiner: ", ")
// returns "hello, world
```

使用外部參數名讓第二個版本的 `join` 函式的呼叫更為有表現力，更為通順，同時還保持了函式體是可讀的和有明確意圖的。

> 注意：
> 當其他人在第一次讀你的程式碼，函式參數的意圖顯得不明顯時，考慮使用外部參數名。如果函式參數名的意圖是很明顯的，那就不需要定義外部參數名了。

簡寫外部參數名（Shorthand External Parameter Names）

如果你需要提供外部參數名，但是局部參數名已經定義好了，那麼你不需要寫兩次這些參數名。相反，只寫一次參數名，並用 `井字號（#）` 作為前綴就可以了。這告訴 Swift 使用這個參數名作為局部和外部參數名。

下面這個範例定義了一個叫 `containsCharacter` 的函式，使用 `井字號（#）` 的方式定義了外部參數名：

```
func containsCharacter(#string: String, #characterToFind: Character) -> Bool {
    for character in string {
        if character == characterToFind {
            return true
        }
    }
    return false
}
```

這樣定義參數名，使得函式體更為可讀，清晰，同時也可以以一個不含糊的方式被呼叫：

```
let containsAVee = containsCharacter(string: "aardvark", characterToFind: "v")
// containsAVee equals true, because "aardvark" contains a "v」
```

預設參數值（Default Parameter Values）

你可以在函式體中為每個參數定義 `預設值` 。當預設值被定義後，呼叫這個函式時可以略去這個參數。

> 注意：
> 將帶有預設值的參數放在函式參數表的最後。這樣可以保證在函式呼叫時，非預設參數的順序是一致的，同時使得相同的函式在不同情況下呼叫時顯得更為清晰。

以下是另一個版本的 `join` 函式，其中 `joiner` 有了預設參數值：

```
func join(string s1: String, toString s2: String, withJoiner joiner: String = " ") -> String {
    return s1 + joiner + s2
}
```

像第一個版本的 `join` 函式一樣，如果 `joiner` 被賦值時，函式將使用這個字串值來連接兩個字串：

```
join(string: "hello", toString: "world", withJoiner: "-")
// returns "hello-world
```

當這個函式被呼叫時，如果 `joiner` 的值沒有被指定，函式會使用預設值（" "）：

```
join(string: "hello", toString:"world")
// returns "hello world"
```

預設值參數的外部參數名（External Names for Parameters with Default Values）

在大多數情況下，給帶預設值的參數起一個外部參數名是很有用的。這樣可以保證當函式被呼叫且帶預設值的參數被提供值時，實參的意圖是明顯的。

為了使定義外部參數名更加簡單，當你未給帶預設值的參數提供外部參數名時，Swift 會自動提供外部名字。此時外部參數名與局部名字是一樣的，就像你已經在局部參數名前寫了`井字號（#）`一樣。

下面是 `join` 函式的另一個版本，這個版本中並沒有為它的參數提供外部參數名，但是 `joiner` 參數依然有外部參數名：

```
func join(s1: String, s2: String, joiner: String = " ") -> String {
    return s1 + joiner + s2
}
```

在這個範例中，Swift 自動為 `joiner` 提供了外部參數名。因此，當函式呼叫時，外部參數名必須使用，這樣使得參數的用途變得清晰。

```
join("hello", "world", joiner: "-")
// returns "hello-world"
```

> 注意：
> 你可以使用`底線（_）`作為預設值參數的外部參數名，這樣可以在呼叫時不用提供外部參數名。但是給帶預設值的參數命名總是更加合適的。

可變參數（Variadic Parameters）

一個`可變參數（variadic parameter）`可以接受一個或多個值。函式呼叫時，你可以用可變參數來傳入不確定數量的輸入參數。通過在變數型別名後面加入`（...）`的方式來定義可變參數。

傳入可變參數的值在函式體內當做這個型別的一個陣列。例如，一個叫做 `numbers` 的 `Double...` 型可變參數，在函式體內可以當做一個叫 `numbers` 的 `Double[]` 型的陣列常數。

下面的這個函式用來計算一組任意長度數字的算術平均數：

```
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)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8, 19)
// returns 10.0, which is the arithmetic mean of these three numbers
```

> 注意：
> 一個函式至多能有一個可變參數，而且它必須是參數表中最後的一個。這樣做是為了避免函式呼叫時出現歧義。

如果函式有一個或多個帶預設值的參數，而且還有一個可變參數，那麼把可變參數放在參數表的最後。

常數參數和變數參數（Constant and Variable Parameters）

函式參數預設是常數。試圖在函式體中更改參數值將會導致編譯錯誤。這意味著你不能錯誤地更改參數值。

但是，有時候，如果函式中有傳入參數的變數值副本將是很有用的。你可以通過指定一個或多個參數為變數參數，從而避免自己在函式中定義新的變數。變數參數不是常數，你可以在函式中把它當做新的可修改副本來使用。

通過在參數名前加關鍵字 `var` 來定義變數參數：

```
func alignRight(var string: String, count: Int, pad: Character) -> String {
    let amountToPad = count - countElements(string)
    for _ in 1...amountToPad {
        string = pad + string
    }
    return string
}
let originalString = "hello"
let paddedString = alignRight(originalString, 10, "-")
// paddedString is equal to "-----hello"
// originalString is still equal to "hello"
```

這個範例中定義了一個新的叫做 `alignRight` 的函式，用來右對齊輸入的字串到一個長的輸出字串中。左側空余的地方用指定的填充字元填充。這個範例中，字串 `"hello"` 被轉換成了 `"-----hello"`。

`alignRight` 函式將參數 `string` 定義為變數參數。這意味著 `string` 現在可以作為一個局部變數，用傳入的字串值初始化，並且可以在函式體中進行操作。

該函式首先計算出多少個字元需要被添加到 `string` 的左邊，以右對齊到總的字串中。這個值存在局部常數 `amountToPad` 中。這個函式然後將 `amountToPad` 多的填充（pad）字元填充到 `string` 左邊，並回傳結果。它使用了 `string` 這個變數參數來進行所有字串操作。

> 注意：
> 對變數參數所進行的修改在函式呼叫結束後便消失了，並且對於函式體外是不可見的。變數參數僅僅存在於函式呼叫的生命周期中。

輸入輸出參數（In-Out Parameters）

變數參數，正如上面所述，僅僅能在函式體內被更改。如果你想要一個函式可以修改參數的值，並且想要在這些修改在函式呼叫結束後仍然存在，那麼就應該把這個參數定義為輸入輸出參數（In-Out Parameters）。

定義一個輸入輸出參數時，在參數定義前加 `inout` 關鍵字。一個輸入輸出參數有傳入函式的值，這個值被函式修改，然後被傳出函式，替換原來的值。

你只能傳入一個變數作為輸入輸出參數。你不能傳入常數或者字面量（literal value），因為這些量是不能被修改的。當傳入的參數作為輸入輸出參數時，需要在參數前加 `&` 符，表示這個值可以被函式修改。

> 注意：
> 輸入輸出參數不能有預設值，而且可變參數不能用 `inout` 標記。如果你用 `inout` 標記一個參數，這個參數不能被 `var` 或者 `let` 標記。

下面是範例，`swapTwoInts` 函式，有兩個分別叫做 `a` 和 `b` 的輸出輸出參數：

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

這個 `swapTwoInts` 函式僅僅交換 `a` 與 `b` 的值。該函式先將 `a` 的值存到一個暫時常數 `temporaryA` 中，然後將 `b` 的值賦給 `a`，最後將 `temporaryA` 幅值給 `b`。

你可以用兩個 `Int` 型的變數來呼叫 `swapTwoInts`。需要注意的是，`someInt` 和 `anotherInt` 在傳入 `swapTwoInts` 函式前，都加了 `&` 的前綴：

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// prints "someInt is now 107, and anotherInt is now 3」
```

從上面這個範例中，我們可以看到 `someInt` 和 `anotherInt` 的原始值在 `swapTwoInts` 函式中被修改，儘管它們的定義在函式體外。

> 注意：
> 輸出輸出參數和回傳值是不一樣的。上面的 `swapTwoInts` 函式並沒有定義任何回傳值，但仍然修改了 `someInt` 和 `anotherInt` 的值。輸入輸出參數是函式對函式體外產生影響的另一種方式。

函式型別（Function Types）

每個函式都有種特定的函式型別，由函式的參數型別和回傳型別組成。

例如：

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

這個範例中定義了兩個簡單的數學函式：`addTwoInts` 和 `multiplyTwoInts`。這兩個函式都傳入兩個 `Int` 型別，回傳一個合適的 `Int` 值。

這兩個函式的型別是 `(Int, Int) -> Int`，可以讀作「這個函式型別，它有兩個 `Int` 型的參數並回傳一個 `Int` 型的值。」。

下面是另一個範例，一個沒有參數，也沒有回傳值的函式：

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

這個函式的型別是：`() -> ()`，或者叫「沒有參數，並回傳 `Void` 型別的函式。」。沒有指定回傳型別的函式總回傳 `Void`。在Swift中，`Void` 與空的元組是一樣的。

使用函式型別（Using Function Types）

在 Swift 中，使用函式型別就像使用其他型別一樣。例如，你可以定義一個型別為函式的常數或變數，並將函式賦值給它：

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

這個可以讀作：

「定義一個叫做 `mathFunction` 的變數，型別是『一個有兩個 `Int` 型的參數並回傳一個 `Int` 型的值的函式』，並讓這個新變數指向 `addTwoInts` 函式」。

`addTwoInts` 和 `mathFunction` 有同樣的型別，所以這個賦值過程在 Swift 型別檢查中是允許的。

現在，你可以用 `mathFunction` 來呼叫被賦值的函式了：

```
println("Result: \(mathFunction(2, 3))")
// prints "Result: 5
```

有相同匹配型別的不同函式可以被賦值給同一個變數，就像非函式型別的變數一樣：

```
mathFunction = multiplyTwoInts
println("Result: \(mathFunction(2, 3))")
// prints "Result: 6"
```

就像其他型別一樣，當賦值一個函式給常數或變數時，你可以讓 Swift 來推斷其函式型別：

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

函式型別作為參數型別（Function Types as Parameter Types）

你可以用 `(Int, Int) -> Int` 這樣的函式型別作為另一個函式的參數型別。這樣你可以將函式的一部分實作交由給函式的呼叫者。

下面是另一個範例，正如上面的函式一樣，同樣是輸出某種數學運算結果：

```
func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {
    println("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// prints "Result: 8」
```

這個範例定義了 `printMathResult` 函式，它有三個參數：第一個參數叫 `mathFunction`，型別是 `(Int, Int) -> Int`，你可以傳入任何這種型別的函式；第二個和第三個參數叫 `a` 和 `b`，它們的型別都是 `Int`，這兩個值作為已給的函式的輸入值。

當 `printMathResult` 被呼叫時，它被傳入 `addTwoInts` 函式和整數 `3` 和 `5`。它用傳入 `3` 和 `5` 呼叫 `addTwoInts`，並輸出結果：`8`。

`printMathResult` 函式的作用就是輸出另一個合適型別的數學函式的呼叫結果。它不關心傳入函式是如何實作的，它只關心這個傳入的函式型別是正確的。這使得 `printMathResult` 可以以一種型別安全（type-safe）的方式來保證傳入函式的呼叫是正確的。

函式型別作為回傳型別（Function Type as Return Types）

你可以用函式型別作為另一個函式的回傳型別。你需要做的是在回傳箭頭（`->`）後寫一個完整的函式型別。

下面的這個範例中定義了兩個簡單函式，分別是 `stepForward` 和 `stepBackward`。`stepForward` 函式回傳一個比輸入值大一的

值。 `stepBackward` 函式回傳一個比輸入值小一的值。這兩個函式的型別都是 `(Int) -> Int` ：

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

下面這個叫做 `chooseStepFunction` 的函式，它的回傳型別是 `(Int) -> Int` 的函式。 `chooseStepFunction` 根據布林值 `backwards` 來回傳 `stepForward` 函式或 `stepBackward` 函式：

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

你現在可以用 `chooseStepFunction` 來獲得一個函式，不管是那個方向：

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

上面這個範例中計算出從 `currentValue` 逐漸接近到 `0` 是需要向正數走還是向負數走。 `currentValue` 的初始值是 `3` ，這意味著 `currentValue > 0` 是真的（ `true` ），這將使得 `chooseStepFunction` 回傳 `stepBackward` 函式。一個指向回傳的函式的參考保存在了 `moveNearerToZero` 常數中。

現在， `moveNearerToZero` 指向了正確的函式，它可以被用來數到 `0` ：

```
println("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// 3...
// 2...
// 1...
// zero!
```

嵌套函式（Nested Functions）

這章中你所見到的所有函式都叫全域函式（global functions），它們定義在全域域中。你也可以把函式定義在別的函式體中，稱作嵌套函式（nested functions）。

預設情況下，嵌套函式是對外界不可見的，但是可以被他們封閉函式（enclosing function）來呼叫。一個封閉函式也可以回傳它的某一個嵌套函式，使得這個函式可以在其他域中被使用。

你可以用回傳嵌套函式的方式重寫 `chooseStepFunction` 函式：

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backwards ? stepBackward : stepForward
}
```

```
var currentValue = -4
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

> 翻譯：wh1100717 校對：lyuka, hcjao

閉包（Closures）

本頁包含內容：

- 閉包表達式（Closure Expressions）
- 尾隨閉包（Trailing Closures）
- 值捕獲（Capturing Values）
- 閉包是參考型別（Closures Are Reference Types）

閉包是自包含的函式程式碼區塊，可以在程式碼中被傳遞和使用。 Swift 中的閉包與 C 和 Objective-C 中的程式碼區塊（blocks）以及其他一些程式語言中的 lambdas 函式比較相似。

閉包可以捕獲和儲存其所在上下文中任意常數和變數的參考。 這就是所謂的閉合並包裹著這些常數和變數，俗稱閉包。Swift 會為你管理在捕獲過程中涉及到的所有內存操作。

> 注意： 如果你不熟悉捕獲（capturing）這個概念也不用擔心，你可以在 值捕獲 章節對其進行詳細了解。

在函式 章節中介紹的全域和嵌套函式實際上也是特殊的閉包，閉包采取如下三種形式之一：

- 全域函式是一個有名字但不會捕獲任何值的閉包
- 嵌套函式是一個有名字並可以捕獲其封閉函式域內值的閉包
- 閉包表達式是一個利用輕量級語法所寫的可以捕獲其上下文中變數或常數值的匿名閉包

Swift 的閉包表達式擁有簡潔的風格，並鼓勵在常見場景中進行語法優化，主要優化如下：

- 利用上下文推斷參數和回傳值型別
- 隱式回傳單表達式閉包，即單表達式閉包可以省略 `return` 關鍵字
- 參數名稱縮寫
- 尾隨（Trailing）閉包語法

閉包表達式（Closure Expressions）

嵌套函式 是一個在較複雜函式中方便進行命名和定義自包含程式碼模塊的方式。當然，有時候撰寫小巧的沒有完整定義和命名的類別函式結構也是很有用處的，尤其是在你處理一些函式並需要將另外一些函式作為該函式的參數時。

閉包表達式是一種利用簡潔語法構建內聯閉包的方式。 閉包表達式提供了一些語法優化，使得撰寫閉包變得簡單明了。 下面閉包表達式的範例通過使用幾次迭代展示了 `sort` 函式定義和語法優化的方式。 每一次迭代都用更簡潔的方式描述了相同的功能。

sorted 函式（The Sorted Function）

Swift 標準函式庫提供了 `sorted` 函式，會根據你提供的基於輸出型別排序的閉包函式將已知型別陣列中的值進行排序。 一旦排序完成，函式會回傳一個與原陣列大小相同的新陣列，該陣列中包含已經正確排序的同型別元素。

下面的閉包表達式示例使用 `sorted` 函式對一個 `String` 型別的陣列進行字母逆序排序，以下是初始陣列值：

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

`sorted` 函式需要傳入兩個參數：

- 已知型別的陣列
- 閉包函式，該閉包函式需要傳入與陣列型別相同的兩個值，並回傳一個布林型別值來告訴 `sort` 函式當排序結束後傳入的第一個參數排在第二個參數前面還是後面。如果第一個參數值出現在第二個參數值前面，排序閉包函式需要回傳 `true`，反之回傳 `false`。

該範例對一個 `String` 型別的陣列進行排序，因此排序閉包函式型別需為 `(String, String) -> Bool`。

提供排序閉包函式的一種方式是撰寫一個符合其型別要求的普通函式，並將其作為 `sort` 函式的第二個參數傳入：

```
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
var reversed = sorted(names, backwards)
// reversed 為 ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

如果第一個字串 (`s1`) 大於第二個字串 (`s2`)， `backwards` 函式回傳 `true`，表示在新的陣列中 `s1` 應該出現在 `s2` 前。 對於字串中的字元來說，「大於」 表示 「按照字母順序較晚出現」。 這意味著字母 `"B"` 大於字母 `"A"`，字串 `"Tom"` 大於字串 `"Tim"`。 其將進行字母逆序排序， `"Barry"` 將會排在 `"Alex"` 之後。

然而，這是一個相當冗長的方式，本質上只是寫了一個單表達式函式 (a > b)。 在下面的範例中，利用閉合表達式語法可以更好的建構一個內聯排序閉包。

閉包表達式語法（Closure Expression Syntax）

閉包表達式語法有如下一般形式：

```
{ (parameters) -> returnType in
    statements
}
```

閉包表達式語法可以使用常數、變數和 `inout` 型別作為參數，不提供預設值。 也可以在參數列表的最後使用可變參數。 元組也可以作為參數和回傳值。

下面的範例展示了之前 `backwards` 函式對應的閉包表達式版本的程式碼：

```
reversed = sort(names, { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

需要注意的是內聯閉包參數和回傳值型別宣告與 `backwards` 函式型別宣告相同。 在這兩種方式中，都寫成了 `(s1: String, s2: String) -> Bool`。 然而在內聯閉包表達式中，函式和回傳值型別都寫在大括號內，而不是大括號外。

閉包的函式體部分由關鍵字 `in` 引入。 該關鍵字表示閉包的參數和回傳值型別定義已經完成，閉包函式體即將開始。

因為這個閉包的函式體部分如此短以至於可以將其改寫成一行程式碼：

```
reversed = sort(names, { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

這說明 `sort` 函式的整體呼叫保持不變，一對圓括號仍然包裹住了函式中整個參數集合。而其中一個參數現在變成了內聯閉包（相比於 `backwards` 版本的程式碼）。

根據上下文推斷型別（Inferring Type From Context）

因為排序閉包函式是作為 `sort` 函式的參數進行傳入的，Swift可以推斷其參數和回傳值的型別。 `sort` 期望第二個參數是型別為 `(String, String) -> Bool` 的函式，因此實際上 `String`, `String` 和 `Bool` 型別並不需要作為閉包表達式定義中的一部分。 因為所有的型別都可以被正確推斷，回傳箭頭 (`->`) 和圍繞在參數周圍的括號也可以被省略：

```
reversed = sort(names, { s1, s2 in return s1 > s2 } )
```

實際上任何情況下，通過內聯閉包表達式建構的閉包作為參數傳遞給函式時，都可以推斷出閉包的參數和回傳值型別，這意味著你幾乎不需要利用完整格式建構任何內聯閉包。

單表達式閉包隱式回傳（Implicit Return From Single-Expression Clossures）

單行表達式閉包可以通過隱藏 `return` 關鍵字來隱式回傳單行表達式的結果，如上版本的範例可以改寫為：

```
reversed = sort(names, { s1, s2 in s1 > s2 } )
```

在這個範例中， `sort` 函式的第二個參數函式型別明確了閉包必須回傳一個 `Bool` 型別值。 因為閉包函式體只包含了一個單一表達式 (`s1 > s2`)，該表達式回傳 `Bool` 型別值，因此這裡沒有歧義， `return` 關鍵字可以省略。

參數名稱縮寫（Shorthand Argument Names）

Swift 自動為內聯函式提供了參數名稱縮寫功能，你可以直接通過 `$0`, `$1`, `$2` 來順序呼叫閉包的參數。

如果你在閉包表達式中使用參數名稱縮寫，你可以在閉包參數列表中省略對其的定義，並且對應參數名稱縮寫的型別會通過函式型別進行推斷。 `in` 關鍵字也同樣可以被省略，因為此時閉包表達式完全由閉包函式體構成：

```
reversed = sort(names, { $0 > $1 } )
```

在這個範例中， `$0` 和 `$1` 表示閉包中第一個和第二個 `String` 型別的參數。

運算子函式（Operator Functions）

實際上還有一種更簡短的方式來撰寫上面範例中的閉包表達式。 Swift 的 `String` 型別定義了關於大於號 (`>`) 的字串實作，其作為一個函式接受兩個 `String` 型別的參數並回傳 `Bool` 型別的值。 而這正好與 `sort` 函式的第二個參數需要的函式型別相符合。 因此，你可以簡單地傳遞一個大於號，Swift可以自動推斷出你想使用大於號的字串函式實作：

```
reversed = sort(names, >)
```

更多關於運算子表達式的內容請查看 運算子函式。

尾隨閉包（Trailing Closures）

如果你需要將一個很長的閉包表達式作為最後一個參數傳遞給函式，可以使用尾隨閉包來增強函式的可讀性。 尾隨閉包是一個書寫在函式括號之後的閉包表達式，函式支援將其作為最後一個參數呼叫。

```
func someFunctionThatTakesAClosure(closure: () -> ()) {
    // 函式體部分
}

// 以下是不使用尾隨閉包進行函式呼叫
```

```
someFunctionThatTakesAClosure({
    // 閉包主體部分
})

// 以下是使用尾隨閉包進行函式呼叫
someFunctionThatTakesAClosure() {
  // 閉包主體部分
}
```

> 注意： 如果函式只需要閉包表達式一個參數，當你使用尾隨閉包時，你甚至可以把 `()` 省略掉。

在上例中作為 `sort` 函式參數的字串排序閉包可以改寫為：

```
reversed = sort(names) { $0 > $1 }
```

當閉包非常長以至於不能在一行中進行書寫時，尾隨閉包變得非常有用。 舉例來說，Swift 的 `Array` 型別有一個 `map` 方法，其獲取一個閉包表達式作為其唯一參數。 陣列中的每一個元素呼叫一次該閉包函式，並回傳該元素所映射的值(也可以是不同型別的值)。 具體的映射方式和回傳值型別由閉包來指定。

當提供給陣列閉包函式後， `map` 方法將回傳一個新的陣列，陣列中包含了與原陣列一一對應的映射後的值。

下例介紹了如何在 `map` 方法中使用尾隨閉包將 `Int` 型別陣列 `[16,58,510]` 轉換為包含對應 `String` 型別的陣列 `["OneSix", "FiveEight", "FiveOneZero"]`：

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

如上程式碼創建了一個數字位和它們名字映射的英文版本字典。 同時定義了一個準備轉換為字串的整型陣列。

你現在可以通過傳遞一個尾隨閉包給 `numbers` 的 `map` 方法來創建對應的字串版本陣列。 需要注意的時呼叫 `numbers.map` 不需要在 `map` 後面包含任何括號，因為其只需要傳遞閉包表達式這一個參數，並且該閉包表達式參數通過尾隨方式進行撰寫：

```
let strings = numbers.map {
    (var number) -> String in
    var output = ""
    while number > 0 {
        output = digitNames[number % 10]! + output
        number /= 10
    }
    return output
}
// strings 常數被推斷為字串型別陣列，即 String[]
// 其值為 ["OneSix", "FiveEight", "FiveOneZero"]
```

`map` 在陣列中為每一個元素呼叫了閉包表達式。 你不需要指定閉包的輸入參數 `number` 的型別，因為可以通過要映射的陣列型別進行推斷。

閉包 `number` 參數被宣告為一個變數參數（變數的具體描述請參看常數參數和變數參數），因此可以在閉包函式體內對其進行修改。閉包表達式制定了回傳型別為 `String`，以表明儲存映射值的新陣列型別為 `String`。

閉包表達式在每次被呼叫的時候創建了一個字串並回傳。 其使用取餘運算子 (number % 10) 計算最後一位數字並利用 `digitNames` 字典獲取所映射的字串。

> 注意： 字典 `digitNames` 下標後跟著一個嘆號 (!)，因為字典下標回傳一個可選值 (optional value)，表明即使該 key 不

> 存在也不會查找失敗。 在上例中，它保證了 `number % 10` 可以總是作為一個 `digitNames` 字典的有效下標 key。 因此嘆號可以用於強制解析 (force-unwrap) 儲存在可選下標項中的 `String` 型別值。

從 `digitNames` 字典中獲取的字串被添加到輸出的前部，逆序建立了一個字串版本的數字。 （在表達式 `number % 10` 中，如果number為16，則回傳6，58回傳8，510回傳0）。

`number` 變數之後除以10。 因為其是整數，在計算過程中未除盡部分被忽略。 因此 16變成了1，58變成了5，510變成了51。

整個過程重複進行，直到 `number /= 10` 為0，這時閉包會將字串輸出，而 `map` 函式則會將字串添加到所映射的陣列中。

上例中尾隨閉包語法在函式後整潔封裝了具體的閉包功能，而不再需要將整個閉包包裹在 `map` 函式的括號內。

捕獲值（Capturing Values）

閉包可以在其定義的上下文中捕獲常數或變數。 即使定義這些常數和變數的原域已經不存在，閉包仍然可以在閉包函式體內參考和修改這些值。

Swift最簡單的閉包形式是嵌套函式，也就是定義在其他函式的函式體內的函式。 嵌套函式可以捕獲其外部函式所有的參數以及定義的常數和變數。

下例為一個叫做 `makeIncrementor` 的函式，其包含了一個叫做 `incrementor` 嵌套函式。 嵌套函式 `incrementor` 從上下文中捕獲了兩個值， `runningTotal` 和 `amount` 。 之後 `makeIncrementor` 將 `incrementor` 作為閉包回傳。 每次呼叫 `incrementor` 時，其會以 `amount` 作為增量增加 `runningTotal` 的值。

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

`makeIncrementor` 回傳型別為 `() -> Int` 。 這意味著其回傳的是一個函式，而不是一個簡單型別值。 該函式在每次呼叫時不接受參數只回傳一個 `Int` 型別的值。 關於函式回傳其他函式的內容，請查看函式型別作為回傳型別。

`makeIncrementor` 函式定義了一個整型變數 `runningTotal` (初始為0) 用來儲存當前跑步總數。 該值通過 `incrementor` 回傳。

`makeIncrementor` 有一個 `Int` 型別的參數，其外部命名為 `forIncrement` ， 內部命名為 `amount` ，表示每次 `incrementor` 被呼叫時 `runningTotal` 將要增加的量。

`incrementor` 函式用來執行實際的增加操作。 該函式簡單地使 `runningTotal` 增加 `amount` ，並將其回傳。

如果我們單獨看這個函式，會發現看上去不同尋常：

```
func incrementor() -> Int {
    runningTotal += amount
    return runningTotal
}
```

`incrementor` 函式並沒有獲取任何參數，但是在函式體內存取了 `runningTotal` 和 `amount` 變數。這是因為其通過捕獲在包含它的函式體內已經存在的 `runningTotal` 和 `amount` 變數而實作。

由於沒有修改 `amount` 變數， `incrementor` 實際上捕獲並儲存了該變數的一個副本，而該副本隨著 `incrementor` 一同被儲存。

然而，因為每次呼叫該函式的時候都會修改 `runningTotal` 的值， `incrementor` 捕獲了當前 `runningTotal` 變數的參考，而不是僅僅複製該變數的初始值。捕獲一個參考保證了當 `makeIncrementor` 結束時候並不會消失，也保證了當下一次執行 `incrementor` 函式時， `runningTotal` 可以繼續增加。

> 注意： Swift 會決定捕獲參考還是拷貝值。 你不需要標注 `amount` 或者 `runningTotal` 來宣告在嵌入的 `incrementor` 函式中的使用方式。 Swift 同時也處理 `runingTotal` 變數的內存管理操作，如果不再被 `incrementor` 函式使用，則會被清除。

下面程式碼為一個使用 `makeIncrementor` 的範例：

```
let incrementByTen = makeIncrementor(forIncrement: 10)
```

該範例定義了一個叫做 `incrementByTen` 的常數，該常數指向一個每次呼叫會加10的 `incrementor` 函式。 呼叫這個函式多次可以得到以下結果：

```
incrementByTen()
// 回傳的值為10
incrementByTen()
// 回傳的值為20
incrementByTen()
// 回傳的值為30
```

如果你創建了另一個 `incrementor` ，其會有一個屬於自己的獨立的 `runningTotal` 變數的參考。 下面的範例中， `incrementBySevne` 捕獲了一個新的 `runningTotal` 變數，該變數和 `incrementByTen` 中捕獲的變數沒有任何聯系：

```
let incrementBySeven = makeIncrementor(forIncrement: 7)
incrementBySeven()
// 回傳的值為7
incrementByTen()
// 回傳的值為40
```

> 注意： 如果你將閉包賦值給一個類別實例的屬性，並且該閉包通過指向該實例或其成員來捕獲了該實例，你將創建一個在閉包和實例間的強參考環。 Swift 使用捕獲列表來打破這種強參考環。更多資訊，請參考 閉包引起的迴圈強參考。

閉包是參考型別（Closures Are Reference Types）

上面的範例中， `incrementBySeven` 和 `incrementByTen` 是常數，但是這些常數指向的閉包仍然可以增加其捕獲的變數值。 這是因為函式和閉包都是參考型別。

無論你將函式/閉包賦值給一個常數還是變數，你實際上都是將常數/變數的值設置為對應函式/閉包的參考。 上面的範例中， `incrementByTen` 指向閉包的參考是一個常數，而並非閉包內容本身。

這也意味著如果你將閉包賦值給了兩個不同的常數/變數，兩個值都會指向同一個閉包：

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// 回傳的值為50
```

翻譯：yankuangshi 校對：shinyzhu

列舉（Enumerations）

本頁內容包含：

列舉定義了一個通用型別的一組相關的值，使你可以在你的程式碼中以一個安全的方式來使用這些值。

如果你熟悉 C 語言，你就會知道，在 C 語言中列舉指定相關名稱為一組整型值。Swift 中的列舉更加靈活，不必給每一個列舉成員提供一個值。如果一個值（被認為是「原始」值）被提供給每個列舉成員，則該值可以是一個字串，一個字元，或是一個整型值或浮點值。

此外，列舉成員可以指定任何型別的實例值儲存到列舉成員值中，就像其他語言中的聯合體（unions）和變體（variants）。你可以定義一組通用的相關成員作為列舉的一部分，每一組都有不同的一組與它相關的適當型別的數值。

在 Swift 中，列舉型別是一等（first-class）型別。它們采用了很多傳統上只被類別（class)所支援的特征，例如計算型屬性（computed properties），用於提供關於列舉當前值的附加資訊， 實例方法（instance methods），用於提供和列舉所代表的值相關聯的功能。列舉也可以定義建構函式（initializers）來提供一個初始成員值；可以在原始的實作基礎上擴展它們的功能；可以遵守協定（protocols）來提供標準的功能。

欲了解更多相關功能，請參見屬性（Properties），方法（Methods），建構過程（Initialization），擴展（Extensions）和協定（Protocols）。

列舉語法

使用 enum 關鍵詞並且把它們的整個定義放在一對大括號內：

```
enum SomeEnumeration {
  // enumeration definition goes here
}
```

以下是指南針四個方向的一個範例：

```
enum CompassPoint {
  case North
  case South
  case East
  case West
}
```

一個列舉中被定義的值（例如 North，South，East 和 West）是列舉的成員值（或者成員）。case 關鍵詞表明新的一行成員值將被定義。

> 注意：
> 不像 C 和 Objective-C 一樣，Swift 的列舉成員在被創建時不會被賦予一個預設的整數值。在上面的 CompassPoints 範例中，North，South，East 和 West 不是隱式的等於 0，1，2 和 3。相反的，這些不同的列舉成員

> 在 `CompassPoint` 的一種顯示定義中擁有各自不同的值。

多個成員值可以出現在同一行上，用逗號隔開：

```
enum Planet {
  case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Nepturn
}
```

每個列舉定義了一個全新的型別。像 Swift 中其他型別一樣，它們的名字（例如 `CompassPoint` 和 `Planet` ）必須以一個大寫字母開頭。給列舉型別起一個單數名字而不是復數名字，以便於讀起來更加容易理解：

```
var directionToHead = CompassPoint.West
```

`directionToHead` 的型別被推斷當它被 `CompassPoint` 的一個可能值初始化。一旦 `directionToHead` 被宣告為一個 `CompassPoint` ，你可以使用更短的點（.）語法將其設置為另一個 `CompassPoint` 的值：

```
directionToHead = .East
```

`directionToHead` 的型別已知時，當設定它的值時，你可以不再寫型別名。使用顯示型別的列舉值可以讓程式碼具有更好的可讀性。

匹配列舉值和 `Switch` 語句

你可以匹配單個列舉值和 `switch` 語句：

```
directionToHead = .South
switch directionToHead {
case .North:
    println("Lots of planets have a north")
case .South:
    println("Watch out for penguins")
case .East:
    println("Where the sun rises")
case .West:
    println("Where the skies are blue")
}
// 輸出 "Watch out for penguins」
```

你可以如此理解這段程式碼：

「考慮 `directionToHead` 的值。當它等於 `.North` ，列印 `「Lots of planets have a north」` 。當它等於 `.South` ，列印 `「Watch out for penguins」` 。」

等等依次類別推。

正如在控制流程（Control Flow）中介紹，當考慮一個列舉的成員們時，一個 `switch` 語句必須全面。如果忽略了 `.West` 這種情況，上面那段程式碼將無法通過編譯，因為它沒有考慮到 `CompassPoint` 的全部成員。全面性的要求確保了列舉成員不會被意外遺漏。

當不需要匹配每個列舉成員的時候，你可以提供一個預設 `default` 分支來涵蓋所有未明確被提出的任何成員：

```
let somePlanet = Planet.Earth
switch somePlanet {
```

```
case .Earth:
    println("Mostly harmless")
default:
    println("Not a safe place for humans")
}
// 輸出 "Mostly harmless」
```

實例值（Associated Values）

上一小節的範例演示了一個列舉的成員是如何被定義（分類別）的。你可以為 `Planet.Earth` 設置一個常數或則變數，並且在之後查看這個值。然而，有時候會很有用如果能夠把其他型別的實例值和成員值一起儲存起來。這能讓你隨著成員值儲存額外的自定義資訊，並且當每次你在程式碼中利用該成員時允許這個資訊產生變化。

你可以定義 Swift 的列舉儲存任何型別的實例值，如果需要的話，每個成員的資料型別可以是各不相同的。列舉的這種特性跟其他語言中的可辨識聯合（discriminated unions），標簽聯合（tagged unions），或者變體（variants）相似。

例如，假設一個函式庫存跟蹤系統需要利用兩種不同型別的條形碼來跟蹤商品。有些商品上標有 UPC-A 格式的一維碼，它使用數字 0 到 9。每一個條形碼都有一個代表「數字系統」的數字，該數字後接 10 個代表「識別符號」的數字。最後一個數字是「檢查」位，用來驗證程式碼是否被正確掃描：

其他商品上標有 QR 碼格式的二維碼，它可以使用任何 ISO8859-1 字元，並且可以編碼一個最多擁有 2,953 字元的字串:

對於函式庫存跟蹤系統來說，能夠把 UPC-A 碼作為三個整型值的元組，和把 QR 碼作為一個任何長度的字串儲存起來是方便的。

在 Swift 中，用來定義兩種商品條碼的列舉是這樣子的：

```
enum Barcode {
  case UPCA(Int, Int, Int)
  case QRCode(String)
}
```

以上程式碼可以這麼理解：

「定義一個名為 `Barcode` 的列舉型別，它可以是 `UPCA` 的一個實例值（ `Int`， `Int`， `Int` ），或者 `QRCode` 的一個字串型別（ `String` ）實例值。」

這個定義不提供任何 `Int` 或 `String` 的實際值，它只是定義了，當 `Barcode` 常數和變數等於 `Barcode.UPCA` 或 `Barcode.QRCode` 時，實例值的型別。

然後可以使用任何一種條碼型別創建新的條碼，如：

```
var productBarcode = Barcode.UPCA(8, 85909_51226, 3)
```

以上範例創建了一個名為 `productBarcode` 的新變數，並且賦給它一個 `Barcode.UPCA` 的實例元組值 `(8, 8590951226, 3)`。提供的「識別符號」值在整數字中有一個底線，使其便於閱讀條形碼。

同一個商品可以被分配給一個不同型別的條形碼，如：

```
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

這時，原始的 `Barcode.UPCA` 和其整數值被新的 `Barcode.QRCode` 和其字串值所替代。條形碼的常數和變數可以儲存一個 `.UPCA` 或者一個 `.QRCode`（連同它的實例值），但是在任何指定時間只能儲存其中之一。

像以前那樣，不同的條形碼型別可以使用一個 switch 語句來檢查，然而這次實例值可以被提取作為 switch 語句的一部分。你可以在 `switch` 的 case 分支程式碼中提取每個實例值作為一個常數（用 `let` 前綴）或者作為一個變數（用 `var` 前綴）來使用：

```
switch productBarcode {
case .UPCA(let numberSystem, let identifier, let check):
    println("UPC-A with value of \(numberSystem), \(identifier), \(check).")
case .QRCode(let productCode):
    println("QR code with value of \(productCode).")
}
// 輸出 "QR code with value of ABCDEFGHIJKLMNOP.」
```

如果一個列舉成員的所有實例值被提取為常數，或者它們全部被提取為變數，為了簡潔，你可以只放置一個 `var` 或者 `let` 標注在成員名稱前：

```
switch productBarcode {
case let .UPCA(numberSystem, identifier, check):
    println("UPC-A with value of \(numberSystem), \(identifier), \(check).")
case let .QRCode(productCode):
    println("QR code with value of \(productCode).")
}
// 輸出 "QR code with value of ABCDEFGHIJKLMNOP."
```

原始值（Raw Values）

在實例值小節的條形碼範例中演示了一個列舉的成員如何宣告它們儲存不同型別的實例值。作為實例值的替代，列舉成員可以被預設值（稱為原始值）預先填充，其中這些原始值具有相同的型別。

這裡是一個列舉成員儲存原始 ASCII 值的範例：

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

在這裡，稱為 `ASCIIControlCharacter` 的列舉的原始值型別被定義為字元型 `Character`，並被設置了一些比較常見的 ASCII 控制字元。字元值的描述請詳見字串和字元 `Strings and Characters` 部分。

注意，原始值和實例值是不相同的。當你開始在你的程式碼中定義列舉的時候原始值是被預先填充的值，像上述三個 ASCII 碼。對於一個特定的列舉成員，它的原始值始終是相同的。實例值是當你在創建一個基於列舉成員的新常數或變數時才會被設置，並且每次當你這麼做得時候，它的值可以是不同的。

原始值可以是字串，字元，或者任何整型值或浮點型值。每個原始值在它的列舉宣告中必須是唯一的。當整型值被用於原始值，如果其他列舉成員沒有值時，它們會自動遞增。

下面的列舉是對之前 `Planet` 這個列舉的一個細化，利用原始整型值來表示每個 planet 在太陽系中的順序：

```
enum Planet: Int {
    case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

自動遞增意味著 `Planet.Venus` 的原始值是 `2`，依次類別推。

使用列舉成員的 `toRaw` 方法可以存取該列舉成員的原始值：

```
let earthsOrder = Planet.Earth.toRaw()
// earthsOrder is 3
```

使用列舉的 `fromRaw` 方法來試圖找到具有特定原始值的列舉成員。這個範例通過原始值 `7` 識別 `Uranus`：

```
let possiblePlanet = Planet.fromRaw(7)
// possiblePlanet is of type Planet? and equals Planet.Uranus
```

然而，並非所有可能的 `Int` 值都可以找到一個匹配的行星。正因為如此， `fromRaw` 方法可以回傳一個可選的列舉成員。在上面的範例中， `possiblePlanet` 是 `Planet?` 型別，或「可選的 `Planet` 」。

如果你試圖尋找一個位置為9的行星，通過 `fromRaw` 回傳的可選 `Planet` 值將是 `nil`：

```
let positionToFind = 9
if let somePlanet = Planet.fromRaw(positionToFind) {
    switch somePlanet {
    case .Earth:
        println("Mostly harmless")
    default:
        println("Not a safe place for humans")
    }
} else {
    println("There isn't a planet at position \(positionToFind)")
}
// 輸出 "There isn't a planet at position 9
```

這個範例使用可選綁定（optional binding），通過原始值 `9` 試圖存取一個行星。 `if let somePlanet = Planet.fromRaw(9)` 語句獲得一個可選 `Planet`，如果可選 `Planet` 可以被獲得，把 `somePlanet` 設置成該可選 `Planet` 的內容。在這個範例中，無法檢索到位置為 `9` 的行星，所以 `else` 分支被執行。

> 翻譯：JaySurplus 校對：sg552

類別和結構

本頁包含內容：

類別和結構是人們構建程式碼所用的一種通用且靈活的建構體。為了在類別和結構中實作各種功能，我們必須要嚴格按照對於常數，變數以及函式所規定的語法規則來定義屬性和添加方法。

與其他程式語言所不同的是，Swift 並不要求你為自定義類別和結構去創建獨立的接口和實作文件。你所要做的是在一個單一文件中定義一個類別或者結構，系統將會自動生成面向其它程式碼的外部接口。

> 注意：
> 通常一個 `類別` 的實例被稱為 `物件` 。然而在Swift 中，類別和結構的關系要比在其他語言中更加的密切，本章中所討論的大部分功能都可以用在類別和結構上。因此，我們會主要使用 `實例` 而不是 `物件` 。

類別和結構對比

Swift 中類別和結構有很多共同點。共同處在於：

- 定義屬性用於儲存值
- 定義方法用於提供功能
- 定義附屬腳本用於存取值
- 定義建構器用於生成初始化值
- 通過擴展以增加預設實作的功能
- 符合協定以對某類別提供標準功能

更多資訊請參見 屬性，方法，下標腳本，初始過程，擴展，和協定。

與結構相比，類別還有如下的附加功能：

- 繼承允許一個類別繼承另一個類別的特征
- 型別轉換允許在執行時檢查和解釋一個類別實例的型別
- 解構器允許一個類別實例釋放任何其所被分配的資源
- 參考計數允許對一個類別的多次參考

更多資訊請參見繼承，型別轉換，初始化，和自動引用計數。

> 注意：
> 結構總是通過被複製的方式在程式碼中傳遞，因此請不要使用參考計數。

定義

類別和結構有著類似的定義方式。我們通過關鍵字 `class` 和 `struct` 來分別表示類別和結構，並在一對大括號中定義它們的具體內容：

```
class SomeClass {
```

```
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

> 注意：
> 在你每次定義一個新類別或者結構的時候，實際上你是有效地定義了一個新的 Swift 型別。因此請使用 `UpperCamelCase` 這種方式來命名（如 `SomeClass` 和 `SomeStructure` 等），以便符合標準Swift 型別的大寫命名風格（如 `String`， `Int` 和 `Bool`）。相反的，請使用 `lowerCamelCase` 這種方式為屬性和方法命名（如 `framerate` 和 `incrementCount`），以便和類別區分。

以下是定義結構和定義類別的示例：

```
struct Resolution {
    var width = 0
    var heigth = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

在上面的示例中我們定義了一個名為 `Resolution` 的結構，用來描述一個顯示器的像素分辨率。這個結構包含了兩個名為 `width` 和 `height` 的儲存屬性。儲存屬性是捆綁和儲存在類別或結構中的常數或變數。當這兩個屬性被初始化為整數 `0` 的時候，它們會被推斷為 `Int` 型別。

在上面的示例中我們還定義了一個名為 `VideoMode` 的類別，用來描述一個視頻顯示器的特定模式。這個類別包含了四個儲存屬性變數。第一個是 `分辨率`，它被初始化為一個新的 `Resolution` 結構的實例，具有 `Resolution` 的屬性型別。新 `VideoMode` 實例同時還會初始化其它三個屬性，它們分別是，初始值為 `false` (意為「non-interlaced video」)的 `inteflaced`，回放幀率初始值為 `0.0` 的 `frameRate` 和值為可選 `String` 的 `name`。 `name` 屬性會被自動賦予一個預設值 `nil`，意為「沒有 `name` 值」，因它是一個可選型別。

類別和結構實例

`Resolution` 結構和 `VideoMode` 類別的定義僅描述了什麼是 `Resolution` 和 `VideoMode`。它們並沒有描述一個特定的分辨率（resolution）或者視頻模式（video mode）。為了描述一個特定的分辨率或者視頻模式，我們需要生成一個它們的實例。

生成結構和類別實例的語法非常相似：

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

結構和類別都使用建構器語法來生成新的實例。建構器語法的最簡單形式是在結構或者類別的型別名稱後跟隨一個空括弧，如 `Resolution()` 或 `VideoMode()`。通過這種方式所創建的類別或者結構實例，其屬均會被初始化為預設值。建構過程章節會對類別和結構的初始化進行更詳細的討論。

屬性存取

通過使用點語法（*dot syntax*）,你可以存取實例中所含有的屬性。其語法規則是，實例名後面緊跟屬性名，兩者通過點號(.)連接：

```
println("The width of someResolution is \(someResolution.width)")
```

```
// 輸出 "The width of someResolution is 0"
```

在上面的範例中，`someResolution.width` 參考 `someResolution` 的 `width` 屬性，回傳 `width` 的初始值 `0`。

你也可以存取子屬性，如何 `VideoMode` 中 `Resolution` 屬性的 `width` 屬性：

```
println("The width of someVideoMode is \(someVideoMode.resolution.width)")
// 輸出 "The width of someVideoMode is 0"
```

你也可以使用點語法為屬性變數賦值：

```
someVideoMode.resolution.width = 12880
println("The width of someVideoMode is now \(someVideoMode.resolution.width)")
// 輸出 "The width of someVideoMode is now 1280"
```

> 注意：
> 與 Objective-C 語言不同的是，Swift 允許直接設置結構屬性的子屬性。上面的最後一個範例，就是直接設置了 `someVideoMode` 中 `resolution` 屬性的 `width` 這個子屬性，以上操作並不需要從新設置 `resolution` 屬性。

結構型別的成員逐一建構器(Memberwise Initializers for structure Types)

所有結構都有一個自動生成的成員逐一建構器，用於初始化新結構實例中成員的屬性。新實例中各個屬性的初始值可以通過屬性的名稱傳遞到成員逐一建構器之中：

```
let vga = resolution(width:640, heigth: 480)
```

與結構不同，類別實例沒有預設的成員逐一建構器。建構過程章節會對建構器進行更詳細的討論。

結構和列舉是值型別

值型別被賦予給一個變數，常數或者本身被傳遞給一個函式的時候，實際上操作的是其的拷貝。

在之前的章節中，我們已經大量使用了值型別。實際上，在 Swift 中，所有的基本型別：整數（Integer）、浮點數（floating-point）、布林值（Booleans）、字串（string)、陣列（array）和字典（dictionaries），都是值型別，並且都是以結構的形式在後台所實作。

在 Swift 中，所有的結構和列舉都是值型別。這意味著它們的實例，以及實例中所包含的任何值型別屬性，在程式碼中傳遞的時候都會被複製。

請看下面這個示例，其使用了前一個示例中 `Resolution` 結構：

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

在以上示例中，宣告了一個名為 `hd` 的常數，其值為一個初始化為全高清視頻分辨率（1920 像素寬，1080 像素高）的 `Resolution` 實例。

然後示例中又宣告了一個名為 `cinema` 的變數，其值為之前宣告的 `hd`。因為 `Resolution` 是一個結構，所以 `cinema` 的值其實是 `hd` 的一個拷貝副本，而不是 `hd` 本身。儘管 `hd` 和 `cinema` 有著相同的寬（width）和高（height）屬性，但是在後台中，它們是兩個完全不同的實例。

下面，為了符合數碼影院放映的需求（2048 像素寬，1080 像素高），`cinema` 的 `width` 屬性需要作如下修改：

```
cinema.width = 2048
```

這裡，將會顯示 `cinema` 的 `width` 屬性確已改為了 `2048`：

```
println("cinema is now  \(cinema.width) pixels wide")
// 輸出 "cinema is now 2048 pixels wide"
```

然而，初始的 `hd` 實例中 `width` 屬性還是 `1920`：

```
println("hd is still \(hd.width    ) pixels wide")
// 輸出 "hd is still 1920 pixels wide"
```

在將 `hd` 賦予給 `cinema` 的時候，實際上是將 `hd` 中所儲存的 `值（values）` 進行拷貝，然後將拷貝的資料儲存到新的 `cinema` 實例中。結果就是兩個完全獨立的實例碰巧包含有相同的數值。由於兩者相互獨立，因此將 `cinema` 的 `width` 修改為 `2048` 並不會影響 `hd` 中的寬（width）。

列舉也遵循相同的行為准則：

```
enum CompassPoint {
    case North, South, East, West
}
var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection
currentDirection = .East
if rememberDirection == .West {
    println("The remembered direction is still .West")
}
// 輸出 "The remembered direction is still .West"
```

上例中 `rememberedDirection` 被賦予了 `currentDirection` 的值（value），實際上它被賦予的是值（value）的一個拷貝。賦值過程結束後再修改 `currentDirection` 的值並不影響 `rememberedDirection` 所儲存的原始值（value）的拷貝。

類別是參考型別

與值型別不同，參考型別在被賦予到一個變數，常數或者被傳遞到一個函式時，操作的並不是其拷貝。因此，參考的是已存在的實例本身而不是其拷貝。

請看下面這個示例，其使用了之前定義的 `VideoMode` 類別：

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

以上示例中，宣告了一個名為 `tenEighty` 的常數，其參考了一個 `VideoMode` 類別的新實例。在之前的示例中，這個視頻模式（video mode）被賦予了HD分辨率（1920*1080）的一個拷貝（ `hd` ）。同時設置為交錯（interlaced）,命名為 `「1080i」` 。最後，其幀率是 `25.0` 幀每秒。

然後， `tenEighty` 被賦予名為 `alsoTenEighty` 的新常數，同時對 `alsoTenEighty` 的幀率進行修改：

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

因為類別是參考型別，所以 `tenEight` 和 `alsoTenEight` 實際上參考的是相同的 `VideoMode` 實例。換句話說，它們只是同一個實例的兩種叫法。

下面，通過查看 `tenEighty` 的 `frameRate` 屬性，我們會發現它正確的顯示了基本 `VideoMode` 實例的新幀率，其值為 `30.0`：

```
println("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// 輸出 "The frameRate property of theEighty is now 30.0"
```

需要注意的是 `tenEighty` 和 `alsoTenEighty` 被宣告為常數（（*constants*）而不是變數。然而你依然可以改變 `tenEighty.frameRate` 和 `alsoTenEighty.frameRate`,因為這兩個常數本身不會改變。它們並不 儲存 這個 `VideoMode` 實例，在後台僅僅是對 `VideoMode` 實例的參考。所以，改變的是被參考的基礎 `VideoMode` 的 `frameRate` 參數，而不改變常數的值。

恆等運算子

因為類別是參考型別，有可能有多個常數和變數在後台同時參考某一個類別實例。（對於結構和列舉來說，這並不成立。因為它們作值型別，在被賦予到常數，變數或者傳遞到函式時，總是會被拷貝。）

如果能夠判定兩個常數或者變數是否參考同一個類別實例將會很有幫助。為了達到這個目的，Swift 內建了兩個恆等運算子：

- 等價於 （ === ）
- 不等價於 （ !== ）

以下是運用這兩個運算子檢測兩個常數或者變數是否參考同一個實例：

```
if tenEighty === alsoTenTighty {
    println("tenTighty and alsoTenEighty refer to the same Resolution instance.")
}
//輸出 "tenEighty and alsoTenEighty refer to the same Resolution instance."
```

請注意「等價於」（用三個等號表示，===） 與「等於」（用兩個等號表示，==）的不同：

- 「等價於」表示兩個類型別（class type）的常數或者變數參考同一個類別實例。
- 「等於」表示兩個實例的值「相等」或「相同」，判定時要遵照類別設計者定義定義的評判標準，因此相比於「相等」，這是一種更加合適的叫法。

當你在定義你的自定義類別和結構的時候，你有義務來決定判定兩個實例「相等」的標準。在章節運算子函式(Operator Functions)中將會詳細介紹實作自定義「等於」和「不等於」運算子的流程。

指針

如果你有 C，C++ 或者 Objective-C 語言的經驗，那麼你也許會知道這些語言使用指針來參考內存中的地址。一個 Swift 常數或者變數參考一個參考型別的實例與 C 語言中的指針類似，不同的是並不直接指向內存中的某個地址，而且也不要求你使用星號（*）來表明你在創建一個參考。Swift 中這些參考與其它的常數或變數的定義方式相同。

類別和結構的選擇

在你的程式碼中，你可以使用類別和結構來定義你的自定義資料型別。

然而，結構實例總是通過值傳遞，類別實例總是通過參考傳遞。這意味兩者適用不同的任務。當你的在考慮一個工程項目的

資料建構和功能的時候，你需要決定每個資料建構是定義成類別還是結構。

按照通用的准則，當符合一條或多條以下條件時，請考慮構建結構：

- 結構的主要目的是用來封裝少量相關簡單資料值。
- 有理由預計一個結構實例在賦值或傳遞時，封裝的資料將會被拷貝而不是被參考。
- 任何在結構中儲存的值型別屬性，也將會被拷貝，而不是被參考。
- 結構不需要去繼承另一個已存在型別的屬性或者行為。

合適的結構候選者包括：

- 幾何形狀的大小，封裝一個 `width` 屬性和 `height` 屬性，兩者均為 `Double` 型別。
- 一定範圍內的路徑，封裝一個 `start` 屬性和 `length` 屬性，兩者均為 `Int` 型別。
- 三維坐標系內一點，封裝 `x`， `y` 和 `z` 屬性，三者均為 `Double` 型別。

在所有其它案例中，定義一個類別，生成一個它的實例，並通過參考來管理和傳遞。實際中，這意味著絕大部分的自定義資料建構都應該是類別，而非結構。

集合（Collection）型別的賦值和拷貝行為

Swift 中 `陣列（Array）` 和 `字典（Dictionary）` 型別均以結構的形式實作。然而當陣列被賦予一個常數或變數，或被傳遞給一個函式或方法時，其拷貝行為與字典和其它結構有些許不同。

以下對 `陣列` 和 `結構` 的行為描述與對 `NSArray` 和 `NSDictionary` 的行為描述在本質上不同，後者是以類別的形式實作，前者是以結構的形式實作。 `NSArray` 和 `NSDictionary` 實例總是以對已有實例參考,而不是拷貝的方式被賦值和傳遞。

> 注意：
> 以下是對於陣列，字典，字串和其它值的 `拷貝` 的描述。 在你的程式碼中，拷貝好像是確實是在有拷貝行為的地方產生過。然而，在 Swift 的後台中，只有確有必要， `實際（actual）` 拷貝才會被執行。Swift 管理所有的值拷貝以確保性能最優化的性能，所以你也沒有必要去避免賦值以保證最優性能。（實際賦值由系統管理優化）

字典型別的賦值和拷貝行為

無論何時將一個 `字典` 實例賦給一個常數或變數，或者傳遞給一個函式或方法，這個字典會即會在賦值或呼叫發生時被拷貝。在章節結構和列舉是值型別中將會對此過程進行詳細介紹。

如果 `字典` 實例中所儲存的鍵（keys）和/或值（values）是值型別（結構或列舉），當賦值或呼叫發生時，它們都會被拷貝。相反，如果鍵（keys）和/或值（values）是參考型別，被拷貝的將會是參考，而不是被它們參考的類別實例或函式。 `字典` 的鍵和值的拷貝行為與結構所儲存的屬性的拷貝行為相同。

下面的示例定義了一個名為 `ages` 的字典，其中儲存了四個人的名字和年齡。 `ages` 字典被賦予了一個名為 `copiedAges` 的新變數，同時 `ages` 在賦值的過程中被拷貝。賦值結束後， `ages` 和 `copiedAges` 成為兩個相互獨立的字典。

```
var ages = ["Peter": 23, "Wei": 35, "Anish": 65, "Katya": 19]
var copiedAges = ages
```

這個字典的鍵（keys）是 `字串（String）` 型別， `值（values）` 是 `整（Int）` 型別。這兩種型別在Swift 中都是值型別（value types），所以當字典被拷貝時，兩者都會被拷貝。

我們可以通過改變一個字典中的年齡值（age value），檢查另一個字典中所對應的值，來證明 `ages` 字典確實是被拷貝了。如果在 `copiedAges` 字典中將 `Peter` 的值設為 `24`，那麼 `ages` 字典仍然會回傳修改前的值 `23`：

```
copiedAges["Peter"] = 24
```

```
println(ages["Peter"])
// 輸出 "23"
```

陣列的賦值和拷貝行為

在Swift 中，陣列（Arrays）型別的賦值和拷貝行為要比字典（Dictionary）型別的複雜的多。當操作陣列內容時，陣列（Array）能提供接近C語言的的性能，並且拷貝行為只有在必要時才會發生。

如果你將一個陣列（Array）實例賦給一個變數或常數，或者將其作為參數傳遞給函式或方法呼叫，在事件發生時陣列的內容不會被拷貝。相反，陣列公用相同的元素序列。當你在一個陣列內修改某一元素，修改結果也會在另一陣列顯示。

對陣列來說，拷貝行為僅僅當操作有可能修改陣列長度時才會發生。這種行為包括了附加（appending）,插入（inserting），刪除（removing）或者使用範圍下標（ranged subscript）去替換這一範圍內的元素。只有當陣列拷貝確要發生時，陣列內容的行為規則與字典中鍵值的相同，參見章節[集合（collection）型別的賦值與複製行為](#assignment_and_copy_behavior_for_collection_types。

下面的示例將一個整數（Int）陣列賦給了一個名為 a 的變數，繼而又被賦給了變數 b 和 c ：

```
var a = [1, 2, 3]
var b = a
var c = a
```

我們可以在 a , b , c 上使用下標語法以得到陣列的第一個元素：

```
println(a[0])
// 1
println(b[0])
// 1
println(c[0])
// 1
```

如果通過下標語法修改陣列中某一元素的值，那麼 a , b , c 中的相應值都會發生改變。請注意當你用下標語法修改某一值時，並沒有拷貝行為伴隨發生，因為下表語法修改值時沒有改變陣列長度的可能：

```
a[0] = 42
println(a[0])
// 42
println(b[0])
// 42
println(c[0])
// 42
```

然而，當你給 a 附加新元素時，陣列的長度會改變。 當附加元素這一事件發生時，Swift 會創建這個陣列的一個拷貝。從此以後， a 將會是原陣列的一個獨立拷貝。

拷貝發生後，如果再修改 a 中元素值的話， a 將會回傳與 b ， c 不同的結果，因為後兩者參考的是原來的陣列：

```
a.append(4)
a[0] = 777
println(a[0])
// 777
println(b[0])
// 42
println(c[0])
// 42
```

確保陣列的唯一性

在操作一個陣列，或將其傳遞給函式以及方法呼叫之前是很有必要先確定這個陣列是有一個唯一拷貝的。通過在陣列變數上呼叫 `unshare` 方法來確定陣列參考的唯一性。（當陣列賦給常數時，不能呼叫 `unshare` 方法）

如果一個陣列被多個變數參考，在其中的一個變數上呼叫 `unshare` 方法，則會拷貝此陣列，此時這個變數將會有屬於它自己的獨立陣列拷貝。當陣列僅被一個變數參考時，則不會有拷貝發生。

在上一個示例的最後， `b` 和 `c` 都參考了同一個陣列。此時在 `b` 上呼叫 `unshare` 方法則會將 `b` 變成一個唯一個拷貝：

```
b.unshare()
```

在 `unshare` 方法呼叫後再修改 `b` 中第一個元素的值，這三個陣列（ `a` , `b` , `c` ）會回傳不同的三個值：

```
b[0] = -105
println(a[0])
// 77
println(b[0])
// -105
println(c[0])
// 42
```

判定兩個陣列是否共用相同元素

我們通過使用恆等運算子（identity operators）（ === 和 !==）來判定兩個陣列或子陣列共用相同的儲存空間或元素。

下面這個示例使用了「等同（identical to）」 運算子（===） 來判定 `b` 和 `c` 是否共用相同的陣列元素：

```
if b === c {
    println("b and c still share the same array elements.")
} else {
    println("b and c now refer to two independent sets of array elements.")
}
```

```
// 輸出 "b and c now refer totwo independent sets of array elements."
```

此外，我們還可以使用恆等運算子來判定兩個子陣列是否共用相同的元素。下面這個示例中，比較了 `b` 的兩個相等的子陣列，並且確定了這兩個子陣列都參考相同的元素：

```
if b[0...1] === b[0...1] {
    println("These two subarrays share the same elements.")
} else {
    println("These two subarrays do not share the same elements.")
}
// 輸出 "These two subarrays share the same elements."
```

強制複製陣列

我們通過呼叫陣列的 `copy` 方法進行強制顯性複製。這個方法對陣列進行了淺拷貝（shallow copy）,並且回傳一個包含此拷貝的新陣列。

下面這個示例中定義了一個 `names` 陣列，其包含了七個人名。還定義了一個 `copiedNames` 變數，用以儲存在 `names` 上呼

叫 `copy` 方法所回傳的結果：

```
var names = ["Mohsen", "Hilary", "Justyn", "Amy", "Rich", "Graham", "Vic"]
var copiedNames = names.copy()
```

我們可以通過修改一個陣列中某元素，並且檢查另一個陣列中對應元素的方法來判定 `names` 陣列確已被複製。如果你將 `copiedNames` 中第一個元素從"`Mohsen`"修改為"`Mo`",則 `names` 陣列回傳的仍是拷貝發生前的"`Mohsen`"：

```
copiedName[0] = "Mo"
println(name[0])
// 輸出 "Mohsen"
```

> 注意：
> 如果你僅需要確保你對陣列的參考是唯一參考，請呼叫 `unshare` 方法，而不是 `copy` 方法。 `unshare` 方法僅會在確有必要時才會創建陣列拷貝。 `copy` 方法會在任何時候都創建一個新的拷貝，即使參考已經是唯一參考。

> 翻譯：shinyzhu 校對：pp-prog

屬性 (Properties)

本頁包含內容：

屬性將值跟特定的類別、結構或列舉關聯。儲存屬性儲存常數或變數作為實例的一部分，計算屬性計算（而不是儲存）一個值。計算屬性可以用於類別、結構和列舉裡，儲存屬性只能用於類別和結構。

儲存屬性和計算屬性通常用於特定型別的實例，但是，屬性也可以直接用於型別本身，這種屬性稱為型別屬性。

另外，還可以定義屬性監視器來監控屬性值的變化，以此來觸發一個自定義的操作。屬性監視器可以添加到自己寫的儲存屬性上，也可以添加到從父類別繼承的屬性上。

儲存屬性

簡單來說，一個儲存屬性就是儲存在特定類別或結構的實例裡的一個常數或變數，儲存屬性可以是變數儲存屬性（用關鍵字 `var` 定義），也可以是常數儲存屬性（用關鍵字 `let` 定義）。

可以在定義儲存屬性的時候指定預設值，請參考建構過程一章的預設屬性值一節。也可以在建構過程中設置或修改儲存屬性的值，甚至修改常數儲存屬性的值，請參考建構過程一章的在初始化階段修改常數儲存屬性一節。

下面的範例定義了一個名為 `FixedLengthRange` 的結構，它描述了一個在創建後無法修改值域寬度的區間：

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// 該區間表示整數0, 1, 2
rangeOfThreeItems.firstValue = 6
// 該區間現在表示整數6, 7, 8
```

`FixedLengthRange` 的實例包含一個名為 `firstValue` 的變數儲存屬性和一個名為 `length` 的常數儲存屬性。在上面的範例中， `length` 在創建實例的時候被賦值，因為它是一個常數儲存屬性，所以之後無法修改它的值。

常數和儲存屬性

如果創建了一個結構的實例並賦值給一個常數，則無法修改實例的任何屬性，即使定義了變數儲存屬性：

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// 該區間表示整數0, 1, 2, 3
rangeOfFourItems.firstValue = 6
// 儘管 firstValue 是個變數屬性，這裡還是會報錯
```

