Concurrency实用教程

介绍

WWDC 21 Apple 在 Swift 5.5 中推出了一个革命性的并发编程技术 — **Concurrency**。本教程主要针对该技术进行介绍，从最基础的语法开始，采用大量的案例进行通俗易懂的讲解，帮助大家快速掌握 Concurrency 的精髓并用于实际开发。

主要内容

- Intro
- async/await
- get async与async let
- Task与TaskGroup
- actor
- Continuations
- AsyncSequence与AsyncStream

环境要求

1. 开发工具：Xcode 13 及以上。
2. 运行环境：iOS 15 / iPadOS 15 / macOS 12 / tvOS 15 / watchOS 8 及以上。
3. **Xcode 13.2 以后语法支持到 iOS 13 / iPadOS 13 / macOS 10.15 / tvOS 13 / watchOS 6，但同时推出的 API 依然需要 iOS 15 以上。**

学习要求

1. 掌握 Swift 基本语法。
2. 熟悉 iOS 开发。

作者

- YungFan，杨帆，高校教师，开发者。
- Email：yungfan@vip.163.com。
- GitHub：https://github.com/yungfan。
- 在线课程：https://ke.qq.com/cgi-bin/agency?aid=67223。
- 微信公众号：YungFan。
- 博客：**掘金**、**腾讯云·云社区** 和 **简书**。

勘误

各位读者在阅读本教程时，如果对其中的内容有疑义或者修改建议，欢迎联系作者。

版权

更新与修订

时间	章节	更新/修订内容
2021.10	**Task与TaskGroup**	Task—案例-增加案例二：加载网络图片
	actor	1.增加GlobalActor 2.增加Sendable
2021.11	**Task与TaskGroup**	TaskGroup—增加生命周期
	actor	Isolation—增加nonisolated
	AsyncSequence与AsyncStream	AsyncStream—增加扩展Timer
2021.12	**所有章节**	Xcode 13.2 以后语法支持到低版本

01 Intro

概念

并发（Concurrency）指的是异步和并行的组合。

1. **异步**是相对于同步而言的，同步指的是代码按照某个事先确定的顺序执行，在当前代码执行完之前，后面的代码不会执行。而异步指的是代码不再简单地按照既定顺序执行，因为**异步代码允许被挂起和恢复**。
2. **并行**是指可以同时运行多段代码，执行多个任务。

重要性

异步和并行的结合即并发性对于现代 App 是必不可少的。在 Swift 5.5 之前，Apple 推出了 GCD、Operation与OperationQueue 来进行并发编程。为了提供一种更高效、更方便、更安全的并发编程方法，在 Swift 5.5 中引入了新的并发编程体系 **Concurrency**。

特点

- Concurrency 允许开发者**以类似同步的方式来编写包含异步代码的复杂逻辑，代码量更少，结构更清晰，可读性更强，理解更容易。**
- Concurrency 虽然建立在线程的基础之上，但与传统的多线程并发又有着巨大的差异。首先它是由纯 Swift 语言特性组成，其次相比多线程，它有如下优势：
  - 会根据 CPU 的核心数创建线程数量，不会出现线程爆炸的问题。
  - 不需要复杂的 CPU 上下文切换（线程切换），不同任务的切换就是不同函数的调用，性能更高。
  - 执行任务的线程不会阻塞，需要挂起时它会主动释放让其能够执行其他任务，效率更高。

> **重要**：使用 Concurrency 时不必关心线程的开辟和管理问题，甚至不需要知道线程的存在，因为 Swift 和 iOS 已经帮我们解决了这个问题，我们只需要按照规范编写异步代码即可，并行执行交给系统。

结构化与非结构化并发

- **结构化并发**是指执行操作的任务之间、任务与其所处的上下文之间具有一定的结构关系。
- **非结构化并发**是指从任意一个地方开始异步任务，然后在另外一个地方对它进行其他操作。
- 使用 `async let` 和 `TaskGroup` 来处理结构化并发。
- 使用 `Task.init` 和 `Task.detached` 来处理没有任何结构的非结构化并发。

02 async/await

概念

- **解决的是并发的第一个问题：异步。**
- 异步代码最主要的问题是结构不像同步代码那样简单。当程序中出现多个异步任务时，执行的过程管理（控制流）会变得非常复杂。例如在函数中进行异步操作时，以前的做法是将耗时的任务放到子线程中执行，在完成以后再用一个逃逸闭包（如 completionHandler）进行**回调**处理，这种方式一旦程序稍微复杂一点，就会存在如下的问题：
  - 回调地狱：回调嵌套，代码可读性差。
  - 错误处理逻辑复杂：每个分支都有可能发生错误，需要回调错误，非常复杂。
  - 嵌套多层后，条件判断变得复杂，且容易出错。
  - 容易忘记回调或者回调后忘记返回。

```
func doSomething() -> String {
    Thread.sleep(forTimeInterval: 3)

    return "async value"
}

// GCD
func asyncOperation(completionHandler: @escaping (String) -> Void) {
    DispatchQueue.global().async { // 开启新线程执行
        // 单个或多个异步操作
        print("before")

        let str = doSomething() // 线程执行到这里会阻塞直到doSomething返回，在这段时间线程将不能执行任何其他操作

        print("after")

        // 回调
        completionHandler(str)
    }
}

asyncOperation { value in
    DispatchQueue.main.async {
        print(value)
    }
}

/**
before
after
async value
*/
```

