



Basic Android

Periodic Tasks, BroadcastReceiver



Проверить, идет ли запись

Меня хорошо видно & слышно?



Ставим "+", если все хорошо
"-", если есть проблемы



Тема вебинара

Periodic Tasks BroadcastReceiver



Раевский Виталий

Руководитель направления мобильной разработки (Альфа-Банк)

Об опыте :

15 лет опыта коммерческой разработки

@PanRaevsky



Правила вебинара



Активно
участвуем



Off-topic обсуждаем
в учебной группе
#Basic-Android-2024-09



Задаем вопрос
в чат или голосом



Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения



Индивидуально



Время, необходимое
на активность



Пишем в чат



Говорим голосом



Документ



Ответьте себе или
задайте вопрос

Маршрут вебинара



AlarmManager

JobScheduler (JobService)

WorkManager

BroadcastReceiver

Практика

Цели вебинара

К концу занятия вы сможете

1. Использовать AlarmManager и разработать свой будильник
2. Разрабатывать периодические задачи на базе WorkManager
3. Использовать BroadcastReceiver и JobScheduler

Смысл

Зачем вам это уметь

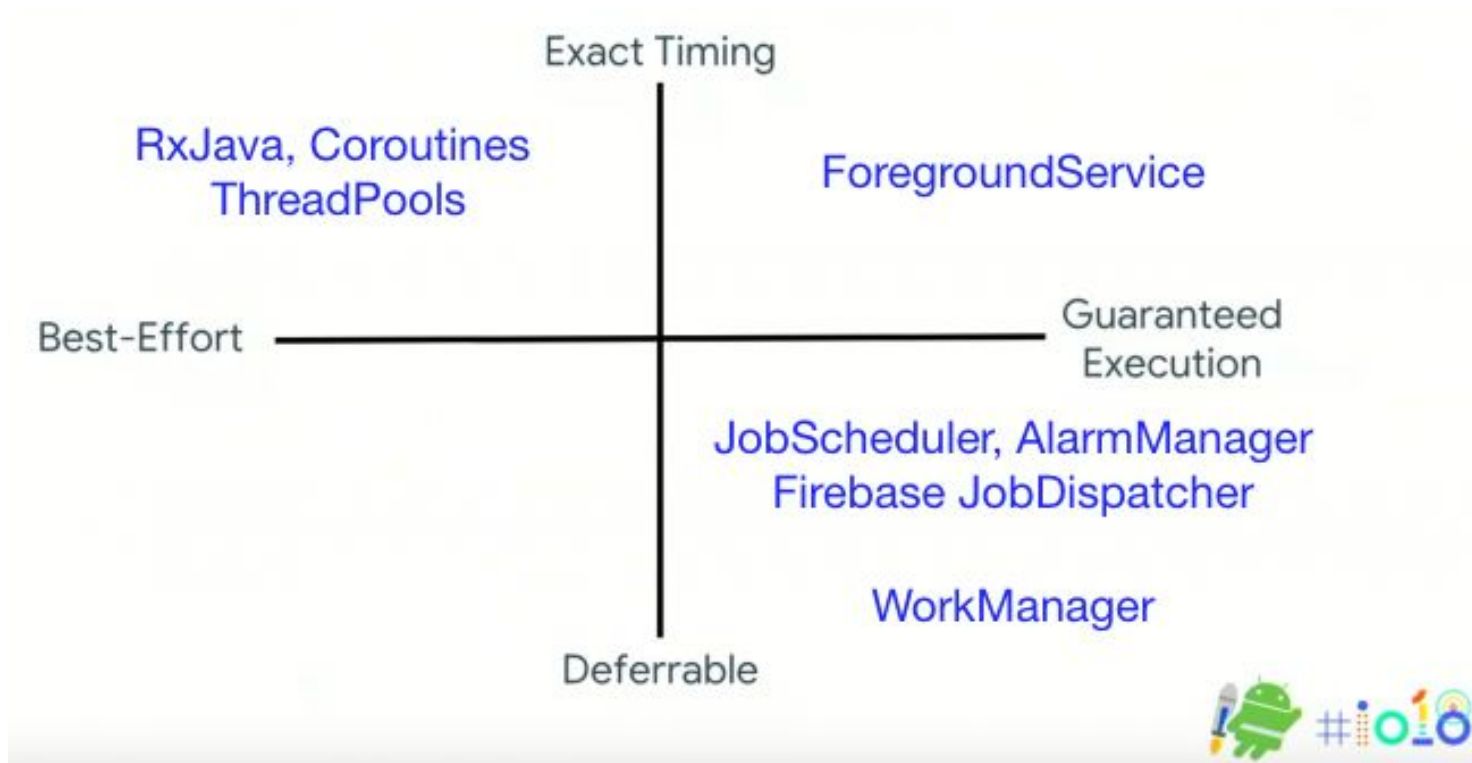
1. Научитесь разрабатывать приложения, не зависящие от ЖЦ Активности
2. Пригодится для работы с виджетами, музыкой, трекерами локации
3. Поймете на базе чего построены условия в WorkManager, JobScheduler



Приведите примеры приложений работающих в фоне?

AlarmManager

Типы задач



AlarmManager используется для отправки пользователю разовых или повторяющихся сообщений в заданное время.

