Interoperate Swift with C

C中的enum是如何桥接到Swift的
(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/248)

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

认识Swift指针家族类型

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

字号
字号
默认主题
金色主题
暗色主题

C语言里，功能强大也最容易产生各种bug的一个语言特性，就是指针。指一块内存，我们就可以天马行空的去访问它。于是，开辟了内存没有归还的代码，有之；由于内存对齐导致的访问越界问题，有之。总之，那些运行起来时好时坏的bug或多或少都和动态内存分配有着关系。

于是，为了既和C在ABI上兼容，又要尽可能控制安全风险，Swift为指针访问设计了一套家族类型，以求让我们尽量在编写代码的时候就明确表达自己的访问意图。作为开始，我们先了解下这个家族的命名方式。

Pointer类型的命名方式

Swift的指针类型家族由5个关键词构成，它们就像开关，当指针类型中出现特定关键词的时候，表示具备这个特性，反之则不具备：

- **Managed**：表示指针指向的内容由ARC统一管理，指针只负责内容的访问，不负责内存的管理；
- **Unsafe**：表示指针指向的内容需要开发者自行管理，所谓自行管理，是指从申请资源、init、deinit到资源回收这一揽子事情，都需要开发者自己来。因此，**managed**和**unsafe**是互斥的，它们不会同时出现在一个指针类型的名称中；
- **Buffer**：表示一段内存地址的view，它可以让我们用不同的形式看待同一块内存地址，稍后我们会看到它的用法；
- **Raw**：表示指针指向的内存没有类型信息，也就是C中的 `void *` 。不带raw的指针类型都会通过泛型参数的形式表示自己指向的内存中包含的对象类型；
- **Mutable**：表示指针指向的内容可修改，否则指针指向的内存是只读的；

最后，所有的指针类型，都以**Pointer**后缀结尾，以表明它们的身份。

> 这里暂时忽略了OpaquePointer类型，我们会在稍后的视频中提到它。

理解了这个规则之后，我们通过一些代码片段，了解这些类型的用法。

在Swift中分配和回收内存

为了在Swift中开辟10个 `Int` 占据的内存空间，可以这样：

```
var intPtr = UnsafeMutablePointer<Int>.allocate(capacity: 10)
intPtr.initialize(to: 0, count: 10)

// Add memory access operations here.

intPtr.deinitialize()
intPtr.deallocate(capacity: 10)
```

不难理解， `intPtr` 是一个需要我们自行打理的内存区域，我们可以修改这个区域的值，并且这个区域存储的对象类型是 `Int` 。

- 首先，我们调用 `allocate` 方法申请内存，它有一个 `capacity` 参数，表示内存包含的 `Int` 个数。此时， `intPtr` 指向内存的状态是**uninitialized**；
- 其次，调用 `initialize` 方法，把10个 `Int` 都初始化成0，这时 `intPtr` 指向内存的状态就从**uninitialized**变成了**initialized**。这一步对内存的访问至关重要；

> 在Swift 3里，我们可以通过一个Collection对象初始化内存区域，像这样：`intPtr.initialize(from: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0])` 。在Swift 4里，这个方法被废除了。

- 第三，当内存访问结束后，我们先调用 `deinitialize` 方法执行必要的清理工作；

- 最后，调用 deallocate 方法交还内存，这里我们传递的 capacity 要和一开始申请的是一致的；

那么，为什么分配之后调用 initialize 这一步至关重要呢？毕竟，如果我们注释掉它执行程序，也顶多就是分配的内存中包含一些未经初始化的值，但也没什么致命问题。这是因为，我们的内存空间中包含的只是一个简单 Int 类型。如果它包含的是一个引用类型，不调用 initialize 方法是有可能导致程序崩溃的。来看下面这段代码：

```
class Foo {
    var m: Int = 0
}

var fooPtr = UnsafeMutablePointer<Foo>.allocate(capacity: 10)

/// You app may crash at any step below
fooPtr[0] = Foo()
fooPtr[1] = Foo()
fooPtr[2] = Foo()

fooPtr.deinitialize()
fooPtr.deallocate(capacity: 10)
```

对于中间这段 fooPtr[] ，我们可以逐个添加，如果 [0] 不崩溃，就继续写 [1] ，写不了几个，你的App就一定会在运行时崩溃。为什么呢?

