Développement Android (4.3)

Multithreading

BENJAMIN BILLET

WARNING

Le contenu de cette présentation est basé sur la documentation anglophone officielle d'Android, diffusée sous licence *Creative Commons Attribution 2.5* :

developer.android.com

La plupart des schémas qui composent ce cours proviennent de cette documentation et sont, par conséquent, soumis à cette même licence.

http://creativecommons.org/licenses/by/2.5/

Au Sommaire !

- Processus et Thread
- Handler/Message
- AsyncTask
- Alarm & Power Managers

PROCESSUS & THREAD

Processus

- Par défaut Android exécute une application et ses composants dans un même processus Linux et dans un même thread (le main thread).
- Quand un composant est démarré, le système cherche d'autres composants actifs pour cette application, et utilise le même processus et le même thread le cas échéant.
- Android permet toutefois aux composants de s'exécuter dans des processus différents, conformément aux directives du manifest.

Processus

- android:process peut être utilisé sur n'importe quel tag <activity>, <service>, <receiver> et <provider>.
- Un nom de processus commençant par ":" indique que le processus est privé.
- Un nom de processus commençant par une lettre minuscule indique que le processus est global et utilisable par les autres applications (si tant est qu'elles partagent le même user ID et soient signées par un même certificat).

```
<service android:name="WordService"
    android:process=":my_process"
    android:icon="@drawable/icon"
    android:label="@string/service_name" >
</service>
```

Processus

- Android classe les processus selon leur importance et détruit, si nécessaire, les moins important en premier.
- Cinq classes d'importance :
  - Foreground process : un processus directement impliqué dans les actions de l'utilisateur.
  - Visible process : un processus qui n'est pas au premier plan, mais qui a un impact sur ce que l'utilisateur voit à l'écran.
  - Service process : un processus qui exécute un service.
  - Background process : un processus qui n'a plus d'impact sur l'expérience utilisateur.
  - Empty process : un processus vide, maintenu pour accélérer l'éventuelle réexécution de l'application.

PROCESSUS

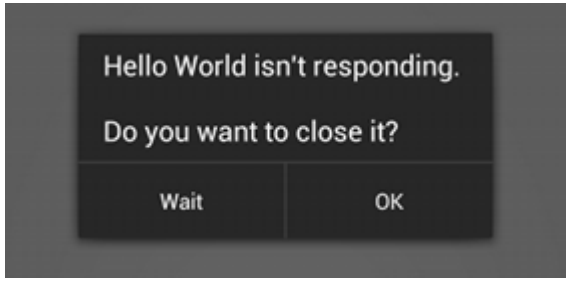
- Foreground process si héberge :
  - Une activité que l'utilisateur manipule (onResume() a été invoquée).
  - Un service lié à cette activité.
  - Un service qui s'exécute au premier plan (démarré avec startForeground()).
  - Un service exécutant un callback onCreate(), onStart() ou onDestroy(), ou un receiver exécutant onReceive().
- Visible process si héberge :
  - Une activité encore visible à l'écran (onPause() a été invoquée), par exemple derrière une boîte de dialogue.
  - Un service lié à cette activité.
- Service process si héberge un service démarré avec startService().
- Background process si héberge des activités non visibles à l'écran (onStop() a été invoqué).
- Empty process si aucun composant n'est hébergé.

Processus

- Android évalue un processus à partir des plus hauts level de chaque composant hébergé.
- La classe d'un processus dépend aussi des processus qui en dépendent. Par exemple :
  - Si un provider du processus A sert un client du processus B, alors A est considéré au moins aussi important que B.
  - Même chose dans le cas d'un composant de B lié à un service de A.
- Remarque : Comme les service process sont mieux placés que les background process, les opérations longues devraient être exécutées dans des services plutôt que dans des threads.

Le Main Thread

- Android gère l'interface et ses évènements dans un thread unique, appelé UI thread ou main thread.



- Tous les évènements sont gérées dans une file d'attente et traitées par un Looper.
- Si le développeur bloque ce thread, alors l'application entière ne répond plus.
- Par extension, Android interdit l'utilisation des vues ailleurs que dans l'UI thread.

