# TODO

本地服务和远程服务的差别：不是服务都需要注册，注册到ServiceManager的服务只不过是为了方便Client去查询并获得它的代理接口。如果你有其它途径去获得一个服务的代理接口，就完全不需要注册

MS和Service差别？binder线程PK进程？

Service一定是独立进程么？

用户定义的服务什么时候加入到servicemanager中呢？还是说自己主动加入到servicemanager？源码内没有自动加入吗？

Q:1、通过startService启动的服务与activity是否既可在一个进程中，也可不在一个进程中？ 2、通过bindService启动的服务与activity是否必定在一个进程中？

A: 在manifest文件里加了remote属性），而service也可能在当前进程里运行，如果是这种情况，就不需要新建个进程了，直接运行 realStartServiceLocked(r, app)就可以

service的启动最终是在ActivityThread类中的service.oncreate()启动.为什么之前在创建新进程之后运行了ActivityThread的main后，(也就是Step 6. ActivityThread.main，)不可以直接运行service.oncreate().而还要通过activitymanageservice再次通信ActivityThread，再调用service.oncreate().？

如果在新进程中运行service，那么新进程中的service如何与主进程中的组件进行通信呢？

Q:请问在service运行在宿主进程而开一个子线程和单独用一个进程运行service，有何优缺点？

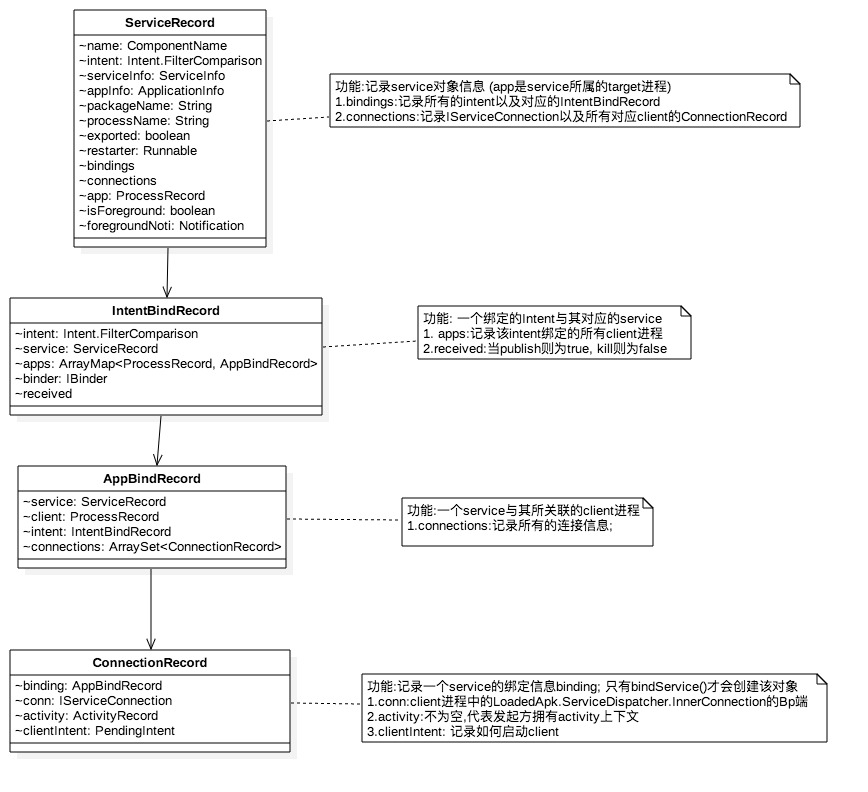
A: 完全是根据你的需要。比如，如果你不想主程序的crash影响到你的service，或者service的crash影响到主程序，就可以选择将service放在一个独立的进程里面, crash杀的单位是进程！

# Service数据结构

每个app进程运行的Service, 对应于system\_server进程中AMS的ServiceRecord对象. 也就是说所有app进程的Service, 都会记录在system\_server进程, 好比一个大管家.

## 类图

先以一幅图来展示AMS管理Service所涉及的相关数据结构： [点击查看大图](http://www.gityuan.com/images/ams/service_record.jpg)



图中，从上至下存在简单的包含关系：

* ServiceRecord可以包含一个或多个IntentBindRecord；
* IntentBindRecord可以包含一个或多个AppBindRecord；
* AppBindRecord可以包含一个或多个ConnectionRecord；

### 2.1 ServiceRecord

ServiceRecord位于system\_server进程，是AMS管理各个app中service的基本单位。 ServiceRecord继承于Binder对象,作为Binder IPC的Bn端；Binder将其传递到Service进程的Bp端， 保存在Service.mToken, 即ServiceRecord的代理对象。

ServiceRecord的成员变量app，用于记录当前service运行在哪个进程。再来说说其他成员变量：

（1）bindings：由于可通过不同的Intent来bind同一个service，故采用bindings来记录所有的intent以及对应的IntentBindRecord ，其数据类型如下：

ArrayMap<Intent.FilterComparison, IntentBindRecord> bindings

（2）connections:记录IServiceConnection以及所有对应client的ConnectionRecord

（3）startRequested：代表service是否由startService方式所启动的

* startService()，则startRequested=true;
* stopService()或stopSelf()，则startRequested=false;

### 2.2 IntentBindRecord

功能: 一个绑定的Intent与其对应的service

（1）apps: 由于同一个intent，可能有多个不同的进程来bind该service，故采用其成员变量aPps来记录该intent绑定的所有client进程， 其数据类型如下：

ArrayMap<ProcessRecord, AppBindRecord> apps

（2）binder: service发布过程，调用publishServiceLocked()来赋值的IBinder对象；也就是bindService后的onBinder()方法 的返回值（作target进程的binder服务）的代理对象。简单来说就是onServiceConnected()的第二个参数。

（3）received：

* 当执行完publishServiceLocked(), 则received=true; requested=true;
* 当执行killServicesLocked(), 则received=false; requested=false;

### 2.3 AppBindRecord

功能:一个service与其所关联的client进程

其成员变量connections记录所有的连接信息;

### 2.4 ConnectionRecord

功能:记录一个service的绑定信息binding; 只有bindService()才会创建该对象。

1. conn:client进程中的LoadedApk.ServiceDispatcher.InnerConnection的Bp端
2. activity:不为空,代表发起方拥有activity上下文
3. clientIntent: 记录如何启动client

执行bindService()便会创建ConnectionRecord对象, 该对象创建后添加到以下对象的列表:

* ServiceRecord.connections: 记录在ServiceRecord的成员变量;
* ActiveServices.mServiceConnections: 效果同上;
* AppBindRecord.connections: 某个service的所有绑定信息会记录在该对象;
* ProcessRecord.connections: 记录在service所对应的client进程
* ActivityRecord.connections: 当bindService()过程是由activity context所启动, 则会添加到该对表;否则跳过;

当bindService过程中指定参数flags Context.BIND\_AUTO\_CREATE, 则会在bind过程创建service;

### 2.5 LoadedApk



LoadedApk对象，往往运行在client进程，通过其成员变量mServices来记录相关的service信息。不同的Context下，对于同一个 ServiceConnection会对应唯一的LoadedApk.ServiceDispatcher对象。ServiceDispatcher用于服务分发，即服务连接或死亡事件的派发。

bindService过程并不是直接把ServiceConnection(只是Interface)传递到AMS，而是创建一个InnerConnection(Binder对象)再传递到AMS，这样便于跨进程间交互。

## 启动/销毁API

对Service的主要操作如下四个方法:

startService(Intent service)

stopService(Intent service)

bindService(Intent service, ServiceConnection conn, int flags)

unbindService(ServiceConnection conn)

其中:

public interface ServiceConnection {

public void onServiceConnected(ComponentName name, IBinder service);

public void onServiceDisconnected(ComponentName name);

}

Service的启动方式主要有startService和bindService的方式. 说明:在这里把执行启动这个动作的一方称为client, 把被启动的陈伟target service.

### (1) startService方式/stopService

* client通过startService()启动target服务;
* target通过覆写onStartCommand可以执行具体的业务逻辑;
* client通过stopService()或者service自身通过stopself()来结束服务;
  + 如果存在多个client启动同一个service, 只需一个client便可以stop该服务;

### (2) bindService方式:

* client通过bindService()启动target服务;
* target通过覆写onBinder将其IBinder对象将其返回给client端;
  + client端通过ServiceConnection的onServiceConnected获取IBinder的代理对象, 便可通过Binder IPC直接操作service的相应业务方法;
* client通过unbindService()来结束服务的连接关系;
  + 多次bind，则需要等量的unbind操作；

#### (3) unbindService

前面介绍的bindService()过程会创建与bind服务，对于unbindService则正好是逆过程unbind和摧毁service。

unbindService(ServiceConnection conn)

说明：

1. unbindService()过程在client进程是对同一个ServiceConnection的bind来执行unbind操作， 从该方法参数也能印证其功能；这里需要注意当采用同一个ServiceConnection执行了bind操作，只需一次unbind就把所有 使用同一个ServiceConnection的bind都一并执行了unbind操作；
2. 站在AMS角度来看，unbindService()，只有当某个service Intent的IntentBindRecord.apps的个数为0时才执行unbind操作。 也就是说当存在多个进程采用同一个intent bind某个service，那么必须等到所有的进程都执行了unbind操作，才能真正unbind。
3. 当service是通过startService方式所启动，那么必须通过stopService或者stopSelf()才能destroy该服务，任何的unbindService 是无法destroy该服务；
4. 当servce时通过bindService带有flags Context.BIND\_AUTO\_CREATE方式启动的，那么unbind过程会判断如果该service的其他ConnectionRecord都不存在设置BIND\_AUTO\_CREATE，则会直接destroy该service；如果存在一个，则不会destroy。可见，决定是否destroy service的过程，是否存在其他的不带BIND\_AUTO\_CREATE的bind service完全忽略

## 启动/销毁流程简述

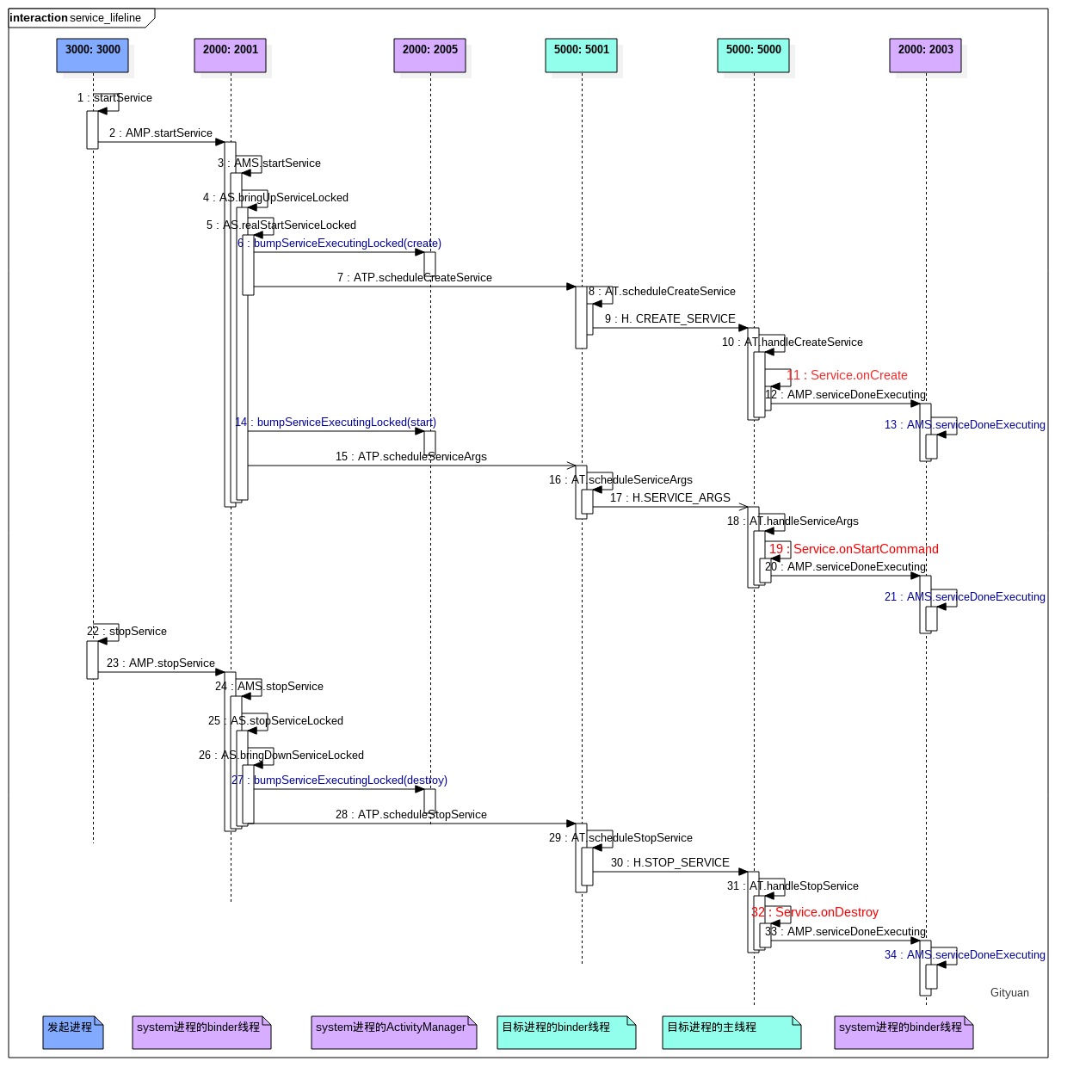
Df

### startService

[startService](http://gityuan.com/2016/03/06/start-service/)的生命周期:

