Async & Await In Swift 5.5

彭玉堂——阿里巴巴淘系技术部

自我介绍

- 彭玉堂，阿里巴巴淘宝技术基础平台部无线开发专家，2009 年毕业于北京航空航天大学信息与计算科学专业，2013 年加入阿里巴巴，参与了虾米音乐、手机天猫、手机淘宝几个移动客户端的架构和性能保障等相关工作。

目录

通过回调的方式编写异步代码

```
func processImageData2a(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, error in
                    guard let imageResult = imageResult else {
                        completionBlock(nil, error)
                        return
                    }
                    completionBlock(imageResult)
                }
            }
        }
    }
}
```

通过回调的方式编写异步代码

- 嵌套多层，代码可读性差
- 错误处理复杂
- 条件执行困难，并且容易出错
- 容易犯低级错误 (忘记回调或者回调后忘记返回等)

其他语言的异步编程方式——Kotlin

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}

fun preparePostAsync(callback: (Token) -> Unit) {
    // 发起请求并立即返回
    // 设置稍后调用的回调
}
```

Kotlin中的回调方式

```
fun postItem(item: Item) {
    launch {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

suspend fun preparePost(): Token {
    // 发起请求并挂起该协程
    return suspendCoroutine { /* ... */ }
}
```

Kotlin中的协程

其他语言的异步编程方式——node.js

```
function requestWithRetry(url, retryCount) {
    if (retryCount) {
        return new Promise((resolve, reject) => {
            const timeout = Math.pow(2, retryCount);

            setTimeout(() => {
                console.log("Waiting", timeout, "ms");
                _requestWithRetry(url, retryCount)
                .then(resolve)
                .catch(reject);
            }, timeout);
        });
    } else {
        return _requestWithRetry(url, 0);
    }
}

function _requestWithRetry(url, retryCount) {
    return request(url, retryCount)
    .catch((err) => {
        if (err.statusCode && err.statusCode >= 500) {
            console.log("Retrying", err.message, retryCount);
            return requestWithRetry(url, ++retryCount);
        }
        throw err;
    });
}
```

node.js中的回调方式

```
async function requestWithRetry(url) {
    const MAX_RETRIES = 10;
    for (let i = 0; i <= MAX_RETRIES; i++) {
        try {
            return await request(url);
        } catch (err) {
            const timeout = Math.pow(2, i);
            console.log("Waiting", timeout, "ms");
            await wait(timeout);
            console.log("Retrying", err.message, i);
        }
    }
}
```

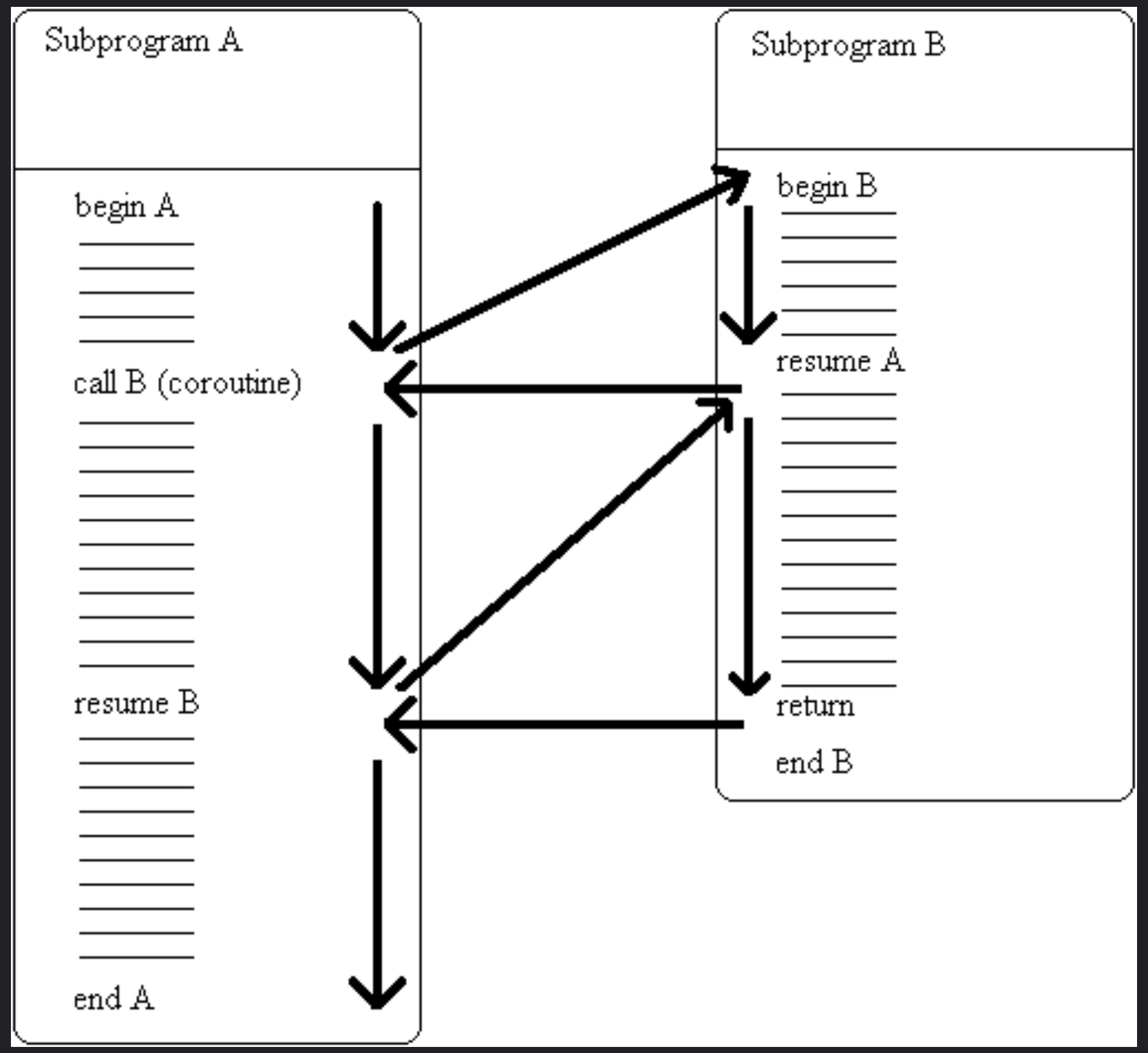
node.js中的协程

协程不仅打开了异步编程的大门，还提供了大量其他的可能性

什么是协程?

协程是一种在非抢占式多任务场景下生成可以
在特定位置挂起和恢复执行入口的程序组件

协程的实现原理



https://commons.wikimedia.org/wiki/File:Coroutine.png

协程的特征

非抢占式

- 无需系统调用
- 协程调度是线程安全，无需锁

挂起执行

- 保存寄存器和栈
- 不影响其他协程执行

恢复执行

- 恢复之前的寄存器和栈
- 无缝切换回之前的执行逻辑

Swift 5.5 为我们带来了协程模型——async/await

async/await in Swift

