Swift 结构化并发

由 王巍 (onevcat) 发布于 2021年9月29日 • 最后更新: 2021年12月9日

> 本文是我的新书《Swift 异步和并发》中的部分内容，介绍了关于 Swift 中结构化并发的相关概念。如果你对学习 Swift 并发的其他话题有兴趣，也许这本书可以作为参考读物。

`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) 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。

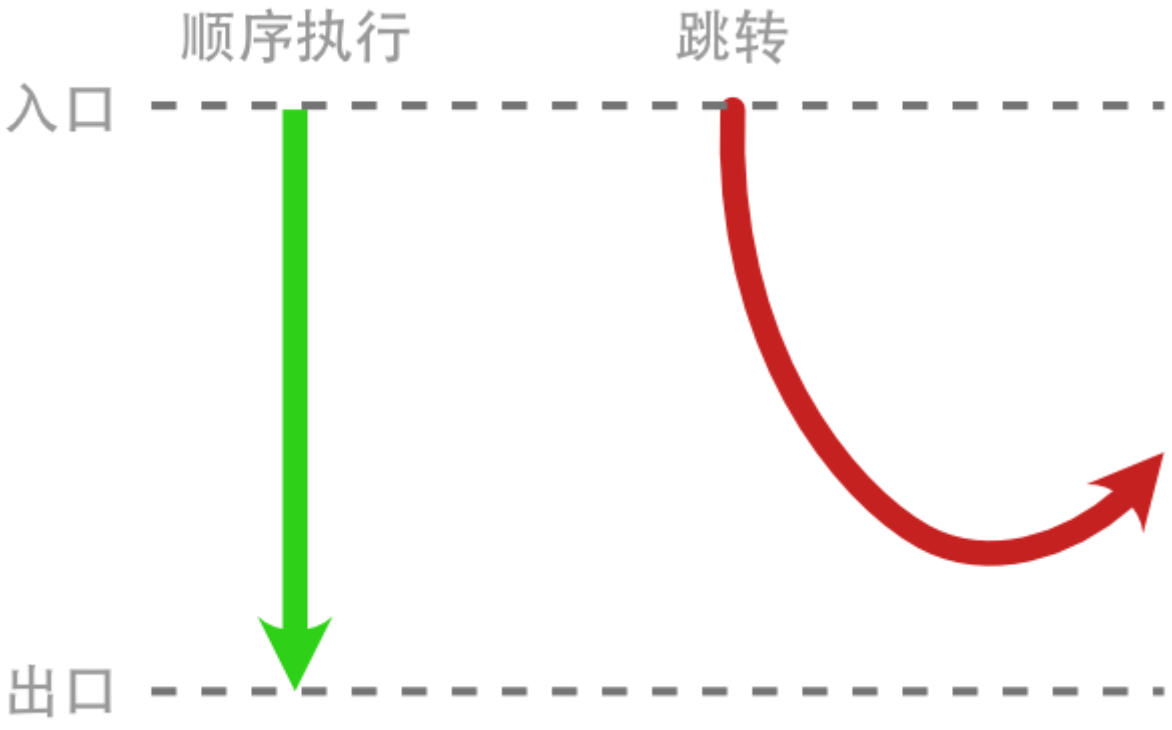
什么是结构化

“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状。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` 语句式微的同样的历史时期，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

