Android 接口定义语言 (AIDL)

本文内容

另请参阅

AIDL（Android 接口定义语言）与您可能使用过的其他 IDL 类似。 您可以利用它定义客户端与服务使用进程间通信 (IPC) 进行相互通信时都认可的编程接口。 在 Android 上，一个进程通常无法访问另一个进程的内存。 尽管如此，进程需要将其对象分解成操作系统能够识别的原语，并将对象编组成跨越边界的对象。 编写执行这一编组操作的代码是一项繁琐的工作，因此 Android 会使用 AIDL 来处理。

> **注**：只有允许不同应用的客户端用 IPC 方式访问服务，并且想要在服务中处理多线程时，才有必要使用 AIDL。 如果您不需要执行跨越不同应用的并发 IPC，就应该通过实现一个 Binder 创建接口；或者，如果您想执行 IPC，但根本不需要处理多线程，则使用 Messenger 类来实现接口。无论如何，在实现 AIDL 之前，请您务必理解绑定服务。

在您开始设计 AIDL 接口之前，要注意 AIDL 接口的调用是直接函数调用。 您不应该假设发生调用的线程。 视调用来自本地进程还是远程进程中的线程，实际情况会有所差异。 具体而言：

- 来自本地进程的调用在发起调用的同一线程内执行。如果该线程是您的主 UI 线程，则该线程继续在 AIDL 接口中执行。 如果该线程是其他线程，则其便是在服务中执行您的代码的线程。 因此，只有在本地线程访问服务时，您才能完全控制哪些线程在服务中执行（但如果真是这种情况，您根本不应该使用 AIDL，而是应该通过实现 Binder 类创建接口）。
- 来自远程进程的调用分派自平台在您的自有进程内部维护的线程池。 您必须为来自未知线程的多次并发传入调用做好准备。 换言之，AIDL 接口的实现必须是完全线程安全实现。
- `oneway` 关键字用于修改远程调用的行为。使用该关键字时，远程调用不会阻塞；它只是发送事务数据并立即返回。接口的实现最终接收此调用时，是以正常远程调用形式将其作为来自 `Binder` 线程池的常规调用进行接收。 如果 `oneway` 用于本地调用，则不会有任何影响，调用仍是同步调用。

定义 AIDL 接口

您必须使用 Java 编程语言语法在 `.aidl` 文件中定义 AIDL 接口，然后将它保存在托管服务的应用以及任何其他绑定到服务的应用的源代码（`src/` 目录）内。

您开发每个包含 `.aidl` 文件的应用时，Android SDK 工具都会生成一个基于该 `.aidl` 文件的 `IBinder` 接口，并将其保存在项目的 `gen/` 目录中。服务必须视情况实现 `IBinder` 接口。然后客户端应用便可绑定到该服务，并调用 `IBinder` 中的方法来执行 IPC。

如需使用 AIDL 创建绑定服务，请执行以下步骤：

1. 创建 .aidl 文件

   此文件定义带有方法签名的编程接口。

2. 实现接口

Android SDK 工具基于您的 `.aidl` 文件，使用 Java 编程语言生成一个接口。此接口具有一个名为 `Stub` 的内部抽象类，用于扩展 `Binder` 类并实现 AIDL 接口中的方法。您必须扩展 `Stub` 类并实现方法。

3. **向客户端公开该接口**

   实现 `Service` 并重写 `onBind()` 以返回 `Stub` 类的实现。

> **注意**：在 AIDL 接口首次发布后对其进行的任何更改都必须保持向后兼容性，以避免中断其他应用对您的服务的使用。 也就是说，因为必须将您的 `.aidl` 文件复制到其他应用，才能让这些应用访问您的服务的接口，因此您必须保留对原始接口的支持。

