SwiftLanguage Guide中文版

极客学院出版

前言

Swift 是一款为 iOS 和 OS X 应用编程设计的全新编程语言。新增更为现代的元素，使编程更为简洁、灵活，也更有趣。界面则基于备受人们喜爱的 Cocoa 和 Cocoa Touch 框架，展示了软件开发的新方向。

简单的说，Swift 具备以下特性：

1. Swift 用来写 iOS 和 OS X 程序。
2. Swift 吸取 C 和 Objective-C 的优点，且更加强大易用。
3. Swift 可以使用现有的 Cocoa 和 Cocoa Touch 框架。
4. Swift 兼具编译语言的高性能（Performance）和脚本语言的交互性（Interactive）。

本教程直接翻译 Swift Language Guide 官方文档 (https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html) 。

图片 .1 Swift Language Guide 中文版

目录

1 关于 Swift

Swift 是一种新的编程语言，用于编写 iOS 和 OS X 应用。Swift 结合了 C 和 Objective-C 的优点并且不受 C 兼容性的限制。Swift 采用安全的编程模式并添加了很多新特性，这将使编程更简单，更灵活，也更有趣。Swift 是基于成熟而且倍受喜爱的 Cocoa 和 Cocoa Touch 框架，它的降临将重新定义软件开发。

Swift 的开发从很久之前就开始了。为了给 Swift 打好基础，苹果公司改进了编译器，调试器和框架结构。我们使用自动引用计数（Automatic Reference Counting, ARC）来简化内存管理。我们在 Foundation 和 Cocoa 的基础上构建框架栈并将其标准化。Objective-C 本身支持块、集合语法和模块，所以框架可以轻松支持现代编程语言技术。正是得益于这些基础工作，我们现在才能发布这样一个用于未来苹果软件开发的新语言。

Objective-C 开发者对 Swift 并不会感到陌生。它采用了 Objective-C 的命名参数以及动态对象模型，可以无缝对接到现有的 Cocoa 框架，并且可以兼容 Objective-C 代码。在此基础之上，Swift 还有许多新特性并且支持过程式编程和面向对象编程。

Swift 对于初学者来说也很友好。它是第一个既满足工业标准又像脚本语言一样充满表现力和趣味的编程语言。它支持代码预览，这个革命性的特性可以允许程序员在不编译和运行应用程序的前提下运行 Swift 代码并实时查看结果。

Swift 将现代编程语言的精华和苹果工程师文化的智慧结合了起来。编译器对性能进行了优化，编程语言对开发进行了优化，两者互不干扰，鱼与熊掌兼得。Swift 既可以用于开发 “hello, world” 这样的小程序，也可以用于开发一套完整的操作系统。所有的这些特性让 Swift 对于开发者和苹果来说都是一项值得的投资。

Swift 是编写 iOS 和 OS X 应用的极佳手段，并将伴随着新的特性和功能持续演进。我们对 Swift 充满信心，你还在等什么！

2

基本运算符

运算符是检查、改变、合并值的特殊符号或短语。例如，加号 `+` 将两个数相加（如 `let i = 1 + 2` ）。复杂些的运算例如逻辑与运算符 `&&` （如 `if enteredDoorCode && passedRetinaScan` ），或让 i 值加1的便捷自增运算符 `++i` 等。

Swift 支持大部分标准 C 语言的运算符，且改进许多特性来减少常规编码错误。如：赋值符（ `=` ）不返回值，以防止把想要判断相等运算符（ `==` ）的地方写成赋值符导致的错误。算术运算符（ `+` ， `-` ， `*` ， `/` ， `%` 等）会检测并不允许值溢出，以此来避免保存变量时由于变量大于或小于其类型所能承载的范围时导致的异常结果。当然允许你使用 Swift 的溢出运算符来实现溢出。详情参见溢出运算符 () 。

区别于 C 语言，在 Swift 中你可以对浮点数进行取余运算（ `%` ），Swift 还提供了 C 语言没有的表达两数之间的值的区间运算符（ `a..<b` 和 `a...b` ），这方便我们表达一个区间内的数值。

本章节只描述了 Swift 中的基本运算符，高级运算符 () 包含了高级运算符，及如何自定义运算符，及如何进行自定义类型的运算符重载。

术语

运算符有一元、二元和三元运算符。

- 一元运算符对单一操作对象操作（如 `-a` ）。一元运算符分前置运算符和后置运算符，前置运算符需紧排操作对象之前（如 `!b` ），后置运算符需紧跟操作对象之后（如 `i++` ）。
- 二元运算符操作两个操作对象（如 `2 + 3` ），是中置的，因为它们出现在两个操作对象之间。
- 三元运算符操作三个操作对象，和 C 语言一样，Swift 只有一个三元运算符，就是三目运算符（ `a ? b : c` ）。

受运算符影响的值叫操作数，在表达式 `1 + 2` 中，加号 `+` 是二元运算符，它的两个操作数是值 `1` 和 `2` 。

赋值运算符

赋值运算（ `a = b` ），表示用 `b` 的值来初始化或更新 `a` 的值：

```
let b = 10
var a = 5
a = b
// a 现在等于 10
```

如果赋值的右边是一个多元组，它的元素可以马上被分解多个常量或变量：

```
let (x, y) = (1, 2)
// 现在 x 等于 1, y 等于 2
```

与 C 语言和 Objective-C 不同，Swift 的赋值操作并不返回任何值。所以以下代码是错误的：

```
if x = y {
    // 此句错误, 因为 x = y 并不返回任何值
}
```

这个特性使你无法把（ `==` ）错写成（ `=` ），由于 `if x = y` 是错误代码，Swift帮你避免此类错误的的发生。

算术运算符

Swift 中所有数值类型都支持了基本的四则算术运算：

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

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

与 C 语言和 Objective-C 不同的是，Swift 默认情况下不允许在数值运算中出现溢出情况。但是你可以使用 Swift 的溢出运算符来实现

加法运算符也可用于`String`的拼接：

```swift
    "hello, " + "world"  // 等于 "hello, world"
```

两个 `Character` 值或一个 `String` 和一个 `Character` 值，相加会生成一个新的 `String` 值：

```
let dog: Character = "d"
let cow: Character = "c"
let dogCow = dog + cow
// 译者注: 原来的引号内是很可爱的小狗和小牛, 但win os下不支持表情字符, 所以改成了普通字符
// dogCow 现在是 "dc"
```

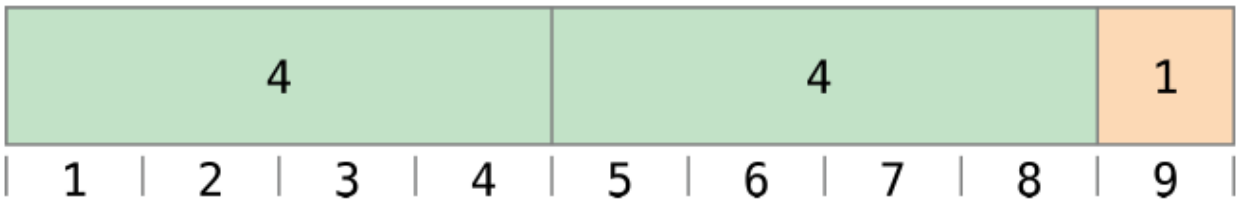
详情参见字符，字符串的拼接 () 。

求余运算符

求余运算（ `a % b` ）是计算 `b` 的多少倍刚刚好可以容入 `a` ，返回多出来的那部分（余数）。

> 注意：
> 求余运算（ `%` ）在其他语言也叫取模运算。然而严格说来，我们看该运算符对负数的操作结果，"求余"比"取模"更合适些。

我们来谈谈取余是怎么回事，计算 `9 % 4` ，你先计算出 `4` 的多少倍会刚好可以容入 `9` 中：



图片 2.1 Image of Basic Operators_1.png

2倍，非常好，那余数是1（用橙色标出）

在 Swift 中可以表达为：

```
9 % 4    // 等于 1
```

为了得到 `a % b` 的结果， `%` 计算了以下等式，并输出 `余数` 作为结果：

```
a = (b × 倍数) + 余数
```

当 `倍数` 取最大值的时候，就会刚好可以容入 `a` 中。

把 `9` 和 `4` 代入等式中，我们得 `1` ：

```
9 = (4 × 2) + 1
```

同样的方法，我来们计算 `-9 % 4` ：

```
-9 % 4   // 等于 -1
```

把 `-9` 和 `4` 代入等式， `-2` 是取到的最大整数：

```
-9 = (4 × -2) + -1
```

余数是 `-1` 。

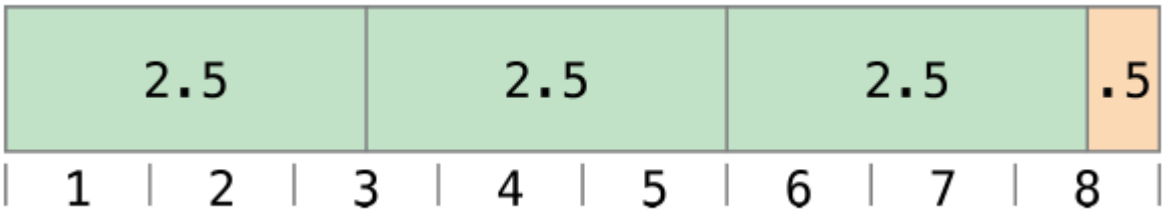
在对负数 `b` 求余时， `b` 的符号会被忽略。这意味着 `a % b` 和 `a % -b` 的结果是相同的。

浮点数求余计算

不同于 C 语言和 Objective-C，Swift 中是可以对浮点数进行求余的。

```
8 % 2.5 // 等于 0.5
```

这个例子中，`8` 除于 `2.5` 等于 `3` 余 `0.5`，所以结果是一个 `Double` 值 `0.5`。



图片 2.2 Image of Basic Operators_2.png

自增和自减运算

和 C 语言一样，Swift 也提供了对变量本身加1或减1的自增（`++`）和自减（`--`）的缩略算符。其操作对象可以是整形和浮点型。

```
var i = 0
++i     // 现在 i = 1
```

每调用一次 `++i`，`i` 的值就会加1。实际上，`++i` 是 `i = i + 1` 的简写，而 `--i` 是 `i = i - 1` 的简写。

`++` 和 `--` 既可以用作前置运算又可以用作后置运算。`++i`，`i++`，`--i` 和 `i--` 都是有效的写法。

我们需要注意的是这些运算符即可修改了 `i` 的值也可以返回 `i` 的值。如果你只想修改 `i` 的值，那你就可以忽略这个返回值。但如果你想使用返回值，你就需要留意前置和后置操作的返回值是不同的，她们遵循以下原则：

- 当 `++` 前置的时候，先自增再返回。
- 当 `++` 后置的时候，先返回再自增。

例如：

```
var a = 0
let b = ++a // a 和 b 现在都是 1
let c = a++ // a 现在 2, 但 c 是 a 自增前的值 1
```

上述例子，`let b = ++a` 先把 `a` 加1了再返回 `a` 的值。所以 `a` 和 `b` 都是新值 `1`。

而 `let c = a++`，是先返回了 `a` 的值，然后 `a` 才加1。所以 `c` 得到了 `a` 的旧值1，而 `a` 加1后变成2。

除非你需要使用 i++ 的特性，不然推荐你使用 ++i 和 --i，因为先修改后返回这样的行为更符合我们的逻辑。

一元负号运算符

数值的正负号可以使用前缀 - （即一元负号）来切换：

```
let three = 3
let minusThree = -three       // minusThree 等于 -3
let plusThree = -minusThree   // plusThree 等于 3, 或 "负负3"
```

一元负号（ - ）写在操作数之前，中间没有空格。

一元正号运算符

一元正号（ + ）不做任何改变地返回操作数的值。

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

虽然一元 + 什么都不会改变，但当你在使用一元负号来表达负数时，你可以使用一元正号来表达正数，如此你的代码会具有对称美。

复合赋值（Compound Assignment Operators）

如同强大的 C 语言，Swift 也提供把其他运算符和赋值运算（ = ）组合的复合赋值运算符，组合加运算（ += ）是其中一个例子：

```
var a = 1
a += 2
// a 现在是 3
```

表达式 a += 2 是 a = a + 2 的简写，一个组合加运算就是把加法运算和赋值运算组合成进一个运算符里，同时完成两个运算任务。

> 注意：
> 复合赋值运算没有返回值， let b = a += 2 这类代码是错误。这不同于上面提到的自增和自减运算符。

在表达式 (https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Expressions.html) 章节里有复合运算符的完整列表。

比较运算符

所有标准 C 语言中的比较运算都可以在 Swift 中使用。

- 等于（ `a == b` ）
- 不等于（ `a != b` ）
- 大于（ `a > b` ）
- 小于（ `a < b` ）
- 大于等于（ `a >= b` ）
- 小于等于（ `a <= b` ）

> 注意： Swift 也提供恒等 `===` 和不恒等 `!==` 这两个比较符来判断两个对象是否引用同一个对象实例。更多细节在类与结构 () 。

每个比较运算都返回了一个标识表达式是否成立的布尔值：

```
1 == 1   // true, 因为 1 等于 1
2 != 1   // true, 因为 2 不等于 1
2 > 1    // true, 因为 2 大于 1
1 < 2    // true, 因为 1 小于2
1 >= 1   // true, 因为 1 大于等于 1
2 <= 1   // false, 因为 2 并不小于等于 1
```

比较运算多用于条件语句，如 `if` 条件：

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

关于 `if` 语句，请看控制流 () 。

三目运算符(Ternary Conditional Operator)

三目运算符的特殊在于它是有三个操作数的运算符，它的原型是 `问题？答案1 : 答案2` 。它简洁地表达根据 `问题` 成立与否作出二选一的操作。如果 `问题` 成立，返回 `答案1` 的结果; 如果不成立，返回 `答案2` 的结果。

三目运算符是以下代码的缩写形式：

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

这里有个计算表格行高的例子。如果有表头，那行高应比内容高度要高出50像素; 如果没有表头，只需高出20像素。

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

这样写会比下面的代码简洁:

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

第一段代码例子使用了三目运算，所以一行代码就能让我们得到正确答案。这比第二段代码简洁得多，无需将 `rowHeight` 定义成变量，因为它的值无需在 `if` 语句中改变。

三目运算提供有效率且便捷的方式来表达二选一的选择。需要注意的事，过度使用三目运算符会使简洁的代码变的难懂。我们应避免在一个组合语句中使用多个三目运算符。

空合运算符(Nil Coalescing Operator)

空合运算符(`a ?? b`)将对可选类型 `a` 进行空判断，如果 `a` 包含一个值就进行解封，否则就返回一个默认值 `b` .这个运算符有两个条件:

- 表达式 `a` 必须是Optional类型
- 默认值 `b` 的类型必须要和 `a` 存储值的类型保持一致

空合并运算符是对以下代码的简短表达方法

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

上述代码使用了三目运算符。当可选类型 a 的值不为空时，进行强制解封(a!)访问 a 中值，反之当 a 中值为空时，返回默认值b。无疑空合运算符(??)提供了一种更为优雅的方式去封装条件判断和解封两种行为，显得简洁以及更具可读性。

> 注意： 如果 a 为非空值(non-nil),那么值 b 将不会被估值。这也就是所谓的短路求值。

下文例子采用空合并运算符，实现了在默认颜色名和可选自定义颜色名之间抉择：

```
let defaultColorName = "red"
var userDefinedColorName:String?   //默认值为nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
//userDefinedColorName的值为空 ，所以colorNameToUse的值为`red`
```

userDefinedColorName 变量被定义为一个可选字符串类型，默认值为nil。由于 userDefinedColorName 是一个可选类型，我们可以使用空合运算符去判断其值。在上一个例子中，通过空合运算符为一个名为 colorNameToUse 的变量赋予一个字符串类型初始值。 由于 userDefinedColorName 值为空，因此表达式 userDefinedColorName ?? defaultColorName 返回默认值，即 red 。

另一种情况，分配一个非空值(non-nil)给 userDefinedColorName ，再次执行空合运算，运算结果为封包在 userDefaultColorName 中的值，而非默认值。

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
//userDefinedColorName非空，因此colorNameToUsede的值为绿色
```

区间运算符

Swift 提供了两个方便表达一个区间的值的运算符。

闭区间运算符

闭区间运算符（ a...b ）定义一个包含从 a 到 b (包括 a 和 b)的所有值的区间， b 必须大于 a 。 闭区间运算符在迭代一个区间的所有值时是非常有用的，如在 for-in 循环中：

```
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
```

关于 for-in ，请看控制流 () 。

半开区间运算符

半开区间（ a..<b ）定义一个从 a 到 b 但不包括 b 的区间。 之所以称为半开区间，是因为该区间包含第一个值而不包括最后的值。

半开区间的实用性在于当你使用一个0始的列表(如数组)时，非常方便地从0数到列表的长度。

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

数组有4个元素，但 0..<count 只数到3(最后一个元素的下标)，因为它是半开区间。关于数组，请查阅数组 () 。

逻辑运算

逻辑运算的操作对象是逻辑布尔值。Swift 支持基于 C 语言的三个标准逻辑运算。

- 逻辑非（ !a ）
- 逻辑与（ a && b ）
- 逻辑或（ a || b ）

逻辑非

逻辑非运算（ !a ）对一个布尔值取反，使得 true 变 false， false 变 true 。

它是一个前置运算符，需出现在操作数之前，且不加空格。读作 非 a ，例子如下：

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

if !allowedEntry 语句可以读作 "如果 非 alowed entry。"，接下一行代码只有在如果 "非 allow entry" 为 true ，即 allowEntry 为 false 时被执行。

在示例代码中，小心地选择布尔常量或变量有助于代码的可读性，并且避免使用双重逻辑非运算，或混乱的逻辑语句。

逻辑与

逻辑与（ a && b ）表达了只有 a 和 b 的值都为 true 时，整个表达式的值才会是 true 。

只要任意一个值为 false ，整个表达式的值就为 false 。事实上，如果第一个值为 false ，那么是不去计算第二个值的，因为它已经不可能影响整个表达式的结果了。这被称做 "短路计算（short-circuit evaluation）"。

以下例子，只有两个 Bool 值都为 true 的时候才允许进入：

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

逻辑或

逻辑或（ a || b ）是一个由两个连续的 | 组成的中置运算符。它表示了两个逻辑表达式的其中一个为 true ，整个表达式就为 true 。

同逻辑与运算类似，逻辑或也是"短路计算"的，当左端的表达式为 true 时，将不计算右边的表达式了，因为它不可能改变整个表达式的值了。

以下示例代码中，第一个布尔值（ hasDoorKey ）为 false ，但第二个值（ knowsOverridePassword ）为 true ，所以整个表达是 true ，于是允许进入：

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

逻辑运算符组合计算

我们可以组合多个逻辑运算来表达一个复合逻辑：

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

```
}
// 输出 "Welcome!"
```

这个例子使用了含多个 `&&` 和 `||` 的复合逻辑。但无论怎样， `&&` 和 `||` 始终只能操作两个值。所以这实际是三个简单逻辑连续操作的结果。我们来解读一下：

如果我们输入了正确的密码并通过了视网膜扫描; 或者我们有一把有效的钥匙; 又或者我们知道紧急情况下重置的密码，我们就能把门打开进入。

前两种情况，我们都不满足，所以前两个简单逻辑的结果是 `false` ，但是我们是知道紧急情况下重置的密码的，所以整个复杂表达式的值还是 `true` 。

使用括号来明确优先级

为了一个复杂表达式更容易读懂，在合适的地方使用括号来明确优先级是很有效的，虽然它并非必要的。在上个关于门的权限的例子中，我们给第一个部分加个括号，使用它看起来逻辑更明确：

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

这括号使得前两个值被看成整个逻辑表达中独立的一个部分。虽然有括号和没括号的输出结果是一样的，但对于读代码的人来说有括号的代码更清晰。可读性比简洁性更重要，请在可以让你代码变清晰地地方加个括号吧！

3

字符串和字符（Strings and Characters）

String 是例如“hello, world”，“海贼王” 这样的有序的 Character （字符）类型的值的集合，通过 String 类型来表示。

Swift 的 String 和 Character 类型提供了一个快速的，兼容 Unicode 的方式来处理代码中的文本信息。 创建和操作字符串的语法与 C 语言中字符串操作相似，轻量并且易读。 字符串连接操作只需要简单地通过 + 号将两个字符串相连即可。 与 Swift 中其他值一样，能否更改字符串的值，取决于其被定义为常量还是变量。

尽管语法简易，但 String 类型是一种快速、现代化的字符串实现。 每一个字符串都是由独立编码的 Unicode 字符组成，并提供了以不同 Unicode 表示（representations）来访问这些字符的支持。

Swift 可以在常量、变量、字面量和表达式中进行字符串插值操作，可以轻松创建用于展示、存储和打印的自定义字符串。

> 注意：
> Swift 的 String 类型与 Foundation NSString 类进行了无缝桥接。如果您利用 Cocoa 或 Cocoa Touch 中的 Foundation 框架进行工作。所有 NSString API 都可以调用您创建的任意 String 类型的值。除此之外，还可以使用本章介绍的 String 特性。您也可以在任意要求传入 NSString 实例作为参数的 API 中使用 String 类型的值作为替代。 更多关于在 Foundation 和 Cocoa 中使用 String 的信息请查看 Using Swift with Cocoa and Objective-C (https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/BuildingCocoaApps/index.html#//apple_ref/doc/uid/TP40014216) 。

字符串字面量（String Literals）

您可以在您的代码中包含一段预定义的字符串值作为字符串字面量。 字符串字面量是由双引号 ("") 包裹着的具有固定顺序的文本字符集。

字符串字面量可以用于为常量和变量提供初始值。

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

> 注意：
> someString 常量通过字符串字面量进行初始化，Swift 因此推断该常量为 String 类型。

字符串字面量可以包含以下特殊字符：

- 转义字符 \0 (空字符)、 \\ (反斜线)、 \t (水平制表符)、 \n (换行符)、 \r (回车符)、 \" (双引号)、 \' (单引号)。
- Unicode 标量，写成 \u{n} (u为小写)，其中 n 为任意的一到八位十六进制数。

下面的代码为各种特殊字符的使用示例。 wiseWords 常量包含了两个转移特殊字符 (双括号)； dollarSign 、 blackHeart 和 sparklingHeart 常量演示了三种不同格式的 Unicode 标量：

```
let wiseWords = "\"我是要成为海贼王的男人\" - 路飞"
// "我是要成为海贼王的男人" - 路飞
let dollarSign = "\u{24}"            // $, Unicode 标量 U+0024
let blackHeart = "\u{2665}"          // ♥, Unicode 标量 U+2665
let sparklingHeart = "\u{1F496}"  // ?, Unicode 标量 U+1F496
```

初始化空字符串 (Initializing an Empty String)

为了构造一个很长的字符串，可以创建一个空字符串作为初始值。 可以将空的字符串字面量赋值给变量，也可以初始化一个新的 String 实例：

```
var emptyString = ""               // 空字符串字面量
var anotherEmptyString = String()  // 初始化 String 实例
// 两个字符串均为空并等价。
```

您可以通过检查其 Boolean 类型的 isEmpty 属性来判断该字符串是否为空：

```
if emptyString.isEmpty {
    println("什么都没有")
}
// 打印输出："什么都没有"
```

字符串可变性 (String Mutability)

您可以通过将一个特定字符串分配给一个变量来对其进行修改，或者分配给一个常量来保证其不会被修改：

```
var variableString = "Horse"
variableString += " and carriage"
// variableString 现在为 "Horse and carriage"
let constantString = "Highlander"
constantString += " and another Highlander"
// 这会报告一个编译错误 (compile-time error) - 常量不可以被修改。
```

> 注意：
> 在 Objective-C 和 Cocoa 中，您通过选择两个不同的类(NSString 和 NSMutableString)来指定该字符串是否可以被修改，Swift 中的字符串是否可以修改仅通过定义的是变量还是常量来决定，实现了多种类型可变性操作的统一。

字符串是值类型（Strings Are Value Types）

Swift 的 `String` 类型是值类型。 如果您创建了一个新的字符串，那么当其进行常量、变量赋值操作或在函数/方法中传递时，会进行值拷贝。 任何情况下，都会对已有字符串值创建新副本，并对该新副本进行传递或赋值操作。 值类型在 结构体和枚举是值类型 () 中进行了说明。

> 注意：
> 与 Cocoa 中的 `NSString` 不同，当您在 Cocoa 中创建了一个 `NSString` 实例，并将其传递给一个函数/方法，或者赋值给一个变量，您传递或赋值的是该 `NSString` 实例的一个引用，除非您特别要求进行值拷贝，否则字符串不会生成新的副本来进行赋值操作。

Swift 默认字符串拷贝的方式保证了在函数/方法中传递的是字符串的值。 很明显无论该值来自于哪里，都是您独自拥有的。 您可以放心您传递的字符串本身不会被更改。

在实际编译时，Swift 编译器会优化字符串的使用，使实际的复制只发生在绝对必要的情况下，这意味着您将字符串作为值类型的同时可以获得极高的性能。

()

使用字符（Working with Characters）

Swift 的 `String` 类型表示特定序列的 `Character`（字符）类型值的集合。 每一个字符值代表一个 Unicode 字符。 您可利用 `for-in` 循环来遍历字符串中的每一个字符：

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

for-in 循环在 For Loops () 中进行了详细描述。

另外，通过标明一个 `Character` 类型注解并通过字符字面量进行赋值，可以建立一个独立的字符常量或变量：

```
let yenSign: Character = "¥"
```

计算字符数量 (Counting Characters)

通过调用全局 countElements 函数，并将字符串作为参数进行传递，可以获取该字符串的字符数量。

```
let unusualMenagerie = "Koala ?, Snail ?, Penguin ?, Dromedary ?"
println("unusualMenagerie has \(countElements(unusualMenagerie)) characters")
// 打印输出："unusualMenagerie has 40 characters"
```

> 注意：
>
> 不同的 Unicode 字符以及相同 Unicode 字符的不同表示方式可能需要不同数量的内存空间来存储。所以 Swift 中的字符在一个字符串中并不一定占用相同的内存空间。因此字符串的长度不得不通过迭代字符串中每一个字符的长度来进行计算。如果您正在处理一个长字符串，需要注意 countElements 函数必须遍历字符串中的字符以精准计算字符串的长度。另外需要注意的是通过 countElements 返回的字符数量并不总是与包含相同字符的 NSString 的 length 属性相同。NSString 的 length 属性是基于利用 UTF-16 表示的十六位代码单元数字，而不是基于 Unicode 字符。为了解决这个问题，NSString 的 length 属性在被 Swift 的 String 访问时会成为 utf16count 。

()

连接字符串和字符 (Concatenating Strings and Characters)

字符串可以通过加法运算符（ + ）相加在一起（或称“串联”）并创建一个新的字符串：

```
let string1 = "hello"
let string2 = " there"
var welcome = string1 +  string2
// welcome 现在等于 "hello there"
```

您也可以通过加法赋值运算符（ += ）将一个字符串添加到一个已经存在字符串变量上：

```
var instruction = "look over"
instruction += string2
// instruction 现在等于  "look over there"
```

你可以用将 append 方法将一个字符附加到一个字符串变量的尾部：

```
let exclamationMark: Character = "!"
welcome.append(exclamationMark)
// welcome 现在等于 "hello there!"
```

> 注意：
>
> 您不能将一个字符串或者字符添加到一个已经存在的字符变量上，因为字符变量只能包含一个字符。

字符串插值 (String Interpolation)

字符串插值是一种构建新字符串的方式，可以在其中包含常量、变量、字面量和表达式。 您插入的字符串字面量的每一项都被包裹在以反斜线为前缀的圆括号中：

```
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"
```

在上面的例子中， `multiplier` 作为 `\(multiplier)` 被插入到一个字符串字面量中。 当创建字符串执行插值计算时此占位符会被替换为 `multiplier` 实际的值。

`multiplier` 的值也作为字符串中后面表达式的一部分。 该表达式计算 `Double(multiplier) * 2.5` 的值并将结果 (7.5) 插入到字符串中。 在这个例子中，表达式写为 `\(Double(multiplier) * 2.5)` 并包含在字符串字面量中。

> 注意：
> 插值字符串中写在括号中的表达式不能包含非转义双引号 (`"`) 和反斜杠 (`\`)，并且不能包含回车或换行符。

比较字符串 (Comparing Strings)

Swift 提供了三种方式来比较字符串的值：字符串相等、前缀相等和后缀相等。

字符串相等 (String Equality)

如果两个字符串以同一顺序包含完全相同的字符，则认为两者字符串相等：

```
let quotation = "我们是一样一样滴."
let sameQuotation = "我们是一样一样滴."
if quotation == sameQuotation {
    println("这两个字符串被认为是相同的")
}
// 打印输出："这两个字符串被认为是相同的"
```

前缀/后缀相等 (Prefix and Suffix Equality)

通过调用字符串的 `hasPrefix` / `hasSuffix` 方法来检查字符串是否拥有特定前缀/后缀。 两个方法均需要以字符串作为参数传入并传出 `Boolean` 值。 两个方法均执行基本字符串和前缀/后缀字符串之间逐个字符的比较操作。

下面的例子以一个字符串数组表示莎士比亚话剧《罗密欧与朱丽叶》中前两场的场景位置：

```
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"
]
```

您可以利用 `hasPrefix` 方法来计算话剧中第一幕的场景数：

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

相似地，您可以用 `hasSuffix` 方法来计算发生在不同地方的场景数：

```
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")
// 打印输出："6 mansion scenes; 2 cell scenes”
```

大写和小写字符串（Uppercase and Lowercase Strings）

您可以通过字符串的 `uppercaseString` 和 `lowercaseString` 属性来访问大写/小写版本的字符串。

```
import Foundation

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

Unicode

Unicode 是一个国际标准，用于文本的编码和表示。 它使您可以用标准格式表示来自任意语言几乎所有的字符，并能够对文本文件或网页这样的外部资源中的字符进行读写操作。

Swift 的字符串和字符类型是完全兼容 Unicode 标准的，它支持如下所述的一系列不同的 Unicode 编码。

Unicode 术语（Unicode Terminology）

Unicode 中每一个字符都可以被解释为一个或多个 unicode 标量。字符的 unicode 标量是一个唯一的21位数字(和名称)，例如 `U+0061` 表示小写的拉丁字母A ("a")，`U+1F425` 表示小鸡表情 ("?")

当 Unicode 字符串被写进文本文件或其他存储结构当中，这些 unicode 标量将会按照 Unicode 定义的集中格式之一进行编码。其包括 `UTF-8`（以8位代码单元进行编码）和 `UTF-16`（以16位代码单元进行编码）。

字符串的 Unicode 表示（Unicode Representations of Strings）

Swift 提供了几种不同的方式来访问字符串的 Unicode 表示。

您可以利用 `for-in` 来对字符串进行遍历，从而以 Unicode 字符的方式访问每一个字符值。该过程在 使用字符 (页 21) 中进行了描述。

另外，能够以其他三种 Unicode 兼容的方式访问字符串的值：

- UTF-8 代码单元集合 (利用字符串的 `utf8` 属性进行访问)
- UTF-16 代码单元集合 (利用字符串的 `utf16` 属性进行访问)
- 21位的 Unicode 标量值集合 (利用字符串的 `unicodeScalars` 属性进行访问)

下面由 `D``o``g``!` 和 `?`（`DOG FACE`，Unicode 标量为 `U+1F436`)组成的字符串中的每一个字符代表着一种不同的表示：

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

UTF-8

您可以通过遍历字符串的 `utf8` 属性来访问它的 `UTF-8` 表示。其为 `UTF8View` 类型的属性，`UTF8View` 是无符号8位（`UInt8`）值的集合，每一个 `UInt8` 值都是一个字符的 UTF-8 表示：

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

上面的例子中，前四个10进制代码单元值 (68, 111, 103, 33) 代表了字符 `D` `o` `g` 和 `!`，它们的 UTF-8 表示与 ASCII 表示相同。后四个代码单元值 (240, 159, 144, 182) 是 `DOG FACE` 的4字节 UTF-8 表示。

UTF-16

您可以通过遍历字符串的 utf16 属性来访问它的 UTF-16 表示。其为 UTF16View 类型的属性， UTF16View 是无符号16位（ UInt16 ）值的集合，每一个 UInt16 都是一个字符的 UTF-16 表示：

```
for codeUnit in dogString.utf16 {
    print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 55357 56374
```

同样，前四个代码单元值 (68, 111, 103, 33) 代表了字符 D 、 o 、 g 和！，它们的 UTF-16 代码单元和 UTF-8 完全相同。

第五和第六个代码单元值 (55357 和 56374) 是 DOG FACE 字符的UTF-16 表示。第一个值为 U+D83D (十进制值为 55357)，第二个值为 U+DC36 (十进制值为 56374)。

Unicode 标量 (Unicode Scalars)

您可以通过遍历字符串的 unicodeScalars 属性来访问它的 Unicode 标量表示。其为 UnicodeScalarView 类型的属性， UnicodeScalarView 是 UnicodeScalar 的集合。 UnicodeScalar 是21位的 Unicode 代码点。

每一个 UnicodeScalar 拥有一个值属性，可以返回对应的21位数值，用 UInt32 来表示。

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

同样，前四个代码单元值 (68, 111, 103, 33) 代表了字符 D 、 o 、 g 和！。第五位数值，128054，是一个十六进制1F436的十进制表示。其等同于 DOG FACE 的Unicode 标量 U+1F436。

作为查询字符值属性的一种替代方法，每个 UnicodeScalar 值也可以用来构建一个新的字符串值，比如在字符串插值中使用：

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

4

集合类型 (Collection Types)

Swift 语言提供经典的数组和字典两种集合类型来存储集合数据。数组用来按顺序存储相同类型的数据。字典虽然无序存储相同类型数据值但是需要由独有的标识符引用和寻址（就是键值对）。

Swift 语言里的数组和字典中存储的数据值类型必须明确。这意味着我们不能把不正确的数据类型插入其中。同时这也说明我们完全可以对获取出的值类型非常自信。Swift 对显式类型集合的使用确保了我们的代码对工作所需要的类型非常清楚，也让我们在开发中可以早早地找到任何的类型不匹配错误。

> 注意：
> Swift 的数组结构在被声明成常量和变量或者被传入函数与方法中时会相对于其他类型展现出不同的特性。获取更多信息请参见泛型 () 章节。

()

数组

数组使用有序列表存储同一类型的多个值。相同的值可以多次出现在一个数组的不同位置中。

Swift 数组特定于它所存储元素的类型。这与 Objective-C 的 NSArray 和 NSMutableArray 不同，这两个类可以存储任意类型的对象，并且不提供所返回对象的任何特别信息。在 Swift 中，数据值在被存储进入某个数组之前类型必须明确，方法是通过显式的类型标注或类型推断，而且不是必须是 class 类型。例如：如果我们创建了一个 Int 值类型的数组，我们不能往其中插入任何不是 Int 类型的数据。Swift 中的数组是类型安全的，并且它们中包含的类型必须明确。

数组的简单语法

写 Swift 数组应该遵循像 Array<SomeType> 这样的形式，其中 SomeType 是这个数组中唯一允许存在的数据类型。我们也可以使用像 [SomeType] 这样的简单语法。尽管两种形式在功能上是一样的，但是推荐较短的那种，而且在本文中都会使用这种形式来使用数组。

数组构造语句

我们可以使用字面量来进行数组构造，这是一种用一个或者多个数值构造数组的简单方法。字面量是一系列由逗号分割并由方括号包含的数值。 [value 1, value 2, value 3] 。

下面这个例子创建了一个叫做 shoppingList 并且存储字符串的数组：

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList 已经被构造并且拥有两个初始项。
```

shoppingList 变量被声明为“字符串值类型的数组“，记作 [String] 。 因为这个数组被规定只有 String 一种数据结构，所以只有 String 类型可以在其中被存取。 在这里， shoppinglist 数组由两个 String 值（ "Eggs" 和 "Milk" ）构造，并且由字面量定义。

> 注意：
> Shoppinglist 数组被声明为变量（ var 关键字创建）而不是常量（ let 创建）是因为以后可能会有更多的数据项被插入其中。

在这个例子中，字面量仅仅包含两个 String 值。匹配了该数组的变量声明（只能包含 String 的数组），所以这个字面量的分配过程就是允许用两个初始项来构造 shoppinglist 。

由于 Swift 的类型推断机制，当我们用字面量构造只拥有相同类型值数组的时候，我们不必把数组的类型定义清楚。 shoppinglist 的构造也可以这样写：

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

因为所有字面量中的值都是相同的类型，Swift 可以推断出 [String] 是 shoppinglist 中变量的正确类型。

()

访问和修改数组

我们可以通过数组的方法和属性来访问和修改数组，或者下标语法。 还可以使用数组的只读属性 count 来获取数组中的数据项数量。

```
println("The shopping list contains \(shoppingList.count) items.")
// 输出"The shopping list contains 2 items."（这个数组有2个项）
```

使用布尔项 isEmpty 来作为检查 count 属性的值是否为 0 的捷径。

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

也可以使用 append 方法在数组后面添加新的数据项：

```
shoppingList.append("Flour")
// shoppingList 现在有3个数据项，有人在摊煎饼
```

除此之外，使用加法赋值运算符（ += ）也可以直接在数组后面添加一个或多个拥有相同类型的数据项：

```
shoppingList += ["Baking Powder"]
// shoppingList 现在有四项了
```

```
shoppingList += ["Chocolate Spread","Cheese","Butter"]
// shoppingList 现在有七项了
```

可以直接使用下标语法来获取数组中的数据项，把我们需要的数据项的索引值放在直接放在数组名称的方括号中：

```
var firstItem = shoppingList[0]
// 第一项是 "Eggs"
```

注意第一项在数组中的索引值是 `0` 而不是 `1` 。 Swift 中的数组索引总是从零开始。

我们也可以用下标来改变某个已有索引值对应的数据值：

```
shoppingList[0] = "Six eggs"
// 其中的第一项现在是 "Six eggs" 而不是 "Eggs"
```

还可以利用下标来一次改变一系列数据值，即使新数据和原有数据的数量是不一样的。下面的例子把 `"Chocolate Spread"` ， `"Cheese"` ，和 `"Butter"` 替换为 `"Bananas"` 和 `"Apples"` ：

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList 现在有六项
```

> 注意：
> 我们不能使用下标语法在数组尾部添加新项。如果我们试着用这种方法对索引越界的数据进行检索或者设置新值的操作，我们会引发一个运行期错误。我们可以使用索引值和数组的 `count` 属性进行比较来在使用某个索引之前先检验是否有效。除了当 `count` 等于 0 时（说明这是个空数组），最大索引值一直是 `count - 1` ，因为数组都是零起索引。

调用数组的 `insert(atIndex:)` 方法来在某个具体索引值之前添加数据项：

```
shoppingList.insert("Maple Syrup", atIndex: 0)
// shoppingList 现在有7项
// "Maple Syrup" 现在是这个列表中的第一项
```

这次 `insert` 函数调用把值为 `"Maple Syrup"` 的新数据项插入列表的最开始位置，并且使用 `0` 作为索引值。

类似的我们可以使用 `removeAtIndex` 方法来移除数组中的某一项。这个方法把数组在特定索引值中存储的数据项移除并且返回这个被移除的数据项（我们不需要的时候就可以无视它）：

```
let mapleSyrup = shoppingList.removeAtIndex(0)
// 索引值为0的数据项被移除
// shoppingList 现在只有6项，而且不包括Maple Syrup
// mapleSyrup常量的值等于被移除数据项的值 "Maple Syrup"
```

数据项被移除后数组中的空出项会被自动填补，所以现在索引值为 `0` 的数据项的值再次等于 `"Six eggs"` ：

```
firstItem = shoppingList[0]
// firstItem 现在等于 "Six eggs"
```

如果我们只想把数组中的最后一项移除，可以使用 removeLast 方法而不是 removeAtIndex 方法来避免我们需要获取数组的 count 属性。就像后者一样，前者也会返回被移除的数据项：

```
let apples = shoppingList.removeLast()
// 数组的最后一项被移除了
// shoppingList现在只有5项，不包括cheese
// apples 常量的值现在等于"Apples" 字符串
```

数组的遍历

我们可以使用 for-in 循环来遍历所有数组中的数据项：

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

如果我们同时需要每个数据项的值和索引值，可以使用全局 enumerate 函数来进行数组遍历。 enumerate 返回一个由每一个数据项索引值和数据值组成的元组。我们可以把这个元组分解成临时常量或者变量来进行遍历：

```
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
```

更多关于 for-in 循环的介绍请参见for 循环 () 。

创建并且构造一个数组

我们可以使用构造语法来创建一个由特定数据类型构成的空数组：

```
var someInts = [Int]()
println("someInts is of type [Int] with \(someInts.count) items。")
// 打印 "someInts is of type [Int] with 0 items。" ( someInts是0数据项的Int[]数组 )
```

注意 someInts 被设置为一个 [Int] 构造函数的输出所以它的变量类型被定义为 [Int] 。

除此之外，如果代码上下文中提供了类型信息， 例如一个函数参数或者一个已经定义好类型的常量或者变量，我们可以使用空数组语句创建一个空数组，它的写法很简单： [] （一对空方括号）：

```
someInts.append(3)
// someInts 现在包含一个INT值
```

```
someInts = []
// someInts 现在是空数组，但是仍然是[Int]类型的。
```

Swift 中的 `Array` 类型还提供一个可以创建特定大小并且所有数据都被默认的构造方法。我们可以把准备加入新数组的数据项数量（ `count` ）和适当类型的初始值（ `repeatedValue` ）传入数组构造函数：

```
var threeDoubles = [Double](count: 3, repeatedValue:0.0)
// threeDoubles 是一种 [Double]数组, 等于 [0.0, 0.0, 0.0]
```

因为类型推断的存在，我们使用这种构造方法的时候不需要特别指定数组中存储的数据类型，因为类型可以从默认值推断出来：

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

最后，我们可以使用加法操作符（ `+` ）来组合两种已存在的相同类型数组。新数组的数据类型会被从两个数组的数据类型中推断出来：

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles 被推断为 [Double], 等于 [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

字典

字典是一种存储多个相同类型的值的容器。每个值（value）都关联唯一的键（key），键作为字典中的这个值数据的标识符。和数组中的数据项不同，字典中的数据项并没有具体顺序。我们在需要通过标识符（键）访问数据的时候使用字典，这种方法很大程度上和我们在现实世界中使用字典查字义的方法一样。

Swift 的字典使用时需要具体规定可以存储键和值类型。不同于 Objective-C 的 `NSDictionary` 和 `NSMutableDictionary` 类可以使用任何类型的对象来作键和值并且不提供任何关于这些对象的本质信息。在 Swift 中，在某个特定字典中可以存储的键和值必须提前定义清楚，方法是通过显性类型标注或者类型推断。

Swift 的字典使用 `Dictionary<KeyType, ValueType>` 定义,其中 `KeyType` 是字典中键的数据类型， `ValueType` 是字典中对应于这些键所存储值的数据类型。

`KeyType` 的唯一限制就是可哈希的，这样可以保证它是独一无二的，所有的 Swift 基本类型（例如 `String` ， `Int` ， `Double` 和 `Bool` ）都是默认可哈希的，并且所有这些类型都可以在字典中当做键使用。未关联值的枚举成员（参见枚举 () ）也是默认可哈希的。

字典字面量

我们可以使用字典字面量来构造字典，它们和我们刚才介绍过的数组字面量拥有相似语法。一个字典字面量是一个定义拥有一个或者多个键值对的字典集合的简单语句。

一个键值对是一个 key 和一个 value 的结合体。在字典字面量中，每一个键值对的键和值都由冒号分割。这些键值对构成一个列表，其中这些键值对由方括号包含并且由逗号分割：

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

下面的例子创建了一个存储国际机场名称的字典。在这个字典中键是三个字母的国际航空运输相关代码，值是机场名称：

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

airports 字典被定义为一种 [String: String] ,它意味着这个字典的键和值都是 String 类型。

> 注意：
> airports 字典被声明为变量（用 var 关键字）而不是常量（ let 关键字）因为后来更多的机场信息会被添加到这个示例字典中。

airports 字典使用字典字面量初始化，包含两个键值对。第一对的键是 TYO ，值是 Tokyo 。第二对的键是 DUB ，值是 Dublin 。

这个字典语句包含了两个 String: String 类型的键值对。它们对应 airports 变量声明的类型（一个只有 String 键和 String 值的字典）所以这个字典字面量是构造两个初始数据项的 airport 字典。

和数组一样，如果我们使用字面量构造字典就不用把类型定义清楚。 airports 的也可以用这种方法简短定义：

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

因为这个语句中所有的键和值都分别是相同的数据类型，Swift 可以推断出 Dictionary<String, String> 是 airports 字典的正确类型。

()

读取和修改字典

我们可以通过字典的方法和属性来读取和修改字典，或者使用下标语法。和数组一样，我们可以通过字典的只读属性 count 来获取某个字典的数据项数量：

```
println("The dictionary of airports contains \(airports.count) items.")
// 打印 "The dictionary of airports contains 2 items." ( 这个字典有两个数据项 )
```

可以使用布尔属性 isEmpty 来快捷的检查字典的 count 属性是否等于0。

```
if airports.isEmpty {
    println("The airports dictionary is empty.")
} else {
    println("The airports dictionary is not empty.")
```

```
}
// 打印 "The airports dictionary is not empty.(这个字典不为空)"
```

我们也可以在字典中使用下标语法来添加新的数据项。可以使用一个合适类型的 key 作为下标索引，并且分配新的合适类型的值：

```
airports["LHR"] = "London"
// airports 字典现在有三个数据项
```

我们也可以使用下标语法来改变特定键对应的值：

```
airports["LHR"] = "London Heathrow"
// "LHR"对应的值 被改为 "London Heathrow
```

作为另一种下标方法，字典的 updateValue(forKey:) 方法可以设置或者更新特定键对应的值。就像上面所示的示例， updateValue(forKey:) 方法在这个键不存在对应值的时候设置值或者在存在时更新已存在的值。和上面的下标方法不一样，这个方法返回更新值之前的原值。这样方便我们检查更新是否成功。

updateValue(forKey:) 函数会返回包含一个字典值类型的可选值。举例来说：对于存储 String 值的字典，这个函数会返回一个 String? 或者“可选 String ”类型的值。如果值存在，则这个可选值值等于被替换的值，否则将会是 nil 。

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

我们也可以使用下标语法来在字典中检索特定键对应的值。由于使用一个没有值的键这种情况是有可能发生的，可选类型返回这个键存在的相关值，否则就返回 nil ：

```
if let airportName = airports["DUB"] {
    println("The name of the airport is \(airportName).")
} else {
    println("That airport is not in the airports dictionary.")
}
// 打印 "The name of the airport is Dublin Internation." ( 机场的名字是都柏林国际 )
```

我们还可以使用下标语法来通过给某个键的对应值赋值为 nil 来从字典里移除一个键值对：

```
airports["APL"] = "Apple Internation"
// "Apple Internation"不是真的 APL机场, 删除它
airports["APL"] = nil
// APL现在被移除了
```

另外， removeValueForKey 方法也可以用来在字典中移除键值对。这个方法在键值对存在的情况下会移除该键值对并且返回被移除的value或者在没有值的情况下返回 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."
```

字典遍历

我们可以使用 `for-in` 循环来遍历某个字典中的键值对。每一个字典中的数据项都由 `(key, value)` 元组形式返回，并且我们可以使用临时常量或者变量来分解这些元组：

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

`for-in` 循环请参见For 循环 () 。

我们也可以通过访问它的 `keys` 或者 `values` 属性（都是可遍历集合）检索一个字典的键或者值：

```
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
```

如果我们只是需要使用某个字典的键集合或者值集合来作为某个接受 `Array` 实例 API 的参数，可以直接使用 `keys` 或者 `values` 属性直接构造一个新数组：

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

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

> 注意：
>
> Swift 的字典类型是无序集合类型。其中字典键，值，键值对在遍历的时候会重新排列，而且其中顺序是不固定的。

()

创建一个空字典

我们可以像数组一样使用构造语法创建一个空字典：

```
var namesOfIntegers = Dictionary<Int, String>()
// namesOfIntegers 是一个空的 Dictionary<Int, String>
```

这个例子创建了一个 Int, String 类型的空字典来储存英语对整数的命名。它的键是 Int 型，值是 String 型。

如果上下文已经提供了信息类型，我们可以使用空字典字面量来创建一个空字典，记作 [:]（中括号中放一个冒号）：

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers 现在包含一个键值对
namesOfIntegers = [:]
// namesOfIntegers 又成为了一个 Int, String类型的空字典
```

> 注意：
>
> 在后台，Swift 的数组和字典都是由泛型集合来实现的，想了解更多泛型和集合信息请参见泛型 () 。

集合的可变性

数组和字典都是在单个集合中存储可变值。如果我们创建一个数组或者字典并且把它分配成一个变量，这个集合将会是可变的。这意味着我们可以在创建之后添加更多或移除已存在的数据项来改变这个集合的大小。与此相反，如果我们把数组或字典分配成常量，那么它就是不可变的，它的大小不能被改变。

对字典来说，不可变性也意味着我们不能替换其中任何现有键所对应的值。不可变字典的内容在被首次设定之后不能更改。 不可变性对数组来说有一点不同，当然我们不能试着改变任何不可变数组的大小，但是我们可以重新设定相对现存索引所对应的值。这使得 Swift 数组在大小被固定的时候依然可以做的很棒。

Swift 数组的可变性行为同时影响了数组实例如何被分配和修改，想获取更多信息，请参见集合在赋值和复制中的行为 () 。

> 注意：
>
> 在我们不需要改变数组大小的时候创建不可变数组是很好的习惯。如此 Swift 编译器可以优化我们创建的集合。

5

控制流

Swift提供了类似 C 语言的流程控制结构，包括可以多次执行任务的 for 和 while 循环，基于特定条件选择执行不同代码分支的 if 和 switch 语句，还有控制流程跳转到其他代码的 break 和 continue 语句。

除了 C 语言里面传统的 for 条件递增（ for-condition-increment ）循环，Swift 还增加了 for-in 循环，用来更简单地遍历数组（array），字典（dictionary），区间（range），字符串（string）和其他序列类型。

Swift 的 switch 语句比 C 语言中更加强大。在 C 语言中，如果某个 case 不小心漏写了 break ，这个 case 就会贯穿（fallthrough）至下一个 case，Swift 无需写 break ，所以不会发生这种贯穿（fallthrough）的情况。case 还可以匹配更多的类型模式，包括区间匹配（range matching），元组（tuple）和特定类型的描述。 switch 的 case 语句中匹配的值可以是由 case 体内部临时的常量或者变量决定，也可以由 where 分句描述更复杂的匹配条件。

()

For 循环

for 循环用来按照指定的次数多次执行一系列语句。Swift 提供两种 for 循环形式：

- for-in 用来遍历一个区间（range），序列（sequence），集合（collection），系列（progression）里面所有的元素执行一系列语句。
- for条件递增（ for-condition-increment ）语句，用来重复执行一系列语句直到达成特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。

For-In

你可以使用 for-in 循环来遍历一个集合里面的所有元素，例如由数字表示的区间、数组中的元素、字符串中的字符。

下面的例子用来输出乘 5 乘法表前面一部分内容：

```
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
```

例子中用来进行遍历的元素是一组使用闭区间操作符（ ... ）表示的从 1 到 5 的数字。 index 被赋值为闭区间中的第一个数字（ 1 ），然后循环中的语句被执行一次。在本例中，这个循环只包含一个语句，用来输出当前 inde

x 值所对应的乘 5 乘法表结果。该语句执行后， index 的值被更新为闭区间中的第二个数字（ 2 ），之后 println 方法会再执行一次。整个过程会进行到闭区间结尾为止。

上面的例子中， index 是一个每次循环遍历开始时被自动赋值的常量。这种情况下， index 在使用前不需要声明，只需要将它包含在循环的声明中，就可以对其进行隐式声明，而无需使用 let 关键字声明。

> 注意：
> index 常量只存在于循环的生命周期里。如果你想在循环完成后访问 index 的值，又或者想让 index 成为一个变量而不是常量，你必须在循环之前自己进行声明。

如果你不需要知道区间内每一项的值，你可以使用下划线（ _ ）替代变量名来忽略对值的访问：

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

这个例子计算 base 这个数的 power 次幂（本例中，是 3 的 10 次幂），从 1 （ 3 的 0 次幂）开始做 3 的乘法， 进行 10 次，使用 1 到 10 的闭区间循环。这个计算并不需要知道每一次循环中计数器具体的值，只需要执行了正确的循环次数即可。下划线符号 _ （替代循环中的变量）能够忽略具体的值，并且不提供循环遍历时对值的访问。

使用 for-in 遍历一个数组所有元素：

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

你也可以通过遍历一个字典来访问它的键值对（key-value pairs）。遍历字典时，字典的每项元素会以 (key, value) 元组的形式返回，你可以在 for-in 循环中使用显式的常量名称来解读 (key, value) 元组。下面的例子中，字典的键（key）解读为常量 animalName ，字典的值会被解读为常量 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
```

字典元素的遍历顺序和插入顺序可能不同，字典的内容在内部是无序的，所以遍历元素时不能保证顺序。关于数组和字典，详情参见集合类型 () 。

除了数组和字典，你也可以使用 for-in 循环来遍历字符串中的字符（ Character ）：

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

For条件递增（for-condition-increment）

除了 for-in 循环，Swift 提供使用条件判断和递增方法的标准 C 样式 for 循环:

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

下面是一般情况下这种循环方式的格式:

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

和 C 语言中一样，分号将循环的定义分为 3 个部分，不同的是，Swift 不需要使用圆括号将“initialization; condition; increment”包括起来。

这个循环执行流程如下:

1. 循环首次启动时，初始化表达式（*initialization expression*）被调用一次，用来初始化循环所需的所有常量和变量。
2. 条件表达式（*condition expression*）被调用，如果表达式调用结果为 false ，循环结束，继续执行 for 循环关闭大括号（ } ）之后的代码。如果表达式调用结果为 true ，则会执行大括号内部的代码（*statements*）。
3. 执行所有语句（*statements*）之后，执行递增表达式（*increment expression*）。通常会增加或减少计数器的值，或者根据语句（*statements*）输出来修改某一个初始化的变量。当递增表达式运行完成后，重复执行第 2 步，条件表达式会再次执行。

上述描述和循环格式等同于：

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

在初始化表达式中声明的常量和变量（比如 var index = 0 ）只在 for 循环的生命周期里有效。如果想在循环结束后访问 index 的值，你必须要在循环生命周期开始前声明 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")
// 输出 "The loop statements were executed 3 times
```

注意 index 在循环结束后最终的值是 3 而不是 2 。最后一次调用递增表达式 ++index 会将 index 设置为 3 ，从而导致 index < 3 条件为 false ，并终止循环。

