Android 软件开发

Services

秦兴国

xgqin@guet.edu.cn

计算机与信息安全学院
桂林电子科技大学

2018 年 11 月 30 日

Outline

Service 简介

服务 (Service) 是 Android 应用开发的四大组件之一，其主要用于在后台执行长时间 (long-running) 任务操作。与活动 (Activity) 相比，服务不直接提供 UI 与用户进行交互。

Service 简介

服务 (Service) 是 Android 应用开发的四大组件之一，其主要用于在后台执行长时间 (long-running) 任务操作。与活动 (Activity) 相比，服务不直接提供 UI 与用户进行交互。

服务可由 App 的其他组件进行启动，即便启动服务后该 App 被 Android 系统切换到后台后，启动的服务也可继续在后台运行而不会被中断。

Service 简介

服务 (Service) 是 Android 应用开发的四大组件之一，其主要用于在后台执行长时间 (long-running) 任务操作。与活动 (Activity) 相比，服务不直接提供 UI 与用户进行交互。

服务可由 App 的其他组件进行启动，即便启动服务后该 App 被 Android 系统切换到后台后，启动的服务也可继续在后台运行而不会被中断。

App 内的组件也可通过绑定的方式获取一个服务实例，并与该服务实例进行交互 (访问该服务的公有方法及属性)。

Service 简介

服务 (Service) 是 Android 应用开发的四大组件之一，其主要用于在后台执行长时间 (long-running) 任务操作。与活动 (Activity) 相比，服务不直接提供 UI 与用户进行交互。

服务可由 App 的其他组件进行启动，即便启动服务后该 App 被 Android 系统切换到后台后，启动的服务也可继续在后台运行而不会被中断。

App 内的组件也可通过绑定的方式获取一个服务实例，并与该服务实例进行交互 (访问该服务的公有方法及属性)。

服务可用来做播放音乐、处理网络切换、执行文件读写或访问内容提供器 (ContentProvider) 等类型的后台任务。

ServiceService 的分类

服务可分为三种类型：

1. **前台服务 (Foreground Service)**，前台服务对于用户是可感知的 (noticeable)，通常需要显示一个通知 (Notification) 以便提供用户交互入口。前台服务在无用户交互的情况下也保持运行。

[1] 服务既可以同时被直接启动也可被绑定启动，这取决于在子类化Service时对onStartCommand()及onBind()方法的实现。

ServiceService 的分类

服务可分为三种类型：

1. **前台服务 (Foreground Service)**，前台服务对于用户是可感知的 (noticeable)，通常需要显示一个通知 (Notification) 以便提供用户交互入口。前台服务在无用户交互的情况下也保持运行。
2. **后台服务 (Background Service)**，后台服务运行的任务通常对于用户而言是不可直接感知的。

[1] 服务既可以同时被直接启动也可被绑定启动，这取决于在子类化Service时对onStartCommand()及onBind()方法的实现。

ServiceService 的分类

服务可分为三种类型：

1. **前台服务 (Foreground Service)**，前台服务对于用户是可感知的 (noticeable)，通常需要显示一个通知 (Notification) 以便提供用户交互入口。前台服务在无用户交互的情况下也保持运行。
2. **后台服务 (Background Service)**，后台服务运行的任务通常对于用户而言是不可直接感知的。
3. **绑定服务 (Bound Service)**[1] ，通过bindService()方法绑定的的服务称为绑定服务。服务可以被多个组件进行绑定，当所有组件解绑该服务后 (unbindService())，服务生命周期结束。

[1] 服务既可以同时被直接启动也可被绑定启动，这取决于在子类化Service时对onStartCommand()及onBind()方法的实现。

服务与线程的区别

服务默认运行在 App 的主线程中，除非在运行服务时显式的指明或在服务中手动创建线程。

服务与线程的区别

服务默认运行在 App 的主线程中，除非在运行服务时显式的指明或在服务中手动创建线程。

如果在服务中需要执行强计算性 (CPU-intensive) 的任务或阻塞性的任务，则应该在服务中创建线程来完成这类任务。

服务与线程的区别

服务默认运行在 App 的主线程中，除非在运行服务时显式的指明或在服务中手动创建线程。

如果在服务中需要执行强计算性 (CPU-intensive) 的任务或阻塞性的任务，则应该在服务中创建线程来完成这类任务。

服务与线程的区别：

- 如果所执行的任务不需要与用户进行交互，则可将其放在服务中执行。

服务与线程的区别

服务默认运行在 App 的主线程中，除非在运行服务时显式的指明或在服务中手动创建线程。

如果在服务中需要执行强计算性 (CPU-intensive) 的任务或阻塞性的任务，则应该在服务中创建线程来完成这类任务。

服务与线程的区别：

- 如果所执行的任务不需要与用户进行交互，则可将其放在服务中执行。
- 如果所执行的任务需要与用户进行交互，则可将其放在线程中执行。

创建及使用服务 -I

创建及使用服务的步骤：

1. 在配置清单 (**AndroidManifest.xml**) 文件中声明服务组件；
2. 定义Service(或 IntentService 子类，并实现相应的回调方法；
3. 管理服务的生命周期；

