The Dark Sides of Singleton

Demonstrated in Objective-C

- Simon Wang

The Dark Sides of Singleton

- The dark sides of singleton initialisation
- The dark sides of singleton lazy property
- The dark sides of singleton pattern
- When to use singleton?
- Thread Safe alternatives

The dark sides of singleton initialisation

Singleton initialisation

- Full mutex lock
- Double-checked Locking
- Dispatch_once

Instruction Reordering

- Step 1: Allocate memory to hold the object
- Step 2: Construct a object in the allocated memory
- Step 3: Make shared instance point to the allocated memory

Double-checked Locking

- Compiler and CPU instruction reorderings
  - Not initialised at all
  - Not fully initialised
- Generally broken and anti-pattern
- Platform dependent
- Memory barriers are essential to make it correct

Dispatch_once

- Static token
- Atomic compare and swap
- CPU branch prediction
- No memory barriers means super fast

The dark sides of singleton lazy property

The dark sides of singleton lazy property

- Dispatch_once doesn’t help
- Full mutex lock is expensive
- Dispatch_sync is also expensive
- Double-checked Locking with memory barriers

lazy property in Swift

> NOTE
>
> If a property marked with the `lazy` modifier is accessed by multiple threads simultaneously and the property has not yet been initialized, there is no guarantee that the property will be initialized only once.

- NOT THREAD SAFE!!! - (well, expected!)
- SR-1042, don’t overkill a property, just a property!

> Greg Parker added a comment - 23 Mar 2016 2:56 PM - edited
>
> A thread-safe lazy property would require a memory barrier on every read on some architectures. (`dispatch_once` avoids that barrier, but it is unsafe for anything other than global variables.)
>
> A thread-safe lazy property that is too large or does not have some unused sentinel value available would also need additional storage for synchronization (stored next to property value, or in the object header, or in a side table like `@synchronized`).
>
> Both of these are expensive and may not be suitable for every lazy property.

The dark sides of singleton pattern

The dark side of singleton pattern

- Global states, and globals are bad
- Dependency hidings
- Grid locks - quite common!
    - Logic Dead Lock
    - Threads Dead Lock
- Initialisation orders - end up to initialise when app starts
- Hard to test - Unit tests - testable code!

!SOLID

- Single responsibility
- Open to extension, close to modification
- Interface Segregation
- Dependency injection
- Inversion of Control

When to use Singleton?

- When it has a genuine meaningful semantic
  - UIDevice, UIApplication
  - Resource Contention
- Think 3 times and make sure it is what you want.
  - Generally, you don’t really need it
  - If you have to use it, then review your code design and architecture

Thread Safety Alternatives

1. Remove lazy loadings altogether ~
2. Eager initialisation
3. Full mutex lock
4. Per thread cache

Thank You
&
Questions