* onCreate
* onStartCommand
* onDestroy

启动流程图, [点击查看大图](http://www.gityuan.com/images/ams/service_lifeline.jpg)

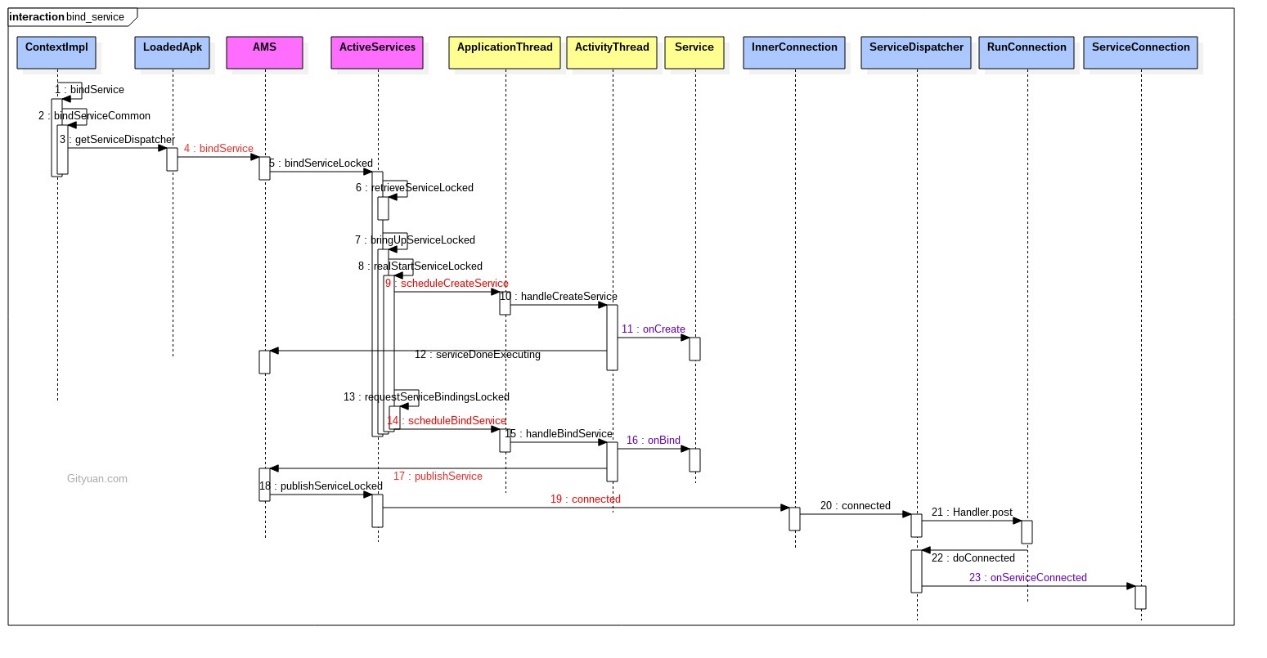


### 3.2.2 bindService

[bindService](http://gityuan.com/2016/05/01/bind-service/)的生命周期:

* onCreate
* onBind
* onUnbind
* onDestroy

启动流程图, [点击查看大图](http://www.gityuan.com/images/ams/bind_service.jpg)



说明：

1. 图中蓝色代表的是Client进程(发起端), 红色代表的是system\_server进程, 黄色代表的是target进程(service所在进程);
2. Client进程: 通过getServiceDispatcher获取Client进程的匿名Binder服务端，即LoadedApk.ServiceDispatcher.InnerConnection,该对象继承于IServiceConnection.Stub； 再通过bindService调用到system\_server进程;
3. system\_server进程: 依次通过scheduleCreateService和scheduleBindService方法, 远程调用到target进程; 4: target进程: 依次执行onCreate()和onBind()方法; 将onBind()方法的返回值IBinder(作为target进程的binder服务端)通过publishService传递到system\_server进程;
4. system\_server进程: 利用IServiceConnection代理对象向Client进程发起connected()调用, 并把target进程的onBind返回Binder对象的代理端传递到Client进程;
5. Client进程: 回调到onServiceConnection()方法, 该方法的第二个参数便是target进程的binder代理端. 到此便成功地拿到了target进程的代理, 可以畅通无阻地进行交互.

另外，说明bindService过程中只有指定参数flags Context.BIND\_AUTO\_CREATE, 才会在bind过程创建/销毁service，即BringUpServiceLocked和bringDownServiceIfNeededLocked过程；

### Foreground

在系统内存资源不足的情况下，会优先杀掉低优先级的进程。对于只有service的进程，有些情况对于用户来说是可感知的app。往往不希望被系统所杀，比如正在听音乐。系统默认service都是后台的，可以通过startForeground()把该service提升到foreground优先级，那么adj便会成为PERCEPTIBLE\_APP\_ADJ(可感知的级别)，这个级别的app一般不会轻易被杀。当用户不再听歌时，应该主动调用stopForeground()，将其优先级还原到后台，为用户提供更好的体验，相关API位于文件Service.java ：

startForeground(int id, Notification notification)

stopForeground(boolean removeNotification)

以上两个方法进入AMS都是调用setServiceForeground()；对于startForeground()，往往通过postNotification()来展示 一个通知；对于stopForeground()，当removeNotification=true,则通过 cancelNotification()来取消通知。

作为前台service，必须要有一个status bar的通知，并且通知不会消失直到service停止或许主动移除前台优先级

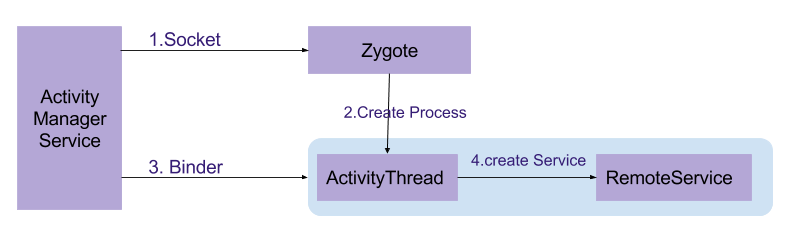
# startService详述

在主进程调用startService函数时，会通过Binder进程间通信机制来通知ActivitManagerService来创建新进程，并且启动指定的服务

在app中启动一个service，就一行语句搞定，

startService()； //或 binderService()

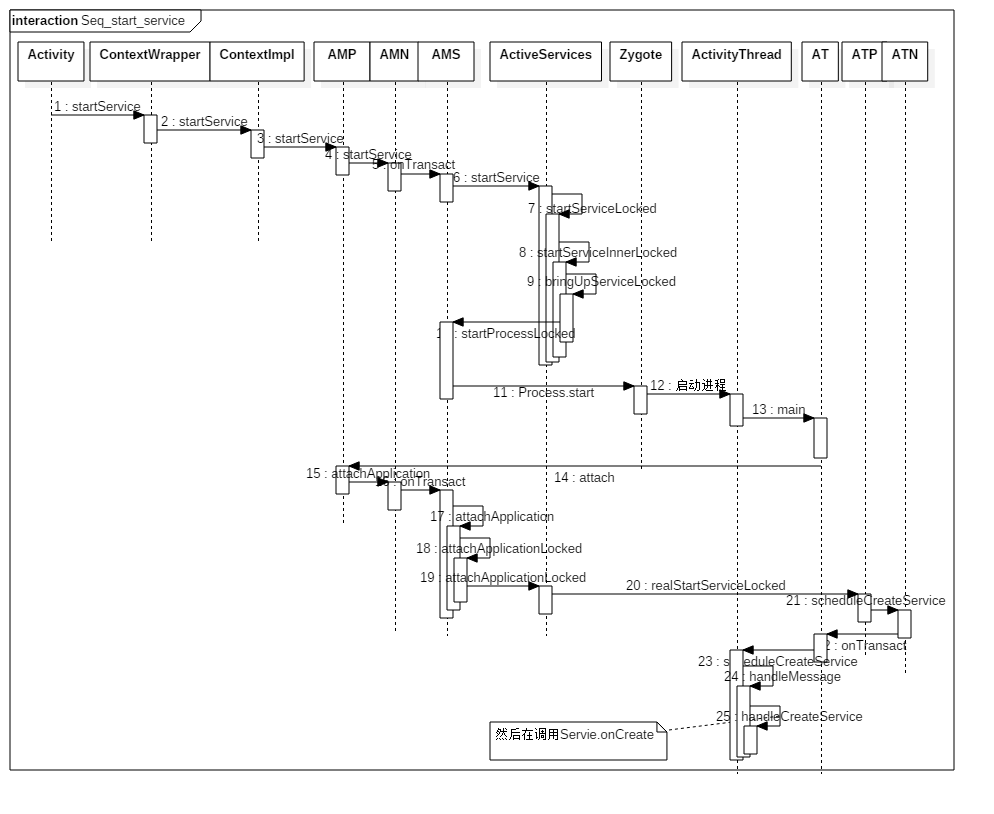
该过程如下：



当App通过调用Android API方法startService()或binderService()来生成并启动服务的过程，主要是由ActivityManagerService来完成的。

1. ActivityManagerService通过Socket通信方式向Zygote进程请求生成(fork)用于承载服务的进程ActivityThread。**此处讲述启动远程服务的过程，即服务运行于单独的进程中**，对于运行本地服务则不需要启动服务的过程。ActivityThread是应用程序的主线程；
2. Zygote通过fork的方法，将zygote进程复制生成新的进程，并将ActivityThread相关的资源加载到新进程；
3. ActivityManagerService向新生成的ActivityThread进程，通过Binder方式发送生成服务的请求；
4. ActivityThread启动运行服务，这便于服务启动的简易过程

## **流程图**



图中涉及的首字母缩写：

* AMP:ActivityManagerProxy
* AMN:ActivityManagerNative
* AMS:ActivityManagerService
* AT:ApplicationThread
* ATP:ApplicationThreadProxy
* ATN:ApplicationThreadNative

接下来，我们正式从代码角度来分析服务启动的过程。首先在我们应用程序的Activity类的调用startService()方法，该方法调用【流程1】的方法。

## 二. 发起进程端

### 1. CW.startService

[-> ContextWrapper.java]

public class ContextWrapper extends Context {

public ComponentName startService(Intent service) {

return mBase.startService(service); //其中mBase为ContextImpl对象 【见流程2】

}

}

### 2. CI.startService

[-> ContextImpl.java]

class ContextImpl extends Context {

@Override

public ComponentName startService(Intent service) {

//当system进程调用此方法时输出warn信息，system进程建立调用startServiceAsUser方法

warnIfCallingFromSystemProcess();

return startServiceCommon(service, mUser); //【见流程3】

}

### 3. CI.startServiceCommon

[-> ContextImpl.java]

private ComponentName startServiceCommon(Intent service, UserHandle user) {

try {

//检验service，当service为空则throw异常

validateServiceIntent(service);

service.prepareToLeaveProcess();

// 调用ActivityManagerNative类 【见流程3.1以及流程4】

ComponentName cn = ActivityManagerNative.getDefault().startService(

mMainThread.getApplicationThread(), service, service.resolveTypeIfNeeded(getContentResolver()), getOpPackageName(), user.getIdentifier());

if (cn != null) {

if (cn.getPackageName().equals("!")) {

throw new SecurityException("Not allowed to start service " +

service + " without permission " + cn.getClassName());

} else if (cn.getPackageName().equals("!!")) {

throw new SecurityException("Unable to start service " +

service ": " + cn.getClassName());

}

}

return cn;

} catch (RemoteException e) {

throw new RuntimeException("Failure from system", e);

}

}

#### AMN.getDefault

[-> ActivityManagerNative.java]

### AMP.startService

该类位于文件ActivityManagerNative.java

public ComponentName startService(IApplicationThread caller, Intent service, String resolvedType, String callingPackage, int userId) throws RemoteException {

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IActivityManager.descriptor);

data.writeStrongBinder(caller != null ? caller.asBinder() : null);

service.writeToParcel(data, 0);

data.writeString(resolvedType);

data.writeString(callingPackage);

data.writeInt(userId);

//通过Binder 传递数据　【见流程5】

mRemote.transact(START\_SERVICE\_TRANSACTION, data, reply, 0);

reply.readException();

ComponentName res = ComponentName.readFromParcel(reply);

data.recycle();

reply.recycle();

return res;

}

mRemote.transact()是binder通信的客户端发起方法，经过binder驱动，最后回到binder服务端ActivityManagerNative的onTransact()方法

## system\_server端

借助于AMP/AMN这对Binder对象，便完成了从发起端所在进程到system\_server的调用过程

### 5. AMN.onTransact

@Override

public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

switch (code) {

...

case START\_SERVICE\_TRANSACTION: {

data.enforceInterface(IActivityManager.descriptor);

IBinder b = data.readStrongBinder();

//生成ApplicationThreadNative的代理对象，即ApplicationThreadProxy对象

IApplicationThread app = ApplicationThreadNative.asInterface(b);

Intent service = Intent.CREATOR.createFromParcel(data);

