服务

本文内容

关键类

- `Service`
- `IntentService`

示例

- `ServiceStartArguments`
- `LocalService`

另请参阅

- 绑定服务

`Service` 是一个可以在后台执行长时间运行操作而不提供用户界面的应用组件。服务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，组件可以绑定到服务，以与之进行交互，甚至是执行进程间通信 (IPC)。例如，服务可以处理网络事务、播放音乐，执行文件 I/O 或与内容提供程序交互，而所有这一切均可在后台进行。

服务基本上分为两种形式：

启动

> 当应用组件（如 Activity）通过调用 `startService()` 启动服务时，服务即处于“启动”状态。一旦启动，服务即可在后台无限期运行，即使启动服务的组件已被销毁也不受影响。 已启动的服务通常是执行单一操作，而且不会将结果返回给调用方。例如，它可能通过网络下载或上传文件。 操作完成后，服务会自行停止运行。

绑定

> 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。绑定服务提供了一个客户端-服务器接口，允许组件与服务进行交互、发送请求、获取结果，甚至是利用进程间通信 (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。 多个组件可以同时绑定到该服务，但全部取消绑定后，该服务即会被销毁。

虽然本文档是分开概括讨论这两种服务，但是您的服务可以同时以这两种方式运行，也就是说，它既可以是启动服务（以无限期运行），也允许绑定。问题只是在于您是否实现了一组回调方法：`onStartCommand()`（允许组件启动服务）和 `onBind()`（允许绑定服务）。

无论应用是处于启动状态还是绑定状态，抑或处于启动并且绑定状态，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，您可以通过清单文件将服务声明为私有服务，并阻止其他应用访问。 使用清单文件声明服务部分将对此做更详尽的阐述。

> **注意：**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（除非另行指定）。 这意味着，如果服务将执行任何 CPU 密集型工作或阻止性操作（例如 MP3 播放或联网），则应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“应用无响应”(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。

基础知识

要创建服务，您必须创建 `Service` 的子类（或使用它的一个现有子类）。在实现中，您需要重写一些回调方法，以处理服务生命周期的某些关键方面并提供一种机制将组件绑定到服务（如适用）。 应重写的最重要的回调方法包括：

`onStartCommand()`

当另一个组件（如 Activity）通过调用 `startService()` 请求启动服务时，系统将调用此方法。一旦执行此方法，服务即会启动并可在后台无限期运行。 如果您实现此方法，则在服务工作完成后，需要由您通过调用 `stopSelf()` 或 `stopService()` 来停止服务。（如果您只想提供绑定，则无需实现此方法。）

`onBind()`

当另一个组件想通过调用 `bindService()` 与服务绑定（例如执行 RPC）时，系统将调用此方法。在此方法的实现中，您必须通过返回 `IBinder` 提供一个接口，供客户端用来与服务进行通信。请务必实现此方法，但如果您并不希望允许绑定，则应返回 null。

`onCreate()`

首次创建服务时，系统将调用此方法来执行一次性设置程序（在调用 `onStartCommand()` 或 `onBind()` 之前）。如果服务已在运行，则不会调用此方法。

`onDestroy()`

当服务不再使用且将被销毁时，系统将调用此方法。服务应该实现此方法来清理所有资源，如线程、注册的侦听器、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（这会导致对 `onStartCommand()` 的调用），则服务将一直运行，直到服务使用 `stopSelf()` 自行停止运行，或由其他组件通过调用 `stopService()` 停止它为止。

如果组件是通过调用 `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。一旦该服务与所有客户端之间的绑定全部取消，系统便会销毁它。

仅当内存过低且必须回收系统资源以供具有用户焦点的 Activity 使用时，Android 系统才会强制停止服务。如果将服务绑定到具有用户焦点的 Activity，则它不太可能会终止；如果将服务声明为在前台运行（稍后讨论），则它几乎永远不会终止。或者，如果服务已启动并要长时间运行，则系统会随着时间的推移降低服务在后台任务列表中的位置，而服务也将随之变得非常容易被终止；如果服务是启动服务，则您必须将其设计为能够妥善处理系统对它的重启。 如果系统终止服务，那么一旦资源变得再次可用，系统便会重启服务（不过这还取决于从 `onStartCommand()` 返回的值，本文稍后会对此加以讨论）。如需了解有关系统会在何时销毁服务的详细信息，请参阅进程和线程文档。

