



SWIFTER

Swift 异步和并发

已对应 Swift 6

王巍 (@onevcat)

2.0 (2025 年 6 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

1	简介	7
	目标读者 8	
	章节结构 9	
	准备工作 10	
2	Swift 并发初步	13
	一些基本概念 14	
	异步函数 21	
	结构化并发 24	
	actor 模型和数据隔离 29	
	小结 34	
3	创建异步函数	35
	异步函数的动机 36	
	转换函数签名 40	
	使用续体改写函数 43	
	Objective-C 自动转换 49	
	Async getter 53	
	小结 58	
4	异步序列	59
	同步序列和异步序列 60	
	异步迭代器 62	
	操作异步序列 67	
	AsyncStream 76	
	异步序列和响应式编程 88	
	小结 94	

5	使用异步函数	95
	网络请求中的异步函数 96	
	Notification 106	
	异步函数的运行环境 108	
	小结 113	
6	结构化并发	114
	什么是结构化 115	
	基于 Task 的结构化并发模型 122	
	非结构化任务 144	
	小结 148	
7	协作式任务取消	150
	任务取消到底做了什么 151	
	处理任务取消 155	
	取消的清理工作 166	
	隐式等待和任务暂停 172	
	小结 174	
8	actor 模型和数据隔离	176
	共享内存模型的困境 177	
	Actor 隔离 180	
	Actor 协议 185	
	Actor 可重入 198	
	使用 @concurrent 控制执行器 204	
	小结 208	

9	Global Actor 和系统集成	210
	全局 Actor 概念 211	
	MainActor：主线程隔离域 212	
	UIKit 框架中的 MainActor 集成 216	
	自定义全局 Actor 221	
	全局 Actor 的设计原则 224	
	与系统框架的集成最佳实践 227	
	Actor 性能优化和调试 230	
	小结 244	
10	Sendable 协议，数据安全和迁移	246
	并发域间数据传递的挑战 247	
	Sendable 协议的设计原理 250	
	类型系统中的 Sendable 实现 252	
	Class 类型的 Sendable 实现 255	
	Actor 类型的自动 Sendable 257	
	函数类型和数据传递安全 258	
	Error 类型的 Sendable 要求 263	
	将 class 转变为 Sendable 264	
	并发检查级别和迁移清单 267	
11	并发线程模型	279
	协同式线程池 280	
	执行器 298	
	任务本地值和任务追踪 312	
	小结 320	

12 总结和展望**321**

总结 324

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，以至于那么多开发者都“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 标准库和日常开发框架 (Foundation, UIKit 等) 中的各个地方。如果您主要还在使用 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 相关的其他内容也会有兴趣，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。我们首先要厘清一些基本的概念，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却具有更广阔的意涵。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”的进阶读者，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) 等较小一些的概念。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，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异步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 --version` 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。在本书中，我们会使用本书修订时最新的 Xcode 16.4 以及其搭载的 Swift 6.1.2：

```
$ swift --version  
Apple Swift version 6.1.2 (swiftlang-xxxx clang-yyyy)
```

我们已经准备好开始进入到 Swift 并发的世界中了！

修订历史

版本 2.0 (2025 年 7 月)

重大里程碑更新：全面适配 Swift 6 和严格并发检查，将书籍从 Swift 5.5 时代升级到现代 Swift 并发标准。

Swift 6 现代化特性：

- 新增 `sending` 关键字详解，提供比 `@Sendable` 更灵活的所有权转移机制
- 介绍 `@isolated(any)` 语法和 `@concurrent` 执行器控制等 Swift 6.2 新特性
- 详细说明严格并发检查的三级模式 (`Complete/Targeted/Minimal`) 和渐进式迁移策略

新增完整章节：

- 《`Sendable` 协议和数据安全》：深入解析 `Sendable` 设计理念、`sending` 关键字应用和并发安全模式。

→ 新增 Swift 6 和并发迁移时的速查清单，帮助逐步迁移到新语言特性。

内容全面升级：

- Actor 章节新增可重入机制深度解析，包括交织现象识别和处理模式
- 结构化并发章节补充任务组错误处理和非结构化任务使用指南，深化了 goto 语句和结构化编程的历史背景，详细阐述了结构化控制流与栈内存管理的协同关系
- 异步序列章节更新错误处理机制，详细介绍 Swift 6 的类型化抛出（typed throws）特性，补充了自定义执行器的相关内容
- 协作式任务取消章节强化了“协作式”概念的解释，明确了为什么需要任务间通力合作才能实现有效取消
- 运行异步函数章节更新了 @main 异步入口点的 MainActor 隔离机制说明，补充了协议中 `async` 方法的使用指南
- 全书 API 签名更新至 Swift 6 标准，包括 `Task.sleep(for:)` 等现代写法，修正了代码示例中的命名冲突
- 新增大量实战案例，涵盖 UIKit 集成、SwiftUI @Observable 模式、`task(id:)` 修饰符等

实用性增强：

- 提供完整的 Swift 6 迁移检查清单和常见错误解决方案
- 重新组织章节结构，将分散内容整合为逻辑清晰的独立章节

版本 1.1 (2023 年 5 月)

- 修正 Swift 5.5 以来至 Swift 5.8 为止的弃用 API 和代码中的警告等，包括 `Task.sleep` 相关 API 和非 MainActor 的 UI 代码等。
- 重写《actor 模型和数据隔离》一章的部分内容，简化对 `isolated` 的讨论。
- 在《全局 actor，可重入和 Sendable》中使用 `@preconcurrency` 替换 `@_unsafeSendable` 的过时内容。
- 修正《并发线程模型》中的部分过时内容。

- 增加 Swift 并发完整编译检查的内容和 Swift 6 中强制开启完整检查的相关讨论。
- 其他的格式调整、用语修正、文本词句润色等。

版本 1.0 (2021 年 9 月)

- 初版发布。

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 (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，它们所需要的时间往往都远大于 16 毫秒。

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（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应该避免的问题之一。

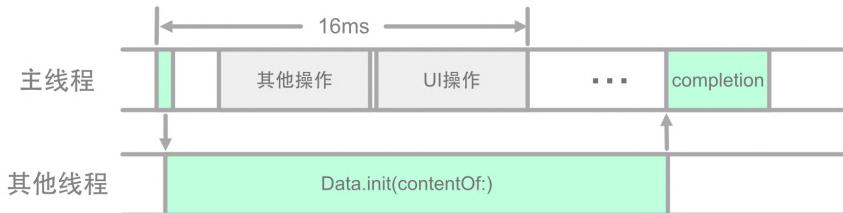
Swift 5.5 之前，要解决这个问题，最常见的做法是将耗时的同步操作转换为**异步操作**：把实际长时间执行的任务放到另外的线程（或者叫做后台线程）运行，然后在操作结束时提供运行在主线程的回调，以供 UI 操作之用：

```
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 等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，也有可能会错误地多次调用 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，只不过它现在在提交派发任务后就立即返回，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：

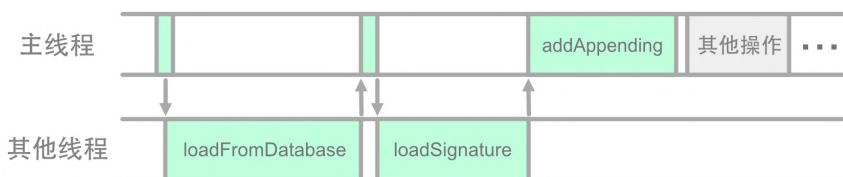
```
func loadFromDatabase(  
    completion: @escaping ([String]?, Error?) -> Void
```

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

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 =  
        loadSignature { signature, error in  
            if let 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(1. 从数据库读取 [String], 2. 从网络下载签名, 3. 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：



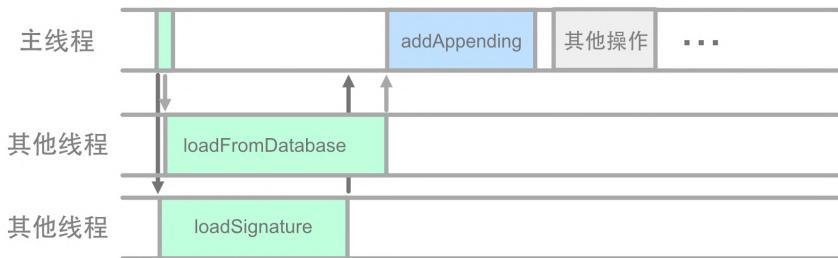
异步操作的串行。虽然图中把 `loadFromDatabase` 和 `loadSignature` 画在了同一个线程里，但事实上它们有可能是在不同线程执行的。不过在上面代码的情况下，它们的先后次序依然是严格不变的。

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（或者说，并行）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，我们还需要一种手段来确保这两个数据都可用。在 GCD 中，通常使用 `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)**。

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，其实是传统并发概念的一个子集：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和知识，才有可能正确地处理上述问题。

- 在 Swift 5.5 中，语言提供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，它们一同解决了第一个问题。
- 在此之外，Swift 利用 **actor** 模型和 **Sendable** 检查，来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，从而解决第二个问题。
- 最后，它们又组合在一起，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。

可能你对本小节提到的人名不太熟悉，但你一定听过他们的观点：戴克斯特拉发表过著名的《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。

如果我们将 `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 = {
        loadSignature { signature, error in
            if let 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，单从代码行数和缩进深度，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，线程的概念被弱化，异步函数的执行环境交由任务 (Task) 决定。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 {  
    @discardableResult  
    init(  
        priority: TaskPriority? = nil,  
       
```

```
operation: sending @escaping @isolated(any)
    () async throws -> Success
)
}
```

这里的代码示例使用了 Swift 6 的语法。如果你使用的是早期版本，某些语法细节可能有所不同，但基本概念是相同的。我们会在后续章节详细解释这些并发关键字。

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`：



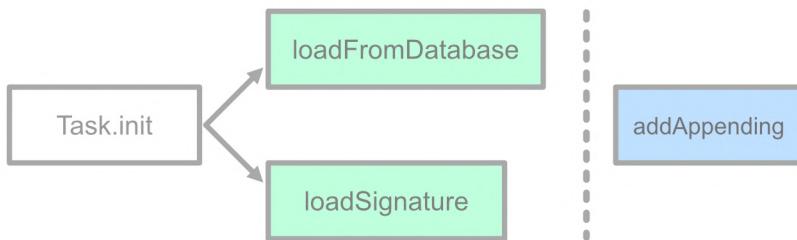
我们当然会希望这两个操作可以同时进行。在两者都准备好后，再调用 `appending` 来实际将签名附加到数据上。这需要任务以结构化的方式进行组织。使用 `async let` 绑定可以做到这一点：

```
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，并将这个子任务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，这时执行方式变为：



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

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

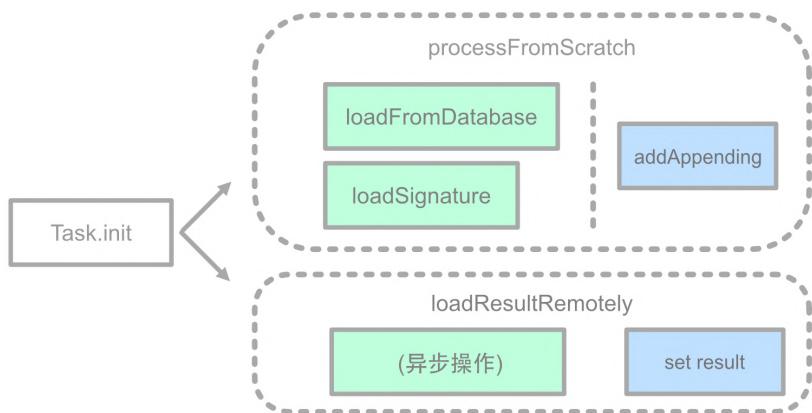
除了 `async let` 外，另一种创建结构化并发的方式，是使用任务组 (Task group)。比如，我们希望在执行 `loadResultRemotely` 的同时，让 `processFromScratch` 一起运行，可以用 `withThrowingTaskGroup` 将两个操作写在同一个 task group 中：

```
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`：

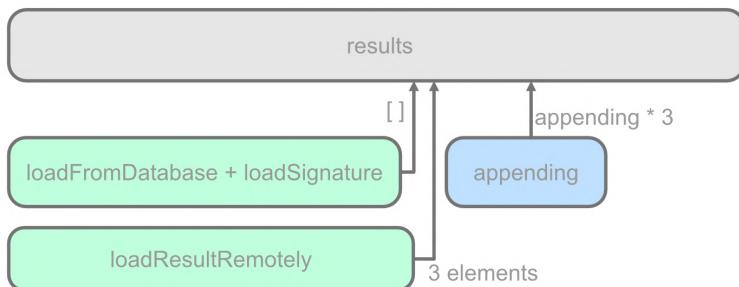
```
func loadResultRemotely() async throws {
    try await Task.sleep(for: .seconds(2))
    results = ["data1^sig", "data2^sig", "data3^sig"]
}
```

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` 设为了空数列，紧接着 `loadResultRemotely` 完成，将它设为正确的结果，然后 `processFromScratch` 中的 `forEach` 把计算得出的三个签名再添加进去。



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

`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，就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最简单的 `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。

actor 的底层实现使用了高效的协作式调度机制。为了方便理解，你现在可以将其想象为“封装了私有队列的类”，但实际机制更加精妙。我们会在后面的章节深入探讨其实现细节。

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，即便使用 actor 版本的 Holder，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：数据隔离只解决由同时访问造成的内存问题（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，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，在 Swift 的早期版本中我们无法指定隔离代码的具体运行方式。不过在 Swift 6 中，通过**自定义执行器**（Custom Executor）的支持，开发者现在可以更精确地控制 actor 的执行环境。

Swift 编译器还引入了**严格并发检查**（Strict Concurrency Checking）模式。在严格模式下，编译器能够在编译时检测出更多潜在的数据竞争问题，确保 `Sendable` 类型的正确性，并强制执行更严格的隔离规则。而且当你启用 Swift 6 语言版本后，这些问题将会以错误的形式出现。

这是一个重要的里程碑，它意味着许多在早期版本中可能在运行时才暴露的并发安全问题，现在可以在编译阶段就被发现和修复。

我们之后也还会看到包括全局 actor、非隔离标记 (nonisolated)、自定义执行器和 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，但是各模块的边界是清晰的：

- 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，有助于我们厘清 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, 并指望这段代码的使用者会乖乖去判断 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(可能你已经发现了, 比如在这里, 我们就忘记了 completion)。成功情况也没有办法阻止错误的多次调用。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 来启用这类警告)。
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设计。这很容易造成意外的阻塞, 也限制了框架使用者能做的事情。就算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。想将同步操作转变基于回调的异步操作, 可能会涉及到方法签名的变更, 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调度器，默认使用全局协作池 (Global Cooperative Pool) 进行任务调度，这为异步代码提供更多的优化空间。在 Swift 6 中还支持自定义执行器，允许更精细的控制。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，关于续体的认知会对理解异步函数的底层实现有所帮助，但是在高层级上这些知识并不必要。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 (这里为了方便阅读，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，对它们的调用都只是最简单的 apply 指令。异步函数虽然具有放弃线程的能力，但它本身并不会主动使用这个能力：它只有通过调用其他异步函数，或者通过主动创建续体，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 并没有表达其他更多的语义或是具有任何的运行时特性。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 中的函数，在作为 actor 的一部分运行时，它对外界表现为异步函数。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]
```

显然，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
}
```

这个方法在接受 `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：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  
}
```

可以转变为这样的异步函数：

```
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法：

```
var started = false
```

```
Task {
    let value = await asyncFunc(started: &started)
    print(value)
}

Task {
    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：新开启的异步任务之外的代码同时访问 started 可能会造成数据安全的问题。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。

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 with...Continuation 的全局函数，让我们暂停当前任务，并捕获当前的续体：

```
func withUnsafeContinuation<T>(
    isolation: isolated (any Actor)? = #isolation,
    _ body: (UnsafeContinuation<T, Never>) -> Void
) async -> T
```

```
func withUnsafeThrowingContinuation<T>(
    isolation: isolated (any Actor)? = #isolation,
    _ body: (UnsafeContinuation<T, Error>) → Void
) async throws → T

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

func withCheckedThrowingContinuation<T>(
    isolation: isolated (any Actor)? = #isolation,
    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。

在 Swift 6 中，续体函数新增了 isolation 参数，用于指定续体恢复时的隔离上下文。默认值 #isolation 表示继承当前的隔离上下文，这确保了更好的 actor 隔离安全性。

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 {
                continuation.resume(throwing: error)
            } else if let values {
                continuation.resume(returning: values)
            } else {
                assertionFailure("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` 必须在未来继续这件事，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（也就是说 `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
            // 故意不调用 resume, 演示续体泄漏
        }
    }
}
```

```
        }
    }
}

// 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 {
                // 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` 的第二个参数：`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，而不是帮我们直接解决这些错误。

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。

比如已经存在这样的协议：

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

想要通过异步函数进行封装，可以实现这个 WorkDelegate，在开始实际调用时，把 continuation 保存在实例变量中：

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

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

    // ...
}
```

```
}
```

最后，在相关的 delegate 方法中进行调用和清理：

```
class Worker: WorkDelegate {
    // ...

    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 框架比较熟悉，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。续体的行为模式也一样，甚至续体版本还准备了接受 `Result` 类型的重载，方便你快速对接：

```
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。

我们在下一章 `AsyncSequence` 里会继续这个话题，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，它不在标准的关键字列表里。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;
```

有了这个标记，在 Swift 中，编译器将不会为它生成异步函数的版本。这是 Apple 有意为之：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。普通的 Swift 函数，只要所有涉及到的参数或返回类型都能在 Objective-C 中表示的话，就可以通过 @objc 暴露给 Objective-C。异步函数也遵守同样的规则，而且由于 `async` 函数的返回值表意相对明确，编译器也确立了更简单的规则：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，会被导入为：

```
- (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

我们解决了将一般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 }  
}
```

我们往往倾向于不在 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 getXXX 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，然后在掉坑里后立即破口大骂。

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，编译器都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) async throws -> Attribute? {
        // 比如 await file[.readonly] == true
        get {
            let attributes = await loadAttributes()
            return attributes[attribute]
        }
    }
}
```

状态依赖

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`

可能会让代码中的其他状态“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()
            // 返回 true 真的正确吗?
            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` 之后却可能是不成立的。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，属性包装器 (`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。如果我们更进一步，希望表达的不仅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，则需要用到一种新的表达方式，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。每次暂停异步函数后，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交回给当前异步函数。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) 来规定它们的一些特性。这个协议就是异步序列 AsyncSequence。异步序列在 Apple 各种 SDK 中都有体现，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

在深入异步序列之前，我们先回顾一下同步世界的序列是如何工作的。

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 事实上做的事情只有一件，那就是用 makeIterator 方法来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 也很简单，它只有一个 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：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。这个序列虽然最终会溢出 Int 范围，但它永远不会返回 nil：

```
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。如果这个计算过程非常耗时，我们会希望能把 next 写为异步函数。这样一来，在获取序列中下一个元素时，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。你可能已经猜到了，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：

```
public 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...
    try await Task.sleep(for: .seconds(1))
    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，其主要目的是为了让开发者也能用类似的 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，想要读者进行思考：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？（对于斐波那契数列..5, 8, 13, 21, 34 ..，当迭代到 21 时，上面的代码会跳出第一个 for await 循环，所以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，从中断处再次进行 for 的时候，应该从之前的中断处继续，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 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, flatMap, 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

上面提到的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` 类型它们都适用，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。以 `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)，而不是另一个 `Sequence`，这和异步序列的情况是不同的。`Array` 确实满足 `Sequence` 协议，但它只是一种特殊的序列：数组中的元素数量是有限的。`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 }
```

在实际执行中，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 的类型是 LazySequence<FibonacciSequence>

let mapped = lazySeq.map { $0 }
// mapped 的类型是
// 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，除了 transform 参数的 async 以外，它们的签名十分相似：

```
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返回额外的映射值，也不会改变原序列，它只是一个单纯的副作用 (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和标注  
struct AsyncStream<Element> {  
    init(  
        _ elementType: Element.Type = Element.self,  
        bufferingPolicy limit: BufferingPolicy = .unbounded,  
        _ build: (AsyncStream<Element>.Continuation) -> Void  
    )  
  
    struct Continuation {  
        func yield(_ value: sending 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 {
            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()
        timer.invalidate()
    }
}

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

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)
    }
}
try await Task.sleep(for: .seconds(2))
```

```
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，我们就需要通过这个参数来创建 AsyncStream 并确定它的类型：

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

除了帮助推断类型之外，它并没有更多的实际用处。

bufferingPolicy 则实打实地影响 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
try await Task.sleep(for: .seconds(5))
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：因为设备总有内存极限，当可能需要缓冲的数据量没有上限时，这种策略就有可能出现问题。另外，在一些情况下，也许迭代速度会比读取缓冲的速度更慢，这也可能让缓冲区中堆积大量的数据，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上新的地址。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，前者的速度可能远远大于涉及到 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。

除了默认的 .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)  
}
```

顾名思义，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: sending Element) → YieldResult

enum YieldResult {
    // 写入队列，并返回告知缓冲区的剩余大小
    case enqueue(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，并添加额外的控制。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))
try await Task.sleep(for: .seconds(5))
for await v in timer {
    print(v)
}
print("Done")
```

输出为：

