内存管理

@M了个J

https://github.com/CoderMJLee

http://cnblogs.com/mjios

实力IT教育 www.520it.com

码拉松

小码哥教育@M了个J

内存管理

- 跟OC一样，Swift也是采取基于引用计数的ARC内存管理方案（针对堆空间）

- Swift的ARC中有3种引用
- 强引用（strong reference）：默认情况下，引用都是强引用

- 弱引用（weak reference）：通过`weak`定义弱引用
  - ✓ 必须是可选类型的`var`，因为实例销毁后，ARC会自动将弱引用设置为`nil`
  - ✓ ARC自动给弱引用设置`nil`时，不会触发属性观察器

- 无主引用（unowned reference）：通过`unowned`定义无主引用
  - ✓ 不会产生强引用，实例销毁后仍然存储着实例的内存地址（类似于OC中的`unsafe_unretained`）
  - ✓ 试图在实例销毁后访问无主引用，会产生运行时错误（野指针）
    - **`Fatal error: Attempted to read an unowned reference but object 0x0 was already deallocated`**

weak、unowned的使用限制

- weak、unowned只能用在类实例上面

```
protocol Livable : AnyObject {}
class Person {}

weak var p0: Person?
weak var p1: AnyObject?
weak var p2: Livable?

unowned var p10: Person?
unowned var p11: AnyObject?
unowned var p12: Livable?
```

小码哥教育 SEEMYGO

Autoreleasepool

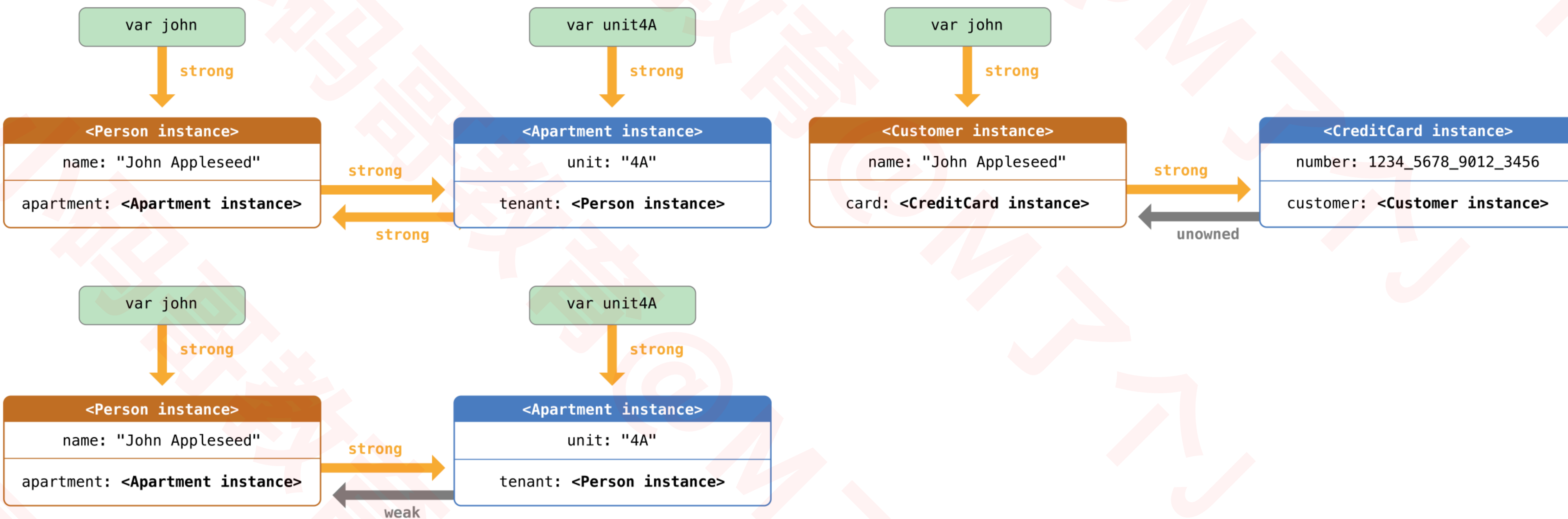
```
public func autoreleasepool<Result>(invoking body: () throws -> Result) rethrows -> Result
```

```
autoreleasepool {
    let p = MJPerson(age: 20, name: "Jack")
    p.run()
}
```

