Service基础知识

原创 xyzso1z 2020-08-19 11:09:05 1188 收藏 1 编辑 版权

分类专栏： Android 文章标签： android service service生命周期

一、简介

`Service` 是一种可在后台执行长时间运行操作而不提供界面的应用组件。 `Service` 可由其他应用组件启动，而且即使用户切换到其他应用，服务仍将在后台继续运行。此外，组件可通过绑定到 `Service` 与之进行交互，甚至执行进程间通信(IPC)。例如，服务可在后台处理网络事物、播放音乐，执行文件I/O或内容提供程序进行交互。

以下三种不同的服务类型：

1. `Foreground`
   前台服务执行一些用户**能注意到**的操作。例如，音频应用汇使用前台服务来播放音频曲目。前台服务必须显示通知。即使用户停止与应用的交互，前台服务仍会继续运行。

2. `Background`
   后台服务执行用户**不会直接注意到**的操作。例如，如果应用使用某个服务来压缩其存储空间，则此服务通常是后台服务。

   > 注: 如果应用面向API级别26或更高版本，当应用本身未在前台运行时，系统会对运行后台服务施加限制(该内容会在最后进行说明)。

3. `Bound`
   当应用组件通过调用 `bindService()` 绑定到服务时，服务即处于绑定状态。绑定服务会提供 `client-server` 接口，以便服务进行交互、发送请求、接收结果，甚至是利用进程通信(IPC)跨进程执行这些操作。仅当与另一个应用组件绑定时，绑定服务才会运行。多个组件可同时绑定到该服务，但全部取消绑定后，该服务即会被销毁。

无论 `Service` 是处于启动状态还是绑定状态(或同时处于这两种状态)，任何应用组件均可像使用 `Activity` 那样，通过调用 `Intent` 来使用服务(即使此服务来自另一应用)。不过，可以通过清单文件将服务声明为私有服务，并阻止其它应用访问该服务。

> 注： 服务在其托管进程的主线程中运行，它既不创建自己的线程，也不在单独的进程中运行(除非另行制定)。**如果服务将执行任何CPU密集型工作或阻塞性操作，则应通过在服务内创建新线程来完成这项工作**。通过使用单独的线程，可以降低发生ANR错误的风险，而应用的主线程仍可继续专注于运行用户与 `Activity` 之间的交互。

二、在 `Service` 和 `Thread` 之间进行选择

`Service` 是一种**即使用户未与应用交互也可以在后台运行的组件**，因此，只有在需要服务时才应创建服务。

如果**必须在主线程之外执行操作，而且只在用户与应用交互时执行此操作**，则应创建新线程。例如，如果只想在 `Activity` 运行的同时播放一些音乐， 则可在 `onCreate()` 中创建线程，在onStart()中启动线程运行，然后在 `onStop()` 中停线程 。

重点：如果使用 `Service` ，则默认情况下，它仍会**在应用的主线程中运行**，因此，**如果服务执行的是密集型或阻塞性操作，则应该 `Service` 内创建新线程。**

三、基础知识

如要创建服务，必须创建 `Service` 的子类(或使用它的一个现有子类)。在实现中，必须重写一些回调方法，从而处理 `Service` 生命周期的某些关键方面，并提供一种机制将组件绑定到 `Service` 。以下是应该重写的最重要的回调方法：

- `onStartCommand()`
  当另一个组件(如 `Activity`)请求启动服务时，系统会通过调用 `startService()` 来调用此方法。执行此方法时，服务即会启动并可在后台无限期运行。如果实现此方法，则在服务工作完成后，需要调用 `stopSelf()` 或 `stopService()` 来停止服务。(如果只想提供绑定，则无需实现此方法)
- `onBind()`
  当另一个组件想要与服务绑定(例如执行RPC)时，系统会通过调用 `bindService()` 来调用此方法。在此方法的实现中，**必须通过返回IBinder提供一个接口，以供client用来与服务进行通信。**请务必实现此方法；但是，如果并不提供允许绑定，则应返回null。
- `onCreate()`
  首次创建服务时，系统会(在调用 `onStartCommand()` 或 `onBind()` 之前)**调用此方法来执行一次性设置程序。**如果服务已在运行，则不会调用此方法。
- `onDestroy()`
  当不再使用服务且准备将其销毁时，系统会调用此方法。服务应通过实现此方法来清理任何资源，如线程、注册的监听器、接收器等。这是服务接收的最后一个调用。