创建及使用服务 -子类化 Service I

创建服务可以直接之类化一个Service类或Service已存在的其他子类。

在此基础上覆盖影响服务生命周期的各回调方法，以及提供组件绑定服务的其他方法。

以下是子类化Service时比较重要的回调方法：

- **onStartCommand()**，当其他组件通过 startService ()启动服务时，Android 系统将调用该回调方法。该方法执行完成后，服务将在后台执行。如果需停止服务，在服务内部调用stopSelf ()方法或其他组件调用stopService ()。如果仅需提供绑定方式启动服务，则可选择不覆盖掉该方法。

创建及使用服务 -子类化 Service II

- **onBind()**，当其他组件通过bindService()绑定服务时，Android 系统将调用该回调方法。在该方法中通过返回IBinder接口的对象为绑定服务的组件提供与服务进行通信的接口。如果不提供绑定方式启动服务，则该方法返回值为*null*。
- **onCreate()**，服务初始化的回调方法，Android 在服务被第一次创建时 (通过onStartCommand()或onBind()) 调用该方法；如果服务已经在运行，则该回调方法将不会被执行。
- **onDestroy()**，当服务不在需要运行时，该回调方法将被执行。该方法是服务最后被执行的方法，在该方法中处理资源的释放清理操作。

停止服务

服务的停止根据其启动方式不同分为如下两种情况：

- 通过 startService ()方式创建的服务，可在服务内部调用 stopSelf ()或由其他组件调用 stopService ()结束；

停止服务

服务的停止根据其启动方式不同分为如下两种情况：

- 通过 startService ()方式创建的服务，可在服务内部调用 stopSelf ()或由其他组件调用 stopService ()结束；
- 通过 bindService ()方式创建的服务，当所有绑定该服务的组件调用 unbindService ()后，服务结束；

停止服务

服务的停止根据其启动方式不同分为如下两种情况：

- 通过 startService ()方式创建的服务，可在服务内部调用 stopSelf ()或由其他组件调用 stopService ()结束；
- 通过 bindService ()方式创建的服务，当所有绑定该服务的组件调用 unbindService ()后，服务结束；

停止服务

服务的停止根据其启动方式不同分为如下两种情况：

- 通过 startService ()方式创建的服务，可在服务内部调用 stopSelf ()或由其他组件调用 stopService ()结束；
- 通过 bindService ()方式创建的服务，当所有绑定该服务的组件调用 unbindService ()后，服务结束；

Android 系统在系统资源不足的情况下会强行停止服务。

- 如果启动或绑定服务的活动 (Activity) 获得当前用户焦点，则该服务被停止的可能性比较小。

停止服务

服务的停止根据其启动方式不同分为如下两种情况：

- 通过 startService ()方式创建的服务，可在服务内部调用 stopSelf ()或由其他组件调用 stopService ()结束；
- 通过 bindService ()方式创建的服务，当所有绑定该服务的组件调用 unbindService ()后，服务结束；

Android 系统在系统资源不足的情况下会强行停止服务。

- 如果启动或绑定服务的活动 (Activity) 获得当前用户焦点，则该服务被停止的可能性比较小。
- 如果服务被声明为前台服务，则其几乎不可能被系统停止。

停止服务

服务的停止根据其启动方式不同分为如下两种情况：

- 通过 startService ()方式创建的服务，可在服务内部调用 stopSelf ()或由其他组件调用 stopService ()结束；
- 通过 bindService ()方式创建的服务，当所有绑定该服务的组件调用 unbindService ()后，服务结束；

Android 系统在系统资源不足的情况下会强行停止服务。

- 如果启动或绑定服务的活动 (Activity) 获得当前用户焦点，则该服务被停止的可能性比较小。
- 如果服务被声明为前台服务，则其几乎不可能被系统停止。
- 如果服务长时间运行于后台，Android 系统会动态调整该服务在后台任务的优先级，这会增加该服务被系统停止可能性。

创建及使用服务 -在配置清单文件中声明服务组件

与活动及其他组件相同，服务组件也必须在配置清单文件中进行声明。

```
1 <manifest ... >
2   ...
3   <application ... >
4       <service android:name="ExampleService"
5                android:exported="false" />
6       ...
7   </application>
8 </manifest>
```

[2] 为确保 App 的安全，通常情况下应该通过显示 *Intent* 启动或绑定服务，在配置清单文件中声明服务时也不应该添加 *Intent Filter*。从 Android 5.0 开始，系统会对通过bindService()隐式绑定服务的操作抛出异常。

创建及使用服务 -在配置清单文件中声明服务组件

与活动及其他组件相同，服务组件也必须在配置清单文件中进行声明。

```
1 <manifest ... >
2   ...
3   <application ... >
4       <service android:name="ExampleService"
5                android:exported="false" />
6       ...
7   </application>
8 </manifest>
```

