前言

前面一篇博文介绍了关于Service的一些基本知识，包括service是什么，怎么创建一个service，创建了一个service之后如何启动它等等。在这一篇博文里有一些需要前一篇铺垫的东西，建议没有看过前一篇博文的同学先去看一下

但是在前一篇博文中也有一些遗漏的东西——主要是关于bindService()这一块的具体细节。由于这一块涉及的东西还是比较多，所以在这里单独提出来了。闲话不多说，进入正文。

正文

1，bindService()

先温故一下上一篇博文的一些内容：

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称为客户端，而称它为服务端。

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。

要完成客户端与服务端的绑定，有两件事要做。

一是在客户端完成bindService的调用以及相关配置，

二是在服务端里面实现onBind()方法的重写，返回一个用做信息交互的IBinder接口。接下来我们就一块一块的来看它的实现方法。

1.1，客户端的配置

客户端原则上来讲调用bindService()方法就可以了，然而事实并没有这么简单。原因就出在bindService()这个方法身上。下面我们来详细的了解一下这个方法：

public boolean bindService(Intent service, ServiceConnection conn, int flags) {

return mBase.bindService(service, conn, flags);

}

可以看到，bindService()方法需要三个参数，

第一个是一个intent，我们都很熟悉——它和startService()里面那个intent是一样的，用来指定启动哪一个service以及传递一些数据过去。

第二个参数可能就有点陌生了，这是个啥？这是实现客户端与服务端通信的一个关键类。要想实现它，就必须重写两个回调方法：onServiceConnected()以及onServiceDisconnected()，而我们可以通过这两个回调方法得到服务端里面的IBinder对象，从而达到通信的目的(下文对此会有更加详细的介绍)。下面是一个例子：

ServiceDemo mService;

//BinderDemo是在ServiceDemo里面的一个继承了Binder的内部类，这是一种得到IBinder接口的方式

//下文会有详述

ServiceDemo.BinderDemo mBinder;

private ServiceConnection mConnection = new ServiceConnection() {

//系统会调用该方法以传递服务的　onBind() 方法返回的 IBinder。

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

//当系统调用 onServiceConnected() 回调方法时，我们可以使用接口定义的方法开始调用服务。

mBinder = (ServiceDemo.BinderDemo) service;

//getService()是BinderDemo中的一个方法

mService = mBinder.getService();

//在此处可以利用得到的ServiceDemo对象调用该类中的构造方法

Log.d(this.getClass().getSimpleName(), "onServiceConnected");

}

//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。当客户端取消绑定时，系统“绝对不会”调用该方法。

@Override

public void onServiceDisconnected(ComponentName name) {

Log.d(this.getClass().getSimpleName(), "onServiceDisconnected");

}

};

上面的例子实现了一个比较普通的ServiceConnection的主要功能，我们可以通过它得到目标service的对象，然后可以调用其内的共有方法，实现客户端与服务端交互的目的。

bindService()方法的第三个参数是一个int值，还叫flag(这flag立的)，它是用来做什么的呢？它是一个指示绑定选项的标志，通常应该是 BIND\_AUTO\_CREATE，以便创建尚未激活的服务。 其他可能的值为 BIND\_DEBUG\_UNBIND 和 BIND\_NOT\_FOREGROUND，或 0（表示无）。

ok，客户端的配置到这里就差不多搞定了，接下来看看服务端需要做些什么。

1.2，服务端的配置

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。

可以看到，这里提出了一个IBinder接口的概念。那么这个IBinder接口是什么呢？它是一个在整个Android系统中都非常重要的东西，是为高性能而设计的轻量级远程调用机制的核心部分。当然，它不仅仅可以用于远程调用，也可以用于进程内调用——事实上，我们现在所说的service这里的IBinder既有可能出现远程调用的场景，比如用它来进行IPC，也有可能出现进程内调用的场景，比如用它来进行同进程内客户端与服务器的交互。IBinder的具体的工作原理在这里就不详述了，以后我应该会就这一块的内容单独写一系列的博客，在这里只需要知道它是客户端用来和服务器进行交互的接口，并且知道可以怎样通过IBinder来实现它们的交互就可以了。

一般来讲，我们有三种方式可以获得IBinder的对象：继承Binder类，使用Messenger类，使用AIDL。接下来我将就这三种方式展开来讲。

1.2.1，继承Binder类

看到这里可能有些同学会问：Binder又是什么？在这里我并不准备给出一个详尽、确切的答案——因为他太复杂了，深入的讲下去的话必将导致这篇博文失去重心。在这里我们只需要知道，它实现了IBinder接口，通过实现Binder类，我们的客户端可以直接通过这个类调用服务端的公有方法。另外，虽然从IPC的角度来讲，Binder是Android中的一种跨进程通信方式，但是其实一般service里面的Binder是不会涉及进程间通信的，所以其在这种情况下显得较为简单。