Thread Java classique

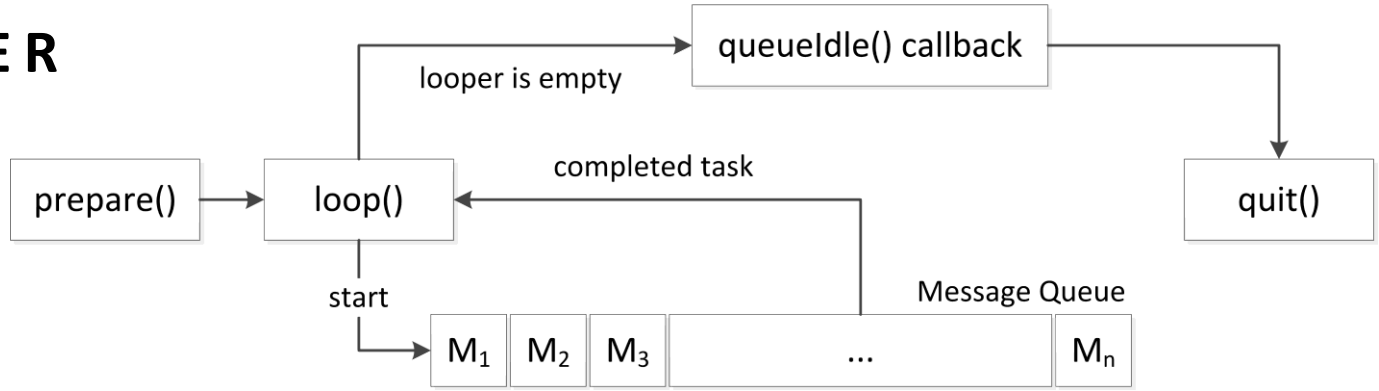
- Android supporte la classe Thread, pour laquelle il est nécessaire de gérer :
  - La synchronisation et la communication avec le main thread.
  - L'interruption des threads.
  - Les pools.
  - Les effets du cycle de vie des activités (e.g., runtime change).
- Le package java.util.concurrent est aussi entièrement supporté (Java 6).

HANDLER/MESSAGE

Handler

- Un handler est un canal de message vers le thread qui l'a instancié.
- Un handler permet notamment à un thread de communiquer avec l'UI thread (passage de résultat pour mettre à jour l'interface graphique par exemple).
- Les messages échangés ne sont pas des Intent, mais des objets Message ou des Runnable (e.g., demande d'exécution de code directement dans l'UI thread).

LOOPER



- Un looper est utilisé pour créer une boucle de message dans un thread.
- Sans looper, un thread ne peut pas instancier un handler (RuntimeException).
- L'UI thread gère son propre Looper, mais les autres threads doivent les créer explicitement si besoin.
- Un looper doit être terminé explicitement avec quit().

Exécution de code

```
new Thread(new Runnable()
{
     @Override
     public void run()
     {
          final Bitmap bitmap = loadSomeImage("http://inria.fr/anImage.png");
          imageView.post(new Runnable()
          {
               @Override
               public void run()
               {
                    imageView.setImageBitmap(bitmap);
               }
          });

          imageView.postDelayed(new Runnable()
          {
               @Override
               public void run()
               {
                    imageView.setImageBitmap(bitmap);
               }
          }, 5000); // in five seconds

          myActivity.runOnUiThread(new Runnable()
          {
               @Override
               public void run()
               {
                    imageView.setImageBitmap(bitmap);
               }
          });
     }
}).start();
```

Remarque : ce type de construction favorise le code “spaghetti”, et il est préférable d’échanger des messages entre les threads plutôt que des Runnable.

Un simple handler (création)

```
public class MyActivity extends Activity
{
	private Handler handler;

	@Override
	protected void onCreate(Bundle savedInstanceState)
	{
		super.onCreate(savedInstanceState);
		handler = new Handler(); // = new Handler(Looper.getMainLooper()) in this specific case
		{
			@Override
			public void handleMessage(Message inputMessage)
			{
				switch (inputMessage.what)
				{
					case 1:
						MyClass something = (MyClass) inputMessage.obj;
						...
						break;
					...
					default:
						super.handleMessage(inputMessage);
				}
			}
		}
	}
}
```

- Un handler nécessite un Looper (celui du thread courant ou le Looper d'un autre thread).
- Looper.getMainLooper() retourne le Looper de l'UI thread.
- Looper.myLooper() retourn le Looper du thread courant.

