4.9. Архитектурный компонент WorkManager

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 4.9. Архитектурный компонент WorkManager

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 17:56

Оглавление

1. Управление фоновыми процессами

В текущем модуле представлены основные механизмы организации фоновых процессов Android. Это только малая часть всех возможных механизмов, предлагаемых разработчиками системы для выполнения действий за пределами пользовательского интерфейса. Основная проблема, которую следует при этом решать - это своевременное прерывание процесса с целью экономии заряда батареи. Всё бОльшие ограничения накладываются на классы организации фоновой работы, от версии к версии появляются новые практики, жестко привязанные к уровню используемых API. Конечно, же это не может не отразиться на процессе разработки приложений. Неудобства испытывают как разработчики поставляемых решений в части их поддержки и обновлений, так и разработчики приложений, вынужденные постоянно проверять установленную на устройстве систему и в зависимости от этого использовать тот или иной инструмент.

С конца 2018 года разработчики системы Android в составе библиотеки *Android Jetpack* представили архитектурное решение WorkManager, принимающее на себя проверку уровня API системы и в зависимости от этого выбирающее нужный механизм запуска и управления.

С ноября 2020 года было объявлено о завершении поддержки основных классов управления фоновыми потоками с переносом их функционала в пакет `WorkManager`. Это позволит разработчику не думать о средствах реализации и сконцентрироваться на самом решении поставленной задачи.

2. WorkManager

Менеджер рабочих задач включает в себя все лучшие практики организации фоновых процессов, работающих за пределами запускающего их приложения. Он содержит механизм определения установленной на устройстве операционной системы и в зависимости от этого выбирающий подходящий метод организации асинхронных потоков. При этом гарантируется, что процессы будут выполнены в любом случае.

Основные идеи работы `WorkManager`:

- организация работы только **фоновых** процессов;
- адаптивность под версию операционный системы;
- постановка процессов в очередь на выполнение;
- отслеживание системных ограничений выполнения процесса, а так же времени выполнения;
- возобновление прерванного процесса при появлении системной возможности его продолжения;
- планирование повторяющихся и одноразовых задач;
- сохранение очереди процессов при перезагрузке системы;
- построение цепочек процессов с последовательным или параллельным выполнением;
- передача данных внутри цепочки запущенных процессов;
- гарантия того, что запланированный процесс будет выполнен системой;
- оптимизация расхода заряда батареи.

Одним из главных преимуществ WorkManager является возможность построения последовательности из запускаемых процессов. При этом гарантируется, что следующий по очереди процесс не будет запущен прежде, чем отработает предыдущий (или предыдущие, если в цепочку добавлены несколько задач, запланированных на параллельное выполнение). Это позволяет реализовать передачу выходных данных предыдущего процесса в качестве входных данных в последующие процессы.

Очередь из процессов хранится в локальной базе данных SQLiteDatabase, организованной внутри WorkManager. Такой подход позволяет организовывать контроль за статусом выполненности задач, откладывать и возобновлять работу потока. Менеджер запускает задачи из базы данных, у которых поле "СТАТУС" имеет значение "ENQUEUED". И по этой же причине все запланированные задачи сохраняются (в отличие от AlarmManager) при перезагрузке устройства.

> Менеджер фоновых задач самостоятельно определяет возможность запуска фонового процесса (цепочки процессов). Таким образом, он не гарантирует выполнение задачи в чётко назначенное время., но гарантирует само выполнение и соблюдение последовательности работы процессов.
>
> Для выполнения заданий в назначенный срок по-прежнему используется AlarmManager

Для использования WorkManager в проекте, необходимо добавить зависимость в файл конфигурации модуля

```
dependencies {
    implementation "androidx.work:work-runtime-ktx:2.5.0"
}
```

Последнюю версию библиотеки можно узнать в документации Android Jetpack.

При подключении зависимости необходима настройка `compileSdkVersion` на уровень не ниже 28

Запуск процесса или цепочки процессов состоит из трёх шагов:

1. описание процесса;
2. настройка потока для запуска процесса;
3. запуск процесса на выполнение в подготовленном потоке.

Со всеми возможными сценариями использования `WorkManager` можно познакомиться в инструкции от разработчиков.