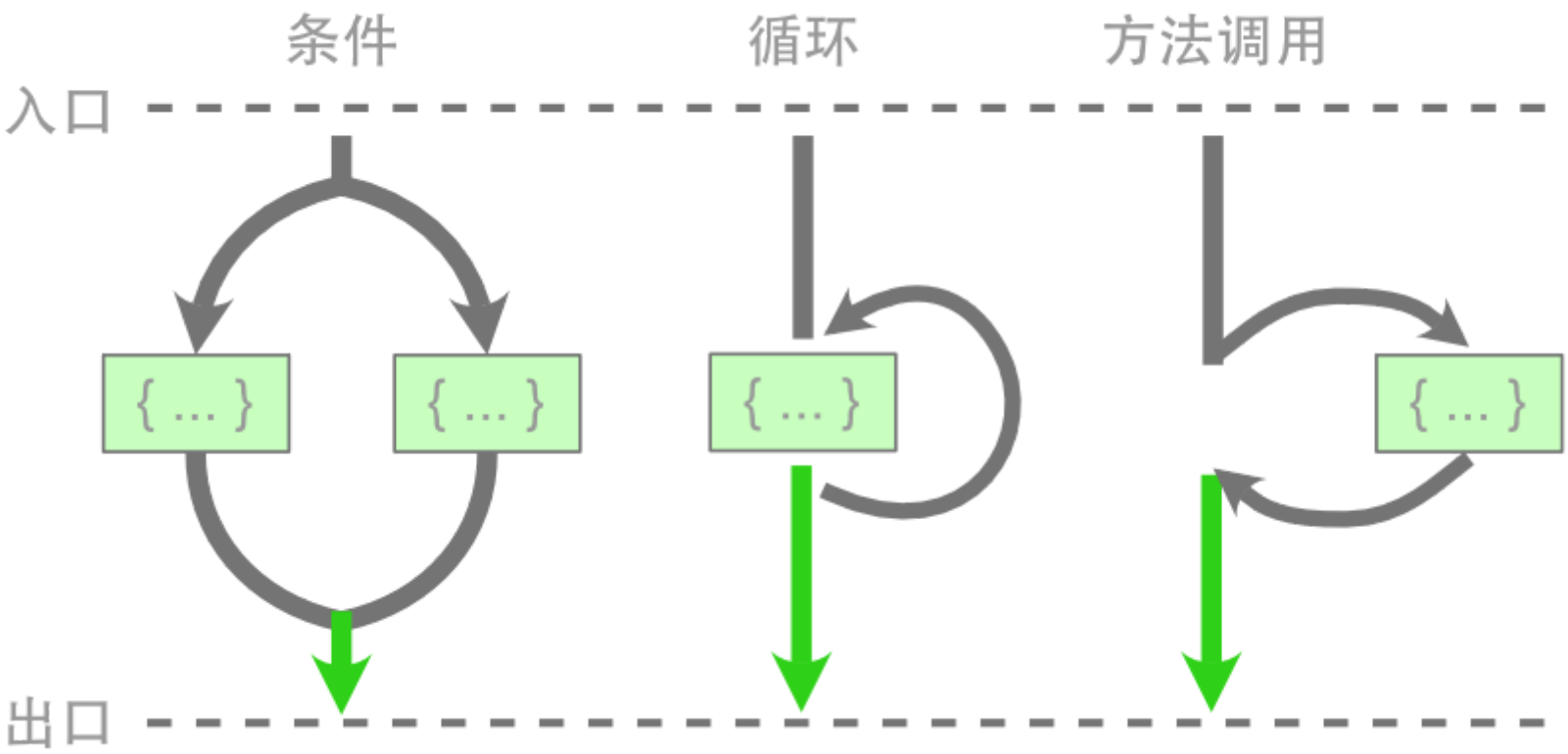
goto 语句是非结构化的，它允许控制流无条件地跳转到某个标签。虽然现在看来 goto 语句已经彻底失败，完全不得人心，但是受限于编程语言的发展，goto 语句在当时是有其生存土壤的。在还没有发明代码块的概念 (也就是 `{ ... }`) 之前，基于顺序执行和跳转的控制流，不仅是最简单的天然选择，也完美契合 CPU 执行指令的方式。顺序执行的语句非常简单，它总可以找到明确的执行入口和出口，但是跳转语句就不一定了：



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，即是我们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只是 `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达到出口。

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。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。这一点对于回收资源 (比如在 `defer` 中关闭文件、切断网络，甚至是自动释放内存等) 是至关重要的。

