WorkManager 在 Kotlin 中的实践

谷歌开发者
已认证的官方帐号

已关注

8 人赞同了该文章

WorkManager 是一个 Android Jetpack 扩展库，它可以让您轻松规划那些可延后、异步但又需要可靠运行的任务。对于绝大部分后台执行任务来说，使用 WorkManager 是目前 Android 平台上的最佳实践。

目前为止 WorkManager 系列已经讨论过:

- Android Jetpack WorkManager | Android 中文教学视频

在这篇文章中，我们将讨论:

- 在 Kotlin 中如何使用 WorkManager
- CoroutineWorker 类
- 如何使用 TestListenableWorkerBuilder 测试您的 CoroutineWorker 类

Kotlin 版的 WorkManager

本文的示例代码是用 Kotlin 编写的并使用了 KTX 库 (Kotlin Extensions)。KTX 版的 WorkManager 提供了更简洁且惯用的 Kotlin 扩展函数。如 WorkManager 发布日志 中描述的那样，只需要在 build.gradle 文件中添加 androidx.work:work-runtime-ktx 依赖项，即可使用 KTX 版的 WorkManager。该组件包含 CoroutineWorker 和其他有用的 WorkManager 扩展方法。

更简洁且惯用

当您需要构造一个数据对象，并且需要将它传入Worker 类或者从 Worker 类返回时，KTX 版 WorkManager 提供了一种语法糖。在这种情况下，用 Java 语法实现的代码如下所示:

```
Data myData = new Data.Builder()
                    .putInt(KEY_ONE_INT, aInt)
                    .putIntArray(KEY_ONE_INT_ARRAY, aIntArray)
                    .putString(KEY_ONE_STRING, aString)
                    .build();
```

而在 Kotlin 中，我们可以借助 workDataOf 辅助函数将代码写的更简洁:

```
inline fun workDataOf(vararg pairs: Pair<String, Any?>): Data
```

因此可以将前面的 Java 表达式改写成:

```
val data = workDataOf(
        KEY_MY_INT to myIntVar,
        KEY_MY_INT_ARRAY to myIntArray,
        KEY_MY_STRING to myString
    )
```

CoroutineWorker

除了可以使用 Java 实现的 Worker 类 (**Worker**、**ListenableWorker** 和 **RxWorker**) 之外，还有唯一一个使用 **Kotlin** 协程实现的 Work 类——**CoroutineWorker**。

Worker 类与 CoroutineWorker 类的主要区别在于: CoroutineWorker 类的 doWork() 方法是一个可以执行异步任务的挂起函数，而 Worker 类的 doWork() 方法只能执行同步任务。CoroutineWorker 的另一个特性是可以自动处理任务的暂停和取消，而 Worker 类需要实现 onStopped() 方法来处理这些情况。

获得完整上下文信息，请参阅官方文档 **在 WorkManager 中进行线程处理**。在这里，我想重点介绍一下什么是 CoroutineWorker，并且涵盖一些细小的但很重要的区别，以及深入了解如何使用在 WorkManager v2.1 中引入的新测试特性，来测试您的 CoroutineWorker 类。

正如前面写的那样，**CoroutineWorker#doWork()** 只是一个挂起函数。它默认是在 Dispatchers.Default 上启动的:

```
class MyWork(context: Context, params: WorkerParameters) :
        CoroutineWorker(context, params) {
override suspend fun doWork(): Result {
return try {
            // 做点什么
            Result.success()
        } catch (error: Throwable) {
            Result.failure()
        }
    }
}
```

需要切记的是，这是使用 CoroutineWorker 代替 Worker 或 ListenableWorker 时的根本区别:

> 与 Worker 不同，此代码不会在 WorkManager 的 Configuration 中指定的 Executor 上运行。

正如刚才所说，CoroutineWorker#doWork() 默认是在 Dispatchers.Default 启动的。您可以使用 withContext() 对此配置进行自定义。

```
class MyWork(context: Context, params: WorkerParameters) :
        CoroutineWorker(context, params) {
override suspend fun doWork(): Result = withContext(Dispatchers.IO) {
return try {
            // 做点什么
            Result.success()
        } catch (error: Throwable) {
            Result.failure()
        }
    }
}
```