```
func load() async {
        var drinks = await store.load()

        // Drop old drinks
        drinks.removeOutdatedDrinks()

        // Assign loaded drinks to model
        currentDrinks = drinks
        await drinksUpdated()

        // Load new data from HealthKit.
        guard await healthKitController.requestAuthorization() else {
            logger.debug("Unable to authorize HealthKit.")
            return
        }

        await self.healthKitController.loadNewDataFromHealthKit()
    }
```

async/await 可以通过顺序书写的方式编写异步代码

async/await中的错误处理

```
extension HKHealthStore {

    @available(iOS 15.0, *)
    public func requestAuthorization(toShare typesToShare: Set<HKSampleType>, read
typesToRead: Set<HKObjectType>) async throws
}
```

```
public func requestAuthorization() async -> Bool {
        guard isAvailable else { return false }

        do {
            try await store.requestAuthorization(toShare: types, read: types)
            self.isAuthorized = true
            return true
        } catch let error {
            self.logger.error("An error occurred while requesting HealthKit Authorization: \
(error.localizedDescription)")
            return false
        }
    }
```

async/await 中通过异常来处理错误

async/await 使用注意事项

```
public func addDrink(mgCaffeine: Double, onDate date: Date) {
        logger.debug("Adding a drink.")

        // Create a local array to hold the changes.
        var drinks = currentDrinks

        // Create a new drink and add it to the array.
        let drink = Drink(mgCaffeine: mgCaffeine, onDate: date)
        drinks.append(drink)

        // Get rid of any drinks that are 24 hours old.
        drinks.removeOutdatedDrinks()

        currentDrinks = drinks

        // Save drink information to HealthKit.
        async {
            await self.healthKitController.save(drink: drink)
            await self.drinksUpdated()
        }
    }
```

同步函数不能简单的直接调用async 函数，编译器会报错，需要使用async 创建一个异步任务

async/await 使用注意事项

```
        async {
                    // Check for updates from HealthKit.
                    let model = CoffeeData.shared

                    let success = await
model.healthKitController.loadNewDataFromHealthKit()

                    if success {
                        // Schedule the next background update.
                        scheduleBackgroundRefreshTasks()
                        self.logger.debug("Background Task Completed Successfully!")
                    }

                    // Mark the task as ended, and request an updated snapshot, if
necessary.
                    backgroundTask.setTaskCompletedWithSnapshot(success)
                }
```

async函数可以调用其他 async函数，也可以直接调用普通的同步函数

async/await 使用注意事项

