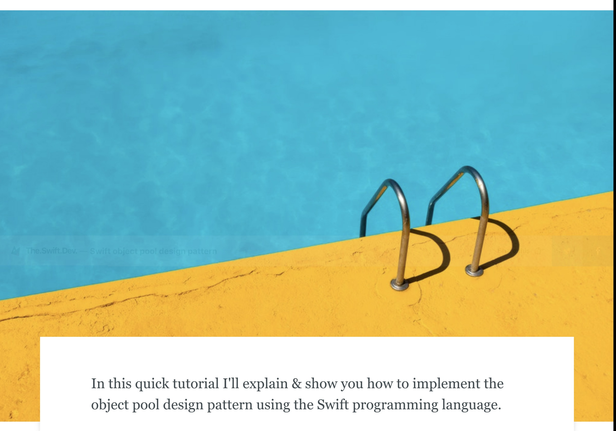
9 DECEMBER 2018 / DESIGN PATTERNS

Swift object pool design pattern



In this quick tutorial I'll explain & show you how to implement the object pool design pattern using the Swift programming language.

A generic object pool in Swift

The object pool pattern is a creational design pattern. The main idea behind it is that first you create a set of objects (a pool), then you acquire & release objects from the pool, instead of constantly creating and releasing them. 👍

Why? Performance improvements. For example the Dispatch framework uses an object pool pattern to give pre-created queues for the developers, because creating a queue (with an associated thread) is an relatively expensive operation.

Another use case of the object pool pattern is workers. For example you have to download hundreds of images from the web, but you'd like to download only 5 simultaneously you can do it with a pool of 5 worker objects. Probably it's going to be a lot cheaper to allocate a small number of workers (that'll actually do the download task), than create a new one for every single image download request. 🏞

What about the downsides of this pattern? There are some. For example if you have workers in your pool, they might contain states or sensitive user data. You have to be very careful with them aka. reset everything. Also if you are running in a multi-threaded environment you have to make your pool **thread-safe**.

Here is a simple generic thread-safe object pool class:

Pool.swift 1.27 KB on GitLab

```swift
import Foundation

class Pool<T> {

    private let lockQueue = DispatchQueue(label: "pool.lock.queue")
    private let semaphore: DispatchSemaphore
    private var items = [T]()

    init(_ items: [T]) {
        self.semaphore = DispatchSemaphore(value: items.count)
        self.items.reserveCapacity(items.count)
        self.items.append(contentsOf: items)
    }

    func acquire() -> T? {
        if self.semaphore.wait(timeout: .distantFuture) == .success, !self.items.isEmpty {
            return self.lockQueue.sync {
                return self.items.remove(at: 0)
            }
        }
        return nil
    }

    func release(_ item: T) {
        self.lockQueue.sync {
            self.items.append(item)
            self.semaphore.signal()
        }
    }
}


let pool = Pool<String>(["a", "b", "c"])

let a = pool.acquire()
print("\(a ?? "n/a") acquired")
let b = pool.acquire()
print("\(b ?? "n/a") acquired")
let c = pool.acquire()
print("\(c ?? "n/a") acquired")

DispatchQueue.global(qos: .default).asyncAfter(deadline: .now() + .seconds(2)) {
    if let item = b {
        pool.release(item)
    }
}

print("No more resource in the pool, blocking thread until...")
let x = pool.acquire()
print("\(x ?? "n/a") acquired again")
```

As you can see the implementation is just a few lines. You have the thread safe array of the generic pool items, a dispatch semaphore that'll block if there are no objects available in the pool, and two methods in order to actually use the object pool.

In the sample you can see that if there are no more objects left in the pool, the current queue will be blocked until a resource is being freed & ready to use. So watch out & don't block the main thread accidentally! 😠

External sources

- Object pool pattern
- A Design Pattern Story in Swift – Chapter 16: Object Pool
- Design patterns: Object pool
- A generic object pool implementation for Swift...

Tibor Bödecs
Read more posts by this author.

Read More

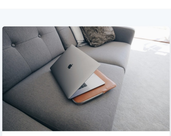
— The.Swift.Dev. —
Design patterns

Lazy initialization in Swift

Swift facade design pattern

Iterator design pattern in Swift

See all 18 posts →



DESIGN PATTERNS
Lazy initialization in Swift

Learn how to use lazy properties in Swift to improve performance, avoid optionals or just to make the init process more clean.

3 MIN READ

UIKIT
UITableView tutorial in Swift

This guide is made for beginners to learn the foundations of the UITableView class programmatically with auto layout in Swift.

7 MIN READ