协议

cnswift.org/protocols

协议为方法、属性、以及其他特定的任务需求或功能定义蓝图。协议可被类、结构体、或枚举类型采纳以提供所需功能的具体实现。满足了协议中需求的任意类型都叫做*遵循*了该协议。

除了指定遵循类型必须实现的要求外，你可以扩展一个协议以实现其中的一些需求或实现一个符合类型的可以利用的附加功能。

协议的语法

定义协议的方式与类、结构体、枚举类型非常相似：

```
1  protocol SomeProtocol {
2      // protocol definition goes here
3  }
```

在自定义类型声明时，将协议名放在类型名的冒号之后来表示该类型采纳一个特定的协议。多个协议可以用逗号分开列出：

```
1  struct SomeStructure: FirstProtocol, AnotherProtocol {
2      // structure definition goes here
3  }
```

若一个类拥有父类，将这个父类名放在其采纳的协议名之前，并用逗号分隔：

```
1  class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
2      // class definition goes here
3  }
```

属性要求

协议可以要求所有遵循该协议的类型提供特定名字和类型的实例属性或类型属性。协议并不会具体说明属性是储存型属性还是计算型属性——它只具体要求属性有特定的名称和类型。协议同时要求一个属性必须明确是可读的或可读的*和*可写的。

若协议要求一个属性为可读和可写的，那么该属性要求不能用常量存储属性或只读计算属性来满足。若协议只要求属性为可读的，那么任何种类的属性都能满足这个要求，而且如果你的代码需要的话，该属性也可以是可写的。

属性要求定义为变量属性，在名称前面使用 var 关键字。可读写的属性使用 { get set } 来写在声明后面来明确，使用 { get } 来明确可读的属性。

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

在协议中定义类型属性时在前面添加 static 关键字。当类的实现使用 class 或 static 关键字前缀声明类型属性要求时，这个规则仍然适用：

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

这里是一个只有一个实例属性要求的协议：

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

上面 FullyNamed 协议要求遵循的类型提供一个完全符合的名字。这个协议并未指定遵循类型的其他任何性质——它只要求这个属性必须为其自身提供一个全名。协议申明了所有 FullyNamed 类型必须有一个可读实例属性 fullName，且为 String 类型。

这里是一个采纳并遵循 FullyNamed 协议的结构体的例子：

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

这个例子定义了一个名为 Person 的结构体，它表示一个有名字的人。它在其第一行定义中表明了它采纳 FullyNamed 协议作为它自身的一部分。

每一个 Person 的实例都有一个名为 fullName 的 String 类型储存属性。这符合了 FullyNamed 协议的单一要求，并且表示 Person 已经正确地遵循了该协议。（若协议的要求没有完全达标，Swift 在编译时会报错。）

这里是一个更加复杂的类，采纳并遵循了 FullyNamed 协议：

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName is "USS Enterprise"
```

这个类为一艘星舰实现了 fullName 计算型只读属性的要求。每一个 Starship 类的实例储存了一个不可选的 name 属性以及一个可选的 prefix 属性。当 prefix 值存在时，fullName 将 prefix 放在 name 之前以创建星舰的全名。

方法要求

协议可以要求采纳的类型实现指定的实例方法和类方法。这些方法作为协议定义的一部分，书写方式与正常实例和类方法的方式完全相同，但是不需要大括号和方法的主体。允许变量拥有参数，与正常的方法使用同样的规则。但在协议的定义中，方法参数不能定义默认值。

正如类型属性要求的那样，当协议中定义类型方法时，你总要在其之前添加 static 关键字。即使在类实现时，类型方法要求使用 class 或 static 作为关键字前缀，前面的规则仍然适用：

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

下面的例子定义了一个只有一个实例方法要求的协议：

```
protocol RandomNumberGenerator {
    func random() -> Double
}
```

这里 RandomNumberGenerator 协议要求所有采用该协议的类型都必须有一个实例方法 random，而且要返回一个 Double 的值，无论这个值叫什么。尽管协议没有明确定义，这里默认这个值在 0.0 到 1.0 （不包括）之间。

RandomNumberGenerator 协议并不为随机值的生成过程做任何定义，它只要求生成器提供一个生成随机数的标准过程。

这里有一个采用并遵循 RandomNumberGenerator 协议的类的实现。这个类实现了著名的 *linear congruential generator* 伪随机数发生器算法：

