协议

甲*协议*定义的该适合特定任务或片的功能的方法，属性和其他要求的蓝图。该协议然后可以*采用*由一个类，结构，或枚举，以提供实际实施方案的这些要求。据说满足协议要求的任何类型都*符合*该协议。

除了指定符合类型必须实现的要求外，还可以扩展协议以实现其中一些要求或实现符合类型可以利用的其他功能。

协议语法

您以与类，结构和枚举类似的方式定义协议：

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

自定义类型声明他们采用特定协议，方法是将协议名称放在类型名称后面，并用冒号分隔，作为其定义的一部分。可以列出多个协议，并用逗号分隔：

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

如果一个类有一个超类，那么在它所采用的任何协议之前列出超类的名称，后跟一个逗号：

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

物业要求

一个协议可以要求任何一致性类型提供一个实例属性或者具有特定名称和类型的类型属性。该协议没有指定属性是否应该是存储属性或计算属性 - 它只指定所需的属性名称和类型。该协议还规定每个属性是否必须是可获取的或可获取*和*可设置的。

如果一个协议要求一个属性是可获取和可设置的，那么该属性要求不能由一个常量存储属性或一个只读计算属性来满足。如果协议只需要一个属性是可以获取的，那么这个需求可以通过任何类型的属性来满足，如果这个属性对你自己的代码有用的话，这个属性也是可以设置的。

属性需求总是被声明为变量属性，并以`var`关键字为前缀。Gettable和可设置的属性通过`{ get set }`在它们的类型声明之后写入来指示，并且可写属性通过写入来指示`{ get }`。

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

`static`在协议中定义关键字时， 始终将关键字的类型属性需求添加到前缀中。即使类型属性要求可以用类`class`或者`static`关键字作为前缀，但是该类仍然适用于类：

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

以下是一个具有单个实例属性要求的协议示例：

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

该FullyNamed协议要求符合类型以提供完全合格的名称。该协议没有指定任何有关符合类型的性质 - 它仅指定该类型必须能够为其自身提供全名。该协议规定任何FullyNamed类型都必须有一个名为gettable的实例属性fullName，它是

以下是采用并符合FullyNamed协议的简单结构示例：

```
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称为的结构Person，它表示一个特定的具名人员。它表示它将该FullyNamed议定书作为其定义第一行的一部分。

每个实例Person都有一个名为的存储属性fullName，它是类型的String。这符合FullyNamed协议的单个要求，并且意味着Person已经正确地符合协议。（如果协议要求未满足，Swift在编译时报告错误。）

这是一个更复杂的类，它也采用并符合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"
```

该类将fullName属性需求作为星舰的计算只读属性实现。每个Starship类实例都存储一个必需的name和一个可选的prefix。该fullName属性使用该prefix值（如果存在），并将其预先设置name为为星舰创建完整名称的开头。

方法要求

协议可以要求特定的实例方法和类型方法通过符合类型来实现。这些方法是作为协议定义的一部分编写的，与普通实例和类型方法完全相同，但没有大括号或方法体。允许变量参数，遵循与正常方法相同的规则。但是，不能为协议定义中的方法参数指定默认值。

与类型属性要求一样，static当在协议中定义关键字时，始终将类型方法要求作为前缀。即使类型方法需求在由类实现时带有classor static关键字前缀，情况也是如此：

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

以下示例使用单个实例方法要求定义一个协议：

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

这个协议RandomNumberGenerator要求任何一致性类型都有一个实例方法调用random，Double它在调用时会返回一个值。虽然它没有被指定为协议的一部分，但是假定这个值将是一个从0.0（但不包括）开始的数字1.0。

该RandomNumberGenerator协议并没有就如何每个随机数将任何假设产生的，它只是需要发电机提供一种标准方法来生成一个新的随机数。

这是一个采用并符合RandomNumberGenerator协议的类的实现。这个类实现了一个称为*线性同余发生器*的伪随机数生成器算法：

```
class L
    var
    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"
```