```swift
func doSomething(completionHandler: ((String) -> Void)? = nil) {
    print("test1")
}

func doSomething() async -> String {
    print("test2")
    return ""
}

override func viewDidLoad() {
    super.viewDidLoad()

    doSomething()      // print test1

    async {
        await doSomething()  // print test2
    }
}
```

如果sync函数和async函数可以用同样的方式调用，编译器会根据当前的上下文环境决定调用哪个函数

async/await 使用注意事项

```
protocol Asynchronous {
  func f() async
}

protocol Synchronous {
  func g()
}

struct S1: Asynchronous {
  func f() async { } // okay, exactly matches
}

struct S2: Asynchronous {
  func f() { } // okay, synchronous function satisfying async requirement
}

struct S3: Synchronous {
  func g() { } // okay, exactly matches
}

struct S4: Synchronous {
  func g() async { } // error: cannot satisfy synchronous requirement with an async function
}
```

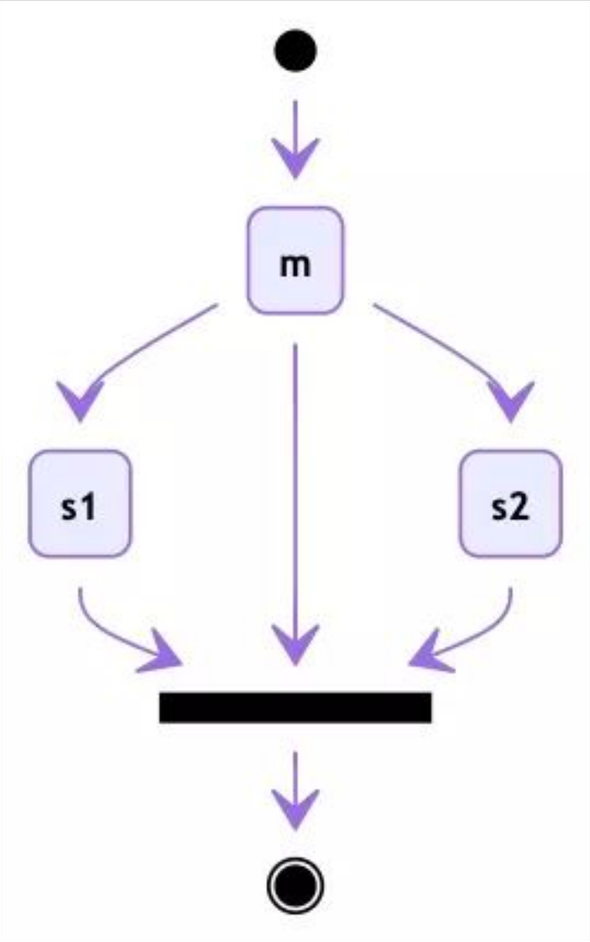
一个async的protocol 可以由sync或者async的函数满足，但是sync的protocol无法被async函数满足

一个做晚餐的案例

```swift
func chopVegetables() async throws -> [Vegetable] { ... }
func marinateMeat() async -> Meat { ... }
func preheatOven(temperature: Double) async throws -> Oven { ... }

// ...

func makeDinner() async throws -> Meal {
  let veggies = try await chopVegetables()  // 处理蔬菜
  let meat = await marinateMeat()           // 腌制肉
  let oven = try await preheatOven(temperature: 350) //预热烤箱

  let dish = Dish(ingredients: [veggies, meat])   // 把蔬菜和肉装盘
  return try await oven.cook(dish, duration: .hours(3))  // 用烤箱做出晚餐
}
```

- 上面处理蔬菜、腌制肉、预热烤箱等都是异步执行的
- 但是上述三个步骤仍然是串行执行的，这使得做晚餐的时间变长了
- 为了让晚餐准备时间变短，我们需要让处理蔬菜、腌制肉、预热烤箱几个步骤并发执行

结构化并发



结构化并发是一种编程范式，旨在通过使用结构化的并发编程方法来提高计算机程序的清晰度、质量和研发效能。

核心理念是通过具有明确入口和出口点并确保所有生成的子任务在退出前完成的控制流构造来封装并发执行任务（这里包括内核和用户线程和进程）。这种封装允许并发任务中的错误传播到控制结构的父作用域，并由每种特定计算机语言的本机错误处理机制进行管理。尽管存在并发性，但它允许控制流通过源代码的结构保持显而易见。为了有效，这个模型必须在程序的所有级别一致地应用——否则并发任务可能会泄漏、成为孤立的或无法正确传播运行时错误。（来自维基百科）

参考自: https://zhuanlan.zhihu.com/p/108759542

任务集合和子任务

