Swift 并发初步

由 王巍 (onevcat) 发布于 2021年7月1日 • 最后更新: 2021年9月29日

> 本文是我的新书《Swift 异步和并发》中第一章内容，主要从概览的方向上介绍了 Swift 5.5 中引入的 Swift 并发特性的使用方法。如果你对学习 Swift 并发有兴趣，也许可以作为参考读物。

> 你可以在这里找到本文中的参考代码。在本文写作的 Xcode 13 beta 2 环境下，你需要额外安装最新的 Swift 5.5 toolchain 来运行这些代码。

虽然可能你已经跃跃欲试，想要创建第一个 Swift 的并发程序，但是“名不正则言不顺”。在实际进入代码之前，作为全书开头，我还是想先对几个重要的相关概念进行说明。这样在今后本书中，当我们提起 Swift 异步和并发时，对具体它指代了什么内容，能够取得统一的认识。本章后半部分，我们会实际着手写一些 Swift 并发代码，来描述整套体系的基本构成和工作流程。

一些基本概念

同步和异步

在我们说到线程的执行方式时，同步 (synchronous) 和异步 (asynchronous) 是这个话题中最基本的一组概念。**同步操作**意味着在操作完成之前，运行这个操作的线程都将被占用，直到函数最终被抛出或者返回。Swift 5.5 之前，所有的函数都是同步函数，我们简单地使用 `func` 关键字来声明这样一个同步函数：

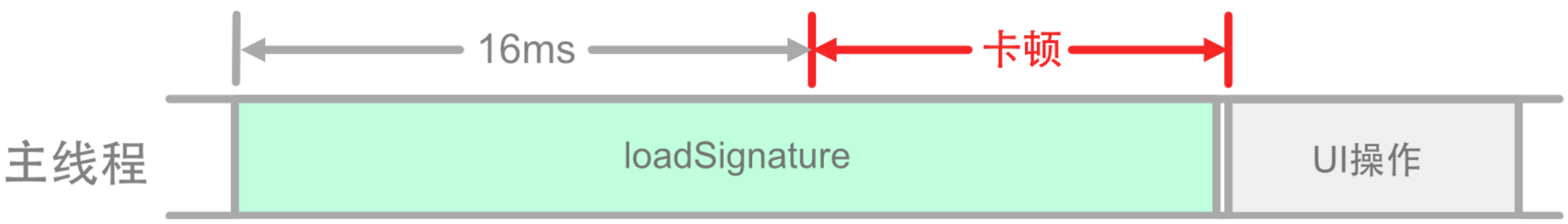
```
1  var results: [String] = []
2  func addAppending(_ value: String, to string: String) {
3      results.append(value.appending(string))
4  }
```

`addAppending` 是一个同步函数，在它返回之前，运行它的线程将无法执行其他操作，或者说它不能被用来运行其他函数，必须等待当前函数执行完成后这个线程才能做其他事情。

在 iOS 开发中，我们使用的 UI 开发框架，也就是 UIKit 或者 SwiftUI，不是线程安全的：对用户输入的处理和 UI 的绘制，必须在与主线程绑定的 main runloop 中进行。假设我们希望用户界面以每秒 60 帧的速率运行，那么主线程中每两次绘制之间，所能允许的处理时间最多只有 16 毫秒 (1 / 60s)。当主线程中要同步处理的其他操作耗时很少时 (比如我们的 `addAppending`，可能耗时只有几十纳秒)，这不会造成什么问题。但是，如果这个同步操作耗时过长的话，主线程将被阻塞。它不能接受用户输入，也无法向 GPU 提交请求去绘制新的 UI，这将导致用户界面掉帧甚至卡死。这种“长耗时”的操作，其实是很常见的：比如从网络请求中获取数据，从磁盘加载一个大文件，或者进行某些非常复杂的加解密运算等。

下面的 `loadSignature` 从某个网络 URL 读取字符串：如果这个操作发生在主线程，且耗时超过 16ms (这是很可能发生的，因为通过握手协议建立网络连接，以及接收数据，都是一系列复杂操作)，那么主线程将无法处理其他任何操作，UI 将不会刷新。

```swift
// 从网络读取一个字符串
func loadSignature() throws -> String? {
  // someURL 是远程 URL，比如 https://example.com
  let data = try Data(contentsOf: someURL)
  return String(data: data, encoding: .utf8)
}
```



`loadSignature` 最终的耗时超过 16 ms，对 UI 的刷新或操作的处理不得不被延后。在用户观感上，将表现为掉帧或者整个界面卡住。这是客户端开发中绝对需要避免的问题之一。

Swift 5.5 之前，要解决这个问题，最常见的做法是将耗时的同步操作转换为**异步操作**：把实际长时间执行的任务放到另外的线程(或者叫做后台线程)运行，然后在操作结束时提供运行在主线程的回调，以供 UI 操作之用：

```swift
func loadSignature(
  _ completion: @escaping (String?, Error?) -> Void
)
{
  DispatchQueue.global().async {
    do {
      let d = try Data(contentsOf: someURL)
      DispatchQueue.main.async {
        completion(String(data: d, encoding: .utf8), nil)
      }
    } catch {
      DispatchQueue.main.async {
        completion(nil, error)
      }
    }
  }
}
```

`DispatchQueue.global` 负责将任务添加到全局后台派发队列。在底层，GCD 库 (Grand Central Dispatch) 会进行线程调度，为实际耗时繁重的 `Data.init(contentsOf:)` 分配合适的线程。耗时任务在主线程外进行处理，完成后再由 `DispatchQueue.main` 派发回主线程，并按照结果调用 `completion` 回调方法。这样一来，主线程不再承担耗时任务，UI 刷新和用户事件处理可以得到保障。

异步操作虽然可以避免卡顿，但是使用起来存在不少问题，最主要包括：

- 错误处理隐藏在回调函数的参数中，无法用 `throw` 的方式明确地告知并强制调用侧去进行错误处理。
- 对回调函数的调用没有编译器保证，开发者可能会忘记调用 `completion` ，或者多次调用 `completion` 。
- 通过 `DispatchQueue` 进行线程调度很快会使代码复杂化。特别是如果线程调度的操作被隐藏在被调用的方法中的时候，不查看源码的话，在 (调用侧的) 回调函数中，几乎无法确定代码当前运行的线程状态。
- 对于正在执行的任务，没有很好的取消机制。

