Gestión de la memoria en Swift

Una explicación sobre la gestión de la memoria y cómo aprovechar las estructuras para optimizar tus abstracciones

Introducción

En Swift, como en muchos otros lenguajes de programación, la gestión de la memoria es una parte esencial para garantizar un rendimiento óptimo y prevenir problemas como fugas de memoria o accesos inválidos a la misma. En Swift, la gestión de la memoria se lleva a cabo principalmente en dos áreas: la pila (**Stack**) y el montón (**Heap**). Paciencia aquí, acabarás familiarizado con estos dos términos pronto...

Listemos algunas diferencias entre **Heap** y **Stack**, montón y pila. No os molestéis en intentar memorizar nada, esta tabla solo busca ser una referencia, un sitio al que podamos volver si tenemos dudas sobre una determinada característica de nuestras dos nuevas mejores amigas.

Stack y Heap, cara a cara

Característica	**Stack**	**Heap**
Asignación de memoria:	Estática, se realiza durante la compilación	Dinámica, tiene lugar en tiempo de ejecución
Acceso:	Rápido, debido a la asignación y liberación automática de memoria	Un poco más lento, implica más gestión de la memoria mediante contadores de referencias o ARC
Usada para almacenar:	Tipos de datos por valor: estructuras, enumerados... *	Tipos de datos por referencia: clases, actores... *
Seguridad:	Cada hilo tiene su propia **Stack** de memoria, por lo que no pueden producirse accesos simultáneos a datos "de estado compartido" que puedan producir las famosas "conditions race". **	Todos los hilos acceden al mismo **Heap** por lo que pueden producirse "conditions race", así que los datos deben de estar protegidos ante esta casuística. **
Rendimiento:	Rendimiento muy alto. ***	Menor rendimiento debido a diversos factores. ***

(*) ¿Quiere decir esto que siempre que cree un tipo de dato mediante **Struct** sus objetos van a ser

almacenados en el **Stack**?, no, desgraciadamente no es tan sencillo.

(**) Cada hilo de ejecución tiene su propio **Stack** de memoria mientras que el **Heap** es compartido por todos los hilos. Como varios hilos de ejecución pueden asignar memoria en el **Heap** al mismo tiempo éste debe protegerse mediante bloqueos, semáforos u otros mecanismos de sincronización. Esto supone un coste bastante grande de rendimiento.

(***) La gestión de la memoria que hace Swift mediante el **Stack**, de forma estática, es muy eficiente. La pila funciona mediante un sistema de LIFO, imagen 1. Swift puede agregar la información que necesite a la pila, "Allocation", de forma muy rápida y limpiarla de ésta igual de rápido, "Deallocation".

En la Imagen 2 podemos observar otra representación gráfica de cómo Swift agregaría datos al Stack y cómo los liberaría, cuando ya no fuesen necesarios, mediante dicho sistema LIFO (Last in, First Out).

Mientras tanto, la asignación dinámica en el **Heap** (montón), implica que debamos encontrar un espacio en memoria con el tamaño apropiado y, cuando ya no la necesitemos, desasignarla adecuadamente, incurriendo en una gestión mucho más compleja, y por ende menos eficiente, que la usada para almacenar en la **Stack** (pila).