变异方法要求

有时一种方法需要修改（或*改变*）它所属的实例。例如，对于值类型（即结构和枚举）mutating的方法，您可以将关键字放在方法的func关键字之前，以指示该方法可以修改它所属的实例以及该实例的任何属性。此过程在“ 从实例方法中修改值类型”中介绍。

如果您定义的协议实例方法要求旨在改变采用协议的任何类型的实例，请将mutating关键字标记为协议定义的一部分。这使得结构和枚举可以采用协议并满足该方法的要求。

> 注意
>
> 如果将协议实例方法要求标记为mutating，则mutating在为该类编写该方法的实现时，不需要编写关键字。该mutating关键字仅用于结构和枚举。

下面的例子定义了一个调用的协议Togglable，它定义了一个调用的单个实例方法需求toggle。顾名思义，该toggle()方法旨在切换或反转任何符合类型的状态，通常通过修改该类型的属性。

该toggle()方法使用mutating关键字作为Togglable协议定义的一部分进行标记，以指示该方法在调用时会改变符合实例的状态：

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

如果您Togglable为结构或枚举实现协议，则该结构或枚举可以通过提供toggle()也被标记为的方法的实现来符合协议mutating。

下面的例子定义了一个枚举OnOffSwitch。此枚举在枚举案例on和枚举所指示的两个状态之间切换off。枚举的toggle实现标记为mutating符合Togglable协议的要求：

```
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
```

初始化程序要求

协议可能需要通过符合类型来实现特定的初始化程序。作为协议定义的一部分，这些初始化方法与正常初始化方法完全相同，但没有大括号或初始化体：

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

协议初始化器要求的类实现

您可以在符合类上实现协议初始值设定项要求作为指定初始值设定项或便利初始值设定项。在这两种情况下，您都必须使用`required`修饰符标记初始化程序实现：

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

`required`修饰符 的使用可以确保您为合格类的所有子类提供初始化器要求的显式或继承实现，以使它们也符合协议。

有关所需初始化程序的更多信息，请参阅必需初始化程序。

> 注意
>
> 您不需要使用带`required`修饰符标记的类上的修饰符标记协议初始化程序实现`final`，因为最终的类不能进行子类化。有关`final`修饰符的更多信息，请参阅防止覆盖。

如果一个子类覆盖了一个超类的指定初始值设定项，并且还从一个协议中实现了一个匹配的初始值设定项要求，那么使用`required`和`override`修饰符标记初始值设定项实现：

```
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
    }
}
```

Failable初始化器要求

协议可以定义符合类型的可分解的初始化器要求，如Failable Initializers中所定义。

可靠的初始化器要求可以通过符合类型的failable或nonfailable初始化器来满足。一个不可破解的初始化器需求可以由一个不可破的初始化器或一个隐式解包的可分解的初始化器来满足。

作为类型的协议

协议本身并不实际实现任何功能。尽管如此，您创建的任何协议都将成为代码中使用的完整类型。

因为它是一种类型，所以可以在允许其他类型的许多地方使用协议，包括：

- 作为函数，方法或初始值设定项中的参数类型或返回类型
- 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项目类型

> 注意
>
> 由于协议的类型，开始他们的名称以大写字母（如`FullyNamed`和`RandomNumberGenerator`），以配合其他类型的雨燕的名称（如`Int`，`String`和`Double`）。