```
[07:48:29] Call yield: enqueue(remaining: 2)
[07:48:30] Call yield: enqueue(remaining: 1)
[07:48:31] Call yield: enqueue(remaining: 0)
[07:48:32] Call yield: dropped(2021-07-21 07:48:29 +0000)
[07:48:33] 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: enqueueed(remaining: 3)
2021-07-21 07:48:34 +0000
Call yield: enqueueed(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，对用户来说可能是没有意义的。因此使用上面的 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“丢弃数据”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产生端的行为，叫做背压 (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> {  
    try? await Task.sleep(for: .seconds(1))  
    return Date()  
} onCancel: { @Sendable in  
    print("Cancelled.")  
}  
  
for await v in timer {  
    print(v)  
}
```

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，而是每次被要求时才产生新的值。

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，来表达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进行迭代，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（不止是 AsyncStream，也包括更一般性的 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。

传统上，Swift 函数在遇到错误时都是使用 throw 来进行的，抛出的错误会被类型擦除为 any Error。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，但在 Swift 6 之前，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。

类型化抛出 (Typed Throws)

Swift 6 引入了 [SE-0413](#) 类型化抛出，允许函数声明它们只会抛出特定类型的错误。这个特性使用 throws(ErrorType) 语法：

```
// 声明只会抛出 MyError 类型的函数
func process() throws(MyError) {
    throw MyError.invalidInput
}

// 在 catch 中可以直接处理具体类型
do {
    try process()
} catch {
    // error 的类型是 MyError，而非 any Error
    switch error {
        case .invalidInput:
            print("Invalid input")
        case .networkFailure:
            print("Network error")
    }
}
```

对于异步序列，我们现在可以创建具有精确错误类型的异步迭代器：

```
struct TypedAsyncIterator: AsyncIteratorProtocol {
    typealias Element = Int
```

```
mutating func next() async throws(MyError) -> Int? {  
    // 只能抛出 MyError 类型的错误  
    guard someCondition else {  
        throw MyError.invalidState  
    }  
    return value  
}  
}
```

使用类型化抛出的异步序列时，错误处理变得更加精确：

```
// 假设我们有一个支持类型化抛出的异步序列  
let typedSequence: some AsyncSequence<Int, MyError>  
  
do {  
    for try await value in typedSequence {  
        print(value)  
    }  
} catch {  
    // error 自动推断为 MyError 类型  
    // 无需类型转换即可访问具体错误  
    handleMyError(error)  
}
```

选择合适的错误处理方式

虽然类型化抛出提供了更精确的错误处理能力，但它并不适合所有场景。在以下情况下，类型化抛出特别有用：

1. **错误条件相对固定**: 当错误类型不太可能随着 API 演进而改变时
2. **性能敏感场景**: 避免了 any Error 的存在类型开销
3. **需要精确错误处理**: 客户端代码需要详尽处理所有可能的错误

而在以下情况下，继续使用非类型化的 throws 可能更合适：

1. API 需要灵活演进：未来可能增加新的错误类型
2. 错误来源多样：函数可能传播来自多个不同源的错误
3. 作为公共 API：避免强制客户端处理具体错误类型

对于异步序列而言，标准库的 AsyncThrowingStream 仍然使用非类型化的错误处理，这保持了最大的灵活性。但在特定领域的实现中，使用类型化抛出可以提供更好的类型安全性和性能。

关于 throw 是否应该总是使用强类型错误，社区的共识是：类型化抛出是一个强大的工具，但应该谨慎使用。默认的非类型化 throws 仍然是大多数 Swift 代码的更好选择，而类型化抛出应该在确实需要其优势的特定场景中使用。

调度和执行

在执行方式和时间维度上，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。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：我们可以通过 Swift 6 中开始支持的自定义执行器来设定执行环境，但它需要我们付出额外努力。

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，笔者认为都不是合理的做法。按照使用场景，选择正确的工具，是更好的做法。

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，我们接下来会看到很多使用异步序列的系统级 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，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。在本章中我们对它们进行介绍。

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 {  
        print("Error: \(error)")  
        return  
    }  
  
    if let 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 dataTask = session.dataTask(with: url)  
dataTask.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 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

除了在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` 是在 Objective-C 中定义的代理协议，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。在本书后面涉及迁移到 Swift 并发代码的时候，我们还会看到更多的在 protocol 中使用 `async` 的例子。

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 来将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 throws -> Bool {
    try await Task.sleep(for: .seconds(1))
    return true
}

func syncMethod() throws {
    Task {
```

```
    try await asyncMethod()  
}  
}  
}
```

这个异步闭包的返回值会作为 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，可以通过 await 来等待这个任务完成，并访问它的 value 属性来获取任务结束后的“返回值”：

```
func anotherAsyncMethod() async throws {  
    let task = Task {  
        try await asyncMethod()  
    }  
    let result = try 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 throws {  
        try await Task.sleep(for: .seconds(1))  
        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 { try await MyApp.main() }  
}
```

`_runAsyncMain` 的实现是开源在 Swift 项目仓库中的，它被执行时将使用`Task.detached` 创建一个异步运行环境。根据 SE-0323 提案，`@main` 标记的异步函数会自动隔离到`MainActor`，确保在主线程上运行。这提供了对 UI 操作的安全访问，同时保证了程序初始化的正确顺序。

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` 修饰符：

```
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(for: .seconds(1))
    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 修饰符，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 修饰符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，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(for:) 抛出 CancellationError，导致 try? await load() 的结果为 nil。

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 ProgressView 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。

SwiftUI 中的其他异步支持：除了上面提到的 task 修饰符，SwiftUI 还提供了一个带 id 的变体版本 task(id:)，当 id 参数变化时，会自动取消之前的任务并启动新任务。这对于响应用户输入或状态变化非常有用。另外，配合 @Observable 宏 (iOS 17+)，可以更轻松地在异步操作中更新 UI 状态，而无需手动切换到主线程。

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(for:) 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：比如取消网络请求，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) 才是解决这个问题的关键，它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。

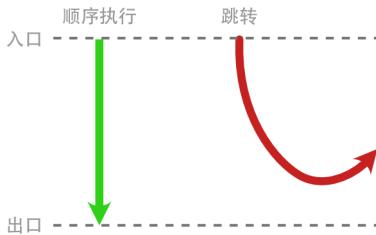
什么是结构化

“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

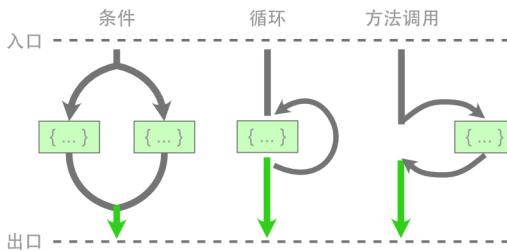
goto 语句是非结构化的，它允许控制流无条件地跳转到某个标签。虽然现在看来 goto 语句已经彻底失败，完全不得人心，但是受限于编程语言的发展，goto 语句在当时是有其生存土壤的。在还没有发明代码块的概念 (也就是 { ... }) 之前，基于顺序执行和跳转的控制流，不仅是最简单的天然选择，也完美契合 CPU 基于程序计数器 (Program Counter) 的指令执行方式。顺序执行的语句非常简单，它总可以找到明确的执行入口和出口，但是跳转语句就不一定了：



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到出口。

这不仅让代码的思维模型变得更简单，也为编译器在低层级进行优化提供了可能。更重要的是，这种结构化的特性与基于栈的内存管理模式形成了天然的协同关系：结构化控制流的嵌套特性与栈的后进先出（LIFO）特性完美匹配，使得函数调用时压栈、返回时出栈，以及局部变量在进入作用域时分配、离开作用域时自动释放的模式变得既直观又高效。

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，编译器可以安全地按照相反的分配顺序释放这些资源。这一点对于自动内存管理和资源回收（比如在 `defer` 中关闭文件、切断网络等）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，在上述语境中的程序结构化，往往指的只是同步程序的结构化，而并发往往是非结构化的。旧版本中 Swift 的并发模型面临的问题，恰恰和当年 `goto` 的情况类似。旧版本中 Swift 所使用的并发手段，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。我们来看一个例子：

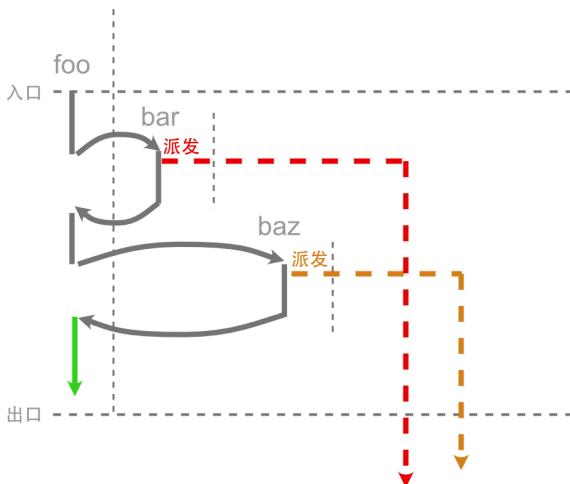
```
func foo() -> Bool {
    bar(completion: { print($0) })
    baz(completion: { print($0) })

    return true
}

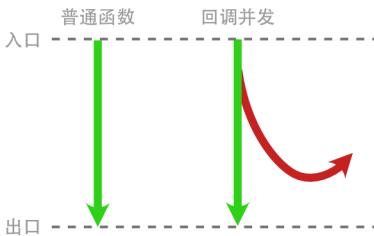
func bar(completion: @escaping (Int) -> Void) {
    DispatchQueue.global().async {
        // ...
        completion(1)
```

```
        }  
    }  
  
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。粗心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将它丑陋的面貌进行了一定程度的遮掩。

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称心如意？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，都会存在这些担忧：

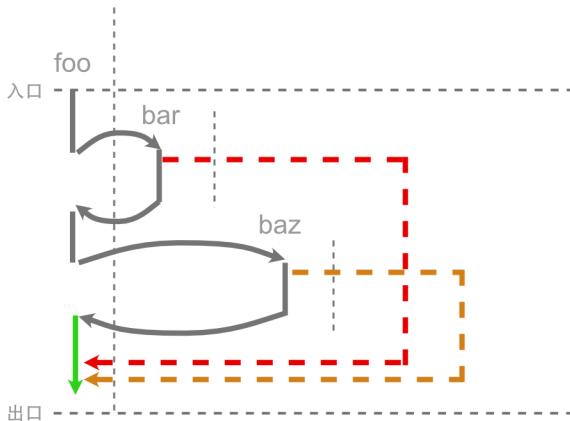
- 这个函数会不会产生一个后台任务？
- 这个函数虽然返回了，但是它所产生的后台任务可能还在运行。它什么时候会结束，它结束后会产生什么样的行为？
- 作为调用者，我应该在哪里、以怎样的方式处理回调？这个回调会发生在哪个线程上？
- 我需要保持这个函数用到的资源吗？后台任务会自动去持有这些资源吗？我需要自己去释放它们吗？我是不是应该要使用 weak 来确保不发生泄漏？
- 后台任务是否可以被管理，比如想要取消的话应该怎么做？
- 派发出去的任务会不会再去派发别的任务？别的这些任务会被正确管理吗？如果取消了这个派发出去的任务，那些被二次派发的任务也会被正确取消吗？

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！）然后不断试错和猜测。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不会逃逸的状态，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，控制流最终一定会回到掌握之中。

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，我们总可以使用 `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。如果当前的这个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。比如 `Task.init` 或者 `Task.detached`。
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。我们也可以调用 `cancel()` 来主动取消这个任务。

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：因为同步函数也可以在某个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：因为在这里它是被 `foo` 调用的，而想要调用 `foo` 这个异步函数，必定需要一个任务环境。

```
func foo() async {
    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。你不应该存储这个值，也不应该在闭包之外调用或访问它的属性和方法，这都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“简便方法”：直接使用 `Task` 的静态属性，就可以获取当前任务的状态，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，需要让内层的 t1 和 t2 与外层 Task 具有某种从属关系。你可能已经猜到了，如果我们将外层任务看作根节点，把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子任务节点，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)")  
try? await Task.sleep(for: .seconds(Double(value)))  
print("Work \$value done")  
return value  
}  
}
```

这段代码在 start 中一开始，就使用了 withTaskGroup 开启一个新的任务组。这个方法完整的函数签名是：

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

1. 该签名看起来十分复杂，有点吓人，我们来详细解释一下：

- 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。
- isolation 指定了隔离域。默认的参数是一个宏 (#isolation)，它代表使用当前任务所在的隔离域作为运行环境。我们会在本书后面对它进行详细介绍。
- 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。

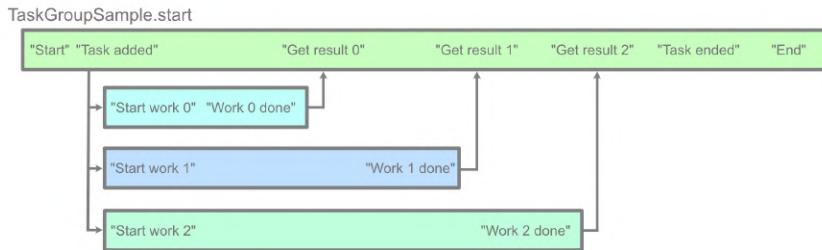
2. addTask API 把新的任务添加到当前任务中。被添加的任务会在调度器获取到可用资源后立即开始执行。在这里的例子里，for...in 循环中的三个任务会被立即添加到任务组里，并开始执行。

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，它一定会在所有子任务完成之后才发生。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()
}
```

任务组的错误处理

在使用 withTaskGroup 时，如果不需要处理错误，子任务应该捕获并处理所有可能的错误。如果需要错误传播，应该使用 withThrowingTaskGroup。当任务组中的某个子任务抛出错误时：

1. 该错误会立即传播到任务组的调用者
2. 任务组会自动取消所有其他正在运行的子任务
3. 任务组会等待所有子任务完成（包括被取消的任务）后才真正结束

```
do {
    try await withThrowingTaskGroup(of: Int.self) { group in
```

```
group.addTask {  
    try await riskyWork(0)  
}  
group.addTask {  
    try await riskyWork(1)  
}  
  
// 如果任一子任务抛出错误，会直接被外层 catch  
for try await result in group {  
    print("Result: \(result)")  
}  
}  
}  
} catch {  
    print("Task group failed: \(error)")  
}
```

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某个值是危险操作，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，如果我们在任务组里操作了 value，那即使只是读取这个 var value 值，也是不被允许的：

```
let v = await withTaskGroup(of: Int.self) { group in
    var value = 0
    for i in 0 ..< 3 {
        group.addTask {
            print("Value \(value)") // 加入这一句
            return await work(i)
        }
    }
    for await result in group {
        value += result
    }
    return value
}

print("End. Result: \(v)")
```

将在 addTask 处给出错误：

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。如果必须在 addTask 中使用 value 的值，可以尝试用 [value] 的语法，来捕获当前的 value。由于 value 是值类型的值，因此它将会遵循值语义，将 addTask 被调用时的当前值复制一份到 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
}
}

// ...
}
```

类似使用 `[value]` 捕获值的做法，实质上是把它“变成”了一个不变值，或者说满足 `Sendable` 的值，从而满足数据安全的要求。我们会在后面的章节仔细讨论 `Sendable` 特性。现在，你可以简单地将它理解为只有“线程安全”的变量能在 `addTask` 闭包中使用。

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` 添加的任务会破坏结构化并发的核心保证：当 `withTaskGroup` 闭包结束时，任务组已经完成了等待所有子任务的过程。在此之后尝试添加新任务是不受支持的操作，会导致不可预测的行为和潜在的内存安全问题。

`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。这个类型仅仅是作为对底层任务管理接口的包装，设计目的是在其作用域内提供创建和管理子任务的方法。任务组的生命周期管理是结构化并发的基础设计约束，而非实现缺陷。

虽然当前编译器的类型系统还无法静态地阻止 `group` 被复制到闭包外部，但这是 Swift 团队已经认识到的限制。未来可能会通过非逃逸类型（non-escapable types）等语言特性来在编译时防止这种错误使用。在此之前，我们必须避免这种违反设计原则的反模式，始终确保所有任务操作都在 `withTaskGroup` 的作用域内完成。

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、按部就班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域和返回路径，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 `for await` 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，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，它们的返回值也必须是同样的类型，这让灵活性下降，而且可能会要求更多的额外实现（比如需要支持不同返回类型时，只能将各个任务的返回值用新类型进行封装等）。`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，最后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。

上面 `withTaskGroup` 的例子中的代码，使用 `async let` 可以改写为以下形式：

```
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
```

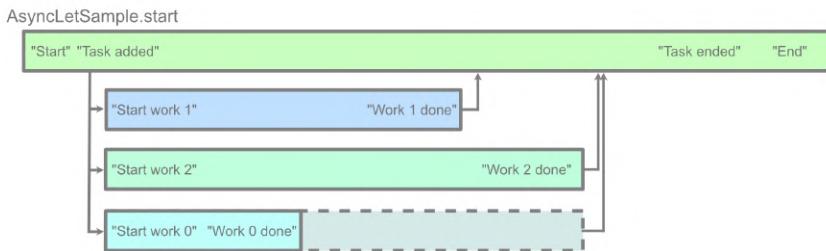
需要特别强调，虽然这里我们顺次进行了 await，看起来好像是在等 v0 求值完毕后，再开始 v1 的暂停；然后在 v1 求值后再开始 v2。但是实际上，在 async let 声明时，这些子任务就一同开始以并发的方式进行了。在例子中，完成 work(n) 的耗时为 n 秒，所以上面的写法将在第 0 秒，第 1 秒和第 2 秒分别得出 v0，v1 和 v2 的值，而不是在第 0 秒，第 1 秒和第 3 秒（1 秒 + 2 秒）后才得到对应值。

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`，两种方式所生成的子任务都是结构化的。不过，其中还有些许差别，我们马上就会谈到这个话题。

async let 的生命周期和隐式行为

`async let` 创建的子任务具有明确的生命周期管理规则：

1. 任务创建：在 `async let` 声明时立即创建并开始执行子任务

2. 结果获取：通过 await 显式等待并获取结果
3. 作用域结束：当绑定的常量离开作用域时，会触发隐式行为

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，编译器也帮我们动了某些“手脚”，以保证单进单出的结构化并发依然成立。

对于没有被 `await` 的 `async let` 绑定，Swift 会在作用域结束时执行以下操作：

- 首先取消（cancel）子任务
- 然后隐式地 `await` 等待任务结束
- 这确保了结构化并发的单入单出特性

这种设计让未使用的异步操作能够尽早停止，避免资源浪费。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) {

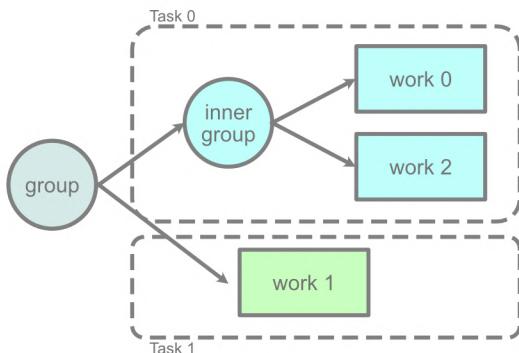
```

```
    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，也不会有什么大问题，甚至结构可能会更清楚一些。

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，两者混用也不会产生什么问题。

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，一旦看到两者，我们就应该意识到程序进入了异步结构化的范畴中。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。在被调用时，它将会生成一棵位于顶层的新任务树，它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。

各种创建任务的方式之间存在着明显的不同：

- 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。

非结构化任务的使用场景

选择使用 Task.init 还是 Task.detached 需要根据具体场景：

使用 Task.init 的场景：

- 需要保持当前的 actor 隔离域（如在 MainActor 中启动的任务）
- 希望继承当前任务的优先级
- 需要访问任务本地值（Task Local Values）

→ 处理与当前上下文相关的操作

使用 Task.detached 的场景：

- 需要完全独立的任务环境，不受当前上下文影响
- 执行后台操作，不应该继承高优先级
- 避免隐式的 actor 隔离域继承（如从 MainActor 启动长时间运行的后台任务）
- 缓存、日志记录等非关键操作

需要注意的是，Task.detached 创建的任务默认运行在全局并发池中，不会继承任何 actor 隔离域。这在某些情况下是必要的，比如避免长时间占用 MainActor。

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 值即使超过作用域，也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，也不会传递到内层 Task。或者，更准确来说，这样的两个 Task 虽然写成了嵌套的结构，但实际上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。此时使用独立任务可能会更合适。

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，而舍弃掉非结构化并发模型呢？现代语言中有这个趋势，但是大部分也都还保留了原来的传统并发模型。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下，对结构化并发提供支持的语言之一。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。

在基于回调的传统派发式并发模型中，取消任务是一件非常困难的事情。由于并发任务可能会逃逸出当前作用范围，而且并发任务之间也缺乏关联，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（比如说在 GCD 模型下的 DispatchWorkItem），才能在适当的时候将它们停止。这其中涉及的复杂度，其实只在理论上可行，实践中很难做到不出现各种 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。

在传统模型里，不论是使用纯 GCD 还是 Operation 类型，想在并发编程中正确处理取消操作，都面临着巨大的挑战。

而有了结构化并发后，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。对于父任务的取消，可以非常容易地传递到作用域内的子任务中，这样子任务就可以及时地对取消作出响应，进行清理资源等操作。更好的地方在于，这个传递甚至是完全自动的。