在下文中，您将了解如何创建各类服务以及如何从其他应用组件使用服务。

您应使用服务还是线程？

简单地说，服务是一种即使用户未与应用交互也可在后台运行的组件。 因此，您应仅在必要时才创建服务。

如需在主线程外部执行工作，不过只是在用户正在与应用交互时才有此需要，则应创建新线程而非服务。 例如，如果您只是想在 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，然后在 `onStop()` 中停止线程。您还可以考虑使用 `AsyncTask` 或 `HandlerThread`，而非传统的 `Thread` 类。如需了解有关线程的详细信息，请参阅进程和线程文档。

请记住，如果您确实要使用服务，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止性操作，则您仍应在服务内创建新线程。

使用清单文件声明服务

如同 Activity（以及其他组件）一样，您必须在应用的清单文件中声明所有服务。

要声明服务，请添加 `<service>` 元素作为 `<application>` 元素的子元素。例如：

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

如需了解有关使用清单文件声明服务的详细信息，请参阅 `<service>` 元素参考文档。

您还可将其他属性包括在 `<service>` 元素中，以定义一些特性，如启动服务及其运行所在进程所需的权限。`android:name` 属性是唯一必需的属性，用于指定服务的类名。应用一旦发布，即不应更改此类名，如若不然，可能会存在因依赖显式 Intent 启动或绑定服务而破坏代码的风险（请阅读博客文章Things That Cannot Change[不能更改的内容]）。

为了确保应用的安全性，**请始终使用显式 Intent 启动或绑定** `Service`，且不要为服务声明 Intent 过滤器。 启动哪个服务存在一定的不确定性，而如果对这种不确定性的考量非常有必要，则可为服务提供 Intent 过滤器并从 `Intent` 中排除相应的组件名称，但随后必须使用 `setPackage()` 方法设置 Intent 的软件包，这样可以充分消除目标服务的不确定性。

此外，还可以通过添加 `android:exported` 属性并将其设置为 `"false"`，确保服务仅适用于您的应用。这可以有效阻止其他应用启动您的服务，即便在使用显式 Intent 时也如此。

创建启动服务

启动服务由另一个组件通过调用 `startService()` 启动，这会导致调用服务的 `onStartCommand()` 方法。

服务启动之后，其生命周期即独立于启动它的组件，并且可以在后台无限期地运行，即使启动服务的组件已被销毁也不受影响。 因此，服务应通过调用 `stopSelf()` 结束工作来自行停止运行，或者由另一个组件通过调用 `stopService()` 来停止它。

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

例如，假设某 Activity 需要将一些数据保存到在线数据库中。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。服务通过 `onStartCommand()` 接收 Intent，连接到互联网并执行数据库事务。事务完成之后，服务会自行停止运行并随即被销毁。

> **注意：**默认情况下，服务与服务声明所在的应用运行于同一进程，而且运行于该应用的主线程中。 因此，如果服务在用户与来自同一应用的 Activity 进行交互时执行密集型或阻止性操作，则会降低 Activity 性能。 为了避免影响应用性能，您应在服务内启动新线程。

从传统上讲，您可以扩展两个类来创建启动服务：

`Service`

> 这是适用于所有服务的基类。扩展此类时，必须创建一个用于执行所有服务工作的新线程，因为默认情况下，服务将使用应用的主线程，这会降低应用正在运行的所有 Activity 的性能。

`IntentService`

> 这是 `Service` 的子类，它使用工作线程逐一处理所有启动请求。如果您不要求服务同时处理多个请求，这是最好的选择。 您只需实现 `onHandleIntent()` 方法即可，该方法会接收每个启动请求的 Intent，使您能够执行后台工作。

下文介绍如何使用其中任一个类来实现服务。

扩展 IntentService 类

由于大多数启动服务都不必同时处理多个请求（实际上，这种多线程情况可能很危险），因此使用 `IntentService` 类实现服务也许是最好的选择。

`IntentService` 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，这样您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，因此您永远不必调用 `stopSelf()`。
- 提供 `onBind()` 的默认实现（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。

综上所述，您只需实现 `onHandleIntent()` 来完成客户端提供的工作即可。（不过，您还需要为服务提供小型构造函数。）

以下是 `IntentService` 的实现示例：

```
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super IntentService(String)
   * 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();
      }
  }
}
```

您只需要一个构造函数和一个 `onHandleIntent()` 实现即可。

如果您决定还重写其他回调方法（如 `onCreate()`、`onStartCommand()` 或 `onDestroy()`），请确保调用超类实现，以便 `IntentService` 能够妥善处理工作线程的生命周期。