```
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).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.37464991998171"
print("And another one: \(generator.random())")
// Prints "And another one: 0.729023776863283"
```

异变方法要求

有时一个方法需要改变（或异变）其所属的实例。例如值类型的实例方法（即结构体或枚举），在方法的 func 关键字之前使用 mutating 关键字，来表示在该方法可以改变其所属的实例，以及该实例的所有属性。这一过程写在了在实例方法中修改值类型中。

若你定义了一个协议的实例方法需求，想要异变任何采用了该协议的类型实例，只需在协议里方法的定义当中使用 mutating 关键字。这允许结构体和枚举类型能采用相应协议并满足方法要求。

> 注意
> 如果你在协议中标记实例方法需求为 mutating，在为类实现该方法的时候不需要写 mutating 关键字。 mutating 关键字只在结构体和枚举类型中需要书写。

下面的例子定义了一个名为 Togglable 的协议，协议只定义了一个实例方法要求叫做 toggle。顾名思义，toggle() 方法将切换或转换任何遵循该协议的类型的状态，典型地，修改该类型的属性。

在Togglable协议的定义中， toggle() 方法使用 mutating 关键字标记，来表明该方法在调用时会改变遵循该协议的实例的状态：

```
protocol Togglable {
    mutating func toggle()
}
```

若使用结构体或枚举实现 Togglable 协议，这个结构体或枚举可以通过使用 mutating 标记这个 toggle() 方法，来保证该实现符合协议要求。

下面的例子定义了一个名为 OnOffSwitch 的枚举。这个枚举在两种状态间改变，即枚举成员 On 和 Off。该枚举的 toggle 实现使用了 mutating 关键字，以满足 Togglable 协议需求：

```
enum OnOffSwitch: Togglable {
    case off, on
    mutating func toggle() {
        switch self {
        case .off:
            self = .on
        case .on:
            self = .off
        }
    }
}
var lightSwitch = OnOffSwitch.off
lightSwitch.toggle()
// lightSwitch is now equal to .on
```

初始化器要求

协议可以要求遵循协议的类型实现指定的初始化器。和一般的初始化器一样，只用将初始化器写在协议的定义当中，只是不用写大括号也就是初始化器的实体：

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

协议初始化器要求的类实现

你可以通过实现指定初始化器或便捷初始化器来使遵循该协议的类满足协议的初始化器要求。在这两种情况下，你都必须使用 required 关键字修饰初始化器的实现：

```
class SomeClass: SomeProtocol {
    required init(someParameter: Int) {
        // initializer implementation goes here
    }
}
```

在遵循协议的类的所有子类中， required 修饰的使用保证了你为协议初始化器要求提供了一个明确的继承实现。

详见必要初始化器。

> 注意
>
> 由于 final 的类不会有子类，如果协议初始化器实现的类使用了 final 标记，你就不需要使用 required 来修饰了。因为这样的类不能被继承子类。详见阻止重写了解更多 final 修饰符的信息。

如果一个子类重写了父类指定的初始化器，并且遵循协议实现了初始化器要求，那么就要为这个初始化器的实现添加 required 和 override 两个修饰符：

```
protocol SomeProtocol {
    init()
}
class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}
class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from SomeSuperClass
    required override init() {
        // initializer implementation goes here
    }
}


```

可失败初始化器要求

如同可失败初始化器定义的一样，协议可以为遵循该协议的类型定义可失败的初始化器。

遵循协议的类型可以使用一个可失败的或不可失败的初始化器满足一个可失败的初始化器要求。不可失败初始化器要求可以使用一个不可失败初始化器或隐式展开的可失败初始化器满足。

将协议作为类型

实际上协议自身并不实现功能。不过你创建的任意协议都可以变为一个功能完备的类型在代码中使用。

由于它是一个类型，你可以在很多其他类型可以使用的地方使用协议，包括：

- 在函数、方法或者初始化器里作为形式参数类型或者返回类型；
- 作为常量、变量或者属性的类型；
- 作为数组、字典或者其他存储器的元素的类型。

> 注意
>
> 由于协议是类型，要开头大写（比如说 FullyNamed 和 RandomNumberGenerator ）来匹配 Swift 里其他类型名称格式（比如说 Int 、 String 还有 Double ）。

这里有一个把协议用作类型的例子：

