

第7讲 内容向导

7.1 本地服务

7.2 隐式和显式启动

7.3 线程启动挂起和停止



7.1 本地服务

■ Service简介

- Service是Android系统的服务组件，适用于开发没有用户界面且长时间在后台运行的应用功能
- 因为手机硬件性能和屏幕尺寸的限制，通常Android系统仅允许一个应用程序处于激活状态并显示在手机屏幕上，而暂停其他处于未激活状态的程序
- 因此，Android系统需要一种后台服务机制，允许在没有用户界面的情况下，使程序能够长时间在后台运行，实现应用程序的后台服务功能，并能够处理事件或数据更新

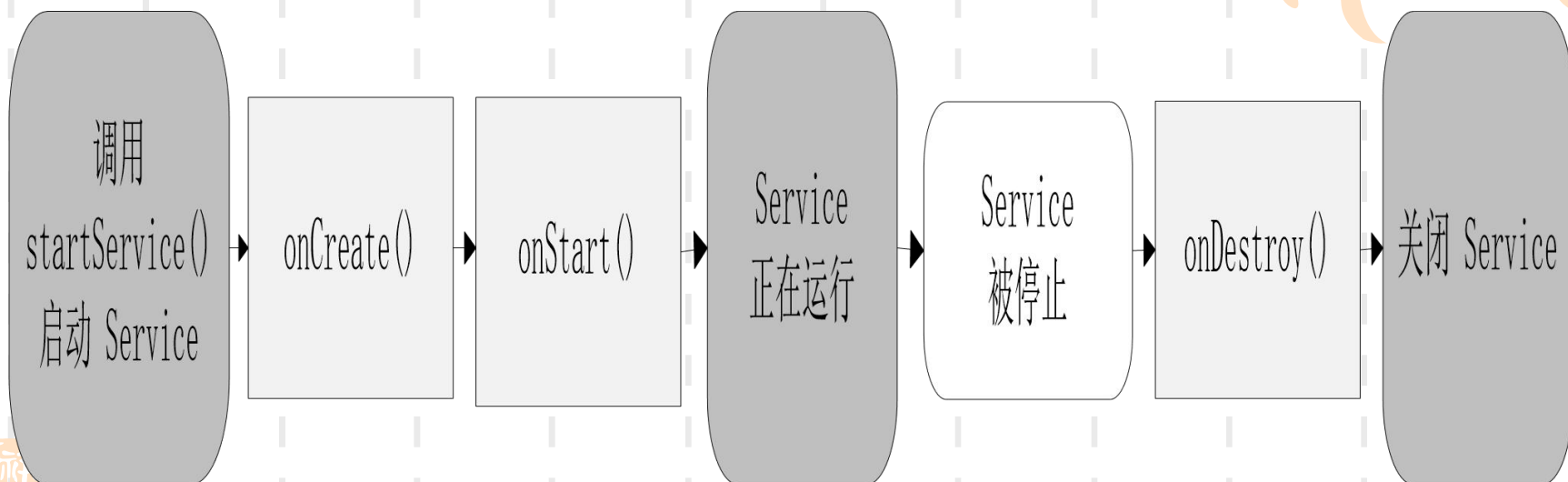
■ 7.1.1 Service生命周期

➤ Service生命周期包括

- 完全生命周期
- 活动生命周期
- **onCreate()**事件回调函数： **Service**的生命周期开始，完成**Service**的初始化工作
- **onStart()**事件回调函数： 活动生命周期开始，但没有与之对应的“停止”函数，因此可以近似认为活动生命周期也是以**onDestroy()**标志结束
- **onDestroy()**事件回调函数： **Service**的生命周期结束，释放**Service**所有占用的资源



吉祥如意



吉祥如意

吉祥如意

吉祥如意

吉祥如意

吉祥如意

■ Service的使用方式一般有两种

➤ 启动方式

➤ 绑定方式



■ 启动方式

- 通过调用 `Context.startService()` 启动 `Service`，通过调用 `Context.stopService()` 或 `Service.stopSelf()` 停止 `Service`。因此，`Service` 一定是由其它的组件启动的，但停止过程可以通过其它组件或自身完成
- 在启动方式中，启动 `Service` 的组件不能够获取到 `Service` 的对象实例，因此无法调用 `Service` 中的任何函数，也不能够获取到 `Service` 中的任何状态和数据信息
- 能够以启动方式使用的 `Service`，需要具备自我管理的能力，而且不需要从通过函数调用获取 `Service` 的功能和数据

■ 绑定方式

- Service的使用是通过服务链接（Connection）实现的，服务链接能够获取Service的对象实例，因此绑定Service的组件可以调用Service中实现的函数，或直接获取Service中的状态和数据信息
- 使用Service的组件通过Context.bindService()建立服务链接，通过Context.unbindService()停止服务链接
- 如果在绑定过程中Service没有启动，Context.bindService()会自动启动Service，而且同一个Service可以绑定多个服务链接，这样可以同时为多个不同的组件提供服务

