




WorkManager: 周期性任务



谷歌开发者 
已认证的官方帐号

已关注

11 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。

如果您一直关注本系列文章，则会发现我们已经讨论过：

- [Android Jetpack WorkManager | Android 中文教学视频](#)
- [WorkManager 在 Kotlin 中的实践](#)

本文将介绍：

- 定义周期性任务
- 取消任务
- 自定义 WorkManager 配置

重复执行的任务

之前的文章中，我们已经介绍过使用 **OneTimeWorkRequest** 来规划任务。但如果您希望任务可以周期性地重复执行，则可以使用 **PeriodicWorkRequest**。

让我们先看看这两种 WorkRequest 之间的区别：

- 最小周期时长为 15 分钟 (与 **JobScheduler** 相同)
- Worker 类不能在 PeriodicWorkRequest 中链式执行

- 在 v2.1-alpha02 之前，无法在创建 `PeriodicWorkRequest` 时设置初始延迟

在与他人的讨论中，我遇到的一些常见问题与周期性任务有关。在本文中，我将会介绍周期性任务的基础知识以及常见用例和错误。另外，我也会介绍几种为 `Worker` 类编写测试的方式。

API

对比以前介绍过的创建一次性任务方法，创建 `PeriodicWorkRequest` 的调用没有很大的不同，只是多出了一个额外的参数用来指定最小重复间隔 (minimum repeat interval)：

```
val work = PeriodicWorkRequestBuilder<MyWorker>(1, TimeUnit.HOURS).build()
```

这个参数被称为“最小间隔”，是因为 Android 的电池优化策略和一些您添加的约束条件会延长两次重复之间的时间间隔。举个例子，如果您指定某个任务只会在设备充电时运行，那么如果设备没在充电，即使过了最小间隔，这个任务也不会执行——直到设备开始充电为止。



`PeriodicWorkRequest` 配合充电状态约束

在这种情景下，我们需要为 `PeriodicWorkRequest` 添加一个充电状态约束 (charging constraint)，并将其加入队列：

```
val constraints = Constraints.Builder(  
    .setRequiresCharging(true)  
    .build()  
  
val work = PeriodicWorkRequestBuilder<MyWorker>(1, TimeUnit.HOURS)  
    .setConstraints(constraints)  
    .build()  
  
val workManager = WorkManager.getInstance(context)  
workManager.enqueuePeriodicWork(work)
```

关于如何获取 `WorkManager` 实例的说明：

WorkManager v2.1 已经弃用了 `WorkManager#getInstance()`，转而使用 `WorkManager#getInstance(context: Context)`。新的方法工作方式与原来相同，不同点是它支持新的 按需初始化 (on-demand initialization) 功能。接下来的内容中，我都会使用需要传入 `context` 的新语法来获取 `WorkManager` 实例。

一个关于“最小间隔”的小提醒：由于 WorkManager 需要平衡两个不同的需求：应用的 WorkRequest 和 Android 系统限制电池消耗的需求，所以即使您为 WorkRequest 设置的所有约束条件都被满足，您的 Work 在增加了一些额外延迟之后仍可以被执行。

Android 包含了一组电池优化的策略：当用户没有使用设备时，系统会尽量减少活动以节省电量。这些策略会对任务的执行造成影响：在您的设备进入 **低电耗模式 (Doze mode)** 时，任务的执行可能会被推迟到下个 **维护窗口 (maintenance window)**。

间隔和弹性间隔 (FlexInterval)

如前文所述，WorkManager 不能保证任务在精确的某个时间去执行，但如果这是您的需求，那您可能需要寻找其他的 API。由于重复间隔实际上是最小间隔，所以 WorkManager 还提供了一个附加参数，您可以使用该参数来指定一个窗口，从而让 Android 可以在窗口中执行您的任务。