因為 `rangeOfFourItems` 宣告成了常數（用 `let` 關鍵字），即使 `firstValue` 是一個變數屬性，也無法再修改它了。

這種行為是由於結構（struct）屬於值型別。當值型別的實例被宣告為常數的時候，它的所有屬性也就成了常數。

屬於參考型別的類別（class）則不一樣，把一個參考型別的實例賦給一個常數後，仍然可以修改實例的變數屬性。

延遲儲存屬性

延遲儲存屬性是指當第一次被呼叫的時候才會計算其初始值的屬性。在屬性宣告前使用 `@lazy` 來標示一個延遲儲存屬性。

> 注意：
> 必須將延遲儲存屬性宣告成變數（使用 `var` 關鍵字），因為屬性的值在實例建構完成之前可能無法得到。而常數屬性在建構過程完成之前必須要有初始值，因此無法宣告成延遲屬性。

延遲屬性很有用，當屬性的值依賴於在實例的建構過程結束前無法知道具體值的外部因素時，或者當屬性的值需要複雜或大量計算時，可以只在需要的時候來計算它。

下面的範例使用了延遲儲存屬性來避免複雜類別的不必要的初始化。範例中定義了 `DataImporter` 和 `DataManager` 兩個類別，下面是部分程式碼：

```
class DataImporter {
    /*
    DataImporter 是一個將外部文件中的資料導入的類別。
    這個類別的初始化會消耗不少時間。
    */
    var fileName = "data.txt"
    // 這是提供資料導入功能
}

class DataManager {
    @lazy var importer = DataImporter()
    var data = String[]()
    // 這是提供資料管理功能
}

let manager = DataManager()
manager.data += "Some data"
manager.data += "Some more data"
// DataImporter 實例的 importer 屬性還沒有被創建
```

`DataManager` 類別包含一個名為 `data` 的儲存屬性，初始值是一個空的字串（ `String` ）陣列。雖然沒有寫出全部程式碼， `DataManager` 類別的目的是管理和提供對這個字串陣列的存取。

`DataManager` 的一個功能是從文件導入資料，該功能由 `DataImporter` 類別提供， `DataImporter` 需要消耗不少時間完成初始化：因為它的實例在初始化時可能要打開文件，還要讀取文件內容到內存。

`DataManager` 也可能不從文件中導入資料。所以當 `DataManager` 的實例被創建時，沒必要創建一個 `DataImporter` 的實例，更明智的是當用到 `DataImporter` 的時候才去創建它。

由於使用了 `@lazy` ， `importer` 屬性只有在第一次被存取的時候才被創建。比如存取它的屬性 `fileName` 時：

```
println(manager.importer.fileName)
// DataImporter 實例的 importer 屬性現在被創建了
// 輸出 "data.txt」
```

儲存屬性和實例變數

如果你有過 Objective-C 經驗，應該知道有兩種方式在類別實例儲存值和參考。對於屬性來說，也可以使用實例變數作為屬性值的後端儲存。

Swift 程式語言中把這些理論統一用屬性來實作。Swift 中的屬性沒有對應的實例變數，屬性的後端儲存也無法直接存取。這就避免了不同場景下存取方式的困擾，同時也將屬性的定義簡化成一個語句。 一個型別中屬性的全部資訊——包括命名、型別和內存管理特征——都在唯一一個地方（型別定義中）定義。

計算屬性

除儲存屬性外，類別、結構和列舉可以定義計算屬性，計算屬性不直接儲存值，而是提供一個 getter 來獲取值，一個可選的 setter 來間接設置其他屬性或變數的值。

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
    get {
        let centerX = origin.x + (size.width / 2)
        let centerY = origin.y + (size.height / 2)
        return Point(x: centerX, y: centerY)
    }
    set(newCenter) {
        origin.x = newCenter.x - (size.width / 2)
        origin.y = newCenter.y - (size.height / 2)
    }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
    size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
println("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// 輸出 "square.origin is now at (10.0, 10.0)」
```

這個範例定義了 3 個幾何形狀的結構：

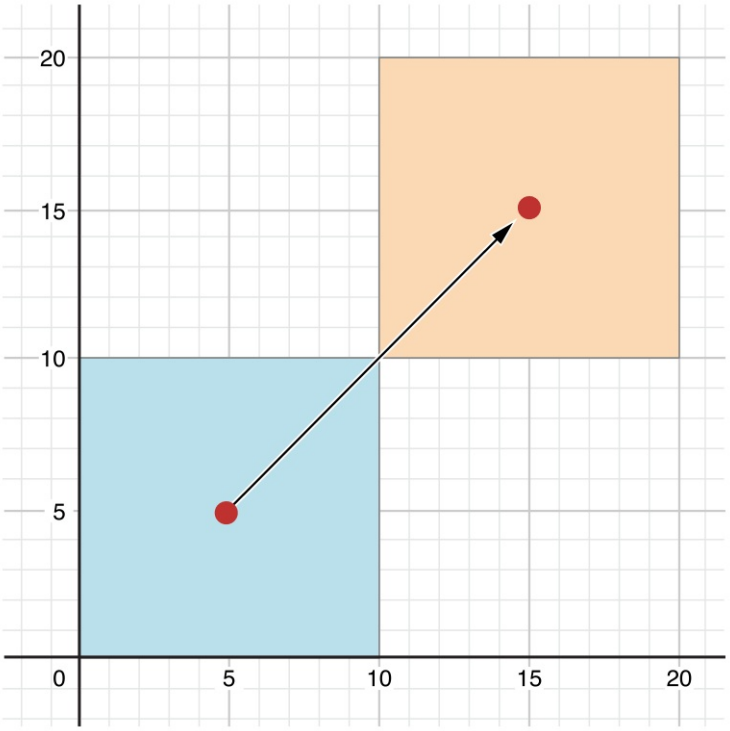
- `Point` 封裝了一個 `(x, y)` 的坐標
- `Size` 封裝了一個 `width` 和 `height`
- `Rect` 表示一個有原點和尺寸的矩形

`Rect` 也提供了一個名為 `center` 的計算屬性。一個矩形的中心點可以從原點和尺寸來算出，所以不需要將它以顯式宣告的 `Point` 來保存。 `Rect` 的計算屬性 `center` 提供了自定義的 getter 和 setter 來獲取和設置矩形的中心點，就像它有一個儲存屬性一樣。

範例中接下來創建了一個名為 `square` 的 `Rect` 實例，初始值原點是 `(0, 0)`，寬度高度都是 `10`。如圖所示藍色正方形。

`square` 的 `center` 屬性可以通過點運算子（ `square.center` ）來存取，這會呼叫 getter 來獲取屬性的值。跟直接回傳已經存在的值不同，getter 實際上通過計算然後回傳一個新的 `Point` 來表示 `square` 的中心點。如程式碼所示，它正確回傳了中心點 `(5, 5)`。

`center` 屬性之後被設置了一個新的值 `(15, 15)`，表示向右上方移動正方形到如圖所示橙色正方形的位置。設置屬性 `center` 的值會呼叫 setter 來修改屬性 `origin` 的 `x` 和 `y` 的值，從而實作移動正方形到新的位置。



便捷 setter 宣告

如果計算屬性的 setter 沒有定義表示新值的參數名，則可以使用預設名稱 `newValue`。下面是使用了便捷 setter 宣告的 `Rect` 結構程式碼：

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
    get {
        let centerX = origin.x + (size.width / 2)
        let centerY = origin.y + (size.height / 2)
        return Point(x: centerX, y: centerY)
    }
    set {
        origin.x = newValue.x - (size.width / 2)
        origin.y = newValue.y - (size.height / 2)
    }
    }
}
```

唯讀計算屬性

只有 getter 沒有 setter 的計算屬性就是唯讀計算屬性。唯讀計算屬性總是回傳一個值，可以通過點運算子存取，但不能設置新的值。

<<<<<<< HEAD

> 注意：
>
> 必須使用 `var` 關鍵字定義計算屬性，包括唯讀計算屬性，因為他們的值不是固定的。 `let` 關鍵字只用來宣告常數屬性，表示初始化後再也無法修改的值。

> 注意：
>
> 必須使用 `var` 關鍵字定義計算屬性，包括唯讀計算屬性，因為它們的值不是固定的。 `let` 關鍵字只用來宣告常數屬性，表示初始化後再也無法修改的值。
>
> a516af6a531a104ec88da0d236ecf389a5ec72af

唯讀計算屬性的宣告可以去掉 `get` 關鍵字和花括號：

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
    return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
println("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// 輸出 "the volume of fourByFiveByTwo is 40.0"
```

這個範例定義了一個名為 `Cuboid` 的結構，表示三維空間的立方體，包含 `width` 、 `height` 和 `depth` 屬性，還有一個名為 `volume` 的唯讀計算屬性用來回傳立方體的體積。設置 `volume` 的值毫無意義，因為通過 `width` 、 `height` 和 `depth` 就能算出 `volume` 。然而， `Cuboid` 提供一個唯讀計算屬性來讓外部使用者直接獲取體積是很有用的。

屬性監視器

屬性監視器監控和響應屬性值的變化，每次屬性被設置值的時候都會呼叫屬性監視器，甚至新的值和現在的值相同的時候也不例外。

可以為除了延遲儲存屬性之外的其他儲存屬性添加屬性監視器，也可以通過重載屬性的方式為繼承的屬性（包括儲存屬性和計算屬性）添加屬性監視器。屬性重載請參考繼承一章的重載。

> 注意：
> 不需要為無法重載的計算屬性添加屬性監視器，因為可以通過 setter 直接監控和響應值的變化。

可以為屬性添加如下的一個或全部監視器：

- `willSet` 在設置新的值之前呼叫
- `didSet` 在新的值被設置之後立即呼叫

`willSet` 監視器會將新的屬性值作為固定參數傳入，在 `willSet` 的實作程式碼中可以為這個參數指定一個名稱，如果不指定則參數仍然可用，這時使用預設名稱 `newValue` 表示。

類似地， `didSet` 監視器會將舊的屬性值作為參數傳入，可以為該參數命名或者使用預設參數名 `oldValue` 。

<<<<<<< HEAD

> 注意：
>
> **`willSet`** 和 **`didSet`** 監視器在屬性初始化過程中不會被呼叫，他們只會當屬性的值在初始化之外的地方被設置時被呼叫。

> 注意：

> `willSet` 和 `didSet` 監視器在屬性初始化過程中不會被呼叫，它們只會當屬性的值在初始化之外的地方被設置時被呼叫。
>
> a516af6a531a104ec88da0d236ecf389a5ec72af

這裡是一個 `willSet` 和 `didSet` 的實際範例，其中定義了一個名為 `StepCounter` 的類別，用來統計當人步行時的總步數，可以跟計步器或其他日常鍛煉的統計裝置的輸入資料配合使用。

```
class StepCounter {
    var totalSteps: Int = 0 {
    willSet(newTotalSteps) {
        println("About to set totalSteps to \(newTotalSteps)")
    }
    didSet {
        if totalSteps > oldValue  {
            println("Added \(totalSteps - oldValue) steps")
        }
    }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

`StepCounter` 類別定義了一個 `Int` 型別的屬性 `totalSteps`，它是一個儲存屬性，包含 `willSet` 和 `didSet` 監視器。

當 `totalSteps` 設置新值的時候，它的 `willSet` 和 `didSet` 監視器都會被呼叫，甚至當新的值和現在的值完全相同也會呼叫。

範例中的 `willSet` 監視器將表示新值的參數自定義為 `newTotalSteps`，這個監視器只是簡單的將新的值輸出。

`didSet` 監視器在 `totalSteps` 的值改變後被呼叫，它把新的值和舊的值進行對比，如果總的步數增加了，就輸出一個訊息表示增加了多少步。 `didSet` 沒有提供自定義名稱，所以預設值 `oldValue` 表示舊值的參數名。

> 注意：
> 如果在 `didSet` 監視器裡為屬性賦值，這個值會替換監視器之前設置的值。

全域變數和局部變數

計算屬性和屬性監視器所描述的模式也可以用於全域變數和局部變數，全域變數是在函式、方法、閉包或任何型別之外定義的變數，局部變數是在函式、方法或閉包內部定義的變數。

前面章節提到的全域或局部變數都屬於儲存型變數，跟儲存屬性類似，它提供特定型別的儲存空間，並允許讀取和寫入。

另外，在全域或局部範圍都可以定義計算型變數和為儲存型變數定義監視器，計算型變數跟計算屬性一樣，回傳一個計算的值而不是儲存值，宣告格式也完全一樣。

> 注意：
> 全域的常數或變數都是延遲計算的，跟延遲儲存屬性相似，不同的地方在於，全域的常數或變數不需要標記 `@lazy` 特性。
> 局部範圍的常數或變數不會延遲計算。

型別屬性

實例的屬性屬於一個特定型別實例，每次型別實例化後都擁有自己的一套屬性值，實例之間的屬性相互獨立。

也可以為型別本身定義屬性，不管型別有多少個實例，這些屬性都只有唯一一份。這種屬性就是型別屬性。

型別屬性用於定義特定型別所有實例共享的資料，比如所有實例都能用的一個常數（就像 C 語言中的靜態常數），或者所有實例都能存取的一個變數（就像 C 語言中的靜態變數）。

對於值型別（指結構和列舉）可以定義儲存型和計算型型別屬性，對於類別（class）則只能定義計算型型別屬性。

值型別的儲存型型別屬性可以是變數或常數，計算型型別屬性跟實例的計算屬性一樣定義成變數屬性。

> 注意：
> 跟實例的儲存屬性不同，必須給儲存型型別屬性指定預設值，因為型別本身無法在初始化過程中使用建構器給型別屬性賦值。

型別屬性語法

在 C 或 Objective-C 中，靜態常數和靜態變數的定義是通過特定型別加上 `global` 關鍵字。在 Swift 程式語言中，型別屬性是作為型別定義的一部分寫在型別最外層的花括號內，因此它的作用範圍也就在型別支援的範圍內。

使用關鍵字 `static` 來定義值型別的型別屬性，關鍵字 `class` 來為類別（class）定義型別屬性。下面的範例演示了儲存型和計算型型別屬性的語法：

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
    // 這裡回傳一個 Int 值
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
    // 這裡回傳一個 Int 值
    }
}
class SomeClass {
    class var computedTypeProperty: Int {
    // 這裡回傳一個 Int 值
    }
}
```

> 注意：
> 範例中的計算型型別屬性是唯讀的，但也可以定義可讀可寫的計算型型別屬性，跟實例計算屬性的語法類似。

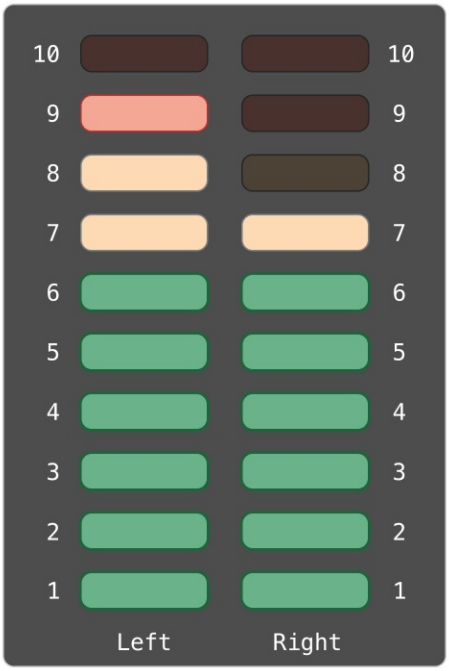
獲取和設置型別屬性的值

跟實例的屬性一樣，型別屬性的存取也是通過點運算子來進行，但是，型別屬性是通過型別本身來獲取和設置，而不是通過實例。比如：

```
println(SomeClass.computedTypeProperty)
// 輸出 "42"

println(SomeStructure.storedTypeProperty)
// 輸出 "Some value."
SomeStructure.storedTypeProperty = "Another value."
println(SomeStructure.storedTypeProperty)
// 輸出 "Another value.」
```

下面的範例定義了一個結構，使用兩個儲存型型別屬性來表示多個聲道的聲音電平值，每個聲道有一個 0 到 10 之間的整數表示聲音電平值。

後面的圖表展示了如何聯合使用兩個聲道來表示一個立體聲的聲音電平值。當聲道的電平值是 0，沒有一個燈會亮；當聲道的電平值是 10，所有燈點亮。本圖中，左聲道的電平是 9，右聲道的電平是 7。



上面所描述的聲道模型使用 `AudioChannel` 結構來表示：

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
    didSet {
        if currentLevel > AudioChannel.thresholdLevel {
            // 將新電平值設置為閥值
            currentLevel = AudioChannel.thresholdLevel
        }
        if currentLevel > AudioChannel.maxInputLevelForAllChannels {
            // 儲存當前電平值作為新的最大輸入電平
            AudioChannel.maxInputLevelForAllChannels = currentLevel
        }
    }
    }
}
```

結構 `AudioChannel` 定義了 2 個儲存型型別屬性來實作上述功能。第一個是 `thresholdLevel`，表示聲音電平的最大上限閾值，它是一個取值為 10 的常數，對所有實例都可見，如果聲音電平高於 10，則取最大上限值 10（見後面描述）。

第二個型別屬性是變數儲存型屬性 `maxInputLevelForAllChannels`，它用來表示所有 `AudioChannel` 實例的電平值的最大值，初始值是 0。

`AudioChannel` 也定義了一個名為 `currentLevel` 的實例儲存屬性，表示當前聲道現在的電平值，取值為 0 到 10。

屬性 `currentLevel` 包含 `didSet` 屬性監視器來檢查每次新設置後的屬性值，有如下兩個檢查：

- 如果 `currentLevel` 的新值大於允許的閾值 `thresholdLevel`，屬性監視器將 `currentLevel` 的值限定為閾值 `thresholdLevel`。
- 如果修正後的 `currentLevel` 值大於任何之前任意 `AudioChannel` 實例中的值，屬性監視器將新值保存在靜態屬

性 `maxInputLevelForAllChannels` 中。

> 注意：
> 在第一個檢查過程中，`didSet` 屬性監視器將 `currentLevel` 設置成了不同的值，但這時不會再次呼叫屬性監視器。

可以使用結構 `AudioChannel` 來創建表示立體聲系統的兩個聲道 `leftChannel` 和 `rightChannel`：

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

如果將左聲道的電平設置成 7，型別屬性 `maxInputLevelForAllChannels` 也會更新成 7：

```
leftChannel.currentLevel = 7
println(leftChannel.currentLevel)
// 輸出 "7"
println(AudioChannel.maxInputLevelForAllChannels)
// 輸出 "7"
```

如果試圖將右聲道的電平設置成 11，則會將右聲道的 `currentLevel` 修正到最大值 10，同時 `maxInputLevelForAllChannels` 的值也會更新到 10：

```
rightChannel.currentLevel = 11
println(rightChannel.currentLevel)
// 輸出 "10"
println(AudioChannel.maxInputLevelForAllChannels)
// 輸出 "10"
```

> 翻譯：pp-prog 校對：zqp

方法（Methods）

本頁包含內容：

- 實例方法(Instance Methods
- 型別方法(Type Methods)

方法是與某些特定型別相關聯的函式。類別、結構、列舉都可以定義實例方法；實例方法為給定型別的實例封裝了具體的任務與功能。類別、結構、列舉也可以定義型別方法；型別方法與型別本身相關聯。型別方法與 Objective-C 中的類別方法（class methods）相似。

結構和列舉能夠定義方法是 Swift 與 C/Objective-C 的主要區別之一。在 Objective-C 中，類別是唯一能定義方法的型別。但在 Swift 中，你不僅能選擇是否要定義一個類別/結構/列舉，還能靈活的在你創建的型別（類別/結構/列舉）上定義方法。

實例方法(Instance Methods)

實例方法是屬於某個特定類別、結構或者列舉型別實例的方法。實例方法提供存取和修改實例屬性的方法或提供與實例目的相關的功能，並以此來支撐實例的功能。實例方法的語法與函式完全一致，詳情參見函式。

實例方法要寫在它所屬的型別的前後大括號之間。實例方法能夠隱式存取它所屬型別的所有的其他實例方法和屬性。實例方法只能被它所屬的類別的某個特定實例呼叫。實例方法不能脫離於現存的實例而被呼叫。

下面的範例，定義一個很簡單的類別 `Counter`， `Counter` 能被用來對一個動作發生的次數進行計數：

```
class Counter {
  var count = 0
  func increment() {
    count++
  }
  func incrementBy(amount: Int) {
    count += amount
  }
  func reset() {
    count = 0
  }
}
```

`Counter` 類別定義了三個實例方法：

- `increment` 讓計數器按一遞增；
- `incrementBy(amount: Int)` 讓計數器按一個指定的整數值遞增；
- `reset` 將計數器重置為0。

`Counter` 這個類別還宣告了一個可變屬性 `count`，用它來保持對當前計數器值的追蹤。

和呼叫屬性一樣，用點語法（dot syntax）呼叫實例方法：

```
let counter = Counter()
// 初始計數值是0
counter.increment()
// 計數值現在是1
counter.incrementBy(5)
// 計數值現在是6
```

```
counter.reset()
// 計數值現在是0
```

方法的局部參數名稱和外部參數名稱(Local and External Parameter Names for Methods)

函式參數可以同時有一個局部名稱（在函式體內部使用）和一個外部名稱（在呼叫函式時使用），詳情參見函式的外部參數名。方法參數也一樣（因為方法就是函式，只是這個函式與某個型別相關聯了）。但是，方法和函式的局部名稱和外部名稱的預設行為是不一樣的。

Swift 中的方法和 Objective-C 中的方法極其相似。像在 Objective-C 中一樣，Swift 中方法的名稱通常用一個介詞指向方法的第一個參數，比如：`with`，`for`，`by`等等。前面的`Counter`類別的範例中`incrementBy`方法就是這樣的。介詞的使用讓方法在被呼叫時能像一個句子一樣被解讀。和函式參數不同，對於方法的參數，Swift 使用不同的預設處理方式，這可以讓方法命名規範更容易寫。

具體來說，Swift 預設僅給方法的第一個參數名稱一個局部參數名稱;預設同時給第二個和後續的參數名稱局部參數名稱和外部參數名稱。這個約定與典型的命名和呼叫約定相適應，與你在寫 Objective-C 的方法時很相似。這個約定還讓表達式方法在呼叫時不需要再限定參數名稱。

看看下面這個`Counter`的另一個版本（它定義了一個更複雜的`incrementBy`方法）：

```
class Counter {
  var count: Int = 0
  func incrementBy(amount: Int, numberOfTimes: Int) {
    count += amount * numberOfTimes
  }
}
```

`incrementBy`方法有兩個參數：`amount`和`numberOfTimes`。預設情況下，Swift 只把`amount`當作一個局部名稱，但是把`numberOfTimes`即看作局部名稱又看作外部名稱。下面呼叫這個方法：

```
let counter = Counter()
counter.incrementBy(5, numberOfTimes: 3)
// counter value is now 15
```

你不必為第一個參數值再定義一個外部變數名：因為從函式名`incrementBy`已經能很清楚地看出它的作用。但是第二個參數，就要被一個外部參數名稱所限定，以便在方法被呼叫時明確它的作用。

這種預設的行為能夠有效的處理方法（method）,類似於在參數`numberOfTimes`前寫一個井字號（`#`）：

```
func incrementBy(amount: Int, #numberOfTimes: Int) {
  count += amount * numberOfTimes
}
```

這種預設行為使上面程式碼意味著：在 Swift 中定義方法使用了與 Objective-C 同樣的語法風格，並且方法將以自然表達式的方式被呼叫。

修改方法的外部參數名稱(Modifying External Parameter Name Behavior for Methods)

有時為方法的第一個參數提供一個外部參數名稱是非常有用的，儘管這不是預設的行為。你可以自己添加一個顯式的外部名稱或者用一個井字號（`#`）作為第一個參數的前綴來把這個局部名稱當作外部名稱使用。

相反，如果你不想為方法的第二個及後續的參數提供一個外部名稱，可以通過使用底線（`_`）作為該參數的顯式外部名稱，這樣做將覆蓋預設行為。

`self` 屬性(The self Property)

型別的每一個實例都有一個隱含屬性叫做 `self`，`self` 完全等同於該實例本身。你可以在一個實例的實例方法中使用這個隱含的 `self` 屬性來參考當前實例。

上面範例中的 `increment` 方法還可以這樣寫：

```
func increment() {
  self.count++
}
```

實際上，你不必在你的程式碼裡面經常寫 `self`。不論何時，只要在一個方法中使用一個已知的屬性或者方法名稱，如果你沒有明確的寫 `self`，Swift 假定你是指當前實例的屬性或者方法。這種假定在上面的 `Counter` 中已經示範了：`Counter` 中的三個實例方法中都使用的是 `count`（而不是 `self.count`）。

使用這條規則的主要場景是實例方法的某個參數名稱與實例的某個屬性名稱相同的時候。在這種情況下，參數名稱享有優先權，並且在參考屬性時必須使用一種更嚴格的方式。這時你可以使用 `self` 屬性來區分參數名稱和屬性名稱。

下面的範例中，`self` 消除方法參數 `x` 和實例屬性 `x` 之間的歧義：

```
struct Point {
  var x = 0.0, y = 0.0
  func isToTheRightOfX(x: Double) -> Bool {
    return self.x > x
  }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOfX(1.0) {
  println("This point is to the right of the line where x == 1.0")
}
// 輸出 "This point is to the right of the line where x == 1.0"（這個點在x等於1.0這條線的右邊）
```

如果不使用 `self` 前綴，Swift 就認為兩次使用的 `x` 都指的是名稱為 `x` 的函式參數。

在實例方法中修改值型別(Modifying Value Types from Within Instance Methods)

結構和列舉是值型別。一般情況下，值型別的屬性不能在它的實例方法中被修改。

但是，如果你確實需要在某個具體的方法中修改結構或者列舉的屬性，你可以選擇 `變異(mutating)` 這個方法，然後方法就可以從方法內部改變它的屬性；並且它做的任何改變在方法結束時還會保留在原始結構中。方法還可以給它隱含的 `self` 屬性賦值一個全新的實例，這個新實例在方法結束後將替換原來的實例。

要使用 `變異` 方法，將關鍵字 `mutating` 放到方法的 `func` 關鍵字之前就可以了：

```
struct Point {
  var x = 0.0, y = 0.0
  mutating func moveByX(deltaX: Double, y deltaY: Double) {
    x += deltaX
    y += deltaY
  }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveByX(2.0, y: 3.0)
println("The point is now at (\(somePoint.x), \(somePoint.y))")
```

```
// 輸出 "The point is now at (3.0, 4.0)"
```

上面的 `Point` 結構定義了一個變異方法（mutating method） `moveByX`， `moveByX` 用來移動點。 `moveByX` 方法在被呼叫時修改了這個點，而不是回傳一個新的點。方法定義時加上 `mutating` 關鍵字,這才讓方法可以修改值型別的屬性。

注意：不能在結構型別常數上呼叫變異方法，因為常數的屬性不能被改變，即使想改變的是常數的變數屬性也不行，詳情參見儲存屬性和實例變數

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveByX(2.0, y: 3.0)
// this will report an error
```

在變異方法中給self賦值(Assigning to self Within a Mutating Method)

變異方法能夠賦給隱含屬性 `self` 一個全新的實例。上面 `Point` 的範例可以用下面的方式改寫：

```
struct Point {
  var x = 0.0, y = 0.0
  mutating func moveByX(deltaX: Double, y deltaY: Double) {
    self = Point(x: x + deltaX, y: y + deltaY)
  }
}
```

新版的變異方法 `moveByX` 創建了一個新的結構（它的 x 和 y 的值都被設定為目標值）。呼叫這個版本的方法和呼叫上個版本的最終結果是一樣的。

列舉的變異方法可以把 `self` 設置為相同的列舉型別中不同的成員：

```
enum TriStateSwitch {
  case Off, Low, High
  mutating func next() {
    switch self {
    case Off:
      self = Low
    case Low:
      self = High
    case High:
      self = Off
    }
  }
}
var ovenLight = TriStateSwitch.Low
ovenLight.next()
// ovenLight 現在等於 .High
ovenLight.next()
// ovenLight 現在等於 .Off
```

上面的範例中定義了一個三態開關的列舉。每次呼叫 `next` 方法時，開關在不同的電源狀態（ `Off`， `Low`， `High` ）之前迴圈切換。

型別方法(Type Methods)

實例方法是被型別的某個實例呼叫的方法。你也可以定義型別本身呼叫的方法，這種方法就叫做型別方法。宣告類別的型別方法，在方法的 `func` 關鍵字之前加上關鍵字 `class` ；宣告結構和列舉的型別方法，在方法的 `func` 關鍵字之前加上關鍵字 `static` 。

> 注意：

> 在 Objective-C 裡面，你只能為 Objective-C 的類別定義型別方法（type-level methods）。在 Swift 中，你可以為所有的類別、結構和列舉定義型別方法：每一個型別方法都被它所支援的型別顯式包含。

型別方法和實例方法一樣用點語法呼叫。但是，你是在型別層面上呼叫這個方法，而不是在實例層面上呼叫。下面是如何在 `SomeClass` 類別上呼叫型別方法的範例：

```
class SomeClass {
  class func someTypeMethod() {
    // type method implementation goes here
  }
}
SomeClass.someTypeMethod()
```

在型別方法的方法體（body）中， `self` 指向這個型別本身，而不是型別的某個實例。對於結構和列舉來說，這意味著你可以用 `self` 來消除靜態屬性和靜態方法參數之間的歧義（類似於我們在前面處理實例屬性和實例方法參數時做的那樣）。

一般來說，任何未限定的方法和屬性名稱，將會來自於本類別中另外的型別級別的方法和屬性。一個型別方法可以呼叫本類別中另一個型別方法的名稱，而無需在方法名稱前面加上型別名稱的前綴。同樣，結構和列舉的型別方法也能夠直接通過靜態屬性的名稱存取靜態屬性，而不需要型別名稱前綴。

下面的範例定義了一個名為 `LevelTracker` 結構。它監測玩家的遊戲發展情況（遊戲的不同層次或階段）。這是一個單人遊戲，但也可以儲存多個玩家在同一設備上的遊戲資訊。

遊戲初始時，所有的遊戲等級（除了等級 1）都被鎖定。每次有玩家完成一個等級，這個等級就對這個設備上的所有玩家解鎖。 `LevelTracker` 結構用靜態屬性和方法監測遊戲的哪個等級已經被解鎖。它還監測每個玩家的當前等級。

```
struct LevelTracker {
  static var highestUnlockedLevel = 1
  static func unlockLevel(level: Int) {
    if level > highestUnlockedLevel { highestUnlockedLevel = level }
  }
  static func levelIsUnlocked(level: Int) -> Bool {
    return level <= highestUnlockedLevel
  }
  var currentLevel = 1
  mutating func advanceToLevel(level: Int) -> Bool {
    if LevelTracker.levelIsUnlocked(level) {
      currentLevel = level
      return true
    } else {
      return false
    }
  }
}
```

`LevelTracker` 監測玩家的已解鎖的最高等級。這個值被儲存在靜態屬性 `highestUnlockedLevel` 中。

`LevelTracker` 還定義了兩個型別方法與 `highestUnlockedLevel` 配合工作。第一個型別方法是 `unlockLevel` ：一旦新等級被解鎖，它會更新 `highestUnlockedLevel` 的值。第二個型別方法是 `levelIsUnlocked` ：如果某個給定的等級已經被解鎖，它將回傳 `true` 。（注意：儘管我們沒有使用類似 `LevelTracker.highestUnlockedLevel` 的寫法，這個型別方法還是能夠存取靜態屬性 `highestUnlockedLevel` ）

除了靜態屬性和型別方法， `LevelTracker` 還監測每個玩家的進度。它用實例屬性 `currentLevel` 來監測玩家當前的等級。

為了便於管理 `currentLevel` 屬性， `LevelTracker` 定義了實例方法 `advanceToLevel` 。這個方法會在更新 `currentLevel` 之前檢查所請求的新等級是否已經解鎖。 `advanceToLevel` 方法回傳布林值以指示是否能夠設置 `currentLevel` 。

下面， `Player` 類別使用 `LevelTracker` 來監測和更新每個玩家的發展進度：

```
class Player {
  var tracker = LevelTracker()
  let playerName: String
  func completedLevel(level: Int) {
    LevelTracker.unlockLevel(level + 1)
    tracker.advanceToLevel(level + 1)
  }
  init(name: String) {
    playerName = name
  }
}
```

`Player`類別創建一個新的`LevelTracker`實例來監測這個使用者的發展進度。它提供了`completedLevel`方法：一旦玩家完成某個指定等級就呼叫它。這個方法為所有玩家解鎖下一等級，並且將當前玩家的進度更新為下一等級。（我們忽略了`advanceToLevel`回傳的布林值，因為之前呼叫`LevelTracker.unlockLevel`時就知道了這個等級已經被解鎖了）。

你還可以為一個新的玩家創建一個`Player`的實例，然後看這個玩家完成等級一時發生了什麼：

```
var player = Player(name: "Argyrios")
player.completedLevel(1)
println("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// 輸出 "highest unlocked level is now 2"（最高等級現在是2）
```

如果你創建了第二個玩家，並嘗試讓它開始一個沒有被任何玩家解鎖的等級，那麼這次設置玩家當前等級的嘗試將會失敗：

```
player = Player(name: "Beto")
if player.tracker.advanceToLevel(6) {
  println("player is now on level 6")
} else {
  println("level 6 has not yet been unlocked")
}
// 輸出 "level 6 has not yet been unlocked"（等級6還沒被解鎖）
```

> 翻譯：siemenliu 校對：zq54zquan

下標腳本（Subscripts）

本頁包含內容：

- 下標腳本語法
- 下標腳本用法
- 下標腳本選項

下標腳本 可以定義在類別（Class）、結構（structure）和列舉（enumeration）這些目標中，可以認為是存取物件、集合或序列的快捷方式，不需要再呼叫實例的特定的賦值和存取方法。舉例來說，用下標腳本存取一個陣列(Array)實例中的元素可以這樣寫 `someArray[index]` ，存取字典(Dictionary)實例中的元素可以這樣寫 `someDictionary[key]` 。

對於同一個目標可以定義多個下標腳本，通過索引值型別的不同來進行重載，而且索引值的個數可以是多個。

> 譯者：這裡附屬腳本重載在本小節中原文並沒有任何演示

下標腳本語法

下標腳本允許你通過在實例後面的方括號中傳入一個或者多個的索引值來對實例進行存取和賦值。語法類似於實例方法和計算型屬性的混合。與定義實例方法類似，定義下標腳本使用 `subscript` 關鍵字，顯式宣告入參（一個或多個）和回傳型別。與實例方法不同的是下標腳本可以設定為讀寫或唯讀。這種方式又有點像計算型屬性的getter和setter：

```
subscript(index: Int) -> Int {
    get {
      // 回傳與入參匹配的Int型別的值
    }

    set(newValue) {
      // 執行賦值操作
    }
}
```

`newValue` 的型別必須和下標腳本定義的回傳型別相同。與計算型屬性相同的是set的入參宣告 `newValue` 就算不寫，在set程式碼區塊中依然可以使用預設的 `newValue` 這個變數來存取新賦的值。

與唯讀計算型屬性一樣，可以直接將原本應該寫在 `get` 程式碼區塊中的程式碼寫在 `subscript` 中：

```
subscript(index: Int) -> Int {
    // 回傳與入參匹配的Int型別的值
}
```

下面程式碼演示了一個在 `TimesTable` 結構中使用唯讀下標腳本的用法，該結構用來展示傳入整數的*n*倍。

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
      return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
println("3的6倍是\(threeTimesTable[6])")
// 輸出 "3的6倍是18"
```

在上例中，通過 `TimesTable` 結構創建了一個用來表示索引值三倍的實例。數值 `3` 作為結構 `建構函式` 入參初始化實例成員 `multiplier`。

你可以通過下標腳本來得到結果，比如 `threeTimesTable[6]`。這條語句存取了 `threeTimesTable` 的第六個元素，回傳 `6` 的 `3` 倍即 `18`。

> 注意：
> `TimesTable` 範例是基於一個固定的數學公式。它並不適合開放寫權限來對 `threeTimesTable[someIndex]` 進行賦值操作，這也是為什麼附屬腳本只定義為唯讀的原因。

下標腳本用法

根據使用場景不同下標腳本也具有不同的含義。通常下標腳本是用來存取集合（collection），列表（list）或序列（sequence）中元素的快捷方式。你可以在你自己特定的類別或結構中自由的實作下標腳本來提供合適的功能。

例如，Swift 的字典（Dictionary）實作了通過下標腳本來對其實例中存放的值進行存取操作。在下標腳本中使用和字典索引相同型別的值，並且把一個字典值型別的值賦值給這個下標腳本來為字典設值：

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

上例定義一個名為 `numberOfLegs` 的變數並用一個字典字面量初始化出了包含三對鍵值的字典實例。`numberOfLegs` 的字典存放值型別推斷為 `Dictionary<String, Int>`。字典實例創建完成之後通過下標腳本的方式將整型值 `2` 賦值到字典實例的索引為 `bird` 的位置中。

更多關於字典（Dictionary）下標腳本的資訊請參考讀取和修改字典

> 注意：
> Swift 中字典的附屬腳本實作中，在 `get` 部分回傳值是 `Int?`，上例中的 `numberOfLegs` 字典通過附屬腳本回傳的是一個 `Int?` 或者說「可選的int」，不是每個字典的索引都能得到一個整型值，對於沒有設過值的索引的存取回傳的結果就是 `nil`；同樣想要從字典實例中刪除某個索引下的值也只需要給這個索引賦值為 `nil` 即可。

下標腳本選項

下標腳本允許任意數量的入參索引，並且每個入參型別也沒有限制。下標腳本的回傳值也可以是任何型別。下標腳本可以使用變數參數和可變參數，但使用寫入讀出（in-out）參數或給參數設置預設值都是不允許的。

一個類別或結構可以根據自身需要提供多個下標腳本實作，在定義下標腳本時通過入參個型別進行區分，使用下標腳本時會自動匹配合適的下標腳本實作執行，這就是下標腳本的重載。

一個下標腳本入參是最常見的情況，但只要有合適的場景也可以定義多個下標腳本入參。如下例定義了一個 `Matrix` 結構，將呈現一個 `Double` 型別的二維矩陣。`Matrix` 結構的下標腳本需要兩個整型參數：

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: Double[]
    init(rows: Int, columns: Int) {
      self.rows = rows
      self.columns = columns
      grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func indexIsValidForRow(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
```

```
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValidForRow(row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValidForRow(row, column: column), "Index out of range")
            grid[(row * columns) + columns] = newValue
        }
    }
}
```

`Matrix` 提供了一個兩個入參的建構方法，入參分別是 `rows` 和 `columns`，創建了一個足夠容納 `rows * columns` 個數的 `Double` 型別陣列。為了儲存，將陣列的大小和陣列每個元素初始值0.0，都傳入陣列的建構方法中來創建一個正確大小的新陣列。關於陣列的建構方法和析構方法請參考創建並且建構一個陣列。

你可以通過傳入合適的 `row` 和 `column` 的數量來建構一個新的 `Matrix` 實例：

```
var matrix = Matrix(rows: 2, columns: 2)
```

上例中創建了一個新的兩行兩列的 `Matrix` 實例。在閱讀順序從左上到右下的 `Matrix` 實例中的陣列實例 `grid` 是矩陣二維陣列的扁平化儲存：

```
// 示意圖
grid = [0.0, 0.0, 0.0, 0.0]

      col0  col1
row0   [0.0,     0.0,
row1    0.0,  0.0]
```

將值賦給帶有 `row` 和 `column` 下標腳本的 `matrix` 實例表達式可以完成賦值操作，下標腳本入參使用逗號分割

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

上面兩條語句分別讓`matrix` 的右上值為 1.5，坐下值為 3.2：

```
[0.0, 1.5,
 3.2, 0.0]
```

`Matrix` 下標腳本的 `getter` 和 `setter` 中同時呼叫了下標腳本入參的 `row` 和 `column` 是否有效的判斷。為了方便進行斷言， `Matrix` 包含了一個名為 `indexIsValid` 的成員方法，用來確認入參的 `row` 或 `column` 值是否會造成陣列越界：

```
func indexIsValidForRow(row: Int, column: Int) -> Bool {
    return row >= 0 && row < rows && column >= 0 && column < columns
}
```

斷言在下標腳本越界時觸發：

```
let someValue = matrix[2, 2]
// 斷言將會觸發，因為 [2, 2] 已經超過了matrix的最大長度
```

> 翻譯：Hawstein 校對：menlongsheng

繼承（Inheritance）

本頁包含內容：

一個類別可以繼承（*inherit*）另一個類別的方法（methods），屬性（property）和其它特性。當一個類別繼承其它類別時，繼承類別叫子類別（*subclass*），被繼承類別叫超類別（或父類別，*superclass*）。在 Swift 中，繼承是區分「類別」與其它型別的一個基本特征。

在 Swift 中，類別可以呼叫和存取超類別的方法，屬性和下標腳本（subscripts），並且可以重寫（override）這些方法，屬性和下標腳本來優化或修改它們的行為。Swift 會檢查你的重寫定義在超類別中是否有匹配的定義，以此確保你的重寫行為是正確的。

可以為類別中繼承來的屬性添加屬性觀察器（property observer），這樣一來，當屬性值改變時，類別就會被通知到。可以為任何屬性添加屬性觀察器，無論它原本被定義為儲存型屬性（stored property）還是計算型屬性（computed property）。

定義一個基類別（Base class）

不繼承於其它類別的類別，稱之為基類別（*base calss*）。

> 注意：
> Swift 中的類別並不是從一個通用的基類別繼承而來。如果你不為你定義的類別指定一個超類別的話，這個類別就自動成為基類別。

下面的範例定義了一個叫 `Vehicle` 的基類別。這個基類別宣告了兩個對所有車輛都通用的屬性（`numberOfWheels` 和 `maxPassengers`）。這些屬性在 `description` 方法中使用，這個方法回傳一個 `String` 型別的，對車輛特征的描述：

```
class Vehicle {
    var numberOfWheels: Int
    var maxPassengers: Int
    func description() -> String {
        return "\(numberOfWheels) wheels; up to \(maxPassengers) passengers"
    }
    init() {
        numberOfWheels = 0
        maxPassengers = 1
    }
}
```

`Vehicle` 類別定義了建構器（*initializer*）來設置屬性的值。建構器會在建構過程一節中詳細介紹，這裡我們做一下簡單介紹，以便於講解子類別中繼承來的屬性如何被修改。

建構器用於創建某個型別的一個新實例。儘管建構器並不是方法，但在語法上，兩者很相似。建構器的工作是準備新實例以供使用，並確保實例中的所有屬性都擁有有效的初始化值。

建構器的最簡單形式就像一個沒有參數的實例方法，使用 `init` 關鍵字：

```
init() {
    // 執行建構過程
}
```

如果要創建一個 `Vehicle` 類別的新實例，使用建構器語法呼叫上面的初始化器，即類別名後面跟一個空的小括號：

```
let someVehicle = Vehicle()
```

這個 `Vehicle` 類別的建構器為任意的一輛車設置一些初始化屬性值（ `numberOfWheels = 0` 和 `maxPassengers = 1` ）。

`Vehicle` 類別定義了車輛的共同特性，但這個類別本身並沒太大用處。為了使它更為實用，你需要進一步細化它來描述更具體的車輛。

子類別生成（Subclassing）

子類別生成（*Subclassing*）指的是在一個已有類別的基礎上創建一個新的類別。子類別繼承超類別的特性，並且可以優化或改變它。你還可以為子類別添加新的特性。

為了指明某個類別的超類別，將超類別名寫在子類別名的後面，用冒號分隔：

```
class SomeClass: SomeSuperclass {
    // 類別的定義
}
```

下一個範例，定義一個更具體的車輛類別叫 `Bicycle` 。這個新類別是在 `Vehicle` 類別的基礎上創建起來。因此你需要將 `Vehicle` 類別放在 `Bicycle` 類別後面，用冒號分隔。

我們可以將這讀作：

「定義一個新的類別叫 `Bicycle` ，它繼承了 `Vehicle` 的特性」；

```
class Bicycle: Vehicle {
    init() {
        super.init()
        numberOfWheels = 2
    }
}
```

preview `Bicycle` 是 `Vehicle` 的子類別， `Vehicle` 是 `Bicycle` 的超類別。新的 `Bicycle` 類別自動獲得 `Vehicle` 類別的特性，比如 `maxPassengers` 和 `numberOfWheels` 屬性。你可以在子類別中定制這些特性，或添加新的特性來更好地描述 `Bicycle` 類別。

`Bicycle` 類別定義了一個建構器來設置它定制的特性（自行車只有2個輪子）。 `Bicycle` 的建構器呼叫了它父類別 `Vehicle` 的建構器 `super.init()` ，以此確保在 `Bicycle` 類別試圖修改那些繼承來的屬性前 `Vehicle` 類別已經初始化過它們了。

> 注意：
> 不像 Objective-C，在 Swift 中，初始化器預設是不繼承的，見初始化器的繼承與重寫

`Vehicle` 類別中 `maxPassengers` 的預設值對自行車來說已經是正確的，因此在 `Bicycle` 的建構器中並沒有改變它。
而 `numberOfWheels` 原來的值對自行車來說是不正確的，因此在初始化器中將它更改為 2。

`Bicycle` 不僅可以繼承 `Vehicle` 的屬性，還可以繼承它的方法。如果你創建了一個 `Bicycle` 類別的實例，你就可以呼叫它繼承來的 `description` 方法，並且可以看到，它輸出的屬性值已經發生了變化：

```
let bicycle = Bicycle()
println("Bicycle: \(bicycle.description())")
// Bicycle: 2 wheels; up to 1 passengers
```

子類別還可以繼續被其它類別繼承：

```
class Tandem: Bicycle {
    init() {
        super.init()
        maxPassengers = 2
    }
}
```

上面的範例創建了 `Bicycle` 的一個子類別：雙人自行車（tandem）。 `Tandem` 從 `Bicycle` 繼承了兩個屬性，而這兩個屬性是 `Bicycle` 從 `Vehicle` 繼承而來的。 `Tandem` 並不修改輪子的數量，因為它仍是一輛自行車，有 2 個輪子。但它需要修改 `maxPassengers` 的值，因為雙人自行車可以坐兩個人。

> 注意：
> 子類別只允許修改從超類別繼承來的變數屬性，而不能修改繼承來的常數屬性。

創建一個 `Tandem` 類別的實例，列印它的描述，即可看到它的屬性已被更新：

```
let tandem = Tandem()
println("Tandem: \(tandem.description())")
// Tandem: 2 wheels; up to 2 passengers
```

注意， `Tandem` 類別也繼承了 `description` 方法。一個類別的實例方法會被這個類別的所有子類別繼承。

重寫（Overriding）

子類別可以為繼承來的實例方法（instance method），類別方法（class method），實例屬性（instance property），或下標腳本（subscript）提供自己定制的實作（implementation）。我們把這種行為叫重寫（*overriding*）。

如果要重寫某個特性，你需要在重寫定義的前面加上 `override` 關鍵字。這麼做，你就表明了你是想提供一個重寫版本，而非錯誤地提供了一個相同的定義。意外的重寫行為可能會導致不可預知的錯誤，任何缺少 `override` 關鍵字的重寫都會在編譯時被診斷為錯誤。

`override` 關鍵字會提醒 Swift 編譯器去檢查該類別的超類別（或其中一個父類別）是否有匹配重寫版本的宣告。這個檢查可以確保你的重寫定義是正確的。

存取超類別的方法，屬性及下標腳本

當你在子類別中重寫超類別的方法，屬性或下標腳本時，有時在你的重寫版本中使用已經存在的超類別實作會大有裨益。比如，你可以優化已有實作的行為，或在一個繼承來的變數中儲存一個修改過的值。

在合適的地方，你可以通過使用 `super` 前綴來存取超類別版本的方法，屬性或下標腳本：

- 在方法 `someMethod` 的重寫實作中，可以通過 `super.someMethod()` 來呼叫超類別版本的 `someMethod` 方法。
- 在屬性 `someProperty` 的 getter 或 setter 的重寫實作中，可以通過 `super.someProperty` 來存取超類別版本的 `someProperty` 屬性。
- 在下標腳本的重寫實作中，可以通過 `super[someIndex]` 來存取超類別版本中的相同下標腳本。

重寫方法

在子類別中，你可以重寫繼承來的實例方法或類別方法，提供一個定制或替代的方法實作。

下面的範例定義了 `Vehicle` 的一個新的子類別，叫 `Car`，它重寫了從 `Vehicle` 類別繼承來的 `description` 方法：

```
class Car: Vehicle {
    var speed: Double = 0.0
    init() {
        super.init()
        maxPassengers = 5
        numberOfWheels = 4
    }
    override func description() -> String {
        return super.description() + "; "
            + "traveling at \(speed) mph"
    }
}
```

`Car` 宣告了一個新的儲存型屬性 `speed`，它是 `Double` 型別的，預設值是 `0.0`，表示「時速是0英裡」。 `Car` 有自己的初始化器，它將乘客的最大數量設為5，輪子數量設為4。

`Car` 重寫了繼承來的 `description` 方法，它的宣告與 `Vehicle` 中的 `description` 方法一致，宣告前面加上了 `override` 關鍵字。

`Car` 中的 `description` 方法並非完全自定義，而是通過 `super.description` 使用了超類別 `Vehicle` 中的 `description` 方法，然後再追加一些額外的資訊，比如汽車的當前速度。

如果你創建一個 `Car` 的新實例，並列印 `description` 方法的輸出，你就會發現描述資訊已經發生了改變：

```
let car = Car()
println("Car: \(car.description())")
// Car: 4 wheels; up to 5 passengers; traveling at 0.0 mph
```

重寫屬性

你可以重寫繼承來的實例屬性或類別屬性，提供自己定制的getter和setter，或添加屬性觀察器使重寫的屬性觀察屬性值什麼時候發生改變。

重寫屬性的Getters和Setters

你可以提供定制的 getter（或 setter）來重寫任意繼承來的屬性，無論繼承來的屬性是儲存型的還是計算型的屬性。子類別並不知道繼承來的屬性是儲存型的還是計算型的，它只知道繼承來的屬性會有一個名字和型別。你在重寫一個屬性時，必需將它的名字和型別都寫出來。這樣才能使編譯器去檢查你重寫的屬性是與超類別中同名同型別的屬性相匹配的。

你可以將一個繼承來的唯讀屬性重寫為一個讀寫屬性，只需要你在重寫版本的屬性裡提供 getter 和 setter 即可。但是，你不可以將一個繼承來的讀寫屬性重寫為一個唯讀屬性。

> 注意：
> 如果你在重寫屬性中提供了 setter，那麼你也一定要提供 getter。如果你不想在重寫版本中的 getter 裡修改繼承來的屬性值，你可以直接回傳 `super.someProperty` 來回傳繼承來的值。正如下面的 `SpeedLimitedCar` 的範例所示。

以下的範例定義了一個新類別，叫 `SpeedLimitedCar`，它是 `Car` 的子類別。類別 `SpeedLimitedCar` 表示安裝了限速裝置的車，它的最高速度只能達到40mph。你可以通過重寫繼承來的 `speed` 屬性來實作這個速度限制：

```
class SpeedLimitedCar: Car {
    override var speed: Double  {
    get {
        return super.speed
    }
    set {
        super.speed = min(newValue, 40.0)
    }
    }
}
```

當你設置一個 `SpeedLimitedCar` 實例的 `speed` 屬性時，屬性setter的實作會去檢查新值與限制值40mph的大小，它會將超類別的 `speed` 設置為 `newValue` 和 `40.0` 中較小的那個。這兩個值哪個較小由 `min` 函式決定，它是Swift標準函式庫中的一個全域函式。 `min` 函式接收兩個或更多的數，回傳其中最小的那個。

如果你嘗試將 `SpeedLimitedCar` 實例的 `speed` 屬性設置為一個大於40mph的數，然後列印 `description` 函式的輸出，你會發現速度被限制在40mph：

```
let limitedCar = SpeedLimitedCar()
limitedCar.speed = 60.0
println("SpeedLimitedCar: \(limitedCar.description())")
// SpeedLimitedCar: 4 wheels; up to 5 passengers; traveling at 40.0 mph
```

重寫屬性觀察器（Property Observer）

你可以在屬性重寫中為一個繼承來的屬性添加屬性觀察器。這樣一來，當繼承來的屬性值發生改變時，你就會被通知到，無論那個屬性原本是如何實作的。關於屬性觀察器的更多內容，請看屬性觀察器。

> 注意：
> 你不可以為繼承來的常數儲存型屬性或繼承來的唯讀計算型屬性添加屬性觀察器。這些屬性的值是不可以被設置的，所以，為它們提供 `willSet` 或 `didSet` 實作是不恰當。此外還要注意，你不可以同時提供重寫的 setter 和重寫的屬性觀察器。如果你想觀察屬性值的變化，並且你已經為那個屬性提供了定制的 setter，那麼你在 setter 中就可以觀察到任何值變化了。

下面的範例定義了一個新類別叫 `AutomaticCar`，它是 `Car` 的子類別。 `AutomaticCar` 表示自動擋汽車，它可以根據當前的速度自動選擇合適的擋位。 `AutomaticCar` 也提供了定制的 `description` 方法，可以輸出當前擋位。

```
class AutomaticCar: Car {
    var gear = 1
    override var speed: Double {
    didSet {
        gear = Int(speed / 10.0) + 1
    }
    }
    override func description() -> String {
        return super.description() + " in gear \(gear)"
    }
}
```

當你設置 `AutomaticCar` 的 `speed` 屬性，屬性的 `didSet` 觀察器就會自動地設置 `gear` 屬性，為新的速度選擇一個合適的擋位。具體來說就是，屬性觀察器將新的速度值除以10，然後向下取得最接近的整數值，最後加1來得到檔位 `gear` 的值。例如，速度為10.0時，擋位為1；速度為35.0時，擋位為4：

```
let automatic = AutomaticCar()
automatic.speed = 35.0
println("AutomaticCar: \(automatic.description())")
// AutomaticCar: 4 wheels; up to 5 passengers; traveling at 35.0 mph in gear 4
```

防止重寫

你可以通過把方法，屬性或下標腳本標記為 `final` 來防止它們被重寫，只需要在宣告關鍵字前加上 `@final` 特性即可。（例如：`@final var`，`@final func`，`@final class func`，以及 `@final subscript`）

如果你重寫了 `final` 方法，屬性或下標腳本，在編譯時會報錯。在擴展中，你添加到類別裡的方法，屬性或下標腳本也可以在擴展的定義裡標記為 final。

你可以通過在關鍵字 `class` 前添加 `@final` 特性（`@final class`）來將整個類別標記為 final 的，這樣的類別是不可被繼承的，否則會報編譯錯誤。

> 翻譯：lifedim 校對：lifedim

建構過程（Initialization）

本頁包含內容：

建構過程是為了使用某個類別、結構或列舉型別的實例而進行的準備過程。這個過程包含了為實例中的每個屬性設置初始值和為其執行必要的準備和初始化任務。

建構過程是通過定義建構器（`Initializers`）來實作的，這些建構器可以看做是用來創建特定型別實例的特殊方法。與 Objective-C 中的建構器不同，Swift 的建構器無需回傳值，它們的主要任務是保證新實例在第一次使用前完成正確的初始化。

類別實例也可以通過定義析構器（`deinitializer`）在類別實例釋放之前執行特定的清除工作。想了解更多關於析構器的內容，請參考析構過程。

儲存型屬性的初始賦值

類別和結構在實例創建時，必須為所有儲存型屬性設置合適的初始值。儲存型屬性的值不能處於一個未知的狀態。

你可以在建構器中為儲存型屬性賦初值，也可以在定義屬性時為其設置預設值。以下章節將詳細介紹這兩種方法。

> 注意：
> 當你為儲存型屬性設置預設值或者在建構器中為其賦值時，它們的值是被直接設置的，不會觸發任何屬性觀測器（`property observers`）。

建構器

建構器在創建某特定型別的新實例時呼叫。它的最簡形式類似於一個不帶任何參數的實例方法，以關鍵字 `init` 命名。

下面範例中定義了一個用來保存華氏溫度的結構 `Fahrenheit`，它擁有一個 `Double` 型別的儲存型屬性 `temperature`：

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
```

```
var f = Fahrenheit()
println("The default temperature is \(f.temperature)° Fahrenheit")
// 輸出 "The default temperature is 32.0° Fahrenheit」
```

這個結構定義了一個不帶參數的建構器 `init`，並在裡面將儲存型屬性 `temperature` 的值初始化為 `32.0`（華攝氏度下水的冰

點）。

預設屬性值

如前所述，你可以在建構器中為儲存型屬性設置初始值；同樣，你也可以在屬性宣告時為其設置預設值。

> 注意：
> 如果一個屬性總是使用同一個初始值，可以為其設置一個預設值。無論定義預設值還是在建構器中賦值，最終它們實作的效果是一樣的，只不過預設值跟屬性建構過程結合的更緊密。使用預設值能讓你的建構器更簡潔、更清晰，且能通過預設值自動推導出屬性的型別；同時，它也能讓你充分利用預設建構器、建構器繼承（後續章節將講到）等特性。

你可以使用更簡單的方式在定義結構 `Fahrenheit` 時為屬性 `temperature` 設置預設值：

```
struct Fahrenheit {
    var temperature = 32.0
}
```

定制化建構過程

你可以通過輸入參數和可選屬性型別來定制建構過程，也可以在建構過程中修改常數屬性。這些都將在後面章節中提到。

建構參數

你可以在定義建構器時提供建構參數，為其提供定制化建構所需值的型別和名字。建構器參數的功能和語法跟函式和方法參數相同。

下面範例中定義了一個包含攝氏度溫度的結構 `Celsius`。它定義了兩個不同的建構器：`init(fromFahrenheit:)` 和 `init(fromKelvin:)`，二者分別通過接受不同刻度表示的溫度值來創建新的實例：

```
struct Celsius {
    var temperatureInCelsius: Double = 0.0
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
```

```
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius 是 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius 是 0.0」
```

第一個建構器擁有一個建構參數，其外部名字為 `fromFahrenheit`，內部名字為 `fahrenheit`；第二個建構器也擁有一個建構參數，其外部名字為 `fromKelvin`，內部名字為 `kelvin`。這兩個建構器都將唯一的參數值轉換成攝氏溫度值，並保存在屬性 `temperatureInCelsius` 中。

內部和外部參數名

跟函式和方法參數相同，建構參數也存在一個在建構器內部使用的參數名字和一個在呼叫建構器時使用的外部參數名字。

然而，建構器並不像函式和方法那樣在括號前有一個可辨別的名字。所以在呼叫建構器時，主要通過建構器中的參數名和型別來確定需要呼叫的建構器。正因為參數如此重要，如果你在定義建構器時沒有提供參數的外部名字，Swift 會為每個建構器的參數自動生成一個跟內部名字相同的外部名，就相當於在每個建構參數之前加了一個雜湊符號。

> 注意：
> 如果你不希望為建構器的某個參數提供外部名字，你可以使用底線 `_` 來顯示描述它的外部名，以此覆蓋上面所說的預設行為。

以下範例中定義了一個結構 `Color`，它包含了三個常數：`red`、`green` 和 `blue`。這些屬性可以儲存0.0到1.0之間的值，用來指示顏色中紅、綠、藍成分的含量。

`Color` 提供了一個建構器，其中包含三個 `Double` 型別的建構參數：

```
struct Color {
    let red = 0.0, green = 0.0, blue = 0.0
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
}
```

每當你創建一個新的 `Color` 實例，你都需要通過三種顏色的外部參數名來傳值，並呼叫建構器。

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
```

注意，如果不通過外部參數名字傳值，你是沒法呼叫這個建構器的。只要建構器定義了某個外部參數名，你就必須使用它，忽略它將導致編譯錯誤：

```
let veryGreen = Color(0.0, 1.0, 0.0)
// 報編譯時錯誤，需要外部名稱
```

可選屬性型別

如果你定制的型別包含一個邏輯上允許取值為空的儲存型屬性--不管是因為它無法在初始化時賦值，還是因為它可以在之後某個時間點可以賦值為空--你都需要將它定義為可選型別 `optional type`。可選型別的屬性將自動初始化為空 `nil`，表示這個屬性是故意在初始化時設置為空的。

下面範例中定義了類別 `SurveyQuestion`，它包含一個可選字串屬性 `response`：

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        println(text)
    }
}
let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")
cheeseQuestion.ask()
// 輸出 "Do you like cheese?"
cheeseQuestion.response = "Yes, I do like cheese.
```

調查問題在問題提出之後，我們才能得到回答。所以我們將屬性回答 `response` 宣告為 `String?` 型別，或者說是可選字串型

別 optional String 。當 SurveyQuestion 實例化時，它將自動賦值為空 nil ，表明暫時還不存在此字串。

建構過程中常數屬性的修改

只要在建構過程結束前常數的值能確定，你可以在建構過程中的任意時間點修改常數屬性的值。

> 注意：
> 對某個類別實例來說，它的常數屬性只能在定義它的類別的建構過程中修改；不能在子類別中修改。

你可以修改上面的 SurveyQuestion 示例，用常數屬性替代變數屬性 text ，指明問題內容 text 在其創建之後不會再被修改。儘管 text 屬性現在是常數，我們仍然可以在其類別的建構器中修改它的值：

```
class SurveyQuestion {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        println(text)
    }
}
let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
// 輸出 "How about beets?"
beetsQuestion.response = "I also like beets. (But not with cheese.)
```

預設建構器

Swift 將為所有屬性已提供預設值的且自身沒有定義任何建構器的結構或基類別，提供一個預設的建構器。這個預設建構器將簡單的創建一個所有屬性值都設置為預設值的實例。

下面範例中創建了一個類別 ShoppingListItem ，它封裝了購物清單中的某一項的屬性：名字（ name ）、數量（ quantity ）和購買狀態 purchase state 。

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()
```

由於 ShoppingListItem 類別中的所有屬性都有預設值，且它是沒有父類別的基類別，它將自動獲得一個可以為所有屬性設置預設值的預設建構器（儘管程式碼中沒有顯式為 name 屬性設置預設值，但由於 name 是可選字串型別，它將預設設置為 nil ）。上面範例中使用預設建構器創造了一個 ShoppingListItem 類別的實例（使用 ShoppingListItem() 形式的建構器語法），並將其賦值給變數 item 。

結構的逐一成員建構器

除上面提到的預設建構器，如果結構對所有儲存型屬性提供了預設值且自身沒有提供定制的建構器，它們能自動獲得一個逐一成員建構器。

逐一成員建構器是用來初始化結構新實例裡成員屬性的快捷方法。我們在呼叫逐一成員建構器時，通過與成員屬性名相同的參數名進行傳值來完成對成員屬性的初始賦值。

下面範例中定義了一個結構 Size ，它包含兩個屬性 width 和 height 。Swift 可以根據這兩個屬性的初始賦值 0.0 自動推導出

它們的型別 `Double`。

由於這兩個儲存型屬性都有預設值，結構 `Size` 自動獲得了一個逐一成員建構器 `init(width:height:)`。你可以用它來為 `Size` 創建新的實例：

```
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

值型別的建構器代理

建構器可以通過呼叫其它建構器來完成實例的部分建構過程。這一過程稱為建構器代理，它能減少多個建構器間的程式碼重複。

建構器代理的實作規則和形式在值型別和類型別中有所不同。值型別（結構和列舉型別）不支援繼承，所以建構器代理的過程相對簡單，因為它們只能代理任務給本身提供的其它建構器。類別則不同，它可以繼承自其它類別（請參考繼承），這意味著類別有責任保證其所有繼承的儲存型屬性在建構時也能正確的初始化。這些責任將在後續章節類別的繼承和建構過程中介紹。

對於值型別，你可以使用 `self.init` 在自定義的建構器中參考其它的屬於相同值型別的建構器。並且你只能在建構器內部呼叫 `self.init`。

注意，如果你為某個值型別定義了一個定制的建構器，你將無法存取到預設建構器（如果是結構，則無法存取逐一物件建構器）。這個限制可以防止你在為值型別定義了一個更複雜的，完成了重要準備建構器之後，別人還是錯誤的使用了那個自動生成的建構器。