While 循环

while 循环运行一系列语句直到条件变成 false 。这类循环适合使用在第一次迭代前迭代次数未知的情况下。Swift 提供两种 while 循环形式：

- while 循环，每次在循环开始时计算条件是否符合；
- do-while 循环，每次在循环结束时计算条件是否符合。

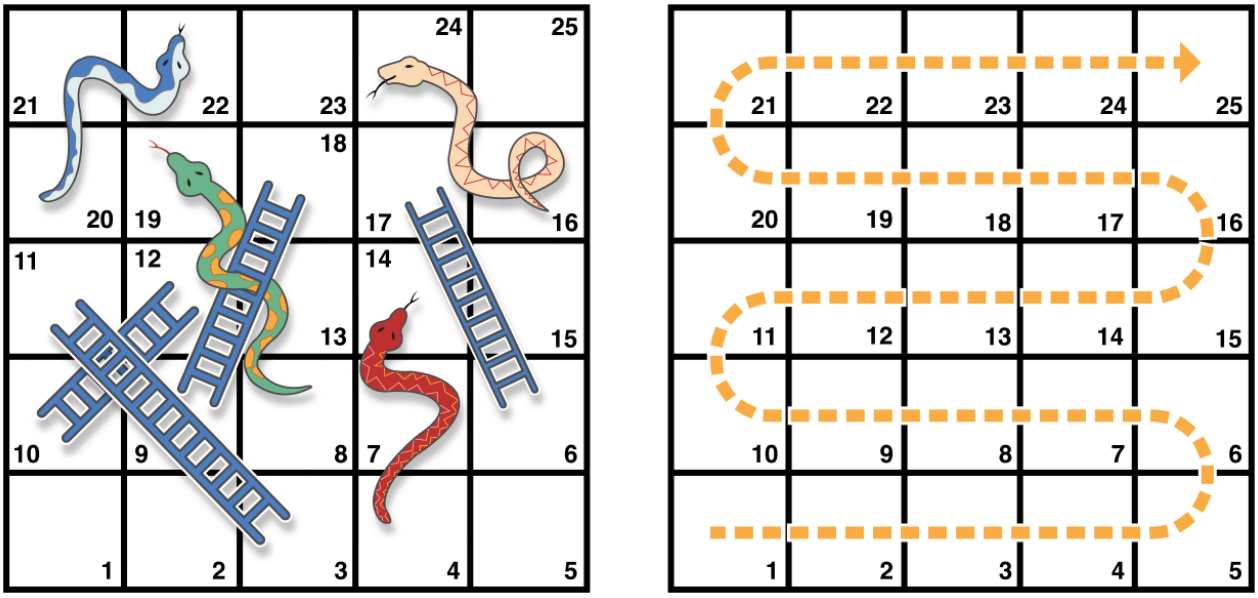
While

while 循环从计算单一条件开始。如果条件为 true ，会重复运行一系列语句，直到条件变为 false 。

下面是一般情况下 while 循环格式：

```
while condition {
statements
}
```

下面的例子来玩一个叫做*蛇和梯子（Snakes and Ladders）*的小游戏，也叫做*滑道和梯子（Chutes and Ladders）*：



图片 5.1 Image of Control_Flow_1.png

游戏的规则如下：

- 游戏盘面包括 25 个方格，游戏目标是达到或者超过第 25 个方格；
- 每一轮，你通过掷一个 6 边的骰子来确定你移动方块的步数，移动的路线由上图中横向的虚线所示；
- 如果在某轮结束，你移动到了梯子的底部，可以顺着梯子爬上去；
- 如果在某轮结束，你移动到了蛇的头部，你会顺着蛇的身体滑下去。

游戏盘面可以使用一个 `Int` 数组来表达。数组的长度由一个 `finalSquare` 常量储存，用来初始化数组和检测最终胜利条件。游戏盘面由 26 个 `Int` 0 值初始化，而不是 25 个（由 `0` 到 `25`，一共 26 个）：

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

一些方块被设置成有蛇或者梯子的指定值。梯子底部的方块是一个正值，使你可以向上移动，蛇头处的方块是一个负值，会让你向下移动：

```
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 号方块是梯子的底部，会让你向上移动到 11 号方格，我们使用 `board[03]` 等于 `+08`（来表示 `11` 和 `3` 之间的差值）。使用一元加运算符（`+i`）是为了和一元减运算符（`−i`）对称，为了让盘面代码整齐，小于 10 的数字都使用 0 补齐（这些风格上的调整都不是必须的，只是为了让代码看起来更加整洁）。

玩家由左下角编号为 0 的方格开始游戏。一般来说玩家第一次掷骰子后才会进入游戏盘面：

```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // 掷骰子
    if ++diceRoll == 7 { diceRoll = 1 }
    // 根据点数移动
    square += diceRoll
    if square < board.count {
        // 如果玩家还在棋盘上，顺着梯子爬上去或者顺着蛇滑下去
        square += board[square]
    }
}
println("Game over!")
```

本例中使用了最简单的方法来模拟掷骰子。 diceRoll 的值并不是一个随机数，而是以 0 为初始值，之后每一次 while 循环， diceRoll 的值使用前置自增操作符(++i)来自增 1 ，然后检测是否超出了最大值。 ++diceRoll 调用完成*后*，返回值等于 diceRoll 自增后的值。任何时候如果 diceRoll 的值等于7时，就超过了骰子的最大值，会被重置为 1 。所以 diceRoll 的取值顺序会一直是 1 ， 2 ， 3 ， 4 ， 5 ， 6 ， 1 ， 2 。

掷完骰子后，玩家向前移动 diceRoll 个方格，如果玩家移动超过了第 25 个方格，这个时候游戏结束，相应地，代码会在 square 增加 board[square] 的值向前或向后移动（遇到了梯子或者蛇）之前，检测 square 的值是否小于 board 的 count 属性。

如果没有这个检测（ square < board.count ）， board[square] 可能会越界访问 board 数组，导致错误。例如如果 square 等于 26 ， 代码会去尝试访问 board[26] ，超过数组的长度。

当本轮 while 循环运行完毕，会再检测循环条件是否需要再运行一次循环。如果玩家移动到或者超过第 25 个方格，循环条件结果为 false ，此时游戏结束。

while 循环比较适合本例中的这种情况，因为在 while 循环开始时，我们并不知道游戏的长度或者循环的次数，只有在达成指定条件时循环才会结束。

Do-While

while 循环的另外一种形式是 do-while ，它和 while 的区别是在判断循环条件之前，先执行一次循环的代码块，然后重复循环直到条件为 false 。

下面是一般情况下 do-while 循环的格式：

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

还是蛇和梯子的游戏，使用 do-while 循环来替代 while 循环。 finalSquare 、 board 、 square 和 diceRoll 的值初始化同 while 循环一样：

```
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 的循环版本，循环中*第一步*就需要去检测是否在梯子或者蛇的方块上。没有梯子会让玩家直接上到第 25 个方格，所以玩家不会通过梯子直接赢得游戏。这样在循环开始时先检测是否踩在梯子或者蛇上是安全的。

游戏开始时，玩家在第 0 个方格上， board[0] 一直等于 0， 不会有什么影响：

```
do {
    // 顺着梯子爬上去或者顺着蛇滑下去
    square += board[square]
    // 掷骰子
    if ++diceRoll == 7 { diceRoll = 1 }
    // 根据点数移动
    square += diceRoll
} while square < finalSquare
println("Game over!")
```

检测完玩家是否踩在梯子或者蛇上之后，开始掷骰子，然后玩家向前移动 diceRoll 个方格，本轮循环结束。

循环条件（ while square < finalSquare ）和 while 方式相同，但是只会在循环结束后进行计算。在这个游戏中， do-while 表现得比 while 循环更好。 do-while 方式会在条件判断 square 没有超出后直接运行 square += board[square] ，这种方式可以去掉 while 版本中的数组越界判断。

条件语句

根据特定的条件执行特定的代码通常是十分有用的，例如：当错误发生时，你可能想运行额外的代码；或者，当输入的值太大或太小时，向用户显示一条消息等。要实现这些功能，你就需要使用*条件语句*。

Swift 提供两种类型的条件语句： if 语句和 switch 语句。通常，当条件较为简单且可能的情况很少时，使用 if 语句。而 switch 语句更适用于条件较复杂、可能情况较多且需要用到模式匹配（pattern-matching）的情境。

If

if 语句最简单的形式就是只包含一个条件，当且仅当该条件为 true 时，才执行相关代码：

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

```
}
// 输出 "It's very cold. Consider wearing a scarf."
```

上面的例子会判断温度是否小于等于 32 华氏度（水的冰点）。如果是，则打印一条消息；否则，不打印任何消息，继续执行 `if` 块后面的代码。

当然， `if` 语句允许二选一，也就是当条件为 `false` 时，执行 *else 语句*：

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

显然，这两条分支中总有一条会被执行。由于温度已升至 40 华氏度，不算太冷，没必要再围围巾——因此， `else` 分支就被触发了。

你可以把多个 `if` 语句链接在一起，像下面这样：

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

在上面的例子中，额外的 `if` 语句用于判断是不是特别热。而最后的 `else` 语句被保留了下来，用于打印既不冷也不热时的消息。

实际上，最后的 `else` 语句是可选的：

```
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.")
}
```

在这个例子中，由于既不冷也不热，所以不会触发 `if` 或 `else if` 分支，也就不会打印任何消息。

Switch

`switch` 语句会尝试把某个值与若干个模式（pattern）进行匹配。根据第一个匹配成功的模式， `switch` 语句会执行对应的代码。当有可能的情况较多时，通常用 `switch` 语句替换 `if` 语句。

`switch` 语句最简单的形式就是把某个值与一个或若干个相同类型的值作比较：

```
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 语句都由*多个 case* 构成。为了匹配某些更特定的值，Swift 提供了几种更复杂的匹配模式，这些模式将在本节的稍后部分提到。

每一个 case 都是代码执行的一条分支，这与 if 语句类似。与之不同的是， switch 语句会决定哪一条分支应该被执行。

switch 语句必须是*完备的*。这就是说，每一个可能的值都必须至少有一个 case 分支与之对应。在某些不可能涵盖所有值的情况下，你可以使用默认（ default ）分支满足该要求，这个默认分支必须在 switch 语句的最后面。

下面的例子使用 switch 语句来匹配一个名为 someCharacter 的小写字符：

```
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")
}
// 输出 "e is a vowel"
```

在这个例子中，第一个 case 分支用于匹配五个元音，第二个 case 分支用于匹配所有的辅音。

由于为其它可能的字符写 case 分支没有实际的意义，因此在这个例子中使用了默认分支来处理剩下的既不是元音也不是辅音的字符——这就保证了 switch 语句的完备性。

不存在隐式的贯穿（No Implicit Fallthrough）

与 C 语言和 Objective-C 中的 switch 语句不同，在 Swift 中，当匹配的 case 分支中的代码执行完毕后，程序会终止 switch 语句，而不会继续执行下一个 case 分支。这也就是说，不需要在 case 分支中显式地使用 break 语句。这使得 switch 语句更安全、更易用，也避免了因忘记写 break 语句而产生的错误。

> 注意：
>
> 你依然可以在 case 分支中的代码执行完毕前跳出，详情请参考Switch 语句中的 break (页 54)。

每一个 case 分支都*必须*包含至少一条语句。像下面这样书写代码是无效的，因为第一个 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 语言里的 `switch` 语句，在 Swift 中，`switch` 语句不会同时匹配 `"a"` 和 `"A"`。相反的，上面的代码会引起编译期错误：`case "a": does not contain any executable statements` ——这就避免了意外地从一个 case 分支贯穿到另外一个，使得代码更安全、也更直观。

一个 case 也可以包含多个模式，用逗号把它们分开（如果太长了也可以分行写）：

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

> 注意：
>
> 如果想要贯穿至特定的 case 分支中，请使用 `fallthrough` 语句，详情请参考贯穿（Fallthrough）(页 55)。

区间匹配（Range Matching）

case 分支的模式也可以是一个值的区间。下面的例子展示了如何使用区间匹配来输出任意数字对应的自然语言格式：

```
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"
```

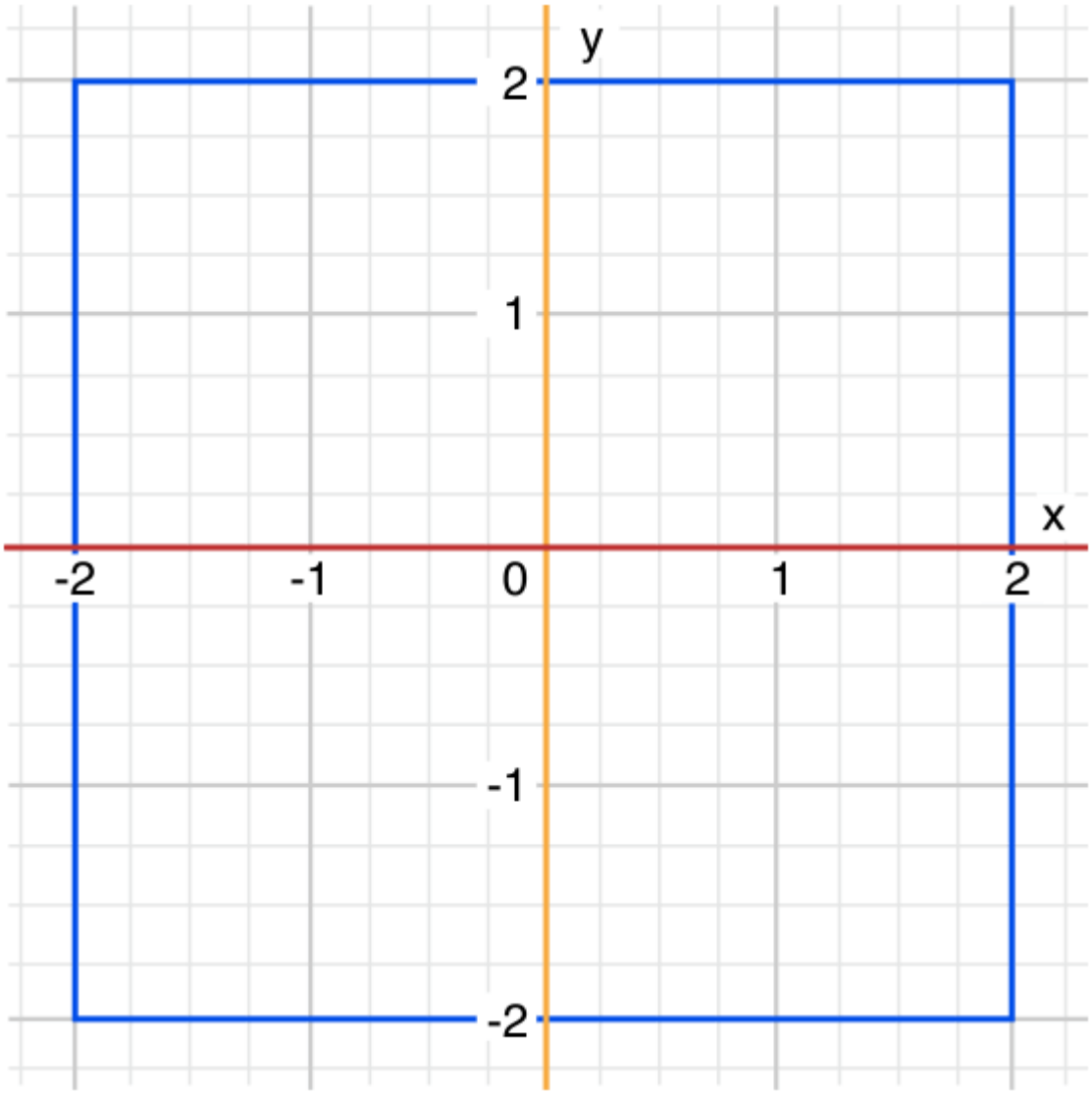
```
default:
    naturalCount = "millions and millions of"
}
println("There are \(naturalCount) \(countedThings).")
// 输出 "There are millions and millions of stars in the Milky Way."
```

元组（Tuple）

你可以使用元组在同一个 switch 语句中测试多个值。元组中的元素可以是值，也可以是区间。另外，使用下划线（ _ ）来匹配所有可能的值。

下面的例子展示了如何使用一个 (Int, Int) 类型的元组来分类下图中的点(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")
}
// 输出 "(1, 1) is inside the box"
```



图片 5.2 Image of Control_Flow_2.png

在上面的例子中，`switch`语句会判断某个点是否是原点(0, 0)，是否在红色的x轴上，是否在黄色y轴上，是否在一个以原点为中心的4x4的矩形里，或者在这个矩形外面。

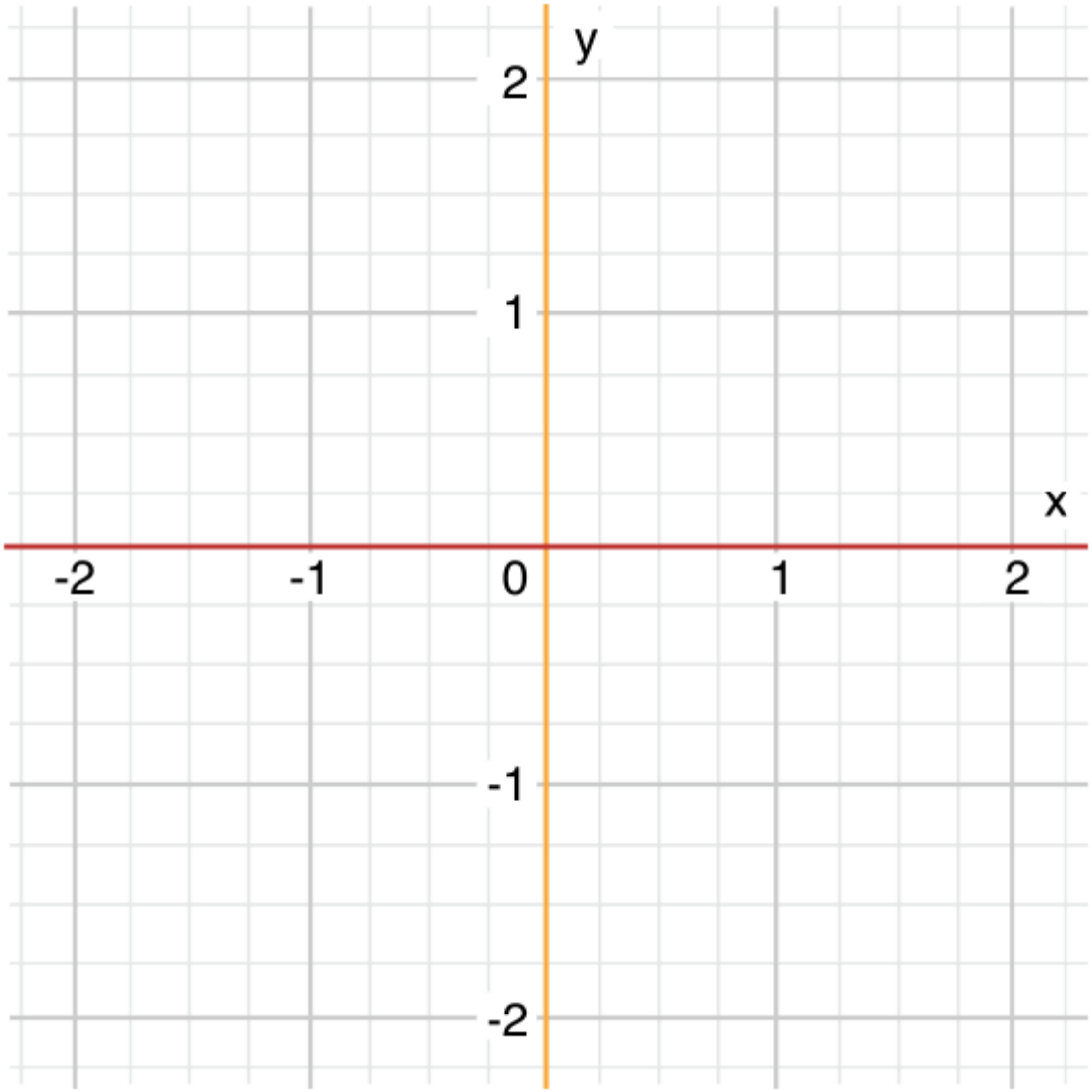
不像 C 语言，Swift 允许多个 case 匹配同一个值。实际上，在这个例子中，点(0, 0)可以匹配所有*四个* *case*。但是，如果存在多个匹配，那么只会执行第一个被匹配到的 case 分支。考虑点(0, 0)会首先匹配`case (0, 0)`，因此剩下的能够匹配(0, 0)的 case 分支都会被忽视掉。

值绑定（Value Bindings）

case 分支的模式允许将匹配的值绑定到一个临时的常量或变量，这些常量或变量在该 case 分支里就可以被引用了——这种行为被称为*值绑定*（value binding）。

下面的例子展示了如何在一个`(Int, Int)`类型的元组中使用值绑定来分类下图中的点(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))")
}
// 输出 "on the x-axis with an x value of 2"
```



图片 5.3 Image of Control_Flow_3.png

在上面的例子中， `switch` 语句会判断某个点是否在红色的x轴上，是否在黄色y轴上，或者不在坐标轴上。

这三个 case 都声明了常量 `x` 和 `y` 的占位符，用于临时获取元组 `anotherPoint` 的一个或两个值。第一个 case —— `case (let x, 0)` 将匹配一个纵坐标为 `0` 的点，并把这个点的横坐标赋给临时的常量 `x` 。类似的，第二个 case —— `case (0, let y)` 将匹配一个横坐标为 `0` 的点，并把这个点的纵坐标赋给临时的常量 `y` 。

一旦声明了这些临时的常量，它们就可以在其对应的 case 分支里引用。在这个例子中，它们用于简化 println 的书写。

请注意，这个 switch 语句不包含默认分支。这是因为最后一个 case —— case let(x, y) 声明了一个可以匹配余下所有值的元组。这使得 switch 语句已经完备了，因此不需要再书写默认分支。

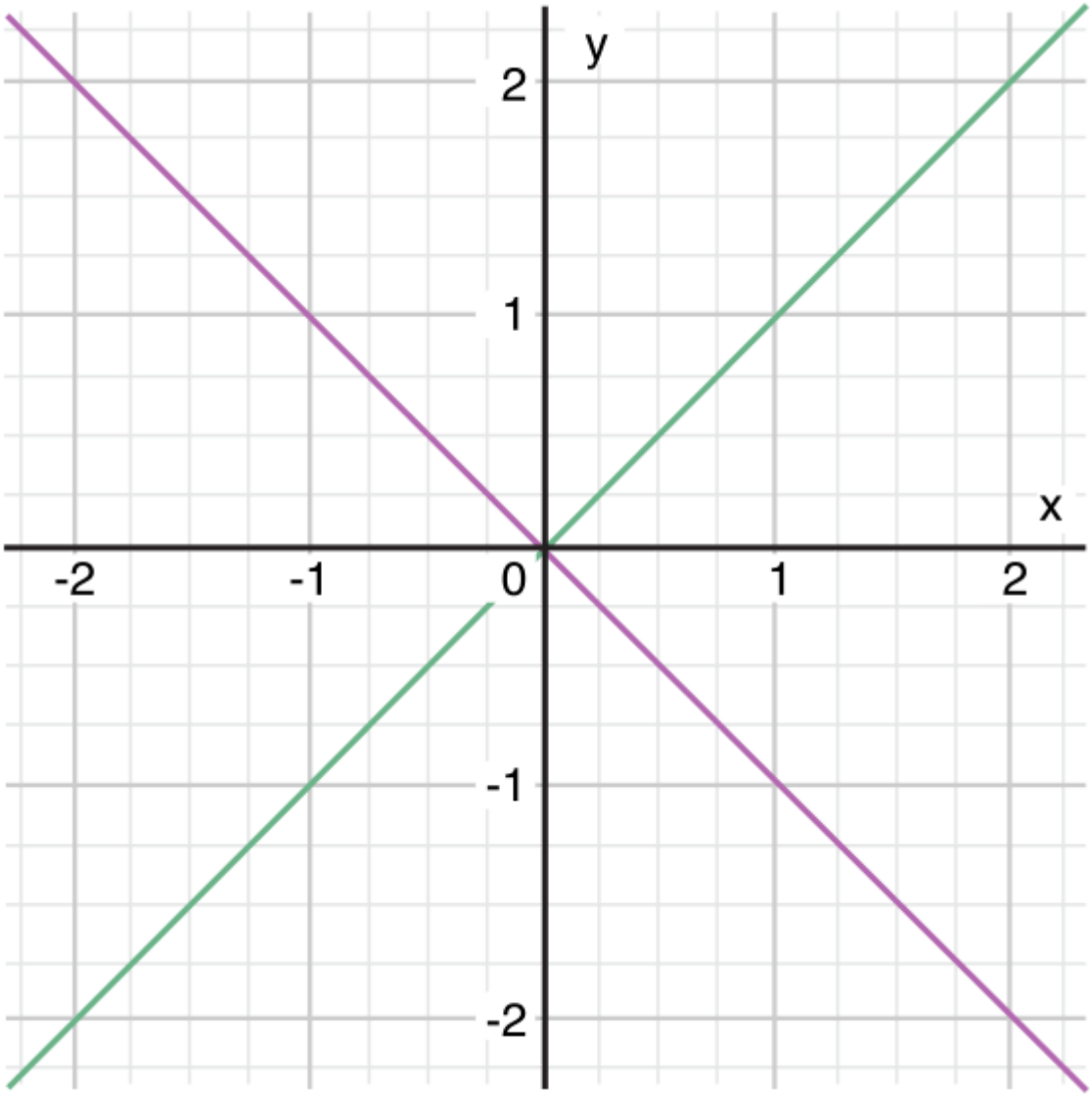
在上面的例子中， x 和 y 是常量，这是因为没有必要在其对应的 case 分支中修改它们的值。然而，它们也可以是变量——程序将会创建临时变量，并用相应的值初始化它。修改这些变量只会影响其对应的 case 分支。

Where

case 分支的模式可以使用 where 语句来判断额外的条件。

下面的例子把下图中的点(x, y)进行了分类：

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



图片 5.4 Image of Control_Flow_4.png

在上面的例子中， `switch` 语句会判断某个点是否在绿色的对角线 `x == y` 上，是否在紫色的对角线 `x == -y` 上，或者不在对角线上。

这三个 case 都声明了常量 `x` 和 `y` 的占位符，用于临时获取元组 `yetAnotherPoint` 的两个值。这些常量被用作 `where` 语句的一部分，从而创建一个动态的过滤器(filter)。当且仅当 `where` 语句的条件为 `true` 时，匹配到的 case 分支才会被执行。

就像是值绑定中的例子，由于最后一个 case 分支匹配了余下所有可能的值， `switch` 语句就已经完备了，因此不需要再书写默认分支。

控制转移语句（Control Transfer Statements）

控制转移语句改变你代码的执行顺序，通过它你可以实现代码的跳转。Swift有四种控制转移语句。

- continue
- break
- fallthrough
- return

我们将会在下面讨论 continue 、 break 和 fallthrough 语句。 return 语句将会在函数 () 章节讨论。

Continue

continue 语句告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。就好像在说“本次循环迭代我已经执行完了”，但是并不会离开整个循环体。

> 注意：
> 在一个for条件递增（ for-condition-increment ）循环体中，在调用 continue 语句后，迭代增量仍然会被计算求值。循环体继续像往常一样工作，仅仅只是循环体中的执行代码会被跳过。

下面的例子把一个小写字符串中的元音字母和空格字符移除，生成了一个含义模糊的短句：

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

在上面的代码中，只要匹配到元音字母或者空格字符，就调用 continue 语句，使本次循环迭代结束，从新开始下次循环迭代。这种行为使 switch 匹配到元音字母和空格字符时不做处理，而不是让每一个匹配到的字符都被打印。

Break

`break` 语句会立刻结束整个控制流的执行。当你想要更早的结束一个 `switch` 代码块或者一个循环体时，你都可以使用 `break` 语句。

循环语句中的 break

当在一个循环体中使用 `break` 时，会立刻中断该循环体的执行，然后跳转到表示循环体结束的大括号(`}`)后的第一行代码。不会再有本次循环迭代的代码被执行，也不会再有下次的循环迭代产生。

()

Switch 语句中的 break

当在一个 `switch` 代码块中使用 `break` 时，会立即中断该 `switch` 代码块的执行，并且跳转到表示 `switch` 代码块结束的大括号(`}`)后的第一行代码。

这种特性可以被用来匹配或者忽略一个或多个分支。因为 Swift 的 `switch` 需要包含所有的分支而且不允许有为空的分支，有时为了使你的意图更明显，需要特意匹配或者忽略某个分支。那么当你想忽略某个分支时，可以在该分支内写上 `break` 语句。当那个分支被匹配到时，分支内的 `break` 语句立即结束 `switch` 代码块。

> 注意：
> 当一个 `switch` 分支仅仅包含注释时，会被报编译时错误。注释不是代码语句而且也不能让 `switch` 分支达到被忽略的效果。你总是可以使用 `break` 来忽略某个分支。

下面的例子通过 `switch` 来判断一个 `Character` 值是否代表下面四种语言之一。为了简洁，多个值被包含在了同一个分支情况中。

```
let numberSymbol: Character = "三"  // 简体中文里的数字 3
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "?":
    possibleIntegerValue = 1
case "2", "٢", "二", "?":
    possibleIntegerValue = 2
case "3", "٣", "三", "?":
    possibleIntegerValue = 3
case "4", "٤", "四", "?":
    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).")
}
// 输出 "The integer value of 三 is 3."
```

这个例子检查 numberSymbol 是否是拉丁，阿拉伯，中文或者泰语中的 1 到 4 之一。如果被匹配到，该 switch 分支语句给 Int? 类型变量 possibleIntegerValue 设置一个整数值。

当 switch 代码块执行完后，接下来的代码通过使用可选绑定来判断 possibleIntegerValue 是否曾经被设置过值。因为是可选类型的缘故， possibleIntegerValue 有一个隐式的初始值 nil ，所以仅仅当 possibleIntegerValue 曾被 switch 代码块的前四个分支中的某个设置过一个值时，可选的绑定将会被判定为成功。

在上面的例子中，想要把 Character 所有的的可能性都枚举出来是不现实的，所以使用 default 分支来包含所有上面没有匹配到字符的情况。由于这个 default 分支不需要执行任何动作，所以它只写了一条 break 语句。一旦落入到 default 分支中后， break 语句就完成了该分支的所有代码操作，代码继续向下，开始执行 if let 语句。

()

贯穿（Fallthrough）

Swift 中的 switch 不会从上一个 case 分支落入到下一个 case 分支中。相反，只要第一个匹配到的 case 分支完成了它需要执行的语句，整个 switch 代码块完成了它的执行。相比之下，C 语言要求你显示的插入 break 语句到每个 switch 分支的末尾来阻止自动落入到下一个 case 分支中。Swift 的这种避免默认落入到下一个分支中的特性意味着它的 switch 功能要比 C 语言的更加清晰和可预测，可以避免无意识地执行多个 case 分支从而引发的错误。

如果你确实需要 C 风格的贯穿（fallthrough）的特性，你可以在每个需要该特性的 case 分支中使用 fallthrough 关键字。下面的例子使用 fallthrough 来创建一个数字的描述语句。

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

这个例子定义了一个 String 类型的变量 description 并且给它设置了一个初始值。函数使用 switch 逻辑来判断 integerToDescribe 变量的值。当 integerToDescribe 的值属于列表中的质数之一时，该函数添加一段文字在 description 后，来表明这个是数字是一个质数。然后它使用 fallthrough 关键字来“贯穿”到 default 分支中。 default 分支添加一段额外的文字在 description 的最后，至此 switch 代码块执行完了。

如果 integerToDescribe 的值不属于列表中的任何质数，那么它不会匹配到第一个 switch 分支。而这里没有其他特别的分支情况，所以 integerToDescribe 匹配到包含所有的 default 分支中。

当 switch 代码块执行完后，使用 println 函数打印该数字的描述。在这个例子中，数字 5 被准确的识别为了一个质数。

> 注意：
> fallthrough 关键字不会检查它下一个将会落入执行的 case 中的匹配条件。 fallthrough 简单地使代码执行继续连接到下一个 case 中的执行代码，这和 C 语言标准中的 switch 语句特性是一样的。

带标签的语句（Labeled Statements）

在 Swift 中，你可以在循环体和 switch 代码块中嵌套循环体和 switch 代码块来创造复杂的控制流结构。然而，循环体和 switch 代码块两者都可以使用 break 语句来提前结束整个方法体。因此，显示地指明 break 语句想要终止的是哪个循环体或者 switch 代码块，会很有用。类似地，如果你有许多嵌套的循环体，显示指明 continue 语句想要影响哪一个循环体也会非常有用。

为了实现这个目的，你可以使用标签来标记一个循环体或者 switch 代码块，当使用 break 或者 continue 时，带上这个标签，可以控制该标签代表对象的中断或者执行。

产生一个带标签的语句是通过在该语句的关键词的同一行前面放置一个标签，并且该标签后面还需带着一个冒号。下面是一个 while 循环体的语法，同样的规则适用于所有的循环体和 switch 代码块。

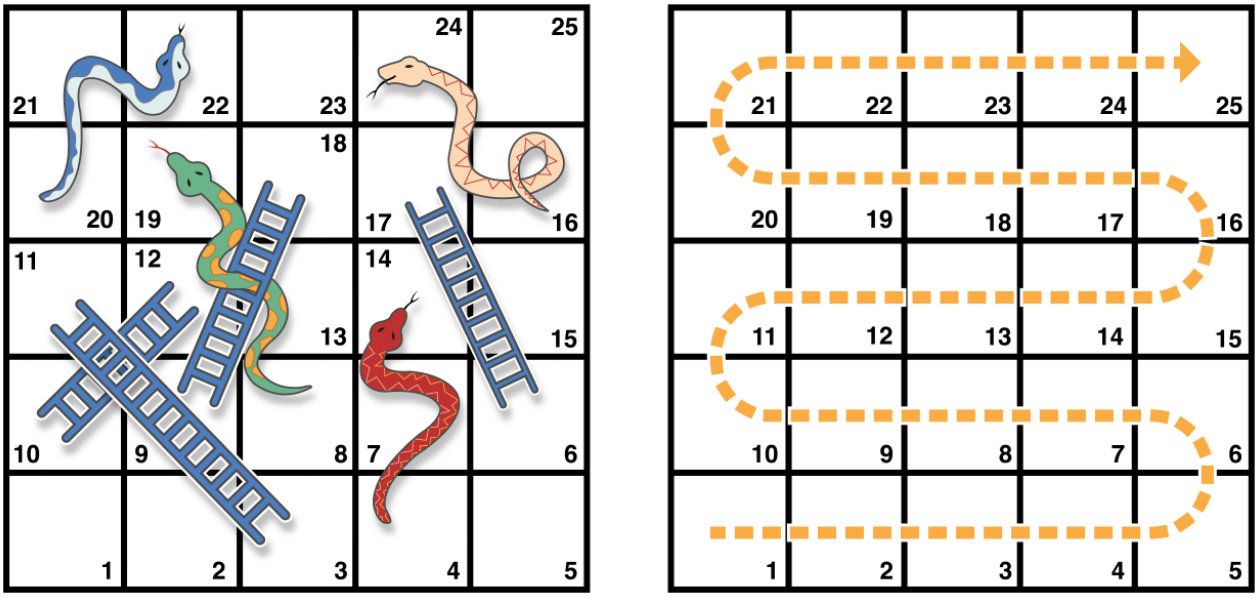
```
label name : while condition {
statements
}
```

下面的例子是在一个带有标签的 while 循环体中调用 break 和 continue 语句，该循环体是前面章节中*蛇和梯子*的改编版本。这次，游戏增加了一条额外的规则：

- 为了获胜，你必须*刚好*落在第 25 个方块中。

如果某次掷骰子使你的移动超出第 25 个方块，你必须重新掷骰子，直到你掷出的骰子数刚好使你能落在第 25 个方块中。

游戏的棋盘和之前一样：



图片 5.5 Image of Control_Flow_5.png

值 finalSquare 、 board 、 square 和 diceRoll 的初始化也和之前一样：

```
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
```

这个版本的游戏使用 while 循环体和 switch 方法块来实现游戏的逻辑。 while 循环体有一个标签名 gameLoop ，来表明它是蛇与梯子的主循环。

该 while 循环体的条件判断语句是 while square !=finalSquare ，这表明你必须刚好落在方格25中。

```
gameLoop: while square != finalSquare {
    if ++diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        // 到达最后一个方块，游戏结束
        break gameLoop
    case let newSquare where newSquare > finalSquare:
        // 超出最后一个方块，再掷一次骰子
        continue gameLoop
    default:
        // 本次移动有效
        square += diceRoll
        square += board[square]
    }
}
println("Game over!")
```

每次循环迭代开始时掷骰子。与之前玩家掷完骰子就立即移动不同，这里使用了 switch 来考虑每次移动可能产生的结果，从而决定玩家本次是否能够移动。

- 如果骰子数刚好使玩家移动到最终的方格里，游戏结束。`break gameLoop` 语句跳转控制去执行 `while` 循环体后的第一行代码，游戏结束。
- 如果骰子数将会使玩家的移动超出最后的方格，那么这种移动是不合法的，玩家需要重新掷骰子。`continue gameLoop` 语句结束本次 `while` 循环的迭代，开始下一次循环迭代。
- 在剩余的所有情况中，骰子数产生的都是合法的移动。玩家向前移动骰子数个方格，然后游戏逻辑再处理玩家当前是否处于蛇头或者梯子的底部。本次循环迭代结束，控制跳转到 `while` 循环体的条件判断语句处，再决定是否能够继续执行下次循环迭代。

> 注意：
> 如果上述的 `break` 语句没有使用 `gameLoop` 标签，那么它将会中断 `switch` 代码块而不是 `while` 循环体。使用 `gameLoop` 标签清晰的表明了 `break` 想要中断的是哪个代码块。同时请注意，当调用 `continue gameLoop` 去跳转到下一次循环迭代时，这里使用 `gameLoop` 标签并不是严格必须的。因为在这个游戏中，只有一个循环体，所以 `continue` 语句会影响到哪个循环体是没有歧义的。然而， `continue` 语句使用 `gameLoop` 标签也是没有危害的。这样做符合标签的使用规则，同时参照旁边的 `break gameLoop` ，能够使游戏的逻辑更加清晰和易于理解。

6

函数（Functions）

函数是用来完成特定任务的独立的代码块。你给一个函数起一个合适的名字，用来标识函数做什么，并且当函数需要执行的时候，这个名字会被“调用”。

Swift 统一的函数语法足够灵活，可以用来表示任何函数，包括从最简单的没有参数名字的 C 风格函数，到复杂的带局部和外部参数名的 Objective-C 风格函数。参数可以提供默认值，以简化函数调用。参数也可以既当做传入参数，也当做传出参数，也就是说，一旦函数执行结束，传入的参数值可以被修改。

在 Swift 中，每个函数都有一种类型，包括函数的参数值类型和返回值类型。你可以把函数类型当做任何其他普通变量类型一样处理，这样就可以更简单地把函数当做别的函数的参数，也可以从其他函数中返回函数。函数的定义可以写在在其他函数定义中，这样可以在嵌套函数范围内实现功能封装。

函数的定义与调用（Defining and Calling Functions）

当你定义一个函数时，你可以定义一个或多个有名字和类型的值，作为函数的输入（称为参数，parameters），也可以定义某种类型的值作为函数执行结束的输出（称为返回类型）。

每个函数有个函数名，用来描述函数执行的任务。要使用一个函数时，你用函数名“调用”，并传给它匹配的输入值（称作实参，arguments）。一个函数的实参必须与函数参数表里参数的顺序一致。

在下面例子中的函数叫做 "greetingForPerson"，之所以叫这个名字是因为这个函数用一个人的名字当做输入，并返回给这个人的问候语。为了完成这个任务，你定义一个输入参数-一个叫做 personName 的 String 值，和一个包含给这个人问候语的 String 类型的返回值：

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

所有的这些信息汇总起来成为函数的定义，并以 func 作为前缀。指定函数返回类型时，用返回箭头 ->（一个连字符后跟一个右尖括号）后跟返回类型的名称的方式来表示。

该定义描述了函数做什么，它期望接收什么和执行结束时它返回的结果是什么。这样的定义使的函数可以在别的地方以一种清晰的方式被调用：

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

调用 sayHello 函数时，在圆括号中传给它一个 String 类型的实参。因为这个函数返回一个 String 类型的值， sayHello 可以被包含在 println 的调用中，用来输出这个函数的返回值，正如上面所示。

在 sayHello 的函数体中，先定义了一个新的名为 greeting 的 String 常量，同时赋值了给 personName 的一个简单问候消息。然后用 return 关键字把这个问候返回出去。一旦 return greeting 被调用，该函数结束它的执行并返回 greeting 的当前值。

你可以用不同的输入值多次调用 sayHello 。上面的例子展示的是用 "Anna" 和 "Brian" 调用的结果，该函数分别返回了不同的结果。

为了简化这个函数的定义，可以将问候消息的创建和返回写成一句：

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

函数参数与返回值（Function Parameters and Return Values）

函数参数与返回值在Swift中极为灵活。你可以定义任何类型的函数，包括从只带一个未名参数的简单函数到复杂的带有表达性参数名和不同参数选项的复杂函数。

多重输入参数（Multiple Input Parameters）

函数可以有多个输入参数，写在圆括号中，用逗号分隔。

下面这个函数用一个半开区间的开始点和结束点，计算出这个范围内包含多少数字：

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

无参函数（Functions Without Parameters）

函数可以没有参数。下面这个函数就是一个无参函数，当被调用时，它返回固定的 String 消息：

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

尽管这个函数没有参数，但是定义中在函数名后还是需要一对圆括号。当被调用时，也需要在函数名后写一对圆括号。

无返回值函数（Functions Without Return Values）

函数可以没有返回值。下面是 `sayHello` 函数的另一个版本，叫 `waveGoodbye`，这个函数直接输出 `String` 值，而不是返回它：

```
func sayGoodbye(personName: String) {
    println("Goodbye, \(personName)!")
}
sayGoodbye("Dave")
// prints "Goodbye, Dave!"
```

因为这个函数不需要返回值，所以这个函数的定义中没有返回箭头（`->`）和返回类型。

> 注意： 严格上来说，虽然没有返回值被定义，`sayGoodbye` 函数依然返回了值。没有定义返回类型的函数会返回特殊的值，叫 `Void`。它其实是一个空的元组（tuple），没有任何元素，可以写成 `()`。

被调用时，一个函数的返回值可以被忽略：

```
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
```

第一个函数 `printAndCount`，输出一个字符串并返回 `Int` 类型的字符数。第二个函数 `printWithoutCounting` 调用了第一个函数，但是忽略了它的返回值。当第二个函数被调用时，消息依然会由第一个函数输出，但是返回值不会被用到。

> 注意： 返回值可以被忽略，但定义了有返回值的函数必须返回一个值，如果在函数定义底部没有返回任何值，这将导致编译错误（compile-time error）。

多重返回值函数（Functions with Multiple Return Values）

你可以用元组（tuple）类型让多个值作为一个复合值从函数中返回。

下面的这个例子中，`count` 函数用来计算一个字符串中元音，辅音和其他字母的个数（基于美式英语的标准）。

```
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` 函数来处理任何一个字符串，返回的值将是一个包含三个 `Int` 型值的元组（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，元组的成员不需要在函数中返回时命名，因为它们的名字已经在函数返回类型中有了定义。

函数参数名称（Function Parameter Names）

以上所有的函数都给它们的参数定义了 `参数名（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，这些参数名仅在函数体中使用，不能在函数调用时使用。这种类型的参数名被称作 `局部参数名（local parameter name）`，因为它们只能在函数体中使用。

()

外部参数名（External Parameter Names）

有时候，调用函数时，给每个参数命名是非常有用的，因为这些参数名可以指出各个实参的用途是什么。

如果你希望函数的使用者在调用函数时提供参数名字，那就需要给每个参数除了局部参数名外再定义一个 `外部参数名`。外部参数名写在局部参数名之前，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： 如果你提供了外部参数名，那么函数在被调用时，必须使用外部参数名。

以下是个例子，这个函数使用一个 `结合者（joiner）` 把两个字符串联在一起：

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

当你调用这个函数时，这三个字符串的用途是不清楚的：

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

为了让这些字符串的用途更为明显，我们为 join 函数添加外部参数名：

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

在这个版本的 join 函数中，第一个参数有一个叫 string 的外部参数名和 s1 的局部参数名，第二个参数有一个叫 toString 的外部参数名和 s2 的局部参数名，第三个参数有一个叫 withJoiner 的外部参数名和 joiner 的局部参数名。

现在，你可以使用这些外部参数名以一种清晰地方式来调用函数了：

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

使用外部参数名让第二个版本的 join 函数的调用更为有表现力，更为通顺，同时还保持了函数体是可读的和有明确意图的。

> 注意： 当其他人在第一次读你的代码，函数参数的意图显得不明显时，考虑使用外部参数名。如果函数参数名的意图是很明显的，那就不需要定义外部参数名了。

简写外部参数名（Shorthand External Parameter Names）

如果你需要提供外部参数名，但是局部参数名已经定义好了，那么你不需要写两次参数名。相反，只写一次参数名，并用 井号（#） 作为前缀就可以了。这告诉 Swift 使用这个参数名作为局部和外部参数名。

下面这个例子定义了一个叫 containsCharacter 的函数，使用 井号（#） 的方式定义了外部参数名：

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

这样定义参数名，使得函数体更为可读，清晰，同时也可以以一个不含糊的方式被调用：

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

默认参数值（Default Parameter Values）

你可以在函数体中为每个参数定义 `默认值` 。当默认值被定义后，调用这个函数时可以忽略这个参数。

> 注意： 将带有默认值的参数放在函数参数列表的最后。这样可以保证在函数调用时，非默认参数的顺序是一致的，同时使得相同的函数在不同情况下调用时显得更为清晰。

以下是另一个版本的 `join` 函数，其中 `joiner` 有了默认参数值：

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

像第一个版本的 `join` 函数一样，如果 `joiner` 被赋值时，函数将使用这个字符串值来连接两个字符串：

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

当这个函数被调用时，如果 `joiner` 的值没有被指定，函数会使用默认值（" "）：

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

默认值参数的外部参数名（External Names for Parameters with Default Values）

在大多数情况下，给带默认值的参数起一个外部参数名是很有用的。这样可以保证当函数被调用且带默认值的参数被提供值时，实参的意图是明显的。

为了使定义外部参数名更加简单，当你未给带默认值的参数提供外部参数名时，Swift 会自动提供外部名字。此时外部参数名与局部名字是一样的，就像你已经在局部参数名前写了 `井号（#）` 一样。

下面是 `join` 函数的另一个版本，这个版本中并没有为它的参数提供外部参数名，但是 `joiner` 参数依然有外部参数名：

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

在这个例子中，Swift 自动为 `joiner` 提供了外部参数名。因此，当函数调用时，外部参数名必须使用，这样使得参数的用途变得清晰。

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

> 注意： 你可以使用 `下划线（_）` 作为默认值参数的外部参数名，这样可以在调用时不用提供外部参数名。但是给带默认值的参数命名总是更加合适的。

可变参数（Variadic Parameters）

一个 可变参数（variadic parameter） 可以接受一个或多个值。函数调用时，你可以用可变参数来传入不确定数量的输入参数。通过在变量类型名后面加入 （...） 的方式来定义可变参数。

传入可变参数的值在函数体内当做这个类型的一个数组。例如，一个叫做 numbers 的 Double... 型可变参数，在函数体内可以当做一个叫 numbers 的 Double[] 型的数组常量。

下面的这个函数用来计算一组任意长度数字的算术平均数：

```
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： 一个函数至多能有一个可变参数，而且它必须是参数表中最后的一个。这样做是为了避免函数调用时出现歧义。

如果函数有一个或多个带默认值的参数，而且还有一个可变参数，那么把可变参数放在参数表的最后。

()

常量参数和变量参数（Constant and Variable Parameters）

函数参数默认是常量。试图在函数体中更改参数值将会导致编译错误。这意味着你不能错误地更改参数值。

但是，有时候，如果函数中有传入参数的变量值副本将是很有用的。你可以通过指定一个或多个参数为变量参数，从而避免自己在函数中定义新的变量。变量参数不是常量，你可以在函数中把它当做新的可修改副本来使用。

通过在参数名前加关键字 var 来定义变量参数：

```
func alignRight(var string: String, count: Int, pad: Character) -> String {
    let amountToPad = count - countElements(string)
    if amountToPad < 1 {
        return string
    }
    let padString = String(pad)
    for _ in 1...amountToPad {
        string = padString + string
    }
```

```
    return string
}
let originalString = "hello"
let paddedString = alignRight(originalString, 10, "-")
// paddedString is equal to "-----hello"
// originalString is still equal to "hello"
```

这个例子中定义了一个新的叫做 alignRight 的函数，用来右对齐输入的字符串到一个长的输出字符串中。左侧空余的地方用指定的填充字符填充。这个例子中，字符串 "hello" 被转换成了 "-----hello" 。

alignRight 函数将参数 string 定义为变量参数。这意味着 string 现在可以作为一个局部变量，用传入的字符串值初始化，并且可以在函数体中进行操作。

该函数首先计算出多少个字符需要被添加到 string 的左边，以右对齐到总的字符串中。这个值存在局部常量 amountToPad 中。这个函数然后将 amountToPad 多的填充（pad）字符填充到 string 左边，并返回结果。它使用了 string 这个变量参数来进行所有字符串操作。

> 注意： 对变量参数所进行的修改在函数调用结束后便消失了，并且对于函数体外是不可见的。变量参数仅仅存在于函数调用的生命周期中。

输入输出参数（In-Out Parameters）

变量参数，正如上面所述，仅仅能在函数体内被更改。如果你想要一个函数可以修改参数的值，并且想要在这些修改在函数调用结束后仍然存在，那么就应该把这个参数定义为输入输出参数（In-Out Parameters）。

定义一个输入输出参数时，在参数定义前加 inout 关键字。一个输入输出参数有传入函数的值，这个值被函数修改，然后被传出函数，替换原来的值。

你只能将变量作为输入输出参数。你不能传入常量或者字面量（literal value），因为这些量是不能被修改的。当传入的参数作为输入输出参数时，需要在参数前加 & 符，表示这个值可以被函数修改。

> 注意： 输入输出参数不能有默认值，而且可变参数不能用 inout 标记。如果你用 inout 标记一个参数，这个参数不能被 var 或者 let 标记。

下面是例子， swapTwoInts 函数，有两个分别叫做 a 和 b 的输入输出参数：

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

这个 swapTwoInts 函数仅仅交换 a 与 b 的值。该函数先将 a 的值存到一个暂时常量 temporaryA 中，然后将 b 的值赋给 a ，最后将 temporaryA 幅值给 b 。

你可以用两个 Int 型的变量来调用 swapTwoInts 。需要注意的是， someInt 和 anotherInt 在传入 swapTwoInts 函数前，都加了 & 的前缀：

```
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”
```

从上面这个例子中，我们可以看到 someInt 和 anotherInt 的原始值在 swapTwoInts 函数中被修改，尽管它们的定义在函数体外。

> 注意： 输入输出参数和返回值是不一样的。上面的 swapTwoInts 函数并没有定义任何返回值，但仍然修改了 someInt 和 anotherInt 的值。输入输出参数是函数对函数体外产生影响的另一种方式。

函数类型（Function Types）

每个函数都有种特定的函数类型，由函数的参数类型和返回类型组成。

例如：

```
func addTwoInts(a: Int, b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(a: Int, b: Int) -> Int {
    return a * b
}
```

这个例子中定义了两个简单的数学函数： addTwoInts 和 multiplyTwoInts 。这两个函数都传入两个 Int 类型， 返回一个合适的 Int 值。

这两个函数的类型是 (Int, Int) -> Int ，可以读作“这个函数类型，它有两个 Int 型的参数并返回一个 Int 型的值。”。

下面是另一个例子，一个没有参数，也没有返回值的函数：

```
func printHelloWorld() {
    println("hello, world")
}
```

这个函数的类型是： () -> () ，或者叫“没有参数，并返回 Void 类型的函数”。没有指定返回类型的函数总返回 Void 。在Swift中， Void 与空的元组是一样的。

使用函数类型（Using Function Types）

在 Swift 中，使用函数类型就像使用其他类型一样。例如，你可以定义一个类型为函数的常量或变量，并将函数赋值给它：

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

这个可以读作：

“定义一个叫做 `mathFunction` 的变量，类型是‘一个有两个 `Int` 型的参数并返回一个 `Int` 型的值的函数’，并让这个新变量指向 `addTwoInts` 函数”。

`addTwoInts` 和 `mathFunction` 有同样的类型，所以这个赋值过程在 Swift 类型检查中是允许的。

现在，你可以用 `mathFunction` 来调用被赋值的函数了：

```
println("Result: \(mathFunction(2, 3))")
// prints "Result: 5"
```

有相同匹配类型的不同函数可以被赋值给同一个变量，就像非函数类型的变量一样：

```
mathFunction = multiplyTwoInts
println("Result: \(mathFunction(2, 3))")
// prints "Result: 6"
```

就像其他类型一样，当赋值一个函数给常量或变量时，你可以让 Swift 来推断其函数类型：

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

函数类型作为参数类型（Function Types as Parameter Types）

你可以用 `(Int, Int) -> Int` 这样的函数类型作为另一个函数的参数类型。这样你可以将函数的一部分实现交由给函数的调用者。

下面是另一个例子，正如上面的函数一样，同样是输出某种数学运算结果：

```
func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {
    println("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// prints "Result: 8”
```

这个例子定义了 `printMathResult` 函数，它有三个参数：第一个参数叫 `mathFunction`，类型是 `(Int, Int) -> Int`，你可以传入任何这种类型的函数；第二个和第三个参数叫 `a` 和 `b`，它们的类型都是 `Int`，这两个值作为已给的函数的输入值。

当 printMathResult 被调用时，它被传入 addTwoInts 函数和整数 3 和 5 。它用传入 3 和 5 调用 addTwoInts ，并输出结果： 8 。

printMathResult 函数的作用就是输出另一个合适类型的数学函数的调用结果。它不关心传入函数是如何实现的，它只关心这个传入的函数类型是正确的。这使得 printMathResult 可以以一种类型安全（type-safe）的方式来保证传入函数的调用是正确的。

()

函数类型作为返回类型（Function Type as Return Types）

你可以用函数类型作为另一个函数的返回类型。你需要做的是在返回箭头（ -> ）后写一个完整的函数类型。

下面的这个例子中定义了两个简单函数，分别是 stepForward 和 stepBackward 。 stepForward 函数返回一个比输入值大一的值。 stepBackward 函数返回一个比输入值小一的值。这两个函数的类型都是 (Int) -> Int ：

```
func stepForward(input: Int) -> Int {
    return input + 1
}
func stepBackward(input: Int) -> Int {
    return input - 1
}
```

下面这个叫做 chooseStepFunction 的函数，它的返回类型是 (Int) -> Int 的函数。 chooseStepFunction 根据布尔值 backwards 来返回 stepForward 函数或 stepBackward 函数：

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}
```

你现在可以用 chooseStepFunction 来获得一个函数，不管是那个方向：

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

上面这个例子中计算出从 currentValue 逐渐接近到 0 是需要向正数走还是向负数走。 currentValue 的初始值是 3 ，这意味着 currentValue > 0 是真的（ true ），这将使得 chooseStepFunction 返回 stepBackward 函数。一个指向返回的函数的引用保存在了 moveNearerToZero 常量中。

现在， moveNearerToZero 指向了正确的函数，它可以被用来数到 0 ：

```
println("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
```

```
    println("zero!")
    // 3...
    // 2...
    // 1...
    // zero!
```

