1. 背景

当我们说后台工作时，这意味着应用程序没有运行，我们正试图通过消耗系统资源（如 RAM、电池、电源等）来做一些事情。在 Marshmallow 发布之前，我们主要使用服务或意图服务来完成这些事情.紧随其后的是 Nougat Doze 诞生了。

Doze 模式基本上是一种旨在通过在设备长时间处于空闲状态时延迟应用程序后台 CPU 和网络活动来延长电池寿命的状态。在这里，我们有一个维护窗口，在此期间应用程序可以完成待处理的工作。

之后，在 Oreo 中，后台运行的服务被删除。为了限制后台服务，出现了前台服务的概念。许多应用程序都在关注它。

多年来，我们一直在使用服务、意图服务、JobScheduler 等来运行我们的后台任务。因此，在经历了所有艰难时期之后，Android 团队提出了一个一站式解决方案来执行后台作业。是 WorkManager。

1. 简介

WorkManager 是 Android Jetpack 库和架构组件的一部分，主要用于执行长时间运行的后台任务。后台任务主要根据执行时间分为3种：

- 即时

- 在确切的时间点

- 在未来的任何时间点（延期任务）

我们可以使用前台服务或线程来执行即时请求，并使用 AlarmManager 来执行特定的时间点任务。

WorkManager 可以在大多数情况下使用，例如即时任务和延迟任务，因为它可以处理任务并确保执行完成，即使应用程序突然进入后台执行操作或设备重新启动也是如此。

我们可以将 WorkManager 定义为用于管理可延迟后台任务的库。延迟任务是可以在未来任何时间点运行而无需用户交互的任务。

WorkManager 提供了一种更好的方式来处理后台任务。

- 支持大多数提供向后兼容性的版本，直到 API 14

- 在有/没有 Google Play 服务的设备上运行。

- 无需担心线程，因为 WorkManager 将确保在后台线程上执行。

- 易于安排、取消、重试和查询工作。它给了我们完全的控制权。

- 具有并行执行支持的顺序。在执行时，后面任务的输入可以是前面任务的输出。

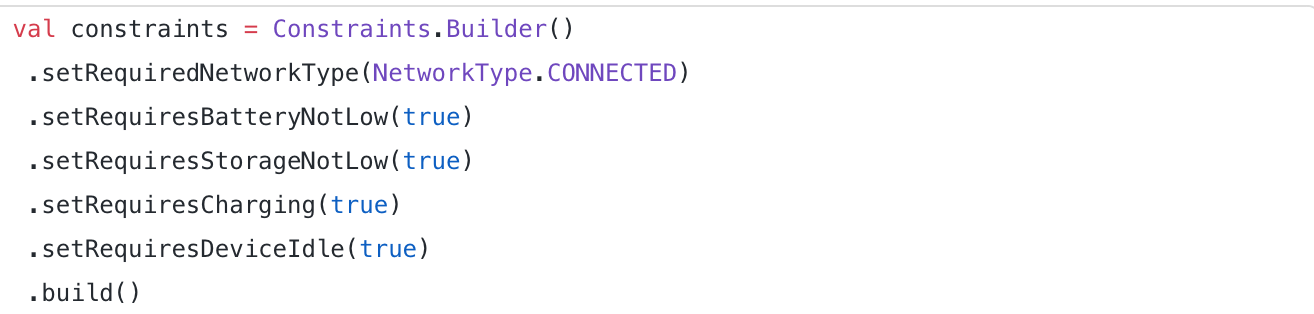
- 保证任务执行。

1. Constraints

可能有一些预定义的条件集，系统在开始工作之前应该满足这些条件，这些条件被称为约束。就像任务可能需要网络连接，如果它与网络有关，或者它可以通过检查它是否在低电量情况下运行繁重的任务等来提高电池效率。

