Android Service那些事

|  |  |  |
| --- | --- | --- |
| 版本号 | 描述 | 日期(更新日期)/更新人 |
| 1.0 | Android Service使用分析 | 2018/08/10 刘志保 |
| 1.1 | onServiceDisconnected()  何时调用,以及C/S两边线程 | 2018/08/23 刘志保 |
| 工程源码 | https://github.com/MMLoveMeMM/AngryPandaService.git | |

目录

[预备知识: 3](#_Toc522789118)

[<1> : IntentService : 这种是在Service的基础上改造了一下,改成Handler+Service,准确的说事Looper+Thread+Service,拥有自己事物消息处理过程. 3](#_Toc522789119)

[<2> : 现在谈谈Messager+Service,其实这个很简单,借助Messager+Handler+Service(常驻),先看看Messager.java类 5](#_Toc522789120)

[<3> : AIDL+Service 产生一个远程Service服务,这个算是常规的远程服务开发方式,这里开发没有太多技术.主要是考虑一个问题:如何让远程服务端知道客户调用端crash了,然后释放资源,又如何让客户端知道远程端服务由于异常crash了而做响应的调整. 8](#_Toc522789121)

[<4> : 拓展一下Service,利用Binder内重写onTransact方法,参考对应工程的TransactService.java代码 13](#_Toc522789122)

[<5> : 再谈谈服务常驻,不死的问题.这个没什么太多经验,唯一最好的方式就是服务程序代码质量优秀,不会出现crash的问题.当然投机取巧方法,如<1>挂钩子,<2>再搞一个独立的service来监听工作服务service,<3>看门狗,<4> 还有网上一个什么开源的杀不死;其实这些都是浮云. 13](#_Toc522789123)

[<6> : 为什么Service crash两次就不会再自动重启呢? 13](#_Toc522789124)

[<7> : 远程服务权限---检查客户端权限,限制非法客户端连接 14](#_Toc522789125)

[<8> : 这个地方onServiceDisconnected()方法什么时候调用.第一印象是觉得客户端调用 16](#_Toc522789126)

预备知识:

参考”Android Handler事件分析处理.docx”文档,熟悉Handler,以及HandlerThread(集成Thread+Looper的类).

本文的目的:

介绍分析Service,远程Service, IntentService之间的区别和相同点.以及和Service通讯的一些注意事项.

大概如下:

<1> : IntentService : 这种是在Service的基础上改造了一下,改成Handler+Service,准确的说事Looper+Thread+Service,拥有自己事物消息处理过程.

首先看看IntentService源码:

先看看有什么:

**private volatile** Looper **mServiceLooper**;  
**private volatile** ServiceHandler **mServiceHandler**;

@Override  
**public void** onCreate() {  
 *//* ***TODO: It would be nice to have an option to hold a partial wakelock*** *// during processing, and to have a static startService(Context, Intent)  
 // method that would launch the service & hand off a wakelock.* **super**.onCreate();  
 HandlerThread thread = **new** HandlerThread(**"IntentService["** + **mName** + **"]"**);  
 thread.start();  
  
 **mServiceLooper** = thread.getLooper();  
 **mServiceHandler** = **new** ServiceHandler(**mServiceLooper**);  
}

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

根据源码主体由三个在支撑,ServiceHandler(即Handler),Looper,以及HandlerThread.整体的意义是这样的:Service内部创建一个线程,并且这个线程自己维护管理一个消息队列,而这个线程和消息处理是通过HandlerThread来实现的,然后返回给**mServiceLooper** 和**mServiceHandler** 在Service里面进行逻辑管理,状态逻辑靠这个完成了,那消息事件怎么来?又怎么处理的过程?

Service被每次startService启动后,会陆续调到

@Override  
**public void** onStart(@Nullable Intent intent, **int** startId) {  
 Message msg = **mServiceHandler**.obtainMessage();  
 msg.**arg1** = startId;  
 msg.**obj** = intent;  
 **mServiceHandler**.sendMessage(msg);  
}

这个就是消息的产生由来,然后利用**mServiceHandler** 将消息发送到消息队列等待处理.当消息被处理回调时:

@Override  
**public void** handleMessage(Message msg) {  
 onHandleIntent((Intent)msg.**obj**);  
 stopSelf(msg.**arg1**);  
}

首先调用onHandleIntent这个将会执行继承IntentService子类中的方法(父类调用子类的具体实现) 也就是UsingIntentService.java类中的onHandleIntent(Intent)方法.