()

嵌套函数（Nested Functions）

这章中你所见到的所有函数都叫全局函数（global functions），它们定义在全局域中。你也可以把函数定义在别的函数体中，称作嵌套函数（nested functions）。

默认情况下，嵌套函数是对外界不可见的，但是可以被他们封闭函数（enclosing function）来调用。一个封闭函数也可以返回它的某一个嵌套函数，使得这个函数可以在其他域中被使用。

你可以用返回嵌套函数的方式重写 `chooseStepFunction` 函数：

```
    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!
```

7

闭包（Closures）

闭包是自包含的函数代码块，可以在代码中被传递和使用。 Swift 中的闭包与 C 和 Objective-C 中的代码块（blocks）以及其他一些编程语言中的 lambdas 函数比较相似。

闭包可以捕获和存储其所在上下文中任意常量和变量的引用。 这就是所谓的闭合并包裹着这些常量和变量，俗称闭包。Swift 会为您管理在捕获过程中涉及到的所有内存操作。

> 注意： 如果您不熟悉捕获（capturing）这个概念也不用担心，您可以在 值捕获 (页 78) 章节对其进行详细了解。

在函数 () 章节中介绍的全局和嵌套函数实际上也是特殊的闭包，闭包采取如下三种形式之一：

- 全局函数是一个有名字但不会捕获任何值的闭包
- 嵌套函数是一个有名字并可以捕获其封闭函数域内值的闭包
- 闭包表达式是一个利用轻量级语法所写的可以捕获其上下文中变量或常量值的匿名闭包

Swift 的闭包表达式拥有简洁的风格，并鼓励在常见场景中进行语法优化，主要优化如下：

- 利用上下文推断参数和返回值类型
- 隐式返回单表达式闭包，即单表达式闭包可以省略 `return` 关键字
- 参数名称缩写
- 尾随（Trailing）闭包语法

闭包表达式（Closure Expressions）

嵌套函数 () 是一个在较复杂函数中方便进行命名和定义自包含代码模块的方式。当然，有时候撰写小巧的没有完整定义和命名的类函数结构也是很有用处的，尤其是在您处理一些函数并需要将另外一些函数作为该函数的参数时。

闭包表达式是一种利用简洁语法构建内联闭包的方式。 闭包表达式提供了一些语法优化，使得撰写闭包变得简单明了。 下面闭包表达式的例子通过使用几次迭代展示了 `sorted` 函数定义和语法优化的方式。 每一次迭代都用更简洁的方式描述了相同的功能。

sorted 函数（The Sorted Function）

Swift 标准库提供了 `sorted` 函数，会根据您提供的基于输出类型排序的闭包函数将已知类型数组中的值进行排序。 一旦排序完成，函数会返回一个与原数组大小相同的新数组，该数组中包含已经正确排序的同类型元素。

下面的闭包表达式示例使用 `sorted` 函数对一个 `String` 类型的数组进行字母逆序排序，以下是初始数组值：

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

sorted 函数需要传入两个参数：

- 已知类型的数组
- 闭包函数，该闭包函数需要传入与数组类型相同的两个值，并返回一个布尔类型值来告诉 sorted 函数当排序结束后传入的第一个参数排在第二个参数前面还是后面。如果第一个参数值出现在第二个参数值前面，排序闭包函数需要返回 true ，反之返回 false 。

该例子对一个 String 类型的数组进行排序，因此排序闭包函数类型需为 (String, String) -> Bool 。

提供排序闭包函数的一种方式是撰写一个符合其类型要求的普通函数，并将其作为 sort 函数的第二个参数传入：

```
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
var reversed = sorted(names, backwards)
// reversed 为 ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

如果第一个字符串（ s1 ）大于第二个字符串（ s2 ）， backwards 函数返回 true ，表示在新的数组中 s1 应该出现在 s2 前。 对于字符串中的字符来说，“大于” 表示 “按照字母顺序较晚出现”。 这意味着字母 "B" 大于字母 "A" ，字符串 "Tom" 大于字符串 "Tim" 。 其将进行字母逆序排序， "Barry" 将会排在 "Alex" 之前。

然而，这是一个相当冗长的方式，本质上只是写了一个单表达式函数 (a > b)。 在下面的例子中，利用闭合表达式语法可以更好的构造一个内联排序闭包。

闭包表达式语法（Closure Expression Syntax）

闭包表达式语法有如下一般形式：

```
{ (parameters) -> returnType in
    statements
}
```

闭包表达式语法可以使用常量、变量和 inout 类型作为参数，不提供默认值。 也可以在参数列表的最后使用可变参数。 元组也可以作为参数和返回值。

下面的例子展示了之前 backwards 函数对应的闭包表达式版本的代码：

```
reversed = sorted(names, { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

需要注意的是内联闭包参数和返回值类型声明与 backwards 函数类型声明相同。 在这两种方式中，都写成了 (s1: String, s2: String) -> Bool 。 然而在内联闭包表达式中，函数和返回值类型都写在大括号内，而不是大括号外。

闭包的函数体部分由关键字 in 引入。该关键字表示闭包的参数和返回值类型定义已经完成，闭包函数体即将开始。

因为这个闭包的函数体部分如此短以至于可以将其改写成一行代码：

```
reversed = sorted(names, { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

这说明 sorted 函数的整体调用保持不变，一对圆括号仍然包裹住了函数中整个参数集合。而其中一个参数现在变成了内联闭包（相比于 backwards 版本的代码）。

根据上下文推断类型（Inferring Type From Context）

因为排序闭包函数是作为 sorted 函数的参数进行传入的，Swift可以推断其参数和返回值的类型。sorted 期望第二个参数是类型为 (String, String) -> Bool 的函数，因此实际上 String，String 和 Bool 类型并不需要作为闭包表达式定义中的一部分。因为所有的类型都可以被正确推断，返回箭头（->）和围绕在参数周围的括号也可以被省略：

```
reversed = sorted(names, { s1, s2 in return s1 > s2 } )
```

实际上任何情况下，通过内联闭包表达式构造的闭包作为参数传递给函数时，都可以推断出闭包的参数和返回值类型，这意味着您几乎不需要利用完整格式构造任何内联闭包。

单表达式闭包隐式返回（Implicit Return From Single-Expression Clossures）

单行表达式闭包可以通过隐藏 return 关键字来隐式返回单行表达式的结果，如上版本的例子可以改写为：

```
reversed = sorted(names, { s1, s2 in s1 > s2 } )
```

在这个例子中，sorted 函数的第二个参数函数类型明确了闭包必须返回一个 Bool 类型值。因为闭包函数体只包含了一个单一表达式（s1 > s2），该表达式返回 Bool 类型值，因此这里没有歧义，return 关键字可以省略。

参数名称缩写（Shorthand Argument Names）

Swift 自动为内联函数提供了参数名称缩写功能，您可以直接通过 $0，$1，$2 来顺序调用闭包的参数。

如果您在闭包表达式中使用参数名称缩写，您可以在闭包参数列表中省略对其的定义，并且对应参数名称缩写的类型会通过函数类型进行推断。in 关键字也同样可以被省略，因为此时闭包表达式完全由闭包函数体构成：

```
reversed = sorted(names, { $0 > $1 } )
```

在这个例子中，$0 和 $1 表示闭包中第一个和第二个 String 类型的参数。

运算符函数（Operator Functions）

实际上还有一种更简短的方式来撰写上面例子中的闭包表达式。 Swift 的 `String` 类型定义了关于大于号（`>`）的字符串实现，其作为一个函数接受两个 `String` 类型的参数并返回 `Bool` 类型的值。 而这正好与 `sorted` 函数的第二个参数需要的函数类型相符合。 因此，您可以简单地传递一个大于号，Swift可以自动推断出您想使用大于号的字符串函数实现：

```
reversed = sorted(names, >)
```

更多关于运算符表达式的内容请查看 运算符函数 () 。

尾随闭包（Trailing Closures）

如果您需要将一个很长的闭包表达式作为最后一个参数传递给函数，可以使用尾随闭包来增强函数的可读性。 尾随闭包是一个书写在函数括号之后的闭包表达式，函数支持将其作为最后一个参数调用。

```
func someFunctionThatTakesAClosure(closure: () -> ()) {
    // 函数体部分
}

// 以下是不使用尾随闭包进行函数调用
someFunctionThatTakesAClosure({
    // 闭包主体部分
})

// 以下是使用尾随闭包进行函数调用
someFunctionThatTakesAClosure() {
    // 闭包主体部分
}
```

> 注意： 如果函数只需要闭包表达式一个参数，当您使用尾随闭包时，您甚至可以把 `()` 省略掉。

在上例中作为 `sorted` 函数参数的字符串排序闭包可以改写为：

```
reversed = sorted(names) { $0 > $1 }
```

当闭包非常长以至于不能在一行中进行书写时，尾随闭包变得非常有用。 举例来说，Swift 的 `Array` 类型有一个 `map` 方法，其获取一个闭包表达式作为其唯一参数。 数组中的每一个元素调用一次该闭包函数，并返回该元素所映射的值(也可以是不同类型的值)。 具体的映射方式和返回值类型由闭包来指定。

当提供给数组闭包函数后， `map` 方法将返回一个新的数组，数组中包含了与原数组一一对应的映射后的值。

下例介绍了如何在 `map` 方法中使用尾随闭包将 `Int` 类型数组 `[16,58,510]` 转换为包含对应 `String` 类型的数组 `["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]
```

如上代码创建了一个数字位和它们名字映射的英文版本字典。 同时定义了一个准备转换为字符串的整型数组。

您现在可以通过传递一个尾随闭包给 numbers 的 map 方法来创建对应的字符串版本数组。 需要注意的时调用 numbers.map 不需要在 map 后面包含任何括号，因为其只需要传递闭包表达式这一个参数，并且该闭包表达式参数通过尾随方式进行撰写：

```
let strings = numbers.map {
    (var number) -> String in
    var output = ""
    while number > 0 {
        output = digitNames[number % 10]! + output
        number /= 10
    }
    return output
}
// strings 常量被推断为字符串类型数组，即 String[]
// 其值为 ["OneSix", "FiveEight", "FiveOneZero"]
```

map 在数组中为每一个元素调用了闭包表达式。 您不需要指定闭包的输入参数 number 的类型，因为可以通过要映射的数组类型进行推断。

闭包 number 参数被声明为一个变量参数（变量的具体描述请参看常量参数和变量参数 ()），因此可以在闭包函数体内对其进行修改。闭包表达式制定了返回类型为 String ，以表明存储映射值的新数组类型为 String 。

闭包表达式在每次被调用的时候创建了一个字符串并返回。 其使用求余运算符 (number % 10) 计算最后一位数字并利用 digitNames 字典获取所映射的字符串。

> 注意： 字典 digitNames 下标后跟着一个叹号 (!)，因为字典下标返回一个可选值 (optional value)，表明即使该 key 不存在也不会查找失败。 在上例中，它保证了 number % 10 可以总是作为一个 digitNames 字典的有效下标 key。 因此叹号可以用于强制解析 (force-unwrap) 存储在可选下标项中的 String 类型值。

从 digitNames 字典中获取的字符串被添加到输出的前部，逆序建立了一个字符串版本的数字。（在表达式 number % 10 中，如果number为16，则返回6，58返回8，510返回0）。

number 变量之后除以10。 因为其是整数，在计算过程中未除尽部分被忽略。 因此 16变成了1，58变成了5，510变成了51。

整个过程重复进行，直到 number /= 10 为0，这时闭包会将字符串输出，而 map 函数则会将字符串添加到所映射的数组中。

上例中尾随闭包语法在函数后整洁封装了具体的闭包功能，而不再需要将整个闭包包裹在 map 函数的括号内。

()

捕获值（Capturing Values）

闭包可以在其定义的上下文中捕获常量或变量。 即使定义这些常量和变量的原域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

Swift最简单的闭包形式是嵌套函数，也就是定义在其他函数的函数体内的函数。 嵌套函数可以捕获其外部函数所有的参数以及定义的常量和变量。

下例为一个叫做 makeIncrementor 的函数，其包含了一个叫做 incrementor 嵌套函数。 嵌套函数 incrementor 从上下文中捕获了两个值， runningTotal 和 amount 。 之后 makeIncrementor 将 incrementor 作为闭包返回。 每次调用 incrementor 时，其会以 amount 作为增量增加 runningTotal 的值。

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

makeIncrementor 返回类型为 () -> Int 。 这意味着其返回的是一个函数，而不是一个简单类型值。 该函数在每次调用时不接受参数只返回一个 Int 类型的值。 关于函数返回其他函数的内容，请查看函数类型作为返回类型 () 。

makeIncrementor 函数定义了一个整型变量 runningTotal (初始为0) 用来存储当前跑步总数。 该值通过 incrementor 返回。

makeIncrementor 有一个 Int 类型的参数，其外部命名为 forIncrement ， 内部命名为 amount ，表示每次 incrementor 被调用时 runningTotal 将要增加的量。

incrementor 函数用来执行实际的增加操作。 该函数简单地使 runningTotal 增加 amount ，并将其返回。

如果我们单独看这个函数，会发现看上去不同寻常：

```
func incrementor() -> Int {
    runningTotal += amount
    return runningTotal
}
```

incrementor 函数并没有获取任何参数，但是在函数体内访问了 runningTotal 和 amount 变量。这是因为其通过捕获在包含它的函数体内已经存在的 runningTotal 和 amount 变量而实现。

由于没有修改 amount 变量， incrementor 实际上捕获并存储了该变量的一个副本，而该副本随着 incrementor 一同被存储。

然而，因为每次调用该函数的时候都会修改 runningTotal 的值， incrementor 捕获了当前 runningTotal 变量的引用，而不是仅仅复制该变量的初始值。捕获一个引用保证了当 makeIncrementor 结束时候并不会消失，也保证了当下一次执行 incrementor 函数时， runningTotal 可以继续增加。

> 注意： Swift 会决定捕获引用还是拷贝值。 您不需要标注 amount 或者 runningTotal 来声明在嵌入的 incrementor 函数中的使用方式。 Swift 同时也处理 runingTotal 变量的内存管理操作，如果不再被 incrementor 函数使用，则会被清除。

下面代码为一个使用 makeIncrementor 的例子：

```
let incrementByTen = makeIncrementor(forIncrement: 10)
```

该例子定义了一个叫做 incrementByTen 的常量，该常量指向一个每次调用会加10的 incrementor 函数。 调用这个函数多次可以得到以下结果：

```
incrementByTen()
// 返回的值为10
incrementByTen()
// 返回的值为20
incrementByTen()
// 返回的值为30
```

如果您创建了另一个 incrementor ，其会有一个属于自己的独立的 runningTotal 变量的引用。 下面的例子中， incrementBySevne 捕获了一个新的 runningTotal 变量，该变量和 incrementByTen 中捕获的变量没有任何联系：

```
let incrementBySeven = makeIncrementor(forIncrement: 7)
incrementBySeven()
// 返回的值为7
incrementByTen()
// 返回的值为40
```

> 注意： 如果您将闭包赋值给一个类实例的属性，并且该闭包通过指向该实例或其成员来捕获了该实例，您将创建一个在闭包和实例间的强引用环。 Swift 使用捕获列表来打破这种强引用环。更多信息，请参考 闭包引起的循环强引用 () 。

闭包是引用类型（Closures Are Reference Types）

上面的例子中， incrementBySeven 和 incrementByTen 是常量，但是这些常量指向的闭包仍然可以增加其捕获的变量值。 这是因为函数和闭包都是引用类型。

无论您将函数/闭包赋值给一个常量还是变量，您实际上都是将常量/变量的值设置为对应函数/闭包的引用。 上面的例子中， `incrementByTen` 指向闭包的引用是一个常量，而并非闭包内容本身。

这也意味着如果您将闭包赋值给了两个不同的常量/变量，两个值都会指向同一个闭包：

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// 返回的值为50
```

8

枚举（Enumerations）

枚举定义了一个通用类型的一组相关的值，使你可以在你的代码中以一个安全的方式来使用这些值。

如果你熟悉 C 语言，你就会知道，在 C 语言中枚举指定相关名称为一组整型值。Swift 中的枚举更加灵活，不必给每一个枚举成员提供一个值。如果一个值（被认为是“原始”值）被提供给每个枚举成员，则该值可以是一个字符串，一个字符，或是一个整型值或浮点值。

此外，枚举成员可以指定任何类型的相关值存储到枚举成员值中，就像其他语言中的联合体（unions）和变体（variants）。你可以定义一组通用的相关成员作为枚举的一部分，每一组都有不同的一组与它相关的适当类型的数值。

在 Swift 中，枚举类型是一等（first-class）类型。它们采用了很多传统上只被类（class）所支持的特征，例如计算型属性（computed properties），用于提供关于枚举当前值的附加信息， 实例方法（instance methods），用于提供和枚举所代表的值相关联的功能。枚举也可以定义构造函数（initializers）来提供一个初始成员值；可以在原始的实现基础上扩展它们的功能；可以遵守协议（protocols）来提供标准的功能。

欲了解更多相关功能，请参见属性（Properties）()，方法（Methods）()，构造过程（Initialization）()，扩展（Extensions）()和协议（Protocols）()。

枚举语法

使用 enum 关键词并且把它们的整个定义放在一对大括号内：

```
enum SomeEnumeration {
// enumeration definition goes here
}
```

以下是指南针四个方向的一个例子：

```
enum CompassPoint {
    case North
    case South
    case East
    case West
}
```

一个枚举中被定义的值（例如 North，South，East 和 West）是枚举的***成员值***（或者***成员***）。case 关键词表明新的一行成员值将被定义。

> 注意：
> 不像 C 和 Objective-C 一样，Swift 的枚举成员在被创建时不会被赋予一个默认的整数值。在上面的 CompassPoints 例子中，North，South，East 和 West 不是隐式的等于 0，1，2 和 3。相反的，这些不同的枚举成员在 CompassPoint 的一种显示定义中拥有各自不同的值。

多个成员值可以出现在同一行上，用逗号隔开：

```
enum Planet {
    case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

每个枚举定义了一个全新的类型。像 Swift 中其他类型一样，它们的名字（例如 CompassPoint 和 Planet ）必须以一个大写字母开头。给枚举类型起一个单数名字而不是复数名字，以便于读起来更加容易理解：

```
var directionToHead = CompassPoint.West
```

directionToHead 的类型被推断当它被 CompassPoint 的一个可能值初始化。一旦 directionToHead 被声明为一个 CompassPoint ，你可以使用更短的点（.）语法将其设置为另一个 CompassPoint 的值：

```
directionToHead = .East
```

directionToHead 的类型已知时，当设定它的值时，你可以不再写类型名。使用显式类型的枚举值可以让代码具有更好的可读性。

匹配枚举值和 Switch 语句

你可以匹配单个枚举值和 switch 语句：

```
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")
}
// 输出 "Watch out for penguins”
```

你可以如此理解这段代码：

“考虑 directionToHead 的值。当它等于 .North ，打印 “Lots of planets have a north” 。当它等于 .South ，打印 “Watch out for penguins” 。”

等等依次类推。

正如在控制流（Control Flow）() 中介绍，当考虑一个枚举的成员们时，一个 switch 语句必须全面。如果忽略了 .West 这种情况，上面那段代码将无法通过编译，因为它没有考虑到 CompassPoint 的全部成员。全面性的要求确保了枚举成员不会被意外遗漏。

当不需要匹配每个枚举成员的时候，你可以提供一个默认 default 分支来涵盖所有未明确被提出的任何成员：

```
let somePlanet = Planet.Earth
switch somePlanet {
```

```
    case .Earth:
        println("Mostly harmless")
    default:
        println("Not a safe place for humans")
    }
    // 输出 "Mostly harmless”
```

()

相关值（Associated Values）

上一小节的例子演示了一个枚举的成员是如何被定义（分类）的。你可以为 Planet.Earth 设置一个常量或则变量，并且在之后查看这个值。不管怎样，如果有时候能够把其他类型的相关值和成员值一起存储起来会很有用。这能让你存储成员值之外的自定义信息，并且当你每次在代码中使用该成员时允许这个信息产生变化。

你可以定义 Swift 的枚举存储任何类型的相关值，如果需要的话，每个成员的数据类型可以是各不相同的。枚举的这种特性跟其他语言中的可辨识联合（discriminated unions），标签联合（tagged unions），或者变体（variants）相似。

例如，假设一个库存跟踪系统需要利用两种不同类型的条形码来跟踪商品。有些商品上标有 UPC-A 格式的一维码，它使用数字 0 到 9。每一个条形码都有一个代表“数字系统”的数字，该数字后接 10 个代表“标识符”的数字。最后一个数字是“检查”位，用来验证代码是否被正确扫描：

![Image of Enumerations_1.png]
(images/Enumerations_1.png)

其他商品上标有 QR 码格式的二维码，它可以使用任何 ISO8859-1 字符，并且可以编码一个最多拥有 2,953 字符的字符串：

![Image of Enumerations_2.png]
(images/Enumerations_2.png)

对于库存跟踪系统来说，能够把 UPC-A 码作为三个整型值的元组，和把 QR 码作为一个任何长度的字符串存储起来是方便的。

在 Swift 中，用来定义两种商品条码的枚举是这样子的：

```
enum Barcode {
    case UPCA(Int, Int, Int)
    case QRCode(String)
}
```

以上代码可以这么理解：

“定义一个名为 Barcode 的枚举类型，它可以是 UPCA 的一个相关值（ Int ， Int ， Int ），或者 QRCode 的一个字符串类型（ String ）相关值。”

这个定义不提供任何 Int 或 String 的实际值，它只是定义了，当 Barcode 常量和变量等于 Barcode.UPCA 或 Barcode.QRCode 时，相关值的类型。

然后可以使用任何一种条码类型创建新的条码，如：

```
var productBarcode = Barcode.UPCA(8, 85909_51226, 3)
```

以上例子创建了一个名为 productBarcode 的新变量，并且赋给它一个 Barcode.UPCA 的相关元组值 (8, 8590 951226, 3) 。提供的“标识符”值在整数字中有一个下划线，使其便于阅读条形码。

同一个商品可以被分配给一个不同类型的条形码，如：

```
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

这时，原始的 Barcode.UPCA 和其整数值被新的 Barcode.QRCode 和其字符串值所替代。条形码的常量和变量可以存储一个 .UPCA 或者一个 .QRCode （连同它的相关值），但是在任何指定时间只能存储其中之一。

像以前那样，不同的条形码类型可以使用一个 switch 语句来检查，然而这次相关值可以被提取作为 switch 语句的一部分。你可以在 switch 的 case 分支代码中提取每个相关值作为一个常量（用 let 前缀）或者作为一个变量（用 var 前缀）来使用：

```
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).")
}
// 输出 "QR code with value of ABCDEFGHIJKLMNOP.”
```

如果一个枚举成员的所有相关值被提取为常量，或者它们全部被提取为变量，为了简洁，你可以只放置一个 var 或者 let 标注在成员名称前：

```
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).")
}
// 输出 "QR code with value of ABCDEFGHIJKLMNOP."
```

原始值（Raw Values）

在相关值 (页 84)小节的条形码例子中演示了一个枚举的成员如何声明它们存储不同类型的相关值。作为相关值的替代，枚举成员可以被默认值（称为原始值）预先填充，其中这些原始值具有相同的类型。

这里是一个枚举成员存储原始 ASCII 值的例子：

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

在这里，称为 ASCIIControlCharacter 的枚举的原始值类型被定义为字符型 Character，并被设置了一些比较常见的 ASCII 控制字符。字符值的描述请详见字符串和字符 Strings and Characters () 部分。

注意，原始值和相关值是不相同的。当你开始在你的代码中定义枚举的时候原始值是被预先填充的值，像上述三个 ASCII 码。对于一个特定的枚举成员，它的原始值始终是相同的。相关值是当你在创建一个基于枚举成员的新常量或变量时才会被设置，并且每次当你这么做得时候，它的值可以是不同的。

原始值可以是字符串，字符，或者任何整型值或浮点型值。每个原始值在它的枚举声明中必须是唯一的。当整型值被用于原始值，如果其他枚举成员没有值时，它们会自动递增。

下面的枚举是对之前 Planet 这个枚举的一个细化，利用原始整型值来表示每个 planet 在太阳系中的顺序：

```
enum Planet: Int {
    case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

自动递增意味着 Planet.Venus 的原始值是 2，依次类推。

使用枚举成员的 rawValue 属性可以访问该枚举成员的原始值：

```
let earthsOrder = Planet.Earth.rawValue
// earthsOrder is 3
```

通过参数为 rawValue 构造函数创建特定原始值的枚举。这个例子通过原始值 7 识别 Uranus：

```
let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet is of type Planet? and equals Planet.Uranus
```

然而，并非所有可能的 Int 值都可以找到一个匹配的行星。正因为如此，构造函数可以返回一个***可选***的枚举成员。在上面的例子中， possiblePlanet 是 Planet? 类型，或“可选的 Planet ”。

如果你试图寻找一个位置为9的行星，通过参数为 rawValue 构造函数返回的可选 Planet 值将是 nil：

```
let positionToFind = 9
if let somePlanet = Planet(rawValue: 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)")
```

```
}
// 输出 "There isn't a planet at position 9
```

这个范例使用可选绑定（optional binding），通过原始值 `9` 试图访问一个行星。`if let somePlanet = Planet(rawValue: 9)` 语句获得一个可选 `Planet`，如果可选 `Planet` 可以被获得，把 `somePlanet` 设置成该可选 `Planet` 的内容。在这个范例中，无法检索到位置为 `9` 的行星，所以 `else` 分支被执行。

9

类和结构体

类和结构体是人们构建代码所用的一种通用且灵活的构造体。为了在类和结构体中实现各种功能，我们必须要严格按照常量、变量以及函数所规定的语法规则来定义属性和添加方法。

与其他编程语言所不同的是，Swift 并不要求你为自定义类和结构去创建独立的接口和实现文件。你所要做的是在一个单一文件中定义一个类或者结构体，系统将会自动生成面向其它代码的外部接口。

> 注意： 通常一个 `类` 的实例被称为 `对象` 。然而在Swift 中，类和结构体的关系要比在其他语言中更加的密切，本章中所讨论的大部分功能都可以用在类和结构体上。因此，我们会主要使用 `实例` 而不是 `对象` 。

类和结构体对比

Swift 中类和结构体有很多共同点。共同处在于：

- 定义属性用于存储值
- 定义方法用于提供功能
- 定义附属脚本用于访问值
- 定义构造器用于生成初始化值
- 通过扩展以增加默认实现的功能
- 符合协议以对某类提供标准功能

更多信息请参见 属性 () ，方法 () ，下标脚本 () ，初始过程 () ，扩展 () ，和协议 () 。

与结构体相比，类还有如下的附加功能：

- 继承允许一个类继承另一个类的特征
- 类型转换允许在运行时检查和解释一个类实例的类型
- 解构器允许一个类实例释放任何其所被分配的资源
- 引用计数允许对一个类的多次引用

更多信息请参见继承 () ，类型转换 () ，初始化 () ，和自动引用计数 () 。

> 注意： 结构体总是通过被复制的方式在代码中传递，因此请不要使用引用计数。

定义

类和结构体有着类似的定义方式。我们通过关键字 `class` 和 `struct` 来分别表示类和结构体，并在一对大括号中定义它们的具体内容：

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

> 注意： 在你每次定义一个新类或者结构体的时候，实际上你是有效地定义了一个新的 Swift 类型。因此请使用 UpperCamelCase 这种方式来命名（如 SomeClass 和 SomeStructure 等），以便符合标准Swift 类型的大写命名风格（如 String， Int 和 Bool ）。相反的，请使用 lowerCamelCase 这种方式为属性和方法命名（如 framerate 和 incrementCount ），以便和类区分。

以下是定义结构体和定义类的示例：

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

在上面的示例中我们定义了一个名为 Resolution 的结构体，用来描述一个显示器的像素分辨率。这个结构体包含了两个名为 width 和 height 的存储属性。存储属性是捆绑和存储在类或结构体中的常量或变量。当这两个属性被初始化为整数 0 的时候，它们会被推断为 Int 类型。

在上面的示例中我们还定义了一个名为 VideoMode 的类，用来描述一个视频显示器的特定模式。这个类包含了四个储存属性变量。第一个是 分辨率 ，它被初始化为一个新的 Resolution 结构体的实例，具有 Resolution 的属性类型。新 VideoMode 实例同时还会初始化其它三个属性，它们分别是，初始值为 false (意为“non-interlaced video”)的 interlaced ，回放帧率初始值为 0.0 的 frameRate 和值为可选 String 的 name 。 name 属性会被自动赋予一个默认值 nil ，意为“没有 name 值”，因为它是一个可选类型。

类和结构体实例

Resolution 结构体和 VideoMode 类的定义仅描述了什么是 Resolution 和 VideoMode 。它们并没有描述一个特定的分辨率（resolution）或者视频模式（video mode）。为了描述一个特定的分辨率或者视频模式，我们需要生成一个它们的实例。

生成结构体和类实例的语法非常相似：

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

结构体和类都使用构造器语法来生成新的实例。构造器语法的最简单形式是在结构体或者类的类型名称后跟随一个空括弧，如 Resolution() 或 VideoMode() 。通过这种方式所创建的类或者结构体实例，其属性均会被初始化为默认值。构造过程 () 章节会对类和结构体的初始化进行更详细的讨论。

属性访问

通过使用*点语法*（*dot syntax*），你可以访问实例中所含有的属性。其语法规则是，实例名后面紧跟属性名，两者通过点号(.)连接：

```
println("The width of someResolution is \(someResolution.width)")
// 输出 "The width of someResolution is 0"
```

在上面的例子中， someResolution.width 引用 someResolution 的 width 属性，返回 width 的初始值 0 。

你也可以访问子属性，如 VideoMode 中 Resolution 属性的 width 属性：

```
println("The width of someVideoMode is \(someVideoMode.resolution.width)")
// 输出 "The width of someVideoMode is 0"
```

你也可以使用点语法为属性变量赋值：

```
someVideoMode.resolution.width = 1280
println("The width of someVideoMode is now \(someVideoMode.resolution.width)")
// 输出 "The width of someVideoMode is now 1280"
```

> 注意： 与 Objective-C 语言不同的是，Swift 允许直接设置结构体属性的子属性。上面的最后一个例子，就是直接设置了 someVideoMode 中 resolution 属性的 width 这个子属性，以上操作并不需要重新设置 resolution 属性。

结构体类型的成员逐一构造器(Memberwise Initializers for structure Types)

所有结构体都有一个自动生成的成员逐一构造器，用于初始化新结构体实例中成员的属性。新实例中各个属性的初始值可以通过属性的名称传递到成员逐一构造器之中：

```
let vga = Resolution(width:640, height: 480)
```

与结构体不同，类实例没有默认的成员逐一构造器。构造过程 () 章节会对构造器进行更详细的讨论。

()

结构体和枚举是值类型

值类型被赋予给一个变量，常数或者本身被传递给一个函数的时候，实际上操作的是其的拷贝。

在之前的章节中，我们已经大量使用了值类型。实际上，在 Swift 中，所有的基本类型：整数（Integer）、浮点数（floating-point）、布尔值（Booleans）、字符串（string)、数组（array）和字典（dictionaries），都是值类型，并且都是以结构体的形式在后台所实现。

在 Swift 中，所有的结构体和枚举都是值类型。这意味着它们的实例，以及实例中所包含的任何值类型属性，在代码中传递的时候都会被复制。

请看下面这个示例，其使用了前一个示例中 Resolution 结构体：

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

在以上示例中，声明了一个名为 hd 的常量，其值为一个初始化为全高清视频分辨率（1920 像素宽，1080 像素高）的 Resolution 实例。

然后示例中又声明了一个名为 cinema 的变量，其值为之前声明的 hd 。因为 Resolution 是一个结构体，所以 cinema 的值其实是 hd 的一个拷贝副本，而不是 hd 本身。尽管 hd 和 cinema 有着相同的宽（width）和高（height）属性，但是在后台中，它们是两个完全不同的实例。

下面，为了符合数码影院放映的需求（2048 像素宽，1080 像素高）， cinema 的 width 属性需要作如下修改：

```
cinema.width = 2048
```

这里，将会显示 cinema 的 width 属性确已改为了 2048 ：

```
println("cinema is now  \(cinema.width) pixels wide")
// 输出 "cinema is now 2048 pixels wide"
```

然而，初始的 hd 实例中 width 属性还是 1920 ：

```
println("hd is still \(hd.width    ) pixels wide")
// 输出 "hd is still 1920 pixels wide"
```

在将 hd 赋予给 cinema 的时候，实际上是将 hd 中所存储的 值（values） 进行拷贝，然后将拷贝的数据存储到新的 cinema 实例中。结果就是两个完全独立的实例碰巧包含有相同的数值。由于两者相互独立，因此将 cinema 的 width 修改为 2048 并不会影响 hd 中的宽（width）。

枚举也遵循相同的行为准则：

```
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")
}
// 输出 "The remembered direction is still .West"
```

上例中 rememberedDirection 被赋予了 currentDirection 的值（value），实际上它被赋予的是值（value）的一个拷贝。赋值过程结束后再修改 currentDirection 的值并不影响 rememberedDirection 所储存的原始值（value）的拷贝。

类是引用类型

与值类型不同，引用类型在被赋予到一个变量、常量或者被传递到一个函数时，操作的是引用，其并不是拷贝。因此，引用的是已存在的实例本身而不是其拷贝。

请看下面这个示例，其使用了之前定义的 VideoMode 类：

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

以上示例中，声明了一个名为 tenEighty 的常量，其引用了一个 VideoMode 类的新实例。在之前的示例中，这个视频模式（video mode）被赋予了HD分辨率（1920*1080）的一个拷贝（ hd ）。同时设置为交错（interlaced）,命名为 “1080i” 。最后，其帧率是 25.0 帧每秒。

然后， tenEighty 被赋予名为 alsoTenEighty 的新常量，同时对 alsoTenEighty 的帧率进行修改：

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

因为类是引用类型，所以 tenEight 和 alsoTenEight 实际上引用的是相同的 VideoMode 实例。换句话说，它们是同一个实例的两种叫法。

下面，通过查看 tenEighty 的 frameRate 属性，我们会发现它正确的显示了基本 VideoMode 实例的新帧率，其值为 30.0 ：

```
println("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// 输出 "The frameRate property of theEighty is now 30.0"
```

需要注意的是 tenEighty 和 alsoTenEighty 被声明为*常量（（constants）*而不是变量。然而你依然可以改变 tenEighty.frameRate 和 alsoTenEighty.frameRate ,因为这两个常量本身不会改变。它们并不 存储 这个 VideoMode 实例，在后台仅仅是对 VideoMode 实例的引用。所以，改变的是被引用的基础 VideoMode 的 frameRate 参数，而不改变常量的值。

恒等运算符

因为类是引用类型，有可能有多个常量和变量在后台同时引用某一个类实例。（对于结构体和枚举来说，这并不成立。因为它们作为值类型，在被赋予到常量、变量或者传递到函数时，其值总是会被拷贝。）

如果能够判定两个常量或者变量是否引用同一个类实例将会很有帮助。为了达到这个目的，Swift 内建了两个恒等运算符：

- 等价于（ === ）
- 不等价于（ !== ）

以下是运用这两个运算符检测两个常量或者变量是否引用同一个实例：

```
if tenEighty === alsoTenTighty {
    println("tenTighty and alsoTenEighty refer to the same Resolution instance.")
}
//输出 "tenEighty and alsoTenEighty refer to the same Resolution instance."
```

请注意 “等价于"（用三个等号表示，===）与 “等于"（用两个等号表示，==）的不同：

- “等价于”表示两个类类型（class type）的常量或者变量引用同一个类实例。
- “等于”表示两个实例的值“相等”或“相同”，判定时要遵照类设计者定义定义的评判标准，因此相比于“相等”，这是一种更加合适的叫法。

当你在定义你的自定义类和结构体的时候，你有义务来决定判定两个实例“相等”的标准。在章节运算符函数(Operator Functions) () 中将会详细介绍实现自定义“等于”和“不等于”运算符的流程。

指针

如果你有 C，C++ 或者 Objective-C 语言的经验，那么你也许会知道这些语言使用指针来引用内存中的地址。一个 Swift 常量或者变量引用一个引用类型的实例与 C 语言中的指针类似，不同的是并不直接指向内存中的某个地址，而且也不要求你使用星号（*）来表明你在创建一个引用。Swift 中这些引用与其它的常量或变量的定义方式相同。

类和结构体的选择

在你的代码中，你可以使用类和结构体来定义你的自定义数据类型。

然而，结构体实例总是通过值传递，类实例总是通过引用传递。这意味两者适用不同的任务。当你在考虑一个工程项目的数据构造和功能的时候，你需要决定每个数据构造是定义成类还是结构体。

按照通用的准则，当符合一条或多条以下条件时，请考虑构建结构体：

- 结构体的主要目的是用来封装少量相关简单数据值。
- 有理由预计一个结构体实例在赋值或传递时，封装的数据将会被拷贝而不是被引用。
- 任何在结构体中储存的值类型属性，也将会被拷贝，而不是被引用。
- 结构体不需要去继承另一个已存在类型的属性或者行为。

合适的结构体候选者包括：

- 几何形状的大小，封装一个 width 属性和 height 属性，两者均为 Double 类型。
- 一定范围内的路径，封装一个 start 属性和 length 属性，两者均为 Int 类型。
- 三维坐标系内一点，封装 x ， y 和 z 属性，三者均为 Double 类型。

在所有其它案例中，定义一个类，生成一个它的实例，并通过引用来管理和传递。实际中，这意味着绝大部分的自定义数据构造都应该是类，而非结构体。

集合（Collection）类型的赋值和拷贝行为

Swift 中 字符串（String），数组（Array）和 字典（Dictionary）类型均以结构体的形式实现。这意味着String，Array，Dictionary类型数据被赋值给新的常量(或变量），或者被传入函数（或方法）中时，它们的值会发生拷贝行为（值传递方式）。

Objective-C中 字符串（NSString），数组（NSArray）和 字典（NSDictionary） 类型均以类的形式实现，这与Swfit中以值传递方式是不同的。NSString，NSArray，NSDictionary在发生赋值或者传入函数（或方法）时，不会发生值拷贝，而是传递已存在实例的引用。

> 注意： 以上是对于数组，字典，字符串和其它值的 拷贝 的描述。 在你的代码中，拷贝好像是确实是在有拷贝行为的地方产生过。然而，在 Swift 的后台中，只有确有必要， 实际（actual） 拷贝才会被执行。Swift 管理所有的值拷贝以确保性能最优化的性能，所以你也没有必要去避免赋值以保证最优性能。（实际赋值由系统管理优化）

10

屬性 (Properties)

属性将值跟特定的类、结构或枚举关联。存储属性存储常量或变量作为实例的一部分，计算属性计算（而不是存储）一个值。计算属性可以用于类、结构体和枚举里，存储属性只能用于类和结构体。

存储属性和计算属性通常用于特定类型的实例，但是，属性也可以直接用于类型本身，这种属性称为类型属性。

另外，还可以定义属性观察器来监控属性值的变化，以此来触发一个自定义的操作。属性观察器可以添加到自己写的存储属性上，也可以添加到从父类继承的属性上。

存储属性

简单来说，一个存储属性就是存储在特定类或结构体的实例里的一个常量或变量，存储属性可以是*变量存储属性*（用关键字 `var` 定义），也可以是*常量存储属性*（用关键字 `let` 定义）。

可以在定义存储属性的时候指定默认值，请参考构造过程 () 一章的默认属性值 () 一节。也可以在构造过程中设置或修改存储属性的值，甚至修改常量存储属性的值，请参考构造过程 () 一章的构造过程中常量属性的修改 () 一节。

下面的例子定义了一个名为 `FixedLengthRange` 的结构体，它描述了一个在创建后无法修改值域宽度的区间：

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// 该区间表示整数0，1，2
rangeOfThreeItems.firstValue = 6
// 该区间现在表示整数6，7，8
```

`FixedLengthRange` 的实例包含一个名为 `firstValue` 的变量存储属性和一个名为 `length` 的常量存储属性。在上面的例子中，`length` 在创建实例的时候被赋值，因为它是一个常量存储属性，所以之后无法修改它的值。

()

常量和存储属性

如果创建了一个结构体的实例并赋值给一个常量，则无法修改实例的任何属性，即使定义了变量存储属性：

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// 该区间表示整数0，1，2，3
rangeOfFourItems.firstValue = 6
// 尽管 firstValue 是个变量属性，这里还是会报错
```

因为 `rangeOfFourItems` 声明成了常量（用 `let` 关键字），即使 `firstValue` 是一个变量属性，也无法再修改它了。

这种行为是由于结构体（struct）属于*值类型*。当值类型的实例被声明为常量的时候，它的所有属性也就成了常量。

属于*引用类型*的类（class）则不一样，把一个引用类型的实例赋给一个常量后，仍然可以修改实例的变量属性。

()

延迟存储属性

延迟存储属性是指当第一次被调用的时候才会计算其初始值的属性。在属性声明前使用 lazy 来标示一个延迟存储属性。

> 注意：
> 必须将延迟存储属性声明成变量（使用 var 关键字），因为属性的值在实例构造完成之前可能无法得到。而常量属性在构造过程完成之前必须要有初始值，因此无法声明成延迟属性。

延迟属性很有用，当属性的值依赖于在实例的构造过程结束前无法知道具体值的外部因素时，或者当属性的值需要复杂或大量计算时，可以只在需要的时候来计算它。

下面的例子使用了延迟存储属性来避免复杂类的不必要的初始化。例子中定义了 DataImporter 和 DataManager 两个类，下面是部分代码：

```
class DataImporter {
    /*
    DataImporter 是一个将外部文件中的数据导入的类。
    这个类的初始化会消耗不少时间。
    */
    var fileName = "data.txt"
    // 这是提供数据导入功能
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
    // 这是提供数据管理功能
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
// DataImporter 实例的 importer 属性还没有被创建
```

DataManager 类包含一个名为 data 的存储属性，初始值是一个空的字符串（ String ）数组。虽然没有写出全部代码， DataManager 类的目的是管理和提供对这个字符串数组的访问。

DataManager 的一个功能是从文件导入数据，该功能由 DataImporter 类提供， DataImporter 需要消耗不少时间完成初始化：因为它的实例在初始化时可能要打开文件，还要读取文件内容到内存。

DataManager 也可能不从文件中导入数据。所以当 DataManager 的实例被创建时，没必要创建一个 DataImporter 的实例，更明智的是当用到 DataImporter 的时候才去创建它。

由于使用了 lazy ， importer 属性只有在第一次被访问的时候才被创建。比如访问它的属性 fileName 时：

```
println(manager.importer.fileName)
// DataImporter 实例的 importer 属性现在被创建了
// 输出 "data.txt”
```

存储属性和实例变量

如果您有过 Objective-C 经验，应该知道Objective-C为类实例存储值和引用提供两种方法。对于属性来说，也可以使用实例变量作为属性值的后端存储。

Swift 编程语言中把这些理论统一用属性来实现。Swift 中的属性没有对应的实例变量，属性的后端存储也无法直接访问。这就避免了不同场景下访问方式的困扰，同时也将属性的定义简化成一个语句。 一个类型中属性的全部信息——包括命名、类型和内存管理特征——都在唯一一个地方（类型定义中）定义。

计算属性

除存储属性外，类、结构体和枚举可以定义*计算属性*，计算属性不直接存储值，而是提供一个 getter 来获取值，一个可选的 setter 来间接设置其他属性或变量的值。

```
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))")
// 输出 "square.origin is now at (10.0, 10.0)”
```

这个例子定义了 3 个几何形状的结构体：

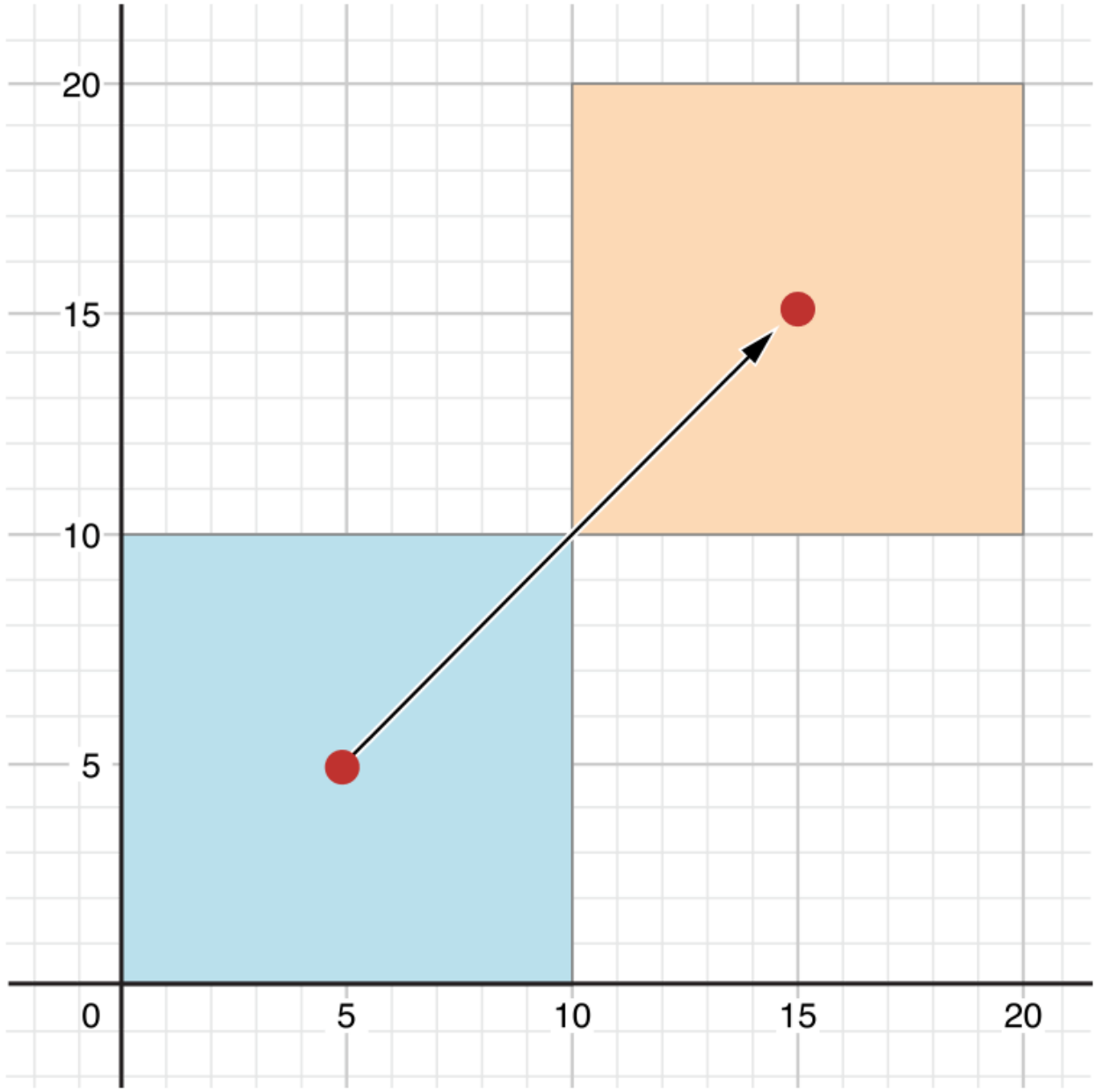
- `Point` 封装了一个 `(x, y)` 的坐标
- `Size` 封装了一个 `width` 和 `height`
- `Rect` 表示一个有原点和尺寸的矩形

`Rect` 也提供了一个名为 `center` 的计算属性。一个矩形的中心点可以从原点和尺寸来算出，所以不需要将它以显式声明的 `Point` 来保存。`Rect` 的计算属性 `center` 提供了自定义的 getter 和 setter 来获取和设置矩形的中心点，就像它有一个存储属性一样。

例子中接下来创建了一个名为 `square` 的 `Rect` 实例，初始值原点是 `(0, 0)`，宽度高度都是 `10`。如图所示蓝色正方形。

`square` 的 `center` 属性可以通过点运算符（`square.center`）来访问，这会调用 getter 来获取属性的值。跟直接返回已经存在的值不同，getter 实际上通过计算然后返回一个新的 `Point` 来表示 `square` 的中心点。如代码所示，它正确返回了中心点 `(5, 5)`。

`center` 属性之后被设置了一个新的值 `(15, 15)`，表示向右上方移动正方形到如图所示橙色正方形的位置。设置属性 `center` 的值会调用 setter 来修改属性 `origin` 的 `x` 和 `y` 的值，从而实现移动正方形到新的位置。



图片 10.1 Image of Properties_1.png

便捷 setter 声明

如果计算属性的 setter 没有定义表示新值的参数名，则可以使用默认名称 `newValue` 。下面是使用了便捷 setter 声明的 `Rect` 结构体代码：

```
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)
    }
    }
}
```

只读计算属性

只有 getter 没有 setter 的计算属性就是*只读计算属性*。只读计算属性总是返回一个值，可以通过点运算符访问，但不能设置新的值。

> 注意：
>
> 必须使用 `var` 关键字定义计算属性，包括只读计算属性，因为它们的值不是固定的。`let` 关键字只用来声明常量属性，表示初始化后再也无法修改的值。

只读计算属性的声明可以去掉 `get` 关键字和花括号：

```
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)")
// 输出 "the volume of fourByFiveByTwo is 40.0"
```

这个例子定义了一个名为 `Cuboid` 的结构体，表示三维空间的立方体，包含 `width` 、 `height` 和 `depth` 属性，还有一个名为 `volume` 的只读计算属性用来返回立方体的体积。设置 `volume` 的值毫无意义，因为通过 `width` 、 `height` 和 `depth` 就能算出 `volume` 。然而， `Cuboid` 提供一个只读计算属性来让外部用户直接获取体积是很有用的。

()

属性观察器

*属性观察器*监控和响应属性值的变化，每次属性被设置值的时候都会调用属性观察器，甚至新的值和现在的值相同的时候也不例外。

可以为除了延迟存储属性之外的其他存储属性添加属性观察器，也可以通过重载属性的方式为继承的属性（包括存储属性和计算属性）添加属性观察器。属性重载请参考继承 () 一章的重载 () 。

> 注意：
> 不需要为无法重载的计算属性添加属性观察器，因为可以通过 setter 直接监控和响应值的变化。

可以为属性添加如下的一个或全部观察器：

- willSet 在设置新的值之前调用
- didSet 在新的值被设置之后立即调用

willSet 观察器会将新的属性值作为固定参数传入，在 willSet 的实现代码中可以为这个参数指定一个名称，如果不指定则参数仍然可用，这时使用默认名称 newValue 表示。

类似地， didSet 观察器会将旧的属性值作为参数传入，可以为该参数命名或者使用默认参数名 oldValue 。

> 注意：
>
> willSet 和 didSet 观察器在属性初始化过程中不会被调用，它们只会当属性的值在初始化之外的地方被设置时被调用。

这里是一个 willSet 和 didSet 的实际例子，其中定义了一个名为 StepCounter 的类，用来统计当人步行时的总步数，可以跟计步器或其他日常锻炼的统计装置的输入数据配合使用。

```
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 类定义了一个 Int 类型的属性 totalSteps ，它是一个存储属性，包含 willSet 和 didSet 观察器。

当 totalSteps 设置新值的时候，它的 willSet 和 didSet 观察器都会被调用，甚至当新的值和现在的值完全相同也会调用。

例子中的 willSet 观察器将表示新值的参数自定义为 newTotalSteps ，这个观察器只是简单的将新的值输出。

didSet 观察器在 totalSteps 的值改变后被调用，它把新的值和旧的值进行对比，如果总的步数增加了，就输出一个消息表示增加了多少步。didSet 没有提供自定义名称，所以默认值 oldValue 表示旧值的参数名。

> 注意：
> 如果在 didSet 观察器里为属性赋值，这个值会替换观察器之前设置的值。

全局变量和局部变量

计算属性和属性观察器所描述的模式也可以用于*全局变量*和*局部变量*，全局变量是在函数、方法、闭包或任何类型之外定义的变量，局部变量是在函数、方法或闭包内部定义的变量。

前面章节提到的全局或局部变量都属于存储型变量，跟存储属性类似，它提供特定类型的存储空间，并允许读取和写入。

另外，在全局或局部范围都可以定义计算型变量和为存储型变量定义观察器，计算型变量跟计算属性一样，返回一个计算的值而不是存储值，声明格式也完全一样。

> 注意：
> 全局的常量或变量都是延迟计算的，跟延迟存储属性 (页 98)相似，不同的地方在于，全局的常量或变量不需要标记 lazy 特性。
> 局部范围的常量或变量不会延迟计算。

类型属性

实例的属性属于一个特定类型实例，每次类型实例化后都拥有自己的一套属性值，实例之间的属性相互独立。

也可以为类型本身定义属性，不管类型有多少个实例，这些属性都只有唯一一份。这种属性就是*类型属性*。

类型属性用于定义特定类型所有实例共享的数据，比如所有实例都能用的一个常量（就像 C 语言中的静态常量），或者所有实例都能访问的一个变量（就像 C 语言中的静态变量）。

对于值类型（指结构体和枚举）可以定义存储型和计算型类型属性，对于类（class）则只能定义计算型类型属性。

值类型的存储型类型属性可以是变量或常量，计算型类型属性跟实例的计算属性一样定义成变量属性。

> 注意：
> 跟实例的存储属性不同，必须给存储型类型属性指定默认值，因为类型本身无法在初始化过程中使用构造器给类型属性赋值。

类型属性语法

在 C 或 Objective-C 中，静态常量和静态变量的定义是通过特定类型加上 global 关键字。在 Swift 编程语言中，类型属性是作为类型定义的一部分写在类型最外层的花括号内，因此它的作用范围也就在类型支持的范围内。

使用关键字 static 来定义值类型的类型属性，关键字 class 来为类（class）定义类型属性。下面的例子演示了存储型和计算型类型属性的语法：

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
    // 这里返回一个 Int 值
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
    // 这里返回一个 Int 值
    }
}
class SomeClass {
    class var computedTypeProperty: Int {
    // 这里返回一个 Int 值
    }
}
```

