

Kotlin 协程入门这一篇就够了



MarioFeng LV.3

2019年06月20日 11:36 · 阅读 22451

关注

本篇文章已授权微信公众号 **guolin_blog**（郭霖）独家发布

协程的作用

协程通过替代回调(callback)来简化异步代码

听起来蛮抽象的，来看代码

```
fun fetchDocs() {  
    val result = get("developer.android.com")  
    show(result)  
}
```

kotlin 复制代码



@稀土掘金技术社区

Android系统为了保证界面的流畅和及时响应用户的输入事件，主线程需要保持每16ms一次的刷新(调用 `onDraw()` 函数)，所以不能在主线程中做耗时的操作(比如 读写数据库，读写文件，做网络请求，解析较大的 Json 文件，处理较大的 list 数据)。

`get()` 通过接口获取用户数据，如果在主线程中调用 `fetchDocs()` 函数就会阻塞(block)主线程，App 会卡顿甚至崩溃。

所以需要在子线程中调用 `get()` 函数，这样主线程就可以刷新界面和处理用户输入，待 `get()` 函数执行完毕后通过 `callback` 拿到结果。

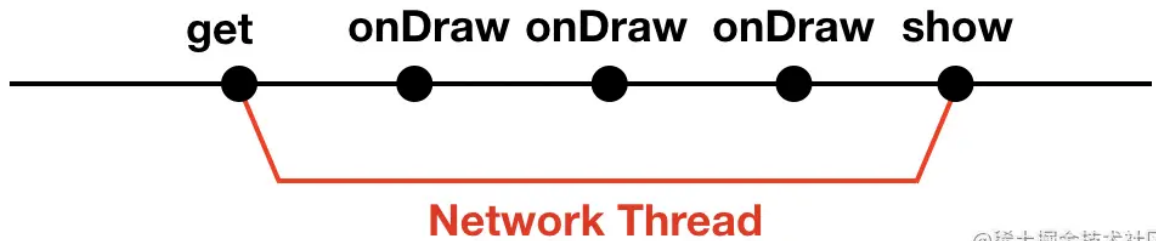
```
fun fetchDocs() {  
    get("developer.android.com") { result ->
```

kotlin 复制代码

```

        show(result)
    }
}

```



@稀土掘金技术社区

callback 是个不错的方式，但是 callback 被过度使用后代码可读性会变差（迷之缩进），而且 callback 不能使用 exception。为了解决这样的问题，欢迎协程（coroutine）闪亮登场

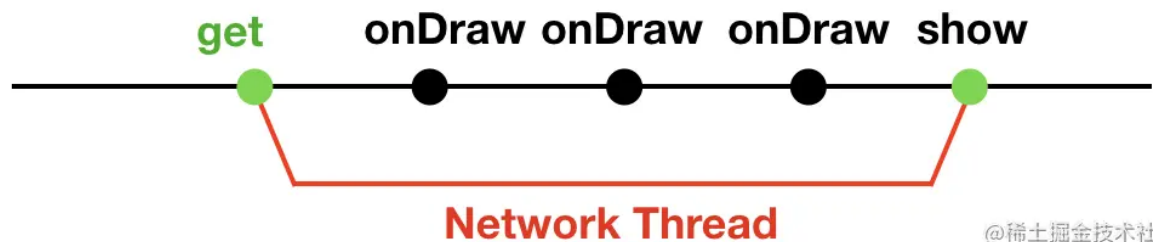
kotlin 复制代码

```

suspend fun fetchDocs() {
    val result = get("developer.android.com")
    show(result)
}

suspend fun get(url: String) =
    withContext(Dispatchers.IO) {
        ...
    }

