WWDC 2016 Concurrent Programming With GCD in Swfit 3

원문

소개

이번 문서는 Swift 3의 새로운 GCD 기능을 이용해 동시성 프로그래밍을 구현하는 방법과 동기화에 관해 안내합니다.

메인 스레드에서 모든 작업을 수행하게 되면 사용자 이벤트를 막고 애플리케이션의 성능을 저하할 수 있습니다. 이러한 문제를 방지하기 위해 GCD와 동기화를 적절하게 사용하는 것은 매우 중요한 일입니다.

개념

Concurrency(동시성)

하나의 애플리케이션에서 여러 가지 작업을 동시에 수행하는 것을 의미하며, 여러개의 스레드를 생성해 동시성을 구현할 수 있습니다.

GCD

애플 플랫폼의 동시성 라이브러리입니다. GCD를 이용하면 Apple Watch, iOS 기기, Apple TV 그리고 Mac에서 작동 가능한 멀티 스레드 코드를 구현할 수 있습니다.

Dispatch Queue

작업 항목을 담아 대기열(Queue)에 추가하는 구조이며 Swift에서 각 작업 항목은 클로저입니다. 작업 항목이 모두 들어오면 Dispatch Queue가 스레드를 불러와 작업을 실행하고 작업이 완료되면 스레드가 스스로 해제됩니다.

추가로 스레드를 생성할 수 있으며, 생성된 스레스드에서 Run Loop를 실행할 수 있습니다.

Dispatch Queue를 이용해 작업하는 방법에는 아래와 같은 두 가지가 있습니다.

Synchronous Execution(동기 작업)

하나의 Dispatch Queue에 여러 가지 작업항목이 쌓여있을 때 스레드를 호출해 하나하나 작업을 순서대로 실행하는 방법입니다.

Asynchronous Execution(비동기 작업)

Dispatch Queue 외 추가적인 스레드를 생성하여 작업을 추가한 뒤 dispatch queue에 제출하면 dispatch queue의 모든 동기 작업이 수행 완료되기 전에 추가 스레드의 작업을 비동기로 실행한 뒤 나머지 작업을 재개하는 방식입니다.

동시성 프로그래밍

Work Off Main Thread

메인 스레드는 Main Run Loop와 Main Queue를 동시에 지닌 특별한 구조를 지닙니다. 메인 스레드를 방해하지 않으면서 여러 Swift작업을 수행하기 위해서는 적절한 Dispatch Queue를 사용해야 하며 실제 코드를 사용하는 방식은 아래와 같습니다.

```
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)
    // UI 업데이트
    DispatchQueue.main.async {
        imageView.image = smallImage
    }
}
```

Controlling Concurrency

적정 수 이상의 dispatch queue를 사용하게 되면 원치 않는 스레드를 부가적으로 발생시키는 일이 연속적으로 발생해 스레드 폭발(Thread Explosion) 현상이 일어날 수 있으므로 주의해야 합니다. (이와 관련해서는 WWDC 2015 Building Responsive and Efficient Apps with GCD를 참고하십시오.)

Dispatch Group

만일 어떤 하나의 작업이 완료된 뒤에 여러 가지 작업을 한 번에 진행하고 싶다면 아래와 같은 방식으로 dispatch group을 활용하십시오.

- 애플리케이션의 데이터 흐름을 확인하십시오.

- 실행될 순서에 따라 작업을 분류하십시오.
- 나중에 수행할 작업(subsystems)을 dispatch group에 비동기로 담으십시오.
- 먼저 수행될 작업을 완료한 후, group에 알림(notify)을 주면 한 번에 dispatch queue를 실행할 수 있습니다.

Serialize State between Subsystems

여러 가지 작업 간의 연속성을 유지하려면 해당 시스템의 프로퍼티에 `.sync` 를 사용하십시오. 이때 작업 간에 데드락(deadlock)이 발생하지 않도록 작업 순서에 유의하십시오.

```
var count: Int {
    queue.sync { self.connections.count }
}
```

Choosing a QoS(Quality of Service)

Dispatch에 QoS를 적용하면 작업 항목을 개발 의도 및 우선도에 맞게 분류할 수 있으며 아래와 같은 종류가 있습니다. (자세한 내용은 WWDC 2015 Building Responsive and Effecient Apps with GCD를 참고하십시오.)

