BoostCamp iOS

MOGAY

Contents

Grand Central Dispatch (GCD) / Dispatch Queue

Grand Central Dispatch (GCD)

GCD는 Apple에서 개발한 멀티코어 프로세서를 위한 Thread 프로그래밍 방법이다. 기존의 쓰레드 관리 기법은 개발자가 직접 락을 걸고, 쓰레드풀을 관리하는 등의 수고가 필요했지만 GCD에서는 Thread를 OS가 자동 관리 및 분배를 해준다. Mac OS 10.6, iOS 4.0 이상 지원한다.

Dispatch Queue

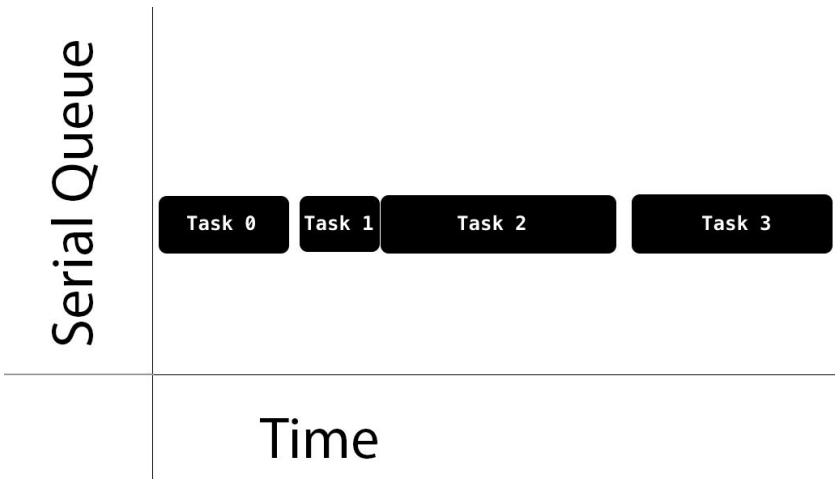
Dispatch Queue는 Task를 적재하는 데이터 구조다. 데이터 구조의 Queue이므로 작동 방식이 Serial(순차적)이든 Concurrent(동시)이든, 언제나 FIFO(First In First Out)방식으로 동작한다.

GCD Queue

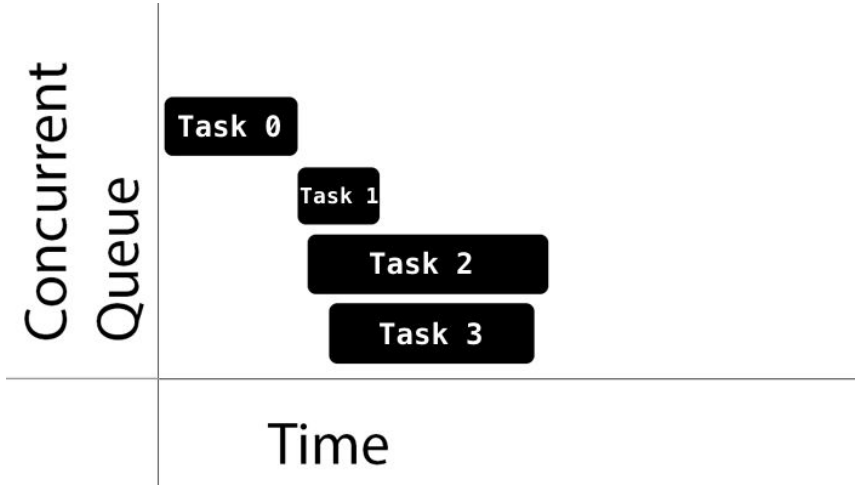
GCD 에서 큐 종류는 크게 2가지 타입이 있다.

