协程中的取消和异常 | 核心概念介绍

谷歌开发者
已认证的官方帐号

已关注

19 人赞同了该文章

在之前的文章里，我们为各位开发者分享了在 Android 中使用协程的一些基础知识，包括在 Android 协程的背景介绍、上手指南和代码实战。本次系列文章 "协程中的取消和异常" 也是 Android 协程相关的内容，我们将与大家深入探讨协程中关于取消操作和异常处理的知识点和技巧。

当我们需要避免多余的处理来减少内存浪费并节省电量时，取消操作就显得尤为重要；而妥善的异常处理也是提高用户体验的关键。本篇是另外两篇文章的基础 (第二篇和第三篇将为大家分别详解协程取消操作和异常处理)， 所以有必要先讲解一些协程的核心概念，比如 CoroutineScope (协程作用域)、Job (任务) 和 CoroutineContext (协程上下文)，这样我们才能够进行更深入的学习。

CoroutineScope

CoroutineScope会追踪每一个您通过launch或者async创建的协程 (这两个是 CoroutineScope 的扩展函数)。任何时候都可通过调用 scope.cancel() 来取消正在进行的工作 (正在运行的协程)。

当您希望在应用程序的某一个层次开启或者控制协程的生命周期时，您需要创建一个 CoroutineScope。对于一些平台，比如 Android，已经有KTX 这样的库在一些类的生命周期里提供了 CoroutineScope，比如viewModelScope和lifecycleScope。

当创建 CoroutineScope 的时候，它会将 CoroutineContext 作为构造函数的参数。您可以通过下面代码创建一个新的 scope 和协程:

```
//Job 和 Dispatcher 已经被集成到了 CoroutineContext
//后面我们详细介绍
val scope = CoroutineScope(Job() + Dispatchers.Main)

val job = scope.launch {
    //新的协程
}
```

Job

Job 用于处理协程。对于每一个您所创建的协程 (通过 launch 或者 async)，它会返回一个 Job实例，该实例是协程的唯一标识，并且负责管理协程的生命周期。正如我们上面看到的，您可以将 Job 实例传递给 CoroutineScope 来控制其生命周期。

CoroutineContext

CoroutineContext是一组用于定义协程行为的元素。它由如下几项构成:

- Job: 控制协程的生命周期；
- CoroutineDispatcher: 向合适的线程分发任务；
- CoroutineName: 协程的名称，调试的时候很有用；
- CoroutineExceptionHandler: 处理未被捕捉的异常，在未来的第三篇文章里会有详细的讲解。

