在 Android 开发中使用协程 | 背景介绍

谷歌开发者
已认证的官方帐号

已关注

高爷等 28 人赞同了该文章

本文是介绍 Android 协程系列中的第一部分，主要会介绍协程是如何工作的，它们主要解决什么问题。

协程用来解决什么问题?

Kotlin 中的协程提供了一种全新处理并发的方式，您可以在 Android 平台上使用它来简化异步执行的代码。协程是从 Kotlin 1.3 版本开始引入，但这一概念在编程世界诞生的黎明之际就有了，最早使用协程的编程语言可以追溯到 1967 年的Simula语言。

在过去几年间，协程这个概念发展势头迅猛，现已经被诸多主流编程语言采用，比如Javascript、C#、Python、Ruby以及Go等。Kotlin 的协程是基于来自其他语言的既定概念。

在 Android 平台上，协程主要用来解决两个问题:

1. **处理耗时任务 (Long running tasks)**，这种任务常常会阻塞住主线程；
2. **保证主线程安全 (Main-safety)** ，即确保安全地从主线程调用任何 suspend 函数。

让我们来深入上述问题，看看该如何将协程运用到我们代码中。

处理耗时任务

获取网页内容或与远程 API 交互都会涉及到发送网络请求，从数据库里获取数据或者从磁盘中读取图片资源涉及到文件的读取操作。通常我们把这类操作归类为耗时任务 —— 应用会停下并等待它们处理完成，这会耗费大量时间。

当今手机处理代码的速度要远快于处理网络请求的速度。以 Pixel 2 为例，单个 CPU 周期耗时低于 0.0000000004 秒，这个数字很难用人类语言来表述，然而，如果将网络请求以 “眨眼间” 来表

述，大概是 400 毫秒 (0.4 秒)，则更容易理解 CPU 运行速度之快。仅仅是一眨眼的功夫内，或是一个速度比较慢的网络请求处理完的时间内，CPU 就已完成了超过 10 亿次的时钟周期了。

Android 中的每个应用都会运行一个主线程，它主要是用来处理 UI (比如进行界面的绘制) 和协调用户交互。如果主线程上需要处理的任务太多，应用运行会变慢，看上去就像是 “卡” 住了，这样是很影响用户体验的。所以想让应用运行上不 “卡”、做到动画能够流畅运行或者能够快速响应用户点击事件，就得让那些耗时的任务不阻塞主线程的运行。

要做到处理网络请求不会阻塞主线程，一个常用的做法就是使用回调。回调就是在之后的某段时间去执行您的回调代码，使用这种方式，请求 developer.android.google.cn 的网站数据的代码就会类似于下面这样:

```
class ViewModel: ViewModel() {
    fun fetchDocs() {
        get("developer.android.google.cn") { result ->
            show(result)
        }
    }
}
```

在上面示例中，即使 get 是在主线程中调用的，但是它会使用另外一个线程来执行网络请求。一旦网络请求返回结果，result 可用后，回调代码就会被主线程调用。这是一个处理耗时任务的好方法，类似于Retrofit这样的库就是采用这种方式帮您处理网络请求，并不会阻塞主线程的执行。

使用协程来处理协程任务

使用协程可以简化您的代码来处理类似 fetchDocs 这样的耗时任务。我们先用协程的方法来重写上面的代码，以此来讲解协程是如何处理耗时任务，从而使代码更清晰简洁的。

```
// Dispatchers.Main
suspend fun fetchDocs() {
    // Dispatchers.Main
    val result = get("developer.android.google.cn")
    // Dispatchers.Main
    show(result)
}
// 在接下来的章节中查看这段代码
suspend fun get(url: String) = withContext(Dispatchers.IO){/*...*/}
```

在上面的示例中，您可能会有很多疑问，难道它不会阻塞主线程吗？get 方法是如何做到不等待网络请求和线程阻塞而返回结果的？其实，是 Kotlin 中的协程提供了这种执行代码而不阻塞主线程的方法。

