Swift 类型擦除

swift.gg/2018/10/11/friday-qa-2017-12-08-type-erasure-in-swift

> 作者：Mike Ash，原文链接，原文日期：2017-12-18

你也许曾听过**类型擦除**，甚至也使用过标准库提供的类型擦除类型如 `AnySequence`。但到底什么是类型擦除? 如何自定义类型擦除? 在这篇文章中，我将讨论如何使用类型擦除以及如何自定义。在此感谢 Lorenzo Boaro 提出这个主题。

前言

有时你想对外部调用者隐藏某个类的具体类型，或是一些实现细节。在一些情况下，这样做能防止静态类型在项目中滥用，或者保证了类型间的交互。类型擦除就是移除某个类的具体类型使其变得更通用的过程。

协议或抽象父类可作为类型擦除简单的实现方式之一。例如 `NSString` 就是一个例子，每次创建一个 `NSString` 实例时，这个对象并不是一个普通的 `NSString` 对象，它通常是某个具体的子类的实例，这个子类一般是私有的，同时这些细节通常是被隐藏起来的。你可以使用子类提供的功能而不用知道它具体的类型，你也没必要将你的代码与它们的具体类型联系起来。

在处理 Swift 泛型以及关联类型协议的时候，可能需要使用一些高级的内容。Swift 不允许把协议当做具体的类型来使用。例如，如果你想编写一个方法，它的参数是一个包含了 `Int` 的序列，那么下面这种做法是不正确的：

```
func f(seq:
Sequence<Int>) { ...
```

你不能这样使用协议类型，这样会在编译时报错。但你可以使用泛型来替代协议，解决这个问题：

```
func f<S: Sequence>(seq: S) where S.Element ==
Int { ...
```

有时候这样写完全可以，但有些地方还存在一些比较麻烦的情况，通常你不可能只在一个地方添加泛型: 一个泛型函数对其他泛型要求更多... 更糟糕的是，你不能将泛型作为返回值或者属性。这就跟我们想的有点不一样了。

```
func g<S: Sequence>() -> S where S.Element ==
Int { ...
```

我们希望函数 `g` 能返回任何符合的类型，但上面这个不同，它允许调用者选择他所需要的类型，然后函数 `g` 来提供一个合适的值。

Swift 标准库中提供了 `AnySequence` 来帮助我们解决这个问题。`AnySequence` 包装了一个任意类型的序列，并擦除了它的类型。使用 `AnySequence` 来访问这个序列，我们来重写一下函数 `f` 与 函数 `g` ：

```
func f(seq:
AnySequence<Int>) { ...

func g() ->
AnySequence<Int> { ...
```

泛型部分不见了，同时具体的类型也被隐藏起来了。由于使用了 `AnySequence` 包装具体的值，它带来了一定的代码复杂性以及运行时间成本。但是代码却更简洁了。

Swift 标准库中提供了很多这样的类型，如 `AnyCollection` 、 `AnyHashable` 及 `AnyIndex` 。这些类型在你自定义泛型或协议的时候非常的管用，你也可以直接使用这些类型来简化你的代码。接下来让我们探索实现类型擦除的多种方式吧。

基于类的类型擦除

有时我们需要在不暴露类型信息的情况下从多个类型中包装一些公共的功能，这听起来就像是父类-子类的关系。事实上我们的确可以使用抽象父类来实现类型擦除。父类提供 API 接口，不用去管谁来实现。而子类根据具体的类型信息实现相应的功能。

接下来我们将使用这种方式来自定义 `AnySequence` ，我们将其命名为 `MAnySequence` ：

```
class MAnySequence<Element>:
Sequence {
```

这个类需要一个 `iterator` 类型作为 `makeIterator` 返回类型。我们必须要做两次类型擦除来隐藏底层的序列类型以及迭代器的类型。我们在 `MAnySequence` 内部定义了一个 `Iterator` 类，该类遵循着 `IteratorProtocol` 协议，并在 `next()` 方法中使用 `fatalError` 抛出异常。Swift 本身不支持抽象类型，但这样也够了：

```
class Iterator: IteratorProtocol {
    func next() -> Element? {
        fatalError("Must override
next()")
    }
}
```

`MAnySequence` 对 `makeIterator` 方法实现也差不多。直接调用将抛出异常，这用来提示子类需要重写这个方法:

```
    func makeIterator() -> Iterator {
        fatalError("Must override
makeIterator()")
    }
}
```

这样就定义了一个基于类的类型擦除的API，私有的子类将来实现这些API。公共类通过元素类型参数化，但私有实现类由它包装的序列类型进行参数化:

```
private class MAnySequenceImpl<Seq: Sequence>:
MAnySequence<Seq.Element> {
```

`MAnySequenceImpl` 需要一个继承于 `Iterator` 的子类:

```
class IteratorImpl:
Iterator {
```

`IteratorImpl` 包装了序列的迭代器:

```
var wrapped:
Seq.Iterator

init(_ wrapped:
Seq.Iterator) {
    self.wrapped =
wrapped
}
```

在 `next` 方法中调用被包装的序列迭代器：

```
    override func next() ->
Seq.Element? {
        return wrapped.next()
    }
}
```

相似地，`MAnySequenceImpl` 包装一个序列：

```
var seq:
Seq

init(_ seq:
Seq) {
    self.seq
= seq
}
```

从序列中获取迭代器，然后将迭代器包装成 `IteratorImpl` 对象返回，这样就实现了 `makeIterator` 的功能。

```
    override func makeIterator() ->
IteratorImpl {
        return
IteratorImpl(seq.makeIterator())
    }

}
```

