Công nghệ càng ngày càng phát triểu, Swift cũng vậy. Nó xử lý tốt việc quản lý bộ nhớ như cấp phát và lấy lại vung nhớ. Nếu các bạn đã biết về objective-C thì hẳng cũng đã nghe qua về ARC(Automatic Reference Countring) nó chính là kỹ thuật để quyết định xem vùng nhớ vào sẽ được tồn tại và bị lấy đi. Bài này sẽ đề cập đến ARC và cơ chế quản lý bộ nhớ trong Swift.

Càng ngày cơ chế ARC càng được cải tiền, bạn không cần chỉ rõ số lượng phần tử ánh xạ của một đối tượng. Tuy nhiên, bạn cần chỉ rõ mỗi quan hệ giữa các đối tượng để tránh memory leaks. Điều này thường được các lập trình viên chưa có kinh nghiệm bỏ qua.

Ví dụ này đề cập đến các phần sau:

* Cách ARC hoạt động.
* Reference cycles là gì, và cách loại bỏ nó.
* Một ví dụ về reference cycle và sử dụng Xcode visualization để phân tích.
* Làm thế nào để kết hợp thạm trị và tham chiếu.

Bắt đầu:

Mở Xcode và chọn File\New\Playground. Chọn iOS Platform, tên MemoryManagement -> Next. Tiếp theo xoá đoạn code thừa khi tạo.

Thêm các đoạn mã sau:

class User {

var name: String

init(name: String) {

self.name = name

print("User \(name) is initialized")

}

deinit {

print("User \(name) is being deallocated")

}

}

Đoạn mã trên định nghĩa ra class User và khởi tạo một biến có kiểu User. Class User có một thuộc tính là name và một hàm khởi tạo(chỉ được gọi sau khi được cấp phát vùng nhớ) và hàm deinit(được gọi ngay trước khi huỷ vùng nhớ). Câu lệnh print để biết khi nào đối tượng được khởi tạo và huỷ.

Chú ý là chương trình sẽ in ra”User John is initialized”. Tuy nhiên hàm deinit không được gọi bởi vì phạm vi hoạt động sau khi khởi tạo chưa có. Đơn giản chỉ cần sử lại phạm vi hoạt động như sau:

do

{

let user1 = User(name: "John")

}

Sau khi sửa lại thì có thể thấy hàm print trong deinit đã hoạt động. Vòng đời của một đối tượng gồm 5 giai đoạn sau:

* Cấp phát vùng nhớ(bộ nhớ được cấp phát vùng nhớ ở stack(struct) hoặc heap(class))
* Hàm khởi tạo(đoạn mã trong init)
* Sử dụng object
* Deinit(đoạn mã ở trong deinit được thực hiện)
* Huỷ vùng nhớ(vùng nhớ được trả lại cho stack hoặc heap)

Chỉ có một điều cần chú ý là đôi khi deinit và deallocate được sử dụng để thay thế cho nhau nhưng thực chất nó là 2 giai đoạn khác nhau trong vòng đời của 1 đối tượng.

Reference counting là một cơ chế lấy lại vùng nhớ của một đối tượng khi nó không còn sử dụng nữa. Vậy lây lại vùng nhớ khi nào là hợp lý, nếu lúc khác cần dung thì sao?. Vì thế reference counting hoạt động trên cơ chế đếm số lượng ánh xạ đến đối tượng => Nếu số lượng ánh xạ là 0 thì vùng nhớ của đối tượng sẽ được giải phóng.

Khi khởi tạo một đối tượgn User, biến đó có reference count là 1, khi ra khỏi vùng được khai báo thì count giảm xuống 0 => đối tượng sẽ deinit và deallocates.

Reference cycles

Đây là một trong những trường hợp gây ra leaking memory. Có thể hiểu đơn giản là có 2 đối tượng tuy không cần sử dụng đến nó nữa nhưng cả 2 ánh xạ lẫn nhau => cả 2 đều không được giải phóng.

“”

Hình vẽ trên được gọi là một strong reference cycle. Nhìn trên hình có thể thấy reference count không bao giờ giảm xuống 0 => object1 và object2 không bao giờ được giải phóng.

Thêm các đoạn mã sau:

class Phone {

let model: String

var owner: User?

init(model: String) {

self.model = model

print("Phone \(model) is initialized")

}

deinit {

print("Phone \(model) is being deallocated")

}

}

do {

let user1 = User(name: "John")

let iPhone = Phone(model: "iPhone 6s Plus")

}

Đoạn mã trên thêm một class có tên Phone, và khởi tạo một biến user1.

Class này với 2 tham số là model và owner(chủ sử hữu) và 2 hàm init và deinit.

Tiếp theo thêm đoạn mã sau vào class User ngay dưới thuộc tính name:

private(set) var phones: [Phone] = []

func add(phone: Phone) {

phones.append(phone)

phone.owner = self

}

Đoạn mã trên khai báo một biến có kiểu mảng các phần tử Phone với access control là private và chỉ Set và gán owner cho điện thoại chính là user đó.

Sửa đoạn mã sau để thấy reference cycle.

do {

let user1 = User(name: "John")

let iPhone = Phone(model: "iPhone 6s Plus")

user1.add(phone: iPhone)

}

Khi thêm đoạn mã trên thì không thấy in ra deallcate vì ở đây thêm iPhone và user1, và nó tự động gán thuộc tính owner(chủ sở hữu) là user1. Vậy một strong reference cycle giữa 2 đối tượng đã làm cho cả 2 đối tượng không được giải phóng.

Các cách khắc phục

Weak References

Để phá vỡ strong reference cycle, có thể chỉ rõ quan hệ giữa các đối tượng là weak, vì khi sử dụng weak thì reference count sẽ không bị tăng lên. Khi khai báo một biến với từ khoá weak thì nó sẽ là optional vì khi đối tượng đó mà có reference count bằng 0 thì nó sẽ giải phóng vùng nhớ => nil.