以下是一个用作类型的协议示例：

```
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的新类`Dice`，它代表了在棋盘游戏中使用的*n方向*骰子。`Dice`实例具有一个调用的整数属性`sides`，它表示它们有多少个边，以及一个叫做的属性`generator`，它提供了一个随机数生成器，从中创建骰子滚动值。

该`generator`属性是类型`RandomNumberGenerator`。因此，您可以将其设置为采用该协议的*任何*类型的实例`RandomNumberGenerator`。除了实例必须采用`RandomNumberGenerator`协议之外，您分配给此属性的实例不需要其他任何东西。

`Dice`也有一个初始化器，来设置它的初始状态。这个初始化器有一个名为的参数`generator`，它也是类型的`RandomNumberGenerator`。初始化新`Dice`实例时，可以将任何符合类型的值传递给此参数。

`Dice`提供了一个实例方法，`roll`它返回`1`和骰子边数之间的整数值。此方法调用生成器的`random()`方法在`0.0`和之间创建一个新的随机数`1.0`，并使用此随机数在正确的范围内创建骰子滚动值。因为`generator`已知采用`RandomNumberGenerator`，所以保证有一个`random()`方法可以调用。

以下是如何使用这个`Dice`类创建一个具有`LinearCongruentialGenerator`实例作为随机数生成器的六面骰子：

```
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
```

代表团

*委托*是一种设计模式，它使类或结构能够*将*其某些职责交给（或*委托*）其他类型的实例。这种设计模式是通过定义一个封装委托职责的协议来实现的，这样一个符合类型（称为委托）就能保证提供已委派的功能。可以使

用委托来响应特定操作，或者从外部源检索数据，而无需知道该源的基本类型。

下面的例子定义了两种用于基于骰子的棋盘游戏的协议：

```
protocc
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate: AnyObject {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
```

该DiceGame协议是一个可以被任何涉及骰子的游戏所采用的协议。

该DiceGameDelegate协议可以被用来跟踪一个进程DiceGame。为了防止强引用周期，代表被声明为弱引用。有关弱引用的信息，请参阅类实例之间的强参考循环。将协议标记为类只允许SnakesAndLadders本章后面的类声明它的委托必须使用弱引用。仅在类中使用的协议标记为它的继承，AnyObject如在纯类协议中讨论的那样。

以下是最初在Control Flow中引入的一个版本的*Snakes and Ladders*游戏。这个版本适合于使用其骰子卷的实例; 采用协议; 并通知其进展情况： DiceDiceGameDiceGameDelegate

```
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
    }
    weak 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)
    }
}
```

有关*Snakes and Ladders*游戏的描述，请参阅Break。

这个版本的游戏被封装为一个叫做的类SnakesAndLadders，它采用了DiceGame协议。它提供了一个gettable dice属性和一个play()方法以符合协议。（该dice属性被声明为一个常量属性，因为它不需要在初始化后更改，并且该协议只需要它是可以获取的。）

该*蛇和梯子*游戏板的设置采取类的内进行init()初始化。所有的游戏逻辑都被移动到协议的play方法中，该方法使用协议的所需dice属性来提供其骰子滚动值。

请注意，该delegate属性被定义为可*选项* DiceGameDelegate，因为不需要委托来玩游戏。由于它是可选类型，因此该delegate属性会自动设置为初始值nil。此后，游戏实例化器可以选择将该属性设置为合适的代理。由于

DiceGameDelegate协议仅为类，因此您可以声明委托weak以防止引用循环。

DiceGameDelegate提供了三种跟踪游戏进度的方法。这三种方法已被纳入上述play()方法中的游戏逻辑，并在新游戏开始，新游戏开始或游戏结束时被调用。

因为该delegate属性是可选的DiceGameDelegate，所以play()每次它在委托上调用方法时，该方法都会使用可选的链接。如果该delegate属性为零，则这些委托调用会优雅而无错地失败。如果该delegate属性为非零，则调用委托方法，并将该SnakesAndLadders实例作为参数传递。

下一个例子显示了一个叫做的类DiceGameTracker，它采用了这个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实现所需的所有三种方法DiceGameDelegate。它使用这些方法来跟踪游戏进行的转数。它numberOfTurns在游戏开始时将一个属性重置为零，每次新一轮开始时将其增加一次，并在游戏结束后打印总转数。

gameDidStart(_:)上面显示 的实现使用该game参数来打印关于即将播放的游戏的一些介绍性信息。该game参数有一个类型DiceGame，不是SnakesAndLadders，因此gameDidStart(_:)只能访问和使用作为DiceGame协议一部分实现的方法和属性。但是，该方法仍然能够使用类型转换来查询基础实例的类型。在这个例子中，它检查game实际上是否是SnakesAndLadders幕后实例，如果是，则打印适当的消息。

该gameDidStart(_:)方法还访问dice传递的game参数的属性。因为game已知符合DiceGame协议，所以它保证有一个dice属性，所以该gameDidStart(_:)方法能够访问和打印骰子的sides属性，而不管正在玩什么类型的游戏。

以下是DiceGameTracker看起来如何行动：

