**Swift 中的指针使用**

Apple 期望在 Swift 中指针能够尽量减少登场几率，因此在 Swift 中指针被映射为了一个泛型类型，并且还比较抽象。这在一定程度上造成了在 Swift 中指针使用的困难，特别是对那些并不熟悉指针，也没有多少指针操作经验的开发者 (包括我自己也是) 来说，在 Swift 中使用指针确实是一个挑战。在这篇文章里，我希望能从最基本的使用开始，总结一下在 Swift 中使用指针的一些常见方式和场景。这篇文章假定你至少知道指针是什么，如果对指针本身的概念不太清楚的话，可以先看看这篇五分钟 C 指针教程 (或者它的中文版本)，应该会很有帮助。

**初步**

在 Swift 中，指针都使用一个特殊的类型来表示，那就是 UnsafePointer<T>。遵循了 Cocoa 的一贯不可变原则，UnsafePointer<T> 也是不可变的。当然对应地，它还有一个可变变体，UnsafeMutablePointer<T>。绝大部分时间里，C 中的指针都会被以这两种类型引入到 Swift 中：C 中 const 修饰的指针对应 UnsafePointer (最常见的应该就是 C 字符串的 const char \* 了)，而其他可变的指针则对应 UnsafeMutablePointer。除此之外，Swift 中存在表示一组连续数据指针的 UnsafeBufferPointer<T>，表示非完整结构的不透明指针 COpaquePointer 等等。另外你可能已经注意到了，能够确定指向内容的指针类型都是泛型的 struct，我们可以通过这个泛型来对指针指向的类型进行约束以提供一定安全性。

对于一个 UnsafePointer<T> 类型，我们可以通过 memory 属性对其进行取值，如果这个指针是可变的 UnsafeMutablePointer<T> 类型，我们还可以通过 memory 对它进行赋值。比如我们想要写一个利用指针直接操作内存的计数器的话，可以这么做：

• func incrementor(ptr: UnsafeMutablePointer<Int>) {

•     ptr.memory += 1

• }

•

• var a = 10

• incrementor(&a)

•

• a  // 11

这里和 C 的指针使用类似，我们通过在变量名前面加上 & 符号就可以将指向这个变量的指针传递到接受指针作为参数的方法中去。在上面的 incrementor 中我们通过直接操作 memory 属性改变了指针指向的内容。

与这种做法类似的是使用 Swift 的 inout 关键字。我们在将变量传入 inout 参数的函数时，同样也使用 & 符号表示地址。不过区别是在函数体内部我们不需要处理指针类型，而是可以对参数直接进行操作。

• func incrementor1(inout num: Int) {

•     num += 1

• }

•

• var b = 10

• incrementor1(&b)

•

• b  // 11

虽然 & 在参数传递时表示的意义和 C 中一样，是某个“变量的地址”，但是在 Swift 中我们没有办法直接通过这个符号获取一个 UnsafePointer 的实例。需要注意这一点和 C 有所不同：

• // 无法编译

• let a = 100

• let b = &a

**指针初始化和内存管理**

在 Swift 中不能直接取到现有对象的地址，我们还是可以创建新的 UnsafeMutablePointer 对象。与 Swift 中其他对象的自动内存管理不同，对于指针的管理，是需要我们手动进行内存的申请和释放的。一个 UnsafeMutablePointer 的内存有三种可能状态：

• 内存没有被分配，这意味着这是一个 null 指针，或者是之前已经释放过

• 内存进行了分配，但是值还没有被初始化

• 内存进行了分配，并且值已经被初始化

其中只有第三种状态下的指针是可以保证正常使用的。UnsafeMutablePointer 的初始化方法 (init) 完成的都是从其他类型转换到 UnsafeMutablePointer 的工作。我们如果想要新建一个指针，需要做的是使用 alloc: 这个类方法。该方法接受一个 num: Int 作为参数，将向系统申请 num 个数的对应泛型类型的内存。下面的代码申请了一个 Int 大小的内存，并返回指向这块内存的指针：

• var intPtr = UnsafeMutablePointer<Int>.alloc(1)

• // "UnsafeMutablePointer(0x7FD3A8E00060)"

接下来应该做的是对这个指针的内容进行初始化，我们可以使用 initialize: 方法来完成初始化：

• intPtr.initialize(10)

• // intPtr.memory 为 10

在完成初始化后，我们就可以通过 memory 来操作指针指向的内存值了。

在使用之后，我们最好尽快释放指针指向的内容和指针本身。与 initialize: 配对使用的 destroy 用来销毁指针指向的对象，而与 alloc: 对应的 dealloc: 用来释放之前申请的内存。它们都应该被配对使用：

• intPtr.destroy()

• intPtr.dealloc(1)

• intPtr = nil

注意其实在这里对于 Int 这样的在 C 中映射为 int 的 “平凡值” 来说，destroy 并不是必要的，因为这些值被分配在常量段上。但是对于像类的对象或者结构体实例来说，如果不保证初始化和摧毁配对的话，是会出现内存泄露的。所以没有特殊考虑的话，不论内存中到底是什么，保证 initialize: 和 destroy 配对会是一个好习惯。

**指向数组的指针**

在 Swift 中将一个数组作为参数传递到 C API 时，Swift 已经帮助我们完成了转换，这在 Apple 的官方博客中有个很好的例子：

• import Accelerate

•

• let a: [Float] = [1, 2, 3, 4]

