|  |
| --- |
| 1、Service(服务)是一个一种可以在后台执行长时间运行操作而没有用户界面的应用组件。  Service则是android系统中的组件，它运行于主线程中，因此在Service中是不可以执行耗时操作的，否则系统会报ANR异常。如果需要让Service执行耗时任务，可在Service中开启单独线程去执行。  几个生命周期回调函数是执行在主线程中。  2、启动状态通过startService()启动服务，无限期运行；通常是单一操作，不返回结果。  服务可由其他应用组件启动（如Activity），服务一旦被启动将在后台一直运行，即使启动服务的组件（Activity）已销毁也不受影响。  要自己做停止服务的操作，stopSelf() 或 stopService()。  虽然不返回结果，还是可以使用广播、eventbus来交互。  有些程序为保证服务不被杀死，就在onDestroy()发送广播，两个服务相互监听、启动。  3、绑定状态调用bindService()绑定服务；允许组件与服务进行交互、发送请求、获取结果。也可跨进程IPC执行这些操作。组件与服务解绑后，服务被销毁。  4、自定义类继承Service基类自定义而来； 在Manifest.xml清单文件中声明。  接着重写onCreate、onStartCommand、onDestroy三个主要的生命周期方法。  （1）onBind(); 当另一个组件想通过调用 bindService() 与服务绑定（例如执行RPC）时，系统将调用此方法。在此方法的实现中，必须返回 一个IBinder 接口的实现类，供客户端用来与服务进行通信。  无论是启动状态还是绑定状态，此方法必须重写，但在启动状态的情况下可以直接返回 null。  (2)onCreate();首次创建服务时，系统将调用此方法来执行一次性设置程序（在调用 onStartCommand() 或onBind() 之前。如果服务已在运行，则不会调用此方法，该方法只调用一次。   1. onStartCommand();当另一个组件（如 Activity）通过调用 startService() 请求启动服务时，系统将调用此方法。一旦执行此方法，服务即会启动并可在后台无限期运行。   如果自己实现此方法，则需要在服务工作完成后，通过调用 stopSelf() 或 stopService() 来停止服务。  （在绑定状态下，无需实现此方法。）  每次通过startService()方法启动Service时都会被回调。  (4)onDestroy();当服务不再使用且将被销毁时，系统将调用此方法。服务应该实现此方法来清理所有资源，如线程、注册的侦听器、接收器等，这是服务接收的最后一个调用。  5、生命周期：第一次调用startService方法时，onCreate方法、onStartCommand方法将依次被调用，而多次调用startService时，只有onStartCommand方法被调用，最后我们调用stopService方法停止服务时onDestory方法被回调，这就是启动状态下Service的执行周期。  6、绑定服务是Service的另一种变形，当Service处于绑定状态时，其代表着客户端-服务器接口中的服务器。当其他组件（如Activity）绑定到服务时，组件（如Activity）可以向Service（也就是服务端）发送请求，或者调用Service（服务端）的方法，此时被绑定的Service（服务端）会接收信息并响应，甚至可以通过绑定服务进行执行进程间通信(即IPC，这个后面再单独分析)。  与启动服务不同的是绑定服务的生命周期通常只在为其他应用组件(如Activity)服务时处于活动状态，不会无限期在后台运行，也就是说宿主(如Activity)解除绑定后，绑定服务就会被销毁。  实际上我们必须提供一个IBinder接口的实现类，该类用以提供客户端用来与服务进行交互的编程接口，该接口可以通过三种方法定义接口：扩展 Binder 类、使用 Messenger、使用 AIDL。  bindService(Intent service, ServiceConnection conn, int flags)  该方法执行绑定服务操作，其中Intent是我们要绑定的服务(也就是LocalService)的意图，而ServiceConnection代表与服务的连接。  unbindService(ServiceConnection conn)  该方法执行解除绑定的操作。  扩展：AIDL、Messenger的使用 |

Service全部内容基本会在本篇涉及到，我们将围绕以下主要知识点进行分析：

Service简单概述

Service在清单文件中的声明

Service启动服务实现方式及其详解

Service绑定服务的三种实现方式

关于启动服务与绑定服务间的转换问题

前台服务以及通知发送

服务Service与线程Thread的区别

管理服务生命周期的要点

Android 5.0以上的隐式启动问题及其解决方案

保证服务不被杀死的实现思路

## Service简单概述

（在后台执行，没有用户界面。比如：网络事件、播放音乐、IO等等）

  Service(服务)是一个一种可以在后台执行长时间运行操作而没有用户界面的应用组件。服务可由其他应用组件启动（如Activity），服务一旦被启动将在后台一直运行，即使启动服务的组件（Activity）已销毁也不受影响。 此外，组件可以绑定到服务，以与之进行交互，甚至是执行进程间通信 (IPC)。 例如，服务可以处理网络事务、播放音乐，执行文件 I/O 或与内容提供程序交互，而所有这一切均可在后台进行，Service基本上分为两种形式：

启动状态

（通过startService()启动服务，无限期运行；通常是单一操作，不返回结果。）

（一般什么场景使用呢？）

  当应用组件（如 Activity）通过调用 startService() 启动服务时，服务即处于“启动”状态。一旦启动，服务即可在后台无限期运行，即使启动服务的组件已被销毁也不受影响，除非手动调用才能停止服务， 已启动的服务通常是执行单一操作，而且不会将结果返回给调用方。

绑定状态

（调用bindService()绑定服务；允许组件与服务进行交互、发送请求、获取结果。也可跨进程IPC执行这些操作。组件与服务解绑后，服务被销毁。）

（一般是什么场景使用？）

  当应用组件通过调用 bindService() 绑定到服务时，服务即处于“绑定”状态。绑定服务提供了一个客户端-服务器接口，允许组件与服务进行交互、发送请求、获取结果，甚至是利用进程间通信 (IPC) 跨进程执行这些操作。 仅当与另一个应用组件绑定时，绑定服务才会运行。 多个组件可以同时绑定到该服务，但全部取消绑定后，该服务即会被销毁。

## Service在清单文件中的声明

（继承Service基类自定义而来； 在Manifest.xml清单文件中声明。）

  前面说过Service分为启动状态和绑定状态两种，但无论哪种具体的Service启动类型，都是通过继承Service基类自定义而来，也都需要在AndroidManifest.xml中声明，那么在分析这两种状态之前，我们先来了解一下Service在AndroidManifest.xml中的声明语法，其格式如下：

<service android:enabled=["true" | "false"]

android:exported=["true" | "false"]

android:icon="drawable resource"

android:isolatedProcess=["true" | "false"]

android:label="string resource"

android:name="string"

android:permission="string"

android:process="string" >

. . .

</service>

android:exported：代表是否能被其他应用隐式调用，其默认值是由service中有无intent-filter决定的，如果有intent-filter，默认值为true，否则为false。为false的情况下，即使有intent-filter匹配，也无法打开，即无法被其他应用隐式调用。

android:name：对应Service类名。

android:permission：是权限声明。

android:process：是否需要在单独的进程中运行,当设置为android:process=”:remote”时，代表Service在单独的进程中运行。注意“：”很重要，它的意思是指要在当前进程名称前面附加上当前的包名，所以“remote”和”:remote”不是同一个意思，前者的进程名称为：remote，而后者的进程名称为：App-packageName:remote。

android:isolatedProcess ：设置 true 意味着，服务会在一个特殊的进程下运行，这个进程与系统其他进程分开且没有自己的权限。与其通信的唯一途径是通过服务的API(bind and start)。

android:enabled：是否可以被系统实例化，默认为 true因为父标签 也有 enable 属性，所以必须两个都为默认值 true 的情况下服务才会被激活，否则不会激活。

  ok~，关于Service在清单文件的声明我们先了解这些就行，接下来分别针对Service启动服务和绑定服务进行详细分析。

## 3.Service启动服务

首先要创建服务，必须创建 Service 的子类（或使用它的一个现有子类如IntentService）。在实现中，我们需要重写一些回调方法，以处理服务生命周期的某些关键过程，下面我们通过简单案例来分析需要重写的回调方法有哪些？

package com.zejian.ipctest.service;

import android.app.Service;

import android.content.Intent;

import android.os.IBinder;

import android.support.annotation.Nullable;

/\*\*

\* Created by zejian

\* Time 2016/9/29.

