Mastering Dispatch Queues in Swift: Understanding, Implementation, and Limitations

VINODH KUMAR · Follow

5 min read · Feb 20, 2024

Photo by Abdelrahman Sobhy on Unsplash

Dispatch Queues, introduced as part of Apple's Grand Central Dispatch (GCD) was introduced in 2009, offer an alternative to threads, allowing you to perform work asynchronously. Let's explore their features and differences compared to operation queues and threads.

Basic Usage

Think of a dispatch queue as a line of tasks waiting to be done. You create one like this:

```
let queue = DispatchQueue(label: "my.queue")
```

Now, if you want something to happen in that queue, you simply add a task to it like this:

```
queue.async {
 print("Doing some work here!")
}
```

Types of Dispatch Queues

Serial Queues

By default, when you create a Dispatch Queue without specifying attributes, you get a **serial queue**. Tasks queued in a serial queue execute one after another, much like cars moving in a single lane. For instance:

```
let serialQueue = DispatchQueue(label: "my.serial.queue")
serialQueue.async {
 print("Task 1")
}
serialQueue.async {
 print("Task 2")
}
```

Concurrent Queues

On the other hand, a **concurrent queue** allows tasks to execute simultaneously. This resembles a multi-lane highway where cars (tasks) can travel side by side. You can create a concurrent queue like this:

```
let concurrentQueue = DispatchQueue(label: "my.concurrent.queue", attributes: .c
concurrentQueue.async {
 print("Concurrent Task 1")
}
concurrentQueue.async {
 print("Concurrent Task 2")
}
```

Delayed Execution

Dispatch queues enable delayed execution of tasks. You can schedule a task to start after a certain time:

```
let delayedQueue = DispatchQueue(label: "delayed.queue")
delayedQueue.asyncAfter(deadline: .now() + 1) {
 print("This task starts one second later!")
}
```

Task Priorities

Dispatch Queues also allow you to set task priorities, determining how soon they should be executed concerning other tasks:

```
let highPriorityQueue = DispatchQueue(
label: "high.priority.queue",
qos: .userInitiated
)
let highPriorityTask = DispatchWorkItem {
 print("This is a high-priority task!")
}
highPriorityQueue.async(execute: highPriorityTask)
```

Task Cancellation

Tasks can be canceled if necessary, but it's crucial to check for cancellation within the task similar to thread and operation queue.

```
var task: DispatchWorkItem!
task = DispatchWorkItem {
 guard !task.isCancelled else {
 print("Task was cancelled.")
 return
 }
 // Do some work
}
queue.async(execute: task)
// To cancel after a certain time
DispatchQueue.global().asyncAfter(deadline: .now() + 3) {
 task.cancel()
}
```

Sharing Data

Dispatch Queues offer `DispatchSpecificKey` to associate custom data with a queue, allowing data sharing among tasks executing in the same context:

```
let id = UUID()
let specificKey = DispatchSpecificKey<UUID>()
queue.setSpecific(key: specificKey, value: id)
if let retrievedId = DispatchQueue.getSpecific(key: specificKey) {
 print("Found the ID: \(retrievedId)")
}
```

Absolutely! Let's break down the concept of targeting in Dispatch Queues with code examples to understand how specifics flow between queues.

Understanding Queue Targeting and Specifics

Setting and Accessing Specifics

Consider creating a queue, setting specifics (custom data) on it, and accessing those specifics within a task running on the same queue:

```
let queue1 = DispatchQueue(label: "queue1")
let idKey = DispatchSpecificKey<Int>()
let dateKey = DispatchSpecificKey<Date>()
queue1.setSpecific(key: idKey, value: 42)
queue1.setSpecific(key: dateKey, value: Date())
queue1.async {
 print("queue1", "id", DispatchQueue.getSpecific(key: idKey))
 print("queue1", "date", DispatchQueue.getSpecific(key: dateKey))
}
```

