스위프트에서 lazy 변수의 스레드-세이프티

https://medium.com/what-i-talk-about-when-i-talk-about-ios-developmen/the-thread-safety-of-lazy-variables-in-swift-b20184ef5a38

Foreword

이 기사는 제 개인적인 관점에 기초하여 작성되었습니다. 때로는 pull requests에서 lazy 변수의 사용을 발견했으며, 대부분 스레드-세이프티 문제가 제대로 처리되지 않았습니다. 이 짧은 기사가 문제에 대한 일반적인 아이디어를 제공하고 자신의 상황에서 lazy 변수를 사용하거나 사용하지 않기로 결정하는 데 도움이 되기를 바랍니다.

Lazy variables (지연 변수)

lazy 변수는, 일반 변수와 달리 선언되고 정의 될 때 초기화되어 메모리에 로드되지 않습니다. 대신, 처음 액세스 할 때 초기화됩니다. 클래스가 있다고 가정하십시오.

```
1  class MyClass {
2    lazy var aLazyInstance = ALazyClass()
3  }
```

MyClass.swift hosted with ❤ by GitHub view raw

A sample class to illustrate lazy variables

인스턴스 **MyClass**가 생성 될 때, **aLazyInstance** 속성이 메모리에 로드되지 않습니다. 그러나 생성된 **MyClass** 인스턴스의 속성에 어딘가에서 액세스하면, 그 순간 속성이 즉시 메모리에 로드됩니다.

lazy 변수를 현명하게 사용하는 것은 프로젝트의 메모리 공간(footprint)을 줄일 수 있습니다. 마찬가지로, 어떤 이유로 인해 많은 양의 데이터를 메모리에 로드하지 않도록 lazy loading를 활용할 수 있습니다. 그러나 일반 변수를 사용하는 것과 달리, lazy 변수를 사용하기 전에 장단점을 철저히 생각하는 것이 좋습니다.

What may go wrong in case of multithreading (멀티스레딩에서 잘못할 수 있는 것)

다음 코드를 살펴 보겠습니다. 스레드 간에 공유되는 **SharedInstanceClass** 인스턴스가 있으며 **testVar** 속성에 액세스 할 수 있습니다 (간단하게 하기 위해, 패키지 액세스 modifier를 사용함).

```swift
class SharedInstanceClass {
    lazy var testVar = {
        return Int.random(in: 0..<99)
    }()
}

let queue = DispatchQueue(label: "test", qos: .default, attributes: [.initiallyInactive,
            .concurrent], autoreleaseFrequency: .workItem, target: nil)
let group = DispatchGroup()


let instance = SharedInstanceClass()
for i in 0 ..< 10 {
    group.enter()
    queue.async(group: group, qos: .default) {
        let id = i
        print("\(id) \(instance.testVar)")
        group.leave()
    }
}
queue.activate()
group.wait()
```

LazyVarRacingCondition.swift hosted with ❤ by GitHub view raw

A simple piece of code to demonstrate race condition across threads

출력을 관찰하면 **testVar**의 값이 변경 될 수 있습니다. 대부분의 경우 개발자가 기대하는 것이 아닙니다. **경쟁 상황(race condition)**이 발생했을 때의 샘플 출력은 다음과 같습니다.

```
3 58

0 5

1 3

2 18

4 18

5 18

6 18

8 18

7 18

9 18
```

보시다시피 **id = 3** 인 스레드는 **testVar**를 58로 설정한 다음, **id = 0** 인 스레드는 **testVar**를 **5**로 설정 한 다음 id = 1 인 스레드는 testVar를 3으로 설정한 다음, testVar의 값은 18로 유지되었습니다.

To lazy or not: My rule of thumb (경험 규칙)

변수를 lazy로 할지 안할지 여부를 결정 해야하는 경우, 내 자신의 경험 규칙이 있습니다. 그렇게하는 자신 만의 규칙이 있다면 좋습니다. 그렇지 않으면, 내 경험을 통해 힌트를 얻을 수 있으므로 나중에 문제에 대한 자신의 통찰력을 얻을 수 있기를 바랍니다.

질문 1 : 실제로 메모리 사용량을 줄입니까?

변수를 게으른 것으로 선언한다고해서 반드시 메모리 공간이 줄어드는 것은 아닙니다. 속성에 빨리 접근하고 피할 수 없다면, lazy loading은 아무 소용이 없습니다.

질문 2 : lazy 변수가 개별 스레드에서 액세스됩니까?

lazy 변수에 액세스하는 장면을 생각하십시오. 그것이 **본질적으로 동시성으로 동작하는 경우**, 예기치 않은 동작이나 **손상된 값을 피하기 위해 일종의 동기화 기능을 사용하여 lazy var를 보호**해야합니다. 또는 이러한 **lazy 변수를 보호하는 데 필요한 자원이 변수보다 훨씬 많은 경우 (예 : boolean 변수를 보호하기 위해 디스패치 큐 사용한다던지) lazy 변수를 만들지 않는 것이 좋습니다.**

동시성이 작동하도록 설계되지 않은 경우, 이러한 lazy 변수를 오용할(misusing) 가능성을 줄이는 인터페이스 설계에 방어적으로 하십시오. 또는 훨씬 좋게, 객체 외부에서 lazy 속성에 액세스할 필요가 없는 경우, 객체 외부로 부터의 부작용을 피하기 위해 private으로 설정하십시오.

Conclusion

lazy loading은 개발자가 메모리 공간을 줄이거나 메모리에 너무 많은 데이터를 로드하는 것을 피하거나 일종의 초기화 또는 설정 문제(예 : self가 초기화 되지 않았을 때, 뷰를 셋팅하여 self에 액세스할 필요가 있을 때)에 도움이 됩니다. 그럼에도 불구하고 lazy loading을 적용하는 것이 항상 쉬운(toll-free) 것은 아닙니다. 때로는 부작용을 피하기 위해 추가 마일을 이동해야 할 수도 있습니다. 마지막으로, lazy 변수를 두려워하지 마십시오. 그것은 여전히 그것의 존재의 목적이 당신을 도울 수 있는 훌륭한 것입니다.

화이팅!