

Benny Huo

学海无涯，其乐无穷



闲话 Swift 协程 (7) : GlobalActor 和异步函数的调度

📅 2022-02-12 | 📅 2022-10-22 | 👁 494

📖 8.8k | ⌚ 16 分钟

我们已经知道可以使用 actor 来确保数据的线程安全，但对于数据的保护总是需要定义专门的 actor 实例是不是太麻烦了一些？

- [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](#)

什么是 GlobalActor

前面我们为了保护特定的状态，就把这些状态包装到一个特定的 actor 实例当中，保护的方式就是将对于这些状态的访问调度到相应的 actor 的调度器当中串行执行。

那么问题来了，如果我有很多分散到不同类甚至不同模块的状态，希望统一调度，该怎么办？最典型的例子就是将 UI 操作调度到主线程，UI 本身就分散在不同的组件当中，对于 UI 的操作更是如此。为了应对这种场景，Swift 在提供了 actor 的基础上又进一步提供了 GlobalActor，旨在提供全局统一的执行调度。



GlobalActor 是一个协议，我们来看一下它的定义：

```
1 public protocol GlobalActor {
2
3     associatedtype ActorType : _Concurrency.Actor
4
5     static var shared: Self.ActorType { get }
6
7     static var sharedUnownedExecutor: _Concurrency.UnownedSerialExecutor { get }
8 }
```

- ActorType 是实现 GlobalActor 协议的类型需要提供的全局唯一的 actor 的类型
- shared 是上述全局唯一的 actor 的实例
- sharedUnownedExecutor 是上述全局唯一的 actor 实例的调度器，它的值要求与 shared.unownedExecutor 一致

所以只要确定了 shared 是谁，那么这个 GlobalActor 也就确定了。

此外，一个类型在实现 GlobalActor 时，我们还可以用 @globalActor 来修饰它，这样我们就可以用这个实现类去修饰需要使用该 GlobalActor 的实现类来隔离的函数或者属性了。这么说比较抽象，我们接下来看看官方目前提供的唯一一个 GlobalActor 实现是怎么定义的。

探索 MainActor

MainActor 是目前唯一一个 GlobalActor 的实现，它用来将对属性或者函数的访问隔离到主线程上执行。我们来看看它的定义：

```
1 @globalActor public final actor MainActor: GlobalActor {
2     public static let shared = MainActor()
3
4     @inlinable
5     public nonisolated var unownedExecutor: UnownedSerialExecutor {
6         return UnownedSerialExecutor(Builtin.buildMainActorExecutorRef())
7     }
8
9     @inlinable
10    public static var sharedUnownedExecutor: UnownedSerialExecutor {
11        return UnownedSerialExecutor(Builtin.buildMainActorExecutorRef())
12    }
13 }
```



```

14     ...
15 }

```

我们看到 `MainActor` 是一个 `actor` 类型, 这对于 `GlobalActor` 协议来说不是必须的, 我们完全可以定义一个 `class` 来实现 `GlobalActor`, 并且把一个 `actor` 类型关联到 `GlobalActor` 上即可。当然, 如果条件允许, 直接用 `actor` 类型来实现 `GlobalActor` 自然更方便一些。

我们在前面提到过, `sharedUnownedExecutor` 要与 `shared.unownedExecutor` 一致, 这里很显然二者本质上都是 `Builtin.buildMainActorExecutorRef()`。

此外, `MainActor` 被 `@globalActor` 修饰之后, 自己就可以被用于修饰属性、函数或者类型, 我们给出几个简单的 `MainActor` 的例子:

修饰属性:

```

1 class State {
2     @MainActor var value: Int = 0
3 }

```

修饰函数:

```

1 @MainActor func calledOnMain() {
2     ...
3 }

```

修饰闭包:

```

1 func runOnMain(block: @MainActor @escaping () -> Void) async {
2     log("runOnMain before")
3     await block()
4     log("runOnMain after")
5 }

```

修饰类:

```

1 @MainActor
2 class UiState {
3     var value: Int = 0
4

```



```
5     func update(value: Int) {  
6         self.value = value  
7     }  
8 }
```

被 `@MainActor` 修饰的函数在调用时，如果当前不在主线程，则必须异步调度到主线程上执行；同样地，被修饰的属性在被其他线程访问时，也必须异步调度到主线程上处理。

被 `@MainActor` 修饰的类的构造器、属性、函数都需要调度到主线程上执行。需要注意的是，为了保证继承的一致性，被修饰的类需要满足或没有父类、或同样被 `@MainActor` 修饰、或父类是 `NSObject`；被修饰的类的子类也将会隐式获得 `@MainActor` 上的状态隔离。

这里的异步访问逻辑实际上与 `actor` 类型的状态和函数的关系相同，即被 `@MainActor` 修饰的函数内部访问同样被 `@MainActor` 修饰的属性时则不需要异步执行，就好像它们都被定义到 `MainActor` 这个 `actor` 类型当中一样。