> 注意：
> 例子中的计算型类型属性是只读的，但也可以定义可读可写的计算型类型属性，跟实例计算属性的语法类似。

获取和设置类型属性的值

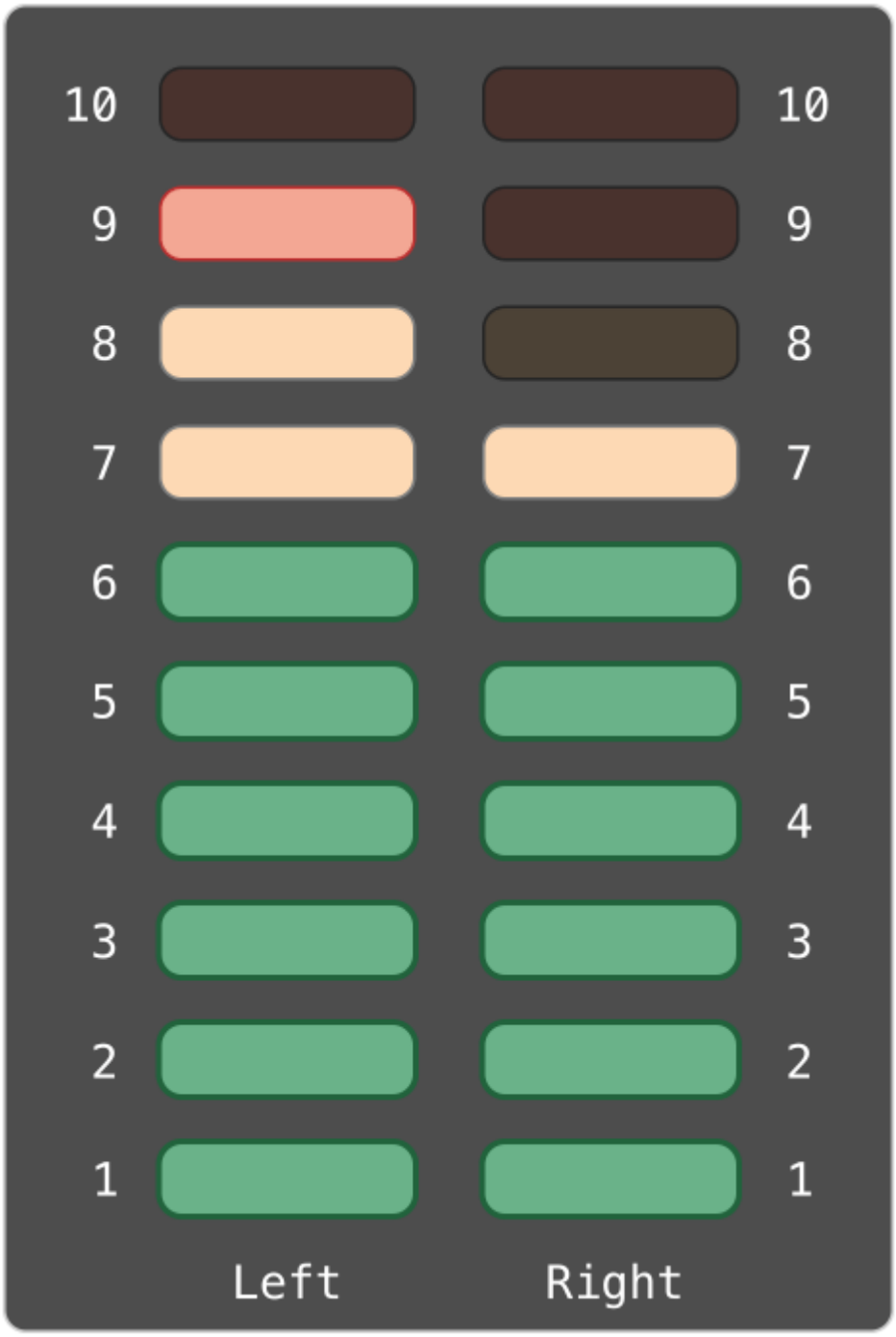
跟实例的属性一样，类型属性的访问也是通过点运算符来进行，但是，类型属性是通过类型本身来获取和设置，而不是通过实例。比如：

```
println(SomeClass.computedTypeProperty)
// 输出 "42"

println(SomeStructure.storedTypeProperty)
// 输出 "Some value."
SomeStructure.storedTypeProperty = "Another value."
println(SomeStructure.storedTypeProperty)
// 输出 "Another value.”
```

下面的例子定义了一个结构体，使用两个存储型类型属性来表示多个声道的声音电平值，每个声道有一个 0 到 10 之间的整数表示声音电平值。

后面的图表展示了如何联合使用两个声道来表示一个立体声的声音电平值。当声道的电平值是 0，没有一个灯会亮；当声道的电平值是 10，所有灯点亮。本图中，左声道的电平是 9，右声道的电平是 7。



图片 10.2 Image of Properties_2.png

上面所描述的声道模型使用 `AudioChannel` 结构体来表示：

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
```

```
    var currentLevel: Int = 0 {
    didSet {
        if currentLevel > AudioChannel.thresholdLevel {
            // 将新电平值设置为阀值
            currentLevel = AudioChannel.thresholdLevel
        }
        if currentLevel > AudioChannel.maxInputLevelForAllChannels {
            // 存储当前电平值作为新的最大输入电平
            AudioChannel.maxInputLevelForAllChannels = currentLevel
        }
    }
    }
}
```

结构 AudioChannel 定义了 2 个存储型类型属性来实现上述功能。第一个是 thresholdLevel ，表示声音电平的最大上限阈值，它是一个取值为 10 的常量，对所有实例都可见，如果声音电平高于 10，则取最大上限值 10（见后面描述）。

第二个类型属性是变量存储型属性 maxInputLevelForAllChannels ，它用来表示所有 AudioChannel 实例的电平值的最大值，初始值是 0。

AudioChannel 也定义了一个名为 currentLevel 的实例存储属性，表示当前声道现在的电平值，取值为 0 到 10。

属性 currentLevel 包含 didSet 属性观察器来检查每次新设置后的属性值，有如下两个检查：

- 如果 currentLevel 的新值大于允许的阈值 thresholdLevel ，属性观察器将 currentLevel 的值限定为阈值 thresholdLevel 。
- 如果修正后的 currentLevel 值大于任何之前任意 AudioChannel 实例中的值，属性观察器将新值保存在静态属性 maxInputLevelForAllChannels 中。

> 注意：
> 在第一个检查过程中， didSet 属性观察器将 currentLevel 设置成了不同的值，但这时不会再次调用属性观察器。

可以使用结构体 AudioChannel 来创建表示立体声系统的两个声道 leftChannel 和 rightChannel ：

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

如果将左声道的电平设置成 7，类型属性 maxInputLevelForAllChannels 也会更新成 7：

```
leftChannel.currentLevel = 7
println(leftChannel.currentLevel)
// 输出 "7"
println(AudioChannel.maxInputLevelForAllChannels)
// 输出 "7"
```

如果试图将右声道的电平设置成 11，则会将右声道的 currentLevel 修正到最大值 10，同时 maxInputLevelForAllChannels 的值也会更新到 10：

```
rightChannel.currentLevel = 11
println(rightChannel.currentLevel)
// 输出 "10"
println(AudioChannel.maxInputLevelForAllChannels)
// 输出 "10"
```

11

方法 – Methods

方法是与特定类型相关联的函数。类、结构体以及枚举均可以定义实例方法，该方法为指定类型的实例封装了特定的任务与功能。类、结构体以及枚举也能定义类型方法，该方法与类型自身相关联。类型方法类似于在 Objective-C 中的类方法。

在 Swift 中，结构体和枚举能够定义方法；事实上这是 Swift 与 C/Objective-C 的主要区别之一。在 Objective-C 中，类是唯一能定义方法的类型。在 Swift 中，你可以选择是否定义一个类、结构体或枚举，且仍可以灵活地对你所创建的类型进行方法的定义。

实例方法

实例方法是某个特定类、结构体或枚举类型的实例的方法。他们通过提供访问的方式和修改实例属性，或提供与实例目的相关的功能性来支持这些实例的功能性。准确的来讲，实例方法的语法与函数完全一致，参考函数 () 说明。

实例方法要写在它所属的类型的前后括号之间。实例方法能够访问他所属类型的所有的其他实例方法和属性。实例方法只能被它所属的类的特定实例调用。实例方法不能被孤立于现存的实例而被调用。

下面定义一个简单的类 `Counter` 的示例（Counter 可以用来对一个动作发生的次数进行计数）：

```
class Counter {
    var count = 0
    func increment() {
        count++
    }
    func incrementBy(amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

类 `Counter` 可以定义三种实例方法：

- `increment` 让计数器按一递增
- `incrementBy(amount: Int)` 让计数器按一个指定的整数值递增
- `reset` 将计数器重置为 0

`Counter` 这个类还声明了一个可变属性 `count`，用它来保持对当前计数器值的追踪。

和调用属性一样，用点语法调用实例方法：

```
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.incrementBy(5)
// the counter's value is now 6
counter.reset()
// the counter's value is now 0
```

方法的局部参数名称和外部参数名称

函数参数有一个局部名称（在函数体内部使用）和一个外部名称（在调用函数时使用），参考外部参数名称 () 。对方法参数也是一样的，因为方法仅仅是与某一类型相关的函数。但是，局部名称和外部名称的默认行为不同于函数和方法。

在 Swift 中的方法和在 Objective-C 中的方法极其相似，像在 Objective-C 一样，在 Swift 中方法的名称名称通常用一个介词指向方法的第一个参数，比如：`with`、`for` 以及 `by` 等等，前面的 `Counter` 类的例子中 incrementBy 方法就是这样的。当其被访问时，介词的使用使方法可被解读为一个句子介词的使用让方法在被调用时能像一个句子一样被解读。Swift 这种方法命名约定很容易落实，因为它是用不同的默认处理方法参数的方式，而不是用函数参数（来实现的）。

具体来说，Swift 默认仅给方法的第一个参数名称一个局部参数名称；但是默认同时给第二个和后续的参数名称局部参数名称和外部参数名称。这个约定与典型的命名和调用约定相匹配，这与你在写 Objective-C 的方法时很相似。这个约定还让 expressive method 调用不需要再检查/限定参数名。

看看下面这个 `Counter` 的替换版本（它定义了一个更复杂的 `incrementBy` 方法）：

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

`incrementBy` 方法有两个参数：`amount` 和 `numberOfTimes`。默认地，Swift 仅把 `amount` 当做一个局部名称，但是把 `numberOfTimes` 既看作局部名称又看做外部名称。调用方法如下：

```
let counter = Counter()
counter.incrementBy(5, numberOfTimes: 3)
// counter value is now 15
```

你不必对第一个参数值进行外部参数名称的定义，因为从函数名 `incrementBy` 已经能很清楚地看出它的目的/作用。但是，第二个参数，就要被一个外部参数名称所限定，以便在方法被调用时明确它的作用。

这种默认的行为能够有效的检查方法，比如你在参数 `numberOfTimes` 前写了个井号（ `#` ）时：

```
func incrementBy(amount: Int, #numberOfTimes: Int) {
    count += amount * numberOfTimes
}
```

这种默认行为使上面代码意味着：在 Swift 中定义方法使用了与 Objective-C 同样的语法风格，并且方法将以自然表达式的方式被调用。

修改方法的外部参数名称

有时，对一个方法的第一个参数提供一个外部参数名是有用的，即使这不是默认行为。你可以自己添加一个明确的外部名称或你也可以用一个 hash 符号作为第一个参数的前缀，然后用这个局部名字作为外部名字。

相反，若你不想为方法的第二或后续参数提供一个外部名称，你可以通过使用下划线 (_) 作为该参数的显式外部名称来覆盖默认行为。

self 属性

类型的每一实例都有一个被称为 `self` 的隐含属性，该属性完全等同于该实例本身。可以在一个实例的实例方法中使用这个隐含的 `self` 属性来引用当前实例。

在上面的例子中， `increment` 方法也可以被写成这样：

```
func increment() {
    self.count++
}
```

实际上，你不必在你的代码里面经常写 `self` 。不论何时，在一个方法中使用一个已知的属性或者方法名称，如果你没有明确的写 `self` ，Swift 假定你是指当前实例的属性或者方法。这种假定在上面的 `Counter` 中已经示范了： `Counter` 中的三个实例方法中都使用的是 `count` (而不是 `self.count`)。

这条规则的主要例外发生在当实例方法的某个参数名称与实例的某个属性名称相同时。在这种情况下，参数名称享有优先权，并且在引用属性时必须使用一种更恰当(被限定更严格)的方式。你可以使用隐藏的 `self` 属性来区分参数名称和属性名称。

下面的例子演示 `self` 消除方法参数x和实例属性 `x` 之间的歧义：

```
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")
prints "This point is to the right of the line where x == 1.0"
```

如果不使用 self 前缀， Swift 就认为两次使用的 x 都指的是名称为 x 的函数参数。

在实例方法中修改值类型

结构体和枚举均属于值类型一般情况下，值类型的属性不能在其实例方法内被修改。

但是，如果在某个具体方法中，你需要对结构体或枚举的属性进行修改，你可以选择变异（mutating）这个方法。方法可以从内部变异它的属性；并且它做的任何改变在方法结束时都会回写到原始结构。方法会给它隐含的 self 属性赋值一个全新的实例，这个新实例在方法结束后将替换原来的实例。

对于变异方法，将关键字 mutating 放到方法的 func 关键字之前就可以了：

```
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))")
// 输出 "The point is now at (3.0, 4.0)"
```

上文 Point 结构体定义一个变异（mutating）方法 moveByx，该方法按一定的数量移动 Point 结构体。moveByX 方法在被调用时修改了这个 point，而不是返回一个新的 point。方法定义是加上 mutating 关键字，因此，方法可以修改值类型的属性。

注意：不能在结构体类型常量上调用变异方法，因为常量的属性不能被改变，即使想改变的是常量的变量属性也不行，详情参见存储属性和实例变量 ()：

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveByX(2.0, y: 3.0)
// this will report an error
```

在变异方法中给 self 赋值

变异方法可以赋予隐含属性 `self` 一个全新的实例。上述 `Point` 的示例也可以采用下面的方式来改写：

```
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)
    }
}
```

新版的变异方法 `moveByX` 创建了一个新的分支结构(他的 x 和 y 的值都被设定为目标值了)。调用这个版本的方法和调用上个版本的最终结果是一样的。

枚举的变异方法可以让 `self` 从相同的枚举设置为不同的成员：

```
enum TriStateSwitch {
    case Off, Low, High
    mutating func next() {
        switch self {
        case Off:
            self = Low
        case Low:
            self = High
        case High:
            self = Off
    }
ovenLight = TriStateSwitch.Low
ovenLight.next()
// ovenLight is now equal to .High
ovenLight.next()
// ovenLight is now equal to .Off
```

上述示例中定义了一个三态开关的枚举。每次调用 `next` 方法时，开关在不同的电源状态(`Off` , `Low` , `Hig h`)之前循环切换。

类型方法

如上所述，实例方法是被类型的某个实例调用的方法。你也可以定义调用类型本身的方法。这种方法就叫做类型方法。声明类的类型方法，在方法的 func 关键字之前加上关键字 `class`；声明结构体和枚举的类型方法，在方法的 `func` 关键字之前加上关键字 `static`。

> 注：在 Objective-C 中，你可以定义仅适用于 Objective-C 的类的 type-evel 方法。在 Swift，你可以对所有的类，结构体和枚举进行 type-evel 方法定义。每种类型方法仅适用于其所支持的类型。

与实例方法相同，也可利用点语法调用类型方法。但是，你需要在本类型上调用类型方法，而不是其类型实例上。下面是如何在 `SomeClass` 类上调用类型方法的示例：

```
class SomeClass {
    class func someTypeMethod() {
    // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

在一个类型方法的方法体内，`self` 指向该类型本身，而不是类型的某个实例。对结构体和枚举来讲，这意味着你可以用 `self` 来消除静态属性和静态方法参数之间的二意性（类似于我们在前面处理实例属性和实例方法参数时做的那样）。

更广泛地说，你在一个类型方法主体内使用的任何未经限定的方法和属性名称将指的是其他 type-level 方法和属性。一种类型方法能用其他方法名称调用另一种类型方法，而无需在方法名称前面加上类型名称的前缀。同样，结构体和枚举的类型方法也能够通过使用不带有类型名称前缀的静态属性名称来访问静态属性。

下述示例中定义名为 `LevelTracker` 的结构体，该结构体通过不同级别或阶段的游戏对玩家的进度进行监测。这是一个单人游戏，但也能在单个设备上储存多个玩家的信息。

所有游戏级别（除了级别 1）在游戏初始时都被锁定。每当玩家完成一个级别，在该设备上，该级别对所有的玩家解锁。`LevelTracker` 结构体用静态属性和方法来监测解锁的游戏级别。也可监测每个玩家的当前等级。

```
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` 结构体监测任何玩家已解锁的最高级别。该值被存储在成为 `highestUnlockedLevel` 的静态属性中。

`LeveTracker` 还定义了两个类型函数与 `highestUnlockedLevel` 配合工作。第一个为 `unockLevel` 类型函数，一旦新的级别被解锁，该函数会更新 `highestUnlockedLevel` 的值。第二个为 `levelIsUnocked` 便利型函数，若某个给定级别数已经被解锁，该函数则返回 `true` 。（注：没用使用 `LevelTracker.highestUnlockedLevel` ，这个类型方法还是能够访问静态属性 `highestUnlockedLevel` 。）

除了其静态属性和类型方法之外， `LeveTracker` 还监测每个玩家的游戏进程。它使用 `currentLevel` 实例属性来监测玩家当前进行的级别。

为便于管理 `currentLevel` 属性， `LeveTracker` 定义了实例方法 `advanceToLevel` 。在更新 `currentLevel` 之前，该方法检查所要求的新级别是否已经解锁。 `advanceToLevel` 方法返回布尔值以指示是否确实能够设置 `currentLevel` 。

下面， `Player` 类使用 `LevelTracker` 来监测和更新每个玩家的发展进度：

```
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` 类使用 `LevelTracker` 来监测该玩家的游戏进程。它也提供 `competedLevel` 方法，一旦玩家完成某个指定等级，调用该方法。该方法为所有玩家解锁下一个级别并将当前玩家进程更新为下一个级别。（忽略了 `advanceToLevel` 布尔返回值，因为之前调用上行时就知道了这个等级已经被解锁了。）

你可以为一个新玩家创建一个 `Player` 类的实例，然后看这个玩家完成等级一时发生了什么：

```
var player = Player(name: "Argyrios")
player.completedLevel(1)
println("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// prints "highest unlocked level is now 2"
```

如果你创建了第二个玩家，并尝试让他开始一个没有被任何玩家解锁的等级，关于设置玩家当前等级的尝试会失败：

```
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")
}
// prints "level 6 has not yet been unlocked"
```

12

下标脚本（Subscripts）

下标脚本 可以定义在类（Class）、结构体（structure）和枚举（enumeration）这些目标中，可以认为是访问对象、集合或序列的快捷方式，不需要再调用实例的特定的赋值和访问方法。举例来说，用下标脚本访问一个数组(Array)实例中的元素可以这样写 `someArray[index]` ，访问字典(Dictionary)实例中的元素可以这样写 `someDictionary[key]` 。

对于同一个目标可以定义多个下标脚本，通过索引值类型的不同来进行重载，而且索引值的个数可以是多个。

> 译者：这里附属脚本重载在本小节中原文并没有任何演示

下标脚本语法

下标脚本允许你通过在实例后面的方括号中传入一个或者多个的索引值来对实例进行访问和赋值。语法类似于实例方法和计算型属性的混合。与定义实例方法类似，定义下标脚本使用 `subscript` 关键字，显式声明入参（一个或多个）和返回类型。与实例方法不同的是下标脚本可以设定为读写或只读。这种方式又有点像计算型属性的getter和setter：

```
subscript(index: Int) -> Int {
    get {
        // 返回与入参匹配的Int类型的值
    }
    set(newValue) {
        // 执行赋值操作
    }
}
```

`newValue` 的类型必须和下标脚本定义的返回类型相同。与计算型属性相同的是set的入参声明 `newValue` 就算不写，在set代码块中依然可以使用默认的 `newValue` 这个变量来访问新赋的值。

与只读计算型属性一样，可以直接将原本应该写在 `get` 代码块中的代码写在 `subscript` 中：

```
subscript(index: Int) -> Int {
    // 返回与入参匹配的Int类型的值
}
```

下面代码演示了一个在 `TimesTable` 结构体中使用只读下标脚本的用法，该结构体用来展示传入整数的*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])")
// 输出 "3的6倍是18"
```

在上例中，通过 `TimesTable` 结构体创建了一个用来表示索引值三倍的实例。数值 `3` 作为结构体 `构造函数` 入参初始化实例成员 `multiplier` 。

你可以通过下标脚本来得到结果，比如 threeTimesTable[6] 。这条语句访问了 threeTimesTable 的第六个元素，返回 6 的 3 倍即 18 。

> 注意：
> TimesTable 例子是基于一个固定的数学公式。它并不适合开放写权限来对 threeTimesTable[someIndex] 进行赋值操作，这也是为什么附属脚本只定义为只读的原因。

下标脚本用法

根据使用场景不同下标脚本也具有不同的含义。通常下标脚本是用来访问集合（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类或结构体中自由的实现下标脚本来提供合适的功能。

例如，Swift 的字典（Dictionary）实现了通过下标脚本来对其实例中存放的值进行存取操作。在下标脚本中使用和字典索引相同类型的值，并且把一个字典值类型的值赋值给这个下标脚本来为字典设值：

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

上例定义一个名为 numberOfLegs 的变量并用一个字典字面量初始化出了包含三对键值的字典实例。 numberOfLegs 的字典存放值类型推断为 Dictionary<String, Int> 。字典实例创建完成之后通过下标脚本的方式将整型值 2 赋值到字典实例的索引为 bird 的位置中。

更多关于字典（Dictionary）下标脚本的信息请参考读取和修改字典 ()

> 注意：
> Swift 中字典的附属脚本实现中，在 get 部分返回值是 Int? ，上例中的 numberOfLegs 字典通过附属脚本返回的是一个 Int? 或者说“可选的int”，不是每个字典的索引都能得到一个整型值，对于没有设过值的索引的访问返回的结果就是 nil ；同样想要从字典实例中删除某个索引下的值也只需要给这个索引赋值为 nil 即可。

下标脚本选项

下标脚本允许任意数量的入参索引，并且每个入参类型也没有限制。下标脚本的返回值也可以是任何类型。下标脚本可以使用变量参数和可变参数，但使用写入读出（in-out）参数或给参数设置默认值都是不允许的。

一个类或结构体可以根据自身需要提供多个下标脚本实现，在定义下标脚本时通过入参个类型进行区分，使用下标脚本时会自动匹配合适的下标脚本实现运行，这就是*下标脚本的重载*。

一个下标脚本入参是最常见的情况，但只要有合适的场景也可以定义多个下标脚本入参。如下例定义了一个 Matrix 结构体，将呈现一个 Double 类型的二维矩阵。 Matrix 结构体的下标脚本需要两个整型参数：

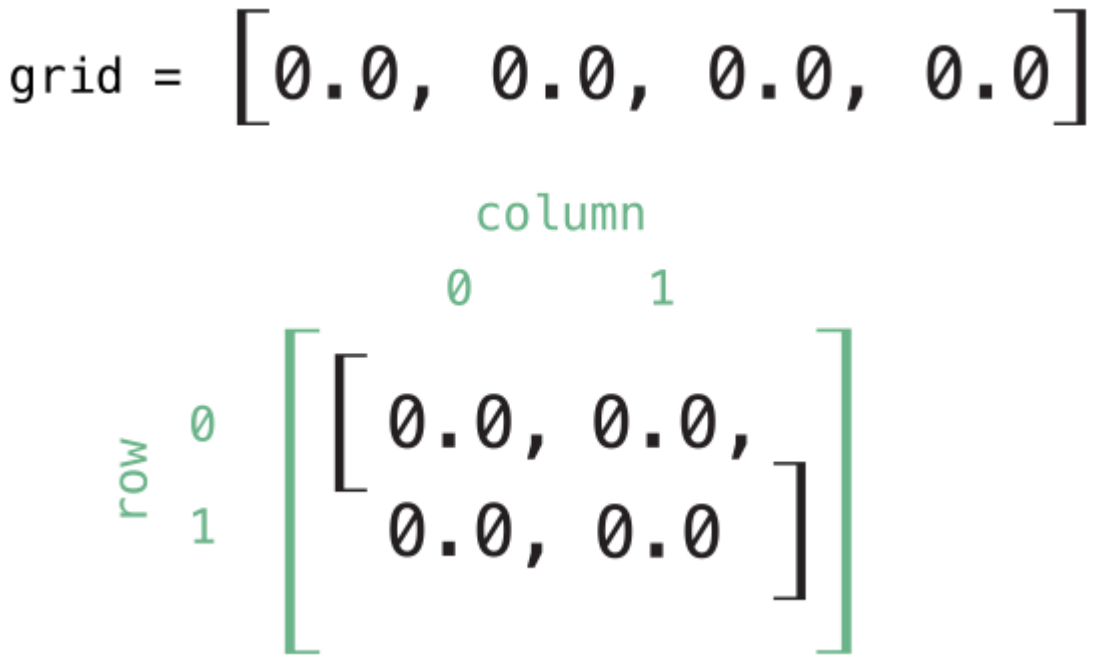
```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func indexIsValidForRow(row: Int, column: Int) -> Bool {
        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) + column] = newValue
        }
    }
    }
```

Matrix 提供了一个两个入参的构造方法，入参分别是 rows 和 columns ，创建了一个足够容纳 rows * columns 个数的 Double 类型数组。为了存储，将数组的大小和数组每个元素初始值0.0，都传入数组的构造方法中来创建一个正确大小的新数组。关于数组的构造方法和析构方法请参考创建并且构造一个数组 () 。

你可以通过传入合适的 row 和 column 的数量来构造一个新的 Matrix 实例：

```
var matrix = Matrix(rows: 2, columns: 2)
```

上例中创建了一个新的两行两列的 Matrix 实例。在阅读顺序从左上到右下的 Matrix 实例中的数组实例 grid 是矩阵二维数组的扁平化存储：

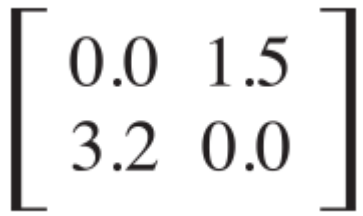


图片 12.1 Image of Subscripts_1.png

将值赋给带有 row 和 column 下标脚本的 matrix 实例表达式可以完成赋值操作，下标脚本入参使用逗号分割

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

上面两条语句分别 让matrix 的右上值为 1.5，坐下值为 3.2：



图片 12.2 Image of Subscripts_2.png

Matrix 下标脚本的 getter 和 setter 中同时调用了下标脚本入参的 row 和 column 是否有效的判断。为了方便进行断言， Matrix 包含了一个名为 indexIsValid 的成员方法，用来确认入参的 row 或 column 值是否会造成数组越界：

```
func indexIsValidForRow(row: Int, column: Int) -> Bool {
    return row >= 0 && row < rows && column >= 0 && column < columns
}
```

断言在下标脚本越界时触发：

```
let someValue = matrix[2, 2]
// 断言将会触发，因为 [2, 2] 已经超过了matrix的最大长度
```

13

继承（Inheritance）

一个类可以*继承（inherit）*另一个类的方法（methods），属性（property）和其它特性。当一个类继承其它类时，继承类叫*子类（subclass）*，被继承类叫*超类（或父类，superclass）*。在 Swift 中，继承是区分「类」与其它类型的一个基本特征。

在 Swift 中，类可以调用和访问超类的方法，属性和下标脚本（subscripts），并且可以重写（override）这些方法，属性和下标脚本来优化或修改它们的行为。Swift 会检查你的重写定义在超类中是否有匹配的定义，以此确保你的重写行为是正确的。

可以为类中继承来的属性添加属性观察器（property observer），这样一来，当属性值改变时，类就会被通知到。可以为任何属性添加属性观察器，无论它原本被定义为存储型属性（stored property）还是计算型属性（computed property）。

定义一个基类（Base class）

不继承于其它类的类，称之为*基类（base calss）*。

> 注意：
> Swift 中的类并不是从一个通用的基类继承而来。如果你不为你定义的类指定一个超类的话，这个类就自动成为基类。

下面的例子定义了一个叫 `Vehicle` 的基类。这个基类声明了一个名为 `currentSpeed`，默认值是0.0的存储属性(属性类型推断为 `Double`)。`currentSpeed` 属性的值被一个 `String` 类型的只读计算型属性 `description` 使用，用来创建车辆的描述。

`Vehicle` 基类也定义了一个名为 `makeNoise` 的方法。这个方法实际上不为 `Vehicle` 实例做任何事，但之后将会被 `Vehicle` 的子类定制

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // 什么也不做-因为车辆不一定会有噪音
    }
}
```

您可以用初始化语法创建一个 `Vehicle` 的新实例，即 `TypeName` 后面跟一个空括号：

```
let someVehicle = Vehicle()
```

现在已经创建了一个 `Vehicle` 的新实例，你可以访问它的 `description` 属性来打印车辆的当前速度。

```
println("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

子类生成（Subclassing）

*子类生成（Subclassing）*指的是在一个已有类的基础上创建一个新的类。子类继承超类的特性，并且可以优化或改变它。你还可以为子类添加新的特性。

为了指明某个类的超类，将超类名写在子类名的后面，用冒号分隔：

```
class SomeClass: SomeSuperclass {
    // 类的定义
}
```

下一个例子，定义一个更具体的车辆类叫 Bicycle 。这个新类是在 Vehicle 类的基础上创建起来。因此你需要将 Vehicle 类放在 Bicycle 类后面，用冒号分隔。

我们可以将这读作：

“定义一个新的类叫 Bicycle ，它继承了 Vehicle 的特性”；

```
class Bicycle: Vehicle {
    var hasBasket = false
}
```

新的 Bicycle 类自动获得 Vehicle 类的所有特性，比如 currentSpeed 和 description 属性，还有它的 makeNoise 方法。

除了它所继承的特性， Bicycle 类还定义了一个默认值为 false 的存储型属性 hasBasket （属性推断为 Bool ）。

默认情况下，你创建任何新的 Bicycle 实例将不会有一个篮子，创建该实例之后，你可以为特定的 Bicycle 实例设置 hasBasket 属性为 ture ：

```
let bicycle = Bicycle()
bicycle.hasBasket = true
```

你还可以修改 Bicycle 实例所继承的 currentSpeed 属性，和查询实例所继承的 description 属性：

```
bicycle.currentSpeed = 15.0
println("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour
```

子类还可以继续被其它类继承，下面的示例为 Bicycle 创建了一个名为 Tandem （双人自行车）的子类：

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
```

Tandem 从 Bicycle 继承了所有的属性与方法，这又使它同时继承了 Vehicle 的所有属性与方法。 Tandem 也增加了一个新的叫做 currentNumberOfPassengers 的存储型属性，默认值为0。

如果你创建了一个 Tandem 的实例，你可以使用它所有的新属性和继承的属性，还能查询从 Vehicle 继承来的只读属性 description：

```
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
println("Tandem: \(tandem.description)")
// Tandem: traveling at 22.0 miles per hour
```

()

重写（Overriding）

子类可以为继承来的实例方法（instance method），类方法（class method），实例属性（instance property），或下标脚本（subscript）提供自己定制的实现（implementation）。我们把这种行为叫*重写（overriding）*。

如果要重写某个特性，你需要在重写定义的前面加上 override 关键字。这么做，你就表明了你是想提供一个重写版本，而非错误地提供了一个相同的定义。意外的重写行为可能会导致不可预知的错误，任何缺少 override 关键字的重写都会在编译时被诊断为错误。

override 关键字会提醒 Swift 编译器去检查该类的超类（或其中一个父类）是否有匹配重写版本的声明。这个检查可以确保你的重写定义是正确的。

访问超类的方法，属性及下标脚本

当你在子类中重写超类的方法，属性或下标脚本时，有时在你的重写版本中使用已经存在的超类实现会大有裨益。比如，你可以优化已有实现的行为，或在一个继承来的变量中存储一个修改过的值。

在合适的地方，你可以通过使用 super 前缀来访问超类版本的方法，属性或下标脚本：

- 在方法 someMethod 的重写实现中，可以通过 super.someMethod() 来调用超类版本的 someMethod 方法。
- 在属性 someProperty 的 getter 或 setter 的重写实现中，可以通过 super.someProperty 来访问超类版本的 someProperty 属性。
- 在下标脚本的重写实现中，可以通过 super[someIndex] 来访问超类版本中的相同下标脚本。

重写方法

在子类中，你可以重写继承来的实例方法或类方法，提供一个定制或替代的方法实现。

下面的例子定义了 `Vehicle` 的一个新的子类，叫 `Train` ，它重写了从 `Vehicle` 类继承来的 `makeNoise` 方法：

```
class Train: Vehicle {
    override func makeNoise() {
        println("Choo Choo")
    }
}
```

如果你创建一个 `Train` 的新实例，并调用了它的 `makeNoise` 方法，你就会发现 `Train` 版本的方法被调用：

```
let train = Train()
train.makeNoise()
// prints "Choo Choo"
```

重写属性

你可以重写继承来的实例属性或类属性，提供自己定制的getter和setter，或添加属性观察器使重写的属性观察属性值什么时候发生改变。

重写属性的Getters和Setters

你可以提供定制的 getter（或 setter）来重写任意继承来的属性，无论继承来的属性是存储型的还是计算型的属性。子类并不知道继承来的属性是存储型的还是计算型的，它只知道继承来的属性会有一个名字和类型。你在重写一个属性时，必需将它的名字和类型都写出来。这样才能使编译器去检查你重写的属性是与超类中同名同类型的属性相匹配的。

你可以将一个继承来的只读属性重写为一个读写属性，只需要你在重写版本的属性里提供 getter 和 setter 即可。但是，你不可以将一个继承来的读写属性重写为一个只读属性。

> 注意：
> 如果你在重写属性中提供了 setter，那么你也一定要提供 getter。如果你不想在重写版本中的 getter 里修改继承来的属性值，你可以直接通过 `super.someProperty` 来返回继承来的值，其中 `someProperty` 是你要重写的属性的名字。

以下的例子定义了一个新类，叫 `Car` ，它是 `Vehicle` 的子类。这个类引入了一个新的存储型属性叫做 `gear` ，默认为整数1。 `Car` 类重写了继承自 `Vehicle` 的description属性，提供自定义的，包含当前档位的描述：

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
```

```
    }
}
```

重写的 description 属性，首先要调用 super.description 返回 Vehicle 类的 description 属性。之后， Car 类版本的 description 在末尾增加了一些额外的文本来提供关于当前档位的信息。

如果你创建了 Car 的实例并且设置了它的 gear 和 currentSpeed 属性，你可以看到它的 description 返回了 Car 中定义的 description ：

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
println("Car: \(car.description)")
// Car: traveling at 25.0 miles per hour in gear 3
```

重写属性观察器（Property Observer）

你可以在属性重写中为一个继承来的属性添加属性观察器。这样一来，当继承来的属性值发生改变时，你就会被通知到，无论那个属性原本是如何实现的。关于属性观察器的更多内容，请看属性观察器 () 。

> 注意：
> 你不可以为继承来的常量存储型属性或继承来的只读计算型属性添加属性观察器。这些属性的值是不可以被设置的，所以，为它们提供 willSet 或 didSet 实现是不恰当。此外还要注意，你不可以同时提供重写的 setter 和重写的属性观察器。如果你想观察属性值的变化，并且你已经为那个属性提供了定制的 setter，那么你在 setter 中就可以观察到任何值变化了。

下面的例子定义了一个新类叫 AutomaticCar ，它是 Car 的子类。 AutomaticCar 表示自动挡汽车，它可以根据当前的速度自动选择合适的挡位:

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

当你设置 AutomaticCar 的 currentSpeed 属性，属性的 didSet 观察器就会自动地设置 gear 属性，为新的速度选择一个合适的挡位。具体来说就是，属性观察器将新的速度值除以10，然后向下取得最接近的整数值，最后加1来得到档位 gear 的值。例如，速度为10.0时，挡位为1；速度为35.0时，挡位为4：

```
let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
println("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

防止重写

你可以通过把方法，属性或下标脚本标记为 *final* 来防止它们被重写，只需要在声明关键字前加上 @final 特性即可。（例如：final var，final func，final class func，以及 final subscript）

如果你重写了 final 方法，属性或下标脚本，在编译时会报错。在扩展中，你添加到类里的方法，属性或下标脚本也可以在扩展的定义里标记为 final。

你可以通过在关键字 class 前添加 final 特性（final class）来将整个类标记为 final 的，这样的类是不可被继承的，否则会报编译错误。

14

构造过程（Initialization）

构造过程是为了使用某个类、结构体或枚举类型的实例而进行的准备过程。这个过程包含了为实例中的每个属性设置初始值和为其执行必要的准备和初始化任务。

构造过程是通过定义构造器（ Initializers ）来实现的，这些构造器可以看做是用来创建特定类型实例的特殊方法。与 Objective-C 中的构造器不同，Swift 的构造器无需返回值，它们的主要任务是保证新实例在第一次使用前完成正确的初始化。

类实例也可以通过定义析构器（ deinitializer ）在类实例释放之前执行特定的清除工作。想了解更多关于析构器的内容，请参考析构过程 () 。

存储型属性的初始赋值

类和结构体在实例创建时，必须为所有存储型属性设置合适的初始值。存储型属性的值不能处于一个未知的状态。

你可以在构造器中为存储型属性赋初值，也可以在定义属性时为其设置默认值。以下章节将详细介绍这两种方法。

> 注意：
> 当你为存储型属性设置默认值或者在构造器中为其赋值时，它们的值是被直接设置的，不会触发任何属性观测器（ property observers ）。

构造器

构造器在创建某特定类型的新实例时调用。它的最简形式类似于一个不带任何参数的实例方法，以关键字 init 命名。

下面例子中定义了一个用来保存华氏温度的结构体 Fahrenheit ，它拥有一个 Double 类型的存储型属性 temperature ：

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
```

```
var f = Fahrenheit()
println("The default temperature is \(f.temperature)°  Fahrenheit")
// 输出 "The default temperature is 32.0°  Fahrenheit”
```

这个结构体定义了一个不带参数的构造器 init ，并在里面将存储型属性 temperature 的值初始化为 32.0 （华摄氏度下水的冰点）。

()

默认属性值

如前所述，你可以在构造器中为存储型属性设置初始值；同样，你也可以在属性声明时为其设置默认值。

> 注意：
> 如果一个属性总是使用同一个初始值，可以为其设置一个默认值。无论定义默认值还是在构造器中赋值，最终它们实现的效果是一样的，只不过默认值将属性的初始化和属性的声明结合的更紧密。使用默认值能让你的构造器更简洁、更清晰，且能通过默认值自动推导出属性的类型；同时，它也能让你充分利用默认构造器、构造器继承（后续章节将讲到）等特性。

你可以使用更简单的方式在定义结构体 Fahrenheit 时为属性 temperature 设置默认值：

```
struct Fahrenheit {
    var temperature = 32.0
}
```

定制化构造过程

你可以通过输入参数和可选属性类型来定制构造过程，也可以在构造过程中修改常量属性。这些都将在后面章节中提到。

构造参数

你可以在定义构造器时提供构造参数，为其提供定制化构造所需值的类型和名字。构造器参数的功能和语法跟函数和方法参数相同。

下面例子中定义了一个包含摄氏度温度的结构体 Celsius 。它定义了两个不同的构造器： init(fromFahrenheit:) 和 init(fromKelvin:) ，二者分别通过接受不同刻度表示的温度值来创建新的实例：

```
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”
```

第一个构造器拥有一个构造参数，其外部名字为 fromFahrenheit ，内部名字为 fahrenheit ；第二个构造器也拥有一个构造参数，其外部名字为 fromKelvin ，内部名字为 kelvin 。这两个构造器都将唯一的参数值转换成摄氏温度值，并保存在属性 temperatureInCelsius 中。

内部和外部参数名

跟函数和方法参数相同，构造参数也存在一个在构造器内部使用的参数名字和一个在调用构造器时使用的外部参数名字。

然而，构造器并不像函数和方法那样在括号前有一个可辨别的名字。所以在调用构造器时，主要通过构造器中的参数名和类型来确定需要调用的构造器。正因为参数如此重要，如果你在定义构造器时没有提供参数的外部名字，Swift 会为每个构造器的参数自动生成一个跟内部名字相同的外部名，就相当于在每个构造参数之前加了一个哈希符号。

> 注意：
> 如果你不希望为构造器的某个参数提供外部名字，你可以使用下划线 _ 来显示描述它的外部名，以此覆盖上面所说的默认行为。

以下例子中定义了一个结构体 Color ，它包含了三个常量： red 、 green 和 blue 。这些属性可以存储0.0到1.0之间的值，用来指示颜色中红、绿、蓝成分的含量。

Color 提供了一个构造器，其中包含三个 Double 类型的构造参数：

```
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
    }
}
```

每当你创建一个新的 Color 实例，你都需要通过三种颜色的外部参数名来传值，并调用构造器。

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
```

注意，如果不通过外部参数名字传值，你是没法调用这个构造器的。只要构造器定义了某个外部参数名，你就必须使用它，忽略它将导致编译错误：

```
let veryGreen = Color(0.0, 1.0, 0.0)
// 报编译时错误，需要外部名称
```

可选属性类型

如果你定制的类型包含一个逻辑上允许取值为空的存储型属性--不管是因为它无法在初始化时赋值，还是因为它可以在之后某个时间点可以赋值为空--你都需要将它定义为可选类型 optional type 。可选类型的属性将自动初始化为空 nil ，表示这个属性是故意在初始化时设置为空的。

下面例子中定义了类 SurveyQuestion ，它包含一个可选字符串属性 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()
// 输出 "Do you like cheese?"
cheeseQuestion.response = "Yes, I do like cheese."
```

调查问题在问题提出之后，我们才能得到回答。所以我们将属性回答 response 声明为 String? 类型，或者说是可选字符串类型 optional String 。当 SurveyQuestion 实例化时，它将自动赋值为空 nil ，表明暂时还不存在此字符串。

()

构造过程中常量属性的修改

只要在构造过程结束前常量的值能确定，你可以在构造过程中的任意时间点修改常量属性的值。

> 注意：
> 对某个类实例来说，它的常量属性只能在定义它的类的构造过程中修改；不能在子类中修改。

你可以修改上面的 SurveyQuestion 示例，用常量属性替代变量属性 text ，指明问题内容 text 在其创建之后不会再被修改。尽管 text 属性现在是常量，我们仍然可以在其类的构造器中设置它的值：

```
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()
// 输出 "How about beets?"
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

默认构造器

Swift 将为所有属性已提供默认值的且自身没有定义任何构造器的结构体或基类，提供一个默认的构造器。这个默认构造器将简单的创建一个所有属性值都设置为默认值的实例。

下面例子中创建了一个类 ShoppingListItem ，它封装了购物清单中的某一项的属性：名字（ name ）、数量（ quantity ）和购买状态 purchase state 。

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()
```

由于 ShoppingListItem 类中的所有属性都有默认值，且它是没有父类的基类，它将自动获得一个可以为所有属性设置默认值的默认构造器（尽管代码中没有显式为 name 属性设置默认值，但由于 name 是可选字符串类型，它将默认设置为 nil ）。上面例子中使用默认构造器创造了一个 ShoppingListItem 类的实例（使用 ShoppingListItem() 形式的构造器语法），并将其赋值给变量 item 。

结构体的逐一成员构造器

除上面提到的默认构造器，如果结构体对所有存储型属性提供了默认值且自身没有提供定制的构造器，它们能自动获得一个逐一成员构造器。

逐一成员构造器是用来初始化结构体新实例里成员属性的快捷方法。我们在调用逐一成员构造器时，通过与成员属性名相同的参数名进行传值来完成对成员属性的初始赋值。

下面例子中定义了一个结构体 Size ，它包含两个属性 width 和 height 。Swift 可以根据这两个属性的初始赋值 0.0 自动推导出它们的类型 Double 。

由于这两个存储型属性都有默认值，结构体 Size 自动获得了一个逐一成员构造器 init(width:height:) 。你可以用它来为 Size 创建新的实例：

```
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

值类型的构造器代理

构造器可以通过调用其它构造器来完成实例的部分构造过程。这一过程称为构造器代理，它能减少多个构造器间的代码重复。

构造器代理的实现规则和形式在值类型和类类型中有所不同。值类型（结构体和枚举类型）不支持继承，所以构造器代理的过程相对简单，因为它们只能代理给本身提供的其它构造器。类则不同，它可以继承自其它类（请参考继承 ()），这意味着类有责任保证其所有继承的存储型属性在构造时也能正确的初始化。这些责任将在后续章节类的继承和构造过程 (页 137)中介绍。

对于值类型，你可以使用 self.init 在自定义的构造器中引用其它的属于相同值类型的构造器。并且你只能在构造器内部调用 self.init 。

注意，如果你为某个值类型定义了一个定制的构造器，你将无法访问到默认构造器（如果是结构体，则无法访问逐一对象构造器）。这个限制可以防止你在为值类型定义了一个更复杂的，完成了重要准备构造器之后，别人还是错误的使用了那个自动生成的构造器。

> 注意：
> 假如你想通过默认构造器、逐一对象构造器以及你自己定制的构造器为值类型创建实例，我们建议你将自己定制的构造器写到扩展（ extension ）中，而不是跟值类型定义混在一起。想查看更多内容，请查看扩展 () 章节。

下面例子将定义一个结构体 Rect ，用来代表几何矩形。这个例子需要两个辅助的结构体 Size 和 Point ，它们各自为其所有的属性提供了初始值 0.0 。

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
```

你可以通过以下三种方式为 Rect 创建实例--使用默认的0值来初始化 origin 和 size 属性；使用特定的 origin 和 size 实例来初始化；使用特定的 center 和 size 来初始化。在下面 Rect 结构体定义中，我们为这三种方式提供了三个自定义的构造器：

```
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)
    }
}
```

第一个 Rect 构造器 init()，在功能上跟没有自定义构造器时自动获得的默认构造器是一样的。这个构造器是一个空函数，使用一对大括号 {} 来描述，它没有执行任何定制的构造过程。调用这个构造器将返回一个 Rect 实例，它的 origin 和 size 属性都使用定义时的默认值 Point(x: 0.0, y: 0.0) 和 Size(width: 0.0, height: 0.0)：

```
let basicRect = Rect()
// basicRect 的原点是 (0.0, 0.0)，尺寸是 (0.0, 0.0)
```

第二个 Rect 构造器 init(origin:size:)，在功能上跟结构体在没有自定义构造器时获得的逐一成员构造器是一样的。这个构造器只是简单地将 origin 和 size 的参数值赋给对应的存储型属性：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
// originRect 的原点是 (2.0, 2.0)，尺寸是 (5.0, 5.0)
```

第三个 Rect 构造器 init(center:size:) 稍微复杂一点。它先通过 center 和 size 的值计算出 origin 的坐标。然后再调用（或代理给）init(origin:size:) 构造器来将新的 origin 和 size 值赋值到对应的属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect 的原点是 (2.5, 2.5)，尺寸是 (3.0, 3.0)
```

构造器 init(center:size:) 可以自己将 origin 和 size 的新值赋值到对应的属性中。然而尽量利用现有的构造器和它所提供的功能来实现 init(center:size:) 的功能，是更方便、更清晰和更直观的方法。

> 注意：
> 如果你想用另外一种不需要自己定义 init() 和 init(origin:size:) 的方式来实现这个例子，请参考扩展 () 。

()

类的继承和构造过程

类里面的所有存储型属性--包括所有继承自父类的属性--都必须在构造过程中设置初始值。

Swift 提供了两种类型的类构造器来确保所有类实例中存储型属性都能获得初始值，它们分别是指定构造器和便利构造器。

指定构造器和便利构造器

指定构造器是类中最主要的构造器。一个指定构造器将初始化类中提供的所有属性，并根据父类链往上调用父类的构造器来实现父类的初始化。

每一个类都必须拥有至少一个指定构造器。在某些情况下，许多类通过继承了父类中的指定构造器而满足了这个条件。具体内容请参考后续章节自动构造器的继承 (页 143)。

便利构造器是类中比较次要的、辅助型的构造器。你可以定义便利构造器来调用同一个类中的指定构造器，并为其参数提供默认值。你也可以定义便利构造器来创建一个特殊用途或特定输入的实例。

你应当只在必要的时候为类提供便利构造器，比方说某种情况下通过使用便利构造器来快捷调用某个指定构造器，能够节省更多开发时间并让类的构造过程更清晰明了。

()

构造器链

为了简化指定构造器和便利构造器之间的调用关系，Swift 采用以下三条规则来限制构造器之间的代理调用：

规则 1
指定构造器必须调用其直接父类的的指定构造器。

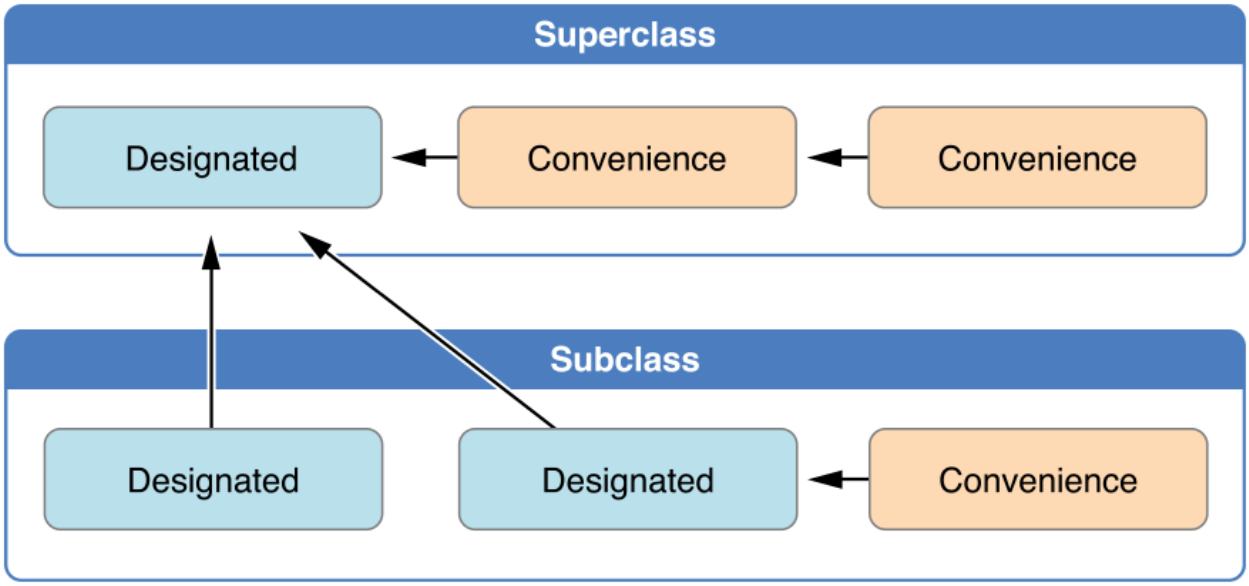
规则 2
便利构造器必须调用同一类中定义的其它构造器。

规则 3
便利构造器必须最终以调用一个指定构造器结束。

一个更方便记忆的方法是：

- 指定构造器必须总是向上代理
- 便利构造器必须总是横向代理

这些规则可以通过下面图例来说明：



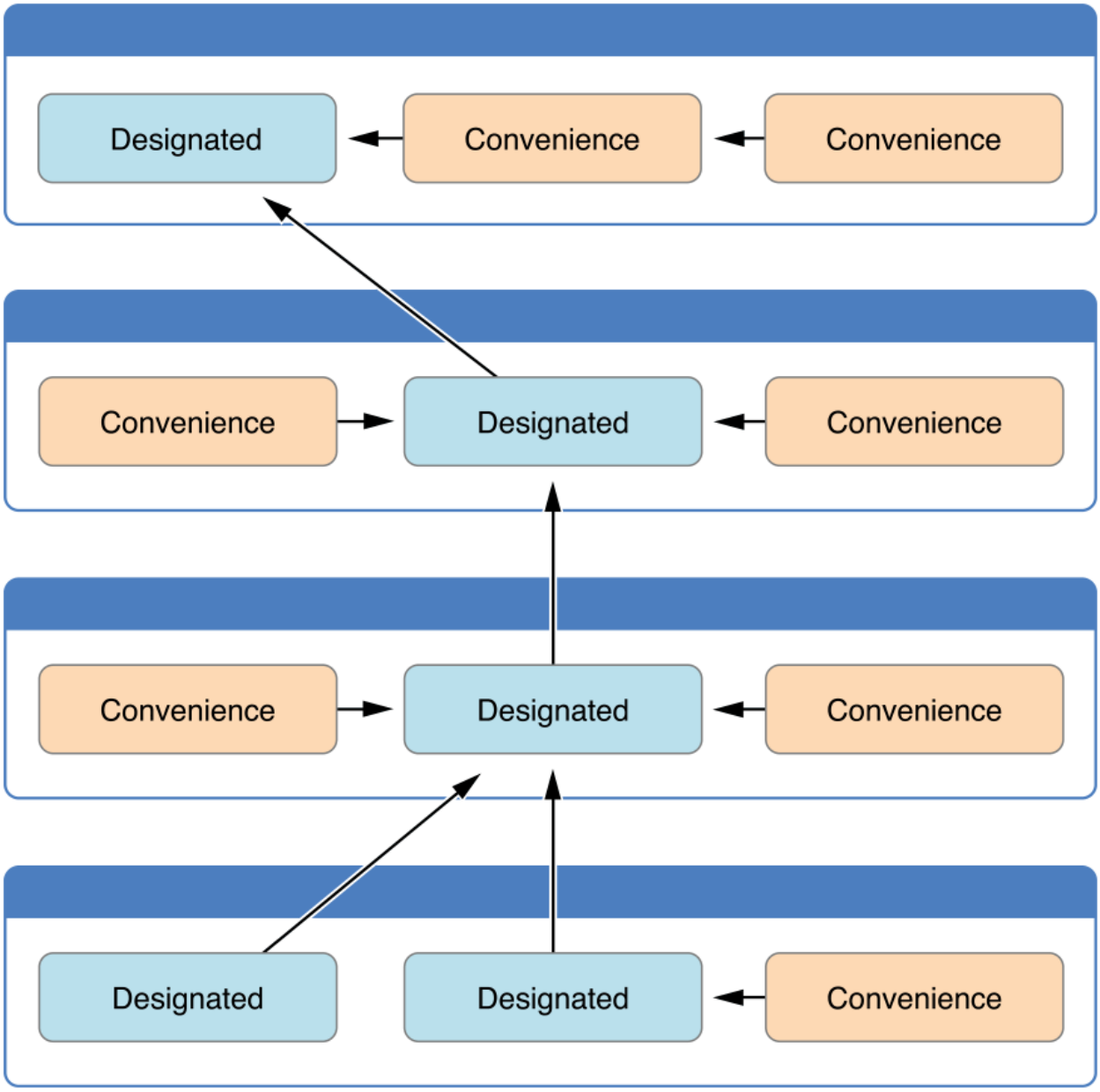
图片 14.1 Image of Initialization_1.png

如图所示，父类中包含一个指定构造器和两个便利构造器。其中一个便利构造器调用了另外一个便利构造器，而后者又调用了唯一的指定构造器。这满足了上面提到的规则2和3。这个父类没有自己的父类，所以规则1没有用到。

子类中包含两个指定构造器和一个便利构造器。便利构造器必须调用两个指定构造器中的任意一个，因为它只能调用同一个类里的其他构造器。这满足了上面提到的规则2和3。而两个指定构造器必须调用父类中唯一的指定构造器，这满足了规则1。

> 注意：
> 这些规则不会影响使用时，如何用类去创建实例。任何上图中展示的构造器都可以用来完整创建对应类的实例。这些规则只在实现类的定义时有影响。

下面图例中展示了一种针对四个类的更复杂的类层级结构。它演示了指定构造器是如何在类层级中充当“管道”的作用，在类的构造器链上简化了类之间的相互关系。



图片 14.2 Image of Initialization_2.png

()

两段式构造过程

Swift 中类的构造过程包含两个阶段。第一个阶段，每个存储型属性通过引入它们的类的构造器来设置初始值。当每一个存储型属性值被确定后，第二阶段开始，它给每个类一次机会在新实例准备使用之前进一步定制它们的存储型属性。

