Mac OS X、iOS平台的Cocoa程序设计

许国雄 xgx@pku.edu.cn

Xcode IDE、Objective-C、Swift、Cocoa框架

2015年3月31日

SWIFT

- **Protocols**

```
// A protocol defines a specific set of features that a class, struct, or other custom
// type can adopt. Once adopted, your type can be used anywhere there's a reference to
// that protocol type.
protocol Flavor {
   var flavor: String { get }
   func tastesLike(flavor otherFlavor: Flavor) -> Bool
}
class IceCream {
   var flavor: String = "Vanilla"
   var scoops: Int = 1
}
class Sundae: IceCream, Flavor {
   func tastesLike(flavor otherFlavor: Flavor) -> Bool {
    return flavor == otherFlavor.flavor
   }
}
class Tofu: Flavor {
   var flavor: String = "Plain"
   var weight: Double = 1.0
   func tastesLike(flavor _: Flavor) -> Bool {
    // Tofu doesn't taste like anything else
    return false
   }
}
var dessert: Flavor = Sundae()
var chicken = Tofu()
dessert.tastesLike(flavor: chicken)
```

与OC不同之处：
Swift protocols do not have optional members.

用途：
Protocols are regularly used for *delegates*.

SWIFT

- **Extensions**

限制:
An extension cannot add ~~stored properties~~ to an existing type. You can only add methods and computed properties.
You cannot override an existing property or method in an extension.

```
class MyWhatsit {
   var name: String
   var location: String
   var image: UIImage?
   init( name: String, location: String = "" ) {
        self.name = name;
        self.location = location
   }
}
// Extensions add new properties or functions to an existing type
extension MyWhatsit {
   var viewImage: UIImage? {
        return image ?? UIImage(named: "camera")
   }
}
var thing = MyWhatsit(name: "Robby the Robot")
if thing.viewImage?.size.width > 100.0 {
}
// Extensions can add methods and properties to types you
// didn't even write or have the source code to.
extension String {
   var inKlingon: String { return self == "Hello" ? "nuqneH" : "nuqjatlh?" }
}
"Hello".inKlingon
"Anything else".inKlingon
```

SWIFT

- **Structures**

```
// A Swift struct
struct SwiftStruct {
   var storedProperty: String = "Remember me?"
   let someFixedNumber = 9
   var computedProperty: String {
     get {
        return storedProperty.lowercaseString
     }
     set {
        storedProperty = newValue
     }
   }

   static func globalMethod() {
   }

   func instanceMethod() {
   }

   init(what: String) {
     // initializer
     storedProperty = what
   }
}
```

与类的相同点：

Can define stored properties (and stored properties can have observers)
Can define computed properties
Can define methods (instance functions)
Can define global functions
Can have custom initializers
Can adopt protocols
Can be augmented with extensions

与类的不同点：

A structure cannot inherit from another structure.
A structure is a value type, not a reference type.
A structure does not have a ~~deinit~~ function.
A global function in a structure is declared using the *static* keyword instead of the *class* keyword.

SWIFT

- **Structures**

```
// Initializing a struct
struct StructWithDefaults {
   var number: Int = 1
   var name: String = "james"
   // A struct with stored properties that all have default values gets two automatic initializers.
   //   A default initializer (just like a class) and a memberwise initializer.
   // init()                              <- generated by Swift
   // init(number: Int, name: String) <- generated by Swift
}
let struct1 = StructWithDefaults()
let struct2 = StructWithDefaults(number: 3, name: “john")

struct StructWithoutDefaults {
   var number: Int
   var name: String
   // A struct with any stored properties that lack a default value does not automatically get a
   //   default initializer, but it still gets a memberwise initializer.
   // init(number: Int, name: String) <- generated by Swift
}
//let struct3 = StructWithoutDefaults() <- no such initializer
let struct4 = StructWithoutDefaults(number: 4, name: “fred")

// A struct that declares its own initializer does not get
// an automatic default initializer or memberwise initializer.
let struct5 = SwiftStruct(what: "up?")  // custom initializer
//let struct6 = SwiftStruct(storedProperty: "down", someFixedNumber: 1) <- invalid
//let struct7 = SwiftStruct() <- invalid
```