Un simple handler (envoi)

```
public class Something extends Thread
{
    private Handler handler;
    public Something(Handler handler)
    {
        this.handler = handler;
    }

    @Override
    public void run()
    {
        Message message = Message.obtain();
        message.obj = someObject;
        message.what = 2;
        Bundle bundle = message.getData();
        ...
        message.setData(bundle);

        Message message = handler.obtainMessage(2, someObject);

        handler.sendMessage(message);
        handler.sendMessageDelayed(message, 5000);
        handler.sendMessageAtTime(System.currentTimeMillis() + 5000);
        handler.sendEmptyMessage(2); // with the what only

        message.setTarget(handler);
        message.sendToTarget();
    }
}
```

Android fournit quelques méthodes statiques pour construire rapidement un message :

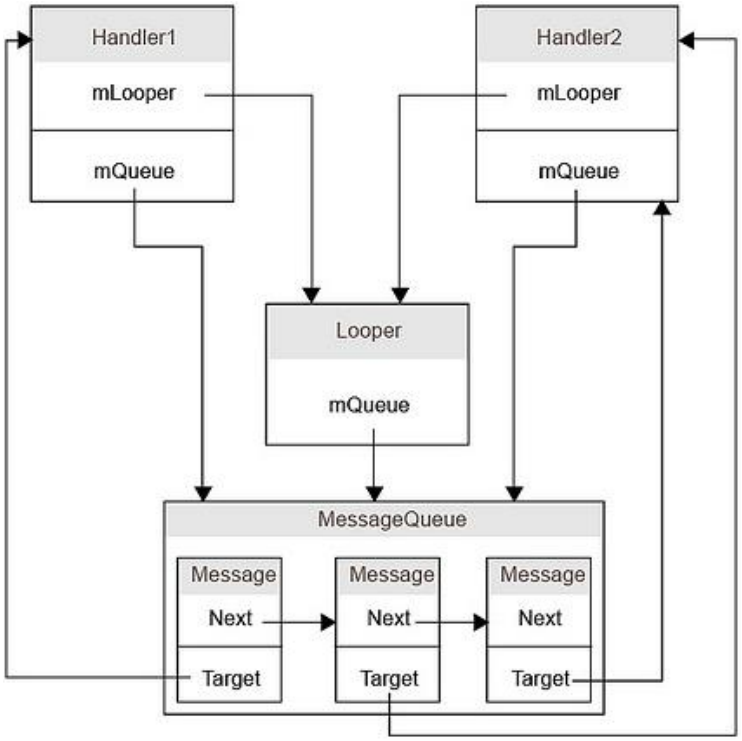
```
Message.obtain(Message orig)
Message.obtain(Handler h)
Message.obtain(Handler h, int what, Object obj)
...
```

Création explicite d'un looper

```
new Thread(new Runnable()
{
	private Handler handler;

	@Override
	public void run()
	{
		Looper.prepare(); // prepare the looper on current thread

		// the handler will automatically bind to the looper of the current thread
		handler = new Handler()
		{
			@Override
			public void handleMessage(Message inputMessage)
			{
				...
			}
		};

		Looper.loop(); // run the message loop (blocking)
	}

	public Handler getHandler()
	{
		return handler;
	}

}).start();

// meantime, anywhere, we can send tasks, messages, or stop the looper
myThread.getHandler().post(new Runnable()
{
	@Override
	public void run()
	{
		myThread.getHandler().getLooper().quit(); // or quitSafely() to process the remaining messages
	}
});
```

Accès à la message queue

```
Looper.prepare();

MessageQueue queue = Looper.myQueue();
queue.addIdleHandler(new MessageQueue.IdleHandler()
{
     @Override
     public boolean queueIdle()
     {
          Looper.myLooper().quit();
          return false;
     }
});

Looper.loop();
```



- queueIdle() est invoqué lorsque la boucle de message a terminé ses traitements.
- Retourner false dans le callback provoque sa destruction (il ne sera alors plus invoqué).