\* Description:service simple demo

\*/

public class SimpleService extends Service {

/\*\*

\* 绑定服务时才会调用

\* 必须要实现的方法

\* @param intent

\* @return

\*/

@Nullable

@Override

public IBinder onBind(Intent intent) {

return null;

}

/\*\*

\* 首次创建服务时，系统将调用此方法来执行一次性设置程序（在调用 onStartCommand() 或 onBind() 之前）。

\* 如果服务已在运行，则不会调用此方法。该方法只被调用一次

\*/

@Override

public void onCreate() {

System.out.println("onCreate invoke");

super.onCreate();

}

/\*\*

\* 每次通过startService()方法启动Service时都会被回调。

\* @param intent

\* @param flags

\* @param startId

\* @return

\*/

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

System.out.println("onStartCommand invoke");

return super.onStartCommand(intent, flags, startId);

}

/\*\*

\* 服务销毁时的回调

\*/

@Override

public void onDestroy() {

System.out.println("onDestroy invoke");

super.onDestroy();

}

}

  从上面的代码我们可以看出SimpleService继承了Service类，并重写了onBind方法，该方法是必须重写的，但是由于此时是启动状态的服务，则该方法无须实现，返回null即可，只有在绑定状态的情况下才需要实现该方法并返回一个IBinder的实现类（这个后面会详细说），接着重写了onCreate、onStartCommand、onDestroy三个主要的生命周期方法，关于这几个方法说明如下:

onBind()

  当另一个组件想通过调用 bindService() 与服务绑定（例如执行 RPC）时，系统将调用此方法。在此方法的实现中，必须返回 一个IBinder 接口的实现类，供客户端用来与服务进行通信。无论是启动状态还是绑定状态，此方法必须重写，但在启动状态的情况下直接返回 null。

onCreate()

  首次创建服务时，系统将调用此方法来执行一次性设置程序（在调用 onStartCommand() 或onBind() 之前）。如果服务已在运行，则不会调用此方法，该方法只调用一次

onStartCommand()

  当另一个组件（如 Activity）通过调用 startService() 请求启动服务时，系统将调用此方法。一旦执行此方法，服务即会启动并可在后台无限期运行。 如果自己实现此方法，则需要在服务工作完成后，通过调用 stopSelf() 或 stopService() 来停止服务。（在绑定状态下，无需实现此方法。）

onDestroy()

  当服务不再使用且将被销毁时，系统将调用此方法。服务应该实现此方法来清理所有资源，如线程、注册的侦听器、接收器等，这是服务接收的最后一个调用。

  我们通过Demo测试一下Service启动状态方法的调用顺序，MainActivity代码如下：

package com.zejian.ipctest;

import android.content.Intent;

import android.os.Bundle;

import android.support.v7.app.AppCompatActivity;

import android.view.View;

import android.widget.Button;

import com.zejian.ipctest.service.SimpleService;

public class MainActivity extends AppCompatActivity implements View.OnClickListener {

private Button startBtn;

private Button stopBtn;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

startBtn= (Button) findViewById(R.id.startService);

stopBtn= (Button) findViewById(R.id.stopService);

startBtn.setOnClickListener(this);

assert stopBtn != null;

stopBtn.setOnClickListener(this);

}

@Override

public void onClick(View v) {

Intent it=new Intent(this, SimpleService.class);

switch (v.getId()){

case R.id.startService:

startService(it);

break;

case R.id.stopService:

stopService(it);

break;

}

}

}

记得在清单配置文件中声明Service(声明方式跟Activity相似)：

<manifest ... >

...

<application ... >

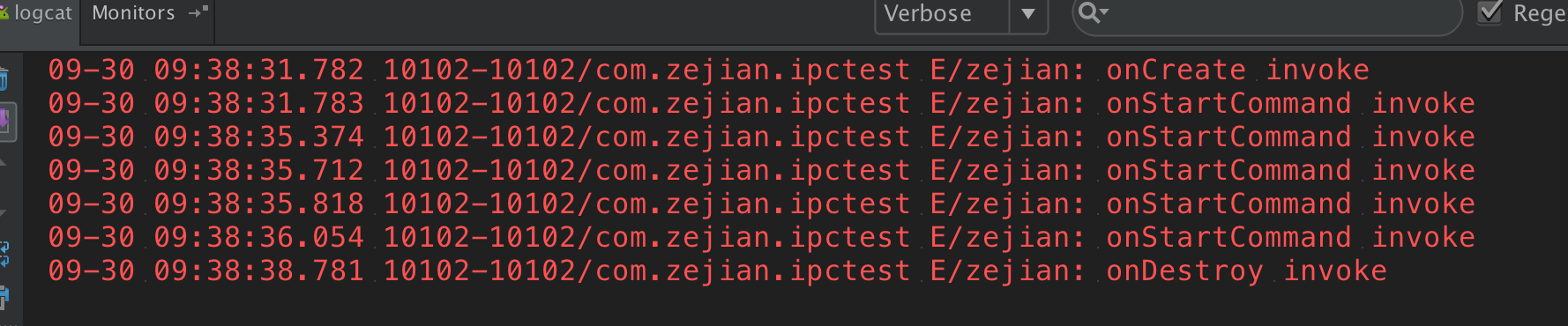
<service android:name=".service.SimpleService" />

...

</application>

</manifest>

  从代码看出，启动服务使用startService(Intent intent)方法，仅需要传递一个Intent对象即可，在Intent对象中指定需要启动的服务。而使用startService()方法启动的服务，在服务的外部，必须使用stopService()方法停止，在服务的内部可以调用stopSelf()方法停止当前服务。如果使用stopService()或者stopSelf()方法请求停止服务，系统会就会尽快销毁这个服务。值得注意的是对于启动服务，一旦启动将与访问它的组件无任何关联，即使访问它的组件被销毁了，这个服务也一直运行下去，直到手动调用停止服务才被销毁，至于onBind方法，只有在绑定服务时才会起作用，在启动状态下，无需关注此方法，ok~，我们运行程序并多次调用startService方法，最后调用stopService方法。Log截图如下：



  从Log可以看出，第一次调用startService方法时，onCreate方法、onStartCommand方法将依次被调用，而多次调用startService时，只有onStartCommand方法被调用，最后我们调用stopService方法停止服务时onDestory方法被回调，这就是启动状态下Service的执行周期。接着我们重新回过头来进一步分析onStartCommand（Intent intent, int flags, int startId），这个方法有3个传入参数，它们的含义如下：

onStartCommand（Intent intent, int flags, int startId）详细分解：

intent ：启动时，启动组件传递过来的Intent，如Activity可利用Intent封装所需要的参数并传递给Service

flags：表示启动请求时是否有额外数据，可选值有 0，START\_FLAG\_REDELIVERY，START\_FLAG\_RETRY，0代表没有，它们具体含义如下：

（1）0，代表没有。

1. START\_FLAG\_REDELIVERY，重新发还。 ----> START\_REDELIVER\_INTENT
2. START\_FLAG\_RETRY，重新调用方法。

START\_FLAG\_RETRY

START\_FLAG\_REDELIVERY

这个值代表了onStartCommand方法的返回值为

START\_REDELIVER\_INTENT，而且在上一次服务被杀死前会去调用stopSelf方法停止服务。其中START\_REDELIVER\_INTENT意味着当Service因内存不足而被系统kill后，则会重建服务，并通过传递给服务的最后一个 Intent 调用 onStartCommand()，此时Intent时有值的。

START\_FLAG\_RETRY

该flag代表当onStartCommand调用后一直没有返回值时，会尝试重新去调用onStartCommand()。

startId ： 指明当前服务的唯一ID，与stopSelfResult (int startId)配合使用，stopSelfResult 可以更安全地根据ID停止服务。

  实际上onStartCommand的返回值int类型才是最最值得注意的，它有三种可选值， START\_STICKY，START\_NOT\_STICKY，START\_REDELIVER\_INTENT，它们具体含义如下：

START\_STICKY

  当Service因内存不足而被系统kill后，一段时间后内存再次空闲时，系统将会尝试重新创建此Service，一旦创建成功后将回调onStartCommand方法，但其中的Intent将是null，除非有挂起的Intent，如pendingintent，这个状态下比较适用于不执行命令、但无限期运行并等待作业的媒体播放器或类似服务。

START\_NOT\_STICKY

  当Service因内存不足而被系统kill后，即使系统内存再次空闲时，系统也不会尝试重新创建此Service。除非程序中再次调用startService启动此Service，这是最安全的选项，可以避免在不必要时以及应用能够轻松重启所有未完成的作业时运行服务。

START\_REDELIVER\_INTENT

  当Service因内存不足而被系统kill后，则会重建服务，并通过传递给服务的最后一个 Intent 调用 onStartCommand()，任何挂起 Intent均依次传递。与START\_STICKY不同的是，其中的传递的Intent将是非空，是最后一次调用startService中的intent。这个值适用于主动执行应该立即恢复的作业（例如下载文件）的服务。

  由于每次启动服务（调用startService）时，onStartCommand方法都会被调用，因此我们可以通过该方法使用Intent给Service传递所需要的参数，然后在onStartCommand方法中处理的事件，最后根据需求选择不同的Flag返回值，以达到对程序更友好的控制。好~，以上便是Service在启动状态下的分析，接着我们在来看看绑定状态的Service又是如何处理的？

