访问控制

4.2

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

访问控制 限制来自其他资源文件和模块的代码对你代码部分的访问。这个特性可以让你隐藏代码的实现细节，并且可以让你指定一个更喜欢的接口。代码可以通过这个接口被访问和使用。

你可以给各个类型（类、结构体和枚举）以及这些类型的属性、方法、初始化函数和下标分配特定的访问级别。协议可以被限制使用在某些上下文中，也可以被限制使用在全局的常量、变量和函数上。

除了提供各种级别的访问控制，Swift 为典型情景提供了默认的访问级别来减少指定显式访问控制级别的需求。其实，如果你在写一个单目标的应用，可能根本不需要声明显式访问控制级别。

> 注
>
> 为了简洁，代码中可以被设置访问控制的各个方面（属性、类型、函数等）在下面的章节中都被称作「实体」。

模块和资源文件

Swift 的访问控制模型是基于模块和资源文件的概念。

模块 是代码分发的独立单元——可以作为一个独立单元被构建和发布的 Framework 或 Application。模块可以被另一个模块用 Swift 的 `import` 关键字引用。

在 Swift 中，Xcode 的每个构建目标（比如 `app bundle` 或 `framework`）都被当作一个独立模块。如果你的团队把应用中部分代码放在一起作为一个独立的 `framework`——可能用来在多个应用中包含和复用代码——然后，当它被一个应用引用和使用，或被另一个 `framework` 使用的时候，在这个 `framework` 中定义的每个东西都将作为独立

模块的一部分。

资源文件 是模块中单独的 Swift 源代码文件（实际上是应用或 `framework` 中的单独文件）。虽然在不同的源文件中定义格子的类型是常见的，但一个单独的源文件可以包含不同类型、函数等的定义。

访问级别

Swift 为代码中的实体提供了五种不同的 *访问级别* 。这些访问级别与定义实体的源文件相关联 ， 也与源文件所属的模块相关联。

- *Open 访问* 和 *public 访问* 使实体可以在定义所在的模块的任何源文件中使用，也可以在引用了定义所在的模块的另一个模块中的源文件中使用。当为 `framework` 声明公用接口时常常使用 *open 访问* 或 *public 访问* 。
- *Internal 访问* 使实体可以在定义所在的模块的任何源文件中使用 ， 但是不能在这个模块之外的任何源文件中使用。当定义一个应用或 `framework` 的内部结构时， 常常使用*Internal 访问* 。
- *File-private 访问* 限制实体只能在它定义所在的源文件中使用。当一个特定功能片段在整个文件中使用时，用 *file-private 访问* 来隐藏它的实现细节。
- *Private 访问* 限制实体用于在同一个文件中附加声明、扩展声明。当一个特定功能片段仅仅被用在单个声明中时，用 *Private 访问* 隐藏它的实现细节。

Open 访问是最高的（限制性更弱的）访问级别，而 private 访问是最低的（限制性更强的）访问级别。

Open 访问仅仅应用于类和类的成员。它在几个方面区别于 public 访问：

- public 访问或其他更多限制访问级别修饰的类，仅可以被其定义所在模块中的类继承。
- public 访问或其他更多限制访问级别修饰的类成员，仅可以被其定义所在模块中的子类重载。
- Open 修饰的类可以被其定义所在模块或任何引用其定义所在模块的模块中的类继承。
- Open 修饰的类成员可以被其定义所在模块或任何引用该模块的模块中的子类重载。

标记一个类为 open 明显意味着了你考虑过其他模块中的代码将这个类作为父类的影响，进而也意味着你设计过这个类的代码。

访问级别的使用原则

iDev_01 翻译于 5个月

0 重译

由 Summer 审阅

iDev_01 翻译于 5个月

0 重译

由 Aufree 审阅

Swift 中的访问级别遵循一个总的使用原则：*任何实体都不能在另一个具有更低的（限制性更强的）访问级别的实体中定义*

举个例子：

- public 修饰的变量不能定义为一个 internal 、 file-private 或 private 修饰的类型。因为在使用该变量的所有地方都无法使用该类型。
- 函数不能拥有比它的参数类型、返回值类型更高的访问权限。因为在这些函数的组成类型对周围的代码不可用的情况下，函数是不能被使用。

