

Benny Huo

学海无涯，其乐无穷



## 闲话 Swift 协程 (5) : Task 的取消

📅 2022-01-28 | 📅 2022-12-31 | 👁 754

📖 7k | ⌚ 13 分钟

但凡是个任务，就有可能被取消。取消了该怎么办呢？

- [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](#)

### ~~Task 的取消就是个状态~~

Task 的取消其实非常简单，就是将 Task 标记为取消状态。那 Task 的执行体要怎么做才能让任务真正取消呢？我们先看个简单的例子：

```
1 let task = Task {
2     print("task start")
3     await Task.sleep(10_000_000_000)
4     print("task finish")
5 }
6
7 await Task.sleep(500_000_000)
```

```
8  task.cancel()
9  print(await task.result)
```

我们创建了一个 Task，正常情况下它应该很快被执行到，因此第一行日志可以打印出来，随即进入 10s 的睡眠状态。但我们在 Task 外部等了 500ms 之后把它取消了，如果不出什么意外的话，在 Task 睡眠时它就被取消了。

既然任务被取消了，凭我们主观的判断，第二句日志应该是打印不出来的，但实际的情况却是：

```
1  task start
2  task finish
3  success()
```

这说明 Task 的取消只是一个状态标记，它不会强制 Task 的执行体中断，换句话说 Task 的取消并不像杀进程那样粗暴。

实际上，我们可以在任务的执行体当中读取到 Task 的取消状态，我们把程序稍作修改如下：

```
1  let task = Task {
2      print("task start")
3      await Task.sleep(10_000_000_000)
4      print("task finish, isCancelled: \(Task.isCancelled)")
5  }
6
7  await Task.sleep(500_000_000)
8  task.cancel()
9  print(await task.result)
```

运行结果如下：

```
1  task start
2  task finish, isCancelled: true
3  success()
```

可以看到，Task 确实被取消了，我们也可以读取到这个状态，如果我们需要让我们的 Task 执行体响应它的取消状态，那就需要做出这个状态的判断，并且做出响应，例如：

```
1  Task {
2      if !Task.isCancelled {
3          print("task start")
```

```
4      await Task.sleep(10_000_000_000)
5      if !Task.isCancelled {
6          print("task finish, isCancelled: \(Task.isCancelled)")
7      }
8  }
9 }
```



当然，这个例子还不够理想，毕竟睡眠的 10s 是不能响应取消的。那如果让 sleep 函数内部也能响应取消，问题是不是就解决了？

## 通过抛 CancellationError 来响应取消

Task 的执行过程中，难免会存在多层异步函数的嵌套的情况，如果最深处的某一个函数响应了取消状态，怎样才能让外部的异步函数也能很好的配合好这个响应？这其实就是在回答上一节留下的 sleep 该如何响应取消的问题。如果想要优雅地给出这个答案，只能通过抛异常的方式了，~~因为任何条件分支的判断都无法实现有效的传播，而异常天然就具备这样的特性。~~

所以常见的异常响应方式非常简单，如果你在编写一个需要响应取消状态的异步函数，当你检查到 Task 被取消时，只需要抛一个 CancellationError 即可，大家都遵守这个规则，那么这个 Task 就能被优雅地结束。

实际上 Task 一共有两个 sleep 函数，我们仔细对比一下它们的定义：

```
1 public static func sleep(_ duration: UInt64) async
2 public static func sleep(nanoseconds duration: UInt64) async throws
```

二者的区别有两处：

- 参数的 label
- 是否会抛出异常

第二个函数明确通过参数的 label 告诉我们参数是纳秒，同时它还会抛出异常。什么异常？自然是在 Task 被取消时抛出 CancellationError。这么看来我们只需要稍微调整一下代码就能完美解决问题：

```
1 let task = Task {
2     print("task start")
3     try await Task.sleep(nanoseconds: 10_000_000_000)
```

```
4     print("task finish, isCancelled: \(Task.isCancelled)")
5 }
6
7 await Task.sleep(500_000_000)
8 task.cancel()
9 print(await task.result)
```