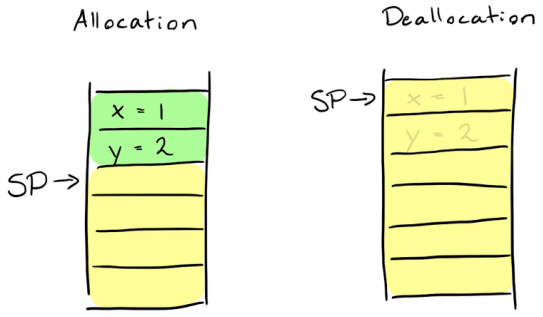


Imagen 1

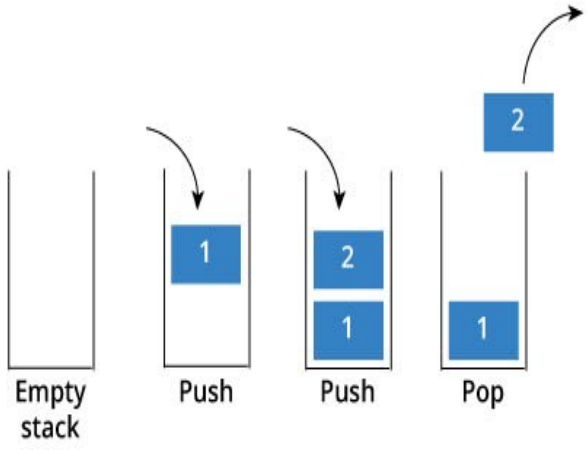


Imagen 2

Si queremos que nuestras abstracciones sean rápidas, y óptimas, necesitamos tener en cuenta todas y cada una de las cuestiones que acabamos de leer, más algunas otras como el despachado dinámico y estático de métodos que veremos en el siguiente artículo.

¿Hay código o no hay código?...

Vamos a verlo con código, usando los mismos ejemplos que puso Apple en la WWDC 2016, conferencia "Understanding Swift Performance".

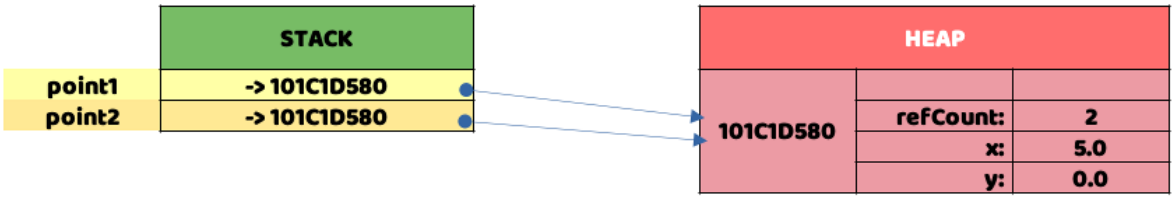
```swift
struct Point {
    var x, y: Double
}

let point1 = Point(x: 0, y: 1)
var point2 = point1
point2.x = 5
```

Este sencillo ejemplo no haría uso del Heap, ambas estructuras serían almacenadas en la Stack de memoria correspondiente. Hemos agregado una representación de cómo quedaría dicho Stack en memoria, solo se trata de una representación gráfica, sin más pretensión que la de aportar una ayuda visual a la explicación. ¿Cómo sería su comportamiento en memoria si usásemos una clase?, veámoslo con un ejemplo:

```swift
class Point {
    var x, y: Double
    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
```



Cómo podemos apreciar en la representación de memoria, para este ejemplo, Swift tiene que hacer uso del Heap para almacenar los datos del objeto creado, con todos los costes de rendimiento, seguridad, etc, que hemos visto en el bloque anterior. Además podemos observar que Swift sigue teniendo que hacer uso de la Stack correspondiente para almacenar las referencias a **101C1D580**, tanto para el objeto point1 como para el objeto point2.

101C1D580 es un referencia de memoria de ejemplo, para simular una real donde se almacenarían los datos. Muchos, al llegar hasta aquí, os preguntaréis por qué hay cuatro "espacios" de memoria cuando para "almacenar" la estructura solo hacían falta dos. Veamos, uno de los espacios, al que llamaremos **refCount**, es usado por ARC para almacenar el número de "referencias activas" que apuntan a dicha posición de memoria y el otro se usa para almacenar el tipo de objeto. Cuando hablemos del despachado dinámico y estático de métodos de clase este "espacio" cobrará su debida importancia, por ahora, obviémoslo, Swift lo necesita y lo reserva en consecuencia.

