




自定义 WorkManager —— 基础概念



谷歌开发者 
已认证的官方帐号

已关注

5 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。

目前为止本系列已经讨论过：

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

在本篇文章中，我们将会讨论自定义配置相关的内容，包括：

- 为什么可能会需要自定义配置
- 如何声明自定义配置
- WorkerFactory 以及自定义 WorkerFactory 的原因
- DelegatingWorkerFactory 详解

本系列的下一篇文章将对依赖注入和 Dagger 展开讨论，请持续关注我们。

使用 WorkManager 时，您需要自己定义 **Worker/CoroutineWorker** 或任何 **ListenableWorker** 的派生类。WorkManager 会在正确的时间点实例化您的 Worker，其时机独立于您应用的运行，不受其运行状态的影响。为了可以初始化您的 Worker，WorkManager 会使用一个 **WorkerFactory**。

默认 WorkerFactory 所创建的 Worker 只包含两个参数：

- Application 的 Context
- **WorkerParameters**

如果您需要通过 Worker 的构造函数传入更多参数，则需要一个自定义的 WorkerFactory。

延伸阅读：我们讲过默认的 WorkerFactory 使用反射来实例化正确的 ListenableWorker 类，但当我们的 Worker 类的类名被 R8 (或 ProGuard) 最小化之后，这个操作就会失败。为了避免这种情况，WorkManager 包含了一个 [proguard-rules.pro](#) 文件来避免您的 Worker 类的类名被混淆。

自定义配置和 WorkerFactory

WorkManager 类遵循 **单例模式**，而且它只能在实例化之前进行配置。这意味着，如果您想自定义它的配置，就必须先禁用默认配置。

如果您尝试通过 `initialize()` 方法再次初始化 WorkManager，该方法就会抛出一个异常 (于 1.0.0 版本中加入)。为了避免异常，您需要禁用默认的初始化。您可以稍后在您的 Application 的 `onCreate` 方法中配置和初始化您的 WorkManager。

2.1.0 版本 中加入了一个更好的初始化 WorkManager 的方式。您可以通过在您的 Application 类中实现 WorkManager 的 **[Configuration.Provider](#)** 接口的方式来使用**按需初始化**。接下来，您只需要使用 **[getInstance\(context\)](#)** 获得实例，WorkManager 就会通过您的配置初始化它自己。

可配置参数

如上所讲，您可以配置用来创建 Worker 的 WorkerFactory，但是您也可以自定义其他的参数。WorkManager 的 **[Configuration.Builder](#)** 参考指南中包含了参数的完整列表。这里我想强调两个附加参数：

- Logging 级别
- JobId 范围

当我们需要时，可以通过修改日志级别方便地理解 WorkManager 中正在发生什么。关于这个话题，我们有一个 **[专门的文档页](#)**。您也可以查看 **[Advanced WorkManager codelab 实战教程](#)**，以了解此功能在真实示例中的实现，以及您可以通过此功能获取到什么样的信息。

如果以在我们的应用中使用 JobScheduler API 一样的方式使用 WorkManager，我们可能也会想要自定义 JobId 范围。因为在这种情况下，您会想要避免在同一个地方使用相同的 JobId 范围。版本 2.4.0 中也加入了一个新的 **[Lint 规则](#)** 来覆盖这种情况。

WorkManager 的 WorkerFactory

我们已经知道，WorkManager 有一个默认的 WorkerFactory，它可以根据我们经由 WorkRequest 传入的 Worker 类的类名，通过反射来找到应该实例化的 Worker 类。

△ 如果您在创建了一个 WorkRequest 后重构了应用，并为您的 Worker 类起了另一个名字，WorkManager 就会因为无法找到正确的类而抛出一个 `ClassNotFoundException`。

您可能会想要为您的 Worker 的构造函数添加其他参数。假设您有一个 Worker 需要引用一个 Retrofit 服务来跟远程服务器进行通讯:

```
/* Copyright 2020 Google LLC.
   SPDX-License-Identifier: Apache-2.0 */
class UpvoteStoryWorker(
    appContext: Context,
    workerParams: WorkerParameters,
    private val service: UpvoteStoryHttpApi)
    : CoroutineWorker(appContext, workerParams) {

    override suspend fun doWork(): Result {

        return try {
            // 投票操作
            Result.success()
        } catch (e: Exception) {
            if (runAttemptCount < MAX_NUMBER_OF_RETRY) {
                Result.retry()
            } else {
                Result.failure()
            }
        }
    }
}
```

如果我们在一个应用上作出上面的修改，程序仍然可被正常编译。但是只要代码被执行、WorkManager 尝试去实例化这个 CoroutineWorker 时，应用就会因为抛出异常而被关闭。异常的描述为无法找到正确的方法来进行实例化:

```
Caused by java.lang.NoSuchMethodException:
<init> [class android.content.Context, class androidx.work.WorkerParameters]
```

这时我们就需要一个自定义的 WorkerFactory。

但是别着急，我们已经看到其中涉及的几个步骤。现在让我们回顾一下我们已经做了的事情，然后深入了解其中每一步的详细信息:

1. 禁用默认初始化
2. 实现一个自定义 WorkerFactory
3. 创建自定义配置
4. 初始化 WorkManager

禁用默认初始化

如 [WorkManager 的文档](#) 中描述，禁用操作要在您的 AndroidManifest.xml 文件中完成。移除默认情况下从 WorkManager 库中自动合并的节点。

```
<!-- Copyright 2020 Google LLC.
      SPDX-License-Identifier: Apache-2.0 -->
<application
...
    <provider
        android:name="androidx.work.impl.WorkManagerInitializer"
        android:authorities="${applicationId}.workmanager-init"
        tools:node="remove" />
</application>
```

实现一个自定义 WorkerFactory

为了创建包含正确参数的 Worker，现在需要实现我们自己的工厂 (factory):

```
/* Copyright 2020 Google LLC.
   SPDX-License-Identifier: Apache-2.0 */
class MyWorkerFactory(
    private val service: UpvoteStoryHttpApi) : WorkerFactory() {

    override fun createWorker(
        appContext: Context,
        workerClassName: String,
        workerParameters: WorkerParameters
    ): ListenableWorker? {
        // 这里只能处理一个 Worker，请不要这样做!
        // 参考下文来更好地使用 DelegatingWorkerFactory
        return UpvoteStoryWorker(appContext, workerParameters,
            DesignerNewsService)
    }
}
```

创建一个自定义 WorkerConfiguration**

接下来，我们必须将我们的工厂注册到我们的 WorkManager 的自定义配置中:

```
/* Copyright 2020 Google LLC.
   SPDX-License-Identifier: Apache-2.0 */
class MyApplication : Application(), Configuration.Provider {
    override fun getWorkManagerConfiguration(): Configuration =
        Configuration.Builder()
            .setMinimumLoggingLevel(android.util.Log.DEBUG)
```

```

        .setWorkerFactory(MyWorkerFactory(DesignerNewsService))
        .build()

    ...
}

```

初始化 WorkManager

当您的应用中只有一个 Worker 类时，以上便是您所需要做的所有事情。而如果您有多个或者预计未来会有多个 Worker 类，更好的解决方案是使用在 2.1 版中加入的 [DelegatingWorkerFactory](#)。

使用 DelegatingWorkerFactory

我们可以通过使用 DelegatingWorkerFactory 来替代将 WorkManager 配置为直接使用某个工厂的操作。我们可以使用 DelegatingWorkerFactory 的 [addFactory\(\)](#) 方法向其添加我们的工厂，这样一来，您就有了多个工厂，其中每个都可以管理一个或多个 Worker。在 DelegatingWorkerFactory 中注册您的工厂，这将有助于协调多个工厂的执行。

在这种情况下，您的工厂需要检查是否知道如何处理作为参数传入的 workerClassName。如果答案是否定的，就返回 null，而 DelegatingWorkerFactory 便会去寻找下一个注册的工厂。如果没有任何被注册的工厂知道如何处理某个类，那么它将回退到使用反射的默认工厂。

下面是我们的工厂类代码，修改为当它不知道如何处理某个 workerClassName 时，将返回 null:

```

/* Copyright 2020 Google LLC.
   SPDX-License-Identifier: Apache-2.0 */
class MyWorkerFactory(
    private val service: DesignerNewsService) : WorkerFactory() {

    override fun createWorker(
        appContext: Context,
        workerClassName: String,
        workerParameters: WorkerParameters
    ): ListenableWorker? {

        return when(workerClassName) {
            UpvoteStoryWorker::class.java.name ->
                ConferenceDataWorker(appContext, workerParameters, service)
            else ->
                // 返回 null，这样基类就可以代理到默认的 WorkerFactory
                null
        }
    }
}

```

我们的 WorkManager 配置会变成:

```
/* Copyright 2020 Google LLC.
   SPDX-License-Identifier: Apache-2.0 */
class MyApplication : Application(), Configuration.Provider {

    override fun getWorkManagerConfiguration(): Configuration {
        val myWorkerFactory = DelegatingWorkingFactory()
        myWorkerFactory.addFactory(MyWorkerFactory(service))
        // 在这里添加您应用中可能会使用的其他 WorkerFactory

        return Configuration.Builder()
            .setMinimumLoggingLevel(android.util.Log.INFO)
            .setWorkerFactory(myWorkerFactory)
            .build()
    }
    ...
}
```

如果您有多个 Worker 需要不同的参数，您可以创建第二个 WorkerFactory，并通过再次调用 addFactory 来添加它。

总结

WorkManager 是一个功能十分强大的库，它的默认配置已经可以覆盖许多常见的使用场景。然而当您遇到某些情况时，诸如需要增加日志级别或需要传入额外参数到您的 Worker 时，则需要一个自定义的配置。

希望您能通过本文对此主题有一个良好的认识。如果您有任何疑问，可以在评论区中留言。

接下来的文章我们将会讨论如何在自定义 WorkManager 配置时使用 Dagger，感兴趣的读者请继续关注。

发布于 10-22

[Android](#) [Android 开发](#)