## 4.Service绑定服务

  绑定服务是Service的另一种变形，当Service处于绑定状态时，其代表着客户端-服务器接口中的服务器。当其他组件（如 Activity）绑定到服务时（有时我们可能需要从Activity组件中去调用Service中的方法，此时Activity以绑定的方式挂靠到Service后，我们就可以轻松地方法到Service中的指定方法），组件（如Activity）可以向Service（也就是服务端）发送请求，或者调用Service（服务端）的方法，此时被绑定的Service（服务端）会接收信息并响应，甚至可以通过绑定服务进行执行进程间通信 (即IPC，这个后面再单独分析)。与启动服务不同的是绑定服务的生命周期通常只在为其他应用组件(如Activity)服务时处于活动状态，不会无限期在后台运行，也就是说宿主(如Activity)解除绑定后，绑定服务就会被销毁。那么在提供绑定的服务时，该如何实现呢？实际上我们必须提供一个 IBinder接口的实现类，该类用以提供客户端用来与服务进行交互的编程接口，该接口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有应用专用的，并且Service(服务端)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 实现中以及Service 中可用的公共方法。如果我们的服务只是自有应用的后台工作线程，则优先采用这种方法。 不采用该方式创建接口的唯一原因是，服务被其他应用或不同的进程调用。

使用 Messenger

  Messenger可以翻译为信使，通过它可以在不同的进程中共传递Message对象(Handler中的Messager，因此 Handler 是 Messenger 的基础)，在Message中可以存放我们需要传递的数据，然后在进程间传递。如果需要让接口跨不同的进程工作，则可使用 Messenger 为服务创建接口，客户端就可利用 Message 对象向服务发送命令。同时客户端也可定义自有 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，也就是说Messenger是以串行的方式处理客户端发来的消息，这样我们就不必对服务进行线程安全设计了。

使用 AIDL

   由于Messenger是以串行的方式处理客户端发来的消息，如果当前有大量消息同时发送到Service(服务端)，Service仍然只能一个个处理，这也就是Messenger跨进程通信的缺点了，因此如果有大量并发请求，Messenger就会显得力不从心了，这时AIDL（Android 接口定义语言）就派上用场了， 但实际上Messenger 的跨进程方式其底层实现 就是AIDL，只不过android系统帮我们封装成透明的Messenger罢了 。因此，如果我们想让服务同时处理多个请求，则应该使用 AIDL。 在此情况下，服务必须具备多线程处理能力，并采用线程安全式设计。使用AIDL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，随后可在服务内对其进行扩展。

  以上3种实现方式，我们可以根据需求自由的选择，但需要注意的是大多数应用“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，本篇中也不打算阐述如何使用AIDL（后面会另开一篇分析AIDL），接下来我们分别针对扩展 Binder 类和Messenger的使用进行分析。

### 4.1 扩展 Binder 类

  前面描述过，如果我们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，让客户端通过该类直接访问服务中的公共方法。其使用开发步骤如下

1.创建BindService服务端，继承自Service并在类中，创建一个实现IBinder 接口的实例对象并提供公共方法给客户端调用

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：此方式只有在客户端和服务位于同一应用和进程内才有效，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，此方式非常有效。另一点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，因为此方式不执行任何跨进程编组。

  以下是一个扩展 Binder 类的实例，先看看Service端的实现BindService.java

package com.zejian.ipctest.service;

import android.app.Service;

import android.content.Intent;

import android.os.Binder;

import android.os.IBinder;

import android.support.annotation.Nullable;

import android.util.Log;

/\*\*

\* Created by zejian

\* Time 2016/10/2.

\* Description:绑定服务简单实例--服务端

\*/

public class LocalService extends Service{

private final static String TAG = "wzj";

private int count;

private boolean quit;

private Thread thread;

private LocalBinder binder = new LocalBinder();

/\*\*

\* 创建Binder对象，返回给客户端即Activity使用，提供数据交换的接口

\*/

public class LocalBinder extends Binder {

// 声明一个方法，getService。（提供给客户端调用）

LocalService getService() {

// 返回当前对象LocalService,这样我们就可在客户端端调用Service的公共方法了

return LocalService.this;

}

}

/\*\*

\* 把Binder类返回给客户端

\*/

@Nullable

@Override

public IBinder onBind(Intent intent) {

return binder;

}

@Override

public void onCreate() {

super.onCreate();

Log.i(TAG, "Service is invoke Created");

thread = new Thread(new Runnable() {

@Override

public void run() {

// 每间隔一秒count加1 ，直到quit为true。

while (!quit) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

count++;

}

}

});

thread.start();

}

/\*\*

\* 公共方法

\* @return

\*/

public int getCount(){

return count;

}

/\*\*

\* 解除绑定时调用

\* @return

\*/

@Override

public boolean onUnbind(Intent intent) {

Log.i(TAG, "Service is invoke onUnbind");

return super.onUnbind(intent);

}

@Override

public void onDestroy() {

Log.i(TAG, "Service is invoke Destroyed");

this.quit = true;

super.onDestroy();

}

}

  BindService类继承自Service，在该类中创建了一个LocalBinder继承自Binder类，LocalBinder中声明了一个getService方法，客户端可访问该方法获取LocalService对象的实例，只要客户端获取到LocalService对象的实例就可调用LocalService服务端的公共方法，如getCount方法，值得注意的是，我们在onBind方法中返回了binder对象，该对象便是LocalBinder的具体实例，而binder对象最终会返回给客户端，客户端通过返回的binder对象便可以与服务端实现交互。接着看看客户端BindActivity的实现：

package com.zejian.ipctest.service;

import android.app.Activity;

import android.app.Service;

import android.content.ComponentName;

import android.content.Intent;

import android.content.ServiceConnection;

import android.os.Bundle;

import android.os.IBinder;

import android.util.Log;

import android.view.View;

import android.widget.Button;

import com.zejian.ipctest.R;

/\*\*

\* Created by zejian

\* Time 2016/10/2.

\* Description:绑定服务实例--客户端

\*/

public class BindActivity extends Activity {

protected static final String TAG = "wzj";

Button btnBind;

Button btnUnBind;

Button btnGetDatas;

/\*\*

\* ServiceConnection代表与服务的连接，它只有两个方法，

\* onServiceConnected和onServiceDisconnected，

\* 前者是在操作者在连接一个服务成功时被调用，而后者是在服务崩溃或被杀死导致的连接中断时被调用

\*/

private ServiceConnection conn;

private LocalService mService;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_bind);

btnBind = (Button) findViewById(R.id.BindService);

btnUnBind = (Button) findViewById(R.id.unBindService);

btnGetDatas = (Button) findViewById(R.id.getServiceDatas);

//创建绑定对象

final Intent intent = new Intent(this, LocalService.class);

// 开启绑定

btnBind.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Log.d(TAG, "绑定调用：bindService");

//调用绑定方法

bindService(intent, conn, Service.BIND\_AUTO\_CREATE);

}

});

// 解除绑定

btnUnBind.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Log.d(TAG, "解除绑定调用：unbindService");

// 解除绑定

if(mService!=null) {

mService = null;

unbindService(conn);

}

}

});

// 获取数据

btnGetDatas.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

if (mService != null) {

// 通过绑定服务传递的Binder对象，获取Service暴露出来的数据

Log.d(TAG, "从服务端获取数据：" + mService.getCount());

} else {

Log.d(TAG, "还没绑定呢，先绑定,无法从服务端获取数据");

}

}

});

conn = new ServiceConnection() {

/\*\*

\* 与服务器端交互的接口方法 绑定服务的时候被回调，在这个方法获取绑定Service传递过来的IBinder对象，

\* 通过这个IBinder对象，实现宿主和Service的交互。

\*/

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

Log.d(TAG, "绑定成功调用：onServiceConnected");

// 获取Binder

LocalService.LocalBinder binder = (LocalService.LocalBinder) service;

mService = binder.getService();

}

/\*\*

\* 当取消绑定的时候被回调。但正常情况下是不被调用的，它的调用时机是当Service服务被意外销毁时，

\* 例如内存的资源不足时这个方法才被自动调用。

\*/

@Override

public void onServiceDisconnected(ComponentName name) {

mService=null;

}

};

}

}

  在客户端中我们创建了一个ServiceConnection对象，该代表与服务的连接，它只有两个方法， onServiceConnected和onServiceDisconnected，其含义如下：

onServiceConnected(ComponentName name, IBinder service)

系统会调用该方法以传递服务的　onBind() 方法返回的 IBinder。其中service便是服务端返回的IBinder实现类对象，通过该对象我们便可以调用获取LocalService实例对象，进而调用服务端的公共方法。而ComponentName是一个封装了组件(Activity, Service, BroadcastReceiver, or ContentProvider)信息的类，如包名，组件描述等信息，较少使用该参数。

onServiceDisconnected(ComponentName name)

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。注意:当客户端取消绑定时，系统“绝对不会”调用该方法。

conn = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

Log.d(TAG, "绑定成功调用：onServiceConnected");

// 获取Binder

LocalService.LocalBinder binder = (LocalService.LocalBinder) service;

mService = binder.getService();

}

@Override

public void onServiceDisconnected(ComponentName name) {

mService=null;

}

};

