Android中的Service：默默的奉献者

前言

IPC跨进程通信:

进程间通信(IPC,Inter-Process Communication)指至少两个进程或线程间传送数据或信号的一些技术或方法

IPC相关的东西，这里面就不可避免的要涉及到service，进程线程这些知识点

最常见的aidl的使用就是Service的跨进程通信了

## 1，什么是service?

Service是一个可以在后台执行长时间操作而不使用用户界面的应用组件。那么问题来了，既然它不使用用户界面，那么它怎么知道应该什么时候开始执行什么操作呢？

答案是——它可以与其他的应用组件形成一些联系，从而可以根据其传来的信息在合适的时候执行合适的操作。

一般来讲，这种联系分为两种：startService()以及bindService()。这两种联系都可以使得一个service开始运行，但是在其他方面有着诸多不同。



注：

1.表格中的“其他组件”不包括Broadcast receiver，其无法直接启动或绑定service。

2.startService()与bindService()并不冲突，同一个service可能既有组件调用了startService()启动它，又有组件与它进行了绑定。当同一个service与其他组件同时存在这两种联系时，其生命周期会发生变化，必须从两种方法的角度看service均停止才能真正停止。

ok，通过这个表格我们可以知道，这两种启动service的方式各有自己的特点，我们可以根据自己的项目需要选择合适的方式。

## 2，如何创建一个service？

创建一个类继承自Service(或它的子类，如IntentService)，重写里面的一些关键的回调方法，如onStartCommand()，onBind()等

在Manifests文件里面为其声明，并根据需要配置一些其他属性。

讲道理，这一切跟新建一个Activity灰常的像，跟孪生兄弟一样——包括其里面的一些关键方法，都充斥着一种浓浓的基情。

##onCreate()##

在每个service的生命周期中这个方法会且仅会调用一次，并且它的调用在onStartCommand()以及onBond()之前，我们可以在这个方法中进行一些一次性的初始化工作。

##onStartCommand()##

当其他组件通过startService()方法启动service时，此方法将会被调用。

##onBind()##

当其他组件通过bindService()方法与service相绑定之后，此方法将会被调用。这个方法有一个IBinder的返回值，这意味着在重写它的时候必须返回一个IBinder对象，它是用来支撑其他组件与service之间的通信的——另外，如果你不想让这个service被其他组件所绑定，可以通过在这个方法返回一个null值来实现。

##onDestroy()##

这是service一生中调用的最后一个方法，当这个方法被调用之后，service就会被销毁。所以我们应当在这个方法里面进行一些资源的清理，比如注册的一些监听器什么的。

在Manifests文件里进行声明的时候，只有android:name属性是必须要有的，其他的属性都可以没有。但是有的时候适当的配置可以让我们的开发进行地更加顺利，所以了解一下注册一个service可以声明哪些属性也是很有必要的。

<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:enabled : 如果为true，则这个service可以被系统实例化，如果为false，则不行。默认为true

android:exported : 如果为true，则其他应用的组件也可以调用这个service并且可以与它进行互动，如果为false，则只有与service同一个应用或者相同user ID的应用可以开启或绑定此service。它的默认值取决于service是否有intent filters。如果一个filter都没有，就意味着只有指定了service的准确的类名才能调用，也就是说这个service只能应用内部使用——其他的应用不知道它的类名。这种情况下exported的默认值就为false。反之，只要有了一个filter，就意味着service是考虑到外界使用的情况的，这时exported的默认值就为true

android:icon : 一个象征着这个service的icon

android:isolatedProcess : 如果设置为true，这个service将运行在一个从系统中其他部分分离出来的特殊进程中，我们只能通过Service API来与它进行交流。默认为false。

android:label : 显示给用户的这个service的名字。如果不设置，将会默认使用<application>的label属性。

android:name : 这个service的路径名，例如“com.lypeer.demo.MyService”。这个属性是唯一一个必须填的属性。

android:permission : 其他组件必须具有所填的权限才能启动这个service。

android:process : service运行的进程的name。默认启动的service是运行在主进程中的。

下面是一个例子：

public class ServiceDemo extends Service {

private static final String TAG = "ServiceDome";

