第 8 天 Android 基础

第八章 Service

- 了解 Service
- Android 中的进程
- 电话窃听器
- Service 的生命周期
- aidl 的使用

1.1 Service 的基本概念

Android 官方文档对 Service 的解释如下：

```
A Service is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.
```

翻译：

Service 是应用组件之一，用来执行需要长时间运行的操作，Service 运行在后台并且跟用户没有交互界面。第三方应用的组件（指四大组件的任意一种）可以启动一个 Service，一旦启动起来，该 Service 就一直在后台运行，就算用户已经将该应用置为后台进程。另外，组件也可以通过绑定的形式跟一个 Service 进行交互，甚至完成进程间通信。比如：Service 可以在后台处理网络传输、播放音乐、进行 I/O 流的读写或者跟内容提供者进行交互。

Service 的特点如下：

- Service 在 Android 中是一种长生命周期的组件，它不实现任何用户界面，是一个没有界面的 Activity
- Service 长期在后台运行，执行不关乎界面的一些操作比如：网易新闻服务，每隔 1 分钟去服务查看是否有最新新闻
- Service 和 Thread 有点相似，但是使用 Thread 不安全，不严谨
- Service 和其他组件一样，都是运行在主线程中，因此不能用它来做耗时的操作

1.2 Android 中的进程

1.2.1 Android 中进程的种类

在 Android 中进程优先级由高到低，依次分为 Foreground process、Visible process、Service process、Background process、Empty process。

Android 官方文档的解释如下：

1. Foreground process 前台进程

```
A process that is required for what the user is currently doing. A process is considered
to be in the foreground if any of the following conditions are true:
   1. It hosts an Activity that the user is interacting with (the Activity's onResume()
   method has been called).
   2. It hosts a Service that's bound to the activity that the user is interacting with.
   3. It hosts a Service that's running "in the foreground"—the service has
   called startForeground().
   4. It hosts a Service that's executing one of its lifecycle callbacks (onCreate(),
   onStart(), or onDestroy()).
   5. It hosts a BroadcastReceiver that's executing its onReceive() method.

Generally, only a few foreground processes exist at any given time. They are killed only
as a last resort—if memory is so low that they cannot all continue to run.

Generally, at that point, the device has reached a memory paging state, so killing
some foreground processes is required to keep the user interface responsive.
```

翻译：

进程是用户执行当前任务所需要的。下面任何一个条件满足了，那么当前进程就视为前台进程：

1. 拥有一个正在和用户交互的 Activity（也就是说 Activity 的 onResume（）方法被执行了）。
2. 拥有一个被用户的正在交互的 Activity 绑定的 Service。
3. 拥有一个以“前台模式”运行的 Service--该 Service 已经被调用了 startForeground()方法
4. 拥有一个正在执行生命周期方法的 Service（onCreate()、onStart()、onDestory()）
5. 拥有一个 BroadCastReceiver 并且正在执行 onReceive()方法

通常情况下，在任何时候系统只存在一小部分前台进程。这些进程只会作为最后的手段才会被杀死-当内存不足以继续运行他们的时候。

通常情况下，在这个时刻，设备已经达到内存分页状态（当系统达到内存分页状态时只能通过虚拟地址访问内存，是不是很不好理解？那大家就理解为达到这个状态时系统已经无法继续分配新的内存空间即可），因此杀死一些前台进程（释放内存空间）以保证应用能够继续响应用户的交互是必要的手段。

2. Visible process 可视进程，可以看见，但不可以交互

```
A process that doesn't have any foreground components, but still can affect what the
user sees on screen. A process is considered to be visible if either of the following
conditions are true:
   1. It hosts an Activity that is not in the foreground, but is still visible to the
user(its onPause() method has been called). This might occur, for example, if the
   foreground activity started a dialog, which allows the previous activity to be seen
```

```
   behind it.
   2. It hosts a Service that's bound to a visible (or foreground) activity.

A visible process is considered extremely important and will not be killed unless doing
so is required to keep all foreground processes running.
```

翻译：

一个不拥有任何前台组件的进程，但是依然可以影响用户在屏幕上看见的控件。如下任何条件之一成立则认为是可视进程：

1. 拥有一个不是前台的 Activity（onPause 方法被调用），但是对用户依然可见。这种情况（处于暂停状态的 Activity）发生，比如，如果前台 Activity 打开了一个 dialog，该 dialog 下面依然可见 Activity。

2. 拥有一个被可见或者前台 Activity 绑定的 Service。

一个可视进程被认为是极其重要的并且一般不会被系统杀死，除非为了保证所有的前台进程去运行不得已为之。

3. Service process 服务进程

```
A process that is running a service that has been started with the startService() method
and does not fall into either of the two higher categories.
Although service processes are not directly tied to anything the user sees, they are
generally doing things that the user cares about (such as playing music in the
background or downloading data on the network), so the system keeps them running unless
there's not enough memory to retain them along with all foreground and visible processes.
```

翻译：

一个拥有正在运行的 Service，并且该 Service 是被 startService()方法启动起来的进程，并且该进程没有被归类到前面的两种（前置进程和可视进程）类型，那么该进程就是服务进程。

尽管服务进程没有与用户可见的控件直接绑定，但是这些进程干的工作依然是用户关心的（比如在后台播放音乐或者从网络上下载数据），因此系统保留这些进程一直运行除非系统没有足够的内存去运行前台进程和可视进程。

4. Background process 后台进程

```
A process holding an activity that's not currently visible to the user (the activity's
onStop() method has been called).
These processes have no direct impact on the user experience, and the system can kill
them at any time to reclaim memory for a foreground, visible,or service process.
Usually there are many background processes running, so they are kept in an LRU (least
recently used) list to ensure that the process with the activity that was most recently
seen by the user is the last to be killed.
If an activity implements its lifecycle methods correctly, and saves its current state,
killing its process will not have a visible effect on the user experience, because when
the user navigates back to the activity, the activity restores all of its visible state.
See the Activities document for information about saving and restoring state.
```

翻译：

一个拥有对用户不可见的 Activity（该 Activity 已经被执行了 onStop()方法）进程叫做后台进程。

后台进程对用户体验没有直接的影响，并且系统会在任何需要为前台进程，可视进程，或服务进程申请内存的时候杀死后台进程。

通常系统中运行着大量的后台进程，这些后台进程保存在一个 LRU（最少最近使用的）列表中，使用 LRU 规则是为了保证让最近被用户使用的 Activity 进程最后被杀死（就是谁最近被使用了，谁最后再被杀

死）。

如果一个 Activity 正确实现了它的生命周期方法，并且保存了它的状态（通常这个状态是系统自动保存的），那么当系统杀死它的进程的时候是对用户的体验没有看得见的影响的，因为当用户导航到之前的 Activity 的时候，这个 Activity 会自动恢复之前保存的视图状态。查看 Activity 文档去获取更多关于 Activity 状态的保存和恢复信息。

5. Empty process 空进程(当程序退出时，进程没有被销毁，而是变成了空进程)

```
A process that doesn't hold any active application components. The only reason to keep
this kind of process alive is for caching purposes, to improve startup time the next
time a component needs to run in it. The system often kills these processes in order to
balance overall system resources between process caches and the underlying kernel caches.
```

翻译：

不拥有任何系统组件（指四大组件）的进程叫空进程。保持空进程存活的唯一理由是为了缓存，这样可以提高下次启动组件的打开速度。当系统需要维持缓存进程和底层内核缓存的资源均衡的时候系统经常会（随时会）杀死该类进程。

1.2.2 进程的回收机制

Android 系统有一套内存回收机制,会根据优先级进行回收。Android 系统会尽可能的维持程序的进程,但是终究还是需要回收一些旧的进程节省内存提供给新的或者重要的进程使用。

进程的回收顺序是：从低到高

1. 当系统内存不够用时，会把空进程一个一个回收掉
2. 当系统回收所有的完空进程不够用时，继续向上回收后台进程，依次类推
3. 当系统回收所有的完空进程不够用时，继续向上回收后台进程，依次类推
4. 但是当回收服务，可视，前台这三种进程时，系统非必要情况下不会轻易回收，如果需要回收掉这三种进程，那么在系统内存够用时，会再给重新启动进程;但是服务进程如果用户手动的关闭服务，这时服务不会再重启了。