下文详细介绍了这个使用原则在 Swift 语言中不同方面的具体实现。

默认访问级别

如果你自己没有指定一个特定的访问级别，代码中的所有实体（一些特定的情况例外，正如本节后面描述的那样）都有默认访问级别 internal。因此，在很多情况下你不需要在代码中指定显式访问级别。

对于单目标应用的访问级别

当你写一个简单的单目标应用，应用中的代码通常在应用中自包含，而不需要在你应用模块外可用。默认访问级别 internal 已经符合了这个需求。因此，你不需要指定一个自定义访问级别。然而，你可能想标记一部分代码为file private 或 private 以达到对应用模块的其他代码隐藏他们的实现细节的目的。

对于框架的访问权限

当你开发框架时，标记该框架面向公开的接口为 open 或 public，它才能被其他模块看到并访问到，比如一个引用框架的应用。这个面向公开的接口是对于该框架来说是应用编程接口（或 API ）。

> 注
>
> 框架的任何内部实现细节可以一直使用默认访问级别 internal ，或者如果你想针对框架的其他内部代码隐藏它们，可以标记为 private 或 file private。仅在你想把实体变成框架的 API 的一部分时，你需要标记它为 open 或 public。

对于单位测试目标的访问级别

当你写一个带有单元测试目标的应用时，为了测试，应用中的代码需要对那个模块设置为可用。默认情况下，仅仅被标记为 open 或 public 的实体对于其他模块是可访问的。然而，如果你用 `@testable` 属性为生产模块标记引用声明并用测试模式编译生产模块时，单元测

iDev_01 翻译于 5个月

0 重译

由 Aufree 审阅

iDev_01 翻译于 5个月

0 重译

由 Aufree 审阅

iDev_01 翻译于 5个月

0 重译

由 Aufree 审阅

试目标可用访问任何内部实体。

访问控制语法

通过将 `open` 、 `public` 、 `internal` 、 `fileprivate` 或 `private` 修饰符放置在实体的介绍词前的方式为实体定义访问级别：

```
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() {}
```

正如 Default Access Levels 中描述的那样，除非另有规定，默认访问级别为 internal。 这意味着 `SomeInternalClass` 和 `someInternalConstant` 可以不用写显式访问级别修饰符，并且依然有访问级别 internal：

```
class SomeInternalClass {}              // 隐式 internal
let someInternalConstant = 0            // 隐式 internal
```

自定义类型

如果你想为自定义类型指定显式访问级别，在你定义类型的地方这样做。它访问级别许可的地方，新类型可以被使用。举个例子，如果你定义了一个 file-private 类，它仅仅可以在它定义所在的源文件中作为一个属性的类使用，或者作为函数参数或返回值的类型使用。

类的访问控制级别可以影响类的*成员*（它的属性、方法、初始化函数和下标）的默认访问级别。如果你定义类的访问级别为 private 或 file private，它的成员的默认访问级别也将会是 private 或 file private。如果你定义类的访问级别为 internal 或 public （或用默认访问级别而不显式地指定访问级别），类成员的默认访问级别将会是 internal。

> 重点
>
> public 类型默认有 internal 成员而不是 public 成员。如果你想一个类成员为 public，你必须这样显式地标记它。这个需求确保类型面向公开的API 是你设置公开的事物，且避

免了错误地表现出类的内部工作方式。

```
public class SomePublicClass {                  // 显式 public 类
    public var somePublicProperty = 0            // 显式 public 类成员
    var someInternalProperty = 0                 // 隐式 internal 类成员
    fileprivate func someFilePrivateMethod() {}  // 显式 file-private 类成员
    private func somePrivateMethod() {}          // 显式 private 类成员
}

class SomeInternalClass {                       // 隐式 internal 类
    var someInternalProperty = 0                 // 隐式 internal 类成员
    fileprivate func someFilePrivateMethod() {}  // 显式 file-private 类成员
    private func somePrivateMethod() {}          // 显式 private 类成员
}

fileprivate class SomeFilePrivateClass {        // 显式 file-private 类
    func someFilePrivateMethod() {}              // 隐式 file-private 类成员
    private func somePrivateMethod() {}          // 显式 private 类成员
}

private class SomePrivateClass {                // 显式 private class
    func somePrivateMethod() {}                  // 隐式 private 类成员
}
```

元组类型

元组类型的访问权限是元组中使用的全部类型的访问级别中限制性最强的。举个例子，如果你组合两种不同类型的元组，一个带有 internal 访问，另一个带有 private 访问，合成的元组类型的访问级别将是 private.

