



# SWIFTER

Swift 异步和并发

王巍 (@onevcat)

1.0 (2021 年 9 月)

© 2021~ ObjC 中国

版权所有

ObjC 中国

在中国地区独家翻译和销售授权

获取更多书籍或文章, 请访问 <https://objccn.io>

电子邮件: mail@objccn.io

<b>1</b>	<b>简介</b>	<b>6</b>
	目标读者 7	
	章节结构 8	
	准备工作 8	
<b>2</b>	<b>Swift 并发初步</b>	<b>10</b>
	一些基本概念 11	
	异步函数 18	
	结构化并发 20	
	actor 模型和数据隔离 25	
	小结 30	
<b>3</b>	<b>创建异步函数</b>	<b>32</b>
	异步函数的动机 33	
	转换函数签名 37	
	使用续体改写函数 41	
	Objective-C 自动转换 47	
	Async getter 50	
	小结 55	
<b>4</b>	<b>异步序列</b>	<b>57</b>
	同步序列和异步序列 58	
	异步迭代器 60	
	操作异步序列 65	
	AsyncStream 74	
	异步序列和响应式编程 86	
	小结 89	

<b>5</b>	<b>使用异步函数</b>	<b>91</b>
	网络请求中的异步函数 <b>92</b>	
	Notification <b>102</b>	
	异步函数的运行环境 <b>104</b>	
	小结 <b>109</b>	
<b>6</b>	<b>结构化并发</b>	<b>111</b>
	什么是结构化 <b>112</b>	
	基于 Task 的结构化并发模型 <b>118</b>	
	非结构化任务 <b>140</b>	
	小结 <b>144</b>	
<b>7</b>	<b>协作式任务取消</b>	<b>145</b>
	任务取消到底做了什么 <b>146</b>	
	处理任务取消 <b>149</b>	
	取消的清理工作 <b>161</b>	
	隐式等待和任务暂停 <b>166</b>	
	小结 <b>168</b>	
<b>8</b>	<b>actor 模型和数据隔离</b>	<b>170</b>
	共享内存模型的困境 <b>171</b>	
	Actor 隔离 <b>174</b>	
	Actor 协议 <b>178</b>	
	小结 <b>192</b>	
<b>9</b>	<b>全局 actor, 可重入和 Sendable</b>	<b>194</b>
	全局 actor <b>195</b>	

可重入 204	
Sendable 208	
小结 227	
<b>10 并发线程模型</b>	<b>228</b>
协同式线程池 229	
执行器 246	
任务本地值和任务追踪 255	
小结 261	
<b>11 总结和展望</b>	<b>262</b>
总结 265	
更新履历 266	

# 简介

1

在 Swift “七年之痒”的 2021 年，“千呼万唤始出来”的 Swift 并发编程犹如一剂强心针，出现在了大家面前。当广大 Swift 开发者们还沉浸在终于得到了 `async` 和 `await` 的欢喜之时，我们不禁要想，对比起一些同级别的语言，这一切似乎有些姗姗来迟：并发和异步编程的前辈语言 C# 早在 2012 年就加入了异步方法和任务 API；隔壁同为主打客户端开发起家的 Kotlin 在 2016 年的 1.1 版中引入了协程 (coroutines) 预览；与 Swift 社区有着不解之缘的 Rust 开发者们，也从 2019 年开始就可以自由地徜徉在异步编程的世界中了。

为什么 Swift 中语言及标准库级别的异步和并发编程开发进度相对缓慢，以至于那么多年都“等白了头”？Swift 的异步编程模型要如何使用，是不是无脑跟着编译器提示写代码就万事大吉？Swift 并发编程和其他语言中类似的概念有什么异同，学习这些概念能对其他语言的使用有所帮助吗？我们要如何更新自己的理解和认知，来利用这些新特性完善我们的代码和项目，并进一步精进自己的技艺呢？在这本书中，我想要带着这些问题，对 Swift 的异步和并发编程做一些研究，和大家一同学习和探索。

## 目标读者

这本书面向的是已经入门 Swift 并至少可以理解 Swift 语法、熟悉标准库使用的开发者。我们会从像是方法调用或线程调度等一些基本概念开始，逐步揭示出 Swift 异步和并发编程世界的特性，同时讲解大部分语言和标准库中的有关概念。但我们并不会逐行解释程序 (特别是那些同步世界中的代码) 的语法和运行机制。如果您想要从零开始学习 Swift，或者接触 Swift 的时间还很短，那么在阅读时有可能会有一些难以理解的地方。如果遇到这种情况，建议您可以先阅读 [Apple 官方的 Swift 教程中的相关内容](#) (您也可以在[这里](#)找到这些文档的中文版本)。

特别地，您可能会需要有一定的 Swift 实际开发经验，才能更透彻地理解 Swift 并发相关的内容，至少您应该使用过像是回调 (callback)，代理 (delegate) 等模式处理基本的异步操作。如果您用 Swift 开发过实际项目的话，这个要求肯定不成问题：相关的操作遍布在 Swift 标准库和日常的开发框架 (Foundation, UIKit 等) 的各个地方。如果您 (不幸地) 主要在使用 Objective-C 进行日常开发，您也可以通过本书理解 Swift 在并发编程方面所具有的巨大优势，这些知识在今后您进行迁移时必将成为助力。

作为本书的读者，可能您对 Swift 相关的其他话题也会有兴趣，欢迎查看我们的其他书籍。您可以在 [ObjC 中国的网站](#) 找到其他所有书籍的信息，它们包含了 Swift 开发的各个方面，相信您一定能找到自己感兴趣的话题，并藉此构建出更加完整的 Swift 知识体系。

# 章节结构

本书分为五个部分，它们包括了一个综述部分，三个对 Swift 并发中主要模块进行详细解释的部分，以及最后对并发底层模型和调度方面说明的部分。具体来说：

- 首先是对 Swift 异步和并发编程模型的综述，在这个部分中，我们希望阐明要解决的问题，并对 Swift 异步和并发的整套方案在宏观上进行一个概览。我们首先要釐清一些基本的概念，虽然 Apple 使用 Swift 并发 (Swift Concurrency) 这个名词来描述 Swift 5.5 中加入的这些特性，但并发这个词在计算机科学中具有更广阔的意涵。我们会通过一个具体的例子，对相关概念进行解释，并基于这个例子讲述最基本的 Swift 并发编程方式。如果你只需要不求甚解地“按照编译器提示”去处理简单的并发工作，那阅读这个部分大概就可以覆盖你的日常需求了。
- 对于希望“知其然，更知其所以然”的读者，第二个部分会开始对 Swift 并发编程进行深入探索，包括研究它的设计思想和部分实现机理。Swift 并发大致由三块主要内容构成：异步函数、结构化并发、以及 actor 模型。另外还有像是异步序列 (async sequence)、任务本地值 (task local value) 等小一些的概念。这些内容环环相扣，一同构建了 Swift 的并发模型。在这部分中，我们力求对这些概念进行详细阐明，并探求一些更深入的话题及细节。
- 最后是对 Swift 并发底层机制的“刨根问底”，包括协作式线程池的调度方式、异步函数的线程模型和执行器相关的话题。我们不会去纠结源码级别的实现细节，但是会揭示 Swift 并发的一些底层思想，以及探索怎样才能让 Swift 并发机制和现有的代码协同合作并保证它的性能优势。

章节之间是有顺序关系的，笔者推荐按照书籍顺序去阅读每个章节，而不是跳过其中某些内容。Swift 异步和并发编程的模型具有很高的综合性，它们环环相扣：语法的设计和标准库中 API 的设计相辅相成，合力解决了一系列在 Apple 平台编程时原本十分困难的问题，同时对外提供了相对优雅的 API 设计和强有力的编译期间保证。按照顺序阅读的话，可以有利于层层递进，逐步了解 Swift 异步和并发模型的设计思路。

# 准备工作

Swift 在 5.5 版本中加入了异步方法的语言特性和一系列并发相关的 API，所以想要实践本书内容，你至少需要搭载 Swift 5.5 的开发环境。如果你使用的是 macOS，最简单的方式就是登入 [Apple 开发者网站](#)，并下载 Xcode 13 或者后续版本。如果你使用 Linux 系统，也可以遵循 [Swift 官方网站](#)的说明，下载并安装对应版本的 Swift。

当你准备好后，可以通过运行 `swift` 命令来确认你的 Swift 工具链版本号，一切正常的话，应该可以看到类似的输出。请确认 Swift 版本号大于等于 5.5：

```
$ swift --version  
Apple Swift version 5.5 (swiftlang-xxxx clang-yyyy)
```

我们已经准备好开始进入到 Swift 并发的世界中了！

# Swift 并发初步

2

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：

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

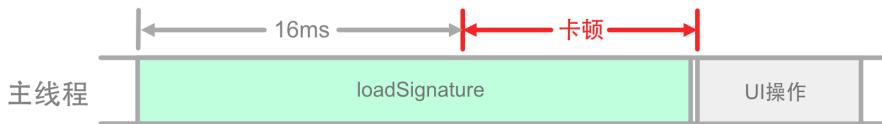
addAppending 是一个同步函数，在它返回之前，运行它的线程将无法执行其他操作，或者说它不能被用来运行其他函数，必须等待当前函数执行完成后这个线程才能做其他事情。



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。

```
// 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)
}
```

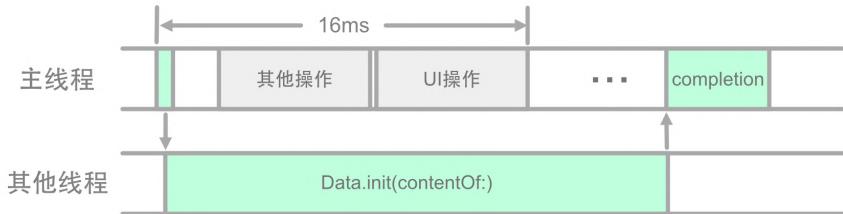


`loadSignature` 最终的耗时超过 16 ms，对 UI 的刷新或操作的处理不得不被延后。在用户观感上，将表现为掉帧或者整个界面卡住。这是客户端开发中绝对需要避免的问题之一。

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：

```
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 {
            // Handle error
        }
    }
}
```

```
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：

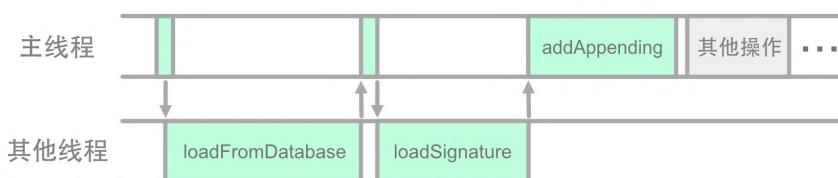
```
func loadFromDatabase(  
    _ completion: @escaping ([String]?, Error?) → Void  
)
```

```
{  
    // ...  
}
```

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 {  
                    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。

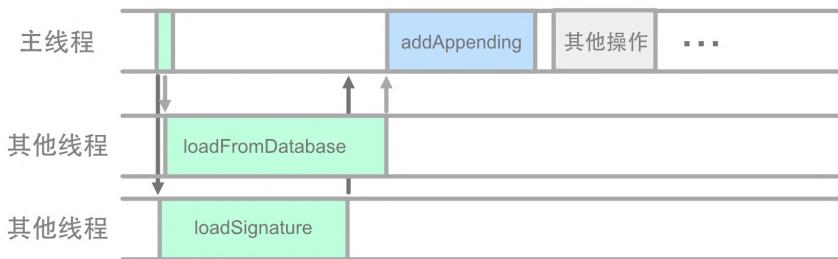
事实上，虽然最后的 `addAppending` 任务同时需要原始数据和签名才能进行，但 `loadFromDatabase` 和 `loadSignature` 之间其实并没有依赖关系。如果它们能够一起执行的话，我们的程序有很大机率能变得更快。这时候，我们会需要更多的线程，来同时执行两个操作：

```
// 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：

```
func loadSignature() async throws → String {  
    fatalError("暂未实现")  
}
```

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：

```
func loadSignature() async throws → String? {
    let (data, _) = try await URLSession.shared.data(from: someURL)
    return String(data: data, encoding: .utf8)
}
```

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：

```
loadFromDatabase { (strings, error) in
    if let strings = strings {
        loadSignature { signature, error in
            if let signature = signature {
```

```
        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：

```
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：

```
struct Task<Success, Failure> where Failure : Error {  
    init(  
        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：

```
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：

```
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：

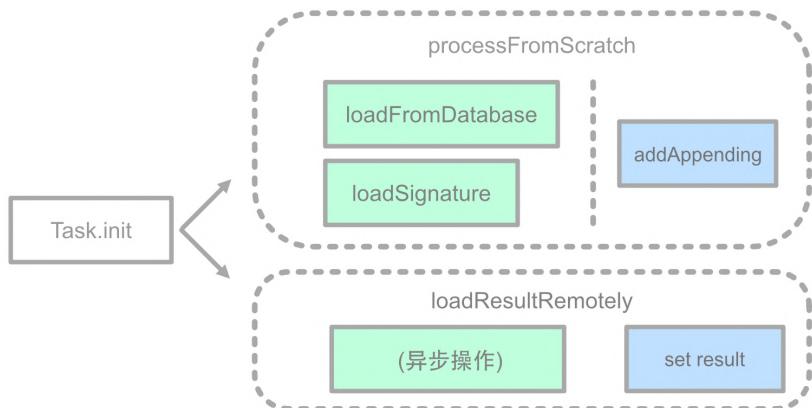
```
func loadResultRemotely() async throws {
    // 模拟网络加载的耗时
    await Task.sleep(2 * NSEC_PER_SEC)
    results = ["data1^sig", "data2^sig", "data3^sig"]
}
```

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)
    }
    // ...
}
```

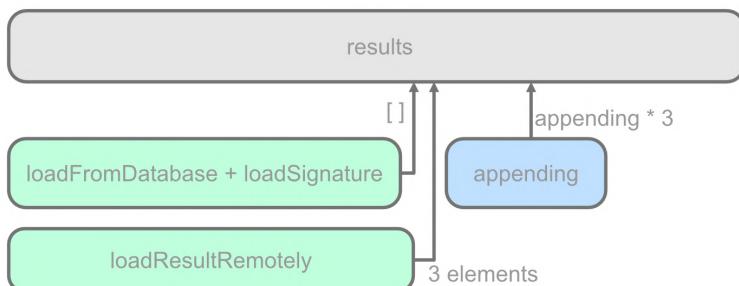
在作为示例的 `loadResultRemotely` 里，我们现在则是直接把结果赋值给了 `results`:

```
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下，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` 设为了空数组，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=0x55a8fdb060c)
```

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：

```
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。

```
// 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：

```
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：

```
// 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。

# 创建异步函数

3

# 异步函数的动机

## 基于回调的异步操作的问题

让我们先回到上一章最开始的基于回调的异步操作的例子，来逐一看看它存在哪些问题。为了简单说明，我们为存在问题的部分标记了序号：

```
func loadSignature(  
    // 5  
    _ completion: @escaping (String?, Error?) → Void  
)  
{  
    // 3  
    guard hasSignature else {  
        // 4  
        return  
    }  
    DispatchQueue.global().async {  
        do {  
            let d = try Data(contentsOf: someURL)  
            // 1  
            DispatchQueue.main.async { // 6  
                completion(String(data: d, encoding: .utf8), nil)  
            }  
        } catch {  
            DispatchQueue.main.async {  
                // 2  
                completion(nil, error)  
            }  
        }  
    }  
}
```

1. **回调地狱**: 多个基于回调的异步操作进行嵌套, 将不可避免地导致回调逐渐内嵌, 不断的缩进使得代码难以阅读和追踪。这里几次 DispatchQueue 的操作, 实际上并没有提供更多的功能, 却已经让代码缩进不可忽视了。想象一下接下来在其中进行更多的异步嵌套, 会导致怎样的效果。
2. **错误处理**: Swift 2 引入了 throw 来处理同步函数中的错误, 但回调函数并没有保有调用栈, 因此其中发生的错误并不能 throw 到调用方。我们必须基于可选值的参数来表示错误, 并指望“有良心”的开发者乖乖去判断 error 是否为 nil 并处理错误。Swift 5 引入了 Result 来将错误进行包装, 一定程度上缓解了这个问题, 但治标不治本: 它依然无法让开发者轻松区分正常路径和错误路径, 对给回的 Result 进行 switch 操作甚至更增加了嵌套深度。
3. **破坏结构**: 在多个异步操作中, 可能需要使用 if 或者 guard 等条件语句来决定是否执行某个操作。这将很快破坏代码的结构: 代码的执行可能不再按照从上向下的顺序, 而要根据条件进行跳转或者提前返回。结合回调地狱, 这些操作也许会被隐藏起来, 难以发现。
4. **错误的 completion 调用**: 由于回调嵌套, 以及失败的代码路径往往被隐藏在正确流程之中, 对 completion 的调用可能会被忘掉(比如在这里, 我们就忘记了 completion)。成功情况也没有办法阻止它被多次调用。默认情况下, 编译器并不会给我们任何警告或帮助(你可以通过在 Build Settings 打开或者在编译时手动加上 -Wcompletion-handler 来激活这类警告)。
5. **异步操作的复杂性**: 因为要提供额外的闭包, 导致了很多时候开发者们, 特别是框架的维护者们, 更倾向于写同步的函数。在明显需要异步操作的地方(比如网络请求等)可能大多数维护者能注意到, 但是在一些“可选”异步的情况下, 往往方法会被首先按照同步的方式书写。这很容易造成意外的堵塞, 也限制了框架的使用者能做的事情, 就算可能使用者们有异步的需求, 但由于框架只提供了同步 API, 他们也无法以异步方式进行处理。将同步操作转变基于回调的异步操作, 可能涉及到复杂的方法签名变更, 会对之后的重构带来麻烦。
6. **隐藏的线程调度**: 对于调用者来说, 回调函数的调用方式是不透明的: 回调函数会在哪个线程被调用是未知的, 使用者往往需要阅读文档或者源码才能确定, 同时这也面临了文档和源码不同步的风险。

异步函数，或者说支持 `async` 和 `await` 的函数，允许开发者使用和同步函数类似的语言结构，这立刻就解决了上面的几乎所有问题：

1. 嵌套的回调可以被写为多个 `async/await`，不再有额外的队列派发所导致的缩进。
2. 异步函数保有调用栈，因此可以使用 `throws` 和正常的返回值来分别表达错误路径和正常路径，调用者需要关心的结果被明确分为两类，且内层错误可以很容易地继续抛出到更外层，以便让合适的调用者进行处理。
3. 使用 `if` 等语句时，行为模式和同步代码一致。这也为调试和测试代码提供了更易用和直观的工具。
4. 异步函数必须有明确的退出路径：要么返回可用的值，要么抛出错误。编译器会保证异步函数结束时，调用者会且仅会收到一个结果，而不像原来忘记调用 `completion` 或者多次调用。
5. 异步函数的函数签名和同步函数更加类似，框架开发者创建异步函数的阻力变小了。只要有异步操作的需求，将同步函数改写为异步函数的难度要远远小于把它改写为回调的难度。这也鼓励了框架和 API 的维护者提供异步函数版本。
6. 开发者不再需要手动进行派发和关心线程调度。虽然在 `await` 后我们依然无法确定线程，但是可以使用 `actor` 类型来提供合理的隔离环境。异步函数和并发底层使用了全新的协作式线程池 (cooperative thread pool) 进行调度，这为异步代码提供更多的优化空间。  
关于这个话题，我们会在本书后半部分关于并发线程调度的章节再详细讨论。

## 线程放弃和暂停点

和同步函数最大的不同在于，异步函数可以放弃自己当前占有的线程。有一些关于异步函数的讨论，会把异步函数的运行理解为：编译器把异步函数切割成多个部分，每个部分拥有自己分离的存储空间，并可以由运行环境进行调度。我们可以把每个这种被切割后剩余的执行单元称作续体 (**continuation**)，而一个异步函数，在执行时，就是多个续体依次运行的结果。

在低层级上，这样的理解会对理解底层实现有所帮助，但是在高层级上这并不必要。我们现在只需要将异步函数想象成和普通函数一样的东西，只不过它具有放弃线程进行暂停，并在稍后再从暂停点继续执行的特殊能力就可以了。这一点其实从 Swift 编译器为异步函数调用生成的 SIL (Swift Intermediate Language) 中间代码也可见一斑。对于下面两个版本的函数：

```
func load() → Int {  
    let initial = 0  
    let result = calculate()  
    return initial + result  
}  
  
func load() async → Int {  
    let initial = 0  
    let result = await calculateAsync()  
    return initial + result  
}
```

它们生成的 SIL 在调用 calculate 时唯一的区别只在于被调用函数是否被标记为 @async (为了方便阅读, 对 SIL 结果进行了简化和调整) :

```
// load()  
%0 = function_ref calculate : $@convention(thin) () → Int  
%1 = apply %0() : $@convention(thin) () → Int  
  
// load() async  
%0 = function_ref calculateAsync : $@convention(thin) @async () → Int  
%1 = apply %0() : $@convention(thin) @async () → Int
```

可以看到, await 在 SIL 中是被完全忽略的: 不论是同步函数还是异步函数, 对它们的调用只是最简单的 apply 指令。异步函数虽然具有放弃线程的能力, 但它本身并不会使用这个能力: 它只会通过对另外的异步函数进行方法调用, 或是通过主动创建续体, 才能有机会暂停。这些被调用的方法和续体, 有时会要求当前异步函数放弃线程并等待某些事情完成(比如续体完结)。当完成后, 本来的函数将会继续执行。

await 充当的角色, 就是标记出一个潜在的暂停点 (**suspend point**)。在异步函数中, 可能发生暂停的地方, 编译器会要求我们明确使用 await 将它标记出来。除此之外, await 并没有其他更多的语义或是运行时的特性。当控制权回到异步函数中时, 它会从之前停止的地方开始继续运行。但是“桃花依旧笑春风”的同时, “人面不知何处去”也会是一个事实: 虽然部分状态, 比如原来的输入参数等, 在 await 前后会被保留, 但是返回到当前异步函数时, 它并不一定还运行在和之前同样的线程中, 异步函数所在类型中的实例成员也可能发生了变化。await 是一个

明确的标识，编译器强制我们写明 `await` 的意义，就是要警示开发者，`await` 两侧的代码会处在完全不同的世界中。

但另一方面，`await` 仅仅只是一个潜在的暂停点，而非必然的暂停点。实际上会不会触发“暂停”，需要看被调用的函数的具体实现和运行时提供的执行器是否需要触发暂停。很多的异步函数不仅仅是异步函数，它们可能是某个 actor 中的同步函数，但作为 actor 的一部分运行，在外界调用时表现为异步函数。Swift 会保证这样的函数能切换到它们自己的 actor 隔离域里完成执行。关于这个话题，我们会在 actor 的章节中继续。

## 转换函数签名

异步函数的目标是使用类似同步的方式来书写异步操作的代码，来修正闭包回调方式的问题，并最终取而代之。随着 Swift 并发在将来的普及，相信今后我们一定会遇到需要将回调方式的异步操作迁移到异步函数的情况，这中间很大部分的工作量会落在如何将闭包回调的代码改写为异步函数这件事上。本节中，我们将总结一些常见的方法和技巧。

## 修改函数签名

对于基于回调的异步操作，一般性的转换原则就是将回调去掉，为函数加上 `async` 修饰。如果回调接受 `Error?` 表示错误的话，新的异步函数应当可以 `throws`，最后把回调参数当作异步函数的返回值即可。

我们在前面的章节中应该已经见到一些这种转换了。再举几个具体的例子：

```
func calculate(input: Int, completion: @escaping (Int) → Void)
// 转换为
func calculate(input: Int) async → Int

func load(completion: @escaping ([String]?, Error?) → Void)
// 转换为
func load() async throws → [String]
```

不难看出，在表现异步操作语义时，异步函数要比闭包回调的版本简洁多了。

当遇到可抛出的异步函数时，编译器要求我们将 `async` 放在 `throws` 前；在这类函数的调用侧，编译器同样做出了强制规定，要求将 `try` 放在 `await` 之前。实际上，`await` 和 `try` 仅仅只是辅助编译器和开发者的关键字，并没有其他语义，也不影响编译器生成的代码，因此这两对先后顺序只是单纯的人为规定，并没有太大实际意义。要说对开发者有什么帮助的话，大概是把顺序定死，能够避免它演变为类似“空格 vs Tab”这样的史诗级战争，为开发者们的人生节省不少时间。

## 带有返回的情况

有些情况下，带有闭包的异步操作函数本身也具有返回值，这种情况会相对比较棘手。比如 `URLSession` 中的一些函数就是这样：

```
// 同时具有 completionHandler 和返回值 (URLSessionDataTask)
class URLSession {
    func dataTask(
        with url: URL,
        completionHandler:
            @escaping (Data?, URLResponse?, Error?) -> Void
    ) -> URLSessionDataTask
}
```

`dataTask` 方法在接受 `completionHandler` 回调的同时，同步地返回一个 `URLSessionDataTask`，使用者可以通过调用 `URLSessionDataTask` 上的 `cancel` 来取消运行中的任务。这种情况下，返回的 `URLSessionDataTask` 肯定不能简单地写在新的异步函数的返回里：异步函数的返回值是经过暂停点后的异步执行结果，它在语义上和同步函数的返回值完全不同。

在 `URLSession` 中，Apple 选择了为异步形式创建了新的 API，在那里 `URLSessionDataTask` 的返回值被完全忽略了，比如：

```
func data(
    from url: URL,
    delegate: URLSessionTaskDelegate? = nil
) async throws -> (Data, URLResponse)
```

在 URLSessionDataTask 这个特例下，这没有造成太大问题。URLSessionDataTask 需要承担的最大的任务，就是取消网络请求。异步函数都是运行在某个任务环境中的，因此可以通过取消任务来间接取消运行中的网络请求。虽然这需要一些额外的努力，但是是可以优雅地做到的：我们会在结构化并发的部分里对任务取消和具体的做法进行更多解释。

其他的对于 URLSessionDataTask 的配置和使用，比如设置 `priority` 或者确认 `state`，可以通过异步函数的第二个参数所定义的 `delegate` 来完成。

如果原来的闭包回调函数的返回值只是一个简单的基本类型的话，也许直接通过 `inout` 参数，在暂停点之前获取这个值，也是一种可选方案。

```
func syncFunc(completion: @escaping (Int) → Void) → Bool {
    someAsyncMethod {
        completion(1)
    }
    return true
}

func asyncFunc(started: inout Bool) async → Int {
    started = true
    await someAsyncMethod()
    return 1
}
```

但是，如果是从 Task 域外传递一个 `inout Bool` 的话，编译器将提示错误，所以这并不是一个完备的解决方案：

```
var started = false
Task {
    let value = await asyncFunc(started: &started)
    print(value)
}
print("Started: \(started)")

// 编译错误:
```

```
// Mutation of captured var 'started' in
// concurrently-executing code
```

使用 Task 将开启一个并发任务，在并发任务中改变任务域外的 started 值，是存在数据安全风险的：新开启的异步任务和 Task 之外的同步代码同时访问 started 可能会造成问题。这部分内容涉及到 @Sendable 和如何确保数据安全，在结构化并发和 actor 模型的章节中，我们会再次回顾这个问题。

## @completionHandlerAsync

异步函数具有极强的“传染性”：一旦你把某个函数转变为异步函数后，对它进行调用的函数往往都需要转变为异步函数。为了保证迁移的顺利，Apple 建议进行从下向上的迁移：先对底层负责具体任务的最小单元开始，将它改变为异步函数，然后逐渐向上去修改它的调用者。

为了保证迁移的顺利，一种值得鼓励的做法是，在为一个原本基于回调的函数添加异步函数版本的同时，暂时保留这个已有的回调版本。这样可以保证项目始终能够编译，API 的使用者能进行逐步迁移。另外，如果是框架维护者，还需要考虑到框架的用户并不一定具有迁移到 Swift 并发的条件，或者因为某种原因不得不继续使用原本的回调版本。这时，同时保留原来的回调版本和新的异步函数版本，就成为了必须项。

为了让迁移顺利，我们可以为原本的回调版本添加 @completionHandlerAsync 注解，告诉编译器存当前函数存在一个异步版本。这样，当使用者在其他异步函数中调用了这个回调版本时，编译器将提醒使用者可以迁移到更合适的异步版本：

```
@completionHandlerAsync("calculate(input:)")
func calculate(input: Int, completion: @escaping (Int) → Void) {
    completion(input + 100)
}

func calculate(input: Int) async → Int {
    return input + 100
}
```

`@completionHandlerAsync` 和 `@available` 标记很类似。它接收一个字符串，并在检测到被标记的函数在异步环境下被调用时，为编译器提供信息，帮助使用者用“fix-it”按钮迁移到异步版本。和一般的 `@available` 的不同之处在于，在同步环境下 `@completionHandlerAsync` 并不会对 `calculate(input:completion:)` 的调用发出警告，它只在异步函数中进行迁移提示，因此提供了更为准确的信息。

```
func syncFunc() {  
    calculate(input: 100, completion: { print($0) })  
    // 不会提示警告  
}  
  
func asyncFunc() async {  
    calculate(input: 100, completion: { print($0) })  
    // 警告：  
    // Consider using asynchronous alternative function  
}
```

如果回调版本的函数并非使用最后一个参数接受回调的话，还可以在标记里追加 `completionHandlerIndex` 对回调参数的位置进行指定。比如，上面对 `calculate` 的标记等效于：

```
@completionHandlerAsync(  
    "calculate(input:)",  
    completionHandlerIndex: 1  
)  
func calculate(input: Int, completion: @escaping (Int) → Void) {  
    completion(input + 100)  
}
```

不论是在迁移过程中，还是需要同时维护回调和异步函数两套 API 时，使用 `@completionHandlerAsync` 来标记原来的回调版本，可以帮助迁移过程更加顺利。

## 使用续体改写函数

我们解决了函数签名的问题，接下来看看函数体本身要如何“异步化”。

在异步函数被引入之前，处理和响应异步事件的主要方式是闭包回调和代理 (delegate) 方法。可能你的代码库里已经大量存在这样的处理方式了，如果你想要提供一套异步函数的接口，但在内部依然复用闭包回调或是代理方法的话，最方便的迁移方式就是捕获续体并暂停运行，然后在异步操作完成时告知这个续体结果，让异步函数从暂停点重新开始。

Swift 提供了一组全局函数，让我们暂停当前任务，并捕获当前的续体：

```
func withUnsafeContinuation<T>(
    _ fn: (UnsafeContinuation<T, Never>) → Void
) async → T

func withUnsafeThrowingContinuation<T>(
    _ fn: (UnsafeContinuation<T, Error>) → Void
) async throws → T

func withCheckedContinuation<T>(
    function: String = #function,
    _ body: (CheckedContinuation<T, Never>) → Void
) async → T

func withCheckedThrowingContinuation<T>(
    function: String = #function,
    _ body: (CheckedContinuation<T, Error>) → Void
) async throws → T
```

普通版本和 Throwing 版本的区别在于这个异步函数是否可以抛出错误，如果不可抛出，那么续体 Continuation 的泛型错误类型将被固定为 Never。在结构化并发的部分 API 中 (比如 withTaskGroup 和 withThrowingTaskGroup)，我们也可以看到类似的设计。

## 续体 resume

在某个异步函数中调用 with\*Continuation 后，这个异步函数暂停，函数的剩余部分作为续体被捕获，代表续体的 UnsafeContinuation 或 CheckedContinuation 被传递给闭包参数，而这

一个闭包也会在当前的任务上下文中立即运行。这个 Continuation 上的 resume 函数，在未来必须且仅需被调用一次，来将控制权交回给调用者。

比如对于下面的闭包回调函数：

```
func load(completion: @escaping ([String]?, Error?) → Void)
```

典型情况下，利用 withUnsafeThrowingContinuation 进行包装：

```
func load() async throws → [String] {
    try await withUnsafeThrowingContinuation { continuation in
        load { values, error in
            if let error = error {
                continuation.resume(throwing: error)
            } else if let values = values {
                continuation.resume(returning: values)
            } else {
                assertFailure("Both parameters are nil")
            }
        }
    }
}
```

resume(throwing:) 和 resume(returning:) 分别对应了发生错误的情况和正确返回的情况，当 continuation 上的这两者任一被调用时，整个异步函数要么抛出错误，要么返回正常值。

Unsafe 和 Checked 版本的区别在于是否对 continuation 的调用状况进行运行时的检查。continuation 必须在未来继续，只是一个开发者和编译器的约定。Unsafe 的版本不进行任何检查，它假设开发者会正确使用这个 API：如果 continuation 没能继续（也就是 continuation 在被释放前，它上面的任意一个 resume 方法都没有调用），那么异步函数将永远停留在暂停点不再继续；反过来，如果 resume 被调用了多次，程序的运行状态将出现错误。

和 Unsafe 的版本稍有不同，Checked 的版本能稍微给我们一些提示。在没能继续的情况下，运行时会在控制台进行输出：

```
func load() async throws → [String] {  
    try await withCheckedThrowingContinuation { continuation in  
        load { values, error in  
            // 什么都不做  
        }  
    }  
}  
  
// 控制台输出：  
// SWIFT TASK CONTINUATION MISUSE: load() leaked its continuation!
```

在调用 `resume` 多次时，这个错误将产生崩溃，以避免像 `Unsafe` 版本那样进入到无法预测的状态：

```
func load() async throws → [String] {  
    try await withCheckedThrowingContinuation { continuation in  
        load { values, error in  
            if let values = values {  
                // 多次调用 `resume`  
                continuation.resume(returning: values)  
                continuation.resume(returning: values)  
            }  
        }  
    }  
}  
  
// 崩溃，错误信息：  
// Fatal error: SWIFT TASK CONTINUATION MISUSE: load()  
// tried to resume its continuation more than once, returning ...
```

控制台警告和错误输出中的方法名“`load()`”来自于 `withCheckedThrowingContinuation` 的第一个参数： `function`，它的默认值是

#function, 也即调用和产生续体的函数名。这个参数帮助我们在调试时能有更容易理解的信息。

可能你已经猜到了，由于 Checked 的一系列特性都和运行时相关，因此对续体的使用情况进行检查(以及存储额外的调试信息)，会带来额外的开销。因此，在一些性能关键的地方，在确认无误的情况下，使用 Unsafe 版本会提升一些性能。因为除了 Checked 和 Unsafe 之外，两个 API 在语法上并没有区别，所以按照 Debug 版本和 Release 版本的编译条件进行互换也并不困难。不过需要记住的是，就算使用 Checked 版本，也不意味着万事大吉，它只是一个很弱的运行时检查：对于没有调用 resume 的情况，虽然异步函数会在续体超出捕获域后自动继续，但是没有 resume 的任务依然被泄漏了；对于多次调用 resume 的情况，运行时崩溃的严重性更是不言而喻。无论如何，它们依然是程序运行的重大错误，Checked 能做的只是帮助我们更容易地发现这些错误。

## 续体暂存

除了在回调版本的异步代码中使用外，我们也可以把捕获到的续体暂存起来，这种方式很适合将 delegate 方式的异步操作转换为异步函数：

```
protocol WorkDelegate {
    func workDidDone(values: [String])
    func workDidFailed(error: Error)
}

class Worker: WorkDelegate {
    var continuation: CheckedContinuation<[String], Error>?

    func doWork() async throws → [String] {
        try await withCheckedThrowingContinuation({ continuation in
            self.continuation = continuation
            performWork(delegate: self)
        })
    }
}
```

```
func workDidDone(values: [String]) {
    continuation?.resume(returning: values)
    continuation = nil
}

func workDidFailed(error: Error) {
    continuation?.resume(throwing: error)
    continuation = nil
}
}
```

很多时候，`delegate` 方法可能被调用不止一次，但是作为 `continuation` 来说，不论成败，它只支持一次 `resume` 调用。在上面的代码中，我们通过 `resume` 调用后将 `self.continuation` 置为 `nil` 来避免重复调用。根据具体情况，你可能可能会需要选择不同的处理方法。如果一个续体需要被多次调用，并产生一系列值，我们会需要涉及到 `AsyncSequence` 和 `AsyncStream` 的使用，在本书后面，我们为这个话题也预留了讨论章节。

## 续体和 Future

如果你对 Combine 比较熟悉，也许已经隐约感受到，续体和 Future 有一些相似：Future 通过提供一个 Promise 来接受未来的 `Result<Output, Failure>` 值，并提供给订阅者，而续体的行为模式也一样，甚至续体版本也准备了接受 `Result` 类型的重载：

```
extension CheckedContinuation {
    func resume(with result: Result<T, E>)
}
```

在一定程度上，笔者认为 `async` 函数（或者更准确说，由续体转换的异步函数）可以取代 Future：同样是在返回一个未来的值，`async` 显然提供了更加简洁的写法。相对于 Combine 基于订阅的使用方式和 `Scheduler` 决定的线程模型，直接使用异步函数需要操心的地方要少很多。但是续体异步函数和 Future 依然有不同。异步函数必定在一定的任务上下文之中执行，这个上下文决定了任务的取消状态、优先级等；单个 Future 如果不和 Combine 框架中的其他 Publisher 或者 Operators 结合（combine）使用的话，它能提供的特性远远不及异步函数丰富。

我们在 AsyncSequence 的部分也会继续这个话题，来讨论 Swift 并发中的异步序列和 Combine 框架的对比。在那里，我们会得出更多结论。

# Objective-C 自动转换

## Objective-C 到 Swift

虽然 Objective-C 中并没有异步函数的语言特性，但不论是 Foundation 还是 UIKit 中，基于回调函数的异步 API 并不少见。如果满足一定的书写规则，Swift 在导入 Objective-C 方法时，可以自动将它们作为异步函数导入。这可以让 Objective-C 框架中已有的异步 API 立即用在 Swift 异步的上下文中。

我们每天使用的一些函数就是现成的例子，比如，在通过 PHPhotoLibrary 检查和申请相册权限的类方法：

```
+[PHPhotoLibrary requestAuthorizationForAccessLevel:handler:]
```

在 Swift 5.5 前，导入到 Swift 中时，这个函数会暴露一个基于回调的接口：

```
extension PHPhotoLibrary {
    class func requestAuthorization(
        for accessLevel: PHAccessLevel,
        handler: @escaping (PHAuthorizationStatus) → Void
    )
}
```

如果你查看 Swift 5.5 中的 Swift 接口，会发现在原来的 handler 版本下面，多了一个异步函数的版本：

```
extension PHPhotoLibrary {
    class func requestAuthorization(
        for accessLevel: PHAccessLevel
    ) async → PHAuthorizationStatus
```

```
}
```

其他类似的方法还有很多。如果一个 Objective-C 函数存在函数参数，且该参数的返回值和整个函数本身的返回值类型都为 void 的话，该 Objective-C 函数就被推断为在执行潜在的基于回调的异步操作。对于这类潜在异步回调，如果它的闭包参数的参数名包含特定关键字，比如 completion, withCompletion, completionHandler, withCompletionHandler, completionBlock, withCompletionBlock 等，那么这个闭包的输入将被提取出来，自动映射成为异步函数的返回值。

上面对 requestAuthorization(for:handler:) 的转换是一个不那么典型的例子，因为它的参数名是 handler。这时候，为了能够进行转换，我们需要给编译器一些帮助。如果你查看 Objective-C 版本的头文件，可以看到在它后面被标注了 NS\_SWIFT\_ASYNC(2)。这告诉了编译器应该把第二个参数看作是传递结果的闭包。在调用时，withUnsafeContinuation (或者它的可抛出版本) 会被用来包装原来的闭包版本的方法，提供一个异步方法的版本：

```
let status =  
    await PHPhotoLibrary.requestAuthorization(for: .readWrite)
```

在编译器加持下，它将等效于类似这样的代码：

```
await withUnsafeContinuation { continuation in  
    PHPhotoLibrary.requestAuthorization(for: .readWrite) { status in  
        continuation.resume(returning: status)  
    }  
}
```

在某些特定情况下，你可能会不想要这样的自动转换，通过为 Objective-C 的方法添加 NS\_SWIFT\_DISABLE\_ASYNC，可以避免编译器为满足条件的方法生成 async 版本的 Swift 接口。比如 UIView 中常用的创建动画的方法：

```
// UIView.h  
+ (void)animateWithDuration:(NSTimeInterval)duration  
    animations:(void (^)(void))animations  
    completion:(void (^)(BOOL finished))completion  
NS_SWIFT_DISABLE_ASYNC;
```

在 Swift 中，编译器将不会为这个它生成异步函数的版本。这是 Apple 有意为之的：对于像是创建动画这样的大部分 UI 操作来说，我们希望的是提交动画，然后立即继续进行其他操作，而不会是等待动画结束后再继续其他操作。如果这个函数也提供了异步版本，那么大概率我们会这样使用它：

```
Task {  
    let finished = await UIView.animate(  
        withDuration: 1.0,  
        animations: { /* animation */ })  
    print("Animation done: \(finished)")  
}  
print("Other operations.")
```

相比起等效的非异步版本：

```
UIView.animate(withDuration: 1.0) {  
    /* animation */  
} completion: { finished in  
    print("Animation done: \(finished)")  
}  
print("Other operations.")
```

额外的 Task 反而让事情变得更复杂了。这也是 UIView 和 UIViewController 上大部分 UI 相关的操作明确标明了不进行异步函数转换的原因。

关于 Objective-C 向 Swift 进行异步转换的标记，还有一些其他的形式或参数，用来在更加精细的粒度上为编译器完成异步转换提供指导。如果你是 Objective-C 代码的维护者，并希望自定义 Swift 异步接口，可以参考 [LLVM 中的相关文档](#)和关于 Objective-C 中的并发转换提案中的相关内容，了解更多关于 Swift 异步标记的细节。

## Swift 到 Objective-C

反过来，由 Swift 定义的异步函数，也可能会想要在 Objective-C 的代码中使用。由于 Swift 异步函数的表意相对明确，这里不存在任何猜测。普通的 Swift 函数可以通过 @objc 暴露给

Objective-C。异步函数也遵守同样的规则：当一个 Swift 中的 `async` 函数被标记为 `@objc` 时，它在 Objective-C 中会由一个带有 `completionHandler` 的回调闭包版本表示。比如：

```
func calculate(input: Int) async → Int
```

在 Objective-C 中，被导入为：

```
- (void)calculate:(NSInteger _Nonnull)input  
completionHandler:(void (^ _Nonnull)(NSInteger))completionHandler;
```

和 Objective-C 向 Swift 的转换一样，编译器也会为 Swift 到 Objective-C 的转换合成额外的实现。由于 Objective-C 中其实不存在可用的 Task 上下文环境，在实际调用 Swift 版本的异步函数前，会使用 `Task.detached` 创建一个完全游离的任务运行环境。从 Objective-C 一侧，调用这个方法时实际进行工作类似于：

```
@objc func calculate(  
    input: Int,  
    completionHandler: @escaping (Int) → Void)  
{  
    Task.detached {  
        let value = await calculate(input: input)  
        completionHandler(value)  
    }  
}
```

如果原来的异步函数可以抛出，那么生成的 Objective-C 接口中的回调闭包也将具有两个参数：一个表示执行成功的实际结果值，另一个表示异常时抛出时的具体错误。