<a> : 这样就实现了一个什么过程呢?

Service为了发过来的事务不阻塞,就在自己内部新建了一个Thread来处理事务,为了保证事务是串行一个一个的处理的,又引入Looper,进行消息队列管理,将事务包装成消息push到消息队列,然后”慢慢”来处理.

<b> : 那么事务处理的生命周期是如何?

@Override  
**protected void** onHandleIntent(@Nullable Intent intent) {  
  
 *// 这里面已经是从消息队列中一个一个被拿出来处理的,  
 // 即使按键再快,也是间隔3s,一个一个执行* Bundle bundle=intent.getBundleExtra(**"bundle"**);  
 String key=bundle.getString(**"key"**);  
 Log.*e*(***TAG***,**"key : "**+key);  
  
 **try** {  
 Thread.*sleep*(3000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
}

生命周期为大概3s(我是说这段代码会让IntentService活过3s),因为在这里每次处理的onHandleIntent执行一次大概就3s.

然后实现:

@Override  
**public void** onDestroy() {  
 **super**.onDestroy();  
 Log.*e*(***TAG***,**"onDestroy"**);  
}

这里直接给个结果吧! 只要每次发给服务处理的事务间隔不超过3s钟,都是使用同一个IntentService对象,如果超过3s(即onHandleIntent执行完成),就会立即调用onDestroy将IntentService销毁.如果是在3s钟内调用,事务就会入消息队列,这个时候只要消息队列还有事务需要处理,IntentService会一直存在,直到处理完所有的消息事务,才会调用onDestroy().所以只要还在IntentService周期内不断有事务需要处理进入消息队列,IntentService就不会销毁,一旦事务处理完毕,就会立即调用onDestroy().在周期这一点上,它不同于普通Service和远程Service,IntentService执行完任务就会销毁自己,而不是长留!

<2> : 现在谈谈Messager+Service,其实这个很简单,借助Messager+Handler+Service(常驻),先看看Messager.java类

**public final class** Messenger **implements** Parcelable {  
 **private final** IMessenger **mTarget**;

在看看系统源码,查看IMessager,发现它就是个IMessage.aidl的接口

**package** [android](http://androidxref.com/8.1.0_r33/s?defs=android&project=frameworks).[os](http://androidxref.com/8.1.0_r33/s?defs=os&project=frameworks);

[19](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IMessenger.aidl" \l "19)

[20](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IMessenger.aidl" \l "20)**import** [android](http://androidxref.com/8.1.0_r33/s?defs=android&project=frameworks).[os](http://androidxref.com/8.1.0_r33/s?defs=os&project=frameworks).[Message](http://androidxref.com/8.1.0_r33/s?defs=Message&project=frameworks);

[21](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IMessenger.aidl" \l "21)

[22](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IMessenger.aidl" \l "22)/\*\* **@hide** \*/

[23](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IMessenger.aidl" \l "23)[oneway](http://androidxref.com/8.1.0_r33/s?defs=oneway&project=frameworks) **interface** [IMessenger](http://androidxref.com/8.1.0_r33/s?defs=IMessenger&project=frameworks) {

[24](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IMessenger.aidl" \l "24) **void** [send](http://androidxref.com/8.1.0_r33/s?defs=send&project=frameworks)([in](http://androidxref.com/8.1.0_r33/s?defs=in&project=frameworks) [Message](http://androidxref.com/8.1.0_r33/s?defs=Message&project=frameworks) [msg](http://androidxref.com/8.1.0_r33/s?defs=msg&project=frameworks));

[25](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IMessenger.aidl" \l "25)}

其中就是一个发送Message的接口.