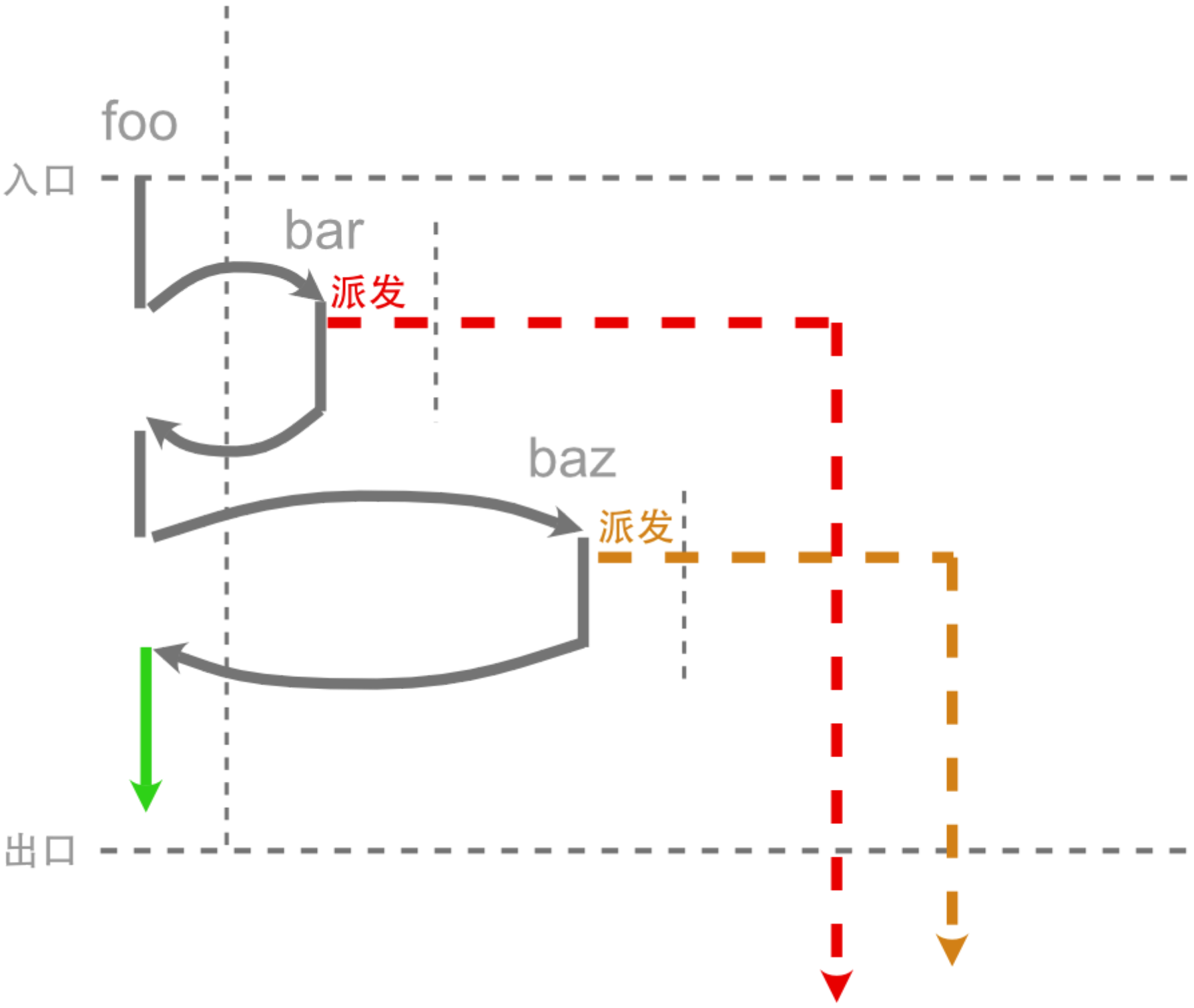
完全禁止使用 `goto` 语句已经成为了大部分现代编程语言的选择。即使有少部分语言还支持 `goto`，它们也大都遵循高德纳 (Donald Ervin Knuth) 所提出的前进分支和后退分支不得交叉的理论。像是 `break`，`continue` 和提前 `return` 这样的控制流，依然遵循着结构化的基本原则：代码拥有单一的入口和出口。事实上我们今天用现代编程语言所写的程序，绝大部分都是结构化的了。当今，结构化编程的习惯已经深入人心，对程序员们来说，使用结构化编程来组织代码，早已如同呼吸一般自然。

