访问控制

cnswift.org/access-control

*访问控制*限制其他源文件和模块对你的代码的访问。这个特性允许你隐藏代码的实现细节，并指定一个偏好的接口让其他代码可以访问和使用。

你可以给特定的单个类型 (类，结构体和枚举)设置访问级别，比如说属性、方法、初始化器以及属于那些类型的下标。协议可以限制在一定的范围内使用，就像全局常量，变量，函数那样。

除了提供各种级别的访问控制，Swift 为典型场景提供默认的访问级别，减少了显式指定访问控制级别的需求。 事实上，如果你编写单目标应用程序，你可能根本不需要显式指定访问控制级别。

> 注意
>
> 简洁起见，代码中可以设置访问级别的部分(属性，类型，函数等)在下面的章节称为“实体”。

模块和源文件

Swift 的访问控制模型基于模块和源文件的概念。

*模块*是单一的代码分配单元——一个框架或应用程序会作为的独立的单元构建和发布并且可以使用 Swift 的
import 关键字导入到另一个模块。

Xcode 中的每个构建目标（例如应用程序包或框架）在 Swift 中被视为一个独立的模块。如果你将应用程序的代码作为独立的框架组合在一起——或许可以在多个应用程序中封装和重用该代码——那么当在一个应用程序中导入和使用时，在该框架中定义的所有内容都将作为独立模块的一部分 ，或是当它在另一个框架中使用时。

*源文件*是一个模块中的单个 Swift 源代码文件（实际上，是一个应用程序或是框架中的单个文件）。虽然通常在单独源文件中定义单个类型，但是一个源文件可以包含多个类型。函数等的定义。

访问级别

Swift 为代码的实体提供个五个不同的访问级别。这些访问级别和定义实体的源文件相关，并且也和源文件所属的模块相关。

- *Open 访问* 和 public 访问 允许实体被定义模块中的任意源文件访问，同样可以被另一模块的源文件通过导入该定义模块来访问。在指定框架的公共接口时，通常使用 open 或 public 访问。 open 和 public 访问 之间的区别将在之后给出；
- *Internal 访问* 允许实体被定义模块中的任意源文件访问，但不能被该模块之外的任何源文件访问。通常在定义应用程序或是框架的内部结构时使用。
- *File-private 访问* 将实体的使用限制于当前定义源文件中。当一些细节在整个文件中

使用时，使用 file-private 访问隐藏特定功能的实现细节。
- *private 访问* 将实体的使用限制于封闭声明中。当一些细节仅在单独的声明中使用时，使用 private 访问隐藏特定功能的实现细节。

open 访问是最高的（限制最少）访问级别，private 是最低的（限制最多）访问级别。

open 访问仅适用于类和类成员，它与 public 访问区别如下：

- public 访问，或任何更严格的访问级别的类，只能在其定义模块中被继承。
- public 访问，或任何更严格访问级别的类成员，只能被其定义模块的子类重写。
- open 类可以在其定义模块中被继承，也可在任何导入定义模块的其他模块中被继承。
- open 类成员可以被其定义模块的子类重写，也可以被导入其定义模块的任何模块重写。

显式地标记类为 open 意味着你考虑过其他模块使用该类作为父类对代码的影响，并且相应地设计了类的代码。

访问级别的指导准则

Swift 中的访问级别遵循一个总体指导准则：*实体不可以被更低（限制更多）访问级别的实体定义。*

例如：

- 一个 public 的变量其类型的访问级别不能是 internal, file-private 或是 private，因为在使用 public 变量的地方可能没有这些类型的访问权限。
- 一个函数不能比它的参数类型和返回类型访问级别高，因为函数可以使用的环境而其参数和返回类型却不能使用。

这个准则对此语言其他方面的明确含义已经详细的列在下面。

默认访问级别

如果你不指明访问级别的话，你的代码中的所有实体（以及本章后续提及的少数例外）都会默认为 internal 级别。因此，大多数情况下你不需要明确指定访问级别。

单目标应用的访问级别

当你编写一个简单的单目标应用时，你的应用中的代码都是在本应用中使用的并且不会在应用模块之外使用。默认的 internal 访问级别已经匹配了这种需求。因此，你不需要明确自定访问级别。但你可能会将代码的一些部分标注为 file private 或private 以对模块中的其他代码隐藏它们的实现细节。

框架的访问级别

当你开发一个框架时，将该框架的面向公众的接口标注为 open 或 public，这样它就能被其他的模块看到或访问，比如导入该框架的应用。这个面相公众的接口就是该框架的应用编程接口（API）。

> 注意
>
> 你框架的任何内部实现细节仍可以使用 internal 默认访问级别，如果你想从框架的其他部分隐藏细节也可以将它们标注为 private 或 file private 。仅当你想将它设为框架的API时你才能将实体标注为 open 或 public 。