HandlerThread

- Crée automatiquement un thread muni d'un looper.
- A partir de ce looper, il est possible de créer directement un handler.

```
HandlerThread thread = new HandlerThread("myThread");
thread.start();

Looper looper = thread.getLooper();
Handler handler = new Handler(looper)
{
    @Override
    public void handleMessage(Message inputMessage)
    {
        ...
    }
};

thread.quit(); // or quitSafely()
```

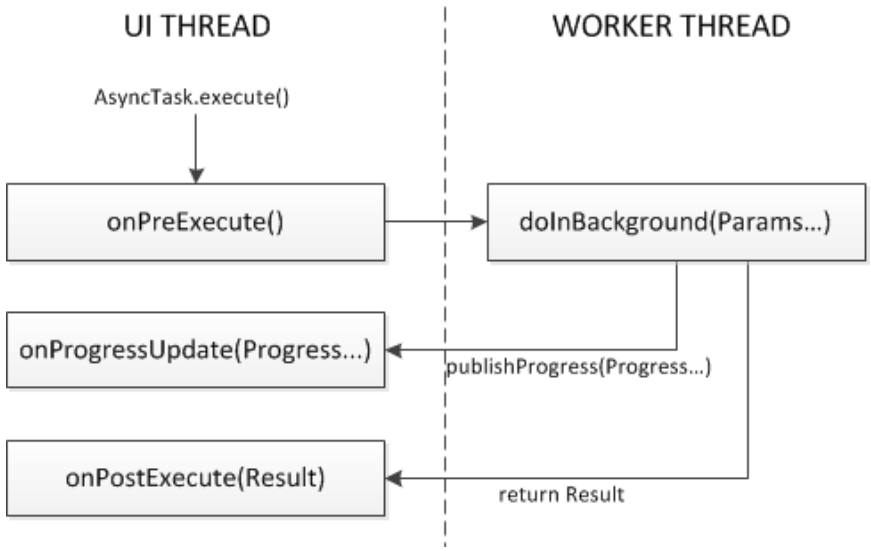
ASYNC TASK

AsyncTask

- Permet d'exécuter une tâche dans un thread (toutes les tâches sont exécutées les unes après les autres par un thread du système).
- Le résultat de la tâche est transmis à l'interface graphique sans avoir besoin d'utiliser un handler.
- Une AsyncTask admet trois paramètres :
  - Params : le type des paramètres reçus en entrée de la tâche.
  - Progress : l'unité pour la valeur de progression (e.g., Integer pour un pourcentage).
  - Result : le type du résultat produit par la tâche.

AsyncTask

- Une AsyncTask DOIT être instanciée puis démarrée dans l'UI Thread.
- Le corps de la tâche peut utiliser publishProgress() pour que l'UI mette à jour un élément visuel de progression.



- Tous les callbacks sont synchronized.
- Une tâche peut être annulée en invoquant cancel()
  - isCancelled() retourne true.
  - onCancelled(Result) est invoquée au lieu de onPostExecute, lorsque la tâche retourne un résultat.

AsyncTask

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> // Void can be used
{
     @Override
     protected Long doInBackground(URL... urls)
     {
          int count = urls.length;
          long totalSize = 0;
          for (int i = 0; i < count; i++)
          {
               totalSize += Downloader.downloadFile(urls[i]);
               publishProgress((int) ((i / (float) count) * 100));
               if (isCancelled())
                    break;
          }
          return totalSize;
     }

     @Override
     protected void onProgressUpdate(Integer... progress)
     {
          myProgressBar.setProgress(progress[0]);
     }

     @Override
     protected void onPostExecute(Long result)
     {
          ...
     }

     @Override
     protected void onCancelled(Long result)
     {
          ...
     }
}
```

AsyncTask

