闲话 Swift 协程（3）：在程序当中调用异步函数

2022-01-21 | 2022-12-31 | 1492

4.7k | 9 分钟

> ~~异步函数需要被异步函数调用，这听上去就是一个鸡生蛋蛋生鸡的问题。关键的问题在于，~~第一个异步函数从哪儿来~~?~~

我们现在已经知道怎么定义异步函数了，也可以很轻松的转换将现有的异步回调 API 转成异步函数。那下一个问题就是，既然普通函数不能调用异步函数，那定义好的这些异步函数该从哪儿开始调用呢?

使用 Task

Task 的创建

其实从上一节我们分析如何将回调转成异步函数的时候就已经发现，异步函数的关键在于 Continuation。所以，只要调用异步函数的位置能让异步函数获取到 Continuation，那么调用异步函数的问题就解决了。~~Swift 标准库提供了 Task 类来提供这个能力。~~

我们给出 Task 的构造器的定义：

```
public init(
    priority: _Concurrency.TaskPriority? = nil,
    operation: @escaping @Sendable () async -> Success)

public init(
    priority: _Concurrency.TaskPriority? = nil,
    operation: @escaping @Sendable () async throws -> Success)
```

它接收一个异步闭包作为参数，创建一个 Task 实例并运行这个异步闭包。而在这个闭包当中，我们就可以调用任意异步函数了：

```
Task {
    let result = await helloAsync()
    print(result)
}
```

除了直接构造 Task 之外，还可以调用 Task 的 detach 函数来创建一个不一样的 Task：

```
Task.detached (operation: {
    await helloAsync()
})
```

这个函数返回的也是一个 Task 实例，我们不妨看一下它的定义：

```
public static func detached(
    priority: _Concurrency.TaskPriority? = nil,
    operation: @escaping @Sendable () async -> Success
) -> _Concurrency.Task<Success, Failure>

public static func detached(
    priority: _Concurrency.TaskPriority? = nil,
    operation: @escaping @Sendable () async throws -> Success
) -> _Concurrency.Task<Success, Failure>
```

注意到它其实是 Task 的静态函数，返回值正是 Task 类型。

两种 Task 的对比

那通过 detached 函数创建的 Task 和直接使用 Task 的构造器创建的 Task 实例有什么不同呢？我们先来看一下文档的说明：

detached 函数的部分注释

```
1  /// Runs the given nonthrowing operation asynchronously
2  /// as part of a new top-level task.
```

Task 类的 **init** 的部分注释

```
1  /// Runs the given nonthrowing operation asynchronously
2  /// as part of a new top-level task on behalf of the current actor.
```

可以看到这两段说明有一个共同点：通过二者创建的 Task 都是 top-level task。这是什么意思呢？这个其实是与在 TaskGroup 当中创建子任务是相对应的，前面介绍的这两种方式创建出来的任务都是顶级任务，没有父任务。TaskGroup 的内容我们下一篇文章再介绍。

接下来就是区别点了，即使用 Task 直接构造的任务实例会 `on behalf of the current actor` 。Actor 我们还没有介绍，不过我们姑且理解为任务启动时所在的运行环境。这里主要包括挂起的异步函数在恢复时如何调度，以及对于 TaskLocal 变量的感知上。这些内容我们后面会专门写文章介绍。

简单来说，通过 `Task { ... }` 创建的任务会对外界的状态有感知，而通过 `Task.detached { ... }` 创建的任务就完全是个孤儿了 —— 也正是因为这一点，官方文档里面也提醒我们一般情况下不要使用 detached 来创建任务。

以上创建 Task 的方式，也被称为**非结构化并发**。

这里并发的意思是，Task 都会把自己的代码块传给一个后台异步队列去执行。非结构化则与添加到 TaskGroup 当中的任务相对应，添加到 TaskGroup 当中的任务的形式被称为结构化并发，这些 Task 会随着整个 TaskGroup 的取消而取消，而相对应地，顶级任务的状态管理都只与自己有关，想要取消也必须调用 Task 的 cancel 显式地对任务进行取消。