String resolvedType = data.readString();

String callingPackage = data.readString();

int userId = data.readInt();

//调用ActivityManagerService的startService()方法【见流程6】

ComponentName cn = startService(app, service, resolvedType, callingPackage, userId);

reply.writeNoException();

ComponentName.writeToParcel(cn, reply);

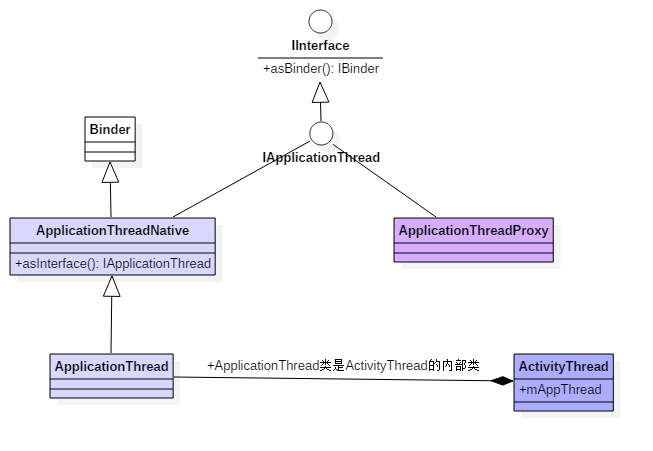
return true;

}

}

在整个调用过程涉及两个进程，不妨令startService的发起进程记为进程A，ServiceManagerService记为进程B；那么进程A通过Binder机制（采用IActivityManager接口）向进程B发起请求服务，进程B则通过Binder机制(采用IApplicationThread接口)向进程A发起请求服务。也就是说进程A与进程B能相互间主动发起请求，进程通信。

这里涉及IApplicationThread，那么下面直接把其相关的类图展示如下：



与IActivityManager的binder通信原理一样，ApplicationThreadProxy作为binder通信的客户端，ApplicationThreadNative作为Binder通信的服务端，其中ApplicationThread继承ApplicationThreadNative类，覆写其中的部分方法。

### 6. AMS.startService

@Override

public ComponentName startService(IApplicationThread caller, Intent service, String resolvedType, String callingPackage, int userId) throws TransactionTooLargeException {

//当调用者是孤立进程，则抛出异常。

enforceNotIsolatedCaller("startService");

if (service != null && service.hasFileDescriptors() == true) {

throw new IllegalArgumentException("File descriptors passed in Intent");

}

if (callingPackage == null) {

throw new IllegalArgumentException("callingPackage cannot be null");

}

if (DEBUG\_SERVICE) Slog.v(TAG\_SERVICE,

"startService: " + service + " type=" + resolvedType);

synchronized(this) {

final int callingPid = Binder.getCallingPid(); //调用者pid

final int callingUid = Binder.getCallingUid(); //调用者uid

final long origId = Binder.clearCallingIdentity();

//此次的mServices为ActiveServices对象 【见流程7】

ComponentName res = mServices.startServiceLocked(caller, service,

resolvedType, callingPid, callingUid, callingPackage, userId);

Binder.restoreCallingIdentity(origId);

return res;

}

}

该方法参数说明：

* caller：IApplicationThread类型，复杂处理
* service：Intent类型，包含需要运行的service信息
* resolvedType：String类型
* callingPackage: String类型，调用该方法的package
* userId: int类型，用户的id

### 7. AS.startServiceLocked

[-> ActiveServices.java]

函数首先通过retrieveServiceLocked来解析service这个Intent，就是解析前面我们在AndroidManifest.xml定义的Service标签的intent-filter相关内容，然后将解析结果放在res.record中，然后继续调用bringUpServiceLocked进一步处理。

ComponentName startServiceLocked(IApplicationThread caller, Intent service, String resolvedType, int callingPid, int callingUid, String callingPackage, int userId) throws TransactionTooLargeException {

final boolean callerFg;

if (caller != null) {

final ProcessRecord callerApp = mAm.getRecordForAppLocked(caller);

if (callerApp == null)

throw new SecurityException(""); //抛出异常，此处省略异常字符串

callerFg = callerApp.setSchedGroup != Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;

} else {

callerFg = true;

}

//检索服务信息

ServiceLookupResult res = retrieveServiceLocked(service, resolvedType, callingPackage,

callingPid, callingUid, userId, true, callerFg);

if (res == null) {

return null;

}

if (res.record == null) {

return new ComponentName("!", res.permission != null

? res.permission : "private to package");

}

ServiceRecord r = res.record;

if (!mAm.getUserManagerLocked().exists(r.userId)) { //检查是否存在启动服务的user

return null;

}

NeededUriGrants neededGrants = mAm.checkGrantUriPermissionFromIntentLocked(

callingUid, r.packageName, service, service.getFlags(), null, r.userId);

r.lastActivity = SystemClock.uptimeMillis();

r.startRequested = true;

r.delayedStop = false;

r.pendingStarts.add(new ServiceRecord.StartItem(r, false, r.makeNextStartId(),

service, neededGrants));

final ServiceMap smap = getServiceMap(r.userId);

boolean addToStarting = false;

//对于非前台进程的调度

if (!callerFg && r.app == null && mAm.mStartedUsers.get(r.userId) != null) {

ProcessRecord proc = mAm.getProcessRecordLocked(r.processName, r.appInfo.uid, false);

if (proc == null || proc.curProcState > ActivityManager.PROCESS\_STATE\_RECEIVER) {

if (r.delayed) { //已计划延迟启动

return r.name;

}

if (smap.mStartingBackground.size() >= mMaxStartingBackground) {

//当超出 同一时间允许后续启动的最大服务数，则将该服务加入延迟启动的队列。

smap.mDelayedStartList.add(r);

r.delayed = true;

return r.name;

}

addToStarting = true;

} else if (proc.curProcState >= ActivityManager.PROCESS\_STATE\_SERVICE) {

//将新的服务加入到后台启动队列，该队列也包含当前正在运行其他services或者receivers的进程

addToStarting = true;

}

}

//【见流程8】

return startServiceInnerLocked(smap, service, r, callerFg, addToStarting);

}

有一种重要的标记符callerFg, 用于标记是前台还是后台:

* 当发起方进程不等于Process.THREAD\_GROUP\_BG\_NONINTERACTIVE,或者发起方为空, 则callerFg= true;
* 否则,callerFg= false;

### 8. AS.startServiceInnerLocked

[-> ActiveServices.java]

ComponentName startServiceInnerLocked(ServiceMap smap, Intent service, ServiceRecord r, boolean callerFg, boolean addToStarting) throws TransactionTooLargeException {

ProcessStats.ServiceState stracker = r.getTracker();

if (stracker != null) {

stracker.setStarted(true, mAm.mProcessStats.getMemFactorLocked(), r.lastActivity);

}

r.callStart = false;

synchronized (r.stats.getBatteryStats()) {

r.stats.startRunningLocked(); //用于耗电统计，开启运行的状态

}

//【见流程9】

String error = bringUpServiceLocked(r, service.getFlags(), callerFg, false);

if (error != null) {

return new ComponentName("!!", error);

}

if (r.startRequested && addToStarting) {

boolean first = smap.mStartingBackground.size() == 0;

smap.mStartingBackground.add(r);

r.startingBgTimeout = SystemClock.uptimeMillis() + BG\_START\_TIMEOUT;

if (first) {

smap.rescheduleDelayedStarts();

}

} else if (callerFg) {

smap.ensureNotStartingBackground(r);

}

return r.name;

}

### 9. AS.bringUpServiceLocked

[-> ActiveServices.java]

这里的appName便是我们前面在AndroidManifest.xml文件定义service标签时指定的android:process属性值了，即“.Server”。

SystemUI就是属于在没有启动进程的时候，利用Service启动Application的，和Application

<**service android:name="SystemUIService"  
 android:exported="true"** />

private final String bringUpServiceLocked(ServiceRecord r, int intentFlags, boolean execInFg, boolean whileRestarting) throws TransactionTooLargeException {

if (r.app != null && r.app.thread != null) {

//调用service.onStartCommand()过程

sendServiceArgsLocked(r, execInFg, false);

return null;

}

if (!whileRestarting && r.restartDelay > 0) {

return null; //等待延迟重启的过程，则直接返回

}

// 启动service前，把service从重启服务队列中移除

if (mRestartingServices.remove(r)) {

r.resetRestartCounter();

clearRestartingIfNeededLocked(r);

}

//service正在启动，将delayed设置为false

if (r.delayed) {

getServiceMap(r.userId).mDelayedStartList.remove(r);

r.delayed = false;

}

//确保拥有该服务的user已经启动，否则停止；

if (mAm.mStartedUsers.get(r.userId) == null) {

String msg = "";

bringDownServiceLocked(r);

return msg;

}

//服务正在启动，设置package停止状态为false

AppGlobals.getPackageManager().setPackageStoppedState(

r.packageName, false, r.userId);

final boolean isolated = (r.serviceInfo.flags&ServiceInfo.FLAG\_ISOLATED\_PROCESS) != 0;

final String procName = r.processName;

ProcessRecord app;

if (!isolated) {

//根据进程名和uid，查询ProcessRecord

app = mAm.getProcessRecordLocked(procName, r.appInfo.uid, false);

if (app != null && app.thread != null) {

try {

app.addPackage(r.appInfo.packageName, r.appInfo.versionCode, mAm.mProcessStats);

// 启动服务 【见流程10】

realStartServiceLocked(r, app, execInFg);

return null;

} catch (TransactionTooLargeException e) {

throw e;

} catch (RemoteException e) {

Slog.w(TAG, "Exception when starting service " + r.shortName, e);

}

}

} else {

app = r.isolatedProc;

}

//对于进程没有启动的情况

if (app == null) {

//启动service所要运行的进程 【见流程9.1】

if ((app=mAm.startProcessLocked(procName, r.appInfo, true, intentFlags,

"service", r.name, false, isolated, false)) == null) {

String msg = ""

bringDownServiceLocked(r); // 进程启动失败

return msg;

}

if (isolated) {

r.isolatedProc = app;

}

}

if (!mPendingServices.contains(r)) {

mPendingServices.add(r);

}

if (r.delayedStop) {

r.delayedStop = false;

if (r.startRequested) {

stopServiceLocked(r); //停止服务

}

}

return null;

}

* 当目标进程已存在，则直接执行realStartServiceLocked()；
* 当目标进程不存在，则先执行[startProcessLocked](http://gityuan.com/2016/10/09/app-process-create-2/)创建进程， 将这个ServiceRecord保存在成员变量mPendingServices列表中。经过层层调用最后会调用到AMS.attachApplicationLocked, 然后再执行realStartServiceLocked()。

对于非前台进程调用而需要启动的服务，如果已经有其他的后台服务正在启动中，那么我们可能希望延迟其启动。这是用来避免启动同时启动过多的进程(非必须的)。

#### 9.1 AMS.attachApplicationLocked

[-> ActivityManagerService.java]

private final boolean attachApplicationLocked(IApplicationThread thread, int pid) {

...

thread.bindApplication(processName, appInfo, providers, app.instrumentationClass,

profilerInfo, app.instrumentationArguments, app.instrumentationWatcher,

app.instrumentationUiAutomationConnection, testMode, enableOpenGlTrace,

isRestrictedBackupMode || !normalMode, app.persistent,

new Configuration(mConfiguration), app.compat,

getCommonServicesLocked(app.isolated),

mCoreSettingsObserver.getCoreSettingsLocked());

...

if (!badApp) {

try {

//寻找所有需要在该进程中运行的服务 【见流程9.2】

didSomething |= mServices.attachApplicationLocked(app, processName);

} catch (Exception e) {

badApp = true;

}

}

...

return true;

}

#### 9.2 AS.attachApplicationLocked

[-> ActiveServices.java]

回忆一下在上面的Step 4中，以新进程的pid值作为key值保存了一个ProcessRecord在mPidsSelfLocked列表中，这里先把它取出来，存放在本地变量app中，并且将app.processName保存在本地变量processName中。

        再回忆一下在上面的Step 3中，在成员变量mPendingServices中，保存了一个ServiceRecord，这里通过进程uid和进程名称将它找出来，然后通过realStartServiceLocked函数来进一步处理。

boolean attachApplicationLocked(ProcessRecord proc, String processName) throws RemoteException {

boolean didSomething = false;

//启动mPendingServices队列中，等待在该进程启动的服务

if (mPendingServices.size() > 0) {

ServiceRecord sr = null;

try {

for (int i=0; i<mPendingServices.size(); i++) {

sr = mPendingServices.get(i);

if (proc != sr.isolatedProc && (proc.uid != sr.appInfo.uid

|| !processName.equals(sr.processName))) {

continue;

}

mPendingServices.remove(i);

i--;

// 将当前服务的包信息加入到proc

proc.addPackage(sr.appInfo.packageName, sr.appInfo.versionCode,

mAm.mProcessStats);

// 启动服务，即将进入服务的生命周期 【见流程10】

realStartServiceLocked(sr, proc, sr.createdFromFg);

didSomething = true;

}

} catch (RemoteException e) {

Slog.w(TAG, "Exception in new application when starting service "

+ sr.shortName, e);

throw e;