2.1. "Что делать?": подготовка процесса к запуску

WorkManager управляет жизнедеятельностью фоновых задач, но не занимается их постановкой. Полностью в организации рабочего пространства задействованы следующие классы:

Worker

Общий для Java и Kotlin класс, выполняющий задачу синхронно в потоке, организованном менеджером фоновых процессов. За выполнение работы отвечает метод класса doWork(), который возвращает экземпляр класса `androidx.work.ListenableWorker.Result`. Общее время работы метода не должно превышать 10 минут.

Класс абстрактный, по этому необходимо в проекте создать класс-наследник и реализовать в нём метод doWork()

```
class MyWork(context: Context, workerParams: WorkerParameters) : Worker(context, workerParams)  {
    override fun doWork(): Result {
        TODO("Not yet implemented")
    }
}
```

В методе `doWork()` даётся описание выполняемой деятельности: получение входных данных, подготовка и упаковка выходных данных или другие действия, выполнение которых планируется в текущем фоновом процессе.

Для описания деятельности при остановке процесса, переопределяется метод ListenableWorker.onStopped()

```
    override fun onStopped() {
        super.onStopped()
    }
```

CoroutineWorker

"Личный" класс Kotlin, аналогичен классу `Worker`, но позволяет процессу запуск в корутине. То есть метод `doWork()` будет запускать асинхронный процесс синхронно. Ещё одним преимуществом класса является наличие собственного метода `onStopped()`, то есть обработка остановки процесса будет выполняться автоматически без переопределения метода. Кроме этого, класс позволяет использовать диспетчеров при организации процесса.

```
class CoWork(appContext: Context, params: WorkerParameters) : CoroutineWorker(appContext, params) {
    override suspend fun doWork(): Result {
            TODO("Not yet implemented")
    }
}
```

В Kotlin рекомендуется использование именно этого класса.

Result

Класс возвращаемого значения из метода `doWork()`, вложен в класс androidx.work.ListenableWorker. Содержит статические методы индикации состояния процесса:

success() success(@NonNull outputData: Data)	Возвращает объект, указывающий на успешное завершение процесса. Обозначает возможность запуска на выполнение процессов, расположенных далее в очереди. Если у метода имеется аргумент, то передаёт его в цепочку процессов как выходные данные для дальнейшего их использования	Result.success() Result.success(Data.Builder().putString("request", "Yes!").build())
failure() failure(@NonNull outputData: Data)	Возвращает объект, указывающий на сбой выполнения процесса. Сигнализирует об остановке цепочки процессов и переводит текущий процесс в состояние "в очереди"	Result.failure() Result.failure(Data.Builder().putString("request", "No").build())
retry()	Запускает процесс на повторное выполнение после временного сбоя. Используется для процессов с условиями выполнения	Result.retry()

Кроме этого, в классе имеется метод `getOutputData()` обращения к выходным данным, который по ключу позволяет доставать данные из процесса, а в классе *ListenableWorker* свойство `inputData`, через которое можно получить значения входных данных, вызвав соответствующий типу данных геттер:

```
inputData.getString("request")
```

При извлечении данных базового типа или массива в геттер кроме ключа передаётся значение по умолчанию, которое будет возвращено в случае отсутствия запрашиваемых данных.

```
inputData.getChar("request",'n')
```

Data

Класс для представления данных, передаваемых по цепочке запускаемых процессов. Представляется ассоциативным массивом, элементы - ассоциативные пары *<ключ, значение>*. Ключ имеет тип данных String, значение - либо строка, либо значение примитивного типа, либо массив строк или элементов примитивного типа. У добавляемых данных есть предельный размер `10 Кб`, помещённый в константу `MAX_DATA_BYTES`.

Создание объекта типа `Data` выполняется через вложенный класс `Builder()` вызовом метода `build()`. В процессе создания в объект заполняются данные в виде ассоциативных пар методами `putType()` в зависимости от типа добавляемых данных.

```
val d = Data.Builder()
            .putString("name", "Noname")
            .putInt("age", 0)
            .build()
```

Для создания данных можно воспользоваться так же статическим методом workDataOf()

```
val d = workDataOf(
            "name" to "Noname",
            "age" to 0
        )
```

Пример класса процесса

```
class CoWork(appContext: Context, params: WorkerParameters) : CoroutineWorker(appContext, params) {
    override suspend fun doWork(): Result {
        Log.d("control_Co","start process")
        var sum = 0
        for(i in Int.MIN_VALUE .. Int.MAX_VALUE)
            sum +=i
        val data = Data.Builder().putInt("d", sum).build()
        Log.d("control_Co","the end of process")
        return Result.success(data)
    }
}
```

2.2. "Как делать?": подготовка потока для одноразового запуска

За этапом подготовки самого процесса следует этап настройки параметров запуска работы. Эта обязанность лежит на классе

WorkRequest.