例如，`onStartCommand()` 必须返回默认实现（即，如何将 Intent 传递给 `onHandleIntent()`）：

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

除 `onHandleIntent()` 之外，您无需从中调用超类的唯一方法就是 `onBind()`（仅当服务允许绑定时，才需要实现该方法）。

在下一部分中，您将了解如何在扩展 `Service` 基类时实现同类服务。该基类包含更多代码，但如需同时处理多个启动请求，则更适合使用该基类。

扩展服务类

正如上一部分中所述，使用 `IntentService` 显著简化了启动服务的实现。但是，若要求服务执行多线程（而不是通过工作队列处理启动请求），则可扩展 `Service` 类来处理每个 Intent。

为了便于比较，以下提供了 `Service` 类实现的代码示例，该类执行的工作与上述使用 `IntentService` 的示例完全相同。也就是说，对于每个启动请求，它均使用工作线程执行作业，且每次仅处理一个请求。

```
public class HelloService extends Service {
  private Looper mServiceLooper;
  private ServiceHandler mServiceHandler;

  // 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 will not 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
    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
  }

  @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 = mServiceHandler.obtainMessage();
      msg.arg1 = startId;
      mServiceHandler.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();
  }
}
```

正如您所见，与使用 `IntentService` 相比，这需要执行更多工作。

但是，因为是由您自己处理对 `onStartCommand()` 的每个调用，因此可以同时执行多个请求。此示例并未这样做，但如果您希望如此，则可为每个请求创建一个新线程，然后立即运行这些线程（而不是等待上一个请求完成）。

请注意，`onStartCommand()` 方法必须返回整型数。整型数是一个值，用于描述系统应该如何在服务终止的情况下继续运行服务（如上所述，`IntentService` 的默认实现将为您处理这种情况，不过您可以对其进行修改）。从 `onStartCommand()` 返回的值必须是以下常量之一：

`START_NOT_STICKY`

> 如果系统在 `onStartCommand()` 返回后终止服务，则除非有挂起 Intent 要传递，否则系统*不会*重建服务。这是最安全的选项，可以避免在不必要时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（在这种情况下，将传递这些 Intent），否则系统会通过空 Intent 调用 `onStartCommand()`。这适用于不执行命令、但无限期运行并等待作业的媒体播放器（或类似服务）。

`START_REDELIVER_INTENT`

> 如果系统在 `onStartCommand()` 返回后终止服务，则会重建服务，并通过传递给服务的最后一个 Intent 调用 `onStartCommand()`。任何挂起 Intent 均依次传递。这适用于主动执行应该立即恢复的作业（例如下载文件）的服务。

有关这些返回值的更多详细信息，请查阅每个常量链接的参考文档。

启动服务

您可以通过将 `Intent`（指定要启动的服务）传递给 `startService()`，从 Activity 或其他应用组件启动服务。Android 系统调用服务的 `onStartCommand()` 方法，并向其传递 `Intent`。（切勿直接调用 `onStartCommand()`。）

例如，Activity 可以结合使用显式 Intent 与 `startService()`，启动上文中的示例服务 (`HelloService`)：

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

`startService()` 方法将立即返回，且 Android 系统调用服务的 `onStartCommand()` 方法。如果服务尚未运行，则系统会先调用 `onCreate()`，然后再调用 `onStartCommand()`。

如果服务亦未提供绑定，则使用 `startService()` 传递的 Intent 是应用组件与服务之间唯一的通信模式。但是，如果您希望服务返回结果，则启动服务的客户端可以为广播创建一个 `PendingIntent`（使用 `getBroadcast()`），并通过启动服务的 `Intent` 传递给服务。然后，服务就可以使用广播传递结果。

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，要停止服务，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。也就是说，除非系统必须回收内存资源，否则系统不会停止或销毁服务，而且服务在 `onStartCommand()` 返回后会继续运行。因此，服务必须通过调用 `stopSelf()` 自行停止运行，或者由另一个组件通过调用 `stopService()` 来停止它。

一旦请求使用 `stopSelf()` 或 `stopService()` 停止服务，系统就会尽快销毁服务。

但是，如果服务同时处理多个 `onStartCommand()` 请求，则您不应在处理完一个启动请求之后停止服务，因为您可能已经收到了新的启动请求（在第一个请求结束时停止服务会终止第二个请求）。为了避免这一问题，您可以使用 `stopSelf(int)` 确保服务停止请求始终基于最近的启动请求。也就说，在调用 `stopSelf(int)` 时，传递与停止请求的 ID 对应的启动请求的 ID（传递给 `onStartCommand()` 的 `startId`）。然后，如果在您能够调用 `stopSelf(int)` 之前服务收到了新的启动请求，ID 就不匹配，服务也就不会停止。