}

}

// 对于正在等待重启并需要运行在该进程的服务，现在是启动它们的大好时机

if (mRestartingServices.size() > 0) {

ServiceRecord sr = null;

for (int i=0; i<mRestartingServices.size(); i++) {

sr = mRestartingServices.get(i);

if (proc != sr.isolatedProc && (proc.uid != sr.appInfo.uid

|| !processName.equals(sr.processName))) {

continue;

}

mAm.mHandler.removeCallbacks(sr.restarter);

mAm.mHandler.post(sr.restarter);

}

}

return didSomething;

}

* 当需要创建新进程,则创建后经历过attachApplicationLocked,则会再调用realStartServiceLocked();
* 当不需要创建进程, 即在[流程9]中直接就进入了realStartServiceLocked();

### 10. AS.realStartServiceLocked

[-> ActiveServices.java]

回忆一下在上面的Step 4中，以新进程的pid值作为key值保存了一个ProcessRecord在mPidsSelfLocked列表中，这里先把它取出来，存放在本地变量app中，并且将app.processName保存在本地变量processName中。

        再回忆一下在上面的Step 3中，在成员变量mPendingServices中，保存了一个ServiceRecord，这里通过进程uid和进程名称将它找出来。

这部分的逻辑 和AMS.startActvity类似的

然后通过realStartServiceLocked函数来进一步处理。

章，转载请附上博文链接！

private final void realStartServiceLocked(ServiceRecord r, ProcessRecord app, boolean execInFg) throws RemoteException {

...

r.app = app;

r.restartTime = r.lastActivity = SystemClock.uptimeMillis();

final boolean newService = app.services.add(r);

//发送delay消息【见流程10.1】

bumpServiceExecutingLocked(r, execInFg, "create");

mAm.updateLruProcessLocked(app, false, null);

mAm.updateOomAdjLocked();

boolean created = false;

try {

synchronized (r.stats.getBatteryStats()) {

r.stats.startLaunchedLocked();

}

mAm.ensurePackageDexOpt(r.serviceInfo.packageName);

app.forceProcessStateUpTo(ActivityManager.PROCESS\_STATE\_SERVICE);

//服务进入 onCreate() 【见流程11】

app.thread.scheduleCreateService(r, r.serviceInfo,

mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),

app.repProcState);

r.postNotification();

created = true;

} catch (DeadObjectException e) {

mAm.appDiedLocked(app); //应用死亡处理

throw e;

} finally {

if (!created) {

final boolean inDestroying = mDestroyingServices.contains(r);

serviceDoneExecutingLocked(r, inDestroying, inDestroying);

if (newService) {

app.services.remove(r);

r.app = null;

}

//尝试重新启动服务

if (!inDestroying) {

scheduleServiceRestartLocked(r, false);

}

}

}

requestServiceBindingsLocked(r, execInFg);

updateServiceClientActivitiesLocked(app, null, true);

if (r.startRequested && r.callStart && r.pendingStarts.size() == 0) {

r.pendingStarts.add(new ServiceRecord.StartItem(r, false, r.makeNextStartId(),

null, null));

}

//服务 进入onStartCommand() 【见流程17】

sendServiceArgsLocked(r, execInFg, true);

if (r.delayed) {

getServiceMap(r.userId).mDelayedStartList.remove(r);

r.delayed = false;

}

if (r.delayedStop) {

r.delayedStop = false;

if (r.startRequested) {

stopServiceLocked(r); //停止服务

}

}

}

在bumpServiceExecutingLocked会发送一个延迟处理的消息SERVICE\_TIMEOUT\_MSG。在方法scheduleCreateService执行完成，也就是onCreate回调执行完成之后，便会remove掉该消息。但是如果没能在延时时间之内remove该消息，则会进入执行service timeout流程。

#### 10.1 AS.bumpServiceExecutingLocked

[-> ActiveServices.java]

private final void bumpServiceExecutingLocked(ServiceRecord r, boolean fg, String why) {

long now = SystemClock.uptimeMillis();

if (r.executeNesting == 0) {

r.executeFg = fg;

...

if (r.app != null) {

r.app.executingServices.add(r);

r.app.execServicesFg |= fg;

if (r.app.executingServices.size() == 1) {

scheduleServiceTimeoutLocked(r.app);

}

}

} else if (r.app != null && fg && !r.app.execServicesFg) {

r.app.execServicesFg = true;

//[见流程10.2]

scheduleServiceTimeoutLocked(r.app);

}

r.executeFg |= fg;

r.executeNesting++;

r.executingStart = now;

}

#### 10.2 scheduleServiceTimeoutLocked

void scheduleServiceTimeoutLocked(ProcessRecord proc) {

if (proc.executingServices.size() == 0 || proc.thread == null) {

return;

}

long now = SystemClock.uptimeMillis();

Message msg = mAm.mHandler.obtainMessage(

ActivityManagerService.SERVICE\_TIMEOUT\_MSG);

msg.obj = proc;

//当超时后仍没有remove该SERVICE\_TIMEOUT\_MSG消息，则执行service Timeout流程

mAm.mHandler.sendMessageAtTime(msg,

proc.execServicesFg ? (now+SERVICE\_TIMEOUT) : (now+ SERVICE\_BACKGROUND\_TIMEOUT));

}

发送延时消息SERVICE\_TIMEOUT\_MSG,延时时长：

* 对于前台服务，则超时为SERVICE\_TIMEOUT，即timeout=20s；
* 对于后台服务，则超时为SERVICE\_BACKGROUND\_TIMEOUT，即timeout=200s；

### 11. ATP.scheduleCreateService

[-> ApplicationThreadProxy.java]

public final void scheduleCreateService(IBinder token, ServiceInfo info, CompatibilityInfo compatInfo, int processState) throws RemoteException {

Parcel data = Parcel.obtain();

data.writeInterfaceToken(IApplicationThread.descriptor);

data.writeStrongBinder(token);

info.writeToParcel(data, 0);

compatInfo.writeToParcel(data, 0);

data.writeInt(processState);

try {

//【见流程12】

mRemote.transact(SCHEDULE\_CREATE\_SERVICE\_TRANSACTION, data, null, IBinder.FLAG\_ONEWAY);

} catch (TransactionTooLargeException e) {

throw e;

}

data.recycle();

}

## 目标进程端

借助于ATP/ATN这对Binder对象，便完成了从system\_server所在进程到Service所在进程调用过程

### 12. ATN.onTransact

[-> ApplicationThreadNative.java]

public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

switch (code) {

case SCHEDULE\_CREATE\_SERVICE\_TRANSACTION: {

data.enforceInterface(IApplicationThread.descriptor);

IBinder token = data.readStrongBinder();

ServiceInfo info = ServiceInfo.CREATOR.createFromParcel(data);

CompatibilityInfo compatInfo = CompatibilityInfo.CREATOR.createFromParcel(data);

int processState = data.readInt();

// 【见流程13】

scheduleCreateService(token, info, compatInfo, processState);

return true;

}

...

}

### 13. AT.scheduleCreateService

[-> ApplicationThread.java]

public final void scheduleCreateService(IBinder token, ServiceInfo info, CompatibilityInfo compatInfo, int processState) {

updateProcessState(processState, false);

CreateServiceData s = new CreateServiceData(); //准备服务创建所需的数据

s.token = token;

s.info = info;

s.compatInfo = compatInfo;

//发送消息 【见流程14】

sendMessage(H.CREATE\_SERVICE, s);

}

该方法的执行在ActivityThread线程

#### 14. handleMessage

[-> ActivityThread.java ::H]

public void handleMessage(Message msg) {

switch (msg.what) {

...

case CREATE\_SERVICE:

handleCreateService((CreateServiceData)msg.obj); //【见流程15】

break;

case BIND\_SERVICE:

handleBindService((BindServiceData)msg.obj);

break;

case UNBIND\_SERVICE:

handleUnbindService((BindServiceData)msg.obj);

break;

case SERVICE\_ARGS:

handleServiceArgs((ServiceArgsData)msg.obj); // serviceStart

break;

case STOP\_SERVICE:

handleStopService((IBinder)msg.obj);

maybeSnapshot();

break;

...

}

}

### 15. AT.handleCreateService

[-> ActivityThread.java]

Step 18. ClassLoader.loadClass

        这一步实现在上面的ActivityThread.handleCreateService函数中：

java.lang.ClassLoader cl = packageInfo.getClassLoader();

service = (Service) cl.loadClass(data.info.name).newInstance();

        Step 19. Obtain Service

        这一步也是实现在上面的ActivityThread.handleCreateService函数中。上面通过ClassLoader.loadClass来导入自定义的服务类shy.luo.ashmem.Server并且创建它的一个实例后，就通过强制类型转换得到一个Service类实例。前面我们说过，自己的服务类必须要继承于Service类，这里就体现出来了为什么要这样做了。

请附上博文链接！

private void handleCreateService(CreateServiceData data) {

//当应用处于后台即将进行GC，而此时被调回到活动状态，则跳过本次gc。

unscheduleGcIdler();

LoadedApk packageInfo = getPackageInfoNoCheck(data.info.applicationInfo, data.compatInfo);

java.lang.ClassLoader cl = packageInfo.getClassLoader();

//通过反射创建目标服务对象

Service service = (Service) cl.loadClass(data.info.name).newInstance();

...

try {

//创建ContextImpl对象

ContextImpl context = ContextImpl.createAppContext(this, packageInfo);

context.setOuterContext(service);

//创建Application对象

Application app = packageInfo.makeApplication(false, mInstrumentation);

service.attach(context, this, data.info.name, data.token, app,

ActivityManagerNative.getDefault());

//调用服务onCreate()方法 【见流程15.1】

service.onCreate();

mServices.put(data.token, service);

//调用服务创建完成【见流程16】

ActivityManagerNative.getDefault().serviceDoneExecuting(

data.token, SERVICE\_DONE\_EXECUTING\_ANON, 0, 0);

} catch (Exception e) {

...

}

}

#### 15.1 Service.onCreate

public abstract class Service extends ContextWrapper implements ComponentCallbacks2 {

public void onCreate(){ }

}

最终调用Service.onCreate()方法，对于目标服务都是继承于Service，并覆写该方式，调用目标服务的onCreate()方法。拨云见日，到此总算是进入了Service的生命周期。

### 16 AMS.serviceDoneExecuting

public void serviceDoneExecuting(IBinder token, int type, int startId, int res) {

synchronized(this) {

...

// [见流程16.1]

mServices.serviceDoneExecutingLocked((ServiceRecord)token, type, startId, res);

}

}

由[流程10.1]的bumpServiceExecutingLocked()发送一个延时消息SERVICE\_TIMEOUT\_MSG

#### 16.1 AS.serviceDoneExecutingLocked

[-> ActiveServices.java]

void serviceDoneExecutingLocked(ServiceRecord r, int type, int startId, int res) {

boolean inDestroying = mDestroyingServices.contains(r);

if (r != null) {

...

final long origId = Binder.clearCallingIdentity();

// [见流程16.2]

serviceDoneExecutingLocked(r, inDestroying, inDestroying);

Binder.restoreCallingIdentity(origId);

}

...

}

#### 16.2 serviceDoneExecutingLocked

[-> ActiveServices.java]

private void serviceDoneExecutingLocked(ServiceRecord r, boolean inDestroying, boolean finishing) {

r.executeNesting--;

if (r.executeNesting <= 0) {

if (r.app != null) {

r.app.execServicesFg = false;

r.app.executingServices.remove(r);

if (r.app.executingServices.size() == 0) {

//移除服务启动超时的消息

mAm.mHandler.removeMessages(ActivityManagerService.SERVICE\_TIMEOUT\_MSG, r.app);

} else if (r.executeFg) {

...

}

if (inDestroying) {

mDestroyingServices.remove(r);

r.bindings.clear();

}

mAm.updateOomAdjLocked(r.app);

}

r.executeFg = false;

...

if (finishing) {

if (r.app != null && !r.app.persistent) {

r.app.services.remove(r);

}

r.app = null;

}

}

}

handleCreateService()执行后便会移除服务启动超时的消息SERVICE\_TIMEOUT\_MSG。 Service启动过程出现ANR，”executing service [发送超时serviceRecord信息]”， 这往往是service的onCreate()回调方法执行时间过长。

前面小节[10]realStartServiceLocked方法在完成onCreate操作,解析来便是进入onStartCommand方法. 见下文.