1.2.3 为什么用服务而不是线程

服务是运行在后台的进程，跟我们熟悉的 Thread 很像，Android 官方文档给出 Service 和 Thread 的区别如下：

【文件 1-1】 Should you use a service or a thread?

```
   A service is simply a component that can run in the background even when the user is
not interacting with your application. Thus, you should create a service only if that is
what you need.
   If you need to perform work outside your main thread, but only while the user is
interacting with your application, then you should probably instead create a new thread
and not a service. For example, if you want to play some music, but only while your
activity is running, you might create a thread in onCreate(), start running it in
onStart(), then stop it in onStop(). Also consider using AsyncTask or HandlerThread,
instead of the traditional Thread class. See the Processes and Threading document for
more information about threads.
   Remember that if you do use a service, it still runs in your application's main
```

```
thread by default, so you should still create a new thread within the service if it
performs intensive or blocking operations.
```

为了照顾到更多的同学，因此我将上面的内容翻译一下：

翻译：

Service 是 Android 中的（四大）组件之一，即使用户跟你的应用（不可见）没交互也依然运行在系统后台。因此，如果你真的需要这样的服务时才需要创建一个 Service。

如果你不需要在主线程中做一些工作，而是仅仅当应用跟用户交互的时候做一些工作，那么你就可以创建一个子线程而不是开启一个服务。比如，如果你想播放音乐，但是仅仅想当你的 Activity 处于运行状态时播放，那么你就可以在 onCreate()方法中创建个线程对象，然后在 onStart()方法中启动该线程，最后在 onStop()方法中停止（终止该线程）播放。当然也可以考虑使用 AsyncTask 和 HandlerThread 来替代传统的 Thread。可以查看 Processes and Threading 文档获取更多关于线程的内容。

在使用 Service 时一定要注意，Service 默认运行在应用的主线程中，因此如果你需要在 Service 中执行大量的或者阻塞（比如网络访问、IO 等）的任务，那么依然需要在 Service 中开启一个 Thread 来完成这些工作。

追加：

其实使用 Service 并不影响我们使用 Thread，而且很多情形下，我们都是在 Service 中开启子 Thread 的混合使用方式。

1.3 案例-电话窃听器

接下来我们共同完成一个案例来引入 Service 的使用。

1.3.1 需求

开启一个服务，在服务中监听用户接到的电话（注意是接收到来电而不是拨出去电话），当电话被接通时开始录音，电话挂断时停止录音。将录音数据保存在 sdcard 上。

监听电话需要在后台实现，因此需要将监听电话的代码逻辑在 Service 中运行。

1.3.2 代码

在该案例中用到两个 java 类，MainActivity 和自定义的 Service 类。

一、自定义 Service

在 Service 的生命周期 onStart()方法中开启电话状态的监听，当电话接通的时候开始录音，当电话挂断的时候停止录音。

【文件 1-2】 CallListenerService .java

```
1. package com.itheima.calllistener;
2.
3. import java.text.SimpleDateFormat;
4. import java.util.Date;
```

```
5.
6. import android.app.Service;
7. import android.content.Intent;
8. import android.media.MediaRecorder;
9. import android.media.MediaRecorder.AudioEncoder;
10. import android.media.MediaRecorder.AudioSource;
11. import android.media.MediaRecorder.OutputFormat;
12. import android.os.Environment;
13. import android.os.IBinder;
14. import android.telephony.PhoneStateListener;
15. import android.telephony.TelephonyManager;
16. import android.util.Log;
17. /**
18.  * 电话监听器 Service
19.  *
20.  * @author wzy 2015-11-23
21.  *
22.  */
23. public class CallListenerService extends Service {
24.
25. private TelephonyManager tm;
26. public MediaRecorder recorder;
27. public boolean isCalling;
28. @Override
29. public IBinder onBind(Intent intent) {
30.     return null;
31. }
32. @Override
33. public void onCreate() {
34.     super.onCreate();
35.     //获取电话管理器
36.     tm = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
37.     //开启监听
38.     /**
39.      * 参数 1：PhoneStateListener 对象
40.      * 参数 2：监听的内容，int 类型，该参数通常是常量，从 PhoneStateListener 中获取
41.      */
42.     tm.listen(new MyPhoneStateListener(), PhoneStateListener.LISTEN_CALL_STATE);
43. }
44. private class MyPhoneStateListener extends PhoneStateListener{
45.
46.     /**
47.      * 参数 1：电话的状态
48.      * 参数 2：打进来的电话号码
49.      */
```

```
50.     @Override
51.     public void onCallStateChanged(int state, String incomingNumber) {
52.         //如果状态变为接听状态
53.         if (TelephonyManager.CALL_STATE_OFFHOOK==state) {
54.             //输出日志
55.             Log.d("tag", "开始通话");
56.             //记录当前状态
57.             isCalling = true;
58.             //新建一个 MediaRecorder 对象
59.             recorder = new MediaRecorder();
60.             //设置声音来源
61.             recorder.setAudioSource(AudioSource.MIC);
62.             //设置输入格式
63.             recorder.setOutputFormat(OutputFormat.THREE_GPP);
64.             //格式化日期，作为文件名称
65.             SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd_hh_mm_ss");
66.             String date = format.format(new Date());
67.             //设置输出到的文件
68.         recorder.setOutputFile(Environment.getExternalStorageDirectory().
69. getAbsolutePath()+"/"+date+".3gp");
70.             //设置音频编码
71.             recorder.setAudioEncoder(AudioEncoder.DEFAULT);
72.             try {
73.                 //录音准备
74.                 recorder.prepare();
75.                 //录音开始
76.                 recorder.start();
77.             } catch (Exception e) {
78.                 e.printStackTrace();
79.             }
80.             //如果空闲状态，并且已经开启了录音
81.         }else if (TelephonyManager.CALL_STATE_IDLE==state&&isCalling) {
82.             //通知录音器
83.             recorder.stop();
84.             //释放录音器资源
85.             recorder.release();
86.             //设置状态为 false
87.             isCalling = false;
88.             Log.d("tag", "录音结束。");
89.         }
90.     }
91. }
92. }
93.
```

二、启动 Service

Service 的显示意图（其实并没有这个说法，Service 的启动分为 startService 和 bindService 两种启动方式，只不过第一种启动方式类似 Activity 的显示意图启动）启动，跟显示意图启动 Activity 几乎一模一样的，唯一的差别是方法名不一样。

【文件 1-3】 MainActivity.java

```
1. package com.itheima.calllistener;
2.
3. import android.os.Bundle;
4. import android.app.Activity;
5. import android.content.Intent;
6.
7. public class MainActivity extends Activity {
8.
9.  @Override
10. protected void onCreate(Bundle savedInstanceState) {
11.     super.onCreate(savedInstanceState);
12.     setContentView(R.layout.activity_main);
13.     /**
14.      * 使用意图启动一个 Service
15.      * 跟我们启动 Activity 是一样的
16.      */
17.     //创建一个 Intent 对象
18.     Intent service = new Intent();
19.     //设置 Context 和服务的字节码，使用显式意图
20.     service.setClass(this, CallListenerService.class);
21.     //启动服务
22.     startService(service);
23. }
24.
25. }
26.
```

1.3.3 在 AndroidManifest.xml 中注册

在该案例中保存录音文件到 sdcard、监听电话状态、录音都需要添加权限。

一、在 AndroidManifest.xml 中声明权限

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
```

二、注册 Service

```
<service android:name="com.itheima.calllistener.CallListenerService"/>
```

1.3.4 总结

将上面的代码部署到模拟器上（本人使用的是 Android4.2 的模拟器），然后打开 eclipse（ADT）的 DDMS 视图，找到 Emulator Control 在 Incoming number 中随便输入一个测试的号码，然后单选选中 Voice，然后点击 Call 按钮。如图 1-1 所示。

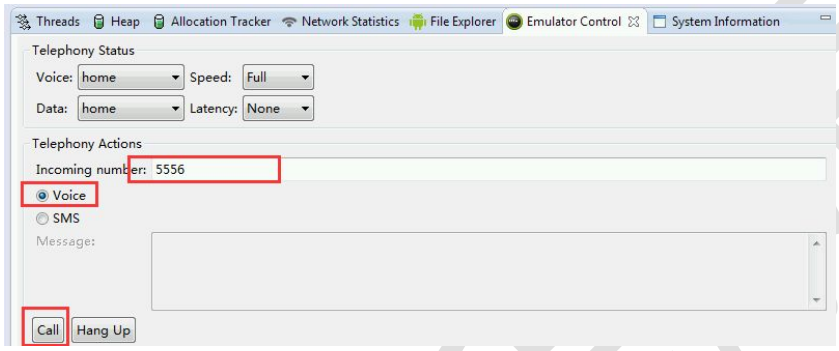

图 1-1　　给模拟器拨打电话

当模拟器响铃时就可以接听，然后可以模拟说几句话语再挂断。

挂断之后观察日志输出如图 1-2 所示。说明我们的代码已经正常执行。重新打开 DDMS，查看 File Explorer 选项，发现在 sdcard 下面已经成功保存了录音文件。