非结构化的并发

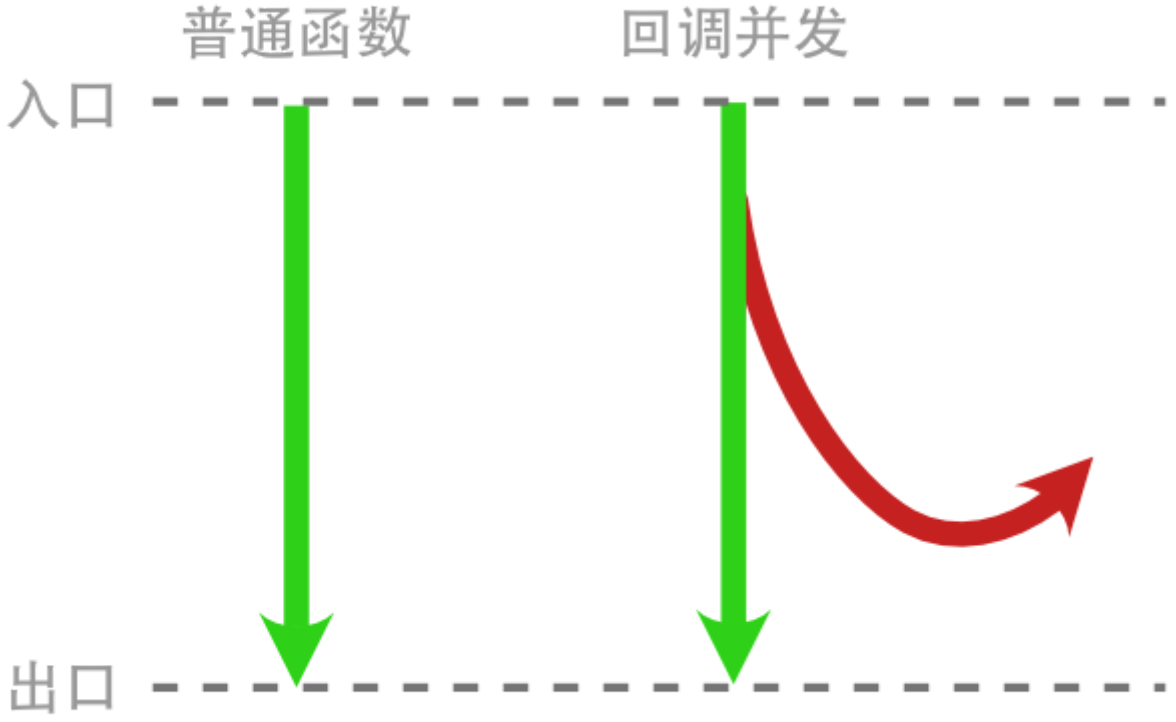
不过，程序的结构化并不意味着并发也是结构化的。相反，Swift 现存的并发模型面临的问题，恰恰和当年 `goto` 的情况类似。Swift 当前的并发手段，最常见的要属使用 `Dispatch` 库将任务派发，并通过回调函数获取结果：

```swift
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。慵懒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让它丑陋的面貌得到了一定程度遮掩。

> 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趁手如意？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，我们都会存在这样的担忧：

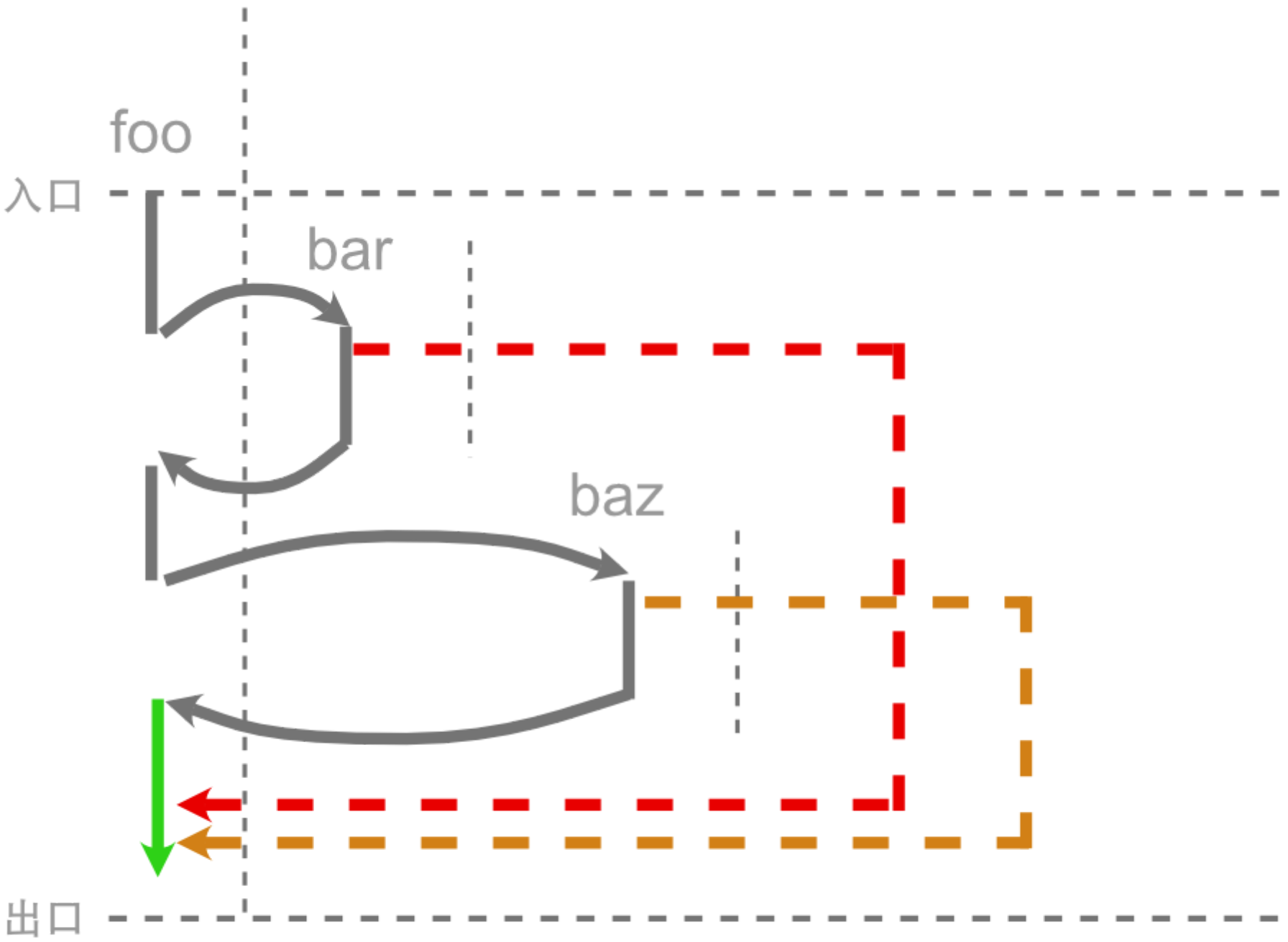
- 这个函数会不会产生一个后台任务？
- 这个函数虽然返回了，但是它所产生的后台任务可能还在运行，它什么时候会结束，它结束后会产生怎么样的行为？
- 作为调用者，我应该在哪里、以怎样的方式处理回调？
- 我需要保持这个函数用到的资源吗？后台任务会自动去持有这些资源吗？我需要自己去释放它们吗？
- 后台任务是否可以被管理，比如想要取消的话应该怎么做？
- 派发出去的任务会不会再去派发别的任务？别的这些任务会被正确管理吗？如果取消了这个派发出去的任务，那些被二次派发的任务也会被正确取消吗？

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！) 然后不断自行猜测。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黑盒状态，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。这也是我们在之前就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，我们总可是使用 `withUnsafeCurrentTask` 来获取和检查当前任务：