“”

Ở bức ảnh trên, mũi tên đứt màu xám thể hiện cho tham chiếu yếu (weak reference). Đầu tiên khi khởi tạo variable1 thì object1 số lượng tham chiếu tăng lên 1 và khởi tạo biên variable2 thì object2 số lượng tham chiếu tăng lên 1, object1 tham chiếu đến object2 => object2 có số lượng tham chiếu là 2. Khi huỷ biến variable1 thì object1 giảm 1 => số lượng tham chiếu là 0 nên nó sẽ được giải phóng vì object1 được giải phóng nên object2 có số lượng tham chiếu giảm đi 1 nhưng vẫn chưa được giải phóng vì còn variable2 tham chiếu đến. Đến khi variable2 huỷ thì => giảm tham chiếu object2 => số lượng là 0 => giải phóng.

Thay vì việc khai báo mặc định bây giờ thêm weak ở trên biến owner

weak var owner: User?

Unowned References

Có một cách khác để không làm tăng số lượng tham chiếu là dùng: unowned.

Vậy điểm khác biệt giữa weak và owned là gì?. Một tham chiếu weak **luôn luôn** là optional và tự động nil khi đối tượng deinit. Đó là lý do tại sao khi khai báo biến weak thì phải dùng var không là lỗi ngay. Ngược lại tham chiếu unowned không bao giờ là kiểu optional. Thôi vào code luôn cho dễ hiểu:

class CarrierSubscription {

let name: String

let countryCode: String

let number: String

let user: User

init(name: String, countryCode: String, number: String, user: User) {

self.name = name

self.countryCode = countryCode

self.number = number

self.user = user

print("CarrierSubscription \(name) is initialized")

}

deinit {

print("CarrierSubscription \(name) is being deallocated")

Lớp CarrierSubscription có 4 thuộc tính: name, countryCode, number, user.

Thêm đoạn mã sau vào class User:

var subscriptions: [CarrierSubscription] = []

}

Thêm thuộc tính subcriptions là một mảng của các đối tượng CarrierSubscription. Thêm đoạn mã sau vào class Phone

var carrierSubscription: CarrierSubscription?

func provision(carrierSubscription: CarrierSubscription) {

self.carrierSubscription = carrierSubscription

}

func decommission() {

self.carrierSubscription = nil

}

}

Đoạn mã trên thêm vào một biến optional và 2 funcs provision và decommission một thuê bao trên điện thoại.

Tiếp theo vào hàm init của class CarrierSubscription trước hàm print thêm đoạn mã sau:

user.subscriptions.append(self)

Đoạn mã này để chắc chắn hãng thuê bao được thêm vào mảng subscriptions.

Cuối cùng chay đổi đoạn mã trong do{} như sau:

do {

let user1 = User(name: "John")

let iPhone = Phone(model: "iPhone 6s Plus")

user1.add(phone: iPhone)

let subscription1 = CarrierSubscription(name: "TelBel", countryCode: "0032", number: "31415926", user: user1)

iPhone.provision(carrierSubscription: subscription1)

}

Có thể thấy cả user, phone và subscription đều không được deallocated. Ảnh dưới đây sẽ lý giải.

Mỗi tham chiều từ user1 đến subscription1 hoặc từ subscription1 đến user1 nên để unowned để phá vỡ reference cycle. Câu hỏi đặt ra là chọn cái nào đẻ cho từ khoá unowned. Cách bạn chỉ cần nghĩ đơn giản là một user có thể sử dụng nhiều loại mạng khác nhau nhưng mạng không thể sở hữu được user. Hơn nữa khi người dùng đăng ký nhà mạng xong thì cái giấy đăng ký đấy không thể chuyển sang cho người khác nên để let user là hợp lý. Thêm unowned vào trước thuộc tính user của class CarrierSubcription:

unowned let user: User

Chỉ cần thay đổi đơn giản là đã phá vỡ được reference cycle:

“”

Reference Cyles with Closures

Reference Cycle xảy ra khi khác đối tượng tham chiếu lẫn nhau. Closures cũng là kiểu tham chiếu và cũng có thể gây ra lỗi trên. Closures giữ lại đối tượng chỗ nó khai báo để thực hiện các khối lệnh ở trong.

Ví dụ nếu sử dụng thuộc tính của một lớp trong closures thì các bạn hay sử dụng từ khoá self, chính nó sẽ tạo một tham chiếu và làm cho đối tượng không thể deallocate.

“”

Thêm đoạn mã sau vào class CarrierSubscription ở dưới thuộc tính user

lazy var completePhoneNumber: () -> String = {

self.countryCode + " " + self.number

}

Thuộc tính sẽ tính toàn và trả về mã quốc gia và số điện thoại. Thuộc tính này sử dụng từ khoá lazy có nghĩa là nó sẽ không gán cho đến khi sử dụng lần đầu. Ở đây sử dụng từ khoá self vì trong block.

Thêm đoạn mã sau vào do{}

print(subscription1.completePhoneNumber())

Bạn có thể thấy là user1 và iPhone deallocate nhưng CarrierSubscription thì không được deallocate vì sau:

“”

Nếu bạn hiểu về Swift thì nó cũng rất đơn giản, cách dễ nhất để phá vỡ break strong reference trong closures chỉ cần một capture list ở nơi định nghĩa closure với từ khoá unowned hoặc weak như sau:

lazy var completePhoneNumber: () -> String = { [unowned self] in

return self.countryCode + " " + self.number

}

Khi chỉnh sửa lại đoạn mã thì có thể thấy CarrierSubscription được deallocate