Android Developers Blog

19 JULY 2010

Multithreading For Performance

[This post is by Gilles Debunne, an engineer in the Android group who loves to get multitasked. — Tim Bray]

A good practice in creating responsive applications is to make sure your main UI thread does the minimum amount of work. Any potentially long task that may hang your application should be handled in a different thread. Typical examples of such tasks are network operations, which involve unpredictable delays. Users will tolerate some pauses, especially if you provide feedback that something is in progress, but a frozen application gives them no clue.

In this article, we will create a simple image downloader that illustrates this pattern. We will populate a ListView with thumbnail images downloaded from the internet. Creating an asynchronous task that downloads in the background will keep our application fast.

An Image downloader

Downloading an image from the web is fairly simple, using the HTTP-related classes provided by the framework. Here is a possible implementation:

```
static Bitmap downloadBitmap(String url) {
    final AndroidHttpClient client = AndroidHttpClient.newInstance("Android");
    final HttpGet getRequest = new HttpGet(url);

    try {
        HttpResponse response = client.execute(getRequest);
        final int statusCode = response.getStatusLine().getStatusCode();
        if (statusCode != HttpStatus.SC_OK) {
            Log.w("ImageDownloader", "Error " + statusCode + " while retrieving bitmap from " +
url);
            return null;
        }

        final HttpEntity entity = response.getEntity();
        if (entity != null) {
            InputStream inputStream = null;
            try {
                inputStream = entity.getContent();
                final Bitmap bitmap = BitmapFactory.decodeStream(inputStream);
                return bitmap;
            } finally {
                if (inputStream != null) {
                    inputStream.close();
                }
                entity.consumeContent();
```

```
            }
        }
    } catch (Exception e) {
        // Could provide a more explicit error message for IOException or IllegalStateException
        getRequest.abort();
        Log.w("ImageDownloader", "Error while retrieving bitmap from " + url, e.toString());
    } finally {
        if (client != null) {
            client.close();
        }
    }
    return null;
}
```

A client and an HTTP request are created. If the request succeeds, the response entity stream containing the image is decoded to create the resulting Bitmap. Your applications' manifest must ask for the `INTERNET` to make this possible.

Note: a bug in the previous versions of `BitmapFactory.decodeStream` may prevent this code from working over a slow connection. Decode a new `FlushedInputStream(inputStream)` instead to fix the problem. Here is the implementation of this helper class:

```
static class FlushedInputStream extends FilterInputStream {
    public FlushedInputStream(InputStream inputStream) {
        super(inputStream);
    }

    @Override
    public long skip(long n) throws IOException {
        long totalBytesSkipped = 0L;
        while (totalBytesSkipped < n) {
            long bytesSkipped = in.skip(n - totalBytesSkipped);
            if (bytesSkipped == 0L) {
                  int byte = read();
                  if (byte < 0) {
                      break;  // we reached EOF
                  } else {
                      bytesSkipped = 1; // we read one byte
                  }
               }
            totalBytesSkipped += bytesSkipped;
        }
        return totalBytesSkipped;
    }
}
```

This ensures that skip() actually skips the provided number of bytes, unless we reach the end of file.

If you were to directly use this method in your ListAdapter's getView method, the resulting scrolling would be unpleasantly jaggy. Each display of a new view has to wait for an image download, which prevents smooth scrolling.

Indeed, this is such a bad idea that the AndroidHttpClient does not allow itself to be started from the main thread. The above code will display "This thread forbids HTTP requests" error messages instead. Use the DefaultHttpClient instead if you really want to shoot yourself in the foot.