在<service>标签中可定义权限等其他属性。android:name属性在 App 发布之后，应保持该值不变。因为其他 App 会通过该值显式启动或绑定该服务[2]。

[2]为确保 App 的安全，通常情况下应该通过显示 *Intent* 启动或绑定服务，在配置清单文件中声明服务时也不应该添加 *Intent Filter*。从 Android 5.0 开始，系统会对通过bindService()隐式绑定服务的操作抛出异常。

创建启动服务 I

启动服务 (**started service**) 是指通过调用 startService ()方式启动的服务 (包括后台服务和前台服务)。

创建启动服务 I

启动服务 (**started service**) 是指通过调用 startService ()方式启动的服务 (包括后台服务和前台服务)。

服务启动后，其生命周期独立于启动它的组件。服务可在后台长时间运行，即使启动该服务的组件已经被销毁。

创建启动服务 I

启动服务 (**started service**) 是指通过调用 startService ()方式启动的服务 (包括后台服务和前台服务)。

服务启动后，其生命周期独立于启动它的组件。服务可在后台长时间运行，即使启动该服务的组件已经被销毁。

因此，服务在完成其工作后应该调用 stopSelf ()或其他组件调用 stopService ()停止。

创建启动服务 I

启动服务 (**started service**) 是指通过调用 startService ()方式启动的服务 (包括后台服务和前台服务)。

服务启动后，其生命周期独立于启动它的组件。服务可在后台长时间运行，即使启动该服务的组件已经被销毁。

因此，服务在完成其工作后应该调用 stopSelf ()或其他组件调用 stopService ()停止。

组件通过 startService ()启动服务时，所传递的参数 *Intent* 对象，指定了启动的服务以及传递给服务的数据。服务可在onStartCommand()方法中接收到该 *Intent* 对象。

创建启动服务 II

创建启动服务的两种方式:

[3] 如果不需要同时处理多个启动服务的请求，使用IntentService是最好的选择。

创建启动服务 II

创建启动服务的两种方式:

1. 继承 Service；继承自Service的服务子类，默认情况下运行于 App 的主线程中，通常需要在该服务中启动新的线程完成其工作。

[3] 如果不需要同时处理多个启动服务的请求，使用IntentService是最好的选择。

创建启动服务 II

创建启动服务的两种方式:

1. 继承 Service；继承自Service的服务子类，默认情况下运行于 App 的主线程中，通常需要在该服务中启动新的线程完成其工作。
2. 继承 IntentService； IntentService 是 Service的子类，它使用一个子线程顺序处理 (one at a time) 所有的启动服务请求[3]。实现onHandleIntent()方法用于处理每一个启动服务的请求，并在该方法中执行所需的任务。

[3] 如果不需要同时处理多个启动服务的请求，使用IntentService是最好的选择。

创建启动服务 III – 继承 IntentService 类

大多数的启动活动不需要处理并发请求的情况。因此实现服务的最佳方式是使用 IntentSevice 。

创建启动服务 III – 继承 IntentService 类

大多数的启动活动不需要处理并发请求的情况。因此实现服务的最佳方式是使用 IntentSevice 。

- IntentSevice 通过创建单独的子线程执行所有分发至onStartCommand()方法的 *Intent* 意图对象；

创建启动服务 III – 继承 IntentService 类

大多数的启动活动不需要处理并发请求的情况。因此实现服务的最佳方式是使用 IntentSevice 。

- IntentSevice 通过创建单独的子线程执行所有分发至onStartCommand()方法的 *Intent* 意图对象；
- 创建一个任务队列并顺序的分发至onHandleIntent()方法中，避免了多线程的并发问题；

创建启动服务 III – 继承 IntentService 类

大多数的启动活动不需要处理并发请求的情况。因此实现服务的最佳方式是使用 IntentSevice 。

- IntentSevice 通过创建单独的子线程执行所有分发至onStartCommand()方法的 *Intent* 意图对象；
- 创建一个任务队列并顺序的分发至onHandleIntent()方法中，避免了多线程的并发问题；
- 处理完所有的请求后，自动调用 stopSelf ()方法停止服务；

创建启动服务 III – 继承 IntentService 类

大多数的启动活动不需要处理并发请求的情况。因此实现服务的最佳方式是使用 IntentSevice 。

- IntentSevice 通过创建单独的子线程执行所有分发至onStartCommand()方法的 *Intent* 意图对象；
- 创建一个任务队列并顺序的分发至onHandleIntent()方法中，避免了多线程的并发问题；
- 处理完所有的请求后，自动调用 stopSelf ()方法停止服务；
- 提供了默认的onBind()方法，并返回 *null*；

创建启动服务 III – 继承 IntentService 类

大多数的启动活动不需要处理并发请求的情况。因此实现服务的最佳方式是使用 IntentSevice 。

- IntentSevice 通过创建单独的子线程执行所有分发至onStartCommand()方法的 *Intent* 意图对象；
- 创建一个任务队列并顺序的分发至onHandleIntent()方法中，避免了多线程的并发问题；
- 处理完所有的请求后，自动调用 stopSelf ()方法停止服务；
- 提供了默认的onBind()方法，并返回 *null*；
- 提供了默认的onStartCommand()方法，并将以队列形式管理启动服务的 *Intent* 意图对象，为onHandleIntent()方法提供数据。

