Interoperate Swift with C

使用Buffer视图改进内存访问

(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/250)

关于C中的字符串指针

(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/252)

C指针是如何桥接到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

在这一节，我们来看C的指针类型和Swift的交互。在C里，指针有两类属性：

- 一类是指针指向的对象是否是常量，例如： `const int *` 和 `int *`，前者不可以更改指针指向的内容，但后者可以；
- 一类是指针自身是否带有类型，例如： `int *` 和 `void *`；

因此，当C中的指针桥接到Swift时，也要针对这些情况进行处理。

带有类型信息的指针

如果指针带有类型信息，在Swift里，我们只要根据指针自身是否是常量进行处理就好了，来看下面这个例子。先在*traditional_oc.h/.m*中添加两个函数：

```
// In traditional_oc.h
void printAddress(const int * p);
int doubler(int *p);

// In traditional_oc.m
void printAddress(const int * p) {
    printf("%016lX\n", (unsigned long)p);
}

int doubler(int *p) {
    *p = (*p) * 2;

    return *p;
}
```

这里， `printAddress` 的参数是 `const int *`，因此，桥接到Swift的时候，就会变成一个 `UnsafePointer<Int>`：

```
func getAddress(_ p: UnsafePointer<CInt>!) -> Void
```

这个签名里，有两点需要注意：

- 由于C里，指针指向的是 `const int`，Swift对应的类型是 `UnsafePointer<CInt>`，我们不能通过这种类型的对象修改指针指向的值；
- 由于C指针可能为 `NULL`，因此指针参数引入到Swift之后，都是implicitly unwrapped optional；

而对于 `doubler` 来说，由于它接受的 `int *`，因此，桥接到Swift的时候就是这样的：

```
func doubler(_ p: UnsafeMutablePointer<CInt>!) -> CInt
```

唯一的区别，就是指针的类型从 `UnsafePointer<CInt>` 变成了 `UnsafeMutablePointer<Int>`。但是，无论是哪种类型的指针，在Swift里调用它们的方式都是一样的：

```
var ten: CInt = 10

getAddress(&ten)
doubler(&ten)
```

和 `inout` 类型的参数一样，我们直接使用 `&` 表示传递变量的地址就好了。但要说明的是， `&ten` 这种用法只有在接受Swift Pointer家族类型的参数时，才会自动进行类型转换，把 `&ten` 变成对应的指针类型。如果你在函数上下文环境外这样写：

```
var pTen = &ten
```

Swift就会产生编译错误，告诉你不能这么干。为了帮你达成目的，Swift提供了两个全局函数：

- 一个是只读内存访问的 `withUnsafePointer`：

```
withUnsafePointer(to: &ten) {
    (ptr: UnsafePointer<CInt>) -> Void in
    print(ptr.pointee)
}
```

它的第一个参数，是一个 inout 参数，表示要“临时”通过指针访问的变量。第二个参数是一个closure，Swift会把第一个参数的地址传递给这个closure，这样，我们就可以在任意位置通过 withUnsafePointer 访问变量的地址了。只是对于这个只读的版本，传递给closure参数的指针类型是 UnsafePointer 。因此，我们可以读取地址和地址的值，但不能修改它。

- 另一个是 withUnsafeMutablePointer ，它的用法和 withUnsafePointer 是相同的，只是closure接受的是一个可读写指针类型，因此，我们可以在这里修改第一个参数的值：

```
withUnsafeMutablePointer(to: &ten) {
    (ptr: UnsafeMutablePointer<CInt>) -> Void in
    ptr.pointee = 20
}
```

这样，执行过之后， ten 的值就变成20了。

> 之所以说第一个参数是“临时”通过指针访问的变量，是因为Swift并不保证传递给closure的参数在closure执行结束仍旧是可用的，因此，**不要把传递给closure参数的指针传递到closure外部使用**。

不带有类型信息的指针

当C函数返回 void * 的时候，除了大小，Swift就指针指向的内存一无所知了，这时Swift就无法对指针指向的内容进行管理，无法对访问进行类型和对齐检查。让Swift如何理解这片内存就成了开发者自己的任务。一个最典型的例子，就是C中的 malloc() ，它在Swift中的签名是这样的：

```
func malloc(_ __size: Int) -> UnsafeMutableRawPointer!
```

在这个返回值的类型里，*Mutable*和*Raw*分别解释了这个类型的含义。但接下来的问题是，既然这个“纯指针”什么信息也没有，我们怎么用呢？其实就像我们无法在C里直接使用 void * 一样，我们要对这种执行依据访问的目的手动进行类型转换。