> 注意
>
> 元组类型没有像类、结构体、枚举、和函数那样的单独的定义。元组类型的访问级别是在使用元组时自动推导出来的，它不能被显式地指定。

函数类型

函数类型的访问级别被计算为函数参数类型和返回值类型的限制性最强的访问级别。 如果函数的计算访问级别和上下文默认的不匹配，你必须显式地指定访问级别作为函数定义的一部分。

下面的例子定义了一个叫做 `someFunction()` 的全局函数。这个函数没有为其本身提供具体的访问级别修饰符。你可能认为这个函数有默认访问级别「internal」，但事实并非如此。事实上，如下所示的 `someFunction()` 不能编译：

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

函数的返回值是由两个定义在自定义类型中的自定义类型组成的元组。其中一个定义为 internal，另一个定义为 private。因此，合成的元组的整体访问级别为 private（元组的组成类型的最小访问级别）。

因为函数的返回值类型是 private，为了函数声明有效，你必须用 `private` 修饰符标记函数的整体访问级别：

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

用 `public` 或 `internal` 修饰符或用默认设置的 internal 标记的 `someFunction()` 定义是无效的，因为函数的 public 或 internal 使用者可能没有对被用于函数的返回值类型的 private 类型有合适的访问权限。

枚举类型

枚举中的独立成员自动使用该枚举类型的访问级别。你不能给独立的成员指明一个不同的访问级别。

在下面的例子中，`CompassPoint` 有一个指明的 public 访问级别。因此，`north`，`south`，`east`，和 `west` 也具有 public 访问级别：

```
public enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

原始值和关联值

枚举定义中的原始值和关联值使用的类型必须有一个不低于枚举的访问级别。例如，你不能

使用一个 private 类型作为一个 internal 访问级别的枚举类型中的原始值类型。

嵌套类型

定义在 private 类型中的嵌套类型自动访问级别为 private 。定义在 file-private 类型中的嵌套类型自动访问级别为 file-private 。定义在 public 类型或者 internal 中的嵌套类型自动访问级别为 internal 。如果你想要在 public 类型中的嵌套类型能够被外部访问，你必须要显示的声明这个嵌套类型为 public。

子类化

你可以在符合当前访问上下文中子类化任何类。子类不能拥有高于其父类的访问等级——举个例子，你不能写父类为 internal 类型而其子类为 public 类型的代码。

除此之外，在确定可见的访问权限上下文中，你可以重写任何类成员（方法，属性，初始化器或者下标）

重写类成员可以使其比父类版本更加开放。比如下面的这个例子，类 `A` 的访问级别类型为 public 并且有一个 file-private 类型方法 `someMethod()` ，类 `B` 为 类 `A` 的子类，且其访问级别为降低了的 "internal" 。尽管如此，类 `B` 提供了一个访问级别为 "internal" 的重写 `someMethod()` 方法，它比父类的原始方法版本访问级别 *更高* 。

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {}
}
```

甚至是在父类成员访问权限低于子类的成员中调用父类成员都是有效的，只要对父类成员的调用发生在一个允许的访问级别上下文中（就是说，在同源文件中 file-private 访问级别的成员可以调用父类中该成员，或者在同模块中的 internal 成员可以调用父类该成员）。

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {
        super.someMethod()
    }
```

```
}
```

因为父类 `A` 和 子类 `B` 都是定义在同一个源文件中，所以在类 `B` 中实现的 `someMethod()` 调用 `super.someMethod()` 是有效的。

常量，变量，属性和下标

一个常量，变量或者属性的访问级别不能比其所代表的类型更高。举个例子，写一个代表了 private 类型的 public 属性是无效的。同样的，一个下标的访问级别不能比其索引或者返回值的访问级别更高。

如果一个常量，变量，属性或者下标代表的是一个 private 类型，则常量，变量，属性或者下标都必须被标记上 `private` ：

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

赋值方法和取值方法

常量，变量，属性和下标的赋值和取值方法能够自动的接收和它们访问级别一致的常量，变量，属性和下标。

你可以设置一个比取值方法更 *低* 访问级别的赋值方法来进行约束属性或下标的读-写范围。你可以在 `var` 或 `subscript` 前加入修饰符 `fileprivate(set)` ，`private(set)` ， 或 `internal(set)` 来给其分配一个较低的访问级别

> 注意
>
> 该规则适用于存储和计算属性。尽管你并没有显式的对存储属性写出赋值或取值方法，但 Swift 仍然会为你的存储属性所存储的值隐式合成赋值和取值方法。使用 `fileprivate(set)` ， `private(set)` ， 或 `internal(set)` 修饰符去改变这个合成赋值方法的访问级别与显式的写出计算属性的赋值方法取得的效果是完全一致的。

下面的事例定义了一个被称为 `TrackedString` 的结构体，它持续的记录了字符串属性修改的次数

