Desarrollo de Apps para iOS

Concurrencia y Usabilidad

IWEB 2019-2020
Santiago Pavón

ver: 2019.12.02

Objetivos

- La interface de usuario:
  - que no se quede bloqueada mientras estamos realizando un cálculo muy largo, descargando recursos de la red, ...
  - que siempre responda ágilmente a las acciones del usuario.
- Soluciones:
  - Timers: En cálculos largos se puede partir una tarea en varias partes y usar Timers para ejecutar las partes. No es concurrencia de verdad. Solo para tareas muy sencillas.
  - Threads: Muy difícil de programar correcta y eficientemente.
  - Operations: Abstracción de alto nivel que crea operaciones y usa un sistema de colas para organizar las tareas.
  - Grand Central Dispatch (GCD): framework de alto nivel para manejar la ejecución asíncrona y concurrente de tareas. Usa un sistema de colas para organizar las tareas

¿Qué Elijo?

- Operation y Operation Queue están construidas sobre GCD.
  - Representan un nivel de abstracción más alto, lo cual es recomendable,
  - pero su uso introduce algo de sobrecarga respecto de usar GCD,
  - pero proporcionan algunas facilidades que GCD no tiene (hay que programárselas):
    - Definir dependencias entre tareas, cancelar o suspender tareas, etc.
  - . . .
- Hay que elegir teniendo en cuenta estos puntos.

UIKit y Main Thread

- Todas las aplicaciones tienen un thread principal (main thread) donde se ejecuta el main run loop.
  - Este thread procesa los eventos, ejecuta las acciones de nuestros controles (target-action), actualiza el interface de usuario.
  - Evitar ralentizarlo por ejecutar en él tareas muy largas.
    - Trocear estas tareas, usar concurrencia.
- Cuidado con la concurrencia:
  - La mayor parte del **UIKit no es thread-safe**.
    - por motivos de eficiencia.
  - Los accesos a la interface de usuario deben hacerse solo desde el main thread.

Grand Central Dispatch

GCD

- GCD es un framework para manejar la ejecución asíncrona y concurrente de tareas.
  - Nos oculta los detalles de la multiprogramación.
  - No tenemos que preocuparnos de los recursos disponibles. GCD se encarga de gestionarlos.
  - Se programa muy fácilmente.
- GCD nos proporciona colas a las que enviaremos tareas para que se ejecuten en orden.
  - Una tarea es un objeto closure.
  - En algún momento GCD asignará un thread libre a la siguiente tarea de la cola para que se ejecute.
    - GCD decide cuántos threads crea, cuándo un thread ejecutará una tarea de la cola, y durante cuánto tiempo.
  - El encolado es **thread-safe**: Se puede acceder a las colas desde distintos threads de forma segura.

DispatchQueue

- Las colas pueden ser serie o concurrentes:
  - **Cola concurrente**:
    - El comienzo de la ejecución de las tareas es FIFO.
      - Las tareas empiezan a ejecutarse en el mismo orden en que se metieron en la cola.
    - Pero una vez que empieza la ejecución de una tarea, se puede comenzar con la ejecución de la siguiente tarea.
      - Es decir, las tareas se ejecutan concurrentemente.
    - Y pueden acabar en cualquier orden.
  - **Cola serie**:
    - Las tareas se ejecutan de una en una en modo FIFO, en el mismo orden en que se encolaron.
    - Hasta que una tarea no termina, no se empieza a ejecutar la siguiente.
- Hay varios tipos de colas disponibles:
  - **Cola main**: cola del sistema para ejecutar tareas en serie en el main thread.
  - **Colas globales**: colas del sistema para ejecutar tareas concurrentemente con distintas prioridades o calidad de servicio.
  - **Colas personales**: podemos crear nuestras propias colas serie o concurrentes en cualquier momento.

DispatchQueue.**main**

- Esta la cola a usar para ejecutar tareas en **serie** en el **Main Thread**.
  - Esta cola la crea el sistema automáticamente.
- Esta cola se obtiene accediendo a la propiedad de clase **`main`** de **`DispatchQueue`**:

```
class var main: DispatchQueue {get}
```

DispatchQueue.**global**

- GCD tiene creadas varias colas globales que podemos usar en nuestra app para ejecutar tareas **concurrentemente**.
  - El comienzo de la ejecución de las tareas es FIFO, pero una vez que empieza la ejecución de una tarea, ya puede comenzar con la ejecución de la siguiente tarea, pudiendo acabar en cualquier orden.
- GCD crea **varias** colas con distintas calidades de servicio (prioridad).
- Estas colas se obtienen con el método de clase `global` de `DispatchQueue`:

```
class func global(qos: DispatchQoS.QoSClass = default) -> DispatchQueue
```

