Développement Android (4.3)

Module 07 – Réseau & Multithreading

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/

Réseau

- Ne pas oublier les droits spécifiques.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

- Il est possible de vérifier l'état du réseau ainsi que le type des connexions actives.

```
ConnectivityManager connMgr = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);

NetworkInfo networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConnected = networkInfo.isConnected();

networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConnected = networkInfo.isConnected();

networkInfo = connMgr.getActiveNetworkInfo();
boolean isOnline = (networkInfo != null && networkInfo.isConnected());
```

HTTP

- La classe Java HttpURLConnection.

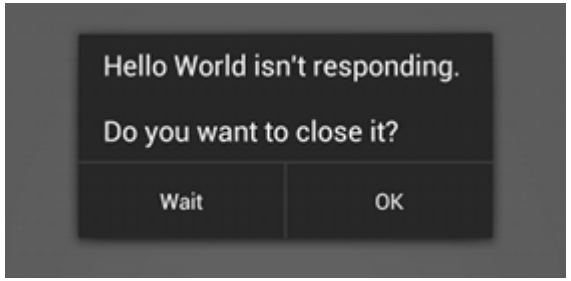
```
private class DownloadImageTask extends AsyncTask<String, Void, Bitmap>
{
     @Override
     protected Bitmap doInBackground(String... urls)
     {
          URL url = new URL(urls[0]);
          HttpURLConnection conn = (HttpURLConnection) url.openConnection();
          conn.setReadTimeout(10000);
          conn.setConnectTimeout(15000);
          conn.setRequestMethod("GET");
          conn.setDoInput(true);
          conn.connect();

          int response = conn.getResponseCode();
          InputStream is = conn.getInputStream();
          Bitmap bitmap = BitmapFactory.decodeStream(is);
          is.close();

          return bitmap;
     }

     @Override
     protected void onPostExecute(Bitmap result)
     {
          ImageView imageView = (ImageView) findViewById(R.id.image_view);
          imageView.setImageBitmap(result);
     }
}
```

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.
- Certaines directives du manifest permettent d'éxecuter les composants dans un processus différent.

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

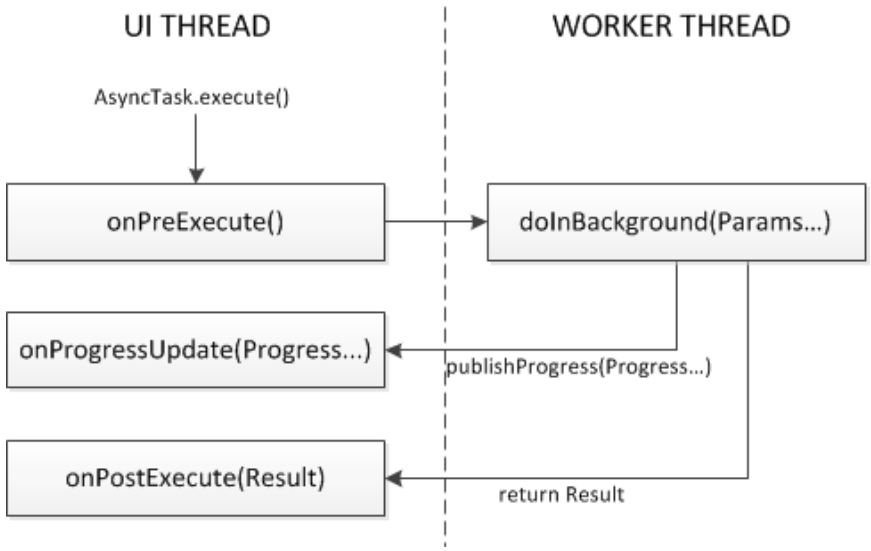
- Android supporte la classe Thread, pour laquelle il est nécessaire de gérer :
  - La synchronisation et la communication avec le main thread.
  - Les interruptions.
  - Les pools.
- Le package java.util.concurrent est aussi entièrement supporté (Java 6).

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).

UN RETOUR SUR LES SOCKET

- Le classique package java.net.

```
BlockingQueue<Runnable> queue = new LinkedBlockingQueue<Runnable>();
ThreadPoolExecutor pool = new ThreadPoolExecutor(20, 100, 5L, TimeUnit.SECONDS, queue);

ServerSocket server = new ServerSocket(8080);
while(shouldStop() == false)
{
     Socket client = server.accept(); // blocking
     pool.execute(new ProcessingTask(client));
}
server.close();

private static class ProcessingTask implements Runnable
{
     private Socket sock;
     public ProcessingTask(final Socket sock)
     {
          this.sock = sock;
     }

     @Override
     public void run()
     {
          BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
          PrintStream out = new PrintStream(sock.getOutputStream());
          message = in.readLine(); // blocking
          out.println(message);
          sock.close();
     }
}
```

Un retour sur les socket

- Le classique package java.net.

```
Socket socket = new Socket("localhost", 8080);
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

out.println("Write something"); // send request

// get response
String tmp;
while((tmp = in.readLine()) != null)
{
     ...
}

socket.close();
```

Un retour sur les socket

- Socket non bloquantes avec java.nio.

```
Selector selector = Selector.open();
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
serverChannel.socket().bind(new InetSocketAddress(8080));
serverChannel.register(selector, SelectionKey.OP_ACCEPT);

while (shouldStop() == false)
{
     selector.select(); // blocking (selectNow is non-blocking)
     Iterator<SelectionKey> it = selector.selectedKeys().iterator();
     while (it.hasNext())
     {
          SelectionKey key = (SelectionKey) it.next();
          it.remove();

          processKey(key);
     }
}

selector.close();
serverChannel.close();
```

Un retour sur les socket

```
public void processKey(SelectorKey key)
{
     if (key.isAcceptable())
     {
          ServerSocketChannel server = (ServerSocketChannel) key.channel(); // = serverChannel
          SocketChannel client = server.accept();
          client.configureBlocking(false);
          client.register(selector, SelectionKey.OP_READ);
     }
     else if (key.isReadable())
     {
          SocketChannel client = (SocketChannel) key.channel();
          StringBuilder data = new StringBuilder();
          ByteBuffer buffer = ByteBuffer.allocate(1024);

          int len = client.read(buffer);               // 0 = nothing to read, should continue
          while (len > 0 || buffer.remaining() == 0)   // < 0 = end of transmission, should close
          {                                            // > 0 = data read
               buffer.flip();
               data.append(Charset.forName("UTF8").newDecoder().decode(buffer).array());
               buffer.clear();
               len = client.read(buffer);
          }

          if (len < 0)
          {
               key.cancel();
               client.close();
          }
          ...
     }
}
```

Un retour sur les socket

- Socket non bloquantes avec java.nio.

```
SocketChannel client = SocketChannel.open();
client.configureBlocking(false);
client.connect(new InetSocketAddress("localhost", 8080));

Selector selector = Selector.open();
client.register(selector, SelectionKey.OP_CONNECT);

while (selector.select(500) > 0)
{
     Iterator<SelectionKey> it = selector.selectedKeys().iterator();
     while (it.hasNext())
     {
          SelectionKey key = (SelectionKey) it.next();
          it.remove();

          if (key.isConnectable())
          {
               SocketChannel channel = (SocketChannel) key.channel();
               if (channel.isConnectionPending())
                    channel.finishConnect();

               ByteBuffer buffer = ByteBuffer.wrap("Some Text".getBytes());
               channel.write(buffer);
               ...
          }
     }
}
```