内容目录

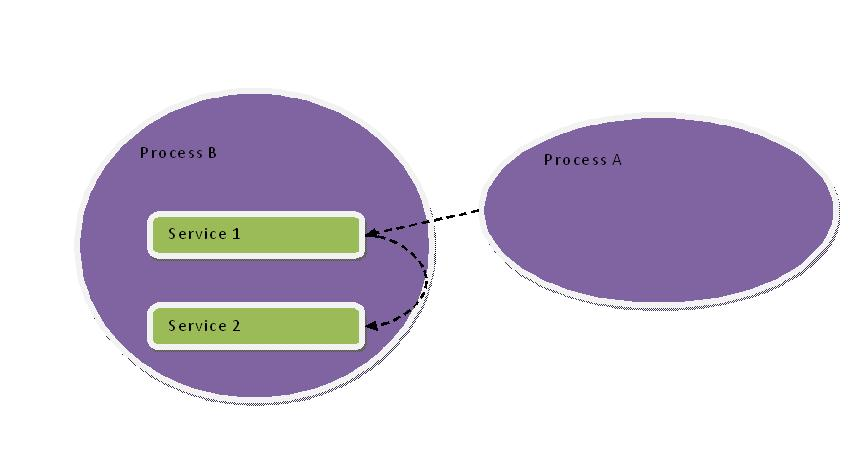
[clearCallingIdentity/restoreCallingIdentity 1](#__RefHeading___Toc7917_695259283)

[binderDied 1](#__RefHeading___Toc7919_695259283)

[oneway 2](#__RefHeading___Toc7921_695259283)

### clearCallingIdentity/restoreCallingIdentity

客户端A调用服务端B的服务，然后B又调用自己的服务或者方法

1. process B在被process A IPC调用时, process B需知道process A的UID和PID，来检查process A的访问权限，此时mCallingUid和mCallingPid保存的是process A的UID和PID。

2. 在IPC远程调用process B的过程中，process B的方法调用了同进程中的service的接口，process B既是调用方也是被调用方，虽然这个过程比较无聊，但是鉴于IPC过程的不透明性，因此process B仍然需要进行权限检测。

在B进程自己调用自己资源的时候可通过如下两个方法

public static final native long clearCallingIdentity();

public static final native void restoreCallingIdentity(long token);

先将A的UID、PID进行备份并且替换B的UID、PID，开始操作B的资源

token = clearCallingIdentity();

调用完成后，再恢复成A的UID、PID

### binderDied

**1. 原理**

当Binder非正常消亡的时候，会导致远程调用失败，这样客户端功能就会受到影响。

解决：给Binder设置一个死亡代理，当Binder死亡时，我们就会收到通知，这个时候可以重新发起连接。

**2. 制作**

**2.1 前期准备**

客户端：MainActivity.java

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

createService();

}

/\*连接Service端，获取mIBookManger\*/

private void createService(){

ServiceConnection connection = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

//初始化mIBookManger

mIBookManager = IBookManager.Stub.asInterface(service);

}

@Override

public void onServiceDisconnected(ComponentName name) {

}

};

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

bindService(intent,connection,BIND\_AUTO\_CREATE);

}

Service端：BookService.java

**2.2使用**

创建IBinder.DeathRecipient接口，重写其中的binderDied()，当Binder死亡时候，回调该方法。

private IBinder.DeathRecipient mDeathRecipient = new IBinder.DeathRecipient() {

@Override

public void binderDied() {

if (mIBookManager != null){

//解除绑定当前接口

mIBookManager.asBinder().unlinkToDeath(mDeathRecipient,0);

}

mIBookManager = null;

createService();

}

};

当然解除绑定之后，还需要绑定接口

/\*连接Service端，获取mIBookManger\*/

private void createService(){

...(连接之前的代码)

try {

//绑定接口,等待回调

mIBookManager.asBinder().linkToDeath(mDeathRecipient,0);

} catch (RemoteException e) {

e.printStackTrace();

}

}

### oneway

oneway 关键字用于修改远程调用的行为。

1.本地调用

如果 oneway 用于本地调用，则不会有任何影响，调用仍是同步调用。

2.远程调用

使用该关键字时，远程调用不会阻塞；它只是发送事务数据并立即返回。接口的实现最终接收此调用时，是以正常远程调用形式将其作为来自 Binder 线程池的常规调用进行接收。

注意，IBinder接口类中定义了一个名为FLAG\_ONEWAY的整型变量，该变量的意义非常重要。当客户端利用Binder机制发起一个跨进程的函数调用时，调用方（即客户端）一般会阻塞，直到服务端返回结果。这种方式和普通的函数调用是一样的。但是在调用Binder函数时，如果指明了FLAG\_ONEWAY标志，则调用方只要把请求发送到Binder驱动即可返回，而不用等待服务端的结果，这就是一种所谓的非阻塞方式。在Native层中，涉及的Binder调用基本都是阻塞的，但是在Java层的framework中，使用FLAG\_ONEWAY进行Binder调用的情况非常多，以后经常会碰到。

思考：使用FLAG\_ONEWAY进行函数调用的程序在设计上有什么特点？这里简单分析一下：对于使用FLAG\_ONEWAY的函数来说，客户端仅向服务端发出了请求，但是并不能确定服务端是否处理了该请求。所以，客户端一般会向服务端注册一个回调（同样是跨进程的Binder调用），一旦服务端处理了该请求，就会调用此回调函数来通知客户端处理结果。当然，这种回调函数也大多采用FLAG\_ONEWAY的方式。

# binder调用 overview

## 框架分析

frameworks/native/cmds/servicemanager/bctest.c

注册服务的过程:

a. binder\_open

b. binder\_call(bs, &msg, &reply, 0, SVC\_MGR\_ADD\_SERVICE)

// 含有服务的名字

// 它会含有servicemanager回复的数据

// 0表示servicemanager

// code: 表示要调用servicemanager中的"addservice函数"

获取服务的过程:

a. binder\_open

b. binder\_call(bs, &msg, &reply, target, SVC\_MGR\_CHECK\_SERVICE)

// 含有服务的名字

// 它会含有servicemanager回复的数据, 表示提供服务的进程

// 0表示servicemanager

// code: 表示要调用servicemanager中的"getservice函数"

Binder系统\_C程序框架分析了binder\_call函数对binder\_io的解析处理，交给binder\_write\_read的write\_buffer/read\_buffer。

## 编写程序

服务端：test\_server.c

客服端：test\_client.c