• let b: [Float] = [0.5, 0.25, 0.125, 0.0625]

• var result: [Float] = [0, 0, 0, 0]

•

• vDSP\_vadd(a, 1, b, 1, &result, 1, 4)

•

• // result now contains [1.5, 2.25, 3.125, 4.0625]

对于一般的接受 const 数组的 C API，其要求的类型为 UnsafePointer，而非 const 的数组则对应 UnsafeMutablePointer。使用时，对于 const 的参数，我们直接将 Swift 数组传入 (上例中的 a 和 b)；而对于可变的数组，在前面加上 & 后传入即可 (上例中的 result)。

对于传参，Swift 进行了简化，使用起来非常方便。但是如果我们想要使用指针来像之前用 memory 的方式直接操作数组的话，就需要借助一个特殊的类型：UnsafeMutableBufferPointer。Buffer Pointer 是一段连续的内存的指针，通常用来表达像是数组或者字典这样的集合类型。

• var array = [1, 2, 3, 4, 5]

• var arrayPtr = UnsafeMutableBufferPointer<Int>(start: &array, count: array.count)

• // baseAddress 是第一个元素的指针

• var basePtr = arrayPtr.baseAddress as UnsafeMutablePointer<Int>

•

• basePtr.memory // 1

• basePtr.memory = 10

• basePtr.memory // 10

•

• //下一个元素

• var nextPtr = basePtr.successor()

• nextPtr.memory // 2

**指针操作和转换**

**withUnsafePointer**

上面我们说过，在 Swift 中不能像 C 里那样使用 & 符号直接获取地址来进行操作。如果我们想对某个变量进行指针操作，我们可以借助 withUnsafePointer 这个辅助方法。这个方法接受两个参数，第一个是 inout 的任意类型，第二个是一个闭包。Swift 会将第一个输入转换为指针，然后将这个转换后的 Unsafe 的指针作为参数，去调用闭包。使用起来大概是这个样子：

• var test = 10

• test = withUnsafeMutablePointer(&test, { (ptr: UnsafeMutablePointer<Int>) -> Int in

•     ptr.memory += 1

•     **return** ptr.memory

• })

•

• test // 11

这里其实我们做了和文章一开始的 incrementor 相同的事情，区别在于不需要通过方法的调用来将值转换为指针。这么做的好处对于那些只会执行一次的指针操作来说是显而易见的，可以将“我们就是想对这个指针做点事儿”这个意图表达得更加清晰明确。

**unsafeBitCast**

unsafeBitCast 是非常危险的操作，它会将一个指针指向的内存强制按位转换为目标的类型。因为这种转换是在 Swift 的类型管理之外进行的，因此编译器无法确保得到的类型是否确实正确，你必须明确地知道你在做什么。比如：

• let arr = NSArray(object: "meow")

• let str = unsafeBitCast(CFArrayGetValueAtIndex(arr, 0), CFString.self)

• str // “meow”

因为 NSArray 是可以存放任意 NSObject 对象的，当我们在使用 CFArrayGetValueAtIndex 从中取值的时候，得到的结果将是一个 UnsafePointer<Void>。由于我们很明白其中存放的是 String 对象，因此可以直接将其强制转换为 CFString。

关于 unsafeBitCast 一种更常见的使用场景是不同类型的指针之间进行转换。因为指针本身所占用的的大小是一定的，所以指针的类型进行转换是不会出什么致命问题的。这在与一些 C API 协作时会很常见。比如有很多 C API 要求的输入是 void \*，对应到 Swift 中为 UnsafePointer<Void>。我们可以通过下面这样的方式将任意指针转换为 UnsafePointer。

• var count = 100

• var voidPtr = withUnsafePointer(&count, { (a: UnsafePointer<Int>) -> UnsafePointer<Void> in

•     **return** unsafeBitCast(a, UnsafePointer<Void>.self)

• })

• // voidPtr 是 UnsafePointer<Void>。相当于 C 中的 void \*

•

• // 转换回 UnsafePointer<Int>

• var intPtr = unsafeBitCast(voidPtr, UnsafePointer<Int>.self)

• intPtr.memory //100

**总结**

Swift 从设计上来说就是以安全作为重要原则的，虽然可能有些啰嗦，但是还是要重申在 Swift 中直接使用和操作指针应该作为最后的手段，它们始终是无法确保安全的。从传统的 C 代码和与之无缝配合的 Objective-C 代码迁移到 Swift 并不是一件小工程，我们的代码库肯定会时不时出现一些和 C 协作的地方。我们当然可以选择使用 Swift 重写部分陈旧代码，但是对于像是安全或者性能至关重要的部分，我们可能除了继续使用 C API 以外别无选择。如果我们想要继续使用那些 API 的话，了解一些基本的 Swift 指针操作和使用的知识会很有帮助。

对于新的代码，尽量避免使用 Unsafe 开头的类型，意味着可以避免很多不必要的麻烦。Swift 给开发者带来的最大好处是可以让我们用更加先进的编程思想，进行更快和更专注的开发。只有在尊重这种思想的前提下，我们才能更好地享受这门新语言带来的种种优势。显然，这种思想是不包括到处使用 UnsafePointer 的 :)

【编辑推荐】

• 我在开发第一个Swift App过程中学到的四件事

• Swift主题色顶级解决方案

• Swift 的坑：如何实现 enum 的多选

• 苹果 Swift 语言开源项目精选汇总

• 编程武林谁与争锋？苹果Swift锋芒毕露