```
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
    }
}
```

这个例子定义了一个叫做 Dice 的新类，它表示一个用于棋盘游戏的 *n* 面骰子。 Dice 实例有一个叫做 sides 的整数属性，它表示了骰子有多少个面，还有个叫做 generator 的属性，它提供了随机数的生成器来生成骰子的值。

generator 属性是 RandomNumberGenerator 类型。因此，你可以把它放到任何采纳了 RandomNumberGenerator 协议的类型的实例里。除了这个实例必须采纳 RandomNumberGenerator 协议以外，没有其他任何要求了。

Dice 也有一个初始化器，来设置初始状态。这个初始化器有一个形式参数叫做 generator ，它同样也是 RandomNumberGenerator 类型。你可以在初始化新的 Dice 实例的时候传入一个任意遵循这个协议的类型的值到这个形式参数里。

Dice 提供了一个形式参数方法， roll ，它返回一个介于 1 和骰子面数之间的整数值。这个方法调用生成器的 random() 方法来创建一个新的介于 0.0 和 1.0 之间的随机数，然后使用这个随机数来在正确的范围创建一个骰子的值。由于 generator 已知采纳了 RandomNumberGenerator ，它保证了会有 random() 方法以供调用。

这里是 Dice 类使用 LinearCongurentialGenerator 实例作为用于创建一个六面骰子的随机数生成器来创建一个六面骰子的过程：

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

委托

委托[1]是一个允许类或者结构体放手（或者说委托）它们自身的某些责任给另外类型实例的设计模式。这个设计模式通过定义一个封装了委托责任的协议来实现，比如遵循了协议的类型（所谓的委托）来保证提供被委托的功能。委托可以用来响应一个特定的行为，或者从外部资源取回数据而不需要了解资源具体的类型。

> 译注
>
> [1], *Delegation 委托，可能也以“代理”而为人熟知，这里我们选择译为“委托”是为了更好的理解避免混淆。*

下面的例子定义了两个协议以用于基于骰子的棋盘游戏：

```
1  protocol DiceGame {
2      var dice: Dice { get }
3      func play()
4  }
5  protocol DiceGameDelegate {
6      func gameDidStart(_ game: DiceGame)
7      func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
8      func gameDidEnd(_ game: DiceGame)
9  }
```

DiceGame 协议是一个给任何与骰子有关的游戏采纳的协议。 DiceGameDelegate 协议可以被任何追踪 DiceGame 进度的类型采纳。

这里有一个原本在控制流中介绍的*蛇与梯子*游戏的一个版本。这个版本采纳了协议以使用 Dice 实例来让它使用骰子；采用 DiceGame 协议；然后通知一个 DiceGameDelegate 关于进度的信息：

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[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)
    }
}
```

要找到*蛇与梯子*游戏的介绍，查看控制流章节的Break小节。

这个版本的游戏使用了叫做 SnakesAndLadders 类包装，它采纳了 DiceGame 协议。它提供了可读的 dice 属性和一个 play() 方法来遵循协议。（ dice 属性声明为常量属性是因为它不需要在初始化后再改变了，而且协议只需要它是可读的。）

*蛇与梯子*游戏棋盘设置都写在了类的 init() 初始化器中。所有的游戏逻辑都移动到了协议的 play 方法里，它使用了协议要求的 dice 属性来提供它的骰子值。

注意 delegate 属性被定义为*可选的 DiceGameDelegate*，是因为玩游戏并不是必须要有委托。由于它是一个可选类型， delegate 属性自动地初始化为 nil 。此后，游戏的实例化者可以选择给属性赋值一个合适的委托。

DiceGameDelegate 提供了三个追踪游戏进度的方法。这三个方法在游戏逻辑的 play() 方法中协作，并且在游戏开始时调用，新一局开始，或者游戏结束。

由于 delegate 属性是可选的 DiceGameDelegate， play() 方法在每次调用委托的时候都使用可选链。如果 delegate 属性是空的，这些委托调用会优雅地失败并且没有错误。如果 delegate 属性非空，委托的方法就被调用了，并且把 SnakesAndLadders 实例作为形式参数传入。

接下来的例子展示了叫做 DiceGameTracker 的类，它次那了 DiceGameDelegate 协议：

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

DiceGameTracker 实现了 DiceGameDelegate 要求的所有方法。它使用这些方法来对游戏开了多少局保持追踪。它在游戏开始的时候重置 numberOfTurns 属性为零，在每次新一轮游戏开始的时候增加，并且一旦游戏结束，打印出游戏一共开了多少轮。