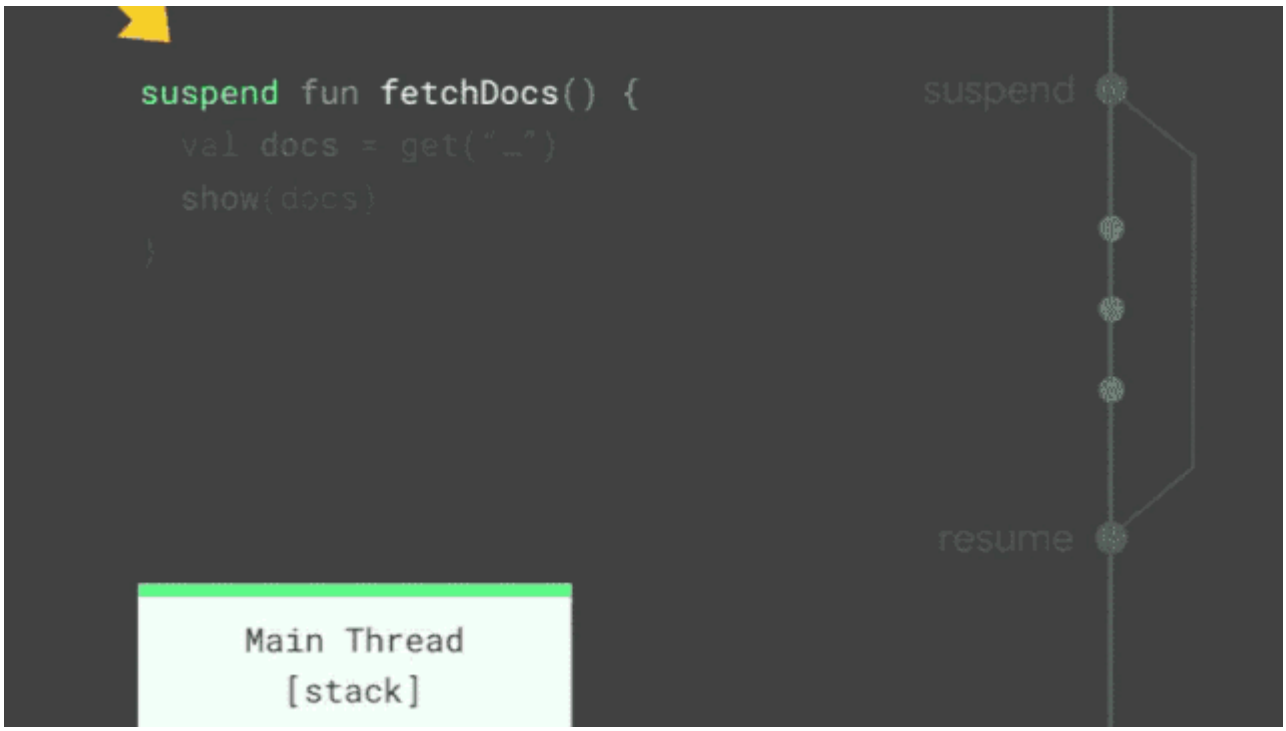
协程在常规函数的基础上新增了两项操作。在**invoke**(或 call) 和**return**之外，协程新增了**suspend**和**resume**:

- **suspend** — 也称挂起或暂停，用于暂停执行当前协程，并保存所有局部变量；
- **resume** — 用于让已暂停的协程从其暂停处继续执行。

Kotlin 通过新增 suspend 关键词来实现上面这些功能。您只能够在 suspend 函数中调用另外的 suspend 函数，或者通过协程构造器 (如launch) 来启动新的协程。

搭配使用 suspend 和 resume 来替代回调的使用。

在上面的示例中，get 仍在主线程上运行，但它会在启动网络请求之前暂停协程。当网络请求完成时，get 会恢复已暂停的协程，而不是使用回调来通知主线程。



上述动画展示了 Kotlin 如何使用 suspend 和 resume 来代替回调

观察上图中 fetchDocs 的执行，就能明白 **suspend** 是如何工作的。Kotlin 使用堆栈帧来管理要运行哪个函数以及所有局部变量。**暂停**协程时，会复制并保存当前的堆栈帧以供稍后使用。**恢复**协程时，会将堆栈帧从其保存位置复制回来，然后函数再次开始运行。在上面的动画中，当主线程下所有的协程都被暂停，主线程处理屏幕绘制和点击事件时就会毫无压力。所以用上述的 suspend 和 resume 的操作来代替回调看起来十分的清爽。

当主线程下所有的协程都被暂停，主线程处理别的事件时就会毫无压力。

即使代码可能看起来像普通的顺序阻塞请求，协程也能确保网络请求避免阻塞主线程。

接下来，让我们来看一下协程是如何保证主线程安全 (main-safety)，并来探讨一下调度器。

使用协程保证主线程安全

在 Kotlin 的协程中，主线程调用编写良好的 suspend 函数通常是安全的。不管那些 suspend 函数是做什么的，它们都应该允许任何线程调用它们。

但是在我们的 Android 应用中有很多的事情处理起来太慢，是不应该放在主线程上去做的，比如网络请求、解析 JSON 数据、从数据库中进行读写操作，甚至是遍历比较大的数组。这些会导致执行时间长从而让用户感觉很 “卡” 的操作都不应该放在主线程上执行。