这是因为，当我们使用下标操作符访问指针时，Swift就假定指针指向的内存已经是经过初始化的了。因此，它会认为 fooPtr 指向的，就是10个已经被正确初始化的 Foo 对象，尽管事实并不如此。

然后，当我们用 fooPtr[0] 修改其内容时，由于 Foo 对象是受ARC托管的，因此，Swift运行时要先释放老的 Foo 对象，再让 fooPtr[0] 指向新的对象。在释放老 Foo 对象的时候，如果内存中存在的是垃圾数据，释放就会失败，于是就触发运行时错误了。因此，**绝对不要忘记在分配了内存空间后使用 initialize 方法对内容进行初始化**。

理解了内存的分配和回收之后，我们来看如何访问被动态分配的内存。

访问动态分配的内存

为了方便观察内存的读写，我们先给 UnsafeMutablePointer 添加一个 dump 方法，它接受一个参数，表示内存区域里对象的个数。然后，我们把内存里的值，逐个对象打印出来。

```
extension UnsafeMutablePointer
    where Pointee: CustomStringConvertible {
    func dump(count: Int) -> Void {
        var info: String = ""

        for i in 0..<count {
            info += self[i].description + " "
        }

        print(info)
    }
}
```

申请到内存之后，我们可以有几种不同的方式访问，它们和C中指针的用法几乎是一样的。拷贝指针，拷贝的是指针指向的地址，而不是地址指向的内容，因此下面的代码， head 和 intPtr 会指向同一个位置：

```
var head = intPtr
```

接下来，为了访问这块包含了10个 Int 的连续空间，我们可以用下面三种不同的方法：

首先，用指针的下标操作符，这和C中访问数组的方法是一样的：

```
head.dump(count: 10)

for i in 1...10 {
    intPtr[i - 1] = i
}

head.dump(count: 10)
```

当我们对指针使用下标操作符时，会自动返回对应下标位置的值，由于 intPtr 是个mutable pointer，因此，我们可以用它直接读写内存。在循环前后，分别调用 head.dump ，就可以看到下面的结果了：

```
0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10
```

其次，用指针的算术运算，这也是从C直接学来的用法，加1或减1，指针就自动跳到下一个或前一个对象的起始地址。因此，刚才的 for 循环还可以写成这样：

```
for i in 1...10 {
    (intPtr + (i - 1)).pointee = i
}
```

其中， intPtr + (i - 1) 的含义，和 intPtr[i - 1] 是一样的，但对指针进行算术运算后，得到的结果仍旧是一个内存地址，为了读到这个地址的值，我们得访问它的 pointee 属性，这是和使用下标操作符唯一的区别。当然，这个循环的结果，和之前，是完全一样的。

第三，调用特定的指针移动方法。实际上，刚才我们介绍的指针算术运算仍旧是beta方法，在有这些运算符之前，指针的移动是通过一组方法完成的：

- intPtr.predecessor()，移动到 intPtr 的上一个位置；
- intPtr.successor()，移动到 intPtr 的下一个位置；
- intPtr.advanced(by: 2)，移动到 intPtr 的下两个位置。但实际上， advanced(by:) 的参数也可以是负数，表示移动到之前的位置，甚至可以是0，就表示当前位置；

理解了这三个方法后，之前的 for 循环还可以写成这样：

```
for i in 1...10 {
    intPtr += 1
    intPtr.predecessor().pointee = i
}
```

只是，在循环结束之后， intPtr 就指向最后一个 Int 的下一个位置了。读取它，会引发未定义的行为。除了刚才介绍的这三个移动位置的方法之外， UnsafeMutablePointer 还有一个计算两个指针之间距离的方法，例如：

```
intPtr.distance(to: head) // -10
```

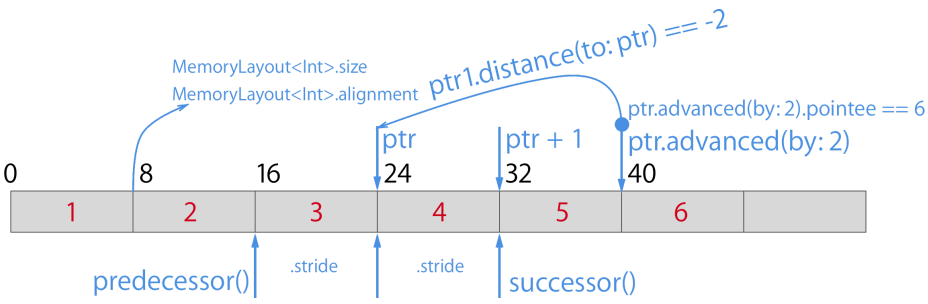
了解类型的内存布局

无论是指针的算术运算，还是调用移动指针的方法。如果我们想要知道单位移动的大小和类型自身占用的内存，在Swift里可以使用一个叫做 MemoryLayout 的 enum，它集成了C中的 sizeof 和 alignof 操作符的功能。例如：

```
MemoryLayout<Int>.size      // 8
MemoryLayout<Int>.stride    // 8
MemoryLayout<Int>.alignment // 8
```

其中， .size 表示对象自身的大小， .stride 表示对象在内存中移动一个步进的大小， .alignment 表示对象在内存中的对齐边界。在64位macOS上，可以看到它们都是8个字节。当代码中的指针运算结果不正确时，就可以通过 MemoryLayout 来查看对象的各种“单位”。

最后，用一张图表示这一节的内容，就是这样的：



What's next?

以上，就是Swift中指针类型的基础内容，在了解了这些沿袭自C的原始访问方式之后，下一节，我们来看个更Swift的方式，如何使用buffer视图改进内存访问的编程体验。

(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/248) (https://www.boxueio.com/series/interoperate-swift-with-c/ebook/250)

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

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/)