# TODO

本地服务和远程服务的差别

<http://gityuan.com/2017/05/25/service_record/>

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

# startService的原理分析

<http://blog.csdn.net/luoshengyang/article/details/6677029>

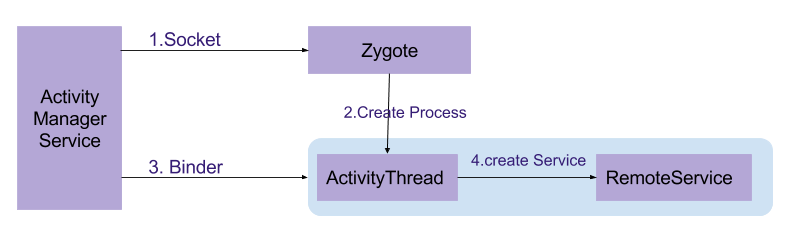
# startService的原理分析

http://blog.csdn.net/luoshengyang/article/details/6677029

在app中启动一个service，就一行语句搞定，

startService()； //或 binderService()

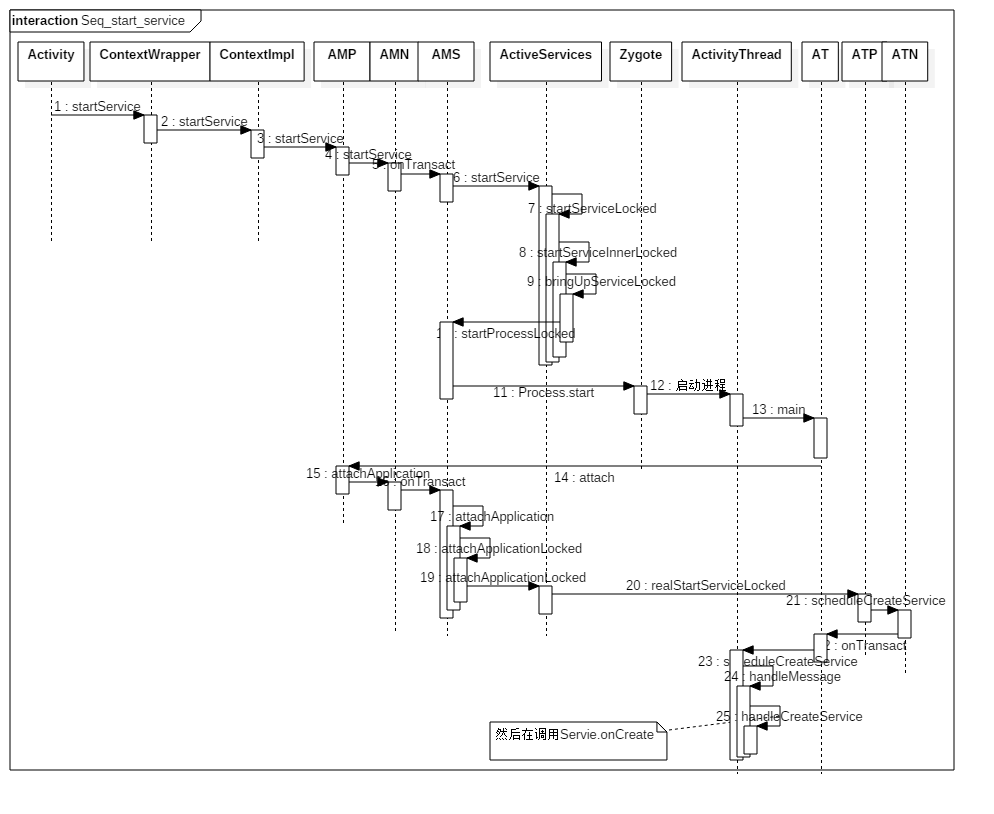
该过程如下：



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);

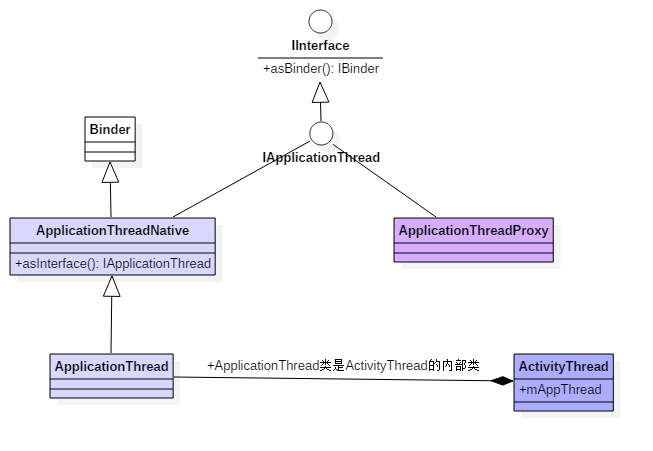
return true;

}

}

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]

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]

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， 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]

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]

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]