小码哥教育@MJ^^

循环引用（Reference Cycle）

- weak、unowned 都能解决循环引用的问题，unowned 要比 weak 少一些性能消耗
- 在生命周期中可能会变为 nil 的使用 weak
- 初始化赋值后再也不会变为 nil 的使用 unowned



闭包的循环引用

- 闭包表达式默认会对用到的外层对象产生额外的强引用（对外层对象进行了retain操作）
- 下面代码会产生循环引用，导致Person对象无法释放（看不到Person的deinit被调用）

```swift
class Person {
    var fn: (() -> ())?
    func run() { print("run") }
    deinit { print("deinit") }
}
func test() {
    let p = Person()
    p.fn = { p.run() }
}
test()
```

- 在闭包表达式的捕获列表声明weak或unowned引用，解决循环引用问题

```swift
p.fn = {
    [weak p] in
    p?.run()
}
```

```swift
p.fn = {
    [unowned p] in
    p.run()
}
```

```swift
p.fn = {
    [weak wp = p, unowned up = p, a = 10 + 20] in
    wp?.run()
}
```

闭包的循环引用

小码哥教育 SEEMYGO

- 如果想在定义闭包属性的同时引用self，这个闭包必须是lazy的（因为在实例初始化完毕之后才能引用self）

```
class Person {
    lazy var fn: (() -> ()) = {
        [weak self] in
        self?.run()
    }
    func run() { print("run") }
    deinit { print("deinit") }
}
```

- 左边的闭包fn内部如果用到了实例成员（属性、方法）
  - 编译器会强制要求明确写出self

- 如果lazy属性是闭包调用的结果，那么不用考虑循环引用的问题（因为闭包调用后，闭包的生命周期就结束了）

```
class Person {
    var age: Int = 0
    lazy var getAge: Int = {
        self.age
    }()
    deinit { print("deinit") }
}
```

@escaping

- 非逃逸闭包、逃逸闭包，一般都是当做参数传递给函数
- 非逃逸闭包：闭包调用发生在函数结束前，闭包调用在函数作用域内
- 逃逸闭包：闭包有可能在函数结束后调用，闭包调用逃离了函数的作用域，需要通过@escaping声明

```swift
import Dispatch
typealias Fn = () -> ()
```

```swift
// fn是非逃逸闭包
func test1(_ fn: Fn) { fn() }
```

```swift
// fn是逃逸闭包
var gFn: Fn?
func test2(_ fn: @escaping Fn) { gFn = fn }
```

```swift
// fn是逃逸闭包
func test3(_ fn: @escaping Fn) {
    DispatchQueue.global().async {
        fn()
    }
}
```

```swift
class Person {
    var fn: Fn
    // fn是逃逸闭包
    init(fn: @escaping Fn) {
        self.fn = fn
    }
    func run() {
        // DispatchQueue.global().async也是一个逃逸闭包
        // 它用到了实例成员（属性、方法），编译器会强制要求明确写出self
        DispatchQueue.global().async {
            self.fn()
        }
    }
}
```

小码哥教育 SEEMYGO

逃逸闭包的注意点

- 逃逸闭包不可以捕获inout参数

```
typealias Fn = () -> ()
func other1(_ fn: Fn) { fn() }
func other2(_ fn: @escaping Fn) { fn() }
func test(value: inout Int) -> Fn {
    other1 { value += 1 }

    // error: 逃逸闭包不能捕获inout参数
    other2 { value += 1 }

    func plus() { value += 1 }
    // error: 逃逸闭包不能捕获inout参数
    return plus
}
```

