Swift内存泄漏详解([weak self]使用场景)

码农渼 简书作者

0.978 2020-06-22 09:53

```swift
// 这是一个非逃逸闭包(立即执行),因此不需要弱引用
func uiViewAnimate() {
        UIView.animate(withDuration: 3.0) { self.view.backgroundColor = .red }
}
```

```swift
// 同样对于高阶函数,也是非逃逸的,因此不需要弱引用[weak self]
    func higherOrderFunctions() {
        let numbers = [1,2,3,4,5,6,7,8,9,10]
        numbers.forEach { self.view.tag = $0 }
        numbers.forEach { num in
            DispatchQueue.global().async {
                print(num)
            }
        }
        _ = numbers.filter { $0 == self.view.tag }
    }
```

```swift
/*这会导致 Controller 内存泄漏, 因为我们没有立即执行动画。我们把它存储为属性
在闭包中使用了self, 同时self也引用了闭包*/
func leakyViewPropertyAnimator() {
        // color won't actually change, because we aren't executing the animation
        let anim = UIViewPropertyAnimator(duration: 2.0, curve: .linear) { self.view.backgroundColor = .red }
        anim.addCompletion { _ in self.view.backgroundColor = .white }
        self.anim = anim
    }
```

```swift
/*如果我们将对要直接操作的属性(view)的引用传递给闭包, 而不是使用self,
*将不会导致内存泄漏, 即使不使用[weak self]*/
func nonLeakyViewPropertyAnimator1() {
        let view = self.view
        // color won't actually change, because we aren't executing the animation
        let anim = UIViewPropertyAnimator(duration: 2.0, curve: .linear) { view?.backgroundColor = .red }
        anim.addCompletion { _ in view?.backgroundColor = .white }
        self.anim = anim
    }
```

```swift
// 如果我们立即启动动画, 没有在其他地方进行强引用, 它不会泄漏控制器, 即使没有使用[weak self]
func nonLeakyViewPropertyAnimator2() {
        let anim = UIViewPropertyAnimator(duration: 2.0, curve: .linear) { self.view.backgroundColor = .red }
        anim.addCompletion { _ in self.view.backgroundColor = .white }
        anim.startAnimation()
    }
```

```swift
// 如果我们存储一个闭包, 它就会逃逸, 如果我们不使用[weak self], 它就会造成循环引用从而导致内存泄漏
func leakyDispatchQueue() {
        let workItem = DispatchWorkItem { self.view.backgroundColor = .red }
        DispatchQueue.main.asyncAfter(deadline: .now() + 1.0, execute: workItem)
        self. workItem = workItem
    }
```

```swift
// 如果我们立即执行闭包而不存储它, 就不需要[weak self]
func nonLeakyDispatchQueue() {
        DispatchQueue.main.asyncAfter(deadline: .now() + 1.0) {
            self.view.backgroundColor = .red
        }

        DispatchQueue.main.async {
            self.view.backgroundColor = .red
        }

        DispatchQueue.global(qos: .background).async {
            print(self.navigationItem.description)
        }
    }
```

```swift
/*此计时器将阻止控制器释放, 因为:
*1、定时器重复执行
*2、self在闭包中引用, 而没有使用[weak self]
*如果这两个条件中的任何一个是错误的, 它都不会引起问题*/
func leakyTimer() {
        let timer = Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true) { timer in
            let currentColor = self.view.backgroundColor
            self.view.backgroundColor = currentColor == .red ? .blue : .red
        }
        timer.tolerance = 0.5
        RunLoop.current.add(timer, forMode: RunLoop.Mode.common)
    }
```

```swift
/*类似于UIViewPropertyAnimator, 如果我们存储一个URLSession任务而不立即执行它, 将导致内存泄漏, 除非我们使用[weak self]*/
func leakyAsyncCall() {
        let url = URL(string: "https://www.github.com")!
        let task = URLSession.shared.downloadTask(with: url) { localURL, _, _ in
            guard let localURL = localURL else { return }
            let contents = (try? String(contentsOf: localURL)) ?? "No contents"
            print(contents)
            print(self.view.description)
        }
        self.closureStorage = task
    }
```

```swift
/*如果立即执行URLSession任务, 但设置了较长的超时间隔, 则会延迟释放
