Android 线程类

启动应用程序时，Android 会创建自己的 Linux 进程。除此系统外，它还为该应用程序创建一个执行线程，称为主线程或 UI 线程。主线程只不过是一个处理程序线程。主线程负责处理来自整个应用程序的事件，例如与生命周期信息相关的回调或来自输入事件的回调。它还可以处理来自其他应用程序的事件。

任何需要运行的代码块都被推入工作队列，然后由主线程提供服务。由于主线程做了很多工作，最好将更长的工作提供给其他线程，以免干扰 UI 线程的渲染任务。必须避免使用主线程执行任何可能最终导致其阻塞导致 ANR（应用程序无响应）的操作。

网络操作或数据库调用或某些组件的加载是一些在主线程上执行时可能导致主线程阻塞的示例。它们是同步执行的，这意味着 UI 将保持完全无响应，直到任务完成。为了避免这种情况，它们通常在单独的线程中执行，这样可以避免在执行任务时阻塞 UI。这意味着它们是从 UI 异步执行的。

Android 提供了很多创建和管理线程的方法，并且有很多第三方库可以让线程管理变得容易很多。

AsyncTask：帮助开启/关闭 UI 线程

AsyncTask 允许正确和轻松地使用 UI 线程。此类允许您执行后台操作并在 UI 线程上发布结果，而无需使用Thread或 Handler。

AsyncTask 被设计为围绕 Thread 和 Handler 的辅助类，并不构成通用线程框架。 AsyncTasks 最好用于短时间的操作（最多几秒钟）。如果您需要保持线程长时间运行，强烈建议您使用 java.util.concurrent 包提供的各种 API，例如 Executor， ThreadPoolExecutor 和 FutureTask。

当一个异步任务被执行时，任务会经过四个步骤：

1. onPreExecute()：在任务执行前在UI线程上调用。这一步通常用于在任务开始之前做一些事情——例如，简单地在用户界面中显示一个进度对话框。

2.doInBackground(Params…)：onPreExecute()执行完毕后在后台线程调用。此步骤用于执行可能需要很长时间的后台计算。异步任务的参数传递到这一步。此步骤必须返回计算结果，并将结果发送到 onPostExecute()。此步骤还可以使用 publishProgress(..) 来发布一个或多个进度单位。

3. onProgressUpdate(Progress...)：在调用 publishProgress(..) 后在 UI 线程上调用。此方法用于在后台计算仍在执行时在用户界面中显示任何形式的进度。例如，它可用于动画进度条或在文本字段中显示日志。

1. onPostExecute(Result)：后台计算完成后在UI线程上调用。后台计算的结果作为参数传递给该步骤。

可以通过调用 cancel(boolean...) 随时取消任务。我们需要通过检查任务是取消还是运行来做到这一点。

private class AsyncTaskRunner extends AsyncTask<String, String, String> {

@Override protected void onPreExecute() {

progressDialog.show();

}

@Override protected String doInBackground(String... params) { . doSomething();

publishProgress("Sleeping..."); // Calls onProgressUpdate()

return resp;

}

@Override protected void onPostExecute(String result) {

// execution of result of Long time consuming operation . progressDialog.dismiss();

updateUIWithResult() ;

}

@Override protected void onProgressUpdate(String... text) {

updateProgressUI();

}

}

AsyncTask 是快速结束需要频繁 UI 更新的短期工作的完美解决方案。

但是，如果需要延迟任务在Activity/Fragment的生命周期之外运行，则异步任务不足。值得注意的是，即使是像屏幕旋转这样简单的事情也会导致 Activity 被破坏。

默认情况下，所有创建的 AsyncTask 将共享同一个线程，并从单个消息队列中按顺序执行。同步执行会影响单个任务。

如果我们希望任务并行执行，我们可以使用 THREAD\_POOL\_EXECUTOR。

HandlerThread：回调线程

处理程序线程是普通 Java 线程类的子类。处理程序线程是一个长时间运行的线程，它从队列中获取工作并对其进行操作。它是其他 Android 原语的组合，即：

Looper：保持线程存活并持有消息队列

MessageQueue：保存由 Looper 调度的消息列表的类

Handler：允许我们发送和处理与线程的 MessageQueue 关联的消息对象

这意味着我们可以让它在后台运行，并一个接一个地为它提供越来越多的工作包，直到我们退出它。 HandlerThreads 在Activity生命周期之外运行，因此需要正确清理它们，否则将遇到线程泄漏。

创建处理程序线程有两种主要方法。

创建一个新的处理程序线程，并获取looper。现在，通过分配创建的处理程序线程的循环器来创建一个新的处理程序，并将您的任务发布到这个处理程序上。

HandlerThread handlerThread = new HandlerThread("TesHandlerThread");

handlerThread.start();

Looper looper = handlerThread.getLooper();

Handler handler = new Handler(looper);

handler.post(new Runnable(){…});

通过创建 CustomHandlerThread 类来扩展处理程序线程。然后，创建一个处理程序来处理任务。如果您知道要执行的任务并且只需要传入参数，您就会采用这种方法。例如，创建一个自定义 HandlerThread 类来下载图像或执行网络任务。

当我们创建一个handler线程时，不要忘记设置它的优先级，因为CPU只能并行处理少量线程，所以设置优先级可以帮助系统在其他线程争抢的时候知道正确的调度这个工作的方法注意力。

注意：当完成后台线程或活动的 onDestroy() 方法时调用 handlerThread.quit()。

我们可以通过使用本地广播或使用主循环器创建处理程序来将更新发布到 UI 线程。

Handler mainHandler = new Handler(context.getMainLooper());

mainHandler.post(myRunnable);

