

# The Swift Programming Language in Chinese

<https://github.com/letsswift>

<http://letsswift.com/>

June 15, 2014

# 目录

<b>1 Swift 中文教程 (二十一) 协议</b>	<b>1</b>
1.1 协议的语法	1
1.2 属性要求	1
1.3 方法要求	3
1.4 突变方法要求	4
1.5 协议类型	5
1.6 委托 (代理) 模式	6
1.6.1 在扩展中添加协议成员	9
1.6.2 通过延展补充协议声明	10
1.6.3 集合中的协议类型	11
1.7 协议的继承	11
1.7.1 协议合成	12
1.7.2 检验协议的一致性	13
1.7.3 可选协议要求	15

## 1 Swift 中文教程 (二十一) 协议

Protocol(协议) 用于统一方法和属性的名称, 而不实现任何功能。协议能够被类, 枚举, 结构体实现, 满足协议要求的类, 枚举, 结构体被称为协议的遵循者。

遵循者需要提供协议指定的成员, 如属性, 方法, 操作符, 下标等。

### 1.1 协议的语法

协议的定义与类, 结构体, 枚举的定义非常相似, 如下所示:

```
protocol SomeProtocol {  
    // 空协议  
}
```

在类, 结构体, 枚举的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 {  
    // 空结构体  
}
```

当某个类含有父类的同时并实现了协议, 应当把父类放在所有的协议之前, 如下所示:

```
class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {  
    // 空类  
}
```

### 1.2 属性要求

协议能够要求其遵循者必须含有一些特定名称和类型的实例属性 (instance property) 或类属性 (type property), 也能够要求属性的 (设置权限)settable 和 (访问权限)gettable, 但它不要求属性是存储型属性 (stored property) 还是计算型属性 (calculate property)。

通常前置 var 关键字将属性声明为变量。在属性声明后写上 { get set } 表示属性为可读写的。{ get } 用来表示属性为可读的。即使你为可读的属性实现了 setter 方法, 它也不会出错。

```
protocol SomeProtocol {
    var musBeSettable : Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

用类来实现协议时，使用 `class` 关键字来表示该属性为类成员；用结构体或枚举实现协议时，则使用 `static` 关键字来表示：

```
protocol AnotherProtocol {
    class var someTypeProperty: Int { get set }
}
```

```
protocol FullyNamed {
    var fullName: String { get }
}
```

`FullyNamed` 协议含有 `fullName` 属性。因此其遵循者必须含有一个名为 `fullName`，类型为 `String` 的可读属性。

```
struct Person: FullyNamed{
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
//john.fullName == "John Appleseed"
```

`Person` 结构体含有一个名为 `fullName` 的存储型属性，完整的遵循了协议。(若协议未被完整遵循，编译时则会报错)。

如下所示，`Starship` 类遵循了 `FullyNamed` 协议：

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil ) {
        self.anme = name
        self.prefix = prefix
    }
    var fullName: String {
```

```
        return (prefix ? prefix ! + " " : " ") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName == "USS Enterprise"
```

Starship 类将 fullName 实现为可读的计算型属性。它的每一个实例都有一个名为 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。

### 1.3 方法要求

协议能够要求其遵循者必备某些特定的实例方法和类方法。协议方法的声明与普通方法声明相似，但它不需要方法内容。

注意：协议方法支持变长参数 (variadic parameter)，不支持默认参数 (default parameter)。

前置 class 关键字表示协议中的成员为类成员；当协议用于被枚举或结构体遵循时，则使用 static 关键字。如下所示：

```
protocol SomeProtocol {
    class func someTypeMethod()
}

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。(我们假设随机数在 [0, 1] 区间内)。

LinearCongruentialGenerator 类遵循了 RandomNumberGenerator 协议，并提供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"

```

## 1.4 突变方法要求

能在方法或函数内部改变实例类型的方法称为突变方法。在值类型 (Value Type)(译者注: 特指结构体和枚举) 中的函数前缀加上 `mutating` 关键字来表示该函数允许改变该实例和其属性的类型。这一变换过程在实例方法 (Instance Methods) 章节中有详细描述。

(译者注: 类中的成员为引用类型 (Reference Type), 可以方便的修改实例及其属性的值而无需改变类型; 而结构体和枚举中的成员均为值类型 (Value Type), 修改变量的值就相当于修改变量的类型, 而 Swift 默认不允许修改类型, 因此需要前置 `mutating` 关键字用来表示该函数中能够修改类型)

注意: 用 `class` 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 `toggle` 函数。根据函数名称推测, `toggle` 可能用于切换或恢复某个属性的状态。`mutating` 关键字表示它为突变方法:

```

protocol Togglable {
    mutating func toggle()
}

```

当使用枚举或结构体来实现 `Togglable` 协议时, 必须在 `toggle` 方法前加上 `mutating` 关键字。

如下所示, OnOffSwitch 枚举遵循了 Toggable 协议, On, Off 两个成员用于表示当前状态

```
enum OnOffSwitch: Toggable {
    case Off, On
    mutating func toggle() {
        switch self {
        case Off:
            self = On
        case On:
            self = Off
        }
    }
}

var lightSwitch = OnOffSwitch.Off
lightSwitch.toggle()
//lightSwitch == On
```

## 1.5 协议类型

协议本身不实现任何功能,但你可以将它当做类型来使用。

使用场景:

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

注意: 协议类型应与其他类型 (Int, Double, String) 的写法相同,使用驼峰式

```
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
    }
}

```

这里定义了一个名为 `Dice` 的类，用来代表桌游中的 `N` 个面的骰子。

`Dice` 含有 `sides` 和 `generator` 两个属性，前者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。由于后者为 `RandomNumberGenerator` 的协议类型。所以它能够被赋值为任意遵循该协议的类型。

此外，使用构造器 (`init`) 来代替之前版本中的 `setup` 操作。构造器中含有一个名为 `generator`，类型为 `RandomNumberGenerator` 的形参，使得它可以接收任意遵循 `RandomNumberGenerator` 协议的类型。

`roll` 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。

如下所示，`LinearCongruentialGenerator` 的实例作为随机数生成器传入 `Dice` 的构造器

```

var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
    println("Random dice roll is \(d6.roll())")
}
// 骰子
//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

```

## 1.6 委托 (代理) 模式

委托是一种设计模式，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。

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的数据，而无需要知道外部数据源的类型。

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: 控制流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

```

class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dic = 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; borad[09] = +09; board[10] = +02
        borad[14] = -10; board[19] = -11; borad[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)
}
}

```

游戏的初始化设置 (setup) 被 SnakesAndLadders 类的构造器 (initializer) 实现。所有的游戏逻辑被转移到了 play 方法中。

注意：因为 delegate 并不是该游戏的必备条件，delegate 被定义为遵循 DiceGameDelegate 协议的可选属性

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，则委托调用优雅地失效。若 delegate 不为 nil，则委托方法被调用

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 协议的方法要求, 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。

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"
```

### 1.6.1 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, 协议等成员。详情请见扩展章节中查看。

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 游戏 :

extension SnakeAndLadders: 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"
```

### 1.6.2 通过延展补充协议声明

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：即时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

### 1.6.3 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

## 1.7 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 {
    // 协议内容
}
```

如下所示，PrettyTextRepresentable 协议继承了 TextRepresentable 协议

```
protocol PrettyTextRepresentable: TextRepresentable {
    func asPrettyText() -> String
}
```

遵循 ‘PrettyTextRepresentable’ 协议, 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:
//
```

### 1.7.1 协议合成

一个协议可由多个协议采用 protocol 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。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。

### 1.7.2 检验协议的一致性

使用 is 检验协议一致性, 使用 as 将协议类型向下转换 (downcast) 为的其他协议类型。检验与转换的语法和之前相同 (详情查看类型检查):

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 用以强制向下转换型。

```
@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。详情查看 [Using Swift with Cocoa and Objective-C](#)。

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

Circle 和 Country 都遵循了 HasArea 协议, 前者把 area 写为计算型属性 (computed property), 后者则把 area 写为存储型属性 (stored property)。

如下所示, Animal 类没有实现任何协议

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

Circle, Country, Animal 并没有一个相同的基类, 所以采用 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)
]
```

如下所示, 在迭代时检查 object 数组的元素是否遵循了 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
```

当数组中的元素遵循 `HasArea` 协议时，通过 `as?` 操作符将其可选绑定 (optional binding) 到 `objectWithArea` 常量上。

`objects` 数组中元素的类型并不会因为向下转型而改变，当它们被赋值给 `objectWithArea` 时只被视为 `HasArea` 类型，因此只有 `area` 属性能够被访问。

### 1.7.3 可选协议要求

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，详细内容在可选链章节中查看。

像 `someOptionalMethod?(someArgument)` 一样，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。

`Counter` 类使用 `CounterDataSource` 类型的外部数据源来提供增量值 (increment amount)，如下所示：

```
@objc protocol CounterDataSource {
    @optional func incrementForCount(count: Int) -> Int
    @optional var fixedIncrement: Int { get }
}
```

`CounterDataSource` 含有 `incrementForCount` 的可选方法和 `fixedIncrement` 的可选属性。

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。

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。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
```

本文部分原文来自于 <http://www.swiftguide.cn/> 翻译小组的译文，共同校对中。

## 参考文献