■ 启动方式和绑定方式的结合

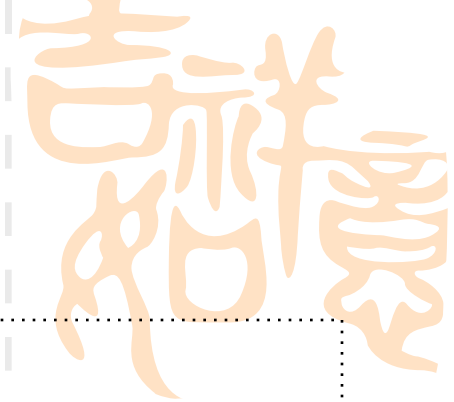
➤ 这两种使用方法并不是完全独立的，在某些情况下可以混合使用

- 以MP3播放器为例，在后台的工作的Service通过`Context.startService()`启动某个音乐播放，但在播放过程中如果用户需要暂停音乐播放，则需要通过`Context.bindService()`获取服务链接和Service对象实例，进而通过调用Service对象实例中的函数，暂停音乐播放过程，并保存相关信息

7.1.2 本地服务

- 本地服务的调用者和服务都在同一个程序中，是不需要跨进程就可以实现服务的调用
- 本地服务涉及服务的建立、启动和停止，服务的绑定和取消绑定，以及如何在线程中实现服务
- 服务管理
 - 服务管理主要指服务的启动和停止
 - 首先说明如何在代码中实现Service。Service是一段在后台运行、没有用户界面的代码，其最小代码集如下

■ 服务管理实例




```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class RandomService extends Service {
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```



■ 服务管理实例

```
public class RandomService extends Service{  
    @Override  
    public void onCreate() {  
        super.onCreate();  
    }  
    @Override  
    public void onStart(Intent intent, int startId) {  
        super.onStart(intent, startId);  
    }  
    @Override  
    public void onDestroy() {  
        super.onDestroy();  
    }  
}
```

- 
- 完成Service类后，需要在AndroidManifest.xml文件中注册这个Service
 - 注册Service非常重要，如果开发人员不对Service进行注册，则Service根本无法启动
 - AndroidManifest.xml文件中注册Service的代码如下

```
<service android:name=".RandomService"/>
```

- 使用<service>标签声明服务，其中的android:name表示Service类的名称，一定要与建立的Service类名称一致

7.2 隐式和显式启动

7.2.1 显示启动

在完成Service代码和在AndroidManifest.xml文件中注册后，下面来说明如何启动和停止Service。

- 有两种方法启动Service，显式启动和隐式启动
- 显式启动需要在Intent中指明Service所在的类，并调用startService(Intent)启动Service，示例代码如下

```
final Intent serviceIntent =  
    new Intent(this, RandomService.class);  
startService(serviceIntent);
```

7.2.2 隐式启动

需要在注册Service时，声明Intent-filter的action属性

```
<service android:name=".RandomService">  
    <intent-filter>  
        <action android:name="edu.cumt.RandomService" />  
    </intent-filter>  
</service>
```

- 在隐式启动**Service**时，需要设置**Intent**的**action**属性，这样则可以在不声明**Service**所在类的情况下启动服务。隐式启动的代码如下