创建启动服务 III – 继承 IntentService 类

继承自 IntentService 的子类，需要实现的方法主要是onHandleIntent()[4]以及类构造方法。

[4] onHandleIntent() 方法由IntentService中的线程进行调用，该方法返回时IntentService适时停止自身。

创建启动服务 III – 继承 IntentService 类

继承自 IntentService 的子类，需要实现的方法主要是onHandleIntent()[4]以及类构造方法。

```
1  public class HelloIntentService extends IntentService {
2
3    public HelloIntentService() {
4        super("HelloIntentService");
5    }
6
7    @Override
8    protected void onHandleIntent(Intent intent) {
9        // Normally we would do some work here, like download a file.
10       // For our sample, we just sleep for 5 seconds.
11       try {
12           Thread.sleep(5000);
13       } catch (InterruptedException e) {
14           // Restore interrupt status.
15           Thread.currentThread().interrupt();
16       }
17   }
18 }
```

[4] onHandleIntent() 方法由IntentService中的线程进行调用，该方法返回时IntentService适时停止自身。

创建启动服务 III – 继承 IntentService 类

除了需要实现 IntentService 的构造方法及onHandleIntent()[5]方法外，如需实现服务的其他方法，则应确保在覆盖的方法中调用父类的原方法以便 IntentService 能够正确处理其子线程的生命周期。

[5]除了onHandleIntent()方法外，onBind()方法也无需调用其父类的相应方法。

创建启动服务 III – 继承 IntentService 类

除了需要实现 IntentService 的构造方法及onHandleIntent()[5]方法外，如需实现服务的其他方法，则应确保在覆盖的方法中调用父类的原方法以便 IntentService 能够正确处理其子线程的生命周期。

```
1 @Override
2 public int onStartCommand(Intent intent, int flags, int startId) {
3     Toast.makeText(this, "service␣starting", Toast.LENGTH_SHORT).show();
4     return super.onStartCommand(intent,flags,startId);
5 }
```

[5]除了onHandleIntent()方法外，onBind()方法也无需调用其父类的相应方法。

创建启动服务 III – 继承 Service 类

IntentService 是实现服务的最佳方式。如需要处理多线程并发的问题，则可通过继承 Service 的方式来实现。

创建启动服务 III – 继承 Service 类

在onStartCommand()中处理每一个启动服务的请求时，可在该方法中创建一个子线程响应每个服务请求。

创建启动服务 III – 继承 Service 类

在onStartCommand()中处理每一个启动服务的请求时，可在该方法中创建一个子线程响应每个服务请求。

onStartCommand()方法的返回值为 *int* 类型。该值表示当服务执行完onStartCommand()方法，被 Android 系统杀掉后，服务再次被创建时的行为。其值可为如下几种情况：

- START_NOT_STICKY，不再重新创建服务，除非存在待处理的 *Intent* 对象。

创建启动服务 III – 继承 Service 类

在onStartCommand()中处理每一个启动服务的请求时，可在该方法中创建一个子线程响应每个服务请求。

onStartCommand()方法的返回值为 *int* 类型。该值表示当服务执行完onStartCommand()方法，被 Android 系统杀掉后，服务再次被创建时的行为。其值可为如下几种情况：

- START_NOT_STICKY，不再重新创建服务，除非存在待处理的 *Intent* 对象。
- START_STICKY，重新创建服务并调用onStartCommand()，如果无待处理的 *Intent* 对象，则分发一个空的 *Intent*。

创建启动服务 III – 继承 Service 类

在onStartCommand()中处理每一个启动服务的请求时，可在该方法中创建一个子线程响应每个服务请求。

onStartCommand()方法的返回值为 *int* 类型。该值表示当服务执行完onStartCommand()方法，被 Android 系统杀掉后，服务再次被创建时的行为。其值可为如下几种情况：

- START_NOT_STICKY，不再重新创建服务，除非存在待处理的 *Intent* 对象。
- START_STICKY，重新创建服务并调用onStartCommand()，如果无待处理的 *Intent* 对象，则分发一个空的 *Intent*。
- START_REDELIVER_INTENT，重新创建服务并将最后一次被分发至服务的 *Intent* 对象再次分发给onStartCommand()，此后再分发待处理的 *Intent* 对象。

服务的启动

服务可在活动 (Activity) 或其他应用组件中启动，通过将 *Intent* 对象传递给 startService ()或 startForegroundService ()启动一个服务 [6]。

[6] 在 Android 8.0 以上，除非 App 运行在前台，否则系统限制该 App 对后台服务的启动及创建操作。

服务的启动

服务可在活动 (Activity) 或其他应用组件中启动，通过将 *Intent* 对象传递给 startService ()或 startForegroundService ()启动一个服务 [6]。