Таким образом вы сможете создавать различные:

- планировщики
- будильники
- реализовать выполнение регулярных сетевых запросов
- запуска трудоёмких или дорогих операций, запланированных на определенное время и
- другие приложения, которые должны срабатывать по расписанию

Пример

```
val am = getSystemService(requireContext(), AlarmManager::class.java)
val intent = Intent("action", null, context, Receiver::class.java)
    .putExtra("extra", "extra")

val pIntentOnce = PendingIntent.getBroadcast(requireContext(), 0, intent, 0)
am?.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + 1_000, pIntentOnce)
```

Пример

```
val am = getSystemService(requireContext(), AlarmManager::class.java)
val intent = Intent("action", null, context, Receiver::class.java)
    .putExtra("extra", "extra")

val pIntentOnce = PendingIntent.getBroadcast(requireContext(), 0, intent, 0)
am?.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + 1_000, pIntentOnce)

am?.setRepeating(AlarmManager.ELAPSED_REALTIME,
    SystemClock.elapsedRealtime() + 500, 1_000, pIntentRepeat)
```

Пример

```
val am = getSystemService(requireContext(), AlarmManager::class.java)
val intent = Intent("action", null, context, Receiver::class.java)
    .putExtra("extra", "extra")

val pIntentOnce = PendingIntent.getBroadcast(requireContext(), 0, intent, 0)
am?.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + 1_000, pIntentOnce)

am?.setRepeating(AlarmManager.ELAPSED_REALTIME,
    SystemClock.elapsedRealtime() + 500, 1_000, pIntentRepeat)

am?.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
    SystemClock.elapsedRealtime() + AlarmManager.INTERVAL_HALF_HOUR,
    AlarmManager.INTERVAL_FIFTEEN_MINUTES, pIntentRepeat)
```

AlarmManager используется для отправки пользователю разовых или повторяющихся сообщений в заданное время.

- Метод **setRepeating()** нужен для повторяющихся оповещений в **точное время (будильник)**
-
- Метод **setInexactRepeating()** нужен для повторяющихся оповещений (**примерное время**)*
- Если вам нужно, чтобы AlarmManager срабатывал один раз – то нужно использовать метод **setAlarmClock()**

*When you use this method, Android synchronizes multiple inexact repeating alarms and fires them at the same time.

Способы запуска

- [ELAPSED_REALTIME](#)
- [ELAPSED_REALTIME_WAKEUP](#)
- [RTC](#)
- [RTC_WAKEUP](#)

PendingIntent используется для описания интента, выполнение которого отложено во времени.

Инстанс класса `PendingIntent` создается одним из статических методов:

- `PendingIntent.getActivity()`
- `PendingIntent.getActivities()`
- `PendingIntent.getBroadcast()`
- `PendingIntent.getService()`

При создании в `PendingIntent` записывается информация о желаемом интенте и о том, какой компонент будет запущен.

Здесь **имя метода определяет, какой тип объекта будет вызван с помощью нашего Intent.**

Объекты `PendingIntent` переживают остановку процесса, поэтому система может использовать `PendingIntent` для старта приложения.

PendingIntent используется для описания интента, выполнение которого отложено во времени.

- **Уведомления.** Чтобы перехватить их, мы можем создать NotificationListenerService, который будет слушать все уведомления приложений и извлекать из них отложенные интенты.
- **SliceProvider.** Этот механизм используется для встраивания частей приложения в другие приложения, например встраивания переключателя быстрых настроек в окно ассистента. Мы можем получить слайс с помощью класса SliceViewManager или ContentResolver и затем получить PendingIntent всех слайсов.
- **MediaBrowserService.** Механизм, позволяющий приложению дать доступ к своей медиатеке с возможностью включить проигрывание медиафайлов. Получить PendingIntent можно, подключившись к приложению с помощью класса MediaBrowser.
- **Виджеты** рабочего стола используют отложенные интенты в качестве действий при нажатии на свои элементы. Класс AppWidgetHost позволяет приложению прикинуться лаунчером и получить доступ к виджетам приложений. Далее PendingIntent можно извлечь из самого виджета.

PendingIntent используется для описания интента, выполнение которого отложено во времени.