Messager.java类中,

**public** Messenger(Handler target) {  
 **mTarget** = target.getIMessenger();  
}

这里突然发现Handler里面居然还有个getIMessenger()方法,再看看Handler源码:

**final** IMessenger getIMessenger() {  
 **synchronized** (**mQueue**) {  
 **if** (mMessenger != **null**) {  
 **return** mMessenger;  
 }  
 mMessenger = **new** MessengerImpl();  
 **return** mMessenger;  
 }  
}  
  
**private final class** MessengerImpl **extends** IMessenger.Stub {  
 **public void** send(Message msg) {  
 msg.**sendingUid** = Binder.*getCallingUid*();  
 Handler.**this**.sendMessage(msg);  
 }  
}

最终还是IMessager.stub,那么Messager.java中:

**public** IBinder getBinder() {  
 **return mTarget**.asBinder();  
}

其中mTarget = Handler.getIMessenger()=IMessenger.Stub

然后**mTarget**.asBinder() = IMessenger.Stub.asBinder()

这样就很容易理解如何使用了

**public class** MessageService **extends** Service {  
  
 **private final static** String ***TAG***=MessageService.**class**.getName();  
 **final** Messenger **mMessenger** = **new** Messenger(**new** EventHandler());  
 @Nullable  
 @Override  
 **public** IBinder onBind(Intent intent) {  
 Log.*d*(***TAG***,**"onBind"**);  
 **return mMessenger**.getBinder();  
 }  
  
 **class** EventHandler **extends** Handler {  
 @Override  
 **public void** handleMessage(Message msg) {  
 **switch** (msg.**what**) {  
 **case** 1001:  
 **case** 1002:  
 Toast.*makeText*(getApplicationContext(), **"hello "**+msg.**what**, Toast.***LENGTH\_SHORT***).show();  
 **break**;  
 **default**:  
 **super**.handleMessage(msg);  
 }  
 }  
 }

注意onBind(Intent intent)中返回IBinder的对象.

应用过程:

Message msg = Message.*obtain*(**null**, 1002, 0, 0);  
MessagerProxy.*getInstance*().sendMessage(msg);

IMessager.aidl发送过程是使用了Message作为数据结构传递数据的.

消息最终通过Messager.java中的

**public void** send(Message message) **throws** RemoteException {  
 **mTarget**.send(message);  
}

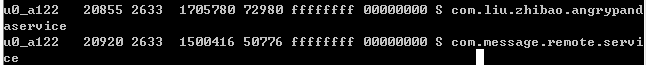
传递到Service端.

综合上述 : Messager+Service其实也就是系统自身自带了IMessager.aidl提供给开发使用的一种快捷绿色通道.相当于系统默认封装了一个AIDL接口,让开发可以快速使用,达到界面和Service通信的目的,不用过分考虑开发自己的AIDL接口.

注意看MessageService.java,居然可以使用Toast弹气泡,说明什么,说明这是在主线程中(UI主线程),下面在修改AndroidManifest.xml配置:

<**service android:name=".service.MessageService"  
 android:exported="true"  
 android:process="com.message.remote.service"**/>

运行查看进程:



然后看结果,还是有气泡弹出.为什么?

我们还是回到Handler.java源码:

**private final class** MessengerImpl **extends** IMessenger.Stub {  
 **public void** send(Message msg) {  
 msg.**sendingUid** = Binder.*getCallingUid*();  
 Handler.**this**.sendMessage(msg);  
 }  
}

然后再修改MessageService.java代码,气泡代码改一下:

Toast.*makeText*(getApplicationContext(), **"hello "**+msg.**what**+**" uid : "**+msg.**sendingUid**, Toast.***LENGTH\_SHORT***).show();

看看UID哈,然后再MainActivity.java中也打印出来:根据包名,获取UID,

**try** {  
 PackageManager pm = getPackageManager();  
 @SuppressLint(**"WrongConstant"**) ApplicationInfo ai = pm.getApplicationInfo(**"com.liu.zhibao.angrypandaservice"**, PackageManager.***GET\_ACTIVITIES***);  
 Log.*e*(***TAG***, **"!!"** + ai.**uid**);  
} **catch** (PackageManager.NameNotFoundException e) {  
 e.printStackTrace();  
}

运行程序,就会发现MainActivity.java和MessageService.java中的气泡弹出的UID值是一样的.为什么,我们再研读研读Handler源码,发现这个在使用的时候,我们并没有开辟新的线程,而仍然在使用sMainLooper,即主线程中操作.

