

Benny Huo

学海无涯，其乐无穷



闲话 Swift 协程 (4) : TaskGroup 与结构化并发

📅 2022-01-22 | 📅 2022-12-31 | 👁 1082

📖 9.6k | ⌚ 18 分钟

上一篇文章我们提到了结构化并发，这听上去很高级。

- [闲话 Swift 协程 \(0\) : 前言](#)
- [闲话 Swift 协程 \(1\) : Swift 协程长什么样?](#)
- [闲话 Swift 协程 \(2\) : 将回调改写成 async 函数](#)
- [闲话 Swift 协程 \(3\) : 在程序当中调用异步函数](#)
- [闲话 Swift 协程 \(4\) : TaskGroup 与结构化并发](#)
- [闲话 Swift 协程 \(5\) : Task 的取消](#)
- [闲话 Swift 协程 \(6\) : Actor 和属性隔离](#)
- [闲话 Swift 协程 \(7\) : GlobalActor 和异步函数的调度](#)
- [闲话 Swift 协程 \(8\) : TaskLocal](#)
- [闲话 Swift 协程 \(9\) : 异步函数与其他语言的互调用](#)

TaskGroup 的基本用法

我们现在已经知道怎么在自己的程序里面调用异步函数了。

不难发现，调用异步函数的关键点是创建 Task 的实例。通过 Task 的构造器或者 detach 函数创建的 Task 实例都是顶级的，这意味着这些实例都需要单独管理。在真实的业务场景中，我们难免会创建很多 Task 实例来执行不同的异步任务，但这些任务之间往往都是存在关联的，因此我们绝大多数情况下更希望这些 Task 实例是作为一个或者几个整体来统一管理的。

这就需要 TaskGroup 了。



创建 TaskGroup 的方式非常简单，使用 `withTaskGroup(of:returning:body:)` 函数即可，它的完整定义如下：

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

它有三个参数，但实际上前两个其实就是泛型参数，其中

- ChildTaskResult 表示这个 TaskGroup 内创建的 Task 的结果类型
- GroupResult TaskGroup 自身的结果类型

后者其实也是第三个参数 body 的返回值类型。

注意到 withTaskGroup 是异步函数，它会在 TaskGroup 当中所有的子 Task 执行完之后再返回。我们可以在 body 当中向 TaskGroup 当中添加子 Task，用到 addTask 函数：

```
1 public mutating func addTask(  
2     priority: _Concurrency.TaskPriority? = nil,  
3     operation: @escaping @Sendable () async -> ChildTaskResult  
4 )
```

其中：

- priority 是当前任务的优先级
- operation 就是任务的执行体

尽管 withTaskGroup 会等待子 Task 执行完，但有些情况下我们希望在 body 当中就提前等待子 Task 的执行结果，这时候我们有两种做法：

- 如果只关心子 Task 是否执行完，可以调用 TaskGroup 的 `waitForAll` 函数。不难想到，这也是一个异步函数。
- 更常见的情况是获取子 Task 的结果，这时候我们可以直接迭代 TaskGroup，或者调用 TaskGroup 的 `next` 函数来获取下一个已完成的子 Task 的结果。注意，获取的结果的顺序取决于子 Task 完成的顺序，而不是它们添加到 TaskGroup 当中的顺序。



一个结构化并发的简单示例

下面我们给大家看一个非常简单的异步分段计算的例子：

```
1  // 定义一个计算 [min, max) 范围内整数的和的闭包，注意前闭后开
2  let add = { (min: Int, max: Int) -> Int in
3      var sum = 0
4      for i in min..
```

通过 `withTaskGroup` 创建了一个 `TaskGroup` 实例，子 `Task` 的结果类型和 `TaskGroup` 的类型都是 `Int`，我们将 `[0, n]` 的整数按照 `seg` 进行分段，每段整数的和通过一个子 `Task` 来完成计

算。

由于子 Task 的实例我们是无法直接拿到的，因此我们需要通过 TaskGroup 的实例来获取子任务的结果。通过上面的例子我们不难发现 group 是可以被迭代的，很自然的能想到 TaskGroup 有以下函数：

```
1 public mutating func next() async -> ChildTaskResult?
```

并且实现了 AsyncSequence 协议：

```
1 extension TaskGroup : _Concurrency.AsyncSequence { ... }
```

AsyncSequence 与 Sequence 的不同之处在于它的迭代器的 next 函数是异步函数，这就与前面 TaskGroup 的 next 函数对应上了。