- Serial Queue
  - 한번에 하나의 작업이 순차적으로 실행됨
  - Swift 3 에서는 기본적으로 serial queue가 생성됨
  - DispatchQueue(label: “label”)
- Concurrent Queue
  - queue 의 작업은 동시적으로 실행됨
  - DispatchQueue(label: “label”, attributes: .concurrent)

Serial Queue / Concurrent Queue



Serial Queue



Concurrent Queue

Dispatch Queue

GCD가 제공하는 3가지 타입의 큐

- Main queue
  - Main thread에서 실행되는 serial queue
  - 모든 UI 작업은 Main Queue에서 수행해야 합니다.
- Global queues
  - 전체 시스템에서 공유되는 concurrent queue이며 우선순위(QoS)를 설정할 수 있다.
  - QoS Class에 대한 설명 -> 다음 슬라이드
- Custom queues
  - 개발자가 임의로 정의한 queue. serial queue나 concurrent queue 둘 다 가능.
  - custom queue는 global queue에서 수행된다.

Quality of Service (QoS) class

Primary QoS classes (우선순위 높은 순)

- User-interactive
- User-initiated
- Utility
- Background

Special QoS classes

- Default
- Unspecified

Quality of Service (QoS) class

Global Queue는 Concurrent Queue로써 작업을 병렬적으로 동시에 처리하기 때문에 작업 완료의 순서는 정할 수 없지만 QoS class를 활용하여 일을 우선적으로 처리할 수 있다.

```
DispatchQueue.global(qos: .userInteractive).async {
    for i in 1...30 {
        print("🅰️\(i)")
    }
    print("🅰️END")
}

DispatchQueue.global(qos: .userInitiated).async {
    for i in 31...60 {
        print("🅿️\(i)")
    }
    print("🅿️🅿️END")
}

DispatchQueue.global(qos: .utility).async {
    for i in 61...90 {
        print("✅\(i)")
    }
    print("✅✅END")
}

DispatchQueue.global(qos: .background).async {
    for i in 91...120 {
        print("0️⃣\(i)")
    }
    print("0️⃣0️⃣END")
}
```

Background

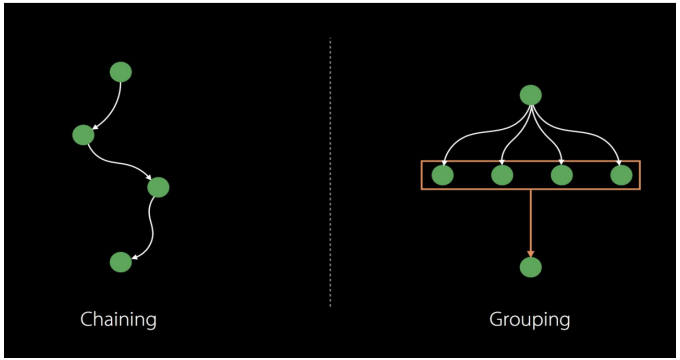
UIKit의 대부분 API들은 메인 스레드 에서 실행되기 때문에 시간이 많이 걸리는 작업들은 백그라운드로 수행하는 것이 좋다.
아래의 예는 시간이 많이 걸리는 작업을 수행하고 그 결과값을 UI 작업에서 활용하는 코드이다.

```
let backgroundQueue = DispatchQueue(label : “mogay”, attributes : .concurrent) // 큐 생성
let mainQueue = DispatchQueue.main
backgroundQueue.async {
     let result = doExpensiveWork()
     mainQueue.async {
          result를 이용해서 UI 작업
     }
}
```

Group - notify

다수의 블럭으로 작업을 수행하고, 전체 작업의 완료에 대한 알림이 필요할 경우

```
let group = DispatchGroup()
let queueOne = DispatchQueue(label: “taskOne”)
let queueTwo = DispatchQueue(label: “taskTwo”)
let queueThree = DispatchQueue(label: “taskThree”)
queueOne.async(group: group) { //background 작업 1}
queueTwo.async(group: group) { //background 작업 2}
queueThree.async(group: group) { //background 작업 3}
group.notify(queue: mainQueue) {
     //main Thread 작업
}
```



main Thread에서 다수의 background의 작업이 다 끝난 통보를 받을 수 있다.

DispatchSemaphore

세마포어를 활용하면 비동기 호출을 동기 호출처럼 만들수 있다.

