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enum是如何桥接到Swift的

(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/248)

C中的struct和union是如何桥接到Swift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

我们先来了解C中的简单 `struct` 以及处理 `struct` 类型的函数是如何桥接到Swift的。所谓简单 `struct` 是指不包含任何指针成员的 `struct` ，任何与指针有关的话题，稍后我们会在专门的内容中分享。

Structures

为了观察桥接的效果，我们先在*traditional_oc.h*中，声明一个 `struct Location`：

```
struct Location {
    double x;
    double y;
};

typedef struct Location Location;
```

这样，Swift就会导入一个同名的 `struct` ，其中：

- 包含C `struct` 中的每一个同名数据成员；
- 包含两个 `init` 方法，一个是默认 `init` ，用于把Swift中的每个属性默认初始化。另一个，是memberwise init，它使用属性名称作为 `init` 的internal/external name，用于按成员初始化；

因此，Swift导入的 `struct` 看上去是这样的：

```
struct Location {
    var x: Double
    var y: Double

    init()
    init(x: Double, y: Double)
}
```

我们可以用下面两种方式来定义 `Location` 对象，它们分别使用了默认初始化以及memberwise初始化：

```
var origin = Location()
var eleven = Location(x: 1, y: 1)
```

除了这些最基本的定义之外，通常一个C `struct` 还会带有一系列用于操作这个类型的全局函数。我们可以使用 `CF_SWIFT_NAME` 宏，把这些函数导入成对应Swift类型的 `extension` 。例如，在*traditional_oc.h*中，声明下面这两个方法：

```
Location moveX(Location loc, double delta);
Location moveY(Location loc, double delta);
```

然后，在*objective_oc.m*中定义它们：

```
Location moveX(Location loc, double delta) {
    loc.x += delta;

    return loc;
}

Location moveY(Location loc, double delta) {
    loc.y += delta;

    return loc;
}
```

此时， `moveX` 和 `moveY` 就会作为全局函数桥接到Swift，我们只能这样调用它们：

```
var pos = Location()
pos = moveX(pos, 11)
```

使用CF_SWIFT_NAME导入普通全局函数

但显然，明确给 moveX 传递 pos 并不是Swift的风格， moveX 应该是 Location 的一个方法，为此，我们可以把 moveX 和 moveY 的声明修改成这样：

```
Location moveX(Location loc, double delta) CF_SWIFT_NAME(Location.moveX(se
lf:delta:));
Location moveY(Location loc, double delta) CF_SWIFT_NAME(Location.moveY(se
lf:delta:));
```

其中， CF_SWIFT_NAME 中方法的写法，和Swift中 #selector 的参数是一样的。这样，Swift就会把上面两个方法导入到 extension Location 里：

```
extension Location {
    func moveX(delta: Double) -> Location {
        return Location(x: self.x + delta, y: self.y)
    }

    func moveY(delta: Double) -> Location {
        return Location(x: self.x, y: self.y + delta)
    }
}
```

然后，我们就可以像这样调用 moveX 和 moveY 了：

```
var pos = Location()
pos = pos.moveX(delta: 10)
```

使用CF_SWIFT_NAME导入init方法

在C中，除了修改 struct 的函数之外，还有直接创建 struct 的工厂方法，例如，我们添加一个在**y=x**直线上创建 Location 的方法：

```
// In traditional_oc.h
Location createWithXY(double xy);

// In traditional_oc.m
Location createWithXY(double xy) {
    Location p = { .x = xy, .y = xy };
    return p;
}
```

这种方法，桥接到Swift后，也不应该是全局函数，而应该是 Location 的某种 init 方法。为此，我们同样可以使用CF_SWIFT_NAME来修饰它：

```
Location createWithXY(double xy) CF_SWIFT_NAME(Location.init(xy:));
```

然后，就可以这样创建 Location 对象了：

```
var eleven = Location(xy: 11)
```

使用CF_SWIFT_NAME导入type property

除了使用 CF_SWIFT_NAME 导入方法之外，我们还可以给 Location 导入type property。例如，假设在C中有一个生成原点位置的方法：

```
// In traditional_oc.h
Location getOrigin(void);

// In traditional_oc.m
Location origin = { .x = 0, .y = 0 };

Location getOrigin(void) {
    return origin;
}
```