> 注意：
> 假如你想通過預設建構器、逐一物件建構器以及你自己定制的建構器為值型別創建實例，我們建議你將自己定制的建構器寫到擴展（`extension`）中，而不是跟值型別定義混在一起。想查看更多內容，請查看擴展章節。

下面範例將定義一個結構 `Rect`，用來展現幾何矩形。這個範例需要兩個輔助的結構 `Size` 和 `Point`，它們各自為其所有的屬性提供了初始值 `0.0`。

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
```

你可以通過以下三種方式為 `Rect` 創建實例--使用預設的0值來初始化 `origin` 和 `size` 屬性；使用特定的 `origin` 和 `size` 實例來初始化；使用特定的 `center` 和 `size` 來初始化。在下面 `Rect` 結構定義中，我們為著三種方式提供了三個自定義的建構器：

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
```

```
}
```

第一個 `Rect` 建構器 `init()`，在功能上跟沒有自定義建構器時自動獲得的預設建構器是一樣的。這個建構器是一個空函式，使用一對大括號 `{}` 來描述，它沒有執行任何定制的建構過程。呼叫這個建構器將回傳一個 `Rect` 實例，它的 `origin` 和 `size` 屬性都使用定義時的預設值 `Point(x: 0.0, y: 0.0)` 和 `Size(width: 0.0, height: 0.0)`：

```
let basicRect = Rect()
// basicRect 的原點是 (0.0, 0.0)，尺寸是 (0.0, 0.0)
```

第二個 `Rect` 建構器 `init(origin:size:)`，在功能上跟結構在沒有自定義建構器時獲得的逐一成員建構器是一樣的。這個建構器只是簡單的將 `origin` 和 `size` 的參數值賦給對應的儲存型屬性：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
// originRect 的原點是 (2.0, 2.0)，尺寸是 (5.0, 5.0)
```

第三個 `Rect` 建構器 `init(center:size:)` 稍微複雜一點。它先通過 `center` 和 `size` 的值計算出 `origin` 的坐標。然後再呼叫（或代理給） `init(origin:size:)` 建構器來將新的 `origin` 和 `size` 值賦值到對應的屬性中：

let centerRect = Rect(center: Point(x: 4.0, y: 4.0),

```
size: Size(width: 3.0, height: 3.0))
```

// centerRect 的原點是 (2.5, 2.5)，尺寸是 (3.0, 3.0)

建構器 `init(center:size:)` 可以自己將 `origin` 和 `size` 的新值賦值到對應的屬性中。然而盡量利用現有的建構器和它所提供的功能來實作 `init(center:size:)` 的功能，是更方便、更清晰和更直觀的方法。

> 注意：
> 如果你想用另外一種不需要自己定義 `init()` 和 `init(origin:size:)` 的方式來實作這個範例，請參考擴展。

類別的繼承和建構過程

類別裡面的所有儲存型屬性--包括所有繼承自父類別的屬性--都必須在建構過程中設置初始值。

Swift 提供了兩種型別的類別建構器來確保所有類別實例中儲存型屬性都能獲得初始值，它們分別是指定建構器和便利建構器。

指定建構器和便利建構器

指定建構器是類別中最主要的建構器。一個指定建構器將初始化類別中提供的所有屬性，並根據父類別鏈往上呼叫父類別的建構器來實作父類別的初始化。

每一個類別都必須擁有至少一個指定建構器。在某些情況下，許多類別通過繼承了父類別中的指定建構器而滿足了這個條件。具體內容請參考後續章節自動建構器的繼承。

便利建構器是類別中比較次要的、輔助型的建構器。你可以定義便利建構器來呼叫同一個類別中的指定建構器，並為其參數提供預設值。你也可以定義便利建構器來創建一個特殊用途或特定輸入的實例。

你應當只在必要的時候為類別提供便利建構器，比方說某種情況下通過使用便利建構器來快捷呼叫某個指定建構器，能夠節省更多開發時間並讓類別的建構過程更清、晰明。

建構器鏈

為了簡化指定建構器和便利建構器之間的呼叫關系，Swift 采用以下三條規則來限制建構器之間的代理呼叫：

規則 1

指定建構器必須呼叫其直接父類別的的指定建構器。

規則 2

便利建構器必須呼叫同一類別中定義的其它建構器。

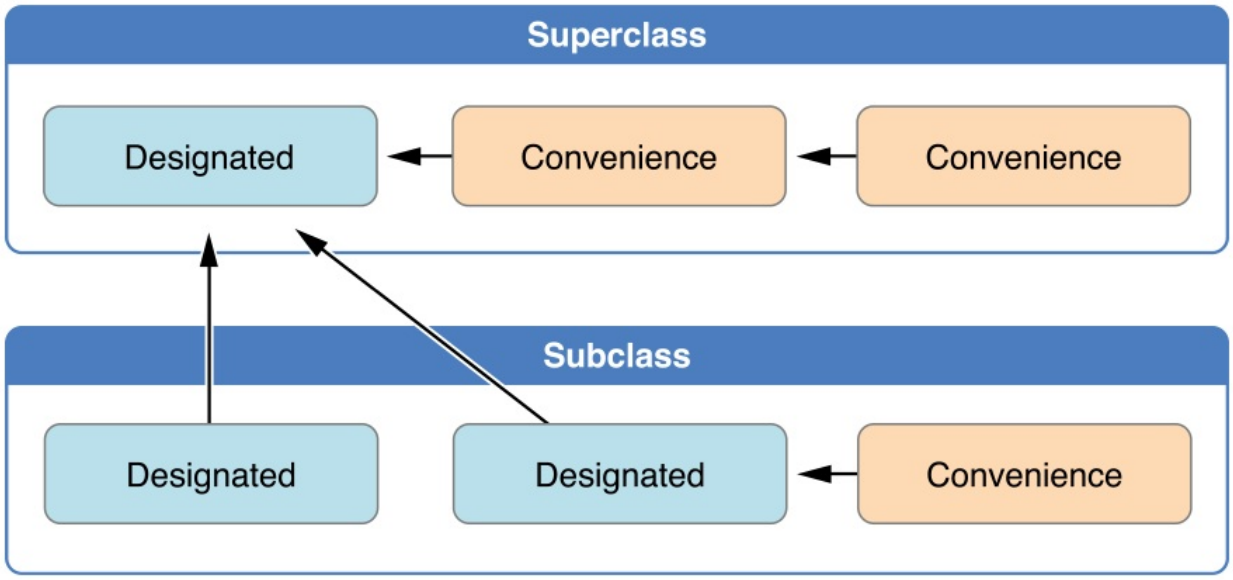
規則 3

便利建構器必須最終以呼叫一個指定建構器結束。

一個更方便記憶的方法是：

- 指定建構器必須總是向上代理
- 便利建構器必須總是橫向代理

這些規則可以通過下面圖例來說明：

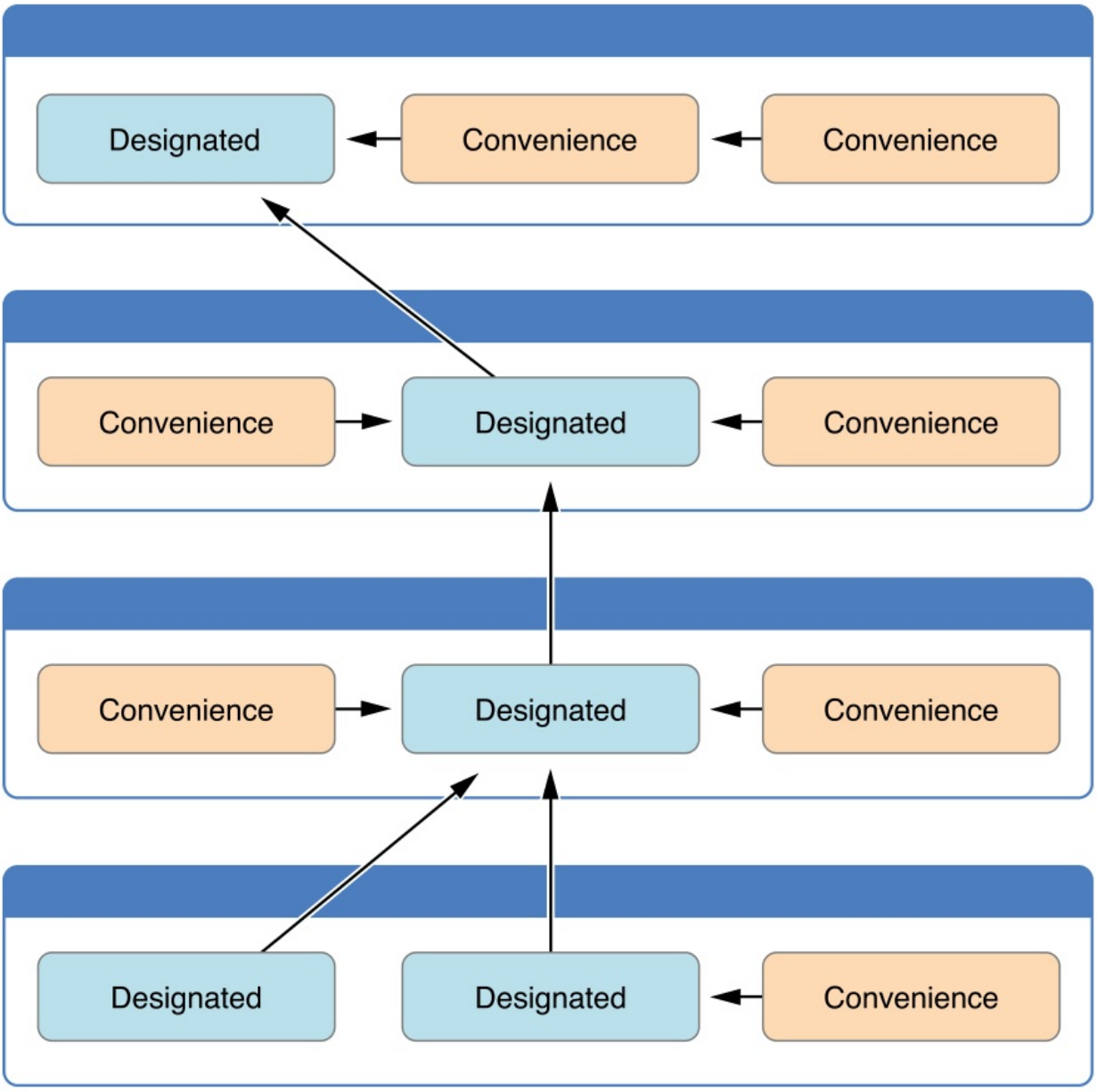


如圖所示，父類別中包含一個指定建構器和兩個便利建構器。其中一個便利建構器呼叫了另外一個便利建構器，而後者又呼叫了唯一的指定建構器。這滿足了上面提到的規則2和3。這個父類別沒有自己的父類別，所以規則1沒有用到。

子類別中包含兩個指定建構器和一個便利建構器。便利建構器必須呼叫兩個指定建構器中的任意一個，因為它只能呼叫同一個類別裡的其他建構器。這滿足了上面提到的規則2和3。而兩個指定建構器必須呼叫父類別中唯一的指定建構器，這滿足了規則1。

> 注意：
> 這些規則不會影響使用時，如何用類別去創建實例。任何上圖中展示的建構器都可以用來完整創建對應類別的實例。
> 這些規則只在實作類別的定義時有影響。

下面圖例中展示了一種更複雜的類別層級結構。它演示了指定建構器是如果在類別層級中充當「管道」的作用，在類別的建構器鏈上簡化了類別之間的內部關系。



兩段式建構過程

Swift 中類別的建構過程包含兩個階段。第一個階段，每個儲存型屬性通過引入它們的類別的建構器來設置初始值。當每一個儲存型屬性值被確定後，第二階段開始，它給每個類別一次機會在新實例準備使用之前進一步定制它們的儲存型屬性。

兩段式建構過程的使用讓建構過程更安全，同時在整個類別層級結構中給予了每個類別完全的靈活性。兩段式建構過程可以防止屬性值在初始化之前被存取；也可以防止屬性被另外一個建構器意外地賦予不同的值。

> 注意：
> Swift的兩段式建構過程跟 Objective-C 中的建構過程類似。最主要的區別在於階段 1，Objective-C 給每一個屬性賦值 `0` 或空值（比如說 `0` 或 `nil`）。Swift 的建構流程則更加靈活，它允許你設置定制的初始值，並自如應對某些屬性不能以 `0` 或 `nil` 作為合法預設值的情況。

Swift 編譯器將執行 4 種有效的安全檢查，以確保兩段式建構過程能順利完成：

安全檢查 1

指定建構器必須保證它所在類別引入的所有屬性都必須先初始化完成，之後才能將其它建構任務向上代理給父類別中的建構

器。

如上所述，一個物件的內存只有在其所有儲存型屬性確定之後才能完全初始化。為了滿足這一規則，指定建構器必須保證它所在類別引入的屬性在它往上代理之前先完成初始化。

安全檢查 2

指定建構器必須先向上代理呼叫父類別建構器，然後再為繼承的屬性設置新值。如果沒這麼做，指定建構器賦予的新值將被父類別中的建構器所覆蓋。

安全檢查 3

便利建構器必須先代理呼叫同一類別中的其它建構器，然後再為任意屬性賦新值。如果沒這麼做，便利建構器賦予的新值將被同一類別中其它指定建構器所覆蓋。

安全檢查 4

建構器在第一階段建構完成之前，不能呼叫任何實例方法、不能讀取任何實例屬性的值，也不能參考 `self` 的值。

以下是兩段式建構過程中基於上述安全檢查的建構流程展示：

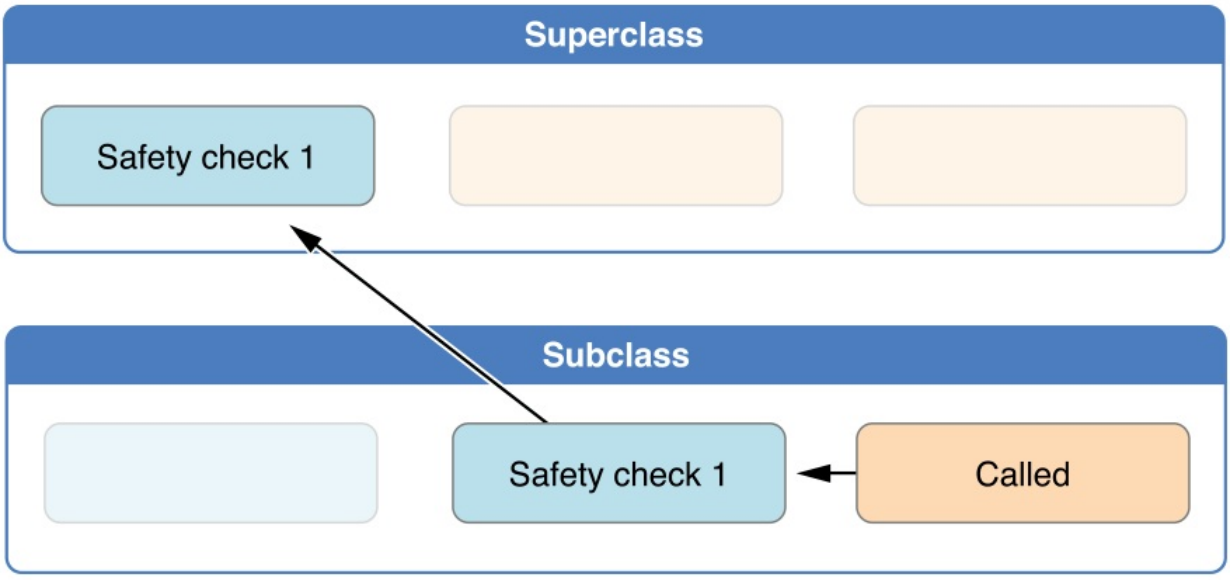
階段 1

- 某個指定建構器或便利建構器被呼叫；
- 完成新實例內存的分配，但此時內存還沒有被初始化；
- 指定建構器確保其所在類別引入的所有儲存型屬性都已賦初值。儲存型屬性所屬的內存完成初始化；
- 指定建構器將呼叫父類別的建構器，完成父類別屬性的初始化；
- 這個呼叫父類別建構器的過程沿著建構器鏈一直往上執行，直到到達建構器鏈的最頂部；
- 當到達了建構器鏈最頂部，且已確保所有實例包含的儲存型屬性都已經賦值，這個實例的內存被認為已經完全初始化。此時階段1完成。

階段 2

- 從頂部建構器鏈一直往下，每個建構器鏈中類別的指定建構器都有機會進一步定制實例。建構器此時可以存取 `self`、修改它的屬性並呼叫實例方法等等。
- 最終，任意建構器鏈中的便利建構器可以有機會定制實例和使用 `self`。

下圖展示了在假定的子類別和父類別之間建構的階段1： ·



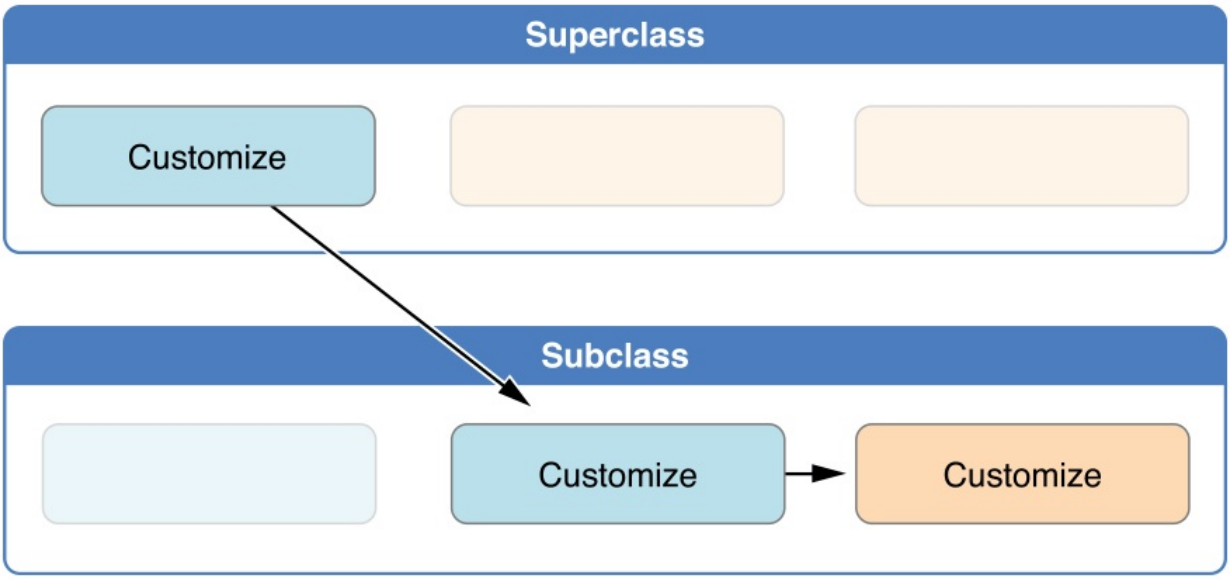
在這個範例中，建構過程從對子類別中一個便利建構器的呼叫開始。這個便利建構器此時沒法修改任何屬性，它把建構任務代理給同一類別中的指定建構器。

如安全檢查1所示，指定建構器將確保所有子類別的屬性都有值。然後它將呼叫父類別的指定建構器，並沿著造器鏈一直往上完成父類別的構建過程。

父類別中的指定建構器確保所有父類別的屬性都有值。由於沒有更多的父類別需要構建，也就無需繼續向上做構建代理。

一旦父類別中所有屬性都有了初始值，實例的內存被認為是完全初始化，而階段1也已完成。

以下展示了相同建構過程的階段2：



父類別中的指定建構器現在有機會進一步來定制實例（儘管它沒有這種必要）。

一旦父類別中的指定建構器完成呼叫，子類別的構指定建構器可以執行更多的定制操作（同樣，它也沒有這種必要）。

最終，一旦子類別的指定建構器完成呼叫，最開始被呼叫的便利建構器可以執行更多的定制操作。

建構器的繼承和重載

跟 Objective-C 中的子類別不同，Swift 中的子類別不會預設繼承父類別的建構器。Swift 的這種機制可以防止一個父類別的簡單建構器被一個更專業的子類別繼承，並被錯誤的用來創建子類別的實例。

假如你希望自定義的子類別中能實作一個或多個跟父類別相同的建構器--也許是為了完成一些定制的建構過程--你可以在你定制的子類別中提供和重載與父類別相同的建構器。

如果你重載的建構器是一個指定建構器，你可以在子類別裡重載它的實作，並在自定義版本的建構器中呼叫父類別版本的建構器。

如果你重載的建構器是一個便利建構器，你的重載過程必須通過呼叫同一類別中提供的其它指定建構器來實作。這一規則的詳細內容請參考建構器鏈。

> 注意：
> 與方法、屬性和下標不同，在重載建構器時你沒有必要使用關鍵字 `override`。

自動建構器的繼承

如上所述，子類別不會預設繼承父類別的建構器。但是如果特定條件可以滿足，父類別建構器是可以被自動繼承的。在實踐中，這意味著對於許多常見場景你不必重載父類別的建構器，並且在盡可能安全的情況下以最小的代價來繼承父類別的建構器。

假設要為子類別中引入的任意新屬性提供預設值，請遵守以下2個規則：

規則 1

如果子類別沒有定義任何指定建構器，它將自動繼承所有父類別的指定建構器。

規則 2

如果子類別提供了所有父類別指定建構器的實作--不管是通過規則1繼承過來的，還是通過自定義實作的--它將自動繼承所有父類別的便利建構器。

即使你在子類別中添加了更多的便利建構器，這兩條規則仍然適用。

> 注意：
> 子類別可以通過部分滿足規則2的方式，使用子類別便利建構器來實作父類別的指定建構器。

指定建構器和便利建構器的語法

類別的指定建構器的寫法跟值型別簡單建構器一樣：

```
init(parameters) {
    statements
}
```

便利建構器也采用相同樣式的寫法，但需要在 `init` 關鍵字之前放置 `convenience` 關鍵字，並使用空格將它們倆分開：

```
convenience init(parameters) {
    statements
}
```

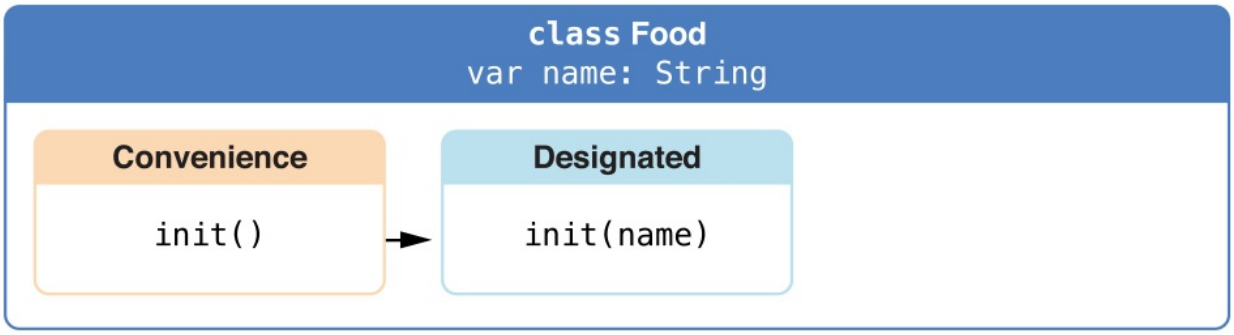
指定建構器和便利建構器實戰

接下來的範例將在實戰中展示指定建構器、便利建構器和自動建構器的繼承。它定義了包含三個類別 `Food`、`RecipeIngredient` 以及 `ShoppingListItem` 的類別層次結構，並將演示它們的建構器是如何相互作用的。

類別層次中的基類別是 `Food`，它是一個簡單的用來封裝食物名字的類別。`Food` 類別引入了一個叫做 `name` 的 `String` 型別屬性，並且提供了兩個建構器來創建 `Food` 實例：

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

下圖中展示了 `Food` 的建構器鏈：



類別沒有提供一個預設的逐一成員建構器，所以 `Food` 類別提供了一個接受單一參數 `name` 的指定建構器。這個建構器可以使用一個特定的名字來創建新的 `Food` 實例：

```
let namedMeat = Food(name: "Bacon")
// namedMeat 的名字是 "Bacon」
```

`Food` 類別中的建構器 `init(name: String)` 被定義為一個指定建構器，因為它能確保所有新 `Food` 實例的中儲存型屬性都被初始化。`Food` 類別沒有父類別，所以 `init(name: String)` 建構器不需要呼叫 `super.init()` 來完成建構。

`Food` 類別同樣提供了一個沒有參數的便利建構器 `init()`。這個 `init()` 建構器為新食物提供了一個預設的占位名字，通過代理呼叫同一類別中定義的指定建構器 `init(name: String)` 並給參數 `name` 傳值 `[Unnamed]` 來實作：

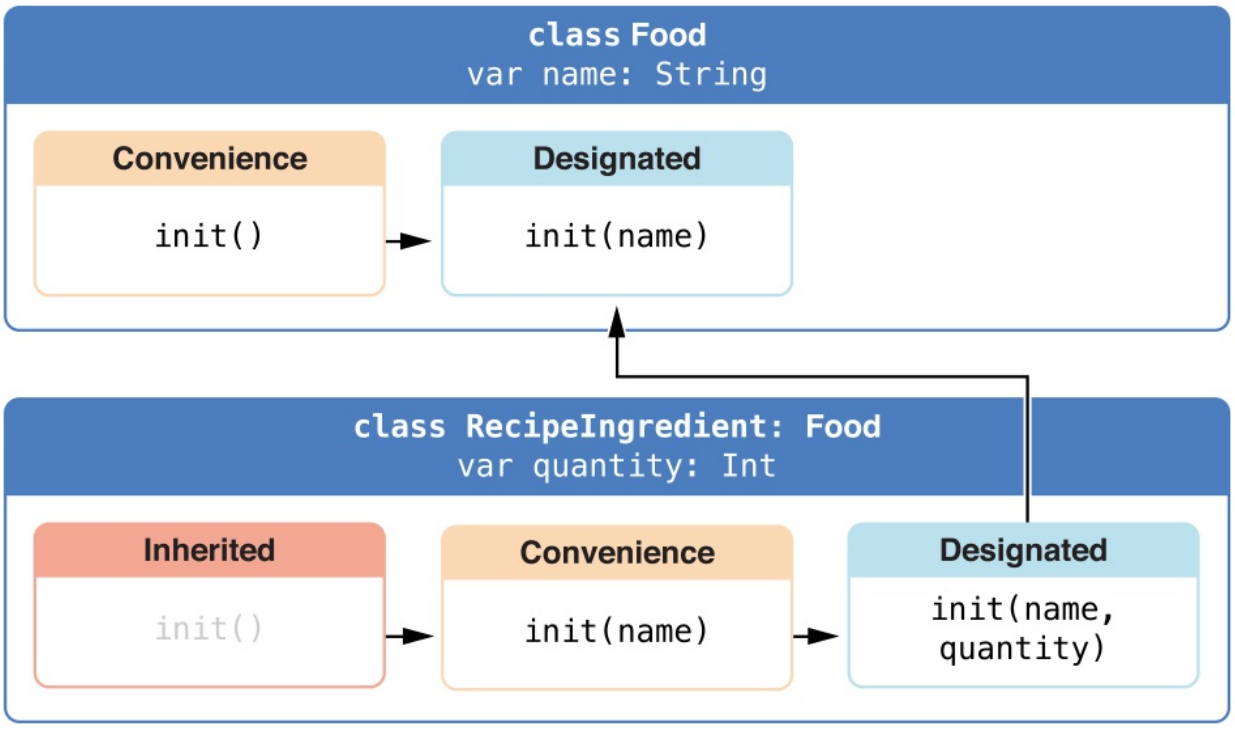
```
let mysteryMeat = Food()
// mysteryMeat 的名字是 [Unnamed]
```

類別層級中的第二個類別是 `Food` 的子類別 `RecipeIngredient`。`RecipeIngredient` 類別構建了食譜中的一味調味劑。它引入了 `Int` 型別的數量屬性 `quantity`（以及從 `Food` 繼承過來的 `name` 屬性），並且定義了兩個建構器來創建 `RecipeIngredient` 實例：

```
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    convenience init(name: String) {
        self.init(name: name, quantity: 1)
```

```
    }
}
```

下圖中展示了 `RecipeIngredient` 類別的建構器鏈：



`RecipeIngredient` 類別擁有一個指定建構器 `init(name: String, quantity: Int)`，它可以用來產生新 `RecipeIngredient` 實例的所有屬性值。這個建構器一開始先將傳入的 `quantity` 參數賦值給 `quantity` 屬性，這個屬性也是唯一在 `RecipeIngredient` 中新引入的屬性。隨後，建構器將任務向上代理給父類別 `Food` 的 `init(name: String)`。這個過程滿足兩段式建構過程中的安全檢查1。

`RecipeIngredient` 也定義了一個便利建構器 `init(name: String)`，它只通過 `name` 來創建 `RecipeIngredient` 的實例。這個便利建構器假設任意 `RecipeIngredient` 實例的 `quantity` 為1，所以不需要顯示指明數量即可創建出實例。這個便利建構器的定義可以讓創建實例更加方便和快捷，並且避免了使用重複的程式碼來創建多個 `quantity` 為 1 的 `RecipeIngredient` 實例。這個便利建構器只是簡單的將任務代理給了同一類別裡提供的指定建構器。

注意，`RecipeIngredient` 的便利建構器 `init(name: String)` 使用了跟 `Food` 中指定建構器 `init(name: String)` 相同的參數。儘管 `RecipeIngredient` 這個建構器是便利建構器，`RecipeIngredient` 依然提供了對所有父類別指定建構器的實作。因此，`RecipeIngredient` 也能自動繼承了所有父類別的便利建構器。

在這個範例中，`RecipeIngredient` 的父類別是 `Food`，它有一個便利建構器 `init()`。這個建構器因此也被 `RecipeIngredient` 繼承。這個繼承的 `init()` 函式版本跟 `Food` 提供的版本是一樣的，除了它是將任務代理給 `RecipeIngredient` 版本的 `init(name: String)` 而不是 `Food` 提供的版本。

所有的這三種建構器都可以用來創建新的 `RecipeIngredient` 實例：

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

類別層級中第三個也是最後一個類別是 `RecipeIngredient` 的子類別，叫做 `ShoppingListItem`。這個類別構建了購物單中出現的某一種調味料。

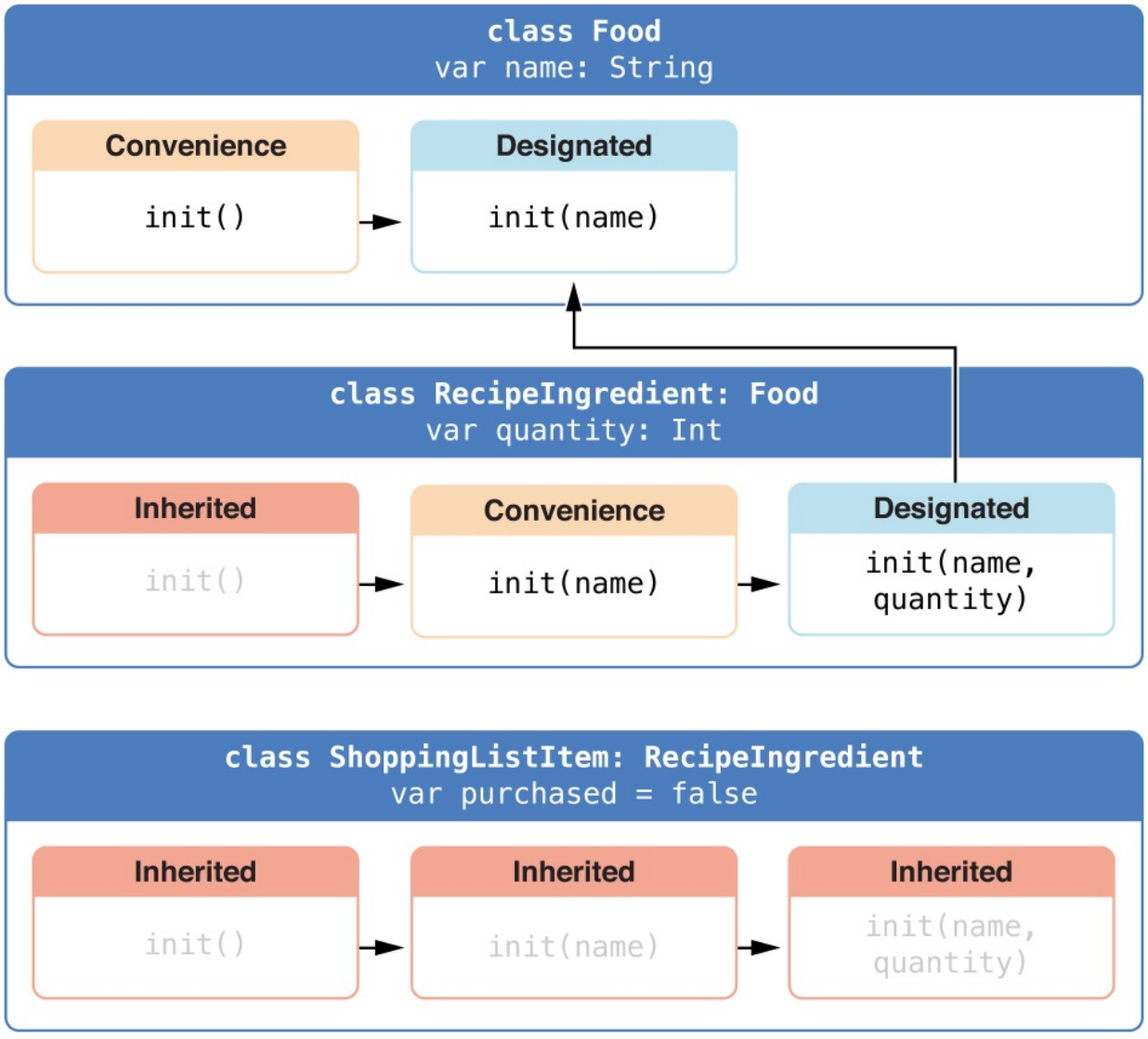
購物單中的每一項總是從 `unpurchased` 未購買狀態開始的。為了展現這一事實， `ShoppingListItem` 引入了一個布林型別的屬性 `purchased` ，它的預設值是 `false` 。 `ShoppingListItem` 還添加了一個計算型屬性 `description` ，它提供了關於 `ShoppingListItem` 實例的一些文字描述：

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
    var output = "\(quantity) x \(name.lowercaseString)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

> 注意：
> `ShoppingListItem` 沒有定義建構器來為 `purchased` 提供初始化值，這是因為任何添加到購物單的項的初始狀態總是未購買。

由於它為自己引入的所有屬性都提供了預設值，並且自己沒有定義任何建構器， `ShoppingListItem` 將自動繼承所有父類別中的指定建構器和便利建構器。

下圖種展示了所有三個類別的建構器鏈：



你可以使用全部三個繼承來的建構器來創建 `ShoppingListItem` 的新實例：

```
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    println(item.description)
}
// 1 x orange juice ✔
// 1 x bacon ✘
// 6 x eggs ✘
```

如上所述，範例中通過字面量方式創建了一個新陣列 `breakfastList`，它包含了三個新的 `ShoppingListItem` 實例，因此陣列的型別也能自動推導為 `ShoppingListItem[]`。在陣列創建完之後，陣列中第一個 `ShoppingListItem` 實例的名字從 `[Unnamed]` 修改為 `Orange juice`，並標記為已購買。接下來通過遍歷陣列每個元素並列印它們的描述值，展示了所有項當前的預設狀態都已按照預期完成了賦值。

通過閉包和函式來設置屬性的預設值

如果某個儲存型屬性的預設值需要特別的定制或準備，你就可以使用閉包或全域函式來為其屬性提供定制的預設值。每當某個屬性所屬的新型別實例創建時，對應的閉包或函式會被呼叫，而它們的回傳值會當做預設值賦值給這個屬性。

這種型別的閉包或函式一般會創建一個跟屬性型別相同的臨時變數，然後修改它的值以滿足預期的初始狀態，最後將這個臨時變數的值作為屬性的預設值進行回傳。

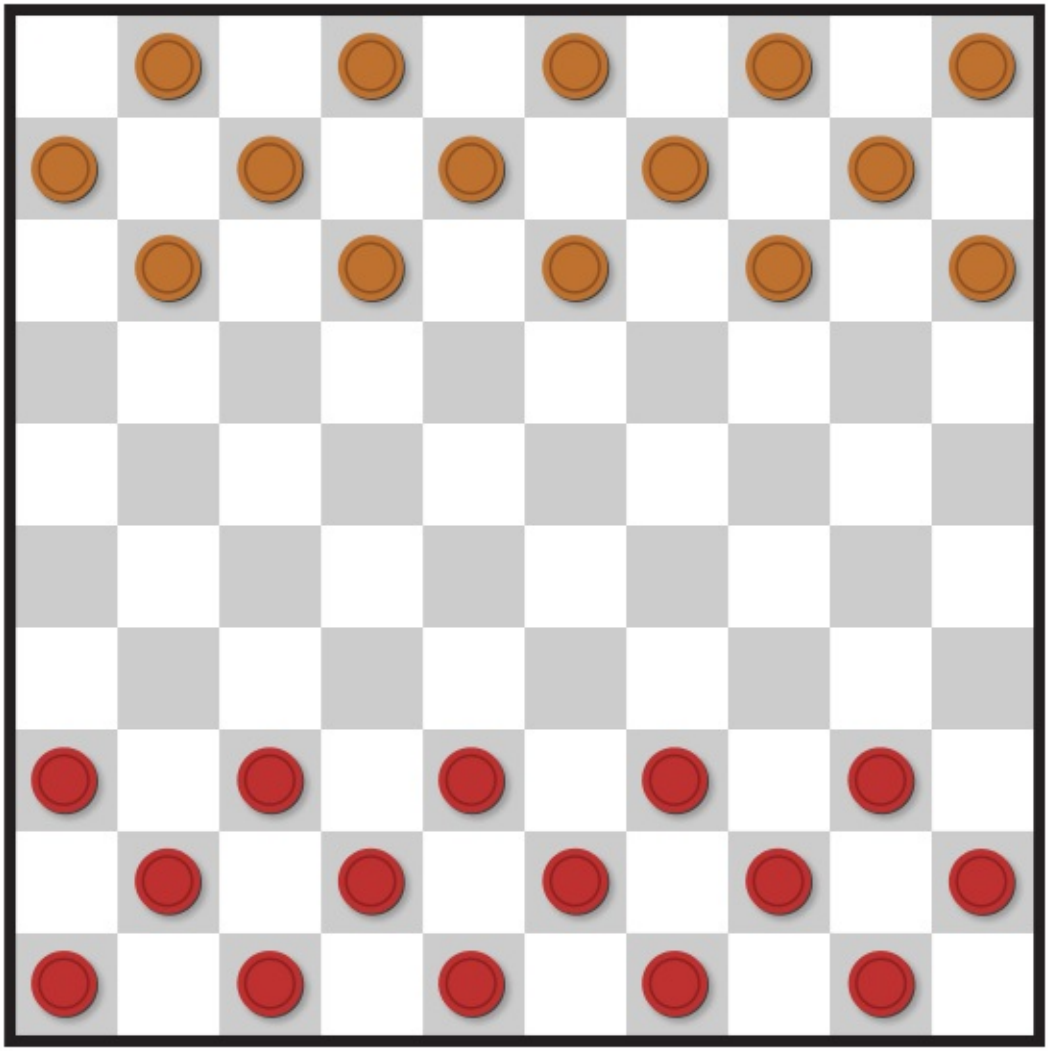
下面列舉了閉包如何提供預設值的程式碼概要：

```
class SomeClass {
    let someProperty: SomeType = {
        // 在這個閉包中給 someProperty 創建一個預設值
        // someValue 必須和 SomeType 型別相同
        return someValue
        }()
}
```

注意閉包結尾的大括號後面接了一對空的小括號。這是用來告訴 Swift 需要立刻執行此閉包。如果你忽略了這對括號，相當於是將閉包本身作為值賦值給了屬性，而不是將閉包的回傳值賦值給屬性。

> 注意：
> 如果你使用閉包來初始化屬性的值，請記住在閉包執行時，實例的其它部分都還沒有初始化。這意味著你不能夠在閉包裡存取其它的屬性，就算這個屬性有預設值也不允許。同樣，你也不能使用隱式的 `self` 屬性，或者呼叫其它的實例方法。

下面範例中定義了一個結構 `Checkerboard`，它構建了西洋跳棋遊戲的棋盤：



西洋跳棋遊戲在一副黑白格交替的 10x10 的棋盤中進行。為了呈現這副遊戲棋盤，`Checkerboard` 結構定義了一個屬性 `boardColors`，它是一個包含 100 個布林值的陣列。陣列中的某元素布林值為 `true` 表示對應的是一個黑格，布林值為 `false` 表示對應的是一個白格。陣列中第一個元素代表棋盤上左上角的格子，最後一個元素代表棋盤上右下角的格子。

`boardColor` 陣列是通過一個閉包來初始化和組裝顏色值的：

```
struct Checkerboard {
    let boardColors: Bool[] = {
        var temporaryBoard = Bool[]()
        var isBlack = false
        for i in 1...10 {
            for j in 1...10 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
        }()
    func squareIsBlackAtRow(row: Int, column: Int) -> Bool {
        return boardColors[(row * 10) + column]
    }
}
```

每當一個新的 `Checkerboard` 實例創建時，對應的賦值閉包會執行，一系列顏色值會被計算出來作為預設值賦值給 `boardColors`。上面範例中描述的閉包將計算出棋盤中每個格子合適的顏色，將這些顏色值保存到一個臨時陣

列 `temporaryBoard` 中，並在構建完成時將此陣列作為閉包回傳值回傳。這個回傳的值將保存到 `boardColors` 中，並可以通 `squareIsBlackAtRow` 這個工具函式來查詢。

```
let board = Checkerboard()
println(board.squareIsBlackAtRow(0, column: 1))
// 輸出 "true"
println(board.squareIsBlackAtRow(9, column: 9))
// 輸出 "false"
```

> 翻譯：bruce0505 校對：fd5788

析構過程（Deinitialization）

本頁包含內容：

- 析構過程原理
- 析構函式操作

在一個類別的實例被釋放之前，析構函式被立即呼叫。用關鍵字 `deinit` 來標示析構函式，類似於初始化函式用 `init` 來標示。析構函式只適用於類型別。

析構過程原理

Swift 會自動釋放不再需要的實例以釋放資源。如自動引用計數那一章描述，Swift 通過自動引用計數（ARC）處理實例的內存管理。通常當你的實例被釋放時不需要手動地去清理。但是，當使用自己的資源時，你可能需要進行一些額外的清理。例如，如果創建了一個自定義的類別來打開一個文件，並寫入一些資料，你可能需要在類別實例被釋放之前關閉該文件。

在類別的定義中，每個類別最多只能有一個析構函式。析構函式不帶任何參數，在寫法上不帶括號：

```
deinit {
    // 執行析構過程
}
```

析構函式是在實例釋放發生前一步被自動呼叫。不允許主動呼叫自己的析構函式。子類別繼承了父類別的析構函式，並且在子類別析構函式實作的最後，父類別的析構函式被自動呼叫。即使子類別沒有提供自己的析構函式，父類別的析構函式也總是被呼叫。

因為直到實例的析構函式被呼叫時，實例才會被釋放，所以析構函式可以存取所有請求實例的屬性，並且根據那些屬性可以修改它的行為（比如查找一個需要被關閉的文件的名稱）。

析構函式操作

這裡是一個析構函式操作的範例。這個範例是一個簡單的遊戲，定義了兩種新型別，`Bank` 和 `Player`。`Bank` 結構管理一個虛擬貨幣的流通，在這個流通中 `Bank` 永遠不可能擁有超過 10,000 的硬幣。在這個遊戲中有且只能有一個 `Bank` 存在，因此 `Bank` 由帶有靜態屬性和靜態方法的結構實作，從而儲存和管理其當前的狀態。

```
struct Bank {
    static var coinsInBank = 10_000
    static func vendCoins(var numberOfCoinsToVend: Int) -> Int {
          numberOfCoinsToVend = min(numberOfCoinsToVend, coinsInBank)
          coinsInBank -= numberOfCoinsToVend
         return numberOfCoinsToVend
    }
    static func receiveCoins(coins: Int) {
        coinsInBank += coins
    }
}
```

`Bank` 根據它的 `coinsInBank` 屬性來跟蹤當前它擁有的硬幣數量。銀行還提供兩個方法——`vendCoins` 和 `receiveCoins`——用來處理硬幣的分發和收集。

`vendCoins` 方法在 bank 分發硬幣之前檢查是否有足夠的硬幣。如果沒有足夠多的硬幣， `Bank` 回傳一個比請求時小的數字(如果沒有硬幣留在 bank 中就回傳 0)。 `vendCoins` 方法宣告 `numberOfCoinsToVend` 為一個變數參數，這樣就可以在方法體的內部修改數字，而不需要定義一個新的變數。 `vendCoins` 方法回傳一個整型值，表明了提供的硬幣的實際數目。

`receiveCoins` 方法只是將 bank 的硬幣儲存和接收到的硬幣數目相加，再保存回 bank。

`Player` 類別描述了遊戲中的一個玩家。每一個 player 在任何時刻都有一定數量的硬幣儲存在他們的錢包中。這通過 player 的 `coinsInPurse` 屬性來體現：

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.vendCoins(coins)
    }
    func winCoins(coins: Int) {
        coinsInPurse += Bank.vendCoins(coins)
    }
    deinit {
        Bank.receiveCoins(coinsInPurse)
    }
}
```

每個 `Player` 實例都由一個指定數目硬幣組成的啟動額度初始化，這些硬幣在 bank 初始化的過程中得到。如果沒有足夠的硬幣可用， `Player` 實例可能收到比指定數目少的硬幣。

`Player` 類別定義了一個 `winCoins` 方法，該方法從銀行獲取一定數量的硬幣，並把它們添加到玩家的錢包。 `Player` 類別還實作了一個析構函式，這個析構函式在 `Player` 實例釋放前一步被呼叫。這裡析構函式只是將玩家的所有硬幣都回傳給銀行：

```
var playerOne: Player? = Player(coins: 100)
println("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
// 輸出 "A new player has joined the game with 100  coins"
println("There are now \(Bank.coinsInBank) coins left   in the bank")
// 輸出 "There are now 9900 coins left in the bank"
```

一個新的 `Player` 實例隨著一個 100 個硬幣（如果有）的請求而被創建。這 `個Player` 實例儲存在一個名為 `playerOne` 的可選 `Player` 變數中。這裡使用一個可選變數，是因為玩家可以隨時離開遊戲。設置為可選使得你可以跟蹤當前是否有玩家在遊戲中。

因為 `playerOne` 是可選的，所以由一個感嘆號（ `!` ）來修飾，每當其 `winCoins` 方法被呼叫時， `coinsInPurse` 屬性被存取並列印出它的預設硬幣數目。

```
playerOne!.winCoins(2_000)
println("PlayerOne won 2000 coins & now has \ (playerOne!.coinsInPurse) coins")
// 輸出 "PlayerOne won 2000 coins & now has 2100 coins"
println("The bank now only has \(Bank.coinsInBank) coins left")
// 輸出 "The bank now only has 7900 coins left"
```

這裡，player 已經贏得了 2,000 硬幣。player 的錢包現在有 2,100 硬幣，bank 只剩余 7,900 硬幣。

```
playerOne = nil
println("PlayerOne has left the game")
// 輸出 "PlayerOne has left the game"
println("The bank now has \(Bank.coinsInBank) coins")
// 輸出 "The bank now has 10000 coins"
```

玩家現在已經離開了遊戲。這表明是要將可選的 `playerOne` 變數設置為 `nil` ，意思是「沒有 `Player` 實例」。當這種情況發生的時候， `playerOne` 變數對 `Player` 實例的參考被破壞了。沒有其它屬性或者變數參考 `Player` 實例，因此為了清空它占用的內

存從而釋放它。在這發生前一步，其析構函式被自動呼叫，其硬幣被回傳到銀行。

> 翻譯：TimothyYe 校對：Hawstein

自動引用計數

本頁包含內容：

Swift 使用自動引用計數（ARC）這一機制來跟蹤和管理你的應用程式的內存。通常情況下，Swift 的內存管理機制會一直起著作用，你無須自己來考慮內存的管理。ARC 會在類別的實例不再被使用時，自動釋放其占用的內存。

然而，在少數情況下，ARC 為了能幫助你管理內存，需要更多的關於你的程式碼之間關系的資訊。本章描述了這些情況，並且為你示範怎樣啟用 ARC 來管理你的應用程式的內存。

> 注意:
> 參考計數僅僅應用於類別的實例。結構和列舉型別是值型別，不是參考型別，也不是通過參考的方式儲存和傳遞。

自動引用計數的工作機制

當你每次創建一個類別的新的實例的時候，ARC 會分配一大塊內存用來儲存實例的資訊。內存中會包含實例的型別資訊，以及這個實例所有相關屬性的值。此外，當實例不再被使用時，ARC 釋放實例所占用的內存，並讓釋放的內存能挪作他用。這確保了不再被使用的實例，不會一直占用內存空間。

然而，當 ARC 收回和釋放了正在被使用中的實例，該實例的屬性和方法將不能再被存取和呼叫。實際上，如果你試圖存取這個實例，你的應用程式很可能會崩潰。

為了確保使用中的實例不會被銷毀，ARC 會跟蹤和計算每一個實例正在被多少屬性，常數和變數所參考。哪怕實例的參考數為一，ARC都不會銷毀這個實例。

為了使之成為可能，無論你將實例賦值給屬性，常數或者是變數，屬性，常數或者變數，都會對此實例創建強參考。之所以稱之為強參考，是因為它會將實例牢牢的保持住，只要強參考還在，實例是不允許被銷毀的。

自動引用計數實踐

下面的範例展示了自動引用計數的工作機制。範例以一個簡單的 `Person` 類別開始，並定義了一個叫 `name` 的常數屬性：

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        println("\(name) is being initialized")
    }
    deinit {
        println("\(name) is being deinitialized")
    }
}
```

`Person`類別有一個建構函式，此建構函式為實例的`name`屬性賦值並列印出資訊，以表明初始化過程生效。`Person`類別同時也擁有析構函式，同樣會在實例被銷毀的時候列印出資訊。

接下來的程式碼片段定義了三個型別為`Person?`的變數，用來按照程式碼片段中的順序，為新的`Person`實例建立多個參考。由於這些變數是被定義為可選型別（Person?，而不是Person），它們的值會被自動初始化為`nil`，目前還不會參考到`Person`類別的實例。

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

現在你可以創建`Person`類別的新實例，並且將它賦值給三個變數其中的一個：

```
reference1 = Person(name: "John Appleseed")
// prints "John Appleseed is being initialized」
```

應當注意到當你呼叫`Person`類別的建構函式的時候，"John Appleseed is being initialized」會被列印出來。由此可以確定建構函式被執行。

由於`Person`類別的新實例被賦值給了`reference1`變數，所以`reference1`到`Person`類別的新實例之間建立了一個強參考。正是因為這個強參考，ARC 會保證`Person`實例被保持在內存中不被銷毀。

如果你將同樣的`Person`實例也賦值給其他兩個變數，該實例又會多出兩個強參考：

```
reference2 = reference1
reference3 = reference1
```

現在這個`Person`實例已經有三個強參考了。

如果你通過給兩個變數賦值`nil`的方式斷開兩個強參考（）包括最先的那個強參考），只留下一個強參考，`Person`實例不會被銷毀：

```
reference2 = nil
reference3 = nil
```

ARC 會在第三個，也即最後一個強參考被斷開的時候，銷毀`Person`實例，這也意味著你不再使用這個`Person`實例：

```
reference3 = nil
// prints "John Appleseed is being deinitialized"
```

類別實例之間的迴圈強參考

在上面的範例中，ARC 會跟蹤你所新創建的`Person`實例的參考數量，並且會在`Person`實例不再被需要時銷毀它。

然而，我們可能會寫出這樣的程式碼，一個類別永遠不會有0個強參考。這種情況發生在兩個類別實例互相保持對方的強參考，並讓對方不被銷毀。這就是所謂的迴圈強參考。

你可以通過定義類別之間的關系為弱參考或者無主參考，以此替代強參考，從而解決迴圈強參考的問題。具體的過程在解決類別實例之間的迴圈強參考中有描述。不管怎樣，在你學習怎樣解決迴圈強參考之前，很有必要了解一下它是怎樣產生的。

下面展示了一個不經意產生迴圈強參考的範例。範例定義了兩個類別：`Person` 和 `Apartment`，用來建模公寓和它其中的居民:

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { println("\(name) is being deinitialized") }
}
```

```
class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    var tenant: Person?
    deinit { println("Apartment #\(number) is being deinitialized") }
}
```

每一個 `Person` 實例有一個型別為 `String`，名字為 `name` 的屬性，並有一個可選的初始化為 `nil` 的 `apartment` 屬性。`apartment` 屬性是可選的，因為一個人並不總是擁有公寓。

類似的，每個 `Apartment` 實例有一個叫 `number`，型別為 `Int` 的屬性，並有一個可選的初始化為 `nil` 的 `tenant` 屬性。`tenant` 屬性是可選的，因為一棟公寓並不總是有居民。

這兩個類別都定義了析構函式，用以在類別實例被析構的時候輸出資訊。這讓你能夠知曉 `Person` 和 `Apartment` 的實例是否像預期的那樣被銷毀。

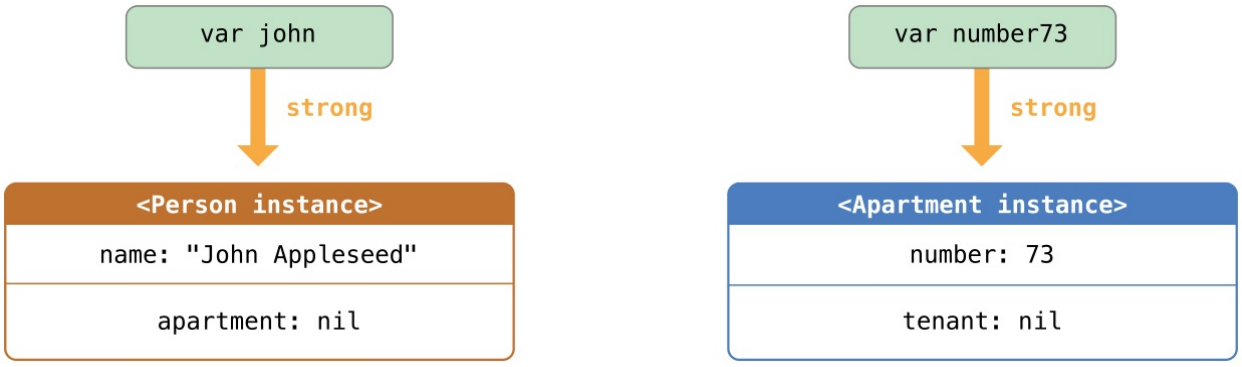
接下來的程式碼片段定義了兩個可選型別的變數 `john` 和 `number73`,並分別被設定為下面的 `Apartment` 和 `Person` 的實例。這兩個變數都被初始化為 `nil`，並為可選的：

```
var john: Person?
var number73: Apartment?
```

現在你可以創建特定的 `Person` 和 `Apartment` 實例並將類別實例賦值給 `john` 和 `number73` 變數：

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

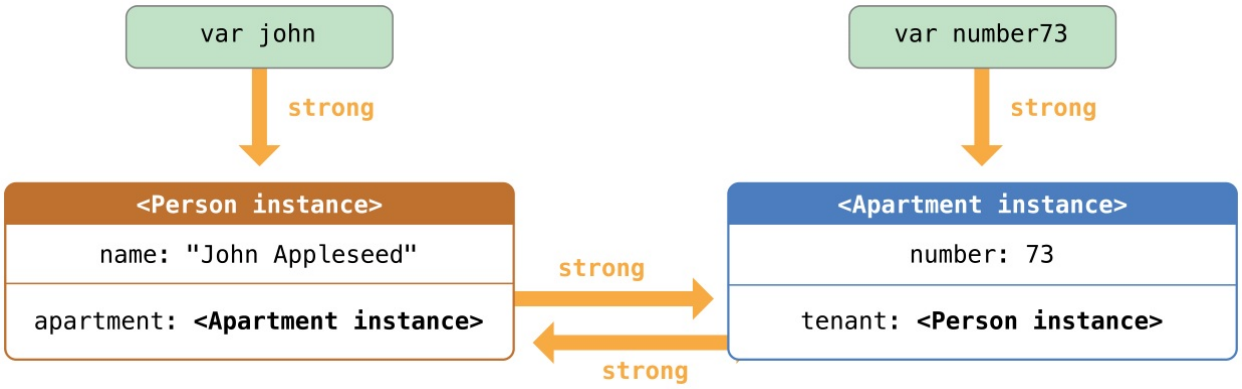
在兩個實例被創建和賦值後，下圖表現了強參考的關係。變數 `john` 現在有一個指向 `Person` 實例的強參考，而變數 `number73` 有一個指向 `Apartment` 實例的強參考：



現在你能夠將這兩個實例關聯在一起，這樣人就能有公寓住了，而公寓也有了房客。注意感嘆號是用來展開和存取可選變數 `john` 和 `number73` 中的實例，這樣實例的屬性才能被賦值：

```
john!.apartment = number73
number73!.tenant = john
```

在將兩個實例聯系在一起之後，強參考的關系如圖所示：

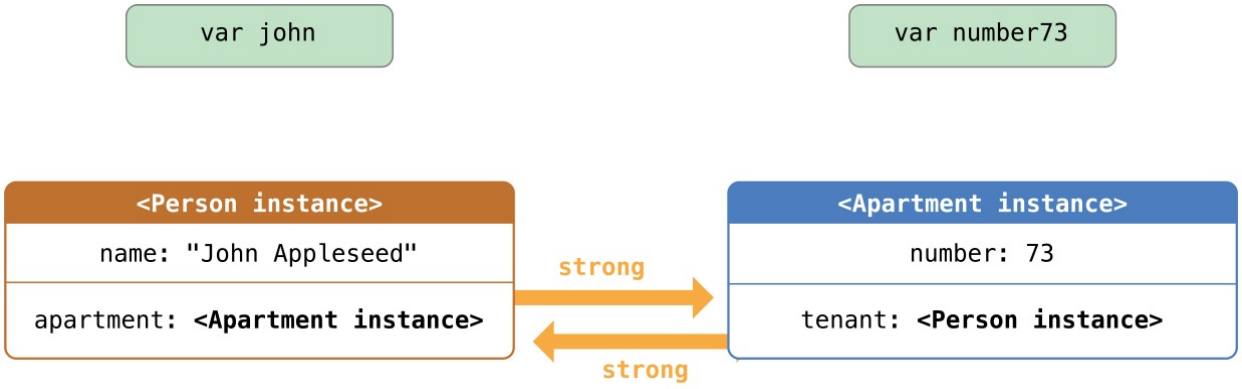


不幸的是，將這兩個實例關聯在一起之後，一個迴圈強參考被創建了。`Person` 實例現在有了一個指向 `Apartment` 實例的強參考，而 `Apartment` 實例也有了一個指向 `Person` 實例的強參考。因此，當你斷開 `john` 和 `number73` 變數所持有的強參考時，參考計數並不會降為 0，實例也不會被 ARC 銷毀：

```
john = nil
number73 = nil
```

注意，當你把這兩個變數設為 `nil` 時，沒有任何一個析構函式被呼叫。強參考迴圈阻止了 `Person` 和 `Apartment` 類別實例的銷毀，並在你的應用程式中造成了內存泄漏。

在你將 `john` 和 `number73` 賦值為 `nil` 後，強參考關系如下圖：



`Person` 和 `Apartment` 實例之間的強參考關系保留了下來並且不會被斷開。

解決實例之間的迴圈強參考

Swift 提供了兩種辦法用來解決你在使用類別的屬性時所遇到的迴圈強參考問題：弱參考（weak reference）和無主參考（unowned reference）。

弱參考和無主參考允許迴圈參考中的一個實例參考另外一個實例而不保持強參考。這樣實例能夠互相參考而不產生迴圈強參考。

對於生命周期中會變為 `nil` 的實例使用弱參考。相反的，對於初始化賦值後再也不會被賦值為 `nil` 的實例，使用無主參考。

弱參考

弱參考不會牢牢保持住參考的實例，並且不會阻止 ARC 銷毀被參考的實例。這種行為阻止了參考變為迴圈強參考。宣告屬性或者變數時，在前面加上 `weak` 關鍵字表明這是一個弱參考。

在實例的生命周期中，如果某些時候參考沒有值，那麼弱參考可以阻止迴圈強參考。如果參考總是有值，則可以使用無主參考，在無主參考中有描述。在上面 `Apartment` 的範例中，一個公寓的生命周期中，有時是沒有「居民」的，因此適合使用弱參考來解決迴圈強參考。

> 注意:
> 弱參考必須被宣告為變數，表明其值能在執行時被修改。弱參考不能被宣告為常數。

因為弱參考可以沒有值，你必須將每一個弱參考宣告為可選型別。可選型別是在 Swift 語言中推薦的用來表示可能沒有值的型別。

因為弱參考不會保持所參考的實例，即使參考存在，實例也有可能被銷毀。因此，ARC 會在參考的實例被銷毀後自動將其賦值為 `nil`。你可以像其他可選值一樣，檢查弱參考的值是否存在，你永遠也不會遇到被銷毀了而不存在的實例。

下面的範例跟上面 `Person` 和 `Apartment` 的範例一致，但是有一個重要的區別。這一次， `Apartment` 的 `tenant` 屬性被宣告為弱參考：

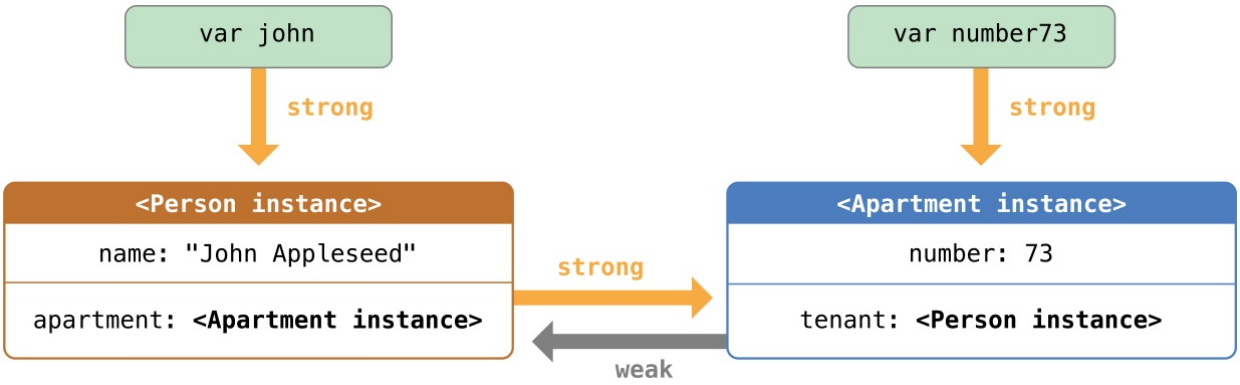
```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { println("\(name) is being deinitialized") }
}
```

```
class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    weak var tenant: Person?
    deinit { println("Apartment #\(number) is being deinitialized") }
}
```