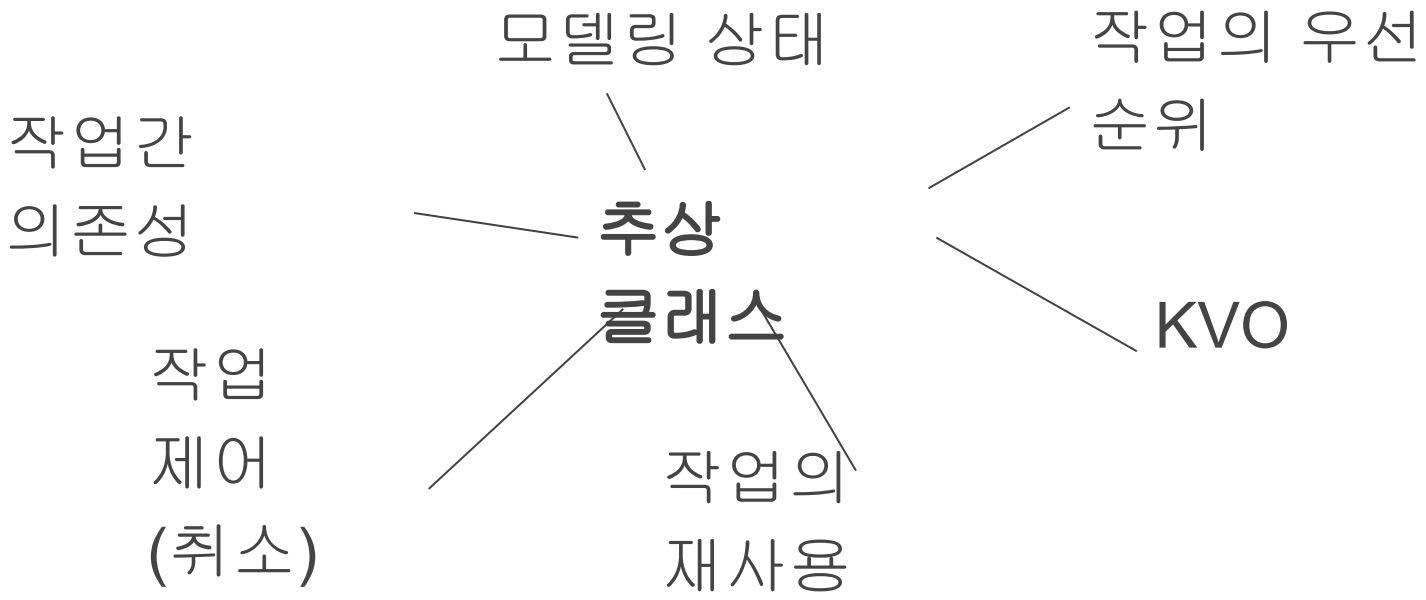
```
let semaphore = DispatchSemaphore(value: 0) // semaphore 값 0
doSomeWorkAsync(completion: {
    semaphore.signal() // doSomeWorkAsync를 실행한 후 semaphore 값을 증가
})
semaphore.wait()
// semaphore가 음수였기 때문에 blocking 상태 였는데,
   semaphore 값이 증가하여 다음 코드를 수행 할 수 있다.
```