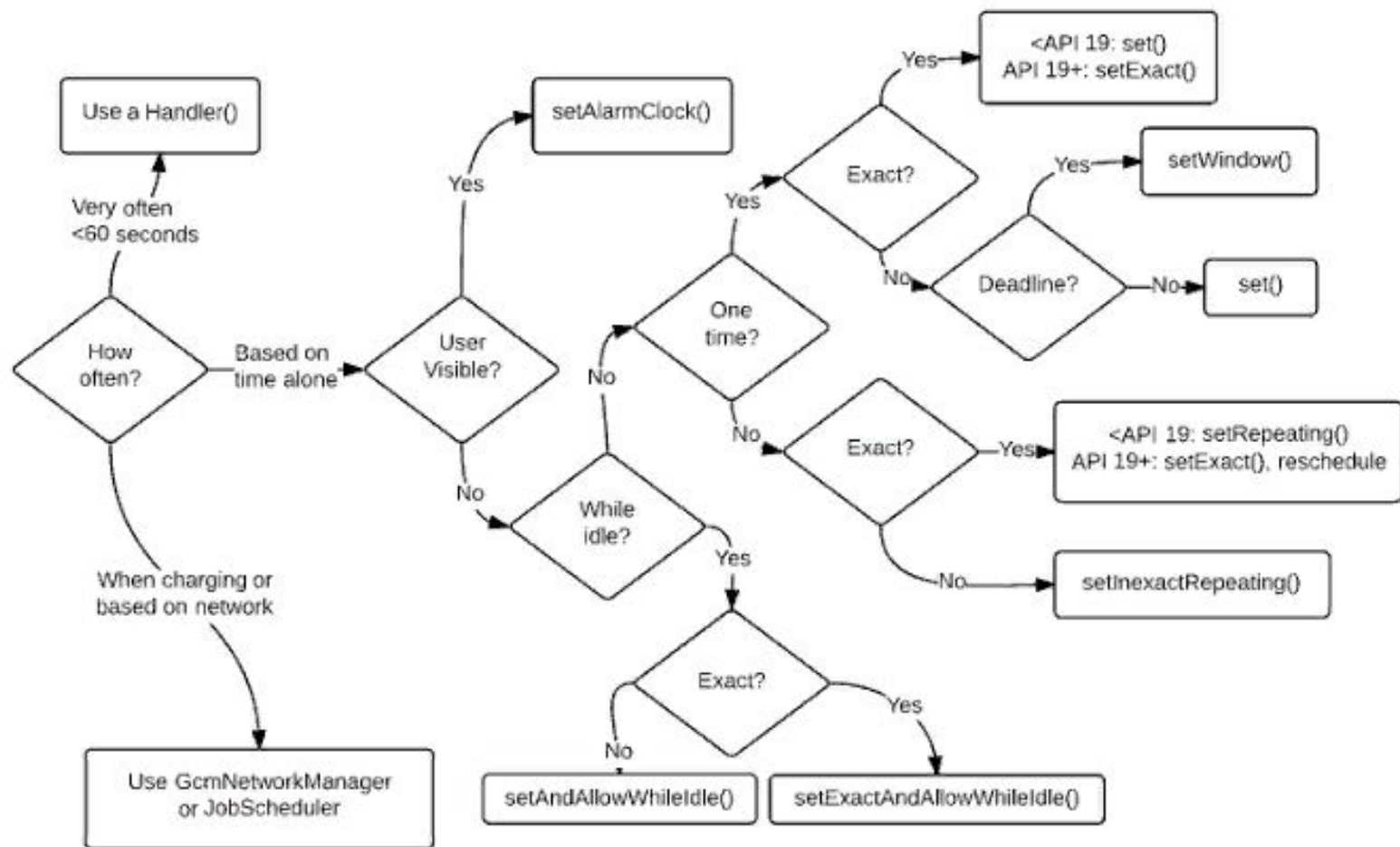
Инстанс класса PendingIntent создается одним из статических методов:

- FLAG_IMMUTABLE
- FLAG_MUTABLE
- FLAG_UPDATE_CURRENT
- FLAG_ONE_SHOT
- FLAG_CANCEL_CURRENT

```
private fun setRepeatingAlarm() {  
    Log.d(TAG, msg: "setRepeatingAlarm")  
    val intent = createIntent( action: "action Repeating", extra: "extra Repeating")  
    pIntentRepeat = PendingIntent.getBroadcast(requireContext(), requestCode: 0, intent, PendingIntent.FLAG_UPDATE_CURRENT)  
    am?.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,  
        triggerAtMillis: SystemClock.elapsedRealtime() + 500, intervalMillis: 1_000, pIntentRepeat)  
}
```

LIVE

JobScheduler (JobService)



*screen off
stationary
on battery*

*maintenance
window*

Doze

Doze



time



Doze Mode

Когда устройство на Android Marshmallow лежит без движения и без зарядки, спустя час оно переходит в Doze Mode. Режим отключки, когда почти все приложения перестают жрать батарею.

Это происходит не сразу, а по шагам:

ACTIVE — Устройство используется или на зарядке

INACTIVE — Устройство недавно вышло из активного режима (пользователь выключил экран, выдернул зарядку и т.п.)

...30 минут

IDLE_PENDING — Устройство готовится перейти в режим ожидания

...30 минут

IDLE — Устройство в режиме бездействия

IDLE_MAINTENANCE — Открыто короткое окно, чтобы приложения выполнили свою работу

В момент, когда устройство переходит в состояние IDLE:

- Доступ приложению к сети отключен, пока приложение не получит high-priority GCM-push.
- Система игнорирует Wake lock'и. Приложения могут сколько угодно пытаться запросить пробуждение процессора — они их не получают.
- Alarm'ы запланированные в AlarmManager не будут вызываться, кроме тех, которые будут обновлены с помощью setAndAllowWhileIdle().
- Система не производит поиска сетей Wi-Fi.
- NetworkPolicyManagerService: пропускает только приложения из белого списка.
- JobSchedulerService: все текущие задачи отменяются. Новые откладываются до пробуждения.
- SyncManager: все текущие отменяются, новые откладываются до пробуждения.
- PowerManagerService: только задачи приложений из белого списка вызовутся.