使用 suspend 并不意味着告诉 Kotlin 要在后台线程上执行一个函数，这里要强调的是，协程会在主线程上运行。事实上，当要响应一个 UI 事件从而启动一个协程时，使用 Dispatchers.Main.immediate 是一个非常好的选择，这样的话哪怕是最终没有执行需要保证主线程安全的耗时任务，也可以在下一帧中给用户提供可用的执行结果。

协程会在主线程中运行，suspend 并不代表后台执行。

如果需要处理一个函数，且这个函数在主线程上执行太耗时，但是又要保证这个函数是主线程安全的，那么您可以让 Kotlin 协程在 Default 或 IO 调度器上执行工作。在 Kotlin 中，所有协程都必须在调度器中运行，即使它们是在主线程上运行也是如此。协程可以**自行暂停**，而调度器负责将其**恢复**。

Kotlin 提供了三个调度器，您可以使用它们来指定应在何处运行协程:

* 如果您在 Room 中使用了 suspend 函数、RxJava或者 LiveData，Room 会自动保障主线程安全。

** 类似于Retrofit 和 Volley 这样的网络库会管理它们自身所使用的线程，所以当您在 Kotlin 协程中调用这些库的代码时不需要专门来处理主线程安全这一问题。

接着前面的示例来讲，您可以使用调度器来重新定义 get 函数。在 get 的主体内，调用 withContext(Dispatchers.IO) 来创建一个在 IO 线程池中运行的块。您放在该块内的任何代码都始终通过 IO 调度器执行。由于 withContext 本身就是一个 suspend 函数，它会使用协程来保证主线程安全。

```
// Dispatchers.Main
suspend fun fetchDocs() {
    // Dispatchers.Main
    val result = get("developer.android.google.cn")
    // Dispatchers.Main
    show(result)
}
// Dispatchers.Main
suspend fun get(url: String) =
    // Dispatchers.Main
    withContext(Dispatchers.IO) {
        // Dispatchers.IO
    }
    // Dispatchers.Main
```

借助协程，您可以通过精细控制来调度线程。由于 withContext 可让您在不引入回调的情况下控制任何代码行的线程池，因此您可以将其应用于非常小的函数，如从数据库中读取数据或执行网络请求。一种不错的做法是使用 withContext 来确保每个函数都是主线程安全的，这意味着，您可以从主线程调用每个函数。这样，调用方就无需再考虑应该使用哪个线程来执行函数了。

在这个示例中，fetchDocs 会在主线程中执行，不过，它可以安全地调用 get 来在后台执行网络请求。因为协程支持 **suspend** 和 **resume**，所以一旦 withContext 块完成后，主线程上的协程就会恢复继续执行。

主线程调用编写良好的 suspend 函数通常是安全的。

确保每个 suspend 函数都是主线程安全的是很有用的。如果某个任务是需要接触到磁盘、网络，甚至只是占用过多的 CPU，那应该使用 withContext 来确保可以安全地从主线程进行调用。这也是类似于 Retrofit 和 Room 这样的代码库所遵循的原则。如果您在写代码的过程中也遵循这一点，那么您的代码将会变得非常简单，并且不会将线程问题与应用逻辑混杂在一起。同时，协程在这个原则下也可以被主线程自由调用，网络请求或数据库操作代码也变得非常简洁，还能确保用户在使用应用的过程中不会觉得 “卡”。

withContext 的性能

withContext 同回调或者是提供主线程安全特性的 RxJava 相比的话，性能是差不多的。在某些情况下，甚至还可以优化 withContext 调用，让它的性能超越基于回调的等效实现。如果某个函数需要对数据库进行 10 次调用，您可以使用外部 withContext 来让 Kotlin 只切换一次线程。这样一来，即使数据库的代码库会不断调用 withContext，它也会留在同一调度器并跟随快速路径，以此来保证性能。此外，在 Dispatchers.Default 和 Dispatchers.IO 中进行切换也得到了优化，以尽可能避免了线程切换所带来的性能损失。

下一步

本篇文章介绍了使用协程来解决什么样的问题。协程是一个计算机编程语言领域比较古老的概念，但因为它们能够让网络请求的代码比较简洁，从而又开始流行起来。

在 Android 平台上，您可以使用协程来处理两个常见问题:

1. 简化处理类似于网络请求、磁盘读取甚至是较大 JSON 数据解析这样的耗时任务；
2. 保证主线程安全，这样可以在不增加代码复杂度和保证代码可读性的前提下做到不会阻塞主线程的执行。

接下来的文章中我们将继续探讨协程在 Android 中是如何使用的，感兴趣的读者请继续关注。

点击这里**利用 Kotlin 协程提升应用性能**

发布于 04-08

Android 开发　Android