Losing Specifics in New Queues

Creating a new queue within an existing queue doesn't automatically inherit specifics:

```
queue1.async {
 let queue2 = DispatchQueue(label: "queue2")
 queue2.setSpecific(key: idKey, value: 1729)
 queue2.async {
 print("queue2", "id", DispatchQueue.getSpecific(key: idKey))
 print("queue2", "date", DispatchQueue.getSpecific(key: dateKey))
 }
}
```

Targeting Queues for Specifics Inheritance

Targeting resolves the issue of specifics loss in new queues:

```
let queue2 = DispatchQueue(label: "queue2", target: queue1)
queue2.setSpecific(key: idKey, value: 1729)
queue2.async {
 print("queue2", "id", DispatchQueue.getSpecific(key: idKey))
 print("queue2", "date", DispatchQueue.getSpecific(key: dateKey))
}
```

Running Parallel Tasks with Targeted Queues

Suppose we have two independent tasks, database query, and a network request, and we want to run them in parallel:

```
func response(for request: URLRequest, queue: DispatchQueue) -> HTTPURLResponse
 let group = DispatchGroup()
let databaseQueue = DispatchQueue(label: "database-request", target: queue)
 databaseQueue.async(group: group) {
 makeDatabaseQuery()
```

```
 }
let networkQueue = DispatchQueue(label: "network-request", target: queue)
 networkQueue.async(group: group) {
 makeNetworkRequest()
 }
group.wait()
return .init()
}
```

Maintaining Specifics Inheritance

To ensure the new queues inherit specifics, pass the parent queue as an argument:

```
response(for: .init(url: .init(string: "https://www.testurl.com")!), queue: queu
```

Optimizing Queue Creation

Optimize by creating a single concurrent server queue and targeting new queues to inherit its properties:

```
let serverQueue = DispatchQueue(label: "server", attributes: .concurrent)
// For each request
let queue = DispatchQueue(label: "request-\(requestId)", attributes: .concurrent
queue.setSpecific(key: requestIdKey, value: requestId)
queue.async {
 response(for: .init(url: .init(string: "https://www.testurl.com")!))
}
```

Limitations:

Dispatch queues in Swift encompass the strengths of both threads and operation queues, offering asynchronous work, priority management,

cancellation, and specific data storage. However, they still pose certain challenges that need attention.

1. **Passing Queues for Specifics Inheritance**

Currently, to inherit specifics within a new queue, we pass the parent queue explicitly, defeating the purpose of implicit data flow:

```
response(for: .init(url: .init(string: "https://www.testurl.com")!), queue: requ
```

This approach contradicts the aim of seamless data sharing across the execution context without passing it layer by layer.

2. Inheriting Cancellation and Thread Management

Although specifics can be inherited between dispatch queues, cancellation of one work item doesn't propagate to child work items. Threads may still escalate in number if not handled properly, potentially leading to resource issues.

3. Queue Starvation and Intense Operations

Creating numerous queues for a single unit of work or running CPU-intensive tasks on a single queue may lead to thread starvation. Dispatch queues lack tools for cooperation between work items, hindering fair CPU utilization.

4. Lack of Cooperative Concurrent Code

While GCD is robust, it lacks features for writing cooperative concurrent

code. Work items contend for CPU time without effectively allowing others to utilize available resources during downtimes.

traditional locks like NSLock.

Summary

In summary, while GCD provides powerful concurrency tools, addressing issues like implicit data flow, cancellation inheritance, CPU resource management, cooperative concurrent code, and effective data race handling remain areas for improvement. GCD tools assist but don't deeply integrate into the concurrency model, placing responsibility on developers to handle these intricacies effectively.

