闲话 Swift 协程（1）：Swift 协程长什么样?

2021-10-11 | 2022-12-31 | 3112

4.3k | 8 分钟

> 2021 年 9 月 20 日，Apple 发布了 Swift 5.5，这个版本当中最亮眼的特性就是对 async await 的支持了。

- 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

协程的基本概念

协程（Coroutines）不是一个语言特有的概念，也没有一个特别严格的定义，维基百科对它定义也只是对它最核心的非抢占式多任务调度进行了简单的描述：

> Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.

简单来说就是，协程是一种非抢占式或者说协作式的计算机程序并发调度的实现，程序可以主动挂起或者恢复执行。

说起任务调度，我们很自然地想到线程。从任务载体的角度来讲，协程和线程在应用场景上的确有很大的重叠之处，协程最初也确实是被应用于操作系统的任务调度的。只不过后来抢占式的调度成为了操作系统的主流实现，因此以协程为执行任务单位的协作式的调度就很少出现在我们眼前了。我们现在提到线程，基本上指的就是操作系统的内核线程；而提到协程，绝大多数都是编程语言层面实现的任务载体——我们看待一个线程，就好像一艘轮船一样，而协程似乎就是装在上面的一个集装箱。

从任务的承载上来讲，线程比协程更重；从调度执行的能力来讲，线程是由操作系统调度的，而协程则是由编程语言的运行时调度的。所以绝大多数的编程语言当中实现的协程都具备更加轻量和更加灵活的特点。对于高负载的服务端，协程的轻量型就表现地很突出；而对于复杂的业务逻辑，特别是与外部异步交互的场景，协程的灵活性就可以发挥作用。

对于 Swift 而言，主要应对的自然是简化复杂的异步逻辑。而针对类似的场景，各家实际上已经给出了近乎一致的语法：async/await。其中 async 用于修饰函数，将其声明为一个异步函数，await 则用于非阻塞地等待异步函数的结果——Swift 也不能免俗。

不过，在有大前端应用场景的语言当中（例如 JavaScript、Dart、C# 等等），有一个“邪教徒”，那就是 Kotlin。相比之下它的语法比较奇葩，只用了一个 suspend 关键字就实现了几乎前面所有的能力（甚至还能做到更多）。Swift 协程与 Kotlin 协程从实现原理上还有代码交互上都颇有渊源，这个我们留在后面专门介绍。

async/await

为了快速了解 Swift 协程的语法，我们先给出一段代码，让大家感受一下它的样子。

在这个例子当中，我们使用 Alamofire 这个网络框架发起网络请求：

```
1  static func getImageData(url: String) async throws -> Data{
2    try await AF.request(url).responseDataAsync() // 调用异步函数，挂起等待结果
3  }
```

这个 responseDataAsync 函数是我对 Alamofire 框架当中的 DataRequest 做的一个扩展：

```
extension DataRequest {
    func responseDecodableAsync<T: Decodable>(...) async throws -> T {
        ...
    }
}
```

它的具体实现我们将在后面给出。

我们先请大家观察这两个函数的形式与普通函数有什么不同。我相信你很容易就能看出来，函数声明的返回值处多了个 async，而在调用函数的时候则多了个 await。使用 async 修饰的函数与普通的同步函数不同，它被称作异步函数。异步函数可以调用其他异步函数，而同步函数则不能调用异步函数。

正如我们前面提到的，async/await 这样的形式其实也是现在主流编程语言所支持的方式，例如：

JavaScript

```
async function delay(seconds) {
  ...
}
async function asyncCall() {
        await delay(2); // 调用异步函数，挂起等待结果
        ...
}
```

我们看到在 JavaScript 当中同样可以通过 async 关键字来声明一个支持挂起调用的异步函数，而在想要调用另一个异步函数的时候，则需要使用 await。从形式上来看，Swift 只是把 async 放到了函数声明的后面而已。

我们不妨也看一下 Kotlin 的的协程，Kotlin 当中也有异步函数的概念，只不过它选择了 suspend 这个关键字，因此我们在 Kotlin 当中更多的称这样的函数为挂起函数（其实是可挂起的函数）：

Kotlin

```
suspend fun delay(seconds: Long) {
  ...
}
```

```
4
5  suspend fun asyncCall() {
6    delay(2) // 调用 suspend 函数，异步挂起
7  }
```