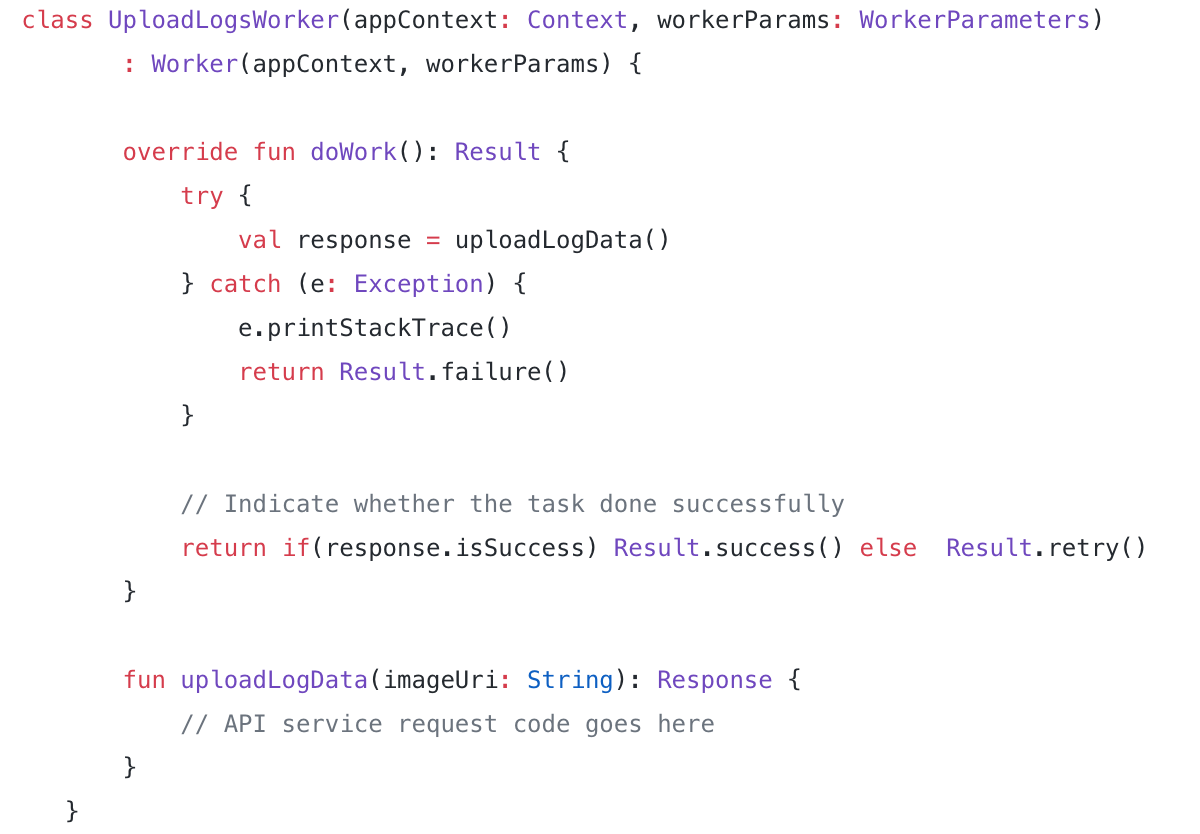


根据需求，它可以是单个或多个约束。



1. Worker

需要执行的任务或工作是使用 Worker 类定义的。它是一个抽象类，因此我们需要对其进行子类化并覆盖我们执行实际工作实现的方法 doWork()。 Worker 类负责在 WorkManager 提供的后台线程上同步执行工作。



Worker的 Result 指定任务是成功执行还是失败并需要重试。

- Result.success() ：返回一个 Result 的实例，可以用来表示工作已经成功完成

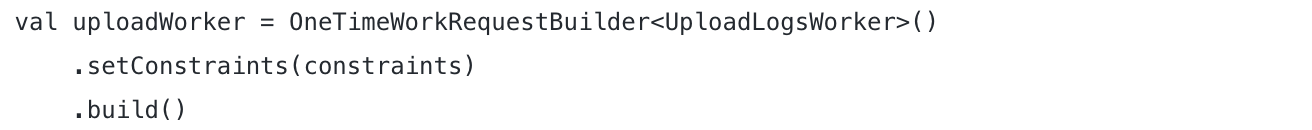
- Result.failure() ：返回 Result 的实例，可用于指示工作已完成但永久失败