SWIFT

- **Tuples:** Tuples are ad hoc, anonymous, **structures**.

```
// You create a tuple value by enclosing a list of values in parentheses,
// optionally naming the members.
var iceCreamOrder = ("Vanilla", 1)
let wantToGet = (flavor: "Chocolate", scoops: 4)
// A tuple can be returned to assigned to any other tuple with the same type.
iceCreamOrder = wantToGet

// A function returning a tuple with named members.
func nextOrder() -> (flavor: String, scoops: Int) {
   // This is how you assemble a tuple from individual values.
   return ("Vanilla", 4)
}

// The tuple can be assigned to a single value. The value
// then acts much like a struct; you use the member names
// to access the individual values.
let order = nextOrder()
if order.scoops > 3 {
   println("\(order.scoops) scoops of \(order.flavor) in a dish.")
} else {
   println("\(order.flavor) cone")
}
```

SWIFT

- **Tuples:** Tuples are ad hoc, anonymous, **structures**.

```
// A tuple can be "decomposed" into individual values when assigned.
let (what,howMany) = nextOrder()
if howMany > 3 {
   println("\(howMany) scoops of \(what) in a dish.")
} else {
   println("\(what) cone")
}

// When you decompose a tuple, you can ignore some of the values
let (flavor2,_) = nextOrder()    // ignore the scoops value
if flavor2 == "Strawberry" { /* ... */ }

// A tuple type can omit the member names
func anonymousOrder() -> (String, Int) {
   return ("Chocolate", 2)
}

// To use a tuple with anonymous members, you must either decompose
//the tuple or use a member's index.
let anon = anonymousOrder()
if anon.0 == "Chocolate" {
   println("\(anon.1) scoops of the good stuff.")
}
```

SWIFT

- **Enumerations**

```
// Create an enumeration by listing the values you want defined.
enum Weekday {
   case Sunday
   case Monday
   case Tuesday
   case Wednesday
   case Thursday
   case Friday
   case Saturday
}

// An enumeration's value is NameOfEnumeration.nameOfValue
var dueDay = Weekday.Thursday

// When assigning to a variable or parameter that already
// has an enum type, you can omit the enum name from the value.
dueDay = .Friday
```

SWIFT

- **Raw Values**

```
// An enum can be made compatible with another type. In this example, the enum values can be
// converted to and from an Int value, called it raw value.
enum WeekdayNumber: Int {
   case Sunday = 0
   case Monday
   case Tuesday
   case Wednesday
   case Thursday
}
// An enum value with a raw value can be converted to that raw
// value using its rawValue property.
let dayNumber: Int = WeekdayNumber.Wednesday.rawValue
// You can also create new enum values from raw values, as follows
let dayFromNumber = WeekdayNumber(rawValue: 3)  // <-- .Wednesday

// Enum values can also be non-numeric, in which case you must assign every raw value individually.
enum WeekdayName: String {
   case Sunday = "Sunday"
   case Monday = "Monday"
   case Tuesday = "Tuesday"
   case Wednesday = "Wednesday"
   case Thursday = "Thursday"
}
let reportDay: WeekdayName = .Monday
println("Reports are due on \(reportDay.rawValue)")
```

SWIFT

- **Associated Values**：将一个枚举值与一个Tuple关联

```
enum Sadness {
   case Sad
   case ReallySad
   case SuperSad
}
```

```
enum WeekdayChild {
   case Sunday
   case Monday
   case Tuesday(sadness: Sadness)
   case Wednesday
   case Thursday(distanceRemaining: Double)
   case Friday(friends: Int, charities: Int)
   case Saturday(workHours: Float)
}

let graceful = WeekdayChild.Monday
let woeful = WeekdayChild.Tuesday(sadness: .ReallySad)
let traveler = WeekdayChild.Thursday(distanceRemaining: 100.0)
let social = WeekdayChild.Friday(friends: 23, charities: 4)
let worker = WeekdayChild.Saturday(workHours: 90.5)
```

SWIFT

- **Associated Values**：只能通过 switch 语句访问到 associated value.

```
var child = WeekdayChild.Thursday(distanceRemaining: 100.0)

switch child {
   case .Sunday, .Monday, .Wednesday:
    break
   case .Tuesday(let mood):
    if mood == .ReallySad {
        println("Tuesday's child is really sad.")
    }
   case .Thursday(let miles):
    println("Thursday's child has \(miles) miles to go.")
   case .Friday(let friendCount, _):
    println("Friday's child has \(friendCount) friends.")
   case .Saturday(let hours):
    println("Saturday's child is working \(hours) hours this week.")
}
```

The reason for this odd mechanism is that an enumeration value could, potentially, contain any of the possible enumeration values. But the associated values are only valid when it is set to a specific enumeration value. The switch case ensures that access is only granted to the associated values for that specific enumeration value.