```
3336    com.itheima.calllistener    tag    开始通话
3336    com.itheima.calllistener    tag    录音结束。
```

图 1-2　　日志输出

图 1-3　　sdcard 中录音文件

将上面的语音文件导出来，然后就可以播放了，不过毕竟是模拟器录的音，录音的前面是杂音，后面才是我们真正的语音。

需要注意的是，可能部分同学在接收到电话的时候 adb 就会自动断掉，因此日志没有输出。这是部分模拟器的 bug，不需要管。

在该案例中我们学了较多的知识点，总结如下：

一、定义并启动一个 Service

自定义一个 Service 不妨起名 MyService 继承 Service。然后覆写 Service 的生命周期回调方法，这里因为我们的业务逻辑是当 Service 启动的时候就开始监听电话状态，因此只需要覆写 onCreate()方法就能满足需求了。

启动 MyService 的时候，只需通过 ContextWrapper.startService（Intent）方法即可。

二、监听电话的状态

1. 获取电话管理器

```
TelephoneManager tm = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
```

2. 开启监听

```
    /**
     * 参数 1：PhoneStateListener 对象
     * 参数 2：监听的内容，int 类型，该参数通常是常量，从 PhoneStateListener 中获取
     */
    tm.listen(new MyPhoneStateListener(), PhoneStateListener.LISTEN_CALL_STATE);
```

3. 自定义监听器

```
/**
 * 自定义电话状态监听器
 * @author wzy 2015-11-23
 *
 */
private class MyPhoneStateListener extends PhoneStateListener{


    /**
     * 参数 1：电话的状态
     * 参数 2：打进来的电话号码
     */
    @Override
    public void onCallStateChanged(int state, String incomingNumber) {
        ...省略非核心代码....
        }
```

在电话监听器中覆写 onCallStateChanged()方法，当电话的状态改变时该方法被系统回调。电话的状态有三种：

1. 空闲状态 TelephonyManager.CALL_STATE_IDLE
2. 响铃状态 TelephonyManager.CALL_STATE_RINGING
3. 摘机状态（接听状态）TelephonyManager.CALL_STATE_OFFHOOK

三、使用 MediaRecorder 实现录音功能

关于 MediaRecorder 的代码大家可以不用去死记硬背，记住个大概过程即可，官方文档已经给出了详细的使用说明和例子，只要我们用的时候（更何况在企业用的概率也不是很大）能够查看文档即可。

如图 1-4 所示是官方给出的 MediaRecorder 状态图。

除此之外官方给出的实例代码如文件 1-4 所示。

【文件 1-4】 MediaRecorder 示例代码

```
1. MediaRecorder recorder = new MediaRecorder();
2.  recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
3.  recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
4.  recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
```

```
5.  recorder.setOutputFile(PATH_NAME);
6.  recorder.prepare();
7.  recorder.start();   // Recording is now started
8.  ...
9.  recorder.stop();
10.  recorder.reset();   // You can reuse the object by going back to setAudioSource() step
11.  recorder.release(); // Now the object cannot be reused
12.
```

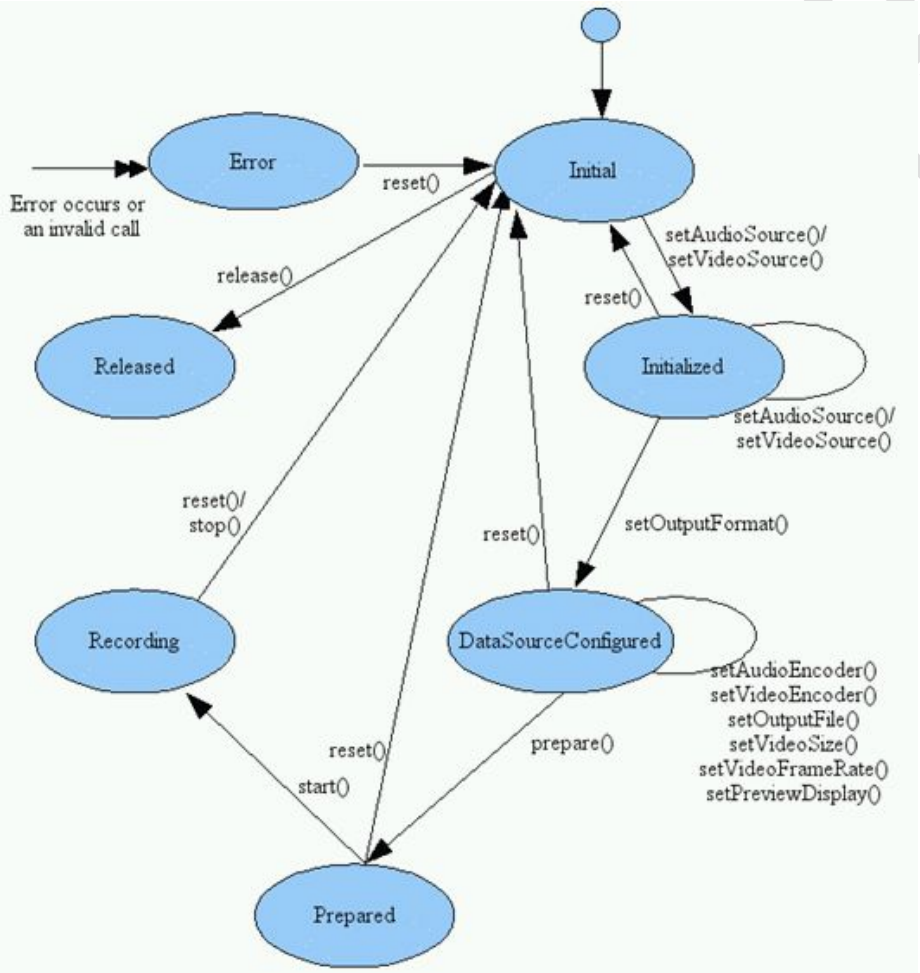

图 1-4　　MediaRecorder 状态图

1.4 Service 的生命周期

关于 Service 的生命周期在 Android 官方文档中有非常详细的说明，甚至比如何一本书籍都要好，因此这里建议英文水平好的直接看官方文档。

自己在学习 Service 以及写关于 Service 的文章时其实大量参考了官方的文档，因为那才是最权威的学

习资料。

跟 Activity 一样，Service 也是有生命周期的，不一样的是 Service 的生命周期，从它被创建开始，到它被销毁为止，可以有两条不同的路径：标准开启模式和绑定模式。

```
   The lifecycle of a service is much simpler than that of an activity. However, it's even
more important that you pay close attention to how your service is created and
destroyed, because a service can run in the background without the user being aware.
```

1.4.1 A started service（标准开启模式）

被开启的 service 通过其他组件调用 startService()被创建。这种 service 可以无限地运行下去，必须调用 stopSelf()方法或者其他组件调用 stopService()方法来停止它。当 service 被停止时，系统会销毁它。

```
   The service is created when another component calls startService(). The service then runs
indefinitely and must stop itself by calling stopSelf(). Another component can also stop the
service by calling stopService(). When the service is stopped, the system destroys it..
```

1.4.2 A bound service（绑定模式）

被绑定的 service 是当其他组件（一个客户）调用 bindService()来创建的。客户可以通过一个 IBinder 接口和 service 进行通信。客户可以通过 unbindService()方法来关闭这种连接。一个 service 可以同时和多个客户绑定，当多个客户都解除绑定之后，系统会销毁 service。

```
   The service is created when another component (a client) calls bindService(). The client
then communicates with the service through an IBinder interface. The client can close the
connection by calling unbindService(). Multiple clients can bind to the same service and when
all of them unbind, the system destroys the service. (The service does not need to stop itself.)
```

也就是说，你可以和一个已经调用了 startService()而被开启的 service 进行绑定。比如，一个后台音乐 service 可能因调用 startService()方法而被开启了，稍后，可能用户想要控制播放器或者得到一些当前歌曲的信息，可以通过 bindService()将一个 activity 和 service 绑定。这种情况下，stopService()或 stopSelf()实际上并不能停止这个 service，除非所有的客户都解除绑定。