**public** Messenger(Handler target) {  
 mTarget = target.getIMessenger();  
}

我们在Service中new Messager时,把getIMessenger()给了操作目标,而这个在Handler里面全局的

**final** IMessenger getIMessenger() {  
 **synchronized** (**mQueue**) {  
 **if** (**mMessenger** != **null**) {  
 **return mMessenger**;  
 }  
 **mMessenger** = **new** MessengerImpl();  
 **return mMessenger**;  
 }  
}

mMessenger只会全局创建一次.不过这个过程有点没说清,还需要再搞一下.

<3> : AIDL+Service 产生一个远程Service服务,这个算是常规的远程服务开发方式,这里开发没有太多技术.主要是考虑一个问题:如何让远程服务端知道客户调用端crash了,然后释放资源,又如何让客户端知道远程端服务由于异常crash了而做响应的调整.

<a> : 首先看看远程服务端Service如何知道客户端已经由于异常crash了或者退出了.

服务端的代码:

**package** com.liu.zhibao.angrypandaservice.service;  
  
**import** android.app.Service;  
**import** android.content.Intent;  
**import** android.os.IBinder;  
**import** android.os.RemoteException;  
**import** android.util.Log;  
  
**import** com.liu.zhibao.angrypandaservice.aidl.IDeathCheckInteface;  
  
**import** org.jetbrains.annotations.Nullable;  
  
*/\*\*  
 \* Created by zhibao.Liu on 2018/8/13.  
 \*  
 \** ***@version*** *:  
 \** ***@date*** *: 2018/8/13  
 \** ***@des*** *:  
 \** ***@see{@link}*** *\*/***public class** DeathCheckService **extends** Service {  
  
 **private final static** String ***TAG***=DeathCheckService.**class**.getName();  
  
 **private** IBinder **mClient** = **null**;  
 @Nullable  
 @Override  
 **public** IBinder onBind(Intent intent) {  
 **return mDeathCheck**;  
 }  
  
 */\*  
 \* 检查客户端是否crash  
 \* \*/* **private class** CheckDeathRecipient **implements** IBinder.DeathRecipient {  
  
 @Override  
 **public void** binderDied() {  
 Log.*e*(***TAG***, **"client has died"**);  
 }  
  
 }  
  
 **private** IDeathCheck **mDeathCheck**=**new** IDeathCheck();  
 **private class** IDeathCheck **extends** IDeathCheckInteface.Stub{  
 @Override  
 **public void** checkDeath(String data) **throws** RemoteException {  
 Log.*e*(***TAG***, **"client send : "**+data);  
 }  
 @Override  
 **public void** setBinder(IBinder client) **throws** RemoteException {  
 **mClient** = client;  
 **mClient**.linkToDeath(**new** CheckDeathRecipient(), 0);  
 }  
 }  
  
}

客户端:

**private** IBinder **mBinder** = **new** Binder();

@Override  
**public void** onServiceConnected(ComponentName name, IBinder service) {  
 **mService** = IDeathCheckInteface.Stub.*asInterface*(service);  
 **try** {  
 **mService**.setBinder(**mBinder**);  
 } **catch** (RemoteException e) {  
 e.printStackTrace();  
 }  
}

在客户端连接到服务端后,将客户端新建的Binder对象传递给远程服务端.然后我们在看看IBinder源码,它是如何去知道IBinder是否还存在的:

[65](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "265) /\*\*

[266](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "266) \* Interface for receiving a callback when the process hosting an IBinder

[267](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "267) \* has gone away.

[268](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "268) \*

[269](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "269) \* **@see** #linkToDeath

[270](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "270) \*/

[271](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "271) **public** **interface** ***[DeathRecipient](http://androidxref.com/8.1.0_r33/s?refs=DeathRecipient&project=frameworks)*** {

[272](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "272) **public** **void** **[binderDied](http://androidxref.com/8.1.0_r33/s?refs=binderDied&project=frameworks)**();

[273](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "273) }

[274](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "274)

[275](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "275) /\*\*

[276](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "276) \* Register the recipient for a notification if this binder

[277](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "277) \* goes away. If this binder object unexpectedly goes away

[278](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "278) \* (typically because its hosting process has been killed),

[279](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "279) \* then the given {**@link** DeathRecipient}'s

[280](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "280) \* {**@link** DeathRecipient#binderDied DeathRecipient.binderDied()} method

[281](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "281) \* will be called.