```
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
```

添加扩展协议一致性

即使您无法访问现有类型的源代码，也可以扩展现有类型以采用并符合新协议。扩展可以向现有类型添加新的属性，方法和下标，因此可以添加协议可能要求的任何要求。有关扩展的更多信息，请参阅扩展。

> 注意
>
> 当一致性被添加到扩展中的实例类型时，现有类型的实例会自动采用并符合协议。

例如，这个被调用的协议TextRepresentable可以通过任何可以被表示为文本的方式来实现。这可能是对其自身的描述，或者是

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

Dice上面 的课程可以扩展为采用并符合TextRepresentable：

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

此扩展采用新协议，方式与Dice原始实现中提供的方式完全相同。协议名称在类型名称之后提供，以冒号分隔，协议的所有要求的实现在扩展的大括号内提供。

Dice现在可以将 任何实例视为TextRepresentable：

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

同样，SnakesAndLadders游戏类可以扩展为采用并符合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"
```

有条件地符合议定书

通用类型只能在某些条件下才能满足协议的要求，例如类型的通用参数符合协议。在扩展类型时，可以通过列出约束来使通用类型有条件地符合协议。通过编写通用where子句，将这些约束条件写在所采用协议的名称后面。有关泛型where子句的更多信息，请参见泛型Where子句。

下面的扩展使得Array实例符合TextRepresentable协议，只要它们存储符合的类型的元素TextRepresentable。

```
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]"
```

使用扩展声明协议采用

如果一个类型已经符合协议的所有要求，但尚未声明它采用该协议，则可以使其采用具有空扩展名的协议：

```
struct Hamster {
    var name: String
```

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

Hamster现在可以在任何TextRepresentable需要的类型中使用 实例：

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

> 注意
> 只有满足其要求，类型才会自动采用协议。他们必须始终明确宣布他们通过协议。

协议类型的集合

协议可以用作要存储在集合中的类型，如数组或字典，如协议类型中所述。这个例子创建了一系列的TextRepresentable东西：

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

现在可以遍历数组中的项目，并打印每个项目的文本描述：

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

请注意，thing常数是类型的TextRepresentable。它不是类型的Dice，或者DiceGame，Hamster即使幕后的实际实例属于这些类型之一。尽管如此，因为它是类型的TextRepresentable，并且任何TextRepresentable已知的textualDescription属性都有，所以thing.textualDescription每次通过循环访问都是安全的。

协议继承

一个协议可以*继承*一个或多个其他协议，并且可以在其继承的需求之上添加更多的需求。协议继承的语法与类继承的语法相似，但可以选择列出多个继承协议（用逗号分隔）：

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

以下是一个继承上述TextRepresentable协议的协议示例：

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

这个例子定义了一个新的协议PrettyTextRepresentable，它从中继承TextRepresentable。任何采取的措施都PrettyTextRepresentable必须满足所强制执行的所有要求TextRepresentable，*以及*强制实施的附加要求PrettyTextRepresentable。在这个例子中，PrettyTextRepresentable增加了一个单独的需求来提供一个叫做prettyTextualDescription返回a 的gettable属性String。