  在onServiceConnected()被回调前，我们还需先把当前Activity绑定到服务LocalService上，绑定服务是通过通过bindService()方法，解绑服务则使用unbindService()方法，这两个方法解析如下：

bindService(Intent service, ServiceConnection conn, int flags)

该方法执行绑定服务操作，其中Intent是我们要绑定的服务(也就是LocalService)的意图，而ServiceConnection代表与服务的连接，它只有两个方法，前面已分析过，flags则是指定绑定时是否自动创建Service。0代表不自动创建、BIND\_AUTO\_CREATE则代表自动创建。

unbindService(ServiceConnection conn)

该方法执行解除绑定的操作，其中ServiceConnection代表与服务的连接，它只有两个方法，前面已分析过。

Activity通过bindService()绑定到LocalService后，ServiceConnection#onServiceConnected()便会被回调并可以获取到LocalService实例对象mService，之后我们就可以调用LocalService服务端的公共方法了，最后还需要在清单文件中声明该Service。而客户端布局文件实现如下：

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical" android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<Button

android:id="@+id/BindService"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="绑定服务器"

/>

<Button

android:id="@+id/unBindService"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="解除绑定"

/>

<Button

android:id="@+id/getServiceDatas"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="获取服务方数据"

/>

</LinearLayout>

  我们运行程序，点击绑定服务并多次点击绑定服务接着多次调用LocalService中的getCount()获取数据，最后调用解除绑定的方法移除服务，其结果如下：



  通过Log可知，当我们第一次点击绑定服务时，LocalService服务端的onCreate()、onBind方法会依次被调用，此时客户端的ServiceConnection#onServiceConnected()被调用并返回LocalBinder对象，接着调用LocalBinder#getService方法返回LocalService实例对象，此时客户端便持有了LocalService的实例对象，也就可以任意调用LocalService类中的声明公共方法了。更值得注意的是，我们多次调用bindService方法绑定LocalService服务端，而LocalService得onBind方法只调用了一次，那就是在第一次调用bindService时才会回调onBind方法。接着我们点击获取服务端的数据，从Log中看出我们点击了3次通过getCount()获取了服务端的3个不同数据，最后点击解除绑定，此时LocalService的onUnBind、onDestroy方法依次被回调，并且多次绑定只需一次解绑即可。此情景也就说明了绑定状态下的Service生命周期方法的调用依次为onCreate()、onBind、onUnBind、onDestroy。ok~，以上便是同一应用同一进程中客户端与服务端的绑定回调方式。

### 4.2 使用Messenger（重看）

  前面了解了如何使用IBinder应用内同一进程的通信后，我们接着来了解服务与远程进程（即不同进程间）通信，而不同进程间的通信，最简单的方式就是使用 Messenger 服务提供通信接口，利用此方式，我们无需使用 AIDL 便可执行进程间通信 (IPC)。以下是 Messenger 使用的主要步骤：

1.服务实现一个 Handler，由其接收来自客户端的每个调用的回调

2.Handler 用于创建 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（引用服务的 Handler）实例化，然后使用Messenger将 Message 对象发送给服务

5.服务在其 Handler 中（在 handleMessage() 方法中）接收每个 Message

以下是一个使用 Messenger 接口的简单服务示例，服务端进程实现如下：

package com.zejian.ipctest.messenger;

import android.app.Service;

import android.content.Intent;

import android.os.Handler;

import android.os.IBinder;

import android.os.Message;

import android.os.Messenger;

import android.util.Log;

/\*\*

\* Created by zejian

\* Time 2016/10/3.

\* Description:Messenger服务端简单实例,服务端进程

\*/

public class MessengerService extends Service {

/\*\* Command to the service to display a message \*/

static final int MSG\_SAY\_HELLO = 1;

private static final String TAG ="wzj" ;

/\*\*

\* 用于接收从客户端传递过来的数据

\*/

class IncomingHandler extends Handler {

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

case MSG\_SAY\_HELLO:

Log.i(TAG, "thanks,Service had receiver message from client!");

break;

default:

super.handleMessage(msg);

}

}

}

/\*\*

\* 创建Messenger并传入Handler实例对象

\*/

final Messenger mMessenger = new Messenger(new IncomingHandler());

/\*\*

\* 当绑定Service时,该方法被调用,将通过mMessenger返回一个实现

\* IBinder接口的实例对象

\*/

@Override

public IBinder onBind(Intent intent) {

Log.i(TAG, "Service is invoke onBind");

return mMessenger.getBinder();

}

}

  首先我们同样需要创建一个服务类MessengerService继承自Service，同时创建一个继承自Handler的IncomingHandler对象来接收客户端进程发送过来的消息并通过其handleMessage(Message msg)进行消息处理。接着通过IncomingHandler对象创建一个Messenger对象，该对象是与客户端交互的特殊对象，然后在Service的onBind中返回这个Messenger对象的底层Binder即可。下面看看客户端进程的实现：

package com.zejian.ipctest.messenger;

import android.app.Activity;

import android.content.ComponentName;

import android.content.Context;

import android.content.Intent;

import android.content.ServiceConnection;

import android.os.Bundle;

import android.os.IBinder;

import android.os.Message;

import android.os.Messenger;

import android.os.RemoteException;

import android.util.Log;

import android.view.View;

import android.widget.Button;

import com.zejian.ipctest.R;

/\*\*

\* Created by zejian

\* Time 2016/10/3.

\* Description: 与服务器交互的客户端

\*/

public class ActivityMessenger extends Activity {

/\*\*

\* 与服务端交互的Messenger

\*/

Messenger mService = null;

/\*\* Flag indicating whether we have called bind on the service. \*/

boolean mBound;

/\*\*

\* 实现与服务端链接的对象

\*/

private ServiceConnection mConnection = new ServiceConnection() {

public void onServiceConnected(ComponentName className, IBinder service) {

/\*\*

\* 通过服务端传递的IBinder对象,创建相应的Messenger

\* 通过该Messenger对象与服务端进行交互

\*/

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;

// 创建与服务交互的消息实体Message

Message msg = Message.obtain(null, MessengerService.MSG\_SAY\_HELLO, 0, 0);

try {

//发送消息

mService.send(msg);

} catch (RemoteException e) {

e.printStackTrace();

}

}

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_messenager);

Button bindService= (Button) findViewById(R.id.bindService);

Button unbindService= (Button) findViewById(R.id.unbindService);

Button sendMsg= (Button) findViewById(R.id.sendMsgToService);

bindService.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Log.d("zj","onClick-->bindService");

//当前Activity绑定服务端

bindService(new Intent(ActivityMessenger.this, MessengerService.class), mConnection,

Context.BIND\_AUTO\_CREATE);

}

});

//发送消息给服务端

sendMsg.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

sayHello(v);

}

});

unbindService.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

// Unbind from the service

if (mBound) {

Log.d("zj","onClick-->unbindService");

unbindService(mConnection);

mBound = false;

}

}

});

}

}

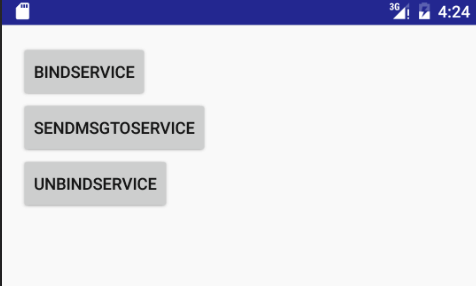
  在客户端进程中，我们需要创建一个ServiceConnection对象，该对象代表与服务端的链接，当调用bindService方法将当前Activity绑定到MessengerService时，onServiceConnected方法被调用，利用服务端传递给来的底层Binder对象构造出与服务端交互的Messenger对象，接着创建与服务交互的消息实体Message，将要发生的信息封装在Message中并通过Messenger实例对象发送给服务端。关于ServiceConnection、bindService方法、unbindService方法，前面已分析过，这里就不重复了，最后我们需要在清单文件声明Service和Activity，由于要测试不同进程的交互，则需要将Service放在单独的进程中，因此Service声明如下：

<service android:name=".messenger.MessengerService"

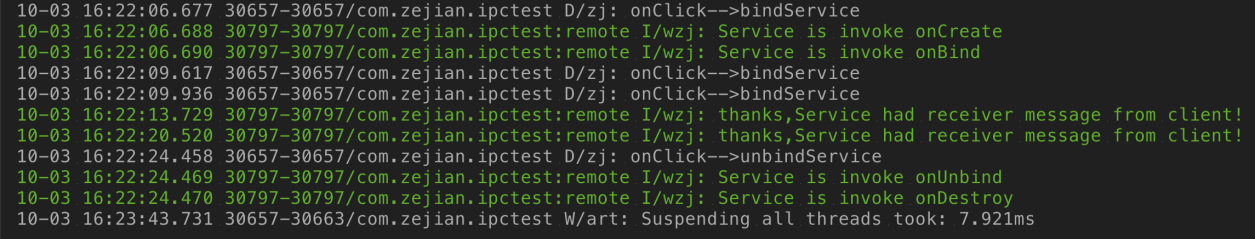
android:process=":remote"