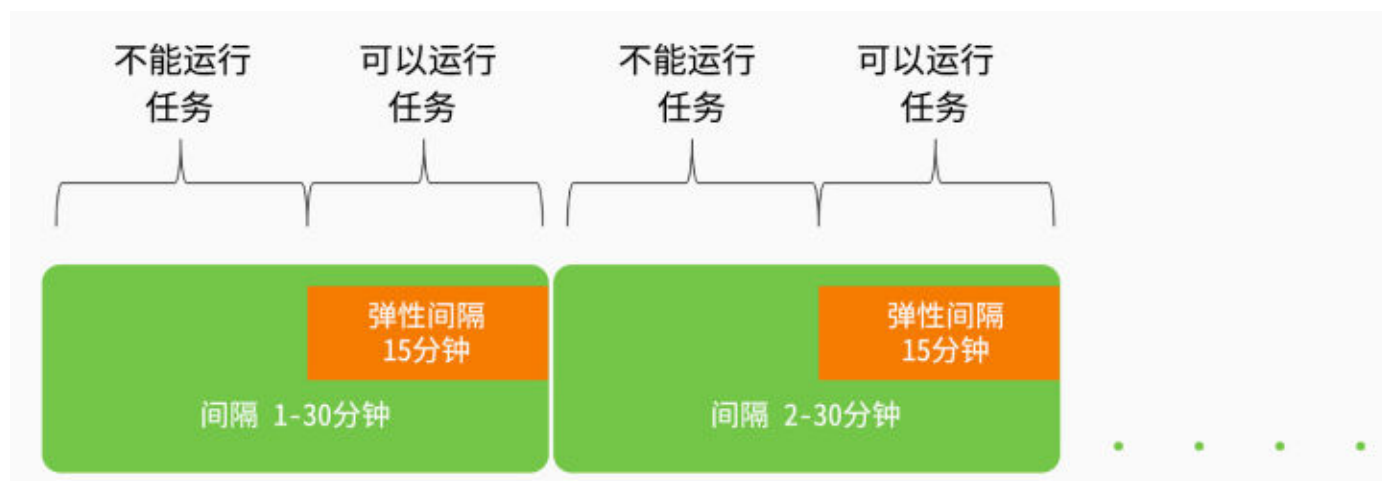
简而言之，您可以指定第二个间隔，从而控制在重复周期内可以运行您的周期性任务的区间。而这第二个间隔 (弹性间隔) 的位置则始终在它所在间隔的末尾。

让我们观察这样一个示例：假设您想要创建一个周期性任务，其重复周期为 30 分钟，您可以指定一个比重复周期小的弹性间隔，这里设为 15 分钟。

基于以上参数，构建 PeriodicWorkRequest 的实际代码为：

```
val logBuilder = PeriodicWorkRequestBuilder<MyWorker>(
    30, TimeUnit.MINUTES,
    15, TimeUnit.MINUTES)
```

结果是，我们的 Worker 会在周期的后半部分执行 (弹性间隔的位置总是在重复周期的末尾)：



间隔为 30 分钟、弹性间隔为 15 分钟的 PeriodicWorkRequest

别忘了，这些时间点始终基于 WorkRequest 中所包含的约束和设备所处的状态。

关于此功能，如果您想要了解更多，可以阅读 [PeriodicWorkRequest.Builder 文档](#)。

每日任务

由于周期性间隔是不精确的，您无法创建在每天指定时间执行的 `PeriodicWorkRequest`，即使我们放宽精度限制也不行。

您可以指定 24 小时为一个周期，但是由于任务的执行与 Android 的电池优化策略有关，您的期望值只能是 Worker 会在指定时间段附近被执行。因此其结果可能是：您的任务会在第一天的 5:00AM、第二天的 5:25AM、第三天的 5:15AM，以及第四天的 5:30AM 被执行，以此类推。随着时间的流逝，误差会被不断累积。

目前，如果您需要在每天的大致同一时间执行某一个 Worker，那么最好的选择是使用 `OneTimeWorkRequest` 并设置初始延迟，这样您便可以在正确的时间执行任务：

```
val currentDate = Calendar.getInstance()
val dueDate = Calendar.getInstance()

// 设置在大约 05:00:00 AM 执行
dueDate.set(Calendar.HOUR_OF_DAY, 5)
dueDate.set(Calendar.MINUTE, 0)
dueDate.set(Calendar.SECOND, 0)

if (dueDate.before(currentDate)) {
    dueDate.add(Calendar.HOUR_OF_DAY, 24)
}

val timeDiff = dueDate.timeInMillis - currentDate.timeInMillis
val dailyWorkRequest = OneTimeWorkRequestBuilder<DailyWorker>
    .setConstraints(constraints) .setInitialDelay(timeDiff,
        TimeUnit.MILLISECONDS)
    .addTag(TAG_OUTPUT) .build()

WorkManager.getInstance(context).enqueue(dailyWorkRequest)
```