Introducing asynchronous tasks

The `AsyncTask` class provides one of the simplest ways to fire off a new task from the UI thread. Let's create an `ImageDownloader` class which will be in charge of creating these tasks. It will provide a `download` method which will assign an image downloaded from its URL to an `ImageView`:

```
public class ImageDownloader {

    public void download(String url, ImageView imageView) {
            BitmapDownloaderTask task = new BitmapDownloaderTask(imageView);
            task.execute(url);
        }
    }

    /* class BitmapDownloaderTask, see below */
}
```

The `BitmapDownloaderTask` is the AsyncTask which will actually download the image. It is started using `execute`, which returns immediately hence making this method really fast which is the whole purpose since it will be called from the UI thread. Here is the implementation of this class:

```
class BitmapDownloaderTask extends AsyncTask<String, Void, Bitmap> {
    private String url;
    private final WeakReference<ImageView> imageViewReference;

    public BitmapDownloaderTask(ImageView imageView) {
        imageViewReference = new WeakReference<ImageView>(imageView);
```

```
    }

    @Override
    // Actual download method, run in the task thread
    protected Bitmap doInBackground(String... params) {
         // params comes from the execute() call: params[0] is the url.
         return downloadBitmap(params[0]);
    }

    @Override
    // Once the image is downloaded, associates it to the imageView
    protected void onPostExecute(Bitmap bitmap) {
        if (isCancelled()) {
            bitmap = null;
        }

        if (imageViewReference != null) {
            ImageView imageView = imageViewReference.get();
            if (imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

The `doInBackground` method is the one which is actually run in its own process by the task. It simply uses the `downloadBitmap` method we implemented at the beginning of this article.

`onPostExecute` is run in the calling UI thread when the task is finished. It takes the resulting Bitmap as a parameter, which is simply associated with the imageView that was provided to `download` and was stored in the `BitmapDownloaderTask`. Note that this ImageView is stored as a `WeakReference`, so that a download in progress does not prevent a killed activity's ImageView from being garbage collected. This explains why we have to check that both the weak reference and the `imageView` are not null (i.e. were not collected) before using them in `onPostExecute`.

This simplified example illustrates the use on an `AsyncTask`, and if you try it, you'll see that these few lines of code actually dramatically improved the performance of the ListView which now scrolls smoothly. Read Painless threading for more details on AsyncTasks.

However, a ListView-specific behavior reveals a problem with our current implementation. Indeed, for memory efficiency reasons, ListView *recycles* the views that are displayed when the user scrolls. If one flings the list, a given ImageView object will be used many times. Each time it is displayed the ImageView correctly triggers an image download task, which will eventually change its image. So where is the problem? As with most parallel applications, the key issue is in the ordering. In our case, there's no guarantee that the download tasks will finish in the order in which they were started. The result is that the image finally displayed in the list may come from a previous item, which simply happened to have taken longer to download. This is not an issue if the images you download are bound once and for all to given ImageViews, but let's fix it for the common case where they are used in a list.

Handling concurrency

To solve this issue, we should remember the order of the downloads, so that the last started one is the one that will effectively be displayed. It is indeed sufficient for each ImageView to remember its last download. We will add this extra information in the ImageView using a dedicated Drawable subclass, which will be temporarily bind to the ImageView while the download is in progress. Here is the code of our `DownloadedDrawable` class:

```
static class DownloadedDrawable extends ColorDrawable {
    private final WeakReference<BitmapDownloaderTask> bitmapDownloaderTaskReference;

    public DownloadedDrawable(BitmapDownloaderTask bitmapDownloaderTask) {
        super(Color.BLACK);
        bitmapDownloaderTaskReference =
            new WeakReference<BitmapDownloaderTask>(bitmapDownloaderTask);
    }

    public BitmapDownloaderTask getBitmapDownloaderTask() {
        return bitmapDownloaderTaskReference.get();
    }
}
```

This implementation is backed by a `ColorDrawable`, which will result in the ImageView displaying a black background while its download is in progress. One could use a “download in progress” image instead, which would provide feedback to the user. Once again, note the use of a WeakReference to limit object dependencies.

Let's change our code to take this new class into account. First, the `download` method will now create an instance of this class and associate it with the imageView:

```
public void download(String url, ImageView imageView) {
     if (cancelPotentialDownload(url, imageView)) {
         BitmapDownloaderTask task = new BitmapDownloaderTask(imageView);
         DownloadedDrawable downloadedDrawable = new DownloadedDrawable(task);
         imageView.setImageDrawable(downloadedDrawable);
         task.execute(url, cookie);
     }
```

```
}
```

The `cancelPotentialDownload` method will stop the possible download in progress on this imageView since a new one is about to start. Note that this is not sufficient to guarantee that the newest download is always displayed, since the task may be finished, waiting in its `onPostExecute` method, which may still may be executed *after* the one of this new download.