```

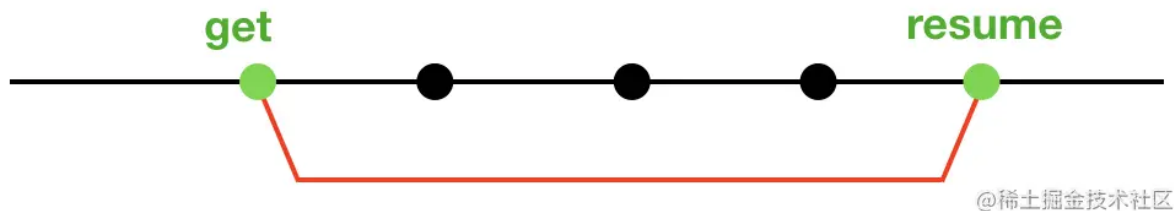


@稀土掘金技术社区

明明是同步的写法为什么不会阻塞主线程？对，因为 **suspend**

被 **suspend** 修饰的函数比普通函数多两个操作（suspend 和 resume）

- suspend：暂停当前协程的执行，保存所有的局部变量
- resume：从协程被暂停的地方继续执行协程



`get()` 函数同样也是一个 `suspend` 函数。

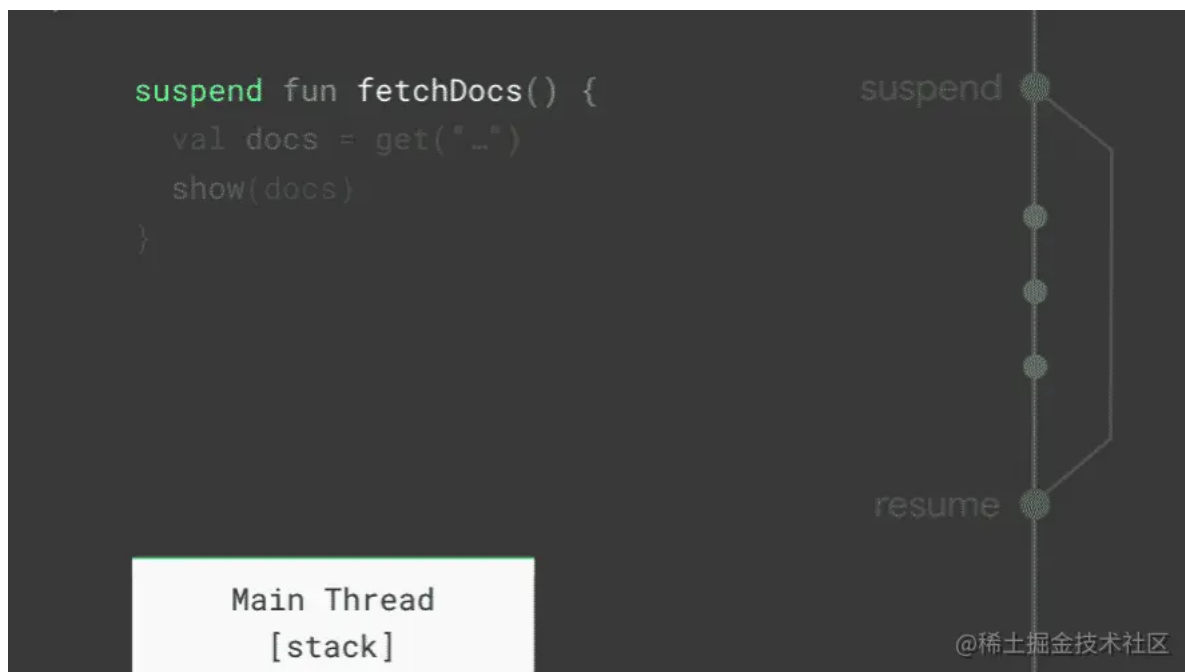
`suspend` 修饰的函数并不意味着运行在子线程中

如果需要指定协程运行的线程，就需要指定 [Dispatchers](#)，常用的有三种：

- `Dispatchers.Main`：Android中的主线程，可以直接操作UI
- `Dispatchers.IO`：针对磁盘和网络IO进行了优化，适合IO密集型的任务，比如：读写文件，操作数据库以及网络请求
- `Dispatchers.Default`：适合CPU密集型的任务，比如解析JSON文件，排序一个较大的list

通过 `withContext()` 可以指定Dispatchers，这里的 `get()` 函数里的 `withContext` 代码块中指定了协程运行在Dispatchers.IO中。

来看下这段代码的具体执行流程



动画出处见文末参考文档

- 每个线程有一个调用栈(call stack), Kotlin使用它来追踪哪个函数在执行和它的局部变量

- 当调用到 `suspend` 修饰的函数的时候，Kotlin需要追踪正在运行的协程而不是正在执行的函数
- 绿色线条表示一个 `suspend` 的标记，绿色上面的是协程，绿色下面的是一个正常的函数
- Kotlin 像正常函数一样调用 `fetchDocs()` 函数，在调用栈上加一个 entry，这里也存储着 `fetchDocs()` 函数的局部变量
- 继续往下执行，直到找到另一个 `suspend` 函数的调用（这里指的是 `get()` 函数调用），这时候Kotlin要去实现 `suspend` 操作（将函数的状态从堆栈复制到一个地方，以便以后保存，所有 `suspend` 的协程都会被放在这里）
- 然后调用 `get()` 函数，同样新建一个entry，当调用到 `withContext()`（`withContext`函数被 `suspend` 修饰）的时候，同样 执行suspend操作（过程和前面一样）。此时主线程里的所有协程都被 `suspend`，所以主线程可以做其他事情（执行 `onDraw`，响应用户输入）
- 等待几秒后，网络请求会返回，这时Kotlin会执行resume操作（获取保存状态并复制回来，重新放回到调用栈上），之后会正常往下执行，如果 `fetchDocs()` 发成错误，会在这里抛出异常