这样一来便能完成第一次执行。接下来我们需要将下一个任务在当前任务成功执行完成时加入队列：

```
class DailyWorker(ctx: Context,
    params: WorkerParameters) : Worker(ctx, params) {

    override fun doWork(): Result {
        val currentDate = Calendar.getInstance()
        val dueDate = Calendar.getInstance()

        // 设置在大约 05:00:00 AM 执行
        dueDate.set(Calendar.HOUR_OF_DAY, 5)
        dueDate.set(Calendar.MINUTE, 0)
```

```
dueDate.set(Calendar.SECOND, 0)
```

```
if (dueDate.before(currentDate)) {  
    dueDate.add(Calendar.HOUR_OF_DAY, 24)  
}
```

```
val timeDiff = dueDate.timeInMillis - currentDate.timeInMillis  
val dailyWorkRequest = OneTimeWorkRequestBuilder<DailyWorker>()  
    .setInitialDelay(timeDiff, TimeUnit.MILLISECONDS)  
    .addTag(TAG_OUTPUT)  
    .build()
```

```
WorkManager.getInstance(applicationContext)  
    .enqueue(dailyWorkRequest)
```

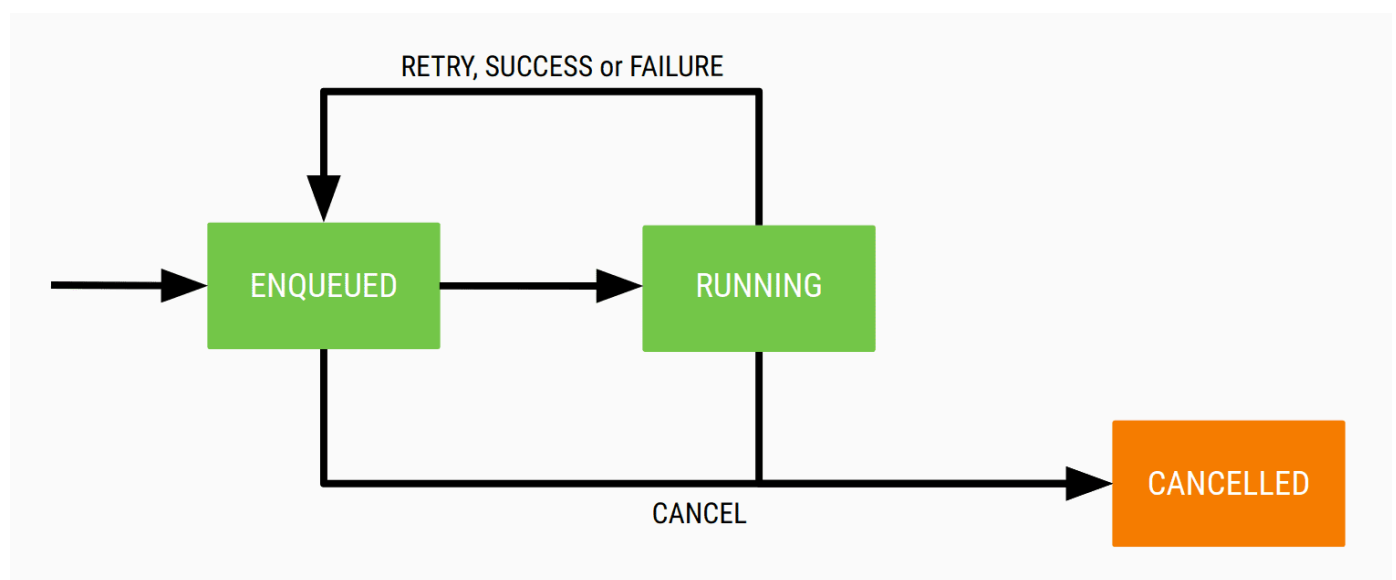
```
return Result.success()  
}
```

```
}
```

请记住，执行 Worker 的实际时间取决于您在 WorkRequest 中使用的约束和 Android 平台的优化操作。

周期性任务的状态

前文已经讲过，周期性任务与一次性任务的其中一个区别便是不能通过 `PeriodicWorkRequest` 建立任务链。之所以存在这一约束，是因为在一个任务链中，您会在一个 Worker 的状态转变为 `SUCCEEDED` 时过渡到任务链中的下一个 Worker，而 `PeriodicWorkRequest` 没有 `SUCCEEDED` 状态。



