Los actors en Swift

Por **Anton Zuev** - 18 abril, 2024

Introducción

Con la versión de Swift 5.5 se han introducido un nuevo tipo de objetos que es un *Actor*. Los actors en Swift son un tipo de objetos de referencia (*reference type*) que

forma parte del modelo de concurrencia avanzado. Su función principal es evitar carreras de datos (*data race*) y garantizar el acceso seguro al estado mutable compartido en entornos de programación concurrentes. Principalmente es una clase que restringe acceso a sus propiedades sin dejar de formar carreras de datos.

Los actors en Swift son entidades que encapsulan el estado y el comportamiento relacionado, garantizando que solo un hilo de ejecución pueda acceder a ese estado a la vez. Además proporcionan exclusión mutua implícita, lo que significa que no necesitas preocuparte por problemas comunes de concurrencia como las condiciones de carrera y los bloqueos.

Para definir un actor en Swift, necesitas utilizar la palabra clave *actor*, seguida del nombre del actor y sus métodos y propiedades. Aquí hay un ejemplo simple de cómo definir un actor en Swift:

```
actor CounterActor {
  private var count = 0
  func getCount() -> Int {
    return count
  }
  func increment() { count += 1 }
}
```

En este ejemplo, hemos creado un actor llamado *CounterActor* que encapsula una variable privada y dos métodos: *increment()* para aumentar el valor y *getCount()* para obtener el valor actual.

Los actores tienen los siguientes características:

1. Se crean utilizando la palabra clave *actor*. Este es un tipo de objetos en Swift, como estructuras, clases y enumeraciones.
2. Al igual que las clases, los actores son tipos de referencia. Esto los hace útiles para compartir el estado.
3. Tienen muchas características de las clases: puede tener propiedades, métodos (asíncronos o no), inicializadores y subíndices, pueden implementar protocolos y pueden ser genéricos.
4. Los actores no admiten la herencia, por lo que no pueden tener inicializadores de conveniencia y no admiten las palabras claves como *final* ni *override*.
5. Todos los actores implementan automáticamente el protocolo *Actor*, que ningún otro tipo puede utilizar. Esto le permite escribir código restringido para trabajar solo con actores.

Cross-actor reference

Cuando decimos que los actores garantizan el aislamiento de los datos, queremos decir que todas las propiedades y funciones mutables dentro de un actor están

aisladas del acceso directo desde el exterior. Este aislamiento es una característica central de los actores y es crucial para garantizar la seguridad de datos en la programación concurrente. Pero ¿qué implica este aislamiento en la práctica?

Básicamente, si deseas leer una propiedad, cambiar un valor o llamar a un método de un actor, no puedes hacerlo directamente como lo harías con una clase o estructura normal. En cambio, debes «esperar tu turno». Esto se hace «enviando una solicitud» al actor. Luego, tu solicitud se pone en cola y se procesa a su vez. Solo cuando sea el turno de manejar tu solicitud podrás leer o modificar las propiedades del actor o llamar a sus métodos.

Este proceso se conoce como *cross-actor reference*. Cuando haces referencia o accedes a algo dentro de un actor desde fuera de ese actor, estás haciendo una referencia entre actores. En la práctica, esto significa utilizar patrones asíncronos, como *async* y *await,* para interactuar con el actor. Aquí hay un ejemplo de cómo interactuar con un actor :

```
let counter = CounterActor()

// Llamada asíncrona al método increment() del actor
await counter.increment()

// Llamada asíncrona al método getCount() del actor
let currentCount = await counter.getCount()
```

Una de las ventajas clave de los actores en Swift es que proporcionan un modelo de concurrencia seguro y estructurado. Debido a la exclusión mutua implícita, no es necesario preocuparse por problemas comunes de concurrencia como las condiciones de carrera y los bloqueos.