除此之外，还有其他一些没有列举的问题。它们都可能成为我们程序中潜在 bug 的温床，在之后关于异步函数的章节里，我们会再回顾这个例子，并仔细探讨这些问题的细节。

需要进行说明的是，虽然我们将运行在后台线程加载数据的行为称为**异步操作**，但是接受回调函数作为参数的 `loadSignature(_:)` 方法，其本身依然是一个**同步函数**。这个方法在返回前仍旧会占据主线程，只不过它现在的执行时间非常短，UI 相关的操作不再受影响。

Swift 5.5 之前，Swift 语言中并没有真正异步函数的概念，我们稍后会看到使用 `async` 修饰的异步函数是如何简化上面的代码的。

串行和并行

另外一组重要的概念是串行和并行。对于通过同步方法执行的同步操作来说，这些操作一定是以串行方式在同一线程中发生的。“做完一件事，然后再进行下一件事”，是最常见的、也是我们人类最容易理解的代码执行方式：

```
if let signature = try loadSignature() {
  addAppending(signature, to: "some data")
}
print(results)
```

`loadSignature`， `addAppending` 和 `print` 被顺次调用，它们在同一线程中按严格的先后顺序发生。这种执行方式，我们将它称为**串行 (serial)**。

同步方法执行的同步操作，是串行的充分但非必要条件。异步操作也可能会以串行方式执行。假设除了 `loadSignature(_:)` 以外，我们还有一个从数据库里读取一系列数据的函数，它使用类似的方法，把具体工作放到其他线程异步执行：

```
func loadFromDatabase(
  _ completion: @escaping ([String]?, Error?) -> Void
)
{
  // ...
}
```

如果我们先从数据库中读取数据，在完成后再使用 `loadSignature` 从网络获取签名，最后将签名附加到每一条数据库中取出的字符串上，可以这么写：

```
loadFromDatabase { (strings, error) in
  if let strings = strings {
    loadSignature { signature, error in
      if let signature = signature {
        strings.forEach {
                strings.forEach {
        strings.forEach {
          addAppending(signature, to: $0)
        }
      } else {
        print("Error")
      }
    }
  } else {
    print("Error.")
  }
}
```

虽然这些操作是**异步**的，但是它们 (从数据库读取 `[String]`，从网络下载签名，最后将签名添加到每条数据中) 依然是**串行**的，加载签名必定发生在读取数据库完成之后，而最后的 `addAppending` 也必然发生在 `loadSignature` 之后：

> 虽然图中把 `loadFromDatabase` 和 `loadSignature` 画在了同一个线程里，但事实上它们有可能是在不同线程执行的。不过在上面代码的情况下，它们的先后次序依然是严格不变的。

事实上，虽然最后的 `addAppending` 任务同时需要原始数据和签名才能进行，但 `loadFromDatabase` 和 `loadSignature` 之间其实并没有依赖关系。如果它们能够一起执行的话，我们的程序有很大机率能变得更快。这时候，我们会需要更多的线程，来同时执行两个操作：

```
// loadFromDatabase { (strings, error) in
//     ...
//     loadSignature { signature, error in {
//     ...

// 可以将串行调用替换为：

loadFromDatabase { (strings, error) in
    //...
}

loadSignature { signature, error in
    //...
}
```

> 为了确保在 `addAppending` 执行时，从数据库加载的内容和从网络下载的签名都已经准备好，我们需要某种手段来确保这些数据的可用性。在 GCD 中，通常可以使用 `DispatchGroup` 或者 `DispatchSemaphore` 来实现这一点。但是我们并不是一本探讨 GCD 的书籍，所以这部分内容就略过了。

两个 `load` 方法同时开始工作，理论上资源充足的话 (足够的 CPU，网络带宽等)，现在它们所消耗的时间会小于串行时的两者之和：

这时候， `loadFromDatabase` 和 `loadSignature` 这两个异步操作，在不同的线程中同时执行。对于这种拥有多套资源同时执行的方式，我们就将它称为**并行 (parallel)**。

Swift 并发是什么

在有了这些基本概念后，最后可以谈谈关于并发 (concurrency) 这个名词了。在计算机科学中，并发指的是多个计算同时执行的特性。并发计算中涉及的**同时执行**，主要是若干个操作的开始和结束时间之间存在重叠。它并不关心具体的执行方式：我们可以把同一个线程中的多个操作交替运行 (这需要这类操作能够暂时被置于暂停状态) 叫做并发，这几个操作将会是分时运行的；我们也可以把在不同处理器核心中运行的任务叫做并发，此时这些任务必定是并行的。

而当 Apple 在定义“Swift 并发”是什么的时候，和上面这个经典的计算机科学中的定义实质上没有太多不同。Swift 官方文档给出了这样的解释：

> Swift 提供内建的支持，让开发者能以结构化的方式书写异步和并行的代码，…并发这个术语，指的是异步和并行这一常见组合。

所以在提到 Swift 并发时，它指的就是**异步和并行代码的组合**。这在语义上，其实是传统并发的一个子集：它限制了实现并发的手段就是异步代码，这个限定降低了我们理解并发的难度。在本书中，如果没有特别说明，我们在提到 Swift 并发时，指的都是“异步和并行代码的组合”这个简化版的意义，或者专指 Swift 5.5 中引入的这一套处理并发的语法和框架。

除了定义方式稍有不同之外，Swift 并发和其他编程语言在处理同样问题时所面临的挑战几乎一样。从戴克斯特拉 (Edsger W. Dijkstra) 提出信号量 (semaphore) 的概念起，到东尼・霍尔爵士 (Tony Hoare) 使用 CSP 描述和尝试解决哲学家就餐问题，再到 actor 模型或者通道模型 (channel model) 的提出，并发编程最大的困难，以及这些工具所要解决的问题大致上只有两个：