> 虽然也有 UnsafeRawPointer 这种只读的无类型指针，但实际上应用并不多。

指针类型的转换

所谓指针的类型转换，就是明确告诉Swift编译器如何理解指针指向内存的类型和大小。

Raw pointer的类型转换

要转换raw pointer的类型，就是以特定的内存规格，重新生成一个带有类型的指针变量。根据内存是否已经初始化，Swift提供了两个函数。

- 当内存尚未初始化时，我们可以使用 bindMemory(to:capacity) 把内存直接绑定到一个typed pointer上：

```
let rawPointer = malloc(10 * MemoryLayout<Int>.size)!
let intPtr = rawPointer.bindMemory(
    to: Int.self, capacity: 5 * MemoryLayout<Int>.size)
```

刚才我们说过， malloc 返回的是 UnsafeMutableRawPointer! ，它只负责申请内存，但不会对内存区域进行初始化。因此，为了转换它的类型，我们要调用 bindMemory 函数。

它的第一个参数，是我们要转换的指针指向的类型，第二个参数，是**转换过的指针**覆盖的内存区域大小。关于 bindMemory 的用法，有三点是要注意的：

首先、 capacity 的值和 malloc 申请时的大小可以是不一样的。例如，我们就只让 intPtr 覆盖了 malloc 申请到的一半的内存空间，而剩下的一半，仍旧是untyped的未开发地带；

其次、 capacity 的值，是根据 to 的类型计算的，因此**不要让 capacity 等于无法和 to 类型大小对齐的值**；

第三、**raw pointer只能被明确绑定一次**，多次绑定到不同类型之后的行为是未定义的；

- 如果raw pointer已经绑定过一个具体的类型，例如这个raw pointer是通过某种typed pointer初始化来的，我们可以使用 assumingMemoryBound(to:) 重新把raw pointer再绑定回去：

```
let rawPointer = UnsafeMutableRawPointer(&ten)
let tenPointer = rawPointer.assumingMemoryBound(to: CInt.self)
```

关于 assumingMemoryBound 唯一要说明的是， **to 的类型必须和之前生成raw pointer的类型相同**，否则行为是未定义的。也就是说，我们不能把 rawPointer 绑定到指向 Int.self 类型，只能是 CInt.self 。

Typed pointer的临时类型转换

除了把raw pointer一次性转换成一个typed pointer之外，我们也可以把一个typed pointer临时变成另外一种类型的指针，这就好比同一块内存区域上存在着多种不同的view。来看下面这个例子：

```
let intPtr = UnsafeMutablePointer<Int32>.allocate(capacity: 1)
intPtr.initialize(to: 0x12345678)

intPtr.withMemoryRebound(to: Int8.self, capacity: 1) {
    (ptr: UnsafeMutablePointer<Int8>) -> Void in

    print(ptr[0])
    print(ptr[1])
    print(ptr[2])
    print(ptr[3])
}
```

intPtr 是一个指向 Int32 区域的指针，我们把这块内存初始化了成0x12345678。接下来，如果我们想把这块内存当作4个 Int8 处理，就可以使用 withMemoryRebound 方法。它的第一个参数，表示新指针看待内存的方式，因此我们传递 Int8.self ； 第二个参数，表示要在内存空间中，绑定多少对象到 Int8 ，由于一个 Int32 可以包含4个 Int8 ，因此我们传递4；第三个参数是一个closure，withMemoryRebound 会把转换后的指针传递给这个closure，这样，我们就可以在这个closure里，使用 UnsafeMutablePointer<Int8> 类型的指针了，也就用某种“变向”的形式完成了指针的类型转换。

需要注意的是，传递给 withMemoryRebound 的目标类型，必须和指针原来指向的类型大小相同，或者有兼容的内存布局。并且，转换过的指针指只在closure内是确保有效的，因此，不要尝试把这个指针从closure传递到外部。

类型信息不完整的指针

至此，我们就已经了解了绝大多数C指针桥接到Swift的场景了。唯一还没有提及的，是在回调函数中的C指针。这种场景说起来有点儿复杂，来看个例子，假设，有下面这样的C函数：

```
// In traditional_oc.h
typedef void (*CALLBACK)(void *);
void aFuncWithCallback(void *, CALLBACK);

// In traditional_oc.m
void aFuncWithCallback(void * context,
    CALLBACK callback) {
    sleep(3); // Simulate some heavy work
    callback(context); // Notify the caller
}
```

aFuncWithCallback 的第一个参数，表示某种形式的上下文内存；第二参数，是一个回调函数，表示在完成某些操作后，通过调用回调函数通知调用者。在 aFuncWithCallback 的实现里，我们只是通过 sleep 模拟了完成某些工作这个过程。