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

[6] 在 Android 8.0 以上，除非 App 运行在前台，否则系统限制该 App 对后台服务的启动及创建操作。

服务的启动

服务可在活动 (Activity) 或其他应用组件中启动，通过将 *Intent* 对象传递给 startService ()或 startForegroundService ()启动一个服务 [6]。

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

如果服务未在运行，则 Android 系统首先调用onCreate()方法。

[6] 在 Android 8.0 以上，除非 App 运行在前台，否则系统限制该 App 对后台服务的启动及创建操作。

服务的启动

服务可在活动 (Activity) 或其他应用组件中启动，通过将 *Intent* 对象传递给 startService ()或 startForegroundService ()启动一个服务 [6]。

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

如果服务未在运行，则 Android 系统首先调用onCreate()方法。

Android 系统调用onStartCommand()方法，并将 *Intent* 对象传递给该方法。

[6] 在 Android 8.0 以上，除非 App 运行在前台，否则系统限制该 App 对后台服务的启动及创建操作。

停止服务

启动的服务需要管理其自身生命周期[7]，服务在执行完成onStartCommand()方法后将继续运行。服务可在其内部调用 stopSelf ()或在其他组件调用 stopService ()停止服务[8]。

如果服务同时处理多个并发请求，在服务完成某一请求后，应该通过 stopSelf (int)方法停止服务。该方法的参数是onStartCommand()中传递进来的 *startId*。

[7] 当 Android 系统启动一个服务后，除非系统内存不足的情况发生，系统不会停止或销毁该服务。

[8] 无论调用多少次 startService ()启动服务，App 仅需要调用一次stopSelf ()或stopService()即可停止服务。

停止服务

启动的服务需要管理其自身生命周期[7]，服务在执行完成onStartCommand()方法后将继续运行。服务可在其内部调用 stopSelf ()或在其他组件调用 stopService ()停止服务[8]。

如果服务同时处理多个并发请求，在服务完成某一请求后，应该通过 stopSelf (int)方法停止服务。该方法的参数是onStartCommand()中传递进来的 *startId*。

如果此时在调用 stopSelf (int)前，有其他启动服务请求发生，则由于启动服务的 startId 与停止服务的 startId 不相同，服务就不会停止，从而确保服务可正确处理并发请求的情况。

[7] 当 Android 系统启动一个服务后，除非系统内存不足的情况发生，系统不会停止或销毁该服务。

[8] 无论调用多少次 startService ()启动服务，App 仅需要调用一次stopSelf ()或stopService()即可停止服务。

创建绑定服务 I

绑定服务 (bound service) 允许其他应用组件通过bindService()的方式[9]建立服务与组件间的长连接。

[9] 绑定服务通常不允许以startService()的方式启动

创建绑定服务 I

绑定服务 (bound service) 允许其他应用组件通过bindService()的方式[9]建立服务与组件间的长连接。

创建绑定服务需要实现onBind()方法，该方法需返回一个 *IBinder* 接口对象，*IBinder* 接口对象用于组件与服务之间进行通信。

[9] 绑定服务通常不允许以startService()的方式启动

创建绑定服务 I

绑定服务 (bound service) 允许其他应用组件通过bindService()的方式[9]建立服务与组件间的长连接。

创建绑定服务需要实现onBind()方法，该方法需返回一个 *IBinder* 接口对象，*IBinder* 接口对象用于组件与服务之间进行通信。

App 的组件可以调用 bindService()方法获得该接口，并通过该接口调用绑定服务的公有方法。

[9] 绑定服务通常不允许以 startService()的方式启动

创建绑定服务 I

绑定服务 (bound service) 允许其他应用组件通过bindService()的方式[9]建立服务与组件间的长连接。

创建绑定服务需要实现onBind()方法，该方法需返回一个 *IBinder* 接口对象，*IBinder* 接口对象用于组件与服务之间进行通信。

App 的组件可以调用 bindService()方法获得该接口，并通过该接口调用绑定服务的公有方法。

当所有绑定服务的组件通过调用unbindService()解除服务绑定后，系统将销毁该服务。

[9] 绑定服务通常不允许以startService()的方式启动

创建绑定服务 II

创建绑定服务最重要的部分是重载onBind()方法，并实现该方法返回接口 *IBinder*。

[10] *Binder* 实现了 *IBinder* 接口。

创建绑定服务 II

创建绑定服务最重要的部分是重载onBind()方法，并实现该方法返回接口 *IBinder*。

```
1    public abstract IBinder onBind (Intent intent);
```

[10] *Binder* 实现了 *IBinder* 接口。

创建绑定服务 II

创建绑定服务最重要的部分是重载onBind()方法，并实现该方法返回接口 *IBinder*。

```
1    public abstract IBinder onBind (Intent intent);
```