两段式构造过程的使用让构造过程更安全，同时在整个类层级结构中给予了每个类完全的灵活性。两段式构造过程可以防止属性值在初始化之前被访问；也可以防止属性被另外一个构造器意外地赋予不同的值。

> 注意：
> Swift的两段式构造过程跟 Objective-C 中的构造过程类似。最主要的区别在于阶段 1，Objective-C 给每一个属性赋值 0 或空值（比如说 0 或 nil ）。Swift 的构造流程则更加灵活，它允许你设置定制的初始值，并自如应对某些属性不能以 0 或 nil 作为合法默认值的情况。

Swift 编译器将执行 4 种有效的安全检查，以确保两段式构造过程能顺利完成：

安全检查 1
指定构造器必须保证它所在类引入的所有属性都必须先初始化完成，之后才能将其它构造任务向上代理给父类中的构造器。

如上所述，一个对象的内存只有在其所有存储型属性确定之后才能完全初始化。为了满足这一规则，指定构造器必须保证它所在类引入的属性在它往上代理之前先完成初始化。

安全检查 2
指定构造器必须先向上代理调用父类构造器，然后再为继承的属性设置新值。如果没这么做，指定构造器赋予的新值将被父类中的构造器所覆盖。

安全检查 3
便利构造器必须先代理调用同一类中的其它构造器，然后再为任意属性赋新值。如果没这么做，便利构造器赋予的新值将被同一类中其它指定构造器所覆盖。

安全检查 4
构造器在第一阶段构造完成之前，不能调用任何实例方法、不能读取任何实例属性的值， self 的值不能被引用。

类实例在第一阶段结束以前并不是完全有效，仅能访问属性和调用方法，一旦完成第一阶段，该实例才会声明为有效实例。

以下是两段式构造过程中基于上述安全检查的构造流程展示：

阶段 1

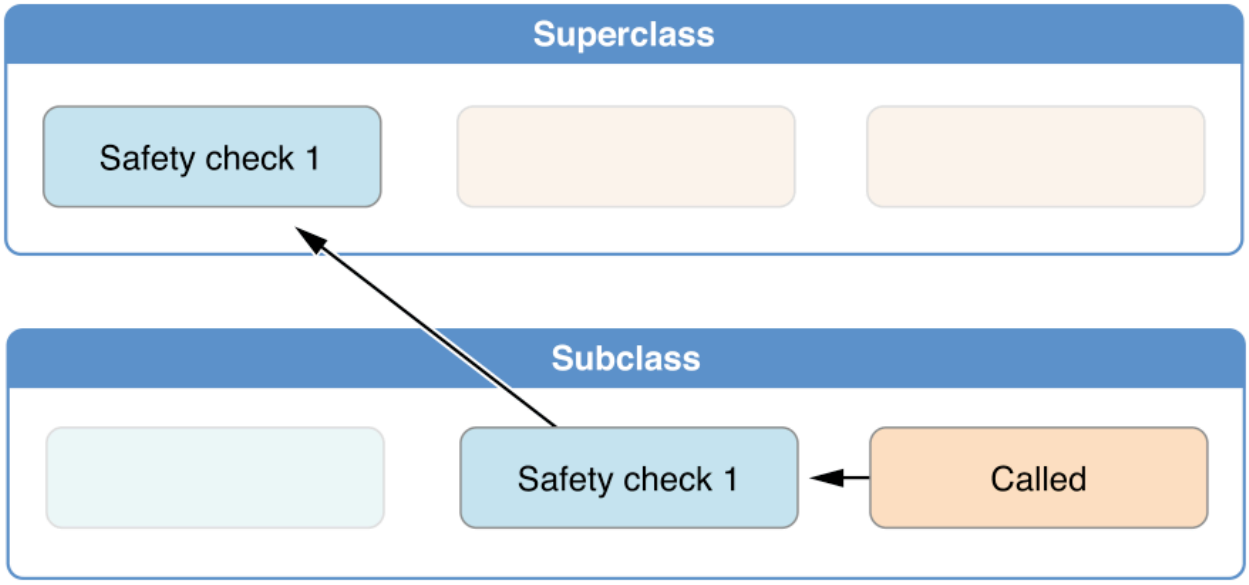
- 某个指定构造器或便利构造器被调用；
- 完成新实例内存的分配，但此时内存还没有被初始化；
- 指定构造器确保其所在类引入的所有存储型属性都已赋初值。存储型属性所属的内存完成初始化；
- 指定构造器将调用父类的构造器，完成父类属性的初始化；
- 这个调用父类构造器的过程沿着构造器链一直往上执行，直到到达构造器链的最顶部；

- 当到达了构造器链最顶部，且已确保所有实例包含的存储型属性都已经赋值，这个实例的内存被认为已经完全初始化。此时阶段1完成。

阶段 2

- 从顶部构造器链一直往下，每个构造器链中类的指定构造器都有机会进一步定制实例。构造器此时可以访问 `self` 、修改它的属性并调用实例方法等等。
- 最终，任意构造器链中的便利构造器可以有机会定制实例和使用 `self` 。

下图展示了在假定的子类和父类之间构造的阶段1：



图片 14.3 Image of Initialization_3.png

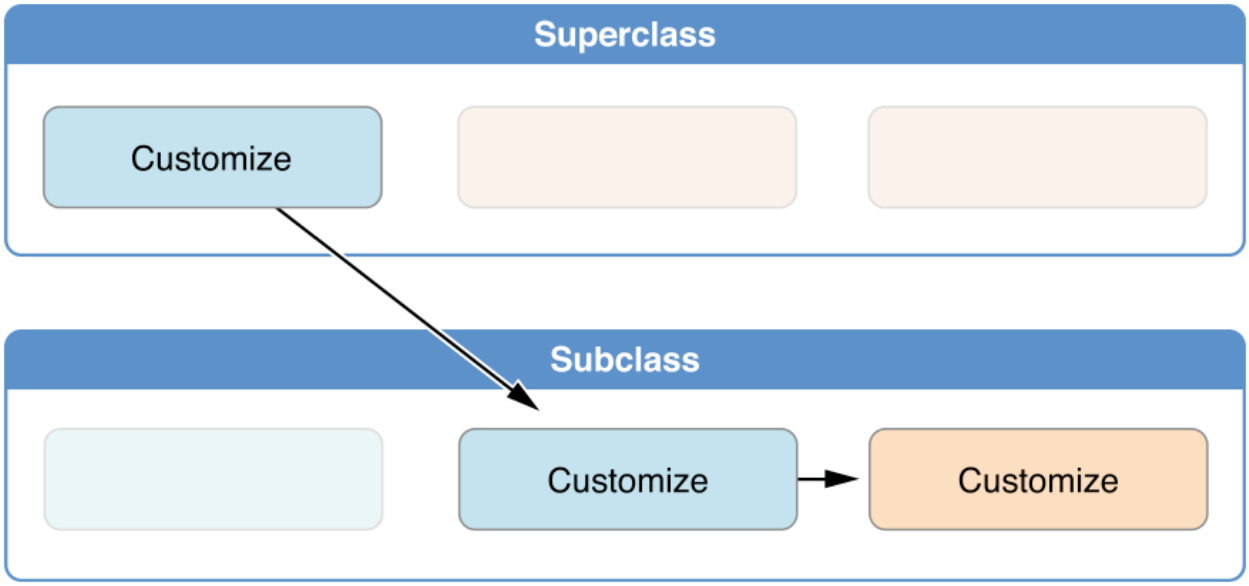
在这个例子中，构造过程从对子类中一个便利构造器的调用开始。这个便利构造器此时没法修改任何属性，它把构造任务代理给同一类中的指定构造器。

如安全检查1所示，指定构造器将确保所有子类的属性都有值。然后它将调用父类的指定构造器，并沿着造器链一直往上完成父类的构建过程。

父类中的指定构造器确保所有父类的属性都有值。由于没有更多的父类需要构建，也就无需继续向上做构建代理。

一旦父类中所有属性都有了初始值，实例的内存被认为是完全初始化，而阶段1也已完成。

以下展示了相同构造过程的阶段2：



图片 14.4 Image of Initialization_4.png

父类中的指定构造器现在有机会进一步来定制实例（尽管它没有这种必要）。

一旦父类中的指定构造器完成调用，子类的构指定构造器可以执行更多的定制操作（同样，它也没有这种必要）。

最终，一旦子类的指定构造器完成调用，最开始被调用的便利构造器可以执行更多的定制操作。

构造器的继承和重载

跟 Objective-C 中的子类不同，Swift 中的子类不会默认继承父类的构造器。Swift 的这种机制可以防止一个父类的简单构造器被一个更专业的子类继承，并被错误的用来创建子类的实例。

假如你希望自定义的子类中能实现一个或多个跟父类相同的构造器--也许是为了完成一些定制的构造过程--你可以在你定制的子类中提供和重载与父类相同的构造器。

如果你重载的构造器是一个指定构造器，你可以在子类里重载它的实现，并在自定义版本的构造器中调用父类版本的构造器。

如果你重载的构造器是一个便利构造器，你的重载过程必须通过调用同一类中提供的其它指定构造器来实现。这一规则的详细内容请参考构造器链 (页 138)。

> 注意：
> 与方法、属性和下标不同，在重载构造器时你没有必要使用关键字 `override` 。

()

自动构造器的继承

如上所述，子类不会默认继承父类的构造器。但是如果特定条件可以满足，父类构造器是可以被自动继承的。在实践中，这意味着对于许多常见场景你不必重载父类的构造器，并且在尽可能安全的情况下以最小的代价来继承父类的构造器。

假设要为子类中引入的任意新属性提供默认值，请遵守以下2个规则：

规则 1
如果子类没有定义任何指定构造器，它将自动继承所有父类的指定构造器。

规则 2
如果子类提供了所有父类指定构造器的实现--不管是通过规则1继承过来的，还是通过自定义实现的--它将自动继承所有父类的便利构造器。

即使你在子类中添加了更多的便利构造器，这两条规则仍然适用。

> 注意：
> 子类可以通过部分满足规则2的方式，使用子类便利构造器来实现父类的指定构造器。

指定构造器和便利构造器的语法

类的指定构造器的写法跟值类型简单构造器一样：

```
init(parameters) {
    statements
}
```

便利构造器也采用相同样式的写法，但需要在 `init` 关键字之前放置 `convenience` 关键字，并使用空格将它们俩分开：

```
convenience init(parameters) {
    statements
}
```

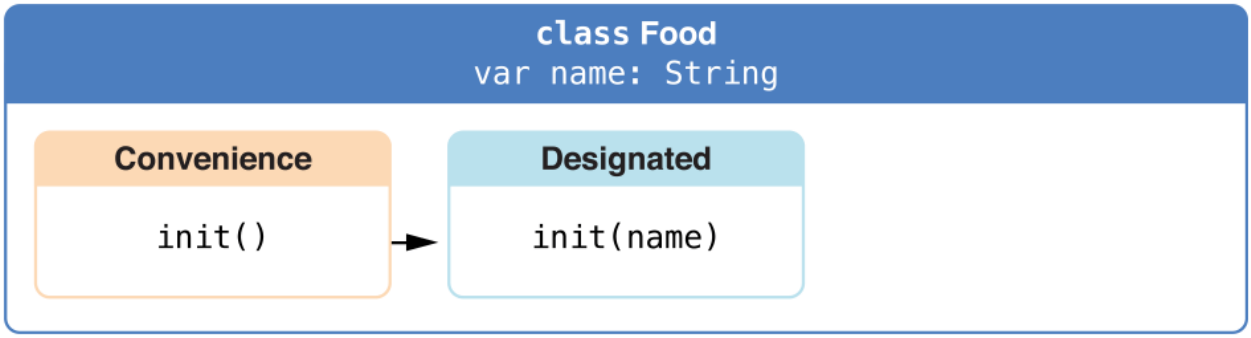
指定构造器和便利构造器实战

接下来的例子将在实战中展示指定构造器、便利构造器和自动构造器的继承。它定义了包含三个类 `Food` 、 `RecipeIngredient` 以及 `ShoppingListItem` 的类层次结构，并将演示它们的构造器是如何相互作用的。

类层次中的基类是 `Food` ，它是一个简单的用来封装食物名字的类。 `Food` 类引入了一个叫做 `name` 的 `String` 类型属性，并且提供了两个构造器来创建 `Food` 实例：

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

下图中展示了 Food 的构造器链：



图片 14.5 Image of Initialization_5.png

类没有提供一个默认的逐一成员构造器，所以 Food 类提供了一个接受单一参数 name 的指定构造器。这个构造器可以使用一个特定的名字来创建新的 Food 实例：

```
let namedMeat = Food(name: "Bacon")
// namedMeat 的名字是 "Bacon”
```

Food 类中的构造器 init(name: String) 被定义为一个指定构造器，因为它能确保所有新 Food 实例的中存储型属性都被初始化。Food 类没有父类，所以 init(name: String) 构造器不需要调用 super.init() 来完成构造。

Food 类同样提供了一个没有参数的便利构造器 init()。这个 init() 构造器为新食物提供了一个默认的占位名字，通过代理调用同一类中定义的指定构造器 init(name: String) 并给参数 name 传值 [Unnamed] 来实现：

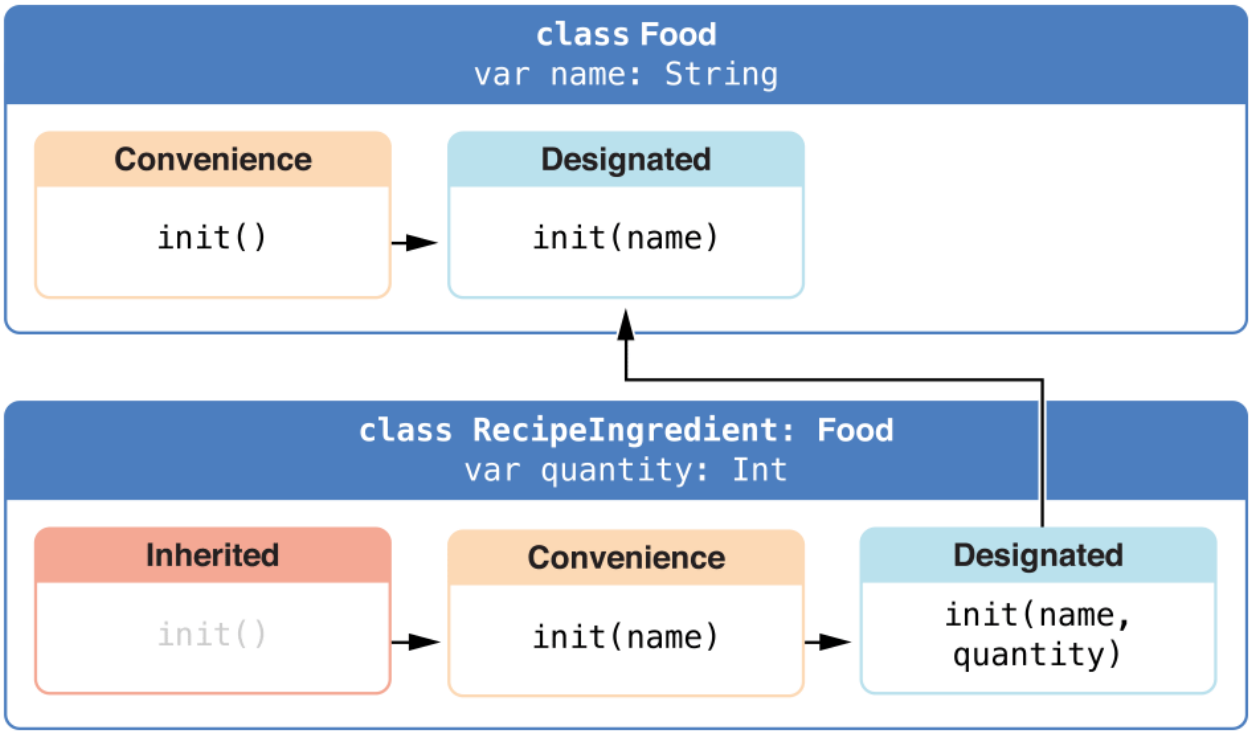
```
let mysteryMeat = Food()
// mysteryMeat 的名字是 [Unnamed]
```

类层级中的第二个类是 Food 的子类 RecipeIngredient。RecipeIngredient 类构建了食谱中的一味调味剂。它引入了 Int 类型的数量属性 quantity（以及从 Food 继承过来的 name 属性），并且定义了两个构造器来创建 RecipeIngredient 实例：

```
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
```

```
    }
}
```

下图中展示了 RecipeIngredient 类的构造器链：



图片 14.6 Image of Initialization_6.png

RecipeIngredient 类拥有一个指定构造器 init(name: String, quantity: Int)，它可以用来产生新 RecipeIngredient 实例的所有属性值。这个构造器一开始先将传入的 quantity 参数赋值给 quantity 属性，这个属性也是唯一在 RecipeIngredient 中新引入的属性。随后，构造器将任务向上代理给父类 Food 的 init(name: String)。这个过程满足两段式构造过程 (页 140)中的安全检查1。

RecipeIngredient 也定义了一个便利构造器 init(name: String)，它只通过 name 来创建 RecipeIngredient 的实例。这个便利构造器假设任意 RecipeIngredient 实例的 quantity 为1，所以不需要显示指明数量即可创建出实例。这个便利构造器的定义可以让创建实例更加方便和快捷，并且避免了使用重复的代码来创建多个 quantity 为 1 的 RecipeIngredient 实例。这个便利构造器只是简单的将任务代理给了同一类里提供的指定构造器。

注意，RecipeIngredient 的便利构造器 init(name: String) 使用了跟 Food 中指定构造器 init(name: String) 相同的参数。因为这个便利构造器重写要父类的指定构造器 init(name: String)，必须在前面使用使用 override 标识。

在这个例子中，RecipeIngredient 的父类是 Food，它有一个便利构造器 init()。这个构造器因此也被 RecipeIngredient 继承。这个继承的 init() 函数版本跟 Food 提供的版本是一样的，除了它是将任务代理给 RecipeIngredient 版本的 init(name: String) 而不是 Food 提供的版本。

所有的这三种构造器都可以用来创建新的 RecipeIngredient 实例：

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

类层级中第三个也是最后一个类是 RecipeIngredient 的子类，叫做 ShoppingListItem 。这个类构建了购物单中出现的某一种调味料。

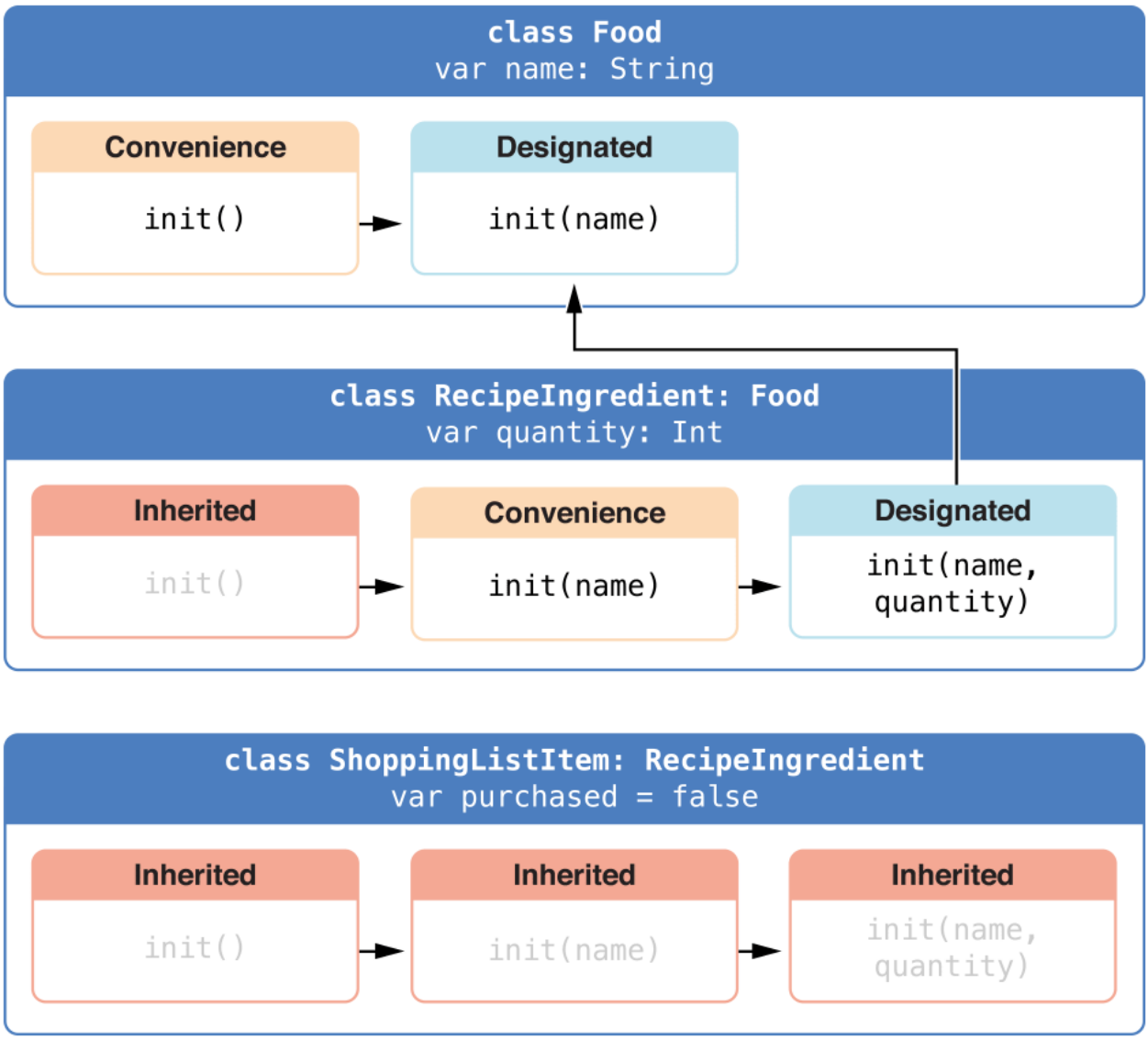
购物单中的每一项总是从 unpurchased 未购买状态开始的。为了展现这一事实， ShoppingListItem 引入了一个布尔类型的属性 purchased ，它的默认值是 false 。 ShoppingListItem 还添加了一个计算型属性 description ，它提供了关于 ShoppingListItem 实例的一些文字描述：

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
    var output = "\(quantity) x \(name.lowercaseString)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

> 注意：
>
> ShoppingListItem 没有定义构造器来为 purchased 提供初始化值，这是因为任何添加到购物单的项的初始状态总是未购买。

由于它为自己引入的所有属性都提供了默认值，并且自己没有定义任何构造器， ShoppingListItem 将自动继承所有父类中的指定构造器和便利构造器。

下图种展示了所有三个类的构造器链：



图片 14.7 Image of Initialization_7.png

你可以使用全部三个继承来的构造器来创建 ShoppingListItem 的新实例：

```
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，例子中通过字面量方式创建了一个新数组 breakfastList ，它包含了三个新的 ShoppingListItem 实例，因此数组的类型也能自动推导为 ShoppingListItem[] 。在数组创建完之后，数组中第一个 ShoppingListIte

m 实例的名字从 [Unnamed] 修改为 Orange juice ，并标记为已购买。接下来通过遍历数组每个元素并打印它们的描述值，展示了所有项当前的默认状态都已按照预期完成了赋值。

可失败构造器

如果一个类，结构体或枚举类型的对象，在构造自身的过程中有可能失败，则为其定义一个可失败构造器，是非常有必要的。这里所指的“失败”是指，如给构造器传入无效的参数值，或缺少某种所需的外部资源，又或是不满足某种必要的条件等。

为了妥善处理这种构造过程中可能会失败的情况。你可以在一个类，结构体或是枚举类型的定义中，添加一个或多个可失败构造器。其语法为在 init 关键字后面加添问号（init?）。

> 注意：
>
> 可失败构造器的参数名和参数类型，不能与其它非可失败构造器的参数名，及其类型相同。

可失败构造器，在构建对象的过程中，创建一个其自身类型为可选类型的对象。你通过 return nil 语句，来表明可失败构造器在何种情况下“失败”。

> 注意：
>
> 严格来说，构造器都不支持返回值。因为构造器本身的作用，只是为了能确保对象自身能被正确构建。所以即使你在表明可失败构造器，失败的这种情况下，用到了 return nil 。也不要在表明可失败构造器成功的这种情况下，使用关键字 return 。

下例中，定义了一个名为 Animal 的结构体，其中有一个名为 species 的， String 类型的常量属性。同时该结构体还定义了一个，带一个 String 类型参数 species 的,可失败构造器。这个可失败构造器，被用来检查传入的参数是否为一个空字符串，如果为空字符串，则该可失败构造器，构建对象失败，否则成功。

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}
```

你可以通过该可失败构造器来构建一个Animal的对象，并检查其构建过程是否成功。

```
let someCreature = Animal(species: "Giraffe")
// someCreature 的类型是 Animal? 而不是 Animal
if let giraffe = someCreature {
    println("An animal was initialized with a species of \(giraffe.species)")
}
// 打印 "An animal was initialized with a species of Giraffe"
```

如果你给该可失败构造器传入一个空字符串作为其参数，则该可失败构造器失败。

```
let anonymousCreature = Animal(species: "")
// anonymousCreature 的类型是 Animal?, 而不是 Animal
if anonymousCreature == nil {
    println("The anonymous creature could not be initialized")
}
// 打印 "The anonymous creature could not be initialized"
```

> 注意：
>
> 空字符串（ "" ）和一个值为 nil 的可选类型的字符串是两个完全不同的概念。上例中的空字符串（ "" ）其实是一个有效的，非可选类型的字符串。这里我们只所以让 Animal 的可失败构造器，构建对象失败，只是因为对于 Animal 这个类的 species 属性来说，它更适合有一个具体的值，而不是空字符串。

枚举类型的可失败构造器

你可以通过构造一个带一个或多个参数的可失败构造器来获取枚举类型中特定的枚举成员。还能在参数不满足你所期望的条件时，导致构造失败。

下例中，定义了一个名为TemperatureUnit的枚举类型。其中包含了三个可能的枚举成员(Kelvin ， Celsius ，和 Fahrenheit)和一个被用来找到 Character 值所对应的枚举成员的可失败构造器：

```
enum TemperatureUnit {
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .Kelvin
        case "C":
            self = .Celsius
        case "F":
            self = .Fahrenheit
        default:
            return nil
        }
    }
}
```

你可以通过给该可失败构造器传递合适的参数来获取这三个枚举成员中相匹配的其中一个枚举成员。当参数的值不能与任意一枚举成员相匹配时，该枚举类型的构建过程失败：

```
let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    println("This is a defined temperature unit, so initialization succeeded.")
}
// 打印 "This is a defined temperature unit, so initialization succeeded."
let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    println("This is not a defined temperature unit, so initialization failed.")
}
// 打印 "This is not a defined temperature unit, so initialization failed."
```

带原始值的枚举类型的可失败构造器

带原始值的枚举类型会自带一个可失败构造器 init?(rawValue:) ,该可失败构造器有一个名为 rawValue 的默认参数,其类型和枚举类型的原始值类型一致，如果该参数的值能够和枚举类型成员所带的原始值匹配，则该构造器构造一个带此原始值的枚举成员，否则构造失败。

因此上面的 TemperatureUnit的例子可以重写为：

```
enum TemperatureUnit: Character {
    case Kelvin = "K", Celsius = "C", Fahrenheit = "F"
}
let fahrenheitUnit = TemperatureUnit(rawValue: "F")
if fahrenheitUnit != nil {
    println("This is a defined temperature unit, so initialization succeeded.")
}
// prints "This is a defined temperature unit, so initialization succeeded."
let unknownUnit = TemperatureUnit(rawValue: "X")
if unknownUnit == nil {
    println("This is not a defined temperature unit, so initialization failed.")
}
// prints "This is not a defined temperature unit, so initialization failed."
```

类的可失败构造器

值类型（如结构体或枚举类型）的可失败构造器，对何时何地触发构造失败这个行为没有任何的限制。比如在前面的例子中，结构体 Animal 的可失败构造器触发失败的行为，甚至发生在 species 属性的值被初始化以前。而对类而言，就没有那么幸运了。类的可失败构造器只能在所有的类属性被初始化后和所有类之间的构造器之间的代理调用发生完后触发失败行为。

下例子中，定义了一个名为 Product 的类，其内部结构和结构体 Animal 很相似，内部也有一个名为 name 的 String 类型的属性。由于该属性的值同样不能为空字符串，所以我们加入了可失败构造器来确保该类满足上述条件。但由于 Product 类不是一个结构体，所以当想要在该类中添加可失败构造器触发失败条件时，必须确保 name 属性被初始化。因此我们把 name 属性的 String 类型做了一点点小小的修改，把其改为隐式解析可选类型（ String! ），来确保可失败构造器触发失败条件时，所有类属性都被初始化了。因为所有可选类型都有一个默认的初始值 nil 。因此最后 Product 类可写为：

```
class Product {
    let name: String!
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}
```

因为 name 属性是一个常量，所以一旦 Product 类构造成功， name 属性肯定有一个非 nil 的值。因此完全可以放心大胆的直接访问 Product 类的 name 属性，而不用考虑去检查 name 属性是否有值。

```
if let bowTie = Product(name: "bow tie") {
    // 不需要检查 bowTie.name == nil
    println("The product's name is \(bowTie.name)")
}
// 打印 "The product's name is bow tie"
```

构造失败的传递

可失败构造器同样满足在构造器链 (页 138)中所描述的构造规则。其允许在同一类，结构体和枚举中横向代理其他的可失败构造器。类似的，子类的可失败构造器也能向上代理基类的可失败构造器。

无论是向上代理还是横向代理，如果你代理的可失败构造器，在构造过程中触发了构造失败的行为，整个构造过程都将被立即终止，接下来任何的构造代码都将不会被执行。

> 注意：
>
> 可失败构造器也可以代理调用其它的非可失败构造器。通过这个方法，你可以为已有的构造过程加入构造失败的条件。

下面这个例子，定义了一个名为 `CartItem` 的 `Product` 类的子类。这个类建立了一个在线购物车中的物品的模型，它有一个名为 `quantity` 的常量参数，用来表示该物品的数量至少为1：

```
class CartItem: Product {
    let quantity: Int!
    init?(name: String, quantity: Int) {
        super.init(name: name)
        if quantity < 1 { return nil }
        self.quantity = quantity
    }
}
```

和 `Product` 类中的 `name` 属性相类似的， `CartItem` 类中的 `quantity` 属性的类型也是一个隐式解析可选类型，只不过由（ `String!` ）变为了（ `Int!` ）。这样做都是为了确保在构造过程中，该属性在被赋予特定的值之前能有一个默认的初始值nil。

可失败构造器总是先向上代理调用基类, `Product` 的构造器 `init(name:)` 。这满足了可失败构造器在触发构造失败这个行为前必须总是执行构造代理调用这个条件。

如果由于 `name` 的值为空而导致基类的构造器在构造过程中失败。则整个 `CartItem` 类的构造过程都将失败，后面的子类的构造过程都将不会被执行。如果基类构建成功，则继续运行子类的构造器代码。

如果你构造了一个 `CartItem` 对象，并且该对象的 `name` 属性不为空以及 `quantity` 属性为1或者更多，则构造成功：

```
if let twoSocks = CartItem(name: "sock", quantity: 2) {
    println("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")
}
// 打印 "Item: sock, quantity: 2"
```

如果你构造一个 CartItem 对象，其 quantity 的值 0，则 CartItem 的可失败构造器触发构造失败的行为:

```
if let zeroShirts = CartItem(name: "shirt", quantity: 0) {
    println("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")
} else {
    println("Unable to initialize zero shirts")
}
// 打印 "Unable to initialize zero shirts"
```

类似的, 如果你构造一个 CartItem 对象，但其 name 的值为空, 则基类 Product 的可失败构造器将触发构造失败的行为，整个 CartItem 的构造行为同样为失败：

```
if let oneUnnamed = CartItem(name: "", quantity: 1) {
    println("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")
} else {
    println("Unable to initialize one unnamed product")
}
// 打印 "Unable to initialize one unnamed product"
```

覆盖一个可失败构造器

就如同其它构造器一样，你也可以用子类的可失败构造器覆盖基类的可失败构造器。或者你也可以用子类的非可失败构造器覆盖一个基类的可失败构造器。这样做的好处是，即使基类的构造器为可失败构造器，但当子类的构造器在构造过程不可能失败时，我们也可以把它修改过来。

注意当你用一个子类的非可失败构造器覆盖了一个父类的可失败构造器时，子类的构造器将不再能向上代理父类的可失败构造器。一个非可失败的构造器永远也不能代理调用一个可失败构造器。

> 注意：
>
> 你可以用一个非可失败构造器覆盖一个可失败构造器，但反过来却行不通。

下例定义了一个名为 Document 的类，这个类中的 name 属性允许为 nil 和一个非空字符串，但不能是一个空字符串：

```
class Document {
    var name: String?
    // 该构造器构建了一个name属性值为nil的document对象
    init() {}
    // 该构造器构建了一个name属性值为非空字符串的document对象
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}
```

下面这个例子，定义了一个名为 AutomaticallyNamedDocument 的 Document 类的子类。这个子类覆盖了基类的两个指定构造器。确保了不论在何种情况下 name 属性总是有一个非空字符串 [Untitled] 的值。

```
class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
    override init(name: String) {
        super.init()
        if name.isEmpty {
            self.name = "[Untitled]"
        } else {
            self.name = name
        }
    }
}
```

`AutomaticallyNamedDocument` 用一个非可失败构造器 `init(name:)` ,覆盖了基类的可失败构造器 `init?(name:)` 。因为子类用不同的方法处理了 `name` 属性的值为一个空字符串的这种情况。所以子类将不再需要一个可失败的构造器。

可失败构造器 init!

通常来说我们通过在 `init` 关键字后添加问号的方式来定义一个可失败构造器，但你也可以使用通过在 `init` 后面添加惊叹号的方式来定义一个可失败构造器 （`init!`），该可失败构造器将会构建一个特定类型的隐式解析可选类型的对象。

你可以在 `init?` 构造器中代理调用 `init!` 构造器，反之亦然。你也可以用 `init?` 覆盖 `init!` ，反之亦然。你还可以用 `init` 代理调用 `init!` ，但这会触发一个断言：是否 `init!` 构造器会触发构造失败?

必要构造器

在类的构造器前添加 `required` 修饰符表明所有该类的子类都必须实现该构造器：

```
class SomeClass {
    required init() {
        // 在这里添加该必要构造器的实现代码
    }
}
```

当子类覆盖基类的必要构造器时，必须在子类的构造器前同样添加 `required` 修饰符以确保当其它类继承该子类时，该构造器同为必要构造器。在覆盖基类的必要构造器时，不需要添加 `override` 修饰符：

```
class SomeSubclass: SomeClass {
    required init() {
        // 在这里添加子类必要构造器的实现代码
    }
}
```

> 注意：

> 如果子类继承的构造器能满足必要构造器的需求，则你无需显示的在子类中提供必要构造器的实现。

通过闭包和函数来设置属性的默认值

如果某个存储型属性的默认值需要特别的定制或准备，你就可以使用闭包或全局函数来为其属性提供定制的默认值。每当某个属性所属的新类型实例创建时，对应的闭包或函数会被调用，而它们的返回值会当做默认值赋值给这个属性。

这种类型的闭包或函数一般会创建一个跟属性类型相同的临时变量，然后修改它的值以满足预期的初始状态，最后将这个临时变量的值作为属性的默认值进行返回。

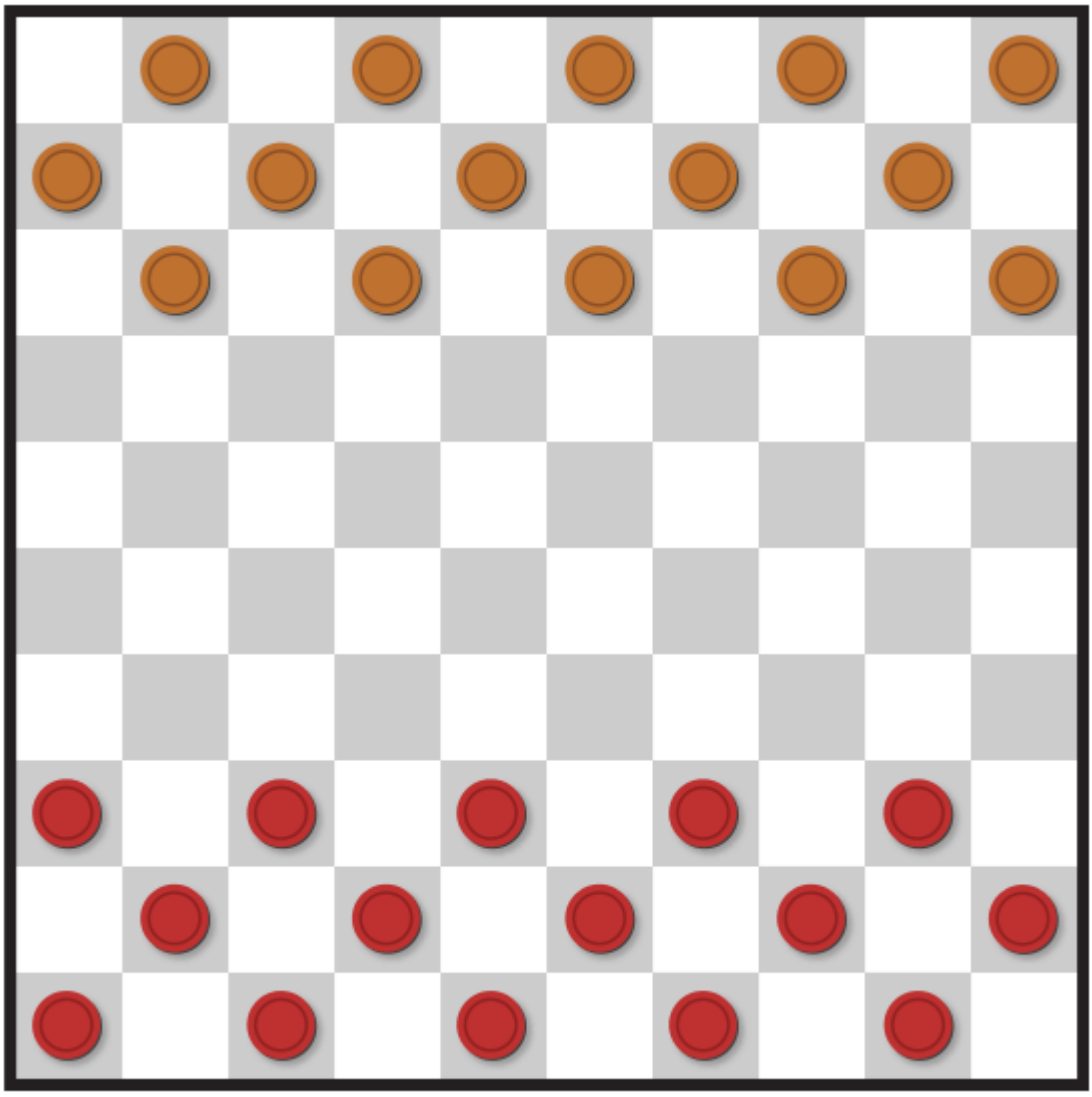
下面列举了闭包如何提供默认值的代码概要：

```
class SomeClass {
let someProperty: SomeType = {
        // 在这个闭包中给 someProperty 创建一个默认值
        // someValue 必须和 SomeType 类型相同
        return someValue
        }()
}
```

注意闭包结尾的大括号后面接了一对空的小括号。这是用来告诉 Swift 需要立刻执行此闭包。如果你忽略了这对括号，相当于是将闭包本身作为值赋值给了属性，而不是将闭包的返回值赋值给属性。

> 注意：
> 如果你使用闭包来初始化属性的值，请记住在闭包执行时，实例的其它部分都还没有初始化。这意味着你不能够在闭包里访问其它的属性，就算这个属性有默认值也不允许。同样，你也不能使用隐式的 `self` 属性，或者调用其它的实例方法。

下面例子中定义了一个结构体 `Checkerboard`，它构建了西洋跳棋游戏的棋盘：



图片 14.8 Image of Initialization_8.png

西洋跳棋游戏在一副黑白格交替的 10x10 的棋盘中进行。为了呈现这副游戏棋盘，`Checkerboard` 结构体定义了一个属性 `boardColors`，它是一个包含 100 个布尔值的数组。数组中的某元素布尔值为 `true` 表示对应的是一个黑格，布尔值为 `false` 表示对应的是一个白格。数组中第一个元素代表棋盘上左上角的格子，最后一个元素代表棋盘上右下角的格子。

`boardColor` 数组是通过一个闭包来初始化和组装颜色值的：

```
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]
    }
}
```

每当一个新的 Checkerboard 实例创建时，对应的赋值闭包会执行，一系列颜色值会被计算出来作为默认值赋值给 boardColors 。上面例子中描述的闭包将计算出棋盘中每个格子合适的颜色，将这些颜色值保存到一个临时数组 temporaryBoard 中，并在构建完成时将此数组作为闭包返回值返回。这个返回的值将保存到 boardColors 中，并可以通 squareIsBlackAtRow 这个工具函数来查询。

```
let board = Checkerboard()
println(board.squareIsBlackAtRow(0, column: 1))
// 输出 "true"
println(board.squareIsBlackAtRow(9, column: 9))
// 输出 "false"
```

15

析构过程（Deinitialization）

在一个类的实例被释放之前，析构函数被立即调用。用关键字 `deinit` 来标示析构函数，类似于初始化函数用 `init` 来标示。析构函数只适用于类类型。

析构过程原理

Swift 会自动释放不再需要的实例以释放资源。如自动引用计数 () 那一章描述，Swift 通过*自动引用计数*（ARC）处理实例的内存管理。通常当你的实例被释放时不需要手动地去清理。但是，当使用自己的资源时，你可能需要进行一些额外的清理。例如，如果创建了一个自定义的类来打开一个文件，并写入一些数据，你可能需要在类实例被释放之前关闭该文件。

在类的定义中，每个类最多只能有一个析构函数。析构函数不带任何参数，在写法上不带括号：

```
deinit {
    // 执行析构过程
}
```

析构函数是在实例释放发生前一步被自动调用。不允许主动调用自己的析构函数。子类继承了父类的析构函数，并且在子类析构函数实现的最后，父类的析构函数被自动调用。即使子类没有提供自己的析构函数，父类的析构函数也总是被调用。

因为直到实例的析构函数被调用时，实例才会被释放，所以析构函数可以访问所有请求实例的属性，并且根据那些属性可以修改它的行为（比如查找一个需要被关闭的文件的名称）。

析构函数操作

这里是一个析构函数操作的例子。这个例子是一个简单的游戏，定义了两种新类型，`Bank` 和 `Player`。`Bank` 结构体管理一个虚拟货币的流通，在这个流通中 `Bank` 永远不可能拥有超过 10,000 的硬币。在这个游戏中有且只能有一个 `Bank` 存在，因此 `Bank` 由带有静态属性和静态方法的结构体实现，从而存储和管理其当前的状态。

```
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` 根据它的 `coinsInBank` 属性来跟踪当前它拥有的硬币数量。银行还提供两个方法——`vendCoins` 和 `receiveCoins` ——用来处理硬币的分发和收集。

vendCoins 方法在 bank 分发硬币之前检查是否有足够的硬币。如果没有足够多的硬币，Bank 返回一个比请求时小的数字(如果没有硬币留在 bank 中就返回 0)。vendCoins 方法声明 numberOfCoinsToVend 为一个变量参数，这样就可以在方法体的内部修改数字，而不需要定义一个新的变量。vendCoins 方法返回一个整型值，表明了提供的硬币的实际数目。

receiveCoins 方法只是将 bank 的硬币存储和接收到的硬币数目相加，再保存回 bank。

Player 类描述了游戏中的一个玩家。每一个 player 在任何时刻都有一定数量的硬币存储在他们的钱包中。这通过 player 的 coinsInPurse 属性来体现：

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.vendCoins(coins)
    }
    func winCoins(coins: Int) {
        coinsInPurse += Bank.vendCoins(coins)
    }
    deinit {
        Bank.receiveCoins(coinsInPurse)
    }
}
```

每个 Player 实例都由一个指定数目硬币组成的启动额度初始化，这些硬币在 bank 初始化的过程中得到。如果没有足够的硬币可用，Player 实例可能收到比指定数目少的硬币。

Player 类定义了一个 winCoins 方法，该方法从银行获取一定数量的硬币，并把它们添加到玩家的钱包。Player 类还实现了一个析构函数，这个析构函数在 Player 实例释放前一步被调用。这里析构函数只是将玩家的所有硬币都返回给银行：

```
var playerOne: Player? = Player(coins: 100)
println("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
// 输出 "A new player has joined the game with 100  coins"
println("There are now \(Bank.coinsInBank) coins left   in the bank")
// 输出 "There are now 9900 coins left in the bank"
```

一个新的 Player 实例随着一个 100 个硬币（如果有）的请求而被创建。这 个Player 实例存储在一个名为 playerOne 的可选 Player 变量中。这里使用一个可选变量，是因为玩家可以随时离开游戏。设置为可选使得你可以跟踪当前是否有玩家在游戏中。

因为 playerOne 是可选的，所以由一个感叹号（！）来修饰，每当其 winCoins 方法被调用时，coinsInPurse 属性被访问并打印出它的默认硬币数目。

```
playerOne!.winCoins(2_000)
println("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")
// 输出 "PlayerOne won 2000 coins & now has 2100 coins"
println("The bank now only has \(Bank.coinsInBank) coins left")
// 输出 "The bank now only has 7900 coins left"
```

这里，player 已经赢得了 2,000 硬币。player 的钱包现在有 2,100 硬币，bank 只剩余 7,900 硬币。

```
playerOne = nil
println("PlayerOne has left the game")
// 输出 "PlayerOne has left the game"
println("The bank now has \(Bank.coinsInBank) coins")
// 输出 "The bank now has 10000 coins"
```

玩家现在已经离开了游戏。这表明是要将可选的 playerOne 变量设置为 nil ，意思是“没有 Player 实例”。当这种情况发生的时候， playerOne 变量对 Player 实例的引用被破坏了。没有其它属性或者变量引用 Player 实例，因此为了清空它占用的内存从而释放它。在这发生前一步，其析构函数被自动调用，其硬币被返回到银行。

16

自动引用计数

Swift 使用自动引用计数（ARC）这一机制来跟踪和管理你的应用程序的内存。通常情况下，Swift 的内存管理机制会一直起着作用，你无须自己来考虑内存的管理。ARC 会在类的实例不再被使用时，自动释放其占用的内存。

然而，在少数情况下，ARC 为了能帮助你管理内存，需要更多的关于你的代码之间关系的信息。本章描述了这些情况，并且为你示范怎样启用 ARC 来管理你的应用程序的内存。

> 注意:
> 引用计数仅仅应用于类的实例。结构体和枚举类型是值类型，不是引用类型，也不是通过引用的方式存储和传递。

自动引用计数的工作机制

当你每次创建一个类的新的实例的时候，ARC 会分配一大块内存用来储存实例的信息。内存中会包含实例的类型信息，以及这个实例所有相关属性的值。此外，当实例不再被使用时，ARC 释放实例所占用的内存，并让释放的内存能挪作他用。这确保了不再被使用的实例，不会一直占用内存空间。

然而，当 ARC 收回和释放了正在被使用中的实例，该实例的属性和方法将不能再被访问和调用。实际上，如果你试图访问这个实例，你的应用程序很可能会崩溃。

为了确保使用中的实例不会被销毁，ARC 会跟踪和计算每一个实例正在被多少属性，常量和变量所引用。哪怕实例的引用数为一，ARC都不会销毁这个实例。

为了使之成为可能，无论你将实例赋值给属性，常量或者是变量，属性，常量或者变量，都会对此实例创建强引用。之所以称之为强引用，是因为它会将实例牢牢的保持住，只要强引用还在，实例是不允许被销毁的。

自动引用计数实践

下面的例子展示了自动引用计数的工作机制。例子以一个简单的 `Person` 类开始，并定义了一个叫 `name` 的常量属性：

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        println("\(name) is being initialized")
    }
    deinit {
        println("\(name) is being deinitialized")
    }
}
```

Person 类有一个构造函数，此构造函数为实例的 name 属性赋值并打印出信息，以表明初始化过程生效。Person 类同时也拥有析构函数，同样会在实例被销毁的时候打印出信息。

接下来的代码片段定义了三个类型为 Person? 的变量，用来按照代码片段中的顺序，为新的 Person 实例建立多个引用。由于这些变量是被定义为可选类型（Person?，而不是Person），它们的值会被自动初始化为 nil，目前还不会引用到 Person 类的实例。

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

现在你可以创建 Person 类的新实例，并且将它赋值给三个变量其中的一个：

```
reference1 = Person(name: "John Appleseed")
// prints "John Appleseed is being initialized”
```

应当注意到当你调用 Person 类的构造函数的时候，"John Appleseed is being initialized”会被打印出来。由此可以确定构造函数被执行。

由于 Person 类的新实例被赋值给了 reference1 变量，所以 reference1 到 Person 类的新实例之间建立了一个强引用。正是因为这个强引用，ARC 会保证 Person 实例被保持在内存中不被销毁。

如果你将同样的 Person 实例也赋值给其他两个变量，该实例又会多出两个强引用：

```
reference2 = reference1
reference3 = reference1
```

现在这个 Person 实例已经有三个强引用了。

如果你通过给两个变量赋值 nil 的方式断开两个强引用（ ）包括最先的那个强引用），只留下一个强引用，Person 实例不会被销毁：

```
reference1 = nil
reference2 = nil
```

ARC 会在第三个，也即最后一个强引用被断开的时候，销毁 Person 实例，这也意味着你不再使用这个 Person 实例：

```
reference3 = nil
// prints "John Appleseed is being deinitialized"
```

类实例之间的循环强引用

在上面的例子中，ARC 会跟踪你所新创建的 Person 实例的引用数量，并且会在 Person 实例不再被需要时销毁它。

然而，我们可能会写出这样的代码，一个类永远不会有0个强引用。这种情况发生在两个类实例互相保持对方的强引用，并让对方不被销毁。这就是所谓的循环强引用。

你可以通过定义类之间的关系为弱引用或者无主引用，以此替代强引用，从而解决循环强引用的问题。具体的过程在解决类实例之间的循环强引用 (页 167)中有描述。不管怎样，在你学习怎样解决循环强引用之前，很有必要了解一下它是怎样产生的。

下面展示了一个不经意产生循环强引用的例子。例子定义了两个类： `Person` 和 `Apartment` ，用来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") }
}
```

每一个 `Person` 实例有一个类型为 `String` ，名字为 `name` 的属性，并有一个可选的初始化为 `nil` 的 `apartment` 属性。 `apartment` 属性是可选的，因为一个人并不总是拥有公寓。

类似的，每个 `Apartment` 实例有一个叫 `number` ，类型为 `Int` 的属性，并有一个可选的初始化为 `nil` 的 `tenant` 属性。 `tenant` 属性是可选的，因为一栋公寓并不总是有居民。

这两个类都定义了析构函数，用以在类实例被析构的时候输出信息。这让你能够知晓 `Person` 和 `Apartment` 的实例是否像预期的那样被销毁。

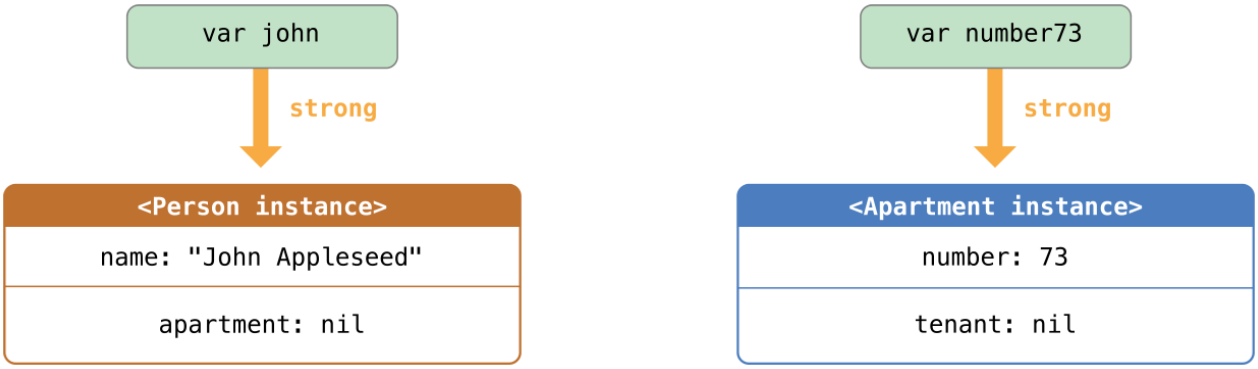
接下来的代码片段定义了两个可选类型的变量 `john` 和 `number73` ,并分别被设定为下面的 `Apartment` 和 `Person` 的实例。这两个变量都被初始化为 `nil` ，并为可选的:

```
var john: Person?
var number73: Apartment?
```

现在你可以创建特定的 `Person` 和 `Apartment` 实例并将类实例赋值给 `john` 和 `number73` 变量:

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

在两个实例被创建和赋值后，下图表现了强引用的关系。变量 `john` 现在有一个指向 `Person` 实例的强引用，而变量 `number73` 有一个指向 `Apartment` 实例的强引用:

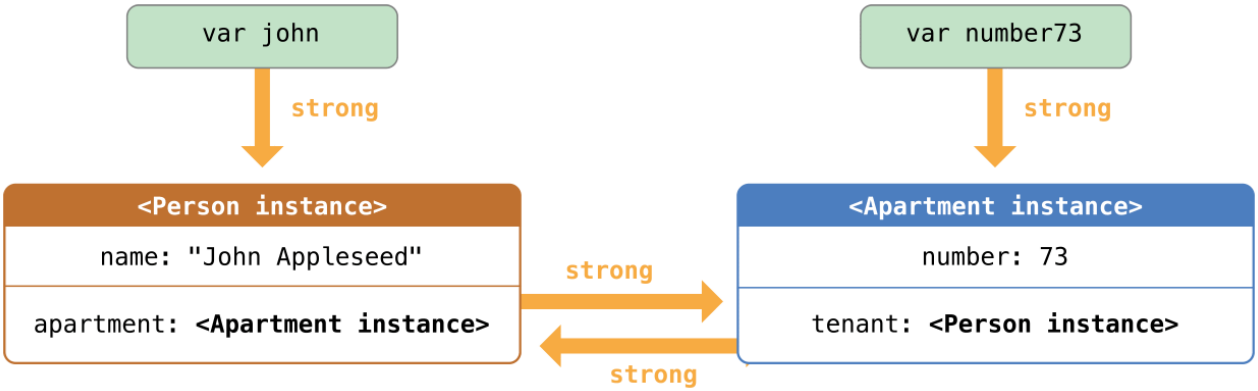


图片 16.1 image of Automatic_Reference_Counting_1.png

现在你能够将这两个实例关联在一起，这样人就能有公寓住了，而公寓也有了房客。注意感叹号是用来展开和访问可选变量 john 和 number73 中的实例，这样实例的属性才能被赋值：

```
john!.apartment = number73
number73!.tenant = john
```