Класс, а точнее его подклассы OneTimeWorkRequest и PeriodicWorkRequest, в которые передаётся задача, подготовленная в виде экземпляра класса `Worker` и производится наложение условий для запуска процесса на выполнение.

Процессы бывают двух видов:

- одноразовый процесс `OneTimeWorkRequest`;
- повторяющийся процесс `PeriodicWorkRequest`.

Создание один раз запускаемого потока для подготовленной задачи выполняется в классе запускающей активности прямым вызовом конструктора:

```
val secondProc = OneTimeWorkRequestBuilder<CoWork>().build()
```

или, как в java, через вложенный класс `Builder()` вызовом метода `build()`

```
val firstProc = OneTimeWorkRequest.Builder(MyWork::class.java).build()
```

а так же вызовом метода from()

```
val zeroProc = OneTimeWorkRequest.from(MyWork::class.java)
```

Параметры запуска процессов

Подготовленный поток можно сразу передать менеджеру для выполнения, но часто возникает необходимость дополнительных настроек запуска.

Наиболее часто используются условия запуска потока, которые представляются в виде объекта класса ограничений.

Constraints

Объект ограничений создаётся через вложенный класс `Builder()` добавлением условий и вызовом конструктора `build():

```
Constraints.Builder()./* список условий */.build()
```

В классе ограничений имеются следующие условия:

- setRequiredNetworkType(@NonNull networkType: NetworkType). Определяет тип сетевого подключения для выполнения процесса. В качестве аргумента принимает одно из значений

CONNECTED	Любое допустимое соединение. Проверяется просто факт наличия подключения
METERED	Наличие мобильного интернета
NOT_REQUIRED	Интернет-соединение не требуется
NOT_ROAMING	Любое допустимое соединение, но не в роуминге
TEMPORARILY_UNMETERED	Сеть, в которой временно отключен мобильный трафик. Распознаёт сети/, использующие мобильные данные, которые в данный момент отключены
UNMETERED	Любое сетевое подключение, кроме мобильного интернета

- setRequiresBatteryNotLow(requiresBatteryNotLow: Boolean). Ставит ограничение на уровень заряда батареи. Если ограничение требуется, то передаётся входной параметр `true`. Значение аргумента `false` соответствует отсутствию ограничения. Критическим является уровень заряда примерно 20%.
- setRequiresCharging(requiresCharging: Boolean). Условие подключения зарядного устройства. Если наличие текущей зарядки устройства требуется, то передаётся входной параметр `true`.
- setRequiresDeviceIdle(requiresDeviceIdle: Boolean). Устройство находится в режиме ожидания. Ограничение срабатывает на устройствах под управлением Android 6.0 и выше.
- setRequiresStorageNotLow(requiresStorageNotLow: Boolean). Ограничивает работу, если недостаточно долговременной памяти.

- addContentUriTrigger(@NonNull uri: Uri, triggerForDescendants: Boolean), setTriggerContentMaxDelay(), setTriggerContentUpdateDelay(). Наследованные из `JobScheduler` управления задержкой запуска задач в зависимости от изменения текущего контента.

Таким образом, поток, запуск которого возможен при наличии достаточного свободного дискового пространства, подключенном питании устройства и наличии немобильного интернета, получит следующий объект ограничений:

```
val constr = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.UNMETERED)
            .setRequiresStorageNotLow(true)
            .setRequiresCharging(true)
            .build()
```

Настройки запуска передаются в соответствующих сеттерах при построении объекта `WorkRequest`.