单元测试目标的访问级别

当你在写一个有单元测试目标的应用时，你的代码应该能被模块访问到以进行测试。默认情况下只有标注为 open 或 public 的才可以被其他模块访问。但是，如果你使用 @testable 属性标注了导入的生产模块并且用使能测试的方式编译了这个模块，单元测试目标就能访问任何 internal 的实体。

访问控制语法

通过在实体的引入之前添加
open ，
public ，
internal ，
fileprivate ，或
private 修饰符来定义访问级别。

```
1  public class SomePublicClass {}
2  internal class SomeInternalClass {}
3  fileprivate class SomeFilePrivateClass {}
4  private class SomePrivateClass {}
5  public var somePublicVariable = 0
6  internal let someInternalConstant = 0
7  fileprivate func someFilePrivateFunction() {}
8  private func somePrivateFunction() {}
9
```

除非已经标注，否则都会使用默认的 internal 访问级别，这一点在默认访问级别一节已经说明。这意味着
SomeInternalClass 和
someInternalConstant 不需要指明访问级别也会是 internal 级别。

```
1  class SomeInternalClass {}          // implicitly internal
2  let someInternalConstant = 0      // implicitly internal
3
```

自定类型

如果你想给自定类型指明访问级别，那就在定义时指明。只要访问级别允许，新类型就可以被使用。例如，你定义了一个 file-private 的类，它就只能在定义文件中被当作属性类型、函数参数或返回类型使用。

类型的访问控制级别也会影响它的成员的默认访问级别（它的属性，方法，初始化方法，下标）。如果你将类型定义为 private 或 file private 级别，那么它的成员的默认访问级别也会是 private 或file private。如果你将类型定义为 internal 或 public级别（或直接使用默

认级别而不显式指出），那么它的成员的默认访问级别会是 internal 。

> 重要
>
> public 的类型默认拥有 internal 级别的成员，而不是 public。如果你想让其中的一个类型成员是 public 的，你必须按实示例代码指明。这个要求确保类型的面向公众的 API 是你选择的，并且可以避免将类型的内部工作细节公开成 API 的失误。

```
public class SomePublicClass {                   // explicitly public class
    public var somePublicProperty = 0            // explicitly public class member
    var someInternalProperty = 0                 // implicitly internal class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-private class
member
    private func somePrivateMethod() {}          ／/ explicitly private class member
}
class SomeInternalClass {                        // implicitly internal class
    var someInternalProperty = 0                 // implicitly internal class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-private class
member
    private func somePrivateMethod() {}          // explicitly private class member
}
fileprivate class SomeFilePrivateClass {         // explicitly file-private class
    func someFilePrivateMethod() {}              // implicitly file-private class member
    private func somePrivateMethod() {}          // explicitly private class member
}
private class SomePrivateClass {                 // explicitly private class
    func somePrivateMethod() {}                  // implicitly private class member
}

```

元组类型

元组类型的访问级别是所有类型里最严格的。例如，如果你将两个不同类型的元素组成一个元组，一个元素的访问级别是 internal，另一个是 private，那么这个元组类型是 private 级别的。

> 注意
>
> 元组类型不像类、结构体、枚举和函数那样有一个单独的定义。元组类型的访问级别会在使用的时候被自动推断出来，不需要显式指明。

函数类型

函数类型的访问级别由函数成员类型和返回类型中的最严格访问级别决定。如果函数的计算访问级别与上下文环境默认级别不匹配，你必须在函数定义时显式指出。

下面的例子定义了一个称为
someFunction() 的全局函数，而没有指明它的访问级别。你或许以为它会是默认的
“internal” 级别，但事实不是这样。这样的
someFuniction()是无法通过编译的：

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

这个函数的返回类型是一个由两个在自定义类型里定义的类组成的元组。其中一个类是“internal”级别的，另一个是“private”。因此，这个元组的访问级别是“private”（元组成员的最严级别）。

由于返回类型是 private 级别的，你必须使用
private 修饰符使其合法:

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

使用
public 或
internal 标注
someFunction() 的定义是无效的，使用默认的 internal 也是无效的，7的函数可能无法访问到 private 的函数返回值。