```
final Intent serviceIntent = new Intent();  
serviceIntent.setAction("edu.cumt.RandomService");
```

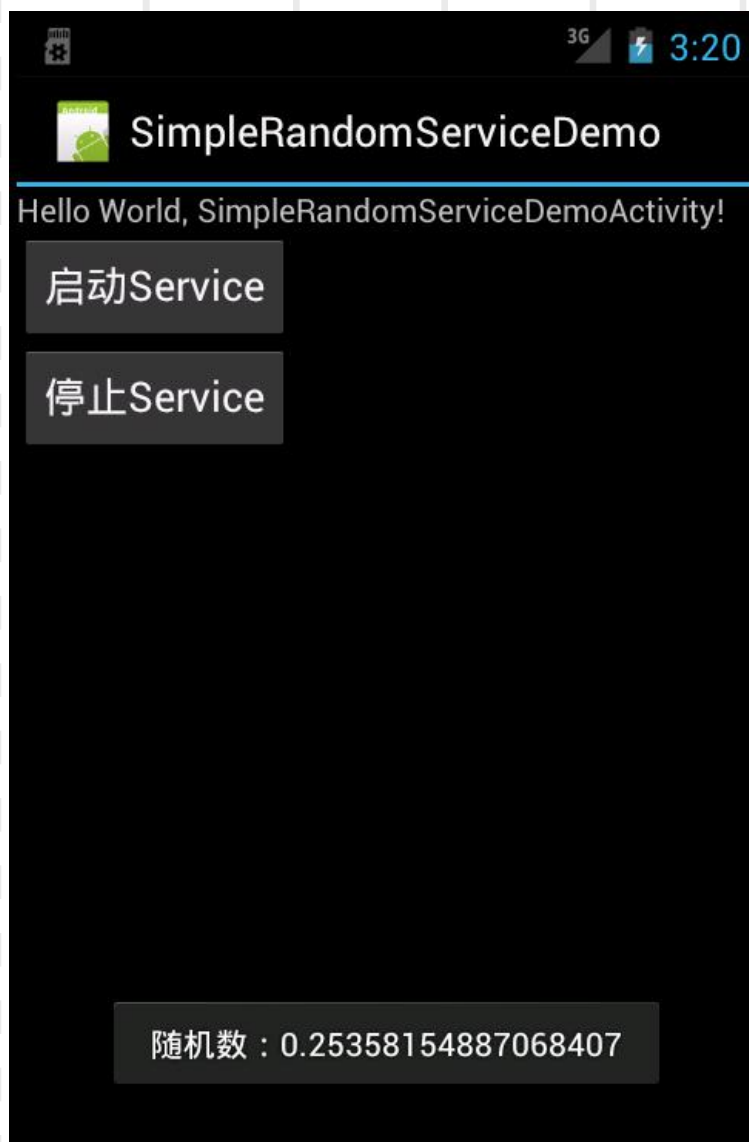
- 如果**Service**和调用服务的组件在同一个应用程序中，可以使用显式启动或隐式启动，显式启动更加易于使用，且代码简洁。但如果服务和调用服务的组件在不同的应用程序中，则只能使用隐式启动

- 无论是显式启动还是隐式启动，停止Service的方法都是相同的，将启动Service的Intent传递给 **stopService(Intent)** 函数即可，示例代码如下

stopService(serviceIntent);

- 在首次调用startService(Intent)函数启动Service后，系统会先后调用onCreate()和onStart()
- 如果是第二次调用startService(Intent)函数，系统则仅调用onStart()，而不再调用onCreate()
- 在调用stopService(Intent)函数停止Service时，系统会调用onDestroy()
- 无论调用过多少次startService(Intent)，在调用stopService(Intent)函数时，系统**仅调用一次onDestroy()**

➤ SimpleRandomServiceDemo是在应用程序中使用Service的示例

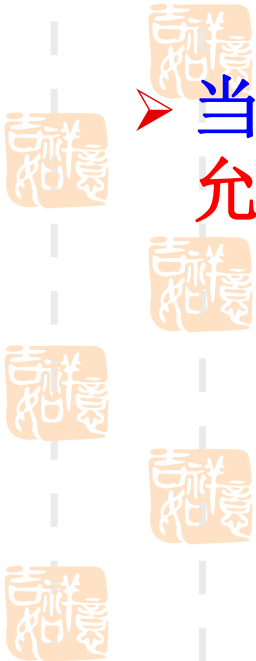


7.3线程启动挂起和停止

■ 7.3.1线程简介

- 在Android系统中，Activity、Service和BroadcastReceiver都是工作在主线程上，因此任何耗时的处理过程都会降低用户界面的响应速度，甚至导致用户界面失去响应

- 当用户界面失去响应超过5秒后，Android系统会允许用户强行关闭应用程序，提示如下图所示



ImplicitRandomServiceDemo
is not responding. Would you
like to close it?

Wait

OK

■ 7.3.2 使用线程

- 无论线程是否真的并行工作，在宏观上可以认为子线程是独立于主线程的，且能与主线程并行工作的程序单元
- 在Java语言中，建立和使用线程比较简单，首先需要实现Java的Runnable接口，并重载run()函数，在run()中放置代码的主体部分

```
private Runnable backgroudWork = new Runnable(){  
    @Override  
    public void run() {  
        //过程代码  
    }  
};
```

- 然后创建Thread对象，并将Runnable对象作为参数传递给Thread对象
- 在Thread的构造函数中，第1个参数用来表示线程组，第2个参数是需要执行的Runnable对象，第3个参数是线程的名称

```
private Thread workThread;
```

```
workThread = new
```

```
Thread(null,backgroundWork,"WorkThread");
```

- 最后，调用start()方法启动线程

```
workThread.start();
```

- 当线程在`run()`方法返回后，线程就自动终止了
- 当然，也可以调用`stop()`在外部终止线程，但这种方法并不推荐使用，因为这方法并不安全，有一定可能性会产生异常
- 最好的方法是通知线程自行终止，一般调用`interrupt()`方法通告线程准备终止，线程会释放它正在使用的资源，在完成所有的清理工作后自行关闭

`workThread.interrupt();`

- 其实`interrupt()`方法并不能直接终止线程，仅是改变了线程内部的一个布尔值，`run()`方法能够检测到这个布尔值的改变，从而在适当的时候释放资源和终止线程

➤ 在`run()`中的代码一般通过`Thread.interrupted()`方法查询线程是否被中断

➤ 一般情况下，子线程需要无限运行，除非外部调用`interrupt()`方法中断线程，所以通常会将程序主体放置在`while()`函数内，并调用`Thread.interrupted()`方法判断线程是否应被中断

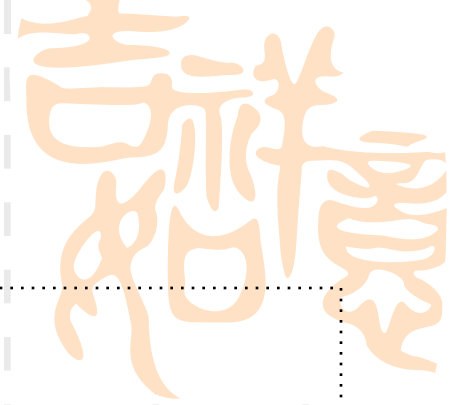
➤ 下面的代码中，以1秒为间隔循环检测线程是否应被中断

- 第4行代码使线程休眠1000毫秒
- 当线程在休眠过程中线程被中断，则会产生 `InterruptedException` 异常
- 因此代码中需要捕获 `InterruptedException` 异常，保证安全终止线程

```
1. public void run() {  
2.     while(!Thread.interrupted()){  
3.         //过程代码  
4.         Thread.sleep(1000);  
5.     }  
6. }
```


➤ 使用Handler更新用户界面

- Handler允许将Runnable对象发送到线程的消息队列中，每个Handler实例绑定到一个单独的线程和消息队列上
- 当用户建立一个新的Handler实例，通过post()方法将Runnable对象从后台线程发送给GUI线程的消息队列，当Runnable对象通过消息队列后，这个Runnable对象将被运行



■ 使用线程