- setConstraints() настройка ограничений, передаваемых в виде объекта Constraints.Builder();

```
val t = OneTimeWorkRequestBuilder<MyWork>()
            .setConstraints(constr)
            .build()
```

- setBackoffCriteria() политика нового запуска задачи после получения результата `Result.retry() в предыдущем запуске. Регулирует время повторных попыток выполнения в двух режимах:

EXPONENTIAL интервал времени новых попыток запуска увеличивается по экспоненте. Это значение используется по умолчанию
LINEAR интервал времени новых попыток запуска увеличивается линейно
Так же в аргументах передаётся время ожидания перед повторной попыткой в виде целого числа и единиц измерения. С версии системы уровня API 26 время ожидания может быть передано объектом Duration.

```
val t = OneTimeWorkRequestBuilder<MyWork>()
            .setConstraints(constr)
            .setBackoffCriteria(BackoffPolicy.LINEAR,OneTimeWorkRequest.MIN_BACKOFF_MILLIS,TimeUnit.MILLISECONDS)
            .build()
```

- setInitialDelay() время задержки выполнения первого запуска процесса. В аргументах указывается целочисленное задержки и единицы измерения. С версии системы 8.0 аргументом может быть объект Duration

```
val t = OneTimeWorkRequestBuilder<MyWork>()
            .setConstraints(constr)
            .setBackoffCriteria(BackoffPolicy.LINEAR,OneTimeWorkRequest.MIN_BACKOFF_MILLIS,TimeUnit.MILLISECONDS)
            .setInitialDelay(15,TimeUnit.MINUTES)
            .build()
```

- setInputData передача в процесс входных данных в виде объекта Data, подготовленном заранее

```
val data = Data.Builder().putString("tag","Test data").build()
val t = OneTimeWorkRequestBuilder<MyWork>()
            .setConstraints(constr)
            .setBackoffCriteria(BackoffPolicy.LINEAR,OneTimeWorkRequest.MIN_BACKOFF_MILLIS,TimeUnit.MILLISECONDS)
            .setInitialDelay(15,TimeUnit.MINUTES)
            .setInputData(data)
            .build()
```

- addTag() настройка тегов для группировки процессов и последующей их остановки. У одного потока может быть установлено несколько тегов, тогда система его остановит, если один из тегов совпадёт с критерием останавливаемой группы;

```
val t = OneTimeWorkRequestBuilder<MyWork>()
            .setConstraints(constr)
            .setBackoffCriteria(BackoffPolicy.LINEAR,OneTimeWorkRequest.MIN_BACKOFF_MILLIS,TimeUnit.MILLISECONDS)
            .setInitialDelay(15,TimeUnit.MINUTES)
            .setInputData(data)
            .addTag("process_1")
            .build()
```

- keepResultsForAtLeast() указатель продолжительности времени хранения данных процесса. Не следует злоупотреблять данной настройкой, поскольку данные располагаются в оперативной памяти и на последующие процессы может не хватить ресурсов.

2.3. "Поехали!": запуск фонового потока

Выполнением подготовленного потока занимается непосредственно сам менеджер задач WorkManager

Запуск потоков

Подготовленные процессы ставятся в очередь на асинхронное выполнение вызовом метода enqueue(). `WorkManager` является синглтоном, по этому для получения экземпляра используется метод извлечения объекта getInstance(@NonNull context: Context) с передачей ему контекста создания.

```
WorkManager.getInstance(this).enqueue(nameProcess)
WorkManager.getInstance(this).enqueue(Arrays.asList(firstProc, secondProc))
```

Метод `enqueue()` в качестве аргумента получает процесс или список процессов и осуществляет его/их постановку в очередь на выполнение. Как только в системе выполнятся все условия для запуска очередного процесса, этот процесс будет запущен. Процессы, поданные в виде списка, будут запущены параллельно. Если по какой-то причине `doWork()` вернёт `Result.retry()`, процесс будет снова поставлен в очередь на выполнение, при этом его статус в соответствующем поле локальной базы данных снова станет `ENQUEUED`, что позволит системе снова запустить данный поток на выполнение.

Менеджер фоновых процессво позволяет запускать последовательности задач, называемых цепочкой процессов. При этом процессы, поданные на одном уровне, будут выполняться параллельно. Процессы, поставленные в цепочку на разных уровнях, будут выполняться последовательно. При этом:

- следующий уровень процессов не будет запущен до тех пор, пока не завершатся все процессы предыдущего уровня;
- если процесс текущего уровня будет остановлен, то остановлена будет вся последующая цепочка до выполнения в системе условий, обеспечивающих завершение остановленного процесса;
- цепочка имеет статус **уникальной**, если при её выполнении никакие другие процессы не могут быть запущены. Такая цепочка обязательно имеет имя, её идентифицирующее.