我们需要一种方法来实际创建这些东西：对 `MAnySequence` 添加一个静态方法，该方法创建一个 `MAnySequenceImpl` 实例，并将其作为 `MAnySequence` 类型返回给调用者。

```
extension MAnySequence {
    static func make<Seq: Sequence>(_ seq: Seq) -> MAnySequence<Element> where Seq.Element ==
Element {
        return MAnySequenceImpl<Seq>(seq)
    }
}
```

在实际开发中，我们可能会做一些额外的操作来让 `MAnySequence` 提供一个初始化方法。

我们来试试 `MAnySequence` ：

```
func printInts(_ seq:
MAnySequence<Int>) {
    for elt in seq {

    }
}

let array = [1, 2, 3, 4, 5]
printInts(MAnySequence.make(array))
printInts(MAnySequence.make(array[1 ..
< 4]))
```

完美!

基于函数的类型擦除

有时我们希望对外暴露支持多种类型的方法，但又不想指定具体的类型。一个简单的办法就是，存储那些签名仅涉及到我们想公开的类型的函数，函数主体在底层已知具体实现类型的上下文中创建。

我们一起看看如何运用这种方法来设计 `MAnySequence`，与前面的实现很类似。它是一个结构体而非类，这是因为它仅仅作为容器使用，不需要有任何的继承关系。

```
struct MAnySequence<Element>:
Sequence {
```

跟之前一样，`MAnySequence` 也需要一个可返回的迭代器（Iterator）。迭代器同样被设计为结构体，并持有一个参数为空并返回 `Element?` 的存储型属性，实际上这个属性是一个函数，被用于 `IteratorProtocol` 协议的 `next` 方法中。接下来 `Iterator` 遵循 `IteratorProtocol` 协议，并在 `next` 方法中调用函数:

```
struct Iterator:
IteratorProtocol {
    let _next: () -> Element?

    func next() -> Element?
{
        return _next()
    }
}
```

`MAnySequence` 与 `Iterator` 很相似：持有一个参数为空返回 `Iterator` 类型的存储型属性。遵循 `Sequence` 协议并在 `makeIterator` 方法中调用这个属性。

```
let _makeIterator: () ->
Iterator

func makeIterator() ->
Iterator {
    return _makeIterator()
}
```

`MAnySequence` 的构造函数正是魔法起作用的地方，它接收任意序列作为参数:

```
init<Seq: Sequence>(_ seq: Seq) where Seq.Element ==
Element {
```

接下来需要在构造函数中包装此序列的功能:

```
_makeIterat
or = {
```

如何生成迭代器？请求 `Seq` 序列生成：

```
var iterator =
seq.makeIterator()
```

接下来我们利用自定义的迭代结构体包装序列生成的迭代器，包装后的 `_next` 属性将会在迭代器协议的 `next()` 方法中被调用:

```
            return Iterator(_next: {
iterator.next() })
        }
    }
}
```

接下来展示如何使用 `MAnySequence` ：

```
func printInts(_ seq:
MAnySequence<Int>) {
    for elt in seq {

    }
}

let array = [1, 2, 3, 4, 5]
printInts(MAnySequence(array))
printInts(MAnySequence(array[1 ..<
4]))
```

正确运行，太棒了！

当需要将小部分功能包装为更大类型的一部分时，这种基于函数的类型擦除方法特别实用，这样做就不需要有单独的类来实现被擦除类型的这部分功能了。

比方说你现在想要编写一些适用于各种集合类型的代码，但它真正需要能够对这些集合执行的操作是获取计数并执行从零开始的整数下标。如访问 `tableView` 数据源。它可能看起来像这样：

```
class GenericDataSource<Element> {
    let count: () -> Int
    let getElement: (Int) -> Element

    init<C: Collection>(_ c: C) where C.Element == Element，C.Index
== Int {
        count = { c.count }
        getElement = { c[$0 - c.startIndex] }
    }
}
```

`GenericDataSource` 其他代码可通过调用 `count()` 或 `getElement()` 来操作传入的集合。且不会让集合类型破坏 `GenericDataSource` 泛型参数。

结束语

类型擦除是一种非常有用的技术，它可用来阻止泛型对代码的侵入，也可用来保证接口简单明了。通过将底层类型包装起来，将API与具体的功能进行拆分。这可以通过使用抽象的公共超类和私有子类或将 API 包装在函数中来实现。对于只需要一些功能的简单情况，基于函数类型擦除极其有效。

Swift 标准库提供了几种可直接利用的类型擦除类型。如 `AnySequence` 包装一个 `Sequence`，正如其名，`AnySequence` 允许你对序列迭代而无需知道序列具体的类型。`AnyIterator` 也是类型擦除的类型，它提供一个类型擦除的迭代器。`AnyHashable` 也同样是类型擦除的类型，它提供了对Hashable类型访问功能。Swift 还有很多基于集合的擦除类型，你可以通过搜索 `Any` 来查

阅。标准库中也为 `Codable` API 设计了类型擦除类型: `KeyedEncodingContainer` 和 `KeyedDecodingContainer`。它们都是容器协议类型包装器，可用来在不知道底层具体类型信息的情况下实现 `Encode` 和 `Decode`。

这就是今天全部的内容了，下次再见。你们的建议对 Friday Q&A 是最好的鼓励，所以如果你关于这个主题有什么好的想法，请 发邮件到这里。

> 本文由 SwiftGG 翻译组翻译，已经获得作者翻译授权，最新文章请访问 http://swift.gg。