## Async getter

### 异步只读属性

我们解决了将函数标记为异步的问题，但对于另一种常见的“函数”，还没有进行定义：那就是计算属性。

除了不接受参数以外，其实计算属性，特别是 getter，和一般函数非常类似：

```
class File {  
    // func getSize() -> Int { return 1024 }  
    var size: Int { return 1024 }  
}
```

一般来说，我们都倾向于不在 getter 中进行复杂的耗时工作。但是编译器并没有阻止我们在 getter 里进行阻塞线程的长时间操作。不论这是由于代码最初书写时候的情况和现在相比已经沧海桑田，年久失修，还是因为 getter 中所使用的别的 API 发生了变化，总之“不在 getter 中进行耗时操作”的假设是人为且脆弱的，编译器并没有对此提供任何保证。一旦这种情况发生，对于这个 getter 属性的访问，就可能导致卡顿：

```
class File {  
    var size: Int {  
        return heavyOperation()  
    }  
  
    func heavyOperation() -> Int {  
        // 很慢的操作，比如大量 I/O  
    }  
}
```

一种解决方式是放弃使用 getter，转而使用回调函数：

```
func getSize(completionHandler: @escaping (Int) -> Void) {  
    // ...  
}
```

但是 getSize 的写法显得非常不 Swift，并带来了一些重复和模板代码。如果类型 API 中同时存在 size getter 和 getSize(completionHandler:) 的话，我想绝大部分使用者会跳过你辛苦准备的文档和苦口婆心的劝告，无脑选择更“简单”的 getter 计算属性。

除了无法异步操作外，现有 getter 还有另一个问题，那就是无法抛出错误。诚然，我们可以使用返回 nil 可选值来表示错误。这种方法可以解决部分问题，但是如果对于 getter 返回值原本就可能使用 nil 表示空值的情况，我们就无法分辨 getter 到底是成功取到了空，还是取值过程中发生了错误。另一种选择是返回 Result 来表征错误，不过这样做的话调用方就需要对 Result 的结果进行判断，很快代码的逻辑会被无关核心部分的额外处理淹没，这也不是理想的解决方案。

在 Swift 5.5 中，getter 得到的强化，它可以使用 `async` 和 `throws` 修饰了。上面的 `File.size` 可以写成 `get async throws` 的异步 getter：

```
class File {
    var size: Int {
        get async throws {
            if corrupted {
                throw FileError.corrupted
            }
            try await heavyOperation()
        }
    }

    func heavyOperation() async throws -> Int {
        // ...
    }
}
```

在使用上，和普通的异步函数类似。因为 `async` 的 `size` 现在代表了一个潜在的暂停点，因此对它的调用必须发生在异步环境中，并使用 `await`：

```
func reportFileSize() async throws {
    let file = File()
    print("File size is: \(try await file.size)")
}
```

现在，我们可以明确地通过 `async getter` 让编译器提示使用者，这个 `getter` 可能会耗费较长时间，并避免意外造成的阻塞了。

异步 `getter` 在 `actor` 模型中非常常用：`actor` 的成员变量是被隔离在 `actor` 中的，外部对它的获取将导致隔离域切换，这是一个潜在的暂停。对于从隔离域外对 `actor` 中成员变量的读取，编译器将为我们合成对应的异步 `getter` 方法。

除了 `getter` 以外，通过下标的读取方法也得到了同样的特性，来提供类似的 `async` 和 `throws` 的支持：

```
class File {  
    subscript(_ attribute: AttributeKey) → Attribute {  
        // 比如 await file[.readonly] = true  
        get async {  
            let attributes = await loadAttributes()  
            return attributes[attribute]  
        }  
    }  
}
```

## 状态依赖

在本章中我们已经提到过，`await` 表示了潜在的暂停点，它在程序中并没有实际的语义，更多的是对开发者的一种警示：在 `await` 前后，程序可能会运行在两个世界中。在支持异步后，`getter` 可能会代码中的其他状态“纠缠”起来，在 `await` 前后我们对某个状态进行假设时，需要特别小心。

我们通过下面这个例子来进行解释：

```
var loaded: Bool = false  
  
var shouldLoad: Bool {  
    get async {  
        if !loaded {
```

```
    await prepare()
    return true
}
return false
}
}

func load() {
loaded = true
// ...
}
```

异步 getter 的 `shouldLoad` 应该在 `loaded` 为 `false` 时返回 `true`, 告诉调用者此时应当进行加载。但是 `await prepare()` 将使程序暂停。和同步函数不同, `loaded` 的状态可能会在 `prepare` 执行期间发生变化, 比如在暂停期间 `load()` 函数被外部调用了。“正是由于 `loaded` 原本为 `false`, 所以程序才进到了 `await prepare()` 条件语句”的这个假设, 在 `await` 之后可能是不成立的。如果在准备期间 `load` 被调用了, 那其实此时 `loaded` 已经为 `true`, 这时候是否还应该按照“`loaded` 为 `false`”的假设, 为 `shouldLoad` 返回 `true` 呢? 这需要成为开发者深入考虑的问题。

对于上面的特定的例子, 也许再次检查 `loaded` 并根据最新值进行返回会是更好的选择:

```
var shouldLoad: Bool {
get async {
if !loaded {
await prepare()
}
return !loaded
}
}
```

但是在 `await` 后再次参照状态值, 也并不是永远正确的选择。在 actor 模型部分关于可重入的话题中, 我们会看到更多的例子, 并在那边对 actor 隔离域下的类似行为进行讨论。

## setter

async 和 throws 的支持，现在只针对属性 getter 和下标读取。对于计算属性的 setter 和下标写入，异步行为暂时还不支持。这并非因为技术上的不可实现，更多的是对于复杂度的考虑，因此将它们排入了较低优先级。

这里提到的复杂度，在于各种 setter 的附加行为。相对于 getter 来说，setter 需要考虑的事情要更多。想要为 setter 定义 async 的话，需要考虑的内容至少包括 inout 的行为， didSet 和 willSet 应该在何时调用，属性包装 (Property wrapper) 要怎么处理等话题。为 getter 定义异步行为是相对比较简单，而且能为实际编程提供很大帮助的“高性价比”努力。相对起来，对 setter 的支持则被延后了。

如果只是单纯地需要对某个属性以异步方式进行设置，以便在设置属性的同时执行某些耗时操作，我们可以直接暴露一个异步函数：

```
var value: String

func setValue(v: String) async throws {
    await someOtherAsyncWork()
    try checkCanWrite()
    value = v
}
```

这可能会带来部分模板代码，但是最大限度保留了兼容性：和 setter 相关的 hook 方法 (willSet, didSet) 以及围绕 setter 的其他特性，都不会发生变化。在未来的 Swift 版本中，也许我们能看到内建的对 setter 和下标写入的异步支持。

## 小结

本章中我们从异步函数相比较同步函数的优势开始着手，逐渐看到了各种异步函数的书写方法。将一个同步世界中的函数转换为异步函数，是将已有项目逐渐迁移到 Swift 并发模型过程中，所不可欠缺的技能。由小到大，从下向上，对同步函数的异步操作逐渐改写，利用异步函数的优势简化代码，最终将它与项目其他部分整合，将为程序的稳定性和兼容性提供重要保障。

再次强调，异步函数并不等于 Swift 并发，它甚至不是 Swift 并发特性想要达到的主要目标。在整个并发模型中，它只负责提供一套更简洁的语法，它充当了工具类的角色。我们在后面的章节中，可以看到这套异步函数的语法是如何在线程协调、任务取消和数据安全上大放异彩的。本章中所涉及到的，仅仅只是用来陈列 Swift 并发这颗璀璨明珠的基座。

# 异步序列

4

async/await 定义的异步函数，提供了一种直观的方式定义“未来”：可以认为异步函数将会返回未来某个时间点的值。如果我们希望表达的不是未来某一个时间点的值，而是未来一系列多个时间点的值，会需要使用一种新的表达方式，那就是异步序列（**Async Sequences**）。

在同一个异步函数中，await 的次数是不受限制的。也就是说，一个异步函数的执行，可以暂停多次。每次暂停异步函数的剩余部分会形成新的续体，并在暂停完成后等待底层调度将控制权重新返回当前异步函数。因此，一个异步函数是可以获取到多个未来时间点的值的。当这些值具有某种联系，成为一个序列时，我们可以将它们进行抽象，并用一个协议（protocol）来规定它们的重要特性。这个协议就是异步序列 AsyncSequence。异步序列在 Apple SDK 的一些地方都有体现，也是结构化并发的构成要件之一。本章中我们将对异步序列进行介绍，探索它的使用和实现方式，并尝试进行一些自定义。

## 同步序列和异步序列

对于使用 Swift 的开发者来说，序列（Sequence）应该是非常熟悉的概念了。它是 Swift 标准库中一个非常基本的协议，用来定义可以通过逐个迭代列举（iterate）而得到的一系列值。在同步世界中，Sequence 事实上做的事情就是定义如何产生一个迭代器（Iterator）：

```
public protocol Sequence {
    associatedtype Iterator : IteratorProtocol
    func makeIterator() -> Self.Iterator
}
```

Iterator 类型需要遵守的 IteratorProtocol 更加简单，它只需要一个 next 方法：在被调用时，如果序列中还有值，那么就返回这个值，如果序列已经结束，则返回 nil：

```
public protocol IteratorProtocol {
    associatedtype Element
    mutating func next() -> Self.Element?
}
```

一般来说，只有在实现一个自定义序列类型的时候，才需要关心迭代器。除此之外，你几乎不会直接去使用它，因为 for 循环才是我们在遍历序列时最常用的方式。定义序列的最直接的好处，是让编译器在处理 for 循环时，可以将它简单地视为一个语法糖。对于这样的代码：

```
for element in someSequence {  
    doSomething(with: element)  
}
```

本质上来说，Swift 编译器会将它转换为：

```
var iterator = someSequence.makeIterator()  
while let element = iterator.next() {  
    doSomething(with: element)  
}
```

大部分序列的迭代器都只产生有限序列，因此 `iterator.next()` 最终会返回 `nil`，并让 `while` 终止。但是也完全可以实现一个无限序列，永无尽头的序列在编程世界中并不罕见，最简单的例子之一就是后一个数字等于前两个数字之和的斐波那契数列 ([Fibonacci sequence](#)) (0, 1, 1, 2, 3, 5, 8, 13, ...)。虽然我们本章的重点不是同步序列，但是为了作为异步序列的对比参考，还是把一个同步的斐波那契数列的实现方式写在下面作为示例：

```
struct FibonacciSequence: Sequence {  
    struct Iterator: IteratorProtocol {  
        var state = (0, 1)  
  
        mutating func next() → Int? {  
            let upcomingNumber = state.0  
            state = (state.1, state.0 + state.1)  
            return upcomingNumber  
        }  
    }  
  
    func makeIterator() → Iterator {  
        .init()  
    }  
}
```

Sequence 提供的只是一个用于创建迭代器的入口。而实际持有序列信息的角色，是迭代器本身。

在 FibonacciSequence 中，迭代器对 next 的计算十分简单。但是同步序列的 next() 方法和其他同步方法一样，是可能会造成阻塞的。如果我们能将 next 写为异步函数，那么在获取序列中下一个元素时，这个迭代器将有能力放弃自己执行的线程，从而避免阻塞。这种支持异步函数方式求值的序列，就是异步序列。

## 异步迭代器

异步序列也是由一个协议定义的：

```
public protocol AsyncSequence {
    associatedtype AsyncIterator : AsyncIteratorProtocol
    func makeAsyncIterator() -> Self.AsyncIterator
}
```

除了要求异步版本的迭代器，它在结构上和 Sequence 是完全一致的。你可以已经猜到了，异步迭代器的协议 AsyncIteratorProtocol 中，除了 next 是一个异步函数以外，其他也和同步版本的 IteratorProtocol 一样：

```
protocol AsyncIteratorProtocol {
    associatedtype Element
    mutating func next() async throws -> Self.Element?
}
```

next 在这里被标记为 `async` 并具备 `throws` 的能力，这些定义赋予了异步序列更多的可能性。比如，我们现在不打算在本地计算斐波那契数列的下一个数，而是通过某个网络 API 去获取这个数。这时 next 将涉及潜在的暂停点：

```
struct AsyncFibonacciSequence: AsyncSequence {
    typealias Element = Int
    struct AsyncIterator: AsyncIteratorProtocol {
```

```
var currentIndex = 0

mutating func next() async throws → Int? {
    defer { currentIndex += 1 }
    return try await loadFibNumber(at: currentIndex)
}

func makeAsyncIterator() → AsyncIterator {
    .init()
}
}
```

对于涉及到的 `loadFibNumber`, 为了简化, 我们用一个 `Task.sleep` 来模拟这个耗时操作。

```
func loadFibNumber(at index: Int) async throws → Int {
    // 使用 Task.sleep 模拟 API 调用 ...
    await Task.sleep(NSEC_PER_SEC)
    return fibNumber(at: index)
}

func fibNumber(at index: Int) → Int {
    if index == 0 { return 0 }
    if index == 1 { return 1 }
    return fibNumber(at: index - 2) + fibNumber(at: index - 1)
}
```

## for await

为异步序列定义类似的结构, 其主要目的是为了能让开发者用类似的 `for...in` 的语法, 简单地对异步序列中的元素进行迭代。不过和同步序列不同的是, 异步序列中的获取每个元素时都是一个潜在暂停点, 因此需要 `await` 明确标记。对上面定义的异步斐波那契序列的使用如下:

```
let asyncFib = AsyncFibonacciSequence()
```

```
for try await v in asyncFib {  
    if v < 20 {  
        print("Async Fib: \$(v)")  
    } else {  
        break  
    }  
}
```

这段代码将从 0 开始列举小于 20 的斐波那契数。因为在获取每个元素时我们进行了等待，因此可以在控制台上看到每隔一秒才会进行一次输出。

和同步序列一样，编译器在遇到 `for await` 时，依然会将它“翻译”成 `while let` 的版本：此时 `next` 是异步函数的事实，强制我们使用 `await` 对它进行调用。上面的代码等效于：

```
let asyncFib = AsyncFibonacciSequence()  
var iter = asyncFib.makeAsyncIterator()  
while let v = try await iter.next() {  
    // ...  
}
```

## 异步迭代器的值语义

对于上面的 `AsyncFibonacciSequence` 的使用，有一个有趣的问题，那就是在 `v < 20` 的条件不再满足，序列迭代停止后，如果再次开始迭代序列，会是怎样的结果？也就是说，下面的代码会输出什么？

```
let asyncFib = AsyncFibonacciSequence()  
for try await v in asyncFib {  
    if v < 20 {  
        // continue  
    } else {  
        break  
    }  
}
```

```
for try await v in asyncFib {  
    print("Next value: \$(v)")  
}
```

它会是从头开始的第一个数字 0 呢？还是会是数列中大于 20 的后一个数字 34？（对于数列..5, 8, 13, 21, 34 ..，当迭代到 21 时跳出第一个 for await 循环，数列中的下一个数字应该是 34）。

我们知道，当每次使用 for 语句开始迭代时，makeAsyncIterator 函数都会被调用，返回一个迭代器。在当前的 AsyncFibonacciSequence 实现中，每次的迭代器都是全新的，因此在调用 next 进行迭代时，迭代器总是从最初的状态开始。所以上面的代码会输出 0。这种情况下，序列满足值语义 (value semantic)：任意的两次迭代互相不会产生影响，它们是独立存在的。到目前为止，我们看到的迭代器都满足值语义。

## 引用语义迭代器和单次遍历

但是，不排除有一些情况下，我们会希望数列从中断的地方继续执行，而不是从头开始。比如 loadFibNumber 的时候网络出现了暂时的错误，而在我们处理完这个错误后，发现对序列的加载是可以继续进行的，我们可能会希望获取序列中的下一个数字，而非从头开始。这类需求可以进一步引申到像是 UI 产生的事件流、下载的断点续传、I/O 读写等等。这类问题在日常中其实并不罕见，这时候我们就需要序列只能被遍历一次，它需要具有引用语义 (reference semantic)。

要将 AsyncFibonacciSequence 按照引用语义进行改写，最简单的方式就是让 makeAsyncIterator 返回同一个 iterator。为了做到这一点，我们可以将序列和迭代器都改成 class，并在序列中持有一个 iterator：

```
// 将序列改为 class  
class ClassFibonacciSequence: AsyncSequence {  
  
    // 将迭代器改为 class  
    class AsyncIterator: AsyncIteratorProtocol {  
        var currentIndex = 0  
        // ...
```

```
}

// 保存当前的迭代器
private var iterator: AsyncIterator?

// 如果已经存在迭代器了，则直接使用它
func makeAsyncIterator() → AsyncIterator {
    if iterator == nil {
        iterator = .init()
    }
    return iterator!
}
}
```

这样，即使中断了，在 `makeAsyncIterator` 被调用时，迭代器中的 `currentIndex` 会是之前停止时的最终值，对于序列的遍历将继续下去。

在之前的实现中，`AsyncSequence` 和 `AsyncIteratorProtocol` 是不同的类型：前者负责提供统一接口，创建迭代器；后者负责计算和存储状态，它是实际上负责产生序列值的类型。我们可以让 `AsyncSequence` 本身满足 `AsyncIteratorProtocol`，这样能将状态存储在自身，从而更简单地提供引用语义。只需要将 `currentIndex` 提取出来，用引用语义包装，就能得到一个单次遍历的更简单的实现了：

```
class Box<T> {
    var value: T
    init(_ value: T) { self.value = value }
}

struct BoxedAsyncFibonacciSequence
    : AsyncSequence, AsyncIteratorProtocol
{
    typealias Element = Int
    var currentIndex = Box(0)
```

```
mutating func next() async throws → Int? {
    defer { currentIndex.value += 1 }
    return try await loadFibNumber(at: currentIndex.value)
}

func makeAsyncIterator() → Self {
    self
}
}
```

在后面关于结构化并发和 TaskGroup 相关的部分，我们会看到相关的任务管理的 API 也是具有引用语义的异步序列。这种单次遍历的特点，将保证任务不会被（错误地）多次执行。另外，在那边我们还会看到如何处理序列的取消操作，以及要特别注意的相关事项。

## 操作异步序列

大部分同步序列上的扩展方法，比如 map, filter, contains 等，在异步序列中也是存在的。因此，基本上来说，只要你会使用同步序列，那么这些概念在异步序列中是完全共通的。

// 从斐波那契数列中取前五个偶数，乘以 2 并输出。

```
let seq = AsyncFibonacciSequence()
    .filter { $0.isMultiple(of: 2) }
    .prefix(5)
    .map { $0 * 2 }

for try await v in seq {
    print(v)
}

// 输出:
// 0 4 16 68 288
```

## Sequence 类型和延迟操作

这些扩展方法大多定义在 `AsyncSequence` 的 protocol extension 中并返回另一个 `AsyncSequence`。因此对于所有的 `AsyncSequence` 类型都适用，这也是为什么上面的各个操作调用可以链式进行的原因。

不过，对比同步的 `Sequence`，异步的 `AsyncSequence` 在返回类型和接受的参数上都有一些区别。拿 `Sequence` 的 `map` 举例，它的定义是：

```
extension Sequence {  
    func map<T>(  
        _ transform: (Self.Element) throws -> T  
    ) rethrows -> [T]  
}
```

虽然定义在了 `Sequence` 上，但是它返回的是数组形式的 `[T]` (或者写作 `Array<T>`，`[T]` 只是它的一种简写)。`Array` 确实满足 `Sequence` 协议，但它只是一种特殊的序列：一个数组的元素是有限的。`Sequence.map` 做的事情是穷尽整个序列，对其中的每一个元素，把它当作参数去调用 `transform`，然后把所有的结果作为数组返回。当被调用的 `Sequence` 会产生无限元素时 (就比如斐波那契数列)，`map` 求值也将没有尽头：`next()` 会被持续调用并产生一个无限循环：

```
// 错误代码  
let f = FibonacciSequence().map { $0 }
```

上面的例子运行时并不会无限循环，而是产生一个运行时的溢出崩溃，这是因为在计算若干次后，斐波那契数将超过 `Int` 上限。

想要让这样的无限序列也能使用 `map`，我们可以将原来的序列变形为 `LazySequence`。通过简单地在原序列上访问 `lazy`，就可以得到一个这样的“延迟求值”的序列：

```
let lazySeq = FibonacciSequence().lazy  
// lazySeq: LazySequence<FibonacciSequence>
```

```
let mapped = lazySeq.map { $0 }

// mapped:
// LazyMapSequence<
//     LazySequence<FibonacciSequence>.Elements,
//     LazySequence<FibonacciSequence>.Element
// >
```

考察 `lazySeq` 的类型，可以看到，它现在不再是普通的 `Sequence` 或 `Array`，而变成了一个新序列 `LazySequence`。这个新类型定义了自己的 `map` 函数，并返回延迟加载的 `LazyMapSequence`。通过在这些“内部序列”中增加延迟求值的逻辑，我们可以不再第一时间就对序列中的元素进行变形，而是等到代码实际访问到这些元素时，再对它们进行操作。

## 异步序列的高阶方法

我们花了一些时间回顾同步 `Sequence` 和懒加载的序列 `LazySequence` 中 `map` 的实现，现在可以来看看 `AsyncSequence` 的情况了。在概念上来说，`AsyncSequence` 与 `LazySequence` 更加相似：在使用像 `map` 这样的方法操作它们时，两者都不会立即对序列通过不断调用 `next` 来进行求值操作。区别在于，`LazySequence` 要求在访问元素时以同步方式提供元素，而 `AsyncSequence` 在变形时允许异步调用。

在操作序列时 `AsyncSequence` 和 `LazySequence` 的相似性，还可以从它们的类型中窥见一二。同样的 `map` 方法，它们的签名除了 `transform` 参数的 `async` 以外，十分相似：

```
extension LazySequenceProtocol {
    func map<U>(
        _ transform: @escaping (Self.Element) -> U
    ) -> LazyMapSequence<Self.Elements, U>
}

extension AsyncSequence {
    func map<Transformed>(
        _ transform: @escaping (Self.Element) async -> Transformed
    ) -> AsyncMapSequence<Self, Transformed>
}
```

AsyncMapSequence 存储了 transform，它的迭代器在通过 await next() 获取每个值时，会附带调用 await transform(value) 来取得并返回一个最终值。概念上来说，AsyncMapSequence 包含了如下实现：

```
extension AsyncMapSequence: AsyncSequence {  
  
    // ...  
  
    struct Iterator : AsyncIteratorProtocol {  
        var transform: (Self.Element) async → Transformed  
        var baseIter: Base.Iterator  
  
        mutating func next() async rethrows → Transformed? {  
            guard let baseValue = await baseIter.next() else {  
                return nil  
            }  
            return await transform(baseValue)  
        }  
    }  
}
```

## 产生和转换新序列

AsyncSequence 上的高阶函数操作大体可以分为两类。一类是上一小节中 map, filter 和 prefix 这样的，将原有序列转换为新的序列的操作；另一类则是 contains, first 和 reduce 这样，将原有序列收敛到一个值上的操作。

对于前者，我们已经看到，Swift 中使用新的序列类型将原来的序列和需要的操作包装起来，除了 AsyncMapSequence，各个操作也对应着自己的专属类型，比如 AsyncFilterSequence 或者 AsyncPrefixSequence。在使用时，我们很少会需要关注序列的具体类型，而只需要关心这个序列依然满足 AsyncSequence 的事实。特别是在经过多次转换后，序列的类型往往不可读，比如：

```
let seq = AsyncFibonacciSequence()
```

```
.filter { $0.isMultiple(of: 2) }
.prefix(5)
.map { $0 * 2 }
```

seq 的类型是：

```
AsyncMapSequence<
    AsyncPrefixSequence<
        AsyncFilterSequence<
            AsyncFibonacciSequence
        >
    >
, Int
>
```

和 SwiftUI 中对 View 的具体类型包装一样，如果我们需要为这样的序列写一个 getter，可以选择使用 some AsyncSequence 作为返回类型，把令人头疼的具体类型推断工作交给编译器：

```
var transformedFibonacciSequence: some AsyncSequence {
    AsyncFibonacciSequence()
        .filter { $0.isMultiple(of: 2) }
        .prefix(5)
        .map { $0 * 2 }
}
```

Swift 提供的高度可扩展性，能让我们自下向上地“改造”语言。如果我们希望像 map 那样扩展新的 AsyncSequence 类型，最灵活的方式就是仿照异步序列中已有的做法，为新的序列指定自己的类型，并基于原有的序列提供合适的实现。我们下面用一个例子来简单说明如何做到这一点。

在这里，我们想要创建一个能执行一定“副作用”的异步序列：它的行为可能和 map 有点相似：对于序列中的每个元素，我们以它为参数，去调用一个传入的函数。只不过和 map 不同，这个传入的函数不会对返回额外的映射值，也不会改变原序列，它只是一个副作用 (side effect)。

首先声明这个异步序列的类型。

```
struct AsyncSideEffectSequence
<Base: AsyncSequence>: AsyncSequence
{
    typealias Element = Base.Element

    private let base: Base
    private let block: (Element) → ()

    init(_ base: Base, block: @escaping (Element) → ()) {
        self.base = base
        self.block = block
    }

    // 1
    func makeAsyncIterator() → AsyncIterator {
        return AsyncIterator(
            base.makeAsyncIterator(),
            // 2
            block: block
        )
    }
}
```

1. 整体上说，它并不需要关心序列中元素的迭代方式，所以只需要对原有序列进行封装，并在 `makeAsyncIterator` 中使用原序列 `base` 的迭代器即可。
2. 从外部调用者接受需要执行的副作用，并将这个 `block` 传入到自己的迭代器中。

和其他异步序列一样，实际产生序列，并进行操作的方法被封装在迭代器中。在 `AsyncSideEffectSequence` 声明自己的迭代器 `AsyncIterator`，并在 `next` 方法中使用 `base` 迭代器获取值，然后执行副作用 `block(value)`。

```
struct AsyncSideEffectSequence
<Base: AsyncSequence>: AsyncSequence
```

```
{  
    // ...  
  
    struct AsyncIterator: AsyncIteratorProtocol {  
        private var base: Base.AsyncIterator  
        private let block: (Element) → ()  
  
        init(  
            _ base: Base.AsyncIterator,  
            block: @escaping (Element) → ()  
        ) {  
            self.base = base  
            self.block = block  
        }  
  
        mutating func next() async throws → Base.Element? {  
            let value = try await base.next()  
            if let value = value {  
                block(value)  
            }  
            return value  
        }  
    }  
}
```

在使用时，基于原有序列创建一个 `AsyncSideEffectSequence` 序列就可以了。为了方便，我们可以为整个 `AsyncSequence` 添加协议扩展方法：

```
extension AsyncSequence {  
    func print() → AsyncSideEffectSequence<Self> {  
        AsyncSideEffectSequence(self) {  
            Swift.print("Got new value: $(0)")  
        }  
    }  
}
```

```
    }
}

let seq = AsyncFibonacciSequence()
  .prefix(5)
  .print() // 打印数列前五个元素
  .filter { $0.isMultiple(of: 2) }

for try await v in seq {
  // 打印数列前五个元素中的偶数
  print("Value: \(v)")
}

// Got new value: 0
// Value: 0
// Got new value: 1
// Got new value: 1
// Got new value: 2
// Value: 4
// Got new value: 3
```

上面的 `AsyncSideEffectSequence` 看起来只是一个没什么用的 `map` 弱化版，不过我们可以进一步进行一些“改造”，比如除了 `next` 时调用的 `block` 外，还可以在 `makeAsyncIterator` 上添加一些钩子 (`hook`) 函数，这样我们可以在每次开始新一轮迭代时得到一些提示，有时候这样能让我们在调试时更好地理解异步序列的生命周期。另外，像是添加一个 `onCancel` 的挂载点，也能让我们为原有序列从外部添加自定义的取消操作：当我们需要实现序列取消时，也会很有用。

在上面 `block` 的类型是 `(Element) -> ()`，但异步序列肯定是在某个异步上下文中运行的，所以这个函数完全可以是异步函数：将 `block` 标记为 `(Element) async -> ()`，可以让使用者在副作用函数中进行异步调用，这毫无疑问会给我们带来更多的灵活性。

对于副作用函数，是否要允许 `block` 抛出（也就是标记为 `(Element) async throws -> ()`），是需要考虑的。在异步环境，特别是如果在一个结构

化并发任务的上下文中，抛出意味着任务失败，这可能会影响结构化中的其他任务。如果只是单纯的副作用，则不应该影响原序列的执行和结果。“抛出”如果作为副作用的话，这个副作用未免也过大了。

## 序列求值

对于另一种高阶函数的操作类型，它们将异步序列收敛到一个值上，这类方法通常需要对序列中的值进行遍历，需要在内部调用 `next()`，因此这种方法本身也是异步函数：

```
extension AsyncSequence {
    func contains(
        where predicate: (Self.Element) async throws → Bool
    ) async rethrows → Bool

    func first(
        where predicate: (Self.Element) async throws → Bool
    ) async rethrows → Self.Element?

    func reduce<Result>(
        _ initialResult: Result,
        _ nextPartialResult:
            (_ partialResult: Result, Self.Element) async throws → Result
    ) async rethrows → Result
}
```

想要为 `AsyncSequence` 添加一个这样的收敛到单一值方法相对简单：对原序列求值，并将参数函数应用在合适的值上即可。比如，我们想要重写一个属于自己的 `contains` 的话：

```
extension AsyncSequence {
    func myContains(
        where predicate: (Self.Element) async throws → Bool
    ) async rethrows → Bool
}
```

```
for try await v in self {
    if try await predicate(v) {
        return true
    }
}
return false
}
```

## AsyncStream

在上一章中，我们已经看到过如何使用续体 (continuations)，将一个已有的基于回调或代理的函数转换为异步函数了。比如一个 `load(completion:)` 函数：

```
func load(completion: @escaping ([String]?, Error?) → Void)
```

要提供它的异步函数版本，可以使用 `withUnsafeThrowingContinuation` 包装：

```
func load() async throws → [String] {
    try await withUnsafeThrowingContinuation { continuation in
        load { values, error in
            // ... 检查 values 和 error，成功路径：
            continuation.resume(returning: values)
        }
    }
}
```

通过 `continuation` 进行转换的这种方式，要求 `continuation` 的 `resume`，不论成功还是失败，只能进行一次调用。或者说，它只接受一个未来值：成功时的返回值或者失败时的抛出值。

但是“未来的单次值”只能覆盖一部分使用情景，如果某个回调或者某个代理方法有可能被多次调用的话，我们将会得到不止一个，而是一系列未来的值：没错，这就是一个异步序列。

对于将这种将多次调用的异步操作转换为一个异步序列的需求，Swift 提供了 `AsyncStream` 类型。和 `with*Continuation API` 类似，`AsyncStream` 也允许我们通过在提供的续体上调用函数，来控制异步函数的执行：

```
// 为了简单，省略了部分类型名称
struct AsyncStream<Element> {
    init(
        _ elementType: Element.Type = Element.self,
        bufferingPolicy limit: BufferingPolicy = .unbounded,
        _ build: (AsyncStream<Element>.Continuation) → Void
    )

    struct Continuation {
        func yield(_ value: Element) → YieldResult
        func finish()
        // ...
    }

    // ...
}
```

`AsyncStream` 的 `init` 方法接受 `build` 闭包作为输入，它拥有一个表示当前续体的参数 `Continuation`。当新的异步值产生时，我们通过调用 `yield` 来在异步序列中添加一个值；当所有事件完成，这个序列不再会产生新的值时，我们需要调用 `finish` 来表示完结。

除了 `AsyncStream` 外，和 Swift 并发话题下的其他一些 API 类似，也存在可抛出错误的版本 `AsyncThrowingStream`：在可抛出版本中，`Continuation` 可以接受 `finish(throwing:)` 表示出错，其他部分和 `AsyncStream` 并没有太多区别。

为了具体来看看 `AsyncStream` 的行为，以及如何将同步世界的代码转换到 `AsyncStream`，我们考虑下面的例子。

假设我们在同步世界里有一个 `Timer` 类型的计时器，每秒调用一个 `tick` 方法，并在十秒后结束：

```
let initial = Date()
Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true) {
    timer in
    let now = Date()
    print("Value: \(now)")
    let diff = now.timeIntervalSince(initial)
    if diff > 10 {
        timer.invalidate()
    }
}
```

通过使用 `AsyncStream` 的初始化方法，可以创建一个同等的异步序列：

```
var timerStream: AsyncStream<Date> {
    AsyncStream<Date> { continuation in
        let initial = Date()

        // 1
        Task {
            let t = Timer.scheduledTimer(
                withTimeInterval: 1.0,
                repeats: true
            ) { timer in
                let now = Date()
                print("Call yield")
                continuation.yield(Date())

                let diff = now.timeIntervalSince(initial)
                if diff > 10 {
                    print("Call finish")
                    continuation.finish()
                }
            }
        }
    }
}
```

```
// 2
continuation.onTermination = { @Sendable state in
    print("onTermination: \(state)")
    t.invalidate()
}
}
}
}
```

1. Timer.scheduledTimer 方法将在当前 runloop 中以默认模式添加计时器。如果不为它准备一个新的任务环境，在 await 时计时器会在 runloop 中一直等待，无法正确工作。
2. 我们可以通过设定续体的 onTermination 属性，在异步序列结束时进行一些清理工作。这个闭包要求满足 @Sendable。我们会在后面涉及数据安全的部分再对 @Sendable 标记以及相关的协议展开讨论。

由于 AsyncStream 是遵守 AsyncSequence 协议的，我们可以类似地通过 for await 的迭代语法进行使用：

```
let t = Task {
    let timer = timerStream
    for await v in timer {
        print(v)
    }
}

// 输出:
// Call yield
// 2021-07-19 06:36:12 +0000
// Call yield
// 2021-07-19 06:36:13 +0000
// ...
// Call yield
```

```
// Call finish
// 2021-07-19 06:36:21 +0000
// onTermination: finished
```

在 10 秒后，timerStream 进入到 diff > 10 的分支，continuation.finish() 被调用，进而触发 onTermination 闭包，且得到的结果为 finished。除了序列完结之外，在运行序列的任务被取消时，AsyncStream 续体的 onTermination 也会被调用，在这种情况下，参数 Termination 将会是 enum 中的另一个成员 cancelled：

```
enum Termination {
    case finished
    case cancelled
}

let t = Task {
    let timer = timerStream
    for await v in timer {
        print(v)
    }
}
await Task.sleep(2 * NSEC_PER_SEC)
t.cancel()

// 输出:
// Call yield
// 2021-07-19 06:43:11 +0000
// Call yield
// 2021-07-19 06:43:12 +0000
// onTermination: cancelled
```

在之后结构化并发的章节中，我们会更详细地介绍各种情况下的取消操作，以及如何正确地对应和实现任务的取消。

## 缓冲策略

在 AsyncStream 的初始化方法中，除了 build 之外，还有几个可选参数：

```
struct AsyncStream<Element> {
    init(
        _ elementType: Element.Type = Element.self,
        bufferingPolicy limit: BufferingPolicy = .unbounded,
        _ build: (AsyncStream<Element>.Continuation) → Void
    )
    // ...
}
```

第一个参数 elementType 是为了编译器和内部实现需要所附加的条件，在创建 timerStream 时，我们明确指定了元素的泛型类型为 AsyncStream<Date>。如果不直接指定泛型类型，我们也可以通过这个参数来创建 AsyncStream：

```
AsyncStream(Date.self) { /* */ }
```

除了帮助推断类型之外，它并没有更多的实际用处。

bufferingPolicy 则实实在在地影响 AsyncStream 的行为。在 timerStream 中，Timer 在 AsyncStream 创建时就已经开始运行并计时了。接下来 yield 方法将被每秒调用一次。如果 for await 没有能及时“消化”这些值的话，它们将被暂时存储到 AsyncStream 的内部缓冲区中。考虑下面的代码，在首次 await 前，我们等待了五秒：

```
let timer = timerStream
await Task.sleep(5 * NSEC_PER_SEC)
for await v in timer {
    print(v)
}
print("Done")
```

此时输出将变为：

```
Call yield
Call yield
Call yield
Call yield
Call yield
2021-07-21 06:51:40 +0000
2021-07-21 06:51:41 +0000
2021-07-21 06:51:42 +0000
2021-07-21 06:51:43 +0000
2021-07-21 06:51:44 +0000
Call yield
2021-07-21 06:51:45 +0000
Call yield
2021-07-21 06:51:46 +0000
```

...

Done

在首次 for await 之前，yield 已经被调用了五次，它们的值被全部存储在 AsyncStream 中，并等待 for await 时给到外部调用者，直到缓冲值被全部消耗光后，回到每秒一次的调用。

AsyncStream 在内部实现了一个由互斥锁保护的高效队列，用来作为 yield 调用的缓冲区。在每次 for await 时，AsyncStream 的迭代器(不要忘记 AsyncStream 是满足异步序列协议的)的 next 方法会向这个内部存储队列请求一个值，并将它返回。只有在缓冲区中没有值时，这个 await 才进入真正的等待状态。

在创建 AsyncStream 时没有指明的情况下，bufferingPolicy 参数接受的是默认值 .unbounded。它会尝试无限量地缓冲接收到的值，直到 for await 迭代开始。在大多数用例下，这个行为符合直觉，并简化了 AsyncStream 的使用。但是这种无界行为天然地存在风险：当可能需要缓冲的数据量没有上限时，这种策略是否可用，取决于内存极限和迭代哪一个会先来到。而且在很多情况下，也许迭代速度会比缓冲速度更慢，这也可能让缓冲区中产生大量的数据堆积，进而造成内存崩溃等问题。

在 timerStream 中，我们通过 for await 循环来消耗缓冲区中的数据，而向缓冲区填充数据，则依靠每秒一次的 Timer 计时。因此，消耗数据的速度要远远大于产生数据的速度。但在实际中，有些情景下状况可能刚好相反：比如在复制大文件时，我们一边读取数据到缓冲区中，一边将缓冲区中的数据写入到硬盘新的地址。一般来说磁盘的物理性能，在都满速的情况下，读取速度会远高于写入速度，因此如果我们依然采用默认行为的话，可能会导致缓冲区溢出，最终由于内存不足而发生崩溃。

只要异步的发送端的速度快于接收端，就可能会出现这样的问题。在 UI 开发中，也能举出其他很多例子：例如 app 从服务器不断收到数据，并需要将这些数据渲染到屏幕上。作为纯数据行为，前者的速度会远远大于涉及到 UI 渲染的后者，如果不采用一些手段和措施，最终界面卡顿或者程序崩溃，都将是预期中的结果。

除了 .unbounded 外，bufferingPolicy 参数还提供了两种可能的选择：

```
enum BufferingPolicy {  
    case unbounded  
  
    case bufferingOldest(Int)  
    case bufferingNewest(Int)  
}
```

顾名思意，bufferingOldest 将在达到上限后抛弃掉新收到的数据，而 bufferingNewest 则相反，它会抛弃最旧的数据。这两种缓冲策略思路是一致的：通过限制缓冲区的大小，来丢弃一些数据，从而达到安全。当我们在调用 yield 向缓冲区写入数据时，这个方法会把写入的结果返回给我们：

```
func yield(_ value: __owned Element) → YieldResult  
  
enum YieldResult {  
    // 写入队列，并返回告知缓冲区的剩余大小  
    case enqueued(remaining: Int)  
    // 缓冲区满，丢弃  
    case dropped(Element)
```

```
// 序列已经完结
case terminated
}
```

通过检测续体 yield 的结果，`AsyncStream` 的实现者可以对写入行为进行确认或额外的控制。举个例子，在 `timerStream` 中我们如果采用 `bufferingNewest(3)` 作为缓冲策略的话，下面的代码：

```
let timer = timerStream(bufferingPolicy: .bufferingNewest(3))
await Task.sleep(5 * NSEC_PER_SEC)
for await v in timer {
    print(v)
}
print("Done")
```

输出为：

```
Call yield: enqueueued(remaining: 2)
Call yield: enqueueued(remaining: 1)
Call yield: enqueueued(remaining: 0)
Call yield: dropped(2021-07-21 07:48:29 +0000)
Call yield: dropped(2021-07-21 07:48:30 +0000)
2021-07-21 07:48:31 +0000
2021-07-21 07:48:32 +0000
2021-07-21 07:48:33 +0000
Call yield: enqueueued(remaining: 3)
2021-07-21 07:48:34 +0000
Call yield: enqueueued(remaining: 3)
```

...