[282](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "282) \*

[283](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "283) \* <p>You will only receive death notifications for remote binders,

[284](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "284) \* as local binders by definition can't die without you dying as well.

[285](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "285) \*

[286](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "286) \* **@throws** *RemoteException* if the target IBinder's

[287](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "287) \* process has already died.

[288](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "288) \*

[289](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "289) \* **@see** #unlinkToDeath

[290](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "290) \*/

[291](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "291) **public** **void** **[linkToDeath](http://androidxref.com/8.1.0_r33/s?refs=linkToDeath&project=frameworks)**(@[NonNull](http://androidxref.com/8.1.0_r33/s?defs=NonNull&project=frameworks) [**DeathRecipient**](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java#DeathRecipient) **[recipient](http://androidxref.com/8.1.0_r33/s?refs=recipient&project=frameworks)**, **int** **[flags](http://androidxref.com/8.1.0_r33/s?refs=flags&project=frameworks)**)

[292](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "292) **throws** [RemoteException](http://androidxref.com/8.1.0_r33/s?defs=RemoteException&project=frameworks);

[293](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "293)

[294](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "294) /\*\*

[295](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "295) \* Remove a previously registered death notification.

[296](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "296) \* The recipient will no longer be called if this object

[297](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "297) \* dies.

[298](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "298) \*

[299](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "299) \* **@return** {**@code** true} if the <var>recipient</var> is successfully

[300](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "300) \* unlinked, assuring you that its

[301](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "301) \* {**@link** DeathRecipient#binderDied DeathRecipient.binderDied()} method

[302](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "302) \* will not be called; {**@code** false} if the target IBinder has already

[303](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "303) \* died, meaning the method has been (or soon will be) called.

[304](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "304) \*

[305](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "305) \* **@throws** *java.util.NoSuchElementException* if the given

[306](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "306) \* <var>recipient</var> has not been registered with the IBinder, and

[307](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "307) \* the IBinder is still alive. Note that if the <var>recipient</var>

[308](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "308) \* was never registered, but the IBinder has already died, then this

[309](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "309) \* exception will <em>not</em> be thrown, and you will receive a false

[310](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "310) \* return value instead.

[311](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "311) \*/

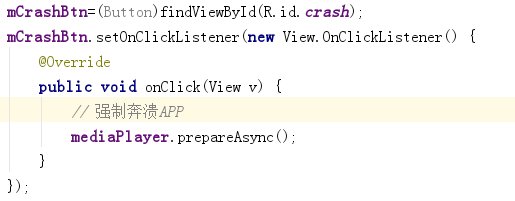
[312](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java" \l "312) **public** **boolean** **[unlinkToDeath](http://androidxref.com/8.1.0_r33/s?refs=unlinkToDeath&project=frameworks)**(@[NonNull](http://androidxref.com/8.1.0_r33/s?defs=NonNull&project=frameworks) [**DeathRecipient**](http://androidxref.com/8.1.0_r33/xref/frameworks/base/core/java/android/os/IBinder.java#DeathRecipient) **[recipient](http://androidxref.com/8.1.0_r33/s?refs=recipient&project=frameworks)**, **int** **[flags](http://androidxref.com/8.1.0_r33/s?refs=flags&project=frameworks)**);

接口***[DeathRecipient](http://androidxref.com/8.1.0_r33/s?refs=DeathRecipient&project=frameworks)*** : 这个是Binder不存在或者释放时会被调用的接口,只要注册了这个接口的对象,对应的Binder只要不存在了的时候,就会调用这个接口.

方法**[linkToDeath](http://androidxref.com/8.1.0_r33/s?refs=linkToDeath&project=frameworks)** : 这个是IBinder注册一个Binder不存在时需要回调的方法.也就是说这个方法配合上面的接口,就可以组成一个监听Binder是否还存在的机制,只要Binder不存在了,就会触发接口的回调.

方法**[unlinkToDeath](http://androidxref.com/8.1.0_r33/s?refs=unlinkToDeath&project=frameworks)** : 与上面的方法相反,反注册,或者说注销监听.Binder不存在时会抛异常.