### 17. AS.sendServiceArgsLocked

[-> ActiveServices.java]

private final void sendServiceArgsLocked(ServiceRecord r, boolean execInFg, boolean oomAdjusted) throws TransactionTooLargeException {

final int N = r.pendingStarts.size();

if (N == 0) {

return;

}

while (r.pendingStarts.size() > 0) {

Exception caughtException = null;

ServiceRecord.StartItem si;

try {

si = r.pendingStarts.remove(0);

if (si.intent == null && N > 1) {

continue;

}

si.deliveredTime = SystemClock.uptimeMillis();

r.deliveredStarts.add(si);

si.deliveryCount++;

if (si.neededGrants != null) {

mAm.grantUriPermissionUncheckedFromIntentLocked(si.neededGrants,

si.getUriPermissionsLocked());

}

//标记启动开始【见10.1】

bumpServiceExecutingLocked(r, execInFg, "start");

if (!oomAdjusted) {

oomAdjusted = true;

mAm.updateOomAdjLocked(r.app);

}

int flags = 0;

if (si.deliveryCount > 1) {

flags |= Service.START\_FLAG\_RETRY;

}

if (si.doneExecutingCount > 0) {

flags |= Service.START\_FLAG\_REDELIVERY;

}

//该过程类似[流程11~16]，最终会调用onStartCommand

r.app.thread.scheduleServiceArgs(r, si.taskRemoved, si.id, flags, si.intent);

} catch (Exception e) {

...

caughtException = e;

}

if (caughtException != null) {

final boolean inDestroying = mDestroyingServices.contains(r);

serviceDoneExecutingLocked(r, inDestroying, inDestroying);

if (caughtException instanceof TransactionTooLargeException) {

throw (TransactionTooLargeException)caughtException;

}

break;

}

}

}

[流程10]中的AS.realStartServiceLocked的过程先后依次执行如下方法：

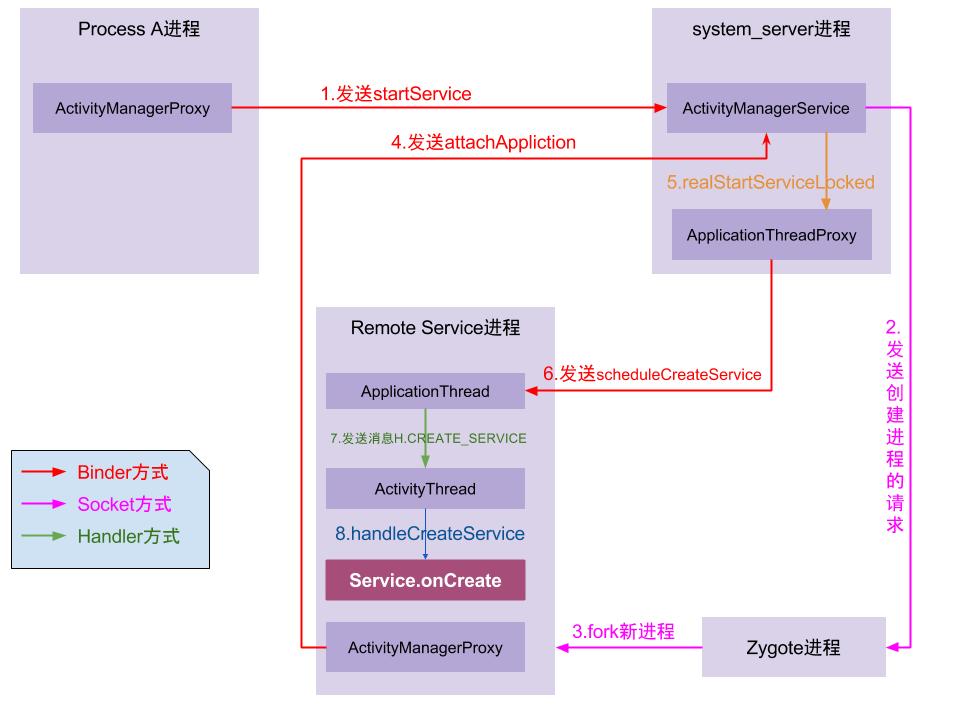
* 执行scheduleCreateService()方法，层层调用最终回调Service.onCreate(); [见流程11~16]
* 执行scheduleServiceArgs()方法，层层调用最终回调Service.onStartCommand(); [见流程17]，这两个过程类似，此处省略。

## 总结

### 5.1 流程说明

在整个startService过程，从进程角度看服务启动过程

* **Process A进程：**是指调用startService命令所在的进程，也就是启动服务的发起端进程，比如点击桌面App图标，此处Process A便是Launcher所在进程。
* **system\_server进程：**系统进程，是java framework框架的核心载体，里面运行了大量的系统服务，比如这里提供ApplicationThreadProxy（简称ATP），ActivityManagerService（简称AMS），这个两个服务都运行在system\_server进程的不同线程中，由于ATP和AMS都是基于IBinder接口，都是binder线程，binder线程的创建与销毁都是由binder驱动来决定的，每个进程binder线程个数的上限为16。
* **Zygote进程：**是由init进程孵化而来的，用于创建Java层进程的母体，所有的Java层进程都是由Zygote进程孵化而来；
* **Remote Service进程：**远程服务所在进程，是由Zygote进程孵化而来的用于运行Remote服务的进程。主线程主要负责Activity/Service等组件的生命周期以及UI相关操作都运行在这个线程； 另外，每个App进程中至少会有两个binder线程 ApplicationThread(简称AT)和ActivityManagerProxy（简称AMP），当然还有其他线程，这里不是重点就不提了。



图中涉及3种IPC通信方式：Binder、Socket以及Handler，在图中分别用3种不同的颜色来代表这3种通信方式。一般来说，同一进程内的线程间通信采用的是 [Handler消息队列机制](http://gityuan.com/2015/12/26/handler-message/)，不同进程间的通信采用的是[binder机制](http://gityuan.com/2015/10/31/binder-prepare/)，另外与Zygote进程通信采用的Socket。

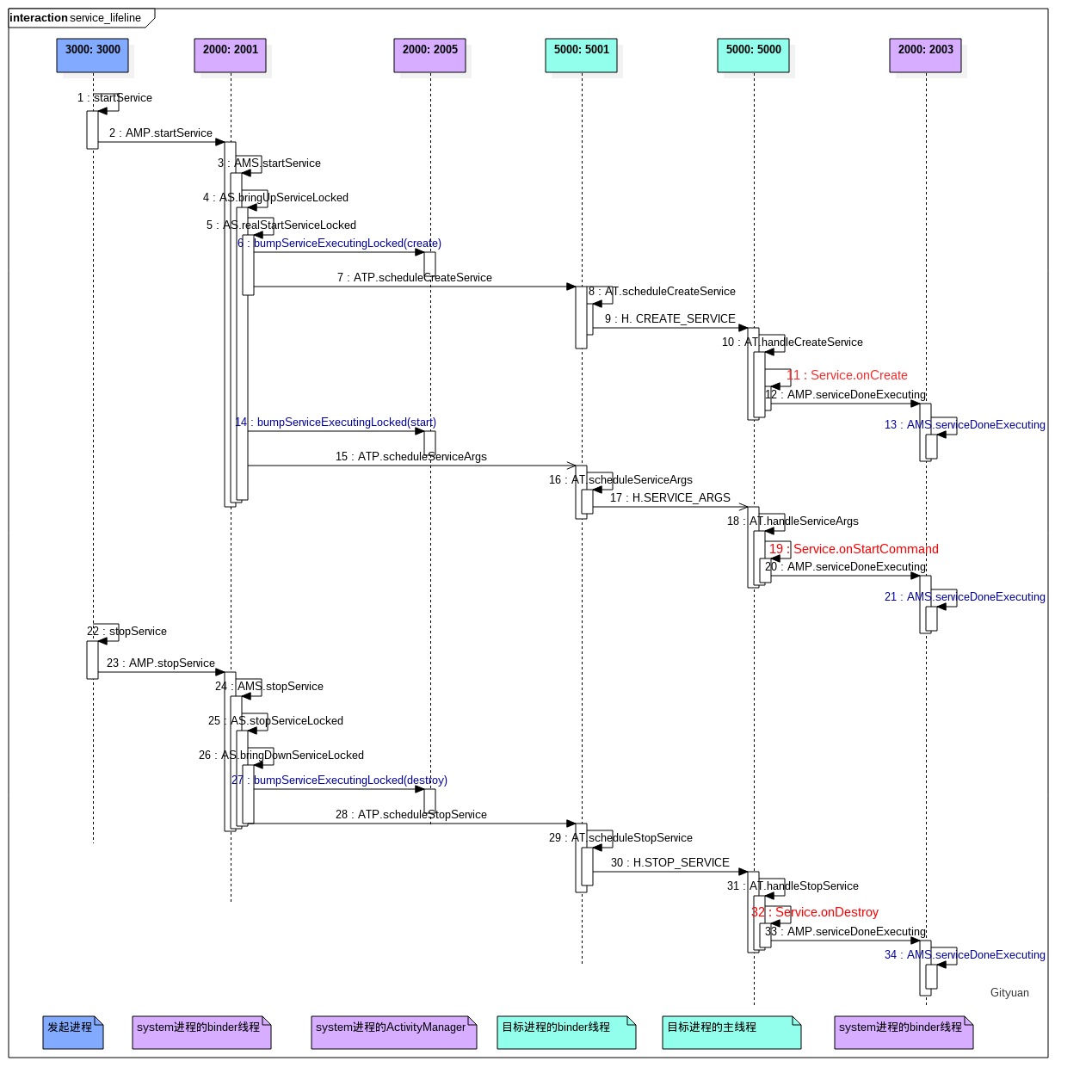
启动流程：

1. Process A进程采用Binder IPC向system\_server进程发起startService请求；
2. system\_server进程接收到请求后，向zygote进程发送创建进程的请求；
3. zygote进程fork出新的子进程Remote Service进程；
4. Remote Service进程，通过Binder IPC向sytem\_server进程发起attachApplication请求；
5. system\_server进程在收到请求后，进行一系列准备工作后，再通过binder IPC向remote Service进程发送scheduleCreateService请求；
6. Remote Service进程的binder线程在收到请求后，通过handler向主线程发送CREATE\_SERVICE消息；
7. 主线程在收到Message后，通过发射机制创建目标Service，并回调Service.onCreate()方法。

到此，服务便正式启动完成。当创建的是本地服务或者服务所属进程已创建时，则无需经过上述步骤2、3，直接创建服务即可。

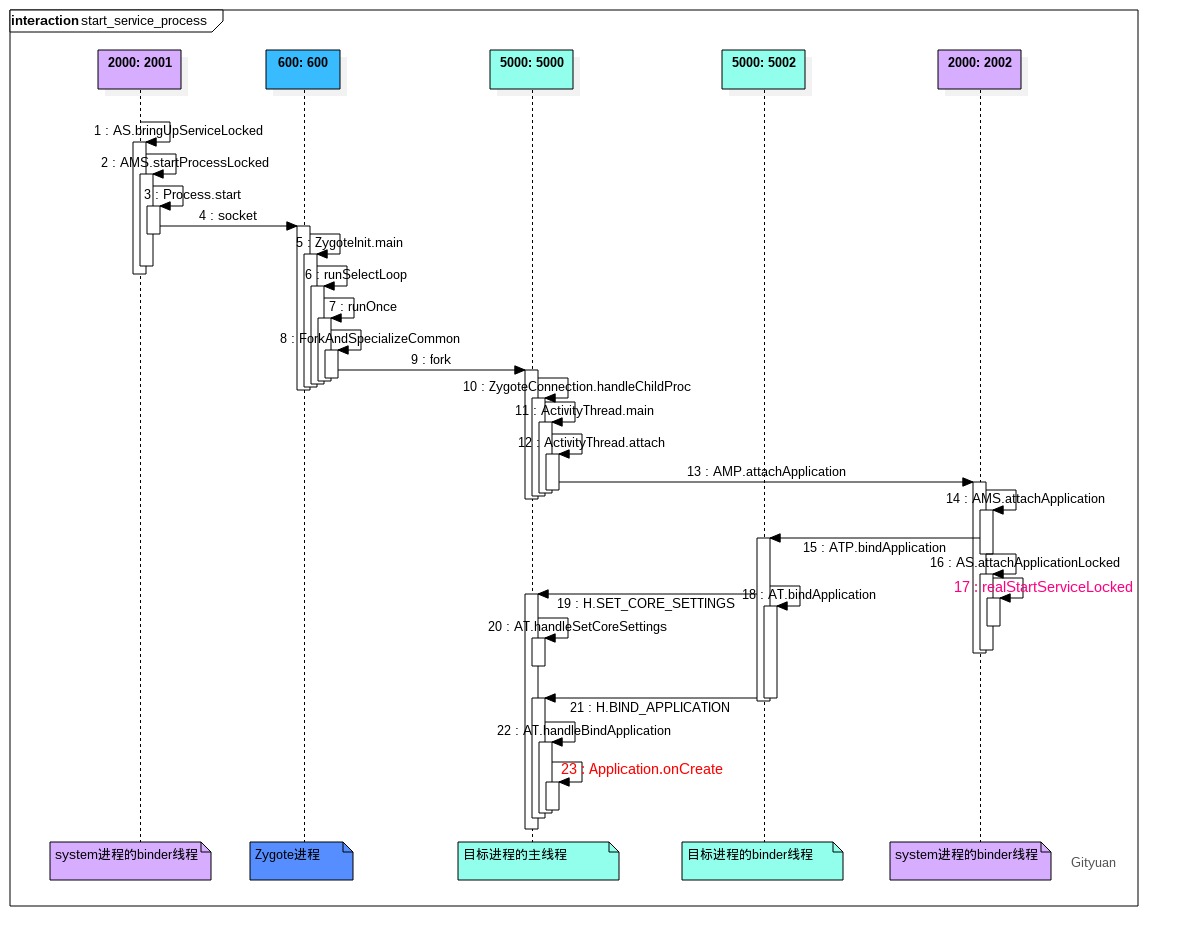
### 5.2 生命周期

startService的生命周期为onCreate, onStartCommand, onDestroy,流程如下图: [点击查看大图](http://www.gityuan.com/images/ams/service_lifeline.jpg)



由上图可见,造成ANR可能的原因有Binder full{step 7, 12}, MessageQueue(step 10), AMS Lock (step 13).