同步与异步函数

- 默认情况下，所有函数都是同步函数，缺点就是执行时会阻塞。
- 异步函数具有**挂起**的特殊能力，而同步函数没有（因此同步函数不能直接调用异步函数，因为不知道如何挂起自己）。
- 一个挂起的函数**不会阻塞它正在运行的线程，它会释放那个线程，被释放的线程就可以执行其他的任务。**
- 当异步函数从挂起状态恢复后可以继续从挂起的位置往后执行，但需要注意**挂起之前和之后的代码可能运行在同一个线程，也可能在不同的线程**，这取决于系统的调度。

使用规则

1. 使用 `async` 关键字标记的函数为异步函数，写在参数之后返回值之前，**表示函数可挂起。**
2. 使用 `await` 关键字调用异步函数，写在异步函数调用之前，**真正挂起异步函数。**
3. 异步函数内可以调用其他的异步函数，也可以调用同步函数，但同步函数**不能**调用异步函数。
4. 异步函数的调用必须用 `await` （使用 `async let` 接收时不需要，后面会讲解）。
5. 执行到 `await` 时，**会挂起异步函数并将控制权交给系统**（执行任务的线程会被系统回收用于执行其他任务），**当异步函数执行完毕后**（异步函数恢复），**系统会将控制权重新返还给调用者并从挂起的位置继续执行后续代码。**

语法

基本语法

定义

```
// 定义异步函数
func generateNum1() async -> Int {
    return 1
}

func generateNum2() async -> Int {
    return 2
}

func generateNum3() async -> Int {
    return 3
}
```

调用

1. 通过异步函数。

```
func callAsync() async {
    // 异步执行，当前代码被挂起
    let x = await generateNum1()
    // 异步执行，当前代码被挂起
    let y = await generateNum2()
    // 异步执行，当前代码被挂起
    let z = await generateNum3()

    let res = x + y + z
    print(res)
}
```

2. 通过 Task。

```
// 1. Task.init中调用，因为同步函数无法调用异步函数，所以需要创建一个异步任务，后面同理。
Task {
    let one = await generateNum1()
    print(one, Thread.current)
}

// 2. Task.detached中调用
Task.detached {
    let two = await generateNum2()
    print(two)
}

Task {
    await callAsync()
}
```

3. 通过 SwiftUI 的 task 修饰符。

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .task {
                let x = await generateNum1()
                let y = await generateNum2()
                let z = await generateNum3()

                let res = x + y + z
                print(res)
            }
    }
}
```

4. 通过 @main。

```
@main
struct MainApp {
    static func main() async {
        let x = await generateNum1()
        let y = await generateNum2()
        let z = await generateNum3()

        let res = x + y + z
        print(res)
    }

    func generateNum1() async -> Int {
        return 1
    }

    func generateNum2() async -> Int {
        return 2
    }

    func generateNum3() async -> Int {
        return 3
    }
}
```

注意：这种方式需要新建一个 macOS 12 及以上的命令行程序。

异常处理

- 抛出异常的语法：`async throws`。
- 处理异常的语法：`try await`。

```
enum CustomError: Error {
    case negative
}

func callAsync() async {
    // 异常处理
    do {
        try await generate(number: 1)
        try await generate(number: -1)
    } catch {
        print(error)
    }
}

// 定义时表示会有异常抛出
func generate(number: Int) async throws {
    if number < 0 {
        // 抛出异常
        throw CustomError.negative
    } else {
        print(number)
    }
}

Task {
    await callAsync()
}

/**
1
negative
*/
```

案例

```
func read() async {
    print("read start \(Thread.current)")
```

```swift
    await write()
    print("read finish \(Thread.current)")
}

func write() async {
    print("write start \(Thread.current)")
    print("write finish \(Thread.current)")
}

Task.detached {
    print("before read \(Thread.current)")
    await read()
    print("after read \(Thread.current)")
}

/**
before read
read start
write start
write finish
read finish
after read
*/
```

1. 程序运行首先执行到 Task 块，首先打印 `before read`。
2. 遇到 await，**挂起异步函数，暂停等待**，并将 read 函数交给某个线程执行。
3. 线程执行 read 函数，打印 `read start`。
4. 线程遇到 await，**挂起异步函数，暂停等待**，并将 write 函数交给某个线程执行。
5. 线程执行 write 函数，打印 `write`，异步函数执行完毕并返回调用处的 await 继续执行。
6. 线程继续执行，打印 `read finish`，异步函数执行完毕并返回调用处的 await 继续执行。
7. 打印 `after read`。

03 get async与async let

get async

除了方法可以异步，计算属性也可以异步，但只有**只读属性**支持 async。

- 结构体或类。

```
class Person {
    let name: String = "zhangsan"
    var bmi: Double {
        get async {
            return await calBMI(weight: 80.0, height: 1.8)
        }
    }

    func calBMI(weight: Double, height: Double) async -> Double {
        weight / (height * height)
    }
}

struct ContentView: View {
    let person = Person()
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    let bmi = await person.bmi
                    print(bmi)
                }
            }
    }
}
```

- 协议。