上边展示的 gameDidStart(_:) 的实现使用了 game 形式参数来打印某些关于游戏的信息。game 形式参数是 DiceGame 类型，不是 SnakesAndLadders，所以 gameDidStart(_:) 只能访问和使用 DiceGame 协议实现的那部分方法和属性。总之，转换类型之后方法还是可以使用的。在这个例子中，它检查 game 在后台是否就是 SnakesAndLadders 实例，如果是，打印合适的信息。

gameDidStart(_:) 方法同样访问传入的 game 形式参数里的 dice 属性。由于 game 已经遵循 DiceGame 协议，这就保证了dice属性的存在，并且 gameDidStart(_:)方法能够访问和打印骰子的 sides 属性，无论玩的是什么类型的游戏。

这里是 DiceGameTracker 的运行结果：

```
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
```

在扩展里添加协议遵循

你可以扩展一个已经存在的类型来采纳和遵循一个新的协议，就算是你无法访问现有类型的源代码也行。扩展可以添加新的属性、方法和下标到已经存在的类型，并且因此允许你添加协议需要的任何需要。要了解更多关于扩展的信息，见扩展。

> 注意
>
> 类型已经存在的实例会在给它的类型扩展中添加遵循协议时自动地采纳和遵循这个协议。

举例来说，这个协议，叫做 TextRepresentable，可以被任何可以用文本表达的类型实现。这可能是它自身的描述，或者是它当前状态的文字版显示：

```
protocol TextRepresentable {
    var textualDescription: String { get }
}
```

先前的 Dice 类可以扩展来采纳和遵循 TextRepresentable ：

```
extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}
```

这个扩展使用了与 Dice 提供它原本实现完全相同的方式来采纳了新的协议。协议名写在类型的名字之后，用冒号隔开，并且在扩展的花括号里实现了所有协议的需要。

任何 Dice 实例现在都可以被视作 TextRepresentable ：

```
let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
print(d12.textualDescription)
// Prints "A 12-sided dice"
```

类似地， SnakesAndLadders 游戏类可以扩展来采纳和遵循 TextRepresentable 协议：

```
extension SnakesAndLadders: TextRepresentable {
    var textualDescription: String {
        return "A game of Snakes and Ladders with \(finalSquare) squares"
    }
}
print(game.textualDescription)
// Prints "A game of Snakes and Ladders with 25 squares"
```

有条件地遵循协议

泛型类型可能只在某些情况下满足一个协议的要求，比如当类型的泛型形式参数遵循对应协议时。你可以通过在扩展类型时列出限制让泛型类型有条件地遵循某协议。在你采纳协议的名字后面写泛型 where 分句。更多关于泛型 where 分句，见泛型Where分句。

下面的扩展让 Array 类型在存储遵循 TextRepresentable 协议的元素时遵循 TextRepresentable 协议。

```
extension Array: TextRepresentable where Element: TextRepresentable {
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joined(separator: ", ") + "]"
    }
}
let myDice = [d6, d12]
print(myDice.textualDescription)
// Prints "[A 6-sided dice, A 12-sided dice]"
```

使用扩展声明采纳协议

如果一个类型已经遵循了协议的所有需求，但是还没有声明它采纳了这个协议，你可以让通过一个空的扩展来让它采纳这个协议：

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

Hamster 实例现在可以用在任何 TextRepresentable 类型可用的地方了：

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
print(somethingTextRepresentable.textualDescription)
// Prints "A hamster named Simon"
```

使用综合实现来采纳协议

Swift 在大多数简单情况下能自动提供 Equatable 、 Hashable 以及 Comparable 协议遵循。使用这些合成实现意味着你不需要自己去使用大量重复代码实现这些协议需求。

Swift 为以下自定义类型提供了 Equatable 的综合实现：

- 只包含遵循 Equatable 协议的存储属性的结构体；
- 只关联遵循 Equatable 协议的类型的枚举；
- 没有关联类型的枚举。

要获取 == 的综合实现，只需要在原本声明的文件中声明其遵循 Equatable 协议，但不要手动实现 == 运算符即可。 Equatable 协议提供了默认的 != 实现。

下面的例子为三维位置向量 (x, y, z) 定义了一个 Vector3D 结构体，与 Vector2D 结构体类似。由于 x 、 y 和 z 属性都是 Equatable 类型， Vector3D 就可以直接使用综合实现中的等价运算符。

Swift为以下自定义类型提供了 Hashable 的综合实现：

- 只包含遵循 Hashable 协议的存储属性的结构体；
- 只关联遵循 Hashable 协议的类型的枚举；
- 没有关联类型的枚举。

要获取 hash(into:) 的综合实现，只需要在原本声明的文件中声明其遵循 Hashable 协议，但不要手动实现 hash(into:) 方法。

Swift为不包含原始值的枚举提供 Comparable 的综合实现。如果枚举拥有关联类型，这些类型必须都遵循 Comparable 协议。要获取 < 的综合实现，只需要在原本声明的文件中声明其遵循 Comparable 协议，但不要手动实现 <运算符。 Comparable 协议的默认实现 <= 、 > 和 >= 提供了其他比较运算符。

下面的例子定义了一个包含 beginners、intermediates以及 experts 情况的枚举 SkillLevel 。Experts 还额外使用数字来记录他们拥有的星星数量等级。