Классы для работы с JobScheduler

- JobScheduler - менеджер, который управляет задачами

Классы для работы с JobScheduler

- JobScheduler - менеджер, который управляет задачами
- JobService - логика работы

Классы для работы с JobScheduler

- JobScheduler - менеджер, который управляет задачами
- JobService - логика работы
- ComponentName - компонент для запуска

Классы для работы с JobScheduler

- JobScheduler - менеджер, который управляет задачами
- JobService - логика работы
- ComponentName - компонент для запуска
- JobInfo - параметры

```
class MainJobService : JobService() {  
  
    override fun onStartJob(jobParameters: JobParameters?): Boolean {  
        //do something  
        return true  
    }  
  
    override fun onStopJob(p0: JobParameters?): Boolean {  
        //do something  
        return true  
    }  
  
}
```

```
class MainJobService : JobService() {  
  
    override fun onStartJob(jobParameters: JobParameters?): Boolean {  
        //do something  
        return false  
    }  
  
    override fun onStopJob(p0: JobParameters?): Boolean {  
        //do something  
        return false  
    }  
  
}
```

```
<service  
    android:name=".MainJobService"  
    android:permission="android.permission.BIND_JOB_SERVICE" />
```



- **onStartJob()** вызывается когда настает время (условия) для выполнения запланированной задачи. Этот метод вызывается в главном потоке и любые тяжелые операции разработчик должен самостоятельно выносить в отдельные потоки
- При делегировании выполнения задачи в другие потоки из **onStartJob()** необходимо вернуть **true**, а если все необходимые действия уже выполнены в теле этого метода, то вернуть нужно **false**.
- **onStopJob()** вызывается тогда, когда требуемые условия для задачи перестали выполняться либо отведенное для задачи время исчерпано. Вызов этого метода информирует сервис, что все фоновые задачи немедленно должны перестать выполняться.
- **onStopJob()** также имеет возвращаемое значение, если это **true** — то *JobScheduler* поставит прерванную задачу в очередь выполнения снова, **false** — задача будет считаться выполненной и будет удалена из очереди, если она не была периодической.

```
val componentName = ComponentName(context, MainJobService::class.java)
```

```
val componentName = ComponentName(context, MainJobService::class.java)
```

```
val jobInfo = JobInfo.Builder(JOB_ID, componentName)  
    .setMinimumLatency(TimeUnit.SECONDS.toMillis(60))  
    .setOverrideDeadline(TimeUnit.SECONDS.toMillis(10))  
    .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)  
    .setRequiresDeviceIdle(false)  
    .setRequiresCharging(false)  
    .setPeriodic(TimeUnit.MINUTES.toMillis(1))  
    .setPersisted(true)  
    .setBackoffCriteria(JobInfo.DEFAULT_INITIAL_BACKOFF_MILLIS,  
        JobInfo.BACKOFF_POLICY_LINEAR)  
    .build()
```



```
val componentName = ComponentName(context, MainJobService::class.java)
```

```
val jobInfo = JobInfo.Builder(JOB_ID, componentName)  
    .setMinimumLatency(TimeUnit.SECONDS.toMillis(60))  
    .setOverrideDeadline(TimeUnit.SECONDS.toMillis(10))  
    .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)  
    .setRequiresDeviceIdle(false)  
    .setRequiresCharging(false)  
    .setPeriodic(TimeUnit.MINUTES.toMillis(1))  
    .setPersisted(true)  
    .setBackoffCriteria(JobInfo.DEFAULT_INITIAL_BACKOFF_MILLIS,  
JobInfo.BACKOFF_POLICY_LINEAR)  
    .build()
```

```
val jobScheduler = context.getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler  
jobScheduler.schedule(jobInfo)
```

```
val componentName = ComponentName(context, MainJobService::class.java)
```

```
val jobInfo = JobInfo.Builder(JOB_ID, componentName)
    .setMinimumLatency(TimeUnit.SECONDS.toMillis(60))
    .setOverrideDeadline(TimeUnit.SECONDS.toMillis(10))
    .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
    .setRequiresDeviceIdle(false)
    .setRequiresCharging(false)
    .setPeriodic(TimeUnit.MINUTES.toMillis(1))
    .setPersisted(true)
    .setBackoffCriteria(JobInfo.DEFAULT_INITIAL_BACKOFF_MILLIS,
        JobInfo.BACKOFF_POLICY_LINEAR)
    .build()
```

```
val jobScheduler = context.getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler
jobScheduler.schedule(jobInfo)
```