Done

只要缓冲区中的值超过 3 个时，序列就将丢弃最旧的值，直到序列缓冲区开始消耗。

在极端情况下，bufferingOldest 或者 bufferingNewest 的绑定值可能被设为 0。这种情况意味着 AsyncStream 中不存在可用的缓冲区，continuation 的 yield 方法所产生的值将被直接抛弃掉。这个特性可以让我们拥有在运行时通过设置缓冲区策略来暂时“关闭”异步序列的能力。

## 背压

在处理一些 UI 的问题时，比如对超大量的数据进行弹幕渲染，于用户来说可能是没有意义的。因此使用 bufferingOldest 或 bufferingNewest 丢弃一部分数据，有时候可以帮助我们更合理地完成任务。但是对于像是文件复制这样的工作，显然不能采用“丢弃”的方式来粗暴处理。为了避免数据堆积导致的缓冲区爆炸，我们需要一种方式来协调向缓冲区写入的速度和从缓冲区读取的速度。用文件复制的例子来说，就是限制其读取速度，让它和写入速度相匹配。换言之，在 AsyncStream 中，这意味着只有在 for await 中请求下一个元素时，序列才生成并提供这个元素。

在事件处理和响应式编程中，我们借用一个流体力学的术语，把这种由数据消耗端按照自己的速率来控制数据发生端的行为，叫做狭义的背压 (backpressure)。AsyncStream 除了接受 build 闭包的初始化方法外，还提供了一个直接返回数据元素的初始化方法，它可以让我们使用背压的方式控制异步序列的产生：

```
struct AsyncStream<Element> {
    init(
        unfolding produce: @escaping () async → Element?,
        onCancel: (@Sendable () → Void)? = nil
    )
}
```

参数的 unfolding 将被用作序列迭代器的 next 函数，每当 for await 请求值时，它会被调用并生成一个新的值。用这个初始化方法，我们可以创建一个类似于上面的 timer stream。只不过它的行为略有不同：它不再是由 Timer 驱动并自动填充缓冲区，而是在每次数据消耗者进行 for await 时，等待一秒并返回序列中的下一个值：

```
AsyncStream<Date> {
    await Task.sleep(NSEC_PER_SEC)
    return Date()
```

```
    } onCancel: { @Sendable in
        print("Cancelled.")
    }

    for await v in timer {
        print(v)
    }
}
```

实际看起来效果和之前 Timer 驱动的序列并无二致，但实际上现在序列的发送已经由使用方来决定了，序列也不再涉及缓冲区的问题。

## 序列完结

让我们从背压的讨论抽身，回到由续体和 yield 方法驱动的 AsyncStream 来，看看在序列完结时的一些特性。

在本节一开始，我们提到过 AsyncStream 和 with\*Continuation 有一些相似：它们同样捕获并提供一个续体用来操作。with\*Continuation 中要求 continuation 的 resume 调用且仅调用一次，来表征异步函数从续体中以成功或者失败的结果继续，多次调用 resume 将导致意外行为。在前面的例子中，我们已经看到了 AsyncStream 的用法。其中 continuation 的 yield 可以被多次调用以产生若干序列值。通过调用 continuation 的 finish 方法，则可以终结一个序列。不过，在 AsyncStream 终结后，你依然可以继续使用 yield 发送数据。下面的代码是完全合法的：

```
continuation.yield(Date())
continuation.yield(Date())
continuation.finish()

// 序列结束后再次 yield
let result = continuation.yield(Date())
// result: YieldResult.terminated
```

调用 finish 方法或者取消运行序列的任务，都会让序列续体进入到完结状态。之后的 yield 并不会将数据写入缓冲区，而是直接返回 .terminated 来告诉 AsyncStream 已经完结了。在某些情况下，除了 onTermination 外，也可以通过这个 yield 的 .terminated 返回来进行资源的清

理工作(比如例中让 Timer 无效化)。但是这样做会导致清理工作依赖于终结后的下一次事件，让待清理的资源的生命周期变长，因此并不推荐这么做。

AsyncStream 本身是 struct，但为了保证单次遍历，它的内部使用了引用语义的 class 作为存储，来对序列的当前状态进行维护。在调用 continuation 上的方法时，实际上是对这个状态进行检查和设置。这涉及到用锁进行数据同步，也是这些方法可以随意调用的代价。事实上，在创建一个 AsyncStream 时，我们应该尽量避免在序列完结后再次发送数据的行为，这样不论对使用者和维护者来说，都可以减轻心智模型上的负担。

## 多任务迭代

虽然多次调用 yield，甚至是在序列完结后再调用 yield，都没有问题，但是在使用 AsyncStream 时要注意，我们不能在多个任务上下文中对同一个序列进行迭代。如果这种情况发生，将会带来运行时崩溃：

```
let timer = timerStream
Task.detached {
    for await v in timer {
        print(v)
    }
}
Task.detached {
    for await v in timer {
        print(v)
    }
}

// 运行时错误
// Fatal error: attempt to await next() on more than one task
```

在上面我们提到过，AsyncStream 的内部实现使用了互斥锁来防止多个线程同时访问缓冲区和内部状态。这保护了续体的独占性：同时只有一个任务可以获取 AsyncStream 暂停时提供的续体，并向其询问和获取下一个元素。如果其他任务同时访问并希望序列向前迭代，会因为无法获得已经被占用的续体，而产生错误。这保证了序列的单向特性和安全。

在实际开发时，保证不在任务之间共享序列，是使用异步序列的一个原则。

## 异步序列和响应式编程

异步序列代表了将来可能出现的一系列值，在这一点上，异步序列和 Combine 框架中的 Publisher 十分相似。也正因如此，社区里经常有一些声音或者尝试，想用 AsyncSequence 和 AsyncStream 替代 Combine。

在原理上两套体系确实相似，也能够进行部分有效替代，但是在本节中，我们还是会看到它们之间存在着不同。这种区别不仅体现在它们提供的 API 的不同所导致的具体处理方式的不同，也体现在从根源开始的设计理念的差别。

### Combine 的转换

AsyncStream 允许我们将一系列事件（包括正常事件值、结束、以及错误）通过续体的形式转换为一个异步序列。如果你对 Combine 框架比较熟悉的话，可能还记得其中的 Publisher 也正代表了这样一个事件流。因为它们所代表的数据模型一致，所以将任意 Publisher 转换为异步序列是轻而易举的，只需要用 AsyncStream 进行简单包装即可：

```
extension Publisher {  
    var asAsyncStream: AsyncThrowingStream<Output, Error> {  
        AsyncThrowingStream(Output.self) { continuation in  
            let cancellable = sink { completion in  
                switch completion {  
                    case .finished:  
                        continuation.finish()  
                    case .failure(let error):  
                        continuation.finish(throwing: error)  
                }  
            } receiveValue: { output in  
                continuation.yield(output)  
            }  
        }  
    }  
}
```

```
continuation.onTermination = {
    @Sendable _ in
    cancellable.cancel()
}
}
}
}
```

有了这个扩展方法，我们就可以把任意的 Publisher 用异步序列表示了：

```
let stream = Timer.publish(every: 1, on: .main, in: .default)
    .autoconnect()
    .asAsyncStream
for try await v in stream {
    print(v)
}
```

反过来，把一个异步序列（不只是 AsyncStream，也包括更一般的 AsyncSequence）转换为 Publisher 也并不困难。

虽然可以互相转换，但是这并不意味着 Combine 和异步序列可以完全等价互换。我们接下来会探讨它们的一些不同之处。

## 异步序列的错误处理

最显著的区别来自于它们对于错误的处理方式。Combine 的 Publisher 通过 associatedtype 的方式，明确地规定了 可能值Output 和 错误值Failure 的类型。进一步，对使用 Operator 进行组合，或者使用 Subscriber 进行订阅时，除了要求可能值的类型一致外，也要求错误值的类型相互匹配。另外，对于错误类型的转换，Combine 中也提供了诸如 mapError 和 setFailureType 这类专门处理错误类型的操作。可以说，在 Combine 的世界中，所有的错误都是被严格对待的，它们的类型至关重要，且被编译器强制保证。

而另一方，Swift 函数，包括异步序列，在遇到错误时都是使用 throw 来进行的。想要一个异步序列支持错误处理，我们会使用支持错误抛出的 AsyncThrowingStream：

```
struct AsyncThrowingStream<Element, Failure>
    where Failure : Error
{
    // ...
}
```

虽然泛型类型中定义了 Failure，但是它只在内部通过 `finish(throwing:)` 时被使用。对于序列的使用者来说，在使用 `for try await` 捕获错误时，并不能体现出 Failure 类型的作用，Swift 的 `catch` 捕获的都是一般性的 Error：

```
let s: AsyncThrowingStream<Int, MyError>

do {
    for try await v in s {
        // ...
    }
} catch {
    // 捕获的是一般性的 Error
    // error: Error
    if let myError = error as? MyError {
        // ...
    }
}
```

在 `catch` 中，将捕获的 `error: Error` 转换为实际的 `MyError`，需要一个 `if let` 绑定。而且这个转换并没有很强的编译器保证：即使 `s` 类型的错误类型在之后变成了其他类型，使用侧的代码依然能够无警告地编译通过。这时 `catch` 中的这个转换将会失败，这让我们非常容易在重构或者外部库升级时错过应有的处理。

究其原因，这在于 `throw` 永远不会抛出一个具体的 `Error` 类型。关于 `throw` 是否应该抛出强类型错误这个问题，从 Swift 支持 `throw` 后就一直是社区争论的焦点。社区开发者们对 Combine 的错误处理方式表现出一致的满意，但如果在 `throw` 中加入强类型，也可能导致抛出错误的类型层层叠加，最终变得十分复杂。除了仔细检查，并用测试用例对错误转换方面进行覆盖以外，当前我们似乎并没有什么更好的选择来处理这个问题。

## 调度和执行

在涉及到执行方式和时间维度时，Combine 使用 Scheduler 协议进行抽象。通过指定调度器 (scheduler)，Combine 实现了一系列有关时间的操作 (比如 delay、debounce 和 throttle 等)，并可以在下游指定谁应该接收事件 (使用 receive(on:options:))。通过调度器，Combine 可以很灵活地组织和自定义异步事件的行为，为整个框架的使用提供了相当的便利。

而相对来说，异步序列的调度和执行就要僵硬一些。包括异步序列在内的异步函数必须在某个任务中运行，而在什么时间什么线程上运行这些任务，则是由内部的执行器 (executor) 来决定的。Swift 的并发模型提供了几种默认的内建执行器，它们的主要目标是保证续体切换的性能或者保证数据安全。现在 Swift 并发还不支持自定义执行器，所以我们没有太多方法来干涉异步序列的执行方式。对于很多 Combine 中内建存在的 Publisher 或者轻而易举就能实现的事件流，在异步序列中实现起来可能要困难一些。

正如其名，Combine 更擅长于将不同的事件流进行变形和合并，生成新的事件流：它的重点在于为响应式编程范式提供工具。而 Swift 异步序列的侧重点有所不同，更多时候，它服务于任务 API 及 actor，用来解决并发编程中的痛点。因此单纯地想用 Combine 代替异步序列，或者反过来用异步序列代替 Combine，笔者认为并不是合理的做法。

如果你对 Combine 编程感兴趣，推荐您参考阅读我的另一本书籍 《SwiftUI 与 Combine 编程》。对于刚才提到的执行器，大概率会作为下一个 Swift 版本中对并发模型的增强和补全出现。我们会在本书稍后介绍完并发模型后，再对它进行一些浅显的探讨。

## 小结

异步函数“返回”一个未来的值，而异步序列则代表未来的一系列值。本章中我们仔细查看了 AsyncSequence 协议和 AsyncStream 类型。通过自行创建一些异步序列类型，研究了它们的行为。

作为 app 开发者来说，在创建 app 时很多时候并不需要自行去创建异步序列，我们接下来就会看到很多使用到异步序列的系统级 API。不过，如果你正在开发一套异步函数兼容的库，并打算提供给别的开发者使用，那么将“随着时间推移而不断产生新值”这样的数据模型，封装到一个异步序列中，将给其他开发者带来极大便利。他们将能够使用 for await 对这些异步值进行迭代。

代，并自由地使用像是 `map`, `filter` 或者 `contains` 这些定义在异步序列扩展中的方法。这些特性可以将异步 API 进行大幅简化，它们不仅让使用者以更易于理解的“类似同步”的方式书写代码，也进一步为结构化并发提供工具，使并发代码的管理变得可行。

# 使用异步函数

5

为了能让开发者在 `async/await` 特性发布第一天开始就能从中获益，Apple 在一些系统框架中已经对部分基于回调的函数进行了异步适配。一些最典型的用例包括在 `URLSession` 和 `Notification` 中。在本节我们对它们进行简单介绍。

另外，异步函数只能在异步任务的上下文中运行，而 `UIKit` 当前提供的入口（比如 `viewDidLoad` 等），基本都还是同步的。我们在遇到一个异步函数时，应该如何执行它，是很多读者实际使用异步函数时所遇到的第一个大问题。本章中我们也会对相关话题进行讨论。

## 网络请求中的异步函数

### 传统的 `URLSession`

毫无疑问，网络请求是我们在日常开发中最常见的耗时操作了。在 iOS 15/macOS 12 的 SDK 中，`URLSession` 额外提供了异步函数的方法，让我们能简单地进行网络请求。

传统的 `URLSession` 使用分为两种主要方式：使用回调函数处理响应，或者使用代理对网络行为进行更多的控制。如果你只关心请求最终的结果，而不关心其中的过程，并且对一些常见的处理，比如服务器的重定向行为（redirection），或者接收到验证挑战（authentication challenge），都采用默认行为的话，基于回调函数的请求方式可以提供最便捷的请求调用：

```
let task = URLSession.shared.dataTask(with: url) {
    data, response, error in
    if let error = error {
        print("Error: \(error)")
        return
    }

    if let data = data {
        print("Data: \(data.count) bytes.")
    }
}

task.resume()
```

```
// 示例输出:  
// Data: 1256 bytes.
```

如果我们需要更细粒度的控制，则需要从创建自定义的 Session 开始，并指定接受代理方法的实例 Delegate：

```
let session = URLSession(  
    configuration: .default,  
    delegate: Delegate(),  
    delegateQueue: nil  
)  
let task = session.dataTask(with: url)  
task.resume()
```

这样，当网络请求过程中发生相应的事件时，Delegate 上的相关方法将被调用，并让我们对加载过程进行控制。

```
class Delegate: NSObject, URLSessionDataDelegate {  
    func urlSession(  
        _ session: URLSession,  
        dataTask: URLSessionDataTask,  
        didReceive response: URLResponse,  
        completionHandler:  
            @escaping (URLSession.ResponseDisposition) → Void  
    ) {  
        print("Receive response")  
        completionHandler(.allow)  
    }  
  
    func urlSession(  
        _ session: URLSession,  
        dataTask: URLSessionDataTask,  
        didReceive data: Data
```

```
) {  
    print("Data chunk: \(data.count)")  
}  
}  
  
// 示例输出:  
// Receive response  
// Data chunk: 1256 bytes.  
  
// 如果数据足够多的话，可以看到多次 Data chunk 输出。
```

这里我们只列举了接受 URL 参数的版本,这会产生一个访问该 URL 的 GET 请求。如果需要进一步定义请求细节,比如发送 POST 请求并附带数据,则需要创建 URLRequest 并使用 URLSession 中对应版本的方法。不过两者只在创建任务时有所不同,其他方面是共通的。另外,除了创建一个请求数据的 data task 外,还有一些生成其他更专门的任务的方法,比如 upload 或者 download task。它们只在细节上略有区别,我们就略过不提了。

## 异步 URLSession 方法

不论是闭包回调,还是基于 delegate,两种方法的共同点在于,它们都需要通过 dataTask 方法创建一个任务,并调用这个对象的 resume() 来实际开始网络请求。这种带有返回值的异步操作,是无法直接映射成异步函数的。Apple 团队为 URLSession 新添加了异步函数请求的方法:

```
extension URLSession {  
    func data(  
        from url: URL,  
        delegate: URLSessionTaskDelegate? = nil  
    ) async throws → (Data, URLResponse)  
}  
  
// 使用
```

```
let (data, response) = try await URLSession.shared.data(from: url)
if let httpResponse = response as? HTTPURLResponse {
    print("Status Code: \(httpResponse.statusCode)")
}
print("Data: \(data.count) bytes.")
```

相比起传统方法，这个新函数优点十分明显。除了具有一般的异步函数的优点外，它还额外提供了一些特性：

1. 网络请求任务将立即开始，不再依赖于调用 `resume`。原先使用 `dataTask` 方法生成的 `URLSessionDataTask` 实例，在创建时将占用一系列 `session` 资源。如果由于某种原因，没有调用 `resume` 的话，直到 `session` 整个结束，这些资源都不会被清理，很容易造成事实上的内存泄漏，而且这很难被察觉到。
2. 和之前针对整个 `session` 的 `delegate` 不同，这里的 `delegate` 是针对单个任务的。这让我们在收到的代理方法调用时，不再需要缓存和区分这个调用到底来自哪个任务，这让控制任务可以在更细粒度上更清晰地实现。

## 基于 `bytes` 的异步序列

在某些情况下，可能我们只对响应中的部分数据有兴趣。比如下载图片时通过 `body` 的前几个字节判断图片类型和尺寸，或者对一个特别大的字符串 `body` 按行读取并寻找关键内容。在以前，除了等待请求完成，完整的 `Data` 被收集以外，我们只能通过检查并收集 `delegate` 的 `urlSession(_:dataTask:didReceive:)` 中给出的 `data` 参数来完成这项任务。不过，现在我们有更好的方式来按字节读取响应中的数据了：

```
extension URLSession {
    func bytes(
        from url: URL,
        delegate: URLSessionTaskDelegate? = nil
    ) async throws → (URLSession.AsyncBytes, URLResponse)
}
```

和 data(from:) 方法等待响应完成并获取所有数据不同，新加入的 bytes(from:) 返回的不是完整的 Data，而是一个代表响应中 body 字节数据的 AsyncBytes 类型。AsyncBytes 是一个异步的数据序列，它的值代表了数据的每个字节。

```
struct AsyncBytes : AsyncSequence {  
    typealias Element = UInt8  
    // ...  
}
```

通过对这个序列进行迭代，我们可以按照字节来接收响应数据：

```
let url = URL(string: "https://example.com")!  
let session = URLSession.shared  
let (bytes, response) = try await session.bytes(from: url)  
for try await byte in bytes {  
    print(byte, terminator: ",")  
}  
  
// 输出，每个接收到的字节  
// 60,33,100,111,99 ...
```

如果我们需要通过前几个字节来判断响应 body 的一些属性的话，这个函数将非常有用：

```
var pngHeader: [UInt8] = [137, 80, 78, 71, 13, 10, 26, 10]  
for try await byte in bytes.prefix(8) {  
    if byte != pngHeader.removeFirst() {  
        print("Not PNG")  
        return  
    }  
}  
print("PNG")
```

我们不再需要等待整个响应完成，而只用最多获取响应中的八个字节，就能检查 body 是不是满足 PNG 图片文件的规范了。

更进一步，基于 AsyncBytes，或者更精确来说，针对 Element 为 UInt8 的异步序列，Apple 还提供了一系列扩展方法，让我们把字节转换为更容易看懂的形式，比如按照每行转为 UTF8 的 String、Character 或者是 UnicodeScalars：

```
extension AsyncSequence where Self.Element = UInt8 {  
    var lines: AsyncLineSequence<Self> { get }  
    var characters: AsyncCharacterSequence<Self> { get }  
    var unicodeScalars: AsyncUnicodeScalarSequence<Self> { get }  
}
```

在异步序列一章中，我们已经看到过如何将异步序列进行转换了，这里的做法并无不同。简单地使用这些包装好的属性，将让代码在可读性上大幅提升：

```
let url = URL(string: "https://example.com")!  
let session = URLSession.shared  
let (bytes, _) = try await session.bytes(from: url)  
  
for try await line in bytes.lines {  
    print(line)  
}  
  
// 按行输出 https://example.com 的响应:  
// <!doctype html>  
// <html>  
// ...
```

除了 URLSession，URL 现在也接受类似的方法：url.resourceBytes 返回一个异步的字节序列，url.lines 按行返回字符串序列。如果你只是想要简单地向某个 URL 发送一个 GET 请求，这应该是最容易的获取结果的方法了：

```
let url = URL(string: "https://example.com")!  
for try await line in url.lines {  
    print(line)  
}
```

当然，上面的 URL 对于本地文件也有效。实际上，除了基于 URLSession 的网络请求外，和文件读取操作相关的 FileHandle API 中也提供了 bytes 方法，来把加载的数据表征为异步序列。同为 I/O 操作，这些新加入的抽象把具体的加载过程省略，而从本质上强调了“异步加载数据”这一核心操作。这样我们就可以使用相似的方式来处理不同数据源的输入了。

## 协议代理方法

除了直接的方法调用中添加了异步函数外，对于部分协议中的代理方法，现在也可以用异步函数的方式来实现了。在 Apple 提供的框架中，有很多这样的例子：代理方法中包含一个闭包参数，在这个代理方法被调用时，Apple 希望我们在处理之后，调用这个函数参数。这将让我们可以以异步的方式来控制框架之后的行为。

URLSessionDataDelegate 中就有一个这样的例子：

```
extension ViewController: URLSessionDataDelegate {
    func urlSession(
        _ session: URLSession,
        dataTask: URLSessionDataTask,
        didReceive response: URLResponse,
        completionHandler: @escaping
            (URLSession.ResponseDisposition) → Void
    ) {
        guard let scheme = response.url?.scheme,
              scheme.starts(with: "https") else
        {
            completionHandler(.cancel)
            return
        }

        completionHandler(.allow)
    }
}
```

这个方法会在连接建立且接收到最初的响应（但可能还没有接收到 body 数据）时被调用。在这个方法期间，响应的接收会被暂时挂起，只有在我们调用 `completionHandler(.allow)` 后，才会继续接收响应数据。如果响应不满足某种预期，我们可以通过传入 `.cancel` 来取消这个请求。

与一般的带有 `completion` 回调的方法一样，在需要回调的代理方法中，编译器也没有能力保证我们在每个条件分支中都调用了 `completionHandler`。如果忘记了调用，造成的结果往往更加糟糕：在这个例子中，请求将被一直挂起，相关的资源也无法得到释放。

Swift 引入异步函数后，只要满足我们之前提到的自动转换原则，这类基于回调的代理方法也将被转换为 `async` 函数。比如在 `URLSessionDelegate` 中，除了上面的回调版本，还有一个异步函数的版本。通过使用这个异步版本，我们可以把上面的方法简单改写成返回 `.cancel` 或 `.allow` 的形式：

```
func urlSession(  
    _ session: URLSession,  
    dataTask: URLSessionDataTask,  
    didReceive response: URLResponse  
) async -> URLSession.ResponseDisposition  
{  
    guard let scheme = response.url?.scheme,  
        scheme.startsWith("https") else  
    {  
        return .cancel  
    }  
  
    return .allow  
}
```

和 `completionHandler` 的版本相比，各条件语句中的强制返回保证了程序流的继续。

`URLSessionDelegate` 是 Objective-C 中定义的代理方法，它在 Swift 中的异步函数版本是通过转换规则得到的。如果在 Swift 中我们同时实现了某个协议方法的回调版本和异步版本，在转换回 Objective-C 的世界时，它们的方法签名将产生冲突。这种情况下，编译器会给出错误，强制我们移除掉一个实现。

不过，同样的事情并不适用于纯 Swift 的协议方法。比如我们定义了下面的协议：

```
protocol P {  
    func doSomething(  
        completionHandler: @escaping (Bool) → Void  
    )  
    func doSomething() async → Bool  
}
```

想要实现协议 P，必须同时实现回调版本的 doSomething(completionHandler:) 和异步版本的 doSomething: 编译器并不会认为它们是等效的方法。

```
class S: P {  
    func doSomething(  
        completionHandler: @escaping (Bool) → Void  
    ) {  
        completionHandler(true)  
    }  
  
    func doSomething() async → Bool {  
        return true  
    }  
}
```

对于接口的维护者来说，一般我们会想要在保持后向兼容的同时，逐渐提供让接口使用者进行迁移的机会。我们可以通过实现协议扩展，来为异步版本的方法提供默认实现，并同时在文档中提醒用户应当选择使用异步版本的实现。使用之前提到过的 withUnsafeContinuation 应该能够胜任这项任务：

```
extension P {  
    func doSomething() async → Bool {  
        await withUnsafeContinuation { continuation in  
            doSomething { v in  
                continuation.resume(returning: v)  
            }  
        }  
    }  
}
```

```
    }
}
}
}
```

extension P 中的 doSomething() async 并不会影响用户自己进行的实现，而它也为接口项目内部的 Swift 并发迁移提供了条件。

当然，如果这个协议原本就是计划提供给 Objective-C 世界使用的话，可以简单地将它标记为 @objc，这样我们就可以利用编译器的自动转换了。不过，要注意在很多时候，如果协议原来并没有限定 @objc 的话，这会带来更大的破坏性（比如 Swift struct 将无法再实现这个协议）：

```
@objc protocol P {
    func doSomething(
        completionHandler: @escaping (Bool) → Void
    )
    func doSomething() async → Bool
}
```

另外一个值得注意的细节是，非异步的方法可以实现协议中异步的方法。也就是说，下面的做法是可以通过编译的：

```
protocol P {
    func doSomething() async → Bool
}

struct S: P {
    func doSomething() → Bool {
        return true
    }
}
```

这个“特性”和 throws 在 protocol 中的表现是类似的：S 中同步函数所能表达的能力，是 P 中异步函数能力的子集，我们总可以在一个异步函数中执行同步操作。然而，反过来却不成立：

如果 P 要求一个同步函数，我们不能在 S 里用一个异步函数来满足它：协议里的同步函数不具备放弃当前线程的能力，因此，你不能用一个要求该能力的函数去实现它。

## Notification

对于所有的在“未来发生的事件”，都可以用异步函数和异步队列进行抽象。除了协议和代理范式外，Notification 也是 Apple 平台开发中用来处理未来事件的常用工具。在 Foundation 中，现在 Notification 也可以用异步序列来表征了：

```
extension NotificationCenter {
    func notifications(
        named name: Notification.Name,
        object: AnyObject? = nil
    ) -> NotificationCenter.Notifications

    class Notifications : AsyncSequence {
        typealias Element = Notification
        // ...
    }
}
```

相比于传统的基于 selector 的 Notification，使用异步序列能让相关代码更加紧凑：我们不再需要添加新的方法并用 @objc 将它暴露给通知中心和 Objective-C 的运行时；在对多个事件进行过滤和变形时，也不再需要新加属性，而是可以使用各种异步序列的扩展方法（比如 filter, map 等）来更有效地表达意图。

```
Task {
    let backgroundNotifications =
        NotificationCenter.default.notifications(
            named: UIApplication.didEnterBackgroundNotification,
            object: nil
        )
    for await notification in backgroundNotifications {
```

```
    print(notification)
}
}
```

不过要注意，使用异步序列处理 Notification 时，Task 和 for await 所导致的程序暂停，将会把还没执行的部分作为续体，并持有调用它们的上下文。也就是说，虽然在 Task 闭包中我们并没有明确写出 self，但在序列没有完成时，self 还是会一直被持有，无法得到释放。如果我们在 UIViewController 这样的环境中监听某个没有明确完结的通知的话，这个泄漏所造成的问题将无法忽视。

在获取到想要的通知后，立即跳出异步序列或是取消 Task，对避免意外的长时间持有会有帮助。比如上例中，如果我们只关心第一次事件，那么完全可以在获取到序列中首个事件后，立即 break 跳出 for await 循环，这会让相关任务结束：

```
for await notification in backgroundNotifications {
    print(notification)
    break
}
```

或者使用 first 来把异步序列收敛到一个异步值：

```
if let notification = await backgroundNotifications
    .first(where: { _ in true })
{
    print(notification)
}
```

不过需要注意的是，这两种方式都假设了序列至少会产生一个值。在产生首个值之前，调用者依然会被持有。在某些情况下，这可能是我们所希望的行为。但在另外的情况下，如果我们并不希望这个持有行为，则可以利用 Task 的 cancel 来让序列提前终结，来避免泄漏：

```
let task = Task {
    let backgroundNotifications = // ...
    for await // ...
```

```
}
```

```
// 稍后, 比如 dismiss 当前 view controller 时
task.cancel()
self.dismiss(animated: true)
```

## 异步函数的运行环境

和可抛出错误的函数一样, 异步函数也具有“传染性”: 由于运行一个异步函数可能会带来潜在的暂停点, 因此它必须要用 `await` 明确标记。而 `await` 又只能在 `async` 标记的异步函数中使用。于是, 将一个函数转换为异步函数时, 往往也意味着它的所有调用者也需要变成异步函数。

处理 `throws` 时, 在最上层, 我们会使用 `do` 的代码块来提供一个可抛出的环境, 并在 `catch` 中捕获错误。类似地, 对于异步函数的使用, 我们也可以“追溯”到一个最上层: 它作为初始环境, 为其他的异步函数运行提供合适的环境。在本节中我们来对这些运行环境进行一些讨论。

## Task 相关 API

将代码从同步世界“转接”到异步世界时, 最重要也是最常使用的方法是利用 `Task` 的相关 API 创建任务环境。在本书前面的章节, 我们也看到过一些例子了。`Task.init` 和 `Task.detached` 都能在当前环境中创建一个非结构化的任务上下文, 它们的主要区别在于是否从当前上下文(如果存在的话)中继承一些特性。简单来说, 如果你想要在当前同步上下文中, 开启一个异步上下文来调用异步方法的话, 大多数情况下 `Task.init` 是最佳选择, 这个初始化方法接受一个类型为 `() async -> Success` 的异步闭包, 你可以在里面调用其他的异步函数:

```
func asyncMethod() async -> Bool {
    await Task.sleep(NSEC_PER_SEC)
    return true
}

func syncMethod() {
    Task {
        await asyncMethod()
```

```
    }
}
```

这个异步闭包的返回值是 Success，它也会作为 Task 执行结束后的结果值，被传送到自身上下文之外。如果你是在一个异步上下文中创建 Task 的话，还可以通过访问它的 value 属性来获取任务结束后的“返回值”：

```
func anotherAsyncMethod() async {
    let task = Task {
        await asyncMethod()
    }
    let result = await task.value
    print(result) // true
}
```

在后面一章，我们会详细讲解 Task 类型和结构化并发的概念。在那里我们会看到如何利用任务上下文对异步函数进行调度以及取消，也包括 Task.init、Task.detached 和另外一些任务相关的 API 的异同。

## @main 提供异步运行环境

如果你要创建的不是一个 iOS 或者 macOS app，而是一个 Swift 的命令行工具或者 server 端程序的话，会需要一个明确的 main 函数作为入口。从 Swift 5.3 开始，可以使用 @main 来标记一个基于类型的程序入口。在引入 Swift 并发后，对于被标记的 @main 类型，我们可以直接将 main 函数声明为 async。这样一来，程序开始时我们就可以拥有一个异步运行环境了：

```
@main
struct MyApp {
    static func main() async {
        await Task.sleep(NSEC_PER_SEC)
        print("Done")
    }
}
```

一切异步函数都需要自己的任务运行环境，`main` 也不例外。`@main` 所标记的类型作为程序入口，会被整个程序传统意义上“真正的”`main` 函数（它是一个同步函数）调用。上面的程序编译后，相当于在真正的`main` 中执行了：

```
func main() {  
    _runAsyncMain { await MyApp.main() }  
}
```

`_runAsyncMain` 的实现是开源在 Swift 项目仓库中的，在 Apple 平台中，它被执行时将使用 `Task.detached` 创建一个异步运行环境，并保证将 `MyApp.main()` 放到主线程进行运行。因此，`@main` 提供的异步环境和我们自行通过 `Task.init` 或 `Task.detached` 创建的环境并没有什么本质不同。

除了 `@main` 标记的基于类型的程序入口外，我们也可以直接在 `main.swift` 顶层调用异步函数。实际上这种做法 Swift 也会用相同的方式为我们创建一个游离的任务环境，在此不再赘述。

## SwiftUI

为了能在 SwiftUI 中简单地使用异步函数，Apple 为 `View` 添加了一个 `task modifier`：

```
extension View {  
    func task(  
        _ action: @escaping () async → Void  
    ) → some View  
}
```

它被调用的时机和 `onAppear` 相同，允许我们在 `View` 出现在 `View` 层级上时，执行一个异步操作：

```
@State private var result = ""  
  
var body: some View {
```

```
ProgressView()
    .task {
        let value = try? await load()
        result = value ?? "<nil>"
    }
    Text(result)
}

func load() async throws -> String {
    // 模拟加载时间，比如从网络获取数据
    try await Task.sleep(nanoseconds: NSEC_PER_SEC)
    return "Hello World"
}
```

由于 task 和 onAppear 的调用时机相同，一个当然的疑问是，为什么需要一个新的 task modifier，它仅仅是为了书写简便而设置的语法糖吗？能不能直接在现有的 onAppear 中使用 Task.init 来开启任务呢？

它们之间有一些区别：task modifier 的任务上下文将和它所修饰的 View 的生命周期绑定：当被修饰的 View identifier 改变（比如被其他 View 取代）或者被从屏幕上移除时时，task 所关联的任务也将被取消；而 onAppear 和 Task.init 所创建的任务，则和 View 的生命周期无关。

承接上面 body 的例子，假设除了 result，我们还有一个 loading State 来控制是否正在加载：

```
@State private var result = ""
@State private var loading = true
```

使用 loading 控制是否显示 ProgressView，并在加载完成后将 loading 置为 false：

```
var body: some View {
    if loading {
        ProgressView()
            .task {
                let value = try? await load()
```

```
    result = value ?? "<nil>"  
    loading = false  
}  
}  
Text(result)  
}
```

在 1 秒的加载时间后，我们可以看到屏幕上显示 "Hello World"。不过，如果我们在 Text(result) 上进行一些修改，比如在显示时就将 loading 设为 false 的话：

```
var body: some View {  
    // ...  
    Text(result)  
        .onAppear { loading = false }  
}
```

我们会在屏幕上立即看到 "<nil>"，这是因为当 loading 为 false 时，ProgressView 将从屏幕上消失，与其绑定的任务随之取消，Task.sleep(nanoseconds:) 抛出错误，导致 try? await load() 的结果为 nil。

和无条件等待完成的 Task.sleep(\_) 不同，Task.sleep(nanoseconds:) 将尊重 Task 的取消，并在被取消时暂停休眠并抛出错误。我们在 Task 相关的详细解说中会看到更多有关任务取消的话题。

如果不希望这个取消行为和 View 的生命周期绑定，那么一个“粗暴”的做法，就是在 onAppear 中管理自己的 Task：

```
if loading {  
    ProgressView()  
        .onAppear {  
            Task {  
                let value = try? await load()  
                result = value ?? "<nil>"  
                loading = false  
            }  
        }  
}
```

```
    }  
}  
}
```

这样，即使 Progress 不再存在，Task 依然会等待加载完成。

不过，想要保持任务不被取消的一种更推荐的方法，是保持 View 的稳定：不再使用 if loading 语句来真正地移除一个 View，而只是使用 modifier 来控制 View 的视觉效果：

```
ProgressView()  
    .opacity(loading ? 1.0 : 0.0)  
    .task {  
        // ...  
    }
```

这是一个生造的例子，单看这个例子，可能并没有什么实际的意义。实际情况往往要比这种例子复杂得多。但无论如何，特别关注 task 的生命周期和 View 绑定的情况，也许会为我们在将来解决相关问题时提供一些思路。

## 小结

在 Swift 5.5 引入异步函数和异步序列的语法支持之上，Apple 为一些常见任务添加了异步版本的 API。在正确的情景下使用这些异步 API，将大幅提升代码可读性和可维护性。另外，这些异步 API 也为书写正确的并发代码提供了很好的基础：在任务上下文中调度这些异步 API，使正确的并发操作成为可能。

虽然我们还没有深入解说，但是在本章中我们已经看到了一些取消任务的例子。不管是 URLSession 所提供的异步请求，还是 Notification 在订阅后所获取的异步序列，又或是 Task.sleep(nanoseconds:) 中的挂起，当它们所在的任务上下文被取消时，它们会相应地作出合适的行为：取消网络请求，停止通知订阅，或是唤醒被挂起的任务等等。这些行为并不是免费获得的，它们需要异步 API 的设计者遵守一定规则进行实现。Apple 所提供和设计的这些异步 API 提供了相当稳定和良好的实现，它们将成为其他开发者的模板。在后面的章节中，我们

会介绍一些在任务环境中设计正确异步 API 的技巧，希望它们能成为读者在设计异步 API 时的有效参考。

# 结构化并发

6

async/await 所引入的异步函数的简单写法，可以在暂停点时放弃线程，这是构建高并发系统所不可或缺的。但是异步函数本身，其实并没有解决并发编程的问题。结构化并发 (structured concurrency) 将用一个高效可预测的模型，来实现优雅的异步代码的并发。

在本书一开始，我们已经通过概览看到过如何使用 Task 的和相关 API 来组织子任务完成并发代码的例子了。本章中我们会对结构化并发的思想和其中一些细节进行探索。

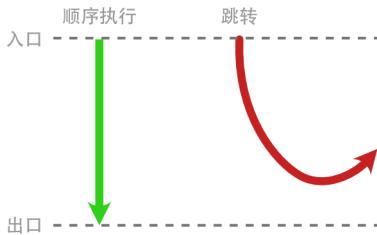
## 什么是结构化

“结构化” (structured) 这个词天生充满了美好的寓意：一切有条不紊、充满合理的逻辑和准则。但是结构化并不是天然的：在计算机编程的发展早期，所使用的汇编语言，甚至到 Fortran 和 Cobol 中，为了更加契合计算机运行的实际方式，只有“顺序执行”和“跳转”这两种基本控制流。使用无条件的跳转 (goto 语句) 可能会让代码运行杂乱无章。在戴克斯特拉的《GOTO 语句有害论》之后，关于是否应该使用结构化编程的争论持续了一段时间。在今天这个时间点上，我们已经可以看到，结构化编程取得了全面胜利：大部分的现代编程语言已经不再支持 goto 语句，或者是将它限制在了极其严苛的条件之下。而基于条件判断 (if)，循环 (for/while) 和方法调用的结构化编程控制流已经是绝对的主流。

不过当话题来到并发编程时，我们似乎看到了当年非结构化编程的影子。也许我们正处在与当年 goto 语句式微的同样的历史时期，也许我们马上会见证一种更为先进的编程范式成为主流。在深入到具体的 Swift 结构化并发模型之前，我们先来看看更一般的结构化编程和结构化并发之间的关系。

### goto 语句

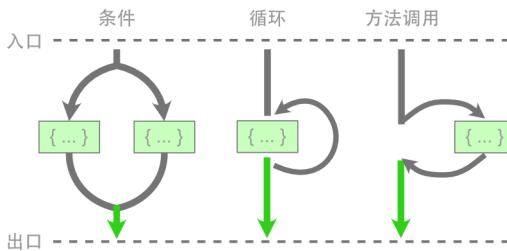
goto 语句是非结构化的，它允许控制流无条件地跳转到某个标签。虽然现在看来 goto 语句已经彻底失败，完全不得人心，但是受限于编程语言的发展，goto 语句在当时是有其生存土壤的。在还没有发明代码块的概念 (也就是 { ... }) 之前，基于顺序执行和跳转的控制流，不仅是最简单的天然选择，也完美契合 CPU 执行指令的方式。顺序执行的语句非常简单，它总可以找到明确的执行入口和出口，但是跳转语句就不一定了：



程序开发的初期，控制流的设计更多地选择了贴近实际执行的方式，这也是 `goto` 语句被大量使用的主要原因。不过 `goto` 的缺点也是相当明显的：不加限制的跳转，会导致代码的可读性急剧下降。如果程序中存在 `goto`，那么就可能在任何时候跳转到任何部分，这样一来，程序就并不是黑匣子了：程序的抽象被破坏，你所调用的方法并不一定会把控制权还给你。另外，多次来回跳转，往往最后会变成面条代码，在调试程序时，这会是每个程序员的噩梦。

## 结构化编程

在代码块的概念出现后，一些基本的封装带来了新的控制流方式，包括我们今天最常使用的条件语句、循环语句以及函数调用。由它们所构成的编程范式，即是我們所熟悉的结构化编程：



实际上，这些控制流也可以使用 `goto` 语句来实现，而且一开始人们也认为这些新控制流仅只是 `goto` 的语法糖。不过相比于 `goto`，新控制流们拥有一个非常显著的特点：控制流从顶部入口开始，然后某些事情发生，最后控制流都在底部结束。除非死循环，否则从入口进入的代码最终一定会执行达到出口。

这不仅让代码的思维模型变得很简单，也为编译器在低层级进行优化提供了可能。如果代码作用域里没有 `goto`，那么在出口处，我们就可以确定在代码块中申请的本地资源肯定不会再被需要。这一点对于回收资源（比如在 `defer` 中关闭文件、切断网络，甚至是自动释放内存等）是至关重要的。

完全禁止使用 `goto` 语句已经成为了大部分现代编程语言的选择。即使有少部分语言还支持 `goto`，它们也大都遵循高德纳（Donald Ervin Knuth）所提出的前进分支和后退分支不得交叉的理论。像是 `break`, `continue` 和提前 `return` 这样的控制流，依然遵循着结构化的基本原则：代码拥有单一的入口和出口。事实上我们今天用现代编程语言所写的程序，绝大部分都是结构化的了。当今，结构化编程的习惯已经深入人心，对程序员们来说，使用结构化编程来组织代码，早已如同呼吸一般自然。

## 非结构化的并发

不过，程序的结构化并不意味着并发也是结构化的。相反，Swift 现存的并发模型面临的问题，恰恰和当年 `goto` 的情况类似。Swift 当前的并发手段，最常见的要属使用 `Dispatch` 库将任务派发，并通过回调函数获取结果：

```
func foo() -> Bool {
    bar(completion: { print($0) })
    baz(completion: { print($0) })

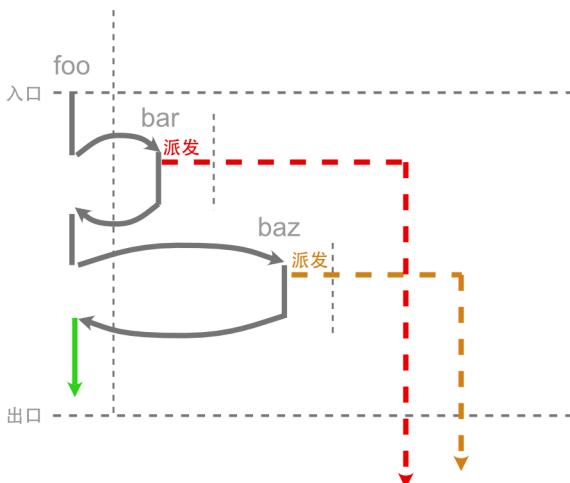
    return true
}

func bar(completion: @escaping (Int) -> Void) {
    DispatchQueue.global().async {
        // ...
        completion(1)
    }
}

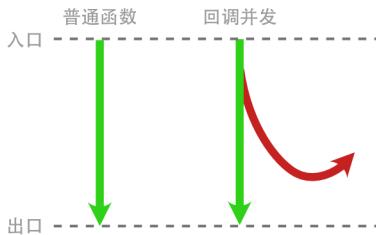
func baz(completion: @escaping (Int) -> Void) {
    DispatchQueue.global().async {
```

```
// ...
completion(2)
}
}
```

bar 和 baz 通过派发，以非阻塞的方式运行任务，并通过 completion 汇报结果。对于调用者的 foo 来说，它作为一段程序，本身是结构化的：在调用 bar 和 baz 后，程序的控制权，至少是当前线程的控制权，会回到 foo 中。最终控制流将到达 foo 的函数块的出口位置。但是，如果我们将视野扩展一些，就会发现在并发角度来看，这个控制流存在很大隐患：在 bar 和 baz 中的派发和回调，事实就是一种函数间无条件的“跳转”行为。bar 和 baz 虽然会立即将控制流交还给 foo，但是并发执行的行为会同时发生。这些被派发的并发操作在运行时中，并不知道自己是从哪里来的，这些调用不存在于，也不能存在于当前的调用栈上。它们在自己的线程中拥有调用栈，生命周期也和 foo 函数的作用域无关：



在 foo 到达出口时，由 foo 初始化的派发任务可能并没有完成。在派发后，实际上从入口开始的单个控制流将被一分为二：其中一个正常地到达程序出口，而另一个则通过派发跳转，最终“不知所踪”。即使在一段时间后，派发出去的操作通过回调函数回到闭包中，但是它并没有关于原来调用者的信息（比如调用栈等），这只不过是一次孤独的跳转。



除了使代码的控制流变得非常复杂以外，这样的非结构化并发还带来了另一个致命的后果：由于和调用者拥有不同的调用栈，因此它们并不知道调用者是谁，所以无法以抛出的方式向上传递错误。在基于回调的 API 中，一般将 Error 作为回调函数的参数传递。慵懒的开发者们总会有意无意忽视掉这种错误，Swift 5.0 中加入的 Result 缓解了这一现象。但是在未来某个未知的上下文中处理“突如其来”的错误，即便对于顶级开发者来说，也不是一件轻而易举的事情。

结构化并发理论认为，这种通过派发所进行的并行，藉由时间或者线程上的错位，实际上实现了任意的跳转。它只是 goto 语句的“高级”一些的形式，在本质上并没有不同，回调和闭包语法只是让它丑陋的面貌得到了一定程度遮掩。

除了回调和闭包，我们也有另外的一些传统并发手段，比如协议和代理模式或者 Future 和 Promise 等，但是它们实际上和回调并没有什么区别，在并发模型上带来的“随意跳转”是等价的。