```swift
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。
3. 对于 `foo` 的调用，发生在上一步的 `Task` 闭包作用范围中，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。我们甚至可以调用 `cancel()` 来取消这个任务。

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：同步函数也可以在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：

```swift
func foo() async {
  withUnsafeCurrentTask { task in
    // ...
  }
  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。你不能存储这个值，也不能在闭包之外调用或访问它的属性和方法，那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 为此准备了一个“简便方法”：使用 `Task` 的静态属性来获取当前状态，比如：

```swift
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：

```swift
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` 中再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，就需要让内层的 `t1` 和 `t2` 与外层 `Task` 具有某种从属关系。你可以已经猜到了，外层任务作为根节点，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叶子节点，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：

```swift
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)")
    await Task.sleep(UInt64(value) * NSEC_PER_SEC)
    print("Work \(value) done")
    return value
  }
}
```

解释一下上面注释中的数字标注。使用 `withTaskGroup` 可以开启一个新的任务组，它的完整的函数签名是：

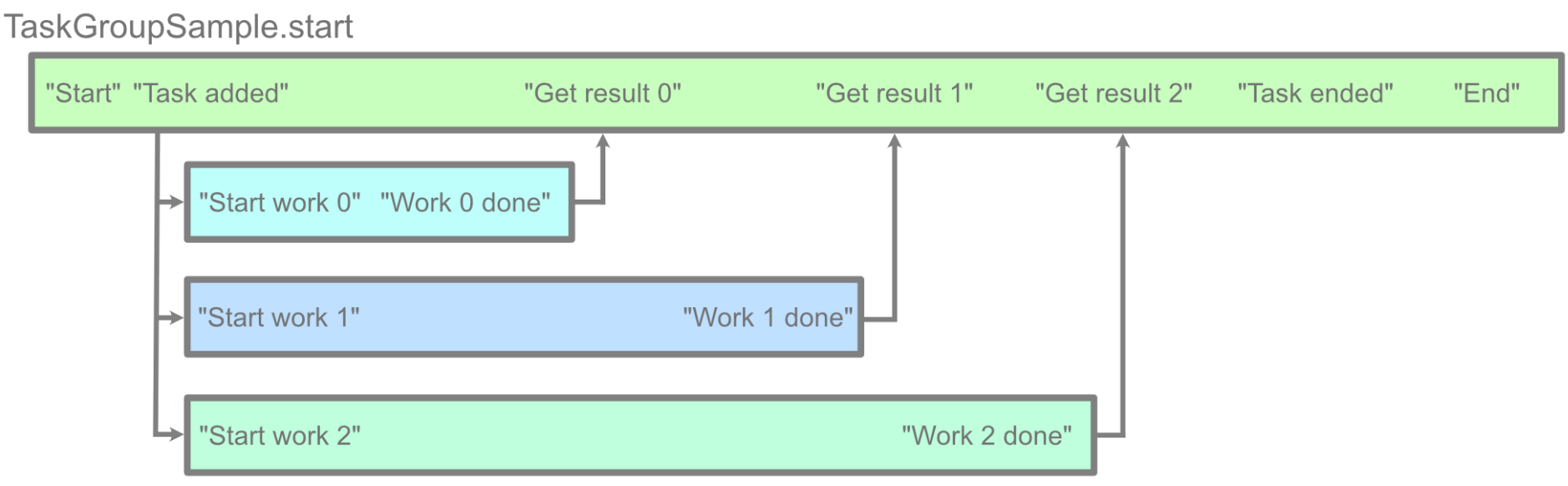
```swift
func withTaskGroup<ChildTaskResult, GroupResult>(
  of childTaskResultType: ChildTaskResult.Type,
  returning returnType: GroupResult.Type = GroupResult.self,
  body: (inout TaskGroup<ChildTaskResult>) async -> GroupResult
) async -> GroupResult
```

1. 这个签名看起来十分复杂，有点吓人，我们来解释一下。 `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。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` 会被调用时，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：