下面我们来看下通过继承Binder类实现客户端与服务端通信应该怎样做：

在service类中，创建一个满足以下任一要求的Binder实例：

包含客户端可调用的公共方法

返回当前Service实例，其中包含客户端可调用的公共方法

返回由当前service承载的其他类的实例，其中包含客户端可调用的公共方法

在onBind()方法中返回这个Binder实例

在客户端中通过onServiceDisconnected()方法接收传过去的Binder实例，并通过它提供的方法进行后续操作

可以看到，在使用这种方法进行客户端与服务端之间的交互是需要有一个强制类型转换的——在onServiceDisconnected()中获得一个经过转换的IBinder对象，我们必须将其转换为service类中的Binder实例的类型才能正确的调用其方法。而这强制类型转换其实就隐含了一个使用这种方法的条件：客户端和服务端应当在同一个进程中！不然在类型转换的时候也许会出现问题——在另一个进程中一定有这个Binder实例么？没有的话就不能完成强制类型转换。

下面是一个Google官方的例子：

public class LocalService extends Service {

// Binder given to clients

private final IBinder mBinder = new LocalBinder();

// Random number generator

private final Random mGenerator = new Random();

/\*\*

\* Class used for the client Binder. Because we know this service always

\* runs in the same process as its clients, we don't need to deal with IPC.

\*/

public class LocalBinder extends Binder {

LocalService getService() {

// Return this instance of LocalService so clients can call public methods

return LocalService.this;

}

}

@Override

public IBinder onBind(Intent intent) {

return mBinder;

}

/\*\* method for clients \*/

public int getRandomNumber() {

return mGenerator.nextInt(100);

}

}

LocalBinder 为客户端提供 getService() 方法，以检索 LocalService 的当前实例。这样，客户端便可调用服务中的公共方法。 例如，客户端可调用服务中的 getRandomNumber():

public class BindingActivity extends Activity {

LocalService mService;

boolean mBound = false;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

}

@Override

protected void onStart() {

super.onStart();

// Bind to LocalService

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

bindService(intent, mConnection, Context.BIND\_AUTO\_CREATE);

}

@Override

protected void onStop() {

super.onStop();

// Unbind from the service

if (mBound) {

unbindService(mConnection);

mBound = false;

}

}

/\*\* Called when a button is clicked (the button in the layout file attaches to

\* this method with the android:onClick attribute) \*/

public void onButtonClick(View v) {

if (mBound) {

// Call a method from the LocalService.

// However, if this call were something that might hang, then this request should

// occur in a separate thread to avoid slowing down the activity performance.

int num = mService.getRandomNumber();

Toast.makeText(this, "number: " + num, Toast.LENGTH\_SHORT).show();

}

}

/\*\* Defines callbacks for service binding, passed to bindService() \*/

private ServiceConnection mConnection = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName className,

IBinder service) {

// We've bound to LocalService, cast the IBinder and get LocalService instance

LocalBinder binder = (LocalBinder) service;

mService = binder.getService();

mBound = true;

}

@Override

public void onServiceDisconnected(ComponentName arg0) {

mBound = false;

}

};

}

当Activity进入onStart()状态时，就会尝试与目标service绑定，而当点击按钮时，如果绑定已经完成，就会调用service中的方法getRandomNumber()，并将其输出。

线程内通信基本上就是这样了，没什么复杂的地方。接下来我们看看这种方式启动的service如何进行IPC。

1.2.2，使用Messenger

以前讲到跨进程通信，我们总是第一时间想到AIDL(Android接口定义语言)，实际上，使用Messenger在很多情况下是比使用AIDL简单得多的，具体是为什么下文会有比较。

大家看到Messenger可能会很轻易的联想到Message，然后很自然的进一步联想到Handler——没错，Messenger的核心其实就是Message以及Handler来进行线程间的通信。下面讲一下通过这种方式实现IPC的步骤：

服务端实现一个Handler，由其接受来自客户端的每个调用的回调

使用实现的Handler创建Messenger对象

通过Messenger得到一个IBinder对象，并将其通过onBind()返回给客户端

客户端使用 IBinder 将 Messenger（引用服务的 Handler）实例化，然后使用后者将 Message 对象发送给服务

服务端在其 Handler 中（具体地讲，是在 handleMessage() 方法中）接收每个 Message

用这种方式，客户端并没有像扩展Binder类那样直接调用服务端的方法，而是采用了用Message来传递信息的方式达到交互的目的。接下来是一个简单的例子：

//服务端

public class MessengerServiceDemo extends Service {

static final int MSG\_SAY\_HELLO = 1;

class ServiceHandler extends Handler {

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

case MSG\_SAY\_HELLO:

//当收到客户端的message时，显示hello

Toast.makeText(getApplicationContext(), "hello!", Toast.LENGTH\_SHORT).show();

break;

default:

super.handleMessage(msg);

}

}

}

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

@Nullable

@Override

public IBinder onBind(Intent intent) {

Toast.makeText(getApplicationContext(), "binding", Toast.LENGTH\_SHORT).show();

//返回给客户端一个IBinder实例

return mMessenger.getBinder();

}

}

服务端主要是返给客户端一个IBinder实例，以供服务端构造Messenger，并且处理客户端发送过来的Message。当然，不要忘了要在Manifests文件里面注册：

<service

android:name=".ActivityMessenger"

android:enabled="true"

android:exported="true">

<intent-filter>

<action android:name="com.lypeer.messenger"></action>

<category android:name="android.intent.category.DEFAULT"/>

</intent-filter>

</service>

可以看到，这里注册的就和我们原先注册的有一些区别了，主要是因为我们在这里要跨进程通信，所以在另外一个进程里面并没有我们的service的实例，此时必须要给其他的进程一个标志，这样才能让其他的进程找到我们的service。讲道理，其实这里的android:exported属性不设置也可以的，因为在有intent-filter的情况下这个属性默认就是true

接下来我们看下客户端应当怎样操作：

//客户端

public class ActivityMessenger extends Activity {

static final int MSG\_SAY\_HELLO = 1;

Messenger mService = null;

boolean mBound;

private ServiceConnection mConnection = new ServiceConnection() {

public void onServiceConnected(ComponentName className, IBinder service) {

//接收onBind()传回来的IBinder，并用它构造Messenger

mService = new Messenger(service);

mBound = true;

}

public void onServiceDisconnected(ComponentName className) {

mService = null;

mBound = false;

}

};

//调用此方法时会发送信息给服务端

public void sayHello(View v) {

if (!mBound) return;

//发送一条信息给服务端

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

try {

mService.send(msg);

} catch (RemoteException e) {

e.printStackTrace();

}

}

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

@Override

protected void onStart() {

super.onStart();

//绑定服务端的服务，此处的action是service在Manifests文件里面声明的

Intent intent = new Intent();

intent.setAction("com.lypeer.messenger");

//不要忘记了包名，不写会报错

intent.setPackage("com.lypeer.ipcserver");

bindService(intent, mConnection, Context.BIND\_AUTO\_CREATE);

}

@Override

protected void onStop() {

super.onStop();

// Unbind from the service

if (mBound) {

unbindService(mConnection);

mBound = false;

}

}

}

客户端就主要是发起与服务端的绑定，以及通过onServiceConnected()方法来过去服务端返回来的IBinder，借此构造Messenger，从而可以通过发送Message的方式与服务端进行交互。上面的例子其实并不完整，因为它只有客户端对服务端单方面的通信，而服务端没有发信息给客户端的功能——这显然是不合理的。而要实现这个其实也很简单，只要客户端里也创建一个Handler实例，让它接收来自服务端的信息，同时让服务端在客户端给它发的请求完成了之后再给客户端发送一条信息即可。

用Messenger来进行IPC的话整体的流程是非常清晰的，Message在其中起到了一个信使的作用，通过它客户端与服务端的信息得以互通。

1.2.3，通过AIDL

AIDL，即Android Interface Definition Language，Android接口定义语言。它是一种IDL语言，可以拿来生成用于IPC的代码。在我看来，它其实就是一个模板。为什么这样说呢？在我们的使用中，实际上起作用的并不是我们写的AIDL代码，而是系统根据它生成的一个IInterface实例的代码。而如果大家多生成几个这样的实例，然后把它们拿来比较，你会发现它们都是有套路的——都是一样的流程，一样的结构，只是根据具体的AIDL文件的不同有细微的变动。所以其实AIDL就是为了避免我们一遍遍的写一些千篇一律的代码而出现的一个模板。

那么如何使用AIDL来通过bindService()进行线程间通信呢？基本上有下面这些步骤：

服务端创建一个AIDL文件，将暴露给客户端的接口在里面声明

在service中实现这些接口

客户端绑定服务端，并将onServiceConnected()得到的IBinder转为AIDL生成的IInterface实例

通过得到的实例调用其暴露的方法

上面的描述其实比较抽象，基本上是那种看了也不知道怎么做的类型——这个如果要展开讲的话就又是长篇大论的了。基于这种考虑，这里只是简单的介绍一下AIDL这个东西，它的具体的语法，到底怎么来实现IPC，我会单独写一篇博客来叙述这方面的东西。