```
enum SkillLevel: Comparable {
    case beginner
    case intermediate
    case expert(stars: Int)
}
var levels = [SkillLevel.intermediate, SkillLevel.beginner,
              SkillLevel.expert(stars: 5), SkillLevel.expert(stars: 3)]
for level in levels.sorted() {
    print(level)
}
// Prints "beginner"
// Prints "intermediate"
// Prints "expert(stars: 3)"
// Prints "expert(stars: 5)"
```

协议类型的集合

协议可以用作储存在集合比如数组或者字典中的类型，如同在协议作为类型（此处应有链接）。这个例子创建了一个 TextRepresentable 实例的数组：

```
let things: [TextRepresentable] = [game, d12, simonTheHamster]
```

现在可以遍历数组中的元素了，并且打印每一个元素的文本化描述：

```
for thing in things {
    print(thing.textualDescription)
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

注意 thing 常量是 TextRepresentable 类型。它不是 Dice 类型，抑或 DiceGame 还是 Hamster，就算后台实际类型是它们之一。总之，由于它是 TextRepresentable，并且 TextRepresentable 唯一已知的信息就是包含了 textualDescription 属性，所以每次循环来访问 thing.textualDescription 是安全的。

协议继承

协议可以继承一个或者多个其他协议并且可以在它继承的基础之上添加更多要求。协议继承的语法与类继承的语法相似，只不过可以选择列出多个继承的协议，使用逗号分隔：

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

这里是一个继承了上边 TextRepresentable 协议的例子：

```
protocol PrettyTextRepresentable: TextRepresentable {
    var prettyTextualDescription: String { get }
}
```

这个例子定义了一个新的协议 PrettyTextRepresentable ，它继承自 TextRepresentable 。任何采用了 PrettyTextRepresentable 的类型都必须满足所有 TextRepresentable 强制的需要，另外还有 PrettyTextRepresentable 强制的要求。在这个例子中，PrettyTextRepresentable 添加了一个叫做 prettyTextualDescription 的可读属性，它返回一个 String 。

SnakesAndLadders 类可以通过扩展来采纳和遵循 PrettyTextRepresentable ：

```
extension SnakesAndLadders: PrettyTextRepresentable {
    var prettyTextualDescription: String {
        var output = textualDescription + ":\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
    }
}
```

这个扩展声明了它采纳了 PrettyTextRepresentable 协议并且为 SnakesAndLadders 类提供了 prettyTextualDescription 属性的实现。任何 PrettyTextRepresentable 必须同时是 TextRepresentable，并且 prettyTextualDescription 起始于访问 TextRepresentable 协议的

textualDescription 属性来开始输出字符串。它追加了一个冒号和一个换行符，并且使用这个作为友好文本输出的开始。它随后遍历棋盘数组，追加几何图形来表示每个方格的内容：

- 如果方格的值大于 0，它是梯子的底部，就表示为 ▲；
- 如果方格的值小于 0，它是蛇的头部，就表示为 ▼；
- 否则，方格的值为 0，是“自由”方格，表示为 ○。

现在 PrettyTextRepresentable 属性可以用来输出任何 SnakesAndLadders 实例的友好文本描述了：

```
print(game.prettyTextualDescription)
// A game of Snakes and Ladders with 25 squares:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

类专用的协议

通过添加 AnyObject 关键字到协议的继承列表，你就可以限制协议只能被类类型采纳（并且不是结构体或者枚举）。

```
protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

在上边的例子当中， SomeClassOnlyProtocol 只能被类类型采纳。如果在结构体或者枚举中尝试采纳 SomeClassOnlyProtocol 就会出发编译时错误。