  - Valores para **qos** (indican la calidad o prioridad de las tareas):

```
.userInteractive // Tareas muy breves que tienen que
                 // hacerse con suma prioridad.
.userInitiated   // Tareas mas largas que también
                 // tienen prioridad alta.
.utility         // Tareas largas que no tienen prioridad máxima.
.background      // Tareas para cosas que no necesito
                 // ahora mismo y tienen menos prioridad.
.default         // Calidad de servicio por defecto.
.unspecified     // Sin calidad de servicio.
```

DispatchQueue: Colas Personalizadas

- Pueden crearse colas personalizadas nuevas en cualquier momento.
  - Se crean usando alguno de los constructores existentes:

```
init(label: String,
     qos: DispatchQoS = default,
     attributes: DispatchQueue.Attributes = default,
     autoreleaseFrequency: DispatchQueue.AutoreleaseFrequency =
                                                        default,
     target: DispatchQueue? = default)
```

- Parámetros:
  - **label**: es un String que identifica la cola para ayudar en la depuración.
  - **qos**: es la calidad del servicio.
  - **attributes**: conjunto de valores

```
initiallyInactive // No ejecutar tareas hasta que se active la cola.
concurrent        // Para crear una cola concurrente.
```

  - **autorelaeaseFrecuency**: indicar cuando se hace el autorelease..
  - **target**: las tareas de las colas personalizadas se envían a una cola global cuando tienen que ejecutarse. Podemos indicar a que cola global queremos que se envíen.

- Por defecto las colas creadas son serie, excepto si se indica **concurrent** en los atributos.
  - Las colas serie ejecutan sus tareas de una en una y en orden FIFO.
    - Si una operación se bloquea, sólo se bloquea su cola. Las demás colas continúan ejecutando sus tareas.
    - Si la cola creada es concurrente, los bloques se desencolan en orden FIFO, y se ejecutan concurrentemente (si hay recursos disponibles para ello). Pueden terminar en cualquier orden.
- Usos:
  - Evitar que el main thread se bloquee.
  - Realizar tareas largas en otro thread.
  - Proteger zonas críticas.
  - …

```swift
func demo(serie: Bool) {
    let attrs = serie ? [] : DispatchQueue.Attributes.concurrent
    let q = DispatchQueue(label: "demo",  attributes: attrs)

    for i in 1...3 {
       q.async {
            print("Empieza la tarea \(i)")
            Thread.sleep(forTimeInterval: 0.5)
            print("Termina la tarea \(i)")
        }
    }
}

demo(serie: true)
// Empieza la tares 1
// Termina la tarea 1
// Empieza la tarea 2
// Termina la tarea 2
// Empieza la tarea 3
// Termina la tarea 3

demo(serie: false)
// Empieza la tarea 2
// Empieza la tarea 3
// Empieza la tarea 1
// Termina la tarea 3
// Termina la tarea 1
// Termina la tarea 2
```

Ejecutar demo creando una cola serie.
Las tareas se ejecutan en serie.

Ejecutar demo creando una cola concurrente.
Las tareas se ejecutan en entrelazadas.

DispatchQueue: Encolar Tareas

- Existen muchos métodos para enviar una tarea a una cola:

```
func async(group: DispatchGroup? = default,
           qos: DispatchQoS = default,
           flags: DispatchWorkItemFlags = default,
           execute work: @escaping () -> Void)

func async(execute workItem: DispatchWorkItem)

func sync(execute: () -> Void)

func sync<T>(flags: DispatchWorkItemFlags,
             execute work: () throws -> T) rethrows -> T
```

  - donde:
    - **async** no es bloqueante.
    - **sync** es bloqueante.
      - Se espera hasta que el closure ha terminado, y devuelve su valor.
    - DispatchWorkItem encapsula un trabajo a ejecutar (un closure).
    - DispatchGroup se usa para agregar trabajos.

DispatchWorkItem

- Este tipo encapsula una tarea a ejecutar.
- Se crea pasando un closure en el constructor.

```
init(qos: DispatchQoS = default,
     flags: DispatchWorkItemFlags = default,
     block: @escaping () -> ())
```

- Existen métodos para esperar hasta un DispatchWorkItem ha terminado su ejecución (**wait**), notificar cuando ha terminado (**notify**), cancelarlo (**cancel**), …

Ejemplo: Bajarse una Foto

```
let imgUrl = "http://www.etsit.upm.es/fileadmin/user_upload/banner_portada_escuela.jpg"

// Construir un URL
if let url = URL(string: imgUrl) {
    // Envio la tarea a un thread
    let queue = DispatchQueue(label: "Download Queue")
    queue.async {
        // Bajar los datos del sitio Web
        if let data = try? Data(contentsOf: url),
            // Construir una imagen con los datos bajados
            let img = UIImage(data: data) {
            // El GUI se actualiza en el Main Thread
            DispatchQueue.main.async {
                self.imageView.image = img
            }
        }
}
```

Uso GCD y envío la tarea a un thread

El GUI solo se actualiza en el Main Thread

Ejemplo: Zona Crítica

- Podemos utilizar colas serie para proteger el acceso a zonas críticas:
  - Hay que hacer que la única forma de acceder a la zona crítica sea enviando una tarea (closure) a través de una cola serie.
  - Como esas tareas se ejecutarán de una en una, se elimina la posibilidad de que varias tareas se ejecuten concurrentemente sobre la zona crítica.