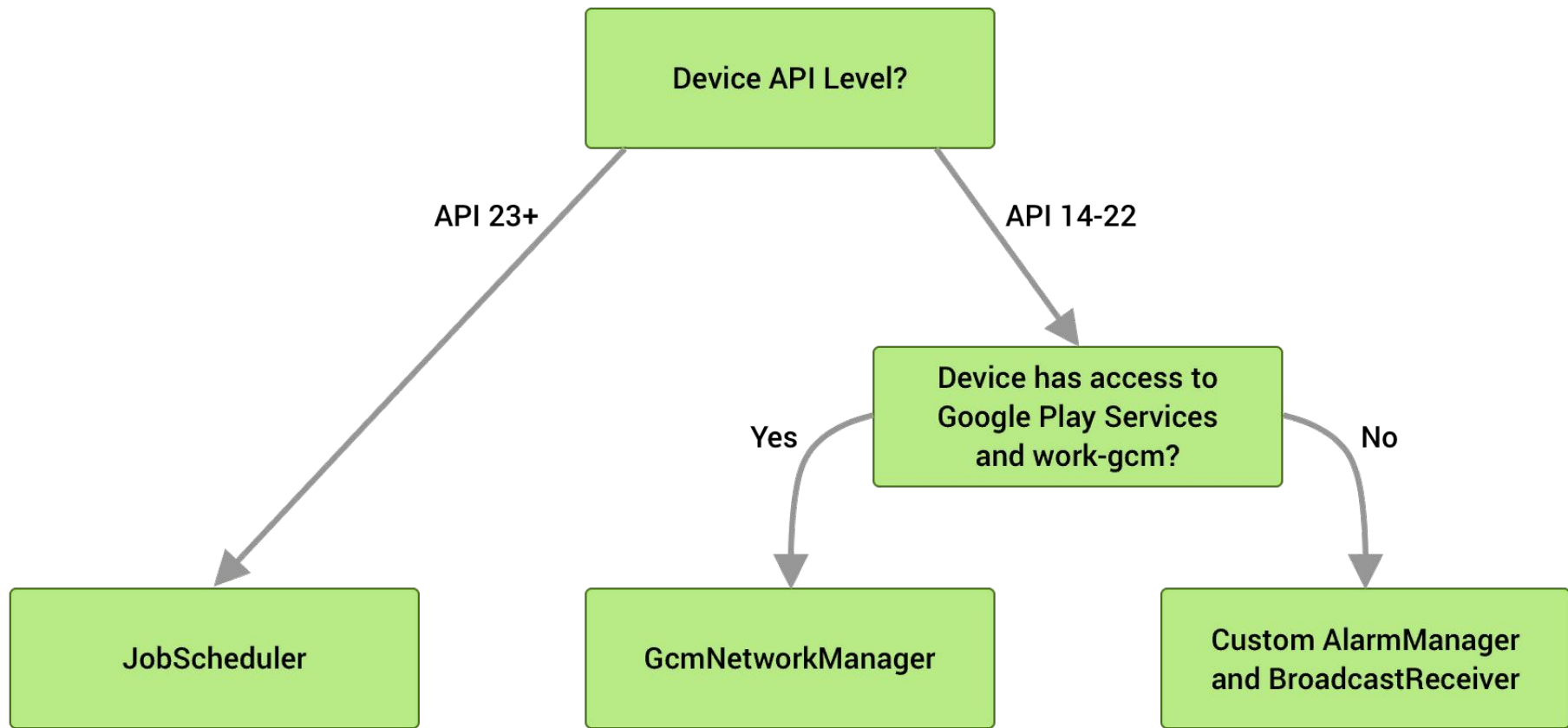
```
jobScheduler.cancel(JOB_ID)
```

LIVE

WorkManager

- **Запуск отложенных задач.** Это позволяло разгрузить приложение от задач, выполнение которых не требуется немедленно - например, синхронизации с сервером в определенное время суток.
- **Constraints** (Констрейнты, ограничения) позволяли запускать задачи на основе определенных состояний устройства – например, только при наличии сети, во время зарядки или в спящем режиме.
- Возможность реализовать **последовательность или цепочку** из нескольких задач, которые зависят от выполнения предыдущих.
- WorkManager поддерживает не только **разовые задачи**, но и задачи, которые должны выполняться **периодически**.
- А также WorkManager позволял узнать и подписаться на **статус выполнения**: находится ли работа в очереди, выполняется, заблокирована или завершена.

- Изначально это был инструмент только для отложенной фоновой работы, но в последних версиях Workmanager появилось несколько новых функций.
- Появилась возможность запросить немедленное выполнение задачи, если приложение находится на переднем плане, и быть при этом уверенным в завершении этой работы.
- Например, приложение может продолжать обрабатывать фото, загружая их на сервер, даже если пользователь уже делает что-то другое.
- WorkManager имеет свою базу данных, что позволяет сохранять информацию и статус работы в случае сбоя процесса или даже перезагрузки устройства. Это дает возможность восстановить статус выполнения работы и продолжить его с того места, на котором остановились, независимо от обстоятельств.



Классы для работы с WorkManager

- Worker — логика работы

Классы для работы с WorkManager

- Worker — логика работы
- WorkRequest — логика запуска задачи (OneTimeWorkRequest / PeriodicWorkRequest)

Классы для работы с WorkManager

- Worker — логика работы
- WorkRequest — логика запуска задачи (OneTimeWorkRequest / PeriodicWorkRequest)
- WorkRequest.Builder — параметры

Классы для работы с WorkManager

- Worker — логика работы
- WorkRequest — логика запуска задачи (OneTimeWorkRequest / PeriodicWorkRequest)
- WorkRequest.Builder — параметры
- Constraints — условия

Классы для работы с WorkManager

- Worker — логика работы
- WorkRequest — логика запуска задачи (OneTimeWorkRequest / PeriodicWorkRequest)
- WorkRequest.Builder — параметры
- Constraints — условия
- WorkManager — менеджер, который управляет задачами

