扩展

4.2

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

`Extensions`：为已存在的类、结构体、枚举或者协议类型增添了一个新的功能。这项功能包括在对你无法访问源码的情况下进行类的扩展的能力（例如“追溯模型”）。Swift 中的 `Extensions` 与 Objective-C 中的 `Categories` 类似。（但与 Objective-C 中 `Categories` 有所不同的是，Swift 中的 `Extensions` 并没有一个具体的命名）

在 Swift 中 `Extensions` 可以做到：

- 添加计算实例属性和计算类型属性
- 定义实例方法和类方法
- 提供新的初始化方法
- 定义下标脚本
- 定义和使用新的嵌套类型
- 使现有类型符合协议

在 Swift 中，甚至你可以实现扩展一个协议去满足你的需求或者增添一个额外的类型去使用。更多信息请参阅 Protocol Extensions

> 注意
>
> `Extensions` 可以为类增添一个新的功能，但却不能重写之前已经存在的功能。

`Extension` 语法

使用关键字 `Extension` 来声明一个扩展：

```
extension SomeType {
    //编写 SomeType 的新功能
```

```
}
```

一个 `Extension` 可以实现扩展现有类型去遵循一个或多个协议。为了保持协议的一致性，请你以在声明这个类或者结构体时的方式去声明这个协议名。

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // 实现协议内容
}
```

以这种方式去保持协议描述的一致性 Adding Protocol Conformance with an Extension.

一个 `Extension` 可以被用于去扩展一个泛型，例如 Extending a Generic Type. 当然你也可以去为所扩展的泛型添加一些功能条件，如下所述 Extensions with a Generic Where Clause.

注意

如果你想为一个已经存在的类型的进行扩展并添加一个新的功能，那么这个功能将会被该类所有的实例使用，即使在这个 `Extension` 被定义之前。

计算属性

`Extensions` 可以将计算实例属性与计算类型属性添加到现有类中去。例如以下就是在 `Swift` 的内置类型 `Double` 中添加了五个计算实例属性，为距离单位提供了基础支持：

```
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"
```

这些计算属性应该被看作是 `Double` 值中的一个特定长度单位。尽管它们被看作是计算属性而被实现的，但是这些属性仍然可以被用于浮点类型的点语法而调用，被作为一种字面值去进行距离转换。

在这个举例中，当 `Double` 值为 `1.0` 时可被看作是 `1米` 。这就是为什么使用 `m` 时的计算属性会返回 `Self` -- 表达式 `1.m` 就表示是 `1.0` 的 `Double` 值。

其它的单位就需要作一些转换以便用作于以米为单位的表达。一公里就表示一千米，所以 `km` 这个计算属性将会通过乘以 `1_000.00` 去转换为米作单位去表达。同样， `3.28084` 英尺也可以表示为 `1米` ，因此 `ft` 的计算属性将用 `Double` 值去除以 `3.28084` ，从而实现从英尺到米的转换。

为了代码的简洁性，这些属性是只读的，所以他们不能使用 `get` 关键字去调用。当然他们的返回值也是 `Double` 类型，并且他们能在任何可以使用 `Double` 类型的地方进行数学计算。

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

> 注意
>
> `Extensions` 可以添加一个新的属性，但是他们不能存储这些属性，也不能为现有类型添加属性观察者。

初始化器

`Extensions` 可以给现有类型添加一个新的初始化构造器。 这使你能够扩展一些其他类型去接收你自定义的类型用作为初始化参数，或者提供该类型的源码实现中未包含的初始化类型。

`Extensions` 可以给类添加一个便利初始化器，但它们不能够提供特定的初始化器或者反初始化器。特定的初始化器和反初始化器都必须由原始类去提供。

如果你要为一个值类型去添加初始化器，并将这个值所存储的属性设置默认值，而又不为其定义任何自定义的初始化器时，那么你可以在你所扩展的初始化器中调用该类型的默认初始化器和成员初始化器。如果你已经将这个初始化器作为这个值类型的原始实现的一部分，那么将不再遵循以上规则, 如下所述 Initializer Delegation for Value Types.

如果你使用一个 `Extension` 去给另一个模块中声明的结构添加一个初始化器，那么在这个模块被定义之前，这个新的初始化器将无法访问 `Self` 。

下面的例子自定义了一个名为 `Rect` 的结构体去表示一个几何矩形。同时还定义了 `Size` 和 `Point` 两个结构体去支持它, 这两个结构体都的属性默认值都被赋值为 `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()
}
```