以上使用方法和细节同样适用于其他 `GlobalActor` 的实现。

自定义 GlobalActor 的实现

了解了 `MainActor` 的定义之后，我们就可以试着给出自定义的 `GlobalActor` 实现了，例如：

```
1  @globalActor actor MyActor: GlobalActor {  
2  
3      // 实现 GlobalActor 协议当中的 associatedtype  
4      public typealias ActorType = MyActor  
5  
6      // 实现 GlobalActor 当中的 shared，返回一个全局共享的 MyActor 实例  
7      static let shared: MyActor = MyActor()  
8  
9      private static let _sharedExecutor = MyExecutor()  
10  
11     // 实现 GlobalActor 当中的 sharedUnownedExecutor，返回自己的调度器  
12     static let sharedUnownedExecutor: UnownedSerialExecutor = _sharedExecutor.asUn  
13  
14     // 显示实现 Actor 协议当中的调度器，避免让编译器自动生成  
15     let unownedExecutor: UnownedSerialExecutor = sharedUnownedExecutor  
16 }
```

其中自定义的调度器 `MyExecutor` 的定义如下:

```
1 final class MyExecutor : SerialExecutor {
2
3     // 自定义 DispatchQueue, 用于真正地调度异步函数
4     private static let dispatcher: DispatchQueue = DispatchQueue(label: "MyActor")
5
6     // 需要调度时, Swift 的协程运行时会创建一个 UnownedJob 实例调用 enqueue 进行调度
7     func enqueue(_ job: UnownedJob) {
8         log("enqueue")
9         MyExecutor.dispatcher.async {
10             // 执行这个 job
11             job._runSynchronously(on: self.asUnownedSerialExecutor())
12         }
13     }
14
15     // 获取 unowned 引用, 得到 UnownedSerialExecutor 实例
16     func asUnownedSerialExecutor() -> UnownedSerialExecutor {
17         UnownedSerialExecutor(ordinary: self)
18     }
19 }
```

大家可以简单阅读代码的注释来了解他们的作用。注意到 `MyActor` 也实现了 `GlobalActor` 协议, 我们也使用 `@globalActor` 来修饰 `MyActor`, 这样我们就可以用 `@MyActor` 像 `@MainActor` 那样去修饰函数、属性和类, 并让它调度到我们自己实现的调度器上了。

有关 `MyActor` 的使用示例, 我们将在下一节进一步讨论。

注意 截至本文撰写时, Swift 的最新版本为 5.5.1。当前 Swift 协程对于自定义调度器的支持还在提案阶段, 细节可参见: [Custom Executors](#)。

深入探讨 Actor 与协程的调度

Swift 的协程在执行调度问题上目前还比较含蓄, 文档当中很少提及异步函数的执行以及异步函数返回时如何恢复。实际上, 异步函数所在的调用位置会关联一个调度器, 这个调度器要么来自于所在的 Task, 要么来自于当前函数所属于的 actor 实例。

Swift 定义了两个默认的调度器, 一个是并发的, 一个是串行的; 另外就是我们前面提到的, 用于将异步函数调度到主线程上的主线程的调度器。

为了搞清楚 Swift 协程究竟是如何调度的，我们用 `MainActor` 和自定义的 `MyActor` 来调度我们的异步函数，看看有什么新发现。

在下面的例子当中，我们使用 `@MainActor` 修饰函数 `calledOnMain`：

```
1  @MainActor func calledOnMain() {
2      log("onMain")
3  }
```

接下来创建一个 `Task` 来调用它：

```
1  Task { () -> Int in
2      log("task start")
3      await calledOnMain()
4      log("task end")
5      return 1
6  }
```

这里我们使用 `log` 这个定义的函数来打印输出，它与 `print` 的不同之处在于它会同时打印当前线程：

```
1  [<NSThread: 0x6000015c41c0>{number = 2, name = (null)}] task start
2  [<_NSMainThread: 0x6000015c4080>{number = 1, name = main}] onMain
3  [<NSThread: 0x6000015c41c0>{number = 2, name = (null)}] task end
```

可以看到，`calledOnMain` 被调度到了 `MainThread` 上执行。`task start` 和 `task end` 执行所在的线程相同（当然也可以不同，但一定是相同的调度器所属的线程），这说明 `calledOnMain` 返回之后 `Task` 又被调度与之关联的调度器上执行。

`@MainActor` 也可以被用于修饰闭包的类型，例如：

```
1  func runOnMain(block: @MainActor @escaping () async -> Void) async {
2      log("runOnMyExecutor start")
3      await block()
4      log("runOnMyExecutor end")
5  }
```

我们试着调用一下这个函数：



```

1 Task { () -> Int in
2     log("task start")
3     await runOnMain {
4         log("on main")
5     }
6     log("task end")
7     return 1
8 }

```

运行结果如下：

```

1  [<NSThread: 0x600000ac8480>{number = 2, name = (null)}] task start
2  [<NSThread: 0x600000ac8480>{number = 2, name = (null)}] runOnMain before
3  [<_NSMainThread: 0x600000ac8380>{number = 1, name = main}] on main
4  [<NSThread: 0x600000ac8480>{number = 2, name = (null)}] runOnMain after
5  [<NSThread: 0x600000ac8480>{number = 2, name = (null)}] task end