从语法的形式上来看，Kotlin 的 suspend 关键字在函数声明时充当了 async 的作用，把函数声明为异步函数；而在调用 suspend 函数的时候则直接相当于强加了 await，如果被调用的 ~~suspend 函数会挂起，那么我们在这个调用点也就只能挂起当前异步函数来等待被调用的异步函~~数的结果返回了。实际上 Swift 的异步函数调用时也会要求使用 await，而 JavaScript 的 await 则在使用和不使用时分别有不同的含义，有关这个设计问题的讨论，我们后面再探讨。

所以讲到这里我希望大家能够了解两个点：

1. 这些编程语言通过 async 关键字将函数分为两类，过去的普通函数为同步函数，被修饰的函数则为异步函数。
2. 调用异步函数的时候需要使用 await 关键字，使得这个异步调用拥有了挂起等待恢复的语义。

async/await 解决了怎样的问题?

在 Swift 5.5 以前，getImageData 的实现通常依赖回调来实现结果的返回：

```
1  static func getImageData(url: String,
2                           onSuccess: @escaping (Data) -> Void,
3                           onError: @escaping (Error) -> Void) {
4      AF.request(url).responseData { response in
5          switch response.result {
6          case .success(let data):
7              onSuccess(data)
8          case .failure(let error):
9              onError(error)
10         }
11     }
12 }
```

很自然地，我们如果想要调用这个函数，代码写出来就像下面这样：

```
1  GitHubApi.getImageData(
2          url: avatar_url,
3          onSuccess: { data in
```

```
            ...
        },
        onError: { error in
            ...
        })
```

那如果我想要在回调当中再触发一些其他的异步操作，结果会怎样呢?

```
GitHubApi.getImageData(
        url: avatar_url,
        onSuccess: { data in
            ...
            cropImage(
                onSuccess: { croppedImage in
                    saveImage(
                        onSuccess: {
                            ...
                        },
                        onError: {
                            ...
                        })
                },
                onError: {
                    ...
                })
        },
        onError: { error in
            ...
        })
```

不难发现，随着逻辑复杂度的增加，代码的缩进会越来越深，可维护性也越来越差。

但这段代码如果用 async/await 改造一下，结果会怎样呢?

```
do {
    let data = await GitHubApiAsync.getImageData(url: userItem.user.avatar_u
    let croppedImage = await cropImage(data)
    await saveImage(croppedImage)
} catch {
    ...
}
```

与 getImageData 函数的同步版本相比，onSuccess 和 onError 这两个回调没有了。尽管结果仍然是异步返回的，但写起来却像是同步返回的一样。这样看来，运用 async/await 可以使回调的层级变少，从而使得代码逻辑变得更清晰。

实际上，对于有一个或两个分支的异步回调，我们都可以很轻松地将其转换为使用 async 修饰的异步函数，进而使用 await 来完成调用。这部分内容我们在后面会专门介绍。

小结

通过前面对协程概念的简单介绍，以及 async/await 与回调的使用对比，我们不难发现协程在简化异步代码的实现方面有着巨大的优势。知道了这一点，我们后续就可以逐步深入去了解 Swift 协程的使用场景和实现细节了。

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（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

coroutines # swift # async await

‹ 闲话 Swift 协程（0）：前言

闲话 Swift 协程（2）：将回调改写成 async 函数 ›

5 条评论

未登录用户

说点什么

支持 Markdown 语法

使用 GitHub 登录 预览

Bei-yuan-qian-jing 发表于 11 个月前

博主您好，Apple 现在对于 Concurrency 的支持目前已经做到一定的向前兼容了：
@available(macOS 10.15, iOS 13.0, watchOS 6.0, tvOS 13.0, *)

bennyhuo 发表于 10 个月前

谢谢提醒！

Mim0sa 发表于 10 个月前

写的太棒了！

shywoody 发表于 9 个月前

> @Bei-yuan-qian-jing
> 博主您好，Apple 现在对于 Concurrency 的支持目前已经做到一定的向前兼容了：
> @available(macOS 10.15, iOS 13.0, watchOS 6.0, tvOS 13.0, *)
> that's very kind of you🙏

bennyhuo 发表于 9 个月前

> 写的太棒了！

🤪 我现在把这个整理成专辑了，阅读体验会更好，快来试试吧：
https://www.bennyhuo.com/book/