> **注意：**为了避免浪费系统资源和消耗电池电量，应用必须在工作完成之后停止其服务。 如有必要，其他组件可以通过调用 `stopService()` 来停止服务。即使为服务启用了绑定，一旦服务收到对 `onStartCommand()` 的调用，您始终仍须亲自停止服务。

如需了解有关服务生命周期的详细信息，请参阅下面有关**管理服务生命周期**的部分。

创建绑定服务

绑定服务允许应用组件通过调用 `bindService()` 与其绑定，以便创建长期连接（通常不允许组件通过调用 `startService()` 来*启动*它）。

如需与 Activity 和其他应用组件中的服务进行交互，或者需要通过进程间通信 (IPC) 向其他应用公开某些应用功能，则应创建绑定服务。

要创建绑定服务，必须实现 `onBind()` 回调方法以返回 `IBinder`，用于定义与服务通信的接口。然后，其他应用组件可以调用 `bindService()` 来检索该接口，并开始对服务调用方法。服务只用于与其绑定的应用组件，因此如果没有组件绑定到服务，则系统会销毁服务（您*不必*按通过 `onStartCommand()` 启动的服务那样来停止绑定服务）。

要创建绑定服务，首先必须定义指定客户端如何与服务通信的接口。 服务与客户端之间的这个接口必须是 `IBinder` 的实现，并且服务必须从 `onBind()` 回调方法返回它。一旦客户端收到 `IBinder`，即可开始通过该接口与服务进行交互。

多个客户端可以同时绑定到服务。客户端完成与服务的交互后，会调用 `unbindService()` 取消绑定。一旦没有客户端绑定到该服务，系统就会销毁它。

有多种方法实现绑定服务，其实现比启动服务更为复杂，因此绑定服务将在有关绑定服务的单独文档中专门讨论。

向用户发送通知

一旦运行起来，服务即可使用 Toast 通知或状态栏通知来通知用户所发生的事件。

Toast 通知是指出现在当前窗口的表面、片刻随即消失不见的消息，而状态栏通知则在状态栏中随消息一起提供图标，用户可以选择该图标来采取操作（例如启动 Activity）。

通常，当某些后台工作已经完成（例如文件下载完成）且用户现在可以对其进行操作时，状态栏通知是最佳方法。 当用户从展开视图中选定通知时，通知即可启动 Activity（例如查看已下载的文件）。

如需了解详细信息，请参阅 Toast 通知或状态栏通知开发者指南。

在前台运行服务

前台服务被认为是用户主动意识到的一种服务，因此在内存不足时，系统也不会考虑将其终止。 前台服务必须为状态栏提供通知，放在“正在进行”标题下方，这意味着除非服务停止或从前台移除，否则不能清除通知。

例如，应该将通过服务播放音乐的音乐播放器设置为在前台运行，这是因为用户明确意识到其操作。 状态栏中的通知可能表示正在播放的歌曲，并允许用户启动 Activity 来与音乐播放器进行交互。

要请求让服务运行于前台，请调用 `startForeground()`。此方法采用两个参数：唯一标识通知的整型数和状态栏的 `Notification`。例如：

```
Notification notification = new Notification(R.drawable.icon, getText(R.string.ticker_text),
        System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);
notification.setLatestEventInfo(this, getText(R.string.notification_title),
        getText(R.string.notification_message), pendingIntent);
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

> **注意：**提供给 `startForeground()` 的整型 ID 不得为 0。

要从前台移除服务，请调用 `stopForeground()`。此方法采用一个布尔值，指示是否也移除状态栏通知。 此方法*不会*停止服务。 但是，如果您在服务正在前台运行时将其停止，则通知也会被移除。

如需了解有关通知的详细信息，请参阅创建状态栏通知。

管理服务生命周期

服务的生命周期比 Activity 的生命周期要简单得多。但是，密切关注如何创建和销毁服务反而更加重要，因为服务可以在用户没有意识到的情况下运行于后台。

服务生命周期（从创建到销毁）可以遵循两条不同的路径：

- 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

  该服务在另一个组件（客户端）调用 `bindService()` 时创建。然后，客户端通过 `IBinder` 接口与服务进行通信。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 *不必*自行停止运行。）

这两条路径并非完全独立。也就是说，您可以绑定到已经使用 `startService()` 启动的服务。例如，可以通过使用 `Intent`（标识要播放的音乐）调用 `startService()` 来启动后台音乐服务。随后，可能在用户需要稍加控制播放器或获取有关当前播放歌曲的信息时，Activity 可以通过调用 `bindService()` 绑定到服务。在这种情况下，除非所有客户端均取消绑定，否则 `stopService()` 或 `stopSelf()` 不会实际停止服务。

实现生命周期回调

与 Activity 类似，服务也拥有生命周期回调方法，您可以实现这些方法来监控服务状态的变化并适时执行工作。以下框架服务展示了每种生命周期方法：

