Protocol和泛型的台前幕后

从隐式接口和编译期多态说起

(https://www.boxueio.com/series/protocol-and-generic/ebook/189)

我们的网络请求代码是怎么变成乱七八糟的?

(https://www.boxueio.com/series/protocol-and-generic/ebook/191)

具象函数和泛型函数的解析顺序

Back to series (/series/protocol-and-generic)

字号
字号
默认主题
金色主题
暗色主题

就像我们在上一个视频结尾提到的一样，如果代码中同时存在着和泛型函数签名“近乎”一样的具象函数，Swift会如何匹配调用的函数呢?

简单来说就是：找到匹配度最高的一个，并且具象函数的匹配度高于泛型函数。接下来，我们在三个不同的场景里，具体了解一下这个匹配规则。

全局函数的匹配顺序

假设，我们有一个计算余数的方法家族 mod ，它有两个接受 Double 和 Float 的具象版本，以及一个接受整数的泛型版本：

```
func mod(_ m: Double, by n: Double) -> Double {
    print("Double ver.")
    return m.truncatingRemainder(dividingBy: n)
}

func mod(_ m: Float, by n: Float) -> Float {
    print("Float ver.")
    return m.truncatingRemainder(dividingBy: n)
}

func mod<T: Integer>(_ m: T, by n: T) -> T {
    print("Generic ver.")
    return m % n
}
```

然后分别用下面的方法来测试：

```
mod(8, by: 2.5) // Double ver.
let eight: Float = 8
mod(eight, by: 2.5) // Float ver.
mod(8, by: 2)   // Generic ver.
```

从注释中的结果可以看到，编译器会根据参数的类型自动为我们选择要调用的函数。但就像我们在上一节中提到的一样，这个过程是在编译期完成的。我们来看下面这个例子：

```
let doubles = [2, 3, 4.5]
doubles.forEach { _ = mod($0, by: 2) }
// Double ver.
// Double ver.
// Double ver.
```

这次，尽管我们的 double 数组中同时包含了整数和浮点数，但编译器并不会生成在运行时根据读取的数值类型不同，调用不同 mod 方法的代码。它只会为我们统一**静态绑定**到type inference推断出来的 Double 版本上。

操作符的匹配顺序

在Swift里，还有一类特殊的函数。就是我们自定义的操作符，和刚才我们的例子类似。如果存在一个操作符的多个重载版本，Swift总是会优先使用非泛型的版本。但不同的是，如果调用在选择上存在歧义，编译器则不会调用泛型版本，而是会要求你明确作出选择。

我们来看下面的例子，把之前计算模数的函数改写成操作符：

```
func %%(lhs: Double, rhs: Double) -> Double {
    print("Double opt.")
    return lhs.truncatingRemainder(dividingBy: rhs)
}

func %%(lhs: Float, rhs: Float) -> Float {
    print("Float opt.")
    return lhs.truncatingRemainder(dividingBy: rhs)
}

func %%<T: Integer>(lhs: T, rhs: T) -> T {
    print("Generic opt.")
    return lhs % rhs
}
```

然后，下面的调用还可以正常工作：

```
8 %% 2.5 // Double opt.
Float(8) %% 2.5 // Float opt.
```

但是，下面的代码却会发生编译错误：

```
// !! Compile time error !!
// Ambiguous use of operator %%
8 %% 2
```

因为对于上面这个表达式，我们定义的三个 %% 的版本语法上都是合法的。为了消除歧义，我们只能通过type annotation帮助编译器明确选择：

```
let number: Int = 8 %% 2 // Generic opt.
```

类型约束也可以作为重载函数的匹配条件

除了具体的类型之外，我们还可以使用Swift中特有的类型约束作为选择重载函数的依据。例如，我们有一个在集合中查找元素的函数：

```
func find<T: Collection>(
    value: T.Iterator.Element,
    in coll: T) -> Bool
    where T.Iterator.Element: Equatable {
        print("Equatable find")

        var pos = coll.makeIterator()
        var tmp = pos.next()

        while let n = tmp {
            if (n == value) {
                return true
            }

            tmp = pos.next()
        }

        return false
}
```

在 `find` 的实现里，我们对 `Collection` 的唯一要求，就是它的元素支持比较操作，在这样的条件下，为了在 `coll` 中找到 `value` ，我们只能顺序从头遍历到尾，此时的 `find` 是一个O(n)的算法。

我们可以用这样的代码试一下：

```
find(value: 10, in: [1, 2, 3, 4, 5])
// false
// Equatable find
```

但如果我们对集合中元素的类型做进一步的约束，例如让它可以是 `Hashable` 的，就可以用通过一个 `Set` 来查找 `value` ，这时，查找过程就变成了一个O(1)的算法：

```
func find<T: Collection>(
    value: T.Iterator.Element,
    in coll: T) -> Bool
    where T.Iterator.Element: Hashable {
        print("Hashable find")
        return Set(coll).contains(value)
}
```

然后，我们之前的测试代码就会在控制台打印"Hashable find"了。

What's next?

通过前两个小节我们看到了，泛型编程可以在编译期为不同的类型生成相同功能的代码。在Swift里，我们可以使用类型替代符定义泛型函数，也可以通过 `associatedtype` 定义泛型类型。下一节，我们来看一个更具体的应用场景，如何利用泛型编程，把一个网络请求保存到Swift model。

◀ 从隐式接口和编译期多态说起
(https://www.boxueio.com/series/protocol-and-generic/ebook/189)

我们的网络请求代码是怎么变成乱七八糟的? ▶
(https://www.boxueio.com/series/protocol-and-generic/ebook/191)