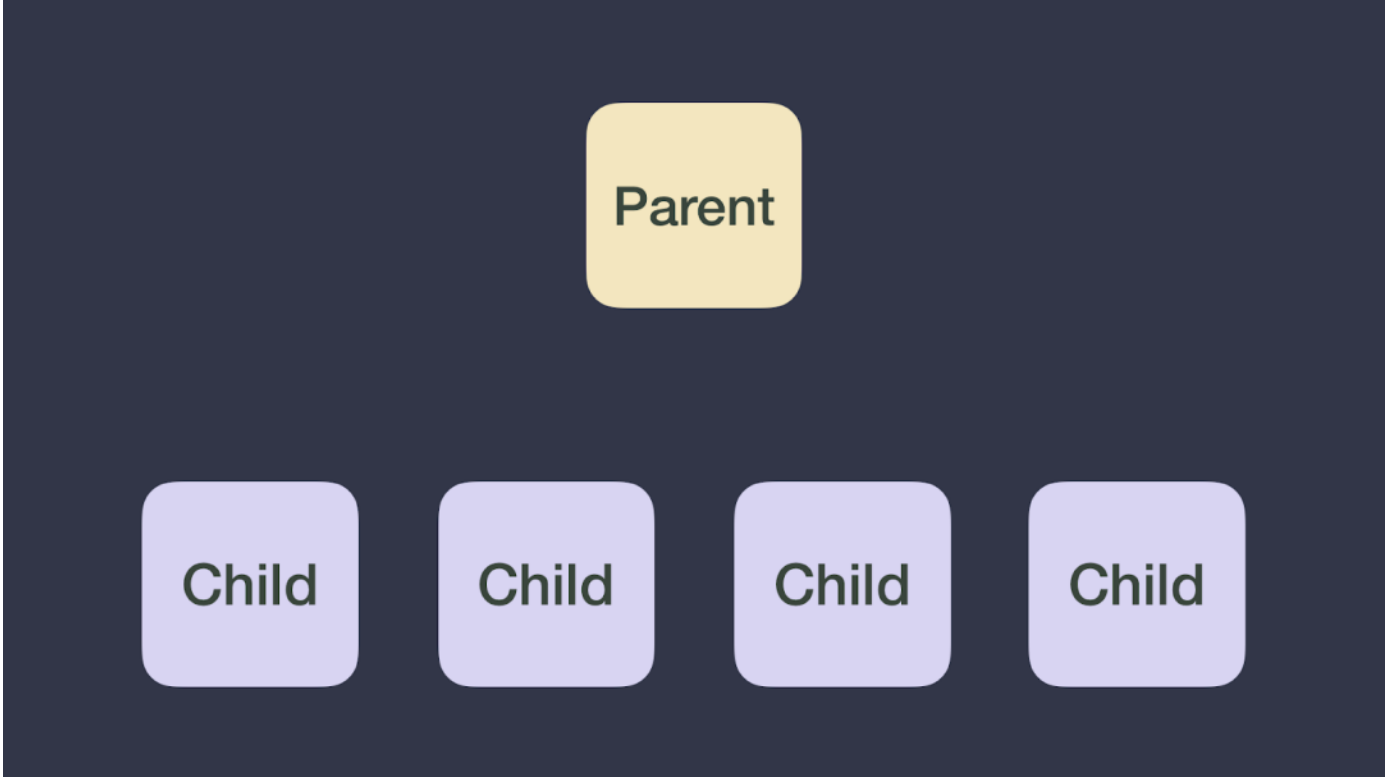
那么对于新创建的协程，它的 CoroutineContext 是什么呢？我们已经知道一个 Job 的实例会被创建，它会帮助我们控制协程的生命周期。而剩下的元素会从 CoroutineContext 的父类继承，该父类可能是另外一个协程或者创建该协程的 CoroutineScope。

由于 CoroutineScope 可以创建协程，而且您可以在协程内部创建更多的协程，因此内部就会隐含一个任务层级。在下面的代码片段中，除了通过 CoroutineScope 创建新的协程，来看看如何在协程中创建更多协程:

```
val scope = CoroutineScope(Job() + Dispatchers.Main)

val job = scope.launch {
    // 新的协程会将 CoroutineScope 作为父级
    val result = async {
        // 通过 launch 创建的新协程会将当前协程作为父级
    }.await()
}
```