- Result.retry() ：返回 Result 的实例，可用于指示工作遇到暂时性失败，应使用退避策略重试

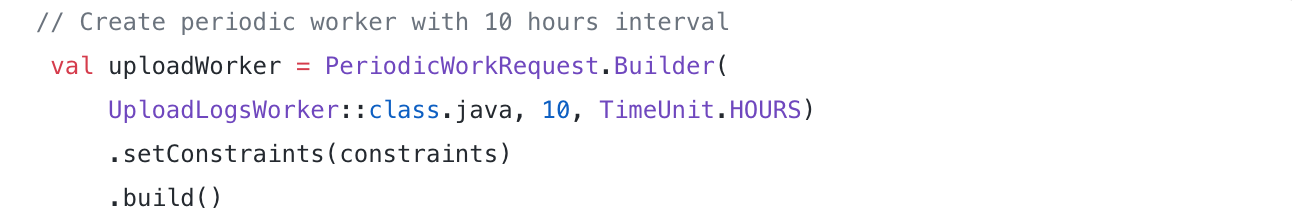
1. WorkRequest

它定义了工作的执行方式和时间。每个工作都必须有一个用于调度的工作请求。我们有两种类型的请求：

OneTimeWorkRequest：主要用于一次性执行任务。对于非重复任务

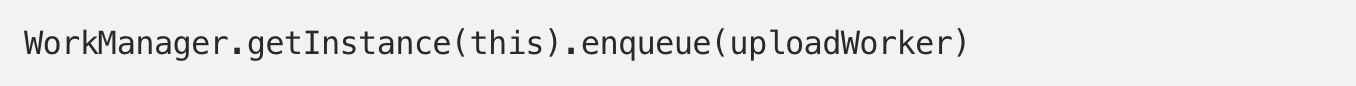


PeriodicWorkRequest：需要定期执行任务的地方。用于重复任务



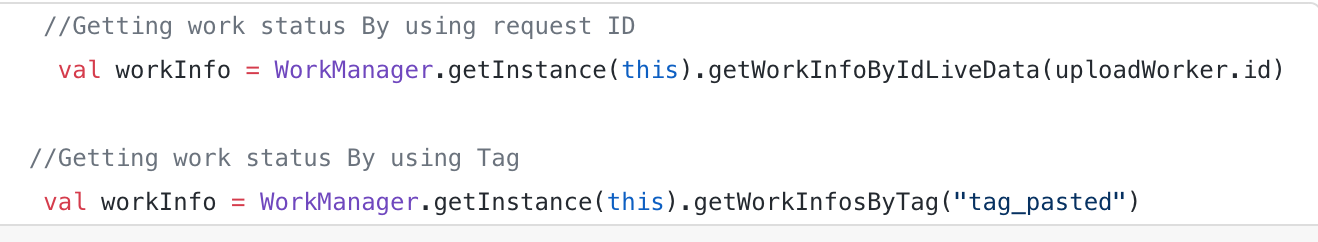
1. WorkManager

一旦定义了工作请求，我们就可以使用 WorkManager 来安排它。我们调用 enqueue() 来安排请求



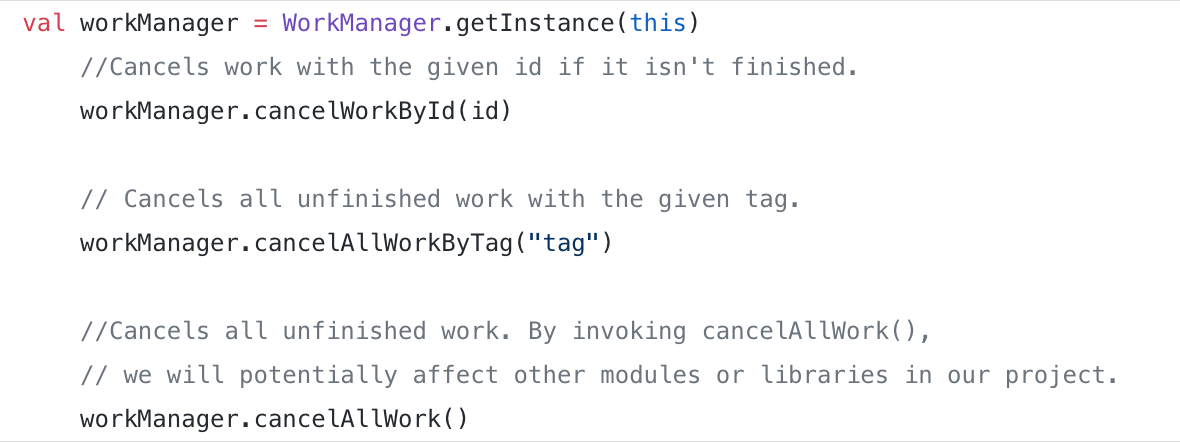
1. WorkStatus

一旦工作入队，我们唯一需要知道的是工作的状态和输出。对于所有计划的工作请求，WorkManager 维护一个 LiveData，我们可以通过使用标签或请求 ID 来获取它。 Tag 是我们在使用 WorkManger 将请求排入队列时提供的字符串