内存访问冲突（Conflicting Access to Memory）

- 内存访问冲突会在两个访问满足下列条件时发生：
- 至少一个是写入操作
- 它们访问的是同一块内存
- 它们的访问时间重叠（比如在同一个函数内）

```
// 不存在内存访问冲突
func plus(_ num: inout Int) -> Int { num + 1 }
var number = 1
number = plus(&number)
```

```
// 存在内存访问冲突
// Simultaneous accesses to 0x0, but modification requires exclusive access
var step = 1
func increment(_ num: inout Int) { num += step }
increment(&step)
```

```
// 解决内存访问冲突
var copyOfStep = step
increment(&copyOfStep)
step = copyOfStep
```

内存访问冲突

```swift
func balance(_ x: inout Int, _ y: inout Int) {
    let sum = x + y
    x = sum / 2
    y = sum - x
}
var num1 = 42
var num2 = 30
balance(&num1, &num2)  // OK
balance(&num1, &num1) // Error
```

```swift
struct Player {
    var name: String
    var health: Int
    var energy: Int
    mutating func shareHealth(with teammate: inout Player) {
        balance(&teammate.health, &health)
    }
}
var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Maria", health: 5, energy: 10)
oscar.shareHealth(with: &maria)  // OK
oscar.shareHealth(with: &oscar) // Error
```

```swift
var tulpe = (health: 10, energy: 20)
// Error
balance(&tulpe.health, &tulpe.energy)

var holly = Player(name: "Holly", health: 10, energy: 10)
// Error
balance(&holly.health, &holly.energy)
```

内存访问冲突

- 如果下面的条件可以满足，就说明重叠访问结构体的属性是安全的
  - 你只访问实例存储属性，不是计算属性或者类属性
  - 结构体是局部变量而非全局变量
  - 结构体要么没有被闭包捕获要么只被非逃逸闭包捕获

```
// Ok
func test() {
    var tulpe = (health: 10, energy: 20)
    balance(&tulpe.health, &tulpe.energy)

    var holly = Player(name: "Holly", health: 10, energy: 10)
    balance(&holly.health, &holly.energy)
}
test()
```

小码哥教育@MJ

指针

- Swift中也有专门的指针类型，这些都被定性为“Unsafe”（不安全的），常见的有以下4种类型
  - UnsafePointer<Pointee> 类似于 const Pointee *
  - UnsafeMutablePointer<Pointee> 类似于 Pointee *
  - UnsafeRawPointer 类似于 const void *
  - UnsafeMutableRawPointer 类似于 void *

```swift
var age = 10
func test1(_ ptr: UnsafeMutablePointer<Int>) {
    ptr.pointee += 10
}
func test2(_ ptr: UnsafePointer<Int>) {
    print(ptr.pointee)
}
test1(&age)
test2(&age) // 20
print(age) // 20
```

```swift
var age = 10
func test3(_ ptr: UnsafeMutableRawPointer) {
    ptr.storeBytes(of: 20, as: Int.self)
}
func test4(_ ptr: UnsafeRawPointer) {
    print(ptr.load(as: Int.self))
}
test3(&age)
test4(&age) // 20
print(age) // 20
```

小码哥教育@MJ

小码哥教育 SEEMYGO

指针的应用示例

```
var arr = NSArray(objects: 11, 22, 33, 44)
arr.enumerateObjects { (obj, idx, stop) in
    print(idx, obj)
    if idx == 2 { // 下标为2就停止遍历
        stop.pointee = true
    }
}
```

```
var arr = NSArray(objects: 11, 22, 33, 44)
for (idx, obj) in arr.enumerated() {
    print(idx, obj)
    if idx == 2 {
        break
    }
}
```