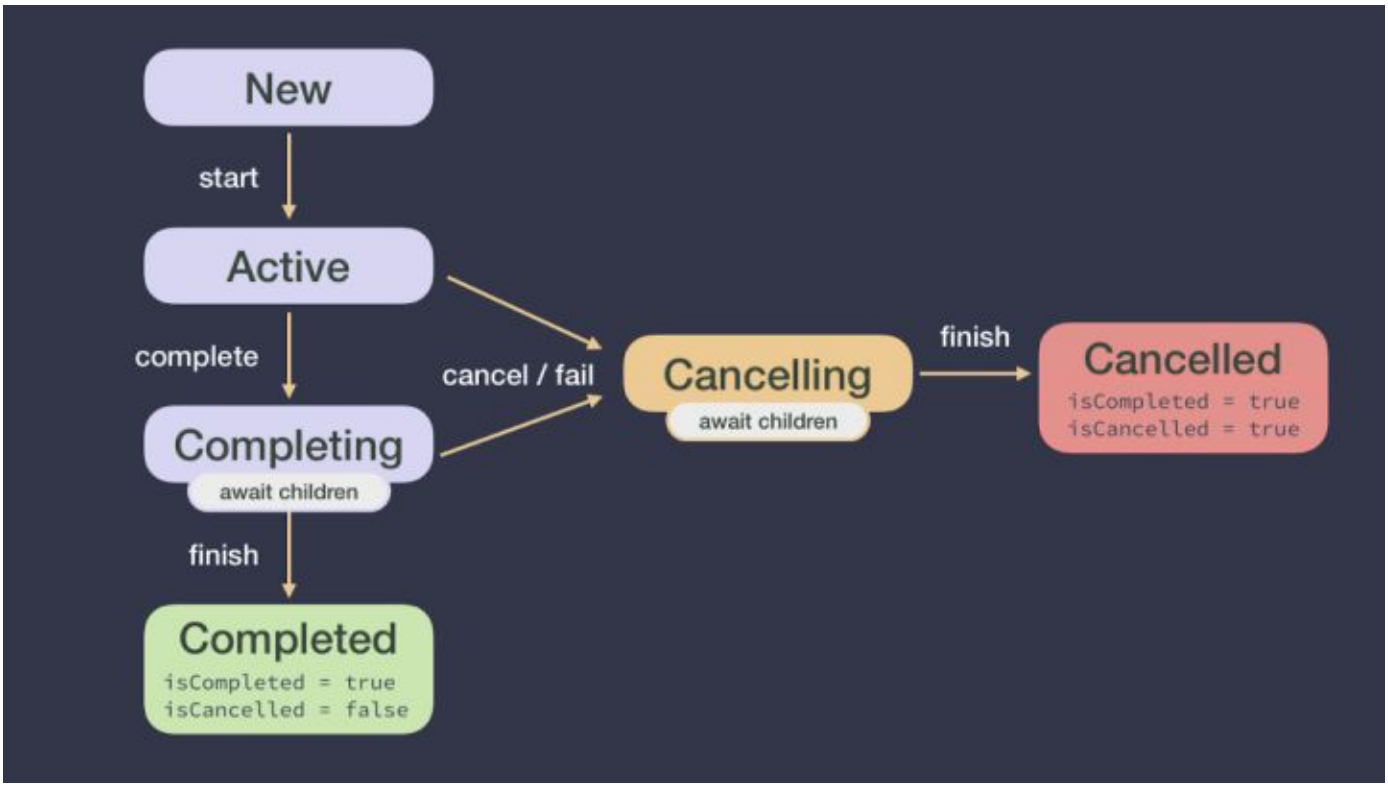
层级的根通常是 CoroutineScope。图形化该层级后如下图所示:



△ 协程是以任务层级为序执行的。父级是 CoroutineScope 或者其它协程

Job 的生命周期

一个任务可以包含一系列状态: 新创建 (New)、活跃 (Active)、完成中 (Completing)、已完成 (Completed)、取消中 (Cancelling) 和已取消 (Cancelled)。虽然我们无法直接访问这些状态，但是我们可以访问 Job 的属性: isActive、isCancelled 和 isCompleted。



△Job 的生命周期

如果协程处于活跃状态，协程运行出错或者调用 job.cancel() 都会将当前任务置为取消中 (Cancelling) 状态 (isActive = false, isCancelled = true)。当所有的子协程都完成后，协程会进入已取消 (Cancelled) 状态，此时 isCompleted = true。