1. 创建 .aidl 文件

AIDL 使用简单语法，使您能通过可带参数和返回值的一个或多个方法来声明接口。 参数和返回值可以是任意类型，甚至可以是其他 AIDL 生成的接口。

您必须使用 Java 编程语言构建 `.aidl` 文件。每个 `.aidl` 文件都必须定义单个接口，并且只需包含接口声明和方法签名。

默认情况下，AIDL 支持下列数据类型：

- Java 编程语言中的所有原语类型（如 `int`、`long`、`char`、`boolean` 等等）
- `String`
- `CharSequence`
- `List`

  `List` 中的所有元素都必须是以上列表中支持的数据类型、其他 AIDL 生成的接口或您声明的可打包类型。 可选择将 `List` 用作“通用”类（例如，`List<String>`）。另一端实际接收的具体类始终是 `ArrayList`，但生成的方法使用的是 `List` 接口。
- `Map`

  `Map` 中的所有元素都必须是以上列表中支持的数据类型、其他 AIDL 生成的接口或您声明的可打包类型。 不支持通用 Map（如 `Map<String,Integer>` 形式的 Map）。 另一端实际接收的具体类始终是 `HashMap`，但生成的方法使用的是 `Map` 接口。

您必须为以上未列出的每个附加类型加入一个 `import` 语句，即使这些类型是在与您的接口相同的软件包中定义。

定义服务接口时，请注意：

- 方法可带零个或多个参数，返回值或空值。
- 所有非原语参数都需要指示数据走向的方向标记。可以是 `in`、`out` 或 `inout`（见以下示例）。

  原语默认为 `in`，不能是其他方向。

  > **注意**：您应该将方向限定为真正需要的方向，因为编组参数的开销极大。
- `.aidl` 文件中包括的所有代码注释都包含在生成的 `IBinder` 接口中（import 和 package 语句之前的注释除外）
- 只支持方法；您不能公开 AIDL 中的静态字段。

以下是一个 `.aidl` 文件示例：

```
// 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);
}
```

只需将您的 `.aidl` 文件保存在项目的 `src/` 目录内，当您开发应用时，SDK 工具会在项目的 `gen/` 目录中生成 `IBinder` 接口文件。生成的文件名与 `.aidl` 文件名一致，只是使用了 `.java` 扩展名（例如，`IRemoteService.aidl` 生成的文件名是 `IRemoteService.java`）。

如果您使用 Android Studio，增量编译几乎会立即生成 Binder 类。 如果您不使用 Android Studio，则 Gradle 工具会在您下一次开发应用时生成 Binder 类 — 您应该在编写完 `.aidl` 文件后立即用 `gradle assembleDebug` （或 `gradle assembleRelease`）编译项目，以便您的代码能够链接到生成的类。

2. 实现接口

当您开发应用时，Android SDK 工具会生成一个以 `.aidl` 文件命名的 `.java` 接口文件。生成的接口包括一个名为 `Stub` 的子类，这个子类是其父接口（例如，`YourInterface.Stub`）的抽象实现，用于声明 `.aidl` 文件中的所有方法。

> **注：**`Stub` 还定义了几个帮助程序方法，其中最引人关注的是 `asInterface()`，该方法带 `IBinder`（通常便是传递给客户端 `onServiceConnected()` 回调方法的参数）并返回存根接口实例。 如需了解如何进行这种转换的更多详细信息，请参见调用 IPC 方法一节。

如需实现 `.aidl` 生成的接口，请扩展生成的 `Binder` 接口（例如，`YourInterface.Stub`）并实现从 `.aidl` 文件继承的方法。

以下是一个使用匿名实例实现名为 `IRemoteService` 的接口（由以上 `IRemoteService.aidl` 示例定义）的示例：

```
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
    }
};
```

现在，`mBinder` 是 `Stub` 类的一个实例（一个 `Binder`），用于定义服务的 RPC 接口。 在下一步中，将向客户端公开该实例，以便客户端能与服务进行交互。