实现该接口的常见方式是定义一个 *Binder* 子类[10]，并将该 *Binder* 子类作为onBind()方法的返回值。

[10] *Binder* 实现了 *IBinder* 接口。

创建绑定服务 III

定义 *Binder* 子类时，通常需要提供如下功能：

创建绑定服务 III

定义 *Binder* 子类时，通常需要提供如下功能：

- *Binder* 子类定义公有方法供客户端使用；
- *Binder* 子类提供获取绑定服务实例的方法，该服务实例提供公有方法供客户端使用；
- 提供绑定服务内部的其他类实例，该类实例包含公有方法供客户端使用；

创建绑定服务 IV

```
1  public class LocalService extends Service {
2      // Binder given to clients
3      private final IBinder mBinder = new LocalBinder();
4      // Random number generator
5      private final Random mGenerator = new Random();
6
7      public class LocalBinder extends Binder {
8          LocalService getService() {
9              // Return this instance of LocalService so clients can call public methods
10             return LocalService.this;
11         }
12     }
13
14     @Override
15     public IBinder onBind(Intent intent) {
16         return mBinder;
17     }
18
19     /** method for clients */
20     public int getRandomNumber() {
21       return mGenerator.nextInt(100);
22     }
23 }
```

创建绑定服务 IV

实现onBind()方法，并返回 *Binder* 类实例；

创建绑定服务 IV

实现onBind()方法，并返回 *Binder* 类实例；

在客户端的 *ServiceConnection* 接口的onServiceConnected()方法中获得 *Binder* 类实例。

创建绑定服务 V

在客户端组件中实现 *ServiceConnection* 接口。

```
1  public class BindingActivity extends Activity {
2      LocalService mService;
3      boolean mBound = false;
4
5      ...
6      /** Defines callbacks for service binding, passed to bindService() */
7      private ServiceConnection mConnection = new ServiceConnection() {
8
9          @Override
10         public void onServiceConnected(ComponentName className,IBinder service) {
11             LocalBinder binder = (LocalBinder) service;
12             mService = binder.getService();
13             mBound = true;
14         }
15
16         @Override
17         public void onServiceDisconnected(ComponentName arg0) {
18             mBound = false;
19         }
20     };
21 }
```

创建绑定服务 VI

```
1     ...
2     @Override
3     protected void onStart() {
4         super.onStart();
5         // Bind to LocalService
6         Intent intent = new Intent(this, LocalService.class);
7         bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
8     }
9
10    @Override
11    protected void onStop() {
12        super.onStop();
13        unbindService(mConnection);
14        mBound = false;
15    }
16    ...
```

创建绑定服务 VII

```
1       ...
2       public void onButtonClick(View v) {
3           if (mBound) {
4
5               int num = mService.getRandomNumber();
6               Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT).show();
7           }
8       }
9       ...
```

创建前台服务 I

前台服务 (foreground service) 是指用户可感知的服务，且当系统内存过低时不会被销毁的服务。

[11] 前台服务用于运行用户可感知的任务 (虽然该任务不一定需要用户与 App 直接交互)，在设置状态栏通知优先级时，必须设定其优先级为 *PRIORITY_LOW* 及以上

[12] 在 Android 9 以上版本中，运行前台服务的 App 需要申请 *FOREGROUND_SERVICE* 权限，否则系统将抛出 *SecurityException* 异常。

创建前台服务 I

前台服务 (foreground service) 是指用户可感知的服务，且当系统内存过低时不会被销毁的服务。

前台服务通常需要以状态栏通知的形式与用户交互[11] [12] 。该通知不可被删除，除非服务停止或被从前台服务列表中删除。

[11] 前台服务用于运行用户可感知的任务 (虽然该任务不一定需要用户与 App 直接交互)，在设置状态栏通知优先级时，必须设定其优先级为 *PRIORITY_LOW* 及以上

[12] 在 Android 9 以上版本中，运行前台服务的 App 需要申请 *FOREGROUND_SERVICE* 权限，否则系统将抛出 *SecurityException* 异常。

创建前台服务 I

前台服务 (foreground service) 是指用户可感知的服务，且当系统内存过低时不会被销毁的服务。

前台服务通常需要以状态栏通知的形式与用户交互[11] [12] 。该通知不可被删除，除非服务停止或被从前台服务列表中删除。

以音乐播放器为例，其使用服务播放音乐，并将该服务运行于前台。此时应该以状态栏通知的形式显示当前播放的音乐信息，以及提供用户启动音乐播放器的操作。

---

[11] 前台服务用于运行用户可感知的任务 (虽然该任务不一定需要用户与 App 直接交互)，在设置状态栏通知优先级时，必须设定其优先级为 *PRIORITY_LOW* 及以上

[12] 在 Android 9 以上版本中，运行前台服务的 App 需要申请 *FOREGROUND_SERVICE* 权限，否则系统将抛出 *SecurityException* 异常。