## 结构化并发

并发程序是很难写好的，想正确地设计一个复杂并发更是难上加难。不过，你有没有怀疑过，这可能并不是我们智商上有什么问题，而是我们所使用的工具并不那么趁手如意？并发难写的原因，也许只是和当年 goto 一样，是我们没有发明合适的理论。

goto 最大的问题，在于它破坏了抽象层：当我们封装一个方法并进行调用时，我们所做的事情是相信这个方法会为我们完成它所声称的事情，把它看作一个黑盒。但是如果存在 goto，这个抽象假设就不再有效。你必须仔细深入到黑盒里面，去研究它的跳转方式：因为黑盒并不一定会乖乖地把控制权还给你，而是会把调用控制流引到其他任意地方去。

非结构化的并发面临类似的问题：一旦我们的并发框架中允许使用派发回调模式，那么我们在调用任意一个函数时，我们都会存在这样的担忧：

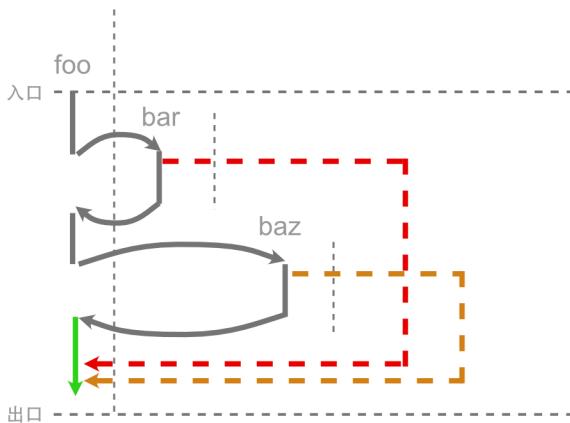
- 这个函数会不会产生一个后台任务？
- 这个函数虽然返回了，但是它所产生的后台任务可能还在运行，它什么时候会结束，它结束后会产生什么样的行为？
- 作为调用者，我应该在哪里、以怎样的方式处理回调？
- 我需要保持这个函数用到的资源吗？后台任务会自动去持有这些资源吗？我需要自己去释放它们吗？
- 后台任务是否可以被管理，比如想要取消的话应该怎么做？
- 派发出去的任务会不会再去派发别的任务？别的这些任务会被正确管理吗？如果取消了这个派发出去的任务，那些被二次派发的任务也会被正确取消吗？

这些答案并没有通用的约定，也没有编译器或运行时的保证。你很可能需要深入到每个函数的实现去寻找答案，或者只能依赖于那些脆弱且容易过时的文档（前提还得有人写文档！）然后不断自行猜测。和 goto 一样，派发回调破坏了并发的黑盒。它让我们所希冀和依赖的抽象大厦轰然坍塌，让我们原本可以用来在并发程序的天空中自由翱翔的双翼霎时折断。

结构化并发并没有很长的历史，它的基本概念由 Martin Sústrik 在 2016 年首次提出，之后 Nathaniel Smith 用一篇《Go 语句有害论》笔记“致敬”了当年对 goto 的批评，并从更高层阐明了结构化并发的做法，同时给出了一个 Python 库来证明和实践这些概念。我相信 Swift 团队在设计并发模型时，或多或少也参考了这些讨论，并吸收了相关经验。就算不是唯一，Swift 现在也是少数几个在原生层面上将结构化并发加入到标准库的语言之一。

那么，到底什么是结构化并发？

如果要用一句话概括，那就是即使进行并发操作，也要保证控制流路径的单一入口和单一出口。程序可以产生多个控制流来实现并发，但是所有的并发路径在出口时都应该处于完成（或取消）状态，并合并到一起。



这种将并发路径统合的做法，带来的一个非常明显的好处：它让抽象层重新有效。foo 现在是严格“自包含”的：在 foo 中产生的额外控制流路径，都将在 foo 中收束。这个方法现在回到了黑盒状态，在结构化并发的语境下，我们可以确信代码不会跳转到结构外，控制流最终会回到掌握之中。

为了将并发路径合并，程序需要具有暂停等待其他部分的能力。异步函数恰恰满足了这个条件：使用异步函数来获取暂停主控制流的能力，函数可以执行其他的异步并发操作并等待它们完成，最后主控制流和并发控制流统合后，从单一出口返回给调用者。这也是我们在之前就将异步函数称为结构化并发基础的原因。

## 基于 Task 的结构化并发模型

在 Swift 并发编程中，结构化并发需要依赖异步函数，而异步函数又必须运行在某个任务上下文中，因此可以说，想要进行结构化并发，必须具有任务上下文。实际上，Swift 结构化并发就是以任务为基本要素进行组织的。

### 当前任务状态

Swift 并发编程把异步操作抽象为任务，在任意的异步函数中，我们总可是使用 `withUnsafeCurrentTask` 来获取和检查当前任务：

```
override func viewDidLoad() {
    super.viewDidLoad()
    withUnsafeCurrentTask { task in
        // 1
        print(task as Any) // ⇒ nil
    }
    Task {
        // 2
        await foo()
    }
}

func foo() async {
    withUnsafeCurrentTask { task in
        // 3
        if let task = task {
            // 4
            print("Cancelled: \(task.isCancelled)")
            // ⇒ Cancelled: false

            print(task.priority)
            // TaskPriority(rawValue: 33)
        } else {
            print("No task")
        }
    }
}
```

1. `withUnsafeCurrentTask` 本身不是异步函数，你也可以在普通的同步函数中使用它。如果当前的函数并没有运行在任何任务上下文环境中，也就是说，到

withUnsafeCurrentTask 为止的调用链中如果没有异步函数的话，这里得到的 task 会是 nil。

2. 使用 Task 的初始化方法，可以得到一个新的任务环境。在上一章中我们已经看到过几种开始任务的方式了。
3. 对于 foo 的调用，发生在上一步的 Task 闭包作用范围内，它的运行环境就是这个新创建的 Task。
4. 对于获取到的 task，可以访问它的 isCancelled 和 priority 属性检查它是否已经被取消以及当前的优先级。我们甚至可以调用 cancel() 来取消这个任务。

要注意任务的存在与否和函数本身是不是异步函数并没有必然关系，这是显然的：同步函数也可以在任务上下文中被调用。比如下面的 syncFunc 中，withUnsafeCurrentTask 也会给回一个有效任务：

```
func foo() async {  
    withUnsafeCurrentTask { task in  
        // ...  
    }  
    syncFunc()  
}  
  
func syncFunc() {  
    withUnsafeCurrentTask { task in  
        print(task as Any)  
        // ⇒ Optional(  
        //     UnsafeCurrentTask(_task: (Opaque Value))  
        // )  
    }  
}
```

使用 withUnsafeCurrentTask 获取到的任务实际上是一个 UnsafeCurrentTask 值。和 Swift 中其他的 Unsafe 系 API 类似，Swift 仅保证它在 withUnsafeCurrentTask 的闭包中有效。你不能存储这个值，也不能在闭包之外调用或访问它的属性和方法，那会导致未定义的行为。

因为检查当前任务的状态相对是比较常用的操作，Swift为此准备了一个“简便方法”：使用Task的静态属性来获取当前状态，比如：

```
extension Task where Success = Never, Failure = Never {
    static var isCancelled: Bool { get }
    static var currentPriority: TaskPriority { get }
}
```

虽然被定义为static var，但是它们并不表示针对所有Task类型通用的某个全局属性，而是表示当前任务的情况。因为一个异步函数的运行环境必须有且仅会有一个任务上下文，所以使用static变量来表示这唯一一个任务的特性，是可以理解的。相比于每次去获取UnsafeCurrentTask，这种写法更加简单。比如，我们可以在不同的任务上下文中使用Task.isCancelled检查任务的取消情况：

```
Task {
    let t1 = Task {
        print("t1: \(Task.isCancelled)")
    }

    let t2 = Task {
        print("t2: \(Task.isCancelled)")
    }

    t1.cancel()
    print("t: \(Task.isCancelled)")
}

// 输出:
// t: false
// t1: true
// t2: false
```

## 任务层级

上例中虽然 t1 和 t2 是在外层 Task 中重新生成并进行并发的，但是它们之间没有从属关系，并不是结构化的。这一点从 `t: false` 先于其他输出就可以看出，t1 和 t2 的执行都是在外层 Task 闭包结束后才进行的，它们逃逸出去了，这和结构化并发的收束规定不符。

想要创建结构化的并发任务，就需要让内层的 t1 和 t2 与外层 Task 具有某种从属关系。你可以已经猜到了，外层任务作为根节点，内层任务作为叶子节点，就可以使用树的数据结构，来描述各个任务的从属关系，并进而构建结构化的并发了。这个层级关系，和 UI 开发时的 View 层级关系十分相似。

通过用树的方式组织任务层级，我们可以获取下面这些有用特性：

- 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（叶子节点）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，在所有子任务正常完成或者抛出之前，父任务是不会被完成的。

当任务的根节点退出时，我们通过等待所有的子节点，来保证并发任务都已经退出。树形结构允许我们在某个子节点扩展出更多的二层子节点，来组织更复杂的任务。这个子节点也许要遵守同样的规则，等待它的二层子节点们完成后，它自身才能完成。这样一来，在这棵树上的所有任务就都结构化了。

在 Swift 并发中，在任务树上创建一个叶子节点，有两种方法：通过任务组 (task group) 或是通过 `async let` 的异步绑定语法。我们来看看两者的一些异同。

## 任务组

### 典型应用

在任务运行上下文中，或者更具体来说，在某个异步函数中，我们可以通过 `withTaskGroup` 为当前的任务添加一组结构化的并发子任务：

```
struct TaskGroupSample {
    func start() async {
        print("Start")
        // 1
        await withTaskGroup(of: Int.self) { group in
            for i in 0 ..< 3 {
                // 2
                group.addTask {
                    await work(i)
                }
            }
            print("Task added")

            // 4
            for await result in group {
                print("Get result: \(result)")
            }
            // 5
            print("Task ended")
        }

        print("End")
    }

    private func work(_ value: Int) async -> Int {
        // 3
        print("Start work \(value)")
        await Task.sleep(UInt64(value) * NSEC_PER_SEC)
        print("Work \(value) done")
        return value
    }
}
```

解释一下上面注释中的数字标注。使用 `withTaskGroup` 可以开启一个新的任务组，它的完整的函数签名是：

```
func withTaskGroup<ChildTaskResult, GroupResult>(
    of childTaskResultType: ChildTaskResult.Type,
    returning returnType: GroupResult.Type = GroupResult.self,
    body: (inout TaskGroup<ChildTaskResult>) async -> GroupResult
) async -> GroupResult
```

1. 这个签名看起来十分复杂，有点吓人，我们来解释一下。`childTaskResultType` 正如其名，我们需要指定子任务们的返回类型。同一个任务组中的子任务只能拥有同样的返回类型，这是为了让 `TaskGroup` 的 API 更加易用，让它可以满足带有强类型的 `AsyncSequence` 协议所需要的假设。`returning` 定义了整个任务组的返回值类型，它拥有默认值，通过推断就可以得到，我们一般不需要理会。在 `body` 的参数中能得到一个 `inout` 修饰的 `TaskGroup`，我们可以通过使用它来向当前任务上下文添加结构化并发子任务。
2. `addTask` API 把新的任务添加到当前任务中。被添加的任务会在调度器获取到可用资源后立即开始执行。在这里的例子里，`for...in` 循环中的三个任务会被立即添加到任务组里，并开始执行。
3. 在实际工作开始时，我们进行了一次 `print` 输出，这让我们可以更容易地观测到事件的顺序。
4. `group` 满足 `AsyncSequence`，因此我们可以使用 `for await` 的语法来获取子任务的执行结果。`group` 中的某个任务完成时，它的结果将被放到异步序列的缓冲区中。每当 `group` 的 `next` 被调用时，如果缓冲区里有值，异步序列就将它作为下一个值给出；如果缓冲区为空，那么就等待下一个任务完成，这是异步序列的标准行为。
5. `for await` 的结束意味着异步序列的 `next` 方法返回了 `nil`，此时 `group` 中的子任务已经全部执行完毕了，`withTaskGroup` 的闭包也来到最后。接下来，外层的“End”也会被输出。整个结构化并发结束执行。

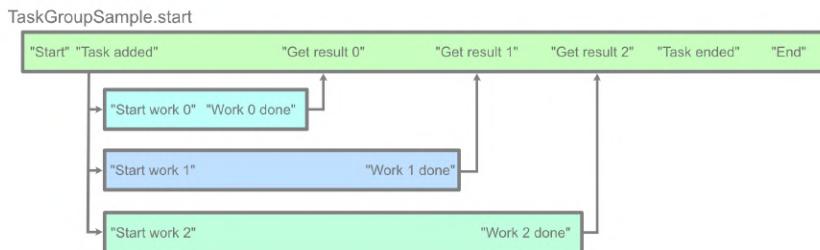
调用上面的代码，输出结果为：

```
Task {
```

```
await TaskGroupSample().start()  
}
```

```
// 输出:  
// Start  
// Task added  
// Start work 0  
// Start work 1  
// Start work 2  
// Work 0 done  
// Get result: 0  
// Work 1 done  
// Get result: 1  
// Work 2 done  
// Get result: 2  
// Task ended  
// End
```

由 `work` 定义的三个异步操作并发执行，它们各自运行在独自的子任务空间中。这些子任务在被添加后即刻开始执行，并最终在离开 `group` 作用域时再汇集到一起。用一个图表，我们可以看出这个结构化并发的运行方式：



## 隐式等待

为了获取子任务的结果，我们在上例中使用 `for await` 明确地等待 `group` 完成。这从语义上明确地满足结构化并发的要求：子任务会在控制流到达底部前结束。不过一个常见的疑问是，其实编译器并没有强制我们书写 `for await` 代码。如果我们因为某种原因，比如由于用不到这些结果，而导致忘了等待 `group`，会发生什么呢？任务组会不会因为没有等待，而导致原来的控制流不会暂停，就这样继续运行并结束？这样是不是违反了结构化并发的需要？

好消息是，即使我们没有明确 `await` 任务组，编译器在检测到结构化并发作用域结束时，会为我们自动添加上 `await` 并在等待所有任务结束后再继续控制流。比如，在上面的代码中，如果我们将 `for await` 部分删去：

```
await withTaskGroup(of: Int.self) { group in
    for i in 0 ... 3 {
        group.addTask {
            await work(i)
        }
    }
    print("Task added")

    // for await ...
    print("Task ended")
}

print("End")
```

输出将变为：

```
// Start
// Task added
// Task ended
// Start work 0
// ...
// Work 2 done
// End
```

虽然“Task ended”的输出似乎提早了，但代表整个任务组完成的“End”的输出依然处于最后，它一定会在子任务全部完成之后才发生。对于结构化的任务组，编译器会为在离开作用域时我们自动生成 await group 的代码，上面的代码其实相当于：

```
await withTaskGroup(of: Int.self) { group in
    for i in 0 ..< 3 {
        group.addTask {
            await work(i)
        }
    }
    print("Task added")
    print("Task ended")

    // 编译器自动生成的代码
    for await _ in group { }

}

print("End")
```

它满足结构化并发控制流的单入单出，将子任务的生命周期控制在任务组的作用域内，这也是结构化并发的最主要目的。即使我们手动 await 了 group 中的部分结果，然后退出了这个异步序列，结构化并发依然会保证在整个闭包退出前，让所有的子任务得以完成：

```
await withTaskGroup(of: Int.self) { group in
    for i in 0 ..< 3 {
        group.addTask {
            await work(i)
        }
    }
    print("Task added")
    for await result in group {
        print("Get result: \(result)")
        // 在首个子任务完成后就跳出
        break
    }
}
```

```
}

print("Task ended")

// 编译器自动生成的代码

await group.waitForAll()

}
```

## 任务组的值捕获

任务组中的每个子任务都拥有返回值，上面例子中 work 返回的 Int 就是子任务的返回值。当 for await 一个任务组时，就可以获取到每个子任务的返回值。任务组必须在所有子任务完成后才能完成，因此我们有机会“整理”所有子任务的返回结果，并为整个任务组设定一个返回值。比如把所有的 work 结果加起来：

```
let v: Int = await withTaskGroup(of: Int.self) { group in
    var value = 0
    for i in 0 ..< 3 {
        group.addTask {
            return await work(i)
        }
    }
    for await result in group {
        value += result
    }
    return value
}
print("End. Result: \(v)")
```

每次 work 子任务完成后，结果的 result 都会和 value 累加，运行这段代码将输出结果 3。

一种很常见的错误，是把 value += result 的逻辑写到 addTask 中：

```
let v: Int = await withTaskGroup(of: Int.self) { group in
    var value = 0
```

```
for i in 0 ..< 3 {  
    group.addTask {  
        let result = await work(i)  
        value += result  
        return result  
    }  
}  
  
// 等待所有子任务完成  
await group.waitForAll()  
return value  
}
```

这样的做法会带来一个编译错误：

```
Mutation of captured var 'value' in concurrently-executing code
```

在将代码通过 `addTask` 添加到任务组时，我们必须有清醒的认识：这些代码有可能以并发方式同时运行。编译器可以检测到这里我们在一个明显的并发上下文中改变了某个共享状态。不加限制地从并发环境中访问是危险操作，可能造成崩溃。得益于结构化并发，现在编译器可以理解任务上下文的区别，在静态检查时就发现这一点，从而从根本上避免了这里的内存风险。

更严格一些，即使只是读取这个 `var value` 值，也是不被允许的：

```
await withTaskGroup(of: Int.self) { group in  
    var value = 0  
    for i in 0 ..< 3 {  
        group.addTask {  
            print("Value: \(value)")  
            return await work(i)  
        }  
    }  
}
```

将给出错误：

```
Reference to captured var 'value' in concurrently-executing code
```

和上面修改 value 的道理一样，由于 value 可能在并发操作执行的同时被外界改变，这样的访问也是不安全的。如果我们能保证 value 的值不会被更改的话，可以把 var value 的声明改为 let value 来避免这个错误：

```
await withTaskGroup(of: Int.self) { group in
    // var value = 0
    let value = 0

    // ...
}
```

或者使用 [value] 的语法，来捕获当前的 value 值。由于 value 是值类型的值，因此它将会遵循值语义，被复制到 addTask 闭包内使用。子任务闭包内的访问将不再使用闭包外的内存，从而保证安全：

```
await withTaskGroup(of: Int.self) { group in
    var value = 0
    for i in 0 ..< 3 {
        // 用 [value] 捕获当前的 value 值 0
        group.addTask { [value] in
            let result = await work(i)
            print("Value: \(value)") // Value: 0
            return result
        }
    }
    // 将 value 改为 100
    value = 100

    // ...
}
```

```
}
```

不过，如果我们把 value 再向上提到类的成员一级的话，这个静态检查将失去作用：

```
// 错误的代码，不要这样做
class TaskGroupSample {
    var value = 0
    func start() async {
        await withTaskGroup(of: Int.self) { group in
            for i in 0 ..< 3 {
                group.addTask {
                    // 可以访问 value
                    print("Value: \(self.value)")

                    // 可以操作 value
                    let result = await self.work(i)
                    self.value += result

                    return result
                }
            }
        }
    }
}

// ...
}
```

在 Swift 5.5 中，虽然它可以编译（而且使用起来，特别是在本地调试时也几乎没有问题），但这样的行为是错误的。和 Rust 不同，Swift 的堆内存所有权模型还无法完全区分内存的借用 (borrow) 和移动 (move)，因此这种数据竞争和内存错误，还需要开发者自行注意。

Swift 编译器并非无法检出上述错误，它只是暂时“容忍”了这种情况。包括静态检测上述错误在内的完全的编译器级别并发数据安全，是未来 Swift 版本中的目标。现在，在并发上下文中

访问共享数据时，Swift 设计了 actor 类型来确保数据安全。我们在介绍后面关于 actor 的章节，以及并发底层模型和内存安全的部分后，你会对这种情况背后的原因有更深入的了解。

## 任务组逃逸

和 withUnsafeCurrentTask 中的 task 类似，withTaskGroup 闭包中的 group 也不应该被外部持有并在作用范围之外使用。虽然 Swift 编译器现在没有阻止我们这样做，但是在 withTaskGroup 闭包外使用 group 的话，将完全破坏结构化并发的假设：

```
// 错误的代码，不要这样做
func start() async {
    var g: TaskGroup<Int>? = nil
    await withTaskGroup(of: Int.self) { group in
        g = group
        // ...
    }
    g?.addTask {
        await work(1)
    }
    print("End")
}
```

通过 g?.addTask 添加的任务有可能在 start 完成后继续运行，这回到了非结构并发的老路；但它也可能让整个任务组进入到难以预测的状态，这将摧毁程序的执行假设。TaskGroup 实际上并不是用来存储 Task 的容器，它也不提供组织任务时需要的树形数据结构，这个类型仅仅只是作为对底层接口的包装，提供了创建任务节点的方法。要注意，在闭包作用范围外添加任务的行为是未定义的，随着 Swift 的升级，今后有可能直接产生运行时的崩溃。虽然现在并没有提供任何语言特性来确保 group 不被复制出去，但是我们绝对应该避免这种反模式的做法。

## async let 异步绑定

除了任务组以外，async let 是另一种创建结构化并发子任务的方式。withTaskGroup 提供了一种非常“正规”的创建结构化并发的方式：它明确地描绘了结构化任务的作用返回，确保在闭包内部生成的每个子任务都在 group 结束时被 await。通过对 group 这个异步序列进行迭代，我

们可以按照异步任务完成的顺序对结果进行处理。只要遵守一定的使用约定，就可以保证并发结构化的正确工作并从中受益。

但是，这些优点有时候也正是 `withTaskGroup` 不足：每次我们想要使用 `withTaskGroup` 时，往往都需要遵循同样的模板，包括创建任务组、定义和添加子任务、使用 `await` 等待完成等，这些都是模板代码。而且对于所有子任务的返回值必须是同样类型的要求，也让灵活性下降或者要求更多的额外实现（比如将各个任务的返回值用新类型封装等）。`withTaskGroup` 的核心在于，生成子任务并将它的返回值（或者错误）向上汇报给父任务，然后父任务将各个子任务的结果汇总起来，最终结束当前的结构化并发作用域。这种数据流模式十分常见，如果能让它简单一些，会大幅简化我们使用结构化并发的难度。`async let` 的语法正是为了简化结构化并发的使用而诞生的。

在 `withTaskGroup` 的例子中的代码，使用 `async let` 可以改写为下面的形式：

```
func start() async {
    print("Start")
    async let v0 = work(0)
    async let v1 = work(1)
    async let v2 = work(2)
    print("Task added")

    let result = await v0 + v1 + v2
    print("Task ended")
    print("End. Result: \(result)")
}
```

`async let` 和 `let` 类似，它定义一个本地常量，并通过等号右侧的表达式来初始化这个常量。区别在于，这个初始化表达式必须是一个异步函数的调用，通过将这个异步函数“绑定”到常量值上，Swift 会创建一个并发执行的子任务，并在其中执行该异步函数。`async let` 赋值后，子任务会立即开始执行。如果想要获取执行的结果（也就是子任务的返回值），可以对赋值的常量使用 `await` 等待它的完成。

在上例中，我们使用了单一 await 来等待 v0、v1 和 v2 完成。和 try 一样，对于有多个表达式都需要暂停等待的情况，我们只需要使用一个 await 就可以了。当然，如果我们愿意，也可以把三个表达式分开来写：

```
let result0 = await v0
let result1 = await v1
let result2 = await v2

let result = result0 + result1 + result2
```

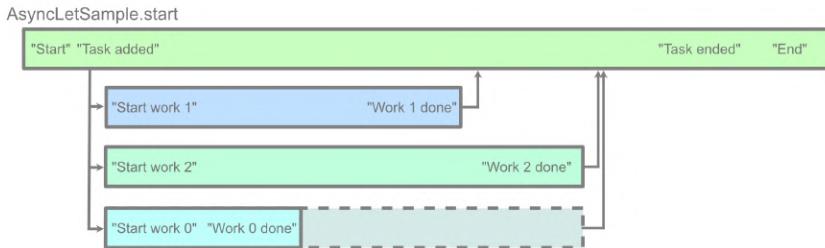
需要特别强调，虽然这里我们顺次进行了 await，看起来好像是在等 v0 求值完毕后，再开始 v1 的暂停；然后在 v1 求值后再开始 v2。但是实际上，在 async let 时，这些子任务就一同开始以并发的方式进行了。在例子中，完成 work(n) 的耗时为 n 秒，所以上面的写法将在第 0 秒，第 1 秒和第 2 秒分别得出 v0, v1 和 v2 的值，而不是在第 0 秒，第 1 秒和第 3 秒 (1 秒 + 2 秒) 后才得到对应值。

由此衍生的另一个疑问是，如果我们修改 await 的顺序，会发生什么呢？比如下面的代码是否会带来不同的时序：

```
let result1 = await v1
let result2 = await v2
let result0 = await v0

let result = result0 + result1 + result2
```

如果是考察每个子任务实际完成的时序，那么答案是没有变化：在 async let 创建子任务时，这个任务就开始执行了，因此 v0、v1 和 v2 真正执行的耗时，依旧是 0 秒，1 秒和 2 秒。但是，使用 await 最终获取 v0 值的时刻，是严格排在获取 v2 值之后的：当 v0 任务完成后，它的结果将被暂存在它自身的续体栈上，等待执行上下文通过 await 切换到自己时，才会把结果返回。也就是说在上例中，通过 async let 把任务绑定并开始执行后，await v1 会在 1 秒后完成；再经过 1 秒时间，await v2 完成；然后紧接着，await v0 会把 2 秒之前就已经完成的结果立即返回给 result0：



这个例子中虽然最终的时序上会和之前有细微不同，但是这并没有违反结构化并发的规定。而且在绝大多数场景下，这也不会影响并发的结果和逻辑。不论是前面提到的任务组，还是 `async let`，它们所生成的子任务都是结构化的。不过，它们还有些许差别，我们马上就会谈到这个话题。

## 隐式取消

在使用 `async let` 时，编译器也没有强制我们书写类似 `await v0` 这样的等待语句。有了 `TaskGroup` 中的经验以及 Swift 里“默认安全”的行为规范，我们不难猜测出，对于没有 `await` 的异步绑定，编译器也帮我们做了某些“手脚”，以保证单进单出的结构化并发依然成立。

如果没有 `await`，那么 Swift 并发会在被绑定的常量离开作用域时，隐式地将绑定的子任务取消掉，然后进行 `await`。也就是说，对于这样的代码：

```

func start() async {
    async let v0 = work(0)

    print("End")
}
  
```

它等效于：

```

func start() async {
    async let v0 = work(0)
    ...
}
  
```

```
print("End")  
  
// 下面是编译器自动生成的伪代码  
// 注意和 Task group 的不同  
  
// v0 绑定的任务被取消  
// 伪代码，实际上绑定中并没有 `task` 这个属性  
v0.task.cancel()  
// 隐式 await，满足结构化并发  
_ = await v0  
}
```

和 TaskGroup API 的不同之处在于，被绑定的任务将先被取消，然后才进行 await。这给了我们额外的机会去清理或者中止那些没有被使用的任务。不过，这种“隐藏行为”在异步函数可以抛出的时候，可能会造成很多的困惑。我们现在还没有涉及到任务的取消行为，以及如何正确处理取消。这是一个相对复杂且单独的话题，我们会在下一章中集中解释这里的细节。现在，你只需要记住，和 TaskGroup 一样，就算没有 await，async let 依然满足结构化并发要求这一结论就可以了。

## 对比任务组

既然同样是为了书写结构化并发的程序，async let 经常会用来和任务组作比较。在语义上，两者所表达的范式是很类似的，因此也会有人认为 async let 只是任务组 API 的语法糖：因为任务组 API 的使用太过于繁琐了，而异步绑定毕竟在语法上要简洁很多。

但实际上它们之间是有差异的。async let 不能动态地表达任务的数量，能够生成的子任务数量在编译时必须是已经确定好的。比如，对于一个输入的数组，我们可以通过 TaskGroup 开始对应数量的子任务，但是我们却无法用 async let 改写这段代码：

```
func startAll(_ items: [Int]) async {  
    await withTaskGroup(of: Int.self) { group in  
        for item in items {  
            group.addTask { await work(item) }  
        }  
    }  
}
```

```
for await value in group {
    print("Value: \(value)")
}
}
```

除了上面那些只能使用某一种方式创建的结构化并发任务外，对于可以互换的情况，任务组 API 和异步绑定 API 的区别在于提供了两种不同风格的编程方式。一个大致的使用原则是，如果我们要比较“严肃”地界定结构化并发的起始，那么用任务组的闭包将它限制起来，并发的结构会显得更加清晰；而如果我们只是想要快速地并发开始少数几个任务，并减少其他模板代码的干扰，那么使用 `async let` 进行异步绑定，会让代码更简洁易读。

## 结构化并发的组合

在只使用一次 `withTaskGroup` 或者一组 `async let` 的单一层级的维度上，我们可能很难看出结构化并发的优势，因为这时对于任务的调度还处于可控状态：我们完全可以使用传统的技术，通过添加一些信号量，来“手动”控制保证并发任务最终可以合并到一起。但是，随着系统逐渐复杂，可能会面临在一些并发的子任务中再次进行任务并发的需求。也就是，形成多个层级的子任务系统。在这种情况下，想依靠原始的信号量来进行任务管理会变得异常复杂。这也是结构化并发这一抽象真正能发挥全部功效的情况。

通过嵌套使用 `withTaskGroup` 或者 `async let`，可以在一般人能够轻易理解的范围内，灵活地构建出这种多层级的并发任务。最简单的方式，是在 `withTaskGroup` 中为 `group` 添加 `task` 时再开启一个 `withTaskGroup`：

```
func start() async {
    // 第一层任务组
    await withTaskGroup(of: Int.self) { group in
        group.addTask {
            // 第二层任务组
            await withTaskGroup(of: Int.self) { innerGroup in
                innerGroup.addTask {

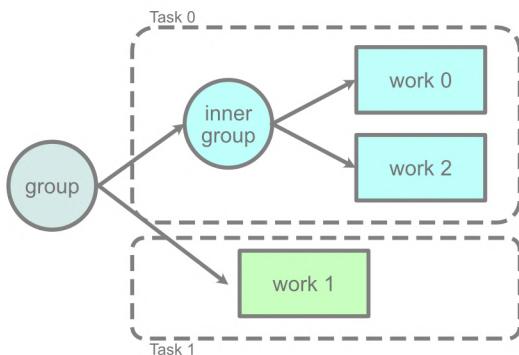
```

```
    await work(0)
}
innerGroup.addTask {
    await work(2)
}

return await innerGroup.reduce(0) {
    result, value in
    result + value
}
}

group.addTask {
    await work(1)
}
}

print("End")
}
```



对于上面使用 `work` 函数的例子来说，多加的一层 `innerGroup` 在执行时并不会造成太大区别：三个任务依然是按照结构化并发执行。不过，这种层级的划分，给了我们更精确控制并发行为。

的机会。在结构化并发的任务模型中，子任务会从其父任务中继承任务优先级以及任务的本地值(**task local value**)；在处理任务取消时，除了父任务会将取消传递给子任务外，在子任务中的抛出也会将取消向上传递。不论是当我们需要精确地在某一组任务中设置这些行为，或者只是单纯地为了更好的可读性，这种通过嵌套得到更加细分的任务层级的方法，都会对我们的目标有所帮助。

任务本地值指的是那些仅存在于当前任务上下文中的，由外界注入的值。我们会在后面的章节中针对这个话题展开讨论。

相对于 `withTaskGroup` 的嵌套，使用 `async let` 会更有技巧性一些。`async let` 赋值等号右边，接受的是一个对异步函数的调用。这个异步函数可以是像 `work` 这样的具体具名的函数，也可以是一个匿名函数。比如，上面的 `withTaskGroup` 嵌套的例子，使用 `async let`，可以简单地写为：

```
func start() async {
    async let v02: Int = {
        async let v0 = work(0)
        async let v2 = work(2)
        return await v0 + v2
    }()
    async let v1 = work(1)
    _ = await v02 + v1
    print("End")
}
```

这里在 `v02` 等号右侧的是一个匿名的异步函数闭包调用，其中通过两个新的 `async let` 开始了嵌套的子任务。特别注意，上例中的写法和下面这样的 `await` 有本质不同：

```
func start() async {
    async let v02: Int = {
        return await work(0) + work(2)
    }()
}
```

```
// ...
}
```

await work(0) + work(2) 将会顺次执行 work(0) 和 work(2)，并把它们的结果相加。这时两个操作不是并发执行的，也不涉及新的子任务。

当然，我们也可以把两个嵌套的 `async let` 提取到一个署名的函数中，这样调用就会回到我们所熟悉的方式：

```
func start() async {
    async let v02 = work02()
    // ...
}

func work02() async -> Int {
    async let v0 = work(0)
    async let v2 = work(2)
    return await v0 + v2
}
```

大部分时候，把子任务的部分提取成具名的函数会更好。不过对于这个简单的例子，直接使用匿名函数，让 `work(0)`、`work(2)` 与另一个子任务中的 `work(1)` 并列起来，可能结构会更清楚。

因为 `withTaskGroup` 和 `async let` 都产生结构性并发任务，因此有时候我们也可以将它们混合起来使用。比如在 `async let` 的右侧写一个 `withTaskGroup`；或者在 `group.addTask` 中用 `async let` 绑定新的任务。不过不论如何，这种“静态”的任务生成方式，理解起来都是相对容易的：只要我们能将生成的任务层级和我们想要的任务层级对应起来，两者混用也不会有什么问题。

## 非结构化任务

TaskGroup.addTask 和 async let 是 Swift 并发中“唯二”的创建结构化并发任务的 API。它们从当前的任务运行环境中继承任务优先级等属性，为即将开始的异步操作创建新的任务环境，然后将新的任务作为子任务添加到当前任务环境中。

除此之外，我们也看到过使用 Task.init 和 Task.detached 来创建新任务，并在其中执行异步函数的方式：

```
func start() async {
    Task {
        await work(1)
    }

    Task.detached {
        await work(2)
    }
    print("End")
}
```

这类任务具有最高的灵活性，它们可以在任何地方被创建。它们生成一棵新的任务树，并位于顶层，不属于任何其他任务的子任务，生命周期不和其他作用域绑定，当然也没有结构化并发的特性。对比三者，可以看出它们之间明显的不同：

- TaskGroup.addTask 和 async let - 创建结构化的子任务，继承优先级和本地值。
- Task.init - 创建非结构化的任务根节点，从当前任务中继承运行环境：比如 actor 隔离域，优先级和本地值等。
- Task.detached - 创建非结构化的任务根节点，不从当前任务中继承优先级和本地值等运行环境，完全新的游离任务环境。

有一种迷思认为，我们在新建根节点任务时，应该尽量使用 Task.init 而避免选用生成一个完全“游离任务”的 Task.detached。其实这并不全然正确，有时候我们希望从当前任务环境中继承一些事实，但也有时候我们确实想要一个“干净”的任务环境。比如 @main 标记的异步程序入口和 SwiftUI task 修饰符，都使用的是 Task.detached。具体是不是有可能从当前任务环境中继承属性，或者应不应该继承这些属性，需要具体问题具体分析。

创建非结构化任务时，我们可以得到一个具体的 Task 值，它充当了这个新建任务的标识。从 Task.init 或 Task.detached 的闭包中返回的值，将作为整个 Task 运行结束后的值。使用 Task.value 这个异步只读属性，我们可以获取到整个 Task 的返回值：

```
extension Task {  
    var value: Success { get async throws }  
}  
  
// 或者当 Task 不会失败时，value 也不会 throw:  
extension Task where Failure = Never {  
    var value: Success { get async }  
}
```

想要访问这个值，和其他任意异步属性一样，需要使用 await：

```
func start() async {  
    let t1 = Task { await work(1) }  
    let t2 = Task.detached { await work(2) }  
  
    let v1 = await t1.value  
    let v2 = await t2.value  
}
```

一旦创建任务，其中的异步任务就会被马上提交并执行。所以上面的代码依然是并发的：t1 和 t2 之间没有暂停，将同时执行，t1 任务在 1 秒后完成，而 t2 在两秒后完成。await t1.value 和 await t2.value 的顺序并不影响最终的执行耗时，即使是我们先 await 了 t2，t1 的预先计算的结果也会被暂存起来，并在它被 await 的时候给出。

用 Task.init 或 Task.detached 明确创建的 Task，是没有结构化并发特性的。Task 值超过作用域并不会导致自动取消或是 await 行为。想要取消一个这样的 Task，必须持有返回的 Task 值并明确调用 cancel：

```
let t1 = Task { await work(1) }
```

```
// 稍后  
t1.cancel()
```

这种非结构化并发中，外层的 Task 的取消，并不会传递到内层 Task。或者，更准确来说，这样的两个 Task 并没有任何从属关系，它们都是顶层任务：

```
let outer = Task {  
    let inner = Task {  
        await work(1)  
    }  
    await work(2)  
}  
  
outer.cancel()  
  
outer.isCancelled // true  
inner.isCancelled // false
```

单是这样的多个 Task，看起来还很简单。但是考虑到 Task.value 其实也是一种异步函数，如果我们将结构化并发和非结构化的任务组合起来使用的话，事情马上就会变得复杂起来。比如下面这个“简单”的例子，它在 `async let` 右侧开启新的 Task：

```
func start() async {  
    async let t1 = Task {  
        await work(1)  
        print("Cancelled: \(Task.isCancelled)")  
    }.value  
  
    async let t2 = Task.detached {  
        await work(2)  
        print("Cancelled: \(Task.isCancelled)")  
    }.value  
}
```

t1 和 t2 确实是结构化的，但是它们开启的新任务，却并非如此：虽然 t1 和 t2 在超出 start 作用域时，由于没有 await，这两个绑定都将被取消，但这个取消并不能传递到非结构化的 Task 中，所以两个 isCancelled 都将输出 false。

除非有特别的理由，我们希望某个任务独立于结构化并发的生命周期，否则我们应该尽量避免在结构化并发的上下文中使用非结构化任务。这可以让结构化的任务树保持简单，而不是随意地产生不受管理的新树。

不过确实也有一些情况我们会倾向于选择非结构化的并发，比如一些并不影响异步系统中其他部分的非关键操作。像是下载文件后将它写入缓存就是一个好例子：在下载完成后我们就可以马上结束“下载”这个核心的异步行为，并在开始缓存的同时，就将文件返回给调用者了。写入缓存作为“顺带”操作，不应该作为结构化任务的一员。此时使用独立任务会更合适。

## 小结

历史已经证明了，完全放弃 goto 语句，使用结构化编程，有利于我们理解和写出正确控制流的程序。而随着计算机的发展和程序设计的演进，现在我们来到了另一个重要的时间节点：我们是否应该完全使用结构化并发，而舍弃掉原有的非结构化并发模型呢？现在有这个趋势，但是大家也都还保留了原来的并发模型。即使要完全转变，可能也需要一些时间。

Swift 是当前少数几个在语言和标准库层面对结构化并发进行支持的语言之一。得益于 Swift 语言默认安全的特性，只要我们遵循一些简单的规定（比如不在闭包外传递和持有 task group 等），就可以写出正确、安全和非常易于理解的结构化并发代码。这为简化并发复杂度提供了有效的工具。withTaskGroup 和 async let 在创建结构化并发上是等效的，但是它们并非可以完全互相代替。两者有各自最适用的情景，在超出作用域的隐式行为细节上也略有不同。切实理解这些不同，可以帮助我们在面对任务时选取最合适的工具。

本章中我们只讨论了结构化并发的完成特性：父任务在子任务全部完成之前，是不会完成的。对于结构化并发来说，这只是其中一部分内容，对于另一个大的话题，任务取消，本章中鲜有涉及。在下一章里，我们会仔细探讨任务取消的相关话题，这会让我们对结构化并发在简化并发编程模型中所带来的优势，有更加深刻的理解。

# 协作式任务取消

7

并发任务往往是一些耗费时间和资源的操作，如果并发任务中途被取消了，我们会希望这些耗时耗力的操作也能及时中止。因此，对于任务的取消是并发编程中一个重要的话题。

在基于回调的派发式并发模型中，取消任务是一件非常困难的事情。并发任务可能会逃逸出当前作用范围，而且并发任务之间缺乏关联，我们往往需要自行维护各个任务之间的关系，持有那些可能被取消的任务（或者说 DispatchWorkItem），并在适当的情况下将它们停止。这其中涉及的复杂度，其实只在理论上可行，而且必然充满了各种 bug。

Operation 类型 通过封装 GCD，提供了一层任务抽象。我们可以为任务之间设定依赖关系，虽然它并不完全等价于结构化并发的任务层级，但是我们可以用任务依赖来“模拟”父任务和子任务。不过，对某个 Operation 值进行取消，并不会使取消操作在这个模拟的任务层级间自动传递，我们需要很多额外代码，才能做到正确地取消相关任务。这为在并发编程中正确处理取消，带来了巨大的难度。

对于结构化并发，事情就简单得多了。由于子任务的作用域和生命周期被完全限制，结构化的父任务和子任务之间有着天然的层级联系。父任务取消可以非常容易地传递到子任务中，这样子任务可以在不持有任务关于父任务的引用的情况下，对取消作出响应（比如清理资源等）。

不过，需要注意的是，结构化并发中取消的传递，并不意味着在任务取消时那些需要手动释放的资源可以被“自动”回收，任务本身在被取消后也并不会自动停止。Swift 并发和任务的取消，是一种基于协作式（cooperative）的取消：换句话说，组成任务层级的各个部分，包括父任务和子任务，往往需要通力合作，才能达到我们最终想要的效果。而结构化并发中取消的传递，仅仅只是协作式取消中的一个部分。

## 任务取消到底做了什么

让我们先忘掉有关结构化并发的事情，看一个最简单的顶层任务的例子。比如下面的任务中，我们每隔一秒把一个字符追加到结果中：

```
func work() async → String {
    var s = ""
    for c in "Hello" {
        // 模拟繁重工作 ...
        await Task.sleep(NSEC_PER_SEC)
```

```
    print("Append: \$(c)")  
    s.append(c)  
}  
  
return s  
}
```

和之前看到的其他例子一样，这里使用了 Task.sleep 来模拟耗时操作。我们创建一个任务，并在其中执行 work，并在一段时间后取消这个任务：

```
let t = Task {  
    let value = await work()  
    print(value)  
}  
  
await Task.sleep(UInt64(2.5 * Double(NSEC_PER_SEC)))  
t.cancel() // 2.5s
```

在 2.5s 时，我们调用了 t.cancel() 取消这个任务。但是当我们查看控制台的输出时，可以看到 t 其实执行到了最后：

```
// 输出：  
// Append: H  
// Append: e  
// Append: l  
// Append: l  
// Append: o  
// Hello
```

它似乎并没有按照我们“预想”的那样，在第三次 sleep 时中止。那么疑问是，cancel 方法到底做了什么？我们知道，Task.isCancelled 可以检查当前任务的取消状态，不妨把它加入到输出中看一看：

```
// print("Append: \$(c)")
```

```
print("Append: \u0027c\u0027, cancelled: \u0027Task.isCancelled\u0027")
```