Por cierto, cuando este **refCount** llega a 0 implica que ya no está siendo "apuntado" por ningún objeto, con lo que Swift procede a reciclarlo de la memoria. ¿Alguna vez os han preguntado en una entrevista por los "retain cycles"?, ¿"circular references"?, no lo veremos hoy, pero como suele decirse: por aquí van los tiros...

Structs ineficientes y alternativas a éstos

Revisemos la siguiente implementación para aprender, con un ejemplo de Apple, de algunas trampas en las que podemos caer al implementar nuestras funcionalidades, abstracciones, etc:

```swift
enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, buble }

var cache [String: UIImage]()

func makeBalloon(_ color: Color, orientation: Orientation, tail: Tail) -> UIImage {
    let key = "\(color):\(orientation):\(tail)"
    if let image = cache[key] { return image }
    // Aquí iría el resto de implementación para devolver el UIImage correspondiente,
    // no nos vale para el ejemplo así que la obvio.
}
```

La función se ocupa de crear los típicos "globitos" que se usan en viñetas, comics, etc, para mostrar una conversación con un lazo-flecha apuntando hacia la persona que está hablando. Como tanto el color, como la orientación y el tipo de lazo son finitos y pueden parametrizarse, contabilizarse, etc, nuestro desarrollador ha usado enums para establecer estos parámetros de forma muy acertada. También ha creado un array para almacenar diccionarios y en base a una key de tipo String poder devolver su UIImage correspondiente si esta ya ha sido procesada anteriormente.

Por desgracia no ha tenido en cuenta que dichos Strings harán uso del Heap e incurrirán en conteo de referencias, necesidad de protección de la información, acceso más lento, etc.
Y aquí es dónde podríamos estar pensando: "¿Pero el String en Swift no se implementa mediante una estructura?, ¿Las estructuras no tenían paso por valor?, ¿no nos has contado que las estructuras usan el Stack?"

En realidad no, ya que esto depende de la implementación interna de String, al igual que, como veremos más adelante en este mismo artículo, la implementación de nuestros tipos de datos serán los que determinen si acaban usando la Stack o el Heap.

¿Cómo podríamos optimizar nuestro código?, Apple nos da la siguiente alternativa:

```swift
enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, buble }

struct Attributes: Hashable {
    var color: Color
    var orientation: Orientation
    var tail: Tail
}

var cache = [Attributes: UIImage]()

func makeBalloon(_ color: Color, orientation: Orientation, tail: Tail) -> UIImage {
    let key = Attributes(color: color, orientation: orientation, tail: tail)
    if let image = cache[key] { return image }
    // Aquí iría el resto de implementación para devolver el UIImage correspondiente,
    // no nos vale para el ejemplo así que la obvio.
    return UIImage()
}
```

Podemos crear una estructura en la que "almacenar" los valores que necesitamos para la creación de nuestro "balloon". Necesitamos que implemente el protocolo Hashable para que pueda hacer de clave en el diccionario. Con esto, además, evitaríamos usar claves que bien podrían no tener nada que ver con el contenido a almacenar. Pero, ¿realmente es notable el cambio?, usemos a nuestro buen amigo XCTest para realizar un par de pruebas de rendimiento:

```swift
import XCTest

enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, buble }

struct Attributes: Hashable {
    var color: Color
    var orientation: Orientation
    var tail: Tail
}

final class MemoryHeapAndStacksTests: XCTestCase {

    func testWithString() throws {
        self.measure {
            for _ in 1...1_000_000 {
                _ = "\(Color.blue):\(Orientation.left):\(Tail.buble)"
            }
        }
    }

    func testWithStruct() throws {
        self.measure {
            for _ in 1...1_000_000 {
                _ = Attributes(color: .blue, orientation: .left, tail: .buble)
            }
        }
    }
}
```

El primer test, testWithString, tardó 1.9 segundos de media en ejecutarse (se ejecuta 10 veces y nos muestra la media de todas las ejecuciones) mientras que el segundo, testWithStruct, solo 0.17 segundos. No parece poca la diferencia...