@Override

public void onCreate() {

super.onCreate();

Log.d(TAG, "onCreate");

//只在service创建的时候调用一次，可以在此进行一些一次性的初始化操作

}

@Override

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

Log.d(TAG, "onStartCommand");

//当其他组件调用startService()方法时，此方法将会被调用

//在这里进行这个service主要的操作

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

}

@Nullable

@Override

public IBinder onBind(Intent intent) {

Log.d(TAG, "onBind");

//当其他组件调用bindService()方法时，此方法将会被调用

//如果不想让这个service被绑定，在此返回null即可

return null;

}

@Override

public void onDestroy() {

Log.d(TAG, "onDestroy");

//service调用的最后一个方法

//在此进行资源的回收

super.onDestroy();

}

}

<!--in manifests -->

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

## 3，如何启动service？

前面讲了两种启动service的方式，但是都讲的比较概括，这里就详细的讲一下这两种方式。

### 3.1，startService()

另一个组件通过调用startService()方法，就可以启动一个特定的service，并且这将导致service中的onStartCommand()方法被调用。在调用startService()方法的时候，其他组件需要在方法中传递一个intent参数，然后service将会在onStartCommand()中接收这个intent，并获取一些数据。比如此时某个Activity要将一些数据存入数据库中，我就可以通过intent把数据传入service，然后让service去进行连接数据库，存储数据等操作，而此时用户可以执行其他的任何操作——甚至包括销毁那个Activity——这并不会影响service存储数据这件事。

当一个service通过这种方式启动之后，它的生命周期就已经不受启动它的组件影响了，它可以在后台无限期的运行下去，只要service自身没有调用stopSelf()并且其他的组件没有调用针对它的stopService()。

另外，如果确定了使用这种方式启动service并且不希望这个service被绑定的话，那么也许除了传统的创建一个类继承service之外我们有一个更好的选择——继承IntentService。

如果是扩建Service类的话，通常情况下我们需要新建一个用于执行工作的新线程，因为默认情况下service将工作于应用的主线程，而这将会降低所有正在运行的Activity的性能。而IntentService就不同了。它是Service的子类，它使用工作线程来注意的处理所有的startService请求。如果你不要求这个service要同时处理多个请求，那么继承这个类显然要比直接继承Service好到不知道哪里去了——IntentService已经做了这些事：

创建默认的工作线程，用于在应用的主线程外执行传递给 onStartCommand() 的所有 Intent

创建工作队列，用于将一个 Intent 逐一传递给 onHandleIntent() 实现，这样的话就永远不必担心多线程问题了

在处理完所有启动请求后停止服务，从此妈妈再也不用担心我忘记调用 stopSelf() 了

提供 onBind() 的默认实现（返回 null）

提供 onStartCommand() 的默认实现，可将 Intent 依次发送到工作队列和 onHandleIntent() 实现

因此我们只需要实现onHandleIntent()方法来完成具体的功能逻辑就可以了。

6不6？必须6！下面是一个栗子：

//现在在一个Activity里面

Intent intent = new Intent(MainActivity.this , IntentServiceDemo.class);

startService(intent);

public class IntentServiceDemo extends IntentService {

public IntentServiceDemo(String name) {

super(name);

//构造方法

}

@Override

protected void onHandleIntent(Intent intent) {

//在这里根据intent进行操作

}

}

相比上面的继承service实现，这个确实要简单许多。但是要注意的是，如果需要重写其他的方法，比如onDestroy()方法，一定不要删掉它的超类实现！因为它的超类实现里面也许包括了对工作线程还有工作队列的初始化以及销毁等操作，如果没有了的话很容易出问题。

但是，如果你有让service同时处理多个请求的需求，这个时候就只能去继承Service了。这个时候就要自己去处理工作线程那些事。下面是一个官方的栗子：

public class HelloService extends Service {

private Looper mServiceLooper;

private ServiceHandler mServiceHandler;

private final class ServiceHandler extends Handler {

public ServiceHandler(Looper looper) {

super(looper);

}

@Override

public void handleMessage(Message msg) {

long endTime = System.currentTimeMillis() + 5\*1000;

while (System.currentTimeMillis() < endTime) {

synchronized (this) {

try {

wait(endTime - System.currentTimeMillis());