```
private static Handler handler = new Handler();
```

```
public static void UpdateGUI(double refreshDouble){  
    handler.post(RefreshLable);  
}
```

```
private static Runnable RefreshLable = new Runnable(){
```

```
    @Override
```

```
    public void run() {
```

```
        //过程代码
```

```
    }
```

```
};
```





3G



8:34



ThreadRandomServiceDemo

0.9941514968083872

启动Service

停止Service

■ 7.3.3 服务绑定

- 以绑定方式使用Service，能够获取到Service实例，不仅能够正常启动Service，还能够调用Service中的公有方法和属性
- 为了使Service支持绑定，需要在Service类中重载onBind()方法，并在onBind()方法中返回Service实例，示例代码如下



■ 服务绑定

```
public class MathService extends Service{  
    private final IBinder mBinder = new LocalBinder();  
  
    public class LocalBinder extends Binder{  
        MathService getService() {  
            return MathService.this;  
        }  
    }  
  
    @Override  
    public IBinder onBind(Intent intent) {  
        return mBinder;  
    }  
}
```

■ 服务绑定

➤ 调用者通过bindService()函数绑定服务

- 调用者通过bindService()函数绑定服务，并在第1个参数中将Intent传递给bindService()函数，声明需要启动的Service
- 第3个参数Context.BIND_AUTO_CREATE表明只要绑定存在，就自动建立Service
- 同时也告知Android系统，这个Service的重要程度与调用者相同，除非考虑终止调用者，否则不要关闭这个Service

```
final Intent serviceIntent = new
```

```
Intent(this,MathService.class);
```

```
bindService(serviceIntent,mConnection,Context.BIND_
AUTO_CREATE);
```

■ 服务绑定

➤ bindService() 函数的第2个参数是 ServiceConnection

- 当绑定成功后，系统将调用 ServiceConnection 的 onServiceConnected() 方法

- 当绑定意外断开后，系统将调用 ServiceConnection 中的 onServiceDisconnected 方法

- 因此，以绑定方式使用 Service，调用者需要声明一个 ServiceConnection，并重载内部的 onServiceConnected() 方法和 onServiceDisconnected 方法，两个方法的重载代码如下