```
protocol SomeProcotol {
    var asyncProperty: Bool { get async }
    // 抛出异常
    var asyncThrowsProperty: String { get async throws }
}
```

async let

- **解决的是并发的另一个问题：并行。**
- **结构化并发的第 1 种形式，可以并行执行多个任务，即异步操作之间为并行执行。**
- 使用 `async let` 接收异步函数返回的结果，这样调用时可以省去 `await` 。
- `async let` 只能在异步函数或者 Task 中使用，不允许在顶层代码和同步函数中使用。
- 目前在 playground 中使用会有问题。

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    await callAsync()
                }

                Task {
                    await callAsyncLet()
                }

                // 2. Task使用
                Task {
                    async let x = generateNum1()
                    async let y = generateNum2()
                    async let z = generateNum3()
```

```
                let res = await x + y + z
                print(res)
            }
        }
}

func generateNum1() async -> Int {
    // 新的休眠方式，单位纳秒
    await Task.sleep(1 * 1000000000)
    print(#function, Thread.current)
    return 1
}

func generateNum2() async -> Int {
    await Task.sleep(2 * 1000000000)
    print(#function, Thread.current)
    return 2
}

func generateNum3() async -> Int {
    await Task.sleep(3 * 1000000000)
    print(#function, Thread.current)
    return 3
}

func callAsync() async {
    // 3个操作顺序执行
    // 耗时1秒
    let x = await generateNum1()
    // 耗时2秒
    let y = await generateNum2()
    // 耗时3秒
    let z = await generateNum3()
    // 6秒以后才会计算
    let res = x + y + z
    print(res)
}

    // 1. 异步函数中调用
    func callAsyncLet() async {
        // 3个操作并行执行
        // 耗时1秒
        async let x = generateNum1()
        // 耗时2秒
        async let y = generateNum2()
        // 耗时3秒
        async let z = generateNum3()
        // 3秒以后就会计算
        let res = await x + y + z
        print(res)
    }
}
```

注意：不能使用 `async var` 。

04 Task与TaskGroup

概念

- 为了提供并发执行的环境和并行执行的能力，Concurrency 还提供了 2 种特殊的类型 Task 和 TaskGroup。
- 单个任务使用 Task，多个任务使用 TaskGroup。

Task

- Task 可以处于以下 3 种状态之一：运行、暂停或完成。
- Task 可以设置优先级：`high`、`medium`、`low`、`userInitiated`、`utility`、`background`。
- 常见属性与方法：
  - `value`：获取当前 Task 的执行结果。
  - `priority`：获取 Task 优先级。
  - `isCancelled`：Task 是否取消。
  - `sleep()`：休眠 Task，单位**纳秒**。与线程睡眠不同是它不会阻塞线程，休眠时该线程会被系统按需用作他用。
  - `cancel()`：取消 Task。
  - `suspend()`：挂起 Task。
  - `yield()`：暂停 Task，当某个 Task 持续执行，可以调用它以释放机会给其他 Task 执行。但调用它并不意味着当前 Task 将停止运行，如果它比其他 Task 的优先级更高，那么调用之后有可能又立即执行。

创建

- 有 2 种常见的创建 Task 的方式：

```
// 简称Task.init
public init(priority: TaskPriority? = nil, operation: @escaping @Sendable () async -> Success)

// 简称Task.detached
public static func detached(priority: TaskPriority? = nil, operation: @escaping @Sendable () async -> Success) -> Task<Succes
```

注意：它们之间的最大区别是 `Task.init` 会继承调用者的优先级、任务本地值和 actor 上下文，而 `Task.detached` 不会继承这些信息。

- 基本使用。

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                eat()

                drink()
            }
    }

    func eat() {
        DispatchQueue.main.async {
            Task {
                print(#function, Thread.current)
            }
        }
    }

    func drink() {
        DispatchQueue.main.async {
            Task.detached {
                print(#function, Thread.current)
            }
        }
    }
}
```

案例

- 案例一：获取网络新闻。

```
// 服务器返回的数据对应的Model
struct NewsModel: Codable {
    var reason: String
    var error_code: Int
    var result: Result
}

struct Result: Codable {
    var stat: String
    var data: [DataItem]
}

struct DataItem: Codable {
    var title: String
    var date: String
    var category: String
    var author_name: String
    var url: String
}

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    await task()
                }
            }
    }

    func task() async {
        // 构造Task并设置优先级
        // Task一旦创建就会自动执行，不需要调用任何方法
        // Task与其他代码并行运行
        let userTask = Task(priority: .high) { () -> NewsModel in
            let url = URL(string: "http://v.juhe.cn/toutiao/index?type=top&key=d1287290b45a69656de361382bc56dcd")!
            // API必须iOS15以上
            let (data, _) = try await URLSession.shared.data(from: url)
            return try JSONDecoder().decode(NewsModel.self, from: data)
        }

        // 1. 通过value属性直接获取结果
        do {
            let newsModel = try await userTask.value
            print(newsModel.result.data.count)
        } catch {
            print(error.localizedDescription)
        }

        // 2. Task也提供了一个result属性（Result 类型）获取结果
        do {
            let result = await userTask.result
            let newsModel = try result.get()
            print(newsModel.result.data.count)
        } catch {
            print(error.localizedDescription)
        }
    }
}
```

- 案例二：加载网络图片。

```
@MainActor
struct ContentView: View {
    // 保存Task，之后可以获取值，也可以取消
    @State private var task: Task<Void, Never>?