```swift
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，它一定会在子任务全部完成之后才发生。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：

```swift
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：

```swift
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：

```swift
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：

```swift
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，即使只是读取这个 `var value` 值，也是不被允许的：

```
await withTaskGroup(of: Int.self) { group in
  var value = 0
  for i in 0 ..< 3 {
    group.addTask {
      print("Value: \(value)")
      return await work(i)
    }
  }
}
```

将给出错误：

```
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。如果我们能保证 `value` 的值不会被更改的话，可以把 `var value` 的声明改为 `let value` 来避免这个错误：

```
await withTaskGroup(of: Int.self) { group in
  // var value = 0
  let value = 0

  // ...
}
```

或者使用 `[value]` 的语法，来捕获当前的 `value` 值。由于 `value` 是值类型的值，因此它将会遵循值语义，被复制到 `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
    }
  }
  // 将 value 改为 100
  value = 100

  // ...
}
```

不过，如果我们把 `value` 再向上提到类的成员一级的话，这个静态检查将失去作用：

```swift
// 错误的代码，不要这样做
class TaskGroupSample {
  var value = 0
  func start() async {
    await withTaskGroup(of: Int.self) { group in
      for i in 0 ..< 3 {
        group.addTask {

          // 可以访问 value
          print("Value: \(self.value)")

          // 可以操作 value
          let result = await self.work(i)
          self.value += result

          return result
        }
      }
    }

  // ...
  }
}
```

在 Swift 5.5 中，虽然它可以编译 (而且使用起来，特别是在本地调试时也几乎不会有问题)，但这样的行为是**错误**的。和 Rust 不同，Swift 的堆内存所有权模型还无法完全区分内存的借用 (borrow) 和移动 (move)，因此这种数据竞争和内存错误，还需要开发者自行注意。

Swift 编译器并非无法检出上述错误，它只是暂时“容忍”了这种情况。包括静态检测上述错误在内的完全的编译器级别并发数据安全，是未来 Swift 版本中的目标。现在，在并发上下文中访问共享数据时，Swift 设计了 actor 类型来确保数据安全。我们在介绍后面关于 actor 的章节，以及并发底层模型和内存安全的部分后，你会对这种情况背后的原因有更深入的了解。

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：

```swift
// 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` 添加的任务有可能在 `start` 完成后继续运行，这回到了非结构并发的老路；但它也可能让整个任务组进入到难以预测的状态，这将摧毁程序的执行假设。 `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，这个类型仅仅只是作为对底层接口的包装，提供了创建任务节点的方法。要注意，在闭包作用范围外添加任务的行为是未定义的，随着 Swift 的升级，今后有可能直接产生运行时的崩溃。虽然现在并没有提供任何语言特性来确保 `group` 不被复制出去，但是我们绝对应该避免这种反模式的做法。

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”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返回，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，往往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的返回值必须是同样类型的要求，也让灵活性下降或者要求更多的额外实现 (比如将各个任务的返回值用新类型封装等)。 `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，最终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。

在 `withTaskGroup` 的例子中的代码，使用 `async let` 可以改写为下面的形式：

