绑定服务

本文内容

关键类

- `Service`
- `ServiceConnection`
- `IBinder`

示例

- `RemoteService`
- `LocalService`

另请参阅

- 服务

绑定服务是客户端-服务器接口中的服务器。绑定服务可让组件（例如 Activity）绑定到服务、发送请求、接收响应，甚至执行进程间通信 (IPC)。 绑定服务通常只在为其他应用组件服务时处于活动状态，不会无限期在后台运行。

本文向您介绍如何创建绑定服务，包括如何绑定到来自其他应用组件的服务。 不过，您还应参阅服务文档，了解有关一般服务的更多信息，例如：如何利用服务传送通知、如何将服务设置为在前台运行等等。

基础知识

绑定服务是 `Service` 类的实现，可让其他应用与其绑定和交互。要提供服务绑定，您必须实现 `onBind()` 回调方法。该方法返回的 `IBinder` 对象定义了客户端用来与服务进行交互的编程接口。

客户端可通过调用 `bindService()` 绑定到服务。调用时，它必须提供 `ServiceConnection` 的实现，后者会监控与服务的连接。`bindService()` 方法会立即无值返回，但当 Android 系统创建客户端与服务之间的连接时，会对 `ServiceConnection` 调用 `onServiceConnected()`，向客户端传递用来与服务通信的 `IBinder`。

多个客户端可同时连接到一个服务。不过，只有在第一个客户端绑定时，系统才会调用服务的 `onBind()` 方法来检索 `IBinder`。系统随后无需再次调用 `onBind()`，便可将同一 `IBinder` 传递至任何其他绑定的客户端。

当最后一个客户端取消与服务的绑定时，系统会将服务销毁（除非 `startService()` 也启动了该服务）。

当您实现绑定服务时，最重要的环节是定义您的 `onBind()` 回调方法返回的接口。您可以通过几种不同的方法定义服务的 `IBinder` 接口，下文对这些方法逐一做了阐述。

绑定到已启动服务

正如服务文档中所述，您可以创建同时具有已启动和绑定两种状态的服务。 也就是说，可通过调用 `startService()` 启动该服务，让服务无限期运行；此外，还可通过调用 `bindService()` 使客户端绑定到服务。

如果您确实允许服务同时具有已启动和绑定状态，则服务启动后，系统“不会”在所有客户端都取消绑定时销毁服务。 为此，您必须通过调用 `stopSelf()` 或 `stopService()` 显式停止服务。

创建绑定服务

> 尽管您通常应该实现 `onBind()` *或* `onStartCommand()`，但有时需要同时实现这两者。例如，音乐播放器可能发现让其服务无限期运行并同时提供绑定很有用处。 这样一来，Activity 便可启动服务进行音乐播放，即使用户离开应用，音乐播放也不会停止。 然后，当用户返回应用时，Activity 可绑定到服务，重新获得回放控制权。
>
> 请务必阅读管理绑定服务的生命周期部分，详细了解有关为已启动服务添加绑定时该服务的生命周期信息。

创建提供绑定的服务时，您必须提供 `IBinder`，用以提供客户端用来与服务进行交互的编程接口。 您可以通过三种方法定义接口：

扩展 Binder 类

如果服务是供您的自有应用专用，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()` 返回它的一个实例来创建接口。客户端收到 `Binder` 后，可利用它直接访问 `Binder` 实现中乃至 `Service` 中可用的公共方法。

如果服务只是您的自有应用的后台工作线程，则优先采用这种方法。 不以这种方式创建接口的唯一原因是，您的服务被其他应用或不同的进程占用。

使用 Messenger

如需让接口跨不同的进程工作，则可使用 `Messenger` 为服务创建接口。服务可以这种方式定义对应于不同类型 `Message` 对象的 `Handler`。此 `Handler` 是 `Messenger` 的基础，后者随后可与客户端分享一个 `IBinder`，从而让客户端能利用 `Message` 对象向服务发送命令。此外，客户端还可定义自有 `Messenger`，以便服务回传消息。

这是执行进程间通信 (IPC) 的最简单方法，因为 `Messenger` 会在单一线程中创建包含所有请求的队列，这样您就不必对服务进行线程安全设计。

使用 AIDL

AIDL（Android 接口定义语言）执行所有将对象分解成原语的工作，操作系统可以识别这些原语并将它们编组到各进程中，以执行 IPC。 之前采用 `Messenger` 的方法实际上是以 AIDL 作为其底层结构。 如上所述，`Messenger` 会在单一线程中创建包含所有客户端请求的队列，以便服务一次接收一个请求。 不过，如果您想让服务同时处理多个请求，则可直接使用 AIDL。 在此情况下，您的服务必须具备多线程处理能力，并采用线程安全式设计。