    @State private var image: UIImage?

    var body: some View {
```

```
        Text("Hello, world!")
            .onAppear {
                startTask()
            }

        Button("取消") {
            task?.cancel()
        }

        if let image = image {
            Image(uiImage: image)
                .resizable()
                .frame(width: 100, height: 100)
        }
    }

    func startTask() {
        task = Task {
            image = try? await downloadImage()

            task = nil
        }
    }

    func downloadImage() async throws -> UIImage? {
        let imageUrl = URL(string: "https://ss0.bdstatic.com/70cFuHSh_Q1YnxGkpoWK1HF6hhy/it/u=2718219500,1861579782&fm=26&gp=
        let imageRequest = URLRequest(url: imageUrl)
        // API必须iOS15以上
        let (imageData, _) = try await URLSession.shared.data(for: imageRequest)
        return UIImage(data: imageData)
    }
}
```

Task Local Value

- 任务本地值，可以使用 TaskLocal 存储/设置某些数据，该数据会存储在 Task 的上下文中，Task 及其所有子 Task 都可以使用它。
- 通过使用 `@TaskLocal` 修饰**静态变量**实现，即 `@TaskLocal static var`，这样就可以在 Task 中读取该数据，并可以通过 `DataType.$yourProperty.withValue(someValue)` 的方式设置该数据。

```
// 枚举、结构体、类都可以
enum TaskStorage {
    @TaskLocal static var name: String?
}

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                // Task.init继承上下文，可以读取
                Task {
                    TaskStorage.$name.withValue("TaskLocal") {
                        print("task:", TaskStorage.name ?? "unknown")

                        // Task.init继承上下文，可以读取
                        Task {
                            print("子task:", TaskStorage.name ?? "unknown")

                            // Task.detached无法读取
                            Task.detached {
                                print("子detached:", TaskStorage.name ?? "unknown")
                            }
                        }

                        // Task.detached无法读取
                        Task.detached {
                            print("detached:", TaskStorage.name ?? "unknown")
                        }

                        // Task.detached可以读取一个新的值
                        Task.detached {
                            TaskStorage.$name.withValue("TaskLocal2") {
                                print("new detached:", TaskStorage.name ?? "unknown")
```

```
                    }
                }
            }
        }
    }
}
```

TaskGroup

- 这是**并行的另一种解决方案。**
- **结构化并发的第 2 种形式，也可以并行执行多个任务。**
- 通过 `withTaskGroup()` 或 `withThrowingTaskGroup()` 方法创建。该方法的第一个参数指定最终生成的结果类型（如果没有任何输出可以写 `Void.self` ）。最后一个参数为闭包，需要在其中处理所有的任务，该闭包只有一个参数，类型为 `TaskGroup<最终的结果类型>` 。
- 常见属性与方法：
  - `isEmpty`：TaskGroup 是否为空。
  - `isCancelled`：TaskGroup 是否取消。
  - `cancelAll()`：取消 TaskGroup 中所有的尚未执行的任务。
  - `addTask()`：添加 Task 到 TaskGroup。
  - `next()`：等待下一个 Task 执行完并返回 Task 返回的值。
  - `waitForAll()`：等待所有 Task 完成后再返回。
- 改写 async let 案例。

```swift
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    await taskGroup()
                }
            }
    }

    func generateNum1() async -> Int {
        await Task.sleep(1 * 1000000000)

        print(#function, Thread.current)

        return 1
    }

    func generateNum2() async -> Int {
        await Task.sleep(2 * 1000000000)

        print(#function, Thread.current)

        return 2
    }

    func generateNum3() async -> Int {
        await Task.sleep(3 * 1000000000)

        print(#function, Thread.current)

        return 3
    }

    func taskGroup() async {
        // TaskGroup
        await withTaskGroup(of: Int.self) { group in
            // 内部Task处理任务
            // 可以设置优先级
            // 3个操作并行执行
            group.addTask(priority: .medium) {
                await self.generateNum1()
            }
            group.addTask {
                await self.generateNum2()
            }
            group.addTask {
```

```
            await self.generateNum3()
        }

        var sum: Int = 0
        // 计算，AsyncSequence
        // 按Task完成顺序处理结果
        // 由于异步for循环，在所有任务完成之前，闭包并不会返回
        // 3秒以后就会计算
        for await res in group {
            sum += res
        }

        print(sum)
    }
  }
}
```

- 案例一：单一结果类型。

```
struct Student {
}

struct Course {
    let id: Int
}

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    let course = await fetchCourse(student: Student())
                    print(course)
                }
            }
    }

    func getCourseIds(for: Student) async -> [Int] {
        return [1, 2, 3]
    }

    func getCourse(withId: Int) async -> Course {
        return Course(id: withId)
    }

    func fetchCourse(student: Student) async -> [Course] {
        // 通过学生获取所有课程id
        let ids = await getCourseIds(for: student)
        // 获取所有课程
        return await withTaskGroup(of: Course.self) { group in
            // 添加任务到任务组获取课程
            for id in ids {
                group.addTask {
                    await self.getCourse(withId: id)
                }
            }

            var courses = [Course]()
            // 任务完成以后获取任务的结果并添加到数组
            for await course in group {
                courses.append(course)
            }

            return courses
        }
    }
}
```

- 案例二：多种结果类型。

```
enum ResultType {
    case course, teacher
}

struct Student {
```

```
    let id: Int
    // 结果类型
    let type: ResultType
}

struct Course {
    let id: Int
}

struct Teacher {
    let id: Int
}

// 任务结果，包含两种结果类型
enum TaskResult {
    case course(Course)
    case teacher(Teacher)
}
```