该SnakesAndLadders级可扩展到通过并符合PrettyTextRepresentable：

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

            switch board[index] {
            case let ladder where ladder > 0:
                output += "▲ "
            case let snake where snake < 0:
                output += "▼ "
            default:
                output += "○ "
            }
        }
        return output
    }
}
```

该扩展指出它采用PrettyTextRepresentable协议并提供prettyTextualDescription该SnakesAndLadders类型属性的实现。任何东西也PrettyTextRepresentable必须是TextRepresentable这样的，所以prettyTextualDescription通过textualDescription从TextRepresentable协议访问属性开始执行输出字符串。它追加冒号和换行符，并将其用作其漂亮文本表示的开始。然后它遍历棋盘方格阵列，并附加一个几何形状来表示每个方格的内容：

- 如果方格的值大于0，则它是梯子的基础，并由其表示▲。
- 如果方格的值小于0，则它是蛇的头部，并由其表示▼。
- 否则，广场的价值是0，这是一个“自由”的平方，由○。

该prettyTextualDescription属性现在可以用来打印任何SnakesAndLadders实例的漂亮文本描述：

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

仅限于类的协议

通过将AnyObject协议添加到协议的继承列表中，您可以将协议采用限制为类类型（而不是结构或枚举）。

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

在上面的例子中，SomeClassOnlyProtocol只能通过类类型来采用。编写尝试采用的结构或枚举定义是编译时错误SomeClassOnlyProtocol。

> 注意
>
> 当该协议的需求定义的行为假设或要求符合类型具有引用语义而不是值语义时，请使用仅类别协议。有关引用和值语义的更多信息，请参阅结构和枚举是值类型和类是引用类型。

协议组成

要求类型同时符合多个协议会很有用。您可以使用*协议组合*将多个协议组合成单个需求。协议组合的行为就好像您定义了一个临时本地协议，该协议具有组合中所有协议的组合要求。协议组合不定义任何新的协议类型。

协议组合具有这种形式SomeProtocol & AnotherProtocol。您可以根据需要列出尽可能多的协议，并用＆符号（&）分隔它们。除协议列表之外，协议组合还可以包含一个类类型，您可以使用它来指定所需的超类。

以下是一个将两个协议调用Named并Aged组合成一个函数参数的单个协议组合需求的示例：

```
protocol Named {
    var name: String { get }
}
protocc
    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!"
```

在这个例子中，该Named协议对于String所谓的gettable 属性有一个单独的要求name。该Aged协议对于Int所谓的gettable 属性有一个单独的要求age。两个协议都被一个叫做结构的结构采用Person。

该示例还定义了一个wishHappyBirthday(to:)函数。celebrator参数的类型是Named & Aged，这意味着“符合Named和Aged协议的任何类型”。只要它符合所需的两种协议，传递给函数的是哪一种特定的类型并不重要。

然后该示例创建一个新Person实例birthdayPerson，并将该新实例传递给该wishHappyBirthday(to:)函数。由于Person符合两种协议，此调用是有效的，并且该wishHappyBirthday(to:)函数可以打印其生日问候语。

下面是一个将Named前一个示例中的协议与一个Location类相结合的示例：

```
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!"
```

该beginConcert(in:)函数接受一个类型参数Location & Named，这意味着“任何类型都是协议的子类，Location并且符合Named协议。”在这种情况下，City满足这两个要求。

传递birthdayPerson给beginConcert(in:)函数是无效的，因为Person它不是的子类Location。同样，如果您创建了Location不符合Named协议的子类，则beginConcert(in:)使用该类型的实例进行调用也是无效的。

检查协议一致性

您可以使用类型转换中描述的is和as运算符来检查协议一致性，并转换为特定的协议。检查并转换为协议遵循与检查和转换为类型完全相同的语法：

- 该is运算符返回true如果一个实例遵循的协议，并返回false，如果它不。

- as?downcast运算符 的版本返回协议类型的可选值，nil如果实例不符合该协议，则此值为。
- as!downcast操作符 的版本强制downcast转换为协议类型，如果downcast不成功，则触发运行时错误。

这个例子定义了

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

这里有两个类，Circle并且Country，这两者的符合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 }
}
```

