不透明类型

cnswift.org/opaquetypes

具有不透明返回类型的函数或者方法会隐藏它返回值的类型信息。相对于提供具体的类型作为函数的返回类型，返回值根据它支持的协议进行描述。隐藏类型信息在模块和调用模块的代码之间的边界处很好用，因为返回值的具体类型可以保持私有。不同于返回一个协议类型的值，不透明类型保持了类型的身份——编译器可以访问类型的信息，但模块的客户端不能。

不透明类型解决的问题

比如说，你在写一个模块来使用 ASCII 绘制图像。最字符化一个 ASCII 图形的基础是 draw() 函数，它返回字符串来表达那个图形，所以你可以把它作为 Shape 协议的需求：

```
protocol Shape {
    func draw() -> String
}

struct Triangle: Shape {
    var size: Int
    func draw() -> String {
        var result = [String]()
        for length in 1...size {
            result.append(String(repeating: "*", count: length))
        }
        return result.joined(separator: "\n")
    }
}
let smallTriangle = Triangle(size: 3)
print(smallTriangle.draw())
// *
// **
// ***
```

你可以使用范型来实现操作比如垂直翻转图形，如同下面显示的代码那样。总之，这里有一个重要的限制就是：反转了的结果返回了与我们创建范型完全一致的类型。

```
struct FlippedShape<T: Shape>: Shape {
    var shape: T
    func draw() -> String {
        let lines = shape.draw().split(separator: "\n")
        return lines.reversed().joined(separator: "\n")
    }
}
let flippedTriangle = FlippedShape(shape: smallTriangle)
print(flippedTriangle.draw())
// ***
// **
// *
```

这个实现定义了一个 JoinedShape<T: Shape, U: Shape> 结构体，它能把两个图形垂直地结合在一起，如同下面的代码显示，一个翻转了的三角形与另一个三角形结合后返回的结果类型类似 JoinedShape<FlippedShape<Triangle>, Triangle>

```
struct JoinedShape<T: Shape, U: Shape>: Shape {
    var top: T
    var bottom: U
    func draw() -> String {
        return top.draw() + "\n" + bottom.draw()
    }
}
let joinedTriangles = JoinedShape(top: smallTriangle, bottom: flippedTriangle)
print(joinedTriangles.draw())
// *
// **
// ***
// ***
// **
// *
```

暴露创建图形允许的类型的具体信息并不意味着由于声明完整的返回类型就得给 ASCII 图形模块的公开接口泄露出去。模块内的代码可能使用不同方式来构建相同的类型，其他模块外的使用图形的代码不应该考虑转换的具体实现。包装类型比如 JoinedShape 和 FlippedShape 并不关心模块的用户，且应该是不可见的。模块的公开接口由一系列的操作做成，比如拼接和翻转图形，这些操作返回另一个 Shape 值。

返回一个不透明类型

你可以把不透明类型想象成一个范型的反义词。范型类型让代码调用的函数根据实现决定函数形式参数和返回值的类型。举例来说，下面代码的函数返回的类型基于其调用：

```
func max<T>(_ x: T, _ y: T) -> T where T: Comparable { ... }
```

调用 max(_:_:) 的代码来选择 x 还是 y 的值，并且这些值的类型决定了 T 的具体类型。调用代码可以使用任何遵循 Comparable 协议的类型。函数内的代码则以范型的方式写就所以它可以处理调用代码提供的任意类型。 max(_:_:) 使用的实现仅对所有 Comparable 类型生效。

这些角色对于拥有不透明类型返回类型的函数来说恰好相反。不透明类型允许函数实现来根据调用它的代码抽象出返回值的类型。比如说，下面例子中的函数返回了一个梯形而没有暴露图形的类型。

```
struct Square: Shape {
    var size: Int
    func draw() -> String {
        let line = String(repeating: "*", count: size)
        let result = Array<String>(repeating: line, count: size)
        return result.joined(separator: "\n")
    }
}

func makeTrapezoid() -> some Shape {
    let top = Triangle(size: 2)
    let middle = Square(size: 2)
    let bottom = FlippedShape(shape: top)
    let trapezoid = JoinedShape(
        top: top,
        bottom: JoinedShape(top: middle, bottom: bottom)
    )
    return trapezoid
}
let trapezoid = makeTrapezoid()
print(trapezoid.draw())
// *
// **
// **
// **
// **
// *
```