Serial Executor

Antes mencionamos que cada actor tiene una cola en serie interna, que es responsable de gestionar las tareas pendientes del actor, procesándolas una por una. Esta cola interna de un actor, conocida como *Serial Executor*, es algo parecida a *Serial DispatchQueue*. Sin embargo, existen diferencias cruciales entre estos dos, particularmente en cómo manejan el orden de ejecución de las tareas.

Una diferencia significativa es que las tareas que esperan en el *Serial Executor* de un actor no necesariamente se ejecutan en el orden en que fueron enviadas. Esta es una diferencia del comportamiento de *Serial DispatchQueue*, que realiza una estricta política de FIFO (primero en entrar, primero en salir). Con *Serial DispatchQueue*, las tareas se ejecutan exactamente en el orden en que se reciben.

Por otro lado, el runtime de un actor de Swift emplea un mecanismo de cola más

liviano y optimizado en comparación con un DispatchQueu, diseñado para aprovechar las capacidades de las funciones asíncronas de Swift. Esta diferencia surge de la naturaleza fundamental de los ejecutores versus DispatchQueues. Un ejecutor es esencialmente un servicio que gestiona el envío y ejecución de tareas. A diferencia de DispatchQueues, los ejecutores no están obligados a ejecutar trabajos estrictamente en el orden en que fueron enviados. En cambio, los ejecutores están diseñados para priorizar las tareas en función de varios factores, incluida la prioridad de las tareas, en lugar de únicamente según el orden de envío.

Hay varias reglas que tenemos que seguir cuando trabajamos con los actores:

- Acceder a propiedades de solo lectura en actores no requiere *async/await* ya que sus valores son inmutables y no cambian.
- Está prohibido modificar variables mutables desde fuera (*cross-actor reference*), incluso con *async/await*. Un actor solo puede cambiar su estado desde dentro.
- Todas las llamadas a actor tienen que ser asíncronas (*async/await*)

```
actor Account {
   let number: String = "IBAN---"
   var balance: Int = 100
   // ...
   func withdraw(amount: Int) {
      guard balance >= amount else { return }
      self.balance = balance - amount
   }
}
let account = Account()
/// ✅
let accountNumber = account.number
/// ✅
let balance = await account.balance
/// aquí hace falta await ❌
let balance = account.balance // Error
/// no hay await y tampoco se puede cambiar el valor con await ❌
account.balance = 1000 // Error
/// tampoco está bien porque llamamos desde cross-actor reference (fuera
await account.balance = 1000 // Error
/// ✅ se puede cambiar la propiedad balance desde dentro del actor
await account.withdraw(100)
```

Pero a veces tenemos los métodos que no cambian el estado de nuestro actor. Realmente el acceso a estos métodos no requiere un aislamiento. Para estos casos se puede utilizar una palabra clave *non-isolated*.

```
actor Account {
 let number: String = "IBAN ---"
```

```
  var balance: Int = 100

  nonisolated func getMaskedNumber() -> String {
     return String.init(repeating: "*", count: 12) + accountNumber.suffix
  }
  // ...
}

let account = Account()
/// ✅
account.getMaskedNumber()
```

Los miembros con la palabra clave (*nonisolated*) permiten acceder a ciertas partes de un actor sin necesidad de realizar llamadas asíncronas. Es muy importante saber distinguir entre los miembros mutables/inmutables para poder optimizar acceso a los miembros de actor y nuestro código.

Conclusiones

En este tutorial, hemos aprendido sobre los actores en Swift, una característica nueva y poderosa para escribir código concurrente de manera segura y eficiente. Los actores proporcionan un modelo estructurado para gestionar el acceso al estado compartido y eliminan muchos problemas comunes de concurrencia. Con los actores en Swift, los desarrolladores pueden escribir código concurrente de manera mucho más fácil y segura, lo que mejora la calidad y la robustez de sus aplicaciones.

Anton Zuev

iOS Developer. Puedes encontrarme en Autentia. Ofrecemos servicios de soporte a desarrollo, factoría y formación. Somos expertos en Java/Java EE/Javascript/Kotlin/Swift.