进程和线程概览

当应用组件启动且该应用未运行任何其他组件时，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会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，且大多数应用都不应改变这一点。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（`<activity>` (/guide/topics/manifest/activity-element)、`<service>` (/guide/topics/manifest/service-element)、`<receiver>` (/guide/topics/manifest/receiver-element) 和 `<provider>` (/guide/topics/manifest/provider-element)）的清单文件条目均支持 `android:process` 属性，此属性可指定该组件应在哪个进程中运行。您可以设置此属性，使每个组件均在各自的进程中运行，或者使某些组件共享一个进程，而其他组件则不共享。您也可设置 `android:process`，以便不同应用的组件在同一进程中运行，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>` (/guide/topics/manifest/application-element) 元素还支持 `android:process` 属性，用来设置适用于所有组件的默认值。

当内存不足，而其他更急于为用户提供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个进程。正因如此，系统会销毁在被终止进程中运行的应用组件。当这些组件需再次运行时，系统将为其重启进程。

决定终止哪个进程时，Android 系统会权衡其对用户的相对重要性。例如，相较于托管可见 Activity 的进程而言，系统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。

如需详细了解进程生命周期及其与应用状态的关系，请参阅进程和应用生命周期 (/guide/topics/processes/process-lifecycle)。

线程

启动应用时，系统会为该应用创建一个称为“main”（主线程）的执行线程。此线程非常重要，因为其负责将事件分派给相应的界面微件，其中包括绘图事件。此外，应用与 Android 界面工具包组件（来自 `android.widget` (/reference/android/widget/package-summary) 和 `android.view` (/reference/android/view/package-summary) 软件包的组件）也几乎都在该线程中进行交互。因此，主线程有时也称为界面线程。但在一些特殊情况下，应用的主线程可能并非其界面线程，相关详情请参阅线程注解 (/studio/write/annotations#thread-annotations)。

系统不会为每个组件实例创建单独的线程。在同一进程中运行的所有组件均在界面线程中进行实例化，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()` (/reference/android/view/View#onKeyDown(int, android.view.KeyEvent)) 或生命周期回调方法）始终在进程的界面线程中运行。

例如，当用户轻触屏幕上的按钮时，应用的界面线程会将轻触事件分派给微件，而微件转而会设置其按下状态，并将失效请求发布到事件队列中。界面线程从队列中取消该请求，并通知该微件对其自身进行重绘。

当应用执行繁重的任务以响应用户交互时，除非您正确实现应用，否则这种单线程模式可能会导致性能低下。具体地讲，如果界面线程需要处理所有任务，则执行耗时较长的操作（例如，网络访问或数据库查询）将会阻塞整个界面线程。一旦被阻塞，线程将无法分派任何事件，包括绘图事件。从用户的角度来看，应用会显示为挂起状态。更糟糕的是，如果界面线程被阻塞超过几秒钟时间（目前大约是 5 秒钟），用户便会看到令人厌烦的“应用无响应 (https://developer.android.com/guide/practices/responsiveness.html)”(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 界面工具包并非线程安全工具包。所以您不得通过工作线程操纵界面，而只能通过界面线程操纵界面。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，如要保证应用界面的响应能力，关键是不能阻塞界面线程。如果执行的操作不能即时完成，则应确保它们在单独的线程（“后台”或“工作”线程）中运行。

但请注意，除了界面线程或“主”线程，您无法更新任何其他线程的界面。

为解决此问题，Android 提供了几种途径，以便您从其他线程访问界面线程。以下列出了几种有用的方法：

- `Activity.runOnUiThread(Runnable)` (/reference/android/app/Activity#runOnUiThread(java.lang.Runnable))
- `View.post(Runnable)` (/reference/android/view/View#post(java.lang.Runnable))
- `View.postDelayed(Runnable, long)` (/reference/android/view/View#postDelayed(java.lang.Runnable, long))

KOTLIN (#KOTLIN)JAVA

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            // a potentially time consuming task
            final Bitmap bitmap =
                    processBitMap("image.png");
            imageView.post(new Runnable() {
                public void run() {
                    imageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

