АНАЛИЗ УСТРОЙСТВА ПАМЯТИ В ЯЗЫКЕ ПРОГРАММИРОВАНИЯ SWIFT

Григорян Д.А.

Григорян Давид Арамович – старший разработчик мобильных приложений,
Компания «Ozon», г. Москва

Аннотация: *в статье анализируется устройство, принцип работы памяти в классах и в структурах языка программирования Swift.*
Ключевые слова: *Swift, iOS, macOS, ARC, memory layout, RAM, Objective-C runtime.*

Как известно, оперативная память служит для оптимального представления данных центральному процессору и другим аппаратным частям персонального компьютера. Данный подход упрощает и оптимизирует ресурсы, используемые в компьютере, и делает ее работу наиболее эффективной, сохраняя при этом высокую производительность. Память разбита, как минимум, на 3 основные области: stack, heap и global data:

- Stack содержит данные, которые определены в локальной области видимости функции,
- Global data хранит в себе данные, которые преимущественно представлены в виде глобальных переменных, строковых констант и типов метаданных,
- Heap - область памяти, которая используется для динамического выделения памяти при создании объекта. Объекты в данной области не живут на протяжении всей работы программы. Неиспользуемые объекты уничтожаются и адреса, в которых хранились уничтоженные объекты, используются повторно для создания других объектов. Объекты, созданные в данной области памяти, не привязаны к какой-либо функции или программе в целом.

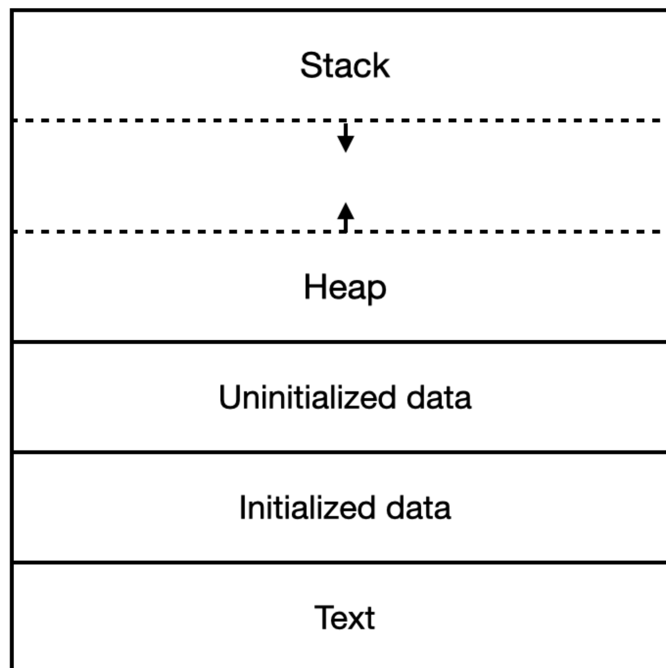

Рис. 1. Структура оперативной памяти

Рассмотрим схему памяти для класса ***Point*** в языке программирования Swift. Класс, помимо перечисленных свойств, содержит в себе поля, такие как **InlineRefCounts** и **isa**-указатель. **InlifeRefCounts** - это структура, которая содержит в себе количество объектов, которые ссылаются на данный объект класса с помощью strong и unowned ссылок. В случае если данный объект удерживается слабой ссылкой, то в классе появляется дополнительное поле **SideTableRefCounts,** которое представляет собой структуру.

```swift
class Point {
    let x: Int
    let y: Int

    init(x: Int, y: Int) {
        self.x = x
        self.y = y
    }
}
```

Bytes	Name
0	isa
8	InlineRefCounts
16	x
24	y

Рис. 2. Служебная информация для класса Point

Если рассмотреть распределение памяти для класса **Point** с помощью Objective-C runtime, то мы увидим, что первые восемь байт занимает **isa** указатель, далее идет **InlineRefCounts** со значением 3. Значение **InlineRefCounts** равняется трем, так как по умолчанию объекты создаются с **refCount** равным единице и так же происходит дополнительная инкрементация за счет присвоения этого объекта локальной переменной.

```swift
let point = Point(x: 10, y: 20)
let pointer = unsafeBitCast(point, to: UnsafeRawPointer.self)

for index in 0..<class_getInstanceSize(Point.self) {
    let byte = pointer.load(fromByteOffset: index, as: UInt8.self)
    print("byte \(index): \(byte)")
}
```

0 8 16 24 32

192	218	19	0	1	0	0	0	3	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	20	0	0	0	0	0	0	0

Рис. 3. Распределение памяти для класса Point

Необходимо отметить, что Swift классы содержат в себе, те же самые поля, что и стандартные Objective-C классы. Помимо этого, в Swift классах также содержатся дополнительные свойства.