```
private ServiceConnection mConnection = new
    ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name,
            IBinder service) {
            mathService =
                ((MathService.LocalBinder)service).getService();
        }
        @Override
        public void onServiceDisconnected(ComponentName
            name) {
            mathService = null;
        }
    };
```



■ 服务绑定

- SimpleMathServiceDemo是绑定方式使用Service的示例



7.4 远程服务

■ 7.4.1 进程间通信

- 在Android系统中，每个应用程序在各自的进程中运行，而且出于安全原因的考虑，这些进程之间彼此是隔离的，进程之间传递数据和对象，需要使用Android支持的进程间通信（Inter-Process Communication, IPC）机制
- 在Unix/Linux系统中，传统的IPC机制包括共享内存、管道、消息队列和socket等等，这些IPC机制虽然被广泛使用，但仍然存在着固有的缺陷，如容易产生错误、难于维护等等
- 在Android系统中，没有使用传统的IPC机制，而是采用Intent和远程服务的方式实现IPC，使应用程序具有更好的独立性和鲁棒性

- Android系统允许应用程序使用Intent启动Activity和服务，同时Intent可以传递数据，是一种简单、高效、易于使用的IPC机制
- Android系统的另一种IPC机制就是远程服务，服务和调用者在不同的两个进程中，调用过程需要跨越进程才能实现
- 在Android系统中使用远程服务，一般按照以下三个步骤实现
 - 使用AIDL语言定义远程服务的接口
 - 根据AIDL语言定义的接口，在具体的Service类中实现接口中定义的方法和属性
 - 在需要调用远程服务的组件中，通过相同的AIDL接口文件，调用远程服务

■ 7.4.2 服务创建与调用

- 在Android系统中，进程之间不能直接访问相互的内存控件，因此为了使数据能够在不同进程间传递，数据必须转换成能够穿越进程边界的系统级原语，同时，在数据完成进程边界穿越后，还需要转换回原有的格式