运行结果如下：

```
1 task start
2 failure(Swift.CancellationError())
```

符合预期。

实际上，如果大家仔细查阅 Swift 的文档，你就会发现第一个 sleep 函数已经被废弃了，它的问题想必大家也已经非常明白了吧。

## checkCancellation：更方便地检查取消状态

前面的例子我们算是躺赢了，但如果实际的代码是下面这样呢？

```
1 let task = Task {
2     print("task start")
3     for i in 0...10000 {
4         doHardWork(i)
5     }
6     print("task finish, isCancelled: \(Task.isCancelled)")
7 }
```

不难，我们只需要加个判断嘛，这样在每次循环的开始，如果 Task 已经被取消，我们就能够及时地停止这个任务的执行：

```
1 let task = Task {
2     print("task start")
3     for i in 0...10000 {
4         if Task.isCancelled {
5             throw CancellationError()
6         }
7         doHardWork(i)
8     }
9 }
```

```

9      print("task finish, isCancelled: \(Task.isCancelled)")
10 }

```

其实，这里有个更方便的写法：



```

1  let task = Task {
2      print("task start")
3      for i in 0...10000 {
4          try Task.checkCancellation()
5          doHardWork(i)
6      }
7      print("task finish, isCancelled: \(Task.isCancelled)")
8  }

```

这个函数也没啥神秘的，因为它的实现非常直接：

```

1  public static func checkCancellation() throws {
2      if Task<Never, Never>.isCancelled {
3          throw _Concurrency.CancellationError()
4      }
5  }

```

## 注册取消回调

前面提到的响应取消的情况实际上是两种类型：

- 调用其他支持响应取消的异步函数，在取消时它会抛出 `CancellationError`
- 自己的代码当中主动检查取消状态，并抛出 `CancellationError`（或者直接退出执行逻辑）

但如果异步的逻辑封装在第三方代码当中，我们只能想办法在 `Task` 取消时调用第三方的取消逻辑来完成响应，这时候情况就复杂一些了。我们就以 `GCD` 的异步 API 为例，首先我们对 `DispatchWorkItem` 做个包装：

```

1  class ContinuationWorkItem<T, E> where E: Error {
2
3      var continuation: CheckedContinuation<T, E>?
4      let block: (ContinuationWorkItem) -> T
5
6      lazy var dispatchItem: DispatchWorkItem = DispatchWorkItem {
7          self.continuation?.resume(returning: self.block(self))
8      }

```

```

9
10     var isCancelled: Bool {
11         get {
12             self.dispatchItem.isCancelled
13         }
14     }
15
16     init(block: @escaping (ContinuationWorkItem<T, E>) -> T) {
17         self.block = block
18     }
19
20     func installContinuation(continuation: CheckedContinuation<T, E>) {
21         self.continuation = continuation
22     }
23
24     func cancel() {
25         dispatchItem.cancel()
26     }
27
28 }

```



这个包装的目的在于支持 `installContinuation`，通过获取 Task 的 `continuation` 来实现异步结果的返回。

这里还有一个细节，`block` 的类型与 `DispatchWorkItem` 的 `block` 多了个参数：

```
1 let block: (ContinuationWorkItem) -> T
```

这主要是为了方面我们在 `block` 当中可以读取到 GCD 的任务是否被取消了。

接下来我们试着用 Task 来封装 GCD 的异步任务，并且实现对取消的响应：

```

1 let task = Task { () -> Int in
2     let asyncRequest = ContinuationWorkItem<Int, Never> { item in
3         print("async start")
4         var i = 0
5         while i < 10 && !item.isCancelled {
6             // 单位 秒
7             Thread.sleep(forTimeInterval: 0.1)
8             i += 1
9             print("i = \(i)")
10        }
11        if item.isCancelled {