解析父级 CoroutineContext

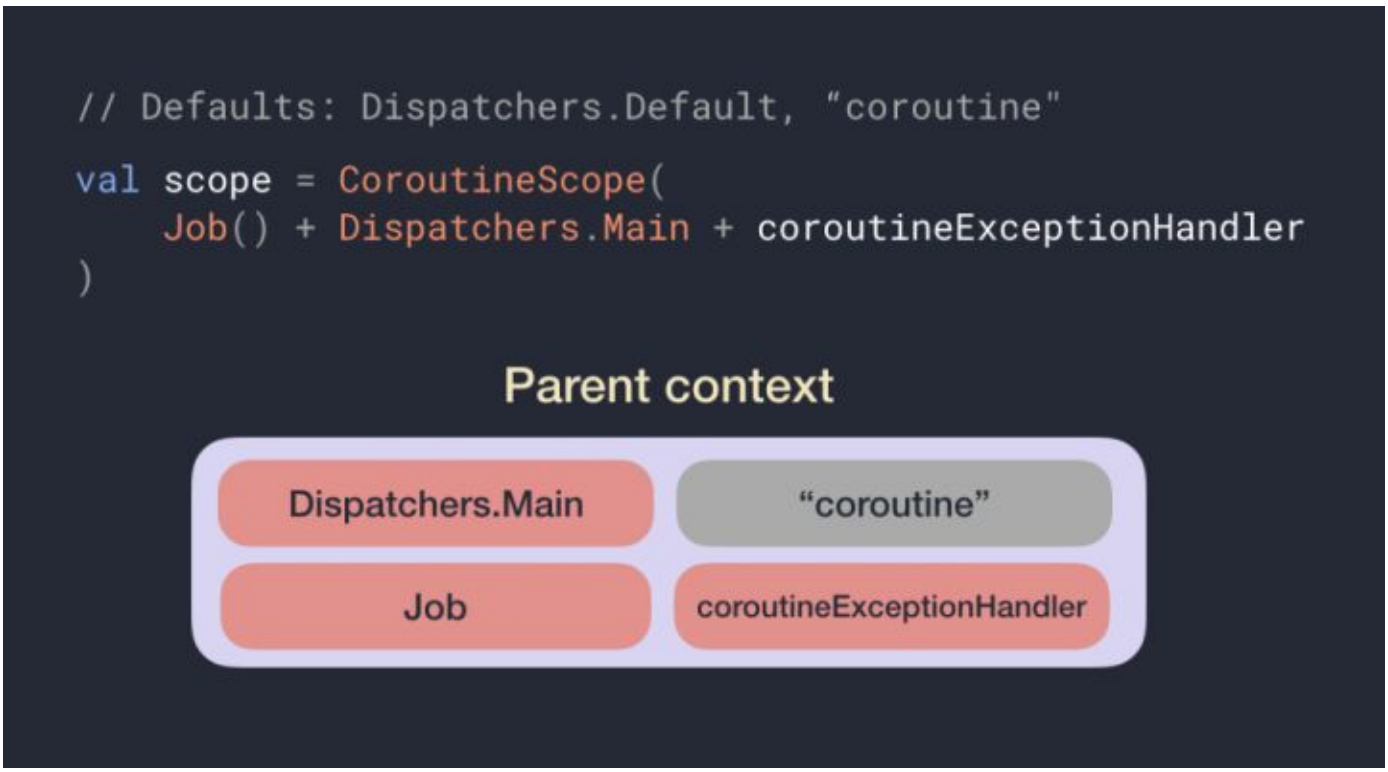
在任务层级中，每个协程都会有一个父级对象，要么是 CoroutineScope 或者另外一个 coroutine。然而，实际上协程的父级 CoroutineContext 和父级协程的 CoroutineContext 是不一样的，因为有如下的公式:

父级上下文= 默认值 + 继承的 CoroutineContext + 参数

其中:

- 一些元素包含**默认**值: Dispatchers.Default 是默认的 CoroutineDispatcher，以及 "coroutine" 作为默认的 CoroutineName；
- **继承的** CoroutineContext 是 CoroutineScope 或者其父协程的 CoroutineContext；
- 传入协程 builder 的**参数**的优先级高于继承的上下文参数，因此会覆盖对应的参数值。

请注意: CoroutineContext 可以使用 " + " 运算符进行合并。由于 CoroutineContext 是由一组元素组成的，所以加号右侧的元素会覆盖加号左侧的元素，进而组成新创建的 CoroutineContext。比如，(Dispatchers.Main, "name") + (Dispatchers.IO) = (Dispatchers.IO, "name")。



△该 CoroutineScope 所创建的每一个协程，CoroutineContext 至少会包含这些元素。这里的 CoroutineName 是灰色的，因为该值源于默认参数值。