PeriodicWorkRequest 的状态

周期性任务不会以 SUCCEEDED 状态结束，它会持续运行直到被取消。当您在周期性任务的 Worker 中调用 Result#success() 或 Result#failure() 时，周期性任务会回到 ENQUEUED 状态并等待下一次执行。

基于这一原因，您无法在使用周期性任务时建立任务链，使用 UniqueWorkRequest 也同样不行。这样一来，PeriodicWorkRequest 也失去了追加任务的能力：您只能使用 KEEP 和 REPLACE，而不能使用 APPEND。

数据的输入和输出

WorkManager 允许您传递一个 Data 对象给您的 Worker，同时在 success 和 failure 方法被调用时，也会返回一个新的 Data 对象给您（由于在您返回 Result#retry() 时 Worker 的执行是无状态的，所以此时没有数据输出选项）。

在一次性 Worker 组成的链中，一个 Worker 的返回值会成为链条中下个 Worker 的输入值。我们已经知道，周期性任务无法使用任务链条，因为其并不会以“成功”的状态结束——它只会被取消操作所结束。

所以，我们要在哪里看到和使用 Result#success(outData) 方法所返回的数据？

我们可以通过 PeriodicWorkRequest 的 WorkInfo 来观察这些 Data。仅在周期任务下一次被执行前，我们可以依靠判断 Worker 是否处于 ENQUEUED 状态来检查它的输出：

```
val myPeriodicWorkRequest =
    PeriodicWorkRequestBuilder<MyPeriodicWorker>(1, TimeUnit.HOURS)
        .build()

WorkManager.getInstance(context).enqueue(myPeriodicWorkRequest)

WorkManager.getInstance()
    .getWorkInfoByIdLiveData(myPeriodicWorkRequest.id)
    .observe(lifecycleOwner, Observer { workInfo ->
        if ((workInfo != null) &&
            (workInfo.state == WorkInfo.State.ENQUEUED)) {
            val myOutputData = workInfo.outputData.getString(KEY_MY_DATA)
        }
    })
```

如果您需要周期性 Worker 能够提供一些结果数据，上述方法可能不是您的最佳选项。一个更好的选择是将数据通过另一个媒介进行传输，比如数据库表。

更多有关获取任务状态的信息，请参考本系列的 [《Android Jetpack WorkManager | Android 中文教学视频》](#) 和 WorkManager 的文档：[任务状态和观察任务](#)。

独特任务

某些 WorkManager 用例可能会陷入一种模式：当应用启动时，会在第一时间将一些任务加入队列。这些任务可能是您想要周期执行的后台同步任务，也可能是预定内容的下载。不论是什么，常见的模式都是需要在应用启动的第一时间将这些任务入队。

我已经看到这种模式几次，在 Application#onCreate 方法中，开发者创建了 WorkRequest 并将其入队。看起来一切正常，直到您发现有些任务重复执行了很多次。这种情况在只要不进行取消操作便不会到达最终状态的周期性任务身上尤其容易出现。

我们常说，即使您的应用被关闭或者设备被重启，WorkManager 仍会保证执行您的任务。所以，在应用每次启动时都尝试将您的 Worker 加入队列，会导致每次启动都添加一个新的 WorkRequest。如果您使用的是 OneTimeWorkRequest，问题可能不大，因为一旦任务执行完成，WorkRequest 也会结束。但对于周期性任务来说，“结束”是一个完全不同的概念，结果是您可能会轻易地将多个周期性任务重复加入队列。

针对这种情况的解决方案是，使用 WorkManager#enqueueUniquePeriodicWork() 将您的 WorkRequest 作为独特任务 (unique Work) 加入队列：

```
class MyApplication: Application() {

    override fun onCreate() {
        super.onCreate()
        val myWork = PeriodicWorkRequestBuilder<MyWorker>(
            1, TimeUnit.HOURS)
            .build()

        WorkManager.getInstance(this).enqueueUniquePeriodicWork(
            "MyUniqueWorkName",
            ExistingPeriodicWorkPolicy.KEEP,
            myWork)
    }
}
```

这样就可以帮您避免任务被重复多次加入队列。

使用 KEEP 或 REPLACE?