当进程启动Service其所在进程还没有启动时, 需要先启动其目标进程,流程如下图: [点击查看大图](http://www.gityuan.com/images/ams/start_service_process.jpg)



# bindService

sdf

## 一. 概述

文章[startService启动过程分析](http://gityuan.com/2016/03/06/start-service/)，介绍了 startService的过程，本文介绍另一种通过bind方式来启动服务。

### 1.1 实例

#### 定义AIDL文件：

interface IRemoteService {

String getBlog();

}

#### 服务端（远程服务进程)

public class RemoteService extends Service {

...

public IBinder onBind(Intent intent) {

return mBnRemoteService;

}

//IRemoteService.Stub 便是由AIDL文件IRemoteService自动生成的

private final IRemoteService.Stub mBnRemoteService = new IRemoteService.Stub() {

@Override

public String getBlog() throws RemoteException {

return ”www.gityuan.com;

}

};

}

#### Client端（发起端进程)

private IRemoteService mBpRemoteService;

private ServiceConnection mConnection = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

mBpRemoteService = IRemoteService.Stub.asInterface(service);

//通过Binder最终会调用远程服务中同名方法来执行，这便完成了一次跨进程

mBpRemoteService.getBlog();

}

@Override

public void onServiceDisconnected(ComponentName name) {

mRemoteService = null;

}

};

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

//Client端通过bindService去绑定远程服务【见下文】

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

文章[startService启动过程分析](http://gityuan.com/2016/03/06/start-service/)，介绍了 startService的过程，本文介绍另一种通过bind方式来启动服务。

## 发起端进程

**A**

### CW.bindService

[-> ContextWrapper.java]

public class ContextWrapper extends Context {

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

//其中mBase为ContextImpl对象 【见流程2】

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

}

}

### 2. CI.bindService

[-> ContextImpl.java]

class ContextImpl extends Context {

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

warnIfCallingFromSystemProcess();

//【见流程3】

return bindServiceCommon(service, conn, flags, Process.myUserHandle());

}

}

### 3. CI.bindServiceCommon

[-> ContextImpl.java]

private boolean bindServiceCommon(Intent service, ServiceConnection conn, int flags, UserHandle user) {

IServiceConnection sd;

...

if (mPackageInfo != null) {

//获取的是内部静态类InnerConnection【见小节3.1】

sd = mPackageInfo.getServiceDispatcher(conn, getOuterContext(),

mMainThread.getHandler(), flags);

} else {

...

}

try {

...

//[见流程4]

int res = ActivityManagerNative.getDefault().bindService(

mMainThread.getApplicationThread(), getActivityToken(), service,

service.resolveTypeIfNeeded(getContentResolver()),

sd, flags, getOpPackageName(), user.getIdentifier());

...

return res != 0;

} catch (RemoteException e) {

throw new RuntimeException("Failure from system", e);

}

}

该方法主要功能:

* 创建对象内部静态类LoadedApk.ServiceDispatcher.InnerConnection的对象;
* 通过IActivityManager接口,向AMS发送bind请求.

这里需要注意的是mMainThread.getApplicationThread()方法返回的是ApplicationThread对象, 该对象继承于ApplicationThreadNative(Binder服务端)

#### 3.1 getServiceDispatcher

[-> LoadedApk.java]

public final IServiceConnection getServiceDispatcher(ServiceConnection c, Context context, Handler handler, int flags) {

synchronized (mServices) {

LoadedApk.ServiceDispatcher sd = null;

ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher> map = mServices.get(context);

if (map != null) {

sd = map.get(c);

}

if (sd == null) {

//创建服务分发对象【见小节3.2】

sd = new ServiceDispatcher(c, context, handler, flags);

if (map == null) {

map = new ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>();

mServices.put(context, map);

}

//以ServiceConnection为key, ServiceDispatcher为value保存到map

map.put(c, sd);

} else {

sd.validate(context, handler);

}

//返回的内部类的对象InnerConnection【见小节3.2】

return sd.getIServiceConnection();

}

}

说明:

* mServices记录着所有context里面, 每个ServiceConnection以及所对应的LoadedApk.ServiceDispatcher对象;同一个ServiceConnection只会创建一次;
* 返回的对象是LoadedApk.ServiceDispatcher.InnerConnection，该对象继承于IServiceConnection.Stub, 该类是由IServiceConnection.aidl自动生成的 作为binder服务端。
* 这里需要注意的是IServiceConnection是属于oneway interface，也就是非阻塞的binder call.

#### ServiceDispatcher

[-> LoadedApk.java ::ServiceDispatcher]

static final class ServiceDispatcher {

//内部类

private final ServiceDispatcher.InnerConnection mIServiceConnection;

//用户传递的参数

private final ServiceConnection mConnection;

private final Context mContext;

private final Handler mActivityThread;

private final ServiceConnectionLeaked mLocation;

//用户传递的参数

private final int mFlags;

private boolean mDied;

private boolean mForgotten;

ServiceDispatcher(ServiceConnection conn,

Context context, Handler activityThread, int flags) {

//创建InnerConnection对象

mIServiceConnection = new InnerConnection(this);

//用户定义的ServiceConnection

mConnection = conn;

mContext = context;

mActivityThread = activityThread;

mLocation = new ServiceConnectionLeaked(null);

mLocation.fillInStackTrace();

mFlags = flags;

}

private static class InnerConnection extends IServiceConnection.Stub {

final WeakReference<LoadedApk.ServiceDispatcher> mDispatcher;

InnerConnection(LoadedApk.ServiceDispatcher sd) {

mDispatcher = new WeakReference<LoadedApk.ServiceDispatcher>(sd);

}

public void connected(ComponentName name, IBinder service) throws RemoteException {

LoadedApk.ServiceDispatcher sd = mDispatcher.get();

if (sd != null) {

sd.connected(name, service);

}

}

}

//获取内部类的InnerConnection对象

IServiceConnection getIServiceConnection() {

return mIServiceConnection;

}

}

ServiceDispatcher是LoadedApk的静态内部类。InnerConnection是ServiceDispatcher的静态内部类, 通过getIServiceConnection()方法返回的便是构造方法中创建的InnerConnection对象.

### 4. AMP.bindService

[-> ActivityManagerNative.java :: AMP]

public int bindService(IApplicationThread caller, IBinder token, Intent service, String resolvedType, IServiceConnection connection, int flags, String callingPackage, int userId) throws RemoteException {

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IActivityManager.descriptor);

data.writeStrongBinder(caller != null ? caller.asBinder() : null);

data.writeStrongBinder(token);

service.writeToParcel(data, 0);

data.writeString(resolvedType);

//将InnerConnection对象传递system\_server

data.writeStrongBinder(connection.asBinder());

data.writeInt(flags);

data.writeString(callingPackage);

data.writeInt(userId);

//通过bind调用，进入system\_server【见流程5】

mRemote.transact(BIND\_SERVICE\_TRANSACTION, data, reply, 0);

reply.readException();

int res = reply.readInt();

data.recycle();

reply.recycle();

return res;

}

经过Binder IPC便进入了system\_server进程.

## 三. system\_server端

### 5. AMN.onTransact

public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

switch (code) {

case BIND\_SERVICE\_TRANSACTION: {

data.enforceInterface(IActivityManager.descriptor);

IBinder b = data.readStrongBinder();

//此处b为ApplicationThread的, 转换后生成即ApplicationThreadProxy对象

IApplicationThread app = ApplicationThreadNative.asInterface(b);

IBinder token = data.readStrongBinder();

Intent service = Intent.CREATOR.createFromParcel(data);

String resolvedType = data.readString();

b = data.readStrongBinder();

int fl = data.readInt();

String callingPackage = data.readString();

int userId = data.readInt();

//生成InnerConnectiond的代理对象

IServiceConnection conn = IServiceConnection.Stub.asInterface(b);

//【见流程6】

int res = bindService(app, token, service, resolvedType, conn, fl,

callingPackage, userId);

reply.writeNoException();

reply.writeInt(res);

return true;

}

...

}

}

该方法的主要功能:

* 参数app: 根据发起端进程传递过来的ApplicationThread对象(Binder服务端), 通过asInterface()方法生成新的代理对象ApplicationThreadProxy类型对象app;
* 参数conn: 根据发起端进程传递过来的InnerConnectiond对象(Binder服务端),同样通过转换后,生成IServiceConnection.Stub.Proxy类型对象conn;
* 参数service: 数据类型为Intent, 是指本次要启动的service的意图;
* 参数callingPackage: 发起方所属的包名;
* 参数fl: 是指flags, 此时等于Context.BIND\_AUTO\_CREATE, 即值为1.

将这些参数传递给AMS来处理

### AMS.bindService

public int bindService(IApplicationThread caller, IBinder token, Intent service, String resolvedType, IServiceConnection connection, int flags, String callingPackage, int userId) throws TransactionTooLargeException {

...

synchronized(this) {

//【见流程7】

return mServices.bindServiceLocked(caller, token, service,

resolvedType, connection, flags, callingPackage, userId);

}

}

### 7. AS.bindServiceLocked

[-> ActiveServices.java]

int bindServiceLocked(IApplicationThread caller, IBinder token, Intent service, String resolvedType, IServiceConnection connection, int flags, String callingPackage, int userId) throws TransactionTooLargeException {

//查询发起端所对应的进程记录结构

final ProcessRecord callerApp = mAm.getRecordForAppLocked(caller);

...

ActivityRecord activity = null;

//token不为空, 代表着发起方具有activity上下文

if (token != null) {

activity = ActivityRecord.isInStackLocked(token);

if (activity == null) {

return 0; //存在token, 却找不到activity为空，则直接返回

}

}

int clientLabel = 0;

PendingIntent clientIntent = null;

if (callerApp.info.uid == Process.SYSTEM\_UID) {

... //发起方是system进程的情况

}

...

//根据发送端所在进程的SchedGroup来决定是否为前台service.

final boolean callerFg = callerApp.setSchedGroup != Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;

//根据用户传递进来Intent来检索相对应的服务【见流程7.1】

ServiceLookupResult res =

retrieveServiceLocked(service, resolvedType, callingPackage,

Binder.getCallingPid(), Binder.getCallingUid(), userId, true, callerFg);

if (res == null) {

return 0;

}

if (res.record == null) {

return -1;

}

//查询到相应的Service

ServiceRecord s = res.record;

final long origId = Binder.clearCallingIdentity();

try {

//取消服务的重启调度

unscheduleServiceRestartLocked(s, callerApp.info.uid, false);

if ((flags&Context.BIND\_AUTO\_CREATE) != 0) {

//更新当前service活动时间

s.lastActivity = SystemClock.uptimeMillis();

...

}

mAm.startAssociationLocked(callerApp.uid, callerApp.processName,

s.appInfo.uid, s.name, s.processName);

//【见流程7.2】

AppBindRecord b = s.retrieveAppBindingLocked(service, callerApp);

//创建对象ConnectionRecord,此处connection来自发起方

ConnectionRecord c = new ConnectionRecord(b, activity,

connection, flags, clientLabel, clientIntent);

IBinder binder = connection.asBinder();

ArrayList<ConnectionRecord> clist = s.connections.get(binder);

if (clist == null) {

clist = new ArrayList<ConnectionRecord>();

s.connections.put(binder, clist);

}

clist.add(c); // clist是ServiceRecord.connections的成员变量

b.connections.add(c); //b是指AppBindRecord

if (activity != null) {

if (activity.connections == null) {

activity.connections = new HashSet<ConnectionRecord>();

}

activity.connections.add(c);

}

b.client.connections.add(c);

if ((c.flags&Context.BIND\_ABOVE\_CLIENT) != 0) {

b.client.hasAboveClient = true;

}

if (s.app != null) {

updateServiceClientActivitiesLocked(s.app, c, true);

}

clist = mServiceConnections.get(binder);

if (clist == null) {

clist = new ArrayList<ConnectionRecord>();

mServiceConnections.put(binder, clist);

}

clist.add(c);

if ((flags&Context.BIND\_AUTO\_CREATE) != 0) {

s.lastActivity = SystemClock.uptimeMillis();

//启动service，这个过程跟startService过程一致【见小节8】

if (bringUpServiceLocked(s, service.getFlags(), callerFg, false) != null) {

return 0;

}

}

if (s.app != null) {

if ((flags&Context.BIND\_TREAT\_LIKE\_ACTIVITY) != 0) {

s.app.treatLikeActivity = true;

}

//更新service所在进程的优先级

mAm.updateLruProcessLocked(s.app, s.app.hasClientActivities

|| s.app.treatLikeActivity, b.client);

mAm.updateOomAdjLocked(s.app);

}

if (s.app != null && b.intent.received) {

try {

//Service已经正在运行，则调用InnerConnection的代理对象

c.conn.connected(s.name, b.intent.binder);

} catch (Exception e) {

...