除了用刚才的方法把 getOrigin 导入成一个方法，我们还可以这样：

```
Location getOrigin(void) CF_SWIFT_NAME(getter:Location.origin());
```

此时， Location 在Swift中的定义就会多出一个type property：

```
extension Location {
    static var origin: Location { get }
}
```

但我们只能读取它的值，而不能改写它：

```
let origin = Location.origin // OK (0.0, 0.0)
Locatioin.origin.x = 11 // Compile time error
```

为了可以修改 origin ，我们还要再定义一个setter：

```
// In traditional_oc.h
Location setOrigin(Location newOrigin) CF_SWIFT_NAME(setter:Location.origi
n(newOrigin:));

// In traditional_oc.m
Location setOrigin(Location newOrigin) {
    origin = newOrigin;

    return origin;
}
```

这样，无论是赋值新对象，还是修改 origin 的属性，都可以正常工作了。

```
Location.origin = Location() // OK
Location.origin.x = 11; // OK
```

Unions是如何桥接到Swift的?

了解了Swift导入C struct 的方式之后，我们来看一种特殊的C结构： union 。由于Swift不支持原生的 union ，因此，C union 导入到Swift后，会变成一个 struct 。来看下面这个例子：

```
// In traditional_oc.h
union ASCII {
    char character; // ASCII character
    int code;       // ASCII code
};

typedef union ASCII ASCII;
```

为了可以在Swift里表达 ASCII 可以被 char 或 int 类型的值初始化， ASCII 导入到Swift后会带有两个 init 方法：

```
struct ASCII {
    init(character: Int8)
    init(code: Int32)
}
```

然后，为了可以让 ASCII 对象可以分别读取到这两个值，Swift还会为 ASCII 生成两个属性：

```
struct ASCII {
    var character: Int8
    var code: Int32
}
```

因此，我们就可以像下面这样使用 ASCII 类型了：

```
let a = ASCII(character: CChar("a")!)
print(a.character)
```

需要注意的是，调用的 ASCII.init 方法，必须和访问的属性对应。例如在上面的例子里，如果我们访问 a.code ，就会发生运行时错误。

匿名的struct和union数据成员是如何桥接到Swift的?

了解了 struct 和 union 桥接到Swift的方法后，我们来看一个特殊的情况，在 struct 的定义里， struct 和 union 类型的数据成员可以是匿名的。例如下面这个例子：

```
// In traditional_oc.h
struct Car {
    union {
        char model;
        int series;
    };

    struct {
        double pricing;
        bool isAvailable;
    } info;
};

typedef struct Car Car;
```

其中， `Car` 包含了两个成员：

- 首先，是匿名的 `union` ，表示车既可以用一个字母表示型号，也可以用数字表示系列；
- 其次，是个具名 `struct` ，包含了汽车的售价以及是否有货；

之前我们说过，C `struct` 导入Swift之后，会创建一个默认的memberwise init方法，但是 `union` 并没有名称，我们该如何初始化呢？来看下面这段Swift代码：

```
let bmw = Car(.init(series: 5),
    info: .init(price: 500000, isAvailable: true))
```

可以看到，Swift不会为匿名的 `union` 生成external name，我们直接调用 `.init` ，type inference会推导出这里需要的类型，我们只要按照之前讲过的方式初始化 `union` 就好了。

初始化完成后，我们可以直接访问 `union` 的数据成员，就像匿名的类型不存在一样：

```
print(bmw.series) // 5
```

What's next?

以上，就是C中的 `struct` 和 `union` 桥接到Swift的方式，尽管我们还没有提及指针类型，但它已经比之前的简单类型复杂一些了。下一节，我们来看 `enum` 是如何桥接到Swift的。

⏮ 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enum是如何桥接到Swift的 ⏭

(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/248)

泊学相关

关于泊学 ›

加入泊学 ›

泊学用户隐私以及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)
QQ: 2085489246

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 靛青K的博客 (http://blog.dianqk.org/)