The Future of Swift?

Wednesday, January 18, 2017

Task Concurrency (5.0)

- `async/await` for elegant async operations
  - Native concurrency at language level
  - Solves completion handler „pyramid of doom“
  - C#, Javascript, Python, Kotlin etc.

Confidence: ✅

Task Concurrency (5.0)

Potential syntax:

```
async func downloadImage(from url: URL) -> Task<UIImage?> {
    do {
        let dataTask: Task<Data> = URLSession.shared.dataTask(with: url)
        let data: Data = try await dataTask
        return UIImage(data: data)
    } catch {
        return nil
    }
}

let image: UIImage? = await downloadImage(from: url)
```

Actor Model (5.0)

- Actor model to define tasks, along with managed state
- Each Actor is effectively
  - A `DispatchQueue`
  - State it manages
  - Operations that act on it
- Erlang, Akka (JVM)

Confidence: 🔜

Actor Model (5.0)

Potential syntax:

```
actor NetworkRequestHandler {
    private var userID: UserID

    async func processRequest(_ connection: Connection) {
        // send messages to other actors
        // create new actors
        // modify local state
    }
}

let requestHandler = NetworkRequestHandler()
await requestHandler.processRequest(connection)
```

One
Eternity
Later

Motivation

```swift
func refreshPlayers(completion: (() -> Void)? = nil) {
    refreshQueue.async {
        self.gameSession.allPlayers { players in
            self.players = players.map(\.nickname)
            completion?()
        }
    }
}
```

Motivation

```swift
func refreshPlayers(completion: (() -> Void)? = nil) {
    refreshQueue.async {
        self.gameSession.allPlayers { players in
            self.players = players.map(\.nickname)
            completion?()
        }
    }
}
```

🆕

```swift
func refreshPlayers() async {
    players = await gameSession.allPlayers().map(\.nickname)
}
```

Where we are

```swift
func processImageData2(completionBlock: (_ result: Image?, _ error: Error?) -> Void) {
    loadWebResource("dataprofile.txt") { dataResource, error in
        guard let dataResource = dataResource else {
            completionBlock(nil, error)
            return
        }
        loadWebResource("imagedata.dat") { imageResource, error in
            guard let imageResource = imageResource else {
                completionBlock(nil, error)
                return
            }
            decodeImage(dataResource, imageResource) { imageTmp, error in
                guard let imageTmp = imageTmp else {
                    completionBlock(nil, error)
                    return
                }
                dewarpAndCleanupImage(imageTmp) { imageResult in
                    guard let imageResult = imageResult else {
                        return // <- forgot to call the block
                    }
                    completionBlock(imageResult)
                }
            }
        }
    }
}

processImageData2 { image, error in
    guard let image = image else {
        display("No image today", error)
        return
    }
    display(image)
}
```

Proposed async/await syntax (1)

```swift
func loadWebResource(_ path: String) async throws -> Resource
func decodeImage(_ r1: Resource, _ r2: Resource) async throws -> Image
func dewarpAndCleanupImage(_ i : Image) async throws -> Image

func processImageData2() async throws -> Image {
    let dataResource = await try loadWebResource("dataprofile.txt")
    let imageResource = await try loadWebResource("imagedata.dat")
    let imageTmp = await try decodeImage(dataResource, imageResource)
    let imageResult = await try dewarpAndCleanupImage(imageTmp)
    return imageResult
}
```

Structured concurrency - tasks and child tasks

```swift
func makeDinner() async throws -> Meal { // Every asynchronous function is executing in a task
    async let veggies = try chopVegetables()
    async let meat = marinateMeat()
    async let oven = try preheatOven(temperature: 350)

    let dish = Dish(ingredients: await [veggies, meat]) // Suspension point
    return await try oven.cook(dish, duration: .hours(3)) // Another suspension point
}

func chop chopVegetables() async throws -> Vegetable {
    await try Task.checkCancellation() // Automatically throws `CancellationError`
    // chop chop chop ...
    await try Task.checkCancellation() // Canceled mid-way through chopping of vegetables
    // chop some more, chop chop chop ...
}
```

Task

A task can be in one of three states:

— A suspended task has more work to do but is not currently running.

— A running task is currently running on a thread.

— A completed task has no more work to do and will never enter any other state.

Partial tasks
Executors

Structured concurrency - task groups

```swift
/// Concurrently chop the vegetables.
func chopVegetables() async throws -> [Vegetable] {
    // Create a task group where each task produces (Int, Vegetable).
    await try Task.withGroup(resultType: (Int, Vegetable).self) { group in
        var veggies: [Vegetable] = gatherRawVeggies()

        // Create a new child task for each vegetable that needs to be
        // chopped.
        for i in veggies.indices {
            await try group.add {
                (i, veggies[i].chopped())
            }
        }

        // Wait for all of the chopping to complete, slotting each result
        // into its place in the array as it becomes available.
        while let (index, choppedVeggie) = await try group.next() {
            veggies[index] = choppedVeggie
        }

        return veggies
    }
}
```

Actors

```
actor class BankAccount {
    private let ownerName: String
    private var balance: Double
}

extension BankAccount {
    func deposit(amount: Double) async {
        balance = balance + amount
    }
}

extension BankAccount {
    func transfer(amount: Double, to other: BankAccount) async throws {
        guard amount <= balance else {
            throw BankError.insufficientFunds
            return
        }

        // Safe: this operation is the only one that has access to the actor's local
        // state right now, and there have not been any suspension points between
        // the place where we checked for sufficient funds and here.
        balance = balance - amount

        // Safe: the deposit operation is queued on the `other` actor, at which
        // point it will update the other account's balance.
        await other.deposit(amount: amount)
    }
}
```

Actors are exclusive executors

Global actors

```swift
@globalActor
struct UIActor {
    static let shared = UIActorInstance()
}

@UIActor
func drawAHouse(graphics: CGGraphics) {
    // draw draw draw ...
}
```

Concurrency Interoperability with Objective-C

Objective-C

```
- (void)fetchShareParticipantWithUserRecordID:(CKRecordID *)userRecordID
        completionHandler:(void (^)(CKShareParticipant * _Nullable, NSError * _Nullable))completionHandler;
```

Swift

```
func fetchShareParticipant(withUserRecordID userRecordID: CKRecord.ID,
                           completionHandler: @escaping (CKShare.Participant?, Error?) -> Void)
```

Swift 🆕

```
func fetchShareParticipant(withUserRecordID userRecordID: CKRecord.ID) async throws -> CKShare.Participant

guard let participant = await try? container.fetchShareParticipant(withUserRecordID: user) else {
    return nil
}
```

Defining asynchronous @objc methods in Swift

Swift 🆕

```swift
@objc func fetchImage(url: URL) async throws -> UIImage
```

Objective-C

```objc
- (void)fetchImage:(URL * _Nonnull)url
        completionHandler:(void (^ _Nullable)(UIImage * _Nullable, NSError * _Nullable))completionHandler;
```

Async handlers

Ability to declare a synchronous actor function as an asynchronous handler.

These functions behave externally like a synchronous function, but internally are handled like an asynchronous function.

This allows traditional “notification” methods, such as those on `UITableViewDelegate`, to perform asynchronous operations without cumbersome setup.

This is just the beginning

SOON Phase 2

Thank you

patrick.gaissert@maibornwolff.de