ESTO ES INCORRECTO

El uso de los métodos **meter** y **sacar** no es **Thread Safe**.

```
// Zona critica
private var total = 0
```

No podemos cambiar total desde diferentes threads.

```
func meter(_ n: Int) {

    total += n;

    totalLabel.text = "\(total)"
}

func sacar(_ n: Int) {

    total -= n;

    totalLabel.text = "\(total)"
}
```

Zona crítica

Solo se puede actualizar el GUI desde el Main Thread

Zona crítica

Solo se puede actualizar el GUI desde el Main Thread

```
// Zona critica
private var total = 0

// Cola SERIE de acceso a la zona critica
private var queue = DispatchQueue("Cola de acceso")

func meter(_ n: Int) {
    queue.async {

        self.total += n;

        DispatchQueue.main.async {
            self.totalLabel.text = "\(self.total)"
        }
    }
}

func sacar(_ n: Int) {
    queue.async {

        self.total -= n;

        DispatchQueue.main.async {
            self.totalLabel.text = "\(self.total)"
        }
    }
}
```

Zona crítica

Crear cola serie para serializar tareas

Así es Thread Safe

Zona crítica

Actualizo el GUI en el main thread

Así es Thread Safe

Zona crítica

Actualizo el GUI en el main thread

Retrasar una Tarea

- Esperar hasta el momento especificado, y entonces enviar asíncronamente una tarea a la cola especificada.

```
let queue = DispatchQueue.global()

// Enviar dentro de 5 segundos
let t = DispatchTime.now() + 5
queue.asyncAfter(deadline: t) {
    // Las sentencias a ejecutar
}
```

- Existen varios métodos:

```
func asyncAfter(deadline: DispatchTime,
                execute: DispatchWorkItem)
func asyncAfter(deadline: DispatchTime,
                qos: DispatchQoS,
                flags: DispatchWorkItemFlags,
                execute: @escaping () -> Void)
func asyncAfter(wallDeadline: DispatchWallTime,
                execute: DispatchWorkItem)
func asyncAfter(wallDeadline: DispatchWallTime,
                qos: DispatchQoS,
                flags: DispatchWorkItemFlags,
                execute: @escaping () -> Void)
```

Ejecutar algo solo una vez

- Casos típicos la creación de singletons, la configuración inicial de algún elemento.
- Pueden usarse variables globales y propiedades estáticas para realizar ejecuciones que solo pueden ocurrir una vez.
  - Se inicializan de forma perezosa la primera vez que se usan desde cualquier thread.

```
let a = UnaClase()    // Inicializacion perezosa.
var b: UnaClase = { let r = UnaClase()
                    r.unaPropiedad = "valor"
                    return r
                  }() // Lo mismo ejecutando un closure.
final class Singleton {
    static let shared = Singleton() // Inicializacion perezosa.
    private init() {}  // Inicializador privado.
}
```

DispatchWorkItemFlags

- Pueden configurarse el comportamiento de una tarea usando las opciones proporcionadas por este tipo.
- Ejemplo:
  - Crear una barrera con la opción DispatchWorkItemFlags.**barrier**.
    - Una barrera es un punto de sincronización en una cola concurrente.
    - Una cola no ejecuta una tarea añadida con esta opción hasta que no han terminado todas las tareas añadidas anteriormente.
    - Cuando las tareas añadidas anteriormente han terminado, se ejecuta la tarea barrera.
    - Cuando termina la tarea barrera, la cola sigue funcionando de forma concurrente.

```
let q = DispatchQueue(label: "cola",  attributes: .concurrent)

q.async { self.producir() }
q.async { self.producir() }
q.async { self.producir() }

q.async(flags: .barrier) { self.barrera() }

q.async { self.consumir() }
q.async { self.consumir() }
q.async { self.consumir() }


func producir() { print("P") }

func barrera() { print("B") }

func consumir() { print("C") }


// PPPBCCC
```

La barrera

API más …

- `DispatchTime`
- `DispatchGroup`
- `DispatchSemaphore`
- `DispatchSource`
- `DispatchData`
- `DispatchIO`
- `DispatchSpecificKey`
- etc. . .