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也可以被“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：所谓“协作”，指的是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，才能共同达到我们最终想要的效果。结构化并发可以帮助在任务间传递取消信号，但这仅仅只是协作式取消中的一个部分，任务的具体实现中也必须包含对取消操作的处理，整个取消系统才能运作。

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(seconds: 1.0)
        print("Append: \(c)")
        s.append(c)
    }

    return s
}
```

这里使用了一个我们自定义的 `Task.sleep(seconds:)` 来模拟耗时操作。和我们在之前几章中使用的 `Task.sleep(nanoseconds:)` 以及 `Task.sleep(for:)` 不同，这个方法是我们自定义的对 GCD 的简单封装，它内部使用了 `asyncAfter`，因此没有对 Swift 并发的取消操作做任何支持：

```
extension Task where Success == Never, Failure == Never {
    static func sleep(seconds: TimeInterval) async {
        await withUnsafeContinuation { continuation in
            DispatchQueue.main.asyncAfter(deadline: .now() + seconds) {
                continuation.resume()
            }
        }
    }
}
```

使用这个不支持取消的 `Task.sleep`，可以让我们更清楚地认识到取消操作在 Swift 并发中的行为和意义。

现在创建一个任务，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(seconds: 2.5)
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: \$(c), cancelled: \$(Task.isCancelled)")

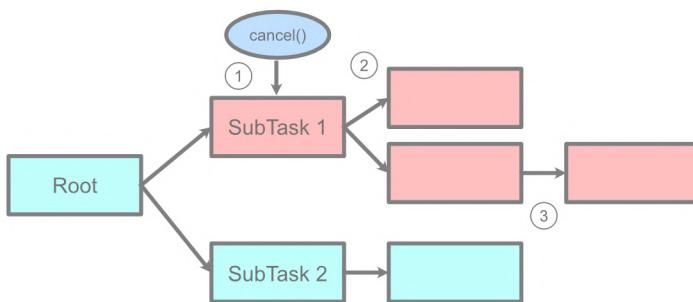
// 输出：
// Append: H, cancelled: false
// Append: e, cancelled: false
// Append: l, cancelled: true
// Append: l, cancelled: true
// Append: o, cancelled: true
// Hello
```

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，做的事情非常单纯，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，同样也只做这两件事。因此，在结构化并发中，取消操作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：因为我们基于 GCD 的 sleep 根本没有对取消进行任何响应。

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(seconds: 1.0)  
        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(seconds: 2.5)
    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，其重要程度并不是相等的。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 {
        if Task.isCancelled {
            print("Cancelled: \(text)")
        }

        try Task.checkCancellation()
        await Task.sleep(seconds: 1.0)
        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(seconds: 2.5)
                        inner.cancelAll()

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

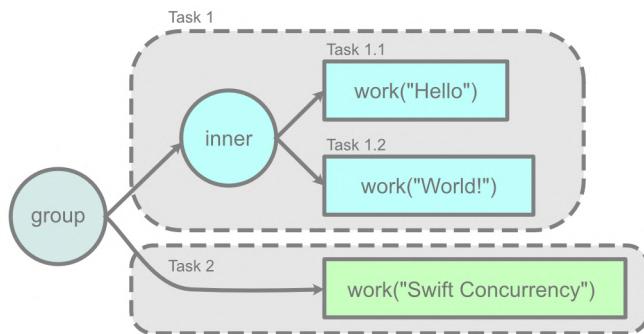
    // Task 2
    group.addTask {
```

```
try await work("Swift Concurrency")
}

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

print(value)
} catch {
    print(error)
}
```

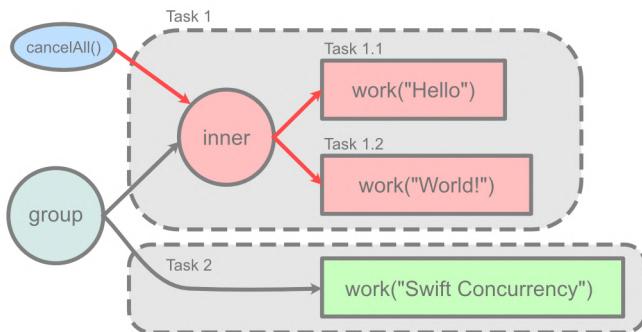
这段代码有点复杂，不过用任务层级树的方式表达，就可以明了一些了，它等效于：



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`。

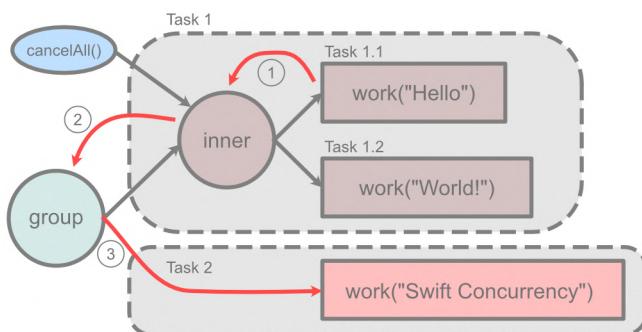
这几个任务是同时开始的，如果不加干涉，这些代码最终会按照 `Task 1.1`, `Task 1.2`, `Task 2` 的顺序完成，最终输出 “Hello World! Swift Concurrency”。不过在 2.5 秒时，我们手动调用了

`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：

1. 尽可能快地对任务取消作出响应，避免额外的非必要工作；
2. 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。虽然从运行结果上来说还是对的，但这种行为完全是对资源的浪费，有时候也是违反直觉的（比如本例就表现为取消操作耗时非常久）。

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，我们“自制”的 Task.sleep(seconds:) 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 中提供的其他 sleep 版本几乎都是可以取消的，它们并被标记为 throws：

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

    static func sleep<C>(
        for duration: C.Instant.Duration ...
    ) async throws
}
```

在遇到取消时，标准库中的 `sleep` 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(seconds: 1.0)

        try await Task.sleep(for: .seconds(1.0))
        s.append(c)
        // ...
    }

    // 输出为：
    // Append: S
    // Append: H
    // Append: W
    // Append: w
    // Append: o
    // Append: e
    // CancellationError()
```

对比我们自定义的实现中，每次在 `await` 前都要进行检查，标准库的相关方法抛出错误更加及时，它不需要等到当前的 `await` 结束后就能进行抛出。相比于原来的处理取消的方式，标准库里提供了更优秀的实现。

在我们实际构建一个真正的并发系统（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")!
        )
        print(data.count)
    } catch {
        print(error)
    }
}

try await Task.sleep(nanoseconds: 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(seconds: 1.0)
    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: () -> Void,  
    isolation: isolated (any Actor)? = #isolation  
) 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。

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() }

            // 离开 task group 作用域
        }
    } catch {
        print("Error: \(error)")
    }
}

await Task.sleep(seconds: 1.0)
t.cancel()

func work() async throws -> Int {
    try await Task.sleep(for: .seconds(3))
```

```
    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() 只生存在 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，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做到的。

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 甚至将 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

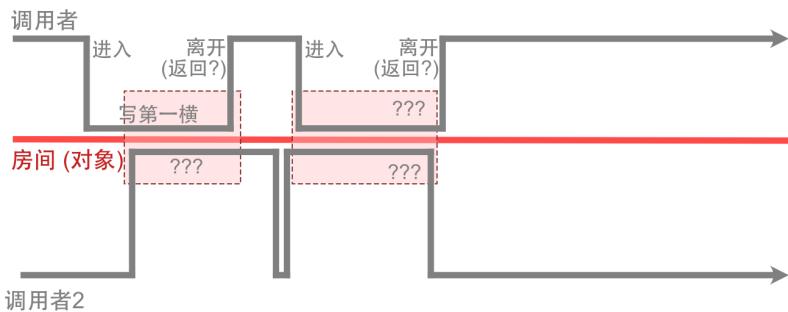
对于单线程的程序，同一时间内只会有一段代码在执行，对内存中状态的访问和改变都是独占发生的。在“远古时期”，程序的执行相对简单，也没有那么多并发的需求，因此使用面向对象编程 (OOP) 模式对事物进行抽象是合理的。

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，并将结果带回，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（在 Apple 平台下，一般使用 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，而后一个调用者“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（内存状态错误）。



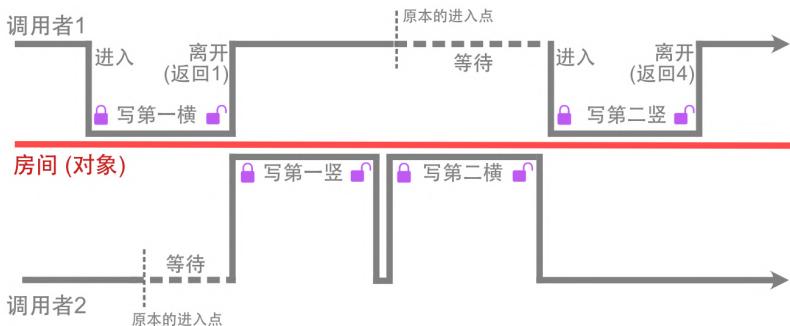
两个调用者同时访问房间的重叠时间，可能造成内部状态的错误，程序在这种情况下的行为是难以预测的，崩溃或者继续以错误的状态值运行都有可能。另外，如果调用者来访的时间不发生重叠，其实就不会出现问题。在开发期间，由于测试样本太小，经常导致侥幸没有访问重叠，所以这类问题难以发现。但是在实际发布版本中，却会由于用户量上升而变得无法忽视，但又难以调试。

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。在这个模型中，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，他就把锁给锁上，直到离开房间再打开。这样，可以保证房间内只会有一人，而在纸上写“正”字的下一笔这个操作，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，这严重地限制了程序的性能：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，很可能出现死锁（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。

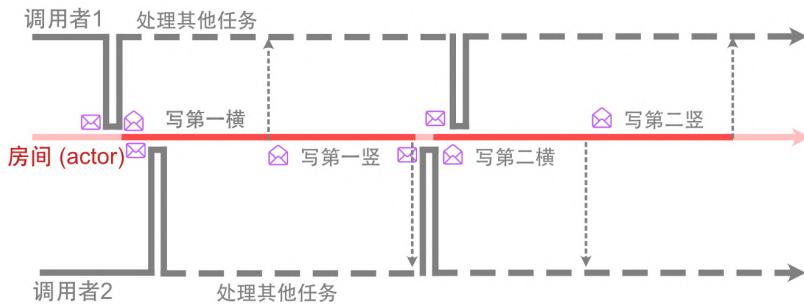
Actor 隔离

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，而不做写字的耗时操作，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，也就是说，两个同时来投递信件的人不能互相撞得满头包。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，默认情况下都是被 actor 隔离的。隔离域对于 actor 自身来说是透明的：在同一个域隔离中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方法中可以使用：

```
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。

编译器的隔离检查机制相当精妙。首先，它会识别出在同一个 actor 隔离域内的代码，这些代码可以自由地同步访问该域内的所有成员。其次，当检测到从隔离域外访问隔离域内的成员时，编译器会强制要求使用异步调用。最后，编译器还会根据调用的上下文来推断访问是否真的跨越了隔离域边界。

因为 actor 要保证隔离状态不被意外改变，因此对于这些跨域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。这个机制的巧妙之处在

于，它确保了同一时刻只有一个任务能够访问 actor 的可变状态，而且所有的跨域访问都是显式的，开发者能够清楚地看到代码中的每一个潜在暂停点。更重要的是，数据竞争的问题在编译时就被彻底防止了，而不是等到运行时才暴露出来。

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，每个跨域调用都会被封装成一个消息，加入到 actor 的队列中。每个 actor 实例都包含了它自己的串行执行器，这个执行器负责确保消息按照顺序处理。而 actor 的方法则在其专属的隔离上下文中执行，这就从根本上保证了内存安全。

从概念上，这和将操作都封装到一个串行派发的 DispatchQueue 中类似，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。这种实现相比传统方式有着明显的优势：它不需要传统的线程阻塞和唤醒机制，从而大大降低了上下文切换的开销；线程可以在等待期间去处理其他任务，系统资源得到了更好的利用；而且运行时可以智能地在可用线程间分配工作，实现了自动的负载均衡。

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 的初始化方法中创建一个执行器，并把它绑定到这个 actor 实例中去。

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 里的协议，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”，所以即使在 protocol 中被声明为了 get async，普通类型还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: any 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：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` 这个引用本身确实是完全受到保护的，问题在于 `Person` 类型是 `class`，它无法能够保护它上面 `name` 成员，这和 `Room` 这个 actor 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`。

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 之外的声明，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。在一些需要隔离的全局的函数中，也许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`，将把这个函数放到参数 actor 实例（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 anotherRoom = Room()  
        // anotherRoom 不在 `self` 隔离域中。需要切换隔离域。  
        await reportRoom(room: anotherRoom)  
    }  
}
```

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 跳跃相比传统的线程切换是轻量级的操作，但并非没有开销。实际的性能影响取决于多个因素。如果跳跃非常频繁，累积的开销会变得显著；在系统高负载的情况下，可能需要真正的线程切换；而如果任务本身很小，跳跃开销在整体时间中的占比就会更高。

大部分情况下 actor 跳跃会在同一个线程中通过协作式调度完成，但我们不应该对实际的执行线程进行假设。Swift 运行时会根据系统状态智能地决定如何调度任务。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在讨论这个话题。

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
}
```

在本书最初写作时，上面的代码是可以正常编译和运行的。后续的 Swift 5.7 版本中，这段代码会产生警告，表示方法内隔离域冲突。这样的写法预计会在 Swift 6 时被编译器彻底判定为错误。因为有编译器的帮助，实际上我们几乎不会遇到同时需要两个隔离域的情况，在这里我们就不再深入讨论了。

如果你对这个话题和它的历史原因感兴趣，可以参看 [Swift 并发路线图](#) 中的相关部分，actor 的完全隔离将作为 Swift 的一个中期目标，而在当前阶段，语言只提供部分的 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，“原则上”应该是按照 `self` 进行 actor 隔离的，但同时它又被声明了参数隔离，被明确表示需要被隔离在参数的域中。此时，Swift 将选择尊重更明确的表述，也就是使用参数定义的隔离域来运行代码：

```
extension Room {
    // 使用 self 隔离
    func baz1(_ another: Room) async {
        print(self.visitorCount)
        print(await another.visitorCount)
        // ...
    }

    // 使用 another 隔离
    func baz2(_ another: isolated Room) async {
        print(await visitorCount)
        print(another.visitorCount)
        // ...
    }
}
```

乍看起来，为 Room.baz2 指定 another 作为隔离域的方式似乎没什么太大意义，我们完全可以直接调用 await another.visitorCount，它不会造成结果上的区别，但是这种方式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)  
        }  
        print(self.visitorCount)  
    }  
}
```

如果在默认的 self 隔离域中，每次对 another 上成员的访问都会产生两次 actor 跳跃：先从 self 域切换到 another 域，然后在 await 结束后再切回 self 域。

actor 跳跃的性能开销并不是可以忽略的。每次跳跃都需要将任务从一个执行器的队列转移到另一个，这涉及到保存当前的执行状态，并在返回时恢复它。虽然大多数情况下这些操作都能在同一线程内完成，但在系统负载较高时，可能会触发真正的线程切换，带来更大的开销。

这种情况下，用 another 去隔离，可以有效减少 actor 跳越的次数。在处理大量跨 actor 操作时，这种优化可以带来可观的性能提升。不过，需要权衡代码清晰度和性能收益，只有在性能关键路径上才建议使用这种技巧。我们在下一章里会看到如何实现这一点。

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。

关于 isolated 还有一些其他话题没有涉及，特别是 Swift 6 等最新版本中对于相关概念的强化，提供了更细粒度的控制以及更好的默认性能。关于这些内容，我们会在下一章中继续讨论。

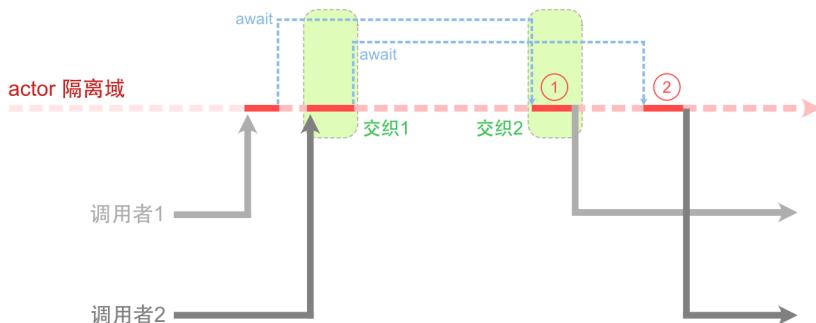
Actor 可重入

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

与其他类型一样，actor 中的方法也可以使用异步函数。在调用异步函数时，当前的方法有可能因为要处理其他任务而处于暂停状态。对于依赖串行调度来保证数据安全的 actor 来说，在这种情况下将面临一个选择：是否允许在一个方法暂停时访问 actor 上的其他成员。

Swift 允许这种访问，并将其称为可重入特性。可重入特性允许在被暂停的 actor 隔离函数继续执行之前，其他任务可以在同一个 actor 上执行（包括其他人再次调用正被暂停的函数）。当一个函数在等待时，另一个函数占用了隔离域并且修改了 actor 上的属性时，我们说这些访问之间发生了“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，这个行为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 隔离域上的属性可以安全且同步地进行修改。假设调用者 1 在 await 前后都依赖于某个 actor 的属性 foo，则需要特别注意，这个属性在第一次交织期间可能会被调用者 2 修改。因此，当 await 结束时，调用者 1 在第二次交织中（在图中标记为交织 2）再次读取 foo 值时，它可能与初始值不同。同样地，经过第二次交织后，当调用者 2 回到自己的执行流程（在图中表示为 actor 隔离域的最后一段访问）时，actor 的状态也可能与进入 actor 时的初始状态不同。因此，即使访问的是同一个变量，我们也不能轻易地假设在 await 前后其值将保持相同。

可重入的风险

可重入特性虽然提升了并发性能，但也引入了一类微妙的问题。当 actor 方法在 await 处暂停时，其他任务可以修改 actor 的内部状态。这种状态变化发生在我们的代码“背后”，如果处理不当，可能导致逻辑错误。

举一个更具体的例子，比如我们之前看到的 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：它的功能是检查 visitorCount，当访问次数较少时，它会调用一个外部的异步方法 analyze(room:)（比如较慢的 AI 分析或者网络请求）来进行原因分析，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(for: .seconds(1))
    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。例如，只有一次访问的情况：

```
let room = Room()
Task {
    _ = await room.visit()
    let r = await room.generateReport()
    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.generateReport()
    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，但生成的报告中可能显示含有 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，即使发生交织导致 `await` 后的实例成员状态和我们“预想”的有所不同，`actor` 隔离域本身仍然是有效且正确的。在 `actor` 隔离域中，两次函数调用仍然是串行执行的，不会发生对同一状态的同时读写或内存危险。然而，是否发生这种交织行为取决于调用的时间，所以从外部观察者的角度来看，部分行为可能会表现出类似数据竞争的特征。

编译器中要求在每个暂停点标记 `await` 的一个主要原因也是为了能强调可能发生的交织现象。在 `actor` 中，我们必须格外小心地处理每个 `await`，因为在 `await` 前后的执行环境里的值可能完全不同，实例的状态也可能发生改变。当我们在 `await` 之前和之后都依赖了某个状态和假设时，我们必须时刻清晰地意识到这一点。

处理可重入的实用模式

虽然可重入带来了挑战，但通过合理的设计模式，我们可以有效地管理这些风险：

1. 状态快照

为了解决上面 generateReport 的问题，我们可以在 await 之前复制所需的值，并依赖这个本地的不变量。通过这种方式，我们可以保持依赖的值不受交织影响，确保代码的正确性：

```
func generateReport() async -> Report? {
    if !isPopular {
        // 在 await 前复制一份 visitorCount
        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。

2. 状态验证

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
}
```

这种方式在 await 之后重新验证了我们的假设，确保只有在条件仍然满足时才会生成报告，由此确保即使状态由于 actor 重入发生了改变，我们依然执行正确的逻辑。

3. 事务性操作

对于更复杂的场景，比如需要保持多个状态之间的一致性，我们可以考虑事务性操作的模式。以 Room 的一个扩展场景为例：

```
actor Room {
    var visitorCount: Int = 0
    var revenue: Double = 0
    let ticketPrice: Double = 10

    // 不好的做法：多个 await 点增加了交织风险
    func badPurchaseTicket() async throws {
        await chargeVisitor(ticketPrice) // 这里可能失败
        visitorCount += 1
        revenue += ticketPrice
        await notifySystem() // 这里也可能发生交织
    }

    // 更好的做法：先收集操作，再批量执行
    func betterPurchaseTicket() async throws {
        // 1. 准备阶段（无 await）
        let newCount = visitorCount + 1
        let newRevenue = revenue + ticketPrice

        // 2. 执行外部操作
        try await chargeVisitor(ticketPrice)

        // 3. 原子性地更新状态
        visitorCount = newCount
        revenue = newRevenue

        // 4. 非关键的异步操作
        Task { await notifySystem() }
    }
}
```

```
    }  
}
```

