闲话 Swift 协程（6）：Actor 和属性隔离

bennyhuo
猿辅导 Android 工程师
关注他

> 本文转自 **Bennyhuo 的博客**
> 原文地址：bennyhuo.com/2022/02/12...

> 异步函数大多数情况下会并发地执行在不同的线程，那么线程安全怎么来保证？

- 闲话 Swift 协程（1）：Swift 协程长什么样?
- 闲话 Swift 协程（2）：将回调改写成 async 函数
- 闲话 Swift 协程（3）：在程序当中调用异步函数
- 闲话 Swift 协程（4）：TaskGroup 与结构化并发
- 闲话 Swift 协程（5）：Task 的取消
- 闲话 Swift 协程（6）：Actor 和属性隔离
- 闲话 Swift 协程（7）：GlobalActor 和异步函数的调度
- 闲话 Swift 协程（8）：TaskLocal
- 闲话 Swift 协程（9）：异步函数与其他语言的互调用

什么是 actor

Swift 为了解决线程安全的问题，引入了一个非常有用的概念叫做 actor。Actor 模型是计算机科学领域的一个用于并行计算的数学模型，其中 actor 是模型当中的基本计算单元。

在 Swift 当中，actor 包含 state、mailbox、executor 三个重要的组成部分，其中：

- state 就是 actor 当中存储的值，它是受到 actor 保护的，访问时会有一些限制以避免数据竞争（data race）。
- mailbox 字面意思是邮箱的意思，在这里我们可以理解成一个消息队列。外部对于 actor 的可变状态的访问需要发送一个异步消息到 mailbox 当中，actor 的 executor 会串行地执行 mailbox 当中的消息以确保 state 是线程安全的。
- executor，actor 的逻辑（包括状态修改、访问等）执行所在的执行器。

下面我们给出一个简单的例子：

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

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

我们定义了一个 actor 叫做 BankAccount（这个例子来自 Swift 的 proposal），不难看出 actor 在形式上与 class 很像，不仅如此，actor 也能像它们一样定义扩展，声明泛型，实现协议等等。

Actor 实际上也是引用类型，所以用起来也更像是确保了数据线程安全的 class，例如：

```
let account = BankAccount(accountNumber: 1234, initialDeposit: 1000)
```

知乎 首发于 bennyhuo

我们可以用类似于 class 的方式来构造 actor，并且创建多个变量指向同一个实例，以及使用 === 来判断是否指向同一个实例。程序运行时，我们也可以看到 account 和 account2 指向的地址是相同的：

```
account = {coroutines.BankAccount} 0x600002c04000 <BankAccount: 0x600002c04000>
account2 = {coroutines.BankAccount} 0x600002c04000 <BankAccount: 0x600002c04000>
```

Actor 的属性隔离

为了描述存钱这个行为，我们可能希望在外部修改 balance 的值，如果是 struct 或者 class，这个行为并不麻烦，但对于 actor 来讲，这个修改可能是不安全的，因此不被允许。

那怎么办？我们前面提到修改 actor 的状态需要发邮件，actor 会在收到邮件之后一个一个处理并异步返回给你结果（有没有一种给领导发邮件审批的感觉），这个叫做 actor-isolated（即属性隔离）。

所以我们打开 outlook 发个邮件？当然不是，开个小玩笑。Swift 的 actor 已经把”发邮件“这个操作设计得非常简洁了，简单说就是两点：

1. actor 的可变状态只能在 actor 内部被修改（隔离嘛）
2. 发邮件其实就是一个异步函数调用的过程

所以我们需要给 BankAccount 定义一个存钱的函数来完成对 balance 的修改：

```
extension BankAccount {
    func deposit(amount: Double) async {
        assert(amount >= 0)
        balance = balance + amount
    }
}
```

我们把它定义在扩展当中，接下来就可以愉快得存钱了：

```
let account = BankAccount(accountNumber: 1234, initialDeposit: 1000)

print(account.accountNumber) // OK，不可变状态
print(await account.balance) // 可变状态的访问需要使用 await

await account.deposit(amount: 90) // actor 的函数调用需要 await
print(await account.balance)
```

这个例子当中有几个细节请大家留意： 1. accountNumber 可以直接访问，因为它不可变。不可变就意味着不存在线程安全问题。 2. 对可变的状态 balance 的访问以及对函数 deposit 的调用都是异步调用，需要用 await，因为这个访问实际上封装了发邮件的过程。

接下来再给大家看一下转账的实现：

```
extension BankAccount {
  enum BankError: Error {
    case insufficientFunds
  }

  func transfer(amount: Double, to other: BankAccount) async throws {
    assert(amount > 0)

    if amount > balance {
```

```
        // other.balance = other.balance + amount 错误示例
        await other.deposit(amount: amount) // OK
    }
}
```

