**扩展与协议**

**1.扩展**

扩展就是向一个已有的类、结构体或枚举类型添加新功能,但是不能重写已有的功能。

### 1.1语法

扩展声明使用关键字 **extension**：

extension SomeType {

    // 加到SomeType的新功能写到这里

}

1.2扩展属性

扩展可以向已有类型添加计算属性和类型属性,但是不能添加存储属性。

### 实例 下面的例子使用扩展向新建的New类型添加了1个计算属性和类型属性：

class New{}

extension New {                //扩展属性

var sub: Int { return 10 }//计算属性

// var sub1:Int=5 //报错，因为是存储属性

    class var PrintHello:String{//类型属性

        return "hello"

    }

}

let subtraction = New()

print("\(New.PrintHello),I am \(subtraction.sub)")

以上程序执行输出结果为：

**hello,I am 10**

## 1.3构造器

扩展可以向已有类型添加新的构造器。

### 实例 下面的例子使用扩展向新建的Newstruct类型添加了1个计算属性和类型属性：

struct Newstruct{

    var num:Int

}

extension Newstruct {                 //扩展构造器

    init(num1:Int){num=num1}

}

let m=Newstruct(num1: 10)

print(m.num)

以上程序执行输出结果为：

**10**

**1.4扩展方法**

扩展可以向已有类型添加新的实例方法和类型方法。

实例 下面的例子使用扩展向新建的NewClass类型添加了1个实例方法和类型方法:

class NewClass{}

extension NewClass{

    func rang(start:Int,end:Int){       //实例方法

        for i in start...end{

            print(i)

        }

    }

    class func PrintSwift(){            //类型方法

        print("Swift")

    }

}

let new=NewClass()

new.rang(start: 2, end: 6)

NewClass.PrintSwift()

以上程序执行输出结果为：

**2**

**3**

**4**

**5**

**6**

**Swift**

1.5扩展下标脚本

扩展可以向一个已有类型添加新下标。

实例 下面的例子使用扩展向NewClass类型添加下标脚本：

extension NewClass{                 //扩展下标脚本

    subscript(index:Int,value:Int)->Int{

        return index/value

    }

}

let newc=NewClass()

print(newc[66,3])

以上程序执行输出结果为：

**22**

**2.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。这些方法和属性并不在协议中具体实现，而只是用来描述这些实现应该是什么样子。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。

**协议的定义和实现方法**

协议的定义与类，结构体，枚举的定义非常相似，如下所示：

protocol 协议名称 {

    // 协议内容

}

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 {

    // 结构体内容

}

对于类来说，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 {

    // 类的内容

}

**协议成员声明——属性**

在协议中可以声明属性。它可以为其遵循者（即实现协议的类型）提供特定名称和类型的实例属性或类型属性，而不管其是存储型属性还是计算型属性，以下主要讲解这两种属性的声明。

实例属性的声明和类型属性的声明一般相同。形式如下：

protocol SomeProtocol {

    (static) var 属性名：数据类型 { get set } //可读写属性

    (static) var 属性名：数据类型 { get } //只读属性

}

其中括号中的内容在实例属性中不存在，存在于类型属性的声明中。

示例 以下实现协议NameProtocol的声明，然后在遵循者中的Sclass类型中进行实现：

protocol NameProtocol{

    static var age:Int { get }

    var name:String { get }

}

class Sclass: NameProtocol{

    var name:String {

        return "swift"

    }

    class var age:Int {

        return 10

    }

}

var ncc = Sclass()

print(ncc.name)

print(Sclass.age)

以上程序执行输出结果为：

**swift**

**10**

**协议成员声明——方法**

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。

在写一种方法的声明如下所示：

protocol 协议名{ //类型方法的声明

static func 方法名(参数名1：数据类型，参数名2：数据类型...)->返回值数据类型

}

protocol RandomNumberGenerator { //实例方法的声明

func 方法名(参数名1：数据类型，参数名2：数据类型...)->返回值数据类型

}

示例 以下实现协议的声明，然后在遵循者中的类型中进行实现：

以上程序执行输出结果为：

**协议成员声明——可变方法**

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)(特指结构体和枚举)中的的函数前缀加上mutating关键字来表示该函数允许改变该实例和其属性的类型。其语法形式如下：

protocol 协议名{

mutating func 方法名(参数名1：数据类型，参数名2：数据类型...)