```
struct ContentView: View {
    let students = [Student(id: 1, type: .course), Student(id: 2, type: .teacher), Student(id: 3, type: .course), Student(id:

    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    let result = await fetchCourseAndTeacher(students: students)
                    for item in result {
                        switch item {
                        case let .course(value):
                            print(value)
                        case let .teacher(value):
                            print(value)
                        }
                    }
                }
            }
    }
```

```
func getCourse(withId: Int) async -> Course {
    return Course(id: withId)
}

func getTeacher(withId: Int) async -> Teacher {
    return Teacher(id: withId)
}

func fetchCourseAndTeacher(students: [Student]) async -> [TaskResult] {
    return await withTaskGroup(of: TaskResult.self) { group in
        // 添加任务到任务组获取课程和教师
        for student in students {
            group.addTask {
                switch student.type {
                case .course:
                    let course = await getCourse(withId: student.id)
                    return TaskResult.course(course)
                case .teacher:
                    let teacher = await getTeacher(withId: student.id)
                    return TaskResult.teacher(teacher)
                }
            }
        }
        // 任务完成以后获取任务的结果并添加到数组
        var results = [TaskResult]()

        for await result in group {
            results.append(result)
        }

        return results
    }
}
```

生命周期

- 没有任何 TaskGroup 的子 Task 可以超越 `withTaskGroup` 的闭包范围。
- TaskGroup 中管理的所有 Task 完成之前，不允许 TaskGroup 完成。
- `group.addTask` 中不能直接修改捕获的上下文中的变量。

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    print("任务组开始")

                    // 无结果类型，传入Void
                    await withTaskGroup(of: Void.self) { group in
                        var a = 10

                        for _ in 0 ... 2 {
                            group.addTask {
                                await self.doSomething()
                                print("执行任务")
                                // 报错：Mutation of captured var 'a' in concurrently-executing code
                                a = 100
                            }
                        }
                        print("完成任务")
                    }

                    print("任务组结束")
                }
            }
    }

    func doSomething() async {
    }
}
```

async let vs TaskGroup

1. `async let` 适用于多个异步操作之间并无关联的情况，而 `TaskGroup` 适合于异步操作之间关联较为紧密的情况。
2. `async let` 没有类似 `TaskGroup` 的取消方法。
3. `async let` 无法像 `TaskGroup` 可以存储以后进行传递。
4. `TaskGroup` 需要通过**异步 for 循环**进行任务结果的处理，而 `async let` 不需要。

> **注意**：实际开发中，建议优先考虑使用 `async let` 。

Tasks Hierarchy

- Task 可以包含子 Task，TaskGroup 既可以包含子 Task 也可以包含子 TaskGroup，子 Task 和 TaskGroup 还可以继续包含，这就构成了 **Tasks Hierarchy**。子 Task 将继承父 Task 的 `priority` 和 `Task Local Value` ，然后将这些获取的值传递给整个层次结构。
- 当父 Task 或 TaskGroup 被取消时，Swift 会自动取消所有子 Task。当需要手动取消一个任务时，可以把 Task 或 TaskGroup 存起来，然后适时调用它们的取消方法。但 **Swift 使用的是 `cooperative cancellation` （协同取消），因此子 Task 必须检查取消状态，否则它们可能会继续执行，只是被取消的 Task 的运行结果会被丢弃。**

05 actor

引入

并发编程必然会产生数据竞争的问题（**多个线程同时访问且有写操作**，即数据同步操作，多线程编程中使用锁或信号量技术来处理）。

```
class SaleTicket {
    // 总票数
    var number = 100

    func saleTicket() async {
        while true {
            // 卖完
            guard number > 0 else {
                print("票已卖完")
                return
            }

            // number减1
            number -= 1
            print("\(Thread.current)卖出去一张票，还剩\(number)张")
        }
    }
}

struct ContentView: View {
    let sale = SaleTicket()

    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task.detached {
                    await sale.saleTicket()
                }

                Task.detached {
                    await sale.saleTicket()
                }

                Task.detached {
                    await sale.saleTicket()
                }
            }
    }
}
```

概念

- 为了解决数据竞争问题，Concurrency 引入了 actor，**同一时刻只允许一个线程来访问它。**（将上面案例中的 class 改为 actor 查看结果）
- actor 类似于 class，也是一种**引用类型**，也可以有方法、属性和构造函数、还可以遵守协议。但 actor **不支持继承**，**不能有便利构造函数**，**也不支持** `final` **或** `override` 。
- actor 都遵守了 `Actor` 协议和 `AnyObject` 协议。

