Swift 扩展 Swift 泛型

Swift 协议

协议规定了用来实现某一特定功能所必需的方法和属性。

任意能够满足协议要求的类型被称为遵循(conform)这个协议。

类，结构体或枚举类型都可以遵循协议，并提供具体实现来完成协议定义的方法和功能。

语法

协议的语法格式如下：

```
protocol SomeProtocol {
    // 协议内容
}
```

要使类遵循某个协议，需要在类型名称后加上协议名称，中间以冒号:分隔，作为类型定义的一部分。遵循多个协议时，各协议之间用逗号,分隔。

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // 结构体内容
}
```

如果类在遵循协议的同时拥有父类，应该将父类名放在协议名之前，以逗号分隔。

```
class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {
    // 类的内容
}
```

对属性的规定

协议用于指定特定的实例属性或类属性，而不用指定是存储型属性或计算型属性。此外还必须指明是只读的还是可读可写的。

协议中的通常用var来声明变量属性，在类型声明后加上{ set get }来表示属性是可读可写的，只读属性则用{ get }来表示。

```
protocol classa {

    var marks: Int { get set }
    var result: Bool { get }

    func attendance() -> String
    func markssecured() -> String

}
```

```
protocol classb: classa {

    var present: Bool { get set }
    var subject: String { get set }
    var stname: String { get set }

}

class classc: classb {
    var marks = 96
    let result = true
    var present = false
    var subject = "Swift 协议"
    var stname = "Protocols"

    func attendance() -> String {
        return "The \(stname) has secured 99% attendance"
    }

    func markssecured() -> String {
        return "\(stname) has scored \(marks)"
    }
}

let studdet = classc()
studdet.stname = "Swift"
studdet.marks = 98
studdet.markssecured()

print(studdet.marks)
print(studdet.result)
print(studdet.present)
print(studdet.subject)
print(studdet.stname)
```

以上程序执行输出结果为：

```
98
true
false
Swift 协议
Swift
```

对 Mutating 方法的规定

有时需要在方法中改变它的实例。

例如，值类型(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它所属的实例及其实例属性的值。

```
protocol daysofaweek {
    mutating func show()
}

enum days: daysofaweek {
    case sun, mon, tue, wed, thurs, fri, sat
    mutating func show() {
        switch self {
        case .sun:
            self = .sun
            print("Sunday")
        case .mon:
            self = .mon
            print("Monday")
        case .tue:
            self = .tue
            print("Tuesday")
        case .wed:
            self = .wed
            print("Wednesday")
        case .thurs:
            self = .thurs
            print("Wednesday")
        case .fri:
            self = .fri
            print("Wednesday")
        case .sat:
            self = .sat
            print("Saturday")
        default:
            print("NO Such Day")
        }
    }
}

var res = days.wed
res.show()
```

以上程序执行输出结果为：

```
Wednesday
```

对构造器的规定

协议可以要求它的遵循者实现指定的构造器。

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声明，但不需要写花括号和构造器的实体，语法如下：

```
protocol SomeProtocol {
   init(someParameter: Int)
}
```

实例

```
protocol tcpprotocol {
   init(aprot: 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，并指定其为类的指定构造器或者便利构造器。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
   }
}

protocol tcpprotocol {
   init(aprot: Int)
}

class tcpClass: tcpprotocol {
   required init(aprot: Int) {
   }
}
```

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。

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 tcpprotocol {
    init(no1: Int)
}

class mainClass {
```

```
    var no1: Int // 局部变量
    init(no1: Int) {
        self.no1 = no1 // 初始化
    }
}

class subClass: mainClass, tcpprotocol {
    var no2: Int
    init(no1: Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 因为遵循协议，需要加上"required"; 因为继承自父类，需要加上"override"
    required override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}
let res = mainClass(no1: 20)
let show = subClass(no1: 30, no2: 50)

print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

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。

协议可以像其他普通类型一样使用，使用场景:

- 作为函数、方法或构造器中的参数类型或返回值类型
- 作为常量、变量或属性的类型
- 作为数组、字典或其他容器中的元素类型

实例

```
protocol Generator {
    associatedtype members
    func next() -> members?
}
```

```
var items = [10,20,30].makeIterator()
while let x = items.next() {
    print(x)
}

for lists in [1,2,3].map( {i in i*5}) {
    print(lists)
}

print([100,200,300])
print([1,2,3].map({i in i*10}))
```

以上程序执行输出结果为：

```
10
20
30
5
10
15
[100, 200, 300]
[10, 20, 30]
```

在扩展中添加协议成员

我们可以可以通过扩展来扩充已存在类型(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。