```
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
```

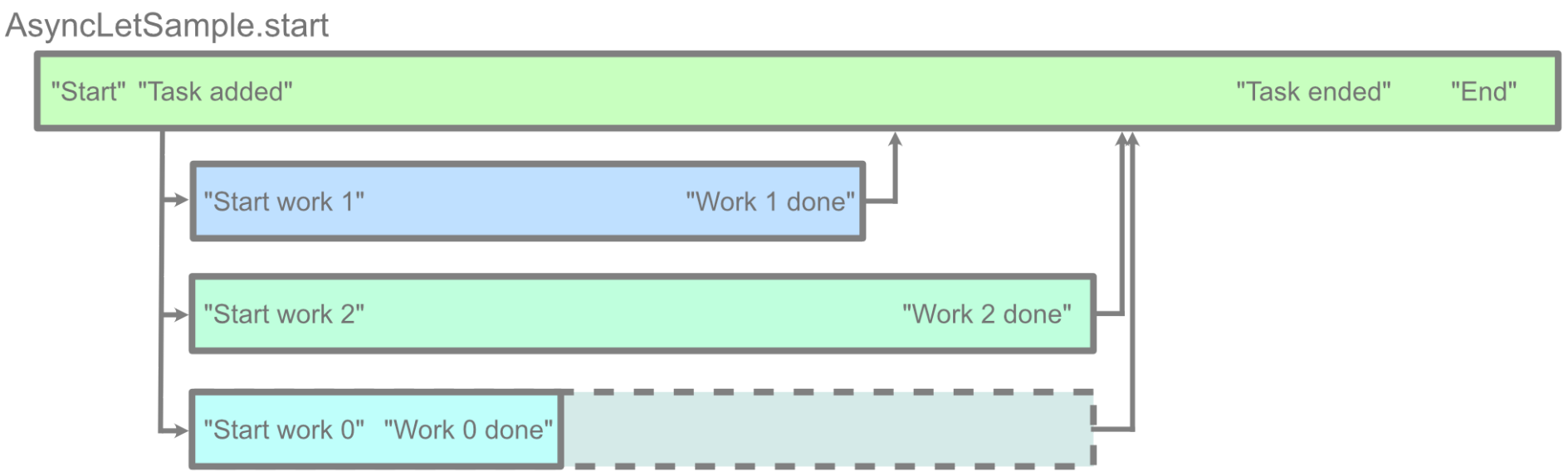
需要特别强调，虽然这里我们顺次进行了 `await` ，看起来好像是在等 `v0` 求值完毕后，再开始 `v1` 的暂停；然后在 `v1` 求值后再开始 `v2` 。但是实际上，在 `async let` 时，这些子任务就一同开始以并发的方式进行了。在例子中，完成 `work(n)` 的耗时为 `n` 秒，所以上面的写法将在第 0 秒，第 1 秒和第 2 秒分别得出 `v0` ， `v1` 和 `v2` 的值，**而不是**在第 0 秒，第 1 秒和第 3 秒 (1 秒 + 2 秒) 后才得到对应值。

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` ，它们所生成的子任务都是结构化的。不过，它们还有些许差别，我们马上就会谈到这个话题。

隐式取消

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，编译器也帮我们做了某些“手脚”，以保证单进单出的结构化并发依然成立。

如果没有 `await` ，那么 Swift 并发会在被绑定的常量离开作用域时，隐式地将绑定的子任务取消掉，然后进行 `await` 。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)")
    }
  }
}
```