熟悉了这个地方.看看前面的测试程序,在客户端new了一个Binder,然后通过AIDL的方法setBinder传递给远程Service,然后在远程Service里面监听这个Binder状态情况,如果客户端销毁了,我在客户端故意制造crash:



MediaPlayer故意不初始化就使用,导致客户端crash.然后服务端就可以看到客户端crash导致客户端的Binder销毁,远程服务的监听回调被触发:



<b> : 现在在看看客户端如何监听远程服务端crash后,客户端如果被触发回调,原理与<a> 是一样的.远程服务连接成功,开始监听 :

@Override  
**public void** onServiceConnected(ComponentName name, IBinder service) {  
 **mService** = IDeathCheckInteface.Stub.*asInterface*(service);  
 **try** {  
 *// 将binder传给远程service去监听* **mService**.setBinder(**mBinder**);  
  
 *// 监听远程服务service是否crash* service.linkToDeath(mCheckDeathRecipient

,0);  
 **mRemoteBinder** = service;  
 } **catch** (RemoteException e) {  
 e.printStackTrace();  
 }  
}

回调处理:

*/\*  
\* 检查远程服务端是否crash  
\* \*/***private** CheckDeathRecipient **mCheckDeathRecipient**=**new** CheckDeathRecipient();  
**private class** CheckDeathRecipient **implements** IBinder.DeathRecipient {  
  
 @Override  
 **public void** binderDied() {  
 Log.*e*(***TAG***, **"remote service has died"**);  
 *// 反注册* **if**(**mRemoteBinder**!=**null**){  
 **mRemoteBinder**.unlinkToDeath(**mCheckDeathRecipient**,0);  
 }  
  
 *// 重新绑定* bindService();  
 }  
  
}

然后在远程服务端:

@Override  
**public void** checkDeath(String data) **throws** RemoteException {  
 Log.*e*(***TAG***, **"client send : "**+data);  
 *// 在这个地方故意产生crash* **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **mediaPlayer**.prepareAsync();  
 }  
 }).start();  
}

一点击按钮,远程服务就会挂掉,然后就有打印日志:



<4> : 拓展一下Service,利用Binder内重写onTransact方法,参考对应工程的TransactService.java代码

<5> : 再谈谈服务常驻,不死的问题.这个没什么太多经验,唯一最好的方式就是服务程序代码质量优秀,不会出现crash的问题.当然投机取巧方法,如<1>挂钩子,<2>再搞一个独立的service来监听工作服务service,<3>看门狗,<4> 还有网上一个什么开源的杀不死;其实这些都是浮云.

<6> : 为什么Service crash两次就不会再自动重启呢?

直接看源码关键位置ActiveServices.java类:从这里开始

**final** **void** **[killServicesLocked](http://androidxref.com/8.1.0_r33/s?refs=killServicesLocked&project=frameworks)**([ProcessRecord](http://androidxref.com/8.1.0_r33/s?defs=ProcessRecord&project=frameworks) **[app](http://androidxref.com/8.1.0_r33/s?refs=app&project=frameworks)**, **boolean** **[allowRestart](http://androidxref.com/8.1.0_r33/s?refs=allowRestart&project=frameworks)**) {

[3020](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3020) // Report disconnected services.

[3021](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3021) **if** (**false**) {

[3022](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3022) // XXX we are letting the client link to the service for

[3023](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3023) // death notifications.

[3024](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3024) **if** ([app](http://androidxref.com/8.1.0_r33/s?defs=app&project=frameworks).[services](http://androidxref.com/8.1.0_r33/s?defs=services&project=frameworks).[size](http://androidxref.com/8.1.0_r33/s?defs=size&project=frameworks)() > 0) {

[3025](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3025) [Iterator](http://androidxref.com/8.1.0_r33/s?defs=Iterator&project=frameworks)<[ServiceRecord](http://androidxref.com/8.1.0_r33/s?defs=ServiceRecord&project=frameworks)> [it](http://androidxref.com/8.1.0_r33/s?defs=it&project=frameworks) = [app](http://androidxref.com/8.1.0_r33/s?defs=app&project=frameworks).[services](http://androidxref.com/8.1.0_r33/s?defs=services&project=frameworks).[iterator](http://androidxref.com/8.1.0_r33/s?defs=iterator&project=frameworks)();

[3026](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3026) **while** ([it](http://androidxref.com/8.1.0_r33/s?defs=it&project=frameworks).[hasNext](http://androidxref.com/8.1.0_r33/s?defs=hasNext&project=frameworks)()) {