> 注意在协议的要求假定或需要遵循的类型拥有引用语意的时候使用类专用的协议而不是值语意。要了解更多关于引用和值语意，见结构体和枚举是值类型和类是引用类型。

协议组合

要求一个类型一次遵循多个协议是很有用的。你可以使用协议组合来复合多个协议到一个要求里。协议组合行为就和你定义的临时局部协议一样拥有构成中所有协议的需求。协议组合不定义任何新的协议类型。

协议组合使用 SomeProtocol & AnotherProtocol 的形式。你可以列举任意数量的协议，用和符号连接（&），使用逗号分隔。除了协议列表，协议组合也能包含类类型，这允许你标明一个需要的父类。

这里是一个复合两个叫做 Named 和 Aged 的协议到函数形式参数里一个协议组合要求的例子：

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(to celebrator: Named & Aged) {
    print("Happy birthday, \(celebrator.name), you're \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(to: birthdayPerson)
// Prints "Happy birthday, Malcolm, you're 21!"
```

这个例子定义了一个叫做 Named 的协议，它只有一个叫做 name 的可读 String 属性要求。它同样定义了一个叫做 Aged 的协议，只有一个叫做 age 的 Int 属性要求。两个协议都被叫做 Person 的结构体采纳。

例子中同样定义了一个叫做 wishHappyBirthday(to:) 的函数，celebrator 形式参数的类型是 Named & Aged ，这意味着“任何同时遵循 Named 和 Aged 的协议。”它不关心具体是什么样的类型传入函数，只要它遵循这两个要求的协议即可。

然后例子中又创建了一个新的叫做 birthdayPerson 的 Person 实例并且把这个新的实例传入 wishHappyBirthday(to:) 函数。由于 Person 同时遵循两个协议，所以这是合法调用，并且 wishHappyBirthday(to:) 函数能够打印出生日祝福。

这里是一个包含了先前例子中 Named 协议以及一个 Location 类的例子：

```
class Location {
    var latitude: Double
    var longitude: Double
    init(latitude: Double, longitude: Double) {
        self.latitude = latitude
        self.longitude = longitude
    }
}
class City: Location, Named {
    var name: String
    init(name: String, latitude: Double, longitude: Double) {
        self.name = name
        super.init(latitude: latitude, longitude: longitude)
    }
}
func beginConcert(in location: Location & Named) {
    print("Hello, \(location.name)!")
}
let seattle = City(name: "Seattle", latitude: 47.6, longitude: -122.3)
beginConcert(in: seattle)
// Prints "Hello, Seattle!"

```

函数 beginConcert(in:) 接收一个 Location & Named 类型的形式参数，也就是说任何 Location 的子类且遵循 Named 协议的类型。具体到这里， City 同事满足两者需求。

如果你尝试过传 birthdayPerson 给 beginConcert(in:) 函数，这是非法的，由于 Person 不是 Location 的子类。同理，如果你创建了一个 Location 的子类但并不遵循 Named 协议，用这个类型调用 beginConcert(in:) 也是非法的。

协议遵循的检查

你可以使用类型转换中描述的 is 和 as 运算符来检查协议遵循，还能转换为特定的协议。检查和转换协议的语法与检查和转换类型是完全一样的：

- 如果实例遵循协议is运算符返回 true 否则返回 false ；
- as? 版本的向下转换运算符返回协议的可选项，如果实例不遵循这个协议的话值就是 nil ；
- as! 版本的向下转换运算符强制转换协议类型并且在失败是触发运行时错误。

这个例子定义了一个叫做 HasArea 的协议，只有一个叫做 area 的可读 Double 属性要求：

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

```

这里有两个类， Circle 和 Country ，这两个类都遵循 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 }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

Circle 类基于存储属性 radius 用计算属性实现了 area 属性要求。 Country 类则直接使用存储属性实现了 area 要求。这两个类都正确地遵循了 HasArea 协议。

这里是一个叫做 Animal 的类，它不遵循 HasArea 协议：

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

Circle、 Country 和 Animal 类并不基于相同的基类。不过它们都是类，所以它们三个类型的实例都可以用于初始化储存类型为 AnyObject 的数组：