1. 如何确保不同运算运行步骤之间的交互或通信可以按照正确的顺序执行
2. 如何确保运算资源在不同运算之间被安全地共享、访问和传递

第一个问题负责并发的逻辑正确，第二个问题负责并发的内存安全。在以前，开发者在使用 GCD 编写并发代码时往往需要很多经验，否则难以正确处理上述问题。Swift 5.5 设计了**异步函数**的书写方法，在此基础上，利用**结构化并发**确保运算步骤的交互和通信正确，利用 **actor 模型**确保共享的计算资源能在隔离的情况下被正确访问和操作。它们组合在一起，提供了一系列工具让开发者能简单地编写出稳定高效的并发代码。我们接下来，会浅显地对这几部分内容进行瞥视，并在后面对各个话题展开探究。

> 戴克斯特拉还发表了著名的《GOTO 语句有害论》(Go To Statement Considered Harmful)，并和霍尔爵士一同推动了结构化编程的发展。霍尔爵士在稍后也提出了对 null 的反对，最终促成了现代语言中普遍采用的 `Optional` (或者叫别的名称，比如 `Maybe` 或 null safety 等) 设计。如果没有他们，也许我们今天在编写代码时还在处理无尽的 goto 和 null 检查，会要辛苦很多。

异步函数

为了更容易和优雅地解决上面两个问题，Swift 需要在语言层面引入新的工具：第一步就是添加异步函数的概念。在函数声明的返回箭头前面，加上 `async` 关键字，就可以把一个函数声明为异步函数：

```
1  func loadSignature() async throws -> String {
2    fatalError("暂未实现")
3  }
```

异步函数的 `async` 关键字会帮助编译器确保两件事情：

1. 它允许我们在函数体内部使用 `await` 关键字；
2. 它要求其他人在调用这个函数时，使用 `await` 关键字。

这和与它处于类似位置的 `throws` 关键字有点相似。在使用 `throws` 时，它允许我们在函数内部使用 `throw` 抛出错误，并要求调用者使用 `try` 来处理可能的抛出。`async` 也扮演了这样一个角色，它要求在特定情况下对当前函数进行标记，这是对于开发者的一种明确的提示，表明这个函数有一些特别的性质：`try/throw` 代表了函数可以被抛出，而 `await` 则代表了函数在此处可能会**放弃当前线程**，它是程序的**潜在暂停点**。

放弃线程的能力，意味着异步方法可以被“暂停”，这个线程可以被用来执行其他代码。如果这个线程是主线程的话，那么界面将不会卡顿。被 `await` 的语句将被底层机制分配到其他合适的线程，在执行完成后，之前的“暂停”将结束，异步方法从刚才的 `await` 语句后开始，继续向下执行。

关于异步函数的设计和更多深入内容，我们会在随后的相关章节展开。在这里，我们先来看看一个简单的异步函数的使用。Foundation 框架中已经为我们提供了很多异步函数，比如使用 `URLSession` 从某个 `URL` 加载数据，现在也有异步版本了。在由 `async` 标记的异步函数中，我们可以调用其他异步函数：

```
1  func loadSignature() async throws -> String? {
2    let (data, _) = try await URLSession.shared.data(from: someURL)
3    return String(data: data, encoding: .utf8)
4  }
```

> 这些 Foundation，或者 AppKit 或 UIKit 中的异步函数，有一部分是重写和新添加的，但更多的情况是由相应的 Objective-C 接口转换而来。满足一定条件的 Objective-C 函数，可以直接转换为 Swift 的异步函数，非常方便。在后一章我们也会具体谈到。

如果我们把 `loadFromDatabase` 也写成异步函数的形式。那么，在上面串行部分，原本的嵌套式的异步操作代码：

```swift
loadFromDatabase { (strings, error) in
  if let strings = strings {
    loadSignature { signature, error in
      if let signature = signature {
        strings.forEach {
                strings.forEach {
        strings.forEach {
          addAppending(signature, to: $0)
        }
      } else {
        print("Error")
      }
    }
  } else {
    print("Error.")
  }
}
```

就可以非常简单地写成这样的形式：

```swift
let strings = try await loadFromDatabase()
if let signature = try await loadSignature() {
  strings.forEach {
    addAppending(signature, to: $0)
  }
} else {
  throw NoSignatureError()
}
```

不用多说，单从代码行数就可以一眼看清优劣了。异步函数极大简化了异步操作的写法，它避免了内嵌的回调，将异步操作按照顺序写成了类似“同步执行”的方法。另外，这种写法允许我们使用 try/throw 的组合对错误进行处理，编译器会对所有的返回路径给出保证，而不必像回调那样时刻检查是不是所有的路径都进行了处理。

结构化并发

对于同步函数来说，线程决定了它的执行环境。而对于异步函数，则由任务 (Task) 决定执行环境。Swift 提供了一系列 `Task` 相关 API 来让开发者创建、组织、检查和取消任务。这些 API 围绕着 `Task` 这一核心类型，为每一组并发任务构建出一棵结构化的任务树：

- 一个任务具有它自己的优先级和取消标识，它可以拥有若干个子任务并在其中执行异步函数。
- 当一个父任务被取消时，这个父任务的取消标识将被设置，并向下传递到所有的子任务中去。
- 无论是正常完成还是抛出错误，子任务会将结果向上报告给父任务，在所有子任务完成之前 (不论是正常结束还是抛出)，父任务是不会完成的。

这些特性看上去和 `Operation` 类有一些相似，不过 `Task` 直接利用异步函数的语法，可以用更简洁的方式进行表达。而 `Operation` 则需要依靠子类或者闭包。

在调用异步函数时，需要在它前面添加 `await` 关键字；而另一方面，只有在异步函数中，我们才能使用 `await` 关键字。那么问题在于，第一个异步函数执行的上下文，或者说任务树的根节点，是怎么来的？

简单地使用 `Task.init` 就可以让我们获取一个任务执行的上下文环境，它接受一个 `async` 标记的闭包：

```swift
struct Task<Success, Failure> where Failure : Error {
  init(
    priority: TaskPriority? = nil,
        priority: TaskPriority? = nil,
    priority: TaskPriority? = nil,
    operation: @escaping @Sendable () async throws -> Success
  )
}
```