如需直接使用 AIDL，您必须创建一个定义编程接口的 `.aidl` 文件。Android SDK 工具利用该文件生成一个实现接口并处理 IPC 的抽象类，您随后可在服务内对其进行扩展。

> **注**：大多数应用“都不会”使用 AIDL 来创建绑定服务，因为它可能要求具备多线程处理能力，并可能导致实现的复杂性增加。因此，AIDL 并不适合大多数应用，本文也不会阐述如何将其用于您的服务。如果您确定自己需要直接使用 AIDL，请参阅 AIDL 文档。

扩展 Binder 类

如果您的服务仅供本地应用使用，不需要跨进程工作，则可以实现自有 `Binder` 类，让您的客户端通过该类直接访问服务中的公共方法。

> **注**：此方法只有在客户端和服务位于同一应用和进程内这一最常见的情况下方才有效。 例如，对于需要将 Activity 绑定到在后台播放音乐的自有服务的音乐应用，此方法非常有效。

以下是具体的设置方法：

1. 在您的服务中，创建一个可满足下列任一要求的 `Binder` 实例：
   - 包含客户端可调用的公共方法
   - 返回当前 `Service` 实例，其中包含客户端可调用的公共方法
   - 或返回由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**：之所以要求服务和客户端必须在同一应用内，是为了便于客户端转换返回的对象和正确调用其 API。服务和客户端还必须在同一进程内，因为此方法不执行任何跨进程编组。

例如，以下这个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 mBinder = 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 methods
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

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

`LocalBinder` 为客户端提供 `getService()` 方法，以检索 `LocalService` 的当前实例。这样，客户端便可调用服务中的公共方法。 例如，客户端可调用服务中的 `getRandomNumber()`。

点击按钮时，以下这个 Activity 会绑定到 `LocalService` 并调用 `getRandomNumber()` ：

```
public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;

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

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to LocalService
        Intent intent = new Intent(this, LocalService.class);
        bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        // Unbind from the service
        if (mBound) {
            unbindService(mConnection);
            mBound = false;
        }
    }

    /** Called when a button is clicked (the button in the layout file attaches to
      * this method with the android:onClick attribute) */
    public void onButtonClick(View v) {
        if (mBound) {
            // Call a method from the LocalService.
            // However, if this call were something that might hang, then this request should
            // occur in a separate thread to avoid slowing down the activity performance.
            int num = mService.getRandomNumber();
            Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT).show();
        }
    }

    /** Defines callbacks for service binding, passed to bindService() */
    private ServiceConnection mConnection = new ServiceConnection() {

        @Override
        public void onServiceConnected(ComponentName className,
                IBinder service) {
            // We've bound to LocalService, cast the IBinder and get LocalService instance
            LocalBinder binder = (LocalBinder) service;
            mService = binder.getService();
            mBound = true;
        }

        @Override
        public void onServiceDisconnected(ComponentName arg0) {
            mBound = false;
        }
    };
}
```

上例说明了客户端如何使用 `ServiceConnection` 的实现和 `onServiceConnected()` 回调绑定到服务。下文更详细介绍了绑定到服务的过程。

> **注**：在上例中，`onStop()` 方法将客户端与服务取消绑定。 客户端应在适当时机与服务取消绑定，如附加说明中所述。

如需查看更多示例代码，请参见 ApiDemos 中的 `LocalService.java` 类和 `LocalServiceActivities.java` 类。

使用 Messenger

如需让服务与远程进程通信，则可使用 `Messenger` 为您的服务提供接口。利用此方法，您无需使用 AIDL 便可执行进程间通信 (IPC)。

以下是 `Messenger` 的使用方法摘要：

- 服务实现一个 `Handler`，由其接收来自客户端的每个调用的回调
- `Handler` 用于创建 `Messenger` 对象（对 `Handler` 的引用）
- `Messenger` 创建一个 `IBinder`，服务通过 `onBind()` 使其返回客户端
- 客户端使用 `IBinder` 将 `Messenger`（引用服务的 `Handler`）实例化，然后使用后者将 `Message` 对象发送给服务
- 服务在其 `Handler` 中（具体地讲，是在 `handleMessage()` 方法中）接收每个 `Message`。

> **与 AIDL 比较**
>
> 当您需要执行 IPC 时，为您的接口使用 `Messenger` 要比使用 AIDL 实现它更加简单，因为 `Messenger` 会将所有服务调用排入队列，而纯粹的 AIDL 接口会同时向服务发送多个请求，服务随后必须应对多线程处理。
>
> 对于大多数应用，服务不需要执行多线程处理，因此使用 `Messenger` 可让服务一次处理一个调用。如果您的服务必须执行多线程处理，则应使用 AIDL 来定义接口。