```
protocol AgeClasificationProtocol {
   var age: Int { get }
   func agetype() -> String
}

class Person {
   let firstname: String
   let lastname: String
   var age: Int
   init(firstname: String, lastname: String) {
      self.firstname = firstname
      self.lastname = lastname
      self.age = 10
   }
}

extension Person : AgeClasificationProtocol {
   func fullname() -> String {
```

```
        var c: String
        c = firstname + " " + lastname
        return c
    }

    func agetype() -> String {
        switch age {
        case 0...2:
            return "Baby"
        case 2...12:
            return "Child"
        case 13...19:
            return "Teenager"
        case let x where x > 65:
            return "Elderly"
        default:
            return "Normal"
        }
    }
}
```

协议的继承

协议能够继承一个或多个其他协议，可以在继承的协议基础上增加新的内容要求。

协议的继承语法与类的继承相似，多个被继承的协议间用逗号分隔：

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // 协议定义
}
```

实例

```
protocol Classa {
    var no1: Int { get set }
    func calc(sum: Int)
}

protocol Result {
    func print(target: Classa)
}

class Student2: Result {
    func print(target: Classa) {
        target.calc(1)
    }
}
```

```
class Classb: Result {
    func print(target: Classa) {
        target.calc(5)
    }
}

class Student: Classa {
    var no1: Int = 10

    func calc(sum: Int) {
        no1 -= sum
        print("学生尝试 \(sum) 次通过")

        if no1 <= 0 {
            print("学生缺席考试")
        }
    }
}

class Player {
    var stmark: Result!

    init(stmark: Result) {
        self.stmark = stmark
    }

    func print(target: Classa) {
        stmark.print(target)
    }
}

var marks = Player(stmark: Student2())
var marksec = Student()

marks.print(marksec)
marks.print(marksec)
marks.print(marksec)
marks.stmark = Classb()
marks.print(marksec)
marks.print(marksec)
marks.print(marksec)
```

以上程序执行输出结果为：

```
学生尝试 1 次通过
学生尝试 1 次通过
学生尝试 1 次通过
学生尝试 5 次通过
```

```
学生尝试 5 次通过
学生缺席考试
学生尝试 5 次通过
学生缺席考试
```

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。

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。格式如下：

```
protocol SomeClassOnlyProtocol: class, SomeInheritedProtocol {
    // 协议定义
}
```

实例

```
protocol TcpProtocol {
    init(no1: Int)
}

class MainClass {
    var no1: Int // 局部变量
    init(no1: Int) {
        self.no1 = no1 // 初始化
    }
}

class SubClass: MainClass, TcpProtocol {
    var no2: Int
    init(no1: Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 因为遵循协议，需要加上"required"; 因为继承自父类，需要加上"override"
    required override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}

let res = MainClass(no1: 20)
let show = SubClass(no1: 30, no2: 50)

print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

协议合成

Swift 支持合成多个协议，这在我们需要同时遵循多个协议时非常有用。

语法格式如下：

```
protocol Stname {
    var name: String { get }
}

protocol Stage {
    var age: Int { get }
}

struct Person: Stname, Stage {
    var name: String
    var age: Int
}

func show(celebrator: Stname & Stage) {
    print("\(celebrator.name) is \(celebrator.age) years old")
}

let studname = Person(name: "Priya", age: 21)
print(studname)

let stud = Person(name: "Rehan", age: 29)
print(stud)

let student = Person(name: "Roshan", age: 19)
print(student)
```

以上程序执行输出结果为：

```
Person(name: "Priya", age: 21)
Person(name: "Rehan", age: 29)
Person(name: "Roshan", age: 19)
```

检验协议的一致性

你可以使用is和as操作符来检查是否遵循某一协议或强制转化为某一类型。

- 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，如果强转失败，会引起运行时错误。

实例

下面的例子定义了一个 HasArea 的协议，要求有一个Double类型可读的 area：

```
protocol HasArea {
    var area: Double { get }
}

// 定义了Circle类，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 }
}

// 定义了Country类，都遵循了HasArea协议
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}

// Animal是一个没有实现HasArea协议的类
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}

let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]

for object in objects {
    // 对迭代出的每一个元素进行检查，看它是否遵循了HasArea协议
    if let objectWithArea = object as? HasArea {
        print("面积为 \(objectWithArea.area)")
    } else {
        print("没有面积")
    }
}
```

以上程序执行输出结果为：

```
面积为 12.5663708
面积为 243610.0
没有面积
```

Swift 扩展 Swift 泛型

点我分享笔记