1.2.4，Messenger与AIDL的比较

首先，在实现的难度上，肯定是Messenger要简单的多——至少不需要写AIDL文件了(虽然如果认真的究其本质，会发现它的底层实现还是AIDL)。另外，使用Messenger还有一个显著的好处是它会把所有的请求排入队列，因此你几乎可以不用担心多线程可能会带来的问题。

但是这样说来，难道AIDL进行IPC就一无是处了么？当然不是，如果是那样的话它早就被淘汰了。一方面是如果项目中有并发处理问题的需求，或者会有大量的并发请求，这个时候Messenger就不适用了——它的特性让它只能串行的解决请求。另外，我们在使用Messenger的时候只能通过Message来传递信息实现交互，但是在有些时候也许我们需要直接跨进程调用服务端的方法，这个时候又怎么办呢？只能使用AIDL。

所以，这两种IPC方式各有各的优点和缺点，具体使用哪种就看具体的需要了——当然，能使用简单的就尽量使用简单的吧。

1.2.5，service的生命周期

当服务与所有客户端之间的绑定全部取消时，Android 系统便会销毁这个服务（除非还使用 onStartCommand() 启动了该服务）。因此，如果服务是纯粹的绑定服务，原则上我们是无需对其生命周期进行管理的—Android 系统会根据它是否绑定到任何客户端帮我们管理。但实际上，我们应该始终在完成与服务的交互时或 Activity 暂停时取消绑定，以便服务能够在未被占用时关闭。

如果你只需要在 Activity 可见时与服务交互，则可以在 onStart() 期间绑定，在 onStop() 期间取消绑定。如果你希望 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。但是注意，这意味着你的 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，那么当你提高该进程的权重时，系统终止该进程的可能性会增加。通常情况下，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，我们应该使发生在这些转换期间的处理保持在最低水平。此外，如果我们的应用内的多个 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。

此外，如果我们的服务已启动并接受绑定，则当系统调用 onUnbind() 方法时，如果我们想在客户端下一次绑定到服务时接收 onRebind() 调用（而不是接收 onBind() 调用），则可选择返回 true。onRebind() 返回空值，但客户端仍在其 onServiceConnected() 回调中接收 IBinder。下图说明了这种生命周期的逻辑：

使用bindService()启动的service的生命周期

2，什么时候用startService什么时候用bindService?

这个其实可以通过它们的特点很轻松的得到结论：它们之间的主要区别其实体现在两点，能否交互，以及生命周期。所以很显然的，startService适合那种启动之后不显式停止它就永远在后台运行，并且不需要客户端与服务端交互的service。比方说一条专门拿来存数据到本地数据库的service，它就一直在后台等着有别的组件startService，然后把拿到的数据存入数据库，这就显然是用startService做的事情。而bindService呢，就适合那种可以交互的，可以掌控它什么时候停什么时候开始的。另外，如果有IPC的需求，那当然bindService是必不可少的了。

我们在上一篇博文里讲过，其实在大多数情况下，startService和bindService都是相辅相成的，它们并不是孤立的存在。比方说我这个时候要做一个音乐播放器，那么后台播放是肯定要的吧？总不能手机一熄屏音乐也没了。另外，控制音乐也是要的吧？什么上一首下一首播放暂停什么的。这不就强强联合了么？当然要注意的是，在这两种启动方式同时存在去启动一个service的时候，service的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。附上一张图：

两种方式的生命周期

结语

有关“Android中的Service”的博文到这里就差不多结束了。这两篇基本上还是比较完整的介绍了service的方方面面的东西，但是也仅限于介绍了——里面的一些点如果讲的话就太过于深入了，这样很容易导致博文失去重心，这是我回顾以前写的一些博文得到的经验。以前有些文章，尤其是源码解析方面的，我喜欢寻根溯源刨根问底，所以会一直顺着源码往下挖掘，直到挖不动为止。这样的话，就我个人而言肯定是能得到很多收获的，但是写成博客之后可能会显得比较的晦涩，因为必须顺着我的思路，跟着我一路挖下去才能理解我的意思——而一篇好的博客应当是简洁明了的，能让观者轻松地有所收获的。

关于IBinder，Binder，Messenger，AIDL，IPC，等等等等，我后续会有一批关于它们的专题文章，在里面会有比较详细的讲解，包括源码解析之类的

<https://blog.csdn.net/luoyanglizi/article/details/51594016>

https://www.jianshu.com/p/45e55c38636e