它继承当前任务上下文的优先级等特性，创建一个新的任务树根节点，我们可以在其中使用异步函数：

```swift
var results: [String] = []

func someSyncMethod() {
  Task {
    try await processFromScratch()
    print("Done: \(results)")
  }
}

func processFromScratch() async throws {
  let strings = try await loadFromDatabase()
  if let signature = try await loadSignature() {
    strings.forEach {
      results.append($0.appending(signature))
    }
  } else {
    throw NoSignatureError()
  }
}
```

注意，在 `processFromScratch` 中的处理依然是串行的：对 `loadFromDatabase` 的 `await` 将使这个异步函数在此暂停，直到实际操作结束，接下来才会执行 `loadSignature` ：

我们当然会希望这两个操作可以同时进行。在两者都准备好后，再调用 `appending` 来实际将签名附加到数据上。这需要任务以结构化的方式进行组织。使用 `async let` 绑定可以做到这一点：

```swift
func processFromScratch() async throws {
  async let loadStrings = loadFromDatabase()
  async let loadSignature = loadSignature()

  results = []

  let strings = try await loadStrings
  if let signature = try await loadSignature {
      strings.forEach {
        addAppending(signature, to: $0)
      }
  } else {
    throw NoSignatureError()
  }
}
```

`async let` 被称为**异步绑定**，它在当前 Task 上下文中创建新的子任务，并将它用作被绑定的异步函数 (也就是 `async let` 右侧的表达式) 的运行环境。和 `Task.init` 新建一个任务根节点不同， `async let` 所创建的子任务是任务树上的叶子节点。被异步绑定的操作会立即开始执行，即使在 `await` 之前执行就已经完成，其结果依然可以等到 `await` 语句时再进行求值。在上面的例子中， `loadFromDatabase` 和 `loadSignature` 将被并发执行。

相对于 GCD 调度的并发，基于任务的结构化并发在控制并发行为上具有得天独厚的优势。为了展示这一优势，我们可以尝试把事情再弄复杂一点。上面的 `processFromScratch` 完成了从本地加载数据，从网络获取签名，最后再将签名附加到每一条数据上这一系列操作。假设我们以前可能就做过类似的事情，并且在服务器上已经存储了所有结果，于是我们有机会在进行本地运算的同时，去尝试直接加载这些结果作为“优化路径”，避免重复的本地计算。类似地，可以用一个异步函数来表示“从网络直接加载结果”的操作：

```
func loadResultRemotely() async throws {
  // 模拟网络加载的耗时
  await Task.sleep(2 * NSEC_PER_SEC)
  results = ["data1^sig", "data2^sig", "data3^sig"]
}
```

除了 `async let` 外，另一种创建结构化并发的方式，是使用任务组 (Task group)。比如，我们希望在执行 `loadResultRemotely` 的同时，让 `processFromScratch` 一起运行，可以用 `withThrowingTaskGroup` 将两个操作写在同一个 task group 中：

```
func someSyncMethod() {
  Task {
    await withThrowingTaskGroup(of: Void.self) { group in
      group.addTask {
        try await self.loadResultRemotely()
      }
      group.addTask(priority: .low) {
        try await self.processFromScratch()
      }
    }
        }
    }
    print("Done: \(results)")
  }
}
```

> 对于 `processFromScratch` ，我们为它特别指定了 `.low` 的优先级，这会导致该任务在另一个低优先级线程中被调度。我们一会儿会看到这一点带来的影响。

`withThrowingTaskGroup` 和它的非抛出版本 `withTaskGroup` 提供了另一种创建结构化并发的组织方式。当在运行时才知道任务数量时，或是我们需要为不同的子任务设置不同优先级时，我们将只能选择使用 Task Group。在其他大部分情况下， `async let` 和 task group 可以混用甚至互相替代：

闭包中的 `group` 满足 `AsyncSequence` 协议，它让我们可以使用 `for await` 的方式用类似同步循环的写法来访问异步操作的结果。另外，通过调用 group 的 `cancelAll` ，我们可以在适当的情况下将任务标记为取消。比如在 `loadResultRemotely` 很快返回时，我们可以取消掉正在进行的 `processFromScratch` ，以节省计算资源。关于异步序列和任务取消这些话题，我们会在稍后专门的章节中继续探讨。

actor 模型和数据隔离

在 `processFromScratch` 里，我们先将 `results` 设置为 `[]` ，然后再处理每条数据，并将结果添加到 `results` 里：

```
func processFromScratch() async throws {
  // ...
    results = []
    strings.forEach {
      addAppending(signature, to: $0)
    }
  // ...
}
```

在作为示例的 `loadResultRemotely` 里，我们现在则是直接把结果赋值给了 `results` ：

```
func loadResultRemotely() async throws {
  await Task.sleep(2 * NSEC_PER_SEC)
  results = ["data1^sig", "data2^sig", "data3^sig"]
}
```

因此，一般来说我们会认为，不论 `processFromScratch` 和 `loadResultRemotely` 执行的先后顺序如何，我们总是应该得到唯一确定的 `results` ，也就是数据 `["data1^sig", "data2^sig", "data3^sig"]` 。但事实上，如果我们对 `loadResultRemotely` 的 `Task.sleep` 时长进行一些调整，让它和 `processFromScratch` 所耗费的时间相仿，就可能会看到出乎意料的结果。在正确输出三个元素的情况外，有时候它会输出六个元素：

```
// 有机率输出：
Done: ["data1^sig", "data2^sig", "data3^sig",
"data1^sig", "data2^sig", "data3^sig"]
```

我们在 `addTask` 时为两个任务指定了不同的优先级，因此它们中的代码将运行在不同的调度线程上。两个异步操作在不同线程同时访问了 `results` ，造成了数据竞争。在上面这个结果中，我们可以将它解释为 `processFromScratch` 先将 `results` 设为了空数列，紧接着 `loadResultRemotely` 完成，将它设为正确的结果，然后 `processFromScratch` 中的 `forEach` 把计算得出的三个签名再添加进去。