```
   These two paths are not entirely separate. That is, you can bind to a service that was
already started with startService(). For example, a background music service could be started
by calling startService() with an Intent that identifies the music to play. Later, possibly
when the user wants to exercise some control over the player or get information about the current
song, an activity can bind to the service by calling bindService(). In cases like this,
stopService() or stopSelf() does not actually stop the service until all clients unbind.
```

1.4.3 Service 的生命周期回调函数

和 Activity 一样，service 也有一系列的生命周期回调函数，你可以实现它们来监测 service 状态的变化，并且在适当的时候执行适当的工作。

```
   Like an activity, a service has lifecycle callback methods that you can implement to monitor
changes in the service's state and perform work at the appropriate times. The following skeleton
service demonstrates each of the lifecycle methods:
```

下面的 service 展示了每一个生命周期的方法：

【文件 1-5】 Service 生命周期回调函数

```
1. public class TestService extends Service {
2.  int mStartMode; // indicates how to behave if the service is killed
3.  IBinder mBinder; // interface for clients that bind
4.  boolean mAllowRebind; // indicates whether onRebind should be used
5.
6.  @Override
7.  public void onCreate() {
8.      // The service is being created
9.  }
10.
11.@Override
12.public int onStartCommand(Intent intent, int flags, int startId) {
13.     // The service is starting, due to a call to startService()
14.     return mStartMode;
15. }
16.
17. @Override
18. public IBinder onBind(Intent intent) {
19.     // A client is binding to the service with bindService()
20.     return mBinder;
21. }
22.
23. @Override
24. public boolean onUnbind(Intent intent) {
25.     // All clients have unbound with unbindService()
26.     return mAllowRebind;
27. }
28.
29. @Override
30. public void onRebind(Intent intent) {
31.     // A client is binding to the service with bindService(),
32.     // after onUnbind() has already been called
33. }
34.
35. @Override
36. public void onDestroy() {
37.     // The service is no longer used and is being destroyed
38. }
39.}
```

注意：不同于 Activity 的生命周期回调方法，Service 不须调用父类的生命周期回调方法。

```
Unlike the activity lifecycle callback methods, you are not required to call the superclass
implementation of these callback methods.
```

1.4.4 Service 的生命周期图

官方给出的 Service 生命周期图如图 1-5 所示。该图左侧展示的是用 startService()方法启动的 Service 的生命周期，右侧展示的是用 bindService()方法启动的 Service 的生命周期。

```
The diagram on the left shows the lifecycle when the service is created with startService()
and the diagram on the right shows the lifecycle when the service is created with bindService().
```

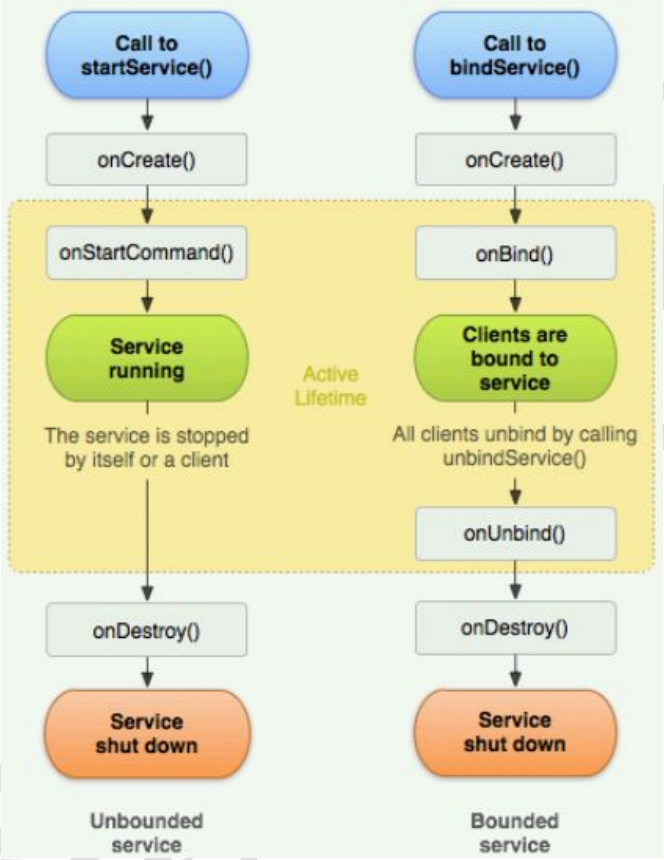

图 1-5　　Service 生命周期图

这个图说明了 Service 典型的回调方法，尽管这个图中将开启的 Service 和绑定的 Service 分开，但是我们需要记住，任何 Service 都潜在地允许被绑定。所以，一个被开启的 Service 仍然可能被绑定。实现这些方法，可以看到两层嵌套的 Service 的生命周期。

1.4.5 整体生命周期（The entire lifetime）

Service 整体的生命时间是从 onCreate()被调用开始，到 onDestroy()方法返回为止。

和 Activity 一样，Service 在 onCreate()中进行它的初始化工作，在 onDestroy()中释放残留的资源。比如，一个音乐播放 service 可以在 onCreate()中创建播放音乐的线程，在 onDestory()中停止这个线程。

onCreate() 和 onDestroy()会被所有的 Service 调用，不论 Service 是通过 startService()还是 bindService()建立的。

```
The entire lifetime of a service happens between the time onCreate() is called and the time
onDestroy() returns. Like an activity, a service does its initial setup in onCreate() and
releases all remaining resources in onDestroy(). For example, a music playback service could
```

```
create the thread where the music will be played in onCreate(), then stop the thread in onDestroy().

The onCreate() and onDestroy() methods are called for all services, whether they're created by startService() or bindService().
```

1.4.6 积极活动的生命周期(The active lifetime)

Service 积极活动的生命周期（active lifetime）是从 onStartCommand() 或 onBind()被调用开始，它们各自处理由 startService()或 bindService()方法传过来的 Intent 对象。

如果 service 是被开启的，那么它的活动生命周期和整个生命周期一同结束。

如果 service 是被绑定的，它们它的活动生命周期是在 onUnbind()方法返回后结束。

```
The active lifetime of a service begins with a call to either onStartCommand() or onBind(). Each method is handed the Intent that was passed to either startService() or bindService(), respectively.
If the service is started, the active lifetime ends the same time that the entire lifetime ends (the service is still active even after onStartCommand() returns). If the service is bound, the active lifetime ends when onUnbind() returns.
```

注意：尽管一个被开启的 service 是通过调用 stopSelf() 或 stopService()来停止的，没有一个对应的回调函数与之对应，即没有 onStop()回调方法。所以，当调用了停止的方法，除非这个 service 和客户组件绑定，否则系统将会直接销毁它，onDestory()方法会被调用，并且是这个时候唯一会被调用的回调方法。

```
Although a started service is stopped by a call to either stopSelf() or stopService(), there is not a respective callback for the service (there's no onStop() callback). So, unless the service is bound to a client, the system destroys it when the service is stopped—onDestroy() is the only callback received.
```

1.4.7 管理绑定 Service 的生命周期

当绑定的 Service 和所有客户端解除绑定之后，Android 系统将会销毁它，（除非它同时被 onStartCommand()方法开启）。

因此，如果你的 service 是一个纯粹的绑定 service，那么你不需要管理它的生命周期。然而，如果你选择实现 onStartCommand()回调方法，那么你必须显式地停止 service，因为 service 此时被看做是开启的。这种情况下，service 会一直运行到它自己调用 stopSelf()或另一个组件调用 stopService()，不论它是否和客户端绑定。

另外，如果你的 service 被开启并且接受绑定，那么当系统调用你的 onUnbind()方法时，如果你想要在下次客户端绑定的时候接受一个 onRebind()的调用（而不是调用 onBind()），你可以选择在 onUnbind()中返回 true。

onRebind()的返回值为 void，但是客户端仍然在它的 onServiceConnected()回调方法中得到 IBinder 对象。