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。

不可重入的设计考虑

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？

确实，如果将 actor 设计为不可重入的实例，那么在处理一条消息时（希望你还记得 actor 的信箱机制），actor 将不再查看信箱和处理其他内容。这类似于锁的行为，在任务执行期间将 actor 完全锁定，以避免交织和在交织中发生的内部状态更改。然而，回到使用锁来避免交织的方式，会带来性能大幅退化和死锁风险等问题，而这两个问题正是我们引入 actor 之初希望解决的关键问题。

因此，尽管 actor 中可能会发生交织现象，我们仍然选择使用 actor 来处理并发情况，而不是回到锁的方式。通过合理设计和使用 actor，我们可以获得更好的并发性能和更简洁的代码，同时避免了锁带来的问题。

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。

使用 @concurrent 控制执行器

随着 Swift 6.2 的发布，Apple 引入了一个新的特性：@concurrent 属性，它为开发者提供了更精细的并发执行控制能力。在本节中，我们将深入了解这个新特性的设计动机和使用方法。

为什么需要 @concurrent

在 Swift 6.2 之前，`nonisolated` 异步函数会自动在后台线程执行。但在 Swift 6.2 中，当启用 `NonisolatedNonsendingByDefault` 特性标志时，这种行为发生了重要变化：`nonisolated` 异步函数现在会在调用者的 `actor` 上执行，而不是自动切换到后台线程。

这个变化的核心动机是提供更好的性能和更明确的执行语义，比如我们可以期待更少的 `actor` 跳跃。然而，在某些情况下，我们确实希望函数能在全局执行器上运行，特别是对于那些 CPU 密集型的操作。这就是 `@concurrent` 属性的用武之地：它配合 `NonisolatedNonsendingByDefault` 编译标记，将那些 `nonisolated` 的执行方法“还原”到开启标记之前的状态，让它们回到全局执行器上运行。

让我们用一个具体的例子来说明这个问题：

```
class DataProcessor {
    // 在 Swift 6.2 之前，这会自动在后台执行
    // 在 Swift 6.2 中，开启 NonisolatedNonsendingByDefault 后，
    // 这会在调用者的 actor 上执行
    nonisolated func heavyJSONParsing(
        _ data: Data
    ) async throws -> [String: Any] {
        // 这个解析可能很耗时，会阻塞调用者的执行器
        return try JSONSerialization.jsonObject(with: data)
            as! [String: Any]
    }
}

@MainActor
class ViewController {
    func loadData() async {
        let data = // ... 获取数据
        // 在 Swift 6.2 中，这个调用会在 MainActor 上执行!
    }
}
```

```
// 可能会阻塞 UI
let result = await processor.heavyJSONParsing(data)
}
}
```

这种新行为虽然在某些场景下能提供更好的性能，但对于 CPU 密集型操作来说，它可能会阻塞调用者的执行器，特别是当调用者是 MainActor 时，这会直接影响 UI 的响应性。

@concurrent 的作用机制

@concurrent 属性允许我们明确指定某个异步函数应该在全局执行器上运行，而不是在调用者的 actor 上：

```
class DataProcessor {
    // 使用 @concurrent 确保在全局执行器上执行
    @concurrent
    nonisolated func heavyJSONParsing(
        _ data: Data
    ) async throws -> [String: Any] {
        // 这个函数现在会在全局执行器上执行，不会阻塞调用者
        return try JSONSerialization.jsonObject(with: data)
            as! [String: Any]
    }
}
```

当我们使用 @concurrent 时，函数会发生以下行为：

1. 强制执行器切换：函数将在全局执行器上运行，即使从主线程调用
2. 隔离域分离：创建一个新的隔离域，与调用者分离
3. **Sendable 要求**：所有参数和返回值必须符合 Sendable 协议

我们会在后面的章节详细讨论关于 Sendable 的话题。

最佳实践和注意事项

不是所有函数都适合使用 `@concurrent`。只有在以下情况下才考虑使用：

- 函数执行时间较长（通常超过几毫秒）
- 函数是 CPU 密集型或者可能阻塞线程的，比如大量的数据转换，I/O 操作等
- 你希望避免阻塞调用者的执行器

```
// ✗ 不适合使用 @concurrent

@concurrent
func simpleAddition(_ a: Int, _ b: Int) async -> Int {
    return a + b // 太简单，切换执行器的开销反而更大
}
```

```
// ✓ 适合使用 @concurrent

@concurrent
func complexCalculation(_ data: [Double]) async -> Double {
    // 复杂的数学计算，值得在后台执行
    return data.reduce(0) { sum, value in
        sum + cos(sin(sqrt(abs(value)))) // 模拟复杂计算
    }
}
```

如果你已经有使用 Swift 6.1 或更早版本编写的并发代码，在迁移到 Swift 6.2 时需要注意评估现有的 `nonisolated` 函数：

```
// Swift 6.1 及以前的行为
class LegacyProcessor {
    nonisolated func processData(_ data: Data) async -> String {
        // 这个函数之前会自动在后台执行
        return "processed"
    }
}
```

```
}

// Swift 6.2 的选择

class ModernProcessor {
    // 选择1：如果希望保持在后台执行

    @concurrent
    nonisolated func processData(_ data: Data) async -> String {
        return "processed"
    }

    // 选择2：如果可以接受在调用者执行器上运行

    nonisolated func processData(_ data: Data) async -> String {
        return "processed"
    }
}
```

正确使用 `@concurrent` 可以显著改善应用的性能特征，特别是在处理 CPU 密集型任务时。但记住，这个特性应该谨慎使用，只在真正需要时才使用。Swift 团队也在开发相关的迁移工具，未来可能会自动为合适的函数添加 `@concurrent` 属性，进一步简化迁移过程。

随着 Swift 并发模型的不断演进，`@concurrent` 代表了语言设计者对于性能和安全平衡的最新思考。它让我们在享受 Swift 并发安全性的同时，也能获得更好的性能控制能力，这为构建更高效、更可靠的并发应用程序奠定了坚实的基础。

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。

依靠 `actor` 模型所提供的能力，在共享状态的同时，编译器可以对访问数据的竞争行为进行规避，这完全避免了一整类常见的并发 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。

Actor 的可重入特性是并发编程中需要重点关注的概念。虽然可重入性允许 actor 在等待异步操作时处理其他任务，提高了并发性能，但也带来了状态交织的挑战。在 actor 方法中遇到 await 时，我们必须意识到后续代码执行时 actor 的状态可能已经改变，需要通过保存本地不变量或重新验证假设来确保逻辑正确性。

结合异步函数的语法，编译器会在合适的地方提醒我们将要发生跨越 actor 隔离域的访问。这些访问可能会在不同 actor 之间跳跃，它们基于底层调度来确保数据安全。相比于传统的可能由于人为疏失而失效的锁同步机制，以这种方式构建的并发系统显然更加稳定。

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, isolated 和 @concurrent 等。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。

Global Actor 和 系统集成

9

在前一章中，我们深入了解了 actor 模型的基础概念、隔离域机制以及可重入特性。Actor 类型在作为局部的数据隔离手段时是非常有效的：编译器可以保证定义在 actor 上的成员的安全。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 隔离域的灵活性在一定程度上得到扩展。

然而，在某些情况下，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单一的 actor 实例中。在这些场景下，我们需要一种作用域更加宽广的隔离手段。本章将重点探讨 Swift 并发中的全局 actor (Global Actor) 概念，以及它们如何与现有的系统框架进行深度集成。

全局 Actor 概念

什么是全局 Actor

全局 actor 是一种特殊的 actor 类型，它可以被当作属性包装来使用，能够被任意地添加到某个属性或方法前，对这个属性或方法进行标注，将它限定在该全局 actor 的隔离域中。

与普通的 actor 实例不同，全局 actor 提供了一个全局共享的隔离域，让分散在代码各处的声明能够被统一管理。

全局 Actor 的设计原理

全局 actor 通过 @globalActor 属性和 GlobalActor 协议来实现。Swift 标准库中的 MainActor 就是最重要的全局 actor 实现：

```
@globalActor final public actor MainActor : GlobalActor {  
    public static let shared: MainActor  
    // ...  
}
```

关键要素包括：

1. **@globalActor 属性**：将 actor 类型标记为全局 actor
2. **GlobalActor 协议**：要求提供一个 shared 静态实例

3. 单例模式：确保全局唯一的隔离域

这种设计让全局 actor 既保持了 actor 模型的安全性，又提供了跨类型、跨模块的统一隔离能力。我们先来看看 MainActor 这个最常用的全局 actor，然后再探讨自定义全局 actor 的方法和场景。

MainActor：主线程隔离域

主线程队列派发的传统方式

在 Swift 并发出现之前，Apple 平台的开发者一般通过 Grand Central Dispatch (GCD) 来管理线程操作。UI 相关的代码必须在主线程上执行，这导致了大量的线程切换代码：

```
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()
```

这种方式虽然可行，但存在一些问题：

- **容易被滥用**：开发者可能在任何地方都使用 `DispatchQueue.main.async`
- **执行顺序问题**：不必要的派发可能改变原本的执行顺序
- **调试困难**：线程安全问题往往只在运行时才能发现

为了改善这种情况，很多开发者会创建类似这样的辅助方法：

```
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 有些相似：当方法在 actor 隔离域时，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。

MainActor 隔离域

MainActor 是 Swift 标准库中定义的特殊 actor 类型，它将主线程抽象为一个 actor 隔离域：

```
@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 类型也只提供唯一一个对应主线程的 actor 隔离域。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 属性包装的使用

通过满足 GlobalActor 协议并添加 @globalActor 标记，MainActor 可以作为属性包装来注解其他类型或方法：

```
@MainActor class UIManager {
```

```
func updateInterface() {}

}

class DataProcessor {
    @MainActor var displayText: String = ""
    @MainActor func updateUI() {}
    func processData() {} // 非隔离方法
}

@MainActor var globalStatus: String = ""
```

根据添加的位置，`@MainActor` 有不同的作用：

- **类型级别**：整个类型的所有成员都被限定在 `MainActor` 隔离域中
- **成员级别**：只有被标记的特定成员处于隔离域中
- **全局级别**：全局变量或函数被限定在 `MainActor` 隔离域中

在 Task 中使用 MainActor

除了通过 `await` 进行 actor 跳跃外，也可以通过将 Task 闭包标记为 `@MainActor` 来使整个闭包切换到主线程隔离域：

```
class NetworkManager {
    func fetchData() {
        Task { await UIManager().updateInterface() }
        Task { @MainActor in UIManager().updateInterface() }
        Task { @MainActor in globalStatus = "Loading..." }
    }

    func fetchDataAsync() async {
        await UIManager().updateInterface()
    }
}
```

MainActor 的执行器在底层使用了 DispatchQueue.main 来确保所有操作都在主线程上执行。有趣的是，Swift 编译器反过来对 DispatchQueue.main.async 有特殊的处理机制，使得传递给它的闭包可以直接调用 @MainActor 隔离的方法，而无需再明确使用 await：

```
class LegacySupport {
    func mixedApproach() {
        // 编译器对 DispatchQueue.main.async 的字面写法有特殊处理
        DispatchQueue.main.async {
            UIManager().updateInterface() // 无需 await
            globalStatus = "Updated"      // 无需 await
        }
    }
}
```

这种编译器的特殊处理是为了帮助代码从 GCD 平滑迁移到 Swift 并发。但要注意，这种特殊处理仅适用于精确的字面写法。如果没有使用直接的 DispatchQueue.main.async，而是通过变量或其他方式引用主线程队列，则编译器不会帮你处理：

```
let queue = DispatchQueue.main
queue.async {
    // 严格检查下，这里会报错，需要想办法 await
    // UIManager().updateInterface()
}
```

一种常见的方法是使用 Task 和 await 来确保在主线程隔离域中调用 @MainActor 成员：

```
let queue = DispatchQueue.main
queue.async {
    Task {
        await UIManager().updateInterface()
    }
}
```

但是这种方法是有风险的：它改变了原来逻辑的执行顺序，`updateInterface` 的调用时机将不再是下一个主线程循环，而是由 Task 管理并异步执行，一般来说这会比原先只包含一次 `async` 派发的代码更晚执行。

这在大部分情况下可能不会造成什么问题，但并不能完全排除你的其他代码依赖于原先的执行顺序的情况。如果你确定旧代码已经通过某种手段保证运行在主线程了，那你完全可以使用 `MainActor.assumeIsolated` 来获取一个被隔离在 `@MainActor` 中的闭包，这样你就能以同步的方式使用其他被 `@MainActor` 隔离的成员了：

```
let queue = DispatchQueue.main
queue.async {
    MainActor.assumeIsolated {
        UIManager().updateInterface() // 在 MainActor 隔离域中调用
    }
}
```

但如果 `MainActor.assumeIsolated` 被错误地使用在非 `@MainActor` 的上下文中，你的程序将会在运行时崩溃。因此，使用 `MainActor.assumeIsolated` 时需要格外小心，最好辅以必要的自动化测试来确保代码的正确性。

虽然我们有一些手段在旧有的 `dispatch` 和新的异步世界中跳跃，但还是不建议在新代码中混用 GCD API 和 Swift 并发 API：这可能造成理解上的困难，并且影响并发性能的优化。

UIKit 框架中的 MainActor 集成

UIKit 类型的 MainActor 标注

Apple 在 UIKit 框架中广泛使用了 `@MainActor`，这是一个相当激进但必要的决定：

```
@MainActor class UIViewController : UIResponder
@MainActor class UIView : UIResponder
@MainActor class UIButton : UIControl
@MainActor class UILabel : UIView
```

在 Swift 并发时代之前，虽然文档规定这些 UI 相关的类型需要在主线程上操作，但这些规则缺乏编译器保证，只能在运行时通过调试工具（如 Main Thread Checker）检测到违规行为。

@MainActor 的引入将这些类型的成员明确圈入隔离域中，提供了编译时的安全保证。

UIViewController 中的异步操作模式

在 UIViewController 中进行异步操作变得更加简单和安全：

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        Task {
            let url = URL(string: "https://api.example.com/data")!
            let (data, _) = try await URLSession.shared.data(from: url)
            self.updateUI(with: data) // 可以同步调用
        }
    }

    private func updateUI(with data: Data) {
        // 更新 UI 的代码
    }
}
```

这里的关键点：

1. ViewController 继承自 UIViewController，处于 @MainActor 隔离域中
2. viewDidLoad 运行在 MainActor 隔离域内
3. Task 创建的任务继承了当前的 actor 运行环境
4. 因此闭包也运行在 MainActor 隔离域中，可以同步调用 updateUI

Task.init vs Task.detached

选择合适的 Task 创建方式很重要：

```
class ViewController: UIViewController {
    func performAsyncOperation() {
        // 继承当前隔离域
        Task {
            let data = await fetchData()
            self.updateUI(with: data) // 同步调用
        }

        // 脱离当前隔离域
        Task.detached {
            let data = await fetchData()
            await self.updateUI(with: data) // 需要 await
        }
    }
}
```

在 UIViewController 环境中，如果希望在异步操作后回到主线程调用自身成员，Task.init 通常是最好的选择。

跨隔离域调用的处理

对于在 UIViewController 外发生的调用，必须使用 await 进行 actor 跳跃：

```
class NetworkService {
    func updateViewController() async {
        let button = await UIButton()
        await button.setTitle("New Title", for: .normal)

        let viewController = ViewController()
```

```
    await viewController.view.addSubview(button)
}
}
```

渐进式迁移策略

Swift 编译器采用了渐进式的迁移策略来避免破坏现有代码：

```
class LegacyCode {
    // 在非异步函数中，编译器暂时容忍这种访问
    func oldStyleUICode() {
        let button = UIButton()
        button.setTitle("Click", for: .normal) // 在宽松模式下暂时不报错
    }

    // 但在异步函数中，会要求严格的 actor 隔离检查
    func newStyleUICode() async {
        let button = await UIButton()
        await button.setTitle("Click", for: .normal) // 必须使用 await
    }
}
```

激活编译器检查

虽然实际上存在于 ViewController 的函数已经是 MainActor 隔离的了，但如果我们要激活编译器检查的话也可以将某个函数明确标记为 @MainActor：

```
class ViewController: UIViewController {
    @MainActor func explicitUIMethod() {
        // 被明确标记的方法
    }
}
```

```
class Consumer {
    func useViewController() {
        // 编译错误：需要在 MainActor 隔离域中调用
        // ViewController().explicitUIMethod()

        Task {
            await ViewController().explicitUIMethod() // 正确方式
        }
    }
}
```

这种方式可以在迁移过程中逐步激活安全检查。

Swift 6 并发检查模式

实际上，隔离域检查的严格程度取决于并发检查模式的设置：

在 Swift 6 中，提供了三种并发检查级别：

1. **Minimal**: 基础检查，只在明确使用 `Sendable` 或 `@MainActor` 的地方进行检查
2. **Targeted**: 中等级别检查，针对性地检查潜在问题
3. **Complete**: 完整的严格并发检查，尝试捕获所有潜在的数据竞争

可以通过以下方式启用：

- 编译器参数: `-strict-concurrency=complete`
- Xcode 设置：“Strict Concurrency Checking” 设为 “Complete”

严格的并发检查会在以下情况下进行：

- 异步方法和函数
- actor 中的方法

- Task 相关 API
- 开启了 Swift 6 模式或严格并发检查启用时的所有代码

因此，如果设置了 Swift 6 的严格模式，即使是同步代码也需要遵守 actor 隔离规则。这意味着上面的 oldStyleUICode 在严格模式下也会要求使用 await 或确保在正确的隔离域中运行。

在迁移到 Swift 6 时，建议逐步启用严格并发检查，最终达成完全的隔离域安全，以确保在编译时就完全消除数据竞争带来的隐患。我们会在后面的章节详细讨论异步迁移的一些常见技巧和方法。

自定义全局 Actor

使用场景

虽然 MainActor 可以处理大部分主线程相关的需求，但在某些情况下可能需要自定义全局 actor：

1. 分组状态管理：多个分散的状态需要串行访问避免数据竞争
2. 全局变量隔离：需要将全局变量纳入特定的隔离域
3. 跨模块协作：不同模块间需要共享同一个隔离域
4. 专门的执行上下文：如数据库访问、文件 I/O 等需要特定执行策略的操作

定义自定义全局 Actor

看到 MainActor 的威力之后，你可能会想：“能不能创建自己的全局 actor 呢？”答案是肯定的，而且比你想象的要简单。

创建自定义全局 actor 的基本模式很直接：

```
@globalActor actor DatabaseActor {  
    static let shared = DatabaseActor()  
}
```

有了这个 DatabaseActor，我们就可以像使用 @MainActor 一样来标记需要数据库隔离的代码：

```
@DatabaseActor var databaseConnection: DatabaseConnection?  
  