Классы для работы с WorkManager

- Worker — логика работы
- WorkRequest — логика запуска задачи (OneTimeWorkRequest / PeriodicWorkRequest)
- WorkRequest.Builder — параметры
- Constraints — условия
- WorkManager — менеджер, который управляет задачами
- WorkStatus — статус задачи


```
dependencies {  
    def work_version = "2.3.4"  
  
    // (Java only)  
    implementation "androidx.work:work-runtime:$work_version"  
  
    // Kotlin + coroutines  
    implementation "androidx.work:work-runtime-ktx:$work_version"  
  
    // optional - RxJava2 support  
    implementation "androidx.work:work-rxjava2:$work_version"  
  
    // optional - GCMNetworkManager support  
    implementation "androidx.work:work-gcm:$work_version"  
  
    // optional - Test helpers  
    androidTestImplementation "androidx.work:work-testing:$work_version"  
}
```



```
class UploadWorker(appContext: Context, workerParams: WorkerParameters)
    : Worker(appContext, workerParams) {

    override fun doWork(): Result {
        // Do the work here--in this case, upload the images.

        uploadImages()

        // Indicate whether the task finished successfully with the Result
        return Result.success()
    }
}
```

```
val constraints = Constraints.Builder()  
    .setRequiresBatteryNotLow(true)  
    .setRequiresCharging(true)  
    .setRequiresDeviceIdle(true)  
    .setRequiresStorageNotLow(true)  
    .setRequiredNetworkType(NetworkType.UNMETERED)  
    .build()
```



```
val constraints = Constraints.Builder()  
    .setRequiresBatteryNotLow(true)  
    .setRequiresCharging(true)  
    .setRequiresDeviceIdle(true)  
    .setRequiresStorageNotLow(true)  
    .setRequiredNetworkType(NetworkType.UNMETERED)  
    .build()
```

```
val upload = OneTimeWorkRequestBuilder<UploadWorker>()  
    .setInputData(createInputData())  
    .addTag(Constants.TAG_OUTPUT)  
    .setConstraints(constraints)  
    .build()
```



```
val constraints = Constraints.Builder()  
    .setRequiresBatteryNotLow(true)  
    .setRequiresCharging(true)  
    .setRequiresDeviceIdle(true)  
    .setRequiresStorageNotLow(true)  
    .setRequiredNetworkType(NetworkType.UNMETERED)  
    .build()
```

```
val upload = OneTimeWorkRequestBuilder<UploadWorker>()  
    .setInputData(createInputData())  
    .addTag(Constants.TAG_OUTPUT)  
    .setConstraints(constraints)  
    .build()
```

```
WorkManager.getInstance(mContext).enqueue(upload)
```

WorkManager.getInstance(**mContext**).cancelAllWorkByTag(Constants.**TAG_OUTPUT**)

WorkManager.getInstance(**mContext**).cancelAllWork()

```
WorkManager.getInstance(mContext).cancelAllWorkByTag(Constants.TAG_OUTPUT)
```

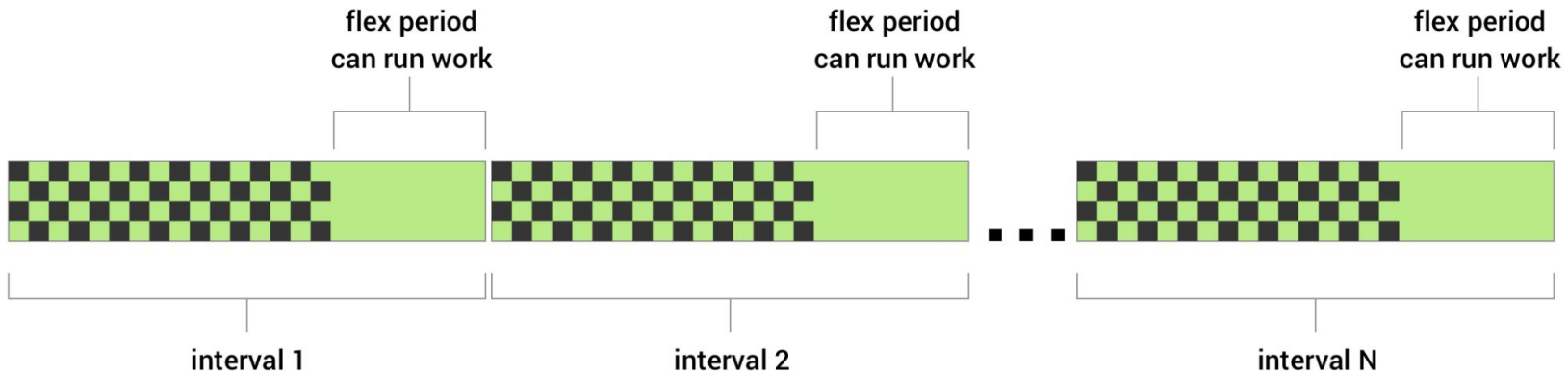
```
WorkManager.getInstance(mContext).cancelAllWork()
```

```
WorkManager.getInstance(mContext)
    .getWorkInfosByTagLiveData(Constants.TAG_OUTPUT)
    .observe(this, Observer { wInfo ->
        Log.d(TAG, "onChanged: " + wInfo.state)
        if (wInfo.state == WorkInfo.State.SUCCEEDED) {

        }
    })
```



```
val refreshWork = PeriodicWorkRequest.Builder(  
    UploadWorker::class.java,  
    1, TimeUnit.HOUR,  
    15, TimeUnit.MINUTES)  
    .addTag(Constants.TAG_OUTPUT)  
    .build()
```




```
var continuation = WorkManager.getInstance(mContext)
    .beginUniqueWork(Constants.IMAGE_MANIPULATION_WORK_NAME,
        ExistingWorkPolicy.REPLACE,
        OneTimeWorkRequest.from(CleanupWorker::class.java))
```