```
When a service is unbound from all clients, the Android system destroys it (unless it was also started with onStartCommand()). As such, you don't have to manage the lifecycle of your service if it's purely a bound service—the Android system manages it for you based on whether it is bound to any clients.
```

However, if you choose to implement the onStartCommand() callback method, then you must explicitly stop the service, because the service is now considered to be started. In this case, the service runs until the service stops itself with stopSelf() or another component calls stopService(), regardless of whether it is bound to any clients.

Additionally, if your service is started and accepts binding, then when the system calls your onUnbind() method, you can optionally return true if you would like to receive a call to onRebind() the next time a client binds to the service (instead of receiving a call to onBind()). onRebind() returns void, but the client still receives the IBinder in its onServiceConnected() callback. Below, figure 1 illustrates the logic for this kind of lifecycle.

下图 1-展示了这种 Service（被开启，还允许绑定）的生命周期：

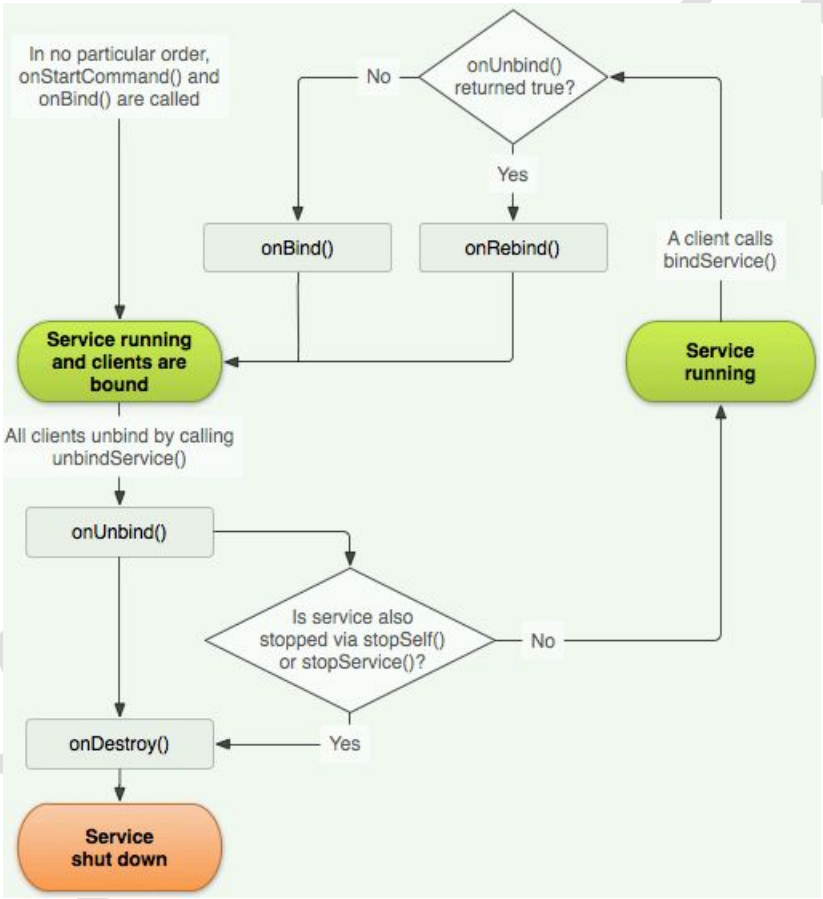

图 1-6　　绑定 Service 的生命周期

1.5 Android 中服务的调用

1.5.1 通过绑定方式调用服务

When creating a service that provides binding, you must provide an IBinder that provides the

programming interface that clients can use to interact with the service. There are three ways you can define the interface:

如果要创建一个提供绑定方式的 Service，那么就必须提供一个实现 IBinder 接口的对象，该对象提供了调用端（客户端、远端）和服务端交互的桥梁。有三种方式可以实现 IBinder 接口：

一、Extending the Binder class

If your service is used only by the local application and does not need to work across processes, then you can implement your own Binder class that provides your client direct access to public methods in the service.

Note: This works only if the client and service are in the same application and process, which is most common. For example, this would work well for a music application that needs to bind an activity to its own service that's playing music in the background.

如果你的 Service 仅仅是在本应用内使用并且不需要实现跨进程工作，那么你就可以定义一个实现 Binder 接口的类，让该类给客户端提供直接访问服务端的公开方法。

注意：这种用法仅仅局限于客户端和服务端在同一个应用和同一个进程中，这是很常见的使用情形。比如，我们的 Activity 可以绑定 Service 让音乐实现后台播放，前台控制。

For example, here's a service that provides clients access to methods in the service through a Binder implementation:

```
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.     /**
8.      * Class used for the client Binder.  Because we know this service always
9.      * runs in the same process as its clients, we don't need to deal with IPC.
10.      */
11.     public class LocalBinder extends Binder {
12.         LocalService getService() {
13.             // Return this instance of LocalService so clients can call public methods
14.             return LocalService.this;
15.         }
16.     }
17.
18.     @Override
19.     public IBinder onBind(Intent intent) {
20.         return mBinder;
21.     }
22.
23.     /** method for clients */
24.     public int getRandomNumber() {
```

```
25.      return mGenerator.nextInt(100);
26.    }
27. }
```

The LocalBinder provides the getService() method for clients to retrieve the current instance of LocalService. This allows clients to call public methods in the service. For example, clients can call getRandomNumber() from the service.

Here's an activity that binds to LocalService and calls getRandomNumber() when a button is clicked:

```
1. public class BindingActivity extends Activity {
2.    LocalService mService;
3.    boolean mBound = false;
4.
5.    @Override
6.    protected void onCreate(Bundle savedInstanceState) {
7.       super.onCreate(savedInstanceState);
8.       setContentView(R.layout.main);
9.    }
10.
11.    @Override
12.    protected void onStart() {
13.       super.onStart();
14.       // Bind to LocalService
15.       Intent intent = new Intent(this, LocalService.class);
16.       bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
17.    }
18.
19.    @Override
20.    protected void onStop() {
21.       super.onStop();
22.       // Unbind from the service
23.       if (mBound) {
24.          unbindService(mConnection);
25.          mBound = false;
26.       }
27.    }
28.
29.    /** Called when a button is clicked (the button in the layout file attaches to
30.      * this method with the android:onClick attribute) */
31.    public void onButtonClick(View v) {
32.       if (mBound) {
33.          // Call a method from the LocalService.
34.          // However, if this call were something that might hang, then this
35. request should
36.          // occur in a separate thread to avoid slowing down the activity performance.
37.          int num = mService.getRandomNumber();
38.          Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT).show();
```

```
39.         }
40.     }
41.
42.     /** Defines callbacks for service binding, passed to bindService() */
43.     private ServiceConnection mConnection = new ServiceConnection() {
44.
45.         @Override
46.         public void onServiceConnected(ComponentName className,
47.                 IBinder service) {
48.             // We've bound to LocalService, cast the IBinder and get
49. LocalService instance
50.             LocalBinder binder = (LocalBinder) service;
51.             mService = binder.getService();
52.             mBound = true;
53.         }
54.
55.         @Override
56.         public void onServiceDisconnected(ComponentName arg0) {
57.             mBound = false;
58.         }
59.     };
60. }
```

The above sample shows how the client binds to the service using an implementation of ServiceConnection and the onServiceConnected() callback. The next section provides more information about this process of binding to the service.

Note: The example above doesn't explicitly unbind from the service, but all clients should unbind at an appropriate time (such as when the activity pauses).

二、Using a Messenger（了解）

关于 Messenger（注意是 Messenger 不是 Message）的使用，在学习之外有余力的同学推荐看下文，我就不再给出译文。

```
If you need your interface to work across different processes, you can create an interface
for the service with a Messenger. In this manner, the service defines a Handler that responds
to different types of Message objects. This Handler is the basis for a Messenger that can then
share an IBinder with the client, allowing the client to send commands to the service using
Message objects. Additionally, the client can define a Messenger of its own so the service
can send messages back.