Построение цепочки осуществляется методами

- beginWith(): процесс или список процессов, которые в цепочке будут запущены первыми - *первый уровень запуска*
  - then() : процесс или список процессов, которые будут запущены после завершения выполнения всех процессов предыдущего уровня. Каждый следующий уровень в цепочку добавляется отдельным вызовом `then()`.

```
WorkManager.getInstance(this)
            .beginWith(Arrays.asList(firstProc, secondProc))
            .then(zeroProc)
            .then(firstProc,thirdProc,fourthProc)
            .enqueue()
```

В приведённой цепочке сначала будут параллельно запущены процессы firstProc и secondProc; после из завершения запустится zeroProc; после него будут параллельно запущены firstProc, thirdProc и fourthProc. При этом первый уровень запуска (firstProc и secondProc) может передать выходные данные второму уровню (zeroProc) в качестве входных данных. При этом можно выполнить объединение входных данных с двух процессов, используя возможности класса ArrayCreatingInputMerger:

```
.setInputMerger(ArrayCreatingInputMerger::class.java)
```

Отмена задания

Процедура меняет статус задачи в базе данных на значение `CANCELLED`. При этом будет выполнен метод `onStopped()` класса определения задачи (наследника `Woker`). Отменить выполнение процесса можно :

- по Id

```
WorkManager.getInstance(this).cancelWorkById(zeroProc.id)
```

- по тегу

```
WorkManager.getInstance(this).cancelAllWorkByTag("process_1")
```

- всю очередь процессов

```
WorkManager.getInstance(this).cancelAllWork()
```

Отслеживание статуса задачи

После постановки задания в очередь на выполнение можно установить наблюдение за изменением статуса процесса. Сделать это можно методом getWorkInfoByIdLiveData(@NonNull id: UUID).

```
WorkManager.getInstance(this).getWorkInfoByIdLiveData(zeroProc.id)
            .observe(this, androidx.lifecycle.Observer {
                Log.d("print_status",""+it.state)
            })
```

2.4. Периодический процесс

В классе повторяющихся процессов процессы можно организовать только двумя способами, `from()` здесь не применим:

```
val fourthProc = PeriodicWorkRequestBuilder<MyWork>(1000,TimeUnit.SECONDS)
val thirdProc = PeriodicWorkRequest.Builder(CoWork::class.java,1500,TimeUnit.SECONDS)
```

Числовой аргумент конструктора - минимальное значение временного интервала между двумя соседними запусками процесса. Он не указывает точного времени следующей итерации запуска. `PeriodicWorkRequest` использует метод setInexactRepeating() класса `AlarmManager` для версий ниже 23 или setPeriodic() класса `JobInfo` библиотеки JobScheduler для версий Android 23 и выше. При этом имеется возможность указать нижнюю границу интервала, тогда повторение запуска процесса будет выполнено не ранее, чем пройдёт указанный период

```
val fourthProc = PeriodicWorkRequestBuilder<MyWork>(15,TimeUnit.MINUTES, 30, TimeUnit.SECONDS)
```

В приведённом примере повторный процесс начнётся не раньше, чем через 30 секунд и не позже, чем через 15 минут от предыдущего запуска.

Последний аргумент методов - единицы измерения временного интервала повторения запусков. Устанавливается через константы класса TimeUnit.

В документации указаны наименьшие используемые значения

- *для временного интервала* - 900 секунд;
- *для нижней границы интервала* - 5 секунд.

Из повторяющихся процессов нельзя построить цепочки, в остальном принцип работы такой же, как и с одноразовым процессом.

3. Упражнение 4.9.1 Организация фоновых процессов

Создадим несколько задач и составим расписание их выполнения. Для этого

1. Напишите пять рабочих классов - наследников Worker или CoroutineWorker. В каждом классе проведите логирование начала и завершения процесса. В классах с поступающими входными данными можно слогировать и получаемые значения.