makeTrapezoid() 函数在这个例子中声明了它的返回类型为 some Shape ；结果就是，函数返回一个遵循 Shape 协议的类型，而不需要标明具体类型。这样写 makeTrapezoid() 能让它在公开接口中表达最基本的期望——返回的值是一个图形——不需要特别明确图形是某个公开接口返回的类型。这个实现使用了两个三角形和一个方形，但函数可以重写成用各种方法绘制一个梯形却无需改变它的返回类型。
这个例子点明了不透明返回类型类似范型的反例。 makeTrapezoid() 内部代码可以返回它需要的任意类型，只要类型遵循 Shape 协议，就像调用范型函数的代码那样。调用函数的代码需要写成范型的方式，就像实现一个范型函数，这样它就可以处理任意 makeTrapezoid() 返回的 Shape 值了。

你也可以用范型来结合不透明返回类型。下面代码中的函数都返回遵循 Shape 协议的某类型的值。

```
func flip<T: Shape>(_ shape: T) -> some Shape {
    return FlippedShape(shape: shape)
}
func join<T: Shape, U: Shape>(_ top: T, _ bottom: U) -> some Shape {
    JoinedShape(top: top, bottom: bottom)
}

let opaqueJoinedTriangles = join(smallTriangle, flip(smallTriangle))
print(opaqueJoinedTriangles.draw())
// *
// **
// ***
// ***
// **
// *
```

opaqueJoinedTriangles 的值在这个例子中与前文不透明类型解决的问题小节中范型例子里的 joinedTriangles 一致。总之，与那个例子中值不同的是， flip(_:) 和 join(_:_:) 包装了范型图形操作返回的具体类型为不透明类型，这就避免了那些类型可见。由于它们依赖的是范型，所以两个函数都是范型，并且 FlippedShape 和 JoinedShape 所需要的类型信息由类型形式参数传递一起传递而来。
如果一个带有不透明返回类型的函数从多处返回，所有可能的返回值必须具有相同的类型。对于范型函数，返回类型可以使用函数的范型类型形式参数，但是它必须是单一的类型。比如说，这里有一个包含方块特殊处理的图形翻转函数的错误版本：

```
func invalidFlip<T: Shape>(_ shape: T) -> some Shape {
    if shape is Square {
        return shape // Error: return types don't match
    }
    return FlippedShape(shape: shape) // Error: return types don't match
}
```

如果你用一个 Square 来调用这个函数，它返回一个 Square ；否则，它就返回一个 FlippedShape 。这违反了返回值必须是一种类型并且让 invalidFlip(_:) 错误。一种修复 invalidFlip(_:) 的方法是把对待方块的特殊情况移动到 FlippedShape 的实现中去，这就使得这个函数总是返回 FlippedShape 值了：

```
struct FlippedShape<T: Shape>: Shape {
    var shape: T
    func draw() -> String {
        if shape is Square {
            return shape.draw()
        }
        let lines = shape.draw().split(separator: "\n")
        return lines.reversed().joined(separator: "\n")
    }
}
```

总是返回一个类型的约束并不能阻止你在返回不透明类型时使用范型。这里有一个合并它类型形式参数到具体返回类型的函数例子：

```
1  func `repeat`<T: Shape>(shape: T, count: Int) -> some Collection {
2      return Array<T>(repeating: shape, count: count)
3  }
```

在这个情况下，返回值的具体类型依赖 T ：无论什么图形传入， repeat(shape:count:) 都会创建和返回这个图形的数组。就算如此，返回的值也总是同一种类型 [T] ，所以它依旧满足返回不透明类型的函数必须返回同一类型的约束。

不透明类型和协议类型的区别

返回不透明类型看起来与使用协议类型作为函数返回类型非常相似，但这两种返回类型区别于它们是否保存类型特征。不透明类型引用为特定的类型，尽管函数的调用者不能看到是那个类型；协议类型可以引用到任何遵循这个协议的类型。通常来讲，协议类型能提供更多存储值的弹性，不透明类型则能给你更多关于具体类型的保证。

