Swift 泛型

Swift 访问控制

访问控制可以限定其他源文件或模块中代码对你代码的访问级别。

你可以明确地给单个类型（类、结构体、枚举）设置访问级别，也可以给这些类型的属性、函数、初始化方法、基本类型、下标索引等设置访问级别。

协议也可以被限定在一定的范围内使用，包括协议里的全局常量、变量和函数。

访问控制基于模块与源文件。

模块指的是以独立单元构建和发布的 Framework 或 Application。在 Swift 中的一个模块可以使用 import 关键字引入另外一个模块。

源文件是单个源码文件，它通常属于一个模块， 源文件可以包含多个类和函数 的定义。

Swift 为代码中的实体提供了四种不同的访问级别:public、internal、fileprivate、private。

访问级别	定义
public	可以访问自己模块中源文件里的任何实体，别人也可以通过引入该模块来访问源文件里的所有实体。
internal	可以访问自己模块中源文件里的任何实体，但是别人不能访问该模块中源文件里的实体。
fileprivate	文件内私有，只能在当前源文件中使用。
private	只能在类中访问，离开了这个类或者结构体的作用域外面就无法访问。

public 为最高级访问级别，private 为最低级访问级别。

语法

通过修饰符public、internal、fileprivate、private来声明实体的访问级别：

实例

```
public class SomePublicClass {}
internal class SomeInternalClass {}
fileprivate class SomeFilePrivateClass {}
private class SomePrivateClass {}
public var somePublicVariable = 0
internal let someInternalConstant = 0
fileprivate func someFilePrivateFunction() {}
private func somePrivateFunction() {}
```

除非有特殊的说明，否则实体都使用默认的访问级别 internal。

未指定访问级别默认为 internal

```
class SomeInternalClass {} // 访问级别为 internal
let someInternalConstant = 0 // 访问级别为 internal
```

函数类型访问权限

函数的访问级别需要根据该函数的参数类型和返回类型的访问级别得出。

下面的例子定义了一个名为someFunction全局函数，并且没有明确地申明其访问级别。

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // 函数实现
}
```

函数中其中一个类 SomeInternalClass 的访问级别是 internal，另一个 SomePrivateClass 的访问级别是 private。所以根据元组访问级别的原则，该元组的访问级别是 private（元组的访问级别与元组中访问级别最低的类型一致）。

因为该函数返回类型的访问级别是 private，所以你必须使用 private 修饰符，明确的声明该函数：

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // 函数实现
}
```

将该函数申明为 public 或 internal，或者使用默认的访问级别 internal 都是错误的，因为如果这样你就无法访问 private 级别的返回值。

枚举类型访问权限

枚举中成员的访问级别继承自该枚举，你不能为枚举中的成员单独申明不同的访问级别。

实例

比如下面的例子，枚举 Student 被明确的申明为 public 级别，那么它的成员 Name，Mark 的访问级别同样也是 public：

实例

```
public enum Student {
case Name(String)
case Mark(Int,Int,Int)
}
var studDetails = Student.Name("Swift")
var studMarks = Student.Mark(98,97,95)
switch studMarks {
case .Name(let studName):
print("学生名: \(studName).")
case .Mark(let Mark1, let Mark2, let Mark3):
print("学生成绩: \(Mark1),\(Mark2),\(Mark3)")
}
```

以上程序执行输出结果为：

```
学生成绩: 98,97,95
```

子类访问权限

子类的访问级别不得高于父类的访问级别。比如说，父类的访问级别是internal，子类的访问级别就不能申明为public。

实例

```
public class SuperClass {
fileprivate func show() {
print("超类")
}
}
// 访问级别不能低于超类 internal > public
internal class SubClass: SuperClass {
override internal func show() {
print("子类")
}
}
let sup = SuperClass()
sup.show()
let sub = SubClass()
sub.show()
```

以上程序执行输出结果为：

```
超类
子类
```

常量、变量、属性、下标访问权限

常量、变量、属性不能拥有比它们的类型更高的访问级别。

比如说，你定义一个public级别的属性，但是它的类型是private级别的，这是编译器所不允许的。

同样，下标也不能拥有比索引类型或返回类型更高的访问级别。

如果常量、变量、属性、下标索引的定义类型是private级别的，那么它们必须要明确的申明访问级别为private:

```
private var privateInstance = SomePrivateClass()
```