的Circle类实现area性能要求作为一个计算的属性的基础上，所存储的radius属性。本Country类实现了area直接需求的存储性能。两个类都正确地符合HasArea协议。

这是一个叫做的类Animal，它不符合HasArea协议：

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

的Circle，Country而Animal类没有共享的基类。尽管如此，它们都是类，所有这三种类型的实例都可以用来初始化一个存储类型值的数组AnyObject：

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

该objects阵列被初始化为常量，其中包含的阵列Circle具有2个单位的半径实例; 一个Country以平方公里的英国地表面积初始化的例子; 和Animal四条腿的例子。

的objects阵列现在可以重复，并且阵列中的每个对象可以被检查，看它是否符合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
```

只要数组中的某个对象符合该HasArea协议，as?运算符返回的可选值就会使用可选的绑定解压到一个名为常量的常量中objectWithArea。该objectWithArea常数是已知的类型HasArea，因此其area属性可以以类型安全的方式访问和打印。

请注意，投射过程不会更改底层对象。他们继续是一个Circle，一个Country和一个Animal。然而，在它们存储在objectWithArea常量中的时候，它们只知道是类型的HasArea，所以只有它们的area属性可以被访问。

可选协议要求

您可以为协议定
分，可选要求由修饰符作为前缀。可选的需求是可用的，以使您可以编写与Objective-C互操作的代码。协议和可选要求都必须用@objc属性标记。请注意，@objc协议只能由继承自Objective-C类或其他@objc类的类采用。它们不能被结构或枚举所采用。

当您在可选需求中使用方法或属性时，其类型自动成为可选项。例如，类型的方法(Int) -> String变为((Int) -> String)?。请注意，整个函数类型都包含在可选项中，而不是方法的返回值。

一个可选的协议需求可以通过可选的链接来调用，以说明需求没有被一个符合协议的类型实现的可能性。通过在调用方法名称后面写一个问号来检查可选方法的实现，例如someOptionalMethod?(someArgument)。有关可选链接的信息，请参阅可选链接。

以下示例定义了一个称为的整数计数类Counter，它使用外部数据源来提供其增量。该数据源由CounterDataSource协议定义，该协议有两个可选要求：

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

该CounterDataSource协议定义了一个调用的可选方法需求increment(forCount:)和一个可选属性需求fixedIncrement。这些要求为数据源定义了两种不同的方式来为Counter实例提供适当的增量。

> 注意
>
> 严格地说，您可以编写一个自定义的类，以符合CounterDataSource不需要执行*任何*协议要求。毕竟，它们都是可选的。尽管技术上允许，但这不会构成非常好的数据源。

的Counter类，下面定义，具有可选的dataSource类型的属性CounterDataSource?：

```
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
        }
    }
}
```

在Counter类存储在一个名为变量属性的当前值count。的Counter类也定义了一个称为方法increment，其中递增count每次方法调用时属性。

该increment()方法首先尝试通过increment(forCount:)在其数据源上查找该方法的实现来检索增量量。该increment()方法使用可选的链接来尝试调用increment(forCount:)，并将当前count值作为方法的单个参数传递。

在本页

请注意，这里有*两个*级别的可选链接。首先，这可能是dataSource可能的nil，因此dataSource在其名称后面有一个问号，表示increment(forCount:)只有在dataSource没有问号时才应该调用它nil。其次，即使dataSource *确实存*在，也无法保证其实施increment(forCount:)，因为这是可选要求。在这里，increment(forCount:)可能没有实现的可能性也由可选链处理。increment(forCount:)只有在increment(forCount:)存在的情况下才会发生呼叫- 即，如果不存在nil。这就是为什么increment(forCount:)在它的名字后面写上一个问号。

由于increment(forCount:)以上两种原因之一致使呼叫可能失败，因此呼叫将返回*可选* Int值。即使increment(forCount:)被定义为Int在定义中返回非选项值，情况也是如此CounterDataSource。即使有两个可选的链接操作，一个接一个，结果仍然包装在一个可选的。有关使用多个可选链接操作的更多信息，请参阅链接多个链接级别。

打完电话后increment(forCount:)，可选的Int，它返回的是解开到一个名为常量amount，使用可选的绑定。如果可选Int包含一个值（即，如果委托和方法都存在，并且该方法返回值），则将解包的amount内容添加到存

储的count属性中，并且增量完成。

如果*无法*从increment(forCount:)方法中检索值（无论是因为dataSourcenil，还是因为数据源未实现），increment(forCount:)那么该increment()方法会尝试从数据源的fixedIncrement属性中检索值。该fixedIncrement
可选Int属性作为CounterDataSource协议定义的一部分。

这是一个简单的CounterDataSource实现，数据源在3每次查询时返回一个常量值。它通过实现可选的fixedIncrement属性需求来实现这一点：

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

您可以使用一个实例ThreeSource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
```

