# 第八章 扩展与协议

## 8.1 扩展

扩展可以用来向类，结构体和枚举添加新功能，但是已有的功能不能够在扩展中重写。也就是说，扩展只添加，不修改。

扩展声明使用关键字 extension：

extension 已定义类型名 {

    // 添加的的新功能

}

使用扩展可以实现以下操作：添加计算型属性和类型属性；定义实例方法和类型方法；提供新的构造器；定义下标；定义和使用新的嵌套类型；使一个已有类型符合某个协议。接下来将会详细介绍如何利用扩展进行操作。

### 8.1.1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。

【例 8-1】创建一个Num类并向这个类型扩展属性，代码如下：

class Num {

    var a:Int=2

    var stt = "hello"

}

extension Num{ //属性的扩展

var b:Double{return 2.1} //计算属性

// var b:Double=2.1 //报错，因为是存储属性

    class var PrintHelloWorld:String{ //类型属性

        return "hello world!!"

    }

}

var number=Num()

print(number.a,number.b,Num.PrintHelloWorld,number.stt)

运行结果为：

2 2.1 hello world!! hello

### 8.1.2 类型方法和实例方法

在扩展中可以向已有类型添加新的类型方法和实例方法。

【例 8-2】使用扩展向类Student添加新的类型方法和实例方法：

class  Student {

    var name:String=""

    var id:String=""

    var age:Int=0

}

//扩展方法

extension Student{

    func printInformation(name:String,id:String,age:Int) {

        print("\(name)'s id is \(id).")

        print("\(name) is \(age).")

    }

    class var isStudent:String{

        return "He/She is a student."

    }

}

print(Student.isStudent)

var st=Student()

st.printInformation(name: "Jane", id: "10015", age: 16)

运行结果为：

He/She is a student.

Jane's id is 10015.

Jane is 16.

对值类型进行扩展时，可以用关键字mutating 修饰某方法，使该方法可以在其内部直接改变实例本身的值。

【例 8-3】使用扩展向值类型添加新的方法来改变实例的值，代码如下：

extension Int{

    mutating func add(){

        self=self+10

    }

}

var h=5

h.add()

print(h)

运行结果为：

15

### 8.1.3 构造器

扩展可以向类型添加新的构造器。

【例 8-4】使用扩展向结构体Newstruct添加新的构造器：

struct Newstruct{

    var num:Int

    var str:String="nothing"

    init(num0:Int) { //原构造器

        num=num0

    }

}

extension Newstruct { //扩展构造器

    init(num1:Int,str1:String){

        num=num1

        str=str1

    }

}

let n=Newstruct(num0: 55)

let m=Newstruct(num1: 66, str1: "Hello")

print(n.num,n.str)

print(m.num,m.str)

运行结果为：

55 nothing

66 Hello

对于类来说，扩展只能添加新的便利构造器 init()，但是不能向类中添加新的指定构造器或析构函数 deinit() 。向类添加构造器时，要在init之前添加关键字convenience。

【例 8-5】使用扩展向Ms类添加新的构造器：

class Ms {

    var f:Int

    var str:String

    init() {

        f=10

        str="jarry"

    }

}

extension Ms{

    convenience init(f:Int,nr:String){

        self.init()

        self.str=nr

        self.f=f

    }

}

var obj=Ms()

print(obj.str)

var obj1=Ms(f: 66, nr: "micheal")

print(obj1.str)

运行结果为：

jarry

micheal

### 8.1.4下标脚本

使用扩展可以向已有类型添加新的下标脚本。创建以subscript为函数名的函数，可以使用下标脚本来运行该函数。

【例8-6】使用扩展向Ss类型添加新的下标脚本：

struct Ss {

    var nee:Int

}

extension Ss{

    subscript(x:Int,y:Int)->Bool{

        return x\*y/nee == 0 ? true:false

    }

}

let ss=Ss(nee: 3)

print(ss[8,22])

运行结果为：

false

## 8.2 协议

协议规定了一系列属性和方法，但又不在协议中具体实现它。这和对结构体中的属性只声明不定义很相像。对于能够按协议要求对协议规定的属性和方法进行实现的类型，被称为遵守该协议。这里的类型包括类，结构体和枚举。

协议的定义形式如下：

protocol 协议名称 {

    // 协议内容

}

对于遵守该协议的类型，以类为例，形式如下：

class 类名: 协议一, 协议二，...{

    // 类内容，包括对协议内容的实现

}

类型名后面跟冒号 ‘：’ 后跟遵守的协议。类型可以遵守多个协议，各协议之间用逗号 ‘，’ 隔开。对于类来说，如果它要继承某个类，那么要把继承的父类名放在所有协议之前，形式如下:

class 类名: 父类, 协议一, 协议二，...{

// 类内容，包括对协议内容的实现

}

下面将会介绍对于协议中不同种类成员的具体声明和其实现。

### 8.2.1属性

协议中可以声明属性。属性类型可以是实例属性或类型属性，可以是存储属性或计算属性。协议中通常用 var 来声明变量属性，而且必须要说明该属性是只读属性还是可读写属性。

对协议中属性的声明如下：

protocol 协议名 {

    var 属性名：数据类型 { get set } //可读写属性

var 属性名：数据类型 { get } //只读属性

static var 属性名：数据类型 { get } //类型属性

}

【例8-7】声明协议Student并定义类Studentclass来遵守该协议：

protocol Stu{

    static var age:Int { get }

    var name:String { get }

}

class Studentclass: Stu{

    var name:String {

        return "Jerry"

    }

    class var age:Int {

        return 10

    }

}

var ncc = Studentclass()

print(ncc.name)

print(Studentclass.age)

运行结果为：

Jerry

10

### 8.2.2方法

协议中可以声明方法。在协议中声明的方法不声明方法内容，只声明该方法的参数及其类型和返回值类型。协议中方法的声明形式如下：

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

//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

}

【例8-8】声明协议PrintHello并定义类SayHello来遵守该协议：

protocol PrintHello {

    func printHello()

}

class SayHello:PrintHello{

    var name:String="Jane"

    func printHello(){

        print("Hello \(name)!!")