这大概率并不是我们想要的结果。不过幸运的是两个操作现在并没有真正“同时”地去更改 `results` 的内存，它们依然有先后顺序，因此只是最后的数据有些奇怪。

`processFromScratch` 和 `loadResultRemotely` 在不同的任务环境中对变量 `results` 进行了操作。由于这两个操作是并发执行的，所以也可能出现一种更糟糕的情况：它们对 `results` 的操作同时发生。如果 `results` 的底层存储被多个操作同时更改的话，我们会得到一个运行时错误。作为示例 (虽然没有太多实际意义)，通过增加 `someSyncMethod` 的运行次数就可以很容易地让程序崩溃：

```
for _ in 0 ..< 10000 {
  someSyncMethod()
}

// 运行时崩溃。一个典型的内存错误
// Thread 10: EXC_BAD_ACCESS (code=1, address=0x55a8fdbc060c)
```

为了确保资源 (在这个例子里，是 `results` 指向的内存) 在不同运算之间被安全地共享和访问，以前通常的做法是将相关的代码放入一个串行的 dispatch queue 中，然后以同步的方式把对资源的访问派发到队列中去执行，这样我们可以避免多个线程同时对资源进行访问。按照这个思路可以进行一些重构，将 `results` 放到新的 `Holder` 类型中，并使用私有的 `DispatchQueue` 将它保护起来：

```swift
class Holder {
    private let queue = DispatchQueue(label: "resultholder.queue")
    private var results: [String] = []

    func getResults() -> [String] {
        queue.sync { results }
    }

    func setResults(_ results: [String]) {
        queue.sync { self.results = results }
    }

    func append(_ value: String) {
        queue.sync { self.results.append(value) }
    }
}
```

接下来，将原来代码中使用到 `results: [String]` 的地方替换为 `Holder` ，并使用暴露出的方法将原来对 `results` 的直接操作进行替换，可以解决运行时崩溃的问题。

```swift
// var results: [String] = []
var holder = Holder()

// ...
// results = []
holder.setResults([])

// results.append(data.appending(signature))
holder.append(data.appending(signature))

// print("Done: \(results)")
print("Done: \(holder.getResults())")
```

在使用 GCD 进行并发操作时，这种模式非常常见。但是它存在一些难以忽视的问题：

1. 大量且易错的模板代码：凡是涉及 `results` 的操作，都需要使用 `queue.sync` 包围起来，但是编译器并没有给我们任何保证。在某些时候忘了使用队列，编译器也不会进行任何提示，这种情况下内存依然存在危险。当有更多资源需要保护时，代码复杂度也将爆炸式上升。
2. 小心死锁：在一个 `queue.sync` 中调用另一个 `queue.sync` 的方法，会造成线程死锁。在代码简单的时候，这很容易避免，但是随着复杂度增加，想要理解当前代码运行是由哪一个队列派发的，它又运行在哪一个线程上，往往会伴随着严重的困难。必须精心设计，避免重复派发。

在一定程度上，我们可以用 `async` 替代 `sync` 派发来缓解死锁的问题；或者放弃队列，转而使用锁 (比如 `NSLock` 或者 `NSRecursiveLock`)。不过不论如何做，都需要开发者对线程调度和这种基于共享内存的数据模型有深刻理解，否则非常容易写出很多坑。

Swift 并发引入了一种在业界已经被多次证明有效的新的数据共享模型，**actor 模型** (参与者模型)，来解决这些问题。虽然有些偏失，但最简单的理解，可以认为 actor 就是一个“封装了私有队列”的 class。将上面 `Holder` 中 `class` 改为 `actor` ，并把 `queue` 的相关部分去掉，我们就可以得到一个 actor 类型。这个类型的特性和 `class` 很相似，它拥有引用语义，在它上面定义属性和方法的方式和普通的 `class` 没有什么不同：

```swift
actor Holder {
  var results: [String] = []
  func setResults(_ results: [String]) {
    self.results = results
  }

  func append(_ value: String) {
    results.append(value)
  }
}
```

对比由私有队列保护的“手动挡”的 class，这个“自动档”的 actor 实现显然简洁得多。actor 内部会提供一个隔离域：在 actor 内部对自身存储属性或其他方法的访问，比如在 `append(_:)` 函数中使用 `results` 时，可以不加任何限制，这些代码都会被自动隔离在被封装的“私有队列”里。但是从外部对 actor 的成员进行访问时，编译器会要求切换到 actor 的隔离域，以确保数据安全。在这个要求发生时，当前执行的程序可能会发生暂停。编译器将自动把要跨隔离域的函数转换为异步函数，并要求我们使用 `await` 来进行调用。

> 虽然实际底层实现中，actor 并非持有一个私有队列，但是现在，你可以就这样简单理解。在本书后面的部分我们会做更深入的探索。

当我们把 `Holder` 从 `class` 转换为 `actor` 后，原来对 `holder` 的调用也需要更新。简单来说，在访问相关成员时，添加 `await` 即可：

```swift
// holder.setResults([])
await holder.setResults([])

// holder.append(data.appending(signature))
await holder.append(data.appending(signature))

// print("Done: \(holder.getResults())")
print("Done: \(await holder.results)")
```

现在，在并发环境中访问 `holder` 不再会造成崩溃了。不过，即时使用 `Holder`，不论是基于 `DispatchQueue` 还是 `actor`，上面代码所得到的结果中依然可能会存在多于三个元素的情况。这是在预期内的：数据隔离只解决同时访问的造成的内存问题 (在 Swift 中，这种不安全行为大多数情况下表现为程序崩溃)。而这里的数据正确性关系到 actor 的**可重入** (reentrancy)。要正确理解可重入，我们必须先对异步函数的特性有更多了解，因此我们会在之后的章节里再谈到这个话题。

另外，`actor` 类型现在还并没有提供指定具体运行方式的手段。虽然我们可以使用 `@MainActor` 来确保 UI 线程的隔离，但是对于一般的 actor，我们还无法指定隔离代码应该以怎样的方式运行在哪一个线程。我们之后也还会看到包括全局 actor、非隔离标记 (nonisolated) 和 actor 的数据模型等内容。