在将两个实例联系在一起之后，强引用的关系如图所示：



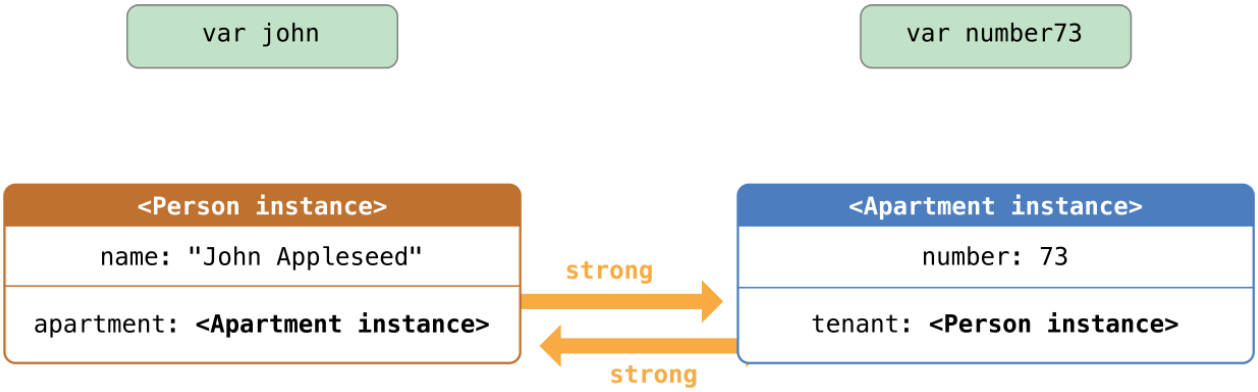
图片 16.2 image of Automatic_Reference_Counting_2.png

不幸的是，将这两个实例关联在一起之后，一个循环强引用被创建了。Person 实例现在有了一个指向 Apartment 实例的强引用，而 Apartment 实例也有了一个指向 Person 实例的强引用。因此，当你断开 john 和 number73 变量所持有的强引用时，引用计数并不会降为 0，实例也不会被 ARC 销毁：

```
john = nil
number73 = nil
```

注意，当你把这两个变量设为 nil 时，没有任何一个析构函数被调用。强引用循环阻止了 Person 和 Apartment 类实例的销毁，并在你的应用程序中造成了内存泄漏。

在你将 john 和 number73 赋值为 nil 后，强引用关系如下图：



图片 16.3 image of Automatic_Reference_Counting_3.png

Person 和 Apartment 实例之间的强引用关系保留了下来并且不会被断开。

()

解决实例之间的循环强引用

Swift 提供了两种办法用来解决你在使用类的属性时所遇到的循环强引用问题：弱引用（weak reference）和无主引用（unowned reference）。

弱引用和无主引用允许循环引用中的一个实例引用另外一个实例而不保持强引用。这样实例能够互相引用而不产生循环强引用。

对于生命周期中会变为 nil 的实例使用弱引用。相反的，对于初始化赋值后再也不会被赋值为 nil 的实例，使用无主引用。

弱引用

弱引用不会牢牢保持住引用的实例，并且不会阻止 ARC 销毁被引用的实例。这种行为阻止了引用变为循环强引用。声明属性或者变量时，在前面加上 weak 关键字表明这是一个弱引用。

在实例的生命周期中，如果某些时候引用没有值，那么弱引用可以阻止循环强引用。如果引用总是有值，则可以使用无主引用，在[无主引用](#unowned references)中有描述。在上面 Apartment 的例子中，一个公寓的生命周期中，有时是没有“居民”的，因此适合使用弱引用来解决循环强引用。

> 注意:
> 弱引用必须被声明为变量，表明其值能在运行时被修改。弱引用不能被声明为常量。

因为弱引用可以没有值，你必须将每一个弱引用声明为可选类型。可选类型是在 Swift 语言中推荐的用来表示可能没有值的类型。

因为弱引用不会保持所引用的实例，即使引用存在，实例也有可能被销毁。因此，ARC 会在引用的实例被销毁后自动将其赋值为 nil 。你可以像其他可选值一样，检查弱引用的值是否存在，你永远也不会遇到被销毁了而不存在的实例。

下面的例子跟上面 Person 和 Apartment 的例子一致，但是有一个重要的区别。这一次， Apartment 的 tenant 属性被声明为弱引用：

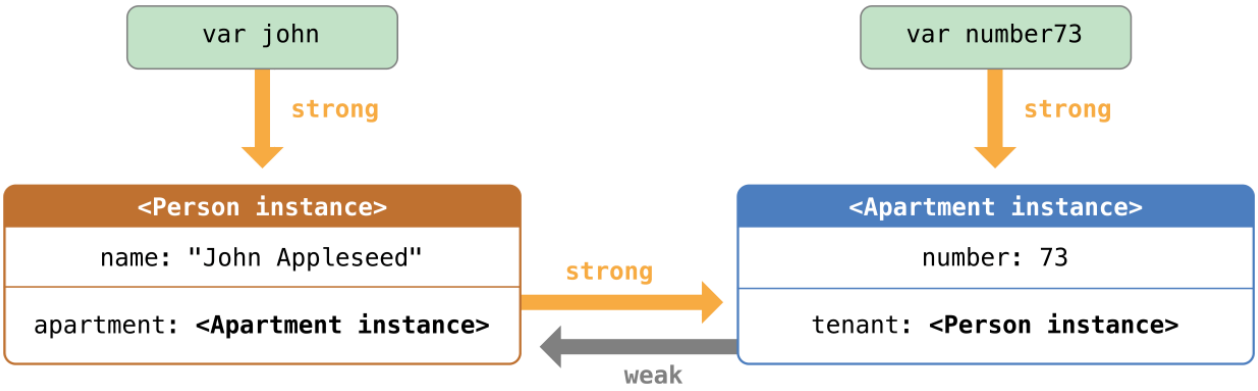
```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { println("\(name) is being deinitialized") }
}
```

```
class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    weak var tenant: Person?
    deinit { println("Apartment #\(number) is being deinitialized") }
}
```

然后跟之前一样，建立两个变量（john和number73）之间的强引用，并关联两个实例：

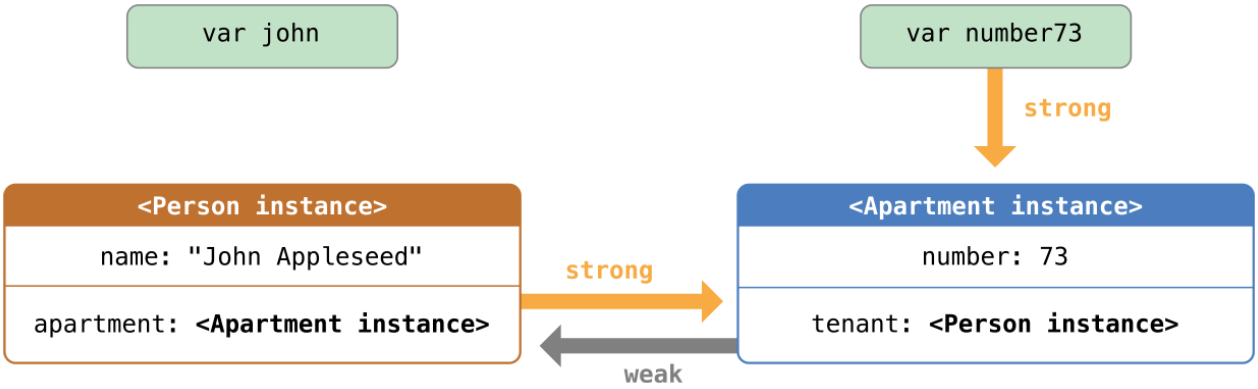
```
var john: Person?
var number73: Apartment?
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
john!.apartment = number73
number73!.tenant = john
```

现在，两个关联在一起的实例的引用关系如下图所示：



图片 16.4 image of Automatic_Reference_Counting_4.png

Person 实例依然保持对 Apartment 实例的强引用，但是 Apartment 实例只是对 Person 实例的弱引用。这意味着当你断开 john 变量所保持的强引用时，再也没有指向 Person 实例的强引用了：

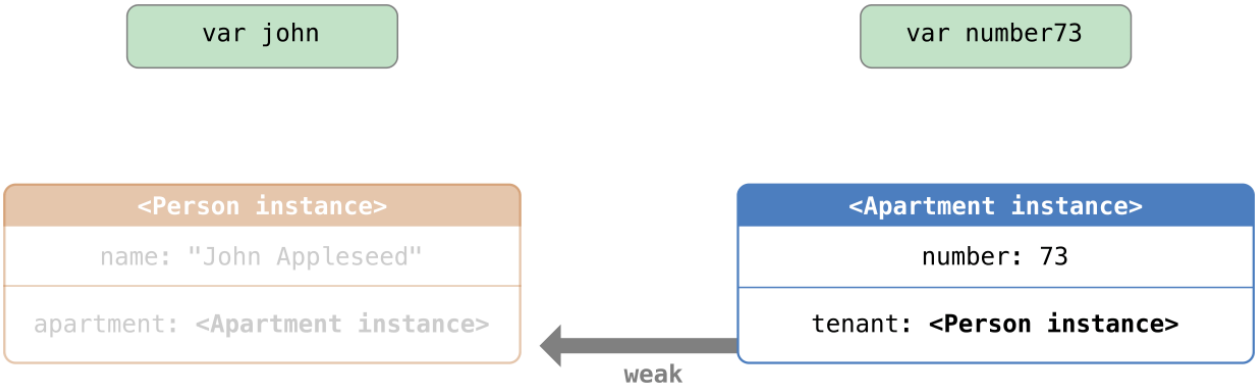


图片 16.5 image of Automatic_Reference_Counting_5.png

由于再也没有指向 Person 实例的强引用，该实例会被销毁：

```
john = nil
// prints "John Appleseed is being deinitialized"
```

唯一剩下的指向 Apartment 实例的强引用来自于变量 number73 。如果你断开这个强引用，再也没有指向 Apartment 实例的强引用了：



图片 16.6 image of Automatic_Reference_Counting_6.png

由于再也没有指向 Apartment 实例的强引用，该实例也会被销毁：

```
number73 = nil
// prints "Apartment #73 is being deinitialized"
```

上面的两段代码展示了变量 john 和 number73 在被赋值为 nil 后， Person 实例和 Apartment 实例的析构函数都打印出“销毁”的信息。这证明了引用循环被打破了。

()

无主引用

和弱引用类似，无主引用不会牢牢保持住引用的实例。和弱引用不同的是，无主引用是永远有值的。因此，无主引用总是被定义为非可选类型（non-optional type）。你可以在声明属性或者变量时，在前面加上关键字 `unowned` 表示这是一个无主引用。

由于无主引用是非可选类型，你不需要在使用它的时候将它展开。无主引用总是可以被直接访问。不过 ARC 无法在实例被销毁后将无主引用设为 `nil`，因为非可选类型的变量不允许被赋值为 `nil`。

> 注意:
> 如果你试图在实例被销毁后，访问该实例的无主引用，会触发运行时错误。使用无主引用，你必须确保引用始终指向一个未销毁的实例。
> 还需要注意的是如果你试图访问实例已经被销毁的无主引用，程序会直接崩溃，而不会发生无法预期的行为。所以你应当避免这样的事情发生。

下面的例子定义了两个类，`Customer` 和 `CreditCard`，模拟了银行客户和客户的信用卡。这两个类中，每一个都将另外一个类的实例作为自身的属性。这种关系会潜在的创造循环强引用。

`Customer` 和 `CreditCard` 之间的关系与前面弱引用例子中 `Apartment` 和 `Person` 的关系截然不同。在这个数据模型中，一个客户可能有或者没有信用卡，但是一张信用卡总是关联着一个客户。为了表示这种关系，`Customer` 类有一个可选类型的 `card` 属性，但是 `CreditCard` 类有一个非可选类型的 `customer` 属性。

此外，只能通过将一个 `number` 值和 `customer` 实例传递给 `CreditCard` 构造函数的方式来创建 `CreditCard` 实例。这样可以确保当创建 `CreditCard` 实例时总是有一个 `customer` 实例与之关联。

由于信用卡总是关联着一个客户，因此将 `customer` 属性定义为无主引用，用以避免循环强引用：

```
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") }
}
```

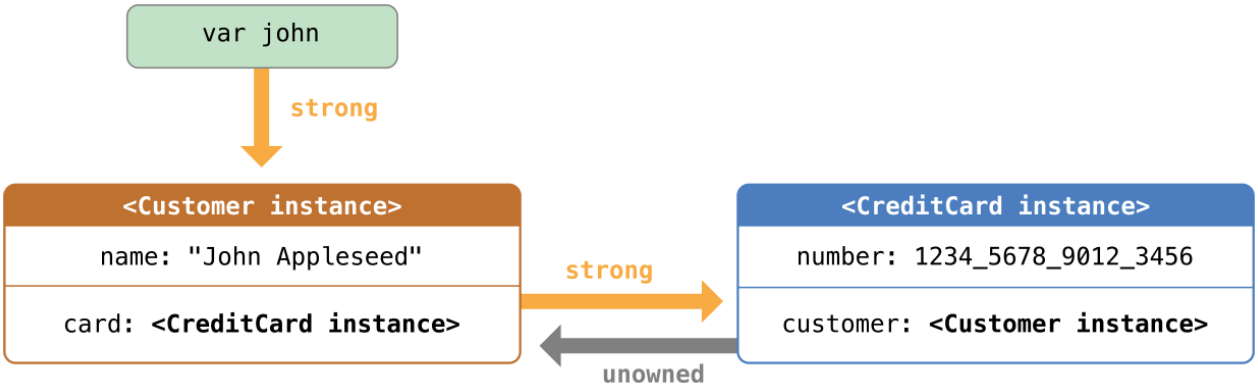
下面的代码片段定义了一个叫 john 的可选类型 Customer 变量，用来保存某个特定客户的引用。由于是可选类型，所以变量被初始化为 nil 。

```
var john: Customer?
```

现在你可以创建 Customer 类的实例，用它初始化 CreditCard 实例，并将新创建的 CreditCard 实例赋值为客户的 card 属性。

```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

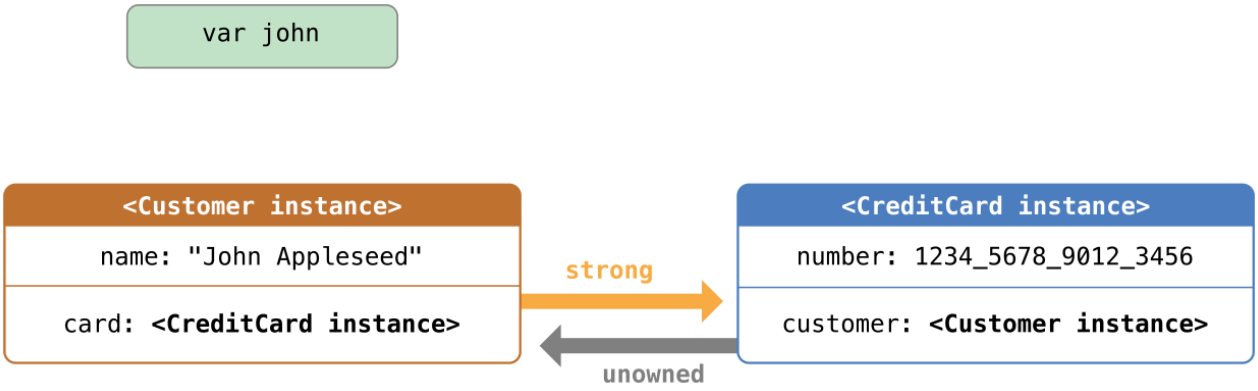
在你关联两个实例后，它们的引用关系如下图所示：



图片 16.7 image of Automatic_Reference_Counting_7.png

Customer 实例持有对 CreditCard 实例的强引用，而 CreditCard 实例持有对 Customer 实例的无主引用。

由于 customer 的无主引用，当你断开 john 变量持有的强引用时，再也没有指向 Customer 实例的强引用了：



图片 16.8 image of Automatic_Reference_Counting_8.png

由于再也没有指向 Customer 实例的强引用，该实例被销毁了。其后，再也没有指向 CreditCard 实例的强引用，该实例也随之被销毁了：

```
john = nil
// prints "John Appleseed is being deinitialized"
// prints "Card #1234567890123456 is being deinitialized"
```

最后的代码展示了在 john 变量被设为 nil 后 Customer 实例和 CreditCard 实例的构造函数都打印出了“销毁”的信息。

无主引用以及隐式解析可选属性

上面弱引用和无主引用的例子涵盖了两种常用的需要打破循环强引用的场景。

Person 和 Apartment 的例子展示了两个属性的值都允许为 nil，并会潜在的产生循环强引用。这种场景最适合用弱引用来解决。

Customer 和 CreditCard 的例子展示了一个属性的值允许为 nil，而另一个属性的值不允许为 nil，并会潜在的产生循环强引用。这种场景最适合通过无主引用来解决。

然而，存在着第三种场景，在这种场景中，两个属性都必须有值，并且初始化完成后不能为 nil 。在这种场景中，需要一个类使用无主属性，而另外一个类使用隐式解析可选属性。

这使两个属性在初始化完成后能被直接访问（不需要可选展开），同时避免了循环引用。这一节将为你展示如何建立这种关系。

下面的例子定义了两个类， Country 和 City，每个类将另外一个类的实例保存为属性。在这个模型中，每个国家必须有首都，而每一个城市必须属于一个国家。为了实现这种关系， Country 类拥有一个 capitalCity 属性，而 City 类有一个 country 属性：

```
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
    }
}
```

为了建立两个类的依赖关系， City 的构造函数有一个 Country 实例的参数，并且将实例保存为 country 属性。

Country 的构造函数调用了 City 的构造函数。然而，只有 Country 的实例完全初始化完后， Country 的构造函数才能把 self 传给 City 的构造函数。（在两段式构造过程中有具体描述 ()）

为了满足这种需求，通过在类型结尾处加上感叹号（City!）的方式，将 Country 的 capitalCity 属性声明为隐式解析可选类型的属性。这表示像其他可选类型一样， capitalCity 属性的默认值为 nil ，但是不需要展开它的值就能访问它。（在隐式解析可选类型中有描述 ()）

由于 capitalCity 默认值为 nil ，一旦 Country 的实例在构造函数中给 name 属性赋值后，整个初始化过程就完成了。这代表一旦 name 属性被赋值后， Country 的构造函数就能引用并传递隐式的 self 。 Country 的构造函数在赋值 capitalCity 时，就能将 self 作为参数传递给 City 的构造函数。

以上的意义在于你可以通过一条语句同时创建 Country 和 City 的实例，而不产生循环强引用，并且 capitalCity 的属性能被直接访问，而不需要通过感叹号来展开它的可选值：

```
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"
```

在上面的例子中，使用隐式解析可选值的意义在于满足了两个类构造函数的需求。 capitalCity 属性在初始化完成后，能像非可选值一样使用和存取同时还避免了循环强引用。

闭包引起的循环强引用

前面我们看到了循环强引用环是在两个类实例属性互相保持对方的强引用时产生的，还知道了如何用弱引用和无主引用来打破循环强引用。

循环强引用还会发生在当你将一个闭包赋值给类实例的某个属性，并且这个闭包体中又使用了实例。这个闭包体中可能访问了实例的某个属性，例如 self.someProperty ，或者闭包中调用了实例的某个方法，例如 self.someMethod 。这两种情况都导致了闭包 “捕获" self ，从而产生了循环强引用。

循环强引用的产生，是因为闭包和类相似，都是引用类型。当你把一个闭包赋值给某个属性时，你也把一个引用赋值给了这个闭包。实质上，这跟之前的问题是一样的－两个强引用让彼此一直有效。但是，和两个类实例不同，这次一个是类实例，另一个是闭包。

Swift 提供了一种优雅的方法来解决这个问题，称之为闭包占用列表（closuer capture list）。同样的，在学习如何用闭包占用列表破坏循环强引用之前，先来了解一下循环强引用是如何产生的，这对我们是很有帮助的。

下面的例子为你展示了当一个闭包引用了 self 后是如何产生一个循环强引用的。例子中定义了一个叫 HTMLElement 的类，用一种简单的模型表示 HTML 中的一个单独的元素：

```
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 类定义了一个 name 属性来表示这个元素的名称，例如代表段落的"p"，或者代表换行的"br"。HTMLElement 还定义了一个可选属性 text，用来设置和展现 HTML 元素的文本。

除了上面的两个属性，HTMLElement 还定义了一个 lazy 属性 asHTML。这个属性引用了一个闭包，将 name 和 text 组合成 HTML 字符串片段。该属性是 () -> String 类型，或者可以理解为“一个没有参数，返回 String 的函数”。

默认情况下，闭包赋值给了 asHTML 属性，这个闭包返回一个代表 HTML 标签的字符串。如果 text 值存在，该标签就包含可选值 text；如果 text 不存在，该标签就不包含文本。对于段落元素，根据 text 是"some text"还是 nil，闭包会返回" some text "或者" "。

可以像实例方法那样去命名、使用 asHTML 属性。然而，由于 asHTML 是闭包而不是实例方法，如果你想改变特定元素的 HTML 处理的话，可以用自定义的闭包来取代默认值。

> 注意:
> asHTML 声明为 lazy 属性，因为只有当元素确实需要处理为HTML输出的字符串时，才需要使用 asHTML。也就是说，在默认的闭包中可以使用 self，因为只有当初始化完成以及 self 确实存在后，才能访问 lazy 属性。

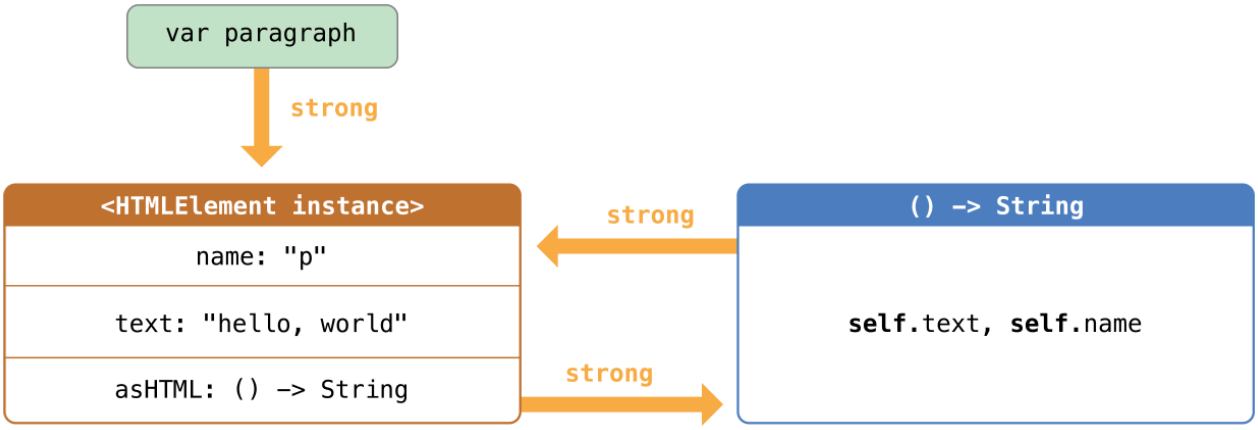
HTMLElement 类只提供一个构造函数，通过 name 和 text （如果有的话）参数来初始化一个元素。该类也定义了一个析构函数，当 HTMLElement 实例被销毁时，打印一条消息。

下面的代码展示了如何用 HTMLElement 类创建实例并打印消息。

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
// prints"hello, world"
```

> 注意:
> 上面的 paragraph 变量定义为 可选HTMLElement，因此我们可以赋值 nil 给它来演示循环强引用。

不幸的是，上面写的 HTMLElement 类产生了类实例和 asHTML 默认值的闭包之间的循环强引用。循环强引用如下图所示：



图片 16.9 image of Automatic_Reference_Counting_9.png

实例的 asHTML 属性持有闭包的强引用。但是，闭包在其闭包体内使用了 self （引用了 self.name 和 self.text ），因此闭包捕获了 self ，这意味着闭包又反过来持有了 HTMLElement 实例的强引用。这样两个对象就产生了循环强引用。（更多关于闭包捕获值的信息，请参考值捕获 () ）。

> 注意:
> 虽然闭包多次使用了 self ，它只捕获 HTMLElement 实例的一个强引用。

如果设置 paragraph 变量为 nil ，打破它持有的 HTMLElement 实例的强引用， HTMLElement 实例和它的闭包都不会被销毁，也是因为循环强引用：

```
paragraph = nil
```

注意 HTMLElementdeinitializer 中的消息并没有别打印，证明了 HTMLElement 实例并没有被销毁。

()

解决闭包引起的循环强引用

在定义闭包时同时定义捕获列表作为闭包的一部分，通过这种方式可以解决闭包和类实例之间的循环强引用。捕获列表定义了闭包体内捕获一个或者多个引用类型的规则。跟解决两个类实例间的循环强引用一样，声明每个捕获的引用为弱引用或无主引用，而不是强引用。应当根据代码关系来决定使用弱引用还是无主引用。

> 注意:
> Swift 有如下要求：只要在闭包内使用 self 的成员，就要用 self.someProperty 或者 self.someMethod （而不只是 someProperty 或 someMethod ）。这提醒你可能会不小心就捕获了 self 。

定义捕获列表

捕获列表中的每个元素都是由 weak 或者 unowned 关键字和实例的引用（如 self 或 someInstance ）成对组成。每一对都在方括号中，通过逗号分开。

捕获列表放置在闭包参数列表和返回类型之前：

```
lazy var someClosure: (Int, String) -> String = {
    [unowned self] (index: Int, stringToProcess: String) -> String in
    // closure body goes here
}
```

如果闭包没有指定参数列表或者返回类型，则可以通过上下文推断，那么可以捕获列表放在闭包开始的地方，跟着是关键字 in ：

```
lazy var someClosure: () -> String = {
    [unowned self] in
    // closure body goes here
}
```

弱引用和无主引用

当闭包和捕获的实例总是互相引用时并且总是同时销毁时，将闭包内的捕获定义为无主引用。

相反的，当捕获引用有时可能会是 nil 时，将闭包内的捕获定义为弱引用。弱引用总是可选类型，并且当引用的实例被销毁后，弱引用的值会自动置为 nil 。这使我们可以在闭包内检查它们是否存在。

> 注意：
> 如果捕获的引用绝对不会置为 nil ，应该用无主引用，而不是弱引用。

前面的 HTMLElement 例子中，无主引用是正确的解决循环强引用的方法。这样编写 HTMLElement 类来避免循环强引用：

```
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 {
```

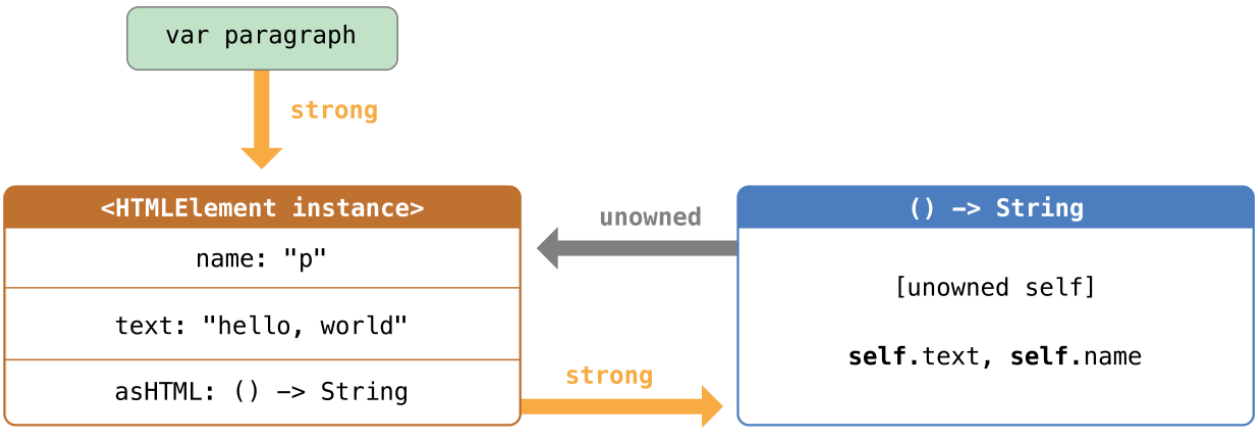
```
        println("\(name) is being deinitialized")
    }
}
```

上面的 HTMLElement 实现和之前的实现一致，只是在 asHTML 闭包中多了一个捕获列表。这里，捕获列表是 [unowned self] ，表示“用无主引用而不是强引用来捕获 self ”。

和之前一样，我们可以创建并打印 HTMLElement 实例：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
// prints "<p>hello, world</p>"
```

使用捕获列表后引用关系如下图所示：



图片 16.10 image of Automatic_Reference_Counting_10.png

这一次，闭包以无主引用的形式捕获 self ，并不会持有 HTMLElement 实例的强引用。如果将 paragraph 赋值为 nil ， HTMLElement 实例将会被销毁，并能看到它的析构函数打印出的消息。

```
paragraph = nil
// prints "p is being deinitialized"
```

17

Optional Chaining

可选链（Optional Chaining）是一种可以请求和调用属性、方法及下标脚本的过程，它的可选性体现于请求或调用的目标当前可能为空（ nil ）。如果可选的目标有值，那么调用就会成功；相反，如果选择的目标为空（ nil ），则这种调用将返回空（ nil ）。多次请求或调用可以被链接在一起形成一个链，如果任何一个节点为空（ nil ）将导致整个链失效。

> 注意：
> Swift 的可选链和 Objective-C 中的消息为空有些相像，但是 Swift 可以使用在任意类型中，并且失败与否可以被检测到。

()

可选链可替代强制解析

通过在想调用的属性、方法、或下标脚本的可选值（ optional value ）（非空）后面放一个问号，可以定义一个可选链。这一点很像在可选值后面放一个叹号来强制拆得其封包内的值。它们的主要的区别在于当可选值为空时可选链即刻失败，然而一般的强制解析将会引发运行时错误。

为了反映可选链可以调用空（ nil ），不论你调用的属性、方法、下标脚本等返回的值是不是可选值，它的返回结果都是一个可选值。你可以利用这个返回值来检测你的可选链是否调用成功，有返回值即成功，返回nil则失败。

调用可选链的返回结果与原本的返回结果具有相同的类型，但是原本的返回结果被包装成了一个可选值，当可选链调用成功时，一个应该返回 Int 的属性将会返回 Int? 。

下面几段代码将解释可选链和强制解析的不同。

首先定义两个类 Person 和 Residence 。

```
class Person {
    var residence: Residence?
}
class Residence {
    var numberOfRooms = 1
}
```

Residence 具有一个 Int 类型的 numberOfRooms ，其值为 1。 Person 具有一个可选 residence 属性，它的类型是 Residence? 。

如果你创建一个新的 Person 实例，它的 residence 属性由于是被定义为可选型的，此属性将默认初始化为空：

```
let john = Person()
```

如果你想使用感叹号（ ! ）强制解析获得这个人 residence 属性 numberOfRooms 属性值，将会引发运行时错误，因为这时没有可以供解析的 residence 值。

```
let roomCount = john.residence!.numberOfRooms
//将导致运行时错误
```

当 john.residence 不是 nil 时，会运行通过，且会将 roomCount 设置为一个 int 类型的合理值。然而，如上所述，当 residence 为空时，这个代码将会导致运行时错误。

可选链提供了一种另一种获得 numberOfRooms 的方法。利用可选链，使用问号来代替原来！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.")
}
// 打印 "Unable to retrieve the number of rooms.
```

这告诉 Swift 来链接可选 residence? 属性，如果 residence 存在则取回 numberOfRooms 的值。

因为这种尝试获得 numberOfRooms 的操作有可能失败，可选链会返回 Int? 类型值，或者称作“可选 Int ”。当 residence 是空的时候（上例），选择 Int 将会为空，因此会出现无法访问 numberOfRooms 的情况。

要注意的是，即使numberOfRooms是非可选 Int （ Int? ）时这一点也成立。只要是通过可选链的请求就意味着最后 numberOfRooms 总是返回一个 Int? 而不是 Int 。

你可以自己定义一个 Residence 实例给 john.residence ，这样它就不再为空了：

```
john.residence = Residence()
```

john.residence 现在有了实际存在的实例而不是nil了。如果你想使用和前面一样的可选链来获得 numberOfRooms ，它将返回一个包含默认值 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.")
}
// 打印 "John's residence has 1 room(s)"。
```

为可选链定义模型类

你可以使用可选链来多层调用属性，方法，和下标脚本。这让你可以利用它们之间的复杂模型来获取更底层的属性，并检查是否可以成功获取此类底层属性。

后面的代码定义了四个将在后面使用的模型类，其中包括多层可选链。这些类是由上面的 Person 和 Residence 模型通过添加一个 Room 和一个 Address 类拓展来。

Person 类定义与之前相同。

```
class Person {
    var residence: Residence?
}
```

Residence 类比之前复杂些。这次，它定义了一个变量 rooms ，它被初始化为一个 Room[] 类型的空数组：

```
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?
}
```

因为 Residence 存储了一个 Room 实例的数组，它的 numberOfRooms 属性值不是一个固定的存储值，而是通过计算而来的。 numberOfRooms 属性值是由返回 rooms 数组的 count 属性值得到的。

为了能快速访问 rooms 数组， Residence 定义了一个只读的下标脚本，通过插入数组的元素角标就可以成功调用。如果该角标存在，下标脚本则将该元素返回。

Residence 中也提供了一个 printNumberOfRooms 的方法，即简单的打印房间个数。

最后， Residence 定义了一个可选属性叫 address （ address? ）。 Address 类的属性将在后面定义。 用于 rooms 数组的 Room 类是一个很简单的类，它只有一个 name 属性和一个设定 room 名的初始化器。

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
```

这个模型中的最终类叫做 Address 。它有三个类型是 String? 的可选属性。前面两个可选属性 buildingName 和 buildingNumber 作为地址的一部分，是定义某个建筑物的两种方式。第三个属性 street ，用于命名地址的街道名：

```
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 类还提供了一个 buildingIdentifier 的方法，它的返回值类型为 String? 。这个方法检查 buildingName 和 buildingNumber 的属性，如果 buildingName 有值则将其返回，或者如果 buildingNumber 有值则将其返回，再或如果没有一个属性有值，返回空。

通过可选链调用属性

正如上面“ 可选链可替代强制解析 (页 179)”中所述，你可以利用可选链的可选值获取属性，并且检查属性是否获取成功。然而，你不能使用可选链为属性赋值。

使用上述定义的类来创建一个人实例，并再次尝试后去它的 numberOfRooms 属性：

```
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.")
}
// 打印 "Unable to retrieve the number of rooms。
```

由于 john.residence 是空，所以这个可选链和之前一样失败了，但是没有运行时错误。

通过可选链调用方法

你可以使用可选链的来调用可选值的方法并检查方法调用是否成功。即使这个方法没有返回值，你依然可以使用可选链来达成这一目的。

Residence 的 printNumberOfRooms 方法会打印 numberOfRooms 的当前值。方法如下：

```
func printNumberOfRooms(){
    println(“The number of rooms is \(numberOfRooms)”)
}
```

这个方法没有返回值。但是，没有返回值类型的函数和方法有一个隐式的返回值类型 Void （参见Function Without Return Values）。

如果你利用可选链调用此方法，这个方法的返回值类型将是 Void? ，而不是 Void ，因为当通过可选链调用方法时返回值总是可选类型（optional type）。即使这个方法本身没有定义返回值，你也可以使用 if 语句来检查是否能成功调用 printNumberOfRooms 方法：如果方法通过可选链调用成功， printNumberOfRooms 的隐式返回值将会是 Void ，如果没有成功，将返回 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.")
```

```
}
// 打印 "It was not possible to print the number of rooms."。
```

使用可选链调用下标脚本

你可以使用可选链来尝试从下标脚本获取值并检查下标脚本的调用是否成功，然而，你不能通过可选链来设置下标脚本。

> 注意：
> 当你使用可选链来获取下标脚本的时候，你应该将问号放在下标脚本括号的前面而不是后面。可选链的问号一般直接跟在表达语句的后面。

下面这个例子用在 `Residence` 类中定义的下标脚本来获取 `john.residence` 数组中第一个房间的名字。因为 `john.residence` 现在是 `nil` ，下标脚本的调用失败了。

```
if let firstRoomName = john.residence?[0].name {
    println("The first room name is \(firstRoomName).")
} else {
    println("Unable to retrieve the first room name.")
}
// 打印 "Unable to retrieve the first room name."。
```

在下标脚本调用中可选链的问号直接跟在 `john.residence` 的后面，在下标脚本括号的前面，因为 `john.residence` 是可选链试图获得的可选值。

如果你创建一个 `Residence` 实例给 `john.residence` ，且在他的 `rooms` 数组中有一个或多个 `Room` 实例，那么你可以使用可选链通过 `Residence` 下标脚本来获取在 `rooms` 数组中的实例了：

```
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.")
}
// 打印 "The first room name is Living Room."。
```

连接多层链接

你可以将多层可选链连接在一起，可以掘取模型内更下层的属性方法和下标脚本。然而多层可选链不能再添加比已经返回的可选值更多的层。 也就是说：

如果你试图获得的类型不是可选类型，由于使用了可选链它将变成可选类型。 如果你试图获得的类型已经是可选类型，由于可选链它也不会提高可选性。

因此：

如果你试图通过可选链获得 Int 值，不论使用了多少层链接返回的总是 Int? 。相似的，如果你试图通过可选链获得 Int? 值，不论使用了多少层链接返回的总是 Int? 。

下面的例子试图获取 john 的 residence 属性里的 address 的 street 属性。这里使用了两层可选链来联系 residence 和 address 属性，它们两者都是可选类型：

```
if let johnsStreet = john.residence?.address?.street {
    println("John's street name is \(johnsStreet).")
} else {
    println("Unable to retrieve the address.")
}
// 打印 "Unable to retrieve the address.”。
```

john.residence 的值现在包含一个 Residence 实例，然而 john.residence.address 现在是 nil ，因此 john.residence?.address?.street 调用失败。

从上面的例子发现，你试图获得 street 属性值。这个属性的类型是 String? 。因此尽管在可选类型属性前使用了两层可选链， john.residence?.address?.street 的返回值类型也是 String? 。

如果你为 Address 设定一个实例来作为 john.residence.address 的值，并为 address 的 street 属性设定一个实际值，你可以通过多层可选链来得到这个属性值。

```
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.")
}
// 打印 "John's street name is Laurel Street."。
```

值得注意的是，“ ！”符号在给 john.residence.address 分配 address 实例时的使用。 john.residence 属性是一个可选类型，因此你需要在它获取 address 属性之前使用 ！解析以获得它的实际值。

链接可选返回值的方法

前面的例子解释了如何通过可选链来获得可选类型属性值。你也可以通过可选链调用一个返回可选类型值的方法并按需链接该方法的返回值。

下面的例子通过可选链调用了 Address 类中的 buildingIdentifier 方法。这个方法的返回值类型是 String? 。如上所述，这个方法在可选链调用后最终的返回值类型依然是 String? ：

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    println("John's building identifier is \(buildingIdentifier).")
}
// 打印 "John's building identifier is The Larches."。
```

如果你还想进一步对方法返回值执行可选链，将可选链问号符放在方法括号的后面：

```
if let upper = john.residence?.address?.buildingIdentifier()?.uppercaseString {
    println("John's uppercase building identifier is \(upper).")
}
// 打印 "John's uppercase building identifier is THE LARCHES."。
```

> 注意：
>
> 在上面的例子中，你将可选链问号符放在括号后面是因为你想要链接的可选值是 `buildingIdentifier` 方法的返回值，不是 `buildingIdentifier` 方法本身。

18

类型转换（Type Casting）

类型转换可以判断实例的类型，也可以将实例看做是其父类或者子类的实例。

类型转换在 Swift 中使用 `is` 和 `as` 操作符实现。这两个操作符提供了一种简单达意的方式去检查值的类型或者转换它的类型。

你也可以用来检查一个类是否实现了某个协议，就像在 Checking for Protocol Conformance () 部分讲述的一样。

定义一个类层次作为例子

你可以将它用在类和子类的层次结构上，检查特定类实例的类型并且转换这个类实例的类型成为这个层次结构中的其他类型。这下面的三个代码段定义了一个类层次和一个包含了几个这些类实例的数组，作为类型转换的例子。

第一个代码片段定义了一个新的基础类 `MediaItem` 。这个类为任何出现在数字媒体库的媒体项提供基础功能。特别的，它声明了一个 `String` 类型的 `name` 属性，和一个 `init name` 初始化器。（它假定所有的媒体项都有个名称。）

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

下一个代码段定义了 `MediaItem` 的两个子类。第一个子类 `Movie` ，在父类（或者说基类）的基础上增加了一个 `director` （导演）属性，和相应的初始化器。第二个类在父类的基础上增加了一个 `artist` （艺术家）属性，和相应的初始化器：

```
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)
    }
}
```

最后一个代码段创建了一个数组常量 `library` ，包含两个 `Movie` 实例和三个 `Song` 实例。 `library` 的类型是在它被初始化时根据它数组中所包含的内容推断来的。Swift 的类型检测器能够演绎出 `Movie` 和 `Song` 有共同的父类 `MediaItem` ，所以它推断出 `MediaItem[]` 类作为 `library` 的类型。

```
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[]
```

在幕后 library 里存储的媒体项依然是 Movie 和 Song 类型的，但是，若你迭代它，取出的实例会是 MediaItem 类型的，而不是 Movie 和 Song 类型的。为了让它们作为它们本来的类型工作，你需要检查它们的类型或者向下转换它们的类型到其它类型，就像下面描述的一样。

检查类型（Checking Type）

用类型检查操作符(is)来检查一个实例是否属于特定子类型。若实例属于那个子类型，类型检查操作符返回 true ，否则返回 false 。

下面的例子定义了两个变量， movieCount 和 songCount ，用来计算数组 library 中 Movie 和 Song 类型的实例数量。

```
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"
```

示例迭代了数组 library 中的所有项。每一次， for - in 循环设置 item 为数组中的下一个 MediaItem 。

若当前 MediaItem 是一个 Movie 类型的实例， item is Movie 返回 true ，相反返回 false 。同样的， item is Song 检查item是否为 Song 类型的实例。在循环结束后， movieCount 和 songCount 的值就是被找到属于各自的类型的实例数量。

向下转型（Downcasting）

某类型的一个常量或变量可能在幕后实际上属于一个子类。你可以相信，上面就是这种情况。你可以尝试向下转到它的子类型，用类型转换操作符(as)

因为向下转型可能会失败，类型转型操作符带有两种不同形式。可选形式（ optional form） as? 返回一个你试图下转成的类型的可选值（optional value）。强制形式 as 把试图向下转型和强制解包（force-unwraps）结果作为一个混合动作。

当你不确定向下转型可以成功时，用类型转换的可选形式(as?)。可选形式的类型转换总是返回一个可选值（optional value），并且若下转是不可能的，可选值将是 nil 。这使你能够检查向下转型是否成功。

只有你可以确定向下转型一定会成功时，才使用强制形式。当你试图向下转型为一个不正确的类型时，强制形式的类型转换会触发一个运行时错误。

下面的例子，迭代了 library 里的每一个 MediaItem ，并打印出适当的描述。要这样做， item 需要真正作为 Movie 或 Song 的类型来使用。不仅仅是作为 MediaItem 。为了能够使用 Movie 或 Song 的 director 或 artist 属性，这是必要的。

在这个示例中，数组中的每一个 item 可能是 Movie 或 Song 。 事前你不知道每个 item 的真实类型，所以这里使用可选形式的类型转换（ as? ）去检查循环里的每次下转。

```
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
```

示例首先试图将 item 下转为 Movie 。因为 item 是一个 MediaItem 类型的实例，它可能是一个 Movie ；同样，它可能是一个 Song ，或者仅仅是基类 MediaItem 。因为不确定， as? 形式在试图下转时将返还一个可选值。 item as Movie 的返回值是 Movie? 类型或“optional Movie ”。

当向下转型为 Movie 应用在两个 Song 实例时将会失败。为了处理这种情况，上面的例子使用了可选绑定（optional binding）来检查可选 Movie 真的包含一个值（这个是为了判断下转是否成功。）可选绑定是这样写的“ if let movie = item as? Movie ”，可以这样解读：

“尝试将 item 转为 Movie 类型。若成功，设置一个新的临时常量 movie 来存储返回的可选 Movie ”

若向下转型成功，然后 movie 的属性将用于打印一个 Movie 实例的描述，包括它的导演的名字 director 。当 Song 被找到时，一个相近的原理被用来检测 Song 实例和打印它的描述。

> 注意： 转换没有真的改变实例或它的值。潜在的根本的实例保持不变；只是简单地把它作为它被转换成的类来使用。

Any 和 AnyObject 的类型转换

Swift为不确定类型提供了两种特殊类型别名：

- AnyObject 可以代表任何class类型的实例。
- Any 可以表示任何类型，除了方法类型（function types）。

> 注意： 只有当你明确的需要它的行为和功能时才使用 Any 和 AnyObject 。在你的代码里使用你期望的明确的类型总是更好的。

AnyObject 类型

当需要在工作中使用 Cocoa APIs，它一般接收一个 AnyObject[] 类型的数组，或者说“一个任何对象类型的数组”。这是因为 Objective-C 没有明确的类型化数组。但是，你常常可以确定包含在仅从你知道的 API 信息提供的这样一个数组中的对象的类型。

在这些情况下，你可以使用强制形式的类型转换(as)来下转在数组中的每一项到比 AnyObject 更明确的类型，不需要可选解析（optional unwrapping）。

下面的示例定义了一个 AnyObject[] 类型的数组并填入三个 Movie 类型的实例：

```
let someObjects: AnyObject[] = [
    Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),
    Movie(name: "Moon", director: "Duncan Jones"),
    Movie(name: "Alien", director: "Ridley Scott")
]
```

因为知道这个数组只包含 Movie 实例，你可以直接用(as)下转并解包到不可选的 Movie 类型（ps：其实就是我们常用的正常类型，这里是为了和可选类型相对比）。

```
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
```

为了变为一个更短的形式，下转 someObjects 数组为 Movie[] 类型来代替下转每一项方式。

```
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 类型

这里有个示例，使用 Any 类型来和混合的不同类型一起工作，包括非 class 类型。它创建了一个可以存储 Any 类型的数组 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.append({ (name: String) -> String in "Hello, \(name)" })
```

things 数组包含两个 Int 值，2个 Double 值，1个 String 值，一个元组 (Double, Double) ，Ivan Reitman 导演的电影“Ghostbusters”。

你可以在 switch cases 里用 is 和 as 操作符来发觉只知道是 Any 或 AnyObject 的常量或变量的类型。下面的示例迭代 things 数组中的每一项的并用 switch 语句查找每一项的类型。这几种 switch 语句的情形绑定它们匹配的值到一个规定类型的常量，让它们可以打印它们的值：

```
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)")
    case let stringConverter as String -> String:
        println(stringConverter("Michael"))
    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
// Hello, Michael
```

> 注意： 在一个switch语句的case中使用强制形式的类型转换操作符（as, 而不是 as?）来检查和转换到一个明确的类型。在 switch case 语句的内容中这种检查总是安全的。

19

嵌套类型

枚举类型常被用于实现特定类或结构体的功能。也能够在有多种变量类型的环境中，方便地定义通用类或结构体来使用，为了实现这种功能，Swift允许你定义嵌套类型，可以在枚举类型、类和结构体中定义支持嵌套的类型。

要在一个类型中嵌套另一个类型，将需要嵌套的类型的定义写在被嵌套类型的区域{}内，而且可以根据需要定义多级嵌套。

嵌套类型实例

下面这个例子定义了一个结构体 BlackjackCard (二十一点)，用来模拟 BlackjackCard 中的扑克牌点数。 BlackjackCard 结构体包含2个嵌套定义的枚举类型 Suit 和 Rank 。

在 BlackjackCard 规则中， Ace 牌可以表示1或者11， Ace 牌的这一特征用一个嵌套在枚举型 Rank 的结构体 Values 来表示。

```
struct BlackjackCard {
    // 嵌套定义枚举型Suit
    enum Suit: Character {
        case Spades = "♠", Hearts = "♡", Diamonds = "♢", Clubs = "♣"
    }
    // 嵌套定义枚举型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 的属性和方法
    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
    }
}
```

枚举型的 Suit 用来描述扑克牌的四种花色，并分别用一个 Character 类型的值代表花色符号。

枚举型的 Rank 用来描述扑克牌从 Ace ~10, J , Q , K ,13张牌，并分别用一个 Int 类型的值表示牌的面值。(这个 Int 类型的值不适用于 Ace , J , Q , K 的牌)。

如上文所提到的，枚举型 Rank 在自己内部定义了一个嵌套结构体 Values 。这个结构体包含两个变量，只有 Ace 有两个数值，其余牌都只有一个数值。结构体 Values 中定义的两个属性：

first , 为 Int second , 为 Int? , 或“optional Int ”

Rank 定义了一个计算属性 values ，这个计算属性会根据牌的面值，用适当的数值去初始化 Values 实例，并赋值给 values 。对于 J , Q , K , Ace 会使用特殊数值，对于数字面值的牌使用 Int 类型的值。

BlackjackCard 结构体自身有两个属性— rank 与 suit ，也同样定义了一个计算属性 description ， description 属性用 rank 和 suit 的中内容来构建对这张扑克牌名字和数值的描述，并用可选类型 second 来检查是否存在第二个值，若存在，则在原有的描述中增加对第二数值的描述。

因为 BlackjackCard 是一个没有自定义构造函数的结构体，在Memberwise Initializers for Structure Types () 中知道结构体有默认的成员构造函数，所以你可以用默认的 initializer 去初始化新的常量 theAceOfSpades :

```
let theAceOfSpades = BlackjackCard(rank: .Ace, suit: .Spades)
println("theAceOfSpades: \(theAceOfSpades.description)")
// 打印出 "theAceOfSpades: suit is ♠, value is 1 or 11"
```

尽管 Rank 和 Suit 嵌套在 BlackjackCard 中，但仍可被引用，所以在初始化实例时能够通过枚举类型中的成员名称单独引用。在上面的例子中 description 属性能正确得输出对 Ace 牌有1和11两个值。

嵌套类型的引用

在外部对嵌套类型的引用，以被嵌套类型的名字为前缀，加上所要引用的属性名：

```
let heartsSymbol = BlackjackCard.Suit.Hearts.toRaw()
// 红心的符号 为 "♡"
```

对于上面这个例子，这样可以使 Suit , Rank , 和 Values 的名字尽可能的短，因为它们的名字会自然的由被定义的上下文来限定。

20

扩展（Extensions）

扩展就是向一个已有的类、结构体或枚举类型添加新功能（functionality）。这包括在没有权限获取原始源代码的情况下扩展类型的能力（即*逆向建模*）。扩展和 Objective-C 中的分类（categories）类似。（不过与Objective-C不同的是，Swift 的扩展没有名字。）

Swift 中的扩展可以：

- 添加计算型属性和计算静态属性
- 定义实例方法和类型方法
- 提供新的构造器
- 定义下标
- 定义和使用新的嵌套类型
- 使一个已有类型符合某个协议

> 注意：
> 如果你定义了一个扩展向一个已有类型添加新功能，那么这个新功能对该类型的所有已有实例中都是可用的，即使它们是在你的这个扩展的前面定义的。

扩展语法（Extension Syntax）

声明一个扩展使用关键字 `extension`：

```
extension SomeType {
    // 加到SomeType的新功能写到这里
}
```

一个扩展可以扩展一个已有类型，使其能够适配一个或多个协议（protocol）。当这种情况发生时，协议的名字应该完全按照类或结构体的名字的方式进行书写：

```
extension SomeType: SomeProtocol, AnotherProctocol {
    // 协议实现写到这里
}
```

按照这种方式添加的协议遵循者（protocol conformance）被称之为在扩展中添加协议遵循者 ()

计算型属性（Computed Properties）

扩展可以向已有类型添加计算型实例属性和计算型类型属性。下面的例子向 Swift 的内建 `Double` 类型添加了5个计算型实例属性，从而提供与距离单位协作的基本支持。

```
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")
// 打印输出："One inch is 0.0254 meters"
let threeFeet = 3.ft
println("Three feet is \(threeFeet) meters")
// 打印输出："Three feet is 0.914399970739201 meters"
```

这些计算属性表达的含义是把一个 Double 型的值看作是某单位下的长度值。即使它们被实现为计算型属性，但这些属性仍可以接一个带有dot语法的浮点型字面值，而这恰恰是使用这些浮点型字面量实现距离转换的方式。

在上述例子中，一个 Double 型的值 1.0 被用来表示“1米”。这就是为什么 m 计算型属性返回 self ——表达式 1.m 被认为是计算 1.0 的 Double 值。

其它单位则需要一些转换来表示在米下测量的值。1千米等于1,000米，所以 km 计算型属性要把值乘以 1_000.00 来转化成单位米下的数值。类似地，1米有3.28024英尺，所以 ft 计算型属性要把对应的 Double 值除以 3.28024 来实现英尺到米的单位换算。

这些属性是只读的计算型属性，所有从简考虑它们不用 get 关键字表示。它们的返回值是 Double 型，而且可以用于所有接受 Double 的数学计算中：

```
let aMarathon = 42.km + 195.m
println("A marathon is \(aMarathon) meters long")
// 打印输出："A marathon is 42195.0 meters long"
```

> 注意：
> 扩展可以添加新的计算属性，但是不可以添加存储属性，也不可以向已有属性添加属性观测器(property observers)。

构造器（Initializers）

扩展可以向已有类型添加新的构造器。这可以让你扩展其它类型，将你自己的定制类型作为构造器参数，或者提供该类型的原始实现中没有包含的额外初始化选项。

扩展能向类中添加新的便利构造器，但是它们不能向类中添加新的指定构造器或析构函数。指定构造器和析构函数必须总是由原始的类实现来提供。

> 注意：
> 如果你使用扩展向一个值类型添加一个构造器，在该值类型已经向所有的存储属性提供默认值，而且没有定义任

> 何定制构造器（custom initializers）时，你可以在值类型的扩展构造器中调用默认构造器(default initializers)和逐一成员构造器(memberwise initializers)。
>
> 正如在值类型的构造器代理 () 中描述的，如果你已经把构造器写成值类型原始实现的一部分，上述规则不再适用。

下面的例子定义了一个用于描述几何矩形的定制结构体 `Rect` 。这个例子同时定义了两个辅助结构体 `Size` 和 `Point` ，它们都把 `0.0` 作为所有属性的默认值：

```
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()
}
```

因为结构体 `Rect` 提供了其所有属性的默认值，所以正如默认构造器中描述的，它可以自动接受一个默认的构造器和一个成员级构造器。这些构造器可以用于构造新的 `Rect` 实例：

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
```

你可以提供一个额外的使用特殊中心点和大小的构造器来扩展 `Rect` 结构体：

```
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)
    }
}
```

这个新的构造器首先根据提供的 `center` 和 `size` 值计算一个合适的原点。然后调用该结构体自动的成员构造器 `init(origin:size:)` ，该构造器将新的原点和大小存到了合适的属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect的原点是 (2.5, 2.5)，大小是 (3.0, 3.0)
```