}

//当第一个app连接到该binding, 且之前已被bind过, 则回调onRebind()方法

if (b.intent.apps.size() == 1 && b.intent.doRebind) {

requestServiceBindingLocked(s, b.intent, callerFg, true);

}

} else if (!b.intent.requested) {

//最终回调onBind()方法

requestServiceBindingLocked(s, b.intent, callerFg, false);

}

getServiceMap(s.userId).ensureNotStartingBackground(s);

} finally {

Binder.restoreCallingIdentity(origId);

}

return 1;

}

该方法主要功能:

* 通过retrieveServiceLocked(),根据用户传递进来Intent来检索相对应的服务
* 通过retrieveAppBindingLocked().创建AppBindRecord对象记录着当前ServiceRecord, intent以及发起方的进程信息。
* 通过bringUpServiceLocked()拉起目标服务;

另外, 将发起发传递过来的LoadedApk.ServiceDispatcher.InnerConnection的代理对象, 即IServiceConnection.Stub.Proxy类型对象connection, 保存到新创建的ConnectionRecord对象的成员变量. 再通过clist.add(c), 将该ConnectionRecord对象添加到clist队列. 后面便可以通过clist来 查询发起方的信息.

#### 7.1 AS.retrieveServiceLocked

[-> ActiveServices.java]

private ServiceLookupResult retrieveServiceLocked(Intent service, String resolvedType, String callingPackage, int callingPid, int callingUid, int userId, boolean createIfNeeded, boolean callingFromFg) {

ServiceRecord r = null;

userId = mAm.handleIncomingUser(callingPid, callingUid, userId,

false, ActivityManagerService.ALLOW\_NON\_FULL\_IN\_PROFILE, "service", null);

ServiceMap smap = getServiceMap(userId);

final ComponentName comp = service.getComponent();

if (comp != null) {

//根据服务名查找相应的ServiceRecord

r = smap.mServicesByName.get(comp);

}

if (r == null) {

Intent.FilterComparison filter = new Intent.FilterComparison(service);

//根据Intent查找相应的ServiceRecord

r = smap.mServicesByIntent.get(filter);

}

if (r == null) {

try {

//通过PKMS来查询相应的service

ResolveInfo rInfo =

AppGlobals.getPackageManager().resolveService(

service, resolvedType,

ActivityManagerService.STOCK\_PM\_FLAGS, userId);

ServiceInfo sInfo = rInfo != null ? rInfo.serviceInfo : null;

if (sInfo == null) {

return null;

}

//获取组件名

ComponentName name = new ComponentName(

sInfo.applicationInfo.packageName, sInfo.name);

if (userId > 0) {

//服务是否属于单例模式

if (mAm.isSingleton(sInfo.processName, sInfo.applicationInfo,

sInfo.name, sInfo.flags)

&& mAm.isValidSingletonCall(callingUid, sInfo.applicationInfo.uid)) {

userId = 0;

smap = getServiceMap(0);

}

sInfo = new ServiceInfo(sInfo);

sInfo.applicationInfo = mAm.getAppInfoForUser(sInfo.applicationInfo, userId);

}

r = smap.mServicesByName.get(name);

if (r == null && createIfNeeded) {

Intent.FilterComparison filter

= new Intent.FilterComparison(service.cloneFilter());

//创建Restarter对象

ServiceRestarter res = new ServiceRestarter();

...

//创建ServiceRecord对象

r = new ServiceRecord(mAm, ss, name, filter, sInfo, callingFromFg, res);

res.setService(r);

smap.mServicesByName.put(name, r);

smap.mServicesByIntent.put(filter, r);

//确保该组件不再位于pending队列

for (int i=mPendingServices.size()-1; i>=0; i--) {

ServiceRecord pr = mPendingServices.get(i);

if (pr.serviceInfo.applicationInfo.uid == sInfo.applicationInfo.uid

&& pr.name.equals(name)) {

mPendingServices.remove(i);

}

}

}

} catch (RemoteException ex) {

//运行在同一个进程，不会发生RemoteException

}

}

if (r != null) {

//各种权限检查，不满足条件则返回为null的service

if (mAm.checkComponentPermission(r.permission,

callingPid, callingUid, r.appInfo.uid, r.exported)

!= PackageManager.PERMISSION\_GRANTED) {

//当exported=false则不允许启动

if (!r.exported) {

return new ServiceLookupResult(null, "not exported from uid "

+ r.appInfo.uid);

}

return new ServiceLookupResult(null, r.permission);

} else if (r.permission != null && callingPackage != null) {

final int opCode = AppOpsManager.permissionToOpCode(r.permission);

if (opCode != AppOpsManager.OP\_NONE && mAm.mAppOpsService.noteOperation(

opCode, callingUid, callingPackage) != AppOpsManager.MODE\_ALLOWED) {

return null;

}

}

if (!mAm.mIntentFirewall.checkService(r.name, service, callingUid, callingPid,

resolvedType, r.appInfo)) {

return null;

}

//创建Service查询结果对象

return new ServiceLookupResult(r, null);

}

return null;

}

服务查询过程：

1. 根据服务名从ServiceMap.mServicesByName中查找相应的ServiceRecord，如果没有找到，则往下执行；
2. 根据Intent从ServiceMap.mServicesByIntent中查找相应的ServiceRecord，如果还是没有找到，则往下执行；
3. 通过PKMS来查询相应的ServiceInfo，如果仍然没有找到，则不再往下执行。

属于isSingleton的情况有以下3类：

1. 组件uid>10000，且同时具有ServiceInfo.FLAG\_SINGLE\_USER flags和INTERACT\_ACROSS\_USERS权限；
2. 组件运行在system进程的情况；
3. 具有ServiceInfo.FLAG\_SINGLE\_USER flags，且uid=Process.PHONE\_UID或者persistent app的情况；

#### 7.2 SR.retrieveAppBindingLocked

[-> ServiceRecord.java]

***public AppBindRecord retrieveAppBindingLocked(Intent intent, ProcessRecord app) {***

***Intent.FilterComparison filter = new Intent.FilterComparison(intent);***

***IntentBindRecord i = bindings.get(filter);***

***if (i == null) {***

***//创建连接ServiceRecord和filter的记录信息***

***i = new IntentBindRecord(this, filter);***

***bindings.put(filter, i);***

***}***

***//此处app是指调用方所在进程***

***AppBindRecord a = i.apps.get(app);***

***if (a != null) {***

***return a;***

***}***

***//创建ServiceRecord跟进程绑定的记录信息***

***a = new AppBindRecord(this, i, app);***

***i.apps.put(app, a);***

***return a;***

***}***

AppBindRecord对象记录着当前ServiceRecord,intent以及发起方的进程信息。

### 8. bringUpServiceLocked

[-> ActiveServices.java]

private final String bringUpServiceLocked(ServiceRecord r, int intentFlags, boolean execInFg, boolean whileRestarting) throws TransactionTooLargeException {

// 进程已存在的情况

if (r.app != null && r.app.thread != null) {

//调用service.onStartCommand()过程

sendServiceArgsLocked(r, execInFg, false);

return null;

}

....

//服务正在启动，设置package停止状态为false

AppGlobals.getPackageManager().setPackageStoppedState(

r.packageName, false, r.userId);

ProcessRecord app;

if (!isolated) {

app = mAm.getProcessRecordLocked(procName, r.appInfo.uid, false);

if (app != null && app.thread != null) {

app.addPackage(r.appInfo.packageName, r.appInfo.versionCode, mAm.mProcessStats);

// 启动服务 【见流程9】

realStartServiceLocked(r, app, execInFg);

return null;

}

//对于进程没有启动的情况

if (app == null) {

//启动service所要运行的进程,最终还是会调用到【见流程9】

if ((app=mAm.startProcessLocked(procName, r.appInfo, true, intentFlags,

"service", r.name, false, isolated, false)) == null) {

...

return msg;

}

}

if (!mPendingServices.contains(r)) {

mPendingServices.add(r);

}

...

return null;

}

### 9. realStartServiceLocked

[-> ActiveServices.java]

private final void realStartServiceLocked(ServiceRecord r, ProcessRecord app, boolean execInFg) throws RemoteException {

...

r.app = app;

r.restartTime = r.lastActivity = SystemClock.uptimeMillis();

final boolean newService = app.services.add(r);

//发送delay消息

bumpServiceExecutingLocked(r, execInFg, "create");

boolean created = false;

try {

...

mAm.ensurePackageDexOpt(r.serviceInfo.packageName);

app.forceProcessStateUpTo(ActivityManager.PROCESS\_STATE\_SERVICE);

//服务进入 onCreate() 【见流程10】

app.thread.scheduleCreateService(r, r.serviceInfo,

mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),

app.repProcState);

r.postNotification();

created = true;

} catch (DeadObjectException e) {

mAm.appDiedLocked(app); //应用死亡处理

throw e;

} finally {

if (!created) {

final boolean inDestroying = mDestroyingServices.contains(r);

serviceDoneExecutingLocked(r, inDestroying, inDestroying);

if (newService) {

app.services.remove(r);

r.app = null;

}

//尝试重新启动服务

if (!inDestroying) {

scheduleServiceRestartLocked(r, false);

}

}

}

//【见流程12】

requestServiceBindingsLocked(r, execInFg);

updateServiceClientActivitiesLocked(app, null, true);

if (r.startRequested && r.callStart && r.pendingStarts.size() == 0) {

r.pendingStarts.add(new ServiceRecord.StartItem(r, false, r.makeNextStartId(),

null, null));

}

sendServiceArgsLocked(r, execInFg, true);

if (r.delayed) {

getServiceMap(r.userId).mDelayedStartList.remove(r);

r.delayed = false;

}

...

}

该方法有几个重要的时间点：

* bumpServiceExecutingLocked；
* AT.scheduleCreateService;
* requestServiceBindingsLocked;
* AT.sendServiceArgsLocked;

## 远程服务进程

scheduleCreateService->MyService.onCreate()同startService

在realStartServiceLocked中请求binding

### requestServiceBindingLocked

d

ActiveServices.java]

private final boolean requestServiceBindingLocked(ServiceRecord r, IntentBindRecord i, boolean execInFg, boolean rebind) throws TransactionTooLargeException {

if (r.app == null || r.app.thread == null) {

return false;

}

if ((!i.requested || rebind) && i.apps.size() > 0) {

try {

//发送bind开始的消息

bumpServiceExecutingLocked(r, execInFg, "bind");

r.app.forceProcessStateUpTo(ActivityManager.PROCESS\_STATE\_SERVICE);

//服务进入 onBind() 【见流程14】

r.app.thread.scheduleBindService(r, i.intent.getIntent(), rebind, r.app.repProcState);

if (!rebind) {

i.requested = true;

}

i.hasBound = true;

i.doRebind = false;

} catch (TransactionTooLargeException e) {

final boolean inDestroying = mDestroyingServices.contains(r);

serviceDoneExecutingLocked(r, inDestroying, inDestroying);

throw e;

} catch (RemoteException e) {

final boolean inDestroying = mDestroyingServices.contains(r);

serviceDoneExecutingLocked(r, inDestroying, inDestroying);

return false;

}

}

return true;

}

### 14. ATP.scheduleBindService

[-> ApplicationThreadProxy.java]

public final void scheduleBindService(IBinder token, Intent intent, boolean rebind, int processState) throws RemoteException {

Parcel data = Parcel.obtain();

data.writeInterfaceToken(IApplicationThread.descriptor);

data.writeStrongBinder(token);

intent.writeToParcel(data, 0);

data.writeInt(rebind ? 1 : 0);

data.writeInt(processState);

//【见流程15】

mRemote.transact(SCHEDULE\_BIND\_SERVICE\_TRANSACTION, data, null,

IBinder.FLAG\_ONEWAY);

data.recycle();

}

## 五. 远程服务进程

### 15. ATN.onTransact

[-> ApplicationThreadNative.java]

public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

switch (code) {

case SCHEDULE\_BIND\_SERVICE\_TRANSACTION: {

data.enforceInterface(IApplicationThread.descriptor);

IBinder token = data.readStrongBinder();

Intent intent = Intent.CREATOR.createFromParcel(data);

boolean rebind = data.readInt() != 0;

int processState = data.readInt();

//【见流程13】

scheduleBindService(token, intent, rebind, processState);

return true;

}

...

}

### 13. scheduleBindService

[-> ApplicationThread.java]

public final void scheduleBindService(IBinder token, Intent intent, boolean rebind, int processState) {

updateProcessState(processState, false);

BindServiceData s = new BindServiceData();

s.token = token;

s.intent = intent;

s.rebind = rebind;

//【见流程14】

sendMessage(H.BIND\_SERVICE, s);

}

通过handler机制, 将H.BIND\_SERVICE消息发送给远程服务进程的主线程的handler来处理

### 14. AT.handleBindService

[-> ActivityThread.java]

***private void handleBindService(BindServiceData data) {***

***Service s = mServices.get(data.token);***

***if (s != null) {***

***try {***

***data.intent.setExtrasClassLoader(s.getClassLoader());***

***data.intent.prepareToEnterProcess();***

***if (!data.rebind) {***

***// 执行Service.onBind()回调方法 [见小节14.1]***

***IBinder binder = s.onBind(data.intent);***

***//将onBind返回值传递回去【见流程15】***

***ActivityManagerNative.getDefault().publishService(***

***data.token, data.intent, binder);***

***} else {***

***s.onRebind(data.intent);***

***ActivityManagerNative.getDefault().serviceDoneExecuting(***

***data.token, SERVICE\_DONE\_EXECUTING\_ANON, 0, 0);***

***}***

***ensureJitEnabled();***

***} catch (Exception e) {***

***...***

***}***

***}***

***}***

#### 14.1 onRebind

[-> Service.java]

public abstract IBinder onBind(Intent intent);

Service的onBind()是抽象方法, 所以大家创建Service子类时必须要覆写该方法, 返回IBinder对象, 也可以直接返回NULL.