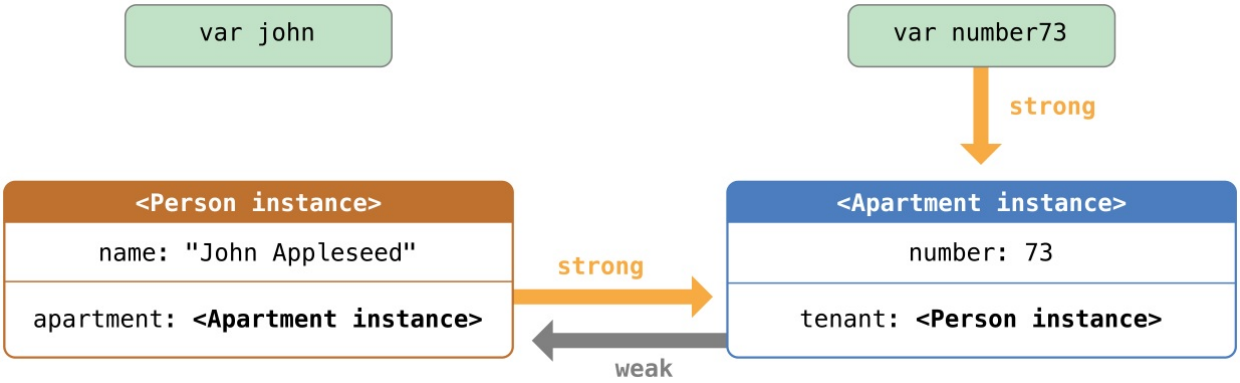
然後跟之前一樣，建立兩個變數（john和number73）之間的強參考，並關聯兩個實例：

```
var john: Person?
var number73: Apartment?

john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)

john!.apartment = number73
number73!.tenant = john
```

現在，兩個關聯在一起的實例的參考關係如下圖所示：



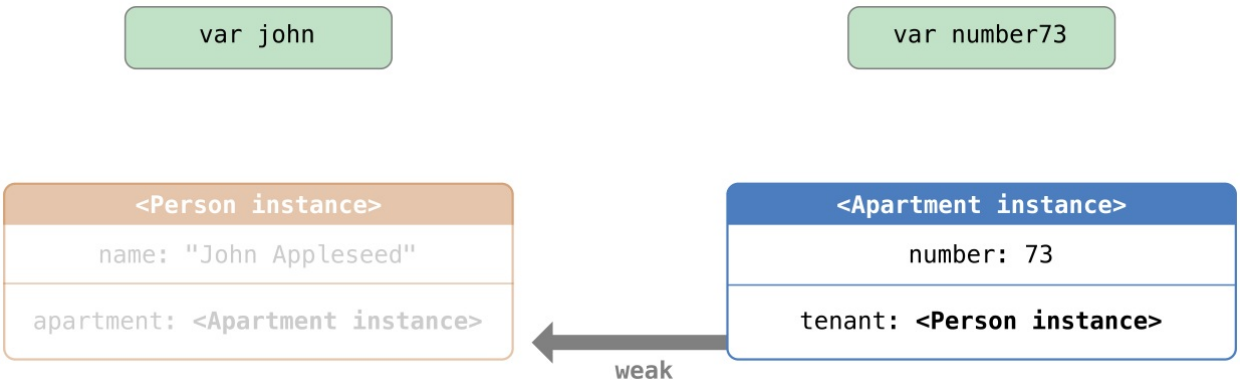
`Person` 實例依然保持對 `Apartment` 實例的強參考，但是 `Apartment` 實例只是對 `Person` 實例的弱參考。這意味著當你斷開 `john` 變數所保持的強參考時，再也沒有指向 `Person` 實例的強參考了：



由於再也沒有指向 `Person` 實例的強參考，該實例會被銷毀：

```
john = nil
// prints "John Appleseed is being deinitialized"
```

唯一剩下的指向 `Apartment` 實例的強參考來自於變數 `number73`。如果你斷開這個強參考，再也沒有指向 `Apartment` 實例的強參考了：



由於再也沒有指向 `Apartment` 實例的強參考，該實例也會被銷毀：

```
number73 = nil
// prints "Apartment #73 is being deinitialized"
```

上面的兩段程式碼展示了變數 `john` 和 `number73` 在被賦值為 `nil` 後，`Person` 實例和 `Apartment` 實例的析構函式都列印出「銷

毀」的資訊。這證明了參考迴圈被打破了。

無主參考

和弱參考類似，無主參考不會牢牢保持住參考的實例。和弱參考不同的是，無主參考是永遠有值的。因此，無主參考總是被定義為非可選型別（non-optional type）。你可以在宣告屬性或者變數時，在前面加上關鍵字 `unowned` 表示這是一個無主參考。

由於無主參考是非可選型別，你不需要在使用它的時候將它展開。無主參考總是可以被直接存取。不過 ARC 無法在實例被銷毀後將無主參考設為 `nil`，因為非可選型別的變數不允許被賦值為 `nil`。

> 注意:
> 如果你試圖在實例被銷毀後，存取該實例的無主參考，會觸發執行時錯誤。使用無主參考，你必須確保參考始終指向一個未銷毀的實例。
> 還需要注意的是如果你試圖存取實例已經被銷毀的無主參考，程式會直接崩潰，而不會發生無法預期的行為。所以你應當避免這樣的事情發生。

下面的範例定義了兩個類別，`Customer` 和 `CreditCard`，模擬了銀行客戶和客戶的信用卡。這兩個類別中，每一個都將另外一個類別的實例作為自身的屬性。這種關係會潛在的創造迴圈強參考。

`Customer` 和 `CreditCard` 之間的關係與前面弱參考範例中 `Apartment` 和 `Person` 的關係截然不同。在這個資料模型中，一個客戶可能有或者沒有信用卡，但是一張信用卡總是關聯著一個客戶。為了表示這種關係，`Customer` 類別有一個可選型別的 `card` 屬性，但是 `CreditCard` 類別有一個非可選型別的 `customer` 屬性。

此外，只能通過將一個 `number` 值和 `customer` 實例傳遞給 `CreditCard` 建構函式的方式來創建 `CreditCard` 實例。這樣可以確保當創建 `CreditCard` 實例時總是有一個 `customer` 實例與之關聯。

由於信用卡總是關聯著一個客戶，因此將 `customer` 屬性定義為無主參考，用以避免迴圈強參考：

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { println("\(name) is being deinitialized") }
}
```

```
class CreditCard {
    let number: Int
    unowned let customer: Customer
    init(number: Int, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { println("Card #\(number) is being deinitialized") }
}
```

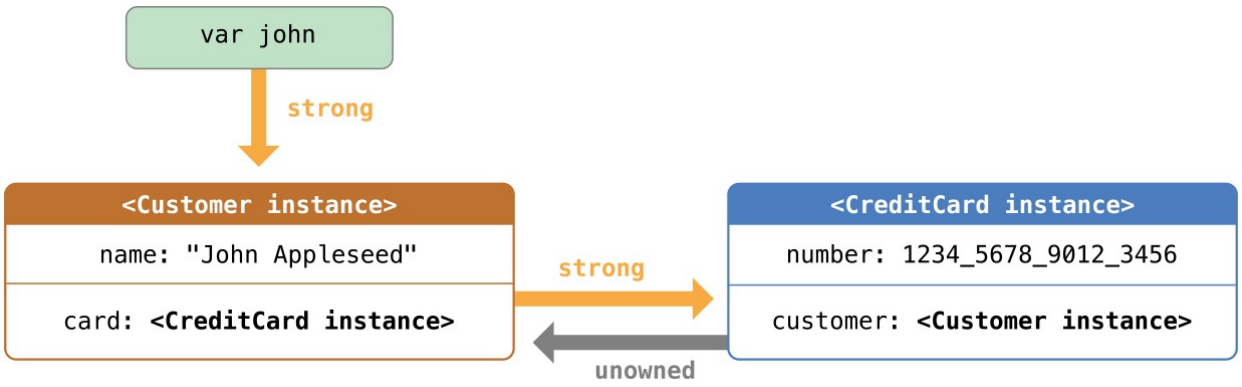
下面的程式碼片段定義了一個叫 `john` 的可選型別 `Customer` 變數，用來保存某個特定客戶的參考。由於是可選型別，所以變數被初始化為 `nil`。

```
var john: Customer?
```

現在你可以創建 `Customer` 類別的實例，用它初始化 `CreditCard` 實例，並將新創建的 `CreditCard` 實例賦值為客戶的 `card` 屬性。

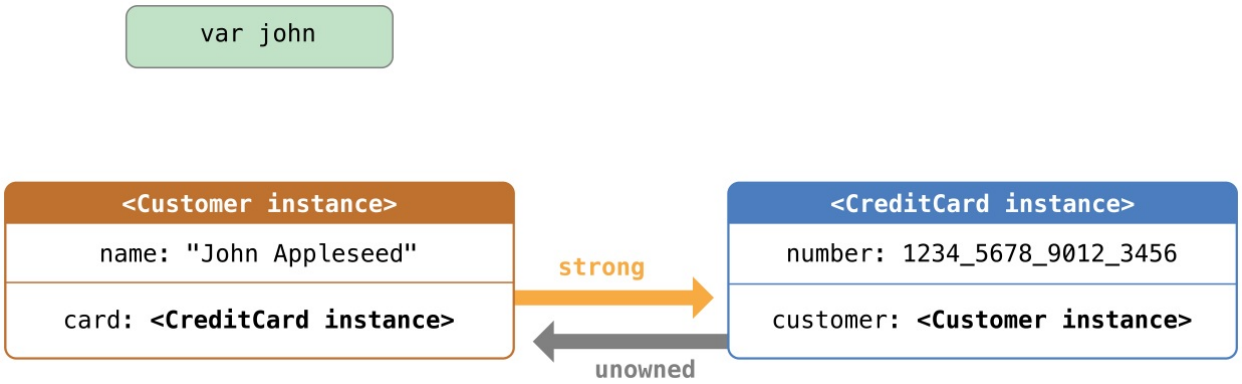
```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

在你關聯兩個實例後，它們的參考關係如下圖所示：



`Customer` 實例持有對 `CreditCard` 實例的強參考，而 `CreditCard` 實例持有對 `Customer` 實例的無主參考。

由於 `customer` 的無主參考，當你斷開 `john` 變數持有的強參考時，再也沒有指向 `Customer` 實例的強參考了：



由於再也沒有指向 `Customer` 實例的強參考，該實例被銷毀了。其後，再也沒有指向 `CreditCard` 實例的強參考，該實例也隨之被銷毀了：

```
john = nil
// prints "John Appleseed is being deinitialized"
// prints "Card #1234567890123456 is being deinitialized"
```

最後的程式碼展示了在 `john` 變數被設為 `nil` 後 `Customer` 實例和 `CreditCard` 實例的建構函式都列印出了「銷毀」的資訊。

無主參考以及隱式解析可選屬性

上面弱參考和無主參考的範例涵蓋了兩種常用的需要打破迴圈強參考的場景。

`Person` 和 `Apartment` 的範例展示了兩個屬性的值都允許為 `nil`，並會潛在的產生迴圈強參考。這種場景最適合用弱參考來解決。

`Customer` 和 `CreditCard` 的範例展示了一個屬性的值允許為 `nil`，而另一個屬性的值不允許為 `nil`，並會潛在的產生迴圈強參考。這種場景最適合通過無主參考來解決。

然而，存在著第三種場景，在這種場景中，兩個屬性都必須有值，並且初始化完成後不能為 `nil`。在這種場景中，需要一個

類別使用無主屬性，而另外一個類別使用隱式解析可選屬性。

這使兩個屬性在初始化完成後能被直接存取（不需要可選展開），同時避免了迴圈參考。這一節將為你展示如何建立這種關係。

下面的範例定義了兩個類別，`Country` 和 `City`，每個類別將另外一個類別的實例保存為屬性。在這個模型中，每個國家必須有首都，而每一個城市必須屬於一個國家。為了實作這種關係，`Country` 類別擁有一個 `capitalCity` 屬性，而 `City` 類別有一個 `country` 屬性：

```
class Country {
    let name: String
    let capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}
```

```
class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

為了建立兩個類別的依賴關係，`City` 的建構函式有一個 `Country` 實例的參數，並且將實例保存為 `country` 屬性。

`Country` 的建構函式呼叫了 `City` 的建構函式。然而，只有 `Country` 的實例完全初始化完後，`Country` 的建構函式才能把 `self` 傳給 `City` 的建構函式。（在兩段式建構過程中有具體描述）

為了滿足這種需求，通過在型別結尾處加上感嘆號（City!）的方式，將 `Country` 的 `capitalCity` 屬性宣告為隱式解析可選型別的屬性。這表示像其他可選型別一樣，`capitalCity` 屬性的預設值為 `nil`，但是不需要展開它的值就能存取它。（在隱式解析可選型別中有描述）

由於 `capitalCity` 預設值為 `nil`，一旦 `Country` 的實例在建構函式中給 `name` 屬性賦值後，整個初始化過程就完成了。這代表一旦 `name` 屬性被賦值後，`Country` 的建構函式就能參考並傳遞隱式的 `self`。`Country` 的建構函式在賦值 `capitalCity` 時，就能將 `self` 作為參數傳遞給 `City` 的建構函式。

以上的意義在於你可以通過一條語句同時創建 `Country` 和 `City` 的實例，而不產生迴圈強參考，並且 `capitalCity` 的屬性能被直接存取，而不需要通過感嘆號來展開它的可選值：

```
var country = Country(name: "Canada", capitalName: "Ottawa")
println("\(country.name)'s capital city is called \(country.capitalCity.name)")
// prints "Canada's capital city is called Ottawa"
```

在上面的範例中，使用隱式解析可選值的意義在於滿足了兩個類別建構函式的需求。`capitalCity` 屬性在初始化完成後，能像非可選值一樣使用和存取同時還避免了迴圈強參考。

閉包引起的迴圈強參考

前面我們看到了迴圈強參考環是在兩個類別實例屬性互相保持對方的強參考時產生的，還知道了如何用弱參考和無主參考來打破迴圈強參考。

迴圈強參考還會發生在當你將一個閉包賦值給類別實例的某個屬性，並且這個閉包體中又使用了實例。這個閉包體中可能存取了實例的某個屬性，例如 `self.someProperty`，或者閉包中呼叫了實例的某個方法，例如 `self.someMethod`。這兩種情況都導致了閉包「捕獲」`self`，從而產生了迴圈強參考。

迴圈強參考的產生，是因為閉包和類別相似，都是參考型別。當你把一個閉包賦值給某個屬性時，你也把一個參考賦值給了這個閉包。實質上，這跟之前的問題是一樣的－兩個強參考讓彼此一直有效。但是，和兩個類別實例不同，這次一個是類別實例，另一個是閉包。

Swift 提供了一種優雅的方法來解決這個問題，稱之為閉包占用列表（closuer capture list）。同樣的，在學習如何用閉包占用列表破壞迴圈強參考之前，先來了解一下迴圈強參考是如何產生的，這對我們是很有幫助的。

下面的範例為你展示了當一個閉包參考了 `self` 後是如何產生一個迴圈強參考的。範例中定義了一個叫 `HTMLElement` 的類別，用一種簡單的模型表示 HTML 中的一個單獨的元素：

```
class HTMLElement {

    let name: String
    let text: String?

    @lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        println("\(name) is being deinitialized")
    }

}
```

`HTMLElement` 類別定義了一個 `name` 屬性來表示這個元素的名稱，例如代表段落的"p"，或者代表換行的"br"。`HTMLElement` 還定義了一個可選屬性 `text`，用來設置和展現 HTML 元素的文字。

除了上面的兩個屬性，`HTMLElement` 還定義了一個 `lazy` 屬性 `asHTML`。這個屬性參考了一個閉包，將 `name` 和 `text` 組合成 HTML 字串片段。該屬性是 `() -> String` 型別，或者可以理解為「一個沒有參數，回傳 `String` 的函式」。

預設情況下，閉包賦值給了 `asHTML` 屬性，這個閉包回傳一個代表 HTML 標簽的字串。如果 `text` 值存在，該標簽就包含可選值 `text`；如果 `text` 不存在，該標簽就不包含文字。對於段落元素，根據 `text` 是"some text"還是 `nil`，閉包會回傳"`<p>some text</p>`"或者"`<p />`"。

可以像實例方法那樣去命名、使用 `asHTML` 屬性。然而，由於 `asHTML` 是閉包而不是實例方法，如果你想改變特定元素的 HTML 處理的話，可以用自定義的閉包來取代預設值。

> 注意:
> `asHTML` 宣告為 `lazy` 屬性，因為只有當元素確實需要處理為HTML輸出的字串時，才需要使用 `asHTML`。也就是說，在預設的閉包中可以使用 `self`，因為只有當初始化完成以及 `self` 確實存在後，才能存取 `lazy` 屬性。

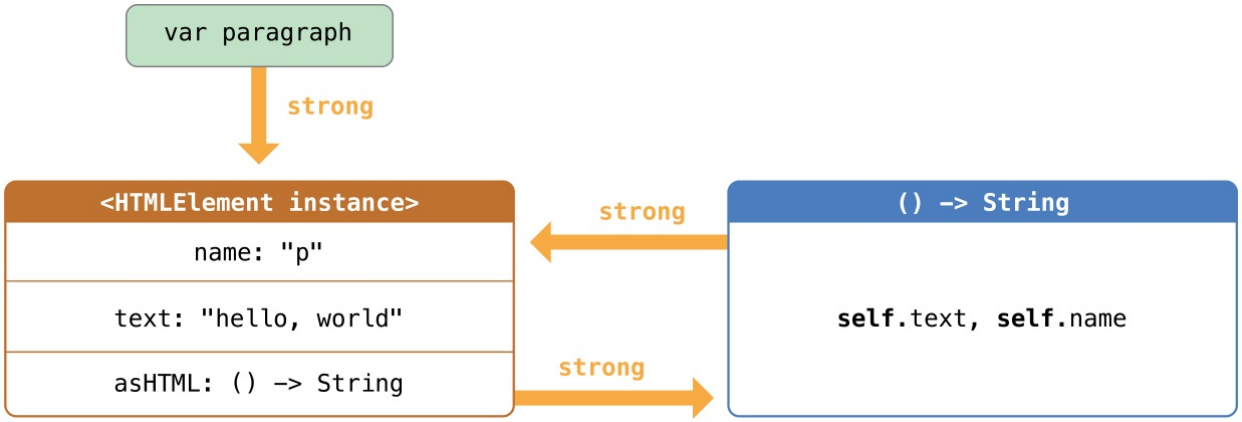
`HTMLElement` 類別只提供一個建構函式，通過 `name` 和 `text`（如果有的話）參數來初始化一個元素。該類別也定義了一個析構函式，當 `HTMLElement` 實例被銷毀時，列印一條訊息。

下面的程式碼展示了如何用 `HTMLElement` 類別創建實例並列印訊息。

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
// prints"hello, world"
```

> 注意:
> 上面的 paragraph 變數定義為 可選HTMLElement，因此我們可以賦值 nil 給它來演示迴圈強參考。

不幸的是，上面寫的 HTMLElement 類別產生了類別實例和 asHTML 預設值的閉包之間的迴圈強參考。迴圈強參考如下圖所示：



實例的 asHTML 屬性持有閉包的強參考。但是，閉包在其閉包體內使用了 self （參考了 self.name 和 self.text ），因此閉包捕獲了 self ，這意味著閉包又反過來持有了 HTMLElement 實例的強參考。這樣兩個物件就產生了迴圈強參考。（更多關於閉包捕獲值的資訊，請參考值捕獲）。

> 注意:
> 雖然閉包多次使用了 self ，它只捕獲 HTMLElement 實例的一個強參考。

如果設置 paragraph 變數為 nil ，打破它持有的 HTMLElement 實例的強參考， HTMLElement 實例和它的閉包都不會被銷毀，也是因為迴圈強參考：

```
paragraph = nil
```

注意 HTMLElementdeinitializer 中的訊息並沒有別列印，證明了 HTMLElement 實例並沒有被銷毀。

解決閉包引起的迴圈強參考

在定義閉包時同時定義捕獲列表作為閉包的一部分，通過這種方式可以解決閉包和類別實例之間的迴圈強參考。捕獲列表定義了閉包體內捕獲一個或者多個參考型別的規則。跟解決兩個類別實例間的迴圈強參考一樣，宣告每個捕獲的參考為弱參考或無主參考，而不是強參考。應當根據程式碼關系來決定使用弱參考還是無主參考。

> 注意:
> Swift 有如下要求：只要在閉包內使用 self 的成員，就要用 self.someProperty 或者 self.someMethod （而不只是 someProperty 或 someMethod ）。這提醒你可能會不小心就捕獲了 self 。

定義捕獲列表

捕獲列表中的每個元素都是由 weak 或者 unowned 關鍵字和實例的參考（如 self 或 someInstance ）成對組成。每一對都在方括號中，通過逗號分開。

捕獲列表放置在閉包參數列表和回傳型別之前：

```
@lazy var someClosure: (Int, String) -> String = {
    [unowned self] (index: Int, stringToProcess: String) -> String in
    // closure body goes here
}
```

如果閉包沒有指定參數列表或者回傳型別，則可以通過上下文推斷，那麼可以捕獲列表放在閉包開始的地方，跟著是關鍵字 `in`：

```
@lazy var someClosure: () -> String = {
    [unowned self] in
    // closure body goes here
}
```

弱參考和無主參考

當閉包和捕獲的實例總是互相參考時並且總是同時銷毀時，將閉包內的捕獲定義為無主參考。

相反的，當捕獲參考有時可能會是 `nil` 時，將閉包內的捕獲定義為弱參考。弱參考總是可選型別，並且當參考的實例被銷毀後，弱參考的值會自動置為 `nil`。這使我們可以在閉包內檢查它們是否存在。

> 注意:
> 如果捕獲的參考絕對不會置為 `nil`，應該用無主參考，而不是弱參考。

前面的 `HTMLElement` 範例中，無主參考是正確的解決迴圈強參考的方法。這樣編寫 `HTMLElement` 類別來避免迴圈強參考：

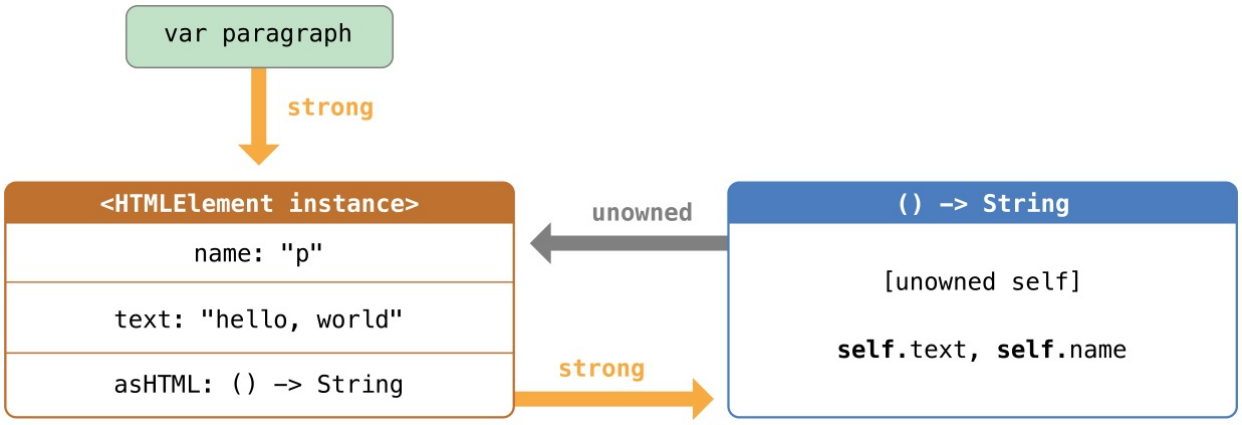
```
class HTMLElement {

    let name: String
    let text: String?

    @lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        println("\(name) is being deinitialized")
    }

}
```

上面的 `HTMLElement` 實作和之前的實作一致，只是在 `asHTML` 閉包中多了一個捕獲列表。這裡，捕獲列表是 `[unowned self]`，表示「用無主參考而不是強參考來捕獲 `self`」。

和之前一樣，我們可以創建並列印 `HTMLElement` 實例：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
// prints "<p>hello, world</p>"
```

使用捕獲列表後參考關係如下圖所示：



這一次，閉包以無主參考的形式捕獲 `self`，並不會持有 `HTMLElement` 實例的強參考。如果將 `paragraph` 賦值為 `nil`， `HTMLElement` 實例將會被銷毀，並能看到它的析構函式列印出的訊息。

```
paragraph = nil
// prints "p is being deinitialized"
```

> 翻譯：Jasonbroker 校對：numbbbbb, stanzhai

Optional Chaining

本頁包含內容：

可選鏈（Optional Chaining）是一種可以請求和呼叫屬性、方法及子腳本的過程，它的可選性體現於請求或呼叫的目標當前可能為空（`nil`）。如果可選的目標有值，那麼呼叫就會成功；相反，如果選擇的目標為空（`nil`），則這種呼叫將回傳空（`nil`）。多次請求或呼叫可以被鏈接在一起形成一個鏈，如果任何一個節點為空（`nil`）將導致整個鏈失效。

> 注意：
> Swift 的可選鏈和 Objective-C 中的訊息為空有些相像，但是 Swift 可以使用在任意型別中，並且失敗與否可以被檢測到。

可選鏈可替代強制解析

通過在想呼叫的屬性、方法、或子腳本的可選值（`optional value`）（非空）後面放一個問號，可以定義一個可選鏈。這一點很像在可選值後面放一個嘆號來強制拆得其封包內的值。它們的主要的區別在於當可選值為空時可選鏈即刻失敗，然而一般的強制解析將會引發執行時錯誤。

為了反映可選鏈可以呼叫空（`nil`），不論你呼叫的屬性、方法、子腳本等回傳的值是不是可選值，它的回傳結果都是一個可選值。你可以利用這個回傳值來檢測你的可選鏈是否呼叫成功，有回傳值即成功，回傳nil則失敗。

呼叫可選鏈的回傳結果與原本的回傳結果具有相同的型別，但是原本的回傳結果被包裝成了一個可選值，當可選鏈呼叫成功時，一個應該回傳 `Int` 的屬性將會回傳 `Int?`。

下面幾段程式碼將解釋可選鏈和強制解析的不同。

首先定義兩個類別 `Person` 和 `Residence`。

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

`Residence` 具有一個 `Int` 型別的 `numberOfRooms`，其值為 1。 `Person` 具有一個可選 `residence` 屬性，它的型別是 `Residence?`。

如果你創建一個新的 `Person` 實例，它的 `residence` 屬性由於是被定義為可選型的，此屬性將預設初始化為空：

```
let john = Person()
```

如果你想使用感嘆號（`!`）強制解析獲得這個人`residence`屬性`numberOfRooms`屬性值，將會引發執行時錯誤，因為這時沒有可以供解析的`residence`值。

```
let roomCount = john.residence!.numberOfRooms
//將導致執行時錯誤
```

當`john.residence`不是`nil`時，會執行通過，且會將`roomCount`設置為一個`int`型別的合理值。然而，如上所述，當`residence`為空時，這個程式碼將會導致執行時錯誤。

可選鏈提供了一種另一種獲得`numberOfRooms`的方法。利用可選鏈，使用問號來代替原來`!`的位置：

```
if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// 列印 "Unable to retrieve the number of rooms.
```

這告訴 Swift 來鏈接可選`residence?`屬性，如果`residence`存在則取回`numberOfRooms`的值。

因為這種嘗試獲得`numberOfRooms`的操作有可能失敗，可選鏈會回傳`Int?`型別值，或者稱作「可選`Int`」。當`residence`是空的時候（上例），選擇`Int`將會為空，因此會出先無法存取`numberOfRooms`的情況。

要注意的是，即使numberOfRooms是非可選`Int`（`Int?`）時這一點也成立。只要是通過可選鏈的請求就意味著最後`numberOfRooms`總是回傳一個`Int?`而不是`Int`。

你可以自己定義一個`Residence`實例給`john.residence`，這樣它就不再為空了：

```
john.residence = Residence()
```

`john.residence`現在有了實際存在的實例而不是nil了。如果你想使用和前面一樣的可選鏈來獲得`numberOfRoooms`，它將回傳一個包含預設值 1 的`Int?`：

```
if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// 列印 "John's residence has 1 room(s)"。
```

為可選鏈定義模型類別

你可以使用可選鏈來多層呼叫屬性，方法，和子腳本。這讓你可以利用它們之間的複雜模型來獲取更底層的屬性，並檢查是否可以成功獲取此類別底層屬性。

後面的程式碼定義了四個將在後面使用的模型類別，其中包括多層可選鏈。這些類別是由上面的`Person`和`Residence`模型通過添加一個`Room`和一個`Address`類別拓展來。

`Person`類別定義與之前相同。

```
class Person {
    var residence: Residence?
}
```

`Residence` 類別比之前複雜些。這次，它定義了一個變數 `rooms`，它被初始化為一個 `Room[]` 型別的空陣列：

```
class Residence {
    var rooms = Room[]()
    var numberOfRooms: Int {
    return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        println("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}
```

因為 `Residence` 儲存了一個 `Room` 實例的陣列，它的 `numberOfRooms` 屬性值不是一個固定的儲存值，而是通過計算而來的。 `numberOfRooms` 屬性值是由回傳 `rooms` 陣列的 `count` 屬性值得到的。

為了能快速存取 `rooms` 陣列， `Residence` 定義了一個唯讀的子腳本，通過插入陣列的元素角標就可以成功呼叫。如果該角標存在，子腳本則將該元素回傳。

`Residence` 中也提供了一個 `printNumberOfRooms` 的方法，即簡單的列印房間個數。

最後， `Residence` 定義了一個可選屬性叫 `address` （ `address?` ）。 `Address` 類別的屬性將在後面定義。 用於 `rooms` 陣列的 `Room` 類別是一個很簡單的類別，它只有一個 `name` 屬性和一個設定 `room` 名的初始化器。

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
```

這個模型中的最終類別叫做 `Address` 。它有三個型別是 `String?` 的可選屬性。前面兩個可選屬性 `buildingName` 和 `buildingNumber` 作為地址的一部分，是定義某個建築物的兩種方式。第三個屬性 `street` ，用於命名地址的街道名：

```
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if buildingName {
            return buildingName
        } else if buildingNumber {
            return buildingNumber
        } else {
            return nil
        }
    }
}
```

`Address` 類別還提供了一個 `buildingIdentifier` 的方法，它的回傳值型別為 `String?` 。這個方法檢查 `buildingName` 和 `buildingNumber` 的屬性，如果 `buildingName` 有值則將其回傳，或者如果 `buildingNumber` 有值則將其回傳，再或如果沒有一個屬性有值，回傳空。

通過可選鏈呼叫屬性

正如上面「可選鏈可替代強制解析」中所述，你可以利用可選鏈的可選值獲取屬性，並且檢查屬性是否獲取成功。然而，你不能使用可選鏈為屬性賦值。

使用上述定義的類別來創建一個人實例，並再次嘗試後去它的 `numberOfRooms` 屬性：

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// 列印 "Unable to retrieve the number of rooms。
```

由於 `john.residence` 是空，所以這個可選鏈和之前一樣失敗了，但是沒有執行時錯誤。

通過可選鏈呼叫方法

你可以使用可選鏈的來呼叫可選值的方法並檢查方法呼叫是否成功。即使這個方法沒有回傳值，你依然可以使用可選鏈來達成這一目的。

`Residence` 的 `printNumberOfRooms` 方法會列印 `numberOfRooms` 的當前值。方法如下：

```
func printNumberOfRooms(){
    println(「The number of rooms is \(numberOfRooms)」)
}
```

這個方法沒有回傳值。但是，沒有回傳值型別的函式和方法有一個隱式的回傳值型別 `Void`（參見Function Without Return Values）。

如果你利用可選鏈呼叫此方法，這個方法的回傳值型別將是 `Void?`，而不是 `Void`，因為當通過可選鏈呼叫方法時回傳值總是可選型別（optional type）。即使這個方法本身沒有定義回傳值，你也可以使用 `if` 語句來檢查是否能成功呼叫 `printNumberOfRooms` 方法：如果方法通過可選鏈呼叫成功，`printNumberOfRooms` 的隱式回傳值將會是 `Void`，如果沒有成功，將回傳 `nil`：

```
if john.residence?.printNumberOfRooms() {
    println("It was possible to print the number of rooms.")
} else {
    println("It was not possible to print the number of rooms.")
}
// 列印 "It was not possible to print the number of rooms."。
```

使用可選鏈呼叫子腳本

你可以使用可選鏈來嘗試從子腳本獲取值並檢查子腳本的呼叫是否成功，然而，你不能通過可選鏈來設置子程式碼。

> 注意：
> 當你使用可選鏈來獲取子腳本的時候，你應該將問號放在子腳本括號的前面而不是後面。可選鏈的問號一般直接跟在表達語句的後面。

下面這個範例用在 `Residence` 類別中定義的子腳本來獲取 `john.residence` 陣列中第一個房間的名字。因為 `john.residence` 現

在是 `nil`，子腳本的呼叫失敗了。

```
if let firstRoomName = john.residence?[0].name {
    println("The first room name is \(firstRoomName).")
} else {
    println("Unable to retrieve the first room name.")
}
// 列印 "Unable to retrieve the first room name."。
```

在子程式碼呼叫中可選鏈的問號直接跟在 `john.residence` 的後面，在子腳本括號的前面，因為 `john.residence` 是可選鏈試圖獲得的可選值。

如果你創建一個 `Residence` 實例給 `john.residence`，且在他的 `rooms` 陣列中有一個或多個 `Room` 實例，那麼你可以使用可選鏈通過 `Residence` 子腳本來獲取在 `rooms` 陣列中的實例了：

```
let johnsHouse = Residence()
johnsHouse.rooms += Room(name: "Living Room")
johnsHouse.rooms += Room(name: "Kitchen")
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    println("The first room name is \(firstRoomName).")
} else {
    println("Unable to retrieve the first room name.")
}
// 列印 "The first room name is Living Room."。
```

連接多層鏈接

你可以將多層可選鏈連接在一起，可以掘取模型內更下層的屬性方法和子腳本。然而多層可選鏈不能再添加比已經回傳的可選值更多的層。 也就是說：

如果你試圖獲得的型別不是可選型別，由於使用了可選鏈它將變成可選型別。 如果你試圖獲得的型別已經是可選型別，由於可選鏈它也不會提高可選性。

因此：

如果你試圖通過可選鏈獲得 `Int` 值，不論使用了多少層鏈接回傳的總是 `Int?`。 相似的，如果你試圖通過可選鏈獲得 `Int?` 值，不論使用了多少層鏈接回傳的總是 `Int?`。

下面的範例試圖獲取 `john` 的 `residence` 屬性裡的 `address` 的 `street` 屬性。這裡使用了兩層可選鏈來聯系 `residence` 和 `address` 屬性，它們兩者都是可選型別：

```
if let johnsStreet = john.residence?.address?.street {
    println("John's street name is \(johnsStreet).")
} else {
    println("Unable to retrieve the address.")
}
// 列印 "Unable to retrieve the address.」。
```

`john.residence` 的值現在包含一個 `Residence` 實例，然而 `john.residence.address` 現在是 `nil`，因此 `john.residence?.address?.street` 呼叫失敗。

從上面的範例發現，你試圖獲得 `street` 屬性值。這個屬性的型別是 `String?`。因此儘管在可選型別屬性前使用了兩層可選鏈， `john.residence?.address?.street` 的回傳值型別也是 `String?`。

如果你為 `Address` 設定一個實例來作為 `john.residence.address` 的值，並為 `address` 的 `street` 屬性設定一個實際值，你可以通過多層可選鏈來得到這個屬性值。

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence!.address = johnsAddress
```

```
if let johnsStreet = john.residence?.address?.street {
    println("John's street name is \(johnsStreet).")
} else {
    println("Unable to retrieve the address.")
}
// 列印 "John's street name is Laurel Street."。
```

值得注意的是，「`!`」符號在給 `john.residence.address` 分配 `address` 實例時的使用。`john.residence` 屬性是一個可選型別，因此你需要在它獲取 `address` 屬性之前使用 `!` 解析以獲得它的實際值。

鏈接可選回傳值的方法

前面的範例解釋了如何通過可選鏈來獲得可選型別屬性值。你也可以通過可選鏈呼叫一個回傳可選型別值的方法並按需鏈接該方法的回傳值。

下面的範例通過可選鏈呼叫了 `Address` 類別中的 `buildingIdentifier` 方法。這個方法的回傳值型別是 `String?`。如上所述，這個方法在可選鏈呼叫後最終的回傳值型別依然是 `String?`：

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    println("John's building identifier is \(buildingIdentifier).")
}
// 列印 "John's building identifier is The Larches."。
```

如果你還想進一步對方法回傳值執行可選鏈，將可選鏈問號符放在方法括號的後面：

```
if let upper = john.residence?.address?.buildingIdentifier()?.uppercaseString {
    println("John's uppercase building identifier is \(upper).")
}
// 列印 "John's uppercase building identifier is THE LARCHES."。
```

> 注意：
> 在上面的範例中，你將可選鏈問號符放在括號後面是因為你想要鏈接的可選值是 `buildingIdentifier` 方法的回傳值，不是 `buildingIdentifier` 方法本身。

翻譯：xiehurricane 校對：happyming

型別檢查（Type Casting）

本頁包含內容：

型別檢查是一種檢查類別實例的方式，並且或者也是讓實例作為它的父類別或者子類別的一種方式。

型別檢查在 Swift 中使用 `is` 和 `as` 運算子實作。這兩個運算子提供了一種簡單達意的方式去檢查值的型別或者轉換它的型別。

你也可以用來檢查一個類別是否實作了某個協定，就像在 Checking for Protocol Conformance部分講述的一樣。

定義一個類別層次作為範例

你可以將它用在類別和子類別的層次結構上，檢查特定類別實例的型別並且轉換這個類別實例的型別成為這個層次結構中的其他型別。這下面的三個程式碼段定義了一個類別層次和一個包含了幾個這些類別實例的陣列，作為型別檢查的範例。

第一個程式碼片段定義了一個新的基礎類別 `MediaItem`。這個類別為任何出現在數字媒體函式庫的媒體項提供基礎功能。特別的，它宣告了一個 `String` 型別的 `name` 屬性，和一個 `init name` 初始化器。（它假定所有的媒體項都有個名稱。）

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

下一個程式碼段定義了 `MediaItem` 的兩個子類別。第一個子類別 `Movie`，在父類別（或者說基類別）的基礎上增加了一個 `director`（導演） 屬性，和相應的初始化器。第二個類別在父類別的基礎上增加了一個 `artist`（藝術家） 屬性，和相應的初始化器：

```
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

最後一個程式碼段創建了一個陣列常數 `library`，包含兩個 `Movie` 實例和三個 `Song` 實例。 `library` 的型別是在它被初始化時根據它陣列中所包含的內容推斷來的。Swift 的型別檢測器能夠演繹出 `Movie` 和 `Song` 有共同的父類別 `MediaItem`，所以它

推斷出 `MediaItem[]` 類別作為 `library` 的型別。

```
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
// the type of "library" is inferred to be MediaItem[]
```

在幕後 `library` 裡儲存的媒體項依然是 `Movie` 和 `Song` 型別的，但是，若你迭代它，取出的實例會是 `MediaItem` 型別的，而不是 `Movie` 和 `Song` 型別的。為了讓它們作為它們本來的型別工作，你需要檢查它們的型別或者向下轉換它們的型別到其它型別，就像下面描述的一樣。

檢查型別（Checking Type）

用型別檢查運算子(`is`)來檢查一個實例是否屬於特定子型別。若實例屬於那個子型別，型別檢查運算子回傳 `true` ，否則回傳 `false` 。

下面的範例定義了兩個變數， `movieCount` 和 `songCount` ，用來計算陣列 `library` 中 `Movie` 和 `Song` 型別的實例數量。

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        ++movieCount
    } else if item is Song {
        ++songCount
    }
}

println("Media library contains \(movieCount) movies and \(songCount) songs")
// prints "Media library contains 2 movies and 3 songs"
```

示例迭代了陣列 `library` 中的所有項。每一次， `for` - `in` 迴圈設置 `item` 為陣列中的下一個 `MediaItem` 。

若當前 `MediaItem` 是一個 `Movie` 型別的實例， `item is Movie` 回傳 `true` ，相反回傳 `false` 。同樣的， `item is Song` 檢查item是否為 `Song` 型別的實例。在迴圈結束後， `movieCount` 和 `songCount` 的值就是被找到屬於各自的型別的實例數量。

向下轉型（Downcasting）

某型別的一個常數或變數可能在幕後實際上屬於一個子類別。你可以相信，上面就是這種情況。你可以嘗試向下轉到它的子型別，用型別檢查運算子(`as`)

因為向下轉型可能會失敗，型別轉型運算子帶有兩種不同形式。可選形式（ optional form） `as?` 回傳一個你試圖下轉成的型別的可選值（optional value）。強制形式 `as` 把試圖向下轉型和強制解包（force-unwraps）結果作為一個混合動作。

當你不確定向下轉型可以成功時，用型別檢查的可選形式(`as?`)。可選形式的型別檢查總是回傳一個可選值（optional value），並且若下轉是不可能的，可選值將是 `nil` 。這使你能夠檢查向下轉型是否成功。

只有你可以確定向下轉型一定會成功時，才使用強制形式。當你試圖向下轉型為一個不正確的型別時，強制形式的型別檢查會觸發一個執行時錯誤。

下面的範例，迭代了 `library` 裡的每一個 `MediaItem` ，並列印出適當的描述。要這樣做， `item` 需要真正作為 `Movie` 或

`Song`的型別來使用。不僅僅是作為`MediaItem`。為了能夠使用`Movie`或`Song`的`director`或`artist`屬性，這是必要的。

在這個示例中，陣列中的每一個`item`可能是`Movie`或`Song`。事前你不知道每個`item`的真實型別，所以這裡使用可選形式的型別檢查（`as?`）去檢查迴圈裡的每次下轉。

```
for item in library {
    if let movie = item as? Movie {
        println("Movie: '\(movie.name)', dir. \(movie.director)")
    } else if let song = item as? Song {
        println("Song: '\(song.name)', by \(song.artist)")
    }
}

// Movie: 'Casablanca', dir. Michael Curtiz
// Song: 'Blue Suede Shoes', by Elvis Presley
// Movie: 'Citizen Kane', dir. Orson Welles
// Song: 'The One And Only', by Chesney Hawkes
// Song: 'Never Gonna Give You Up', by Rick Astley
```

示例首先試圖將`item`下轉為`Movie`。因為`item`是一個`MediaItem`型別的實例，它可能是一個`Movie`；同樣，它可能是一個`Song`，或者僅僅是基類別`MediaItem`。因為不確定，`as?`形式在試圖下轉時將返還一個可選值。`item as Movie`的回傳值是`Movie?`型別或「optional `Movie`」。

當向下轉型為`Movie`應用在兩個`Song`實例時將會失敗。為了處理這種情況，上面的範例使用了可選綁定（optional binding）來檢查可選`Movie`真的包含一個值（這個是為了判斷下轉是否成功。）可選綁定是這樣寫的「`if let movie = item as? Movie`」，可以這樣解讀：

「嘗試將`item`轉為`Movie`型別。若成功，設置一個新的臨時常數`movie`來儲存回傳的可選`Movie`」

若向下轉型成功，然後`movie`的屬性將用於列印一個`Movie`實例的描述，包括它的導演的名字`director`。當`Song`被找到時，一個相近的原理被用來檢測`Song`實例和列印它的描述。

> 注意：
> 轉換沒有真的改變實例或它的值。潛在的根本的實例保持不變；只是簡單地把它作為它被轉換成的類別來使用。

`Any`和`AnyObject`的型別檢查

Swift為不確定型別提供了兩種特殊型別別名：

- `AnyObject`可以代表任何class型別的實例。
- `Any`可以表示任何型別，除了方法型別（function types）。

> 注意：
> 只有當你明確的需要它的行為和功能時才使用`Any`和`AnyObject`。在你的程式碼裡使用你期望的明確的型別總是更好的。

`AnyObject`型別

當需要在工作中使用 Cocoa APIs，它一般接收一個`AnyObject[]`型別的陣列，或者說「一個任何物件型別的陣列」。這是因為 Objective-C 沒有明確的型別化陣列。但是，你常常可以確定包含在僅從你知道的 API 資訊提供的這樣一個陣列中的物件的型別。

在這些情況下，你可以使用強制形式的型別檢查(`as`)來下轉在陣列中的每一項到比`AnyObject`更明確的型別，不需要可選解析（optional unwrapping）。

下面的示例定義了一個`AnyObject[]`型別的陣列並填入三個`Movie`型別的實例：

```
let someObjects: AnyObject[] = [
    Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),
    Movie(name: "Moon", director: "Duncan Jones"),
    Movie(name: "Alien", director: "Ridley Scott")
]
```

因為知道這個陣列只包含 `Movie` 實例，你可以直接用(`as`)下轉並解包到不可選的 `Movie` 型別（ps：其實就是我們常用的正常型別，這裡是為了和可選型別相對比）。

```
for object in someObjects {
    let movie = object as Movie
    println("Movie: '\(movie.name)', dir. \(movie.director)")
}
// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick
// Movie: 'Moon', dir. Duncan Jones
// Movie: 'Alien', dir. Ridley Scott
```

為了變為一個更短的形式，下轉 `someObjects` 陣列為 `Movie[]` 型別來代替下轉每一項方式。

```
for movie in someObjects as Movie[] {
    println("Movie: '\(movie.name)', dir. \(movie.director)")
}
// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick
// Movie: 'Moon', dir. Duncan Jones
// Movie: 'Alien', dir. Ridley Scott
```

`Any` 型別

這裡有個示例，使用 `Any` 型別來和混合的不同型別一起工作，包括非 `class` 型別。它創建了一個可以儲存 `Any` 型別的陣列 `things` 。

```
var things = Any[]()

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
```

`things` 陣列包含兩個 `Int` 值，2個 `Double` 值，1個 `String` 值，一個元組 `(Double, Double)` ，Ivan Reitman 導演的電影「Ghostbusters」。

你可以在 `switch` `cases` 裡用 `is` 和 `as` 運算子來發覺只知道是 `Any` 或 `AnyObject` 的常數或變數的型別。 下面的示例迭代 `things` 陣列中的每一項的並用 `switch` 語句查找每一項的型別。這幾種 `switch` 語句的情形綁定它們匹配的值到一個規定型別的常數，讓它們可以列印它們的值：

```
for thing in things {
    switch thing {
    case 0 as Int:
        println("zero as an Int")
    case 0 as Double:
        println("zero as a Double")
    case let someInt as Int:
        println("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        println("a positive double value of \(someDouble)")
    case is Double:
```

```
            println("some other double value that I don't want to print")
        case let someString as String:
            println("a string value of \"\(someString)\"")
        case let (x, y) as (Double, Double):
            println("an (x, y) point at \(x), \(y)")
        case let movie as Movie:
            println("a movie called '\(movie.name)', dir. \(movie.director)")
        default:
            println("something else")
        }
    }

    // zero as an Int
    // zero as a Double
    // an integer value of 42
    // a positive double value of 3.14159
    // a string value of "hello"
    // an (x, y) point at 3.0, 5.0
    // a movie called 'Ghostbusters', dir. Ivan Reitman
```

> 注意：
> 在一個switch語句的case中使用強制形式的型別檢查運算子（as, 而不是 as?）來檢查和轉換到一個明確的型別。在switch case 語句的內容中這種檢查總是安全的。

> 翻譯：Lin-H 校對：shinyzhu

型別嵌套

本頁包含內容：

- 型別嵌套實例
- 型別嵌套的參考

列舉型別常被用於實作特定類別或結構的功能。也能夠在有多種變數型別的環境中，方便地定義通用類別或結構來使用，為了實作這種功能，Swift允許你定義型別嵌套，可以在列舉型別、類別和結構中定義支援嵌套的型別。

要在一個型別中嵌套另一個型別，將需要嵌套的型別的定義寫在被嵌套型別的區域{}內，而且可以根據需要定義多級嵌套。

型別嵌套實例

下面這個範例定義了一個結構 `BlackjackCard` (二十一點)，用來模擬 `BlackjackCard` 中的撲克牌點數。 `BlackjackCard` 結構包含2個嵌套定義的列舉型別 `Suit` 和 `Rank` 。

在 `BlackjackCard` 規則中， `Ace` 牌可以表示1或者11， `Ace` 牌的這一特征用一個嵌套在列舉型 `Rank` 的結構 `Values` 來表示。

```
struct BlackjackCard {
    // 嵌套定義列舉型Suit
    enum Suit: Character {
       case Spades = "♠", Hearts = "♡", Diamonds = "◇", Clubs = "♣"
    }

    // 嵌套定義列舉型Rank
    enum Rank: Int {
       case Two = 2, Three, Four, Five, Six, Seven, Eight, Nine, Ten
       case Jack, Queen, King, Ace
       struct Values {
           let first: Int, second: Int?
       }
       var values: Values {
        switch self {
        case .Ace:
            return Values(first: 1, second: 11)
        case .Jack, .Queen, .King:
            return Values(first: 10, second: nil)
        default:
            return Values(first: self.toRaw(), second: nil)
            }
       }
    }

    // BlackjackCard 的屬性和方法
    let rank: Rank, suit: Suit
    var description: String {
    var output = "suit is \(suit.toRaw()),"
        output += " value is \(rank.values.first)"
        if let second = rank.values.second {
            output += " or \(second)"
        }
        return output
    }
}
```

列舉型的 `Suit` 用來描述撲克牌的四種花色，並分別用一個 `Character` 型別的值代表花色符號。

列舉型的 `Rank` 用來描述撲克牌從 `Ace` ~10, `J` , `Q` , `K` ,13張牌，並分別用一個 `Int` 型別的值表示牌的面值。(這個 `Int` 型別的值

不適用於 `Ace`，`J`，`Q`，`K` 的牌)。

如上文所提到的，列舉型 `Rank` 在自己內部定義了一個嵌套結構 `Values`。這個結構包含兩個變數，只有 `Ace` 有兩個數值，其余牌都只有一個數值。結構 `Values` 中定義的兩個屬性：

`first`，為 `Int` `second`，為 `Int?`，或「optional `Int`」

`Rank` 定義了一個計算屬性 `values`，這個計算屬性會根據牌的面值，用適當的數值去初始化 `Values` 實例，並賦值給 `values`。對於 `J`，`Q`，`K`，`Ace` 會使用特殊數值，對於數字面值的牌使用 `Int` 型別的值。

`BlackjackCard` 結構自身有兩個屬性— `rank` 與 `suit`，也同樣定義了一個計算屬性 `description`，`description` 屬性用 `rank` 和 `suit` 的中內容來構建對這張撲克牌名字和數值的描述，並用可選型別 `second` 來檢查是否存在第二個值，若存在，則在原有的描述中增加對第二數值的描述。

因為 `BlackjackCard` 是一個沒有自定義建構函式的結構，在Memberwise Initializers for Structure Types中知道結構有預設的成員建構函式，所以你可以用預設的 `initializer` 去初始化新的常數 `theAceOfSpades`：

```
let theAceOfSpades = BlackjackCard(rank: .Ace, suit: .Spades)
println("theAceOfSpades: \(theAceOfSpades.description)")
// 列印出 "theAceOfSpades: suit is ♠, value is 1 or 11"
```

儘管 `Rank` 和 `Suit` 嵌套在 `BlackjackCard` 中，但仍可被參考，所以在初始化實例時能夠通過列舉型別中的成員名稱單獨參考。在上面的範例中 `description` 屬性能正確得輸出對 `Ace` 牌有1和11兩個值。

型別嵌套的參考

在外部對嵌套型別的參考，以被嵌套型別的名字為前綴，加上所要參考的屬性名：

```
let heartsSymbol = BlackjackCard.Suit.Hearts.toRaw()
// 紅心的符號 為 "♡"
```

對於上面這個範例，這樣可以使 `Suit`，`Rank`，和 `Values` 的名字盡可能的短，因為它們的名字會自然的由被定義的上下文來限定。

preview

> 翻譯：lyuka 校對：Hawstein

擴展（Extensions）

本頁包含內容：

擴展就是向一個已有的類別、結構或列舉型別添加新功能（functionality）。這包括在沒有權限獲取原始源程式碼的情況下擴展型別的能力（即逆向建模）。擴展和 Objective-C 中的分類別（categories）類似。（不過與Objective-C不同的是，Swift 的擴展沒有名字。）

Swift 中的擴展可以：

- 添加計算型屬性和計算靜態屬性
- 定義實例方法和型別方法
- 提供新的建構器
- 定義下標
- 定義和使用新的嵌套型別
- 使一個已有型別符合某個協定

> 注意：
> 如果你定義了一個擴展向一個已有型別添加新功能，那麼這個新功能對該型別的所有已有實例中都是可用的，即使它們是在你的這個擴展的前面定義的。

擴展語法（Extension Syntax）

宣告一個擴展使用關鍵字 `extension`：

```
extension SomeType {
    // 加到SomeType的新功能寫到這裡
}
```

一個擴展可以擴展一個已有型別，使其能夠適配一個或多個協定（protocol）。當這種情況發生時，協定的名字應該完全按照類別或結構的名字的方式進行書寫：

```
extension SomeType: SomeProtocol, AnotherProctocol {
    // 協定實作寫到這裡
}
```

按照這種方式添加的協定遵循者（protocol conformance）被稱之為在擴展中添加協定遵循者

計算型屬性（Computed Properties）

擴展可以向已有型別添加計算型實例屬性和計算型型別屬性。下面的範例向 Swift 的內建 `Double` 型別添加了5個計算型實例屬性，從而提供與距離單位協作的基本支援。

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m : Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
println("One inch is \(oneInch) meters")
// 列印輸出："One inch is 0.0254 meters"
let threeFeet = 3.ft
println("Three feet is \(threeFeet) meters")
// 列印輸出："Three feet is 0.914399970739201 meters"
```

這些計算屬性表達的含義是把一個 `Double` 型的值看作是某單位下的長度值。即使它們被實作為計算型屬性，但這些屬性仍可以接一個帶有dot語法的浮點型字面值，而這恰恰是使用這些浮點型字面量實作距離轉換的方式。

在上述範例中，一個 `Double` 型的值 `1.0` 被用來表示「1米」。這就是為什麼 `m` 計算型屬性回傳 `self` ——表達式 `1.m` 被認為是計算 `1.0` 的 `Double` 值。

其它單位則需要一些轉換來表示在米下測量的值。1千米等於1,000米，所以 `km` 計算型屬性要把值乘以 `1_000.00` 來轉化成單位米下的數值。類似地，1米有3.28024英尺，所以 `ft` 計算型屬性要把對應的 `Double` 值除以 `3.28024` 來實作英尺到米的單位換算。

這些屬性是唯讀的計算型屬性，所有從簡考慮它們不用 `get` 關鍵字表示。它們的回傳值是 `Double` 型，而且可以用於所有接受 `Double` 的數學計算中：

```
let aMarathon = 42.km + 195.m
println("A marathon is \(aMarathon) meters long")
// 列印輸出："A marathon is 42495.0 meters long"
```

> 注意：
> 擴展可以添加新的計算屬性，但是不可以添加儲存屬性，也不可以向已有屬性添加屬性觀測器(property observers)。

建構器（Initializers）

擴展可以向已有型別添加新的建構器。這可以讓你擴展其它型別，將你自己的定制型別作為建構器參數，或者提供該型別的原始實作中沒有包含的額外初始化選項。

擴展能向類別中添加新的便利建構器，但是它們不能向類別中添加新的指定建構器或析構函式。指定建構器和析構函式必須總是由原始的類別實作來提供。

> 注意：
> 如果你使用擴展向一個值型別添加一個建構器，該建構器向所有的儲存屬性提供預設值，而且沒有定義任何定制建構器（custom initializers），那麼對於來自你的擴展建構器中的值型別，你可以呼叫預設建構器(default initializers)和逐一成員建構器(memberwise initializers)。
> 正如在值型別的建構器授權中描述的，如果你已經把建構器寫成值型別原始實作的一部分，上述規則不再適用。

下面的範例定義了一個用於描述幾何矩形的定制結構 `Rect` 。這個範例同時定義了兩個輔助結構 `Size` 和 `Point` ，它們都把 `0.0` 作為所有屬性的預設值：

```
struct Size {
```

```
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}
```

因為結構 `Rect` 提供了其所有屬性的預設值，所以正如預設建構器中描述的，它可以自動接受一個預設的建構器和一個成員級建構器。這些建構器可以用於建構新的 `Rect` 實例：

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
```

你可以提供一個額外的使用特殊中心點和大小的建構器來擴展 `Rect` 結構：

```
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

這個新的建構器首先根據提供的 `center` 和 `size` 值計算一個合適的原點。然後呼叫該結構自動的成員建構器 `init(origin:size:)`，該建構器將新的原點和大小存到了合適的屬性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect的原點是 (2.5, 2.5)，大小是 (3.0, 3.0)
```

> 注意：
> 如果你使用擴展提供了一個新的建構器，你依舊有責任保證建構過程能夠讓所有實例完全初始化。

方法（Methods）

擴展可以向已有型別添加新的實例方法和型別方法。下面的範例向 `Int` 型別添加一個名為 `repetitions` 的新實例方法：

```
extension Int {
    func repetitions(task: () -> ()) {
        for i in 0..self {
            task()
        }
    }
}
```

這個 `repetitions` 方法使用了一個 `() -> ()` 型別的單參數（single argument），表明函式沒有參數而且沒有回傳值。

定義該擴展之後，你就可以對任意整數呼叫 `repetitions` 方法,實作的功能則是多次執行某任務：

```
3.repetitions({
    println("Hello!")
    })
```

```
// Hello!
// Hello!
// Hello!
```

可以使用 trailing 閉包使呼叫更加簡潔：

```
3.repetitions{
    println("Goodbye!")
}
// Goodbye!
// Goodbye!
// Goodbye!
```

修改實例方法（Mutating Instance Methods）

通過擴展添加的實例方法也可以修改該實例本身。結構和列舉型別中修改 `self` 或其屬性的方法必須將該實例方法標注為 `mutating`，正如來自原始實作的修改方法一樣。

下面的範例向Swift的 `Int` 型別添加了一個新的名為 `square` 的修改方法，來實作一個原始值的平方計算：

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt 現在值是 9
```

下標（Subscripts）

擴展可以向一個已有型別添加新下標。這個範例向Swift內建型別 `Int` 添加了一個整型下標。該下標 `[n]` 回傳十進制數字從右向左數的第n個數字

- 123456789[0]回傳9
- 123456789[1]回傳8

...等等

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
            for _ in 1...digitIndex {
                decimalBase *= 10
            }
        return (self / decimalBase) % 10
    }
}
746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

如果該 `Int` 值沒有足夠的位數，即下標越界，那麼上述實作的下標會回傳0，因為它會在數字左邊自動補0：

```
746381295[9]
//returns 0, 即等同於：
0746381295[9]
```

嵌套型別（Nested Types）

擴展可以向已有的類別、結構和列舉添加新的嵌套型別：

```
extension Character {
    enum Kind {
        case Vowel, Consonant, Other
    }
    var kind: Kind {
        switch String(self).lowercaseString {
        case "a", "e", "i", "o", "u":
            return .Vowel
        case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
             "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
            return .Consonant
        default:
            return .Other
        }
    }
}
```

該範例向 `Character` 添加了新的嵌套列舉。這個名為 `Kind` 的列舉表示特定字元的型別。具體來說，就是表示一個標準的拉丁腳本中的字元是母音還是子音（不考慮口語和地方變種），或者是其它型別。

這個類別子還向 `Character` 添加了一個新的計算實例屬性，即 `kind`，用來回傳合適的 `Kind` 列舉成員。

現在，這個嵌套列舉可以和一個 `Character` 值聯合使用了：

```
func printLetterKinds(word: String) {
    println("'\\(word)' is made up of the following kinds of letters:")
    for character in word {
        switch character.kind {
        case .Vowel:
            print("vowel ")
        case .Consonant:
            print("consonant ")
        case .Other:
            print("other ")
        }
    }
    print("\n")
}
printLetterKinds("Hello")
// 'Hello' is made up of the following kinds of letters:
// consonant vowel consonant consonant vowel
```

函式 `printLetterKinds` 的輸入是一個 `String` 值並對其字元進行迭代。在每次迭代過程中，考慮當前字元的 `kind` 計算屬性，並列印出合適的類別別描述。所以 `printLetterKinds` 就可以用來列印一個完整單詞中所有字母的型別，正如上述單詞 `"hello"` 所展示的。

> 注意：
> 由於已知 `character.kind` 是 `Character.Kind` 型，所以 `Character.Kind` 中的所有成員值都可以使用 `switch` 語句裡的形式簡寫，比如使用 `.Vowel` 代替 `Character.Kind.Vowel`

> 翻譯：geek5nan 校對：dabing1022

協定

本頁包含內容：

`Protocol(協定)` 用於統一方法和屬性的名稱，而不實作任何功能。 `協定` 能夠被類別，列舉，結構實作，滿足協定要求的類別，列舉，結構被稱為協定的 `遵循者` 。

`遵循者` 需要提供 `協定` 指定的成員，如屬性，方法，運算子，下標等。

協定的語法

`協定` 的定義與類別，結構，列舉的定義非常相似，如下所示：

```
protocol SomeProtocol {
    // 協定內容
}
```

在類別，結構，列舉的名稱後加上 `協定名稱` ，中間以冒號 `:` 分隔即可實作協定；實作多個協定時，各協定之間用逗號 `,` 分隔，如下所示：

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // 結構內容
}
```

當某個類別含有父類別的同時並實作了協定，應當把父類別放在所有的協定之前，如下所示：

```
class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {
    // 類別的內容
}
```

屬性要求

協定 能夠要求其 遵循者 必須含有一些特定名稱和型別的 實例屬性(instance property) 或 類別屬性 (type property)，也能夠要求屬性具有 (設置權限)settable 和 (存取權限)gettable，但它不要求 屬性 是 儲存型屬性(stored property) 還是 計算型屬性(calculate property)。

如果協定要求屬性具有設置權限和存取權限，那常數儲存型屬性或者唯讀計算型屬性都無法滿足此要求。如果協定只要求屬性具有存取權限，那任何型別的屬性都可以滿足此要求，無論這些屬性是否具有設置權限。

通常前綴 var 關鍵字將屬性宣告為變數。在屬性宣告後寫上 { get set } 表示屬性為可讀寫的。 { get } 用來表示屬性為可讀的。即使你為可讀的屬性實作了 setter 方法，它也不會出錯。

```
protocol SomeProtocol {
    var musBeSettable : Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

用類別來實作協定時，使用 class 關鍵字來表示該屬性為類別成員；用結構或列舉實作協定時，則使用 static 關鍵字來表示：

```
protocol AnotherProtocol {
    class var someTypeProperty: Int { get set }
}

protocol FullyNamed {
    var fullName: String { get }
}
```

FullyNamed 協定含有 fullName 屬性。因此其 遵循者 必須含有一個名為 fullName，型別為 String 的可讀屬性。

```
struct Person: FullyNamed{
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
//john.fullName 為 "John Appleseed"
```

Person 結構含有一個名為 fullName 的 儲存型屬性，完整的 遵循 了協定。(若協定未被完整遵循，編譯時則會報錯)。

如下所示， Startship 類別 遵循 了 FullyNamed 協定：

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil ) {
        self.anme = name
        self.prefix = prefix
    }
    var fullName: String {
    return (prefix ? prefix ! + " " : " ") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName == "USS Enterprise"
```

Starship 類別將 fullName 實作為可讀的 計算型屬性。它的每一個實例都有一個名為 name 的必備屬性和一個名為 prefix 的可選屬性。 當 prefix 存在時，將 prefix 插入到 name 之前來為 Starship 構建 fullName。

方法要求

協定 能夠要求其 遵循者 必備某些特定的 實例方法 和 類別方法 。協定方法的宣告與普通方法宣告相似，但它不需要 方法 內容。

> 注意： 協定方法支援 變長參數(variadic parameter) ，不支援 預設參數(default parameter) 。

前綴 class 關鍵字表示協定中的成員為 類別成員 ；當協定用於被 列舉 或 結構 遵循時，則使用 static 關鍵字。如下所示：

```
protocol SomeProtocol {
    class func someTypeMethod()
}

protocol RandomNumberGenerator {
    func random() -> Double
}
```

RandomNumberGenerator 協定要求其 遵循者 必須擁有一個名為 random ， 回傳值型別為 Double 的實例方法。(我們假設隨機數在[0，1]區間內)。

LinearCongruentialGenerator 類別 遵循 了 RandomNumberGenerator 協定，並提供了一個叫做線性同余生成器*(linear congruential generator)*的偽隨機數算法。