@DatabaseActor func performDatabaseOperation() async {  
    // 所有数据库操作都在同一个隔离域中  
    guard let connection = databaseConnection else { return }  
    // 安全地访问数据库连接  
}
```

这样做好处是显而易见的：所有标记为 @DatabaseActor 的代码都会在同一个隔离域中运行，确保数据库访问的线程安全。

设计良好的全局 Actor

不过，在实际设计自定义全局 actor 时，有一个重要的最佳实践需要注意。你可能会想在全局 actor 中添加一些实例属性或方法，但这往往会导致混淆。

让我们看看一个容易让人困惑的设计：

```
// 不推荐的设计  
  
@globalActor actor MyActor {  
    static let shared = MyActor()  
    var value: Int = 0 // 这个实例属性容易造成混淆  
}  
  
@MyActor func confusingFunction(actor: MyActor) async {  
    // 这里访问的是参数 actor 的 value  
    print(await actor.value)  
    // 这里访问的是 MyActor.shared 的 value  
    print(await MyActor.shared.value)  
}
```

看到问题了吗？在 `confusingFunction` 中，我们有两个不同的 `MyActor` 实例：一个是参数传入的，另一个是全局共享的 `shared`。这种设计很容易让开发者（包括你自己）在后续维护时产生混淆。

更好的做法是将全局 actor 设计为纯粹的“标记”，不允许外部创建新的 actor 实例：

```
@globalActor actor DatabaseActor {  
    static let shared = DatabaseActor()  
    private init() {} // 私有初始化，防止外部创建实例  
}
```

通过私有化初始化方法，我们确保了只有一个 `DatabaseActor` 实例存在，消除了潜在的混淆。

跨模块的协作

自定义全局 actor 的一个强大之处在于它们可以跨模块工作。想象一下，你正在开发一个涉及文件系统操作的框架，同时主应用也需要进行类似的操作。如果没有统一的隔离域，这两部分代码可能会产生数据竞争。

这时候，全局 actor 就派上用场了：

```
// 在你的 Framework Module 中  
  
@globalActor public actor FileSystemActor {  
    public static let shared = FileSystemActor()  
    private init() {}  
}  
  
@FileSystemActor public func readConfigFile() async -> Config {  
    // 框架中的文件系统操作  
}
```

然后在主应用中：

```
// Application Module
```

```
import FrameworkModule

@FileSystemActor func updateConfig() async {
    let config = await readConfigFile() // 可能的优化：同一隔离域
    // 修改配置...
}
```

这样，框架和应用的文件系统操作都在同一个隔离域中，既保证了线程安全，又为编译器提供了优化的可能性。

全局 Actor 的性能优势

使用自定义全局 actor 的一个重要好处是性能优化。当多个操作都在同一个隔离域中时，编译器可以减少不必要的 actor 跳跃。这一点在后面的“Actor 性能优化和调试”章节中会有详细讨论，包括如何使用 isolated 参数来获得最佳性能。

全局 Actor 的设计原则

保持单一职责

在设计自定义全局 actor 时，有一个金科玉律：一个 actor 只做一类事情。就像 MainActor 只负责主线程相关的操作一样，你的自定义 actor 也应该有明确的边界。

比如，如果你的应用需要同时处理网络请求和文件操作，分别创建两个 actor 会比一个“万能”actor 更好：

```
@globalActor actor NetworkActor {
    static let shared = NetworkActor()
    private init() { }
}

@globalActor actor FileSystemActor {
    static let shared = FileSystemActor()
```

```
private init() { }

// 各自在自己的领域中发挥作用
@NetworkActor func fetchUserData() async -> UserData {
    // 网络请求相关的代码
}

@FileSystemActor func saveUserData(_ data: UserData) async {
    // 文件存储相关的代码
}
```

这样的设计让代码更加清晰，也方便后期维护。

避免隔离域的“踩脚”

在使用多个全局 actor 时，有一个容易掉入的陷阱：让一个函数同时处理多个不同隔离域的数据。这不仅会让编译器困惑，也会让代码变得难以理解。

比如，下面这样的代码就存在问题：

```
// 不好的设计：一个函数要处理两个不同的隔离域
func processData(
    @NetworkActor networkData: NetworkData,
    @FileSystemActor fileData: FileData
) async {
    // 这个函数应该在哪个隔离域中运行？
    // 编译器也不知道怎么办！
}
```

更好的做法是让每个函数只在一个隔离域中工作：

```
// 在网络隔离域中处理网络数据
@NetworkActor func processNetworkData(
```

```
_ data: NetworkData
) async → ProcessedData {
    // 专心做好网络相关的事情
    return ProcessedData(from: data)
}

// 在文件系统隔离域中保存数据
@FileSystemActor func saveProcessedData(
    _ data: ProcessedData
) async {
    // 专心做好文件相关的事情
}
```

这样的设计不仅让编译器开心，也让代码的意图更加明确。

从现有项目开始

如果你正在维护一个现有的项目，想要引入自定义全局 actor，记住一个词：“渐进”。不要试图一次性重构整个代码库，那样只会让你无法应付。

一个实用的迁移策略可能是这样的：

```
// 第一步：先定义你的 actor
@globalActor actor CacheActor {
    static let shared = CacheActor()
    private init() { }
}

// 第二步：找到那些“明显”需要隔离的地方
@CacheActor var memoryCache: [String: Any] = [:]

// 第三步：在新功能中使用 actor
@CacheActor func updateCache(key: String, value: Any) {
    memoryCache[key] = value // 安全的访问
}
```

```
}
```

然后逐步迁移现有的代码，比如在下一次重构时加上 @CacheActor 注解。这样的方式可以让你在保持系统稳定的同时，逐步享受到 Swift 并发的好处。

与系统框架的集成最佳实践

UIKit 集成策略

在使用 UIKit 时，合理利用 @MainActor：

```
class ModernViewController: UIViewController {
    // 显式标记重要的 UI 方法
    @MainActor override func viewDidLoad() {
        super.viewDidLoad()
        setupUI()
    }

    // 使用 Task 进行异步操作
    @MainActor func loadData() {
        Task {
            do {
                let data = try await dataService.fetchData()
                updateUI(with: data)
            } catch {
                showError(error)
            }
        }
    }

    // 保持 UI 方法的同步性
    private func updateUI(with data: Data) {
```

```
// UI 更新逻辑
}

private func showError(_ error: Error) {
    // 错误显示逻辑
}
}
```

在 SwiftUI 中的应用

SwiftUI 的情况更加有趣。在 SwiftUI 中，整个 View 协议都被标记为 `@MainActor`，这意味着你的所有 UI 代码都在主线程上运行。而在现代 SwiftUI 开发中，我们通常使用 `@Observable` 宏来创建数据模型。

不过，这里有一个重要的细节需要注意。`@Observable` 宏生成的类型默认不会自动隔离到 `@MainActor`。但在实际使用中，对于驱动 View 的状态来说，绝大部分情况我们都希望它们在主线程上运行。因此，通常我们会将这样的数据模型标记 `@MainActor`：

```
@Observable
@MainActor
class DataModel {
    var data: [Item] = []
    var isLoading = false

    func loadData() async {
        isLoading = true
        let newData = await dataService.fetchItems()

        data = newData
        isLoading = false
    }
}
```

```
struct ContentView: View {
    private var model = DataModel()

    var body: some View {
        VStack {
            if model.isLoading { // SwiftUI 框架处理了线程安全
                ProgressView("Loading...")
            } else {
                List(model.data) { item in
                    Text(item.title)
                }
            }
        }.task {
            await model.loadData()
        }
    }
}
```

与第三方库的集成

在使用 UIKit 和其他系统框架时，你会发现很多类型已经被标记为 `@MainActor`，但对于第三方库，情况可能有所不同。大多数第三方库可能还没有完全适配 Swift 并发，这时候你可能需要创建适配层或使用一些高级技巧。

具体的处理方法我们将在下一章《Sendable 协议和数据安全》中详细探讨，特别是 `@unchecked Sendable` 的使用场景和最佳实践。

Actor 性能优化和调试

理解 Actor 跳跃的成本

在深入 Swift 并发的性能优化之前，我们需要先理解一个基本概念：**actor 跳跃的成本**。每当我们使用 `await` 调用不同隔离域中的方法时，都可能发生线程切换和任务重新调度，这些操作都有其固有的开销。

让我们通过一个具体的例子来看看这种开销有多大：

```
actor DataProcessor {
    private var processedCount = 0

    func processItem(_ item: String) -> String {
        processedCount += 1
        return "Processed: \(item) (\#\((processedCount)))"
    }

    func processBatch(_ items: [String]) -> [String] {
        var s: [String] = []
        for i in items {
            processedCount += 1
            s.append("Processed: \(i) (\#\((processedCount))")
        }
        return s
    }

    func getProcessedCount() -> Int {
        return processedCount
    }
}
```

```
@MainActor
class PerformanceDemo {
    func demonstrateActorHoppingCost() async {
        let processor = DataProcessor()
        let startTime = CFAbsoluteTimeGetCurrent()

        // 方案 1: 频繁的 actor 跳跃
        for i in 0..<10000 {
            let _ = await processor.processItem("item-\(i)")
        }

        let timeWithHopping = CFAbsoluteTimeGetCurrent() - startTime
        print("频繁 actor 跳跃耗时: \(timeWithHopping) 秒")

        // 方案 2: 批量处理减少跳跃
        let startTime2 = CFAbsoluteTimeGetCurrent()
        let items = (0..<10000).map { "item-\($0)" }
        let _ = await processor.processBatch(items)

        let timeWithBatching = CFAbsoluteTimeGetCurrent() - startTime2
        print("批量处理耗时: \(timeWithBatching) 秒")
        print("性能提升: \(timeWithHopping / timeWithBatching)x")
    }
}
```

频繁 actor 跳跃耗时: 0.1255350112915039 秒

批量处理耗时: 0.005108952522277832 秒

性能提升: 24.571575238584128x

在一些基准测试中，频繁的 actor 跳跃可能比批量处理慢 20-30 倍，当然这只是一个粗略的估计，具体数字取决于更多要素，但这个例子清楚地展示了 actor 跳跃是存在成本的。

使用 isolated 进行隔离域设定

isolated 参数是一个强大的性能优化工具，它的核心思想是让整个函数在指定的 actor 隔离域中执行，从而避免频繁的 actor 跳跃。

isolated 参数的威力

让我们通过一个对比来看看 isolated 参数的效果。假设我们有两个不同的 actor 类型。

```
actor DataProcessor {
    private var processedCount = 0

    func processItem(_ item: String) -> String {
        processedCount += 1
        return "Processed: \(item) (\#\((processedCount))"
    }
}

actor ExternalActor {
    private var value = 0

    func getValue() -> Int {
        return value
    }

    func updateValue() {
        value += 1
    }
}
```

现在，我们需要在 DataProcessor 中频繁调用 ExternalActor 的方法，而且由于 API 设计的原因，我们难以将 await 从 for 循环中拿出来：

```
// ✗ 普通方法：在循环中频繁跳跃

actor DataProcessor {
    func processWithExternalActor(_ external: ExternalActor) async {
        // 糟糕的性能：每次访问都需要 actor 跳跃
        for _ in 0..<100 {
            print(await external.getValue())      // 跳跃到 external
            // 执行完毕后回到 DataProcessor 的隔离域
            await external.updateValue()         // 再次跳跃到 external
        }
    }
}
```

这种方法的性能非常低下，因为每次调用 `await` 都会导致隔离域切换到 `DataProcessor` 中。明明大部分工作都是在 `ExternalActor` 的隔离域中完成的，但是仅只是由于我们需要在 `DataProcessor` 中调用它的 API，我们就得承担这些性能开销的代价。

将 `ExternalActor` 参数标记为 `isolated` 可以改变这个状况。整个函数将不再使用 `DataProcessor` 的隔离域，而是直接在 `ExternalActor` 的隔离域中执行：

```
// ✓ 优化方法：使用 isolated 参数

actor DataProcessor {
    func processWithIsolatedParameter(
        _ external: isolated ExternalActor
    ) {
        // 好多了：整个函数在 ExternalActor 的隔离域中执行
        for _ in 0..<100 {
            print(external.getValue()) // 同步访问，无需跳跃
            external.updateValue()   // 同步访问，无需跳跃
        }
    }
}
```

在实际测试中，使用 `isolated` 参数的版本通常比普通方法的版本快数十倍，这种优化在处理大量数据或频繁操作时尤其重要。

#isolation 宏

在了解了 `isolated` 参数的强大功能后，你可能会想：“如果我想让函数继承调用者的隔离域，但又不想每次都显式传递 `actor` 参数，该怎么办？”这正是 `#isolation` 宏的用武之地。

`#isolation` 是 Swift 6 引入的一个编译期宏，它可以自动捕获当前代码的隔离上下文。当作为 `isolated` 参数的默认值使用时，它让函数能够动态继承调用者的隔离域，实现了隔离信息的自动传递。

举个例子，考虑下面 actor：

```
actor Presenter {
    func processForDisplay(
        _ data: Data,
        isolation: isolated (any Actor)? = #isolation
    ) async {
        // 默认在调用者的隔离域中执行
    }

    func animatePresentation(
        isolation: isolated (any Actor)? = #isolation
    ) async {
        // 默认在调用者的隔离域中执行
    }

    func notifyCompletion() async {
        // 在 self (actor) 隔离域中执行
    }
}
```

在调用时，如果不传入任何 isolation 参数，那么它的默认值将会导致虽然 processForDisplay 和 animatePresentation 是声明在 Presenter 这个 actor 中的，但是实际上它运行在调用者所定义的 MainActor 的隔离域中：

```
@MainActor
func presentData() async {
    let data = Data() // 假设这是需要处理的数据
    let presenter = Presenter()

    // 在 MainActor 上执行
    await presenter.processForDisplay(data)

    // 在 MainActor 上执行
    await presenter.animatePresentation()

    // 在 presenter 上执行
    await presenter.notifyCompletion()
}
```

值得注意的是，#isolation 不仅可以捕获具体的 actor 隔离域，也可以表示“非隔离”状态。这就是为什么 isolation 参数的类型是可选的 (any Actor)?:

- 如果调用处在某个 actor 的隔离域中，#isolation 就是那个 actor
- 如果调用处是非隔离的，#isolation 就是 nil。这会让被调用者运行在原本应在的隔离域中。

这种设计让 #isolation 成为了一个真正通用的隔离域传递机制，无论在什么上下文中都能正确工作。

使用 @isolated(any) 继承隔离域

在我们深入了解了 isolated 参数带来的性能优化后，有必要介绍 Swift 6 中另一个重要的性能改进：@isolated(any) 函数类型。这个改进直接影响了我们日常使用最频繁的 Task 初始化器，让并发代码的性能得到了显著提升。

Task 初始化的性能问题

在 Swift 5.5 中，虽然 Task 的闭包会继承创建时的隔离域，但 Task.init 作为标准库 API，无法在运行时获知传入闭包的具体隔离信息。在 Swift 5.5 中，Task.init 的签名为：

```
// Swift 5.5
struct Task {
    init(
        priority: TaskPriority? = nil,
        operation: @escaping @Sendable () async -> Success
    )
}
```

这导致了一个微妙但重要的性能问题：

```
@MainActor
func updateUI() {
    print("更新UI组件")
}

// 在 Swift 5.5 中，这样的代码存在性能问题
func createTask() {
    Task {
        // 编译器不知道应该在 MainActor 上开始
        await updateUI()
    }
}
```

虽然 updateUI() 被标记为 @MainActor，但 Task 初始化器只能看到一个普通的闭包，无法获知它最终需要在 MainActor 上执行。因此，Task 会先在全局并发执行器上启动，然后再“跳转”到 MainActor，造成不必要的执行器切换开销。

@isolated(any) 的解决方案

Swift 6.0 中为函数参数引入了 @isolated(any) 标注，它可以在运行时携带函数的隔离信息：

```
// @isolated(any) 函数可以暴露其隔离信息
func schedule(
    _ operation: @escaping @isolated(any) () async -> Void
) {
    if let isolation = operation.isolation {
        print("函数隔离于: \(isolation)")
    } else {
        print("函数没有特定隔离")
    }
}
```

现在，Task.init 的参数被更新为接受 @isolated(any) 闭包：

```
// Swift 6.0
struct Task {
    init(
        priority: TaskPriority? = nil,
        operation: sending @escaping @isolated(any)
            () async -> Success
    )
}
```

这意味着它可以：

1. 在运行时检查闭包的隔离信息
2. 直接在正确的执行器上同步入队任务
3. 避免不必要的执行器切换

实际的性能改进

让我们通过具体的例子来理解这个改进的影响：

```
@MainActor
func demonstrateTaskImprovement() {
    let vm = ViewModel()

    // Swift 5.5 的行为（概念性）：
    // Task 会先在全局执行器上启动，然后“跳”到 MainActor
    // 这导致了不可预测的任务顺序

    // Swift 6 + SE-0431 的改进：
    // Task 可以直接在 MainActor 上同步入队

    Task {
        vm.increment() // 直接在 MainActor 上执行，无需跳转
    }

    Task {
        vm.increment() // 这个任务会按预期的顺序执行
    }
}
```

这个改进带来了两个重要好处：

1. **性能提升**：减少了执行器切换的开销，特别是在频繁创建 Task 的场景下
2. **任务顺序可预测**：在 MainActor 等有序执行器上，任务现在会按照提交的顺序执行

对于任务顺序，让我们看一个更清晰的例子：

```
actor DataProcessor {
    var events: [String] = []
```

```
func processEvent(_ event: String) {  
    events.append(event)  
    print("处理: \(event), 总计: \(events.count)")  
}  
  
func demonstrateOrdering() {  
    // Swift 6.0 之前: 这些任务的执行顺序不可预测  
    // Swift 6.0 之后: 在隔离上下文中, 顺次执行1-5  
    for i in 1...5 {  
        Task {  
            processEvent("事件 \(i)")  
        }  
    }  
}
```

对开发者的影响

这个改进对日常开发的影响是潜移默化的:

1. 无需修改代码: 现有的 Task 创建代码会自动获得性能提升
2. 更符合直觉的行为: 特别是在 UI 开发中, 任务的执行顺序更可预测
3. 为未来优化奠定基础: @isolated(any) 的设计为未来更多的优化留下了空间

值得注意的是, Swift 6.0 中使用 @isolated(any) 而不是简单的 @isolated, 这是为了给未来的扩展预留空间。未来可能会引入能够捕获不仅是隔离值, 还包括其类型信息的机制。

这个改进是 Swift 并发演进的一个典型例子: 在保持 API 兼容的同时, 通过底层优化为开发者带来实实在在的性能提升。结合之前介绍的 isolated 参数和 #isolation 宏, Swift 6 为我们提供了一套完整的隔离域控制工具, 让并发代码既安全又高效。

常见的性能优化方式

最后来看看几种我们应该注意的场景，对 actor 进行性能优化的核心原则是减少隔离域的跳跃，这项任务的前提是我们需要能拥有一定的“嗅觉”，去察觉到代码中可能存在大量跳跃的部分。

手段 1：避免循环中的 Actor 跳跃

在循环中大量 await 是最常见的导致性能问题的原因：

```
// ✗ 低效：每次循环都跳跃
@MainActor
class InefficientExample {
    func updateUI(with items: [DataItem]) async {
        let processor = DataProcessor()

        for item in items {
            // 每次循环都要跳跃到 DataProcessor
            let processed = await processor.processItem(item.rawData)
            // 然后跳回 MainActor 更新 UI
            updateSingleItem(with: processed)
        }
    }

    private func updateSingleItem(with data: String) {
        // UI 更新代码
    }
}
```

如果你在循环中频繁调用 await，它每次都会导致线程切换和任务调度，这会显著降低性能。

一个原则上的优化方法是批量处理数据，尽量不要在循环内部 await，减少跳跃次数：

```
// ✓ 高效：批量处理
```

```
@MainActor
```

```
class EfficientExample {
    func updateUI(with items: [DataItem]) async {
        let processor = DataProcessor()

        // 一次性发送所有数据进行处理
        let rawData = items.map { $0.rawData }
        let processedData = await processor.processBatch(rawData)

        // 在 MainActor 中批量更新 UI
        for (index, processed) in processedData.enumerated() {
            updateSingleItem(with: processed)
        }
    }

    private func updateSingleItem(with data: String) {
        // UI 更新代码
    }
}
```

手段 2：用观察代替跨隔离域轮询

另一个常见问题是频繁查询 actor 的状态：

```
// ✗ 低效：频繁查询状态
@MainActor
class StatusQueryExample {
    func monitorProgress() async {
        let processor = DataProcessor()

        // 启动处理任务
        Task {
            await processor.startLongRunningTask()
        }
    }
}
```

```
// 频繁查询进度
while await processor.getProcessedCount() < 1000 {
    let count = await processor.getProcessedCount()
    updateProgressBar(progress: Float(count) / 1000.0)
    try? await Task.sleep(nanoseconds: 100_000_000) // 0.1 秒
}
}

private func updateProgressBar(progress: Float) {
    // 更新进度条
}
}
```

相比与轮询，特别是跨隔离域的轮询，使用类似“观察”的方式，让 actor 主动通知状态变化会更高效，比如 AsyncSequence：

```
// ✅ 高效：使用 AsyncSequence 流式更新
actor DataProcessor {
    private var processedCount = 0
    private var progressContinuation: AsyncStream<Int>.Continuation?

    func startLongRunningTaskWithProgress() -> AsyncStream<Int> {
        return AsyncStream<Int> { continuation in
            self.progressContinuation = continuation

            Task {
                await self.performLongRunningTask()
                continuation.finish()
            }
        }
    }
}
```

```
private func performLongRunningTask() async {
    for i in 0..<1000 {
        // 模拟处理
        processedCount = i + 1
        progressContinuation?.yield(processedCount)
        // 实际处理逻辑...
    }
}

}

@MainActor
class StreamingExample {
    func monitorProgressEfficiently() async {
        let processor = DataProcessor()

        for await progress
            in await processor.startLongRunningTaskWithProgress()
        {
            updateProgressBar(progress: Float(progress) / 1000.0)
        }
    }

    private func updateProgressBar(progress: Float) {
        // 更新进度条
    }
}
```

手段 3: 使用合适的 isolated 进行参数隔离

我们在上一节已经看到了使用 `isolated` 来指定参数隔离的具体例子。在使用标准的 Task API 时, Swift 6 已经帮我们做了很多工作, 来尽量提升性能, 减少跳转。但是在自定义的多隔离域系统中, 如果能够根据我们自己的代码逻辑, 选择最合适的隔离域, 将会为性能带来很大提升。

小结

经过本章的学习，相信你已经对全局 actor 有了深入的理解。全局 actor 确实是 Swift 并发模型中一个相当巧妙的设计，它让我们可以在更大的范围内管理隔离域，同时保持代码的安全性和可读性。

特别是 MainActor，它可能是你在日常开发中最常接触的全局 actor。通过将主线程抽象为一个 actor 隔离域，它为 UI 编程提供了编译时的安全保证，这在以前基于 GCD 的时代是无法实现的。

让我们回顾一下本章的几个重要收获：

1. 全局 actor 的魅力：通过 @globalActor 和单例模式，我们可以创建跨类型、跨模块的统一隔离域，这为大型项目中散落在各处甚至是跨越模块的并发管理提供了新的可能性。
2. MainActor 的实用价值：从繁琐的 DispatchQueue.main.async 到简洁的 @MainActor 标注，这不仅是语法上的改进，更是编译时安全的质的飞跃。
3. 与系统框架的无缝集成：看到 UIKit 和 SwiftUI 如何拥抱 @MainActor，我们可以感受到 Apple 对 Swift 并发的重视和投入。
4. 自定义全局 actor 的灵活性：当 MainActor 不够用时，我们可以创建专门的隔离域来解决特定问题，比如数据库访问、网络操作等。
5. 性能优化的注意事项：通过合理的设计和 isolated 参数的使用，我们可以显著减少 actor 跳跃，获得数十倍的性能提升。

全局 actor 为现有项目的并发迁移提供了一条可行的路径。你不需要一次性重构整个代码库，而是可以采用渐进式的策略，在保持系统稳定的同时，逐步享受到 Swift 并发带来的好处。

在实际项目中，我建议你从 UI 层开始尝试 @MainActor，然后根据需要在必要时引入其他的自定义全局 actor。记住，性能优化永远是一个持续的过程，通过本章介绍的工具和技巧，你可以建立有效的监控机制，及时发现和解决性能瓶颈。

接下来，我们将在下一章探讨 `Sendable` 协议和数据安全。如果说全局 actor 解决了“在哪里执行”的问题，那么 `Sendable` 协议就是要解决“那些东西可以安全传递”的问题。这在 Swift 6 严格并发检查的时代显得尤为重要，让我们继续这段并发安全的探索之旅。

Sendable 协议， 数据安全和迁移

10

在前面的章节中，我们已经深入了解了 actor 模型、隔离域机制以及全局 actor 的应用。这些概念为我们提供了强大的并发安全工具，但在 Swift 并发模型中，还有一个同样重要的概念需要我们掌握：**Sendable 协议**。

Sendable 协议是 Swift 并发安全的基石，它定义了哪些数据可以安全地在不同并发域之间传递。随着 Swift 6 引入严格并发检查，Sendable 协议的重要性更加凸显，成为现代 Swift 开发中不可或缺的核心概念。本章将全面探讨 Sendable 协议的设计、使用以及在实际项目中的迁移策略。

并发域间数据传递的挑战

数据安全的核心问题

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，我们可以进入隔离域进行操作。然而，这种跳跃往往涉及数据的传递。例如，actor 上的方法可能需要接受参数，它们也可能会返回某些值，这些数据都会随着调用和返回一同跨越隔离域。

让我们考虑一个简单的示例，比如这样一个 actor：