/>

其中android:process=":remote"代表该Service在单独的进程中创建，最后我们运行程序，结果如下：



  接着多次点击绑定服务，然后发送信息给服务端，最后解除绑定，Log打印如下：



  通过上述例子可知Service服务端确实收到了客户端发送的信息，而且在Messenger中进行数据传递必须将数据封装到Message中，因为Message和Messenger都实现了Parcelable接口，可以轻松跨进程传递数据（关于Parcelable接口可以看博主的另一篇文章：序列化与反序列化之Parcelable和Serializable浅析），而Message可以传递的信息载体有，what,arg1,arg2,Bundle以及replyTo，至于object字段，对于同一进程中的数据传递确实很实用，但对于进程间的通信，则显得相当尴尬，在android2.2前，object不支持跨进程传输，但即便是android2.2之后也只能传递android系统提供的实现了Parcelable接口的对象，也就是说我们通过自定义实现Parcelable接口的对象无法通过object字段来传递，因此object字段的实用性在跨进程中也变得相当低了。不过所幸我们还有Bundle对象，Bundle可以支持大量的数据类型。接着从Log我们也看出无论是使用拓展Binder类的实现方式还是使用Messenger的实现方式，它们的生命周期方法的调用顺序基本是一样的，即onCreate()、onBind、onUnBind、onDestroy，而且多次绑定中也只有第一次时才调用onBind()。好~，以上的例子演示了如何在服务端解释客户端发送的消息，但有时候我们可能还需要服务端能回应客户端，这时便需要提供双向消息传递了，下面就来实现一个简单服务端与客户端双向消息传递的简单例子。

  先来看看服务端的修改，在服务端，我们只需修改IncomingHandler，收到消息后，给客户端回复一条信息。

/\*\*

\* 用于接收从客户端传递过来的数据

\*/

class IncomingHandler extends Handler {

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

case MSG\_SAY\_HELLO:

Log.i(TAG, "thanks,Service had receiver message from client!");

//回复客户端信息,该对象由客户端传递过来

Messenger client=msg.replyTo;

//获取回复信息的消息实体

Message replyMsg=Message.obtain(null,MessengerService.MSG\_SAY\_HELLO);

Bundle bundle=new Bundle();

bundle.putString("reply","ok~,I had receiver message from you! ");

replyMsg.setData(bundle);

//向客户端发送消息

try {

client.send(replyMsg);

} catch (RemoteException e) {

e.printStackTrace();

}

break;

default:

super.handleMessage(msg);

}

}

}

  接着修改客户端，为了接收服务端的回复，客户端也需要一个接收消息的Messenger和Handler，其实现如下：

/\*\*

\* 用于接收服务器返回的信息

\*/

private Messenger mRecevierReplyMsg= new Messenger(new ReceiverReplyMsgHandler());

private static class ReceiverReplyMsgHandler extends Handler{

private static final String TAG = "zj";

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

//接收服务端回复

case MessengerService.MSG\_SAY\_HELLO:

Log.i(TAG, "receiver message from

service:"+msg.getData().getString("reply"));

break;

default:

super.handleMessage(msg);

}

}

}

  除了添加以上代码，还需要在发送信息时把接收服务器端的回复的Messenger通过Message的replyTo参数传递给服务端，以便作为同学桥梁，代码如下：

public void sayHello(View v) {

if (!mBound) return;

// 创建与服务交互的消息实体Message

Message msg = Message.obtain(null, MessengerService.MSG\_SAY\_HELLO, 0, 0);

//把接收服务器端的回复的Messenger通过Message的replyTo参数传递给服务端

msg.replyTo=mRecevierReplyMsg;

try {

//发送消息

mService.send(msg);

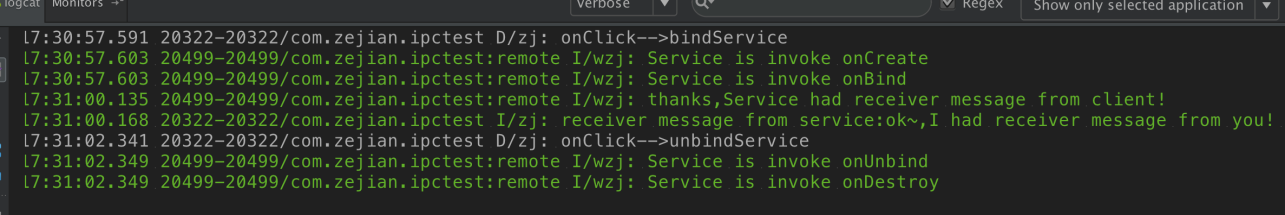
} catch (RemoteException e) {

e.printStackTrace();

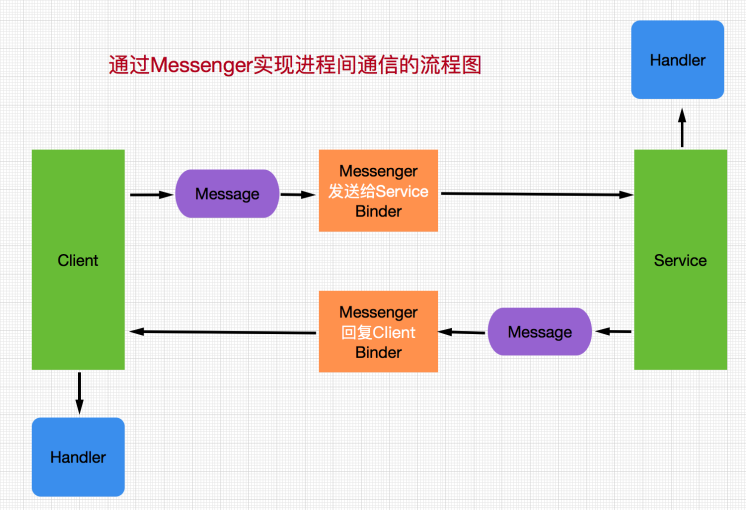
}

}

  ok~，到此服务端与客户端双向消息传递的简单例子修改完成，我们运行一下代码，看看Log打印，如下：



  由Log可知，服务端和客户端确实各自收到了信息，到此我们就把采用Messenge进行跨进程通信的方式分析完了，最后为了辅助大家理解，这里提供一张通过Messenge方式进行进程间通信的原理图：



### 4.3 关于绑定服务的注意点

  1.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）。

  2.通常情况下我们应该在客户端生命周期（如Activity的生命周期）的引入 (bring-up) 和退出 (tear-down) 时刻设置绑定和取消绑定操作，以便控制绑定状态下的Service，一般有以下两种情况：

如果只需要在 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。需要注意的是，这意味着 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，那么当提高该进程的权重时，系统很可能会终止该进程。

  3.通常情况下(注意)，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，这样反复绑定与解绑是不合理的。此外，如果应用内的多个 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 在下一次绑定（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，因此服务的绑定不应该发生在 Activity 的 onResume() 和 onPause()中。

  4.我们应该始终捕获 DeadObjectException DeadObjectException 异常，该异常是在连接中断时引发的，表示调用的对象已死亡，也就是Service对象已销毁，这是远程方法引发的唯一异常，DeadObjectException继承自RemoteException，因此我们也可以捕获RemoteException异常。

  5.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，而该绑定是异步执行的。

## 5.关于启动服务与绑定服务间的转换问题

  通过前面对两种服务状态的分析，相信大家已对Service的两种状态有了比较清晰的了解，那么现在我们就来分析一下当启动状态和绑定状态同时存在时，又会是怎么的场景？

  虽然服务的状态有启动和绑定两种，但实际上一个服务可以同时是这两种状态，也就是说，它既可以是启动服务（以无限期运行），也可以是绑定服务。有点需要注意的是Android系统仅会为一个Service创建一个实例对象，所以不管是启动服务还是绑定服务，操作的是同一个Service实例，而且由于绑定服务或者启动服务执行顺序问题将会出现以下两种情况：

先绑定服务后启动服务

  如果当前Service实例先以绑定状态运行，然后再以启动状态运行，那么绑定服务将会转为启动服务运行，这时如果之前绑定的宿主（Activity）被销毁了，也不会影响服务的运行，服务还是会一直运行下去，指定收到调用停止服务或者内存不足时才会销毁该服务。

先启动服务后绑定服务

  如果当前Service实例先以启动状态运行，然后再以绑定状态运行，当前启动服务并不会转为绑定服务，但是还是会与宿主绑定，只是即使宿主解除绑定后，服务依然按启动服务的生命周期在后台运行，直到有Context调用了stopService()或是服务本身调用了stopSelf()方法抑或内存不足时才会销毁服务。