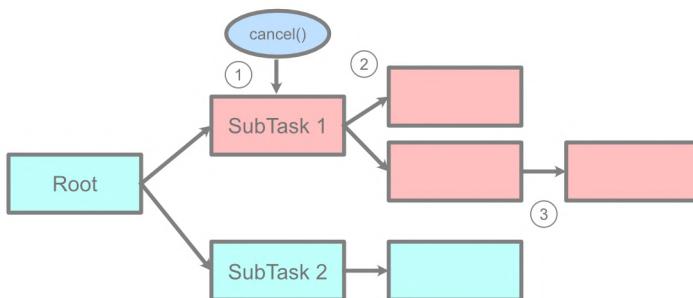
```
// 输出:  
// Append: H, cancelled: false  
// Append: e, cancelled: false  
// Append: l, cancelled: true  
// Append: l, cancelled: true  
// Append: o, cancelled: true  
// Hello
```

在第三次 sleep 结束时，任务的 isCancelled 已经是 true，这说明取消操作确实生效了，但是任务并没有停下来，还是执行到了最后。

实际上，Swift 并发中对某个任务调用 cancel，做的事情只有两件：

- 将自身任务的 isCancelled 标识置为 true。
- 在结构化并发中，如果该任务有子任务，那么取消子任务。

子任务在被取消时，同样做这两件事。在结构化并发中，取消会被传递给任务树中当前任务节点下方的所有子节点。



1. SubTask 1 和 SubTask 2 都是 Root 任务的子任务。如果对 SubTask 1 调用 cancel()，SubTask 1 的 isCancelled 被标记为 true。

2. 接下来取消被传递给 SubTask 1 的所有子任务，它们的 `isCancelled` 也被标记为 `true`。
3. 取消操作在结构化任务树中一直向下传递，直到最末端的叶子节点。

`cancel()` 调用只负责维护一个布尔变量，仅此而已。它不会涉及其他任何事情：任务不会因为被取消而强制停止，也不会让自己提早返回。这也是为什么我们把 Swift 并发中的取消叫做“协作式取消”的原因：各个任务需要合作，才能达到最终停止执行的目标。父任务要做的工作就是向子任务传递 `isCancelled`，并将自身的 `isCancelled` 状态设置为 `true`。当父任务已经完成它自己的工作后，接下来的事情就要交给各个子任务的实现，它们要负责检查 `isCancelled` 并作出合适的响应。换言之，如果谁都没有检查 `isCancelled` 的话，协作式的取消就不成立了，整个任务层级向外将呈现出根本不支持取消操作的状态。这就是为什么在我们的例子中任务一直执行到了最后的原因。

## 处理任务取消

现在让我们来看看在任务中要如何实际利用 `isCancelled` 来停止异步任务。结构化并发要求异步函数的执行不超过任务作用域，因此在遇到任务取消时，如果我们想要进行处理并提前结束任务，大致只有两类选择：

- 提前返回一个空值或者部分已经计算出来的值，让当前任务正常结束。
- 通过抛出错误并汇报给父层级任务，让当前任务异常结束。

我们分别来看看这两种处理方式。

## 返回空值或部分值

当任务的取消不影响流程，或者异步任务只能获取部分结果的情况也被考虑为正常的时候，我们可以通过提前返回空值或者部分值，来完成当前任务。通过检查 `Task.isCancelled`，我们可以做到这一点。比如将上面的 `work` 改写为：

```
func work() async → String {  
    var s = ""  
    for c in "Hello" {
```

```
// 检查取消状态
guard !Task.isCancelled else { return s }

await Task.sleep(NSEC_PER_SEC)
print("Append: \(c)")
s.append(c)

}
return s
}
```

这里的策略是，每次进行耗时操作之前，先对 `isCancelled` 进行检查。只有在 `isCancelled` 为 `false` 时，才进行操作，否则立即将当前的部分结果返回。使用和上面同样的代码，在 2.5 秒后取消任务，我们能得到预想中的结果：

```
func start() async {
    let t = Task {
        let value = await work()
        print(value)
    }

    await Task.sleep(UInt64(2.5 * Double(NSEC_PER_SEC)))
    t.cancel()
}

// 输出:
// Append: H
// Append: e
// Append: l
// Hel
```

在第四次进入 `for...in` 循环时，时间过去了 3 秒，这时任务已经被取消了。新的 `work` 实现在每次执行耗时操作时都检查了任务是否已经取消，并避免任务取消后的额外工作，并返回已经完成计算的部分结果（本例中的“Hel”）。

有时候我们希望返回一个空值来表示“没有获取到完整结果”这件事。把上面的实装调整为返回空值，是轻而易举的事情。只要把 `work` 的返回值改为 `String?`，并在 `guard` 语句里返回 `nil` 就行了。根据具体情景不同，我们需要作出不同的选择。

## 抛出错误

如果某个任务的完成情况（或者说，返回值）在并发操作中具有关键作用，其他任务必须依赖该任务确实完成才能继续进行的话，返回空值或者部分值就不再是一个可行的选项了。

举个例子，比如我们正在实现一个图片下载和缓存的框架，大体上有三个步骤：

1. 首先我们从网络下载图片数据
2. 然后把这个数据缓存到磁盘
3. 最后将图片本身提供给框架的调用者

这三个任务：下载数据、缓存数据以及提供图片，其重要程度并非对等。缓存任务和提供图片的任务是依赖于下载任务的：只有当下载数据确实完整，缓存和提供图片才有意义。但是提供图片的任务并不依赖于缓存任务：即使缓存失败了，也可以从下载的数据中生成图片。因此，在设计这些任务时，当缓存任务被取消时，我们可以选择返回部分结果或者 `nil`；但是当下载任务被取消时，我们只能抛出错误，告诉框架调用者任务无法完成。

## 约定错误和自定义错误

Swift 并发中为取消处理规定了一些约定俗成的通用方法。

回到 `work` 的例子，我们来尝试将这个例子改写为取消时抛出错误的形式。为了能抛出错误，我们必须把这个函数声明为 `throws`。在检查到任务被取消时，异步函数不再返回部分值或 `nil`，而是直接抛出一个 `CancellationError` 值：

```
func work() async throws -> String {
    var s = ""
    for c in "Hello" {
        // 检查取消状态
    }
}
```

```
guard !Task.isCancelled else {
    throw CancellationError()
}
// ...
```

CancellationError 是定义在标准库内的一个特殊的错误类型，除了一个初始化方法，它并没有暴露更多的内容：

```
struct CancellationError : Error {
    init()
}
```

这是一个标准库和开发者“约定”好了的错误类型：当任务由于被取消而抛出时，Swift 并发系统和它的使用者（也就是其他开发者们），会期望捕获到一个 CancellationError 类型的错误。这样，所有人在获取错误时，都可以使用这个共通的方式来检查这个抛出是不是由于任务取消所造成的：

```
do {
    let value = try await work()
    print(value)
} catch is CancellationError {
    print("任务被取消")
} catch {
    print("其他错误: \(error)")
}
```

区分任务取消和其他类型的错误，在最终进行错误处理的时候是很有意义的。比如实现一个大文件的下载界面时，我们可以提供一个取消按钮来中断正在进行的下载。用户点击取消按钮后，我们可以在统一的路径中处理抛出的错误，通过判断它的类型来决定是否需要在 UI 上向用户作出提示。

由于在处理任务取消时，这种“检测 isCancelled 布尔值”然后“抛出 CancellationError 错误”的模式十分常用，Swift 甚至把它们封装成了一个单独的方法，并放到了标准库中。在需要检查取消状态并抛出错误的时候，我们只需要调用 Task.checkCancellation 就可以了：

```
func work() async throws → String {  
    var s = ""  
    for c in "Hello" {  
        // 检查取消状态  
        try Task.checkCancellation()  
  
        // ...  
    }  
}
```

这种在开始一个耗时异步操作前，对取消状态进行检查和抛出的做法，是十分常见并被鼓励的。作为协作式取消的一环，我们在实现自己的 API 时，也应该遵守这个约定，通过抛出 `CancellationError` 来提早中止任务运行并清理资源。

有一种争论认为，应该通过使用返回 `Result` 的方式来表达错误。比如上例中，将 `throws -> String` 换为 `-> Result<String, Error>`，并在取消时返回 `.failure(CancellationError())`。笔者不赞同这种做法：`Result` 本身的引入，就是为了解决回调函数无法 `throw` 的“缺陷”的，而在异步函数的环境下，`throw` 成为可能，自然也就没有 `Result` 的用武之地了。

`Result` 唯一的优势在于，可以对错误类型进行限定：比如如果一个任务除了被取消外，不会以任何其他错误方式抛出，那么我们可以把返回值写为 `Result<String, CancellationError>`，来在编译期间提供更好的静态提示。这确实比单纯的 `throws` 表达了更精确的信息，但是考虑到 `Task` 相关的 API 和整个既有生态，都在使用 `throws` 来处理错误的现实，单纯为了这一点优势而放弃整个体系，似乎有点得不偿失。

## 对任务树上其他分支的影响

上面都只涉及了单个任务，接下来让我们来考虑一个复杂一点的结构化并发例子。现在的 `work` 函数需要处理的字符串是硬编码写死的“Hello”。改写一下这个函数，让它接受任意的字符串输入：

```
func work(_ text: String) async throws → String {  
    var s = ""  
    for c in text {  
        // 检查取消状态  
        try Task.checkCancellation()  
  
        // ...  
    }  
}
```

```
for c in text {
    if Task.isCancelled {
        print("Cancelled: \(text)")
    }

    try Task.checkCancellation()
    await Task.sleep(NSEC_PER_SEC)
    print("Append: \(c)")
    s.append(c)
}

print("Done: \(s)")
return s
}
```

为了让之后的行为更加清晰，我们还添加了一些额外的输出：当任务处于取消状态时，打印“Cancelled: (text)”。

接下来，我们利用这个新的 work 函数构建一棵复杂一些的结构化并发的任务树：

```
do {
    let value: String =
        try await withThrowingTaskGroup(of: String.self) {
            group in

            // Task 1
            group.addTask {
                try await withThrowingTaskGroup(of: String.self) {
                    inner in
                    // Task 1.1
                    inner.addTask { try await work("Hello") }
                    // Task 1.2
                    inner.addTask { try await work("World!") }

                    // 取消任务组 inner
                }
            }
        }
}
```

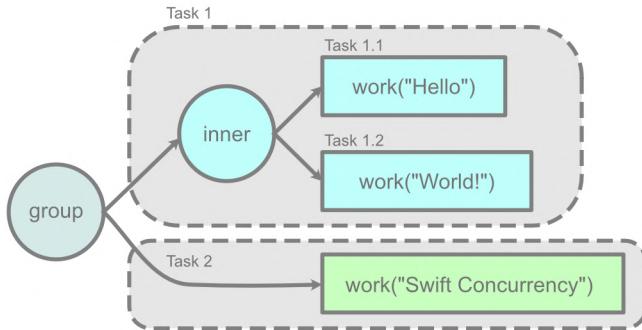
```
await Task.sleep(UInt64(2.5 * Double(NSEC_PER_SEC)))
inner.cancelAll()

return try await inner.reduce([]) {
    $0 + [$1]
}.joined(separator: " ")
}

// Task 2
group.addTask {
    try await work("Swift Concurrency")
}

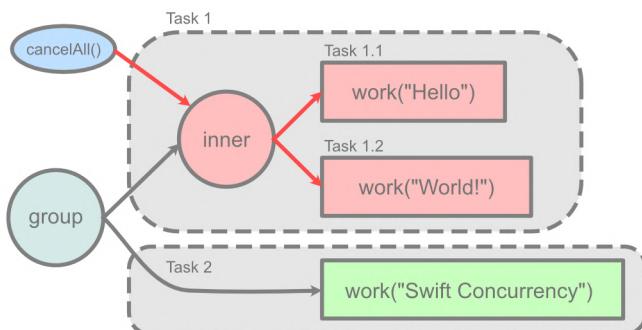
return try await group.reduce([]) {
    $0 + [$1]
}.joined(separator: " ")
}
print(value)
} catch {
    print(error)
}
```

这段代码有点复杂，不过用任务层级树的方式表达，就可以明了一些了，它等效于：



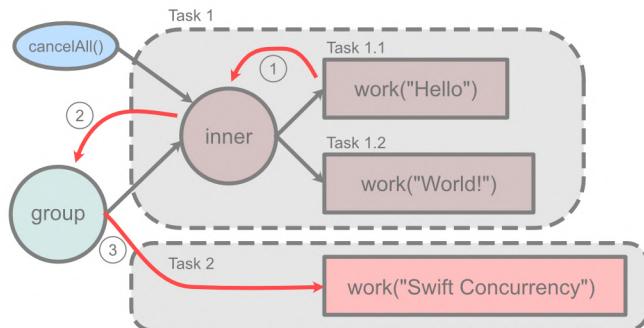
虽然代码有点长，但我相信它的结构相对还是清楚的：这段程序通过 group 创建了两个任务 Task 1 和 Task 2。其中 Task 2 以“Swift Concurrency”为输入调用 work。我们已经知道 work 的实现了，在 17 秒后（因为输入有 17 个字符），这个任务会完成并返回输入的字符串。Task 1 相对复杂一些，在它里面我们又创建了另一个任务组 inner，并以“Hello”和“World!”为输入，生成次级子任务 Task 1.1 和 Task 1.2。

这几个任务是同时开始的，如果不加干涉，这些代码最终会按照 Task 1.1, Task 1.2, Task 2 的顺序完成，最终输出“Hello World! Swift Concurrency”。不过在 2.5 秒时，我们手动调用了 inner.cancelAll()，来将整个 Task 1 取消掉。结构化并发会把取消操作向下传递：也就是说，Task 1.1 和 Task 1.2 也会被取消：



在之前介绍任务组 API 并生成任务组时，为了简洁，我们用的是 `withTaskGroup` 函数。而这里我们使用了可抛出的版本 `withThrowingTaskGroup`。Swift 并发中的取消，被视为错误抛出。因此为了支持取消操作，我们必须使用可抛出的 API 版本。

结构化并发里一个任务接受到子任务抛出的错误后，会先将其他子任务取消掉，然后再等待所有子任务结束后，把首先接到的错误抛出到更外层。在上例中：



1. 在 `work` 的实现中，我们使用了 `try Task.checkCancellation()` 检测任务的取消情况，并抛出 `CancellationError` 错误。Task 1.1 或 Task 1.2 中的这部分代码将被触发，并将错误抛给 `inner`。
2. 这个错误并没有在 `inner.addTask` 中被处理，于是它将被进一步抛出到上层，也就是 `group` 中。
3. 作为父任务，外层 `group` 在接受到 Task 1 的错误后，会主动取消掉任务树中所有的子任务，等待子任务们全部执行完毕（不论是正常返回还是抛出错误）后，再进行错误处理。在这里，`group` 中除了 Task 1 外，只有一个其他子任务 Task 2。于是，Task 2 的 `isCancelled` 也被置为 `true`，并触发 `work` 中的相关检查抛出取消错误。

在三个 Task 全部完成抛出后，`group` 离开其闭包作用域，并将它所接受到的第一个错误抛出到上层。如果运行上面的代码，得到的输出可以佐证这个行为：

```
// 输出：  
// Append: S  
// Append: H
```

```
// Append: W
// ...
// Cancelled: World!
// Append: l
// Cancelled: Hello
// Append: f
// Cancelled: Swift Concurrency
// CancellationError()
```

这是一个运行良好的协作式取消的例子：在任务树的某个部分被取消时，树上所有的耗时操作都及时停止了。这种行为满足 API 使用者的期待，也应该成为我们在设计并发系统时需要遵守的规范：我们总是应该尽可能快地对任务取消作出响应，避免额外的非必要工作，并迅速通过抛出来完成任务，将结构化并发的控制权交回给调用者。

“遵守规范”其实需要精确的设计才能实现。在设计并发系统时，即使我们没有处理取消操作，编译器也不会报错或警告。但是一旦我们没有能正确处理取消，比如忘了检查 `isCancelled` 或没有抛出错误，任务的执行可能会超出我们的想定。举个实际中不太可能出现的生硬例子，比如在上面 `work` 中任务取消时，添加一个检查条件，只在输入的字符数小于 10 的时候才考虑取消任务：

```
// try Task.checkCancellation()
if text.count < 10 {
    try Task.checkCancellation()
}
```

这样一来，`group` 对 `Task 2` 的取消将不再“有效”。结构化并发要求在任务离开作用域时，子任务们必须全部完结。虽然 `Task 2` 的 `isCancelled` 被标为 `true`，但它依然会继续执行到最后。再之后，整个 `group` 才会抛出来自 `Task 1` 的取消错误，这种行为既是对资源的浪费，往往也是违反直觉的。

## 内建 API 的取消

你可能会觉得有点儿麻烦，因为在设计并发系统时，如果我们想要尽快地响应取消，则需要在每个 await 前后添加 try Task.checkCancellation()。虽然这并不困难，但是显然是一种重复劳动和模板代码。

在上例中，Task.sleep(\_::) 本身并不支持取消：它会忠实地计数到设定的时间后再将控制流交还。不过，Swift 并发在 Task 的 API 中还提供了一个可以取消的 sleep 版本，它接受一个命名参数 nanoseconds，并被标记为 throws，以示区别：

```
extension Task where Success = Never, Failure = Never {
    static func sleep(nanoseconds duration: UInt64) async throws
}
```

在遇到取消时，sleep(nanoseconds:) 会直接中断，并抛出 CancellationError。如果我们使用这个版本的 sleep 来改写 work，则可以不再手动进行 checkCancellation：

```
func work(_ text: String) async throws -> String {
    var s = ""
    for c in text {
        if Task.isCancelled {
            print("Cancelled: \(text)")
        }
        // try Task.checkCancellation()
        // await Task.sleep(NSEC_PER_SEC)

        try await Task.sleep(nanoseconds: NSEC_PER_SEC)
        s.append(c)
        // ...
    }

    // 输出为：
    // Append: S
    // Append: H
    // Append: W
    // Append: w
```

```
// Append: o  
// Append: e  
// CancellationError()
```

对比 `sleep(_:)` 中每次在 `await` 前进行检查的版本，`sleep(nanoseconds:)` 在抛出错误时更加及时，它不需要等到当前的 `await` 结束后再进行抛出。相比于原来的处理取消的方式，`sleep(nanoseconds:)` 是更优秀的实现。

在我们实际构建一个真正的并发系统时（而不是使用 `Task.sleep` 来模拟工作时），也有类似的选择：在大多数情况下，实际的异步操作是通过使用一些系统层级提供的异步 API 来完成的。相比于自己书写代码来检查任务的取消状态，我们首先要做的应该是确认我们所使用的异步 API 是否已经支持了协作式取消。在标准库和 Foundation 中，有很多这样的例子，比如 `URLSession` 新加入的几个异步方法，都是默认支持任务取消的，使用它们时我们并不需要自己去检查 `isCancelled`，不过它们抛出的错误类型可能会根据 API 的差异也有所不同：

```
let t = Task {  
    do {  
        let (data, _) = try await URLSession.shared.data(  
            from: URL(string: "https://example.com")!,  
            delegate: nil  
        )  
        print(data.count)  
    } catch {  
        print(error)  
    }  
}  
  
await Task.sleep(100)  
t.cancel()  
  
// 输出：  
// Error Domain=NSURLErrorDomain Code=-999 "cancelled" ...
```

Apple 在标准库和 Foundation 中对协作式取消的支持，也为我们在实现自己的异步系统时提供了参考。如果我们要为其他开发者（有时候这就是我们自己！）提供异步 API 时，记得遵守协作式取消的需求，这会让所有人都更加开心和轻松。

## 取消的清理工作

我们已经了解了 Swift 并发中协作式取消的特点和实现的一般方式，不过关于结构化并发和协作式取消的话题，还有一些更细节的内容，比如如何在取消的同时合理地进行资源清理。笔者也想要对它们进行一些提示和讨论。

### defer

某些操作可能会占用资源，需要在使用完毕后及时进行清理。比如在访问沙盒外的安全作用域的 URL (security-scoped URL) 时，我们需要先调用 `startAccessingSecurityScopedResource` 来向系统请求对这个 URL 的访问权限。在使用结束后，我们需要及时调用停止方法 `stopAccessingSecurityScopedResource` 来放弃访问权限，否则将造成内核资源的泄漏。一般情况下，工作流程是首先进行申请，然后使用 URL，最后在做完事情后，放弃权限：

```
func load(url: URL) async {
    let started = url.startAccessingSecurityScopedResource()
    if started {
        await doSomething(url)
        url.stopAccessingSecurityScopedResource()
    }
}
```

得益于结构化并发，我们可以保证 `stop` 方法会被正确调用到。但是如果考虑到任务取消的情况，上面的代码就会出现泄漏：

```
func load(url: URL) async throws {
    let started = url.startAccessingSecurityScopedResource()
    if started {
        try Task.checkCancellation()
```

```
await doSomething(url)
try Task.checkCancellation()
await doAnotherThing(url)

// 调用可能没有被执行到
url.stopAccessingSecurityScopedResource()
}

}
```

在同步的世界中，为了避免在各个退出路径上重复写清理代码，我们往往使用 `defer` 来确保代码在离开作用域后进行调用。这个技巧在异步操作中也是适用的，在上面的代码中，我们只需要在 `if started` 内加上 `defer`，就可以应对取消时的资源清理工作了：

```
func load(url: URL) async throws {
let started = url.startAccessingSecurityScopedResource()
if started {
    defer {
        url.stopAccessingSecurityScopedResource()
    }

    await doSomething(url)
    try Task.checkCancellation()
    await doAnotherThing(url)
    try Task.checkCancellation()
}
}
```

在结构化并发中的 `defer`，会等到子任务 `await` 全部完成后再调用。虽然这符合我们对 `defer` 的一般认知，但是在某些情况下并不明显。比如在使用 `async let` 创建子任务，然后没有使用这些子任务，导致自动取消的情况：

```
Task {
    defer {
        print("Defer")
    }
}
```

```
}

async let v = work()

}

func work() async {
    await Task.sleep(NSEC_PER_SEC)
    print("Done")
}
```

这种情况下，结构化并发在离开 Task 作用域前，会补全对 v 的取消和 await v 的调用。即使在这种情况下，defer 也会在最后的隐式 await 之后，再进行调用，所以无论如何，你总是能够安全地在 defer 中进行清理工作。

虽然和 Swift 并发无关，但稍微值得一提的是，defer 的意思是“当退出当前代码块 {} 的作用域范围时执行 defer 中的代码”，而不是“退出当前函数时，执行代码”。所以，文件权限 start 和 stop 的例子中，stop 方法会在 if started 结束时执行，而不是等到函数完毕后再执行。在这个例子中它是正确的行为，因为我们必须要停止不再需要的访问权限。但是，如果我们把 defer 写错了地方，可能会造成不想要的结果，例如：

```
func load(url: URL) async throws {
    let started = url.startAccessingSecurityScopedResource()
    if started {
        defer { url.stopAccessingSecurityScopedResource() }
    }

    if started && shouldDoWork {
        // ...
    }
}
```

在 if started 结束后，URL 的访问权限立即就被归还了。这样，在后一个带有 shouldDoWork 的条件中，我们就无法打开这个 URL 了。

除了 defer 外，还有其他一些资源清理的方式，我们会在后面的讨论中逐渐看到。

## Cancellation Handler

在使用 `defer` 时，只有在异步操作返回或者抛出时，`defer` 才会被触发。如果我们使用 `checkCancellation` 在每次 `await` 时检查取消的话，实际上抛出错误的时机比任务被取消的时机要晚一些：在异步函数执行暂停期间的取消，并不会立即导致抛出，只有在下一次调用 `checkCancellation` 进行检查时，才进行抛出并触发 `defer` 进行资源清理。虽然在大部分情况下，这一点时间差应该不会带来问题，但是对于下面两种情况，我们可能会希望有一种更加“实时”的方法来处理取消。

- 需要在取消发生的时刻，立即作出一些响应：比如关键资源的清理，或者想要获取精确的取消时间。
- 在某些情况下，无法通过 `checkCancellation` 抛出错误时。假如使用的是外部的非 Swift 并发的异步实现（例如包装了传统的 GCD 实现等），这种时候原来的异步实现往往不支持抛出错误，或者抛出的错误无法传递到 Swift 并发中，也无法用来取消任务。

这些情况下，我们可以考虑使用 `withTaskCancellationHandler`。它接受两个闭包：一个是待执行的异步任务 `operation`，另一个是当取消发生时会被立即调用的闭包 `onCancel`：

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

这个方法并不会创建任何新的任务上下文，它只负责为当前任务提供一个在取消时被调用的闭包。因为对 `onCancel` 的调用会在任务被取消时立即发生，它可能会在任何时间任意线程上下文被调用，所以 `onCancel` 接受的函数被标记为了 `@Sendable`。如果你还有印象，在本书前面我们也遇到过一些 `Sendable` 标记的例子，比如在 `AsyncStream` 的 `onTermination` 中，我们也必须为它设置一个 `@Sendable` 的函数。`@Sendable` 是一个纯编译期间的标记，它作为提示，是对开发者的一种警示，表示被标记的函数可以在并发域之间进行传递，因此该函数只能持有那些跨并发域也安全的数据类型。这个标记只作为编译检查，它在运行时并不会有什么作用，但是如果我们的函数体没有能真的满足 `@Sendable`，但却被标记为 `@Sendable` 的话，我们很可能会引入潜在的并发危险。在本书稍后我们详细解释了 `actor` 和并发隔离域的概念后，会更完整地对 `Sendable` 进行阐明。

在实际中，有时候 `withTaskCancellationHandler` 会与 `withUnsafeThrowingContinuation` 配合使用。后者在将闭包回调的异步操作封装成异步函数时，为了能在任务取消时正确释放某些资源，会用到 `onCancel`：

```
func asyncObserve() async throws → String {
    let observer = Observer()
    return try await withTaskCancellationHandler {
        observer.start()
        return try await withUnsafeThrowingContinuation {
            continuation in
            observer.waitForNextValue { value, error in
                if let value = value {
                    continuation.resume(returning: value)
                } else {
                    continuation.resume(throwing: error!)
                }
            }
        }
    } onCancel: {
        // 取消时清理资源
        observer.stop()
    }
}
```

如果没有 `withTaskCancellationHandler`，我们在封装这种带有“取消”功能的异步操作时，将不得不以轮询的方式，在 `continuation.resume` 之前去不断检查 `Task.isCancelled`，这会让取消变得不及时，甚至导致如果新的事件不发生的话，持有的资源就永远无法释放。相比起来，`onCancel` 给了我们更加正确和优雅的解决方式。

## 异步序列的取消

异步序列协议最重要的部分，就是 `AsyncIterator` 所定义的 `next() async throws` 函数。这个函数已经被标记为 `throws` 了，因此和其他的异步操作一样，我们可以选择在实现 `next()` 时检查任务是否已经取消，并抛出相应的错误：

```
mutating func next() async throws → Int? {  
    try Task.checkCancellation()  
    return try await getNextNumber()  
}
```

这样，当通过 for try await 运行的异步序列的 Task 被取消时，在序列中计算下一个元素时，序列将会抛出并终结。

当然，和普通的异步函数取消相同，对异步序列也还有另一种选择：让 next 返回 nil 使序列正常终结。异步序列本身的目的就是产生一系列值，所以相对于抛出错误，这种“正常返回”的处理方式，有时候可能更切合于异步序列的初衷。这在无穷序列中更加常见：当任务取消时，序列也随之取消，不再产生新值。这个序列在取消时可能已经产生了部分值，并处理了我们所需要的逻辑。而取消时，如果我们不太关心后续的值，那么选择不抛出错误，代码就可以按照正常路径退出。在《使用异步函数》一章中，异步的 NotificationCenter 方法所给出的异步序列就是典型的例子，它们不会在取消时抛出错误，而只是默默结束序列。

## 隐式等待和任务暂停

### 结构化并发的潜在暂停点

在介绍异步函数时，我们提到过，await 代表潜在暂停点。我们需要特别注意在 await 前后，异步函数的执行上下文可能发生变化，这包括任务的取消状态。因此，如果我们选择使用 isCancelled 或 checkCancellation 检查任务取消的话，await 会是一个很好的标志：在 await 前后对任务的取消状态进行检查，是一种省心省力的做法。

不过，在结构化并发中，会存在隐式 await 的情况。我们在前面已经说过，在 TaskGroup 中，如果我们没有明确地等待 group 中的任务，它们将会在离开 group 作用域前被隐式等待：

```
let t = Task {  
    do {  
        try await withThrowingTaskGroup(of: Int.self) { group in  
            group.addTask { try await work() }  
            group.addTask { try await work() }  
        }  
    }  
}
```

```
        }
    } catch {
        print("Error: \(error)")
    }
}

await Task.sleep(NSEC_PER_SEC)
t.cancel()

func work() async throws → Int {
    try await Task.sleep(nanoseconds: 3 * NSEC_PER_SEC)
    print("Done")
    return 1
}
```

运行上面的代码，你既看不到 work 中的“Done”输出，也看不到 catch 块中“Error”的输出。这是因为我们没有明确对 group 进行 try await 操作。try await work 只存在 addTask 内，它的抛出会向上传递到 group 中，但由于我们没有明确地 try await group，这个错误并不会继续传递到 withThrowingTaskGroup 的外层。在离开作用域时的隐式等待，会选择自行消化这个错误，而不是进行抛出，这一点并不是特别明显。上面这样看似覆盖完整的代码分支却两边都不执行，这种情况非常难以进行调试和理解。

没有完整 await 的 group 所面临的假设和情况，要比完整写出 await 的时候复杂得多。所以笔者建议，不论我们最终需不需要子任务的返回值，都应该保持明确写出对 group 等待操作的好习惯。比如，在离开作用域时补上 try await，就可以让 catch 代码块在接收到取消时正确工作：

```
do {
    try await withThrowingTaskGroup(of: Int.self) {
        group in
        group.addTask { try await work() }
        group.addTask { try await work() }
        try await group.waitForAll()
    } catch {
        print("Error: \(error)")
    }
}
```

```
}
```

对于没有使用的 `async let` 异步绑定值，情况有些类似。不过要注意 `async let` 会直接先取消，再进行隐式等待，这和 `group` 子任务的行为不同。如果我们确实需要不加取消地执行某个子任务，用样明确地 `await` 它会是最好的选择。

不论任务是否已经被取消，在 `group` 通过 `addTask` 追加子任务后，子任务将立即开始执行。如果我们不希望在任务已经结束时还创建新的子任务，可以使用 `addTaskUnlessCancelled` 来在相应的情况下跳过子任务的追加：

```
let added = group.addTaskUnlessCancelled { try await work() }
```

```
// 等价于
if !Task.isCancelled {
    group.addTask { try await work() }
}
```

当子任务是一个非常消耗资源，且不能中途取消的时候，使用 `addTaskUnlessCancelled` 在很多情况下可以减少资源使用的足迹。但是，需要特别注意，如果你的代码逻辑依赖于子任务的成功时，相比于使用 `addTask`，`addTaskUnlessCancelled` 可能会带来不同的结果：使用 `addTask`，子任务一定会被添加。不过，被添加后，由于父任务已经取消这一事实，子任务中如果有 `checkCancellation` 调用的话，它会被立即抛出，并让整个 `group` 执行抛出错误；但是，如果在任务已经取消的情况下使用 `addTaskUnlessCancelled` 的话，任务根本就不会被加入到 `group` 里，也就不存在任务取消的错误：对这样的 `group` (空的任务组) 进行 `await`，会得到“正确完成”的结果。

## 小结

协作式的任务取消是 Swift 并发中重要的一环。相比于传统的并发模型，在处理“正常路径”时，也许结构化并发的优势并没有那么明显，但是在处理错误或者取消时，取消标记在任务层级树中的传递以及检查，可以帮助我们轻而易举地写出正确、稳定和高效的复杂并发代码。这在以前的传统并发时代是难以想象的。

不过，在使用协作式取消这一新工具时，我们也需要承担更多的责任。如果要实现自己的并发系统，我们需要确保异步任务能够正确处理协作式取消。只有这样，我们的并发系统才能符合 Swift 并发体系的规范和要求。这在别人使用我们创建的并发系统，以及把这个系统集成到别的并发系统中时，是十分关键的。

在任务中抛出错误或者处理取消，意味着我们会提前退出任务上下文。这为我们带来了另一个重要的话题：资源清理。结构化并发保证了并发操作的生命周期不会超过函数作用域，这为资源清理带来了巨大的便利，我们可以确保在退出任务时，没有任何子任务还在运行并需要这些资源。得益于异步函数的特点和结构化并发模型对异步操作生命周期的规定，我们可以用与处理同步函数类似的方式（比如 `defer`），把原本需要分散在各个地方的清理代码进行统合。这进一步降低了创建并发系统的难度，也减少了在程序中不小心写出 bug 的可能性。

# actor 模型和数 据隔离

8

相比于年轻的结构化并发，actor 模型的概念可以说和并发编程几乎一样古老了。卡尔·休伊特 (Carl Hewitt) 早在上世纪 70 年代就提出了 actor 模型的理论，2006 年时他将这套理论进一步发展并实用化，针对符合现代语言的特点进行了改进。现如今，许多语言和框架都已经实践了 actor 模型，像是 Erlang 中的 process，Java 或 Scala 中的 Akka 框架，或者是 Ruby 的 celluloid 项目，都是 actor 模型的成功运用。

和大部分支持多线程并发操作的语言一样，Swift 也面临着线程安全和数据竞争的问题。而这也正是 actor 模型所擅长解决的。Swift 并发中汲取了其他语言和框架中的成功经验 (Apple 甚至把 Akka 的维护者挖到了 Swift 的核心团队中)，把 actor 模型的支持带入到了原生语言中。配合异步函数，actor 模型完成了并发这一课题的最后一块拼图，并依赖编译器保证了并发数据的安全，大大降低了写出正确无竞争的并发代码的难度。

在本章中，我们会仔细看看 actor 模型想要解决的问题到底是什么，以及它在 Swift 并发中使用上的一些注意事项。

## 共享内存模型的困境

### 面向对象的封装缺陷

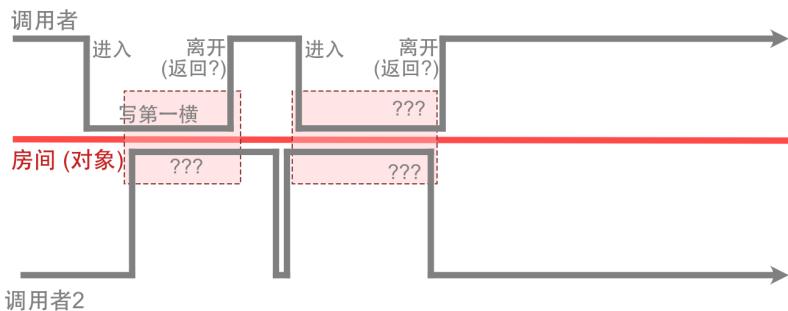
对于单线程的程序，同一时间内只会有一段代码在执行，对内存中状态的访问和改变都是独占发生的。在“远古时期”，程序的执行相对简单，也没有那么多并发的需求，因此使用面向对象编程 (OOP) 模式对事物进行抽象是合理的。

为了方便理解，我们来将抽象的概念再具象化一些。把一个对象想象为一间房间，房间里有一张纸，上面会用画“正”字的方式记录这个房间被访问的次数。知道这个房间房号的人，可以进入到房间里，并在纸上添加上自己的来访记录，然后把房间的总访问次数记下，并带回去。在这个例子中，画“正”字的纸是房间的内部可变状态 (mutating state)；而在纸上添加一笔，并把结果带回，是被封装在房间对象上的一个方法 (method)。

单线程下，只有一个可用的人，因此同一时间永远只会有一人访问这个房间。如果想要多次访问房间，只有等到这个人回来以后，才能把他再次派出到目标房间进行访问。这样，第一次访问时写“正”的第一横，并带回数字 1；第二次写竖，并带回数字 2；一切井井有条：



早期 CPU 的发展专注于提高时钟频率，也就是让调用者更快速地进出房间以及完成写字这一操作。但是随着硬件发展的物理瓶颈，多核多线程成为了解决性能需求的“没有办法下的办法”。多线程编程的需求催生了基于线程调度（或者说 GCD）的并发模型。在这个模型中，会存在不同的调用者（不同的线程）同时持有房间号，并一起访问这个房间的情况。房间只有一个，而调用者有若干个，他们在访问房间时，很可能出现这样的情况：前一个调用者正在写第一横，而下一个调用者“冲入”房间要在纸上记录自己的来访。在第一个来访者刚写了一半时，这时候的第二个来访者到底该在纸上记录什么呢？对于第二个来访者来说，这时候纸上的状态是他无法认知的，而两个人强行在纸上同时留下笔迹，很可能导致这张纸上的内容不再能被识别。换句话说，这张纸作废了（内存 corrupted，状态错误）。



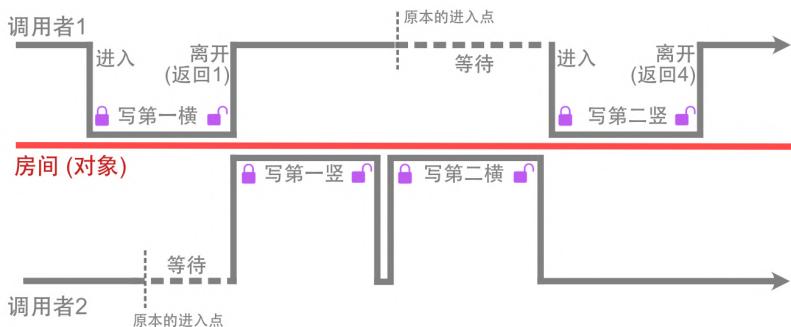
两个调用者同时访问房间的重叠时间（图中），可能造成内部状态的错误，程序在这种情况下的行为是难以预测的，崩溃或者继续以错误的状态值运行都有可能。另外，如果调用者来访的时间不发生重叠，其实就不会出现问题，这也正是这类问题在开发期间由于测试样本太小而难以发现，但是在实际发布版本中却会由于用户量上升而变得无法忽视，但又难以调试的原因。

这类错误的本质在于，复数个调用者共享了同一块资源（在这里是内存）。面向对象的封装，其实是将一块包含有内部状态的内存用 class 类型的“蓝图”进行描述。这个模型中，在多个线程中共享同一个对象，就意味着共享内存。对于内存的操作，天生就不安全。

## 锁的使用和问题

要解决内存共享的问题，最简单的想法就是让同一时间只有一个线程能够访问内存。比如，给房间门装一把锁，只有在锁打开着的情况下，访问者才能进入房间；一旦访问者进入房间，他就将把锁给锁上，直到离开房间为止。这样，保证了房间内只会有一人，而在纸上写“正”字的下一步操作，也不会被其他访问者干扰了。

锁的使用可以保证单个调用者的安全，但是这也意味着其他访问者将被“拒之门外”。后来的调用者只能等待房间里的人完成工作，这其实严重地限制了并发的可用性：系统需要暂停整个线程并在稍后恢复它，在现代 CPU 架构下这是一个非常繁重的操作。同时，后来的调用者（或者说，线程），除了等待之外无法做任何其他有意义的事情。移动设备自不必说，即使对于高性能的桌面系统，这也是难以接受的损失。



除了带来性能上的退化，锁的另一个问题在于需要精心设计。单个锁看起来问题不大，但是随着这套“安保系统”的逐渐升级，可能出现死锁（deadlock）的问题。

继续用这个房间举例，现在我们加入一个有点儿笨的电话认证系统来提升安全等级：在来访者作出任何动作（包括开门和在纸上写字）之前，都必须拨打一个特定电话进行报告，且这个电话在动作完成前无法挂断。首个来访者打电话报告进门后，把门锁上并将电话挂断，开始准备纸笔；然后第二个来访者立即到达，打电话对开门动作进行报告。但是此时门已经上锁，他无法

进入，这个电话一直会处于接通状态直到他有机会完成开门动作。第一个来访者找到房间内的纸张后，需要打电话对写字的行为进行报告，不过由于后面的来访者正在对进门动作进行通话，所以这个电话将永远占线无法打通。这样一来，就出现了两个来访者互相等待的情况，而且由于不会有任何进展，它们将永远等待下去，两者都无法继续自己的工作。

使用锁来解决基于内存共享模型的并发，面临着相当的挑战：

- 如果没有设计足够的锁，那么对象状态可能由于多线程的同时访问而损坏。
- 如果设计了太多的锁，由于等待将造成性能的损失，甚至导致程序无法继续的死锁。

保证数据安全是并发编程中的一大难题，而锁是一个看起来很自然的答案。即使对最优秀的工程师而言，设计出一套精妙的、不出问题的锁系统都是高难度的课题。随着项目越发复杂，锁将越来越不可靠。我们需要寻找一种新的可扩展的模型来解决数据共享的问题。Actor 模型看起来就是可能的答案之一。

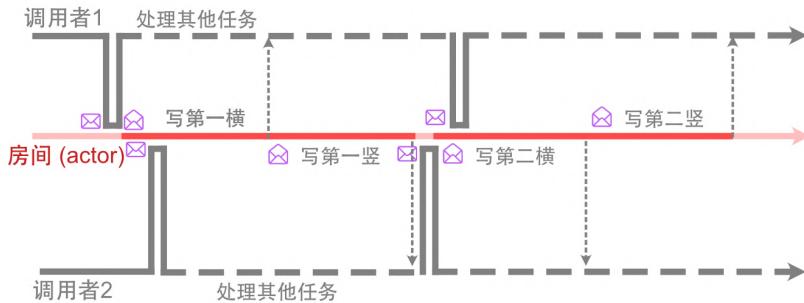
## Actor 隔离

### 具象化的 actor

还是这个房间，我们换一换思路。现在把所有的安全设施都撤掉，只在它的门口设置一个信箱，并在房间内配置一个专员。来访者不再被允许亲自进入房间进行操作，他们只能携带一封信，并将这封信投递到信箱里，表明自己到访过。房间里的专员会负责检查信箱，每次拿出一封信进行处理。在获取信后，专员在房间里的“正”字纸上添加一笔，然后把结果封好作为回信寄回给来访者的地址。这样一来我们“轻易”地解决了上面的问题：

- 因为只有一个操作专员，且他每次只处理一封信，所以同一时间只会有一人在纸上写字。内存状态不会遭到破坏。
- 因为来访者现在只需要进行信件投递，这不需要任何等待。投递完成后来访者(调用线程)就可以去做其他事情了，直到房间回信到达才需要回头处理结果。对锁的完全去除，也从根本上消除了死锁的可能性。

这样一个房间的模型，我们就把它称作 actor。



Actor 能够运作的关键在于，“信件投递”这件事情是线程安全的。我们会在后面关于 Swift 并发模型的章节，再说明 Swift 是如何确保这个前提的。在此之前，你可以把信件投递想象为发生在一个特殊的串行队列中，先假设信件投递是高效且安全的操作。