如果组件通过调用 `startService()` 启动服务(这会引起对 `onStartCommand()` 的调用)，则服务会一直运行，知道其使用 `stopSelf()` 自行停止运行，或由其他通过调用 `stopService()` 将其停止为止。

如果组件通过调用 `bindService()` 来创建服务，且未调用 `onStartCommand()` ，则服务只会在该组件与其绑定时运行。当该服务与其所有组件取消绑定后，系统便会将其销毁。

只有在内存过低且必须回收系统资源以供拥有用户焦点的 `Activity` 使用时，Android系统才会停止服务。如果**将服务绑定到拥有用户焦点的Activity，则它不太可能会终止**；如果**将服务声明为在前台运行，则其几乎永远不会终止。**如果服务已启动并长时间运行，则系统会逐渐降低其在后台任务列表中的位置，而服务被终止的概率也会大幅提升，如果服务是已启动，则必须将其设计为能够妥善处理系统执行的重启。如果系统终止服务，则其会在资源可用时立即重启服务，但这还取决于 `onStartComand()` 返回的值。

下面将介绍如何创建 `startService()` 和 `bindService()` 服务方法，以及如何通过其他应用组件使用这些方法。

使用清单文件声明服务
和对 `Activity` 及其他组件的操作一样，必须在应用的清单文件中声明所有服务。
如要生命服务，请添加元素作为元素的子元素。如下：

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

> 注： **为确保应用的安全性，在启动 `Service` 时，应该始终使用显示Intent，且不要为服务声明Intent过滤器。**使用隐式 `Intent` 启动服务存在安全隐患，因为无法确定哪些服务会响应 `Intent` ，而用户也无法看到哪些服务已启动。**从Android 5.0(API级别21)开始，如果使用隐式 `Intent` 调用 `bindService()` ，则系统会抛出异常。**

可以通过添加 `android:exported` 属性并将其设置为 `false` ，确保服务仅适用于你的应用。这可以有效阻止其他应用启动你的服务，即便在使用显示 `Intent` 时也是如此。

四、创建启动服务

启动 `Service` 由另一个组件通过 `startService()` 启动，这会导致调用 `Service` 的 `onStartCommand()` 方法。

`Service` 启动后，其生命周期即独立于启动它的组件。即使系统已销毁启动服务的组件，该服务仍可在后台无限期地运行。因此，服务应在其工作完成时通过调用 `stopSelf()` 来自行停止运行，或者由另一个组件通过调用 `stopService()` 来将其停止。

应用组件(如 `Activity`)可以通过 `startService()` 方法并传递 `Intent` 对象(指定服务并包含待使用的所有数据)来启动 `Service` 。 `Service` 会在 `onStartCommand()` 方法接收此 `Intent` 。

例如，假设某个 `Activity` 需要将一些数据保存后台数据库中。该 `Activity` 可以启动一个协同服务，并通过向 `startService()` 传递一个 `Intent` ，为该服务提供要保存的数据。服务会通过 `onStartCommand()` 接收 `Intent` ，连接到数据上传后，服务将自行停止并销毁。

通常，可以扩展两个类来创建启动服务：

1. `Service`
   这是使用所有服务的基类。扩展此类时，必须创建用于执行所有工作的新线程，因此服务默认使用应用的主线程，这会降低应用正在运行的 `Activity` 的性能。
2. `IntentService`
   这是 `Service` 的子类，其使用工作线程逐一处理所有启动请求。如果不要求服务同时处理多个请求，此类为最佳选择。实现 `onHandleIntent()` ，该方法会接受到每个启动请求的 `Intent` ，以便执行后台工作。

下面介绍如何使用其中任一类来实现服务。

扩展 `IntentService` 类
由于大多数启动服务无需同时处理多个请求(实际上，这种多线程情况可能很危险)，因此**最佳选择是利用 `IntentService` 类实现服务。**
`IntentService` 类会执行以下操作：

- 创建默认的工作线程，用于在应用的主线程外执行传递给 `onStartCommand()` 的所有 `Intent` 。
- 创建工作队列，用于将 `Intent` 逐一传递给 `onHandleIntent()` 实现，这样就永远不必担心多线程问题。
- 在处理完所有启动请求后停止服务，因此不用调用 `stopSelf()` 。
- 提供 `onBind()` 的默认是实现(返回 `null`)。