```
func makeDinner() async throws -> Meal {
 var veggies: [Vegetable]?
  var meat: Meat?
  var oven: Oven?
  enum CookingStep {
    case veggies([Vegetable])
    case meat(Meat)
    case oven(Oven)
  }
 try await withThrowingTaskGroup(of: CookingStep.self)
{ group in
    group.async {
      try await .veggies(chopVegetables())
    }
    group.async {
      await .meat(marinateMeat())
    }
    group.async {
      try await .oven(preheatOven(temperature: 350))
    }

    for try await finishedStep in group {
      switch finishedStep {
        case .veggies(let v): veggies = v
        case .meat(let m): meat = m
        case .oven(let o): oven = o
      }
    }
  }
  let dish = Dish(ingredients: [veggies!, meat!])
  return try await oven!.cook(dish, duration: .hours(3))
}
```

- Task group 定义了一个生命周期，可以在其中以编程方式创建新的子任务。与所有子任务一样，group 生命周期内的子任务必须在生命周期结束时完成，如果父任务退出并抛出错误，则将首先隐式取消子任务。
- group.async 中不能直接修改捕获的父生命周期中的变量

Async let bindings

```swift
func makeDinner() async throws -> Meal {
  async let veggies = chopVegetables()
  async let meat = marinateMeat()
  async let oven = preheatOven(temperature: 350)

  let dish = Dish(ingredients: await [try veggies, meat])
  return try await oven.cook(dish, duration: .hours(3))
}
```

- 虽然 Task Group 非常强大，但它们在使用上还是比较麻烦的
- Swift 5.5 引入了 async let , 它可以用一种更加简单的方式来创建子任务并等待子任务的执行结果

Async let bindings

- Async let 只能在async function 或者 async closure 中使用

```
func greet() async -> String { "hi" }

func asynchronous() async {
  async let hello = greet()
  // ...
  await hello
}
```

- Async let 不允许在顶层代码、同步函数中使用

```
async let top = ... // error: 'async let' in a function that does not support concurrency

func sync() { // note: add 'async' to function 'sync()' to make it asynchronous
  async let x = ... // error: 'async let' in a function that does not support concurrency
}

func syncMe(later: () -> String) { ... }
syncMe {
  async let x = ... // error: invalid conversion from 'async' function of type '() async -> String' to synchronous function type '() -> Stri
}
```

- Async let 可以和元组等结合使用

```
func left() async -> String { "l" }
func right() async -> String { "r" }

async let (l, r) = (left(), right())