} catch (Exception e) {

}

}

}

stopSelf(msg.arg1);

}

}

@Override

public void onCreate() {

HandlerThread thread = new HandlerThread("ServiceStartArguments",

Process.THREAD\_PRIORITY\_BACKGROUND);

thread.start();

mServiceLooper = thread.getLooper();

mServiceHandler = new ServiceHandler(mServiceLooper);

}

@Override

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

Toast.makeText(this, "service starting", Toast.LENGTH\_SHORT).show();

Message msg = mServiceHandler.obtainMessage();

msg.arg1 = startId;

mServiceHandler.sendMessage(msg);

return START\_STICKY;

}

@Override

public IBinder onBind(Intent intent) {

return null;

}

@Override

public void onDestroy() {

Toast.makeText(this, "service done", Toast.LENGTH\_SHORT).show();

}

}

比起IntentService来，显然做的工作要多的多的多。但是，由于是自己处理的对onStartCommand的调用，它可以同时执行多个请求——虽然官方的栗子里没有这样做。但是如果你想这样做，就可以为每一个请求创建一个线程，然后立即运行这些请求。

另外，我们注意到onStartCommand()的返回值是一个很奇怪的值START\_STICKY，这是个什么呢？或者说这个方法的返回值是用来干嘛的呢？事实上，它的返回值是用来指定系统对当前线程的行为的。它的返回值必须是以下常量之一：

- START\_NOT\_STICKY : 如果系统在 onStartCommand() 返回后终止服务，则除非有挂起 Intent 要传递，否则系统不会重建服务。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\_STICKY : 如果系统在 onStartCommand() 返回后终止服务，则会重建服务并调用 onStartCommand()，但绝对不会重新传递最后一个 Intent。相反，除非有挂起 Intent 要启动服务（在这种情况下，将传递这些 Intent ），否则系统会通过空 Intent 调用 onStartCommand()。这适用于不执行命令、但无限期运行并等待作业的媒体播放器（或类似服务）。

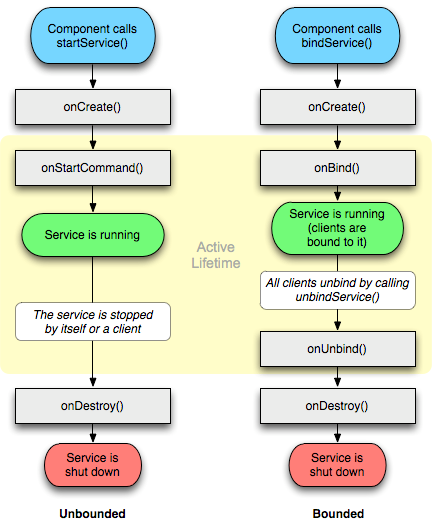
START\_REDELIVER\_INTENT : 如果系统在 onStartCommand() 返回后终止服务，则会重建服务，并通过传递给服务的最后一个 Intent 调用 onStartCommand()。任何挂起 Intent 均依次传递。这适用于主动执行应该立即恢复的作业（例如下载文件）的服务。

除此之外，继承service来实现的service，一定要记得，停止这个service。因为除非系统必须回收内存资源，否则系统不会停止或销毁service。事实上，如果这个service是与可见组件绑定，其几乎永远不会被停止或销毁。在这种情况下，如果你忘记了在其工作完成之后将其停止，势必会造成系统资源的浪费以及电池电量的消耗，而这应当是我们要极力避免的。

### 3.2，bindService()

这是一种比startService更复杂的启动方式，同时使用这种方式启动的service也能完成更多的事情，比如其他组件可向其发送请求，接受来自它的响应，甚至通过它来进行IPC等等。我们通常将绑定它的组件成为客户端，而称它为服务器。

如果要创建一个支持绑定的service，我们必须要重写它的onBind()方法。这个方法会返回一个IBinder对象，它是客户端用来和服务器进行交互的接口。而要得到IBinder接口，我们通常有三种方式：继承Binder类，使用Messenger类，使用AIDL。



https://blog.csdn.net/luoyanglizi/article/details/51586437