This is the simplest way to perform interprocess communication (IPC), because the Messenger
queues all requests into a single thread so that you don't have to design your service to be
thread-safe.
```

Here's a summary of how to use a Messenger:

- The service implements a Handler that receives a callback for each call from a client.
- The Handler is used to create a Messenger object (which is a reference to the Handler).
- The Messenger creates an IBinder that the service returns to clients from onBind().
- Clients use the IBinder to instantiate the Messenger (that references the service's Handler), which the client uses to send Message objects to the service.
- The service receives each Message in its Handler—specifically, in the handleMessage() method.

In this way, there are no "methods" for the client to call on the service. Instead, the client delivers "messages" (Message objects) that the service receives in its Handler.
Here's a simple example service that uses a Messenger interface:

```
1. public class MessengerService extends Service {
2.     /** Command to the service to display a message */
3.     static final int MSG_SAY_HELLO = 1;
4.
5.     /**
6.      * Handler of incoming messages from clients.
7.      */
8.     class IncomingHandler extends Handler {
9.        @Override
10.        public void handleMessage(Message msg) {
11.            switch (msg.what) {
12.                case MSG_SAY_HELLO:
13.                    Toast.makeText(getApplicationContext(),
14. "hello!", Toast.LENGTH_SHORT).show();
15.                    break;
16.                default:
17.                    super.handleMessage(msg);
18.            }
19.        }
20.    }
21.
22.    /**
23.     * Target we publish for clients to send messages to IncomingHandler.
24.     */
25.    final Messenger mMessenger = new Messenger(new IncomingHandler());
26.
27.    /**
28.     * When binding to the service, we return an interface to our messenger
29.     * for sending messages to the service.
30.     */
31.    @Override
32.    public IBinder onBind(Intent intent) {
```

```
        Toast.makeText(getApplicationContext(),
"binding", Toast.LENGTH_SHORT).show();
        return mMessenger.getBinder();
    }
}
```

Notice that the handleMessage() method in the Handler is where the service receives the incoming Message and decides what to do, based on the what member.

All that a client needs to do is create a Messenger based on the IBinder returned by the service and send a message using send(). For example, here's a simple activity that binds to the service and delivers the MSG_SAY_HELLO message to the service:

```
public class ActivityMessenger extends Activity {
    /** Messenger for communicating with the service. */
    Messenger mService = null;

    /** Flag indicating whether we have called bind on the service. */
    boolean mBound;

    /**
     * Class for interacting with the main interface of the service.
     */
    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            // This is called when the connection with the service has been
            // established, giving us the object we can use to
            // interact with the service.  We are communicating with the
            // service using a Messenger, so here we get a client-side
            // representation of that from the raw IBinder object.
            mService = new Messenger(service);
            mBound = true;
        }

        public void onServiceDisconnected(ComponentName className) {
            // This is called when the connection with the service has been
            // unexpectedly disconnected -- that is, its process crashed.
            mService = null;
            mBound = false;
        }
    };

    public void sayHello(View v) {
        if (!mBound) return;
        // Create and send a message to the service, using a supported 'what' value
        Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
```

```
34.         try {
35.             mService.send(msg);
36.         } catch (RemoteException e) {
37.             e.printStackTrace();
38.         }
39.     }
40.
41.     @Override
42.     protected void onCreate(Bundle savedInstanceState) {
43.         super.onCreate(savedInstanceState);
44.         setContentView(R.layout.main);
45.     }
46.
47.     @Override
48.     protected void onStart() {
49.         super.onStart();
50.         // Bind to the service
51.         bindService(new Intent(this, MessengerService.class), mConnection,
52.             Context.BIND_AUTO_CREATE);
53.     }
54.
55.     @Override
56.     protected void onStop() {
57.         super.onStop();
58.         // Unbind from the service
59.         if (mBound) {
60.             unbindService(mConnection);
61.             mBound = false;
62.         }
63.     }
64. }
```

```
Notice that this example does not show how the service can respond to the client. If you want
the service to respond, then you need to also create a Messenger in the client. Then when the
client receives the onServiceConnected() callback, it sends a Message to the service that
includes the client's Messenger in the replyTo parameter of the send() method.
```

三、Using AIDL

aidl 是 Android 中实现进程间通信的方式之一，也是难点，在企业开发中使用的并不是很多，但是在面试中是一个亮点。关于 aidl 的使用我单独作为一章讲解，这里先点到为止。

1.5.2 案例-音乐播放器

一、需求

通过该案例，我们可以更加直观地感受 Bound Service 的使用方法。

在后台（Service）运行播放音乐服务，在界面（Activity）控制音乐播放器。音乐播放器原型图如图 1-7 所示。因为要播放音乐，因此需要提前准备一个音频文件，然后我把该文件放到 res 下的 raw 目录下，注意，raw 目录需要我们手动创建，该文件名字必须为 raw，这是约定好的，如图 1-8 所示。

图 1-7　音乐播放中界面

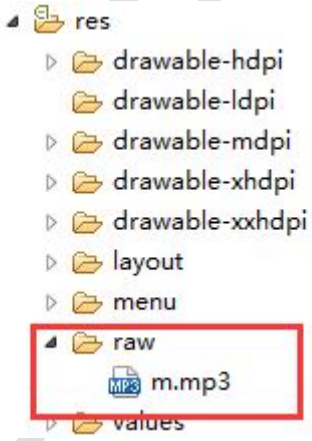

图 1-8　创建 raw 目录

二、布局

布局很简单，如下所示。

【文件 1-6】 activity_main.xml

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     xmlns:tools="http://schemas.android.com/tools"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:orientation="vertical"
6.     tools:context=".MainActivity" >
```

```
7.
8.    <TextView
9.       android:layout_width="wrap_content"
10.       android:layout_height="wrap_content"
11.       android:layout_gravity="center_horizontal"
12.       android:text="音乐播放器"
13.       android:textColor="#ff0000"
14.       android:textSize="28sp" />
15.
16.    <LinearLayout
17.       android:layout_width="match_parent"
18.       android:layout_height="wrap_content"
19.       android:orientation="horizontal" >
20.
21.       <Button
22.          android:id="@+id/bt_play"
23.          android:layout_width="0dp"
24.          android:layout_height="wrap_content"
25.          android:layout_weight="1"
26.          android:onClick="play"
27.          android:text="播放" />
28.
29.       <Button
30.          android:id="@+id/bt_pause"
31.          android:layout_width="0dp"
32.          android:layout_height="wrap_content"
33.          android:layout_weight="1"
34.          android:onClick="pause"
35.          android:text="暂停" />
36.
37.       <Button
38.          android:id="@+id/bt_stop"
39.          android:layout_width="0dp"
40.          android:layout_height="wrap_content"
41.          android:layout_weight="1"
42.          android:onClick="stop"
43.          android:text="停止" />
44.    </LinearLayout>
45.    <ProgressBar
46.       android:layout_width="match_parent"
47.       android:layout_height="wrap_content"
48.       style="?android:attr/progressBarStyleHorizontal"
49.       android:id="@+id/pb"
50.       />
51. </LinearLayout>
```

三、代码

在该案例中使用到了两个类，一个是自定义的 Service，用于播放音乐。另外一个是 MainActivity，用于界面展示和播放器控制。

【文件 1-7】 MediaService.java

```
1. public class MediaService extends Service {
2.  //声明一个 MediaPlayer 对象
3.  private MediaPlayer player;
4.
5.  @Override
6.  public IBinder onBind(Intent intent) {
7.      System.out.println("服务返回 MediaController 对象了......");
8.      return new MediaController();
9.  }
10.
11. @Override
12. public void onCreate() {
13.     System.out.println("音乐服务已经被创建......");
14.     //初始化音乐播放器
15.     player = MediaPlayer.create(this, R.raw.m);
16. }
17. //自定义一个 Binder 对象，Binder 是 IBinder 接口的子类
18. class MediaController extends Binder{
19.     public void play(){
20.         player.start();
21.     }
22.     public void pause(){
23.         player.pause();
24.     }
25. public void stop(){
26.         player.stop();
27.     }
28.     //获取音乐的总时长
29.     public int getDuration(){
30.         return player.getDuration();
31.     }
32.     //获取当前播放位置
33.     public int getCurrentPostion(){
34.         return player.getCurrentPosition();
35.     }
36.     //判断是否在播放
37.     public boolean isPlaying(){
38.         return player.isPlaying();
```

```
    }

}
}
```