```
DownloadFilesTask task = new DownloadFilesTask();
task.execute(url1, url2, url3);
...
Long result = task.get() // wait until the task is completed
```

- Une instance d'AsyncTask ne peut être exécutée qu'une seule fois !
- Les AsyncTask (et tous les threads en général) peuvent être détruites lors d'un Runtime change. Le développeur doit gérer la sauvegarde/reprise des tâches.
- L'état d'une tâche est obtenu par getStatus() :
  - PENDING : la tâche n'a pas encore été exécutée.
  - RUNNING : en cours d'exécution.
  - FINISHED : l'exécution est terminée.

Thread Pool

- Basé sur le package java.util.concurrent (ExecutorService).
- Un pool est limité en nombre de thread par une limite douce (corePoolSize) et une limite forte (maxPoolSize).
- Lorsque corePoolSize est atteint, le pool peut continuer à construire des threads, mais ceux-ci sont automatiquement détruit au bout d'un certain temps d'inactivité (keepAliveTime).

Thread Pool

- Un pool utilise file d'attente pour gérer les tâches :
  1. Si corePoolSize n'est pas atteint, la file n'est pas utilisée.
  2. Si corePoolSize est dépassé, l'utilisation de la file est privilégiée.
  3. Si la requête ne peut pas être placée en file d'attente, de nouveaux threads sont crées. Si maxPoolSize est atteint, la tâche est rejetée.
- Le type de la file d'attente conditionne le comportement du pool.
  - Pas de file d'attente (SynchronousQueue), rejet si maxPoolSize est atteint.
  - File illimitée (ex : LinkedBlockingQueue sans capacité), le cas 3 ne peut donc jamais se produire.
  - File d'attente limitée (ex : ArrayBlockingQueue). Une grande file d'attente et un pool réduit permettent d'économiser les ressources, mais peuvent conduire à de fort délais.

Thread Pool

- Executors est une factory destinée à simplifier la création de pool.
  - Executors.newCachedThreadPool() : un pool illimité (maxPoolSize = Integer.MAX_VALUE, pas de file).
  - Executors.newFixedThreadPool(number) : un pool de taille limitée (corePoolSize = maxPoolSize, file illimitée).
  - Executors.newSingleThreadExecutor() : un pool avec un seul thread (corePoolSize = 1, maxPoolSize = 1, file illimitée).

Thread Pool

```
int nbCores = Runtime.getRuntime().availableProcessors();

ExecutorService myPool = Executors.newFixedThreadPool(nbCores);
// equivalent to
ExecutorService myPool = new ThreadPoolExecutor(
                                    nbCores, // corePoolSize
                                    nbCores, // maxPoolSize
                                    0l, // keepAliveTime
                                    TimeUnit.MILLISECONDS, // keepAliveTime unit
                                    new LinkedBlockingQueue<Runnable>() // queue
);

myPool.execute(new Runnable()
{
     @Override
     public void run()
     {
          ...
     }
});

AsyncTask task = ...;
task.executeOnExecutor(myPool, Params...);
task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, Params...);
task.executeOnExecutor(AsynTask.SERIAL_EXECUTOR, Params...);
```

- Le système maintient deux Executor (un ThreadPoolExecutor et un SingleThreadExecutor).
- Depuis Honeycomb, SERIAL_EXECUTOR est utilisé par défaut (pour simplifier le travail des développeurs quant aux synchronisations).

ALARM & POWER MANAGERS

Power Manager

- Android cherche à mettre le téléphone en veille le plus rapidement possible (arrêt de l'écran et mise en pause du CPU).
- Avant cela, le système vérifie s'il existe des verrous (wake locks) :
  - Si aucun wake lock n'est actif, le système met le CPU en pause et éteint l'écran.
  - Si un wake lock partiel est actif, le système éteint l'écran mais laisse le CPU actif.
  - Si un wake lock complet est actif, le système ne fait rien.

Power Manager

- Toutes les applications déclarant le droit android.permission.WAKE_LOCK dans le manifest.

