协议

@M了个J

https://github.com/CoderMJLee

http://cnblogs.com/mjios

码拉松

协议（Protocol）

- 协议可以用来定义方法、属性、下标的声明，协议可以被枚举、结构体、类遵守（多个协议之间用逗号隔开）

```
protocol Drawable {
    func draw()
    var x: Int { get set }
    var y: Int { get }
    subscript(index: Int) -> Int { get set }
}
```

```
protocol Test1 {}
protocol Test2 {}
protocol Test3 {}
class TestClass : Test1, Test2, Test3 {}
```

- 协议中定义方法时不能有默认参数值
- 默认情况下，协议中定义的内容必须全部都实现
- 也有办法办到只实现部分内容，以后的课程会讲到

协议中的属性

```
protocol Drawable {
    func draw()
    var x: Int { get set }
    var y: Int { get }
    subscript(index: Int) -> Int { get set }
}
```

- 协议中定义属性时必须用var关键字
- 实现协议时的属性权限要不小于协议中定义的属性权限
  - 协议定义get、set，用var存储属性或get、set计算属性去实现
  - 协议定义get，用任何属性都可以实现

```
class Person : Drawable {
    var x: Int = 0
    let y: Int = 0
    func draw() {
        print("Person draw")
    }
    subscript(index: Int) -> Int {
        set {}
        get { index }
    }
}
```

```
class Person : Drawable {
    var x: Int {
        get { 0 }
        set {}
    }
    var y: Int { 0 }
    func draw() { print("Person draw") }
    subscript(index: Int) -> Int {
        set {}
        get { index }
    }
}
```

static、class

- 为了保证通用，协议中必须用static定义类型方法、类型属性、类型下标

```
protocol Drawable {
    static func draw()
}

class Person1 : Drawable {
    class func draw() {
        print("Person1 draw")
    }
}

class Person2 : Drawable {
    static func draw() {
        print("Person2 draw")
    }
}
```

mutating

- 只有将协议中的实例方法标记为mutating
  - 才允许结构体、枚举的具体实现修改自身内存
  - 类在实现方法时不用加mutating，枚举、结构体才需要加mutating

```
protocol Drawable {
    mutating func draw()
}

class Size : Drawable {
    var width: Int = 0
    func draw() {
        width = 10
    }
}

struct Point : Drawable {
    var x: Int = 0
    mutating func draw() {
        x = 10
    }
}
```

init

- 协议中还可以定义初始化器init
  - 非final类实现时必须加上required

```
protocol Drawable {
    init(x: Int, y: Int)
}

class Point : Drawable {
    required init(x: Int, y: Int) {}
}

final class Size : Drawable {
    init(x: Int, y: Int) {}
}
```

- 如果从协议实现的初始化器，刚好是重写了父类的指定初始化器
  - 那么这个初始化必须同时加required、override

```
protocol Livable {
    init(age: Int)
}

class Person {
    init(age: Int) {}
}

class Student : Person, Livable {
    required override init(age: Int) {
        super.init(age: age)
    }
}
```

init、init?、init!

- 协议中定义的init?、init!，可以用init、init?、init!去实现
- 协议中定义的init，可以用init、init!去实现

```
protocol Livable {
    init()
    init?(age: Int)
    init!(no: Int)
}
```

```
class Person : Livable {
    required init() {}
    // required init!() {}

    required init?(age: Int) {}
    // required init!(age: Int) {}
    // required init(age: Int) {}

    required init!(no: Int) {}
    // required init?(no: Int) {}
    // required init(no: Int) {}
}
```

协议的继承

- 一个协议可以继承其他协议

```
protocol Runnable {
    func run()
}

protocol Livable : Runnable {
    func breath()
}

class Person : Livable {
    func breath() {}
    func run() {}
}
```

协议组合

```
protocol Livable {}
protocol Runnable {}
class Person {}
```

- 协议组合，可以包含1个类类型（最多1个）

```
// 接收Person或者其子类的实例
func fn0(obj: Person) {}
// 接收遵守Livable协议的实例
func fn1(obj: Livable) {}
// 接收同时遵守Livable、Runnable协议的实例
func fn2(obj: Livable & Runnable) {}
// 接收同时遵守Livable、Runnable协议、并且是Person或者其子类的实例
func fn3(obj: Person & Livable & Runnable) {}
```

```
typealias RealPerson = Person & Livable & Runnable
// 接收同时遵守Livable、Runnable协议、并且是Person或者其子类的实例
func fn4(obj: RealPerson) {}
```