### 15. AMP.publishService

[-> ActivityManagerNative.java ::ActivityManagerProxy]

public void publishService(IBinder token, Intent intent, IBinder service) throws RemoteException {

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IActivityManager.descriptor);

data.writeStrongBinder(token);

intent.writeToParcel(data, 0);

//将service.onBind的返回值传递给远程进程

data.writeStrongBinder(service);

// [见流程16]

mRemote.transact(PUBLISH\_SERVICE\_TRANSACTION, data, reply, 0);

reply.readException();

data.recycle();

reply.recycle();

}

经过Binder IPC进入system\_server进程交由AMS来处理

## 六. system\_server进程

### 16. AMS.publishService

public void publishService(IBinder token, Intent intent, IBinder service) {

...

synchronized(this) {

if (!(token instanceof ServiceRecord)) {

throw new IllegalArgumentException("Invalid service token");

}

//【见流程17】

mServices.publishServiceLocked((ServiceRecord)token, intent, service);

}

}

远程服务的onBind()的返回值的IBinder(Bn端), 在AMP.publishService()过程中经过data.writeStrongBinder(service)传递到底层, 再回到system\_server进程中AMN.onTransact()中经过data.readStrongBinder()方法会获取该service所相对应的代理对象(Bp端).

简言之,此处的IBinder类型的service就是远程服务进程中的Bp端对象.

### 17. publishServiceLocked

[-> ActiveServices.java]

***void publishServiceLocked(ServiceRecord r, Intent intent, IBinder service) {***

***final long origId = Binder.clearCallingIdentity();***

***try {***

***if (r != null) {***

***Intent.FilterComparison filter = new Intent.FilterComparison(intent);***

***IntentBindRecord b = r.bindings.get(filter);***

***if (b != null && !b.received) {***

***b.binder = service;***

***b.requested = true;***

***b.received = true;***

***for (int conni=r.connections.size()-1; conni>=0; conni--) {***

***ArrayList<ConnectionRecord> clist = r.connections.valueAt(conni);***

***for (int i=0; i<clist.size(); i++) {***

***ConnectionRecord c = clist.get(i);***

***if (!filter.equals(c.binding.intent.intent)) {***

***continue;***

***}***

***try {***

***//【见流程18】***

***c.conn.connected(r.name, service);***

***} catch (Exception e) {***

***...***

***}***

***}***

***}***

***}***

***serviceDoneExecutingLocked(r, mDestroyingServices.contains(r), false);***

***}***

***} finally {***

***Binder.restoreCallingIdentity(origId);***

***}***

***}***

[小节7]AS.bindServiceLocked的过程中初始化, 可知c.conn是指通往发起端进程的IServiceConnection.Stub.Proxy代理对象. 通过Binder IPC调用, 进入发起方进程的IServiceConnection.Stub对象. 由于LoadedApk.ServiceDispatcher.InnerConnection 继承于IServiceConnection.Stub. 所以,接下来便由回到发起方进程中的InnerConnection对象.

## 七. 发起方进程

### 18. InnerConnection.connected

[-> LoadedApk.ServiceDispatcher.InnerConnection]

private static class InnerConnection extends IServiceConnection.Stub {

final WeakReference<LoadedApk.ServiceDispatcher> mDispatcher;

InnerConnection(LoadedApk.ServiceDispatcher sd) {

mDispatcher = new WeakReference<LoadedApk.ServiceDispatcher>(sd);

}

public void connected(ComponentName name, IBinder service) throws RemoteException {

LoadedApk.ServiceDispatcher sd = mDispatcher.get();

if (sd != null) {

sd.connected(name, service); //[见流程19]

}

}

}

### 19. ServiceDispatcher.connected

[-> LoadedApk.ServiceDispatcher]

public void connected(ComponentName name, IBinder service) {

if (mActivityThread != null) {

//这是主线程的Handler 【见流程20】

mActivityThread.post(new RunConnection(name, service, 0));

} else {

doConnected(name, service);

}

}

### 20. RunConnection.run

[-> LoadedApk.ServiceDispatcher.RunConnection]

private final class RunConnection implements Runnable {

RunConnection(ComponentName name, IBinder service, int command) {

mName = name;

mService = service;

mCommand = command; //此时为0

}

public void run() {

if (mCommand == 0) {

doConnected(mName, mService); //【见流程21】

} else if (mCommand == 1) {

doDeath(mName, mService);

}

}

final ComponentName mName;

final IBinder mService;

final int mCommand;

}

* 此处的mName是指远程服务的组件名对象ComponentName;
* 此处的mService是指远程服务的onBind()返回的IBinder代理对象;

### 21. doConnected

[-> LoadedApk.ServiceDispatcher]

public void doConnected(ComponentName name, IBinder service) {

ServiceDispatcher.ConnectionInfo old;

ServiceDispatcher.ConnectionInfo info;

synchronized (this) {

if (mForgotten) {

return;

}

old = mActiveConnections.get(name);

if (old != null && old.binder == service) {

return;

}

if (service != null) {

mDied = false;

info = new ConnectionInfo();

info.binder = service;

//创建死亡监听对象

info.deathMonitor = new DeathMonitor(name, service);

try {

//建立死亡通知

service.linkToDeath(info.deathMonitor, 0);

mActiveConnections.put(name, info);

} catch (RemoteException e) {

mActiveConnections.remove(name);

return;

}

} else {

mActiveConnections.remove(name);

}

if (old != null) {

old.binder.unlinkToDeath(old.deathMonitor, 0);

}

}

if (old != null) {

mConnection.onServiceDisconnected(name);

}

if (service != null) {

//回调用户定义的ServiceConnection()

mConnection.onServiceConnected(name, service);

}

}

此处创建了死亡监听对象,也是内部类:LoadedApk.ServiceDispatcher.DeathMonitor,定义如下:

private final class DeathMonitor implements IBinder.DeathRecipient {

DeathMonitor(ComponentName name, IBinder service) {

mName = name;

mService = service;

}

public void binderDied() {

death(mName, mService); //【见流程18.2】

}

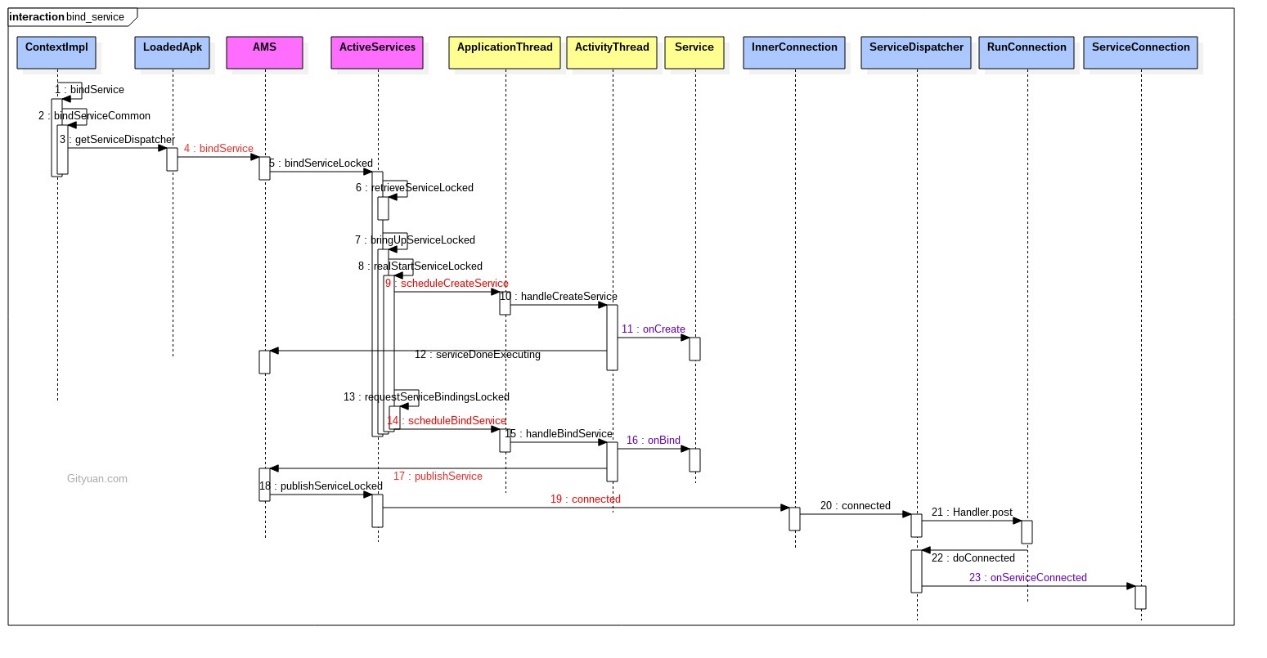
final ComponentName mName;

final IBinder mService;

}

## 总结

整体调用流程图：[大图](http://www.gityuan.com/images/ams/bind_service.jpg)



说明：

1. 图中蓝色代表的是Client进程(发起端), 红色代表的是system\_server进程, 黄色代表的是target进程(service所在进程);
2. Client进程: 通过getServiceDispatcher获取Client进程的匿名Binder服务端，即LoadedApk.ServiceDispatcher.InnerConnection,该对象继承于IServiceConnection.Stub； 再通过bindService调用到system\_server进程;
3. system\_server进程: 依次通过scheduleCreateService和scheduleBindService方法, 远程调用到target进程; 4: target进程: 依次执行onCreate()和onBind()方法; 将onBind()方法的返回值IBinder(作为target进程的binder服务端)通过publishService传递到system\_server进程;
4. system\_server进程: 利用IServiceConnection代理对象向Client进程发起connected()调用, 并把target进程的onBind返回Binder对象的代理端传递到Client进程;
5. Client进程: 回调到onServiceConnection()方法, 该方法的第二个参数便是target进程的binder代理端. 到此便成功地拿到了target进程的代理, 可以畅通无阻地进行交互.

## 案例分析

蓝牙绑定服务

# unbindService

***frameworks/base/core/java/android/app/ContextImpl.java***

***/frameworks/base/core/java/android/app/LoadedApk.java***

***/frameworks/base/core/java/android/app/IServiceConnection.aidl(自动生成Binder两端)***

## 一. unbind

文章[bindService启动过程分析](http://gityuan.com/2016/05/01/bind-service/)，介绍了 bindService的过程，本文介绍其对应的另一个操作unbind。

**unbind调用链:**

AMP.unbindService

AMS.unbindService

AS.unbindServiceLocked

AS.removeConnectionLocked

ATP.scheduleUnbindService

AT.scheduleUnbindService

AT.handleUnbindService

Service.onUnbind

AS.bringDownServiceIfNeededLocked

AS.bringDownServiceLocked

ATP.scheduleUnbindService

AT.scheduleUnbindService

ATP.scheduleStopService

AT.scheduleStopService

### 1.1 AMP.unbindService

… //省略，未完待续

## 二. onServiceDisconnected

当service所在进程死亡后，binderDied死亡回调后触发的。

### 2.1 binderDied

[-> LoadedApk.ServiceDispatcher.DeathMonitor]

private final class DeathMonitor implements IBinder.DeathRecipient {

DeathMonitor(ComponentName name, IBinder service) {

mName = name;

mService = service;

}

public void binderDied() {

death(mName, mService); //【见流程2.2】

}

final ComponentName mName;

final IBinder mService;

}

### 2.2 death

[-> LoadedApk.ServiceDispatcher]

public void death(ComponentName name, IBinder service) {

ServiceDispatcher.ConnectionInfo old;

synchronized (this) {

mDied = true;

old = mActiveConnections.remove(name);

if (old == null || old.binder != service) {

return;

}

old.binder.unlinkToDeath(old.deathMonitor, 0);

}

if (mActivityThread != null) {

//【见流程2.3】

mActivityThread.post(new RunConnection(name, service, 1));

} else {

doDeath(name, service);

}

}

### 2.3 run

[-> LoadedApk.ServiceDispatcher.RunConnection]

private final class RunConnection implements Runnable {

RunConnection(ComponentName name, IBinder service, int command) {

mName = name;

mService = service;

mCommand = command;

}

public void run() {

if (mCommand == 0) {

doConnected(mName, mService);

} else if (mCommand == 1) {

doDeath(mName, mService); //【见流程2.4】

}

}

}

### 2.4 doDeath

[-> LoadedApk.ServiceDispatcher]

public void doDeath(ComponentName name, IBinder service) {

//回调用户定义的onServiceDisconnected方法

mConnection.onServiceDisconnected(name);

}

## 三. 总结

1. unbind()是bind的逆操作，主要是清理bind相关对象，并不会回调onServiceDisconnected.
2. 当Service进程死亡，经过Binder死亡回调，则会进入Client端进程来执行binderDied()，经过层层调用， 最终回调用户定义的onServiceDisconnected方法。
3. 当或者stopService过程被service彻底destroy的过程，也会回调onServiceDisconnected方法。

# 参考

四大组件之ServiceRecord

http://gityuan.com/2017/05/25/service\_record/

startService启动过程分析

<http://gityuan.com/2016/03/06/start-service/>

bindService启动过程分析

<http://gityuan.com/2016/05/01/bind-service/>

unbindService流程分析

http://gityuan.com/2016/05/02/unbind-service/