## Swift 中的 actor 和隔离检查

在 Swift 中，可以使用 actor 类型来对这个模型进行抽象：

```
actor Room {  
    let roomNumber = "101"  
    var visitorCount: Int = 0  
  
    init() {}  
  
    func visit() -> Int {  
        visitorCount += 1  
        return visitorCount  
    }  
}
```

actor 类型和 class 类型在结构上十分相似，它可以拥有初始化方法、普通的方法、属性以及下标。它们能够被扩展并满足协议，可以支持泛型等。和 class 的主要不同在于，actor 将保护其中的内部状态（或者说存储属性），让自身免受数据竞争带来的困扰。这种保护通过 Swift 编译

器的一系列限制来达成，这主要包括对 actor 中成员（包括实例方法和属性）的访问限制。在 actor 中，对属性的直接访问只允许发生在 self 里，像是上例中的 visit，可以直接操作 visitorCount 并返回它：

```
func visit() → Int {  
    visitorCount += 1  
    return visitorCount  
}
```

但是，如果我们想要在 Room 外进行这些操作，编译器会给出错误：

```
class Op {  
    func foo() {  
        let room = Room()  
        room.visitorCount += 1  
        print(room.visitorCount)  
    }  
}  
  
// Error:  
// Actor-isolated property 'visitorCount' can not be  
// mutated (referenced) from a non-isolated context
```

从外部直接操作和访问内部状态 visitorCount 的行为是被限制的，我们把这种限制称作 actor 隔离：Room 的成员被隔离在了 actor 自身所定义的隔离域（actor isolated scope）中。

我们甚至不能直接调用 visit 方法，它也是在隔离域中的：

```
func foo() {  
    let room = Room()  
    room.visit()  
}  
  
// Actor-isolated instance method 'visit()' can not
```

```
// be referenced from a non-isolated context
```

上面这些代码，如果在 Room 是 class 的情况下，都是被允许的。但是在 class 中它们并不安全，如果不加锁，任何线程都可以任意访问它们，这会面临数据竞争的风险。和 class 不同，在 actor 实例上所有的声明，包括那些存储和计算属性（比如 visitorCount）、实例方法（比如 visit()）、实例的下标等，默认情况下都是 actor 隔离的。隔离域对于自身来说是透明的：被同一个域隔离的成员可以自由地互相访问，比如 visit() 中可以自由操作 visitorCount。

从 actor 外部持有对这个 actor 的引用，并对某个具有 actor 隔离特性的声明的访问，叫做跨 actor 调用。这种调用只有在异步时可以使用：

```
func bar() async {
    let room = Room()
    let visitCount = await room.visit()
    print(visitCount)
    print(await room.visitorCount)
}
```

具体来说，像是 visit() 和 visitorCount 这样的异步访问将被转换为消息，来请求 actor 在安全的时候执行对应的任务。这些消息就是投递到 actor 信箱中的“信件”，调用者开始一个异步函数调用，直到 actor 处理完信箱中的对应的消息之前，这个调用都会被置于暂停状态。而在此期间，负责的线程可以去处理其他任务。

虽然 Room.visit 并没有被标记为 async 函数，但是编译期间 Swift 会对 actor Room 进行隔离检查，它会决定哪些调用是跨 actor 隔离域的调用。因为 actor 要保证隔离状态不被意外改变，因此对于这些调用，必须等待到合适的时间才能处理。编译器会应用上面的规则，要求调用方引入潜在暂停点 await。Swift 中 actor 模型的特点，要求了对隔离域上的调用，必须发生在异步任务执行上下文中。

要注意，actor 隔离域是按照 actor 实例进行隔离的：也就是说，不同的 Room 实例拥有不同的隔离域。如果要进行消息的“转发”，我们必须明确使用 await：

```
actor Room {
    func forwardVisit(_ anotherRoom: Room) async → Int {
```

```
    await anotherRoom.visit()  
}  
}
```

在底层，每一个 actor 对信箱中的消息处理是顺序进行的，这确保了在 actor 隔离的代码中，不会有两个同时运行的任务。也就确保了 actor 隔离的状态，不存在数据竞争。从实现角度来看：消息是一个异步调用的局部任务，每个 actor 实例都包含了它自己的串行执行器，这个执行器实际对作用域进行了限制。串行执行器负责在某个线程内循序执行这些局部任务（包括处理消息，实际访问实例上的状态等）。从概念上，这和串行派发的 DispatchQueue 类似，但是 actor 在实际运行时，是基于协作式的线程派发和 Swift 异步函数续体的，相比于传统的线程调度，它是一个更轻量级的实现。

关于 Swift 异步具体的执行模型，以及执行器的有关讨论，我们会放在后面的章节再展开。

对于 Swift 中的 actor 模型，最重要的就是理解隔离域，并记住两个结论：

- 某个隔离域中的声明，可以无缝访问相同隔离域中的其他成员；
- 某个隔离域外的声明，不论它位于传统的非隔离中，还是位于其他 actor 的隔离域中，都无法直接访问这个隔离域的成员。只有通过异步消息的方法，才能跨越隔离域进行访问。

## Actor 协议

所有的 actor 类型都隐式地遵守 Actor 协议，它的定义是：

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

关于 Sendable 和 Executor，我们会在后面再次提到。一般来说，我们不会需要去手动让 class 实现 Actor 协议。如果有需要，我们直接声明 actor 类型就可以了，编译器将在 actor 的初始化方法中“注入”创建执行器的调用，为 actor 绑定一个串行的执行器。

## 隔离声明

actor 类型默认的声明都是被隔离在 actor 域中的。这带来一个很现实的问题，在让这个 actor 类型满足某个一般性质的协议时，会有一些困难。比如，我们如果有如下的 Popular 协议，来判断这个房间是不是被访问了多次：

```
protocol Popular {
    var popular: Bool { get }
}
```

使用 actor 类型定义的隔离域是一个非常强的假设：对于某个 actor 实例所形成的隔离域，任何一个函数声明，要么在隔离域中，要么在隔离域外。Popular 中定义的 var popular 不在任何 actor 隔离域中，它是一个普通的同步协议方法。当我们尝试像普通 class 那样去让 Room 实现 Popular 时，会遇到编译错误：

```
// 错误代码
extension Room: Popular {
    var popular: Bool {
        visitorCount > 10
    }
}

// Actor-isolated property 'popular' cannot be used to
// satisfy a protocol requirement
```

这里的 popular 是定义在 actor Room 中的，它是在 actor 隔离域中的声明。Room.popular 和 Popular.popular 产生了隔离域上的冲突，必须有一方进行妥协。

## Actor 协议细分

第一种方式是让 Popular 也能在某个隔离域中。这里可以让 PopularActor 作为 Actor 协议的细分存在：

```
protocol PopularActor: Actor {
```

```
var popularActor: Bool { get }  
}
```

这样，在当 Room 实现 PopularActor 时，其中的 popularActor 也将作为隔离域中的一部分存在，于是 Room 将可以在同一个隔离域中访问到 visitorCount：

```
extension Room: PopularActor {  
    var popularActor: Bool {  
        visitorCount > 10  
    }  
}
```

当然，因为 popularActor 现在是 actor 的一部分了，从隔离域外对它的访问，都需要经过 await 进行。这一点和 actor 上的其他成员的默认行为是一致的。PopularActor 现在是 Actor 的细分协议，因此也只有 actor 类型能满足这个协议了。

## 定义异步协议方法

让 Popular 协议对 actor 适用的第二种方法，是将涉及到的成员设计为异步方法或属性；也就是说，让它在语法上明确满足“可暂停”的特点：

```
protocol PopularAsync {  
    var popularAsync: Bool { get async }  
}  
  
extension Room: PopularAsync {  
    var popularAsync: Bool {  
        get async {  
            visitorCount > 10  
        }  
    }  
}
```

这样的实现没有影响 Room.popularAsync 处于 actor 隔离域中的事实。因此，在隔离域外我们可以用类似于访问其他成员的方式，通过 await 的方式访问到 popularAsync（实际上 get async 也要求我们使用 await）：

```
let room = Room()  
print("Is popular? \$(await room.popularAsync)")
```

也可以把 await 写到 print 语句外面：

```
await print("Is popular? \$(room.popularAsync)")
```

在前面章节我们已经介绍过，await 纯粹就是一个编译期间的标记，它的作用是辅助开发者，提示我们这里代码可能发生暂停。选择将 await 写在整个表达式的开头，还是选择让它紧接实际可能暂停的代码，只要风格统一，都是可行的。不过，让 await 的位置尽可能靠近实际可暂停的表达式，可能会让代码的意思更加清楚。

虽然 popularAsync 的声明是处于 Room 的 actor 隔离域内的，但是它本身是一个异步 getter，域内的其他方法要访问它时，依然需要 await。这有时候很不方便，而且具有传染性：当某个域内方法本身是同步方法时，是不允许调用这个异步 getter 的：

```
actor Room {  
    // ...  
  
    func reportPopular() {  
        if popularAsync {  
            print("Popular")  
        }  
    }  
}  
  
// 'async' property access in a function that does not  
// support concurrency
```

为了避免重复逻辑，同时保持 popularAsync 以满足异步协议，也许我们可以添加一个内部使用的同步的 getter，然后在 popularAsync 中把它简单地转换为异步：

```
extension Room: PopularAsync {
    private var internalPopular: Bool {
        visitorCount > 10
    }

    var popularAsync: Bool {
        get async {
            internalPopular
        }
    }
}
```

这样，刚才的 reportPopular 就可以直接使用隔离域内的同步方法了：

```
actor Room {
    // ...

    func reportPopular() {
        if internalPopular {
            print("Popular")
        }
    }
}
```

如果我们需要一个协议既能被 class 或 struct 这样的“传统”类型满足，又能以安全的方式工作在 actor 里，可以考虑将协议的成员声明为上面这样的异步成员。因为同步函数其实是异步函数的子集和“特例”，所以普通类型是可以用同步函数来实现这个协议的异步定义的：

```
class RoomClass: PopularAsync {
    var popularAsync: Bool { return true }
}
```

```
let room = RoomClass()
print(room.popularAsync)
```

不过，当然了，如果我们要使用 PopularAsync 作为实例类型（或者类型约束）的话，由于类型信息不足以判断 popularAsync 的具体实现是否是同步，我们必须加上 await 才能进行调用：

```
func foo() async {
    let room: PopularAsync = RoomClass()
    print(await room.popularAsync)
}

func bar<T: PopularAsync>(value: T) async {
    print(await value.popularAsync)
}
```

## nonisolated

在上面 PopularActor 和 PopularAsync 的例子中，我们都更改了协议本身的定义。但是当这个协议是外部定义的或者早已存在于现有同步系统中的话，改变协议本身是很困难、甚至不可能的事情。比如，如果我们想为 Room 加上一段描述，想办法让它满足 Swift 的 CustomStringConvertible：

```
public protocol CustomStringConvertible {
    var description: String { get }
}
```

这是一个同步协议，不可能纳入到 actor 隔离域内。想要在 Room 中满足这样一个协议，唯一的方法是明确将 Room.description 声明放到隔离域外，使用 nonisolated 标记可以让编译器做到这一点：

```
extension Room: CustomStringConvertible {
    nonisolated var description: String {
        "A room"
    }
}
```

```
    }  
}
```

当然，可能你已经猜到了，隔离域外的成员 `description` 是不能同步访问隔离域内的内容的。比如下面的代码会给出编译错误：

```
extension Room: CustomStringConvertible {  
    nonisolated var description: String {  
        "Room Visited: \(visitorCount)"  
    }  
}  
  
// Actor-isolated property 'visitorCount' can not be  
// referenced from a non-isolated context
```

不过，`Room` 中用 `let` 声明的存储变量，是一个例外。这类 `let` 成员的值不会在并发模型中改变，因此它们自然是线程安全的。在同一个模块内，从域外访问这样的值是透明的：

```
actor Room {  
    let roomNumber = "101"  
    // ...  
}  
  
extension Room: CustomStringConvertible {  
    nonisolated var description: String {  
        "Room Number: \(roomNumber)"  
    }  
}
```

需要强调的是，这个“安全例外”只发生在同一模块内。从别的模块访问 `let` 定义的 `roomNumber` 时，依然需要加上 `await`。这样的安排是刻意为之的：根据版本原则，将 `public let` 替换为 `public var`，应该是一个仅添加了特性 (setter)，可以后向兼容的变化，它不应该引起原有使用者的编译错误。但是，如果我们将 `public let` 也作为例外，让模块外的代码可以直接使用的话，在未来我们将它换为 `public var` 时，原有的域外代码将会失效 (此时需要

await 才能跨域访问)。因此，一开始就由编译器作出规定，必须使用 await 来进行跨模块跨域访问，是更合理的选择。

nonisolated 标记的成员，无法访问那些隔离域内的成员，否则将违反基本的并发安全假设，让 actor 类型变得不安全。

另外，actor 中的存储属性的成员安全保证，只针对具体的值和引用。而对于那些被引用的实际对象，如果它们的类型不是 actor，而是普通的 class 的话，在域外对这些对象上成员的访问依然是不安全的。举例来说，如果我们有一个 class Person 类型，来代表房间中的某个人：

```
class Person: CustomStringConvertible {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }

    var description: String {
        "Name: \(name), age: \(age)"
    }
}
```

我们可能会在 Room 中持有一个 Person 对象：

```
actor Room {
    private let person: Person
    // ...

    init() {
        person = Person(name: "Lee", age: 18)
    }
}
```

在隔离域内，我们可以改变这个人的名字：

```
extension Room {  
    func changePersonName() {  
        person.name = "Bruce"  
        print("Person: \(person)")  
    }  
}
```

这里，可以确信 `changePersonName` 的执行是在域内，它是安全的，因此 `person.name` 的修改在此时也是安全的。但是，我们也可以在 `Room` 的一个 `nonisolated` 域外方法中修改这个属性，编译器并没有阻止我们这么做：

```
extension Room {  
    nonisolated func unsafeChangePersonName() {  
        person.name = "Bruce"  
        print("Person: \(person)")  
    }  
}
```

由于 `unsafeChangePersonName` 在隔离域外，它可能在多个线程中以并行的方式被调用，此时对 `name` 的修改将造成内存的数据竞争。因此这段代码是不安全的。

当然，这并没有违反 actor 关于“保护成员”的目标：因为 `person` 这个引用确实是完全受到保护的，问题在于 `Person` 类型没有能够保护它的 `name` 成员，这是由于 `Person` 是 `class` 这一特质所造成的，和 `Room` 是 `actor` 这一事实并无关系。想要增加安全性，我们可以选择把 `Person` 也声明为 `actor`，或者让它满足一个稍微弱化的假设，让 `Person` 满足 `Sendable`。在未来，编译器会添加这类问题的静态检查，并彻底防止此类数据竞争的问题。

到现在为止，我们已经在不少地方看到 `Sendable` 了。但是请允许我把这个话题再留到后面一些，因为我们关于隔离域声明的探索还没有结束。

## isolated

到目前为止，我们已经掌握了下面几个事实：

- 在 actor 中的声明，默认处于 actor 的隔离域中。
- 在 actor 中的声明，如果加上 nonisolated，那么它将被置于隔离域外。
- 在 actor 外的声明，默认处于 actor 隔离域外。

现在拼图还有最后一环：对于 actor 外的声明，我们有没有办法让它处于某个隔离域中呢？

答案是肯定的，我们可以使用 isolated 关键字来修饰函数的某个 actor 类型的参数，这会明确表示函数体应该运行在该 actor 的隔离域中。通常在一些需要隔离的全局的函数中，可以见到这样的用法：

```
func reportRoom(room: isolated Room) {  
    print("Room: \(room.visitorCount)")  
}
```

在函数参数的类型前，加上 isolated，将把这个函数放到该参数 actor (这里是 room) 的隔离域中。这样，在函数体内部调用隔离域内的成员，就可以使用同步的方式了。

根据调用者和参数的不同，在调用这个全局函数时，编译器会要求我们添加 await。规则和一般对 actor 的成员调用完全一致：当从隔离域内部使用时，可以以同步方式直接访问；但当从隔离域外使用时，则需要 await：

```
actor Room {  
    func doSomething() async {  
        // `self` 在自身的隔离域中  
        reportRoom(room: self)  
  
        let room = Room()  
        // room 不在 `self` 隔离域中。需要切换隔离域。  
        await reportRoom(room: room)  
    }  
}
```

## 隔离域切换

在上面的例子中，提到了隔离域切换的概念，使用一个更“正规”一些的名称，我们把它叫做 **actor 跳跃 (actor hopping)**。对于隔离域中的成员，比如方法调用，actor 跳跃是隐式发生的，编译器在生成最终代码时，会在需要的位置插入 actor 跳跃的指令：典型的地方是方法开头时跳到 `self`，以及显式 `await` 调用其他 actor 成员时跳到对应的 actor。比如，上例中 `doSomething` 在编译后，等效于：

```
actor Room {  
    func doSomething() async {  
        hop_to_executor(self)  
  
        // `self` 在自身的隔离域中  
        reportRoom(room: self)  
  
        let room = Room()  
  
        // room 不在 `self` 隔离域中。需要切换隔离域。  
        hop_to_executor(room)  
        reportRoom(room: room)  
  
        // `room` 隔离域执行完毕，跳回 `self`  
        hop_to_executor(self)  
    }  
}
```

actor 跳跃是轻量级的操作，大部分情况下它们会在同一个线程中完成，但我们不应该对实际的执行线程进行假设。通过理解 actor 跳跃，以及异步函数和任务执行时的堆栈情况，我们可以对 Swift 并发程序的性能有基本的判断。我们会在本书最后的章节讨论这个话题。

## 隔离域冲突

通过 `isolated` 设定函数的隔离域，天然地面临着一个问题：那就是在设定的多个 `isolated` 参数时，会发生什么。比如说，某个全局函数接受两个不同的 `isolated` actor：

```
// 危险代码

func addCount(room1: isolated Room, room2: isolated Room) → Int {
    let count = room1.visitorCount + room2.visitorCount
    return count
}
```

在本书写作时，上面的代码是可以正常编译和运行的。那么，这时的隔离域应该是 room1 呢，还是 room2 呢，又或者是其他某种行为？

如果我们在 Room 中为 room1 和 room2 都传入 self 的话，隔离域的表述是清晰的，就是 self 自身的隔离域：

```
extension Room {
    func addSelf() {
        _ = addCount(room1: self, room2: self)
    }
}
```

如果同时传入 self 和其他某个新的 Room 对象：

```
extension Room {
    func add(_ another: Room) async {
        print(visitorCount)
        _ = await addCount(room1: self, room2: another)
    }
}
```

这种情况下，当前编译器会选择使用第一个非 self 的 actor 隔离域作为调用时的隔离域。上面的代码，在初步编译后等效于：

```
extension Room {
    func add(_ another: Room) async {
        hop_to_executor(self)
        print(visitorCount)
```

```
    hop_to_executor(another)
    _ = addCount(room1: self, room2: another)
    hop_to_executor(self)
}
}
```

这带来一个问题：在由 `isolated` 标记参数的 `addCount` 中，运行环境的隔离域是 `another actor` 值。在其中以同步的方式访问第一个参数 `self` 上的成员，是不安全的行为。

这种行为理论上应该被编译器禁止，但是 Swift 并发路线图 中指出，`actor` 的完全隔离将作为第二阶段的目标，而 Swift 5.5 中的第一阶段，只提供部分的 `actor` 隔离。我们在上一节里谈到的不安全的 `unsafeChangePersonName`，也属于暂时没有确保安全的操作。

虽然在静态上不安全，但是 Swift 并发底层所依赖的 GCD 的新实现，可以通过合理的调度，让 `self` 隔离域和 `another` 隔离域处在同一个并发域中“交替”执行，以此避免内存问题。不过，这并没有文档说明，也不是一个很强的保证，我们最好不要依赖这个实现细节来进行假设。

和上面的使用两个 `isolated` 隔离参数造成冲突类似，如果我们在某个 `actor` 中声明了接受 `isolated` 参数的方法，即使它只接受一个这种参数，其隔离域依然存在冲突：方法本身在 `actor` 中，是按照 `self` 进行 `actor` 隔离的，同时它又被声明了参数隔离，我们将无法确定最终这个函数的隔离状态：

```
// 危险代码
extension Room {
    func baz(_ another: isolated Room) {
        print(self.visitorCount)
        print(another.visitorCount)
        // ...
    }
}
```

要解决这个隔离上的模糊，我们可以选择去除 isolated，这样就能明确地选择使用 self 隔离；或者为方法添加 nonisolated 来去掉 self 的隔离，从而选择 another 的隔离域。在跨越隔离域访问成员时，按照规则使用 await，会是更加明确的做法：

```
extension Room {  
    // 使用 self 隔离  
    func baz1(_ another: Room) async {  
        print(self.visitorCount)  
        print(await another.visitorCount)  
        // ...  
    }  
  
    // 使用 another 隔离  
    nonisolated func baz2(_ another: isolated Room) async {  
        print(await self.visitorCount)  
        print(another.visitorCount)  
        // ...  
    }  
}
```

乍看起来，为 Room.baz2 添加 nonisolated，并指定 isolated 的方式似乎很蠢，但是它确实有实际的使用途径。考虑下面这种情况，我们在方法内需要多次访问 another actor 隔离域中的成员：

```
extension Room {  
    func baz(_ another: Room) async {  
        for _ in 0 ..< 100 {  
            print(await another.visitorCount)  
            _ await another.visit()  
            print(await another.visitorCount)  
        }  
    }  
}
```

在默认的 self 隔离中，每次对 another 上成员的访问都会产生两次 actor 跳跃：从 self 域切换到 another 域，然后在 await 结束后再切回 self。这种情况下，使用 nonisolated，可以减少 actor 跳越的次数，在某些特定情况下对性能提升会有一定帮助。

总而言之，尽可能避免隔离域的冲突，让一个成员拥有单一而明确的隔离域，往往可以帮助我们避开很多潜在的问题。

## 小结

本章开头提到的并发编程内存共享模型的困境，几乎是所有支持多线程的编程语言都会遇到的问题，Swift 也不例外。Swift 的 class 提供了一种定义可变状态和在程序中共享这些状态的机制。不过，在多线程中使用 class 通常需要依靠加锁这样的手动同步来避免数据竞争，这恰恰是最容易出错的地方。

actor 模型提供了一种能力，在共享状态的同时，提供静态的数据竞争的检测，它完全避免一整类常见的并发 bug。将私有的串行执行器封装到 actor 中，以此实现内部状态的保护。隔离域外部使用 actor 时，则通过消息发送的方式，让 actor 有能力确保同一时间对内部状态的访问是独占的。

结合异步函数的语法，编译器可以在合适的地方提醒我们会发生跨越 actor 隔离域的访问。这些访问在可能会在不同 actor 之间跳跃，它们基于的底层调度会确保数据安全。以此构建的并发系统，相比于传统的可能由于人为疏失而失效的锁同步机制，显然是更加稳定的。

同时为了保证能灵活处理，Swift 引入了一些可以改变隔离域的关键字，比如 nonisolated 和 isolated。善用这些关键字会让代码更加合理、简洁。但同时要记住，这些关键字其实是为高阶开发者准备的工具，滥用它们也将为数据安全带来潜在危害。真正理解这些关键字背后所做的事情，在处理 actor 的问题时，时常提醒自己检查隔离域是否正确（当然，编译器会帮助我们处理掉绝大部分内容），是正确使用 actor 隔离域的不二选择。

actor 模型除了保护状态以外，还可能衍生出更多的扩展应用。相比于锁，actor 模型更容易扩展到分布式场景下：一个 actor 拥有自包含的隔离状态，这个特性让 actor 甚至能运行在复数个物理主机组成的集群上。只要 actor 之间互相知晓信箱地址，它们就能够发送消息完成通讯，同时保证不发生数据竞争。而这些激动人心，可能会改变未来编程方式的特性，也已经开始准备陆续加入到 Swift 中了。

业界已经有很多案例证明了 actor 模型的能力，相比起来，当前 Swift 并发中对 actor 的引入还非常初步，就算是 actor 隔离，也还有不少需要完善的地方。我们有理由期待在未来版本的 Swift 中，actor 获取一些更加强大的特性。

全局 actor, 可重  
入和 Sendable

9

上一章中我们已经看到了 actor 在数据隔离和避免竞争时的强大特性。本章我们会补充几个有关 actor 的话题，它们包括全局 actor，actor 的可重入特性，以及 Sendable 协议。

# 全局 actor

actor 类型作为局部的数据隔离手段，是非常有效的：编译器可以保证定义在 actor 上的成员的安全。在上一章最后，我们也介绍了使用 `isolated` 把函数的隔离域设定为某个参数隔离域的方式，来让 actor 隔离域在一定程度上得到扩展。如果需要保护的状态存在于 actor 外部，或者这些代码不可能汇集到一个单一的 actor 实例中时，我们可能会需要一种作用域更加宽广的隔离手段。

为了解决这个问题，我们可以声明并使用全局 actor。作为属性包装，它可以被任意地使用在某个属性或方法前，对这个属性或方法进行标注，把它限定在该全局 actor 的隔离域中。Swift 标准库中的 `MainActor` 就是这样一个全局 actor。我们先来看看这个特殊 actor 的行为，然后会说明我们在什么情况下可能会需要自定义一个全局 actor。

## MainActor

### 主线程队列派发

需要在大范围内，对零散代码进行隔离域限制的一个典型的例子，是关于 UI 相关代码的隔离。UIKit 或 AppKit 中，对 UI 的操作必须在主线程上进行；一些重要的由框架调用的方法（比如 `viewDidLoad` 等），也会被保证在主线程上运行。假设代码在某个后台线程中获取了数据（这在使用 `URLSession` 进行网络请求的时候是很常见的情况），我们想用这些数据设置 UI 的话，传统 GCD 中，会需要使用 `DispatchQueue.main` 将操作派发到 UI 队列，并由 UI 队列把工作分配给它所绑定的主线程，来进行实际的工作：

```
let url = URL(string: "https://example.com")!
let task = URLSession.shared.dataTask(with: url) {
    data, response, error in
    print(Thread.current.isMainThread) // false

    DispatchQueue.main.async {
```

```
    print(Thread.current.isMainThread) // true
    self.updateUI(data)
}
}

task.resume()
```

这个方法在 Apple 平台的 app 开发中可以称得上是被广泛使用的，以至于很多时候开发者会“无脑”在任何时候都使用这个方式，或者甚至将它滥用，作为“黑魔法”来解决一些比如布局等 UI 时序上的问题。将操作派发到 DispatchQueue.main 上，特别是如果原来就在主线程上，但依然进行这样的派发的话，往往会改变原本应有的执行顺序，为今后埋下巨大的隐患。在使用 DispatchQueue.main 时，有一种做法是先判断当前是否是主线程，如果是的话，则直接执行需要的操作；如果当前不是主线程，那么再进行派发。这样，我们就可以避免不必要的派发，并在一定程度上减轻派发对主线程操作执行顺序的改变，就像下面这样：

```
extension DispatchQueue {
    static func mainAsyncOrExecute(
        _ work: @escaping () -> Void)
    {
        if Thread.current.isMainThread {
            work()
        } else {
            main.async { work() }
        }
    }
}
```

这个模式其实和 actor 高度相似：当方法在 actor 隔离域时，我们就可以用同步方式直接访问 actor 成员，在隔离域外时，则需要异步访问。我们完全可以把主线程看作是一个特殊的 actor 隔离域：这个隔离域绑定的执行线程就是主线程，任何来自其他隔离域的调用，需要通过 await 来进行 actor 跳跃。在 Swift 中，这个特殊的 actor 就是 MainActor。

## MainActor 隔离域

MainActor 是标准库中定义的一个特殊 actor 类型：

```
@globalActor final public actor MainActor : GlobalActor {  
    public static let shared: MainActor  
    // ...  
}
```

整个程序只有一个主线程，因此 MainActor 类型也只应该提供唯一一个对应主线程的 actor 隔离域。它通过 shared 来提供一个全局实例，以满足这个要求。所有被限制在 MainActor 隔离域中的代码，事实上都被隔离在 MainActor.shared 的 actor 隔离域中。

## @MainActor 属性包装

如果我们看 GlobalActor，会发现这个 shared 成员正是 GlobalActor 协议的要求。通过满足 GlobalActor 并且在类型前添加 @globalActor 标记，我们就可以将这个 MainActor 类型作为属性包装 (property wrapper)，用来注记其他的类型或者方法：

```
@MainActor class C1 {  
    func method() {}  
}  
  
class C2 {  
    @MainActor var value: Int?  
    @MainActor func method() {}  
    func nonisolatedMethod() {}  
}  
  
@MainActor var globalValue: String = ""
```

按照添加的地方，@MainActor 有不同的作用。对于 C1，整个类型都被标记为 @MainActor：这意味着其中所有的方法和属性都会被限定在 MainActor 规定的隔离域中。而 C2 的话，只有部分方法被标记为 @MainActor。另外，对于定义在全局范围的变量或者函数，也可以用 @MainActor 限定它的作用返回，上面代码中的 globalValue 就是一个例子。在使用它们时，需要切换到 MainActor 隔离域。和其他一般的 actor 一样，可以通过 await 来完成这个 actor 跳跃；也可以通过将 Task 闭包标记为 @MainActor 来将整个闭包“切换”到与 C1 同样的隔离域，这样就可以使用同步的方式访问 C1 的成员了：

```
class Sample {  
    func foo() {  
        Task { await C1().method() }  
        Task { @MainActor in C1().method() }  
        Task { @MainActor in globalValue = "Hello" }  
    }  
  
    func bar() async {  
        await C1().method()  
    }  
}
```

我们提到过，每个 actor 实际都有一个串行的执行器，来保证同一时间对成员访问是唯一的。 MainActor 的执行器在内部所做的事情，其实就是调用 DispatchQueue.main 的 `async`，在需要的时候把操作派发到主队列中。事实上，这样的派发所接受的闭包，在 Swift 并发中也是被隐式标记为 `@MainActor` 的，所以下面的代码，虽然看起来 `foo` 在 `MainActor` 的隔离域外，但在闭包中对隔离域内的成员可以同步访问：

```
class Sample {  
    func foo() {  
        // ...  
        DispatchQueue.main.async {  
            C1().method()  
            globalValue = "World"  
        }  
    }  
}
```

不过，笔者不是很建议把原有的 GCD 的 API (比如 `DispatchQueue.main.async`) 和新的 `Task` 及 `await` 的方式进行混用。这种混用在 `MainActor` 这个特例中执行时不会有太大问题，但是却引入了不同的并发风格，可能造成理解上的困难。另外，在 Swift 并发中，其实线程和通过 GCD 进行线程调度的概念，更多时候是被隐藏起来的。如果没有绝对的必要，我们最好让并发的底层机制为我们进行调度，以保证并发性能。我们在下一章会看到关于这些底层机制的更多内容。

## UIKit 中的 @MainActor

一个很有趣，看起来也比较激进的事实，是 UIKit 中一些非常常用的类，都被 `@MainActor` 修饰了。比如 `UIViewController` 或者 `UIView`，甚至是它们的所有子类，如果你查看它们的文档，会发现这样的声明：

```
@MainActor class UIViewController : UIResponder  
@MainActor class UIView : UIResponder  
@MainActor class UIButton : UIControl
```

在 Swift 并发的时代之前，虽然在文档中规定了这些 UI 相关的类型比如在主线程上操作，但这些规则并没有编译器保证，只能在运行时通过调试手段（比如打开 Main Thread Checker）检测到。`@MainActor` 的出现，将这些类型上的成员明确地圈入隔离域中。从 `UIViewController` 自身内部的调用，可以直接使用同步方式。在以前，切换到后台队列的线程去执行某项任务（比如加载网络资源），然后再切回主线程设置 UI，是非常常见的做法。这类操作现在用 `Task.init` 书写的话，可以更简单：

```
class ViewController: UIViewController {  
    override func viewDidLoad() {  
        let url = URL(string: "https://example.com")!  
        Task {  
            let (data, _) = try await URLSession.shared.data(from: url)  
            self.updateUI(data)  
        }  
    }  
  
    private func updateUI(_ data: Data?) {  
        // ...  
    }  
}
```

上例中，继承自 `UIViewController` 的 `ViewController` 在 `@MainActor` 域中，因此 `viewDidLoad` 的运行环境也在同一隔离域里。通过 `Task` 新建的任务，将继承 `actor` 的运行环境，也就是说，它的闭包也运行在 `MainActor` 隔离域中的，这也是可以同步调用 `updateUI` 的。

原因。如果我们将这里的 Task.init 换为 Task.detached 的话，闭包的运行将无视原有隔离域。此时，想要调用 updateUI，我们需要添加 await 以确保 actor 跳跃能够发生：

```
Task.detached {  
    let (data, _) = try await URLSession.shared.data(from: url)  
  
    // 编译错误，需要明确使用 await  
    // self.updateUI(data)  
    await self.updateUI(data)  
}
```

所以，在 UIViewController 的环境中，如果我们希望开始某个异步操作，然后在主线程中调用自身成员的话，Task.init 一般会是更好的选择：它能保证 await 后，后续代码能通过 actor 跳跃回到 MainActor 中执行。

对于 UIViewController 外发生的调用，在 Swift 并发上下文中 (比如 async 函数或者某个一般 actor 内)，由于无法确定调用域，因此必须使用 await 的方式进行 actor 跳跃，来保证它们运行在主线程上：

```
class Sample {  
    func bar() async {  
        let button = UIButton()  
        // 注意，`Sample` 不在 `@MainActor` 中  
        // 缺少 await 的话，将报错  
        await button.setTitle("Click", for: .normal)  
        await ViewController().view.addSubview(button)  
    }  
}
```

严格来说，这对旧的代码会是一个破坏性的变化：在引入 Swift 并发之前，ViewController 之外 (或者说是在 MainActor 隔离域外) 的代码，如果操作和使用了 ViewController 的话，由于无法确定原来的隔离域，理论上都应该编译报错。但是事实上，编译器却“容忍”了这类问题。比如下面的代码可能会在任意隔离域，甚至是非隔离域中被调用，但它并不会产生错误：

```
class Sample {  
    func foo() { // 注意，和 `bar` 不同，foo 不是异步函数  
        let button = UIButton()  
        // 没有 await 也不报错  
        button.setTitle("Click", for: .normal)  
        ViewController().view.addSubview(button)  
    }  
}
```

如果编译器对所有像是 UIViewController 或者 UIView 这些类型进行严格的隔离域检查，那么可以想见已有的代码将几乎不可能完成隔离域的迁移。因此，Swift 现在选择了只在明确的存在任务上下文的环境中，对 @MainActor 进行隔离域检查：也就是说，检查只发生在异步方法里、actor 中、以及 Task 相关 API 等地方。对于引入 Swift 并发之前就存在的纯同步的代码，这个检查被关闭了。

在原有的同步代码中忽略掉 @MainActor，是一种工程上的妥协。在编译器内部，这依靠 @\_unsafeMainActor 内部标注完成。通过逐步递进的方式提供可行的迁移方案，一直是 Swift 并发路线图上重要的目标之一。在未来我们也许会看到更加严格和绝对安全的 UI 代码，但是那并不是一个近期目标。

不过如果我们明确地将某些成员标记为 @MainActor 的话，可以用这个“本地”的声明覆盖编译器的针对全体的默认行为。在为已有项目进行迁移时，我们也可以利用这一点重新“激活”编译器的检查，让我们得到更加安全的代码：

```
class ViewController: UIViewController {  
    // ...  
  
    @MainActor func explicitUIMethod() {  
  
    }  
}  
  
class Sample {
```

```
func foo() {  
    // 编译错误，现在 `explicitUIMethod` 有 MainActor 的明确要求  
    // ViewController().explicitUIMethod(), 但 `Sample.foo` 没有  
    // 明确的 actor 隔离域声明。  
  
    // 让 `explicitUIMethod` 能够跳跃到 MainActor 隔离域  
    Task {  
        await ViewController().explicitUIMethod()  
    }  
}  
}
```

通过标注 `@MainActor`，Swift 编译器实际上已经具备了提供完全的主线程安全的能力。尽可能多地明确设定这类注解，不仅可以在当下立即增加代码的安全性；在未来某天（也许是 Swift 6），如果 Swift 编译器决定不再容忍非主线程的 UI 代码时，我们迁移起来也能轻松一些。

## 自定义全局 actor

`@MainActor` 可以帮助将散落在代码各处的需要在主线程进行的操作统一隔离起来。对于其他的隔离域，通常直接使用 `actor` 类型就能胜任。但是有一些情况，我们可能会需要和 `@MainActor` 类似的手段，来创建自己的全局隔离域。比如下面几种典型情况：

- 若干个可以成组的状态：它们散落在各个类型里，但是需要以串行的方式避免写入时的数据竞争；
- 存在需要被某个 `actor` 隔离的全局变量，此时单纯的 `actor` 类型无法做到将其纳入隔离域；
- 需要隔离的状态是跨越模块的：提供隔离域的人希望某些由其他开发者生成的状态被纳入该隔离域以保证执行时的安全。开发者在编译自己的模块时，可以向外部提供一个全局 `actor`，这样别人就能使用定义在模块内部的隔离域了。

对于这些需求，可以像 `MainActor` 那样，使用 `@globalActor` 来把一个 `actor` 类型声明为全局 `actor`，这会把它转换为一个属性包装。和 `MainActor` 类似，该类型需要提供一个单例，所有被属性包装标注的内容，都会被限制在这个 `actor` 单例的隔离域中：

```
@globalActor actor MyActor {  
    static let shared = MyActor()  
}  
  
@MyActor var foo = "some value"
```

虽然被标记为全局 actor，但 MyActor 自身也确实还是一个 actor 类型，所以这并不妨碍它被作为普通的 actor 来使用：比如持有一些成员变量或者方法时，这些 actor 内的声明会拥有自己的隔离域。有时候这会造成一些困惑。比如下面这样的代码：

```
@globalActor actor MyActor {  
    static let shared = MyActor()  
    var value: Int = 0  
}  
  
@MyActor func bar(actor: MyActor) async {  
    print(await actor.value)  
    print(await MyActor.shared.value)  
}
```

bar 方法被 @MyActor 标记，因此它运行在 MyActor.shared 的隔离域中。对于 bar 方法里作为参数的 actor，我们无法判断它是否和 MyActor.shared 属于同一隔离域，因此必须使用 await。即便访问的就是 MyActor.shared 上的属性 value，编译器现在也还无法指出它和 @MyActor 其实是在相同的隔离域中，所以这里也需要明确的 await 才能访问。为了从根本上避免这类困扰，我们通常会选择不在 @globalActor 里持有变量和实例方法，并且为它声明一个私有的初始化方法。这样，我们就能将这个 actor 类型作为纯粹的标记来使用，减少一些迷惑：

```
@globalActor actor MyActor {  
    static let shared = MyActor()  
    private init() {}  
}
```

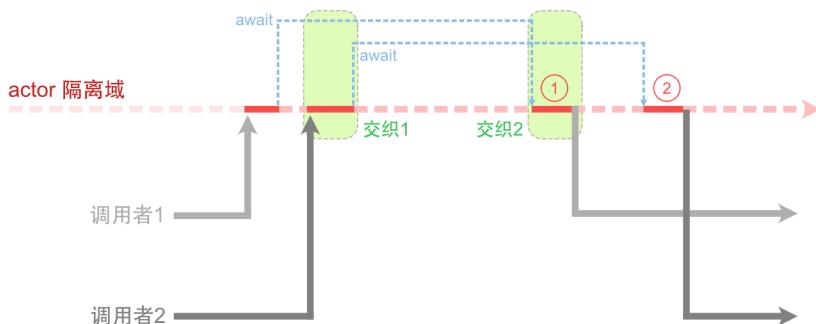
# 可重入

在 actor 的话题下，另一个很容易让人掉坑里的问题是 actor 方法的可重入特性 (reentrancy)。

## actor 中的异步方法和交织

和其他类型一样，actor 中的方法里也可以使用异步函数。在对异步函数调用时，当前的方法有可能因为要处理其他任务而处于暂停状态。依靠串行调度来保证数据安全的 actor 在这种情况下将面临一个选择：是否要在在一个方法暂停时，允许 actor 上的其他成员被访问。

Swift 的选择是允许这样的访问，我们把这个特性称为可重入：在被暂停的 actor 隔离函数继续之前，可重入特性允许其他工作在这个 actor 上执行（也包括被暂停的函数被其他人再次调用）。当 actor 上的一个函数还在等待，但另一个函数占用了隔离域并可能修改 actor 上的属性时，我们就说这些访问之间就发生了“交织”（interleaving）。



在上图中，调用者 1 通过访问 actor 的某个方法，进入到隔离域中。接下来，这个方法通过 await 调用某个异步函数，并将自己暂停，这也会放弃对隔离域的占用。在暂停期间，调用者 2 进入 actor 隔离域时，actor 依然可以进行处理，这时候就发生了交织（图中交织 1）。类似地，当调用者 2 的执行被暂停期间，调用者 1 的 await 结束并继续在 actor 隔离域上工作，此时也会产生交织（交织 2）。

actor 隔离域上的属性是可以被安全并同步地修改的：假设调用者 1 在 await 前后都依赖了某个 actor 的属性 foo，要特别注意它可能会在第一次交织期间被调用者 2 修改。这样一来，当这个 await 结束时，在 1 中（图中的交织 2 里）调用者 1 再次读取到的 foo 值，有可能和一开始的值不同。同理，经过第二次交织后，当调用者 2 回到 2（图中 actor 隔离域的最后一段访问）时，actor 的状态也可能和它一开始进入 actor 的时候有所不同。这样一来，就算是访问的是同一个变量，我们也不能轻易作出假设，认为在 await 前后它们的值会相同。

## 可重入的风险

举一个更具体的例子，比如在上一章提到过的 actor Room，它拥有一个被隔离的 visitorCount 计数器，以及一个 visit 方法来在域内递增这个计数器：

```
actor Room {  
    var visitorCount: Int = 0  
  
    func visit() → Int {  
        visitorCount += 1  
        return visitorCount  
    }  
}
```

假设我们为它添加一个方法 generateReport，在访问次数较少时，调用一个外部的异步方法（比如较慢的 AI 分析或者网络请求）分析原因，并且提出一个包含访问者数和分析原因的报告：

```
struct Report {  
    let reason: String  
    let visitorCount: Int  
}  
  
func analyze(room: Room) async → String {  
    try? await Task.sleep(nanoseconds: NSEC_PER_SEC)  
    return "Some Reason"  
}
```

```
actor Room {  
    // ...  
    var isPopular: Bool { visitorCount > 10 }  
  
    func generateReport() async → Report? {  
        if !isPopular {  
            let reason = await analyze(room: self)  
            return Report(reason: reason, visitorCount: visitorCount)  
        }  
        return nil  
    }  
}
```

在 `generateReport` 中，我们在 `await` 前访问了 `isPopular`，其中涉及到 `visitorCount`；在 `await` 之后，生成 `Report` 时，我们再次使用了 `visitorCount`。在预想情况下，我们可以生成正确的报告：

```
let room = Room()  
Task {  
    _ = await room.visit()  
    let r = await room.study()  
    print(String(describing: r))  
}  
  
// Optional(Report(reason: "Some Reason", visitorCount: 1))
```