CaseIterable

- 让枚举遵守CaseIterable协议，可以实现遍历枚举值

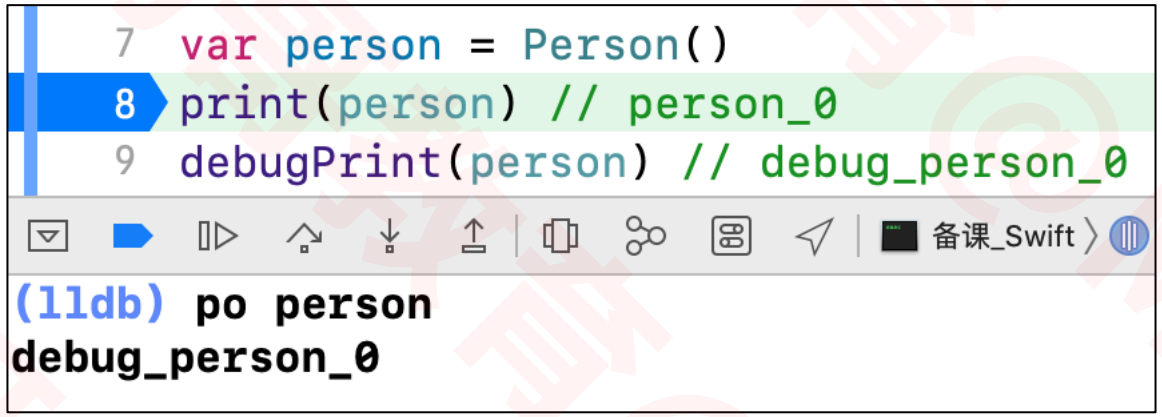
```
enum Season : CaseIterable {
    case spring, summer, autumn, winter
}
let seasons = Season.allCases
print(seasons.count) // 4
for season in seasons {
    print(season)
} // spring summer autumn winter
```

CustomStringConvertible

- 遵守CustomStringConvertible、 CustomDebugStringConvertible协议，都可以自定义实例的打印字符串

```
class Person : CustomStringConvertible, CustomDebugStringConvertible {
    var age = 0
    var description: String { "person_\(age)" }
    var debugDescription: String { "debug_person_\(age)" }
}
var person = Person()
print(person) // person_0
debugPrint(person) // debug_person_0
```

- print调用的是CustomStringConvertible协议的description
- debugPrint、po调用的是CustomDebugStringConvertible协议的debugDescription



Any、AnyObject

- Swift提供了2种特殊的类型：Any、AnyObject
  - Any：可以代表任意类型（枚举、结构体、类，也包括函数类型）
  - AnyObject：可以代表任意类类型（在协议后面写上：AnyObject代表只有类能遵守这个协议）
    - ✓ 在协议后面写上：class也代表只有类能遵守这个协议

```
var stu: Any = 10
stu = "Jack"
stu = Student()
```

```
// 创建1个能存放任意类型的数组
// var data = Array<Any>()
var data = [Any]()
data.append(1)
data.append(3.14)
data.append(Student())
data.append("Jack")
data.append({ 10 })
```

is、as?、as!、as

- is用来判断是否为某种类型，as用来做强制类型转换

```
protocol Runnable { func run() }
class Person {}
class Student : Person, Runnable {
    func run() {
        print("Student run")
    }
    func study() {
        print("Student study")
    }
}
```

```
var stu: Any = 10
print(stu is Int) // true
stu = "Jack"
print(stu is String) // true
stu = Student()
print(stu is Person) // true
print(stu is Student) // true
print(stu is Runnable) // true
```

```
var stu: Any = 10
(stu as? Student)?.study() // 没有调用study
stu = Student()
(stu as? Student)?.study() // Student study
(stu as! Student).study() // Student study
(stu as? Runnable)?.run() // Student run
```

```
var data = [Any]()
data.append(Int("123") as Any)
```

```
var d = 10 as Double
print(d) // 10.0
```

X.self、X.Type、AnyClass

- X.self是一个元类型（metadata）的指针，metadata存放着类型相关信息
- X.self属于X.Type类型

```
class Person {}
class Student : Person {}
var perType: Person.Type = Person.self
var stuType: Student.Type = Student.self
perType = Student.self
```

```
var anyType: AnyObject.Type = Person.self
anyType = Student.self

public typealias AnyClass = AnyObject.Type
var anyType2: AnyClass = Person.self
anyType2 = Student.self
```

```
var per = Person()
var perType = type(of: per) // Person.self
print(Person.self == type(of: per)) // true
```

小码哥教育 SEEMYGO

元类型的应用

```
class Animal { required init() {} }
class Cat : Animal {}
class Dog : Animal {}
class Pig : Animal {}

func create(_ clses: [Animal.Type]) -> [Animal] {
    var arr = [Animal]()
    for cls in clses {
        arr.append(cls.init())
    }
    return arr
}

print(create([Cat.self, Dog.self, Pig.self]))
```

元类型的应用

```
import Foundation
class Person {
    var age: Int = 0
}
class Student : Person {
    var no: Int = 0
}
print(class_getInstanceSize(Student.self)) // 32
print(class_getSuperclass(Student.self)!) // Person
print(class_getSuperclass(Person.self)!) // Swift._SwiftObject
```

- 从结果可以看得出来，Swift还有个隐藏的基类：**Swift._SwiftObject**
- 可以参考Swift源码：https://github.com/apple/swift/blob/master/stdlib/public/runtime/SwiftObject.h

Self

- Self代表当前类型

```
class Person {
    var age = 1
    static var count = 2
    func run() {
        print(self.age) // 1
        print(Self.count) // 2
    }
}
```

- Self一般用作返回值类型，限定返回值跟方法调用者必须是同一类型（也可以作为参数类型）

```
protocol Runnable {
    func test() -> Self
}
class Person : Runnable {
    required init() {}
    func test() -> Self { type(of: self).init() }
}
class Student : Person {}
```

```
var p = Person()
// Person
print(p.test())

var stu = Student()
// Student
print(stu.test())
```