进程和线程

本文内容

当某个应用组件启动且该应用没有运行其他任何组件时，Android 系统会使用单个执行线程为应用启动新的 Linux 进程。默认情况下，同一应用的所有组件在相同的进程和线程（称为“主”线程）中运行。 如果某个应用组件启动且该应用已存在进程（因为存在该应用的其他组件），则该组件会在此进程内启动并使用相同的执行线程。 但是，您可以安排应用中的其他组件在单独的进程中运行，并为任何进程创建额外的线程。

本文档介绍进程和线程在 Android 应用中的工作方式。

进程

默认情况下，同一应用的所有组件均在相同的进程中运行，且大多数应用都不会改变这一点。 但是，如果您发现需要控制某个组件所属的进程，则可在清单文件中执行此操作。

各类组件元素的清单文件条目—`<activity>`、`<service>`、`<receiver>` 和 `<provider>`—均支持 `android:process` 属性，此属性可以指定该组件应在哪个进程运行。您可以设置此属性，使每个组件均在各自的进程中运行，或者使一些组件共享一个进程，而其他组件则不共享。 此外，您还可以设置 `android:process`，使不同应用的组件在相同的进程中运行，但前提是这些应用共享相同的 Linux 用户 ID 并使用相同的证书进行签署。

此外，`<application>` 元素还支持 `android:process` 属性，以设置适用于所有组件的默认值。

如果内存不足，而其他为用户提供更紧急服务的进程又需要内存时，Android 可能会决定在某一时刻关闭某一进程。在被终止进程中运行的应用组件也会随之销毁。 当这些组件需要再次运行时，系统将为它们重启进程。

决定终止哪个进程时，Android 系统将权衡它们对用户的相对重要程度。例如，相对于托管可见 Activity 的进程而言，它更有可能关闭托管屏幕上不再可见的 Activity 的进程。 因此，是否终止某个进程的决定取决于该进程中所运行组件的状态。 下面，我们介绍决定终止进程所用的规则。

进程生命周期

Android 系统将尽量长时间地保持应用进程，但为了新建进程或运行更重要的进程，最终需要移除旧进程来回收内存。 为了确定保留或终止哪些进程，系统会根据进程中正在运行的组件以及这些组件的状态，将每个进程放入“重要性层次结构”中。 必要时，系统会首先消除重要性最低的进程，然后是重要性略逊的进程，依此类推，以回收系统资源。

重要性层次结构一共有 5 级。以下列表按照重要程度列出了各类进程（第一个进程*最重要*，将是*最后一个被终止*的进程）：

1. **前台进程**

   用户当前操作所必需的进程。如果一个进程满足以下任一条件，即视为前台进程：

   - 托管用户正在交互的 `Activity`（已调用 `Activity` 的 `onResume()` 方法）
   - 托管某个 `Service`，后者绑定到用户正在交互的 Activity
   - 托管正在“前台”运行的 `Service`（服务已调用 `startForeground()`）

- 托管正执行一个生命周期回调的 `Service`（`onCreate()`、`onStart()` 或 `onDestroy()`）
- 托管正执行其 `onReceive()` 方法的 `BroadcastReceiver`

通常，在任意给定时间前台进程都为数不多。只有在内存不足以支持它们同时继续运行这一万不得已的情况下，系统才会终止它们。 此时，设备往往已达到内存分页状态，因此需要终止一些前台进程来确保用户界面正常响应。

2. **可见进程**

   没有任何前台组件、但仍会影响用户在屏幕上所见内容的进程。 如果一个进程满足以下任一条件，即视为可见进程：

   - 托管不在前台、但仍对用户可见的 `Activity`（已调用其 `onPause()` 方法）。例如，如果前台 Activity 启动了一个对话框，允许在其后显示上一 Activity，则有可能会发生这种情况。
   - 托管绑定到可见（或前台）Activity 的 `Service`。

   可见进程被视为是极其重要的进程，除非为了维持所有前台进程同时运行而必须终止，否则系统不会终止这些进程。

3. **服务进程**

   正在运行已使用 `startService()` 方法启动的服务且不属于上述两个更高类别进程的进程。尽管服务进程与用户所见内容没有直接关联，但是它们通常在执行一些用户关心的操作（例如，在后台播放音乐或从网络下载数据）。因此，除非内存不足以维持所有前台进程和可见进程同时运行，否则系统会让服务进程保持运行状态。

4. **后台进程**

   包含目前对用户不可见的 Activity 的进程（已调用 Activity 的 `onStop()` 方法）。这些进程对用户体验没有直接影响，系统可能随时终止它们，以回收内存供前台进程、可见进程或服务进程使用。 通常会有很多后台进程在运行，因此它们会保存在 LRU （最近最少使用）列表中，以确保包含用户最近查看的 Activity 的进程最后一个被终止。如果某个 Activity 正确实现了生命周期方法，并保存了其当前状态，则终止其进程不会对用户体验产生明显影响，因为当用户导航回该 Activity 时，Activity 会恢复其所有可见状态。 有关保存和恢复状态的信息，请参阅 Activity文档。