```
class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c) % m)
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
println("Here's a random number: \(generator.random())")
// 輸出 : "Here's a random number: 0.37464991998171"
println("And another one: \(generator.random())")
// 輸出 : "And another one: 0.729023776863283"
```

突變方法要求

能在 方法 或 函式 內部改變實例型別的方法稱為 突變方法 。在 值型別(Value Type) (譯者注：特指結構和列舉)中的的 函式 前綴加上 mutating 關鍵字來表示該函式允許改變該實例和其屬性的型別。 這一變換過程在實例方法(Instance Methods)章節中有詳細描述。

(譯者注：類別中的成員為 參考型別*(Reference Type)* ，可以方便的修改實例及其屬性的值而無需改變型別；而 結構 和 列舉 中的成員均為 值型別*(Value Type)* ，修改變數的值就相當於修改變數的型別，而 *Swift* 預設不允許修改型別，因此需要前綴 *mutating* 關鍵字用來表示該 函式 中能夠修改型別)

> 注意： 用 class 實作協定中的 mutating 方法時，不用寫 mutating 關鍵字；用 結構， 列舉 實作協定中的 mutating 方法時，必須寫 mutating 關鍵字。

如下所示， Togglable 協定含有 toggle 函式。根據函式名稱推測， toggle 可能用於切換或恢復某個屬性的狀態。 mutating 關鍵字表示它為 突變方法 ：

```
protocol Togglable {
    mutating func toggle()
}
```

當使用 列舉 或 結構 來實作 Togglabl 協定時，必須在 toggle 方法前加上 mutating 關鍵字。

如下所示， `OnOffSwitch` 列舉 `遵循` 了 `Togglable` 協定， `On` ， `Off` 兩個成員用於表示當前狀態

```
enum OnOffSwitch: Togglable {
    case Off, On
    mutating func toggle() {
        switch self {
        case Off:
            self = On
        case On:
            self = Off
        }
    }
}
var lightSwitch = OnOffSwitch.Off
lightSwitch.toggle()
//lightSwitch 現在的值為 .On
```

協定型別

`協定` 本身不實作任何功能，但你可以將它當做 `型別` 來使用。

使用場景：

- 作為函式，方法或建構器中的參數型別，回傳值型別
- 作為常數，變數，屬性的型別
- 作為陣列，字典或其他容器中的元素型別

> 注意： 協定型別應與其他型別(Int，Double，String)的寫法相同，使用駝峰式

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) +1
    }
}
```

這裡定義了一個名為 `Dice` 的類別，用來代表桌游中的N個面的骰子。

`Dice` 含有 `sides` 和 `generator` 兩個屬性，前者用來表示骰子有幾個面，後者為骰子提供一個隨機數生成器。由於後者為 `RandomNumberGenerator` 的協定型別。所以它能夠被賦值為任意 `遵循` 該協定的型別。

此外，使用 `建構器(init)` 來代替之前版本中的 `setup` 操作。建構器中含有一個名為 `generator` ，型別為 `RandomNumberGenerator` 的形參，使得它可以接收任意遵循 `RandomNumberGenerator` 協定的型別。

`roll` 方法用來模擬骰子的面值。它先使用 `generator` 的 `random` 方法來創建一個[0-1]區間內的隨機數種子，然後加工這個隨機數種子生成骰子的面值。

如下所示， `LinearCongruentialGenerator` 的實例作為隨機數生成器傳入 `Dice` 的 `建構器`

```
var d6 = Dice(sides: 6,generator: LinearCongruentialGenerator())
for _ in 1...5 {
    println("Random dice roll is \(d6.roll())")
}
//輸出結果
```

```
//Random dice roll is 3
//Random dice roll is 5
//Random dice roll is 4
//Random dice roll is 5
//Random dice roll is 4
```

委托(代理)模式

委托是一種設計模式，它允許類別或結構將一些需要它們負責的功能 `交由(委托)` 給其他的型別。

委托模式的實作很簡單： 定義 `協定` 來 `封裝` 那些需要被委托的 `函式和方法`， 使其 `遵循者` 擁有這些被委托的 `函式和方法` 。

委托模式可以用來響應特定的動作或接收外部資料源提供的資料，而無需要知道外部資料源的型別。

下文是兩個基於骰子遊戲的協定：

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}

protocol DiceGameDelegate {
    func gameDidStart(game: DiceGame)
    func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll:Int)
    func gameDidEnd(game: DiceGame)
}
```

`DiceGame` 協定可以在任意含有骰子的遊戲中實作， `DiceGameDelegate` 協定可以用來追蹤 `DiceGame` 的遊戲過程。

如下所示， `SnakesAndLadders` 是 `Snakes and Ladders` (譯者注：控制流程章節有該遊戲的詳細介紹)遊戲的新版本。新版本使用 `Dice` 作為骰子，並且實作了 `DiceGame` 和 `DiceGameDelegate` 協定

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dic = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: Int[]
    init() {
        board = Int[](count: finalSquare + 1, repeatedValue: 0)
        board[03] = +08; board[06] = +11; borad[09] = +09; board[10] = +02
        borad[14] = -10; board[19] = -11; borad[22] = -02; board[24] = -08
    }
     var delegate: DiceGameDelegate?
     func play() {
         square = 0
         delegate?.gameDidStart(self)
         gameLoop: while square != finalSquare {
             let diceRoll = dice.roll()
             delegate?.game(self,didStartNewTurnWithDiceRoll: diceRoll)
             switch square + diceRoll {
             case finalSquare:
                 break gameLoop
             case let newSquare where newSquare > finalSquare:
                 continue gameLoop
             default:
             square += diceRoll
             square += board[square]
             }
         }
         delegate?.gameDIdEnd(self)
     }
}
```

遊戲的 初始化設置(setup) 被 SnakesAndLadders 類別的 建構器(initializer) 實作。所有的遊戲邏輯被轉移到了 play 方法中。

> 注意： 因為 delegate 並不是該遊戲的必備條件， delegate 被定義為遵循 DiceGameDelegate 協定的可選屬性

DicegameDelegate 協定提供了三個方法用來追蹤遊戲過程。被放置於遊戲的邏輯中，即 play() 方法內。分別在遊戲開始時，新一輪開始時，遊戲結束時被呼叫。

因為 delegate 是一個遵循 DiceGameDelegate 的可選屬性，因此在 play() 方法中使用了 可選鏈 來呼叫委托方法。若 delegate 屬性為 nil， 則委托呼叫優雅地失效。若 delegate 不為 nil，則委托方法被呼叫

如下所示， DiceGameTracker 遵循了 DiceGameDelegate 協定

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            println("Started a new game of Snakes and Ladders")
        }
        println("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        ++numberOfTurns
        println("Rolled a \(diceRoll)")
    }
    func gameDidEnd(game: DiceGame) {
        println("The game lasted for \(numberOfTurns) turns")
    }
}
```

DiceGameTracker 實作了 DiceGameDelegate 協定的方法要求，用來記錄遊戲已經進行的輪數。 當遊戲開始時， numberOfTurns 屬性被賦值為0；在每新一輪中遞加；遊戲結束後，輸出列印遊戲的總輪數。

gameDidStart 方法從 game 參數獲取遊戲資訊並輸出。 game 在方法中被當做 DiceGame 型別而不是 SnakeAndLadders 型別，所以方法中只能存取 DiceGame 協定中的成員。

DiceGameTracker 的執行情況，如下所示：

```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Started a new game of Snakes and Ladders
// The game is using a 6-sided dice
// Rolled a 3
// Rolled a 5
// Rolled a 4
// Rolled a 5
// The game lasted for 4 turns
```

在擴展中添加協定成員

即便無法修改源程式碼，依然可以通過 擴展(Extension) 來擴充已存在型別(譯者注： 類別，結構，列舉等)。 擴展 可以為已存在的型別添加 屬性， 方法， 下標， 協定 等成員。詳情請在擴展章節中查看。

> 注意： 通過 擴展 為已存在的型別 遵循 協定時，該型別的所有實例也會隨之添加協定中的方法

TextRepresentable 協定含有一個 asText， 如下所示：

```
protocol TextRepresentable {
    func asText() -> String
}
```

通過 `擴展` 為上一節中提到的 `Dice` 類別遵循 `TextRepresentable` 協定

```
extension Dice: TextRepresentable {
    cun asText() -> String {
        return "A \(sides)-sided dice"
    }
}
```

從現在起， `Dice` 型別的實例可被當作 `TextRepresentable` 型別：

```
let d12 = Dice(sides: 12,generator: LinearCongruentialGenerator())
println(d12.asText())
// 輸出 "A 12-sided dice"
```

`SnakesAndLadders` 類別也可以通過 `擴展` 的方式來遵循協定：

```
extension SnakeAndLadders: TextRepresentable {
    func asText() -> String {
        return "A game of Snakes and Ladders with \(finalSquare) squares"
    }
}
println(game.asText())
// 輸出 "A game of Snakes and Ladders with 25 squares"
```

通過擴展補充協定宣告

當一個型別已經實作了協定中的所有要求，卻沒有宣告時，可以通過 `擴展` 來補充協定宣告：

```
struct Hamster {
    var name: String
    func asText() -> String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentabl {}
```

從現在起， `Hamster` 的實例可以作為 `TextRepresentable` 型別使用

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentabl = simonTheHamester
println(somethingTextRepresentable.asText())
// 輸出 "A hamster named Simon"
```

> 注意： 即時滿足了協定的所有要求，型別也不會自動轉變，因此你必須為它做出明顯的協定宣告

集合中的協定型別

協定型別可以被集合使用，表示集合中的元素均為協定型別：

```
let things: TextRepresentable[] = [game,d12,simoTheHamster]
```

如下所示，`things`陣列可以被直接遍歷，並呼叫其中元素的`asText()`函式：

```
for thing in things {
    println(thing.asText())
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

`thing`被當做是`TextRepresentable`型別而不是`Dice`，`DiceGame`，`Hamster`等型別。因此能且僅能呼叫`asText`方法

協定的繼承

協定能夠繼承一到多個其他協定。語法與類別的繼承相似，多個協定間用逗號`,`分隔

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // 協定定義
}
```

如下所示，`PrettyTextRepresentable`協定繼承了`TextRepresentable`協定

```
protocol PrettyTextRepresentable: TextRepresentable {
    func asPrettyText() -> String
}
```

`遵循``PrettyTextRepresentable`協定的同時，也需要`遵循`TextRepresentable`協定。

如下所示，用`擴展`為`SnakesAndLadders`遵循`PrettyTextRepresentable`協定：

```
extension SnakesAndLadders: PrettyTextRepresentable {
    func asPrettyText() -> String {
        var output = asText() + ":\n"
        for index in 1...finalSquare {
            switch board[index] {
                case let ladder where ladder > 0:
                output += "▲ "
            case let snake where snake < 0:
                output += "▼ "
            default:
                output += "○ "
            }
        }
        return output
    }
}
```

在`for in`中迭代出了`board`陣列中的每一個元素：

- 當從陣列中迭代出的元素的值大於0時，用`▲`表示
- 當從陣列中迭代出的元素的值小於0時，用`▼`表示
- 當從陣列中迭代出的元素的值等於0時，用`○`表示

任意`SankesAndLadders`的實例都可以使用`asPrettyText()`方法。

```
println(game.asPrettyText())
// A game of Snakes and Ladders with 25 squares:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

協定合成

一個協定可由多個協定采用 `protocol<SomeProtocol, AnotherProtocol>` 這樣的格式進行組合，稱為 `協定合成(protocol composition)` 。

舉個範例：

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(celebrator: protocol<Named, Aged>) {
    println("Happy birthday \(celebrator.name) - you're \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(birthdayPerson)
// 輸出 "Happy birthday Malcolm - you're 21!
```

`Named` 協定包含 `String` 型別的 `name` 屬性； `Aged` 協定包含 `Int` 型別的 `age` 屬性。 `Person` 結構 `遵循` 了這兩個協定。

`wishHappyBirthday` 函式的形參 `celebrator` 的型別為 `protocol<Named,Aged>` 。可以傳入任意 `遵循` 這兩個協定的型別的實例

> 注意： `協定合成` 並不會生成一個新協定型別，而是將多個協定合成為一個臨時的協定，超出範圍後立即失效。

檢驗協定的一致性

使用 `is` 檢驗協定一致性，使用 `as` 將協定型別 `向下轉換(downcast)` 為的其他協定型別。檢驗與轉換的語法和之前相同(詳情查看型別檢查)：

- `is` 運算子用來檢查實例是否 `遵循` 了某個 `協定` 。
- `as?` 回傳一個可選值，當實例 `遵循` 協定時，回傳該協定型別；否則回傳 `nil`
- `as` 用以強制向下轉換型。

```
@objc protocol HasArea {
    var area: Double { get }
}
```

> 注意： `@objc` 用來表示協定是可選的，也可以用來表示暴露給 `Objective-C` 的程式碼，此外， `@objc` 型協定只對 `類別` 有效，因此只能在 `類別` 中檢查協定的一致性。詳情查看*Using Siwft with Cocoa and Objectivei-c*。

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area:=radius }
    init(radius: Double) { self.radius = radius }
}
```

```
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

`Circle` 和 `Country` 都遵循了 `HasArea` 協定，前者把 `area` 寫為計算型屬性（computed property），後者則把 `area` 寫為儲存型屬性（stored property）。

如下所示， `Animal` 類別沒有實作任何協定

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

`Circle,Country,Animal` 並沒有一個相同的基類別，所以采用 `AnyObject` 型別的陣列來裝載在它們的實例，如下所示：

```
let objects: AnyObject[] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]
```

如下所示，在迭代時檢查 `object` 陣列的元素是否 `遵循` 了 `HasArea` 協定：

```
for object in objects {
    if let objectWithArea = object as? HasArea {
        println("Area is \(objectWithArea.area)")
    } else {
        println("Something that doesn't have an area")
    }
}
// Area is 12.5663708
// Area is 243610.0
// Something that doesn't have an area
```

當陣列中的元素遵循 `HasArea` 協定時，通過 `as?` 運算子將其 `可選綁定(optional binding)` 到 `objectWithArea` 常數上。

`objects` 陣列中元素的型別並不會因為 `向下轉型` 而改變，當它們被賦值給 `objectWithArea` 時只被視為 `HasArea` 型別，因此只有 `area` 屬性能夠被存取。

可選協定要求

可選協定含有可選成員，其 `遵循者` 可以選擇是否實作這些成員。在協定中使用 `@optional` 關鍵字作為前綴來定義可選成員。

可選協定在呼叫時使用 `可選鏈` ，詳細內容在可選鏈章節中查看。

像 `someOptionalMethod?(someArgument)` 一樣，你可以在可選方法名稱後加上 `?` 來檢查該方法是否被實作。 `可選方法` 和 `可選屬性` 都會回傳一個 `可選值(optional value)` ，當其不可存取時， `?` 之後語句不會執行，並回傳 `nil` 。

> 注意： 可選協定只能在含有 `@objc` 前綴的協定中生效。且 `@objc` 的協定只能被 `類別` 遵循。

`Counter` 類別使用 `CounterDataSource` 型別的外部資料源來提供 `增量值(increment amount)` ，如下所示：

```
@objc protocol CounterDataSource {
    @optional func incrementForCount(count: Int) -> Int
```

```
    @optional var fixedIncrement: Int { get }
}
```

CounterDataSource 含有 incrementForCount 的 可選方法 和 fiexdIncrement 的 可選屬性 。

> 注意： CounterDataSource 中的屬性和方法都是可選的，因此可以在類別中宣告但不實作這些成員，儘管技術上允許這樣做，不過最好不要這樣寫。

Counter 類別含有 CounterDataSource? 型別的可選屬性 dataSource ，如下所示：

```
@objc class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.incrementForCount?(count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement? {
            count += amount
        }
    }
}
```

count 屬性用於儲存當前的值， increment 方法用來為 count 賦值。

increment 方法通過 可選鏈 ，嘗試從兩種 可選成員 中獲取 count 。

1. 由於 dataSource 可能為 nil ，因此在 dataSource 後邊加上了 ? 標記來表明只在 dataSource 非空時才去呼叫 incrementForCount`方法。
2. 即使 dataSource 存在，但是也無法保證其是否實作了 incrementForCount 方法，因此在 incrementForCount 方法後邊也加有 ? 標記。

在呼叫 incrementForCount 方法後， Int 型 可選值 通過 可選綁定(optional binding) 自動拆包並賦值給常數 amount 。

當 incrementForCount 不能被呼叫時，嘗試使用 可選屬性``fixedIncrement 來代替。

ThreeSource 實作了 CounterDataSource 協定，如下所示：

```
class ThreeSource: CounterDataSource {
    let fixedIncrement = 3
}
```

使用 ThreeSource 作為資料源開實例化一個 Counter ：

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    println(counter.count)
}
// 3
// 6
// 9
// 12
```

TowardsZeroSource 實作了 CounterDataSource 協定中的 incrementForCount 方法，如下所示：

```
class TowardsZeroSource: CounterDataSource {
```

```
func incrementForCount(count: Int) -> Int {
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```

下邊是執行的程式碼：

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    println(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

> 翻譯：takalard 校對：lifedim

泛型

本頁包含內容：

泛型程式碼可以讓你寫出根據自我需求定義、適用於任何型別的，靈活且可重用的函式和型別。它的可以讓你避免重複的程式碼，用一種清晰和抽像的方式來表達程式碼的意圖。

泛型是 Swift 強大特征中的其中一個，許多 Swift 標準函式庫是通過泛型程式碼構建出來的。事實上，泛型的使用貫穿了整本語言手冊，只是你沒有發現而已。例如，Swift 的陣列和字典型別都是泛型集。你可以創建一個 `Int` 陣列，也可創建一個 `String` 陣列，或者甚至於可以是任何其他 Swift 的型別資料陣列。同樣的，你也可以創建儲存任何指定型別的字典（dictionary），而且這些型別可以是沒有限制的。

泛型所解決的問題

這裡是一個標準的，非泛型函式 `swapTwoInts`,用來交換兩個Int值：

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

這個函式使用寫入讀出（in-out）參數來交換 `a` 和 `b` 的值，請參考寫入讀出參數。

`swapTwoInts` 函式可以交換 `b` 的原始值到 `a`，也可以交換a的原始值到 `b`，你可以呼叫這個函式交換兩個 `Int` 變數值：

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

`swapTwoInts` 函式是非常有用的，但是它只能交換 `Int` 值，如果你想要交換兩個 `String` 或者 `Double`，就不得不寫更多的函式，如 `swapTwoStrings` 和 `swapTwoDoublesfunctions`，如同如下所示：

```
func swapTwoStrings(inout a: String, inout b: String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(inout a: Double, inout b: Double) {
```

```
    let temporaryA = a
    a = b
    b = temporaryA
}
```

你可能注意到 `swapTwoInts`、 `swapTwoStrings` 和 `swapTwoDoubles` 函式功能都是相同的，唯一不同之處就在於傳入的變數型別不同，分別是 `Int`、 `String` 和 `Double`。

但實際應用中通常需要一個用處更強大並且盡可能的考慮到更多的靈活性單個函式，可以用來交換兩個任何型別值，很幸運的是，泛型程式碼幫你解決了這種問題。（一個這種泛型函式後面已經定義好了。）

> 注意：
> 在所有三個函式中， `a` 和 `b` 的型別是一樣的。如果 `a` 和 `b` 不是相同的型別，那它們倆就不能互換值。Swift 是型別安全的語言，所以它不允許一個 `String` 型別的變數和一個 `Double` 型別的變數互相交換值。如果一定要做，Swift 將報編譯錯誤。

泛型函式

`泛型函式` 可以工作於任何型別，這裡是一個上面 `swapTwoInts` 函式的泛型版本，用於交換兩個值：

```
func swapTwoValues<T>(inout a: T, inout b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

`swapTwoValues` 函式主體和 `swapTwoInts` 函式是一樣的，它只在第一行稍微有那麼一點點不同於 `swapTwoInts`，如下所示：

```
func swapTwoInts(inout a: Int, inout b: Int)
func swapTwoValues<T>(inout a: T, inout b: T)
```

這個函式的泛型版本使用了占位型別名字（通常此情況下用字母 `T` 來表示）來代替實際型別名（如 `In`、 `String` 或 `Doubl`）。占位型別名沒有提示 `T` 必須是什麼型別，但是它提示了 `a` 和 `b` 必須是同一型別 `T`，而不管 `T` 表示什麼型別。只有 `swapTwoValues` 函式在每次呼叫時所傳入的實際型別才能決定 `T` 所代表的型別。

另外一個不同之處在於這個泛型函式名後面跟著的展位型別名字（T）是用角括號括起來的（）。這個角括號告訴 Swift 那個 `T` 是 `swapTwoValues` 函式所定義的一個型別。因為 `T` 是一個占位命名型別，Swift 不會去查找命名為T的實際型別。

`swapTwoValues` 函式除了要求傳入的兩個任何型別值是同一型別外，也可以作為 `swapTwoInts` 函式被呼叫。每次 `swapTwoValues` 被呼叫，T所代表的型別值都會傳給函式。

在下面的兩個範例中, `T` 分別代表 `Int` 和 `String`：

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3
```

```
var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

> 注意
> 上面定義的函式 `swapTwoValues` 是受 `swap` 函式啟發而實作的。 `swap` 函式存在於 Swift 標準函式庫，並可以在其它類別中任意使用。如果你在自己程式碼中需要類似 `swapTwoValues` 函式的功能，你可以使用已存在的交換函式 `swap` 函式。

型別參數

在上面的 `swapTwoValues` 範例中，占位型別 `T` 是一種型別參數的示例。型別參數指定並命名為一個占位型別，並且緊隨在函式名後面，使用一對角括號括起來（如）。

一旦一個型別參數被指定，那麼其可以被使用來定義一個函式的參數型別（如 `swapTwoValues` 函式中的參數 `a` 和 `b` ），或作為一個函式回傳型別，或用作函式主體中的注釋型別。在這種情況下，被型別參數所代表的占位型別不管函式任何時候被呼叫，都會被實際型別所替換（在上面 `swapTwoValues` 範例中，當函式第一次被呼叫時， `T` 被 `Int` 替換，第二次呼叫時，被 `String` 替換。）。

你可支援多個型別參數，命名在角括號中，用逗號分開。

命名型別參數

在簡單的情況下，泛型函式或泛型型別需要指定一個占位型別（如上面的 `swapTwoValues` 泛型函式，或一個儲存單一型別的泛型集，如陣列），通常用一單個字母 `T` 來命名型別參數。不過，你可以使用任何有效的識別符號來作為型別參數名。

如果你使用多個參數定義更複雜的泛型函式或泛型型別，那麼使用更多的描述型別參數是非常有用的。例如，Swift 字典（Dictionary）型別有兩個型別參數，一個是鍵，另外一個是值。如果你自己寫字典，你或許會定義這兩個型別參數為 `KeyType` 和 `ValueType` ，用來記住它們在你的泛型程式碼中的作用。

> 注意
> 請始終使用大寫字母開頭的駝峰式命名法（例如 `T` 和 `KeyType` ）來給型別參數命名，以表明它們是型別的占位符，而非型別值。

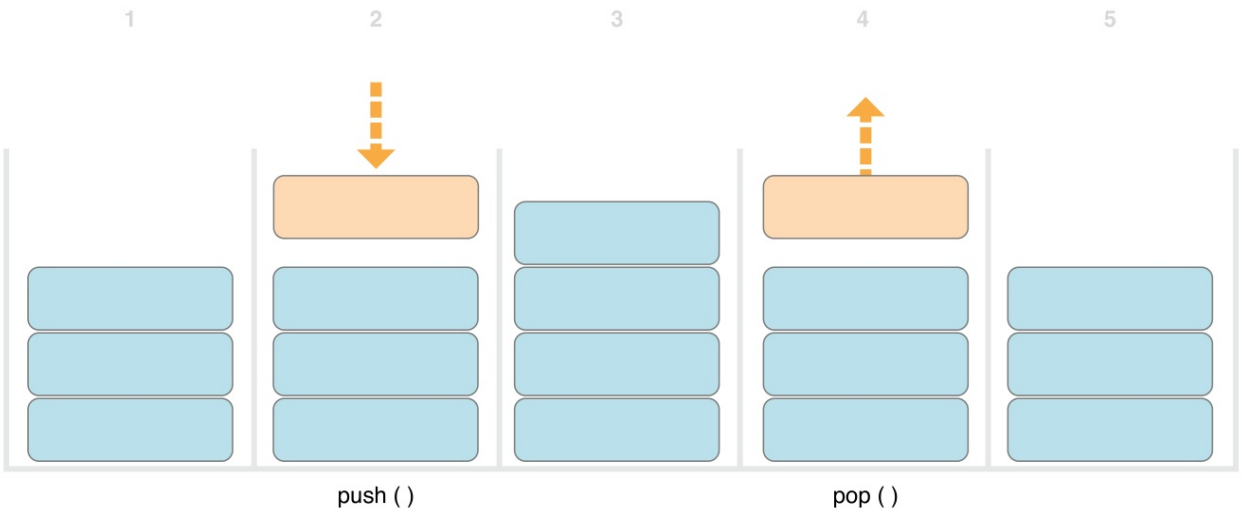
泛型型別

通常在泛型函式中，Swift 允許你定義你自己的泛型型別。這些自定義類別、結構和列舉作用於任何型別，如同 `Array` 和 `Dictionary` 的用法。

這部分向你展示如何寫一個泛型集型別-- `Stack` （棧）。一個棧是一系列值域的集合，和 `Array` （陣列）類似，但其是一個比 Swift 的 `Array` 型別更多限制的集合。一個陣列可以允許其裡面任何位置的插入/刪除操作，而棧，只允許在集合的末端添加新的項（如同*push*一個新值進棧）。同樣的一個棧也只能從末端移除項（如同*pop*一個值出棧）。

> 注意
> 棧的概念已被 `UINavigationController` 類別使用來模擬試圖控制器的導航結構。你通過呼叫 `UINavigationController` 的 `pushViewController:animated:` 方法來為導航棧添加（add）新的試圖控制器；而通過 `popViewControllerAnimated:` 的方法來從導航棧中移除（pop）某個試圖控制器。每當你需要一個嚴格的 `後進先出` 方式來管理集合，堆棧都是最實用的模型。

下圖展示了一個棧的壓棧(push)/出棧(pop)的行為：



1. 現在有三個值在棧中；
2. 第四個值「pushed」到棧的頂部；
3. 現在有四個值在棧中，最近的那個在頂部；
4. 棧中最頂部的那個項被移除，或稱之為「popped」；
5. 移除掉一個值後，現在棧又重新只有三個值。

這裡展示了如何寫一個非泛型版本的棧，`Int`值型的棧：

```
struct IntStack {
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

這個結構在棧中使用一個`Array`性質的`items`儲存值。`Stack`提供兩個方法：`push`和`pop`，從棧中壓進一個值和移除一個值。這些方法標記為可變的，因為它們需要修改（或轉換）結構的`items`陣列。

上面所展現的`IntStack`型別只能用於`Int`值，不過，其對於定義一個泛型`Stack`類別（可以處理任何型別值的棧）是非常有用的。

這裡是一個相同程式碼的泛型版本：

```
struct Stack<T> {
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

注意到`Stack`的泛型版本基本上和非泛型版本相同，但是泛型版本的占位型別參數為T代替了實際`Int`型別。這種型別參數包含在一對角括號裡（`<T>`），緊隨在結構名字後面。

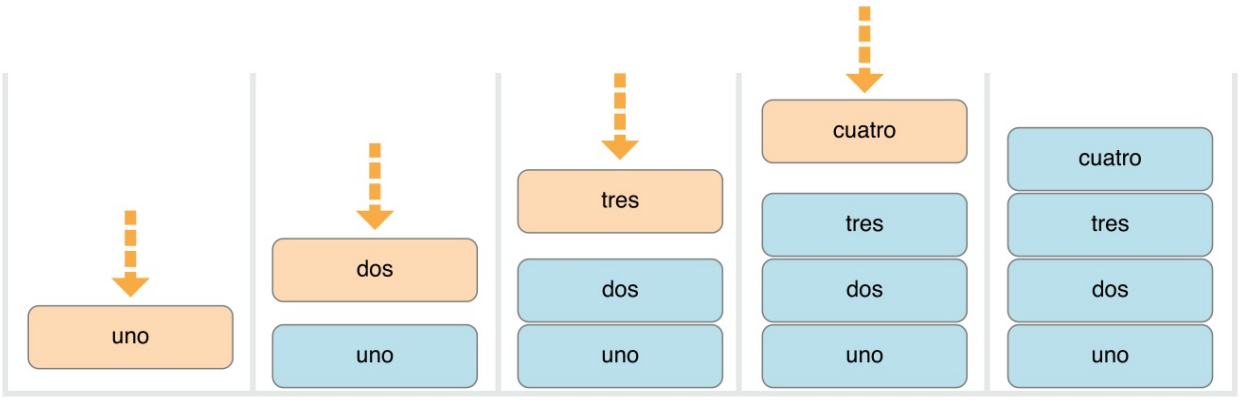
`T`定義了一個名為「某種型別T」的節點提供給後來用。這種將來型別可以在結構的定義裡任何地方表示為「T」。在這種情況下，`T`在如下三個地方被用作節點：

- 創建一個名為 `items` 的屬性，使用空的T型別值陣列對其進行初始化；
- 指定一個包含一個參數名為 `item` 的 `push` 方法，該參數必須是T型別；
- 指定一個 `pop` 方法的回傳值，該回傳值將是一個T型別值。

當創建一個新單例並初始化時， 通過用一對緊隨在型別名後的角括號裡寫出實際指定棧用到型別，創建一個 `Stack` 實例，同創建 `Array` 和 `Dictionary` 一樣：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// 現在棧已經有4個string了
```

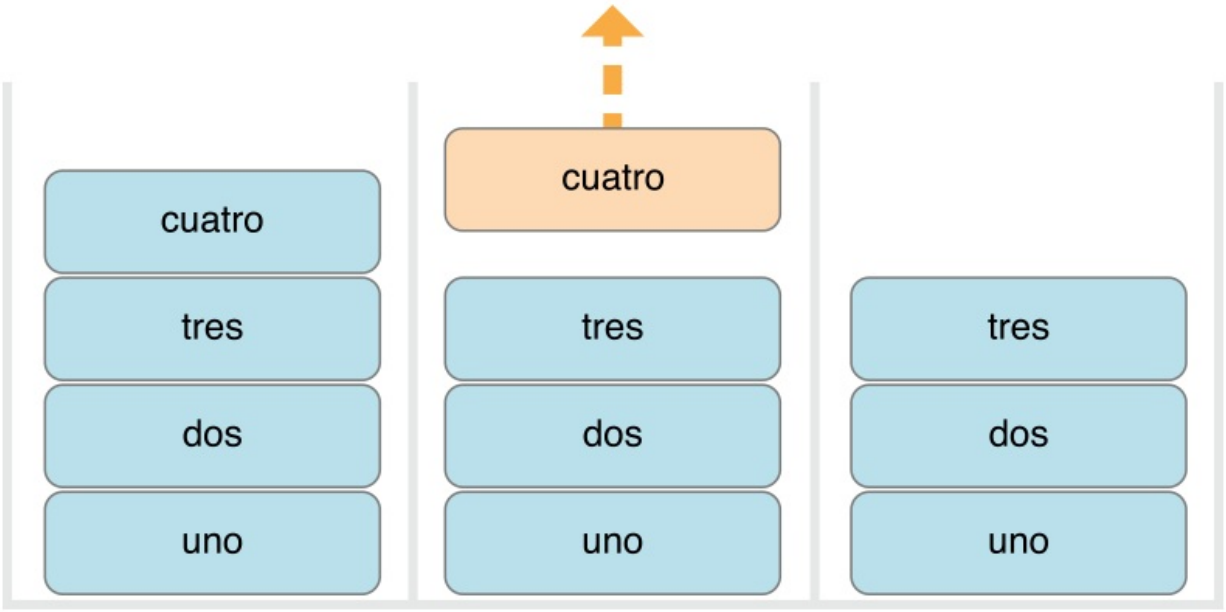
下圖將展示 `stackOfStrings` 如何 `push` 這四個值進棧的過程：



從棧中 `pop` 並移除值"cuatro"：

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

下圖展示了如何從棧中pop一個值的過程：



由於 `Stack` 是泛型型別，所以在 Swift 中其可以用來創建任何有效型別的棧，這種方式如同 `Array` 和 `Dictionary`。

型別約束

`swapTwoValues` 函式和 `Stack` 型別可以作用於任何型別，不過，有的時候對使用在泛型函式和泛型型別上的型別強制約束為某種特定型別是非常有用的。型別約束指定了一個必須繼承自指定類別的型別參數，或者遵循一個特定的協定或協定構成。

例如，Swift 的 `Dictionary` 型別對作用於其鍵的型別做了些限制。在字典的描述中，字典的鍵型別必須是可雜湊，也就是說，必須有一種方法可以使其是唯一的表示。 `Dictionary` 之所以需要其鍵是可雜湊是為了以便於其檢查其是否包含某個特定鍵的值。如無此需求， `Dictionary` 即不會告訴是否插入或者替換了某個特定鍵的值，也不能查找到已經儲存在字典裡面的給定鍵值。

這個需求強制加上一個型別約束作用於 `Dictionary` 的鍵上，當然其鍵型別必須遵循 `Hashable` 協定（Swift 標準函式庫中定義的一個特定協定）。所有的 Swift 基本型別（如 `String`， `Int`， `Double` 和 `Bool`）預設都是可雜湊。

當你創建自定義泛型型別時，你可以定義你自己的型別約束，當然，這些約束要支援泛型編程的強力特征中的多數。抽像概念如 `可雜湊` 具有的型別特征是根據它們概念特征來界定的，而不是它們的直接型別特征。

型別約束語法

你可以寫一個在一個型別參數名後面的型別約束，通過冒號分割，來作為型別參數鏈的一部分。這種作用於泛型函式的型別約束的基礎語法如下所示（和泛型型別的語法相同）：

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

上面這個假定函式有兩個型別參數。第一個型別參數 `T`，有一個需要 `T` 必須是 `SomeClass` 子類別的型別約束；第二個型別參數 `U`，有一個需要 `U` 必須遵循 `SomeProtocol` 協定的型別約束。

型別約束行為

這裡有個名為 `findStringIndex` 的非泛型函式，該函式功能是去查找包含一給定 `String` 值的陣列。若查找到匹配的字串， `findStringIndex` 函式回傳該字串在陣列中的索引值（ `Int` ），反之則回傳 `nil` ：

```
func findStringIndex(array: String[], valueToFind: String) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

`findStringIndex` 函式可以作用於查找一字串陣列中的某個字串:

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findStringIndex(strings, "llama") {
    println("The index of llama is \(foundIndex)")
}
// 輸出 "The index of llama is 2"
```

如果只是針對字串而言查找在陣列中的某個值的索引，用處不是很大，不過，你可以寫出相同功能的泛型函式 `findIndex`，用某個型別 `T` 值替換掉提到的字串。

這裡展示如何寫一個你或許期望的 `findStringIndex` 的泛型版本 `findIndex`。請注意這個函式仍然回傳 `Int`，是不是有點迷惑呢，而不是泛型型別?那是因為函式回傳的是一個可選的索引數，而不是從陣列中得到的一個可選值。需要提醒的是，這個函式不會編譯，原因在範例後面會說明：

```
func findIndex<T>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

上面所寫的函式不會編譯。這個問題的位置在等式的檢查上，`「if value == valueToFind」`。不是所有的 Swift 中的型別都可以用等式符（==）進行比較。例如，如果你創建一個你自己的類別或結構來表示一個複雜的資料模型，那麼 Swift 沒法猜到對於這個類別或結構而言「等於」的意思。正因如此，這部分程式碼不能可能保證工作於每個可能的型別 `T`，當你試圖編譯這部分程式碼時估計會出現相應的錯誤。

不過，所有的這些並不會讓我們無從下手。Swift 標準函式庫中定義了一個 `Equatable` 協定，該協定要求任何遵循的型別實作等式符（==）和不等符（!=）對任何兩個該型別進行比較。所有的 Swift 標準型別自動支援 `Equatable` 協定。

任何 `Equatable` 型別都可以安全的使用在 `findIndex` 函式中，因為其保證支援等式操作。為了說明這個事實，當你定義一個函式時，你可以寫一個 `Equatable` 型別約束作為型別參數定義的一部分：

```
func findIndex<T: Equatable>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

`findIndex` 中這個單個型別參數寫做：`T: Equatable`，也就意味著「任何T型別都遵循 `Equatable` 協定」。

`findIndex` 函式現在則可以成功的編譯過，並且作用於任何遵循 `Equatable` 的型別，如 `Double` 或 `String`：

```
let doubleIndex = findIndex([3.14159, 0.1, 0.25], 9.3)
// doubleIndex is an optional Int with no value, because 9.3 is not in the array
let stringIndex = findIndex(["Mike", "Malcolm", "Andrea"], "Andrea")
// stringIndex is an optional Int containing a value of 2
```

關聯型別

當定義一個協定時，有的時候宣告一個或多個關聯型別作為協定定義的一部分是非常有用的。一個關聯型別給定作用於協定部分的型別一個節點名（或別名）。作用於關聯型別上實際型別是不需要指定的，直到該協定接受。關聯型別被指定為 `typealias` 關鍵字。

關聯型別行為

這裡是一個 `Container` 協定的範例，定義了一個ItemType關聯型別：

```
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
```

```
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
```

`Container` 協定定義了三個任何容器必須支援的相容要求：

- 必須可能通過 `append` 方法添加一個新item到容器裡；
- 必須可能通過使用 `count` 屬性獲取容器裡items的數量，並回傳一個 `Int` 值；
- 必須可能通過容器的 `Int` 索引值下標可以檢索到每一個item。

這個協定沒有指定容器裡item是如何儲存的或何種型別是允許的。這個協定只指定三個任何遵循 `Container` 型別所必須支援的功能點。一個遵循的型別也可以提供其他額外的功能，只要滿足這三個條件。

任何遵循 `Container` 協定的型別必須指定儲存在其裡面的值型別，必須保證只有正確型別的items可以加進容器裡，必須明確可以通過其下標回傳item型別。

為了定義這三個條件， `Container` 協定需要一個方法指定容器裡的元素將會保留，而不需要知道特定容器的型別。 `Container` 協定需要指定任何通過 `append` 方法添加到容器裡的值和容器裡元素是相同型別，並且通過容器下標回傳的容器元素型別的值的型別是相同型別。

為了達到此目的， `Container` 協定宣告了一個ItemType的關聯型別，寫作 `typealias ItemType` 。The protocol does not define what ItemType is an alias for—that information is left for any conforming type to provide（這個協定不會定義 `ItemType` 是遵循型別所提供的何種資訊的別名）。儘管如此， `ItemType` 別名支援一種方法識別在一個容器裡的items型別，以及定義一種使用在 `append` 方法和下標中的型別，以便保證任何期望的 `Container` 的行為是強制性的。

這裡是一個早前IntStack型別的非泛型版本，適用於遵循Container協定：

```
struct IntStack: Container {
    // original IntStack implementation
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias ItemType = Int
    mutating func append(item: Int) {
        self.push(item)
    }
    var count: Int {
    return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

`IntStack` 型別實作了 `Container` 協定的所有三個要求，在 `IntStack` 型別的每個包含部分的功能都滿足這些要求。

此外， `IntStack` 指定了 `Container` 的實作，適用的ItemType被用作 `Int` 型別。對於這個 `Container` 協定實作而言，定義 `typealias ItemType = Int` ，將抽像的 `ItemType` 型別轉換為具體的 `Int` 型別。

感謝Swift型別參考，你不用在 `IntStack` 定義部分宣告一個具體的 `Int` 的 `ItemType` 。由於 `IntStack` 遵循 `Container` 協定的所有要求，只要通過簡單的查找 `append` 方法的item參數型別和下標回傳的型別，Swift就可以推斷出合適的 `ItemType` 來使用。確實，如果上面的程式碼中你刪除了 `typealias ItemType = Int` 這一行，一切仍舊可以工作，因為它清楚的知道ItemType使用的是何種型別。

你也可以生成遵循 `Container` 協定的泛型 `Stack` 型別：

```
struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }
    var count: Int {
    return items.count
    }
    subscript(i: Int) -> T {
        return items[i]
    }
}
```

這個時候，占位型別參數 `T` 被用作 `append` 方法的item參數和下標的回傳型別。Swift 因此可以推斷出被用作這個特定容器的 `ItemType` 的 `T` 的合適型別。

擴展一個存在的型別為一指定關聯型別

在使用擴展來添加協定相容性中有描述擴展一個存在的型別添加遵循一個協定。這個型別包含一個關聯型別的協定。

Swift的 `Array` 已經提供 `append` 方法，一個 `count` 屬性和通過下標來查找一個自己的元素。這三個功能都達到 `Container` 協定的要求。也就意味著你可以擴展 `Array` 去遵循 `Container` 協定，只要通過簡單宣告 `Array` 適用於該協定而已。如何實踐這樣一個空擴展，在使用擴展來宣告協定的采納中有描述這樣一個實作一個空擴展的行為：

```
extension Array: Container {}
```

如同上面的泛型 `Stack` 型別一樣， `Array的append` 方法和下標保證 `Swift` 可以推斷出 `ItemType` 所使用的適用的型別。定義了這個擴展後，你可以將任何 `Array` 當作 `Container` 來使用。

Where 語句

型別約束中描述的型別約束確保你定義關於型別參數的需求和一泛型函式或型別有關聯。

對於關聯型別的定義需求也是非常有用的。你可以通過這樣去定義*where*語句作為一個型別參數隊列的一部分。一個 `where` 語句使你能夠要求一個關聯型別遵循一個特定的協定，以及（或）那個特定的型別參數和關聯型別可以是相同的。你可寫一個 `where` 語句，通過緊隨放置 `where` 關鍵字在型別參數隊列後面，其後跟著一個或者多個針對關聯型別的約束，以及（或）一個或多個型別和關聯型別的等於關系。

下面的列子定義了一個名為 `allItemsMatch` 的泛型函式，用來檢查是否兩個 `Container` 單例包含具有相同順序的相同元素。如果匹配到所有的元素，那麼回傳一個為 `true` 的 `Boolean` 值，反之，則相反。

這兩個容器可以被檢查出是否是相同型別的容器（雖然它們可以是），但它們確實擁有相同型別的元素。這個需求通過一個型別約束和 `where` 語句結合來表示：

```
func allItemsMatch<
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {

        // check that both containers contain the same number of items
```

```
        if someContainer.count != anotherContainer.count {
            return false
        }

        // check each pair of items to see if they are equivalent
        for i in 0..someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // all items match, so return true
        return true

}
```

這個函式用了兩個參數： `someContainer` 和 `anotherContainer` 。 `someContainer` 參數是型別 `C1` ， `anotherContainer` 參數是型別 `C2` 。 `C1` 和 `C2` 是容器的兩個占位型別參數，決定了這個函式何時被呼叫。

這個函式的型別參數列緊隨在兩個型別參數需求的後面：

- `C1` 必須遵循 `Container` 協定 (寫作 `C1: Container`)。
- `C2` 必須遵循 `Container` 協定 (寫作 `C2: Container`)。
- `C1` 的 `ItemType` 同樣是C2的 `ItemType` （寫作 `C1.ItemType == C2.ItemType` ）。
- `C1` 的 `ItemType` 必須遵循 `Equatable` 協定 (寫作 `C1.ItemType: Equatable`)。

第三個和第四個要求被定義為一個 `where` 語句的一部分，寫在關鍵字 `where` 後面，作為函式型別參數鏈的一部分。

這些要求意思是：

`someContainer` 是一個 `C1` 型別的容器。 `anotherContainer` 是一個 `C2` 型別的容器。 `someContainer` 和 `anotherContainer` 包含相同的元素型別。 `someContainer` 中的元素可以通過不等於操作(`!=`)來檢查它們是否彼此不同。

第三個和第四個要求結合起來的意思是 `anotherContainer` 中的元素也可以通過 `!=` 操作來檢查，因為它們在 `someContainer` 中元素確實是相同的型別。

這些要求能夠使 `allItemsMatch` 函式比較兩個容器，即便它們是不同的容器型別。

`allItemsMatch` 首先檢查兩個容器是否擁有同樣數目的items，如果它們的元素數目不同，沒有辦法進行匹配，函式就會 `false` 。

檢查完之後，函式通過 `for-in` 迴圈和半閉區間操作（..）來迭代 `someContainer` 中的所有元素。對於每個元素，函式檢查是否 `someContainer` 中的元素不等於對應的 `anotherContainer` 中的元素，如果這兩個元素不等，則這兩個容器不匹配，回傳 `false` 。

如果迴圈結束後未發現沒有任何的不匹配，那表明兩個容器匹配，函式回傳 `true` 。

這裡演示了allItemsMatch函式運算的過程：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    println("All items match.")
} else {
    println("Not all items match.")
}
// 輸出 "All items match."
```

上面的範例創建一個 `Stack` 單例來儲存 `String`，然後壓了三個字串進棧。這個範例也創建了一個 `Array` 單例，並初始化包含三個同棧裡一樣的原始字串。即便棧和陣列否是不同的型別，但它們都遵循 `Container` 協定，而且它們都包含同樣的型別值。你因此可以呼叫 `allItemsMatch` 函式，用這兩個容器作為它的參數。在上面的範例中， `allItemsMatch` 函式正確的顯示了所有的這兩個容器的 `items` 匹配。

> 翻譯：xielingwang 校對：numbbbbb

進階運算子

本頁內容包括：

除了基本運算子中所講的運算子，Swift 還有許多複雜的進階運算子，包括了 C 和 Objective-C 中的位元運算子和移位運算子。

不同於 C 中的數值計算，Swift 的數值計算預設是不可溢位的。溢位行為會被捕獲並報告為錯誤。你可以使用 Swift 準備的另一套預設允許溢位的數值運算子，如溢位加法運算子 `&+`。所有允許溢位的運算子都是以 `&` 開始的。

自定義的結構、類別和列舉，是否可以使用標準的運算子來定義操作？當然可以！在 Swift 中，你可以為你創建的所有型別製定運算子操作。

可製定的運算子並不限於那些預設的運算子，自定義客製化的中綴、前綴、後綴及賦值運算子，當然還有優先級和結合性。這些運算子的實作可以運用預設的運算子，也可以運用之前製定的運算子。

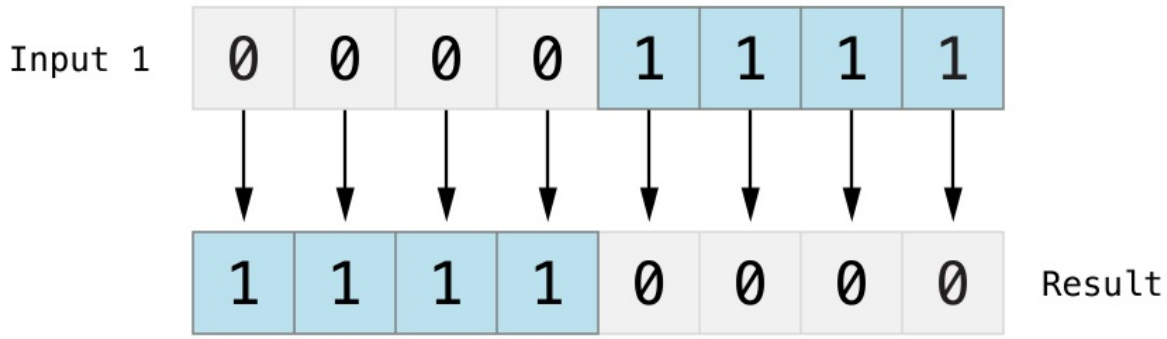
位元運算子（Bitwise Operators）

位元運算子通常在諸如圖像處理和創建設備驅動等底層開發中使用，使用它可以單獨運算元據結構中原始資料的位元。在使用一個自定義的協定進行通信的時候，運用位元運算子來對原始資料進行編碼和解碼也是非常有效的。

Swift 支援如下所有 C 的位元運算子：

位元補數運算子（Bitwise NOT Operator）

位元補數運算子 `~` 對一個運算元的每一位都取補數。



這個運算子是前綴的，運算元之前不加任何空格。

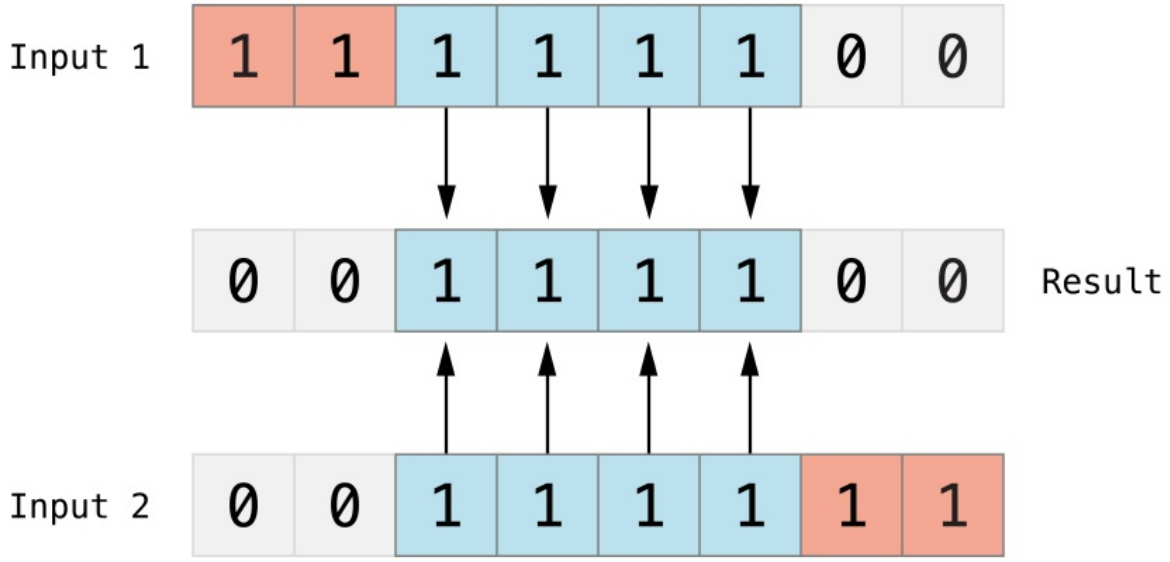
```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // 等於 0b11110000
```

`UInt8` 是 8 位元無號整數，可以儲存 0 到 255 之間的任意數。這個範例初始化一個 `UInt8` 為二進制值 `00001111` (前 4 位為 `0`，後 4 位為 `1`)，它的十進制值為 `15`。

使用位元補數運算 `~` 對 `initialBits` 操作，然後賦值給 `invertedBits` 這個新常數。這個新常數的值等於所有位都取補數的 `initialBits`，即 `1` 變成 `0`， `0` 變成 `1`，變成了 `11110000`，十進制值為 `240`。

位元 AND 運算子（Bitwise AND Operator）

位元 AND 運算子對兩個數進行操作，然後回傳一個新的數，這個數的每個位元都需要兩個輸入數的同一位元都為 1 時才為 1。

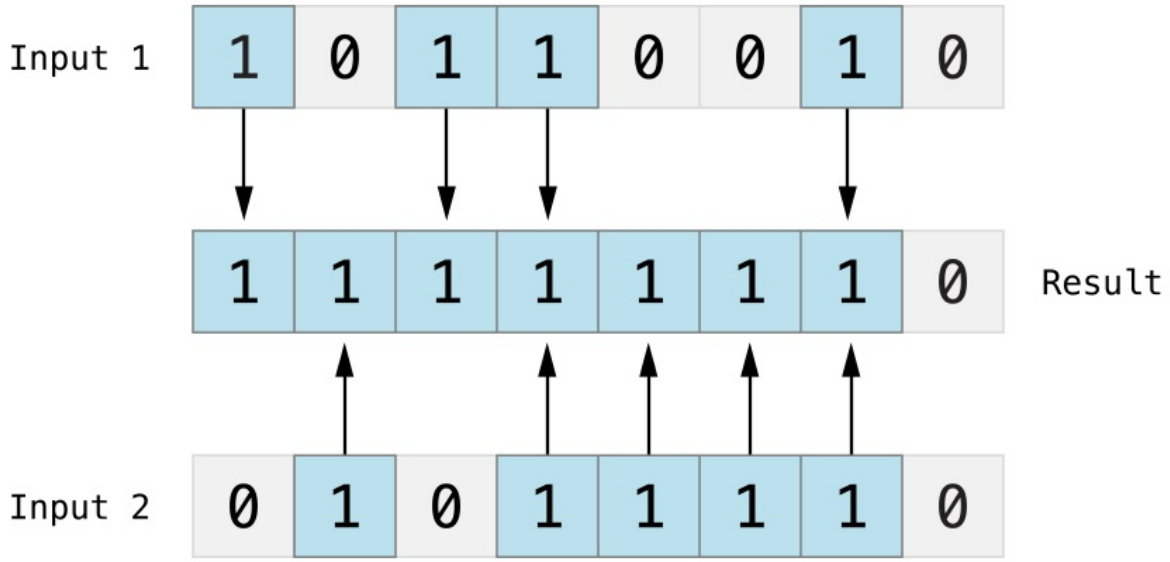


以下程式碼， `firstSixBits` 和 `lastSixBits` 中間 4 個位元都為 `1`。對它倆進行位元 AND 運算後，就得到了 `00111100`，即十進制的 `60`。

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // 等於 00111100
```

位元 OR 運算（Bitwise OR Operator）

位元 OR 運算子 `|` 比較兩個數，然後回傳一個新的數，這個數的每一位元設置 `1` 的條件是兩個輸入數的同一位元都不為 `0`（即任意一個為 `1`，或都為 `1`）。

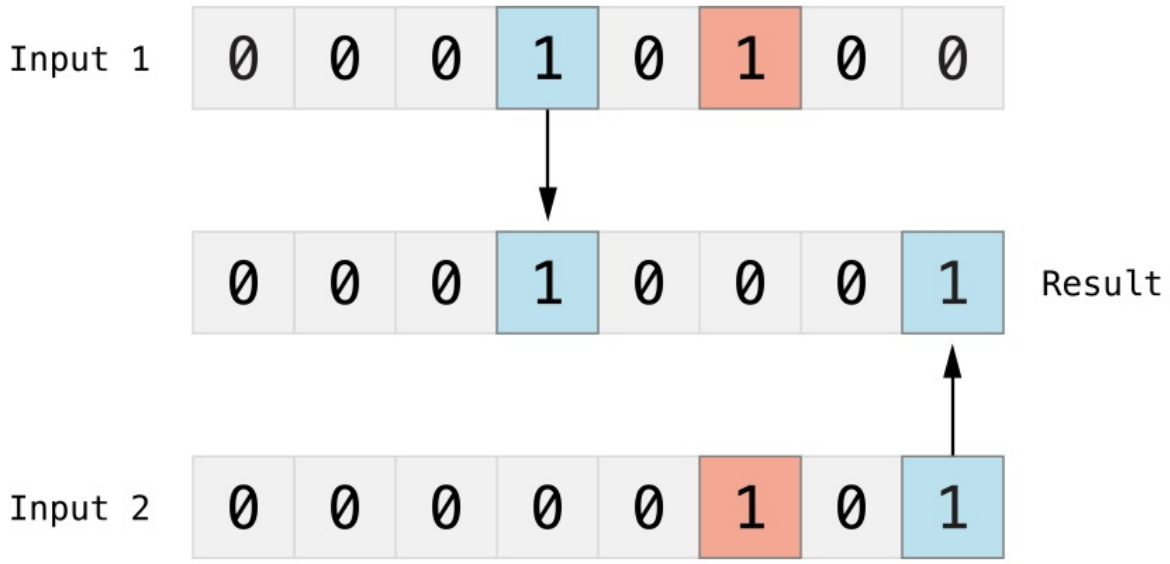


如下程式碼，`someBits` 和 `moreBits` 在不同位上有 `1`。位元 OR 執行的結果是 `11111110`，即十進制的 `254`。

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // 等於 11111110
```

位元 XOR 運算子（Bitwise XOR Operator）

位元 XOR 運算子 `^` 比較兩個數，然後回傳一個數，這個數的每個位元設為 `1` 的條件是兩個輸入數的同一位元不同，如果相同就設為 `0`。



以下程式碼，`firstBits` 和 `otherBits` 都有一個 `1` 跟另一個數不同的。所以位元 XOR 的結果是把它這些位置為 `1`，其他都置為 `0`。

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // 等於 00010001
```

移位運算子（Shift Operator）

左移運算子 `<<` 和右移運算子 `>>` 會把一個數的所有位元按以下定義的規則向左或向右移動指定位數。

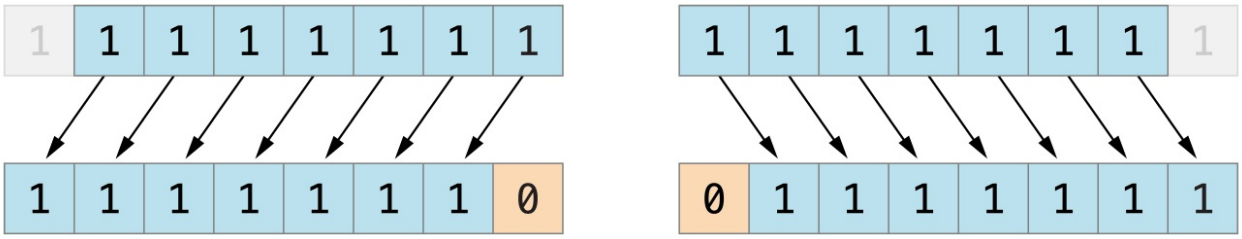
按位左移和按位右移的效果相當把一個整數乘於或除於一個因數為 `2` 的整數。向左移動一個整數的位元相當於把這個數乘於 `2`，向右移一位就是除於 `2`。

無號整數的移位運算

對無號整數的移位的效果如下：

已經存在的位元向左或向右移動指定的位數。被移出整數儲存邊界的的位數直接拋棄，移動留下的空白位用 `0` 來填充。這種方法稱為邏輯移位（logical shift）。

以下這張圖把展示了 `11111111 << 1`（`11111111` 向左移 1 位），和 `11111111 >> 1`（`11111111` 向右移 1 位）。藍色的是被移位的，灰色是被拋棄的，橙色的 `0` 是被填充進來的。




```
let shiftBits: UInt8 = 4   // 即二進制的 00000100
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
shiftBits >> 2             // 00000001
```

你可以使用移位運算進行其他資料型別的編碼和解碼。

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16    // redComponent 是 0xCC, 即 204
let greenComponent = (pink & 0x00FF00) >> 8   // greenComponent 是 0x66, 即 102
let blueComponent = pink & 0x0000FF           // blueComponent 是 0x99, 即 153
```

這個範例使用了一個 `UInt32` 型別，名為 `pink` 的常數來儲存 CSS 中粉紅色的色碼，色碼 `#CC6699` 在 Swift 用十六進制 `0xCC6699` 來表示。然後使用位元 AND 運算（&）和位元右移運算就可以從這個色碼中解析出紅（CC）、綠（66）、藍（99）三個部分。

對 `0xCC6699` 和 `0xFF0000` 進行位元 AND 運算就可以得到紅色部分。`0xFF0000` 中的 `0` 了遮罩（mask）了 `0xCC6699` 的第二和第三個位元組，這樣 `6699` 被忽略了，只留下 `0xCC0000`。

然後，按向右移動 16 位，即 `>> 16`。十六進制中每兩個字元是 8 位元，所以移動 16 位的結果是把 `0xCC0000` 變成 `0x0000CC`。這和 `0xCC` 是相等的，都是十進制的 `204`。

同樣的，綠色部分來自於 `0xCC6699` 和 `0x00FF00` 的按位操作得到 `0x006600`。然後向右移動8們，得到 `0x66`，即十進制的 `102`。

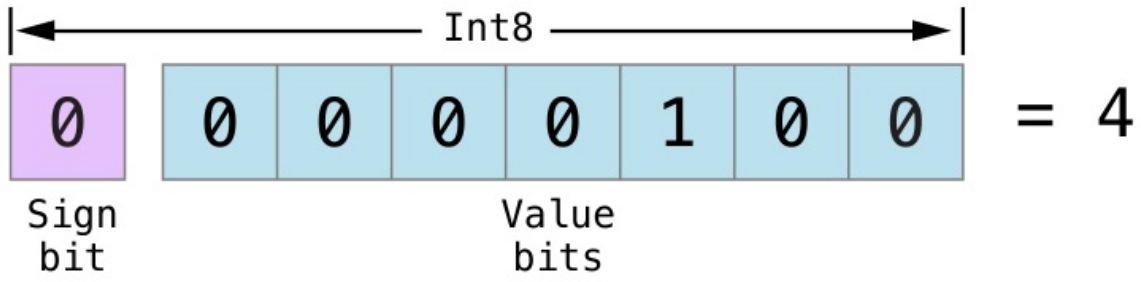
最後，藍色部分對 `0xCC6699` 和 `0x0000FF` 進行位元 AND 運算，得到 `0x000099`，無需向右移位了，所以結果就是 `0x99`，即十進制的 `153`。

有號整數的移位運算

有號整數的移位運算相對複雜得多，因為正負號也是用二進制位表示的。(這裡舉的範例雖然都是 8 位的，但它的原理是通用的。)

有號整數通過第 1 個位元（稱為符號位，sign bit）來表達這個整數是正數還是負數。`0`代表正數，`1`代表負數。

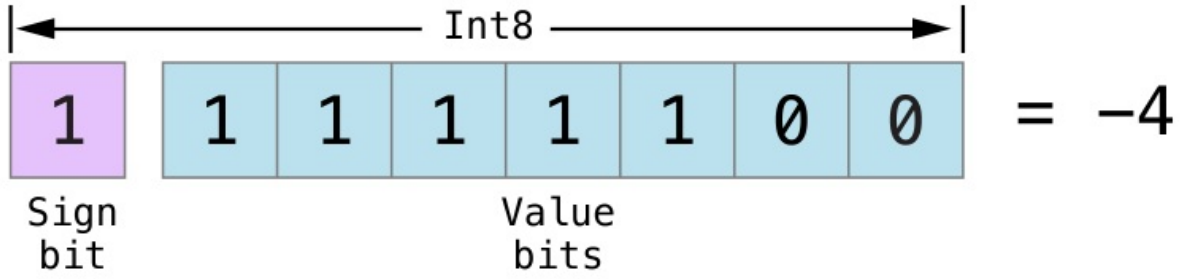
其餘的位元（稱為數值位，value bits）儲存其實值。有號正整數和無號正整數在電腦裡的儲存結果是一樣的，下來我們來看`+4`內部的二進制結構。