Next Steps: Ready to master concurrency in Swift? Let's explore Tasks in our next installment: Mastering Concurrency: Task

Series Navigation

- **Part 1**: Exploring Threads in Swift
- **Part 2**: Exploring Swift's Operation Queues: Enhancing Threading with Limitations
- **Part 3**: Mastering Dispatch Queues in Swift: Understanding, Implementation, and Limitations
- **Part 4:** Mastering Concurrency: Task

- **Part 5:** Swift Concurrency: Safeguarding Data with @Sendable and Actors

IOS | IOS App Development | Swift Programming | Swiftui | IOS Development

Written by VINODH KUMAR

Follow

50 Followers

📱 Senior iOS Developer | Swift Enthusiast | Tech Blogger 🖥️

More from VINODH KUMAR

VINODH KUMAR

Swift Concurrency: Safeguarding Data with @Sendable and Actors

VINODH KUMAR

Understanding Memory Allocation in Swift: Stack vs. Heap

Before delving deeper, let's recall the issue of data synchronization and data races that we...

6 min read · Feb 20, 2024

100

Memory management is a fundamental aspect of programming, critical for efficient...

5 min read · Apr 17, 2024

28 2

VINODH KUMAR

Mastering Concurrency: Task

Swift 5.5 brought in powerful concurrency tools that simplify and strengthen our...

6 min read · Feb 20, 2024

105

VINODH KUMAR

Exploring Swift's Operation Queues: Enhancing Threading wit...

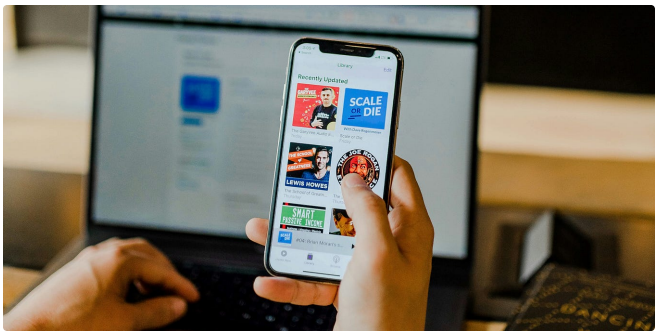
Apple has introduced abstractions over threads to address the issues we discussed...

4 min read · Feb 20, 2024

3

See all from VINODH KUMAR

Recommended from Medium



VINODH KUMAR in Stackademic

Exploring Threads in Swift

Concurrency, or doing multiple things at the same time, is an important part of...

9 min read · Feb 20, 2024

49

Yury Lebedev

Differences between weak and unowned in Swift with examples

Introduction

4 min read · Jan 16, 2024

48

Lists

Apple's Vision Pro

7 stories · 68 saves

Tech & Tools

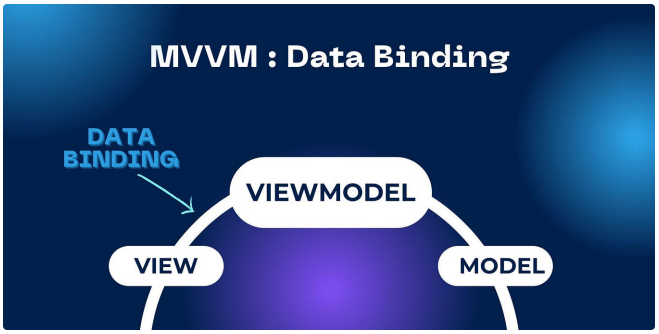
16 stories · 237 saves

Icon Design

36 stories · 310 saves

Productivity

241 stories · 434 saves



Akash Patel

Valentin Jahanmanesh

MVVM Data Binding in iOS (UIKit)

In this article, we'll explore how to implement MVVM data binding in iOS UIKit projects...

3 min read · Apr 17, 2024

68 1

Swift Actors—in depth

Learn, don't memorize.

15 min read · Nov 23, 2023

465 6

Vladyslav Shkodych

Mastering SwiftUI: Are You Really as Good as You Think?

How deep do you think you know what SwiftUI is and how it works? Have you ever wondere...

16 min read · May 10, 2024

413 6

Pulkit Vora

iOS Concurrency Showdown

NSOperationQueue vs Grand Central Dispatch (GCD) (Part 1)

3 min read · Apr 16, 2024

2

See more recommendations