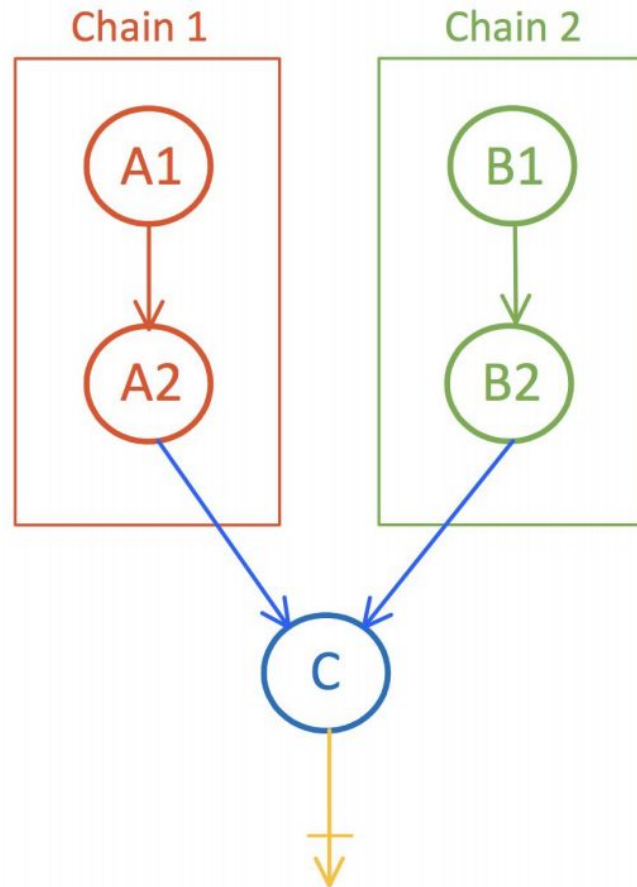
```
val upload = OneTimeWorkRequestBuilder<UploadWorker>()
    .setInputData(createInputData())
    .addTag(Constants.TAG_OUTPUT)
    .build()
continuation = continuation.then(upload)
```

```
continuation.enqueue()
```

```
val workA1 = OneTimeWorkRequest.Builder(SimpleBgTask::class.java)
    .addTag("tag1").build()
val workA2 = OneTimeWorkRequest.Builder(SimpleBgTask2::class.java)
    .addTag("tag1").build()
val workB1 = OneTimeWorkRequest.Builder(SimpleBgTask::class.java)
    .addTag("tag1").build()
val workB2 = OneTimeWorkRequest.Builder(SimpleBgTask2::class.java)
    .addTag("tag1").build()
val workC = OneTimeWorkRequest.Builder(SimpleBgTask3::class.java)
    .addTag("tag1").build()

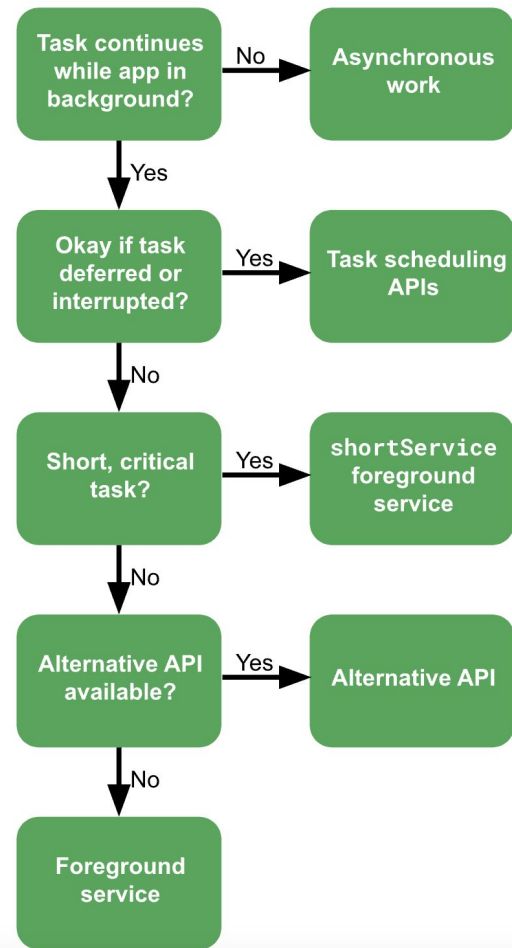
val manager = WorkManager.getInstance(this)
val chain1 = manager.beginWith(workA1).then(workA2)
val chain2 = manager.beginWith(workB1).then(workB2)
val chain = WorkContinuation.combine(listOf(chain1, chain2))
    .then(workC)

chain.enqueue()
```



