Isolation

- 隔离是 actor 中的一个重要概念，隔离的是 actor 的内部状态和外界访问。
- actor 内部：可以通过**同步/异步**的方式**读取/设置**其属性和进行方法的调用。
- actor 外部：只能通过**异步**的方式**读取**其属性和进行方法的调用，且**必须使用 await。无论如何都不允许外部设置属性。**

```
// 火车站A
actor StationA {
    // 总票数
    var totalTicket = 100000

    // actor可以同步或异步地对内部的属性进行读取/设置
```

```
    // 卖票窗口1
    func saleWindowOne(ticket: Int) {
        totalTicket -= ticket
        print(ticket)
    }

    // 卖票窗口2（异地）
    func saleWindowTwo(ticket: Int) async {
        totalTicket -= ticket
        print(ticket)
    }
}

// actor可以同步或异步地调用内部的方法
extension StationA {
    // 售票员One卖票
    func salerOne(ticket: Int) {
        saleWindowOne(ticket: ticket)
    }

    // 售票员Two卖票（异地）
    func salerTwo(ticket: Int) async {
        await saleWindowTwo(ticket: ticket)
    }
}

// 火车站B
actor StationB {
    // actor外可以异步读取其属性，但不能设置
    func hasTicket(station: StationA) async -> Bool {
        // 修改报错：Actor-isolated property 'totalTicket' can only be mutated from inside the actor
        // station.totalTicket -= 1
        return await station.totalTicket >= 0
    }
}

// actor外只能通过异步的方式读取其属性与调用其方法
func salerThree(ticket: Int, to station: StationA) async {
    await station.totalTicket
    await station.saleWindowOne(ticket: ticket)
    await station.saleWindowTwo(ticket: ticket)
}
```

- 可以使用 `isolated` 对函数参数进行隔离。

```
actor StationA {
    var totalTicket: Int = 100000

    func saleWindowThree(ticket: Int) {
        totalTicket -= ticket

        print(ticket)
    }
}

// 虽然在actor外部，但加了isolated后函数就像在actor内使用一样，可以直接访问actor的属性与方法，不需要加await
func salerOne(station: isolated StationA) {
    station.saleWindowThree(ticket: 1)
}

func salerTwo(station: StationA) async {
    // 报错：Call to actor-isolated global function 'salerOne(station:)' in a synchronous nonisolated context
    // salerOne(station: station)

    // 调用salerOne必须在异步环境中
    await salerOne(station: station)
}
```

nonisolated

可以使用 `nonisolated` 对计算属性（存储属性不行）和方法进行隔离解除，这样就可以在外部调用它而无需使用 `await` 。

```
actor StationA {
```

```
    var a = 10

    nonisolated var totalTicket: Int {
        get {
            100000
        }
        set {
        }
    }

    // 无法在内部读取/设置非nonisolated属性
    nonisolated func saleWindow(ticket: Int) {
        let remainTicket = totalTicket - ticket

        print(remainTicket)
    }
}

// 外部使用
actor StationB {
    init() {
        let actor = StationA()
        print(actor.totalTicket)
        actor.saleWindow(ticket: 10)
    }
}
```

MainActor

- **由于 Concurrency 并不能保证执行 await 前面代码的线程与执行后面代码的线程相同**，而 iOS 又规定 UI 的处理必须在主线程，为了保证当前更新 UI 的代码一定运行在主线程，可以使用 MainActor。MainActor 是一种特殊类型的 actor，它总是运行在主线程之上，表示为 `@MainActor`。
- 使用 `@MainActor` 可以修饰类型、函数（参数）、属性、属性包装，它符合以下规则：
  - 修饰结构体/枚举时，其**所有成员**都会变为 MainActor。
  - 修饰结构体/枚举中的方法，该方法会变为 MainActor。
  - 修饰类时，其**所有成员和子类**都会变为 MainActor。
  - 修饰类中的方法，该方法及其任何重写也会变为 MainActor。
  - 修饰函数的参数，该参数会变为 MainActor。
  - 类型中使用了 `@MainActor` 修饰的属性包装，该类型自动变为 MainActor（如 @State、@StateObject 和 @ObservedObject）。
- **对 MainActor 中的属性的任何读写和方法的调用都会发生在主线程**，这意味着可以消除类似 `DispatchQueue.main.async` 这样显式回到主线程执行的代码块。
- 在 Swift 5.5 中，所有 UIKit 和 SwiftUI 组件都被已经标记为 MainActor，因此在使用 Concurrency 完成后台操作进行 UI 更新时，不用再担心忘记回到主线程。

```
// 传统方式
class Model1 {
    func handleData() {
        DispatchQueue.global().async {
            print("处理数据")

            DispatchQueue.main.async {
                print("更新UI")
            }
        }
    }
}

// 修饰属性
struct Model2 {
    @MainActor
    var name: String = ""
}

// 修饰方法
class Model3 {
    @MainActor
    func handleData() {
        print("处理数据")
    }
}
```

```swift
// 修饰类型
@MainActor
struct Model4 {
    func handleData() {
        print("处理数据")
    }
}

// 修饰参数
func handleData(completion: @MainActor @escaping () -> Void) {
    DispatchQueue.global().async {
        Task {
            await completion()
        }
    }
}
```

- 案例。

```swift
class Model {
    @MainActor
    func handleData() async {
        print("Handle Data: \(Thread.current)")
    }

    func handleData2() {
        print("Handle Data2: \(Thread.current)")
    }

    func handleData3() async {
        print("Handle Data3: \(Thread.current)")
    }
}

@MainActor
struct ContentView: View {
    let model = Model()

    var body: some View {
        Text("Hello, world!")
            .onAppear {
                // 方式一
                Task.detached {
                    await self.model.handleData()
                }

                // 方式二
                Task.detached {
                    await MainActor.run {
                        self.model.handleData2()
                    }
                }

                // 方式三
                Task.detached { @MainActor in
                    await self.model.handleData3()
                }
            }
    }
}
```

