# Bound services

原创 xyzso1z 2020-08-23 15:10:18 29 收藏 编辑 版权

分类专栏: Android 文章标签: Service BoundService

## 一、基础知识

`Bound Service` 是 `Service` 类的实现，可让其它应用与其进行绑定和交互。如要为服务提供绑定，必须实现 `onBind()` 回调方法。该方法会返回 `IBinder` 对象，该对象定义的编程接口可供客户端用来与服务进行交互。

## 二、创建绑定服务

创建提供绑定的服务时，必须提供 `IBinder` ，进而提供编程接口，以便客户端使用此接口与服务进行交互。可以通过三种方法定义接口：

- **扩展`Binder`类**
  如果服务是提供给自己应用专用，并且在与客户端相同的进程中运行（常见情况），则应通过扩展 `Binder` 类并从 `onBind()` 返回该类的实例来创建接口。收到 `Binder` 后，客户端可利用其直接访问 `Binder` 实现或 `Service` 中可用的公共方法。
- **使用`Messenger`**
  如果让接口跨不同进程工作，可以使用 `Messenger` 为服务创建接口。服务可借此方法定义 `Handler` ，以响应不同类型的 `Message` 对象。此 `Handler` 是 `Messenger` 的基础，后者随后可与客户端分享一个 `IBinder` ，以便客户端能利用 `Message` 对象向服务发送命令。此外，客户端还可定义自有 `Massenger` ，以便服务回传消息。
  这是执行进程间通信最简单的方法，因为 `Massenger` 会在单个线程中创建包含所有请求的队列，这样就不必对服务进行线程安全设计。
- **使用AIDL**
  对于 `Messenger` 的方法而言，其实际上是以AIDL作为其底层结构。如上所述，`Messenger` 会在单个线程中创建包含所有客户端请求的队列，以便 `Service` 一次接收一个请求。不过，如果想让服务同时处理多个请求，则可直接使用AIDL。在此情况下， `Service` 必须达到线程安全的要求，并且能够进行多线程处理。
  如果直接使用AIDL,需要创建定义编程接口的.aidl文件。Android SDK工具会利用该文件生成实现接口和处理IPC的抽象类，随后可在服务内对该类进行扩展。

  > 大多数应用不应该使用AIDL来创建绑定服务，因为它可能需要多线程处理能力，并可能导致更为复杂的实现。因此，AIDL并不适合大多数应用。

### 2.1 扩展 `Binder` 类

如果 `Service` 仅供本地应用使用，且无需跨进程工作，则可以实现自有 `Binder` 类，让客户端通过该类直接访问服务中的公共方法。

> 注意：只有客户端和服务处于同一应用和进程内（最常见的情况）时，此方法才有效。例如，此方法非常适用于将Activity绑定到某个音乐应用的自有服务，进而实现在后台播放音乐。

以下是具体的设置方法：

1. 在 `Service` 中，创建可执行以下某种操作的 `Binder` 实例：
    1. 包含客户端可调用的公共方法。
    2. 返回当前的 `Service` 实例，该实例中包含客户端可调用的公共方法。
    3. 返回有服务承载的其他类的实例，其中包含客户端可调用的公共方法。
2. 从 `onBind()` 回调方法返回此 `Binder` 实例。
3. 在客户端中，从 `onServiceConnected()` 回调方法接收 `Binder` ，并使用提供的方法调用绑定服务。

> 注意：服务和客户端必须在同一个应用内，这样客户端才能转换返回的对象并正确调用其API。服务和客户端还必须在同一进程中，因此此方法不执行任何跨进程组。

例如，以下服务可让客户端通过 `Binder` 实现访问服务中的方法：

```
public class LocalService extends Service {
    // Binder given to clients
    private final IBinder binder = new LocalBinder();
    // Random number generator
    private final Random mGenerator = new Random();

    /**
     * Class used for the client Binder.  Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
    public class LocalBinder extends Binder {
        LocalService getService() {
            // Return this instance of LocalService so clients can call public me
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
```

```
    }

    /** method for clients */
    public int getRandomNumber() {
      return mGenerator.nextInt(100);
    }
}
```

上例展示客户端如何使用 `ServiceConnection` 的实现和 `onServiceConnected()` 回调绑定到服务。

> 注意：在上例中， `onStop()` 方法取消了客户端与服务的绑定。

## 2.2 使用 Messenger

如需让 `Service` 与远程进程通信，则可使用 `Messenger` 为 `Service` 提供接口。借助此方法，无需使用AIDL便可执行进程间通信。

为接口使用 `Messenger` 比使用AIDL更简单，因为 `Messenger` 会将所有Service调用加入队列。纯AIDL接口会同时向 `Service` 发送多个请求， `Service` 随后必须执行多线程处理。

对于大多数应用， **`Service` 无需执行多线程处理，因此使用 `Messenger` 即可让 `Service` 一次处理一个调用。如果服务必须执行多线程处理，则应该使用AIDL来定义接口。**
以下是 `Messenger` 的使用方法摘要：