【文件 1-8】 MainActivity.java

```
public class MainActivity extends Activity {
 //声明进度条
 private ProgressBar pb;
 //声明自定义的 MediaController 对象
 private MediaController mediaController;
 private boolean isRunning;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);
     //实例化进度条
     pb = (ProgressBar) findViewById(R.id.pb);
     //创建一个用于启动服务的显示意图，指向我们自定义的 MediaService 类
     Intent intent = new Intent(this, MediaService.class);
     //绑定服务，同时服务开启，如果成功则返回 true 否则返回 false
     isRunning = bindService(intent, new MediaConnection(), BIND_AUTO_CREATE);
     if (isRunning) {
         System.out.println("音乐播放器服务绑定成功！");
     } else {
         System.out.println("音乐播放器服务绑定失败！");
     }
 }
 //用于循环更新当前播放进度
 private void updateProgressBar() {
     new Thread(new Runnable() {

         @Override
         public void run() {
             while (true) {
                 SystemClock.sleep(400);
                 pb.setProgress(mediaController.getCurrentPostion());
                 if (mediaController.getDuration()
 == mediaController.getCurrentPostion()) {
                     break;
                 }
             }
         }
     }).start();
 }
```

```
41.
42. public void play(View view) {
43.     if (mediaController!=null) {
44.         //如果音乐正在播放则不能再次播放
45.         if(mediaController.isPlaying()){
46.             Toast.makeText(this, "音乐播放中", 0).show();
47.             return;
48.         }else {
49.             mediaController.play();
50.             Toast.makeText(this, "音乐开是播放", 0).show();
51.         }
52.     }
53. }
54. //暂停
55. public void pause(View view) {
56.     if (mediaController != null) {
57.         mediaController.pause();
58.     }
59. }
60. //停止
61. public void stop(View view) {
62.     if (mediaController!=null) {
63.         //停止的时候将进度条设置为初始位置
64.         pb.setProgress(0);
65.         mediaController.stop();
66.         Toast.makeText(this, "音乐已经关闭！", 0).show();
67.     }
68. }
69. //新建一个 ServiceConnection 类
70. class MediaConnection implements ServiceConnection {
71.     /**
72.      * 当 service 被绑定的时候回调该函数
73.      */
74.     @Override
75.     public void onServiceConnected(ComponentName name, IBinder service) {
76.         //返回的 IBinder 对象其实就是我们自定义的 MediaController 类对象
77.         mediaController = (MediaController) service;
78.         //给进度条设置最大值
79.         pb.setMax(mediaController.getDuration());
80.         //更新进度条
81.         updateProgressBar();
82.         System.out.println("服务已经连接......");
83.     }
84.     /**
```

```
85.      * 服务被关闭或者断开的时候调用该方法
86.      */
87.     @Override
88.     public void onServiceDisconnected(ComponentName name) {
89.         System.out.println("服务已经断开......");
90.     }
91. }
92. }
```

四、AndroidManifest.xml

在 AndroidManifest.xml 中注册服务

```
<service android:name="com.itheima.musicPlayer.MediaService"/>
```

五、总结

在上面的案例中使用到了 MediaPlayer，关于 MediaPlayer 的用法在官方文档上也有清楚的解释，因此这里就不再赘述。

在上面的 MediaService 类中的 onBind()方法中，返回了 MediaController 对象，该对象是 MediaService 中的内部类，该类实现了 Binder 接口。该对象在 MediaService 被绑定的时候被返回。我们只要获取到了 MediaController 对象就可以对 MediaPlayer 进行控制了。

1.5.3 AIDL 实现进程间通信

在 Android 平台中，各个组件运行在自己的进程中，他们之间是不能相互访问的，但是在程序之间是不可避免的要传递一些对象，在进程之间相互通信。为了实现进程之间的相互通信，Android 采用了一种轻量级的实现方式 RPC(Remote Procedure Call 远程进程调用)来完成进程之间的通信，并且 Android 通过接口定义语言（Android Interface Definition Language ，AIDL）来生成两个进程之间相互访问的代码，例如，你在 Activity 里的代码需要访问 Service 中的一个方法，那么就可以通过这种方式来实现了。

AIDL 是 Android 的一种接口描述语言。编译器可以通过 aidl 文件生成一段代码，通过预先定义的接口达到两个进程内部通信进程的目的. 如果需要在一个 Activity 中, 访问另一个 Service 中的某个对象, 需要先将对象转化成 AIDL 可识别的参数(可能是多个参数), 然后使用 AIDL 来传递这些参数, 在消息的接收端, 使用这些参数组装成自己需要的对象。

如果想让我们的 Service 可以提供远程服务，那么就必须定义一个.aidl 文件，该文件使用的是 java 语法，类似 java 的接口。然后将该文件在客户端和服务端的 src 目录下各自保存一份，这样编译器就会根据 aidl 文件自动生成一个 java 类，也就说在客户端和服务端都拥有了相同的类文件了。

aidl 的使用分为如下三个步骤（见官方文档）：

一、Create the .aidl file

```
AIDL uses a simple syntax that lets you declare an interface with one or more methods that
can take parameters and return values. The parameters and return values can be of any type,
```

even other AIDL-generated interfaces.

You must construct the .aidl file using the Java programming language. Each .aidl file must define a single interface and requires only the interface declaration and method signatures.

By default, AIDL supports the following data types:

- All primitive types in the Java programming language (such as int, long, char, boolean, and so on)
- String
- CharSequence
- List
- All elements in the List must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you've declared. A List may optionally be used as a "generic" class (for example, List<String>). The actual concrete class that the other side receives is always an ArrayList, although the method is generated to use the List interface.
- Map

All elements in the Map must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you've declared. Generic maps, (such as those of the form Map<String,Integer> are not supported. The actual concrete class that the other side receives is always a HashMap, although the method is generated to use the Map interface.

You must include an import statement for each additional type not listed above, even if they are defined in the same package as your interface.

When defining your service interface, be aware that:

Methods can take zero or more parameters, and return a value or void.
All non-primitive parameters require a directional tag indicating which way the data goes. Either in, out, or inout (see the example below).

Primitives are in by default, and cannot be otherwise.

Caution: You should limit the direction to what is truly needed, because marshalling parameters is expensive.
All code comments included in the .aidl file are included in the generated IBinder interface (except for comments before the import and package statements).
Only methods are supported; you cannot expose static fields in AIDL.

Here is an example .aidl file:

```
// IRemoteService.aidl
package com.example.android;
```

```
// Declare any non-default types here with import statements

/** Example service interface */
interface IRemoteService {
    /** Request the process ID of this service, to do evil things with it. */
    int getPid();

    /** Demonstrates some basic types that you can use as parameters
     * and return values in AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat,
            double aDouble, String aString);
}
```

Simply save your .aidl file in your project's src/ directory and when you build your application, the SDK tools generate the IBinder interface file in your project's gen/ directory. The generated file name matches the .aidl file name, but with a .java extension (for example, IRemoteService.aidl results in IRemoteService.java).

If you use Eclipse, the incremental build generates the binder class almost immediately. If you do not use Eclipse, then the Ant tool generates the binder class next time you build your application—you should build your project with ant debug (or ant release) as soon as you're finished writing the .aidl file, so that your code can link against the generated class.

二、Implement the interface

```
    When you build your application, the Android SDK tools generate a .java interface file
named after your .aidl file. The generated interface includes a subclass named Stub that is
an abstract implementation of its parent interface (for example, YourInterface.Stub) and
declares all the methods from the .aidl file.
To implement the interface generated from the .aidl, extend the generated Binder interface
(for example, YourInterface.Stub) and implement the methods inherited from the .aidl file.

Here is an example implementation of an interface called IRemoteService (defined by the
IRemoteService.aidl example, above) using an anonymous instance:

private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
    public int getPid(){
        return Process.myPid();
    }
    public void basicTypes(int anInt, long aLong, boolean aBoolean,
        float aFloat, double aDouble, String aString) {
        // Does nothing
    }
};
```

Now the mBinder is an instance of the Stub class (a Binder), which defines the RPC interface for the service. In the next step, this instance is exposed to clients so they can interact with the service.

There are a few rules you should be aware of when implementing your AIDL interface:

- Incoming calls are not guaranteed to be executed on the main thread, so you need to think about multithreading from the start and properly build your service to be thread-safe.
- By default, RPC calls are synchronous. If you know that the service takes more than a few milliseconds to complete a request, you should not call it from the activity's main thread, because it might hang the application (Android might display an "Application is Not Responding" dialog)—you should usually call them from a separate thread in the client.
- No exceptions that you throw are sent back to the caller.