小结

我想本章应该已经有足够多的内容了。我们从最基本的概念开始，展示了使用 GCD 或者其他一些“原始”手段来处理并发程序时可能面临的困难，并在此基础上介绍了 Swift 并发中处理和解决这些问题的方式。

Swift 并发虽然涉及的概念很多，但是各种的模块边界是清晰的：

- 异步函数：提供语法工具，使用更简洁和高效的方式，表达异步行为。
- 结构化并发：提供并发的运行环境，负责正确的函数调度、取消和执行顺序以及任务的生命周期。
- actor 模型：提供封装良好的数据隔离，确保并发代码的安全。

熟悉这些边界，有助于我们清晰地理解 Swift 并发各个部分的设计意图，从而让我们手中的工具可以被运用在正确的地方。作为概览，在本章中读者应该已经看到如何使用 Swift 并发的工具书写并发代码了。本书接下来的部分，将会对每个模块做更加深入的探讨，以求将更多隐藏在宏观概念下的细节暴露出来。

能工巧匠集, Swift

swift 编程语言 并发

接下来阅读

2021年9月29日

Swift 结构化并发

本文是我的新书《Swift 异步和并发》中的部分内容，介绍了关于 Swift 中结构…

2014年6月3日

关于 Swift 的一点初步看法

虽然四点半就起床去排队等入场，结果还是只能坐在了蛮后面的位置看着大屏幕…

2014年6月7日

行走于 Swift 的世界中

2014 年 7 月 13 日更新：根据 beta 3 的情况修正了文中过时的部分 从周一 Swift …

较早文章
用树莓派打造一个超薄魔镜的简单教程

较新文章
Swift 结构化并发

42 Comments - *powered by utteranc.es*

hjw6160602 commented on 2021年7月1日

瞄大yyds 顺便问下 Xcode 13 beta 2 环境 安装最新的 Swift 5.5 toolchain 写出来的 并发代码可以跑在iOS14以下的系统上吗?

onevcat commented on 2021年7月1日 Owner

> 瞄大yyds 顺便问下 Xcode 13 beta 2 环境 安装最新的 Swift 5.5 toolchain 写出来的 并发代码可以跑在iOS14以下的系统上吗?

并不可以 😂

😂 2

HideOnBushTuT commented on 2021年7月2日

那完全没有动力去学啊

😂 1

onevcat commented on 2021年7月2日 Owner

如果是 iOS 15 才能用上的话，确实没动力..不过社区一直在吵要不要back port到之前的版本 (比如 iOS 12)。Xcode 的 release note 里是把 “from iOS 15” 这件事情列在 Known issue 里了，Core Team 和 Darwin 的人也知道社区对这个的不满，但是他们表示工作量非常大，不能给什么承诺或者时间表。

所以说其实还有转机...看 Apple 要不要强迫员工加班加到死了 😂

👍 1 👀 1

hjw6160602 commented on 2021年7月2日

> 那完全没有动力去学啊

这是你不学习的借口吗？[加油]

hjw6160602 commented on 2021年7月2日

> 如果是 iOS 15 才能用上的话，确实没动力..不过社区一直在吵要不要back port到之前的版本 (比如 iOS 12)。Xcode 的 release note 里是把 “from iOS 15” 这件事情列在 Known issue 里了，Core Team 和 Darwin 的人也知道社区对这个的不满，但是他们表示工作量非常大，不能给什么承诺或者时间表。

翻喵大之前微博看到了 不过这玩意儿 等等 要真能从iOS12开始支持确实太棒了

fatbobman commented on 2021年7月3日

期待尽早完稿！

anchao-lu commented on 2021年7月6日

真要 15 才能用的话，会很难受的，光看着好，就是不能用

wang542413041 commented on 2021年7月7日

不能像以前的运行时库一起从Apple store下载么？

taosiyu commented on 2021年7月27日

哈哈哈，不错，可以来个优惠码

is0bnd commented on 2021年8月18日

假如某个异步任务需要在某种未完成情况下取消，async/await 是否提供了中途取消的机制，需要怎么做？

onevcat commented on 2021年8月18日 Owner

> 假如某个异步任务需要在某种未完成情况下取消，async/await 是否提供了中途取消的机制，需要怎么做？

Swift 并发采用的是协作式的取消方式，从而满足结构化并发的要求。单纯的 `Task.cancel` 调用只是将 `Task` 的取消标识设为 `true`，并不会终止 (也无法终止) 任务的运行。实际的取消依赖异步函数的具体实现。在书中的话，会有关于这个话题一整章的详细展开。

venn0126 commented on 2021年8月19日

那是不是可以理解为Task算是异步模型的一个高层抽象，对于取消的实际操作还是依靠异步功能函数来实现，而异步函数对于已经在执行中的无法取消，而是对于等待或者排队中的任务才可以取消？

cache0928 commented on 2021年8月19日

```
class Person {
  var name: String
  let birthDate: Date

  init(name: String, birthDate: Date) {
    self.name = name
    self.birthDate = birthDate
  }
}

actor BankAccount {
```

```
  var owners: [Person] = [Person(name: "", birthDate: Date.now)]
  func primaryOwner() -> Person? { return owners.first }
}

let account = BankAccount()
Task.detached(priority: .background) {
  if let primary = await account.primaryOwner() {
    primary.name = "The Honorable " + primary.name  // problem: concurrent mutation of actor-
    print(primary.name)
  }
}
```

onevcat commented on 2021年8月19日 Owner

@venn0126

> 那是不是可以理解为Task算是异步模型的一个高层抽象

`Task` 主要是一套为了实践结构化并发搞的 API。在这个角度上，你可以说 `Task` 是对异步模型 (主要是结构化并发) 的一个高层抽象，但是它和异步函数的关系仅仅只是后者提供了工具。。

> 而异步函数对于已经在执行中的无法取消，而是对于等待或者排队中的任务才可以取消

这是不准确的。

事实上，在结构化并发的 `Task` API 中，是没有我们理解的所谓的传统的"取消"，或者说任务运行到一半就突然中止了这种事情的。异步操作的生命周期一定和任务的 `{ }` 作用域绑定。