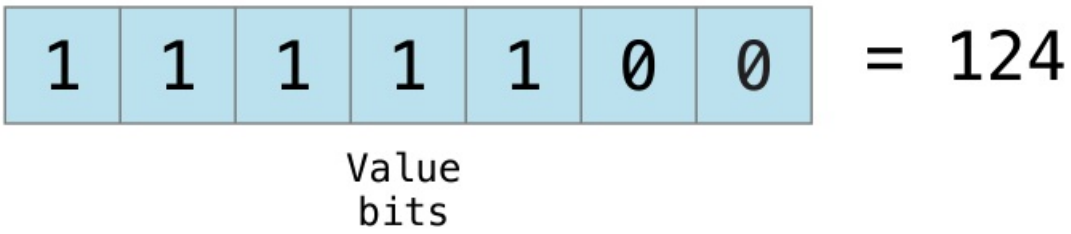
符號位為`0`，代表正數，另外 7 個位元的二進制表示的實際值就剛好是`4`。

負數跟正數不同。負數儲存的是 2 的 n 次方減去它的絕對值，n 為數值位的位數。一個 8 位元的數有 7 個數值位，所以是 2 的 7 次方，即128。

我們來看`-4`儲存的二進制結構。

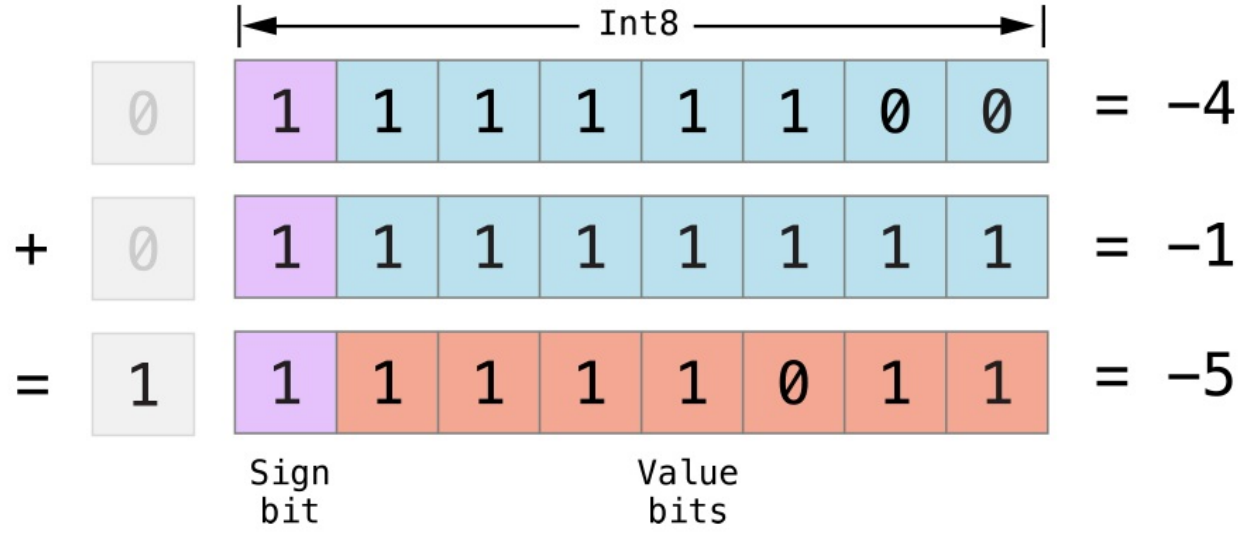


現在符號位為`1`，代表負數，7 個數值位要表達的二進制值是 124，即 128 - 4。



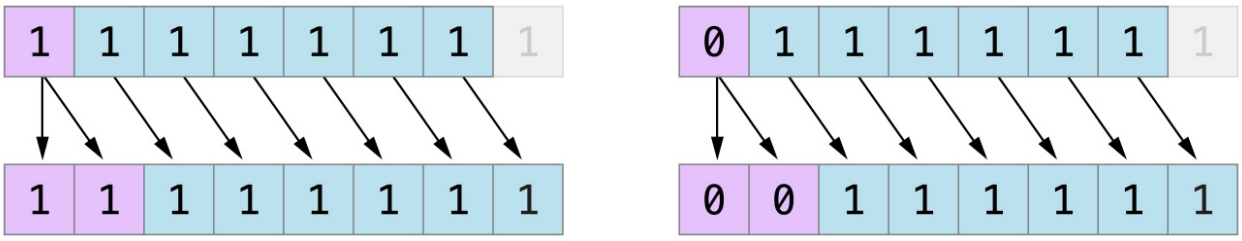
負數的編碼方式稱為二進制補數表示。這種表示方式看起來很奇怪，但它有幾個優點。

首先，只需要對全部 8 個位元（包括符號）做標準的二進制加法就可以完成`-1 + -4`的運算，忽略加法過程產生的超過 8 個位元表達的任何資訊。



第二，由於使用二進制補數表示，我們可以和正數一樣對負數進行按位左移右移的，同樣也是左移 1 位時乘於 `2`，右移 1 位時除於 `2`。要達到此目的，對有號整數的右移有一個特別的要求：

對有號整數按位右移時，使用符號位（正數為 `0`，負數為 `1`）填充空白位。



這就確保了在右移的過程中，有號整數的符號不會發生變化。這稱為算術移位（arithmetic shift）。

正因為正數和負數特殊的儲存方式，向右移位使它接近於 `0`。移位過程中保持符號會不變，負數在接近 `0` 的過程中一直是負數。

溢位運算子（Overflow Operators）

預設情況下，一個整數常數或變數被賦予一個它不能容納的大數時，Swift 會回報錯誤。這在操作過大或過小的數的時候很安全。

例如，`Int16` 整數能容納的整數範圍是 `-32768` 到 `32767`，如果給它賦上超過這個範圍的數，就會的到錯誤訊息：

```
var potentialOverflow = Int16.max
// potentialOverflow 等於 32767, 這是 Int16 能容納的最大整數
potentialOverflow += 1
// 噢，出錯了
```

對過大或過小的數值進行錯誤處理讓你的數值邊界條件更靈活。

當然，你有意在溢位時對有效位進行截斷時，可以使用溢位運算，而非錯誤處理。Swfit 為整數計算提供了 5 個 `&` 符號開頭的溢位運算子。

- 溢位加法 `&+`
- 溢位減法 `&-`

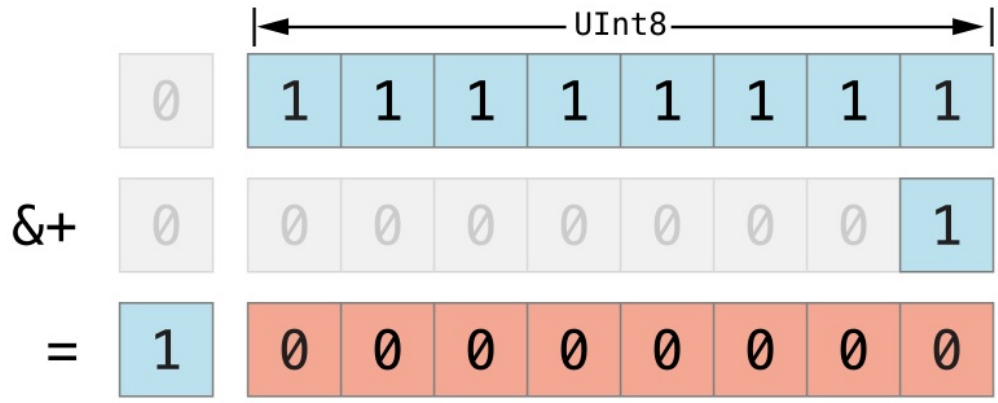
- 溢位乘法 `&*`
- 溢位除法 `&/`
- 溢位取餘 `&%`

值的上溢位

下面範例使用了溢位加法 `&+` 來解釋無號整數的上溢位

```
var willOverflow = UInt8.max
// willOverflow 等於 UInt8 的最大整數 255
willOverflow = willOverflow &+ 1
// 此時 willOverflow 等於 0
```

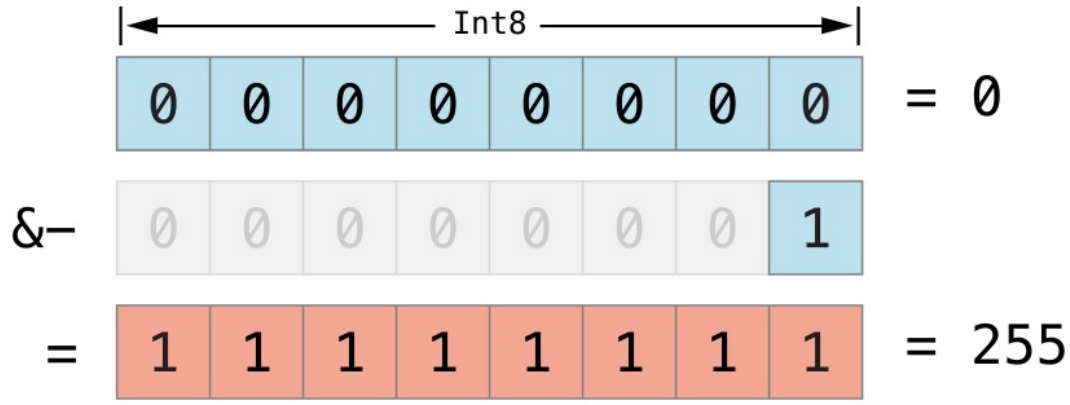
開始時 `willOverflow` 等於 `Int8` 所能容納的最大值 `255` (二進制 `11111111`)，然後用 `&+` 加 1。這使得 `UInt8` 型別無法表達這個新值的二進制，也就導致了這個新值的上溢位，可以參考下圖。溢位後，新值在 `UInt8` 的容納範圍內的部分是 `00000000`，也就是 `0`。



值的下溢位

數值也有可能因為太小而溢位。舉個範例：

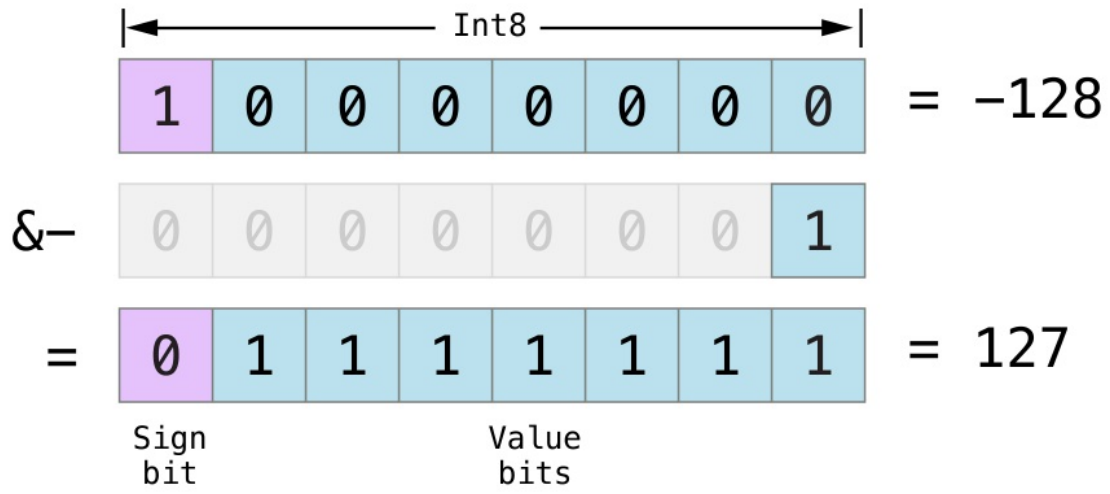
`UInt8` 的最小值是 `0` (二進制為 `00000000`)。使用 `&-` 進行溢位減 1，就會得到二進制的 `11111111` 即十進制的 `255`。



Swift 程式碼是這樣的:

```
var willUnderflow = UInt8.min
// willUnderflow 等於 UInt8 的最小值 0
willUnderflow = willUnderflow &- 1
// 此時 willUnderflow 等於 255
```

有號整數也有類似的下溢位，有號整數所有的減法也都是對包括在符號位在內的二進制數進行二進制減法的，這在「移位運算子」一節提到過。最小的有號整數是 `-128`，即二進制的 `10000000`。用溢位減法減去 1 後，變成了 `01111111`，即 `UInt8` 所能容納的最大整數 `127`。



來看看 Swift 程式碼：

```
var signedUnderflow = Int8.min
// signedUnderflow 等於最小的有號整數 -128
signedUnderflow = signedUnderflow &- 1
// 此時 signedUnderflow 等於 127
```

除以零溢位

對一個數除以 0（`i / 0`），或者對 0 取餘數（`i % 0`），會產生一個錯誤。

```
let x = 1
let y = x / 0
```

使用它們對應的可溢位的版本運算子 `&/` 和 `&%` 進行除以 0 運算時就會得到 `0` 值。

```
let x = 1
let y = x &/ 0
// y 等於 0
```

優先級和結合性（Precedence and Associativity）

運算子的優先級使得一些運算子優先於其他運算子，高優先級的運算子會先被計算。

結合性定義相同優先級的運算子在一起時是怎麼組合或關聯的，是和左邊一組還是和右邊一組，意思是「到底是和左邊的表達式結合呢，還是和右邊的表達式結合」。

在混合表達式中，運算子的優先級和結合性是非常重要的。舉個範例，為什麼下列表達式的結果為 `4`？

```
2 + 3 * 4 % 5
// 結果是 4
```

```
```

如果嚴格地從左計算到右，計算過程會是這樣：

- 2 plus 3 equals 5;
- 2 + 3 = 5
- 5 times 4 equals 20;
- 5 * 4 = 20
- 20 remainder 5 equals 0
- 20 / 5 = 4 余 0

但是正確答案是 `4` 而不是 `0`。優先級高的運算子要先計算，在 Swift 和 C 中，都是先乘除後加減的。所以，執行完乘法和取餘運算才能執行加減運算。

乘法和取餘擁有相同的優先級，在運算過程中，我們還需要結合性，乘法和取餘運算都是左結合的。這相當於在表達式中有隱藏的括號讓運算從左開始。

```
2 + ((3 * 4) % 5)
```

(3 * 4) = 12，所以這相當於：

```
2 + (12 % 5)
```

(12 % 5) = 2，所這又相當於

```
2 + 2
```

計算結果為 4。

查閱 Swift 運算子的優先級和結合性的完整列表，請看表達式。

> 注意：
> Swift 的運算子比 C 和 Objective-C 來得更簡單和保守，這意味著 Swift 跟以 C 為基礎的語言可能不一樣。所以，在移植已有程式碼到 Swift 時，注意去確保程式碼按你想的那樣去執行。

運算子函式（Operator Functions）

讓已有的運算子也可以對自定義的類別和結構進行運算，這稱為運算子重載。

這個範例展示了如何用 `+` 讓一個自定義的結構做加法。算術運算子 `+` 是一個二元運算子，因為它有兩個運算元，而且它必須出現在兩個運算元之間。

範例中定義了一個名為 `Vector2D` 的二維坐標向量 `(x, y)` 的結構，然後定義了讓兩個 `Vector2D` 的物件相加的運算子函式。

```
struct Vector2D {
    var x = 0.0, y = 0.0
}
@infix func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
```

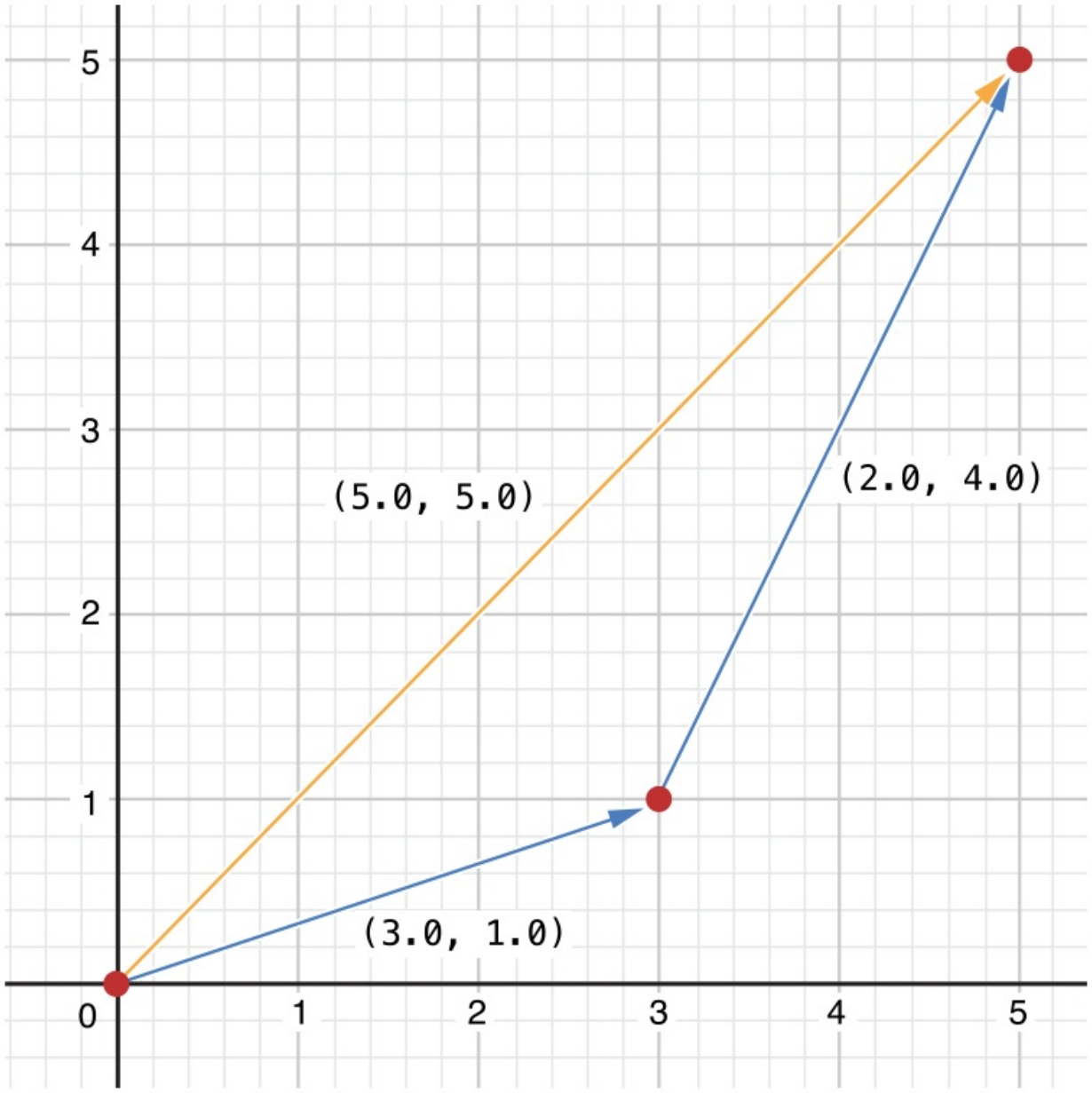
該運算子函式定義了一個全域的 `+` 函式，這個函式需要兩個 `Vector2D` 型別的參數，回傳值也是 `Vector2D` 。需要定義和實作一個中綴運算的時候，在關鍵字 `func` 之前寫上屬性 `@infix` 即可。

在這個程式碼實作中，參數被命名為了 `left` 和 `right` ，代表 `+` 左邊和右邊的兩個 `Vector2D` 物件。函式回傳了一個新的 `Vector2D` 的物件，這個物件的 `x` 和 `y` 分別等於兩個參數物件的 `x` 和 `y` 的和。

這個函式是全域的，而不是 `Vector2D` 結構的成員方法，所以任意兩個 `Vector2D` 物件都可以使用這個中綴運算子。

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector 是一個新的 `Vector2D`, 值為 (5.0, 5.0)
```

這個範例實作兩個向量 `(3.0, 1.0)` 和 `(2.0, 4.0)` 相加，得到向量 `(5.0, 5.0)` 的過程。如下圖示：



前綴和後綴運算子

一元 上個範例演示了一個二元中綴運算子的自定義實作，同樣我們也可以實作標準一元運算子。一元運算子只有一個運算

元，在運算元之前就是前綴的，如 `-a`；在運算元之後就是後綴的，如 `i++`。

實作一個前綴或後綴運算子時，在定義該運算子的時候於關鍵字 `func` 之前標注 `@prefix` 或 `@postfix` 屬性。

```
@prefix func - (vector: Vector2D) -> Vector2D {
    return Vector2D(x: -vector.x, y: -vector.y)
}
```

這段程式碼為 `Vector2D` 型別提供了一元減法運算 `-a`， `@prefix` 屬性表明這是個前綴運算子。

對於數值，一元減法運算子可以把正數變負數，把負數變正數。對於 `Vector2D`，一元減運算將其 `x` 和 `y` 都進進行一元減法運算。

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative 為 (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive 為 (3.0, 4.0)
```

組合賦值運算子

組合賦值是其他運算子和賦值運算子一起執行的運算。例如 `+=` 把加運算和賦值運算組合成一個運算。實作一個組合賦值符號需要使用 `@assignment` 屬性，還需要把運算子的左參數設置成 `inout`，因為這個參數會在運算子函式內直接修改它的值。

```
@assignment func += (inout left: Vector2D, right: Vector2D) {
    left = left + right
}
```

因為加法運算在之前定義過了，這裡無需重新定義。所以，加賦運算子函式使用已經存在的加法運算子函式來執行左值加右值的運算。

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original 現在為 (4.0, 6.0)
```

你可以將 `@assignment` 屬性和 `@prefix` 或 `@postfix` 屬性起來組合，實作一個 `Vector2D` 的前綴運算子。

```
@prefix @assignment func ++ (inout vector: Vector2D) -> Vector2D {
    vector += Vector2D(x: 1.0, y: 1.0)
    return vector
}
```

這個前綴使用了已經定義好的加法賦值運算，將自己加上一個值為 `(1.0, 1.0)` 的物件然後賦給自己，然後再將自己回傳。

```
var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
// toIncrement 現在是 (4.0, 5.0)
// afterIncrement 現在也是 (4.0, 5.0)
```

> 注意：
> 預設的賦值運算子是不可重載的。只有組合賦值運算子可以重載。三元條件運算子 `a?b:c` 也是不可重載。

相等運算子（Equivalence Operators）

自定義的類別或結構沒有預設的「相等（`==`）」或「不相等（`!=`）」這樣的相等運算子，所以自定義的類別和結構要使用相等運算 `==` 或 `!=` 就需要重載。

定義相等運算子函式跟定義其他中綴運算子雷同：

```
@infix func == (left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}

@infix func != (left: Vector2D, right: Vector2D) -> Bool {
    return !(left == right)
}
```

上述程式碼實作了相等運算子 `==` 來判斷兩個 `Vector2D` 物件是否有相等的值，相等的概念就是它們有相同的 `x` 值和相同的 `y` 值，我們就用這個邏輯來實作。接著使用 `==` 的結果實作了不相等運算子 `!=`。

現在我們可以使用這兩個運算子來判斷兩個 `Vector2D` 物件是否相等。

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    println("These two vectors are equivalent.")
}
// prints "These two vectors are equivalent."
```

自定義運算子

你可以宣告一些自定義的運算子，但自定義的運算子只能使用這些字元：`/ = - + * % < > ! & | ^ . ~`。

新的運算子宣告需在全域使用 `operator` 關鍵字宣告，可以宣告為前綴，中綴或後綴的。

```
operator prefix +++ {}
```

這段程式碼定義了一個新的前綴運算子叫 `+++`，在這之前 Swift 並不存在這個運算子。為了示範，我們讓 `+++` 對 `Vector2D` 物件的運算定義為「加倍」（doubling incrementer）這樣一個獨有的操作，這個操作使用了之前定義的加法賦值運算來實作自己加上自己然後回傳的運算。

```
@prefix @assignment func +++ (inout vector: Vector2D) -> Vector2D {
    vector += vector
    return vector
}
```

`Vector2D` 的 `+++` 的實作和 `++` 的實作很接近，唯一不同的前者是加自己，後者是加值為 `(1.0, 1.0)` 的向量。

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled 現在是 (2.0, 8.0)
// afterDoubling 現在也是 (2.0, 8.0)
```

自定義中綴運算子的優先級和結合性

可以為自定義的中綴運算子指定優先級和結合性。可以回頭看看優先級和結合性解釋這兩個因素是如何影響多種中綴運算子混合的表達式的計算結果。

結合性（associativity）的定義可以是 `left`、`right` 或 `none`。左結合運算子跟其他優先級相同的左結合運算子寫在一起時，會跟左邊的運算元結合。同理，右結合運算子會跟右邊的運算元結合。而非結合運算子不能跟其他相同優先級的運算子寫在一起。

結合性預設為 `none`，優先級（precedence）預設為 `100`。

以下範例定義了一個新的中綴運算子 `+-`，結合性為左結合，優先級為 `140`。

```
operator infix +- { associativity left precedence 140 }
func +- (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y - right.y)
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector 此時的值為 (4.0, -2.0)
```

這個運算子把兩個向量的 `x` 相加，把向量的 `y` 相減。因為他實際上是屬於加減運算，所以讓它保持了和加法一樣的結合性和優先級(`left` 和 `140`)。查閱完整的 Swift 預設結合性和優先級的設定，請參考表達式。

> 翻譯：dabing1022 校對：numbbbbb

關於語言附註

本頁內容包括：

- 如何閱讀語法

本書的這一節描述了Swift程式語言的形式語法。這裡描述的語法是為了幫助你更詳細的了解該語言，而不是讓你直接實作一個直譯器或編譯器。

Swift語言相對小點，這是由於在Swift程式碼中幾乎無處不在的許多常見的的型別，函式以及運算子都由Swift標準函式庫來定義。雖然這些型別，函式和運算子不是Swift語言本身的一部分，但是它們被廣泛用於這本書的討論和程式碼範例。

如何閱讀語法

用來描述Swift程式語言形式語法的記法遵循下面幾個約定：

-(https://github.com/numbbbbb)箭頭（→）用來標記語法產式，可以被理解為“可以包含”。

- 句法範疇由斜體文字表示，並出現在一個語法產式規則兩側。
- 義詞和標點符號由粗體固定寬度的文字顯示和只出現在一個語法產式規則的右邊。
- 選擇性的語法產式由豎線（|）分隔。當可選用的語法產式太多時，為了閱讀方便，它們將被拆分為多行語法產式規則。
- 在少數情況下，常見字體文字用來描述語法產式規則的右邊。
- 可選的句法範疇和文字用尾標 `opt` 來標記。

舉個範例，getter-setter的語法塊的定義如下：

> GRAMMAR OF A GETTER-SETTER BLOCK
> *getter-setter-block* → { *getter-clause setter-clause*opt } | { *setter-clause getter-clause*}

這個定義表明，一個getter-setter方法塊可以由一個getter子句後跟一個可選的setter子句構成，用大括號括起來，或者由一個setter子句後跟一個getter子句構成，用大括號括起來。上述的文法產生等價於下面的兩個產生，明確闡明如何二中擇一：

> GRAMMAR OF A GETTER-SETTER BLOCK
> getter-setter-block → { *getter-clause setter-clause*opt }
> getter-setter-block → { *setter-clause getter-clause*}

> 翻譯：superkam 校對：numbbbbb

詞法結構

本頁包含內容：

- 空白與註解（*Whitespace and Comments*）
- 識別符號（*Identifiers*）
- 關鍵字（*Keywords*）
- 字面量（*Literals*）
- 運算子（*Operators*）

Swift 的“詞法結構（*lexical structure*）”描述了如何在該語言中用字元序列構建合法標記，組成該語言中最底層的程式碼區塊，並在之後的章節中用於描述語言的其他部分。

通常，標記在隨後介紹的語法約束下，由 Swift 原始碼的輸入文字中提取可能的最長子字串生成。這種方法稱為“最長匹配項（*longest match*）”，或者“最大適合”（*maximal munch*）。

空白與註解

空白（*whitespace*）有兩個用途：分隔原始碼中的標記和區分運算子屬於前綴還是後綴，（參見 運算子）在其他情況下則會被忽略。以下的字元會被當作空白：空格（*space*）（U+0020）、換行（*line feed*）（U+000A）、回車（*carriage return*）（U+000D）、水平 tab（*horizontal tab*）（U+0009）、垂直 tab（*vertical tab*）（U+000B）、換頁符號（*form feed*）（U+000C）以及空（*null*）（U+0000）。

註解（*comments*）被編譯器當作空白處理。單行註解由 `//` 開始直到該行結束。多行註解由 `/*` 開始，以 `*/` 結束。可以嵌套註解，但注意註解標記必須匹配。

識別符號

識別符號（*identifiers*）可以由以下的字元開始：大寫或小寫的字母 `A` 到 `Z`、底線 `_`、基本多語言面（*Basic Multilingual Plane*）中的 Unicode 非組合字元以及基本多語言面以外的非專用區（*Private Use Area*）字元。首字元之後，識別符號允許使用數字和 Unicode 字元組合。

使用保留字（*reserved word*）作為識別符號，需要在其前後增加反引號 `` ` ``。例如，`class` 不是合法的識別符號，但可以使用 `` `class` ``。反引號不屬於識別符號的一部分，`` `x` `` 和 `x` 表示同一識別符號。

閉包（*closure*）中如果沒有明確指定參數名稱，參數將被隱式命名為 `$0`、`$1`、`$2`... 這些命名在閉包作用域內是合法的識別符號。

> 識別符號語法
> 識別符號 → 識別符號頭*(Head)* 識別符號字元列表 可選
> 識別符號 → `` ` `` 識別符號頭*(Head)* 識別符號字元列表 可選 `` ` ``
> 識別符號 → 隱式參數名
> 識別符號列表 → 識別符號 | 識別符號 , 識別符號列表
> 識別符號頭*(Head)* → Upper- or lowercase letter A through Z
> 識別符號頭*(Head)* → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or U+00B7–U+00BA
> 識別符號頭*(Head)* → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, or U+00F8–U+00FF
> 識別符號頭*(Head)* → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, or U+180F–U+1DBF
> 識別符號頭*(Head)* → U+1E00–U+1FFF

> 識別符號頭*(Head)* → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040, U+2054, or U+2060–U+206F
> 識別符號頭*(Head)* → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, or U+2776–U+2793
> 識別符號頭*(Head)* → U+2C00–U+2DFF or U+2E80–U+2FFF
> 識別符號頭*(Head)* → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, or U+3040–U+D7FF
> 識別符號頭*(Head)* → U+F900–U+FD3D, U+FD40–U+FDCF, U+FDF0–U+FE1F, or U+FE30–U+FE44
> 識別符號頭*(Head)* → U+FE47–U+FFFD
> 識別符號頭*(Head)* → U+10000–U+1FFFD, U+20000–U+2FFFD, U+30000–U+3FFFD, or U+40000–U+4FFFD
> 識別符號頭*(Head)* → U+50000–U+5FFFD, U+60000–U+6FFFD, U+70000–U+7FFFD, or U+80000–U+8FFFD
> 識別符號頭*(Head)* → U+90000–U+9FFFD, U+A0000–U+AFFFD, U+B0000–U+BFFFD, or U+C0000–U+CFFFD
> 識別符號頭*(Head)* → U+D0000–U+DFFFD or U+E0000–U+EFFFD
> 識別符號字元 → 數值 0 到 9
> 識別符號字元 → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–U+FE2F
> 識別符號字元 → 識別符號頭*(Head)*
> 識別符號字元列表 → 識別符號字元 識別符號字元列表 可選
> 隱式參數名 → **$** 十進制數字列表

關鍵字

被保留的關鍵字（*keywords*）不允許用作識別符號，除非被反引號跳脫，參見 識別符號。

- 用作宣告的關鍵字：
  class、*deinit*、*enum*、*extension*、*func*、*import*、*init*、*let*、*protocol*、*static*、*struct*、*subscript*、*typealias*、*var*
- 用作語句的關鍵字：
  break、*case*、*continue*、*default*、*do*、*else*、*fallthrough*、*if*、*in*、*for*、*return*、*switch*、*where*、*while*
- 用作表達和型別的關鍵字：
  as、*dynamicType*、*is*、*new*、*super*、*self*、*Self*、*Type*、*__COLUMN__*、*__FILE__*、*__FUNCTION__*、*__LINE__*
- 特定上下文中被保留的關鍵字：
  associativity、*didSet*、*get*、*infix*、*inout*、*left*、*mutating*、*none*、*nonmutating*、*operator*、*override*、*postfix*、*precedence*、*prefix*、*right*、*set*、*unowned*、*unowned(safe)*、*unowned(unsafe)*、*weak*、*willSet*，這些關鍵字在特定上下文之外可以被用於識別符號。

字面量

字面值表示整型、浮點型數字或文字型別的值，舉例如下：

```
42                // 整型字面量
3.14159           // 浮點型字面量
"Hello, world!"   // 文字型字面量
```

> 字面量語法
> 字面量 → 整型字面量 | 浮點數字面量 | 字串字面量

整型字面量

整型字面量（*integer literals*）表示未指定精度整型數的值。整型字面量預設用十進制表示，可以加前綴來指定其他的進制，二進制字面量加 `0b`，八進制字面量加 `0o`，十六進制字面量加 `0x`。

十進制字面量包含數字 `0` 至 `9`。二進制字面量只包含 `0` 或 `1`，八進制字面量包含數字 `0` 至 `7`，十六進制字面量包含數字 `0` 至 `9` 以及字母 `A` 至 `F` （大小寫均可）。

負整數的字面量在數字前加減號 `-`，比如 `-42`。

允許使用底線 `_` 來增加數字的可讀性，底線不會影響字面量的值。整型字面量也可以在數字前加 `0`，同樣不會影響字面量的值。

```
1000_000    // 等於 1000000
005         // 等於 5
```

除非特殊指定，整型字面量的預設型別為 Swift 標準函式庫型別中的 `Int`。Swift 標準函式庫還定義了其他不同長度以及是否帶符號的整數型別，請參考 整數型別。

> 整型字面量語法
> 整型字面量 → 二進制字面量
> 整型字面量 → 八進制字面量
> 整型字面量 → 十進制字面量
> 整型字面量 → 十六進制字面量
> 二進制字面量 → **0b** 二進制數字 二進制字面量字元列表 可選
> 二進制數字 → 數值 0 到 1
> 二進制字面量字元 → 二進制數字 | _
> 二進制字面量字元列表 → 二進制字面量字元 二進制字面量字元列表 可選
> 八進制字面量 → **0o** 八進字數字 八進制字元列表 可選
> 八進字數字 → 數值 0 到 7
> 八進制字元 → 八進字數字 | _
> 八進制字元列表 → 八進制字元 八進制字元列表 可選
> 十進制字面量 → 十進制數字 十進制字元列表 可選
> 十進制數字 → 數值 0 到 9
> 十進制數字列表 → 十進制數字 十進制數字列表 可選
> 十進制字元 → 十進制數字 | _
> 十進制字元列表 → 十進制字元 十進制字元列表 可選
> 十六進制字面量 → **0x** 十六進制數字 十六進制字面量字元列表 可選
> 十六進制數字 → 數值 0 到 9, a through f, or A through F
> 十六進制字元 → 十六進制數字 | _
> 十六進制字面量字元列表 → 十六進制字元 十六進制字面量字元列表 可選

浮點型字面量

浮點型字面量（*floating-point literals*）表示未指定精度浮點數的值。

浮點型字面量預設用十進制表示（無前綴），也可以用十六進制表示（加前綴 `0x`）。

十進制浮點型字面量（*decimal floating-point literals*）由十進制數字串後跟小數部分或指數部分（或兩者皆有）組成。十進制小數部分由小數點 `.` 後跟十進制數字串組成。指數部分由大寫或小寫字母 `e` 後跟十進制數字串組成，這串數字表示 `e` 之前的數量乘以 10 的幾次方。例如：`1.25e2` 表示 `1.25 × 10^2`，也就是 `125.0`；同樣，`1.25e−2` 表示 `1.25 × 10^−2`，也就是 `0.0125`。

十六進制浮點型字面量（*hexadecimal floating-point literals*）由前綴 `0x` 後跟可選的十六進制小數部分以及十六進制指數部分組成。十六進制小數部分由小數點後跟十六進制數字串組成。指數部分由大寫或小寫字母 `p` 後跟十進制數字串組成，這串數字表示 `p` 之前的數量乘以 2 的幾次方。例如：`0xFp2` 表示 `15 × 2^2`，也就是 `60`；同樣，`0xFp-2` 表示 `15 × 2^-2`，也就是 `3.75`。

與整型字面量不同，負的浮點型字面量由一元運算子減號 `-` 和浮點型字面量組成，例如 `-42.0`。這代表一個表達式，而不是一個浮點整型字面量。

允許使用底線 `_` 來增強可讀性，底線不會影響字面量的值。浮點型字面量也可以在數字前加 `0`，同樣不會影響字面量的值。

```
10_000.56     // 等於 10000.56
005000.76     // 等於 5000.76
```

除非特殊指定，浮點型字面量的預設型別為 Swift 標準函式庫型別中的 `Double`，表示64位浮點數。Swift 標準函式庫也定義 `Float` 型別，表示32位浮點數。

> 浮點型字面量語法
> 浮點數字面量 → 十進制字面量 十進制分數 可選 十進制指數 可選
> 浮點數字面量 → 十六進制字面量 十六進制分數 可選 十六進制指數
> 十進制分數 → **.** 十進制字面量
> 十進制指數 → 浮點數*e* 正負號 可選 十進制字面量
> 十六進制分數 → **.** 十六進制字面量 可選
> 十六進制指數 → 浮點數*p* 正負號 可選 十六進制字面量
> 浮點數*e* → **e** | **E**
> 浮點數*p* → **p** | **P**
> 正負號 → **+** | **-**

文字型字面量

文字型字面量（*string literal*）由雙引號中的字串組成，形式如下：

```
"characters"
```

文字型字面量中不能包含未跳脫的雙引號 `"`、未跳脫的反斜線 `\`、回車（*carriage return*）或換行（*line feed*）。

可以在文字型字面量中使用的跳脫特殊符號如下：

- 空字元（Null Character） `\0`
- 反斜線（Backslash） `\\`
- 水平 Tab （Horizontal Tab） `\t`
- 換行（Line Feed） `\n`
- 回車（Carriage Return） `\r`
- 雙引號（Double Quote） `\"`
- 單引號（Single Quote） `\'`

字元也可以用以下方式表示：

- `\x` 後跟兩位十六進制數字
- `\u` 後跟四位十六進制數字
- `\U` 後跟八位十六進制數字

後跟的數字表示一個 Unicode 碼點。

文字型字面量允許在反斜線小括號 `\()` 中插入表達式的值。插入表達式（*interpolated expression*）不能包含未跳脫的雙引號 `"`、反斜線 `\`、回車或者換行。表達式值的型別必須在 *String* 類別中有對應的初始化方法。

例如，以下所有文字型字面量的值相同：

```
"1 2 3"
"1 2 \(3)"
"1 2 \(1 + 2)"
var x = 3; "1 2 \(x)"
```

文字型字面量的預設型別為 `String`。組成字串的字元型別為 `Character`。更多有關 `String` 和 `Character` 的資訊請參照 字串和字元。

> 字元型字面量語法
> 字串字面量 → **"** 參考文字 **"**
> 參考文字 → 參考文字條目 參考文字 可選
> 參考文字條目 → 跳脫字元
> 參考文字條目 → **(** 表達式 **)**
> 參考文字條目 → 除了", \, U+000A, or U+000D的所有Unicode的字元
> 跳脫字元 → **\0** | **** | **\t** | **\n** | **\r** | **\"** | **\'**
> 跳脫字元 → **\x** 十六進制數字 十六進制數字
> 跳脫字元 → **\u** 十六進制數字 十六進制數字 十六進制數字 十六進制數字
> 跳脫字元 → **\U** 十六進制數字 十六進制數字 十六進制數字 十六進制數字 十六進制數字 十六進制數字 十六進制數字 十六進制數字

運算子

Swift 標準函式庫定義了許多可供使用的運算子，其中大部分在 基礎運算子 和 高級運算子 中進行了闡述。這裡將描述哪些字元能用作運算子。

運算子由一個或多個以下字元組成：`/`、`=`、`-`、`+`、`!`、`*`、`%`、`<`、`>`、`&`、`|`、`^`、`~`、`.`。也就是說，標記 `=`，`->`、`//`、`/*`、`*/`、`.` 以及一元前綴運算子 `&` 屬於保留字，這些標記不能被重寫或用於自定義運算子。

運算子兩側的空白被用來區分該運算子是否為前綴運算子（*prefix operator*）、後綴運算子（*postfix operator*）或二元運算子（*binary operator*）。規則總結如下：

- 如果運算子兩側都有空白或兩側都無空白，將被看作二元運算子。例如：`a+b` 和 `a + b` 中的運算子 `+` 被看作二元運算子。
- 如果運算子只有左側空白，將被看作前綴一元運算子。例如 `a ++b` 中的 `++` 被看作前綴一元運算子。
- 如果運算子只有右側空白，將被看作後綴一元運算子。例如 `a++ b` 中的 `++` 被看作後綴一元運算子。
- 如果運算子左側沒有空白並緊跟 `.`，將被看作後綴一元運算子。例如 `a++.b` 中的 `++` 被看作後綴一元運算子（同理，`a++ . b` 中的 `++` 是後綴一元運算子而 `a ++ .b` 中的 `++` 不是）.

鑒於這些規則，運算子前的字元 `(`、`[` 和 `{` ；運算子後的字元 `)`、`]` 和 `}` 以及字元 `,`、`;` 和 `:` 都將用於空白檢測。

以上規則需注意一點，如果運算子 `!` 或 `?` 左側沒有空白，則不管右側是否有空白都將被看作後綴運算子。如果將 `?` 用作可選型別（*optional type*）修飾，左側必須無空白。如果用於條件運算子 `? :`，必須兩側都有空白。

在特定構成中，以 `<` 或 `>` 開頭的運算子會被分離成兩個或多個標記，剩余部分以同樣的方式會被再次分離。因此，在 `Dictionary<String, Array<Int>>` 中沒有必要添加空白來消除閉合字元 `>` 的歧義。在這個範例中， 閉合字元 `>` 被看作單字元標記，而不會被誤解為移位運算子 `>>`。

要學習如何自定義新的運算子，請參考 自定義運算子 和 運算子宣告。學習如何重寫現有運算子，請參考 運算子方法。

> 運算子語法語法
> 運算子 → 運算子字元 運算子 可選
> 運算子字元 → **/** | **=** | **-** | **+** | **!** | ***** | **%** | **<** | **>** | **&** | **|** | **^** | **~** | **.**
> 二元運算子 → 運算子
> 前綴運算子 → 運算子
> 後綴運算子 → 運算子

> 翻譯：lyuka 校對：numbbbbb, stanzhai

型別（Types）

本頁包含內容：

Swift 語言存在兩種型別：命名型型別和複合型型別。命名型型別是指定義時可以給定名字的型別。命名型型別包括類別、結構、列舉和協定。比如，一個使用者定義的類別 `MyClass` 的實例擁有型別 `MyClass`。除了使用者定義的命名型型別，Swift 標準函式庫也定義了很多常用的命名型型別，包括那些表示陣列、字典和可選值的型別。

那些通常被其它語言認為是基本或初級的資料型型別（Data types）——比如表示數字、字元和字串——實際上就是命名型型別，Swift 標準函式庫是使用結構定義和實作它們的。因為它們是命名型型別，因此你可以按照“擴展和擴展宣告”章節裡討論的那樣，宣告一個擴展來增加它們的行為以適應你程式的需求。

複合型型別是沒有名字的型別，它由 Swift 本身定義。Swift 存在兩種複合型型別：函式型別和元組型別。一個複合型型別可以包含命名型型別和其它複合型型別。例如，元組型別 `(Int, (Int, Int))` 包含兩個元素：第一個是命名型型別 `Int`，第二個是另一個複合型型別 `(Int, Int)`.

本節討論 Swift 語言本身定義的型別，並描述 Swift 中的型別推斷行為。

> 型別語法
> 型別 → 陣列型別 | 函式型別 | 型別標識 | 元組型別 | 可選型別 | 隱式解析可選型別 | 協定合成型別 | 元型型別

型別注解

型別注解顯式地指定一個變數或表達式的值。型別注解始於冒號 `:` 終於型別，比如下面兩個範例：

```
let someTuple: (Double, Double) = (3.14159, 2.71828)
func someFunction(a: Int){ /* ... */ }
```

在第一個範例中，表達式 `someTuple` 的型別被指定為 `(Double, Double)`。在第二個範例中，函式 `someFunction` 的參數 `a` 的型別被指定為 `Int`。

型別注解可以在型別之前包含一個型別特性（type attributes）的可選列表。

> 型別注解語法
> 型別注解 → **:** 特性*(Attributes)*列表 可選 型別

型別識別符號

型別識別符號參考命名型型別或者是命名型/複合型型別的別名。

大多數情況下，型別識別符號參考的是同名的命名型型別。例如型別識別符號`Int`參考命名型型別`Int`，同樣，型別識別符號`Dictionary<String, Int>`參考命名型型別`Dictionary<String, Int>`。

在兩種情況下型別識別符號參考的不是同名的型別。情況一，型別識別符號參考的是命名型/複合型型別的型別別名。比如，在下面的範例中，型別識別符號使用`Point`來參考元組`(Int, Int)`：

```
typealias Point = (Int, Int)
let origin: Point = (0, 0)
```

情況二，型別識別符號使用dot(`.`)語法來表示在其它模塊（modules）或其它型別嵌套內宣告的命名型型別。例如，下面範例中的型別識別符號參考在`ExampleModule`模塊中宣告的命名型型別`MyType`：

```
var someValue: ExampleModule.MyType
```

> 型別標識語法
> 型別標識 → 型別名稱 泛型參數子句 可選 | 型別名稱 泛型參數子句 可選 . 型別標識
> 類別名 → 識別符號

元組型別

元組型別使用逗號隔開並使用括號括起來的0個或多個型別組成的列表。

你可以使用元組型別作為一個函式的回傳型別，這樣就可以使函式回傳多個值。你也可以命名元組型別中的元素，然後用這些名字來參考每個元素的值。元素的名字由一個識別符號和`:`組成。“函式和多回傳值”章節裡有一個展示上述特性的範例。

`void`是空元組型別`()`的別名。如果括號內只有一個元素，那麼該型別就是括號內元素的型別。比如，`(Int)`的型別是`Int`而不是`(Int)`。所以，只有當元組型別包含兩個元素以上時才可以標記元組元素。

> 元組型別語法
> 元組型別 → **(** 元組型別主體 可選 **)**
> 元組型別主體 → 元組型別的元素列表 **...** 可選
> 元組型別的元素列表 → 元組型別的元素 | 元組型別的元素 **,** 元組型別的元素列表
> 元組型別的元素 → 特性*(Attributes)*列表 可選 **inout** 可選 型別 | **inout** 可選 元素名 型別注解
> 元素名 → 識別符號

函式型別

函式型別表示一個函式、方法或閉包的型別，它由一個參數型別和回傳值型別組成，中間用箭頭`->`隔開：

- `parameter type` -> `return type`

由於 參數型別 和 回傳值型別 可以是元組型別，所以函式型別可以讓函式與方法支援多參數與多回傳值。

你可以對函式型別應用帶有參數型別`()`並回傳表達式型別的`auto_closure`屬性（見型別屬性章節）。一個自動閉包函式捕獲特定表達式上的隱式閉包而非表達式本身。下面的範例使用`auto_closure`屬性來定義一個很簡單的assert函式：

```
func simpleAssert(condition: @auto_closure () -> Bool, message: String){
    if !condition(){
        println(message)
    }
}
let testNumber = 5
simpleAssert(testNumber % 2 == 0, "testNumber isn't an even number.")
// prints "testNumber isn't an even number."
```

函式型別可以擁有一個可變長參數作為參數型別中的最後一個參數。從語法角度上講，可變長參數由一個基礎型別名字和 `...` 組成，如 `Int...`。可變長參數被認為是一個包含了基礎型別元素的陣列。即 `Int...` 就是 `Int[]`。關於使用可變長參數的範例，見章節“可變長參數”。

為了指定一個 `in-out` 參數，可以在參數型別前加 `inout` 前綴。但是你不可以對可變長參數或回傳值型別使用 `inout`。關於In-Out參數的討論見章節In-Out參數部分。

柯裡化函式（curried function）的型別相當於一個嵌套函式型別。例如，下面的柯裡化函式 `addTwoNumber()()` 的型別是 `Int -> Int -> Int`：

```
func addTwoNumbers(a: Int)(b: Int) -> Int{
    return a + b
}
addTwoNumbers(4)(5)      // returns 9
```

柯裡化函式的函式型別從右向左組成一組。例如，函式型別 `Int -> Int -> Int` 可以被理解為 `Int -> (Int -> Int)`——也就是說，一個函式傳入一個 `Int` 然後輸出作為另一個函式的輸入，然後又回傳一個 `Int`。例如，你可以使用如下嵌套函式來重寫柯裡化函式 `addTwoNumbers()()`：

```
func addTwoNumbers(a: Int) -> (Int -> Int){
    func addTheSecondNumber(b: Int) -> Int{
        return a + b
    }
    return addTheSecondNumber
}
addTwoNumbers(4)(5)     // Returns 9
```

> 函式型別語法
> 函式型別 → 型別 **->** 型別

陣列型別

Swift語言使用型別名緊接中括號 `[]` 來簡化標準函式庫中定義的命名型型別 `Array<T>`。換句話說，下面兩個宣告是等價的：

```
let someArray: String[] = ["Alex", "Brian", "Dave"]
let someArray: Array<String> = ["Alex", "Brian", "Dave"]
```

上面兩種情況下，常數 `someArray` 都被宣告為字串陣列。陣列的元素也可以通過 `[]` 獲取存取：`someArray[0]` 是指第0個元素 `“Alex”`。

上面的範例同時顯示，你可以使用 `[]` 作為初始值建構陣列，空的 `[]` 則用來來建構指定型別的空陣列。

```
var emptyArray: Double[] = []
```

你也可以使用鏈接起來的多個 `[]` 集合來建構多維陣列。例如，下例使用三個 `[]` 集合來建構三維整型陣列：

```
var array3D: Int[][][] = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
```

存取一個多維陣列的元素時，最左邊的下標指向最外層陣列的相應位置元素。接下來往右的下標指向第一層嵌入的相應位置元素，依次類別推。這就意味著，在上面的範例中，`array3D[0]` 是指 `[[1, 2], [3, 4]]`，`array3D[0][1]` 是指 `[3, 4]`，`array3D[0][1][1]` 則是指值 `4`。

關於Swift標準函式庫中 `Array` 型別的細節討論，見章節Arrays。

> 陣列型別語法
> 陣列型別 → 型別 **[ ]** | 陣列型別 **[ ]**

可選型別

Swift定義後綴 `?` 來作為標準函式庫中的定義的命名型型別 `Optional<T>` 的簡寫。換句話說，下面兩個宣告是等價的：

```
var optionalInteger: Int?
var optionalInteger: Optional<Int>
```

在上述兩種情況下，變數 `optionalInteger` 都被宣告為可選整型型別。注意在型別和 `?` 之間沒有空格。

型別 `Optional<T>` 是一個列舉，有兩種形式，`None` 和 `Some(T)`，又來代表可能出現或可能不出現的值。任意型別都可以被顯式的宣告（或隱式的轉換）為可選型別。當宣告一個可選型別時，確保使用括號給 `?` 提供合適的作用範圍。比如說，宣告一個整型的可選陣列，應寫作 `(Int[])?`，寫成 `Int[]?` 的話則會出錯。

如果你在宣告或定義可選變數或特性的時候沒有提供初始值，它的值則會自動賦成缺省值 `nil`。

可選符合 `LogicValue` 協定，因此可以出現在布林值環境下。此時，如果一個可選型別 `T?` 實例包含有型別為 `T` 的值（也就是說值為 `Optional.Some(T)`），那麼此可選型別就為 `true`，否則為 `false`。

如果一個可選型別的實例包含一個值，那麼你就可以使用後綴運算子 `!` 來獲取該值，正如下面描述的：

```
optionalInteger = 42
optionalInteger!      // 42
```

使用 `!` 運算子獲取值為 `nil` 的可選項會導致執行錯誤（runtime error）。

你也可以使用可選鏈和可選綁定來選擇性的執行可選表達式上的操作。如果值為 `nil`，不會執行任何操作因此也就沒有執行錯誤產生。

更多細節以及更多如何使用可選型別的範例，見章節“可選”。

> 可選型別語法
> 可選型別 → 型別 **?**

隱式解析可選型別

Swift語言定義後綴 `!` 作為標準函式庫中命名型別 `ImplicitlyUnwrappedOptional<T>` 的簡寫。換句話說，下面兩個宣告等價：

```
var implicitlyUnwrappedString: String!
var implicitlyUnwrappedString: ImplicitlyUnwrappedOptional<String>
```

上述兩種情況下，變數 `implicitlyUnwrappedString` 被宣告為一個隱式解析可選型別的字串。注意型別與 `!` 之間沒有空格。

你可以在使用可選的地方同樣使用隱式解析可選。比如，你可以將隱式解析可選的值賦給變數、常數和可選特性，反之亦然。

有了可選，你在宣告隱式解析可選變數或特性的時候就不用指定初始值，因為它有缺省值 `nil`。

由於隱式解析可選的值會在使用時自動解析，所以沒必要使用運算子 `!` 來解析它。也就是說，如果你使用值為 `nil` 的隱式解析可選，就會導致執行錯誤。

使用可選鏈會選擇性的執行隱式解析可選表達式上的某一個操作。如果值為 `nil`，就不會執行任何操作，因此也不會產生執行錯誤。

關於隱式解析可選的更多細節，見章節“隱式解析可選”。

> 隱式解析可選型別(Implicitly Unwrapped Optional Type)語法
> 隱式解析可選型別 → 型別 **!**

協定合成型別

協定合成型別是一種符合每個協定的指定協定列表型別。協定合成型別可能會用在型別注解和泛型參數中。

協定合成型別的形式如下：

```
protocol<Protocol 1, Procotol 2>
```

協定合成型別允許你指定一個值，其型別可以適配多個協定的條件，而且不需要定義一個新的命名型協定來繼承其它想要適配的各個協定。比如，協定合成型別 `protocol<Protocol A, Protocol B, Protocol C>` 等效於一個從 `Protocol A`， `Protocol B`， `Protocol C` 繼承而來的新協定 `Protocol D`，很顯然這樣做有效率的多，甚至不需引入一個新名字。

協定合成列表中的每項必須是協定名或協定合成型別的型別別名。如果列表為空，它就會指定一個空協定合成列表，這樣每個型別都能適配。

> 協定合成型別語法
> 協定合成型別 → **protocol <** 協定識別符號列表 可選 **>**
> 協定識別符號列表 → 協定識別符號 | 協定識別符號 , 協定識別符號列表
> 協定識別符號 → 型別標識

元型別

元型別是指所有型別的型別，包括類別、結構、列舉和協定。

類別、結構或列舉型別的元型別是相應的型別名緊跟 `.Type`。協定型別的元型別——並不是執行時適配該協定的具體型別——是該協定名字緊跟 `.Protocol`。比如，類別 `SomeClass` 的元型別就是 `SomeClass.Type`， 協定 `SomeProtocol` 的元型別就是 `SomeProtocal.Protocol`。

你可以使用後綴 `self` 表達式來獲取型別。比如， `SomeClass.self` 回傳 `SomeClass` 本身，而不是 `SomeClass` 的一個實例。同樣， `SomeProtocol.self` 回傳 `SomeProtocol` 本身，而不是執行時適配 `SomeProtocol` 的某個型別的實例。還可以對型別的實例

使用 `dynamicType` 表達式來獲取該實例在執行階段的型別，如下所示：

```
class SomeBaseClass {
    class func printClassName() {
        println("SomeBaseClass")
    }
}
class SomeSubClass: SomeBaseClass {
    override class func printClassName() {
        println("SomeSubClass")
    }
}
let someInstance: SomeBaseClass = SomeSubClass()
// someInstance is of type SomeBaseClass at compile time, but
// someInstance is of type SomeSubClass at runtime
someInstance.dynamicType.printClassName()
// prints "SomeSubClass
```

> 元(Metatype)型別語法
> 元型別 → 型別 **.** **Type** | 型別 **.** **Protocol** x

型別繼承子句

型別繼承子句被用來指定一個命名型型別繼承哪個類別且適配哪些協定。型別繼承子句開始於冒號 `:`，緊跟由 `,` 隔開的型別識別符號列表。

類別可以繼承單個超類別，適配任意數量的協定。當定義一個類別時，超類別的名字必須出現在型別識別符號列表首位，然後跟上該類別需要適配的任意數量的協定。如果一個類別不是從其它類別繼承而來，那麼列表可以以協定開頭。關於類別繼承更多的討論和範例，見章節“繼承”。

其它命名型型別可能只繼承或適配一個協定列表。協定型別可能繼承於其它任意數量的協定。當一個協定型別繼承於其它協定時，其它協定的條件集合會被集成在一起，然後其它從當前協定繼承的任意型別必須適配所有這些條件。

列舉定義中的型別繼承子句可以是一個協定列表，或是指定原始值的列舉，一個單獨的指定原始值型別的命名型型別。使用型別繼承子句來指定原始值型別的列舉定義的範例，見章節“原始值”。

> 型別繼承子句語法
> 型別繼承子句 → **:** 型別繼承列表
> 型別繼承列表 → 型別標識 | 型別標識 **,** 型別繼承列表

型別推斷

Swift廣泛的使用型別推斷，從而允許你可以忽略很多變數和表達式的型別或部分型別。比如，對於 `var x: Int = 0`，你可以完全忽略型別而簡寫成 `var x = 0` ——編譯器會正確的推斷出 `x` 的型別 `Int`。類似的，當完整的型別可以從上下文推斷出來時，你也可以忽略型別的一部分。比如，如果你寫了 `let dict: Dictionary = ["A": 1]`，編譯提也能推斷出 `dict` 的型別是 `Dictionary<String, Int>`。

在上面的兩個範例中，型別資訊從表達式樹（expression tree）的葉子節點傳向根節點。也就是說，`var x: Int = 0` 中 `x` 的型別首先根據 `0` 的型別進行推斷，然後將該型別資訊傳遞到根節點（變數 `x`）。

在Swift中，型別資訊也可以反方向流動——從根節點傳向葉子節點。在下面的範例中，常數 `eFloat` 上的顯式型別注解（`:Float`）導致數字字面量 `2.71828` 的型別是 `Float` 而非 `Double`。

```
let e = 2.71828 // The type of e is inferred to be Double.
let eFloat: Float = 2.71828 // The type of eFloat is Float.
```

Swift中的型別推斷在單獨的表達式或語句水平上進行。這意味著所有用於推斷型別的資訊必須可以從表達式或其某個子表達式的型別檢查中獲取。

> 翻譯：sg552 校對：numbbbbb, stanzhai

表達式（Expressions）

本頁包含內容：

Swift 中存在四種表達式： 前綴（prefix）表達式，二元（binary）表達式，主要（primary）表達式和後綴（postfix）表達式。表達式可以回傳一個值，以及執行某些邏輯（causes a side effect）。

前綴表達式和二元表達式就是對某些表達式使用各種運算子（operators）。 主要表達式是最短小的表達式，它提供了獲取（變數的）值的一種途徑。 後綴表達式則允許你建立複雜的表達式，例如配合函式呼叫和成員存取。 每種表達式都在下面有詳細論述～

> 表達式語法
> 表達式 → 前綴表達式 二元表達式列表 可選
> 表達式列表 → 表達式 | 表達式 , 表達式列表

前綴表達式（Prefix Expressions）

前綴表達式由 前綴符號和表達式組成。（這個前綴符號只能接收一個參數）

Swift 標準函式庫支援如下的前綴運算子：

- ++ 累加1 （increment）
- -- 累減1 （decrement）
- ! 邏輯否 （Logical NOT ）
- ~ 按位否 （Bitwise NOT ）
- + 加（Unary plus）
- - 減（Unary minus）

對於這些運算子的使用，請參見： Basic Operators and Advanced Operators

作為對上面標準函式庫運算子的補充，你也可以對 某個函式的參數使用 '&'運算子。 更多資訊，請參見： "In-Out parameters".

> 前綴表達式語法
> 前綴表達式 → 前綴運算子 可選 後綴表達式
> 前綴表達式 → 寫入寫出*(in-out)*表達式
> 寫入寫出*(in-out)*表達式 → & 識別符號

二元表達式（Binary Expressions）

二元表達式由 "左邊參數" + "二元運算子" + "右邊參數" 組成, 它有如下的形式：

> `left-hand argument` `operator` `right-hand argument`

Swift 標準函式庫提供了如下的二元運算子：

- 求冪相關（無結合，優先級160）
  - << 按位左移（Bitwise left shift）
  - > > 按位右移（Bitwise right shift）
- 乘除法相關（左結合，優先級150）
  - * 乘
  - / 除
  - % 取餘
  - &* 乘法，忽略溢出（ Multiply, ignoring overflow）
  - &/ 除法，忽略溢出（Divide, ignoring overflow）
  - &% 取餘, 忽略溢出（ Remainder, ignoring overflow）
  - & 位與（ Bitwise AND）
- 加減法相關（左結合, 優先級140）
  - + 加
  - - 減
  - &+ Add with overflow
  - &- Subtract with overflow
  - | 按位或（Bitwise OR ）
  - ^ 按位異或（Bitwise XOR）
- Range （無結合,優先級 135）
  - .. 半閉值域 Half-closed range
  - ... 全閉值域 Closed range
- 型別轉換 （無結合,優先級 132）
  - is 型別檢查（ type check）
  - as 型別轉換（ type cast）
- Comparative （無結合,優先級 130）
  - < 小於
  - <= 小於等於
  - > 大於
  - >= 大於等於
  - == 等於
  - != 不等
  - === 恆等於
  - !== 不恆等
  - ~= 模式匹配（ Pattern match）
- 合取（ Conjunctive）（左結合,優先級 120）
  - && 邏輯且（Logical AND）
- 析取（Disjunctive）（左結合,優先級 110）
  - || 邏輯或（ Logical OR）
- 三元條件（Ternary Conditional ）（右結合,優先級 100）
  - ?: 三元條件 Ternary conditional
- 賦值 （Assignment）（右結合, 優先級 90）
  - = 賦值（Assign）
  - *= Multiply and assign
  - /= Divide and assign
  - %= Remainder and assign
  - += Add and assign
  - -= Subtract and assign

- <<= Left bit shift and assign
- = Right bit shift and assign

- &= Bitwise AND and assign
- ^= Bitwise XOR and assign
- |= Bitwise OR and assign
- &&= Logical AND and assign
- ||= Logical OR and assign

關於這些運算子（operators）的更多資訊，請參見：Basic Operators and Advanced Operators.

> 注意
> 在解析時, 一個二元表達式表示為一個一級陣列（a flat list）, 這個陣列（List）根據運算子的先後順序，被轉換成了一個tree. 例如： 2 + 3 *5* 首先被認為是： *2*, +, `3`, , 5. 隨後它被轉換成 tree （2 + （3 * 5））

> 二元表達式語法
> 二元表達式 → 二元運算子 前綴表達式
> 二元表達式 → 賦值運算子 前綴表達式
> 二元表達式 → 條件運算子 前綴表達式
> 二元表達式 → 型別轉換運算子
> 二元表達式列表 → 二元表達式 二元表達式列表 可選

賦值表達式（Assignment Operator）

The assigment operator sets a new value for a given expression. It has the following form: 賦值表達式會對某個給定的表達式賦值。 它有如下的形式；

> `expression` = `value`

就是把右邊的 *value* 賦值給左邊的 *expression*. 如果左邊的*expression* 需要接收多個參數（是一個tuple ），那麼右邊必須也是一個具有同樣數量參數的tuple. （允許嵌套的tuple）

```
(a, _, (b, c)) = ("test", 9.45, (12, 3))
// a is "test", b is 12, c is 3, and 9.45 is ignored
```

賦值運算子不回傳任何值。

> 賦值運算子語法
> 賦值運算子 → =

三元條件運算子（Ternary Conditional Operator）

三元條件運算子 是根據條件來獲取值。 形式如下：

> `condition` ? `expression used if true` : `expression used if false`

如果 `condition` 是true, 那麼回傳 第一個表達式的值（此時不會呼叫第二個表達式）， 否則回傳第二個表達式的值（此時不會呼叫第一個表達式）。

想看三元條件運算子的範例，請參見： Ternary Conditional Operator.

> 三元條件運算子語法

> 三元條件運算子 → **?** 表達式 **:**

型別轉換運算子（Type-Casting Operators）

有兩種型別轉換運算子： as 和 is. 它們有如下的形式：

> `expression` as `type`
> `expression` as? `type`
> `expression` is `type`

as 運算子會把 `目標表達式` 轉換成指定的 `型別`（specified type），過程如下：

- 如果型別轉換成功， 那麼目標表達式就會回傳指定型別的實例（instance）. 例如：把子類別（subclass）變成父類別（superclass）時.
- 如果轉換失敗，則會拋出編譯錯誤（ compile-time error）。
- 如果上述兩個情況都不是（也就是說，編譯器在編譯時期無法確定轉換能否成功，） 那麼目標表達式就會變成指定的型別的optional. （is an optional of the specified type ） 然後在執行時，如果轉換成功， 目標表達式就會作為 optional的一部分來回傳， 否則，目標表達式回傳nil. 對應的範例是： 把一個 superclass 轉換成一個 subclass.