但是，因为可重入的存在，在 `generateReport` 中 `analyze` 执行期间，其他并发代码也可以使用 `room`，比如多次调用 `visit` 进行访问：

```
let room = Room()  
Task {  
    _ = await room.visit()  
    let r = await room.study()  
    print(String(describing: r))
```

```
}

Task {
    for _ in 0 ..< 100 {
        _ = await room.visit()
    }
}

// Optional(Report(reason: "Some Reason", visitorCount: 101))
```

得到的这份报告就比较尴尬了：我们设定的 `isPopular` 条件是访问次数大于十，但生成的报告中含有 101 次访问，你很难再说这样一个房间“不受欢迎”。如果接下来关于 `Report` 的程序逻辑依赖之前我们对 `visitorCount` 所做的假设的话，将造成逻辑上的错误，而编译器对此无能为力。

要指出的是，在 `actor` 中，即使交织发生，`await` 后的实例成员状态和我们“预想”的有所不同，但是 `actor` 隔离域本身依然是有效的：两次函数调用在 `actor` 隔离域中依旧是串行执行的，它们并不会造成对某个状态的同时读写和内存危险。不过，这些交织是否发生还是取决于调用的时间，所以从外部看起来，它的部分行为会表现得和数据竞争有些相似。

在暂停点发生交织的可能性，是编译器要求在每个暂停点标记 `await` 的主要原因之一。在 `actor` 中，面对每次 `await`，我们都需要格外小心：`await` 前后的执行环境可能完全不同，实例上的状态也可能发生改变。在 `await` 前后依赖某些状态和假设时，必须时刻清晰认识到这一点。我们可以通过在 `await` 之前就复制一份需要的值，并依赖这个不变量，来解决 `generateReport` 的问题：

```
func generateReport() async → Report? {
    if !isPopular {
        let count = self.visitorCount
        let reason = await analyze(room: self)
        return Report(reason: reason, visitorCount: count)
    }
    return nil
}
```

不过，要注意只有值语义的类型适合这种做法，引用类型依然有被改变的风险。另外，根据具体的逻辑，也许我们也会有其他的解决方式，比如在 `await` 后再次检查并判断我们所依赖的状态：

```
func generateReport() async → Report? {
    if !isPopular {
        let reason = await analyze(room: self)
        return isPopular ? nil :
            Report(reason: reason, visitorCount: visitorCount)
    }
    return nil
}
```

在处理可重入时，往往需要具体问题具体分析，并没有所谓的万灵丹。我们的首要任务是认识到可重入可能发生的条件，在 `actor` 方法中看到 `await` 关键字时，我们就需要提高警惕了。

## 不可重入的设计

既然可重入和 `actor` 中的异步函数会造成执行上的交织，并带来这种编译器无法侦测的危险，那么为什么 Swift 并发依旧选择支持 `actor` 可重入呢？

诚然，如果让 `actor` 成为不可重入的实例，那么就意味着当 `actor` 在处理一条消息时（希望你还记得 `actor` 的信箱机制），它将不再查看邮箱和处理其他内容。这其实就回到了类似锁的世界中，在任务运行时，我们将 `actor` 锁上，以此避免交织和内部状态在交织中的改变。回到使用锁来避免交织的话，同时也会把锁的问题带回来，那就是大幅退化的性能以及出现死锁的风险。这两者恰恰是我们引入 `actor` 想要解决的重要问题。

有一些其他语言和框架选择在禁止可重入的同时，为 `actor` 调用加上超时检测。不过这要求所有的异步函数都可以 `throw`，而且超时检测本身也会带来性能开销，这与 Swift 并发的初衷并不相符。

## **Sendable**

关于 actor 数据安全的最后一个大话题，是关于 `Sendable` 的讨论。其实我们在之前的章节就看到过不少对 `Sendable` 的使用了，但它在 actor 中的语义相对更加明确，所以笔者决定把它放在本章中。

## 在隔离域间传递数据

和 `Hashable` 表示类型可以求取哈希值，`Equatable` 表示支持判等类似，`Sendable` 协议也表达了类型的一种能力，那就是该类型可以安全地在不同并发域之间传递。

以 actor 的执行环境为例：actor 隔离域提供了一个串行的执行环境，通过在域外使用 `await` 调用 actor 上的方法，我们可以跳跃到隔离域内。不过，这种跳跃往往伴随着一些数据的转移：比如 actor 上的方法接受某个参数，或者这些方法所返回的值，都会随着调用和返回一同跨越隔离域。考虑下面的 actor：

```
actor Room {  
    let roomName: String  
    init(roomName: String) {  
        self.roomName = roomName  
    }  
}
```

它定义了自己的隔离域，我们可以为它声明一些有用的方法，比如：

```
extension Room {  
    func visit(_ visitor: PersonStruct) → PersonStruct {  
        var result = visitor  
        result.message = "Hello, \(visitor.name). From \(roomName)."  
        return result  
    }  
}  
  
struct PersonStruct {  
    let name: String  
    var message: String = ""
```

```
init(name: String) {
    self.name = name
}
}
```

visit 接受一个 PersonStruct，并返回一个 PersonStruct。这个 PersonStruct 的所有成员（在这里是两个字符串 name 和 message）都具有值语义，因此在从隔离域外调用 Room.visit 时，它的参数 visitor 也是一个具有值语义的值。对于这样的值，在 actor 的 visit 方法内部想要对它进行修改，只能先复制一份后再进行。同样，返回的 PersonStruct 在被调用者使用时，也遵守值语义的特性。在这种情况下，不同隔离域中对 PersonStruct 的访问，实际上访问的是内存中不同的值，它们不会造成问题。因此这样做是安全的：

```
let person = PersonStruct(name: "onevcat")

for i in 0 ..< 10000 {
    Task {
        let room = Room(roomName: "room \(i)")
        let p = await room.visit(person)
        print(p.message)
    }
}

// 输出:
// Hello, onevcat. From room 0.
// Hello, onevcat. From room 1.
// ...
// Hello, onevcat. From room 9999.
```

不过，当我们使用一个具有引用语义的类型（比如 class 类型）时，情况就不一样了。在 Swift 5.5（Xcode 13）中，如果换成 PersonClass，它将高概率因为内存错误产生崩溃：

```
class PersonClass {
    let name: String
```

```
var message: String = ""

init(name: String) {
    self.name = name
}

let person = PersonClass(name: "onevcat")

for i in 0 ..< 10000 {
    Task {
        let room = Room(roomName: "room \(i)")
        let p = await room.visit(person)
        print(p.message)
    }
}
```

对于 PersonClass，在传递给不同 Room 隔离域时，它们指向的是同一个内存引用。因此，调用 visit 和 p.message 期间，存在从多个线程同时访问共享内存的风险。actor Room 虽然能保证自己的成员免于数据竞争，但是它并不能确保跨越隔离域的参数或返回值的安全。

除了 actor 以外，在使用 Task 相关的 API 创建和运行新的任务时，我们也面临着类似的问题。一些数据可能会在创建 Task 时从非任务的运行环境传递到任务运行环境中，同时它也可能在其他并发运行的环境中被访问：

```
class Sample {

    var value: String = ""

    func foo() {
        Task { value += "hello" }
        Task { value += "world" }
        // sometime later
        print(value)
    }
}
```

为了保证数据安全，我们需要一种方法来对此进行检查，核心问题是：“我们应该在什么时候，以什么方式允许数据在不同的并发域中传递？”

这个问题的答案是相当明确的：只有那些不会在并发访问时发生竞争和危险的类型，可以在并发域之间自由传递。不过即使答案明确，它所涵盖的具体类型也是多种多样的：

- 像是 `PersonStruct` 这样的所有成员都具有值语义，它自身也具有值语义的类型是安全的；
- 即使是 `class` 这样的引用类型，只要它的成员都是不可变量并满足 `Sendable` 的话，它也是安全的；
- 在内部存在数据保护机制的引用类型，比如 `actor` 类型或是成员访问时通过加锁来进行状态安全保证的类型；
- 可以通过深拷贝 (`deep copy`) 来把内存结构完全复制的情况；
- ...还有其他方式可以保证数据安全。

在 Swift 并发在设计针对跨越并发域的数据安全时，想要做到的事情有三件：

1. 对于那些跨越并发域时可能不受保护的可变状态，编译器则应该给出错误，以保证数据安全；
2. Swift 的并发设计，是鼓励使用值类型的。但是有些情况下引用类型确实可以带来更优秀的性能。对于 1 中的限制，应该为资深程序员留有余地，让他们可以自由设计 API，同时保证数据安全和性能；
3. Swift 5.5 之前已经存在大量的代码，如果强制开始 1 的话，可能会造成大量的编译错误。我们需要平滑和渐进式的迁移过程。

对于 1，我们使用 `Sendable` 来标记那些安全的类型；对于 2，Swift 留有 `@unchecked Sendable` 让开发者可以在需要时绕过编译器的检查；对于 3，Swift 5.5 中大部分关于 `Sendable` 的检查默认都是关闭的（这也是我们可以写出上面 `PersonClass` 的例子并编译通过，却让程序崩溃的原因）。接下来，我们就将围绕这三点展开讨论。

## Sendable 协议

Sendable 和现存在 Swift 中的所有协议都不同, 它是一个标志协议 (marker protocol), 没有任何具体的要求:

```
@_marker
public protocol Sendable { }
```

Sendable 所定义的“能在并发域之间被安全传递”的能力, 并不像 Hashable 的 hash(into:) 或者 Equatable 的 == 方法那样, 可以用若干个明确的方法进行要求。Sendable 这样的标志协议具有的是语义上的属性, 它完全是一个编译期间的辅助标记, 只会由编译器使用, 不会在运行期间产生任何影响。这意味着, 像是 x is Sendable 或者 x as? Sendable 这样的运行时判定是无法编译的。

虽然 Sendable 协议里没有任何要求, 但是如果我们明确声明某个类型满足 Sendable 的话, 编译器会对它的进行检查, 来确认是否确实满足要求。

## 值类型

Swift 标准库中的大部分基本类型, 都是满足 Sendable 协议的:

```
extension Int: Sendable {}
extension Bool: Sendable {}
extension String: Sendable {}
// ...
```

它们构成了其他标准库中的“容器”类型的基石。只要容器内的元素满足 Sendable, 那么容器本身也满足 Sendable。这类容器包括可选值、数组、字典等:

```
extension Optional: Sendable
    where Wrapped: Sendable {}
extension Array: Sendable
    where Element: Sendable {}
extension Dictionary: Sendable
    where Key: Sendable, Value: Sendable {}
```

这些类型在边界传递时，将发生复制或者稍后的写时复制 (Copy-on-write)，在新的并发域中，它们和原来的值不相关，因此它们是安全的。类似地，如果我们自己的 struct 类型中只包含 Sendable 的变量，那么这个类型本身也是 Sendable 的，PersonStruct 就是一个例子，我们可以明确地把它标记为 Sendable：

```
struct PersonStruct: Sendable {
    let name: String
    var message: String = ""

    init(name: String) {
        self.name = name
    }
}
```

当 struct 上有非 Sendable 成员时 (比如 class 这样的引用类型)，该成员可能会被多个并发域同时修改。显而易见，此时这个类型不能再满足 Sendable。这种时候为该类型添加 Sendable 协议，编译器将会给出错误：

```
class A {}

struct PersonStruct: Sendable {
    let name: String
    var message: String = ""
    let a: A = A()

    init(name: String) {
        self.name = name
    }
}

// Stored property 'a' of 'Sendable'-conforming struct
// 'PersonStruct' has non-sendable type 'A'
```

实际上，在同一模块中，如果一个 struct 满足它的所有成员都是 Sendable，我们甚至不需要明确地为它标记 Sendable。编译器将会帮助我们推断出这件事情，并自动让该类型满足协议：

```
// PersonStruct 被推断为 Sendable
struct PersonStruct {
    let name: String
    var message: String = ""

    init(name: String) {
        self.name = name
    }
}

// foo 函数要求参数满足 Sendable
func foo<T: Sendable>(value: T) {
    print(value)
}

// 可以编译通过：
foo(value: PersonStruct(name: "onevcat"))
```

这一点和我们熟知的 Hashable 或者 Equatable 有所不同。就算一个类型上的所有成员都满足 Hashable，我们也还是需要明确地声明这个类型满足 Hashable，编译器才能帮助我们自动生成所需要的实现。不过由于 Sendable 的使用会更加广泛，而且不像其他协议，为某个类型添加一个 Sendable 标志协议不会带来任何运行时的影响，所以尽可能地自动为合适的类型添加 Sendable，有助于保持简洁和避免开发者的重复劳动。

在上面我们强调了“同一模块”这个条件。当我们把该类型声明为 public 时，这个前提就不再存在了，在模块外，这样的类型不会被自动视为 Sendable。这是因为可能存在只有在模块内部才可见的 internal 或 private 成员。从模块外部，编译器将无法确定类型中所有的成员是否都是 Sendable，也就不能再帮助我们进行这个推断：

```
// ModuleA
public struct PersonModuleA {
```

```
let name: String
var message: String = ""

// 编译器无法确定非 public 成员
// 可能存在非 Sendable 成员，无法为 `PersonModuleA` 推断 `Sendable`
// let a = A()

public init(name: String) {
    self.name = name
}

// Module B
import ModuleA
foo(value: PersonModuleA(name: "onevcat"))
// 'foo(value:)' requires that 'PersonModuleA' conform to 'Sendable'
```

在把某个类型标记为 public 时，如果有条件，我们应该把它也明确标记为 Sendable，并作为公开 API 保证的一部分。这样有利于使用者将它在并发域之间进行传递。

如果我们能够确定某个 struct 不会再改变，我们可以使用 @frozen 进行声明。对于这样的 struct，它的成员不会再被修改，编译器也将有机会直接“窥视”并确定它的内部结构，从而隐式添加 Sendable，而不必担心在未来这个假设失效。不过要注意，在开发框架时这意味着对该 struct 的成员进行修改的话，将引入对 ABI 的破坏。

## class 类型

要让 class 类型满足 Sendable，条件要严苛得多：

- 这个 class 必须是 final 的，不允许继承，否则任何它的子类都有可能添加破坏数据安全的成员。
- 该 class 类型的成员必须都是 Sendable 的。

→ 所有的成员都必须使用 let 声明为不变量。

这些条件可以确保 class 类型在不同并发域中的安全。不过，即使如此，编译器也不会像对待 struct 那样，为它自动添加 Sendable。想要让 class 类型满足 Sendable，我们必须明确进行声明：

```
class PersonClass: Sendable {  
    let name: String  
    let message: String = ""  
  
    init(name: String) {  
        self.name = name  
    }  
}
```

## actor 类型

虽然和 class 一样，actor 也是引用类型，不过 actor 内部的隔离机制保证了内部状态的安全。在不同并发域中对 actor 成员进行访问，最终都会在该 actor 的隔离域中发生。因此所有的 actor 类型都可以随意地在并发域之间传递，它们都满足 Sendable：不论在哪个模块中，也不论 actor 拥有什么类型的存储成员，编译器都会为它们加上 Sendable。

## 函数类型和 @Sendable 标注

除了像是 struct、enum、class 或者 actor 这样具体类型的值，函数也是会经常在并发域之间传递的类型之一。在 Swift 中，函数类型的值可能会有各种形式：比如全局函数、getter/setter 或者匿名闭包等等。在 Swift 中，函数类型也是引用类型，它会在函数体内部持有对外部值的引用。在跨越并发域时，在函数体内的这些被引用的值可能会发生变化。想要保证数据安全的话，我们必须规定函数闭包所持有的值都满足 Sendable 且它们都为不变量。

不过在 Swift 语法中，函数类型本身并不能满足任何协议。为了表示某个函数参数必须满足 Sendable，我们使用 @Sendable 对这个函数进行标注。这种模式和在处理闭包逃逸时所使用的 @escaping 有些相似：

```
func bar(value: Int, block: @Sendable () → Void) {  
    block()  
}
```

我们也可以在 Task 或者 AsyncStream 等相关 API 的设计中看到这个标注，它们广泛存在于 Swift 异步 API 中：

```
extension Task where Failure = Never {  
    init(  
        priority: TaskPriority? = nil,  
        operation: @escaping @Sendable () async → Success  
    )  
}  
  
struct TaskGroup<ChildTaskResult> {  
    mutating func addTask(  
        priority: TaskPriority? = nil,  
        operation: @escaping @Sendable () async → ChildTaskResult  
    )  
}  
  
struct AsyncStream<Element> {  
    struct Continuation : Sendable {  
        var onTermination: (@Sendable (Termination) → Void)?  
        { get nonmutating set }  
    }  
}
```

在使用这些函数时，编译器会对传入的函数进行检查：

- 被函数体持有的变量，必须都是 Sendable 的；
- 所有被持有的值，都必须是使用 let 声明的不变量。

比如，下面这段代码将产生编译错误：

```
var name = "hello"
Task {
    let validName = "onevcat" = name
}

// 编译错误：
// Mutation of captured var 'name' in concurrently-executing code
```

虽然 `String` 是 `Sendable`，但是在 `Task.init` 的闭包中，对它的捕获将导致编译错误：`name` 有可能在多个不同任务上下文中被更改。如果我们只需要读取 `name` 的内容，可以将它明确地写在闭包的捕获列表中，或者使用 `let` 来声明这个变量（这样它会被作为值隐式地复制到闭包中）：

```
var name = "hello"
Task { [name] in
    let validName = "onevcat" = name
}

// 或者

let name = "hello"
Task {
    let validName = "onevcat" = name
}
```

因为函数类型实际上是引用类型，你可以把它想象成一个 `class` 类型，其成员就是它所持有的所有捕获值。这样一来，让一个函数类型满足 `Sendable` 所需要的条件，就可以和 `class` 类型所需要的条件进行类比了。

不过在当前 Swift 5.5 中，只有本地声明（像是上面的 `name`）的 `var` 或者 `let` 会在 `@Sendable` 中被检查。对于被声明在其他代码域中的变量，还并不在默认的检查列表中。对于被捕获的变量是否遵守 `Sendable` 的检查，也还没有开启。这是为了迁移方便的考虑，完整的数据安全检查将在 Swift 6 中实现，我们后面会再讨论这个话题。

## 错误

另一类重要的会在并发域中传递的类型是各种错误：当异步函数出错时，我们会使用 throws 抛出错误，这在语义上其实相当于从并发域中返回了一个错误给调用者。基于安全要求，Error 类型应该始终是 Sendable 的。其实现在 Swift 中 Error 协议也确实被这么标记了：

```
protocol Error : Sendable {  
}
```

更改 protocol 的需求，显然是一个破坏性的修改。试想如果在 Swift 5.5 之前，我们已经有一个满足 Error 但是存在可变状态的类型，那么在编译器要求 Error 强制满足 Sendable 后，之前的代码将无法继续编译，例如这样一个类型：

```
class Detail {  
    var text: String  
}  
  
struct SomeError: Error {  
    var detail: Detail  
}
```

将会因为 detail.text 无法被保护，而无法满足 Sendable。这是一个源码级别的不兼容。为了平滑迁移，编译器在 Swift 6 之前都会“网开一面”，暂时允许这样的 Error 类型存在。但是我们必须提早进行准备并进行迁移，以免在 Swift 6 中实现完全的数据安全时，被无尽的编译错误包围。

## @unchecked Sendable

actor 并不是引用类型保证数据安全的唯一手段，在 Swift 并发之前，为了 class 成员的数据安全，我们就已经有诸如加锁或者设定内部串行派发队列的方案了。这种在自身内部具有数据安全保证、原本就线程安全的类型，自然是在不同并发域之间安全传递的。但是 Sendable 的检查并没有办法在编译期间确定它的安全性，这种类型也无法直接被声明满足 Sendable。如果我们确信该类型是安全的，可以在 Sendable 前面加上 @unchecked 标记，这样编译器就会跳过类型的成员检查，相信开发者的判断，直接把这个类型认为是满足 Sendable 的。

```
class MyClass: @unchecked Sendable {
    private var value: Int = 0
    private let lock = NSLock()
    func update(_ value: Int) {
        lock.lock()
        self.value = value
        lock.unlock()
    }
}
```

不仅仅是在将旧有代码进行迁移时有用，在我们确实需要使用引用类型实现一些高效的数据结构，同时又需要和 Swift 并发的其他部分协作时，有时 `@unchecked Sendable` 也是唯一的可行选择。不过，要特别注意，能力越大，责任也就越大：通过 `@unchecked` 你可以让任意类型都满足 `Sendable`，但是错误的实现将会引入 bug，让你在并发运行时面临数据安全的问题。

## 渐进式迁移和 `@_unsafeSendable`

如果你正在开发一个会被别人使用到的模块（比如某个框架，或者主 app 中可能用到的某个 framework target），那么在迁移时还有一些额外需要注意的事项。

Swift 并发的理想状态，是在静态环境下（也就是编译时）避免所有的由于共享可变状态所带来的数据安全问题。这其中很大一部分，可以依靠 `Sendable` 的检查来进行。但是我们已经看到了，在 Swift 5.5 中，编译器并没有对 `Sendable` 进行完整的检查。不像异步函数、结构化任务和 actor 这些新加入的概念，`Sendable` 的存在和 Swift 5.5 以前的代码可能发生关联，比如我们可能会在并发域边界传递一个之前已有的 class。我们当然可以通过把这个 class 改为 actor 来让它满足 `Sendable`，但这同时这也意味着项目中其他使用了这个类型的部分，必须有异步运行环境来继续调用这个新的 actor。这种迁移的“扩散性”和它所伴随的难度，往往出乎预料，迁移本身也会需要花费更多时间才能完成。

在这种情况下，强行开始完整的 `Sendable` 检查，可能带来非常多的错误，无限期地拖长迁移所需要的时间，甚至让新的 API 也无法使用。因此作为第一阶段，Swift 选择了延后静态检查的时间点，让 Swift 5.5 的迁移相对容易一些。

对迁移来说，另一个重要的问题是不同模块之间的关系。假设你在开发一个模块，并尝试将它迁移到 Swift 并发，并用 Sendable 来保证数据安全，你将同时面临来自上游和下游的压力。和你关联的各个模块不太可能同时完成 Sendable 迁移，当你正在开发的模块打算迁移时，这个模块所依赖的模块，以及依赖这么模块的其他使用者，有可能还没有进行迁移。

Swift 并发数据安全的迁移不会是一蹴而就的，Swift 也需要避免这种“大面积传染”的迁移需求，尽可能让模块的迁移能够独立进行。为了达到这个目的，Swift 需要能区分已经完成了迁移的模块和尚未进行迁移的模块，并在编译时区别对待它们。

## 迁移的模块先于依赖的模块

如果你所依赖的模块还没有迁移到 Sendable，但你需要在并发域间使用某些其中的类型，那么这些类型有可能无法安全访问。

假设你正在依赖的模块 A 中有这样的定义：

```
// Module A
public class Cube {
    public let edge: CGFloat
    public init(edge: CGFloat) {
        self.edge = edge
    }
}
```

它虽然在语义上满足 Sendable，但是由于 Module A 还没有完成迁移，因此在我们自己的模块中，如果有某个类型使用了 Cube，那么理论上它将不能被推断为 Sendable：

```
import ModuleA

// 无法对 `CubeOwner` 进行 `Sendable` 推断
// 因为 Module A 还没有迁移，也不知道 `Cube` 是否满足 `Sendable`。
struct CubeOwner {
    let name: String
    let cube: Cube
```

```
}
```

因为 Cube 来源于其他人的模块, 所以在这种情况下, 想让 CubeOwner 满足 Sendable, 只有两种方式:

- 在自己的模块中为 Cube 添加 @unchecked Sendable 的假设, 这样编译器就可以推断出 CubeOwner 满足 Sendable 了:

```
extension Cube: @unchecked Sendable {}
```

- 或者直接将 CubeOwner 声明为 @unchecked Sendable, 无视掉 Cube 的真实状况:

```
struct CubeOwner: @unchecked Sendable {  
    let name: String  
    let cube: Cube  
}
```

对于未迁移模块中的 Cube, 因为没有更多信息, 为它做 @unchecked Sendable 的假设无可厚非, 这是我们能做到的最好的方式, 但是它在未来终将成为隐患。如果在 Module A 完成迁移后, 事实上 Cube 无法满足 Sendable, 但由于 Cube 存在 @unchecked Sendable 的假设, 编译器不会对 Cube 并非 Sendable 的事实作出任何反应。你的模块依然处于危险之中:

```
// Module A  
  
class A {}  
  
// 由于 class A 的存在, `Cube` 并非 `Sendable`  
  
public class Cube {  
    public let edge: CGFloat  
    private var a = A()  
  
    public init(edge: CGFloat) {  
        self.edge = edge  
    }  
}
```

```
// 自己的模块
import ModuleA
// 危险！但是编译器不会有任何警告或错误。
extension Cube: @unchecked Sendable {}
```

为了避免这个问题，并保证在未来启用完整的数据安全时，编译器能正确地发现此类问题，Swift 会启用渐进式的迁移策略，在导入依赖模块时进行判定：如果导入的模块还没有完成并发适配，那么先假设 Cube 这样的类型可以进行隐式 Sendable 推断。这样，就算没有明确为 Cube 标记 @unchecked Sendable，我们的模块也可以让 CubeOwner 满足 Sendable：

```
// Module A
public class Cube {
    public let edge: CGFloat
    public init(edge: CGFloat) {
        self.edge = edge
    }
}
```

```
// 自己的模块
import ModuleA
// Module A 还没有完成迁移，`Cube` 可以被认为是 `Sendable`
// 从而 `CubeOwner` 可以被判定为 `Sendable`
struct CubeOwner {
    let name: String
    let cube: Cube
}

let owner = CubeOwner(name: "name", cube: .init())
Task {
    // owner 可以在并发域之间传递
    print(owner.cube)
}
```

虽然可以进行隐式推断，但这里的 Cube 依然是有风险的。不过在当下，它所提供的安全性，和 @unchecked Sendable 是同等的。这让我们可以避免在我们自己的模块中强行添加我们无法确定的 @unchecked 标记。

在未来，当 Module A 完成并发迁移后，Cube 会有两种可能性：

- 如果 Cube 确实满足 Sendable，那么它会在 public API 中被标记为 Sendable。这种情况下，CubeOwner 的 Sendable 推断依然有效，而且现在可以保证这些代码是安全的了。
- 如果 Cube 最终不能满足 Sendable，那么在完全数据安全被启用时，编译器将会检测出 CubeOwner 不满足 Sendable 的错误。你至少可以确认代码并不安全，并再进一步考虑处理的方式。

无论 Module A 中最后是哪种结果，我们自己的模块都有机会避免由于 @unchecked 带来的错误假设和危险。

## 开发的模块先于模块的用户

模块迁移的另一个方向，是你的模块的用户还没有完成迁移，但他们需要依赖并使用你的已经完成迁移的模块。

如果你的模块的使用者还没有迁移到 Sendable，但你已经声明了某些函数只能接受 Sendable 的参数（比如某个方法只接受 @Sendable），那么这个方法将无法由使用者进行安全地调用。比如你的模块中原来有这样的代码：

```
// Your Module
public func bar<T>(
    value: T, block: () → Void
) {
    // ...
}
```

在完成迁移后，bar 可能会需要参数 value 和 block 都满足数据安全，这要求 value 的泛型类型从 T 变为 T: Sendable，且 block 应该被 @Sendable 修饰。所以在适配后，它的签名应该是：

```
public func bar<T: Sendable>(  
    value: T, block: @Sendable () -> Void  
) {  
    // ...  
}
```

如果进行严格的检查，那么在你的模块迁移后，这个模块的用户将无法再进行编译：因为在他的模块还没有迁移，参数类型不可能满足 Sendable：

```
import YourModule  
  
class A {}  
  
let a = A()  
// 在 YourModule 迁移前可以编译，但迁移后无法编译  
bar(value: a, block: { print(a) })  
// a 不满足 `Sendable`  
// block 捕获了 `Sendable` 的值，它不满足 `@Sendable`
```

为了避免这种情况的发生，在用户完全适配 Sendable 之前，编译器会选择忽略掉这些错误，最多给出一些警告：

- 对于泛型类型参数 value 的 Sendable 需求，和上一小节中的例子一样，编译器会默认 A 是 Sendable 的。虽然这可能带来潜在的数据安全问题，但是这在 YourModule 迁移之前也是一直存在的，它并没有让事情变糟。
- 对于 @Sendable，则在迁移前直接忽略掉这个检查。

你可以使用 @\_unsafeSendable 来替代函数的 @Sendable 标记，明确表明让编译器忽略检查行为。这样即使在未来 Swift 开启全面检查后，你的函数依然可以提供最大的兼容性，被“不安全”地调用。这个标记现在还只是 Swift 内部的私有标记，在未来它将被公开。

## 小结

本章中我们看到了几个和 actor 有关联的进阶话题。

全局 actor 的语言特性，主要是为了实现 `MainActor` 和保证 UI 编程的主线程特性而专门准备的。它让我们可以不受制约地在程序的各个部分共享一个特殊的 actor 隔离域。虽然不会很常用，但自定义全局 actor 的能力是并发编程工具箱中的利器，在某些情况下它可以帮助我们跨越类型、源码文件、甚至跨越模块，来把异步操作限定在同一个隔离域中。

可重入是并发编程中会面对的重要话题，actor 一方面对内部状态进行保护，一方面也允许异步函数调用带来的暂停。在交织执行期间，actor 状态的改变是合法合理的。在 actor 的函数中，`await` 前后可能会是完全不同的世界，这种情况下，对于函数执行环境和 actor 的内部状态，我们便不能再假设它们是不变的。

除了内部状态外，标志协议 `Sendable` 可以用来保护在并发域间传递的值的安全性。在 Swift 5.5 中，`Sendable` 并不会对写出异步和并发代码有太大影响。但是，想要写出正确的异步和并发代码，我们必须关注任何可能产生的数据竞争。即使编译器默认还不会给出错误，但为合适的类型添加 `Sendable`，不仅能够大幅提升代码安全性，也会帮助我们在之后 Swift 6 发布时轻松一些。

# 并发线程模型

10

并发本身的是更高效地完成多个任务。在前面的章节中，我们已经看到为了达成这个目的，Swift 并发提供了三种工具：

- 异步函数可以帮助我们写出简单的异步代码，Swift 并发中很多 API 也都是通过异步函数提供的；
- 通过组织结构化并发，可以保证任务的执行顺序、正确的生命周期和良好的取消操作；
- 利用 actor 和 Sendable 等，编译器能保证数据的安全。

有了这些工具，我们可以构建出一套安全有效的并发机制。但是我们有时候可能会好奇，这套机制背后是怎么运行的，它的效率究竟如何，我们怎么才能保证并发程序运行良好？这些将会是本章想要尝试解释的话题。

## 协同式线程池

在 Swift 并发中，我们其实很少直接用“线程”的概念，这是因为组织和运行异步函数的单元并非线程。相比起说“一个同步函数运行在某个线程中”，对于异步函数，我们经常看到的描述是一个异步函数运行在某个任务中”。线程之于同步函数，就如任务之于异步函数。

虽然在最终，一段代码，无论它位于同步函数中还是异步函数中，都必须由某个具体线程来运行。但是和始终运行在同一个线程上的同步函数不同，异步函数可能被 await 分割，并由多个不的线程协同运行。Swift 并发在底层使用的是一个新实现的协同式线程池 (cooperative thread pool) 的调度方式：由一个串行队列负责调度工作，它将函数中剩余的运行内容被抽象为更轻量的续体 (continuation)，来进行调度。实际的工作运行由一个全局的并行队列负责，具体的工作可能会被分配到至多为 CPU 核心数量的线程中去。

需要强调的是，协同式线程池之所以要限制线程数量，是为了避免线程级别的切换，进而避免性能问题。具体来说，Swift 并发中的续体代表了一个运行时的状态包，await 将函数的剩余部分“注册”为一个续体并暂停起来，然后在某个工作线程执行当前的语句。Swift 并发的运行时可以轻易地在多个续体间进行切换，它更像一个轻量级的线程。在其他支持并行的语言中，也有类似（但不完全一样）的概念，比如 Go 中的 Goroutines，Rust 的 tokio 中的 task 或者 Crystal 的 fiber。我们有时候也会用绿色线程 (green thread)、协程 (coroutine) 或者纤程 (fiber) 来称呼这些概念。虽然它们的涵盖范围略有不同，但是核心是一致的：它们提供一种和

系统级别的线程不一样的，更轻量的调度方式。在续体间切换的性能消耗，与普通的方法调用可以等价。这种续体切换要比在线程间进行切换容易得多。

为了能进一步研究续体的调度方式，我们先来看看传统线程调度，也就是 GCD 所面临的问题。

## 线程切换和线程爆炸

在 Swift 并发被引入之前，GCD 是最主流的线程调度方式：它以“抢占式”的方式管理一个线程池，在需要的时候，GCD 会尝试从线程池里获取已经创建但闲置的线程，但如果有需要或者线程池中已经没有可用线程时，它则会尝试创建新的线程。一个线程可能会被耗时操作占用或者需要等待某个锁，此时这个线程就处于被阻塞的“不可被分配”状态。这种时候，有新任务需要处理时，新的线程将被创建，并分配给某个 CPU 核心去执行新任务中的指令。

理想状况下，如果正在运行的线程数小于或等于 CPU 的核心数，那么每个线程会被分配到一个它单独占有的核心上，这个 CPU 核心可以“专心地”运行和处理该线程中的指令。不过，如果线程数量超过了核心数的话，新创建的线程将会被分配给已经正在运行其他线程的核心。这时，一个 CPU 核心将会同时处理两个或更多的线程。

你可以访问 `ProcessInfo` 的 `activeProcessorCount` 属性来获取设备上的 CPU 核心数。有时候这对指导你根据硬件条件写出更高效的并发代码会有帮助。

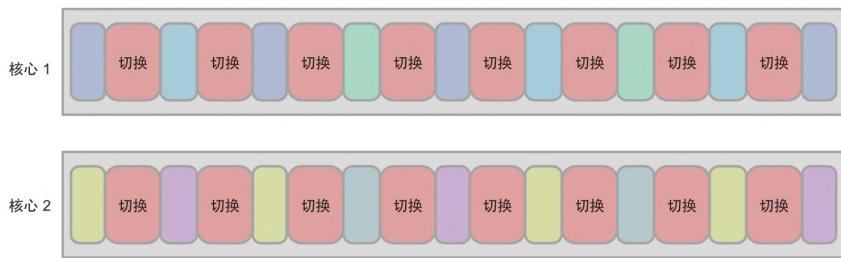
传统意义上，线程是操作系统进行运算调度的最小单位。线程们可以共享它所在进程中的内存堆等资源，同时它也拥有属于自己的一些资源：比如调用栈，自己的寄存器环境，以及动态申请的栈空间上的内存等。CPU 核心只是一个简单的指令执行器，在同一个 CPU 核心上同时执行两个线程的事实，决定了需要通过时分复用的策略让这两个线程共享 CPU 核心的计算资源，这也意味着在运行不同线程时，执行环境需要从一个线程切换到另一个线程。这种切换涉及了整个线程资源的切换，包括像是寄存器、栈指针和栈内存等。它相对轻量，但是也还是需要消耗时间：



在 GCD 中，调度库对并行队列和串行队列能够创建的线程总数是有限制的。不同的运行环境下限制会有不同，以目前已发布的最新 iOS 系统 (iOS 15) 和最新的硬件环境 (iPhone 13) 来说，单个并行队列最多可以创建 64 个线程，而串行队列可以创建 512 个线程。移动设备的 CPU 核心数和内存容量都是有限的，它们无法承载无限多的线程，这也是 Apple 在文档中要求我们避免创建过多线程的原因。

在 iOS 中，主线程的栈内存空间为 1 MB，其他次级线程的内存空间为 512 KB。如果我们不加限制地让 GCD 创建新线程，这些线程所占用的栈内存空间也将急速上升。一对串行队列和并行队列达到 GCD 限制时，栈内存就将占用到接近 300 MB。考虑到程序中有可能存在多个队列的事实，最后这会是一个不容忽视的数字。

太多线程不仅意味着更多的内存压力，更严重的是，这些线程在有限的 CPU 核心上的运行，会伴随着非常多的线程上下文切换。有时候，相比于实际执行我们需要的指令，这些切换所耗费的资源和时间反而成了主要部分。这种由于线程被阻塞的同时，又不断有新的任务被以 `async` 方式提交到并行队列，并造成过多新线程创建的行为，我们把它成为**线程爆炸 (thread explosion)**：



线程爆炸造成了过多的线程上下文切换，是传统并发编程中导致性能退化的重要原因之一。在 GCD 中，调度库对线程数量进行了限制，相比于直接使用 `NSThread` 的 API，通过 GCD 进行调度已经为性能优化带来了很大改善，但这并不十分理想：作为开发者，我们依然需要把精力分配给线程创建这样的细节，特别是在 GCD 中，线程的分配和创建细节是被隐藏起来的，稍不留意就可能造成问题。

下面的一段代码会造成典型的线程爆炸。由于 `sQueue` 被阻塞，导致并行队列 `.global() async` 所调用的闭包无法及时完成，新的 `async` 将一直创建新的线程，直到上限：

```
let sQueue = DispatchQueue(label: "serial-queue")

for i in 0 ..< 10000 {
    DispatchQueue.global().async {
        print("Start \(i).")
        self.sQueue.sync {
            Thread.sleep(forTimeInterval: 0.1)
            print("End: \(i).")
        }
    }
}
```

如果在运行期间，你使用 Xcode 的 debugger 暂停按钮，可以在调试面板中看到所有的运行中的线程。用对应的 LLDB 命令 `thread list` 也能得到同样的结果：

```
(lldb) thread list
```

```
* thread #1: ... queue = 'com.apple.main-thread'  
thread #2: ... queue = 'com.apple.root.default-qos'  
...  
thread #65: ... queue = 'com.apple.root.default-qos'  
...
```

你得到的结果可能序号会有所不同，但是 default-qos 队列（也就是 .global 队列）对应的线程总数为 64。序号不同是由于除了 default-qos 创建的线程外，还会有一些其他的默认存在的线程（比如 UIKit 的获取事件的辅助线程等）。

对于超过线程限制数的 `async` 派发，GCD 将把被派发的闭包缓存在堆上，并在任一原来被阻塞的线程完成任务后，将被缓存的闭包交给这个线程执行。

在 Swift 并发中，Apple 对线程调度进行了进一步的封装，把“线程”的概念整个隐藏到了幕后。但实际上，不论是 Task 相关结构化任务 API 的调度，还是 actor 隔离域之间的切换，在幕后都会涉及到执行线程的问题。甚至可以说，如果不对现有的线程调度方式进行革新，想要支撑 Swift 并发的新一套 API，在底层可能会带来更多的潜在的线程切换的机会。这种切换带来的性能上的问题，将会摧毁 Swift 并发被大规模使用的可能。为此，Apple 需要一套相应的手段来避免线程爆炸和它所带来的问题。

## 非阻塞线程约定

为了解决线程爆炸的问题，似乎最直截了当的方法是人为限制线程数量，让调度系统不创建多于 CPU 核心的线程数。GCD 现在似乎已经为我们把并发队列的线程数限制为 64 了，那是不是进一步限制到 6 或者 8 就能解决问题？

答案是否定的，不然我们也不需要在这里啰嗦了。其实当前 GCD 对线程数量的限制，是一种迫不得已的权衡。线程爆炸的核心原因在于串行队列的线程被阻塞，进而使并行队列线程进入等待，导致 CPU “空闲”。在这个前提下，GCD 倾向于创建更多线程来让 CPU 继续工作。另一方面，有时候被某个线程持有的资源（比如某个信号量）可能会在其他线程被释放，足够的线程数可以保证程序不被永远挂起。在线程数太少时，这种挂起将更容易发生。比如在刚才上面的代码例子中，改为用 `DispatchSemaphore` 控制程序执行的话：

```
let sQueue = DispatchQueue(label: "syncqueue")
```

```
let count = 10

// 申请 10 个 DispatchSemaphore
let semaphores = [DispatchSemaphore](
    repeating: .init(value: 0),
    count: count
)

// 设置信号等待
for i in 0 ..< count {
    DispatchQueue.global().async {
        print("Start \(i).")
        self.sQueue.sync {
            // 阻塞 `sQueue`，等待 semaphores[i] 的信号
            semaphores[i].wait()
            print("End: \(i).")
        }
    }
}

// 发送信号
for i in 0 ..< count {
    DispatchQueue.global().asyncAfter(deadline: .now() + 0.5) {
        // 在其他线程向信号量发送信号
        semaphores[i].signal()
    }
}
```

当 count 为 10 时，GCD 会为设置信号的并发队列的 async 分配十个线程。接下来在发送信号时，GCD 创建新的可用线程，来发送这些信号，此时 semaphores[i].wait() 所造成的等待会依次结束，sQueue 得以继续执行并最终输出 End：

```
// 输出 (你大概率会得到不同的输出顺序):  
// Start 0.  
// Start 3.  
// Start 1.  
// ...  
// End: 7.  
// End: 8.  
// End: 9.
```

不过，如果我们把 count 修改一下，比如当它是一个大于 63 的值时，我们就看不到任何 End 的输出了：

```
// let count = 10  
let count = 100
```

```
// 输出:  
// Start 0.  
// Start 2.  
// Start 1.  
// ...  
// Start 63.
```

前 64 个派发的闭包被分配到了各自的执行线程上，但它们永远卡在了等待信号上：因为这些发出信号的工作自身也在等待可以使用的线程，但所有可用线程都在等待信号，新的可用线程将永远不会出现。这类问题会非常难调试，还可能随着运行环境的不同而产生不同的现象。

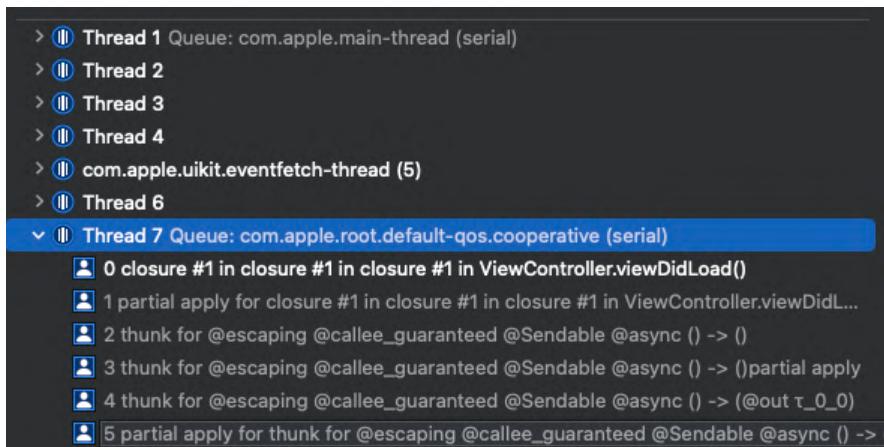
不仅是信号量，像是锁或者其他一些同步手段，在跨越线程进行操作时，都有可能让线程产生这种滞止现象。当可用线程数较少时，这种情况就尤为严重。但是为了避免在同一个 CPU 核心上进行线程切换，我们又想要较少的线程数。在传统 GCD 模型下，这是一对难以调和的矛盾。