1. 服务实现一个 `Handler` ， 由该类为每个客户端调用接收回调。
2. 服务使用 `Handler` 来创建 `Messenger` 对象（对 `Handler` 的引用）。
3. `Messenger` 创建一个 `IBinder` ， 服务通过 `onBind()` 使其返回客户端。
4. 客户端使用 `IBinder` 将 `Messenger` （其引用服务的 `Handler` ）实例化，然后使用后者将 `message` 对象发送给服务。
5. 服务在其 `Handler` 中（具体是在 `handleMessage()` 方法中）接收每个 `message` 。

以下示例展示如何使用 `Messenger` 接口：

```
public class MessengerService extends Service {
    /**
     * Command to the service to display a message
     */
    static final int MSG_SAY_HELLO = 1;


```

```
    /**
     * Handler of incoming messages from clients.
     */
    static class IncomingHandler extends Handler {
        private Context applicationContext;

        IncomingHandler(Context context) {
            applicationContext = context.getApplicationContext();
        }

        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_SAY_HELLO:
                    Toast.makeText(applicationContext, "hello!", Toast.LENGTH_SHO
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }

    /**
     * Target we publish for clients to send messages to IncomingHandler.
     */
    Messenger mMessenger;

    /**
     * When binding to the service, we return an interface to our messenger
     * for sending messages to the service.
     */
    @Override
    public IBinder onBind(Intent intent) {
        Toast.makeText(getApplicationContext(), "binding", Toast.LENGTH_SHORT).sh
        mMessenger = new Messenger(new IncomingHandler(this));
        return mMessenger.getBinder();
    }
}

```

注意，服务会在 `Handler` 的 `handleMessage()` 方法中接收传入的 `Message` ， 并根据 `what` 成员决定下一步操作。

客户端只需根据服务返回的 `IBinder` 创建 `Messenger` ， 然后利用 `send()` 发送消息。例如，以下简单 `Activity` 展示如何绑定到服务并向服务传递 `MSG_SAY_HELLO` 消息：

```
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 bound;

    /**
     * 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);
            bound = 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;
            bound = false;
        }
    };

    public void sayHello(View v) {
        if (!bound) return;
        // Create and send a message to the service, using a supported 'what' val
        Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
        try {
            mService.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @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 (bound) {
60              unbindService(mConnection);
61              bound = false;
62          }
63      }
64  }
65
```

此示例并未说明服务说明如何对客户端做出响应。如果想让服务做出响应，还需在客户端中创建一个 `Messenger` 。收到 `onServiceConnected()` 回调时，客户端会向服务发送 `Message` ，并在其 `send()` 方法的 `replyTo` 参数中加入客户端的 `Messenger` 。

## 三、绑定到服务

应用组件（客户端）可通过调用 `bindService()` 绑定到服务。然后，Android 系统会调用服务的 `onBind()` 方法，该方法会返回用于与服务交互的 `IBinder` 。

绑定为异步操作，并且 `bindService()` 无需将 `IBinder` 返回至客户端即立即返回。如要接收 `IBinder` ，客户端必须创建一个 `ServiceConnection` 实例，并将其传递给 `bindService()` 。 `ServiceConnection` 包含一个回调方法，系统通过调用该方法来传递 `IBinder` 。

> 注意：只有 `Activity` 、 `Service` 和 `ContentProvider` 可以绑定到服务，无法从广播接收器绑定到服务。

如要从客户端绑定到服务，应该执行以下步骤：

1. 实现 `ServiceConnection` 。必须重写两个回调方法： `onServiceConnected()` 系统会调用该方法，进而传递服务的 `onBind()` 方法所返回的 `IBinder` ； `onServiceDisconnected()` 当与服务的连接意外中断(例如服务崩溃或被终止)时，系统会调用该方法。**当客户端取消绑定时，系统不会调用该方法。**
2. 调用 `bindService()` ，从而传递 `ServiceConnection` 实现。如果该方法返回“ `false` ”，则说明客户端未与服务进行有效连接，但是，客户端仍应调用 `unbindService()` ；

3. 当系统调用 `onServiceConnected()` 回调方法时，可以使用接口定义的方法开始调用服务。
4. 如果断开与服务的连接，应该调用 `unbindService()` 。

**说明**:
在匹配客户端生命周期的引入(bring-up)和退出(tear-down)时，需要配对绑定和取消绑定:

- 如果要在 `Activity` 可见时与 `Service` 交互，则应在 `onStart()` 期间进行绑定，在 `onStop()` 期间取消绑定。
- 当 `Activity` 在后台处于停止运行状态时，若仍希望其能够接受响应，则可在 `onCreate()` 期间进行绑定，在 `onDestroy()` 期间取消绑定。