`Task` API 的取消只是设置一个 cancel 值。异步函数的能力 (或者更精确说，GCD 中新加的 cooperative thread pool 的调度执行的能力)，有能力将任务细分为若干个等待和执行的段，在每一段开始前，实现这个异步函数的人有机会检查 cancel 值，并决定要继续执行还是抛出。

当然，这只是在高层级上可以做到的事情。如果能够深入到操作 `continuation` (比如这个或者这个)，针对取消，就会有更多的选择。

onevcat commented on 2021年8月19日 Owner

@cache0928

对，现在这个没有按照 proposal 的提案报错，可能是还没有来得及实装。

不过其实关于这个是有一点争论的。 `Person` 作为 class，本来就不应该有并发的数据安全。在 `actor BackAccount` 中，它的成员 `owners` 确实是被保护的；而 `Person.name` 作为 `Person` 的一员，它的特性应该由 `Person` 来决定。如果要保护 `name` ，那么应该由开发者明确地把 `Person` 也做成 actor 或者 `Sendable` 。

可能这部分暂时会被作为容忍范围内的问题，会在 Roadmap 提到的第二阶段里再进行实装。

> The second phase will enforce full actor isolation
>
> Class component memory can also be accessed from any code that hold a reference to the class. This means that while the reference to the class may be protected by an actor, passing that reference between actors exposes its properties to data races. This also includes references to classes held within value types, when these are passed between actors.
>
> The goal of full actor isolation is to ensure that (this is) protected by default.

cache0928 commented on 2021年8月19日

> @cache0928
>
> 对，现在这个没有按照 proposal 的提案报错，可能是还没有来得及实装。
>
> 不过其实关于这个是有一点争论的。 `Person` 作为 class，本来就不应该有并发的数据安全。在 `actor BackAccount` 中，它的成员 `owners` 确实是被保护的；而 `Person.name` 作为 `Person` 的一员，它的特性应该由 `Person` 来决定。如果要保护 `name` ，那么应该由开发者明确地把 `Person` 也做成 actor 或者 `Sendable` 。
>
> 可能这部分暂时会被作为容忍范围内的问题，会在 Roadmap 提到的第二阶段里再进行实装。
>
> > The second phase will enforce full actor isolation
> >
> > Class component memory can also be accessed from any code that hold a reference to the

> class. This means that while the reference to the class may be protected by an actor, passing that reference between actors exposes its properties to data races. This also includes references to classes held within value types, when these are passed between actors.
>
> The goal of full actor isolation is to ensure that (this is) protected by default.

我在Task的构造器@sendable闭包中直接捕获非Sendable的实例（比如Person的实例）也不会报错，@sendable闭包不是不能捕获非Sendable成员的吗?

venn0126 commented on 2021年8月20日

翻了一些资料和你的官方API的查看，其实这个"Task or Task Group"其实就是把aync和parallel对高级开发接口的扩充，这也是官方Beta文档中指出的结构化的主题，那么对于你说的" GCD 中新加的 cooperative thread pool 的调度执行的能力"，是对结构化的底层的支持么？如果是有相关的这方面的说明资料么，如果不是，那对结构化底层的支持是什么?

Wei Wang ***@***.***>于2021年8月19日 周四17:44写道：

...

onevcat commented on 2021年8月20日 Owner

`Task` 只是对 LLVM Builtin 的任务指针和其他一些方法以及 flag (优先级，是否是子任务等) 的封装，GCD 使用这个封装来标记和确认如何在新的 cooperative thread pool 进行调度，而对于 `Task` 来说，执行这个调度的调度器是一个全局的并行 executor。

async 函数或者 await 等，都只是编译期间的辅助，实际上它们只是表示了这些东西必须跑在一个任务中，或者说，接受全局调度器的调度。因为 cooperative thread pool 调度的特点，调用栈可以在 continuation 之间得到保留，因此返回和抛出都成为可能，结构化并发也就自然实现了。

venn0126 commented on 2021年8月20日

可否是这样理解（非科班出身，请忽略作图美观）

Wei Wang ***@***.***>于2021年8月20日 周五10:37写道：

...

venn0126 commented on 2021年8月20日

如果看不到请点击这里

onevcat commented on 2021年8月20日 Owner

@venn0126 嗯，我的理解差不多是这样..

venn0126 commented on 2021年8月20日

@onevcat 感谢回复❤️

RbBtSn0w commented on 2021年8月27日

官方发文了. 可以了.
书本要畅销了. 来个促销庆祝活动吧

onevcat commented on 2021年8月27日 Owner

> 瞄大yyds 顺便问下 Xcode 13 beta 2 环境 安装最新的 Swift 5.5 toolchain 写出来的 并发代码可以跑在iOS14以下的系统上吗?

并不可以 😂

apple/swift#39051

Core Team 居然真的在发布之前加班加出来了...

hjw6160602 commented on 2021年8月27日

> apple/swift#39051
> Core Team 居然真的在发布之前加班加出来了...

@onevcat 哈哈 太好了 看到说是macOS的开发都跑去开发iOS了 因为macOS发布时间普遍晚一个月

nuomi1 commented on 2021年8月27日

```
if (SWIFT_ALLOW_BACK_DEPLOY_CONCURRENCY)
  set(swift_concurrency_availability "macOS 10.15, iOS 13.0, watchOS 6.0, tvOS 13.0")
else()
  set(swift_concurrency_availability "macOS 12.0, iOS 15.0, watchOS 8.0, tvOS 15.0")
endif()
```

最低 iOS 13~

👍 1

imWildCat commented on 2021年8月27日

> 官方发文了. 可以了.
> 书本要畅销了. 来个促销庆祝活动吧

@RbBtSn0w 来源请求?