上述实现属于线程安全型：在单独的线程中完成后台操作，同时始终在界面线程中操纵 `ImageView` (/reference/android/widget/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。如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` (/reference/android/os/Handler) 处理来自界面线程的消息。当然，最好的解决方案或许是扩展 `AsyncTask` (/reference/android/os/AsyncTask) 类，此类可简化与界面进行交互所需执行的工作线程任务。

使用 AsyncTask

`AsyncTask` (/reference/android/os/AsyncTask) 允许对界面执行异步操作。它会先阻塞工作线程中的操作，然后在界面线程中发布结果，而无需您亲自处理线程和/或处理程序。

如要使用该类，您必须创建 `AsyncTask` (/reference/android/os/AsyncTask) 的子类并实现 `doInBackground()` (/reference/android/os/AsyncTask#doInBackground(Params...)) 回调方法，该方法会在后台线程池中运行。如要更新界面，您应实现 `onPostExecute()` (/reference/android/os/AsyncTask#onPostExecute(Result)) （该方法会传递 `doInBackground()` (/reference/android/os/AsyncTask#doInBackground(Params...)) 返回的结果并在界面线程中运行），以便安全更新界面。然后，您可以通过从界面线程调用 `execute()` (/reference/android/os/AsyncTask#execute(Params...)) 来运行任务。

如要全面了解如何使用此类，请阅读 `AsyncTask` (/reference/android/os/AsyncTask) 参考文档。

线程安全方法

在某些情况下，系统可能会从多个线程调用您实现的方法，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 (/guide/components/bound-services)中的方法。如果对 `IBinder` (/reference/android/os/IBinder) 中所实现方法的调用源自运行 `IBinder` (/reference/android/os/IBinder) 的同一进程，则系统会在调用方的线程中执行该方法。但是，如果调用源自其他进程，则系统会选择线程池中的某个线程，并在此线程中（而不是在进程的界面线程中）执行该方法，线程池由系统在与 `IBinder` (/reference/android/os/IBinder) 相同的进程中进行维护。例如，即使服务的 `onBind()` (/reference/android/app/Service#onBind(android.content.Intent)) 方法通过服务进程的界面线程调用，在 `onBind()` (/reference/android/app/Service#onBind(android.content.Intent)) 所返回对象中实现的方法（例如，实现 RPC 方法的子类）仍会通过线程池中的线程调用。由于服务可以有多个客户端，因此多个池线程可同时使用相同的 `IBinder` (/reference/android/os/IBinder) 方法。因此，`IBinder` (/reference/android/os/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` (/reference/android/content/ContentResolver) 和 `ContentProvider` (/reference/android/content/ContentProvider) 类隐藏了如何管理进程间通信的细节，但系统会从内容提供程序进程的线程池

（而非进程的界面线程）调用响应这些请求的 `ContentProvider` (/reference/android/content/ContentProvider) 方法（`query()` (/reference/android/content/ContentProvider#query(android.net.Uri, java.lang.String[], android.os.Bundle, android.os.CancellationSignal))、`insert()` (/reference/android/content/ContentProvider#insert(android.net.Uri, android.content.ContentValues))、`delete()` (/reference/android/content/ContentProvider#delete(android.net.Uri, java.lang.String, java.lang.String[]))、`update()` (/reference/android/content/ContentProvider#update(android.net.Uri, android.content.ContentValues, java.lang.String, java.lang.String[])) 和 `getType()` (/reference/android/content/ContentProvider#getType(android.net.Uri)) 方法）。由于系统可能会同时从任意数量的线程调用这些方法，因此它们也必须实现为线程安全的方法。

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，在此机制中，系统会（在其他进程中）远程执行由 Activity 或其他应用组件调用的方法，并将所有结果返回给调用方。因此，您需将方法调用及其数据分解至操作系统可识别的程度，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。

如要执行 IPC，您必须使用 `bindService()` (/reference/android/content/Context#bindService(android.content.Intent, android.content.ServiceConnection, int)) 将应用绑定到服务。如需了解详细信息，请参阅服务 (/guide/components/services)开发者指南。

Last updated 2020-07-08 UTC.