比如，这里有一个版本的 flip(_:) 它返回一个协议类型的值而不是不透明类型：

```
1  func protoFlip<T: Shape>(_ shape: T) -> Shape {
2      return FlippedShape(shape: shape)
3  }
```

这个版本的 protoFlip(_:) 代码和 flip(_:) 一样，并且它也总是返回相同类型的值。和 flip(_:) 不同的是， protoFlip(_:) 返回的值并不要求总是返回相同的类型——只要遵循 Shape 协议就好了。换句话来说， protoFlip(_:) 使得 API 要求远比 flip(_:) 要松。它保留了返回多种类型的弹性：

```
1  func protoFlip<T: Shape>(_ shape: T) -> Shape {
2      if shape is Square {
3          return shape
4      }
5
6      return FlippedShape(shape: shape)
7  }
```

修改过的代码返回一个 Square 的实例或者是 FlippedShape 的实例，基于传入的图形决定。这个函数返回的两个翻转过的图形可能拥有完全不同的类型。其他此函数的合法版本会在翻转多个相同图形的实例时返回不同类型的值。 protoFlip(_:) 具有更少的特定返回类型信息，这就意味着很多依赖类型信息的操作无法完成。比如， == 运算符就无法比较这个函数返回的结果。

```
1  let protoFlippedTriangle = protoFlip(smallTriangle)
2  let sameThing = protoFlip(smallTriangle)
3  protoFlippedTriangle == sameThing  // Error
```

最后一行的错误有很多引发原因。最首先是 Shape 并没有 == 作为自身协议的需求。如果你尝试添加，那么接下来就会遇到 == 运算符需要知道左手实际参数和右手实际参数的类型。这一系列运算符通常取实际参数的类型为 Self 类型，匹配任何遵循协议的具体类型，但添加 Self 需求给协议并不能让类型保证你在使用协议作为类型时能匹配成功。

使用协议类型作为函数的返回类型能给你带来不少弹性以返回任意遵循协议的类型。总之，这样弹性的代价就是返回值无法使用某些运算。例子展示了 == 符为何不可用——它需要基于特定的类型信息但协议类型无法提供。

这么做的另一个问题是图形转换不能嵌套。翻转三角形的结果是一个 Shape 类型的值，protoFlip(_:) 函数接受一个遵循 Shape 协议的某类型作为实际参数。总之，协议类型的值并不遵循那个协议； protoFlip(_:) 返回的值并不遵循 Shape 。这就意味着类似 protoFlip(protoFlip(smallTriange)) 这样应用多个转换的代码是不合法的，因为翻转了的图形不是 protoFlip(_:) 合法的实际参数。

相反，不透明类型保持了具体类型的特征。Swift 可以推断相关类型，这就使得你能在某些不能把协议类型作为返回类型的地方使用不透明类型。举例来说，这里有一个版本的 Container 协议，来自范型：

```
protocol Container {
    associatedtype Item
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
extension Array: Container { }
```

你不能使用 Container 作为函数的返回类型，因为这个协议有一个关联类型。你也不能使用它作为范型返回类型的约束因为它在函数体外没有足够的信息来推断它到底需要成为什么范型类型。

```
// Error: Protocol with associated types can't be used as a return type.
func makeProtocolContainer<T>(item: T) -> Container {
    return [item]
}

// Error: Not enough information to infer C.
func makeProtocolContainer<T, C: Container>(item: T) -> C {
    return [item]
}
```

使用不透明类型 some Container 作为返回类型则能够表达期望的 API 约束——函数返回一个容器，但不指定特定的容器类型：

```
1  func makeOpaqueContainer<T>(item: T) -> some Container {
2      return [item]
3  }
4  let opaqueContainer = makeOpaqueContainer(item: 12)
5  let twelve = opaqueContainer[0]
6  print(type(of: twelve))
7  // Prints "Int"
```

twelve 类型被推断为 Int ，这展示了类型类型推断能够在不透明类型上正常运行的事实。在 makeOpaqueContainer(item:) 的实现中，不透明容器的具体类型是 [T] 。在这个例子中，T 是 Int ，所以返回值是一个整数的数组并且 Item 的关联类型被推断为 Int 。 Container 的下标返回 Item ，也就是说 twelve 的类型也被推断为 Int 。