DispatchSemaphore

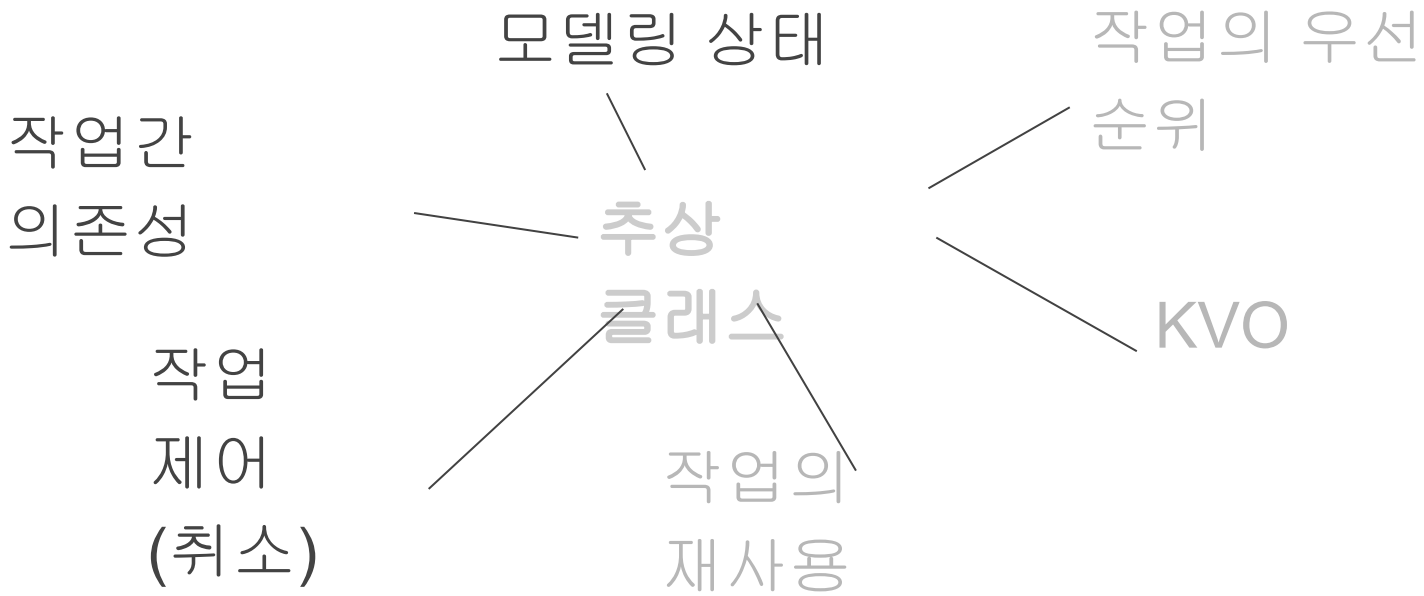
```
class LimitedWorker {
      private let concurrentQueue = DispatchQueue(label: "mogay", attriubutes: .concurrent)
      private let semaphore: DispatchSemaphore
      init(limit: Int) {
            semaphore = DispatchSemaphore(limit)
      }
      func enqueueWork(work: @escaping () -> ()) {
            concurrentQueue.async {
                  semaphore.wait()
                  work()
                  semaphore.signal()
                  }
            }
      }
}
```

특정 리소스에 일정한 수의 연결만을 허용하기 원할 때 semaphore을 활용 할 수 있다.