- 提供 `onStartCommand()` 的默认实现，可将 `Intent` 依次发送到工作队列和 `onHandleIntent()` 实现。

如要完成client提供的工作，应该实现 `onHandleIntent()` 。

```
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super <code><a href="/reference/android/app/IntentService.html#IntentService(j
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns, IntentService
   * stops the service, as appropriate.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
      // Normally we would do some work here, like download a file.
      // For our sample, we just sleep for 5 seconds.
      try {
          Thread.sleep(5000);
      } catch (InterruptedException e) {
          // Restore interrupt status.
          Thread.currentThread().interrupt();
      }
  }
}
```

只需要一个构造函数和一个 `onHandleIntent()` 实现即可。

如果需要重写其他回调方法（如 `onCreate()` 、 `onStartCommand()` 或 `onDestroy()` ），必须调用超类实现，以便 `IntentService` 能够妥善处理工作线程的生命周期。

接下来，讲解如何扩展 `Service` 基类是同类服务，此类包含更多代码，但如需同时处理多个启动请求，则更适合使用该基类。

扩展 `Service` 类
借助 `IntentService` 类，我们可以非常轻松地实现服务。但是，若要求服务执行多线程（而非通过工作队列处理启动请求），则可以通过扩展 `Service` 类来处理每个 `Intent` 。
为进行比较，以下示例代码展示了 `Service` 类的实现，该类执行的工作与上述使用 `IntentService` 的示例完全相同。换言之，对于每个启动请求，其均使用工作线程来执行作业，且每次仅处理一个请求。

```
public class HelloService extends Service {
  private Looper serviceLooper;
  private ServiceHandler serviceHandler;

  // Handler that receives messages from the thread
  private final class ServiceHandler extends Handler {
      public ServiceHandler(Looper looper) {
          super(looper);
      }
      @Override
      public void handleMessage(Message msg) {
          // Normally we would do some work here, like download a file.
          // For our sample, we just sleep for 5 seconds.
          try {
              Thread.sleep(5000);
          } catch (InterruptedException e) {
              // Restore interrupt status.
              Thread.currentThread().interrupt();
          }
          // Stop the service using the startId, so that we don't stop
          // the service in the middle of handling another job
          stopSelf(msg.arg1);
      }

```

```
    }

    @Override
    public void onCreate() {
      // Start up the thread running the service. Note that we create a
      // separate thread because the service normally runs in the process's
      // main thread, which we don't want to block. We also make it
      // background priority so CPU-intensive work doesn't disrupt our UI.
      HandlerThread thread = new HandlerThread("ServiceStartArguments",
              Process.THREAD_PRIORITY_BACKGROUND);
      thread.start();

      // Get the HandlerThread's Looper and use it for our Handler
      serviceLooper = thread.getLooper();
      serviceHandler = new ServiceHandler(serviceLooper);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

        // For each start request, send a message to start a job and deliver the
        // start ID so we know which request we're stopping when we finish the job
        Message msg = serviceHandler.obtainMessage();
        msg.arg1 = startId;
        serviceHandler.sendMessage(msg);

        // If we get killed, after returning from here, restart
        return START_STICKY;
    }

    @Override
    public IBinder onBind(Intent intent) {
        // We don't provide binding, so return null
        return null;
    }

    @Override
    public void onDestroy() {
      Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
    }
  }
```

相较于使用 `IntentService` ，此示例需要执行更多工作。
但是，由于 `onStartCommand()` 的每个调用均由自己处理，因此可以同时执行多个请求。
`onStartCommand()` 方法必须返回整型数。整型数是一个用于描述系统应该如何在系统终止服务的情况下继续运行服务。 `IntentService` 的默认实现会为处理此情况，但可以进行修改。 `onStartCommand()` 返回值必须为以下常量：

- `START_NOT_STICKY`
  如果系统在 `onStartCommand()` 返回后终止服务，则除非有待传递的挂起 `Intent` ，否则系统不会重建服务。这是最安全的选项，可以避免在不必要以及应用能够轻松重启所有未完成的作业时运行服务。
- `START_STICKY`
  如果系统在 `onStartCommand()` 返回后终止服务，则其会重建服务并调用 `onStartCommand()` ，但不会重新传递最后一个 `Intent` 。相反，除非有挂起Intent要启动服务，否则系统会调用包含空 `Intent` 的 `onStartCommand()` 。在此情况下，系统会传递这些 `Intent` 。此常量适用于不执行命令、但无限期运行并等待作业的谋体播放器（类似服务）。
- `START_REDELIVER_INTENT`
  如果系统在 `onStartCommand()` 返回后终止服务，则其会重建服务，并通过传递给服务的最后一个 `Intent` 调用 `onStartCommand()` 。所有挂起 `Intent` 均依次传递。此常量使用于主动执行应立即恢复的作业（例如下载文件）的服务。