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”级别。里面的成员
north ,
south ,
east , 和
west 因此是“public”：

```
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 级别的枚举类型中的原始值类型。

嵌套类型

private 级别的类型中定义的嵌套类型自动为 private 级别。fileprivate 级别的类型中定义的嵌套类型自动为 fileprivate 级别。public 或 internal 级别的类型中定义的嵌套类型自动为 internal 级别。如果你想让嵌套类型是 public 级别的，你必须将其显式指明为 public。

子类

你可以继承任何类只要是在当前可以访问的上下文环境中。但子类不能高于父类的访问级别，例如，你不能写一个 internal 父类的 public 子类。

而且，你可以重写任何类成员（方法，属性，初始化器或下标），只要是在确定的访问域中是可见的。

重写可以让一个继承类成员比它的父类中的更容易访问。在下例中，public 级别的类
A 有一个 fileprivate 级别的
someMethod() 函数。
B 是
A 的子类，有一个降低的“internal”级别。但是，类
B 对
someMethod() 函数进行了重写即改为“internal”级别，这比
someMethod() 的原本实现级别*更高*：

```
1  public class A {
2          fileprivate func someMethod() {}
3  }
4  internal class B: A {
5          override internal func someMethod() {}
6  }
```

子类成员调用父类中比子类更低访问级别的成员，只要这个调用发生在一个允许的访问级别上下文中（即对 fileprivate 成员的调用要求父类在同一个源文件中，对 internal 成员的调用要求父类在同一个模块中）：

```
1  public class A {
2          fileprivate func someMethod() {}
3  }
4  internal class B: A {
5          override internal func someMethod() {
6          super.someMethod()
7          }
8  }
```

因为父类
A 和子类
B 定义在同一个源文件中，那么
B 类可以在
someMethod() 中调用父类的
someMethod()。

常量，变量，属性和下标

常量、变量、属性不能拥有比它们类型更高的访问级别。例如，你不能写一个public 的属

性而它的类型是 private 的。类似的，下标也不能拥有比它的索引类型和返回类型更高的访问级别。

如果常量、变量、属性或下标由private类型组成，那么常量、变量、属性或下标也要被标注为
private ：

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

Getters 和 Setters

常量、变量、属性和下标的 getter 和 setter 自动接收它们所属常量、变量、属性和下标的访问级别。

你可以给 setter 函数一个比相对应 getter 函数更低的访问级别以限制变量、属性、下标的读写权限。你可以通过在
var 和
subscript 的置入器之前书写
fileprivate(set) ,
private(set) , 或
internal(set) 来声明更低的访问级别。

> 注意
>
> 这个规则应用于存储属性和计算属性。即使你没有给一个存储属性书写一个明确的 getter 和 setter，Swift 会为你合成一个 getter 和 setter 以访问到存储属性的隐式存储。使用
> fileprivate(set) ,
> private(set) 和
> internal(set) 可以改变这个合成的 setter 的访问级别，同样也可以改变计算属性的访问级别。

下面的例子定义了一个称为
TrackedString 的结构体，它保持追踪一个字符串属性的修改次数：

```
1  struct TrackedString {
2          private(set) var numberOfEdits = 0
3          var value: String = "" {
4          didSet {
5              numberOfEdits += 1
6          }
7          }
8  }
```

TrackedString 结构体定义了一个可存储字符串的属性
value ，它又一个初始值
"" （空字符串）。这个结构图同样定义了一个可存储整数的属性
numberOfEdits ，它被用于记录
value 的修改次数。这个记录由
value 属性中的didset属性实现，它会增加

numberOfEdits 的值一旦
value 被设为一个新值。

TrackedString 结构体和
value 属性都没有显式指出访问级别修饰符，因此它们都遵循默认的
internal 级别。
numberOfEdits 属性的访问级别已经标注为
private(set) 以说明这个属性的 getter 是默认的 internal 级别，但是这个属性只能被
TrackedString 内的代码设置。这允许
TrackedString 在内部修改
numberOfEdits 属性，而且可以展示这个属性作为一个只读属性当在结构体定义之外使用时——包括
TrackedString 的扩展。

如果你创建了一个
TrackedString 的实例并修改了几次字符串的值，你可以看到
numberOfEdits 属性的值更新到匹配修改的次数：

```
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)")
// Prints "The number of edits is 3"
```

尽管你可以从别的源文件中询问到
numberOfEdits 属性的当前值，但你不能从别的源文件中修改该属性的值。这个限制保护了
TrackedString 编辑追踪功能的实现细节，并同时为该功能的一个方面提供方便的访问。

你若有必要也可以显式指明 getter 和 setter方法。下面的例子提供了一个定义为
public 级别的
TrackedString 结构体。结构体成员（包括
numberOfEdits 属性）因此有一个默认的 internal 级别。你可以设置
numberOfEdits 属性的getter方法为 public，setter 方法为 private 级别，通过结合
public 和
private(set) 访问级别修饰符：