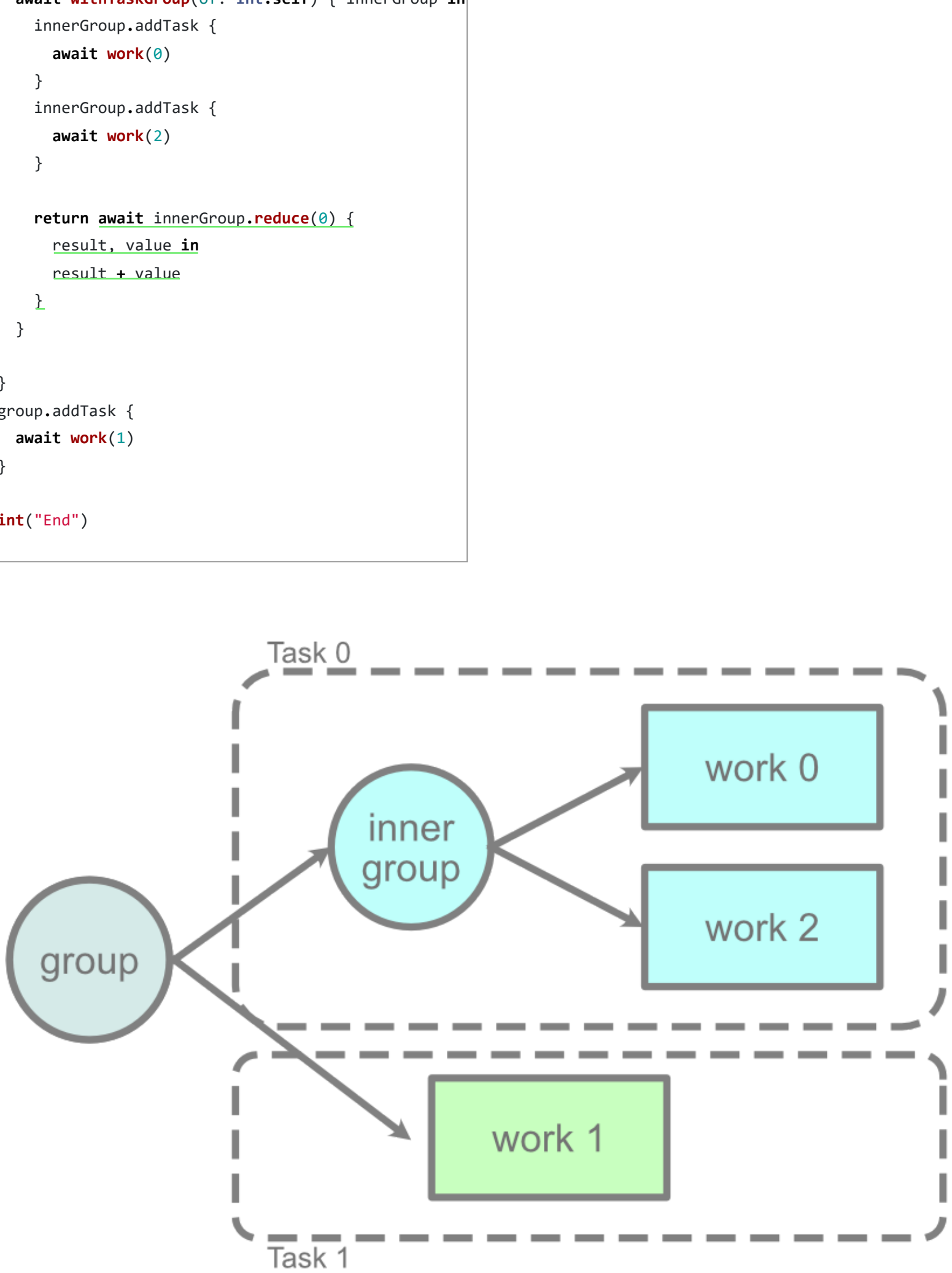
除了上面那些只能使用某一种方式创建的结构化并发任务外，对于可以互换的情况，任务组 API 和异步绑定 API 的区别在于提供了两种不同风格的编程方式。一个大致的使用原则是，如果我们需要比较“严肃”地界定结构化并发的起始，那么用任务组的闭包将它限制起来，并发的结构会显得更加清晰；而如果我们只是想要快速地并发开始少数几个任务，并减少其他模板代码的干扰，那么使用 `async let` 进行异步绑定，会让代码更简洁易读。

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` ：

```swift
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为的机会。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，可能结构会更清楚。

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，两者混用也不会有什么问题。

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。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：

```swift
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。它们生成一棵新的任务树，并位于顶层，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。对比三者，可以看出它们之间明显的不同：

- `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。

有一种迷思认为，我们在新建根节点任务时，应该尽量使用 `Task.init` 而避免选用生成一个完全“游离任务”的 `Task.detached` 。其实这并不全然正确，有时候我们希望从当前任务环境中继承一些事实，但也有时候我们确实想要一个“干净”的任务环境。比如 `@main` 标记的异步程序入口和 SwiftUI `task` 修饰符，都使用的是 `Task.detached` 。具体是不是有可能从当前任务环境中继承属性，或者应不应该继承这些属性，需要具体问题具体分析。

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：

```swift
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` ：

```swift
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` 值超过作用域并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` ：

```swift
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，并不会传递到内层 `Task` 。或者，更准确来说，这样的两个 `Task` 并没有任何从属关系，它们都是顶层任务：

```
let outer = Task {
  let innner = 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，而舍弃掉原有的非结构化并发模型呢？现在有这个趋势，但是大家也都还保留了原来的并发模型。即使要完全转变，可能也还需要一些时间。

Swift 是当前少数几个在语言和标准库层面对结构化并发进行支持的语言之一。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的工具。

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。

能工巧匠集, Swift

swift 编程语言 并发