五、启动\终止服务

启动服务

可以通过将 `Intent` 传递给 `startService()` 或 `startForegroundService()` ，从 `Activity` 或其他应用组件启动服务。Android系统会调用服务的 `onStartCommand()` 方法，并向其传递 `Intent` ,从而制定要启动的服务。

> 注：如果在API 26或者更高版本，除非应用本身在前台运行，否则系统不会对使用或创建后台服务施加限制。**如果应用需要创建前台服务，则其应调用 `startForegroundService` 。**此方法会会创建后台服务，但它会向系统发出信号，表明服务会将自行提升至前台，创建服务后，该服务必须在五秒内调用自己的 `startForeground()` 方法。

`Activity` 可以结合使用显式 `Intent` 与 `startService()` ，如下：

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

`startService()` 方法会立即返回，并且系统会调用服务的 `onStartCommand()` 方法。如果服务尚未运行，则系统首先会调用 `onCreate()` ，然后调用 `onStartCommand()` 。

如果服务业未提供绑定，则应用组件与服务间的唯一通信模式便是使用 `startService()` 传递的 `Intent` 。

多个服务启动请求会导致多次对服务的 `onStartCommand()` 进行相应的调用。但是，如果停止服务，只需一个服务停止请求（使用 `stopSelf()` 或 `stopService()` ）即可。

停止服务
启动服务必须管理自己的生命周期。换言之，除非必须回收内存资源，否则系统不会停止或销毁服务，并且 `Service` 在 `onStartCommand()` 返回后仍会继续运行。 `Service` 必须通过调用 `stopSelf()` 自行停止运行，或者由另一个组件通过调用 `stopService()` 来停止它。

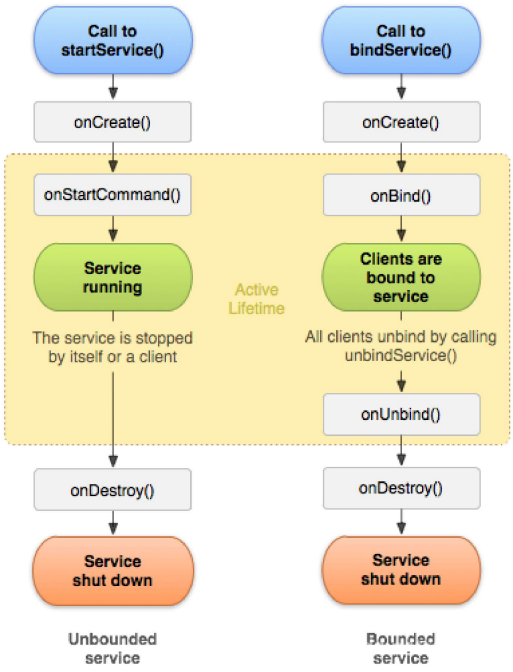
如果 `Service` 同时处理多个对 `onStartCommand()` 的请求，不应该在处理完一个请求之后停止服务，因为可能已收到新的启动请求（在第一个请求结束时停止服务会终止第二个请求）。为避免此问题，可以使用 `stopSelf(int)` 确保服务停止请求始终基于最近的启动请求 ID（传递给 `onStartCommand()` 的 `startId` ）。如果 `Service` 在调用 `stopSelf(int)` 之前收到新启动请求，则ID不匹配，服务也不会停止。

> 为避免浪费系统资源和消耗电池电量，必须确保应用在服务工作完成之后停止该服务。

六、管理服务的生命周期

`Service` 的生命周期比 `Activity` 的生命周期要简单。我们更应该关注如何创建和销毁服务，因为服务可以在用户未意识到的情况下运行于后台。
`Service` 生命周期（从创建到销毁）可遵循以下任一路径：

- 启动服务
  该服务在其他组件调用 `startService()` 时创建，然后无限期运行，且必须通过调用 `stopSelf()` 来自行停止运行。此外，其他组件也可以通过调用 `stopService()` 来停止服务。服务停止后，系统会将其销毁。
- 绑定服务
  该服务在其他组件(客户端)调用 `bindService()` 时创建。然后客户端通过 `IBinder` 接口与 `Service` 进行通信。客户端可以通过调用 `unbindService()` 关闭连接。**多个客户端可以绑定到相同服务，而且当所有绑定全部取消后，系统即会销毁该服务。**



服务生命周期。左图显示使用 `startService()` 创建的服务的生命周期，右图显示使用 `bindService()` 创建的服务的生命周期