> 注意：
>
> 如果你使用扩展提供了一个新的构造器，你依旧有责任保证构造过程能够让所有实例完全初始化。

方法（Methods）

扩展可以向已有类型添加新的实例方法和类型方法。下面的例子向 Int 类型添加一个名为 repetitions 的新实例方法：

```
extension Int {
    func repetitions(task: () -> ()) {
        for i in 0..<self {
            task()
        }
    }
}
```

这个 repetitions 方法使用了一个 () -> () 类型的单参数（single argument），表明函数没有参数而且没有返回值。

定义该扩展之后，你就可以对任意整数调用 repetitions 方法,实现的功能则是多次执行某任务：

```
3.repetitions({
    println("Hello!")
    })
// Hello!
// Hello!
// Hello!
```

可以使用 trailing 闭包使调用更加简洁：

```
3.repetitions{
    println("Goodbye!")
}
// Goodbye!
// Goodbye!
// Goodbye!
```

修改实例方法（Mutating Instance Methods）

通过扩展添加的实例方法也可以修改该实例本身。结构体和枚举类型中修改 self 或其属性的方法必须将该实例方法标注为 mutating，正如来自原始实现的修改方法一样。

下面的例子向Swift的 Int 类型添加了一个新的名为 square 的修改方法，来实现一个原始值的平方计算：

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt 现在值是 9
```

下标（Subscripts）

扩展可以向一个已有类型添加新下标。这个例子向Swift内建类型 `Int` 添加了一个整型下标。该下标 `[n]` 返回十进制数字从右向左数的第n个数字

- 123456789[0]返回9
- 123456789[1]返回8

...等等

```
extension Int {
    subscript(var digitIndex: Int) -> Int {
        var decimalBase = 1
            while digitIndex > 0 {
                decimalBase *= 10
                --digitIndex
            }
            return (self / decimalBase) % 10
    }
}
746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

如果该 `Int` 值没有足够的位数，即下标越界，那么上述实现的下标会返回0，因为它会在数字左边自动补0：

```
746381295[9]
//returns 0, 即等同于：
0746381295[9]
```

嵌套类型（Nested Types）

扩展可以向已有的类、结构体和枚举添加新的嵌套类型：

```
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
        }
    }
}
```

该例子向 Character 添加了新的嵌套枚举。这个名为 Kind 的枚举表示特定字符的类型。具体来说，就是表示一个标准的拉丁脚本中的字符是元音还是辅音（不考虑口语和地方变种），或者是其它类型。

这个例子还向 Character 添加了一个新的计算实例属性，即 kind ，用来返回合适的 Kind 枚举成员。

现在，这个嵌套枚举可以和一个 Character 值联合使用了：

```
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
```

函数 printLetterKinds 的输入是一个 String 值并对其字符进行迭代。在每次迭代过程中，考虑当前字符的 kind 计算属性，并打印出合适的类别描述。所以 printLetterKinds 就可以用来打印一个完整单词中所有字母的类型，正如上述单词 "hello" 所展示的。

> 注意：
> 由于已知 character.kind 是 Character.Kind 型，所以 Character.Kind 中的所有成员值都可以使用 switch 语句里的形式简写，比如使用 .Vowel 代替 Character.Kind.Vowel

21

协议

协议(Protocol) 用于定义完成某项任务或功能所必须的方法和属性，协议实际上并不提供这些功能或任务的具体实现(Implementation) ——而只用来描述这些实现应该是什么样的。类，结构体，枚举通过提供协议所要求的方法，属性的具体实现来 采用(adopt) 协议。任意能够满足协议要求的类型被称为协议的 遵循者 。

协议 可以要求其 遵循者 提供特定的实例属性，实例方法，类方法，操作符或下标脚本等。

协议的语法

协议 的定义方式与 类，结构体，枚举 的定义都非常相似，如下所示:

```
protocol SomeProtocol {
    // 协议内容
}
```

在类型名称后加上 协议名称 ，中间以冒号 : 分隔即可实现协议；实现多个协议时，各协议之间用逗号 , 分隔，如下所示:

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // 结构体内容
}
```

如果一个类在含有 父类 的同时也采用了协议，应当把 父类 放在所有的 协议 之前，如下所示:

```
class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {
    // 类的内容
}
```

对属性的规定

协议可以规定其 遵循者 提供特定名称与类型的 实例属性(instance property) 或 类属性(type property) ，而不管其是 存储型属性(stored property) 还是 计算型属性(calculate property) 。此外也可以指定属性是只读的还是可读写的。

如果协议要求属性是可读写的，那么这个属性不能是常量 存储型属性 或只读 计算型属性 ；如果协议要求属性是只读的(gettable)，那么 计算型属性 或 存储型属性 都能满足协议对属性的规定，在你的代码中，即使为只读属性实现了写方法(settable)也依然有效。

协议中的属性经常被加以 var 前缀声明其为变量属性，在声明后加上 { set get } 来表示属性是可读写的，只读的属性则写作 { get } ，如下所示:

```
protocol SomeProtocol {
    var mustBeSettable : Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

如下所示，通常在协议的定义中使用 class 前缀表示该属性为类成员；在枚举和结构体实现协议时中，需要使用 static 关键字作为前缀。

```
protocol AnotherProtocol {
    class var someTypeProperty: Int { get set }
}
```

如下所示，这是一个含有一个实例属性要求的协议:

```
protocol FullyNamed {
    var fullName: String { get }
}
```

FullyNamed 协议定义了任何拥有 fullName 的类型。它并不指定具体类型，而只是要求类型必须提供一个 fullName 。任何 FullyNamed 类型都得有一个只读的 fullName 属性，类型为 String 。

如下所示，这是一个实现了 FullyNamed 协议的简单结构体:

```
struct Person: FullyNamed{
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
//john.fullName 为 "John Appleseed"
```

这个例子中定义了一个叫做 Person 的结构体，用来表示具有指定名字的人。从第一行代码中可以看出，它采用了 FullyNamed 协议。

Person 结构体的每一个实例都有一个叫做 fullName ， String 类型的存储型属性，这正好匹配了 FullyNamed 协议的要求，也就意味着， Person 结构体完整的 遵循 了协议。(如果协议要求未被完全满足,在编译时会报错)

这有一个更为复杂的类，它采用并实现了 FullyNamed 协议，如下所示:

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil ) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
    return (prefix != nil ? prefix! + " " : " ") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName == "USS Enterprise"
```

Starship 类把 fullName 属性实现为只读的 计算型属性 。每一个 Starship 类的实例都有一个名为 name 的必备属性和一个名为 prefix 的可选属性。 当 prefix 存在时，将 prefix 插入到 name 之前来为 Starship 构建 fullName ， prefix 不存在时，则将直接用 name 构建 fullName

对方法的规定

协议 可以要求其 遵循者 实现某些指定的 实例方法 或 类方法 。这些方法作为协议的一部分，像普通的方法一样清晰的放在协议的定义中，而不需要大括号和方法体。

> 注意： 协议中的方法支持 变长参数(variadic parameter) ，不支持 参数默认值(default value) 。

如下所示，协议中类方法的定义与类属性的定义相似，在协议定义的方法前置 class 关键字来表示。当在 枚举 或 结构体 实现类方法时，需要使用 static 关键字来代替。

```
protocol SomeProtocol {
    class func someTypeMethod()
}
```

如下所示，定义了含有一个实例方法的的协议。

```
protocol RandomNumberGenerator {
    func random() -> Double
}
```

RandomNumberGenerator 协议要求其 遵循者 必须拥有一个名为 random ， 返回值类型为 Double 的实例方法。(尽管这里并未指明，但是我们假设返回值在[0，1]区间内)。

RandomNumberGenerator 协议并不在意每一个随机数是怎样生成的，它只强调这里有一个随机数生成器。

如下所示，下边的是一个遵循了 RandomNumberGenerator 协议的类。该类实现了一个叫做*线性同余生成器(linear congruential generator)*的伪随机数算法。

```
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())")
// 输出 : "Here's a random number: 0.37464991998171"
println("And another one: \(generator.random())")
// 输出 : "And another one: 0.729023776863283"
```

对突变方法的规定

有时不得不在方法中更改实例的所属类型。在基于 值类型(value types) (结构体，枚举)的实例方法中，将 mutating 关键字作为函数的前缀，写在 func 之前，表示可以在该方法中修改实例及其属性的所属类型。这一过程在Modifyting Value Types from Within Instance Methods () 章节中有详细描述。

如果协议中的实例方法打算改变其 遵循者 实例的类型，那么在协议定义时需要在方法前加 mutating 关键字，才能使 结构体，枚举 来采用并满足协议中对方法的规定。

> 注意: 用 类 实现协议中的 mutating 方法时，不用写 mutating 关键字;用 结构体 ， 枚举 实现协议中的 mutating 方法时，必须写 mutating 关键字。

如下所示， Togglable 协议含有名为 toggle 的突变实例方法。根据名称推测， toggle 方法应该是用于切换或恢复其 遵循者 实例或其属性的类型。

```
protocol Togglable {
    mutating func toggle()
}
```

当使用 枚举 或 结构体 来实现 Togglabl 协议时，需要提供一个带有 mutating 前缀的 toggle 方法。

如下所示， OnOffSwitch 枚举 遵循 了 Togglable 协议， On ， Off 两个成员用于表示当前状态。枚举的 toggle 方法被标记为 mutating ，用以匹配 Togglabel 协议的规定。

```
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 现在的值为 .On
```

对构造器的规定

协议可以要求它的遵循类型实现特定的构造器。你可以像书写普通的构造器那样，在协议的定义里写下构造器的需求，但不需要写花括号和构造器的实体：

```
protocol SomeProtocol {
    init(someParameter: Int)
}
```

协议构造器规定在类中的实现

你可以在遵循该协议的类中实现构造器，并指定其为类的特定构造器或者便捷构造器。在这两种情况下，你都必须给构造器实现标上"required"修饰符：

```
class SomeClass: SomeProtocol {
    required init(someParameter: Int) {
        //构造器实现
    }
}
```

使用 required 修饰符可以保证：所有的遵循该协议的子类，同样能为构造器规定提供一个显式的实现或继承实现。

关于 required 构造器的更多内容，请参考The Basics ()

> 注意
>
> 如果类已经被“final”修饰符所标示，你就不需要在协议构造器规定的实现中使用"required"修饰符。因为final类不能有子类。关于 final 修饰符的更多内容，请参见initialization ()

如果一个子类重写了父类的指定构造器，并且该构造器遵循了某个协议的规定，那么该构造器的实现需要被同时标示 required 和 override 修饰符

```
protocol SomeProtocol {
    init()
}
class SomeSuperClass {
    init() {
        //协议定义
    }
}
class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from SomeSuperClass
    required override init() {
        // 构造器实现
    }
}
```

可失败构造器的规定

可以通过给协议 Protocols 中添加可失败构造器来使遵循该协议的类型必须实现该可失败构造器。

如果在协议中定义一个可失败构造器，则在遵顼该协议的类型中必须添加同名同参数的可失败构造器或非可失败构造器。 如果在协议中定义一个非可失败构造器，则在遵循该协议的类型中必须添加同名同参数的非可失败构造器或隐式解析类型的可失败构造器（ init! ）。

协议类型

尽管 协议 本身并不实现任何功能，但是 协议 可以被当做类型来使用。

使用场景:

- 协议类型 作为函数、方法或构造器中的参数类型或返回值类型
- 协议类型 作为常量、变量或属性的类型
- 协议类型 作为数组、字典或其他容器中的元素类型

> 注意: 协议是一种类型，因此协议类型的名称应与其他类型(Int，Double，String)的写法相同，使用驼峰式写法

如下所示，这个示例中将协议当做类型来使用

```
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
    }
}
```

例子中又一个 Dice 类，用来代表桌游中的拥有N个面的骰子。 Dice 的实例含有 sides 和 generator 两个属性，前者是整型，用来表示骰子有几个面，后者为骰子提供一个随机数生成器。

generator 属性的类型为 RandomNumberGenerator ，因此任何遵循了 RandomNumberGenerator 协议的类型的实例都可以赋值给 generator ，除此之外，无其他要求。

Dice 类中也有一个 构造器(initializer) ，用来进行初始化操作。构造器中含有一个名为 generator ，类型为 RandomNumberGenerator 的形参。在调用构造方法时创建 Dice 的实例时，可以传入任何遵循 RandomNumberGenerator 协议的实例给generator。

Dice 类也提供了一个名为 roll 的实例方法用来模拟骰子的面值。它先使用 generator 的 random 方法来创建一个[0-1]区间内的随机数种子，然后加工这个随机数种子生成骰子的面值。generator被认为是遵循了 RandomNumberGenerator 的类型，因而保证了 random 方法可以被调用。

如下所示，这里展示了如何使用 LinearCongruentialGenerator 的实例作为随机数生成器创建一个六面骰子:

```
var d6 = Dice(sides: 6,generator: LinearCongruentialGenerator())
for _ in 1...5 {
    println("Random dice roll is \(d6.roll())")
```

```
}
//输出结果
//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

委托是一种设计模式(*译者注: 想起了那年 UITableViewDelegate 中的奔跑，那是我逝去的Objective-C。。。*)，它允许 类 或 结构体 将一些需要它们负责的功能 交由(委托) 给其他的类型的实例。

委托模式的实现很简单: 定义 协议 来 封装 那些需要被委托的 函数和方法 ， 使其 遵循者 拥有这些被委托的 函数和方法 。

委托模式可以用来响应特定的动作或接收外部数据源提供的数据，而无需要知道外部数据源的所属类型(*译者注:只要求外部数据源 遵循 某协议*)。

下文是两个基于骰子游戏的协议:

```
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 协议可以在任意含有骰子的游戏中实现， DiceGameDelegate 协议可以用来追踪 DiceGame 的游戏过程

如下所示， SnakesAndLadders 是 Snakes and Ladders (译者注:Control Flow () 章节有该游戏的详细介绍)游戏的新版本。新版本使用 Dice 作为骰子，并且实现了 DiceGame 和 DiceGameDelegate 协议，后者用来记录游戏的过程:

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = 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; board[09] = +09; board[10] = +02
        board[14] = −10; board[19] = −11; board[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)
    }
}
```

这个版本的游戏封装到了 SnakesAndLadders 类中，该类采用了 DiceGame 协议，并且提供了 dice 属性和 play 实例方法用来 遵循 协议。(dice 属性在构造之后就不在改变，且协议只要求 dice 为只读的，因此将 dice 声明为常量属性。)

在 SnakesAndLadders 类的 构造器(initializer) 初始化游戏。所有的游戏逻辑被转移到了 play 方法中， play 方法使用协议规定的 dice 属性提供骰子摇出的值。

> 注意: delegate 并不是游戏的必备条件，因此 delegate 被定义为遵循 DiceGameDelegate 协议的可选属性， delegate 使用 nil 作为初始值。

DicegameDelegate 协议提供了三个方法用来追踪游戏过程。被放置于游戏的逻辑中，即 play() 方法内。分别在游戏开始时，新一轮开始时，游戏结束时被调用。

因为 delegate 是一个遵循 DiceGameDelegate 的可选属性，因此在 play() 方法中使用了 可选链 来调用委托方法。 若 delegate 属性为 nil ， 则delegate所调用的方法失效。若 delegate 不为 nil ，则方法能够被调用

如下所示， DiceGameTracker 遵循了 DiceGameDelegate 协议

```
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 实现了 DiceGameDelegate 协议规定的三个方法，用来记录游戏已经进行的轮数。 当游戏开始时， numberOfTurns 属性被赋值为0; 在每新一轮中递加; 游戏结束后，输出打印游戏的总轮数。

gameDidStart 方法从 game 参数获取游戏信息并输出。 game 在方法中被当做 DiceGame 类型而不是 SnakeAndLadders 类型，所以方法中只能访问 DiceGame 协议中的成员。当然了，这些方法也可以在类型转换之后调用。在上例代码中，通过 is 操作符检查 game 是否为 SnakesAndLadders 类型的实例，如果是，则打印出相应的内容。

无论当前进行的是何种游戏， game 都遵循 DiceGame 协议以确保 game 含有 dice 属性，因此在 gameDidStart 方法中可以通过传入的 game 参数来访问 dice 属性，进而打印出 dice 的 sides 属性的值。

DiceGameTracker 的运行情况，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
```

在扩展中添加协议成员

即便无法修改源代码，依然可以通过 扩展(Extension) 来扩充已存在类型(*译者注: 类，结构体，枚举等*)。 扩展 可以为已存在的类型添加 属性 ， 方法 ， 下标脚本 ， 协议 等成员。详情请在扩展 () 章节中查看。

> 注意: 通过 扩展 为已存在的类型 遵循 协议时，该类型的所有实例也会随之添加协议中的方法

TextRepresentable 协议含有一个 asText ，如下所示:

```
protocol TextRepresentable {
    func asText() -> String
}
```

通过 扩展 为上一节中提到的 Dice 类遵循 TextRepresentable 协议

```
extension Dice: TextRepresentable {
    func asText() -> String {
        return "A \(sides)-sided dice"
    }
}
```

从现在起， Dice 类型的实例可被当作 TextRepresentable 类型:

```
let d12 = Dice(sides: 12,generator: LinearCongruentialGenerator())
println(d12.asText())
// 输出 "A 12-sided dice"
```

SnakesAndLadders 类也可以通过 扩展 的方式来遵循协议：

```
extension SnakesAndLadders: TextRepresentable {
    func asText() -> String {
        return "A game of Snakes and Ladders with \(finalSquare) squares"
    }
}
println(game.asText())
// 输出 "A game of Snakes and Ladders with 25 squares"
```

通过扩展补充协议声明

当一个类型已经实现了协议中的所有要求，却没有声明时，可以通过 扩展 来补充协议声明:

```
struct Hamster {
    var name: String
    func asText() -> String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

从现在起， Hamster 的实例可以作为 TextRepresentable 类型使用

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
println(somethingTextRepresentable.asText())
// 输出 "A hamster named Simon"
```

> 注意: 即使满足了协议的所有要求，类型也不会自动转变，因此你必须为它做出明显的协议声明

集合中的协议类型

协议类型可以被集合使用，表示集合中的元素均为协议类型:

```
let things: [TextRepresentable] = [game,d12,simonTheHamster]
```

如下所示， things 数组可以被直接遍历，并调用其中元素的 asText() 函数:

```
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 被当做是 TextRepresentable 类型而不是 Dice ， DiceGame ， Hamster 等类型。因此能且仅能调用 asText 方法

协议的继承

协议能够继承一到多个其他协议。语法与类的继承相似，多个协议间用逗号 ， 分隔

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // 协议定义
}
```

如下所示， PrettyTextRepresentable 协议继承了 TextRepresentable 协议

```
protocol PrettyTextRepresentable: TextRepresentable {
    func asPrettyText() -> String
}
```

遵循 PrettyTextRepresentable 协议的同时，也需要遵循 TextRepresentable 协议。

如下所示，用 扩展 为 SnakesAndLadders 遵循 PrettyTextRepresentable 协议:

```
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 数组中的每一个元素:

- 当从数组中迭代出的元素的值大于0时，用 ▲ 表示
- 当从数组中迭代出的元素的值小于0时，用 ▼ 表示
- 当从数组中迭代出的元素的值等于0时，用 ○ 表示

任意 SankesAndLadders 的实例都可以使用 asPrettyText() 方法。

```
println(game.asPrettyText())
// A game of Snakes and Ladders with 25 squares:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

类专属协议

你可以在协议的继承列表中,通过添加“class”关键字,限制协议只能适配到类（class）类型。（结构体或枚举不能遵循该协议）。该“class”关键字必须是第一个出现在协议的继承列表中，其后，才是其他继承协议。

```
protocol SomeClassOnlyProtocol: class, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

在以上例子中，协议SomeClassOnlyProtocol只能被类（class）类型适配。如果尝试让结构体或枚举类型适配该协议，则会出现编译错误。

> 注意
>
> 当协议需求定义的行为，要求（或假设）它的遵循类型必须是引用语义而非值语义时，应该采用类专属协议。关于引用语义，值语义的更多内容，请查看结构体和枚举是值类型 (https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html#//apple_ref/doc/uid/TP40014097-CH13-XID_145) 和类是引用类型 (https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html#//apple_ref/doc/uid/TP40014097-CH13-XID_146)

协议合成

一个协议可由多个协议采用 `protocol<SomeProtocol， AnotherProtocol>` 这样的格式进行组合，称为 `协议合成(protocol composition)` 。

举个例子：

```
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)
// 输出 "Happy birthday Malcolm - you're 21!
```

Named 协议包含 String 类型的 name 属性; Aged 协议包含 Int 类型的 age 属性。 Person 结构体 遵循 了这两个协议。

wishHappyBirthday 函数的形参 celebrator 的类型为 protocol<Named，Aged> 。可以传入任意 遵循 这两个协议的类型的实例

> 注意: 协议合成 并不会生成一个新协议类型，而是将多个协议合成为一个临时的协议，超出范围后立即失效。

检验协议的一致性

使用 is 和 as 操作符来检查协议的一致性或转化协议类型。检查和转化的语法和之前相同(*详情查看Typy Casting章节* ()):

- is 操作符用来检查实例是否 遵循 了某个 协议 。
- as? 返回一个可选值，当实例 遵循 协议时，返回该协议类型;否则返回 nil
- as 用以强制向下转型。

```
@objc protocol HasArea {
    var area: Double { get }
}
```

> 注意: @objc 用来表示协议是可选的，也可以用来表示暴露给 Objective-C 的代码，此外， @objc 型协议只对 类 有效，因此只能在 类 中检查协议的一致性。

如下所示，定义了 Circle 和 Country 类，它们都遵循了 HasArea 协议

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

Circle 类把 area 实现为基于 存储型属性 radius的 计算型属性 ， Country 类则把 area 实现为 存储型属性 。这两个类都 遵循 了 HasArea 协议。

如下所示，Animal是一个没有实现 HasArea 协议的类

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

Circle，Country，Animal 并没有一个相同的基类，因而采用 AnyObject 类型的数组来装载在他们的实例，如下所示:

```
let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]
```

objects 数组使用字面量初始化，数组包含一个 radius 为2。0的 Circle 的实例，一个保存了英国面积的 Country 实例和一个 legs 为4的 Animal 实例。

如下所示， objects 数组可以被迭代，对迭代出的每一个元素进行检查，看它是否遵循了 HasArea 协议:

```
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
```

当迭代出的元素遵循 HasArea 协议时，通过 as? 操作符将其 可选绑定(optional binding) 到 objectWithArea 常量上。 objectWithArea 是 HasArea 协议类型的实例，因此 area 属性是可以被访问和打印的。

objects 数组中元素的类型并不会因为 向下转型 而改变，它们仍然是 Circle ， Country ， Animal 类型。然而，当它们被赋值给 objectWithArea 常量时，则只被视为 HasArea 类型，因此只有 area 属性能够被访问。

对可选协议的规定

可选协议含有可选成员，其 遵循者 可以选择是否实现这些成员。在协议中使用 @optional 关键字作为前缀来定义可选成员。

可选协议在调用时使用 可选链 ，详细内容在Optional Chaining () 章节中查看。

像 someOptionalMethod?(someArgument) 这样，你可以在可选方法名称后加上 ？ 来检查该方法是否被实现。 可选方法 和 可选属性 都会返回一个 可选值(optional value) ，当其不可访问时， ？ 之后语句不会执行，并整体返回 nil

> 注意: 可选协议只能在含有 @objc 前缀的协议中生效。且 @objc 的协议只能被 类 遵循

如下所示， Counter 类使用含有两个可选成员的 CounterDataSource 协议类型的外部数据源来提供 增量值(increment amount)

```
@objc protocol CounterDataSource {
    optional func incrementForCount(count: Int) -> Int
    optional var fixedIncrement: Int { get }
}
```

CounterDataSource 含有 incrementForCount 的 可选方法 和 fiexdIncrement 的 可选属性 ，它们使用了不同的方法来从数据源中获取合适的增量值。

> 注意: CounterDataSource 中的属性和方法都是可选的，因此可以在类中声明但不实现这些成员，尽管技术上允许这样做，不过最好不要这样写。

Counter 类含有 CounterDataSource? 类型的可选属性 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 属性用于存储当前的值， increment 方法用来为 count 赋值。

increment 方法通过 可选链 ，尝试从两种 可选成员 中获取 count 。

1. 由于 dataSource 可能为 nil ，因此在 dataSource 后边加上了 ？标记来表明只在 dataSource 非空时才去调用 incrementForCount 方法。
2. 即使 dataSource 存在，但是也无法保证其是否实现了 incrementForCount 方法，因此在 incrementForCount 方法后边也加有 ？标记

在调用 incrementForCount 方法后， Int 型 可选值 通过 可选绑定(optional binding) 自动拆包并赋值给常量 amount 。

当 incrementForCount 不能被调用时，尝试使用可选属性 fixedIncrement 来代替。

ThreeSource 实现了 CounterDataSource 协议，如下所示:

```
class ThreeSource: CounterDataSource {
    let fixedIncrement = 3
}
```

使用 ThreeSource 作为数据源开实例化一个 Counter :

```
var counter = Counter()
counter.dataSource = ThreeSource()
```

```
for _ in 1...4 {
    counter.increment()
    println(counter.count)
}
// 3
// 6
// 9
// 12
```

TowardsZeroSource 实现了 CounterDataSource 协议中的 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
        }
    }
}
```

下边是执行的代码:

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    println(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

22

泛型

泛型代码可以让你写出根据自我需求定义、适用于任何类型的，灵活且可重用的函数和类型。它的可以让你避免重复的代码，用一种清晰和抽象的方式来表达代码的意图。

泛型是 Swift 强大特征中的其中一个，许多 Swift 标准库是通过泛型代码构建出来的。事实上，泛型的使用贯穿了整本语言手册，只是你没有发现而已。例如，Swift 的数组和字典类型都是泛型集。你可以创建一个 Int 数组，也可创建一个 String 数组，或者甚至于可以是任何其他 Swift 的类型数据数组。同样的，你也可以创建存储任何指定类型的字典（dictionary），而且这些类型可以是没有限制的。

泛型所解决的问题

这里是一个标准的，非泛型函数 swapTwoInts ,用来交换两个Int值：

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

这个函数使用写入读出（in-out）参数来交换 a 和 b 的值，请参考写入读出参数 () 。

swapTwoInts 函数可以交换 b 的原始值到 a ，也可以交换a的原始值到 b ，你可以调用这个函数交换两个 Int 变量值：

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// 输出 "someInt is now 107, and anotherInt is now 3"
```

swapTwoInts 函数是非常有用的，但是它只能交换 Int 值，如果你想要交换两个 String 或者 Double ，就不得不写更多的函数，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 函数功能都是相同的，唯一不同之处就在于传入的变量类型不同，分别是 Int 、 String 和 Double 。

但实际应用中通常需要一个用处更强大并且尽可能的考虑到更多的灵活性单个函数，可以用来交换两个任何类型值，很幸运的是，泛型代码帮你解决了这种问题。（一个这种泛型函数后面已经定义好了。）

> 注意：
> 在所有三个函数中， a 和 b 的类型是一样的。如果 a 和 b 不是相同的类型，那它们俩就不能互换值。Swift 是类型安全的语言，所以它不允许一个 String 类型的变量和一个 Double 类型的变量互相交换值。如果一定要做，Swift 将报编译错误。

泛型函数

泛型函数 可以工作于任何类型，这里是一个上面 swapTwoInts 函数的泛型版本，用于交换两个值：

```
func swapTwoValues<T>(inout a: T, inout b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

swapTwoValues 函数主体和 swapTwoInts 函数是一样的，它只在第一行稍微有那么一点点不同于 swapTwoInts ，如下所示：

```
func swapTwoInts(inout a: Int, inout b: Int)
func swapTwoValues<T>(inout a: T, inout b: T)
```

这个函数的泛型版本使用了占位类型名字（通常此情况下用字母 T 来表示）来代替实际类型名（如 Int 、 String 或 Double ）。占位类型名没有提示 T 必须是什么类型，但是它提示了 a 和 b 必须是同一类型 T ，而不管 T 表示什么类型。只有 swapTwoValues 函数在每次调用时所传入的实际类型才能决定 T 所代表的类型。

另外一个不同之处在于这个泛型函数名后面跟着的占位类型名字（T）是用尖括号括起来的（ <T> ）。这个尖括号告诉 Swift 那个 T 是 swapTwoValues 函数所定义的一个类型。因为 T 是一个占位命名类型，Swift 不会去查找命名为T的实际类型。

swapTwoValues 函数除了要求传入的两个任何类型值是同一类型外，也可以作为 swapTwoInts 函数被调用。每次 swapTwoValues 被调用，T所代表的类型值都会传给函数。

在下面的两个例子中, T 分别代表 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
> 上面定义的函数 swapTwoValues 是受 swap 函数启发而实现的。 swap 函数存在于 Swift 标准库，并可以在

> 其它类中任意使用。如果你在自己代码中需要类似 swapTwoValues 函数的功能，你可以使用已存在的交换函数 swap 函数。

类型参数

在上面的 swapTwoValues 例子中，占位类型 T 是一种类型参数的示例。类型参数指定并命名为一个占位类型，并且紧随在函数名后面，使用一对尖括号括起来（如 <T> ）。

一旦一个类型参数被指定，那么其可以被使用来定义一个函数的参数类型（如 swapTwoValues 函数中的参数 a 和 b ），或作为一个函数返回类型，或用作函数主体中的注释类型。在这种情况下，被类型参数所代表的占位类型不管函数任何时候被调用，都会被实际类型所替换（在上面 swapTwoValues 例子中，当函数第一次被调用时， T 被 Int 替换，第二次调用时，被 String 替换。）。

你可支持多个类型参数，命名在尖括号中，用逗号分开。

命名类型参数

在简单的情况下，泛型函数或泛型类型需要指定一个占位类型（如上面的 swapTwoValues 泛型函数，或一个存储单一类型的泛型集，如数组），通常用一单个字母 T 来命名类型参数。不过，你可以使用任何有效的标识符来作为类型参数名。

如果你使用多个参数定义更复杂的泛型函数或泛型类型，那么使用更多的描述类型参数是非常有用的。例如，Swift 字典（Dictionary）类型有两个类型参数，一个是键，另外一个是值。如果你自己写字典，你或许会定义这两个类型参数为 KeyType 和 ValueType ，用来记住它们在你的泛型代码中的作用。

> 注意
> 请始终使用大写字母开头的驼峰式命名法（例如 T 和 KeyType ）来给类型参数命名，以表明它们是类型的占位符，而非类型值。

泛型类型

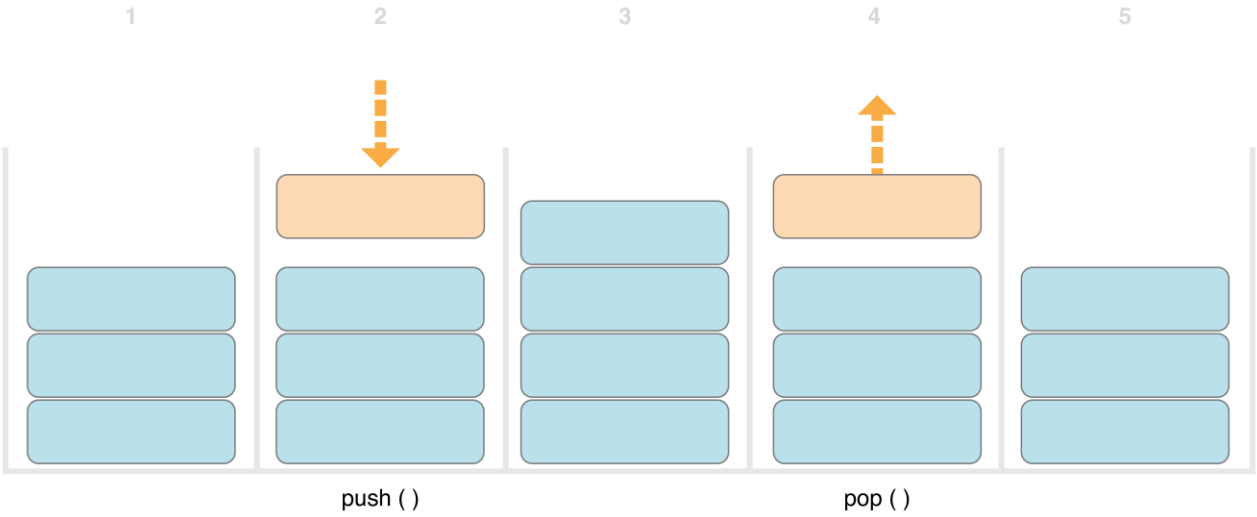
通常在泛型函数中，Swift 允许你定义你自己的泛型类型。这些自定义类、结构体和枚举作用于任何类型，如同 Array 和 Dictionary 的用法。

这部分向你展示如何写一个泛型集类型--- Stack （栈）。一个栈是一系列值域的集合，和 Array （数组）类似，但其是一个比 Swift 的 Array 类型更多限制的集合。一个数组可以允许其里面任何位置的插入/删除操作，而

栈，只允许在集合的末端添加新的项（如同*push*一个新值进栈）。同样的一个栈也只能从末端移除项（如同*pop*一个值出栈）。

> 注意
>
> 栈的概念已被 `UINavigationController` 类使用来模拟试图控制器的导航结构。你通过调用 `UINavigationController` 的 `pushViewController:animated:` 方法来为导航栈添加（add）新的试图控制器；而通过 `popViewControllerAnimated:` 的方法来从导航栈中移除（pop）某个试图控制器。每当你需要一个严格的 `后进先出` 方式来管理集合，堆栈都是最实用的模型。

下图展示了一个栈的压栈(push)/出栈(pop)的行为：



图片 22.1 Image of Generics_1.png

1. 现在有三个值在栈中；
2. 第四个值“pushed”到栈的顶部；
3. 现在有四个值在栈中，最近的那个在顶部；
4. 栈中最顶部的那个项被移除，或称之为“popped”；
5. 移除掉一个值后，现在栈又重新只有三个值。

这里展示了如何写一个非泛型版本的栈， `Int` 值型的栈：

```
struct IntStack {
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

这个结构体在栈中使用一个 Array 性质的 items 存储值。 Stack 提供两个方法： push 和 pop ，从栈中压进一个值和移除一个值。这些方法标记为可变的，因为它们需要修改（或*转换*）结构体的 items 数组。

上面所展现的 IntStack 类型只能用于 Int 值，不过，其对于定义一个泛型 Stack 类（可以处理*任何*类型值的栈）是非常有用的。

这里是一个相同代码的泛型版本：

```
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类型参数为T代替了实际 Int 类型。这种类型参数包含在一对尖括号里（ <T> ），紧随在结构体名字后面。

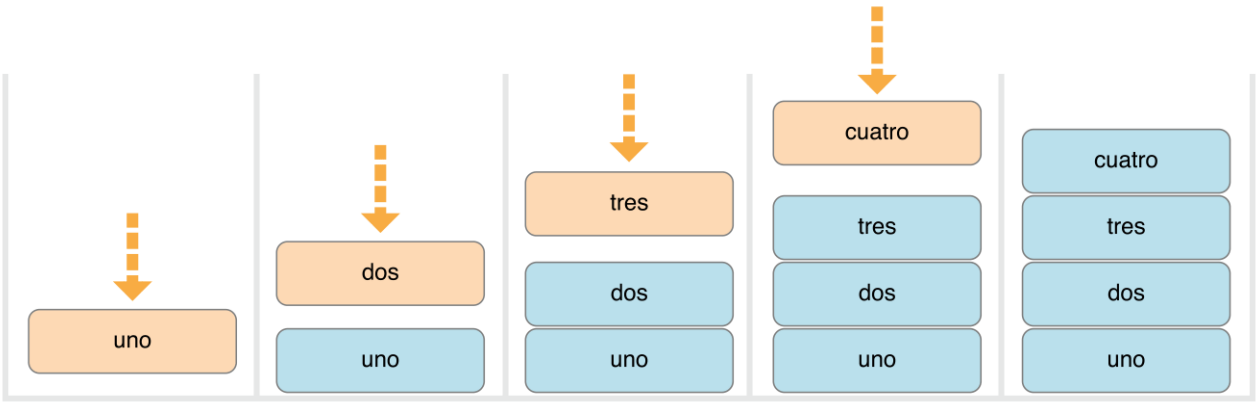
T 定义了一个名为“某种类型T”的节点提供给后来用。这种将来类型可以在结构体的定义里任何地方表示为“T”。在这种情况下， T 在如下三个地方被用作节点：

- 创建一个名为 items 的属性，使用空的T类型值数组对其进行初始化；
- 指定一个包含一个参数名为 item 的 push 方法，该参数必须是T类型；
- 指定一个 pop 方法的返回值，该返回值将是一个T类型值。

当创建一个新单例并初始化时， 通过用一对紧随在类型名后的尖括号里写出实际指定栈用到类型，创建一个 Stack 实例，同创建 Array 和 Dictionary 一样：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// 现在栈已经有4个string了
```

下图将展示 stackOfStrings 如何 push 这四个值进栈的过程：

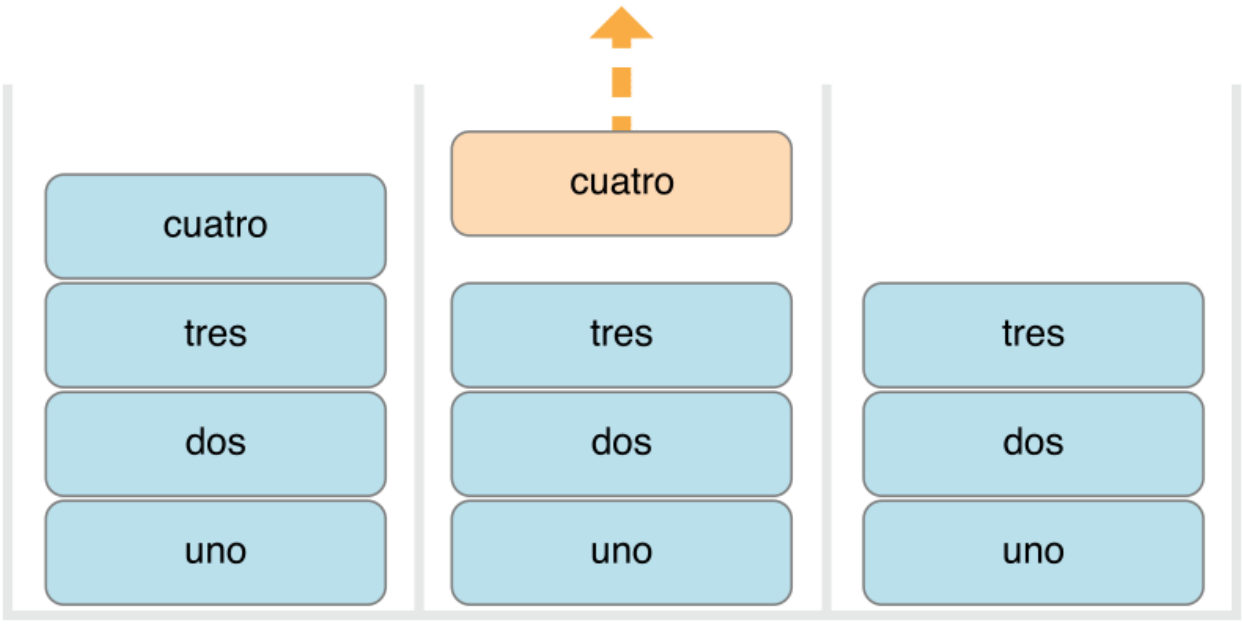


图片 22.2 Image of Generics_2.png

从栈中 pop 并移除值"cuatro"：

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

下图展示了如何从栈中pop一个值的过程：



图片 22.3 Image of Generics_3.png

由于 Stack 是泛型类型，所以在 Swift 中其可以用来创建任何有效类型的栈，这种方式如同 Array 和 Dictionary 。

()

类型约束

`swapTwoValues` 函数和 `Stack` 类型可以作用于任何类型，不过，有的时候对使用在泛型函数和泛型类型上的类型强制约束为某种特定类型是非常有用的。类型约束指定了一个必须继承自指定类的类型参数，或者遵循一个特定的协议或协议构成。

例如，Swift 的 `Dictionary` 类型对作用于其键的类型做了些限制。在字典 () 的描述中，字典的键类型必须是*可哈希*，也就是说，必须有一种方法可以使其被唯一的表示。`Dictionary` 之所以需要其键是可哈希是为了以便于其检查其是否已经包含某个特定键的值。如无此需求，`Dictionary` 既不会告诉是否插入或者替换了某个特定键的值，也不能查找到已经存储在字典里面的给定键值。

这个需求强制加上一个类型约束作用于 `Dictionary` 的键上，当然其键类型必须遵循 `Hashable` 协议（Swift 标准库中定义的一个特定协议）。所有的 Swift 基本类型（如 `String`，`Int`，`Double` 和 `Bool`）默认都是可哈希。

当你创建自定义泛型类型时，你可以定义你自己的类型约束，当然，这些约束要支持泛型编程的强力特征中的多数。抽象概念如 `可哈希` 具有的类型特征是根据它们概念特征来界定的，而不是它们的直接类型特征。

类型约束语法

你可以写一个在一个类型参数名后面的类型约束，通过冒号分割，来作为类型参数链的一部分。这种作用于泛型函数的类型约束的基础语法如下所示（和泛型类型的语法相同）：

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

上面这个假定函数有两个类型参数。第一个类型参数 `T`，有一个需要 `T` 必须是 `SomeClass` 子类的类型约束；第二个类型参数 `U`，有一个需要 `U` 必须遵循 `SomeProtocol` 协议的类型约束。

类型约束行为

这里有个名为 `findStringIndex` 的非泛型函数，该函数功能是去查找包含一给定 `String` 值的数组。若查找到匹配的字符串，`findStringIndex` 函数返回该字符串在数组中的索引值（`Int`），反之则返回 `nil`：

```
func findStringIndex(array: [String], valueToFind: String) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
```

```
    return nil
}
```

findStringIndex 函数可以作用于查找一字符串数组中的某个字符串:

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findStringIndex(strings, "llama") {
    println("The index of llama is \(foundIndex)")
}
// 输出 "The index of llama is 2"
```

如果只是针对字符串而言查找在数组中的某个值的索引，用处不是很大，不过，你可以写出相同功能的泛型函数 findIndex ，用某个类型 T 值替换掉提到的字符串。

这里展示如何写一个你或许期望的 findStringIndex 的泛型版本 findIndex 。请注意这个函数仍然返回 Int ，是不是有点迷惑呢，而不是泛型类型?那是因为函数返回的是一个可选的索引数，而不是从数组中得到的一个可选值。需要提醒的是，这个函数不会编译，原因在例子后面会说明:

```
func findIndex<T>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

上面所写的函数不会编译。这个问题的位置在等式的检查上， “if value == valueToFind” 。不是所有的 Swift 中的类型都可以用等式符（ == ）进行比较。例如，如果你创建一个你自己的类或结构体来表示一个复杂的数据模型，那么 Swift 没法猜到对于这个类或结构体而言“等于”的意思。正因如此，这部分代码不能可能保证工作于每个可能的类型 T ，当你试图编译这部分代码时估计会出现相应的错误。

不过，所有的这些并不会让我们无从下手。Swift 标准库中定义了一个 Equatable 协议，该协议要求任何遵循的类型实现等式符（ == ）和不等符（ != ）对任何两个该类型进行比较。所有的 Swift 标准类型自动支持 Equatable 协议。

任何 Equatable 类型都可以安全的使用在 findIndex 函数中，因为其保证支持等式操作。为了说明这个事实，当你定义一个函数时，你可以写一个 Equatable 类型约束作为类型参数定义的一部分:

```
func findIndex<T: Equatable>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

findIndex 中这个单个类型参数写做： T: Equatable ，也就意味着“任何T类型都遵循 Equatable 协议”。

findIndex 函数现在则可以成功的编译过，并且作用于任何遵循 Equatable 的类型，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
```

关联类型(Associated Types)

当定义一个协议时，有的时候声明一个或多个关联类型作为协议定义的一部分是非常有用的。一个关联类型作为协议的一部分，给定了类型的一个占位名（或别名）。作用于关联类型上实际类型在协议被实现前是不需要指定的。关联类型被指定为 typealias 关键字。

关联类型行为

这里是一个 Container 协议的例子，定义了一个ItemType关联类型：

```
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
```

Container 协议定义了三个任何容器必须支持的兼容要求：

- 必须可能通过 append 方法添加一个新item到容器里；
- 必须可能通过使用 count 属性获取容器里items的数量，并返回一个 Int 值；
- 必须可能通过容器的 Int 索引值下标可以检索到每一个item。

这个协议没有指定容器里item是如何存储的或何种类型是允许的。这个协议只指定三个任何遵循 Container 类型所必须支持的功能点。一个遵循的类型在满足这三个条件的情况下也可以提供其他额外的功能。

任何遵循 Container 协议的类型必须指定存储在其里面的值类型，必须保证只有正确类型的items可以加进容器里，必须明确可以通过其下标返回item类型。

为了定义这三个条件， Container 协议需要一个方法指定容器里的元素将会保留，而不需要知道特定容器的类型。 Container 协议需要指定任何通过 append 方法添加到容器里的值和容器里元素是相同类型，并且通过容器下标返回的容器元素类型的值的类型是相同类型。

为了达到此目的， Container 协议声明了一个ItemType的关联类型，写作 typealias ItemType 。这个协议不会定义 ItemType 是什么的别名，这个信息将由任何遵循协议的类型来提供。尽管如此， ItemType 别名提供了一种识别Container中Items类型的方法，并且用于 append 方法和 subscript 方法的类型定义，以便保证任何 Container 期望的行为能够被执行。

这里是一个早前IntStack类型的非泛型版本，遵循Container协议：

```
struct IntStack: Container {
    // IntStack的原始实现
    var items = [Int]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // 遵循Container协议的实现
    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` 类型实现了 `Container` 协议的所有三个要求，在 `IntStack` 类型的每个包含部分的功能都满足这些要求。

此外， `IntStack` 指定了 `Container` 的实现，适用的ItemType被用作 `Int` 类型。对于这个 `Container` 协议实现而言，定义 `typealias ItemType = Int` ，将抽象的 `ItemType` 类型转换为具体的 `Int` 类型。

感谢Swift类型参考，你不用在 `IntStack` 定义部分声明一个具体的 `Int` 的 `ItemType` 。由于 `IntStack` 遵循 `Container` 协议的所有要求，只要通过简单的查找 `append` 方法的item参数类型和下标返回的类型，Swift就可以推断出合适的 `ItemType` 来使用。确实，如果上面的代码中你删除了 `typealias ItemType = Int` 这一行，一切仍旧可以工作，因为它清楚的知道ItemType使用的是何种类型。

你也可以生成遵循 `Container` 协议的泛型 `Stack` 类型：

```
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]
```

```
    }
}
```

这个时候，占位类型参数 T 被用作 append 方法的item参数和下标的返回类型。Swift 因此可以推断出被用作这个特定容器的 ItemType 的 T 的合适类型。

扩展一个存在的类型为一指定关联类型

在使用扩展来添加协议兼容性 () 中有描述扩展一个存在的类型添加遵循一个协议。这个类型包含一个关联类型的协议。

Swift的 Array 已经提供 append 方法，一个 count 属性和通过下标来查找一个自己的元素。这三个功能都达到 Container 协议的要求。也就意味着你可以扩展 Array 去遵循 Container 协议，只要通过简单声明 Array 适用于该协议而已。如何实践这样一个空扩展，在使用扩展来声明协议的采纳 () 中有描述这样一个实现一个空扩展的行为：

```
extension Array: Container {}
```

如同上面的泛型 Stack 类型一样， Array的append 方法和下标保证 Swift 可以推断出 ItemType 所使用的适用的类型。定义了这个扩展后，你可以将任何 Array 当作 Container 来使用。

Where 语句

类型约束 (页 226)能够确保类型符合泛型函数或类的定义约束。

对关联类型定义约束是非常有用的。你可以在参数列表中通过*where*语句定义参数的约束。一个 where 语句能够使一个关联类型遵循一个特定的协议，以及（或）那个特定的类型参数和关联类型可以是相同的。你可以写一个 where 语句，紧跟在在类型参数列表后面，where语句后跟一个或者多个针对关联类型的约束，以及（或）一个或多个类型和关联类型间的等价(equality)关系。

下面的例子定义了一个名为 allItemsMatch 的泛型函数，用来检查两个 Container 实例是否包含相同顺序的相同元素。如果所有的元素能够匹配，那么返回一个为 true 的 Boolean 值，反之则为 false 。

被检查的两个 Container 可以不是相同类型的容器（虽然它们可以是），但它们确实拥有相同类型的元素。这个需求通过一个类型约束和 where 语句结合来表示：

```
func allItemsMatch<
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {

        // 检查两个Container的元素个数是否相同
        if someContainer.count != anotherContainer.count {
            return false
```

```
        }

    // 检查两个Container相应位置的元素彼此是否相等
    for i in 0..<someContainer.count {
        if someContainer[i] != anotherContainer[i] {
            return false
        }
    }

    // 如果所有元素检查都相同则返回true
    return true

}
```

这个函数用了两个参数： someContainer 和 anotherContainer 。 someContainer 参数是类型 C1 ， anotherContainer 参数是类型 C2 。 C1 和 C2 是容器的两个占位类型参数，决定了这个函数何时被调用。

这个函数的类型参数列紧随在两个类型参数需求的后面：

- C1 必须遵循 Container 协议 (写作 C1: Container)。
- C2 必须遵循 Container 协议 (写作 C2: Container)。
- C1 的 ItemType 同样是C2的 ItemType （写作 C1.ItemType == C2.ItemType ）。
- C1 的 ItemType 必须遵循 Equatable 协议 (写作 C1.ItemType: Equatable)。

第三个和第四个要求被定义为一个 where 语句的一部分，写在关键字 where 后面，作为函数类型参数链的一部分。

这些要求意思是：

someContainer 是一个 C1 类型的容器。 anotherContainer 是一个 C2 类型的容器。 someContainer 和 anotherContainer 包含相同的元素类型。 someContainer 中的元素可以通过不等于操作(!=)来检查它们是否彼此不同。

第三个和第四个要求结合起来的意思是 anotherContainer 中的元素也可以通过 != 操作来检查，因为它们在 someContainer 中元素确实是相同的类型。

这些要求能够使 allItemsMatch 函数比较两个容器，即便它们是不同的容器类型。

allItemsMatch 首先检查两个容器是否拥有同样数目的items，如果它们的元素数目不同，没有办法进行匹配，函数就会 false 。

检查完之后，函数通过 for-in 循环和半闭区间操作（ .. ）来迭代 someContainer 中的所有元素。对于每个元素，函数检查是否 someContainer 中的元素不等于对应的 anotherContainer 中的元素，如果这两个元素不等，则这两个容器不匹配，返回 false 。

如果循环体结束后未发现没有任何的不匹配，那表明两个容器匹配，函数返回 true 。

这里演示了allItemsMatch函数运算的过程：

```
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.")
}
// 输出 "All items match."
```

上面的例子创建一个 Stack 单例来存储 String ，然后压了三个字符串进栈。这个例子也创建了一个 Array 单例，并初始化包含三个同栈里一样的原始字符串。即便栈和数组是不同的类型，但它们都遵循 Container 协议，而且它们都包含同样的类型值。因此你可以调用 allItemsMatch 函数，用这两个容器作为它的参数。在上面的例子中， allItemsMatch 函数正确的显示了所有的这两个容器的 items 匹配。

23

访问控制

访问控制可以限定你在源文件或模块中访问代码的级别，也就是说可以控制哪些代码你可以访问，哪些代码你不能访问。这个特性可以让我们隐藏功能实现的一些细节，并且可以明确的指定我们提供给其他人的接口中哪些部分是他们可以使用的，哪些是他们看不到的。

你可以明确的给类、结构体、枚举、设置访问级别，也可以给属性、函数、初始化方法、基本类型、下标索引等设置访问级别。协议也可以被限定在一定的范围内使用，包括协议里的全局常量、变量和函数。

在提供了不同访问级别的同时，Swift 并没有规定我们要在任何时候都要在代码中明确指定访问级别。其实，如果我们作为独立开发者在开发我们自己的 app，而不是在开发一些 Framework 的时候，我们完全可以不用明确的指定代码的访问级别。

> 注意：为方便起见，在代码中可以设置访问级别的它们（属性、基本类型、函数等）在下面的章节中我们称之为“实体”。

模块和源文件

Swift 中的访问控制模型基于模块和源文件这两个概念。

模块指的是 Framework 或 App bundle 。在 Swift 中，可以用 import 关键字引入自己的工程。

在 Swift 中， Framework 或 App bundle 被作为模块处理。如果你是为了实现某个通用的功能，或者是为了封装一些常用方法而将代码打包成 Framework ，这个 Framework 在 Swift 中就被称为模块。不论它被引入到某个 App 工程或者其他的 Framework ，它里面的一切（属性、函数等）都属于这个模块。

源文件指的是 Swift 中的 Swift File ，就是编写 Swift 代码的文件，它通常属于一个模块。通常一个源文件包含一个 类 ，在 类 中又包含 函数 、 属性 等类型。

访问级别

Swift 提供了三种不同的访问级别。这些访问级别相对于源文件中定义的实体，同时也相对于这些源文件所属的模块。

- Public ：可以访问自己模块或应用中源文件里的任何实体，别人也可以访问引入该模块中源文件里的所有实体。通常情况下，某个接口或 Framework 是可以被任何人使用时，你可以将其设置为 public 级别。
- Internal ：可以访问自己模块或应用中源文件里的任何实体，但是别人不能访问该模块中源文件里的实体。通常情况下，某个接口或 Framework 作为内部结构使用时，你可以将其设置为 internal 级别。
- Private ：只能在当前源文件中使用的实体，称为私有实体。使用 private 级别，可以用作隐藏某些功能的实现细节。

Public 为最高级访问级别， Private 为最低级访问级别。

访问级别的使用原则

在 Swift 中，访问级别有如下使用原则：访问级别统一性。 比如说：

- 一个 public 访问级别的变量，不能将它的类型定义为 internal 和 private 的类型。因为变量可以被任何人访问，但是定义它的类型不可以，所以这样就会出现错误。
- 函数的访问级别不能高于它的参数、返回类型的访问级别。因为如果函数定义为 public 而参数或者返回类型定义为 internal 或 private ，就会出现函数可以被任何人访问，但是它的参数和返回类型不可以，同样会出现错误。

()

默认访问级别

代码中的所有实体，如果你不明确的定义其访问级别，那么它们默认为 internal 级别。在大多数情况下，我们不需要明确的设置实体的访问级别，因为我们大多数时候都是在开发一个 App bundle。

单目标应用程序的访问级别

当你编写一个单目标应用程序时，该应用的所有功能都是为该应用服务，不需要提供给其他应用或者模块使用，所以我们不需要明确设置访问级别，使用默认的访问级别 internal 即可。但是如果你愿意，你也可以使用 private 级别，用于隐藏一些功能的实现细节。

Framework的访问级别

当你开发 Framework 时，就需要把一些实体定义为 public 级别，以便其他人导入该 Framework 后可以正常使用其功能。这些被你定义为 public 的实体，就是这个 Framework 的API。