协程的组成

```
val viewModelJob = Job()    //用来取消协程
```

scss 复制代码

```
val uiScope = CoroutineScope(Dispatchers.Main + viewModelJob)    //初始化CoroutineScope 并
```

```
uiScope.launch { //启动一个协程
    updateUI() //suspend函数运行在协程内或者suspend另外一个函数内
}
```

```
suspend fun updateUI() {
    delay(1000L) //delay是一个 suspend 函数
    textView.text = "Hello, from coroutines!"
}
```

kotlin 复制代码

```
viewModelJob.cancel()//取消协程
```

scss 复制代码

- 启动一个协程需要 `CoroutineScope`，为什么需要？一会解释
- `CoroutineScope` 接受 `CoroutineContext` 作为参数，`CoroutineContext` 由一组协程的配置参数组成，可以指定协程的名称，协程运行所在线程，异常处理等等。可以通过 `plus` 操作符来组合这些参数。上面的代码指定了协程运行在主线程中，并且提供了一个 `Job`，可用于取消协程
 - `CoroutineName` (指定协程名称)

- **Job**（协程的生命周期，用于取消协程）
- **CoroutineDispatcher**，可以指定协程运行的线程
- 有了CoroutineScope之后可以通过一系列的 **Coroutine builders** 来启动协程，协程运行在 **Coroutine builders** 的代码块里面
 - launch 启动一个协程，返回一个 **Job**，可用来取消协程；有异常直接抛出
 - async 启动一个带返回结果的协程，可以通过Deferred.await()获取结果；有异常并不会直接抛出，只会在调用 await 的时候抛出
 - withContext 启动一个协程，传入 **CoroutineContext** 改变协程运行的上下文

结构化并发（**Structured concurrency**）

如果在 **foo** 里协程启动了 **bar** 协程，那么 **bar** 协程必须在 **foo** 协程之前完成



@稀土掘金技术社区

foo 里协程启动了 **bar** 协程，但是 **bar** 并没有在 **foo** 完成之前执行完成，所以不是结构化并发



@稀土掘金技术社区

foo 里协程启动了 **bar** 协程，并且 **bar** 在 **foo** 完成之前执行完成，所以是结构化并发

结构化并发能够带来什么优势呢？下面一点点阐述。

协程的泄漏

尽管协程本身是轻量级的，但是协程做的工作一般比较重，比如读写文件或者网络请求。使用代码手动跟踪大量的协程是相当困难的，这样的代码比较容易出错，一旦对协程失去追踪，那么就会导致泄漏。这比内存泄漏更加严重，因为失去追踪的协程在 `resume` 的时候可能会消耗内存，CPU，磁盘，甚至会进行不再必要的网络请求。

如何避免泄漏呢？这其实就是 `CoroutineScope` 的作用，通过 `launch` 或者 `async` 启动一个协程需要指定 `CoroutineScope`，当要取消协程的时候只需要调用 `CoroutineScope.cancel()`，kotlin 会帮我们自动取消在这个作用域里面启动的协程。

结构化并发可以保证当一个作用域被取消，作用域里面的所有协程会被取消

如果使用架构组件（Architecture Components），比较适合在 `ViewModel` 中启动协程，并且在 `onCleared` 回调方法中取消协程

```
override fun onCleared() {  
    super.onCleared()  
    viewModelJob.cancel() //取消ViewModel中启动的协程  
}
```

kotlin 复制代码

自己写 `CoroutineScope` 比较麻烦，架构组件提供了 `viewModelScope` 这个扩展属性，可以替代前面的 `uiScope`。

看下 `viewModelScope` 这个扩展属性是如何实现的：

```
val ViewModel.viewModelScope: CoroutineScope  
    get() {  
        val scope: CoroutineScope? = this.getTag(JOB_KEY)  
        if (scope != null) {  
            return scope  
        }  
        return setTagIfAbsent(JOB_KEY,  
            CloseableCoroutineScope(Job() + Dispatchers.Main))  
    }
```

kotlin 复制代码

```
internal class CloseableCoroutineScope(context: CoroutineContext) : Closeable, CoroutineScope {  
    override val coroutineContext: CoroutineContext = context  
  
    override fun close() {
```

```

        coroutineContext.cancel()
    }
}

```

同样是初始化一个CoroutinesScope，指定Dispatchers.Main和 Job

scss 复制代码

```

##ViewModel
@MainThread
final void clear() {
    mCleared = true;
    // Since clear() is final, this method is still called on mock objects
    // and in those cases, mBagOfTags is null. It'll always be empty though
    // because setTagIfAbsent and getTag are not final so we can skip
    // clearing it
    if (mBagOfTags != null) {
        for (Object value : mBagOfTags.values()) {
            // see comment for the similar call in setTagIfAbsent
            closeWithRuntimeException(value);
        }
    }
    onCleared();
}

private static void closeWithRuntimeException(Object obj) {
    if (obj instanceof Closeable) {
        try {
            ((Closeable) obj).close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
}