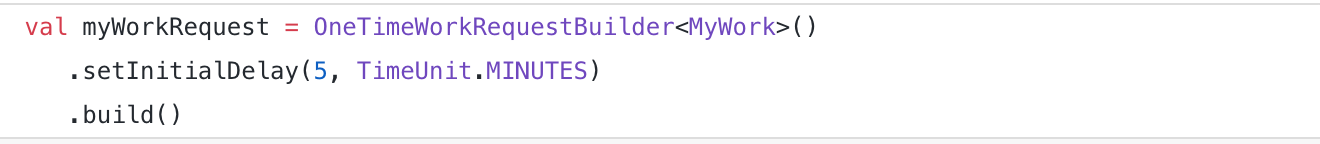
1. Canceling

如果我们想取消任何计划的工作，WorkManger 为我们提供了灵活性，可以通过使用工作请求的 id 或标签来执行此操作。



1. Delayed Work

可以使用 setInitialDelay() 延迟初始工作执行：

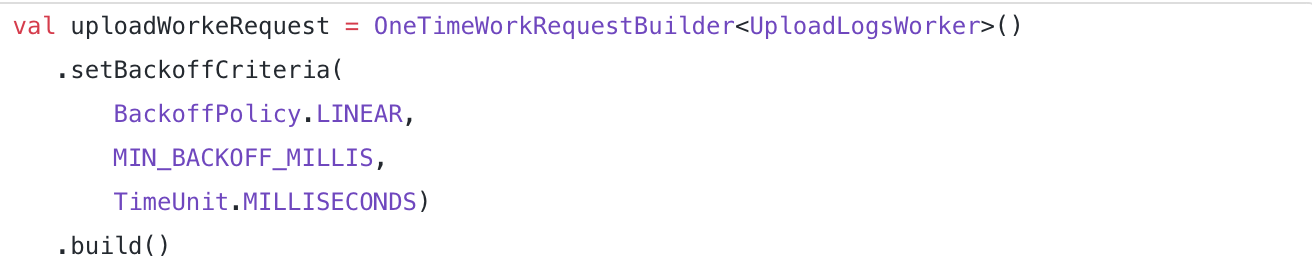


1. Retry and Backoff policy

当Worker返回 Result.retry() 时，WorkManger 将根据退避策略重新安排工作。退避策略由两个属性定义：

- BackoffDelay 指定在重试工作之前等待的最小时间。默认为 10 秒

- BackoffPolicy 定义了后退延迟应如何随着时间的推移而增加，以进行后续重试尝试。 WorkManager 支持 2 种退避策略，LINEAR 和 EXPONENTIAL。默认情况下，退避策略是指数的，但可以设置为线性。



1. Chaining

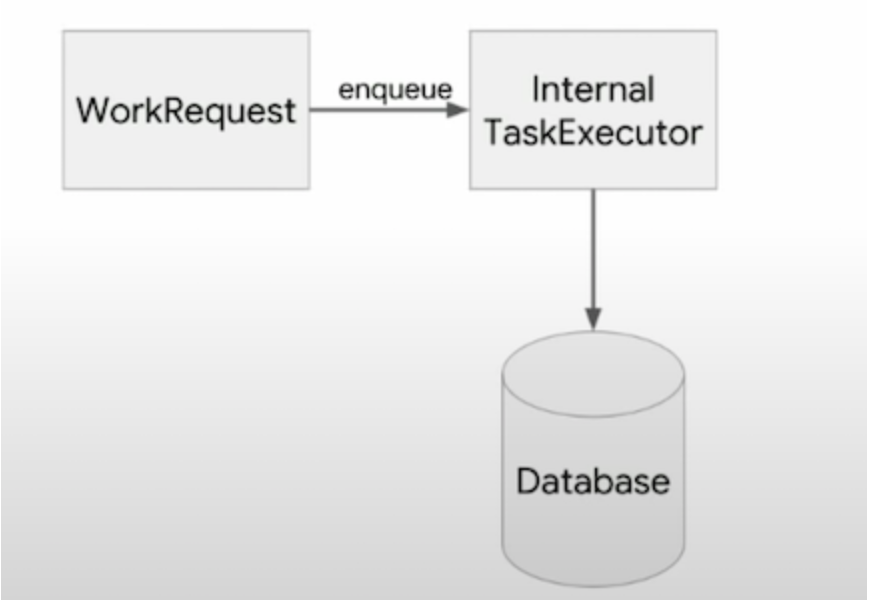
可以通过 WorkManager 轻松链接多个请求。假设我们在开始时要并行执行三个请求，并且一旦前三个请求成功完成后应该执行下一个请求，然后我们可以使用 begin 和 then 来指定任务。