> 注意： Framework 的内部实现细节依然可以使用默认的 internal 级别，或者也可以定义为 private 级别。只有你想将它作为 API 的实体，才将其定义为 public 级别。

访问控制语法

通过修饰符 public 、 internal 、 private 来声明实体的访问级别：

```
public class SomePublicClass {}
internal class SomeInternalClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
private func somePrivateFunction() {}
```

除非有特殊的说明，否则实体都使用默认的访问级别 internal ，可以查阅［默认访问级别］（#default_access_levels）这一节。这意味着 SomeInternalClass 和 someInternalConstant 不用明确的使用修饰符声明访问级别，但是他们任然拥有隐式的访问级别 internal ：

```
class SomeInternalClass {}              // 隐式访问级别 internal
var someInternalConstant = 0            // 隐式访问级别 internal
```

()

自定义类型

如果你想为一个自定义类型指定一个明确的访问级别，那么你要明确一点。那就是你要确保新类型的访问级别和它实际的作用域相匹配。比如说，如果某个类里的属性、函数、返回值它们的作用域仅在当前的源文件中，那么你就可以将这个类申明为 private 类，而不需要申明为 public 或者 internal 类。

类的访问级别也可以影响到类成员（属性、函数、初始化方法等）的默认访问级别。如果你将类申明为 private 类，那么该类的所有成员的默认访问级别也会成为 private 。如果你将类申明为 public 或者 internal 类（或者不明确的指定访问级别，而使用默认的 internal 访问级别），那么该类的所有成员的访问级别是 internal 。

> 注意：上面提到，一个 public 类的所有成员的访问级别默认为 internal 级别，而不是 public 级别。如果你想将某个成员申明为 public 级别，那么你必须使用修饰符明确的申明该成员。这样做的好处是，在你定义公共接口API的时候，可以明确的选择哪些属性或方法是需要公开的，哪些是内部使用的，可以避免将内部使用的属性方法公开成公共API的错误。

```
public class SomePublicClass {          // 显示的 public 类
    public var somePublicProperty = 0    // 显示的 public 类成员
    var someInternalProperty = 0         // 隐式的 internal 类成员
    private func somePrivateMethod() {}  // 显示的 private 类成员
}

class SomeInternalClass {               // 隐式的 internal 类
    var someInternalProperty = 0         // 隐式的 internal 类成员
    private func somePrivateMethod() {}  // 显示的 private 类成员
}

private class SomePrivateClass {        // 显示的 private 类
    var somePrivateProperty = 0          // 隐式的 private 类成员
    func somePrivateMethod() {}          // 隐式的 private 类成员
}
```

元组类型

元组的访问级别使用是所有类型的访问级别使用中最为严谨的。比如说，如果你构建一个包含两种不同类型元素的元组，其中一个元素类型的访问级别为 internal ，另一个为 private 级别，那么这个元组的访问级别为 private 。也就是说元组的访问级别遵循它里面元组中最低级的访问级别。

> 注意：元组不同于类、结构体、枚举、函数那样有单独的定义。元组的访问级别是在它被使用时自动推导出的，而不是明确的申明。

函数类型

函数的访问级别需要根据该函数的参数类型访问级别、返回类型访问级别得出。如果根据参数类型和返回类型得出的函数访问级别不符合上下文，那么就需要明确的申明该函数的访问级别。

下面的例子中定义了一个全局函数名为 someFunction ，并且没有明确的申明其访问级别。你也许会认为该函数应该拥有默认的访问级别 internal ，但事实并非如此。事实上，如果按下面这种写法，编译器是无法编译通过的：

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

我们可以看到，这个函数的返回类型是一个元组，该元组中包含两个自定义的类（可查阅自定义类型 (页 237)）。其中一个类的访问级别是 internal ，另一个的访问级别是 private ，所以根据元组访问级别的原则，该元组的访问级别是 private （元组的访问级别遵循它里面元组中最低级的访问级别）。

因为该函数返回类型的访问级别是 private ，所以你必须使用 private 修饰符，明确的申请该函数：

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

将该函数申明为 public 或 internal ，或者使用默认的访问级别 internal 都是错误的，因为如果把该函数当做 public 或 internal 级别来使用的话，是无法得到 private 级别的返回值的。

枚举类型

枚举中成员的访问级别继承自该枚举，你不能为枚举中的成员指定访问级别。

比如下面的例子，枚举 CompassPoint 被明确的申明为 public 级别，那么它的成员 North ， South ， East ， West 的访问级别同样也是 public ：

```
public enum CompassPoint {
    case North
    case South
    case East
    case West
}
```

原始值和关联值

用于枚举定义中的任何原始值，或关联的值类型必须有一个访问级别，至少要高于枚举的访问级别。比如说，你不能在一个 internal 访问级别的枚举中定义 private 级别的原始值类型。

嵌套类型

如果在 private 级别的类型中定义嵌套类型，那么该嵌套类型就自动拥有 private 访问级别。如果在 public 或者 internal 级别的类型中定义嵌套类型，那么该嵌套类型自动拥有 internal 访问级别。如果想让嵌套类型拥有 public 访问级别，那么需要对该嵌套类型进行明确的访问级别申明。

子类

子类的访问级别不得高于父类的访问级别。比如说，父类的访问级别是 internal ，子类的访问级别就不能申明为 public 。

此外，在满足子类不高于父类访问级别以及遵循各访问级别作用域（即模块或源文件）的前提下，你可以重写任意类成员（方法、属性、初始化方法、下标索引等）。

如果我们无法直接访问某个类中的属性或函数等，那么可以继承该类，从而可以更容易的访问到该类的类成员。下面的例子中，类 A 的访问级别是 public ，它包含一个函数 someMethod ，访问级别为 private 。类 B 继承类 A ，并且访问级别申明为 internal ，但是在类 B 中重写了类 A 中访问级别为 private 的方法 someMethod ，并重新申明为 internal 级别。通过这种方式，我们就可以访问到某类中 private 级别的类成员，并且可以重新申明其访问级别，以便其他人使用：

```
public class A {
    private func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {}
}
```

只要满足子类不高于父类访问级别以及遵循各访问级别作用域的前提下（即 private 的作用域在同一个源文件中， internal 的作用域在同一个模块下），我们甚至可以在子类中，用子类成员访问父类成员，哪怕父类成员的访问级别比子类成员的要低：

```
public class A {
    private func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {
        super.someMethod()
    }
}
```

因为父类 A 和子类 B 定义在同一个源文件中，所以在类 B 中可以在重写的 someMethod 方法中调用 super.someMethod() 。

常量、变量、属性、下标

常量、变量、属性不能拥有比它们的类型更高的访问级别。比如说，你定义一个 public 级别的属性，但是它的类型是 private 级别的，这是编译器不允许的。同样，下标也不能拥有比索引类型或返回类型更高的访问级别。

如果常量、变量、属性、下标索引的定义类型是 private 级别的，那么它们必须要明确的申明访问级别为 private ：

```
private var privateInstance = SomePrivateClass()
```

Getter和Setter

常量、变量、属性、下标索引的 Getters 和 Setters 的访问级别继承自它们所属成员的访问级别。

Setter 的访问级别可以低于对应的 Getter 的访问级别，这样就可以控制变量、属性或下标索引的读写权限。在 var 或 subscript 定义作用域之前，你可以通过 private(set) 或 internal(set) 先为它门的写权限申明一个较低的访问级别。

> 注意：这个规定适用于用作存储的属性或用作计算的属性。即使你不明确的申明存储属性的 Getter 、 Setter ，Swift也会隐式的为其创建 Getter 和 Setter ，用于对该属性进行读取操作。使用 private(set) 和 internal(set) 可以改变Swift隐式创建的 Setter 的访问级别。在计算属性中也是同样的。

下面的例子中定义了一个结构体名为 TrackedString ，它记录了 value 属性被修改的次数：

```
struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
    didSet {
        numberOfEdits++
    }
    }
}
```

TrackedString 结构体定义了一个用于存储的属性名为 value ，类型为 String ，并将初始化值设为 "" （即一个空字符串）。该结构体同时也定义了另一个用于存储的属性名为 numberOfEdits ，类型为 Int ，它用于记录属性 value 被修改的次数。这个功能的实现通过属性 value 的 didSet 方法实现，每当给 value 赋新值时就会调用 didSet 方法，给 numberOfEdits 加一。

结构体 TrackedString 和它的属性 value 均没有明确的申明访问级别，所以它们都拥有默认的访问级别 internal 。但是该结构体的 numberOfEdits 属性使用 private(set) 修饰符进行申明，这意味着 numberOfEdits 属性只能在定义该结构体的源文件中赋值。 numberOfEdits 属性的 Getter 依然是默认的访问级别 internal ，但是 Setter 的访问级别是 private ，这表示该属性只有在当前的源文件中是可读可写的，在当前源文件所属的模块中它只是一个可读的属性。

如果你实例化 TrackedString 结构体，并且多次对 value 属性的值进行修改，你就会看到 numberOfEdits 的值会随着修改次数更改：

```
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment numberOfEdits."
stringToEdit.value += " So will this one."
println("The number of edits is \(stringToEdit.numberOfEdits)")
// prints "The number of edits is 3"
```

虽然你可以在其他的源文件中实例化该结构体并且获取到 numberOfEdits 属性的值，但是你不能对其进行赋值。这样就能很好的告诉使用者，你只管使用，而不需要知道其实现细节。

初始化

我们可以给自定义的初始化方法指定访问级别，但是必须要低于或等于它所属类的访问级别。但如果该初始化方法是必须要使用的话，那它的访问级别就必须和所属类的访问级别相同。

如同函数或方法参数，初始化方法参数的访问级别也不能低于初始化方法的访问级别。

<

默认初始化方法

Swift为结构体、类都提供了一个默认的无参初始化方法，用于给它们的所有属性提供赋值操作，但不会给出具体值。默认初始化方法可以参阅Default Initializers () 。默认初始化方法的访问级别与所属类型的访问级别相同。

> 注意：如果一个类型被申明为 public 级别，那么默认的初始化方法的访问级别为 internal 。如果你想让无参的初始化方法在其他模块中可以被使用，那么你必须提供一个具有 public 访问级别的无参初始化方法。

结构体的默认成员初始化方法

如果结构体中的任一存储属性的访问级别为 `private`，那么它的默认成员初始化方法访问级别就是 `private`。尽管如此，结构体的初始化方法的访问级别依然是 `internal`。

如果你想在其他模块中使用该结构体的默认成员初始化方法，那么你需要提供一个访问级别为 `public` 的默认成员初始化方法。

协议

如果你想为一个协议明确的申明访问级别，那么有一点需要注意，就是你要确保该协议只在你申明的访问级别作用域中使用。

协议中的每一个必须要实现的函数都具有和该协议相同的访问级别。这样才能确保该协议的使用者可以实现它所提供的函数。

> 注意：如果你定义了一个 `public` 访问级别的协议，那么实现该协议提供的必要函数也会是 `public` 的访问级别。这一点不同于其他类型，比如， `public` 访问级别的其他类型，他们成员的访问级别为 `internal`。

协议继承

如果定义了一个新的协议，并且该协议继承了一个已知的协议，那么新协议拥有的访问级别最高也只和被继承协议的访问级别相同。比如说，你不能定义一个 `public` 的协议而去继承一个 `internal` 的协议。

协议一致性

类可以采用比自身访问级别低的协议。比如说，你可以定义一个 `public` 级别的类，可以让它在其他模块中使用，同时它也可以采用一个 `internal` 级别的协议，并且只能在定义了该协议的模块中使用。

采用了协议的类的访问级别遵循它本身和采用协议中最低的访问级别。也就是说如果一个类是 `public` 级别，采用的协议是 `internal` 级别，那个采用了这个协议后，该类的访问级别也是 `internal`。

如果你采用了协议，那么实现了协议必须的方法后，该方法的访问级别遵循协议的访问级别。比如说，一个 `public` 级别的类，采用了 `internal` 级别的协议，那么该类实现协议的方法至少也得是 `internal`。

> 注意：在Swift中和Objective-C中一样，协议的一致性保证了一个类不可能在同一个程序中用不同的方法采用同一个协议。

扩展

你可以在条件允许的情况下对类、结构体、枚举进行扩展。扩展成员应该具有和原始类成员一致的访问级别。比如你扩展了一个公共类型，那么你新加的成员应该具有和原始成员一样的默认的 `internal` 访问级别。

或者，你可以明确申明扩展的访问级别（比如使用 `private extension` ）给该扩展内所有成员指定一个新的默认访问级别。这个新的默认访问级别仍然可以被单独成员所指定的访问级别所覆盖。

协议的扩展

如果一个扩展采用了某个协议，那么你就不能对该扩展使用访问级别修饰符来申明了。该扩展中实现协议的方法都会遵循该协议的访问级别。

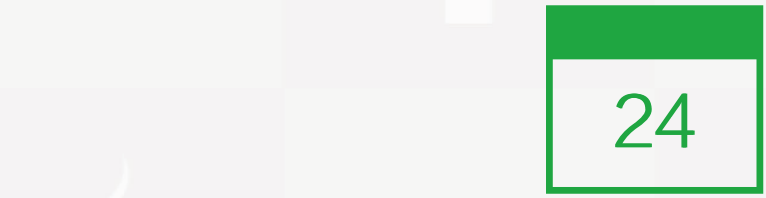
泛型

泛型类型或泛型函数的访问级别遵循泛型类型、函数本身、泛型类型参数三者中访问级别最低的级别。

类型别名

任何被你定义的类型别名都会被视作为不同的类型，这些类型用于访问控制。一个类型别名的访问级别可以低于或等于这个类型的访问级别。比如说，一个 `private` 级别的类型别名可以设定给一个 `public` 、 `internal` 、 `private` 的类型，但是一个 `public` 级别的类型别名只能设定给一个 `public` 级别的类型，不能设定给 `internal` 或 `private` 的类类型。

> 注意：这条规则也适用于为满足协议一致性而给相关类型命名别名。



24

高级运算符

除了基本操作符 () 中所讲的运算符，Swift还有许多复杂的高级运算符，包括了C语言和Objective-C中的位运算符和移位运算。

不同于C语言中的数值计算，Swift的数值计算默认是不可溢出的。溢出行为会被捕获并报告为错误。你是故意的？好吧，你可以使用Swift为你准备的另一套默认允许溢出的数值运算符，如可溢出的加号为 `&+` 。所有允许溢出的运算符都是以 `&` 开始的。

自定义的结构，类和枚举，是否可以使用标准的运算符来定义操作？当然可以！在Swift中，你可以为你创建的所有类型定制运算符的操作。

可定制的运算符并不限于那些预设的运算符，你可以自定义中置，前置，后置及赋值运算符，当然还有优先级和结合性。这些运算符在代码中可以像预设的运算符一样使用，你也可以扩展已有的类型以支持你自定义的运算符。

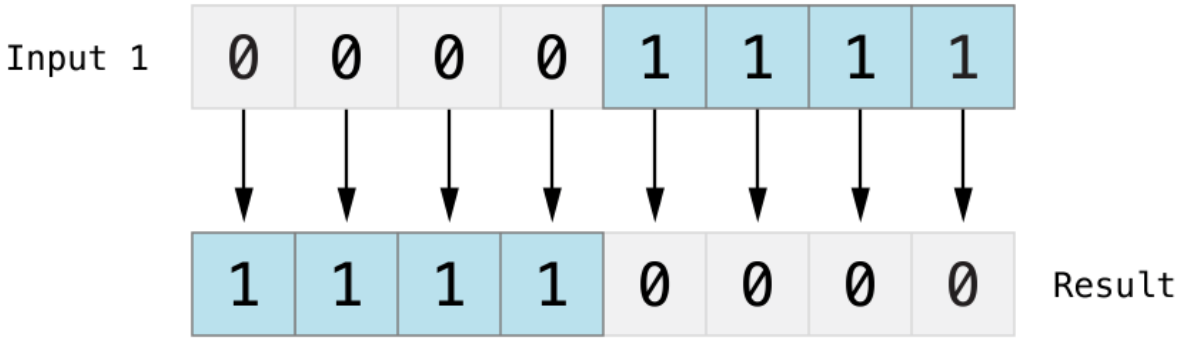
位运算符

位操作符可以操作数据结构中原始数据的每个比特位。位操作符通常在诸如图像处理和创建设备驱动等底层开发中使用，位操作符在同外部资源的数据进行交互的时候也很有用，比如在使用用户协议进行通信的时候，运用位运算符来对原始数据进行编码和解码。

Swift支持如下所有C语言的位运算符：

按位取反运算符

按位取反运算符 `~` 对一个操作数的每一位都取反。



图片 24.1 Image of Advanced_Operators_1.png

这个运算符是前置的，所以请不加任何空格地写在操作数之前。

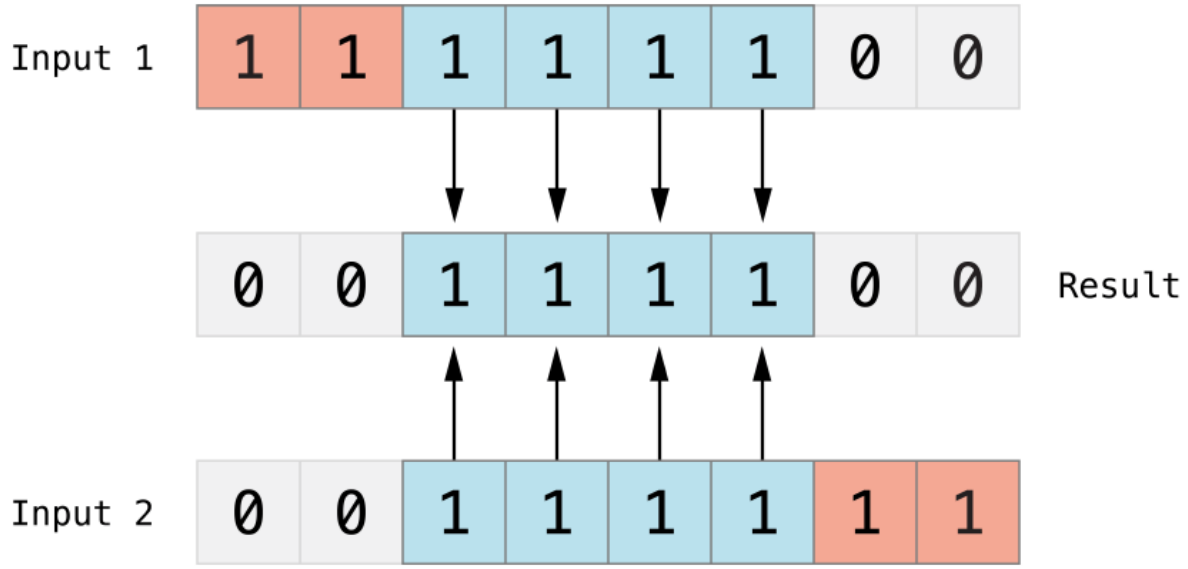
```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // 等于 0b11110000
```

`UInt8` 是8位无符整型，可以存储0~255之间的任意数。这个例子初始化一个整型为二进制值 `00001111` (前4位为 `0` ，后4位为 `1`)，它的十进制值为 `15` 。

使用按位取反运算 `~` 对 `initialBits` 操作，然后赋值给 `invertedBits` 这个新常量。这个新常量的值等于所有位都取反的 `initialBits` ，即 `1` 变成 `0` ， `0` 变成 `1` ，变成了 `11110000` ，十进制值为 `240` 。

按位与运算符

按位与运算符对两个数进行操作，然后返回一个新的数，这个数的每个位都需要两个输入数的同一位都为1时才为1。



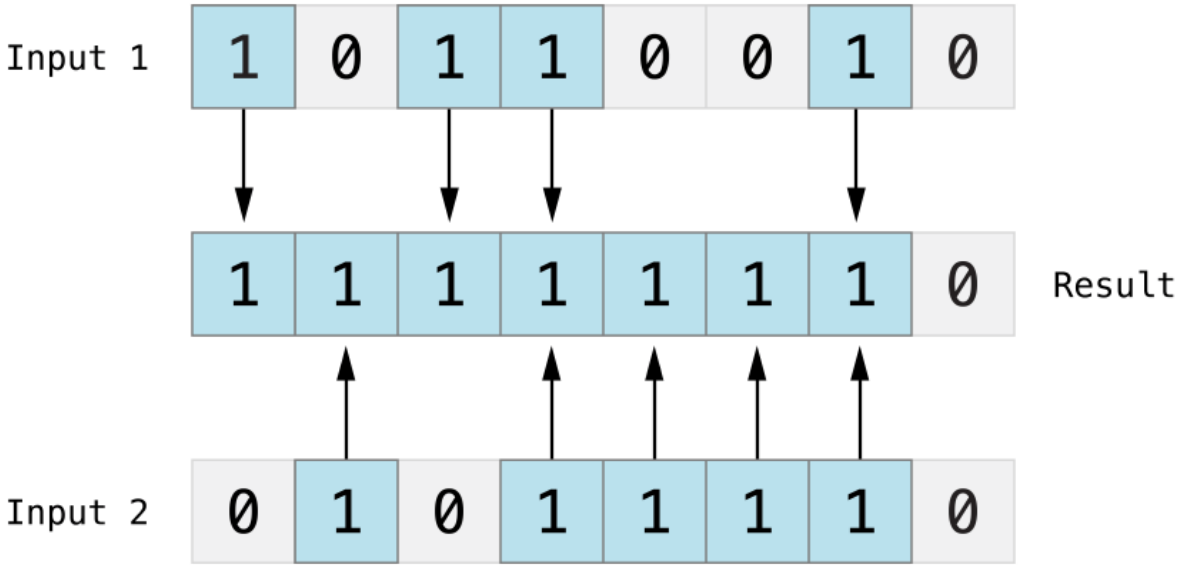
图片 24.2 Image of Advanced_Operators_2.png

以下代码， `firstSixBits` 和 `lastSixBits` 中间4个位都为1。对它俩进行按位与运算后，就得到了 `00111100` ，即十进制的 `60` 。

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // 等于 00111100
```

按位或运算

按位或运算符 `|` 比较两个数，然后返回一个新的数，这个数的每一位设置1的条件是两个输入数的同一位都不为0(即任意一个为1，或都为1)。



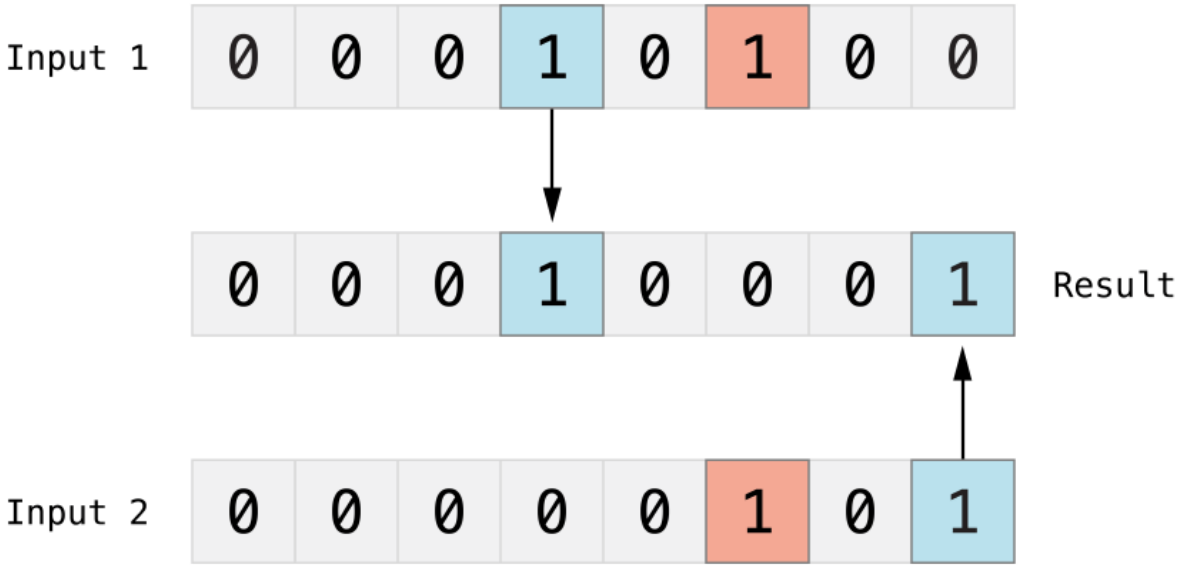
图片 24.3 Image of Advanced_Operators_3.png

如下代码， someBits 和 moreBits 在不同位上有 1 。按位或运行的结果是 11111110 ，即十进制的 254 。

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // 等于 11111110
```

按位异或运算符

按位异或运算符 ^ 比较两个数，然后返回一个数，这个数的每个位设为 1 的条件是两个输入数的同一位不同，如果相同就设为 0 。



图片 24.4 Image of Advanced_Operators_4.png

以下代码， firstBits 和 otherBits 都有一个 1 跟另一个数不同的。所以按位异或的结果是把它这些位置为 1 ，其他都置为 0 。

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // 等于 00010001
```

按位左移/右移运算符

左移运算符 << 和右移运算符 >> 会把一个数的所有比特位按以下定义的规则向左或向右移动指定位数。

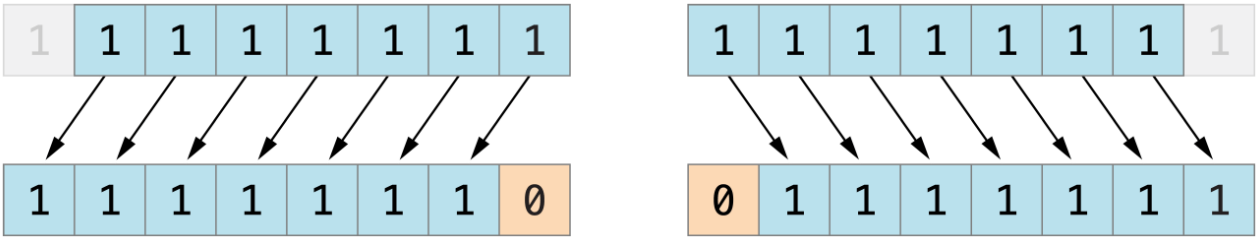
按位左移和按位右移的效果相当把一个整数乘于或除于一个因子为 2 的整数。向左移动一个整型的比特位相当于把这个数乘于 2 ，向右移一位就是除于 2 。

无符整型的移位操作

对无符整型的移位的效果如下：

已经存在的比特位向左或向右移动指定的位数。被移出整型存储边界的的位数直接抛弃，移动留下的空白位用零 0 来填充。这种方法称为逻辑移位。

以下这张把展示了 11111111 << 1 (11111111 向左移1位)，和 11111111 >> 1 (11111111 向右移1位)。蓝色的是被移位的，灰色是被抛弃的，橙色的 0 是被填充进来的。



图片 24.5 Image of Advanced_Operators_5.png

```
let shiftBits: UInt8 = 4   // 即二进制的00000100
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
shiftBits >> 2             // 00000001
```

你可以使用移位操作进行其他数据类型的编码和解码。

```
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
```

这个例子使用了一个 UInt32 的命名为 pink 的常量来存储层叠样式表 CSS 中粉色的颜色值， CSS 颜色 #CC6699 在Swift用十六进制 0xCC6699 来表示。然后使用按位与(&)和按位右移就可以从这个颜色值中解析出红(CC)，绿(66)，蓝(99)三个部分。

对 0xCC6699 和 0xFF0000 进行按位与 & 操作就可以得到红色部分。 0xFF0000 中的 0 了遮盖了 OxCC6699 的第二和第三个字节，这样 6699 被忽略了，只留下 0xCC0000 。

然后，按向右移动16位，即 >> 16 。十六进制中每两个字符是8比特位，所以移动16位的结果是把 0xCC0000 变成 0x0000CC 。这和 0xCC 是相等的，就是十进制的 204 。

同样的，绿色部分来自于 0xCC6699 和 0x00FF00 的按位操作得到 0x006600 。然后向右移动8位，得到 0x66 ，即十进制的 102 。

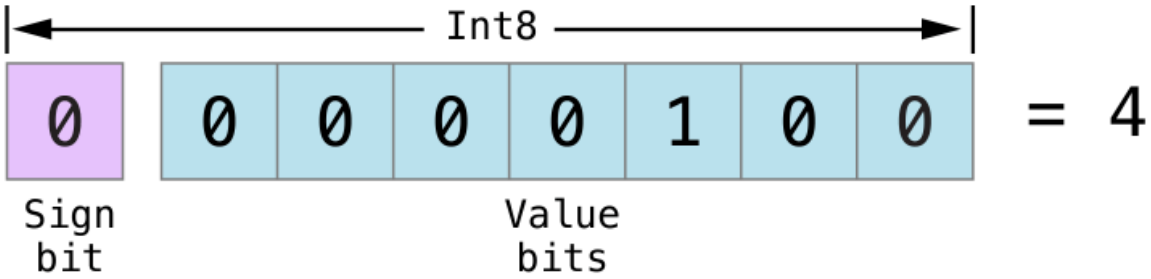
最后，蓝色部分对 0xCC6699 和 0x0000FF 进行按位与运算，得到 0x000099 ，无需向右移位了，所以结果就是 0x99 ，即十进制的 153 。

有符整型的移位操作

有符整型的移位操作相对复杂得多，因为正负号也是用二进制位表示的。(这里举的例子虽然都是8位的，但它的原理是通用的。)

有符整型通过第1个比特位(称为符号位)来表达这个整数是正数还是负数。 0 代表正数， 1 代表负数。

其余的比特位(称为数值位)存储其实值。有符正整数和无符正整数在计算机里的存储结果是一样的，下来我们来看 +4 内部的二进制结构。

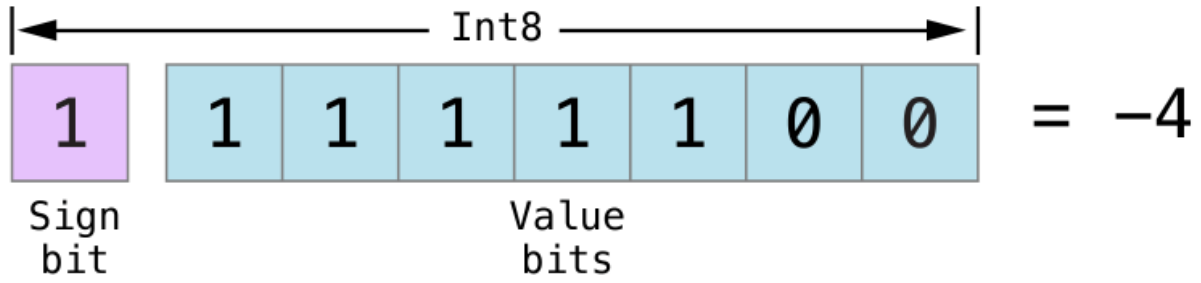


图片 24.6 Image of Advanced_Operators_6.png

符号位为 0 ，代表正数，另外7比特位二进制表示的实际值就刚好是 4 。

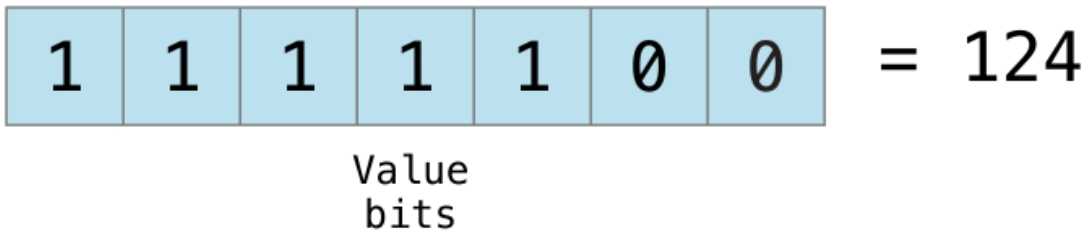
负数呢，跟正数不同。负数存储的是2的n次方减去它的绝对值，n为数值位的位数。一个8比特的数有7个数值位，所以是2的7次方，即128。

我们来看 -4 存储的二进制结构。



图片 24.7 Image of Advanced_Operators_7.png

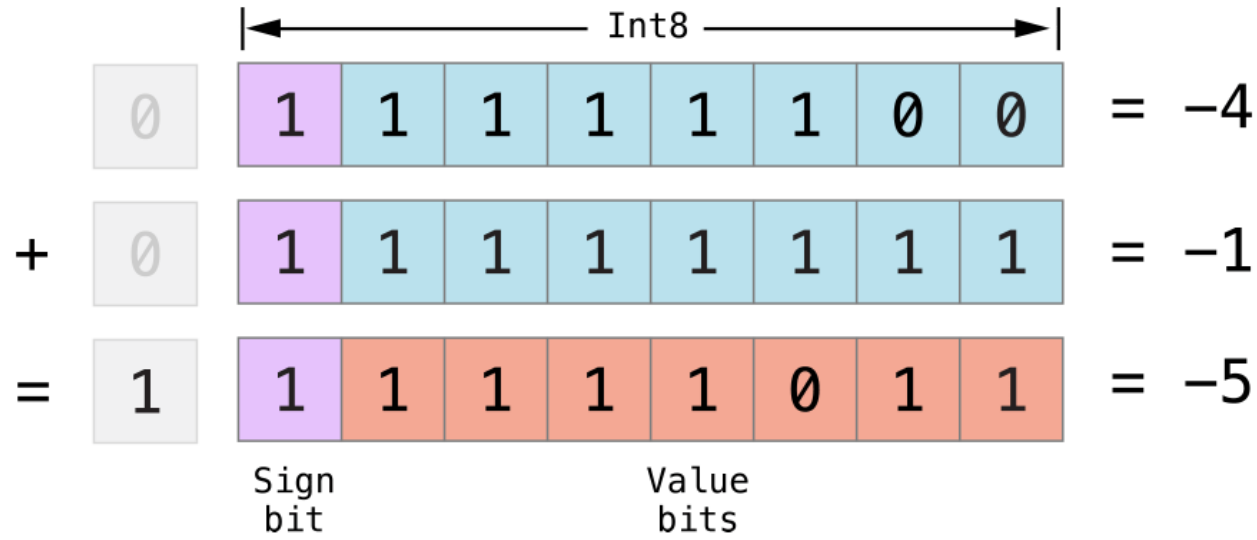
现在符号位为 `1`，代表负数，7个数值位要表达的二进制值是124，即128 − 4。



图片 24.8 Image of Advanced_Operators_8.png

负数的编码方式称为二进制补码表示。这种表示方式看起来很奇怪，但它有几个优点。

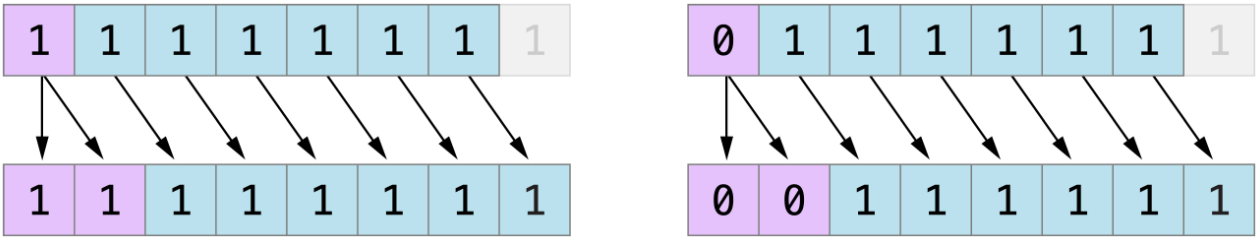
首先，只需要对全部8个比特位(包括符号)做标准的二进制加法就可以完成 `−1 + −4` 的操作，忽略加法过程产生的超过8个比特位表达的任何信息。



图片 24.9 Image of Advanced_Operators_9.png

第二，由于使用二进制补码表示，我们可以和正数一样对负数进行按位左移右移的，同样也是左移1位时乘于 `2`，右移1位时除于 `2`。要达到此目的，对有符整型的右移有一个特别的要求：

对有符整型按位右移时，不使用0填充空白位，而是根据符号位(正数为 `0` ，负数为 `1`)填充空白位。



图片 24.10 Image of Advanced_Operators_10.png

这就确保了在右移的过程中，有符整型的符号不会发生变化。这称为算术移位。

正因为正数和负数特殊的存储方式，向右移位使它接近于 `0` 。移位过程中保持符号会不变，负数在接近 `0` 的过程中一直是负数。

()

溢出运算符

默认情况下，当你往一个整型常量或变量赋于一个它不能承载的大数时，Swift不会让你这么干的，它会报错。这样，在操作过大或过小的数的时候就很安全了。

例如， `Int16` 整型能承载的整数范围是 `-32768` 到 `32767` ，如果给它赋上超过这个范围的数，就会报错：

```
var potentialOverflow = Int16.max
// potentialOverflow 等于 32767, 这是 Int16 能承载的最大整数
potentialOverflow += 1
// 噢, 出错了
```

对过大或过小的数值进行错误处理让你的数值边界条件更灵活。

当然，你有意在溢出时对有效位进行截断，你可采用溢出运算，而非错误处理。Swfit为整型计算提供了5个 `&` 符号开头的溢出运算符。

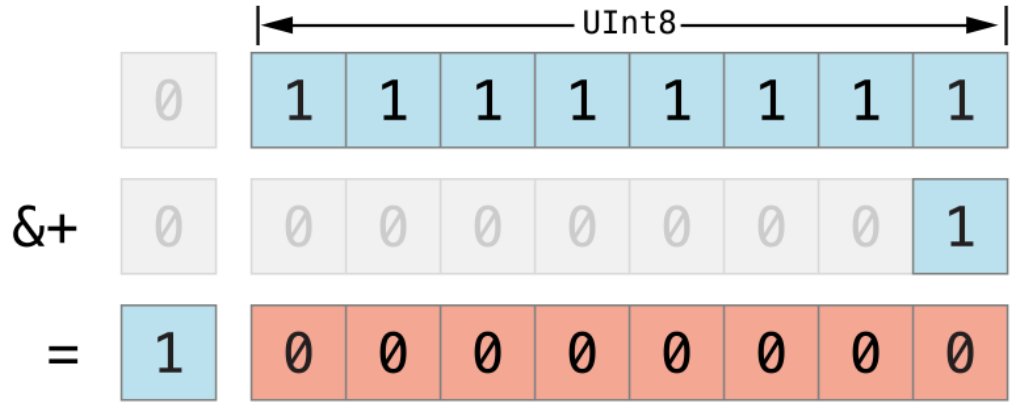
- 溢出加法 `&+`
- 溢出减法 `&-`
- 溢出乘法 `&*`
- 溢出除法 `&/`
- 溢出求余 `&%`

值的上溢出

下面例子使用了溢出加法 `&+` 来解剖的无符整数的上溢出

```
var willOverflow = UInt8.max
// willOverflow 等于UInt8的最大整数 255
willOverflow = willOverflow &+ 1
// 此时 willOverflow 等于 0
```

`willOverflow` 用 `Int8` 所能承载的最大值 `255` (二进制 `11111111`)，然后用 `&+` 加1。然后 `UInt8` 就无法表达这个新值的二进制了，也就导致了这个新值上溢出了，大家可以看下图。溢出后，新值在 `UInt8` 的承载范围内的那部分是 `00000000` ，也就是 `0` 。

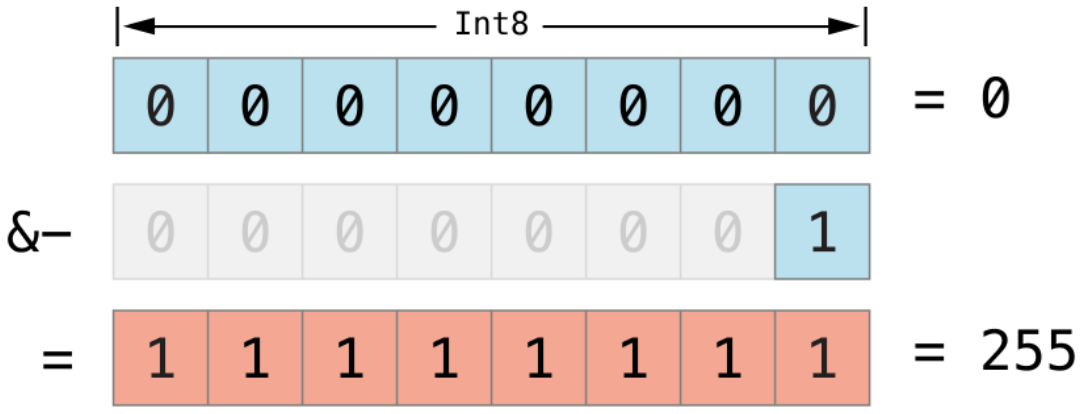


图片 24.11 Image of Advanced_Operators_11.png

值的下溢出

数值也有可能因为太小而越界。举个例子：

`UInt8` 的最小值是 `0` (二进制为 `00000000`)。使用 `&-` 进行溢出减1，就会得到二进制的 `11111111` 即十进制的 `255` 。

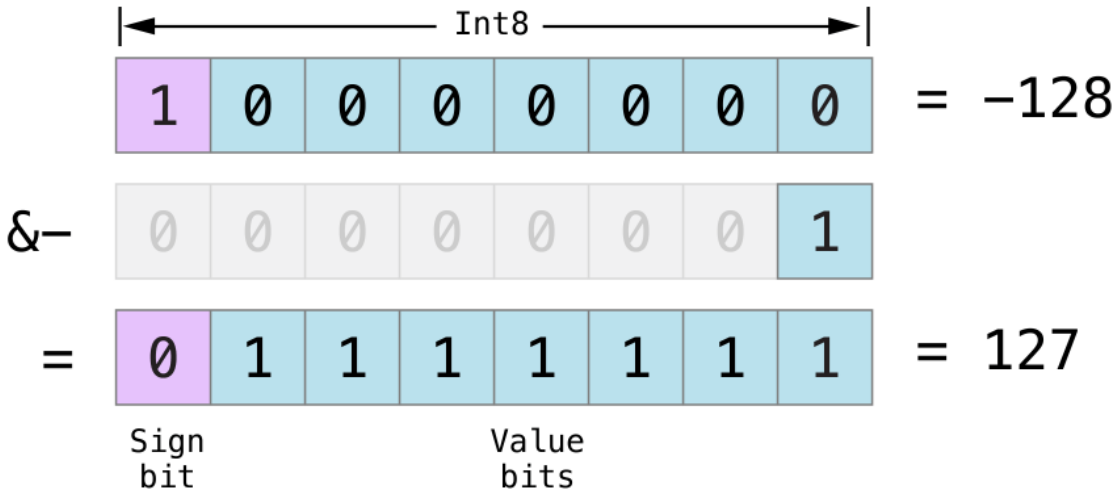


图片 24.12 Image of Advanced_Operators_12.png

Swift代码是这样的:

```
var willUnderflow = UInt8.min
// willUnderflow 等于UInt8的最小值0
willUnderflow = willUnderflow &- 1
// 此时 willUnderflow 等于 255
```

有符整型也有类似的下溢出，有符整型所有的减法也都是对包括在符号位在内的二进制数进行二进制减法的，这在 "按位左移/右移运算符" 一节提到过。最小的有符整数是 `-128`，即二进制的 `10000000`。用溢出减法减去去1后，变成了 `01111111`，即UInt8所能承载的最大整数 `127`。



图片 24.13 Image of Advanced_Operators_13.png

来看看Swift代码:

```
var signedUnderflow = Int8.min
// signedUnderflow 等于最小的有符整数 -128
signedUnderflow = signedUnderflow &- 1
// 此时 signedUnderflow 等于 127
```

除零溢出

一个数除以0 `i / 0`，或者对0求余数 `i % 0`，就会产生一个错误。

```
let x = 1
let y = x / 0
```

使用它们对应的可溢出的版本的运算符 `&/` 和 `&%` 进行除0操作时就会得到 `0` 值。

```
let x = 1
let y = x &/ 0
// y 等于 0
```

()

优先级和结合性

运算符的优先级使得一些运算符优先于其他运算符，高优先级的运算符会先被计算。

结合性定义相同优先级的运算符在一起时是怎么组合或关联的，是和左边的一组呢，还是和右边的一组。意思就是，到底是和左边的表达式结合呢，还是和右边的表达式结合?

在混合表达式中，运算符的优先级和结合性是非常重要的。举个例子，为什么下列表达式的结果为 `4` ?

```
2 + 3 * 4 % 5
// 结果是 4
```

如果严格地从左计算到右，计算过程会是这样:

- 2 + 3 = 5
- 5 * 4 = 20
- 20 / 5 = 4 余 0

但是正确答案是 `4` 而不是 `0` 。优先级高的运算符要先计算，在Swift和C语言中，都是先乘除后加减的。所以，执行完乘法和求余运算才能执行加减运算。

乘法和求余拥有相同的优先级，在运算过程中，我们还需要结合性，乘法和求余运算都是左结合的。这相当于在表达式中有隐藏的括号让运算从左开始。

```
2 + ((3 * 4) % 5)
```

3 * 4 = 12，所以这相当于:

```
2 + (12 % 5)
```

12 % 5 = 2，所这又相当于

```
2 + 2
```

计算结果为 4。

查阅Swift运算符的优先级和结合性的完整列表，请看表达式 (https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Expressions.html) 。

> 注意：
> Swift的运算符较C语言和Objective-C来得更简单和保守，这意味着跟基于C的语言可能不一样。所以，在移植已有代码到Swift时，注意去确保代码按你想的那样去执行。

()

运算符函数

让已有的运算符也可以对自定义的类和结构进行运算，这称为运算符重载。

这个例子展示了如何用 + 让一个自定义的结构做加法。算术运算符 + 是一个两目运算符，因为它有两个操作数，而且它必须出现在两个操作数之间。

例子中定义了一个名为 Vector2D 的二维坐标向量 (x，y) 的结构，然后定义了让两个 Vector2D 的对象相加的运算符函数。

```
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)
}
```

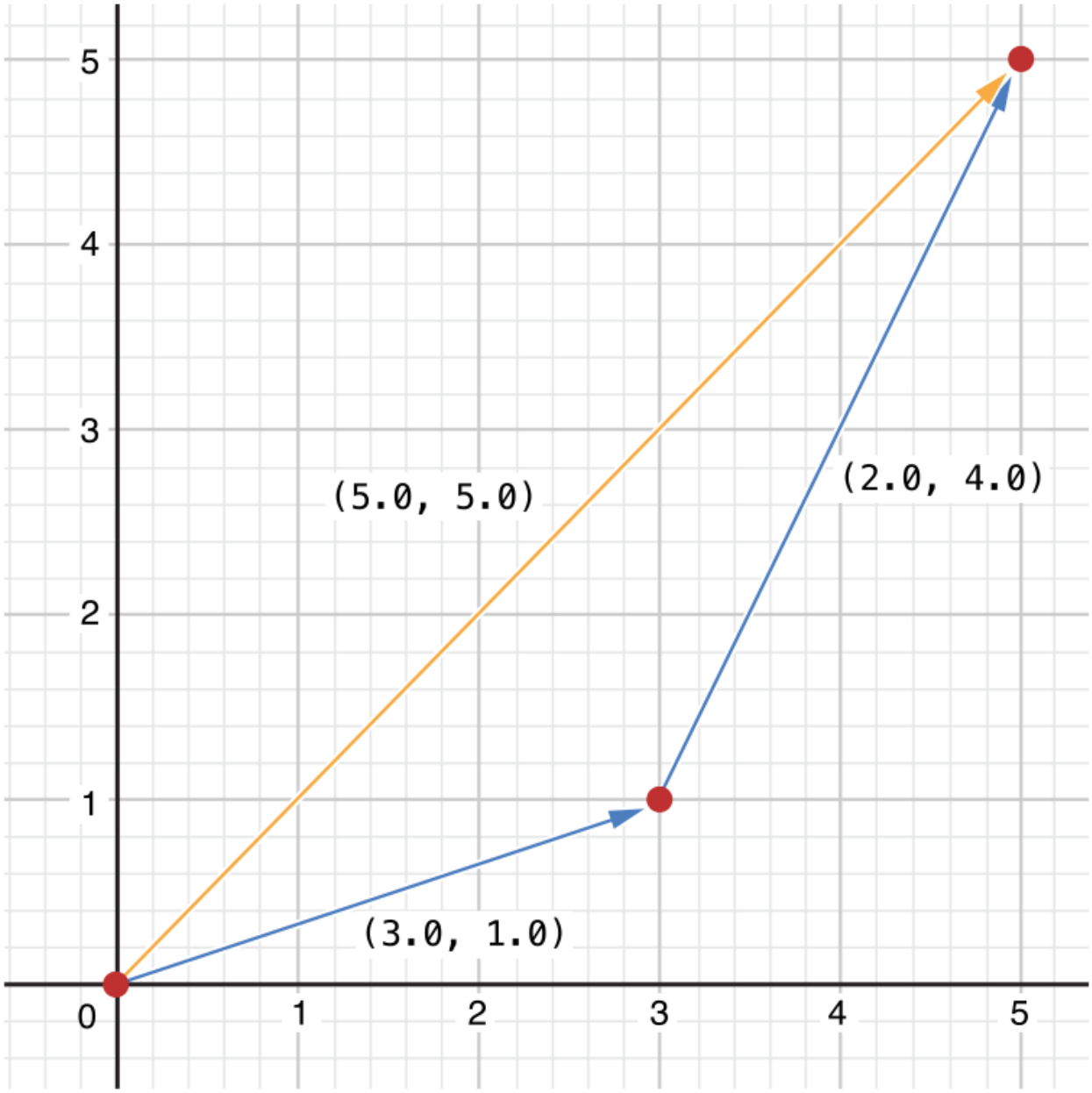
该运算符函数定义了一个全局的 + 函数，这个函数需要两个 Vector2D 类型的参数，返回值也是 Vector2D 类型。需要定义和实现一个中置运算的时候，在关键字 func 之前写上属性 @infix 就可以了。

在这个代码实现中，参数被命名为了 left 和 right ，代表 + 左边和右边的两个 Vector2D 对象。函数返回了一个新的 Vector2D 的对象，这个对象的 x 和 y 分别等于两个参数对象的 x 和 y 的和。

这个函数是全局的，而不是 Vector2D 结构的成员方法，所以任意两个 Vector2D 对象都可以使用这个中置运算符。

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector 是一个新的Vector2D, 值为 (5.0, 5.0)
```

这个例子实现两个向量 (3.0，1.0) 和 (2.0，4.0) 相加，得到向量 (5.0，5.0) 的过程。如下图示：



图片 24.14 Image of Advanced_Operators_14.png

前置和后置运算符

上个例子演示了一个双目中置运算符的自定义实现，同样我们也可以玩标准单目运算符的实现。单目运算符只有一个操作数，在操作数之前就是前置的，如 `-a`；在操作数之后就是后置的，如 `i++` 。

实现一个前置或后置运算符时，在定义该运算符的时候于关键字 `func` 之前标注 `@prefix` 或 `@postfix` 属性。

```
@prefix func - (vector: Vector2D) -> Vector2D {
    return Vector2D(x: -vector.x, y: -vector.y)
}
```

这段代码为 Vector2D 类型提供了单目减运算 -a ， @prefix 属性表明这是个前置运算符。

对于数值，单目减运算符可以把正数变负数，把负数变正数。对于 Vector2D ，单目减运算将其 x 和 y 都进进行单目减运算。

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative 为 (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive 为 (3.0, 4.0)
```

组合赋值运算符

组合赋值是其他运算符和赋值运算符一起执行的运算。如 += 把加运算和赋值运算组合成一个操作。实现一个组合赋值符号需要使用 @assignment 属性，还需要把运算符的左参数设置成 inout ，因为这个参数会在运算符函数内直接修改它的值。

```
@assignment func += (inout left: Vector2D, right: Vector2D) {
    left = left + right
}
```

因为加法运算在之前定义过了，这里无需重新定义。所以，加赋运算符函数使用已经存在的高级加法运算符函数来执行左值加右值的运算。

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original 现在为 (4.0, 6.0)
```

你可以将 @assignment 属性和 @prefix 或 @postfix 属性起来组合，实现一个 Vector2D 的前置运算符。

```
@prefix @assignment func ++ (inout vector: Vector2D) -> Vector2D {
    vector += Vector2D(x: 1.0, y: 1.0)
    return vector
}
```

这个前置使用了已经定义好的高级加赋运算，将自己加上一个值为 (1.0，1.0) 的对象然后赋给自己，然后再将自己返回。

```
var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
// toIncrement 现在是 (4.0, 5.0)
// afterIncrement 现在也是 (4.0, 5.0)
```

> 注意：
>
> 默认的赋值符(=)是不可重载的。只有组合赋值符可以重载。三目条件运算符 a? b: c 也是不可重载。

比较运算符

Swift无所知道自定义类型是否相等或不等，因为等于或者不等于由你的代码说了算了。所以自定义的类和结构要使用比较符 `==` 或 `!=` 就需要重载。

定义相等运算符函数跟定义其他中置运算符雷同：

```
@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)
}
```

上述代码实现了相等运算符 `==` 来判断两个 `Vector2D` 对象是否有相等的值，相等的概念就是它们有相同的 `x` 值和相同的 `y` 值，我们就用这个逻辑来实现。接着使用 `==` 的结果实现了不相等运算符 `!=` 。

现在我们可以使用这两个运算符来判断两个 `Vector2D` 对象是否相等。

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    println("这两个向量是相等的.")
}
// prints "这两个向量是相等的."
```

自定义运算符

标准的运算符不够玩，那你可以声明一些个性的运算符，但个性的运算符只能使用这些字符 `/ = - + * % < > ! & | ^ . ~` 。

新的运算符声明需在全局域使用 `operator` 关键字声明，可以声明为前置，中置或后置的。

```
operator prefix +++ {}
```

这段代码定义了一个新的前置运算符叫 `+++` ，此前Swift并不存在这个运算符。此处为了演示，我们让 `+++` 对 `Vector2D` 对象的操作定义为 `双自增` 这样一个独有的操作，这个操作使用了之前定义的加赋运算实现了自己加上自己然后返回的运算。

```
@prefix @assignment func +++ (inout vector: Vector2D) -> Vector2D {
    vector += vector
    return vector
}
```

`Vector2D` 的 `+++` 的实现和 `++` 的实现很接近, 唯一不同的是前者是加自己, 后者是加值为 `(1.0, 1.0)` 的向量.

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled 现在是 (2.0, 8.0)
// afterDoubling 现在也是 (2.0, 8.0)
```

自定义中置运算符的优先级和结合性

可以为自定义的中置运算符指定优先级和结合性。可以回头看看优先级和结合性 (页 254)解释这两个因素是如何影响多种中置运算符混合的表达式的计算的。

结合性(associativity)的值可取的值有 `left` ， `right` 和 `none` 。左结合运算符跟其他优先级相同的左结合运算符写在一起时，会跟左边的操作数结合。同理，右结合运算符会跟右边的操作数结合。而非结合运算符不能跟其他相同优先级的运算符写在一起。

结合性(associativity)的值默认为 `none` ，优先级(precedence)默认为 `100` 。

以下例子定义了一个新的中置符 `+-` ，是左结合的 `left` ，优先级为 `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 此时的值为 (4.0, -2.0)
```

这个运算符把两个向量的 `x` 相加，把向量的 `y` 相减。因为他实际是属于加减运算，所以让它保持了和加法一样的结合性和优先级(`left` 和 `140`)。查阅完整的Swift默认结合性和优先级的设置，请移步表达式 (https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Expressions.html) 。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/swift-language-guide/