Interoperate Swift with C

返回视频 (/series/interoperate-swift-with-c)

C中的简单函数是如何桥接到Swift的 (https://www.boxueio.com/series/interoperate-swift-with-c/ebook/246)

C中的基本类型在Swift中是如何表示的

Back to series (/series/interoperate-swift-with-c)

字号
字号
默认主题
金色主题
暗色主题

自从Swift开源并被移植到更多平台之后，一个日益显现的问题就是它需要更多地和C进行混编，调用OS API也好，使用第三方程序库也好。因此，接下来的一个话题就是，从各种基础类型、`struct`、函数、指针到OC对C的各种扩展，这些语言元素是如何桥接到Swift的呢？这个系列里，我们就通过一些实际的场景来了解Swift和C交互的方式。

在这个系列的绝大部分视频里，我们都会在*traditional_oc.m/.h*中编写传统的C代码，并在*main.swift*中访问这些C代码。

C语言中的基础类型

首先要介绍的，是C中的基础类型，大家可以在这里(https://developer.apple.com/library/content/documentation/Swift/Conceptual/BuildingCocoaApps/InteractingWithCAPIs.html#//apple_ref/doc/uid/TP40014216-CH8-ID17)找到完整的基本类型对应表，简单来说，就是**对C中的类型采取驼峰式命名之后，加上前缀字母C**。例如：

- `int` 变成 `CInt`；
- `unsigned char` 变成 `CUnsignedChar`；
- `unsigned long long` 变成 `CUnsignedLongLong`；

其中，只有有三个表示宽字符的类型是特殊的：

- `wchat_t` 变成 `CWideChar`；
- `char16_t` 变成 `CChar16`；
- `char32_t` 变成 `CChar32`；

于是，在Swift里，我们可以直接使用这些类型来定义变量，例如：

```
let cInt: CInt = 10
let cChar: CChar = 49
```

在Xcode里按住option点击这些类型就会看到，它们都是 `typealias` 。例如， `CInt` 的定义是这样的：

```
typealias CInt = Int32
```

但是，即便我们知道了这些C类型对应的Swift类型，当和C代码交互的时候，**我们也应该总是使用这些类型的 `typealias` 版本**，而不要直接使用这些别名对应的原生类型。

导入基本类型的全局变量

例如，我们在*traditionial_oc.h*中声明一个常量全局变量：

```
const int global_ten;
```

并在*traditional_oc.m*中定义它：

```
const int global_ten = 10;
```

这样， `global_ten` 就会作为一个Swift全局常量被引入，我们可以直接读取它的值，但不可以改写：

```
let ten = global_ten
global_ten = 10.0 // Compile time error: `global_ten` is a constant.
```

导入NS_STRING_ENUM修饰的类型

在OC代码里，我们经常会定义一组有共同前缀的常量来模拟 enum 的特性。例如，先在*traditional_oc.h*中，添加下面的代码：

```
typedef NSString * TrafficLightColor NS_STRING_ENUM;

TrafficLightColor const TrafficLightColorRed;
TrafficLightColor const TrafficLightColorYellow;
TrafficLightColor const TrafficLightColorGreen;
```

然后，在*traditional_oc.m*中，初始化这些全局变量：

```
TrafficLightColor const TrafficLightColorRed = @"Red";
TrafficLightColor const TrafficLightColorYellow = @"Yellow";
TrafficLightColor const TrafficLightColorGreen = @"Green";
```

于是，对于用 NS_STRING_ENUM 修饰的 TrafficLightColor ，引入到Swift就会变成一个类似这样的 struct：

```
struct TrafficLightColor: RawRepresentable {
    typealias RawValue = String

    init(rawValue: RawValue)
    var rawValue: RawValue { get }

    static var red: TrafficLightColor { get }
    static var yellow: TrafficLightColor { get }
    static var green: TrafficLightColor { get }
}
```

这个转换规则是这样的：

- 根据 NS_STRING_ENUM 修饰的类型决定导入到Swift时 struct 的名字，因此，导入的类型名称就是 TrafficLightColor；
- 去掉和类型名称相同的公共前缀，并把剩余部分首字母小写后，变成 struct 的type property；

于是，在Swift里，我们就可以这样来使用这些OC常量了：

```
let redColor: TrafficLightColor = .red
let redColorRawValue = redColor.rawValue // Red
```

就像我们刚才说过的， NS_STRING_ENUM 修饰的类型，通常表示某个范围里，值固定的类型。例如我们不会再期望给它添加个蓝灯这样的属性。但并不是所有的类型都如此，如果一个类型的值有可能扩展，我们可以使用 NS_EXTENSIBLE_STRING_ENUM 来修饰它。

NS_EXTENSIBLE_STRING_ENUM

例如，在*traditional_oc.h*中，添加下面的声明：

```
typedef int Shape NS_EXTENSIBLE_STRING_ENUM;

Shape const ShapeCircle;
Shape const ShapeTriangle;
Shape const ShapeSquare;
```

并在*traditional_oc.m*中定义它们：

```
Shape const ShapeCircle = 1;
Shape const ShapeTriangle = 2;
Shape const ShapeSquare = 3;
```

这样，按照之前的逻辑，类型 Shape 在Swift中会被导入成一个 struct ，它和 TrafficLight 唯一不同的地方在于，多了一个可以省略参数的 init 方法，使得我们可以在Swift里，这样扩展 Shape 的值：

```
extension Shape {
    static var ellipse: Shape {
        return Shape(4)
    }
}
```

然后，我们就可以定义椭圆了： let e: Shape = .ellipse 。

当然，这并不是说使用 NS_STRING_ENUM 导入的类型就不可以扩展，例如，我们也可以在Swift里，这样扩展 TrafficLightColor：

```
extension TrafficLightColor {
    static var blue: TrafficLightColor {
        return TrafficLightColor(rawValue: "Blue")
    }
}
```

从语法上来说，这没有任何问题。因此， `NS_STRING_ENUM` 和 `NS_EXTENSIBLE_STRING_ENUM` 并不是什么语言层面上的限制，而只是语义上的差别。面对这种差别，Swift为 `NS_EXTENSIBLE_STRING_ENUM` 提供了更为方便的扩展方法罢了。

What's next?

以上，就是这一节的内容。我们首先了解了C中基本类型映射到Swift的方式；并了解了这些基本类型的全局变量桥接到Swift之后的访问方法。下一节，我们来看C中的简单函数是如何桥接到Swift的。

⏮ 返回视频

(/series/interoperate-swift-with-c)

C中的简单函数是如何桥接到Swift的 ⏭

(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/246)