```

```
12         print("async cancelled, \(i)")
13         return 0
14     } else {
15         print("async finish")
16         return 1
17     }
18 }
19
20 return await withTaskCancellationHandler {
21     await withCheckedContinuation { (continuation: CheckedContinuation<
22         asyncRequest.installContinuation(continuation: continuation)
23         DispatchQueue.global().async(execute: asyncRequest.dispatchItem
24     )
25 } onCancel: {
26     asyncRequest.cancel()
27 }
28 }
29
30 await Task.sleep(500_000_000)
31 task.cancel()
32 print(await task.result)
```

asyncRequest 其实就是我们创建的对 ContinuationWorkItem 实例，它对 DispatchWorkItem 做了包装，在后面的代码当中传给了 DispatchQueue 去异步执行。为了能够及时感知到 Task 的取消状态变化，我们用到了 withTaskCancellationHandler 这个函数，它的定义如下：

```
1 public func withTaskCancellationHandler<T>(  
2     operation: () async throws -> T,  
3     onCancel handler: @Sendable () -> Void  
4 ) async rethrows -> T
```


显然，这个函数也是个异步函数，它有两个参数，分别是：

- operation，即我们要在当前 Task 当中执行的代码逻辑
- onCancel，在 operation 执行时，如果 Task 被取消，该回调立即执行

有了这个函数，我们就可以在调用第三方异步操作时，及时感知到 Task 的取消状态，并通知第三方取消异步操作。

## TaskGroup 的取消

TaskGroup 也可以被取消, 很容易理解, 所有从属于 TaskGroup 的 Task 在前者被取消以后也会被取消。下面我们给出一个非常简单的例子来说明这个问题:



```
1  let max = 10
2  let taskCount = 10
3
4  await withTaskGroup(of: (Int, Int).self) { group -> Void in
5      for i in 0..
```

我们在 TaskGroup 当中启动了 10 个 Task, 这些 Task 每隔约 1 ~ 1.5 秒就会令 count 加 1, 最终把 Task 的序号和 count 的值返回。TaskGroup 则在启动了所有的 Task 之后 5.5 秒的时候取消, 因此前面的 Task 大多只能将 count 增加到 5 左右。运行结果如下:

```
1  Task: 4, count: 1
2  Task: 6, count: 1
3  ...
4  Task: 2, count: 4
5  Task: 8, count: 4
6  result: (8, 4)
7  Task: 9, count: 4
8  result: (9, 4)
9  Task: 5, count: 4
10 Task: 7, count: 4
```



```
11  result: (5, 4)
12  result: (7, 4)
13  Task: 4, count: 5
14  result: (4, 5)
15  Task: 6, count: 5
16  Task: 3, count: 5
17  result: (6, 5)
18  result: (3, 5)
19  Task: 2, count: 5
20  Task: 0, count: 5
21  result: (2, 5)
22  result: (0, 5)
23  Task: 1, count: 5
24  result: (1, 5)
```



我们省略了部分相似的输出，大家只需要关注包含 result 的行，其中 Task 9 返回的 count 为 4，Task 1 返回的 count 为 5。这说明 TaskGroup 在取消时其中的 Task 确实都被取消了。

## 小结

本文我们重点讨论了 Task 的取消的设计，包括取消状态的概念，如何在不同情况下响应取消状态；最后也通过一个简单地例子了解了一下 TaskGroup 的取消。

大家只需要牢记一点，Task 的取消只是一个状态，需要内部执行逻辑的响应。

## 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 \(4\) : TaskGroup 与结构化并发](#)



[# coroutines](#)

[# swift](#)

[# async await](#)

[◀ 闲话 Swift 协程 \(4\) : TaskGroup 与结构化并发](#)

[闲话 Swift 协程 \(6\) : Actor 和属性隔离 ▶](#)

[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