计算 totalSum 除了使用经典的 for 循环以外，我们也可以使用 reduce：

```
1 let totalSum = await group.reduce(0) { acc, i in  
2     acc + i  
3 }
```

其中 reduce 的第一个参数是初始值，第二个参数是个闭包，它的参数 acc 是累积的结果，i 是当前的元素，返回值则会作为下一个元素调用时的 acc 传入，最终得到的就是所有子 Task 的结果的和。

会抛异常的 TaskGroup

大家可能发现了，我们前面创建的 TaskGroup 里面的子 Task 不能抛异常。因此我们很自然的想到还有一套可以抛异常的 TaskGroup 的函数：

```
1 public func withThrowingTaskGroup<ChildTaskResult, GroupResult>(  
2     of childTaskResultType: ChildTaskResult.Type,  
3     returning returnType: GroupResult.Type = GroupResult.self,  
4     body: (inout _Concurrency.ThrowingTaskGroup<ChildTaskResult, Error>) asy  
5 ) async rethrows -> GroupResult
```

通过它创建的 TaskGroup 的类型是：

```
1 @frozen public struct ThrowingTaskGroup<ChildTaskResult, Failure> where Fail
```

ThrowingTaskGroup 与 TaskGroup 的本质是一致的，只不过 ThrowingTaskGroup 的所有成员函数都增加了 throws 关键字。

```
1 do {
2     _ = try await withThrowingTaskGroup(of: Int.self) { group -> String in
3         try await Task.sleep(nanoseconds: 1000000)
4         return "OK"
5     }
6 } catch {
7     ...
8 }
```

注意到 withThrowingTaskGroup 是 rethrows 的，如果闭包参数里面有异常抛出，调用时也需要做异常处理。例子当中调用到了 Task 的 sleep 函数，需要大家注意的是 Task 有两个 sleep 函数，带 nanoseconds 的这个版本是会抛异常的：

```
1 // 参数没有 label，没有标记为 throws，调用时不需要处理异常
2 public static func sleep(_ duration: UInt64) async
3
4 // 参数有 label，标记为 throws
5 public static func sleep(nanoseconds duration: UInt64) async throws
```

因此这里需要使用 withThrowingTaskGroup 来做异常的传递。

除抛异常这个点以外，ThrowingTaskGroup 的用法与 TaskGroup 完全一致。

子 Task 的异常处理

在 TaskGroup 当中，子 Task 如果抛出了异常，当外部调用者试图通过 TaskGroup 实例获取它的结果时也会抛出这个异常。需要注意的是，由于子 Task 结果的获取顺序取决于实际 Task 的完成时间，因此获取结果时需要注意对单个 Task 的结果进行异常捕获，以免影响其他 Task 的结果：

```
1 let result = await withThrowingTaskGroup(of: Int.self) { group -> Int in
2     group.addTask {
3         await Task.sleep(500_000_000)
```



```
4         return -1
5     }
6
7     group.addTask {
8         await Task.sleep(1000_000_000)
9         try await errorThrown()
10        return 0
11    }
12
13    group.addTask {
14        await Task.sleep(1500_000_000)
15        return 1
16    }
17
18    while(!group.isEmpty) {
19        do {
20            print(try await group.next() ?? "Nil")
21        } catch {
22            print(error)
23        }
24    }
25
26    return 100
27 }
```

这个例子当中，返回 0 的子 Task 抛了异常，我们在试图遍历 group 时就会遇到这个异常：

```
1  -1
2  Runtime Error
3  1
```

而其他的子 Task 的结果是可以正常获取的。可见 TaskGroup 当中的 Task 抛异常并不会影响其他 Task 的运行。

不要把 TaskGroup 的实例泄漏到外部

从前面的例子我们大致可以看出，Swift 的 TaskGroup 的 API 设计还是非常谨慎的，TaskGroup 的实例只有在 withTaskGroup 的闭包参数当中使用，外部没有办法直接获取。

那有没有办法能让 TaskGroup 的实例逃逸出这个闭包呢？我们来做一点儿小尝试：



```

1  var taskGroup: TaskGroup<Int>?
2  _ = await withTaskGroup(of: Int.self) { (group) -> Int in
3      taskGroup = group
4      group.addTask { 1 }
5      return 0
6  }
7
8  guard let group = taskGroup else {
9      print("group is nil")
10     return
11 }
12
13 for await i in group {
14     print(i)
15 }