OPERATION

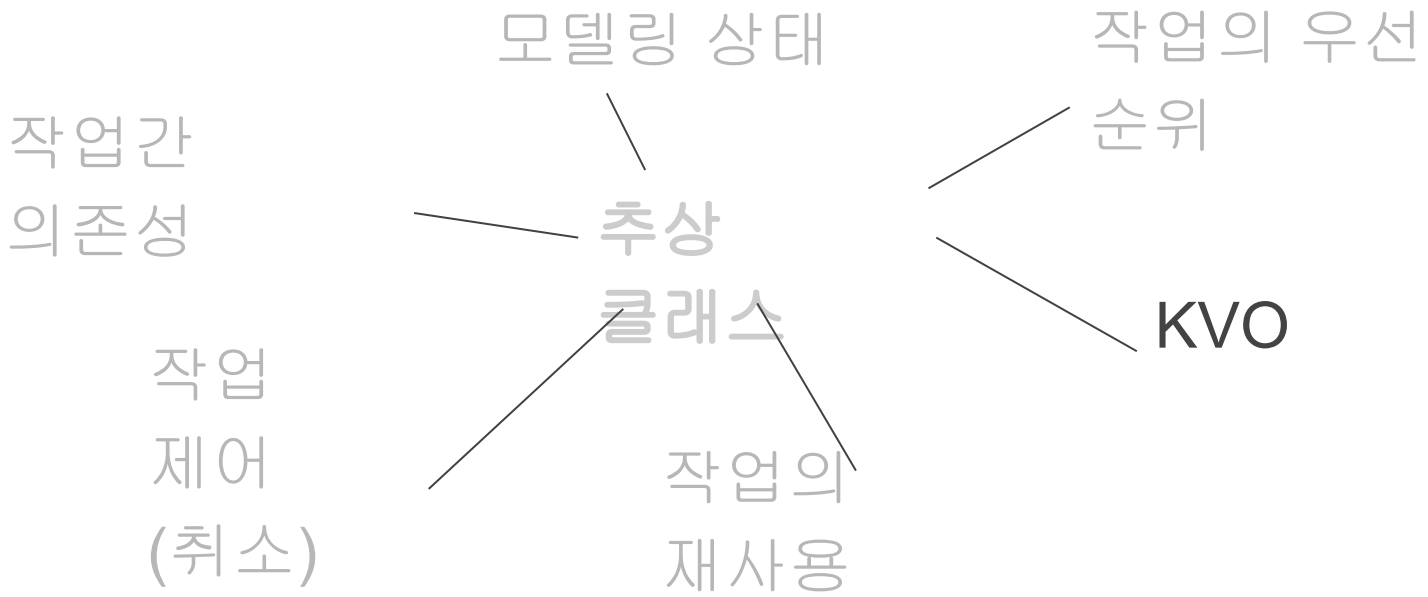


OPERATION



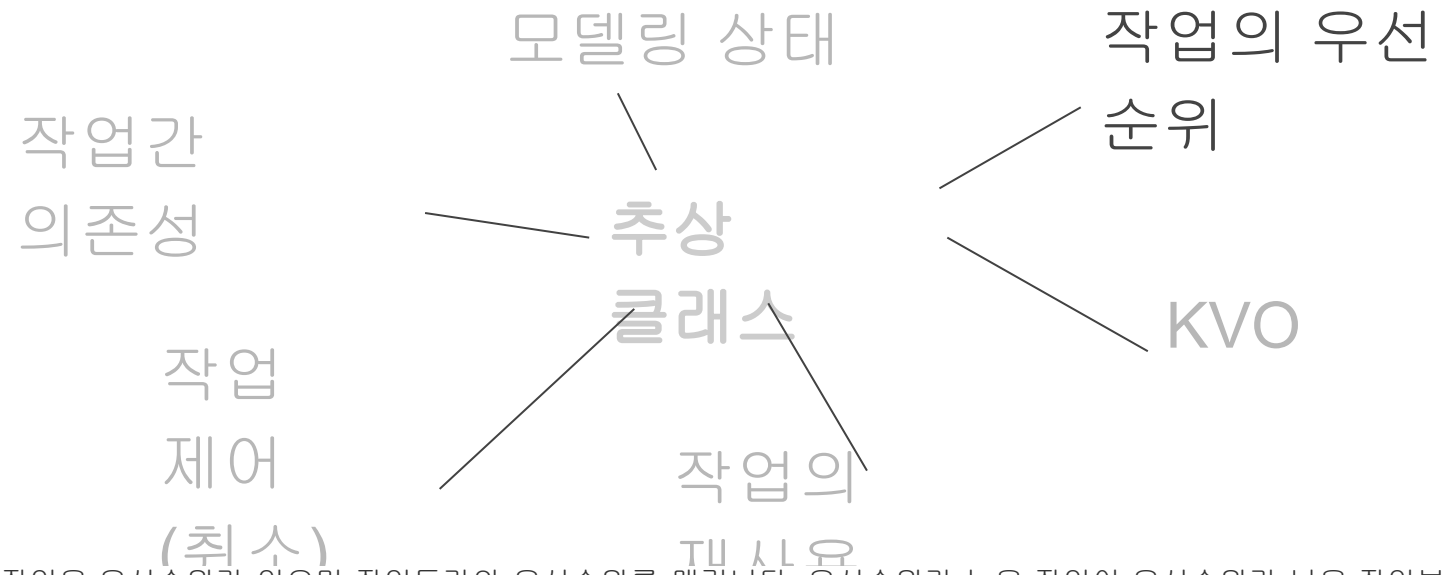
의존성 있는 작업이 실패하였을때, 혹은 유저에 의해 명시적으로 취소가 될 때 작업을 취소하는것은 유용하게 사용됩니다.