三、Expose the interface to clients

Once you've implemented the interface for your service, you need to expose it to clients so they can bind to it. To expose the interface for your service, extend Service and implement onBind() to return an instance of your class that implements the generated Stub (as discussed in the previous section). Here's an example service that exposes the IRemoteService example interface to clients.

```
public class RemoteService extends Service {
    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public IBinder onBind(Intent intent) {
        // Return the interface
        return mBinder;
    }

    private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
        public int getPid(){
            return Process.myPid();
        }
        public void basicTypes(int anInt, long aLong, boolean aBoolean,
            float aFloat, double aDouble, String aString) {
            // Does nothing
        }
    };
```

```
}
Now, when a client (such as an activity) calls bindService() to connect to this service, the client's onServiceConnected() callback receives the mBinder instance returned by the service's onBind() method.

The client must also have access to the interface class, so if the client and service are in separate applications, then the client's application must have a copy of the .aidl file in its src/ directory (which generates the android.os.Binder interface—providing the client access to the AIDL methods).

When the client receives the IBinder in the onServiceConnected() callback, it must call YourServiceInterface.Stub.asInterface(service) to cast the returned parameter to YourServiceInterface type. For example:

IRemoteService mIRemoteService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Called when the connection with the service is established
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Following the example above for an AIDL interface,
        // this gets an instance of the IRemoteInterface, which we can use to call on the service
        mIRemoteService = IRemoteService.Stub.asInterface(service);
    }

    // Called when the connection with the service disconnects unexpectedly
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "Service has unexpectedly disconnected");
        mIRemoteService = null;
    }
};
```

1.5.4 案例-调用远程服务

一、需求

该案例模拟一个购物类 app 暂且叫商户调用支付宝 app 的过程（仅仅是模拟，跟真实的是不一样的），我们需要创建两个 Android 工程，商户和支付宝。支付宝提供支付服务，商户通过 aidl 对支付宝进行远程调用。

二、代码

《支付宝》工程核心代码：

在 src 目录下定义一个 aidl 文件，代码如文件 1-9 所示。

【文件 1-9】 AlipayRemoteService.aidl

```
1. package com.itheima.alipay.aidl;
2.
3. interface AlipayRemoteService{
4.  boolean forwardPayMoney(float money);
5. }
```

当该 aidl 文件创建好以后 ADT 会自动在 gen 目录下创建对应的类。如图 1-9 所示。

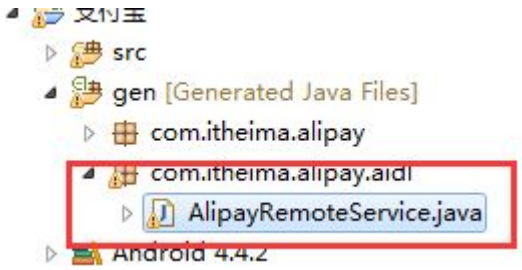

图 1-9　aidl 文件自动生成的 java 类

定义一个 Service 类，代码如文件 1-10 所示。

【文件 1-10】 AlipayService .java

```
6. public class AlipayService extends Service {
7.
8.  @Override
9.  public IBinder onBind(Intent intent) {
10.     return new PayController();
11. }
12. public boolean pay(float money){
13.     System.out.println("成功付款"+money);
14.     return true;
15. }
16. /**
17.  * 因为 Stub 已经继承了 IBinder 接口，因此 PayController 类也间接继承了该接口
18.  * @author wzy Dec 13, 2014
19.  *
20.  */
21. public class PayController extends Stub{
22.
23.     @Override
24.     public boolean forwardPayMoney(float money) throws RemoteException {
25.         return pay(money);
26.     }
27. }
28. }
29.
```

《商户》工程核心代码：

将《支付宝》工程中创建的 aidl 文件拷贝到 src 目录下，如图 1-10 所示。

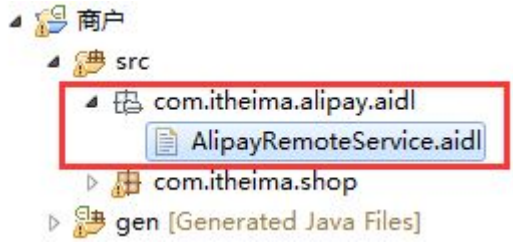

图 1-10　将 aidl 拷贝到客户端工程中

【文件 1-11】　MainActivity.java

```
1. public class MainActivity extends Activity {
2.  //声明一个 AlipayRemoteService 对象，该类是根据 aidl 文件自动生成
3.  private AlipayRemoteService alipayRemoteService;
4.  private EditText et;
5.
6.  @Override
7.  protected void onCreate(Bundle savedInstanceState) {
8.      super.onCreate(savedInstanceState);
9.      setContentView(R.layout.activity_main);
10.     et = (EditText) findViewById(R.id.et);
11.     //创建一个隐式意图，用于启动《支付宝》中的 Service
12.     Intent intent = new Intent();
13.     intent.setAction("com.itheima.alipay");
14.     //绑定远程服务
15.     boolean bindService = bindService(intent, new
16. MyConnection(), Context.BIND_AUTO_CREATE);
17.     if (bindService) {
18.         Toast.makeText(this, "服务绑定成功", 1).show();
19.         System.out.println("服务绑定成功");
20.     }else {
21.         Toast.makeText(this, "服务绑定失败", 1).show();
22.         System.out.println("服务绑定失败");
23.     }
24. }
25. public void pay(View view){
26.     float money = Float.valueOf(et.getText().toString());
27.     try {
28.         alipayRemoteService.forwardPayMoney(money);
29.     } catch (RemoteException e) {
30.         e.printStackTrace();
31.         Toast.makeText(this, "付款失败", 1).show();
32.     }
33.     Toast.makeText(this, "成功转账："+money+"元！", 0).show();
34. }
35. class MyConnection implements ServiceConnection{
```

```
36.     /**
37.      * 通过 Stub 的静态方法 asInterface 将 IBinder 对象转化为本地 AlipayRemoteService 对象
38.      */
39.     @Override
40.     public void onServiceConnected(ComponentName name, IBinder service) {
41.         System.out.println("服务已经连接。。。");
42.         alipayRemoteService = Stub.asInterface(service);
43.     }
44.
45.     @Override
46.     public void onServiceDisconnected(ComponentName name) {
47.         System.out.println("服务已经关闭。");
48.     }
49. }
50. }
```

三、布局

《商户》应用需要一个界面，布局文件如文件 1-12 所示。

【文件 1-12】 activity_main.java

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     xmlns:tools="http://schemas.android.com/tools"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:orientation="vertical"
6.     tools:context=".MainActivity" >
7.     <TextView
8.        android:layout_gravity="center_horizontal"
9.        android:textColor="#ff0000"
10.        android:textSize="28sp"
11.        android:layout_width="wrap_content"
12.        android:layout_height="wrap_content"
13.        android:text="商城支付-调用远程服务" />
14.     <EditText
15.        android:layout_width="match_parent"
16.        android:layout_height="wrap_content"
17.        android:hint="请输入要转正的金额"
18.        android:inputType="number"
19.        android:id="@+id/et"
20.        />
21.     <Button
22.        android:layout_gravity="right"
23.        android:layout_width="wrap_content"
```

```
24.         android:layout_height="wrap_content"
25.         android:text="确定支付"
26.         android:onClick="pay"
27.         />
28.
29. </LinearLayout>
```

四、AndroidManifest.xml

在《支付宝》工程的清单文件中注册服务。

```
<service  android:name="com.itheima.alipay.service.AlipayService">
          <intent-filter >
             <action android:name="com.itheima.alipay"></action>
          </intent-filter>
 </service>
```

五、总结

先将《支付宝》部署到模拟器，然后将《商户》部署到模拟器，然后在《商户》界面输入一个金额，然后点击确定支付，发现《商户》工程已经成功通过远程服务调用了《支付宝》中的服务。运行图 1-11 所示。

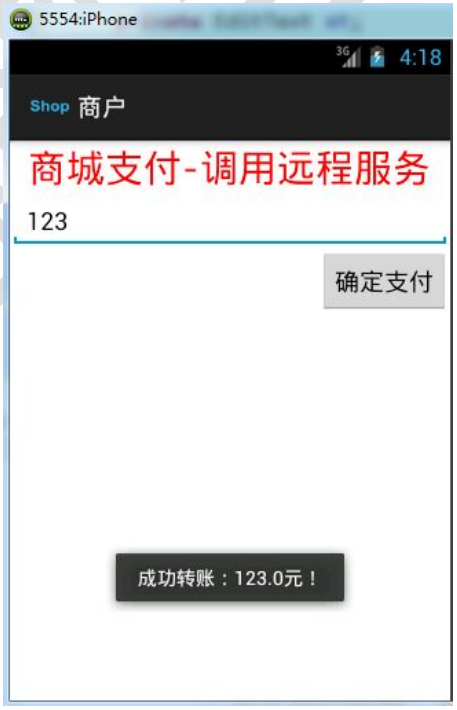

图 1-11