  以上两种情况显示出启动服务的优先级确实比绑定服务高一些。不过无论Service是处于启动状态还是绑定状态，或处于启动并且绑定状态，我们都可以像使用Activity那样通过调用 Intent 来使用服务(即使此服务来自另一应用)。 当然，我们也可以通过清单文件将服务声明为私有服务，阻止其他应用访问。最后这里有点需要特殊说明一下的，由于服务在其托管进程的主线程中运行（UI线程），它既不创建自己的线程，也不在单独的进程中运行（除非另行指定）。 这意味着，如果服务将执行任何耗时事件或阻止性操作（例如 MP3 播放或联网）时，则应在服务内创建新线程来完成这项工作，简而言之，耗时操作应该另起线程执行。只有通过使用单独的线程，才可以降低发生“应用无响应”(ANR) 错误的风险，这样应用的主线程才能专注于用户与 Activity 之间的交互， 以达到更好的用户体验。

## 6.前台服务以及通知发送

  前台服务被认为是用户主动意识到的一种服务，因此在内存不足时，系统也不会考虑将其终止。 前台服务必须为状态栏提供通知，状态栏位于“正在进行”标题下方，这意味着除非服务停止或从前台删除，否则不能清除通知。例如将从服务播放音乐的音乐播放器设置为在前台运行，这是因为用户明确意识到其操作。 状态栏中的通知可能表示正在播放的歌曲，并允许用户启动 Activity 来与音乐播放器进行交互。如果需要设置服务运行于前台， 我们该如何才能实现呢？Android官方给我们提供了两个方法，分别是startForeground()和stopForeground()，这两个方式解析如下：

startForeground(int id, Notification notification)

该方法的作用是把当前服务设置为前台服务，其中id参数代表唯一标识通知的整型数，需要注意的是提供给 startForeground() 的整型 ID 不得为 0，而notification是一个状态栏的通知。

stopForeground(boolean removeNotification)

该方法是用来从前台删除服务，此方法传入一个布尔值，指示是否也删除状态栏通知，true为删除。 注意该方法并不会停止服务。 但是，如果在服务正在前台运行时将其停止，则通知也会被删除。

下面我们结合一个简单案例来使用以上两个方法，ForegroundService代码如下：

package com.zejian.ipctest.foregroundService;

import android.app.Notification;

import android.app.Service;

import android.content.Intent;

import android.graphics.BitmapFactory;

import android.os.IBinder;

import android.support.annotation.Nullable;

import android.support.v4.app.NotificationCompat;

import com.zejian.ipctest.R;

/\*\*

\* Created by zejian

\* Time 2016/10/4.

\* Description:启动前台服务Demo

\*/

public class ForegroundService extends Service {

/\*\*

\* id不可设置为0,否则不能设置为前台service

\*/

private static final int NOTIFICATION\_DOWNLOAD\_PROGRESS\_ID = 0x0001;

private boolean isRemove=false;//是否需要移除

/\*\*

\* Notification

\*/

public void createNotification(){

//使用兼容版本

NotificationCompat.Builder builder=new NotificationCompat.Builder(this);

//设置状态栏的通知图标

builder.setSmallIcon(R.mipmap.ic\_launcher);

//设置通知栏横条的图标

builder.setLargeIcon(BitmapFactory.decodeResource(getResources(),R.drawable.screenflash\_logo));

//禁止用户点击删除按钮删除

builder.setAutoCancel(false);

//禁止滑动删除

builder.setOngoing(true);

//右上角的时间显示

builder.setShowWhen(true);

//设置通知栏的标题内容

builder.setContentTitle("I am Foreground Service!!!");

//创建通知

Notification notification = builder.build();

//设置为前台服务

startForeground(NOTIFICATION\_DOWNLOAD\_PROGRESS\_ID,notification);

}

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

int i=intent.getExtras().getInt("cmd");

if(i==0){

if(!isRemove) {

createNotification();

}

isRemove=true;

}else {

//移除前台服务

if (isRemove) {

stopForeground(true);

}

isRemove=false;

}

return super.onStartCommand(intent, flags, startId);

}

@Override

public void onDestroy() {

//移除前台服务

if (isRemove) {

stopForeground(true);

}

isRemove=false;

super.onDestroy();

}

@Nullable

@Override

public IBinder onBind(Intent intent) {

return null;

}

}

  在ForegroundService类中，创建了一个notification的通知，并通过启动Service时传递过来的参数判断是启动前台服务还是关闭前台服务，最后在onDestroy方法被调用时，也应该移除前台服务。以下是ForegroundActivity的实现：

package com.zejian.ipctest.foregroundService;

import android.app.Activity;

import android.content.Intent;

import android.os.Bundle;

import android.view.View;

import android.widget.Button;

import com.zejian.ipctest.R;

/\*\*

\* Created by zejian

\* Time 2016/10/4.

\* Description:

\*/

public class ForegroundActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_foreground);

Button btnStart= (Button) findViewById(R.id.startForeground);

Button btnStop= (Button) findViewById(R.id.stopForeground);

final Intent intent = new Intent(this,ForegroundService.class);

btnStart.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

intent.putExtra("cmd",0);//0,开启前台服务,1,关闭前台服务

startService(intent);

}

});

btnStop.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

intent.putExtra("cmd",1);//0,开启前台服务,1,关闭前台服务

startService(intent);

}

});

}

}

  代码比较简单，我们直接运行程序看看结果：

  ok~，以上便是有关于Service前台服务的内容，接下来再聊聊服务与线程的区别

## 7.服务Service与线程Thread的区别

两者概念的迥异

Thread 是程序执行的最小单元，它是分配CPU的基本单位，android系统中UI线程也是线程的一种，当然Thread还可以用于执行一些耗时异步的操作。

Service是Android的一种机制，服务是运行在主线程上的，它是由系统进程托管。它与其他组件之间的通信类似于client和server，是一种轻量级的IPC通信，这种通信的载体是binder，它是在linux层交换信息的一种IPC，而所谓的Service后台任务只不过是指没有UI的组件罢了。

两者的执行任务迥异

在android系统中，线程一般指的是工作线程(即后台线程)，而主线程是一种特殊的工作线程，它负责将事件分派给相应的用户界面小工具，如绘图事件及事件响应，因此为了保证应用 UI 的响应能力主线程上不可执行耗时操作。如果执行的操作不能很快完成，则应确保它们在单独的工作线程执行。

Service 则是android系统中的组件，一般情况下它运行于主线程中，因此在Service中是不可以执行耗时操作的，否则系统会报ANR异常，之所以称Service为后台服务，大部分原因是它本身没有UI，用户无法感知(当然也可以利用某些手段让用户知道)，但如果需要让Service执行耗时任务，可在Service中开启单独线程去执行。

两者使用场景

当要执行耗时的网络或者数据库查询以及其他阻塞UI线程或密集使用CPU的任务时，都应该使用工作线程(Thread)，这样才能保证UI线程不被占用而影响用户体验。

在应用程序中，如果需要长时间的在后台运行，而且不需要交互的情况下，使用服务。比如播放音乐，通过Service+Notification方式在后台执行同时在通知栏显示着。

两者的最佳使用方式

在大部分情况下，Thread和Service都会结合着使用，比如下载文件，一般会通过Service在后台执行+Notification在通知栏显示+Thread异步下载，再如应用程序会维持一个Service来从网络中获取推送服务。在Android官方看来也是如此，所以官网提供了一个Thread与Service的结合来方便我们执行后台耗时任务，它就是IntentService，(如果想更深入了解IntentService，可以看博主的另一篇文章：Android 多线程之IntentService 完全详解)，当然 IntentService并不适用于所有的场景，但它的优点是使用方便、代码简洁，不需要我们创建Service实例并同时也创建线程，某些场景下还是非常赞的！由于IntentService是单个worker thread，所以任务需要排队，因此不适合大多数的多任务情况。

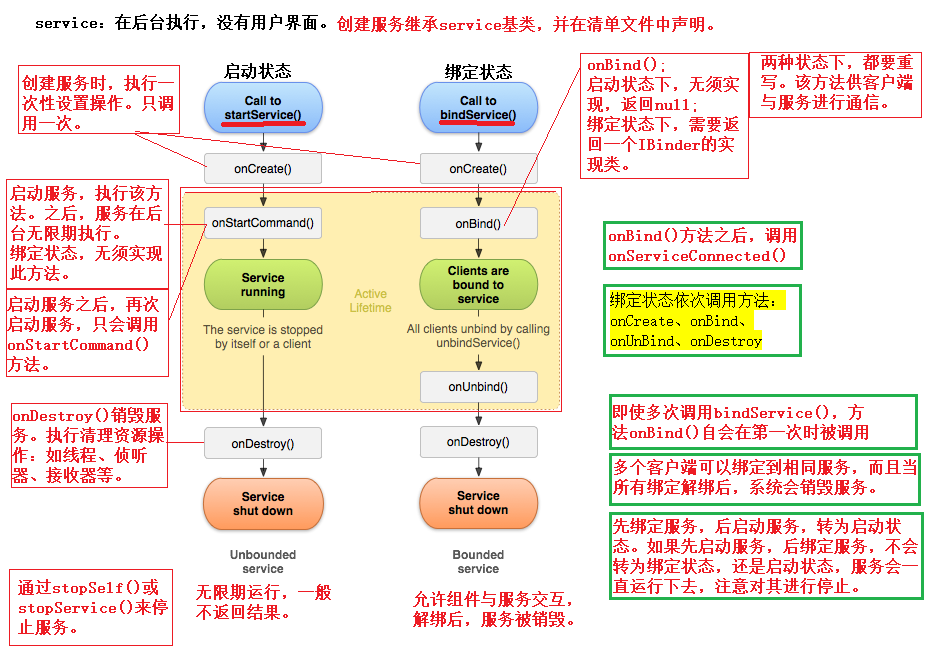
两者的真正关系

两者没有半毛钱关系。

## 8.管理服务生命周期

  关于Service生命周期方法的执行顺序，前面我们已分析得差不多了，这里重新给出一张执行的流程图（出自Android官网）





其中左图显示了使用 startService() 所创建的服务的生命周期，右图显示了使用 bindService() 所创建的服务的生命周期。通过图中的生命周期方法，我们可以监控Service的整体执行过程，包括创建，运行，销毁，关于Service不同状态下的方法回调在前面的分析中已描述得很清楚，这里就不重复了，下面给出官网对生命周期的原文描述：