OPERATION



NSOperation은 isCancelled, isFinished등 작업의 상태가 변경되었는지를 알 수 있으며 좀 더 세세한 작업을 할 수 있습니다.

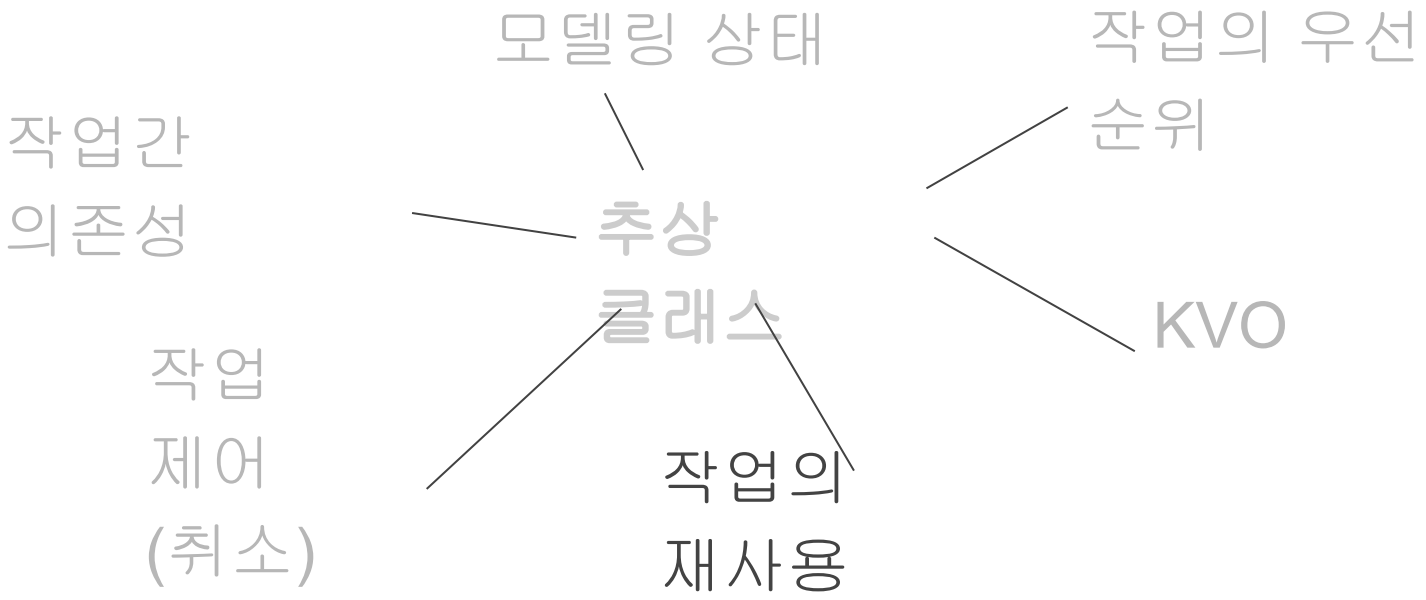
OPERATION



각 작업은 우선순위가 있으며 작업들간의 우선순위를 매깁니다. 우선순위가 높은 작업이 우선순위가 낮은 작업보다 먼저 수행이 됩니다.

cf) GCD도 우선순위를 가지지만 같은 작업에 대해서는 직접적인 방법은 없으며, 개별 블럭이 아닌 전체 큐에 대한 우선순위를 설정합니다.

OPERATION



NSOperation의 자식 클래스를 만들어서 원하는 형태로 작업이 가능하며 작업이 끝나더라도 재사용할 수 있습니다.