```
actor Room {  
    let roomName: String  
    init(roomName: String) {  
        self.roomName = roomName  
    }  
}
```

我们用 PersonStruct 定义一个“访客”的概念，当它进入某个 Room 时，可以从 Room 获取一条包含房间名称的问候消息：

```
struct PersonStruct {  
    let name: String  
    var message: String = ""  
  
    init(name: String) {  
        self.name = name  
    }  
}  
  
extension Room {  
    func visit(_ visitor: PersonStruct) -> PersonStruct {  
        var result = visitor  
        result.message = "Hello, \(visitor.name). From \(roomName)."  
        return result  
    }  
}
```

值类型的安全性

visit 方法接受一个 PersonStruct 参数，并返回一个新的 PersonStruct 值。PersonStruct 的所有成员都具有值语义，因此在从隔离域外部调用 Room.visit 方法时，参数 visitor 也是一个具有值语义的值。

对于这样的值，在 actor 的 visit 方法内部如果想要对其进行修改，必须先复制一份副本，然后对副本进行操作。同样地，返回的 PersonStruct 在被调用者使用时，也遵循值语义的特性。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，它们之间不会产生问题。因此，这种做法是安全的：

```
let person = PersonStruct(name: "onevcat")  
  
for i in 0 ..< 1000 {  
    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.
```

引用类型的危险

然而，当我们使用具有引用语义的类型（比如 class 类型）时，情况就完全不同了：

```
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)")
        // 危险：多个并发任务共享同一个 PersonClass 实例
        let p = await room.visit(person)
        print(p.message)
    }
}
```

```
}
```

对于 PersonClass 类型，在传递给不同的 Room 隔离域时，它们指向的是同一个内存引用。因此，调用 visit 和访问 p.message 期间，存在从多个线程同时访问共享内存的风险。虽然 actor Room 可以保证其自身成员免受数据竞争的影响，但它无法确保跨越隔离域的参数或返回值的安全性。

Task 中的数据安全问题

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：

```
class Sample {
    var value: String = ""
    func foo() {
        Task { value += "hello" } // 数据竞争风险
        Task { value += "world" } // 数据竞争风险
        // 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，这里的核心问题是：“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？”。这就是 Sendable 协议需要解决的问题。

Sendable 协议的设计原理

标志协议的特性

Sendable 和现存在 Swift 中的所有协议不同，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。

Swift 并发设计的三个目标

在 Swift 并发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. **编译时安全**: 对于那些跨越并发域时可能不受保护的可变状态，编译器应该给出错误，以保证数据安全
2. **性能和灵活性**: Swift 的并发设计鼓励使用值类型，但在某些情况下引用类型确实可以带来更优秀的性能。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. **渐进式迁移**: 在 Swift 引入并发之前，已经存在大量的代码，需要支持平滑和渐进式的迁移过程

对于这三个目标，Swift 的解决方案是：

- 使用 Sendable 来标记那些安全的类型
- 提供 @unchecked Sendable 让开发者可以在需要时绕过编译器的检查
- 在 Swift 5 的语言模式中大部分关于 Sendable 的检查默认都是关闭的，Swift 6 中将启用完整检查

我们来逐个看看这些方案是如何帮助编译器实现目标的。

类型系统中的 Sendable 实现

值类型的 Sendable 实现

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 {}
extension Double: Sendable {}
extension Data: Sendable {}
// ...
```

这些基础类型构成了其他“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：

```
extension Optional: Sendable
    where Wrapped: Sendable {}
extension Array: Sendable
    where Element: Sendable {}
extension Dictionary: Sendable
    where Key: Sendable, Value: Sendable {}
extension Set: Sendable
    where Element: Sendable {}
```

自定义结构体的 Sendable

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可以被直接标记为 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（比如一个普通的 class 类型），该成员可能会被多个并发域同时修改，此时这个类型不能再满足 Sendable：

```
class NonSendableClass {}

struct ProblematicStruct: Sendable { // 编译错误
    let name: String
    var message: String = ""
    let nonSendable: NonSendableClass = NonSendableClass()
}

// 错误: Stored property 'nonSendable' of 'Sendable'-conforming struct
// 'ProblematicStruct' has non-sendable type 'NonSendableClass'
```

自动 Sendable 推断

在同一模块中，如果一个 struct 满足它的所有成员都是 Sendable，编译器会自动进行推断：

```
// 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。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。

跨模块的 `Sendable` 考虑

不过，当我们将类型声明为 `public` 时，自动推断就不再适用了。因为可能存在只有在模块内部可见的 `internal` 或 `private` 成员，因此在模块外部，这样的类型将无法被自动被视为 `Sendable`：

```
// ModuleA
public struct PersonModuleA {
    public let name: String
    public 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`
    // private let hidden: NonSendableType?

    public init(name: String) {
        self.name = name
    }
}

// Module B
import ModuleA
```

```
foo(value: PersonModuleA(name: "onevcat"))
// 错误: 'foo(value:)' requires that
// 'PersonModuleA' conform to 'Sendable'
```

因此，当我们把某个类型标记为 public 时，如果有条件，我们最好也考虑它是否能够满足 Sendable，并将它作为公开 API 保证的一部分。

@frozen 结构体的特殊处理

如果我们能够确定某个 struct 不会再被修改，可以使用 @frozen 进行声明：

```
@frozen public struct FrozenStruct: Sendable {
    public let value: Int
    public let name: String

    public init(value: Int, name: String) {
        self.value = value
        self.name = name
    }
}
```

对于这样的 struct，它的成员不会再被修改，编译器将有机会直接检查其内部结构并隐式添加 Sendable，而无需担心这个假设在未来失效。然而，需要注意的是，当后续在开发框架时违反 frozen 承诺，对该 struct 的成员进行修改，将会破坏 ABI（应用二进制接口）。

Class 类型的 Sendable 实现

严格的要求

要让 class 类型满足 Sendable，条件则要严苛得多：

1. 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

2. 该 class 类型的成员必须都是 Sendable 的
3. 所有的成员都必须使用 let 声明为不变量

```
final class PersonClass: Sendable {  
    let name: String  
    let message: String = ""  
  
    init(name: String) {  
        self.name = name  
    }  
}
```

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。

NSObject 和 Foundation 类型

许多 Foundation 框架中的类型并不满足 Sendable，这在迁移过程中可能会遇到挑战：

```
// 这些类型通常不是 Sendable  
// NSMutableArray, NSMutableDictionary, UIViewController, etc.  
  
// 需要小心处理。比如使用串行队列保证线程安全。  
class DataManager {  
    private let queue = DispatchQueue(label: "data.queue")  
    private var cache: NSMutableDictionary = [:]  
  
    func getValue(for key: String) -> Any? {  
        queue.sync { cache[key] }  
    }  
}
```

对于这类情况，通常需要：

1. 使用 Swift 原生的 Sendable 类型替代
2. 通过合适的同步机制确保线程安全
3. 在必要时使用 @unchecked Sendable

我们会在后面的章节中详细讨论 @unchecked Sendable 的使用场景和注意事项。

Actor 类型的自动 Sendable

虽然 actor 和 class 一样都是引用类型，但是 actor 内部的隔离机制确保了内部状态的安全性。在不同的并发域中对 actor 成员的访问最终都会发生在该 actor 的隔离域中。因此，所有的 actor 类型都可以自由地在并发域之间传递，并且它们都满足 Sendable 的要求：

```
actor DataProcessor: Sendable { // 隐式满足 Sendable
    private var cache: [String: Data] = [:]
    private var isProcessing: Bool = false

    func process(_ data: Data) async -> ProcessedData {
        // actor 的隔离机制保证了线程安全
        isProcessing = true
        defer { isProcessing = false }

        // 处理逻辑...
        return ProcessedData(data)
    }
}
```

不论 actor 位于哪个模块，也不论 actor 拥有什么类型的存储成员，编译器都会为其自动添加 Sendable。

处于全局 Actor 下的类型

如果一个 class 被全局 actor（比如 @MainActor）标注，那么它也会被自动视为 Sendable。这个推论基于全局 actor 的隔离特性，全局 actor 确保了该类型的所有实例都只能在特定的并发域中访问，从而保证了数据安全。

函数类型和数据传递安全

函数类型的特殊处理

除了具体类型之外，函数也经常在并发域之间传递。在 Swift 中，函数类型也是引用类型，它在函数体内部持有对外部值的引用。当跨越并发域时，被函数体引用的值可能发生变化。这是一种线程不安全的行为，为了确保数据安全，我们需要一套机制来控制函数参数的安全传递。

在 Swift 并发的演进过程中，出现了两种重要的标注方式：@Sendable 和 sending。让我们先从传统的 @Sendable 开始了解。

传统的 @Sendable 函数标注

在 Swift 5.5 引入并发特性时，为了表示函数参数必须满足安全传递的要求，我们使用 @Sendable 对该函数进行标注，这相当是一种针对函数的 Sendable 类型。比如下面两个单纯用来举例的函数：

```
func performSendable(
    operation: @Sendable () -> Void
) {
    operation()
}

func performSendableAsync(
    operation: @Sendable () async -> Void
) async {
```

```
    await operation()  
}
```

在使用这些被标记为 @Sendable 的函数时, 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, 都必须声明为不变量

这其实和普通类型的 Sendable 要求是一致的。比如下面这个例子在严格模式下会编译失败:

```
class NonSendableData {  
    var name: String = ""  
    func process() -> String { "processed: \(name)" }  
}  
  
let data = NonSendableData()  
performSendable {  
    // ❌ 编译错误: NonSendableData 无法在 @Sendable 闭包中使用  
    let result = data.process()  
}
```

这些额外限制带来了更严格的类型安全检查以及绝对安全的并发性, 但同时它也是迁移到 Swift 并发模型的最困难的部分之一: 一个函数的 @Sendable 标注往往会造成大规模的“传染”, 它会要求其中的变量全都是 Sendable, 有时这意味着巨大的工作量, 甚至可能让迁移过程失控。

Swift 6 的突破性进展: sending 关键字

Swift 6 引入了 sending 关键字 (来自 SE-0430 提案), 为数据传递提供了更加灵活的解决方案。与 @Sendable 要求函数类型本身完全必须是线程安全不同, sending 其实是一个所有权的关键字, 它关注的是传递过程的安全性:

```
func performSending(  
    // 注意我们换成了 sending
```

```
operation: sending () -> Void
) {
    operation()
}

// 现在可以安全地传递非 Sendable 类型了
func modernProcessing() {
    let data = NonSendableData()
    data.name = "example"

    // ✅ 使用 sending 关键字，通过所有权转移保证安全
    performSending {
        // data 的所有权被转移
        let result = data.process()
        print(result)
    }

    // 编译器确保这里不能再使用 data
    // ❌ 编译错误！所有权已转移
    // data.name = "changed"
}
```

sending 的核心思想是“所有权转移”。当我们使用 sending 时：

1. 该值的所有权完全转移给接收方
2. 原始持有者不能再访问该值
3. 编译器确保没有其他引用同时持有该值

由于所有权整个被转移了，因此不再会有代码能并发访问这个参数，这让我们可以在保证安全的前提下，传递那些本身不是 Sendable 的类型。这在大多数情况下都是我们想要的行为，也大幅简化了迁移的难度。

Swift 6 中的 API 演进

Swift 6 对许多核心并发 API 进行了重大更新，引入了更精细的控制机制，将很多以前的 @Sendable 函数改为了 sending。让我们看几个最重要的变化：

Task.init 的签名演进

Swift 5.5 时代：

```
struct Task {  
    init(  
        priority: TaskPriority? = nil,  
        operation: @escaping @Sendable () async -> Success  
    )  
}
```

Swift 6.0 现在：

```
extension Task where Failure == Never {  
    @discardableResult  
    init(  
        priority: TaskPriority? = nil,  
        operation: sending @escaping @isolated(any)  
            () async -> Success  
    )  
}
```

关于 @isolated(any)，我们在上一章已经介绍过了，它会将函数的隔离信息保留下。在新版 API 签名中，sending 代替了原来的 @Sendable，前者通过所有权转移，允许传递捕获了非 Sendable 值的闭包，相比原先的方案要灵活得多。

这意味着我们现在可以这样使用：

```
@MainActor
```

```
func updateUIAfterTask() {
    let data = NonSendableData()
    data.name = "UI Update"

    // Swift 6: 直接传递非 Sendable 数据, 同时保持 MainActor 隔离
    Task {
        // 所有权转移, 安全执行
        let result = data.process()

        // 原有的 @MainActor 被保留了 (通过 @isolated(any))
        print("Result: \(result)")
    }
}
```

TaskGroup.addTask 的同步更新

TaskGroup.addTask 也采用了相同的现代化签名:

```
struct TaskGroup<ChildTaskResult> {
    mutating func addTask(
        priority: TaskPriority? = nil,
        operation: sending @escaping @isolated(any)
            () async -> ChildTaskResult
    )
}
```

这让我们能够在 TaskGroup 中更灵活地处理数据:

```
func processMultipleItems() async {
    let items = [NonSendableData(), NonSendableData(), NonSendableData()]

    await withTaskGroup(of: String.self) { group in
        for (index, item) in items.enumerated() {
            item.name = "Item \(index)"
        }
    }
}
```

```
// 每个 item 通过 sending 安全地转移给对应的任务
group.addTask {
    return item.process() // 所有权转移，并发安全
}
}

for await result in group {
    print(result)
}
}
}
```

Error 类型的 Sendable 要求

另一类在并发域中传递的重要类型是各种错误值。当异步函数发生错误时，我们使用 `throws` 关键字来抛出错误，从语义上讲，这相当于将错误返回给调用者。基于安全性要求，`Error` 类型应始终满足 `Sendable` 的要求：

```
protocol Error : Sendable {
}
```

破坏性变更的处理

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：

```
class Detail {
    var text: String = ""
}

struct SomeError: Error {
    var detail: Detail // 问题：Detail 不是 Sendable
```

```
}
```

这样的代码会因为 `detail.text` 无法被保护, 从而无法满足 Sendable。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。但是如果你打开了严格检查和 Swift 6 的语言模式, 那么你就必须想办法处理这样的问题, 并让 `SomeError` 满足 Sendable 的要求。

将 class 转变为 Sendable

在 Swift Concurrency 迁移时, 一个最常见的任务就是将包括可变成员的 class 的非安全类型转换为 Sendable。在本章早些时候, 我们已经提到过想让 class 满足 Sendable 所需要的严格条件, 要满足这些条件并不容易, 特别对于在 Swift Concurrency 引入之前就编写的代码来说, 更是如此。

如果 class 中已经基于某些机制实现了完整的线程安全, 那么我们可以为它添加 `@unchecked Sendable`, 从而在编译阶段跳过对该类型的 Sendable 检查。不过这么做的前提是开发者需要确保该类型在并发环境下的安全性, 并需要自行承担若该类型实际上并不满足线程安全时所产生的风险。另外, 这种情况下, 新加入的成员即使不满足 Sendable 的要求, 编译器也不再会进行检查和报错。

当然, 如果有条件, 也可以将整个 class 转变为 actor, 这样就可以自动满足 Sendable 的要求, 并且可以享受 actor 提供的并发安全性, 但这通常需要对代码进行较大的重构。在本节中, 我们将重点聚焦在处理现有的 class 类型上。

使用 OSAllocatedUnfairLock

如果你的项目支持 iOS 16.0+, 那么使用 `OSAllocatedUnfairLock` 是一个高效且优雅的选择。它是一个轻量级的锁实现, 专为高性能并发设计。将 class 中需要保护的可变量放到 `OSAllocatedUnfairLock` 中, 并通过 `withLock` 来访问或修改其中的值:

```
import os

// 使用 OSAllocatedUnfairLock 确保线程安全
```

```
final class UnfairLockProtectedCounter: Sendable {
    let counter = OSAllocatedUnfairLock(initialState: 0)

    func increment() {
        counter.withLock { state in
            state += 1
        }
    }
}
```

由于 OSAllocatedUnfairLock 本身满足 Sendable，且 counter 在 class 内是不变量，因此整个 UnfairLockProtectedCounter class 也满足 Sendable 的要求。我们可以将这个 class 直接标记为 Sendable，这会为我们提供面向未来的更可靠的编译保证。

使用 @unchecked Sendable

在 Swift 并发之前，为了保证 class 成员的数据安全，你的代码很可能已经有了一些方案，比如加锁或者在内部设置串行调度队列。这些方案可以在 class 自身内部提供数据安全保证，因此它们可以安全地在不同的并发域之间传递。

然而，由于使用的是运行期间的特性（比如锁和队列），在编译期间，编译器无法静态地确定这些类型的安全性，也就无从直接通过声明让它们满足 Sendable 的要求。如果我们确信某个类型是安全的，可以在 Sendable 之前加上 @unchecked 标记跳过检查。下面是一些常见的手段：

使用传统锁的线程安全类型

```
class LockProtectedCounter: @unchecked Sendable {
    private var _value: Int = 0
    private let lock = NSLock()

    var value: Int {
        lock.lock()
        defer { lock.unlock() }
        return _value
    }
}
```

```
}

func increment() {
    lock.lock()
    defer { lock.unlock() }
    _value += 1
}
}
```

使用 NSLock 时要特别小心, 多次调用 lock() 而忘记 unlock() 会让你的代码陷入死锁。保持锁的粒度尽可能小, 并确保在所有可能的代码路径中都能正确释放锁。如果对性能不十分在意, 也可以使用 NSRecursiveLock, 它允许同一个线程多次获取锁而不会死锁。

使用队列的串行化访问

```
class QueueProtectedData: @unchecked Sendable {
    private var _data: [String] = []
    private let queue = DispatchQueue(label: "data.queue")

    func addItem(_ item: String) {
        queue.sync {
            self._data.append(item)
        }
    }

    func getItems() -> [String] {
        return queue.sync {
            return _data
        }
    }
}
```

使用串行队列可以确保对 `_data` 的访问是线程安全的。通过 `sync` 方法, 我们可以在队列中同步执行代码块, 从而避免数据竞争。但是, 和使用锁一样, 使用队列时也要小心避免死锁: 比如不小心在 `queue.sync` 中再次调用了包含 `queue.sync` 的代码, 会导致死锁。

另外, 队列派发的性能开销可能会比直接使用锁更高, 在关键场景下, 也可以考虑并发队列和 `barrier` 方法来实现更高效的线程安全。但这会需要更加精细的队列调度设计, 这往往不容易做对。

其他选择和风险

确保 `Sendable` 的类型安全的方式还有很多, 比如属性的原子操作 (Apple 为此专门提供了一个 `swift-atomics` 库), 比如其他的各种锁。但不论选择哪一种, 最终在将 `class` 声明为 `@unchecked Sendable` 时, 都需要十分谨慎:

- **开发者责任:** 通过使用 `@unchecked`, 你可以让任意类型都被认为是 `Sendable`, 但错误的实现可能会引入 bug
- **运行时风险:** 编译器跳过了安全检查, 数据竞争问题可能在运行时出现
- **维护成本:** 需要确保在类型演进过程中始终保持线程安全, 特别是后续添加新成员时, 要注意 `@unchecked` 标记将阻止编译器进一步检查

在有条件时, 将 `class` 转变为 `actor` 或者至少使用 `OSAllocatedUnfairLock` 是更推荐的做法。只有在无法避免的情况下, 才使用 `@unchecked Sendable`, 并确保有充分的测试覆盖来验证其线程安全性。

并发检查级别和迁移清单

检查级别的演进

Swift 通过 `SWIFT_STRICT_CONCURRENCY` 编译器标志提供了三个并发检查级别, 让开发者可以根据项目情况选择合适的严格程度:

- **Minimal (最小级别)**: 只检查最基本的并发安全问题, 主要关注明显的数据竞争和线程安全违规。这个级别允许大部分现有代码继续编译通过, 适合刚开始迁移到 Swift 并发模型的项目。在这个级别下, 只有明确标记了 Swift 并发特性的代码 (如 @MainActor) 才会被检查。
- **Targeted (目标级别)**: 对使用了 Swift 并发特性的代码进行严格检查, 包括 `async/await`、`actor` 等, 但对未采用并发特性的旧代码保持宽松。这个级别会检查 `Sendable` 协议的正确使用, 确保 `actor` 隔离的正确性。
- **Complete (完全级别)**: 对所有代码进行最严格的并发安全检查。要求所有跨并发域传递的类型都必须是 `Sendable`, 检查所有全局变量和静态属性的线程安全性, 强制执行完整的 `actor` 隔离规则。这个级别可能需要大量代码修改才能通过编译, 适合新项目或已完全迁移到 Swift 并发的项目。

启用严格检查

在当前版本中, 可以通过构建设置来启用严格检查:

```
# Xcode Build Settings
SWIFT_STRICT_CONCURRENCY = complete

#      Package.swift
.addTarget(
    name: "MyTarget",
    swiftSettings: [
        .enableUpcomingFeature("StrictConcurrency")
    ]
)
```

传统项目的初始设置是 `Minimal`, 这意味着只有使用了 Swift 并发特性的代码才会被检查。随着项目逐渐迁移到 Swift 并发模型, 可以逐步将检查级别提高到 `Targeted` 甚至 `Complete` 级别, 以获取完整的检查。

常见错误和应对清单

当你开始在项目中启用Swift 6的严格并发检查时, 很可能会遇到大量的编译错误。不要慌张——这些错误实际上帮助你发现了代码中潜在的并发安全问题。在这一节中, 我们将一起探讨最常见的并发错误类型, 并学习如何逐一解决它们。

1. 全局变量: 并发安全的“雷区”

全局变量是并发编程中的一个重要风险点。考虑这样的代码:

```
// 错误: 全局可变状态在严格模式下不被允许
var globalCounter = 0

func incrementGlobal() {
    globalCounter += 1 // ✗ 错误: 全局变量的访问需要同步保证
}
```

这段代码的问题是: 任何线程都可以同时访问 `globalCounter` 变量, 如果在多线程环境下同时读写, 就会造成数据竞争。

解决方案是使用 `actor` 隔离保护这些全局状态:

```
// 解决方案: 使用全局 actor 隔离 (或 @MainActor, 如果合适的话)
@CounterActor var globalCounter = 0

@CounterActor func incrementGlobal() {
    globalCounter += 1
}
```

2. 单例模式的并发改造

很多项目中都有单例模式, 虽然可能我们已经把单例本身标记为 `let` 了, 但它的内容在并发环境下还是可能被同时改变且访问的:

```
final class ConfigurationManager {  
    static let shared = ConfigurationManager()  
    var settings = [String: Any]() // ✗ 全局可访问但内部状态可变  
  
    private init() {}  
}
```

即使单例本身是线程安全创建的，但如果它包含可变状态，仍然存在并发问题。解决方案是将整个类型设计为线程安全的：

```
// 方案1：使用MainActor（适合UI相关的配置）  
@MainActor  
final class ConfigurationManager {  
    static let shared = ConfigurationManager()  
    var settings = [String: Any]()  
  
    private init() {}  
}  
  
// 方案2：转换为actor（适合独立的数据管理）  
actor ConfigurationManager {  
    static let shared = ConfigurationManager()  
    var settings = [String: Any]()  
  
    private init() {}  
}
```

3.UIKit 访问：最常见的“拦路虎”

如果你的项目是个 app，并和 UI 相关，那当你启用严格并发检查后，最先遇到的很可能是与 UI 框架相关的错误。让我们看一个典型的例子，比如下面这个 controller（注意，它并不是 ViewController 的子类）：

```
final class AnimationController: NSObject {
```

```
private(set) lazy var animationView: UIView? = {
    guard animationEnabled else { return nil }
    return UIView()
}()

func startAnimation() {
    guard let animationView else { return }
    animationView.alpha = 0.0
    UIView.animate(withDuration: 0.3) {
        animationView.alpha = 1.0
    }
}
}

// ✖ 编译器会告诉你：UIView 需要在 MainActor 上访问
```

这个错误的根本原因是：在 Swift 6 中，所有 UIKit 类（包括 `UIView`、`UIViewController` 等）都被标记为 `MainActor` 隔离。这意味着任何访问这些类的代码都必须运行在主线程上。

如果 `AnimationController` 所控制的所有内容都是 UI 相关的，那解决方案就很直接了——将整个类标记为 `@MainActor`：

```
@MainActor
final class AnimationController: NSObject {
    private(set) lazy var animationView: UIView? = {
        // ...
    }

    func startAnimation() {
        // ...
    }
}
```

需要注意的是，这种修改会产生“连锁反应”。一旦将某个类标记为 `@MainActor`，所有调用这个类的代码也需要相应调整。这正是为什么严格并发检查会产生大量错误的原因：它们往往是

相互关联的。最终，如果我们明确某个“入口”不应该被 MainActor 隔离，那么我们就需要使用 Task 来开启一个异步运行环境，去执行主线程的代码。

4.Sendable协议：类型安全的“门票”

当你尝试在不同 actor 之间传递数据时，Swift 要求这些数据类型是 Sendable 的。非 Sendable 类型被传递时，我们会看到这个常见错误：

```
class UserData { // 注意：没有声明Sendable
    var name: String = ""
    var age: Int = 0
}

actor DataManager {
    var userData = UserData()
}

Task {
    let manager = DataManager()
    _ = await manager.userData // ✗ UserData不能跨越actor边界
}
```

问题在于：UserData 类型没有声明为 Sendable，因此不能在不同的 actor 之间安全传递。如果是简单的值类型，解决方案很直接，加上 Sendable 即可（在上面我们提到过，对于同一个模块，编译器甚至会为我们自动添加）：

```
struct UserData: Sendable { // 添加Sendable协议
    var name: String
    var age: Int
}
```

但对于类类型，要求就严格得多：

```
final class UserData: Sendable {
```

```
var name: String // ✗ Sendable类不能有可变存储属性  
let age: Int  
}
```

这时你有几个选择：

```
// 选择 1：对于不需要引用语义的类型，尽可能改用 struct  
struct UserData: Sendable {  
    var name: String // struct可以有可变属性  
    var age: Int  
}  
  
// 选择2：改用 actor  
actor UserData {  
    var name: String // actor 天然是Sendable的  
    var age: Int  
}  
  
// 选择3：使用锁保护（复杂但有时必要）  
final class UserData: Sendable {  
    private let nameSync = OSAllocatedUnfairLock(initialState: "")  
  
    var name: String {  
        get { nameSync.withLock { $0 } }  
        set { nameSync.withLock { $0 = newValue } }  
    }  
}
```

我们本章中也介绍了一些其他标记 Sendable 的方式，可以进行参考。

5. 协议与实现的“身份不符”

另一个常见问题是协议定义与具体实现之间的 actor 隔离不匹配：

```
// 协议定义
public protocol UserInterfaceService {
    func presentViewController(on parent: UIViewController)
}

// 具体实现
struct DefaultUserInterfaceService: UserInterfaceService {
    @MainActor func presentViewController(
        on parent: UIViewController
    ) {
        let viewController = UIViewController()
        parent.present(viewController, animated: true)
    }
}
// ✗ 协议要求非隔离函数，但实现却被 MainActor 隔离了
```

问题在于：协议声明了一个普通函数，但实现需要访问 UIKit，因此必须在 MainActor 上运行。解决这个矛盾的方法是在协议定义阶段就考虑好并发需求。比如如果协议要求的方法本身就应该被 MainActor 隔离的，那么我们在定义它时就加上 @MainActor：

```
public protocol UserInterfaceService {
    @MainActor func presentViewController(on parent: UIViewController)
}

struct DefaultUserInterfaceService: UserInterfaceService {
    @MainActor func presentViewController(on parent: UIViewController) {
        let viewController = UIViewController()
        parent.present(viewController, animated: true)
    }
}
```

对于更一般的协议，如果我们希望它们能够在不同的 actor 中使用，可以考虑将方法定义为 `async`。同步版本的函数可以满足 `async` 版本的协议要求。这样一来，不论是在隔离域中的类型，还是没有明确隔离的类型，都可以安全地实现这个协议：

```
protocol Foo {
    func doSomething() async
}

actor Bar: Foo {
    func doSomething() {
        print("Doing something in Bar")
    }
}

class Baz: Foo {
    func doSomething() {
        print("Doing something in Baz")
    }
}
```

这提醒我们：在今后设计 API 时要提前考虑并发需求，而不是事后补救。

6. 闭包捕获：异步编程的“陷阱”

在异步编程中，闭包经常需要符合 `Sendable` 协议，这时其中就不能捕获非 `Sendable` 对象。这有时会是一个非常严格且棘手的问题，而且为项目中不必要的隔离带来大量错误和传染性。

假设我们有这样两个类型：

```
class PerformanceLogger {
    func start() { }
    func stop() -> TimeInterval { 0.0 }
}
```

```
class DataRepository {  
    func fetchData(completion: @escaping @Sendable () -> Void) {}  
}
```

在使用它们时，fetchData 所要求的 @Sendable 闭包中，我们不能直接使用 PerformanceLogger，因为它不是 Sendable 类型：

```
func performTask(_ completion: @escaping @Sendable () -> Void) {  
    let logger = PerformanceLogger() // 假设这是非 Sendable 类型  
    let repository = DataRepository()  
  
    logger.start()  
    repository.fetchData { // 这个闭包需要是Sendable的  
        // ✗ 不能在 Sendable 闭包中引用非 Sendable 的 logger  
        let duration = logger.stop()  
        completion()  
    }  
}
```

当然，我们可以想办法让 PerformanceLogger 满足 Sendable。但最优雅和更简单的解决方案是将基于回调的 API 转换为 async/await，这样我们可以避免闭包捕获的问题：

```
func performTask() async {  
    let logger = PerformanceLogger()  
    let repository = DataRepository()  
  
    logger.start()  
    await repository.fetchData() // 使用async/await，无需闭包  
    let duration = logger.stop() // 现在可以安全访问logger  
}
```

这不仅解决了 Sendable 问题，还让代码更易读。

7. 危险的”快速修复”: 避免使用`assumesIsolated`

当面对大量并发错误时, 你可能会想寻找“快速修复”的方法。MainActor.assumesIsolated 看起来像是一个救星:

```
func createViewController() -> UIViewController {
    MainActor.assumeIsolated {
        UIViewController()
    }
}
// ! 危险: 如果实际不在MainActor上运行, 会导致运行时崩溃
```

建议不要这样做! assumesIsolated 毕竟是一个不安全的API, 如果你的假设错误, 应用程序会在运行时崩溃。正确的做法是使用适当的 actor 标注:

```
@MainActor
func createViewController() -> UIViewController {
    UIViewController()
}
```

循序渐进的迁移策略

在实际项目中, 面对这么多并发错误, 一开始你可能感到不知所措。不过这里有一些实用的迁移建议, 也许能帮助你快速上手。在迁移到 Swift 6 的严格并发检查之前, 就可以先从以下几个方面入手改善现有代码库:

立即可以开始做的事情

1. 为所有 UI 相关代码添加 `@MainActor` - 这是最直接, 也是最安全的改进
2. 将回调式 API 逐步改为 `async/await` - 这能解决很多 Sendable 相关的问题
3. 让简单的数据类型符合 `Sendable` - 从简单的 `struct` 开始, 逐步扩展到所有类型
4. 避免创建新的全局可变状态 - 新代码应该从一开始就考虑并发安全

迁移策略

总结本书所提及的内容，我们可以按照下面的原则拟定迁移策略。同时注意避免一些不好的做法：

1. 从独立模块开始 - 避免一次性修改整个项目造成的级联影响
2. 先处理 Sendable，再处理 actor 隔离 - Sendable 的影响通常较小
3. 逐模块启用严格检查 - 不要试图一次性修复所有问题，而是逐步提高各个模块的检查级别

需要避免的做法

1. 不要滥用 `@unchecked Sendable` - 除非你完全理解其含义和风险
2. 不要用 `assumesIsolated` 来“快速修复” - 这只是把问题推迟到运行时
3. 不要在同步接口中隐藏异步操作 - 这会让并发问题更加隐蔽

记住，严格并发检查产生的错误数量可能很惊人（大型项目中经常是数千个），但这些错误实际上揭示了代码中真实存在的并发安全问题。修复它们不仅让你的代码符合 Swift 6 的要求，更重要的是让你的应用程序实际上也更加稳定和安全。

这个过程需要耐心和系统性的方法，但最终的收益——更安全、更清晰的并发代码。我认为相对于收益这些努力还是值得的。

并发线程模型

11

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。

虽然无论代码位于同步函数还是异步函数中，最终都需要由某个具体线程来执行，但异步函数与一直在同一线程上运行的同步函数不同，异步函数可能被 await 分割，并由多个不同的线程协同运行。Swift 并发在底层采用了一种新的调度方式，称为 **协同式线程池** (cooperative thread pool)：它使用一个串行队列来调度工作，将函数中剩余的执行内容抽象为轻量级的续体 (continuation)，然后进行调度。实际的工作运行则由一个全局的并行队列来处理，具体的任务可能会被分配到最多与 CPU 核心数量相等的线程中去。

需要强调的是，协同式线程池限制线程数量的目的是为了避免线程级别的切换，从而避免性能问题。在 Swift 并发中，续体代表了一个运行时的状态包，当使用 await 时，函数的剩余部分被“注册”为一个续体并暂存起来，然后在某个工作线程上执行当前的语句。Swift 并发的运行时可以轻松地在多个续体之间进行切换，类似于一个轻量级的线程。其他支持并行的语言中也有类似但不完全相同的概念，例如 Go 中的 Goroutines，Rust 中的 tokio 中的 task，或者 Crystal 中的 fiber。有时我们也会使用绿色线程 (green thread)、协程 (coroutine) 或者纤程 (fiber) 来称呼这些概念。尽管它们的涵盖范围略有不同，但核心思想是一致的：它们提供了一种与系统级线程不同、更轻量级的调度方式。虽然续体之间的切换比线程切换要轻量得多，但

并非完全无开销。在实际使用中，频繁的 `await` 调用仍然会产生一定的性能成本，特别是在自定义执行器的场景下。Apple 在 Swift 6 中引入了新的执行器优化机制，但在某些情况下自定义执行器可能比默认执行器慢 10-15%。因此，在性能敏感的场景中，我们需要仔细权衡异步调用的频率和粒度。

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

在引入 Swift 并发之前，GCD 是最主流的线程调度方式。它采用“抢占式”调度，管理一个线程池。在需要时，GCD 尝试从线程池中获取已创建但闲置的线程。然而，当有需要时或线程池中没有可用线程时，GCD 也将尝试创建新的线程。某个线程可能会被耗时操作占用或需要等待某个锁，这会导致该线程处于被阻塞的状态，无法被分配。在这种情况下，当有新任务需要处理时，新的线程就将被创建，并分配给某个 CPU 核心，以执行新任务中的指令。

在理想情况下，如果正在运行的线程数小于或等于 CPU 的核心数，每个线程将被分配到一个独立的核心上，这样该核心可以专注地执行和处理该线程中的指令。然而，当线程数量超过核心数时，新创建的线程将会被分配给已经在运行其他线程的核心。这时，一个 CPU 核心将同时处理两个或多个线程。

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。线程可以共享所在进程的内存堆等资源，同时也拥有自己的一些资源，如调用栈、寄存器环境和动态申请的栈空间上的内存。CPU 核心只是一个简单的指令执行器，因此在同一个核心上同时执行多个线程需要通过时分复用的策略来让这些线程共享计算资源。这意味着在运行不同的线程时，执行环境需要从一个线程切换到另一个线程。这种切换涉及整个线程资源的切换，包括寄存器、栈指针和栈内存等。虽然线程切换相对轻量，但仍然需要消耗时间：

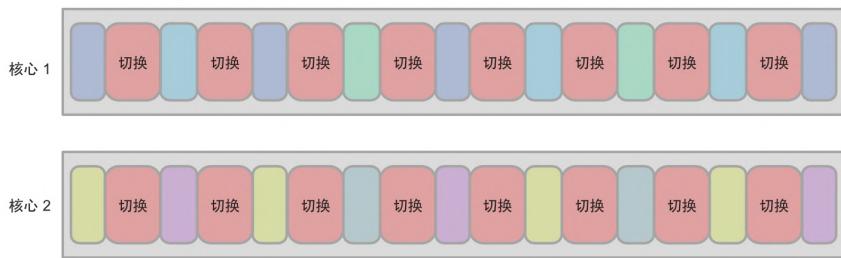


在 GCD 中，调度库对并行队列和串行队列能够创建的线程总数进行了限制。这个限制在不同的运行环境下会有所不同。就目前已发布的最新 iOS 系统和最新的硬件环境而言，单个并行队列最多可以创建 64 个线程，而串行队列可以创建 512 个线程。移动设备的 CPU 核心数和内存容量都是有限的，因此无法承载无限多的线程。这也是为什么苹果在文档中要求我们避免创建过多线程的原因，限制线程数量可以避免资源的过度占用和性能下降。

然而，在现代 Apple Silicon 设备上，情况变得更加复杂。这些设备采用异构架构，拥有不同类型的 CPU 核心。例如，M3 芯片包含高性能核心 (P-core) 和高效核心 (E-core)，系统会根据任务的 QoS 优先级智能地分配工作负载：低优先级的后台任务会被分配到 E-core 上运行，而用户交互相关的任务会优先使用 P-core。这种架构使得传统的线程数量限制变得不那么绝对，因为系统可以更灵活地管理不同类型的工作负载。

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。

当存在过多线程时，不仅会增加内存压力，更重要的是，这些线程在有限的 CPU 核心上运行，导致频繁的线程上下文切换。有时候，与实际执行需要的指令相比，这些上下文切换所消耗的资源和时间反而占据了主要部分。这种情况下，由于线程被阻塞的同时，又不断有新的任务以 `async` 方式提交到并行队列，导致过多的新线程被创建，我们将其称为**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，将“线程”这个概念完全隐藏在幕后。然而，实际上，无论是使用 Task 相关的结构化任务 API 进行调度，还是在不同的 actor 隔离域之间进行切换，都涉及到执行线程的问题。事实上，如果不对现有的线程调度方式进行革新，以支持 Swift 并发的新 API，在底层可能会增加更多潜在的线程切换机会。这种切换所带来的性能问题可能会限制 Swift 并发的广泛应用。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.
```

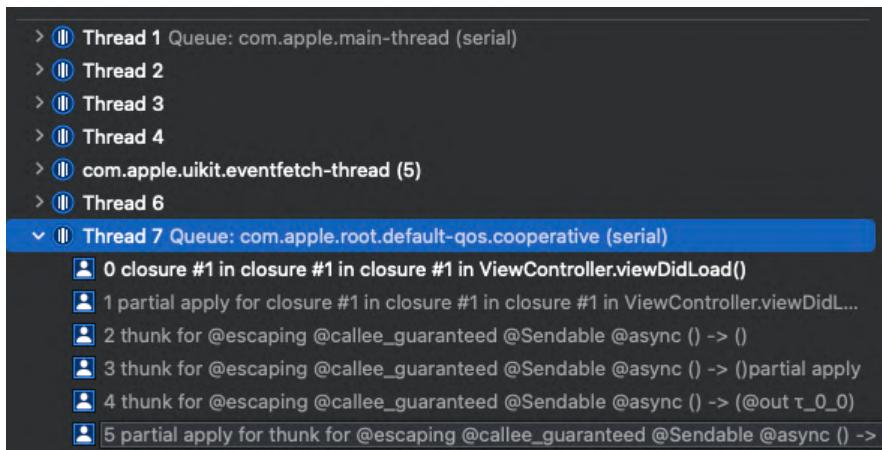
前 64 个派发的闭包被分配到了各自的执行线程上，但它们永远卡在了等待信号上：因为这些发出信号的工作自身也在等待可以使用的线程，但所有可用线程都在等待信号，新的可用线程将永远不会出现。这类问题会非常难调试，还可能随着运行环境的不同而产生不同的现象。

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。当可用线程数不够多时，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，我们又只想要较少的线程数。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

Swift 并发在所有平台上底层都使用 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：

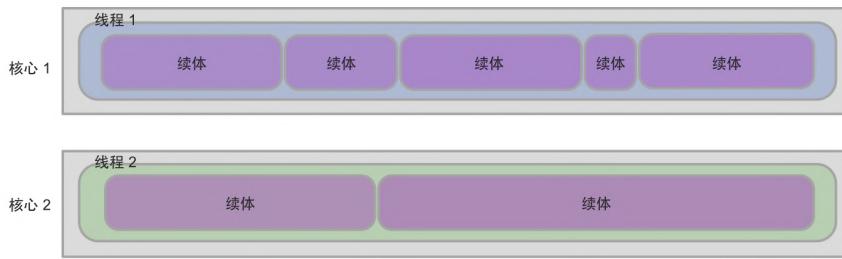


The screenshot shows a stack trace from Xcode. Thread 7 is highlighted with a blue background. The stack trace includes:

- > ⓘ 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.viewDidLoadL...
 - 👤 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 τ_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，协同式队列的调度中实现了额外的能力，它可以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。这样一来，我们在同一个核心中，就只有续体的轻量级切换，而避免了线程级别的切换：



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()
}

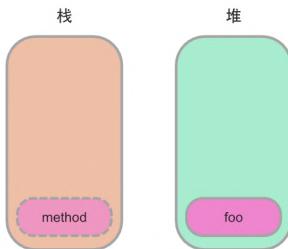
func baz() async {
    bar1()
}

func foo() async {
    bar1()
    await bar2()
    await bar3()
}

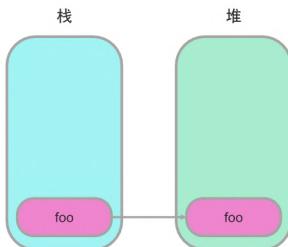
func method() {
    Task {
        await foo()
    }
}
```