```

这次只有 block 才会被调度到 MainThread 上，因为只有它被 @MainActor 修饰。

从这个例子当中我们其实还能推测出调度发生的位置，即：

- 异步函数开始执行
- 异步函数返回之处

实际上除此之外，Task 开始时也可能会发生一次调度。这些都是可能的调度位置，Swift 的运行时会根据实际情况判断调度前后是不是属于同一个调度器，以决定是不是真的需要发生调度。这些也能从我们待会儿的例子当中得到印证。

接下来我们使用 MyActor 依样画葫芦，完成类似的例子：

首先是函数的定义：

```

1 func runOnMyExecutor(block: @MyActor @escaping () async -> Void) async {
2     log("runOnMyExecutor start")
3     await block()
4     log("runOnMyExecutor end")
5 }
6
7 @MyActor func calledOnMyExecutor() {

```



```

8     log("onMyExecutor")
9 }

```

然后调用它们：

```

1 Task { () -> Int in
2     log("task start")
3     await calledOnMyExecutor()
4
5     await runOnMyExecutor {
6         log("on MyExecutor before sleep")
7         await Task.sleep(1000_000_000)
8         log("on MyExecutor after sleep")
9     }
10    log("task end")
11    return 1
12 }

```

运行结果如下：

```

1  [<NSThread: 0x600003eb4040>{number = 2, name = (null)}] task start
2  [<NSThread: 0x600003eb4040>{number = 2, name = (null)}] enqueue
3  [<NSThread: 0x600003eb4040>{number = 2, name = (null)}] onMyExecutor
4  [<NSThread: 0x600003eb4040>{number = 2, name = (null)}] runOnMyExecutor start
5  [<NSThread: 0x600003eb4040>{number = 2, name = (null)}] enqueue
6  [<NSThread: 0x600003eb4040>{number = 2, name = (null)}] on MyExecutor before sleep
7  [<NSThread: 0x600003eb8040>{number = 3, name = (null)}] enqueue
8  [<NSThread: 0x600003eb8040>{number = 3, name = (null)}] on MyExecutor after sleep
9  [<NSThread: 0x600003eb8040>{number = 3, name = (null)}] runOnMyExecutor end
10 [<NSThread: 0x600003eb8040>{number = 3, name = (null)}] task end

```

注意到 `calledOnMyExecutor` 调用时、`runOnMyExecutor` 当中的 `block` 执行时、`block` 当中的 `sleep` 之后恢复时分别执行了一次 `enqueue`。大家有兴趣的话也可以在其中穿插一些需要调度到主线程的函数调用，看看实际的调度情况。

Task 与 actor 上下文

我们在前面介绍 Task 的构造时，讲到过可以使用 `init` 和 `detached` 两种方式来构造 Task 实例，前者会继承外部的上下文，包括 `actor`、`TaskLocal` 等，后者则不会。



下面的例子将会证明这其中有关 actor 的部分：

```

1 Task { () -> Int in
2     log("task start")
3     await runOnMain {
4         await Task {
5             log("task in runOnMain")
6         }.value
7
8         await Task.detached {
9             log("detached task in runOnMain")
10        }.value
11    }
12    log("task end")
13    return 1
14 }
```

通过前面的介绍，我们已经知道 runOnMain 的参数 block 会被调度到 MainThread 上执行，那么其中的两个 Task 的日志输出理论上会有不同的表现：

```

1  [<NSThread: 0x600001520180>{number = 2, name = (null)}] task start
2  [<NSThread: 0x600001520180>{number = 2, name = (null)}] runOnMyExecutor start
3  [<_NSMainThread: 0x600001520080>{number = 1, name = main}] task in runOnMain
4  [<NSThread: 0x600001520180>{number = 2, name = (null)}] detached task in runOnMain
5  [<_NSMainThread: 0x600001520080>{number = 1, name = main}] runOnMyExecutor end
6  [<_NSMainThread: 0x600001520080>{number = 1, name = main}] task end
```

实际上也正是如此，task in runOnMain 打印到了 MainThread 上，而 detached task in runOnMain 因为通过 detached 创建的 Task 实例不会继承外部的 actor（以及其调度器），因此打印到了其他线程上（也就是默认的调度器上）。

Task 的两种不同的构造方式对于 TaskLocal 的继承情况同样如此，我们将在下一篇文章当中再给出对比示例。

小结

本文我们详细介绍了 GlobalActor 的设计初衷、实现方式以及使用方法，也探讨了 Swift 协程的调度细节，相信读者看到这里时，已经掌握了绝大多数 Swift 协程的相关知识。



下一篇文章我们将简单介绍一下 TaskLocal 的使用方法。

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 \(6\) : Actor 和属性隔离](#)

[闲话 Swift 协程 \(8\) : TaskLocal](#) ▶




未找到相关的 [Issues](#) 进行评论

请联系 @bennyhuo 初始化创建

使用 GitHub 登录



京ICP备16022265号-3

© 2018 – 2022  Benny Huo |  476k |  14:25

由 Hexo & NexT.Pisces 强力驱动