GlobalActor

全局参与者， MainActor 就是一种全局参与者，它所修饰的内容会在同一个线程上运行。

```swift
// 换成class也可以
@globalActor
struct SomeActor {
    actor ActorType {
    }

    static let shared = ActorType()
}
```

```swift
struct Person {
    let id = UUID()
    let name: String
}

struct ContentView: View {
    @State @SomeActor var person: [Person] = []

    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task.detached {
                    await addPeople()
                }

                Task.detached {
                    if let p = await randomPeople() {
                        print(p.name)
                    }
                }

                Task.detached {
                    await showName()
                }
            }
    }

    @SomeActor
    func addPeople() {
        print(#function, Thread.current)
        person += [Person(name: "zhangsan"), Person(name: "lisi")]
    }

    @SomeActor
    func randomPeople() -> Person? {
        print(#function, Thread.current)
        return person.randomElement()
    }

    @SomeActor
    func showName() async {
        print(#function, Thread.current)
        for p in person {
            print(p.name)
        }
    }
}
```

Sendable

- Sendable 类型指的是**可以安全地并发共享的类型**。
- Sendable 类型可以进行编译时检查来帮助编写正确并发代码。
- 哪些数据属于 Sendable 类型?
    - 所有内置的值类型，如 Bool、Int、String 等。
    - 包装值类型的可选型，如 Bool?、Int?、String? 等。
    - 元素是值类型的集合，如 Array、Dictionary<Int, String>、Set。
    - 元素是值类型的元组，如 (String, String, Int, Int)。
    - 元类型，如 String.self。
- 如果是自定义类型，那么需要注意：
    - 如果是 actor，其内部已经做了处理，自动符合 `Sendable`，不需要处理。
    - 如果是 enum 或者 struct，如果里面包含的数据符合 `Sendable`，那么其本身也就自动符合 `Sendable`。
    - 如果是 class，要么继承自 NSObject，要么所有属性都是**常量**且符合 `Sendable`，同时二者都要求 class 被标记为 **final**。
- 让类型遵守 `Sendable` 协议以后就是 Sendable 类型。
- 对于可以跨 actor 传递的函数，可以将它们标记为 `@Sendable`。

06 Continuations

概念

- 将基于回调的老异步代码转换为支持 Concurrency 语法的新异步代码的技术称之为 **Continuations**。
- Swift 提供了 `withCheckedContinuation()` 和 `withCheckedThrowingContinuation()` 方法用于将基于回调的异步代码改造为 Concurrency 的形式，二者的区别在于后者可以抛出异常。
- Xcode 13 提供了一键转换的方法：选中某个需要转换的方法名，然后右击选择 Refactor 弹出菜单，共有 3 种转换方式。
  i. `Convert Function to Async`：原方法直接转换为异步方式。
  ii. `Add Async Alternative`：原方法保留，但标记为 `deprecated`，然后新建一个异步方法调用原方法。
  iii. `Add Async Wrapper`：原方法保留，新建一个异步方法调用原方法。（**建议使用**）

withCheckedContinuation

最后一个参数为 `(CheckedContinuation<T, Never>) -> Void`，T 为返回的结果类型，即老异步代码中逃逸闭包的参数类型。

```
struct ContentView: View {
    @State private var text = ""

    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    text = await getData()
                }
            }

        Text(text)
            .bold()
            .foregroundColor(.red)
    }

// 以前的方法，不需要改
// 需要返回的结果类型为String
func getData(completionHandler: @escaping (String) -> Void) {
    DispatchQueue.global().async {
        completionHandler("Data")
    }
}

// 再构造一个方法，改造为Concurrency形式
func getData() async -> String {
    // 在withCheckedContinuation()方法中包装对getData的调用
    return await withCheckedContinuation { continuation in
        getData { result in
            // 恢复等待继续的任务，让它从暂停点正常返回
            continuation.resume(returning: result)
        }
    }
}
```

withCheckedThrowingContinuation

最后一个参数为 `(CheckedContinuation<T, Error>) -> Void`，T 的含义同上 **withCheckedContinuation**。

```
enum CustomError: Error {
    case nilError
}

@MainActor
struct ContentView: View {
    @State private var image: UIImage?

    var body: some View {
```

```
        Text("Hello, world!")
            .onAppear {
                Task {
                    do {
                        image = try await downloadImage()
                    } catch {
                        print(error)
                    }
                }
            }

        if let image = image {
            Image(uiImage: image)
                .resizable()
                .frame(width: 200, height: 200)
        }
    }
}

// 需要返回的真正结果为UIImage?
func downloadImage(completionHandler: @escaping (_ image: UIImage?, _ error: Error?) -> Void) {
    let imageUrl = URL(string: "https://ss0.bdstatic.com/70cFuHSh_Q1YnxGkpoWK1HF6hhy/it/u=2718219500,1861579782&fm=26&gp=

    let imageTask = URLSession.shared.dataTask(with: imageUrl) { data, response, _ in
        guard let data = data, let image = UIImage(data: data), (response as? HTTPURLResponse)?.statusCode == 200 else {
            completionHandler(nil, CustomError.nilError)
            return
        }

        completionHandler(image, nil)
    }

    imageTask.resume()
}

// 再构造一个方法，改造为Concurrency形式
func downloadImage() async throws -> UIImage {
    // 在withCheckedThrowingContinuation()方法中包装对downloadImage的调用
    return try await withCheckedThrowingContinuation { continuation in
        downloadImage { result, error in
            // 必须每个分支都调用一次continuation
            if let error = error {
                continuation.resume(throwing: error)
                return
            }
            guard let result = result else {
                fatalError("Expected non-nil result 'result' for nil error")
            }
            continuation.resume(returning: result)
        }
    }
}
```