在实现 AIDL 接口时应注意遵守以下这几个规则：

- 由于不能保证在主线程上执行传入调用，因此您一开始就需要做好多线程处理准备，并将您的服务正确地编译为线程安全服务。
- 默认情况下，RPC 调用是同步调用。如果您明知服务完成请求的时间不止几毫秒，就不应该从 Activity 的主线程调用服务，因为这样做可能会使应用挂起（Android 可能会显示“Application is Not Responding”对话框）— 您通常应该从客户端内的单独线程调用服务。
- 您引发的任何异常都不会回传给调用方。

3. 向客户端公开该接口

您为服务实现该接口后，就需要向客户端公开该接口，以便客户端进行绑定。 要为您的服务公开该接口，请扩展 `Service` 并实现 `onBind()`，以返回一个类实例，这个类实现了生成的 `Stub`（见前文所述）。以下是一个向客户端公开 `IRemoteService` 示例接口的服务示例。

```
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
        }
    };
}
```

现在，当客户端（如 Activity）调用 `bindService()` 以连接此服务时，客户端的 `onServiceConnected()` 回调会接收服务的 `onBind()` 方法返回的 `mBinder` 实例。

客户端还必须具有对 interface 类的访问权限，因此如果客户端和服务在不同的应用内，则客户端的应用 `src/` 目录内必须包含 `.aidl` 文件（它生成 `android.os.Binder` 接口 — 为客户端提供对 AIDL 方法的访问权限）的副本。

当客户端在 `onServiceConnected()` 回调中收到 `IBinder` 时，它必须调用 *`YourServiceInterface`*`.Stub.asInterface(service)` 以将返回的参数转换成 *`YourServiceInterface`* 类型。例如：

```
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;
    }
};
```

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

通过 IPC 传递对象

通过 IPC 接口把某个类从一个进程发送到另一个进程是可以实现的。 不过，您必须确保该类的代码对 IPC 通道的另一端可用，并且该类必须支持 `Parcelable` 接口。支持 `Parcelable` 接口很重要，因为 Android 系统可通过它将对象分解成可编组到各进程的原语。

如需创建支持 `Parcelable` 协议的类，您必须执行以下操作：

1. 让您的类实现 `Parcelable` 接口。
2. 实现 `writeToParcel`，它会获取对象的当前状态并将其写入 `Parcel`。
3. 为您的类添加一个名为 `CREATOR` 的静态字段，这个字段是一个实现 `Parcelable.Creator` 接口的对象。
4. 最后，创建一个声明可打包类的 `.aidl` 文件（按照下文 `Rect.aidl` 文件所示步骤）。

   如果您使用的是自定义编译进程，切勿在您的编译中添加 `.aidl` 文件。 此 `.aidl` 文件与 C 语言中的头文件类似，并未编译。

AIDL 在它生成的代码中使用这些方法和字段将您的对象编组和取消编组。

例如，以下这个 `Rect.aidl` 文件可创建一个可打包的 `Rect` 类：

```
package android.graphics;

// Declare Rect so AIDL can find it and knows that it implements
// the parcelable protocol.
parcelable Rect;
```

以下示例展示了 `Rect` 类如何实现 `Parcelable` 协议。

```
import android.os.Parcel;
import android.os.Parcelable;

public final class Rect implements Parcelable {
    public int left;
    public int top;
    public int right;
    public int bottom;

    public static final Parcelable.Creator<Rect> CREATOR = new
Parcelable.Creator<Rect>() {
        public Rect createFromParcel(Parcel in) {
            return new Rect(in);
        }

        public Rect[] newArray(int size) {
            return new Rect[size];
        }
    };

    public Rect() {
    }

    private Rect(Parcel in) {
        readFromParcel(in);
    }

    public void writeToParcel(Parcel out) {
        out.writeInt(left);
        out.writeInt(top);
        out.writeInt(right);
        out.writeInt(bottom);
    }

    public void readFromParcel(Parcel in) {
        left = in.readInt();
        top = in.readInt();
        right = in.readInt();
        bottom = in.readInt();
    }
}
```