await l // at this point `r` is also assumed awaited-on
```

Async let bindings

- 在 async 函数中定义了Async let, 但是并未await，那会出现隐式的await调用

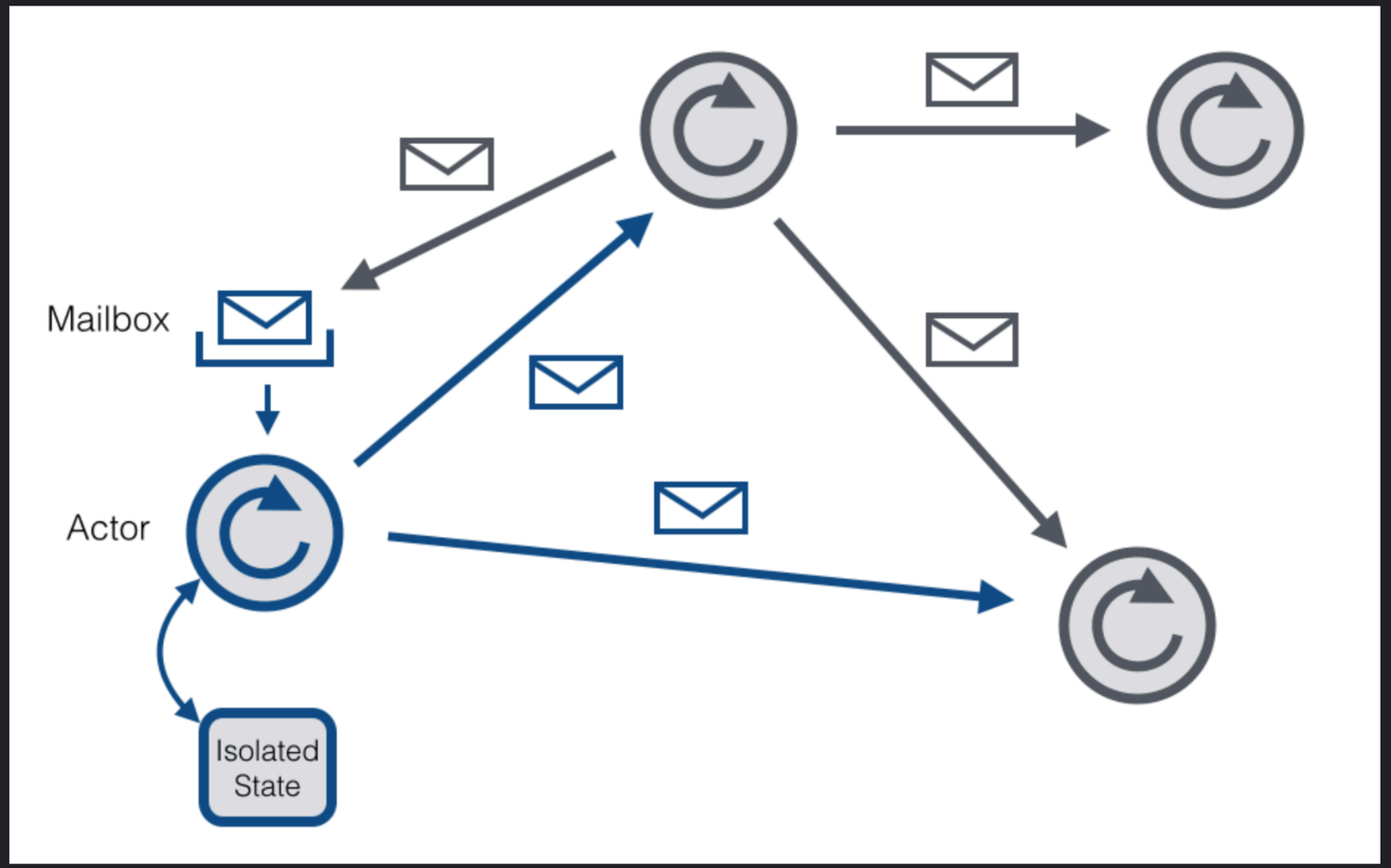
```
func go() async {
  async let f = fast() // 300ms
  async let s = slow() // 3seconds
  return "nevermind..."
  // implicitly: cancels f
  // implicitly: cancels s
  // implicitly: await f
  // implicitly: await s
}
```

Swift 5.5 中的并发模型旨在提供一个安全的编程模型，可以静态检测数据竞争和其他常见的并发错误。结构化并发为函数和闭包提供多线程竞争安全性保障。该模型适用于许多常见的设计模式，包括并行映射和并发回调模式等，但仅限于处理由闭包捕获的状态。

Swift 中的类提供了一种机制来声明多线程共享的可变状态。然而，众所周知，类很难在并发程序中正确使用，需要通过一些类似锁、信号量等同步机制以避免数据竞争。最理想的情况是既提供使用共享可变状态的能力，同时仍然能够对数据竞争和其他常见并发错误的静态检测。

为此 Swift 5.5 引入了新的并发模型——Actor，它通过数据隔离保护内部的数据，确保在给定时间只有一个线程可以访问该数据

Actors



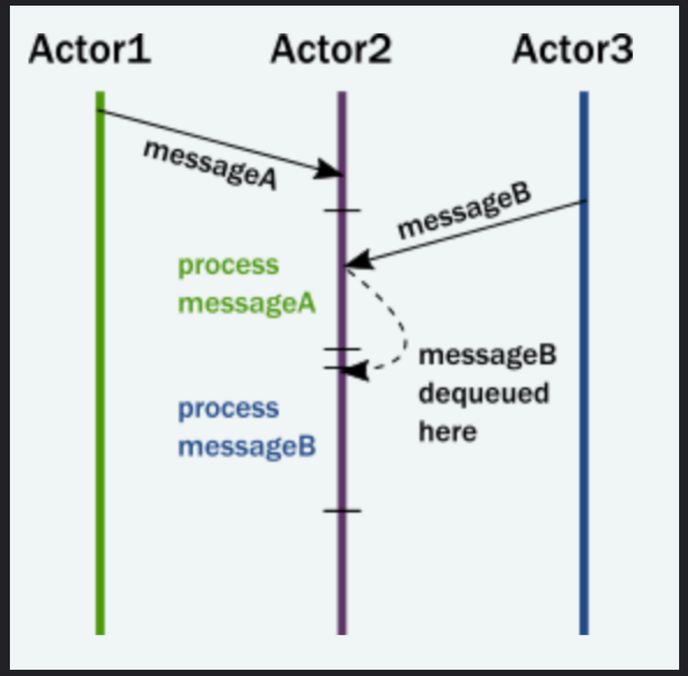
- 状态私有
- 只能接收和处理消息
- 每个actor一次执行一个消息

- mailbox: 存储message的队列
- Isolated State: actor的状态，内部变量等
- message: 类似于OOP的方法调用的参数

参考自: https://cwiki.apache.org/confluence/display/FLINK/Akka+and+Actors

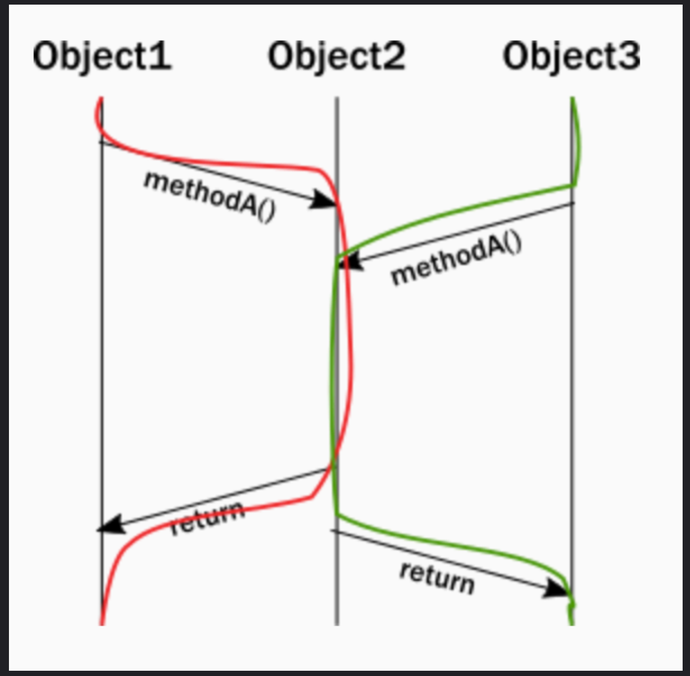
Actors

actor



- actor状态私有，不可共享
- actor通过消息进行通信，不需要传递执行线程
- actor内部串行处理消息，可以保障内部状态和变量的正确性

传统OOP调用



- OOP状态可共享，外部可以访问和改变
- OOP方法调用会传递执行线程
- OOP方法调用不保证执行线程，因此需要使用锁来确保内部状态和变量的正确性

参考自：https://doc.akka.io/docs/akka/current/guide/actors-intro.html

参考自：https://doc.akka.io/docs/akka/current/guide/actors-motivation.html

Actor