```

clear() 中会自动取消作用域中的协程。有了 **viewModelScope** 这个扩展属性可以少些很多模板代码。

再看一个稍复杂的场景，同时发起两个或者多个网络请求。这就意味着要开启更多的协程，随处开启协程可能导致潜在的泄漏问题，调用者可能不知道新开启的协程，因此也没法追踪他们。这时候就需要 **coroutineScope** 或者 **supervisorScope**（注意不是 **CoroutinesScope**）。

kotlin 复制代码

```

suspend fun fetchTwoDocs() {
    coroutineScope {
        launch { fetchDoc(1) }
        launch { fetchDoc(2) }
    }
}

```

```
}  
}
```

这个示例中，同时发起两个网络请求。在suspend 函数里面可以通过 `coroutineScope` 或 `supervisorScope` 安全地启动协程。为了避免泄漏，我们希望 `fetchTwoDocs` 这样的函数返回的时候，在函数内部启动的协程都能执行完成。

结构化并发保证当suspend函数返回的时候，函数里面的所有工作都已经完成


Kotlin可以保证使用 `coroutineScope` 不会从 `fetchTwoDocs` 函数中发生泄漏，`coroutineScope` 会 `suspend` 自己直到在它里面启动的所有协程执行完成。正是因为这样，`fetchTwoDocs` 不会在 `coroutineScope` 内部启动的协程完成前返回。

如果有更多的协程呢？

kotlin 复制代码

```
suspend fun loadLots() {  
    coroutineScope {  
        repeat(1000) {  
            launch { fetchDoc(it) }  
        }  
    }  
}
```

这里在suspend函数中启动了更多的协程，会泄露吗？并不会。



```
suspend fun loadLots() {  
    coroutineScope {  
        repeat(1_000) {  
            launch { fetchDocs() }  
        }  
    }  
}
```

动画出处见文末参考文档

由于这里的 `loadLots` 是一个 `suspend` 函数，所以 `loadLots` 函数会在一个 `CoroutineScope` 中被调用，`coroutineScope` 构造器会使用这个 `CoroutineScope` 作为父作用域生成一个新的 `CoroutineScope`。在 `coroutineScope` 代码块内部，`launch` 函数会在这个新的 `CoroutineScope` 中启动新的协程，这个新的 `CoroutineScope` 会追踪这些新的协程，当所有的协程执行完毕，`loadLots` 函数才会返回。

`coroutineScope` 和 `supervisorScope` 会等到所有的子协程执行完毕。

使用 `coroutineScope` 或者 `supervisorScope` 可以安全地在 `suspend` 函数里面启动新的协程，不会造成泄漏，因为总是会 `suspend` 调用者直到所有的协程执行完毕。`coroutineScope` 会新建一个子作用域（child scope），所以如果父作用域被取消，它会把取消的信息往下传递给所有新的协程。

另外 `coroutineScope` 和 `supervisorScope` 的区别在于：`coroutineScope` 会在任意一个协程发生异常后取消所有的子协程的运行，而 `supervisorScope` 并不会取消其他的子协程。

如何保证收到异常

前面有介绍过 `async` 里面如果发生异常是不会直接抛出的，直到 `await` 得到调用，所以下面的代码不会抛出异常。

kotlin 复制代码

```
val unrelatedScope = MainScope()
// example of a lost error
suspend fun lostError() {
    // async without structured concurrency
    unrelatedScope.async {
        throw InAsyncNoOneCanHearYou("except")
    }
}
```

但是 `coroutineScope` 会等到协程执行完毕，所以发生异常后会抛出。下面的代码会抛出异常。

kotlin 复制代码

```
suspend fun foundError() {
    coroutineScope {
        async {
            throw StructuredConcurrencyWill("throw")
        }
    }
}
```

```
}  
}
```

结构化并发保证当协程出错时，协程的调用者或者他的做用户会得到通知

由此可见 结构化并发可以保证代码更加安全，避免了协程的泄漏问题

- 当作用域被取消，里面所有的协程被取消，因而可以取消不再需要的任务
- 当 `suspend` 函数返回，里面的工作能保证完成，因而可以追踪正在执行的任务
- 当协程出错，调用者或者作用域会收到通知，从而可以进行异常处理

参考文档：

[Coroutines on Android \(part I\): Getting the background](#)

[Coroutines on Android \(part II\): Getting started](#)

[Understand Kotlin Coroutines on Android \(Google I/O'19\)](#)

[Using Kotlin Coroutines in your Android App](#)

分类： Android 标签： Kotlin

安装掘金浏览器插件

多内容聚合浏览、多引擎快捷搜索、多工具便捷提效、多模式随心畅享，你想要的，这里都有！

[前往安装](#)