线程爆炸的最本质原因是串行队列线程的阻塞，因此，如果我们能找到一种办法，让串行队列不会阻塞的话，就能确保各并发线程都不会因为要等待串行线程而停滞，那么我们就可以实现一种用较少线程调度所有工作的方式。这种新的调度方式就是 Swift 并发中加入的协同式线程

池，而非阻塞线程的约定则是实现这种调度方式并令其保持高效运转的重要前提，也是异步函数得以实现的基础。

## 协同式调度线程模型

在本书写作时，Swift 并发在所有平台上底层都还是使用 GCD 进行调度，但是这并不是旧版本系统搭载的原味 GCD 库，而是一个带有全新的协同式实现的闭源版本。除非设定了 @MainActor，否则我们通过 Task API 提交给 Swift 并发运行的闭包，都会交给一个 cooperative 的串行队列进行处理。如果我们在一段异步代码中设置断点，很有可能会在栈列表中看到这个队列的名字：



The screenshot shows a call stack debugger in Xcode. Thread 7 is highlighted with a blue bar. The stack trace for Thread 7 is as follows:

- > ⓘ Thread 1 Queue: com.apple.main-thread (serial)
- > ⓘ Thread 2
- > ⓘ Thread 3
- > ⓘ Thread 4
- > ⓘ com.apple.uikit.eventfetch-thread (5)
- > ⓘ Thread 6
- ⌄ ⓘ Thread 7 Queue: com.apple.root.default-qos.cooperative (serial)
  - 0 closure #1 in closure #1 in closure #1 in ViewController.viewDidLoad()
  - 1 partial apply for closure #1 in closure #1 in closure #1 in ViewController.viewDidLoad(...)
  - 2 thunk for @escaping @callee\_guaranteed @Sendable @async () -> ()
  - 3 thunk for @escaping @callee\_guaranteed @Sendable @async () -> ()partial apply
  - 4 thunk for @escaping @callee\_guaranteed @Sendable @async () -> (@out t\_0\_0)
  - 5 partial apply for thunk for @escaping @callee\_guaranteed @Sendable @async () ->

异步函数执行的另一个可能的队列是绑定了主线程的 main queue。当异步代码需要被隔离在 MainActor 时，将等效于 DispatchQueue.main 的派发，这个派发会选择主线程来执行指令。

为了能把线程数控制在设备上 CPU 的核心数以内，我们不能让 cooperative 串行队列对应的线程被阻塞。运行在这个线程上的异步函数需要具有放弃线程的能力，这样该线程才能保持向前，去执行其他操作。为了做到这一点，协同式队列的调度需要具有额外的能力，把还未执行的函数部分和必要的变量包装起来，作为续体暂存到其他地方（比如堆上），然后等待空闲的线程去执行它。Swift 并发的调度器会组织这些续体，让它们在线程上运行：



## 图示异步线程模型

我们通过一些图解来仔细看看这个串行队列(以及对应它的线程)是如何做到保持不阻塞的。假设我们有下面的代码：

```
func bar1() {}

func bar2() async {}

func bar3() async {
    await baz()
}

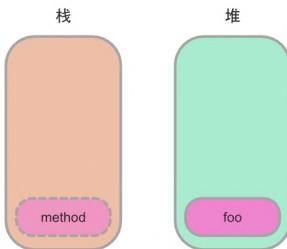
func baz() async {
    bar1()
}

func foo() async {
    bar1()
    await bar2()
    await bar3()
}

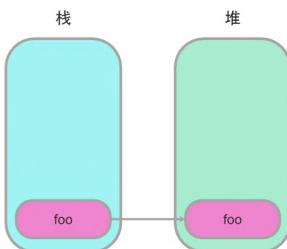
func method() {
    Task {
        await foo()
    }
}
```

{

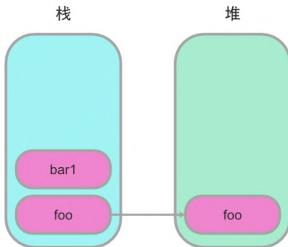
- 当某个线程执行 method 时，Task.init 首先被入栈，它是一个普通的初始化方法，在执行完毕后立即出栈，method 函数随之结束。通过 Task.init 参数闭包传入的 await foo()，被提交给协同式线程池，如果协同式调度队列正在执行其他工作，那么它被存放在堆上，等待空闲线程：



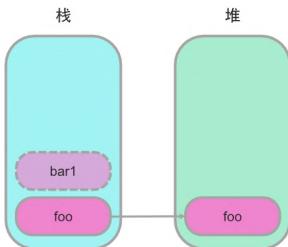
- 当有适合的线程可以运行协同式调度队列中的工作时，执行器读取 foo 并将它推入这个线程的栈上，开始执行。需要注意的是，这里的“适合线程”和 method 所在的线程并不需要一致，它可能是另外的空闲线程：



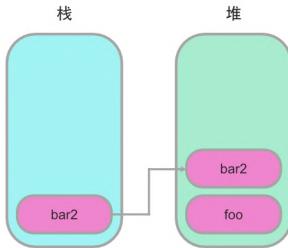
- foo 中的第一个调用是一个同步函数 bar1。在异步函数中调用同步函数并没有什么不同，bar1 将被作为新的 frame 被推入栈中执行：



4. 当 bar1 执行完毕后，它对应的 frame 被出栈，控制权回到 foo，准备执行其中的第二个调用 await bar2():

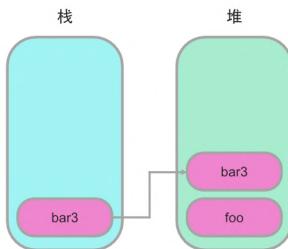


5. 接下来我们会在一个线程中执行到 await bar2(), 它是一个异步函数调用。为了不阻塞当前线程，异步函数 foo 可能会在此处暂停并放弃线程。当前的执行环境(如 bar2 和 foo 的关系)会被记录到堆中，以便之后它在调度栈上继续运行。此时，执行器有机会到堆中寻找下一个需要执行的工作。在这里，我们假设它找到的就是 bar2。它将被装载到栈上，替换掉当前的栈空间，当前线程就可以继续执行，而不至于阻塞了：



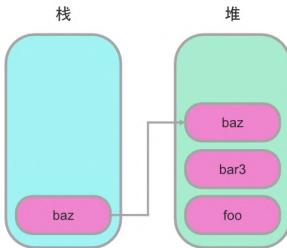
当然，执行器也有可能寻找到其他的工作（比如最近有优先级更高的任务被加入），这种情况下 `bar2` 就将被挂起一段时间，直到调度栈有机会再次寻找下一个工作。不过不论如何，串行调度队列都不会停止工作。它要么会去执行 `bar2`，要么会去执行其他找到的工作，唯独不会傻傻等待。

- 当 `bar2` 执行完毕后，它被从堆上移除。因为在执行 `bar2` 前，我们在堆上保持了 `foo` 和 `bar2` 的关系，因此在 `await bar2()` 结束后，执行器可以从堆中装载 `foo`，并发现接下来需要运行的指令。在我们的例子中，`await bar3()` 将被运行。和 `bar2` 时的情况类似，底层调度使用 `bar3` 替换掉栈的内容，并继续在堆上维护返回时要继续执行的续体：

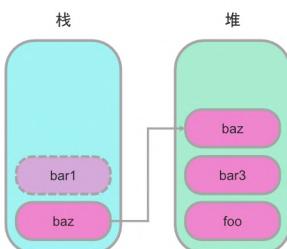
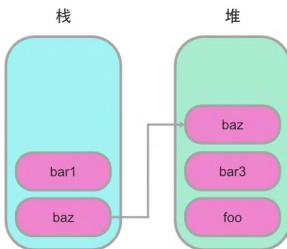


需要注意，`await bar2()` 前后代码可能会运行在不同线程上，除非指定了 `MainActor`，否则协作式调度队列并不会对具体运行的线程作出保证。

7. `bar3` 中的第一个调用是 `await baz()`。这是一个在异步函数中调用其他的异步函数的情况，实质上它的情况和 `foo` 中调用 `await bar2()` 或 `await bar3()` 是相同的。`baz` 会替换调度队列所对应的线程的栈：



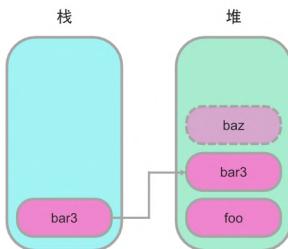
8. 在这个栈中，同步方法 bar1 的调用依然在当前栈上进行普通的入栈和出栈：



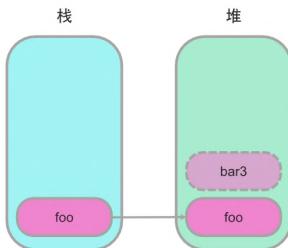
在异步函数定义的栈上调用同步函数，所执行的是普通的出入栈操作。因此在 Swift 异步函数中，我们是可以透明地调用 C 或者 Objective-C 的同步函数的。在底层，这种调用就是普通的同步调用。

9. 当 baz 完成后，执行器从堆中找到接下来的续体部分，也就是 bar3，并将它替换到某个线程的栈中。虽然已经多次说明，但笔者依然想再强调一次，此时 bar3 的执行线程可

能会和 baz 不同，也可能和 bar3 最早的执行线程不同，(虽然大部分情况下是一致的，但这是一个实现细节) 我们不应该对具体的执行线程进行任何假设：



10. 最后，bar3 的执行也结束了，执行器最终寻找到一开始的 foo，并最终完成整个 Task 的执行：



## 异步线程模型小结

调度库将续体暂存在堆上，并在需要的时候用它“替换”掉调度队列线程的运行栈，是异步函数拥有放弃线程能力的基础。在调度线程空闲时（比如 await 后），执行器会为它寻找接下来需要处理的指令，这个指定可能是 await 所需要执行的部分，也可能是和之前完全不相关的其他任务。和传统 GCD 调度的资源抢占式不同，这种调度方式通过协作的方式，由执行器、需要处理的工作和调度队列一同，来保证线程向前运行，这也是我们把它叫做协同式线程池的原因。

## 调度队列的阻塞

非阻塞队列是协同式调度的基础，因此任何破坏这个假设，并让该调度队列阻塞的操作，都可能导致 Swift 并发的性能退化甚至是完全卡死。

Swift 在语言层面上，使用 await 和 Task 相关的 API 来在编译期间保证非阻塞线程的约定：当我们在使用这些语言特性时，线程模型可以在堆上追踪执行工作所需的依赖，并按需替换栈上内容，保持线程在某个方法暂停后，能找到接下来的工作并继续执行。

但是引入 Swift 并发之前的其他一些线程同步手段，可能会造成不同程度的问题。

## 锁

像 NSLock 这样的锁，在不跨越 await 的前提下，是安全的。比如要保护非 actor 实例中的变量。考虑下面的代码：

```
class A {
    private let lock = NSLock()
    var values: [Int: Int] = [:]

    func foo() {
        Task.detached {
            self.lock.lock()
            self.values[1] = 100
            self.lock.unlock()
            await self.baz()
            print("Task Done")
        }
    }

    func bar() {
        lock.lock()
        values[1] = 0
        lock.unlock()
    }
}
```

```
func baz() async {
    try? await Task.sleep(nanoseconds:100)
}
```

上面的代码中，实例变量 values 存在被多个线程同时修改的风险，必须采取合理的数据同步手段。没有加锁的话，多线程下对它的调用将导致崩溃：

```
class A {
    func foo() {
        Task.detached {
            // self.lock.lock()
            self.values[1] = 100
            // self.lock.unlock()
            await self.baz()
            print("Task Done")
        }
    }

    // ...
}

let a = A()
for i in 0 ..< 10000 {
    // 数据竞争导致崩溃
    DispatchQueue.global().async {
        a.foo()
        a.bar()
    }
}
```

最好的解决方式当然是把整个类型改为 actor，依靠编译器来保证读写的单一性。但是如果由于某些原因，我们不得不用锁来让实例达到线程安全的话，则必须保证 lock() 和 unlock() 的调用发生在 await 同一侧。如果无法遵守这个约定，那么在协同式调度队列的线程第一次 await

完成替换后，lock 还处于锁定状态，接下来在同一个线程可能会去执行的其他任务（比如第二次 foo()），由于要等待 lock 资源，这可能会让调度线程处于阻塞状态，从而其他所有任务也都无法继续执行。因此，我们应该杜绝这样的代码：

```
class A {
    func foo() {
        Task.detached {
            self.lock.lock()
            self.values[1] = 100
            await self.baz()

            // 危险！可能永远无法执行
            self.lock.unlock()
            print("Task Done")
        }
    }

    // ...
}
```

## 信号量

和锁不同的是，由 DispatchSemaphore 或 NSCondition 代表的信号量在 wait 时将无条件直接阻塞当前的线程。在协同式调度的上下文中，调度线程被信号量阻塞，意味着直到某个其他线程发出信号前，这个协同调度都将无法执行其他操作。这样一来，调度队列线程是否能够运行，将取决于某个完全无法预先确定的其他线程的行为。除非经过非常精心的同步设计，否则使用信号量大概率会导致调度线程不再工作，从而违反非阻塞线程的约定。

## 耗时的同步函数

另外一种应该注意的情况是，我们不应该在调度线程中执行长耗时的同步任务：比如大的 I/O 或者其他可能长时间占用线程的操作。这些操作虽然不会导致调度线程完全停滞不前，但是在串行队列线程中执行这样的操作，无疑将会拖慢其他并发任务的调度，使并发性能退化：

```
// 避免类似这样的代码：  
func blockingMethod() async → Bool {  
    // 举例，某个阻塞线程的例子。可能是某个耗时的同步函数  
    Thread.sleep(forTimeInterval: 1.0)  
    return true  
}  
  
for _ in 0 ..< 10000 {  
    Task {  
        _ = await blockingMethod()  
        print("Done")  
    }  
}  
  
// 所有 `blockingMethod` 执行完毕需要一万秒，而不是一秒！
```

在这个例子中，我们使用了 `Thread.sleep` 来让线程休眠，来模拟可能阻塞线程的操作。要注意，`Thread.sleep` 和 `Task.sleep` 是完全不一样的：后者将通过派发把一个用于等待的工作添加到调度队列中，而不是阻塞当前线程。

对于系统 SDK 给出的异步函数，比如 `Task.sleep` 或者 `URLSession` 中的异步函数版本，在它们内部实现中，具体工作被转换为可以被执行器处理的对象并提交给执行器。随后执行器在协同式线程池中为它寻找合适的工作线程（这通常是一个并行队列管理的线程）并进行执行。对于一般的同步函数定义的耗时工作（比如直接通过 URL 初始化数据的 `Data.init(contentsOf:options:)` 这类方法），我们暂时还没有办法将它直接提交给执行器并纳入到协同式线程调度的系统中。因此，在串行调度队列中，我们需要小心处理这样的耗时操作。

在下面我们具体了解 Swift 并发执行器后，会再回到这个问题并看看应该如何处理。

## 执行器

非阻塞线程的保证解决了如何有效进行异步和并发调度的问题。而 Swift 并发底层的另一个模块，执行器 (`executor`)，则实际负责创建线程并保证接受协同式调度的线程不多于 CPU 核心数。

当前 Swift 并发提供了两种类型的执行器：一种是全局的并发执行器，它负责寻找合适的并发队列来为并发操作提供线程；另一种是串行执行器，它主要被用在 actor 中。每个 actor 会持有一个串行执行器，它负责保证 actor 隔离域的方法在串行队列中执行。

## 全局并发执行器

全局并发执行器对 Swift 并发高层 API 来说，是完全被隐藏的，它是 Swift 并发库中由 C++ 进行实现的部分，高层 API 无法对它进行直接调用。想要使用它，只能在 SIL 或更低层完成，对于 app 开发者来说，这是很难做到的。

为了合理地管理线程数，在运行时全局只有一个这样的并发执行器。通过 Task API 提交的任务以及调度队列中加入的任务，都将由协同式线程池的调度队列进行派发。不同于传统的 GCD，这是一个基于 GCD 的闭源实现。当协同式线程池中出现空闲线程时，这些工作将被并发队列实际分配给线程池中的线程进行运行。

协同式线程池是 Apple 的新版本系统的一部分。在像是 Windows 或者 WSAM 等非 Apple 的目标平台上，这个行为会有所不同。在这些不带有新的协同式线程池的环境中，执行器自己管理一个链表，并使用传统的派发方式进行调度。如果你面向不同平台开发，需要注意这可能将会导致并发代码性能的差异。

## Actor 执行器

除了隐藏起来的全局并发执行器外，Swift 并发还定义了另一种执行器：它们是串行的执行器。为了方便今后自定义执行器，Swift 在 5.5 中把执行器的协议暴露出来了：

```
public protocol Executor: AnyObject, Sendable {
    func enqueue(_ job: UnownedJob)
}
```

执行器协议需要做的事情只有一件：决定一项工作要如何被加入到执行器管理的队列中。Actor 中使用的串行执行器则是 Executor 一种细分协议：

```
public protocol SerialExecutor: Executor {
```

```
func enqueue(_ job: UnownedJob)
func asUnownedSerialExecutor() → UnownedSerialExecutor
}
```

每个 actor 都会拥有一个对串行执行器的引用：

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

在用 actor 关键字声明一个 actor 类型时，实际编译器会在 actor 的 init 和 deinit 中为我们加上对应的执行器初始化代码：

```
actor MyActor {
    // 等效编译为
    init() {
        // ...
        _defaultActorInitialize(self)
    }

    deinit {
        // ...
        _defaultActorDestroy(self)
    }
}
```

每个在 actor 隔离域外对 actor 的调用，会被转换为一次执行器的 enqueue，来将需要的操作作为“消息”加入到队列“信箱”中。执行器负责通过协同式线程池以串行方式为这些工作分配合适的线程。在实际中，除了 MainActor 需要被派发到主线程外，大部分情况下 actor 的执行会直接使用上面提到的负责协同调度的串行队列进行，这样可以避免线程切换以提高性能。不过我们在高层级上依然不应该进行这个假设，一方面是因为实现细节可能会改变，另一方面同一段代码的运行环境的改变（比如被移动到了 MainActor 中），也可能让这个假设失效。由于可能会影响调度线程，因此在 actor 的方法中，我们也不应该进行繁重的同步调用。这种耗时的同步工作依然有可能造成 Swift 并发性能的退化，甚至让其他任务无法运行。

## GCD 代码性能

在串行调度队列中，我们提到了应该避免耗时的同步调用。如果能够把这些同步调用进行封装，并传递给全局执行器让它负责将这些调用通过并发队列派发到各个工作线程的话，我们就可以解决这个问题。但不幸的是，这个派发方法现在还不对外部开发者开放。在本书写作时，Swift 已经拥有一个自定义执行器的提案，但是它还没有被正式接受和实现。上面的 Executor 协议虽然在事实上规定了一些实现执行器所需要的方法，但是我们还不能真正使用它们。在今后，如果自定义执行器的特性被加入 Swift 并发的话，我们也许可以使用这样的代码来将一个耗时的同步任务提交给协同式线程池进行调度：

```
// 当前还不可用
await globalExecutor.run {
    someHeavyMethod()
}
```

现在，在遇到这种情况时，我们的常见做法是使用 `withUnsafeContinuation` 或 `withCheckedContinuation` 来把它们封装起来。我们在早先刚介绍异步函数的章节中已经看到过这两个方法了。不过需要特别注意，这两个方法的闭包依然是运行在串行调度队列中的。所以，为了避免阻塞，一般我们还是会选择使用 GCD 直接进行调度：

```
await withUnsafeContinuation { continuation in
    DispatchQueue.global().async {
        let result = someHeavyMethod()
        continuation.resume(returning: result)
    }
}
```

在 Swift 并发中直接用到 GCD 其实并非罕见：除了这种情况以外，更多时候我们可能会需要把原有的 GCD 代码迁移到异步函数。使用 `continuation` 来对 GCD 的调度进行包装是一个有效的方法，但不幸的是，这种包装所造成的线程调度，并不会被自动“转换”到协同式线程池中，而会保持是一个“原汁原味”的 GCD 调用。这就意味着，如果我们没有留意派发关系，让并发队列对应的多个线程等待了某个串行队列线程的话，线程爆炸的情况依旧可能发生。因此，在处理 `withUnsafeContinuation` 时，我们需要小心这个问题。另外，除非真的有什么好的理由，否则我们都应该避免在异步函数中直接进行 GCD 派发，因为这种行为会绕开续体，并破坏结构化

并发的假设。在不得不使用 GCD 时，应当始终将它用 `with*Continuation` 包装起来，转换到 Swift 并发的 API 中。

## 任务优先级处理

传统 GCD 调度中，在进行派发把任务加入队列时，可以通过 QoS (Quality of Service) 指定“优先级”：

```
let queue = DispatchQueue(label: "com.swiftpal.dispatch.qos")  
  
queue.async(qos: .background) {  
    print("后台执行, 低优先级")  
}  
queue.async(qos: .userInitiated) {  
    print("用户触发, 高优先级")  
}
```

不过 GCD 中加入队列的任务是先入先出的：一个高优先级任务如果在低优先级任务之后才被加入到派发队列，那么它也会在这些低优先级任务之后再被提交和运行，这是 GCD 中无法改变的。如果严格按照加入时的优先级执行，那么可能发生优先级反转的问题。比如低优先级任务首先获取了一个锁，那么一个依赖同样锁的高优先级任务在加入后会被挂起，它需要等待这个锁被释放。大部分情况下因为低优先级任务也会运行，它最终会释放这个锁，所以暂时的等待不成问题。但是考虑如果有另外一些不需要锁的中优先级任务也在执行，调度器将会为这些中优先级的任务分配运行资源，这也意味着低优先级的任务可能被一直挂起，而导致锁始终无法释放。从最终结果来说，那个需要锁的高优先级任务会一直无法进行，而中优先级的任务反而顺利完成。这就是一个典型的优先级反转。

为了避免这种问题，GCD 采取的策略，是在检测到队列中有高优先级任务正在等待时，就把前面加入的低优先级任务也一并“翻转”为和高优先级任务相同的优先级，来让它们在调度时拥有同样的重要性：



这种方案能解决问题，但是由于调度的限制，远远谈不上优雅。和传统 GCD 在调度任务时的先入先出不同，Swift 并发中 Task 相关 API 在处理任务时，并发执行器对任务实际的派发，会灵活按照优先级将需要进行的工作在运行时调整到合适的位置。这让 Task 相关 API 的优先级设置能以更加可预测的方式工作：

```
for _ in 0 ..< 4 {
    Task(priority: .background) {
        print("in background task: \(Task.currentPriority)")
    }
}

Task(priority: .userInitiated) {
    print("in userInitiated task: \(Task.currentPriority)")
}

// 输出:
// in userInitiated task: TaskPriority(rawValue: 25)
// in background task: TaskPriority(rawValue: 9)
```



和 Task.init 和 Task.detached 类似，Task group 的 API 在添加子任务时，也有类似地接受优先级的方法：

```
await withTaskGroup(of: Void.self, body: { group in
    group.addTask(priority: .medium) {
        try? await Task.sleep(nanoseconds: 100)
        print("medium")
    }
    group.addTask(priority: .low) {
        try? await Task.sleep(nanoseconds: 100)
        print("low")
    }
    group.addTask(priority: .high) {
        try? await Task.sleep(nanoseconds: 100)
        print("high")
    }
})
// 输出:
// high
// low
// medium
```

如果在同一个任务组中，我们期望某些子任务获取更多的执行资源，那么为它们指定更高的优先级是有效的做法。

对于串行执行器 (也就是在 actor 中的调度)，我们必须确保执行顺序。原来的优先级提升的方法依然有效：当一个高优先级的任务被加入到串行执行器中，当前在执行的任务必须将优先级提升到和这个新加入的任务同样的优先级，以确保新的高优先级任务能够以正确的效率运行。这种情况下，原来的低优先级任务的 Task.currentPriority 并不会随着高优先级任务的加入而更改。因此，我们最好不要依赖 Task.currentPriority 这个 API 来决定代码的逻辑。就算需要使用，我们也应该记住它有可能并不能反应当前任务真实的运行优先级。

## 任务让行

由于同样优先级的任务共用一个调度线程，所以像是下面这样的代码，会导致其他任务无法运行：

```
func shouldLoopAgain() → Bool {  
    // 只是一个例子  
    return true  
}
```

```
Task.detached {  
    print("Task 1")  
    var loop = true  
    while loop {  
        // 实际工作  
        // ...  
        loop = shouldLoopAgain()  
    }  
    print("All Done")  
}
```

```
Task.detached {  
    print("Task 2")  
}
```

这种模式在一些等待/响应循环中 (比如分批次读入数据，或者服务器等待请求接入等) 会十分有用，但是 while true 循环将把整个调度线程占用住，导致其他任务无法运行。在并发中，我们

有时会把这种某些任务无法得到机会执行的现象叫做资源饥饿 (starvation)。上面的代码中，在 `shouldLoopAgain()` 返回 `false` 之前，“Task 2” 是没有机会运行的：

```
// 输出:  
// Task 1
```

为了其他任务能够运行，“Task 1” 必须要有暂时放弃线程的能力。最简单的方法是调用 `Task.yield`。这个方法将会通知当前任务挂起，并把剩余工作重新进行包装后再次放入执行器中进行派发。这样，当前线程就将被让出，它可以有机会执行其他任务：

```
// ...  
  
while loop {  
    loop = shouldLoopAgain()  
  
    if loop {  
        await Task.yield()  
    }  
}  
  
// ...  
  
// 输出:  
// Task 1  
// Task 2
```

一个细节是，在 Apple 平台的全局并发执行器中，为了性能考虑，它会为每个优先级创建并缓存一个协同式调度队列。这个细节使得下面这样的代码，在优先级不同时，即使 “Task 1” 没有 `yield`，“Task 2” 依然能够运行：

```
Task.detached(priority: .high) {  
    print("Task 1")  
    while true {  
        // 实际工作  
        // .. 可以不需要 yield, Task 2 也能执行  
    }  
}
```

```
}

Task.detached(priority: .low) {
    print("Task 2")
}

// 输出:
// Task 1
// Task 2
```

这是由于 .high 的任务和 .low 的任务是在不同调度队列上运行的，因此 .low 反而不受影响。不过，这完全是 Swift 并发执行器的内部实现，我们不应该依赖于这样的细节来组织任务的运行。而且就算 .low 的任务可以运行，其他的 .high 任务依然会被卡住。

实际上，虽然 Swift 并发在处理优先级时，要比传统 GCD 更容易预测一些，但是如果在情况变得复杂时，比如出现任务组和 actor 共同使用的情况下，优先级会迅速变得复杂起来。笔者的建议是，除非经过完善的性能测试，能确认并发运行的关键瓶颈就是某个任务的优先级，否则最好还是避免设定过于复杂的优先级。使用更简单的优先级体系，就更能保证派发队列也保持简单，从而使一些问题暴露得更加明显，这有利于我们在开发中尽早发现和修复它们。

## 任务本地值和任务追踪

对于任务的优先级 priority，以及是否被取消的 isCancelled flag，我们可以通过 Task 上的 static 属性进行获取：

```
func foo() async {
    let priority: TaskPriority = Task.currentPriority
    let cancelled: Bool = Task.isCancelled
}
```

虽然使用的是定义在类型上的 static 属性，但是实际获取到的值是当前运行环境中的具体任务实例的值。在异步函数中，只会存在单一的运行环境，所以直接使用 static 的属性可以合理地简化写法。比如 isCancelled 在 Swift 并发中的实现，就只是对当前任务的包装：

```
extension Task {  
    public static var isCancelled: Bool {  
        withUnsafeCurrentTask { task in  
            task?.isCancelled ?? false  
        }  
    }  
}
```

如果每次都要写像是 `withUnsafeCurrentTask` 这么复杂的语句的，会非常麻烦。使用 `static` 属性来在当前任务中共享值的方式虽然一开始看起来有点反直觉，但是确实十分便利，它利用了 `Task` 类型空间来携带一些元数据。Swift 并发中提供了一种语法特性，**任务本地值 (task local value)**，可以让我们也可以用类似的 `static var` 的方式，把元数据“注入到”当前任务绑定的某个自定义值中。

具体来说，对于任意一个类型中的静态的存储属性，我们都可以用 `@TaskLocal` 属性包装对它进行声明，将它暴露为任务本地值。和 `static` 的 `isCancelled` 类似，这个值只在当前任务中有效：

```
enum Log {  
    @TaskLocal static var id: String?  
}
```

`@TaskLocal` 属性包装会为被修饰的 `id` 属性添加一些特性。首先，它会把这个属性转变为只读，任何对它的直接设置将给出一个错误：

```
Log.id = "Hello"  
//Cannot assign to property: 'id' is a get-only property
```

想要设置这个值，必须通过使用 `Log.$id` 的 `WithValue` 方法。它接受一个值和一个闭包。在闭包中，读取 `Log.id` 将获得被设置的值，我们可以像访问通常的属性那样访问到它：

```
Log.$id.withValue("Hello") {  
    print("Log.id: \(Log.id ?? "")")  
}
```

```
// 输出:  
// Log.id: Hello
```

这个值可以在闭包中再一次被 Log.\$id 上的 `WithValue` 调用覆盖，并在任务之间进行传递，比如：

```
Log.$id.withValue("Outer") {  
    Task {  
        print("Log.id: \$(Log.id ?? "")")  
        await Log.$id.withValue("Inner") {  
            await Task {  
                print("Log.id: \$(Log.id ?? "")")  
                }.value  
        }  
        print("Log.id: \$(Log.id ?? "")")  
    }  
}  
  
// 输出:  
// Log.id: Outer  
// Log.id: Inner  
// Log.id: Outer
```

简单说，`Log.id` 将会寻找和返回最后一次 `Log.$id.withValue` 中所设定的值；如果一直向上都没有设定的话，它将返回定义时 `Log.id` 的初始值。在具体实现上，`WithValue` 被调用时，`@TaskLocal` 修饰的变量会将自己作为 `key`，把设置的值和当前任务的引用一并加入到一个链表维护的栈中，直到作用域结束后再出栈。它的简化后的代码类似于：

```
func WithValue<R>(  
    _ valueDuringOperation: Value,  
    operation: () async throws → R  
) async rethrows → R {
```

```
pushLocalValue(  
    key: self, value: valueDuringOperation, task: currentTask  
)  
defer { popLocalValue() }  
  
return try await operation()  
}
```

在使用 `Log.id` 获取值时，它会从当前任务开始，寻找对应 `key` 是否存储了某个值。如果在当前任务中没有找到，则到上层任务中继续寻找，直到寻找到对应值或者到达任务的根节点并返回默认值。沿着任务层级进行寻找，让我们可以避免在任务之间复制这些值。

如果你对 SwiftUI 比较熟悉，这个行为看起来会和环境值有些相似。在 SwiftUI 中，`@Environment` 和各种通过 `view modifier` 设定的数值（如 `padding` 等）环境值会从外层 `view` 传递到内层 `view`；而通过 `withValue` 设定的值，则是从顶层任务传递到底层任务。在 SwiftUI 中，环境值通常用来跨越 `view` 层级传递配置，让我们免于把某项配置层层传递。虽然在 Swift 并发中，我们也可以用任务本地值来做类似的事情，但是这种用法并不被推荐。它和 SwiftUI 的环境值面临类似的问题：究竟是谁为当前任务设置了这个本地值并不明确，而且当你将一个任务移动到其他地方时，可能原来的基于任务本地值的假设也会被破坏，但是编译器却无法检测到这种情况。SwiftUI 环境值和 `view` 的本体是一同工作的，它们共同构成一个有效和正确的 `view`，因此 `view` 的移动相对起来还能保持设定的完整。但这个情况在任务本地值中可能并不适用。如果想要在任务之间传递某种配置值，更好的方法还是明确地通过函数参数来进行，任务本地值不是为了传递参数而被设计出来的。

Apple 更推荐使用任务本地值来进行任务的追踪。比如，在多个页面中对同一个 API 进行请求时，单纯地通过控制台 `log` 或者断点，都很难追踪每一个任务：因为各个请求是并发运行的，它们产生的 `log` 可能也是重叠的，对它们进行断点，程序也可能多次停在重复的地方。比如：

```
func loadData() async → String {  
    print("loadData started.")  
    let profile = await getUserProfile()  
    print("profile got.")  
    let session = await getUserSession(profile.id, token)  
    print("session got.")
```

```
let result = await loadUserData(from: session)
print("loaded.")
Task {
    await self.storeToDatabase(result)
    print("cached.")
}
return result
}

// app 首页
_ = await loadData()

// app 设定页面
_ = await loadData()

// ... 其他
```

对于不同请求发送和接收的时机，我们可能看到的输出会是：

```
// 输出：
// loadData started.
// loadData started.
// profile got.
// session got.
// profile got.
// loaded.
// session got.
// loaded.
// cached.
// cached.
```

很难分辨输出对应的请求到底是来自哪个页面，也很难为断点设置合适的条件让它只在某个页面进行请求时停下。这种情况下，使用任务本地值就可以很好地解决任务追踪的问题：

```
// app 首页
Log.$id.withValue("app home page") {
    _ = await loadData()
}

// app 设定页面
Log.$id.withValue("app setting page") {
    _ = await loadData()
}
```

我们只需要将 print 语句的内容都加上 Log.id，输出就可以一目了然了，这也完全可以成为条件断点时使用的断言：

```
print("loadData started from \(Log.id ?? "not set").")
```

```
// 输出：
// loadData started from app home page.
// loadData started from app setting page.
// profile got from app setting page..
// session got from app home page.
// ...
// cached from app home page.
// cached from app setting page.
```

在上例中，storeToDatabase 被写在了 Task.init 的闭包中。在前面的章节我们也提到过，Task.init 会从当前任务环境中继承优先级和隔离环境等任务相关的属性，任务本地值也在其中：当通过初始化方法创建新任务时，当前任务的本地值链表将被复制到新的任务环境里。如果你想要一个完全“干净”的任务环境，可以使用 Task.detached 来创建游离任务。

当前 Apple 还没有在 Instruments app 中提供追踪和调试 Swift 并发性能的工具模板，不过任务本地值和它带来的任务追踪的能力，为今后进一步构建更强大的并发性能追踪工具提供了基础。

## 小结

本章中我们探索了一些 Swift 并发的幕后话题。为了避免线程爆炸和调度线程阻塞，Apple 提出了新的线程调度模型，这是异步函数可以放弃线程的基础；通过全局并发执行器，Swift 将任务包装并派发到协同式线程池，在那里完成底层线程的调度。这个线程池为了避免不必要的线程切换，不会创建多于 CPU 核心数的线程，而合理的调度方式也让各个线程不会产生资源饥饿。这些背后机制支撑了 Swift 并发，它们协同工作并保证在正确书写的前提下（比如避免在调度线程进行繁重工作，避免混用传统 GCD 派发等），即使情况变得非常复杂，Swift 并发也可以维持优秀的性能。

本章中我们也提到了一些其他的话题，比如任务优先级、任务让行和使用任务本地值来追踪任务等。在日常开发中，这些内容可能只会在很有限的情景下才被用到。不过相信这些知识可以帮助我们更深入地理解 Swift 并发的性能特点，并让我们可以追踪并发任务的执行。如果你在设计并发 API 时遇到了性能上的问题，希望本章中的内容能给你带去些许灵感。

# 总结和展望

11

Swift 并发的相关特性，可以说是 Swift 诞生以来到目前为止在语言层面上最重要的新特性了。通过在语言层面上支持异步函数，在调度上实现源生的结构化并发，以及加入 actor 保证数据同步和内存安全，Swift 在支持更容易和可用的并发方向上，迈出了坚实的一步。在 Swift 并发中使用的各种概念，大部分都是经过业界验证和实践的，Swift 团队也招募了不少具有丰富相关经验的开发者参与到 Swift 并发的开发中。Swift 并发的最终发布经过了漫长的讨论和完整的测试，不论从稳定性还是从易用性来说，可以认为 Swift 并发会都具有相当的保证。

当然，另一方面，作为一个新加入的特性，Swift 并发还是有很多可以改善和进化的地方。笔者在最后一章里，想对 Swift 并发的未来进行一些展望。

## 更低的系统要求

当前 Swift 并发的相关 API，包括 Task 和 actor 以及使用到异步函数的 URLSession 等的新加入的方法，都只能在 iOS 15/macOS 12 或更新的系统中使用。这个限制的主要原因是能支持新的调度方法，Apple 需要重新实现部分 GCD 的内部功能，而 GCD 是作为系统库的一部分打包在设备系统中的。暂时 Apple 只能依靠整个系统的版本来组织 API 的可用性的，现在生态中还缺乏单独和强制升级某个系统库的方式。

过高的系统版本要求，对于 Swift 并发的普及和推广显然是非常不利的。并发编程是大多数开发者每天都会面临的问题，而相关代码的编写可能也会经常进行。由于 Swift 异步和并发相关特性无法支持旧版本系统，从现在开始开发者们就可能面临着每天都在写“过时”代码的风险，并在今后若干年内不断积累技术债。从 Swift 并发在 WWDC 2021 上正式公布的那天开始，开发者社区就有很大的呼声，希望 Apple 能为旧版本的操作系统提供兼容。

Swift 团队确认正在努力为并发特性添加向后移植 (backport) 的支持，一些已经合并的代码表明这种支持最早可能可以部署到 Swift 5.1 的环境及 iOS 13 系列的操作系统中。这种支持将通过把相关代码额外单独打包为库并内嵌到 app 里，并通过扩展目标系统中旧的 Swift 运行时及 GCD，与它们合作的方式运行。这种手段和 Swift ABI 稳定前将整个 Swift 运行时打包到 app 中的方式有些类似，但是由于不能提供一整套运行环境，而需要使用旧版本 Swift 和 GCD，因此要实现起来会比之前困难许多。

除此之外，由于旧版本 GCD 缺少一些关键的特性（比如利用协同式线程池来限制和调度线程），可能兼容版本中只能退而求其次，用一些的取巧的方式或稍微低效的实现，来完成同样的内容，这可能也会导致 Swift 并发在旧系统上以兼容模式运行时出现一些性能上的差异。

在 Swift 5.5 的发布时，这个兼容支持尚未开发完成，有传言表示开发者们可能会在下一个 Swift 版本中看到 Swift 并发的兼容支持。

## 更完整的数据安全

`actor` 类型为被定义在其中的属性提供了安全保证，但是这并没有涉及到更一般的类型。Swift 并发当前在默认情况下并不是安全的，数据竞争甚至运行时由此造成的崩溃依然可能发生。

为了让从传统并发到 Swift 并发的迁移更加顺利并鼓励这种迁移，Swift 现在并没有进行强制和完整的数据安全检查。虽然已经定义了 `Sendable` 协议，但是不论是在多个模块之间的协作，还是在同一模块中跨越隔离域的方法调用，对 `Sendable` 的检查都不是默认开启的：这避免了一系列难以解决的编译错误，但是也牺牲了部分默认情况下应该在静态检查中就可以获取的数据安全特性。

按照 Swift 并发路线图的规划，默认的完整数据安全将在 Swift 6 中被开启。这个行为是破坏源码兼容性级别的修改，因此也只能在主版本升级中进行。如果你有机会在 Swift 6 之前就进行 Swift 并发的迁移，那么可以尽量在理解并发内存模型的基础上，去为合适的类型和它们的成员添加 `Sendable` 的标注。这让编译器可以在 Swift 6 到来前，就开始进行明确的检查，这样在今后破坏性更新到来的时候，你可以不用那么仓促。

一旦具有完整的数据安全后，我们就可以杜绝掉一整个系列的运行时错误。这种静态检查下就能确认数据安全和避免数据竞争的能力，是十分强大而诱人的。

## 更严格的内存模型

Swift 在很早就提出了关于内存的《所有权宣言》，它指出了内存管理的一些发展方向，比如强制性的内存独占原则等。Rust 以严格的内存所有权模型著称，并已经实现了其中大部分概念。Swift 可能在未来借鉴像是 `move` 或 `borrow` 这样的概念，来强化独占性。虽然这并不是专门为 Swift 并发所设计的语言特性，但事实上它将从比 `Sendable` 更加底层的位置，来确保内存安全。同时，它也可以提供更多的优化机会以提供更好的并发性能：比如把更多的值移动到栈上，减少独占值的复制，避免不必要的 `actor` 跳跃等。

虽然 Swift 团队已经在编译器底层对一些明显的所有权转移问题进行了实现和优化，但完整的内存所有权并不是 Swift 的短期目标，而且它可能带来更多复杂的概念（想一想让 Rust 代码编

译通过的难度吧！）。这和 Swift “保持简单易用”的目标，其实有些背道而驰。Apple 承诺不会让作为用户的程序员们在写代码时感受到“翻天覆地”的变化，但 Swift 要如何在这个话题上进行演进，是我们可以继续关注的方向。

## 更友好的调试体验

并发编程的一大难点在于调试和调优。人的大脑在理解线性叙事时，有着很强的能力。但是一旦将事情交织起来形成网状，人们就很难再正确对它们进行判断。并发编程中的各种事件互相联系，同时发生，它们所形成的正是一张这样复杂的网：如果遇到问题，开发者在处理起来会非常棘手。

结构化并发把这张网进行了一定程度的简化和假设：子任务的生命周期现在被严格框在了父任务中，不会产生逃逸，我们可以在父任务的生命周期内评估子任务们的行为。每个任务都具有确定的开始和结束，只要避免随意地创建完全的新任务，而是尽量依赖子任务来组织工作，我们就可以依据任务们的关系对它们进行追踪并确定合理的分析方式，这为并发编程的结构梳理打下了良好的基础。

我们已经有一些调试和监测 GCD 性能的工具了：比如 Instruments 中的 Time Profiler，可以帮助我们检查各个线程的工作情况；MetricKit 可以从真实的用户设备上收集到用户的性能信息。不过现在暂时还没有完整的专门针对 Swift 并发的方案。社区中已经开始有一些开发者尝试使用自定义的 Instruments 模板来监测并发编程任务。相信在未来一段时间内，Xcode 开发团队能为开发者们提供更加优秀的并发开发体验。

## 总结

对于 Swift 并发的探索，到这里就要告一段落了。并发编程虽然已经有很长的历史了，但是 Swift 并发本身却还很年轻，它仍然在高速发展。在本书里，我们已经对并发相关的若干话题进行了详细说明，但是不可避免地，还会有很多没有能涉及到的方面：比如分布式的 actor，自定义执行器的效果和具体方法，并发旧版本系统兼容的具体使用等等。它们之中有一些更多地是针对服务器开发的方向，有一些则是单纯因为 Swift 团队还没有准备好将它们公开。在接下来的几个 Swift 版本中，我们一定会看到更多的 Swift 并发相关特性，它们会为开发者带来更加优雅的处理方式和更加高效的运行能力。

本书在可以预见的未来，会有一些追加、更新和修正。为了方便记录这些历史，在后面我们也准备了一个简单的列表，可以帮助已经读过本书的读者追踪需要补充阅读的内容。如果在未来你有机会再次拾掇起这本书的话，希望这个列表能为你节省一些时间。

那么，感谢阅读。预祝一切顺利，生活美满！

## 更新履历

### 1.0 (2021 年 9 月)

→ 初版发布。