扩展

*扩展*为现有类，结构，枚举或协议类型添加新功能。这包括扩展您无法访问原始源代码的类型（称为*追溯建模*）的功能。扩展与Objective-C中的类别相似。（与Objective-C类别不同，Swift扩展没有名称。）

Swift中的扩展可以：

- 添加计算的实例属性和计算的类型属性
- 定义实例方法和类型方法
- 提供新的初始化程序
- 定义下标
- 定义和使用新的嵌套类型
- 使现有类型符合协议

在Swift中，您甚至可以扩展协议以提供其需求的实现，或者添加符合类型可以利用的其他功能。有关更多详细信息，请参阅协议扩展。

> 注意
>
> 扩展可以向类型添加新功能，但是它们不能覆盖现有功能。

扩展语法

用`extension`关键字声明扩展名：

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

扩展可以扩展现有类型以使其采用一个或多个协议。要添加协议一致性，您需要按照为类或结构编写协议名称的方式编写协议名称：

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

在添加扩展协议一致性中描述了以这种方式添加协议一致性。

可以使用扩展来扩展现有的泛型类型，如扩展泛型类型中所述。您还可以扩展通用类型以有条件地添加功能，如扩展中使用通用Where子句中所述。

> 注意
>
> 如果您定义了一个扩展来为现有类型添加新功能，即使在定义扩展之前创建了新功能，新功能也可用于该类型的所有现有实例。

计算属性

扩展可以将计算的实例属性和计算的类型属性添加到现有类型。本示例向Swift的内置`Double`类型添加了五个计算实例属性，为使用距离单位提供基本支持：

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

这些计算出的属性表示Double应将某个值视为某个长度单位。虽然它们是作为计算属性实现的，但可以使用点语法将这些属性的名称附加到浮点文字值，作为使用该文字值执行距离转换的一种方法。

在这个例子中，Double值1.0被认为代表“一米”。这就是为什么m计算属性返回self- 表达式1.m被认为是计算Double值1.0。

其他单位需要将某些转换表达为以米为单位的值。一公里与1,000米相同，所以km计算出的特性乘以该值1_000.00以转换成以米为单位的数字。同样，一米内有3.28084英尺，所以ft计算出的财产将基础Double价值除以3.28084，将其从英尺转换为米。

这些属性是只读的计算属性，因此get为了简洁起见，它们表达为没有关键字。它们的返回值是类型的Double，可以在数学计算中用于任何Double被接受的地方：

```
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，但不能添加存储的属性，或者将属性观察器添加到现有属性。

初始化器

扩展可以添加新的初始化器到现有的类型。这使您可以扩展其他类型以接受您自己的自定义类型作为初始化参数，或提供附加的初始化选项，这些初始化选项不包含在类型的原始实现中。

扩展可以将新的便捷初始值设定项添加到类中，但不能将新的指定初始值设定项或去初始化项添加到类中。指定的初始化器和去初始化器必须始终由原始类实现提供。

如果使用扩展将初始值设定项添加到为其所有存储属性提供默认值并且未定义任何自定义初始值设定项的值类型，则可以在扩展的初始值设定项中为该值类型调用默认初始值设定项和成员初始值设定项。如果您已将初始值设定项作为值类型原始实现的一部分编写，则情况不会如此，如“初始值设定项的值类型委派”中所述。

如果使用扩展将初始化程序添加到在另一个模块中声明的结构中，则新的初始化程序self只有在从定义模块调用初始化程序之后才能访问。

下面的例子定义了一个Rect表示几何矩形的自定义结构。该示例还定义了两个称为支撑结构Size和Point，两者都提供的默认值0.0对于所有其属性的：

```
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()
}
```

由于Rect结构为其所有属性提供默认值，因此它会自动接收默认初始值设定项和成员项初始值设定项，如默认初始值设定项中所述。这些初始化器可以用来创建新的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))
```

您可以扩展该Re

```
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)
    }
}
```

这个新的初始化器通过基于提供的center点和size值计算适当的原点开始。初始化程序然后调用结构的自动成员初始化程序init(origin:size:)，它将新的原点和大小值存储在适当的属性中：

```
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
> 如果您为扩展提供了新的初始化程序，则仍然有责任确保初始化程序完成后每个实例都完全初始化。

方法

扩展可以添加新的实例方法和类型方法到现有的类型。以下示例添加了一个名为repetitions该Int类型的新实例方法：

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

该repetitions(task:)方法只接受一个类型的参数() -> Void，它表示一个没有参数且不返回值的函数。

定义此扩展后，您可以调用repetitions(task:)任何整数的方法来执行多次任务：

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

突变实例方法

添加了扩展的实例方法也可以修改（或*改变*）实例本身。修改的结构和枚举方法self或其属性必须将实例方法标记为mutating，就像从原始实现中改变方法一样。

下面的例子添加了一个叫做squareSwift Int类型的新的变异方法，它将原始值平方：

```
extension Int {
    mutating func square() {
        self = self * self
    }
```

```
}
var someInt = 3
someInt.square()
// some
```

标

扩展可以将新的下标添加到现有类型。这个例子为Swift的内置`Int`类型添加了一个整数下标。该下标从`[n]`数字`n`右侧返回小数位数：

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

...等等：

```
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
```

如果该`Int`值对于所请求的索引没有足够的位数，则下标实现将返回`0`，就好像该数字已用左边的零填充：

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

嵌套类型

扩展可以将新的嵌套类型添加到现有的类，结构和枚举中：

```
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
        }
    }
}
```

这个例子添加了一个新的嵌套枚举Int。此枚举称为Kind表示特定整数表示的数字种类。具体而言，它表示数字是否为负数，零或正数。

这个例子还增加了一个新的计算实例属性Int，称为kind，它返回该Kind整数的适当枚举大小写。

嵌套的枚举现在可以用于任何Int值：

```
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 + "
```

这个函数printIntegerKinds(_:)接受一个输入Int值的数组并依次迭代这些值。对于数组中的每个整数，函数会考虑该kind整数的计算属性，并打印适当的描述。

> 注意
>
> number.kind已知是类型的Int.Kind。正因为如此，所有的Int.Kind情况下，值可以简写形式里面switch的语句，如.negative不是Int.Kind.negative。