07 AsyncSequence与AsyncStream

本内容属于 Concurrency 的应用，为了能够利用 Concurrency 设计更加现代化的并发 API，Swift 5.5 同时推出了一些新的协议和类型，主要有 AsyncSequence 与 AsyncStream。

AsyncSequence

- Sequence（序列） 和 Collection（集合） 协议构成了 Swift 集合类型的基础，其中是 Sequence 协议是基础， Collection 协议继承自 Sequence 协议。之前学习的 Array、Set、Dictionary 又实现了 Collection 协议。
- AsyncSequence 表示异步序列，用于安全地处理多个异步的序列值。
- AsyncSequence 也可以使用 `map、filter、contains、reduce` 等高级操作。
- AsyncSequence 已经在 URL、URLSession 和 Notifications 等 API 中使用。

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    do {
                        try await self.asyncSequenceUsageURL()
                        // try await self.asyncSequenceUsageURLSession()
                        // try await self.asyncSequenceUsageNotification()
                    } catch {
                        print(error.localizedDescription)
                    }
                }
            }
    }

    // URL
    // API必须iOS15以上
    func asyncSequenceUsageURL() async throws {
        let url = URL(string: "https://www.abc.edu.cn")!
        // 边下边打印，所以会打印多次
        for try await line in url.lines {
            print(line, Thread.current)
        }
    }

    // URLSession
    // API必须iOS15以上
    func asyncSequenceUsageURLSession() async throws {
        let url = URL(string: "https://www.abc.edu.cn")!
        let request = URLRequest(url: url)
        let (bytes, _) = try await URLSession.shared.bytes(for: request)
        for try await byte in bytes {
            print(byte, Thread.current)
        }
    }

    // Notification
    // API必须iOS15以上
    func asyncSequenceUsageNotification() async throws {
        let notificationCenter = NotificationCenter.default
        let notifications = notificationCenter.notifications(named: .init(rawValue: "customName"))
        for await notification in notifications {
            print(notification, Thread.current)
        }
    }
}
```

AsyncStream

- 一个结构体，它遵守了 AsyncSequence 协议（类似于 Array 与 Sequence 的关系），表示一种有序的、异步生成的元素序列。
- 基本使用。

```
class AsyncFetcher {
    var cities = ["北京", "南京", "西安", "杭州", "广州"]

    // 使用Concurrency
    func getCities() async -> [String] {
        var result = [String]()

        while !cities.isEmpty {
            Thread.sleep(forTimeInterval: 1.0)
            result.append(cities.popLast() ?? "")
        }

        return result
    }
}

class AsyncStreamFetcher {
    var cities = ["北京", "南京", "西安", "杭州", "广州"]

    // 使用AsyncStream
    func getCities() -> AsyncStream<String> {
        AsyncStream(String.self) { continuation in
            // 内部需要使用 continuation.yield()产生值或使用 continuation.finish()结束
            Task {
                while !cities.isEmpty {
                    Thread.sleep(forTimeInterval: 1.0)
                    continuation.yield(cities.popLast() ?? "")
                }
                // 结束
                continuation.finish()
            }
        }
    }
}

struct ContentView: View {
    let one = AsyncFetcher()
    let two = AsyncStreamFetcher()

    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    let result = await one.getCities()
                    // 等待5秒，然后一次性返回整个数组
                    print(result)
                    print("获取了 \(result.count) 个数据")
                }

                Task {
                    var result = [String]()
                    // 遍历
                    for await city in two.getCities() {
                        // 1秒输出一个
                        print(city)

                        result.append(city)
                    }
                    // 调用了finish()以后才会往下执行
                    print("获取了 \(result.count) 个数据")
                }
            }
    }
}
```

- 扩展 Timer。

```
extension Timer {
    private static var count = 0

    static var stream: AsyncStream<Date> {
        AsyncStream(Date.self) { continuation in
            Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true) { _ in
                continuation.yield(Date())
```

```
                // 计数
                count += 1
                // 10次
                if count == 10 {
                    continuation.finish()
                }
            }
        }
    }
}

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .onAppear {
                Task {
                    print("Timer开始")
                    for try await time in Timer.stream {
                        print(time) // 不停打印当前时间
                    }
                    print("Timer结束")
                }
            }
    }
}
```

总结

Apple 已经在 URLSession、Notifications、CoreData、HealthKit 等多个模块和框架中推出了支持 Concurrency 的 API，就像当年推出 Swift 替换 Objective-C、SwiftUI 替换 UIKit 中的 UI 一样，这是一种不可逆的趋势，相信随着时间的推移，会有越来越多的 API 跟进，Concurrency 必将逐步替换现有的多线程的并发编程方式，成为 iOS 开发中必知必会的重要知识点。

源代码

源代码 GitHub 地址

视频教程

作者录制的系列视频教程已经发布至腾讯课堂，详情请查看视频教程。