SWIFT

- **Extended Enumerations**

能：An enumeration can define *computed properties* and have *static and instance methods*. It can *adopt protocols*, can be *extended by extensions*, and can have *custom initializers*.
不能：You cannot add ~~stored properties~~; use associated values for that.

```
var nextWeekDayNumber: Int = 0
enum WeekdayNumber: Int {
   case Sunday = 0
   case Monday
    ……
   case Saturday
   // Enums can have computed properties and member functions
   var range: NSRange { return NSRange(location: 0, length: 7) }
   func dayOfJulienDate(date: NSTimeInterval) -> WeekdayNumber {
    return WeekdayNumber(rawValue: Int(date) % 7 + 2)!
   }
   // A member function or property accesses its value using self
   var weekend: Bool { return self == .Sunday || self == .Saturday }
   var weekday: Bool { return !weekend }
   init(random: Bool) {
    if random {
        // Pick a day at random
        nextWeekDayNumber = Int(arc4random_uniform(7))
    }
    self = WeekdayNumber(rawValue: nextWeekDayNumber)!
    // post increment to the next day
    nextWeekDayNumber++
    if nextWeekDayNumber > WeekdayNumber.Saturday.rawValue {
        nextWeekDayNumber = WeekdayNumber.Sunday.rawValue
    }
   }
}
```

SWIFT

- **Numeric Values:**
  - Integer types come in a variety of sizes and signs: **Int, UInt, Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64,** and **UInt64.** The vast majority of the time you should use **Int**, occasionally *UInt*, and almost nothing else. [可避免类型转换]。32位系统中，Int,UInt为32位；64位系统中，则为64位。
  - Floating-point numbers come in two flavors: **Float** and **Double**. Float is always 32 bits, and Double is always 64-bits.
  - The iOS Foundation library defines the **CGFloat** type. It is 32-bits (Float) when compiled for 32-bit processors and 64-bits (Double) on 64-bit processors.
- 数字类型间必须采用显式转换，当转换过程中发生溢出时，将发生错误；浮点数转为整数时采用截断小数的方法。unsigned = **UInt**(signed)
- 长的数字可以分段用下划线连接
- Numbers Are Types Too： A numeric type can have computed properties and methods and can be extended with extensions.
- 在运算符前加 &，可忽略溢出错误

SWIFT

***Table 20-2.** Literal Number Formats*

Form	Description
`123`	Positive integer number
`-86`	Negative integer number
`0xa113cafe`	Hexadecimal number
`0o557`	Octal number
`0b010110110`	Binary number
`1.5`	Floating-point number
`6.022e23`	Floating-point number with exponent
`0x2ff0.7p2`	Hexadecimal floating-point number with exponent

SWIFT

- String and Character Literals

Table 20-3. String Literal Escape Codes

Escape Sequence	Character in String
\\	A single \ character
\"	A single ” character
\'	A single ’ character
\t	Horizontal tab character
\r	Carriage return character
\0	Null character
\u{*nnnn*}	Unicode scalar with the hexadecimal code of *nnnn*
\(*expression*)	String representation of expression (string interpolation)

SWIFT

- **Optional：解决 nil 的问题，使用 ! 解包，在 if 中自动解包**
  - Optional Chaining：*cheshire?.friend?.doorMouse?.recitePoem()*
  - Failable Initializers：init?
  - 隐式解包：var dangerWillRobinson: String!，常用于Interface Builder： @IBOutlet var label: UILabel!

SWIFT

- **Type Casting**
  - 用 is 判断类型
  - 用 as? 进行 down casting

SWIFT

- **Working with C and Objective-C**

Table 20-4. Toll-Free Bridge Types

Swift	Objective-C	C
Array	NSArray	CFArrayRef
	NSAttributedString	CFAttributedStringRef
	NSCharacterSet	CFCharacterSetRef
	NSData	CFDataRef
	NSDate	CFDateRef
Dictionary	NSDictionary	CFDictionaryRef
	NSMutableArray	CFMutableArrayRef
	NSMutableAttributedString	CFMutableAttributedStringRef
	NSMutableCharacterSet	CFMutableCharacterSetRef
	NSMutableData	CFMutableDataRef
	NSMutableDictionary	CFMutableDictionaryRef
	NSMutableSet	CFMutableSetRef
	NSMutableString	CFMutableStringRef
	NSNumber	CFNumberRef
	NSSet	CFSetRef
String	NSString	CFStringRef
	NSURL	CFURLRef