5. **空进程**

   不含任何活动应用组件的进程。保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。 为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。

根据进程中当前活动组件的重要程度，Android 会将进程评定为它可能达到的最高级别。例如，如果某进程托管着服务和可见 Activity，则会将此进程评定为可见进程，而不是服务进程。

此外，一个进程的级别可能会因其他进程对它的依赖而有所提高，即服务于另一进程的进程其级别永远不会低于其所服务的进程。 例如，如果进程 A 中的内容提供程序为进程 B 中的客户端提供服务，或者如果进程 A 中的服务绑定到进程 B 中的组件，则进程 A 始终被视为至少与进程 B 同样重要。

由于运行服务的进程其级别高于托管后台 Activity 的进程，因此启动长时间运行操作的 Activity 最好为该操作启动服务，而不是简单地创建工作线程，当操作有可能比 Activity 更加持久时尤要如此。例如，正在将图片上传到网站的 Activity 应该启动服务来执行上传，这样一来，即使用户退出 Activity，仍可在后台继续执行上传操作。使用服务可以保证，无论 Activity 发生什么情况，该操作至少具备“服务进程”优先级。 同理，广播接收器也应使用服务，而不是简单地将耗时冗长的操作放入线程中。

线程

应用启动时，系统会为应用创建一个名为“主线程”的执行线程。 此线程非常重要，因为它负责将事件分派给相应的用户界面小部件，其中包括绘图事件。 此外，它也是应用与 Android UI 工具包组件（来自 `android.widget` 和 `android.view` 软件包的组件）进行交互的线程。因此，主线程有时也称为 UI 线程。

系统*不会*为每个组件实例创建单独的线程。运行于同一进程的所有组件均在 UI 线程中实例化，并且对每个组件的系统调用均由该线程进行分派。 因此，响应系统回调的方法（例如，报告用户操作的 `onKeyDown()` 或生命周期回调方法）始终在进程的 UI 线程中运行。

例如，当用户触摸屏幕上的按钮时，应用的 UI 线程会将触摸事件分派给小部件，而小部件反过来又设置其按下状态，并将失效请求发布到事件队列中。 UI 线程从队列中取消该请求并通知小部件应该重绘自身。

在应用执行繁重的任务以响应用户交互时，除非正确实现应用，否则这种单线程模式可能会导致性能低下。 具体地讲，如果 UI 线程需要处理所有任务，则执行耗时很长的操作（例如，网络访问或数据库查询）将会阻塞整个 UI。 一旦线程被阻塞，将无法分派任何事件，包括绘图事件。 从用户的角度来看，应用显示为挂起。 更糟糕的是，如果 UI 线程被阻塞超过几秒钟时间（目前大约是 5 秒钟），用户就会看到一个让人

厌烦的“应用无响应”(ANR) 对话框。如果引起用户不满，他们可能就会决定退出并卸载此应用。

此外，Android UI 工具包*并非*线程安全工具包。因此，您不得通过工作线程操纵 UI，而只能通过 UI 线程操纵用户界面。 因此，Android 的单线程模式必须遵守两条规则：

1. 不要阻塞 UI 线程

2. 不要在 UI 线程之外访问 Android UI 工具包

工作线程

根据上述单线程模式，要保证应用 UI 的响应能力，关键是不能阻塞 UI 线程。 如果执行的操作不能很快完成，则应确保它们在单独的线程（“后台”或“工作”线程）中运行。