```
class RiskyCollector {
    var deck: Set<String>

    init(deck: Set<String>) {
        self.deck = deck
    }

    func send(card selected: String, to person: RiskyCollector) -> Bool {
        guard deck.contains(selected) else { return false }

        deck.remove(selected)
        person.transfer(card: selected)
        return true
    }

    func transfer(card: String) {
        deck.insert(card)
    }
}
```

- 在多线程环境下RiskyCollector类的访问并不安全，可能会出现访问冲突

Actor

```
actor SafeCollector {
    var deck: Set<String>

    init(deck: Set<String>) {
        self.deck = deck
    }

    func send(card selected: String, to person: SafeCollector) async -> Bool {
        guard deck.contains(selected) else { return false }

        deck.remove(selected)
        await person.transfer(card: selected)
        return true
    }

    func transfer(card: String) {
        deck.insert(card)
    }
}
```

- 在Swift 5.5中 Actor 使用 actor 关键字创建
- send() 方法标示为async，该方法需要等待person的transfer完成
- 尽管transfer并不是async方法, 但是在调用其他对象的方法的时候仍然需要通过await进行调用

Actor isolation

```swift
actor BankAccount {
  let accountNumber: Int
  var balance: Double

  init(accountNumber: Int, initialDeposit: Double) {
    self.accountNumber = accountNumber
    self.balance = initialDeposit
  }

  enum BankError: Error {
    case insufficientFunds
  }

  func transfer(amount: Double, to other: BankAccount) throws {
    if amount > balance {
      throw BankError.insufficientFunds
    }

    print("Transferring \(amount) from \(accountNumber) to \(other.accountNumber)")

    balance = balance - amount
    other.balance = other.balance + amount  // error: actor-isolated property 'balance' can only be referenced on 'self'
  }
}
```

- 如果BankAccount 是一个 class ，那上述代码是没有任何编译问题的，但是会出现多线程问题
- 在actor 中尝试访问other.balance 会出发编译错误，因为balance只能通过self访问

Actor isolation

```
extension BankAccount {
  func deposit(amount: Double) async {
    assert(amount >= 0)
    balance = balance + amount
  }
  func transfer(amount: Double, to other: BankAccount) async throws {
    assert(amount > 0)

    if amount > balance {
      throw BankError.insufficientFunds
    }

    print("Transferring \(amount) from \(accountNumber) to \(other.accountNumber)")

    // Safe: this operation is the only one that has access to the actor's isolated
    // 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 placed in the `other` actor's mailbox; when
    // that actor retrieves the operation from its mailbox to execute it, the
    // other account's balance will get updated.
    await other.deposit(amount: amount)
  }
}
```

可重入的Actor

```
actor Person {
  let friend: Friend

  // actor-isolated opinion
  var opinion: Judgment = .noIdea

  func thinkOfGoodIdea() async -> Decision {
    opinion = .goodIdea                          // <1>
    await friend.tell(opinion, heldBy: self)  // <2>
    return opinion // 😐                          // <3>
  }

  func thinkOfBadIdea() async -> Decision {
    opinion = .badIdea                          // <4>
    await friend.tell(opinion, heldBy: self) // <5>
    return opinion // 🤨                         // <6>
  }
}

let goodThink = detach { await person.thinkOfGoodIdea() }  // runs async
let badThink = detach { await person.thinkOfBadIdea() } // runs async

let shouldBeGood = await goodThink.get()
let shouldBeBad = await badThink.get()

await shouldBeGood // could be .goodIdea or .badIdea ☠️
await shouldBeBad
```

- Actor的函数是可重入的, 当一个actor的函数挂起, 重入机制可以允许挂起恢复前其他工作继续执行，这个就是交替执行，重入可以解决死锁的问题

Actor with Objc