- User Interactive
- User Initiated
- Utility
- Background

```
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print("Button tapped")
}
```

DispatchWorkItem

- 기본적으로 `.async` 는 작업이 제출되는 시점의 작업을 캡처하는데 `DispatchWorkItem` 의 `.assignCurrentContext` flag를 사용하면 생성 시점을 캡처할 수 있습니다.

```
let item = DispatchWorkItem(flags: .assignCurrnetContext) {
    print("Hello WWDC 2016")
}

queue.async(execute: item)
```

- `DispatchWorkItem` 의 `.wait` 기능을 이용하면 특정 작업이 완료된 이후까지 기다렸다가 다음 작업을 수행시킬 수 있습니다.

동기화 프로그래밍

전역 변수는 atomic으로 초기화되는 반면, 클래스 프로퍼티와 Lazy 프로퍼티는 atomic으로 초기화되지 않습니다. 이러한 프로퍼티의 초기화를 동시에 호출하게 되면 두번 초기화 될 수 있으므로 동기화 작업이 필요합니다. 만일 적절한 시기에 동기화 작업이 이루어지지 않으면 애플리케이션이 종료되거나 사용자의 데이터를 잃을 수 있으므로 주의해야 합니다. 이번 문서에서 Swift에서 동기화를 어떻게 구현할 수 있는지 알아봅니다.

Traditional C Lock in Swift

동기화에 전통적으로 사용되는 구조체 기반의 C lock 타입 `pthread_mutex_t` 를 Swift에서 사용하는 것은 추천하지 않습니다. 대신 클래스 기반의 `Foundation.Lock` 을 이용하는 것이 안전합니다. 하지만 `Foundation.Lock` 은 전통적인 C lock과 다를 수 있으므로 비슷한 형태로 사용하고 싶다면 Objective-C를 사용해야 합니다.

Use GCD for Synchronization

- 동기화를 위해 Dispatch Queue를 이용하면 기존의 lock 방식을 이용하는 것보다 오용의 위험성이 적을 수 있습니다. 왜냐하면, dispatch queue를 이용하면 lock을 해제하는 것을 잊어버리는 일이 없기 때문입니다. 그리고 Xcode를 이용해 디버깅하기 용이합니다.

```
class MyObject {
    private let internalState: Int
    private let internalQueue: DispatchQueue
    var state: Int {
        get {
            return internalQueue.sync { internalState }
        }
        set (newState) {
            internalQueue.sync ( internalState = newState }
        }
    }
}
```

- Dispatch Queue를 활용하면 전제조건(precondition)을 설정하여 동기화가 예상하는 방식으로 이루어질 수 있도록 수행 여부를 설정 할 수 있습니다.

```
dispatchPrecondition(.onQueue(expectedQueue))
dispatchPrecondition(.notOnQueue(unexpectedQueue))
```

Object Lifecycle in a Concurrent World

동시성 시스템 내에서 객체의 생명주기는 아래와 같은 흐름을 지니게 됩니다.

- 단일 스레드 생성 `setup`
- 동시성 상태 `activate`
- 동시성 상태 `invalidate`
- 단일 스레드 메모리 해제 `deallocation`

```
class BusyController: SubsystemObserving {
    private var invalidated: Bool = false

    init(...) { ,,, }

    // Activate
    func activate() {
        DataTransform.sharedInstance.register(observer: self, queue: DispatchQueu.main)
    }

    // Invalidate
    func invalidateion() {
        dispatchPrecondition(.onQueue(DispatchQueue.main))
        invalidated = true
        DataTransform.sharedInstance.unregister(obsever: self)
    }

    // Dealoocation
    deinit {
        precondition(invalidated)
    }
}
```

GCD Object Lifecycle

- `setup` : 속성 및 타겟 queue 지정, 소스 핸들러 설정

  ```
  let q = DispatchQueue(label: "com.example.queue", attributes: [.autoreleaseWorkItem])
  let source = DispatchSource.read(fileDescriptor: fd, queue: q)

  source.setEventHandler { ... }
  source.setCancelHandeler { close(fd) }
  ```

- `activate` : 추가 queue 생성 가능 *
  `swift extension DispatchObject { func activate() } let queue = DispatchQueue(label: "com.example.queue", attributes: [.initialyInacti`

- `invalidate` : 이벤트 감지 종료, 모든 핸들러 해제

  ```
  extension DispatchObject {
      func cancel()
  }
  source.setCancelHandler { close(fd) }
  ```

- `deallocation` : GCD 객체의 활성여부 확인 및 해제

Summary

- 애플리케이션 내부 데이터 흐름에 따라 독립적인 시스템으로 분리하여 관리하십시오.
- Dispatch Queue를 이용해 동기화 작업을 수행하십시오.
- 동시성 및 동기화된 객체를 사용할 때는 Activate/invalidate 패턴을 활용하십시오.