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

objects 数组使用包含 Circle 两个单位半径的实例、 Country 以平方公里为单位英国面积实例、 Animal 有四条腿实例的数组字面量初始化。

objects 数组现在可以遍历了，而且数组中每一个对象都能检查是否遵循 HasArea 协议：

```
for object in objects {
    if let objectWithArea = object as? HasArea {
        print("Area is \(objectWithArea.area)")
    } else {
        print("Something that doesn't have an area")
    }
}
// Area is 12.5663708
// Area is 243610.0
// Something that doesn't have an area
```

当数组中的对象遵循 HasArea 协议， as? 运算符返回的可选项就通过可选绑定赋值到一个叫做 objectWithArea 的常量当中。 objectWithArea 已知类型为 HasArea，所以它的 area 属性可以通过类型安全的方式访问和打印。

注意使用的对象并没有通过转换过程而改变。他们仍然是 Circle 、 Country 和 Animal 。总之，在储存在 objectWithArea 常量中的那一刻，他们仅被所知为 HasArea ，所以只有 area 属性可以访问。

可选协议要求

你可以给协议定义*可选要求*，这些要求不需要强制遵循协议的类型实现。可选要求使用 optional 修饰符作为前缀放在协议的定义中。可选要求允许你的代码与 Objective-C 操作。协议和可选要求必须使用 @objc 标志标记。注意 @objc 协议只能被继承自 Objective-C 类或其他 @objc 类采纳。它们不能被结构体或者枚举采纳。

当你在可选要求中使用方法或属性是，它的类型自动变成可选项。比如说，一个 (Int) -> String 类型的方法会变成 ((Int) -> String)? 。注意是这个函数类型变成可选项，不是方法的返回值。

可选协议要求可以在可选链中调用，来说明要求没有被遵循协议的类型实现的概率。你可以通过在调用方法的时候在方法名后边写一个问号来检查它是否被实现，比如 someOptionalMethod?(someArgument) 。更多关于可选链的信息，见可选链。

下面的例子定义了一个叫做 Counter 的整数计数的类，它使用一个外部数据源来提供它的增量。这个数据源通过 CounterDataSource 协议定义，它有两个可选要求：

```
1  @objc protocol CounterDataSource {
2      @objc optional func increment(forCount count: Int) -> Int
3      @objc optional var fixedIncrement: Int { get }
4  }
```

CounterDataSource 协议定义了一个叫做 increment(forCount:) 的可选方法要求以及一个叫做 fixedIncrement 的可选属性要求。这些要求给 Counter 实例定义了两个不同的提供合适增量的方法。

> 注意
>
> 严格来讲，你可以写一个遵循 CounterDataSource 的自定义类而不实现*任何*协议要求。反正它们都是可选的。尽管技术上来讲是可以的，但这样的话就不能做一个好的数据源了。

```
class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.increment?(forCount: count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
        }
    }
}
```

Counter 类在一个叫做 count 的变量属性里储存当前值。 Counter 类同样定义了一个叫做 increment 的方法，它在每次被调用的时候增加 count 属性。

increment() 方法首先通过查找自身数据源的 increment(forCount:) 的实现来尝试获取增量。 increment() 方法使用可选链来尝试调用 increment(forCount:)，同时传入当前 count 值作为方法的唯一实际参数。

注意这里有两层可选链。首先， dataSource 有可能是 nil，所以 dataSource 名字后边有一个问号以表示只有 dataSource 不是 nil 的时候才能调用 incrementForCount(forCount:) 。其次，就算 dataSource 确实存在，也没有人能保证它实现了 incrementForCount(forCount:)，因为它是可选要求。所以， incrementForCount(forCount:) 没有被实现的可能也被可选链处理。 incrementForCount(forCount:) 的调用只发生在 incrementForCount(forCount:) 存在的情况下——也就是说，不是 nil 。这就是为什么 incrementForCount(forCount:) 也在名字后边写一个问号。

由于对 incrementForCount(forCount:) 的调用可以两个理由中的任何一个而失败，调用返回一个*可选的 Int* 值。这就算 incrementForCount(forCount:) 在 CounterDataSource 中定义为返回一个非可选的 Int 值也生效。尽管这里有两个可选链运算，但结果仍然封装在一个可选项中。要了解更多关于多个可选链运算的信息，见 链的多层连接。

在 increment(forCount:) 调用之后，返回的可选的 Int 使用可选绑定展开到一个叫做 amount 的常量中。如果可选 Int 包含值——也就是说，如果委托和方法都存在，并且方法返回值——展开的 amount 添加到 count 存储属性，增加完成。