```
@objc actor MyActor {
    @objc func synchronous() { } // error: part of actor's isolation domain
    @objc func asynchronous() async { } // okay: asynchronous, exposed to Objective-C as a method that accepts a completion handler
    @objc nonisolated func notIsolated() { } // okay: non-isolated
}
```

Global Actor

```
class ViewController {
    func refreshUI() -> Bool {
        guard Thread.isMainThread else {
            return false
        }

        print("updating ui…")
        return true
    }
}
```

在应用程序中，主线程通常负责执行主要事件处理循环，该循环处理来自各种来源的事件并将它们传递给应用程序代码。图形应用程序通常在主线程上传递用户交互事件（键盘按下、触摸交互），并且要求用户界面的任何状态更新也发生在主线程。

Global Actor

```swift
class ViewController {
    @MainActor func refreshUI() {
        print("updating ui…")
    }
}

@MainActor var globalTextSize: Int

@MainActor func increaseTextSize() {
  globalTextSize += 2   // okay:
}

func notOnTheMainActor() async {
  globalTextSize = 12  // error: globalTextSize is isolated to MainActor
  increaseTextSize()   // error: increaseTextSize is isolated to MainActor, cannot call synchronously
  await increaseTextSize() // okay: asynchronous call hops over to the main thread and executes there
}
```

- Global Actor 将actor 隔离的概念扩展到了全局状态，即使状态和函数分散在许多不同的模块中，Global Actor 可以再并发程序中安全地使用全局变量，例如 @MainActor 限制属性和方法只能在主线程访问

Actor 和 class 对比

- Actor 和 class 共同点有：
  - 两者都是引用类型，因此它们可用于共享状态。
  - 它们可以有方法、属性、初始值设定项和下标。
  - 它们可以实现协议。
  - 任何静态属性和方法在这两种类型中的行为都相同。
- 除了Actor isolation之外，actor 和 class 之间还有另外两个重要的区别：
  - Actor 目前不支持继承，这使得它们的初始化器更加简单——不需要方便的初始化器、覆盖、final 关键字等等。这在未来可能会改变。
  - 所有actor都隐含地遵守一个新的actor协议；没有其他具体类型可以使用它

Async sequence

```
struct Counter : AsyncSequence {
  let howHigh: Int

  struct AsyncIterator : AsyncIteratorProtocol {
    let howHigh: Int
    var current = 1
    mutating func next() async -> Int? {
      // We could use the `Task` API to check for cancellation here and return early.
      guard current <= howHigh else {
        return nil
      }

      let result = current
      current += 1
      return result
    }
  }

  func makeAsyncIterator() -> AsyncIterator {
    return AsyncIterator(howHigh: howHigh)
  }
}

for await i in Counter(howHigh: 3) {
  print(i)
}

/*
Prints the following, and finishes the loop:
1
2
3
*/
```

Readonly properties

```swift
enum FileError: Error {
    case missing, unreadable
}

struct BundleFile {
    let filename: String

    var contents: String {
        get async throws {
            guard let url = Bundle.main.url(forResource: filename, withExtension: nil) else {
                throw FileError.missing
            }

            do {
                return try String(contentsOf: url)
            } catch {
                throw FileError.unreadable
            }
        }
    }
}

func printHighScores() async throws {
    let file = BundleFile(filename: "highscores")
    try await print(file.contents)
}
```

- 在Swift 5.5 中只读计算属性也可以支持async了，可以按照上图所示代码使用

Continuations

```
func fetchLatestNews(completion: @escaping ([String]) -> Void) {
    DispatchQueue.main.async {
        completion(["Swift 5.5 release", "Apple acquires Apollo"])
    }
}
```

- 上述旧的异步代码如何转化为async/await可以使用的代码呢?

Continuations

```
func fetchLatestNews() async -> [String] {
    await withCheckedContinuation { continuation in
        fetchLatestNews { items in
            continuation.resume(returning: items)
        }
    }
}

func printNews() async {
    let items = await fetchLatestNews()

    for item in items {
        print(item)
    }
}
```

- Continuations 可以让开发者在旧的回调函数和async函数之间创建一个桥接，以便开发者将旧代码包装在更现代的 API 中。例如，withCheckedContinuation() 函数创建一个新的 continuation，它可以运行你想要的任何代码，然后在你准备好时调用 resume(returning:) 来返回一个值——这就是完成处理程序闭包的一部分。

Continuations