- **AIDL** (Android Interface Definition Language) 是Android系统自定义的接口描述语言，可以简化进程间数据格式转换和数据交换的代码，通过定义Service内部的公共方法，允许在不同进程间的调用者和Service之间相互传递数据
- AIDL的IPC机制、COM和Corba都是基于接口的轻量级进程通信机制

➤ **AIDL语言**的语法与Java语言的接口定义非常相似，唯一不同之处在于，AIDL允许定义函数参数的传递方向

➤ **AIDL支持三种方向：in、out和inout**

- 标识为**in**的参数将从调用者传递到远程服务中
- 标识为**out**的参数将从远程服务传递到调用者中
- 标识为**inout**的参数将先从调用者传递到远程服务中，再从远程服务返回给调用者

➤ 如果不标识参数的传递方向，默认所有函数的传递方向为in

➤ 出于性能方面的考虑，不要在参数中标识不需要的传递方向

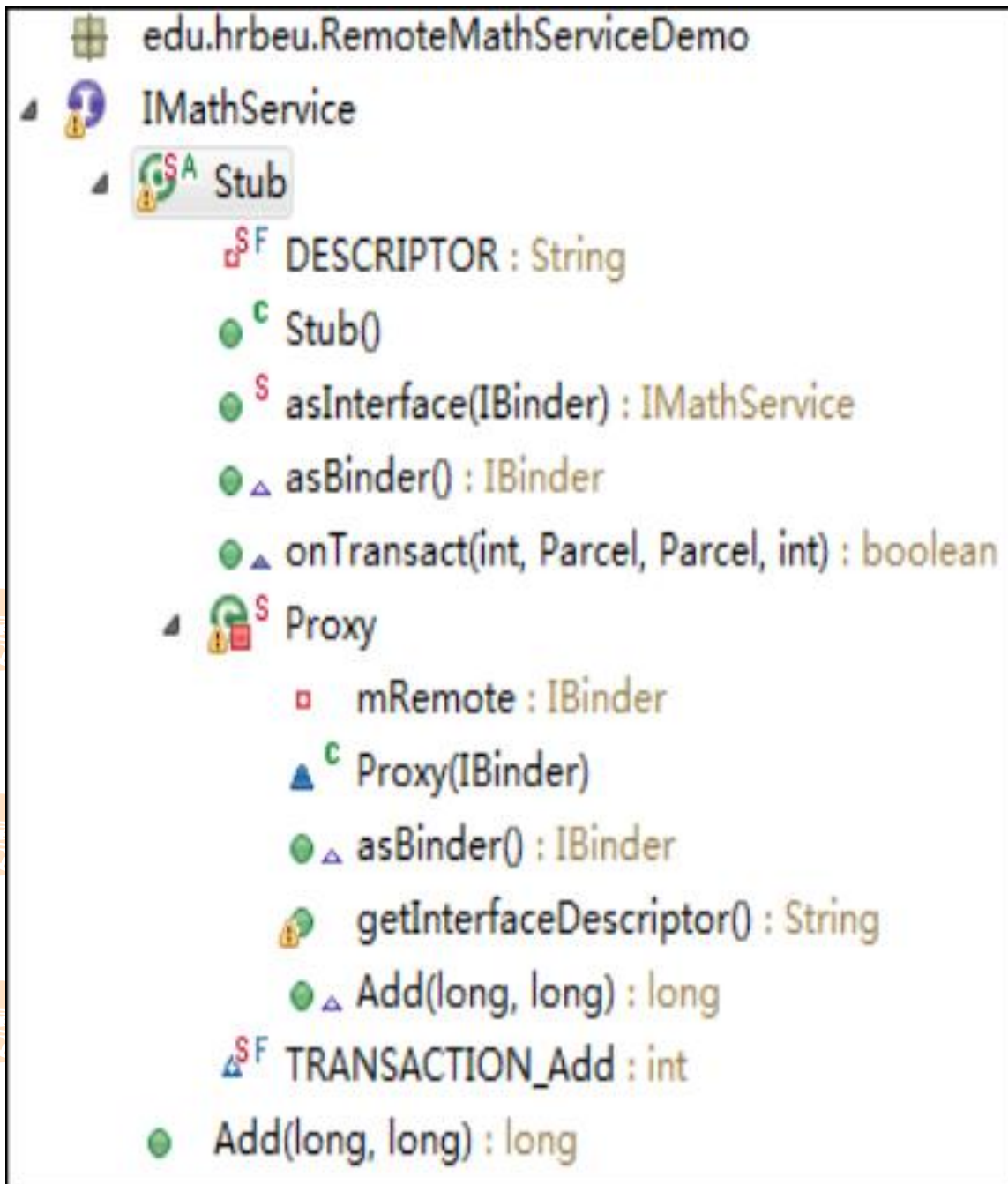
➤ 远程服务的创建和调用需要使用AIDL语言，一般分为以下几个过程

- 使用AIDL语言定义远程服务的接口
- 通过继承Service类实现远程服务
- 绑定和使用远程服务



- 下面以RemoteMathServiceDemo示例为参考，说明如何创建远程服务



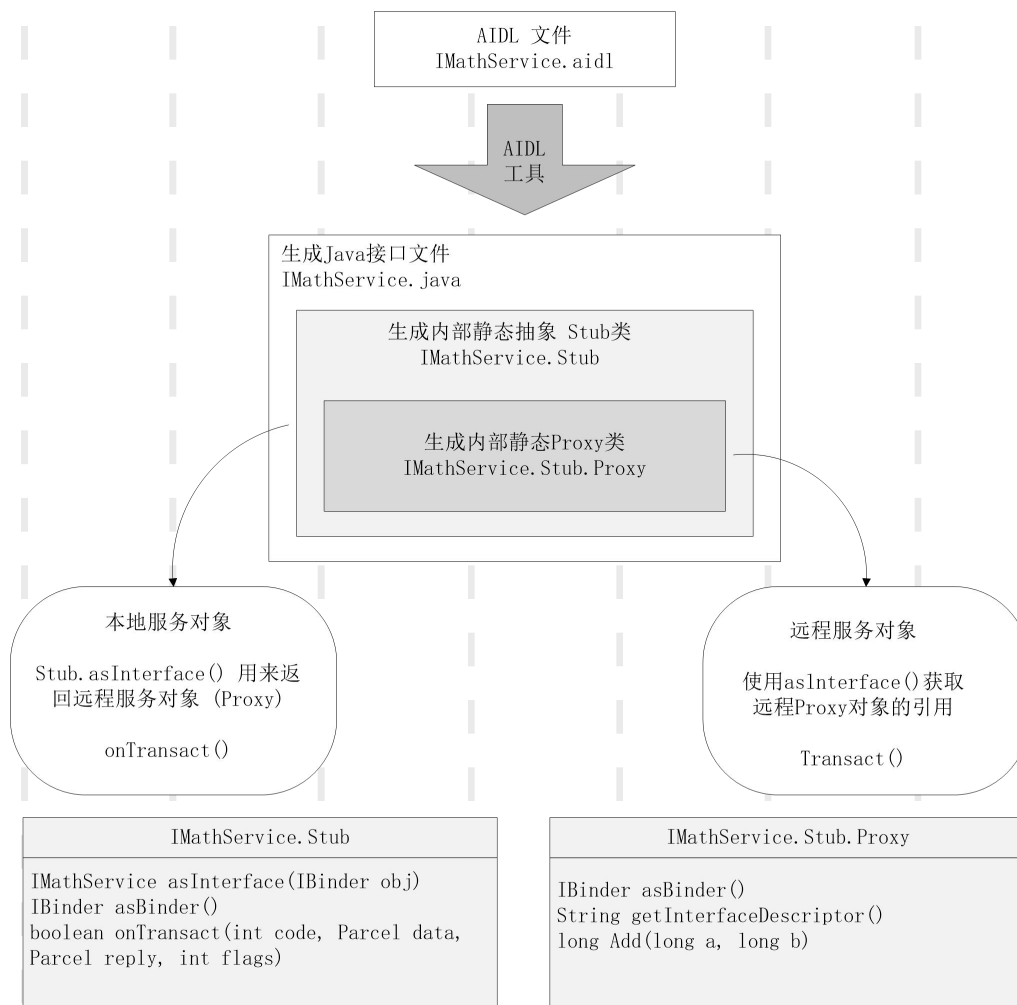


➤ 使用AIDL语言定义远程服务的接口

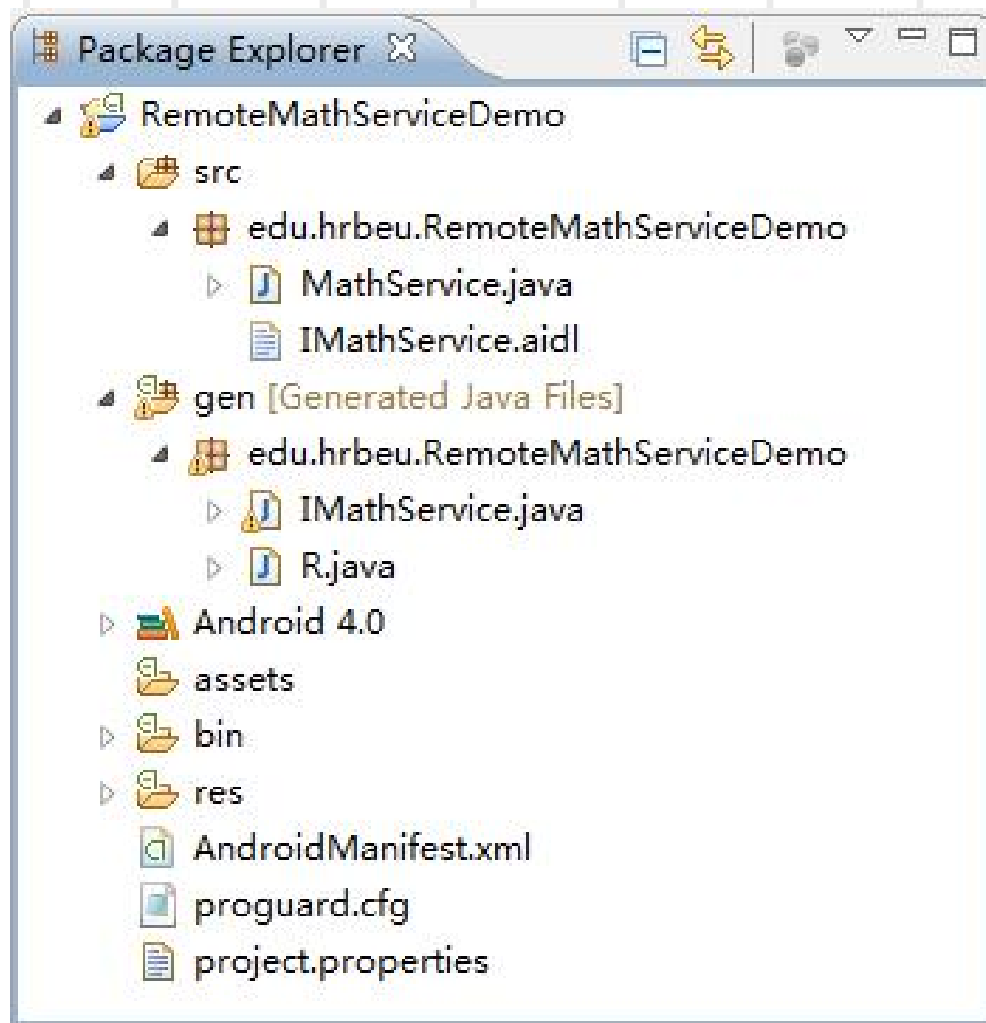
- IMathService.java文件根据IMathService.aidl的定义，生成了一个内部静态抽象类Stub，Stub继承了Binder类，并实现IMathService接口
- 在Stub类中，还包含一个重要的静态类Proxy。可以认为Stub类用来实现本地服务调用，Proxy类用来实现远程服务调用，将Proxy作为Stub的内部类完全是出于使用方便的目

➤ 使用AIDL语言定义远程服务的接口

■ Stub类和Proxy类关系图



- 通过继承**Service**类实现跨进程服务
 - **RemoteMathServiceDemo**示例文件结构



➤ 绑定和使用远程服务

- RemoteMathCallerDemo示例说明如何调用RemoteMathServiceDemo示例中的远程服务。RemoteMathCallerDemo的界面如下图所示



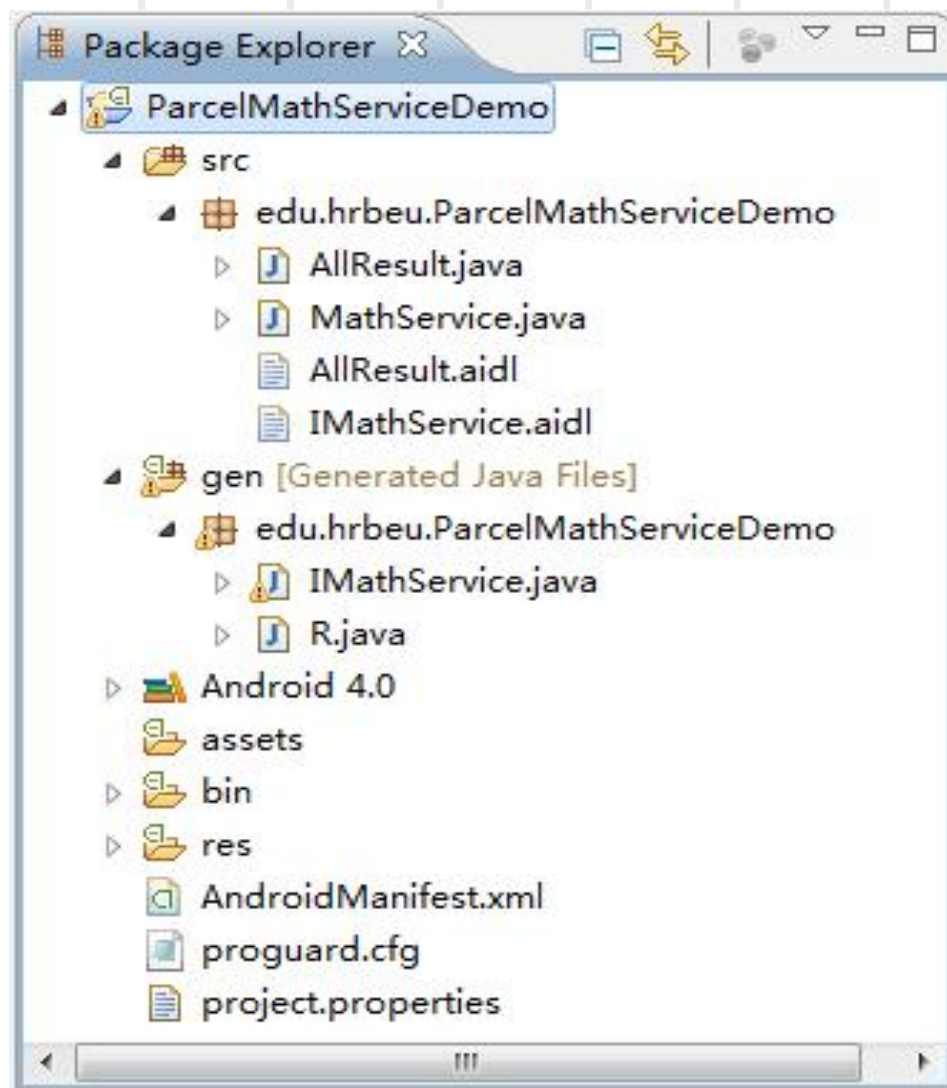
■ 7.4.3 数据传递

- 在Android系统中，进程间传递的数据包括：
 - Java语言支持的基本数据类型