现在， aFuncWithCallback 桥接到Swift后，如果我们给它的第一个参数传递一个指向Swift类对象的指针，在从 aFuncWithCallback 到回调函数这个环境切换的过程中，ARC会确保指针指向的类对象一直存在么？

答案是不一定，我们不能假设此时closure中得到的指针仍旧指向一个可用的Swift类对象。为了保证类对象的可用性，我们得明确把它“保护”起来。

首先，在*main.swfit*中，添加下面的 class：

```
class Foo {
    var foo = "Foo"

    init() {
        print("Foo get initialzied.")
    }

    deinit {
        print("Foo gets released.")
    }
}
```

为了可以把 Foo 对象当作上下文安全的传递给 aFuncWithCallback ，我们得这样：

- 首先，创建一个不由ARC托管的 Foo 对象：

```
var fooObj = Foo()
let unmanagedFoo = Unmanaged.passRetained(fooObj)
```

这时， unmanagedFoo 对 Foo 对象的引用就不再受ARC监管了，如果我们自己处理不当，这个引用就会造成内存泄漏。

- 其次，用 unmanagedFoo 创建一个 UnsafeMutableRawPointer ：

```
let unmanagedPtr = unmanagedFoo.toOpaque()
```

- 最后，我们就可以把 unmanagedPtr 安全的传递给 aFuncWithCallback 了：

```
aFuncWithCallback(unmanagedPtr) {
    (ptr: UnsafeMutableRawPointer?) -> Void in
    let fooObj =
        Unmanaged<Foo>.fromOpaque(ptr!).takeUnretainedValue()
    print(fooObj.foo) // Foo
}
```

这次，我们用一个closure作为了 aFuncWithCallback 的回调函数，在它的实现里，有两点值得说明：

首先，是从一个 UnsafeMutableRawPointer 获取到非托管对象引用的方法，我们调用了 Unmanaged<Foo>.fromOpaque ，**用这个方法得到的引用并不会增加对象的引用计数**；

其次，是从非托管引用获取对象值的方法。Swift提供了两个方法： takeUnretainedValue() 和 takeRetainedValue() ，它们唯一的区别，就是前者不会让对象的引用计数减1。

因此，在我们的例子里，closure中的 fooObj 指向的就是之前创建的 fooObj 对象。并且，它不会对这个对象的引用计数有任何影响。接下来，我们就能安全的访问它了。

最后，我们可以把整个代码放在一个 if scope里：

```
if true {
    let fooObj = Foo()

    let unmanagedFoo = Unmanaged.passRetained(fooObj)
    let unmanagedPtr = unmanagedFoo.toOpaque()

    aFuncWithCallback(unmanagedPtr) {
        (ptr: UnsafeMutableRawPointer?) -> Void in
        let fooObj =
            Unmanaged<Foo>.fromOpaque(ptr!).takeUnretainedValue()
        print(fooObj.foo)
    }
}
```

然后，分别使用 takeUnretainedValue 和 takeRetainedValue 执行，就可以在控制台看到 Foo 对象的初始化和释放的差异了。

What's next?

以上，就是C指针和Swift交互的绝大部份细节，之所以不是全部，是因为在C和Swif中，字符串的处理逻辑是不同的。在C里， char * 桥接到Swift会变成 UnsafeMutablePointer<Int8> ，为此，要想在Swift中正常调用和字符串相关的C函数，我们额外做一些工作，这就是我们下一节的话题。

⏮ 使用Buffer视图改进内存访问
(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/250)

关于C中的字符串指针 ⏭
(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/252)

职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表， 清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二） (https://www.boxueio.com/after-the-full-upgrade-to-swift3)
Mar 4, 2017

人生中第一次创业的"10有" (https://www.boxueio.com/founder-chat)
Jan 9, 2016

猎云网采访报道泊学 (http://www.lieyunwang.com/archives/144329)
Dec 31, 2015

What most schools do not teach (https://www.boxueio.com/what-most-schools-do-not-teach)
Dec 21, 2015

一个工作十年PM终创业的故事（一） (https://www.boxueio.com/founder-story)
May 8, 2015

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

2017 © Boxue, All Rights Reserved. 京ICP备15057653号-1 (http://www.miibeian.gov.cn/) 京公网安备 11010802020752号 (http://www.beian.gov.cn/portal/registerSystemInfo?recordcode=11010802020752)

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/)