处理程序线程是不需要 UI 更新的长时间运行的后台工作的完美解决方案。

ThreadPoolExecutor：运行大量并行工作

线程池基本上是等待被赋予任务的线程池。分配给这些线程的任务将并行运行。由于任务并行执行，我们可能希望确保我们的代码是线程安全的。线程池主要解决两个问题：

由于减少了每个任务的开销，在执行大量异步任务时提高了性能

执行任务集合时限制和管理资源（包括线程）的一种方法

让我们考虑一个例子：如果我们要解码 40BMP，其中每个位图需要 4 毫秒来解码，如果我们在单个线程上进行，则需要 160 毫秒以上来解码所有位图。

然而，如果我们用 10 个线程来做，每个线程解码四个位图。因此，解码这 40 个位图所需的时间仅为 16 毫秒。

这里的问题是如何将工作传递给每个线程，如何安排工作，以及如何管理这些线程。这是一个非常大的问题。这就是 ThreadPoolExecutor 发挥作用的地方。

ThreadPoolExecutor 是一个扩展 AbstractExecutorService 的类。 ThreadPoolExecutor 负责所有线程。

它将任务分配给线程

它让他们活着

它终止线程

它在后台工作的方式是将要运行的任务维护在工作队列中。只要池中的线程空闲或可用，就会从工作队列中将任务分配给线程。

Runnable

它是一个由类实现的接口，其实例旨在由线程执行。简单地说：它是一个可以执行的命令或任务。它经常用于在不同的线程中运行代码。

Runnable mRunnable = new Runnable() {

@Override

public void run() {

// Do some work

}

};

Executor

Executor 是用于将任务提交与任务执行解耦的接口。它是一个执行 Runnable 的对象。

Executor mExecutor = Executors.newSingleThreadExecutor(); mExecutor.execute(mRunnable);

ExecutorService

管理异步任务的 Executor。

ExecutorService mExecutorService = Executors.newFixedThreadPool(10); mExecutorService.execute(mRunnable);

ThreadPoolExecutor

将任务分配给线程池的 ExecutorService。

更多的线程并不总是好的，因为 CPU 只能并行执行一定数量的线程。一旦我们超过了这个数字，CPU 必须进行一些昂贵的计算来决定应该根据优先级分配哪个线程。

在创建 ThreadPoolExecutor 的实例时，我们可以指定初始线程数和最大线程数。随着线程池中的工作负载发生变化，它会缩放活动线程的数量以匹配。

通常建议根据可用内核数分配线程。这可以通过以下方式实现：

int NUMBER\_OF\_CORES = Runtime.getRuntime().availableProcessors();

注意：这不一定会返回设备上的实际物理内核数。 CPU可能会停用某些内核以节省电池等。

ThreadPoolExecutor(

int corePoolSize, // Initial pool size

int maximumPoolSize, // Max pool size

long keepAliveTime, // Time idle thread waits before terminating

TimeUnit unit // Sets the Time Unit for keepAliveTime

BlockingQueue<Runnable> workQueue) // Work Queue

corePoolSize：保留在池中的最小线程数。最初，池中有零个线程。但是随着任务被添加到队列中，新的线程被创建。如果运行的线程少于 corePoolSize，则 Executor 总是更喜欢添加新线程而不是排队。

maximumPoolSize：池中允许的最大线程数。如果这超过了 corePoolSize 并且当前线程数 >= corePoolSize，那么只有当队列已满时才会创建新的工作线程。

keepAliveTime：当线程数大于核心时，非核心线程（多余的空闲线程）会等待一个新的任务，如果在这个参数定义的时间内没有得到一个，就会终止。

unit：keepAliveTime 的时间单位。

workQueue：任务队列，只会存放可运行的任务。它必须是一个 BlockingQueue。

ThreadPoolExecutor 是一个强大的任务执行框架，我们可以在需要并行执行大量任务时使用——因为它支持在队列中添加任务、取消任务和确定任务优先级。

IntentService：帮助从 UI 线程中获取意图

IntentService 是继承自 Service 的子类。要了解 IntentService，我们需要了解 Service。

Service是Android编程中非常重要的组件。有时，即使在应用程序关闭后，我们也可能有要执行的任务。这是服务将更有帮助的场景。 Service可以通过startService()/stopService()调用和取消，并在后台长时间运行。它也可以通过在其中调用 stopSelf() 来取消。

让我们看看一些有助于执行操作的重写方法

onCreate()：只会被调用一次，直到它停止。

onStartCommand()：该函数会在第一次onCreate()之后调用，但是第二次以后任何有意图的组件调用startService()都会直接调用。

onDestroy()：当我们停止服务时调用它。

IntentService 的启动方式与普通 Service 相同（通过从主线程调用 startService()）。它处理 onHandleIntent() 中的每个意图，而不是 onStartCommand()。它使用一个工作线程并在工作结束时自行停止。要使用它，我们需要扩展 IntentService 并实现 onHandleIntent()。

注意：IntentService 运行在单个工作线程上，而 Service 运行在主线程上。一次只会处理一个请求。

IntentService 受 Android 8.0（API 级别 26）施加的所有后台执行限制的约束。在大多数情况下，您最好使用 JobIntentService，它在 Android 8.0 或更高版本上运行时使用作业而不是服务。

IntentService 按需处理异步请求。如果不要求服务同时处理多个请求，这是最佳选择。