Getter 和 Setter访问权限

常量、变量、属性、下标索引的Getters和Setters的访问级别继承自它们所属成员的访问级别。

Setter的访问级别可以低于对应的Getter的访问级别，这样就可以控制变量、属性或下标索引的读写权限。

实例

```
class Samplepgm {
fileprivate var counter: Int = 0{
willSet(newTotal){
print("计数器: \(newTotal)")
}
```

```
didSet{
if counter > oldValue {
print("新增加数量 \(counter - oldValue)")
}
}
}
}
let NewCounter = Samplepgm()
NewCounter.counter = 100
NewCounter.counter = 800
```

counter 的访问级别为 fileprivate，在文件内可以访问。

以上程序执行输出结果为：

```
计数器: 100
新增加数量 100
计数器: 800
新增加数量 700
```

构造器和默认构造器访问权限

初始化

我们可以给自定义的初始化方法申明访问级别，但是要不高于它所属类的访问级别。但必要构造器例外，它的访问级别必须和所属类的访问级别相同。

如同函数或方法参数，初始化方法参数的访问级别也不能低于初始化方法的访问级别。

默认初始化方法

Swift为结构体、类都提供了一个默认的无参初始化方法，用于给它们的所有属性提供赋值操作，但不会给出具体值。

默认初始化方法的访问级别与所属类型的访问级别相同。

实例

在每个子类的 init() 方法前使用 required 关键字声明访问权限。

实例

```
class classA {
required init() {
var a = 10
print(a)
}
}
class classB: classA {
required init() {
var b = 30
print(b)
}
}
```

```
let res = classA()
let show = classB()
```

以上程序执行输出结果为：

```
10
30
10
```

协议访问权限

如果想为一个协议明确的申明访问级别，那么需要注意一点，就是你要确保该协议只在你申明的访问级别作用域中使用。

如果你定义了一个public访问级别的协议，那么实现该协议提供的必要函数也会是public的访问级别。这一点不同于其他类型，比如，public访问级别的其他类型，他们成员的访问级别为internal。

实例

```
public protocol TcpProtocol {
init(no1: Int)
}
public class MainClass {
var no1: Int // local storage
init(no1: Int) {
self.no1 = no1 // initialization
}
}
class SubClass: MainClass, TcpProtocol {
var no2: Int
init(no1: Int, no2 : Int) {
self.no2 = no2
super.init(no1:no1)
}
// Requires only one parameter for convenient method
required override convenience init(no1: Int) {
self.init(no1:no1, no2:0)
}
}
let res = MainClass(no1: 20)
let show = SubClass(no1: 30, no2: 50)
print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

扩展访问权限

你可以在条件允许的情况下对类、结构体、枚举进行扩展。扩展成员应该具有和原始类成员一致的访问级别。比如你扩展了一个公共类型，那么你新加的成员应该具有和原始成员一样的默认的internal访问级别。

或者，你可以明确申明扩展的访问级别（比如使用private extension）给该扩展内所有成员申明一个新的默认访问级别。这个新的默认访问级别仍然可以被单独成员所申明的访问级别所覆盖。

泛型访问权限

泛型类型或泛型函数的访问级别取泛型类型、函数本身、泛型类型参数三者中的最低访问级别。

实例

```
public struct TOS<T> {
var items = [T]()
private mutating func push(item: T) {
items.append(item)
}
mutating func pop() -> T {
return items.removeLast()
}
}
var tos = TOS<String>()
tos.push("Swift")
print(tos.items)
tos.push("泛型")
print(tos.items)
tos.push("类型参数")
print(tos.items)
tos.push("类型参数名")
print(tos.items)
let deletetos = tos.pop()
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "类型参数"]
["Swift", "泛型", "类型参数", "类型参数名"]
```

类型别名

任何你定义的类型别名都会被当作不同的类型，以便于进行访问控制。一个类型别名的访问级别不可高于原类型的访问级别。比如说，一个private级别的类型别名可以设定给一个public、internal、private的类型，但是一个public级别的类型别名只能设定给一个public级别的类型，不能设定给internal或private 级别的类型。

> *注意：这条规则也适用于为满足协议一致性而给相关类型命名别名的情况。*

实例

```
public protocol Container {
typealias ItemType
mutating func append(item: ItemType)
var count: Int { get }
subscript(i: Int) -> ItemType { get }
}
struct Stack<T>: Container {
// original Stack<T> implementation
var items = [T]()
mutating func push(item: T) {
items.append(item)
}
mutating func pop() -> T {
return items.removeLast()
}
// conformance to the Container protocol
mutating func append(item: T) {
self.push(item)
}
var count: Int {
return items.count
}
subscript(i: Int) -> T {
return items[i]
}
}
func allItemsMatch<
C1: Container, C2: Container
where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
(someContainer: C1, anotherContainer: C2) -> Bool {
// check that both containers contain the same number of items
if someContainer.count != anotherContainer.count {
return false
}
// check each pair of items to see if they are equivalent
for i in 0..<someContainer.count {
if someContainer[i] != anotherContainer[i] {
return false
}
}
// all items match, so return true
return true
}
var tos = Stack<String>()
tos.push("Swift")
print(tos.items)
tos.push("泛型")
print(tos.items)
tos.push("Where 语句")
print(tos.items)
var eos = ["Swift", "泛型", "Where 语句"]
print(eos)
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "Where 语句"]
["Swift", "泛型", "Where 语句"]
```

Swift 泛型