现在你应该对 TaskGroup、Actor、TaskLocal 之类的概念也产生了兴趣，如果不能理解，也先不着急，我们等后面再慢慢展开介绍。

不管怎样，讲到这里，我们已经知道如何在程序当中使用异步函数了，下面我们给出一个完整的命令行程序：

```
func helloAsync() async -> Int {
    await withCheckedContinuation { continuation in
        DispatchQueue.global().async {
            continuation.resume(returning: Int(arc4random()))
        }
    }
}

Task.detached {
    print(await helloAsync())
}

Task {
    print(await helloAsync())
}

// 主线程等待 1s，防止程序提前退出导致异步任务没有执行
Thread.sleep(forTimeInterval: 1)
```

运行这个程序可以得到：

```
1804289383
846930886
```

嗯，这是两个随机数。在这个例子当中，我们既没有定义 Actor，也没有定义 TaskLocal，因此创建出来的两个 Task 其实是没有什么本质的区别的。

> 说明：Swift 的协程需要 macOS 12.0，iOS 15.0 及以上版本才可以运行，因此大家可以在 iOS 15.0 的设备或者模拟器上体验异步函数的调用。有趣的是，在 Windows 和 Linux 上安装 Swift 5.5 的编译器之后，上述程序是可以运行的。

Task 的结果

Task 的闭包有返回值作为它的结果返回。由于 Task 是异步执行的，它的结果自然也是异步的：

```
// Task
public var value: Success { get async throws }
```

我们可以在其他异步函数当中使用 await 来获取它的结果：

```
let task = Task {
    await helloAsync()
}

print(try await task.value)
```

由于 Task 的闭包可以抛出异常，因此对于每一个 Task 来讲，异常也是结果的一种可能。如果我们只是任性地启动了一个 Task 而不去获取它的结果的话，Task 内部抛出的任何异常都与外部无关：

```
func errorThrown() async throws {
    throw "Runtime Error"
}

func taskWithError() async throws {
    let task = Task {
        try await errorThrown()
    }

    // 避免程序过早退出，等 1s
    await Task.sleep(1000_000_000)
}
```

如果我们想要看看 Task 究竟抛出了什么异常，我们可以在读取它的 value 时对异常进行捕获：

```
func taskWithError() async throws {
    let task = Task {
        try await errorThrown()
    }

    do {
        try await task.value
    } catch {
        print(error)
    }
}
```

我们前面定义的 Task 时传入的闭包会抛异常，这样一来 Task 的第二个泛型参数 Failure 就不可能是 Never。这种情况下获取 value 的操作需要使用 try 关键字。

异步 main 函数

通过创建 Task 的方式适用于所有在同步函数当中需要调用异步函数的情形。当然，对于命令行程序来讲，我们还可以直接把 main 函数定义为 async 函数：

App.swift

```
@main
struct App {
    static func main() async throws {
        ...
    }
}
```

首先我们定义一个结构体（或者类），将其标注为 @main；接着定义一个静态的 main 函数，这个函数可以是同步函数也可以是异步函数。

> 注意，通过这种方式，main.swift 文件要留空（或者直接删掉）。

这样我们就可以愉快地调用异步函数了：

```
import Foundation

@main
struct App {
    static func main() async throws {
        print(await helloAsync())

        let detachedTask = Task.detached { () -> Int in
            print(await helloAsync())
            return 1
        }

        let task = Task { () -> Int in
            print(await helloAsync())
            return 2
        }

```

```
18          print("detached task result: \(try await detachedTask.value)")
19          print("task result: \(try await task.value)")
20      }
21  }
```

> 说明：异步 main 函数同样受到 macOS 运行时版本的限制，但在 Windows 和 Linux 上不受限制。

小结

本文我们主要介绍了如何创建调用异步函数的条件的问题，大家也可以自己体验一下 Swift 的协程了。

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（4）：TaskGroup 与结构化并发
- 闲话 Swift 协程（5）：Task 的取消

coroutines # swift # async await

2021 总结 – bennyhuo

闲话 Swift 协程（4）：TaskGroup 与结构化并发

2 条评论

未登录用户

说点什么

支持 Markdown 语法

使用 GitHub 登录 预览

langyangyangzzZ 发表于 9 个月前

沙发

bennyhuo 发表于 9 个月前

> 沙发

哈

京ICP备16022265号-3

© 2018 — 2022 Benny Huo | 478k | 14:29

由 Hexo & NexT.Pisces 强力驱动