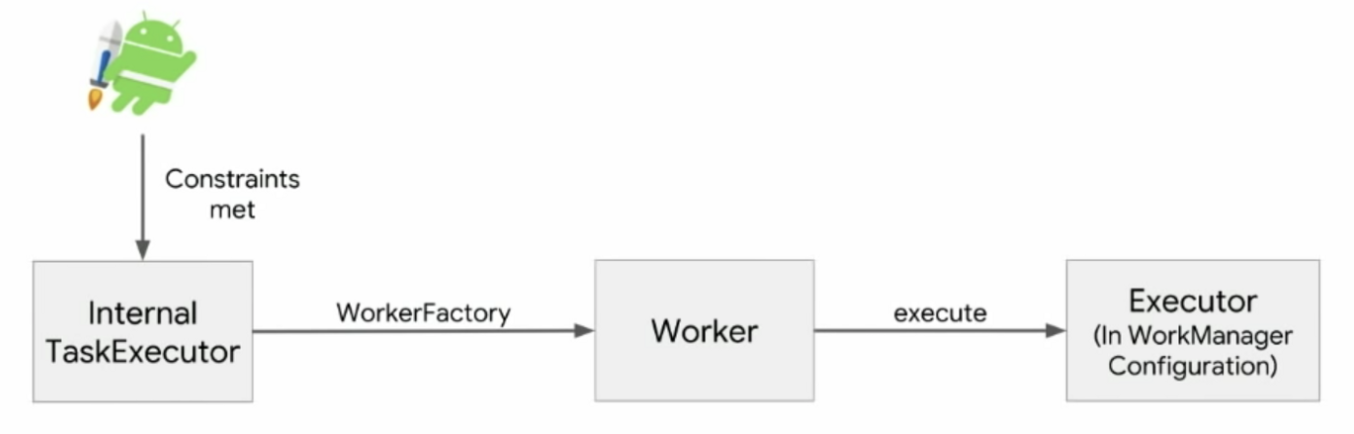
OneTimeWorkRequests，如果链中的任何工作失败或被取消，其所有相关工作都会继承该状态并且永远不会运行。

1. 原理

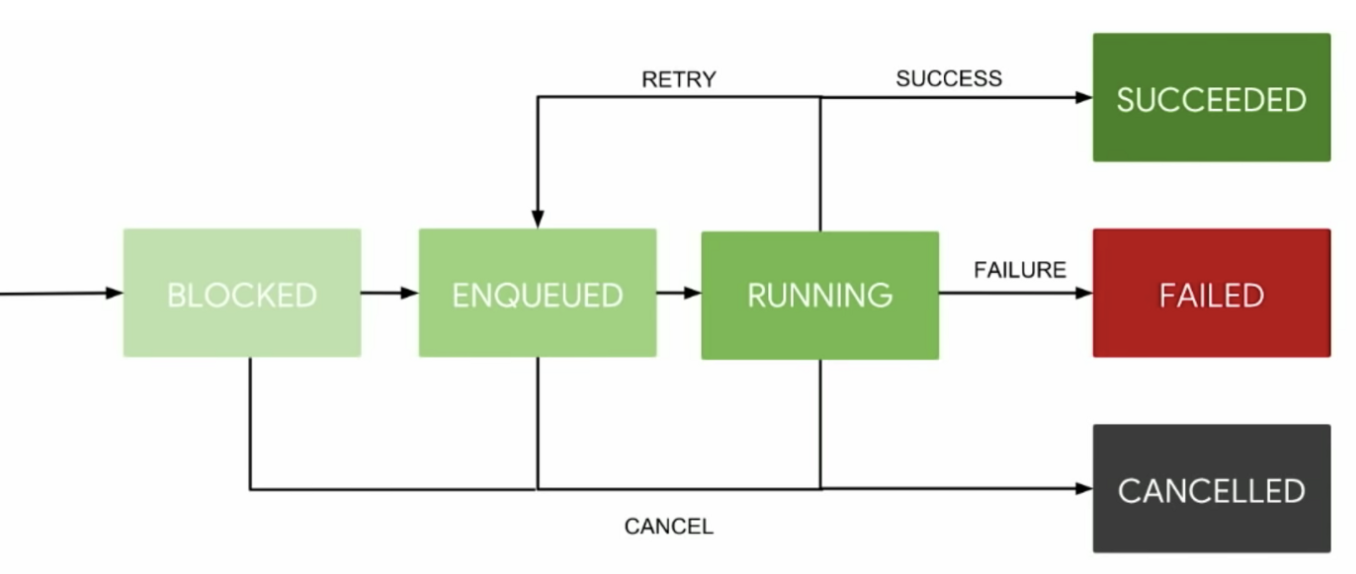
当我们使用 WorkManger 将工作排入队列时，内部 TaskExecutor 会将我们的 WorkRequest 信息保存到 WorkManager 数据库中。