如果不能从 increment(forCount:) 方法取回值——无论是由于 dataSource 为空还是由于数据源没有实现 increment(forCount:) ——那么 increment() 方法就尝试从数据源的 fixedIncrement 属性取回值。 fixedIncrement 属性同样是一个可选要求，所以值是一个可选的 Int 值，就算 fixedIncrement 在 CounterDataSource 协议的定义中被定义为非可选 Int 属性。

这里是 CounterDataSource 的简单实现，数据源在每次查询时返回固定值 3 .它通过实现可选 fixedIncrement 属性要求来实现这一点：

```
class ThreeSource: NSObject, CounterDataSource {
    let fixedIncrement = 3
}
```

你可以使用 ThreeSource 的实例作为新 Counter 实例的数据源：

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    print(counter.count)
}
// 3
// 6
// 9
// 12
```

下边的代码创建了一个新的 Counter 实例；设置它的数据源为新的 ThreeSource 实例；并且调用计数器的 increment() 方法四次。按照预期，计数器的 count 属性在每次 increment() 调用是增加三。

这里有一个更加复杂一点的 TowardsZeroSource ，它使 Counter 实例依照它当前的 count 值往上或往下朝着零计数：

```
@objc class TowardsZeroSource: NSObject, CounterDataSource {
    func increment(forCount count: Int) -> Int {
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```

TowardsZeroSource 类实现了 CounterDataSource 协议的可选 increment(forCount:) 方法并且使用 count 实际参数来计算改朝哪个方向计数。如果 count 已经是零，方法返回 0 来表示无需继续计数。

你可以使用 TowardsZeroSource 给现存的 Counter 实例来从 -4 到零。一旦计数器到零，就不会再变化：

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    print(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

协议扩展

协议可以通过扩展来提供方法和属性的实现以遵循类型。这就允许你在协议自身定义行为，而不是在每一个遵循或者在全局函数里定义。比如说，RandomNumberGenerator 协议可以扩展来提供 randomBool() 方法，它使用要求的 random() 方法来返回随机的 Bool 值：

```
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}
```

通过给协议创建扩展，所有的遵循类型自动获得这个方法的实现而不需要任何额外的修改。

```
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.37464991998171"
print("And here's a random Boolean: \(generator.randomBool())")
// Prints "And here's a random Boolean: true"
```

提供默认实现

你可以使用协议扩展来给协议的任意方法或者计算属性要求提供默认实现。如果遵循类型给这个协议的要求提供了它自己的实现，那么它就会替代扩展中提供的默认实现。

> 注意
>
> 通过扩展给协议要求提供默认实现与可选协议要求的区别是明确的。尽管遵循协议都不需要提供它们自己的实现。有默认实现的要求不需要使用可选链就能调用。

举例来说，继承自 TextRepresentable 的 PrettyTextRepresentable 协议可以给它要求的 prettyTextualDescription 属性提供一个默认实现来简单的返回访问 textualDescription 属性的结果：

```
extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```

给协议扩展添加限制

当你定义一个协议扩展，你可以明确遵循类型必须在扩展的方法和属性可用之前满足的限制。如同 Where 分句（此处应有链接）中描述的那样，在扩展协议名字后边使用 where 分句来写这些限制。比如说，你可以给 Collection 定义一个扩展来应用于任意元素遵循上面 TextRepresentable 协议的集合。

```
extension Collection where Iterator.Element: TextRepresentable {
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joined(separator: ", ") + "]"
    }
}
```

textualDescription 属性返回整个集合写在花括号里通过用逗号组合集合中每个元素的文本化表示的文本化描述。

考虑之前的 Hamster 结构体，它遵循 TextRepresentable 协议，Hamster 值的数组：

```
let murrayTheHamster = Hamster(name: "Murray")
let morganTheHamster = Hamster(name: "Morgan")
let mauriceTheHamster = Hamster(name: "Maurice")
let hamsters = [murrayTheHamster, morganTheHamster, mauriceTheHamster]
```

由于 Array 遵循 Collection 并且数组的元素遵循 TextRepresentable 协议，数组可以使用 textualDescription 属性来获取它内容的文本化表示：

```
print(hamsters.textualDescription)
// Prints "[A hamster named Murray, A hamster named Morgan, A hamster named Maurice]"
```

> 注意
>
> 如果遵循类型满足了为相同方法或者属性提供实现的多限制扩展的要求，Swift 会使用最匹配限制的实现。