 - 用户自定义的数据类型
- 为了使数据能够穿越进程边界，所有数据都必须是“可打包”的
- 对于Java语言的基本数据类型，打包过程是自动完成的
- 但对于自定义的数据类型，用户则需要实现Parcelable接口，使自定义的数据类型能够转换为系统级原语保存在Parcel对象中，穿越进程边界后可再转换为初始格式

➤ AIDL支持的数据类型表

类型	说明	需要引入
Java语言的基本类型	包括boolean、byte、short、int、float和double等	否
String	java.lang.String	否
CharSequence	java.lang.CharSequence	否
List	其中所有的元素都必须是AIDL支持的数据类型	否
Map	其中所有的键和元素都必须是AIDL支持的数据类型	
其它AIDL接口	任何其它使用AIDL语言生成的接口类型	是
Parcelable对象	实现Parcelable接口的对象	是

- 下面以ParcelMathServiceDemo示例为参考，说明如何在远程服务中使用自定义数据类型





3G



9:17



ParcelMathCallerDemo

$57 + 69 = 126$

$57 - 69 = -12$

$57 * 69 = 3933$

$57 / 69 = 0.8260869565217391$

远程服务绑定

取消服务绑定

全运算

Service与Activity进行通信

- Activity主要负责前台页面的展示
- Service主要负责需要长期运行的任务
- 而Service执行某些操作之后想要更新UI线程，就需要与Activity进行通信
 - 例如：后台执行下载任务，每秒更新一次进度，需要在前台显示下载的进度或者执行过程中的等待提示
 - 又比如：当前 Activity 利用广播向一个 Service 发送数据，Service 应当做出响应

Service与Activity进行通信

- Activity 调用 `bindService (Intent service, ServiceConnection conn, int flags)` 方法，得到 **Service** 对象的引用，这样 **Activity** 就可以直接调用 **Service** 中的方法了。

- **Service** 向 **Activity** 发送消息，可以利用 **Handler**，还可以使用广播。

举例：Service与Activity进行通信

- Activity_main界面，两个按钮，分别执行

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
```