关注下面几行(当然最好全部看,这里不可能全部贴出来)

**if** (**[allowRestart](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "allowRestart)** && [sr](http://androidxref.com/8.1.0_r33/s?defs=sr&project=frameworks).[crashCount](http://androidxref.com/8.1.0_r33/s?defs=crashCount&project=frameworks) >= [**mAm**](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java#mAm).[mConstants](http://androidxref.com/8.1.0_r33/s?defs=mConstants&project=frameworks).[BOUND\_SERVICE\_MAX\_CRASH\_RETRY](http://androidxref.com/8.1.0_r33/s?defs=BOUND_SERVICE_MAX_CRASH_RETRY&project=frameworks)

[3145](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3145) && ([sr](http://androidxref.com/8.1.0_r33/s?defs=sr&project=frameworks).[serviceInfo](http://androidxref.com/8.1.0_r33/s?defs=serviceInfo&project=frameworks).[applicationInfo](http://androidxref.com/8.1.0_r33/s?defs=applicationInfo&project=frameworks).[flags](http://androidxref.com/8.1.0_r33/s?defs=flags&project=frameworks)

[3146](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3146) &[ApplicationInfo](http://androidxref.com/8.1.0_r33/s?defs=ApplicationInfo&project=frameworks).[FLAG\_PERSISTENT](http://androidxref.com/8.1.0_r33/s?defs=FLAG_PERSISTENT&project=frameworks)) == 0) {

[3147](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "3147) [Slog](http://androidxref.com/8.1.0_r33/s?defs=Slog&project=frameworks).w(**[TAG](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java" \l "TAG)**, "Service crashed " + [sr](http://androidxref.com/8.1.0_r33/s?defs=sr&project=frameworks).[crashCount](http://androidxref.com/8.1.0_r33/s?defs=crashCount&project=frameworks)

[**mAm**](http://androidxref.com/8.1.0_r33/xref/frameworks/base/services/core/java/com/android/server/am/ActiveServices.java#mAm).[mConstants](http://androidxref.com/8.1.0_r33/s?defs=mConstants&project=frameworks).[BOUND\_SERVICE\_MAX\_CRASH\_RETRY](http://androidxref.com/8.1.0_r33/s?defs=BOUND_SERVICE_MAX_CRASH_RETRY&project=frameworks) 这个值系统默认是2,也就是说,service crash两次以后(即超过2次),系统就不会帮你把service重启了,这个很容易试出来,在service触发crash,第三次在命令行就看不到这个service进程ID了.

<7> : 远程服务权限---检查客户端权限,限制非法客户端连接

通过AIDL调用,只要知道AIDL接口,任何其他APP,甚至非法APP都可以连接到远程服务,并且任意修改程序数据,这就很危险了,采用的办法很多,最简单的办法就是检查权限并且检查指定包的权限,因为APK反编译,是可以看到访问的权限.

在工程Manifest.xml先自定义个权限并且使用:

<**permission android:name="com.liu.zhibao.ACCESS\_REMOTE\_SERVICE"  
 android:protectionLevel="normal"**/>  
<**uses-permission android:name="com.liu.zhibao.ACCESS\_REMOTE\_SERVICE"**/>

<**service  
 android:name=".service.RemoteService"  
 android:exported="true"  
 android:permission="com.liu.zhibao.ACCESS\_REMOTE\_SERVICE"  
 android:process="com.liu.zhibao.remote.process"** />

然后再RemoteService.java进行权限检查,这个有两种方式,

<a> : 在连接RemoteService时检查权限:

@Override  
**public** IBinder onBind(Intent intent) {  
  
 Log.*e*(***TAG***,**"pkg name : "**+intent.getComponent().getPackageName());  
 String callingPkgName = intent.getComponent().getPackageName();  
 **if**(checkPermission(getApplicationContext(),***PERMISSION\_NAME***,callingPkgName*/\* 客户端(连接服务者)的包名 \*/*)) {  
 */\*  
 \* 有指定权限的APP 才能够连接到这个服务,否则不准对方连接  
 \* \*/* **return stub**;  
 }**else**{  
 */\*  
 \* 无权限禁止连接  
 \* \*/* **return null**;  
 }  
}