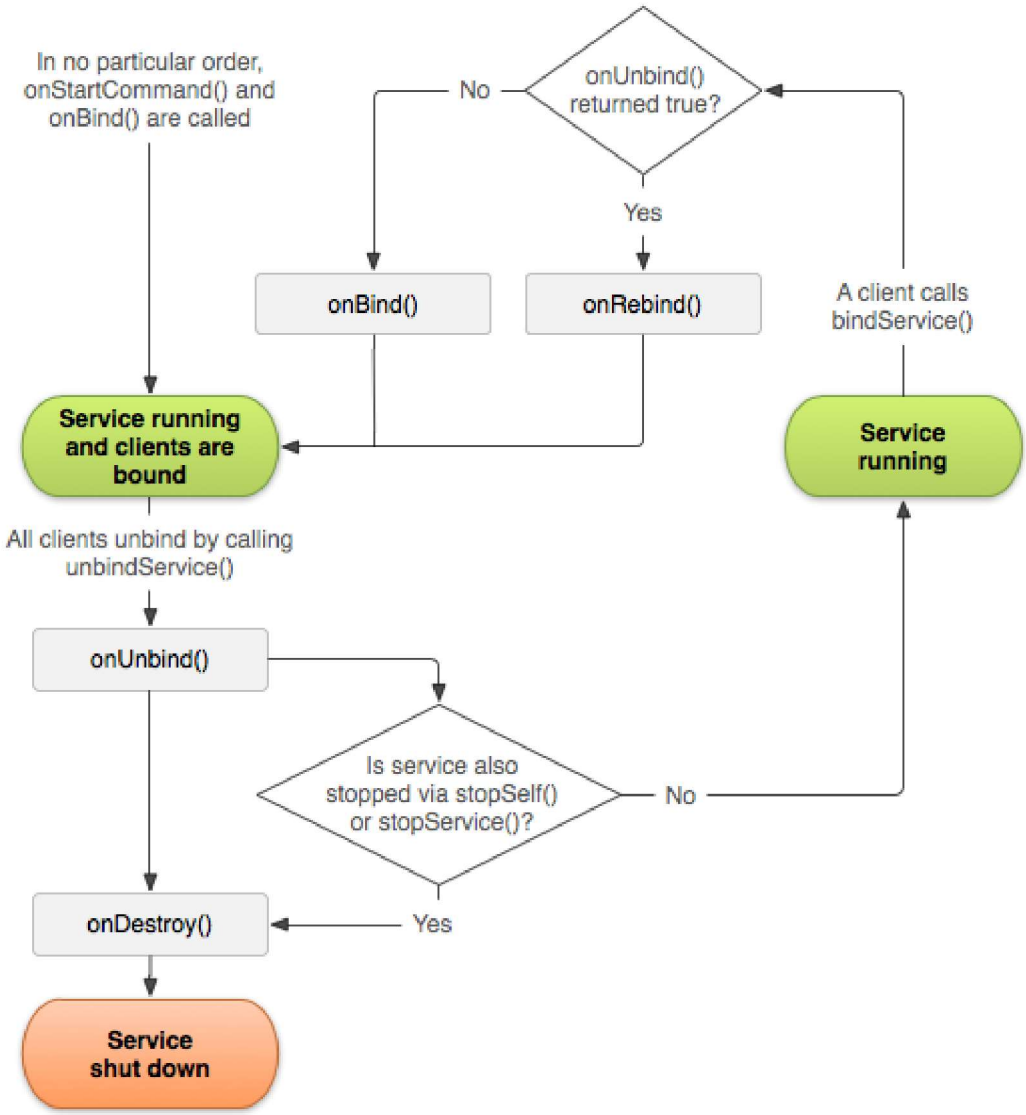
> 通常情况下，不应该在 `Activity` 的 `onResume()` 和 `onPause()` 期间绑定和取消绑定，因为每次切换生命周期状态时都会发生这些回调，并且应让这些转换期间的处理工作保持最少。

## 四、管理绑定服务的生命周期

当取消服务与所有客户端之间的绑定时，系统会销毁该服务（除非还使用 `onStartCommand()` 启动了该服务 ）。因此，如果服务完全是绑定服务，则无需管理其生命周期，系统会根据它是否绑定到任何客户端管理。

如果，选择实现 `onStartCommand()` 回调方法，则必须显示停止服务，因为系统现已将其视为已启动状态。在此情况下，服务将一直运行，直到其通过 `stopSelf()` 自行停止，或其他组件调用 `stopService()` 。

此外，如果服务已启动并接受绑定，则当系统调用 `onUnbind()` 方法时，如果想在客户端下一次绑定到服务时接收 `onRebind()` 调用，则可返回 `true` 。 `onRebind()` 返回空值，客户端仍在其 `onServiceCOnnected()` 回调中接收 `IBinder` 。 下图说明这些生命周期的逻辑。


In no particular order, onStartCommand() and onBind() are called
No
onUnbind() returned true?
Yes
onBind()
onRebind()
A client calls bindService()
Service running and clients are bound
Service running
All clients unbind by calling unbindService()
onUnbind()
Is service also stopped via stopSelf() or stopService()?
No
Yes
onDestroy()
Service shut down