```swift
class MyOperation: Operation {
  let continuation: UnsafeContinuation<OperationResult, Never>
  var result: OperationResult?

  init(continuation: UnsafeContinuation<OperationResult, Never>) {
    self.continuation = continuation
  }

  /* rest of operation populates `result`... */

  override func finish() {
    continuation.resume(returning: result!)
  }
}

func doOperation() async -> OperationResult {
  return await withUnsafeContinuation { continuation in
    MyOperation(continuation: continuation).start()
  }
}
```

- Continuations 不仅可以直接用于桥接callback相关的旧代码，也可以作为变量保存下来，进行更加复杂的桥接操作，比如像delegate异步的桥接等

Continuations

```swift
func download(url: URL) async throws -> Data? {
  var urlSessionTask: URLSessionTask?

  return try Task.withCancellationHandler {
    urlSessionTask?.cancel()
  } operation: {
    let result: Data? = try await withUnsafeThrowingContinuation { continuation in
      urlSessionTask = URLSession.shared.dataTask(with: url) { data, _, error in
        if case (let cancelled as NSURLErrorCancelled)? = error {
          continuation.resume(returning: nil)
        } else if let error = error {
          continuation.resume(throwing: error)
        } else {
          continuation.resume(returning: data)
        }
      }
      urlSessionTask?.resume()
    }
    if let result = result {
      return result
    } else {
      Task.cancel()
      return nil
    }
  }
}
```

- 开发者也可以根据父任务的取消来取消桥接的任务，像上面我们使用Continuation来桥接了URLSession的请求，当task取消的时候，取消下载任务

总结

更新项	描述链接	适用系统
async/await	https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md	iOS15及以上
async sequences	https://github.com/apple/swift-evolution/blob/main/proposals/0298-asyncsequence.md	iOS15及以上
Effectful read-only properties	https://github.com/apple/swift-evolution/blob/main/proposals/0310-effectful-readonly-	iOS15及以上
Structured Concurrency	https://github.com/apple/swift-evolution/blob/main/proposals/0304-structured-concurrency.md	iOS15及以上
async let	https://github.com/apple/swift-evolution/blob/main/proposals/0317-async-let.md	iOS15及以上
Continuations for interfacing async tasks with synchronous code	https://github.com/apple/swift-evolution/blob/main/proposals/0300-continuation.md	iOS15及以上
Actors	https://github.com/apple/swift-evolution/blob/main/proposals/0306-actors.md	iOS15及以上
Global Actors	https://github.com/apple/swift-evolution/blob/main/proposals/0316-global-actors.md	iOS15及以上
Sendable and @Sendable closures	https://github.com/apple/swift-evolution/blob/main/proposals/0302-concurrent-value-and-	不限制
#if for postfix member expressions	https://github.com/apple/swift-evolution/blob/main/proposals/0308-postfix-if-config-	不限制
Codable synthesis for enums with associated values	https://github.com/apple/swift-evolution/blob/main/proposals/0295-codable-synthesis-for-	不限制
lazy now works in local contexts		不限制
Extend Property Wrappers to Function and Closure Parameters	https://github.com/apple/swift-evolution/blob/main/proposals/0293-extend-property-wrappers-	不限制
Extending Static Member Lookup in Generic Contexts	https://github.com/apple/swift-evolution/blob/main/proposals/0299-extend-generic-static-	不限制

Swift 5.5的重要更新以及适用的系统

总结

- Swift 5.5 带来了很多振奋人心的更新
  - Async/await 可以降低异步编程的复杂度，书写出更加简洁高效可读性强的异步代码逻辑
  - 结构化并发和Actor编程模型的引入可以让开发者写出更加安全的并发代码
  - Continuation可以让开发者快速将已有的代码迁移到新的异步特性中
- 但是让人失望的是Concurrency相关的更新只能在iOS15及以上的设备中运行
- 尽管此次更新的限制诸多，但是我仍然很看好Swift，相信未来Swift这么语言可以给我们带来更多的惊喜

参考资料

- https://github.com/twostraws/whats-new-in-swift-5-5
- https://github.com/apple/swift-evolution/blob/main/proposals/0300-continuation.md
- https://github.com/apple/swift-evolution/blob/main/proposals/0310-effectful-readonly-properties.md
- https://github.com/apple/swift-evolution/blob/main/proposals/0298-asyncsequence.md
- https://github.com/apple/swift-evolution/blob/main/proposals/0304-structured-concurrency.md#cancellation
- https://github.com/apple/swift-evolution/blob/main/proposals/0306-actors.md#protocol-conformance
- https://github.com/apple/swift-evolution/blob/main/proposals/0302-concurrent-value-and-concurrent-closures.md
- https://github.com/apple/swift-evolution/blob/main/proposals/0317-async-let.md
- https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md

Q & A