上面的代码创建一个新的Counter实例; 将其数据源设置为新ThreeSource实例; 并调用计数器的increment()方法四次。正如所料，该柜台的count财产每次增加三次increment()。

这是一个更复杂的数据源TowardsZeroSource，它使Counter实例从当前count值向上或向下趋近于零：

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

的TowardsZeroSource类实现可选的increment(forCount:)从方法CounterDataSource协议并使用该count参数值，以计算出在计数的方向。如果count已经是零，则该方法返回0到表示没有进一步的计数应该发生。

您可以使用TowardsZeroSource现有Counter实例的实例从-40 开始计数。一旦计数器达到零，不再进行计数：

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

协议扩展

可以扩展协议以向符合类型提供方法，初始化程序，下标和计算属性实现。这允许您定义协议本身的行为，而不是每个类型的单独一致性或全局函数。

例如，该RandomNumberGenerator协议可以扩展为提供一种randomBool()方法。该方法使用所需random()方法的结果返回一个

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

通过在协议上创建扩展，所有符合类型自动获得此方法实现，无需任何额外的修改。

```
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"
```

协议扩展可以将实现添加到符合类型，但不能使协议扩展或从另一个协议继承。协议继承总是在协议声明本身中指定的。

提供默认实现

您可以使用协议扩展来为该协议的任何方法或计算属性要求提供默认实现。如果一致性类型提供了自己的必需方法或属性的实现，则将使用该实现来代替扩展提供的实现。

> 注意
>
> 通过扩展提供的默认实现的协议要求与可选的协议要求不同。尽管符合类型不必提供它们自己的实现，但可以在没有可选链接的情况下调用具有默认实现的需求。

例如，PrettyTextRepresentable继承TextRepresentable协议的协议可以提供其必需prettyTextualDescription属性的默认实现，以简单地返回访问该textualDescription属性的结果：

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

将约束添加到协议扩展

定义协议扩展时，可以指定符合类型在扩展的方法和属性可用之前必须满足的约束。你通过编写一个通用的where子句，在你要扩展的协议的名字后写这些约束。有关泛型where子句的更多信息，请参见泛型Where子句。

例如，您可以定义Collection适用于元素符合Equatable协议的任何集合的协议的扩展。通过将集合的元素约束到Equatable协议（标准库的一部分），可以使用==和!=运算符来检查两个元素之间的等式和不等式。

```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

该`allEqual()`方法`true`仅在集合中的所有元素相等时才返回。

考虑两个整数数组，一个是所有元素都相同，另一个不是：

```
let eqı
let differentNumbers = [100, 100, 200, 100, 200]
```

因为数组符合`Collection`和整数符合`Equatable`，`equalNumbers`并且`differentNumbers`可以使用该`allEqual()`方法：

```
print(equalNumbers.allEqual())
// Prints "true"
print(differentNumbers.allEqual())
// Prints "false"
```

> 注意
>
> 如果符合类型满足多个约束扩展的要求，这些扩展为相同的方法或属性提供实现，则Swift使用与最专用约束相对应的实现。