> *很少需要改变 CoroutineWorker 使用的 Dispatcher，因为 Dispatchers.Default 可以满足大多数情况下的需求。*

要了解关于如何在 Kotlin 中使用 WorkManager，可以尝试这个 **codelab**。

测试 Worker 类

WorkManager 有几个额外的工具类，可以很方便地测试您的 Work。您可以在 **WorkManager 测试文档页面**和新的 **使用 WorkManager 2.1.0 进行测试** 的指南中了解更多相关信息。测试工具的原始实现使得自定义 WorkManager 成为可能，这样一来我们便可以使其表现为同步执行，进而可以使用 **WorkManagerTestInitHelper#getTestDriver()** 来模拟延迟和测试周期性任务。

WorkManager v2.1 版中增加了一个新的工具类: TestListenableWorkerBuilder，它引入了一种全新的测试 Worker 类的方式。

这对于 CoroutineWorker 类来说是一个非常重要的更新，因为您可以通过 TestListenableWorkerBuilder 直接运行 Worker 类，来测试它们的逻辑是否正确。

```
@RunWith(JUnit4::class)
class MyWorkTest {
    private lateinit var context: Context
    @Before
    fun setup() {
        context = ApplicationProvider.getApplicationContext()
    }
    @Test
    fun testMyWork() {
        // 获取 ListenableWorker 的实例
        val worker =
            TestListenableWorkerBuilder<MyWork>(context).build()
        // 同步的运行 worker
        val result = worker.startWork().get()
        assertThat(result, `is`(Result.success()))
```

```
    }
}
```

这里的重点是可以同步获取 CoroutineWorker 的运行结果，然后可以直接检查 Worker 类的逻辑行为是否正确。

使用 TestListenableWorkerBuilder 也可以将输入数据传递给 Worker 或设置 runAttemptCount，这对于测试 Worker 内部的重试逻辑是非常有用的。

比如，如果要将一些数据上传到服务器，考虑到连接可能出现问题，您也许会添加一些重试逻辑:

```
class MyWork(context: Context, params: WorkerParameters) :
        CoroutineWorker(context, params) {
    override suspend fun doWork(): Result {
        val serverUrl = inputData.getString("SERVER_URL")
        return try {
            // 通过 URL 做点什么
            Result.success()
        } catch (error: TitleRefreshError) {
            if (runAttemptCount <3) {
                Result.retry()
            } else {
                Result.failure()
            }
        }
    }
}
```

然后您可以在测试中，使用 TestListenableWorkerBuilder 来测试这个重试逻辑是否正确:

```
@Test
fun testMyWorkRetry() {
    val data = workDataOf("SERVER_URL" to
        "[http://fake.url](http://fake.url)")
    // 获取 ListenableWorker，并将 RunAttemptCount 设置为 2
    val worker = TestListenableWorkerBuilder<MyWork>(context)
                    .setInputData(data)
                    .setRunAttemptCount(2)
                    .build()
    // 启动同步执行的任务
    val result = worker.startWork().get()
    assertThat(result, `is`(Result.retry()))
}
@Test
fun testMyWorkFailure() {
    val data = workDataOf("SERVER_URL" to
```

```
            "[http://fake.url](http://fake.url)")
        // 获取 ListenableWorker，并将 RunAttemptCount 设置为 3
        val worker = TestListenableWorkerBuilder<MyWork>(context)
                        .setInputData(data)
                        .setRunAttemptCount(3)
                        .build()
        // 启动同步执行的任务
        val result = worker.startWork().get()
        assertThat(result, `is`(Result.failure()))
    }
```

总结

随着 WorkManager v2.1 以及 workManager-testing 中新特性的发布，CoroutineWorker 因其简单易用而大放光彩。现在您可以非常容易的对 Worker 类进行测试，并且 WorkManager 在 Kotlin 中的整体使用体验也非常棒。

如果您还没有在项目中使用 CoroutineWorker 以及 workmanager-runtime-ktx 中包含的其他扩展，强烈建议您在项目中使用它们。当使用 Kotlin 进行开发 (已经成为我的日常) 时，这是我使用 WorkManager 的首选方式。

希望这篇文章对您有所帮助，欢迎您在评论区积极留言，分享您在 WorkManager 使用中的见解或者问题。

WorkManager 相关资源

- **开发者指南 | 在 WorkManager 中进行线程处理**
- **参考指南 | androidx.work**
- **Codelab | 使用 WorkManager 处理后台任务**
- **WorkManager 的公开问题追踪器**
- **发行日志 | WorkManager**
- **Stack Overflow 的 [android-workmanager] 标签**
- **WorkManager 的源码 (AOSP的一部分)**

发布于 09-27

Android　Kotlin　Android 开发