```
class SomeSuperType {}
class SomeType: SomeSuperType {}
class SomeChildType: SomeType {}
let s = SomeType()

let x = s as SomeSuperType  // known to succeed; type is SomeSuperType
let y = s as Int            // known to fail; compile-time error
let z = s as SomeChildType  // might fail at runtime; type is SomeChildType?
```

使用'as'做型別轉換跟正常的型別宣告，對於編譯器來說是一樣的。例如：

```
let y1 = x as SomeType  // Type information from 'as'
let y2: SomeType = x    // Type information from an annotation
```

'is' 運算子在“執行時（runtime）”會做檢查。 成功會回傳true, 否則 false

The check must not be known to be true or false at compile time. The following are invalid: 上述檢查在“編譯時（compile time）”不能使用。 例如下面的使用是錯誤的：

```
"hello" is String
"hello" is Int
```

關於型別轉換的更多內容和範例，請參見： Type Casting.

> 型別轉換運算子(type-casting-operator)語法
> 型別轉換運算子 → **is** 型別 | **as ?** 可選 型別

主表達式（Primary Expressions）

`主表達式` 是最基本的表達式。 它們可以跟 前綴表達式，二元表達式，後綴表達式以及其他主要表達式組合使用。

> 主表達式語法
> 主表達式 → 識別符號 泛型參數子句 可選

> 主表達式 → 字面量表達式
> 主表達式 → *self*表達式
> 主表達式 → 超類別表達式
> 主表達式 → 閉包表達式
> 主表達式 → 圓括號表達式
> 主表達式 → 隱式成員表達式
> 主表達式 → 通配符表達式

字元型表達式（Literal Expression）

由這些內容組成：普通的字元（string, number），一個字元的字典或者陣列，或者下面列表中的特殊字元。

字元（**Literal**）	型別（**Type**）	值（**Value**）
*_FILE_*	String	所在的文件名
*_LINE_*	Int	所在的行數
*_COLUMN_*	Int	所在的列數
*_FUNCTION_*	String	所在的function 的名字

在某個函式（function）中，`__FUNCTION__` 會回傳當前函式的名字。 在某個方法（method）中，它會回傳當前方法的名字。在某個property 的getter/setter中會回傳這個屬性的名字。 在init/subscript中 只有的特殊成員（member）中會回傳這個keyword的名字，在某個文件的頂端（the top level of a file），它回傳的是當前module的名字。

一個array literal，是一個有序的值的集合。 它的形式是：

> `[ value 1 , value 2 , ... ]`

陣列中的最後一個表達式可以緊跟一個逗號（','）. []表示空陣列 。 array literal的type是 T[], 這個T就是陣列中元素的type. 如果該陣列中有多種type, T則是跟這些type的公共supertype最接近的type.（closest common supertype）

一個 `dictionary literal` 是一個包含無序的鍵值對（key-value pairs）的集合，它的形式是:

> `[ key 1 : value 1 , key 2 : value 2 , ... ]`

dictionary 的最後一個表達式可以是一個逗號（','）. [:] 表示一個空的dictionary. 它的type是 Dictionary （這裡KeyType表示key的type, ValueType表示 value的type） 如果這個dictionary 中包含多種 types, 那麼KeyType, Value 則對應著它們的公共supertype最接近的type（ closest common supertype）.

> 字面量表達式語法
> 字面量表達式 → 字面量
> 字面量表達式 → 陣列字面量 | 字典字面量
> 字面量表達式 → **__FILE__** | **__LINE__** | **__COLUMN__** | **__FUNCTION__**
> 陣列字面量 → **[** 陣列字面量項列表 可選 **]**
> 陣列字面量項列表 → 陣列字面量項 **,** 可選 | 陣列字面量項 **,** 陣列字面量項列表
> 陣列字面量項 → 表達式
> 字典字面量 → **[** 字典字面量項列表 **]** | **[** **:** **]**
> 字典字面量項列表 → 字典字面量項 **,** 可選 | 字典字面量項 **,** 字典字面量項列表
> 字典字面量項 → 表達式 **:** 表達式

self表達式（Self Expression）

self表達式是對 當前type 或者當前instance的參考。它的形式如下：

> self
> self.`member name`
> self[`subscript index`]
> self（`initializer arguments`）
> self.init（`initializer arguments`）

如果在 initializer, subscript, instance method中， self等同於當前type的instance. 在一個靜態方法（static method）, 類別方法（class method）中， self等同於當前的type.

當存取 member（成員變數時）， self 用來區分重名變數（例如函式的參數）. 例如， （下面的 self.greeting 指的是 var greeting: String, 而不是 init（greeting: String））

```
class SomeClass {
    var greeting: String
    init (greeting: String) {
        self.greeting = greeting
    }
}
```

在mutating 方法中， 你可以使用self 對 該instance進行賦值。

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX (deltaX: Double, y deltaY: Double) {
        self = Point (x: x + deltaX, y: y + deltaY)
    }
}
```

> Self 表達式語法
> *self*表達式 → **self**
> *self*表達式 → **self .** 識別符號
> *self*表達式 → **self [** 表達式 **]**
> *self*表達式 → **self . init**

超類別表達式（Superclass Expression）

超類別表達式可以使我們在某個class中存取它的超類別. 它有如下形式：

> super.`member name`
> super[`subscript index`]
> super.init（`initializer arguments`）

形式1 用來存取超類別的某個成員（member）. 形式2 用來存取該超類別的 subscript 實作。 形式3 用來存取該超類別的 initializer.

子類別（subclass）可以通過超類別（superclass）表達式在它們的 member, subscripting 和 initializers 中來利用它們超類別中的某些實作（既有的方法或者邏輯）。

> 超類別(superclass)表達式語法
> 超類別表達式 → 超類別方法表達式 | 超類別下標表達式 | 超類別建構器表達式
> 超類別方法表達式 → **super .** 識別符號
> 超類別下標表達式 → **super [** 表達式 **]**
> 超類別建構器表達式 → **super . init**

閉包表達式（Closure Expression）

閉包（closure） 表達式可以建立一個閉包（在其他語言中也叫 lambda, 或者 匿名函式（anonymous function））. 跟函式（function）的宣告一樣， 閉包（closure）包含了可執行的程式碼（跟方法主體（statement）類似） 以及接收（capture）的參數。 它的形式如下：

```
{ (parameters) -> return type in
    statements
}
```

閉包的參數宣告形式跟方法中的宣告一樣, 請參見：Function Declaration.

閉包還有幾種特殊的形式, 讓使用更加簡潔：

- 閉包可以省略 它的參數的type 和回傳值的type. 如果省略了參數和參數型別，就也要省略 'in'關鍵字。 如果被省略的type無法被編譯器獲知（inferred） ，那麼就會拋出編譯錯誤。
- 閉包可以省略參數，轉而在方法體（statement）中使用 $0, $1, $2 來參考出現的第一個，第二個，第三個參數。
- 如果閉包中只包含了一個表達式，那麼該表達式就會自動成為該閉包的回傳值。 在執行 'type inference '時，該表達式也會回傳。

下面幾個 閉包表達式是 等價的：

```
myFunction {
    (x: Int, y: Int) -> Int in
    return x + y
}

myFunction {
    (x, y) in
    return x + y
}

myFunction { return $0 + $1 }

myFunction { $0 + $1 }
```

關於 向閉包中傳遞參數的內容，參見： Function Call Expression.

閉包表達式可以通過一個參數列表（capture list） 來顯式指定它需要的參數。 參數列表 由中括號 [] 括起來，裡面的參數由逗號','分隔。一旦使用了參數列表，就必須使用'in'關鍵字（在任何情況下都得這樣做，包括忽略參數的名字，type, 回傳值時等等）。

在閉包的參數列表（ capture list）中， 參數可以宣告為 'weak' 或者 'unowned' .

```
myFunction { print (self.title) }                   // strong capture
myFunction { [weak self] in print (self!.title) }   // weak capture
myFunction { [unowned self] in print (self.title) } // unowned capture
```

在參數列表中，也可以使用任意表達式來賦值. 該表達式會在 閉包被執行時賦值，然後按照不同的力度來獲取（這句話請慎重理解）。（captured with the specified strength. ） 例如：

```
// Weak capture of "self.parent" as "parent"
myFunction { [weak parent = self.parent] in print (parent!.title) }
```

關於閉包表達式的更多資訊和範例，請參見： Closure Expressions.

> 閉包表達式語法

> 閉包表達式 → **{** 閉包簽名*(Signational)* 可選 多條語句*(Statements)* **}**
> 閉包簽名*(Signational)* → 參數子句 函式結果 可選 **in**
> 閉包簽名*(Signational)* → 識別符號列表 函式結果 可選 **in**
> 閉包簽名*(Signational)* → 捕獲*(Capature)*列表 參數子句 函式結果 可選 **in**
> 閉包簽名*(Signational)* → 捕獲*(Capature)*列表 識別符號列表 函式結果 可選 **in**
> 閉包簽名*(Signational)* → 捕獲*(Capature)*列表 **in**
> 捕獲*(Capature)*列表 → **[** 捕獲*(Capature)*說明符 表達式 **]**
> 捕獲*(Capature)*說明符 → **weak** | **unowned** | **unowned(safe)** | **unowned(unsafe)**

隱式成員表達式（Implicit Member Expression）

在可以判斷出型別（type）的上下文（context）中，隱式成員表達式是存取某個type的member（ 例如 class method, enumeration case） 的簡潔方法。 它的形式是：

> `. member name`

範例：

```
var x = MyEnumeration.SomeValue
x = .AnotherValue
```

> 隱式成員表達式語法
> 隱式成員表達式 → **.** 識別符號

圓括號表達式（Parenthesized Expression）

圓括號表達式由多個子表達式和逗號','組成。 每個子表達式前面可以有 identifier x: 這樣的可選前綴。形式如下：

> `(` `identifier 1` `:` `expression 1` `,` `identifier 2` `:` `expression 2` `,` `...` `)`

圓括號表達式用來建立tuples， 然後把它做為參數傳遞給 function. 如果某個圓括號表達式中只有一個 子表達式，那麼它的type就是 子表達式的type。例如： （1）的 type是Int, 而不是（Int）

> 圓括號表達式(Parenthesized Expression)語法
> 圓括號表達式 → **(** 表達式元素列表 可選 **)**
> 表達式元素列表 → 表達式元素 | 表達式元素 **,** 表達式元素列表
> 表達式元素 → 表達式 | 識別符號 **:** 表達式

通配符表達式（Wildcard Expression）

通配符表達式用來忽略傳遞進來的某個參數。例如：下面的程式碼中，10被傳遞給x, 20被忽略（譯注：好奇葩的語法。。。）

```
(x, _) = (10, 20)
// x is 10, 20 is ignored
```

> 通配符表達式語法
> 通配符表達式 → _

後綴表達式（Postfix Expressions）

後綴表達式就是在某個表達式的後面加上 運算子。 嚴格的講，每個主要表達式（primary expression）都是一個後綴表達式

Swift 標準函式庫提供了下列後綴表達式：

- ++ Increment
- -- Decrement

對於這些運算子的使用，請參見： Basic Operators and Advanced Operators

> 後綴表達式語法
> 後綴表達式 → 主表達式
> 後綴表達式 → 後綴表達式 後綴運算子
> 後綴表達式 → 函式呼叫表達式
> 後綴表達式 → 建構器表達式
> 後綴表達式 → 顯示成員表達式
> 後綴表達式 → 後綴*self*表達式
> 後綴表達式 → 動態型別表達式
> 後綴表達式 → 下標表達式
> 後綴表達式 → 強制取值*(Forced Value)*表達式
> 後綴表達式 → 可選鏈*(Optional Chaining)*表達式

函式呼叫表達式（Function Call Expression）

函式呼叫表達式由函式名和參數列表組成。它的形式如下：

> `function name` （ `argument value 1` , `argument value 2` ）

The function name can be any expression whose value is of a function type. （不用翻譯了, 太羅嗦）

如果該function 的宣告中指定了參數的名字，那麼在呼叫的時候也必須得寫出來. 例如：

> `function name` （ `argument name 1` : `argument value 1` , `argument name 2` : `argument value 2` ）

可以在 函式呼叫表達式的尾部（最後一個參數之後）加上 一個閉包（closure）， 該閉包會被目標函式理解並執行。它具有如下兩種寫法：

```
// someFunction takes an integer and a closure as its arguments
someFunction (x, {$0 == 13})
someFunction (x)  {$0 == 13}
```

如果閉包是該函式的唯一參數，那麼圓括號可以省略。

```
// someFunction takes a closure as its only argument
myData.someMethod ()  {$0 == 13}
myData.someMethod {$0 == 13}
```

> 函式呼叫表達式語法
> 函式呼叫表達式 → 後綴表達式 圓括號表達式
> 函式呼叫表達式 → 後綴表達式 圓括號表達式 可選 後綴閉包*(Trailing Closure)*
> 後綴閉包*(Trailing Closure)* → 閉包表達式

初始化函式表達式（Initializer Expression）

Initializer表達式用來給某個Type初始化。 它的形式如下：

> `expression`.init（`initializer arguments`）

（Initializer表達式用來給某個Type初始化。） 跟函式（function）不同， initializer 不能回傳值。

```
var x = SomeClass.someClassFunction // ok
var y = SomeClass.init              // error
```

可以通過 initializer 表達式來委托呼叫（delegate to ）到superclass的initializers.

```
class SomeSubClass: SomeSuperClass {
    init () {
        // subclass initialization goes here
        super.init ()
    }
}
```

> 建構器表達式語法
> 建構器表達式 → 後綴表達式 **.** **init**

顯式成員表達式（Explicit Member Expression）

顯示成員表達式允許我們存取type, tuple, module的成員變數。它的形式如下：

> `expression` . `member name`

該member 就是某個type在宣告時候所定義（declaration or extension） 的變數, 例如：

```
class SomeClass {
    var someProperty = 42
}
let c = SomeClass ()
let y = c.someProperty  // Member access
```

對於tuple, 要根據它們出現的順序（0, 1, 2...）來使用:

```
var t =  (10, 20, 30)
t.0 = t.1
// Now t is  (20, 20, 30)
```

The members of a module access the top-level declarations of that module. （不確定：對於某個module的member的呼叫，只能呼叫在top-level宣告中的member.）

> 顯式成員表達式語法
> 顯示成員表達式 → 後綴表達式 **.** 十進制數字
> 顯示成員表達式 → 後綴表達式 **.** 識別符號 泛型參數子句 可選

後綴self表達式（Postfix Self Expression）

後綴表達式由 某個表達式 + '.self' 組成. 形式如下：

> `expression`.self
> `type`.self

形式1 表示會回傳 expression 的值。例如： x.self 回傳 x

形式2：回傳對應的type。我們可以用它來動態的獲取某個instance的type。

> 後綴Self 表達式語法
> 後綴*self*表達式 → 後綴表達式 **. self**

dynamic表達式（Dynamic Type Expression）

（因為dynamicType是一個獨有的方法，所以這裡保留了英文單詞，未作翻譯, --- 類似與self expression）

dynamicType 表達式由 某個表達式 + '.dynamicType' 組成。

> `expression`.dynamicType

上面的形式中， expression 不能是某type的名字（當然了，如果我都知道它的名字了還需要動態來獲取它嗎）。動態型別表達式會回傳"執行時"某個instance的type, 具體請看下面的列子：

```
class SomeBaseClass {
    class func printClassName () {
        println ("SomeBaseClass")
    }
}
class SomeSubClass: SomeBaseClass {
    override class func printClassName () {
        println ("SomeSubClass")
    }
}
let someInstance: SomeBaseClass = SomeSubClass ()

// someInstance is of type SomeBaseClass at compile time, but
// someInstance is of type SomeSubClass at runtime
someInstance.dynamicType.printClassName ()
// prints "SomeSubClass"
```

> 動態型別表達式語法
> 動態型別表達式 → 後綴表達式 **. dynamicType**

下標腳本表達式（Subscript Expression）

下標腳本表達式提供了通過下標腳本存取getter/setter 的方法。它的形式是：

> `expression` [`index expressions`]

可以通過下標腳本表達式通過getter獲取某個值，或者通過setter賦予某個值.

關於subscript的宣告，請參見： Protocol Subscript Declaration.

> 附屬腳本表達式語法
> 附屬腳本表達式 → 後綴表達式 **[** 表達式列表 **]**

強制取值表達式（Forced-Value Expression）

強制取值表達式用來獲取某個目標表達式的值（該目標表達式的值必須不是nil ）。它的形式如下：

> `expression` !

如果該表達式的值不是nil, 則回傳對應的值。 否則，拋出執行時錯誤（runtime error）。

> 強制取值(Forced Value)語法
> 強制取值*(Forced Value)*表達式 → 後綴表達式 **!**

可選鏈表達式（Optional-Chaining Expression）

可選鏈表達式由目標表達式 + '?' 組成，形式如下：

> `expression` ?

後綴'?' 回傳目標表達式的值，把它做為可選的參數傳遞給後續的表達式

如果某個後綴表達式包含了可選鏈表達式，那麼它的執行過程就比較特殊： 首先先判斷該可選鏈表達式的值，如果是 nil, 整個後綴表達式都回傳 nil, 如果該可選鏈的值不是nil, 則正常回傳該後綴表達式的值（依次執行它的各個子表達式）。在這兩種情況下，該後綴表達式仍然是一個optional type（In either case, the value of the postfix expression is still of an optional type）

如果某個"後綴表達式"的"子表達式"中包含了"可選鏈表達式"，那麼只有最外層的表達式回傳的才是一個optional type. 例如，在下面的範例中， 如果c 不是nil, 那麼 c?.property.performAction（） 這句程式碼在執行時，就會先獲得c 的property方法，然後呼叫 performAction（）方法。 然後對於 "c?.property.performAction（）" 這個整體，它的回傳值是一個optional type.

```
var c: SomeClass?
var result: Bool? = c?.property.performAction ()
```

如果不使用可選鏈表達式，那麼 上面範例的程式碼跟下面範例等價：

```
if let unwrappedC = c {
    result = unwrappedC.property.performAction ()
}
```

> 可選鏈表達式語法
> 可選鏈表達式 → 後綴表達式 **?**

> 翻譯：coverxit 校對：numbbbbb, coverxit, stanzhai

語句

本頁包含內容：

- 迴圈語句
- 分支語句
- 帶標簽的語句
- 控制傳遞語句

在 Swift 中，有兩種型別的語句：簡單語句和控制流程語句。簡單語句是最常見的，用於建構表達式和宣告。控制流程語句則用於控制程式執行的流程，Swift 中有三種型別的控制流程語句：迴圈語句、分支語句和控制傳遞語句。

迴圈語句用於重複執行程式碼區塊；分支語句用於執行滿足特定條件的程式碼區塊；控制傳遞語句則用於修改程式碼的執行順序。在稍後的敘述中，將會詳細地介紹每一種型別的控制流程語句。

是否將分號（`;`）添加到語句的結尾處是可選的。但若要在同一行內寫多條獨立語句，請務必使用分號。

> 語句語法
> 語句 → 表達式 ; 可選
> 語句 → 宣告 ; 可選
> 語句 → 迴圈語句 ; 可選
> 語句 → 分支語句 ; 可選
> 語句 → 標記語句*(Labeled Statement)*
> 語句 → 控制轉移語句 ; 可選
> 多條語句*(Statements)* → 語句 多條語句*(Statements)* 可選

迴圈語句

取決於特定的迴圈條件，迴圈語句允許重複執行程式碼區塊。Swift 提供四種型別的迴圈語句：`for` 語句、`for-in` 語句、`while` 語句和 `do-while` 語句。

通過 `break` 語句和 `continue` 語句可以改變迴圈語句的控制流程。有關這兩條語句，詳情參見 Break 語句和 Continue 語句。

> 迴圈語句語法
> 迴圈語句 → *for*語句
> 迴圈語句 → *for-in*語句
> 迴圈語句 → *while*語句
> 迴圈語句 → *do-while*語句

For 語句

`for` 語句允許在重複執行程式碼區塊的同時，遞增一個計數器。

`for` 語句的形式如下：

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

initialzation、*condition* 和 *increment* 之間的分號，以及包圍迴圈 *statements* 的大括號都是不可省略的。

`for` 語句的執行流程如下：

1. *initialzation* 只會被執行一次，通常用於宣告和初始化在接下來的迴圈中需要使用的變數。
2. 計算 *condition* 表達式： 如果為 `true`， *statements* 將會被執行，然後轉到第3步。如果為 `false`， *statements* 和 *increment* 都不會被執行， `for` 至此執行完畢。
3. 計算 *increment* 表達式，然後轉到第2步。

定義在 *initialzation* 中的變數僅在 `for` 語句的作用域以內有效。*condition* 表達式的值的型別必須遵循 `LogicValue` 協定。

> For 迴圈語法
> *for*語句 → **for** *for*初始條件 可選 **;** 表達式 可選 **;** 表達式 可選 程式碼區塊
> *for*語句 → **for (** *for*初始條件 可選 **;** 表達式 可選 **;** 表達式 可選 **)** 程式碼區塊
> *for*初始條件 → 變數宣告 | 表達式列表

For-In 語句

`for-in` 語句允許在重複執行程式碼區塊的同時，迭代集合（或遵循 `Sequence` 協定的任意型別）中的每一項。

`for-in` 語句的形式如下：

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

`for-in` 語句在迴圈開始前會呼叫 *collection* 表達式的 `generate` 方法來獲取一個生成器型別（這是一個遵循 `Generator` 協定的型別）的值。接下來迴圈開始，呼叫 *collection* 表達式的 `next` 方法。如果其回傳值不是 `None`， 它將會被賦給 *item*，然後執行 *statements*，執行完畢後回到迴圈開始處；否則，將不會賦值給 *item* 也不會執行 *statements*， `for-in` 至此執行完畢。

> For-In 迴圈語法
> *for-in*語句 → **for** 模式 **in** 表達式 程式碼區塊

While 語句

`while` 語句允許重複執行程式碼區塊。

`while` 語句的形式如下：

```
while condition {
statements
}
```

`while` 語句的執行流程如下：

1. 計算 *condition* 表達式： 如果為真 `true`， 轉到第2步。如果為 `false`， `while` 至此執行完畢。
2. 執行 *statements*， 然後轉到第1步。

由於 *condition* 的值在 *statements* 執行前就已計算出，因此 `while` 語句中的 *statements* 可能會被執行若干次，也可能不會被執行。

condition 表達式的值的型別必須遵循 `LogicValue` 協定。同時，*condition* 表達式也可以使用可選綁定，詳情參見可選綁定。

> While 迴圈語法
> *while*語句 → **while** *while*條件 程式碼區塊
> *while*條件 → 表達式 | 宣告

Do-While 語句

`do-while` 語句允許程式碼區塊被執行一次或多次。

`do-while` 語句的形式如下：

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

`do-while` 語句的執行流程如下：

1. 執行 *statements*，然後轉到第2步。
2. 計算 *condition* 表達式： 如果為 `true`，轉到第1步。如果為 `false`， `do-while` 至此執行完畢。

由於 *condition* 表達式的值是在 *statements* 執行後才計算出，因此 `do-while` 語句中的 *statements* 至少會被執行一次。

condition 表達式的值的型別必須遵循 `LogicValue` 協定。同時，*condition* 表達式也可以使用可選綁定，詳情參見可選綁定。

> Do-While 迴圈語法
> *do-while*語句 → **do** 程式碼區塊 **while** *while*條件

分支語句

取決於一個或者多個條件的值，分支語句允許程式執行指定部分的程式碼。顯然，分支語句中條件的值將會決定如何分支以及執行哪一塊程式碼。Swift 提供兩種型別的分支語句： `if` 語句和 `switch` 語句。

`switch` 語句中的控制流程可以用 `break` 語句修改，詳情請見Break 語句。

> 分支語句語法
> 分支語句 → *if*語句
> 分支語句 → *switch*語句

If 語句

取決於一個或多個條件的值， `if` 語句將決定執行哪一塊程式碼。

`if` 語句有兩種標準形式，在這兩種形式裡都必須有大括號。

第一種形式是當且僅當條件為真時執行程式碼，像下面這樣：

```
if condition {
statements
}
```

第二種形式是在第一種形式的基礎上添加 *else* 語句，當只有一個 else 語句時，像下面這樣：

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

同時，else 語句也可包含 `if` 語句，從而形成一條鏈來測試更多的條件，像下面這樣：

```
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` 語句中條件的值的型別必須遵循 `LogicValue` 協定。同時，條件也可以使用可選綁定，詳情參見可選綁定。

> If語句語法
> *if*語句 → **if** *if*條件 程式碼區塊 *else*子句*(Clause)* 可選
> *if*條件 → 表達式 | 宣告
> *else*子句*(Clause)* → **else** 程式碼區塊 | **else** *if*語句

Switch 語句

取決於 `switch` 語句的控制表達式（*control expression*）， `switch` 語句將決定執行哪一塊程式碼。

`switch` 語句的形式如下：

> 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` 語句的控制表達式（*control expression*）會首先被計算，然後與每一個 case 的模式（pattern）進行匹配。如果匹配成功，程式將會執行對應的 case 分支裡的 *statements*。另外，每一個 case 分支都不能為空，也就是說在每一個 case 分支中至少有一條語句。如果你不想在匹配到的 case 分支中執行程式碼，只需在該分支裡寫一條 `break` 語句即可。

可以用作控制表達式的值是十分靈活的，除了純量型別(scalar types，如 `Int` 、 `Character`)外，你可以使用任何型別的值，包括浮點數、字串、元組、自定義類別的實例和可選（optional）型別，甚至是列舉型別中的成員值和指定的範圍(range)等。關於在 `switch` 語句中使用這些型別，詳情參見控制流程一章的 Switch。

你可以在模式後面添加一個起保護作用的表達式(guard expression)。起保護作用的表達式是這樣構成的：關鍵字 `where` 後面跟著一個作為額外測試條件的表達式。因此，當且僅當控制表達式匹配一個*case*的某個模式且起保護作用的表達式為真時，對應 case 分支中的 *statements* 才會被執行。在下面的範例中，控制表達式只會匹配含兩個相等元素的元組，如 `(1, 1)` ：

```
case let (x, y) where x == y:
```

正如上面這個範例，也可以在模式中使用 `let` （或 `var` ）語句來綁定常數（或變數）。這些常數（或變數）可以在其對應的起保護作用的表達式和其對應的*case*塊裡的程式碼中參考。但是，如果 case 中有多個模式匹配控制表達式，那麼這些模式都不能綁定常數（或變數）。

`switch` 語句也可以包含預設（ `default` ）分支，只有其它 case 分支都無法匹配控制表達式時，預設分支中的程式碼才會被執行。一個 `switch` 語句只能有一個預設分支，而且必須在 `switch` 語句的最後面。

儘管模式匹配操作實際的執行順序，特別是模式的計算順序是不可知的，但是 Swift 規定 `switch` 語句中的模式匹配的順序和書寫源程式碼的順序保持一致。因此，當多個模式含有相同的值且能夠匹配控制表達式時，程式只會執行源程式碼中第一個匹配的 case 分支中的程式碼。

Switch 語句必須是完備的

在 Swift 中，`switch` 語句中控制表達式的每一個可能的值都必須至少有一個 case 分支與之對應。在某些情況下（例如，表達式的型別是 `Int`），你可以使用預設塊滿足該要求。

不存在隱式的貫穿(fall through)

當匹配的 case 分支中的程式碼執行完畢後，程式會終止 `switch` 語句，而不會繼續執行下一個 case 分支。這就意味著，如果你想執行下一個 case 分支，需要顯式地在你需要的 case 分支裡使用 `fallthrough` 語句。關於 `fallthrough` 語句的更多資訊，詳情參見 Fallthrough 語句。

> Switch語句語法
> *switch*語句 → **switch** 表達式 **{** *SwitchCase*列表 可選 **}**
> *SwitchCase*列表 → *SwitchCase* *SwitchCase*列表 可選
> *SwitchCase* → *case*標簽 多條語句*(Statements)* | *default*標簽 多條語句*(Statements)*
> *SwitchCase* → *case*標簽 **:** | *default*標簽 **:**
> *case*標簽 → **case** *case*項列表 **:**
> *case*項列表 → 模式 *guard-clause* 可選 | 模式 *guard-clause* 可選 **,** *case*項列表
> *default*標簽 → **default :**
> *guard-clause* → **where** *guard-expression*
> *guard-expression* → 表達式

帶標簽的語句

你可以在迴圈語句或 `switch` 語句前面加上標簽，它由標簽名和緊隨其後的冒號(:)組成。在 `break` 和 `continue` 後面跟上標簽名可以顯式地在迴圈語句或 `switch` 語句中更改控制流程，把控制權傳遞給指定標簽標記的語句。關於這兩條語句用法，詳情參見 Break 語句和 Continue 語句。

標簽的作用域是該標簽所標記的語句之後的所有語句。你可以不使用帶標簽的語句，但只要使用它，標簽名就必唯一。

關於使用帶標簽的語句的範例，詳情參見控制流程一章的帶標簽的語句。

> 標記語句語法
> 標記語句*(Labeled Statement)* → 語句標簽 迴圈語句 | 語句標簽 *switch*語句
> 語句標簽 → 標簽名稱 **:**
> 標簽名稱 → 識別符號

控制傳遞語句

通過無條件地把控制權從一片程式碼傳遞到另一片程式碼，控制傳遞語句能夠改變程式碼執行的順序。Swift 提供四種型別的控制傳遞語句：`break` 語句、 `continue` 語句、 `fallthrough` 語句和 `return` 語句。

> 控制傳遞語句(Control Transfer Statement) 語法
> 控制傳遞語句 → *break*語句
> 控制傳遞語句 → *continue*語句
> 控制傳遞語句 → *fallthrough*語句
> 控制傳遞語句 → *return*語句

Break 語句

`break` 語句用於終止迴圈或 `switch` 語句的執行。使用 `break` 語句時，可以只寫 `break` 這個關鍵詞，也可以在 `break` 後面跟上標簽名（label name），像下面這樣：

> break
> break `label name`

當 `break` 語句後面帶標簽名時，可用於終止由這個標簽標記的迴圈或 `switch` 語句的執行。

而當只寫 `break` 時，則會終止 `switch` 語句或上下文中包含 `break` 語句的最內層迴圈的執行。

在這兩種情況下，控制權都會被傳遞給迴圈或 `switch` 語句外面的第一行語句。

關於使用 `break` 語句的範例，詳情參見控制流程一章的 Break 和帶標簽的語句。

> Break 語句語法
> *break*語句 → **break** 標簽名稱 可選

Continue 語句

`continue` 語句用於終止迴圈中當前迭代的執行，但不會終止該迴圈的執行。使用 `continue` 語句時，可以只寫 `continue` 這個關鍵詞，也可以在 `continue` 後面跟上標簽名（label name），像下面這樣：

```
continue
continue label name
```

當 `continue` 語句後面帶標簽名時，可用於終止由這個標簽標記的迴圈中當前迭代的執行。

而當只寫 `break` 時，可用於終止上下文中包含 `continue` 語句的最內層迴圈中當前迭代的執行。

在這兩種情況下，控制權都會被傳遞給迴圈外面的第一行語句。

在 `for` 語句中， `continue` 語句執行後，*increment* 表達式還是會被計算，這是因為每次迴圈執行完畢後 *increment* 表達式都會被計算。

關於使用 `continue` 語句的範例，詳情參見控制流程一章的 Continue 和帶標簽的語句。

> Continue 語句語法
> *continue*語句 → **continue** 標簽名稱 可選

Fallthrough 語句

`fallthrough` 語句用於在 `switch` 語句中傳遞控制權。 `fallthrough` 語句會把控制權從 `switch` 語句中的一個 case 傳遞給下一個 case 。這種傳遞是無條件的，即使下一個 case 的模式與 `switch` 語句的控制表達式的值不匹配。

`fallthrough` 語句可出現在 `switch` 語句中的任意 case 裡，但不能出現在最後一個 case 分支中。同時， `fallthrough` 語句也不能把控制權傳遞給使用了可選綁定的 case 分支。

關於在 `switch` 語句中使用 `fallthrough` 語句的範例，詳情參見控制流程一章的控制傳遞語句。

> Fallthrough 語句語法
> *fallthrough*語句 → **fallthrough**

Return 語句

`return` 語句用於在函式或方法的實作中將控制權傳遞給呼叫者，接著程式將會從呼叫者的位置繼續向下執行。

使用 `return` 語句時，可以只寫 `return` 這個關鍵詞，也可以在 `return` 後面跟上表達式，像下面這樣：

```
return
return expression
```

當 `return` 語句後面帶表達式時，表達式的值將會回傳給呼叫者。如果表達式值的型別與呼叫者期望的型別不匹配，Swift 則

會在回傳表達式的值之前將表達式值的型別轉換為呼叫者期望的型別。

而當只寫 `return` 時，僅僅是將控制權從該函式或方法傳遞給呼叫者，而不回傳一個值。（這就是說，該函式或方法的回傳型別為 `Void` 或 `()`）

> Return 語句語法
> *return*語句 → **return** 表達式 可選

> 翻譯：marsprince 校對：numbbbbb, stanzhai

宣告

本頁包含內容：

一條宣告可以在你的程式裡引入新的名字和建構。舉例來說，你可以使用宣告來引入函式和方法，變數和常數，或者來定義新的命名好的列舉，結構，類別和協定型別。你也可以使用一條宣告來延長一個已經存在的命名好的型別的行為。或者在你的 程式裡引入在其他地方宣告的符號。

在swift中，大多數宣告在某種意義上講也是執行或同事宣告它們的初始化定義。這意味著，因為協定和它們的成員不匹配，大多數協定成員需要單獨的宣告。為了方便起見，也因為這些區別在swift裡不是很重要，宣告語句同時包含了宣告和定義。

> 宣告語法
> 宣告 → 導入宣告
> 宣告 → 常數宣告
> 宣告 → 變數宣告
> 宣告 → 型別別名宣告
> 宣告 → 函式宣告
> 宣告 → 列舉宣告
> 宣告 → 結構宣告
> 宣告 → 類別宣告
> 宣告 → 協定宣告
> 宣告 → 建構器宣告
> 宣告 → 析構器宣告
> 宣告 → 擴展宣告
> 宣告 → 附屬腳本宣告
> 宣告 → 運算子宣告
> 宣告*(Declarations)*列表 → 宣告 宣告*(Declarations)*列表 可選
> 宣告描述符*(Specifiers)*列表 → 宣告描述符*(Specifier)* 宣告描述符*(Specifiers)*列表 可選
> 宣告描述符*(Specifier)* → **class** | **mutating** | **nonmutating** | **override** | **static** | **unowned** | **unowned(safe)** | **unowned(unsafe)** | **weak**

模塊範圍

模塊範圍定義了對模塊中其他原始碼可見的程式碼。（注：待改進）在swift的原始碼中，最高級別的程式碼由零個或多個語句， 宣告和表達組成。變數，常數和其他的宣告語句在一個原始碼的最頂級被宣告，使得它們對同一模塊中的每個原始碼都是可見的。

> 頂級(Top Level) 宣告語法
> 頂級宣告 → 多條語句*(Statements)* 可選

程式碼區塊

程式碼區塊用來將一些宣告和控制結構的語句組織在一起。它有如下的形式：

```
{
statements
}
```

程式碼區塊中的語句包括宣告，表達式和各種其他型別的語句，它們按照在源碼中的出現順序被依次執行。

> 程式碼區塊語法
> 程式碼區塊 → **{** 多條語句*(Statements)* 可選 **}**

引入宣告

引入宣告使你可以使用在其他文件中宣告的內容。引入語句的基本形式是引入整個程式碼模塊；它由import關鍵字開始，後面 緊跟一個模塊名：

```
import module
```

你可以提供更多的細節來限制引入的符號，如宣告一個特殊的子模塊或者在一個模塊或子模塊中做特殊的宣告。（待改進）當你使用了這些細節後，在當前的程式彙總只有引入的符號是可用的（並不是宣告的整個模塊）。

```
import import kind module.symbol name
import module.submodule
```

> 導入(Import)宣告語法
> 導入宣告 → 特性*(Attributes)*列表 可選 **import** 導入型別 可選 導入路徑
> 導入型別 → **typealias** | **struct** | **class** | **enum** | **protocol** | **var** | **func**
> 導入路徑 → 導入路徑識別符號 | 導入路徑識別符號 **.** 導入路徑
> 導入路徑識別符號 → 識別符號 | 運算子

常數宣告

常數宣告可以在你的程式裡命名一個常數。常數以關鍵詞let來宣告，遵循如下的格式:

```
let constant name: type = expression
```

當常數的值被給定後，常數就將常數名稱和表達式初始值不變的結合在了一起，而且不能更改。 這意味著如果常數以類別的形式被初始化，類別本身的內容是可以改變的，但是常數和類別之間的結合關係是不能改變的。 當一個常數被宣告為全域變數，它必須被給定一個初始值。當一個常數在類別或者結構中被宣告時，它被認為是一個常數 屬性。常數並不是可計算的屬性，因此不包含getters和setters。（譯者注：getters和setters不知道怎麼翻譯，待改進）

如果常數名是一個元祖形式，元祖中的每一項初始化表達式中都要有對應的值

```
let (firstNumber, secondNumber) = (10, 42)
```

在上例中，firstNumber是一個值為10的常數，secnodeName是一個值為42的常數。所有常數都可以獨立的使用：

```
println("The first number is \(firstNumber).")
// prints "The first number is 10."
println("The second number is \(secondNumber).")
// prints "The second number is 42."
```

型別註解（:type）在常數宣告中是一個可選項，它可以用來描述在型別推斷（type inference）中找到的型別。

宣告一個靜態常數要使用關鍵字static。靜態屬性在型別屬性（type propetries）中有介紹。

如果還想獲得更多關於常數的資訊或者想在使用中獲得幫助，請查看常數和變數（constants and variables）, 儲存屬性（stored properties）等節。

> 常數宣告語法
> 常數宣告 → 特性*(Attributes)*列表 可選 宣告描述符*(Specifiers)*列表 可選 **let** 模式建構器列表
> 模式建構器列表 → 模式建構器 | 模式建構器 , 模式建構器列表
> 模式建構器 → 模式 建構器 可選
> 建構器 → **=** 表達式

變數宣告

變數宣告可以在你的程式裡宣告一個變數，它以關鍵字var來宣告。根據宣告變數型別和值的不同，如儲存和計算 變數和屬性，儲存變數和屬性監視，和靜態變數屬性，有著不同的宣告形式。（待改進） 所使用的宣告形式取決於變數所宣告的範圍和你打算宣告的變數型別。

> 注意：
> 你也可以在協定宣告的上下文宣告屬性，詳情參見型別屬性宣告。

儲存型變數和儲存型屬性

下面的形式宣告了一個儲存型變數或儲存型變數屬性

> var `variable name` : `type` = `expression`

你可以在全域，函式內，或者在類別和結構的宣告(context)中使用這種形式來宣告一個變數。當變數以這種形式 在全域或者一個函式內被宣告時，它代表一個儲存型變數。當它在類別或者結構中被宣告時，它代表一個儲存型變數屬性。

建構器表達式可以被

和常數宣告相比，如果變數名是一個元祖型別，元祖的每一項的名字都要和初始化表達式一致。

正如名字一樣，儲存型變數的值或儲存型變數屬性儲存在內存中。

計算型變數和計算型屬性

如下形式宣告一個一個儲存型變數或儲存型屬性：

> var `variable name` : `type` {
> get {
> `statements`

```
}
set( setter name ) {
statements
}
}
```

你可以在全域，函式體內或者類別，結構，列舉，擴展宣告的上下文中使用這種形式的宣告。 當變數以這種形式在全域或者一個函式內被宣告時，它代表一個計算型變數。當它在類別，結構，列舉，擴展宣告的上下文 中中被宣告時，它代表一個計算型變數屬性。

getter用來讀取變數值，setter用來寫入變數值。setter子句是可選擇的，只有getter是必需的，你可以將這些語句 都省略，只是簡單的直接回傳請求值，正如在唯讀計算屬性(read-only computed properites)中描述的那樣。 但是如果你提供了一個setter語句，你也必需提供一個getter語句。

setter的名字和圓括號內的語句是可選的。如果你寫了一個setter名，它就會作為setter的參數被使用。如果你不寫setter名，setter的初始名為newValue，正如在seter宣告速記(shorthand setter declaration)中提到的那樣。

不像儲存型變數和儲存型屬性那樣，計算型屬性和計算型變數的值不儲存在內存中。

獲得更多資訊，查看更多關於計算型屬性的範例，請查看計算型屬性(computed properties)一節。

儲存型變數監視器和屬性監視器

你可以用willset和didset監視器來宣告一個儲存型變數或屬性。一個包含監視器的儲存型變數或屬性按如下的形式宣告：

```
var variable name： type = expression {
willSet(setter name) {
statements
}
didSet( setter name ) {
statements
}
}
```

你可以在全域，函式體內或者類別，結構，列舉，擴展宣告的上下文中使用這種形式的宣告。 當變數以這種形式在全域或者一個函式內被宣告時，監視器代表一個儲存型變數監視器；當它在類別，結構，列舉，擴展宣告的上下文中被宣告時，監視器代表屬性監視器。

你可以為適合的監視器添加任何儲存型屬性。你也可以通過重寫子類別屬性的方式為適合的監視器添加任何繼承的屬性 (無論是儲存型還是計算型的)，參見重寫屬性監視器(overriding properyt observers)。

初始化表達式在類別或者結構的宣告中是可選的，但是在其他地方是必需的。無論在什麼地方宣告， 所有包含監視器的變數宣告都必須有型別註解(type annotation)。

當變數或屬性的值被改變時，willset和didset監視器提供了一個監視方法（適當的回應）。 監視器不會在變數或屬性第一次初始化時不會被執行，它們只有在值被外部初始化語句改變時才會被執行。

willset監視器只有在變數或屬性值被改變之前執行。新的值作為一個常數經過過willset監視器，因此不可以在 willset語句中改變它。didset監視器在變數或屬性值被改變後立即執行。和willset監視器相反，為了以防止你仍然 需要獲得舊的資料，舊變數值或者屬性會經過didset監視器。這意味著，如果你在變數或屬性自身的didiset監視器語句 中設置了一個值，你設置的新值會取代剛剛在willset監視器中經過的那個值。

在willset和didset語句中，setter名和圓括號的語句是可選的。如果你寫了一個setter名，它就會作為willset和didset的參數被使用。如果你不寫setter名， willset監視器初始名為newvalue，didset監視器初始名為oldvalue。

當你提供一個willset語句時，didset語句是可選的。同樣的，在你提供了一個didset語句時，willset語句是可選的。

獲得更多資訊，查看如何使用屬性監視器的範例，請查看屬性監視器(prpperty observers)一節。

類別和靜態變數屬性

class關鍵字用來宣告類別的計算型屬性。static關鍵字用來宣告類別的靜態變數屬性。類別和靜態變數在型別屬性(type properties)中有詳細討論。

> 變數宣告語法
> 變數宣告 → 變數宣告頭*(Head)* 模式建構器列表
> 變數宣告 → 變數宣告頭*(Head)* 變數名 型別注解 程式碼區塊
> 變數宣告 → 變數宣告頭*(Head)* 變數名 型別注解 *getter-setter*塊
> 變數宣告 → 變數宣告頭*(Head)* 變數名 型別注解 *getter-setter*關鍵字*(Keyword)*塊
> 變數宣告 → 變數宣告頭*(Head)* 變數名 型別注解 建構器 可選 *willSet-didSet*程式碼區塊
> 變數宣告頭*(Head)* → 特性*(Attributes)*列表 可選 宣告描述符*(Specifiers)*列表 可選 **var**
> 變數名稱 → 識別符號
> *getter-setter*塊 → **{** *getter*子句 *setter*子句 可選 **}**
> *getter-setter*塊 → **{** *setter*子句 *getter*子句 **}**
> *getter*子句 → 特性*(Attributes)*列表 可選 **get** 程式碼區塊
> *setter*子句 → 特性*(Attributes)*列表 可選 **set** *setter*名稱 可選 程式碼區塊
> *setter*名稱 → **(** 識別符號 **)**
> *getter-setter*關鍵字*(Keyword)*塊 → **{** *getter*關鍵字*(Keyword)*子句 *setter*關鍵字*(Keyword)*子句 可選 **}**
> *getter-setter*關鍵字*(Keyword)*塊 → **{** *setter*關鍵字*(Keyword)*子句 *getter*關鍵字*(Keyword)*子句 **}**
> *getter*關鍵字*(Keyword)*子句 → 特性*(Attributes)*列表 可選 **get**
> *setter*關鍵字*(Keyword)*子句 → 特性*(Attributes)*列表 可選 **set**
> *willSet-didSet*程式碼區塊 → **{** *willSet*子句 *didSet*子句 可選 **}**
> *willSet-didSet*程式碼區塊 → **{** *didSet*子句 *willSet*子句 **}**
> *willSet*子句 → 特性*(Attributes)*列表 可選 **willSet** *setter*名稱 可選 程式碼區塊
> *didSet*子句 → 特性*(Attributes)*列表 可選 **didSet** *setter*名稱 可選 程式碼區塊

型別的別名宣告

型別別名的宣告可以在你的程式裡為一個已存在的型別宣告一個別名。型別的別名宣告以關鍵字typealias開始，遵循如下的形式：

```
typealias name = existing type
```

當一個型別被別名被宣告後，你可以在你程式的任何地方使用別名來代替已存在的型別。已存在的型別可以是已經被命名的型別或者是混合型別。型別的別名不產生新的型別，它只是簡單的和已存在的型別做名稱替換。

查看更多Protocol Associated Type Declaration.

> 型別別名宣告語法
> 型別別名宣告 → 型別別名頭*(Head)* 型別別名賦值
> 型別別名頭*(Head)* → **typealias** 型別別名名稱
> 型別別名名稱 → 識別符號
> 型別別名賦值 → **=** 型別

函式宣告

你可以使用函式宣告在你的程式裡引入新的函式。函式可以在類別的上下文，結構，列舉，或者作為方法的協定中被宣告。函式宣告使用關鍵字func，遵循如下的形式：

> func function name (parameters) -> return type {
> statements
> }

如果函式不回傳任何值，回傳型別可以被忽略，如下所示：

> func function name (parameters) {
> statements
> }

每個參數的型別都要標明，它們不能被推斷出來。初始時函式的參數是常值。在這些參數前面添加var使它們成為變數， 作用域內任何對變數的改變只在函式體內有效，或者用inout使的這些改變可以在呼叫域內生效。 更多關於in-out參數的討論，參見in-out參數(in-out parameters)

函式可以使用元組型別作為回傳值來回傳多個變數。

函式定義可以出現在另一個函式宣告內。這種函式被稱作nested函式。更多關於nested函式的討論，參見nestde functions。

參數名

函式的參數是一個以逗號分隔的列表 。函式呼叫是的變數順序必須和函式宣告時的參數順序一致。 最簡單的參數列表有著如下的形式：

> parameter name : parameter type

對於函式參數來講，參數名在函式體內被使用，而不是在函式呼叫時使用。對於方法參數，參數名在函式體內被使用， 同時也在方法被呼叫時作為標簽被使用。該方法的第一個參數名僅僅在函式體內被使用，就像函式的參數一樣，舉例來講：

```
func f(x: Int, y: String) -> String {
    return y + String(x)
}
f(7, "hello")  // x and y have no name
```

```
class C {
    func f(x: Int, y: String) -> String {
       return y + String(x)
    }
}
let c = C()
c.f(7, y: "hello")  // x沒有名稱，y有名稱
```

你可以按如下的形式，重寫參數名被使用的過程：

> external parameter name local parameter name : parameter type
> # parameter name : parameter type
> _ local parameter name : parameter type

在本地參數前命名的第二名稱(second name)使得參數有一個擴展名。且不同於本地的參數名。 擴展參數名在函式被呼叫時必須被使用。對應的參數在方法或函式被呼叫時必須有擴展名 。

在參數名前所寫的雜湊符號(#)代表著這個參數名可以同時作為外部或本體參數名來使用。等同於書寫兩次本地參數名。 在函式或方法呼叫時，與其對應的語句必須包含這個名字。

本地參數名前的強調字元(_)使參數在函式被呼叫時沒有名稱。在函式或方法呼叫時，與其對應的語句必須沒有名字。

特殊型別的參數

參數可以被忽略，值可以是變化的，並且提供一個初始值，這種方法有著如下的形式：

> _ : <#parameter type#.
> `parameter name`: `parameter type`...
> `parameter name`: `parameter type` = `default argument value`

以強調符(_)命名的參數明確的在函式體內不能被存取。

一個以基礎型別名的參數，如果緊跟著三個點(...)，被理解為是可變參數。一個函式至多可以擁有一個可變參數， 且必須是最後一個參數。可變參數被作為該基本型別名的陣列來看待。舉例來講，可變參數int...被看做是int[]。 查看可變參數的使用範例，詳見可變參數(variadic parameters)一節。

在參數的型別後面有一個以等號(=)連接的表達式，這樣的參數被看做有著給定表達式的初試值。如果參數在函式 呼叫時被省略了，就會使用初始值。如果參數沒有勝率，那麼它在函式呼叫是必須有自己的名字.舉例來講， f()和f(x:7)都是只有一個變數x的函式的有效呼叫，但是f(7)是非法的，因為它提供了一個值而不是名稱。

特殊方法

以self修飾的列舉或結構方法必須以mutating關鍵字作為函式宣告頭。

子類別重寫的方法必須以override關鍵字作為函式宣告頭。不用override關鍵字重寫的方法，使用了override關鍵字 卻並沒有重寫父類別方法都會報錯。

和型別相關而不是和型別實例相關的方法必須在static宣告的結構以或列舉內，亦或是以class關鍵字定義的類別內。

柯裡化函式和方法

柯裡化函式或方法有著如下的形式：

> func `function name`(`parameters`)(`parameters`) -> `return type` {
> `statements`
> }

以這種形式定義的函式的回傳值是另一個函式。舉例來說，下面的兩個宣告時等價的:

```
func addTwoNumbers(a: Int)(b: Int) -> Int {
    return a + b
}
func addTwoNumbers(a: Int) -> (Int -> Int) {
    func addTheSecondNumber(b: Int) -> Int {
        return a + b
    }
    return addTheSecondNumber
}
```

```
addTwoNumbers(4)(5) // Returns 9
```

多級柯裡化應用如下

> 函式宣告語法
> 函式宣告 → 函式頭 函式名 泛型參數子句 可選 函式簽名*(Signature)* 函式體
> 函式頭 → 特性*(Attributes)*列表 可選 宣告描述符*(Specifiers)*列表 可選 **func**
> 函式名 → 識別符號 | 運算子
> 函式簽名*(Signature)* → *parameter-clauses* 函式結果 可選
> 函式結果 → **->** 特性*(Attributes)*列表 可選 型別

> 函式體 → 程式碼區塊
> *parameter-clauses* → 參數子句 *parameter-clauses* 可選
> 參數子句 → **(** **)** | **(** 參數列表 **...** 可選 **)**
> 參數列表 → 參數 | 參數 **,** 參數列表
> 參數 → **inout** 可選 **let** 可選 # 可選 參數名 本地參數名 可選 型別注解 預設參數子句 可選
> 參數 → **inout** 可選 **var** # 可選 參數名 本地參數名 可選 型別注解 預設參數子句 可選
> 參數 → 特性*(Attributes)*列表 可選 型別
> 參數名 → 識別符號 | _
> 本地參數名 → 識別符號 | _
> 預設參數子句 → **=** 表達式

列舉宣告

在你的程式裡使用列舉宣告來引入一個列舉型別。

列舉宣告有兩種基本的形式，使用關鍵字enum來宣告。列舉宣告體使用從零開始的變數——叫做列舉事件，和任意數量的宣告，包括計算型屬性，實例方法，靜態方法，建構器，型別別名，甚至其他列舉，結構，和類別。列舉宣告不能 包含析構器或者協定宣告。

不像類別或者結構。列舉型別並不提供隱式的初始建構器，所有建構器必須顯式的宣告。建構器可以委托列舉中的其他 建構器，但是建構過程僅當建構器將一個列舉時間完成後才全部完成。

和結構類似但是和類別不同，列舉是值型別：列舉實例在賦予變數或常數時，或者被函式呼叫時被複製。 更多關於值型別的資訊，參見結構和列舉都是值型別(Structures and Enumerations Are Value Types)一節。

你可以擴展列舉型別，正如在擴展名宣告(Extension Declaration)中討論的一樣。

任意事件型別的列舉

如下的形式宣告了一個包含任意型別列舉時間的列舉變數

```
enum enumeration name {
case enumeration case 1
case enumeration case 2 ( associated value types )
}
```

這種形式的列舉宣告在其他語言中有時被叫做可識別聯合(discrinminated)。

這種形式中，每一個事件塊由關鍵字case開始，後面緊接著一個或多個以逗號分隔的列舉事件。每一個事件名必須是 獨一無二的。每一個事件也可以指定它所儲存的指定型別的值，這些型別在關聯值型別的元祖裡被指定，立即書寫在事件 名後。獲得更多關於關聯值型別的資訊和範例，請查看關聯值(associated values)一節。

使用原始事件值的列舉

以下的形式宣告了一個包含相同基礎型別的列舉事件的列舉：

```
enum enumeration name : raw value type {
case enumeration case 1 = raw value 1
case enumeration case 2 = raw value 2
}
```

在這種形式中，每一個事件塊由case關鍵字開始，後面緊接著一個或多個以逗號分隔的列舉事件。和第一種形式的列舉 事件不同，這種形式的列舉事件包含一個同型別的基礎值，叫做原始值(raw value)。這些值的型別在原始值型別(raw value type)中被指定，必須是字面上的整數，浮點數，字元或者字串。

每一個事件必須有唯一的名字，必須有一個唯一的初始值。如果初始值型別被指定為int，則不必為事件顯式的指定值， 它們會隱式的被標為值0,1,2等。每一個沒有被賦值的Int型別時間會隱式的賦予一個初始值，它們是自動遞增的。

```
num ExampleEnum: Int {
    case A, B, C = 5, D
}
```

在上面的範例中，ExampleEnum.A的值是0，ExampleEnum.B的值是。因為ExampleEnum.C的值被顯式的設定為5，因此ExampleEnum.D的值會自動增長為6.

列舉事件的初始值可以呼叫方法roRaw獲得，如ExampleEnum.B.toRaw()。你也可以通過呼叫fromRaw方法來使用初始值找到 其對應的事件，並回傳一個可選的事件。查看更多資訊和獲取初始值型別事件的資訊，參閱初始值(raw values)。

獲得列舉事件

使用點(.)來參考列舉型別的事件，如 EnumerationType.EnumerationCase。當列舉型別可以上下文推斷出時，你可以 省略它(.仍然需要)，參照列舉語法(Enumeration Syntax)和顯式成員表達(Implicit Member Expression).

使用switch語句來檢驗列舉事件的值，正如使用switch語句匹配列舉值（Matching Enumeration Values with a Switch Statement)一節描述的那樣。

列舉型別是模式匹配(pattern-matched)的，和其相反的是switch語句case塊中列舉事件匹配，在列舉事件型別(Enumeration Case Pattern)中有描述。

> 列舉宣告語法
> 列舉宣告 → 特性*(Attributes)*列表 可選 聯合式列舉 | 特性*(Attributes)*列表 可選 原始值式列舉
> 聯合式列舉 → 列舉名 泛型參數子句 可選 **{** *union-style-enum-members* 可選 **}**
> *union-style-enum-members* → *union-style-enum-member union-style-enum-members* 可選
> *union-style-enum-member* → 宣告 | 聯合式*(Union Style)*的列舉*case*子句
> 聯合式*(Union Style)*的列舉*case*子句 → 特性*(Attributes)*列表 可選 **case** 聯合式*(Union Style)*的列舉*case*列表
> 聯合式*(Union Style)*的列舉*case*列表 → 聯合式*(Union Style)*的*case* | 聯合式*(Union Style)*的*case* **,** 聯合式*(Union Style)*的列舉*case*列表
> 聯合式*(Union Style)*的*case* → 列舉的*case*名 元組型別 可選
> 列舉名 → 識別符號
> 列舉的*case*名 → 識別符號
> 原始值式列舉 → 列舉名 泛型參數子句 可選 **:** 型別標識 **{** 原始值式列舉成員列表 可選 **}**
> 原始值式列舉成員列表 → 原始值式列舉成員 原始值式列舉成員列表 可選
> 原始值式列舉成員 → 宣告 | 原始值式列舉*case*子句
> 原始值式列舉*case*子句 → 特性*(Attributes)*列表 可選 **case** 原始值式列舉*case*列表
> 原始值式列舉*case*列表 → 原始值式列舉*case* | 原始值式列舉*case* **,** 原始值式列舉*case*列表
> 原始值式列舉*case* → 列舉的*case*名 原始值賦值 可選
> 原始值賦值 → **=** 字面量

結構宣告

使用結構宣告可以在你的程式裡引入一個結構型別。結構宣告使用struct關鍵字，遵循如下的形式：

> struct `structure name` : `adopted protocols` {
> `declarations`
> }

結構內包含零或多個宣告。這些宣告可以包括儲存型和計算型屬性，靜態屬性，實例方法，靜態方法，建構器， 型別別名，甚至其他結構，類別，和列舉宣告。結構宣告不能包含析構器或者協定宣告。詳細討論和包含多種結構 宣告的實例，參見類

別和結構一節。

結構可以包含任意數量的協定，但是不能繼承自類別，列舉或者其他結構。

有三種方法可以創建一個宣告過的結構實例：

-呼叫結構內宣告的建構器，參照建構器(initializers)一節。

—如果沒有宣告建構器，呼叫結構的逐個建構器，詳情參見Memberwise Initializers for Structure Types.

—如果沒有宣告析構器，結構的所有屬性都有初始值，呼叫結構的預設建構器，詳情參見預設建構器(Default Initializers).

結構的建構過程參見初始化(initiaization)一節。

結構實例屬性可以用點(.)來獲得，詳情參見獲得屬性(Accessing Properties)一節。

結構是值型別；結構的實例在被賦予變數或常數，被函式呼叫時被複製。獲得關於值型別更多資訊，參見 結構和列舉都是值型別(Structures and Enumerations Are Value Types)一節。

你可以使用擴展宣告來擴展結構型別的行為，參見擴展宣告(Extension Declaration).

> 結構宣告語法
> 結構宣告 → 特性*(Attributes)*列表 可選 **struct** 結構名稱 泛型參數子句 可選 型別繼承子句 可選 結構主體
> 結構名稱 → 識別符號
> 結構主體 → **{** 宣告*(Declarations)*列表 可選 **}**

類別宣告

你可以在你的程式中使用類別宣告來引入一個類別。類別宣告使用關鍵字class，遵循如下的形式：

```
class class name: superclass, adopted protocols {
declarations
}
```

一個類別內包含零或多個宣告。這些宣告可以包括儲存型和計算型屬性，實例方法，類別方法，建構器，單獨的析構器方法， 型別別名，甚至其他結構，類別，和列舉宣告。類別宣告不能包含協定宣告。詳細討論和包含多種類別宣告的實例，參見類別和 結構一節。

一個類別只能繼承一個父類別，超類別，但是可以包含任意數量的協定。這些超類別第一次在type-inheritance-clause出現，遵循任意協定。

正如在初始化宣告(Initializer Declaration)談及的那樣，類別可以有指定和方便的建構器。當你宣告任一中建構器時， 你可以使用requierd變數來標記建構器，要求任意子類別來重寫它。指定類別的建構器必須初始化類別所有的已宣告的屬性， 它必須在子類別建構器呼叫前被執行。

類別可以重寫屬性，方法和它的超類別的建構器。重寫的方法和屬性必須以override標注。

雖然超類別的屬性和方法宣告可以被當前類別繼承，但是超類別宣告的指定建構器卻不能。這意味著，如果當前類別重寫了超類別 的所有指定建構器，它就繼承了超類別的方便建構器。Swift的類別並不是繼承自一個全域基礎類別。

有兩種方法來創建已宣告的類別的實例：

- 呼叫類別的一個建構器，參見建構器(initializers)。
- 如果沒有宣告建構器，而且類別的所有屬性都被賦予了初始值，呼叫類別的預設建構器，參見預設建構器(default initializers).

類別實例屬性可以用點(.)來獲得，詳情參見獲得屬性(Accessing Properties)一節。

類別是參考型別；當被賦予常數或變數，函式呼叫時，類別的實例是被參考，而不是複製。獲得更多關於參考型別的資訊，結構和列舉都是值型別(Structures and Enumerations Are Value Types)一節。

你可以使用擴展宣告來擴展類別的行為，參見擴展宣告(Extension Declaration).

> 類別宣告語法
> 類別宣告 → 特性*(Attributes)*列表 可選 **class** 類別名 泛型參數子句 可選 型別繼承子句 可選 類別主體
> 類別名 → 識別符號
> 類別主體 → **{** 宣告*(Declarations)*列表 可選 **}**

協定宣告(translated by 小一)

一個協定宣告為你的程式引入一個命名了的協定型別。協定宣告使用 `protocol` 關鍵詞來進行宣告並有下面這樣的形式：