创建前台服务 I

前台服务 (foreground service) 是指用户可感知的服务，且当系统内存过低时不会被销毁的服务。

前台服务通常需要以状态栏通知的形式与用户交互[11] [12] 。该通知不可被删除，除非服务停止或被从前台服务列表中删除。

以音乐播放器为例，其使用服务播放音乐，并将该服务运行于前台。此时应该以状态栏通知的形式显示当前播放的音乐信息，以及提供用户启动音乐播放器的操作。

---

[11] 前台服务用于运行用户可感知的任务 (虽然该任务不一定需要用户与 App 直接交互)，在设置状态栏通知优先级时，必须设定其优先级为 *PRIORITY_LOW* 及以上

[12] 在 Android 9 以上版本中，运行前台服务的 App 需要申请 *FOREGROUND_SERVICE* 权限，否则系统将抛出 *SecurityException* 异常。

创建前台服务 II

创建前台运行服务，首先使用 startForegroundService ()方法启动服务[13]。在启动服务后，调用startForeground ()方法设置对应的状态栏通知。

[13]使用该方法启动的服务仍为后台运行的服务。

[14]该方法参数指明是否需要移除该服务对应的状态栏通知。

创建前台服务 II

创建前台运行服务，首先使用 startForegroundService ()方法启动服务[13]。在启动服务后，调用startForeground ()方法设置对应的状态栏通知。

```
1  Intent notificationIntent = new Intent(this, ExampleActivity.class);
2  PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);
3
4  Notification notification = new Notification.Builder(this, CHANNEL_DEFAULT_IMPORTANCE)
5      .setContentTitle(getText(R.string.notification_title))
6      .setContentText(getText(R.string.notification_message))
7      .setSmallIcon(R.drawable.icon)
8      .setContentIntent(pendingIntent)
9      .setTicker(getText(R.string.ticker_text))
10     .build();
11
12 startForeground(ONGOING_NOTIFICATION_ID, notification);
```

[13]使用该方法启动的服务仍为后台运行的服务。

[14]该方法参数指明是否需要移除该服务对应的状态栏通知。

创建前台服务 II

创建前台运行服务，首先使用 startForegroundService ()方法启动服务[13]。在启动服务后，调用startForeground ()方法设置对应的状态栏通知。

```
1  Intent notificationIntent = new Intent(this, ExampleActivity.class);
2  PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);
3
4  Notification notification = new Notification.Builder(this, CHANNEL_DEFAULT_IMPORTANCE)
5      .setContentTitle(getText(R.string.notification_title))
6      .setContentText(getText(R.string.notification_message))
7      .setSmallIcon(R.drawable.icon)
8      .setContentIntent(pendingIntent)
9      .setTicker(getText(R.string.ticker_text))
10     .build();
11
12 startForeground(ONGOING_NOTIFICATION_ID, notification);
```