```
public struct TrackedString {
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
}
```

初始化器

我们可以给自定义初始化方法设置一个低于或等于它的所属的类的访问级别。唯一的例外是必要初始化器（定义在必要初始化器）。必要初始化器必须和它所属类的访问级别一致。

就像函数和方法的参数一样，初始化器的参数类型不能比初始化方法的访问级别还低。

默认初始化器

正如默认初始化器中描述的那样，Swift 自动为任何结构体和类提供一个无参数的默认初始化方法，以给它的属性提供默认值但不会提供给初始化器自身。

默认初始化方法与所属类的访问级别一致，除非该类型定义为
public 。如果一个类定义为
public ，那么默认初始化方法为 internal 级别。如果你想一个 public 类可以被一个无参初始化器初始化当在另一个模块中使用时，你必须显式提供一个 public 的无参初始化方法。

结构体的默认成员初始化器

如果结构体的存储属性时 private 的，那么它的默认成员初始化方法就是 private 级别。如果结构体的存储属性时 file private 的，那么它的默认成员初始化方法就是 file private 级别。否则就是默认的 internal 级别。正如以上默认初始化的描述，如果你想在另一个模块中使用结构体的成员初始化方法，你必须提供在定义中提供一个 public 的成员初始化方法。

协议

如果你想给一个协议类型分配一个显式的访问级别，那就在定义时指明。这让你创建的协议可以在一个明确的访问上下文中被接受。

协议定义中的每一个要求的访问级别都自动设为与该协议相同。你不能将一个协议要求的访问级别设为与协议不同。这保证协议的所有要求都能被接受该协议的类型所见。

注意如果你定义了一个 public 的协议，该协议的规定要求在被实现时拥有一个 public 的访问级别。这个行为不同于其他类型，一个 public 的类型的成员时 internal 访问级别。

协议继承

如果你定义了一个继承已有协议的协议，这个新协议最高与它继承的协议访问级别一致。例如你不能写一个 public 的协议继承一个 internal 的协议。

协议遵循

类型可以遵循更低访问级别的协议。例如，你可以定义一个可在其他模块使用的 public 类型，但它就只能在定义模块中使用如果遵循一个 internal 的协议。

遵循了协议的类的访问级别取这个协议和该类的访问级别的最小者。如果这个类型是 public 级别的，它所遵循的协议是 internal 级别，这个类型就是 internal 级别的。

当你写或是扩张一个类型以遵循协议时，你必须确保该类按协议要求的实现方法与该协议的访问级别一致。例如，一个 public 的类遵循一个 internal 协议，该类的方法实现至少是 “internal” 的。

> 注意
>
> 在 Swift 和 Objective-C 中协议遵循是全局的——一个类不可能在一个程序中用不同方法遵循一个协议。

扩展

你可以在任何可访问的上下文环境中对类、结构体、或枚举进行扩展。在扩展中添加的任何类型成员都有着被扩展类型相同的访问权限。如果你扩展一个公开或者内部类型，你添加的任何新类型成员都拥有默认的内部访问权限。如果你扩展一个文件内私有的类型，你添加的任何新类型成员都拥有默认的私有访问权限。如果你扩展一个私有类型，你添加的任何新类型成员都拥有默认的私有访问权限。

或者，你可以显式标注扩展的访问级别（例如，
private extension ）已给扩展中的成员设置新的默认访问级别。这个默认同样可以在扩展中为单个类型成员重写。

你不能给用于协议遵循的扩展显式标注访问权限修饰符。相反，在扩展中使用协议自身的访问权限作为协议实现的默认访问权限。

扩展中的私有成员

在同一文件中的扩展比如类、结构体或者枚举，可以写成类似多个部分的类型声明。你可以：

- 在原本的声明中声明一个私有成员，然后在同一文件的扩展中访问它；
- 在扩展中声明一个私有成员，然后在同一文件的其他扩展中访问它；
- 在扩展中声明一个私有成员，然后在同一文件的原本声明中访问它。

这样的行为意味着你可以和组织代码一样使用扩展，无论你的类型是否拥有私有成员。比如说，假设下面这样的简单协议：

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

你可以使用扩展来添加协议遵循，比如这样：

```
struct SomeStruct {
    private var privateVariable = 12
}
extension SomeStruct: SomeProtocol {
    func doSomething() {
        print(privateVariable)
    }
}

```

泛型

泛指类型和泛指函数的访问级别取泛指类型或函数以及泛型类型参数的访问级别的最小值。

类型别名

任何你定义的类型同义名都被视为不同的类型以进行访问控制。一个类型同义名的访问级别不高于原类型。例如，一个 private 的类型同义名可联系到 private ，file-private ，internal ，public 或open 的类型，但 public 的类型同义名不可联系到 internal ，file-private 或 private 类型。

> 注意
>
> 这条规则适用于为满足协议遵循而给类型别名关联值的情况。