扩展

cnswift.org/extensions

扩展为现有的类、结构体、枚举类型、或协议添加了新功能。这也包括了为无访问权限的源代码扩展类型的能力（即所谓的*逆向建模*）。扩展和 Objective-C 中的分类类似。（与 Objective-C 的分类不同的是，Swift 的扩展没有名字。）

Swift 中的扩展可以：

- 添加计算实例属性和计算类型属性；
- 定义实例方法和类型方法；
- 提供新初始化器；
- 定义下标；
- 定义和使用新内嵌类型；
- 使现有的类型遵循某协议

在 Swift 中，你甚至可以扩展一个协议，以提供其要求的实现或添加符合类型的附加功能。详见协议扩展。

> 注意
>
> 扩展可以向一个类型添加新的方法，但是不能重写已有的方法。

扩展的语法

使用 extension 关键字来声明扩展：

```
1  extension SomeType {
2      // new functionality to add to SomeType goes here
3  }
```

扩展可以使已有的类型遵循一个或多个协议。在这种情况下，协议名的书写方式与类或结构体完全一样：

```
1  extension SomeType: SomeProtocol, AnotherProtocol {
2      // implementation of protocol requirements goes here
3  }
```

用这种方式添加协议一致性详见在扩展里添加协议遵循。

如同扩展一个泛型类型中描述的那样，扩展可以用于丰富现有泛型类型。如同带有泛型 Where 分句的扩展中描述的那样，你也可以可选地给泛型添加功能。

> 注意
>
> 如果你向已存在的类型添加新功能，新功能会在该类型的所有实例中可用，即使实例在该扩展定义之前就已经创建。

计算属性

扩展可以向已有的类型添加计算实例属性和计算类型属性。下面的例子向 Swift 内建的 Double 类型添加了五个计算实例属性，以提供对距离单位的基本支持：

```
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
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters"
let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// Prints "Three feet is 0.914399970739201 meters"
```

这些计算属性表述了 Double 值应被看作是确定的长度单位。尽管它们被实现为计算属性，这些属性的名字仍可使用点符号添加在浮点型的字面量之后，作为一种使用该字面量来执行距离转换的方法。

在这个例子中，一个 1.0 的 Double 值表示“一米”。这就是为什么 m 计算属性要返回 self ——表达式 1.m 表示计算 1.0 的 Double 值。

其他的单位则在以米作为计量值的基础上加以转换表示。一千米表示1000米，所以 km 计算属性将值乘 1_000.00 以用米来表示。类似的，一米有3.28084英尺，所以 ft 计算属性用 Double 值除以3.28084，将英尺转换为米。

上述属性为只读计算属性，为了简洁没有使用 get 关键字。他们都返回 Double 类型的值，可用于所有使用 Double 值的数学计算中：

```
let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
// Prints "A marathon is 42195.0 meters long"
```

> 注意
>
> 扩展可以添加新的计算属性，但是不能添加存储属性，也不能向已有的属性添加属性观察者。

初始化器

扩展可向已有的类型添加新的初始化器。这允许你扩展其他类型以使初始化器接收你的自定义类型作为形式参数，或提供该类型的原始实现中未包含的额外初始化选项。

扩展能为类添加新的便捷初始化器，但是不能为类添加指定初始化器或反初始化器。指定初始化器和反初始化器 必须由原来类的实现提供。

> 注意
>
> 如果你使用扩展为一个值类型添加初始化器，且该值类型为其所有储存的属性提供默认值，而又不定义任何自定义初始化器时，你可以在你扩展的初始化器中调用该类型默认的初始化器和成员初始化器。
>
> 如同在值类型的初始化器委托中所述，如果你在值类型的原始实现中写过它的初始化器了，上述规则就不再适用了。

下面的例子定义了一个自定义的 Rect 结构体用于描述几何矩形。这个例子也定义了两个辅助结构体 Size 和 Point，二者的默认值都是 0.0 ：

```
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()
}
```

如同默认初始化器中描述的那样，由于 Rect 结构体为其所有属性提供了默认值，它将自动接收一个默认的初始化器和一个成员初始化器。这些初始化器能用于创建新的 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))
```

你可以扩展 Rect 结构体以额外提供一个接收特定原点和大小的初始化器：

```
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)
    }
}
```

这个初始化器首先基于提供的 center 点和 size 值计算合适的原点。然后初始化器调用该结构体的自动成员初始化器 init(origin:size:) ，这样就将新的原点和大小值保存在了对应属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

> 注意
>
> 如果你使用扩展提供了一个新的初始化器，你仍应确保每一个实例都在初始化完成时完全初始化。

方法

扩展可以为已有的类型添加新的实例方法和类型方法。下面的例子为 Int 类型添加了一个名为 repetitions 的新实例方法：

```
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }
}
```

repetitions(task:) 方法接收一个 () -> Void 类型的单一实际参数，它表示一个没有参数且无返回值的函数。

在这个扩展定义之后，你可以在任何整型数字处调用 repetitions(task:) 方法，以执行相应次数的操作：

```
3.repetitions {
    print("Hello!")
}
// Hello!
// Hello!
// Hello!
```

异变实例方法

增加了扩展的实例方法仍可以修改（或异变）实例本身。结构体和枚举类型方法在修改 self 或本身的属性时必须标记实例方法为 mutating，和原本实现的异变方法一样。

下面的例子为 Swift 的 Int 类型添加了一个新的异变方法 square，以表示原值的平方：

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt is now 9
```

下标

扩展能为已有的类型添加新的下标。下面的例子为 Swift 内建的 Int 类型添加了一个整型下标。这个下标 [n] 返回了从右开始第 n 位的十进制数字：

- 123456789[0] 返回 9
- 123456789[1] 返回 8

.....以此类推：

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}
746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

若 Int 值没有所需索引的那么多数字，下标实现返回 0，就像是这个数左边用零填充：

```
746381295[9]
// returns 0, as if you had requested:
0746381295[9]
```

内嵌类型

扩展可以为已有的类、结构体和枚举类型添加新的内嵌类型：

```
extension Int {
    enum Kind {
        case negative, zero, positive
    }
    var kind: Kind {
        switch self {
        case 0:
            return .zero
        case let x where x > 0:
            return .positive
        default:
            return .negative
        }
    }
}
```

这个例子为 Int 添加了新的内嵌枚举类型。这个名为 Kind 的枚举类型表示一个特定整数的类型。具体表示了这个数字是负数、零还是正数。

这个例还向 Int 中添加了新的计算实例属性 kind，以返回该整数的合适 Kind 枚举情况。

这个内嵌的枚举类型可以和任意 Int 一起使用：

```
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .negative:
            print("- ", terminator: "")
        case .zero:
            print("0 ", terminator: "")
        case .positive:
            print("+ ", terminator: "")
        }
    }
    print("")
}
printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// Prints "+ + - 0 - 0 + "
```

这里 printIntegerKinds(_:) 函数接收一个 Int 的数组并对这些值进行遍历。对数组的每一个数字，函数考虑这个整数的 kind 计算属性，并输出合适的描述。

> 注意
>
> 已知 number.kind 是 Int.Kind 类型。因此， switch 语句中的所有 Int.Kind 情况值都可以简写，例如用 .Negative 而不是 Int.Kind.Negative 。