在满足约束时保存请求后，内部 TaskExecutor 使用 WorkerFactory 创建一个 Worker 。我们可以定制worker factory。一旦创建了worker，它就会在执行器中执行

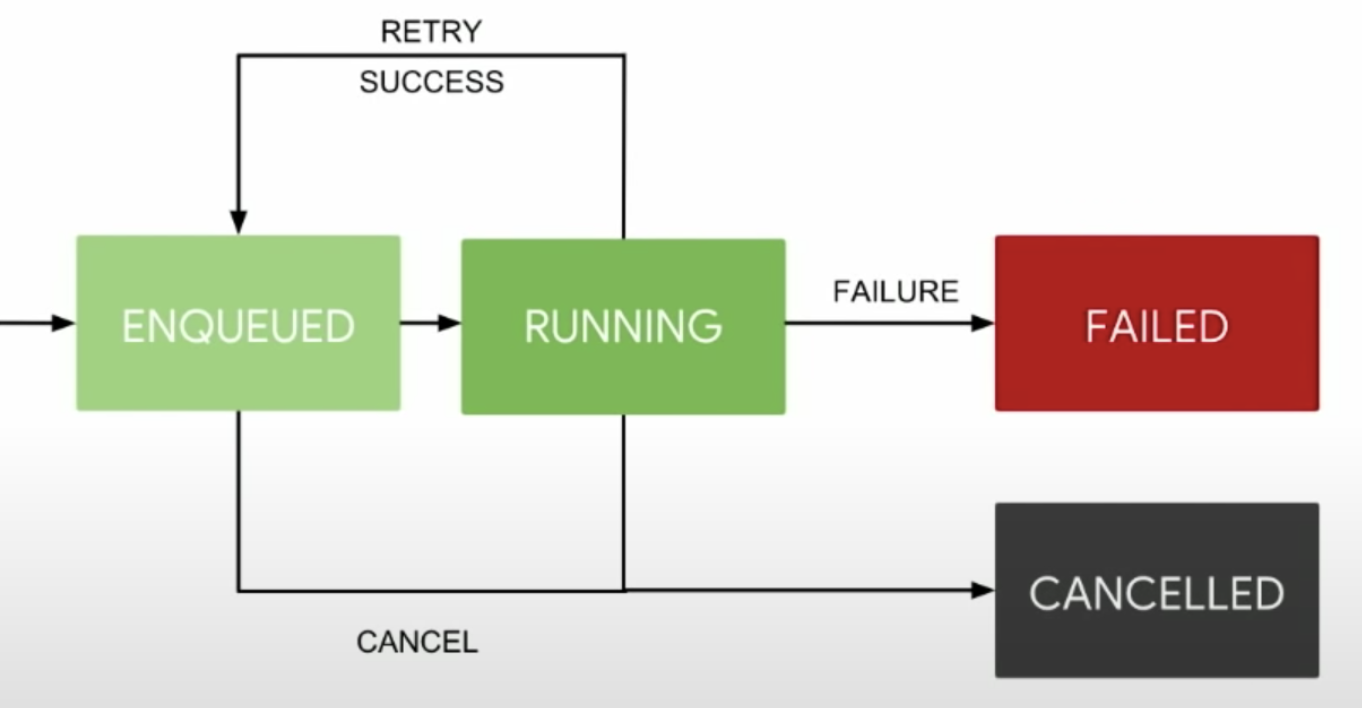


一次性请求的生命周期：



正如我们所见，根据 WorkManager 的状态，初始状态将被阻塞或大部分排队。一旦workers执行开始，它就会进入运行状态。从运行状态来看，它主要应该导致成功或失败。但在某些情况下，我们可能希望根据需求重试或取消正在进行的请求。

周期性请求的生命周期：



在大多数情况下，它类似于一次性请求，但唯一的变化是即使请求成功执行，它也会根据我们提供的时间间隔再次入队。