```
struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
```

PJHubs 翻译于 4个月

0 重译

由 Summer 审阅

PJHubs 翻译于 4个月

0 重译

由 Summer 审阅

```
        didSet {
            numberOfEdits += 1
        }
    }
}
```

`TrackedString` 结构体定义了一个被称为 `value` 字符串存储属性，并且初始化为 `""` （空字符串）。这个结构体还定义了一个被称为 `numberOfEdits` 的整形存储属性，它被用于记跟踪 `value` 修改的次数。该修改跟踪是由 `didSet` 属性监听 `value` 属性实现的，每次当 `value` 属性赋新值时都会递增 `numberOfEdits`

`TrackedString` 结构体和 `value` 属性都不提供访问级别修饰符的显式修改方法，因此它们二者的访问级别都与内部的默认访问级别对应。但是， `numberOfEdits` 属性的访问级别被标记为了 `private(set)` 修饰符，表明了该属性的取值方法在内部依然还是默认访问级别，但该属性的赋值代码只能在 `TrackedString` 的结构体中。这就可在 `TrackedString` 结构体内部去修改 `numberOfEdits` 属性值，但在该结构体定义之外去使用时则表现出只读属性。

如果你创建一个 `TrackedString` 实例并且修改了几次它的字符串值，你可以看到 `numberOfEdits` 属性值的更新和与之对应的修改值：

```
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment numberOfEdits."
stringToEdit.value += " So will this one."
print("The number of edits is \(stringToEdit.numberOfEdits)")
// 打印 "The number of edits is 3"
```

尽管你可以在另一个源文件中查询 `numberOfEdits` 属性当前的值，但你不能在另一个源文件中 *修改* 该属性。该约束保护了 `TrackedString` 编辑跟踪功能的实现细节，同时还提供了访问该功能的便捷方法。

注意，如果需要的话，你可以给赋值和取值方法显式设置访问级别。下面的例子展示了其中一个版本的 `TrackedString` 结构体代码，该版本中定义了显式公开访问级别。该结构体成员（包括 `numberOfEdits` 属性）因此拥有了默认内部访问级别。通过结合 `pulic` 和 `private(set)` 访问级别修饰符，你可以让该结构体的 `numberOfEdits` 属性取值方法为公开的，而它的赋值方法是私有的。

```
public struct TrackedString {
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
```