例如，以下代码演示了一个点击侦听器从单独的线程下载图像并将其显示在 `ImageView` 中：

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");
            mImageView.setImageBitmap(b);
        }
    }).start();
}
```

乍看起来，这段代码似乎运行良好，因为它创建了一个新线程来处理网络操作。 但是，它违反了单线程模式的第二条规则：*不要在 UI 线程之外访问 Android UI 工具包* — 此示例从工作线程（而不是 UI 线程）修改了 `ImageView`。 这可能导致出现不明确、不可预见的行为，但要跟踪此行为困难而又费时。

为解决此问题，Android 提供了几种途径来从其他线程访问 UI 线程。 以下列出了几种有用的方法：

- `Activity.runOnUiThread(Runnable)`
- `View.post(Runnable)`
- `View.postDelayed(Runnable, long)`

例如，您可以通过使用 `View.post(Runnable)` 方法修复上述代码：

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap =
                    loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

现在，上述实现属于线程安全型：在单独的线程中完成网络操作，而在 UI 线程中操纵 `ImageView`。

但是，随着操作日趋复杂，这类代码也会变得复杂且难以维护。 要通过工作线程处理更复杂的交互，可以考虑在工作线程中使用 `Handler` 处理来自 UI 线程的消息。当然，最好的解决方案或许是扩展 `AsyncTask` 类，此类简化了与 UI 进行交互所需执行的工作线程任务。

使用 AsyncTask

`AsyncTask` 允许对用户界面执行异步操作。 它会先阻塞工作线程中的操作，然后在 UI 线程中发布结果，而无需您亲自处理线程和/或处理程序。

要使用它，必须创建 `AsyncTask` 的子类并实现 `doInBackground()` 回调方法，该方法将在后台线程池中运行。 要更新 UI，应该实现 `onPostExecute()` 以传递 `doInBackground()` 返回的结果并在 UI 线程中运行，以便您安全地更新 UI。 稍后，您可以通过从 UI 线程调用 `execute()` 来运行任务。

例如，您可以通过以下方式使用 `AsyncTask` 来实现上述示例：

```
public void onClick(View v) {
    new DownloadImageTask().execute("http://example.com/image.png");
}

private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
    /** The system calls this to perform work in a worker thread and
      * delivers it the parameters given to AsyncTask.execute() */
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    /** The system calls this to perform work in the UI thread and delivers
      * the result from doInBackground() */
    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
```

现在 UI 是安全的，代码也得到简化，因为任务分解成了两部分：一部分应在工作线程内完成，另一部分应在 UI 线程内完成。

下面简要概述了 AsyncTask 的工作方法，但要全面了解如何使用此类，您应阅读 `AsyncTask` 参考文档：

- 可以使用泛型指定参数类型、进度值和任务最终值
- 方法 `doInBackground()` 会在工作线程上自动执行
- `onPreExecute()`、`onPostExecute()` 和 `onProgressUpdate()` 均在 UI 线程中调用
- `doInBackground()` 返回的值将发送到 `onPostExecute()`
- 您可以随时在 `doInBackground()` 中调用`publishProgress()`，以在 UI 线程中执行 `onProgressUpdate()`
- 您可以随时取消任何线程中的任务

> **注意**：使用工作线程时可能会遇到另一个问题，即：运行时配置变更（例如，用户更改了屏幕方向）导致 Activity 意外重启，这可能会销毁工作线程。 要了解如何在这种重启情况下坚持执行任务，以及如何在 Activity 被销毁时正确地取消任务，请参阅书架示例应用的源代码。

线程安全方法

在某些情况下，您实现的方法可能会从多个线程调用，因此编写这些方法时必须确保其满足线程安全的要求。

这一点主要适用于可以远程调用的方法，如绑定服务中的方法。如果对 `IBinder` 中所实现方法的调用源自运行 `IBinder` 的同一进程，则该方法在调用方的线程中执行。但是，如果调用源自其他进程，则该方法将在从线程池选择的某个线程中执行（而不是在进程的 UI 线程中执行），线程池由系统在与 `IBinder` 相同的进程中维护。 例如，即使服务的 `onBind()` 方法将从服务进程的 UI 线程调用，在 `onBind()` 返回的对象中实现的方法（例如，实现 RPC 方法的子类）仍会从线程池中的线程调用。 由于一个服务可以有多个客户端，因此可能会有多个池线程在同一时间使用同一 `IBinder` 方法。因此，`IBinder` 方法必须实现为线程安全方法。

同样，内容提供程序也可接收来自其他进程的数据请求。尽管 `ContentResolver` 和 `ContentProvider` 类隐藏了如何管理进程间通信的细节，但响应这些请求的 `ContentProvider` 方法（`query()`、`insert()`、`delete()`、`update()` 和 `getType()` 方法）将从内容提供程序所在进程的线程池中调用，而不是从进程的 UI 线程调用。 由于这些方法可能会同时从任意数量的线程调用，因此它们也必须实现为线程安全方法。

进程间通信

Android 利用远程过程调用 (RPC) 提供了一种进程间通信 (IPC) 机制，通过这种机制，由 Activity 或其他应用组件调用的方法将（在其他进程中）远程执行，而所有结果将返回给调用方。 这就要求把方法调用及其数据分解至操作系统可以识别的程度，并将其从本地进程和地址空间传输至远程进程和地址空间，然后在远程进程中重新组装并执行该调用。 然后，返回值将沿相反方向传输回来。 Android 提供了执行这些 IPC 事务所需的全部代码，因此您只需集中精力定义和实现 RPC 编程接口即可。

要执行 IPC，必须使用 `bindService()` 将应用绑定到服务上。如需了解详细信息，请参阅服务开发者指南。