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="send bind data" />
```

第一种通信方法：
调用**bindService**

```
<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="send broadcast data" />
```

第二种通信方法：
使用广播

```
</LinearLayout>
```

举例：Service与Activity进行通信

■ MainActivity.java

```
public class MainActivity extends Activity {
```

```
    private Button btn1;
```

```
    private Button btn2;
```

```
    /* 通过Binder，实现Activity与Service通信 */
```

```
    private
```

```
    private ServiceConnection connection = new ServiceConnection() {
```

```
        @Override
```

```
        public void                (ComponentName name) {
```

```
    }
```

```
        @Override
```

```
        public void                (ComponentName name, IBinder service) {
```

```
            mBinderService = (MainService.ServiceBinder) service;
```

```
            try {
```

```
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

第一种通信方法：
调用bindService

举例：Service与Activity进行通信

■ MainActivity.java

@Override

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);
```

第一种通信方法：
调用bindService

```
    btn1=(Button)findViewById(R.id.button1);  
    btn1.setOnClickListener(new OnClickListener() {
```

@Override

```
public void onClick(View arg0) {
```

```
    Intent bindIntent = new Intent(MainActivity.this,           );
```

调用bindService

```
    }  
});
```

举例：Service与Activity进行通信

■ MainActivity.java

```
btn2=(Button)findViewById(R.id.button2);  
btn2.setOnClickListener(new OnClickListener() {  
    @Override  
    public void onClick(View arg0) {
```

```
};  
});  
}
```

第二种通信方法：
使用广播

举例：Service与Activity进行通信

- Manifest.xml 注册Service和Broadcast

```
<application>
```

```
<activity>
```

```
.....
```

```
</activity>
```



举例：Service与Activity进行通信

■ MainService.java 第一种通信方式，

```
public class MainService extends Service{

    private String TAG = "MainService";

    /* 重写 Binder 的onBind() 函数，返回派生类 */
    @Override
    public IBinder onBind(Intent arg0) {
        return mBinder;
    }

    @Override
    public void onCreate() {
        Toast.makeText( MainService.this, "Service Create...", Toast.LENGTH_SHORT).show();
        super.onCreate();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Toast.makeText(MainService.this, "Service StartCommand",
            Toast.LENGTH_SHORT).show();
    }
}
```


举例：Service与Activity进行通信

■ MainService.java 第一种通信方式，

```
@Override
public void onDestroy() {
    Toast.makeText( MainService.this, "Service Destroy", Toast.LENGTH_SHORT).show();
}
```

/ 第一种模式通信：Binder */*

```
public void                                throws InterruptedException {
    /* 模拟下载，休眠2秒 */
    Toast.makeText( MainService.this, “模拟下载2秒钟,, 开始下载...”,
    Toast.LENGTH_SHORT).show();

    Toast.makeText( MainService.this, "下载结束...",
    Toast.LENGTH_SHORT).show();
}
}
```

举例：Service与Activity进行通信

■ BroadcastTest.java 第二种通信方式，

```
public class                                {  
  
    //private NetworkInfo netInfo;  
    //private ConnectivityManager mConnectivityManager;  
  
    public void                            (Context context, Intent intent) {  
  
        if (extras != null) {  
            if(extras.containsKey("value")){  
                Toast.makeText( context, "收到广播 => "+  
Toast.LENGTH_SHORT).show();  
                System.out.println("Value is:"+extras.get("value"));  
            }  
        }  
    }  
}
```

第二种通信方法：
使用广播

android中的通信机制总结(1/4)

1、使用 handler 进行通信

- handler 是主线程(UI线程)的一个子线程
- 它给主线程(UI 线程)发送数据，更新主线程
- 它的耗时操作不会阻塞主线程（阻塞主线程超过5秒，系统会系统提示强制关闭ANR）

android中的通信机制总结(2/4)

2、Activity 转跳

- 转跳的实现方式是使用 Intent ，然后 startActivity 。
- 转跳可以带数据。
 - 比如从 A 跳到 B 可以把 A 中的一些数据通过 Intent.putExtra() 传递给 B 。

android中的通信机制总结(3/4)

3、广播的发送与接收

➤ **Android** 开发中如果要对两个完全没关系的程序之间进行通信，可以使用发送接收广播的机制来实现。

➤ 例如程序 **A** 发送了一个广播，程序 **B** 接收广播。

android中的通信机制总结(4/4)

4、Notification 通知栏信息

- Notification 通知栏会在屏幕上方向用户提示信息
- Notification 的好处：不会影响用户正在进行的操作