    }

}

let ls=SayHello()

ls.printHello()

运行结果为：

Hello Jane!!

当实例为值类型时，如果想利用协议中的方法改变实例中成员的值，必须要用到关键字‘mutating’，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...)

}

【例8-9】声明协议Change并定义枚举number来遵守该协议：

protocol Change {

    mutating func changeNum()

}

enum numberChoice:Double,Change{

    case a=1.1

    case b=2.2

    case c=3.3

    mutating func changeNum() { //改变枚举中属性的值

        switch self{

        case .a:

            self = .c

        case .b:

            break

        case .c:

            self = .a

        }

    }

}

var nn=numberChoice.a

nn.changeNum()

print(nn.rawValue)

运行结果为：

3.3

### 8.2.3可选协议

所谓可选协议，就是遵守该协议的类型可以选择实现或不实现协议成员，可以实现全部成员或部分成员。这让协议的灵活性大大增强。可选协议名前要加上关键字 @objc，协议中使用关键字 @objc optional 作为前缀来定义可选成员。其一般形式如下：

@objc protocol 协议名 {

    @objc optional func 方法名(参数名1：数据类型，参数名2：数据类型...)-> 数据类型

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

}

【例8-10】声明可选协议DataChange并定义类Data来遵守该协议：

@objc protocol DataMul{

    @objc optional var value:Int{get}

    @objc optional func dataMultiplied(data:Double)->Double

}

//以下类没有实现协议中所有成员

class Data:DataMul{

    var n=1.0

    var value1 = 3

    @objc func dataMultiplied(data: Double) -> Double {

        for \_ in 1...value1{

             n=n\*data

        }

    return n

    }

}

let data\_x=Data()

print(data\_x.dataMultiplied(data: 2.2))

运行结果为：

10.648

### 8.2.4协议类型

协议可以作为类型被使用。如某协议作为某函数参数的类型，那么这个参数可以是任意遵守了此协议的数据类型。

【例8-11】声明协议Naaa,使用该协议类型作为方法cak中参数的参数类型，并定义结构体Sd来遵守该协议：

protocol Naaa {                            //协议

    var h:Double{get}

    var style:String{get}

    var r:Double{get}

}

func cake(pa:Naaa){                        //参数为协议类型

    let h=pa.h

    let area=pa.r\*pa.r

    let name=pa.style

    print(h,area,name)

}

struct CakeStyle:Naaa{                    //遵守协议的结构体

    var h:Double

    var style: String

    var r: Double

}

//结构体实例化

let s = CakeStyle(h: 0.2, style: "pudding", r: 0.6)

cake(pa:s)                                //结构体类型作为参数

运行结果为：

0.2 0.36 pudding

假如某数组的类型为协议类型，如：

var array:Array<OneProtocol>

这表示数组里的所有元素都要遵守协议。

### 8.2.5协议的继承

协议也可以像类一样被其他协议继承。对于继承了某协议的协议，相当于拥有父类协议声明的所有成员。而遵守了该协议的数据类型，要实现该协议本身和其父协议声明的所有成员。

【例8-12】声明协议FirstProtocol，声明协议SecondProtocol继承FirstProtocol并定义结构体Mystruct来遵守SecondProtocol协议：

protocol FirstProtocol {

    var a:Int{get}

    var b:Double{get}

}

protocol SecondProtocol:FirstProtocol { //继承协议

    var s:String{get}

}

struct Mystruct:SecondProtocol { //结构体遵守协议

    var a=5

    var b=1.26

    var s="hello Swift"

}

### 8.2.6协议的其他功能

有的类型要同时遵循好几个协议，Swift协议可以使用关键字 ‘&’ 进行合成，这样可以在某些情况下简化代码的书写。一般形式如下：

protocol case {}

protocol case1 {}

protocol case2 {}

class Saa:case&case1&case2{}

使用 ‘&’ 把各协议隔开就可实现协议的合成。

如果想让某协议只能被类遵守，可以用关键字class来说明，如：

protocol Myprotocol:class {}

有的实例遵守多个协议或者遵守的协议继承其他协议，当想要检查某实例是否遵循某协议或想把该实例转化为某协议类型时，就要用到关键字 ’is’、’as!’、’as?’ 。这几个关键字的功能分别是：

（1）is操作符用来检查实例是否遵循了某个协议。

（2）as?是当实例遵循某协议时，返回该协议类型;否则返回nil。

（3）as！用以强制向下转型。使用时一定要确定要转型的协议是所需要的。如果强转失败，会引起运行时错误。

## 8.3 小结

本章主要介绍了扩展与协议的相关知识，如基本形式、调用方法、自身特性和自身其他功能。本章的重点在于掌握扩展与协议的基本用法，了解扩展与协议的全部用法，熟练应用扩展与协议的声明定义和调用方法。