}

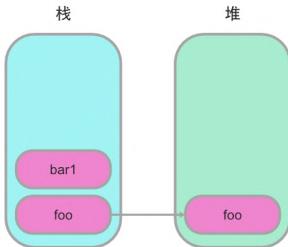
- 当某个线程执行 method 时，Task.init 首先被入栈，它是一个普通的初始化方法，在执行完毕后立即出栈，method 函数随之结束。通过 Task.init 参数闭包传入的 await foo()，被提交给协同式线程池，如果协同式调度队列正在执行其他工作，那么它被存放在堆上，等待空闲线程：



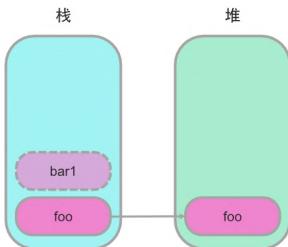
- 当有适合的线程可以运行协同式调度队列中的工作时，执行器读取 foo 并将它推入这个线程的栈上，开始执行。需要注意的是，这里的“适合线程”和 method 所在的线程并不需要一致，它可能是另外的空闲线程 (因此我们这里使用的颜色和上面的栈有所不同)：



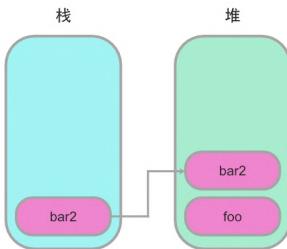
- foo 中的第一个调用是一个同步函数 bar1。在异步函数中调用同步函数并没有什么不同，bar1 将被作为新的 frame 被推入栈中执行：



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():

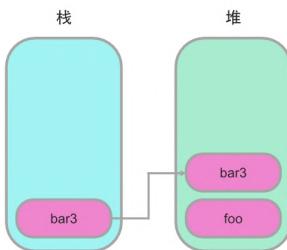


5. 接下来我们会在一个线程中执行到 await bar2(), 它是一个异步函数调用。为了不阻塞当前线程, 异步函数 foo 可能会在此处暂停并放弃线程。当前的执行环境(如 bar2 和 foo 的关系)会被记录到堆中, 以便之后它在调度栈上继续运行。此时, 执行器有机会到堆中寻找下一个需要执行的工作。在这里, 我们假设它找到的就是 bar2。它将被装载到栈上, 替换掉当前的栈空间, 当前线程就可以继续执行, 而不至于阻塞了:



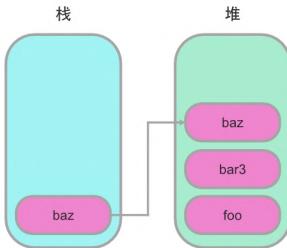
当然，执行器也有可能寻找到其他的工作（比如最近有优先级更高的任务被加入），这种情况下 `bar2` 就将被挂起一段时间，直到调度栈有机会再次寻找下一个工作。不过不论如何，串行调度队列都不会停止工作。它要么会去执行 `bar2`，要么会去执行其他找到的工作，唯独不会傻傻等待。

- 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：

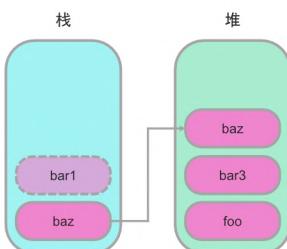
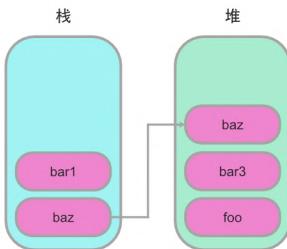


需要注意，`await bar2()` 前后代码可能会运行在不同线程上，除非指定了 `MainActor`，否则协作式调度队列并不会对具体运行的线程作出保证。

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：



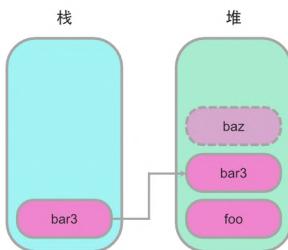
8. 在这个栈中，同步方法 bar1 的调用依然在当前栈上进行普通的入栈和出栈：



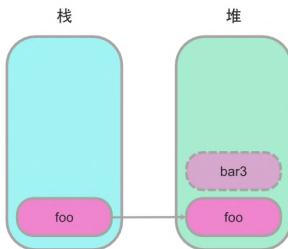
在异步函数定义的栈上调用同步函数，所执行的是普通的出入栈操作。因此在 Swift 异步函数中，我们是可以透明地调用 C 或者 Objective-C 的同步函数的。在底层，这种调用就是普通的同步调用。

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

调度库将续体存储在堆上，并在需要时使用它来替换调度队列线程的运行栈，这是异步函数具备放弃线程能力的基础。当调度线程空闲时 (例如在 await 之后)，执行器会为其查找下一条要执行的指令，这个指令可能是与之前的任务相关的 await 部分，也可能是完全不相关的其他任务。

与传统的抢占式 GCD 调度方式不同，这种调度方式采用协作的方式，由执行器、待处理的工作和调度队列共同确保线程的前进。因此，我们将其称为**协同式线程池**。

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")
    }
}
```

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。在这个例子中，全部 `blockingMethod` 执行完毕需要一万秒，而不是你想象中的一秒！

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。

在下面我们将具体了解 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。

目前，Swift 并发主要提供了以下执行器：

1. 全局的并发执行器，它负责寻找适合的并发队列来为并发操作提供线程。这个全局执行器会根据系统的资源和需求，动态地管理线程的创建和调度，以实现并发操作的执行。
2. 串行执行器，主要用于 actor。每个 actor 都会持有一个串行执行器，它负责确保 actor 隔离域中的方法在串行队列中按顺序执行。这个串行执行器保证了 actor 的内部状态的一致性，避免了多线程并发访问导致的数据竞争问题。
3. 其他的自定义执行器，允许开发者根据特定需求实现自己的执行器。这些自定义执行器可以用于特定的线程模型或性能优化场景：如 TaskExecutor 自定义 Task 的执行环境，RunLoopExecutor 基于程序 run loop 定义执行器，SchedulableExecutor 用于定时任务和取消等。

我们会先介绍全局并发执行器，因为它相对特殊。在稍后的章节，我们再仔细看看其他几种执行器的自定义方法。

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 或更底层完成，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。

GCD 代码性能

在串行调度队列中，我们提到了应避免耗时的同步调用。如果我们能将这些同步调用封装起来，并传递给全局执行器，让它负责通过并发队列将这些调用派发到各个工作线程，那么我们就能够解决这个问题。然而，不幸的是，目前这种派发方法尚未对外开放给开发者使用。

我们可以创建自定义的 TaskExecutor 来改变默认行为，这能让我们能够更好地控制任务的执行环境，特别是在需要与特定线程模型集成时，能有更稳定的调度方式：

```
// Swift 6 中使用自定义执行器处理耗时任务
// CustomTaskExecutor 是满足 TaskExecutor 协议的类型
let heavyWorkExecutor = CustomTaskExecutor(
    label: "heavy-work", qos: .userInitiated
)
await withTaskExecutorPreference(heavyWorkExecutor) {
    someHeavyMethod()
}
```

在下一节，我们会看到更多的关于执行器的例子、如何实现一个执行器以及它们的适用场景。虽然上面的做法很优雅，但相关方法需要 iOS 18 以上才能使用。如果你需要支持旧版本，在遇到这种情况时，我们通常会使用 withUnsafeContinuation 或 withCheckedContinuation 来把封装这些同步调用。我们在早先介绍异步函数的章节中已经看到过这两个方法。然而需要特别注意，这两个方法的闭包依然是运行在串行调度队列中的。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()
}
```

```
    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。

自定义执行器

Swift 5.9 开始通过 [SE-0392](#) 提案引入了自定义 actor 执行器，Swift 6 里更是通过 [SE-0417](#) 引入了自定义执行器的完整支持。而在 iOS 26 中，`RunLoopExecutor` 和 `SchedulableExecutor` 等协议的加入，进一步扩展了可定义执行器的范围。各种自定义执行器允许我们控制异步代码的执行环境，这在需要与特定线程模型集成时特别有用。

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: consuming ExecutorJob)
}
```

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)
    }
}
```

在 actor 隔离域外对 actor 的调用会被转换为一次执行器的 enqueue 操作，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。在实际应用中，除了 MainActor 需要被派发到主线程外，在大部分情况下 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。然而，我们不能过于依赖这个假设，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。

TaskExecutor 协议

TaskExecutor 协议是 Swift 6 中新增的协议，它扩展了 Executor 协议。通过实现自定义的任务执行器，我们能为代码提供更灵活和稳定的任务调度，比如为它们指定特性的派发环境。下面是一个自定义 TaskExecutor 的完整示例：

```
final class CustomTaskExecutor: TaskExecutor {
    private let queue: DispatchQueue

    init(label: String, qos: DispatchQoS = .default) {
        self.queue = DispatchQueue(label: label, qos: qos)
    }

    func enqueue(_ job: UnownedJob) {
        queue.async {
            job.runSynchronously(on: self.asUnownedTaskExecutor())
        }
    }

    func asUnownedTaskExecutor() -> UnownedTaskExecutor {
        UnownedTaskExecutor(ordinary: self)
    }
}
```

使用自定义执行器时，可以通过 `withTaskExecutorPreference` 方法指定它：

```
let executor = CustomTaskExecutor(label: "com.example.custom")

await withTaskExecutorPreference(executor) {
    // 这里的代码将在自定义执行器上运行
    await performWork()
}
```

与现有框架集成

自定义执行器的一个重要应用场景是与现有的事件循环框架集成，比如 SwiftNIO 中就有类似的代码：

```
final class EventLoopExecutor: TaskExecutor, SerialExecutor {
    let eventLoop: EventLoop

    init(eventLoop: EventLoop) {
        self.eventLoop = eventLoop
    }

    func enqueue(_ job: consuming ExecutorJob) {
        let job = UnownedJob(job)
        eventLoop.execute {
            job.runSynchronously(on: self.asUnownedTaskExecutor())
        }
    }

    func asUnownedTaskExecutor() -> UnownedTaskExecutor {
        UnownedTaskExecutor(ordinary: self)
    }

    func asUnownedSerialExecutor() -> UnownedSerialExecutor {
        UnownedSerialExecutor(complexEquality: self)
    }

    func checkIsolated() {
        precondition(
            eventLoop.inEventLoop, "Not isolated to this EventLoop"
        )
    }
}
```

通过将业务逻辑绑定到与 I/O 相同的已有的 EventLoop，可以确保没有不必要的上下文切换，从而获得显著的性能提升。

性能考虑

虽然自定义执行器提供了灵活性，但使用时需要注意性能影响。根据实际测试，使用自定义执行器的首次实现可能比非自定义执行器慢约 15%。这主要是因为自定义执行器可能会引入额外的调度开销。因此，在使用自定义执行器时，应该注意：

1. 只在确实需要特定执行环境时使用
2. 尽量复用执行器实例，避免频繁创建
3. 进行性能测试，确保收益大于开销

任务优先级处理

传统 GCD 调度中，在通过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。我们来进行一些假设：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。在线程资源充足的情况下，低优先级任务也能得到运行，它最终会释放这个锁，所以暂时的等待问题不大。但是考虑如果还有另外很多不需要锁的中优先

级任务也在执行，严格按照优先级的话，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)
```

```
// in background task: TaskPriority(rawValue: 9)  
// in background task: TaskPriority(rawValue: 9)  
// 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，也有类似的接受优先级的方法：

```
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。

Task.yield 的深入理解

Task.yield() 是一个协作式的让行机制，它的工作原理包括：

1. 暂停当前任务：将当前任务的续体保存到堆上
2. 重新排队：将任务重新加入到执行器的队列末尾
3. 让出线程：允许其他任务使用当前线程

这个过程是轻量级的，但并非完全无开销：

```
// 性能敏感的循环中的让行策略
var counter = 0
while shouldContinue() {
    // 执行一些工作
    processItem()

    counter += 1
```

```
// 每处理 100 个项目后让行一次
if counter % 100 == 0 {
    await Task.yield()
}
}
```

实际应用场景

任务让行在以下场景中特别有用：

1. 长时间运行的计算任务：

```
func computePrimes(upTo limit: Int) async -> [Int] {
    var primes: [Int] = []

    for number in 2...limit {
        if isPrime(number) {
            primes.append(number)
        }
    }

    // 定期让行，避免阻塞其他任务
    if number % 1000 == 0 {
        await Task.yield()
    }
}

return primes
}
```

2. 事件处理循环：

```
actor EventProcessor {
    private var events: [Event] = []
    private var isRunning = true
```

```
func start() async {
    while isRunning {
        if let event = events.first {
            events.removeFirst()
            await process(event)
        } else {
            // 没有事件时让行，避免空转占用 CPU
            await Task.yield()
        }
    }
}

private func process(_ event: Event) async {
    // 处理事件
}
```

3. 资源密集型操作的批处理：

```
func batchProcess<T>(
    items: [T],
    batchSize: Int = 50,
    process: (T) async throws -> Void
) async throws {
    for (index, item) in items.enumerated() {
        try await process(item)

        // 每处理完一批后让行
        if (index + 1) % batchSize == 0 {
            await Task.yield()
        }
    }
}
```

```
}
```

让行的最佳实践

1. 避免过度让行：频繁的 yield 会增加调度开销
2. 基于工作量让行：根据实际处理的工作量决定让行频率
3. 考虑使用 Task.sleep：对于需要定期执行的任务，使用 Task.sleep 可能更合适

```
// 定期轮询的正确方式
while isPolling {
    await checkForUpdates()

    // 使用 sleep 而不是 yield
    try? await Task.sleep(for: .milliseconds(100)) // 100ms
}
```

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，优先级相关的概念和行为也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，让同一个任务环境使用相同的优先级，就能让派发队列保持简单，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 这么复杂的语句的话，会变得非常繁琐。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，但是它确实十分便利。Swift 并发中提供了一种语法特性，**任务本地值 (task local value)**，它利用了 Task 类型空间来携带一些元数据，可以让我们也可以用类似 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。

@TaskLocal 属性声明

具体来说，对于任意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
```

使用 `WithValue` 设置值

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
```

任务本地值的实现原理

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。

与 SwiftUI 环境值的对比

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）会以环境值的方式从外层 `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。

任务追踪的实际应用

在多任务中追踪

对于任务本地值的用法，Apple 更推荐使用它来进行任务的追踪。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。

为 Swift Testing 提供依赖

除了用于任务追踪外，`@TaskLocal` 的另一个重要应用场景是在测试中进行依赖注入。在并发安全的测试环境中，我们经常需要为特定的测试用例提供 mock 数据或者控制某些环境变量，而 `@TaskLocal` 为此提供了一种优雅且线程安全的解决方案。

以时间相关的测试为例，假设我们有一个依赖当前时间的业务逻辑：

```
// 业务逻辑代码  
struct DateEnvironment {  
    @TaskLocal static var currentDate: () → Date = Date.init  
}  
  
func processOrder() → Order {  
    let now = DateEnvironment.currentDate()  
    return Order(  
        id: UUID(),  
        createdAt: now,  
        status: now.timeIntervalSince1970 > 1640995200 ? .active : .pending
```

```
)  
}
```

在传统的测试方法中，我们可能需要使用全局变量或者单例模式来注入 mock 时间，这在并发测试环境中容易产生竞态条件。而使用 @TaskLocal，我们可以为每个测试任务提供独立的时间环境：

```
import Testing  
  
struct OrderTests {  
    @Test("订单处理 - 固定时间")  
    func testOrderProcessingWithFixedDate() async {  
        let fixedDate = Date(timeIntervalSince1970: 1640995200)  
  
        await DateEnvironment.$currentDate.withValue({ fixedDate }) {  
            let order = processOrder()  
            #expect(order.createdAt == fixedDate)  
            #expect(order.status == .pending)  
        }  
    }  
  
    @Test("订单处理 - 未来时间")  
    func testOrderProcessingWithFutureDate() async {  
        let futureDate = Date(timeIntervalSince1970: 1672531200)  
  
        await DateEnvironment.$currentDate.withValue({ futureDate }) {  
            let order = processOrder()  
            #expect(order.createdAt == futureDate)  
            #expect(order.status == .active)  
        }  
    }  
}
```

这种方式的优势在于每个测试都在自己的任务环境中运行，互不干扰，它们可以安全地并行执行。如果没有 `@TaskLocal`，我们很可能就只有使用 `@Suite(.serialized)` 来把整个 `OrderTests` 标记为串行执行，才有办法稳定地对每个 `currentDate` 进行注入了。

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 提出了一种新的线程调度模型，这也是异步函数可以放弃线程的基础；Swift 通过全局并发执行器，将任务包装并派发到协同式线程池中，在那里完成底层线程的调度。这个线程池的设计避免了不必要的线程切换，它会根据需要创建适量的线程，使线程数不会超过 CPU 核心数，并采用合理的调度策略，确保各个线程不会饥饿。

这些背后的机制共同支持了 Swift 并发。它们相互协作，并在正确使用的前提下（例如避免在调度线程上执行繁重工作，避免混用传统 GCD 派发等），即使在复杂的情况下，Swift 并发也能保持出色的性能。

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

12

Swift 并发的相关特性可以说是自 Swift 诞生以来至今在语言层面上最重要的新特性之一。通过在语言层面上支持异步函数，并实现原生的结构化并发调度，以及引入了 actor 来确保数据同步和内存安全，Swift 在更轻松、更可靠的并发编程方向上迈出了坚实的一步。在 Swift 并发中采用的各种概念大多经过了业界的验证和实践，同时 Swift 团队还吸纳了许多具备丰富相关经验的开发者参与了 Swift 并发的开发过程。Swift 并发的最终发布经历了长时间的讨论和全面的测试，无论是在稳定性还是易用性方面，都可以说 Swift 并发具备相当的保证。

当然，作为一个新引入的特性，Swift 并发在未来仍有许多改进和进化的空间。在本文的最后一章中，我想对 Swift 并发的未来进行一些展望。

更友好的版本要求

在 Apple 的努力下，包括 `async` 和 `await` 在内的异步函数语法，以及 Task 相关的 API 和 actor 类型，都已经被后向兼容到 iOS 13 了。在 iOS 15 以上的新系统中，Apple 重新实现了部分 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。在旧版本里，Swift 异步的相关能力被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。

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。

那些包含在 Foundation 中的方法和成员，比如 URLSession 中的新加入的异步方法，还是只能在 iOS 15/macOS 12 或更新的系统中使用。因此，想要完全释放 Swift 并发编程所带来的便利和潜力，我们最好还是将最低支持系统提升到 iOS 15 或以上。

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。同时，尽早打开完整的并发编译检查，将成为减轻未来工作量的关键。

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

在并发编程中，调试和调优是一个重要的挑战。人类大脑在理解线性叙事时具有很强的能力，但一旦事情交织在一起形成网络，我们就很难准确地对其进行判断。并发编程中的各种事件相

互关联，同时发生，它们构成了一个复杂的网络。如果出现问题，开发者将面临非常棘手的处理任务。

结构化并发在一定程度上简化了任务的关系：子任务的生命周期严格限制在父任务内部，不会出现逃逸情况，这使得我们能够在父任务的生命周期内评估子任务的行为。每个任务都有确定的开始和结束，只需避免随意创建全新的任务，而是尽可能依赖子任务来组织工作，我们就能根据任务之间的关系进行追踪并确定合理的分析方法。这为并发编程提供了一个良好的结构基础。

目前，我们已经有一些用于调试和监测 GCD 性能的工具。例如，Instruments 中的 Time Profiler 可以帮助我们检查各个线程的工作情况，而 MetricKit 可以从真实用户设备上收集用户的性能信息。在 WWDC 22 上，Apple 在 Instruments 中集成了监控和优化异步编程的工具。相信在未来，Xcode 开发团队将继续为开发者们提供更优秀的并发开发体验。

总结

对于 Swift 并发的探索，我们在这里告一段落。虽然并发编程已经有很长的历史，但 Swift 并发本身还很年轻，仍在快速发展中。在本书中，我们对一些并发相关的主题进行了详细说明，但不可避免地，还有许多方面未能涉及。例如，分布式的 actor、自定义执行器的效果和具体方法，以及并发旧版本系统的兼容性等等。其中一些方面更多地针对服务器开发，而另一些则是因为 Swift 团队还没有准备好公开它们。在接下来的几个 Swift 版本中，我们肯定会看到更多与 Swift 并发相关的特性，它们将为开发者带来更优雅的处理方式和更高效的运行能力。