小码哥教育 SEEMYGO

获得指向某个变量的指针

```
var age = 11
var ptr1 = withUnsafeMutablePointer(to: &age) { $0 }
var ptr2 = withUnsafePointer(to: &age) { $0 }
ptr1.pointee = 22
print(ptr2.pointee) // 22
print(age) // 22

var ptr3 = withUnsafeMutablePointer(to: &age) { UnsafeMutableRawPointer($0) }
var ptr4 = withUnsafePointer(to: &age) { UnsafeRawPointer($0) }
ptr3.storeBytes(of: 33, as: Int.self)
print(ptr4.load(as: Int.self)) // 33
print(age) // 33
```

小码哥教育 SEEMYGO

获得指向堆空间实例的指针

```
class Person {}
var person = Person()
var ptr = withUnsafePointer(to: &person) { UnsafeRawPointer($0) }
var heapPtr = UnsafeRawPointer(bitPattern: ptr.load(as: UInt.self))
print(heapPtr!)
```

小码哥教育@M了个J

小码哥教育 SEEMYGO

创建指针

```swift
var ptr = UnsafeRawPointer(bitPattern: 0x100001234)
```

```swift
// 创建
var ptr = malloc(16)
// 存
ptr?.storeBytes(of: 11, as: Int.self)
ptr?.storeBytes(of: 22, toByteOffset: 8, as: Int.self)
// 取
print((ptr?.load(as: Int.self))!) // 11
print((ptr?.load(fromByteOffset: 8, as: Int.self))!) // 22
// 销毁
free(ptr)
```

```swift
var ptr = UnsafeMutableRawPointer.allocate(byteCount: 16, alignment: 1)
ptr.storeBytes(of: 11, as: Int.self)
ptr.advanced(by: 8).storeBytes(of: 22, as: Int.self)
print(ptr.load(as: Int.self)) // 11
print(ptr.advanced(by: 8).load(as: Int.self)) // 22
ptr.deallocate()
```

创建指针

```
var ptr = UnsafeMutablePointer<Int>.allocate(capacity: 3)
ptr.initialize(to: 11)
ptr.successor().initialize(to: 22)
ptr.successor().successor().initialize(to: 33)

print(ptr.pointee) // 11
print((ptr + 1).pointee) // 22
print((ptr + 2).pointee) // 33

print(ptr[0]) // 11
print(ptr[1]) // 22
print(ptr[2]) // 33

ptr.deinitialize(count: 3)
ptr.deallocate()
```

小码哥教育@MJ

创建指针

```
class Person {
    var age: Int
    var name: String
    init(age: Int, name: String) {
        self.age = age
        self.name = name
    }
    deinit { print(name, "deinit") }
}
```

```
var ptr = UnsafeMutablePointer<Person>.allocate(capacity: 3)
ptr.initialize(to: Person(age: 10, name: "Jack"))
(ptr + 1).initialize(to: Person(age: 11, name: "Rose"))
(ptr + 2).initialize(to: Person(age: 12, name: "Kate"))
// Jack deinit
// Rose deinit
// Kate deinit
ptr.deinitialize(count: 3)
ptr.deallocate()
```

小码哥教育 SEEMYGO

指针之间的转换

```
var ptr = UnsafeMutableRawPointer.allocate(byteCount: 16, alignment: 1)

ptr.assumingMemoryBound(to: Int.self).pointee = 11
(ptr + 8).assumingMemoryBound(to: Double.self).pointee = 22.0

print(unsafeBitCast(ptr, to: UnsafePointer<Int>.self).pointee) // 11
print(unsafeBitCast(ptr + 8, to: UnsafePointer<Double>.self).pointee) // 22.0

ptr.deallocate()
```

- unsafeBitCast是忽略数据类型的强制转换，不会因为数据类型的变化而改变原来的内存数据
  - 类似于C++中的reinterpret_cast

```
class Person {}
var person = Person()
var ptr = unsafeBitCast(person, to: UnsafeRawPointer.self)
print(ptr)
```