```
PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
boolean isScreenOn = pm.isScreenOn();

PowerManager.WakeLock lock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "a tag for debugging purpose");
lock.acquire();
// the cpu will stay on
lock.release();
```

Flag	CPU	Ecran	Clavier
PARTIAL_WAKE_LOCK	On	Off	Off
SCREEN_DIM_WAKE_LOCK (deprecated)	*On*	*Ombré*	*Off*
SCREEN_BRIGHT_WAKE_LOCK (deprecated)	*On*	*Eclairé*	*Off*
FULL_WAKE_LOCK (deprecated)	*On*	*Eclairé*	*Eclairé*

Power Manager

- Un verrou consomme de l'énergie !
- N'utilisez un verrou que lorsque cela est parfaitement nécessaire, et sur de courte durée :
  - Acquisition du verrou.
  - Action.
  - Libération du verrou.
- Le système libère implicitement tous les verrous non partiels lorsque l'utilisateur presse le bouton Power.
- Certains composants du systèmes gèrent eux même des verrous pour simplifier certaines tâches (e.g., MediaPlayer pour la lecture audio, AlarmManager, etc.).

Power Manager

- Tous les verrous non partiels sont dépreciés.
- Google suggère d'utiliser plutôt l'option FLAG_KEEP_SCREEN_ON (ou android:keepScreenOn sur n'importe quelle vue) du WindowManager.
  - Ne nécessite pas de droit spécifique.
  - N'est valide que lorsque l'application est visible.

```
protected void onCreate(Bundle savedInstanceState)
{
    ...
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON); // add the flag
    getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON); // remove the flag
    ...
}
```

Alarm Manager

- Permet d'exécuter du code à un instant donné (ou répéter une action), même si l'application n'est pas active.
- Une alarme se présente sous la forme d'un broadcast.
- Deux types de temps :
  - Temps écoulé depuis le démarrage : SystemClock.elapsedRealtime().
  - Temps absolu (wall clock) : System.currentTimeMillis().
- Deux modes :
  - Pas de réveil : si le périphérique est en veille, l'alarme ne sera délivrée que lors du réveil.
  - Réveil : le device est réveillé pour que l'alarme soit délivrée.

Alarm Manager

- Une alarme programmée dans le passé est instantannément délivrée.
- Pour les alarmes répétées en mode sans réveil, il n'y a pas de décalage (ex : une alarme programmée toutes les heures sur un téléphone en veille de 7h45 à 8h20 sera délivrée à 8h20 puis naturellement à 9h00).
- Pendant l'exécution du onReceive() sur le receiver, l'Alarm Manager maintient un wake lock.
  - Le système peut se rendormir instantannément après. Ainsi un service démarré lors d'une alarme devrait gérer son propre wave lock pour être initialisé correctement.

Alarm Manager

The way you would check to see if it is active is to:

boolean alarmUp = (PendingIntent.getBroadcast(context, 0, new Intent("com.my.package.MY_UNIQUE_ACTION"), PendingIntent.FLAG_NO_CREATE) != null

The key here is the FLAG_NO_CREATE which as described in the javadoc: if the described PendingIntent does not already exist, then simply return null inst

Alarm Manager

```
protected void onCreate(Bundle savedInstanceState)
{
	...
	br = new BroadcastReceiver()
	{
		@Override
		public void onReceive(Context c, Intent i)
		{
			...
		}
	};
	registerReceiver(br, new IntentFilter("fr.inria.alarm"));

	alarmMgr = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
	PendingIntent pIntent = PendingIntent.getBroadcast(this, 42, new Intent("fr.inria.alarm"), 0);

	long triggeredAt = SystemClock.elapsedRealtime() + 20000;
	alarmMgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, triggeredAt , pIntent);
	alarmMgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP, triggeredAt, 60000, pIntent);
}

protected void onDestroy()
{
	alarmMgr.cancel(pi);
	unregisterReceiver(br);
	super.onDestroy();
}
```

Une alarme peut-être répétée avec “inexactitude”, c’est à dire sans garantie forte sur l’intervalle de répétion (celui-ci peut être plus grand que désiré). Le système peut alors grouper certains réveils pour économiser de l’énergie.

```
alarMgr.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP, triggeredAt, 60000, pIntent);
```

PETITE ASTUCE

- Quand le système passe en veille, les threads sont bloqués eux aussi. Un hack permet alors de se synchroniser sur les alarmes d'autres applications.

```
new Thread(new Runnable()
{
     @Override
     public void run()
     {
          while (Thread.interrupted() == false)
          {
               doSomething();
               Thread.sleep(5000);
          }
     }
}).start();
```