    *直到您取消任务、获得响应或超时。使用[weak self]可以防止延迟
    *注意: url使用端口81有助于模拟请求超时 */
func delayedAllocAsyncCall() {
        let url = URL(string: "https://www.google.com:81")!

        let sessionConfig = URLSessionConfiguration.default
        sessionConfig.timeoutIntervalForRequest = 999.0
        sessionConfig.timeoutIntervalForResource = 999.0
        let session = URLSession(configuration: sessionConfig)

        let task = session.downloadTask(with: url) { localURL, _, error in
            guard let localURL = localURL else { return }
            let contents = (try? String(contentsOf: localURL)) ?? "No contents"
            print(contents)
            print(self.view.description)
        }
        task.resume()
    }
```

```swift
/*这里导致了一个循环引用, 因为闭包和“self”相互引用而不是用[weak self],注意这里需要'@escaping', 因为我们正在保存闭包(导致逃逸)以备以后使用。*/
func savedClosure() {

        func run(closure: @escaping () -> Void) {
            closure()
            self.closureStorage = closure // 'self' stores the closure
        }

        run {
            self.view.backgroundColor = .red // the closure references 'self'
        }
    }
```

```swift
/*这里不需要[weak self], 因为闭包没有转义 (没有存储在任何地方) 。*/
func unsavedClosure() {

        func run(closure: () -> Void) {
            closure()
        }

        run {
            self.view.backgroundColor = .red // the closure references 'self'
        }
    }
```

```swift
/*直接将闭包函数传递给closure属性是很方便的, 但会导致控制器内存泄漏!
*原因: self被闭包隐式捕获(在swift中,如UIViewController的viewDidLoad中,可以直接修改view.backgroundColor,而不需要self.view.backgroundColor), self拥有拥有printingButton, 从而创建一个引用循环*/
func setupLeakyButton() {
        printingButton?.closure = printer
    }

func printer() {
        print("Executing the closure attached to the button")
    }
```

```swift
// 需要[weak self]来打破这个循环, 即使它会使语法更难看
func setupNonLeakyButton() {
        printingButton?.closure = { [weak self] in self?.printer() }
    }
func printer() {
        print("Executing the closure attached to the button")
    }
```

```swift
/*尽管这个Dispatch是立即执行的, 但是有一个sempaphore(信号量)阻塞了闭包的返回, 并且超时很长。这不会导致泄漏, 但会导致延迟释放“self”, 因为引用self时没有使用“weak”或“unowned”关键字*/
 func delayedAllocSemaphore() {
        DispatchQueue.global(qos: .userInitiated).async {
            let semaphore = DispatchSemaphore(value: 0)
            print(self.view.description)
            _ = semaphore.wait(timeout: .now() + 99.0)
        }
    }
```

```swift
/*尽管使用了[weak self], 这个嵌套的闭包还是会泄漏, 因为与DispatchWorkItem关联的转义闭包使用其嵌套闭包的[weak self]关键字创建对“self”的强引用。因此, 我们需要将[弱自我]提升一级, 到最外层的封闭处(DispatchWorkItem
func leakyNestedClosure() {

        let workItem = DispatchWorkItem {
            UIView.animate(withDuration: 1.0) { [weak self] in
                self?.view.backgroundColor = .red
            }
        }
DispatchWorkItem {[weak self] in xxxx }

        self.closureStorage = workItem
        DispatchQueue.main.async(execute: workItem)
    }
```

```swift
// 以下是三种延时释放的情况,假设在执行DispatchQueue.main.asyncAfter(deadline: .now() + 10)后,未到10秒时self将要释放
DispatchQueue.main.asyncAfter(deadline: .now() + 10) {
      print(self)// 此时如果将要释放self,会导致无法立即释放,因为闭包引用了self,而self没有引用闭包,10秒后,先打印self,后走deinit{}释放self
}

DispatchQueue.main.asyncAfter(deadline: .now() + 10) { [weak self] in
      print(self)// 此时如果将要释放self,因为此处弱引用了self,所以self释放.先走deinit{},10秒后打印self(此时为nil)
}

DispatchQueue.main.asyncAfter(deadline: .now() + 10) {
      print("Hello word")// 此时如果将要释放self,因为没有引用self,所以先走deinit{},10秒后打印Hello word
}
```


评论 写评论

lukyy
各种案例, 应有尽有, 大大的👍
3楼 2021.12.30 18:45

72c9a6505ce7
说的好详细
2楼 2020.07.23 15:24 2