- В первом классе разместите музыкальное оформление работы приложения

```
class Woker1(c: Context, wp: WorkerParameters): Worker(c, wp) {
    override fun doWork(): Result {
        Log.d("tagg1","Work1 is start")
        val song: MediaPlayer = MediaPlayer.create(applicationContext, R.raw.song)
        song.start()
        Log.d("tagg1","Work1 is complete")
        return Result.success()
    }
}
```

обязательно добавьте в ресурсах папку `raw` для медиа -файлов и поместите туда файл `song.mp3`

- Во втором классе "усыпите" поток на 5 секунд и добавьте выходные данные о его пробуждении

```
class Worker2(context: Context, workerParams: WorkerParameters): Worker(context, workerParams) {
    override fun doWork(): Result {
        Log.d("tagg1","Work2 is start")
        sleep(5000)
        Log.d("tagg1","Work2 is complete")
        return Result.success(Data.Builder().putString("answer","Я проснулся").build())
    }
}
```

- Третий класс нагрузите вычислительным процессом суммирования всех целочисленных значений и передайте через него дальше сообщение из второго класса

```
class Worker3(context: Context, workerParams: WorkerParameters) : Worker(context, workerParams){
    override fun doWork(): Result {
        Log.d("tagg1","Work3 is start")
        val aS = inputData.getString("answer")
        var s: Long = 0
        for ( i in Int.MIN_VALUE..Int.MAX_VALUE){
            s+=i
        }
        val d = Data.Builder().putLong("resS",s).putString("answer", aS).build()
        Log.d("tagg1","Work3 is complete")
        return Result.success(d)
    }
}
```

- В четвёртом классе вычислите произведение положительных целых чисел

```
class Worker4(context: Context, workerParams: WorkerParameters):Worker(context, workerParams){
    override fun doWork(): Result {
        Log.d("tagg1","Work4 is start")
        var m: Long = 1
        for ( i in 1..500){
           m*=i
        }
        val d = Data.Builder().putLong("resM",m).build()
        Log.d("tagg1","Work4 is complete")
        return Result.success(d)
    }
}
```

- В пятом классе выполните деление двух входящих значений из предыдущего уровня запуска процессов

```
class Worker5(context: Context, workerParams: WorkerParameters) : Worker(context, workerParams){
    override fun doWork(): Result {
        Log.d("tagg1","Work5 is start")
        val sum = inputData.getLong("resS",0)
        val mult = inputData.getLong("resM",0)
        val aS = inputData.getString("answer")
        val res = mult.toFloat() / sum
        val output = workDataOf("result" to res,
            "s" to aS)
        Log.d("tagg1","Work5 is complete\n result: "+ output.getFloat("result",-10f) + "\t" + output.getString("s"))
        return Result.success(output)
    }
}
```

2. В методе `onCreate()` главного потока создайте однозапусковые процессы из классов Worker2 - Worker5 и повторяющийся процесс из класса Worker1ю

```
val pProc = PeriodicWorkRequestBuilder<Woker1>(3,TimeUnit.MINUTES).addTag("song").build()
        val proc2 = OneTimeWorkRequest.Builder(Worker2::class.java).build()
        val proc3 = OneTimeWorkRequestBuilder<Worker3>().build()
        val proc4 = OneTimeWorkRequestBuilder<Worker4>().build()
        val proc5 = OneTimeWorkRequestBuilder<Worker5>().build()
```

Повторяющемуся процессу установите тег, чтобы иметь возможность его останавливать.

3. Отмените повторяющийся процесс в методе `onStop()`

```
 override fun onStop() {
        super.onStop()
        WorkManager.getInstance(this).cancelAllWorkByTag("song")
    }
```

4. В методе `onCreate()` выполните запуск периодической задачи

```
WorkManager.getInstance(this).enqueue(pProc)
```

5. Там же создайте последовательность запусков сначала "спящего" процесса, затем два вычислительных процесса, и на третьем уровне запустите последний процесс из класса Work5

```
 WorkManager.getInstance(this)
            .beginWith(proc2)
            .then(Arrays.asList(proc3,proc4))
            .then(proc5)
            .enqueue()
```

6. Выполните запуск и отследите последовательность запусков процессов.

Выполненный проект сравните с заготовкой.

Начать тур для пользователя на этой странице