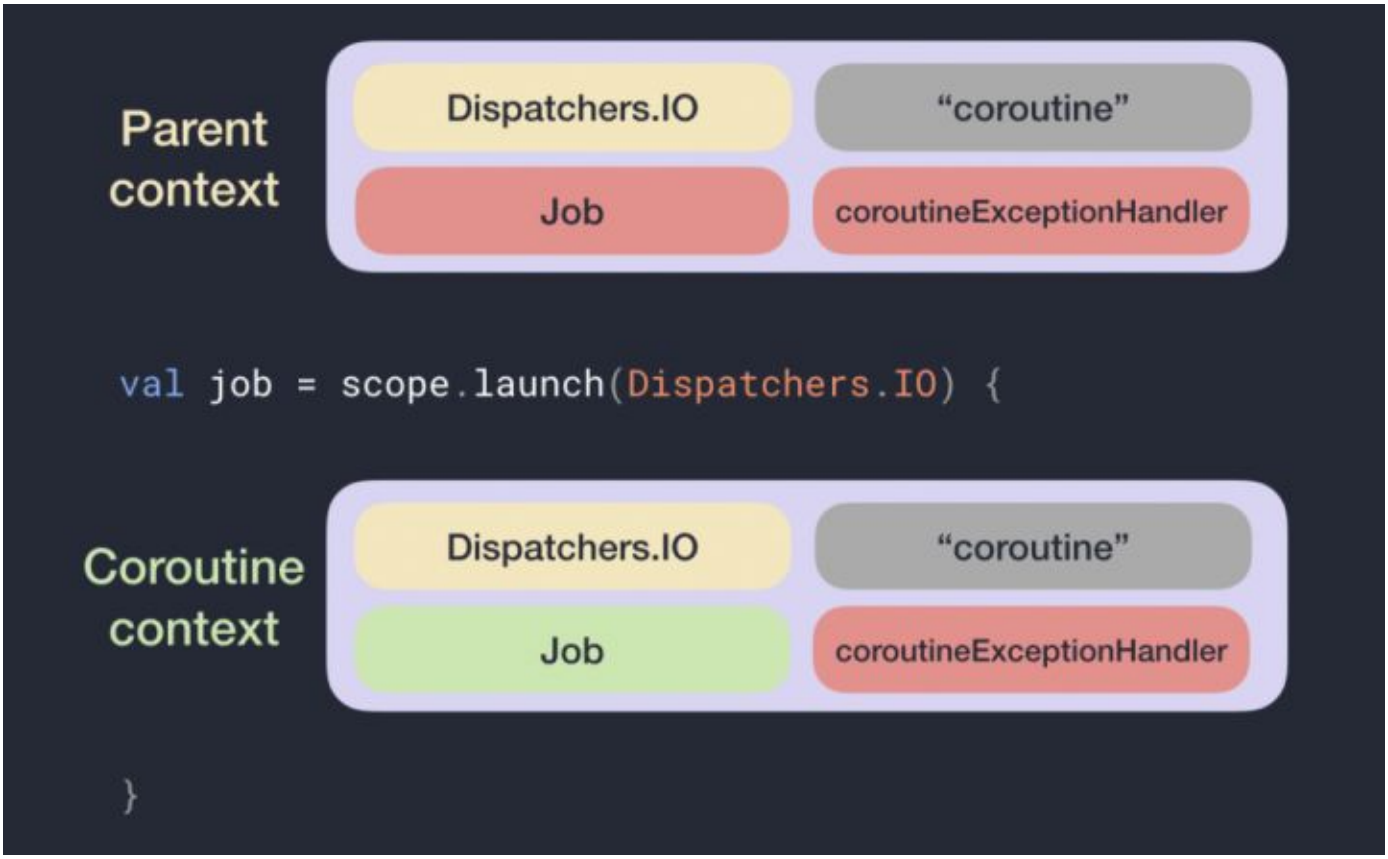
那么现在我们明白新协程的父级 CoroutineContext 是什么样的了，它实际的 CoroutineContext 是:

新的 CoroutineContext= 父级 CoroutineContext + Job()

如果使用上图中的 CoroutineScope ，我们可以像下面这样创建新的协程:

```
val job = scope.launch(Dispatchers.IO) {
    //新协程
}
```

而该协程的父级 CoroutineContext 和它实际的 CoroutineContext 是什么样的呢？请看下面这张图。



△ CoroutineContext 里的 Job 和父级上下文里的不可能是通过一个实例，因为新的协程总会拿到一个 Job 的新实例

最终的父级 CoroutineContext 会内含 Dispatchers.IO 而不是 scope 对象里的 CoroutineDispatcher，因为它被协程的 builder 里的参数覆盖了。此外，注意一下父级 CoroutineContext 里的 Job 是 scope 对象的 Job (红色)，而新的 Job 实例 (绿色) 会赋值给新的协程的 CoroutineContext。

在我们这个系列的第三部分中，CoroutineScope 会有另外一个 Job 的实现称为 SupervisorJob 被包含在其 CoroutineContext 中，该对象改变了 CoroutineScope 处理异常的方式。因此，由该 scope 对象创建的新协程会将一个 SupervisorJob 作为其父级 Job。不过，当一个协程的父级是另外一个协程时，父级的 Job 会仍然是 Job 类型。

现在，大家了解了协程的一些基本概念，在接下来的文章中，我们将在第二篇继续深入探讨协程的取消、第三篇探讨协程的异常处理，感兴趣的读者请继续关注我们的更新。

发布于 05-21

协程　　异步 I/O　　Android 工程师