}

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。

示例 以下在协议中定义可变方法，然后在遵循中进行实现。代码如下：

protocol Togglable {         //可变方法

    mutating func toggle()

}

enum OnOffSwitch:Int,Togglable {

    case Off=0

    case On=1

    mutating func toggle() {

        switch self {

        case .Off:

            self = .On

        case .On:

            self = .Off

        }

    }

}

var lightSwitch = OnOffSwitch.Off

lightSwitch.toggle()

//lightSwitch 现在的值为 .On

**协议成员声明——可选协议**

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。这样可以使代码量减少，从而解决了代码冗余的问题。

在协议中使用 @objc optional 关键字作为前缀来定义可选成员。其定义协议和声明可选成员的形式如下：

@objc protocol 协议名 {

    @objc optional func 方法名(参数名1：数据类型，参数名1：数据类型...)-> 数据类型

    @objc optional var 属性名: 数据类型 { get } //只读属性

}

同样方法适用于其他协议成员。可选成员可以让遵循协议者选择是否实现这些成员。

示例 以下声明可选协议和可选成员，并定义类来部分实现它。代码如下：

objc protocol CounterDataSource {    //可选协议

    @objc optional func printInCount(count:Int)->Int

    @objc optional var value:Int{get}

    }

class NewClass: CounterDataSource {

    @objc func printInCount(count:Int)->Int{

        let n=count\*count

        return n

    }

    @objc var value:Int{

        return 100

    }

}

let newclass=NewClass()

print(newclass.printInCount(count: 25))

 //输出结果为625

## 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:1、作为函数、方法或构造器中的参数类型或返回值类型；2、作为常量、变量或属性的类型；3、作为数组、字典或其他容器中的元素类型

示例 以下代码将协议类型用在各个场景，代码如下：

protocol Name {

    var name:String{get}

}

//定义类NewClass,并实现Name协议

class NewClass0:Name{

    var name: String="swift"

}

let value:Name=NewClass0()//协议类型作为变量的数据类型

print(value.name)

class NewClass1{

    var value1 : Name

    //将协议作为构造器的参数

    init(value1:Name) {

        self.value1=value1

    }

}

let newclass1=NewClass1(value1: NewClass0())

print(newclass1.value1.name)

class NewClass2:Name{

    var name:String{

        return "hello"

    }

}

let array:[Name]=[NewClass2(),NewClass0(),NewClass2()]//协议类型作为数组的元素类型

for i in 0..<3{

    print("array[\(i)]=\(array[i].name)")

}

以上程序执行输出结果为：

**swift**

**swift**

**array[0]=hello**

**array[1]=swift**

**array[2]=hello**

## **协议的继承和合成**

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，其语法形式如下：

protocol 协议名n: 协议名1, 协议名2，... {

    // 协议定义

}

一个协议可由多个协议采用protocol<SomeProtocol, AnotherProtocol>这样的格式进行组合，称为协议合成(protocol composition)。

示例 以下代码将使用协议合成海鲜对生日祝福的输出，代码如下：

protocol Named {   //协议的组合

    var name: String { get }

}

protocol Aged {

    var age: Int { get }

}

struct Person: Named, Aged {

    var name: String

    var age: Int

}

func wishHappyBirthday(celebrator: Named & Aged) {

    print("Happy birthday \(celebrator.name) - you're \(celebrator.age)!")

}

let birthdayPerson = Person(name: "Malcol", age: 21)

wishHappyBirthday(celebrator: birthdayPerson)

以上程序执行输出结果为：

**Happy birthday Malcol - you're 21!**

## 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**