```
private static boolean cancelPotentialDownload(String url, ImageView imageView) {
    BitmapDownloaderTask bitmapDownloaderTask = getBitmapDownloaderTask(imageView);

    if (bitmapDownloaderTask != null) {
        String bitmapUrl = bitmapDownloaderTask.url;
        if ((bitmapUrl == null) || (!bitmapUrl.equals(url))) {
            bitmapDownloaderTask.cancel(true);
        } else {
            // The same URL is already being downloaded.
            return false;
        }
    }
    return true;
}
```

`cancelPotentialDownload` uses the `cancel` method of the AsyncTask class to stop the download in progress. It returns `true` most of the time, so that the download can be started in `download`. The only reason we don't want this to happen is when a download is already in progress on the same URL in which case we let it continue. Note that with this implementation, if an ImageView is garbage collected, its associated download is not stopped. A `RecyclerListener` might be used for that.

This method uses a helper `getBitmapDownloaderTask` function, which is pretty straigthforward:

```
private static BitmapDownloaderTask getBitmapDownloaderTask(ImageView imageView) {
    if (imageView != null) {
        Drawable drawable = imageView.getDrawable();
        if (drawable instanceof DownloadedDrawable) {
            DownloadedDrawable downloadedDrawable = (DownloadedDrawable)drawable;
            return downloadedDrawable.getBitmapDownloaderTask();
        }
    }
    return null;
}
```

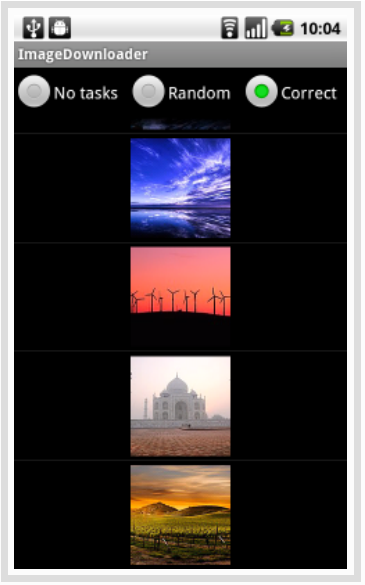
Finally, `onPostExecute` has to be modified so that it will bind the Bitmap only if this ImageView is still associated with *this* download process:

```
if (imageViewReference != null) {
    ImageView imageView = imageViewReference.get();
    BitmapDownloaderTask bitmapDownloaderTask = getBitmapDownloaderTask(imageView);
    // Change bitmap only if this process is still associated with it
    if (this == bitmapDownloaderTask) {
        imageView.setImageBitmap(bitmap);
    }
}
```

With these modifications, our `ImageDownloader` class provides the basic services we expect from it. Feel free to use it or the asynchronous pattern it illustrates in your applications to ensure their responsiveness.

Demo

The source code of this article is available online on Google Code. You can switch between and compare the three different implementations that are described in this article (no asynchronous task, no bitmap to task association and the final correct version). Note that the cache size has been limited to 10 images to better demonstrate the issues.



Future work

This code was simplified to focus on its parallel aspects and many useful features are missing from our implementation. The `ImageDownloader` class would first clearly benefit from a cache, especially if it is used in conjuction with a ListView, which will probably display the same image many times as the user scrolls back and forth. This can easily be implemented using a Least Recently Used cache backed by a `LinkedHashMap` of URL to Bitmap `SoftReferences`. More involved cache mechanism could also rely on a local disk storage of the image. Thumbnails creation and image resizing could also be added if needed.

Download errors and time-outs are correctly handled by our implementation, which will return a `null` Bitmap in these case. One may want to display an error image instead.

Our HTTP request is pretty simple. One may want to add parameters or cookies to the request as required by certain web sites.

The AsyncTask class used in this article is a really convenient and easy way to defer some work from the UI thread. You may want to use the `Handler` class to have a finer control on what you do, such as controlling the total number of download threads which are running in parallel in this case.

Posted by Unknown at 11:41 AM
Labels: App Components, Performance, threading

Links to this post

Create a Link

Newer Post Home Older Post