  服务的整个生命周期从调用 onCreate() 开始起，到 onDestroy() 返回时结束。与 Activity 类似，服务也在 onCreate() 中完成初始设置，并在 onDestroy() 中释放所有剩余资源。例如，音乐播放服务可以在 onCreate() 中创建用于播放音乐的线程，然后在 onDestroy() 中停止该线程。

  无论服务是通过 startService() 还是 bindService() 创建，都会为所有服务调用 onCreate() 和 onDestroy() 方法。

  服务的有效生命周期从调用 onStartCommand() 或 onBind() 方法开始。每种方法均有 Intent 对象，该对象分别传递到 startService() 或 bindService()。

  对于启动服务，有效生命周期与整个生命周期同时结束（即便是在 onStartCommand() 返回之后，服务仍然处于活动状态）。对于绑定服务，有效生命周期在 onUnbind() 返回时结束。

  从执行流程图来看，服务的生命周期比 Activity 的生命周期要简单得多。但是，我们必须密切关注如何创建和销毁服务，因为服务可以在用户没有意识到的情况下运行于后台。管理服务的生命周期（从创建到销毁）有以下两种情况：

启动服务

该服务在其他组件调用 startService() 时创建，然后无限期运行，且必须通过调用 stopSelf() 来自行停止运行。此外，其他组件也可以通过调用 stopService() 来停止服务。服务停止后，系统会将其销毁。

绑定服务

该服务在另一个组件（客户端）调用 bindService() 时创建。然后，客户端通过 IBinder 接口与服务进行通信。客户端可以通过调用 unbindService() 关闭连接。多个客户端可以绑定到相同服务，而且当所有绑定全部取消后，系统即会销毁该服务。 （服务不必自行停止运行）

  虽然可以通过以上两种情况管理服务的生命周期，但是我们还必须考虑另外一种情况，也就是启动服务与绑定服务的结合体，也就是说，我们可以绑定到已经使用 startService() 启动的服务。例如，可以通过使用 Intent（标识要播放的音乐）调用 startService() 来启动后台音乐服务。随后，可能在用户需要稍加控制播放器或获取有关当前播放歌曲的信息时，Activity 可以通过调用 bindService() 绑定到服务。在这种情况下，除非所有客户端均取消绑定，否则 stopService() 或 stopSelf() 不会真正停止服务。因此在这种情况下我们需要特别注意。

## 9.Android 5.0以上的隐式启动问题

既然有隐式启动，那么就会有显示启动，那就先来了解一下什么是隐式启动和显示启动。

显示启动

直接上代码一目了然，不解释了。

//显示启动

Intent intent = new Intent(this,ForegroundService.class);

startService(intent);

隐式启动

需要设置一个Action，我们可以把Action的名字设置成Service的全路径名字，在这种情况下android:exported默认为true。

final Intent serviceIntent=new Intent();

serviceIntent.setAction("com.android.ForegroundService");

startService(serviceIntent);

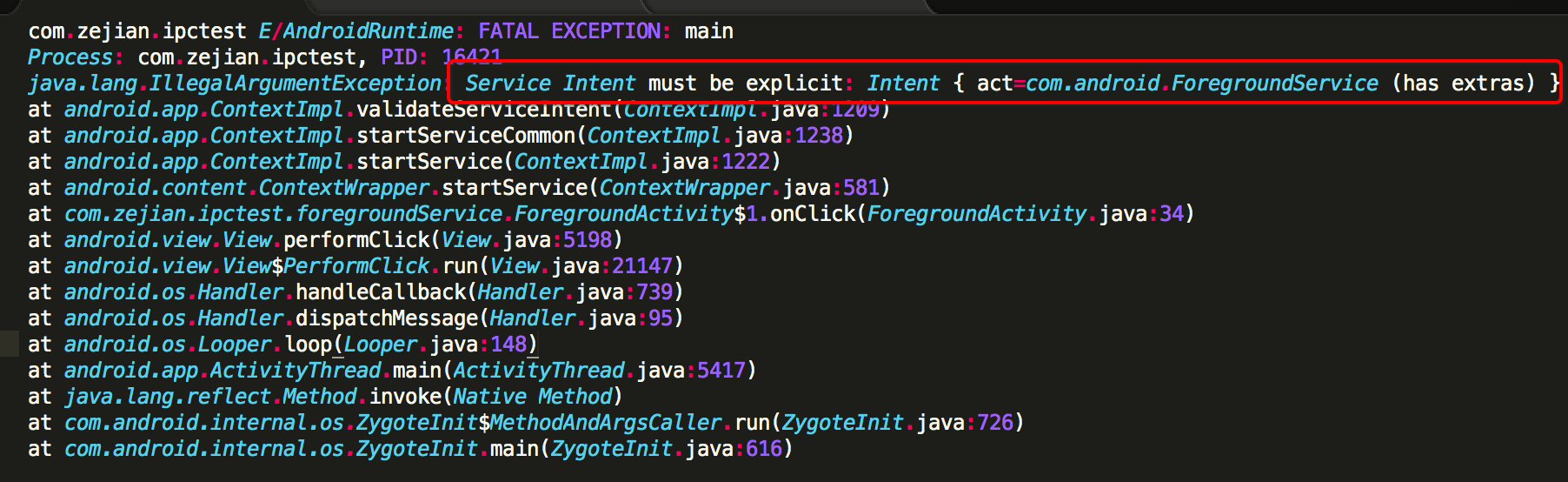
存在的意义

如果在同一个应用中，两者都可以用。

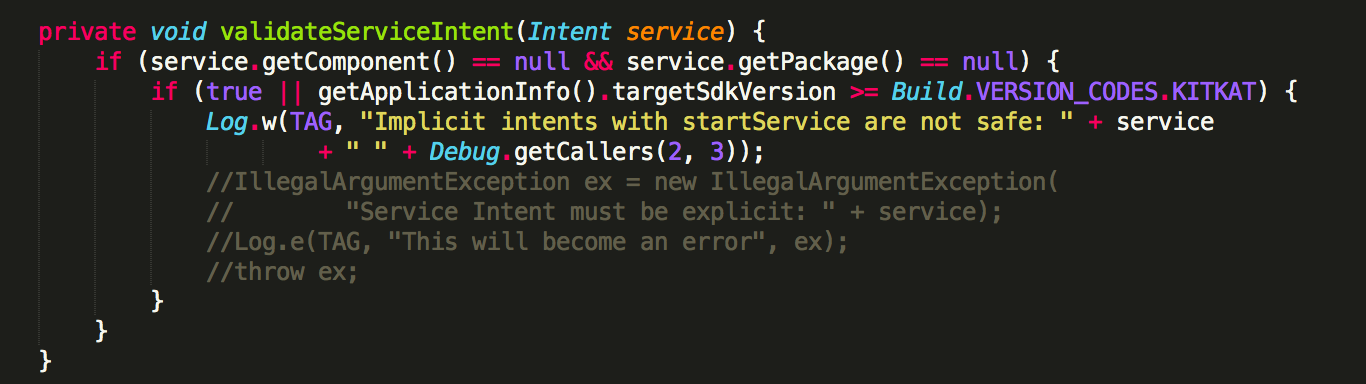
在不同应用时，只能用隐式启动。

Android 5.0以上的隐式启动问题

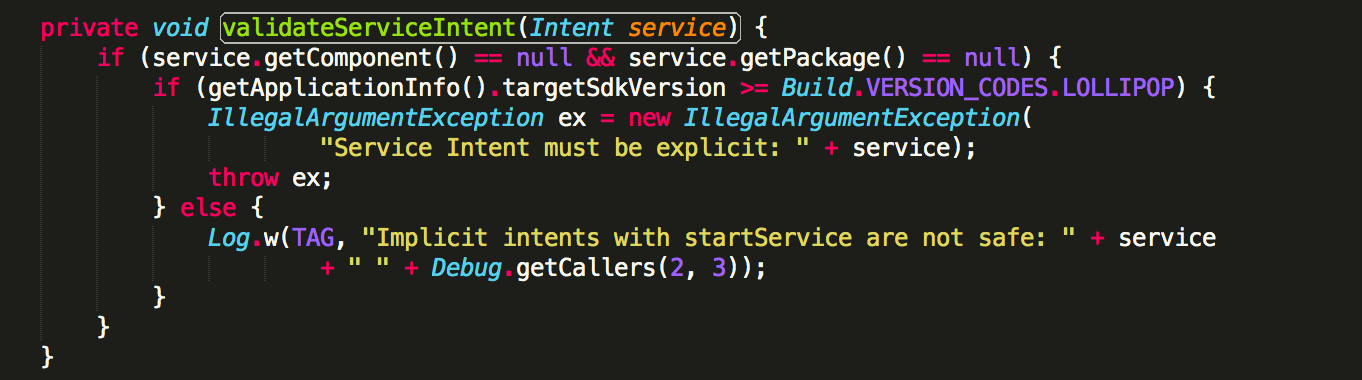
  Android 5.0之后google出于安全的角度禁止了隐式声明Intent来启动Service。如果使用隐式启动Service，会出没有指明Intent的错误，如下：