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：

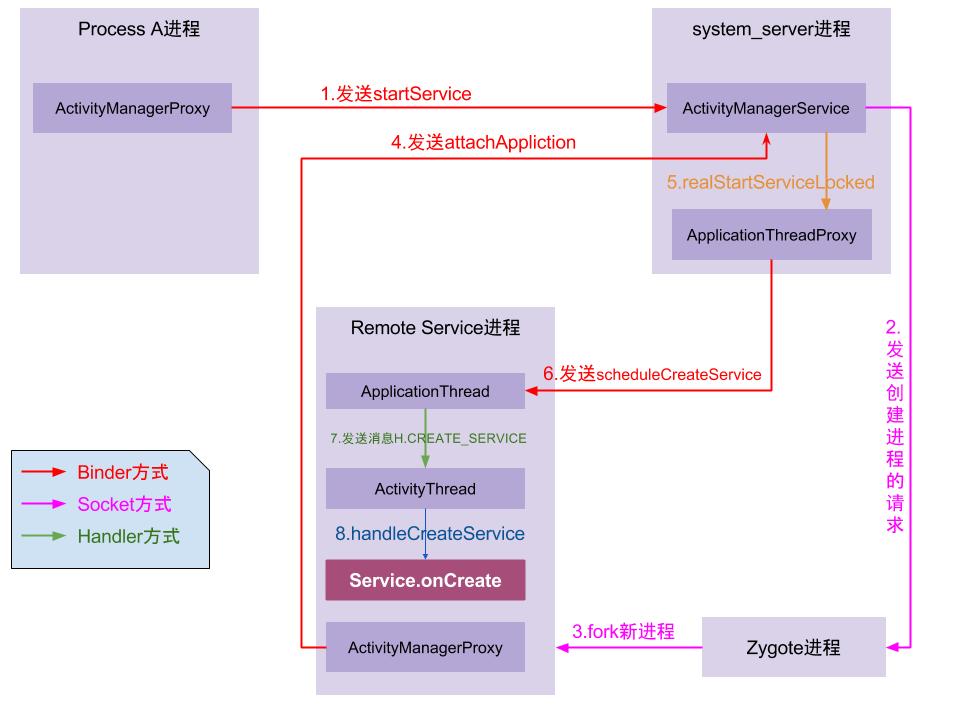
* 执行scheduleCreateService()方法，层层调用最终回调Service.onCreate(); [见流程11~16]
* 执行scheduleServiceArgs()方法，层层调用最终回调Service.onStartCommand(); [见流程17]，这两个过程类似，此处省略。