```
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
}
```

初始化器

自定义初始化器能够分配小于或等于当前它们初始化类型的访问等级。只有必须初始化器是例外 （定义在这 Required Initializers）。必须初始化器的访问级别要与当前类一致。

与函数和方法参数一样，初始化器的参数访问级别类型不能比该初始化器本身拥有的访问级别更低。

默认初始化器

如默认初始化器中所说，Swift 自动提供了一个无参 *默认初始化器* 给任何结构或基类，该初始化器给任何结构或者基类的所有属性提供了默认值，并且其本身提供了不止一个初始化器。

默认初始化器拥有和初始化对象类型相同的访问级别，除非该类型为 `public` 。当初始化对象类型被定义为 `public` 时，它的默认初始化器被认为是 internal 级别。如果你想要在另外一个模块中使用无参初始化器去初始化 public 类型参数，你必须显式的提供一个 public 类型的无参初始化器作为类型定义的一部分

结构体类型的默认成员初始化器

如果结构体类型中的任何一个存储属性为 private 类型，则该结构体类型的默认成员初始化器被认为是 private 类型。同样的，如果该结构体类型的任何一个存储属性是 file private 类型，则其初始化器为 file private 类型。否则，其初始化器的访问级别为 internal 。

与上文的默认初始化器一样，当在另外一个模块中，如果你想要成员初始化器初始化 public 的结构体类型时，你必须要提供一个 public 类型的成员初始化器作为该结构体类型定义的一部分。

协议

如果要为协议类型分配显式访问级别，请在定义协议时执行此操作。这可以使你创建的协议只能在特定的上下文中访问。

协议定义中每个要求的访问级别自动设置为与协议相同的访问级别。你不能将协议要求设置为与协议不同的访问级别。这可确保在采用该协议的任何类型上都可以看到所有协议的要求。

> 注意
>
> 如果定义一个 public 协议，则协议的要求在实施时也需要是 public 访问级别。这个行为与其它的类型不同，其它的 public 类型的成员默认访问级别是 internal 。

协议继承

如果你定义一个新的协议是从现有协议中继承而来，那么新协议的访问级别最多可以与其继承的协议相同。例如，你无法编写一个从 internal 协议继承过来的 public 协议。

协议一致性

一个类型可以符合比它本身访问级别更低的协议。 例如，你可以定义一个能在别的模块中使用的 public 类型，但其与 internal 协议的一致性只能在 internal 协议定义的模块中使用。

符合特定协议的类型上下文是类型和协议访问级别的最小值。如果类型是 public 类型，但它符合的协议是 internal 类型，那么该类型与协议的一致性也是 internal 的。

当你编写或扩展一个符合某协议的类型时，你必须确保它实现的每一个协议的要求至少具有与该协议类型一致的访问级别。例如，如果一个 public 类型去符合一个 internal 协议，那么每个协议要求的类型实现必须至少为「internal」。

> 注意
>
> 在 Swift 中，与 Objective-C 一样，协议一致性是全局的 -- 类型不可能在同一个程序中以两种不同的方式符合协议。

扩展

你可以在类，结构体或枚举的任何可访问上下文中扩展它们。在扩展中添加的任何类型成员都与原始类型声明中的成员具有相同的默认访问级别。如果扩展 public 或 internal 类型，则新添加的类型成员的默认访问级别为 internal 。如果扩展一个 file-private 类型，则新添加的任何类型成员具有 file private 的默认访问级别。如果扩展 private 类型，则新添加的类型成员默认访问级别为 private 。

或者，你可以使用显示访问级别修饰符（例如 `private extension` ）标记扩展，用来

给扩展中定义的成员重新设置默认的访问级别。这个新的默认值仍然可以在单个类型成员的扩展中被覆盖。

如果你使用扩展来添加协议一致性，则无法为扩展提供显式访问级别修饰符。相反，协议本身的访问级别为扩展中的每个协议要求的实现提供默认的访问级别。

扩展中的私有成员

与类、结构体或枚举值写在同一个文件中的扩展的行为和原始类型声明的部分一样。因此，你可以：

- 在原始声明中声明一个私有成员，并在同一个文件的扩展中访问它。
- 在一个扩展中声明一个私有成员，并在同一个文件中的另一个扩展中访问它。
- 在一个扩展中声明一个私有成员，并在同一个文件中的原始声明里访问它。

这意味着无论你的类型是否具有私有实体，你都可以以相同的方式使用扩展来组织代码。举个例子，给出以下简单协议：

```
protocol SomeProtocol {
    func doSomething()
}
```

你可以使用扩展来添加协议一致性，像这样：

```
struct SomeStruct {
    private var privateVariable = 12
}

extension SomeStruct: SomeProtocol {
    func doSomething() {
        print(privateVariable)
    }
}
```

泛型类型

任何泛型参数或泛型函数的访问级别是该泛型参数或函数自身访问级别的最小值，以及对其类型参数的任何类型约束的访问级别。

类型别名

hisoka 翻译于 4个月前

0 重译

由 Summer 审阅

为了达到访问控制的目的，你定义的任何类型别名会被看作是不同类型。类型别名可以拥有小于或等于其所代表类型的访问级别。举个例子，一个 private 的类型别名可以代表 private ， file-private ， internal ， public 或 open 类型。但是 public 级别的类型别名不能代表 internal ， file-private ， 或 private 类型。

> 注意
>
> 以上规则也适用于满足协议一致性中的 associated 类型的类型别名，

PJHubs 翻译于 4个月

0 重译

由 Summer 审阅

上一篇 下一篇

参与译者：6

更多职位

讨论数量: 0

发起讨论 只看当前版本讨论

暂无话题~