  主要原因我们可以从源码中找到，这里看看Android 4.4的ContextImpl源码中的validateServiceIntent(Intent service),可知如果启动service的intent的component和package都为空并且版本大于KITKAT的时候只是报出一个警报,告诉开发者隐式声明intent去启动Service是不安全的.



  而在android5.0之后呢？我们这里看的是android6.0的源码如下（sublime text查android各个版本源码就是爽呀！！）：



  从源码可以看出如果启动service的intent的component和package都为空并且版本大于LOLLIPOP(5.0)的时候,直接抛出异常，该异常与之前隐式启动所报的异常时一致的。那么该如何解决呢？

解决方式

设置Action和packageName

final Intent serviceIntent=new Intent();

serviceIntent.setAction("com.android.ForegroundService");

serviceIntent.setPackage(getPackageName());//设置应用的包名

startService(serviceIntent);

将隐式启动转换为显示启动

public static Intent getExplicitIntent(Context context, Intent implicitIntent) {

// Retrieve all services that can match the given intent

PackageManager pm = context.getPackageManager();

List<ResolveInfo> resolveInfo = pm.queryIntentServices(implicitIntent, 0);

// Make sure only one match was found

if (resolveInfo == null || resolveInfo.size() != 1) {

return null;

}

// Get component info and create ComponentName

ResolveInfo serviceInfo = resolveInfo.get(0);

String packageName = serviceInfo.serviceInfo.packageName;

String className = serviceInfo.serviceInfo.name;

ComponentName component = new ComponentName(packageName, className);

// Create a new intent. Use the old one for extras and such reuse

Intent explicitIntent = new Intent(implicitIntent);

// Set the component to be explicit

explicitIntent.setComponent(component);

return explicitIntent;

}

调用方式如下：

Intent mIntent=new Intent();//辅助Intent

mIntent.setAction("com.android.ForegroundService");

final Intent serviceIntent=new Intent(getExplicitIntent(this,mIntent));

startService(serviceIntent);

到此问题完美解决。

## 10.如何保证服务不被杀死

  实际上这种做法并不推荐，但是既然谈到了，我们这里就给出一些实现思路吧。主要分以下3种情况

因内存资源不足而杀死Service

这种情况比较容易处理，可将onStartCommand() 方法的返回值设为 START\_STICKY或START\_REDELIVER\_INTENT ，该值表示服务在内存资源紧张时被杀死后，在内存资源足够时再恢复。也可将Service设置为前台服务，这样就有比较高的优先级，在内存资源紧张时也不会被杀掉。这两点的实现，我们在前面已分析过和实现过这里就不重复。简单代码如下：

/\*\*

\* 返回 START\_STICKY或START\_REDELIVER\_INTENT

\* @param intent

\* @param flags

\* @param startId

\* @return

\*/

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

// return super.onStartCommand(intent, flags, startId);

return START\_STICKY;

}

用户通过 settings -> Apps -> Running -> Stop 方式杀死Service

这种情况是用户手动干预的，不过幸运的是这个过程会执行Service的生命周期，也就是onDestory方法会被调用，这时便可以在 onDestory() 中发送广播重新启动。这样杀死服务后会立即启动。这种方案是行得通的，但为程序更健全，我们可开启两个服务，相互监听，相互启动。服务A监听B的广播来启动B，服务B监听A的广播来启动A。这里给出第一种方式的代码实现如下：

package com.zejian.ipctest.neverKilledService;

import android.app.Service;

import android.content.BroadcastReceiver;

import android.content.Context;

import android.content.Intent;

import android.content.IntentFilter;

import android.os.IBinder;

import android.support.annotation.Nullable;

/\*\*

\* Created by zejian

\* Time 2016/10/4.

\* Description:用户通过 settings -> Apps -> Running -> Stop 方式杀死Service

\*/

public class ServiceKilledByAppStop extends Service{

private BroadcastReceiver mReceiver;

private IntentFilter mIF;

@Nullable

@Override

public IBinder onBind(Intent intent) {

return null;

}

@Override

public void onCreate() {

super.onCreate();

mReceiver = new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

Intent a = new Intent(ServiceKilledByAppStop.this, ServiceKilledByAppStop.class);

startService(a);

}

};

mIF = new IntentFilter();

//自定义action

mIF.addAction("com.restart.service");

//注册广播接者

registerReceiver(mReceiver, mIF);

}

@Override

public void onDestroy() {

super.onDestroy();

Intent intent = new Intent();

intent.setAction("com.restart.service");

//发送广播

sendBroadcast(intent);

unregisterReceiver(mReceiver);

}

}

用户通过 settings -> Apps -> Downloaded -> Force Stop 方式强制性杀死Service

这种方式就比较悲剧了，因为是直接kill运行程序的，不会走生命周期的过程,前面两种情况只要是执行Force Stop ，也就废了。也就是说这种情况下无法让服务重启，或者只能去设置Force Stop 无法操作了，不过也就没必要了，太流氓了。。。。

ok~，以上便是保证服务在一定场景下不被杀死的解决思路，关于第3种情况，如果有解决方案，请留言哈。好，关于Service的全部介绍就此完结。

## 11 Service优先级

--------------------------------------------------------------------------------------------

service：是一个后台服务，专门用来处理常驻后台的工作的组件。

即时通讯：service来做常驻后台的心跳传输。

1.良民：核心服务尽可能地轻！！！

很多人喜欢把所有的后台操作都集中在一个service里面。

为核心服务专门做一个进程，跟其他的所有后台操作隔离。

树大招风，核心服务千万要轻。

一、优先级

进程的重要性优先级：（越往后的就越容易被系统杀死）

1.前台进程；Foreground process

1）用户正在交互的Activity（onResume（））

2）当某个Service绑定正在交互的Activity。

3）被主动调用为前台Service（startForeground()）

4）组件正在执行生命周期的回调（onCreate()/onStart()/onDestroy()）

5）BroadcastReceiver 正在执行onReceive();

2.可见进程；Visible process

1)我们的Activity处在onPause()（没有进入onStop()）

2)绑定到前台Activity的Service。

3.服务进程；Service process

简单的startService()启动。

4.后台进程；Background process

对用户没有直接影响的进程----Activity出于onStop()的时候。

android:process=":xxx"

5.空进程； Empty process

不含有任何的活动的组件。（android设计的，为了第二次启动更快，采取的一个权衡）

二、如何提升进程的优先级（尽量做到不轻易被系统杀死）

1.QQ采取在锁屏的时候启动一个1个像素的Activity，当用户解锁以后将这个Activity结束掉（顺便同时把自己的核心服务再开启一次）。被用户发现了就不好了。

故事：小米撕逼。

背景：当手机锁屏的时候什么都干死了，为了省电。

锁屏界面在上面盖住了。

监听锁屏广播，锁了---启动这个Activity。

监听锁屏的， 开启---结束掉这个Activity。

要监听锁屏的广播---动态注册。

ScreenListener.begin(new xxxListener

onScreenOff()

);

被系统无法杀死的进程。

2.app运营商和手机厂商可能有合作关系---白名单。

3.双进程守护---可以防止单个进程杀死，同时可以防止第三方的360清理掉。

一个进程被杀死，另外一个进程又被他启动。相互监听启动。

A<--->B

杀进程是一个一个杀的。本质是和杀进程时间赛跑。

4.JobScheduler

把任务加到系统调度队列中，当到达任务窗口期的时候就会执行，我们可以在这个任务里面启动我们的进程。

这样可以做到将近杀不死的进程。

5.监听QQ,微信，系统应用，友盟，小米推送等等的广播，然后把自己启动了。

6.利用账号同步机制唤醒我们的进程

AccountManager

7.NDK来解决，Native进程来实现双进程守护。

总结：要根据自己的需要来使用。