`Rect` 类中的编组相当简单。看一看 `Parcel` 上的其他方法，了解您可以向 Parcel 写入哪些其他类型的值。

> **警告：**别忘记从其他进程接收数据的安全影响。 在本例中，`Rect` 从 `Parcel` 读取四个数字，但要由您来确保无论调用方目的为何这些数字都在相应的可接受值范围内。 如需了解有关如何防止应用受到恶意软件侵害、保证应用安全的更多信息，请参见安全与权限。

调用 IPC 方法

调用类必须执行以下步骤，才能调用使用 AIDL 定义的远程接口：

1. 在项目 `src/` 目录中加入 `.aidl` 文件。

2. 声明一个 `IBinder` 接口实例（基于 AIDL 生成）。

3. 实现 `ServiceConnection`。
4. 调用 `Context.bindService()`，以传入您的 `ServiceConnection` 实现。
5. 在您的 `onServiceConnected()` 实现中，您将收到一个 `IBinder` 实例（名为 `service`）。调用 *`YourInterfaceName`*`.Stub.asInterface((IBinder)`*`service`*`)`，以将返回的参数转换为 *YourInterface* 类型。
6. 调用您在接口上定义的方法。您应该始终捕获 `DeadObjectException` 异常，它们是在连接中断时引发的；这将是远程方法引发的唯一异常。
7. 如需断开连接，请使用您的接口实例调用 `Context.unbindService()`。