LIVE

Алгоритм принятия решения

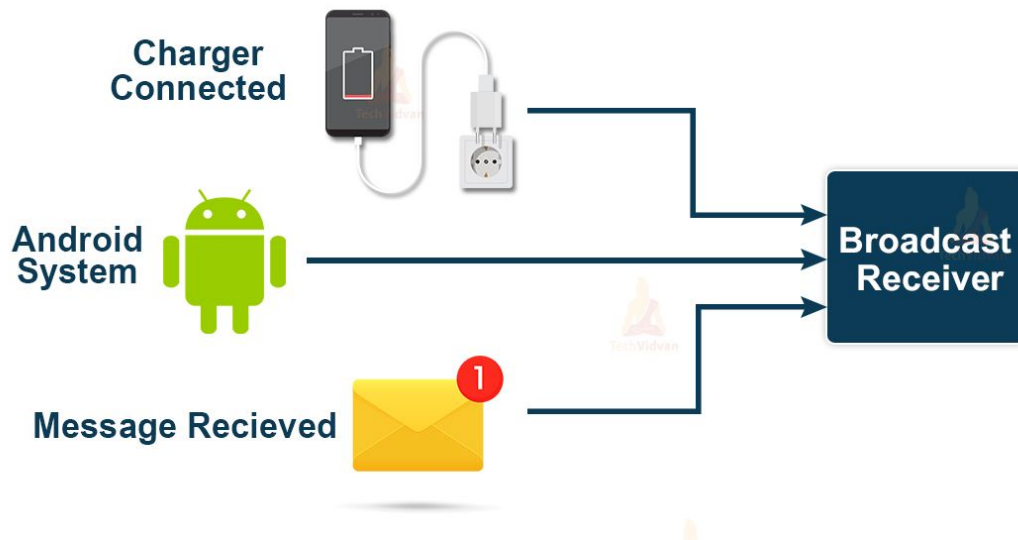


Use Case	Examples	Solution
Guaranteed execution of deferrable work	<ul style="list-style-type: none"> • Upload logs to your server • Encrypt/Decrypt content to upload/download 	WorkManager
A task initiated in response to an external event	<ul style="list-style-type: none"> • Syncing new online content like email 	FCM + WorkManager
Continue user-initiated work that needs to run immediately even if the user leaves the app	<ul style="list-style-type: none"> • Music player • Tracking activity • Transit navigation 	Foreground Service
Trigger actions that involve user interactions, like notifications at an exact time.	<ul style="list-style-type: none"> • Alarm clock • Medicine reminder • Notification about a TV show that is about to start 	AlarmManager

		Jobs	Alarms	High Priority FCM	Network
Active	   	Unrestricted			
Working Set	   	Restricted		Unrestricted	
Frequent	   	Restricted			Unrestricted
Rare	   	Restricted			

BroadcastReceiver

BroadcastReceiver — приемник сообщений, посылаемых системой или другим приложением при каком-либо событии,



BroadcastReceiver — приемник сообщений, посылаемых системой или другим приложением при каком-либо событии,

- Системные события или события от других приложений
- Полный перечень можно найти в файле `broadcast_actions.txt`
- Например `AIRPLANE_MODE`, `BATTERY_LOW` etc

Регистрация в AndroidManifest.xml

```
<!-- If this receiver listens for broadcasts sent from the system or from
      other apps, even other apps that you own, set android:exported to "true". -->
<receiver android:name=".MyBroadcastReceiver" android:exported="false">
    <intent-filter>
        <action android:name="APP_SPECIFIC_BROADCAST" />
    </intent-filter>
</receiver>
```

Kotlin

Java

```
val listenToBroadcastsFromOtherApps = false
val receiverFlags = if (listenToBroadcastsFromOtherApps) {
    ContextCompat.RECEIVER_EXPORTED
} else {
    ContextCompat.RECEIVER_NOT_EXPORTED
}
```



Caution: If the broadcast receiver is exported, other apps could send unprotected broadcasts to your app.



Получение данных

```
class SampleBootReceiver : BroadcastReceiver() {  
  
    override fun onReceive(context: Context, intent: Intent) {  
        if (intent.action == "android.intent.action.BOOT_COMPLETED") {  
            // Set the alarm here.  
        }  
    }  
}
```

Ключевые тезисы

1. AlarmManager позволяет реализовать приложения по типу будильник, календарь с напоминаниями
2. WorkManager и JobScheduler позволяют реализовать периодические задачи
3. Для настройки WorkManager используются constraints (например можно указать тип сети, уровень заряда батареи)

Вопросы?



Ставим “+”,
если вопросы есть



Ставим “-”,
если вопросов нет

Список материалов для изучения

1. Документация Android <https://developer.android.com/develop/background-work/background-tasks>
2. Документация BroadcastReceivers
<https://developer.android.com/develop/background-work/background-tasks/broadcasts>
3. Don't Kill My App <https://dontkillmyapp.com/>

**Заполните, пожалуйста,
опрос о занятии
по ссылке в чате**