OPERATION KVO Compliant Properties

```
var queuePriority: Operation.QueuePriority      KVO(readable & writable)

var completionBlock: (() -> Void)?              KVO(readable & writable)

var isCancelled: Bool                           KVO(read-only)
var isExecuting: Bool                           KVO(read-only)
var isFinished: Bool                            KVO(read-only)
var isConcurrent: Bool
var isAsynchronous: Bool                        KVO(read-only)
var isReady: Bool                               KVO(read-only)
var name: String?

var dependencies: [Operation]                   KVO(read-only)
```

```
Operation.QueuePriority
case veryLow
case low
case normal
case high
case veryHigh
```

OPERATIONQUEUE KVO Compliant Properties

OperationQueue: Operation을 관리하는 대기큐, 우선순위를 가지는 FIFO로 관리.

var operations: [Operation]	KVO(read-only)
var operationCount: Int	KVO(read-only)
var maxConcurrentOperationCount: Int	KVO(readable & writable)
var qualityOfService: QualityOfService	
var isSuspended: Bool	KVO(readable & writable)
var underlyingQueue: DispatchQueue?	
var name: String?	KVO(readable & writable)

OPERATION Dependencies

- 종속성은 특정 순서로 작업을 실행하는 편리한 방법이다.
- addDependency(_:), removeDependency(_:)메소드를 사용하여 종속성 추가 및 제거 가능
- 종속성이 있는 Operation Object는 모든 종속 Operation Object의 실행이 완료될 때까지 준비 상태로 간주되지 않는다.

> ! But, 마지막 종속 Operation이 끝나면, 준비 상태가 되어 실행될수 있음.

Sync. vs Async.

>> Synchronous (by default)

■ 코드에서 start() 메서드를 호출하면 작업이 현재 스레드에서 즉시 실행됨

■ start()메서드가 호출자에게 제어를 반환 할 때까지 작업 완료

■ 작업을 실행하기 위해 항상 Operation을 사용할 때 동기식이 더 간단하다.

>> Asynchronous (by manually)

■ 별도의 스레드에서 작업 예약, 연산 시 '새로운 스레드 시작', '비동기 메서드 호출', '디스패치 큐에 블록 제출' 등을 통해 수행한다.

■ 비동기 Operation의 start()를 호출하면, 해당 메서드가 완료되기 전에 해당 메서드가 반환될 수 있다.

■ 비동기는 작업의 진행 상태를 모니터링하고 KVO 알림을 사용하여 상태의 변경 사항을 보고해야 하기 때문에 더 많은 작업이 필요하다

■ 수동으로 실행 된 작업이 호출 스레드를 차단하지 않도록 하려면 비동기 작업을 정의하는 것이 유용하다.

실행 우선순위 설정

>> qualityOfService

■ CPU 시간, 네트워크 리소스, 디스크 리소스 등과 같은 시스템 리소스에 대한 액세스 권한이 부여되는 우선 순위에 영향을 준다.

■ 높은 품질의 서비스 클래스로 작업하면 리소스 사용 시 더많은 리소스가 수신된다.

■ 반환 값으로는 userInteractive, userInitiated, utility, background, default가 있다.

>> queuePriority

■ opration 큐에서의 실행 우선 순위를 결정한다.

■ 우선 순위 값은 다른 Operation 객체 간에 종속성 관리를 구현하는 데 사용 되어서는 안된다.
(addDependency(_:)사용)

■ 반환 값으로는 veryLow, low, normal(dafault), high, veryHigh 이 있다.

■ 정의 된 반환 값 이외의 우선 순위 값을 지정하려고 하면 normal 을 기준으로 조정된다. ✱ 예시: -10은 very low로 값을 조정

OperationQueue vs GCD

Operation Queue		GCD
High-level		Low-level
Objective-C		C-based
iOS 2.0+		iOS 4.0+
easy to cancel		hard to cancel
reuse (child class)		x

OperationQueue vs GCD

Operation Queue		GCD
Block Code -> Object		Block Code
(ready -> execute -> finish)		cannot use state
dependency		x
completion		x
priority (같은 작업도 구분 가능)		priority (전체 큐에 대한)

OperationQueue vs GCD example

Operation Queue	GCD
complex task	not much complex task
login sequence	optimum CPU performance
image download & resize (use dependency)	

Reference

– – –

- http://outofbedlam.github.io/swift/2016/05/11/GCD/
- http://bartysways.net/?p=514
- https://brunch.co.kr/@tilltue/29

감사합니다