函数 transfer 是 BankAccount 自己的函数，修改自己 balance 的值自然没有什么问题。但修改 other 这个 BankAccount 实例的 balance 的值却是不行的，因为 tranfer 函数执行时实际上是 self 这个实例在处理自己的邮件，这里面如果偷偷修改了 other 的 balance 的值就可能导致 other 的状态出现问题（试想一下你处理自己的邮件的时候偷偷把领导的邮件给删了，看他发现了之后骂不骂你）。

这个例子告诉我们，actor 的状态只能在自己实例的函数内部修改，而不能跨实例修改。

外部函数修改 actor 的状态

前面我们反复提到 actor 的状态只能在自己的函数内部修改，是因为 actor 的函数的调用是在对应的 executor 上安全地执行的。如果外部的函数也能够满足这个调用条件，那么理论上也是安全的。

Swift 提供了 actor-isolated paramters 这样的特性，字面意思即满足 actor 状态隔离的参数，如果我们在定义外部函数时将需要访问的 actor 类型的参数声明为 isolated，那么我们就可以在函数内部修改这个 actor 的状态了。

基于这一点，我们也可以把 deposit 函数定义成顶级函数：

```
func deposit(amount: Double, to account: isolated BankAccount) {
    assert(amount >= 0)
    account.balance = account.balance + amount
}
```

注意到参数 account 的类型被关键字 isolated 修饰，表明函数 deposit 的调用需要保证 account 的状态修改安全。不难想到，对于这个函数的调用，我们需要使用 await：

```
await deposit(amount: 1000, to: account)
```

显然，这里的 isolated 参数不能有多个（至少现在是这样），不然在实现起来会比较麻烦。

声明不需要隔离的属性或函数

Actor 的属性默认都是需要被隔离保护的，但也有一些属性可能并不需要被保护，例如我们前面提到的不可变的状态。Swift 允许为 actor 声明不需要隔离的属性：

```
extension BankAccount : CustomStringConvertible {
    nonisolated var description: String {
        "Bank account #\(accountNumber)"
    }
}
```

注意到 description 被声明为 nonisolated，这样对于它的访问就不会受到 balance 那么多的限制了。

nonisolated 同样可以用来修饰函数，但这样的函数就不能直接访问被隔离的状态了，只能像外部函数一样使用 await 来异步访问。

这个特性在 Actor 实现 Protocol 的时候也显得非常有用，例如：

知乎 首发于 bennyhuo

```
        lhs.accountNumber == rhs.accountNumber
    }

    nonisolated func hash(into hasher: inout Hasher) {
        hasher.combine(accountNumber)
    }

    nonisolated var hashValue: Int {
        get {
            accountNumber.hashValue
        }
    }
}
```

如果不加 nonisolated，编译器会给出如下提示：

顺便提一句，在早期的提案当中，你可能会见到 @actorIndependent，它后来被重命名为 nonisolated，这样在语法上与 nonmutating 也更加一致。

Actor 与 @Sendable

在介绍协程的过程中，我们见过很多函数的闭包都被声明为 `@Sendable`，例如：

```
public func withTaskCancellationHandler<T>(
    operation: () async throws -> T,
    onCancel handler: @Sendable () -> Void
) async rethrows -> T
```

其中 onCancel 就被声明为 @Sendable，这表明只有实现了 `Sendable` 协议的类型实例才能被这个闭包所捕获。

Actor 天生就是线程安全的，因此也是符合 Sendable 协议的。实际上 Swift 的每一个 actor 类型都隐式地实现了一个叫做 `Actor` 的协议，而这个协议也正实现了 `Sendable` 协议。

我们看一下 `Actor` 的定义：

```
public protocol Actor : AnyObject, Sendable {
    nonisolated var unownedExecutor: _Concurrency.UnownedSerialExecutor { get }
}
```

除了定义了调度器之外，它也继承了 Sendable 协议。因此如果大家遇到 @Sendable 闭包需要捕获变量的问题，不妨试一试使用 Actor 来做一层封装。

顺便提一句，actor 的调度器目前主要由编译器提供默认的实现。官方目前对于自定义调度器的途径还没有给出明确的支持，不过我们将在下一篇文章当中详细探索一下调度器的使用。

小结

本文我们主要介绍了 Swift 协程当中的 actor 的基本用法，并重点对属性隔离做了详细介绍。

有关 actor 的调度器的内容，我们将在下一篇文章当中详细介绍。

霍丙乾 bennyhuo，Kotlin 布道师，Google 认证 Kotlin 开发专家（Kotlin GDE）；**《深入理解 Kotlin 协程》** 作者（机械工业出版社，2020.6）；前腾讯高级工程师，现就职于猿辅导

- GitHub：github.com/bennyhuo
- 博客：bennyhuo.com
- bilibili：**bennyhuo不是算命的**
- 微信公众号：**bennyhuo**

发布于 2022-02-28 20:44

iOS Swift 语言 协程