## 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。

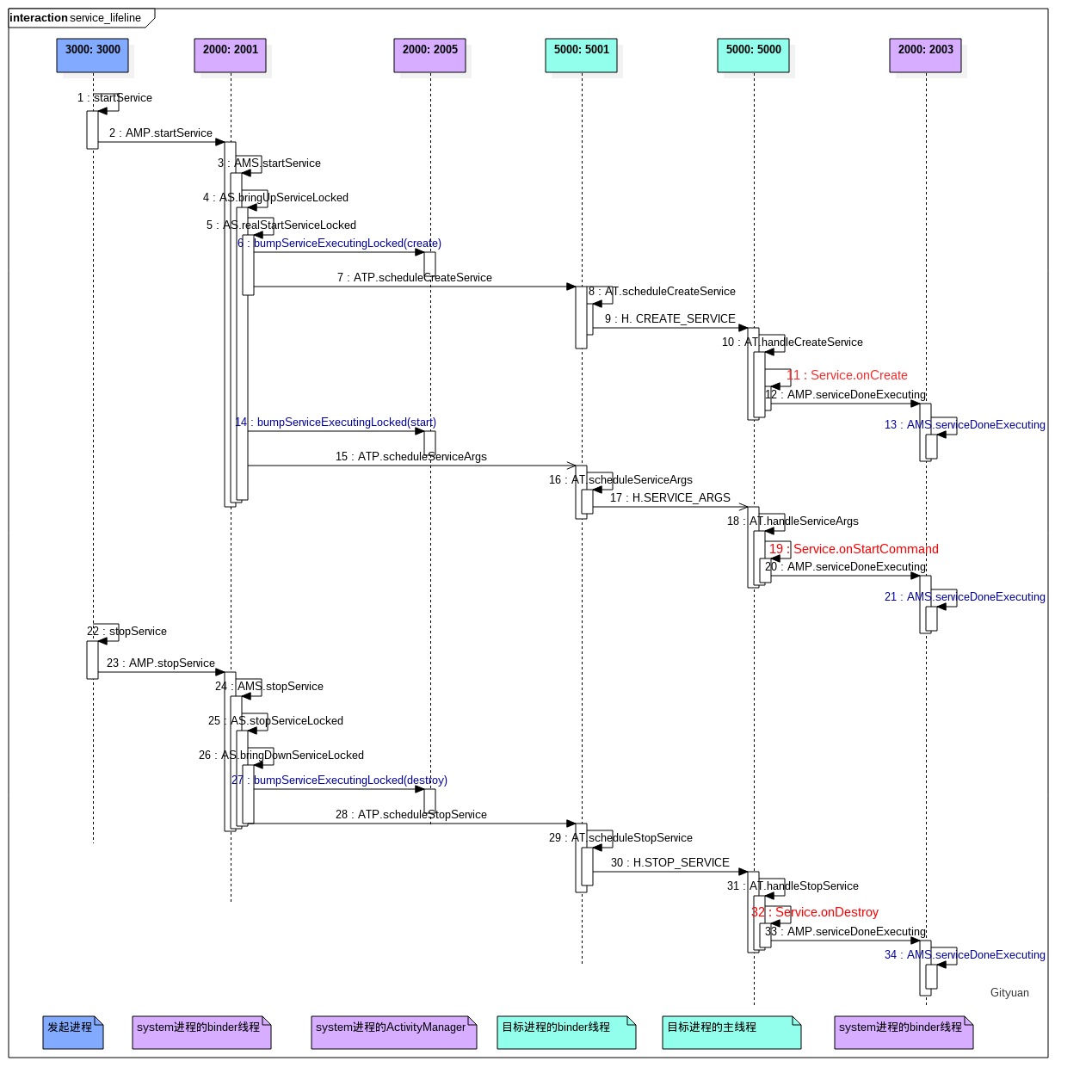
启动流程：

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。

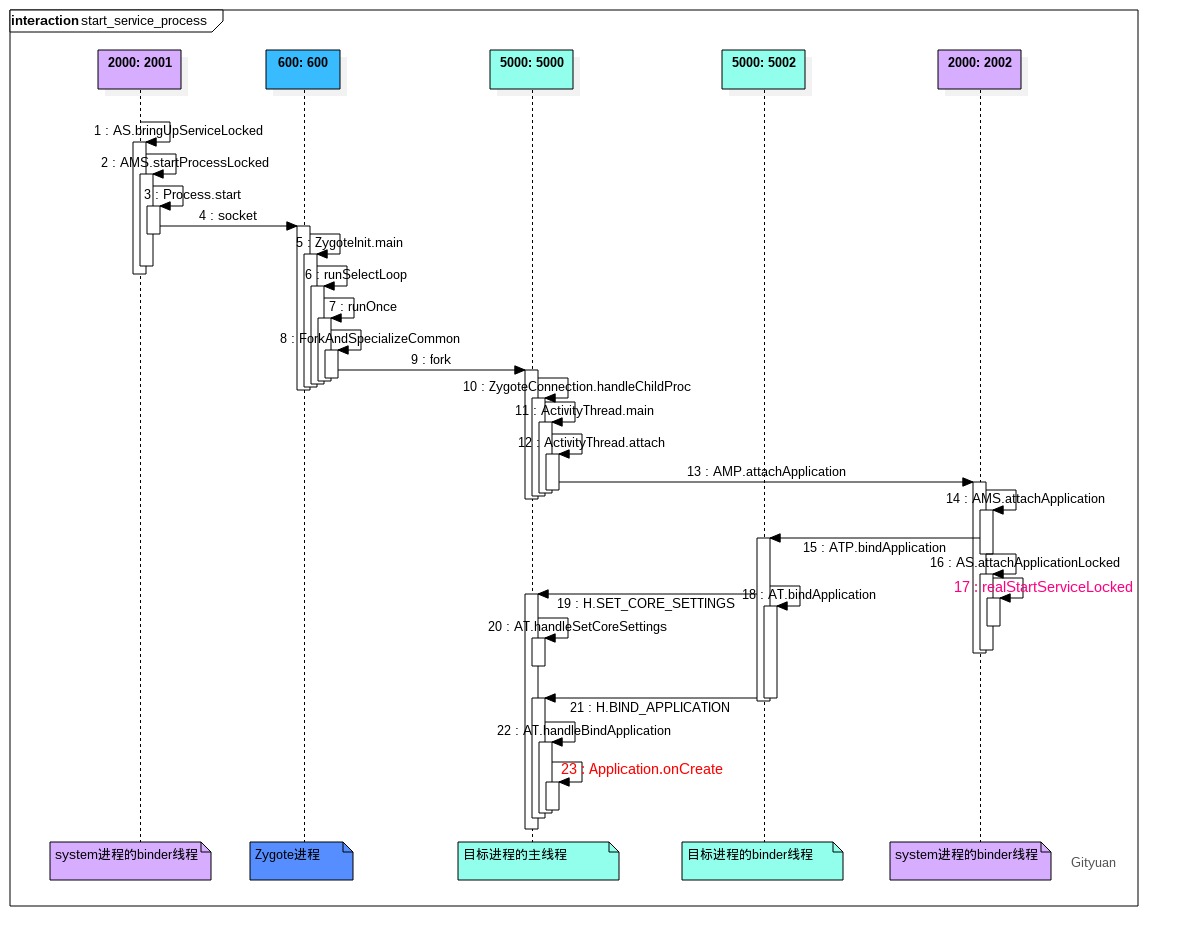
### 5.2 生命周期

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)



**欢迎关注微信公**