因为 `Rect` 结构体为其属性提供了默认值, 所以他可以接受它的默认初始化器以及成员初始化器， 如下描述 Default Initializers 这些初始化器可以被用于去创建一个新的 `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))
```

你可以为 `Rect` 结构体扩展一个额外的初始化器，并且给与这个额外的初始化器特定的 `Center` 和 `Size` 值:

```
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)
    }
}
```

这个新的初始化器可以基于 `Center` 和 `Size` 的值去计算一个恰当的初始点，然后通过这个初始化器去调用 `init(origin:size:)` 这个自动初始化成员方法，这将会把新的 `Origin` 和 `Size` 保存在相对的属性中:

```
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

如果你声明了一个带有扩展的新的初始化器，那么你仍然有责任去确认在这个初始化器完成之后，每一个实例都已经被初始化完成。

方法

扩展可以向已经存在的类型添加实例方法或类方法。

在下面的例子中，我们将为 `Int` 类型添加一个新的实例方法 `repetitions` ：

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

`repetitions(task:)` 这个方法接收一个 `() -> Void` 类型的参数，也就是一个没有参数没有返回值的闭包。

在成功定义这个扩展之后，你可以通过点语法在任意 `Int` 类型之后调用 `repetitions` 这个方法，并在闭包中输入你希望执行的代码，代码被执行的次数由这个 `Int` 决定。

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

可变实例方法

我们可以通过在扩展中添加实例方法来实现修改变量。

MarvinSu 翻译于 5个月

0 重译

由 Aufree 审阅

在结构体以及枚举当中，我们只需要对需要改变成员变量的方法添加 `mutating` 关键字即可。

在下面这个例子当中，由于 `square` 这个方法已经修改了 `self` 的值，所以我们需要在方法前加上 `mutating` 关键字。

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt 现在已经变成 9 了
```

下标

Extensions 能够对已经存在的类型添加下标。 下面的例子就是对 Swift 的内置 `Int` 类型添加一个整型下标。通过下标 `[n]` 返回十进制数字的从右数 `n` 个位置的数字:

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

...等等:

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

```
// 返回 7
```

如果这个 `Int` 类型的值对于请求的索引没有足够的数字，使用下标访问会返回 `0` ，就如同用0填充了数字的左边:

```
746381295[9]
// 返回 0, 如同这样的请求:
0746381295[9]
```

嵌套类型

我们可以通过扩展 `Extensions` 来向任何已经存在的类、结构体或枚举添加新的嵌套类型。

```
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
        }
    }
}
```

这是一个针对 `Int` 类型的扩展 `Kind` ，在这个扩展中，我们可以针对任意一个 `Int` 类型作出进一步的细分，比如：**负数（negative）**，**零 （zero）**，与**正数（positive）**。

在这个例子中还另外为 `Int` 添加了一个计算属性 `kind` ，这个属性可以根据具体的数字返回对于该数字的描述。

现在这个嵌套枚举已经可以在所有的 `Int` 变量中使用:

```
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
```

MarvinSu 翻译于 5个月

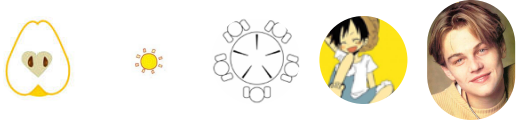
0 重译

由 Aufree 审阅

```
        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 + "
```

在 `printIntegerKinds(_:)` 这个函数中，它接收一个 `Int` 类型的数组，然后对其进行遍历，再根据嵌套枚举中 `kind` 这个计算属性对每一个元素输出对应的描述。

> 注意
>
> 因为 `number.kind` 已经定义在 `Int.Kind` 扩展中，所以我们可以在 `switch` 分支语句中直接使用，出于简洁性考虑，对比 `Int.Kind.negative` 这种写法，`.negative` 会显得更 Swift。



更多职位

讨论数量: 0

发起讨论

只看当前版本讨论

暂无话题~