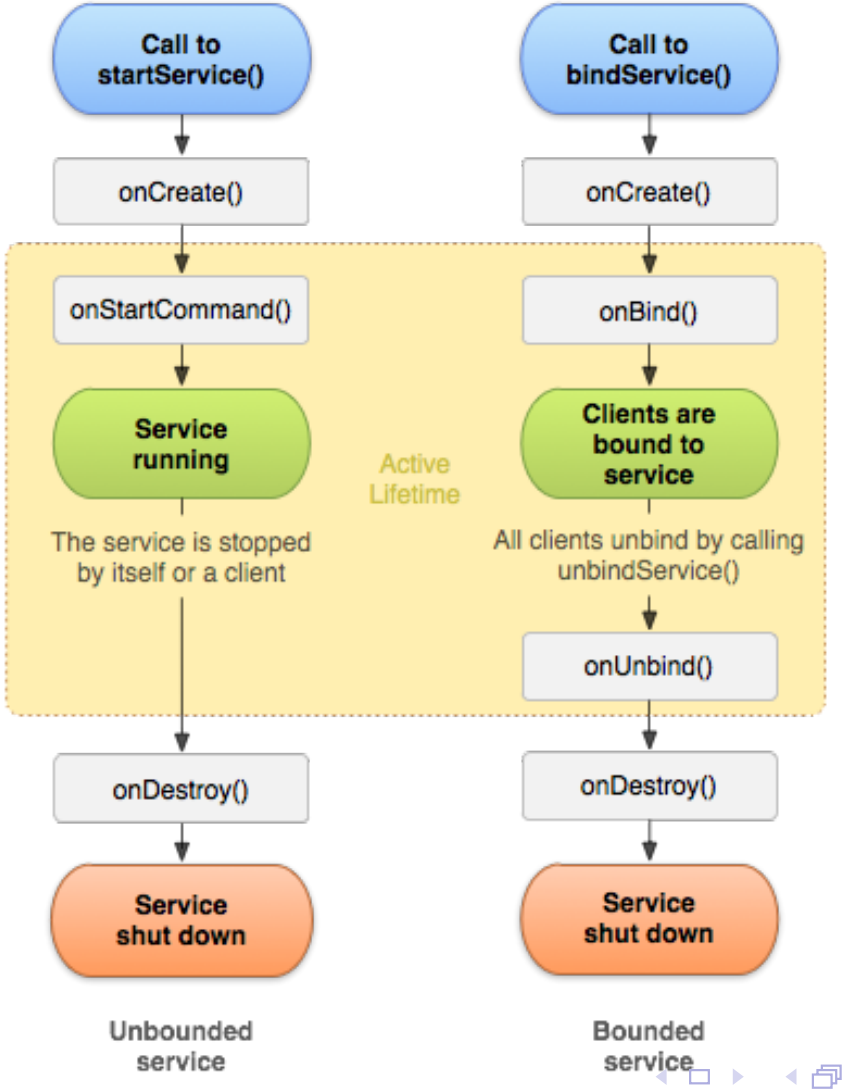
调用stopForeground(boolean)方法[14]停止前台服务。

[13] 使用该方法启动的服务仍为后台运行的服务。

[14] 该方法参数指明是否需要移除该服务对应的状态栏通知。

服务的生命周期 I

服务的生命周期要比活动的生命周期简单得多。对服务生命周期的管理主要集中在服务的创建及销毁上。



服务的生命周期 II

启动服务 (started service) 的生命周期：

- 服务由其他组件通过 startService ()方法启动。
- 如果服务实例不存在，则执行onCreate()方法创建服务实例；
- 否则执行onStartCommand()方法；
- onStartCommand()方法执行完成后，服务继续在后台运行，直到其自身调用 stopSelf ()或由其他组件调用 stopService ()方法停止。

服务的生命周期 III

绑定服务 (bound service) 的生命周期：

- 服务由其他组件通过bindService()方法启动。
- 如果服务实例不存在，则执行onCreate()方法创建服务实例；
- 否则执行onBind()方法，绑定服务的组件 (称为客户端, client) 该方法返回的 *IBinder* 对象与服务进行通信；
- 客户端通过unbindService()方法解除与服务的绑定。
- 多个客户端可与同一服务实例进行绑定，当所有绑定的服务的组件都解除绑定后，系统将销毁服务。

服务的生命周期 III

绑定服务 (bound service) 的生命周期：

- 服务由其他组件通过bindService()方法启动。
- 如果服务实例不存在，则执行onCreate()方法创建服务实例；
- 否则执行onBind()方法，绑定服务的组件 (称为客户端, client) 该方法返回的 *IBinder* 对象与服务进行通信；
- 客户端通过unbindService()方法解除与服务的绑定。
- 多个客户端可与同一服务实例进行绑定，当所有绑定的服务的组件都解除绑定后，系统将销毁服务。

启动服务与绑定服务实例的生命周期并不是完全独立的。无论服务如何启动，客户端均可通过绑定形式绑定该服务。

Service 使用多线程处理并发请求示例 I

```
1  public class HelloService extends Service {
2    private Looper mServiceLooper;
3    private ServiceHandler mServiceHandler;
4
5    // Handler that receives messages from the thread
6    private final class ServiceHandler extends Handler {
7        public ServiceHandler(Looper looper) {
8            super(looper);
9        }
10       @Override
11       public void handleMessage(Message msg) {
12           // Normally we would do some work here, like download a file.
13           // For our sample, we just sleep for 5 seconds.
14           try {
15               Thread.sleep(5000);
16           } catch (InterruptedException e) {
17               // Restore interrupt status.
18               Thread.currentThread().interrupt();
19           }
20           // Stop the service using the startId, so that we don't stop
21           // the service in the middle of handling another job
```

Service 使用多线程处理并发请求示例 II

```
22              stopSelf(msg.arg1);
23          }
24      }
25
26      @Override
27      public void onCreate() {
28        // Start up the thread running the service. Note that we create a
29        // separate thread because the service normally runs in the process's
30        // main thread, which we don't want to block. We also make it
31        // background priority so CPU-intensive work doesn't disrupt our UI.
32        HandlerThread thread = new HandlerThread("ServiceStartArguments",
33                Process.THREAD_PRIORITY_BACKGROUND);
34        thread.start();
35
36        // Get the HandlerThread's Looper and use it for our Handler
37        mServiceLooper = thread.getLooper();
38        mServiceHandler = new ServiceHandler(mServiceLooper);
39      }
40
41      @Override
42      public int onStartCommand(Intent intent, int flags, int startId) {
43          Toast.makeText(this, "service␣starting", Toast.LENGTH_SHORT).show();
44
```

Service 使用多线程处理并发请求示例 III

```
45          // For each start request, send a message to start a job and deliver the
46          // start ID so we know which request we're stopping when we finish the job
47          Message msg = mServiceHandler.obtainMessage();
48          msg.arg1 = startId;
49          mServiceHandler.sendMessage(msg);
50
51          // If we get killed, after returning from here, restart
52          return START_STICKY;
53      }
54
55      @Override
56      public IBinder onBind(Intent intent) {
57          // We don't provide binding, so return null
58          return null;
59      }
60
61      @Override
62      public void onDestroy() {
63        Toast.makeText(this, "service␣done", Toast.LENGTH_SHORT).show();
64      }
65  }
```