选择哪种策略取决于您在 Worker 中执行什么样的操作。个人而言，我通常会使用 KEEP 策略，因为它更轻量，不必替换现有的 WorkRequest，同时，这一策略也可以避免取消已经在运行的 Worker。

我只会在有恰当理由时才会使用 REPLACE 策略，比如，当我想要在某个 Worker 的 doWork() 方法中对它自己重新排期时。

如果您选择使用 REPLACE 策略，您的 Worker 应当适当地处理停止状态，因为这种策略下，如果一个新的 WorkRequest 在 Worker 正在运行时加入队列，WorkManager 就可能不得不取消正在运行的实例。不过您也应该在任何情况下都处理好停止状态，因为 Worker 正在被执行时，如果某个约束条件不再被满足，WorkManager 也可能会停止您的任务。

有关独特任务的更多信息，请参阅文档：[唯一工作](#)。

测试周期性任务

[WorkManager 的测试文档](#) 十分详尽，覆盖了基本的测试方案。在 WorkManager v2.1 发布后，您有两种方式测试您的 Worker：

- [WorkManagerTestInitHelper](#)
- [TestWorkerBuilder](#) 和 [TestListenableWorkerBuilder](#)

使用 WorkManagerTestInitHelper，您可以在测试您的 Worker 类时模拟延迟、约束条件和周期要求被满足等情况。这种测试方法的优势在于，它可以处理 Worker 入队自己或另一个 Worker 类的情况，正如前面示例——实现了每天大约在同一时间运行的“DailyWorker”——中所看到的。了解更多信息，请查阅：[WorkManager 的测试文档](#)。

如果您需要测试 [CoroutineWorker](#)、[RxWorker](#) 和 [ListenableWorker](#)，使用 WorkManagerTestInitHelper 会带来一些额外的复杂性，因为这时您无法依赖它的 SynchronousExecutor。

为了更加直接地测试这几个类，WorkManager v2.1 加入了一组新的 WorkRequest 构造器：

- TestWorkerBuilder 用于直接调用 Worker 类
- TestListenableWorkerBuilder 用于直接调用 ListenableWorker、RxWorker 或 CoroutineWorker

这些新构造器的优点是，您可以使用它们测试任何种类的 Worker 类，因为在使用它们时，您可以直接运行对应的 Worker。

您可以通过阅读 [使用 WorkManager 2.1.0 进行测试](#) 这篇文档来了解更多，也可以查看 [Sunflower 示例应用](#) 中使用这些新的构造器进行测试的示例：

```
import android.content.Context
import androidx.test.core.app.ApplicationProvider
import androidx.work.ListenableWorker.Result
import androidx.work.WorkManager
import androidx.work.testing.TestListenableWorkerBuilder
import com.google.samples.apps.sunflower.workers.SeedDatabaseWorker
import org.hamcrest.CoreMatchers.`is`
import org.junit.Assert.assertThat
import org.junit.Before
import org.junit.Test
```



```

import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class RefreshMainDataWorkTest {
    private lateinit var context: Context

    @Before
    fun setup() {
        context = ApplicationProvider.getApplicationContext()
    }

    @Test
    fun testRefreshMainDataWork() {
        // 获取 ListenableWorker
        val worker = TestListenableWorkerBuilder<SeedDatabaseWorker>(context)
            .build()

        // 同步执行该任务
        val result = worker.startWork().get()
        assertThat(result, `is`(Result.success()))
    }
}

```

总结

希望本文对您有所帮助，我也很愿意倾听您使用 WorkManager 的方式。如果您对解读 WorkManager 的功能以及撰写相关文章有更好的想法，欢迎您在 Twitter 上联系我 @pfmaggi。

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](#)
- [发行日志 | WorkManager](#)
- [Codelab | 使用 WorkManager 处理后台任务](#)
- [WorkManager 的源码 \(AOSP的一部分\)](#)
- [演讲 | 使用 WorkManager \(2018 Android 开发者峰会\)](#)
- [Issue Tracker](#)
- [Stack Overflow 的 \[android-workmanager\] 标签](#)
- [Android 开发者博客上关于 Power 的文章系列](#)

发布于 10-12