```
public class ExampleService extends Service {
    int mStartMode;       // indicates how to behave if the service is killed
    IBinder mBinder;      // interface for clients that bind
    boolean mAllowRebind; // indicates whether onRebind should be used

    @Override
    public void onCreate() {
        // The service is being created
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // The service is starting, due to a call to startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
        return mBinder;
    }
    @Override
    public boolean onUnbind(Intent intent) {
        // All clients have unbound with unbindService()
        return mAllowRebind;
    }
    @Override
    public void onRebind(Intent intent) {
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }
    @Override
    public void onDestroy() {
        // The service is no longer used and is being destroyed
    }
}
```

> **注：**与 Activity 生命周期回调方法不同，您*不*需要调用这些回调方法的超类实现。

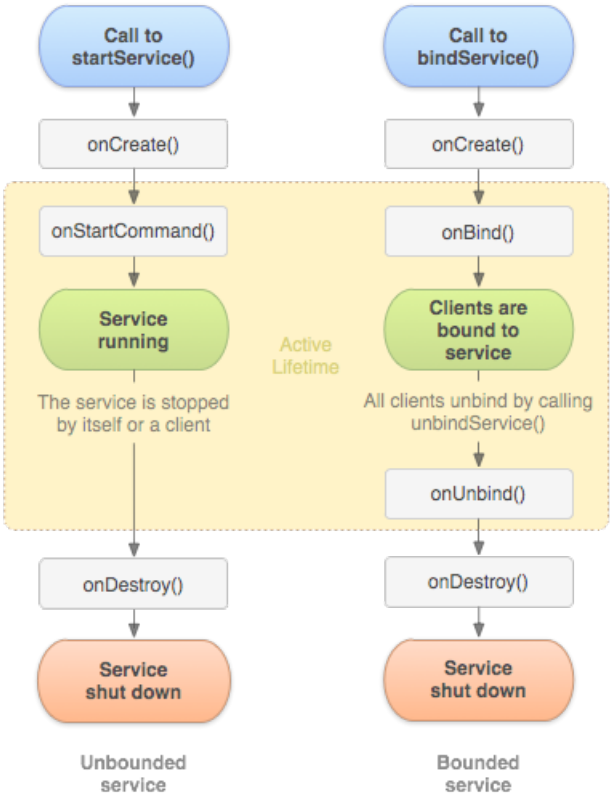

图 2. 服务生命周期。左图显示了使用 `startService()` 所创建的服务的生命周期，右图显示了使用 `bindService()` 所创建的服务的生命周期。

通过实现这些方法，您可以监控服务生命周期的两个嵌套循环：

- 服务的**整个生命周期**从调用 `onCreate()` 开始起，到 `onDestroy()` 返回时结束。与 Activity 类似，服务也在 `onCreate()` 中完成初始设置，并在 `onDestroy()` 中释放所有剩余资源。例如，音乐播放服务可以在 `onCreate()` 中创建用于播放音乐的线程，然后在 `onDestroy()` 中停止该线程。

  无论服务是通过 `startService()` 还是 `bindService()` 创建，都会为所有服务调用 `onCreate()` 和 `onDestroy()` 方法。

- 服务的**有效生命周期**从调用 `onStartCommand()` 或 `onBind()` 方法开始。每种方法均有 {`Intent` 对象，该对象分别传递到 `startService()` 或 `bindService()`。

  对于启动服务，有效生命周期与整个生命周期同时结束（即便是在 `onStartCommand()` 返回之后，服务仍然处于活动状态）。对于绑定服务，有效生命周期在 `onUnbind()` 返回时结束。

> **注：**尽管启动服务是通过调用 `stopSelf()` 或 `stopService()` 来停止，但是该服务并无相应的回调（没有 `onStop()` 回调）。因此，除非服务绑定到客户端，否则在服务停止时，系统会将其销毁 — `onDestroy()` 是接收到的唯一回调。

图 2 说明了服务的典型回调方法。尽管该图分开介绍通过 `startService()` 创建的服务和通过 `bindService()` 创建的服务，但是请记住，不管启动方式如何，任何服务均有可能允许客户端与其绑定。因此，最初使用 `onStartCommand()`（通过客户端调用 `startService()`）启动的服务仍可接收对 `onBind()` 的调用（当客户端调用 `bindService()` 时）。

如需了解有关创建提供绑定的服务的详细信息，请参阅绑定服务文档，该文档的管理绑定服务的生命周期部分提供了有关 `onRebind()` 回调方法的更多信息。