```

我们在闭包外面定义一个变量 `taskGroup`，在闭包里面给 `taskGroup` 赋值。接下来我们在外面尝试访问以下 `taskGroup` 的子任务结果，运行之后就会发现：

```

1  Process finished with exit code 133 (interrupted by signal 5: SIGTRAP)

```

错误发生的位置就是这里： `for await i in group { ... }`。

为什么会出现异常呢？我们前面提到过， `withTaskGroup` 会在所有的子 Task 执行完以后再返回，这是否意味着 `TaskGroup` 的实例也会在此时被销毁呢？

遇到这种问题，我们只需要翻阅一下 swift 的源码：

```

1  public func withTaskGroup<ChildTaskResult, GroupResult>(
2      of childTaskResultType: ChildTaskResult.Type,
3      returning returnType: GroupResult.Type = GroupResult.self,
4      body: (inout TaskGroup<ChildTaskResult>) async -> GroupResult
5  ) async -> GroupResult {
6      let _group = Builtin.createTaskGroup(ChildTaskResult.self)
7      var group = TaskGroup<ChildTaskResult>(group: _group)
8
9      // Run the withTaskGroup body.
10     let result = await body(&group)
11
12     await group.awaitAllRemainingTasks()
13
14     Builtin.destroyTaskGroup(_group)

```



```
15     return result
16 }
```

可见，withTaskGroup 返回前会先等待所有的子 Task 执行完毕，然后将 TaskGroup 销毁。因此将 TaskGroup 的实例泄漏到外部没有任何意义。

不要在子 Task 当中修改 TaskGroup

TaskGroup 泄漏到外部是危险的，这其实很容易想到。那么在子 Task 当中呢？

```
1  await withTaskGroup(of: Void.self) { (group) -> Void in
2      group.addTask {
3          group.addTask { // error!
4              print("inner task")
5          }
6      }
7  }
```

如果你尝试在子 Task 当中去修改 group（addTask 是 mutating func），你会得到这样的错误：

```
1  Mutation of captured parameter 'group' in concurrently-executing code
```

正如前面提到不能把 TaskGroup 的实例泄漏到外面一样，它也同样不能泄漏到子 Task 的执行体当中。道理也很简单，子 Task 的执行体可能会被调度到不同的线程上，这样就导致对 TaskGroup 的修改是并发的，不安全。

async let

除了使用 TaskGroup 添加子 Task 的方式来构造结构化并发以外，我们还有一种比较便捷的方式，那就是使用 async let。async let 一方面可以让子 Task 的创建和结果的返回变得更加简单，~~另一方面也可以解决了子 Task 的结果不好定位的问题（因为遍历 TaskGroup 时子 Task 的结果返回顺序不确定）。~~

下面我们给出一个简单的例子来说明这个问题：

```
1  struct User {
2      let name: String
```




```
3     let info: String
4     let followers: [String]
5     let projects: [String]
6 }
```

定义一个数据结构 User，我们现在需要通过访问网络情况来构造这样一个实例，其中：

```
1 func getUserInfo(_ user: String) async -> String {
2     ...
3 }
4
5 func getFollowers(_ user: String) async -> [String] {
6     ...
7 }
8
9 func getProjects(_ user: String) async -> [String] {
10    ...
11 }
```

以上三个函数将发送异步网络请求去获取对应字段的数据。如果使用 TaskGroup，代码写起来将会比较复杂：

```
1 enum Result {
2     case info(value: String)
3     case followers(value: [String])
4     case projects(value: [String])
5 }
6
7 func getUser(name: String) async -> User {
8     await withTaskGroup(of: Result.self) { group in
9         group.addTask {
10             .info(value: await getUserInfo(name))
11         }
12
13         group.addTask {
14             .followers(value: await getFollowers(name))
15         }
16
17         group.addTask {
18             .projects(value: await getProjects(name))
19         }
20
21         var info: String? = nil
22         var followers: [String]? = nil
```



```
23     var projects: [String]? = nil
24     for await r in group {
25         switch r {
26             case .info(value: let value):
27                 info = value
28             case .followers(value: let value):
29                 followers = value
30             case .projects(value: let value):
31                 projects = value
32         }
33     }
34
35     return User(name: name, info: info ?? "", followers: followers ?? [
36 ]
37 }
```