这样，客户端并没有调用服务的“方法”。而客户端传递的“消息”（`Message` 对象）是服务在其 `Handler` 中接收的。

以下是一个使用 `Messenger` 接口的简单服务示例：

```
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.
     */
    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_SAY_HELLO:
                    Toast.makeText(getApplicationContext(), "hello!", Toast.LENGTH_SHORT).show();
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }

    /**
     * Target we publish for clients to send messages to IncomingHandler.
     */
    final Messenger mMessenger = new Messenger(new IncomingHandler());

    /**
     * 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).show();
        return mMessenger.getBinder();
    }
}
```

请注意，服务就是在 `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` 创建一个 `Messenger`，然后利用 `send()` 发送一条消息。例如，以下就是一个绑定到服务并向服务传递 `MSG_SAY_HELLO` 消息的简单 Activity：

```
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);
        try {
            mService.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

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

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to the service
        bindService(new Intent(this, MessengerService.class), mConnection,
            Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        // Unbind from the service
        if (mBound) {
            unbindService(mConnection);
            mBound = false;
        }
    }
}
```

请注意，此示例并未说明服务如何对客户端作出响应。如果您想让服务作出响应，则还需要在客户端中创建一个 `Messenger`。然后，当客户端收到 `onServiceConnected()` 回调时，会向服务发送一条 `Message`，并在其 `send()` 方法的 `replyTo` 参数中包含客户端的 `Messenger`。

如需查看如何提供双向消息传递的示例，请参阅 `MessengerService.java`（服务）和 `MessengerServiceActivities.java`（客户端）示例。

绑定到服务

应用组件（客户端）可通过调用 `bindService()` 绑定到服务。Android 系统随后调用服务的 `onBind()` 方法，该方法返回用于与服务交互的 `IBinder`。

绑定是异步的。`bindService()` 会立即返回，“不会”使 `IBinder` 返回客户端。要接收 `IBinder`，客户端必须创建一个 `ServiceConnection` 实例，并将其传递给 `bindService()`。`ServiceConnection` 包括一个回调方法，系统通过调用它来传递 `IBinder`。

> **注**：只有 Activity、服务和内容提供程序可以绑定到服务 — 您**无法**从广播接收器绑定到服务。

因此，要想从您的客户端绑定到服务，您必须：

1. 实现 `ServiceConnection`。

   您的实现必须重写两个回调方法：

   `onServiceConnected()`

   系统会调用该方法以传递服务的 `onBind()` 方法返回的 `IBinder`。

   `onServiceDisconnected()`

   Android 系统会在与服务的连接意外中断时（例如当服务崩溃或被终止时）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()`，传递 `ServiceConnection` 实现。
3. 当系统调用您的 `onServiceConnected()` 回调方法时，您可以使用接口定义的方法开始调用服务。
4. 要断开与服务的连接，请调用 `unbindService()`。

   如果应用在客户端仍绑定到服务时销毁客户端，则销毁会导致客户端取消绑定。 更好的做法是在客户端与服务交互完成后立即取消绑定客户端。 这样可以关闭空闲服务。如需了解有关绑定和取消绑定的适当时机的详细信息，请参阅附加说明。

例如，以下代码段通过扩展 Binder 类将客户端与上面创建的服务相连，因此它只需将返回的 `IBinder` 转换为 `LocalService` 类并请求 `LocalService` 实例：

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Called when the connection with the service is established
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Because we have bound to an explicit
        // service that is running in our own process, we can
        // cast its IBinder to a concrete class and directly access it.
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }

    // Called when the connection with the service disconnects unexpectedly
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "onServiceDisconnected");
        mBound = false;
    }
};
```

客户端可通过将此 `ServiceConnection` 传递至 `bindService()` 绑定到服务。例如：

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

- `bindService()` 的第一个参数是一个 `Intent`，用于显式命名要绑定的服务（但 Intent 可能是隐式的）
- 第二个参数是 `ServiceConnection` 对象

- 第三个参数是一个指示绑定选项的标志。它通常应该是 `BIND_AUTO_CREATE`，以便创建尚未激活的服务。其他可能的值为 `BIND_DEBUG_UNBIND` 和 `BIND_NOT_FOREGROUND`，或 `0`（表示无）。

附加说明

以下是一些有关绑定到服务的重要说明：

- 您应该始终捕获 `DeadObjectException` 异常，它们是在连接中断时引发的。这是远程方法引发的唯一异常。
- 对象是跨进程计数的引用。
- 您通常应该在客户端生命周期的匹配引入 (bring-up) 和退出 (tear-down) 时刻期间配对绑定和取消绑定。 例如：
  - 如果您只需要在 Activity 可见时与服务交互，则应在 `onStart()` 期间绑定，在 `onStop()` 期间取消绑定。
  - 如果您希望 Activity 在后台停止运行状态下仍可接收响应，则可在 `onCreate()` 期间绑定，在 `onDestroy()` 期间取消绑定。请注意，这意味着您的 Activity 在其整个运行过程中（甚至包括后台运行期间）都需要使用服务，因此如果服务位于其他进程内，那么当您提高该进程的权重时，系统终止该进程的可能性会增加。

> **注**：通常情况下，**切勿**在 Activity 的 `onResume()` 和 `onPause()` 期间绑定和取消绑定，因为每一次生命周期转换都会发生这些回调，您应该使发生在这些转换期间的处理保持在最低水平。此外，如果您的应用内的多个 Activity 绑定到同一服务，并且其中两个 Activity 之间发生了转换，则如果当前 Activity 在下一个 Activity 绑定（恢复期间）之前取消绑定（暂停期间），系统可能会销毁服务并重建服务。（Activity文档中介绍了这种有关 Activity 如何协调其生命周期的 Activity 转换。）

如需查看更多显示如何绑定到服务的示例代码，请参阅 ApiDemos 中的 `RemoteService.java` 类。

管理绑定服务的生命周期

当服务与所有客户端之间的绑定全部取消时，Android 系统便会销毁服务（除非还使用 `onStartCommand()` 启动了该服务）。因此，如果您的服务是纯粹的绑定服务，则无需对其生命周期进行管理 — Android 系统会根据它是否绑定到任何客户端代您管理。

不过，如果您选择实现 `onStartCommand()` 回调方法，则您必须显式停止服务，因为系统现在已将服务视为*已启动*。在此情况下，服务将一直运行到其通过 `stopSelf()` 自行停止，或其他组件调用 `stopService()` 为止，无论其是否绑定到任何客户端。

此外，如果您的服务已启动并接受绑定，则当系统调用您的 `onUnbind()` 方法时，如果您想在客户端下一次绑定到服务时接收 `onRebind()` 调用，则可选择返回 `true`。`onRebind()` 返回空值，但客户端仍在其 `onServiceConnected()` 回调中接收 `IBinder`。下文图 1 说明了这种生命周期的逻辑。

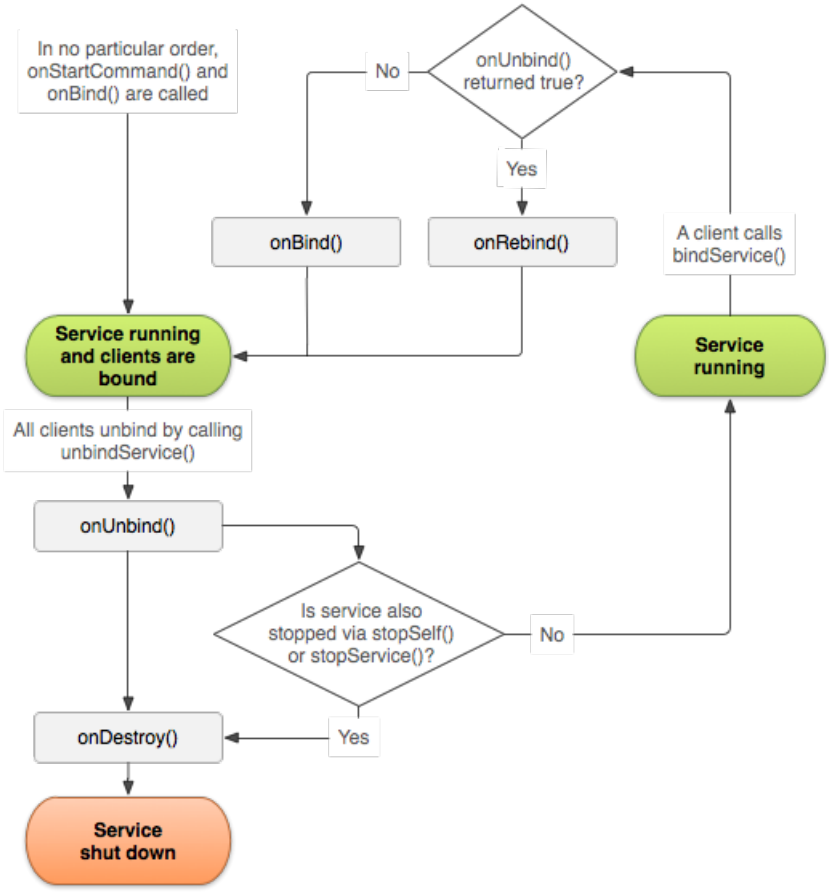

图 1. 允许绑定的已启动服务的生命周期。

如需了解有关已启动服务生命周期的详细信息，请参阅服务文档。