```
protocol protocol name：inherited protocols {
protocol member declarations
}
```

協定的主體包含零或多個協定成員宣告，這些成員描述了任何采用該協定必須滿足的一致性要求。特別的，一個協定可以宣告必須實作某些屬性、方法、初始化程式及下標腳本的一致性型別。協定也可以宣告專用種類別的型別別名，叫做關聯型別，它可以指定協定的不同宣告之間的關系。協定成員宣告會在下面的詳情裡進行討論。

協定型別可以從很多其它協定那繼承。當一個協定型別從其它協定那繼承的時候，來自其它協定的所有要求就集合了，而且從當前協定繼承的任何型別必須符合所有的這些要求。對於如何使用協定繼承的範例，查看協定繼承

> 注意：
> 你也可以使用協定合成型別集合多個協定的一致性要求，詳情參見協定合成型別和協定合成

你可以通過采用在型別的擴展宣告中的協定來為之前宣告的型別添加協定一致性。在擴展中你必須實作所有采用協定的要求。如果該型別已經實作了所有的要求，你可以讓這個擴展宣告的主題留空。

預設地，符合某一個協定的型別必須實作所有宣告在協定中的屬性、方法和下標腳本。也就是說，你可以用 `optional` 屬性標注這些協定成員宣告以指定它們的一致性型別實作是可選的。 `optional` 屬性僅僅可以用於使用 `objc` 屬性標記過的協定。這樣的結果就是僅僅類型別可以采用並符合包含可選成員要求的協定。更多關於如何使用 `optional` 屬性的資訊及如何存取可選協定成員的指導——比如當你不能肯定是否一致性的型別實作了它們——參見可選協定要求

為了限制協定的采用僅僅針對類型別，需要使用 `class_protocol` 屬性標記整個協定宣告。任意繼承自標記有 `class_protocol` 屬性協定的協定都可以智能地僅能被類型別采用。

> 注意：
> 如果協定已經用 `object` 屬性標記了， `class_protocol` 屬性就隱性地應用於該協定；沒有必要再明確地使用 `class_protocol` 屬性來標記該協定了。

協定是命名的型別，因此它們可以以另一個命名型別出現在你程式碼的所有地方，就像協定型別裡討論的那樣。然而你不能建構一個協定的實例，因為協定實際上不提供它們指定的要求的實作。

你可以使用協定來宣告一個類別的代理的方法或者應該實作的結構，就像委托(代理)模式描述的那樣。

> 協定(Protocol)宣告語法
> 協定宣告 → 特性*(Attributes)*列表 可選 **protocol** 協定名 型別繼承子句 可選 協定主體
> 協定名 → 識別符號
> 協定主體 → **{** 協定成員宣告*(Declarations)*列表 可選 **}**
> 協定成員宣告 → 協定屬性宣告

> 協定成員宣告 → 協定方法宣告
> 協定成員宣告 → 協定建構器宣告
> 協定成員宣告 → 協定附屬腳本宣告
> 協定成員宣告 → 協定關聯型別宣告
> 協定成員宣告*(Declarations)*列表 → 協定成員宣告 協定成員宣告*(Declarations)*列表 可選

協定屬性宣告

協定宣告了一致性型別必須在協定宣告的主體裡通過引入一個協定屬性宣告來實作一個屬性。協定屬性宣告有一種特殊的型別宣告形式：

> var `property name` : `type` { get set }

同其它協定成員宣告一樣，這些屬性宣告僅僅針對符合該協定的型別宣告了 `getter` 和 `setter` 要求。結果就是你不需要在協定裡它被宣告的地方實作 `getter` 和 `setter` 。

`getter` 和 `setter` 要求可以通過一致性型別以各種方式滿足。如果屬性宣告包含 `get` 和 `set` 關鍵詞，一致性型別就可以用可讀寫（實作了 `getter` 和 `setter` ）的儲存型變數屬性或計算型屬性，但是屬性不能以常數屬性或唯讀計算型屬性實作。如果屬性宣告僅僅包含 `get` 關鍵詞的話，它可以作為任意型別的屬性被實作。比如說實作了協定的屬性要求的一致性型別，參見 屬性要求

更多參見變數宣告

> 協定屬性宣告語法
> 協定屬性宣告 → 變數宣告頭*(Head)* 變數名 型別注解 *getter-setter*關鍵字*(Keyword)*塊

協定方法宣告

協定宣告了一致性型別必須在協定宣告的主體裡通過引入一個協定方法宣告來實作一個方法. 協定方法宣告和函式方法宣告有著相同的形式，包含如下兩條規則：它們不包括函式體，你不能在類別的宣告內為它們的 參數提供初始值.舉例來說，符合的型別執行協定必需的方法。參見必需方法一節。

使用關鍵字class可以在協定宣告中宣告一個類別或必需的靜態方法。執行這些方法的類別也用關鍵字class宣告。 相反的，執行這些方法的結構必須以關鍵字static宣告。如果你想使用擴展方法，在擴展類別時使用class關鍵字， 在擴展結構時使用static關鍵字。

更多請參閱函式宣告。

> 協定方法宣告語法
> 協定方法宣告 → 函式頭 函式名 泛型參數子句 可選 函式簽名*(Signature)*

協定建構器宣告

協定宣告了一致性型別必須在協定宣告的主體裡通過引入一個協定建構器宣告來實作一個建構器。協定建構器宣告 除了不包含建構器體外，和建構器宣告有著相同的形式，

更多請參閱建構器宣告。

> 協定建構器宣告語法
> 協定建構器宣告 → 建構器頭*(Head)* 泛型參數子句 可選 參數子句

協定下標腳本宣告

協定宣告了一致性型別必須在協定宣告的主體裡通過引入一個協定下標腳本宣告來實作一個下標腳本。協定屬性宣告 對下標腳本宣告有一個特殊的形式：

> subscript (`parameters`) -> `return type` { get set }

下標腳本宣告只為和協定一致的型別宣告了必需的最小數量的的getter和setter。如果下標腳本申明包含get和set關鍵字， 一致的型別也必須有一個getter和setter語句。如果下標腳本宣告值包含get關鍵字，一致的型別必須至少包含一個 getter語句，可以選擇是否包含setter語句。

更多參閱下標腳本宣告。

> 協定附屬腳本宣告語法
> 協定附屬腳本宣告 → 附屬腳本頭*(Head)* 附屬腳本結果*(Result)* *getter-setter*關鍵字*(Keyword)*塊

協定相關型別宣告

協定宣告相關型別使用關鍵字typealias。相關型別為作為協定宣告的一部分的型別提供了一個別名。相關型別和參數 語句中的型別參數很相似，但是它們在宣告的協定中包含self關鍵字。在這些語句中，self指代和協定一致的可能的型別。 獲得更多資訊和範例， 查看相關型別或型別別名宣告。

> 協定關聯型別宣告語法
> 協定關聯型別宣告 → 型別別名頭*(Head)* 型別繼承子句 可選 型別別名賦值 可選

建構器宣告

建構器宣告會為程式內的類別，結構或列舉引入建構器。建構器使用關鍵字Init來宣告，遵循兩條基本形式。

結構，列舉，類別可以有任意數量的建構器，但是類別的建構器的規則和行為是不一樣的。不像結構和列舉那樣，類別 有兩種結構，designed initializers 和convenience initializers，參見建構器一節。

如下的形式宣告了結構，列舉和類別的指定建構器：

```
init( parameters ) {
statements
}
```

類別的指定建構器將類別的所有屬性直接初始化。如果類別有超類別，它不能呼叫該類別的其他建構器,它只能呼叫超類別的一個 指定建構器。如果該類別從它的超類別處繼承了任何屬性，這些屬性在當前類別內被賦值或修飾時，必須帶哦用一個超類別的 指定建構器。

指定建構器可以在類別宣告的上下文中宣告，因此它不能用擴展宣告的方法加入一個類別中。

結構和列舉的建構器可以帶哦用其他的已宣告的建構器，來委托其中一個火全部進行初始化過程。

以關鍵字convenience來宣告一個類別的便利建構器：

```
convenience init( parameters ) {
statements
}
```

便利建構器可以將初始化過程委托給另一個便利建構器或類別的一個指定建構器。這意味著，類別的初始化過程必須 以一個將所有類別屬性完全初始化的指定建構器的呼叫作為結束。便利建構器不能呼叫超類別的建構器。

你可以使用requierd關鍵字，將便利建構器和指定建構器標記為每個子類別的建構器都必須擁有的。因為指定建構器 不被子類別繼承，它們必須被立即執行。當子類別直接執行所有超類別的指定建構器(或使用便利建構器重寫指定建構器)時， 必需的便利建構器可以被隱式的執行，亦可以被繼承。不像方法，下標腳本那樣，你不需要為這些重寫的建構器標注 overrride關鍵字。

查看更多關於不同宣告方法的建構器的範例，參閱建構過程一節。

> 建構器宣告語法
> 建構器宣告 → 建構器頭*(Head)* 泛型參數子句 可選 參數子句 建構器主體
> 建構器頭*(Head)* → 特性*(Attributes)*列表 可選 **convenience** 可選 **init**
> 建構器主體 → 程式碼區塊

析構宣告

析構宣告為類別宣告了一個析構器。析構器沒有參數，遵循如下的格式：

```
deinit {
statements
}
```

當類別沒有任何語句時將要被釋放時，析構器會自動的被呼叫。析構器在類別的宣告體內只能被宣告一次——但是不能在 類別的擴展宣告內，每個類別最多只能有一個。

子類別繼承了它的超類別的析構器，在子類別將要被釋放時隱式的呼叫。子類別在所有析構器被執行完畢前不會被釋放。

析構器不會被直接呼叫。

查看範例和如何在類別的宣告中使用析構器，參見析構過程一節。

> 析構器宣告語法
> 析構器宣告 → 特性*(Attributes)*列表 可選 **deinit** 程式碼區塊

擴展宣告

擴展宣告用於擴展一個現存的類別，結構，列舉的行為。擴展宣告以關鍵字extension開始，遵循如下的規則：

```
extension type : adopted protocols {
declarations
}
```

一個擴展宣告體包括零個或多個宣告。這些宣告可以包括計算型屬性，計算型靜態屬性，實例方法，靜態和類別方法，建構器， 下標腳本宣告，甚至其他結構，類別，和列舉宣告。擴展宣告不能包含析構器，協定宣告，儲存型屬性，屬性監測器或其他 的擴展屬性。詳細討論和查看包含多種擴展宣告的實例，參見擴展一節。

擴展宣告可以向現存的類別，結構，列舉內添加一致的協定。擴展宣告不能向一個類別中添加繼承的類別，因此 type-inheritance-clause是一個只包含協定列表的擴展宣告。

屬性，方法，現存型別的建構器不能被它們型別的擴展所重寫。

擴展宣告可以包含建構器宣告，這意味著，如果你擴展的型別在其他模塊中定義，建構器宣告必須委托另一個在 那個模塊裡宣告的建構器來恰當的初始化。

> 擴展(Extension)宣告語法
> 擴展宣告 → **extension** 型別標識 型別繼承子句 可選 *extension-body*
> *extension-body* → **{** 宣告*(Declarations)*列表 可選 **}**

下標腳本宣告(translated by 林)

附屬腳本用於向特定型別添加附屬腳本支援，通常為存取集合，列表和序列的元素時提供語法便利。附屬腳本宣告使用關鍵字 `subscript`，宣告形式如下：

```
subscript ( parameter ) -> (return type){
get{
statements
}
set( setter name ){
statements
}
}
```

附屬腳本宣告只能在類別，結構，列舉，擴展和協定宣告的上下文進行宣告。

變數*(parameters)*指定一個或多個用於在相關型別的下標腳本中存取元素的索引（例如，表達式 `object[i]` 中的 `i`）。儘管用於元素存取的索引可以是任意型別的，但是每個變數必須包含一個用於指定每種索引型別的型別標注。回傳型別*(return type)*指定被存取的元素的型別。

和計算性屬性一樣，下標腳本宣告支援對存取元素的讀寫操作。getter用於讀取值，setter用於寫入值。setter子句是可選的，當僅需要一個getter子句時，可以將二者都忽略且直接回傳請求的值即可。也就是說，如果使用了setter子句，就必須使用getter子句。

setter的名字和封閉的括號是可選的。如果使用了setter名稱，它會被當做傳給setter的變數的名稱。如果不使用setter名稱，那麼傳給setter的變數的名稱預設是 `value`。setter名稱的型別必須與回傳型別*(return type)*的型別相同。

可以在下標腳本宣告的型別中，可以重載下標腳本，只要變數*(parameters)*或回傳型別*(return type)*與先前的不同即可。此時，必須使用 `override` 關鍵字宣告那個被覆蓋的下標腳本。(注：好亂啊！到底是重載還是覆蓋？！)

同樣可以在協定宣告的上下文中宣告下標腳本，Protocol Subscript Declaration中有所描述。

更多關於下標腳本和下標腳本宣告的範例，請參考Subscripts。

> 附屬腳本宣告語法
> 附屬腳本宣告 → 附屬腳本頭*(Head)* 附屬腳本結果*(Result)* 程式碼區塊
> 附屬腳本宣告 → 附屬腳本頭*(Head)* 附屬腳本結果*(Result)* *getter-setter*塊
> 附屬腳本宣告 → 附屬腳本頭*(Head)* 附屬腳本結果*(Result)* *getter-setter*關鍵字*(Keyword)*塊
> 附屬腳本頭*(Head)* → 特性*(Attributes)*列表 可選 **subscript** 參數子句
> 附屬腳本結果*(Result)* → **->** 特性*(Attributes)*列表 可選 型別

運算子宣告(translated by 林)

運算子宣告會向程式中引入中綴、前綴或後綴運算子，它使用上下文關鍵字 `operator` 宣告。 可以宣告三種不同的綴性：中綴、前綴和後綴。運算子的綴性描述了運算子與它的運算元的相對位置。 運算子宣告有三種基本形式，每種綴性各一種。運算子的綴性通過在 `operator` 和運算子之間添加上下文關鍵字 `infix`， `prefix` 或 `postfix` 來指定。每種形式中，運算子的名字只能包含Operators中定義的運算子字元。

下面的這種形式宣告了一個新的中綴運算子：

```
operator infix operator name {
previewprecedence precedence level
associativity associativity
}
```

中綴運算子是二元運算子，它可以被置於兩個運算元之間，比如表達式 `1 + 2` 中的加法運算子(`+`)。

中綴運算子可以可選地指定優先級，結合性，或兩者同時指定。

運算子的優先級可以指定在沒有括號包圍的情況下，運算子與它的運算元如何緊密綁定的。可以使用上下文關鍵字 `precedence` 並優先級*(precedence level)*一起來指定一個運算子的優先級。優先級可以是0到255之間的任何一個數字(十進制整數)；與十進制整數字面量不同的是，它不可以包含任何底線字元。儘管優先級是一個特定的數字，但它僅用作與另一個運算子比較(大小)。也就是說，一個運算元可以同時被兩個運算子使用時，例如 `2 + 3 * 5`，優先級更高的運算子將優先與運算元綁定。

運算子的結合性可以指定在沒有括號包圍的情況下，優先級相同的運算子以何種順序被分組的。可以使用上下文關鍵字 `associativity` 並結合性*(associativity)*一起來指定一個運算子的結合性，其中結合性可以說是上下文關鍵字 `left`， `right` 或 `none` 中的任何一個。左結合運算子以從左到右的形式分組。例如，減法運算子(`-`)具有左結合性，因此 `4 - 5 - 6` 被以 `(4 - 5) - 6` 的形式分組，其結果為 `-7`。 右結合運算子以從右到左的形式分組，對於設置為 `none` 的非結合運算子，它們不以任何形式分組。具有相同優先級的非結合運算子，不可以互相鄰接。例如，表達式 `1 < 2 < 3` 非法的。

宣告時不指定任何優先級或結合性的中綴運算子，它們的優先級會被初始化為100，結合性被初始化為 `none`。

下面的這種形式宣告了一個新的前綴運算子：

> operator prefix `operator name` {}

緊跟在運算元前邊的前綴運算子*(prefix operator)*是一元運算子，例如表達式 `++i` 中的前綴遞增運算子(`++`)。

前綴運算子的宣告中不指定優先級。前綴運算子是非結合的。

下面的這種形式宣告了一個新的後綴運算子：

> operator postfix `operator name` {}

緊跟在運算元後邊的後綴運算子*(postfix operator)*是一元運算子，例如表達式 `i++` 中的前綴遞增運算子(`++`)。

和前綴運算子一樣，後綴運算子的宣告中不指定優先級。後綴運算子是非結合的。

宣告了一個新的運算子以後，需要宣告一個跟這個運算子同名的函式來實作這個運算子。如何實作一個新的運算子，請參考Custom Operators。

> 運算子宣告語法
> 運算子宣告 → 前綴運算子宣告 | 後綴運算子宣告 | 中綴運算子宣告
> 前綴運算子宣告 → 運算子 **prefix** 運算子 **{** **}**
> 後綴運算子宣告 → 運算子 **postfix** 運算子 **{** **}**
> 中綴運算子宣告 → 運算子 **infix** 運算子 **{** 中綴運算子屬性 可選 **}**
> 中綴運算子屬性 → 優先級子句 可選 結和性子句 可選
> 優先級子句 → **precedence** 優先級水平
> 優先級水平 → 數值 0 到 255
> 結和性子句 → **associativity** 結和性
> 結和性 → **left** | **right** | **none**

> 翻譯：Hawstein 校對：numbbbbb, stanzhai

特性

本頁內容包括：

- 宣告特性
- 型別特性

特性提供了關於宣告和型別的更多資訊。在Swift中有兩類別特性，用於修飾宣告的以及用於修飾型別的。例如，`required`特性，當應用於一個類別的指定或便利初始化器宣告時，表明它的每個子類別都必須實作那個初始化器。再比如`noreturn`特性，當應用於函式或方法型別時，表明該函式或方法不會回傳到它的呼叫者。

通過以下方式指定一個特性：符號`@`後面跟特性名，如果包含參數，則把參數帶上：

> @`attribute name`
> @`attribute name`(`attribute arguments`)

有些宣告特性通過接收參數來指定特性的更多資訊以及它是如何修飾一個特定的宣告的。這些特性的參數寫在小括號內，它們的格式由它們所屬的特性來定義。

宣告特性

宣告特性只能應用於宣告。然而，你也可以將`noreturn`特性應用於函式或方法型別。

`assignment`

該特性用於修飾重載了複合賦值運算子的函式。重載了複合賦值運算子的函式必需將它們的初始輸入參數標記為`inout`。如何使用`assignment`特性的一個範例，請見：複合賦值運算子。

`class_protocol`

該特性用於修飾一個協定表明該協定只能被類型別采用[待改：adopted]。

如果你用`objc`特性修飾一個協定，`class_protocol`特性就會隱式地應用到該協定，因此無需顯式地用`class_protocol`特性標記該協定。

`exported`

該特性用於修飾導入宣告，以此來導出已導入的模塊，子模塊，或當前模塊的宣告。如果另一個模塊導入了當前模塊，那麼那個模塊可以存取當前模塊的導出項。

`final`

該特性用於修飾一個類別或類別中的屬性，方法，以及下標成員。如果用它修飾一個類別，那麼這個類別則不能被繼承。如果用它修飾類別中的屬性，方法或下標，則表示在子類別中，它們不能被重寫。

`lazy`

該特性用於修飾類別或結構中的儲存型變數屬性，表示該屬性的初始值最多只被計算和儲存一次，且發生在第一次存取它時。如何使用`lazy`特性的一個範例，請見：惰性儲存型屬性。

`noreturn`

該特性用於修飾函式或方法宣告，表明該函式或方法的對應型別，`T`，是`@noreturn T`。你可以用這個特性修飾函式或方法

的型別，這樣一來，函式或方法就不會回傳到它的呼叫者中去。

對於一個沒有用 `noreturn` 特性標記的函式或方法，你可以將它重寫(override)為用該特性標記的。相反，對於一個已經用 `noreturn` 特性標記的函式或方法，你則不可以將它重寫為沒使用該特性標記的。相同的規則試用於當你在一個comforming型別中實作一個協定方法時。

`NSCopying`

該特性用於修飾一個類別的儲存型變數屬性。該特性將使屬性的setter與屬性值的一個副本合成，由 `copyWithZone` 方法回傳，而不是屬性本身的值。該屬性的型別必需遵循 `NSCopying` 協定。

`NSCopying` 特性的行為與Objective-C中的 `copy` 特性相似。

`NSManaged`

該特性用於修飾 `NSManagedObject` 子類別中的儲存型變數屬性，表明屬性的儲存和實作由Core Data在執行時基於相關實體描述動態提供。

`objc`

該特性用於修飾任意可以在Objective-C中表示的宣告，比如，非嵌套類別，協定，類別和協定中的屬性和方法（包含getter和setter），初始化器，析構器，以下下標。 `objc` 特性告訴編譯器該宣告可以在Objective-C程式碼中使用。

如果你將 `objc` 特性應用於一個類別或協定，它也會隱式地應用於那個類別或協定的成員。對於標記了 `objc` 特性的類別，編譯器會隱式地為它的子類別添加 `objc` 特性。標記了 `objc` 特性的協定不能繼承自沒有標記 `objc` 的協定。

`objc` 特性有一個可選的參數，由標記符組成。當你想把 `objc` 所修飾的實體以一個不同的名字暴露給Objective-C，你就可以使用這個特性參數。你可以使用這個參數來命名類別，協定，方法，getters，setters，以及初始化器。下面的範例把 `ExampleClass` 中 `enabled` 屬性的getter暴露給Objective-C，名字是 `isEnabled` ，而不是它原來的屬性名。

```
@objc
class ExampleClass {
    var enabled: Bool {
    @objc(isEnabled) get {
        // Return the appropriate value
    }
    }
}
```

`optional`

用該特性修飾協定的屬性，方法或下標成員，表示實作這些成員並不需要一致性型別（conforming type）。

你只能用 `optional` 特性修飾那些標記了 `objc` 特性的協定。因此，只有類型別可以adopt和comform to那些包含可選成員需求的協定。更多關於如何使用 `optional` 特性以及如何存取可選協定成員的指導，例如，當你不確定一個conforming型別是否實作了它們，請見：可選協定需求。

`required`

用該特性修飾一個類別的指定或便利初始化器，表示該類別的所有子類別都必需實作該初始化器。

加了該特性的指定初始化器必需顯式地實作，而便利初始化器既可顯式地實作，也可以在子類別實作了超類別所有指定初始化器後繼承而來（或者當子類別使用便利初始化器重寫了指定初始化器）。

Interface Builder使用的宣告特性

Interface Builder特性是Interface Builder用來與Xcode同步的宣告特性。Swift提供了以下的Interface Builder特性： `IBAction` ， `IBDesignable` ， `IBInspectable` ，以及 `IBOutlet` 。這些特性與Objective-C中對應的特性在概念上是相同的。

`IBOutlet` 和 `IBInspectable` 用於修飾一個類別的屬性宣告；`IBAction` 特性用於修飾一個類別的方法宣告；`IBDesignable` 用於修飾類別的宣告。

型別特性

型別特性只能用於修飾型別。然而，你也可以用 `noreturn` 特性去修飾函式或方法宣告。

`auto_closure`

這個特性通過自動地將表達式封閉到一個無參數閉包中來延遲表達式的求值。使用該特性修飾無參的函式或方法型別，回傳表達式的型別。一個如何使用 `auto_closure` 特性的範例，見函式型別

`noreturn`

該特性用於修飾函式或方法的型別，表明該函式或方法不會回傳到它的呼叫者中去。你也可以用它標記函式或方法的宣告，表示函式或方法的相應型別，`T`，是 `@noreturn T`。

> 特性語法
> 特性 → @ 特性名 特性參數子句 可選
> 特性名 → 識別符號
> 特性參數子句 → **(** 平衡語彙單元列表 可選 **)**
> 特性*(Attributes)*列表 → 特色 特性*(Attributes)*列表 可選
> 平衡語彙單元列表 → 平衡語彙單元 平衡語彙單元列表 可選
> 平衡語彙單元 → **(** 平衡語彙單元列表 可選 **)**
> 平衡語彙單元 → **[** 平衡語彙單元列表 可選 **]**
> 平衡語彙單元 → **{** 平衡語彙單元列表 可選 **}**
> 平衡語彙單元 → 任意識別符號, 關鍵字, 字面量或運算子
> 平衡語彙單元 → 任意標點除了**(**, **)**, **[**, **]**, **{**, 或 **}**

> 翻譯：honghaoz 校對：numbbbbb, stanzhai

模式（Patterns）

本頁內容包括：

模式（pattern）代表了單個值或者複合值的結構。例如，元組 `(1, 2)` 的結構是逗號分隔的，包含兩個元素的列表。因為模式代表一種值的結構，而不是特定的某個值，你可以把模式和各種同型別的值匹配起來。比如，`(x, y)` 可以匹配元組 `(1, 2)`，以及任何含兩個元素的元組。除了將模式與一個值匹配外，你可以從合成值中提取出部分或全部，然後分別把各個部分和一個常數或變數綁定起來。

在Swift中，模式出現在變數和常數的宣告（在它們的左側），`for-in` 語句和 `switch` 語句（在它們的case標簽）中。儘管任何模式都可以出現在 `switch` 語句的case標簽中，但在其他情況下，只有通配符模式（wildcard pattern），識別符號模式（identifier pattern）和包含這兩種模式的模式才能出現。

你可以為通配符模式（wildcard pattern），識別符號模式（identifier pattern）和元組模式（tuple pattern）指定型別註解，用來限制這種模式只匹配某種型別的值。

> 模式(Patterns) 語法
> 模式 → 通配符模式 型別注解 可選
> 模式 → 識別符號模式 型別注解on) 可選
> 模式 → 值綁定模式
> 模式 → 元組模式 型別注解 可選
> 模式 → *enum-case-pattern*
> 模式 → *type-casting-pattern*
> 模式 → 表達式模式

通配符模式（Wildcard Pattern）

通配符模式匹配並忽略任何值，包含一個底線（_）。當你不關心被匹配的值時，可以使用此模式。例如，下面這段程式碼進行了 `1...3` 的迴圈，並忽略了每次迴圈的值：

```
for _ in 1...3 {
    // Do something three times.
}
```

> 通配符模式語法
> 通配符模式 → _

識別符號模式（Identifier Pattern）

識別符號模式匹配任何值，並將匹配的值和一個變數或常數綁定起來。例如，在下面的常數申明中，`someValue` 是一個識別

符號模式，匹配了型別是 `Int` 的 `42`。

```
let someValue = 42
```

當匹配成功時，`42` 被綁定（賦值）給常數 `someValue`。

當一個變數或常數申明的左邊是識別符號模式時，此時，識別符號模式是隱式的值綁定模式（value-binding pattern）。

> 識別符號模式語法
> 識別符號模式 → 識別符號

值綁定模式（Value-Binding Pattern）

值綁定模式綁定匹配的值到一個變數或常數。當綁定匹配值給常數時，用關鍵字 `let`，綁定給變數時，用關鍵之 `var`。

識別符號模式包含在值綁定模式中，綁定新的變數或常數到匹配的值。例如，你可以分解一個元組的元素，並把每個元素綁定到相應的識別符號模式中。

```
let point = (3, 2)
switch point {
    // Bind x and y to the elements of point.
case let (x, y):
    println("The point is at (\(x), \(y)).")
}
// prints "The point is at (3, 2).”
```

在上面這個範例中，`let` 將元組模式 `(x, y)` 分配到各個識別符號模式。因為這種行為，`switch` 語句中 `case let (x, y):` 和 `case (let x, let y):` 匹配的值是一樣的。

> 值綁定(Value Binding)模式語法
> 值綁定模式 → **var** 模式 | **let** 模式

元組模式（Tuple Pattern）

元組模式是逗號分隔的列表，包含一個或多個模式，並包含在一對圓括號中。元組模式匹配相應元組型別的值。

你可以使用型別註解來限制一個元組模式來匹配某種元組型別。例如，在常數申明 `let (x, y): (Int, Int) = (1, 2)` 中的元組模式 `(x, y): (Int, Int)`，只匹配兩個元素都是 `Int` 這種型別的元組。如果僅需要限制一個元組模式中的某幾個元素，只需要直接對這幾個元素提供型別註解即可。例如，在 `let (x: String, y)` 中的元組模式，只要某個元組型別是包含兩個元素，且第一個元素型別是 `String`，則被匹配。

當元組模式被用在 `for-in` 語句或者變數或常數申明時，它可以包含通配符模式，識別符號模式或者其他包含這兩種模式的模式。例如，下面這段程式碼是不正確的，因為 `(x, 0)` 中的元素 `0` 是一個表達式模式：

```
let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]
// This code isn't valid.
for (x, 0) in points {
    /* ... */
}
```

對於只包含一個元素的元組，括號是不起作用的。模式匹配那個單個元素的型別。例如，下面是等效的：

```
let a = 2        // a: Int = 2
let (a) = 2      // a: Int = 2
let (a): Int = 2 // a: Int = 2
```

> 元組模式語法
> 元組模式 → **(** 元組模式元素列表 可選 **)**
> 元組模式元素列表 → 元組模式元素 | 元組模式元素 **,** 元組模式元素列表
> 元組模式元素 → 模式

列舉用例模式（Enumeration Case Pattern）

列舉用例模式匹配現有的列舉型別的某種用例。列舉用例模式僅在 `switch` 語句中的 `case` 標簽中出現。

如果你準備匹配的列舉用例有任何關聯的值，則相應的列舉用例模式必須指定一個包含每個關聯值元素的元組模式。關於使用 `switch` 語句來匹配包含關聯值列舉用例的範例，請參閱 `Associated Values`.

> 列舉用例模式語法
> *enum-case-pattern* → 型別標識 可選 **.** 列舉的*case*名 元組模式 可選

型別轉換模式（Type-Casting Patterns）

有兩種型別轉換模式，`is` 模式和 `as` 模式。這兩種模式均只出現在 `switch` 語句中的 `case` 標簽中。`is` 模式和 `as` 模式有以下形式：

> is `type`
> `pattern` as `type`

`is` 模式匹配一個值，如果這個值的型別在執行時（runtime）和 `is` 模式右邊的指定型別（或者那個型別的子類別）是一致的。`is` 模式和 `is` 運算子一樣，它們都進行型別轉換，但是拋棄了回傳的型別。

`as` 模式匹配一個值，如果這個值的型別在執行時（runtime）和 `as` 模式右邊的指定型別（或者那個型別的子類別）是一致的。一旦匹配成功，匹配的值的型別被轉換成 `as` 模式左邊指定的模式。

關於使用 `switch` 語句來匹配 `is` 模式和 `as` 模式值的範例，請參閱 `Type Casting for Any and AnyObject`。

> 型別轉換模式語法
> *type-casting-pattern* → *is*模式 | *as*模式
> *is*模式 → **is** 型別
> *as*模式 → 模式 **as** 型別

表達式模式（Expression Pattern）

表達式模式代表了一個表達式的值。這個模式只出現在 `switch` 語句中的 `case` 標簽中。

由表達式模式所代表的表達式用Swift標準函式庫中的 `~=` 運算子與輸入表達式的值進行比較。如果 `~=` 運算子回傳 `true`，則匹配成功。預設情況下，`~=` 運算子使用 `==` 運算子來比較兩個相同型別的值。它也可以匹配一個整數值與一個 `Range` 物件中的整數範圍，正如下面這個範例所示：

```
let point = (1, 2)
switch point {
case (0, 0):
    println("(0, 0) is at the origin.")
```

```
case (-2...2, -2...2):
    println("(\(point.0), \(point.1)) is near the origin.")
default:
    println("The point is at (\(point.0), \(point.1)).")
}
// prints "(1, 2) is near the origin.”
```

你可以重載 `~=` 運算子來提供自定義的表達式行為。例如，你可以重寫上面的範例，以實作用字串表達的點來比較 `point` 表達式。

```
// Overload the ~= operator to match a string with an integer
func ~=(pattern: String, value: Int) -> Bool {
    return pattern == "\(value)"
}
switch point {
case ("0", "0"):
    println("(0, 0) is at the origin.")
case ("-2...2", "-2...2"):
    println("(\(point.0), \(point.1)) is near the origin.")
default:
    println("The point is at (\(point.0), \(point.1)).")
}
// prints "(1, 2) is near the origin.”
```

> 表達式模式語法
> 表達式模式 → 表達式

> 翻譯：fd5788 校對：yankuangshi, stanzhai

泛型參數

本頁包含內容：

- 泛型形參子句
- 泛型實參子句

本節涉及泛型型別、泛型函式以及泛型建構器的參數，包括形參和實參。宣告泛型型別、函式或建構器時，須指定相應的型別參數。型別參數相當於一個占位符，當實例化泛型型別、呼叫泛型函式或泛型建構器時，就用具體的型別實參替代之。

關於 Swift 語言的泛型概述，見泛型(第二部分第22章)。

泛型形參子句

泛型形參子句指定泛型型別或函式的型別形參，以及這些參數的關聯約束和要求。泛型形參子句用角括號（<>）包住，並且有以下兩種形式：

> < `generic parameter list` >
> < `generic parameter list` where `requirements` >

泛型形參列表中泛型形參用逗號分開，每一個采用以下形式：

> `type parameter` : `constrain`

泛型形參由兩部分組成：型別形參及其後的可選約束。型別形參只是占位符型別（如T，U，V，KeyType，ValueType等）的名字而已。你可以在泛型型別、函式的其余部分或者建構器宣告，以及函式或建構器的簽名中使用它。

約束用於指明該型別形參繼承自某個類別或者遵守某個協定或協定的一部分。例如，在下面的泛型中，泛型形參 `T: Comparable` 表示任何用於替代型別形參 `T` 的型別實參必須滿足 `Comparable` 協定。

```
func simpleMin<T: COmparable>(x: T, y: T) -> T {
    if x < y {
        return y
    }
    return x
}
```

如， `Int` 和 `Double` 均滿足 `Comparable` 協定，該函式接受任何一種型別。與泛型型別相反，呼叫泛型函式或建構器時不需要指定泛型實參子句。型別實參由傳遞給函式或建構器的實參推斷而出。

```
simpleMin(17, 42) // T is inferred to be Int
simpleMin(3.14159, 2.71828) // T is inferred to be Double
```

Where 子句

要想對型別形參及其關聯型別指定額外要求，可以在泛型形參列表之後添加 `where` 子句。 `where` 子句由關鍵字 `where` 及其後的用逗號分割的多個要求組成。

`where` 子句中的要求用於指明該型別形參繼承自某個類別或遵守某個協定或協定的一部分。儘管 `where` 子句有助於表達型別形參上的簡單約束（如 `T: Comparable` 等同於 `T where T: Comparable`，等等），但是依然可以用來對型別形參及其關聯約束提供更複雜的約束。如，`<T where T: C, T: P>` 表示泛型型別 `T` 繼承自類別 `C` 且遵守協定 `P`。

如上所述，可以強制約束型別形參的關聯型別遵守某個協定。`<T: Generator where T.Element: Equatable>` 表示 `T` 遵守 `Generator` 協定，而且 `T` 的關聯型別 `T.Element` 遵守 `Eauatable` 協定（`T` 有關聯型別是因為 `Generator` 宣告了 `Element`，而 `T` 遵守 `Generator` 協定）。

也可以用運算子 `==` 來指定兩個型別等效的要求。例如，有這樣一個約束：`T` 和 `U` 遵守 `Generator` 協定，同時要求它們的關聯型別等同，可以這樣來表達：`<T: Generator, U: Generator where T.Element == U.Element>`。

當然，替代型別形參的型別實參必須滿足所有型別形參所要求的約束和要求。

泛型函式或建構器可以重載，但在泛型形參子句中的型別形參必須有不同的約束或要求，抑或二者皆不同。當呼叫重載的泛型函式或建構器時，編譯器會用這些約束來決定呼叫哪個重載函式或建構器。

泛型類別可以生成一個子類別，但是這個子類別也必須是泛型類別。

> 泛型形參子句語法
> 泛型參數子句 → **<** 泛型參數列表 約束子句 可選 **>**
> 泛型參數列表 → 泛形參數 | 泛形參數 **,** 泛型參數列表
> 泛形參數 → 型別名稱
> 泛形參數 → 型別名稱 **:** 型別標識
> 泛形參數 → 型別名稱 **:** 協定合成型別
> 約束子句 → **where** 約束列表
> 約束列表 → 約束 | 約束 **,** 約束列表
> 約束 → 一致性約束 | 同型別約束
> 一致性約束 → 型別標識 **:** 型別標識
> 一致性約束 → 型別標識 **:** 協定合成型別
> 同型別約束 → 型別標識 **==** 型別標識

泛型實參子句

泛型實參子句指定泛型型別的型別實參。泛型實參子句用角括號（<>）包住，形式如下：

> < `generic argument list` >

泛型實參列表中型別實參有逗號分開。型別實參是實際具體型別的名字，用來替代泛型型別的泛型形參子句中的相應的型別形參。從而得到泛型型別的一個特化版本。如，Swift標準函式庫的泛型字典型別定義如下：

```
struct Dictionary<KeyTypel: Hashable, ValueType>: Collection, DictionaryLiteralConvertible {
    /* .. */
}
```

泛型 `Dictionary` 型別的特化版本，`Dictionary<String, Int>` 就是用具體的 `String` 和 `Int` 型別替代泛型型別 `KeyType: Hashable` 和 `ValueType` 產生的。每一個型別實參必須滿足它所替代的泛型形參的所有約束，包括任何 `where` 子句所指定的額外的要求。上面的範例中，型別形參 `KeyType` 要求滿足 `Hashable` 協定，因此 `String` 也必須滿足 `Hashable` 協定。

可以用本身就是泛型型別的特化版本的型別實參替代型別形參（假設已滿足合適的約束和要求）。例如，為了生成一個元素型別是整型陣列的陣列，可以用陣列的特化版本 `Array<Int>` 替代泛型型別 `Array<T>` 的型別形參 `T` 來實作。

```
let arrayOfArrays: Array<Array<Int>> = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

如泛型形參子句所述，不能用泛型實參子句來指定泛型函式或建構器的型別實參。

> 泛型實參子句語法
> *(泛型參數子句Generic Argument Clause)* → **<** 泛型參數列表 **>**
> 泛型參數列表 → 泛型參數 | 泛型參數 **,** 泛型參數列表
> 泛型參數 → 型別

> 翻譯：stanzhai 校對：xielingwang

語法總結

本頁包含內容：

語句

> 語句語法
> 語句 → 表達式 ; 可選
> 語句 → 宣告 ; 可選
> 語句 → 迴圈語句 ; 可選
> 語句 → 分支語句 ; 可選
> 語句 → 標記語句*(Labeled Statement)*
> 語句 → 控制轉移語句 ; 可選
> 多條語句*(Statements)* → 語句 多條語句*(Statements)* 可選

> 迴圈語句語法
> 迴圈語句 → *for*語句
> 迴圈語句 → *for-in*語句
> 迴圈語句 → *while*語句
> 迴圈語句 → *do-while*語句

> For 迴圈語法
> *for*語句 → **for** *for*初始條件 可選 **;** 表達式 可選 **;** 表達式 可選 程式碼區塊
> *for*語句 → **for (** *for*初始條件 可選 **;** 表達式 可選 **;** 表達式 可選 **)** 程式碼區塊
> *for*初始條件 → 變數宣告 | 表達式列表

> For-In 迴圈語法
> *for-in*語句 → **for** 模式 **in** 表達式 程式碼區塊

> While 迴圈語法
> *while*語句 → **while** *while*條件 程式碼區塊
> *while*條件 → 表達式 | 宣告

> Do-While 迴圈語法
> *do-while*語句 → **do** 程式碼區塊 **while** *while*條件

> 分支語句語法
> 分支語句 → *if*語句
> 分支語句 → *switch*語句

> If語句語法
> *if*語句 → **if** *if*條件 程式碼區塊 *else*子句*(Clause)* 可選
> *if*條件 → 表達式 | 宣告
> *else*子句*(Clause)* → **else** 程式碼區塊 | **else** *if*語句

> Switch語句語法
> *switch*語句 → **switch** 表達式 **{** *SwitchCase*列表 可選 **}**
> *SwitchCase*列表 → *SwitchCase* *SwitchCase*列表 可選
> *SwitchCase* → *case*標簽 多條語句*(Statements)* | *default*標簽 多條語句*(Statements)*
> *SwitchCase* → *case*標簽 **;** | *default*標簽 **;**
> *case*標簽 → **case** *case*項列表 **:**
> *case*項列表 → 模式 *guard-clause* 可選 | 模式 *guard-clause* 可選 **,** *case*項列表
> *default*標簽 → **default :**
> *guard-clause* → **where** *guard-expression*
> *guard-expression* → 表達式

> 標記語句語法
> 標記語句*(Labeled Statement)* → 語句標簽 迴圈語句 | 語句標簽 *switch*語句
> 語句標簽 → 標簽名稱 **:**
> 標簽名稱 → 識別符號

> 控制傳遞語句(Control Transfer Statement) 語法
> 控制傳遞語句 → *break*語句
> 控制傳遞語句 → *continue*語句
> 控制傳遞語句 → *fallthrough*語句
> 控制傳遞語句 → *return*語句

> Break 語句語法
> *break*語句 → **break** 標簽名稱 可選

> Continue 語句語法
> *continue*語句 → **continue** 標簽名稱 可選

> Fallthrough 語句語法
> *fallthrough*語句 → **fallthrough**

> Return 語句語法
> *return*語句 → **return** 表達式 可選

泛型參數

> 泛型形參子句(Generic Parameter Clause) 語法
> 泛型參數子句 → **<** 泛型參數列表 約束子句 可選 **>**
> 泛型參數列表 → 泛形參數 | 泛形參數 **,** 泛型參數列表
> 泛形參數 → 型別名稱
> 泛形參數 → 型別名稱 **:** 型別標識
> 泛形參數 → 型別名稱 **:** 協定合成型別
> 約束子句 → **where** 約束列表
> 約束列表 → 約束 | 約束 **,** 約束列表
> 約束 → 一致性約束 | 同型別約束
> 一致性約束 → 型別標識 **:** 型別標識
> 一致性約束 → 型別標識 **:** 協定合成型別
> 同型別約束 → 型別標識 **==** 型別標識

泛型實參子句語法
*(*泛型參數子句*Generic Argument Clause)* → **<** 泛型參數列表 **>**
泛型參數列表 → 泛型參數 | 泛型參數 **,** 泛型參數列表
泛型參數 → 型別

宣告 (Declarations)

宣告語法
宣告 → 導入宣告
宣告 → 常數宣告
宣告 → 變數宣告
宣告 → 型別別名宣告
宣告 → 函式宣告
宣告 → 列舉宣告
宣告 → 結構宣告
宣告 → 類別宣告
宣告 → 協定宣告
宣告 → 建構器宣告
宣告 → 析構器宣告
宣告 → 擴展宣告
宣告 → 下標腳本宣告
宣告 → 運算子宣告
宣告*(Declarations)*列表 → 宣告 宣告*(Declarations)*列表 可選
宣告描述符*(Specifiers)*列表 → 宣告描述符*(Specifier)* 宣告描述符*(Specifiers)*列表 可選
宣告描述符*(Specifier)* → **class** | **mutating** | **nonmutating** | **override** | **static** | **unowned** | **unowned(safe)** | **unowned(unsafe)** | **weak**

頂級(Top Level) 宣告語法
頂級宣告 → 多條語句*(Statements)* 可選

程式碼區塊語法
程式碼區塊 → **{** 多條語句*(Statements)* 可選 **}**

導入(Import)宣告語法
導入宣告 → 特性*(Attributes)*列表 可選 **import** 導入型別 可選 導入路徑
導入型別 → **typealias** | **struct** | **class** | **enum** | **protocol** | **var** | **func**
導入路徑 → 導入路徑識別符號 | 導入路徑識別符號 **.** 導入路徑
導入路徑識別符號 → 識別符號 | 運算子

常數宣告語法
常數宣告 → 特性*(Attributes)*列表 可選 宣告描述符*(Specifiers)*列表 可選 **let** 模式建構器列表
模式建構器列表 → 模式建構器 | 模式建構器 **,** 模式建構器列表
模式建構器 → 模式 建構器 可選
建構器 → **=** 表達式

變數宣告語法
變數宣告 → 變數宣告頭*(Head)* 模式建構器列表
變數宣告 → 變數宣告頭*(Head)* 變數名 型別注解 程式碼區塊
變數宣告 → 變數宣告頭*(Head)* 變數名 型別注解 *getter-setter*塊
變數宣告 → 變數宣告頭*(Head)* 變數名 型別注解 *getter-setter*關鍵字*(Keyword)*塊
變數宣告 → 變數宣告頭*(Head)* 變數名 型別注解 建構器 可選 *willSet-didSet*程式碼區塊
變數宣告頭*(Head)* → 特性*(Attributes)*列表 可選 宣告描述符*(Specifiers)*列表 可選 **var**
變數名稱 → 識別符號

*getter-setter*塊 → **{** *getter*子句 *setter*子句 可選 **}**
*getter-setter*塊 → **{** *setter*子句 *getter*子句 **}**
*getter*子句 → 特性*(Attributes)*列表 可選 **get** 程式碼區塊
*setter*子句 → 特性*(Attributes)*列表 可選 **set** *setter*名稱 可選 程式碼區塊
*setter*名稱 → **(** 識別符號 **)**
*getter-setter*關鍵字*(Keyword)*塊 → **{** *getter*關鍵字*(Keyword)*子句 *setter*關鍵字*(Keyword)*子句 可選 **}**
*getter-setter*關鍵字*(Keyword)*塊 → **{** *setter*關鍵字*(Keyword)*子句 *getter*關鍵字*(Keyword)*子句 **}**
*getter*關鍵字*(Keyword)*子句 → 特性*(Attributes)*列表 可選 **get**
*setter*關鍵字*(Keyword)*子句 → 特性*(Attributes)*列表 可選 **set**
*willSet-didSet*程式碼區塊 → **{** *willSet*子句 *didSet*子句 可選 **}**
*willSet-didSet*程式碼區塊 → **{** *didSet*子句 *willSet*子句 **}**
*willSet*子句 → 特性*(Attributes)*列表 可選 **willSet** *setter*名稱 可選 程式碼區塊
*didSet*子句 → 特性*(Attributes)*列表 可選 **didSet** *setter*名稱 可選 程式碼區塊

型別別名宣告語法
型別別名宣告 → 型別別名頭*(Head)* 型別別名賦值
型別別名頭*(Head)* → **typealias** 型別別名名稱
型別別名名稱 → 識別符號
型別別名賦值 → **=** 型別

函式宣告語法
函式宣告 → 函式頭 函式名 泛型參數子句 可選 函式簽名*(Signature)* 函式體
函式頭 → 特性*(Attributes)*列表 可選 宣告描述符*(Specifiers)*列表 可選 **func**
函式名 → 識別符號 | 運算子
函式簽名*(Signature)* → *parameter-clauses* 函式結果 可選
函式結果 → **->** 特性*(Attributes)*列表 可選 型別
函式體 → 程式碼區塊
parameter-clauses → 參數子句 *parameter-clauses* 可選
參數子句 → **( )** | **(** 參數列表 **...** 可選 **)**
參數列表 → 參數 | 參數 **,** 參數列表
參數 → **inout** 可選 **let** 可選 # 可選 參數名 本地參數名 可選 型別注解 預設參數子句 可選
參數 → **inout** 可選 **var** # 可選 參數名 本地參數名 可選 型別注解 預設參數子句 可選
參數 → 特性*(Attributes)*列表 可選 型別
參數名 → 識別符號 | _
本地參數名 → 識別符號 | _
預設參數子句 → **=** 表達式

列舉宣告語法
列舉宣告 → 特性*(Attributes)*列表 可選 聯合式列舉 | 特性*(Attributes)*列表 可選 原始值式列舉
聯合式列舉 → 列舉名 泛型參數子句 可選 **{** *union-style-enum-members* 可選 **}**
union-style-enum-members → *union-style-enum-member union-style-enum-members* 可選
union-style-enum-member → 宣告 | 聯合式*(Union Style)*的列舉*case*子句
聯合式*(Union Style)*的列舉*case*子句 → 特性*(Attributes)*列表 可選 **case** 聯合式*(Union Style)*的列舉*case*列表
聯合式*(Union Style)*的列舉*case*列表 → 聯合式*(Union Style)*的*case* | 聯合式*(Union Style)*的*case* **,** 聯合式*(Union Style)*的列舉*case*列表
聯合式*(Union Style)*的*case* → 列舉的*case*名 元組型別 可選
列舉名 → 識別符號
列舉的*case*名 → 識別符號
原始值式列舉 → 列舉名 泛型參數子句 可選 **:** 型別標識 **{** 原始值式列舉成員列表 可選 **}**
原始值式列舉成員列表 → 原始值式列舉成員 原始值式列舉成員列表 可選
原始值式列舉成員 → 宣告 | 原始值式列舉*case*子句
原始值式列舉*case*子句 → 特性*(Attributes)*列表 可選 **case** 原始值式列舉*case*列表
原始值式列舉*case*列表 → 原始值式列舉*case* | 原始值式列舉*case* **,** 原始值式列舉*case*列表

> 原始值式列舉*case* → 列舉的*case*名 原始值賦值 可選
> 原始值賦值 → **=** 字面量

> 結構宣告語法
> 結構宣告 → 特性*(Attributes)*列表 可選 **struct** 結構名稱 泛型參數子句 可選 型別繼承子句 可選 結構主體
> 結構名稱 → 識別符號
> 結構主體 → **{** 宣告*(Declarations)*列表 可選 **}**

> 類別宣告語法
> 類別宣告 → 特性*(Attributes)*列表 可選 **class** 類別名 泛型參數子句 可選 型別繼承子句 可選 類別主體
> 類別名 → 識別符號
> 類別主體 → **{** 宣告*(Declarations)*列表 可選 **}**

> 協定(Protocol)宣告語法
> 協定宣告 → 特性*(Attributes)*列表 可選 **protocol** 協定名 型別繼承子句 可選 協定主體
> 協定名 → 識別符號
> 協定主體 → **{** 協定成員宣告*(Declarations)*列表 可選 **}**
> 協定成員宣告 → 協定屬性宣告
> 協定成員宣告 → 協定方法宣告
> 協定成員宣告 → 協定建構器宣告
> 協定成員宣告 → 協定下標腳本宣告
> 協定成員宣告 → 協定關聯型別宣告
> 協定成員宣告*(Declarations)*列表 → 協定成員宣告 協定成員宣告*(Declarations)*列表 可選

> 協定屬性宣告語法
> 協定屬性宣告 → 變數宣告頭*(Head)* 變數名 型別注解 *getter-setter*關鍵字*(Keyword)*塊

> 協定方法宣告語法
> 協定方法宣告 → 函式頭 函式名 泛型參數子句 可選 函式簽名*(Signature)*

> 協定建構器宣告語法
> 協定建構器宣告 → 建構器頭*(Head)* 泛型參數子句 可選 參數子句

> 協定下標腳本宣告語法
> 協定下標腳本宣告 → 下標腳本頭*(Head)* 下標腳本結果*(Result)* *getter-setter*關鍵字*(Keyword)*塊

> 協定關聯型別宣告語法
> 協定關聯型別宣告 → 型別別名頭*(Head)* 型別繼承子句 可選 型別別名賦值 可選

> 建構器宣告語法
> 建構器宣告 → 建構器頭*(Head)* 泛型參數子句 可選 參數子句 建構器主體
> 建構器頭*(Head)* → 特性*(Attributes)*列表 可選 **convenience** 可選 **init**
> 建構器主體 → 程式碼區塊

> 析構器宣告語法
> 析構器宣告 → 特性*(Attributes)*列表 可選 **deinit** 程式碼區塊

> 擴展(Extension)宣告語法
> 擴展宣告 → **extension** 型別標識 型別繼承子句 可選 *extension-body*
> *extension-body* → **{** 宣告*(Declarations)*列表 可選 **}**

> 下標腳本宣告語法
> 下標腳本宣告 → 下標腳本頭*(Head)* 下標腳本結果*(Result)* 程式碼區塊
> 下標腳本宣告 → 下標腳本頭*(Head)* 下標腳本結果*(Result)* *getter-setter*塊
> 下標腳本宣告 → 下標腳本頭*(Head)* 下標腳本結果*(Result)* *getter-setter*關鍵字*(Keyword)*塊

> 下標腳本頭*(Head)* → 特性*(Attributes)*列表 可選 **subscript** 參數子句
> 下標腳本結果*(Result)* → **->** 特性*(Attributes)*列表 可選 型別

> 運算子宣告語法
> 運算子宣告 → 前綴運算子宣告 | 後綴運算子宣告 | 中綴運算子宣告
> 前綴運算子宣告 → 運算子 **prefix** 運算子 **{ }**
> 後綴運算子宣告 → 運算子 **postfix** 運算子 **{ }**
> 中綴運算子宣告 → 運算子 **infix** 運算子 **{** 中綴運算子屬性 可選 **}**
> 中綴運算子屬性 → 優先級子句 可選 結和性子句 可選
> 優先級子句 → **precedence** 優先級水平
> 優先級水平 → 數值 0 到 255
> 結和性子句 → **associativity** 結和性
> 結和性 → **left** | **right** | **none**

模式

> 模式(Patterns) 語法
> 模式 → 通配符模式 型別注解 可選
> 模式 → 識別符號模式 型別注解on) 可選
> 模式 → 值綁定模式
> 模式 → 元組模式 型別注解 可選
> 模式 → *enum-case-pattern*
> 模式 → *type-casting-pattern*
> 模式 → 表達式模式

> 通配符模式語法
> 通配符模式 → _

> 識別符號模式語法
> 識別符號模式 → 識別符號

> 值綁定(Value Binding)模式語法
> 值綁定模式 → **var** 模式 | **let** 模式

> 元組模式語法
> 元組模式 → **(** 元組模式元素列表 可選 **)**
> 元組模式元素列表 → 元組模式元素 | 元組模式元素 **,** 元組模式元素列表
> 元組模式元素 → 模式

> 列舉用例模式語法
> *enum-case-pattern* → 型別標識 可選 **.** 列舉的*case*名 元組模式 可選

> 型別轉換模式語法
> *type-casting-pattern* → *is*模式 | *as*模式
> *is*模式 → **is** 型別
> *as*模式 → 模式 **as** 型別

> 表達式模式語法
> 表達式模式 → 表達式

特性

> 特性語法

> 特色 → @ 特性名 特性參數子句 可選
> 特性名 → 識別符號
> 特性參數子句 → **(** 平衡語彙單元列表 可選 **)**
> 特性*(Attributes)*列表 → 特色 特性*(Attributes)*列表 可選
> 平衡語彙單元列表 → 平衡語彙單元 平衡語彙單元列表 可選
> 平衡語彙單元 → **(** 平衡語彙單元列表 可選 **)**
> 平衡語彙單元 → **[** 平衡語彙單元列表 可選 **]**
> 平衡語彙單元 → **{** 平衡語彙單元列表 可選 **}**
> 平衡語彙單元 → 任意識別符號, 關鍵字, 字面量或運算子
> 平衡語彙單元 → 任意標點除了**(**, **)**, **[**, **]**, **{**, 或 **}**

表達式

> 表達式語法
> 表達式 → 前綴表達式 二元表達式列表 可選
> 表達式列表 → 表達式 | 表達式 **,** 表達式列表

> 前綴表達式語法
> 前綴表達式 → 前綴運算子 可選 後綴表達式
> 前綴表達式 → 寫入寫出*(in-out)*表達式
> 寫入寫出*(in-out)*表達式 → **&** 識別符號

> 二元表達式語法
> 二元表達式 → 二元運算子 前綴表達式
> 二元表達式 → 賦值運算子 前綴表達式
> 二元表達式 → 條件運算子 前綴表達式
> 二元表達式 → 型別轉換運算子
> 二元表達式列表 → 二元表達式 二元表達式列表 可選

> 賦值運算子語法
> 賦值運算子 → **=**

> 三元條件運算子語法
> 三元條件運算子 → **?** 表達式 **:**

> 型別轉換運算子語法
> 型別轉換運算子 → **is** 型別 | **as ?** 可選 型別

> 主表達式語法
> 主表達式 → 識別符號 泛型參數子句 可選
> 主表達式 → 字面量表達式
> 主表達式 → *self*表達式
> 主表達式 → 超類別表達式
> 主表達式 → 閉包表達式
> 主表達式 → 圓括號表達式
> 主表達式 → 隱式成員表達式
> 主表達式 → 通配符表達式

> 字面量表達式語法
> 字面量表達式 → 字面量
> 字面量表達式 → 陣列字面量 | 字典字面量
> 字面量表達式 → **__FILE__** | **__LINE__** | **__COLUMN__** | **__FUNCTION__**
> 陣列字面量 → **[** 陣列字面量項列表 可選 **]**

> 陣列字面量項列表 → 陣列字面量項 , 可選 | 陣列字面量項 , 陣列字面量項列表
> 陣列字面量項 → 表達式
> 字典字面量 → [字典字面量項列表] | [:]
> 字典字面量項列表 → 字典字面量項 , 可選 | 字典字面量項 , 字典字面量項列表
> 字典字面量項 → 表達式 : 表達式

> Self 表達式語法
> *self*表達式 → **self**
> *self*表達式 → **self .** 識別符號
> *self*表達式 → **self [** 表達式 **]**
> *self*表達式 → **self . init**

> 超類別表達式語法
> 超類別表達式 → 超類別方法表達式 | 超類別下標表達式 | 超類別建構器表達式
> 超類別方法表達式 → **super .** 識別符號
> 超類別下標表達式 → **super [** 表達式 **]**
> 超類別建構器表達式 → **super . init**

> 閉包表達式語法
> 閉包表達式 → **{** 閉包簽名*(Signational)* 可選 多條語句*(Statements)* **}**
> 閉包簽名*(Signational)* → 參數子句 函式結果 可選 **in**
> 閉包簽名*(Signational)* → 識別符號列表 函式結果 可選 **in**
> 閉包簽名*(Signational)* → 捕獲*(Capature)*列表 參數子句 函式結果 可選 **in**
> 閉包簽名*(Signational)* → 捕獲*(Capature)*列表 識別符號列表 函式結果 可選 **in**
> 閉包簽名*(Signational)* → 捕獲*(Capature)*列表 **in**
> 捕獲*(Capature)*列表 → **[** 捕獲*(Capature)*說明符 表達式 **]**
> 捕獲*(Capature)*說明符 → **weak** | **unowned** | **unowned(safe)** | **unowned(unsafe)**

> 隱式成員表達式語法
> 隱式成員表達式 → **.** 識別符號

> 圓括號表達式(Parenthesized Expression)語法
> 圓括號表達式 → **(** 表達式元素列表 可選 **)**
> 表達式元素列表 → 表達式元素 | 表達式元素 , 表達式元素列表
> 表達式元素 → 表達式 | 識別符號 **:** 表達式

> 通配符表達式語法
> 通配符表達式 → _

> 後綴表達式語法
> 後綴表達式 → 主表達式
> 後綴表達式 → 後綴表達式 後綴運算子
> 後綴表達式 → 函式呼叫表達式
> 後綴表達式 → 建構器表達式
> 後綴表達式 → 顯示成員表達式
> 後綴表達式 → 後綴*self*表達式
> 後綴表達式 → 動態型別表達式
> 後綴表達式 → 下標表達式
> 後綴表達式 → 強制取值*(Forced Value)*表達式
> 後綴表達式 → 可選鏈*(Optional Chaining)*表達式

> 函式呼叫表達式語法
> 函式呼叫表達式 → 後綴表達式 圓括號表達式
> 函式呼叫表達式 → 後綴表達式 圓括號表達式 可選 後綴閉包*(Trailing Closure)*

> 後綴閉包*(Trailing Closure)* → 閉包表達式

> 建構器表達式語法
> 建構器表達式 → 後綴表達式 **.** **init**

> 顯式成員表達式語法
> 顯示成員表達式 → 後綴表達式 **.** 十進制數字
> 顯示成員表達式 → 後綴表達式 **.** 識別符號 泛型參數子句 可選

> 後綴Self 表達式語法
> 後綴*self*表達式 → 後綴表達式 **.** **self**

> 動態型別表達式語法
> 動態型別表達式 → 後綴表達式 **.** **dynamicType**

> 附屬腳本表達式語法
> 附屬腳本表達式 → 後綴表達式 **[** 表達式列表 **]**

> 強制取值(Forced Value)語法
> 強制取值*(Forced Value)*表達式 → 後綴表達式 **!**

> 可選鏈表達式語法
> 可選鏈表達式 → 後綴表達式 **?**

詞法結構

> 識別符號語法
> 識別符號 → 識別符號頭*(Head)* 識別符號字元列表 可選
> 識別符號 → **`** 識別符號頭*(Head)* 識別符號字元列表 可選 **`**
> 識別符號 → 隱式參數名
> 識別符號列表 → 識別符號 **|** 識別符號 **,** 識別符號列表
> 識別符號頭*(Head)* → Upper- or lowercase letter A through Z
> 識別符號頭*(Head)* → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or U+00B7–U+00BA
> 識別符號頭*(Head)* → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, or U+00F8–U+00FF
> 識別符號頭*(Head)* → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, or U+180F–U+1DBF
> 識別符號頭*(Head)* → U+1E00–U+1FFF
> 識別符號頭*(Head)* → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040, U+2054, or U+2060–U+206F
> 識別符號頭*(Head)* → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, or U+2776–U+2793
> 識別符號頭*(Head)* → U+2C00–U+2DFF or U+2E80–U+2FFF
> 識別符號頭*(Head)* → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, or U+3040–U+D7FF
> 識別符號頭*(Head)* → U+F900–U+FD3D, U+FD40–U+FDCF, U+FDF0–U+FE1F, or U+FE30–U+FE44
> 識別符號頭*(Head)* → U+FE47–U+FFFD
> 識別符號頭*(Head)* → U+10000–U+1FFFD, U+20000–U+2FFFD, U+30000–U+3FFFD, or U+40000–U+4FFFD
> 識別符號頭*(Head)* → U+50000–U+5FFFD, U+60000–U+6FFFD, U+70000–U+7FFFD, or U+80000–U+8FFFD
> 識別符號頭*(Head)* → U+90000–U+9FFFD, U+A0000–U+AFFFD, U+B0000–U+BFFFD, or U+C0000–U+CFFFD
> 識別符號頭*(Head)* → U+D0000–U+DFFFD or U+E0000–U+EFFFD
> 識別符號字元 → 數值 0 到 9
> 識別符號字元 → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–U+FE2F
> 識別符號字元 → 識別符號頭*(Head)*
> 識別符號字元列表 → 識別符號字元 識別符號字元列表 可選
> 隱式參數名 → **$** 十進制數字列表

> 字面量語法

> 字面量 → 整型字面量 | 浮點數字面量 | 字串字面量

> 整型字面量語法
> 整型字面量 → 二進制字面量
> 整型字面量 → 八進制字面量
> 整型字面量 → 十進制字面量
> 整型字面量 → 十六進制字面量
> 二進制字面量 → **0b** 二進制數字 二進制字面量字元列表 可選
> 二進制數字 → 數值 0 到 1
> 二進制字面量字元 → 二進制數字 | _
> 二進制字面量字元列表 → 二進制字面量字元 二進制字面量字元列表 可選
> 八進制字面量 → **0o** 八進字數字 八進制字元列表 可選
> 八進字數字 → 數值 0 到 7
> 八進制字元 → 八進字數字 | _
> 八進制字元列表 → 八進制字元 八進制字元列表 可選
> 十進制字面量 → 十進制數字 十進制字元列表 可選
> 十進制數字 → 數值 0 到 9
> 十進制數字列表 → 十進制數字 十進制數字列表 可選
> 十進制字元 → 十進制數字 | _
> 十進制字元列表 → 十進制字元 十進制字元列表 可選
> 十六進制字面量 → **0x** 十六進制數字 十六進制字面量字元列表 可選
> 十六進制數字 → 數值 0 到 9, a through f, or A through F
> 十六進制字元 → 十六進制數字 | _
> 十六進制字面量字元列表 → 十六進制字元 十六進制字面量字元列表 可選

> 浮點型字面量語法
> 浮點數字面量 → 十進制字面量 十進制分數 可選 十進制指數 可選
> 浮點數字面量 → 十六進制字面量 十六進制分數 可選 十六進制指數
> 十進制分數 → **.** 十進制字面量
> 十進制指數 → 浮點數*e* 正負號 可選 十進制字面量
> 十六進制分數 → **.** 十六進制字面量 可選
> 十六進制指數 → 浮點數*p* 正負號 可選 十六進制字面量
> 浮點數*e* → **e** | **E**
> 浮點數*p* → **p** | **P**
> 正負號 → **+** | **-**

> 字元型字面量語法
> 字串字面量 → **"** 參考文字 **"**
> 參考文字 → 參考文字條目 參考文字 可選
> 參考文字條目 → 跳脫字元
> 參考文字條目 → **(** 表達式 **)**
> 參考文字條目 → 除了", \, U+000A, or U+000D的所有Unicode的字元
> 跳脫字元 → **\0** | **** | **\t** | **\n** | **\r** | **\"** | **\'**
> 跳脫字元 → **\x** 十六進制數字 十六進制數字
> 跳脫字元 → **\u** 十六進制數字 十六進制數字 十六進制數字 十六進制數字
> 跳脫字元 → **\U** 十六進制數字 十六進制數字 十六進制數字 十六進制數字 十六進制數字 十六進制數字 十六進制數字 十六進制數字

> 運算子語法語法
> 運算子 → 運算子字元 運算子 可選
> 運算子字元 → **/** | **=** | **-** | **+** | **!** | ***** | **%** | **<** | **>** | **&** | **|** | **^** | **~** | **.**
> 二元運算子 → 運算子
> 前綴運算子 → 運算子
> 後綴運算子 → 運算子

型別

型別語法
型別 → 陣列型別 | 函式型別 | 型別標識 | 元組型別 | 可選型別 | 隱式解析可選型別 | 協定合成型別 | 元型型別

型別注解語法
型別注解 → **:** 特性*(Attributes)*列表 可選 型別

型別標識語法
型別標識 → 型別名稱 泛型參數子句 可選 | 型別名稱 泛型參數子句 可選 **.** 型別標識
類別名 → 識別符號

元組型別語法
元組型別 → **(** 元組型別主體 可選 **)**
元組型別主體 → 元組型別的元素列表 **...** 可選
元組型別的元素列表 → 元組型別的元素 | 元組型別的元素 **,** 元組型別的元素列表
元組型別的元素 → 特性*(Attributes)*列表 可選 **inout** 可選 型別 | **inout** 可選 元素名 型別注解
元素名 → 識別符號

函式型別語法
函式型別 → 型別 **->** 型別

陣列型別語法
陣列型別 → 型別 **[ ]** | 陣列型別 **[ ]**

可選型別語法
可選型別 → 型別 **?**

隱式解析可選型別(Implicitly Unwrapped Optional Type)語法
隱式解析可選型別 → 型別 **!**

協定合成型別語法
協定合成型別 → **protocol <** 協定識別符號列表 可選 **>**
協定識別符號列表 → 協定識別符號 | 協定識別符號 **,** 協定識別符號列表
協定識別符號 → 型別標識

元(Metatype)型別語法
元型別 → 型別 **. Type** | 型別 **. Protocol**

型別繼承子句語法
型別繼承子句 → **:** 型別繼承列表
型別繼承列表 → 型別標識 | 型別標識 **,** 型別繼承列表