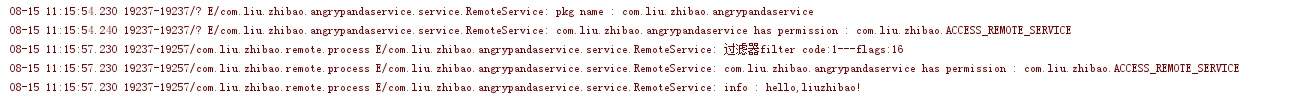
*/\*  
\* 还是需要通过检查对应连接者的包名来检查这个连接者是否有制定的权限  
\* \*/***private boolean** checkPermission(Context context, String permName, String pkgName){  
 PackageManager pm = context.getPackageManager();  
 **if**(PackageManager.***PERMISSION\_GRANTED*** == pm.checkPermission(permName, pkgName)){  
 Log.*e*(***TAG***,pkgName + **" has permission : "** + permName);  
 **return true**;  
 }**else**{  
 Log.*e*(***TAG***,pkgName + **"has no permission : "** + permName);  
 **return false**;  
 }  
}

不过这里还没有检查包名,只检查了APK是否有权限,个人建议最好也检查包名,只给指定的包名APP连接.

<b> : 允许连接到服务,调用接口处检查权限,没权限,接口直接屏蔽.

@Override  
**public boolean** onTransact(**int** code, Parcel data, Parcel reply, **int** flags) **throws** RemoteException {  
 Log.*e*(***TAG***,**"过滤器filter code:"** + code + **"---flags:"** + flags);  
 String packageName = **null**;  
 String[] packages = getPackageManager().  
 getPackagesForUid(*getCallingUid*());  
 **if** (packages != **null** && packages.**length** > 0) {  
 packageName = packages[0];  
 }  
 **if** (packageName == **null**) {  
 **return false**;  
 }  
 **boolean** checkPermission = checkPermission(RemoteService.**this**, ***PERMISSION\_NAME***, packageName);  
 **if** (!checkPermission) {  
 Log.*e*(***TAG***,**"onTransact check permission : no permission !"**);  
 **return false**;  
 }  
 **return super**.onTransact(code, data, reply, flags);  
}

运行后:



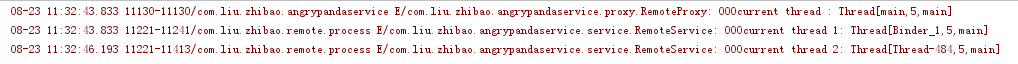
<8> : 这个地方onServiceDisconnected()方法什么时候调用.第一印象是觉得客户端调用

**public void** unBindService(){  
 **if**(**mService**!=**null**) {  
 *mContext*.unbindService(*instance*);  
 }  
}

是否会出呢?实际上这个调用不会触发的,所以平时不要对这个接口有所期待,因为正常情况都不会被调用.

总结网上和实际测验的,是service销毁(无论何种原因)或者crash奔溃的时候才会被调用.(测验程序见github 的remoteService)

<9> : 线程,我在C端和S端(服务端)打印出当前线程信息:



线程组是一样的,但是线程是不一样的.

指的是当前线程的字符串表示形式。Thread[main,5,main]为主线程的表示方式，中括号里的第一个值为当前主线程的名字，第二个为线程级别，第三个为线程组。Thread[Thread-0,5,main]Thread-0为另一个启动的线程，级别为5，属于线程组main

注意事项 :

为了服务端的稳定,尽量不要在AIDL接口实现端的地方直接写业务逻辑代码,这样很危险,因为业务逻辑代码不容易做到100%的安全,如果逻辑业务代码不完美,出现异常或者崩溃,会造成Binder\_1这个线程直接崩溃,连带客户端也随之崩溃.同理,服务端回调到客户端的接口,客户端也不要做实际的业务逻辑代码,如果崩溃,同样会造成C/S两端同时崩溃.个人建议对于APP来说,如果实际要写在接口里面,就要try{}catch(){}将异常捕获.,如果不捕获,或者很难捕获,就使用Handle以消息的形式发送出去处理.推荐的方式:客户端采用try{}catch(){}捕获异常,服务端采用Handle,实现Service+Handle的模式,这种模式还能够解决多个客户端同时调用Service的接口的问题.