有关调用 IPC 服务的几点说明：

- 对象是跨进程计数的引用。
- 您可以将匿名对象作为方法参数发送。

如需了解有关绑定到服务的详细信息，请阅读绑定服务文档。

以下这些示例代码摘自 ApiDemos 项目的远程服务示例代码，展示了如何调用 AIDL 创建的服务。

```
public static class Binding extends Activity {
    /** The primary interface we will be calling on the service. */
    IRemoteService mService = null;
    /** Another interface we use on the service. */
    ISecondary mSecondaryService = null;

    Button mKillButton;
    TextView mCallbackText;

    private boolean mIsBound;

    /**
     * Standard initialization of this activity.  Set up the UI, then wait
     * for the user to poke it before doing anything.
     */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.remote_service_binding);

        // Watch for button clicks.
        Button button = (Button)findViewById(R.id.bind);
        button.setOnClickListener(mBindListener);
        button = (Button)findViewById(R.id.unbind);
        button.setOnClickListener(mUnbindListener);
        mKillButton = (Button)findViewById(R.id.kill);
        mKillButton.setOnClickListener(mKillListener);
        mKillButton.setEnabled(false);

        mCallbackText = (TextView)findViewById(R.id.callback);
        mCallbackText.setText("Not attached.");
    }

    /**
     * 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 service object we can use to
            // interact with the service.  We are communicating with our
            // service through an IDL interface, so get a client-side
            // representation of that from the raw service object.
            mService = IRemoteService.Stub.asInterface(service);
            mKillButton.setEnabled(true);
            mCallbackText.setText("Attached.");
```

```
            // We want to monitor the service for as long as we are
            // connected to it.
            try {
                mService.registerCallback(mCallback);
            } catch (RemoteException e) {
                // In this case the service has crashed before we could even
                // do anything with it; we can count on soon being
                // disconnected (and then reconnected if it can be restarted)
                // so there is no need to do anything here.
            }

            // As part of the sample, tell the user what happened.
            Toast.makeText(Binding.this, R.string.remote_service_connected,
                    Toast.LENGTH_SHORT).show();
        }

        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;
            mKillButton.setEnabled(false);
            mCallbackText.setText("Disconnected.");

            // As part of the sample, tell the user what happened.
            Toast.makeText(Binding.this, R.string.remote_service_disconnected,
                    Toast.LENGTH_SHORT).show();
        }
    };

    /**
     * Class for interacting with the secondary interface of the service.
     */
    private ServiceConnection mSecondaryConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className,
                IBinder service) {
            // Connecting to a secondary interface is the same as any
            // other interface.
            mSecondaryService = ISecondary.Stub.asInterface(service);
            mKillButton.setEnabled(true);
        }

        public void onServiceDisconnected(ComponentName className) {
            mSecondaryService = null;
            mKillButton.setEnabled(false);
        }
    };

    private OnClickListener mBindListener = new OnClickListener() {
        public void onClick(View v) {
            // Establish a couple connections with the service, binding
            // by interface names.  This allows other applications to be
            // installed that replace the remote service by implementing
            // the same interface.
            Intent intent = new Intent(Binding.this, RemoteService.class);
            intent.setAction(IRemoteService.class.getName());
            bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
            intent.setAction(ISecondary.class.getName());
            bindService(intent, mSecondaryConnection, Context.BIND_AUTO_CREATE);
            mIsBound = true;
            mCallbackText.setText("Binding.");
        }
    };

    private OnClickListener mUnbindListener = new OnClickListener() {
        public void onClick(View v) {
            if (mIsBound) {
                // If we have received the service, and hence registered with
                // it, then now is the time to unregister.
                if (mService != null) {
                    try {
                        mService.unregisterCallback(mCallback);
                    } catch (RemoteException e) {
                        // There is nothing special we need to do if the service
```

```
                    // has crashed.
                }
            }

            // Detach our existing connection.
            unbindService(mConnection);
            unbindService(mSecondaryConnection);
            mKillButton.setEnabled(false);
            mIsBound = false;
            mCallbackText.setText("Unbinding.");
        }
    }
};

private OnClickListener mKillListener = new OnClickListener() {
    public void onClick(View v) {
        // To kill the process hosting our service, we need to know its
        // PID.  Conveniently our service has a call that will return
        // to us that information.
        if (mSecondaryService != null) {
            try {
                int pid = mSecondaryService.getPid();
                // Note that, though this API allows us to request to
                // kill any process based on its PID, the kernel will
                // still impose standard restrictions on which PIDs you
                // are actually able to kill.  Typically this means only
                // the process running your application and any additional
                // processes created by that app as shown here; packages
                // sharing a common UID will also be able to kill each
                // other's processes.
                Process.killProcess(pid);
                mCallbackText.setText("Killed service process.");
            } catch (RemoteException ex) {
                // Recover gracefully from the process hosting the
                // server dying.
                // Just for purposes of the sample, put up a notification.
                Toast.makeText(Binding.this,
                        R.string.remote_call_failed,
                        Toast.LENGTH_SHORT).show();
            }
        }
    }
};

// ----------------------------------------------------------------------
// Code showing how to deal with callbacks.
// ----------------------------------------------------------------------

/**
 * This implementation is used to receive callbacks from the remote
 * service.
 */
private IRemoteServiceCallback mCallback = new IRemoteServiceCallback.Stub() {
    /**
     * This is called by the remote service regularly to tell us about
     * new values.  Note that IPC calls are dispatched through a thread
     * pool running in each process, so the code executing here will
     * NOT be running in our main thread like most other things -- so,
     * to update the UI, we need to use a Handler to hop over there.
     */
    public void valueChanged(int value) {
        mHandler.sendMessage(mHandler.obtainMessage(BUMP_MSG, value, 0));
    }
};

private static final int BUMP_MSG = 1;

private Handler mHandler = new Handler() {
    @Override public void handleMessage(Message msg) {
        switch (msg.what) {
            case BUMP_MSG:
                mCallbackText.setText("Received from service: " + msg.arg1);
```

```
                    break;
                default:
                    super.handleMessage(msg);
            }
        }

    };
}
```