bestwnh commented on 2021年8月28日

> ```cmake
> if (SWIFT_ALLOW_BACK_DEPLOY_CONCURRENCY)
>   set(swift_concurrency_availability "macOS 10.15, iOS 13.0, watchOS 6.0, tvOS 13.0")
> else()
>   set(swift_concurrency_availability "macOS 12.0, iOS 15.0, watchOS 8.0, tvOS 15.0")
> endif()
> ```
>
> 最低 iOS 13~

iOS13已经好很多了，更新进程可以提早两年了。

yongyang007 commented on 2021年9月17日

> > ```cmake
> > if (SWIFT_ALLOW_BACK_DEPLOY_CONCURRENCY)
> >   set(swift_concurrency_availability "macOS 10.15, iOS 13.0, watchOS 6.0, tvOS 13.0")
> > else()
> >   set(swift_concurrency_availability "macOS 12.0, iOS 15.0, watchOS 8.0, tvOS 15.0")
> > endif()
> > ```
> >
> > 最低 iOS 13~
>
> iOS13已经好很多了，更新进程可以提早两年了。

看来短时间内还不能实现
https://forums.swift.org/t/swift-concurrency-back-deployment/51908/3

venn0126 commented on 2021年9月17日

> > ```cmake
> > if (SWIFT_ALLOW_BACK_DEPLOY_CONCURRENCY)
> >   set(swift_concurrency_availability "macOS 10.15, iOS 13.0, watchOS 6.0, tvOS 13.
> > else()
> >   set(swift_concurrency_availability "macOS 12.0, iOS 15.0, watchOS 8.0, tvOS 15.0
> > endif()
> > ```
>
> 最低 iOS 13~
>
> iOS13已经好很多了，更新进程可以提早两年了。

看来短时间内还不能实现
https://forums.swift.org/t/swift-concurrency-back-deployment/51908/3

先提前学习吧

imWildCat commented on 2021年9月17日

PR 在做了：apple/swift#39342

onevcat commented on 2021年9月17日 Owner

十分怀疑能不能赶得上正式 release...这个东西不经过一段时间验证的话还是很慌的。

之前有 rumor 是说要推后到 Swift 5.6。不过以最近 Apple 的 beta 当 alpha，正式版当 beta 的尿性..搞不好不测了直接强行在 Swift 5.5 就弄上也不无可能...

imWildCat commented on 2021年9月25日

@onevcat back-port 合并到 5.5 branch 了，我们赶紧去买书 😂

😂 1 🎉 2

qhd commented on 2022年2月24日

发现 @mainactor 修饰的方法没有在主线程执行，这是 bug 吗

```
override func viewDidLoad() {
    super.viewDidLoad()

    Task {
        await doSomething()
    }
}

func doSomething() async {
    await Task.sleep(2)
    print("doSomething thread \(Thread.current)")
    updateUI()
}

@MainActor
func updateUI() {
    print("updateUI thread \(Thread.current)")
}
```

onevcat commented on 2022年2月24日 Owner

@qhd 如果你是在一个没有“完全迁移”到 Swift Concurrency Safe 的项目的话，可能需要在 class 申明上也加上 `@MainActor` 来让它生效：

```
@MainActor
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
```

```
    Task {
        await doSomething()
    }
}

func doSomething() async {
    await Task.sleep(2)
    print("doSomething thread \(Thread.current)")
    updateUI()
}

@MainActor
func updateUI() {
    print("updateUI thread \(Thread.current)")
}
```

onevcat commented on 2022年2月24日 Owner

另外，需要指出的是，`@MainActor` 需要 async 环境来完成 actor 的切换。

你最早的例子中只有 `doSomething` 会运行在自己的线程里，为全局的 `ViewController` 加上 `@MainActor` 意味着 `doSomething` 也许要 main actor。但是这并不影响 `updateUI`。

所以另一种"修正"的方法（可能也是你更想要的方法），是把 `updateUI` 标记成 async：

```
func doSomething() async {
    await Task.sleep(2)
    print("doSomething thread \(Thread.current)")
    await updateUI()
}

@MainActor
func updateUI() async {
    print("updateUI thread \(Thread.current)")
}
```

onevcat commented on 2022年2月24日 Owner

这个问题，严格来说，应该算是一种编译器的考虑不周？因为 MainActor 中的函数（`updateUI`）在从外部使用时，应该是要强制加上 await 的。实际上 `doSomething` 是在 MainActor 之外的。但是在 Swift 5.5 这个尴尬的时间点上，编译器没有给提示。

所以 Swift 5.5 里想要认真写 Swift 并发的话，建议还是把 `-Xfrontend -warn-concurrency -Xfrontend -enable-actor-data-race-checks` 加到 `OTHER_SWIFT_FLAGS`，这样能帮你预先抓到一些类似的问题。

bestwnh commented on 2022年2月25日

> 实际上 `doSomething` 是在 MainActor 之外的。

@onevcat 其实为什么 `doSomething` 会在MainActor之外？不是class声明为MainActor的话，下面所有的属性和方法都自动变成MainActor的么？或者说不是因为UIViewController已经是MainActor，所以继承它的ViewController已经是MainActor了么？

MainActor的行为相当的不清晰，另外想问一下目前有方法让一些方法可以根据调用者的线程保持在对应线程执行而不指定线程的么？像Then这样的框架如果加了上面的 `OTHER_SWIFT_FLAGS` 目前会出现不声明MainActor则没法执行MainActor的代码，加了的话就没法执行非MainActor的代码这样的必须取舍的情况。

BackWorld commented on 2022年3月25日

✅

BackWorld commented on 2022年3月25日

@bestwnh 你那里的 `doSomething()` 方法是个 `async` 方法，这相当于我下面写的 `detached` 一个方法里执行的操作，所以也相当于 `nonisolated func`，所以脱离了 `MainActor` 的隔离区域，因为一般情况下，你的 `MainActor` 里的操作都是 `sync` 同步的，不可能出现 `async`，一旦你标识为 `async`，系统可能（猜想）会隐式的给你 `kick out of the main actor isolated context`，所以打印出来才不是主线程。

上述全是个人见解（猜想），不知道对不对。

```
@MainActor
class MyViewController: UIViewController {
    func onPress(...) { ... } // implicitly @MainActor

// 这个方法可以脱离主线程运行
    nonisolated func fetchLatestAndDisplay() async { ... }
}
```

fanthus commented on 2022年7月12日

学习了，同步函数和同步操作并不是一个概念。

Write Preview

Sign in to comment