Objective-C Classes

Class isa
Class super_class
const char *name
long version
long info
long instance_size
struct objc_ivar_list *ivars
struct objc_method_list **methodLists
struct objc_cache *cache
struct objc_protocol_list *protocols

Swift Classes

uint32_t flags
uint32_t instanceAddressOffset
uint32_t instanceSize
uint16_t instanceAlignMask
uint16_t reserved

uint32_t classSize
uint32_t classAddressOffset
void *description

Рис. 4. Служебная информация в Objective-C и Swift классах

Когда происходит вызов метода в Swift, выполняется поиск в таблице методов. Принцип работы следующий:

1) Из объекта извлекается **isa** указатель,
2) Из полученного указателя извлекается нужный метод, передав в указатель на класс соответствующий сдвиг,
3) Далее осуществляется вызов полученного метода, передав в качестве аргумента искомый объект.

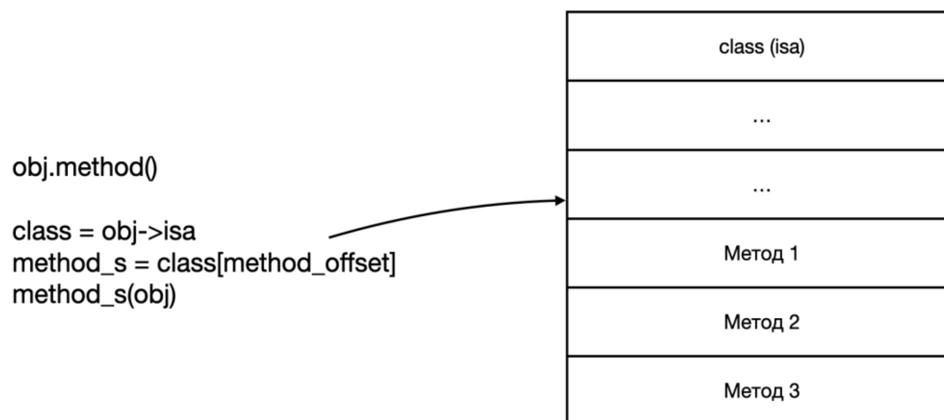

Рис. 5. Процедура поиска и вызова методов в Swift

В языке программирования Swift доступно около восьми типов указателей:

- UnsafeMutablePointer<T>
- UnsafePointer<T>
- UnsafeMutableBufferPointer<T>
- UnsafeBufferPointer<T>
- UnsafeMutableRawPointer
- UnsafeRawPointer
- UnsafeMutableRawBufferPointer
- UnsafeRawBufferPointer

Изменяемые указатели (mutable) позволяют записывать любые данные в память.

"Сырые" указатели (или не типизированные, raw) позволяют пользователю вначале указать тип данных, с которым он будет работать в памяти, и потом уже выполнять нужные операции.

Буферизированные указатели позволяют работать с данными как с коллекцией.

Strideable указатели (такие как UnsafeMutablePointer, UnsafePointer, UnsafeMutableRawPointer и UnsafeMutableRawBufferPointer) позволяют выполнять стандартные арифметические операции над указателями.

Для определения размера инициализированного объекта структуры мы можем воспользоваться специальным классом с обобщенным типом - **MemoryLayout**, указав в качестве обобщенного типа необходимую структуру. У класса **MemoryLayout** имеется статическое свойство, **size,** которое возвращает нам размер структуры.

Для структуры **Point**, вызов свойства **size** вернет значение, равное 16 байт. Данное значение, в большинстве случаев, рассчитывается с помощью простой формулы: *(количество свойств в структуре) * (MemoryLayout<StructureName>.alignment).*

Однако, существуют исключения которые позволяют оптимизировано и более эффективно использовать память в языке программирования Swift. Рассмотрим пример с той же самой структурой, только с дополнительным свойством, тип которого определен как boolean. Несмотря на то, что тип boolean фактически занимает 1 бит памяти (где 1 это истина, а 0 - ложь), в классической схеме, в памяти используется 1 байт для операций чтения и записи, так как адресация происходит с размером, равным машинному слову.

В Swift размер структуры **Point** будет зависеть от расположения булевого свойства в структуре. Если оно располагается не в конце структуры, то ее размер будет 24 байта. Если же расположить булевое свойство в конец структуры, то размер структуры примет значение 17 байт (где размер первых двух полей с типом Int это 16 байт + 1 байт для булевого поля).

Заключение

В данной статье были рассмотрены основные механизмы работы памяти для классов и структур в языке программирования Swift. Несмотря на то, что Swift это новый язык, он унаследовал основную часть механизмов из Objective-C, расширив свой функционал с использованием ссылочной семантики и семантики значений.

Список литературы

1. Официальная документация компании Apple, Memory Layout [Электронный ресурс], 2022. Режим доступа: https://developer.apple.com/documentation/swift/memorylayout/ (дата обращения: 22.03.2022).
2. *Mike Ash.* Конференция GOTO, Exploring Swift Memory Layout. [Электронный ресурс], 2016. Режим доступа: https://academy.realm.io/posts/goto-mike-ash-exploring-swift-memory-layout/ (дата обращения: 22.03.2022).
3. *Mike Ash*. Персональный блог, Exploring Swift Memory Layout. [Электронный ресурс], 2014. Режим доступа: https://www.mikeash.com/pyblog/friday-qa-2014-07-18-exploring-swift-memory-layout.html/ (дата обращения: 22.03.2022).