前面多次提到对 TaskGroup 进行遍历获取子 Task 的结果时存在顺序的不确定性，为了解决这个问题我们定义了一个枚举 Result 将子 Task 的结果与枚举值进行绑定，方便后续读取结果。这个过程异常繁琐，且引入额外的类型实现结果的绑定让问题变得更加复杂。

如果使用 async let，这个问题就会变得非常简单：

```
1 func getUser(name: String) async -> User {
2     async let info = getUserInfo(name)
3     async let followers = getFollowers(name)
4     async let projects = getProjects(name)
5
6     return User(name: name, info: await info, followers: await followers, pr
7 }
```

async let 会创建一个子 Task 来完成后面的调用，并且把结果绑定到对应的变量当中。以 info 为例，~~当我们需要读取其结果时，只需要 await info 即可，这样就大大降低了我们获取异步子 Task 的结果的复杂度。~~

另外稍微提一句的是，在 Swift 当中，async 函数的调用必须使用 await 来等待这个限制会强制我们等待异步函数的结果，如果希望同时触发多个异步函数的调用，async let 能解决的问题也是有限的。例如我们想要并发获取多个 User 的数据，需要实现以下函数：

```
1 func getUsers(names:[String]) async -> [User]
```



我们可以基于前面的 `getUser` 来实现这个函数，为了保证 `User` 数据的获取的并发性，我们需要同时创建多个 `Task` 来完成请求：

```
1 func getUsers(names:[String]) async -> [User] {
2     await withTaskGroup(of: User.self) { group in
3         for name in names {
4             group.addTask {
5                 await getUser(name: name)
6             }
7         }
8
9         return await group.reduce(into: Array<User>()) { (partialResult, user) in
10             partialResult.append(user)
11         }
12     }
13 }
```

这种情况下 `async let` 就显得有点儿力不从心了。

更进一步，如果这里要求返回的 `User` 跟传入的 `name` 能够在顺序上一一对应，使用 `TaskGroup` 实现就会比较麻烦，因为 `TaskGroup` 的结果顺序是子 `Task` 完成的顺序。

实际上，保证结果的顺序与 `Task` 的添加顺序一致是有实际需求的，我们也可以使用一组 `Task` 而不是 `TaskGroup` 来实现这个需求：

```
1 func getUsers(names: [String]) async throws -> [User] {
2     let tasks = names.map { name in
3         Task { () -> User in
4             return await getUser(name: name)
5         }
6     }
7
8     return try await withTaskCancellationHandler(operation: {
9         var users = Array<User>()
10        for task in tasks {
11            users.append(try await task.value)
12        }
13        return users
14    }, onCancel: {
15        tasks.forEach { task in task.cancel() }
```



```
16     })  
17 }
```

由于这时候我们创建的 Task 都是不隶属于 TaskGroup 的（即非结构化并发），此时我们要小心处理 Task 取消的情况，以免出现内存泄漏和逻辑错误。

小结

本文我们简单介绍了一下 TaskGroup 的用法，大家可以基于这些内容开始做一些简单的尝试了。结构化并发当中还有一些重要的概念我们将在接下来的几篇文章当中逐步介绍。

关于作者

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

- GitHub: <https://github.com/bennyhuo>
- 博客: <https://www.bennyhuo.com>
- bilibili: [bennyhuo不是算命的](#)
- 微信公众号: bennyhuo

相关推荐

- [闲话 Swift 协程 \(0\) : 前言](#)
- [闲话 Swift 协程 \(1\) : Swift 协程长什么样?](#)
- [闲话 Swift 协程 \(2\) : 将回调改写成 async 函数](#)
- [闲话 Swift 协程 \(3\) : 在程序当中调用异步函数](#)
- [闲话 Swift 协程 \(5\) : Task 的取消](#)

[# coroutines](#) [# swift](#) [# async await](#)




[◀ 闲话 Swift 协程 \(3\) : 在程序当中调用异步函数](#)

闲话 Swift 协程 (5) : Task 的取消 [▶](#)

[0 条评论](#)

未登录用户 [▼](#)



说点什么

[支持 Markdown 语法](#)

使用 GitHub 登录

预览

来做第一个留言的人吧!

[京ICP备16022265号-3](#)

© 2018 — 2022 [👤 Benny Huo](#) | [📄 478k](#) | [☕ 14:29](#)

由 [Hexo](#) & [NexT.Pisces](#) 强力驱动